Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Phu Pham

Deep learning methods for modeling forest biomass and structures from hyperspectral imagery

Master's Thesis
Espoo, May 27, 2019

Supervisor: D.Sc. (Tech.) Jorma Laaksonen
Advisor: D.Sc. (Tech.) Jorma Laaksonen

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Phu Pham
Title:	Deep learning methods for modeling forest biomass and structures from hyperspectral imagery
Date:	May 27, 2019 **Pages:** 86
Supervisor:	D.Sc. (Tech.) Jorma Laaksonen
Advisor:	D.Sc. (Tech.) Jorma Laaksonen

Forests affect the environment and ecosystems in multiple ways. Hence, understanding the forest processes and vegetation characteristics help us protect the environment better, reserve the biodiversity, and mitigate the hazardous impacts of climate change. There are studies in hyperspectral remote sensing that employ both empirical and artificial intelligence (AI) methods to analyze and predict the vegetation parameters. However, these methods have weaknesses. First, the empirical methods are inefficient because they cannot fully utilize a large amount of hyperspectral data. Secondly, even though the existing AI-based methods can achieve remarkable results, they are only validated on small-scale datasets that have simple forest structures. Thus, a robust technique that can effectively model complex forest structures on large-scale datasets is an open challenge.

This thesis directly addresses the challenge by proposing a novel deep learning architecture that can jointly learn and model four discrete and twelve continuous forest parameters. The final model is comprised of three 3D convolution layers, a 3D multi-scale convolution block, a shared fully-connected layer, and two fully-connected layers for each learning task. The model uses a loss, namely focal loss, to address class imbalance problem and the gradient normalization for multi-task learning.

Then, we record and compare the results of our comprehensive experiments. Overall, the proposed model reaches 78.32% class-balanced accuracy for the four classification tasks. For the regression tasks, the model achieves a notably low average mean absolute error (0.052) and high Pearson correlation coefficient (≈ 0.9) between predicted and target labels. In the end, the shortcomings of the thesis work are discussed and potential research areas for future work are suggested.

Keywords:	Deep learning, Convolutional neural network, Multi-task learning, Computer vision, Hyperspectral remote sensing, Imbalanced data
Language:	English

Acknowledgements

My Master's program at Aalto University has given me the opportunities to explore the areas of machine learning and artificial intelligence. I am grateful for the academic knowledge and expertise that I've gained during my time at Aalto.

This work has been done under the supervision of my supervisor and advisor, Dr. Jorma Laaksonen. I want to express my sincere gratitude and respect to Dr. Jorma for all of his guidance, support, and thorough feedback on my thesis work. Additionally, I would like to thank him for the opportunity to join the Content-Based Image and Information Retrieval (CBIR) group. Here, I had the chance to learn from and work with my incredible colleagues in the group.

I wish to thank the Artificial Intelligence for Retrieval of Forest Biomass & Structure (AIROBEST) team, including Dr. Matti Mõttus, M.Sc. Eelis Halme, and M.Sc. Matthieu Molinier, for their support and perspicacious discussions. I also want to thank the Academy of Finland for the funding of this research, and the CSC-IT center for science and the Aalto Science-IT for providing the computational power.

I want to thank my colleagues Aditya Kaushik, Arturs Polis, and Héctor Laria Mantecón for the valuable white-board discussions we had.

I would like to offer my special thanks to my family for their unending love and inspiration, to my friends in Vietnam, Finland, and the US for their support, and finally, to my little bunny Quin for being the source of motivation.

Espoo, May 27, 2019

Phu Pham

Acronyms and Symbols

Acronyms	**Expansions**
AI	Artificial Intelligence
AUC	Area Under Curve
BA	Balanced Accuracy
BN	Batch Normalization
CE	Cross Entropy
CNN	Convolutional Neural Network
CRL	Class Rectification Loss
FC	Fully-connected
FN	False Negative
FP	False Positive
FPR	False Positive Rate
HSI	Hyperspectral image
MAE	Mean Absolute Error
MLP	Multilayer Perceptron
MSE	Mean Squared Error
MTL	Multi-task Learning
OA	Overall Accuracy
ReLU	Rectified Linear Unit
RMSE	Root Mean Squared Error

ROC	Receiver Operating Characteristics
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SOTA	State-of-the-art
TN	True Negative
TP	True Positive
TPR	True Positive Rate

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A forest is a complex ecosystem that includes numerous types of trees, plants, animals and potentially undiscovered living species. Forests provide protective canopies that prevent soil erosion, clean the air, conserve heat and lessen the impacts of natural disasters. Moreover, since the forest trees produce O_2 and absorb CO_2, forests contribute significantly to the world's carbon stock (estimated around 32.6 gigatonnes per year [45]). All of these reasons make *forestry*, an applied science that studies forests and their ecosystem, an important research field. Collecting forest information and statistics enables forest planning and management which, in turn, may result in long-term benefits for the environment. However, since forests, which cover a third of the earth's surface, are distributed around the globe, monitoring the forest characteristics becomes an arduous task.

To address this problem, remote sensing, a method to obtain information about an object without physical contact, is the only feasible approach to study forests on a large scale. Satellite and airborne images taken by recent remote sensors provide earth observation data with higher spatial and spectral resolutions. This type of hyperspectral data presents useful information on many characteristics of vegetation, such as woody biomass, mean height, and leaf area index. Due to the exponential growth in the amount of remote sensing data, existing data extraction algorithms such as table look-up [42] or physically-based inversion [25] become inefficient since they cannot fully utilize the data. More advanced data extraction approaches are needed to take advantage of the information-rich data.

Artificial intelligence (AI) has recently shown its superior performance when dealing with an abundant amount of training data. AI-based approaches are efficient and flexible for handling a large volume of data, thereby becoming promising options to surpass the existing empirical methods. In fact, there have been studies in hyperspectral remote sensing that have

adopted AI in their research work [1, 6, 7, 19, 32, 38, 57]. Most of these studies focused on a small-scale dataset comprised of canopies with simple structures. In practice, however, extracting more complicated structured vegetation from a large-scale dataset is often more beneficial. Recent developments in deep learning, a subfield of AI, suggest that neural networks can learn hidden and complex structures from data. The fact that some state-of-the-art (SOTA) models can even surpass human level in object detection and recognition, video gaming, etc. makes deep learning a top choice for computer vision tasks.

Therefore, the goal of this thesis is to develop a novel deep learning model that can retrieve structural parameters of the forest under study. To be more specific, this thesis will examine various neural network architectures to model forest biomass and structures from a hyperspectral image (HSI).

1.1 Problem statement

Given a hyperspectral image, the task of our research group is to train a model that can accurately predict 16 variables that represent canopy structures and characteristics from an unseen hyperspectral image. These variables can be either continuous or discrete. There have been active efforts to use deep learning approaches for HSI classification [6, 7, 19, 32, 38]. However, since no standard large-scale HSI dataset is publicly available, most of these approaches were designed for small-scale datasets. Moreover, they were trained for a single classification task only. On the other hand, this thesis work deals with a new dataset which is significantly larger than existing HSI datasets. As a result, the recent deep learning approaches in this domain cannot be directly applied to our problem and thus a new model is required. To this end, we will construct a convolutional neural network that is able to jointly learn the data distributions of multiple tasks.

Our system comprises of two stages: feature extraction and prediction. In the feature extraction stage, the model learns the mapping between the input image and a fixed-length feature vector. This feature vector is then fed into the task-specific layers for the prediction stage. In this work, we focus more on the feature extraction stage to learn the shared representation. We will experiment with various architectural designs including the depth and width of the network as well as regularization techniques. For the classification variables, we will compare several methods to tackle the imbalanced data. Regression variables do not need much attention since the network is already good at learning the true distributions. Finally, we will explore the options to balance the training rate between tasks in a multi-task learning setting.

Our final model will be the one that yields the best results evaluated by standard metrics.

1.2 Structure of the thesis

The remainder of the thesis is structured as follows. Chapter 2 reviews some background information related to the task of modeling forest parameters using deep learning methods. Chapter 3 describes the HSI dataset that is used. In Chapter 4, some metrics to evaluate the performance of the model are discussed in detail. A baseline model is introduced in Chapter 5. Next, Chapter 6 explains the process of how the final model architecture was derived in the course of the work. Chapter 7 discusses the experiments and analyzes the results. Finally, Chapter 8 concludes the thesis and suggests directions for future research work.

Chapter 2

Background

In this chapter, we will review some of the basic concepts related to the thesis work. The background concepts include remote sensing, machine learning, deep learning and its basic components, optimization and regularization techniques. Additionally, an overview of multi-task learning will be discussed.

2.1 Remote sensing

Remote sensing refers to the science of acquiring information about an object or area from a remote distance using a recording device (airplanes, satellites, or drones) that is not in physical contact with the targeted object or area [4].

The remote sensors collect data by measuring the energy that is reflected and emitted from the earth's surface. The important advantage of these sensors is that they record the whole range of wavelengths in the electromagnetic spectrum. Since our human eyes can only respond to wavelengths ranging from 380 to 740 nanometers, we miss out on a lot of useful information from the wavelengths beyond the visible spectrum. By scrutinizing the spectral bands (or groups of continuous wavelengths), we can study other properties of the targeted object such as vegetation structures, land cover, soil types, etc. This information enables applications of remote sensing imagery in different fields such as coastal and forest protection, hazard assessment, natural resource management, and many others.

2.2 Machine learning

Machine learning is a subfield of AI that studies and models computer algorithms to perform a specific task without being programmed explicitly. In

other words, it focuses on detecting hidden patterns from the sample data (i.e. training data). These patterns help us to understand the data generation process which may allow us to predict unseen data in the future. Machine learning algorithms are utilized in decision-making, email filtering, self-driving cars, speech recognition, computer vision, and so on.

2.3 Deep learning

Deep learning is a machine learning approach that emulates the learning process of the human brain for tasks requiring a high level of abstraction such as pattern recognition and decision-making.

2.3.1 Deep forward networks

A deep feedforward network, also called a feedforward neural network, or multilayer perceptron (MLP) [48], is the simplest type of artificial neural network and is also considered the most important type of deep learning model. Deep feedforward networks are the building blocks of many important neural networks such as recurrent neural networks [20, 52] and convolutional neural networks [29, 30]. The goal of a feedforward network is to learn an unknown mapping f with parameters $\boldsymbol{\theta}$: $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ that transforms an input $\mathbf{x}$ to a corresponding output $\mathbf{y}$. A feedforward network learns the best value of the parameters $\boldsymbol{\theta}$ that best approximate the true function f^*.

A typical neural network includes an input layer, some optional hidden layers, and an output layer. If the network does not contain a hidden layer, it is classified as a single-layer neural network. Otherwise, it is classified as a multilayer neural network.

In a feedforward network, the information flows in only one forward direction. It flows first from the input layer, then through hidden layers, and finally to the output layers. There are no loops or cycles in the network in which the model's outputs are fed back into itself. The deeper the neural network is, the more function parameters it has, and thus is able to approximate a more complicated mapping function.

2.3.2 Activation functions

An activation function is a non-linear function applied to the output of a neuron to introduce non-linearity to a neural network. Nonlinearity is crucial for a network to learn complex functional mappings from inputs to outputs. Without nonlinearity, no matter how deep the network is, it can always be

replaced by a single-layer network. Specifically, all the model can learn is just a linear function which has limited power and is not useful in most cases. Real-world problems are hard to approximate since the relationship between the input and the output is complicated. Hence, they often require a high capacity neural network that can learn a complex, non-linear mapping function.

Furthermore, an activation function also has a normalizing effect since it squashes the output amplitude of a neuron to a finite range. This behavior prevents the output of each neuron from becoming too large or too small due to the cascading effect after passing through several layers in the network [17].

There are a few popular activation functions: sigmoid, tanh and ReLU. Among the three, ReLU [43] is most frequently used in recent neural network architectures due to its cheap derivative computation. The mathematical formulas and graphs of these functions are shown in Figure 2.1.

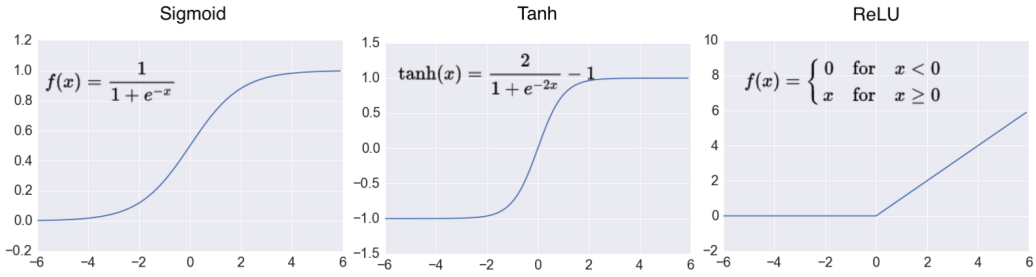


Figure 2.1: The three mostly used activation functions.[1]

2.3.3 Loss functions

A machine learning model is represented by its parameters and the ultimate goal is to learn these model parameters such that the model yields minimal errors on future predictions. In order to evaluate the performance of a model, we need a loss function that measures how much the predicted values differ from the actual values. This loss function should return a large error value if the prediction deviates too much from the ground truth and vice versa.

Since there is no universal loss function that fits all problems, there are different loss functions for different problems. Hence, choosing the right loss function is an important aspect in the design of a machine learning algorithm.

[1] http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe

Depending on a specific problem that is being solved, loss functions can be classified into two main categories: regression losses and classification losses.

Consider a dataset with N data samples. Let y_i and $\hat{y}_i$ be the target and predicted label for the i^{th} data sample.

For regression, y_i and $\hat{y}_i$ have continuous values. The common loss functions for regression tasks are mean squared error, mean absolute error and Huber loss. For classification, y_i and $\hat{y}_i$ have discrete values. Common loss functions for classification are cross-entropy loss and Hinge loss.

2.3.3.1 Mean squared error

Mean squared error (MSE) loss function aims to minimize the sum of squared differences between predicted and target labels. The standard form of MSE is defined as:

$$J = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \,, \tag{2.1}$$

in which the quantity $(y_i - \hat{y}_i)$ is called the residual.

MSE is sometimes called quadratic or L2 loss. MSE and L2 loss are not the same but they are very similar. The only difference is that L2 loss does not have the term $\frac{1}{N}$ in its formula. In the context of optimization, this term is just a scaling factor and does not affect the learning process.

2.3.3.2 Mean absolute error

Mean absolute error (MAE) minimizes the average sum of the absolute differences between the target and predicted labels. MSE loss can be written as follows:

$$J = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \,. \tag{2.2}$$

MAE is sometimes referred to as L1 loss. Similar to the subtle difference between MSE and L2 loss, L1 is MAE without the average term $\frac{1}{N}$. MAE is not differentiable at zero due to the absolute operation.

2.3.3.3 Huber loss

Huber loss [22] can be considered as both L1 and L2 losses. To be more specific, Huber loss becomes L2 loss when the magnitude of the residual $y_i - \hat{y}_i$ is smaller than a certain threshold δ and becomes L1 loss otherwise.

The formula for Huber loss is as follows:

$$J = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2 & \text{if } |y_i - \hat{y}_i| \leq \delta \ , \\ \delta|y_i - \hat{y}_i| - \frac{1}{2}\delta^2 & \text{otherwise} \ , \end{cases} \tag{2.3}$$

where δ is a tunable hyperparameter.

Since the threshold δ is small, Huber loss is likely to treat residuals with L1 loss. Moreover, Huber loss is always differentiable; it is sometimes called smooth L1 loss.

2.3.3.4 Cross-entropy loss

Given that $y_i \in \{0, 1\}$ and $\hat{y}_i \in [0, 1]$, the cross-entropy (CE) loss for binary classification is computed by:

$$J = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \ . \tag{2.4}$$

CE loss optimizes the difference between target and predicted distributions. If the cross entropy is large, the difference between the two probability distributions are large and vice versa. For multi-class classification, binary CE loss is computed for each class. The final loss is the sum of all class losses.

2.3.3.5 Hinge loss

Hinge loss aims to maximize the margin between the decision boundary and the data points to ensure that the data points are classified correctly with high confidence. In the case of hinge loss, $y_i \in \{-1, 1\}$ and $\hat{y}_i$ is the raw ouput or the predicted probability that it belongs to a class. Hinge loss is calculated as:

$$J = \frac{1}{N}\sum_{i=1}^{N} \max(0, 1 - y_i \cdot \hat{y}_i) \ . \tag{2.5}$$

Equation (2.5) shows that hinge loss penalizes not only the wrong predictions but also correct predictions with low confidence. Hinge loss equals zero only when the data is correctly predicted with absolute confidence (-1 or $+1$).

To train a classifier with hinge loss for a multi-class classification, we train multiple classifiers, one for each class and treat the other classes as one common class.

2.3.4 Backpropagation

The process in which the information flows from an input $\mathbf{x}$ through the network to produce an output $\mathbf{y}$ is known as a forward propagation. The output is then fed into a loss function to compute a scalar cost $J(\boldsymbol{\theta})$. When the cost or error is propagated back to the network, the network can use this information to optimize the learning process by using gradient descent. This process is called backpropagation or backprop.

Backprop was first introduced in the 1970s as an optimization method for automatic differentiation of a complex function [37]. However, it did not get much attention until a paper by Rumelhart et al. was published in 1986 [52]. The paper indicates that backprop works much better than earlier approaches in representation learning algorithms and thus opens new possibilities to use artificial neural networks to solve the seemingly insoluble problems.

For simplicity, assume the bias term b is equal to zero. The parameters $\boldsymbol{\theta}$ are now referred to as the weights $\mathbf{w}$. The main idea of backprop is to efficiently compute the gradient of the loss function $\partial J/\partial \mathbf{w}$ with respect to the weights $\mathbf{w}$ in the network. The backprop algorithm is straightforward as it follows the chain rule of calculus for derivative computation. As a quick reminder, the chain rule can be stated as follows: If y is a function of $u : y = f(u)$ and u is a function of $x : u = g(x)$, then the derivative of y with respect to x is equal to the derivative of y with respect to u times the derivative of u with respect to x:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x} \, . \tag{2.6}$$

This chain rule can be generalized beyond the scalar case, meaning that the same rule still applies when u and x are tensors. The magnitude of the gradient indicates the sensitivity of the loss corresponding to the changes in the value of weights and biases. With that said, the loss is more sensitive to the weight $\mathbf{w}$ if $\partial J/\partial \mathbf{w}$ is large and vice versa.

2.3.5 Optimization

Most machine learning algorithms can be seen as an optimization technique of some sort. A machine learning algorithm normally tries to optimize an objective function $f(\mathbf{x}, \boldsymbol{\theta})$ with regard to a set of parameters $\boldsymbol{\theta}$. This function is also called loss function, cost function or error function in the machine learning community. The optimal solution $\boldsymbol{\theta}^*$ is the one that minimizes the

function f: $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} f(\mathbf{x}, \boldsymbol{\theta})$. In case we want to maximize a function f, it is equivalent to minimizing $-f$.

2.3.5.1 Gradient descent

Gradient descent is a very common optimization method in machine learning. It is a first-order iterative approach to finding the minimum of a function. The idea of gradient descent is to use the gradient of the objective function at a certain point $\mathbf{w}$ to find the direction towards the minimum.

The simplified version of gradient descent is illustrated in Figure 2.2. The gradient of J at the point $\mathbf{w}$ tells us how a small change in $\mathbf{w}$ results in a change in J. If the gradient is positive, a small increase/decrease in the value of $\mathbf{w}$ will lead to a small increase/decrease in the value of J. On the contrary, if the gradient is negative, a small increase/decrease in the value of $\mathbf{w}$ will lead to a small decrease/increase in the value of J. Thus, moving towards the opposite direction or the descending direction of the gradient ensures a reduction in the value of J. Therefore, it is called gradient descent. In Figure 2.2, the gradient (or the slope of the dashed black line) of J associated with the initial weight $\mathbf{w}$ is positive, thus reducing $\mathbf{w}$ will make J smaller. Formally, the update rule for the weight can be defined as:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}) \,, \tag{2.7}$$

in which η, known as the learning rate, is a small scalar value chosen heuristically. $\mathbf{X}$ is a matrix of input data and $\mathbf{y}$ is the target label vector.

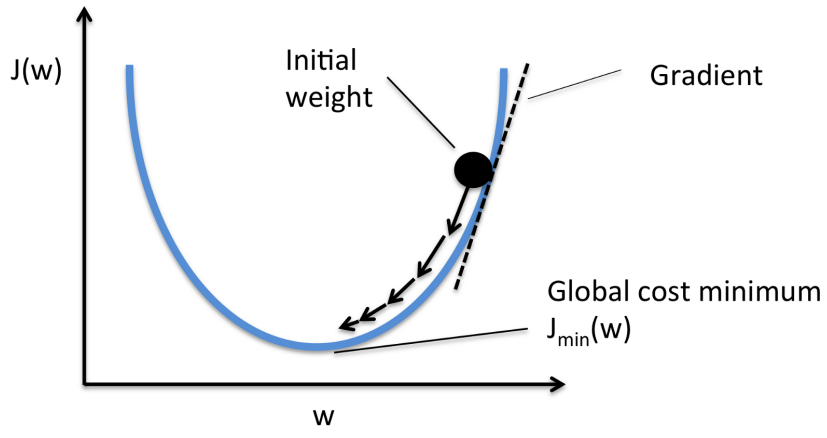


Figure 2.2: An illustration of gradient descent algorithm.[2]

[2]https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

The efficiency of gradient descent depends on the convexity of the objective function. A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex in its domain if it satisfies:

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y) \ ,$$

for all $x, y \in \mathbb{R}^n$ and all $\alpha, \beta \in [0, 1]$ such that $\alpha + \beta = 1$ [3]. A convex function, in simple terms, is one that has a bowl-like shape and thus has only one global minimum like the one in Figure 2.2. This shape guarantees that the gradient descent method can always find the minimum point at the bottom of the bowl. In real-world problems, however, the objective functions are a lot more complicated and hardly convex. There can be multiple local minima and global minima. Finding the global minimum is extremely difficult if not impossible.

Figure 2.3 indicates an example of a non-convex function. There exist both global and local minima where the gradient is zero. Since one of the local minima is very close to the global minimum, we can accept the suboptimal solutions as long as they result in significantly low values of the objective function.

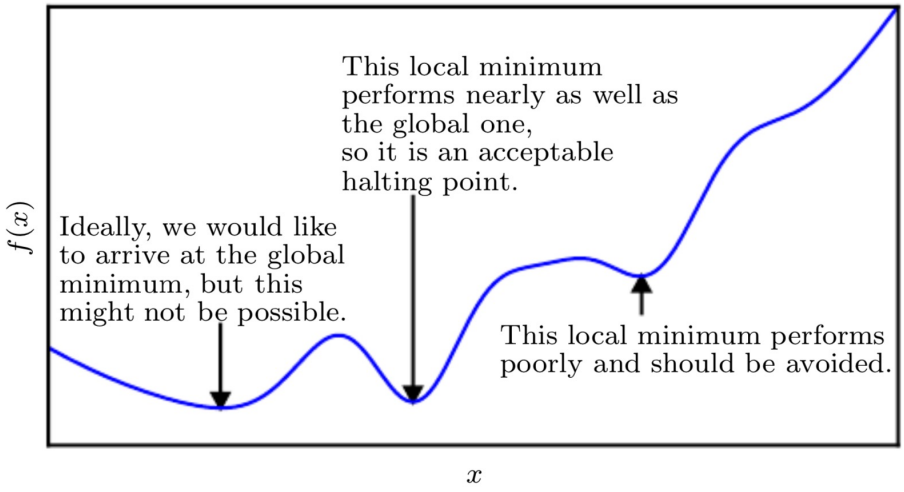


Figure 2.3: An example of a non-convex function with multiple local optima [14].

Gradient descent is a batch optimization method. In order to update the model parameters, one needs to use the entire training set to compute the gradient of the loss function. This is computationally prohibitive and sometimes intractable if the training set is too large to fit in the main memory of a single machine. Moreover, the computation has to be done at every

single training step. Since gradient descent is an iterative method, it often requires many iterations to converge to local optima, which could result in a very slow training process. To overcome this problem, we can estimate the loss by computing the average loss of a small fraction of the data and use the gradient of this mini-batch to update the parameters. There are several popular derivatives of the traditional gradient descent algorithm that adopt this mini-batch approach.

2.3.5.2 Stochastic gradient descent

Stochastic gradient descent (SGD) [2] and its variants are widely used in the machine learning community. SGD uses only one single data sample (batch size of 1) to estimate the loss at each iteration, making it a very bad estimation. SGD is much cheaper to compute but requires many iterations with a very small step size. Given that the data sample is randomly chosen, SGD gives a noisier path to local minima but a faster convergence. The parameter update is given by:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla_{\mathbf{w}} J(\mathbf{w}_t; x_i, y_i) \ , \tag{2.8}$$

in which x_i, y_i is a random pair from the training set.

In practice, we normally use a mini-batch of the training data, typically about 10 to 1000 examples, to compromise between batch gradient descent and SGD. This mini-batch SGD method brings two advantages: firstly, it reduces the variance in the parameter update, making it less noisy than SGD, but still faster to converge than batch gradient descent, and secondly, it allows more effective computation by utilizing matrix multiplications, thus reduces training time. For mini-batch SGD to work, the data needs to be shuffled before feeding to the algorithm. The reason for this step is that the model can be biased if the given data has some meaningful order, which can lead to performance degradation.

In SGD, tuning the learning rate is of crucial importance. The learning rate in SGD is usually much smaller in comparison to batch gradient descent due to the wide variance in the update of SGD. A well-tested method for choosing the right learning rate is to start with a small enough learning rate, then gradually decrease it over time. A common method to update the learning rate is to evaluate the improvement in the objective function for the validation set after each epoch. If the improvement is less than a small threshold, decrease the learning rate. Another method is to linearly decay the learning rate until iteration τ:

$$\eta_k = (1 - \alpha)\eta_0 + \alpha\eta_\tau \ , \tag{2.9}$$

with $\alpha = \frac{k}{\tau}$, k is the current iteration, η_0 is the initial learning rate. After iteration τ, the learning rate is thus kept constant. In practice, one can specify a list of values for τ to set multiple milestones for the learning rate update.

2.3.5.3 Momentum

Even though SGD is much more efficient than batch gradient descent, it can be slow in some cases. One typical case where SGD does not work well is when the path to a local minimum has a shape of a long shallow ravine, i.e. the surface curves are much steeper in one dimension than in another [49]. Since the negative gradient will point towards the steep sides, SGD will tend to oscillate between the two walls of the ravine instead of moving along the ravine itself, resulting in slow convergence. Momentum [46] is a method that can accelerate learning by pushing the objective to move in the correct direction and dampening the oscillations of SGD.

The idea of momentum is to calculate the exponentially moving average of past gradients and then update the parameters to move in those directions. The momentum update rule is given by:

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta \nabla_{\mathbf{w}} J(\mathbf{w}) \ , \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - \mathbf{v}_t \ , \end{aligned} \tag{2.10}$$

in which γ is the momentum term, usually has value of 0.9, η is the learning rate, $\mathbf{v}_t$ is the velocity vector at time t.

The effect of momentum is illustrated in Figure 2.4. Training with momentum results in a less noisy updating path and faster convergence since it does not waste time moving back and forth along the ravine as training without momentum does.

2.3.5.4 Nesterov Momentum

Nesterov Momentum is a variant of momentum algorithm, introduced by Sutskever et al. [54]. The algorithm was inspired by Nesterov's accelerated gradient method [44] which allows the algorithm to look ahead towards the next position of the parameters. We know that the momentum term of the standard momentum algorithm will nudge the parameter $\mathbf{w}$ by $\gamma \mathbf{v}_{t-1}$. Taking advantage of this prior information, we can look ahead by computing the next value of the parameter $\mathbf{w}_t - \gamma \mathbf{v}_{t-1}$ and use this future value $\mathbf{w}$ instead of the current one to calculate the gradient of the objective function. The update rules can be given by:

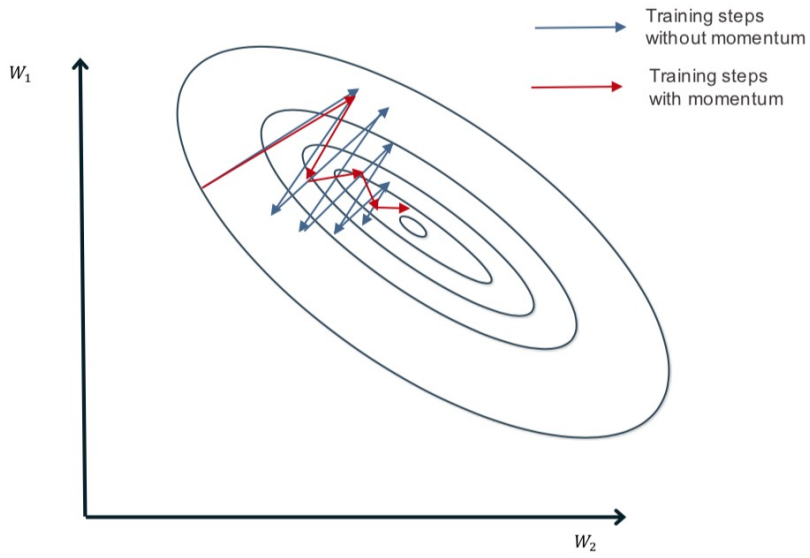


Figure 2.4: Effect of momentum update on training [27].

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta \nabla_{\mathbf{w}} J(\mathbf{w}_t - \gamma \mathbf{v}_{t-1}) \ , \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - \mathbf{v}_t \ . \end{aligned} \tag{2.11}$$

2.3.5.5 AdaGrad

AdaGrad [10] is an adaptive learning rate algorithm that applies different learning rates for each model parameters individually. Each parameter is scaled inversely to the square root of the cumulative sum of their squared value from previous time steps. The update rules of AdaGrad are as follow:

$$\begin{aligned} \mathbf{g}_t &= \nabla_{\mathbf{w}} J(\mathbf{w}_t) \ , \\ \mathbf{r}_t &= \mathbf{r}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t \ , \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{\eta}{\epsilon + \sqrt{\mathbf{r}_t}} \odot \mathbf{g}_t \ , \end{aligned} \tag{2.12}$$

in which $\mathbf{g}_t$ is the gradient at time t, $\odot$ represents the Hadamard product or element-wise product, $\mathbf{r}_t$ is the accumulated squared gradient at time t, ϵ is a smoothing term (usually in range from 10^{-4} to 10^{-8}) that avoids division by zero when $\mathbf{r}_t$ is extremely small.

AdaGrad eliminates the need to tune the learning rate manually. However, since the cumulative sum of squared gradients continuously increases during training, the learning rate excessively decreases and thus makes the model unable to learn additional knowledge.

2.3.5.6 RMSProp

RMSProp is a modification of AdaGrad proposed by Tieleman and Hinton [56] to resolve the vanishing learning rate in AdaGrad algorithm. RMSProp replaces the accumulative sum of squared gradients by an exponentially weighted moving average to converge quickly in a locally convex region.

$$\begin{aligned} \mathbf{g}_t &= \nabla_{\mathbf{w}} J(\mathbf{w}_t) \ , \\ \mathbf{r}_t &= \rho \mathbf{r}_{t-1} + (1-\rho)\mathbf{g}_t \odot \mathbf{g}_t \ , \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{\eta}{\sqrt{\mathbf{r}_t + \epsilon}} \odot \mathbf{g}_t \ , \end{aligned} \tag{2.13}$$

in which $\mathbf{r}_t$ is the exponentially moving average of accumulated squared gradients at time t and ρ is a hyperparameter that controls the scale of the exponentially weighted moving average.

2.3.5.7 Adam

Adam or "Adaptive moments" is another adaptive learning rate optimization algorithm proposed by Kingma and Ba [26]. It is a combination of RMSProp and momentum with a few modifications. Adam keeps track of an exponentially weighted moving average of past gradients (first-order moments) like in momentum and also of past squared gradients (second-order moments) like in RMSProp. The first and second order moments are computed by:

$$\begin{aligned} \mathbf{s}_t &= \rho_1 \mathbf{s}_{t-1} + (1-\rho_1)\mathbf{g}_t \ , \\ \mathbf{r}_t &= \rho_2 \mathbf{r}_{t-1} + (1-\rho_2)\mathbf{g}_t^2 \ , \end{aligned} \tag{2.14}$$

in which $\mathbf{s}_t$ and $\mathbf{r}_t$ are first and second order moments at time t respectively.

Since $\mathbf{s}_t$ and $\mathbf{r}_t$ are initialized as $\mathbf{0}$ vectors, they are biased towards zero. In order to avoid this problem, Adam applies the bias corrections to the estimates of the moments:

$$\begin{aligned} \hat{\mathbf{s}_t} &= \frac{\mathbf{s}_t}{1-\rho_1^t} \ , \\ \hat{\mathbf{r}_t} &= \frac{\mathbf{r}_t}{1-\rho_2^t} \ . \end{aligned} \tag{2.15}$$

These corrected moments are then used to update the parameters:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\hat{\mathbf{s}}_t}{\sqrt{\hat{\mathbf{r}}_t} + \epsilon} \ . \tag{2.16}$$

The suggested values for the parameters are: $\rho_1 = 0.9, \rho_2 = 0.999, \eta = 0.001, \epsilon = 10^{-8}$ [26]. Adam is a widely adopted as a robust adaptive learning algorithm. It is empirically proven to work well in practice.

2.3.6 Regularization

One of the core problems in machine learning is how to make the model perform well on both the data it has seen during the training as well as new data that it has not seen before. Thus, the goal of a machine learning model is to minimize the test error instead of the training error. In other words, we want to generalize the machine learning model so that the performance will not degrade on new datasets. According to Goodfellow et al. [14], regularization is any modification to the learning algorithm that reduces the generalization error but not the training error. There are some regularization strategies commonly used in deep learning.

2.3.6.1 L2 parameter regularization

L2 parameter regularization, also known as **weight decay**, **ridge regression** or **Tikhonov regularization**, is a simplest parameter norm penalty. This strategy forces all the parameters closer to the origin by adding an additional term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{w}\|_2^2$ to the objective function.

For simplicity, let's ignore the bias term. Thus, $\boldsymbol{\theta}$ is equivalent to $\mathbf{w}$. The objective function becomes:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y}) \ , \tag{2.17}$$

and the corresponding gradient of the new objective function is:

$$\nabla_{\mathbf{w}}\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha\mathbf{w} + \nabla_{\mathbf{w}}J(\mathbf{w}; \mathbf{X}, \mathbf{y}) \ , \tag{2.18}$$

and the update rule is then written as:

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \epsilon\nabla_{\mathbf{w}}\tilde{J}(\mathbf{w}_t; \mathbf{X}, \mathbf{y}) \ , & (2.19)\\ &= \mathbf{w}_t - \epsilon(\alpha\mathbf{w}_t + \nabla_{\mathbf{w}}J(\mathbf{w}_t; \mathbf{X}, \mathbf{y})) \ , & (2.20)\\ &= (1 - \epsilon\alpha)\mathbf{w}_t - \epsilon\nabla_{\mathbf{w}}J(\mathbf{w}_t; \mathbf{X}, \mathbf{y}) \ . & (2.21) \end{aligned}$$

Equation (2.21) indicates that the weight vector shrinks by a constant factor $1 - \epsilon\alpha$ before performing the regular update.

L2 regularization is proven to have the effect of preserving important dimensions of the parameters that contribute substantially to minimizing the objective function while diminishing the less important dimensions. The effect of L2 regularization is indicated in Figure 2.5. For simplicity, assume our model has two parameters: w_1 and w_2. The point where the contours and the constrained region touch $\mathbf{w}^*$ is the optimal solution for the regularized objective function. The constrained region of the regularization term is a circle. Varying w_2 along its direction will have more effect on loss value than varying w_1. That implies the constrained region is more likely to touch the contours when they extend in the direction of w_2. Hence, L2 regularization tends to give the solution that preserves important directions.

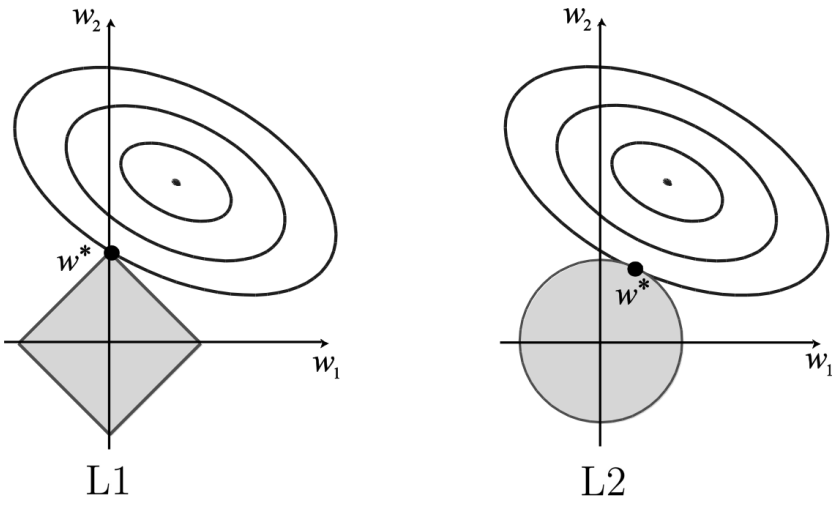


Figure 2.5: Estimation for L1 (left) and L2 (right) regularization. The ellipses are the contours for the value of the original loss function. The solid grey areas are the constraint regions of L1 and L2 norm, respectively. [3]

2.3.6.2 L1 parameter regularization

L1 parameter regularization is another weight decay method that uses L1 norm of the weights as the regularization term instead of L2 norm. With the

[3]https://codeburst.io/what-is-regularization-in-machine-learning-aed5a1c36590

regularization term $\Omega(\boldsymbol{\theta}) = \|\mathbf{w}\|_1 = \sum_i |w_i|$, the objective function becomes:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \|\mathbf{w}\|_1 + J(\mathbf{w}; \mathbf{X}, \mathbf{y}) \ , \tag{2.22}$$

and the corresponding gradient becomes:

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \ \text{sign}(\mathbf{w}) + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}) \ . \tag{2.23}$$

Equation (2.23) indicates that the update does not shrink the weight w_i linearly but instead depends on the sign of w_i. Thus, the effect of L1 is different from L2 regularization. Specifically, L1 regularization encourages a sparse solution, meaning that most of the weights are zero and some are non-zero. This can be explained intuitively in Figure 2.5.

Since L1 regularization has the constraint regions of a diamond shape, it more likely touches the contours at the corners. When the solution occurs at the corners, one parameter w_i is equal to zero. When the model has more parameters, the diamond becomes a rhomboid. Thus, the optimal solution still has many zero values or sparse. Because of the sparsity in the solution of L1 regularization, it has been widely used for feature selection.

2.3.6.3 Data augmentation

One crucial requirement for a machine learning algorithm is to have enough training data. Otherwise, the algorithm will likely suffer from overfitting, a modeling error which occurs when the model fits too well to the training data but performs poorly on unseen data. In real-world scenarios, since collecting and processing data is time-consuming and expensive, data is usually limited. One cheap solution for this problem is to generate a synthetic dataset from a given one. This method is called data augmentation.

There are various ways to augment the data, but the most common transformations are color augmentation (changing brightness, contrast, hue, saturation, etc.), image flipping, random cropping, and noise injection. A combination of any of the approaches above is also an augmentation scheme.

Data augmentation makes a machine learning model invariant to the transformations used and thus generalize better. Data augmentation has proven to be effective especially for classification tasks and speech recognition tasks.

In practice, when doing data augmentation, one must consider which transformations to apply. For example, in the MNIST dataset, if we apply a horizontal flip, number '**6**' becomes '**9**' and vice versa. Hence, the augmented data would confuse the neural network and result in mediocre performance.

2.3.6.4 Early stopping

During the training phase, a common way to recognize overfitting is to look at the training and validation errors. It is often observed that the validation error starts to raise even though the training error keeps on decreasing. Thus, if we continue training the model, the model's parameter setting returned at the last epoch is not optimal because it does not give the lowest validation error. Early stopping is the strategy that utilizes this idea: stop the training after seeing no improvement in the validation error after a specified number of training steps and return the model parameters that achieved the best validation error so far.

Early stopping can be seen as a strategy to tune the model hyperparameters in which the number of training steps or epochs is also a hyperparameter. While most of the hyperparameters are expensive to tune as it requires us to run the actual training process, choosing the "training step" hyperparameter via early stopping is much cheaper. The only considerable cost for this regularization method is the model evaluation on the validation set during training. However, this can be done in parallel with the training process on a separate machine. In general, it will not affect training time significantly. Due to its effectiveness and simplicity, early stopping is the most commonly used regularization method.

2.3.6.5 Dropout

Dropout is another regularization technique that randomly drops out some non-output nodes in the network. Dropping out a node can be understood as temporarily deactivating that node by setting its value to zero.

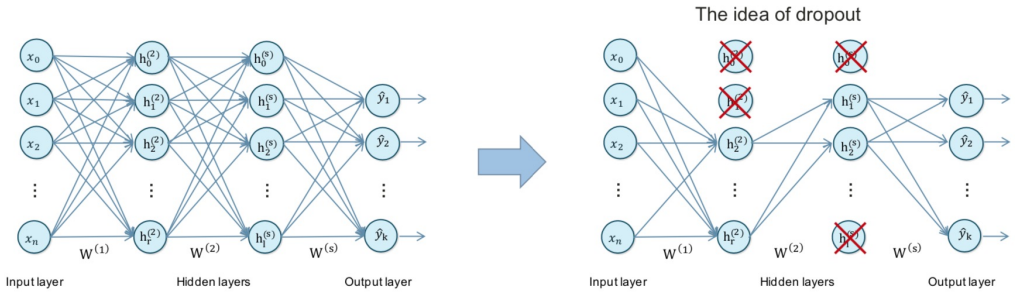


Figure 2.6: An example of dropout [27].

The idea of dropout is illustrated in Figure 2.6 in which the network on the left is just a regular neural network and the network on the right is trained with dropout algorithm. The nodes with a red cross are deactivated.

Dropout is usually trained with a mini-batch algorithm. Specifically, whenever a mini-batch of the input is loaded, a binary mask is randomly sampled for each input and hidden node of the network. These masks are sampled independently based on a predefined sampling probability, typically a value of 0.5.

We can understand that dropout trains an ensemble of an exponentially large number of sub-networks and they share the model parameters. Each network inherits a subset of parameters from the original neural network. It is also worth noticing that dropout does not train all these sub-networks explicitly. The reason is that the original network is normally large, thus it is not sensible to train an extremely large number of sub-networks. In fact, only a part of these sub-networks is trained during a training step. The remaining parts are adjusted by other sub-networks via parameter sharing, resulting in a generalized setting of the parameters. Another way to look at dropout is to see it as a noise injection to the hidden units. In fact, applying masking noise to hidden units contributes substantially to the advantages of dropout.

Dropout is implemented only in the training phase. At test time, the whole network is used. Dropout is widely adopted in the deep learning community because of its efficacy. This regularization method is computationally cheap and can be applied to any model that uses distributed representation and stochastic gradient descent.

2.3.7 Batch normalization

While training a neural network, the distribution of the input in each layer changes since the parameters of the previous layer change. This phenomenon is known as internal covariate shift. This makes the neural network harder and slower to train since it requires lower learning rate and suitable parameter initialization. To better understand the internal covariate shift problem, let's consider a neural cat detector. The detector is trained on a training set of black cats. It is obvious that the network will perform poorly if we test on images of colored cats. The reason behind this behavior is that the distribution of the data in both training and test sets are different. In formal terms, if a network has learned some mapping from input $\mathbf{X}$ to output $\mathbf{Y}$ and the distribution of $\mathbf{X}$ changes, the network needs to relearn that mapping.

Based on the idea that whitening the input, i.e. normalizing input to have zero means and unit variances, will speed up the convergence of a network training, Sergey et al. [23] proposed an algorithm to reduce the internal covariate shift. This can be done by whitening the output of a previous activation layer to fix their distribution before feeding it to the next layer.

Since full-batch whitening is computationally expensive and sometimes non-differentiable, the authors made two simplifications: whitening each scalar feature of the layer's input independently and using mini-batch instead of the whole batch to estimate the mean and variance of each activation.

Consider a mini-batch β of size m and a particular activation x. The batch normalization algorithm is presented in Algorithm 1 in which ϵ is a small constant added for numerical stability, γ and β are two extra learnable parameters.

Algorithm 1 Batch normalization algorithm [23].

Input: Values of $\mathbf{x}$ over a mini-batch: $\beta = \{x_i \dots x_m\}$
Parameters to be learned: γ, β
Output: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$$\mu_\beta \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \triangleright \text{ mini-batch mean}$$

$$\sigma_\beta^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2 \qquad \triangleright \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}} \qquad \triangleright \text{ normalize}$$

$$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \qquad \triangleright \text{ scale and shift}$$

Batch normalization brings some advantages in training a neural network. First, it allows the network to be trained with saturating nonlinearities, making it possible to use a higher learning rate and making it more tolerant to parameter initialization. Furthermore, batch normalization acts as a regularizer in the network, hence, supersedes other regularizers such as dropout. Lastly, batch normalization can significantly speed up the training process without hurting the performance.

2.3.8 Convolutional neural networks

Convolutional neural networks (CNNs) are a special kind of neural network, designed to utilize the spatial information from the input data such as images or time-series data. The name convolutional neural network comes from the fact that the network employs a special math operation, namely "convolution" in at least one of its layers.

The building blocks of a CNN includes a convolution layer, pooling layer, and fully-connected layer. Most of the CNN architectures comprise a combination of these layers together as shown in Figure 2.7.

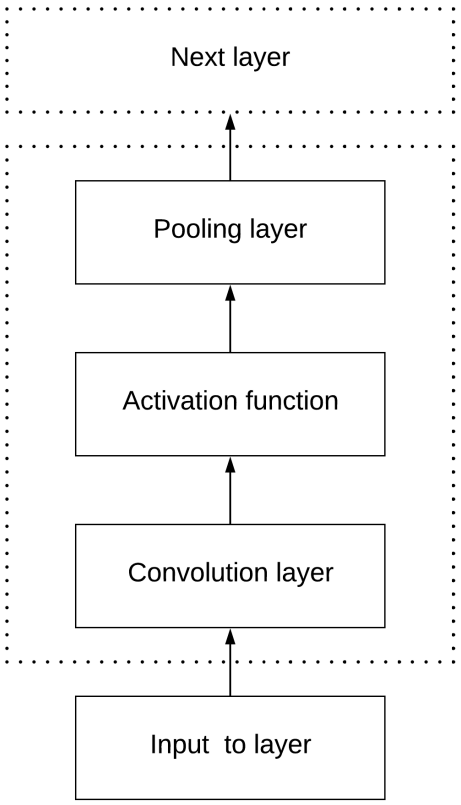


Figure 2.7: Typical building blocks of a CNN architecture [14].

2.3.8.1 Convolution layer

The convolution (or Conv) layer is the core layer in a CNN and does the heaviest computations. In a traditional neural network, a neuron in one layer is fully-connected to all neurons of the previous layers and does not share connections with other neurons in the same layer. In a convolutional neural network, however, each neuron in a layer is connected to only a subregion of the input also known as the receptive field, making a sparsely connected network. The sparsity is defined by the parameter settings of the learnable filters of the convolution operation. Each filter is a tensor of parameters that will learn some specific features from the input. These filters will slide over the whole volume of the input, extract the responses at each spatial location by computing the dot product between the filters and the corresponding input areas. Intuitively, the model will learn the filters that can detect meaningful patterns from the input such as edges, shapes, and even more complex structures at high layers of the network. The filters at each

layer will create a corresponding activation map. These activation maps are then stacked together to produce an output.

There are several types of convolution operations. The distinguishing feature to categorize these operations is the shape of the convolution filter. The four mostly used are 1D, 2D, 3D and 1×1 convolutions.

1D convolution is the operation that uses 1D filters to convolve the input. 1D convolution is normally used for single spatial dimension data such as text, time series, audio signals, etc. This type of convolution is widely used in natural language processing, speech processing, and graph smoothing.

If the convolution is performed along two dimensions of the input with 2D filters, it is then referred to as 2D convolution. 2D convolution is the most commonly used operation in the field of machine learning, especially in computer vision. 2D convolution can extract spatial features such as edges, curves and more abstracted spatial features. If the input is in 3D, the filters will be automatically spread through the whole depth of the input to produce a 2D mapping. If multiple filters are used, their mappings are stacked together.

3D convolution is similar to 1D and 2D convolutions, the only difference is that the kernel is a cube and thus applied along three dimensions of the input volume: width, height, and depth. Unlike 2D convolution, the filter in 3D convolution does not spread the whole depth of the input, instead, the 3D filter will slide along the depth channels to produce a 3D mapping. If multiple filters are used, their 3D mappings are stacked together to create 4D output.

1×1 convolution, also sometimes called "network in network" [34], refers to the operation with the filter of size 1×1. With the input with size $W \times H \times 1$ (depth size of 1), 1×1 convolution does not bring any benefit as what it really does is just multiplying the input with an integer value of the kernel. However, when the depth size is larger than 1, 1×1 convolution has a few advantages. First, 1×1 convolution is good for dimension reduction. Suppose the input image has dimension $W \times H \times D$, if we use N filters of size $1 \times 1 \times D$, the output will have size $W \times H \times N$. In a layer of a very deep network, using multiple filters of size 3×3 or 5×5 can be prohibitively expensive if the previous layer has already used many filters. The 1×1 convolution operation could be applied to reduce dimension depth-wise by using a small value of N ($N < D$) before applying expensive convolution operations with filters of a bigger size. Secondly, using $1 \times 1 \times D$ kernels is computationally cheap since less matrix multiplication is required. Moreover, a non-linear activation function can be applied after dimension reduction by 1×1 convolution, allowing a neural network to add more layers and thus learn a more complex function.

2.3.8.2 Pooling layer

In a typical CNN architecture, it is common to insert a pooling (Pool) layer between the successive convolution layers. The main purpose of this pooling layer is to reduce the number of parameters in the network and to make the representation invariant to small translations of the input. This can be done by reducing the spatial size of the output of one layer before feeding it to the next layer.

The pooling operation replaces a subregion of the output of a Conv layer by a summary statistic of the surrounding pixels. There are several pooling operations; each type is categorized based on the method used for the summary statistic. The most popular is max pooling that takes the max value of the whole subregion as the summary statistic. Other types of pooling include the average of the subregion, the L2-norm of the subregion and weighted average based on the distance from the centering pixel. Pooling operation slides a window, normally of size 2×2, along the width and height dimensions. However, pooling is applied depth-wise, meaning it operates independently on every depth slice of the input. Thus, pooling resizes the input spatially while the depth dimension remains unchanged.

Besides reducing the number of parameters to improve the computational efficiency, the pooling layer is added to make the function learned by the network become invariant to small translations. For example, face detecting should be able to detect faces even though the face's features such as eyes, nose, mouth are not exactly the same for different people. This invariant improves the generalization of the network.

Pooling is also useful for handling various input sizes. For example, a CNN classifier wants to classify images of different sizes, the input size for the classification layer must be fixed. Varying the offset size between pooling regions can accomplish this requirement.

Even though pooling is very popular in the pipeline of a CNN, some recent architectures, such as the one proposed by Springenberg et al. [53], are removing the pooling layer out of a typical pipeline. These architectures use a larger stride in the Conv layer to reduce the size of the representation. [53] proves that dropping pooling layers allows some SOTA neural networks [15, 31, 34, 58] to achieve competitive results.

2.3.8.3 Fully-connected layer

Fully-connected (FC) layers are just like layers in regular neural networks in which every neuron in one layer is connected to all other neurons in the previous layer. The FC layers are normally added after the last convolution

and pooling layers. Intuitively, the FC layers look at the high-level features that are strongly correlated to the output. In case of a classifier, the last FC layer will output the probabilities that the input belongs to each class.

2.4 Multi-task learning

Multi-task learning (MTL) is an area in machine learning that aims to solve multiple tasks simultaneously by utilizing the similarity between these tasks. Based on an assumption that the tasks are related, MTL leverages the useful information contained in the tasks to regularize the learning model. MTL has empirically and theoretically proven that jointly learning multiple tasks enabled the model to perform better than learning them separately [50]. Depending on the nature of the tasks, there are several MTL settings: multi-task supervised learning, multi-task unsupervised learning, multi-task semi-supervised learning, multi-task active learning, multi-task reinforcement learning, and multi-task online learning. In this work, we use a multi-task supervised learning setting in which each task is a supervised learning task.

In the context of deep learning, there are two common prevalent techniques used for multi-task learning: hard parameter sharing and soft parameter sharing.

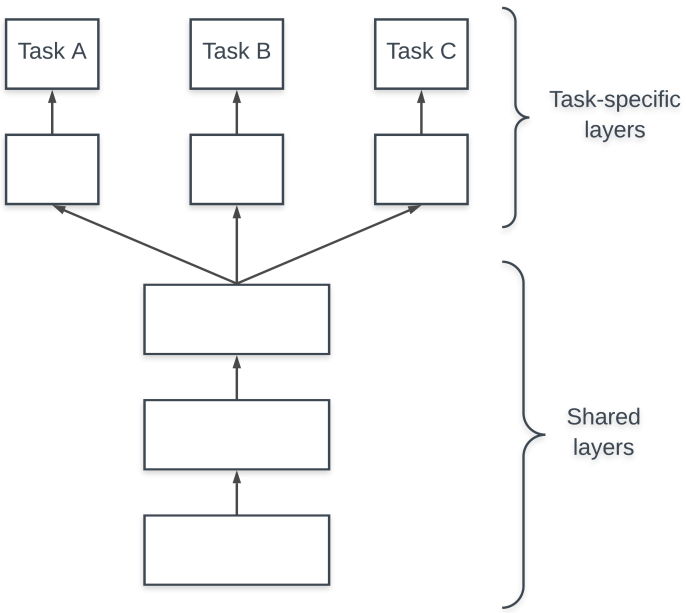


Figure 2.8: Hard parameter sharing for multi-task learning [50].

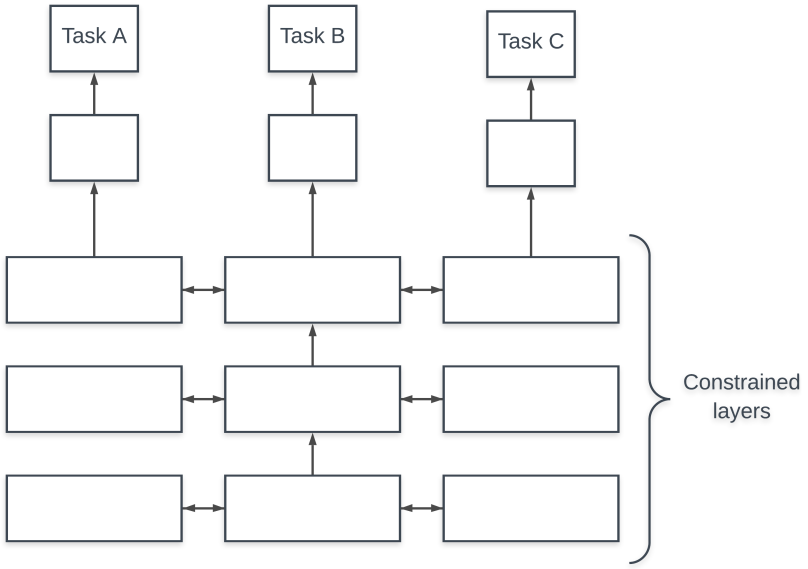


Figure 2.9: Soft parameter sharing for multi-task learning [50].

2.4.1 Hard parameter sharing

Hard parameter sharing [5] is the most popular technique for MTL in neural networks. The architecture can be generally divided into two parts with corresponding model parameters as shown in Figure 2.8: shared layers: these are typically the lower hidden layers of the neural network that are shared across all the tasks and task-specific layers: these are typically the upper output layers of the neural network and are only associated with only one task. Since the model learns a shared representation for various tasks, the shared part of the model is forced to learn good values, resulting in a more generalized model and thus can reduce the risk of overfitting.

2.4.2 Soft parameter sharing

In contrast to the hard parameter sharing approach, soft parameter sharing approach requires a separate model for each task [11, 60]. However, the parameters of these models are regularized to have a small distance. The distance regularization encourages the models to have similar parameters that can represent all tasks. The architecture is shown in Figure 2.9.

The motivation behind MTL is to mimic how a human learns to perform a particular task. We, as humans, often learn one task by transferring the knowledge we already have from other tasks. For example, a baby first learns

how to stand and balance, then how to walk and finally how to run. It is also biologically inspired that learning multiple tasks simultaneously improve the learning process. A person learning to play guitar and piano at the same time will likely take less time to master both instruments in comparison to the total time required for him to learn two instruments separately from the beginning.

From a machine learning perspective, MTL is used for several plausible reasons. First, MTL implicitly acts as a regularization method. Since each task has a different noise pattern, jointly learning multiple tasks encourages the model to learn a general representation that ignores data-dependent noise. Secondly, MTL makes the model robust to feature selection. If the data is noisy and has many dimensions, choosing the relevant features can be hard for single-task learning. However, in MTL settings, learning the features that work for various tasks is easier since the model is more constrained towards relevant features. Moreover, MTL allows learning complex features through eavesdropping. Some features are difficult to learn for a particular task A, but are easy for task B. MTL enables the model to learn those difficult features of task A through task B. Finally, MTL makes the model to favor the representation that benefits most of the tasks.

Chapter 3

Hyperspectral forest data

This section describes the data used in this work and the essential preprocessing steps to make the data compatible with the learning algorithm. All the forest data is collected from various sources and processed by the AIROBEST team. AIROBEST is a joint research project between the Land Remote Sensing group of VTT Technical Research Centre of Finland and the Department of Computer Science at Aalto University, funded by the Academy of Finland in 2018-2021.

3.1 Data description

Figure 3.1 indicates the boreal zones whose forest data will be used in AIROBEST experiments. In this work, only the data of zone A is used for training the deep learning models.

In general, the training data includes a hyperspectral image and corresponding forest labels. The hyperspectral image is an airborne image of a forest region of southern Finland, near Hyytiälä forestry field station. The image has a resolution of 12143×12826 pixels and 128 spectral bands or color channels. Figure 3.2 shows the hyperspectral image in RGB format.

The forest labels are harvested from forestry database of Finnish forest structural variables. There are 16 variables in total, in which four are discrete while the rest are continuous variables. Some of the variables are synthesized based on other variables. Table 3.1 shows a list of all 16 variables that are being modeled in this work. The last column *No. classes* is only applicable for discrete variables and indicates the number of classes of the corresponding variable before and after removing extreme minority classes.

In forest sciences, a forest is represented by a collection of *stands*. A stand is a small area in the forest where trees share distinct characteristics, such as

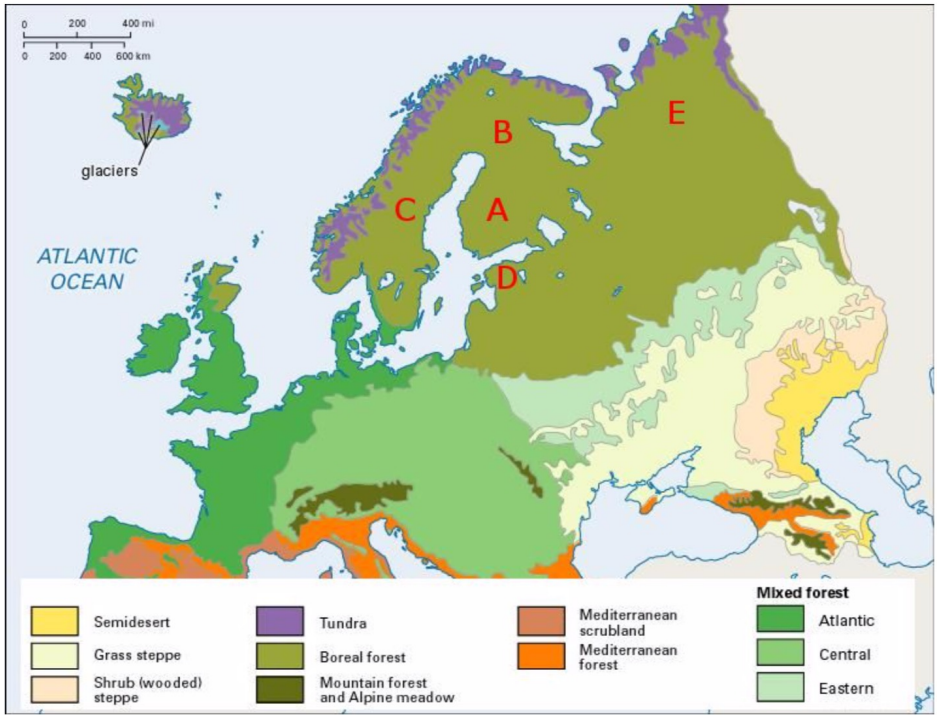


Figure 3.1: The geographical locations for data used by AIROBEST in the European boreal zone. A: Hyytiälä, Finland, B: Sodankylä, Finland, C: Swedish National Forest Inventory data, D: Estonian state-owned forest database, E: Pechora-Ilych nature reserve, Komi republic, Russia. [4]

age, size, species, biomass. Initially, the forest data were collected at stand level. Since we use CNNs to model the forest variables in this work, we need to have forest labels at pixel level. Thus, to generate the forest labels for each pixel in the hyperspectral image, we duplicated the labels for all pixels in the same stand.

3.2 Processing

Since the data contains some synthesized variables, it is quite noisy and needs to be preprocessed before training the deep learning model. For the four classification or discrete variables, the distribution of the classes is extremely rare. The original class distributions of these variables are shown in Figure 3.3.

As we can see from the class distribution, each variable has some major

[4] britannica.com

Figure 3.2: Image of the forest in RGB format.

classes and minor classes. There are some minor classes that only account for less than 0.1% of the whole dataset. This leads to severe performance degradation of a deep learning model. This class imbalance problem makes the model favor the major classes and ignore the minor classes. To have a fair evaluation of the model, we remove all the classes that have very little (less than 5%) data. This operation aims to remove the extremely skewed classes but still tries to keep the imbalance nature in the dataset. Figure 3.4 illustrates the class distributions after removing all the extremely skewed classes.

Besides the classification variables, there are twelve continuous or regression variables. Since the value range of each variable is different, the regression values are normalized to the 0-1 range using min-max scaling. Regression labels are also noisy. Some pixels from the hyperspectral image have extremely high regression values. Thus, when normalizing these values, labels of most of the pixels are pushed towards zero. Consequently, the model

Name	Discrete	Continuous	No. classes
Fertility class	✓		7 / 4
Soil type	✓		10/ 4
Development class	✓		9 / 4
Main tree species	✓		3 / 3
Basal area		✓	N/A
Mean age		✓	N/A
Stem count		✓	N/A
Mean height		✓	N/A
Percentage of main tree species		✓	N/A
Percentage of pine		✓	N/A
Percentage of broadleaf		✓	N/A
Woody biomass		✓	N/A
Leaf area index		✓	N/A
Effective leaf area index		✓	N/A
Diameter at breast height		✓	N/A

Table 3.1: Predictive variables in AIROBEST's HSI dataset.

performs poorly since all it learns is to predict zero values. To avoid this problem, we choose a thresholding value at 98^{th} percentile of the label values and clip larger values to this threshold before normalizing.

Figures 3.5 and 3.6 illustrate the labels of one discrete variable and one continuous variable respectively. In both figures, *stands* are shown as regions bounded by black border and target labels are color-coded. All pixels in a stand have the same classification and regression values, thus having the same color. Figure 3.5 shows class labels of the main tree species. Since the main-tree-species variable has three classes, namely *pine*, *birch*, and *spruce*, three colors are used to discriminate between the classes. Similarly, Figure 3.6 visualizes the normalized regression labels of the mean height. The color bar on the right shows the value range of the labels. The brighter color is, the higher value the pixel has.

To train a regular CNN, it is common to feed the network with a batch of images from the training set. This is reasonable when the training set contains many image samples. However, in this work, the data we have is a single high-resolution hyperspectral remote sensing image. To have enough training samples, we need to split the original hyperspectral image into much smaller patches, typically of size from $5 \times 5 \times B$ to $27 \times 27 \times B$ in which B is the number of spectral bands. As the hyperspectral image is typically taken from an airplane or satellite, it contains a large geographical area of the

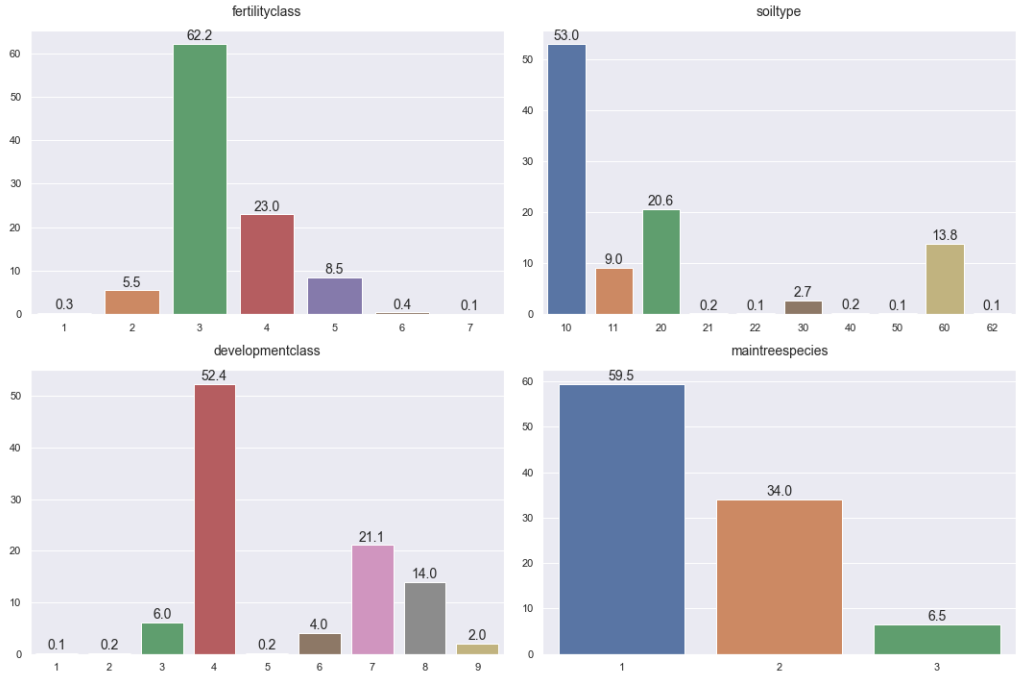


Figure 3.3: Original class distributions of the four classification variables.

forest, including non-forest areas such as lakes, roads, residential areas, etc. These non-forest areas will add noise to the model and degrade the model's performance. To mitigate this problem, when selecting the patches, we ignore the patches that have non-forest pixels. Those patches are then randomly shuffled and are divided into train (72%), validation (8%), and test (20%) sets. We also fix the random seed before shuffling the data to make sure the sets contain the same patches in every training run, assuming the patch size does not change. The size of the train, validation and test sets varies depending on the patch size. In our experiments, we use non-overlapping patches of size $27 \times 27 \times B$, as training samples. The labels of the sample are the labels of the centering pixel. During training, the train set is augmented to created more training samples. The statistics of the train, validation and test sets with and without augmentation are shown in Figure 3.2.

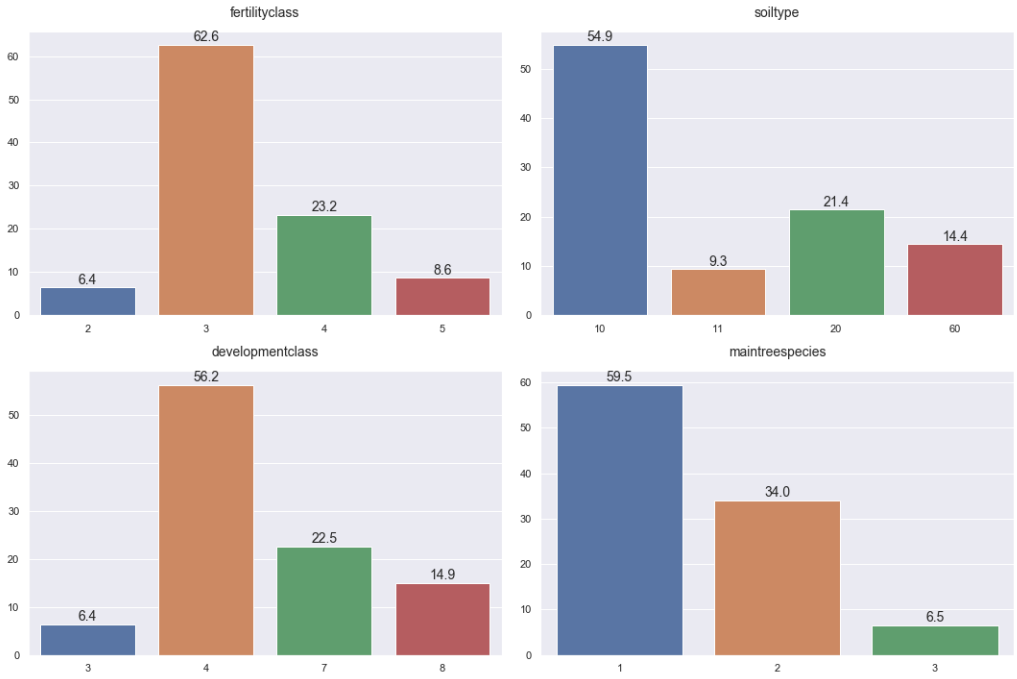


Figure 3.4: Final class distributions of the four classification variables.

Set	**Number of samples**	
	w/o augmentation	**w/ augmentation**
Train	17594	42832
Validation	1955	4759
Test	4888	4888

Table 3.2: Number of data samples of train, validation and test sets with a patch size of 27.

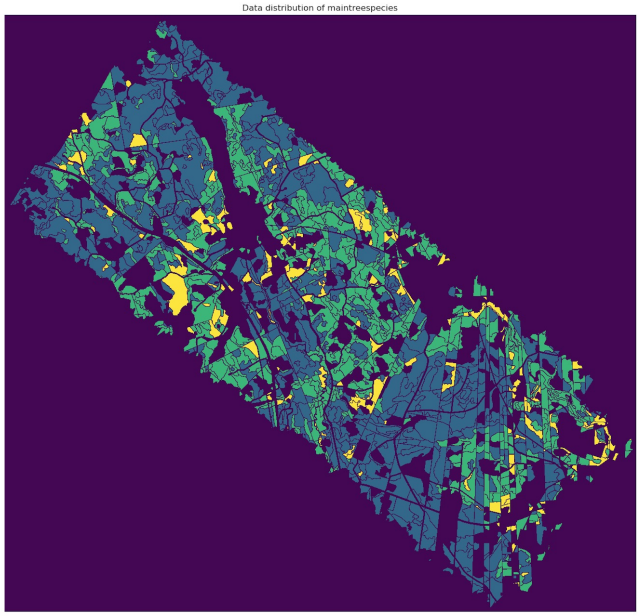


Figure 3.5: Classification labels of the ***main tree species****.*

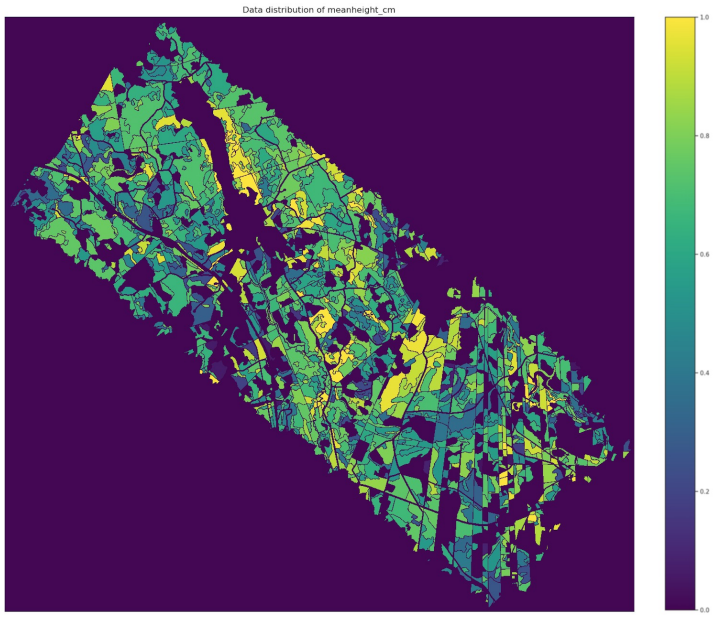


Figure 3.6: Normalized regression labels of the ***mean height****.*

Chapter 4

Evaluation metrics

Choosing the right evaluation metric for training a machine learning model heavily affects the performance of the model. Depending on the requirements of a task, optimizing the wrong metric may result in a useless model. This chapter discusses the relevant metrics to evaluate our deep learning model, both on regression tasks and classification tasks.

There are several metrics to evaluate the performance of a regression model. The most popular ones are mean squared error, root mean squared error and mean absolute error. Similarly, we have several metrics for classification tasks, including accuracy, precision and recall, area under the receiver operating characteristic curve. We will briefly introduce these metrics and describe what they measure.

4.1 Mean squared error

Mean squared error (MSE) is considered the simplest and most commonly used metric for regression evaluation. Let N be the number of data points, y_i is the target label and $\hat{y}_i$ is the predicted label. The MSE error is defined as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \ . \tag{4.1}$$

MSE estimates the squared difference between the true and predicted labels. MSE incorporates both variance and bias of the model, hence, it is also called the second moment of the error. The mathematical formulation indicates that MSE is a non-negative quantity. When the value of MSE is zero, the model perfectly predicted the labels for all data points. This

situation is practically impossible since the collected data always has a certain level of noise. If the value of MSE is equal to the variance of the model, we say the model is unbiased. Among the unbiased models, the one with the lowest variance is considered the best one.

Since MSE penalizes the squared difference, it is very sensitive to the outliers. If the model badly predicts some labels, the MSE can be quite large and it is hard to evaluate the model's goodness. This problem is more severe if the data is noisy. Moreover, if the target and predicted labels have very small values, the MSE metric may underestimate the model's badness.

4.2 Root mean squared error

Root mean squared error (RMSE) is simply the square root of the MSE. The RMSE measures the magnitudes of the errors:

$$RMSE = \sqrt{MSE}\,. \tag{4.2}$$

RMSE is one of the most popular evaluation metrics as it is interpretable in the same unit as the quantity being measured, making it easier to construe the information. Thus, it is a better metric to evaluate the goodness of the model in comparison to the MSE. However, while training a machine learning model, MSE is easier to work with since it is easier and faster to compute the gradients.

4.3 Mean absolute error

Mean absolute error (MAE) calculates the average magnitudes of the errors between the target and predicted values. The following formula defines the MAE:

$$MAE = \frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{y}_i|\,. \tag{4.3}$$

MAE weights the individual error equally, thus it does not penalize the outliers as harshly as MSE. Similar to RMSE, MAE is also easy to interpret and correlate the information.

Due to the absolute function, MAE is not always differentiable. The gradient of MAE is undefined when the value of MAE is zero. This condition only happens when the model perfectly predicts all the labels, which is unlikely to happen. This problem can be solved by setting the derivative to zero for this phenomenon.

4.4 Accuracy, precision and recall

One of the most commonly used metrics for classification is prediction accuracy. For binary classification, accuracy is simply computed as the percentage of correct predictions over the whole dataset. For multi-class cases, it is the average of accuracy among all classes. However, accuracy is sometimes misleading and not informative enough to evaluate a model. Precision and recall are two additional crucial performance metrics, especially for information retrieval and binary classification. In order to understand precision and recall, one needs to grasp the definition of a confusion matrix.

A confusion matrix is a table that summarizes the performance of a classifier on a labeled dataset. This matrix displays the count of correct and incorrect predictions for each class. Without loss of generality, consider a binary classifier with two classes: 0 (negative) and 1 (positive). Figure 4.1 illustrates an example of a confusion matrix M. For class 0, 39 samples are correctly predicted (true negatives or TN) and 14 samples are incorrectly predicted (false negatives or FN). For class 1, on the other hand, 7 samples are incorrectly predicted (false positives or FP) and 34 samples are correctly predicted (true positives or TP).

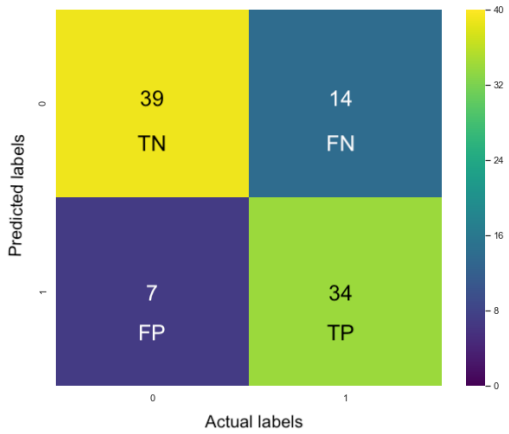


Figure 4.1: An example of confusion matrix for binary classifier.

To avoid confusion, the terms "positive" and "negative" regard the two classes, while "true" and "false" regard the correctness of the classification prediction. In multi-class settings, for each class, we consider it as positive, the rest as negative.

Precision is defined as the fraction of true positives over the sum of true

positive and false positives. In mathematical form, precision P is written as:

$$P = \frac{TP}{TP + FP} \,. \tag{4.4}$$

Precision answers the question: "Among all samples predicted as positive, how often are they correct?". Precision is usually used to minimize the number of false positives.

Meanwhile, recall tries to answer the question: "Among all positive samples, how often are they correctly predicted?". Recall aims to reduce the number of false negatives and is defined as:

$$R = \frac{TP}{TP + FN} \,. \tag{4.5}$$

4.5 Balanced accuracy

Even though accuracy, precision, and recall are reliable metrics to evaluate the performance of a model, they only work for a balanced dataset. If the training dataset is imbalanced meaning that a large portion of the training data belongs to only a few classes (majority classes) and very little data belongs to other classes (minority classes), accuracy is no longer a suitable metric. The reason is that a naive classifier that completely ignores the minority classes and always predicts all the test samples as majority classes can easily achieve high accuracy. If the purpose of the task is to distinguish minority classes such as in rare event detection task, optimizing the overall accuracy metric is not helpful. For this reason, other metrics that balance between multiple classes are more desirable. A sensitivity measure, namely class-balanced accuracy or simply balanced accuracy, addresses this problem by considering the class distributions more importantly.

For a single-label case, we compute the accuracy R_i for each class i in a total of c classes. From the confusion matrix M, R_i is quantified as:

$$R_i = \frac{M_{i,i}}{\sum_{j=1}^{c} M_{j,i}} \,, \tag{4.6}$$

in which $M_{j,i}$ is the number of samples belonging to class i that were predicted as class j. The balanced accuracy is the average of all the class accuracies R_i:

$$BA = \frac{1}{c} \sum_{i=1}^{c} R_i \,. \tag{4.7}$$

For multi-label classification, the final accuracy is the mean of the class-balanced accuracies across all labels.

4.6 ROC curve and AUC

Besides balanced accuracy, another metric to measure the performance of a classifier for an unbalanced dataset is the receiver operating characteristics (ROC) curve. ROC curve is a plot illustrating the relationship between the true positive rate (TPR) and false positive rate (FPR).

TPR is just another name of recall or sensitivity. Meanwhile, FPR demonstrates how often the model incorrectly predicts a negative sample as a positive one. FPR is defined as follows:

$$FPR = \frac{FP}{TN + FP} \, . \tag{4.8}$$

The ROC curve gives us a visual analysis of the model's performance. The ROC curve can be represented by a single scalar value by computing the *area under curve* (AUC). The ROC curve or AUC tells us how well the classifier can distinguish between classes. A higher value of AUC means better separability. An ideal classifier has an AUC of one. The worst model has an AUC of zero, meaning that it cannot predict any sample correctly. We can compare the performance of two classifiers by looking at their AUC values. Figure 4.2 shows an example of the ROC curve.

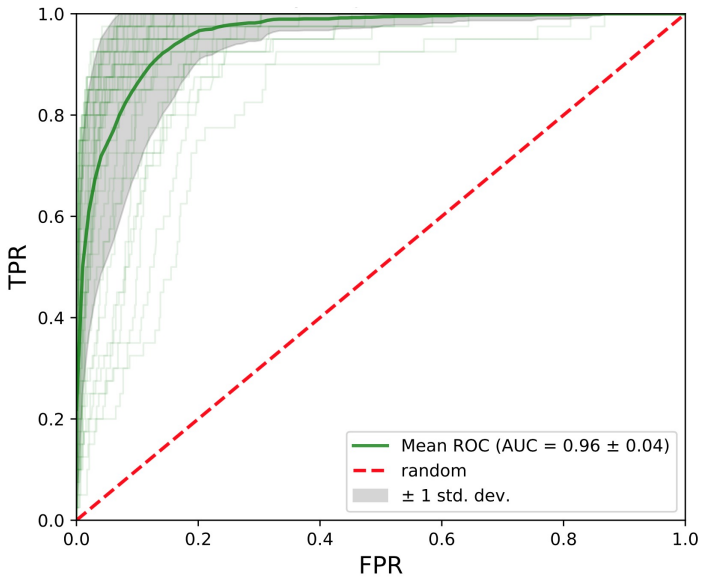


Figure 4.2: An example of ROC curve.[4]

[4]https://docs.eyesopen.com/toolkits/cookbook/python/plotting/fprocs.html

Due to the imbalanced nature of our dataset, regular accuracy, precision, and recall are not suitable metrics for the classification tasks. Instead, we adopt balanced accuracy and AUC to evaluate the performance of our model. For our regression tasks, we use MSE as the loss function and MAE as a performance metric.

Chapter 5

Baseline model

In this chapter, we will discuss the baseline of the CNN models and examine in detail the model architectures experimented in this thesis work.

5.1 Pipeline

Chen et al. [6] experimented with various CNN architectures and found that 3D CNN exploited the benefits of both 1D CNN and 2D CNN. They illustrated that while 1D CNN extracted spectral features and 2D CNN extracted local spatial features, 3D CNN could simultaneously utilize both spatial and spectral features from the hyperspectral image. These learned features later contributed substantially to the classification stage of the network. Inspired by the success of their models, we developed our baseline 3D CNN model with some modifications in the feature extraction part and added the task-specific layers for our multi-task learning problem.

The pipeline of the experimented architectures is illustrated in Figure 5.1. We adopted the hard parameter sharing approach for our multi-task learning problem. The pipeline includes two parts: shared layers and task-specific layers. The shared layers are Conv layers and Pool layers stacked together. There are also FC layers attached on top of these layers. The task-specific layers are several FC layers stacked on top of each other. These layers learn the mappings between the extracted features and the output for each individual task.

5.2 Baseline model

The baseline architecture for modeling forest parameters follows the pipeline shown in Figure 5.2. The shared part consists of three Conv layers, inter-

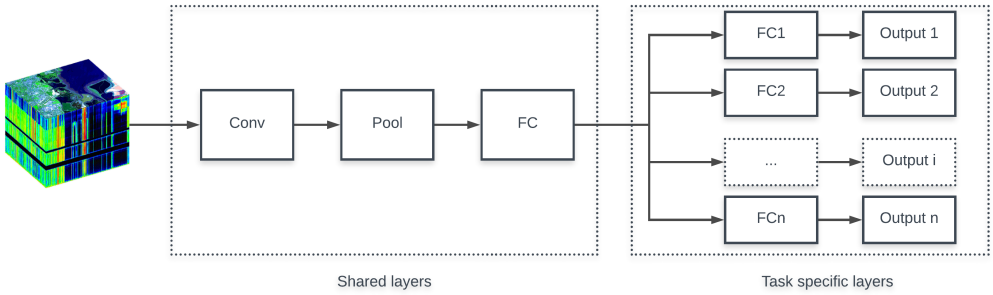


Figure 5.1: The pipeline of the 3D CNNs.

leaved with two Pool layers, followed by a shared FC layer. At each Conv layer, we choose a $3 \times 3 \times 32$ kernel with a stride of one and padding of zero for the 3D convolution operation. For the pooling operation, a kernel of size $2 \times 2 \times 1$ is used. The stride is the same as the kernel size and the padding is zero. After each pooling layer, we apply a non-linear activation function, which is ReLu in this case. The output from the third pooling layer is then flattened and fed in the shared FC layer. The FC layer outputs a feature vector with a size of 512. The non-shared part includes two FC layers for each individual task. The first FC layer takes a 512-length feature vector from the share FC layer and transforms it into a 200-length feature vector. If the task is a classification task with C classes, this vector is mapped to a vector of length C in the last FC layer and then fed into the Softmax function to compute the probabilities for all the classes. Otherwise, if the task is a regression task, the last FC layer is used to predict the output directly. To predict the labels for a pixel in the hyperspectral image, we select a patch of surrounding pixels as an input image to the CNN. The patch has a size of $K \times K \times B$, in which K is the patch size and B is the number of bands of the original hyperspectral image. Depending on the CNN architecture, K and B should be larger than some values. For the baseline architecture, K should be larger than 24 and B should be larger than 94. The baseline architecture is presented in Figure 5.2.

Regarding the loss function, CE loss (presented in Section 2.3.3.4) is used for classification tasks and MSE loss (presented in Section 2.3.3.1) is used for regression tasks.

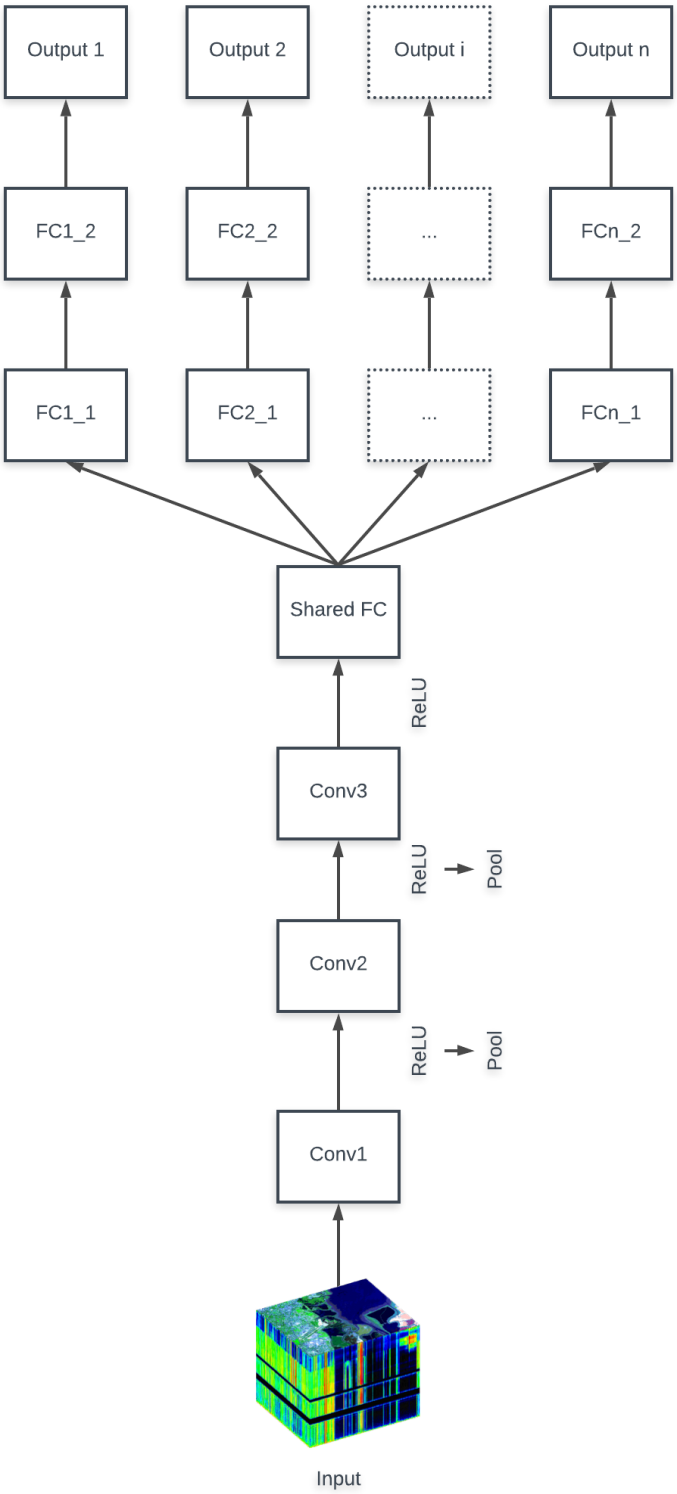


Figure 5.2: The baseline architecture for modeling forest parameters from hyperspectral images.

Chapter 6

Deriving CNN model

Since a hyperspectral image often has many spectral bands (up to a few hundred bands), it contains rich spectral information. Thus, extracting good feature vectors that capture most of the useful information will certainly accelerate learning and improve the performance of the model. Feature vectors in a dog classification task, for example, should encapsulate spatial information such as ears, claws, eyes, and nose. In the case of a hyperspectral image, the feature vectors should capture both spatial and spectral information in order to fully utilize the advantages of it.

In this chapter, we describe the experiments to improve the baseline model by focusing on the feature extraction part. We developed several variants of the baseline architecture. The main difference between these variants lies in the shared layers. To be more specific, we developed new models by adding bells and whistles to the baseline model, for example, stacking more Conv layers, changing the configurations of 3D convolution and pooling operations, using different regularization techniques, etc. in the shared layers.

6.1 Regularization

A deep learning model, or a machine learning model in general, requires a lot of data for the training phase. This is due to the fact the model often has a lot of parameters to learn. Since the amount of training data is limited, especially for supervised learning, overfitting becomes a common problem for deep learning approaches. The problem is even worse for models trained on hyperspectral data in which the training set size is small.

To overcome this problem, we combine several regularization techniques. The first technique is the L2 regularization. As introduced in Section 2.3.6.1, L2 regularization penalizes the extreme parameter values, forcing the model

parameters to be small and close to the origin. Having large parameters makes the model unstable and sensitive to specific examples, statistical noise or training dataset. This kind of model has large variance and small bias. It means when there is a small change in the input, the output can be very different. Thus, having small parameters gives a more generalized model. In our experiment, we tune the weight decay term α in equation (2.17) using grid search. The best value of weight decay α is recorded at 10^{-4}.

The second regularization technique we adopted is dropout. Dropout approximates training a large number of subnetworks in parallel to prevent co-adaptations between network layers. We apply dropout after the first two pooling layers and the third convolution layer.

In addition to using dropout, we also experimented the effect of batch normalization. As presented in Section 2.3.7, batch normalization makes the model less sensitive to the learning rate and parameter initialization. Batch normalization is used after each convolution layer to mitigate internal covariance shift.

The last regularization technique we use is data augmentation. Since overfitting is caused by lacking training data, augmenting the training data would be a natural solution. For a deep learning model that takes an RGB image as the input, such as object detector, pose estimator, image captioning system, etc., tricks like changing the image color, brightness, or creating random crops from the original image would not change the prediction results as long as the important information is retained. Originating from similar ideas, Chen et al. [6] and Lee et al. [32] proposed some data augmentation approaches for hyperspectral input. In this work, we adopt some of those approaches.

The first one is a radiation-based method. This method was derived from the fact that the hyperspectral image captures a large area, whereas objects from the same class can have different radiation at different locations. The radiation-based approach creates new samples $\mathbf{y}$ by multiplying the true sample $\mathbf{x}$ with a random factor α and then adding random noise to it:

$$\mathbf{y} = \alpha\mathbf{x} + \beta\mathbf{n} \,, \tag{6.1}$$

in which β is the weight of Gaussian noise $\mathbf{n}$.

The second method proposed by Chen et al. [6] is mixture-based virtual samples. Inspired by mixture, a phenomenon in remote sensing created due to the long distance between the object and the sensor, this mixture-based approach generates new samples $\mathbf{y}$ by taking the weighted arithmetic mean of two real samples $\mathbf{x}_i$ and $\mathbf{x}_j$ and then adding random Gaussian noise to the

Layers	Conv1	Conv2	Conv3
Kernels	$3 \times 3 \times 32$	$3 \times 3 \times 32$	$3 \times 3 \times 32$
	$4 \times 4 \times 32$	$4 \times 4 \times 32$	$4 \times 4 \times 32$
	$5 \times 5 \times 32$	$3 \times 3 \times 32$	$4 \times 4 \times 32$
	$3 \times 3 \times 54$	$3 \times 3 \times 32$	$3 \times 3 \times 32$

Table 6.1: Size of kernels for convolution operations

term:

$$\mathbf{y} = \frac{\alpha_i \mathbf{x}_i + \alpha_j \mathbf{x}_j}{\alpha_i + \alpha_j} + \beta \mathbf{n} \ , \tag{6.2}$$

in which α_i and α_j are randomly chosen, β is the weight for the Gaussian noise.

Another simple augmentation method is flipping, used in Lee's model [32]. The idea is to mirror the true samples across horizontal, vertical, and diagonal axes. In our work, we only use two axes: horizontal and vertical.

6.2 Fine-tuning model architecture

Fine-tuning a machine learning model is very computationally expensive and time-consuming, especially in the case of deep learning since the training time can take hours or days. In this section, we present our model architecture and hyperparameters of the algorithms used in this work.

Our baseline model uses three $3 \times 3 \times 32$ kernels for the convolution layers. The size of the kernels affects the size of the input patch and the number of parameters a model must learn. We have tested different combinations of the convolution kernels of size $4 \times 4 \times B$ and $5 \times 5 \times B$ in which B is the depth of the kernels. The combinations are shown in Table 6.1. We empirically found that the best combination is $3 \times 3 \times 54$, $3 \times 3 \times 32$ and $3 \times 3 \times 32$ for three convolution layers in our baseline model.

In addition to the kernel sizes, the number of kernels at each convolution layers is also important in learning image features. The more kernels used, the more features the model can learn. However, increasing the number of kernels also means having more parameters to train, thus requiring more training time and increasing the risk of overfitting. Our best model uses 128 kernels for the first, 64 for the second, and 32 for the third convolution layer.

6.3 Multi-scale convolution block

Multi-scale convolution has been used and has proven its effectiveness in various computer vision tasks [13, 19, 32, 33]. To adopt this idea in our model, we use a multi-scale convolution block that comprises four convolution layers with the kernel size of $1 \times 1 \times 1$, $1 \times 1 \times 3$, $1 \times 1 \times 5$, $1 \times 1 \times 11$ respectively. These convolution filters are used to exploit the spectral correlations. No pooling is applied after these layers. The outputs of these convolution layers are then summed together.

We also experimented with larger kernels ($3 \times 3 \times B$, $5 \times 5 \times B$), but did not get any performance gain. Larger kernels exploit the spatial features but also increase the number of trainable parameters. Moreover, to fully utilize the locally extracted spectral features, stacking the outputs of the four convolution layers would make sense. However, stacking them together substantially increases the size of the network without any performance benefit. Instead, we sum the outputs together.

6.4 Class imbalance

Class imbalanced data, in which the distributions across multiple classes are significantly skewed, is a typical problem in machine learning classification tasks. The classes with a substantially high number of data samples are called majority classes, while the classes with much fewer samples are called minority classes. As a machine learning algorithm relies on the dataset it is trained on, the resulting model tends to be biased toward the majority classes. In other words, the model focuses more on accurately predicting majority classes while ignoring the minor classes. This inductive bias can lead to poor performance since minority classes are often what we are interested in and might contain primal information.

While dealing with imbalanced data, it is also imperative to choose the correct metrics and adjust the learning goals. For example, suppose we want to train a model to diagnose cancer from pathology images. The training set contains 95% of the images that belong to the non-cancer class and only 5% of the images belong to the cancer class. If the model learns to always predict the input image as non-cancer, the accuracy of the model is 95% which is generally acceptable. However, this accuracy metric is meaningless since the goal is to predict cancer images. Instead, a metric that measures the true positive rate and false positive rate such as ROC (Section 4.6) is a better metric to optimize. This kind of problem appears in many machine learning and data mining applications, especially in rare event discovery such

as credit card fraud detection, disease diagnosis, defect detection, etc.

In this section, we discuss our attempts to tackle this data imbalance problem. A simple approach is to balance the training data either by oversampling the minority classes or undersampling the majority classes. Oversampling means to duplicate the instances in minority classes and undersampling means to remove instances from majority classes until the dataset has balanced distribution. This sampling method, however, does not work well in practice. For a general single-label classification, oversampling leads to model overfitting by seeing rare instances too often and undersampling degrades the performance as it throws away important information from removed instances. In our multi-label multi-class classification, sampling does not help because a sample can belong to a minority class of a label but majority class of another label. Thus, trying to re-sample this instance to mitigate the problem for one label can worsen the problem for another label. For this reason, we did not use sampling in our experiments. Instead, we adopt three other methods including cost-sensitive learning, class rectification loss, and focal loss.

6.4.1 Cost-sensitive learning

In a regular learning algorithm, we often penalize the misclassifications equally. This can cause severe performance deterioration in imbalanced classification problems as the algorithm is biased towards majority classes. Cost-sensitive learning tries to avoid this problem by considering the misclassification costs.

Cost-sensitive learning was published mostly in [12] and is still an active research area in machine learning [36, 47, 55, 59]. The motivation of cost-sensitive learning came from the observation that the difference in the cost of misclassification errors can be quite large. Once again, consider the case of cancer detection from pathology images as an example. Misclassifying a cancer patient as non-cancer (false negative) is much more serious than misclassifying a non-cancer patient as cancer (false positive). The false negative prediction can affect the patient's life because it delays the treatment. Thus, the false negative prediction error should be weighted much more than the false positive one.

The idea of cost-sensitive learning is to use a cost matrix to specify the costs for false positive (FP), false negative (FN), true positive (TP) and true negative (TN) predictions. Without loss of generality, consider a binary classification problem. An example of the cost matrix is shown in Table 6.2. The entry $C(i, j)$ represents the cost of classifying a sample as an instance of class i while its actual class is j. The value of $C(i, j)$ can be given by experts or learned from the data.

	Actual negative	Actual positive
Predict negative	$C(0,0)$ or TN	$C(0,1)$ or FN
Predict positive	$C(1,0)$ or FP	$C(1,1)$ or TP

Table 6.2: Cost matrix for binary classification

The cost matrix allows the model to penalize the dangerous prediction errors more heavily than the innocuous ones. In general, we can assign bigger weights to the prediction errors of the minority classes than we do with the majority classes.

In the context of machine learning, it is common to have the cost equal to the inverse class frequency in the dataset. Let p_i be the frequency of class i in the dataset. The cost for the wrong prediction for class i is computed as:

$$c_i = \frac{1}{p_i} \, . \tag{6.3}$$

This scheme does not work well in the case of an extremely imbalanced dataset. Some extreme minority classes can be assigned very large weights, making the model focus too much on these classes. In our work, in addition to this weighting scheme, we also experimented with inverse median frequency and found it performed better on our dataset. Let p_m is the median of class frequencies, then the weight for class i is:

$$c_i = \frac{p_m}{p_i} \, . \tag{6.4}$$

6.4.2 Class rectification loss

Most of the existing methods tackle class imbalance at a moderate scale. Re-sampling and cost-sensitive learning work to a certain extent, but are not suitable for an extreme imbalanced dataset. Class rectification loss (CRL) is a batch-wise optimization approach proposed by Dong et al. [9] to address the large-scale class imbalance problem for multi-label classification. CRL tries to rectify the learning bias due to the imbalanced training data by leveraging minority class hard sample mining. Hard sample mining has been actively studied in computer vision, especially for object detection, image segmentation, and object recognition tasks. The motivation for this approach is that hard negative (i.e. unexpected) samples are more informative than easy negatives (i.e. expected) ones.

The idea of CRL can be explained as follows. Consider a multi-class multi-label classification problem. Assume we have n_{attr} labels in total, each

label has Z_j classes numbered from one to Z_j. Let $\mathbf{h}^j = [h_1^j, \ldots, h_k^j, \ldots, h_{Z_j}^j]$ be the class distribution of label j where h_k^j is the number of training samples belongs to class k. The set of minority classes C_{min} in a minibatch of size n_{bs} are defined as:

$$\sum_{k \in C_{min}} h_k^j \leq \rho \cdot n_{bs} \ , \tag{6.5}$$

in which ρ is a threshold value ($\rho = 50\%$ in the paper).

After profiling the minority classes, the authors examined hard mining method. For any label j, a sample $x_{i,j}$ is considered as a class-level hard positive sample of a minority class c if $x_{i,j}$ is an instance of class c but has a low prediction score on c by the current model. In contrast, $x_{i,j}$ is a hard negative sample of class c if $x_{i,j}$ is not an instance of class c but is predicted with a high confidence score on class c by the current model. Class-level hard sample mining is illustrated in Figure 6.1.

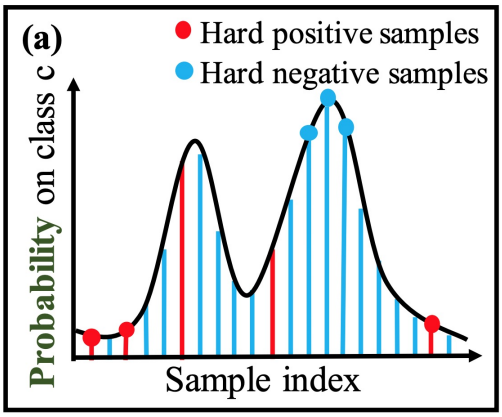


Figure 6.1: Illustration of class-level hard sample mining [9]. Top-3 hard positive samples (red dots) and hard negative samples (blue dots) are shown.

Given the hard sample mining technique, Dong et al. formulate the CRL loss as a triplet ranking loss. Each sample from all the minority classes of label j is regarded as an **anchor**. For each anchor $x_{a,j}$ of class c, they construct the triplets by mining top-k hard positives ($x_{+,j}$) and top-k hard negatives ($x_{-,j}$). Let T be the set of triplets $(x_{a,j}, x_{+,j}, x_{-,j})$. Suppose there are n_{ac} anchors, the set T then contains $n_{ac}k^2$ triplets. The CRL loss is then formulated as:

$$\mathcal{L}_{crl} = \frac{\sum_T \max(0, m_j + d(x_{a,j}, x_{+,j}) - d(x_{a,j}, x_{-,j}))}{|T|} \ , \tag{6.6}$$

where m_j is the class margin of label j and $d(\cdot)$ is the distance between two samples which is defined as:

$$d(x_{a,j}, x_{+,j}) = |p_{a,j} - p_{+,j}|, \quad d(x_{a,j}, x_{-,j}) = p_{a,j} - p_{-,j} \ , \tag{6.7}$$

where $p_{*,j}$ is the prediction score of $x_{*,j}$ on class c of label j, $* \in \{a, +, -\}$.

The mini-batch CRL loss is further combined with the cross-entropy loss to construct a weighted balanced loss $\mathcal{L}_{bln}$:

$$\mathcal{L}_{bln} = \alpha \mathcal{L}_{crl} + (1 - \alpha)\mathcal{L}_{ce}, \quad \alpha = \eta \Omega_{imb} \ , \tag{6.8}$$

in which α is a weighting factor, Ω_{imb} is the class imbalance measure, defined as minimum percentage of data samples across all classes to form a uniform distribution for label j, η is a hyperparameter that measures the linear relationship between the α and Ω_{imb}.

CRL proved its effectiveness on standard datasets and tested on different deep learning models (CifarNet [28], ResNet32 [18], DenseNet [21]).

In this work, we only implemented the CRL loss at class-level due to time constraint. We used the recommended values for the hyperparameters in our implementation: $m_j = 0.5$, $\eta = 0.01$. However, the results of CRL for our problem did not meet our expectation. This was the reason for us to experiment with another loss function, namely focal loss, which will be described in Section 6.4.3.

6.4.3 Focal loss

The standard cross-entropy (CE) loss optimizes the learning algorithm based on the assumption that all classification errors are equally important. This assumption is no longer correct in a class imbalanced setting. The model is biased towards the majority class and thus leading to a suboptimal solution. Besides subsampling, cost-sensitive learning, and class rectification loss, another approach to mitigate this class imbalance problem is to use a loss function, regarded as focal loss. Focal loss is a novel loss function proposed by Lin et al. [35] that dynamically scales the standard CE loss depending on the prediction confidence for the correct classes. This behavior allows the focal loss to pay less attention to the easy samples and focus more on hard samples.

The focal loss was designed to address the extreme imbalance between the foreground and background classes in the object detection task. Consider the binary cross-entropy loss:

$$\mathcal{L}_{ce} = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1 - p) & \text{otherwise,} \end{cases} \tag{6.9}$$

in which $y \in \pm 1$ is the ground-truth label and $p \in [0, 1]$ is the predicted score that the sample belongs to class with label $y = 1$. If we define:

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{otherwise,} \end{cases} \tag{6.10}$$

the cross entropy loss becomes: $\mathcal{L}_{ce} = -\log p_t$.

As to balance the cross-entropy loss, the authors introduced the weighting factor α for class $y = 1$ and $1 - \alpha$ for class $y = -1$. The α-balanced CE loss is then defined as:

$$\mathcal{L}_{ce} = -\alpha_t \log p_t \ , \tag{6.11}$$

where α_t is defined in the same fashion as p_t.

Even though α-balanced CE loss rectifies the contribution of positive and negative samples, it does not distinguish the difference between easy and hard samples. To avoid this issue, Lin et al. added the regulating factor $(1 - p_t)^\gamma$ to the standard CE loss, making a novel loss - the focal loss:

$$\mathcal{L}_{fl} = -\alpha_t (1 - p_t)^\gamma \log p_t \ , \tag{6.12}$$

in which γ is a focusing hyperparameter.

It can be seen from the focal loss that when a sample is badly misclassified, which means p_t is small, the modulating factor is close to one, the loss does not change. However, when the sample is well-predicted, which means p_t is close to one, the modulating factor becomes smaller. As a result, the loss of easy samples is down-weighted. This is the desired behavior to improve imbalance learning.

In this work, we used the recommended values for the balance factor ($\alpha = 0.25$) and focusing parameter ($\gamma = 2$). With the focal loss, our model achieved better results on our multi-class multi-label classification tasks. The results will be presented in Chapter 7.

6.5 Task balancing for multi-task learning

As described in Chapter 2, multi-task learning can offer regularization effect and thus improve training time and performance for deep learning models. However, training these models is rather challenging due to the complex relationship between the tasks. Regarding the prior work on multi-task learning, there are two main directions: architecture-based approach and loss-based approach. The architecture-based approach requires more parameters to learn and is often error-prone for novel tasks. Some of the architecture-based approaches are multilinear relationship networks [39], fully-adaptive

feature sharing [40], cross-stitch networks [41], sluice networks [51], and a joint many-task model [16]. The loss-based approach tries to form a novel loss function that adaptively weighs the contribution of each task's loss in the total loss. Recent loss-based techniques include uncertainty loss [24] and GradNorm loss [8].

In this work, we only explore the loss-based approaches. Most of the previous work in this direction usually uses either uniform or manually tuned weighted sum of losses:

$$\mathcal{L}(t) = \sum_{i=1}^{N} w_i(t)\mathcal{L}_i(t) \ , \tag{6.13}$$

where N is the number of tasks, $w_i(t)$ and $\mathcal{L}_i(t)$ are the task weights and task loss at timestep t. $w_i(t)$ is often a constant in naive approaches.

Tuning these weights is expensive and often requires prior domain knowledge. Therefore, it is advantageous to be able to learn the optimal weights automatically. There are two recent approaches that address this issue. The first one uses uncertainty and the second one uses gradient normalization to adaptively balance the losses. These two approaches were experimented in this work and will be described in the following sections.

6.5.1 Uncertainty loss

Kendall et al. [24] proposed a novel approach by considering the homoscedastic uncertainty of every single task in both classification and regression settings. This approach, which is regarded as uncertainty loss, tries to optimize the task weightings using probabilistic modeling.

Homoscedastic uncertainty is a type of uncertainty that does not depend on input data. Instead, it is a task-dependent uncertainty that varies between different tasks. In multi-task learning, homoscedastic uncertainty can capture the relationship between related tasks.

Let $\mathbf{f}(\mathbf{x}; \mathbf{W})$ be the output of a deep neural network, in which $\mathbf{W}$ is the weights and $\mathbf{x}$ is the input. Assume the regression output follows a Gaussian distribution with mean $\mathbf{f}(\mathbf{x}; \mathbf{W})$ and variance σ^2, the likelihood of the output is:

$$p(\mathbf{y}|\mathbf{f}(\mathbf{x}; \mathbf{W})) = \mathcal{N}(\mathbf{f}(\mathbf{x}; \mathbf{W}), \sigma^2) \ . \tag{6.14}$$

For classification, the likelihood of the output is normally fed into a softmax function. Moreover, Kendall et al. used a scaled version of the model output [24]:

$$p(\mathbf{y}|\mathbf{f}(\mathbf{x}; \mathbf{W}), \sigma) = \text{Softmax}(\frac{1}{\sigma^2}\mathbf{f}(\mathbf{x}; \mathbf{W})) \ , \tag{6.15}$$

in which σ is the temperature.

The model is optimized by maximizing the log-likelihood of the output. The log-likelihood of regression output is:

$$\log p(\mathbf{y}|\mathbf{f}(\mathbf{x};\mathbf{W})) \propto -\frac{1}{2\sigma^2}\|\mathbf{y}-\mathbf{f}(\mathbf{x};\mathbf{W})\|^2 - \log\sigma \ , \tag{6.16}$$

and that of classification output is:

$$\begin{aligned}
\log p(\mathbf{y}|\mathbf{f}(\mathbf{x};\mathbf{W}),\sigma) &= \log \text{Softmax}(\frac{1}{\sigma^2}\mathbf{f}(\mathbf{x};\mathbf{W})) \\
&= \log \frac{\exp(\frac{1}{\sigma^2}f_c(\mathbf{x};\mathbf{W}))}{\sum_{c'}\exp(\frac{1}{\sigma^2}f_{c'}(\mathbf{x};\mathbf{W}))} \\
&= \frac{1}{\sigma^2}f_c(\mathbf{x};\mathbf{W}) - \log\sum_{c'}\exp(\frac{1}{\sigma^2}f_{c'}(\mathbf{x};\mathbf{W})) \ ,
\end{aligned}$$

in which $f_c(\mathbf{x};\mathbf{W})$ is the c^{th} element of vector $\mathbf{f}(\mathbf{x};\mathbf{W})$.

Without the loss of generality, assume the neural network has two outputs: one for regression $\mathbf{y}_1$ and one for classification $\mathbf{y}_2$. The joint loss $\mathcal{L}(\mathbf{W},\sigma_1,\sigma_2)$ can be written as [24]:

$$\begin{aligned}
\mathcal{L}(\mathbf{W},\sigma_1,\sigma_2) &= -\log p(\mathbf{y}_1,\mathbf{y}_2=c|\mathbf{f}(\mathbf{x};\mathbf{W})) \\
&= -\log\mathcal{N}(\mathbf{y}_1;\mathbf{f}(\mathbf{x};\mathbf{W}),\sigma_1^2) - \log p(\mathbf{y}_2=c|\mathbf{f}(\mathbf{x};\mathbf{W}),\sigma_2) \\
&= \frac{1}{2\sigma_1^2}\|\mathbf{y}_1 - f_c(\mathbf{x};\mathbf{W})\|^2 + \log\sigma_1 - \frac{1}{\sigma_2^2}f_c(\mathbf{x};\mathbf{W}) \\
&\qquad + \log\sum_{c'}\exp(\frac{1}{\sigma_2^2}f_{c'}(\mathbf{x};\mathbf{W})) \\
&= \frac{1}{2\sigma_1^2}\mathcal{L}_1(\mathbf{W}) + \log\sigma_1 - \frac{1}{\sigma_2^2}f_c(\mathbf{x};\mathbf{W}) + \log\sum_{c'}\exp(f_{c'}(\mathbf{x};\mathbf{W}))^{\frac{1}{\sigma_2^2}} \\
&\qquad + \log\sum_{c'}\exp(\frac{1}{\sigma_2^2}f_{c'}(\mathbf{x};\mathbf{W})) - \log\sum_{c'}\exp(f_{c'}(\mathbf{x};\mathbf{W}))^{\frac{1}{\sigma_2^2}} \\
&= \frac{1}{2\sigma_1^2}\mathcal{L}_1(\mathbf{W}) + \log\sigma_1 + \frac{1}{\sigma_2^2}\Big(-f_c(\mathbf{x};\mathbf{W}) + \log\sum_{c'}\exp(f_{c'}(\mathbf{x};\mathbf{W}))\Big) \\
&\qquad + \log\frac{\sum_{c'}\exp(\frac{1}{\sigma_2^2}f_{c'}(\mathbf{x};\mathbf{W}))}{\sum_{c'}\exp(f_{c'}(\mathbf{x};\mathbf{W}))^{\frac{1}{\sigma_2^2}}} \\
&\approx \frac{1}{2\sigma_1^2}\mathcal{L}_1(\mathbf{W}) + \log\sigma_1 + \frac{1}{\sigma_2^2}\mathcal{L}_2(\mathbf{W}) + \log\sigma_2 \ ,
\end{aligned}$$

where $\mathcal{L}_1(\mathbf{W}) = \|\mathbf{y}_1 - f_c(\mathbf{x};\mathbf{W})\|^2$ and $\mathcal{L}_2(\mathbf{W}) = -\log\text{Softmax}(\mathbf{y}_2, f_c(\mathbf{x};\mathbf{W}))$. The authors derived the final equation based on the following assumption: $\frac{1}{\sigma_2^2}\sum_{c'}\exp(\frac{1}{\sigma_2^2}f_{c'}(\mathbf{x};\mathbf{W})) \approx \sum_{c'}\exp(f_{c'}(\mathbf{x};\mathbf{W}))^{\frac{1}{\sigma_2^2}}$ when $\sigma_2 \to 1$.

The objective function indicates that σ_1 and σ_2 regulate the contributions of $\mathcal{L}_1(\mathbf{W})$ and $\mathcal{L}_2(\mathbf{W})$ to the total loss. The uncertainty loss can be easily extended to multiple regression and classification outputs in the same fashion.

From the implementation point of view, the authors trained the log variance $s = \log(\sigma^2)$ for numerical stability. Our implementation of uncertainty loss also uses this trick.

6.5.2 GradNorm loss

Besides uncertainty loss, we also tested our model with another loss-based method known as GradNorm loss. This method helped improve the overall accuracy of our model.

GradNorm loss is proposed by Chen et al. [8] to learn the function $w_i(t)$ to balance the gradient by penalizing the gradient during backpropagation procedure. Like uncertainty loss, this function regulates the contributions of each task loss so that the tasks are trained at similar rates.

GradNorm algorithm requires several quantities to compute the gradient norm, including the subset of the network weights $\mathbf{W}$, the $L2$ norm of the gradient of a weighted task loss at time step t:

$$G^{(i)}_{\mathbf{W}}(t) = \|\nabla_{\mathbf{W}} w_i(t)\mathcal{L}_i(t)\|_2 \,, \tag{6.17}$$

and the average gradient norm:

$$\bar{G}_{\mathbf{W}}(t) = E[G^{(i)}_{\mathbf{W}}(t)] \,. \tag{6.18}$$

Besides, the algorithm defines loss ratio of task i at time step t as:

$$\tilde{\mathcal{L}}_i(t) = \frac{\mathcal{L}_i(t)}{\mathcal{L}_i(0)} \,. \tag{6.19}$$

For a single task i, $\tilde{\mathcal{L}}_i(t)$ measures the inverse training rate, meaning that the value of $\tilde{\mathcal{L}}_i(t)$ is inversely proportional to how fast task i is trained. This leads to the formation of the relative inverse training rate:

$$r_i(t) = \frac{\tilde{\mathcal{L}}_i(t)}{E_{\text{task}}[\tilde{\mathcal{L}}_i(t)]} \,,$$

that indicates how much the gradient magnitudes is needed to train task i. The desired gradient norm for task i is then given by:

$$G^{(i)}_{\mathbf{W}}(t) \mapsto \bar{G}_{\mathbf{W}}(t) \cdot [r_i(t)]^{\alpha} \,, \tag{6.20}$$

where α is a hyperparameter that modulates the strength of training rate balancing. When the tasks are very different in training rate, the value of α should be large and vice versa.

We want to train the weights such that the current gradient norm gets closer to the desired gradient norm. Thus, we want to optimize the distance norm between the actual gradient and the desired gradient. The GradNorm is then computed as:

$$\mathcal{L}_{\text{grad}}(t; w_i(t)) = \sum_i \| G^{(i)}_{\mathbf{W}}(t) - \bar{G}_{\mathbf{W}}(t) \cdot [r_i(t)]^{\alpha} \|_1 \ . \tag{6.21}$$

The pseudo code of GradNorm algorithm is shown in Algorithm 2. GradNorm proved to be an efficient algorithm for multi-task learning. It was tested on both synthetic and real datasets.

Algorithm 2 Training with GradNorm [8].

Initialize $w_i(0) = 1 \ \forall i$
Initialize network weight $\mathcal{W}$
Pick value for $\alpha > 0$ and pick the weights $\mathbf{W}$
for $t = 0$ **to** *max_train_step* **do**
  Input batch x_i to compute $L_i(t) \ \forall i$:
    $\mathcal{L}(t) = \sum_{i=1}^{N} w_i(t)\mathcal{L}_i(t)$. [forward pass]
  Compute $G^{(i)}_{\mathbf{W}}(t)$ and $r_i(t) \ \forall i$
  Compute $\bar{G}_{\mathbf{W}}(t)$ by averaging the $G^{(i)}_{\mathbf{W}}(t)$
  Compute $\mathcal{L}_{\text{grad}}(t) = \sum_i \| G^{(i)}_{\mathbf{W}}(t) - \bar{G}_{\mathbf{W}}(t) \cdot [r_i(t)]^{\alpha} \|_1$
  Compute GradNorm gradients $\nabla_{w_i}\mathcal{L}_{grad}(t)$, keeping target
    $\bar{G}_{\mathbf{W}}(t) \cdot [r_i(t)]^{\alpha}$ constant
  Compute standard gradient $\nabla_{\mathcal{W}}\mathcal{L}(t)$
  Update $w_i(t) \mapsto w_i(t+1)$ using $\nabla_{w_i}\mathcal{L}_{grad}$
  Update weights $\mathcal{W}(t) \mapsto \mathcal{W}(t+1)$ using $\nabla_{\mathcal{W}}\mathcal{L}(t)$ [backward pass]
  Renormalize $w_i(t+1)$ so that $\sum_i^N w_i(t+1) = T$
end for

6.6 Final model and parameter settings

In the end, we propose our final model as a novel CNN for multi-task learning as shown in Figure 6.2. Our CNN architecture includes seven convolution layers, one shared FC layer, and two FC layers for each specific task. The depth of the network is seven, which is considered deep since most of the CNNs for HSI related tasks are shallower.

Layers	C1	C1-1	C1-2	C1-3	C1-4	C2	C3
Knl	$3 \times 3 \times 48$	$1 \times 1 \times 1$	$1 \times 1 \times 3$	$1 \times 1 \times 5$	$1 \times 1 \times 11$	$3 \times 3 \times 32$	$3 \times 3 \times 32$
Pad	0	0	$0 \times 0 \times 1$	$0 \times 0 \times 2$	$0 \times 0 \times 5$	0	0
C_{out}	128	128	128	128	128	64	32
$Pool$	$2 \times 2 \times 1$	-	-	-	-	$2 \times 2 \times 1$	-

Table 6.3: Details of the proposed CNN architecture. We use some abbreviations here due to lack of space: Knl = kernel size, Pad = Padding size, C_{out} = number of output plans in 3D convolution, Pool = pooling kernel size.

Table 6.3 shows the proposed architecture in detail. In the table, layer C1 stands for Conv1 layer in the Figure 6.2. Similarly, C1-1 stands for Conv1-1 layer and so on. Besides the convolution layers, the shared FC layer takes the output from the last convolution layer, flattens it and transforms it into a 512-length feature vector. The task-specific layers are the same as in the baseline model which is described in Section 5.2.

In all our experiments, we trained our models with the learning rate at 10^{-4}, batch size of 64 samples, and patch size of 27 pixels for the input patches. We also used Adam optimizer (Section 2.3.5.7) with the weight decay η at 10^{-4}, ρ_1 at 0.9 and ρ_2 at 0.999. For CRL (Section 6.4.2), focal loss (Section 6.4.3), uncertainty loss (Section 6.5.1) and GradNorm loss (Section 6.5.2), we used the same parameters suggested by the original authors.

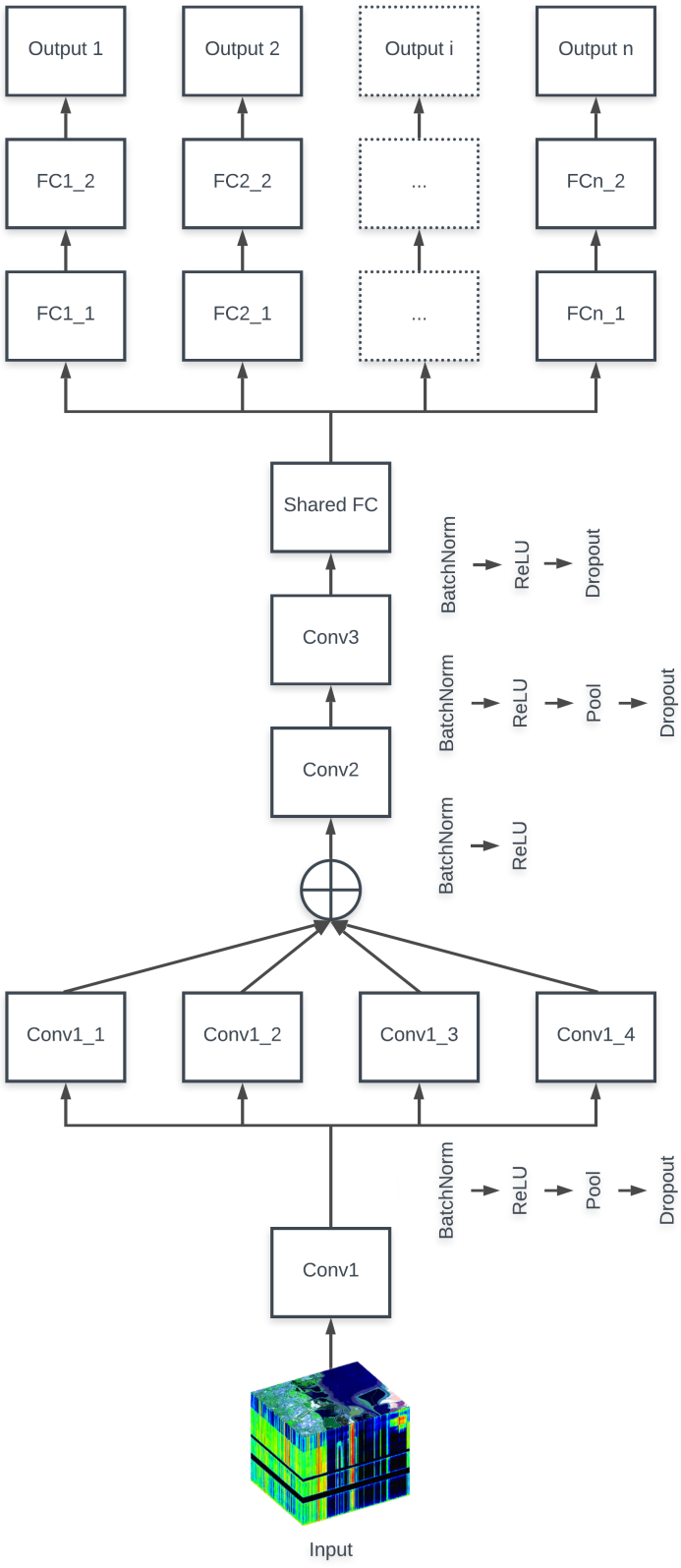


Figure 6.2: The proposed CNN architecture for HSI multi-task learning.

Chapter 7

Experiments and results

We have discussed the core concepts in machine learning as well as various implementation considerations in previous chapters. In this chapter, experiments and results of the proposed CNN architecture for modeling forest biomass and structures are presented. The performance of the proposed model is evaluated based on the balanced accuracy and AUC score metrics introduced in Chapter 4.

Other related CNN architectures for similar tasks were mainly trained on the three existing HSI datasets: Indian Pines (size of 145×145 pixels, 220 spectral bands), the University of Pavia (size of 610×340 pixels, 103 spectral bands) and Salinas (size of 512×217 pixels, 204 spectral bands). However, these datasets are small and thus have limited samples. Most of the CNNs trained on these datasets use overlapped patches as training samples [6, 19, 32]. Moreover, they were trained for a multi-class classification task with a single label. Meanwhile, our model is trained on a different dataset and is designed for multi-task learning, including regression and multi-label multi-class classification. Hence, it is not possible to fairly compare the performance between our model and other existing models for HSI.

7.1 Effect of multi-scale convolution block

This experiment aimed to understand the effect of using multi-scale convolution block. To this end, we built five different models and named them M1 to M5. Model M1 has the same architecture as our final model, as shown in Figure 6.2. M2 is an extended version of M1 with another multi-scale block after Conv2 layer. M3 is the same as M1, but uses four kernels with the same size of $3 \times 3 \times 5$ with padding $(1 \times 1 \times 0)$. M4 is also similar to M1, but applies kernels of size to $3 \times 3 \times B_1$ with padding $(1 \times 1 \times B_2)$ in which the

Models	**OA**	**BA**	**AUC**	**MAE**
M1	71.25	**73.79**	**0.827**	0.082
M2	71.36	70.35	0.792	**0.080**
M3	72.14	71.63	0.801	0.087
M4	**73.45**	71.89	0.814	0.093
M5	73.24	72.29	0.803	0.094

Table 7.1: Evaluation of models with multi-scale convolution blocks.

kernel's depth B_1 and the padding's depth B_2 are kept unchanged. M5 is M1 without the multi-scale block. These models were trained for 50 epochs and results are shown in Table 7.1, where OA stands for overall accuracy and BA stands for balanced accuracy.

Table 7.1 indicates that the overall accuracies of models M2, M3 and M4, in all of which the importance of classes are weighted equally, are better. However, the balanced accuracies and AUC score are worse. The results suggest that having too many multi-scale convolution blocks or having a bigger kernel size for the convolution operations is not always beneficial. A possible reason for this could be that adding more multi-scale blocks or expanding the kernel size increases the size of the network, thus leading to overfitting.

7.2 Effect of cost-sensitive learning, focal loss, CRL

In order to address the class imbalance issue, we adopted three methods: cost-sensitive learning, focal loss, and class rectification loss. For cost-sensitive learning, the inverse median frequency scheme, as described in Section 6.4.1, was used to compute class weights. This weighting scheme penalizes errors for samples from minority classes more severely than the ones from majority classes. For focal loss and CRL, we also used the same weighting scheme for the cross-entropy term in these two losses. We found that without the weights for the CE loss, these losses performed worse than the cost-sensitive learning method. We trained our proposed CNN architecture with all these loss functions for 50 epochs. Table 7.2 contains the results of these training runs.

As Table 7.2 suggests, focal loss with class weights in the CE term gave the best results. Without class weights, both CRL and focal loss performed much worse. In fact, high overall accuracy indicates that they were biased towards

Method	OA	BA	AUC
Cost sensitive	71.08	72.12	0.813
CRL w/o class weights	75.12	63.45	0.627
CRL w/ class weights	72.14	72.54	0.811
Focal loss w/o class weights	**77.78**	66.43	0.694
Focal loss w/ class weights	71.52	**73.79**	**0.827**

Table 7.2: Evaluation of three different methods for class imbalance problem: cost-sensitive learning, class rectification loss, focal loss.

Model	w/ BN	OA	BA	AUC	MAE
M1	No	63.60	65.41	0.759	0.087
M1	Yes	71.25	**73.79**	**0.827**	0.082
M2	No	64.80	65.02	0.742	0.088
M2	Yes	71.36	70.35	0.791	**0.080**
M3	No	64.79	65.11	0.744	0.089
M3	Yes	72.14	71.63	0.803	0.087
M4	No	66.23	63.65	0.710	0.097
M4	Yes	**73.45**	71.89	0.812	0.093
M5	No	62.15	65.21	0.704	0.099
M5	Yes	73.24	72.29	0.805	0.095

Table 7.3: Performance evaluation with batch normalization.

the majority classes. The CRL improved the performance in comparison to the cost-sensitive learning, but the difference is small and insignificant.

7.3 Effect of batch normalization and dropout

Batch normalization (BN) and dropout have proven to be efficient regularizers for neural networks and are widely adopted by the deep learning community. To study the effect of batch normalization and dropout, we again used five models M1-M5 described in Section 7.1. We compared the performance of these models in three settings: with BN and dropout, without BN, and without dropout. If BN and dropout are used, they are placed after each convolution layer. For the multi-scale convolution blocks, however, we only apply batch normalization after summing up all the convolution layers. The results for BN's effect and dropout's effect are shown in Tables 7.3 and 7.4 respectively.

It is evident that batch normalization made a substantial performance

Model	w/ BN	OA	BA	AUC	MAE
M1	No	70.35	71.20	0.799	0.083
M1	Yes	71.25	**73.79**	**0.827**	0.082
M2	No	71.03	68.46	0.792	0.091
M2	Yes	71.36	70.35	0.791	**0.080**
M3	No	72.15	70.21	0.794	0.097
M3	Yes	72.14	71.63	0.803	0.087
M4	No	75.01	70.19	0.801	0.091
M4	Yes	**73.45**	71.89	0.812	0.093
M5	No	69.26	72.10	0.783	0.101
M5	Yes	73.24	72.29	0.805	0.095

Table 7.4: Performance evaluation with dropout.

gain among all models. The biggest improvement was achieved by model M1 with 8.39% increase in balanced accuracy. The regression task, however, did not get much benefit. The validation MAE slightly reduced from 0.087 to 0.082.

Like batch normalization, dropout had a positive impact on a model's performance. However, the improvement was not as significant as that of batch normalization. Without dropout, the balanced accuracy dropped about 1-2 percentages on average for all five models. The AUC scores and MAE show a modest drop in most of the models.

Batch normalization and dropout contributed significantly to the convergence of the model. Figure 7.1 illustrates the impact of these two regularizers in the training speed of the model. It is shown that both dropout and batch normalization accelerated the learning process by reducing the training error more quickly. Batch norm again demonstrated its dominance over dropout by showing a larger contribution to the error reduction for this dataset.

7.4 GradNorm loss, uncertainty loss

Task imbalance is a long-standing problem in multi-task learning. This section presents our findings for two loss functions, namely GradNorm loss and uncertainty loss, which were designed to automatically adjust the task weights according to their contribution to the total loss. To measure the performance of these two approaches, we applied GradNorm and uncertainty losses to our proposed model and kept other settings the same.

Our model has to predict four categorical and twelve continuous variables,

Figure 7.1: Training loss with and without batch normalization and dropout.

making up 16 tasks in total. The training losses of these tasks are not on the same scale. The training task losses are shown in Figure 7.2. Tasks are numbered from 0 to 15. The first four tasks are classification and the rest are regression tasks. It is evident that classification losses are much larger compared to regression ones. If we treat these losses equally, the training rate for regression tasks could become much slower than the classification tasks because the model focuses more on the tasks that contribute more to the total loss, which are classification tasks in this case. GradNorm and uncertainty losses aim to balance the training rate by adaptively regulate their weights as shown in Figure 7.3.

Figure 7.3 shows how the task weights change after 15000 training steps. At first, all task weights were initialized to one. We can see in both Figures 7.3a and 7.3b that among the five tasks that have the smallest weights, four of them are classification tasks. The results illustrate that both algorithms correctly adjusted the task weights according to the contribution of each task in the total loss.

Looking at the numerical results would give a more accurate assessment of the efficacy of these methods. Table 7.5 compares the performance of the proposed model trained with three different weighting schemes: equal (naive approach), GradNorm and uncertainty weighting. The results suggest that both GradNorm and uncertainty enhanced the model's performance.

Figure 7.2: Training losses for 16 different tasks.

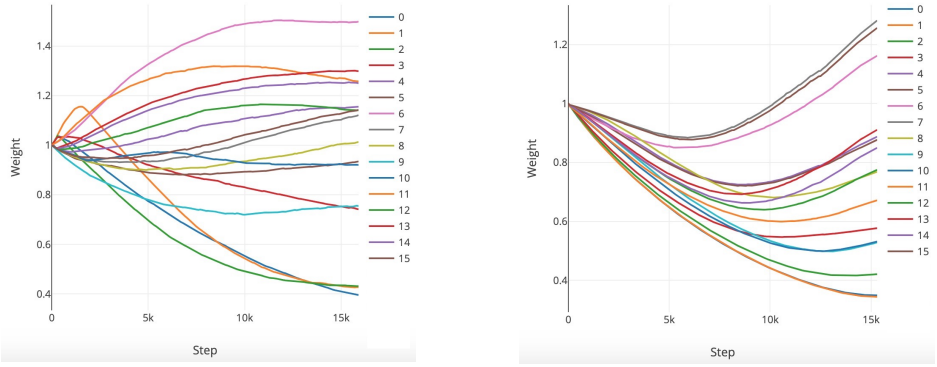
Methods	OA	BA	AUC	MAE
Equal weights	71.25	73.79	0.827	0.082
GradNorm	**72.34**	**75.53**	**0.857**	**0.072**
Uncertainty	71.12	74.37	0.828	0.081

Table 7.5: Performance evaluation for GradNorm and uncertainty weighting schemes.

7.5 Final results

In this section, the final results of the thesis work are presented. Our best model is a combination of the final CNN architecture shown in Figure 6.2, the focal loss for classification tasks, the MSE loss for regression tasks and GradNorm loss for multi-task learning. We also applied data augmentation methods introduced in Section 6.1.

Figure 7.4 indicates the normalized confusion matrices for the classification tasks at the beginning and end of the training. We ran the model after every ten epochs on the validation set to supervise the learning process. In the beginning, the model tried to predict the minority class due to heavy



(a) GradNorm *(b) Uncertainty*

Figure 7.3: Adaptive task weights learned by two algorithms.

penalty from the classification loss function. At the end of the training, the model performed much better after seeing the training samples many times.

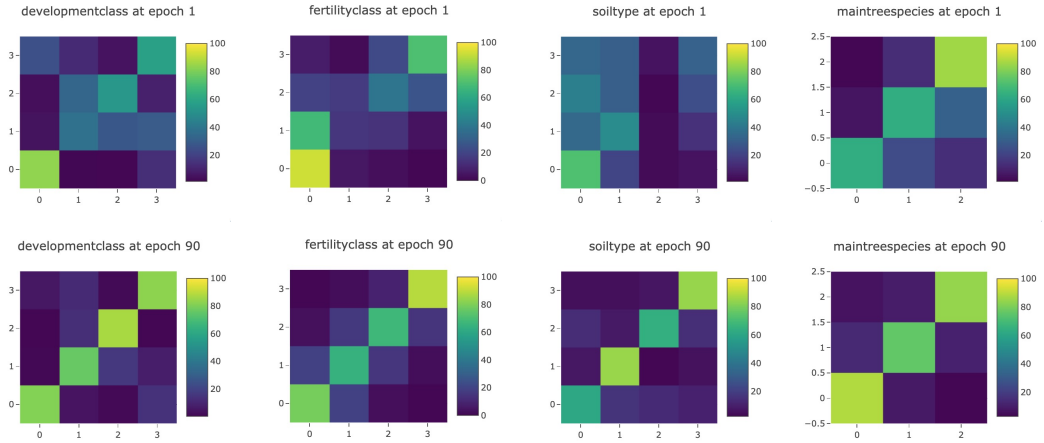


Figure 7.4: Normalized confusion matrices for four classification tasks at epoch 1 and 90. The horizontal axis shows the true classes, and the vertical axis shows the predicted classes.

The numerical results for classification tasks are displayed in Table 7.6. The predictions on the development class and the main tree species are significantly better than on the fertility class and the soil type. The reason is because of the noise in the data. The collected data for soil type and fertility class is noisier than the development class and the main tree species.

For regression, results of three randomly selected variables are shown in Figure 7.5. Regarding the scatter plots (on rows 1, 3, 5), the x-axis represents

Task	OA	BA	AUC
Development class	79.89	81.77	0.856
Fertility class	68.17	74.94	0.846
Soil type	67.31	73.69	0.811
Main tree species	**84.60**	**82.88**	**0.873**
Average	74.99	78.32	0.847

Table 7.6: Accuracy and AUC scores for classification tasks.

the target labels and the y-axis represents the predicted labels. Ideally, all the points should be on the line $y = x$ or the red dashed line in the plots. The marginal plot on top of each plot is the true distribution while the marginal plot on the right is the predicted distribution. Similarly, the kernel density estimation plots (on rows 2, 4, 6) indicate the density of the points on the corresponding scatter plots. The darker a region is, the more points it has. A robust model should have the darkest regions around the modes of target distribution.

The results presented in Figure 7.5 for the three variables indicates that the predicted distributions are fairly close to the true distributions. High Pearson correlation coefficients (≈ 0.9 on average) in the scatter plots at the 90^{th} epoch and a low average MAE (0.052) also prove that the prediction and target labels for all the regression tasks are highly correlated.

Figure 7.6 shows the error histograms and error frequency cumulations of the three selected variables (Figures 7.6a, 7.6b, 7.6c) and the sum of errors of all regression tasks (Figure 7.6d). A histogram was created by computing the absolute error for each test sample and plotting the histogram of the error values with 100 bins. A good model will have most of the errors falling in bins that are close to zero. In the error histograms produced by our models, the modes are very close to zero. The error histogram of the error sum of all tasks gives an overview of how well the model performs in general. Figure 7.6d suggests that the mode of the distribution is around 0.5 which is acceptable. These results affirm that our model performed reasonably well on the regression tasks.

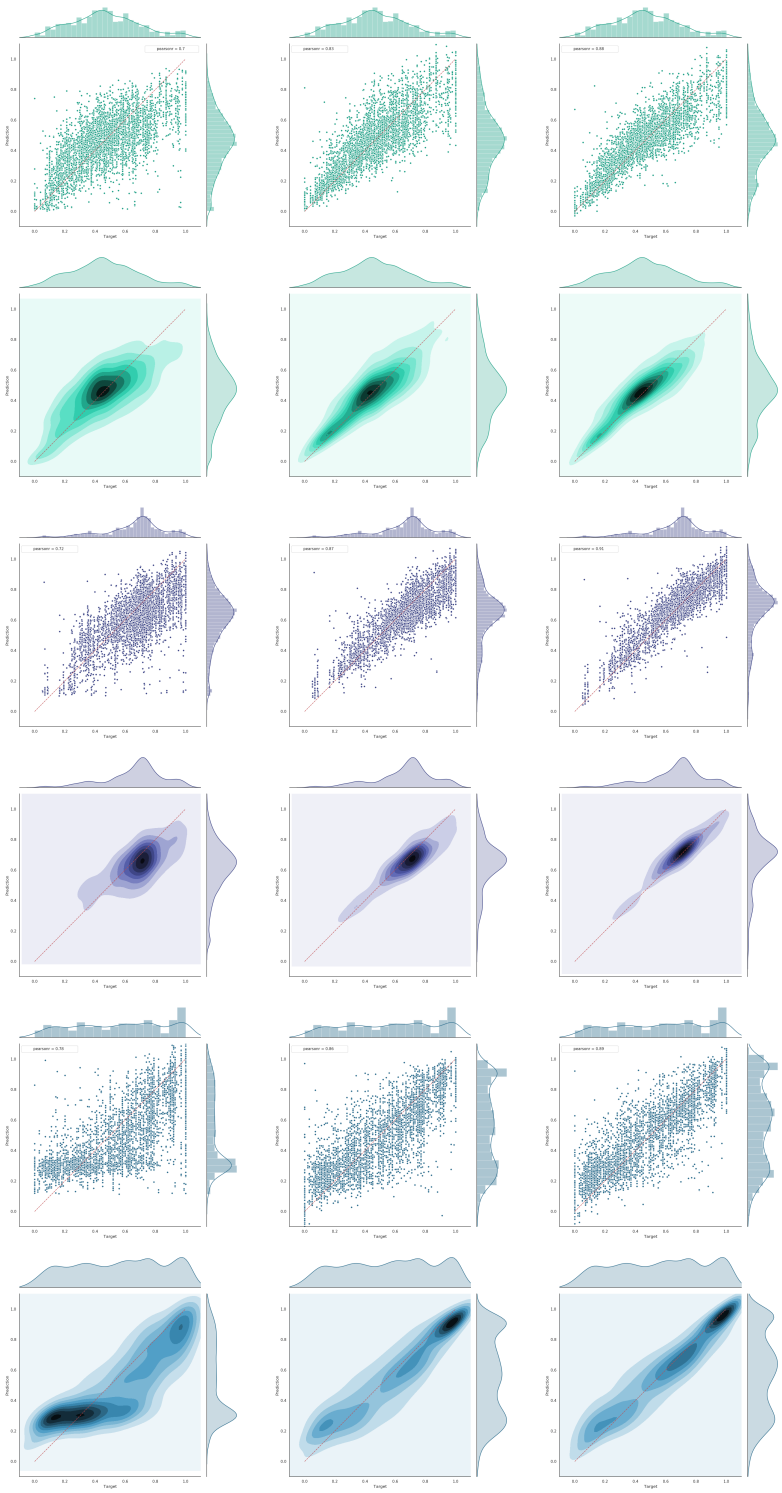
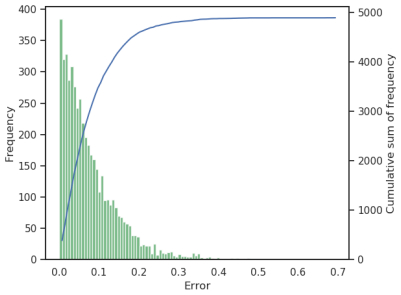
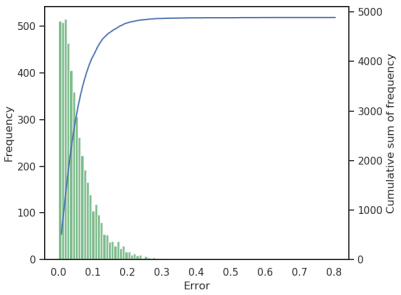


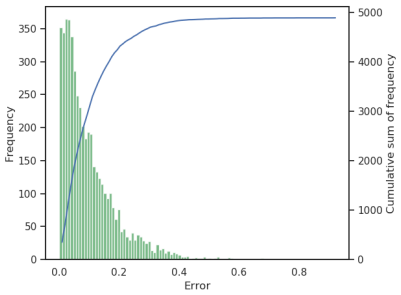
Figure 7.5: Scatter plots and kernel density estimation plots between prediction and target labels for three variables woody biomass (rows 1-2), mean height (rows 3-4) and percentage of pine (rows 4-6) after the $1^{st}, 40^{th}, 90^{th}$ epochs. The Pearson correlation coefficients for the woody biomass, the mean height and the percentage of pine variables at 90^{th} epoch are 0.88, 0.91 and 0.89 respectively.



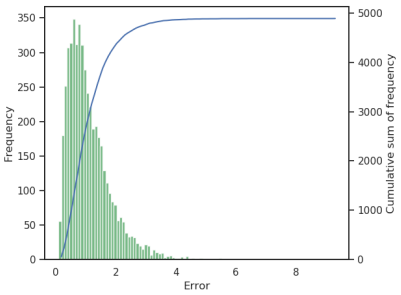
(a) Woody biomass



(b) Mean height



(c) Percentage of pine



(d) Sum of all tasks

Figure 7.6: Error histograms and error frequency cumulations.

Chapter 8

Conclusions

The objective of this thesis was to investigate and develop deep learning methods for modeling forest biomass and structures from hyperspectral imagery (HSI). In this work, we introduced a novel convolutional neural network (CNN) architecture that can simultaneously learn to approximate various probability distributions of the parameters in a large HSI dataset. The proposed CNN includes a shared part and a task-specific part. The shared part consists of three convolution layers, a multi-scale convolution block, and a fully-connected layer while the task-specific part contains two fully-connected layers for each learnable task. The three regular 3D convolution layers aim to extract the spatial and spectral correlations. Meanwhile, the multi-scale convolutional block consists of four convolution layers with different filter sizes to further exploit the spectral information. The combination of these two parts results in a 7-layer CNN which is deeper than most of the existing CNNs for HSI.

Besides the sole architecture design, we experimented with different techniques to address the following problems: overfitting, class imbalance, and multi-task learning. To prevent overfitting, we applied four regularization methods, including L2 regularization, batch normalization, dropout, and data augmentation. The results of the experiments showed that all of these methods improved our model's performance and that batch normalization had the most notable contribution. To address the class imbalance problem in our dataset, we employed three approaches: cost-sensitive learning, class rectification loss, and focal loss. Among these three, focal loss proved to be superior to the others. Regarding the multi-task learning, we adopted two loss functions, namely GradNorm and uncertainty, that balanced the training rate by adaptively regulating the task's contribution to the total loss. GradNorm outperformed uncertainty loss in our experiments.

In the end, we combined the techniques that produced the best results

to form the final setup. Based on the results, the proposed model could classify discrete variables with balanced accuracy up to 78.32% and estimate continuous variables with a significantly low average mean absolute error (0.052) and high Pearson correlation coefficients (≈ 0.9).

In summary, we proposed a robust CNN architecture that can adequately estimate the forest structures from HSI. Even though it meets our original objective, there are still improvements that can be made. First of all, a possible reason why we could not achieve higher balanced accuracy in the classification task is because of the class imbalance issue. Even though the methods we implemented boosted the model's performance, the improvement is not significant in comparison to the naive cost-sensitive learning approach. This opens an opportunity for future research in addressing the skewness of our HSI dataset. Another area deserving attention is residual learning. Residual learning can considerably increase the depth of the network and thus possibly improve the model's performance. We did not explore this option since a large number of spectral bands made the size of the network explode. Future work can focus on first finding a good representation of the hyperspectral image before adding residual blocks to the network.

Bibliography

[1] Atzberger, C. Object-based retrieval of biophysical canopy variables using artificial neural nets and radiative transfer models. *Remote Sensing of Environment 93*, 1 (2004), 53 – 67.

[2] Bottou, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010* (Heidelberg, 2010), Y. Lechevallier and G. Saporta, Eds., Physica-Verlag HD, pp. 177–186.

[3] Boyd, S., and Vandenberghe, L. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[4] Campbell, J., and Wynne, R. *Introduction to Remote Sensing*. Guilford Publications, 2011.

[5] Caruana, R. Multitask learning: A knowledge-based source of inductive bias. In *Proceedings of the Tenth International Conference on Machine Learning* (1993), Morgan Kaufmann, pp. 41–48.

[6] Chen, Y., Jiang, H., Li, C., Jia, X., and Ghamisi, P. Deep feature extraction and classification of hyperspectral images based on convolutional neural networks. *IEEE Transactions on Geoscience and Remote Sensing 54*, 10 (Oct 2016), 6232–6251.

[7] Chen, Y., Lin, Z., Zhao, X., Wang, G., and Gu, Y. Deep learning-based classification of hyperspectral data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 7*, 6 (June 2014), 2094–2107.

[8] Chen, Z., Badrinarayanan, V., Lee, C., and Rabinovich, A. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. *CoRR abs/1711.02257* (2017).

[9] Dong, Q., Gong, S., and Zhu, X. Imbalanced deep learning by minority class incremental rectification. *CoRR abs/1804.10851* (2018).

[10] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res. 12* (July 2011), 2121–2159.

[11] Duong, L., Cohn, T., Bird, S., and Cook, P. Low resource dependency parsing: Cross-lingual parameter sharing in a neural network parser. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)* (Beijing, China, July 2015), Association for Computational Linguistics, pp. 845–850.

[12] Elkan, C. The foundations of cost-sensitive learning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2* (San Francisco, CA, USA, 2001), IJCAI'01, Morgan Kaufmann Publishers Inc., pp. 973–978.

[13] Ghafoorian, M., Karssemeijer, N., Heskes, T., Bergkamp, M., Wissink, J., Obels, J., Keizer, K., de Leeuw, F., van Ginneken, B., Marchiori, E., and Platel, B. Deep multi-scale location-aware 3d convolutional neural networks for automated detection of lacunes of presumed vascular origin. *CoRR abs/1610.07442* (2016).

[14] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[15] Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. Maxout networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (2013), ICML'13, JMLR.org, pp. III–1319–III–1327.

[16] Hashimoto, K., Xiong, C., Tsuruoka, Y., and Socher, R. A joint many-task model: Growing a neural network for multiple NLP tasks. *CoRR abs/1611.01587* (2016).

[17] Haykin, S. S. *Neural networks and learning machines*, third ed. Pearson Education, Upper Saddle River, NJ, 2009.

[18] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[19] He, M., Li, B., and Chen, H. Multi-scale 3d deep convolutional neural network for hyperspectral image classification. In *2017 IEEE International Conference on Image Processing (ICIP)* (Sep. 2017), pp. 3904–3908.

[20] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural Computation 9*, 8 (1997), 1735–1780.

[21] Huang, G., Liu, Z., and Weinberger, K. Q. Densely connected convolutional networks. *CoRR abs/1608.06993* (2016).

[22] Huber, P. J. Robust estimation of a location parameter. *Ann. Math. Statist. 35*, 1 (03 1964), 73–101.

[23] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR abs/1502.03167* (2015).

[24] Kendall, A., Gal, Y., and Cipolla, R. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. *CoRR abs/1705.07115* (2017).

[25] Kimes, D., Knyazikhin, Y., Privette, J., Abuelgasim, A., and Gao, F. Inversion methods for physically-based models. 381–439. Exported from https://app.dimensions.ai on 2019/05/23.

[26] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015).

[27] Kivinen, J., Keurulainen, A., and Laaksonen, J. Lecture notes: Cs-e4890: Deep learning, Oct 2017.

[28] Krizhevsky, A. Learning multiple layers of features from tiny images. Tech. rep., 2009.

[29] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[30] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (Nov 1998), 2278–2324.

[31] Lee, C.-Y., Xie, S., Gallagher, P., Zhang, Z., and Tu, Z. Deeply-Supervised Nets. *arXiv e-prints* (Sep 2014), arXiv:1409.5185.

[32] Lee, H., and Kwon, H. Going deeper with contextual cnn for hyperspectral image classification. *IEEE Transactions on Image Processing 26*, 10 (Oct 2017), 4843–4855.

[33] Li, J., Zhang, S., and Huang, T. Multi-scale 3d convolution network for video based person re-identification. *CoRR abs/1811.07468* (2018).

[34] Lin, M., Chen, Q., and Yan, S. Network in network, 2013.

[35] Lin, T., Goyal, P., Girshick, R. B., He, K., and Dollár, P. Focal loss for dense object detection. *CoRR abs/1708.02002* (2017).

[36] Ling, C. X., and Sheng, V. S. Cost-sensitive learning and the class imbalance problem.

[37] Linnainmaa, S. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's thesis, Univ. Helsinki, 1970.

[38] Liu, P., Zhang, H., and Eom, K. B. Active deep learning for classification of hyperspectral images. *CoRR abs/1611.10031* (2016).

[39] Long, M., and Wang, J. Learning multiple tasks with deep relationship networks. *CoRR abs/1506.02117* (2015).

[40] Lu, Y., Kumar, A., Zhai, S., Cheng, Y., Javidi, T., and Feris, R. S. Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification. *CoRR abs/1611.05377* (2016).

[41] Misra, I., Shrivastava, A., Gupta, A., and Hebert, M. Cross-stitch networks for multi-task learning. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2016), pp. 3994–4003.

[42] Mobley, C. D., Sundman, L. K., Davis, C. O., Bowles, J. H., Downes, T. V., Leathers, R. A., Montes, M. J., Bissett, W. P., Kohler, D. D. R., Reid, R. P., Louchard, E. M., and Gleason, A. Interpretation of hyperspectral remote-sensing imagery

by spectrum matching and look-up tables. *Appl. Opt. 44*, 17 (Jun 2005), 3576–3592.

[43] Nair, V., and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning* (USA, 2010), ICML'10, Omnipress, pp. 807–814.

[44] NESTEROV, Y. A method for unconstrained convex minimization problem with the rate of convergence o($1/k^2$). *Doklady AN USSR 269* (1983), 543–547.

[45] Pan, Y., Birdsey, R. A., Phillips, O. L., and Jackson, R. B. The structure, distribution, and biomass of the world's forests. *Annual Review of Ecology, Evolution, and Systematics 44*, 1 (2013), 593–622.

[46] Qian, N. On the momentum term in gradient descent learning algorithms. *Neural Networks 12*, 1 (1999), 145 – 151.

[47] Qin, Z., Zhang, C., Wang, T., and Zhang, S. Cost sensitive classification in data mining. In *Advanced Data Mining and Applications* (Berlin, Heidelberg, 2010), L. Cao, Y. Feng, and J. Zhong, Eds., Springer Berlin Heidelberg, pp. 1–11.

[48] Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* (1958), 65–386.

[49] Ruder, S. An overview of gradient descent optimization algorithms. *CoRR abs/1609.04747* (2016).

[50] Ruder, S. An overview of multi-task learning in deep neural networks. *CoRR abs/1706.05098* (2017).

[51] Ruder, S., Bingel, J., Augenstein, I., and Søgaard, A. Latent Multi-task Architecture Learning. *arXiv e-prints* (May 2017), arXiv:1705.08142.

[52] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Neurocomputing: Foundations of research. MIT Press, Cambridge, MA, USA, 1988, ch. Learning Representations by Back-propagating Errors, pp. 696–699.

[53] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. Striving for simplicity: The all convolutional net, 2014.

[54] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In *International conference on machine learning* (2013), pp. 1139–1147.

[55] Thai-Nghe, N., Gantner, Z., and Schmidt-Thieme, L. Cost-sensitive learning methods for imbalanced data. In *The 2010 International Joint Conference on Neural Networks (IJCNN)* (July 2010), pp. 1–8.

[56] Tieleman, T., and Hinton, G. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[57] Verrelst, J., Rivera, J. P., Gitelson, A., Delegido, J., Moreno, J., and Camps-Valls, G. Spectral band selection for vegetation properties retrieval using gaussian processes regression. *International Journal of Applied Earth Observation and Geoinformation 52* (2016), 554 – 567.

[58] Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning* (Atlanta, Georgia, USA, 17–19 Jun 2013), S. Dasgupta and D. McAllester, Eds., vol. 28 of *Proceedings of Machine Learning Research*, PMLR, pp. 1058–1066.

[59] Weiss, G. M., McCarthy, K., and Zabar, B. Cost-sensitive learning vs. sampling: Which is best for handling unbalanced classes with unequal error costs? In *DMIN* (2007).

[60] Yang, Y., and Hospedales, T. M. Trace norm regularised deep multi-task learning. *CoRR abs/1606.04038* (2016).