Deep Learning Classification of Spinal Osteoporotic Compression Fractures on Radiographs

Qifei Dong

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2024

Reading Committee:

Gang Luo, Chair

Nathan M. Cross

Jeffrey Jarvik

Program Authorized to Offer Degree:

Biomedical Informatics and Medical Education

University of Washington

Abstract

Deep Learning Classification of Spinal Osteoporotic Compression Fractures on Radiographs

Qifei Dong

Chair of the Supervisory Committee:
Gang Luo
Biomedical Informatics and Medical Education

Although osteoporosis is a debilitating disease that affects 9% of individuals over 50 years of age in the US and 200 million women globally, osteoporosis screening is underutilized. A complementary approach to osteoporosis screening is opportunistic screening using pre-existing images to detect spinal osteoporotic compression fractures (OCFs). Spinal OCFs are often incidental findings and under-reported. An automated opportunistic screening tool can ensure earlier diagnosis and treatment of spinal OCFs and osteoporosis. A crucial component for the automated opportunistic screening tool is an OCF classifier that detects OCF on each vertebral body. In this research, we focus on building this OCF classifier. To do this, two spine radiograph datasets were obtained, whose radiographs are in the Digital Imaging and Communications in Medicine (DICOM) format. To annotate the data, we designed DicomAnnotator, a configurable

open-source software program for efficient DICOM image annotation. With the annotated radiographs, we used five deep learning algorithms to build the OCF classifier. Training a deep learning model on a large dataset is often time-consuming. During deep learning model training, it is desirable to offer a non-trivial progress indicator that can continuously project the remaining model training time and the fraction of model training work completed. This makes the deep learning model training process more user-friendly. We designed the first set of techniques to support progress indication for deep learning model training that allows early stopping. In summary, we realized the following three aims in this research:

1) **Aim 1:** Design DicomAnnotator. Usability evaluation shows that DicomAnnotator is easy to learn, is efficient to use, and allows annotators to quickly make several types of annotations on a large set of DICOM images.
2) **Aim 2:** Build the OCF classifier. Model evaluation results show that our OCF classifier has some generalizability to clinical data and a suitable performance for our future opportunistic osteoporosis screening.
3) **Aim 3:** Design progress indication methods for deep learning model training. Our experiments show that our progress indicator can offer useful information even if the run-time system load varies over time and can self-correct its initial estimation errors, if any, over time.

TABLE OF CONTENTS

List of Figures v

List of Tables ix

List of Abbreviations xi

List of Symbols xiii

Chapter 1. Introduction 1

1.1 Background 1

1.2 Contributions 5

1.3 Dissertation Overview 6

Chapter 2. DicomAnnotator: A Configurable Open-Source Software Program for Efficient DICOM Image Annotation 7

2.1 Introduction 8

2.2 Functional Modules 11

2.2.1 Configuration File 12

2.2.2 Login Page 14

2.2.3 Image Viewing Module 15

2.2.4 Annotation Module 17

2.2.5 Result File 18

2.2.6 An Example of Modifying the Configuration File 19

2.3 Ancillary Features 20

2.3.1 Operation Modes 21
2.3.2 Commenting System 22
2.3.3 Automatic Window and Level Adjustment 23
2.3.4 Splitting an Image Set and Merging Result Files 24
2.4 Usability Evaluation 25
2.4.1 Users' Backgrounds 25
2.4.2 Process of Usability Evaluation 26
2.4.3 Results of Usability Evaluation 28
2.5 Discussion 30
2.6 Conclusion 32
Chapter 3. Deep Learning Classification of Spinal OCFs on Radiographs 33
3.1 Introduction 34
3.2 Datasets 35
3.2.1 UW dataset 36
3.2.2 MrOS dataset 41
3.2.3 Summary 45
3.3 Data Pre-processing and Augmentation 46
3.4 Model Training 51
3.4.1 Overview of Model Training 51
3.4.2 Details of Model Training 53
3.5 Model Evaluation 58
3.5.1 Description of Model Evaluation 58
3.5.2 Results of Model Evaluation 59

3.5.3 Comparison between the Models 62
3.6 Discussion 63
3.7 Conclusion 68
Chapter 4. Progress Indication for Deep Learning Model Training 69
4.1 Introduction 70
4.2 Related Work 73
4.3 Some Concepts and Notations 75
4.4 Basic Progress Indication Method 76
4.4.1 Innovation of the Basic Progress Indication Method 77
4.4.2 Overview of the Basic Progress Indication Method 78
4.4.3 Estimating the Trend Curve 83
4.4.4 Estimating the Random Noise's Variance 84
4.4.5 Projecting the Number of Validation Points Needed for Model Training 84
4.5 Improved Progress Indication Method 86
4.5.1 Innovation of the Improved Progress Indication Method 87
4.5.2 Overview of the Improved Progress Indication Method 90
4.5.3 Our Approach to Insert Extra Validation Points between the Original Validation Points 92
4.5.4 Setting V' 106
4.5.5 Relationship between the Random Noise's Variance and the Size of the Actual Validation Set Used at the Validation Point 107
4.5.6 Estimating the Trend Curve and the Variance of the Random Noise for Future Validation Points 110

4.5.7 Determining V_{min} 126

4.5.8 Estimating the Model Training Cost Based upon the Projected Number of Original Validation Points Needed for Model Training 128

4.6 Performance 129

4.6.1 Description of the Experiments 129

4.6.2 Accuracy Measure 131

4.6.3 Comparison of Three Progress Indication Methods for Deep Learning Model Training 132

4.6.4 Test Results for Adopting a Constant Learning Rate 135

4.6.5 Test Results for Applying an Exponential Decay Method to the Learning Rate 139

4.6.6 Test Results for Applying a Step Decay Method to the Learning Rate to Train GoogLeNet 143

4.6.7 Summary of the Performance Test Results 145

4.7 Discussion 145

4.8 Conclusion 146

Chapter 5. Summary 148

Bibliography 150

LIST OF FIGURES

Figure 1.1. Our future automated opportunistic screening tool detecting OCFs on radiographs. 2
Figure 1.2. A progress indicator for deep learning model training. 4
Figure 2.1. Our approach to annotating multiple regions of interest in an image. 9
Figure 2.2. The steps to using DicomAnnotator to annotate an image dataset. 12
Figure 2.3. The login page. 14
Figure 2.4. The main page of DicomAnnotator that is divided into 12 panels. 16
Figure 2.5. Illustration of user's interaction with the image in the annotation process. ... 17
Figure 2.6. The final annotations of an example spine image. 18
Figure 2.7. The configuration file for the spine CT image annotation task in JSON format. 20
Figure 2.8. DicomAnnotator demonstrating display and annotations of an example sagittal lumbar spine CT image. 20
Figure 2.9. The main page in the "View Only" mode. 22
Figure 2.10. The confirmation dialog that is displayed when switching from the "View Only" mode to the "Edit" mode. 22
Figure 2.11. An example of comments displayed in the comment panel for an image. ... 23
Figure 2.12. The usability evaluation process. 26
Figure 3.1. Construction of the UW dataset and partitioning it into the training, validation, and test sets. 39
Figure 3.2. The MrOS dataset was divided into the test, validation, and training sets by subject. 44
Figure 3.3. This process of generating a vertebral patch was performed for each vertebral body labeled in a radiograph using the four corner points indicated by red stars. The blue and purple arrows demonstrate the creation of the vertebral patches without and with the augmentation steps, respectively. 48
Figure 3.4. The flowchart of OCF classification using deep learning. 52

Figure 3.5. The performance of the model, which was built using the ensemble averaging algorithm in Task 2 and evaluated on the test set of the UW-m2ABQ dataset. 60
Figure 3.6. The performance of the model, which was built using the ensemble averaging algorithm in Task 2 and evaluated on the test set of the MrOS-m2ABQ dataset. 61
Figure 3.7. The performance of the model, which was built using the ensemble averaging algorithm in Task 3 and evaluated on the test set of the UW-m2ABQ dataset. 62
Figure 4.1. The validation curve = a trend curve + some random noise. 78
Figure 4.2. Decomposition of the model training cost that has been incurred when we finish the work at the first original validation point. 98
Figure 4.3. A typical shape of p_j as a function of j. 106
Figure 4.4. The learning rate over epochs and a typical validation curve when an exponential decay schedule for the learning rate is used. 112
Figure 4.5. When the learning rate changes over time based upon a step decay method, the learning rate over epochs and an example validation curve. 122
Figure 4.6. Employing the method in Section 4.5.6.1 to estimate the trend curve when one arrives at a validation point that is not far after the most recent decay point. 123
Figure 4.7. The flowchart of estimating the number of original validation points needed for model training when the present validation point resides on the k-th ($k \geqslant 2$) piece of the validation curve. 124
Figure 4.8. The areas of the regions employed to calculate the average prediction error.132
Figure 4.9. Model training cost estimated over time (using Adam and a constant learning rate to train GoogLeNet). 135
Figure 4.10. Model training speed over time (using Adam and a constant learning rate to train GoogLeNet). 136
Figure 4.11. Estimated remaining model training time (using Adam and a constant learning rate to train GoogLeNet). 136
Figure 4.12. Estimate of the remaining model training time at the early stage of model training (using Adam and a constant learning rate to train GoogLeNet). 137
Figure 4.13. Finished percentage estimated over time (using Adam and a constant learning rate to train GoogLeNet). 137

Figure 4.14. Model training cost estimated over time (using RMSprop and a constant learning rate to train the GRU model)..... 138
Figure 4.15. Model training speed over time (using RMSprop and a constant learning rate to train the GRU model). 138
Figure 4.16. Estimated remaining model training time (using RMSprop and a constant learning rate to train the GRU model)..... 139
Figure 4.17. Finished percentage estimated over time (using RMSprop and a constant learning rate to train the GRU model)..... 139
Figure 4.18. Model training cost estimated over time (using Adam and applying an exponential decay method to the learning rate to train GoogLeNet)..... 140
Figure 4.19. Model training speed over time (using Adam and applying an exponential decay method to the learning rate to train GoogLeNet). 140
Figure 4.20. Estimated remaining model training time (using Adam and applying an exponential decay method to the learning rate to train GoogLeNet)..... 141
Figure 4.21. Finished percentage estimated over time (using Adam and applying an exponential decay method to the learning rate to train GoogLeNet)..... 141
Figure 4.22. Model training cost estimated over time (using RMSprop and applying an exponential decay method to the learning rate to train the GRU model). 142
Figure 4.23. Model training speed over time (using RMSprop and applying an exponential decay method to the learning rate to train the GRU model). 142
Figure 4.24. Estimated remaining model training time (using RMSprop and applying an exponential decay method to the learning rate to train the GRU model). 142
Figure 4.25. Finished percentage estimated over time (using RMSprop and applying an exponential decay method to the learning rate to train the GRU model). 143
Figure 4.26. Model training cost estimated over time (using Adam and applying a step decay method to the learning rate to train GoogLeNet). 143
Figure 4.27. Model training speed over time (using Adam and applying a step decay method to the learning rate to train GoogLeNet). 144
Figure 4.28. Estimated remaining model training time (using Adam and applying a step decay method to the learning rate to train GoogLeNet). 144

Figure 4.29. Estimate of the remaining model training time at the early stage of model training (using Adam and applying a step decay method to the learning rate to train GoogLeNet). 144
Figure 4.30. Finished percentage estimated over time (using Adam and applying a step decay method to the learning rate to train GoogLeNet). 145

LIST OF TABLES

Table 2.1. The main attributes in the configuration file, the attributes' meanings, and the modules affected by the attributes. 13
Table 2.2. Description of each user in the usability evaluation listing their occupation, medical imaging experience, and the number of images each annotated. 25
Table 2.3. These survey questions were used to determine the usability of DicomAnnotator across a variety of factors. 28
Table 2.4. A summary of each user's average amount of time needed to annotate a radiograph. 29
Table 2.5. A summary of the ratings for the survey questions in the first round and second round of the annotation session, and the *p*-values to describe the difference in the mean of the ratings between these two rounds. 30
Table 3.6. Metadata for the training, validation, and test sets of the UW dataset, as well as the entire UW dataset. 40
Table 3.7. Demographic information of the subjects in each of the entire, training, validation, and test sets from the MrOS dataset. 45
Table 3.8. The number of vertebral bodies for each (dataset, OCF classification criteria) combination. 46
Table 3.9. Five hyper-parameters that were tuned by random search. 57
Table 3.10. For each GoogLeNet, Inception-ResNet-v2, and EfficientNet-B1, the optimal value of each hyper-parameter in each training task. 57
Table 3.11. F_1 scores, AUC-PR, and AUC-ROC for each (deep learning algorithm, training task, test set) combination. In this table, yellow and magenta are used to mark the MrOS dataset and the UW dataset, respectively. 63
Table 4.12. The datasets that we used to test our progress indication method. 130
Table 4.13. For each of GoogLeNet and the GRU model, the n_0, q, and V' set by the approach given in Section 4.5.3. 131

Table 4.14. For each of the 24 tests, the mean as well as the standard deviation of the average prediction error over the five runs for each of the three progress indication methods.133

LIST OF ABBREVIATIONS

Adam	adaptive moment estimation
AdaGrad	adaptive gradient
AUC-PR	area under the precision-recall curve
AUC-ROC	area under the receiver operating characteristic curve
BERT	Bidirectional Encoder Representations from Transformers
CI	confidence interval
CPU	central processing unit
CT	computed tomography
DICOM	Digital Imaging and Communications in Medicine
FDR	false discovery rate
GIF	Graphics Interchange Format
GPU	graphics processing unit
GRU	Gated Recurrent Unit
GUI	graphical user interface
IRB	institutional review board
JPEG	Joint Photographic Experts Group
m2ABQ	modified-2 algorithm-based qualitative
mABQ	modified algorithm-based qualitative
mSQ	modification of the Genant semiquantitative
MsOS	Osteoporotic Fractures in Women
MrOS	Osteoporotic Fractures in Men
NLP	natural language processing
NPV	negative predictive value
OCF	osteoporotic compression fracture
PACS	picture archiving and communication system
PNG	Portable Network Graphics

PPV	positive predictive value
PR	precision-recall
RIS	radiology information system
RMSprop	root mean square propagation
ROC	receiver operating characteristic
SGD	stochastic gradient descent
TIFF	Tagged Image File Format
UW	University of Washington
VNC	Virtual Network Computing

LIST OF SYMBOLS

$\lceil\rceil$	Ceiling function.
$\lfloor\rfloor$	Floor function.
$\lfloor\rceil$	Nearest integer function.
a	Scaling factor of the inverse power-law function.
b	Exponent of the inverse power-law function.
b_{max}	Maximum number of batches allowed for model training.
b_l	Lower bound of the validation error at any validation point.
$\hat{b}_l$	Estimate of b_l.
b_u	Upper bound of the validation error at any validation point.
$\hat{b}_u$	Estimate of b_u.
B	Number of training instances in each batch.
c	Bias term of the inverse power-law function.
c_0	Model training cost that has been incurred when we finish the work at the first original validation point, excluding the progress indicator's overhead of calculating the validation errors at the added validation points.
c_j	Number of validation instances that are misclassified by the model and in the actual validation set used at the j-th validation point.
c_v	Cost to calculate the validation error at the first original validation point.
c_γ	Coefficient used to compute γ.
C	Upper threshold of the model training cost that has been incurred when we finish the work at the fourth validation point.
D	Image's bit depth.
e_j	Model's generalization error at the j-th validation point.
$\hat{e}_j$	Validation error of the model at the j-th validation point.
$\tilde{e}_j$	Validation error of the model at the j-th original validation point.
$f(q)$	A function of q.

g Number of batches of model training between two consecutive validation points.

$h(n)$ Present validation point's sequence number on the present piece of the validation curve.

I Intensity level of a pixel.

I' Intensity level of a pixel after the automatic window and level adjustment.

K Size of the sliding time window used for computing the model training speed.

l_j Number of validation points that are on the j-th piece of the validation curve.

L Percentage of usability problems a user can find on average.

L_d Deepest frozen layer.

m Number of users.

m_e Maximum number of epochs allowed for model training.

n Number of validation points obtained thus far.

n_0 Number of validation points added before the first original validation point.

n_j Number of validation points added between the j-th and the (j+1)-th original validation points.

n_v Number of original validation points needed for model training.

N Total number of usability problems.

p Patience.

p_j Percentage increase in the model training cost that the progress indicator causes during the period from when model training starts to the time we finish the work at the j-th original validation point.

p_{max} Maximum intensity level of an image patch.

p_{min} Minimum intensity level of an image patch.

P_1 Maximum allowed percentage increase in the model training cost that the progress indicator causes during the period from when model training starts to the time we finish the work at the first original validation point.

P_v Maximum allowed percentage increase in the model training cost that the progress indicator causes during the period from when model training starts to the time we finish the work at the v_{max}-th original validation point.

q Constant regulating the decay rate of n_j ($0 \leq j \leq v_{max} - 1$) in the exponential decay schema.

r	Number of disjoint intervals into which the possible range of the simulated number of validation points needed for model training is divided.
r_0	Beginning learning rate adopted in the exponential decay method.
r_j	Learning rate right before the j-th validation point.
$r(i)$	Regression function's value at the i-th validation point.
$s_{k\text{-}1}$	Sequence number of the final validation point that is on the prior piece of the validation curve.
T	Number of data instances in the training set.
U	Unit of work.
v_{max}	Maximum number of validation points allowed for model training.
V	Number of data instances that are in the full validation set.
V_j	Number of data instances that are in the actual validation set used at the j-th validation point.
V_{min}	Minimum number of data instances needed in the randomly sampled subset of the full validation set used at an added validation point.
V'	Uniform number of data instances that are in the randomly sampled subset of the full validation set used at each added validation point.
w	Number of validation points used to fit the regression function.
w'	Maximum number of validation points allowed to fit the regression function.
x_j	Normalized number of batches of model training finished before the j-th validation point.
z	Constant regulating the decay rate of n_j ($0 \leq j \leq v_{max} - 1$) in the linear decay schema.
α	The beta distribution's first shape parameter.
β	The beta distribution's second shape parameter.
$B(\alpha, \beta)$	Normalization constant in the probability density function of the beta distribution.
γ	Threshold used to decide which interval is regarded as a local mode.
δ	min_delta.
ε_j	Random noise at the j-th validation point.
λ	Ratio of the variance of the model's generalization error to the square of the learning rate.
μ_j	Mean of the model's generalization error at the j-th validation point.
μ'_j	Mean of the beta distribution linking to the j-th validation point.

$\tilde{\mu}_j$	Mean of a normal distribution linking to the *j*-th validation point.
ρ	Constant regulating the decay rate of the learning rate in the exponential decay method.
σ_j^2	Variance of the model's generalization error at the *j*-th validation point.
$\sigma_j'^2$	Variance of the beta distribution linking to the *j*-th validation point.
$\hat{\sigma}^2$	Estimated variance of the random noise when a fixed learning rate is used during the entire model training process.
$\tilde{\sigma}_j^2$	Variance of a normal distribution linking to the *j*-th validation point.
τ_v	Threshold on the number of validation points reached, beyond which we use the validation curve to refine the projected number of validation points needed for model training.
$\Phi()$	Cumulative distribution function of the standard normal distribution.

ACKNOWLEDGEMENTS

I could not have reached this goal without the help of many people in my PhD years. I'd like to take this opportunity to thank them for their support.

First, I would like to express my sincere thanks to my research advisor and the chair of my committee, Professor Gang Luo, for his invaluable patience and feedback. It was he who guided me into the world of research and taught me how to consider and solve a problem like a researcher. It was he who consistently provided much wisdom and valuable advice to help me overcome difficulties and steer me in the correct direction. It was he who always extensively edited my papers with great patience and effort. It was he who had conceived the idea of progress indication for deep learning model training in the first place, which became an aim of this dissertation. It was he who recruited me and offered me the opportunity to work as a research assistant in the UW CLEAR center. I am grateful for all of the tedious paperwork he handled to help me continuously get funded during the entire PhD career.

Second, I would like to extend my sincere gratitude to the advisor of my research assistant work, Professor Nathan Cross, who is also a member of my committee. Thanks for his feedback and advice on my research assistant work. Thanks for his patient explanation of the clinical background of my research, which became an important topic of this dissertation. Thanks for all his contributions that led to the completion of this dissertation, including obtaining the Osteoporotic Fractures in Men (MrOS) Study dataset from the MrOS team, leading the annotation group, annotating the images, and helping edit my papers.

Third, I would like to express my gratitude to Professor Jeffrey Jarvik, who is another member of my committee. He is the director of the UW CLEAR center, where I worked as a research assistant during my entire PhD career. Thanks for financially supporting me to go through the PhD years. Also, I am grateful for his assistance in annotating the images for this research and his insights during the regular meetings.

Fourth, I would like to express my appreciation for the help from Professor Fei Xia and Professor Sean Mooney. Professor Fei Xia is the Graduate School Representative on my committee. She kindly provided feedback on my research. Professor Sean Mooney was a member

of my committee and now is the Director of the NIH Center for Information Technology. Although he is no longer on my committee, he identified some issues in my research and kindly provided suggestions. Best wishes for his new career in NIH.

Fifth, I would like to acknowledge the support from other members in the Image Processing Group (IPG) of the UW CLEAR center, the group in which I worked as the research assistant. Thank Dr. Sandra Johnston, who is also the administrative director of the UW CLEAR center, for her hard work in coordinating everything. Thank Professor David Haynor for actively contributing new ideas during the IPG meetings. Thank Dr. Brian Chang, Jonathan Renslo, and Jessica Perry for valuable discussions. Besides the IPG, I would like to thank the annotation group for annotating the images and the MrOS team for providing the MrOS dataset and their feedback on this research.

Sixth, I would like to acknowledge the help from all members in Professor Gang Luo's lab. Thanks for helping me practice my presentation and providing useful suggestions. Additionally, I extend my gratitude to Xiaoyi Zhang for valuable discussions on the progress indication topic and assisting with paper revisions.

Seventh, I would like to thank the BIME family. I am grateful to Heidi Krueger, Marni Levy, and Heather Clausnitzer for all your administrative support. I appreciate Professor John Gennari and Professor Peter Tarczy-Hornoch for all your administrative and academic help. Thank all professors who taught me the BIME core courses in my first and second years, which showed me the breadth of the biomedical informatics world.

Last but not least, I would like to express my appreciation for the emotional support from my family and friends. I am particularly grateful to my parents and grandparents for their encouragement and care. When there was a harsh time, especially during the pandemic, their video calls always cheered me up. Thank George Xu and Yifan Wu for your friendship. Cheers!

Chapter 1. INTRODUCTION

1.1 BACKGROUND

Osteoporosis affects 9% of individuals over 50 years old in the US [1] and 200 million women globally [2]. In developed countries, one out of three individuals will suffer an osteoporotic compression fracture (OCF) in their lifetime [2]. After the first OCF, the risk for subsequent OCFs increases greatly [3-5]. Even one OCF can decrease quality of life and increase risk of mortality [6].

Osteoporosis screening is evidence-based and is endorsed by many organizations, including the US Preventive Services Task Force, but remains underutilized. Between 2004 and 2006, more than 2/3 of women who should have been screened for osteoporosis were not [7]. From 2006-2010, screening of US women with Medicare using dual-energy X-ray absorptiometry decreased by 56% [8]. The rate of osteoporosis screening for high-risk men is also low [9].

Opportunistic osteoporosis screening, which uses pre-existing imaging to increase osteoporosis detection rates, can complement current osteoporosis screening methods and is desired to introduce minimum extra cost. Several approaches to opportunistic osteoporosis screening have been proposed [10-29]. Many research groups used computed tomography (CT) images [10-22], while few used radiographs [23-29]. Radiography is a ubiquitous imaging modality used early in diagnostic workup of many conditions with an estimated 183 million exams in US hospitals in 2010 [30]. Thus, using radiographs to conduct opportunistic osteoporosis screening is as important as using CT and could potentially reach a broader patient population. Using radiographs, Lee *et al.* [23] and Zhang *et al.* [24] used machine learning algorithms to estimate bone mineral density. However, using bone mineral density as a biomarker of

osteoporosis detection has known limitations [31, 32]. Spinal OCFs can serve as an additional osteoporosis biomarker and are often incidental on chest or abdominal images and frequently under-reported, resulting in under-diagnosis and under-treatment [33]. Applying automated opportunistic OCF screening to existing imaging studies could result in earlier and more extensive osteoporosis identification and treatment. Multiple studies [25-29] have attempted to automatically detect OCFs using radiographs. However, these studies had limitations including single center data leading to possible overfitting [25-28] and unclear dataset construction processes [29].

We ultimately aim to build an automated opportunistic OCF screening tool to detect OCFs using the radiographs that contain lumbar and/or thoracic vertebrae. Such radiographs include lumbar spine radiographs, thoracic spine radiographs, and chest X-rays. Our opportunistic OCF screening tool has three primary sequential components (see Figure 1.1): 1) image segmentation and extraction of vertebral bodies; 2) a binary OCF classifier predicting whether each vertebral body has a moderate to severe OCF or not; and 3) a subject-level classifier integrating the OCF predictions of all vertebral bodies with additional structured data to determine this subject's OCF status. Adequate performance of any clinical test can only be judged in the context of the use case. Considering a screening tool for large volumes of studies, a tool with too many false positives could unduly burden the health care system. Thus, we prioritize positive predictive value (PPV) and specificity of the model rather than sensitivity.

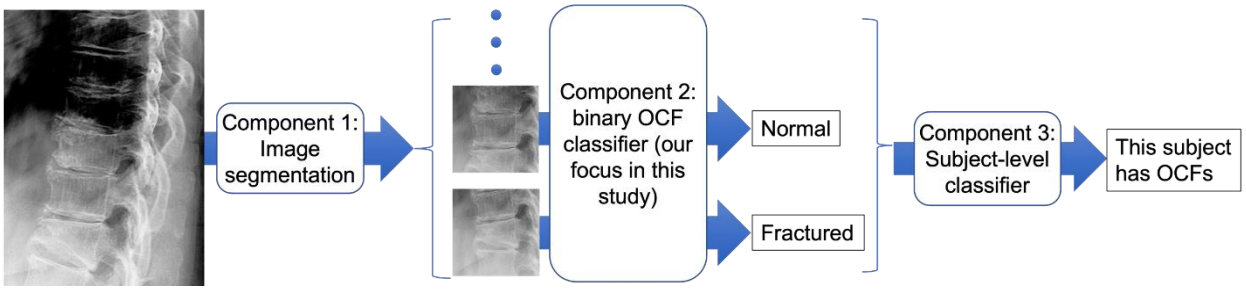


Figure 1.1. Our future automated opportunistic screening tool detecting OCFs on radiographs.

In this dissertation, we focus on the second component, the binary OCF classifier (see Figure 1.1). This component predicts whether an image patch containing a single vertebral body (termed *vertebral patch*) has a moderate to severe OCF or not. Each of the first and the third components is a distinct body of work [34]. Recall that the first component is used to automatically extract the vertebral patches. Since we do not use the first component in this dissertation, we extract each vertebral body using manually annotated corner points. Connecting the three components to build the entire automated opportunistic screening tool is one of our future directions.

To build the OCF classifier, we obtain two spine radiograph datasets with multicenter data: the Osteoporotic Fractures in Men (MrOS) Study dataset [35, 36] and the University of Washington (UW) dataset [37]. For each of the lumbar and thoracic vertebral bodies in each radiograph, we manually annotate its corner points and assign it a fracture label. As manual annotation is usually labor-intensive and time-consuming, a well-designed software program can aid and expedite the annotation process. This program is required to be configurable for various annotation tasks, enable efficient placement of several types of annotations on an image or a region of an image, attribute annotations to individual annotators, and be able to display the Digital Imaging and Communications in Medicine (DICOM)-formatted [38] images. Multiple annotation programs are publicly available for general [39-45] and medical [46-50] images. However, to the best of our knowledge, none of them fulfill all of the requirements mentioned above. Additionally, some of these programs have limited licensing models that are challenging for a small project budget. Therefore, we developed a configurable open-source software program named DicomAnnotator [51] to fill this gap for DICOM image annotation.

A deep learning model is a multi-layer neural network that can extract features from unstructured data such as medical images. Since deep learning significantly outperforms other

machine learning algorithms for image classification [52], we used deep learning to build our OCF classifier. We tried five deep learning algorithms and compared their performance on OCF classification.

The development of deep learning models is often time-consuming on large datasets and requires substantial computing resources. Using 50 graphics processing units (GPUs), a Google team spent two months training a deep neural network on 300 million images [53]. With 200 central processing units (CPUs), Weyand *et al.* [54] took 2.5 months to train a convolutional neural network on 126 million photos. With one GPU, Colón-Ruiz *et al.* [55] spent more than 22 hours training a Bidirectional Encoder Representations from Transformers (BERT) model [56] on 215,063 drug reviews for sentiment analysis. Using a GPU, Guo *et al.* [57] spent ten hours to train a convolutional neural network on approximately 10 million image patches with a resolution of 28×28 for tumor segmentation. Sufficient computing resources could be difficult for individual clinicians or clinical research groups to obtain [58], making deep learning model training more time-consuming. As a standard human-computer interaction principle [59], for each task running longer than 10 seconds, we need a non-trivial progress indicator (see Figure 1.2) to continuously project the remaining task running time and the fraction of the task completed. Thus, progress indicators are desirable to make the deep learning model training process more user-friendly.

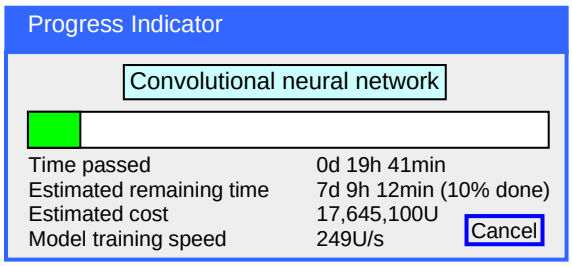


Figure 1.2. A progress indicator for deep learning model training.

A neural network is trained in one or more epochs, each of which requires going through all of the training instances once. Some deep learning software supplies trivial progress indicators during model training, e.g., by displaying the number of epochs that have been completed [60] or the value of the objective function achieved [61] over time. However, this information is too coarse-grained for many purposes. On a large dataset, a large amount of time is needed to go through an epoch. Moreover, early stopping is widely used in deep learning model training to help avoid overfitting. When early stopping is allowed, the number of epochs needed for model training is unknown beforehand but dynamically decided during model training based on a stopping criterion [62]. How to support such progress indicators in the presence of early stopping remains an open problem.

1.2 CONTRIBUTIONS

To fill the gap for DICOM image annotation, in this dissertation we present our annotation software program named DicomAnnotator [51]. It is configurable for various annotation tasks, enables efficient placement of several types of annotations on an image or a region of an image, attributes annotations to individual annotators, and can display DICOM-formatted images.

We build the OCF classifier [37, 63], the critical component of our future automated opportunistic screening tool (see Figure 1.1). Our OCF classifier achieves an area under the precision-recall (PR) curve (AUC-PR) > 0.70 and an area under the receiver operating characteristic (ROC) curve (AUC-ROC) > 0.90 on both the MrOS and the UW radiograph datasets. Performance evaluation of our OCF classifier shows that it has some generalizability to clinical data and a suitable performance for our future opportunistic screening tool.

We introduce the first set of techniques to support non-trivial progress indicators for deep learning model training when early stopping is allowed [64]. A follow-up method is further

developed to improve the accuracy of progress indication for deep learning model training [65]. We implemented our techniques in TensorFlow [66], an open-source deep learning software package. With negligible run-time overhead, the resulting progress indicator can provide useful information even if the run-time system load varies over time [64] and can self-correct its initial estimation errors, if any, over time. Our progress indication methods can handle various combinations of the deep learning model, the learning rate schedule (e.g., learning rate decay), and the optimization method.

1.3 DISSERTATION OVERVIEW

In this dissertation, we have the following three aims:

1) **Aim 1:** Develop DicomAnnotator and use it to annotate the datasets.
2) **Aim 2:** Build the OCF classifier using deep learning and the annotated datasets.
3) **Aim 3:** Design progress indication methods that can support non-trivial progress indicators for deep learning model training in the presence of early stopping.

In Chapter 2, we introduce DicomAnnotator's modules and features, followed by the usability evaluation of DicomAnnotator.

Chapter 3 describes how to use the annotated datasets and deep learning to build the OCF classifier. In this chapter, we introduce the UW and the MrOS datasets, followed by how to pre-process the radiographs. Then we describe how to train the OCF classifier, followed by the performance evaluation of our OCF classifier.

In Chapter 4, we describe two progress indication methods and show the performance of the corresponding progress indicators.

Chapter 5 summarizes the entire dissertation.

Chapter 2. DICOMANNOTATOR: A CONFIGURABLE OPEN-SOURCE SOFTWARE PROGRAM FOR EFFICIENT DICOM IMAGE ANNOTATION

Modern, supervised machine learning approaches to medical image classification, image segmentation, and object detection usually require many annotated images [67, 68]. As manual annotation is usually labor-intensive and time-consuming, a well-designed software program can aid and expedite the annotation process. Ideally, this program should be configurable for various annotation tasks, enable efficient placement of several types of annotations on an image or a region of an image, attribute annotations to individual annotators, and be able to display DICOM-formatted images. No current open-source software program fulfills these requirements. To fill this gap, we developed DicomAnnotator [51], a configurable open-source software program for DICOM image annotation. This program fulfills the above requirements and provides user-friendly features to aid the annotation process. In this chapter, we present the design and implementation of DicomAnnotator. Using spine image annotation as a test case, our evaluation showed that annotators with various backgrounds can use DicomAnnotator to annotate DICOM images efficiently. DicomAnnotator is freely available at https://github.com/UW-CLEAR-Center/DICOM-Annotator under the GPLv3 license.

In this chapter, we start with the introduction to DicomAnnotator in Section 2.1. Then we describe DicomAnnotator's functional modules and ancillary features in Sections 2.2 and 2.3, respectively. In Section 2.4, we describe the usability evaluation of DicomAnnotator. In Section 2.5, we discuss the strengths, limitations, and future work of DicomAnnotator. Section 2.6 concludes this chapter.

2.1 INTRODUCTION

Modern, supervised machine learning approaches to image classification, segmentation, and object detection typically require many annotated images. Sa *et al.* [67] used 974 annotated X-ray images to build a model for intervertebral disc detection. Esteva *et al*. [68] used 129,450 annotated clinical images to construct a model for classifying skin cancers. Performing manual image annotation is labor-intensive, tedious, and time-consuming, but often necessary for generating a high-quality dataset. Annotators frequently need to place several types of annotations in multiple areas of an image. In tasks with ambiguity or subjectivity, multiple experts whose time is expensive need to participate in the annotation task to reach consensus, multiplying the effort required for each dataset.

A well-designed software program can expedite image annotation. Ideally, the annotation program should meet the following requirements:

1) Customizable configurations should be available to support diverse annotation tasks. Tasks could require using differing types of labels, such as one label for the whole image vs. several other labels for the regions of interest in the image. Also, differing shapes might be needed to segment the regions of interest in the image. For instance, four corner points can outline a vertebral body on a spine image, whereas, a more complex shape would be needed to segment a lobulated or irregular mass.
2) The program should allow efficient placement of several types of annotations on one image, such as polygons to outline the regions of interest (termed *bounding polygons*), labels for the regions of interest, and a label for the whole image.
3) Several labels could be needed for each bounding polygon or region of interest. Consider an annotation task dealing with many regions of interest, each with its own identifier (see Figure

2.1). For each region, some programs require the annotator to put a bounding polygon on the region, select its identifier from a long list, and then input the labels [39, 40]. Instead of having this inefficient workflow, an annotation program should support an optimized annotation approach, where multiple labels can be efficiently applied to a region of interest.

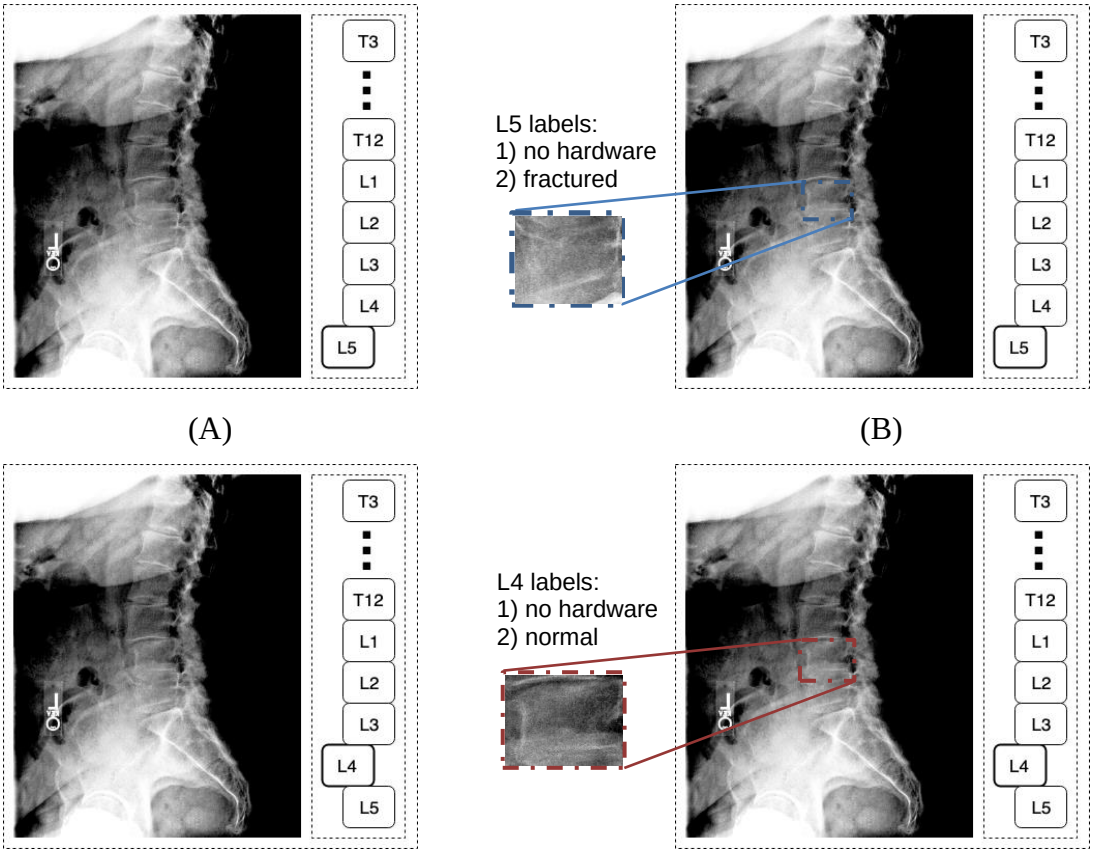


Figure 2.1. Our approach to annotating multiple regions of interest in an image.
The subfigures are: (A) the "L5" region identifier is selected, (B) the annotator provides a bounding polygon for L5 and then inputs the labels for L5 (no hardware, fractured), (C) the "L4" identifier is automatically selected after completing the last labels for L5, and (D) the annotator provides a bounding polygon for L4 and then inputs the labels for L4 (no hardware, normal). This process continues until image annotation is complete.

4) For consensus annotations, the program should track which label was made by which annotator. This can facilitate comparing the annotations and computing inter-reader agreement.
5) Most medical images are of a higher bit depth than the standard 8-bit graphic formats like Joint Photographic Experts Group (JPEG) and Graphics Interchange Format (GIF) support. DICOM is the standard format encapsulating medical images, patient information, and relevant imaging metadata [38]. Within a DICOM file, images are often stored in a high-fidelity state, either uncompressed or compressed with lossless and less frequently lossy compression. Depending on the modality, the bit depth is often higher than 8-bit. Ideally, a program should allow direct viewing of DICOM formatted medical images with window/leveling capabilities to access the full bit depth of the original data. Given the high spatial resolution of many medical images, additional image manipulation functions like zooming and panning are also useful.

Multiple annotation programs for general [39-45] and medical [46-50] images are publicly available. But, to the best of our knowledge, none of them fulfills all of the requirements mentioned above. In addition, some of these programs have limited licensing models that are challenging for a small project's budget. To fill this gap, we developed DicomAnnotator, a configurable open-source software program for DICOM image annotation. We designed several basic functional modules (see Section 2.2) to satisfy the above requirements and four user-friendly features (see Section 2.3) to aid and accelerate the annotation process. Although this program is designed for annotating DICOM images, it also supports displaying and annotating JPEG, Portable Network Graphics (PNG), and Tagged Image File Format (TIFF).

We built DicomAnnotator in Python 3.7.3 and used the Anaconda3 distribution to manage packages. The main packages include PyQt5 for designing the user interface, SimpleITK [69] for reading DICOM images, Matplotlib for displaying images, and NumPy for processing arrays and

matrices. DicomAnnotator and its installation instructions are available at https://github.com/UW-CLEAR-Center/DICOM-Annotator.

2.2 FUNCTIONAL MODULES

DicomAnnotator was designed to fulfill the requirements mentioned above through a user-friendly graphical user interface (GUI). We designed several basic functional modules to satisfy the requirements. These modules are illustrated below using our spine image annotation task as a running example. Given a spine image, our annotators needed to perform five basic annotation sub-tasks: 1) assign an osteoporosis category [70] (normal, possible osteopenia, and definite osteopenia) to the whole image; 2) outline the vertebral bodies using bounding polygons, which are quadrilaterals not necessarily oriented with the x and y axes; 3) determine each vertebral body's anatomic level (e.g., L4, L5, and S1); 4) label whether each vertebral body had any overlying artificial structure (e.g., spine hardware, metallic object, or catheter); and 5) score each vertebral body using one of several fracture classification systems (e.g., the semi-quantitative criteria [71], the algorithm-based qualitative criteria [72], or the modified algorithm-based qualitative (mABQ) criteria [73]). In the rest of Chapter 2, we call the labels for the whole image "*image labels*" and the labels for the regions of interest "*region labels.*"

The annotation program is divided into five modules: 1) the configuration file, 2) the login page, 3) the image viewing module, 4) the annotation module, and 5) the result file. The configuration file eases the configuration process, as the annotator can modify the configuration file without touching the program's source code to adapt the program to a new annotation task. The login page allows the annotator to enter a username to track the annotator's annotations. The image viewing module supports image displaying, zooming, panning, and windowing/leveling.

The annotation module is used for placing annotations on an image. The result file stores the annotation results and other information generated during the annotation process.

As Figure 2.2 shows, the annotator uses the modules sequentially to complete an annotation task, although they are integrated into a common user interface. Before running the program for the first time, the configuration for the given annotation task is set up in the configuration file. After the program starts running, the annotator needs to provide a username on the login page. Next, the main page for image viewing and annotation loads. The annotator uses the GUI to annotate and navigate through the images in the dataset. Throughout this process, the annotations are stored in a result file for later use.

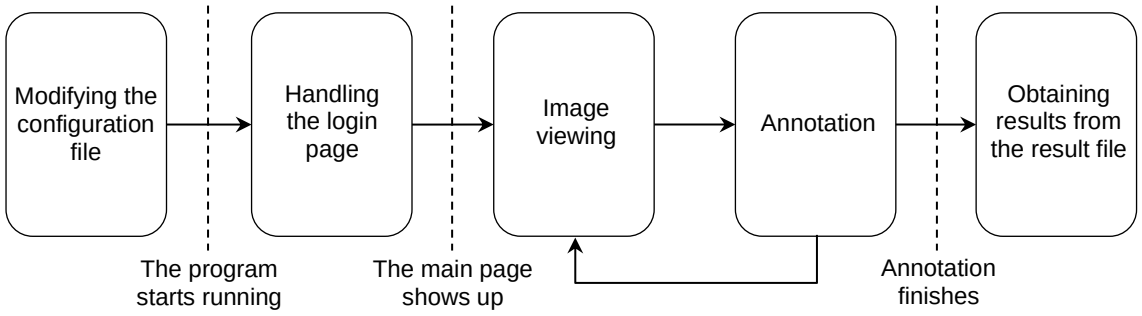


Figure 2.2. The steps to using DicomAnnotator to annotate an image dataset.

In Sections 2.2.1-2.2.5, we show these modules' functionality using a radiograph. In Section 2.2.6, we show how our program can handle CT images.

2.2.1 *Configuration File*

By modifying some attributes in the configuration file, an annotator can adapt the program to a new annotation task. For example, in the spine annotation task, one sub-task is to assign an osteoporosis label to the whole image. The osteoporosis label has three candidate categories: normal, possible osteopenia, and definite osteopenia. In the configuration file, the annotator can

type these categories under the “image label” attribute, which is used to customize the candidate categories of an image label. Then when the program runs, these categories will appear on the program’s user interface.

Adjusting some of the attribute values in the configuration file can improve user experience. One example is the “zooming speed” attribute, which determines how quickly the image zooms in/out when the annotator scrolls the mouse wheel.

Table 2.1. The main attributes in the configuration file, the attributes’ meanings, and the modules affected by the attributes.

Attribute	**Meaning**	**Affected module**
List of region identifiers	List of regions needing to be labeled for each image, e.g., the anatomic levels (L1, L2, L3…).	The annotation module
Number of vertices of the bounding polygon	Number of vertices of the polygon that is used to capture each region of interest.	
List of region labels	Possible values of each region label. Here, the annotator provides a two-dimensional array. Each element of this array appears on a separate line and contains the possible values of a region label. In the spine image annotation example, the annotator assigns two types of region labels. Accordingly, the array is displayed in two lines. The first line lists the possible values of whether any artificial object overlays the vertebral body: yes and no. The second line lists the possible category values used in a fracture classification system like the mABQ criteria [73].	
Image label	Possible values of the image label.	
Input directory	Directory from which the program reads the input images.	The image viewing module
Zooming speed	Determines how fast the image zooms in/out when the annotator scrolls the mouse wheel.	
Windowing/leveling sensitivity	Controls the rate at which the window and the level change when the mouse moves a unit of length.	
Name/path of the result file	Specifies the file storing the results.	The result file

Customizing the attributes could affect several functional modules including the annotation module, the image viewing module, and the result file. Table 2.1 lists the main attributes in the configuration file, the attributes' meanings, and the modules affected by the attributes. An example of modifying the configuration file is given in Section 2.2.6.

2.2.2 *Login Page*

The login page includes a field for an annotator to enter their username. When an annotator makes an annotation, the program maps the annotation to the supplied username and records the activity in the result file. In this way, the program knows the annotation is made by this annotator. When a group of annotators collaborate on an annotation task, this helps the program track which annotation is made by which annotator.

The login page has another function allowing the annotators to quickly set some basic configurations. For an annotation task, if only a few configurations need to be specified, directly customizing them on the login page is more efficient than editing the configuration file. For instance, Figure 2.3 shows the login page for the spine annotation task. The first entry is for the annotator to input their username. The next two rows are used to select the configurations: the fracture scoring system and the order of the region identifiers shown on the program's main page. When the "START" button is clicked, the program will document the username and open the main page.

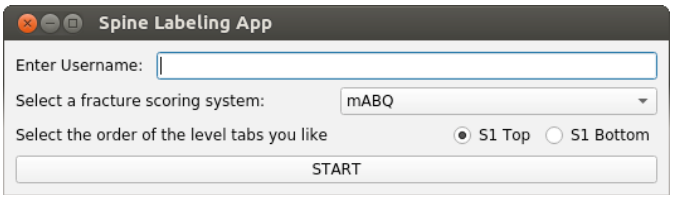


Figure 2.3. The login page.

2.2.3 *Image Viewing Module*

The image viewing module supports displaying DICOM images with a high bit depth and resolution, regardless of the medical imaging modality. It also supports displaying JPEG, PNG, and TIFF images. Figure 2.4 is a screenshot showing how our program displays a DICOM image on the main page. The widgets in each panel of Figure 2.4 are explained in the following:

1) **Panel 1:** Radio buttons used to select an operation mode.
2) **Panel 2:** A group of buttons that allow the user to move between images, remove annotations, reset the image display, manually save annotations, and display help text.
3) **Panel 3:** Text box showing details about the currently displayed image and the annotation process.
4) **Panel 4:** Buttons used to set an image to unreadable when it is of low or non-diagnostic quality and to horizontally flip the image.
5) **Panel 5:** Buttons used to flag/unflag an image for later review and to navigate through the flagged images.
6) **Panel 6:** Commenting system where comments from any user are displayed and new comments can be added.
7) **Panel 7:** Indicator of whether new annotations have been stored in the result file.
8) **Panel 8:** Canvas displaying an image.
9) **Panel 9:** Radio buttons for assigning an image label.
10) **Panel 10:** Annotation table which has been configured to apply multiple annotations to each region of interest.
11) **Panel 11:** Buttons used to toggle off the annotated points in the image and to invert the image's grayscale.

12) **Panel 12:** Text boxes showing the identifiers of the regions that are not assigned the default region label like “Normal.”

Figure 2.4. The main page of DicomAnnotator that is divided into 12 panels.

The image viewing module supports zooming, panning, and windowing/leveling. In Figure 2.5(B), the target vertebral body (L2) has been enlarged and centered using zooming and panning. The image’s contrast and brightness has been adjusted using windowing/leveling. By default, zooming is done by scrolling the mouse wheel, panning by holding down the right mouse button and moving the mouse, and window/level adjustment by holding down the “shift” button on the keyboard and moving the mouse horizontally and vertically to adjust the window and level, respectively. The program can record the window and level values adjusted by the annotator in the result file, allowing the annotator to view the image with these adjusted values when returning to this image.

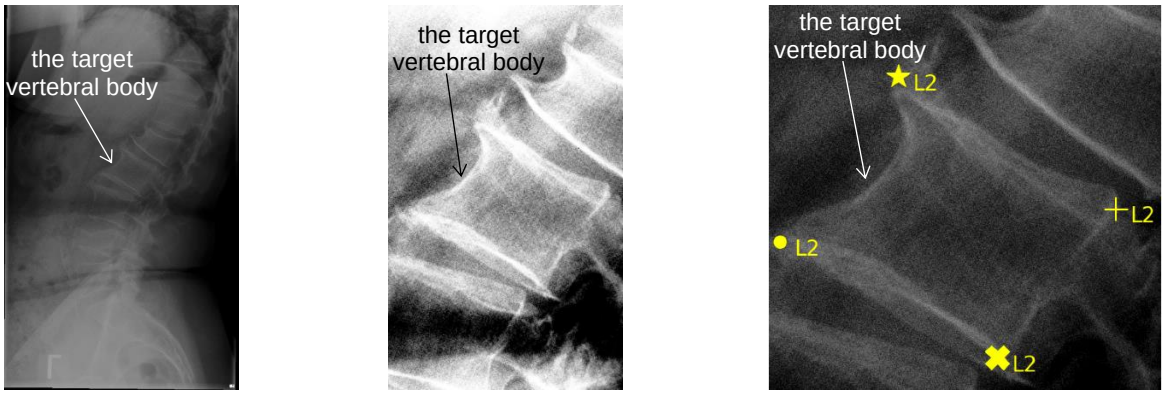


Figure 2.5. Illustration of user's interaction with the image in the annotation process.
(A) The target vertebral body for annotation is identified by the user using the default display parameters; (B) The display window and level, zooming and panning are employed to optimize visualization of the target vertebral body; (C) The boundary of the target vertebral body is marked by placing points at its four corners.

2.2.4 *Annotation Module*

The annotation module is used to place bounding polygons, region labels, and image labels. The approach to placing bounding polygons and region labels satisfies the third requirement mentioned in Section 2.1. We created an annotation table (Panel 10 of the main page) with multiple tabs, each representing a region of interest like a vertebral body in the spine annotation task. Before placing the bounding polygon and the region labels, the associated tab is highlighted (e.g., S1 is highlighted in Figure 2.4). The annotator interacts with the widgets inside the tab to assign the related labels and clicks the canvas to place the bounding polygon.

A bounding polygon is outlined by putting a given number of points on the canvas as the polygon's vertices. Each point is given by clicking a place on the canvas. As shown in Figure 2.5(C), by clicking the four corners of the vertebral body, the associated points are placed, forming a quadrilateral for extracting the vertebral body.

Finally, an image label is assigned using Panel 9 (see Figure 2.4). Figure 2.6 shows the final annotations of a spine image.

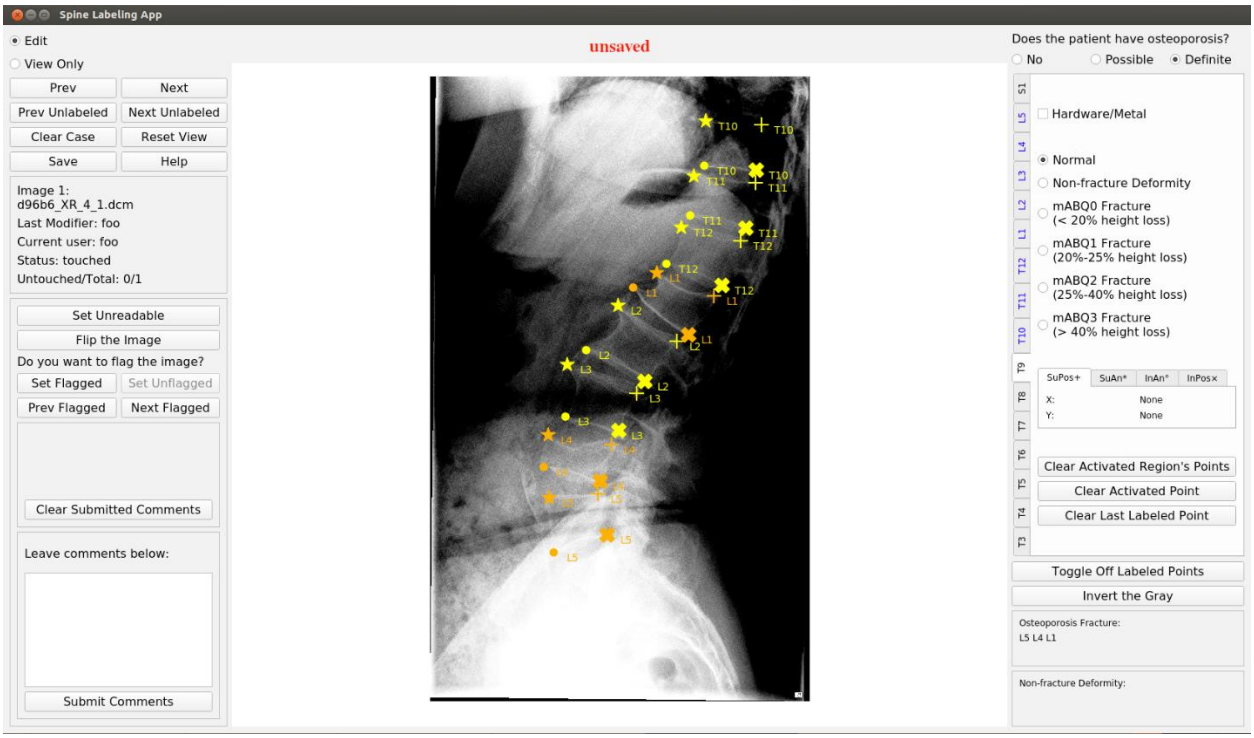


Figure 2.6. The final annotations of an example spine image.

2.2.5 *Result File*

The result file stores the annotation results. It also records certain useful information including the username, the window and level values adjusted by the annotator for each image, the annotator's comments (see Section 2.3.2 for the commenting system), and whether each image has been annotated. The username tracks which annotator places which annotation. With the window/level information, the annotator can review an image with its previously adjusted window and level values. The annotator's comments remind the annotator of the concerns raised during annotation. If during the annotation process, the annotator closed and then restarts the program, the recorded

information on whether each image has been annotated enables the program to load the first unannotated image and resume the annotation process.

2.2.6 *An Example of Modifying the Configuration File*

We give an example of modifying the configuration file for annotating spine CT images, which is different from the running example used in Sections 2.2.1-2.2.5. This annotation task requires an annotator to annotate lumbar vertebral bodies in a sequence of four sub-tasks: 1) decide each vertebral body's anatomic level; 2) outline each vertebral body by placing four points at the vertebral body's corners and two points in the middle of the vertebral body's endplates; 3) for each vertebral body, check whether it is fractured and whether any artificial object overlays it; 4) for the CT image, decide an osteopenia score for the spine based on Saville's method [74], which used five different grades (Grades 0 to 4) to describe various osteopenia severities. Sub-task 2 is to place the bounding polygon. Sub-task 3 is to assign labels to each region of interest. Sub-task 4 is to provide the image label.

For this annotation task, we modify the configuration file as shown in Figure 2.7. Figure 2.8 shows DicomAnnotator's main page after the annotation task is done. In the configuration file, the image_label_description attribute describes the image label and is shown at the top right corner of the main page (see Figure 2.8), where the annotator selects an image label. The region_labels attribute gives all possible region labels. There, the checkbox sub-attribute shows those possible region labels, each of which appears as a separate check box in the annotation table on the main page (see Figure 2.8). The radiobuttons sub-attribute shows those possible region labels that are listed as radio buttons in the annotation table on the main page (see Figure 2.8). The radiobuttons sub-attribute is equivalent to the "list of region labels" attribute listed in Table 2.1. Table 2.1 also describes the other attributes shown in Figure 2.7.

```
{
        "input_directory": "images/",
        "zooming_speed": 1.2,
        "window_level_sensitivity": 1,
        "path_for_result_file": "results.csv",
        "list_region_identifiers": ["L5", "L4", "L3", "L2", "L1"],
        "image_label_description": "Osteopenia grade",
        "image_label": ["0", "1", "2", "3", "4"],
        "region_labels":
        {
                "checkbox": "Artificial structure overlying",
                "radiobuttons": [["Normal", "Fractured", "Unsure"]]
        },
        "number_bounding_polygon_vertices": 6
}
```

Figure 2.7. The configuration file for the spine CT image annotation task in JSON format.

Figure 2.8. DicomAnnotator demonstrating display and annotations of an example sagittal lumbar spine CT image.

2.3 Ancillary Features

We use four ancillary features to make the annotation program more user-friendly. First, two operation modes, “Edit” and “View Only,” are used to enable and disable the annotation module,

respectively. Second, a commenting system allows annotators to leave comments on each image during annotation. Third, an automatic window and level adjustment function is used to reduce the need to adjust the window and level of an image. Fourth, two functions, one for splitting an image set into several parts and the other for merging the result files from multiple annotators, are available to help multiple annotators collaborate on the same annotation task. To illustrate these features, we still use our spine image annotation task used in Sections 2.2.1-2.2.5 as a running example.

2.3.1 *Operation Modes*

Our software has two operation modes. The "Edit" mode allows the annotator to annotate the images. The "View Only" mode disables the annotation module, preventing new annotations from being put on the images and allowing the annotator to only view the images and the previously placed annotations. As Figure 2.9 shows, when the "View Only" mode is chosen, the widgets for placing annotations (e.g., the annotation table) are disabled. On the canvas, the function for placing the polygon's vertices is disabled, while image manipulations are still allowed.

The "View Only" mode is useful when an annotator checks the annotations. Disabling the annotation module can avoid unintended modification to the existing annotations, e.g., clicking the canvas and accidentally placing an extra point. Depending on whether the annotator is annotating the image or checking the annotations supplied by others, our program can automatically select a proper mode. If the image is previously annotated by another annotator, the program regards the current annotator to be checking the existing annotations and switches to the "View Only" mode. Otherwise, the program runs in the "Edit" mode.

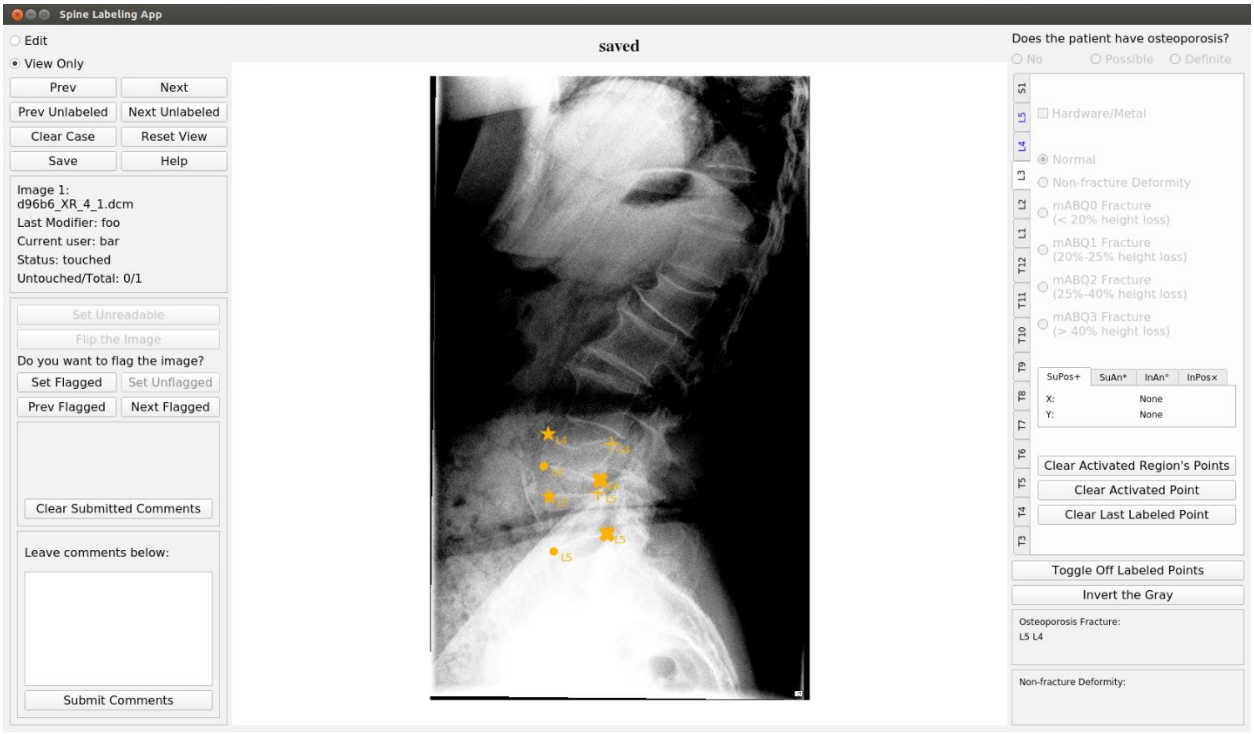


Figure 2.9. The main page in the "View Only" mode.

Regardless, annotators can manually switch the mode by selecting one of the radio buttons on Panel 1 (see Figure 2.4). When the annotator tries to switch from the "View Only" mode to the "Edit" mode, a dialog (Figure 2.10) pops up asking the annotator for confirmation. This further reduces the risk of accidentally modifying the existing annotations.

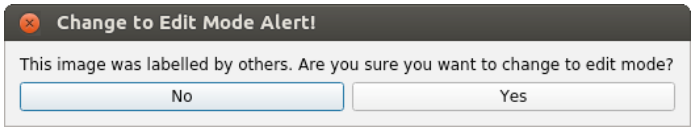


Figure 2.10. The confirmation dialog that is displayed when switching from the "View Only" mode to the "Edit" mode.

2.3.2 *Commenting System*

During the annotation process, an annotator could have concerns on issues like the image quality and the placement of certain annotations. We embed a commenting system into the program to

document the annotator's thoughts. Panel 6 of Figure 2.4 gives the commenting system's user interface. For each image, comments can be entered in the text box which are saved to the result file after the annotator clicks "submit comments." The commenting system allows different annotators to comment on the same image. Figure 2.11 shows how such comments are displayed.

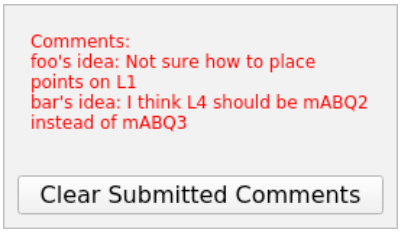


Figure 2.11. An example of comments displayed in the comment panel for an image.

2.3.3 *Automatic Window and Level Adjustment*

To reduce an annotator's efforts of adjusting the window and level, we embedded an automatic window and level adjustment function based on contrast stretching [75]. Given an image, let D denote the image's bit depth. $2^D - 1$ is the maximum intensity level the image can reach. A patch at the center of the image is extracted. The patch's width and height are 1/4 of the image's width and height, respectively. Let p_{max} and p_{min} denote the maximum and minimum intensity levels of the patch, respectively. In the regions of interest on the image, we regard most pixels' intensity levels to be within the range of p_{min} to p_{max}. For each image pixel, we map its intensity level I to

$$I' = (2^D - 1)(I - p_{min})/(p_{max} - p_{min}).$$

This horizontally stretches the image's histogram, mapping p_{min} and p_{max} in the original histogram to zero and $2^D - 1$ in the transformed histogram, respectively. After doing the mapping, some pixels' intensity levels could become $> 2^D - 1$ or < 0. For each intensity level $> 2^D - 1$, we set it to $2^D - 1$. For each intensity level < 0, we set it to 0.

2.3.4 *Splitting an Image Set and Merging Result Files*

Multiple annotators can use our program to collaboratively annotate a set of images in the following way:

1) Subsets of the image set are made, one per annotator. Each subset of images is put into a separate folder.
2) For each annotator, a copy of the program and the folder containing the assigned images is moved into another folder termed the annotation folder. The annotation folder is transferred to one of the following locations:
    a. A server accessible by all of the annotators: Each annotator's folder goes into the corresponding home folder. The annotator then logs into their account and uses the program to annotate the images.
    b. When each annotator uses their own computer: Electronically send (e.g., SCP, secure FTP, and website download) the annotation folder and the instructions for installing the program on their own computer.
    c. In a cloud environment (e.g., Amazon Web Services, Google Cloud, and Microsoft Azure): The annotation folder is copied to the annotator's individual cloud instance. The annotator could use Virtual Network Computing (VNC), which allows him/her to access the GUI of their cloud instance.
3) Each annotator runs DicomAnnotator and annotates the assigned images. If the annotator uses a Linux system, there is a Bash script to start DicomAnnotator with an associated environment containing all the necessary dependencies. This gives Linux novices an easy way to run DicomAnnotator without having to manage the Python environment and type complex Linux commands.

4) After the images are annotated, the administrator of the annotation team can obtain and merge the annotators' result files.

The program offers the functions of splitting the image set into multiple subsets and merging the result files from multiple annotators.

2.4 USABILITY EVALUATION

2.4.1 *Users' Backgrounds*

To understand the usability of DicomAnnotator, we conducted a usability evaluation on six users of various backgrounds. As Table 2.2 shows, half of the users were neuroradiologists. The other half were undergraduate and graduate students. These six users had a wide variety of familiarity with medical imaging: from no experience to several decades of experience. Before doing the annotation, the neuroradiologists had never seen the program. The undergraduate and graduate students had used the program to view a small number of images. The number of cases that each user reviewed with the program was recorded.

Table 2.2. Description of each user in the usability evaluation listing their occupation, medical imaging experience, and the number of images each annotated.

User ID	**Background**	**Experience with medical imaging (years)**	**Number of images annotated**
1	Neuroradiologist	5	9
2	Neuroradiologist	7	9
3	Neuroradiologist	31	9
4	Graduate student	0	9
5	Undergraduate student	0	60
6	Undergraduate student	0	60

The literature suggests using six users can identify most of the usability problems. Nielsen [59] proposed an empirical formula $N(1-(1-L)^m)$ to estimate the number of usability problems

that a group of users can find collectively. Here, *N* is the total number of usability problems. *L* is the percentage of usability problems a user can find on average. A typical value of *L* is 31%. *m* is the number of users. This formula indicates that six users can identify ~89% of the usability problems.

2.4.2 *Process of Usability Evaluation*

As Figure 2.12 shows, the usability evaluation process had two components: an annotation session followed by a survey session. In the annotation session, the six users were asked to annotate spine radiographs. DicomAnnotator's log file was used to analyze the program's usability. In the survey session, we did a web-based survey on these users.

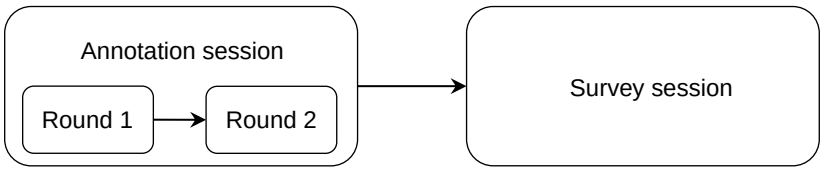


Figure 2.12. The usability evaluation process.

The program and radiograph DICOM files were used on a server running the Ubuntu operating system (16.04.6 LTS). From their personal computer, each user remotely logged into the server via a VNC client to do the annotation tasks. Before the annotation session, we created a user account for each user on the server and gave each user a Bash script to run the program.

During the annotation session, the users performed several annotation tasks including: 1) assigning an osteoporosis category to the whole image; 2) identifying the four corners of each visible thoracic/lumbar vertebral body; 3) determining each vertebral body's anatomic level; 4) identifying any overlying artificial structure; 5) using the mABQ criteria to score each vertebral body [73]; 6) identifying unreadable radiographs due to the wrong study type (cervical spine, chest,

abdominal, or pelvic radiographs), a non-lateral image, or low image quality; and 7) identifying incorrect orientation of the image and flipping it as needed to ensure the patient faces to the left of the screen.

As Figure 2.12 shows, the annotation session included two rounds. The first round used nine radiographs. After being introduced to the program, each of the three radiologists and the graduate student did all of the tasks on the nine radiographs. The second round used two different sets of radiographs: the first with 20 radiographs and the second with 40 radiographs. After being oriented to the program and some basic spine anatomy, two undergraduate students with no prior experience in medical imaging did tasks 2, 3, 4, 6, and 7. Each of these two students practiced on the first set of radiographs and then formally annotated the second set of radiographs. In each of the two rounds, each user was given two weeks to do the assigned tasks at their own pace. The data logged by the program were used to estimate the average amount of time the user spent on a radiograph. In the second round, the 20 radiographs used for practice were unused for estimating the average amount of time the user spent on a radiograph.

In the survey session, we used Google Forms to do a web-based survey to understand the users' thoughts on the program. We used the usability factors defined in DeVito Dabbs *et al*. [76] to design our survey questions. The usability factors and the survey questions are listed in Table 2.3. Each of the first seven questions measures a distinct usability factor. The last open-ended question is used to gather additional user feedback. In Question 4, if a user chose one or more errors, we asked the user to also list the errors in a text box.

Table 2.3. These survey questions were used to determine the usability of DicomAnnotator across a variety of factors.

Sequence number of the question	Usability factor	Question
1	Learnability	Initially, how easy was it to learn to use the program and its functions? Ratings are on a 1-5 scale with anchors of difficult/easy.
2	Effectiveness	Does the program include all of the functions you need for completing your image annotation task? Ratings are on a 1-5 scale with anchors of not at all/everything I needed.
3	Efficiency	Did the program help you annotate the images efficiently? Ratings are on a 1-5 scale with anchors of inefficient/efficient.
4	Errors	When using the program, how many times did you encounter severe errors such as software crash, software having no response, and annotation not stored? The choices for the response are 0, 1-3, 4-10, 11-20, and 20+.
5	Flexibility	Does the program give enough shortcuts and ways of access (e.g., window/level, pan, and navigating images) for the annotation task? Ratings are on a 1-5 scale with anchors of cumbersome or hard to use functions/fast and easy access to all of the functions.
6	Memorability	Are the program's features and functions easy to remember? Ratings are on a 1-5 scale with anchors of hard/easy.
7	User satisfaction	Overall, are you satisfied with the program? Ratings are on a 1-5 scale with anchors of unsatisfied/satisfied.
8		Please provide comments, thoughts, or features that you wish DicomAnnotator to have, if any, in the text box below.

2.4.3 *Results of Usability Evaluation*

Average amount of time to annotate a radiograph

Table 2.4 lists each user's average amount of time to annotate a radiograph, as well as the mean and the standard deviation of these numbers in each round in the annotation session. The undergraduate students (users 5 and 6) did only some, but not all of the annotation tasks assigned to the neuroradiologists (users 1-3) and the graduate student (user 4). As tasks take different amounts of time to complete, there is no basis to directly compare the average amount of time the

undergraduate students spent on annotating a radiograph with that of the neuroradiologists and the graduate student.

Table 2.4. A summary of each user's average amount of time needed to annotate a radiograph.

User ID	Average amount of time to annotate a radiograph (seconds)	Mean (standard deviation) of the average amount of time needed to annotate a radiograph (seconds)
1	243.1	284.1 (43.7)
2	275.4	
3	345.9	
4	271.8	
5	91.8	82.8 (12.7)
6	73.8	

Survey results

For each round in the annotation session and each of the survey questions 1, 2, 3, 5, 6, and 7, Table 2.5 shows the mean and the standard deviation of the ratings the users gave in their responses to the question, as well as the *p*-value of the *t*-test that checks whether these two rounds have the same mean of the ratings. The survey questions used to assess the usability factors are listed in Table 2.3. Note that results of survey question 4 are not included in Table 2.5 because survey question 4 uses a list of ranges which are difficult to summarize with a single mean and standard deviation.

In Question 4, five users reported encountering no severe error in the program. One user in the second round of the annotation session reported running into severe errors 4-10 times. In particular, the user reported errors running the Bash script to start the program, and said that every time they was able to get around by copying the commands from the Bash script to the terminal and running them one by one to start the program. Thus, it is likely that either that copy of the

Bash starting script contained some errors or the user used the Bash script incorrectly, rather than the errors being attributable to DicomAnnotator. This behavior has not been reproduced and no other users have since complained of this problem.

Table 2.5. A summary of the ratings for the survey questions in the first round and second round of the annotation session, and the *p*-values to describe the difference in the mean of the ratings between these two rounds.

Sequence number of the question	**Usability factor**	**Mean (standard deviation) of the ratings given by the users in the first round of the annotation session**	**Mean (standard deviation) of the ratings given by the users in the second round of the annotation session**	***p*-value**
1	Learnability	4.50 (0.58)	3.00 (1.41)	0.39
2	Effectiveness	5.00 (0.00)	4.50 (0.71)	0.50
3	Efficiency	4.75 (0.50)	3.50 (0.71)	0.15
5	Flexibility	4.50 (0.58)	4.00 (1.41)	0.71
6	Memorability	4.75 (0.50)	4.00 (1.41)	0.60
7	User satisfaction	4.75 (0.50)	4.00 (1.41)	0.60

The users' comments to Question 8 include: 1) "This is an excellent piece of software that is intuitive and remarkably stable;" 2) "Windowing/leveling remains somewhat challenging;" 3) "I wish that there was a brightness/contrast bar to adjust the images instead of having to press down on shift and move your mouse a certain way, because it's not very straightforward to know which way to shift the mouse at which angle to adjust the brightness." The first two comments came from the users in the first round of the annotation session. The third comment came from a user in the second round of the annotation session.

2.5 DISCUSSION

Our annotation program fulfills the requirements mentioned in Section 2.1 and offers several user-friendly features. The usability evaluation's annotation session showed that without previously

using the program, users of various backgrounds can use our program to quickly annotate medical images with multiple types of labels. This demonstrates that the program is easy to learn and efficient to use. The survey results showed that most users thought our program is easy to learn, effective, efficient to use, and flexible. They agreed the program's features and functions are easy to remember. Overall, they were satisfied with our program. One user reported running into severe errors. Yet, after working with the user, we found the errors are likely not due to the program. As there are no contradictory user complaints, we consider the program to be stable.

Based on the users' feedback, we optimized the program. Two users requested additional options for controlling window and level settings. One of these two users suggested brightness/contrast bars. Yet, after discussing with the other users, we reached the consensus that using brightness/contrast bars is inefficient. If the brightness/contrast bars were used, an annotator would need to frequently move their mouse cursor between the brightness/contrast bars and the program's annotation area (the main page's canvas or annotation table). To avoid this inefficiency, we give the user multiple ways to adjust the window and level: 1) hold down the "shift" key and move the mouse horizontally and vertically to adjust the window and level, respectively; and 2) on the keyboard, press "a" or "d" to increment or decrement the window setting and "w" or "s" to increment or decrement the level setting.

There are several interesting areas for future work. The program is currently optimized for displaying single images. The tasks for annotating video or stacks of cross-sectional data could require other types of annotations, like assigning labels on the entire video or stack of images from a cross-sectional series. This would require new functionality to be built into the program. Moreover, some annotation tasks could require a specific shape or aspect ratio of the bounding

polygon like a rectangle. New functions for constraining the bounding polygon's shape can be built into the program.

2.6 CONCLUSION

Conducting medical image classification, image segmentation, and object detection usually requires many annotated images. To reduce the burden of manual annotation, we designed DicomAnnotator, a DICOM image annotation program. It integrates multiple functional modules to meet several annotation requirements and provides four ancillary user-friendly features. The program is easy to learn, is efficient to use, and allows annotators to quickly make several types of annotations on a large set of DICOM images.

Chapter 3. DEEP LEARNING CLASSIFICATION OF SPINAL OCFS ON RADIOGRAPHS

Osteoporosis is a debilitating disease [1-6]. Spinal OCFs can serve as an early biomarker for osteoporosis but are often subtle, incidental, and under-reported. To ensure early diagnosis and treatment of osteoporosis, we aim to build a deep learning vertebral body classifier for OCFs as a critical component of our future automated opportunistic screening tool.

We retrospectively assemble two radiograph datasets containing lateral thoracic and lumbar spine radiographs: the UW dataset [37] and the MrOS dataset [35, 36]. Using both datasets, five deep learning algorithms were trained to classify each individual vertebral body of spine radiographs. Classification performance was compared for the trained models using multiple metrics including the AUC-ROC, AUC-PR, sensitivity, specificity, and PPV.

Our best model achieved an AUC-ROC of 0.948 and 0.936 on the UW dataset's test set and the MrOS dataset's test set, respectively. It achieved an AUC-PR of 0.730 and 0.811. After setting the cutoff threshold to prioritize PPV, this model achieved a sensitivity of 54.5% and 47.8%, a specificity of 99.7% and 99.6%, and a PPV of 89.8% and 94.8%.

Our model achieved an AUC-ROC > 0.90 and AUC-PR > 0.70 on both datasets. This testing shows some generalizability to clinical data and a suitable performance for our future opportunistic osteoporosis screening tool.

The rest of the chapter is organized as follows. Section 3.1 briefly reviews the background and objectives of this study. More details of the background have been given in Section 1.1. Section 3.2 introduces the two radiograph datasets. Section 3.3 describes the data pre-processing steps. Section 3.4 describes the details of model training. Section 3.5 describes the details of model evaluation and reports the evaluation results. Section 3.6 discusses our observations to the model

evaluation results, the related work, the limitations of our model, and the future work. Section 3.7 concludes this chapter.

3.1 INTRODUCTION

Osteoporosis is a debilitating disease [1-6]. Osteoporosis screening is evidence-based and is endorsed by many organizations, including the US Preventive Services Task Force, but remains underutilized [7-9]. Opportunistic osteoporosis screening, which uses pre-existing imaging, can complement current osteoporosis screening methods while introducing minimum extra cost. Since radiography is a ubiquitous imaging modality [30], using radiographs to conduct opportunistic osteoporosis screening could potentially reach a broader patient population. Spinal OCFs can serve as an osteoporosis biomarker and are often incidental on chest or abdominal images and frequently under-reported, resulting in under-diagnosis and under-treatment [33]. Applying automated opportunistic OCF screening to existing radiographs could result in earlier and more extensive osteoporosis identification and treatment. Multiple studies [25-29] have attempted to automatically detect OCFs using radiographs. However, these studies had limitations including single center data leading to possible overfitting [25-28] and unclear dataset construction processes [29].

We ultimately aim to build an automated opportunistic screening tool to detect OCFs using the radiographs that contain lumbar and/or thoracic vertebrae. Our opportunistic screening tool has three primary sequential components (see Figure 1.1): 1) image segmentation and extraction of vertebral bodies; 2) a binary OCF classifier predicting whether each vertebral body has a moderate to severe OCF or not; and 3) a subject-level classifier integrating the OCF predictions of all vertebral bodies with additional structured data to determine this subject's OCF status. Considering a screening tool for large volumes of studies, a tool with too many false positives could unduly

burden the health care system. Thus, we prioritized PPV and specificity of the model rather than sensitivity.

In this dissertation, we focus on the second component, the binary OCF classifier (see Figure 1.1), whose inputs are vertebral patches extracted from radiographs. Recall that the first component is used to automatically extract the vertebral patches. Since the first component is a distinct body of work [34] and is not our focus in this dissertation, we extracted each vertebral body using manually annotated corner points.

We used two spine radiograph datasets with multicenter data: 1) the UW dataset assembled from multiple clinical sites across the UW Medical Center [37] and 2) the MrOS dataset [35, 36]. These two datasets contain lateral thoracic and lumbar spine radiographs. To detect OCF on each vertebral patch, we used deep learning, the state-of-the-art technique for image classification. Our objective is to build a performant and generalizable OCF classifier with an AUC-PR > 0.70 and an AUC-ROC > 0.90 on the multicenter data mentioned above.

3.2 DATASETS

We obtained two datasets containing lateral thoracic and lumbar spine radiographs: the clinically derived UW dataset [37] and the research MrOS dataset [35, 36]. The UW dataset contains clinical data acquired in varied clinical settings for diagnostic purposes. The MrOS dataset was generated for research and includes only male subjects, and thus has lower diversity than the UW dataset. To make the deep learning models performant on clinical data, we typically used the UW dataset to fine-tune the models. Both datasets were used to test the models.

Retrieval of the UW dataset was covered under the local retrospective institutional review board (IRB) for Diagnosis Radiology Images Deep Learning Project with a waiver of informed

consent. For the MrOS dataset, at each medical center, a local IRB approved the MrOS study. All MrOS participants gave written informed consent at the time of the study.

We describe the UW dataset and the MrOS dataset in Sections 3.2.1 and 3.2.2, respectively. Section 3.2.3 provides a summary of the datasets.

3.2.1 *UW dataset*

UW dataset's construction

The UW dataset contains clinical data acquired in varied clinical settings for diagnostic purposes. The spine radiographs in this retrospective dataset were acquired from 2000 to 2017 at multiple clinical sites (inpatient, outpatient, and emergency) across the UW medical center. The mean ages (±standard deviation) of female and male subjects were 75±8 years and 75±9 years, respectively.

The UW dataset was constructed as Figure 3.1 shows. zVision (Intelerad; Montreal, Canada), a radiology information search tool, queried the radiology information system (RIS) to identify subjects and exams fitting the inclusion criteria. A natural language processing (NLP) system called LireNLPSystem [77] analyzed each exam's radiology report to roughly determine whether it described a fracture. The NLP result for each exam's radiology note served as a weak label for this exam. These weak labels could help roughly balance the training set. Radiographs of the subjects that satisfied the inclusion criteria were retrieved from the picture archiving and communication system (PACS). From the retrieved radiographs, we randomly selected 1,200 subjects and a single radiograph of each subject to form the validation and test sets. From these 1,200 radiographs, 879 were randomly assigned to form the test set and the remaining 321 were assigned to the validation set. To avoid overlap between the training set and the other two sets, the radiographs in the training set were sampled from the 13,667 radiographs excluding those of the

1,200 subjects that had been selected for the validation and test sets. To form the training set, 778 radiographs were sampled. To improve the balance of the training set, 75% of the radiographs were randomly sampled from the ones labeled as “fracture” by NLP. The remaining radiographs were randomly sampled from the ones labeled as “no fracture” or “not mentioned” by NLP. Finally, the UW dataset was annotated. Further data pre-processing and augmentation steps (including other data balancing steps) are introduced in Section 3.3.

Two other researchers reviewed each radiograph to guarantee that they were de-identified and contained no protected health information. All radiographs were originally in the DICOM format. The DICOM tags, which could contain protected health information, were removed by converting the DICOM radiograph to a TIFF image.

UW dataset’s annotation

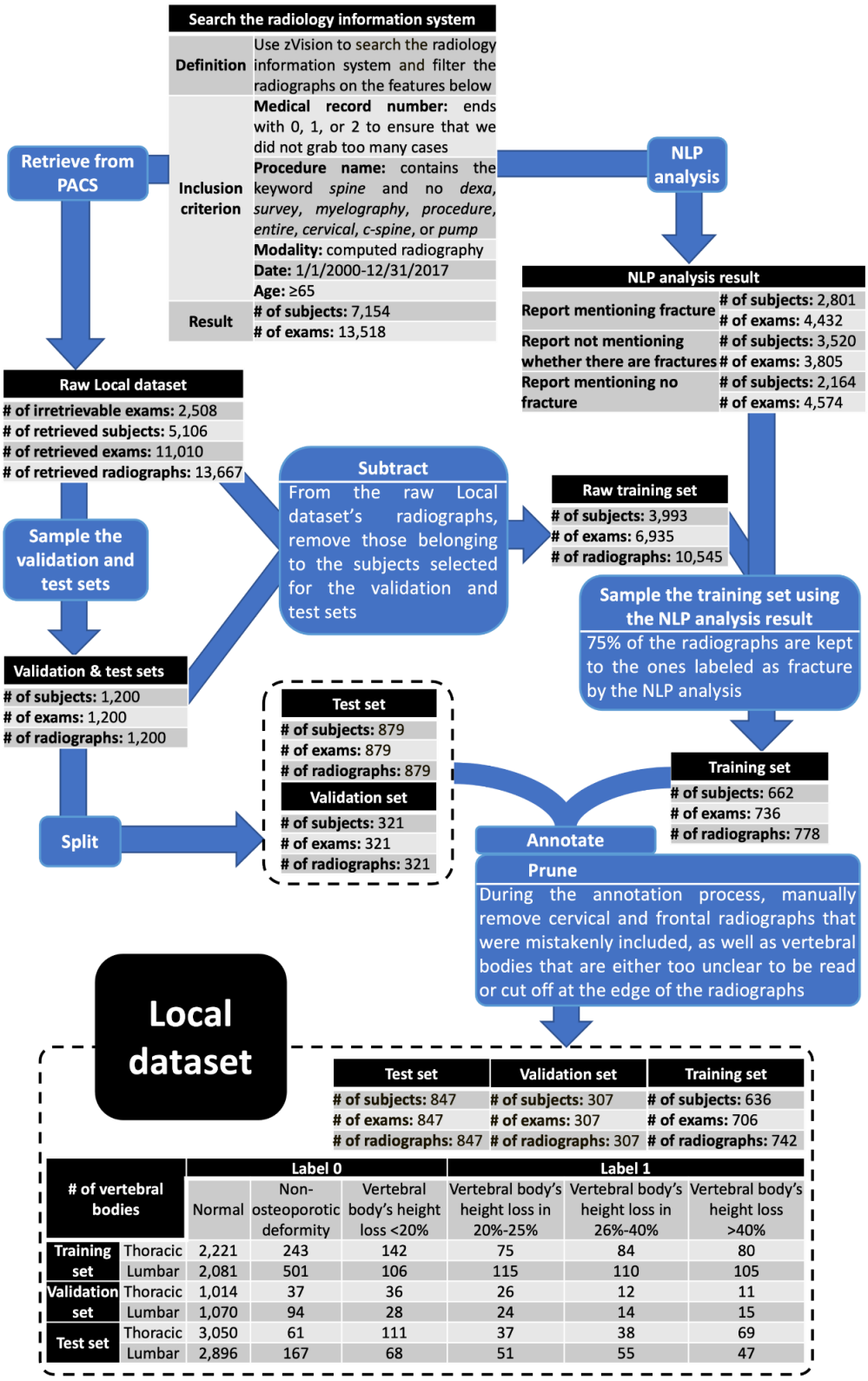
On each radiograph in the UW dataset, we annotated each vertebral body’s four corner points and severity of OCF using DicomAnnotator [51] (see Chapter 2). Multiple groups participated in the process of annotating the corner points of each vertebral body. The OCF severity of each vertebral body was annotated using the modified-2 algorithm-based qualitative (m2ABQ) criteria [78], a revised version of the mABQ criteria [73]. Five individuals annotated OCF severity of each vertebral body, including three faculty radiologists (27, 17, and 10 years of experience, respectively), one neuroradiology fellow (7 years of experience), and one biomedical informatics graduate student. This process consisted of 17 rounds. We randomly split the UW dataset into 17 subsets. In the first eight rounds, at least two individuals annotated each radiograph. For each of these first eight rounds, we computed Fleiss’ kappa and Cohen’s kappa to measure the inter-reader agreement, as well as held a consensus meeting to discuss the disputed annotations. In the last nine

rounds, each radiograph was annotated by one annotator. More details about the UW dataset annotation are presented in another paper [78].

Classification systems and radiologists struggle to accurately classify mild (or subtle) OCFs, which are often confounded by parallax artifact, remote traumatic injuries, and congenital variations [73]. Our future opportunistic screening tool is intended to complement the current clinical standard of care while introducing a minimum of extra cost. Including mild OCFs into our classification system could substantially increase false positives, which would cause more downstream cost. Our use case, to alert or not alert a provider or a radiologist to a potentially missed fracture, required a binary classification, defined as highly probable OCF vs normal/non-osteoporotic deformity/mild or questionable fracture. Therefore, we dichotomized the m2ABQ categories: “label 0” representing normal/non-osteoporotic deformity/mild or questionable fracture vs. “label 1” representing moderate or severe fracture.

UW dataset’s partitioning

The UW dataset was partitioned into the training, validation, and test sets. As shown in Figure 3.1, the training set was balanced for better model training. In contrast, we kept the class distributions of the validation and test sets consistent with those in the original population. The number of subjects in each of the training and test sets was determined by striking the balance between obtaining a large set and reducing manual annotation time. A large set is more likely to contain diverse data. Thus, a large training set can reduce model overfitting. A large test set can ensure accurate measures of model performance. However, since manual annotation is time-consuming, we could not wait to train and test our models after annotating a very large number of radiographs.



	Test set	Validation set	Training set
	# of subjects: 847 # of exams: 847 # of radiographs: 847	# of subjects: 307 # of exams: 307 # of radiographs: 307	# of subjects: 636 # of exams: 706 # of radiographs: 742

# of vertebral bodies		Label 0			Label 1		
		Normal	Non-osteoporotic deformity	Vertebral body's height loss <20%	Vertebral body's height loss in 20%-25%	Vertebral body's height loss in 26%-40%	Vertebral body's height loss >40%
Training set	Thoracic	2,221	243	142	75	84	80
	Lumbar	2,081	501	106	115	110	105
Validation set	Thoracic	1,014	37	36	26	12	11
	Lumbar	1,070	94	28	24	14	15
Test set	Thoracic	3,050	61	111	37	38	69
	Lumbar	2,896	167	68	51	55	47

Figure 3.1. Construction of the UW dataset and partitioning it into the training, validation, and test sets.

UW dataset's demographic information and other metadata

Table 3.6 shows the UW dataset's metadata, including age, sex, race, ethnicity, radiograph generation year, and X-ray system vendor. We show the metadata for the training, validation, and test sets of the UW dataset, as well as the entire UW dataset. The age data were retrieved from the RIS. The sex data were obtained from the DICOM metadata of the radiographs. The race and ethnicity data were retrieved from the electronic health record system. A subject could have multiple exams, which might not be from the same year. Consequently, multiple ages could be recorded for a subject. In each set, for every range of ages, we reported the number of recorded ages rather than the number of subjects. If a subject had multiple ages recorded, all of them were used to calculate the mean and the standard deviation.

Table 3.6. Metadata for the training, validation, and test sets of the UW dataset, as well as the entire UW dataset.

	Training set	Validation set	Test set	Entire UW dataset
	Number (percentage) of recorded ages			
Age at exam				
65-74	395 (53.2%)	181 (59.0%)	479 (56.6%)	1,055 (55.7%)
75-84	234 (31.6%)	84 (27.4%)	255 (30.1%)	573 (30.2%)
85-94	98 (13.2%)	32 (10.4%)	102 (12.0%)	232 (12.2%)
≥95	15 (2.0%)	10 (3.2%)	11 (1.3%)	36 (1.9%)
	Number			
Total recorded ages	742	307	847	1,896
	Mean ± standard deviation of ages in years			
Female	76 ± 9	75 ± 9	75 ± 8	75 ± 8
Male	75 ± 9	75 ± 9	75 ± 9	75 ± 9
All	75 ± 9	75 ± 9	75 ± 9	75 ± 9
	Number (percentage) of subjects			
Sex				
Female	339 (53.3%)	172 (56.0%)	467 (55.1%)	978 (54.6%)
Male	296 (46.5%)	135 (44.0%)	379 (44.8%)	810 (45.3%)
Not recorded	1 (0.2%)	0 (0%)	1 (0.1%)	2 (0.1%)

Continued on next page

Race				
American Indian and Alaska Native	2 (0.3%)	2 (0.7%)	6 (0.7%)	10 (0.6%)
Asian	68 (10.7%)	37 (12.0%)	72 (8.5%)	177 (9.9%)
Black or African American	39 (6.2%)	20 (6.5%)	51 (6.0%)	110 (6.1%)
Native Hawaiian and Other Pacific Islander	2 (0.3%)	1 (0.3%)	3 (0.4%)	6 (0.3%)
White	474 (74.5%)	220 (71.7%)	654 (77.2%)	1348 (75.3%)
Multiple races	49 (7.7%)	25 (8.1%)	57 (6.7%)	131 (7.3%)
Not recorded	2 (0.3%)	2 (0.7%)	4 (0.5%)	8 (0.4%)
Ethnicity				
Hispanic or Latino	9 (1.4%)	5 (1.6%)	16 (1.9%)	30 (1.7%)
Not Hispanic or Latino	189 (29.7%)	138 (45.0%)	358 (42.3%)	685 (38.3%)
Not recorded	438 (68.9%)	164 (53.4%)	473 (55.8%)	1,075 (60.0%)
		Number		
Total subjects	636	307	847	1,790
		Number (percentage) of radiographs		
Radiograph generation year				
2000-2005	127 (17.1%)	49 (15.9%)	113 (13.3%)	289 (15.2%)
2006-2011	354 (47.7%)	135 (44.0%)	405 (47.8%)	894 (47.2%)
2012-2017	261 (35.2%)	123 (40.1%)	329 (38.9%)	713 (37.6%)
X-ray machine vendor				
Canon	5 (0.7%)	0 (0%)	5 (0.6%)	10 (0.5%)
DeJarnette Research Systems	48 (6.5%)	21 (6.8%)	48 (5.7%)	117 (6.2%)
Fujifilm	378 (50.9%)	157 (51.2%)	427 (50.4%)	962 (50.8%)
General Electric	202 (27.2%)	74 (24.1%)	232 (27.3%)	508 (26.8%)
Philips	104 (14.0%)	50 (16.3%)	127 (15.0%)	281 (14.8%)
Hybrid General Electric and Fujifilm	5 (0.7%)	5 (1.6%)	8 (1.0%)	18 (0.9%)
		Number		
Total radiographs	742	307	847	1,896

3.2.2 *MrOS dataset*

MrOS dataset's construction

The MrOS dataset was previously described in Orwoll *et al*. [35]. The MrOS study collected clinical and laboratory imaging data from six US academic medical centers in Birmingham AL, Minneapolis MN, Palo Alto CA, Pittsburgh PA, Portland OR, and San Diego CA. This study recruited 5,994 males aged 65 and older. The data were collected at the initial visit (Visit 1) and the follow-up visit (Visit 2) average 4.5 years later.

We obtained a de-identified copy of this dataset under a data use agreement with the San Francisco Coordinating Center. After cleaning the dataset, we finally obtained 4,461 subjects with 8,915 radiographs from Visit 1 and 3,309 subjects with 6,609 radiographs from Visit 2. From both visits, we obtained 100,409 vertebral bodies consisting of 69,453 lumbar vertebral bodies and 30,956 thoracic vertebral bodies.

MrOS dataset's original annotation

Cawthon *et al*. [79] had previously annotated the MrOS dataset to 1) identify the margin of each vertebral body and 2) assign it an OCF label based on a modification [79] of the Genant semi-quantitative (mSQ) criteria [71]. To outline each vertebral body in each radiograph, four corners of this vertebral body and two midpoints of the superior and inferior endplates were pinpointed. The numbers of thoracic and lumbar vertebral bodies in each mSQ class are listed in another paper [63].

To determine OCFs, the mSQ criteria require the presence of endplate depression, making these criteria closer to the mABQ criteria [73]. To adapt to our binary OCF classification, the mSQ categories were simplified into two classes ("label 0" representing moderate or severe fracture vs. "label 1" representing normal/trace/mild fracture) [63]. This is similar to the m2ABQ simplification discussed in Section 3.2.1.

MrOS dataset's partitioning

The MrOS dataset was partitioned into the training, validation, and test sets by subject. The radiographs from one subject only appear in one of the three sets. We randomly selected 76.5%, 8.5%, and 15% of the subjects to form the training, validation, and test sets, respectively (see Figure 3.2). The percentage used to form the validation instances was set smaller than usual (e.g.,

10% or 20%) due to the class imbalance of the dataset. For effective training, the training set needed subsampling to correct the class imbalance between label 1 and label 0 (see Figure 3.2). To preserve the class distribution of the original population and to most closely evaluate real world performance, no corrections of the imbalance in the validation and test sets were performed. A smaller validation set allowed for a larger training set, even after downsampling the majority class in the training set. In the training set, data instances of label 0 were randomly sampled at a ratio of 2.5:1 (label 0 : label 1) to better balance the two classes. The ratio of 2.5:1 was determined in an earlier experiment, in which different ratios were tried and we chose the ratio to maximize the AUC-PR on the validation set.

MrOS dataset's extra annotation using the m2ABQ criteria

For a model built using the UW dataset, we can test this model's generalizability using the MrOS dataset. Recall that the UW dataset was annotated using the m2ABQ criteria (see Section 3.2.1). To test the generalizability of the models built using the UW dataset, we further used the m2ABQ criteria to annotate some radiographs in the MrOS dataset. From the MrOS dataset's test set, we randomly selected 122 radiographs containing 844 vertebral bodies. Each radiograph was assigned an m2ABQ label using DicomAnnotator [51] (see Chapter 2).

MrOS dataset's demographic information

Table 3.7 shows the characteristics of the subjects in the entire, training, validation, and test sets of the MrOS dataset. Recall that the MrOS dataset is imbalanced with far more vertebral bodies with label 0 than vertebral bodies with label 1. To balance the training set, vertebral bodies with label 0 were downsampled. Both the characteristics of the subjects in the training set before downsampling and those in the training set after downsampling are shown in Table 3.7. The mean

(standard deviation) of the ages and body mass indices were recorded at the baseline (Visit 1) and the follow-up (Visit 2) visits. Race and ethnicity and the total number of subjects are also provided for each set.

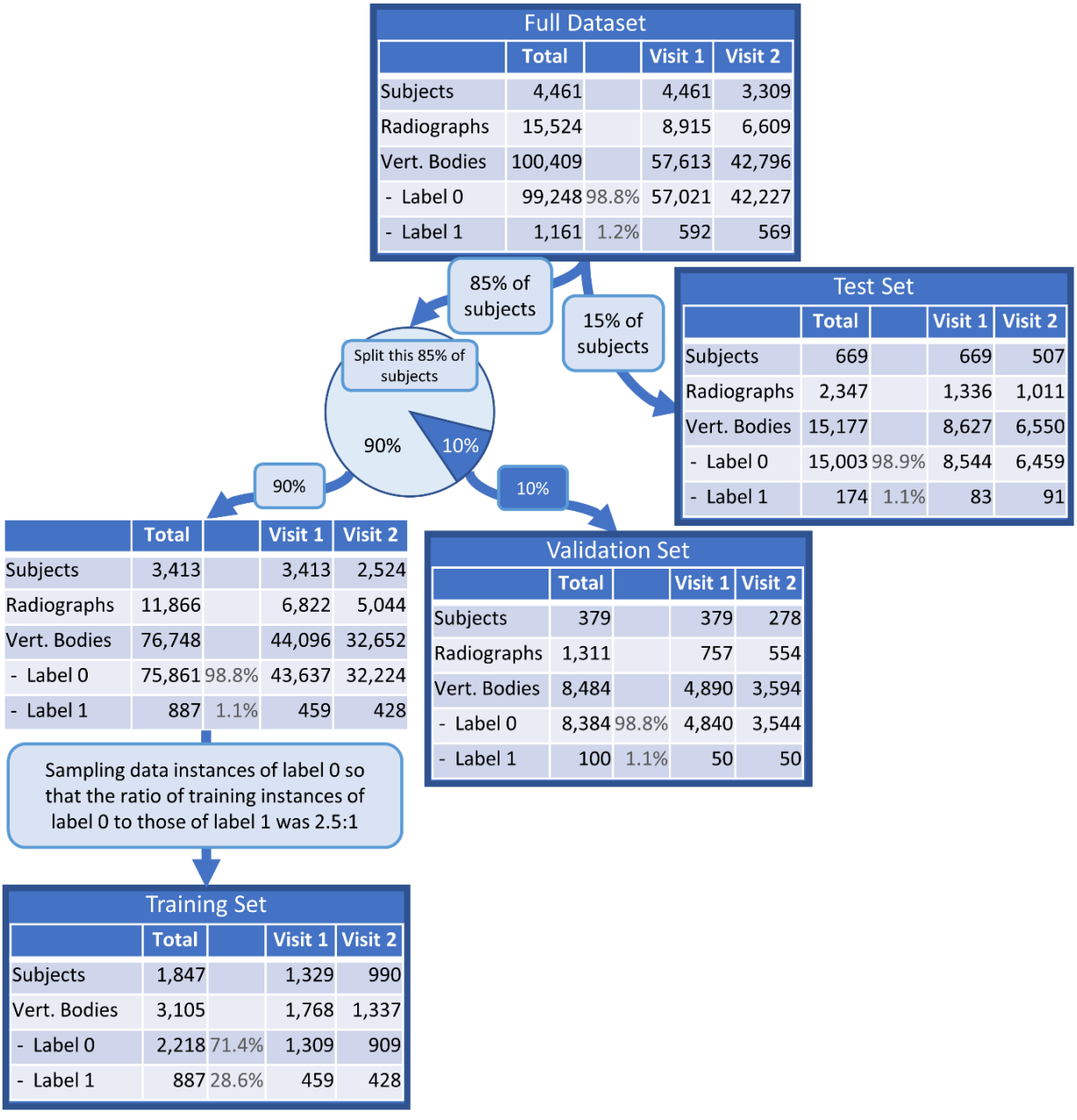


Figure 3.2. The MrOS dataset was divided into the test, validation, and training sets by subject.

Table 3.7. Demographic information of the subjects in each of the entire, training, validation, and test sets from the MrOS dataset.

	Training set before downsampling the data instances with label 0	Training set after downsampling the data instances with label 0	Validation set	Test set	Entire dataset
		Mean ± standard deviation			
Age at Visit 1	73.7±5.9	73.7±5.8	74.1±6.2	73.5±5.7	73.7±5.9
Body mass index at Visit 1	27.8±3.9	27.9±4.0	27.2±3.5	27.6±3.5	27.4±3.8
Age at Visit 2	77.8±5.6	77.9±5.6	77.9±5.6	77.5±5.4	77.7±5.6
Body mass index at Visit 2	27.3±3.9	27.3±4.0	27.4±4.0	27.3±3.9	27.3±3.9
		Percentage			
Race/ethnicity					
American Indian or Alaska Native	0.8%	0.7%	1.8%	1.2%	0.9%
Asian	3.2%	2.5%	3.1%	3.7%	3.2%
Black or African American	4.2%	3.0%	5.4%	3.1%	4.2%
Hispanic or Latino	2.0%	2.0%	2.8%	2.2%	2.1%
Native Hawaiian or Other Pacific Islander	0.2%	0.3%	0.8%	0.1%	0.2%
White	89.6%	91.5%	86.1%	89.7%	89.4%
		Number			
Total subjects	5,016	1,874	392	681	6,089

3.2.3 *Summary*

We obtained two datasets containing lateral thoracic and lumbar spine radiographs: the clinically derived UW dataset and the research MrOS dataset. The UW dataset contains clinical data acquired in varied clinical settings for diagnostic purposes. The MrOS dataset was generated for research and includes only male subjects, and thus has lower diversity than the UW dataset. To make the deep learning models performant on clinical data, we typically used the UW dataset to fine-tune the models. Both datasets were used to test the models.

Each vertebral body in each dataset has four corner points pinpointed, which were used to manually extract the vertebral patch (see Section 3.3). The UW dataset was annotated using the m2ABQ criteria. The MrOS dataset was originally annotated using the mSQ criteria. We further

randomly sampled some radiographs from the MrOS dataset's test set and annotated them using the m2ABQ criteria. Table 3.8 shows the number of vertebral bodies for each (dataset, OCF classification criteria) combination. In the rest of this chapter, each of these combinations is denoted by "dataset-classification criteria." For example, MrOS-m2ABQ denotes the dataset whose data are from the MrOS dataset and are annotated using the m2ABQ criteria.

Table 3.8. The number of vertebral bodies for each (dataset, OCF classification criteria) combination.

	UW dataset				**MrOS dataset**			
	Training set	**Validation set**	**Test set**	**Total**	**Training set**	**Validation set**	**Test set**	**Total**
m2ABQ	5,968	2,394	6,688	15,050	0	0	844	844
mSQ	NA				76,748	8,484	15,177	100,409

3.3 Data Pre-processing and Augmentation

This section presents the image pre-processing and augmentation steps. First, we wanted to extract the vertebral patches from the radiographs. Each input data instance for our OCF classifier was a vertebral patch. Since the automated image segmentation tool is under development, each vertebral patch in this research was extracted using the manually annotated corner points of the corresponding vertebral body. Second, we attempted to control the heterogeneity among the vertebral patches to a moderate range. Excessive heterogeneity in the dataset can confound deep learning models when extracting relevant features, while too little heterogeneity could result in poor generalizability of the trained model. As Figure 3.3 shows, the general steps of image pre-processing and augmentation are listed in the following:

1) Flip the radiograph horizontally to conform to the convention that the subject faces left.
2) Form two coordinate axes with the x-axis bisecting the angle between the two diagonals connecting the four corner points.

3) Obtain the smallest square that bounds the four corner points with edges parallel to the corresponding coordinate axes.
4) Expand the square from its center to increase the area by four times, preventing cutoff of part of the vertebral body and providing surrounding image context.
5) Augment the vertebral patch by scaling, rotating, and translating the square.
6) Extract the square as a vertebral patch.
7) Invert the grayscale if the bones are darker than the background.
8) Augment the vertebral patch by changing the contrast and brightness and adding Gaussian noise to the vertebral patch.
9) Resize the vertebral patch to 224×224 pixels and normalize each pixel value by subtracting the mean and then dividing by the standard deviation [52].

Image augmentation is a ubiquitous approach to increase the trained model's performance by creating subtly modified data instances for the model to be trained on. By definition, image augmentation was applied to only the training set. Among all steps shown in Figure 3.3, Steps 5 and 8 are image augmentation procedures applied to only the training set, while the other steps are image pre-processing steps applied to each of the training, validation, and test sets. Our code for image pre-processing and augmentation was uploaded to https://github.com/UW-CLEAR-Center/Preprocessing_for_Spinal_OCF_Detection_Multi_Datasets. In the following, we describe the details of these image pre-processing and augmentation steps.

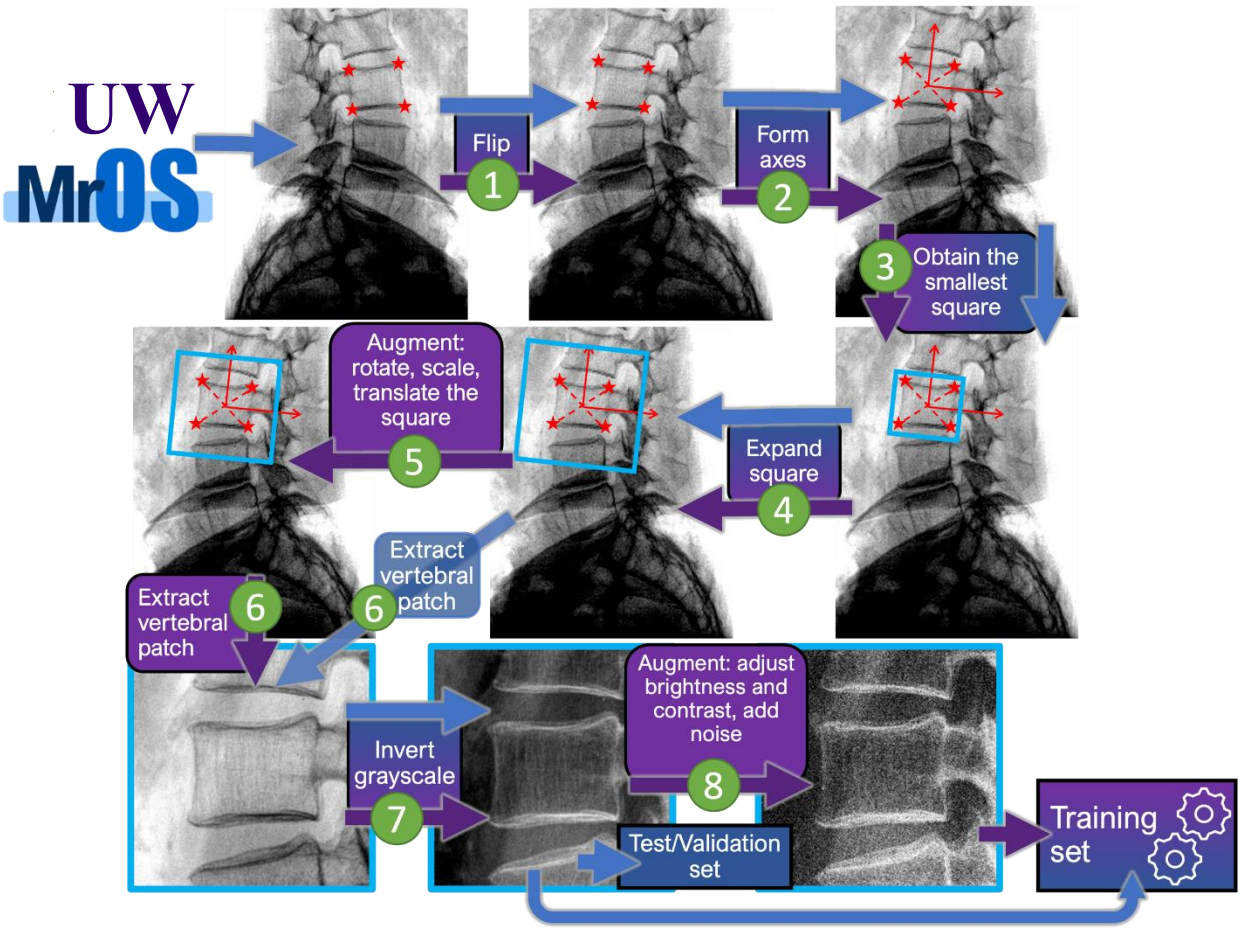


Figure 3.3. This process of generating a vertebral patch was performed for each vertebral body labeled in a radiograph using the four corner points indicated by red stars. The blue and purple arrows demonstrate the creation of the vertebral patches without and with the augmentation steps, respectively.

In each vertebral patch, we controlled the variation of three features: 1) the vertebral body's position; 2) the percentage of the vertebral patch's area occupied by the vertebral body; and 3) the vertebral body's tilt angle. The aspect ratio of each vertebral body was fixed after extracting it. Initially, horizontally flipping was performed if needed, to conform to the convention that the subject faces left (see Step 1 of Figure 3.3). To extract a vertebral body, the four manually annotated corner points were used to generate two diagonals. We built two coordinate axes with the x-axis bisecting the angle between the two diagonals connecting the four corner points (see

Step 2 of Figure 3.3). The angle between this bisector and the x-axis of the vertebral patch defined the vertebral body's tilt angle. To keep the extracted vertebral body's aspect ratio constant, we bounded the vertebral body by a square fulfilling two requirements: 1) one side of the square is perpendicular to the aforementioned bisector; and 2) the square is the smallest square with none of the four corner points lying outside of it. Requirement 1 guarantees that the vertebral body is not tilted inside the square. Requirement 2 assures that the vertebral body is at the center of the square. This smallest square cannot always bound the whole vertebral body because of osteophytes and parallax. To avoid accidental cropping of the vertebral body and to provide surrounding image context during the extraction step, we expanded this smallest square around its center to enlarge its area by four times. If the enlarged square exceeded the boundary of the spine image, we zero padded the excess area. This enlarged square served as the vertebral patch. In summary, the extraction process assures that the vertebral body is positioned at the center of a patch, occupies one quarter of the patch, and is not tilted.

Affine transformation, a data augmentation method, was conducted during vertebral body extraction. For each vertebral patch, rotation, scaling, and translation were applied to the vertebral body sequentially. The requirements of the affine transformation are: 1) scale the vertebral body's area by s%, where s was randomly selected from the range [81, 121]; 2) rotate the vertebral body by a degree randomly selected from the range [-5°, 5°]; and 3) translate the vertebral body along the x-axis and y-axis, each by a distance equal to a value randomly and independently sampled from the range [-0.05, 0.05] × the length of vertebral patch's edge. Affine transformation of the vertebral body inside the vertebral patch is basically the same as that of the square on the spine image (Step 5 in Figure 3.3). To expand (or shrink) the vertebral body's area by s%, we shrank (or expanded) the square's area by $(1/s\%)\times 100\%$. To rotate the vertebral body by an angle, we rotated

the square by the same angle in the opposite direction. To translate the vertebral body by a distance, we translated the square by the same distance in the opposite direction. For each original square (the square after Step 4 in Figure 3.3), we conducted affine transformations on it four times to generate four augmented vertebral patches.

While the bone of the vertebral body is brighter than the background in most vertebral patches, "inverted patches" also exist, in which the bone of the vertebral body is darker than background. Figure 3.3 includes an inverted vertebral patch, which is shown to be inverted back in step 7 according to the convention used in this project (bone brighter than background and air). In our experiments, mixing these two types of vertebral patches in the dataset was one major negative factor on the deep learning model's performance [63]. Thus, it is necessary to invert the inverted patches to make the bone of the vertebral body in each vertebral patch brighter than the background. Note that all radiographs were originally stored in the DICOM format. One DICOM tag called Photometric Interpretation [80] can be used to determine whether each radiograph and its corresponding vertebral patches are inverted. If the value of Photometric Interpretation is MONOCHROME 1, the radiograph is inverted. Otherwise, if the value of Photometric Interpretation is MONOCHROME 2, the radiograph is not inverted.

Afterwards, we conducted two additional data augmentation steps on each vertebral patch generated by affine transformation: 1) adjust the contrast and brightness using the SigmoidContrast class in the imgaug package [81]; and 2) add Gaussian noise using the AdditiveGaussianNoise class in the imgaug package [81]. When adjusting the contrast and brightness, two parameters were required: gain and cutoff. Gain was randomly sampled from the range [4, 8]. Cutoff was randomly selected from the range [0.4, 0.6]. The standard deviation of the Gaussian noise was randomly sampled from the range [0, 39].

As a result, for each vertebral body in the training set, we obtained one original vertebral patch and four augmented vertebral patches. These five patches were used for model training. The validation and test sets only contain the original vertebral patches. Before feeding into the neural network, each vertebral patch was resized to 224×224 pixels and each pixel value was normalized by subtracting the mean and then dividing by the standard deviation of the vertebral patch [52].

3.4 Model Training

3.4.1 *Overview of Model Training*

We trained five deep learning algorithms (see Figure 3.4), including GoogLeNet [82], Inception-ResNet-v2 [83], EfficientNet-B1 [84], and two ensemble algorithms. Taking an individual vertebral patch as an input, each of the five deep learning algorithms was used to build models to classify the vertebral patch to have label 0 or label 1.

To train GoogLeNet, Inception-ResNet-v2, and EfficientNet-B1, transfer learning was used by pre-training a model on ImageNet [85] and fine-tuning the model on a target dataset. Besides this common transfer learning technique, we also built a model by first pre-training it on ImageNet, then tuning it on the MrOS-mSQ dataset, and finally fine-tuning it on the UW-m2ABQ dataset. Recall that the UW dataset contains clinical data, while the MrOS dataset was generated for research. To make the model performant on the clinical data, we finally fine-tuned each model on only the UW-m2ABQ dataset rather than the combination of both the UW-m2ABQ dataset and the MrOS-mSQ dataset. Since both the MrOS dataset and the UW dataset contain vertebral patches, a model tuned on the MrOS-mSQ dataset before finally fine-tuned on the UW-m2ABQ dataset can learn more relevant image features.

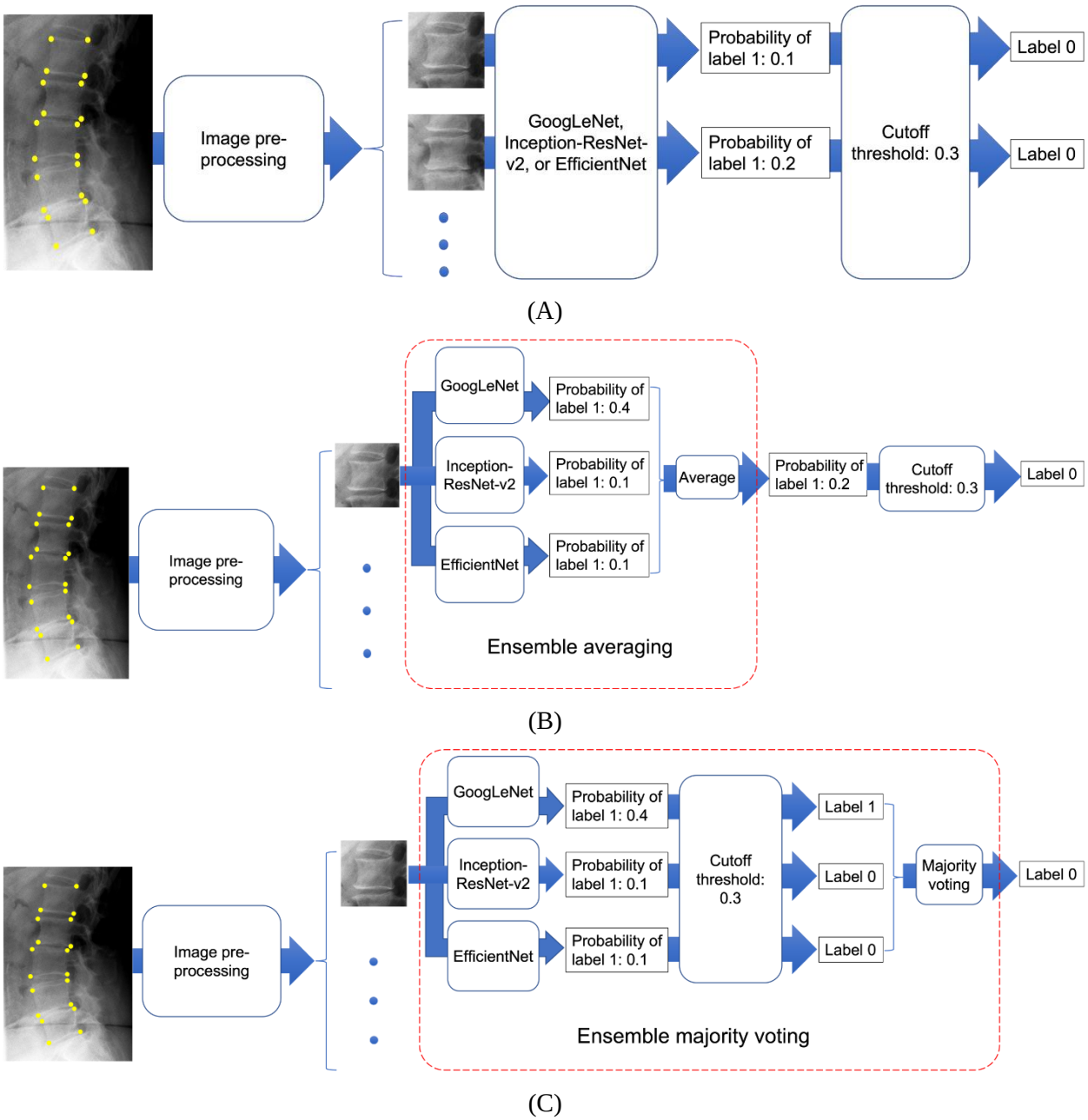


Figure 3.4. The flowchart of OCF classification using deep learning.
(A) The flowchart of OCF classification by GoogLeNet, Inception-ResNet-v2, or EfficientNet-B1. (B) The flowchart of OCF classification by ensemble averaging. (C) The flowchart of OCF classification by ensemble majority voting.

After training the models using the three individual algorithms mentioned above, two ensemble models were created using the ensemble averaging algorithm and the ensemble majority

voting algorithm (see Figure 3.4(B) and Figure 3.4(C)). Note that each of the three individual algorithms could output a probability that the vertebral patch should be classified to have label 1 (see Figure 3.4(A)). For ensemble averaging, we averaged the probabilities output by the three individual models. Then the classification result was obtained by comparing the average probability and a pre-set cutoff threshold. The classification result of ensemble majority voting was the majority classification result of the three individual models.

In summary, three deep learning models and two ensemble models were generated in each of the following three training tasks:

1) **Task 1:** Pre-train the model on ImageNet and fine-tune the model on the MrOS-mSQ dataset's training set (ImageNet → MrOS-mSQ).
2) **Task 2:** Pre-train the model on ImageNet and fine-tune the model on the UW-m2ABQ dataset's training set (ImageNet → UW-m2ABQ).
3) **Task 3:** The model tuned in Task 1 was further fine-tuned on the UW-m2ABQ dataset's training set (ImageNet → MrOS-mSQ → UW-m2ABQ).

In total, 15 models (5 models per task × 3 tasks) were built.

3.4.2 *Details of Model Training*

To build models for OCF classification, we used the five deep learning algorithms (see Figure 3.4), namely, GoogLeNet, Inception-ResNet-v2, EfficientNet-B1, and the two ensemble algorithms. These algorithms were implemented using Python 3.7.6, TensorFlow 2.4.1 [66], and TF-Slim 1.1.0 [86].

We chose each of these three individual algorithms because of the combination of the following two factors:

1) The performance of the algorithm in published benchmark analysis.

2) The number of parameters that the algorithm has. Considering that we have very limited data instances, too many parameters could cause overfitting.

In each of the three training tasks (see Section 3.4.1), we only needed to train GoogLeNet, Inception-ResNet-v2, and EfficientNet-B1. Given the classification results of these three models after training, the classification result of each ensemble model was computed (see Figure 3.4(B) and Figure 3.4(C)). Thus, in all three training tasks, we built 15 deep learning models, 9 of which through model training.

To boost the performance of the models, we applied transfer learning [52], which consists of the following steps: 1) pre-training a model on a large source dataset and 2) fine-tuning the weights of the model on the target dataset. ImageNet [85] is one of our source datasets. The TensorFlow Model Garden [87] provides GoogLeNet and Inception-ResNet-v2 models pre-trained on ImageNet. Another online open-source code [88] provides an EfficientNet-B1 model pretrained on ImageNet. Before fine-tuning a model that was pre-trained on ImageNet, the output layer of the corresponding neural network was adjusted for binary classification to fit our OCF classification. This modified output layer could not be initialized using the weights of the model pretrained on ImageNet. Instead, this output layer was initialized using He initialization [89]. When tuning each model that was pre-trained on ImageNet, we conducted the following two steps:

1) Freeze all layers except the output layer. Train the model with a fixed learning rate of 10^{-3}, a batch size of 20, adaptive moment estimation (Adam) optimization [90], and 15 epochs.
2) Keep freezing several layers close to the input layer and unfreeze the other layers. Use a smaller learning rate to continue to tune the model. The batch size was 20. Adam was used.

Some details of the second step are shown in the following:

1) The number of frozen layers was a hyper-parameter.

2) Learning rate decay was applied to the weights of the unfrozen layers. If the AUC-PR of the model evaluated after each epoch on the validation set did not increase in any of the subsequent two epochs, the learning rate was multiplied by a decay factor of < 1. Once the learning rate was decayed, the neural network with the weights leading to the best validation result ever obtained during training this model was reloaded. Subsequently, the decayed learning rate was used to continue the model training process. Both the learning rate when model training starts (termed the initial learning rate) and the decay factor were hyper-parameters.
3) The weighted cross-entropy loss function [91] was used to penalize false positives and false negatives in different ways. The factor controlling the false-negative weight was set to 1. The factor controlling the false-positive weight (denoted by pos_weight) was a hyper-parameter.
4) To avoid overfitting, early stopping [62] was used during model training. If the model AUC-PR evaluated after each epoch on the validation set did not increase in any of the subsequent 10 epochs, model training was ended.

A three-step transfer learning technique was also designed (see Task 3 in Section 3.4.1). Before finally fine-tuning a model on the UW-m2ABQ dataset, the model was already tuned on the training set of the MrOS-mSQ dataset. Recall that before tuning the model on the training set of the MrOS-mSQ dataset, the mSQ categories were simplified into two classes (see Section 3.2.2). Thus, the model tuned on the training set of the MrOS-mSQ dataset was already used for binary classification. All model weights tuned on the training set of the MrOS-mSQ dataset were used to initialize the model in the fine-tuning step. When fine-tuning each model that was already tuned on the MrOS-mSQ dataset, we froze several layers close to the input layer and used a small learning rate to fine-tune the model. The batch size was 20 and Adam was used. The aforementioned learning rate decay, weighted cross-entropy loss function, and early stopping were

used. The number of frozen layers, the initial learning rate, the decay rate, and pos_weight were hyper-parameters that required tuning.

In each step of each model tuning process, dropout [92] was used to avoid overfitting. For each unit in the fully connected layer before the output layer, the probability of dropping this unit is a hyper-parameter referred to as the dropout rate.

In summary, in the training process of each model, five hyper-parameters required tuning. These hyper-parameters, listed in Table 3.9, were tuned by random search [62] for 2,000 rounds, with the goal of maximizing the AUC-PR on the validation set. The initial learning rate, decay factor, and pos_weight were determined on the logarithmic scale. The dropout rate was determined on a linear scale. The deepest frozen layer L_d was searched on a list. The search space of L_d was different for GoogLeNet, Inception-ResNet-v2, and EfficientNet-B1 because they have distinct architectures. The code representing each layer of each neural network is provided in its open-source code [88, 93, 94] and original paper [82-84]. Table 3.10 lists the optimal values of these hyper-parameters for each model training process. The hyper-parameters not mentioned in this section were set to their default values given by the original papers [82-84] and open-source code [88, 93, 94] of these deep learning algorithms.

Hyper-parameter tuning was performed on two Ubuntu Linux servers concurrently:1) Xeon E5-2630 with four Nvidia GeForce TITAN Xp GPUs and 512 GB of memory and 2) Xeon Gold 5215 with four Nvidia GeForce 2080 Ti GPUs and 96 GB of memory.

Table 3.9. Five hyper-parameters that were tuned by random search.

Hyper-parameter	**Description**	**Search range or search space**
Initial learning rate	Learning rate when model training begins.	$[10^{-6}, 10^{-3}]$
Decay factor	Value by which the learning rate is multiplied to decrease the learning rate.	$[10^{-3}, 1]$
pos_weight	Factor controlling the false-positive weight in the weighted cross entropy loss function.	[0, 10]
Dropout rate	Probability of dropping each unit of the fully connected layer before the output layer.	[0, 1]
L_d	Deepest frozen layer. If $L_d \neq$ none, the input layer up to L_d are frozen. Otherwise, if $L_d =$ none, no layer is frozen.	GoogLeNet: none, 1a, 2b, 2c Inception-ResNet-v2: none, 1a, 2a, 2b, 3b, 4a EfficientNet-B1: none, stem, block1, block2, block3, block4, block5

Table 3.10. For each GoogLeNet, Inception-ResNet-v2, and EfficientNet-B1, the optimal value of each hyper-parameter in each training task.

		Optimal value		
		Task 1: ImageNet → MrOS-mSQ	Task 2: ImageNet → UW-m2ABQ	Task 3: ImageNet → MrOS-mSQ → UW-m2ABQ
GoogLeNet	Initial learning rate	6.95×10^{-4}	5.43×10^{-4}	3.02×10^{-4}
	Decay factor	8.53	16.03	654.70
	pos_weight	0.14	0.35	0.29
	Dropout rate	0.25	0.48	0.33
	L_d	None	1a	1a
Inception-ResNet-v2	Initial learning rate	2.2×10^{-4}	2.90×10^{-4}	1.53×10^{-4}
	Decay factor	71.65	5.17	1.87
	pos_weight	0.71	6.48	0.64
	Dropout rate	0.70	0.42	0.24
	L_d	1a	1a	None
EfficientNet-B1	Initial learning rate	7.96×10^{-3}	2.85×10^{-3}	1.65×10^{-4}
	Decay factor	14.02	1.71	14.06
	pos_weight	0.14	0.31	1.69
	Dropout rate	0.94	0.20	0.97
	L_d	block1	block3	stem

3.5 Model Evaluation

3.5.1 *Description of Model Evaluation*

Using both the UW-m2ABQ dataset's test set and the MrOS-m2ABQ dataset's test set, we tested each of the 15 trained models described in Section 3.4.1. Each model trained in Task 1 was also tested on the MrOS-mSQ dataset's test set. All of the performance measures mentioned in this section were computed using the classification results on individual vertebral patches.

The ensemble majority voting algorithm does not output a numerical value on which a range of cutoff thresholds can be set (see Figure 3.4(C)). Thus, the AUC-PR and the AUC-ROC of the models built using the ensemble majority voting algorithm could not be computed. Instead, the following performance measures were computed: accuracy, sensitivity, specificity, PPV, negative predictive value (NPV), false discovery rate (FDR = 1 – PPV), and F1 score.

For the other trained models, all of the performance measures mentioned above were computed, including the AUC-PR and the AUC-ROC. For measures other than AUC-PR and AUC-ROC, a cutoff threshold was required. To set the cutoff threshold for each of these models, we used two thresholding methods, each applied to the validation set of the dataset whose training set was used to finally fine-tune the model. The same cutoff threshold was then used when testing the model on different test sets. The two thresholding methods are as follows:

1) Set the cutoff threshold to maximize the F_1 score. This automatically sets the cutoff threshold and balances the sensitivity and the PPV.
2) Manually set the threshold to make the PPV approximate 90%. Recall that we prioritize the PPV rather than the sensitivity for our opportunistic screening tool (see Section 3.1). Our initial consultation with local clinicians showed that a PPV of approximately 90% was appropriate.

The 95% confidence interval (CI) of each performance measure was computed using 2,000-fold bootstrap analysis.

Each model was evaluated using a GPU on Xeon E5-2630 with 512 GB of memory.

3.5.2 *Results of Model Evaluation*

We report the performance of our ensemble averaging algorithm in Tasks 2 (ImageNet → UW-m2ABQ) and 3 (ImageNet → MrOS-mSQ → UW-m2ABQ) in this section. The performance of the other models is reported in the appendix in another paper [37].

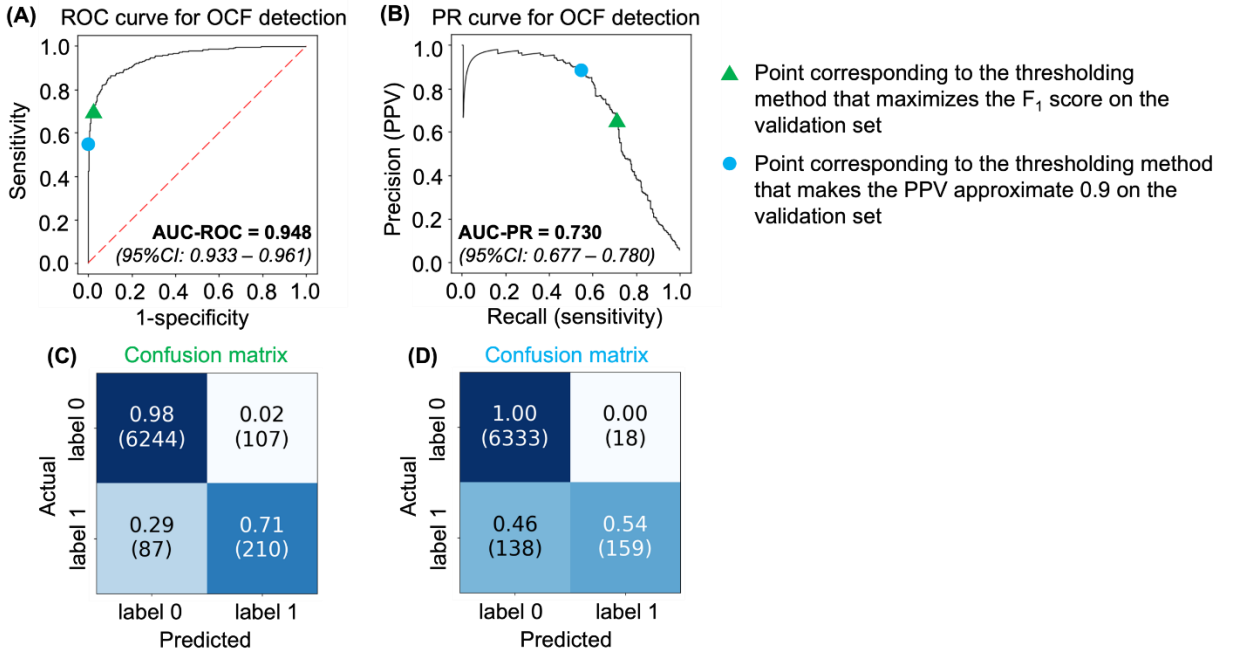
Figure 3.5 and Figure 3.6 show the performance of the model built using the ensemble averaging algorithm in Task 2. Figure 3.5 and Figure 3.6 show this model's performance on the UW-m2ABQ dataset's test set and the MrOS-m2ABQ dataset's test set, respectively.

On the UW-m2ABQ dataset's test set, the model mentioned above yielded an AUC-ROC of 0.948 and an AUC-PR of 0.730. After setting the cutoff threshold to make the PPV approximate 90% on the UW-m2ABQ dataset's validation set, this model achieved a sensitivity of 54.5%, a specificity of 99.7%, a PPV of 89.8%, an NPV of 97.9%, an FDR of 10.2%, an F_1 score of 0.671, and an accuracy of 97.7%.

On the MrOS-m2ABQ dataset's test set, the model mentioned above yielded an AUC-ROC of 0.936 and an AUC-PR of 0.811. After setting the cutoff threshold to make the PPV approximate 90% on the UW-m2ABQ dataset's validation set, this model achieved a sensitivity of 47.8%, a specificity of 99.6%, a PPV of 94.8%, an NPV of 92.4%, an FDR of 5.2%, an F_1 score of 0.636, and an accuracy of 92.5%.

Figure 3.7 shows the performance of the model built using the ensemble averaging algorithm in Task 3 and evaluated on the UW-m2ABQ dataset's test set. This model yielded an AUC-ROC of 0.955 and an AUC-PR of 0.764. After setting the cutoff threshold to make the PPV approximate

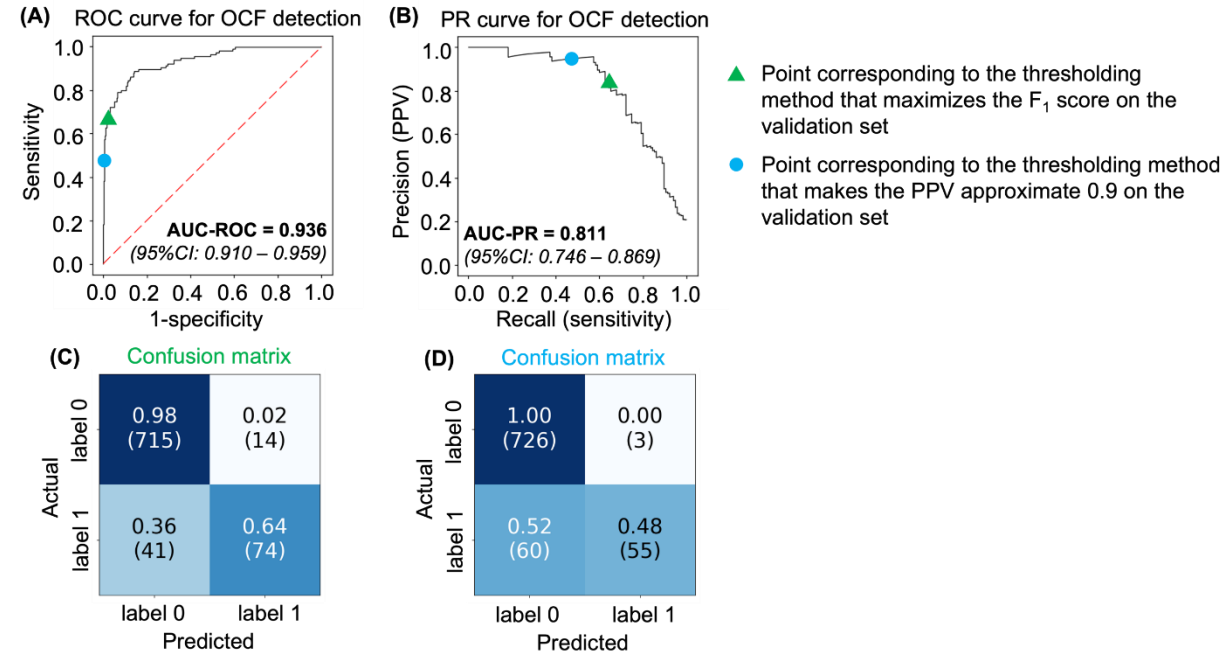
90% on the UW-m2ABQ dataset's validation set, this model achieved a sensitivity of 53.9%, a specificity of 99.7%, a PPV of 89.4%, an NPV of 97.9%, an FDR of 10.6%, an F_1 score of 0.672, and an accuracy of 97.7%.



(E)

Cutoff threshold	Sensitivity	Specificity	PPV	NPV	FDR	F_1 score	Accuracy
0.327	70.7% [65.3%-75.9%]	98.3% [98.0%-98.6%]	66.2% [60.9%-71.6%]	98.6% [98.3%-98.9%]	33.8% [28.4%-39.1%]	0.684 [0.630-0.737]	97.1% [96.7%-97.5%]
0.800	54.5% [48.1%-59.1%]	99.7% [99.6%-99.7%]	89.8% [85.1%-94.1%]	97.9% [97.5%-98.2%]	10.2% [5.9%-14.9%]	0.671 [61.5%-72.6%]	97.7% [97.3%-98.0%]

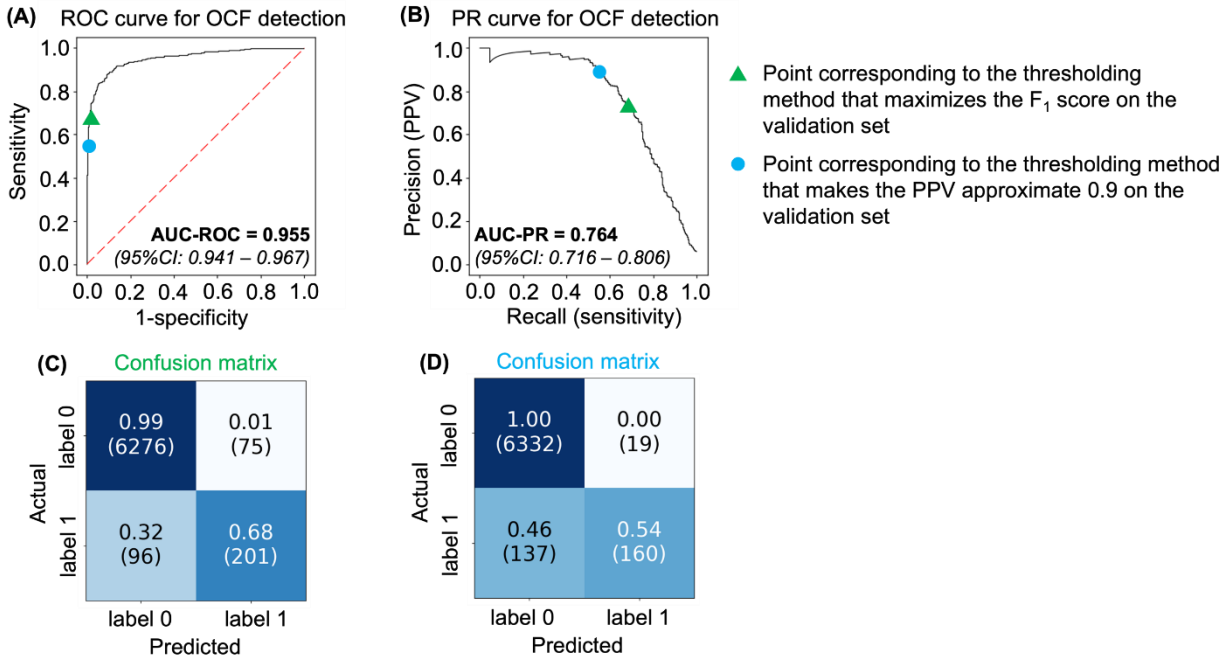
Figure 3.5. The performance of the model, which was built using the ensemble averaging algorithm in Task 2 and evaluated on the test set of the UW-m2ABQ dataset.
(A) The ROC curve and the AUC-ROC with its 95% CI. (B) The PR curve and the AUC-PR with its 95% CI. (C) When the cutoff threshold (0.327) is set to maximize the F_1 score on the UW-m2ABQ dataset's validation set, the confusion matrix with the number of vertebral bodies in each of the four cells shown in the parentheses. (D) The confusion matrix when the cutoff threshold (0.800) is manually set to make the PPV approximate 90% on the UW-m2ABQ dataset's validation set. (E) Using each thresholding method, the sensitivity, specificity, PPV, NPV, FDR, F_1 score, and accuracy with their 95% CIs.



(E)

Cutoff threshold	Sensitivity	Specificity	PPV	NPV	FDR	F_1 score	Accuracy
0.327	64.3% [55.3%-72.8%]	98.1% [97.1%-99.0%]	84.1% [75.9%-91.5%]	94.6% [92.9%-96.2%]	15.9% [8.5%-24.1%]	0.729 [0.640-0.811]	93.5% [91.8%-95.1%]
0.800	47.8% [39.6%-56.8%]	99.6% [99.0%-100%]	94.8% [88.9%-100%]	92.4% [90.5%-94.1%]	5.2% [0-11.1%]	0.636 [54.7%-72.4%]	92.5% [90.9%-94.3%]

Figure 3.6. The performance of the model, which was built using the ensemble averaging algorithm in Task 2 and evaluated on the test set of the MrOS-m2ABQ dataset.
(A) The ROC curve and the AUC-ROC with its 95% CI. (B) The PR curve and the AUC-PR with its 95% CI. (C) When the cutoff threshold (0.327) is set to maximize the F_1 score on the UW-m2ABQ dataset's validation set, the confusion matrix with the number of vertebral bodies in each of the four cells shown in the parentheses. (D) The confusion matrix when the cutoff threshold (0.800) is manually set to make the PPV approximate 90% on the UW-m2ABQ dataset's validation set. (E) Using each thresholding method, the sensitivity, specificity, PPV, NPV, FDR, F_1 score, and accuracy with their 95% CIs.



(E)

Cutoff threshold	Sensitivity	Specificity	PPV	NPV	FDR	F_1 score	Accuracy
0.764	67.7% [62.2%-73.0%]	98.8% [98.6%-99.1%]	72.8% [67.7%-78.1%]	98.5% [98.2%-98.8%]	27.2% [21.9%-32.3%]	0.702 [0.648-0.755]	97.4% [97.1%-97.8%]
0.900	53.9% [48.4%-59.5%]	99.7% [99.6%-99.8%]	89.4% [84.7%-93.8%]	97.9% [97.5%-98.2%]	10.6% [6.2%-15.3%]	67.2% [61.6%-72.3%]	97.7% [97.2%-98.0%]

Figure 3.7. The performance of the model, which was built using the ensemble averaging algorithm in Task 3 and evaluated on the test set of the UW-m2ABQ dataset.
(A) The ROC curve and the AUC-ROC with its 95% CI. (B) The PR curve and the AUC-PR with its 95% CI. (C) When the cutoff threshold (0.764) is set to maximize the F_1 score on the UW-m2ABQ dataset's validation set, the confusion matrix with the number of vertebral bodies in each of the four cells shown in the parentheses. (D) The confusion matrix when the cutoff threshold (0.900) is manually set to make the PPV approximate 90% on the UW-m2ABQ dataset's validation set. (E) Using each thresholding method, the sensitivity, specificity, PPV, NPV, FDR, F_1 score, and accuracy with their 95% CIs.

3.5.3 *Comparison between the Models*

For each deep learning algorithm, there were three training tasks. Each model was tested using two or three test sets. Table 3.11 shows the F_1 score, the AUC-PR, and the AUC-ROC of each

(deep learning algorithm, training task, test set) combination. Note that the AUC-PR and the AUC-ROC of the models built using the ensemble majority voting algorithm could not be computed (see Section 3.5.1). In this section, each model's cutoff threshold was set to maximize the F_1 score on the corresponding validation set.

Table 3.11. F_1 scores, AUC-PR, and AUC-ROC for each (deep learning algorithm, training task, test set) combination. In this table, yellow and magenta are used to mark the MrOS dataset and the UW dataset, respectively.

Training task	**Task 1**: ImageNet → MrOS-mSQ			**Task 2**: ImageNet → UW-m2ABQ		**Task 3**: ImageNet → MrOS-mSQ → UW-m2ABQ	
Test set	MrOS-mSQ	MrOS-m2ABQ	UW-m2ABQ	MrOS-m2ABQ	UW-m2ABQ	MrOS-m2ABQ	UW-m2ABQ
				F_1 score			
GoogLeNet	0.751	0.691	0.579	0.698	0.668	0.694	0.701
Inception-ResNet-V2	0.729	0.652	0.523	0.670	0.659	0.698	0.674
EfficientNet-B1	0.743	0.667	0.543	0.705	0.650	0.747	0.689
Ensemble averaging	0.773	**0.677**	**0.566**	**0.729**	0.684	**0.761**	0.702
Ensemble majority voting	**0.776**	0.648	0.553	0.706	**0.694**	0.713	**0.712**
				AUC-PR			
GoogLeNet	0.817	0.782	0.606	0.784	0.698	0.804	0.736
Inception-ResNet-V2	0.798	0.795	0.636	0.809	0.656	0.801	0.696
EfficientNet-B1	0.816	**0.796**	0.628	0.785	0.703	0.808	0.746
Ensemble averaging	**0.841**	**0.796**	**0.658**	**0.811**	**0.730**	**0.831**	**0.764**
				AUC-ROC			
GoogLeNet	0.990	0.897	0.918	0.927	0.941	0.933	0.949
Inception-ResNet-V2	**0.993**	**0.925**	0.914	0.930	0.925	0.922	0.947
EfficientNet-B1	**0.993**	0.914	0.916	0.914	0.941	0.933	**0.958**
Ensemble averaging	0.992	0.911	**0.930**	**0.936**	**0.948**	**0.940**	0.955

3.6 Discussion

The ensemble averaging model trained in Task 2 achieved our pre-specified objectives of AUC-PR > 0.70 and AUC-ROC > 0.90 on both the UW dataset and the MrOS dataset. When setting the

cutoff threshold to make the PPV approximately 90% on the UW-m2ABQ dataset's validation set, we obtained high PPVs and specificities with moderate sensitivities on both datasets. This is acceptable for our clinical use case of an opportunistic screening tool described in Section 1.1 and Section 3.1, in which the PPV and specificity rather than the sensitivity should be prioritized. An opportunistic screening tool could be clinically useful with a moderate sensitivity and a high specificity or PPV. Given the volume of radiographic exams that cover some portion of the thoracic and lumbar spine at most medical institutions, it is prudent to consider the downstream effects of positive and negative predictive results. A positive predictive result would result in provider efforts guiding the patient to the appropriate clinical care as well as patient expense, worry, radiation exposure, and potential harm. A negative predictive result would result in no further action and would not affect the current standard of clinical care. Our opportunistic screening tool will only augment current clinical practice rather than replace radiologist interpretation or any other step in the current clinical workflow. In this setting, a false negative is a missed opportunity, but could still be possibly caught by the current standard of care. A false positive triggers extra work that has no obvious benefit to the patient but potential harm and financial burden. Our model with a PPV of about 90% and a sensitivity of about 50% can detect nearly half of the unreported fractured vertebral bodies with limited extra cost. It is worth noting that many diagnostic tests in use today have modest sensitivities. Papanicolaou smear has a sensitivity of 55.4% and a specificity of 94.6% [95].

In Section 3.5.3, we compared the performance of each (deep learning algorithm, training task, test set) combination. We have six observations:

1) In each (training task, test set) combination, the models built using the two ensemble algorithms typically outperformed the other models.

2) In each (training task, test set) combination, the two ensemble algorithms typically produced models with similar F_1 scores. Unlike the ensemble majority voting algorithm that outputs categorical values, the ensemble averaging algorithm provided numerical outputs to which different cutoff thresholds could be applied. Thus, the ensemble averaging algorithm is more flexible and can be adapted for different clinical use cases.
3) In Task 2 (ImageNet → UW-m2ABQ), the model built using the ensemble averaging algorithm had a better F_1 score and a higher AUC-PR on the MrOS-m2ABQ dataset than on the UW-m2ABQ dataset. This shows that the model built using the ensemble averaging algorithm has some generalizability. Counterintuitively, this model performed worse on the test set of the UW-m2ABQ dataset, whose training set was used for fine-tuning this model, than on the MrOS-m2ABQ dataset. The reason could be that the data in the UW dataset are more diverse, especially in subject positioning and image artifacts, increasing difficulty of OCF classification.
4) On each test set, each model trained in Task 3 (ImageNet → MrOS-mSQ → UW-m2ABQ) typically had a higher F_1 score and a better AUC-PR than the corresponding model trained in Task 2 (ImageNet → UW-m2ABQ) did. Our transfer learning technique in Task 3 could improve models' performance. However, since each model trained in Task 3 was tuned using both datasets, we cannot claim that this model is generalizable. We need more datasets to show these models' generalizability.
5) In Task 1 (ImageNet → MrOS-mSQ), the AUC-PR of each model tested on the MrOS-mSQ dataset was higher than that of each model tested on the MrOS-m2ABQ dataset but to a limited degree (e.g., 5.7% for the ensemble averaging algorithm). This could imply that our two binary

OCF labeling systems (simplified from the mSQ criteria and the m2ABQ criteria, respectively) are similar.

6) In Task 1, the F_1 score and the AUC-PR of each model tested on the MrOS-mSQ dataset were higher than those of each model tested on the UW-m2ABQ dataset, respectively (e.g., 36.6% and 27.8% greater, respectively by the F_1 score and the AUC-PR, for the ensemble averaging algorithm). The models fine-tuned on the MrOS-mSQ dataset were not generalizable to the UW-m2ABQ dataset. The MrOS dataset was obtained for research, while the UW dataset was extracted from clinical data that were more diverse in demographics, X-ray techniques, and image artifact variations. This greater diversity is likely the cause of poor performance by models only fine-tuned on the MrOS dataset.

Researchers from other research projects [25-29] reported approaches to automatically detecting OCFs using radiographs. Using lumbar or thoracolumbar spine radiographs, Chou *et al.* [25] did automatic segmentation to extract the vertebral bodies and classified each vertebral body using an ensemble method. Using similar methods, Li *et al.* [26] trained models to automatically detect vertebral fractures on lateral spine radiographs. Chen *et al.* [27] and Murata *et al.* [28] respectively trained a deep learning model to detect vertebral fractures on a radiograph without vertebral body segmentation. The main limitation of each of the above projects is that a single-site dataset was used. This resulted in a more homogeneous population, making the trained models less generalizable.

Xiao *et al.* [29] trained and tested their models on women's lateral spine and chest radiographs from multiple sites, showing that their models had good generalizability and could serve as an opportunistic screening tool for female OCF screening. Based on their models, they developed a software program with a user interface. However, except for two datasets, they did not mention

the source, the dataset construction process, and the demographic information of the other datasets in detail. The two known datasets were retrieved from the Osteoporotic Fractures in Women (MsOS) Hong Kong dataset [96]. Like the MrOS dataset, the MsOS Hong Kong dataset was originally collected for research and has some selection bias. Their recruitment criteria included that all subjects were able to walk without assistance [96]. The radiographs in this dataset likely contain far fewer imaging chain artifacts like angulation, position, overlapping, motion, and equipment, which are commonly seen in standard clinical imaging, and are seen when comparing the UW and MrOS datasets in our study.

In contrast to the above projects, we used data assembled from multiple sites with detailed description of the dataset construction process and demographic information (see Section 3.2). Our UW dataset was retrieved from local clinical sites and thus is more consistent with the distribution of clinical data. Shown in Table 3.6, the UW dataset contains subjects that have varied race, ethnicity, and gender, as well as radiographs generated from different X-ray machines, which could help improve the generalizability of our trained models.

Our models have several limitations:

1) We used lateral spine radiographs to build our classifiers. This type of radiograph is optimized to show bones, and thus a rational initial target for research. However, to increase the target population in the future, other radiographs like lateral chest or abdominal radiographs should be used.
2) Our current model classifies individual vertebral bodies extracted from spine radiographs using manual annotation. This ensures that the vertebral bodies are correctly bounded on a radiograph but is not automated or scalable. As mentioned in Section 3.1, we are testing and

separately reporting image segmentation models to automatically localize the vertebral bodies on a radiograph [34].

3) Currently, we only have one dataset (the UW dataset) containing data acquired in varied clinical settings for diagnostic purposes. The number of annotated radiographs in the UW dataset is small. We need more annotated clinical data to train our model and test its generalizability. In the future, we will annotate more radiographs from various clinical sites using semi-automated approaches.
4) In this study, the cutoff thresholds set using the two thresholding methods might not be the best for the clinical use case. We have already surveyed a variety of clinical providers to determine an acceptable performance threshold for automated opportunistic OCF screening. We will further analyze our survey results to determine the most appropriate cutoff threshold for the clinical use case.
5) In this study, we did not analyze incorrectly classified cases and explore how image features contribute to each model's outputs. These two tasks should be implemented in the future to understand how the model works, its failure modes, and how to further improve the model.

3.7 CONCLUSION

We used five deep learning algorithms to train models that detected OCFs of vertebral bodies extracted from spine radiographs. The ensemble averaging model trained in Task 2 achieved our pre-specified objectives of AUC-PR > 0.70 and AUC-ROC > 0.90 on both the UW dataset and the MrOS dataset. This model has good performance and some generalizability and can serve as a critical component of our future automated opportunistic screening tool.

Chapter 4. PROGRESS INDICATION FOR DEEP LEARNING MODEL TRAINING

Deep learning is the state-of-the-art learning algorithm for many machine learning tasks [62]. Yet, training a deep learning model on a large dataset is often time-consuming, taking several days or even months [53-58]. During model training, it is desirable to offer a non-trivial progress indicator that can continuously project the remaining model training time and the fraction of model training work completed [59]. This makes the training process more user-friendly. In addition, we can use the information given by the progress indicator to assist in workload management [97, 98]. In this chapter, we present the first method [64] to support non-trivial progress indicators for deep learning model training when early stopping [62] is allowed. We term this method the basic method. This basic method revises its predicted model training cost using information gathered at the validation points, where the model's error rate is computed on the validation set. Due to the sparsity of validation points, the resulting progress indicators often have a long delay in gathering information from enough validation points and obtaining relatively accurate progress estimates. In this chapter, we further propose an improved progress indication method [65] to overcome this shortcoming by judiciously inserting extra validation points between the original validation points.

We report an implementation of our progress indication methods in TensorFlow [66] and our evaluation results for both convolutional and recurrent neural networks. Our experiments show that our progress indicator can offer useful information even if the run-time system load varies over time [64]. In addition, the progress indicator can self-correct its initial estimation errors, if any, over time. Compared with using the basic progress indication method, using the improved method reduces the progress indicator's prediction error of the model training time left by 57.5%

on average. Also, with a low overhead, this improved method enables us to obtain relatively accurate progress estimates faster.

The rest of the chapter is organized as follows. Section 4.1 introduces the background and motivations of this study. Section 4.2 presents the related work to this study. Section 0 introduces some concepts and notations that will be used throughout this chapter. Sections 4.4 and 4.5 describes our basic and improved progress indication methods for deep learning model training, respectively. Section 4.6 reports the performance test results by implementing our progress indication methods in TensorFlow. Section 4.7 points out some directions for future work. Section 4.8 gives the conclusion.

4.1 INTRODUCTION

The need for non-trivial progress indicators for deep learning model training

Deep learning is the state-of-the-art learning algorithm for many machine learning tasks like image classification, NLP, and speech recognition [62]. But, building a deep learning model on a large dataset is often time-consuming. Using 50 GPUs, a Google team spent two months training a deep neural network on 300 million images [53]. With 200 CPUs, Weyand *et al.* [54] took 2.5 months to train a convolutional neural network on 126 million photos. With one GPU, Colón-Ruiz *et al.* [55] spent more than 22 hours training a BERT model [56] on 215,063 drug reviews for sentiment analysis. Using a GPU, Guo *et al.* [57] spent ten hours to train a convolutional neural network on approximately 10 million image patches with a resolution of 28×28 for tumor segmentation. Sufficient computing resources could be difficult for individual clinicians or clinical research groups to obtain [58], making deep learning model training more time-consuming. As a standard human-computer interaction principle [59], for each task running longer than 10 seconds, we need a non-trivial progress indicator (see Figure 1.2) to continuously project the remaining task

running time and the fraction of the task completed. Thus, progress indicators are desirable to make the deep learning model training process more user-friendly.

Besides making the deep learning model training process more user-friendly, we can use the information given by the progress indicator to assist with workload management as outlined in our other papers [97, 98]. We once talked with Yasser M. Ibrahim, the previous head of distributed machine learning at Amazon. He mentioned that using a large computer cluster, his team took several months to train a deep neural network supporting Alexa's speech recognition function. Every so often, his team retrained this neural network and would like to finish the re-training in a given amount of time. As the amount of training data, the neural network's hyper-parameter values, and the server capacity continue changing over time, his team needed a method to find an appropriate cluster configuration for each round of re-training. A workload management approach aided by progress indicators would serve this purpose [97].

A neural network is trained in one or more epochs, each of which requires going through all of the training instances once. Some deep learning software supplies trivial progress indicators during model training, e.g., by displaying the number of epochs that have been completed [60] or the value of the objective function achieved [61] over time. However, this information is too coarse-grained for many purposes. On a large dataset, a large amount of time is needed to go through an epoch. Moreover, early stopping is widely used in deep learning model training to help avoid overfitting. When early stopping is allowed, the number of epochs needed for model training is unknown beforehand but dynamically decided during model training based on a stopping criterion [62]. How to support such progress indicators in the presence of early stopping remains an open problem.

To address the gap, we propose the first method [64] to support non-trivial progress indicators for deep learning model training when early stopping is allowed. We term this method the basic method. With low overhead, the basic method can handle various combinations of the deep learning model, the learning rate schedule like learning rate decay, and the optimization method. The progress indicator built using the basic method can offer useful information even if the run-time system load varies over time [64]. In addition, the progress indicator can self-correct its initial estimation errors, if any, over time.

The objective of the improved progress indication method for deep learning model training

The basic progress indication method [64] computes progress estimates for the model training process using information gathered at the validation points. Despite producing useful results, this method has a shortcoming. Due to the sparsity of validation points, the resulting progress indicators often have a long delay in obtaining relatively accurate progress estimates. More specifically, at the beginning of model training, we come up with a crude estimate of the model training cost that is usually inaccurate. In our basic method, at least three data points are needed to estimate the three parameters of the regression function that is used to predict the model training cost. Consequently, the predicted model training cost is revised starting from the third validation point, which is too late. Then a revision is made only at each subsequent validation point, which is infrequent. The combination of these two factors often causes a long delay in gathering information from enough validation points and obtaining relatively accurate progress estimates.

For example, Goyal *et al.* [99] used eight Nvidia Tesla P100 GPUs to train the ResNet-50 convolutional neural network on the ImageNet Large-Scale Visual Recognition Competition dataset [100]. About 19 minutes passed between two successive validation points [99, 101]. By the time the progress indicator revised its predicted model training cost for the first time, $3 \times 19 =$

57 minutes had elapsed. This is a long delay that takes up a non-trivial fraction of the 29-hour model training time [99].

The objective of designing the improved progress indication method is to overcome the basic progress indication method's shortcoming of having a long delay in obtaining relatively accurate progress estimates for the deep learning model training process. Compared with the basic progress indication method, the improved method starts revising the predicted model training cost earlier and revises the predicted model training cost more frequently. This helps the progress indicator reduce its prediction error of the remaining model training time and obtain relatively accurate progress estimates faster.

In this chapter, we focus more on the improved progress indication method than the basic method. We report our implementation of our improved progress indication method in TensorFlow [66], an open-source software package for deep learning. We present our performance test results for both convolutional and recurrent neural networks. Our results show that compared with using our basic method, using this improved method reduces the progress indicator's prediction error of the remaining model training time by 57.5% on average. Also, with a low overhead, this improved method enables us to obtain relatively accurate progress estimates faster.

4.2 Related Work

This section provides a brief review of the related work. A detailed discussion of the related work is provided in another paper [97].

Sophisticated progress indicators

Several research groups have proposed sophisticated progress indicators for static program analysis [102], software model checking [103], program compilation [104], database queries [98,

105-108], MapReduce jobs [109, 110], subgraph queries [111], and automatic machine learning model selection [112, 113]. In addition, for training machine learning models, we have created sophisticated progress indicators for random forest, decision tree, as well as neural network [64, 65, 97, 114, 115]

Estimating the training time of deep learning models

To estimate the running time of an epoch before deep learning model training begins, Justus *et al*. [116] developed a meta learning approach that uses multiple features of the computing resources, the present deep learning model, and the training dataset employed to train another deep learning model. That approach projects neither the amount of time nor the number of epochs required to train a deep learning model.

To project the amount of time required to train a deep learning model before model training begins, researchers have developed multiple methods including meta learning employing support vector regression [117], meta learning employing Multivariate Adaptive Regression Splines [118], meta learning employing polynomial regression [119], and Bayesian optimization [120]. The estimates given by these methods are not kept being refined, are often inaccurate, and can diverge greatly from the real model training time on a loaded computer. In comparison, our progress indication method for deep learning model training keeps refining its estimates and considers the load on the computer when projecting the remaining model training time.

Complexity analysis for training neural networks

Many researchers have studied the time complexity of training a neural network [121, 122]. However, the time complexity information gives no estimate of the model training time on a loaded computer and is insufficient for us to develop progress indicators. Typically, time complexity

considers neither data properties that affect the model training cost nor the coefficients and the lower order terms required to predict the model training cost. A good progress indicator should keep refining its estimated model training cost during model training.

4.3 Some Concepts and Notations

In this section, we introduce some concepts and notations that will be used throughout this chapter. To control model training, the user of the deep learning software specifies an early stopping condition and three positive integers B, g, and m_e. The deep learning model is trained in batches, each processing B training instances to calculate parameter value updates to the model. We reach an original validation point after finishing every g batches of model training. There, we first calculate the validation error, which is the model's error rate on the validation set. Then we assess whether the early stopping condition is fulfilled. If so, we end model training. m_e denotes the maximum number of epochs allowed for model training. If the early stopping condition remains unfulfilled by the time we finish the m_e-th epoch, we end model training at that time. Thus, the maximum number of batches allowed for model training is

$$b_{max} = \text{the number of data instances that are in the training set} \times m_e / B.$$

The maximum number of original validation points allowed for model training is

$$v_{max} = \lfloor b_{max} / g \rfloor, \tag{4.1}$$

where $\lfloor \rfloor$ is the floor function, e.g., $\lfloor 4.4 \rfloor = 4$.

The validation curve depicts the validation errors obtained over time during model training. Many early stopping criteria exist, most of which are based on the validation curve [62, 123-125]. One criterion is to stop model training when the validation error has not improved over the best one recorded for a given number of validation points [62, 123]. Another criterion adopts the idea of

stopping model training when the validation error is over the best one recorded by at least a given threshold, while the model's error rate on the training set no longer improves much [123]. Duvenaud *et al.* [124] proposed a criterion based on estimating the log marginal likelihood without using a validation set. Mahsereci *et al.* [125] proposed a criterion based on some local statistics of the computed gradients without using a validation set.

The goal of this work is not to deal with every early stopping condition that exists. Instead, we focus on a commonly used early stopping condition [62, 126]. Through a case study on the condition, we demonstrate that when early stopping is allowed, it is feasible 1) to provide non-trivial and useful progress indication for deep learning model training and 2) to obtain relatively accurate progress estimates faster by judiciously inserting extra validation points between the original validation points. The early stopping condition uses two pre-determined numbers: patience $p > 0$ and min_delta $\delta \geq 0$. The condition is fulfilled when the validation error drops by $< \delta$ for p original validation points in a row. In other words, letting $\tilde{e}_j$ denote the validation error of the model at the j-th original validation point, we end model training at the k-th original validation point when $\tilde{e}_{k-p} - \tilde{e}_i$ is $< \delta$ for each i between $k - p + 1$ and k.

4.4 Basic Progress Indication Method

In this section we present the basic method to support non-trivial progress indicators for deep learning model training when early stopping is allowed. Our presentation focuses on using deep learning for classification. Section 4.4.1 describes the innovation of this study. Section 4.4.2 gives an overview of our basic progress indication method. Sections 4.4.3-4.4.5 show how to estimate the number of batches needed for model training when a fixed learning rate is used during the entire model training process.

4.4.1 *Innovation of the Basic Progress Indication Method*

A deep learning model is trained in batches. In each batch, a fixed number of training instances are used to compute the updates to the model's parameters. Each batch's running cost is relatively stable and can be quickly measured. Thus, the key to estimating the progress of model training is to project the number of batches needed for model training. During model training, we use the non-smooth validation curve to make this projection. As Figure 4.1 shows, the validation error tends to reduce over time before early stopping occurs and also oscillates over time. If we use a monotonically decreasing function to model the validation curve without accommodating the oscillations, and directly apply the early stopping criterion to the projected curve, we seldom obtain a good estimate of the number of batches needed for model training. To address this challenge, we regard the validation curve as the sum of a smooth trend curve and some zero-mean random noise. We use a regression function to estimate the trend curve, and historical data to gauge the random noise's variance. If the learning rate changes over time, we also model the change's impact on the random noise's variance. Then we use a Monte Carlo simulation approach to project the number of batches needed for model training. By adding simulated random noise to the projected trend curve, we generate several synthetic validation curves. On each of them, we apply the early stopping criterion to obtain a simulated number of batches needed for model training. The estimated mode of these simulated numbers forms the basis for the projected number of batches needed for model training. To the best of our knowledge, this is the first time Monte Carlo simulation has been used for progress indication and is a main innovation of this work.

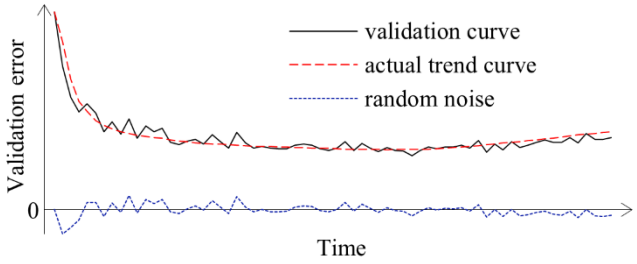


Figure 4.1. The validation curve = a trend curve + some random noise.

4.4.2 *Overview of the Basic Progress Indication Method*

In this section, we give an overview of our progress indication method. We start with an initial estimate of the model training cost. Both the predicted model training cost and the current model training speed are gauged by *U*, the unit of work. Each *U* depicts the average amount of work needed for processing each training instance once in two steps in model training. The first step is to go forward through the neural network once to compute its prediction result on the training instance. The second step is to go backwards through the neural network once for backpropagation.

During model training, we keep gathering multiple statistics, such as the number of batches done, and use them to keep refining the estimated model training cost. We keep checking the model training speed defined as the number of *U*s completed per second during the *K* seconds before the current time point. By default, *K*'s value is 10. At any moment,

the projected remaining model training time
= the projected remaining model training cost / the current model training speed.

Periodically, we update the progress indicator with the latest information. As the model training task keeps running, we gather more precise information about it. As a result, our estimates tend to become increasingly accurate over time.

Computing the model training cost

The model training cost is dominated by two components and can be roughly regarded as their sum. The first component is the cost of processing the training instances. The second component is the cost of computing the validation errors. The first one is easy to compute.

The cost of processing the training instances
= the number of batches needed for model training × the number of training instances per batch × the average amount of work needed for processing a training instance once in model training
= the number of batches needed for model training × B × 1
= the number of batches needed for model training × B.

Next, we compute the second component. We call each data instance in the validation set a validation instance.

The cost of computing the validation errors
= the number of validation points needed for model training × the number of data instances in the validation set × the average amount of work needed for processing a validation instance once to compute the validation error.

To process a validation instance once, we need to go forward through the neural network once to compute its prediction result on the validation instance. We use the number of multiplication operations needed to estimate the processing cost. Each neuron typically takes multiple inputs, each of which links to a distinct connection weight. When going forward through the neural network, we need to compute the neuron's output by multiplying each input by its linked connection weight. In comparison, when going backwards through the neural network, we need to compute a partial derivative with respect to each input and a partial derivative with respect to each connection weight. The former step requires doing a multiplication with the connection weight linked to the input. The latter requires doing a multiplication with the input linked to the connection weight. Hence, as a

rough approximation based on the number of multiplication operations needed, we regard the cost of going backwards through the neural network once to be twice that of going forward through the neural network once. That is, the average amount of work needed for processing a validation instance one time = U / 3. Consequently,

the cost of computing the validation errors
= the number of validation points needed for model training × V / 3,

with V being the number of data instances in the validation set.

Summing the two components, we have

the model training cost
= the number of batches needed for model training × B + the number of validation points needed for model training × V / 3.

Before model training starts, we can easily know B and V's values. Thus, to estimate the model training cost, we mainly need to estimate the number of batches and the number of validation points needed for model training.

Let T denote the number of data instances in the training set. Before a deep neural network is trained, the user of the deep learning software needs to specify the value of a hyper-parameter m_e showing the maximum number of epochs allowed for model training. Each epoch requires passing through all of the training instances once and includes T / B batches of model training. The maximum number of batches allowed for model training is

$$b_{max} = m_e \times T / B.$$

Before model training starts, we can easily know T and B's values and subsequently b_{max}'s value. Recall that a validation point is reached every g batches of model training. If early stopping occurs before finishing the b_{max}-th batch,

$$\begin{aligned} &\text{the number of batches needed for model training} \\ = \ &\text{the number of validation points needed for model training} \times g. \end{aligned}$$

If early stopping never occurs and model training reaches the maximum number of batches allowed,

$$\text{the number of batches needed for model training} = b_{max},$$

and

$$\text{the number of validation points needed for model training} = v_{max} = \lfloor b_{max} / g \rfloor.$$

Recall that v_{max} is the maximum number of validation points allowed for model training. The last equation holds because of formula (4.1). Thus, the key to estimating the model training cost is to estimate the number of validation points needed for model training, and subsequently, whether early stopping will ever occur.

Estimating the number of validation points needed for model training

Initially, with no extra information, we estimate the number of validation points needed for model training to be v_{max}, the maximum number of validation points allowed for model training. During model training, once the number of validation points reached is ≥ a given threshold τ_v, we start using the validation curve to keep refining the projected number of validation points needed for model training. In our implementation, we choose 3 as τ_v's default value to strike a balance between having enough validation points to make a reasonable projection and not having to wait too long before the initial projected number could be refined.

As Figure 4.1 shows, the validation curve often oscillates over time. We regard it as the sum of a smooth trend curve and some zero-mean random noise. At each validation point that is after the τ_v-th one and where the early stopping criterion is unmet, we first fit a smooth regression function to the validation curve up to this point, and then use the fitted function to estimate the trend curve

beyond this point. Since the regression function is smooth, the estimated trend curve does not reflect the oscillations on the validation curve. Thus, directly applying the early stopping criterion to the estimated trend curve often does not lead to a good estimate of the number of validation points needed for model training. For example, as the validation error tends to decrease over time, we use a monotonically decreasing regression function to estimate the trend curve. When the min_delta $\delta = 0$, the early stopping criterion includes a term that the validation error increases at some point. Thus, the criterion is never met on the estimated trend curve, even if early stopping occurs frequently in practice.

To address this issue, we use historical data to gauge the random noise's variance. Then we use a Monte Carlo simulation approach to project the number of validation points needed for model training. By adding simulated random noise to the projected trend curve, we generate several synthetic validation curves. To each of them, we apply the early stopping criterion and obtain the number of validation points needed. The smaller of this number and v_{max}, the maximum number of validation points allowed for model training, becomes a simulated number of validation points needed for model training. The estimated mode of these simulated numbers forms the basis for our projected number of validation points needed for model training.

In the rest of this Section 4.4, we focus on the case where a fixed learning rate is used during the entire model training process. We show how to estimate the number of validation points needed for model training upon reaching a validation point that is after the τ_v-th one and at which the early stopping criterion is unmet. Section 4.4.3 shows the regression method used to estimate the trend curve. Section 4.4.4 covers how to estimate the random noise's variance. Section 4.4.5 presents the Monte Carlo simulation approach used to project the number of validation points needed for model training.

4.4.3 *Estimating the Trend Curve*

The validation error tends to decrease over time, whereas the rate of decrease typically reduces over time. In keeping with this, we use the same inverse power law function [97, 127-130] of the form

$$r(i) = ai^{-b} + c$$

as the regression function to model both the validation and trend curves (see Figure 4.1). Here, i is the sequence number of the validation point, $a > 0$, $b > 0$, and $c > 0$. We first fit the function to the validation curve up to the current validation point, and then use the fitted function to estimate the trend curve beyond that point.

Intuitively, the validation points well before the current one may not accurately reflect the validation curve's trend beyond the current validation point and could be unsuitable for function fitting. Thus, we use a pre-set window size w' whose default value is 50 to skip these validation points. Let n denote the number of validation points obtained thus far. When fitting the regression function to the validation curve, we use the last

$$w = \min(w', n)$$

validation points instead of all of the n validation points obtained thus far. To compute a, b, and c's values, we solve a constrained minimization problem:

$$\min \sum_{i=n-w+1}^{n} [\tilde{e}_i - (ai^{-b} + c)]^2$$

the sum of the squared errors at the last w' validation points, subject to the constraints that $a > 0$, $b > 0$, and $c > 0$. Recall that $\tilde{e}_i$ is the validation error at the i-th validation point. One way to do constrained minimization is to use the truncated Newton method [131] and initialize a, b, and c as one, one, and zero, respectively.

4.4.4 *Estimating the Random Noise's Variance*

Recall that we regard the validation curve as the sum of a smooth trend curve and some zero-mean random noise. $\tilde{e}_i$, $r(i)$, and $\tilde{e}_i - r(i)$ are the validation error, the estimated value of the trend curve, and the estimated value of the random noise at the i-th validation point, respectively. n is the number of validation points obtained thus far. w is the number of validation points used to fit the regression function. We use the last w validation points to estimate the random noise's variance as

$$\hat{\sigma}^2 = \frac{1}{w}\sum_{i=n-w+1}^{n}[\tilde{e}_i - r(i)]^2.$$

4.4.5 *Projecting the Number of Validation Points Needed for Model Training*

We use a Monte Carlo simulation method to project the number of validation points needed for model training. To the best of our knowledge, this is the first time Monte Carlo simulation has been used for progress indication. Our method works as follows:

1) **Step 1:** For each i ($n + 1 \leq i \leq v_{max}$), compute the estimated value $r(i)$ of the trend curve at the i-th validation point. Recall that n is the number of validation points obtained thus far. v_{max} is the maximum number of validation points allowed for model training. All of these $r(i)$ ($n + 1 \leq i \leq v_{max}$) form the estimated trend curve beyond the current validation point, up to the last one allowed for model training.
2) **Step 2:** For each i ($n + 1 \leq i \leq v_{max}$), randomly sample a number n_i from the normal distribution $N(0, \hat{\sigma}^2)$ as simulated random noise at the i-th validation point. Recall that $\hat{\sigma}^2$ is the estimated variance of the random noise. $r(i) + n_i$ is a simulated validation error at the i-th validation point. All of the $r(i) + n_i$ ($n + 1 \leq i \leq v_{max}$) form a synthetic validation curve beyond the current validation point, up to the last one allowed for model training.

3) **Step 3:** Connect the actual validation curve up to the current validation point and the synthetic validation curve beyond that point to obtain a full synthetic validation curve, which goes from the first validation point to the last one allowed for model training.
4) **Step 4:** For each i ($n + 1 \leq i \leq v_{max}$), check one by one whether the early stopping criterion is met on the full synthetic validation curve at the i-th validation point. If the early stopping criterion is not met anywhere, we obtain v_{max} as a simulated number of validation points needed for model training, and b_{max} as a simulated number of batches needed for model training. Recall that v_{max} and b_{max} are the maximum number of validation points and the maximum number of batches allowed for model training, respectively. Otherwise, if the early stopping criterion is met on the full synthetic validation curve for the first time at the j-th ($n + 1 \leq i \leq v_{max}$) validation point, we obtain j as a simulated number of validation points needed for model training, and $j \times g$ as a simulated number of batches needed for model training. Recall that g is the number of batches of model training between two consecutive validation points.
5) **Step 5:** Repeat Steps 2-4 k times to obtain k simulated numbers of validation points needed for model training, which we term simulated estimates. k is a pre-set parameter. We choose 2,000 as its default value to obtain enough simulated estimates for our projection purpose without incurring excessive simulation overhead.

One could use the mode of the k simulated estimates as the projected number of validation points needed for model training. Compared to the mean, the mode is a more robust statistic in the presence of outliers [132]. Yet, using the mode directly is suboptimal. When there are ≥ 2 local modes with roughly the same frequency, which one of them is the global mode is somewhat random, resulting in instability of the projection. Considering this, we make a projection in the following way.

6) **Step 6:** By definition, every simulated estimate $\in [n + 1, v_{max}]$. Divide $[n + 1, v_{max}]$ into r disjoint intervals of equal width. r is a pre-set parameter whose default value is 200. Set a threshold

$$\gamma = k \times c_\gamma,$$

where c_γ is a coefficient whose default value is 0.04. Group the k simulated estimates by interval. Find every interval containing $> \gamma$ simulated estimates. Each such interval is regarded as a local mode. If the number of such intervals is ≥ 1, average the simulated estimates in all such intervals as the projected number of validation points needed for model training. Otherwise, if no such interval exists, the k simulated estimates spread relatively evenly across a wide range with no significant local mode. Their mean becomes the projected number of validation points needed for model training.

4.5 Improved Progress Indication Method

The basic progress indication method presented in Section 4.4 revises its predicted model training cost using information gathered at the validation points. Due to the sparsity of validation points, the resulting progress indicators often have a long delay in gathering information from enough validation points and obtaining relatively accurate progress estimates. We design the improved progress indication method to overcome this shortcoming by judiciously inserting extra validation points between the original validation points.

In this section, we present our improved progress indication method for deep learning model training. Our presentation focuses on using deep learning for classification and the steps related to estimating the trend curve, the variance of the random noise, and the model training cost based upon the predicted number of original validation points needed for model training. The approaches to conduct Monte Carlo simulation to estimate the number of original validation points needed for

model training, to monitor the present model training speed, and to estimate the remaining model training time based upon the projected remaining model training cost and the present model training speed are identical to those used in our basic progress indication method for deep learning model training (see Section 4.4) and are omitted.

This section is organized in the following way. Section 4.5.1 presents the innovation of the improved progress indication method for deep learning model training. Section 4.5.2 provides an overview of this new progress indication method. Section 4.5.3 presents our approach to insert extra validation points between the original validation points. Section 4.5.4 shows how to set V', the uniform size of the randomly sampled subset of the full validation set that will be used at each added validation point. Section 4.5.5 derives the relationship between the random noise's variance and the size of the actual validation set used at the validation point. Section 4.5.6 shows how to estimate the trend curve and the variance of the random noise for future validation points. Section 4.5.7 describes how to determine V_{min}, the minimum size needed for the randomly sampled subset of the full validation set used at an added validation point. Section 4.5.8 shows how to estimate the model training cost based upon the predicted number of original validation points needed for model training.

In the rest of this chapter, whenever we mention validation points, we mean both original and added validation points, unless original validation points or added validation points are explicitly mentioned.

4.5.1 *Innovation of the Improved Progress Indication Method*

The objective of the improved progress indication method is to overcome our basic progress indication method's shortcoming of having a long delay in obtaining relatively accurate progress estimates for the deep learning model training process. To obtain relatively accurate progress

estimates faster, our improved progress indication method for deep learning model training judiciously inserts extra validation points between the original validation points. The predicted model training cost is revised at both the original and the added validation points. Consequently, compared with our basic progress indication method, our improved progress indication method starts revising the predicted model training cost earlier and revises the predicted model training cost more frequently. This helps the progress indicator reduce its prediction error of the remaining model training time and obtain relatively accurate progress estimates faster.

A good progress indicator should have a low run-time overhead [97]. In our improved progress indication method, a large part of the progress indicator's run-time overhead comes from computing the validation error at the added validation points. To lower this part of the run-time overhead, at each added validation point, we calculate the validation error on a randomly sampled subset of the full validation set rather than on the full validation set.

To fill in the rest of our improved progress indication method, we need to solve three technical challenges. First, we need to set 1) n_j ($j \geq 0$), the number of validation points to be added between the j-th and the (j+1)-th original validation points, and 2) V', the uniform size of the randomly sampled subset of the full validation set that will be used at each added validation point. Through theoretical reasoning, we show that n_j should decrease as j increases. For this purpose, exponential decay works better than linear decay. V' is chosen to control the total overhead of computing the validation error at the validation points added before the first original validation point, while keeping the randomly sampled subset of the full validation set large enough for reasonably estimating the model's generalization error [133] at each added validation point.

Second, as in Section 4.4, we use the validation curve to predict when early stopping will occur. As shown in Figure 4.1, this curve shows the validation errors obtained over time, is non-smooth,

and can be regarded as the sum of some zero-mean random noise and a smooth trend curve. The random noise's variance depends on the size of the actual validation set used at the validation point. The relationship between these two numbers is previously unknown and difficult to be derived directly. However, we need to know this relationship in order to use both the original and the added validation points to predict when early stopping will occur. Noting that the random noise's variance is equal to the validation error's variance, we use an indirect approach to derive this relationship. We first compute the conditional mean and the conditional variance of the validation error given the model's generalization error, both of which can be expressed using the model's generalization error and the size of the actual validation set used at the validation point. Then we use the conditional mean, the conditional variance, and the law of total variance [134] to compute the validation error's variance, which is expressed using the mean and the variance of the model's generalization error and the size of the actual validation set used at the validation point.

Third, using the above-mentioned relationship and maximum likelihood estimation [134], we estimate the trend curve and the variance of the random noise. To the best of our knowledge, this is the first time that maximum likelihood estimation is employed for progress indication. The likelihood function is the product of multiple integrals, which are difficult to be used directly for numerical optimization. To overcome this hurdle, for each integral, we use the probability density function of a normal distribution to approximate a key component of the integrand. In this way, we acquire a simplified form of the likelihood function, which is easy to use for numerical optimization.

We implemented our improved progress indication method in TensorFlow [66], an open-source software package for deep learning. We present our performance test results for both convolutional and recurrent neural networks. Our results show that compared with using our basic method, using this improved method reduces the progress indicator's prediction error of the model training time

left by 57.5% on average. Also, with a low overhead, this improved method enables us to obtain relatively accurate progress estimates faster.

4.5.2 *Overview of the Improved Progress Indication Method*

This section provides an overview of the improved progress indication method for deep learning model training. To obtain relatively accurate progress estimates faster, we judiciously insert extra validation points between the original validation points. Using the validation errors obtained at both the original and the added validation points that we have encountered so far, we revise the predicted model training cost at both the original and the added validation points. Consequently, compared with our basic progress indication method, our improved progress indication method starts revising the predicted model training cost earlier and revises the predicted model training cost more frequently. This helps the progress indicator reduce its prediction error of the remaining model training time and obtain relatively accurate progress estimates faster.

Our basic progress indication method roughly approximates the model training cost as the sum of two components: the cost to process the training instances and the cost to calculate the validation errors at the original validation points. In addition to these two components, our improved progress indication method adds a third component to the model training cost: the cost to calculate the validation errors at the added validation points. Our discussion of the model training cost focuses on these three dominating components.

Similar to our basic progress indication method (see Section 4.4.2), to predict the model training cost, we mainly need to predict n_v, the number of original validation points needed for model training. When model training starts, we estimate n_v to be v_{max}, the maximum number of original validation points allowed for model training. We deem the validation curve to be the sum of some zero-mean random noise and a smooth trend curve. Our improved progress indication method uses four

parameters to estimate the trend curve and the variance of the random noise (see Section 4.5.6). Since at least $\tau_v = 4$ data points are needed to estimate the four parameters, we refine the estimated n_v only when we reach a validation point whose sequence number is $\geq \tau_v$ and where the early stopping condition is unfulfilled.

A good progress indicator should have a low run-time overhead [97]. In our improved progress indication method, a large part of the progress indicator's run-time overhead comes from computing the validation error at the added validation points. To lower this part of the run-time overhead, at each added validation point, we calculate the validation error on a randomly sampled subset of the full validation set rather than on the full validation set. The sampling is done without replacement. The subset is usually much smaller than the full validation set and could be biased. If we keep using the same biased subset at each added validation point, the bias could have a large negative impact on our estimation accuracy of the trend curve, the variance of the random noise, and subsequently the model training cost. To address this issue, we re-sample the full validation set to obtain a new subset at each added validation point to calculate the validation error. Each subset includes the same number V' of data instances. At each original validation point, we use the full validation set to calculate the validation error.

The random noise's variance depends on the size of the actual validation set used at the validation point. We use an indirect approach to derive the relationship between these two numbers. Using this relationship, the validation curve obtained so far, and maximum likelihood estimation [134], we estimate the trend curve and the variance of the random noise for future validation points. We use the Monte Carlo simulation approach the same as in our basic progress indication method (see Section 4.4.5) to predict n_v, the number of original validation points needed for model training. Finally, we revise the predicted model training cost based upon the projected n_v.

4.5.3 *Our Approach to Insert Extra Validation Points between the Original Validation Points*

This section describes our approach to insert extra validation points between the original validation points. We regard the beginning of model training as the 0-th original validation point, although the validation error is not computed there. For each pair of successive original validation points, we insert extra validation points evenly between them. More specifically, recall that g denotes the number of batches of model training between two consecutive original validation points. v_{max} denotes the maximum number of original validation points allowed for model training. n_j ($0 \leq j \leq v_{max} - 1$) denotes the number of validation points to be added between the j-th and the (j+1)-th original validation points. When $j = 0$, n_0 denotes the number of validation points to be added before the first original validation point. We ensure that n_j is $\leq g - 1$ for every j between 0 and $v_{max} - 1$. Starting from the j-th original validation point, we do

$$\lfloor kg / (n_j + 1) \rceil$$

batches of model training to reach the k-th ($1 \leq k \leq n_j$) of the n_j validation points added between the j-th and the (j+1)-th original validation points. Here, $\lfloor \rceil$ is the nearest integer function, e.g., $\lfloor 4.4 \rceil = 4$ and $\lfloor 4.6 \rceil = 5$.

The rest of this section is organized in the following way. Section 4.5.3.1 provides an overview of how we set n_j ($0 \leq j \leq v_{max} - 1$), the number of validation points to be added between the j-th and the (j+1)-th original validation points. Section 4.5.3.2 describes how to set n_0, the number of validation points to be added before the first original validation point. Section 4.5.3.3 shows how to set q, the constant regulating the decay rate of n_j ($0 \leq j \leq v_{max} - 1$) in the exponential decay schema.

4.5.3.1 Overview of how we set n_j ($0 \leq j \leq v_{max} - 1$)

This section provides an overview of how we set n_j ($0 \leq j \leq v_{max} - 1$), the number of validation points to be added between the j-th and the (j+1)-th original validation points.

Recall that n_v denotes the number of original validation points needed for model training. Our initial estimate of n_v is usually inaccurate and is not refined until we reach the fourth validation point. As we accumulate more data points over time, our estimate of n_v tends to become more accurate. To refine our initial estimate of n_v as soon as possible and to obtain relatively accurate estimates of n_v faster, we insert more validation points for use at the early stages of model training than at the later stages of model training. In other words, we decrease n_j ($0 \leq j \leq v_{max} - 1$), the number of validation points to be added between the j-th and the (j+1)-th original validation points, as j increases. Furthermore, we want n_0, the number of validation points to be added before the first original validation point, to be reasonably large. This is particularly the case when a non-trivial progress indicator is most needed: the training set is large, many batches of model training are performed between two successive validation points, and model training takes a long time.

One could decrease n_j either linearly or exponentially as j increases. For our purpose, exponential decay works better than linear decay. To compare these two decay schemata of n_j and show this, we consider two model training processes that have the same setting except for the decay schema used. Recall that n_0 denotes the number of validation points to be added before the first original validation point. v_{max} is the maximum number of original validation points allowed for model training. One model training process uses the exponential decay schema, where

$$n_j = \lfloor n_0 q^j \rceil \ (1 \leq j \leq v_{max} - 1),$$

q ($0 \leq q < 1$) is a constant regulating the decay rate of n_j, and 0^0 is defined to be 1. The other model training process uses the linear decay schema, where

$$n_j = \max(\lceil n_0 - jz \rceil, 0)\ (1 \leq j \leq v_{max} - 1)$$

and z is a constant > 0 regulating the decay rate of n_j. Given the same mean cost of calculating the validation error at each added validation point, the total cost of calculating the validation errors at all added validation points is $\propto$ the total number of validation points added between the original validation points. To have the same total cost of calculating the validation errors at all added validation points, in the two model training processes we insert the same total number of validation points between the original validation points. For a sufficiently large v_{max}, the total number of validation points added between the original validation points is roughly

$$\sum_{j=0}^{+\infty} n_0 q^j = n_0/(1-q)$$

and

$$\sum_{j=0}^{\lfloor n_0/z \rfloor}(n_0 - jz) \approx n_0^2/(2z)$$

for the exponential decay schema and the linear decay schema, respectively. Recall that we want n_0 to be reasonably large. Thus, we expect the n_0 used in the linear decay schema to be typically > $2z / (1 - q)$. In this case, the n_0 used in the exponential decay schema is larger than the n_0 used in the linear decay schema. Adopting a larger n_0 makes the early stage of model training include more added validation points, which is what we want. Thus, we employ the exponential decay schema instead of the linear decay schema. In the exponential decay schema, once n_0 and q are set using the approach given in Sections 4.5.3.2 and 4.5.3.3, respectively, n_j is known for each j between 0 and $v_{max} - 1$.

4.5.3.2 Setting n_0

In this section, we describe how to set n_0, the number of validation points to be added before the first original validation point. When setting n_0, we try to fulfill the following two requirements if possible:

1) **Requirement 1**: When we finish the work at the fourth validation point, the model training cost that has been incurred is $\leq C$ units of work, where C is a pre-set number > 0. Requirement 1 is used to control the amount of time that elapses before we refine our beginning estimate of the model training cost for the first time at the fourth validation point. This amount should not be too large.
2) **Requirement 2**: From when model training starts to the time we finish the work at the first original validation point, the cost to calculate the validation errors at the added validation points is $\leq c_0P_1$. Here, P_1 is a pre-set percentage > 0. c_0 denotes the model training cost that has been incurred when we finish the work at the first original validation point, excluding the progress indicator's overhead of calculating the validation errors at the added validation points. That is, c_0 is = the cost to process the training instances before we reach the first original validation point + the cost to calculate the validation error at the first original validation point. Requirement 2 is used to control the progress indicator's overhead that has been incurred for calculating the validation errors at the added validation points when we finish the work at the first original validation point. This overhead should not be too large.

These two requirements are soft requirements, as it may not always be possible to fully fulfill both requirements.

We have two considerations when setting the value of C in Requirement 1. On one hand, to prevent the user of the deep learning software from waiting too long before our beginning estimate

of the model training cost is refined for the first time at the fourth validation point, we do not want C to be too large. On the other hand, the smaller the C, the more validation points need to be added before the first original validation point, and subsequently due to Requirement 2, the smaller the cost of calculating the validation error at an added validation point can be. At each added validation point, the cost to calculate the validation error is $\propto$ the size of the randomly sampled subset of the full validation set used to calculate the validation error. If C is too small, this subset will not be large enough for reasonably estimating the model's generalization error. This will lower the progress indicator's projection accuracy of the model training cost and is undesirable. To strike a balance between the two considerations, we set C's default value to 20,000 × the number of GPUs, TPUs, or CPUs used to train the model. This allows a non-trivial number of batches of model training to appear between two consecutive validation points, as a batch of model training typically involves much < 20,000 / 4 = 5,000 units of work on any GPU, TPU, or CPU.

We have two considerations when setting the value of P_1 in Requirement 2. On one hand, we want P_1 to be small so that the progress indicator does not cause a large increase in the model training cost during the period from when model training starts to the time we finish the work at the first original validation point. On the other hand, if P_1 is too small, at each added validation point, the randomly sampled subset of the full validation set used to calculate the validation error will not be large enough for reasonably estimating the model's generalization error. This is undesirable. There is also no need to make P_1 too small. Recall that n_j ($0 \leq j \leq v_{max} - 1$) denotes the number of validation points to be added between the j-th and the (j+1)-th original validation points. As n_j decreases as j increases, the progress indicator's overhead of calculating the validation errors at the validation points added before the first original validation point can be amortized over time during model training. To strike a balance between the two considerations, we set the default value of P_1 to 5%.

Recall that c_0 is the model training cost that has been incurred when we finish the work at the first original validation point, excluding the progress indicator's overhead of calculating the validation errors at the added validation points. n_0 denotes the number of validation points to be added before the first original validation point. We first compute c_0 and then decide the value of n_0.

Computing c_0

Recall that g denotes the number of batches of model training between two consecutive original validation points. B is the number of training instances in every batch. c_0 is the sum of two parts. The first part is the cost to process the training instances before we reach the first original validation point

= the number of batches of model training before the first original validation point × the number of training instances in every batch × the mean amount of work taken to process a training instance one time in model training
= $g \times B \times 1$
= gB.

In Section 4.4.2, we show that the mean amount of work taken to process a validation instance one time to calculate the validation error is 1/3 unit of work. Recall that V is the number of data instances that are in the full validation set. The second part of c_0 is c_v, the cost to calculate the validation error at the first original validation point. c_v is

= the number of data instances that are in the full validation set × the mean amount of work taken to process a validation instance one time to calculate the validation error
= $V/3$.

Adding the two components, we have $c_0 = gB + V/3$.

Deciding the value of n_0

Recall that c_0 is the model training cost that has been incurred when we finish the work at the first original validation point, excluding the progress indicator's overhead of calculating the validation errors at the added validation points. P_1 is the maximum allowed percentage increase in the model training cost that the progress indicator causes during the period from when model training starts to the time we finish the work at the first original validation point. C is the upper threshold of the model training cost that has been incurred when we finish the work at the fourth validation point. c_v is the cost to calculate the validation error at the first original validation point. n_0 denotes the number of validation points to be added before the first original validation point.

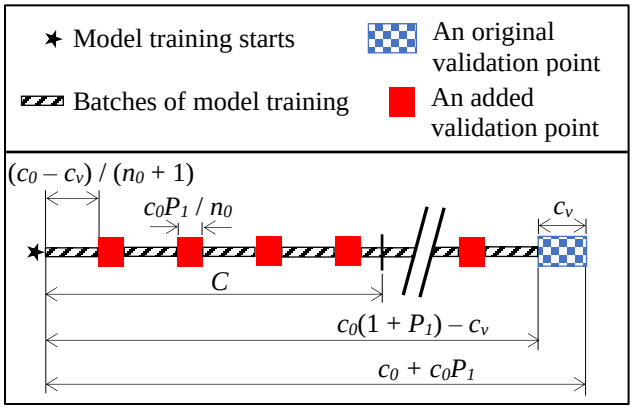


Figure 4.2. Decomposition of the model training cost that has been incurred when we finish the work at the first original validation point.

When setting n_0, we try to fulfill Requirements 1 and 2 mentioned above if possible. In attempting to fulfill Requirement 2, we can aim the cost to calculate the validation errors at the n_0 validation points added before the first original validation point to be c_0P_1. There are two possible cases:

1) **Case 1**: The model training cost that has been incurred when we are just about to arrive at the first original validation point is $\geq C$ (see Figure 4.2). That is,

$$c_0 + c_0P_1 - c_v = c_0(1 + P_1) - c_v$$
$$\geq C.$$

In this case, we show that if n_0 is set to

$$\lceil 4[c_0(1 + P_1) - c_v] / C \rceil$$

that is ≥ 4, Requirement 1 is fulfilled. Here, $\lceil \rceil$ is the ceiling function, e.g., $\lceil 4.4 \rceil = 5$. We note that:

a) The cost to calculate the validation error at each of the n_0 validation points added before the first original validation point is c_0P_1 / n_0.

b) The cost to process the training instances that has been incurred when we are just about to arrive at the first original validation point is $c_0 - c_v$, which is > 0. With n_0 validation points inserted before it, the first original validation point is the $(n_0 + 1)$-th validation point. Thus, before we finish the work at the first original validation point, the cost to process the training instances between two successive validation points is $(c_0 - c_v) / (n_0 + 1)$.

The fourth validation point is the fourth validation point added before the first original validation point. The model training cost that has been incurred when we finish the work at the fourth validation point is the sum of two components:

a) $4c_0P_1 / n_0$, the cost to calculate the validation errors at the first four validation points added before the first original validation point; and

b) $4(c_0 - c_v) / (n_0 + 1)$, the cost to process the training instances before we reach the fourth validation point.

Adding these two components, we get

the model training cost that has been incurred when we finish the work at the fourth validation point

$$= 4c_0P_1 / n_0 + 4(c_0 - c_v) / (n_0 + 1)$$
$$< 4c_0P_1 / n_0 + 4(c_0 - c_v) / n_0$$
$$= 4[c_0(1 + P_1) - c_v] / n_0$$
$$= C \times 4[c_0(1 + P_1) - c_v] / C / \lceil 4[c_0(1 + P_1) - c_v] / C \rceil$$
$$\leq C.$$

This verifies that Requirement 1 is fulfilled.

2) **Case 2:** The model training cost that has been incurred when we are just about to arrive at the first original validation point is < C. That is,

$$c_0(1 + P_1) - c_v < C.$$

In this case, if n_0 is set to 4, the fourth validation point is the fourth validation point added before the first original validation point. The model training cost that has been incurred when we finish the work at the fourth validation point is < that when we are just about to arrive at the first original validation point, and thus is < C. This shows that Requirement 1 is fulfilled.

Recall that g denotes the number of batches of model training between two consecutive original validation points. At least one batch of model training needs to occur between two successive validation points. Thus, n_0 cannot exceed $g - 1$. To fulfill this, we set n_0 to

$$\min(\lceil 4[c_0(1 + P_1) - c_v] / C \rceil, g - 1)$$

if $c_0(1 + P_1) - c_v$ is $\geq C$. Otherwise, if $c_0(1 + P_1) - c_v$ is < C, we set n_0 to $\min(4, g - 1)$.

4.5.3.3 Setting q

In this section, we show how to set q, the constant regulating the decay rate of n_j ($0 \leq j \leq v_{max} - 1$) in the exponential decay schema. Recall that v_{max} denotes the maximum number of original validation points allowed for model training. n_j ($0 \leq j \leq v_{max} - 1$) denotes the number of validation points to be added between the j-th and the (j+1)-th original validation points. n_0 denotes the number of validation

points to be added before the first original validation point. In the exponential decay schema, $n_j = \lfloor n_0 q^j \rceil$ $(0 \leq j \leq v_{max} - 1)$.

Let p_j $(1 \leq j \leq v_{max})$ denote the percentage increase in the model training cost that the progress indicator causes during the period from when model training starts to the time we finish the work at the j-th original validation point. When setting q, we try to fulfill the following requirement if possible:

Requirement 3: $p_{v_{max}}$ is $\leq P_v$, where P_v is a pre-set percentage > 0.

This requirement is a soft requirement, as it may not always be possible to fully fulfill this requirement.

The increase in the model training cost caused by the progress indicator comes from calculating the validation errors at the added validation points. Since the same number of validation instances are used to calculate the validation error at each added validation point, the cost to calculate the validation error at an added validation point is a constant. Thus, during the period from when model training starts to the time we finish the work at the j-th $(1 \leq j \leq v_{max})$ original validation point, the increase in the model training cost caused by the progress indicator is $\propto \sum_{k=0}^{j-1} n_k$, the total number of validation points added before the j-th original validation point. During the same period, the model training cost excluding the progress indicator's overhead of calculating the validation errors at the added validation points is $\propto j$, as both the cost to process the training instances between two successive original validation points and the cost to calculate the validation error at an original validation point are constants. As the ratio of the increase in the model training cost caused by the progress indicator to the model training cost excluding the progress indicator's overhead, p_j $(1 \leq j \leq v_{max})$ is

$$\propto \sum_{k=0}^{j-1} n_k / j$$

$$= \sum_{k=0}^{j-1} \lfloor n_0 q^k \rfloor / j. \quad (4.2)$$

As j increases, n_j and subsequently p_j strictly decrease. Thus, P_v in Requirement 3 should be < P_1, the maximum allowed percentage increase in the model training cost that the progress indicator causes during the period from when model training starts to the time we finish the work at the first original validation point. In addition, we have two other considerations when setting the value of P_v. On the one hand, we want P_v to be small, as a good progress indicator should have a low run-time overhead [97]. On the other hand, the larger the P_v, the more validation points we can add before model training finishes. This helps us obtain more accurate progress estimates for the model training process. To strike a balance between these two considerations, we set the default value of P_v to 0.5%.

Recall that when deciding the value of n_0, we aim p_1 to be = P_1 in attempting to fulfill Requirement 2. In the following derivation used to set q, we regard p_1 to be = P_1. There are two possible cases: 1) v_{max} is < P_1 / P_v and 2) v_{max} is ≥ P_1 / P_v. We discuss the two cases sequentially.

Case 1: v_{max} is < P_1 / P_v

We first discuss the case when v_{max} is < P_1 / P_v. Recall that v_{max} denotes the maximum number of original validation points allowed for model training. n_j ($0 \leq j \leq v_{max} - 1$) denotes the number of validation points to be added between the j-th and the (j+1)-th original validation points. q ($0 \leq q < 1$) is the constant regulating the decay rate of n_j in the exponential decay schema. P_1 is the maximum allowed percentage increase in the model training cost that the progress indicator causes during the period from when model training starts to the time we finish the work at the first original validation point. p_j ($1 \leq j \leq v_{max}$) is the percentage increase in the model training cost that the progress indicator causes during the period from when model training starts to the time we finish the work at the j-th original validation point. We regard p_1 to be = P_1.

Formula (4.2) shows that p_j ($1 \leq j \leq v_{max}$) is

$$\propto \sum_{k=0}^{j-1} \lfloor n_0 q^k \rceil / j.$$

For $j = v_{max}$, we have

$$p_{v_{max}} \propto \sum_{k=0}^{v_{max}-1} \lfloor n_0 q^k \rceil / v_{max}.$$

For $j = 1$, we have

$$p_1 \propto n_0 / 1.$$

When q is 0, $p_{v_{max}}$ reaches its smallest value, which is $\propto n_0 / v_{max}$ and is $= p_1 / v_{max} = P_1 / v_{max}$. When v_{max} is $< P_1 / P_v$, $p_{v_{max}}$ must be $> P_v$. Requirement 3 cannot be fully fulfilled. To minimize $p_{v_{max}}$ and fulfill Requirement 3 as much as possible, we set q to 0.

Case 2: v_{max} is $\geq P_1 / P_v$

Next, we discuss the case when v_{max} is $\geq P_1 / P_v$. When v_{max} is $= P_1 / P_v$, we set q to 0 to let $p_{v_{max}}$ reach its smallest value $P_1 / v_{max} = P_v$ and fulfill Requirement 3. When v_{max} is $> P_1 / P_v$, we proceed as follows.

Formula (4.2) shows that p_j ($1 \leq j \leq v_{max}$) is

$$\begin{aligned} &\propto \sum_{k=0}^{j-1} \lfloor n_0 q^k \rceil / j \\ &\approx \sum_{k=0}^{j-1} n_0 q^k / j. \end{aligned} \tag{4.3}$$

For $j = v_{max}$, we roughly have

$$p_{v_{max}} \propto \sum_{k=0}^{v_{max}-1} n_0 q^k / v_{max}. \tag{4.4}$$

For $j = 1$, we have

$$p_1 \propto n_0 / 1. \tag{4.5}$$

Dividing each side of formula (4.4) by the corresponding side of formula (4.5), we roughly have

$$p_{v_{max}}/p_1 = \sum_{k=0}^{v_{max}-1} q^k \, /v_{max}. \tag{4.6}$$

Regarding p_1 to be = P_1 and rearranging formula (4.6) lead to

$$\sum_{k=0}^{v_{max}-1} q^k - v_{max} p_{v_{max}}/P_1 = 0.$$

If we make the function of q

$$\begin{aligned} f(q) &\stackrel{\text{def}}{=} \sum_{k=0}^{v_{max}-1} q^k - v_{max} P_v/P_1 \\ &= 0, \end{aligned}$$

we can have $p_{v_{max}} = P_v$ and fulfill Requirement 3. Recall that $P_1 > P_v > 0$. The following theorem holds.

Theorem. For any $v_{max} > P_1 / P_v$, $f(q)$ must have a unique root q in (0, 1).

Proof. For each k ($1 \leq k \leq v_{max} - 1$), q^k is continuous and strictly increasing on [0, 1]. Thus, $f(q)$ is continuous and strictly increasing on [0, 1].

$$f(0) = 1 - v_{max} P_v / P_1$$

is < 0 because v_{max} is > P_1 / P_v.

$$f(1) = v_{max} - v_{max} P_v / P_1$$

is > 0 because P_1 is > P_v. According to the intermediate value theorem [135], $f(q)$ must have a root in (0, 1). As $f(q)$ is strictly increasing on [0, 1], this root is unique. ∎

For any $q \neq 1$, $f(q)$ is

$$= (1 - q^{v_{max}})/(1 - q) - v_{max} P_v/P_1.$$

We use the bisection method to find $f(q)$'s unique root in (0, 1) and set q to this root.

In summary, we set q to 0 if v_{max} is $\leq P_1 / P_v$. Otherwise, if v_{max} is $> P_1 / P_v$, we set q to $f(q)$'s unique root in (0, 1).

The shape of p_j as a function of j

Recall that p_j ($1 \leq j \leq v_{max}$) strictly decreases as j increases. In this section, we show that p_j decreases quickly as j increases, indicating that the progress indicator usually has a low run-time overhead.

When v_{max} is $\leq P_1 / P_v$, q is set to 0. Formula (4.2) shows that p_j ($1 \leq j \leq v_{max}$) is

$$\begin{aligned} &\propto \sum_{k=0}^{j-1} \lfloor n_0 q^k \rceil / j \\ &= n_0 / j. \end{aligned}$$

For $j = 1$, we have

$$p_1 \propto n_0 / 1.$$

Thus, $p_j = p_1 / j$. This is a rapidly decreasing function of j. Typically, the patience p in the early stopping condition is ≥ 2. When the early stopping condition is fulfilled, we have encountered ≥ 3 original validation points (i.e., $j \geq 3$) and p_j is $\leq 5\% / 3 \approx 1.7\%$ if p_1 is $= P_1 = 5\%$.

When v_{max} is $> P_1 / P_v$, q is set to a number in (0, 1). Formula (4.3) shows that p_j ($1 \leq j \leq v_{max}$) is roughly

$$\begin{aligned} &\propto \sum_{k=0}^{j-1} n_0 q^k / j \\ &= n_0 (1 - q^j) / (1 - q) / j \\ &< n_0 / (1 - q) / j. \end{aligned}$$

Since p_1 is $\propto n_0 / 1$, p_j decreases faster than $p_1 / (1 - q) / j$ as j increases. Figure 4.3 shows a typical shape of p_j as a function of j.

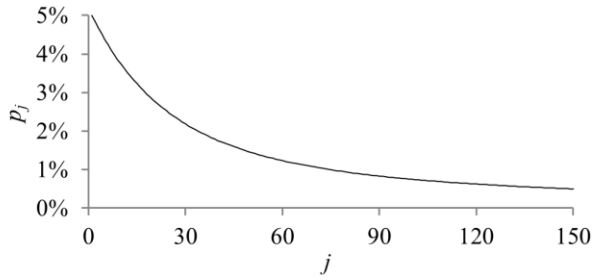


Figure 4.3. A typical shape of p_j as a function of j.

4.5.4 *Setting V′*

At each added validation point, we use a distinct randomly sampled subset of the full validation set to calculate the validation error. Every subset contains the same number of data instances. In this section, we show how to set V', the number of data instances that are in the subset.

In Section 4.4.2, we show that the mean amount of work taken to process a validation instance one time to calculate the validation error is 1/3 unit of work. The cost to calculate the validation errors at the n_0 validation points added before the first original validation point is

= n_0 × the number of data instances that are in the randomly sampled subset of the full validation set used at each added validation point × the mean amount of work taken to process a validation instance one time to calculate the validation error

= $n_0 V'/3$.

Recall that c_0 is the model training cost that has been incurred when we finish the work at the first original validation point, excluding the progress indicator's overhead of calculating the validation errors at the added validation points. P_1 is the maximum allowed percentage increase in the model training cost that the progress indicator causes during the period from when model training starts to the time we finish the work at the first original validation point. If we set

$$V' = \lfloor c_0 P_1 / n_0 / (1/3) \rceil$$
$$= \lfloor 3 c_0 P_1 / n_0 \rceil,$$

we have $n_0V'/3 \approx c_0P_1$ fulfilling Requirement 2.

As described in Sections 4.5.6.1 and 4.5.7, our estimation method of the trend curve and the variance of the random noise requires V' to be $\geq$ a threshold V_{min}. This may occasionally cause Requirement 2 to be not fully fulfilled. Moreover, V' should be $\leq V$, the number of data instances that are in the full validation set. Given all the above considerations, we set

$$V' = \min(\max(\lfloor 3c_0P_1 / n_0 \rceil, V_{min}), V). \quad (4.7)$$

4.5.5 *Relationship between the Random Noise's Variance and the Size of the Actual Validation Set Used at the Validation Point*

At each original validation point, the actual validation set used is the full validation set. At each added validation point, the actual validation set used is a randomly sampled subset of the full validation set. Recall that we deem the validation curve to be the sum of some zero-mean random noise and a smooth trend curve. The random noise's variance depends on the size of the actual validation set used at the validation point. The relationship between these two numbers is previously unknown and difficult to be derived directly. However, we need to know this relationship in order to use both the original and the added validation points to predict when early stopping will occur. Noting that the random noise's variance is equal to the validation error's variance, we use an indirect approach to derive this relationship in two steps:

1) **Step 1:** Compute the conditional mean and the conditional variance of the validation error given the model's generalization error [133], both of which can be expressed using the model's generalization error and the size of the actual validation set used at the validation point.
2) **Step 2:** Use the conditional mean, the conditional variance, and the law of total variance [134] to compute the validation error's variance, which is expressed using the mean and the variance

of the model's generalization error and the size of the actual validation set used at the validation point.

In the following, we first define a model's generalization error and then present the two steps sequentially.

A model's generalization error

For a classification task, a model's generalization error is defined as the probability that a data instance is misclassified by the model [133]. A deep learning model's generalization error at any validation point is a random variable, as three factors introduce randomness into the model training process. First, the model is trained in batches using stochastic gradient descent (SGD) [62]. Each batch processes B training instances randomly chosen from the training set. Second, the weights of the neural network model are frequently randomly initialized [62]. Third, dropout [92] is often used in model training. When using dropout, in every batch of model training, we randomly omit some nodes along with their connections of the neural network model.

Step 1: Compute the conditional mean and the conditional variance of the validation error given the model's generalization error

Let V_j ($V_j \geq 1$) denote the number of data instances that are in the actual validation set used at the j-th validation point. If the j-th validation point is an original validation point, V_j is = V, the number of data instances that are in the full validation set. If the j-th validation point is an added validation point, V_j is = V', the uniform number of data instances that are in the randomly sampled subset of the full validation set used at each added validation point. Let e_j ($0 \leq e_j \leq 1$) denote the model's generalization error at the j-th validation point, c_j denote the number of validation instances

that are misclassified by the model and in the actual validation set used at the *j*-th validation point, and

$$\hat{e}_j = c_j/V_j \ (0 \leq \hat{e}_j \leq 1) \tag{4.8}$$

denote the validation error of the model at the *j*-th validation point. As an estimate of e_j, $\hat{e}_j$ is a discrete random variable.

A standard assumption used in machine learning is that all data instances are independently and identically sampled from an underlying distribution [133]. The probability that a data instance is misclassified by the model is e_j. Given e_j, c_j follows a binomial distribution. Its probability mass function is

$$P(c_j|e_j) = \binom{V_j}{c_j} e_j^{c_j}(1-e_j)^{V_j-c_j}. \tag{4.9}$$

The conditional mean and the conditional variance of c_j given e_j are $E(c_j|e_j) = V_j e_j$ and $Var(c_j|e_j) = V_j e_j(1 - e_j)$, respectively. From formulas (4.8) and (4.9), we have

$$\begin{aligned} E(\hat{e}_j|e_j) &= E(c_j|e_j)/V_j \\ &= e_j \end{aligned} \tag{4.10}$$

and

$$\begin{aligned} Var(\hat{e}_j|e_j) &= Var(c_j|e_j)/V_j^2 \\ &= e_j(1-e_j)/V_j. \end{aligned} \tag{4.11}$$

Step 2: Compute the validation error's variance

Recall that V_j ($V_j \geq 1$) denotes the number of data instances that are in the actual validation set used at the *j*-th validation point. $\hat{e}_j$ denotes the validation error of the model at the *j*-th validation point. e_j denotes the model's generalization error at the *j*-th validation point. Let μ_j ($0 \leq \mu_j \leq 1$) and

σ_j^2 denote the mean and the variance of e_j, respectively. Given two random variables X and Y, the law of total variance [134] is

$$Var(X) = E[Var(X|Y)] + Var[E(X|Y)].$$

We have

$$\begin{aligned} Var(\hat{e}_j) &= E[Var(\hat{e}_j|e_j)] + Var[E(\hat{e}_j|e_j)] \\ &= E[e_j(1-e_j)/V_j] + Var(e_j) \quad \text{(plug in formulas (4.10) and (4.11))} \\ &= [E(e_j) - E(e_j^2)]/V_j + \sigma_j^2 \\ &= [\mu_j - (Var(e_j) + E(e_j)^2)]/V_j + \sigma_j^2 \quad \text{(as } Var(X) = E(X^2) - E(X)^2\text{)} \\ &= (\mu_j - \sigma_j^2 - \mu_j^2)/V_j + \sigma_j^2 \\ &= (\mu_j - \mu_j^2)/V_j + (1 - 1/V_j)\sigma_j^2. \end{aligned} \tag{4.12}$$

At the j-th validation point, the variance of the random noise is = $Var(\hat{e}_j)$ computed by formula (4.12).

4.5.6 *Estimating the Trend Curve and the Variance of the Random Noise for Future Validation Points*

Recall that we re-estimate the number of original validation points needed for model training only when we reach a validation point whose sequence number is $\geq \tau_v$ and where the early stopping condition is unfulfilled. In this section, we show at such a validation point, how to estimate the trend curve and the variance of the random noise for future validation points. To do this, we need to only estimate for each $j \geq 1$, the mean μ_j and the variance σ_j^2 of the model's generalization error at the j-th validation point. Once μ_j and σ_j^2 are obtained, the random noise's variance at the j-th validation point can be computed by formula (4.12). Moreover, the trend curve's value at the j-th validation point is = μ_j. To show this, recall that $\hat{e}_j$ is the validation error of the model at the j-th

validation point. e_j is the model's generalization error at the j-th validation point. We deem the validation curve to be the sum of some zero-mean random noise and a smooth trend curve. The trend curve's value at the j-th validation point is $= E(\hat{e}_j)$. Given two random variables X and Y, the law of total expectation [134] is

$$E(X) = E[E(X|Y)].$$

We have

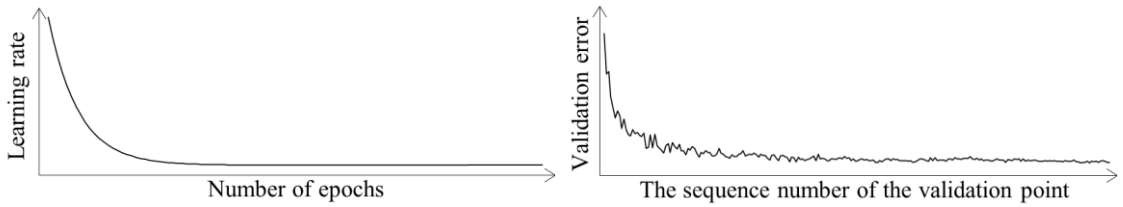
$$\begin{aligned} E(\hat{e}_j) &= E[E(\hat{e}_j|e_j)] \\ &= E(e_j) \qquad \text{(plug in formula (4.10))} \\ &= \mu_j. \end{aligned}$$

We use maximum likelihood estimation [134] to estimate μ_j and σ_j^2. To the best of our knowledge, this is the first time that maximum likelihood estimation is used for progress indication. We consider three cases: 1) a continuous decay method is applied to the learning rate, 2) a constant learning rate is adopted, and 3) a step decay method is applied to the learning rate. The three cases are handled in Sections 4.5.6.1 to 4.5.6.3, respectively.

4.5.6.1 Estimating μ_j and σ_j^2 when a continuous decay method is applied to the learning rate

This section describes how to estimate for each $j \geq 1$, the mean μ_j and the variance σ_j^2 of the model's generalization error at the j-th validation point when the learning rate changes over time based upon a continuous decay method. In such a decay schedule, the learning rate continuously decreases over epochs. For instance, in an exponential decay method, the learning rate adopted in the k-th epoch ($k \geq 1$) is $r_0 e^{-(k-1)\rho}$ (see Figure 4.4(A)). Here, $\rho > 0$ is a constant regulating the decay rate of the learning rate. $r_0 > 0$ is the beginning learning rate. Figure 4.4(B) shows a typical validation curve in this case. To estimate μ_j and σ_j^2, we need to estimate only four parameters: a, b, and c used to model μ_j and λ

used to model σ_j^2. In the following, we introduce these four parameters and then show how to estimate them.



(A) (B)

Figure 4.4. The learning rate over epochs and a typical validation curve when an exponential decay schedule for the learning rate is used.
(A) The learning rate over epochs. (B) A typical validation curve.

a, b, and c used to model μ_j

As in our basic progress indication method (see Section 4.4.3), we use an inverse power function [97, 127-130] to model the trend curve. Recall that the trend curve's value at the *j*-th validation point is = μ_j, the mean of the model's generalization error at the *j*-th validation point. Thus, we have

$$\mu_j = ax_j^{-b} + c, \tag{4.13}$$

where *a* is > 0, *b* is > 0, *c* is > 0, *j* is the validation point's sequence number, and x_j is the normalized number of batches of model training finished before the *j*-th validation point

$\stackrel{\text{def}}{=}$ the number of batches of model training finished before the *j*-th validation point / the number of batches of model training between two consecutive original validation points.

To estimate μ_j, we need to estimate only *a*, *b*, and *c*.

λ used to model σ_j^2

The variance of the model's generalization error varies with the learning rate. The learning rate regulates how much the weights of the neural network and therefore the model's generalization error change over time as well as due to random variations. The larger the learning rate, the larger the changes are likely to be. When the learning rate is 0, neither the weights of the neural network nor the model's generalization error would ever differ from their initial values. In this case, the variance of the model's generalization error is 0. Based upon this insight, we deem the standard deviation and the variance of the model's generalization error to be approximately $\propto$ the learning rate and its square, respectively. Let $\lambda > 0$ denote the ratio of the variance of the model's generalization error to the square of the learning rate. Let r_j denote the learning rate right before the j-th validation point. The variance of the model's generalization error at the j-th validation point is modelled by

$$\sigma_j^2 = \lambda r_j^2. \tag{4.14}$$

For each $j \geq 1$, r_j is known. To estimate σ_j^2, we need to estimate only λ.

Overview of estimating the parameters a, b, c, and λ

We use maximum likelihood estimation [134] to estimate the parameters a, b, c, and λ. The likelihood function is the product of multiple integrals, which are difficult to be used directly for numerical optimization. To overcome this hurdle, for each integral, we use the probability density function of a normal distribution to approximate a key component of the integrand. In this way, we acquire a simplified form of the likelihood function, which is easy to use for numerical optimization.

In the following, we show how to estimate the parameters a, b, c, and λ in six steps. First, we present the likelihood function as the product of multiple probabilities. Second, we express each

probability as an integral. Third, we show how to approximate a key component of the integrand of the integral. Fourth, we give a simplified expression of the probability. Fifth, we describe the constrained numerical optimization problem for maximizing the likelihood function and estimating a, b, c, and λ. Finally, we discuss the software package and its setting used to do numerical optimization.

The likelihood function

We employ the validation curve up to the present validation point to estimate the parameters a, b, c, and λ. These parameters are then adopted to estimate the trend curve and the variance of the random noise for future validation points based upon formulas (4.12), (4.13), and (4.14). As an intuition, the validation points long before the present validation point may not well manifest the validation curve's trend for future validation points and could be unsuited for estimating a, b, c, and λ. Like our basic progress indication method (see Section 4.4.3), to estimate a, b, c, and λ, we employ the last

$$w = \min(n, w')$$

validation points rather than all the validation points that we have reached so far. Here, n denotes the present validation point's sequence number. w' is a pre-chosen window size with a default value of 50.

Recall that $\hat{e}_j$ denotes the validation error of the model at the j-th validation point. We deem the validation curve to be the sum of some zero-mean random noise and a smooth trend curve. The trend curve's value at the j-th validation point is = μ_j. Let ε_j denote the random noise at the j-th validation point. We have

$$\hat{e}_j = \mu_j + \varepsilon_j.$$

We regard the random noises at distinct validation points to be independent of each other. Formula (4.13) shows that μ_j is a function of a, b, and c. The likelihood function that we want to maximize and covers the validation errors at the last w validation points is

$$\begin{aligned} & L(a,b,c,\lambda|\hat{e}_{n-w+1},\hat{e}_{n-w+2},\ldots,\hat{e}_n) \\ &= P(\hat{e}_{n-w+1},\hat{e}_{n-w+2},\ldots,\hat{e}_n;a,b,c,\lambda) \\ &= P(\mu_{n-w+1}+\varepsilon_{n-w+1},\ \mu_{n-w+2}+\varepsilon_{n-w+2},\ldots,\ \mu_n+\varepsilon_n;a,b,c,\lambda) \\ &= P(\varepsilon_{n-w+1},\varepsilon_{n-w+2},\ldots,\varepsilon_n;a,b,c,\lambda) \\ &= \prod_{j=n-w+1}^{n} P(\varepsilon_j;a,b,c,\lambda) \\ &= \prod_{j=n-w+1}^{n} P(\mu_j+\varepsilon_j;a,b,c,\lambda) \\ &= \prod_{j=n-w+1}^{n} P(\hat{e}_j;a,b,c,\lambda) \end{aligned} \tag{4.15}$$

Expressing $P(\hat{e}_j;a,b,c,\lambda)$ as an integral

Recall that $\hat{e}_j$ and e_j ($0 \le e_j \le 1$) are the validation error and the model's generalization error at the j-th validation point, respectively. Using the law of total probability and Bayes' theorem [134], we have

$$\begin{aligned} & P(\hat{e}_j;a,b,c,\lambda) \\ &= \int_0^1 P(\hat{e}_j,e_j;a,b,c,\lambda)de_j \\ &= \int_0^1 P(\hat{e}_j|e_j;a,b,c,\lambda)P(e_j;a,b,c,\lambda)de_j. \end{aligned} \tag{4.16}$$

Recall that μ_j and σ_j^2 are the mean and the variance of the model's generalization error at the j-th validation point, respectively. Formula (4.13) shows that μ_j is a function of a, b, and c. Formula (4.14) shows that σ_j^2 is a function of λ. We regard e_j to follow a normal distribution with mean μ_j and variance σ_j^2. That is,

$$P(e_j;a,b,c,\lambda) \ = \ P(e_j;\mu_j,\sigma_j^2)$$

$$= \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(e_j-\mu_j)^2}{2\sigma_j^2}\right) \quad (4.17)$$

Recall that c_j is the number of validation instances that are misclassified by the model and in the actual validation set used at the j-th validation point. V_j is the number of data instances that are in the actual validation set used at the j-th validation point. We have

$$\begin{aligned} & P(\hat{e}_j|e_j; a, b, c, \lambda) \\ &= P(c_j/V_j|e_j; a, b, c, \lambda) && \text{(plug in formula (4.8))} \\ &= P(c_j|e_j) \\ &= \binom{V_j}{c_j} e_j^{c_j}(1-e_j)^{V_j-c_j} && \text{(plug in formula (4.9))} \\ &= \binom{V_j}{V_j\hat{e}_j} e_j^{V_j\hat{e}_j}(1-e_j)^{V_j(1-\hat{e}_j)}. && (c_j = V_j\hat{e}_j \text{ based upon formula (4.8))} \end{aligned}$$

When maximizing the likelihood function, we can ignore the positive constant $\binom{V_j}{V_j\hat{e}_j}$ and focus on

$$P(\hat{e}_j|e_j; a, b, c, \lambda) \propto e_j^{V_j\hat{e}_j}(1-e_j)^{V_j(1-\hat{e}_j)}. \quad (4.18)$$

Plugging formulas (4.17) and (4.18) into formula (4.16), we get

$$\begin{aligned} & P\left(\hat{e}_j; a, b, c, \lambda\right) \\ &\propto \int_0^1 e_j^{V_j\hat{e}_j}(1-e_j)^{V_j(1-\hat{e}_j)} \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(e_j-\mu_j)^2}{2\sigma_j^2}\right) de_j. \end{aligned} \quad (4.19)$$

Approximating $e_j^{V_j\hat{e}_j}(1-e_j)^{V_j(1-\hat{e}_j)}$

Formula (4.15) shows that the likelihood function is the product of multiple integrals of the form given in formula (4.19). This form is difficult to be used directly for numerical optimization. To overcome the hurdle, for each integral, we use the probability density function of a normal distribution to approximate

$$e_j^{V_j \hat{e}_j}(1 - e_j)^{V_j(1-\hat{e}_j)},$$

a key component of the integrand. This enables us to obtain a simplified form of the integral, which is easy to use for numerical optimization.

Recall that V_j is the number of data instances that are in the actual validation set used at the j-th validation point. $\hat{e}_j$ and e_j ($0 \leq e_j \leq 1$) are the validation error and the model's generalization error at the j-th validation point, respectively. When we reach the j-th validation point, both V_j and $\hat{e}_j$ are known.

$$e_j^{V_j \hat{e}_j}(1 - e_j)^{V_j(1-\hat{e}_j)}$$

is $\propto$ a beta distribution's probability density function [134]

$$x^{\alpha-1}(1 - x)^{\beta-1} / \mathrm{B}(\alpha, \beta),$$

where $x = e_j$ ($0 \leq x \leq 1$) is the variable,

$$\alpha = V_j \hat{e}_j + 1,$$
$$\beta = V_j(1 - \hat{e}_j) + 1,$$

and $\mathrm{B}(\alpha, \beta)$ is a normalization constant. The mean and the variance of the beta distribution are

$$\mu_j' = \alpha/(\alpha + \beta)$$
$$= (V_j \hat{e}_j + 1)/(V_j + 2)$$

and

$$\sigma_j'^2 = \alpha\beta/[(\alpha + \beta)^2(\alpha + \beta + 1)]$$
$$= (V_j \hat{e}_j + 1)[V_j(1 - \hat{e}_j) + 1]/[(V_j + 2)^2(V_j + 3)],$$

respectively.

When α is ≥ 10 and β is ≥ 10, we can approximate the beta distribution by a normal distribution that has the same mean and variance as the beta distribution [136]. That is, we roughly have

$$e_j^{V_j\hat{e}_j}(1-e_j)^{V_j(1-\hat{e}_j)} \propto \frac{1}{\sqrt{\sigma_j'^2}}\exp\left(-\frac{(e_j-\mu_j')^2}{2\sigma_j'^2}\right). \tag{4.20}$$

Usually, V_j is large enough to make $\alpha \geq 10$ and $\beta \geq 10$. For example, even if $\hat{e}_j$ is as small as 0.02, having $V_j \geq 450$ is sufficient to make $\alpha \geq 10$ and $\beta \geq 10$. Occasionally for an j, which typically links to an added validation point, V_j may not be large enough to make $\alpha \geq 10$ and $\beta \geq 10$. In this case, we employ the approach described in Section 4.5.7 to increase V_j and make $\alpha \geq 10$ and $\beta \geq 10$ if possible. Regardless of whether α is ≥ 10 and β is ≥ 10, we always use formula (4.20) to simplify the expression of $P(\hat{e}_j; a, b, c, \lambda)$

Computing a simplified expression of $P(\hat{e}_j; a, b, c, \lambda)$

Plugging formula (4.20) into formula (4.19), the integrand in formula (4.19) is roughly

$$\begin{aligned}
&\propto \frac{1}{\sqrt{\sigma_j'^2}}\exp\left(-\frac{(e_j-\mu_j')^2}{2\sigma_j'^2}\right)\frac{1}{\sqrt{2\pi\sigma_j^2}}\exp\left(-\frac{(e_j-\mu_j)^2}{2\sigma_j^2}\right) \\
&= \frac{1}{\sqrt{\sigma_j^2+\sigma_j'^2}}\exp\left(-\frac{(\mu_j'-\mu_j)^2}{2(\sigma_j^2+\sigma_j'^2)}\right)\left[\frac{1}{\sqrt{2\pi\tilde{\sigma}_j^2}}\exp\left(-\frac{(e_j-\tilde{\mu}_j)^2}{2\tilde{\sigma}_j^2}\right)\right],
\end{aligned} \tag{4.21}$$

where

$$\tilde{\mu}_j = (\sigma_j^2\mu_j' + \sigma_j'^2\mu_j)/(\sigma_j^2 + \sigma_j'^2) \tag{4.22}$$

and

$$\tilde{\sigma}_j^2 = \sigma_j^2\sigma_j'^2/(\sigma_j^2 + \sigma_j'^2). \tag{4.23}$$

In formula (4.21), the part in the square brackets is the probability density function of a normal distribution with mean $\tilde{\mu}_j$ and variance $\tilde{\sigma}_j^2$. The part outside the square brackets has nothing to do with e_j. Let $\Phi(x)$ denote the cumulative distribution function of a standard normal distribution [134]. Plugging formula (4.21) into formula (4.19), we roughly have

$$\begin{aligned} & P\left(\hat{e}_j; a, b, c, \lambda\right) \\ \propto & \frac{1}{\sqrt{\sigma_j^2+\sigma_j'^2}} \exp\left(-\frac{\left(\mu_j'-\mu_j\right)^2}{2\left(\sigma_j^2+\sigma_j'^2\right)}\right) \int_0^1 \frac{1}{\sqrt{2\pi\tilde{\sigma}_j^2}} \exp\left(-\frac{(e_j-\tilde{\mu}_j)^2}{2\tilde{\sigma}_j^2}\right) de_j \\ = & \frac{1}{\sqrt{\sigma_j^2+\sigma_j'^2}} \exp\left(-\frac{\left(\mu_j'-\mu_j\right)^2}{2\left(\sigma_j^2+\sigma_j'^2\right)}\right) \left[\Phi\left(\frac{1-\tilde{\mu}_j}{\tilde{\sigma}_j}\right) - \Phi\left(\frac{-\tilde{\mu}_j}{\tilde{\sigma}_j}\right)\right]. \end{aligned} \tag{4.24}$$

Maximizing the likelihood function

According to formula (4.15), the log-likelihood function is

$$\sum_{j=n-w+1}^{n} \ln P(\hat{e}_j; a, b, c, \lambda). \tag{4.25}$$

Plugging formula (4.24) into formula (4.25) shows that to maximize the log-likelihood function, we only need to minimize

$$\sum_{j=n-w+1}^{n} \left[\ln\left(\sigma_j^2 + \sigma_j'^2\right) + \frac{\left(\mu_j'-\mu_j\right)^2}{\sigma_j^2+\sigma_j'^2} - 2\ln\left(\Phi\left(\frac{1-\tilde{\mu}_j}{\tilde{\sigma}_j}\right) - \Phi\left(\frac{-\tilde{\mu}_j}{\tilde{\sigma}_j}\right)\right)\right]. \tag{4.26}$$

Plugging formulas (4.13) and (4.14) into formulas (4.22), (4.23), and (4.26), we obtain the objective function to be minimized:

$$\sum_{j=n-w+1}^{n} \left[\ln\left(\lambda r_j^2 + \sigma_j'^2\right) + \frac{\left(\mu_j'-ax_j^{-b}-c\right)^2}{\lambda r_j^2+\sigma_j'^2} - 2\ln\left(\Phi\left(\frac{1-\tilde{\mu}_j}{\tilde{\sigma}_j}\right) - \Phi\left(\frac{-\tilde{\mu}_j}{\tilde{\sigma}_j}\right)\right)\right], \tag{4.27}$$

where

$$\tilde{\mu}_j = [\lambda r_j^2 \mu_j' + \sigma_j'^2(ax_j^{-b} + c)]/(\lambda r_j^2 + \sigma_j'^2) \tag{4.28}$$

and

$$\tilde{\sigma}_j^2 = \lambda r_j^2 \sigma_j'^2 / (\lambda r_j^2 + \sigma_j'^2).$$

This numerical optimization problem is subject to five constraints: $a > 0$, $b > 0$, $c > 0$, $\lambda > 0$, and

$$a x_{n-w+1}^{-b} + c \leq 1.$$

Recall that x_j denotes the normalized number of batches of model training finished before the j-th validation point. To derive the last constraint, recall that w denotes the number of validation points used to estimate a, b, c, and λ. n denotes the present validation point's sequence number. μ_j ($0 \leq \mu_j \leq 1$) is the mean of the model's generalization error at the j-th validation point. Formula (4.13) shows that

$$\mu_j = a x_j^{-b} + c.$$

As j increases, x_j strictly increases and hence μ_j strictly decreases. μ_j is always > 0. If

$$\mu_{n-w+1} = a x_{n-w+1}^{-b} + c$$

is ≤ 1, μ_j is in [0, 1] for each j between $n - w + 1$ and n.

In summary, we estimate a, b, c, and λ by minimizing the objective function given by formula (4.27) subject to five constraints: $a > 0$, $b > 0$, $c > 0$, $\lambda > 0$, and

$$a x_{n-w+1}^{-b} + c \leq 1.$$

The software package and its setting used to do numerical optimization

We use the interior-point algorithm [131, 137] implemented in the software package Artelys Knitro [138] to solve this constrained minimization problem. Typically, the estimated a, b, c, and λ are roughly on the order of magnitude of 0.1, 0.1 [127-129], 0.1, and 100, respectively. Accordingly,

when conducting numerical optimization, we initialize a, b, c, and λ to 0.1, 0.1, 0.1, and 100, respectively.

During the constrained numerical optimization process, one could allow the constraints to be violated [131]. However, if the constraint

$$ax_{n-w+1}^{-b} + c \leq 1$$

is violated, $\tilde{\mu}_j$ could be > 1 for one or more j between $n - w + 1$ and n (see formula (4.28)). If $\tilde{\mu}_j$ is $\gg 1$ and $\tilde{\sigma}_j$ is small, numerical underflow could occur in computing

$$\Phi\big((1 - \tilde{\mu}_j)/\tilde{\sigma}_j\big) - \Phi\big(-\tilde{\mu}_j/\tilde{\sigma}_j\big),$$

causing issues when we compute

$$\ln\left(\Phi\big((1 - \tilde{\mu}_j)/\tilde{\sigma}_j\big) - \Phi\big(-\tilde{\mu}_j/\tilde{\sigma}_j\big)\right)$$

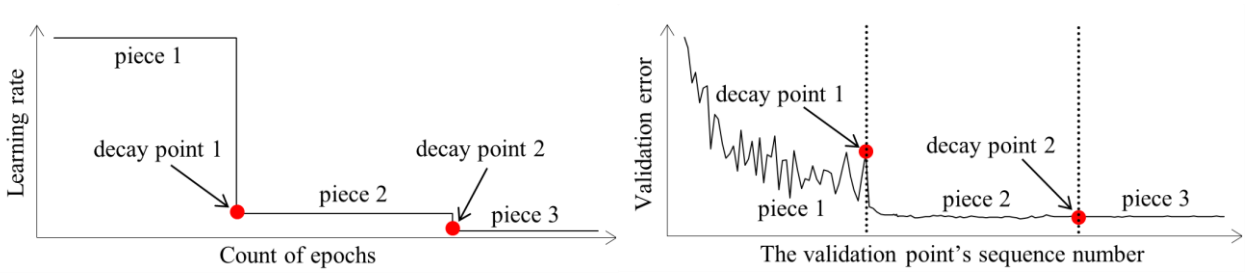
in formula (4.27). To avoid this issue, we set the bar_feasible parameter in Artelys Knitro to either 1 or 3 to ensure that the five constraints are always satisfied during the entire constrained numerical optimization process [131].

4.5.6.2 Estimating μ_j and σ_j^2 when a constant learning rate is Adopted

In this section, we describe how to estimate for each $j \geq 1$, the mean μ_j and the variance σ_j^2 of the model's generalization error at the j-th validation point when a constant learning rate is used. This case is a special case of applying an exponential decay method to the learning rate, when the constant ρ regulating the decay rate of the learning rate is 0. We employ the same approach in Section 4.5.6.1 to estimate μ_j and σ_j^2 for each $j \geq 1$.

4.5.6.3 Estimating μ_j and σ_j^2 when a step decay method is applied to the learning rate

This section describes how to estimate for each $j \geq 1$, the mean μ_j and the variance σ_j^2 of the model's generalization error at the j-th validation point when the learning rate changes over time based upon a step decay method.



(A) (B)

Figure 4.5. When the learning rate changes over time based upon a step decay method, the learning rate over epochs and an example validation curve.
(A) The learning rate over epochs. (B) An example validation curve.

As Figure 4.5(A) shows, in a step decay method, we cut the learning rate by a pre-chosen factor that is > 1 after a given number of epochs. This factor could change over epochs in a pre-determined fashion. Figure 4.5(B) presents a correspondent example validation curve. A decay point is defined as an original validation point at which the learning rate is cut. The decay points partition the validation curve into several pieces. For every $j \geq 1$, the first original validation point on the (j+1)-th piece is the j-th decay point. When model training begins, both the learning rate used on and the position of each piece are known.

As we move from one piece of the validation curve to the next, both the learning rate and the variance of the model's generalization error change. We consider this when estimating μ_j and σ_j^2 for each $j \geq 1$. As in Section 4.5.6.1, to estimate μ_j and σ_j^2, we need to estimate only the four parameters a, b, c, and λ used to model μ_j and σ_j^2. There are two possible cases: 1) the present validation point

resides on the first piece of the validation curve, and 2) the present validation point resides on the *k*-th ($k \geq 2$) piece of the validation curve. We discuss the two cases sequentially.

Case 1: The present validation point resides on the first piece of the validation curve

When the present validation point resides on the first piece of the validation curve, we adopt the method in Section 4.5.6.1 to estimate a, b, c, and λ.

Case 2: The present validation point resides on the k-th ($k \geq 2$) piece of the validation curve

Next, we discuss the case of the present validation point residing on the *k*-th ($k \geq 2$) piece of the validation curve. As shown in Figure 4.5(B), because of the decay of the learning rate at a decay point, the validation curve frequently drops abruptly at this point as well as at the next few validation points. As Figure 4.6 shows, when one arrives at a validation point that is not far after such a decay point, this drop could result in an inaccurately estimated trend curve if the estimation method in Section 4.5.6.1 were used.

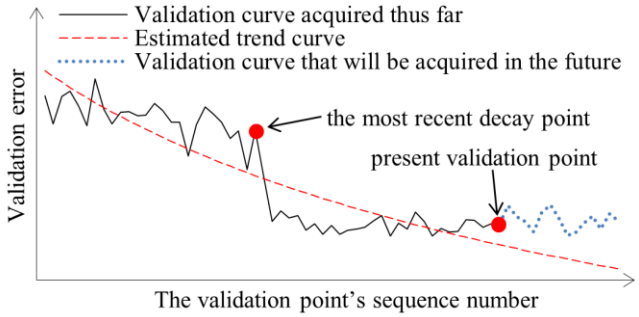


Figure 4.6. Employing the method in Section 4.5.6.1 to estimate the trend curve when one arrives at a validation point that is not far after the most recent decay point.

To deal with this issue, we revise the estimation method in Section 4.5.6.1. Let l_j ($j \geq 1$) denote the number of validation points that are on the *j*-th piece of the validation curve. Each l_j is known

beforehand. Recall that at least $\tau_v = 4$ data points are needed to estimate a, b, c, and λ. Usually, l_j is $\geq \tau_v$ for each $j \geq 1$.

$$s_{k-1} = \sum_{j=1}^{k-1} l_j$$

is the sequence number of the final validation point that is on the prior piece of the validation curve. Let $v_{k\text{-}1}$ denote the number of both original and added validation points needed for model training that is projected at the final validation point on the prior piece. If the $v_{k\text{-}1}$-th validation point resides on the present k-th piece, $v_{k\text{-}1} - s_{k\text{-}1}$ is this validation point's sequence number on the present k-th piece. Recall that n is the present validation point's sequence number. Let $h(n)$ denote the present validation point's sequence number on the present k-th piece. $h(n)$ is $\leq l_k$. There are two possible scenarios (see Figure 4.7).

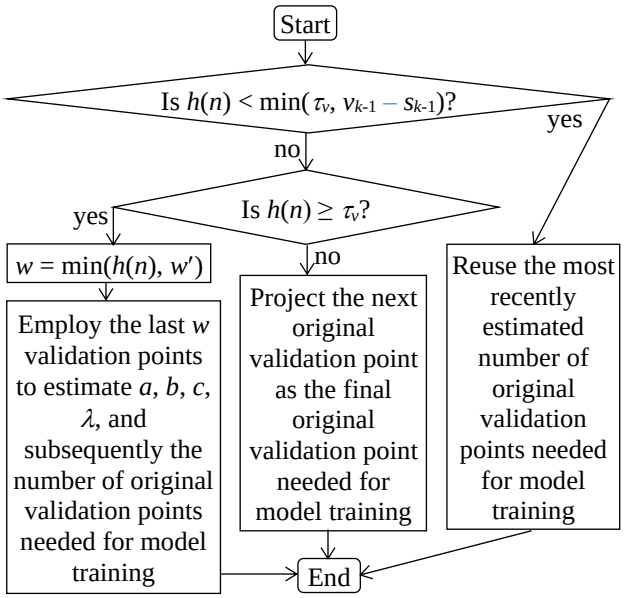


Figure 4.7. The flowchart of estimating the number of original validation points needed for model training when the present validation point resides on the k-th ($k \geq 2$) piece of the validation curve.

In the first scenario, $h(n)$ is $< \min(\tau_v, v_{k-1} - s_{k-1})$. In this case, we do not have enough validation points to estimate a, b, c, and λ. We reuse the most recently estimated number of original validation points needed for model training. Since τ_v is small, we often pass the phase of not updating the estimated number of original validation points needed for model training in a reasonably short period of time.

In the second scenario, $h(n)$ is $\geq \min(\tau_v, v_{k-1} - s_{k-1})$. If $v_{k-1} - s_{k-1} \leq h(n) < \tau_v$, we project the next original validation point as the final original validation point needed for model training. Otherwise, if $h(n)$ is $\geq \tau_v$, we revise the method in Section 4.5.6.1 in the following two ways to estimate a, b, c, and λ.

First, recall that x_j denotes the normalized number of batches of model training finished before the j-th validation point. The trend curve's value at the j-th validation point is $= \mu_j$. As shown in Figure 4.5(B), if moved to the left by $x_{s_{k-1}}$, the present piece of the trend curve has approximately the same form as an inverse power function. We adopt the same shifted inverse power function

$$\mu_j = a(x_j - x_{s_{k-1}})^{-b} + c$$

rather than formula (4.13) to model μ_j.

Second, recall that w' denotes the maximum number of validation points allowed to estimate a, b, c, and λ. n denotes the present validation point's sequence number. $h(n)$ denotes the present validation point's sequence number on the present piece of the validation curve. We employ the last

$$w = \min(h(n), w')$$

validation points on the present piece of the validation curve rather than the last $\min(n, w')$ validation points to estimate a, b, c, and λ.

4.5.7 *Determining V_{min}*

In this section, we show how to determine V_{min}, the minimum number of data instances needed in the randomly sampled subset of the full validation set used at an added validation point.

Recall that V_j ($j \geq 1$) is the number of data instances that are in the actual validation set used at the j-th validation point. $\hat{e}_j$ denotes the validation error of the model at the j-th validation point. At an added validation point, V_j is computed by formula (4.7) that involves V_{min}. In Section 4.5.6.1, we use a normal distribution to approximate a beta distribution with parameters

$$\alpha = V_j \hat{e}_j + 1$$

and

$$\beta = V_j(1 - \hat{e}_j) + 1.$$

This approximation is reasonably precise if α is ≥ 10 and β is ≥ 10 [136], which is equivalent to $V_j \geq 9/\hat{e}_j$ and $V_j \geq 9/(1 - \hat{e}_j)$. If we know $\hat{e}_j$'s lower bound $b_l > 0$ and upper bound $b_u < 1$, we can set V_{min} to

$$9 / \min(b_l, 1 - b_u)$$

to raise the chance of α being ≥ 10 and β being ≥ 10 for each $j \geq 1$. However, b_l and b_u are unknown beforehand. To address this issue, we start from an initial estimate $\hat{b}_l$ of b_l and an initial estimate $\hat{b}_u$ of b_u and set V_{min} to

$$9 / \min(\hat{b}_l, 1 - \hat{b}_u). \quad (4.29)$$

During model training, $\hat{e}_j$ could fall out of $[\hat{b}_l, \hat{b}_u]$ at some added validation point, making it possible to have $\alpha < 10$ or $\beta < 10$. At any added validation point, if $\hat{e}_j$ falls out of $[\hat{b}_l, \hat{b}_u]$, we lower $\hat{b}_l$ or

raise $\hat{b}_u$ to make $[\hat{b}_l, \hat{b}_u]$ include $\hat{e}_j$ and then re-compute V_{min} to make it larger. At any original validation point, if $\hat{e}_j$ falls out of $[\hat{b}_l, \hat{b}_u]$, we do not adjust $\hat{b}_l$ and $\hat{b}_u$ because the full validation set is used and there is no way to make V_j larger.

We have two considerations when setting the initial values of $\hat{b}_l$ and $\hat{b}_u$. First, the larger the $\hat{b}_l$ and the smaller the $\hat{b}_u$, the more likely $\hat{e}_j$ will fall out of $[\hat{b}_l, \hat{b}_u]$ at some added validation point during model training, which is undesirable. Second, if $\hat{b}_l$ is too small or $\hat{b}_u$ is too large, the V_{min} computed by formula (4.29) will be too large. Consequently, V_j could also be too large, undesirably increasing the progress indicator's run-time overhead. To strike a balance between these two considerations, we set the initial values of $\hat{b}_l$ and $\hat{b}_u$ to 0.02 and 0.98, respectively.

During model training, if the validation error $\hat{e}_j$ at an added validation point is outside of $[\hat{b}_l, \hat{b}_u]$, we proceed as follows:

1) **Step 1:** If $\hat{e}_j$ is > $\hat{b}_u$, we change $\hat{b}_u$ to $\hat{e}_j$. If $\hat{e}_j$ is < $\hat{b}_l$, we change $\hat{b}_l$ to $\hat{e}_j$.
2) **Step 2:** Use formula (4.29) to re-compute V_{min}. If $\hat{e}_j$ is = 0 or 1, which is unlikely to occur in practice, we set V_{min} to $+\infty$.
3) **Step 3:** Use formula (4.7) to re-compute V', the uniform number of data instances that are in the randomly sampled subset of the full validation set used at each added validation point.
4) **Step 4:** If the new V' differs from the old V', we re-sample the full validation set to obtain a new subset and re-compute $\hat{e}_j$, the validation error on the subset. The number of data instances that are in the subset is the new V', which will also be used at each added validation point after the present validation point.
5) **Step 5:** If $\hat{e}_j$ is re-computed in Step 4 and the new $\hat{e}_j$ is outside of $[\hat{b}_l, \hat{b}_u]$, we repeat Steps 1-4 until the new $\hat{e}_j$ is within $[\hat{b}_l, \hat{b}_u]$.

In practice, we rarely need to change V' from its initially computed value because 1) the initial [$\hat{b}_l$, $\hat{b}_u$] is wide and has a high likelihood to include $\hat{e}_j$, and 2) if the initially computed V' is > the V_{min} re-computed in Step 2, no value change will be made to V' in Step 3.

4.5.8 *Estimating the Model Training Cost Based upon the Projected Number of Original Validation Points Needed for Model Training*

After estimating the trend curve and the variance of the random noise, we can project the model training cost. The Monte Carlo simulation method in our basic progress indication method (Section 4.4.5) is used to estimate n_v, the number of original validation points needed for model training. Recall that V' is the uniform number of data instances that are in the randomly sampled subset of the full validation set used at each added validation point. n_j ($0 \leq j \leq v_{max} - 1$) is the number of validation points to be added between the j-th and the (j+1)-th original validation points. q is the constant regulating the decay rate of n_j ($0 \leq j \leq v_{max} - 1$) in the exponential decay schema. In Section 4.4.2, we show that the mean amount of work taken to process a validation instance one time to calculate the validation error is 1/3 unit of work. The model training cost is the sum of three components:

1) The cost to process the training instances (see Section 4.4.2).
2) The cost to calculate the validation errors at the original validation points (see Section 4.4.2).
3) The cost to calculate the validation errors at the added validation points

   = the total number of validation points added before the n_v-th original validation point × the uniform number of data instances that are in the randomly sampled subset of the full validation set used at each added validation point × the mean amount of work taken to process a validation instance one time to calculate the validation error

   = $\sum_{j=0}^{n_v} n_j \times V'/3$

$= \sum_{j=0}^{n_v} \lfloor n_0 q^j \rceil \times V'/3.$

4.6 PERFORMANCE

This section presents the performance test results of our progress indication methods for deep learning model training. We mainly show the performance test results of our improved progress indication method, as well as compare them with the performance test results of our basic progress indication method. TensorFlow is a commonly used open-source software package for deep learning created by Google [66]. We implemented our methods in TensorFlow Version 1.13.1. In each test, our progress indicators gave informative estimates and revised them every 10 seconds with minute overhead, fulfilling the progress indication goals of low overhead, continuously revised updates, and reasonable pacing listed in our prior paper [97].

4.6.1 *Description of the Experiments*

The experiments were performed by running TensorFlow on a Digital Storm workstation. The workstation runs the Ubuntu 18.04.02 operating system and has 64GB memory, one eight-core Intel Core i7-9800X 3.8GHz CPU, one GeForce RTX 2080 Ti GPU, one 3TB SATA disk, and one 500GB solid-state drive. Every deep learning model was trained on an unloaded system and using the GPU.

We tested two standard deep learning models: the Gated Recurrent Unit (GRU) model, a recurrent neural network, used in Purushotham *et al.* [139] and the convolutional neural network GoogLeNet [82]. For every model, we tested four standard optimization algorithms for deep learning model training: root mean square propagation (RMSprop) [140], classical SGD [141], adaptive gradient (AdaGrad) [142], and Adam [90]. For each (deep learning model, optimization algorithm) pair, three learning rate decay methods were tested: using an exponential decay method, a step decay method, and a constant learning rate. We present the test results for GoogLeNet using Adam and the

GRU model using RMSprop. The test results for the other (deep learning model, optimization algorithm) pairs are similar and shown in the Appendix in the full version of another paper [65]. There is one exception. For the step decay method, we present the test results for GoogLeNet using Adam. The test results for using RMSprop and the step decay method to train the GRU model are similar and shown in the Appendix in the full version of another paper [65].

We employed two popular benchmark datasets shown in Table 4.12: CIFAR-10 [143] and MIMIC-III [144]. GoogLeNet was trained on CIFAR-10. In CIFAR-10, every data instance is an image of size 32×32. CIFAR-10 was split into a validation set and a training set as described in Krizhevsky [143]. The GRU model was trained on a subset of the MIMIC-III dataset called "Feature Set C, 48-h data" to perform the "ICD-9 code group prediction" task in Purushotham *et al*. [139]. In the subset, every data instance is a sequence of length 48. The subset was partitioned into a validation set and a training set as described in Purushotham *et al*. [139].

Table 4.12. The datasets that we used to test our progress indication method.

Name	**Number of data instances that are in the validation set**	**Number of data instances that are in the training set**	**Number of classes**	**Data instance size**
CIFAR-10	10,000	50,000	10	image size: 32×32
Feature Set C, 48-h data	6,845	20,532	20	sequence length: 48

Except for the maximum number of epochs allowed for model training and the learning rate decay method, all the hyper-parameters were given their default values that appeared in the open source code of GoogLeNet and the GRU model [145, 146]. In particular, the number of training instances in every batch was = 100 and 128 for the GRU model and GoogLeNet, respectively. In each test, the beginning learning rate was = 0.001. The patience p was = 11, an integer randomly selected from [3, 25]. The min_delta δ was = 0.00131, a number randomly selected from [0, 0.01].

The maximum number of epochs allowed for model training was = 150. An original validation point was put at or near the end of each epoch of model training. Accordingly, the number of batches of model training between two consecutive original validation points was 390 and 205 for GoogLeNet and the GRU model, respectively.

Recall that v_{max} denotes the maximum number of original validation points allowed for model training. n_j ($0 \leq j \leq v_{max} - 1$) is the number of validation points added between the j-th and the (j+1)-th original validation points. n_0 is the number of validation points added before the first original validation point. q is the constant regulating the decay rate of n_j ($0 \leq j \leq v_{max} - 1$) in the exponential decay schema. V' is the uniform number of data instances that are in the randomly sampled subset of the full validation set used at each added validation point. For each of GoogLeNet and the GRU model, Table 4.13 shows the n_0, q, and V' set by the approach given in Section 4.5.3. In our experiments, V' never changed during model training.

Table 4.13. For each of GoogLeNet and the GRU model, the n_0, q, and V' set by the approach given in Section 4.5.3.

Model	n_0	q	V'
GoogLeNet	11	0.93	726
GRU	5	0.93	683

4.6.2 *Accuracy Measure*

We used the average prediction error adopted in Chaudhuri *et al*. [105] to gauge the progress indicator's estimation accuracy. The average prediction error is the ratio of a numerator to a denominator (see Figure 4.8). The area of the region between a straight diagonal line and a curve is the numerator. The straight line shows the real remaining model training time. The curve shows the progress indicator's estimate of the remaining model training time over time. The area of the triangle

created by the straight diagonal line, the y-axis, and the x-axis is the denominator. The larger the average prediction error, the less accurate the estimates given by the progress indicator.

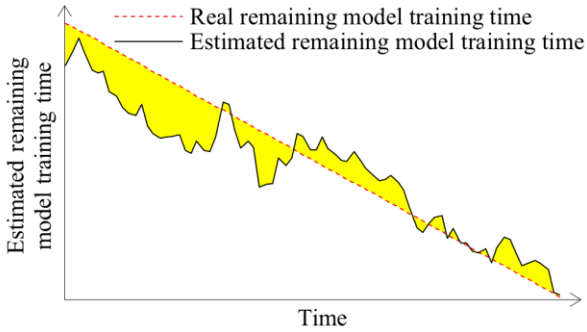


Figure 4.8. The areas of the regions employed to calculate the average prediction error.

4.6.3 *Comparison of Three Progress Indication Methods for Deep Learning Model Training*

We compared the accuracy of the progress estimates provided by three progress indication methods for deep learning model training:

1) **Method 1**: This is our basic method (see Section 4.4).
2) **Method 2**: This is a hybrid of our basic and improved methods. We use the approach in Section 4.5.3 to insert extra validation points between the original validation points, the approach in Section 4.5.4 to set the uniform number of data instances that are in the randomly sampled subset of the full validation set used at each added validation point, the approach in Sections 4.4.3 to 4.4.5 to predict the number of original validation points needed for model training, and the approach in Section 4.5.8 to estimate the model training cost based upon the projected number. We disregard the dependency of the random noise's variance on the size of the actual validation set used at the validation point. Instead, we deem the random noise's variance to be approximately $\propto$ the square of the learning rate with no reliance on the size of the actual validation set used at the validation point [64].
3) **Method 3**: This is our improved method shown in Section 4.5.

We conducted 24 tests, one for every combination of a deep learning model, an optimization algorithm, and a learning rate decay method. In each test, we trained the deep learning model five times, each in a distinct run. In each run, we used each of the three progress indication methods to provide progress estimates. For each test, Table 4.14 shows the standard deviation and the mean of the average prediction error over the five runs for each of the three methods. For each test, the smallest mean of the average prediction error over the five runs across the three methods is marked in bold in Table 4.14.

Table 4.14. For each of the 24 tests, the mean as well as the standard deviation of the average prediction error over the five runs for each of the three progress indication methods.

Deep learning model	Learning rate decay method	Optimization algorithm	Average prediction error		
			Progress indication method 1	Progress indication method 2	Progress indication method 3
GoogLeNet	Using a constant learning rate	Adam	0.50±0.10	**0.45**±0.12	0.51±0.14
		RMSprop	0.53±0.25	**0.42**±0.11	**0.42**±0.13
		SGD	0.18±0.03	0.30±0.01	**0.11**±0.01
		AdaGrad	0.17±0.07	0.41±0.02	**0.15**±0.02
	Exponential decay method	Adam	2.46±1.20	1.46±0.66	**0.89**±0.26
		RMSprop	1.20±0.51	0.79±0.19	**0.66**±0.05
		SGD	1.32±0.53	0.97±0.31	**0.70**±0.20
		AdaGrad	1.22±0.29	0.80±0.16	**0.58**±0.09
	Step decay method	Adam	0.45±0.06	0.45±0.07	**0.44**±0.11
		RMSprop	0.73±0.50	**0.54**±0.14	0.57±0.14
		SGD	0.40±0.04	0.49±0.05	**0.34**±0.09
		AdaGrad	**0.35**±0.04	0.44±0.05	0.52±0.09
GRU	Using a constant learning rate	Adam	1.94±0.67	0.54±0.08	**0.48**±0.05
		RMSprop	1.55±0.53	0.60±0.17	**0.52**±0.19
		SGD	0.65±0.08	**0.43**±0.08	0.58±0.12
		AdaGrad	0.93±0.60	0.52±0.03	**0.48**±0.08
	Exponential decay method	Adam	2.40±1.17	0.60±0.18	**0.44**±0.13
		RMSprop	1.27±0.22	0.44±0.13	**0.25**±0.09
		SGD	1.39±0.25	0.93±0.15	**0.51**±0.07
		AdaGrad	1.45±0.62	0.66±0.58	**0.42**±0.26
	Step decay method	Adam	1.94±0.60	0.55±0.18	**0.46**±0.17
		RMSprop	1.59±0.17	0.51±0.07	**0.47**±0.13
		SGD	0.57±0.10	**0.41**±0.08	0.55±0.12
		AdaGrad	1.99±0.50	0.63±0.21	**0.45**±0.16
Over all runs in all tests			1.13±0.84	0.60±0.33	**0.48**±0.21

Comparison of methods 1 and 3

In 20 of the 24 tests, method 3 beat method 1 and had a smaller mean of the average prediction error over the five runs. Method 1 outperformed method 3 in the other two tests: 1) using Adam and a constant learning rate to train GoogLeNet, and 2) using AdaGrad and applying a step decay method to the learning rate to train GoogLeNet. The mean of the average prediction error over all runs in all tests for method 3 is 0.48, which is 57.5% lower than the corresponding mean of 1.13 for method 1. Thus, compared with using our basic method, using our improved method reduces the progress indicator's prediction error of the remaining model training time. Moreover, our improved method gave decently accurate estimates of the remaining model training time.

Comparison of methods 2 and 3

In 18 of the 24 tests, method 3 beat method 2 and had a smaller mean of the average prediction error over the five runs. Method 2 outperformed method 3 in the other five tests: 1) using Adam and a constant learning rate to train GoogLeNet, 2) using RMSprop and applying a step decay method to the learning rate to train GoogLeNet, 3) using AdaGrad and applying a step decay method to the learning rate to train GoogLeNet, 4) using SGD and a constant learning rate to train the GRU model, and 5) using SGD and applying a step decay method to the learning rate to train the GRU model. The mean of the average prediction error over all runs in all tests for method 3 is 0.48, which is 20.0% lower than the corresponding mean of 0.60 for method 2. Thus, considering the dependency of the random noise's variance on the size of the actual validation set used at the validation point raises the progress indicator's prediction accuracy.

In Sections 4.6.4 to 4.6.6, we focus on the improved progress indication method described in Section 4.5. Yet, for the remaining model training time, we show the estimates provided by both the

basic and the improved progress indication methods. Recall that in each of the 24 tests, we trained the deep learning model five times, each in a distinct run. We randomly selected one of the five runs and present the outputs of the progress indicator over time for that run.

4.6.4 *Test Results for Adopting a Constant Learning Rate*

This section presents the test results for adopting a constant learning rate.

4.6.4.1 Test results for training GoogLeNet

In the test, we used the Adam optimization algorithm and a constant learning rate to training GoogLeNet. Figure 4.9 depicts the progress indicator's estimated model training cost over time, with the dotted horizontal line showing the real model training cost. Before reaching τ_v = 4 validation points within 39 seconds, the progress indicator estimated the model training cost based upon the maximum number of original validation points allowed for model training, which diverged notably from the real number of original validation points needed for model training. As a result, the estimated model training cost greatly differed from the real model training cost. After reaching four or more validation points, the progress indicator refined the estimated model training cost for it to become more accurate over time.

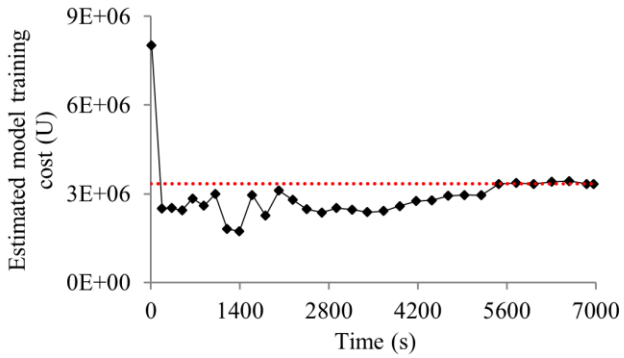


Figure 4.9. Model training cost estimated over time (using Adam and a constant learning rate to train GoogLeNet).

Figure 4.10 depicts the model training speed that the progress indicator observed over time. This speed was relatively stable during the whole model training process.

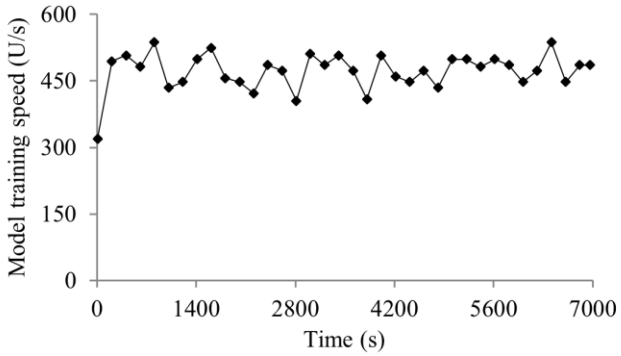


Figure 4.10. Model training speed over time (using Adam and a constant learning rate to train GoogLeNet).

Figure 4.11 and Figure 4.12 depict the remaining model training time estimated by the basic and the improved progress indication methods over time, with the dashed line showing the real remaining model training time. Before 691 seconds, the basic method's estimate of the remaining model training time differed notably from the real remaining model training time. The improved method reached the stage of giving relatively accurate estimates of the remaining model training time much faster than the basic method.

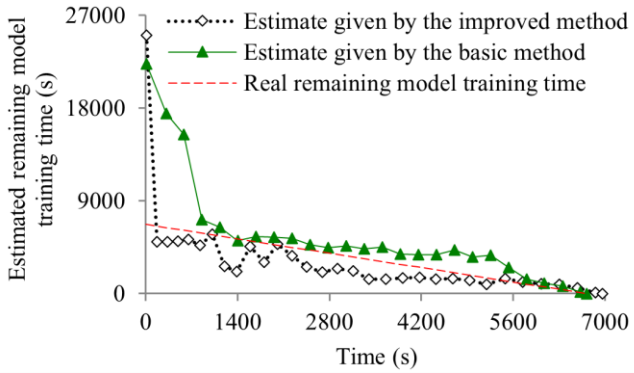


Figure 4.11. Estimated remaining model training time (using Adam and a constant learning rate to train GoogLeNet).

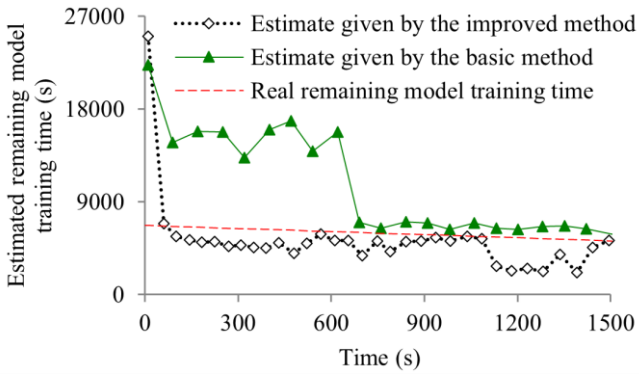


Figure 4.12. Estimate of the remaining model training time at the early stage of model training (using Adam and a constant learning rate to train GoogLeNet).

Figure 4.13 depicts the progress indicator's estimate over time of the finished percentage of model training work. The curve showing the estimated finished percentage is reasonably close to the diagonal dotted line linking the upper right and the lower left corners.

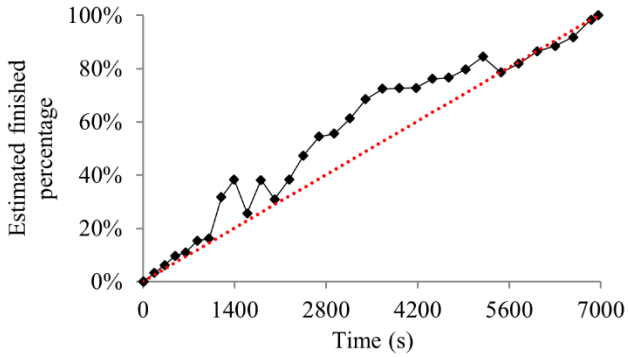


Figure 4.13. Finished percentage estimated over time (using Adam and a constant learning rate to train GoogLeNet).

4.6.4.2 Test results for training the GRU model

In the test, we used the RMSprop optimization algorithm and a constant learning rate to train the GRU model. We wanted to show that the estimates given by the progress indicator can be decently accurate for distinct kinds of neural networks.

Figure 4.14 depicts the progress indicator's estimated model training cost over time, with the dotted horizontal line showing the real model training cost. After we reached $\tau_v = 4$ validation points

within 7 seconds, the estimated model training cost became decently accurate for the rest of the model training process.

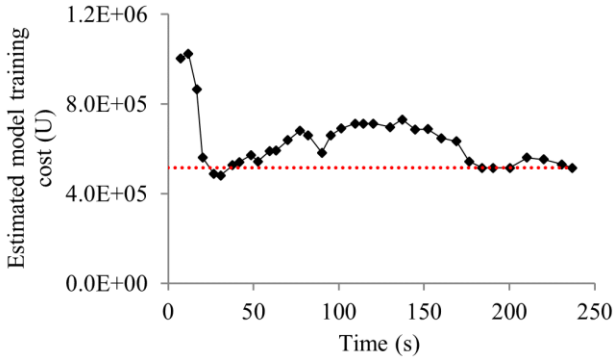


Figure 4.14. Model training cost estimated over time (using RMSprop and a constant learning rate to train the GRU model).

Figure 4.15 depicts the model training speed that the progress indicator observed over time. This speed was relatively stable during the whole model training process.

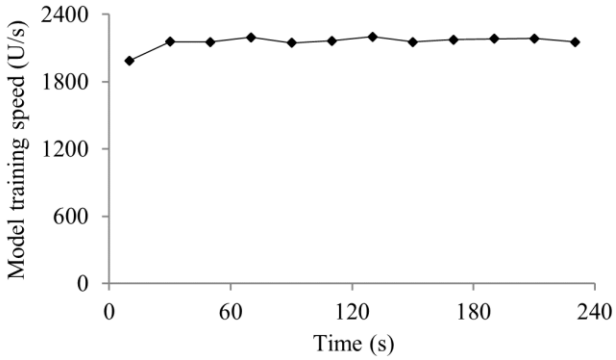


Figure 4.15. Model training speed over time (using RMSprop and a constant learning rate to train the GRU model).

Figure 4.16 depicts the remaining model training time estimated by the basic and the improved progress indication methods over time, with the dashed line showing the real remaining model training time. The improved method reached the stage of giving relatively accurate estimates of the remaining model training time much faster than the basic method. In fact, the improved method's

estimate of the remaining model training time was decently accurate during the whole model training process.

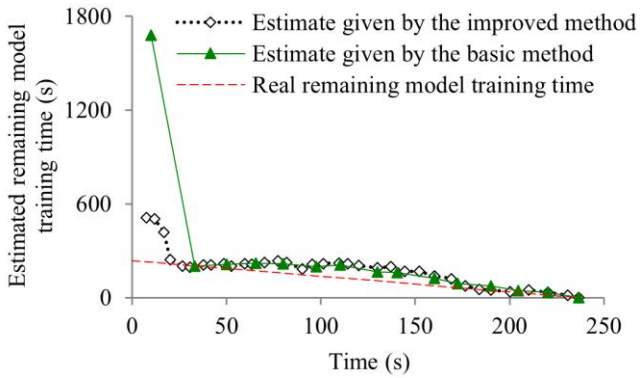


Figure 4.16. Estimated remaining model training time (using RMSprop and a constant learning rate to train the GRU model).

Figure 4.17 depicts the progress indicator's estimate over time of the finished percentage of model training work. The curve showing the estimated finished percentage is reasonably close to the diagonal dotted line linking the upper right and the lower left corners.

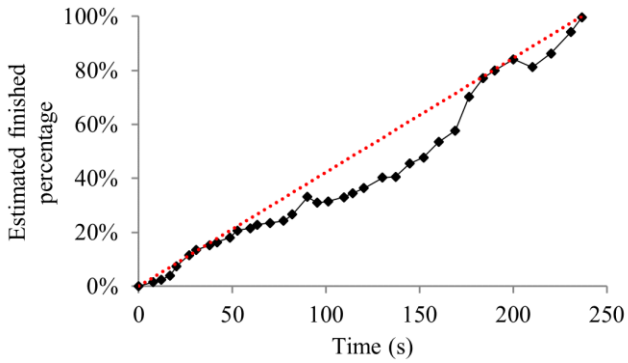


Figure 4.17. Finished percentage estimated over time (using RMSprop and a constant learning rate to train the GRU model).

4.6.5 *Test Results for Applying an Exponential Decay Method to the Learning Rate*

This section presents the test results for applying an exponential decay method to the learning rate. We set the constant ρ regulating the decay rate of the learning rate to 0.05.

4.6.5.1 Test results for training GoogLeNet

In the test, we used the Adam optimization algorithm and applied an exponential decay method to the learning rate to train GoogLeNet. Figure 4.18-Figure 4.21 depict the results for this test. From 0 to 2,002 seconds, the model training cost estimated by the new progress indication method oscillated and differed notably from the real model training cost most of the time. This difference led to inaccurate estimates of the remaining model training time and the percentage of model training work finished. After 2,002 seconds, the improved progress indication method gave more accurate progress estimates. The improved method reached the stage of giving relatively accurate estimates of the remaining model training time much faster than the old method.

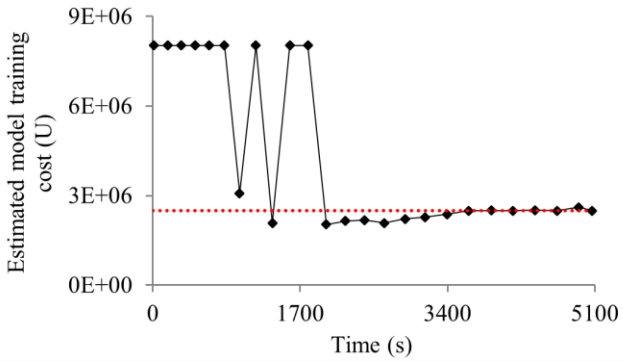


Figure 4.18. Model training cost estimated over time (using Adam and applying an exponential decay method to the learning rate to train GoogLeNet).

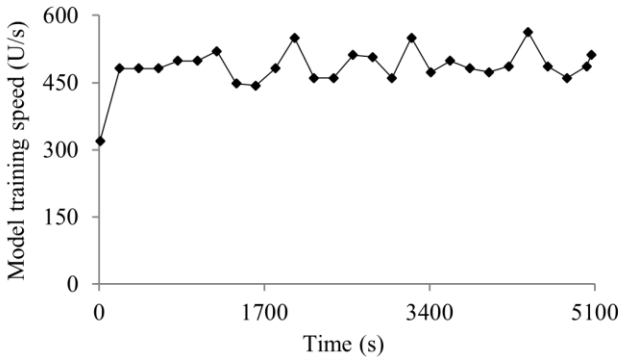


Figure 4.19. Model training speed over time (using Adam and applying an exponential decay method to the learning rate to train GoogLeNet).

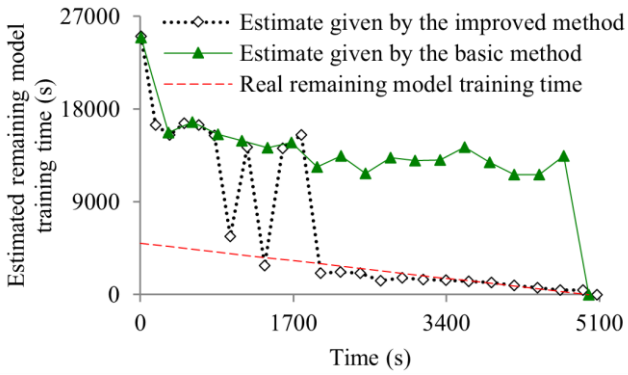


Figure 4.20. Estimated remaining model training time (using Adam and applying an exponential decay method to the learning rate to train GoogLeNet).

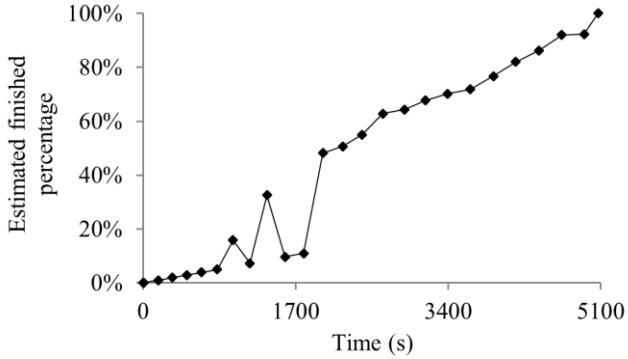


Figure 4.21. Finished percentage estimated over time (using Adam and applying an exponential decay method to the learning rate to train GoogLeNet).

4.6.5.2 Test results for training the GRU model

In the test, we used the RMSprop optimization algorithm and applied an exponential decay method to the learning rate to train the GRU model. Figure 4.22-Figure 4.25 depict the results for this test, showing that our improved progress indication method gave decently accurate estimates during most of the model training process. The improved method reached the stage of giving relatively accurate estimates of the remaining model training time much faster than the basic method.

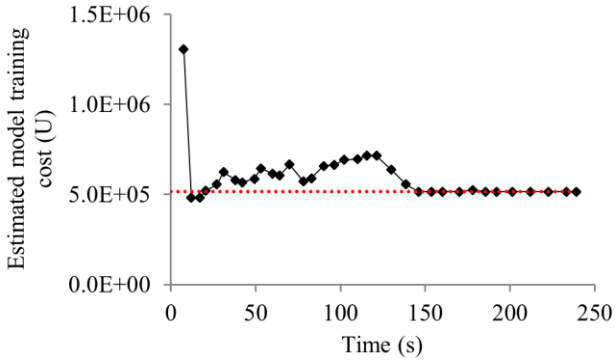


Figure 4.22. Model training cost estimated over time (using RMSprop and applying an exponential decay method to the learning rate to train the GRU model).

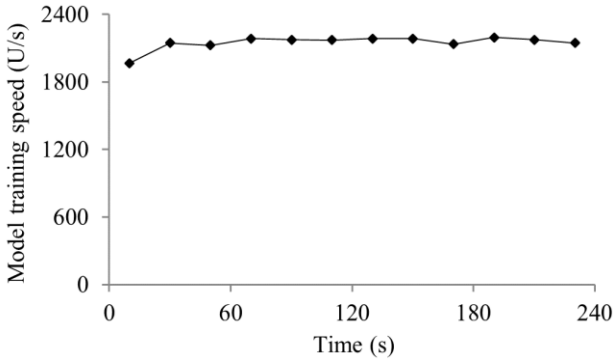


Figure 4.23. Model training speed over time (using RMSprop and applying an exponential decay method to the learning rate to train the GRU model).

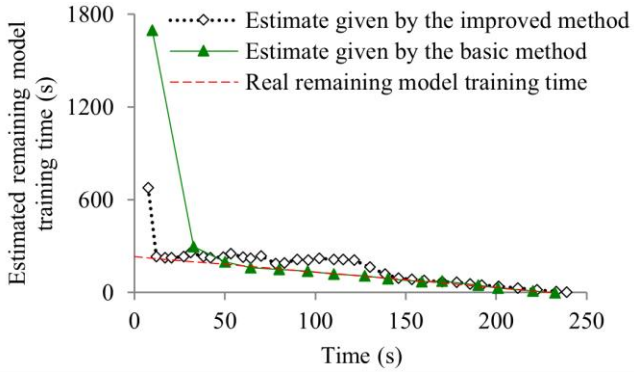


Figure 4.24. Estimated remaining model training time (using RMSprop and applying an exponential decay method to the learning rate to train the GRU model).

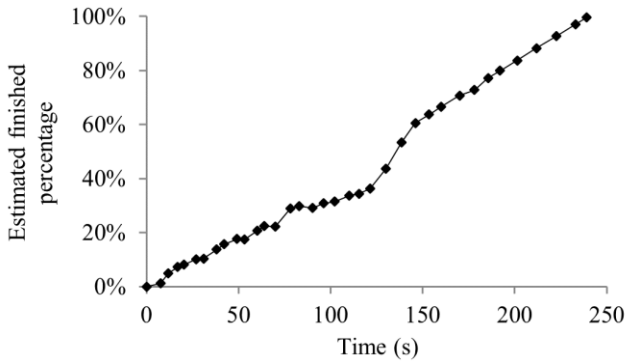


Figure 4.25. Finished percentage estimated over time (using RMSprop and applying an exponential decay method to the learning rate to train the GRU model).

4.6.6 *Test Results for Applying a Step Decay Method to the Learning Rate to Train GoogLeNet*

This section presents the test results for adopting the Adam optimization algorithm and applying a step decay method to the learning rate to train GoogLeNet. We cut the learning rate from 10^{-3} to 10^{-4} at the start of the 64-th epoch, and subsequently to 10^{-5} at the start of the 115-th epoch. In the test, early stopping happened on the first piece of the validation curve. Figure 4.26-Figure 4.30 present the test results, which are akin to those presented in Figure 4.9-Figure 4.13.

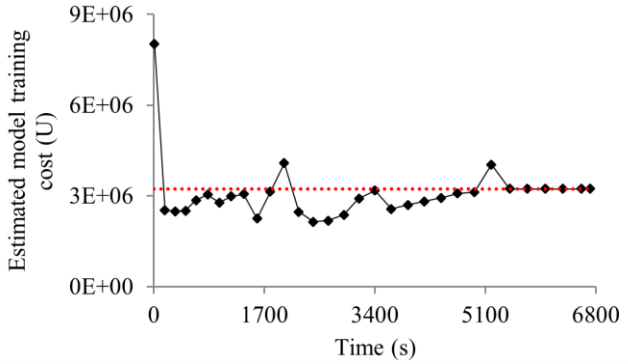


Figure 4.26. Model training cost estimated over time (using Adam and applying a step decay method to the learning rate to train GoogLeNet).

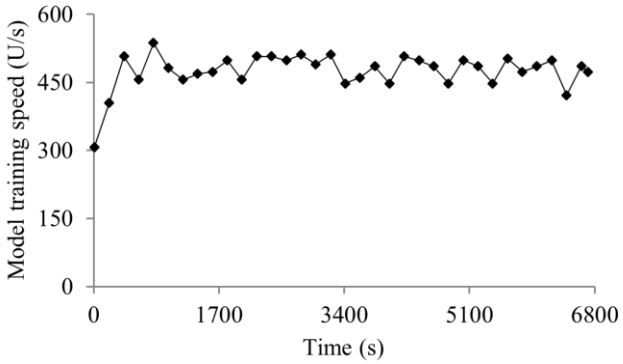


Figure 4.27. Model training speed over time (using Adam and applying a step decay method to the learning rate to train GoogLeNet).

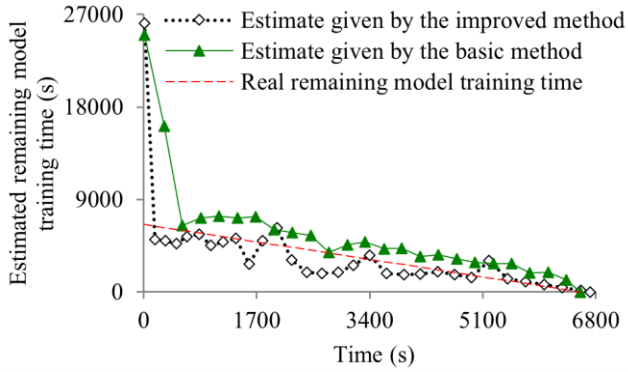


Figure 4.28. Estimated remaining model training time (using Adam and applying a step decay method to the learning rate to train GoogLeNet).

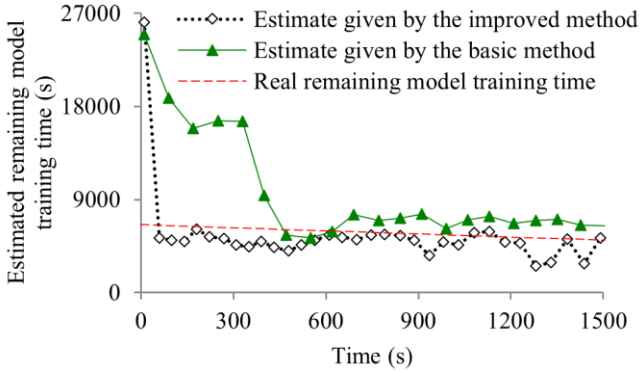


Figure 4.29. Estimate of the remaining model training time at the early stage of model training (using Adam and applying a step decay method to the learning rate to train GoogLeNet).

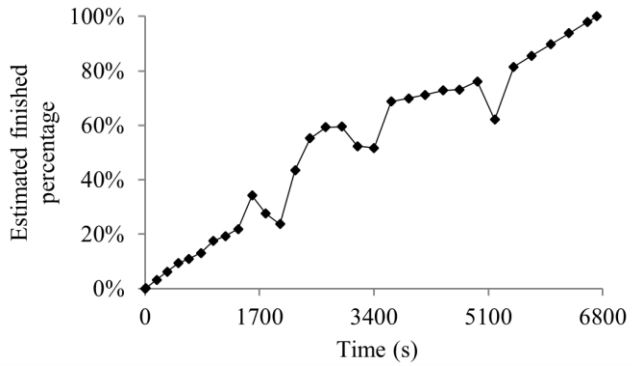


Figure 4.30. Finished percentage estimated over time (using Adam and applying a step decay method to the learning rate to train GoogLeNet).

4.6.7 *Summary of the Performance Test Results*

In summary, our experiments show that our progress indicators can offer useful information even if the run-time system load varies over time [62]. In addition, the progress indicator can self-correct its initial estimation errors, if any, over time. Compared with using our basic progress indication method, using the improved method reduces the progress indicator's prediction error. Moreover, the improved method enables us to obtain relatively accurate progress estimates faster with a low overhead.

4.7 DISCUSSION

In this section, we outline some directions for future work.

This work does not give any upper bound for the progress indicator's projection errors of the model training cost. To derive such upper bounds in the future, we could employ an approach that is akin to the approach used by Chaudhuri *et al*. [147] for progress indication for executing database queries.

We use the same single early stopping condition to do a case study to demonstrate that it is feasible to build non-trivial progress indicators for deep learning model training. Besides this early

stopping condition, many other early stopping conditions exist [62, 123-125]. In the future, we plan to investigate how our present progress indication techniques work for some other popular early stopping conditions and whether our present techniques require any changes to work well for those conditions.

This work focuses on using deep learning for classification. Deep learning can also be used for regression. In the future, we plan to investigate how to revise our improved progress indication method to handle deep learning regression models. When training a deep learning classification model, the validation error given the model's generalization error follows a discrete distribution linked to a binomial distribution. This is used in Section 4.5.5 to derive the relationship between the random noise's variance and the size of the actual validation set used at the validation point. In comparison, when training a deep learning regression model, the validation error given the model's generalization error follows a continuous distribution. Accordingly, to enable the new progress indication method to handle regression models, we need to derive a different relationship between the random noise's variance and the size of the actual validation set used at the validation point.

4.8 Conclusion

In this chapter, we propose two progress indication methods for deep learning model training that allows early stopping. Our main idea is to use the validation curve to project the number of validation points (or number of batches) needed for model training. During model training, we keep refining the projected model training cost and checking the current model training speed. Periodically, we revise the projected fraction of model training work completed and the projected remaining model training time displayed to the user. Our experiments show that the resulting progress indicator can offer useful information even if the run-time system load varies over time [62]. In addition, the

progress indicator can self-correct its initial estimation errors, if any, over time. By judiciously inserting extra validation points between the original validation points and revising the predicted model training cost at both the original and the added validation points, our improved method could address our basic method's shortcoming of having a long delay in obtaining relatively accurate progress estimates for the model training process. Our experimental results show that compared with using our basic method, using the improved method not only greatly reduces the progress indicator's prediction error of the remaining model training time, but also enables us to obtain relatively accurate progress estimates faster.

Chapter 5. SUMMARY

Although osteoporosis is a debilitating disease, osteoporosis screening is underutilized. One complementary approach to osteoporosis screening is opportunistic screening using pre-existing images to detect spinal OCFs. However, OCFs are often incidental findings and under-reported. An accurate automated opportunistic screening tool could improve diagnosis and enable more and earlier treatment of osteoporosis. To build this OCF classifier, we obtained two spine radiograph datasets: the UW dataset and the MrOS dataset. To annotate the data, we designed DicomAnnotator, the configurable open-source software program for efficient DICOM image annotation. To build the OCF classifier, we used five different deep learning algorithms and the above two datasets. Training a deep learning model on a large dataset is often time-consuming. During deep learning model training, it is desirable to offer a non-trivial progress indicator to make the deep learning model training process more user-friendly. We designed the basic progress indication method for deep learning model training that allows early stopping, as well as the improved method to increase the accuracy of the progress indication.

Recall that we had the following three aims:

4) **Aim 1:** Develop DicomAnnotator and use it to annotate the datasets.
5) **Aim 2:** Build the OCF classifier using deep learning and the annotated datasets.
6) **Aim 3:** Design progress indication methods that can support non-trivial progress indicators for deep learning model training in the presence of early stopping.

We realized all of these three aims.

For Aim 1, our designed DicomAnnotator integrates multiple functional modules to meet several annotation requirements and provides four ancillary user-friendly features. The program is

easy to learn, is efficient to use, and allows annotators to quickly make several types of annotations on a large set of DICOM images.

For Aim 2, we used five deep learning algorithms to train the OCF classifier that can detect OCFs on vertebral patches. This OCF classifier has good performance and some generalizability and can serve as a critical component of our future automated opportunistic screening tool.

For Aim 3, we proposed two progress indication methods for deep learning model training that allows early stopping. During model training, we keep refining the projected model training cost and checking the current model training speed. Periodically, we revise the projected fraction of model training work completed and the projected remaining model training time displayed to the user. The resulting progress indicator can offer useful information even if the run-time system load varies over time [62]. In addition, the progress indicator can self-correct its initial estimation errors, if any, over time.

BIBLIOGRAPHY

1. Looker, A.C., et al., *Osteoporosis or low bone mass at the femur neck or lumbar spine in older adults: United States, 2005-2008.* NCHS Data Brief, 2012(93): p. 1-8.
2. Kanis, J.A., *Assessment of Osteoporosis at the Primary Health Care Level*. 2007, World Health Organization Collaborating Centre for Metabolic Bone Diseases, University of Sheffield, UK.
3. Hodsman, A.B., et al., *10-year probability of recurrent fractures following wrist and other osteoporotic fractures in a large clinical cohort: an analysis from the Manitoba Bone Density Program.* Arch Intern Med, 2008. **168**(20): p. 2261-2267.
4. Roux, S., et al., *The World Health Organization Fracture Risk Assessment Tool (FRAX) underestimates incident and recurrent fractures in consecutive patients with fragility fractures.* J Clin Endocrinol Metab, 2014. **99**(7): p. 2400-2408.
5. Robinson, C.M., et al., *Refractures in patients at least forty-five years old: a prospective analysis of twenty-two thousand and sixty patients.* J Bone Joint Surg Am, 2002. **84**(9): p. 1528-1533.
6. Center, J.R., et al., *Mortality after all major types of osteoporotic fracture in men and women: an observational study.* Lancet, 1999. **353**(9156): p. 878-882.
7. Meadows, E.S., et al., *Compliance with mammography and bone mineral density screening in women at least 50 years old.* Menopause, 2011. **18**(7): p. 794-801.
8. King, A.B. and D.M. Fiorentino, *Medicare payment cuts for osteoporosis testing reduced use despite tests' benefit in reducing fractures.* Health Aff (Millwood), 2011. **30**(12): p. 2362-2370.
9. Jain, S., et al., *Are men at high risk for osteoporosis underscreened? A quality improvement project.* Perm J, 2016. **20**(1): p. 60-64.
10. Pickhardt, P.J., et al., *Opportunistic screening for osteoporosis using abdominal computed tomography scans obtained for other indications.* Ann Intern Med, 2013. **158**(8): p. 588-595.
11. Anderson, P.A., et al., *Clinical use of opportunistic computed tomography screening for osteoporosis.* J Bone Joint Surg Am, 2018. **100**(23): p. 2073-2081.
12. Alacreu, E., D. Moratal, and E. Arana, *Opportunistic screening for osteoporosis by routine CT in Southern Europe.* Osteoporos Int, 2017. **28**(3): p. 983-990.
13. Li, Y.L., et al., *Opportunistic screening for osteoporosis in abdominal computed tomography for Chinese population.* Arch Osteoporos, 2018. **13**(1): p. 76.
14. Cheng, X., et al., *Opportunistic screening using low-dose CT and the prevalence of osteoporosis in China: a nationwide, multicenter study.* J Bone Miner Res, 2021. **36**(3): p. 427-435.
15. Fang, Y., et al., *Opportunistic osteoporosis screening in multi-detector CT images using deep convolutional neural networks.* Eur Radiol, 2021. **31**(4): p. 1831-1842.
16. Nam, K.H., et al., *Machine learning model to predict osteoporotic spine with Hounsfield units on lumbar computed tomography.* J Korean Neurosurg Soc, 2019. **62**(4): p. 442-449.
17. Loffler, M.T., et al., *Automatic opportunistic osteoporosis screening in routine CT: improved prediction of patients with prevalent vertebral fractures compared to DXA.* Eur Radiol, 2021. **31**(8): p. 6069-6077.
18. Yasaka, K., et al., *Prediction of bone mineral density from computed tomography: application of deep learning with a convolutional neural network.* Eur Radiol, 2020. **30**(6): p. 3549-3557.
19. Bar, A., et al., *Compression fractures detection on CT*, in *Proceedings of SPIE*. 2017. p. 1013440.
20. Yilmaz, E.B., et al., *Automated deep learning-based detection of osteoporotic fractures in CT images*, in *Proceedings of MLMI@MICCAI*. 2021. p. 376-385.
21. Husseini, M.E., et al., *Conditioned variational auto-encoder for detecting osteoporotic vertebral fractures*, in *Proceedings of CSI@MICCAI*. 2019. p. 29-38.
22. Tomita, N., Y.Y. Cheung, and S. Hassanpour, *Deep neural networks for automatic detection of osteoporotic vertebral fractures on CT scans.* Comput Biol Med, 2018. **98**: p. 8-15.
23. Lee, S., et al., *The exploration of feature extraction and machine learning for predicting bone density from simple spine X-ray images in a Korean population.* Skeletal Radiol, 2020. **49**(4): p. 613-618.
24. Zhang, B., et al., *Deep learning of lumbar spine X-ray for osteopenia and osteoporosis screening: a multicenter retrospective cohort study.* Bone, 2020. **140**: p. 115561.
25. Chou, P.H., et al., *Ground truth generalizability affects performance of the artificial intelligence model in automated vertebral fracture detection on plain lateral radiographs of the spine.* Spine J, 2022. **22**(4): p. 511-523.
26. Li, Y.C., et al., *Can a deep-learning model for the automated detection of vertebral fractures approach the performance level of human subspecialists?* Clin Orthop Relat Res, 2021. **479**(7): p. 1598-1612.
27. Chen, H.Y., et al., *Application of deep learning algorithm to detect and visualize vertebral fractures on plain frontal radiographs.* PLoS One, 2021. **16**(1): p. e0245992.
28. Murata, K., et al., *Artificial intelligence for the detection of vertebral fractures on plain spinal radiography.* Sci Rep, 2020. **10**(1): p. 20031.
29. Xiao, B.H., et al., *A software program for automated compressive vertebral fracture detection on elderly women's lateral chest radiograph: Ofeye 1.0.* Quant Imaging Med Surg, 2022. **12**(8): p. 4259-4271.
30. Staff, T. *IMV: DR use rises; procedures grow while installed units drop*. 2011 [cited 2024 2/13]; Available from: https://healthimaging.com/topics/healthcare-management/medical-practice-management/ahra-2011/imv-dr-use-rises-procedures-grow.
31. Bolotin, H.H., *DXA in vivo BMD methodology: an erroneous and misleading research and clinical gauge of bone mineral status, bone fragility, and bone remodelling.* Bone, 2007. **41**(1): p. 138-154.
32. Kim, T.Y. and A.L. Schafer, *Variability in DXA reporting and other challenges in osteoporosis evaluation.* JAMA Intern Med, 2016. **176**(3): p. 393-395.

33. Carberry, G.A., et al., *Unreported vertebral body compression fractures at abdominal multidetector CT.* Radiology, 2013. **268**(1): p. 120-126.
34. Chang, B.C., et al., *Using an ensemble of segmentation methods to detect vertebral bodies on radiographs.* AJNR, 2024.
35. Orwoll, E., et al., *Design and baseline characteristics of the osteoporotic fractures in men (MrOS) study — a large observational study of the determinants of fracture in older men.* Contemp Clin Trials, 2005. **26**(5): p. 569-585.
36. Blank, J.B., et al., *Overview of recruitment for the osteoporotic fractures in men study (MrOS).* Contemp Clin Trials, 2005. **26**(5): p. 557-568.
37. Dong, Q., et al., *Generalizability of deep learning classification of spinal osteoporotic compression fractures on radiographs using an adaptation of the modified-2 algorithm-based qualitative criteria.* Acad Radiol, 2023. **30**(12): p. 2973-2987.
38. Pianykh, O.S., *Digital Imaging and Communications in Medicine (DICOM)*. 2nd ed. 2012, Berlin, Heidelberg: Springer.
39. *Labelbox*. [cited 2024 2/13]; Available from: https://labelbox.com.
40. *DataTurks*. [cited 2024 2/13]; Available from: https://github.com/DataTurks/DataTurks.
41. Halaschek-Wiener, C., et al., *Photostuff-an image annotation tool for the semantic web*, in *Proceedings of ISWC*. 2005. p. 6-10.
42. Tuffield, M., et al., *Image annotation with photocopain*, in *Proceedings of SWAMM@WWW*. 2006.
43. Saathoff, C., S. Schenk, and A. Scherp, *Kat: the k-space annotation tool*, in *Proccedings of the SAMT*. 2008.
44. Giró-i-Nieto, X., N. Camps, and F. Marqués, *GAT: a Graphical Annotation Tool for semantic regions.* Multim. Tools Appl., 2010. **46**(2-3): p. 155-174.
45. Dutta, A. and A. Zisserman, *The VIA annotation software for images, audio and video*, in *Proceedings of ACM Multimedia*. 2019. p. 2276-2279.
46. Rueden, C.T., et al., *ImageJ2: ImageJ for the next generation of scientific image data.* BMC Bioinformatics, 2017. **18**(1): p. 529.
47. McAuliffe, M.J., et al., *Medical image processing, analysis & visualization in clinical research*, in *Proceedings of CBMS*. 2001. p. 381.
48. Philbrick, K.A., et al., *RIL-Contour: a medical imaging dataset annotation tool for and with deep learning.* J Digit Imaging, 2019. **32**(4): p. 571-581.
49. Rubin, D.L., et al., *ePAD: an image annotation and analysis platform for quantitative imaging.* Tomography, 2019. **5**(1): p. 170-183.
50. *MD.ai. The platform for medical AI*. [cited 2024 2/13]; Available from: https://www.md.ai.
51. Dong, Q., et al., *DicomAnnotator: a configurable open-source software program for efficient DICOM image annotation.* J Digit Imaging, 2020. **33**(6): p. 1514-1526.
52. Khan, S.H., et al., *A Guide to Convolutional Neural Networks for Computer Vision*. 2018: Morgan & Claypool Publishers.
53. Sun, C., et al., *Revisiting unreasonable effectiveness of data in deep learning era*, in *Proceedings of ICCV*. 2017. p. 843-852.
54. Weyand, T., I. Kostrikov, and J. Philbin, *PlaNet — photo geolocation with convolutional neural networks*, in *Proceedings of ECCV*. 2016. p. 37-55.
55. Colón-Ruiz, C. and I. Segura-Bedmar, *Comparing deep learning architectures for sentiment analysis on drug reviews.* J. Biomed. Informatics, 2020. **110**: p. 103539.
56. Devlin, J., et al., *BERT: pre-training of deep bidirectional transformers for language understanding*, in *Proceedings of NAACL-HLT*. 2019. p. 4171-4186.
57. Guo, Z., et al., *Deep learning-based image segmentation on multimodal medical imaging.* IEEE Trans Radiat Plasma Med Sci, 2019. **3**(2): p. 162-169.
58. Korot, E., et al., *Code-free deep learning for multi-modality medical image classification.* Nat. Mach. Intell., 2021. **3**(4): p. 288-298.
59. Nielsen, J., *Usability engineering*. 1993: Academic Press.
60. *Keras integration with TQDM progress bars. GitHub*. [cited 2024 2/13]; Available from: https://github.com/bstriner/keras-tqdm.
61. *TensorBoard: visualization learning. GitHub*. [cited 2024 2/13]; Available from: https://www.tensorflow.org/tensorboard/r1/summaries.
62. Goodfellow, I.J., Y. Bengio, and A.C. Courville, *Deep Learning*. 2016: MIT Press.
63. Dong, Q., et al., *Deep learning classification of spinal osteoporotic compression fractures on radiographs using an adaptation of the Genant semiquantitative criteria.* Acad Radiol, 2022. **29**(12): p. 1819-1832.
64. Dong, Q. and G. Luo, *Progress indication for deep learning model training: a feasibility demonstration.* IEEE Access, 2020. **8**: p. 79811-79843.
65. Dong, Q., X. Zhang, and G. Luo, *Improving the accuracy of progress indication for constructing deep learning models.* IEEE Access, 2022. **10**: p. 63754-63781.
66. *TensorFlow*. [cited 2024 2/26]; Available from: https://www.tensorflow.org.
67. Ruhan, S., et al., *Intervertebral disc detection in X-ray images using faster R-CNN.* Annu Int Conf IEEE Eng Med Biol Soc, 2017. **2017**: p. 564-567.
68. Esteva, A., et al., *Dermatologist-level classification of skin cancer with deep neural networks.* Nature, 2017. **542**(7639): p. 115-118.
69. Lowekamp, B.C., et al., *The design of SimpleITK.* Front Neuroinform, 2013. **7**: p. 45.
70. Masud, T., et al., *Assessment of osteopenia from spine radiographs using two different methods: the Chingford Study.* Br J Radiol, 1996. **69**(821): p. 451-456.
71. Genant, H.K., et al., *Vertebral fracture assessment using a semiquantitative technique.* J Bone Miner Res, 1993. **8**(9): p. 1137-1148.
72. Jiang, G., et al., *Comparison of methods for the visual identification of prevalent vertebral fracture in osteoporosis.* Osteoporos Int, 2004. **15**(11): p. 887-896.
73. Lentle, B.C., et al., *Comparative analysis of the radiology of osteoporotic vertebral fractures in women and men: cross-sectional and longitudinal observations from the Canadian multicentre osteoporosis study (CaMos).* J Bone Miner Res, 2018. **33**(4): p. 569-579.
74. Saville, P.D., *A quantitative approach to simple radiographic diagnosis of osteoporosis: its application to the osteoporosis of rheumatoid arthritis.* Arthritis Rheum, 1967. **10**(5): p. 416-422.
75. Gonzalez, R.C. and R.E. Woods, *Digital Image Processing*. 2018: Pearson.
76. DeVito Dabbs, A., et al., *User-centered design and interactive health technologies for patients.* Comput Inform Nurs, 2009. **27**(3): p. 175-183.
77. *LireNLPSystem package documentation*. [cited 2024 2/15]; Available from: https://github.com/UW-CLEAR-Center/LireNLPSystem.

78. Aaltonen, H.L., et al., *m2ABQ — a proposed refinement of the modified algorithm-based qualitative classification of osteoporotic vertebral fractures.* Osteoporos Int, 2023. **34**(1): p. 137-145.
79. Cawthon, P.M., et al., *Methods and reliability of radiographic vertebral fracture detection in older men: the osteoporotic fractures in men study*. Bone, 2014. **67**: p. 152-155.
80. *Photometric Interpretation Attribute*. [cited 2024 2/16]; Available from: https://dicom.innolitics.com/ciods/rt-dose/image-pixel/00280004.
81. *imgaug*. [cited 2024 2/16]; Available from: https://imgaug.readthedocs.io/en/latest/source/api_imgaug.html.
82. Szegedy, C., et al., *Going deeper with convolutions*, in *Proceedings of CVPR*. 2015. p. 1-9.
83. Szegedy, C., et al., *Inception-v4, Inception-ResNet and the impact of residual connections on learning*, in *Proceedings of AAAI*. 2017. p. 4278-4284.
84. Tan, M. and Q.V. Le, *EfficientNet: rethinking model scaling for convolutional neural networks*, in *Proceedings of ICML*. 2019. p. 6105-6114.
85. Deng, J., et al., *ImageNet: a large-scale hierarchical image database*, in *Proceedings of CVPR*. 2009. p. 248-255.
86. *TF-Slim: a high level library to define complex models in TensorFlow*. [cited 2024 2/16]; Available from: https://blog.research.google/2016/08/tf-slim-high-level-library-to-define.html.
87. *TensorFlow model garden*. [cited 2024 2/16]; Available from: https://github.com/tensorflow/models.
88. *EfficientNet Keras (and TensorFlow Keras)*. [cited 2024 2/16]; Available from: https://github.com/qubvel/efficientnet.
89. He, K., et al., *Delving deep into rectifiers: surpassing human-level performance on ImageNet classification*, in *Proceedings of ICCV*. 2015. p. 1026-1034.
90. Kingma, D.P. and J. Ba, *Adam: a method for stochastic optimization*, in *ICLR (Poster)*. 2015.
91. *tf.nn.weighted_cross_entropy_with_logits*. [cited 2024 2/16]; Available from: https://www.tensorflow.org/api_docs/python/tf/nn/weighted_cross_entropy_with_logits.
92. Srivastava, N., et al., *Dropout: a simple way to prevent neural networks from overfitting.* J. Mach. Learn. Res., 2014. **15**(1): p. 1929-1958.
93. *inception_v1.py*. [cited 2024 2/16]; Available from: https://github.com/tensorflow/models/blob/master/research/slim/nets/inception_v1.py.
94. *inception_resnet_v2.py*. [cited 2024 2/16]; Available from: https://github.com/tensorflow/models/blob/master/research/slim/nets/inception_resnet_v2.py.
95. Kripke, C., *Pap smear vs. HPV screening tests for cervical cancer.* American Family Physician, 2008. **77**(12): p. 1740-1742.
96. Wang, Y.X.J., et al., *'Healthier Chinese spine': an update of osteoporotic fractures in men (MrOS) and in women (MsOS) Hong Kong spine radiograph studies.* Quant Imaging Med Surg, 2022. **12**(3): p. 2090-2105.
97. Luo, G., *Toward a progress indicator for machine learning model building and data mining algorithm execution: a position paper.* SIGKDD Explor., 2017. **19**(2): p. 13-24.
98. Luo, G., J.F. Naughton, and P.S. Yu, *Multi-query SQL progress indicators*, in *Proceedings of EDBT*. 2006. p. 921-941.
99. Goyal, P., et al., *Accurate, large minibatch SGD: training ImageNet in 1 hour*, in *arXiv: 1706.02677*. 2017.
100. Russakovsky, O., et al., *ImageNet large scale visual recognition challenge.* Int. J. Comput. Vis., 2015. **115**(3): p. 211-252.
101. *Tensorpack*. [cited 2024 2/16]; Available from: https://github.com/tensorpack/tensorpack.
102. Lee, W., H. Oh, and K. Yi, *A progress bar for static analyzers*, in *Proceedings of SAS*. 2014. p. 184-200.
103. Wang, K., et al., *A progress bar for the JPF search using program executions.* ACM SIGSOFT Softw. Eng. Notes, 2018. **43**(4): p. 55.
104. Luo, G., T. Chen, and H. Yu, *Toward a progress indicator for program compilation.* Softw. Pract. Exp., 2007. **37**(9): p. 909-933.
105. Chaudhuri, S., V. Narasayya, and R. Ramamurthy, *Estimating progress of execution for SQL queries*, in *Proceedings of SIGMOD Conference*. 2004. p. 803-814.
106. Lee, K., et al., *Operator and query progress estimation in Microsoft SQL server live query statistics*, in *Proceedings of SIGMOD Conference*. 2016. p. 1753-1764.
107. Luo, G., et al., *Increasing the accuracy and coverage of SQL progress indicators*, in *Proceedings of ICDE*. 2005. p. 853-864.
108. Luo, G., et al., *Toward a progress indicator for database queries*, in *Proceedings of SIGMOD Conference*. 2004. p. 791-802.
109. Morton, K., M. Balazinska, and D. Grossman, *ParaTimer: a progress indicator for MapReduce DAGs*, in *Proceedings of SIGMOD Conference*. 2010. p. 507-518.
110. Morton, K., et al., *Estimating the progress of MapReduce pipelines*, in *Proceedings of ICDE*. 2010. p. 681-684.
111. Xie, X., et al., *PIGEON: progress indicator for subgraph queries*, in *Proceedings of ICDE*. 2015. p. 1492-1495.
112. Luo, G., *PredicT-ML: a tool for automating machine learning model building with big clinical data.* Health Inf. Sci. Syst., 2016. **4**(1): p. 5.
113. Luo, G., et al., *Automating construction of machine learning models with clinical big data: proposal rationale and methods.* JMIR Res Protoc, 2017. **6**(8): p. e175.
114. Luo, G., *Progress indication for machine learning model building: a feasibility demonstration.* SIGKDD Explor, 2018. **20**(2): p. 1-12.
115. Dong, Q. and G. Luo, *Progress estimation for end-to-end training of deep learning models with online data preprocessing.* IEEE Access, 2024.
116. Justus, D., et al., *Predicting the computational cost of deep learning models*, in *Proceedings of IEEE BigData*. 2018. p. 3873-3882.
117. Reif, M., F. Shafait, and A. Dengel, *Prediction of classifier training time including parameter optimization*, in *Proceedings of KI*. 2011. p. 260-271.
118. Doan, T. and J. Kalita, *Predicting run time of classification algorithms using meta-learning.* Int. J. Mach. Learn. Cybern., 2017. **8**(6): p. 1929-1943.
119. Yang, C., et al., *OBOE: collaborative filtering for AutoML model selection*, in *Proceedings of KDD*. 2019. p. 1173-1183.
120. Snoek, J., H. Larochelle, and R.P. Adams, *Practical Bayesian optimization of machine learning algorithms*, in *Proceedings of NIPS*. 2012. p. 2960-2968.
121. Anthony, M. and P.L. Bartlett, *Neural Network Learning — Theoretical Foundations*. 2002: Cambridge University Press.
122. Livni, R., S. Shalev-Shwartz, and O. Shamir, *On the computational efficiency of training neural networks*, in *Proceedings of NIPS*. 2014. p. 855-863.
123. Prechelt, L., *Early stopping — but when?*, in *Neural Networks: Tricks of the Trade — Second Edition*. 2012. p. 53-67.

124. Duvenaud, D., D. Maclaurin, and R.P. Adams, *Early stopping as nonparametric variational inference*, in *Proceedings of AISTATS*. 2016. p. 1070-1077.
125. Mahsereci, M., et al., *Early stopping without a validation set*, in *arXiv: 1703.09580*. 2017.
126. *tf.keras.callbacks.EarlyStopping*. [cited 2024 2/16]; Available from: https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/keras/callbacks/EarlyStopping.
127. Hernandez, D., et al., *Scaling laws for transfer*, in *arXiv: 2102.01293*. 2021.
128. Kaplan, J., et al., *Scaling laws for neural language models*, in *arXiv: 2001.08361*. 2020.
129. Henighan, T., et al., *Scaling laws for autoregressive generative modeling*, in *arXiv: 2010.14701*. 2020.
130. Komatsuzaki, A., *One epoch is all you need*, in *arXiv: 1906.06669*. 2019.
131. Nocedal, J.W., Stephen J., *Numerical Optimization*. 2nd ed. 2006, New York, NY, USA: Springer.
132. Huber, P.J.R., Elvezio M., *Robust Statistics*. 2nd ed. 2011, Hoboken, NJ, USA: Wiley.
133. Mohri, M., A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. 2012: MIT Press.
134. Rohatgi, V.K.S., A. K. Md. Ehsanes, *An Introduction to Probability and Statistics*. 3rd ed. 2015, Hoboken, NJ, USA: Wiley.
135. *Intermediate value theorem*. [cited 2024 2/17]; Available from: https://en.wikipedia.org/wiki/Intermediate_value_theorem.
136. *Normal approximation to the beta distribution*. [cited 2024 2/18]; Available from: https://www.vosesoftware.com/riskwiki/NormalapproximationtotheBetadistribution.php.
137. *Artelys Knitro user's manual: algorithms*. [cited 2024 2/18]; Available from: https://www.artelys.com/docs/knitro/2_userGuide/algorithms.html.
138. *The most advanced solver for nonlinear optimization*. [cited 2014 2/18]; Available from: https://www.artelys.com/solvers/knitro.
139. Purushotham, S., et al., *Benchmarking deep learning models on large healthcare datasets.* J Biomed Inform, 2018. **83**: p. 112-134.
140. Aatila, M., M. Lachgar, and A. Kartit, *An overview of gradient descent algorithm optimization in machine learning: application in the ophthalmology field*, in *Proceedings of SADASC*. 2020. p. 349-359.
141. Bottou, L., *Large-scale machine learning with stochastic gradient descent*, in *Proceedings of COMPSTAT*. 2010. p. 177-186.
142. Duchi, J.C., E. Hazan, and Y. Singer, *Adaptive subgradient methods for online learning and stochastic optimization.* J. Mach. Learn. Res., 2011. **12**: p. 2121-2159.
143. Krizhevsky, A., *Learning multiple layers of features from tiny images*, in *Department of Computer Science*. 2009, University of Toronto, Toronto, Canada.
144. Johnson, A.E., et al., *MIMIC-III, a freely accessible critical care database.* Sci Data, 2016. **3**: p. 160035.
145. *GoogLeNet-Inception*. [cited 2024 2/19]; Available from: https://github.com/conan7882/GoogLeNet-Inception.
146. *Benchmarking_DL_MIMICIII*. [cited 2024 2/19]; Available from: https://github.com/USC-Melady/Benchmarking_DL_MIMICIIIhttps://github.com/USC-Melady/Benchmarking_DL_MIMICIII.
147. Chaudhuri, S., R. Kaushik, and R. Ramamurthy, *When can we trust progress estimators for SQL queries?*, in *Proceedings of SIGMOD Conference*. 2005. p. 575-586.

VITA

QIFEI DONG received the B.S. degree in electrical engineering from Zhejiang University, Hangzhou, Zhejiang Province, P.R. China, in 2016 and the M.S. degree in electrical and computer engineering from the University of Michigan, Ann Arbor, MI, USA, in 2018. He pursued the PhD degree in biomedical informatics and medical education at the University of Washington, Seattle, WA, USA.

Since 2018, he has been a Research Assistant with the University of Washington Clinical Learning, Evidence and Research Center for Musculoskeletal Disorders, Seattle, WA, USA. His research interests include machine learning, computer vision, natural language processing, and clinical informatics.