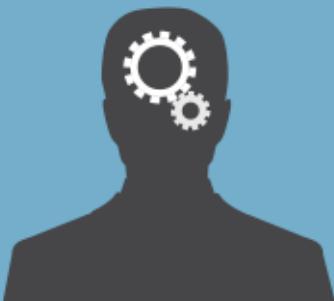


Better Deep Learning

Train Faster, Reduce Overfitting,
and Make Better Predictions

Jason Brownlee

MACHINE
LEARNING
MASTERY



Disclaimer

The information contained within this eBook is strictly for educational purposes. If you wish to apply ideas contained in this eBook, you are taking full responsibility for your actions.

The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic or mechanical, recording or by any information storage and retrieval system, without written permission from the author.

Acknowledgements

Special thanks to my proofreader Sarah Martin and my technical editors Andrei Cheremskoy, Michael Sanderson, Arun Koshy.

Copyright

Better Deep Learning

© Copyright 2020 Jason Brownlee. All Rights Reserved.

Edition: v1.8

Contents

Copyright	i
Contents	ii
Preface	iii
Introduction	v
Welcome	v
Framework for Better Deep Learning	x
Diagnostic Learning Curves	xix
I Better Learning	1
1 Improve Learning by Understanding Optimization	3
1.1 Neural Nets Learn a Mapping Function	3
1.2 Learning Network Weights Is Hard	4
1.3 Key Features of the Error Surface	6
1.4 Navigating the Non-Convex Error Surface	8
1.5 Implications for Training	9
1.6 Components of the Learning Algorithm	9
1.7 Further Reading	11
1.8 Summary	12
2 Configure Capacity with Nodes and Layers	14
2.1 Neural Network Model Capacity	14
2.2 Nodes and Layers Keras API	15
2.3 Model Capacity Case Study	17
2.4 Extensions	27
2.5 Further Reading	27
2.6 Summary	28

3 Configure Gradient Precision with Batch Size	29
3.1 Batch Size and Gradient Descent	29
3.2 Gradient Descent Keras API	31
3.3 Batch Size Case Study	32
3.4 Extensions	47
3.5 Further Reading	47
3.6 Summary	48
4 Configure What to Optimize with Loss Functions	49
4.1 Loss Functions	49
4.2 Regression Loss Functions Case Study	56
4.3 Binary Classification Loss Functions Case Study	64
4.4 Multiclass Classification Loss Functions Case Study	74
4.5 Extensions	84
4.6 Further Reading	84
4.7 Summary	86
5 Configure Speed of Learning with Learning Rate	87
5.1 Learning Rate	87
5.2 Learning Rate Keras API	93
5.3 Learning Rate Case Study	95
5.4 Extensions	118
5.5 Further Reading	118
5.6 Summary	119
6 Stabilize Learning with Data Scaling	121
6.1 Data Scaling	121
6.2 Data Scaling scikit-learn API	123
6.3 Data Scaling Case Study	125
6.4 Extensions	138
6.5 Further Reading	139
6.6 Summary	139
7 Fix Vanishing Gradients with ReLU	141
7.1 Vanishing Gradients and ReLU	141
7.2 ReLU Keras API	152
7.3 ReLU Case Study	152
7.4 Extensions	164
7.5 Further Reading	165
7.6 Summary	166
8 Fix Exploding Gradients with Gradient Clipping	167
8.1 Exploding Gradients and Clipping	167
8.2 Gradient Clipping Keras API	169
8.3 Gradient Clipping Case Study	170
8.4 Extensions	178
8.5 Further Reading	179

8.6 Summary	179
9 Accelerate Learning with Batch Normalization	180
9.1 Batch Normalization	180
9.2 Batch Normalization Keras API	186
9.3 Batch Normalization Case Study	189
9.4 Extensions	198
9.5 Further Reading	198
9.6 Summary	200
10 Deeper Models with Greedy Layer-Wise Pretraining	201
10.1 Greedy Layer-Wise Pretraining	201
10.2 Greedy Layer-Wise Pretraining Case Study	204
10.3 Extensions	219
10.4 Further Reading	219
10.5 Summary	220
11 Jump-Start Training with Transfer Learning	221
11.1 Transfer Learning	221
11.2 Transfer Learning Case Study	222
11.3 Extensions	240
11.4 Further Reading	240
11.5 Summary	241
II Better Generalization	243
12 Fix Overfitting with Regularization	245
12.1 Problem of Model Generalization and Overfitting	245
12.2 Reduce Overfitting by Constraining Complexity	247
12.3 Regularization Methods for Neural Networks	248
12.4 Regularization Recommendations	249
12.5 Further Reading	250
12.6 Summary	251
13 Penalize Large Weights with Weight Regularization	252
13.1 Weight Regularization	252
13.2 Weight Regularization Keras API	258
13.3 Weight Regularization Case Study	260
13.4 Extensions	268
13.5 Further Reading	269
13.6 Summary	270
14 Sparse Representations with Activity Regularization	272
14.1 Activity Regularization	272
14.2 Activity Regularization Keras API	276
14.3 Activity Regularization Case Study	278

14.4 Extensions	287
14.5 Further Reading	288
14.6 Summary	289
15 Force Small Weights with Weight Constraints	290
15.1 Weight Constraints	290
15.2 Weight Constraints Keras API	293
15.3 Weight Constraints Case Study	295
15.4 Extensions	302
15.5 Further Reading	303
15.6 Summary	304
16 Decouple Layers with Dropout	305
16.1 Dropout	305
16.2 Dropout Keras API	310
16.3 Dropout Case Study	314
16.4 Extensions	320
16.5 Further Reading	321
16.6 Summary	322
17 Promote Robustness with Noise	323
17.1 Noise Regularization	323
17.2 Noise Regularization Keras API	328
17.3 Noise Regularization Case Study	330
17.4 Extensions	341
17.5 Further Reading	341
17.6 Summary	343
18 Halt Training at the Right Time with Early Stopping	344
18.1 Early Stopping	344
18.2 Early Stopping Keras API	350
18.3 Early Stopping Case Study	353
18.4 Extensions	364
18.5 Further Reading	364
18.6 Summary	366
III Better Predictions	367
19 Reduce Model Variance with Ensemble Learning	369
19.1 High Variance of Neural Network Models	369
19.2 Reduce Variance Using an Ensemble of Models	370
19.3 How to Ensemble Neural Network Models	371
19.4 Summary of Ensemble Techniques	375
19.5 Further Reading	376
19.6 Summary	377

20 Combine Models From Multiple Runs with Model Averaging Ensemble	379
20.1 Model Averaging Ensemble	379
20.2 Ensembles in Keras	380
20.3 Model Averaging Ensemble Case Study	382
20.4 Extensions	396
20.5 Further Reading	396
20.6 Summary	397
21 Contribute Proportional to Trust with Weighted Average Ensemble	398
21.1 Weighted Average Ensemble	398
21.2 Weighted Average Ensemble Case Study	399
21.3 Extensions	417
21.4 Further Reading	417
21.5 Summary	419
22 Fit Models on Different Samples with Resampling Ensembles	420
22.1 Resampling Ensembles	420
22.2 Resampling Ensembles Case Study	422
22.3 Extensions	440
22.4 Further Reading	441
22.5 Summary	442
23 Models from Contiguous Epochs with Horizontal Voting Ensembles	443
23.1 Horizontal Voting Ensemble	443
23.2 Horizontal Voting Ensembles Case Study	444
23.3 Extensions	456
23.4 Further Reading	456
23.5 Summary	457
24 Cyclic Learning Rate and Snapshot Ensembles	458
24.1 Snapshot Ensembles	458
24.2 Snapshot Ensembles Case Study	460
24.3 Extensions	479
24.4 Further Reading	480
24.5 Summary	480
25 Learn to Combine Predictions with Stacked Generalization Ensemble	482
25.1 Stacked Generalization Ensemble	482
25.2 Stacked Generalization Ensemble Case Study	484
25.3 Extensions	500
25.4 Further Reading	500
25.5 Summary	502
26 Combine Model Parameters with Average Model Weights Ensemble	503
26.1 Average Model Weight Ensemble	503
26.2 Average Model Weight Ensemble Case Study	505
26.3 Extensions	523

26.4 Further Reading	523
26.5 Summary	524
IV Appendix	525
A Getting Help	526
A.1 Applied Neural Networks	526
A.2 Official Keras Destinations	526
A.3 Where to Get Help with Keras	527
A.4 How to Ask Questions	527
A.5 Contact the Author	528
B How to Setup Your Workstation	529
B.1 Overview	529
B.2 Download Anaconda	529
B.3 Install Anaconda	531
B.4 Start and Update Anaconda	533
B.5 Install Deep Learning Libraries	536
B.6 Further Reading	537
B.7 Summary	537
V Conclusions	538
How Far You Have Come	539

Preface

Modern open source libraries for developing neural network models are amazing. Gone are the days where we might spend weeks debugging the translation of poorly documented mathematics into code in the hopes of getting even the simplest model running. Today, using libraries like Keras, we can define and begin fitting a Multilayer Perceptron, Convolutional or even Recurrent Neural Network model of arbitrary complexity in minutes.

While defining and fitting models has become trivial, getting good performance with a neural network model on a specific problem remains challenging. Traditionally, configuring neural networks in order to get good performance was referred to as a *dark art*. This is because there are no clear rules on how to best prepare data and configure a model for a given problem. Instead, experience must be developed over time from working on many different projects. Nevertheless, neural networks have been used in academia and industry for decades now and there are a suite of standard techniques, tips, and configurations that you can use to greatly increase the likelihood of getting better-than-average performance with a neural network model.

I wrote this book to pull together the best classical and modern techniques in order to provide a playbook that you can use to get better performance on your next project using deep learning neural networks. A lot has changed in the last 5 to 10 years and there are activation functions, regularization methods, and even new ensemble methods that result in remarkably faster learning, lower generalization error, and more robust results. I hope that you're as excited as me about the journey ahead.

Jason Brownlee
2020

Introduction

Welcome

Welcome to *Better Deep Learning*. Deep learning neural networks have become easy to define and fit, but it remains challenging to achieve good predictive modeling performance. Neural networks have been studied in academia and used in industry for decades, and there is a wealth of techniques, tips, and model configurations that are known to result in better than average performance. In addition, the last 5 to 10 years has seen the development and adoption of modern network configurations, regularization techniques, and ensemble algorithms that result in superior performance. I designed this book to tie together classical and modern techniques into a single playbook and teach you step-by-step how to improve the performance of deep learning neural network models on your predictive modeling problems.

Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that know some applied machine learning and some deep learning. Maybe you want or need to start using deep learning on your research project or on a project at work. This guide was written to help you do that quickly and efficiently by compressing years of knowledge and experience into a laser-focused course of hands-on tutorials. The lessons in this book assume a few things about you, such as:

- You know your way around basic Python for programming.
- You know your way around basic NumPy for array manipulation.
- You know your way around basic Keras for deep learning.

This guide was written in the top-down and results-first machine learning style that you're used to from MachineLearningMastery.com.

About Your Outcomes

This book will teach you how to get results as a machine learning practitioner interested in getting the most out of deep learning models on your own predictive modeling projects. Techniques are demonstrated in the context of small well-understood predictive modeling problems and Multilayer Perceptron neural network models, but can easily be applied to a wide range of problems and with a suite of different types of neural network models. After reading and working through this book, you will know:

- A checklist of techniques that you can use to improve the performance of deep learning neural network models on your own predictive modeling problems.
- How to accelerate learning through better configured stochastic gradient descent batch size, loss functions, learning rates, and to avoid exploding gradients via gradient clipping.
- How to accelerate learning through correct data scaling, batch normalization, and use of modern activation functions such as the rectified linear activation function.
- How to accelerate learning through choosing better initial weights with greedy layer-wise pretraining and transfer learning.
- A gentle introduction to the problem of overfitting and a tour of regularization techniques.
- How to reduce overfitting by updating the loss function using techniques such as weight regularization, weight constraints, and activation regularization.
- How to reduce overfitting using techniques such as dropout, the addition of noise, and early stopping.
- A gentle introduction to how to combine the predictions from multiple models and a tour of ensemble learning techniques.
- How to combine the predictions from multiple different models using techniques such as weighted averaging ensembles and stacked generalization ensembles, also known as *blending*.
- How to combine the predictions from multiple models saved during a single training run with techniques such as horizontal ensembles and snapshot ensembles.

This new understanding of applied deep learning methods will impact your practice of working through predictive modeling problems in the following ways:

- Confidently diagnose poor model training and problems such as premature convergence and accelerate the model training process using one or a combination of modifications to the learning algorithm.
- Confidently diagnose cases of overfitting the training dataset and reduce generalization error using one or a combination of modifications of the model, loss function, or learning algorithm.
- Confidently diagnose high variance in a final model and improve the average predictive skill by combining the predictions from multiple models trained over a single or multiple training runs.

This book will teach you how to get good results, quickly, but will *NOT* teach you how to be a research scientist nor will it teach you all the theory behind why specific methods work. For that, I would recommend good research papers and textbooks. See the *Further Reading* section at the end of each tutorial for a good starting point.

How to Read This Book

This book was written to be read linearly, from start to finish. That being said, if you know the basics and need help with a specific problem type or technique, then you can flip straight to that section and get started. This book was designed for you to read on your workstation, on the screen. Not away from the computer or on a tablet or eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.

This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then apply your new understanding to your own predictive modeling projects. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them. Try some of the extensions presented at the end of each lesson and let me know how you do.

About the Book Structure

This book was designed around three main activities for getting better results with deep learning models: better or faster learning, better generalization to new data, and better predictions when using final models. There are a lot of things you could learn about getting better results from neural network models, from theory to applications to APIs. My goal is to take you straight to getting results with laser-focused tutorials. I designed the tutorials to focus on how to get things done. They give you the tools to both rapidly understand and apply each technique to your own predictive modeling problems.

Each of the tutorials are designed to take you about one hour to read through and complete, excluding the extensions and further reading. You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and this book is intended to be read and used, not to sit idle. I would recommend picking a schedule and sticking to it. The tutorials are divided into three parts:

- **Part 1: Better Learning.** Discover the techniques to improve and accelerate the process used to learn or optimize the weights of a neural network model.
- **Part 2: Better Generalization.** Discover the techniques to reduce overfitting of the training dataset and improve the generalization of models on new data.
- **Part 3: Better Predictions.** Discover the techniques to improve the performance of final models when used to make predictions on new data.

Each part targets a specific approach to improving model performance, and each tutorial targets a specific learning outcome for a technique. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and do not get bogged down in the math or near-infinite number of configuration parameters. The tutorials were not designed to teach you everything there is to know about each of the techniques. They were designed to give you an understanding of how they work and how to use them on your projects the fastest way I know how: to learn by doing.

About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. For this reason, the examples are highly targeted.

- Models were demonstrated on small contrived datasets to give you the context and confidence to bring the techniques to your own projects.
- Model configurations used were discovered through trial and error and are skillful but not optimized. This leaves the door open for you to explore new and possibly better configurations.
- Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third parties required beyond the installation of the required packages.

A complete working example is presented with each tutorial for you to inspect and copy and paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues. The provided code was developed in a text editor and intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead.

Neural network algorithms are stochastic. This means that they will make different predictions when the same model configuration is run on the same training data. On top of that, each experimental problem in this book is based around generating stochastic predictions. As a result, this means you will not get exactly the same sample output presented in this book. This is by design. I want you to get used to the stochastic nature of the neural network algorithms. If this bothers you, please note:

- You can re-run a given example a few times and your results should be close to the values reported.
- You can make the output consistent by fixing the random number seed.
- You can develop a robust estimate of the skill of a model by fitting and evaluating it multiple times and taking the average of the final skill score (highly recommended).

All code examples were tested on a POSIX-compatible machine with Python 3 and Keras 2. All code examples will run on modest and modern computer hardware and were executed on a CPU. No GPUs are required to run the presented examples, although a GPU would make the code run faster. I am only human and there may be a bug in the sample code. If you discover a bug, please let me know so I can fix it and update the book and send out a free update.

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Research papers.

- Books and book chapters.
- Webpages.
- API documentation.

Wherever possible, I try to list and link to the relevant API documentation for key objects and functions used in each lesson so you can learn more about them. When it comes to research papers, I try to list papers that are first to use a specific technique or first in a specific problem domain. These are not required reading, but can give you more technical details, theory, and configuration details if you're looking for it. Wherever possible, I have tried to link to the freely available version of the paper on *arxiv.org*. You can search for and download any of the papers listed on Google Scholar Search, *scholar.google.com*. Wherever possible, I have tried to link to books on Amazon.

I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

About Getting Help

You might need help along the way. Don't worry; you are not alone.

- **Help with a Technique?** If you need help with the technical aspects of a specific model or method, see the *Further Reading* sections at the end of each lesson.
- **Help with Keras?** If you need help with using the Keras library, see the list of resources in *Appendix A*.
- **Help with your workstation?** If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in *Appendix B*.
- **Help in general?** You can shoot me an email. My details are in *Appendix A*.

Next

Are you ready? Let's dive in! In the next tutorial, you will discover a framework that you can use to diagnose problems with your deep learning neural network and techniques that you can use to address each identified problem.

Framework for Better Deep Learning

Modern deep learning libraries such as Keras allow you to define and start fitting a wide range of neural network models in minutes with just a few lines of code. Nevertheless, it is still challenging to configure a neural network to get good performance on a new predictive modeling problem. The challenge of getting good performance can be broken down into three main areas: problems with learning, problems with generalization, and problems with predictions. Once you have diagnosed the specific type of problem that you are having with a network, a suite of classical and modern techniques can then be selected to address the issue and improve performance. In this tutorial, you will discover a framework for diagnosing performance problems with deep learning models and techniques that you can use to target and improve each specific performance problem. After reading this tutorial, you will know:

- Defining and fitting neural networks has never been easier, although getting good performance on new problems remains challenging.
- Neural network model performance problems can be decomposed into learning, generalization, and prediction type problems.
- There are decades of techniques as well as modern methods that can be used to target each type of model performance problem.

Let's get started.

Overview

This tutorial is divided into seven parts; they are:

1. Neural Network Renaissance
2. Challenge of Configuring Neural Networks
3. Framework for Systematically Better Deep Learning
4. Better Learning Techniques
5. Better Generalization Techniques
6. Better Prediction Techniques
7. How to Use the Framework

Neural Network Renaissance

Historically, neural network models had to be coded from scratch. You might spend days or weeks translating poorly described mathematics into code and days or weeks more debugging your code just to get a simple neural network model to run. Those days are in the past. Today, you can define and begin fitting most types of neural networks in minutes with just a few lines of code, thanks to open source libraries such as Keras built on top of sophisticated mathematical libraries such as TensorFlow.

This means that standard models such as Multilayer Perceptrons can be developed and evaluated rapidly, as well as more sophisticated models that may previously have been beyond the capabilities of most practitioners to implement such as Convolutional Neural Networks and Recurrent Neural Networks like the Long Short-Term Memory network. As deep learning practitioners, we live in amazing and productive times. Nevertheless, even through new neural network models can be defined and evaluated rapidly, there remains little guidance on how to actually configure neural network models in order to get the most out of them.

Challenge of Configuring Neural Networks

Configuring neural network models is often referred to as a *dark art*. This is because there are no hard and fast rules for configuring a network for a given problem. We cannot analytically calculate the optimal model type or model configuration for a given dataset. Instead, there are decades worth of techniques, heuristics, tips, tricks, and other tacit knowledge spread across code, papers, blog posts, and in peoples' heads. A shortcut to configuring a neural network on a problem is to copy the configuration of another network for a similar problem. But this strategy rarely leads to good results as model configurations are not transferable across problems. It is also likely that you work on predictive modeling problems that are unlike other problems described in the literature.

Fortunately, there are techniques that are known to address specific issues when configuring and training a neural network that are available in modern deep learning libraries like Keras. Further, discoveries have been made in the past 5 to 10 years in areas such as activation functions, adaptive learning rates, regularization methods, and ensemble techniques that have been shown to dramatically improve the performance of neural network models regardless of their specific type. The techniques are available; you just need to know what they are and when to use them.

Framework for Systematically Better Deep Learning

Unfortunately, you cannot efficiently grid search across the techniques used to improve deep learning performance. Almost universally, they uniquely change aspects of the training data, learning process, model architecture, and more. Instead, you must diagnose the type of performance problem you are having with your model, then carefully choose and evaluate a given intervention tailored to that diagnosed problem. There are three types of problems that are straightforward to diagnose with regard to poor performance of a deep learning neural network model; they are:

- **Problems with Learning.** Problems with learning manifest in a model that cannot effectively learn a training dataset or shows slow progress or bad performance when

learning the training dataset.

- **Problems with Generalization.** Problems with generalization manifest in a model that overfits the training dataset and makes poor predictions on a holdout dataset.
- **Problems with Predictions.** Problems with predictions manifest in the stochastic training algorithm having a strong influence on the final model, causing high variance in behavior and performance.

This framework provides a systematic approach to thinking about the performance of your deep learning model. There is some natural overlap and interaction between these areas of concern. For example, problems with learning affect the ability of the model to generalize as well as the variance in the predictions made from a final model. The sequential relationship between the three areas in the proposed framework allows the issue of deep learning model performance to be first isolated, then targeted with a specific technique or methodology. We can summarize techniques that assist with each of these problems as follows:

- **Better Learning.** Techniques that improve or accelerate the adaptation of neural network model weights in response to a training dataset.
- **Better Generalization.** Techniques that improve the performance of a neural network model on a holdout dataset.
- **Better Predictions.** Techniques that reduce the variance in the performance of a final model.

Now that we have a framework for systematically diagnosing a performance problem with a deep learning neural network, let's take a look at some examples of techniques that may be used in each of these three areas of concern.

Better Learning Techniques

Better learning techniques are those changes to a neural network model or learning algorithm that improve or accelerate the adaptation of the model weights in response to a training dataset (Chapter 1). In this section, we will review the techniques used to improve the adaptation of the model weights. This begins with the careful configuration of the capacity of the model and the hyperparameters related to optimizing the neural network model using the stochastic gradient descent algorithm and updating the weights using the backpropagation of error algorithm; for example:

- **Configure Capacity.** Including the number of nodes in a layer and the number of layers used to define the scope of functions that can be learned by the model (Chapter 2).
- **Configure Batch Size.** Including exploring whether variations such as batch, stochastic (online), or minibatch gradient descent are more appropriate (Chapter 3).
- **Configure Loss Function.** Including understanding the way different loss functions must be interpreted and whether an alternate loss function would be appropriate for your problem (Chapter 4).

- **Configure Learning Rate.** Including understanding the effect of different learning rates on your problem and whether modern adaptive learning rate methods such as Adam would be appropriate (Chapter 5).

This also includes simple data preparation and the automatic rescaling of inputs at deeper layers.

- **Data Scaling Techniques.** Including the sensitivity that small network weights have to the scale of input variables and the impact of large errors in the target variable have on weight updates (Chapter 6).
- **Batch Normalization.** Including the sensitivity to changes in the distribution of inputs to layers deep in a network model and the benefits of standardizing layer inputs to add consistency of input and stability to the learning process (Chapter 9).

Stochastic gradient descent is a general optimization algorithm that can be applied to a wide range of problems. Nevertheless, the optimization process (or learning process) can become unstable and specific interventions are required; for example:

- **Vanishing Gradients.** Prevent the training of deep multiple-layered networks causing layers close to the input layer to not have their weights updated; that can be addressed using modern activation functions such as the rectified linear activation function (Chapter 7).
- **Exploding Gradients.** Large weight updates cause a numerical overflow or underflow making the network weights take on a NaN or Inf value; that can be addressed using gradient scaling or gradient clipping (Chapter 8).

The lack of data on some predictive modeling problems can prevent effective learning. Specialized techniques can be used to jump-start the optimization process, providing a useful initial set of weights or even whole models that can be used for feature extraction; for example:

- **Greedy Layer-Wise Pretraining.** Where layers are added one at a time to a model, learning to interpret the output of prior layers and permitting the development of much deeper models: a milestone technique in the field of deep learning (Chapter 10).
- **Transfer Learning.** Where a model is trained on a different, but somehow related, predictive modeling problem and then used to seed the weights or used wholesale as a feature extraction model to provide input to a model trained on the problem of interest (Chapter 11).

Better Generalization Techniques

Better generalization techniques are those that change the neural network model or learning algorithm to reduce the effect of the model overfitting the training dataset and improve the performance of the model on a holdout validation or test dataset (Chapter 12). In this section, we will review the techniques to reduce the generalization error of the model during training. Techniques that are designed to reduce generalization error are commonly referred to

as *regularization techniques*. Almost universally, regularization is achieved by somehow reducing or limiting model complexity.

Perhaps the most widely understood measure of model complexity is the size or magnitude of the model weights. A model with large weights is a sign that it may be overly specialized to the inputs in the training data, making it unstable when used when making a prediction on new unseen data. Keeping weights small via weight regularization is a powerful and widely used technique.

- **Weight Regularization.** A change to the loss function that penalizes a model in proportion to the norm (magnitude) of the model weights, encouraging smaller weights and, in turn, a lower complexity model (Chapter 13).

Rather than simply encouraging the weights to remain small via an updated loss function, it is possible to force the weights to be small using a constraint.

- **Weight Constraint.** Update to the model to rescale the weights when the vector norm of the weights exceeds a threshold (Chapter 15).

The output of a neural network layer, regardless of where that layer is in the stack of layers, can be thought of as an internal representation or set of extracted features with regard to the input. Simpler internal representations can have a regularizing effect on the model and can be encouraged through constraints that encourage sparsity (zero values).

- **Activity Regularization.** A change to the loss function that penalizes a model in proportion to the norm (magnitude) of the layer activations, encouraging smaller or more sparse internal representations (Chapter 14).

Noise can be added to the model to encourage robustness with regard to the raw inputs or outputs from prior layers during training; for example:

- **Dropout.** Probabilistically removing connections (weights) while training the network to break tight coupling between nodes across layers (Chapter 16).
- **Input Noise.** Addition of statistical variation or noise at the input layer or between hidden layers to reduce the model's dependence on specific input values (Chapter 17).

Often, overfitting can occur due simply to training the model for too long on the training dataset. A simple solution is to stop the training early.

- **Early Stopping.** Monitor model performance on the hold out validation dataset during training and stop the training process when performance on the validation set starts to degrade (Chapter 18).

Better Prediction Techniques

Better prediction techniques are those that complement the model training process in order to reduce the variance in the expected performance of the final model (Chapter 19). In this section, we will review the techniques to reduce the expected variance of a final deep learning neural network model. The variance in the performance of the final model can be reduced by adding bias. The most common way to introduce bias to the final model is to combine the predictions from multiple models. This is referred to as ensemble learning.

More than reducing the variance of the performance of a final model, ensemble learning can also result in better predictive performance. Effective ensemble learning methods require that each contributing model have skill, meaning that the models make predictions that are better than random, but that the prediction errors between the models have a low correlation. This means, that the ensemble member models should have skill, but in different ways. This can be achieved by varying one aspect of the ensemble; for example:

- Vary the training data used to fit each member.
- Vary the members that contribute to the ensemble prediction.
- Vary the way that the predictions from the ensemble members are combined.

The training data can be varied by fitting models on different subsamples of the dataset. This might involve fitting and retaining models on different randomly selected subsets of the training dataset, retaining models for each fold in a k -fold cross-validation, or retaining models across different samples with replacement using the bootstrap method (e.g. bootstrap aggregation). Collectively, we can think of these methods as resampling ensembles.

- **Resampling Ensemble.** Ensemble of models fit on different samples of the training dataset (Chapter 22).

Perhaps the simplest way to vary the members of the ensemble involves gathering models from multiple runs of the learning algorithm on the training dataset. The stochastic learning algorithm will cause a slightly different fit on each run that, in turn, will have a slightly different fit. Averaging the models across multiple runs will ensure the performance remains consistent.

- **Model Averaging Ensemble.** Retrain models across multiple runs of the same learning algorithm on the same dataset (Chapter 20).

Variations on this approach may involve training models with different hyperparameter configurations. It can be expensive to train multiple final deep learning models, especially when one model may take days or weeks to fit. An alternative is to collect models for use as contributing ensemble members during a single training run; for example:

- **Horizontal Ensemble.** Ensemble members collected from a contiguous block of training epochs towards the end of a single training run (Chapter 23).
- **Snapshot Ensemble.** A training run using an aggressive cyclic learning rate where ensemble members are collected at the trough of each cycle of the learning rate (Chapter 24).

The simplest way to combine the predictions from multiple ensemble members is to calculate the average of the predictions in the case of regression, or the statistical mode (most frequent prediction) in the case of classification. Alternately, the best way to combine the predictions from multiple models can be learned; for example:

- **Weighted Average Ensemble (blending).** The contribution from each ensemble member to an ensemble prediction is weighted using learned coefficients that indicates the trust in each model (Chapter 21).
- **Stacked Generalization (stacking).** A new model is trained to learn how to best combine the predictions from the ensemble members (Chapter 25).

An alternative to combining the predictions from the ensemble members, the models themselves may be combined; for example:

- **Average Model Weight Ensemble.** Weights from multiple neural network models are averaged into a single model used to make a prediction (Chapter 26).

How to Use the Framework

We can think of the organization of techniques into the three areas of better learning, generalization, and prediction as a systematic framework for improving the performance of your neural network model. There are too many techniques to reasonably investigate and evaluate each in your project. Instead, you need to be methodical and use the techniques in a targeted way to address a defined problem.

Step 1: Diagnose the Performance Problem

The first step in using this framework is to diagnose the performance problem that you are having with your model. A robust diagnostic tool is to calculate a learning curve of loss and a problem-specific metric (like RMSE for regression or accuracy for classification) on a train and validation dataset over a given number of training epochs.

- If the loss on the training dataset is poor, stuck, or fails to improve, perhaps you have a learning problem.
- If the loss or problem-specific metric on the training dataset continues to improve and gets worse on the validation dataset, perhaps you have a generalization problem.
- If the loss or problem-specific metric on the validation dataset shows a high variance towards the end of the run, perhaps you have a prediction problem.

Step 2: Select and Evaluate a Technique

Review the techniques that are designed to address your problem. Select a technique that appears to be a good fit for your model and problem. This may require some prior experience with the techniques and may be challenging for a beginner. Thankfully, there are heuristics and best-practices that work well on most problems. For example:

- **Learning Problem:** Tuning the hyperparameters of the learning algorithm; specifically, the learning rate offers the biggest leverage.
- **Generalization Problem:** Using weight regularization and early stopping works well on most models with most problems, or try dropout with early stopping.
- **Prediction Problem:** Average the prediction from models collected over multiple runs or multiple epochs on one run to add sufficient bias.

Pick an intervention, then read-up a little bit on it, including how it works, why it works, and importantly, find examples for how practitioners before you have used it to get an idea for how you might use it on your problem.

Step 3: Go To Step 1

Once you have identified an issue and addressed it with an intervention, repeat the process. Developing a better model is an iterative process that may require multiple interventions at multiple levels that complement each other. This is an empirical process. This means that you are reliant on the robustness of your test harness to give you a reliable summary of performance before and after an intervention. Spend the time to ensure your test harness is robust, guarantee that the train, test, and validation datasets are clean and provide a suitably representative sample of observation from your problem domain.

Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Books

- *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- *Pattern Recognition and Machine Learning*, 2006.
<https://amzn.to/2Q2rEeP>
- *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
<https://amzn.to/2poq0xc>

Papers

- *Practical Recommendations for Gradient-Based Training of Deep Architectures*, 2012.
<https://arxiv.org/abs/1206.5533>

Articles

- Neural Network FAQ.
<ftp://ftp.sas.com/pub/neural/FAQ.html>

Summary

In this tutorial, you discovered a framework for diagnosing performance problems with deep learning models and techniques that you can use to target and improve each specific performance problem. Specifically, you learned:

- Defining and fitting neural networks has never been easier, although getting good performance on new problems remains challenging.
- Neural network model performance problems can be decomposed into learning, generalization, and prediction type problems.
- There are decades of techniques as well as modern methods that can be used to target each type of model performance problem.

Next

In the next tutorial you will discover how to use learning curves to diagnose mode behavior and examples of common model dynamics.

Diagnostic Learning Curves

A learning curve is a plot of model learning performance over experience or time. Learning curves are a widely used diagnostic tool in machine learning for algorithms that learn from a training dataset incrementally. The model can be evaluated on the training dataset and on a hold out validation dataset after each update during training and plots of the measured performance can be created to show learning curves. Reviewing learning curves of models during training can be used to diagnose problems with learning, such as an underfit or overfit model, as well as whether the training and validation datasets are suitably representative. In this tutorial, you will discover learning curves and how they can be used to diagnose the learning and generalization behavior of machine learning models, with example plots showing common learning problems. After reading this tutorial, you will know:

- Learning curves are plots that show changes in learning performance over time in terms of experience.
- Learning curves of model performance on the train and validation datasets can be used to diagnose an underfit, overfit, or well-fit model.
- Learning curves of model performance can be used to diagnose whether the train or validation datasets are not relatively representative of the problem domain.

Let's get started.

Overview

This tutorial is divided into three parts; they are:

1. Learning Curves
2. Diagnosing Model Behavior
3. Diagnosing Unrepresentative Datasets

Learning Curves in Machine Learning

Generally, a learning curve is a plot that shows time or experience on the x -axis and learning or improvement on the y -axis.

Learning curves (LCs) are deemed effective tools for monitoring the performance of workers exposed to a new task. LCs provide a mathematical representation of the learning process that takes place as task repetition occurs.

— Learning curve models and applications: Literature review and research directions, 2011.

For example, if you were learning a musical instrument, your skill on the instrument could be evaluated and assigned a numerical score each week for one year. A plot of the scores over the 52 weeks is a learning curve and would show how your learning of the instrument has changed over time.

- **Learning Curve:** Line plot of learning (y -axis) over experience (x -axis).

Learning curves are widely used in machine learning for algorithms that learn (optimize their internal parameters) incrementally over time, such as deep learning neural networks. The metric used to evaluate learning could be maximizing, meaning that better scores (larger numbers) indicate more learning. An example would be classification accuracy. It is more common to use a score that is minimizing, such as loss or error whereby better scores (smaller numbers) indicate more learning and a value of 0.0 indicates that the training dataset was learned perfectly and no mistakes were made. During the training of a machine learning model, the current state of the model at each step of the training algorithm can be evaluated. It can be evaluated on the training dataset to give an idea of how well the model is *learning*. It can also be evaluated on a hold-out validation dataset that is not part of the training dataset. Evaluation on the validation dataset gives an idea of how well the model is *generalizing*.

- **Train Learning Curve:** Learning curve calculated from the training dataset that gives an idea of how well the model is learning.
- **Validation Learning Curve:** Learning curve calculated from a hold-out validation dataset that gives an idea of how well the model is generalizing.

It is common to create dual learning curves for a machine learning model during training on both the training and validation datasets. In some cases, it is also common to create learning curves for multiple metrics, such as in the case of classification predictive modeling problems, where the model may be optimized according to cross-entropy loss and model performance is evaluated using classification accuracy. In this case, two plots are created, one for the learning curves of each metric, and each plot can show two learning curves, one for each of the train and validation datasets.

- **Optimization Learning Curves:** Learning curves calculated on the metric by which the parameters of the model are being optimized, e.g. loss.
- **Performance Learning Curves:** Learning curves calculated on the metric by which the model will be evaluated and selected, e.g. accuracy.

Now that we are familiar with the use of learning curves in machine learning, let's look at some common shapes observed in learning curve plots.

Diagnosing Model Behavior

The shape and dynamics of a learning curve can be used to diagnose the behavior of a machine learning model and in turn perhaps suggest at the type of configuration changes that may be made to improve learning and/or performance. There are three common dynamics that you are likely to observe in learning curves; they are:

- Underfit.
- Overfit.
- Good Fit.

We will take a closer look at each with examples. The examples will assume that we are looking at a minimizing metric, meaning that smaller relative scores on the y -axis indicate more or better learning.

Underfit Learning Curves

Underfitting refers to a model that cannot learn the training dataset.

Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set.

— Page 111, *Deep Learning*, 2016.

An underfit model can be identified from the learning curve of the training loss only. It may show a flat line or noisy values of relatively high loss, indicating that the model was unable to learn the training dataset at all. An example of this is provided below and is common when the model does not have a suitable capacity for the complexity of the dataset.

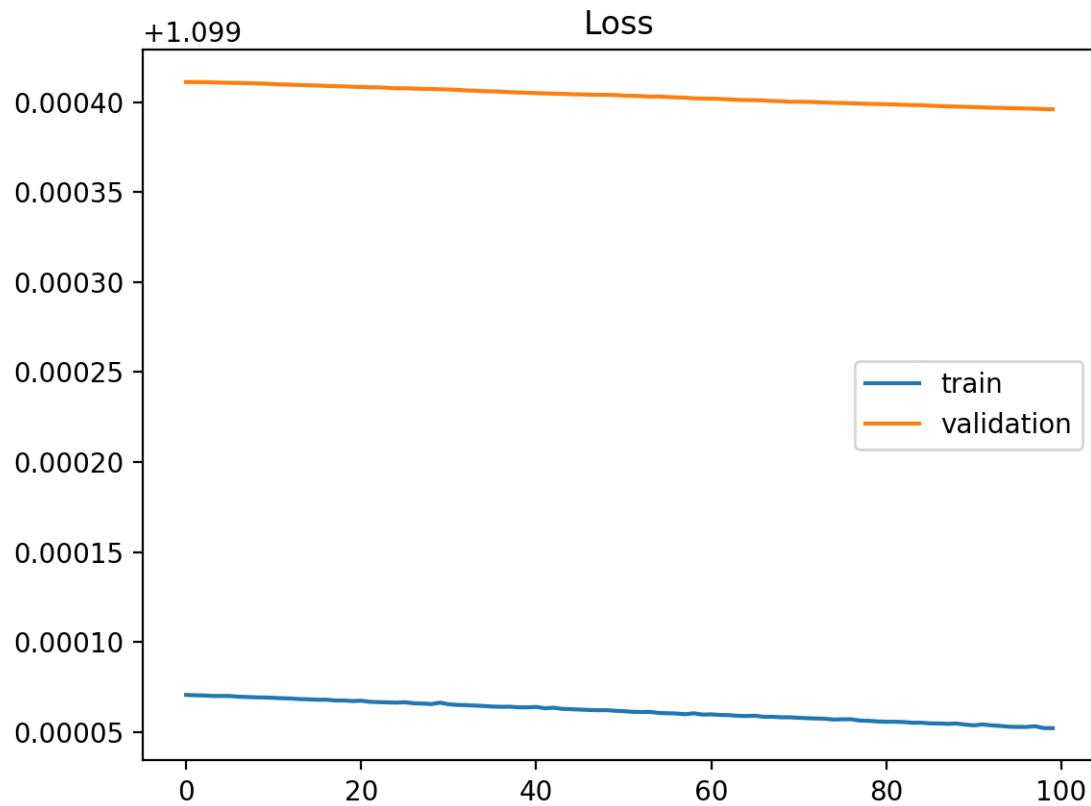


Figure 1: Example of Training Learning Curve Showing An Underfit Model That Does Not Have Sufficient Capacity.

An underfit model may also be identified by a training loss that is decreasing and continues to decrease at the end of the plot. This indicates that the model is capable of further learning and possible further improvements and that the training process was halted prematurely.

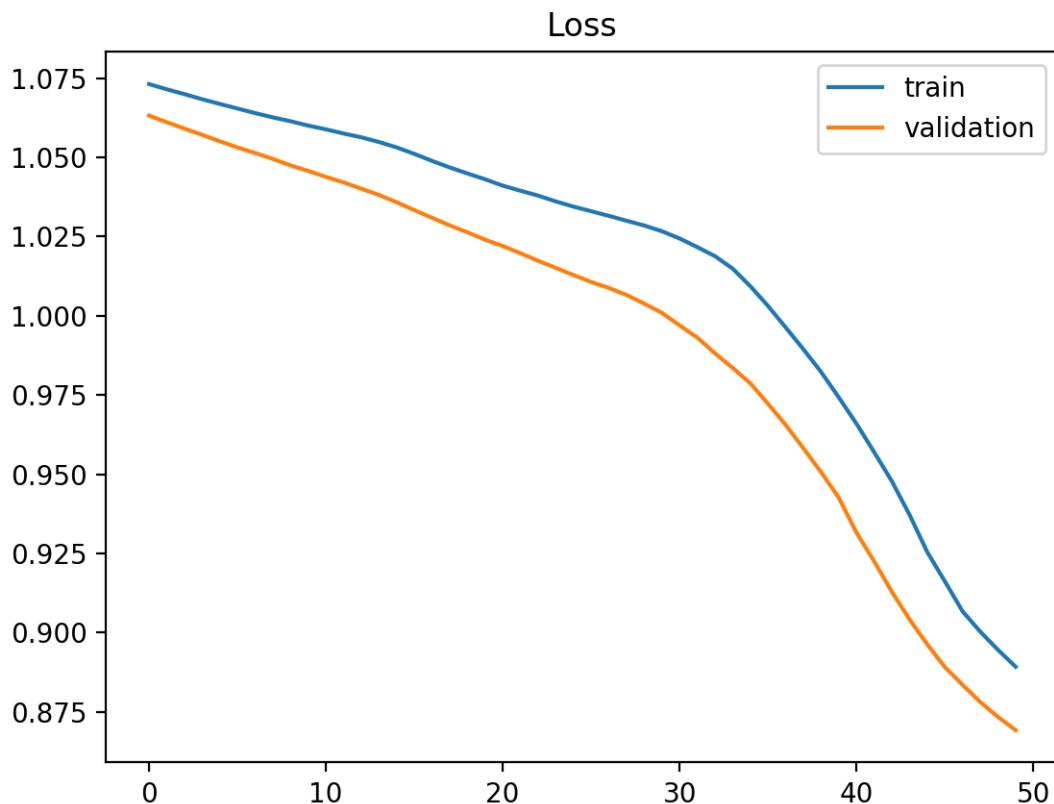


Figure 2: Example of Training Learning Curve Showing an Underfit Model That Requires Further Training.

A plot of learning curves shows underfitting if:

- The training loss remains flat regardless of training.
- The training loss continues to decrease until the end of training.

Overfit Learning Curves

Overfitting refers to a model that has learned the training dataset too well, including the statistical noise or random fluctuations in the training dataset.

... fitting a more flexible model requires estimating a greater number of parameters. These more complex models can lead to a phenomenon known as overfitting the data, which essentially means they follow the errors, or noise, too closely.

— Page 22, *An Introduction to Statistical Learning: with Applications in R*, 2013.

The problem with overfitting, is that the more specialized the model becomes to training data, the less well it is able to generalize to new data, resulting in an increase in generalization error. This increase in generalization error can be measured by the performance of the model on the validation dataset.

This is an example of overfitting the data, [...]. It is an undesirable situation because the fit obtained will not yield accurate estimates of the response on new observations that were not part of the original training data set.

— Page 24, *An Introduction to Statistical Learning: with Applications in R*, 2013.

This often occurs if the model has more capacity than is required for the problem, and, in turn, too much flexibility. It can also occur if the model is trained for too long. A plot of learning curves shows overfitting if:

- The plot of training loss continues to decrease with experience.
- The plot of validation loss decreases to a point and begins increasing again.

The inflection point in validation loss may be the point at which training could be halted as experience after that point shows the dynamics of overfitting. The example plot below demonstrates a case of overfitting.

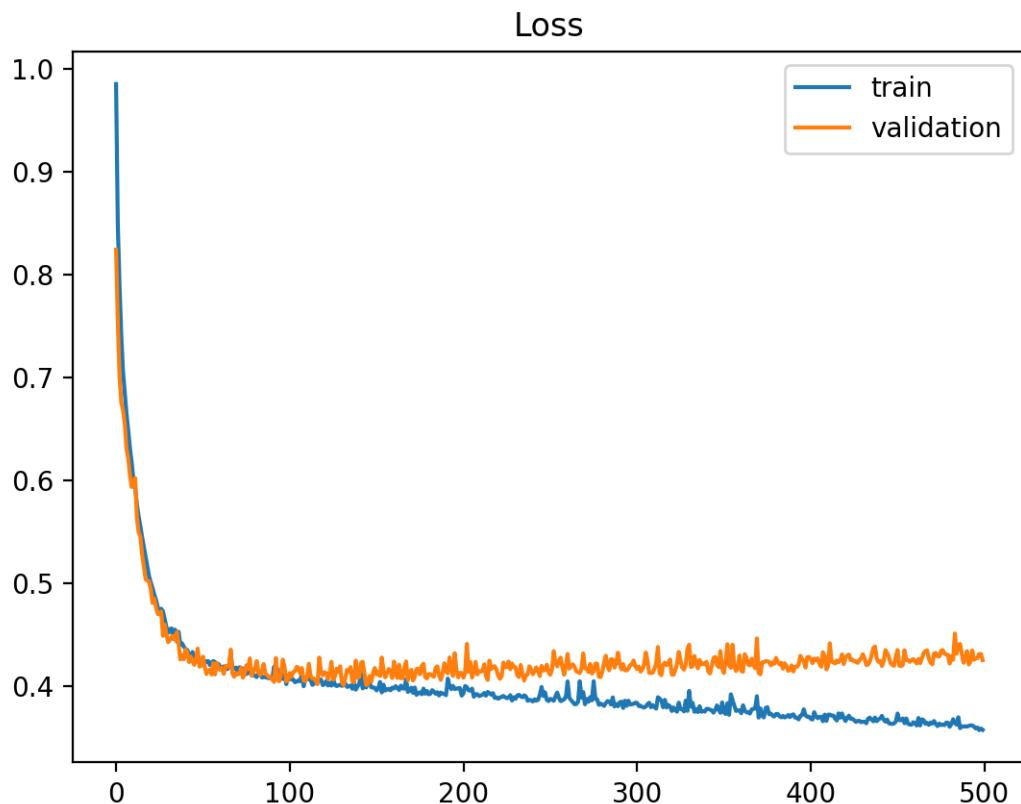


Figure 3: Example of Train and Validation Learning Curves Showing an Overfit Model.

Good Fit Learning Curves

A good fit is the goal of the learning algorithm and exists between an overfit and underfit model. A good fit is identified by a training and validation loss that decreases to a point of stability with a minimal gap between the two final loss values. The loss of the model will almost always be lower on the training dataset than the validation dataset. This means that we should expect some gap between the train and validation loss learning curves. This gap is referred to as the *generalization gap*. A plot of learning curves shows a good fit if:

- The plot of training loss decreases to a point of stability.
- The plot of validation loss decreases to a point of stability and has a small gap with the training loss.

Continued training of a good fit will likely lead to an overfit. The example plot below demonstrates a case of a good fit.

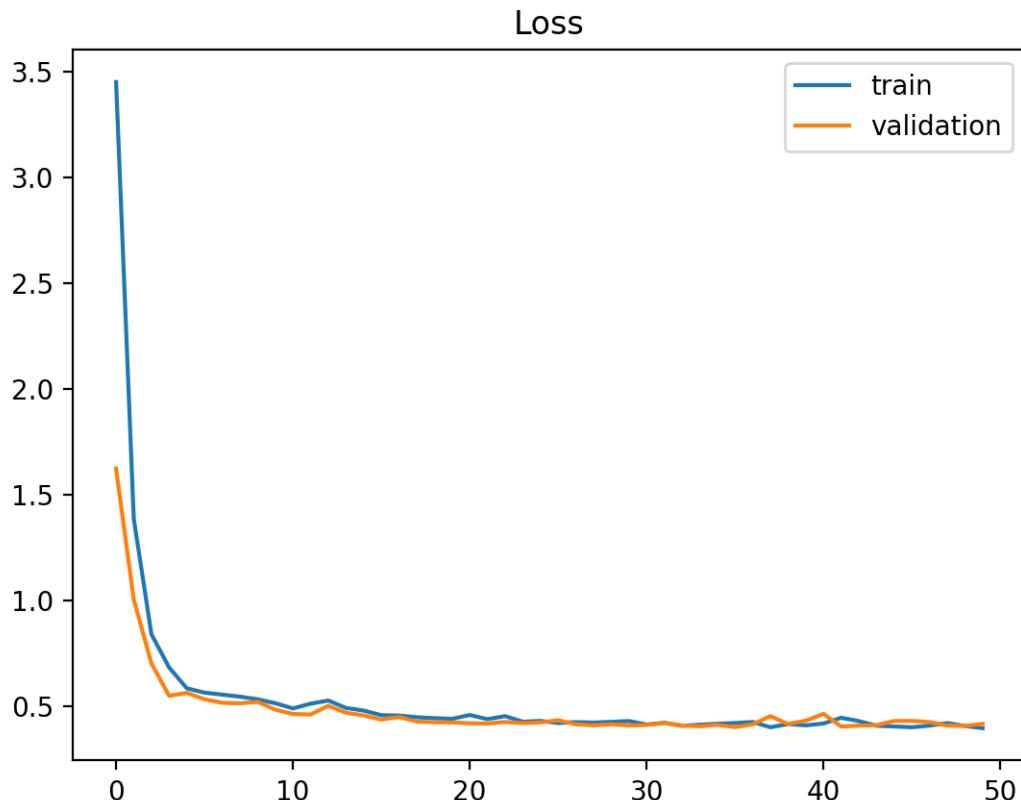


Figure 4: Example of Train and Validation Learning Curves Showing a Good Fit.

Diagnosing Unrepresentative Datasets

Learning curves can also be used to diagnose properties of a dataset and whether it is relatively representative. An unrepresentative dataset means a dataset that may not capture the statistical

characteristics relative to another dataset drawn from the same domain, such as between a train and a validation dataset. This can commonly occur if the number of samples in a dataset is too small, relative to another dataset.

There are two common cases that could be observed; they are:

- Training dataset is relatively unrepresentative.
- Validation dataset is relatively unrepresentative.

Unrepresentative Train Dataset

An unrepresentative training dataset means that the training dataset does not provide sufficient information to learn the problem, relative to the validation dataset used to evaluate it. This may occur if the training dataset has too few examples as compared to the validation dataset. This situation can be identified by a learning curve for training loss that shows improvement and similarly a learning curve for validation loss that shows improvement, but a large gap remains between both curves.

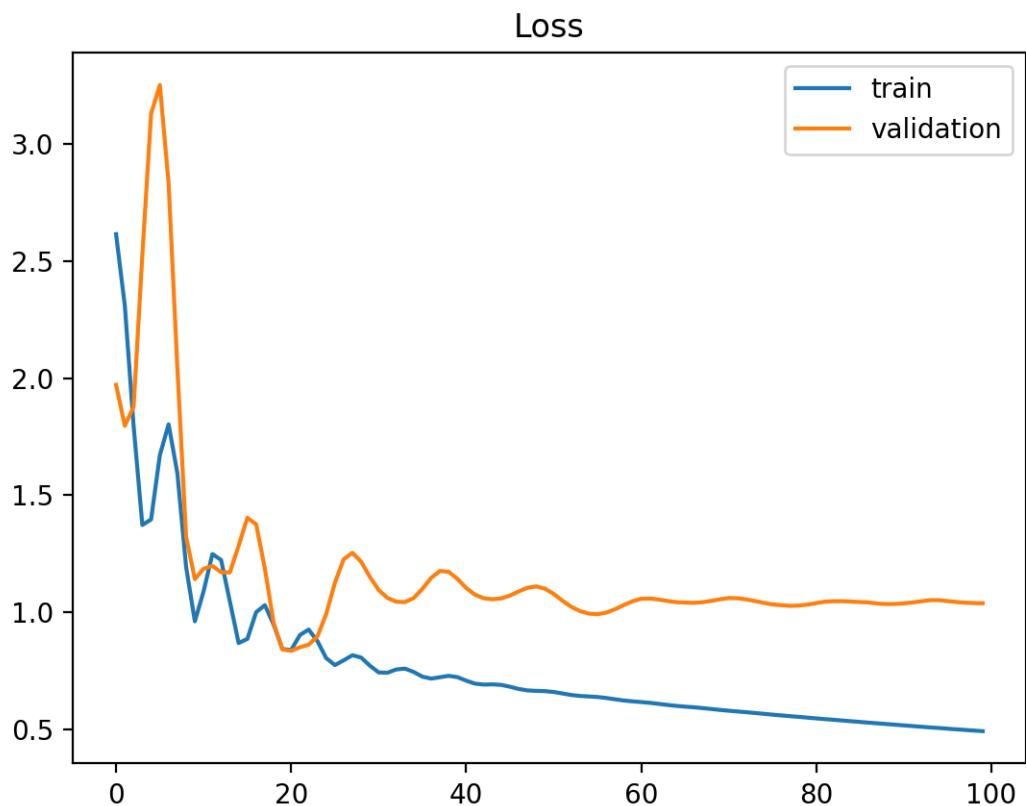


Figure 5: Example of Train and Validation Learning Curves Showing a Training Dataset That May Be too Small Relative to the Validation Dataset.

Unrepresentative Validation Dataset

An unrepresentative validation dataset means that the validation dataset does not provide sufficient information to evaluate the ability of the model to generalize. This may occur if the validation dataset has too few examples as compared to the training dataset. This case can be identified by a learning curve for training loss that looks like a good fit (or other fits) and a learning curve for validation loss that shows noisy movements around the training loss.

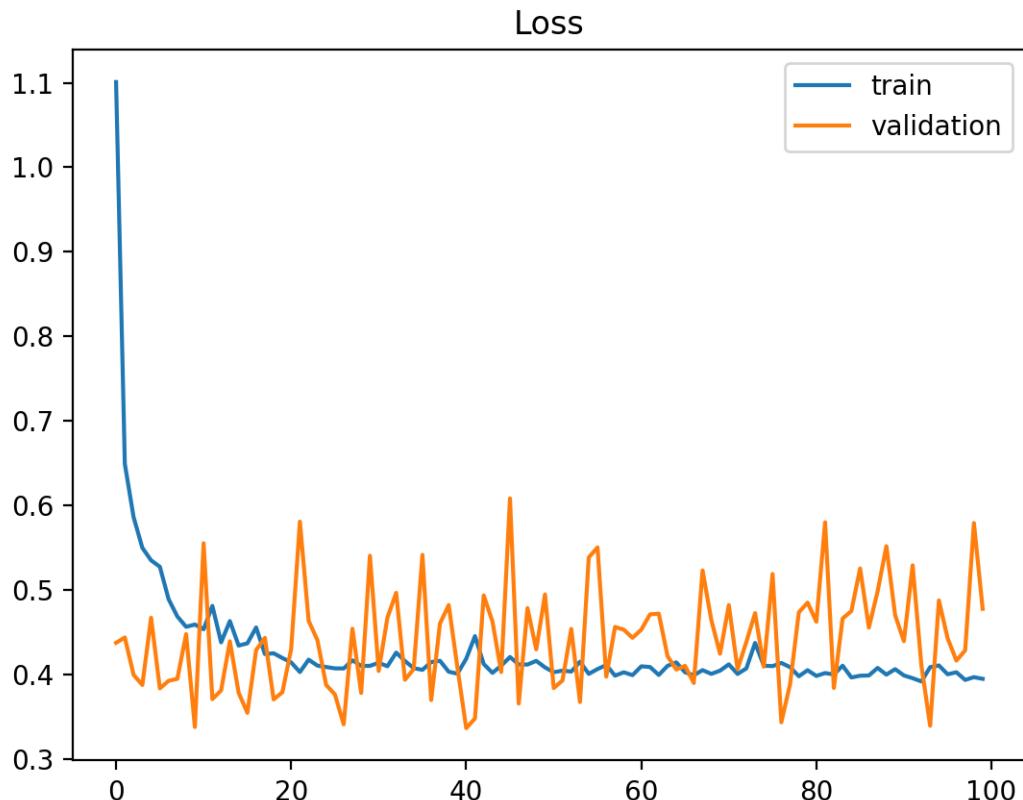


Figure 6: Example of Train and Validation Learning Curves Showing a Validation Dataset That May Be too Small Relative to the Training Dataset.

It may also be identified by a validation loss that is lower than the training loss. In this case, it indicates that the validation dataset may be easier for the model to predict than the training dataset.

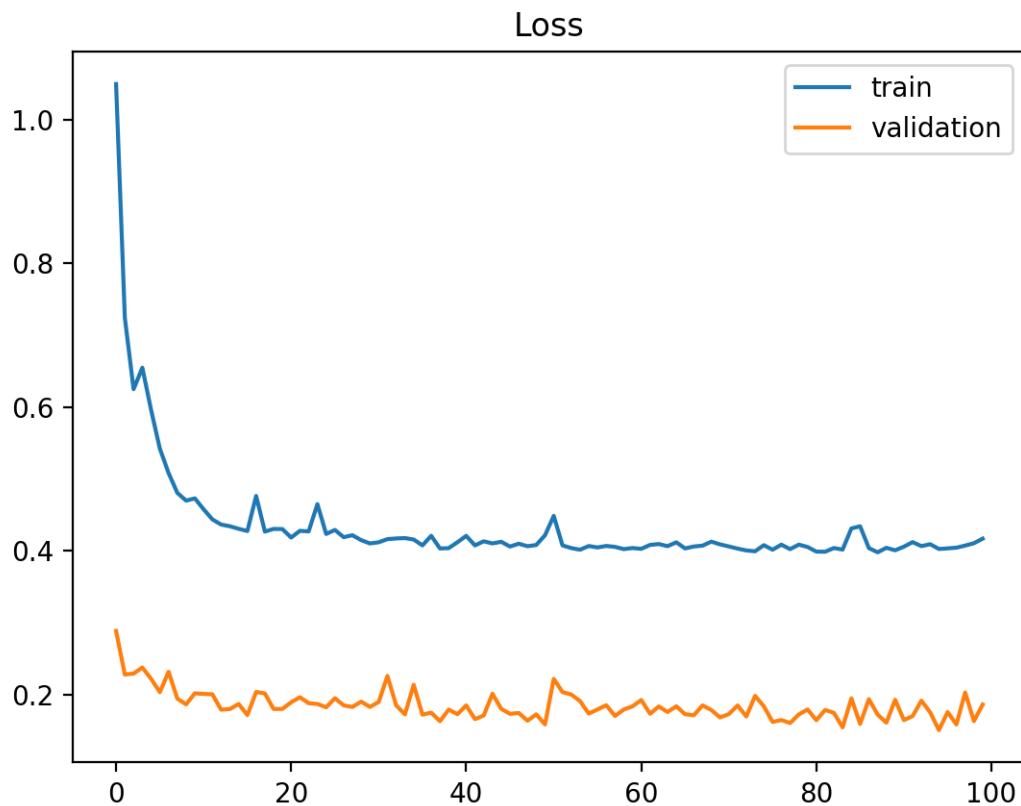


Figure 7: Example of Train and Validation Learning Curves Showing a Validation Dataset That Is Easier to Predict Than the Training Dataset.

Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Books

- *Deep Learning*, 2016.
<https://amzn.to/2SmfnCA>
- *An Introduction to Statistical Learning: with Applications in R*, 2013.
<https://amzn.to/2SkKXAy>

Papers

- *Learning curve models and applications: Literature review and research directions*, 2011.
<https://www.sciencedirect.com/science/article/pii/S016981411100062X>

Articles

- Learning curve, Wikipedia.
https://en.wikipedia.org/wiki/Learning_curve
- Overfitting, Wikipedia.
<https://en.wikipedia.org/wiki/Overfitting>

Summary

In this tutorial, you discovered learning curves and how they can be used to diagnose the learning and generalization behavior of machine learning models. Specifically, you learned:

- Learning curves are plots that show changes in learning performance over time in terms of experience.
- Learning curves of model performance on the train and validation datasets can be used to diagnose an underfit, overfit, or well-fit model.
- Learning curves of model performance can be used to diagnose whether the train or validation datasets are not relatively representative of the problem domain.

Next

Now that you are familiar with the framework for improving deep learning neural network model performance and how to diagnose model behavior, we can start to look at specific techniques. In the next part, you will discover techniques for improving convergence when training neural network models.

Part I

Better Learning

Overview

In this part you will discover techniques to improve the optimization problem of adapting neural network model weights in order to learn a training dataset. After reading the chapters in this part, you will know:

- How the challenge of training a neural network involves solving a non-convex optimization problem with no guarantees of convergence on a viable solution (Chapter 1).
- How the capacity of the model controls the scope of functions that can be learned and how the number of nodes and layers control capacity (Chapter 2).
- How the batch size controls the precision of the estimate of error used to update model weights and in turn the stability and speed of convergence (Chapter 3).
- How loss functions control the nature of the function approximation problem that is being solved (Chapter 4).
- How the learning rate controls the amount that model parameters are updated and in turn the stability and speed of convergence (Chapter 5).
- How the training process is sensitive to the scale of input and target variables and how normalization and standardization processes can dramatically improve model convergence (Chapter 6).
- How the vanishing gradient problem can be addressed with the rectified linear activation function and dramatically improve the likelihood and speed of convergence (Chapter 7).
- How the exploding gradient problem can be addressed with gradient norm scaling and gradient value clipping (Chapter 8).
- How the speed of convergence can be accelerated through the standardizing of internal representations with batch normalization (Chapter 9).
- How greedy layer-wise pretraining can facilitate the development of deeper models, a milestone in the field of deep learning (Chapter 10).
- How transfer learning can be used as a feature extraction and weight initialization scheme to short-cut the training process (Chapter 11).

Chapter 1

Improve Learning by Understanding Optimization

Deep learning neural networks learn a mapping function from inputs to outputs. This is achieved by updating the weights of the network in response to the errors the model makes on the training dataset. Updates are made to continually reduce this error until either a good enough model is found or the learning process gets stuck and stops. The process of training neural networks is the most challenging part of using the technique in general and is by far the most time consuming, both in terms of effort required to configure the process and computational complexity required to execute the process. In this tutorial, you will discover the challenge of finding model parameters for deep learning neural networks. After reading this tutorial, you will know:

- Neural networks learn a mapping function from inputs to outputs that can be summarized as solving the problem of function approximation.
- Unlike other machine learning algorithms, the parameters of a neural network must be found by solving a non-convex optimization problem with many good solutions and many misleadingly good solutions.
- The stochastic gradient descent algorithm is used to solve the optimization problem where model parameters are updated each iteration using the backpropagation algorithm.

Let's get started.

1.1 Neural Nets Learn a Mapping Function

Deep learning neural networks learn a mapping function. Developing a model requires historical data from the domain that is used as training data. This data is comprised of observations or examples from the domain with input elements that describe the conditions and an output element that captures what the observation means. For example, a problem where the output is a quantity would be described generally as a regression predictive modeling problem. Whereas a problem where the output is a label would be described generally as a classification predictive modeling problem.

A neural network model uses the examples to learn how to map specific sets of input variables to the output variable. It must do this in such a way that this mapping works well for the training dataset, but also works well on new examples not seen by the model during training. This ability to work well on specific examples and new examples is called the ability of the model to generalize.

A multilayer perceptron is just a mathematical function mapping some set of input values to output values.

— Page 5, *Deep Learning*, 2016.

We can describe the relationship between the input variables and the output variables as a complex mathematical function. For a given modeling problem, we must believe that a true mapping function exists to best map input variables to output variables and that a neural network model can do a reasonable job at approximating the true unknown underlying mapping function.

A feedforward network defines a mapping and learns the value of the parameters that result in the best function approximation.

— Page 168, *Deep Learning*, 2016.

As such, we can describe the broader problem that neural networks solve as *function approximation*. They learn to approximate an unknown underlying mapping function given a training dataset. They do this by learning weights (the model parameters), given a specific network structure that we must specify.

It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

— Page 169, *Deep Learning*, 2016.

1.2 Learning Network Weights Is Hard

Training deep learning neural networks is very challenging.

Optimization in general is an extremely difficult task. [...] When training neural networks, we must confront the general non-convex case.

— Page 282, *Deep Learning*, 2016.

An optimization process can be understood conceptually as a search through a landscape for a candidate solution that is sufficiently satisfactory. A point on the landscape is a specific set of weights for the model, and the elevation of that point is an evaluation of the set of weights, where valleys represent good models with small values of loss. This is a common conceptualization of optimization problems and the landscape is referred to as an *error surface*.

In general, $E(w)$ [the error function of the weights] is a multidimensional function and impossible to visualize. If it could be plotted as a function of w [the weights], however, E [the error function] might look like a landscape with hills and valleys ...

— Page 113, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

The optimization algorithm iteratively steps across this landscape, updating the weights and seeking out good or low elevation areas. For simple optimization problems, the shape of the landscape is a big bowl and finding the bottom is easy, so easy that very efficient algorithms can be designed to find the best solution. These types of optimization problems are referred to mathematically as convex.

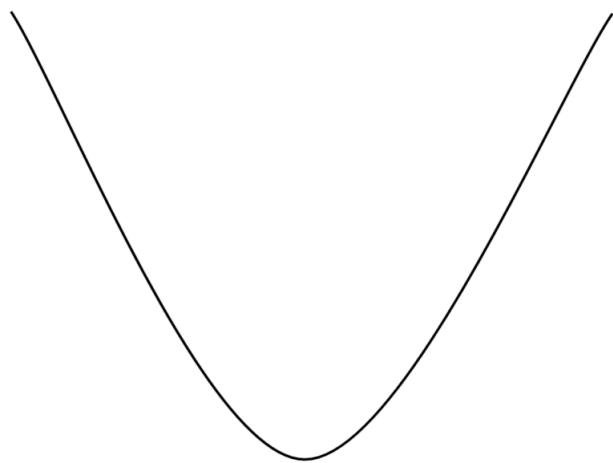


Figure 1.1: Example of a Convex Error Surface.

The error surface we wish to navigate when optimizing the weights of a neural network is not a bowl shape. It is a landscape with many hills and valleys. These type of optimization problems are referred to mathematically as non-convex.

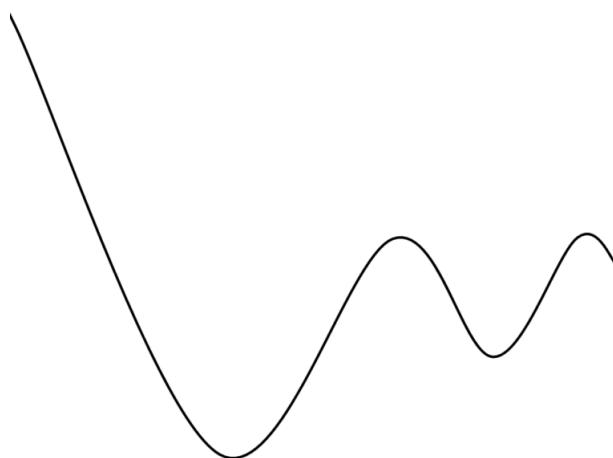


Figure 1.2: Example of a Non-Convex Error Surface.

In fact, there does not exist an algorithm to solve the problem of finding an optimal set of weights for a neural network in polynomial time. Mathematically, the optimization problem solved by training a neural network is referred to as NP-complete (i.e. it belongs to a class of hard to solve optimization problems).

We prove this problem NP-complete and thus demonstrate that learning in neural networks has no efficient general solution.

— *Neural Network Design and the Complexity of Learning*, 1988.

1.3 Key Features of the Error Surface

There are many types of non-convex optimization problems, but the specific type of problem we are solving when training a neural network is particularly challenging. We can characterize the difficulty in terms of the features of the landscape or error surface that the optimization algorithm may encounter and must navigate in order to be able to deliver a good solution. There are many aspects of the optimization of neural network weights that make the problem challenging, but three often-mentioned features of the error landscape are the presence of local minima, flat regions, and the high-dimensionality of the search space.

Backpropagation can be very slow particularly for multilayered networks where the cost surface is typically non-quadratic, non-convex, and high dimensional with many local minima and/or flat regions.

— Page 13, *Neural Networks: Tricks of the Trade*, 2012.

1.3.1 Local Minima

Local minimal or local optima refer to the fact that the error landscape contains multiple regions where the loss is relatively low. These are valleys, where solutions in those valleys look good relative to the slopes and peaks around them. The problem is, in the broader view of the entire landscape, the valley has a relatively high elevation and better solutions may exist. It is hard to know whether the optimization algorithm is in a local minima or not, therefore, it is good practice to start the optimization process with a lot of noise, allowing the landscape to be sampled widely before selecting a valley to fall into.

By contrast, the lowest point in the landscape is referred to as the *global minima*. Neural networks may have one or more global minima, and the challenge is that the difference between the local and global minima may not make a lot of difference. The implication of this is that often finding a *good enough* set of weights is more tractable and, in turn, more desirable than finding a global optimal or best set of weights.

Nonlinear networks usually have multiple local minima of differing depths. The goal of training is to locate one of these minima.

— Page 14, *Neural Networks: Tricks of the Trade*, 2012.

A classical approach to addressing the problem of local minima is to restart the search process multiple times with a different starting point (random initial weights) and allow the optimization algorithm to find a different, and hopefully better, local minima. This is called *multiple restarts* or *random restarts*.

Random Restarts: One of the simplest ways to deal with local minima is to train many different networks with different initial weights.

— Page 121, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

1.3.2 Flat Regions (Saddle Points)

A flat region or saddle point is a point on the landscape where the gradient is zero. These are flat regions at the bottom of valleys or regions between peaks. The problem is that a zero gradient means that the optimization algorithm does not know which direction to move in order to improve the model.

... the presence of saddlepoints, or regions where the error function is very flat, can cause some iterative algorithms to become ‘stuck’ for extensive periods of time, thereby mimicking local minima.

— Page 255, *Neural Networks for Pattern Recognition*, 1995.

Nevertheless, recent work may suggest that perhaps local minima and flat regions may be less of a challenge than was previously believed.

Do neural networks enter and escape a series of local minima? Do they move at varying speed as they approach and then pass a variety of saddle points? [...] we present evidence strongly suggesting that the answer to all of these questions is no.

— *Qualitatively characterizing neural network optimization problems*, 2015.

1.3.3 High-Dimensional

The optimization problem solved when training a neural network is high-dimensional. Each weight in the network represents another parameter or dimension of the error surface. Deep neural networks often have millions of parameters, making the landscape to be navigated by the algorithm extremely high-dimensional, as compared to more traditional machine learning algorithms. The problem of navigating a high-dimensional space is that the addition of each new dimension dramatically increases the distance between points in the search space. This is often referred to as the *curse of dimensionality*.

This phenomenon is known as the curse of dimensionality. Of particular concern is that the number of possible distinct configurations of a set of variables increases exponentially as the number of variables increases.

— Page 155, *Deep Learning*, 2016.

1.4 Navigating the Non-Convex Error Surface

A model with a specific set of weights can be evaluated on the training dataset and the average error over all training examples can be thought of as the error of the model. A change to the model weights will result in a change to the model error. Therefore, we seek a set of weights that result in a model with a small error. This involves repeating the steps of evaluating the model and updating the model parameters in order to step down the error surface. This process is repeated until a set of parameters is found that is good enough or the search process gets stuck. The settling of the optimization process on a solution is referred to as *convergence*, as the process has converged on a solution.

This is a search or an optimization process and we refer to optimization algorithms that operate in this way as gradient optimization algorithms, as they naively follow along the error gradient. They are computationally expensive, slow, and their empirical behavior means that using them in practice is more art than science. The algorithm that is most commonly used to navigate the error surface is called stochastic gradient descent, or SGD for short.

Nearly all of deep learning is powered by one very important algorithm: stochastic gradient descent or SGD.

— Page 151, *Deep Learning*, 2016.

Other global optimization algorithms designed for non-convex optimization problems could be used, such as a genetic algorithm, but stochastic gradient descent is more efficient as it uses the gradient information specifically to update the model weights via an algorithm called backpropagation.

[Backpropagation] describes a method to calculate the derivatives of the network training error with respect to the weights by a clever application of the derivative chain-rule.

— Page 49, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

Backpropagation refers to a technique from calculus to calculate the derivative (e.g. the slope or the gradient) of the model error for specific model parameters, allowing model weights to be updated to move down the gradient. As such, the algorithm used to train neural networks is also often referred to as simply backpropagation.

Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient.

— Page 204, *Deep Learning*, 2016.

1.5 Implications for Training

The challenging nature of optimization problems to be solved when using deep learning neural networks has implications when training models in practice. The best general algorithm known for solving this problem is stochastic gradient descent, that although is effective, makes no guarantees.

There is no formula to guarantee that (1) the network will converge to a good solution, (2) convergence is swift, or (3) convergence even occurs at all.

— Page 13, *Neural Networks: Tricks of the Trade*, 2012.

We can summarize these implications as follows:

- **Possibly Questionable Solution Quality.** The optimization process may or may not find a good solution and solutions can only be compared relatively, due to deceptive local minima.
- **Possibly Long Training Time.** The optimization process may take a long time to find a satisfactory solution, due to the iterative nature of the search.
- **Possible Failure.** The optimization process may fail to progress (get stuck) or fail to locate a viable solution, due to the presence of flat regions.

The task of effective training is to carefully configure, test, and tune the hyperparameters of the model and the learning process itself to best address this challenge. Thankfully, modern advancements can dramatically simplify the search space and accelerate the search process, often discovering models much larger, deeper, and with better performance than previously thought possible.

1.6 Components of the Learning Algorithm

Training a deep learning neural network model using stochastic gradient descent with backpropagation involves choosing a number of components and hyperparameters, they are:

- Network Topology.
- Loss Function.
- Weight Initialization.
- Batch Size.
- Learning Rate.
- Epochs.
- Data Preparation.

They are not all equal, some are more important than others. Further, they are not all independent, some choices can lead to sensible defaults for other configuration elements. Let's take a closer look at each in turn.

The capacity of a neural network defines the scope of the mapping functions that the model can approximate. A larger capacity means that the model is more flexible, but harder to train as it has many more parameters that have to be learned and provides a more challenging optimization problem to solve. The number of nodes in the hidden layer define the capacity, and a network with a single hidden layer with a sufficient number of nodes can approximate any mapping function (so-called universal approximation). Although, in practice this is not practical and the addition of layers with fewer nodes can increase the capacity of the model and make the optimization problem easier to solve.

- **Network Topology.** The number of nodes (or equivalent) in the hidden layers and the number of hidden layers in the network (Chapter 2).

An error function must be chosen, often called the objective function, cost function, or the loss function. Typically, a specific probabilistic framework for inference is chosen called Maximum Likelihood. Under this framework, the commonly chosen loss functions are cross-entropy for classification problems and mean squared error for regression problems.

- **Loss Function.** The function used to measure the performance of a model with a specific set of weights on examples from the training dataset (Chapter 4).

The search or optimization process requires a starting point from which to begin model updates. The starting point is defined by the initial model parameters or weights. Because the error surface is non-convex, the optimization algorithm is sensitive to the initial starting point. As such, small random values are chosen as the initial model weights, although different techniques can be used to select the scale and distribution of these values. These techniques are referred to as *weight initialization* methods. This can be tied to the choice of activation function (covered in (Chapter 7)).

- **Weight Initialization.** The procedure by which the initial small random values are assigned to model weights at the beginning of the training process.

When updating the model, a number of examples from the training dataset must be used to calculate the model error, often referred to simply as *loss*. All examples in the training dataset may be used, which may be appropriate for smaller datasets. Alternately, a single example may be used which may be appropriate for problems where examples are streamed or where the data changes often. A hybrid approach may be used where the number of examples from the training dataset may be chosen and used to estimate the error gradient. The choice of the number of examples is referred to as the batch size.

- **Batch Size.** The number of examples used to estimate the error gradient before updating the model parameters (Chapter 3).

Once an error gradient has been estimated, the derivative of the activation function can be calculated and used to update each parameter. There may be statistical noise in the training dataset and in the estimate of the error gradient. Also, the depth of the model (number of

layers) and the fact that model parameters are updated separately means that it is hard to calculate exactly how much to change each model parameter to best way to move the whole model down the error surface. Instead, a small portion of the update to the weights is performed each iteration. A hyperparameter called the *learning rate* controls how much to update model weights and, in turn, controls how fast a model learns on the training dataset.

- **Learning Rate:** The amount that each model parameter is updated per iteration of the learning algorithm (Chapter 5).

The training process must be repeated many times until a good or good enough set of model parameters is discovered. The total number of iterations of the process is bounded by the number of complete passes through the training dataset after which the training process is terminated. This is referred to as the number of training *epochs*. This hyperparameter is tightly related to both the choice of learning rate and batch size and can be set to a large value and almost ignored when using some regularization methods (for example see early stopping in Chapter 18).

- **Epochs.** The number of complete passes through the training dataset before the training process is terminated.

An often neglected aspect is the nature of the data used to learn the mapping function. The scale of the target variable is tightly related to the choice of activation function in the output layer of the network. The scale of the input variables will strongly effect the scale of the weights in the input and first few hidden layers of the network and in turn the stability of the learning process. The concerns of the scale and structure of the data used for learning is referred to as data preparation.

- **Data Preparation.** The schemes used to prepare the data prior to modeling in order to ensure that it is suitable for the problem and for developing a stable model (Chapter 6).

There are many extensions to the learning algorithm, although these components and hyperparameters generally control the learning algorithm for deep learning neural networks. What makes this more challenging is that there are no rules to best configure a network for a given problem.

Designing and training a network using backprop requires making many seemingly arbitrary choices such as the number and types of nodes, layers, learning rates, training and test sets, and so forth. These choices can be critical, yet there is no foolproof recipe for deciding them because they are largely problem and data dependent.

— *Efficient BackProp*, 1998.

1.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

1.7.1 Books

- *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
<https://amzn.to/2S8qRdI>
- *Neural Networks for Pattern Recognition*, 1995.
<https://amzn.to/2S8qdwt>
- *Neural Networks: Tricks of the Trade*, 2012.
<https://amzn.to/2DX69sk>

1.7.2 Papers

- *Efficient BackProp*, 1998.
<http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>
- *Training a 3-Node Neural Network is NP-Complete*, 1992.
<https://www.sciencedirect.com/science/article/pii/S0893608005800103>
- *Qualitatively characterizing neural network optimization problems*, 2015.
<https://arxiv.org/abs/1412.6544>
- *Neural Network Design and the Complexity of Learning*, 1988.
<https://authors.library.caltech.edu/26705/1/88-20.pdf>

1.7.3 Articles

- Saddle point, Wikipedia.
https://en.wikipedia.org/wiki/Saddle_point
- Curse of dimensionality, Wikipedia.
https://en.wikipedia.org/wiki/Curse_of_dimensionality
- NP-completeness, Wikipedia.
<https://en.wikipedia.org/wiki/NP-completeness>

1.8 Summary

In this tutorial, you discovered the challenge of finding model parameters for deep learning neural networks. Specifically, you learned:

- Neural networks learn a mapping function from inputs to outputs that can be summarized as solving the problem of function approximation.
- Unlike other machine learning algorithms, the parameters of a neural network must be found by solving a non-convex optimization problem with many good solutions and many misleadingly good solutions.

- The stochastic gradient descent algorithm is used to solve the optimization problem where model parameters are updated each iteration using the backpropagation algorithm.

1.8.1 Next

In the next tutorial, you will discover how the number of nodes controls the capacity of a neural network and in turn the scope of the functions that can be learned.

Chapter 2

Configure Capacity with Nodes and Layers

The capacity of a deep learning neural network model controls the scope of the mapping functions that it is able to learn. A model with too little capacity cannot learn the training dataset meaning it will underfit, whereas a model with too much capacity may memorize the training dataset, meaning it will overfit or may get stuck or lost during the optimization process. The capacity of a neural network model is defined by configuring the number of nodes and the number of layers. In this tutorial, you will discover how to control the capacity of a neural network model and how capacity impacts what a model is capable of learning. After completing this tutorial, you will know:

- Neural network model capacity is controlled both by the number of nodes and the number of layers in the model.
- A model with a single hidden layer and sufficient number of nodes has the capability of learning any mapping function, but the chosen learning algorithm may or may not be able to realize this capability.
- Increasing the number of layers provides a short-cut to increasing the capacity of the model with fewer resources, and modern techniques allow learning algorithms to successfully train deep models.

Let's get started.

2.1 Neural Network Model Capacity

The goal of a neural network is to learn how to map input examples to output examples. Neural networks learn mapping functions. The capacity of a network refers to the range or scope of the functions that the model can approximate.

Informally, a model's capacity is its ability to fit a wide variety of functions.

— Pages 111-112, *Deep Learning*, 2016.

A model with less capacity may not be able to sufficiently learn the training dataset. A model with more capacity can model more different functions and may be able to learn a function to sufficiently map inputs to outputs in the training dataset. Whereas a model with too much capacity may memorize the training dataset and fail to generalize or get lost or stuck in the search for a suitable mapping function. Generally, we can think of model capacity as a control over whether the model is likely to underfit or overfit a training dataset.

We can control whether a model is more likely to overfit or underfit by altering its capacity.

— Page 111, *Deep Learning*, 2016.

The capacity of a neural network can be controlled by two aspects of the model:

- Number of Nodes.
- Number of Layers.

A model with more nodes or more layers has a greater capacity and, in turn, is potentially capable of navigating a larger set of mapping functions.

A model with more layers and more hidden units per layer has higher representational capacity - it is capable of representing more complicated functions.

— Page 428, *Deep Learning*, 2016.

The number of nodes in a layer is referred to as the *width*. Developing wide networks with one layer and many nodes was relatively straightforward. In theory, a network with enough nodes in the single hidden layer can learn to approximate any mapping function, although in practice, we don't know how many nodes are sufficient or how to train such a model. The number of layers in a model is referred to as its *depth*. Increasing the depth increases the capacity of the model. Training deep models, e.g. those with many hidden layers, can be computationally more efficient than training a single layer network with a vast number of nodes.

Modern deep learning provides a very powerful framework for supervised learning. By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity.

— Page 167, *Deep Learning*, 2016.

Traditionally, it has been challenging to train neural network models with more than a few layers due to problems such as vanishing gradients. More recently, modern methods have allowed the training of deep network models, allowing the developing of models of surprising depth that are capable of achieving impressive performance on challenging problems in a wide range of domains.

2.2 Nodes and Layers Keras API

Keras allows you to easily add nodes and layers to your model.

2.2.1 Configuring Model Nodes

The first argument of the layer specifies the number of nodes used in the layer. Fully connected layers for the Multilayer Perceptron, or MLP, model are added via the `Dense` layer. For example, we can create one fully-connected layer with 32 nodes as follows:

```
...  
layer = Dense(32)
```

Listing 2.1: Example of specifying the number of nodes for a `Dense` layer.

Similarly, the number of nodes can be specified for recurrent neural network layers in the same way. For example, we can create one LSTM layer with 32 nodes (or units) as follows:

```
...  
layer = LSTM(32)
```

Listing 2.2: Example of specifying the number of nodes for an LSTM layer.

Convolutional neural networks, or CNNs, don't have nodes, instead specify the number of filter maps and their shape. The number and size of filter maps define the capacity of the layer. We can define a two-dimensional CNN with 32 filter maps, each with a size of 3 by 3, as follows:

```
...  
layer = Conv2D(32, (3,3))
```

Listing 2.3: Example of specifying the number of filter maps for a CNN layer.

2.2.2 Configuring Model Layers

Layers are added to a sequential model via calls to the `add()` function and passing in the layer. Fully connected layers for the MLP can be added via repeated calls to the `add()` function passing in the configured `Dense` layers; for example:

```
...  
model = Sequential()  
model.add(Dense(32))  
model.add(Dense(64))
```

Listing 2.4: Example of specifying the number of layers for an MLP.

Similarly, the number of layers for a recurrent network can be added in the same way to give a stacked recurrent model. An important difference is that recurrent layers expect a three-dimensional input, therefore the prior recurrent layer must return the full sequence of outputs rather than the single output for each node at the end of the input sequence. This can be achieved by setting the `return_sequences` argument to `True`. For example:

```
...  
model = Sequential()  
model.add(LSTM(32, return_sequences=True))  
model.add(LSTM(32))
```

Listing 2.5: Example of specifying the number of layers for an LSTM.

Convolutional layers can be stacked directly, and it is common to stack one or two convolutional layers together followed by a pooling layer, then repeat this pattern of layers; for example:

```
...
model = Sequential()
model.add(Conv2D(16, (3,3)))
model.add(Conv2D(16, (3,3)))
model.add(MaxPooling2D((2,2)))
model.add(Conv2D(32, (3,3)))
model.add(Conv2D(32, (3,3)))
model.add(MaxPooling2D((2,2)))
```

Listing 2.6: Example of specifying the number of layers for a CNN.

Now that we know how to configure the number of nodes and layers for models in Keras, we can look at how the capacity affects model performance on a multiclass classification problem.

2.3 Model Capacity Case Study

In this section, we will demonstrate how to use model capacity to control learning with a MLP on a simple classification problem. This example provides a template for exploring model capacity with your own neural network for classification and regression problems.

2.3.1 Multiclass Classification Problem

We will use a standard multiclass classification problem as the basis to demonstrate the effect of model capacity on model performance. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. We can configure the problem to have a specific number of input variables via the `n_features` argument, and a specific number of classes or centers via the `centers` argument. The `random_state` can be used to seed the pseudorandom number generator to ensure that we always get the same samples each time the function is called. For example, the call below generates 1,000 examples for a three class problem with two input variables.

```
...
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2)
```

Listing 2.7: Example of creating samples for the blobs problem.

The results are the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can plot each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot for each class value
for class_value in range(3):
    # select indices of points with the class label
    row_ix = where(y == class_value)
```

```
# scatter plot for points with a different color
pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 2.8: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the chosen standard deviation of 2.0 means that the classes are not linearly separable (separable by a line), causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions.

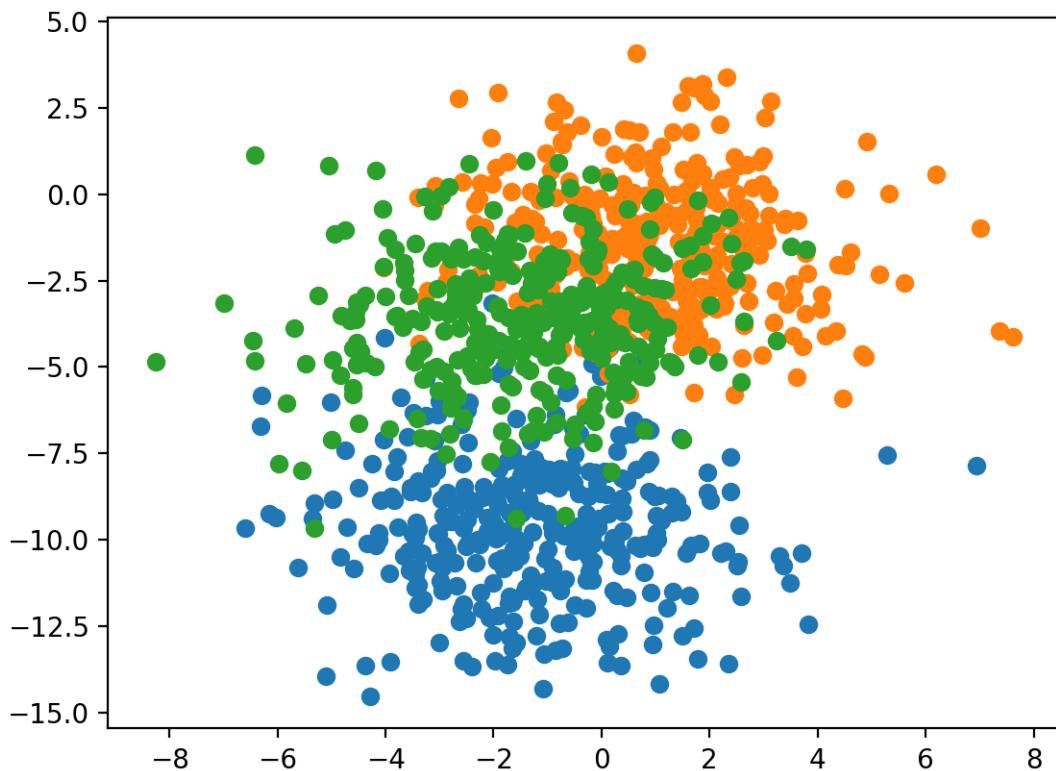


Figure 2.1: Scatter Plot of Blobs Dataset With Three Classes and Points Colored by Class Value.

In order to explore model capacity, we need more complexity in the problem than three classes and two variables. For the purposes of the following experiments, we will use 100 input features and 20 classes; for example:

```
...
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=20, n_features=100, cluster_std=2, random_state=2)
```

Listing 2.9: Example of creating a large number of samples for the blobs problem.

2.3.2 Change Model Capacity With Nodes

In this section, we will develop a Multilayer Perceptron model, or MLP, for the blobs multiclass classification problem and demonstrate the effect that the number of nodes has on the ability of the model to learn. We can start off by developing a function to prepare the dataset. The input and output elements of the dataset can be created using the `make_blobs()` function as described in the previous section. Next, the target variable must be one hot encoded. This is so that the model can learn to predict the probability of an input example belonging to each of the 20 classes. We can use the `to_categorical()` Keras utility function to do this, for example:

```
# one hot encode output variable
y = to_categorical(y)
```

Listing 2.10: Example of one hot encoding the target variable.

Next, we can split the 1,000 examples in half and use 500 examples as the training dataset and 500 to evaluate the model.

```
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
return trainX, trainy, testX, testy
```

Listing 2.11: Example of preparing the dataset for modeling.

The `create_dataset()` function below ties these elements together and returns the train and test sets in terms of the input and output elements.

```
# prepare multiclass classification dataset
def create_dataset():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=20, n_features=100, cluster_std=2,
                      random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy
```

Listing 2.12: Example of defining a function for preparing the dataset for modeling.

We can call this function to prepare the dataset.

```
# prepare dataset
trainX, trainy, testX, testy = create_dataset()
```

Listing 2.13: Example of preparing the dataset.

Next, we can define a function that will create the model, fit it on the training dataset, and then evaluate it on the test dataset. The model needs to know the number of input variables in order to configure the input layer and the number of target classes in order to configure the output layer. These properties can be extracted from the training dataset directly.

```
# configure the model based on the data
n_input, n_classes = trainX.shape[1], testy.shape[1]
```

Listing 2.14: Example of determining the number of inputs and classes.

We will define an MLP model with a single hidden layer that uses the rectified linear activation function and the He random weight initialization method. The output layer will use the softmax activation function in order to predict a probability for each target class. The number of nodes in the hidden layer will be provided via an argument called `n_nodes`.

```
# define model
model = Sequential()
model.add(Dense(n_nodes, input_dim=n_input, activation='relu',
    kernel_initializer='he_uniform'))
model.add(Dense(n_classes, activation='softmax'))
```

Listing 2.15: Example of defining an MLP model.

The model will be optimized using stochastic gradient descent with a modest learning rate of 0.01 with a high momentum of 0.9, and a categorical cross-entropy loss function will be used, suitable for multiclass classification.

```
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 2.16: Example of compiling an MLP model.

The model will be fit for 100 training epochs, then the model will be evaluated on the test dataset.

```
# fit model on train set
history = model.fit(trainX, trainy, epochs=100, verbose=0)
# evaluate model on test set
_, test_acc = model.evaluate(testX, testy, verbose=0)
```

Listing 2.17: Example of fitting and evaluating an MLP model.

Tying these elements together, the `evaluate_model()` function below takes the number of nodes and dataset as arguments and returns the history of the training loss at the end of each epoch and the accuracy of the final model on the test dataset.

```
# fit model with given number of nodes, returns test set accuracy
def evaluate_model(n_nodes, trainX, trainy, testX, testy):
    # configure the model based on the data
    n_input, n_classes = trainX.shape[1], testy.shape[1]
    # define model
    model = Sequential()
    model.add(Dense(n_nodes, input_dim=n_input, activation='relu',
        kernel_initializer='he_uniform'))
    model.add(Dense(n_classes, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    # fit model on train set
    history = model.fit(trainX, trainy, epochs=100, verbose=0)
    # evaluate model on test set
```

```
_, test_acc = model.evaluate(testX, testy, verbose=0)
return history, test_acc
```

Listing 2.18: Example of defining a function to define, fit and evaluate an MLP model.

We can call this function with different numbers of nodes to use in the hidden layer. The problem is relatively simple; therefore, we will review the performance of the model with 1 to 7 nodes. We would expect that increasing the number of nodes would increase the capacity of the model and allow the model to better learn the training dataset, at least to a point limited by the chosen configuration for the learning algorithm (e.g. learning rate, batch size, and epochs). The test accuracy for each configuration will be printed and the learning curves of training accuracy with each configuration will be plotted.

```
# evaluate model and plot learning curve with given number of nodes
num_nodes = [1, 2, 3, 4, 5, 6, 7]
for n_nodes in num_nodes:
    # evaluate model with a given number of nodes
    history, result = evaluate_model(n_nodes, trainX, trainy, testX, testy)
    # summarize final test set accuracy
    print('nodes=%d: %.3f' % (n_nodes, result))
    # plot learning curve
    pyplot.plot(history.history['loss'], label=str(n_nodes))
# show the plot
pyplot.legend()
pyplot.show()
```

Listing 2.19: Example of evaluating models with different numbers of nodes.

The full code listing is provided below for completeness.

```
# study of mlp learning curves given different number of nodes for multi-class
# classification
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare multi-class classification dataset
def create_dataset():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=20, n_features=100, cluster_std=2,
                      random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy

# fit model with given number of nodes, returns test set accuracy
def evaluate_model(n_nodes, trainX, trainy, testX, testy):
    # configure the model based on the data
    n_input, n_classes = trainX.shape[1], testy.shape[1]
```

```

# define model
model = Sequential()
model.add(Dense(n_nodes, input_dim=n_input, activation='relu',
    kernel_initializer='he_uniform'))
model.add(Dense(n_classes, activation='softmax'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model on train set
history = model.fit(trainX, trainy, epochs=100, verbose=0)
# evaluate model on test set
_, test_acc = model.evaluate(testX, testy, verbose=0)
return history, test_acc

# prepare dataset
trainX, trainy, testX, testy = create_dataset()
# evaluate model and plot learning curve with given number of nodes
num_nodes = [1, 2, 3, 4, 5, 6, 7]
for n_nodes in num_nodes:
    # evaluate model with a given number of nodes
    history, result = evaluate_model(n_nodes, trainX, trainy, testX, testy)
    # summarize final test set accuracy
    print('nodes=%d: %.3f' % (n_nodes, result))
    # plot learning curve
    pyplot.plot(history.history['loss'], label=str(n_nodes))
# show the plot
pyplot.legend()
pyplot.show()

```

Listing 2.20: Example of evaluating MLP models with differing numbers of nodes on the blobs problem.

Running the example first prints the test accuracy for each model configuration.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that as the number of nodes is increased, the capacity of the model to learn the problem is increased. This results in a progressive lowering of the generalization error of the model on the test dataset until 6 and 7 nodes when the model learns the problem perfectly.

```

nodes=1: 0.138
nodes=2: 0.380
nodes=3: 0.582
nodes=4: 0.890
nodes=5: 0.844
nodes=6: 1.000
nodes=7: 1.000

```

Listing 2.21: Example output from evaluating MLP models with differing numbers of nodes on the blobs problem.

A line plot is also created showing cross-entropy loss on the training dataset for each model configuration (1 to 7 nodes in the hidden layer) over the 100 training epochs. We can see that

as the number of nodes is increased, the model is able to better decrease the loss, e.g. to better learn the training dataset. This plot shows the direct relationship between model capacity, as defined by the number of nodes in the hidden layer and the model's ability to learn.

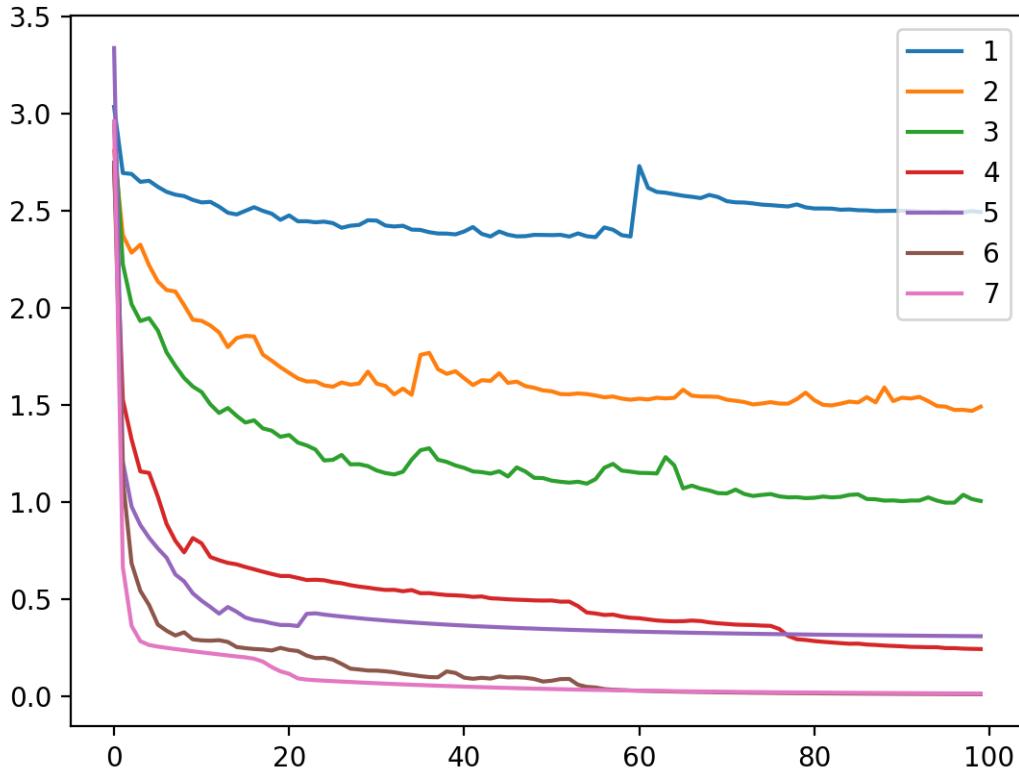


Figure 2.2: Line Plot of Cross-Entropy Loss Over Training Epochs for an MLP on the Training Dataset for the Blobs Multiclass Classification Problem When Varying Model Nodes.

The number of nodes can be increased to the point (e.g. 1,000 nodes) where the learning algorithm is no longer able to sufficiently learn the mapping function.

2.3.3 Change Model Capacity With Layers

We can perform a similar analysis and evaluate how the number of layers impacts the ability of the model to learn the mapping function. Increasing the number of layers can often greatly increase the capacity of the model, acting like a computational and learning shortcut to modeling a problem. For example, a model with one hidden layer of 10 nodes is not equivalent to a model with two hidden layers with five nodes each. The latter has a much greater capacity. The danger is that a model with more capacity than is required is likely to overfit the training data, and as with a model that has too many nodes, a model with too many layers will likely be unable to learn the training dataset, getting lost or stuck during the optimization process.

First, we can update the `evaluate_model()` function to fit an MLP model with a given number of layers. We know from the previous section that an MLP with about seven or more

nodes fit for 100 epochs will learn the problem perfectly. We will, therefore, use 10 nodes in each layer to ensure the model has enough capacity in just one layer to learn the problem. The updated function is listed below, taking the number of layers and dataset as arguments and returning the training history and test accuracy of the model.

```
# fit model with given number of layers, returns test set accuracy
def evaluate_model(n_layers, trainX, trainy, testX, testy):
    # configure the model based on the data
    n_input, n_classes = trainX.shape[1], testy.shape[1]
    # define model
    model = Sequential()
    model.add(Dense(10, input_dim=n_input, activation='relu',
        kernel_initializer='he_uniform'))
    for _ in range(1, n_layers):
        model.add(Dense(10, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(n_classes, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, epochs=100, verbose=0)
    # evaluate model on test set
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    return history, test_acc
```

Listing 2.22: Example of a function for evaluating a model with a differing number of layers.

Given that a single hidden layer model has enough capacity to learn this problem, we will explore increasing the number of layers to the point where the learning algorithm becomes unstable and can no longer learn the problem. If the chosen modeling problem was more complex, we could explore increasing the layers and review the improvements in model performance to a point of diminishing returns. In this case, we will evaluate the model with 1 to 5 layers, with the expectation that at some point, the number of layers will result in a model that the chosen learning algorithm is unable to adapt to the training data.

```
# evaluate model and plot learning curve of model with given number of layers
all_history = list()
num_layers = [1, 2, 3, 4, 5]
for n_layers in num_layers:
    # evaluate model with a given number of layers
    history, result = evaluate_model(n_layers, trainX, trainy, testX, testy)
    print('layers=%d: %.3f' % (n_layers, result))
    # plot learning curve
    pyplot.plot(history.history['loss'], label=str(n_layers))
pyplot.legend()
pyplot.show()
```

Listing 2.23: Example of evaluating MLPs with differing numbers of layers.

Tying these elements together, the complete example is listed below.

```
# study of mlp learning curves given different number of layers for multi-class
# classification
from sklearn.datasets import make_blobs
from keras.models import Sequential
from keras.layers import Dense
```

```

from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare multi-class classification dataset
def create_dataset():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=20, n_features=100, cluster_std=2,
                       random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy

# fit model with given number of layers, returns test set accuracy
def evaluate_model(n_layers, trainX, trainy, testX, testy):
    # configure the model based on the data
    n_input, n_classes = trainX.shape[1], testy.shape[1]
    # define model
    model = Sequential()
    model.add(Dense(10, input_dim=n_input, activation='relu',
                   kernel_initializer='he_uniform'))
    for _ in range(1, n_layers):
        model.add(Dense(10, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(n_classes, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, epochs=100, verbose=0)
    # evaluate model on test set
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    return history, test_acc

# get dataset
trainX, trainy, testX, testy = create_dataset()
# evaluate model and plot learning curve of model with given number of layers
all_history = list()
num_layers = [1, 2, 3, 4, 5]
for n_layers in num_layers:
    # evaluate model with a given number of layers
    history, result = evaluate_model(n_layers, trainX, trainy, testX, testy)
    print('layers=%d: %.3f' % (n_layers, result))
    # plot learning curve
    pyplot.plot(history.history['loss'], label=str(n_layers))
pyplot.legend()
pyplot.show()

```

Listing 2.24: Example of evaluating MLP models with differing numbers of layers on the blobs problem.

Running the example first prints the test accuracy for each model configuration.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model is capable of learning the problem well with up to three layers, then begins to falter. We can see that performance really drops with five layers and is expected to continue to fall if the number of layers is increased further.

```
layers=1: 1.000
layers=2: 1.000
layers=3: 1.000
layers=4: 0.948
layers=5: 0.794
```

Listing 2.25: Example output from evaluating MLP models with differing numbers of layers on the blobs problem.

A line plot is also created showing cross-entropy loss on the training dataset for each model configuration (1 to 5 layers) over the 100 training epochs. We can see that the dynamics of the model with 1, 2, and 3 models (blue, orange and green) are pretty similar, learning the problem quickly. Surprisingly, training loss with four and five layers shows signs of initially doing well, then leaping up, suggesting that the model is likely stuck with a sub-optimal set of weights rather than overfitting the training dataset.

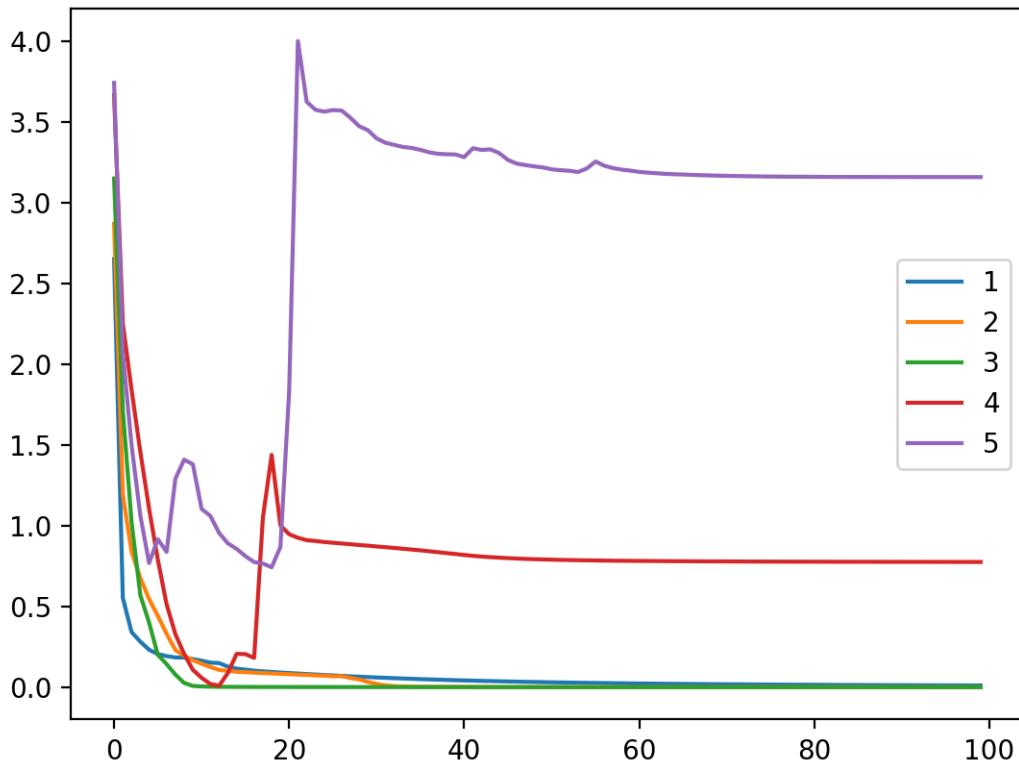


Figure 2.3: Line Plot of Cross-Entropy Loss Over Training Epochs for an MLP on the Training Dataset for the Blobs Multiclass Classification Problem When Varying Model Layers.

The analysis shows that increasing the capacity of the model via increasing depth is a very effective tool that must be used with caution as it can quickly result in a model with a large capacity that may not be capable of learning the training dataset easily.

2.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Too Many Nodes.** Update the experiment of increasing nodes to find the point where the learning algorithm is no longer capable of learning the problem.
- **Repeated Evaluation.** Update an experiment to use the repeated evaluation of each configuration to counter the stochastic nature of the learning algorithm.
- **Harder Problem.** Repeat the experiment of increasing layers on a problem that requires the increased capacity provided by increased depth in order to perform well.

If you explore any of these extensions, I'd love to know.

2.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

2.5.1 Books

- *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
<https://amzn.to/2vhyW8j>
- *Deep Learning*, 2016.
<https://amzn.to/2IXzUIY>

2.5.2 APIs

- Keras Core Layers API.
<https://keras.io/layers/core/>
- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>
- Keras Recurrent Layers API.
<https://keras.io/layers/recurrent/>
- Keras Utility Functions.
<https://keras.io/utils/>
- `sklearn.datasets.make_blobs` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

2.5.3 Articles

- How many hidden layers should I use?, comp.ai.neural-nets FAQ.
<http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-9.html>

2.6 Summary

In this tutorial, you discovered how to control the capacity of a neural network model and how capacity impacts what a model is capable of learning. Specifically, you learned:

- Neural network model capacity is controlled both by the number of nodes and the number of layers in the model.
- A model with a single hidden layer and a sufficient number of nodes has the capability of learning any mapping function, but the chosen learning algorithm may or may not be able to realize this capability.
- Increasing the number of layers provides a short-cut to increasing the capacity of the model with fewer resources, and modern techniques allow learning algorithms to successfully train deep models.

2.6.1 Next

In the next tutorial, you will discover how the batch size controls the precision of the estimate of the error gradient that in turn controls the stability and speed of convergence.

Chapter 3

Configure Gradient Precision with Batch Size

Neural networks are trained using gradient descent where the estimate of the error used to update the weights is calculated based on a subset of the training dataset. The number of examples from the training dataset used in the estimate of the error gradient is called the batch size and is an important hyperparameter that influences the dynamics of the learning algorithm. It is important to explore the dynamics of your model to ensure that you're getting the most out of it. In this tutorial, you will discover three different flavors of gradient descent and how to explore and diagnose the effect of batch size on the learning process. After completing this tutorial, you will know:

- Batch size controls the accuracy of the estimate of the error gradient when training neural networks.
- Batch, Stochastic, and Minibatch gradient descent are the three main flavors of the learning algorithm.
- There is a tension between batch size and the speed and stability of the learning process.

Let's get started.

3.1 Batch Size and Gradient Descent

Neural networks are trained using the stochastic gradient descent optimization algorithm. This involves using the current state of the model to make a prediction, comparing the prediction to the actual values, and using the difference as an estimate of the error gradient. This error gradient is then used to update the model weights and the process is repeated. The error gradient is a statistical estimate. The more training examples used in the estimate, the more accurate this estimate will be and the more likely that the weights of the network will be adjusted in a way that will improve the performance of the model. The improved estimate of the error gradient comes at the computational cost of having to use the model to make many more predictions before the estimate can be calculated, and in turn, the weights updated.

Optimization algorithms that use the entire training set are called batch or deterministic gradient methods, because they process all of the training examples simultaneously in a large batch.

— Page 278, *Deep Learning*, 2016.

Alternately, using fewer examples results in a less accurate estimate of the error gradient that is highly dependent on the specific training examples used. This results in a noisy estimate that, in turn, results in noisy updates to the model weights, e.g. many updates with perhaps quite different estimates of the error gradient. Nevertheless, these noisy updates can result in faster learning and sometimes a more robust model.

Optimization algorithms that use only a single example at a time are sometimes called stochastic or sometimes online methods. The term online is usually reserved for the case where the examples are drawn from a stream of continually created examples rather than from a fixed-size training set over which several passes are made.

— Page 278, *Deep Learning*, 2016.

The number of training examples used in the estimate of the error gradient is a hyperparameter for the learning algorithm called the *batch size*, or simply the *batch*. A batch size of 32 means that 32 samples from the training dataset will be used to estimate the error gradient before the model weights are updated. One training epoch means that the learning algorithm has made one pass through the training dataset (using every example once), where examples were separated into randomly selected *batch size* groups.

Historically, a training algorithm where the batch size is set to the total number of training examples is called *batch gradient descent* and a training algorithm where the batch size is set to 1 training example is called *stochastic gradient descent* or *online gradient descent*. A configuration of the batch size anywhere in between (e.g. more than 1 example and less than the number of examples in the training dataset) is called *minibatch gradient descent*.

- **Batch Gradient Descent.** Batch size is set to the total number of examples in the training dataset.
- **Stochastic Gradient Descent.** Batch size is set to one.
- **Minibatch Gradient Descent.** Batch size is set to more than one and less than the total number of examples in the training dataset.

For shorthand, the algorithm is often referred to as stochastic gradient descent regardless of the batch size. Given that very large datasets are often used to train deep learning neural networks, the batch size is rarely set to the size of the training dataset. Smaller batch sizes are used for two main reasons:

- Smaller batch sizes are noisy, offering a regularizing effect and lower generalization error.
- Smaller batch sizes make it easier to fit one batch worth of training data in memory (i.e. when using a GPU that has access to less local memory than system RAM).

A third reason is that the batch size is often set at something small, such as 32 examples, and is not tuned by the practitioner. Small batch sizes such as 32 do work well generally.

... [batch size] is typically chosen between 1 and a few hundreds, e.g. [batch size] = 32 is a good default value

— *Practical recommendations for gradient-based training of deep architectures*, 2012.

The presented results confirm that using small batch sizes achieves the best training stability and generalization performance, for a given computational cost, across a wide range of experiments. In all cases the best results have been obtained with batch sizes $m = 32$ or smaller, often as small as $m = 2$ or $m = 4$.

— *Revisiting Small Batch Training for Deep Neural Networks*, 2018.

Nevertheless, the batch size impacts how quickly a model learns and the stability of the learning process. It is an important hyperparameter that should be well understood and tuned by the deep learning practitioner.

3.2 Gradient Descent Keras API

Keras allows you to train your model using stochastic, batch, or minibatch gradient descent. This can be achieved by setting the `batch_size` argument on the call to the `fit()` function when training your model. Let's take a look at each approach in turn.

3.2.1 Batch Size Keras API

The example below sets the `batch_size` argument to 1 for stochastic gradient descent.

```
...
model.fit(trainX, trainy, batch_size=1)
```

Listing 3.1: Example of stochastic gradient descent in Keras.

3.2.2 Batch Gradient Descent in Keras

The example below sets the `batch_size` argument to the number of samples in the training dataset for batch gradient descent.

```
...
model.fit(trainX, trainy, batch_size=len(trainX))
```

Listing 3.2: Example of batch gradient descent in Keras.

3.2.3 Minibatch Gradient Descent in Keras

The example below uses the default batch size of 32 for the `batch_size` argument, which is more than 1 for stochastic gradient descent and less than the size of your training dataset for batch gradient descent.

```
...  
model.fit(trainX, trainy)
```

Listing 3.3: Example of minibatch gradient descent with default batch size in Keras.

Alternately, the `batch_size` can be specified to something other than 1 or the number of samples in the training dataset, such as 64.

```
...  
model.fit(trainX, trainy, batch_size=64)
```

Listing 3.4: Example of minibatch gradient descent in Keras.

3.3 Batch Size Case Study

In this section, we will demonstrate how to use gradient descent batch size to control learning with a MLP on a simple classification problem. This example provides a template for exploring batch size with your own neural network for classification and regression problems.

3.3.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate the effect of batch size on learning. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. The problem can be configured to have two input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same data points.

```
# generate 2d classification dataset  
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

Listing 3.5: Example of generating samples for the blobs problem.

The results are the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can plot each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# scatter plot of blobs dataset  
from sklearn.datasets import make_blobs  
from matplotlib import pyplot  
from numpy import where  
# generate 2d classification dataset  
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)  
# scatter plot for each class value  
for class_value in range(3):  
    # select indices of points with the class label
```

```

row_ix = where(y == class_value)
# scatter plot for points with a different color
pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()

```

Listing 3.6: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (separable by a line) causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions.

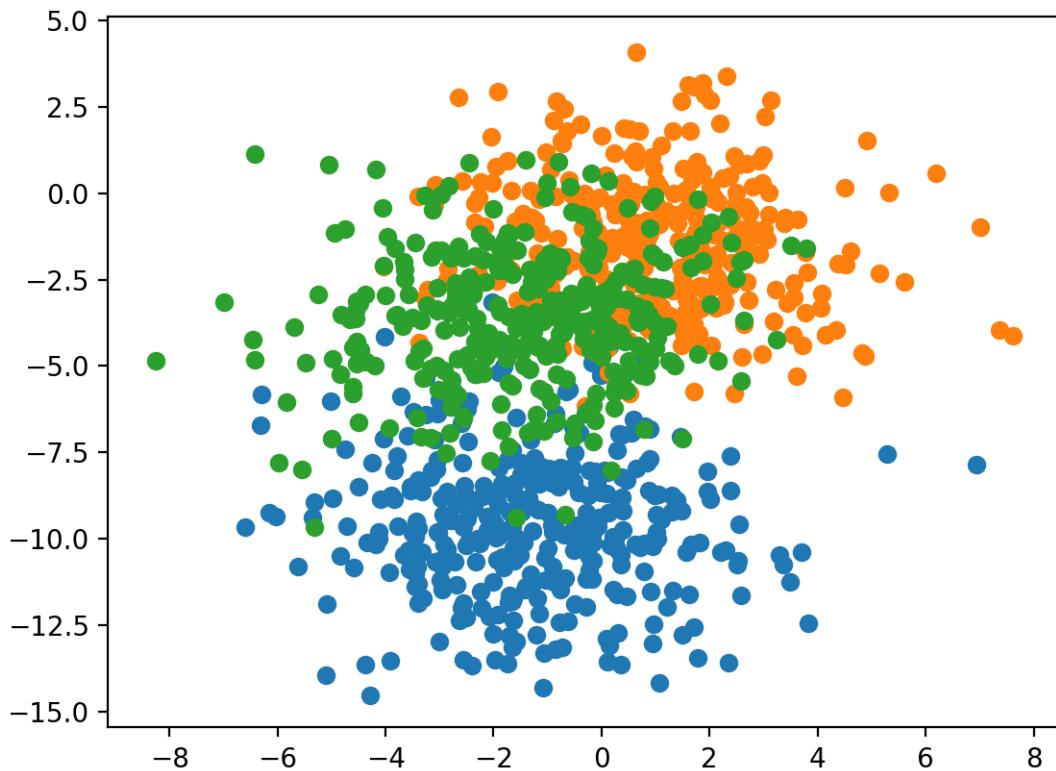


Figure 3.1: Scatter Plot of Blobs Dataset With Three Classes and Points Colored by Class Value.

3.3.2 MLP Fit With Batch Gradient Descent

We can develop a Multilayer Perceptron model (MLP) to address the multiclass classification problem described in the previous section and train it using batch gradient descent. Firstly, we need to one hot encode the target variable, transforming the integer class values into binary vectors. This will allow the model to predict the probability of each example belonging to each of the three classes, providing more nuance in the predictions and context when training the model.

```
# one hot encode output variable
y = to_categorical(y)
```

Listing 3.7: Example of one hot encoding the target variable.

Next, we will split the training dataset of 1,000 examples into a train and test dataset with 500 examples each. This even split will allow us to evaluate and compare the performance of different configurations of the batch size on the model and its performance.

```
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 3.8: Example of preparing the dataset for modeling.

We will define an MLP model with an input layer that expects two input variables, for the two variables in the dataset. The model will have a single hidden layer with 50 nodes and a rectified linear activation function and He random weight initialization. Finally, the output layer has 3 nodes in order to make predictions for the three classes and a softmax activation function.

```
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))
```

Listing 3.9: Example of defining the MLP model.

We will optimize the model with stochastic gradient descent and use categorical cross-entropy to calculate the error of the model during training. In this example, we will use *batch gradient descent*, meaning that the batch size will be set to the size of the training dataset. The model will be fit for 200 training epochs and the test dataset will be used as the validation set in order to monitor the performance of the model on a holdout set during training. The effect will be more time between weight updates and we would expect faster training than other batch sizes, and more stable estimates of the gradient, which should result in a more stable performance of the model during training.

```
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0,
batch_size=len(trainX))
```

Listing 3.10: Example of compiling and fitting the MLP model.

Once the model is fit, the performance is evaluated and reported on the train and test datasets.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 3.11: Example of evaluating the MLP model.

A line plot is created showing the train and test set accuracy of the model for each training epoch. These learning curves provide an indication of three things: how quickly the model learns the problem, how well it has learned the problem, and how noisy the updates were to the model during training.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 3.12: Example of plotting learning curves for the MLP model.

Tying these elements together, the complete example is listed below.

```
# mlp for the blobs problem with batch gradient descent
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0,
    batch_size=len(trainX))
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
```

```
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 3.13: Example of batch gradient descent for an MLP on the blobs problem.

Running the example first reports the performance of the model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that performance was similar between the train and test sets with 81% and 83% respectively.

```
Train: 0.814, Test: 0.834
```

Listing 3.14: Example output from batch gradient descent for an MLP on the blobs problem.

A line plot of model loss and classification accuracy on the train (blue) and test (orange) dataset is created. We can see that the model is relatively slow to learn this problem, converging on a solution after about 100 epochs after which changes in model performance are minor.

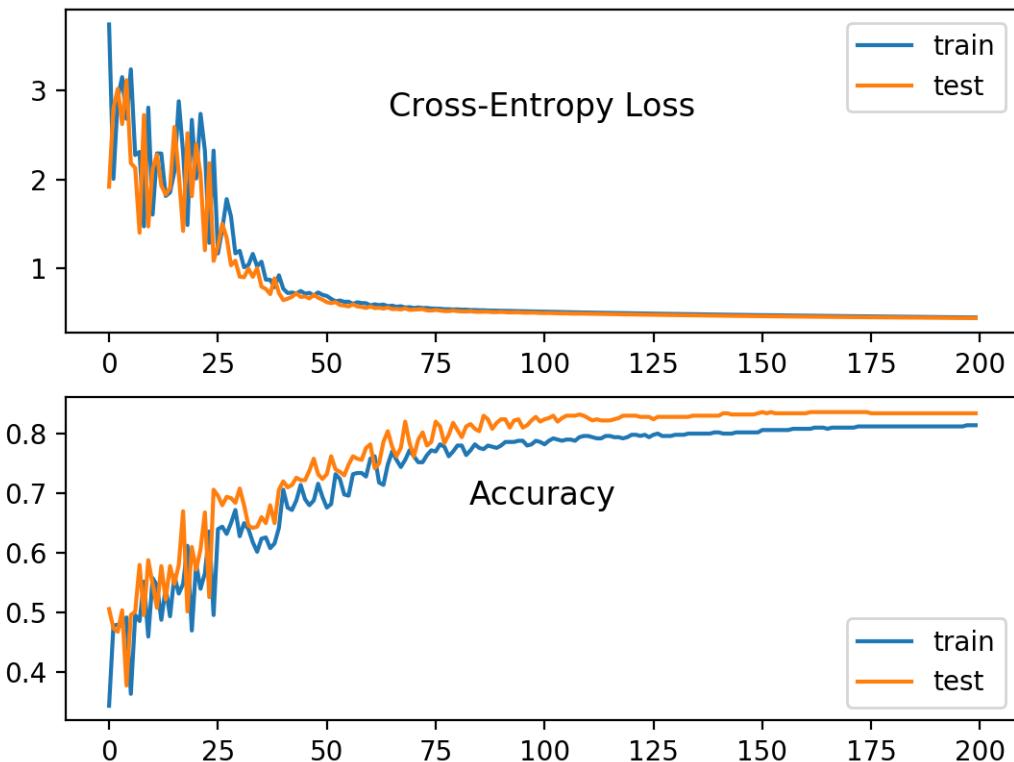


Figure 3.2: Line Plot of Classification Accuracy on Train and Tests Sets of an MLP Fit With Batch Gradient Descent.

3.3.3 MLP Fit With Stochastic Gradient Descent

The example of batch gradient descent from the previous section can be updated to instead use stochastic gradient descent. This requires changing the batch size from the size of the training dataset to 1.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0,
batch_size=1)
```

Listing 3.15: Example of configuring stochastic gradient descent.

Stochastic gradient descent requires that the model make a prediction and have the weights updated for each training example. This has the effect of dramatically slowing down the training process as compared to batch gradient descent. The expectation of this change is that the model learns faster (e.g. in terms of the learning curve) and that changes to the model are noisy, resulting, in turn, in noisy performance over training epochs. The complete example with this change is listed below.

```
# mlp for the blobs problem with stochastic gradient descent
from sklearn.datasets import make_blobs
from keras.layers import Dense
```

```

from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0,
    batch_size=1)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 3.16: Example of stochastic gradient descent for an MLP on the blobs problem.

Running the example first reports the performance of the model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that performance was similar between the train and test sets, around 50% accuracy, but was dramatically worse (about 30 percentage points) than using batch gradient descent. At least for this problem and the chosen model and model configuration, stochastic (online) gradient descent is not appropriate.

Train: 0.508, Test: 0.530

Listing 3.17: Example output from stochastic gradient descent for an MLP on the blobs problem.

A line plot of model loss and classification accuracy on the train (blue) and test (orange) dataset is created. The plot shows the unstable nature of the training process with the chosen configuration. The poor performance and erratic changes to the model suggest that the learning rate used to update weights after each training example may be too large and that a smaller learning rate may make the learning process more stable.

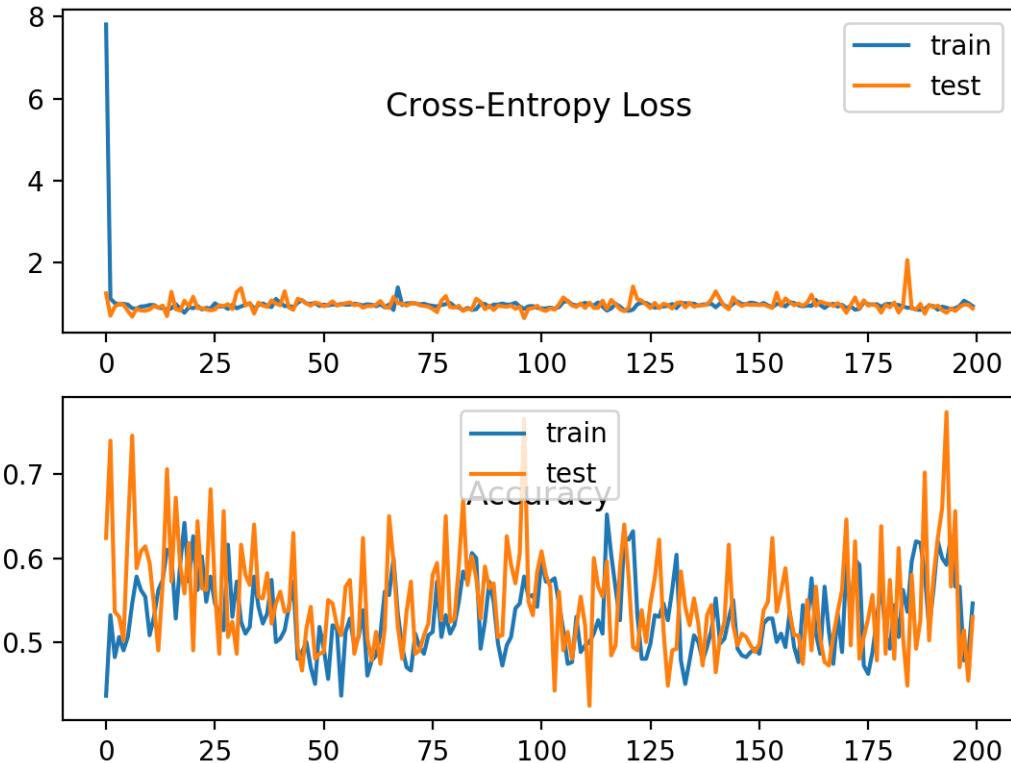


Figure 3.3: Line Plot of Classification Accuracy on Train and Tests Sets of an MLP Fit With Stochastic Gradient Descent.

We can test this by re-running the model fit with stochastic gradient descent and a smaller learning rate. For example, we can drop the learning rate by an order of magnitude from 0.01 to 0.001.

```
# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 3.18: Example of changing the learning rate for stochastic gradient descent.

The full code listing with this change is provided below for completeness.

```
# mlp for the blobs problem with stochastic gradient descent (smaller learning rate)
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
```

```

from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))
# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0,
    batch_size=1)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 3.19: Example of stochastic gradient descent with smaller learning rate for an MLP on the blobs problem.

Running this example tells a very different story.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The reported performance is greatly improved, achieving classification accuracy on the train and test sets on par with fit using batch gradient descent.

Train: 0.830, Test: 0.824

Listing 3.20: Example output from stochastic gradient descent with smaller learning rate for an MLP on the blobs problem.

The line plot shows the expected behavior. Namely, that the model rapidly learns the problem as compared to batch gradient descent, leaping up to about 80% accuracy in about 25 epochs rather than the 100 epochs seen when using batch gradient descent. We could have stopped training at epoch 50 instead of epoch 200 due to the faster training. This is not surprising. With batch gradient descent, 100 epochs involved 100 estimates of error and 100 weight updates. In stochastic gradient descent, 25 epochs involved (500×25) or 12,500 weight updates, providing more than 10-times more feedback, albeit more noisy feedback, about how to improve the model.

The line plot also shows that train and test performance remain comparable during training, as compared to the dynamics with batch gradient descent where the performance on the test set was slightly better and remained so throughout training. Unlike batch gradient descent, we can see that the noisy updates result in noisy performance throughout the duration of training. This variance in the model means that it may be challenging to choose which model to use as the final model, as opposed to batch gradient descent where performance is stabilized because the model has converged.

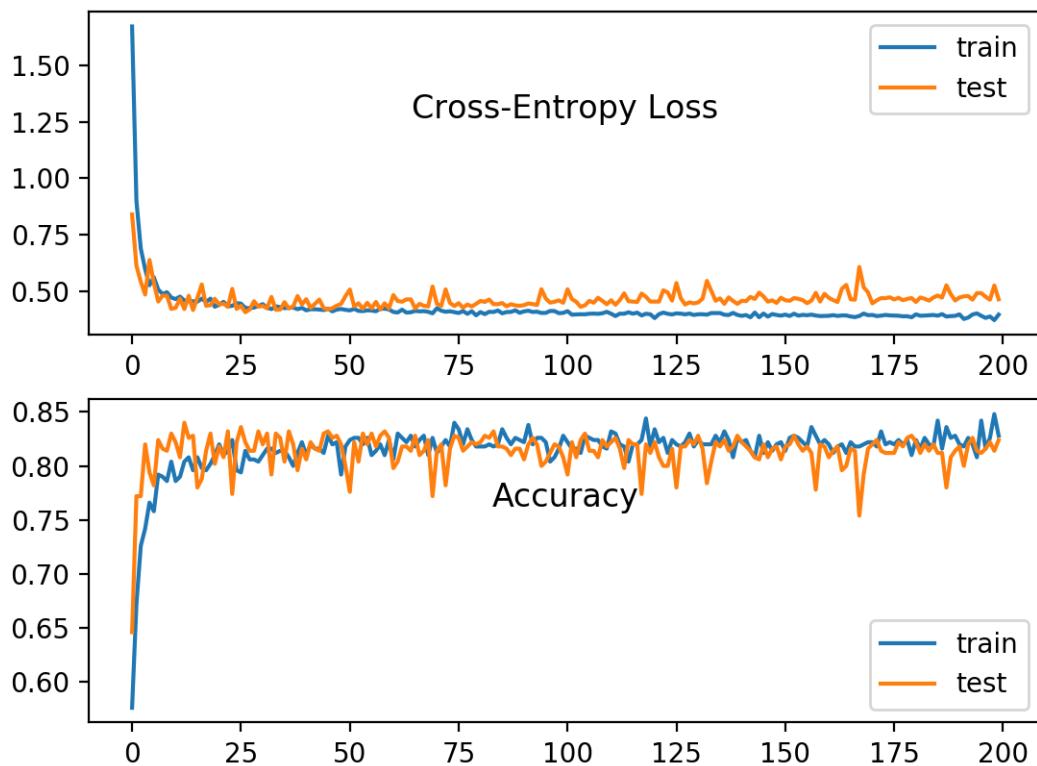


Figure 3.4: Line Plot of Classification Accuracy on Train and Tests Sets of an MLP Fit With Stochastic Gradient Descent and Smaller Learning Rate.

This example highlights the important relationship between batch size and the learning rate. Namely, more noisy updates to the model require a smaller learning rate, whereas less noisy

more accurate estimates of the error gradient may be applied to the model more liberally. We can summarize this as follows:

- **Batch Gradient Descent:** Use a relatively larger learning rate and more training epochs.
- **Stochastic Gradient Descent:** Use a relatively smaller learning rate and fewer training epochs.

Mini-batch gradient descent provides an alternative approach.

3.3.4 MLP Fit With Minibatch Gradient Descent

An alternative to using stochastic gradient descent and tuning the learning rate is to hold the learning rate constant and to change the batch size. In effect, it means that we specify the rate of learning or amount of change to apply to the weights each time we estimate the error gradient, but to vary the accuracy of the gradient based on the number of samples used to estimate it. Holding the learning rate at 0.01 as we did with batch gradient descent, we can set the batch size to 32, a widely adopted default batch size.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0,
batch_size=32)
```

Listing 3.21: Example of configuring the model for minibatch gradient descent.

We would expect to get some of the benefits of stochastic gradient descent with a larger learning rate. The complete example with this modification is listed below.

```
# mlp for the blobs problem with minibatch gradient descent
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0,
batch_size=32)
# evaluate the model
```

```

_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 3.22: Example of minibatch gradient descent for an MLP on the blobs problem.

Running the example reports similar performance on both train and test sets, comparable with batch gradient descent and stochastic gradient descent after we reduced the learning rate.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

Train: 0.808, Test: 0.826

Listing 3.23: Example output from minibatch gradient descent for an MLP on the blobs problem.

The line plot shows the dynamics of both stochastic and batch gradient descent. Specifically, the model learns fast and has noisy updates but also stabilizes more towards the end of the run, more so than stochastic gradient descent. Holding learning rate constant and varying the batch size allows you to dial in the best of both approaches.

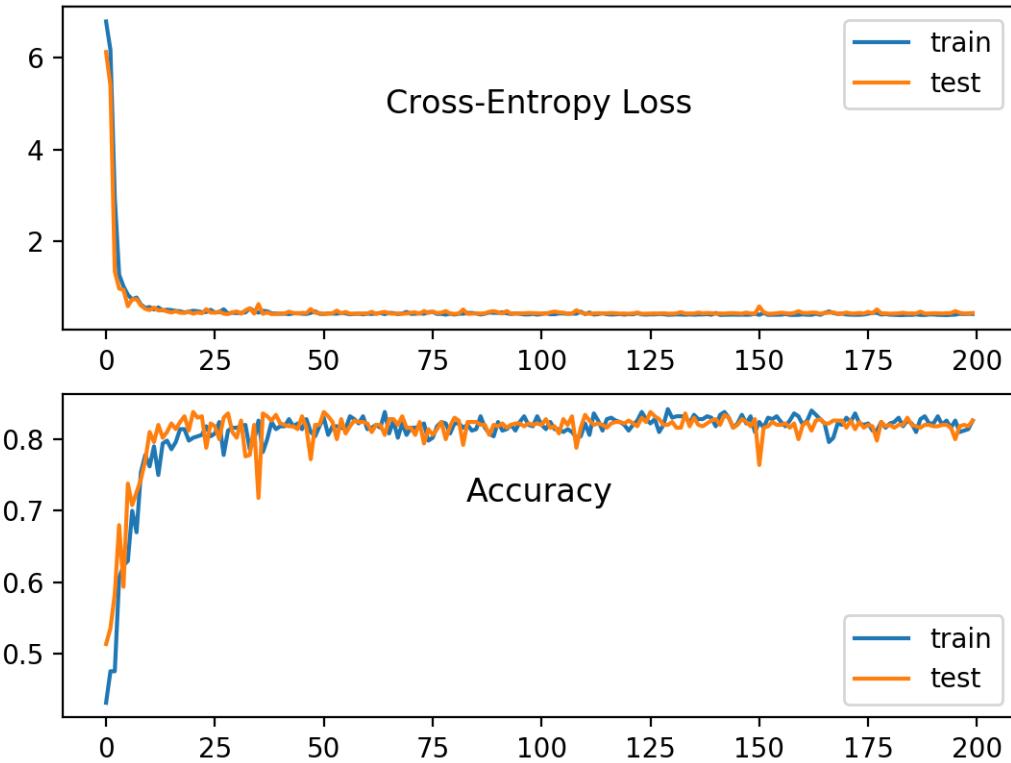


Figure 3.5: Line Plot of Classification Accuracy on Train and Tests Sets of an MLP Fit With Minibatch Gradient Descent.

3.3.5 Effect of Batch Size on Model Behavior

We can refit the model with different batch sizes and review the impact the change in batch size has on the speed of learning, stability during learning, and on the final result. First, we can clean up the code and create a function to prepare the dataset.

```
# prepare train and test dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy
```

Listing 3.24: Example of a function for preparing the data for modeling.

Next, we can create a function to fit a model on the problem with a given batch size and plot the learning curves of classification accuracy on the train and test datasets.

```
# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, n_batch):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200,
                         verbose=0, batch_size=n_batch)
    # plot learning curves
    pyplot.plot(history.history['accuracy'], label='train')
    pyplot.plot(history.history['val_accuracy'], label='test')
    pyplot.title('batch=' + str(n_batch), pad=-40)
```

Listing 3.25: Example of a function for evaluating a model with a given batch size.

Finally, we can evaluate the model behavior with a suite of different batch sizes while holding everything else about the model constant, including the learning rate.

```
# prepare dataset
trainX, trainy, testX, testy = prepare_data()
# create learning curves for different batch sizes
batch_sizes = [4, 8, 16, 32, 64, 128, 256, 450]
for i in range(len(batch_sizes)):
    # determine the plot number
    plot_no = 420 + (i+1)
    pyplot.subplot(plot_no)
    # fit model and plot learning curves for a batch size
    fit_model(trainX, trainy, testX, testy, batch_sizes[i])
# show learning curves
pyplot.show()
```

Listing 3.26: Example of evaluating models with different batch sizes.

The result will be a figure with eight plots of model behavior with eight different batch sizes. Tying this together, the complete example is listed below.

```
# mlp for the blobs problem with minibatch gradient descent with varied batch size
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare train and test dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
```

```

trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
return trainX, trainy, testX, testy

# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, n_batch):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200,
        verbose=0, batch_size=n_batch)
    # plot learning curves
    pyplot.plot(history.history['accuracy'], label='train')
    pyplot.plot(history.history['val_accuracy'], label='test')
    pyplot.title('batch=' + str(n_batch), pad=-40)

# prepare dataset
trainX, trainy, testX, testy = prepare_data()
# create learning curves for different batch sizes
batch_sizes = [4, 8, 16, 32, 64, 128, 256, 450]
for i in range(len(batch_sizes)):
    # determine the plot number
    plot_no = 420 + (i+1)
    pyplot.subplot(plot_no)
    # fit model and plot learning curves for a batch size
    fit_model(trainX, trainy, testX, testy, batch_sizes[i])
# show learning curves
pyplot.show()

```

Listing 3.27: Example of evaluating and comparing models with different batch sizes.

Running the example creates a figure with eight line plots showing the classification accuracy on the train and test sets of models with different batch sizes when using minibatch gradient descent.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The plots show that small batch results generally in rapid learning but a volatile learning process with higher variance in the classification accuracy. Larger batch sizes slow down the learning process (in terms of the learning curves) but the final stages result in a convergence to a more stable model exemplified by lower variance in classification accuracy.

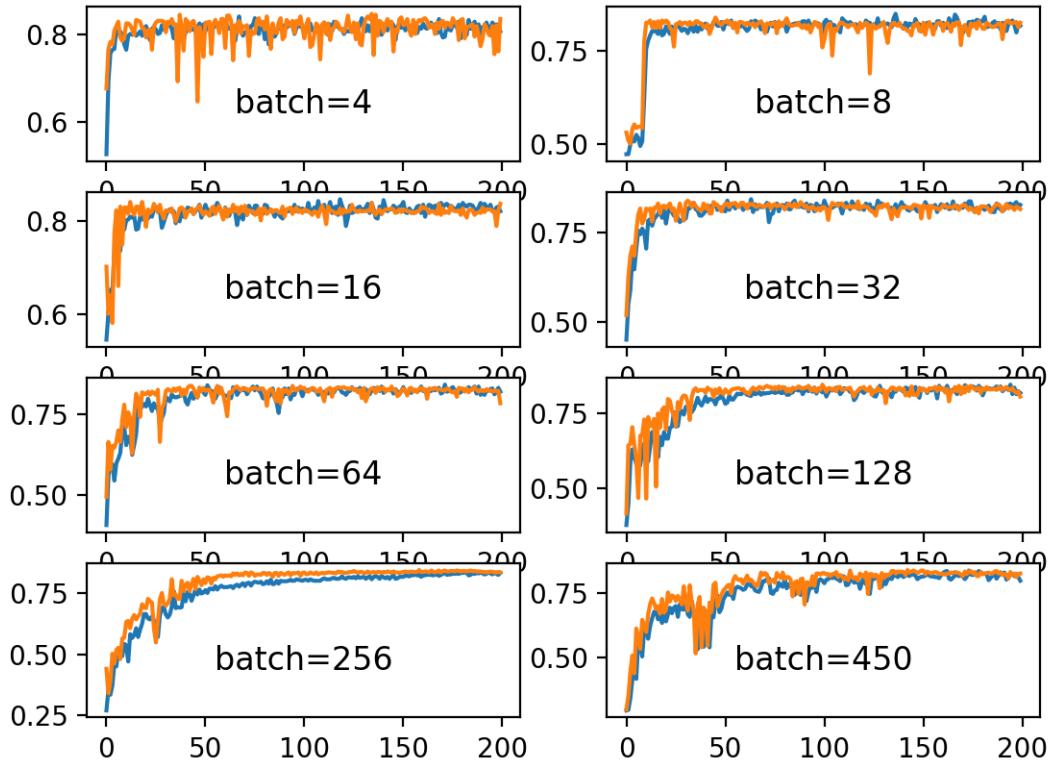


Figure 3.6: Line Plots of Classification Accuracy on Train and Test Datasets With Different Batch Sizes.

3.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Vary Learning Rate.** Study the effect of different learning rate values on a logarithmic scale with stochastic (online) gradient descent.
- **Vary Epochs.** Study the number of epochs required for convergence as the batch size is increased to the size of the training dataset with minibatch gradient descent.

If you explore any of these extensions, I'd love to know.

3.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

3.5.1 Books

- Section 8.1.3: Batch and Minibatch Algorithms, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>

3.5.2 Papers

- *Revisiting Small Batch Training for Deep Neural Networks*, 2018.
<https://arxiv.org/abs/1804.07612>
- *Practical recommendations for gradient-based training of deep architectures*, 2012.
<https://arxiv.org/abs/1206.5533>

3.5.3 Articles

- Stochastic gradient descent, Wikipedia.
https://en.wikipedia.org/wiki/Stochastic_gradient_descent

3.6 Summary

In this tutorial, you discovered three different flavors of gradient descent and how to explore and diagnose the effect of batch size on the learning process. Specifically, you learned:

- Batch size controls the accuracy of the estimate of the error gradient when training neural networks.
- Batch, Stochastic, and Minibatch gradient descent are the three main flavors of the learning algorithm.
- There is a tension between batch size and the speed and stability of the learning process.

3.6.1 Next

In the next tutorial, you will discover how the loss function controls the nature of the optimization problem that is being solved by stochastic gradient descent.

Chapter 4

Configure What to Optimize with Loss Functions

Neural networks are trained using stochastic gradient descent and require that you choose a loss function when designing and configuring your model. There are many loss functions to choose from and it can be challenging to know what to choose, or even what a loss function is and the role it plays when training a neural network. In this tutorial, you will discover the role of loss and loss functions in training deep learning neural networks and how to choose the right loss function for your predictive modeling problems. After reading this tutorial, you will know:

- Neural networks are trained using an optimization process that requires a loss function to calculate the model error.
- Maximum Likelihood provides a framework for choosing a loss function when training neural networks and machine learning models in general.
- Cross-entropy and mean squared error are the two main types of loss functions to use when training neural network models.

Let's get started.

4.1 Loss Functions

In this section you will discover loss functions and how they are used to define the nature of the optimization problem that is solved when adapting neural network weights to a training dataset.

4.1.1 Neural Network Learning as Optimization

A deep learning neural network learns to map a set of inputs to a set of outputs from training data. We cannot calculate the perfect weights for a neural network; there are too many unknowns. Instead, the problem of learning is cast as a search or optimization problem and an algorithm is used to navigate the space of possible sets of weights the model may use in order to make good or good enough predictions. Typically, a neural network model is trained using the stochastic gradient descent optimization algorithm and weights are updated using the backpropagation of error algorithm.

The *gradient* in gradient descent refers to an error gradient. The model with a given set of weights is used to make predictions and the error for those predictions is calculated. The gradient descent algorithm seeks to change the weights so that the next evaluation reduces the error, meaning the optimization algorithm is navigating down the gradient (or slope) of error. Now that we know that training neural nets solves an optimization problem, we can look at how the error of a given set of weights is calculated.

4.1.2 What Is a Loss Function and Loss?

In the context of an optimization algorithm, the function used to evaluate a candidate solution (i.e. a set of weights) is referred to as the objective function. We may seek to maximize or minimize the objective function, meaning that we are searching for a candidate solution that has the highest or lowest score respectively. Typically, with neural networks, we seek to minimize the error. As such, the objective function is often referred to as a cost function or a loss function and the value calculated by the loss function is referred to as simply *loss*.

The function we want to minimize or maximize is called the objective function or criterion. When we are minimizing it, we may also call it the cost function, loss function, or error function.

— Page 82, *Deep Learning*, 2016.

The cost or loss function has an important job in that it must faithfully distill all aspects of the model down into a single number in such a way that improvements in that number are a sign of a better model.

The cost function reduces all the various good and bad aspects of a possibly complex system down to a single number, a scalar value, which allows candidate solutions to be ranked and compared.

— Page 155, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

In calculating the error of the model during the optimization process, a loss function must be chosen. This can be a challenging problem as the function must capture the properties of the problem and be motivated by concerns that are important to the project and stakeholders.

It is important, therefore, that the function faithfully represent our design goals. If we choose a poor error function and obtain unsatisfactory results, the fault is ours for badly specifying the goal of the search.

— Page 155, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

Now that we are familiar with the loss function and loss, we need to know what functions to use.

4.1.3 Maximum Likelihood

There are many functions that could be used to estimate the error of a set of weights in a neural network. We prefer a function where the space of candidate solutions maps onto a smooth (but high-dimensional) landscape that the optimization algorithm can reasonably navigate via iterative updates to the model weights. Maximum likelihood estimation, or MLE, is a framework for inference for finding the best statistical estimates of parameters from historical training data: exactly what we are trying to do with the neural network.

Maximum likelihood seeks to find the optimum values for the parameters by maximizing a likelihood function derived from the training data.

— Page 39, *Neural Networks for Pattern Recognition*, 1995.

We have a training dataset with one or more input variables and we require a model to estimate model weight parameters that best map examples of the inputs to the output or target variable. Given input, the model is trying to make predictions that match the data distribution of the target variable. Under maximum likelihood, a loss function estimates how closely the distribution of predictions made by a model matches the distribution of target variables in the training data.

One way to interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution [...] defined by the training set and the model distribution, with the degree of dissimilarity between the two measured by the KL divergence. [...] Minimizing this KL divergence corresponds exactly to minimizing the cross-entropy between the distributions.

— Page 132, *Deep Learning*, 2016.

A benefit of using maximum likelihood as a framework for estimating the model parameters (weights) for neural networks and in machine learning in general is that as the number of examples in the training dataset is increased, the estimate of the model parameters improves. This is called the property of *consistency*.

Under appropriate conditions, the maximum likelihood estimator has the property of consistency [...], meaning that as the number of training examples approaches infinity, the maximum likelihood estimate of a parameter converges to the true value of the parameter.

— Page 134, *Deep Learning*, 2016.

Now that we are familiar with the general approach of maximum likelihood, we can look at the error function.

4.1.4 Maximum Likelihood and Cross-Entropy

Under the maximum likelihood framework, the error between two probability distributions is measured using cross-entropy. When modeling a classification problem where we are interested in mapping input variables to a class label, we can model the problem as predicting the probability of an example belonging to each class. In a binary classification problem, there would be two classes, so we may predict the probability of the example belonging to the first class. In the case of multiple-class classification, we can predict a probability for the example belonging to each of the classes.

In the training dataset, the probability of an example belonging to a given class would be 1 or 0, as each sample in the training dataset is a known example from the domain. We know the answer. Therefore, under maximum likelihood estimation, we would seek a set of model weights that minimize the difference between the model's predicted probability distribution given the dataset and the distribution of probabilities in the training dataset. This is called the cross-entropy.

In most cases, our parametric model defines a distribution [...] and we simply use the principle of maximum likelihood. This means we use the cross-entropy between the training data and the model's predictions as the cost function.

— Page 178, *Deep Learning*, 2016.

Technically, cross-entropy comes from the field of information theory and has the unit of *bits*. It is used to estimate the difference between an estimated and a predicted probability distribution. In the case of regression problems where a quantity is predicted, it is common to use the mean squared error (MSE) loss function instead.

A few basic functions are very commonly used. The mean squared error is popular for function approximation (regression) problems [...] The cross-entropy error function is often used for classification problems when outputs are interpreted as probabilities of membership in an indicated class.

— Page 155-156, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

Nevertheless, under the framework of maximum likelihood estimation and assuming a Gaussian distribution for the target variable, mean squared error can be considered the cross-entropy between the distribution of the model predictions and the distribution of the target variable.

Many authors use the term “cross-entropy” to identify specifically the negative log-likelihood of a Bernoulli or softmax distribution, but that is a misnomer. Any loss consisting of a negative log-likelihood is a cross-entropy between the empirical distribution defined by the training set and the probability distribution defined by the model. For example, mean squared error is the cross-entropy between the empirical distribution and a Gaussian model.

— Page 132, *Deep Learning*, 2016.

Therefore, when using the framework of maximum likelihood estimation, we will implement a cross-entropy loss function, which often in practice means a cross-entropy loss function for classification problems and a mean squared error loss function for regression problems. Almost universally, deep learning neural networks are trained under the framework of maximum likelihood using cross-entropy as the loss function.

Most modern neural networks are trained using maximum likelihood. This means that the cost function is [...] described as the cross-entropy between the training data and the model distribution.

— Pages 178-179, *Deep Learning*, 2016.

In fact, adopting this framework may be considered a milestone in deep learning, as before being fully formalized, it was sometimes common for neural networks for classification to use a mean squared error loss function.

One of these algorithmic changes was the replacement of mean squared error with the cross-entropy family of loss functions. Mean squared error was popular in the 1980s and 1990s, but was gradually replaced by cross-entropy losses and the principle of maximum likelihood as ideas spread between the statistics community and the machine learning community.

— Page 226, *Deep Learning*, 2016.

The maximum likelihood approach was adopted almost universally not just because of the theoretical framework, but primarily because of the results it produces. Specifically, neural networks for classification that use a sigmoid or softmax activation function in the output layer learn faster and more robustly using a cross-entropy loss function.

The use of cross-entropy losses greatly improved the performance of models with sigmoid and softmax outputs, which had previously suffered from saturation and slow learning when using the mean squared error loss.

— Page 226, *Deep Learning*, 2016.

4.1.5 What Loss Function to Use?

We can summarize the previous section and directly suggest the loss functions that you should use under a framework of maximum likelihood. Importantly, the choice of loss function is directly related to the activation function used in the output layer of your neural network. These two design elements are connected. Think of the configuration of the output layer as a choice about the framing of your prediction problem, and the choice of the loss function as the way to calculate the error for a given framing of your problem.

The choice of cost function is tightly coupled with the choice of output unit. Most of the time, we simply use the cross-entropy between the data distribution and the model distribution. The choice of how to represent the output then determines the form of the cross-entropy function.

— Page 181, *Deep Learning*, 2016.

We will review best practice or default values for each problem type with regard to the output layer and loss function.

Regression Problem

A problem where you predict a real-value quantity.

- **Output Layer Configuration:** One node with a linear activation unit.
- **Loss Function:** Mean Squared Error (MSE).

Binary Classification Problem

A problem where you classify an example as belonging to one of two classes. The problem is framed as predicting the likelihood of an example belonging to class one, e.g. the class that you assign the integer value 1, whereas the other class is assigned the value 0.

- **Output Layer Configuration:** One node with a sigmoid activation unit.
- **Loss Function:** Cross-Entropy, also referred to as Logarithmic loss.

Multiclass Classification Problem

A problem where you classify an example as belonging to one of more than two classes. The problem is framed as predicting the likelihood of an example belonging to each class.

- **Output Layer Configuration:** One node for each class using the softmax activation function.
- **Loss Function:** Cross-Entropy, also referred to as Logarithmic loss.

4.1.6 How to Implement Loss Functions

In order to make the loss functions concrete, this section explains how each of the main types of loss function works and how to calculate the score in Python.

Mean Squared Error Loss

Mean Squared Error loss, or MSE for short, is calculated as the average of the squared differences between the predicted and actual values. The result is always positive regardless of the sign of the predicted and actual values and a perfect value is 0.0. The loss value is minimized, although it can be used in a maximization optimization process by making the score negative. The Python function below provides a pseudocode-like working implementation of a function for calculating the mean squared error for a list of actual and a list of predicted real-valued quantities.

```
# calculate mean squared error
def mean_squared_error(actual, predicted):
    sum_square_error = 0.0
    for i in range(len(actual)):
        sum_square_error += (actual[i] - predicted[i])**2.0
    mean_square_error = 1.0 / len(actual) * sum_square_error
    return mean_square_error
```

Listing 4.1: Example of a function for implementing mean squared error.

For an efficient implementation, I'd encourage you to use the scikit-learn `mean_squared_error()` function¹.

Cross-Entropy Loss (or Log Loss)

Cross-entropy loss is often simply referred to as *cross-entropy*, *logarithmic loss*, *logistic loss*, or *log loss* for short. Each predicted probability is compared to the actual class output value (0 or 1) and a score is calculated that penalizes the probability based on the distance from the expected value. The penalty is logarithmic, offering a small score for small differences (0.1 or 0.2) and enormous score for a large difference (0.9 or 1.0).

Cross-entropy loss is minimized, where smaller values represent a better model than larger values. A model that predicts perfect probabilities has a cross-entropy or log loss of 0.0. Cross-entropy for a binary or two class prediction problem is actually calculated as the average cross-entropy across all examples. The Python function below provides a pseudocode-like working implementation of a function for calculating the cross-entropy for a list of actual 0 and 1 values compared to predicted probabilities for the class 1.

```
from math import log

# calculate binary cross-entropy
def binary_cross_entropy(actual, predicted):
    sum_score = 0.0
    for i in range(len(actual)):
        sum_score += actual[i] * log(1e-15 + predicted[i])
    mean_sum_score = 1.0 / len(actual) * sum_score
    return -mean_sum_score
```

Listing 4.2: Example of a function for implementing binary cross-entropy.

Note, we add a very small value (in this case 1E-15) to the predicted probabilities to avoid ever calculating the log of 0.0. This means that in practice, the best possible loss will be a value very close to zero, but not exactly zero. Cross-entropy can be calculated for multiple-class classification. The classes have been one hot encoded, meaning that there is a binary feature for each class value and the predictions must have predicted probabilities for each of the classes. The cross-entropy is then summed across each binary feature and averaged across all examples in the dataset. The Python function below provides a pseudocode-like working implementation of a function for calculating the cross-entropy for a list of actual one hot encoded values compared to predicted probabilities for each class.

```
from math import log

# calculate categorical cross-entropy
def categorical_cross_entropy(actual, predicted):
    sum_score = 0.0
    for i in range(len(actual)):
        for j in range(len(actual[i])):
            sum_score += actual[i][j] * log(1e-15 + predicted[i][j])
    mean_sum_score = 1.0 / len(actual) * sum_score
    return -mean_sum_score
```

¹https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html

Listing 4.3: Example of a function for implementing multiclass cross-entropy.

For an efficient implementation, I'd encourage you to use the scikit-learn `log_loss()` function².

4.1.7 Loss Functions and Reported Model Performance

Given the framework of maximum likelihood, we want to use a cross-entropy or mean squared error loss function in general with stochastic gradient descent. Nevertheless, we may or may not want to report the performance of the model using the loss function. For example, logarithmic loss is challenging to interpret, especially for non-machine learning practitioner stakeholders. The same can be said for the mean squared error. Instead, it may be more important to report the accuracy and root mean squared error for models used for classification and regression respectively.

It may also be desirable to choose models based on these metrics instead of loss. This is an important consideration, as the model with the minimum loss may not be the model with best metric that is important to project stakeholders. A good division to consider is to use the loss to evaluate and diagnose how well the model is learning. This includes all of the considerations of the optimization process, such as overfitting, underfitting, and convergence. An alternate metric can then be chosen that has meaning to the project stakeholders to both evaluate model performance and perform model selection.

- **Loss:** Used to evaluate and diagnose model optimization only.
- **Metric:** Used to evaluate and choose models in the context of the project.

The same metric can be used for both concerns but it is more likely that the concerns of the optimization process will differ from the goals of the project and different scores will be required. Nevertheless, it is often the case that improving the loss improves or, at worst, has no effect on the metric of interest.

4.2 Regression Loss Functions Case Study

A regression predictive modeling problem involves predicting a real-valued quantity. In this section, we will investigate loss functions that are appropriate for regression predictive modeling problems. As the context for this investigation, we will use a standard regression problem generator provided by the scikit-learn library in the `make_regression()` function. This function will generate examples from a simple regression problem with a given number of input variables, statistical noise, and other properties. We will use this function to define a problem that has 20 input features; 10 of the features will be meaningful and 10 will not be relevant. A total of 1,000 examples will be randomly generated. The pseudorandom number generator will be fixed to ensure that we get the same 1,000 examples each time the code is run.

²https://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html

```
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
```

Listing 4.4: Example of generating samples for the regression problem.

Neural networks generally perform better when the real-valued input and output variables are scaled to a sensible range. For this problem, each of the input variables and the target variable have a Gaussian distribution; therefore, standardizing the data in this case is desirable. We can achieve this using the `StandardScaler` transformer class also from the scikit-learn library. On a real problem, we would prepare the scaler on the training dataset and apply it to the train and test sets, but for simplicity, we will scale all of the data together before splitting into train and test sets.

```
# standardize dataset
X = StandardScaler().fit_transform(X)
y = StandardScaler().fit_transform(y.reshape(len(y),1))[:,0]
```

Listing 4.5: Example of standardizing data samples.

Once scaled, the data will be split evenly into train and test sets.

```
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 4.6: Example of splitting data into train and test sets.

A small Multilayer Perceptron (MLP) model will be defined to address this problem and provide the basis for exploring different loss functions. The model will expect 20 features as input as defined by the problem. The model will have one hidden layer with 25 nodes and will use the rectified linear activation function. The output layer will have 1 node, given the one real-value to be predicted, and will use the linear activation function.

```
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
```

Listing 4.7: Example of defining the MLP model.

The model will be fit with stochastic gradient descent with a learning rate of 0.01 and a momentum of 0.9, both sensible default values. Training will be performed for 100 epochs and the test set will be evaluated at the end of each epoch so that we can plot learning curves at the end of the run.

```
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='...', optimizer=opt)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
```

Listing 4.8: Example of compiling and fitting the MLP model.

Now that we have the basis of a problem and model, we can take a look evaluating three common loss functions that are appropriate for a regression predictive modeling problem. Although an MLP is used in these examples, the same loss functions can be used when training CNN and RNN models for regression.

4.2.1 Mean Squared Error Loss

The Mean Squared Error, or MSE, loss is the default loss to use for regression problems. Mathematically, it is the preferred loss function under the inference framework of maximum likelihood if the distribution of the target variable is Gaussian. It is the loss function to be evaluated first and only changed if you have a good reason. Mean squared error is calculated as the average of the squared differences between the predicted and actual values. The result is always positive regardless of the sign of the predicted and actual values and a perfect value is 0.0. The squaring means that larger mistakes result in more error than smaller mistakes, meaning that the model is punished for making larger mistakes. The mean squared error loss function can be used in Keras by specifying ‘mse’ or ‘mean_squared_error’ as the loss function when compiling the model.

```
model.compile(loss='mean_squared_error')
```

Listing 4.9: Example of using mean squared error loss function.

It is recommended that the output layer has one node for the target variable and the linear activation function is used.

```
model.add(Dense(1, activation='linear'))
```

Listing 4.10: Example of using linear activation function.

A complete example of demonstrating an MLP on the described regression problem is listed below.

```
# mlp for regression with mse loss function
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# standardize dataset
X = StandardScaler().fit_transform(X)
y = StandardScaler().fit_transform(y.reshape(len(y),1))[:,0]
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='mean_squared_error', optimizer=opt)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
```

```
pyplot.title('Mean Squared Error Loss')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 4.11: Example of mean squared error loss for MLP on the regression problem.

Running the example first prints the mean squared error for the model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model learned the problem achieving zero error, at least to three decimal places.

```
Train: 0.001, Test: 0.002
```

Listing 4.12: Example output from mean squared error loss for MLP on the regression problem.

A line plot is also created showing the mean squared error loss over the training epochs for both the train (blue) and test (orange) sets. We can see that the model converged reasonably quickly and both train and test performance remained equivalent. The performance and convergence behavior of the model suggest that mean squared error is a good match for a neural network learning this problem.

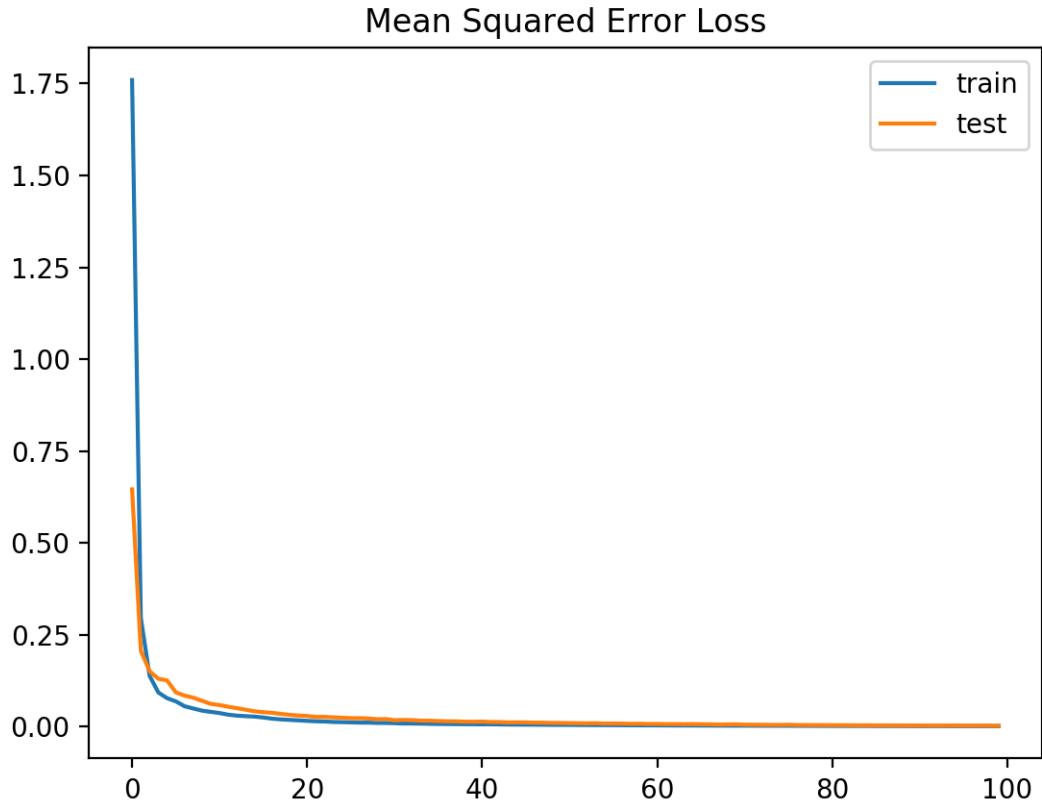


Figure 4.1: Line plot of Mean Squared Error Loss over Training Epochs When Optimizing the Mean Squared Error Loss Function.

4.2.2 Mean Squared Logarithmic Error Loss

There may be regression problems in which the target value has a spread of values and when predicting a large value, you may not want to punish a model as heavily as mean squared error. Instead, you can first calculate the natural logarithm of each of the predicted values, then calculate the mean squared error. This is called the Mean Squared Logarithmic Error loss, or MSLE for short. It has the effect of relaxing the punishing effect of large differences in large predicted values.

As a loss measure, it may be more appropriate when the model is predicting unscaled quantities directly. Nevertheless, we can demonstrate this loss function using our simple regression problem. The model can be updated to use the '`mean_squared_logarithmic_error`' loss function and keep the same configuration for the output layer. We will also track the mean squared error as a metric when fitting the model so that we can use it as a measure of performance and plot the learning curve.

```
model.compile(loss='mean_squared_logarithmic_error', optimizer=opt, metrics=['mse'])
```

Listing 4.13: Example of using mean squared logistic error loss function.

The complete example of using the MSLE loss function is listed below.

```

# mlp for regression with msle loss function
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# standardize dataset
X = StandardScaler().fit_transform(X)
y = StandardScaler().fit_transform(y.reshape(len(y),1))[:,0]
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='mean_squared_logarithmic_error', optimizer=opt, metrics=['mse'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
_, train_mse = model.evaluate(trainX, trainy, verbose=0)
_, test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.subplot(211)
pyplot.title('Mean Squared Logarithmic Error Loss', pad=-20)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot mse during training
pyplot.subplot(212)
pyplot.title('Mean Squared Error', pad=-20)
pyplot.plot(history.history['mse'], label='train')
pyplot.plot(history.history['val_mse'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 4.14: Example of mean squared logistic error loss for MLP on the regression problem.

Running the example first prints the mean squared error for the model on the train and test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model resulted in slightly worse MSE on both the training and test dataset. It may not be a good fit for this problem as the distribution of the target variable is a standard Gaussian.

Train: 0.206, Test: 0.257

Listing 4.15: Example output from mean squared logistic error loss for MLP on the regression problem.

A line plot is also created showing the mean squared logistic error loss over the training epochs for both the train (blue) and test (orange) sets (top), and a similar plot for the mean squared error (bottom). We can see that the MSLE converged well over the 100 epochs algorithm; it appears that the MSE may be showing signs of overfitting the problem, dropping fast and starting to rise from epoch 20 onwards.

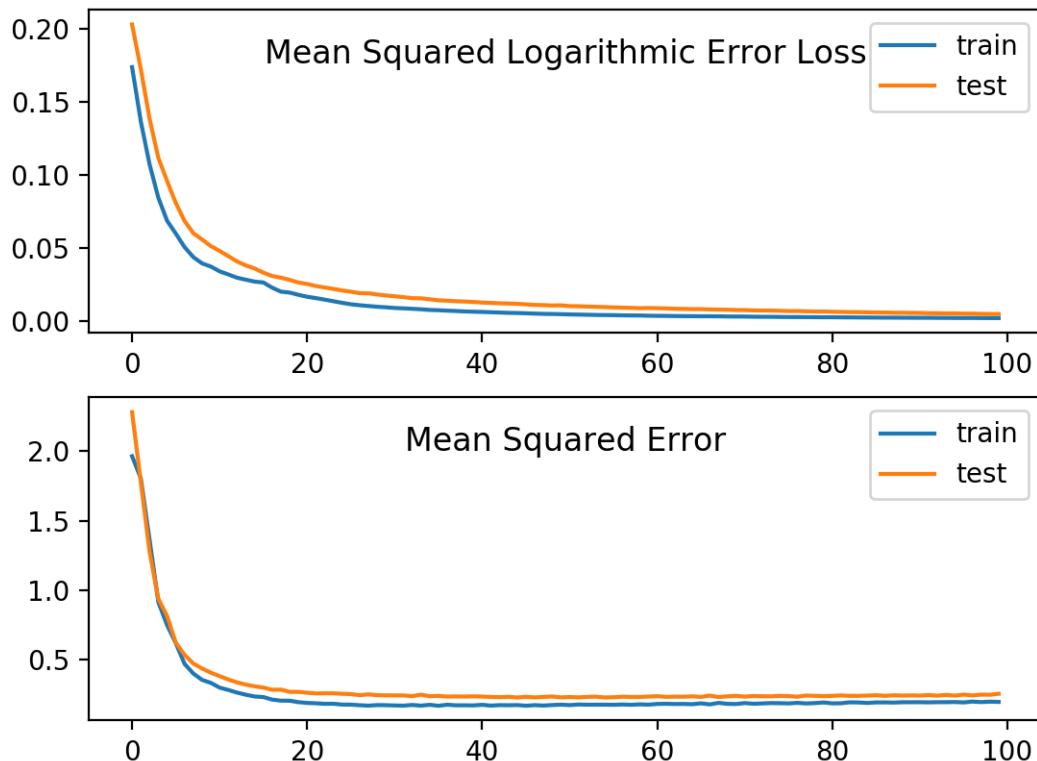


Figure 4.2: Line Plots of Mean Squared Logistic Error Loss and Mean Squared Error Over Training Epochs.

4.2.3 Mean Absolute Error Loss

On some regression problems, the distribution of the target variable may be mostly Gaussian, but may have outliers, e.g. large or small values far from the mean value. The Mean Absolute Error, or MAE, loss is an appropriate loss function in this case as it is more robust to outliers. It is calculated as the average of the absolute difference between the actual and predicted values. The model can be updated to use the ‘`mean_absolute_error`’ loss function and keep the same configuration for the output layer.

```
model.compile(loss='mean_absolute_error', optimizer=opt, metrics=['mse'])
```

Listing 4.16: Example of using mean absolute error loss function.

The complete example using the mean absolute error as the loss function on the regression test problem is listed below.

```
# mlp for regression with mae loss function
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# standardize dataset
X = StandardScaler().fit_transform(X)
y = StandardScaler().fit_transform(y.reshape(len(y),1))[:,0]
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='mean_absolute_error', optimizer=opt, metrics=['mse'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
_, train_mse = model.evaluate(trainX, trainy, verbose=0)
_, test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.subplot(211)
pyplot.title('Mean Absolute Error Loss', pad=-20)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot mse during training
pyplot.subplot(212)
pyplot.title('Mean Squared Error', pad=-20)
pyplot.plot(history.history['mse'], label='train')
pyplot.plot(history.history['val_mse'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 4.17: Example of mean absolute error loss for MLP on the regression problem.

Running the example first prints the mean squared error for the model on the train and test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model learned the problem, achieving a near zero error, at least to three decimal places.

Train: 0.002, Test: 0.002

Listing 4.18: Example output from mean absolute error loss for MLP on the regression problem.

A line plot is also created showing the mean absolute error loss over the training epochs for both the train (blue) and test (orange) sets (top), and a similar plot for the mean squared error (bottom). In this case, we can see that MAE does converge but shows a bumpy course, although the dynamics of MSE don't appear greatly affected. We know that the target variable is a standard Gaussian with no large outliers, so MAE would not be a good fit in this case. It might be more appropriate on this problem if we did not scale the target variable first.

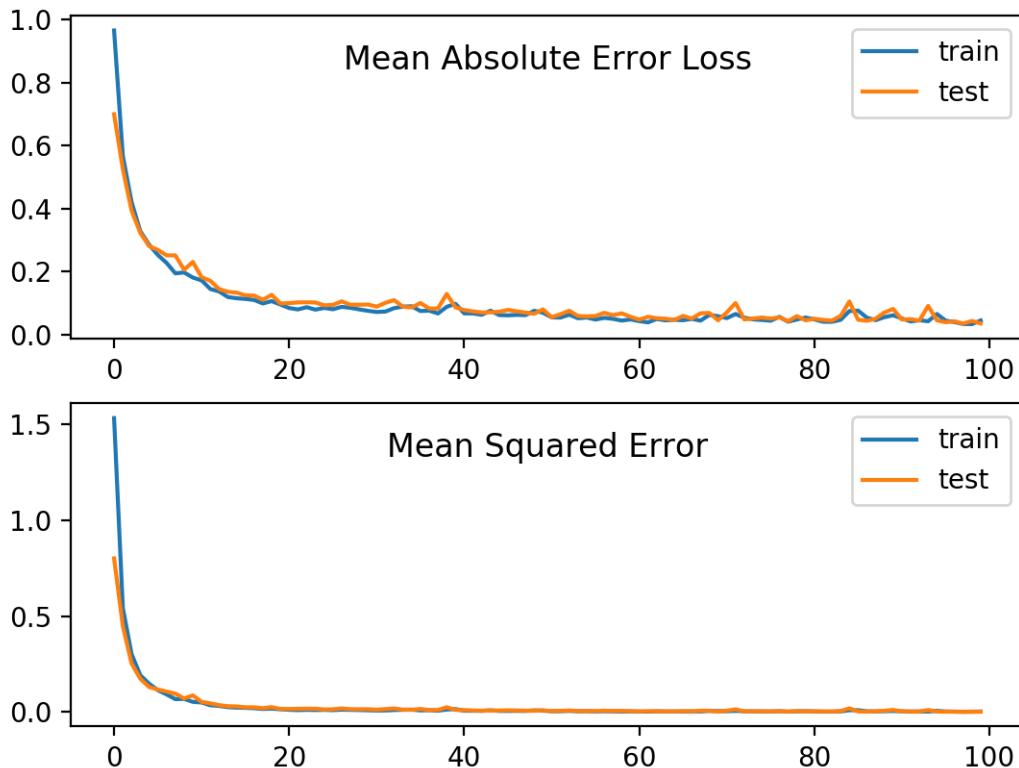


Figure 4.3: Line plots of Mean Absolute Error Loss and Mean Squared Error over Training Epochs.

4.3 Binary Classification Loss Functions Case Study

Binary classification predictive modeling problems are those where examples are assigned one of two labels. The problem is often framed as predicting a value of 0 or 1 for the first or second class and is often implemented as predicting the probability of the example belonging to class value 1. In this section, we will investigate loss functions that are appropriate for binary

classification predictive modeling problems. We will generate examples from the circles test problem in scikit-learn as the basis for this investigation. The circles problem involves samples drawn from two concentric circles on a two-dimensional plane, where points on the outer circle belong to class 0 and points for the inner circle belong to class 1. Statistical noise is added to the samples to add ambiguity and make the problem more challenging to learn. We will generate 1,000 examples and add 10% statistical noise. The pseudorandom number generator will be seeded with the same value to ensure that we always get the same 1,000 examples.

```
# generate circles
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
```

Listing 4.19: Example of generating samples from the two circles problem.

We can create a scatter plot of the dataset to get an idea of the problem we are modeling. The complete example is listed below.

```
# scatter plot of the circles dataset with points colored by class
from sklearn.datasets import make_circles
from numpy import where
from matplotlib import pyplot
# generate circles
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# select indices of points with each class label
for i in range(2):
    samples_ix = where(y == i)
    pyplot.scatter(X[samples_ix, 0], X[samples_ix, 1], label=str(i))
pyplot.legend()
pyplot.show()
```

Listing 4.20: Example of plotting samples from the two circles problem.

Running the example creates a scatter plot of the examples, where the input variables define the location of the point and the class value defines the color, with class 0 blue and class 1 orange.

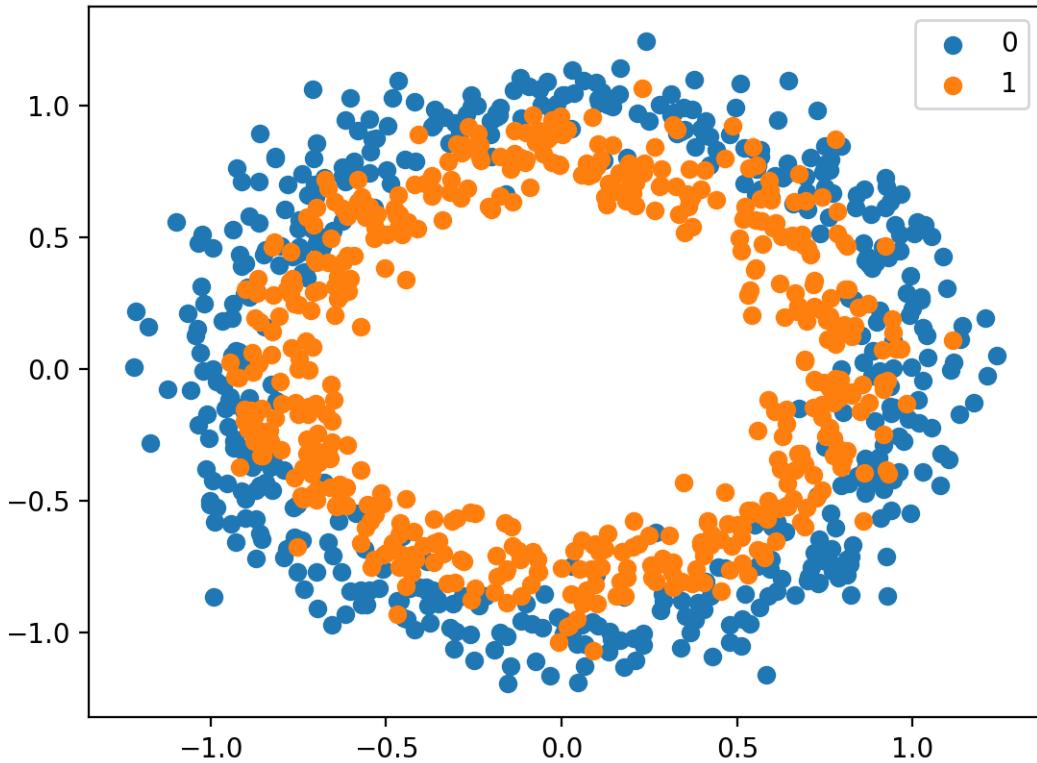


Figure 4.4: Scatter Plot of Dataset for the Circles Binary Classification Problem.

The points are already reasonably scaled around 0, almost in [-1,1]. We won't rescale them in this case. The dataset is split evenly for train and test sets.

```
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 4.21: Example of splitting data into train and test datasets.

A simple MLP model can be defined to address this problem that expects two inputs for the two features in the dataset, a hidden layer with 50 nodes, a rectified linear activation function and an output layer that will need to be configured for the choice of loss function.

```
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='...'))
```

Listing 4.22: Example of defining the MLP model.

The model will be fit using stochastic gradient descent with the sensible default learning rate of 0.01 and momentum of 0.9.

```
opt = SGD(lr=0.01, momentum=0.9)
```

```
model.compile(loss='...', optimizer=opt, metrics=['accuracy'])
```

Listing 4.23: Example of compiling the MLP model.

We will fit the model for 200 training epochs and evaluate the performance of the model against the loss and accuracy at the end of each epoch so that we can plot learning curves.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
```

Listing 4.24: Example of fitting the MLP model.

Now that we have the basis of a problem and model, we can take a look evaluating three common loss functions that are appropriate for a binary classification predictive modeling problem. Although an MLP is used in these examples, the same loss functions can be used when training CNN and RNN models for binary classification.

4.3.1 Binary Cross-Entropy Loss

Cross-entropy is the default loss function to use for binary classification problems. It is intended for use with binary classification where the target values are in the set {0, 1}. Mathematically, it is the preferred loss function under the inference framework of maximum likelihood. It is the loss function to be evaluated first and only changed if you have a good reason. Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for predicting class 1. The score is minimized and a perfect cross-entropy value is 0. Cross-entropy can be specified as the loss function in Keras by specifying ‘binary_crossentropy’ when compiling the model.

```
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 4.25: Example of using binary cross-entropy loss.

The function requires that the output layer is configured with a single node and a ‘sigmoid’ activation in order to predict the probability for class 1.

```
model.add(Dense(1, activation='sigmoid'))
```

Listing 4.26: Example of using the sigmoid activation function.

The complete example of an MLP with cross-entropy loss for the two circles binary classification problem is listed below.

```
# mlp for the circles problem with cross-entropy loss
from sklearn.datasets import make_circles
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
```

```

model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss during training
pyplot.subplot(211)
pyplot.title('Binary Cross-Entropy Loss', pad=-20)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy during training
pyplot.subplot(212)
pyplot.title('Classification Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 4.27: Example of using binary cross-entropy for the two circles problem.

Running the example first prints the classification accuracy for the model on the train and test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model learned the problem reasonably well, achieving about 83% accuracy on the training dataset and about 84% on the test dataset. The scores are reasonably close, suggesting the model is probably not over or underfit.

Train: 0.834, Test: 0.848

Listing 4.28: Example output from using binary cross-entropy for the two circles problem.

A figure is also created showing two line plots, the top with the cross-entropy loss over epochs for the train (blue) and test (orange) dataset, and the bottom plot showing classification accuracy over epochs. The plot shows that the training process converged well. The plot for loss is smooth, given the continuous nature of the error between the probability distributions, whereas the line plot for accuracy shows bumps, given examples in the train and test set can ultimately only be predicted as correct or incorrect, providing less granular feedback on performance.

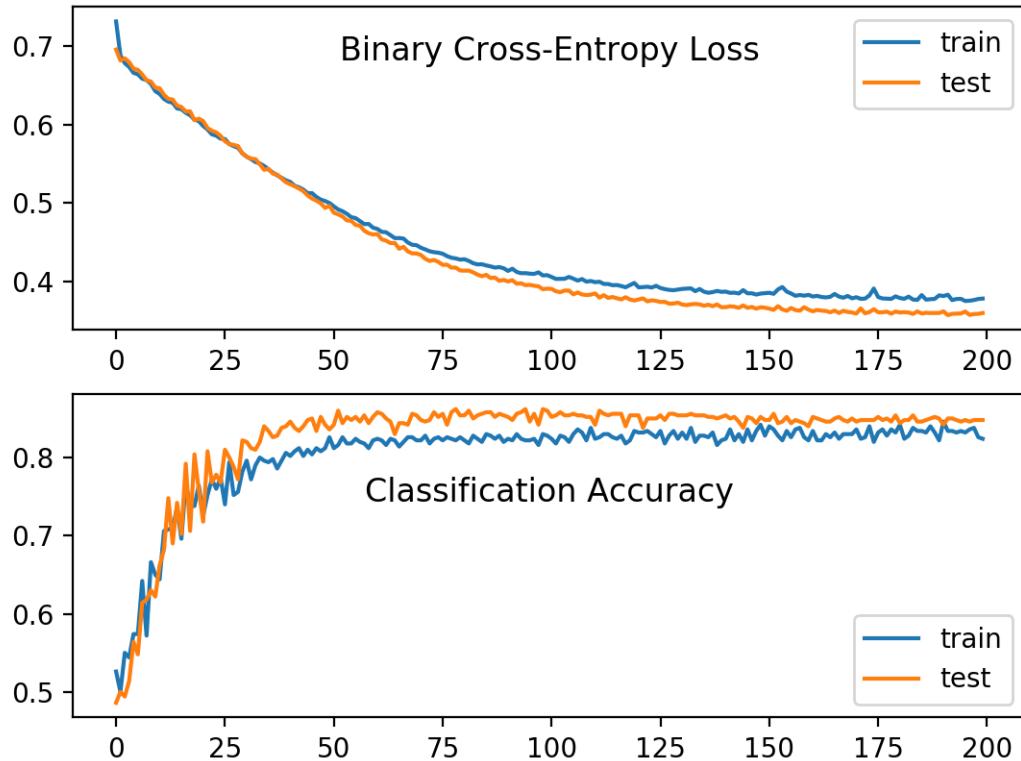


Figure 4.5: Line Plots of Cross-Entropy Loss and Classification Accuracy over Training Epochs on the Two Circles Binary Classification Problem.

4.3.2 Hinge Loss

An alternative to cross-entropy for binary classification problems is the hinge loss function, primarily developed for use with Support Vector Machine (SVM) models. It is intended for use with binary classification where the target values are in the set $\{-1, 1\}$. The hinge loss function encourages examples to have the correct sign, assigning more error when there is a difference in the sign between the actual and predicted class values. Reports of performance with the hinge loss are mixed, sometimes resulting in better performance than cross-entropy on binary classification problems. Firstly, the target variable must be modified to have values in the set $\{-1, 1\}$.

```
# change y from {0,1} to {-1,1}
y[where(y == 0)] = -1
```

Listing 4.29: Example of preparing the target variable.

The hinge loss function can then be specified as ‘`hinge`’ in the `compile` function.

```
model.compile(loss='hinge', optimizer=opt, metrics=['accuracy'])
```

Listing 4.30: Example of using the hinge loss function.

Finally, the output layer of the network must be configured to have a single node with a hyperbolic tangent activation function capable of outputting a single value in the range [-1, 1].

```
model.add(Dense(1, activation='tanh'))
```

Listing 4.31: Example of using the hyperbolic tangent activation function.

The complete example of an MLP with a hinge loss function for the two circles binary classification problem is listed below.

```
# mlp for the circles problem with hinge loss
from sklearn.datasets import make_circles
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
from numpy import where

# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# change y from {0,1} to {-1,1}
y[where(y == 0)] = -1
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='tanh'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='hinge', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss during training
pyplot.subplot(211)
pyplot.title('Hinge Loss', pad=-20)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy during training
pyplot.subplot(212)
pyplot.title('Classification Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 4.32: Example of using the hinge loss function for the two circles problem.

Running the example first prints the classification accuracy for the model on the train and test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see slightly worse performance than using cross-entropy, with the chosen model configuration with less than 80% accuracy on the train and test sets.

Train: 0.794, Test: 0.752

Listing 4.33: Example output from using the hinge loss function for the two circles problem.

A figure is also created showing two line plots, the top with the hinge loss over epochs for the train (blue) and test (orange) dataset, and the bottom plot showing classification accuracy over epochs. The plot of hinge loss shows that the model has converged and has reasonable loss on both datasets. The plot of classification accuracy also shows signs of convergence, albeit at a lower level of skill than may be desirable on this problem.

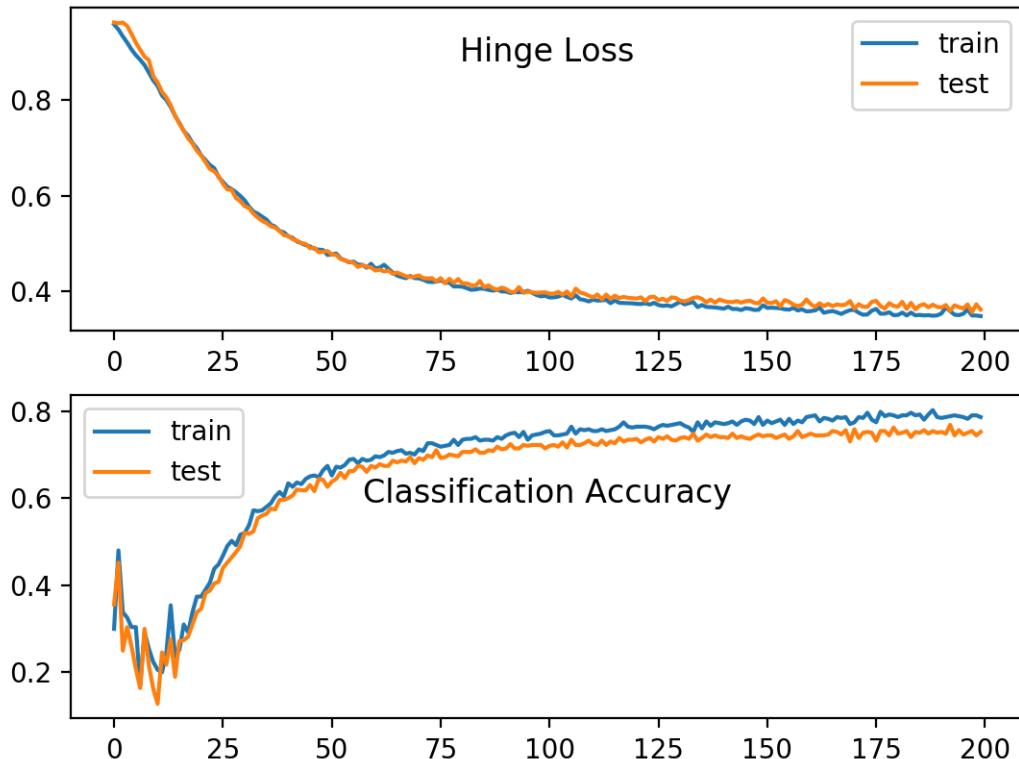


Figure 4.6: Line Plots of Hinge Loss and Classification Accuracy over Training Epochs on the Two Circles Binary Classification Problem.

4.3.3 Squared Hinge Loss

The hinge loss function has many extensions, often the subject of investigation with SVM models. A popular extension is called the squared hinge loss that simply calculates the square of

the score hinge loss. It has the effect of smoothing the surface of the error function and making it numerically easier to work with. If using a hinge loss does result in better performance on a given binary classification problem, is likely that a squared hinge loss may be appropriate. As with using the hinge loss function, the target variable must be modified to have values in the set $\{-1, 1\}$.

```
# change y from {0,1} to {-1,1}
y[where(y == 0)] = -1
```

Listing 4.34: Example of preparing the target variable.

The squared hinge loss can be specified as ‘`squared_hinge`’ in the `compile()` function when defining the model.

```
model.compile(loss='squared_hinge', optimizer=opt, metrics=['accuracy'])
```

Listing 4.35: Example of using the squared hinge loss function.

And finally, the output layer must use a single node with a hyperbolic tangent activation function capable of outputting continuous values in the range $[-1, 1]$.

```
model.add(Dense(1, activation='tanh'))
```

Listing 4.36: Example of using the hyperbolic tangent activation function.

The complete example of an MLP with the squared hinge loss function on the two circles binary classification problem is listed below.

```
# mlp for the circles problem with squared hinge loss
from sklearn.datasets import make_circles
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# change y from {0,1} to {-1,1}
y[where(y == 0)] = -1
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='tanh'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='squared_hinge', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss during training
pyplot.subplot(211)
```

```

pyplot.title('Squared Hinge Loss', pad=-20)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy during training
pyplot.subplot(212)
pyplot.title('Classification Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 4.37: Example of using the squared hinge loss function for the two circles problem.

Running the example first prints the classification accuracy for the model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that for this problem and the chosen model configuration, the hinge squared loss may not be appropriate, resulting in classification accuracy of less than 70% on the train and test sets.

```
Train: 0.664, Test: 0.624
```

Listing 4.38: Example output from using the squared hinge loss function for the two circles problem.

A figure is also created showing two line plots, the top with the squared hinge loss over epochs for the train (blue) and test (orange) dataset, and the bottom plot showing classification accuracy over epochs. The plot of loss shows that indeed, the model converged, but the shape of the error surface is not as smooth as other loss functions where small changes to the weights are causing large changes in loss.

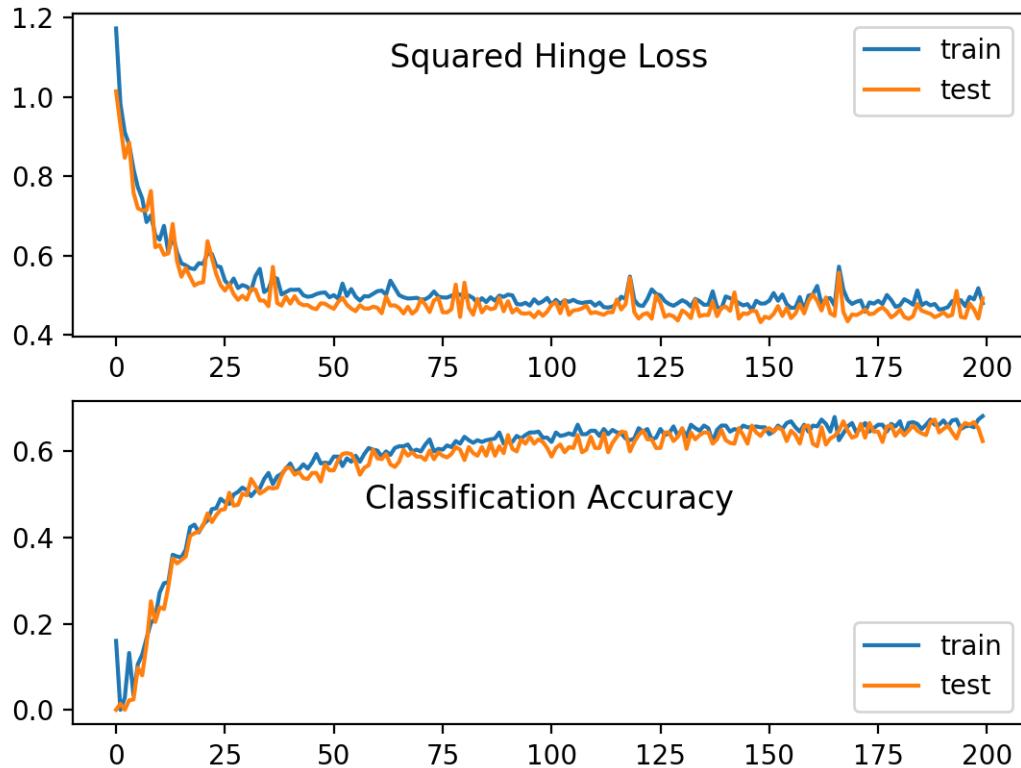


Figure 4.7: Line Plots of Squared Hinge Loss and Classification Accuracy over Training Epochs on the Two Circles Binary Classification Problem.

4.4 Multiclass Classification Loss Functions Case Study

Multiclass classification predictive modeling problems are those where examples are assigned one of more than two classes. The problem is often framed as predicting an integer value, where each class is assigned a unique integer value from 0 to (`num_classes` - 1). The problem is often implemented as predicting the probability of the example belonging to each class. In this section, we will investigate loss functions that are appropriate for multiclass classification predictive modeling problems.

We will use the blobs problem as the basis for the investigation. The `make_blobs()` function provided by scikit-learn provides a way to generate examples given a specified number of classes and input features. We will use this function to generate 1,000 examples for a 3-class classification problem with 2 input variables. The pseudorandom number generator will be seeded consistently so that the same 1,000 examples are generated each time the code is run.

```
# generate dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

Listing 4.39: Example of generating samples for the blobs problem.

The two input variables can be taken as x and y coordinates for points on a two-dimensional plane. The example below creates a scatter plot of the entire dataset coloring points by their class membership.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from numpy import where
from matplotlib import pyplot
# generate dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# select indices of points with each class label
for i in range(3):
    samples_ix = where(y == i)
    pyplot.scatter(X[samples_ix, 0], X[samples_ix, 1])
pyplot.show()
```

Listing 4.40: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot showing the 1,000 examples in the dataset with examples belonging to the 0, 1, and 2 classes colored blue, orange, and green respectively.

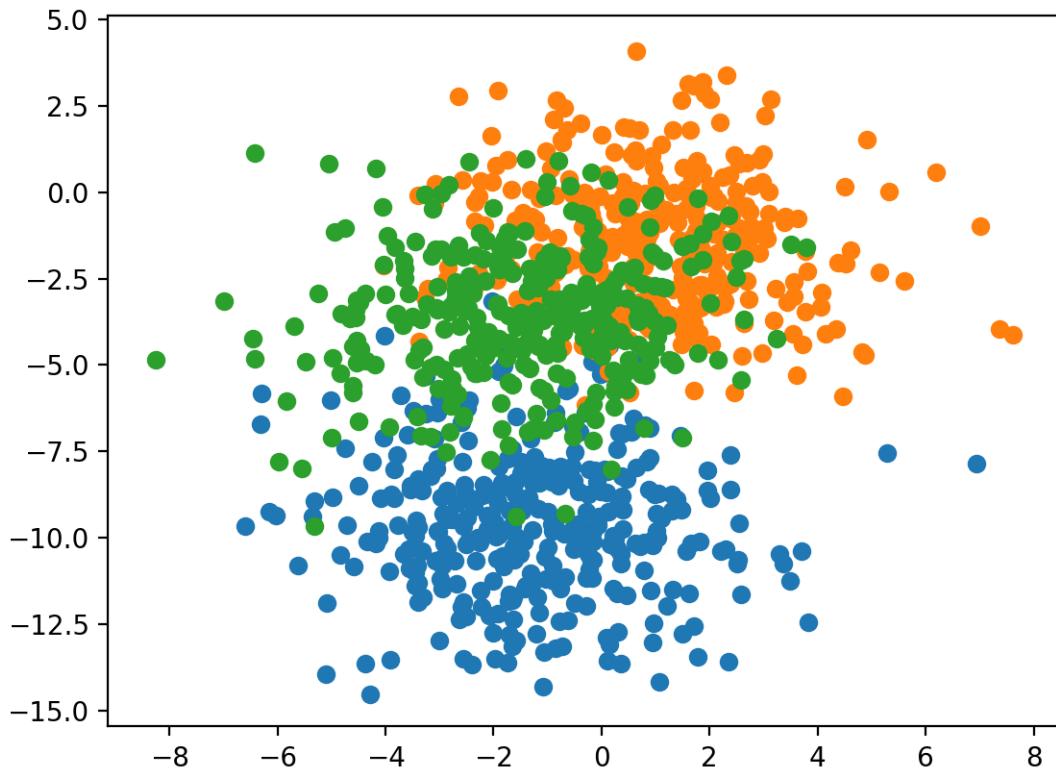


Figure 4.8: Scatter Plot of Examples Generated from the Blobs Multiclass Classification Problem.

The input features are Gaussian and could benefit from standardization; nevertheless, we will keep the values unscaled in this example for brevity. The dataset will be split evenly between train and test sets.

```
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 4.41: Example of splitting the dataset into train and test datasets.

A small MLP model will be used as the basis for exploring loss functions. The model expects two input variables, has 50 nodes in the hidden layer with the rectified linear activation function, and an output layer that must be customized based on the selection of the loss function.

```
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(..., activation='...'))
```

Listing 4.42: Example of defining the MLP model.

The model is fit using stochastic gradient descent with a sensible default learning rate of 0.01 and a momentum of 0.9.

```
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='...', optimizer=opt, metrics=['accuracy'])
```

Listing 4.43: Example of compiling the MLP model.

The model will be fit for 100 epochs on the training dataset and the test dataset will be used as a validation dataset, allowing us to evaluate both loss and classification accuracy on the train and test sets at the end of each training epoch and draw learning curves.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
```

Listing 4.44: Example of fitting the MLP model.

Now that we have the basis of a problem and model, we can take a look evaluating three common loss functions that are appropriate for a multiclass classification predictive modeling problem. Although an MLP is used in these examples, the same loss functions can be used when training CNN and RNN models for multiclass classification.

4.4.1 Multiclass Cross-Entropy Loss

Cross-entropy is the default loss function to use for multiclass classification problems. In this case, it is intended for use with multiclass classification where the target values are in the set $\{0, 1, 3, \dots, n\}$, where each class is assigned a unique integer value. Mathematically, it is the preferred loss function under the inference framework of maximum likelihood. It is the loss function to be evaluated first and only changed if you have a good reason. Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for all classes in the problem. The score is minimized and a perfect cross-entropy value is 0. Cross-entropy can be specified as the loss function in Keras by specifying ‘`categorical_crossentropy`’ when compiling the model.

```
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 4.45: Example of using the categorical cross-entropy loss function.

The function requires that the output layer is configured with n nodes (one for each class), in this case three nodes, and a ‘softmax’ activation in order to predict the probability for each class.

```
model.add(Dense(3, activation='softmax'))
```

Listing 4.46: Example of using the softmax activation function.

In turn, this means that the target variable must be one hot encoded. This is to ensure that each example has an expected probability of 1.0 for the actual class value and an expected probability of 0.0 for all other class values. This can be achieved using the `to_categorical()` Keras function.

```
# one hot encode output variable
y = to_categorical(y)
```

Listing 4.47: Example of one hot encoding the target variable.

The complete example of an MLP with cross-entropy loss for the multiclass blobs classification problem is listed below.

```
# mlp for the blobs multi-class classification problem with cross-entropy loss
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss during training
pyplot.subplot(211)
pyplot.title('Categorical Cross-Entropy Loss', pad=-20)
```

```
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy during training
pyplot.subplot(212)
pyplot.title('Classification Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 4.48: Example of using the categorical cross-entropy loss function for the blobs problem.

Running the example first prints the classification accuracy for the model on the train and test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see the model performed well, achieving a classification accuracy of about 82% on the training dataset and about 82% on the test dataset.

```
Train: 0.824, Test: 0.820
```

Listing 4.49: Example output from using the categorical cross-entropy loss function for the blobs problem.

A figure is also created showing two line plots, the top with the cross-entropy loss over epochs for the train (blue) and test (orange) datasets, and the bottom plot showing classification accuracy over epochs. In this case, the plot shows the model seems to have converged. The line plots for both cross-entropy and accuracy both show good convergence behavior, although somewhat bumpy. The model may be well configured given no sign of over or underfitting. The learning rate or batch size may be tuned to even out the smoothness of the convergence in this case.

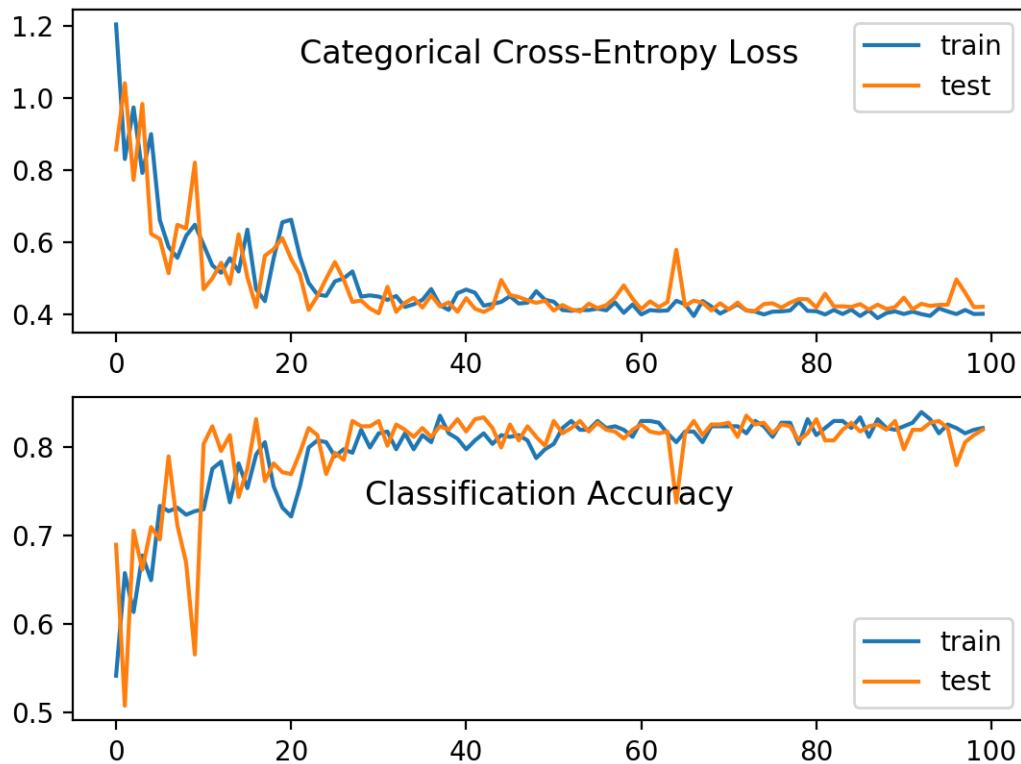


Figure 4.9: Line Plots of Cross-Entropy Loss and Classification Accuracy over Training Epochs on the Blobs Multiclass Classification Problem.

4.4.2 Sparse Multiclass Cross-Entropy Loss

A possible cause of frustration when using cross-entropy with classification problems with a large number of labels is the one hot encoding process. For example, predicting words in a vocabulary may have tens or hundreds of thousands of categories, one for each label. This can mean that the target element of each training example may require a one hot encoded vector with tens or hundreds of thousands of zero values, requiring significant memory. Sparse cross-entropy addresses this by performing the same cross-entropy calculation of error, without requiring that the target variable be one hot encoded prior to training. Sparse cross-entropy can be used in Keras for multiclass classification by using ‘`sparse_categorical_crossentropy`’ when calling the `compile()` function.

```
model.compile(loss='sparse_categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 4.50: Example of using the sparse categorical cross-entropy loss function.

The function requires that the output layer is configured with n nodes (one for each class), in this case three nodes, and a ‘softmax’ activation in order to predict the probability for each class.

```
model.add(Dense(3, activation='softmax'))
```

Listing 4.51: Example of using the softmax activation function.

No one hot encoding of the target variable is required, a benefit of this loss function. The complete example of training an MLP with sparse cross-entropy on the blobs multiclass classification problem is listed below.

```
# mlp for the blobs multi-class classification problem with sparse cross-entropy loss
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='sparse_categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss during training
pyplot.subplot(211)
pyplot.title('Sparse Categorical Cross-Entropy Loss', pad=-20)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy during training
pyplot.subplot(212)
pyplot.title('Classification Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 4.52: Example of using the sparse categorical cross-entropy loss function for the blobs problem.

Running the example first prints the classification accuracy for the model on the train and test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see the model achieves good performance on the problem. In fact, if you repeat the experiment many times, the average performance of sparse and non-sparse cross-entropy should be comparable.

```
Train: 0.818, Test: 0.828
```

Listing 4.53: Example output from using the sparse categorical cross-entropy loss function for the blobs problem.

A figure is also created showing two line plots, the top with the sparse cross-entropy loss over epochs for the train (blue) and test (orange) dataset, and the bottom plot showing classification accuracy over epochs. In this case, the plot shows good convergence of the model over training with regard to loss and classification accuracy.

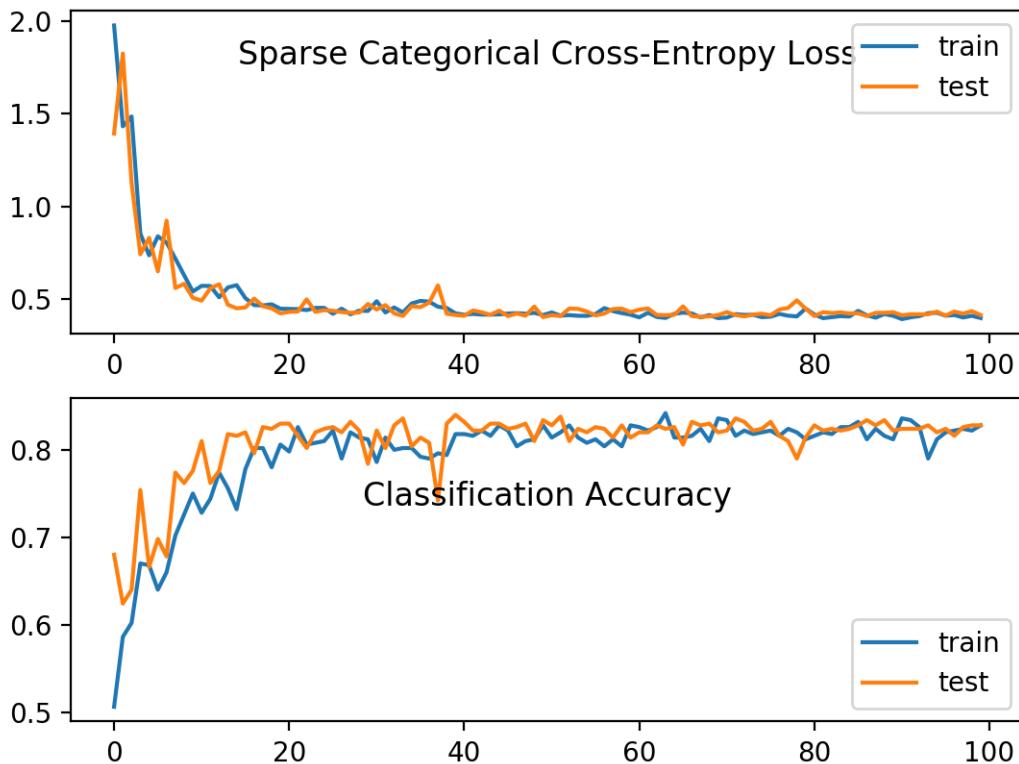


Figure 4.10: Line Plots of Sparse Cross-Entropy Loss and Classification Accuracy over Training Epochs on the Blobs Multiclass Classification Problem.

4.4.3 Kullback Leibler Divergence Loss

Kullback Leibler Divergence, or *KL Divergence* for short, is a measure of how one probability distribution differs from a baseline distribution. A KL divergence loss of 0 suggests the distributions are identical. In practice, the behavior of KL Divergence is very similar to cross-entropy. It calculates how much information is lost (in terms of bits) if the predicted probability distribution is used to approximate the desired target probability distribution.

As such, the KL divergence loss function is more commonly used when using models that learn to approximate a more complex function than simply multiclass classification, such as in the case of an autoencoder used for learning a dense feature representation under a model that must reconstruct the original input. In this case, KL divergence loss would be preferred. Nevertheless, it can be used for multiclass classification, in which case it is functionally equivalent to multiclass cross-entropy. KL divergence loss can be used in Keras by specifying ‘`kullback_leibler_divergence`’ in the `compile()` function.

```
model.compile(loss='kullback_leibler_divergence', optimizer=opt, metrics=['accuracy'])
```

Listing 4.54: Example of using the KL divergence loss function.

As with cross-entropy, the output layer is configured with n nodes (one for each class), in this case three nodes, and a ‘softmax’ activation in order to predict the probability for each class. Also, as with categorical cross-entropy, we must one hot encode the target variable to have an expected probability of 1.0 for the class value and 0.0 for all other class values.

```
# one hot encode output variable
y = to_categorical(y)
```

Listing 4.55: Example of one hot encoding the target variable.

The complete example of training an MLP with KL divergence loss for the blobs multiclass classification problem is listed below.

```
# mlp for the blobs multi-class classification problem with kl divergence loss
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='kullback_leibler_divergence', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss during training
pyplot.subplot(211)
pyplot.title('Kullback Leibler Divergence Loss', pad=-20)
```

```
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy during training
pyplot.subplot(212)
pyplot.title('Classification Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 4.56: Example of using the KL divergence loss function for the blobs problem.

Running the example first prints the classification accuracy for the model on the train and test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we see performance that is similar to those results seen with cross-entropy loss, in this case about 80% accuracy on the train and test dataset.

```
Train: 0.826, Test: 0.808
```

Listing 4.57: Example output from using the KL divergence loss function for the blobs problem.

A figure is also created showing two line plots, the top with the KL divergence loss over epochs for the train (blue) and test (orange) dataset, and the bottom plot showing classification accuracy over epochs. In this case, the plot shows good convergence behavior for both loss and classification accuracy. It is very likely that an evaluation of cross-entropy would result in nearly identical behavior given the similarities in the measure.

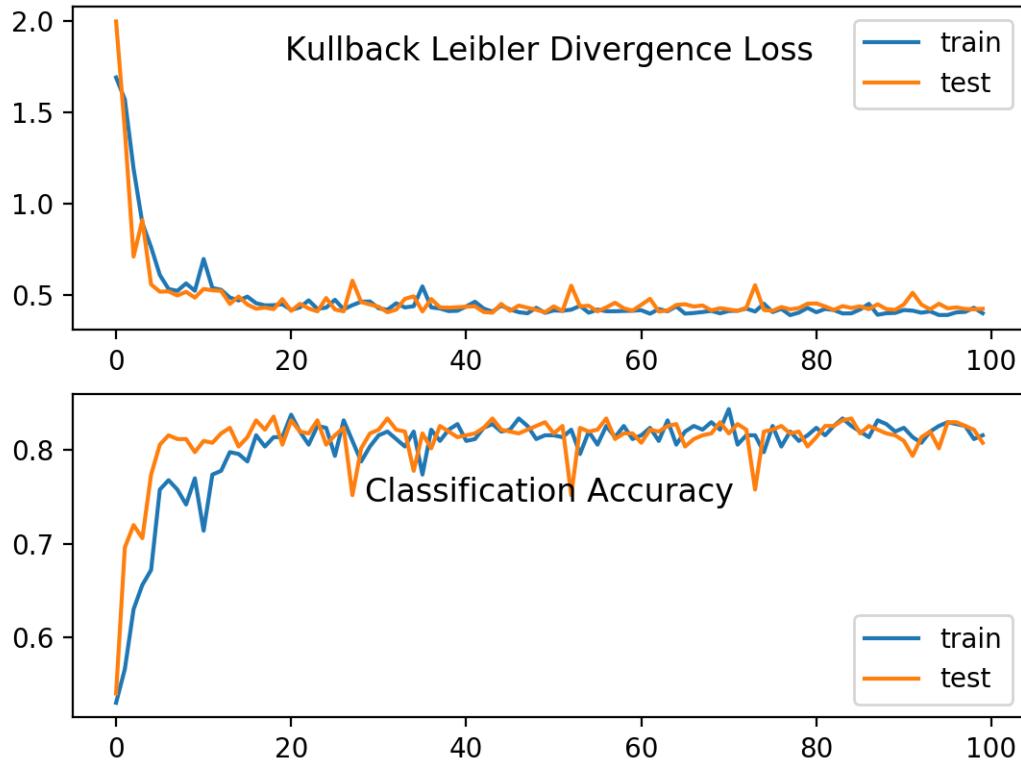


Figure 4.11: Line Plots of KL Divergence Loss and Classification Accuracy over Training Epochs on the Blobs Multiclass Classification Problem.

4.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Alternate Regression Loss.** Experiment with alternate loss functions for regression such as Mean Absolute Percentage Error or MAPE.
- **Alternate Classification Loss.** Experiment with alternate loss functions for classification such as categorical hinge loss.
- **Repeated Evaluation.** Create a repeated evaluation experiment and compare the average final loss and accuracy for categorical cross-entropy, sparse cross-entropy and KL divergence to confirm they are functionally equivalent.

If you explore any of these extensions, I'd love to know.

4.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

4.6.1 Books

- *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
<https://amzn.to/2S8qRdI>
- *Neural Networks for Pattern Recognition*, 1995.
<https://amzn.to/2S8qdwt>

4.6.2 Papers

- *On Loss Functions for Deep Neural Networks in Classification*, 2017.
<https://arxiv.org/abs/1702.05659>

4.6.3 APIs

- Keras Loss Functions API.
<https://keras.io/losses/>
- Keras Activation Functions API.
<https://keras.io/activations/>
- sklearn.preprocessing.StandardScaler API.
<http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>
- sklearn.datasets.make_regression API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html
- sklearn.datasets.make_circles API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html
- sklearn.datasets.make_blobs API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

4.6.4 Articles

- Maximum likelihood estimation, Wikipedia.
https://en.wikipedia.org/wiki/Maximum_likelihood_estimation
- Kullback-Leibler divergence, Wikipedia.
https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence
- Cross entropy, Wikipedia.
https://en.wikipedia.org/wiki/Cross_entropy

- Mean squared error, Wikipedia.
https://en.wikipedia.org/wiki/Mean_squared_error
- Hinge loss, Wikipedia.
https://en.wikipedia.org/wiki/Hinge_loss
- Kullback-Leibler divergence, Wikipedia.
https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence
- Loss Functions in Neural Networks, 2017.
https://isaacchanghau.github.io/post/loss_functions/
- Log Loss, FastAI Wiki.
http://wiki.fast.ai/index.php/Log_Loss

4.7 Summary

In this tutorial, you discovered the role of loss and loss functions in training deep learning neural networks and how to choose the right loss function for your predictive modeling problems. Specifically, you learned:

- Neural networks are trained using an optimization process that requires a loss function to calculate the model error.
- Maximum Likelihood provides a framework for choosing a loss function when training neural networks and machine learning models in general.
- Cross-entropy and mean squared error are the two main types of loss functions to use when training neural network models.

4.7.1 Next

In the next tutorial, you will discover how the learning rate controls the amount that model parameters are updated and in turn the stability and speed of convergence.

Chapter 5

Configure Speed of Learning with Learning Rate

The weights of a neural network cannot be calculated using an analytical method. Instead, the weights must be discovered via an empirical optimization procedure called stochastic gradient descent. The optimization problem addressed by stochastic gradient descent for neural networks is challenging and the space of solutions (sets of weights) may be comprised of many good solutions (called global optima) as well as easy to find, but low in skill solutions (called local optima). The amount of change to the model during each step of this search process, or the step size, is called the *learning rate* and provides perhaps the most important hyperparameter to tune for your neural network in order to achieve good performance on your problem. In this tutorial, you will discover the learning rate hyperparameter used when training deep learning neural networks. After completing this tutorial, you will know:

- Learning rate controls how quickly or slowly a neural network model learns a problem.
- How to configure the learning rate with sensible defaults, diagnose behavior, and develop a sensitivity analysis.
- How to further improve performance with learning rate schedules, momentum, and adaptive learning rates.

Let's get started.

5.1 Learning Rate

In this section you will discover the learning rate, the effect it has on the model during training and tips on how to configure the learning rate when training your own neural network models.

5.1.1 What Is the Learning Rate?

Deep learning neural networks are trained using the stochastic gradient descent algorithm. Stochastic gradient descent is an optimization algorithm that estimates the error gradient for the current state of the model using examples from the training dataset, then updates the weights of the model using the backpropagation of errors algorithm, referred to as simply backpropagation.

The amount that the weights are updated during training is referred to as the step size or the *learning rate*. Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0.

... learning rate, a positive scalar determining the size of the step.

— Page 86, *Deep Learning*, 2016.

The learning rate is often represented using the notation of the lowercase Greek letter eta (η). During training, the backpropagation of error estimates the amount of error for which each weight in a node in the network is responsible. Instead of updating the weight with the full amount, it is scaled by the learning rate. This means that a learning rate of 0.1, a traditionally common default value, would mean that weights in the network are updated $0.1 \times (\text{estimated weight error})$ or 10% of the estimated weight error each time the weights are updated.

5.1.2 Effect of Learning Rate

A neural network learns or approximates a function to best map inputs to outputs from examples in the training dataset. The learning rate hyperparameter controls the rate or speed at which the model learns. Specifically, it controls the amount of apportioned error that the weights of the model are updated with each time they are updated, such as at the end of each batch of training examples. Given a perfectly configured learning rate, the model will learn to best approximate the function given available resources (the number of layers and the number of nodes per layer) in a given number of training epochs (passes through the training data).

Generally, a large learning rate allows the model to learn faster, at the cost of arriving on a sub-optimal final set of weights. A smaller learning rate may allow the model to learn a more optimal or even globally optimal set of weights but may take significantly longer to train. At extremes, a learning rate that is too large will result in weight updates that will be too large and the performance of the model (such as its loss on the training dataset) will oscillate over training epochs. Oscillating performance is said to be caused by weights that diverge (are divergent). A learning rate that is too small may never converge or may get stuck on a suboptimal solution.

When the learning rate is too large, gradient descent can inadvertently increase rather than decrease the training error. [...] When the learning rate is too small, training is not only slower, but may become permanently stuck with a high training error.

— Page 429, *Deep Learning*, 2016.

In the worst case, weight updates that are too large may cause the weights to explode (i.e. result in a numerical overflow).

When using high learning rates, it is possible to encounter a positive feedback loop in which large weights induce large gradients which then induce a large update to the weights. If these updates consistently increase the size of the weights, then [the weights] rapidly moves away from the origin until numerical overflow occurs.

— Page 238, *Deep Learning*, 2016.

Therefore, we should not use a learning rate that is too large or too small. Nevertheless, we must configure the model in such a way that on average a *good enough* set of weights is found to approximate the mapping problem as represented by the training dataset.

5.1.3 How to Configure Learning Rate

It is important to find a good value for the learning rate for your model on your training dataset. The learning rate may, in fact, be the most important hyperparameter to configure for your model.

The initial learning rate [...] This is often the single most important hyperparameter and one should always make sure that it has been tuned [...] If there is only time to optimize one hyper-parameter and one uses stochastic gradient descent, then this is the hyper-parameter that is worth tuning

— *Practical recommendations for gradient-based training of deep architectures*, 2012.

In fact, if there are resources to tune hyperparameters, much of this time should be dedicated to tuning the learning rate.

The learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate.

— Page 429, *Deep Learning*, 2016.

Unfortunately, we cannot analytically calculate the optimal learning rate for a given model on a given dataset. Instead, a good (or good enough) learning rate must be discovered via trial and error.

... in general, it is not possible to calculate the best learning rate a priori.

— Page 72, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

The range of values to consider for the learning rate is less than 1.0 and greater than 10^{-6} .

Typical values for a neural network with standardized inputs (or inputs mapped to the (0,1) interval) are less than 1 and greater than 10^{-6}

— *Practical recommendations for gradient-based training of deep architectures*, 2012.

The learning rate will interact with many other aspects of the optimization process, and the interactions may be nonlinear. Nevertheless, in general, smaller learning rates will require more training epochs. Conversely, larger learning rates will require fewer training epochs. Further, smaller batch sizes are better suited to smaller learning rates given the noisy estimate of the error gradient. A traditional default value for the learning rate is 0.1 or 0.01, and this may represent a good starting point on your problem.

A default value of 0.01 typically works for standard multi-layer neural networks but it would be foolish to rely exclusively on this default value

— *Practical recommendations for gradient-based training of deep architectures*, 2012.

Diagnostic plots can be used to investigate how the learning rate impacts the rate of learning and learning dynamics of the model. One example is to create a line plot of loss over training epochs during training. The line plot can show many properties, such as:

- The rate of learning over training epochs, such as fast or slow.
- Whether model has learned too quickly (sharp rise and plateau) or is learning too slowly (little or no change).
- Whether the learning rate might be too large via oscillations in loss.

Configuring the learning rate is challenging and time-consuming.

The choice of the value for [the learning rate] can be fairly critical, since if it is too small the reduction in error will be very slow, while, if it is too large, divergent oscillations can result.

— Page 95, *Neural Networks for Pattern Recognition*, 1995.

An alternative approach is to perform a sensitivity analysis of the learning rate for the chosen model, also called a grid search. This can help to both highlight an order of magnitude where good learning rates may reside, as well as describe the relationship between learning rate and performance. It is common to grid search learning rates on a log scale from 0.1 to 10^{-5} or 10^{-6} .

Typically, a grid search involves picking values approximately on a logarithmic scale, e.g., a learning rate taken within the set $\{.1, .01, 10^{-3}, 10^{-4}, 10^{-5}\}$

— Page 434, *Deep Learning*, 2016.

When plotted, the results of such a sensitivity analysis often show a *U* shape, where loss decreases (performance improves) as the learning rate is decreased with a fixed number of training epochs to a point where loss sharply increases again because the model fails to converge.

5.1.4 Add Momentum to the Learning Process

Training a neural network can be made easier with the addition of history to the weight update. Specifically, an exponentially weighted average of the prior updates to the weight can be included when the weights are updated. This change to stochastic gradient descent is called *momentum* and adds inertia to the update procedure, causing many past updates in one direction to continue in that direction in the future.

The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

— Page 296, *Deep Learning*, 2016.

Momentum can accelerate learning on those problems where the high-dimensional *weight space* that is being navigated by the optimization process has structures that mislead the gradient descent algorithm, such as flat regions or steep curvature.

The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

— Page 296, *Deep Learning*, 2016.

The amount of inertia of past updates is controlled via the addition of a new hyperparameter, often referred to as the *momentum* or *velocity* and uses the notation of the Greek lowercase letter alpha (α).

... the momentum algorithm introduces a variable v that plays the role of velocity – it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient.

— Page 296, *Deep Learning*, 2016.

It has the effect of smoothing the optimization process, slowing updates to continue in the previous direction instead of getting stuck or oscillating.

One very simple technique for dealing with the problem of widely differing eigenvalues is to add a momentum term to the gradient descent formula. This effectively adds inertia to the motion through weight space and smoothes out the oscillations

— Page 267, *Neural Networks for Pattern Recognition*, 1995.

Momentum is set to a value greater than 0.0 and less than one, where common values such as 0.9 and 0.99 are used in practice.

Common values of [momentum] used in practice include .5, .9, and .99.

— Page 298, *Deep Learning*, 2016.

Momentum does not make it easier to configure the learning rate, as the step size is independent of the momentum. Instead, momentum can improve the speed of the optimization process in concert with the step size, improving the likelihood that a better set of weights is discovered in fewer training epochs.

5.1.5 Use a Learning Rate Schedule

An alternative to using a fixed learning rate is to instead vary the learning rate over the training process. The way in which the learning rate changes over time (training epochs) is referred to as the learning rate schedule or learning rate decay. Perhaps the simplest learning rate schedule is to decrease the learning rate linearly from a large initial value to a small value. This allows large weight changes in the beginning of the learning process and small changes or fine-tuning towards the end of the learning process.

In practice, it is necessary to gradually decrease the learning rate over time, so we now denote the learning rate at iteration [...] This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum.

— Page 294, *Deep Learning*, 2016.

In fact, using a learning rate schedule may be a best practice when training neural networks. Instead of choosing a fixed learning rate hyperparameter, the configuration challenge involves choosing the initial learning rate and a learning rate schedule. It is possible that the choice of the initial learning rate is less sensitive than choosing a fixed learning rate, given the better performance that a learning rate schedule may permit. The learning rate can be decayed to a small value close to zero. Alternately, the learning rate can be decayed over a fixed number of training epochs, then kept constant at a small value for the remaining training epochs to facilitate more time fine-tuning.

In practice, it is common to decay the learning rate linearly until iteration [tau]. After iteration [tau], it is common to leave [the learning rate] constant.

— Page 295, *Deep Learning*, 2016.

5.1.6 Adaptive Learning Rates

The performance of the model on the training dataset can be monitored by the learning algorithm and the learning rate can be adjusted in response. This is called an adaptive learning rate. Perhaps the simplest implementation is to make the learning rate smaller once the performance of the model plateaus, such as by decreasing the learning rate by a factor of two or an order of magnitude.

A reasonable choice of optimization algorithm is SGD with momentum with a decaying learning rate (popular decay schemes that perform better or worse on different problems include decaying linearly until reaching a fixed minimum learning rate, decaying exponentially, or decreasing the learning rate by a factor of 2-10 each time validation error plateaus).

— Page 425, *Deep Learning*, 2016.

Alternately, the learning rate can be increased again if performance does not improve for a fixed number of training epochs. An adaptive learning rate method will generally outperform a model with a badly configured learning rate.

The difficulty of choosing a good learning rate a priori is one of the reasons adaptive learning rate methods are so useful and popular. A good adaptive algorithm will usually converge much faster than simple back-propagation with a poorly chosen fixed learning rate.

— Page 72, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

Although no single method works best on all problems, there are three adaptive learning rate methods that have proven to be robust over many types of neural network architectures and problem types. They are AdaGrad, RMSProp, and Adam, and all maintain and adapt learning rates for each of the weights in the model. Perhaps the most popular is Adam, as it builds upon RMSProp and adds momentum.

At this point, a natural question is: which algorithm should one choose? Unfortunately, there is currently no consensus on this point. Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta and Adam.

— Page 309, *Deep Learning*, 2016.

A robust strategy may be to first evaluate the performance of a model with a modern version of stochastic gradient descent with adaptive learning rates, such as Adam, and use the result as a baseline. Then, if time permits, explore whether improvements can be achieved with a carefully selected learning rate or simpler learning rate schedule.

5.2 Learning Rate Keras API

The Keras deep learning library allows you to easily configure the learning rate for a number of different variations of the stochastic gradient descent optimization algorithm.

5.2.1 Stochastic Gradient Descent

Keras provides the SGD class that implements the stochastic gradient descent optimizer with a learning rate and momentum. First, an instance of the class must be created and configured, then specified to the `optimizer` argument when calling the `compile()` function on the model. The default learning rate is 0.01 and no momentum is used by default.

```
from keras.optimizers import SGD
...
opt = SGD()
model.compile(..., optimizer=opt)
```

Listing 5.1: Example of gradient descent with the default learning rate in Keras.

The learning rate can be specified via the `lr` argument and the momentum can be specified via the `momentum` argument.

```
from keras.optimizers import SGD
...
opt = SGD(lr=0.01, momentum=0.9)
model.compile(..., optimizer=opt)
```

Listing 5.2: Example of gradient descent with specified learning rate and momentum in Keras.

The class also supports weight decay via the `decay` argument. With learning rate decay, the learning rate is calculated each update (e.g. end of each mini-batch) as follows:

$$\text{lrate} = \text{initial_lrate} \times \frac{1}{1 + \text{decay} \times \text{iteration}} \quad (5.1)$$

Where `lrate` is the learning rate for the current epoch, `initial_lrate` is the learning rate specified as an argument to SGD, `decay` is the decay rate which is greater than zero and `iteration` is the current update number.

```
from keras.optimizers import SGD
...
opt = SGD(lr=0.01, momentum=0.9, decay=0.01)
model.compile(..., optimizer=opt)
```

Listing 5.3: Example of gradient descent with learning rate decay in Keras.

5.2.2 Learning Rate Schedule

Keras supports learning rate schedules via callbacks (for more on callbacks, see Section 18.2). The callbacks operate separately from the optimization algorithm, although they adjust the learning rate used by the optimization algorithm. It is recommended to use the SGD when using a learning rate schedule callback. Callbacks are instantiated and configured, then specified in a list to the `callbacks` argument of the `fit()` function when training the model.

Keras provides the `ReduceLROnPlateau` callback that will adjust the learning rate when a plateau in model performance is detected, e.g. no change for a given number of training epochs. This callback is designed to reduce the learning rate after the model stops improving with the hope of fine-tuning model weights. The `ReduceLROnPlateau` callback requires you to specify the metric to monitor during training via the `monitor` argument, the value that the learning rate will be multiplied by via the `factor` argument and the `patience` argument that specifies the number of training epochs to wait before triggering the change in learning rate. For example, we can monitor the validation loss and reduce the learning rate by an order of magnitude if validation loss does not improve for 100 epochs:

```
# snippet of using the ReduceLROnPlateau callback
from keras.callbacks import ReduceLROnPlateau
...
rlrop = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=100)
model.fit(..., callbacks=[rlrop])
```

Listing 5.4: Example of using the `ReduceLROnPlateau` callback in Keras.

Keras also provides `LearningRateScheduler` callback that allows you to specify a function that is called each epoch in order to adjust the learning rate. You can define your Python function that takes two arguments (`epoch` and current learning rate `lrate`) and returns the new learning rate.

```
# snippet of using the LearningRateScheduler callback
from keras.callbacks import LearningRateScheduler
...
def my_learning_rate(epoch, lrate)
    return lrate

lrs = LearningRateScheduler(my_learning_rate)
model.fit(..., callbacks=[lrs])
```

Listing 5.5: Example of using the `LearningRateScheduler` callback in Keras.

5.2.3 Adaptive Learning Rate Gradient Descent

Keras also provides a suite of extensions of simple stochastic gradient descent that support adaptive learning rates. Because each method adapts the learning rate, often one learning rate per model weight, little configuration is often required. Three commonly used adaptive learning rate methods include:

RMSProp Optimizer

```
from keras.optimizers import RMSprop
...
opt = RMSprop()
model.compile(..., optimizer=opt)
```

Listing 5.6: Example of the RMSprop optimizer in Keras.

Adagrad Optimizer

```
from keras.optimizers import Adagrad
...
opt = Adagrad()
model.compile(..., optimizer=opt)
```

Listing 5.7: Example of the Adagrad optimizer in Keras.

Adam Optimizer

```
from keras.optimizers import Adam
...
opt = Adam()
model.compile(..., optimizer=opt)
```

Listing 5.8: Example of the Adam optimizer in Keras.

5.3 Learning Rate Case Study

In this section, we will demonstrate how to use the learning rate to control convergence with a MLP on a simple classification problem. This example provides a template for exploring the learning rate with your own neural network for classification and regression problems.

5.3.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate the effect of learning rate on model performance. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. The problem can be configured to have two input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same data points.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

Listing 5.9: Example of creating samples for the blobs problem.

The results are the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can plot each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot for each class value
for class_value in range(3):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 5.10: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (separable by a line), causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions.

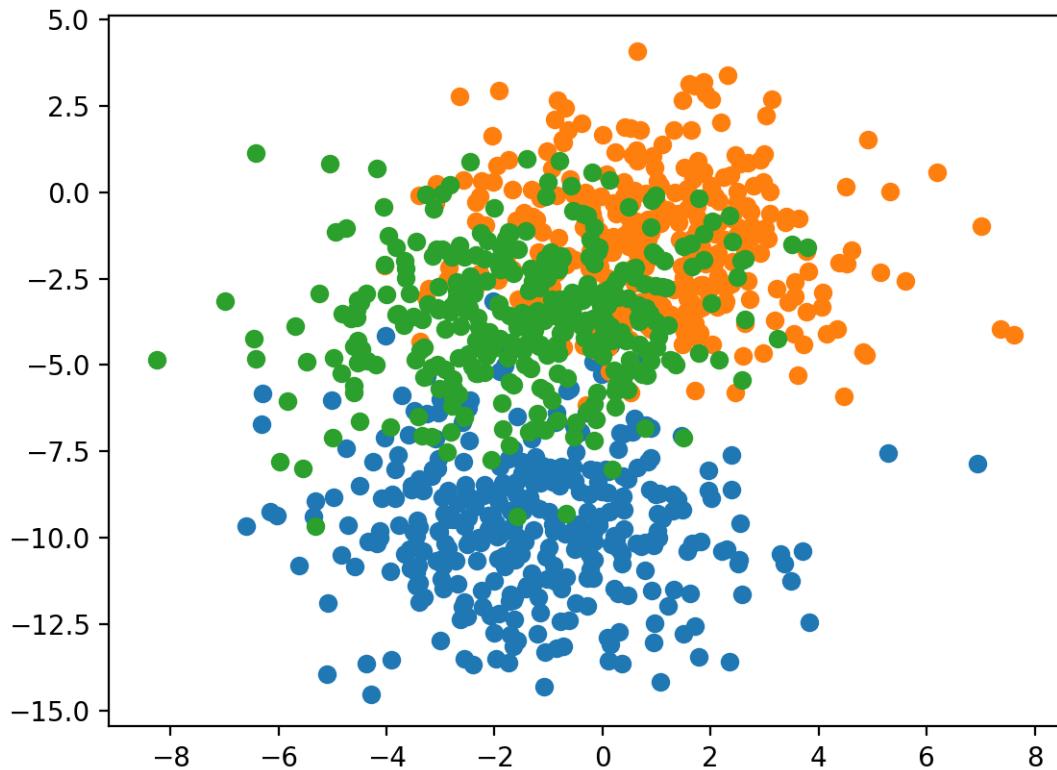


Figure 5.1: Scatter Plot of Blobs Dataset With Three Classes and Points Colored by Class Value.

5.3.2 Effect of Learning Rate and Momentum

In this section, we will develop a Multilayer Perceptron (MLP) model to address the blobs classification problem and investigate the effect of different learning rates and momentum.

Learning Rate Dynamics

The first step is to develop a function that will create the samples from the problem and split them into train and test datasets. Additionally, we must also one hot encode the target variable so that we can develop a model that predicts the probability of an example belonging to each class. The `prepare_data()` function below implements this behavior, returning train and test sets split into input and output elements.

```
# prepare train and test dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
```

```

trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
return trainX, trainy, testX, testy

```

Listing 5.11: Example of a function for preparing the dataset for modeling.

Next, we can develop a function to fit and evaluate an MLP model. First, we will define a simple MLP model that expects two input variables from the blobs problem, has a single hidden layer with 50 nodes, and an output layer with three nodes to predict the probability for each of the three classes. Nodes in the hidden layer will use the rectified linear activation function, whereas nodes in the output layer will use the softmax activation function.

```

# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))

```

Listing 5.12: Example of defining the MLP model.

We will use the stochastic gradient descent optimizer and require that the learning rate be specified so that we can evaluate different rates. The model will be trained to minimize cross-entropy.

```

# compile model
opt = SGD(lr=lrate)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])

```

Listing 5.13: Example of compiling the MLP model.

The model will be fit for 200 training epochs, found with a little trial and error, and the test set will be used as the validation dataset so we can get an idea of the generalization error of the model during training.

```

# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)

```

Listing 5.14: Example of fitting the MLP model.

Once fit, we will plot the accuracy of the model on the train and test sets over the training epochs.

```

# plot learning curves
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.title('lrate=' + str(lrate), pad=-50)

```

Listing 5.15: Example of plotting learning curves for the MLP model.

The `fit_model()` function below ties together these elements and will fit a model and plot its performance given the train and test datasets as well as a specific learning rate to evaluate.

```

# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, lrate):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model

```

```

opt = SGD(lr=lrate)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
# plot learning curves
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.title('lrate=' + str(lrate), pad=-50)

```

Listing 5.16: Example of defining a function for fitting and evaluating an MLP model.

We can now investigate the dynamics of different learning rates on the train and test accuracy of the model. In this example, we will evaluate learning rates on a logarithmic scale from 1E-0 (1.0) to 1E-7 and create line plots for each learning rate by calling the `fit_model()` function.

```

# create learning curves for different learning rates
learning_rates = [1E-0, 1E-1, 1E-2, 1E-3, 1E-4, 1E-5, 1E-6, 1E-7]
for i in range(len(learning_rates)):
    # determine the plot number
    plot_no = 420 + (i+1)
    pyplot.subplot(plot_no)
    # fit model and plot learning curves for a learning rate
    fit_model(trainX, trainy, testX, testy, learning_rates[i])
# show learning curves
pyplot.show()

```

Listing 5.17: Example of evaluating a range of different learning rates.

Tying all of this together, the complete example is listed below.

```

# study of learning rate on accuracy for blobs problem
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare train and test dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy

# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, lrate):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model

```

```

opt = SGD(lr=lrate)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
# plot learning curves
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.title('lrate=' + str(lrate), pad=-50)

# prepare dataset
trainX, trainy, testX, testy = prepare_data()
# create learning curves for different learning rates
learning_rates = [1E-0, 1E-1, 1E-2, 1E-3, 1E-4, 1E-5, 1E-6, 1E-7]
for i in range(len(learning_rates)):
    # determine the plot number
    plot_no = 420 + (i+1)
    pyplot.subplot(plot_no)
    # fit model and plot learning curves for a learning rate
    fit_model(trainX, trainy, testX, testy, learning_rates[i])
# show learning curves
pyplot.show()

```

Listing 5.18: Example of evaluating the dynamics of a range of different learning rates for an MLP on the blobs problem.

Running the example creates a single figure that contains eight line plots for the eight different evaluated learning rates. Classification accuracy on the training dataset is marked in blue, whereas accuracy on the test dataset is marked in orange.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The plots show oscillations in behavior for the too-large learning rate of 1.0 and the inability of the model to learn anything with the too-small learning rates of 1E-6 and 1E-7. We can see that the model was able to learn the problem well with the learning rates 1E-1, 1E-2 and 1E-3, although successively slower as the learning rate was decreased. With the chosen model configuration, the results suggest a moderate learning rate of 0.1 results in good model performance on the train and test sets.

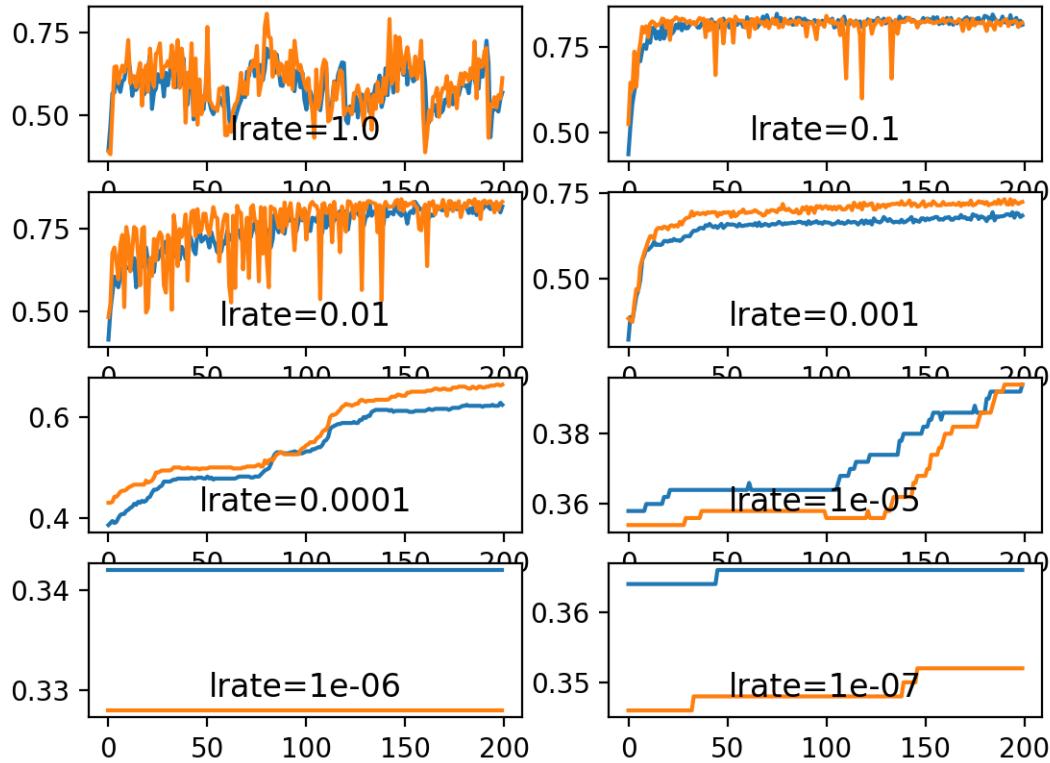


Figure 5.2: Line Plots of Train and Test Accuracy for a Suite of Learning Rates on the Blobs Classification Problem.

Momentum Dynamics

Momentum can smooth the progression of the learning algorithm that, in turn, can accelerate the training process. We can adapt the example from the previous section to evaluate the effect of momentum with a fixed learning rate. In this case, we will choose the learning rate of 0.01 that in the previous section converged to a reasonable solution, but required more epochs than the learning rate of 0.1. The `fit_model()` function can be updated to take a `momentum` argument instead of a learning rate argument, that can be used in the configuration of the SGD class and reported on the resulting plot. The updated version of this function is listed below.

```
# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, momentum):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=momentum)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
```

```
# plot learning curves
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.title('momentum=' + str(momentum), pad=-80)
```

Listing 5.19: Example updated function for evaluating MLPs with different values of momentum.

It is common to use momentum values close to 1.0, such as 0.9 and 0.99. In this example, we will demonstrate the dynamics of the model without momentum compared to the model with momentum values of 0.5 and higher momentum values.

```
# create learning curves for different momentums
momentums = [0.0, 0.5, 0.9, 0.99]
for i in range(len(momentums)):
    # determine the plot number
    plot_no = 220 + (i+1)
    pyplot.subplot(plot_no)
    # fit model and plot learning curves for a momentum
    fit_model(trainX, trainy, testX, testy, momentums[i])
# show learning curves
pyplot.show()
```

Listing 5.20: Example evaluating different momentum values.

Tying all of this together, the complete example is listed below.

```
# study of momentum on accuracy for blobs problem
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare train and test dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy

# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, momentum):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=momentum)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
```

```
# plot learning curves
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.title('momentum=' + str(momentum), pad=-80)

# prepare dataset
trainX, trainy, testX, testy = prepare_data()
# create learning curves for different momentums
momentums = [0.0, 0.5, 0.9, 0.99]
for i in range(len(momentums)):
    # determine the plot number
    plot_no = 220 + (i+1)
    pyplot.subplot(plot_no)
    # fit model and plot learning curves for a momentum
    fit_model(trainX, trainy, testX, testy, momentums[i])
# show learning curves
pyplot.show()
```

Listing 5.21: Example of evaluating the dynamics of a range of different momentum values for an MLP on the blobs problem.

Running the example creates a single figure that contains four line plots for the different evaluated momentum values. Classification accuracy on the training dataset is marked in blue, whereas accuracy on the test dataset is marked in orange.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that the addition of momentum does accelerate the training of the model. Specifically, momentum values of 0.9 and 0.99 achieve reasonable train and test accuracy within about 50 training epochs as opposed to 200 training epochs when momentum is not used. In all cases where momentum is used, the accuracy of the model on the holdout test dataset appears to be more stable, showing less volatility over the training epochs.

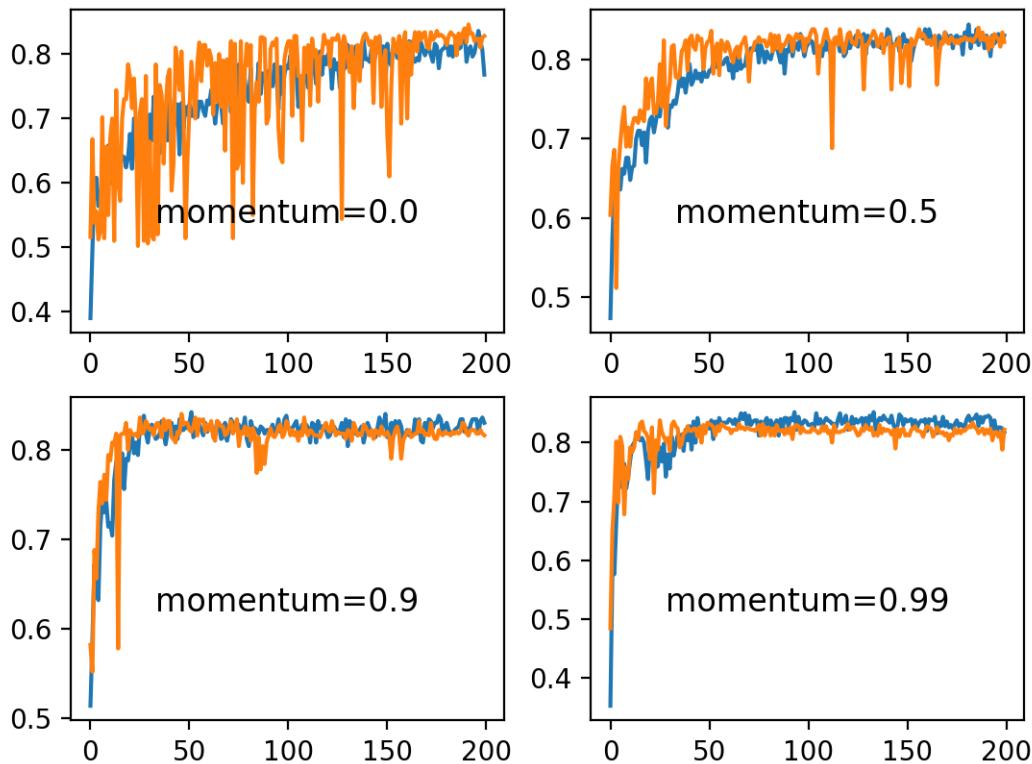


Figure 5.3: Line Plots of Train and Test Accuracy for a Suite of Momentums on the Blobs Classification Problem.

5.3.3 Effect of Learning Rate Schedules

We will look at two learning rate schedules in this section. The first is the decay built into the SGD class and the second is the `ReduceLROnPlateau` callback.

Learning Rate Decay

The SGD class provides the `decay` argument that specifies the learning rate decay. It may not be clear from the equation or the code as to the effect that this decay has on the learning rate over updates. We can make this clearer with a worked example. The function below implements the learning rate decay as implemented in the SGD class.

```
# learning rate decay
def decay_lrate(initial_lrate, decay, iteration):
    return initial_lrate * (1.0 / (1.0 + decay * iteration))
```

Listing 5.22: Example of a function for calculating a decaying learning rate.

We can use this function to calculate the learning rate over multiple updates with different decay values. We will compare a range of decay values [1E-1, 1E-2, 1E-3, 1E-4] with an initial learning rate of 0.01 and 200 weight updates.

```

decays = [1E-1, 1E-2, 1E-3, 1E-4]
lrate = 0.01
n_updates = 200
for decay in decays:
    # calculate learning rates for updates
    lrates = [decay_lrate(lrate, decay, i) for i in range(n_updates)]
    # plot result
    pyplot.plot(lrates, label=str(decay))

```

Listing 5.23: Example of evaluating different decay rates.

The complete example is listed below.

```

# demonstrate the effect of decay on the learning rate
from matplotlib import pyplot

# learning rate decay
def decay_lrate(initial_lrate, decay, iteration):
    return initial_lrate * (1.0 / (1.0 + decay * iteration))

decays = [1E-1, 1E-2, 1E-3, 1E-4]
lrate = 0.01
n_updates = 200
for decay in decays:
    # calculate learning rates for updates
    lrates = [decay_lrate(lrate, decay, i) for i in range(n_updates)]
    # plot result
    pyplot.plot(lrates, label=str(decay))
pyplot.legend()
pyplot.show()

```

Listing 5.24: Example of plotting the effect of different learning rate decay rates.

Running the example creates a line plot showing learning rates over updates for different decay values. We can see that in all cases, the learning rate starts at the initial value of 0.01. We can see that a small decay value of 1E-4 (red) has almost no effect, whereas a large decay value of 1E-1 (blue) has a dramatic effect, reducing the learning rate to below 0.002 within 50 epochs (about one order of magnitude less than the initial value) and arriving at the final value of about 0.0004 (about two orders of magnitude less than the initial value).

We can see that the change to the learning rate is not linear. We can also see that changes to the learning rate are dependent on the batch size, after which an update is performed. In the example from the previous section, a default batch size of 32 across 500 examples results in 16 updates per epoch and 3,200 updates across the 200 epochs. Using a decay of 0.1 and an initial learning rate of 0.01, we can calculate the final learning rate to be a tiny value of about 3.1E-05.

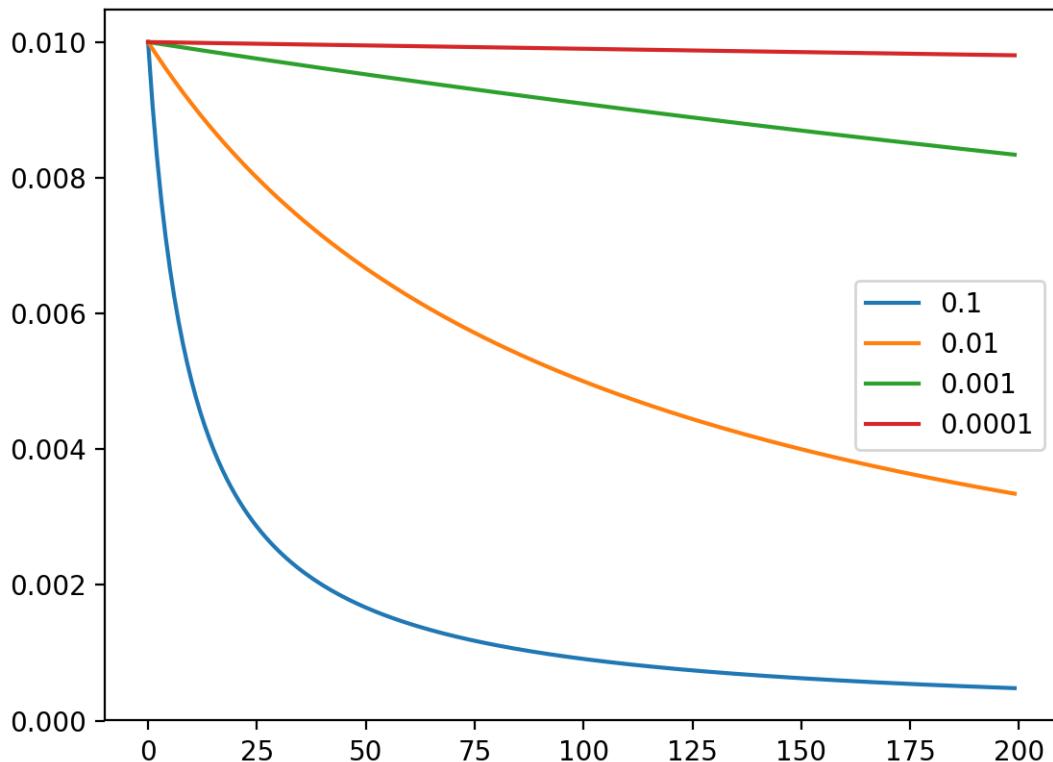


Figure 5.4: Line Plot of the Effect of Decay on Learning Rate Over Multiple Weight Updates.

We can update the example from the previous section to evaluate the dynamics of different learning rate decay values. Fixing the learning rate at 0.01 and not using momentum, we would expect that a very small learning rate decay would be preferred, as a large learning rate decay would rapidly result in a learning rate that is too small for the model to learn effectively. The `fit_model()` function can be updated to take a `decay` argument that can be used to configure decay for the SGD class. The updated version of the function is listed below.

```
# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, decay):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, decay=decay)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
    # plot learning curves
    pyplot.plot(history.history['accuracy'], label='train')
    pyplot.plot(history.history['val_accuracy'], label='test')
    pyplot.title('decay=' + str(decay), pad=-80)
```

Listing 5.25: Example of the updated function for evaluating model with different learning rate decay rates.

We can evaluate the same four decay values of [1E-1, 1E-2, 1E-3, 1E-4] and their effect on model accuracy. The complete example is listed below.

```
# study of decay rate on accuracy for blobs problem
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare train and test dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy

# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, decay):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, decay=decay)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
    # plot learning curves
    pyplot.plot(history.history['accuracy'], label='train')
    pyplot.plot(history.history['val_accuracy'], label='test')
    pyplot.title('decay=' + str(decay), pad=-80)

# prepare dataset
trainX, trainy, testX, testy = prepare_data()
# create learning curves for different decay rates
decay_rates = [1E-1, 1E-2, 1E-3, 1E-4]
for i in range(len(decay_rates)):
    # determine the plot number
    plot_no = 220 + (i+1)
    pyplot.subplot(plot_no)
    # fit model and plot learning curves for a decay rate
    fit_model(trainX, trainy, testX, testy, decay_rates[i])
# show learning curves
pyplot.show()
```

Listing 5.26: Example of evaluating the dynamics of different learning rate decay rates with an MLP on the blobs problem.

Running the example creates a single figure that contains four line plots for the different evaluated learning rate decay values. Classification accuracy on the training dataset is marked in blue, whereas accuracy on the test dataset is marked in orange.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that the large decay values of 1E-1 and 1E-2 indeed decay the learning rate too rapidly for this model on this problem and result in poor performance. The smaller decay values result in better performance, with the value of 1E-4 perhaps causing in a similar result as not using decay at all. In fact, we can calculate the final learning rate with a decay of 1E-4 to be about 0.0075, only a little bit smaller than the initial value of 0.01.

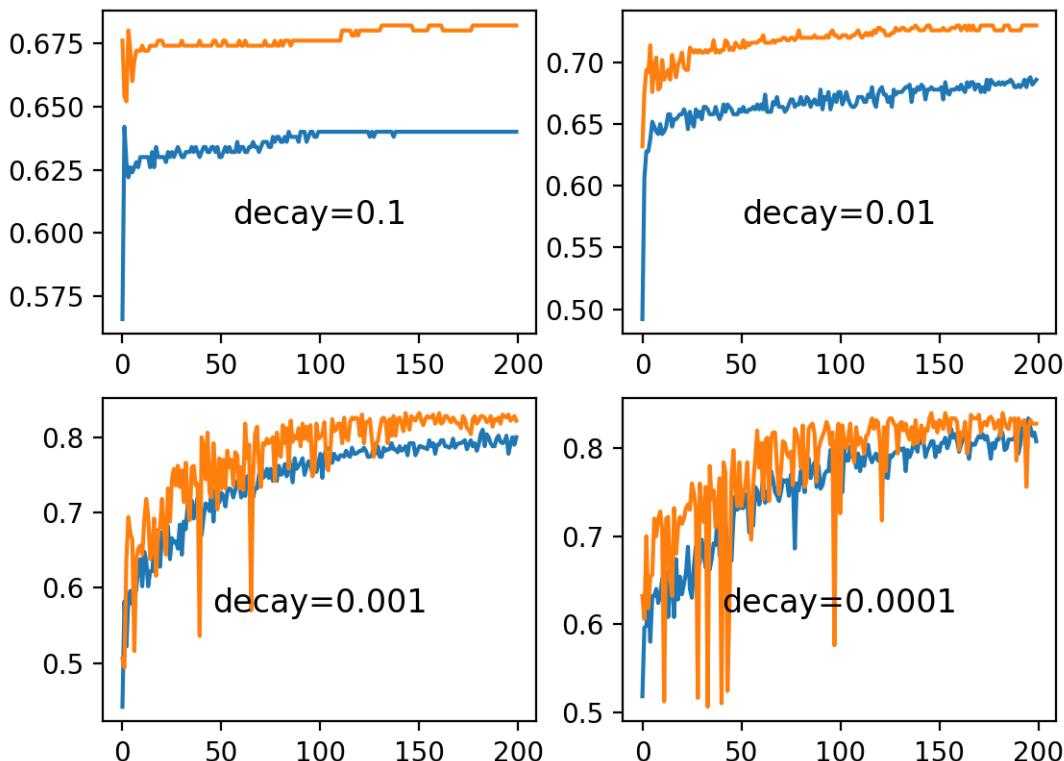


Figure 5.5: Line Plots of Train and Test Accuracy for a Suite of Decay Rates on the Blobs Classification Problem.

Drop Learning Rate on Plateau

The `ReduceLROnPlateau` will drop the learning rate by a factor after no change in a monitored metric for a given number of epochs. We can explore the effect of different `patience` values,

which is the number of epochs to wait for a change before dropping the learning rate. We will use the default learning rate of 0.01 and drop the learning rate by an order of magnitude by setting the `factor` argument to 0.1.

```
rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=patience, min_delta=1E-7)
```

Listing 5.27: Example of configuring the `ReduceLROnPlateau` callback.

It will be interesting to review the effect on the learning rate over the training epochs. We can do that by creating a new Keras `Callback` that is responsible for recording the learning rate at the end of each training epoch. We can then retrieve the recorded learning rates and create a line plot to see how the learning rate was affected by drops. We can create a custom `Callback` called `LearningRateMonitor`. The `on_train_begin()` function is called at the start of training, and in it we can define an empty list of learning rates. The `on_epoch_end()` function is called at the end of each training epoch and in it we can retrieve the optimizer and the current learning rate from the optimizer and store it in the list. The complete `LearningRateMonitor` callback is listed below.

```
# monitor the learning rate
class LearningRateMonitor(Callback):
    # start of training
    def on_train_begin(self, logs={}):
        self.lrates = list()

    # end of each training epoch
    def on_epoch_end(self, epoch, logs={}):
        # get and store the learning rate
        optimizer = self.model.optimizer
        lrate = float(backend.get_value(optimizer.lr))
        self.lrates.append(lrate)
```

Listing 5.28: Example of defining a custom callback to monitor the value of the learning rate.

The `fit_model()` function developed in the previous sections can be updated to create and configure the `ReduceLROnPlateau` callback and our new `LearningRateMonitor` callback and register them with the model in the call to fit. The function will also take `patience` as an argument so that we can evaluate different values.

```
# fit model
rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=patience, min_delta=1E-7)
lrm = LearningRateMonitor()
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0,
                     callbacks=[rlrp, lrm])
```

Listing 5.29: Example of fitting a model with learning rate callbacks.

We will want to create a few plots in this example, so instead of creating subplots directly, the `fit_model()` function will return the list of learning rates as well as loss and accuracy on the training dataset for each training epochs. The function with these updates is listed below.

```
# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, patience):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
```

```
# compile model
opt = SGD(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=patience,
    min_delta=1E-7)
lrm = LearningRateMonitor()
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200,
    verbose=0, callbacks=[rlrp, lrm])
return lrm.lrates, history.history['loss'], history.history['accuracy']
```

Listing 5.30: Example of updated function to evaluate learning rate schedule with different patience values.

The patience in the `ReduceLROnPlateau` controls how often the learning rate will be dropped. We will test a few different patience values suited for this model on the blobs problem and keep track of the learning rate, loss, and accuracy series from each run.

```
# create learning curves for different patiences
patiences = [2, 5, 10, 15]
lr_list, loss_list, acc_list, = list(), list(), list()
for i in range(len(patiences)):
    # fit model and plot learning curves for a patience
    lr, loss, acc = fit_model(trainX, trainy, testX, testy, patiences[i])
    lr_list.append(lr)
    loss_list.append(loss)
    acc_list.append(acc)
```

Listing 5.31: Example of evaluating a range of different patience values for the learning rate schedule.

At the end of the run, we will create figures with line plots for each of the patience values for the learning rates, training loss, and training accuracy for each patience value. We can create a helper function to easily create a figure with subplots for each series that we have recorded.

```
# create line plots for a series
def line_plots(patiences, series):
    for i in range(len(patiences)):
        pyplot.subplot(220 + (i+1))
        pyplot.plot(series[i])
        pyplot.title('patience=' + str(patiences[i]), pad=-80)
    pyplot.show()
```

Listing 5.32: Example of plotting the dynamics of different patience values for the learning rate schedule.

Tying these elements together, the complete example is listed below.

```
# study of patience for the learning rate drop schedule on the blobs problem
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from keras.callbacks import Callback
from keras.callbacks import ReduceLROnPlateau
from keras import backend
```

```
from matplotlib import pyplot

# monitor the learning rate
class LearningRateMonitor(Callback):
    # start of training
    def on_train_begin(self, logs={}):
        self.lrates = list()

    # end of each training epoch
    def on_epoch_end(self, epoch, logs={}):
        # get and store the learning rate
        optimizer = self.model.optimizer
        lrate = float(backend.get_value(optimizer.lr))
        self.lrates.append(lrate)

    # prepare train and test dataset
    def prepare_data():
        # generate 2d classification dataset
        X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
        # one hot encode output variable
        y = to_categorical(y)
        # split into train and test
        n_train = 500
        trainX, testX = X[:n_train, :], X[n_train:, :]
        trainy, testy = y[:n_train], y[n_train:]
        return trainX, trainy, testX, testy

    # fit a model and plot learning curve
    def fit_model(trainX, trainy, testX, testy, patience):
        # define model
        model = Sequential()
        model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
        model.add(Dense(3, activation='softmax'))
        # compile model
        opt = SGD(lr=0.01)
        model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
        # fit model
        rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=patience,
                                 min_delta=1E-7)
        lrm = LearningRateMonitor()
        history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200,
                            verbose=0, callbacks=[rlrp, lrm])
        return lrm.lrates, history.history['loss'], history.history['accuracy']

    # create line plots for a series
    def line_plots(patiences, series):
        for i in range(len(patiences)):
            pyplot.subplot(220 + (i+1))
            pyplot.plot(series[i])
            pyplot.title('patience=' + str(patiences[i]), pad=-80)
        pyplot.show()

    # prepare dataset
    trainX, trainy, testX, testy = prepare_data()
    # create learning curves for different patiences
    patiences = [2, 5, 10, 15]
```

```
lr_list, loss_list, acc_list, = list(), list(), list()
for i in range(len(patiences)):
    # fit model and plot learning curves for a patience
    lr, loss, acc = fit_model(trainX, trainy, testX, testy, patiences[i])
    lr_list.append(lr)
    loss_list.append(loss)
    acc_list.append(acc)
# plot learning rates
line_plots(patiences, lr_list)
# plot loss
line_plots(patiences, loss_list)
# plot accuracy
line_plots(patiences, acc_list)
```

Listing 5.33: Example of evaluating the dynamics of learning rate schedules with an MLP on the blobs problem.

Running the example creates three figures, each containing a line plot for the different patience values.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The first figure shows line plots of the learning rate over the training epochs for each of the evaluated patience values. We can see that the smallest patience value of two rapidly drops the learning rate to a minimum value within 25 epochs, the largest patience of 15 only suffers one drop in the learning rate. From these plots, we would expect the patience values of 5 and 10 for this model on this problem to result in better performance as they allow the larger learning rate to be used for some time before dropping the rate to refine the weights.

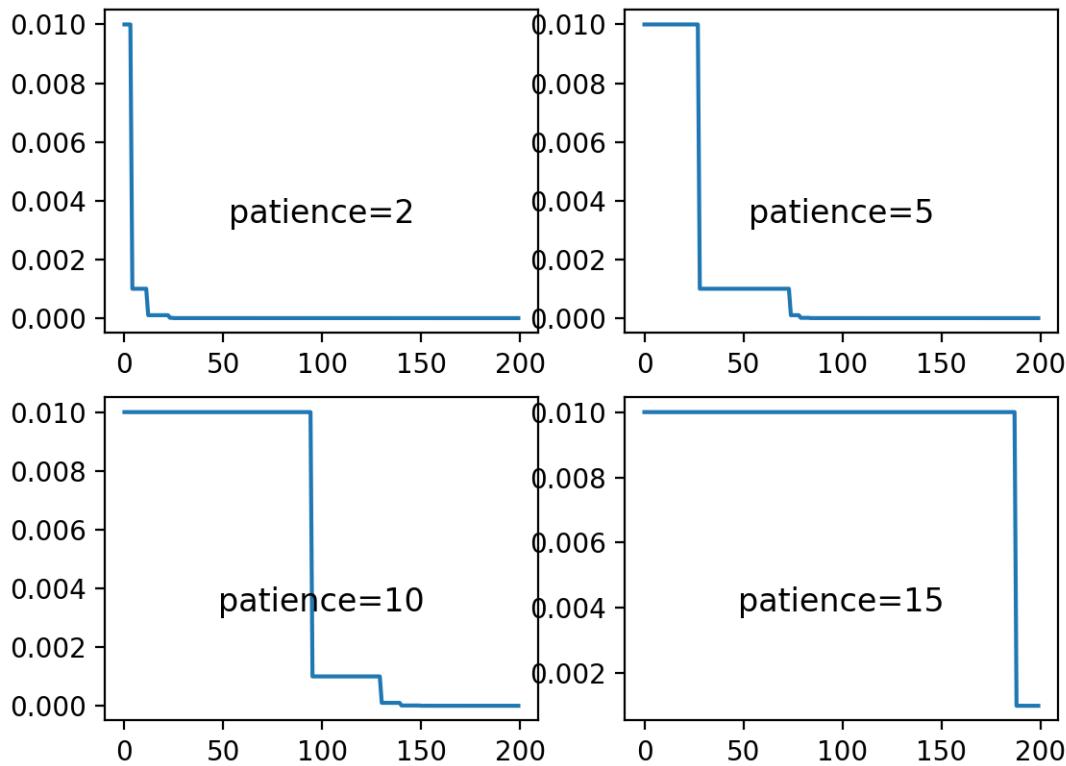


Figure 5.6: Line Plots of Learning Rate Over Epochs for Different Patience Values Used in the ReduceLROnPlateau Schedule.

The next figure shows the loss on the training dataset for each of the patience values. The plot shows that the patience values of 2 and 5 result in a rapid convergence of the model, perhaps to a sub-optimal loss value. In the case of a patience level of 10 and 15, loss drops reasonably until the learning rate is dropped below a level that large changes to the loss can be seen. This occurs halfway for the patience of 10 and nearly the end of the run for patience 15.

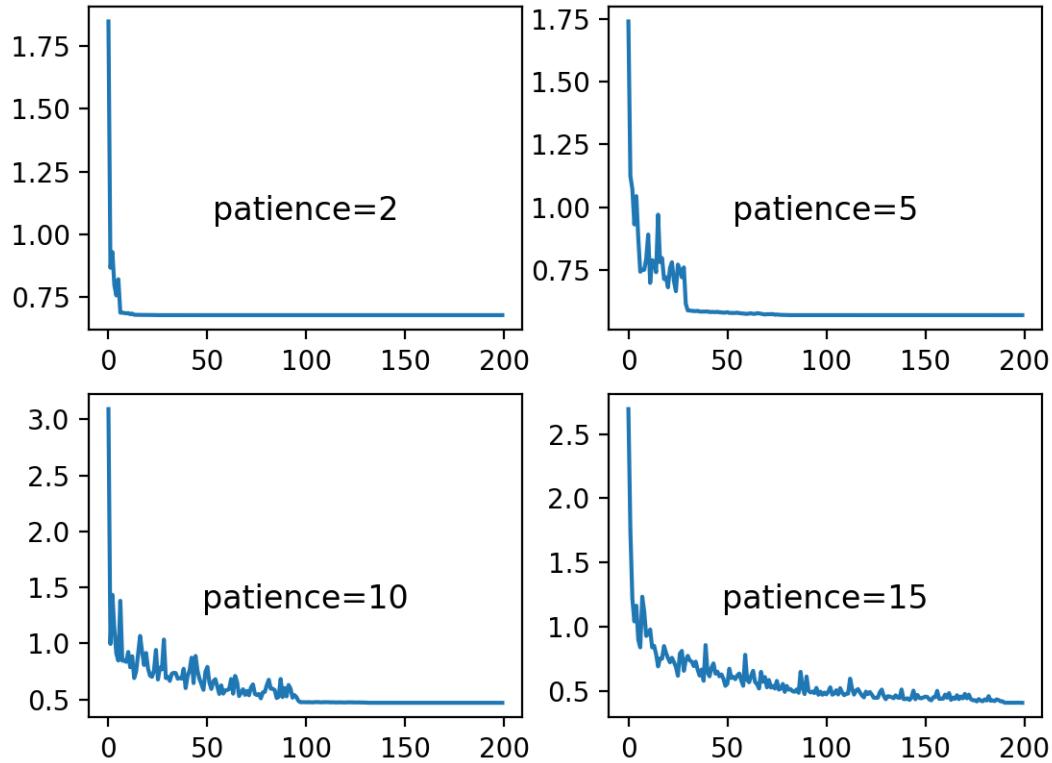


Figure 5.7: Line Plots of Training Loss Over Epochs for Different Patience Values Used in the `ReduceLROnPlateau` Schedule.

The final figure shows the training set accuracy over training epochs for each patience value. We can see that indeed the small patience values of 2 and 5 epochs results in premature convergence of the model to a less-than-optimal model at around 65% and less than 75% accuracy respectively. The larger patience values result in better performing models, with the patience of 10 showing convergence just before 150 epochs, whereas the patience 15 continues to show the effects of a volatile accuracy given the nearly completely unchanged learning rate. These plots show how a learning rate that is decreased a sensible way for the problem and chosen model configuration can result in both a skillful and converged stable set of final weights, a desirable property in a final model at the end of a training run.

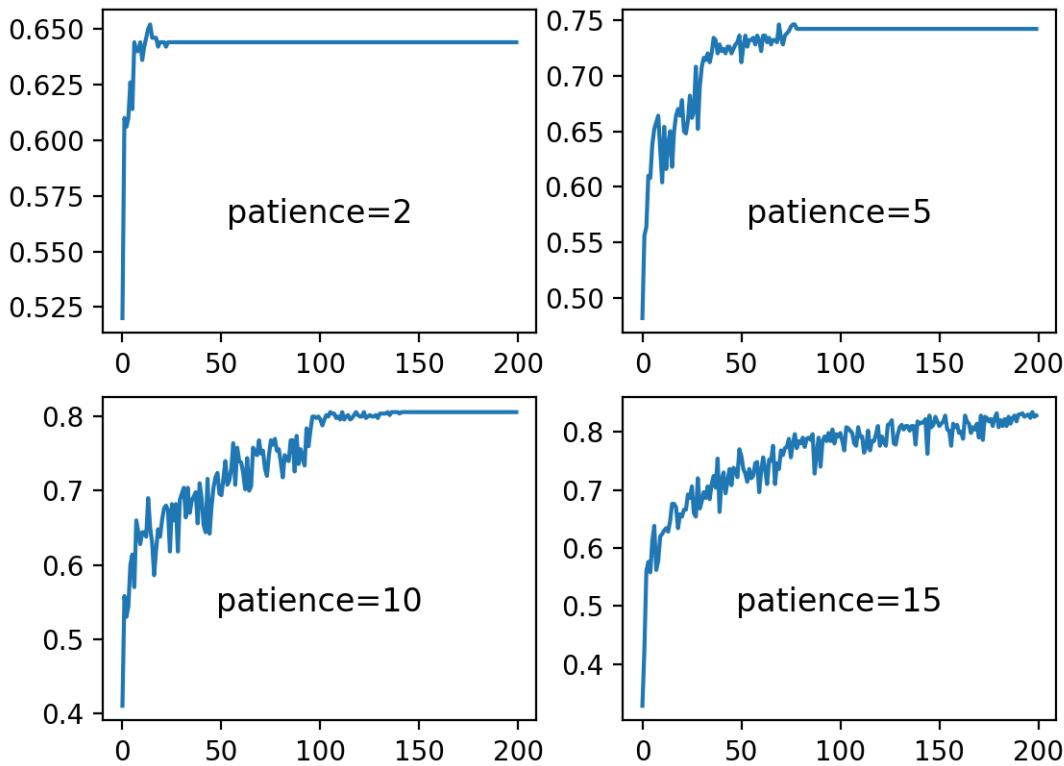


Figure 5.8: Line Plots of Training Accuracy Over Epochs for Different Patience Values Used in the `ReduceLROnPlateau` Schedule.

5.3.4 Effect of Adaptive Learning Rates

Learning rates and learning rate schedules are both challenging to configure and critical to the performance of a deep learning neural network model. Keras provides a number of different popular variations of stochastic gradient descent with adaptive learning rates, such as:

- Adaptive Gradient Algorithm (AdaGrad).
- Root Mean Square Propagation (RMSprop).
- Adaptive Moment Estimation (Adam).

Each provides a different methodology for adapting learning rates for each weight in the network. There is no single best algorithm, and the results of racing optimization algorithms (comparing the performance of many methods) on one problem are unlikely to be transferable to new problems. We can study the dynamics of different adaptive learning rate methods on the blobs problem. The `fit_model()` function can be updated to take the name of an optimization algorithm to evaluate, which can be specified to the `optimizer` argument when the MLP model is compiled. The default parameters for each method will then be used. The updated version of the function is listed below.

```
# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, optimizer):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
    # plot learning curves
    pyplot.plot(history.history['accuracy'], label='train')
    pyplot.plot(history.history['val_accuracy'], label='test')
    pyplot.title('opt=' + optimizer, pad=-80)
```

Listing 5.34: Example of updated function to evaluate different adaptive learning rate techniques.

We can explore the three popular methods of RMSprop, AdaGrad and Adam and compare their behavior to simple stochastic gradient descent with a static learning rate. We would expect the adaptive learning rate versions of the algorithm to perform similarly or better, perhaps adapting to the problem in fewer training epochs, but importantly, to result in a more stable model.

```
# prepare dataset
trainX, trainy, testX, testy = prepare_data()
# create learning curves for different optimizers
optimizers = ['sgd', 'rmsprop', 'adagrad', 'adam']
for i in range(len(optimizers)):
    # determine the plot number
    plot_no = 220 + (i+1)
    pyplot.subplot(plot_no)
    # fit model and plot learning curves for an optimizer
    fit_model(trainX, trainy, testX, testy, optimizers[i])
# show learning curves
pyplot.show()
```

Listing 5.35: Example of evaluating different adaptive learning rate techniques.

Tying these elements together, the complete example is listed below.

```
# study of sgd with adaptive learning rates in the blobs problem
from sklearn.datasets import make_blobs
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare train and test dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
```

```

trainy, testy = y[:n_train], y[n_train:]
return trainX, trainy, testX, testy

# fit a model and plot learning curve
def fit_model(trainX, trainy, testX, testy, optimizer):
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
    # plot learning curves
    pyplot.plot(history.history['accuracy'], label='train')
    pyplot.plot(history.history['val_accuracy'], label='test')
    pyplot.title('opt=' + optimizer, pad=-80)

# prepare dataset
trainX, trainy, testX, testy = prepare_data()
# create learning curves for different optimizers
optimizers = ['sgd', 'rmsprop', 'adagrad', 'adam']
for i in range(len(optimizers)):
    # determine the plot number
    plot_no = 220 + (i+1)
    pyplot.subplot(plot_no)
    # fit model and plot learning curves for an optimizer
    fit_model(trainX, trainy, testX, testy, optimizers[i])
# show learning curves
pyplot.show()

```

Listing 5.36: Example of evaluating the dynamics of adaptive learning rate techniques with an MLP on the blobs problem.

Running the example creates a single figure that contains four line plots for the different evaluated optimization algorithms. Classification accuracy on the training dataset is marked in blue, whereas accuracy on the test dataset is marked in orange.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

Again, we can see that SGD with a default learning rate of 0.01 and no momentum does learn the problem, but requires nearly all 200 epochs and results in volatile accuracy on the training data and much more so on the test dataset. The plots show that all three adaptive learning rate methods learning the problem faster and with dramatically less volatility in train and test set accuracy.

Both RMSProp and Adam demonstrate similar performance, effectively learning the problem within 50 training epochs and spending the remaining training time making very minor weight updates, but not converging as we saw with the learning rate schedules in the previous section.

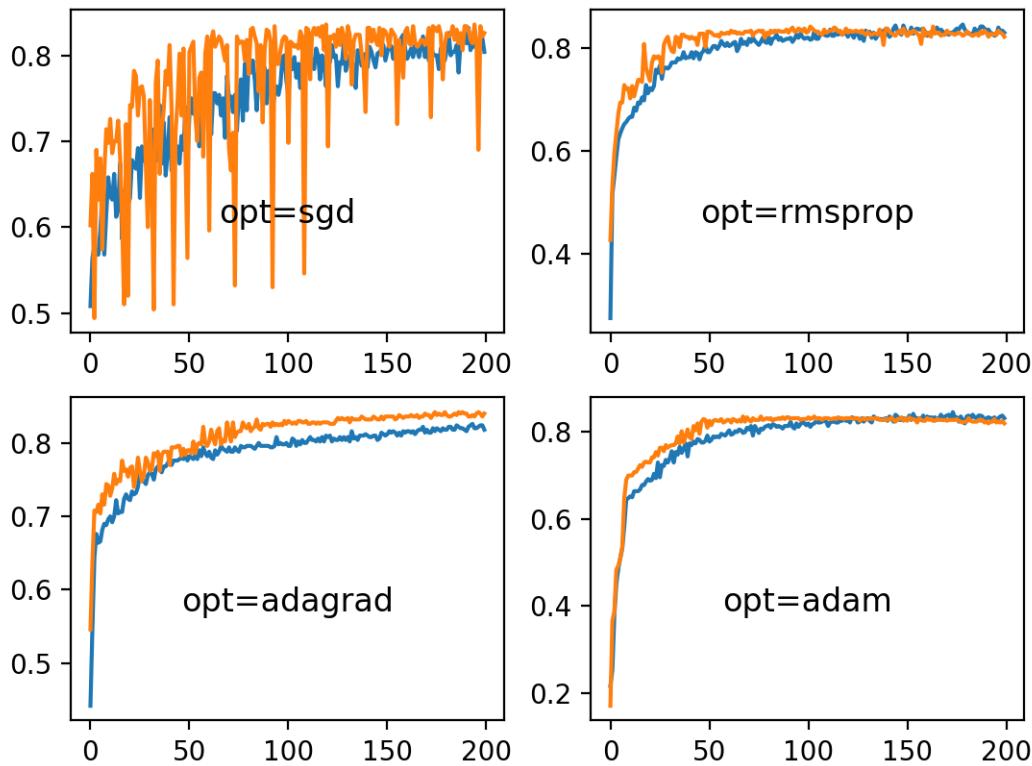


Figure 5.9: Line Plots of Train and Test Accuracy for a Suite of Adaptive Learning Rate Methods on the Blobs Classification Problem.

5.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Tune Learning.** Update the learning rate experiment to automatically vary the learning rate within a discovered best order of magnitude.
- **Initial Learning Rate.** Vary the initial learning for an adaptive learning rate method such as Adam and compare results.

If you explore any of these extensions, I'd love to know.

5.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

5.5.1 Books

- Chapter 8: Optimization for Training Deep Models, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- Chapter 6: Learning Rate and Momentum, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
<https://amzn.to/2S8qRdI>
- Section 5.7: Gradient descent, *Neural Networks for Pattern Recognition*, 1995.
<https://amzn.to/2S8qdwt>

5.5.2 Papers

- *Practical recommendations for gradient-based training of deep architectures*, 2012.
<https://arxiv.org/abs/1206.5533>

5.5.3 APIs

- Keras Optimizers API.
<https://keras.io/optimizers/>
- Keras Callbacks API.
<https://keras.io/callbacks/>
- optimizers.py Keras Source Code.
<https://github.com/keras-team/keras/blob/master/keras/optimizers.py>
- sklearn.datasets.make_blobs API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

5.5.4 Articles

- Stochastic gradient descent, Wikipedia.
https://en.wikipedia.org/wiki/Stochastic_gradient_descent
- What learning rate should be used for backprop?, Neural Network FAQ.
ftp://ftp.sas.com/pub/neural/FAQ2.html#A_learn_rate

5.6 Summary

In this tutorial, you discovered the learning rate hyperparameter used when training deep learning neural networks. Specifically, you learned:

- Learning rate controls how quickly or slowly a neural network model learns a problem.
- How to configure the learning rate with sensible defaults, diagnose behavior, and develop a sensitivity analysis.

- How to further improve performance with learning rate schedules, momentum, and adaptive learning rates.

5.6.1 Next

In the next tutorial, you will discover how the learning process is sensitive the scale of input and target variables, and how data normalization and standardization can have a dramatic effect on convergence.

Chapter 6

Stabilize Learning with Data Scaling

Deep learning neural networks learn how to map inputs to outputs from examples in a training dataset. The weights of the model are initialized to small random values and updated via an optimization algorithm in response to estimates of error on the training dataset. Given the use of small weights in the model and the use of error between predictions and actual values, the scale of inputs and outputs used to train the model are an important factor. Unscaled input variables can result in a slow or unstable learning process, whereas unscaled target variables on regression problems can result in exploding gradients causing the learning process to fail. Data preparation involves using techniques such as normalization and standardization to rescale input and output variables prior to training a neural network model. In this tutorial, you will discover how to improve neural network stability and modeling performance by scaling data. After completing this tutorial, you will know:

- Data scaling is a recommended pre-processing step when working with deep learning neural networks.
- Data scaling can be achieved by normalizing or standardizing real-valued input and output variables.
- How to apply standardization and normalization to improve the performance of a Multilayer Perceptron model on a regression predictive modeling problem.

Let's get started.

6.1 Data Scaling

In this section you will discover the data scaling, the effect it has on the model during training and tips on how to scale input and target variables when training your own neural network models.

6.1.1 The Scale of Your Data Matters

Deep learning neural network models learn a mapping from input variables to an output variable. As such, the scale and distribution of the data may be different for each variable. Input variables may have different units (e.g. feet, kilometers, and hours) that, in turn, may mean the variables

have different scales. Differences in the scales across input variables may increase the difficulty of the problem being modeled. An example of this is that large input values (e.g. a spread of hundreds or thousands of units) can result in a model that learns large weight values. A model with large weight values is often unstable, meaning that it may suffer from poor performance during learning and sensitivity to input values resulting in higher generalization error.

One of the most common forms of pre-processing consists of a simple linear rescaling of the input variables.

— Page 298, *Neural Networks for Pattern Recognition*, 1995.

A target variable with a large spread of values, in turn, may result in large error gradient values causing weight values to change dramatically, making the learning process unstable. Scaling input and output variables is a critical step in using neural network models.

In practice it is nearly always advantageous to apply pre-processing transformations to the input data before it is presented to a network. Similarly, the outputs of the network are often post-processed to give the required output values.

— Page 296, *Neural Networks for Pattern Recognition*, 1995.

6.1.2 Scaling Input Variables

The input variables are those that the network takes on the input or visible layer in order to make a prediction. A good rule of thumb is that input variables should be small values, probably in the range of 0-1 or standardized with a zero mean and a standard deviation of one. Whether input variables require scaling depends on the specifics of your problem and of each variable. You may have a sequence of quantities as inputs, such as prices or temperatures.

Scaling speeds learning because it helps to balance out the rate at which the weights connected to the input nodes learn.

— *Efficient BackProp*, 1998.

If the distribution of the quantity is normal, then it should be standardized, otherwise the data should be normalized. This applies if the range of quantity values is large (10s, 100s, etc.) or small (0.01, 0.0001). If the quantity values are small (near 0-1) and the distribution is limited (e.g. standard deviation near 1) then perhaps you can get away with no scaling of the data. Problems can be complex and it may not be clear how to best scale input data. If in doubt, normalize the input sequence. If you have the resources, explore modeling with the raw data, standardized data, and normalized data and see if there is a beneficial difference in the performance of the resulting model.

If the input variables are combined linearly, as in an MLP [Multilayer Perceptron], then it is rarely strictly necessary to standardize the inputs, at least in theory. [...] However, there are a variety of practical reasons why standardizing the inputs can make training faster and reduce the chances of getting stuck in local optima.

— *Should I normalize/standardize/rescale the data?* Neural Nets FAQ.

6.1.3 Scaling Output Variables

The output variable is the variable predicted by the network. You must ensure that the scale of your output variable matches the scale of the activation function (transfer function) on the output layer of your network.

If your output activation function has a range of [0,1], then obviously you must ensure that the target values lie within that range. But it is generally better to choose an output activation function suited to the distribution of the targets than to force your data to conform to the output activation function.

— *Should I normalize/standardize/rescale the data?* Neural Nets FAQ.

If your problem is a regression problem, then the output will be a real value. This is best modeled with a linear activation function. If the distribution of the value is normal, then you can standardize the output variable. Otherwise, the output variable can be normalized.

6.2 Data Scaling scikit-learn API

There are two types of scaling of your data that you may want to consider: normalization and standardization. These can both be achieved using the scikit-learn library.

6.2.1 Data Normalization

Normalization is a rescaling of the data from the original range so that all values are within the range of 0 and 1. Normalization requires that you know or are able to accurately estimate the minimum and maximum observable values. You may be able to estimate these values from your available data. If an x value to be normalized is outside the bounds of the minimum and maximum values, the resulting value will not be in the range of 0 and 1. You could check for these observations prior to making predictions and either remove them from the dataset or limit them to the pre-defined maximum or minimum values. You can normalize your dataset using the scikit-learn object `MinMaxScaler`. Good practice usage with the `MinMaxScaler` and other scaling techniques is as follows:

- **Fit the scaler using available training data.** For normalization, this means the training data will be used to estimate the minimum and maximum observable values. This is done by calling the `fit()` function.
- **Apply the scale to training data.** This means you can use the normalized data to train your model. This is done by calling the `transform()` function.
- **Apply the scale to data going forward.** This means you can prepare new data in the future on which you want to make predictions.

The default scale for the `MinMaxScaler` is to rescale variables into the range [0, 1], although a preferred scale can be specified via the `feature_range` argument and specify a tuple including the min and the max for all variables.

```
# create scaler
scaler = MinMaxScaler(feature_range=(-1,1))
```

Listing 6.1: Example of defining a `MinMaxScaler`.

If needed, the transform can be inverted. This is useful for converting predictions back into their original scale for reporting or plotting. This can be done by calling the `inverse_transform()` function. The example below provides a general demonstration for using the `MinMaxScaler` to normalize data.

```
# demonstrate data normalization with sklearn
from sklearn.preprocessing import MinMaxScaler
# load data
data = ...
# create scaler
scaler = MinMaxScaler()
# fit scaler on data
scaler.fit(data)
# apply transform
normalized = scaler.transform(data)
# inverse transform
inverse = scaler.inverse_transform(normalized)
```

Listing 6.2: Example of using a `MinMaxScaler`.

You can also perform the fit and transform in a single step using the `fit_transform()` function; for example:

```
# demonstrate data normalization with sklearn
from sklearn.preprocessing import MinMaxScaler
# load data
data = ...
# create scaler
scaler = MinMaxScaler()
# fit and transform in one step
normalized = scaler.fit_transform(data)
# inverse transform
inverse = scaler.inverse_transform(normalized)
```

Listing 6.3: Example of alternate way of using a `MinMaxScaler`.

6.2.2 Data Standardization

Standardizing a dataset involves rescaling the distribution of values so that the mean of observed values is 0 and the standard deviation is 1. It is sometimes referred to as *whitening*. This can be thought of as subtracting the mean value or centering the data. Like normalization, standardization can be useful, and even required in some machine learning algorithms when your data has input values with differing scales. Standardization assumes that your observations fit a Gaussian distribution (bell curve) with a well behaved mean and standard deviation. You can still standardize your data if this expectation is not met, but you may not get reliable results. It may be a preferred data preparation scheme for use with neural networks.

Convergence is usually faster if the average of each input variable over the training set is close to zero.

— Efficient BackProp, 1998.

Standardization requires that you know or are able to accurately estimate the mean and standard deviation of observable values. You may be able to estimate these values from your training data. You can standardize your dataset using the scikit-learn object `StandardScaler`.

```
# demonstrate data standardization with sklearn
from sklearn.preprocessing import StandardScaler
# load data
data = ...
# create scaler
scaler = StandardScaler()
# fit scaler on data
scaler.fit(data)
# apply transform
standardized = scaler.transform(data)
# inverse transform
inverse = scaler.inverse_transform(standardized)
```

Listing 6.4: Example of using a `StandardScaler`.

You can also perform the fit and transform in a single step using the `fit_transform()` function; for example:

```
# demonstrate data standardization with sklearn
from sklearn.preprocessing import StandardScaler
# load data
data = ...
# create scaler
scaler = StandardScaler()
# fit and transform in one step
standardized = scaler.fit_transform(data)
# inverse transform
inverse = scaler.inverse_transform(standardized)
```

Listing 6.5: Example of alternate way of using a `StandardScaler`.

6.3 Data Scaling Case Study

In this section, we will demonstrate how to use data scaling to improve convergence with a MLP on a simple classification problem. This example provides a template for exploring data scaling with your own neural network for classification and regression problems.

6.3.1 Regression Predictive Modeling Problem

A regression predictive modeling problem involves predicting a real-valued quantity. We can use a standard regression problem generator provided by the scikit-learn library in the `make_regression()` function. This function will generate examples from a simple regression problem with a given number of input variables, statistical noise, and other properties. We will use this function to define a problem that has 20 input features; 10 of the features will be meaningful and 10 will not be relevant. A total of 1,000 examples will be randomly generated. The pseudorandom number generator will be fixed to ensure that we get the same 1,000 examples each time the code is run.

```
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
```

Listing 6.6: Example of generating samples for the regression problem.

Each input variable has a Gaussian distribution, as does the target variable. We can demonstrate this by creating histograms of some of the input variables and the output variable.

```
# regression predictive modeling problem
from sklearn.datasets import make_regression
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# histograms of input variables
pyplot.subplot(211)
pyplot.hist(X[:, 0])
pyplot.subplot(212)
pyplot.hist(X[:, 1])
pyplot.show()
# histogram of target variable
pyplot.hist(y)
pyplot.show()
```

Listing 6.7: Example of generating samples and plotting their distribution for the regression problem.

Running the example creates two figures. The first shows histograms of the first two of the twenty input variables, showing that each has a Gaussian data distribution.

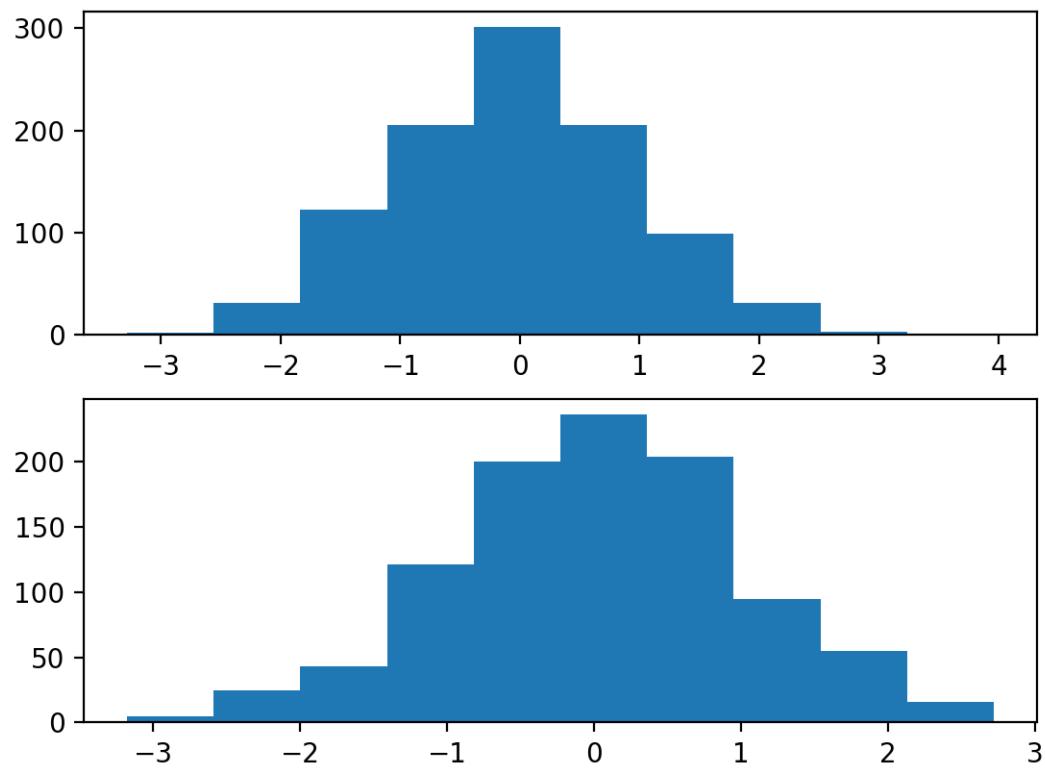


Figure 6.1: Histograms of Two of the Twenty Input Variables for the Regression Problem.

The second figure shows a histogram of the target variable, showing a much larger range for the variable as compared to the input variables and, again, a Gaussian data distribution.

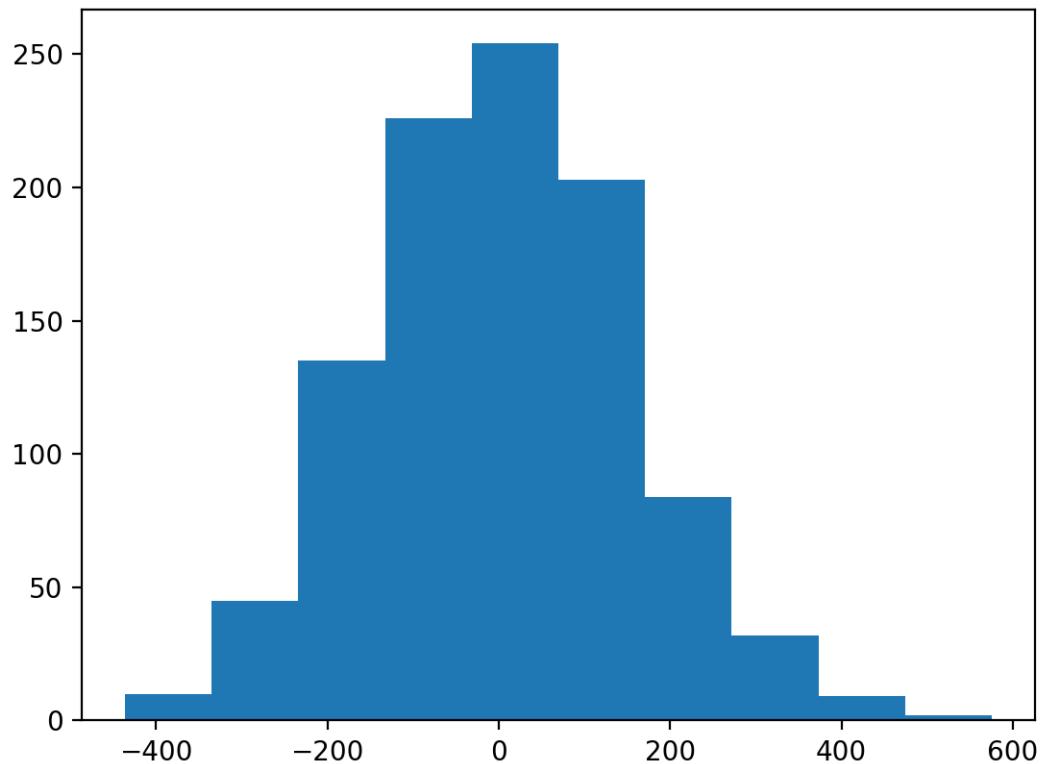


Figure 6.2: Histogram of the Target Variable for the Regression Problem.

Now that we have a regression problem that we can use as the basis for the investigation, we can develop a model to address it.

6.3.2 Multilayer Perceptron With Unscaled Data

We can develop a Multilayer Perceptron (MLP) model for the regression problem. A model will be demonstrated on the raw data, without any scaling of the input or output variables. We expect that model performance will be generally poor. The first step is to split the data into train and test sets so that we can fit and evaluate a model. We will generate 1,000 examples from the domain and split the dataset in half, using 500 examples for the train and test datasets.

```
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 6.8: Example of preparing the dataset for modeling.

Next, we can define an MLP model. The model will expect 20 inputs for the 20 input variables in the problem. A single hidden layer will be used with 25 nodes and a rectified linear activation function. The output layer has one node for the single target variable and a linear activation function to predict real values directly.

```
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
```

Listing 6.9: Example of defining the MLP model.

The mean squared error loss function will be used to optimize the model and the stochastic gradient descent optimization algorithm will be used with the sensible default configuration of a learning rate of 0.01 and a momentum of 0.9.

```
# compile model
model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.01, momentum=0.9))
```

Listing 6.10: Example of compiling the MLP model.

The model will be fit for 100 training epochs and the test set will be used as a validation set, evaluated at the end of each training epoch. The mean squared error is calculated on the train and test datasets at the end of training to get an idea of how well the model learned the problem.

```
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
```

Listing 6.11: Example of evaluating the MLP model.

Finally, learning curves of mean squared error on the train and test sets at the end of each training epoch are graphed using line plots, providing learning curves to get an idea of the dynamics of the model while learning the problem.

```
# plot loss during training
pyplot.title('Mean Squared Error')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 6.12: Example of plotting learning curves for the MLP model.

Tying these elements together, the complete example is listed below.

```
# mlp with unscaled data for the regression problem
from sklearn.datasets import make_regression
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
```

```

model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.01, momentum=0.9))
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.title('Mean Squared Error')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 6.13: Example of evaluating an MLP model on the unscaled regression problem.

Running the example fits the model and calculates the mean squared error on the train and test sets. In this case, the model is unable to learn the problem, resulting in predictions of NaN values. The model weights exploded during training given the very large errors and, in turn, error gradients calculated for weight updates.

```
Train: nan, Test: nan
```

Listing 6.14: Example output from evaluating an MLP model on the unscaled regression problem.

This demonstrates that, at the very least, some data scaling is required for the target variable. A line plot of training history is created but does not show anything as the model almost immediately results in a NaN mean squared error.

6.3.3 Multilayer Perceptron With Scaled Output Variables

The example can be updated to scale the target variable. Reducing the scale of the target variable will, in turn, reduce the size of the gradient used to update the weights and result in a more stable model and training process. Given the Gaussian distribution of the target variable, a natural method for rescaling the variable would be to standardize the variable. This requires estimating the mean and standard deviation of the variable and using these estimates to perform the rescaling. It is best practice is to estimate the mean and standard deviation of the training dataset and use these variables to scale the train and test dataset. This is to avoid any data leakage during the model evaluation process. The scikit-learn transformers expect input data to be matrices of rows and columns, therefore the 1D arrays for the target variable will have to be reshaped into 2D arrays prior to the transforms.

```

# reshape 1d arrays to 2d arrays
trainy = trainy.reshape(len(trainy), 1)
testy = testy.reshape(len(trainy), 1)

```

Listing 6.15: Example of reshaping the target variables.

We can then create and apply the `StandardScaler` to rescale the target variable.

```

# created scaler
scaler = StandardScaler()
# fit scaler on training dataset

```

```

scaler.fit(trainy)
# transform training dataset
trainy = scaler.transform(trainy)
# transform test dataset
testy = scaler.transform(testy)

```

Listing 6.16: Example of standardizing the target variables.

Rescaling the target variable means that estimating the performance of the model and plotting the learning curves will calculate an MSE in squared units of the scaled variable rather than squared units of the original scale. This can make interpreting the error within the context of the domain challenging. In practice, it may be helpful to estimate the performance of the model by first inverting the transform on the test dataset target variable and on the model predictions and estimating model performance using the root mean squared error on the unscaled data. This is left as an exercise to the reader. The complete example of standardizing the target variable for the MLP on the regression problem is listed below.

```

# mlp with scaled outputs on the regression problem
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# reshape 1d arrays to 2d arrays
trainy = trainy.reshape(len(trainy), 1)
testy = testy.reshape(len(trainy), 1)
# created scaler
scaler = StandardScaler()
# fit scaler on training dataset
scaler.fit(trainy)
# transform training dataset
trainy = scaler.transform(trainy)
# transform test dataset
testy = scaler.transform(testy)
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.01, momentum=0.9))
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.title('Mean Squared Error Loss')

```

```
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 6.17: Example of evaluating an MLP model regression problem with standardized target variables.

Running the example fits the model and calculates the mean squared error on the train and test sets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the model does appear to learn the problem and achieves near-zero mean squared error, at least to three decimal places.

```
Train: 0.003, Test: 0.007
```

Listing 6.18: Example output from evaluating an MLP model regression problem with standardized target variables.

A line plot of the mean squared error on the train (blue) and test (orange) dataset over each training epoch is created. In this case, we can see that the model rapidly learns to effectively map inputs to outputs for the regression problem and achieves good performance on both datasets over the course of the run, neither overfitting or underfitting the training dataset.

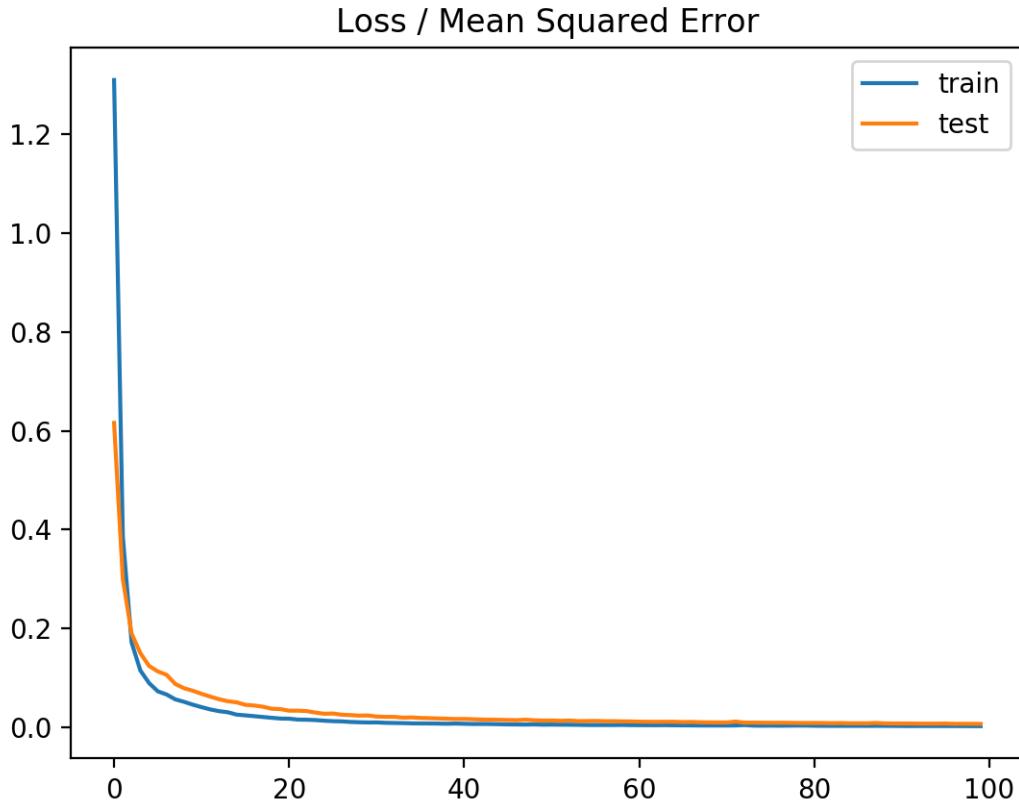


Figure 6.3: Line Plot of Mean Squared Error on the Train and Test Datasets for Each Training Epoch.

It may be interesting to repeat this experiment and normalize the target variable instead and compare results.

6.3.4 Multilayer Perceptron With Scaled Input Variables

We have seen that data scaling can stabilize the training process when fitting a model for regression with a target variable that has a wide spread. It is also possible to improve the stability and performance of the model by scaling the input variables. In this section, we will design an experiment to compare the performance of different scaling methods for the input variables. The input variables also have a Gaussian data distribution, like the target variable, therefore we would expect that standardizing the data would be the best approach. This is just a heuristic and it is always best to evaluate different scaling methods and discover what actually works best.

We can compare the performance of the unscaled input variables to models fit with either standardized or normalized input variables. The first step is to define a function to create the same 1,000 data samples, split them into train and test sets, and apply the data scaling methods specified via input arguments. The `get_dataset()` function below implements this, requiring the scaler to be provided for the input and target variables and returns the train and test datasets split into input and output components ready to train and evaluate a model.

```
# prepare dataset with input and output scalers, can be none
def get_dataset(input_scaler, output_scaler):
    # generate dataset
    X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    # scale inputs
    if input_scaler is not None:
        # fit scaler
        input_scaler.fit(trainX)
        # transform training dataset
        trainX = input_scaler.transform(trainX)
        # transform test dataset
        testX = input_scaler.transform(testX)
    if output_scaler is not None:
        # reshape 1d arrays to 2d arrays
        trainy = trainy.reshape(len(trainy), 1)
        testy = testy.reshape(len(testy), 1)
        # fit scaler on training dataset
        output_scaler.fit(trainy)
        # transform training dataset
        trainy = output_scaler.transform(trainy)
        # transform test dataset
        testy = output_scaler.transform(testy)
    return trainX, trainy, testX, testy
```

Listing 6.19: Example of a function for scaling the data for modeling.

Next, we can define a function to fit an MLP model on a given dataset and return the mean squared error for the fit model on the test dataset. The `evaluate_model()` function below implements this behavior.

```
# fit and evaluate mse of model on test set
def evaluate_model(trainX, trainy, testX, testy):
    # define model
    model = Sequential()
    model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(1, activation='linear'))
    # compile model
    model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.01, momentum=0.9))
    # fit model
    model.fit(trainX, trainy, epochs=100, verbose=0)
    # evaluate the model
    test_mse = model.evaluate(testX, testy, verbose=0)
    return test_mse
```

Listing 6.20: Example of a function for fitting and evaluating an MLP model.

Neural networks are trained using a stochastic learning algorithm. This means that the same model fit on the same data may result in a different performance. We can address this in our experiment by repeating the evaluation of each model configuration, in this case a choice of data scaling, multiple times and report performance as the mean of the error scores across all of the runs. We will repeat each run 30 times to ensure the mean is statistically robust.

The `repeated_evaluation()` function below implements this, taking the scaler for input and output variables as arguments, evaluating a model 30 times with those scalers, printing error scores along the way, and returning a list of the calculated error scores from each run.

```
# evaluate model multiple times with given input and output scalers
def repeated_evaluation(input_scaler, output_scaler, n_repeats=30):
    # get dataset
    trainX, trainy, testX, testy = get_dataset(input_scaler, output_scaler)
    # repeated evaluation of model
    results = []
    for _ in range(n_repeats):
        test_mse = evaluate_model(trainX, trainy, testX, testy)
        print('%.3f' % test_mse)
        results.append(test_mse)
    return results
```

Listing 6.21: Example of a function the repeated evaluation of an MLP model.

Finally, we can run the experiment and evaluate the same model on the same dataset three different ways:

- No scaling of inputs, standardized outputs.
- Normalized inputs, standardized outputs.
- Standardized inputs, standardized outputs.

The mean and standard deviation of the error for each configuration is reported, then box and whisker plots are created to summarize the error scores for each configuration.

```
# unscaled inputs
results_unscaled_inputs = repeated_evaluation(None, StandardScaler())
# normalized inputs
results_normalized_inputs = repeated_evaluation(StandardScaler(), StandardScaler())
# standardized inputs
results_standardized_inputs = repeated_evaluation(StandardScaler(), StandardScaler())
# summarize results
print('Unscaled: %.3f (%.3f)' % (mean(results_unscaled_inputs),
    std(results_unscaled_inputs)))
print('Normalized: %.3f (%.3f)' % (mean(results_normalized_inputs),
    std(results_normalized_inputs)))
print('Standardized: %.3f (%.3f)' % (mean(results_standardized_inputs),
    std(results_standardized_inputs)))
# plot results
results = [results_unscaled_inputs, results_normalized_inputs, results_standardized_inputs]
labels = ['unscaled', 'normalized', 'standardized']
pyplot.boxplot(results, labels=labels)
pyplot.show()
```

Listing 6.22: Example of a evaluating a range of different input scaling procedures.

Tying these elements together, the complete example is listed below.

```
# compare scaling methods for mlp inputs on regression problem
from sklearn.datasets import make_regression
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
```

```
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from matplotlib import pyplot
from numpy import mean
from numpy import std

# prepare dataset with input and output scalers, can be none
def get_dataset(input_scaler, output_scaler):
    # generate dataset
    X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    # scale inputs
    if input_scaler is not None:
        # fit scaler
        input_scaler.fit(trainX)
        # transform training dataset
        trainX = input_scaler.transform(trainX)
        # transform test dataset
        testX = input_scaler.transform(testX)
    if output_scaler is not None:
        # reshape 1d arrays to 2d arrays
        trainy = trainy.reshape(len(trainy), 1)
        testy = testy.reshape(len(trainy), 1)
        # fit scaler on training dataset
        output_scaler.fit(trainy)
        # transform training dataset
        trainy = output_scaler.transform(trainy)
        # transform test dataset
        testy = output_scaler.transform(testy)
    return trainX, trainy, testX, testy

# fit and evaluate mse of model on test set
def evaluate_model(trainX, trainy, testX, testy):
    # define model
    model = Sequential()
    model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(1, activation='linear'))
    # compile model
    model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.01, momentum=0.9))
    # fit model
    model.fit(trainX, trainy, epochs=100, verbose=0)
    # evaluate the model
    test_mse = model.evaluate(testX, testy, verbose=0)
    return test_mse

# evaluate model multiple times with given input and output scalers
def repeated_evaluation(input_scaler, output_scaler, n_repeats=30):
    # get dataset
    trainX, trainy, testX, testy = get_dataset(input_scaler, output_scaler)
    # repeated evaluation of model
    results = list()
    for _ in range(n_repeats):
```

```

test_mse = evaluate_model(trainX, trainy, testX, testy)
print('%.3f' % test_mse)
results.append(test_mse)
return results

# unscaled inputs
results_unscaled_inputs = repeated_evaluation(None, StandardScaler())
# normalized inputs
results_normalized_inputs = repeated_evaluation(StandardScaler(), StandardScaler())
# standardized inputs
results_standardized_inputs = repeated_evaluation(StandardScaler(), StandardScaler())
# summarize results
print('Unscaled: %.3f (%.3f)' % (mean(results_unscaled_inputs),
    std(results_unscaled_inputs)))
print('Normalized: %.3f (%.3f)' % (mean(results_normalized_inputs),
    std(results_normalized_inputs)))
print('Standardized: %.3f (%.3f)' % (mean(results_standardized_inputs),
    std(results_standardized_inputs)))
# plot results
results = [results_unscaled_inputs, results_normalized_inputs, results_standardized_inputs]
labels = ['unscaled', 'normalized', 'standardized']
pyplot.boxplot(results, labels=labels)
pyplot.show()

```

Listing 6.23: Example of evaluating an MLP model with different methods for scaling the input.

Running the example prints the mean squared error for each model run along the way. After each of the three configurations have been evaluated 30 times each, the mean errors for each are reported.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that as we expected, scaling the input variables does result in a model with better performance. Unexpectedly, better performance is seen using normalized inputs instead of standardized inputs. This may be related to the choice of the rectified linear activation function in the first hidden layer.

```

...
>0.010
>0.012
>0.005
>0.008
>0.008
Unscaled: 0.007 (0.004)
Normalized: 0.001 (0.000)
Standardized: 0.008 (0.004)

```

Listing 6.24: Example output from evaluating an MLP model with different methods for scaling the input.

A figure with three box and whisker plots is created summarizing the spread of error scores for each configuration. The plots show that there was little difference between the distributions of error scores for the unscaled and standardized input variables, and that the normalized input variables result in better performance and more stable or a tighter distribution of error scores.

These results highlight that it is important to actually experiment and confirm the results of data scaling methods rather than assuming that a given data preparation scheme will work best based on the observed distribution of the data.

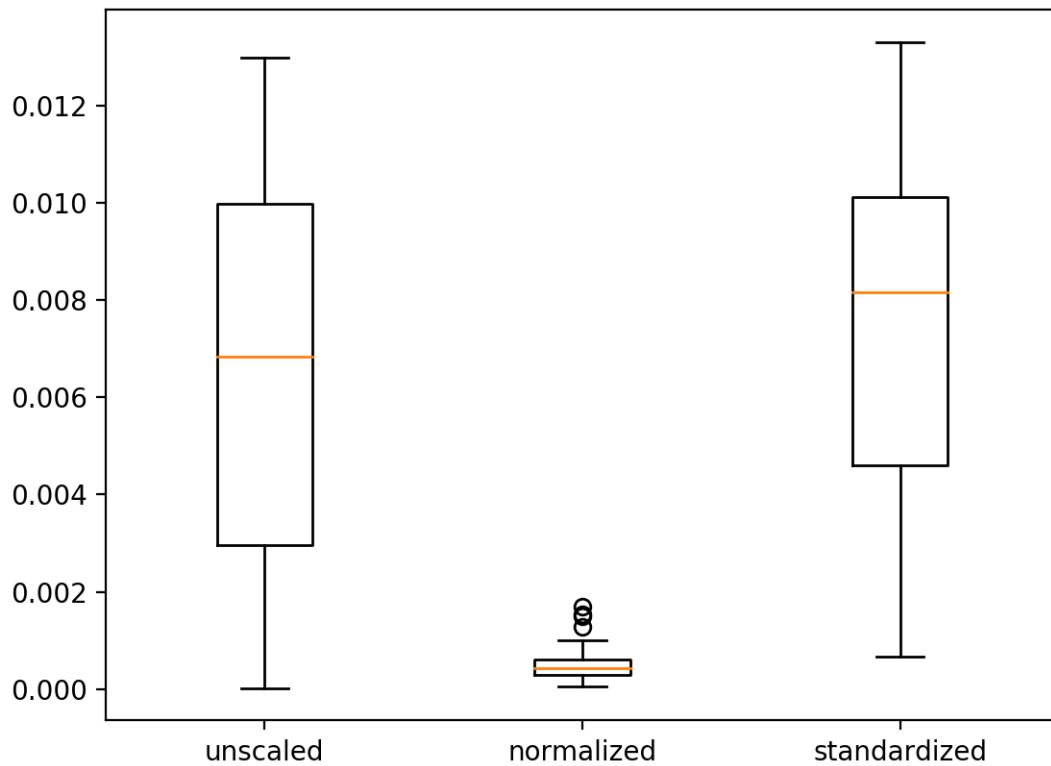


Figure 6.4: Box and Whisker Plots of Mean Squared Error With Unscaled, Normalized and Standardized Input Variables for the Regression Problem.

6.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Normalize Target Variable.** Update the example and normalize instead of standardize the target variable and compare results.
- **Compared Scaling for Target Variable.** Update the example to compare standardizing and normalizing the target variable using repeated experiments and compare the results.
- **Other Scales.** Update the example to evaluate other min/max scales when normalizing and compare performance, e.g. [-1, 1] and [0.0, 0.5].

If you explore any of these extensions, I'd love to know.

6.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

6.5.1 Books

- Section 8.2: Input normalization and encoding, *Neural Networks for Pattern Recognition*, 1995.
<https://amzn.to/2S8qdwt>

6.5.2 Papers

- *Efficient BackProp*, 1998.
<http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>

6.5.3 APIs

- `sklearn.datasets.make_regression` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html
- `sklearn.preprocessing.MinMaxScaler` API.
<http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>
- `sklearn.preprocessing.StandardScaler` API.
<http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>

6.5.4 Articles

- Should I normalize/standardize/rescale the data? Neural Nets FAQ.
ftp://ftp.sas.com/pub/neural/FAQ2.html#A_std

6.6 Summary

In this tutorial, you discovered how to improve neural network stability and modeling performance by scaling data. Specifically, you learned:

- Data scaling is a recommended pre-processing step when working with deep learning neural networks.
- Data scaling can be achieved by normalizing or standardizing real-valued input and output variables.
- How to apply standardization and normalization to improve the performance of a Multilayer Perceptron model on a regression predictive modeling problem.

6.6.1 Next

In the next tutorial, you will discover how to solve the vanishing gradients problem via the use of the rectified linear activation function.

Chapter 7

Fix Vanishing Gradients with ReLU

In a neural network, the activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input. The rectified linear activation function is a piecewise linear function that will output the input directly if is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance. In this tutorial, you will discover the rectified linear activation function for deep learning neural networks. After completing this tutorial, you will know:

- The sigmoid and hyperbolic tangent activation functions cannot be used in networks with many layers due to the vanishing gradient problem.
- The rectified linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better.
- The rectified linear activation is the default activation when developing Multilayer Perceptron and convolutional neural networks.

Let's get started.

7.1 Vanishing Gradients and ReLU

In this section you will discover problem of vanishing gradients, the effect it has on the model during training and the use of the rectified linear activation function to address the problem.

7.1.1 Limitations of Sigmoid and Tanh Activation Functions

A neural network is comprised of layers of nodes and learns to map examples of inputs to outputs. For a given node, the inputs are multiplied by the weights in a node and summed together. This value is referred to as the summed activation of the node. The summed activation is then transformed via an activation function and defines the specific output or *activation* of the node. The simplest activation function is referred to as the linear activation, where no transform is applied at all. A network comprised of only linear activation functions is very easy to train, but cannot learn complex mapping functions. Linear activation functions are still used in the output layer for networks that predict a quantity (e.g. regression problems).

Nonlinear activation functions are preferred as they allow the nodes to learn more complex structures in the data. Traditionally, two widely used nonlinear activation functions are the sigmoid and hyperbolic tangent activation functions. The sigmoid activation function, also called the logistic function, is traditionally a very popular activation function for neural networks. The input to the function is transformed into a value between 0.0 and 1.0. Inputs that are much larger than 1.0 are transformed to the value 1.0, similarly, values much smaller than 0.0 are snapped to 0.0. The shape of the function for all possible inputs is an S-shape from zero up through 0.5 to 1.0. For a long time, through the early 1990s, it was the default activation used on neural networks. The hyperbolic tangent function, or tanh for short, is a similar shaped nonlinear activation function that outputs values between -1.0 and 1.0. In the later 1990s and through the 2000s, the tanh function was preferred over the sigmoid activation function as models that used it were easier to train and often had better predictive performance.

... the hyperbolic tangent activation function typically performs better than the logistic sigmoid.

— Page 195, *Deep Learning*, 2016.

A general problem with both the sigmoid and tanh functions is that they saturate. This means that large values snap to 1.0 and small values snap to -1 or 0 for tanh and sigmoid respectively. Further, the functions are only really sensitive to changes around the mid-point of their input, such as 0.5 for sigmoid and 0.0 for tanh. The limited sensitivity and saturation of the function happen regardless of whether the summed activation from the node provided as input contains useful information or not. Once saturated, it becomes challenging for the learning algorithm to continue to adapt the weights to improve the performance of the model.

... sigmoidal units saturate across most of their domain—they saturate to a high value when z is very positive, saturate to a low value when z is very negative, and are only strongly sensitive to their input when z is near 0.

— Page 195, *Deep Learning*, 2016.

Finally, as the capability of hardware increased through GPUs, very deep neural networks using sigmoid and tanh activation functions could not easily be trained. Layers deep in large networks using these nonlinear activation functions fail to receive useful gradient information. Error is back propagated through the network and used to update the weights. The amount of error decreases dramatically with each additional layer through which it is propagated, given the derivative of the chosen activation function. This is called the vanishing gradient problem and prevents deep (multilayered) networks from learning effectively.

Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function

— Page 290, *Deep Learning*, 2016.

Although the use of nonlinear activation functions allows neural networks to learn complex mapping functions, they effectively prevent the learning algorithm from working with deep networks. Workarounds were found in the late 2000s and early 2010s using alternate network types such as Boltzmann machines and layer-wise training or unsupervised pre-training.

7.1.2 Rectified Linear Activation Function

In order to use stochastic gradient descent with backpropagation of errors to train deep neural networks, an activation function is needed that looks and acts like a linear function, but is, in fact, a nonlinear function allowing complex relationships in the data to be learned. The function must also provide more sensitivity to the activation sum input and avoid easy saturation. The solution had been bouncing around in the field for some time, although was not highlighted until papers in 2009 and 2011 shone a light on it. The solution is to use the rectified linear activation function, or ReL for short. A node or unit that implements this activation function is referred to as a rectified linear activation unit, or ReLU for short. Often, networks that use the rectifier function for the hidden layers are referred to as rectified networks.

Adoption of ReLU may easily be considered one of the few milestones in the deep learning revolution, e.g. the techniques that now permit the routine development of very deep neural networks.

[another] major algorithmic change that has greatly improved the performance of feedforward networks was the replacement of sigmoid hidden units with piecewise linear hidden units, such as rectified linear units.

— Page 226, *Deep Learning*, 2016.

The rectified linear activation function is a simple calculation that returns the value provided as input directly, or the value 0.0 if the input is 0.0 or less. We can describe this using a simple if-statement:

```
if input > 0:
    return input
else:
    return 0
```

Listing 7.1: Example of rectified linear activation function with an if-statement.

We can describe this function $g()$ mathematically using the `max()` function over the set of 0.0 and the input z ; for example:

$$g(z) = \max\{0, z\} \quad (7.1)$$

The function is linear for values greater than zero, meaning it has a lot of the desirable properties of a linear activation function when training a neural network using backpropagation. Yet, it is a nonlinear function as negative values are always output as zero.

Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods. They also preserve many of the properties that make linear models generalize well.

— Page 175, *Deep Learning*, 2016.

Because the rectified function is linear for half of the input domain and nonlinear for the other half, it is referred to as a piecewise linear function or a hinge function.

However, the function remains very close to linear, in the sense that is a piecewise linear function with two linear pieces.

— Page 175, *Deep Learning*, 2016.

Now that we are familiar with the rectified linear activation function, let's look at how we can implement it in Python.

7.1.3 How to Implement the Rectified Linear Activation Function

We can implement the rectified linear activation function easily in Python. Perhaps the simplest implementation is using the `max()` function; for example:

```
# rectified linear function
def rectified(x):
    return max(0.0, x)
```

Listing 7.2: Example implementation of the rectified linear activation function.

We expect that any positive value will be returned unchanged whereas an input value of 0.0 or a negative value will be returned as the value 0.0. Below are a few examples of inputs and outputs of the rectified linear activation function.

```
# demonstrate the rectified linear function

# rectified linear function
def rectified(x):
    return max(0.0, x)

# demonstrate with a positive input
x = 1.0
print('rectified(%.1f) is %.1f' % (x, rectified(x)))
x = 1000.0
print('rectified(%.1f) is %.1f' % (x, rectified(x)))
# demonstrate with a zero input
x = 0.0
print('rectified(%.1f) is %.1f' % (x, rectified(x)))
# demonstrate with a negative input
x = -1.0
print('rectified(%.1f) is %.1f' % (x, rectified(x)))
x = -1000.0
print('rectified(%.1f) is %.1f' % (x, rectified(x)))
```

Listing 7.3: Example of transforms with the rectified linear activation function.

Running the example, we can see that positive values are returned regardless of their size, whereas negative values are snapped to the value 0.0.

```
rectified(1.0) is 1.0
rectified(1000.0) is 1000.0
rectified(0.0) is 0.0
rectified(-1.0) is 0.0
rectified(-1000.0) is 0.0
```

Listing 7.4: Example output from transforms with the rectified linear activation function.

We can get an idea of the relationship between inputs and outputs of the function by plotting a series of inputs and the calculated outputs. The example below generates a series of integers from -10 to 10 and calculates the rectified linear activation for each input, then plots the result.

```
# plot inputs and outputs
from matplotlib import pyplot

# rectified linear function
def rectified(x):
    return max(0.0, x)

# define a series of inputs
series_in = [x for x in range(-10, 11)]
# calculate outputs for our inputs
series_out = [rectified(x) for x in series_in]
# line plot of raw inputs to rectified outputs
pyplot.plot(series_in, series_out)
pyplot.show()
```

Listing 7.5: Example of plotting transforms with the rectified linear activation function.

Running the example creates a line plot showing that all negative values and zero inputs are snapped to 0.0, whereas the positive outputs are returned as-is, resulting in a linearly increasing slope, given that we created a linearly increasing series of positive values (e.g. 1 to 10).

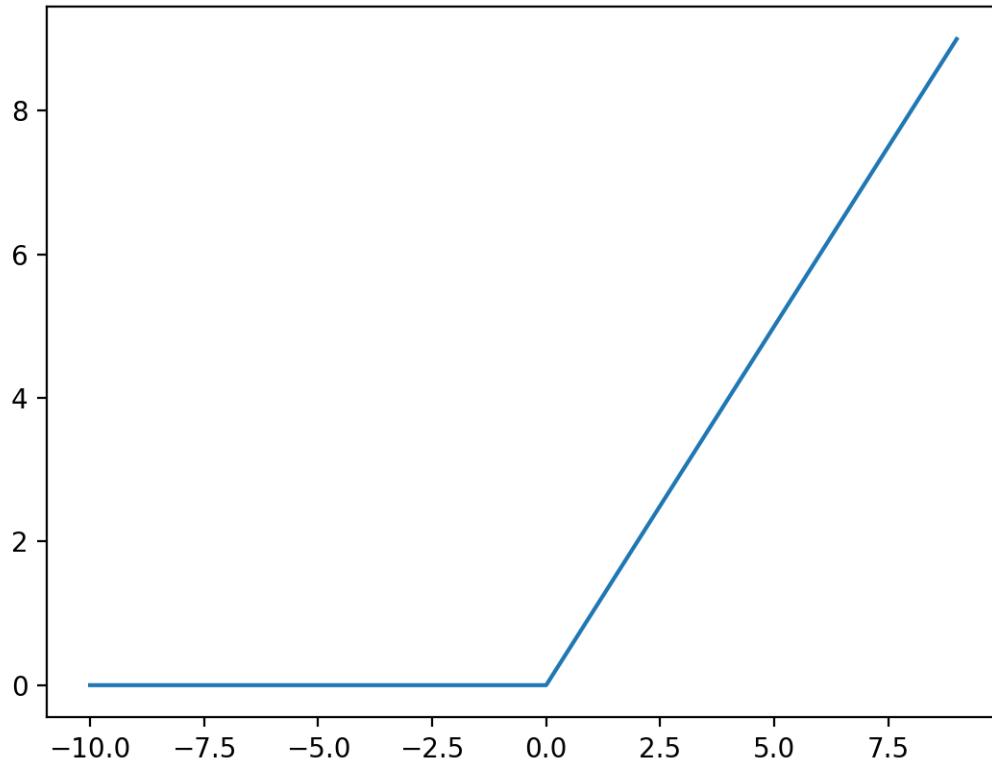


Figure 7.1: Line Plot of Rectified Linear Activation for Negative and Positive Inputs.

The derivative of the rectified linear function is also easy to calculate. Recall that the derivative of the activation function is required when updating the weights of a node as part of

the backpropagation of error. The derivative of the function is the slope. The slope for negative values is 0.0 and the slope for positive values is 1.0. Traditionally, the field of neural networks has avoided any activation function that was not completely differentiable, perhaps delaying the adoption of the rectified linear function and other piecewise-linear functions. Technically, we cannot calculate the derivative when the input is 0.0, therefore, we can assume it is zero. This is not a problem in practice.

For example, the rectified linear function $g(z) = \max\{0, z\}$ is not differentiable at $z = 0$. This may seem like it invalidates g for use with a gradient-based learning algorithm. In practice, gradient descent still performs well enough for these models to be used for machine learning tasks.

— Page 192, *Deep Learning*, 2016.

Using the rectified linear activation function offers many advantages; let's take a look at a few in the next section.

7.1.4 Advantages of the Rectified Linear Activation Function

The rectified linear activation function has rapidly become the default activation function when developing most types of neural networks. As such, it is important to take a moment to review some of the benefits of the approach, first highlighted by Xavier Glorot, et al. in their milestone 2012 paper on using ReLU titled *Deep Sparse Rectifier Neural Networks*.

Computational Simplicity

The rectifier function is trivial to implement, requiring a `max()` function. This is unlike the tanh and sigmoid activation function that require the use of an exponential calculation.

Computations are also cheaper: there is no need for computing the exponential function in activations

— *Deep Sparse Rectifier Neural Networks*, 2011.

Representational Sparsity

An important benefit of the rectifier function is that it is capable of outputting a true zero value. This is unlike the tanh and sigmoid activation functions that learn to approximate a zero output, e.g. a value very close to zero, but not a true zero value. This means that negative inputs can output true zero values allowing the activation of hidden layers in neural networks to contain one or more true zero values. This is called a sparse representation and is a desirable property in representational learning as it can accelerate learning and simplify the model. An area where efficient representations such as sparsity are studied and sought is in autoencoders, where a network learns a compact representation of an input (called the code layer), such as an image or series, before it is reconstructed from the compact representation.

One way to achieve actual zeros in h for sparse (and denoising) autoencoders [...] The idea is to use rectified linear units to produce the code layer. With a prior that actually pushes the representations to zero (like the absolute value penalty), one can thus indirectly control the average number of zeros in the representation.

— Page 507, *Deep Learning*, 2016.

Linear Behavior

The rectifier function mostly looks and acts like a linear activation function. In general, a neural network is easier to optimize when its behavior is linear or close to linear.

Rectified linear units [...] are based on the principle that models are easier to optimize if their behavior is closer to linear.

— Page 194, *Deep Learning*, 2016.

Key to this property is that networks trained with this activation function almost completely avoid the problem of vanishing gradients, as the gradients remain proportional to the node activations.

Because of this linearity, gradients flow well on the active paths of neurons (there is no gradient vanishing effect due to activation non-linearities of sigmoid or tanh units).

— *Deep Sparse Rectifier Neural Networks*, 2011.

Train Deep Networks

Importantly, the (re-)discovery and adoption of the rectified linear activation function meant that it became possible to exploit improvements in hardware and successfully train deep multilayered networks with a nonlinear activation function using backpropagation. In turn, cumbersome networks such as Boltzmann machines could be left behind as well as cumbersome training schemes such as layer-wise training and unlabeled pre-training.

... deep rectifier networks can reach their best performance without requiring any unsupervised pre-training on purely supervised tasks with large labeled datasets. Hence, these results can be seen as a new milestone in the attempts at understanding the difficulty in training deep but purely supervised neural networks, and closing the performance gap between neural networks learnt with and without unsupervised pre-training.

— *Deep Sparse Rectifier Neural Networks*, 2011.

7.1.5 Tips for Using the Rectified Linear Activation

In this section, we'll take a look at some tips when using the rectified linear activation function in your own deep learning neural networks.

Use ReLU as the Default Activation Function

For a long time, the default activation to use was the sigmoid activation function. Later, it was the tanh activation function. For modern deep learning neural networks, the default activation function is the rectified linear activation function.

Prior to the introduction of rectified linear units, most neural networks used the logistic sigmoid activation function or the hyperbolic tangent activation function.

— Page 195, *Deep Learning*, 2016.

Most papers that achieve state-of-the-art results will describe a network using ReLU. For example, in the milestone 2012 paper by Alex Krizhevsky, et al. titled *ImageNet Classification with Deep Convolutional Neural Networks*, the authors developed a deep convolutional neural network with ReLU activations that achieved state-of-the-art results on the ImageNet photo classification dataset.

... we refer to neurons with this nonlinearity as Rectified Linear Units (ReLUs). Deep convolutional neural networks with ReLUs train several times faster than their equivalents with tanh units.

If in doubt, start with ReLU in your neural network, then perhaps try other piecewise linear activation functions to see how their performance compares.

In modern neural networks, the default recommendation is to use the rectified linear unit or ReLU

— Page 174, *Deep Learning*, 2016.

Use ReLU with MLPs, CNNs, but Probably Not RNNs

The ReLU can be used with most types of neural networks. It is recommended as the default for both Multilayer Perceptron (MLP) and Convolutional Neural Networks (CNNs). The use of ReLU with CNNs has been investigated thoroughly, and almost universally results in an improvement in results, initially, surprisingly so.

... how do the non-linearities that follow the filter banks influence the recognition accuracy. The surprising answer is that using a rectifying non-linearity is the single most important factor in improving the performance of a recognition system.

— *What is the best multi-stage architecture for object recognition?*, 2009.

Work investigating ReLU with CNNs is what provoked their use with other network types.

[others] have explored various rectified nonlinearities [...] in the context of convolutional networks and have found them to improve discriminative performance.

— *Rectified Linear Units Improve Restricted Boltzmann Machines*, 2010.

When using ReLU with CNNs, they can be used as the activation function on the filter maps themselves, followed then by a pooling layer.

A typical layer of a convolutional network consists of three stages [...] In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the detector stage.

— Page 339, *Deep Learning*, 2016.

Traditionally, LSTMs use the tanh activation function for the activation of the cell state and the sigmoid activation function for the node output. Given their careful design, ReLU were thought to not be appropriate for Recurrent Neural Networks (RNNs) such as the Long Short-Term Memory Network (LSTM) by default.

At first sight, ReLUs seem inappropriate for RNNs because they can have very large outputs so they might be expected to be far more likely to explode than units that have bounded values.

— *A Simple Way to Initialize Recurrent Networks of Rectified Linear Units*, 2015.

Nevertheless, there has been some work on investigating the use of ReLU as the output activation in LSTMs, the result of which is a careful initialization of network weights to ensure that the network is stable prior to training. This is outlined in the 2015 paper titled *A Simple Way to Initialize Recurrent Networks of Rectified Linear Units*.

Try a Smaller Bias Input Value

The bias is the input on the node that has a fixed value. The bias has the effect of shifting the activation function and it is traditional to set the bias input value to 1.0. When using ReLU in your network, consider setting the bias to a small value, such as 0.1.

... it can be a good practice to set all elements of [the bias] to a small, positive value, such as 0.1. This makes it very likely that the rectified linear units will be initially active for most inputs in the training set and allow the derivatives to pass through.

— Page 193, *Deep Learning*, 2016.

There are some conflicting reports as to whether this is required, so compare performance to a model with a 1.0 bias input.

Use *He* Weight Initialization

Before training a neural network, the weights of the network must be initialized to small random values. When using ReLU in your network and initializing weights to small random values centered on zero, then by default half of the units in the network will output a zero value.

For example, after uniform initialization of the weights, around 50% of hidden units continuous output values are real zeros

— *Deep Sparse Rectifier Neural Networks*, 2011.

There are many heuristic methods to initialize the weights for a neural network, yet there is no best weight initialization scheme and little relationship beyond general guidelines for mapping weight initialization schemes to the choice of activation function. Prior to the wide adoption of ReLU, Xavier Glorot and Yoshua Bengio proposed an initialization scheme in their 2010 paper titled *Understanding the difficulty of training deep feedforward neural networks* that quickly became the default when using sigmoid and tanh activation functions, generally referred to as *Xavier initialization*. Weights are set at random values sampled uniformly from a range

proportional to the size of the number of nodes in the previous layer (specifically $+/- \frac{1}{\sqrt{n}}$ where n is the number of nodes in the prior layer). Kaiming He, et al. in their 2015 paper titled *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* suggested that Xavier initialization and other schemes were not appropriate for ReLU and extensions.

Glorot and Bengio proposed to adopt a properly scaled uniform distribution for initialization. This is called “Xavier” initialization [...]. Its derivation is based on the assumption that the activations are linear. This assumption is invalid for ReLU

— *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, 2015.

They proposed a small modification of Xavier initialization to make it suitable for use with ReLU, now commonly referred to as *He initialization* (specifically $+/- \frac{2}{\sqrt{n}}$ where n is the number of nodes in the prior layer). In practice, both Gaussian and uniform versions of the scheme can be used.

Scale Input Data

It is good practice to scale input data prior to using a neural network. This may involve standardizing variables to have a zero mean and unit variance or normalizing each value to the scale 0-to-1. Without data scaling on many problems, the weights of the neural network can grow large, making the network unstable and increasing the generalization error. This good practice of scaling inputs applies whether using ReLU for your network or not.

Use Weight Penalty

By design, the output from ReLU is unbounded in the positive domain. This means that in some cases, the output can continue to grow in size. As such, it may be a good idea to use a form of weight regularization, such as an L1 or L2 vector norm.

Another problem could arise due to the unbounded behavior of the activations; one may thus want to use a regularizer to prevent potential numerical problems. Therefore, we use the L1 penalty on the activation values, which also promotes additional sparsity

— *Deep Sparse Rectifier Neural Networks*, 2011.

This can be a good practice to both promote sparse representations (e.g. with L1 regularization) and reduced generalization error of the model.

7.1.6 Extensions and Alternatives to ReLU

The ReLU does have some limitations. Key among the limitations of ReLU is the case where large weight updates can mean that the summed input to the activation function is always negative, regardless of the input to the network. This means that a node with this problem will forever output an activation value of 0.0. This is referred to as a *dying ReLU*.

the gradient is 0 whenever the unit is not active. This could lead to cases where a unit never activates as a gradient-based optimization algorithm will not adjust the weights of a unit that never activates initially. Further, like the vanishing gradients problem, we might expect learning to be slow when training ReL networks with constant 0 gradients.

— *Rectifier Nonlinearities Improve Neural Network Acoustic Models*, 2013.

Some popular extensions to the ReLU relax the nonlinear output of the function to allow small negative values in some way. The Leaky ReLU (LReLU or LReL) modifies the function to allow small negative values when the input is less than zero.

The leaky rectifier allows for a small, non-zero gradient when the unit is saturated and not active

— *Rectifier Nonlinearities Improve Neural Network Acoustic Models*, 2013.

The Exponential Linear Unit, or ELU, is a generalization of the ReLU that uses a parameterized exponential function to transition from the positive to small negative values.

ELUs have negative values which pushes the mean of the activations closer to zero. Mean activations that are closer to zero enable faster learning as they bring the gradient closer to the natural gradient

— *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*, 2016.

The Parametric ReLU, or PReLU, learns parameters that control the shape and leaky-ness of the function.

... we propose a new generalization of ReLU, which we call Parametric Rectified Linear Unit (PReLU). This activation function adaptively learns the parameters of the rectifiers

— *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, 2015.

Maxout is an alternative piecewise linear function that returns the maximum of the inputs, designed to be used in conjunction with the dropout regularization technique.

We define a simple new model called maxout (so named because its output is the max of a set of inputs, and because it is a natural companion to dropout) designed to both facilitate optimization by dropout and improve the accuracy of dropout's fast approximate model averaging technique.

— *Maxout Networks*, 2013.

7.2 ReLU Keras API

The rectified linear activation function can be used directly in Keras. It can be specified as a standalone layer via the `Activation` layer.

```
...
model.append(Activation('relu'))
```

Listing 7.6: Example of ReLU activation layer.

The rectifier activation function can also be specified directly on the layer, such as in the case of a `Dense`, `Conv2D`, and `LSTM` layer via the `activation` argument.

```
...
model.append(Dense(..., activation='relu'))
```

Listing 7.7: Example of ReLU activation on a `Dense` layer.

Keras also supports variations of the rectified linear activation function as standalone layers, such as the parametric ReLU via the `PReLU` layer and the leaky ReLU via the `LeakyReLU` layer.

7.3 ReLU Case Study

In this section, we will demonstrate how to use ReLU to counter the vanishing gradient problem with a MLP on a simple classification problem. This example provides a template for exploring ReLU with your own neural network for classification and regression problems.

7.3.1 Binary Classification Problem

As the basis for our exploration, we will use a very simple two-class or binary classification problem. The scikit-learn class provides the `make_circles()` function that can be used to create a binary classification problem with the prescribed number of samples and statistical noise. Each example has two input variables that define the x and y coordinates of the point on a two-dimensional plane. The points are arranged in two concentric circles (they have the same center) for the two classes. The number of points in the dataset is specified by a parameter, half of which will be drawn from each circle. Gaussian noise can be added when sampling the points via the `noise` argument that defines the standard deviation of the noise, where 0.0 indicates no noise or points drawn exactly from the circles. The seed for the pseudorandom number generator can be specified via the `random_state` argument that allows the exact same points to be sampled each time the function is called. The example below generates 1,000 examples from the two circles with noise and a value of 1 to seed the pseudorandom number generator.

```
# generate circles
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
```

Listing 7.8: Example of generating samples from the two circles problem.

We can create a graph of the dataset, plotting the x and y coordinates of the input variables (X) and coloring each point by the class value (0 or 1). The complete example is listed below.

```
# scatter plot of the circles dataset with points colored by class
from sklearn.datasets import make_circles
from numpy import where
```

```

from matplotlib import pyplot
# generate circles
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# select indices of points with each class label
for i in range(2):
    samples_ix = where(y == i)
    pyplot.scatter(X[samples_ix, 0], X[samples_ix, 1], label=str(i))
pyplot.legend()
pyplot.show()

```

Listing 7.9: Example of plotting samples from the two circles problem.

Running the example creates a plot showing the 1,000 generated data points with the class value of each point used to color each point. We can see points for class 0 are blue and represent the outer circle, and points for class 1 are orange and represent the inner circle. The statistical noise of the generated samples means that there is some overlap of points between the two circles, adding some ambiguity to the problem, making it non-trivial. This is desirable as a neural network may choose one of among many possible solutions to classify the points between the two circles and always make some errors.

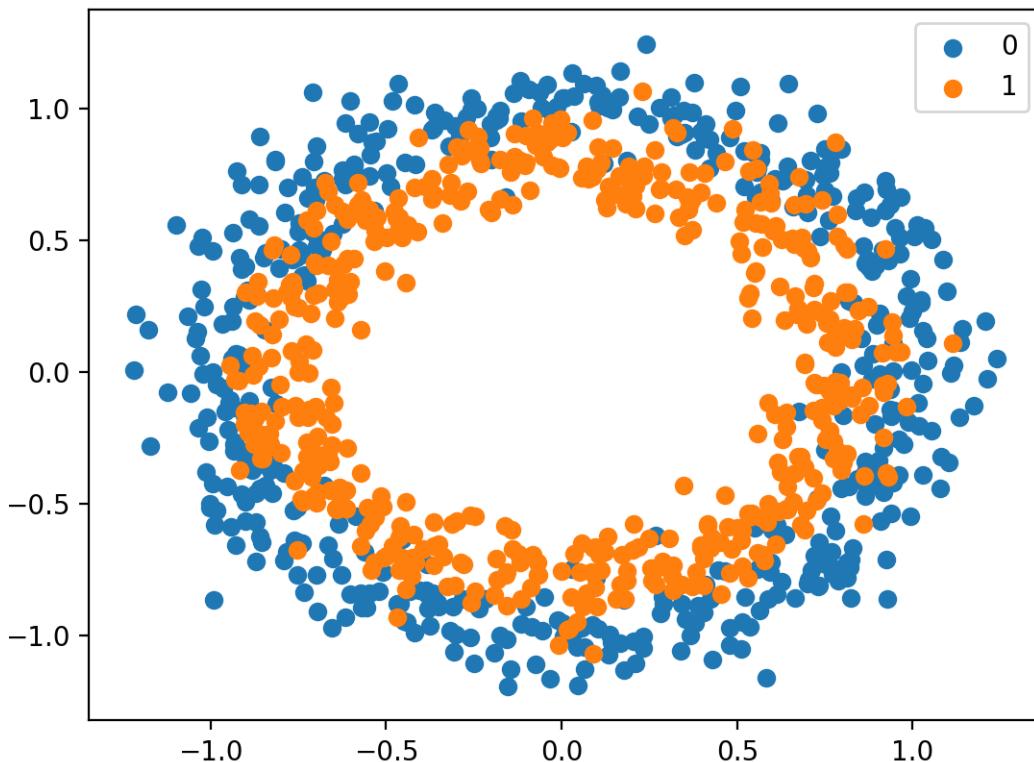


Figure 7.2: Scatter Plot of Circles Dataset With Points Colored By Class Value.

Now that we have defined a problem as the basis for our exploration, we can look at developing a model to address it.

7.3.2 Multilayer Perceptron Model

We can develop a Multilayer Perceptron model to address the two circles problem. This will be a simple feedforward neural network model, designed as we were taught in the late 1990s and early 2000s. First, we will generate 1,000 data points from the two circles problem and rescale the inputs to the range $[-1, 1]$. The data is almost already in this range, but we will make sure. Normally, we would prepare the data scaling using a training dataset and apply it to a test dataset. To keep things simple in this tutorial, we will scale all of the data together before splitting it into train and test sets.

```
# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# scale input data to [-1,1]
scaler = MinMaxScaler(feature_range=(-1, 1))
X = scaler.fit_transform(X)
```

Listing 7.10: Example of scaling the data ready for modeling.

Next, we will split the data into train and test sets. Half of the data will be used for training and the remaining 500 examples will be used as the test set. In this tutorial, the test set will also serve as the validation dataset so we can get an idea of how the model performs on the holdout set during training.

```
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 7.11: Example of splitting the data into train and test sets.

Next, we will define the model. The model will have an input layer with two inputs, for the two variables in the dataset, one hidden layer with five nodes, and an output layer with one node used to predict the class probability. The hidden layer will use the hyperbolic tangent activation function (`tanh`) and the output layer will use the logistic activation function (`sigmoid`) to predict class 0 or class 1 or something in between. Using the hyperbolic tangent activation function in hidden layers was the best practice in the 1990s and 2000s, performing generally better than the logistic function when used in the hidden layer. It was also good practice to initialize the network weights to small random values from a uniform distribution. Here, we will initialize weights randomly from the range $[0.0, 1.0]$.

```
# define model
model = Sequential()
init = RandomUniform(minval=0, maxval=1)
model.add(Dense(5, input_dim=2, activation='tanh', kernel_initializer=init))
model.add(Dense(1, activation='sigmoid', kernel_initializer=init))
```

Listing 7.12: Example of defining the MLP model.

The model uses the binary cross-entropy loss function and is optimized using stochastic gradient descent with a learning rate of 0.01 and a large momentum of 0.9.

```
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 7.13: Example of compiling the MLP model.

The model is trained for 500 training epochs and the test dataset is evaluated at the end of each epoch along with the training dataset.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=500, verbose=0)
```

Listing 7.14: Example of fitting the MLP model.

After the model is fit, it is evaluated on both the train and test dataset and the accuracy scores are displayed.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 7.15: Example of evaluating the MLP model.

Finally, the accuracy of the model during each step of training is graphed as a line plot, showing the dynamics of the model as it learned the problem.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 7.16: Example of plotting learning curves for the MLP model.

Tying all of this together, the complete example is listed below.

```
# mlp with tanh for the two circles classification problem
from sklearn.datasets import make_circles
from sklearn.preprocessing import MinMaxScaler
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.initializers import RandomUniform
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# scale input data to [-1,1]
scaler = MinMaxScaler(feature_range=(-1, 1))
X = scaler.fit_transform(X)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
```

```

init = RandomUniform(minval=0, maxval=1)
model.add(Dense(5, input_dim=2, activation='tanh', kernel_initializer=init))
model.add(Dense(1, activation='sigmoid', kernel_initializer=init))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=500, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 7.17: Example of evaluating a tanh-based MLP for the two circles problem.

Running the example fits the model in just a few seconds. The model performance on the train and test sets is calculated and displayed.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that in this case, the model learned the problem well, achieving an accuracy of about 81.6% on both the train and test datasets.

Train: 0.800, Test: 0.820

Listing 7.18: Example output from evaluating an old-style MLP for the two circles problem.

A line plot of model loss and accuracy on the train and test sets is created, showing the change in performance over all 500 training epochs. The plots suggest, for this run, that the performance begins to slow around epoch 200 at about 80% accuracy for both the train and test sets.

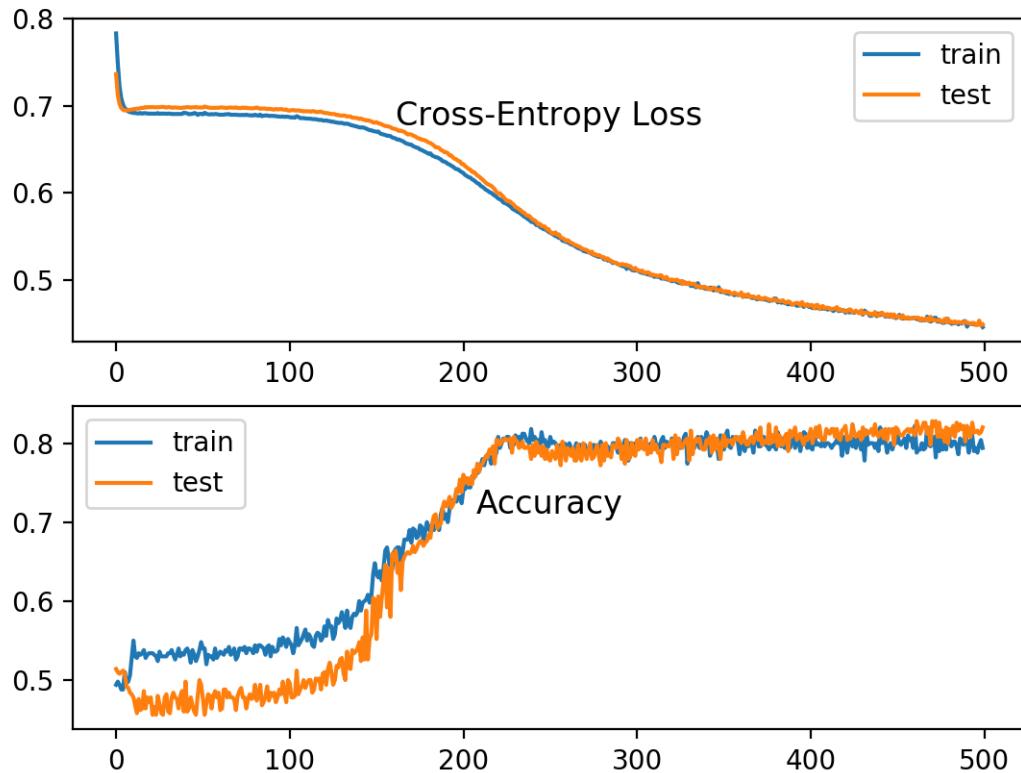


Figure 7.3: Line Plot of Train and Test Set Accuracy Over Training Epochs for MLP in the Two Circles Problem.

Now that we have seen how to develop a classical MLP using the tanh activation function for the two circles problem, we can look at modifying the model to have many more hidden layers.

7.3.3 Deeper MLP Model

Traditionally, developing deep Multilayer Perceptron models was challenging. Deep models using the hyperbolic tangent activation function do not train easily, and much of this poor performance is blamed on the vanishing gradient problem. We can attempt to investigate this using the MLP model developed in the previous section. The number of hidden layers can be increased from 1 to 5; for example:

```
# define model
init = RandomUniform(minval=0, maxval=1)
model = Sequential()
model.add(Dense(5, input_dim=2, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(1, activation='sigmoid', kernel_initializer=init))
```

Listing 7.19: Example of defining a much deeper tanh-based MLP.

We can then re-run the example and review the results. The complete example of the deeper MLP is listed below.

```
# deeper mlp with tanh for the two circles classification problem
from sklearn.datasets import make_circles
from sklearn.preprocessing import MinMaxScaler
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.initializers import RandomUniform
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
scaler = MinMaxScaler(feature_range=(-1, 1))
X = scaler.fit_transform(X)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
init = RandomUniform(minval=0, maxval=1)
model = Sequential()
model.add(Dense(5, input_dim=2, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(5, activation='tanh', kernel_initializer=init))
model.add(Dense(1, activation='sigmoid', kernel_initializer=init))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=500, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 7.20: Example of evaluating a deeper tanh-based MLP for the two circles problem.

Running the example first prints the performance of the fit model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that performance is quite poor on both the train and test sets achieving around 50% accuracy. This suggests that the model as configured could not learn the problem nor generalize a solution.

```
Train: 0.588, Test: 0.552
```

Listing 7.21: Example output from evaluating a deeper tanh-based MLP for the two circles problem.

The line plots of model accuracy on the train and test sets during training tell a similar story. We can see that performance is bad and actually gets worse as training progresses.

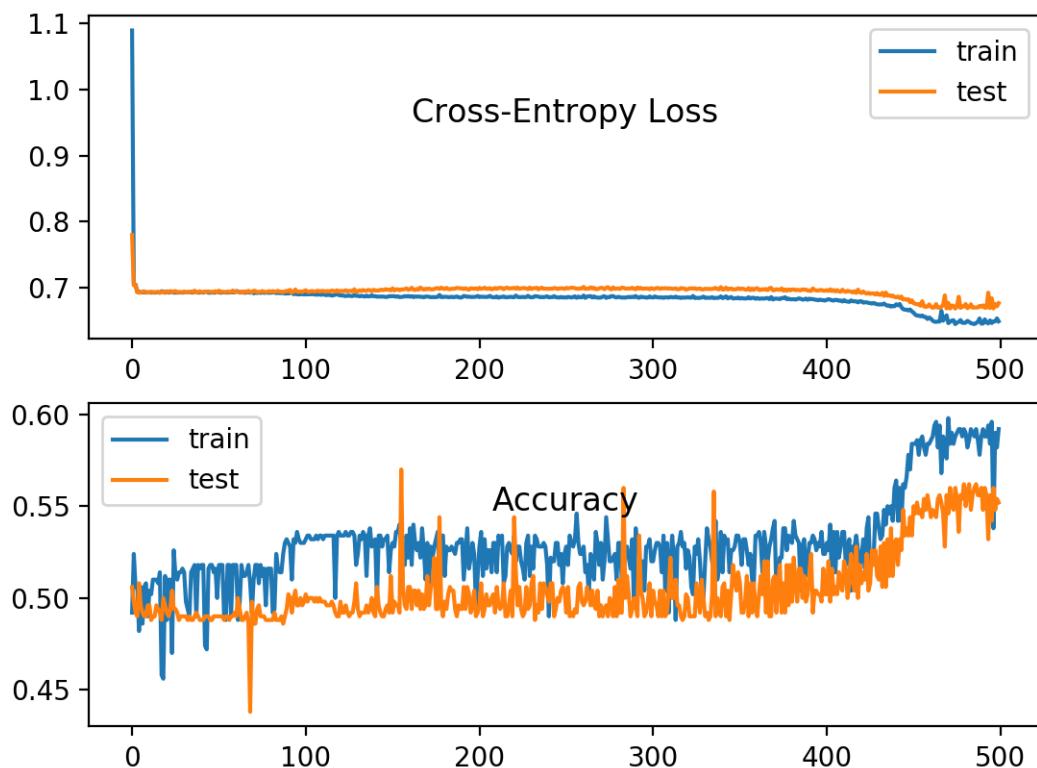


Figure 7.4: Line Plot of Train and Test Set Accuracy of Over Training Epochs for Deep MLP in the Two Circles Problem.

7.3.4 Deeper MLP Model with ReLU

The rectified linear activation function has supplanted the hyperbolic tangent activation function as the new preferred default when developing Multilayer Perceptron networks, as well as other

network types like CNNs. This is because the activation function looks and acts like a linear function, making it easier to train and less likely to saturate, but is, in fact, a nonlinear function, forcing negative inputs to the value 0. It is claimed as one possible approach to addressing the vanishing gradients problem when training deeper models. When using the rectified linear activation function (or ReLU for short), it is good practice to use the He weight initialization scheme. We can define the MLP with five hidden layers using ReLU and He initialization, listed below.

```
# define model
model = Sequential()
model.add(Dense(5, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid'))
```

Listing 7.22: Example of defining a much deeper ReLU-based MLP.

Tying this together, the complete code example is listed below.

```
# deeper mlp with relu for the two circles classification problem (5 hidden layers)
from sklearn.datasets import make_circles
from sklearn.preprocessing import MinMaxScaler
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
scaler = MinMaxScaler(feature_range=(-1, 1))
X = scaler.fit_transform(X)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(5, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid'))
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=500, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
```

```
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 7.23: Example of evaluating a deeper ReLU-based MLP for the two circles problem.

Running the example prints the performance of the model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that this small change has allowed the model to learn the problem, achieving about 84% accuracy on both datasets, outperforming the single layer model using the tanh activation function.

```
Train: 0.842, Test: 0.846
```

Listing 7.24: Example output from evaluating a deeper ReLU-based MLP for the two circles problem.

A line plot of model accuracy on the train and test sets over training epochs is also created. The plot shows quite different dynamics to what we have seen so far. The model appears to rapidly learn the problem, converging on a solution in about 100 epochs.

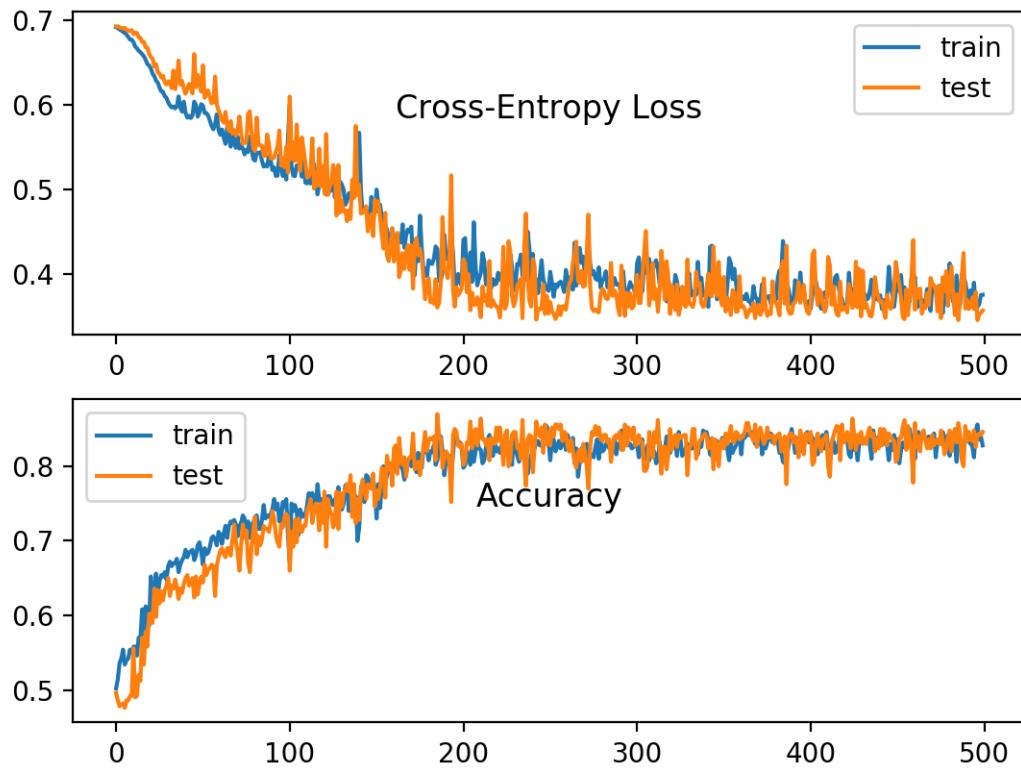


Figure 7.5: Line Plot of Train and Test Set Accuracy of Over Training Epochs for Deep MLP with ReLU in the Two Circles Problem.

Use of the ReLU activation function has allowed us to fit a much deeper model for this simple problem, but this capability does not extend infinitely. For example, increasing the number of layers results in slower learning to a point at about 20 layers where the model is no longer capable of learning the problem, at least with the chosen configuration. For example, below is a line plot of train and test accuracy of the same model with 15 hidden layers that shows that it is still capable of learning the problem (at least some of the time).

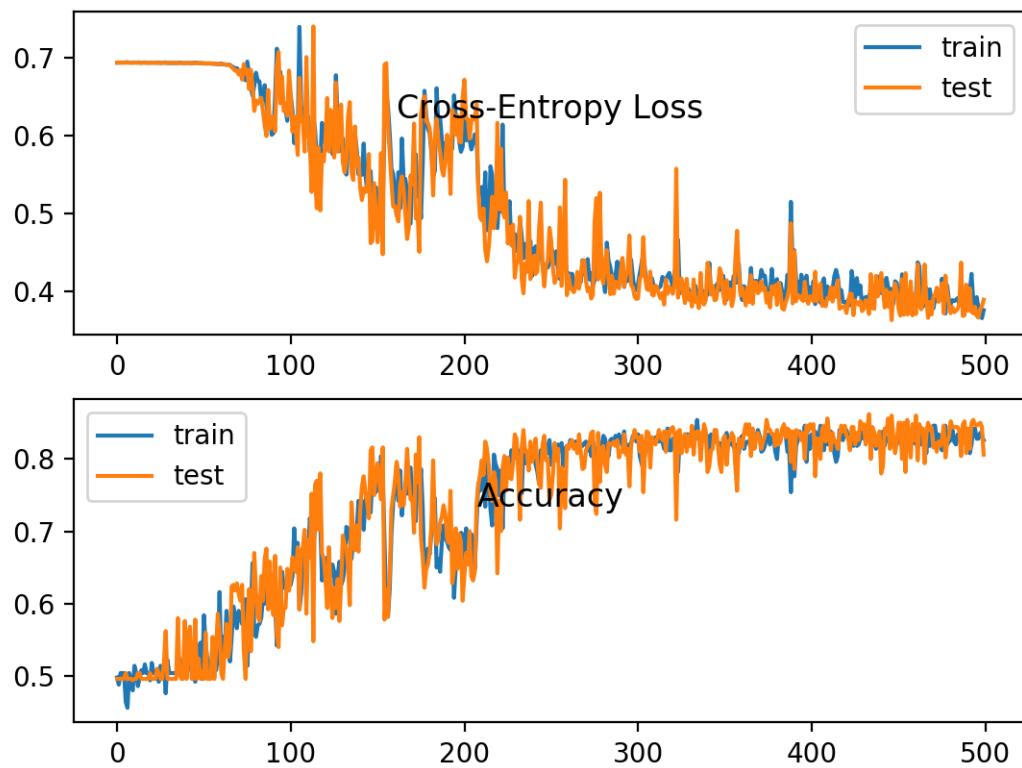


Figure 7.6: Line Plot of Train and Test Set Accuracy of Over Training Epochs for Deep MLP with ReLU with 15 Hidden Layers.

Below is a line plot of train and test accuracy over epochs with the same model with 20 layers, showing that the configuration is no longer capable of learning the problem (at least some of the time).

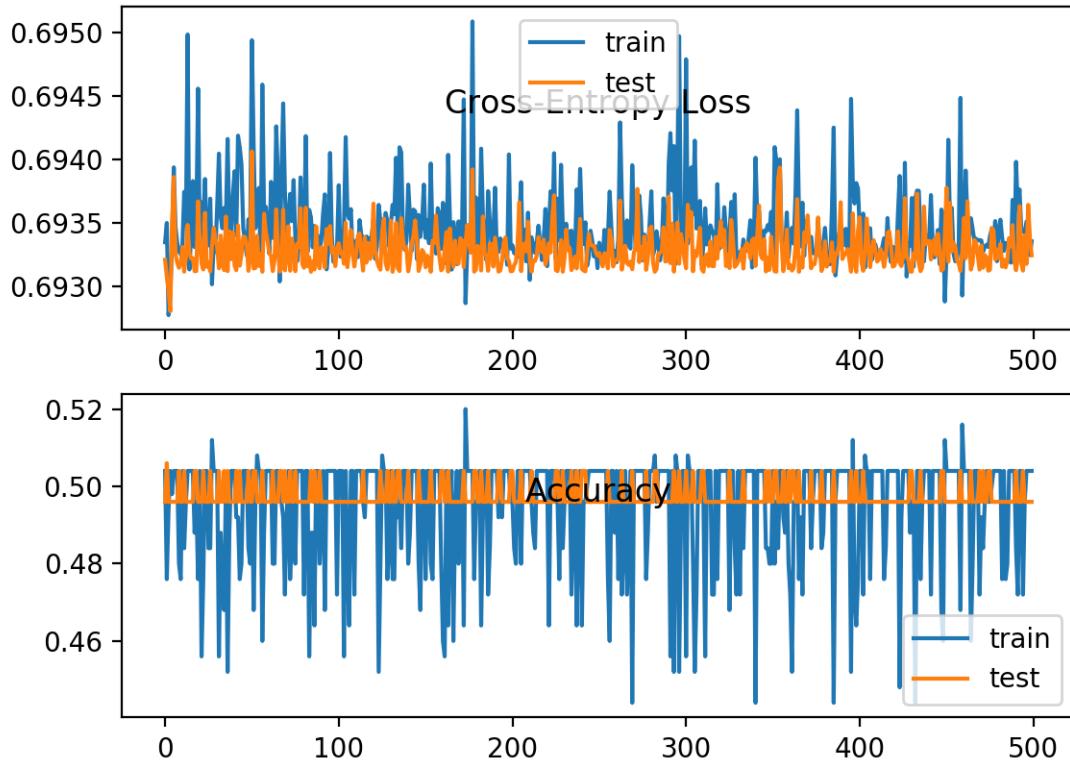


Figure 7.7: Line Plot of Train and Test Set Accuracy of Over Training Epochs for Deep MLP with ReLU with 20 Hidden Layers.

Although use of the ReLU worked, we cannot be confident that use of the tanh function failed because of vanishing gradients and ReLU succeed because it overcame this problem.

7.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Weight Initialization.** Update the deep MLP with tanh activation to use Xavier uniform weight initialization and report the results.
- **Learning Algorithm.** Update the deep MLP with tanh activation to use an adaptive learning algorithm such as Adam and report the results.
- **Weight Changes.** Update the tanh and relu examples to record and plot the L1 vector norm of model weights each epoch as a proxy for how much each layer is changed during training and compare results.
- **Study Model Depth.** Create an experiment using the MLP with tanh activation and report the performance of models as the number of hidden layers is increased from 1 to 10.

- **Increase Breadth.** Increase the number of nodes in the hidden layers of the MLP with tanh activation from 5 to 25 and report performance as the number of layers are increased from 1 to 10.

If you explore any of these extensions, I'd love to know.

7.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

7.5.1 Books

- Section 6.3.1: Rectified Linear Units and Their Generalizations, *Deep Learning*, 2016.
<https://amzn.to/2QHVVmW>

7.5.2 Papers

- *What is the best multi-stage architecture for object recognition?*, 2009
<https://ieeexplore.ieee.org/document/5459469>
- *Rectified Linear Units Improve Restricted Boltzmann Machines*, 2010.
<https://dl.acm.org/citation.cfm?id=3104425>
- *Deep Sparse Rectifier Neural Networks*, 2011.
<http://proceedings.mlr.press/v15/glorot11a>
- *Rectifier Nonlinearities Improve Neural Network Acoustic Models*, 2013.
http://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf
- *Understanding the difficulty of training deep feedforward neural networks*, 2010.
<http://proceedings.mlr.press/v9/glorot10a.html>
- *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, 2015.
<https://arxiv.org/abs/1502.01852>
- *Maxout Networks*, 2013.
<https://arxiv.org/abs/1302.4389>
- *Random Walk Initialization for Training Very Deep Feedforward Networks*, 2014.
<https://arxiv.org/abs/1412.6558>
- *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies*, 2001.
<http://www.iro.umontreal.ca/~lisa/publications2/index.php/publications/show/55>

7.5.3 APIs

- Keras Activation Functions API.
<https://keras.io/activations/>
- Keras Advanced Activation Functions API.
<https://keras.io/layers/advanced-activations/>
- max API.
<https://docs.python.org/3/library/functions.html#max>
- RandomUniform Keras Weight Initialization API.
<https://keras.io/initializers/#randomuniform>
- SGD Keras Optimizer API.
<https://keras.io/optimizers/#sgd>

7.5.4 Articles

- Neural Network FAQ.
<ftp://ftp.sas.com/pub/neural/FAQ.html>
- Activation function, Wikipedia.
https://en.wikipedia.org/wiki/Activation_function
- Vanishing gradient problem, Wikipedia.
https://en.wikipedia.org/wiki/Vanishing_gradient_problem
- Rectifier (neural networks), Wikipedia.
[https://en.wikipedia.org/wiki/Rectifier_\(neural_networks\)](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))
- Piecewise Linear Function, Wikipedia.
https://en.wikipedia.org/wiki/Piecewise_linear_function

7.6 Summary

In this tutorial, you discovered the rectified linear activation function for deep learning neural networks. Specifically, you learned:

- The sigmoid and hyperbolic tangent activation functions cannot be used in networks with many layers due to the vanishing gradient problem.
- The rectified linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better.
- The rectified linear activation is the default activation when developing Multilayer Perceptron and convolutional neural networks.

7.6.1 Next

In the next tutorial, you will discover how to fix the exploding gradients problem through the use of gradient clipping.

Chapter 8

Fix Exploding Gradients with Gradient Clipping

Training a neural network can become unstable given the choice of error function, learning rate, or even the scale of the target variable. Large updates to weights during training can cause a numerical overflow or underflow often referred to as *exploding gradients*. The problem of exploding gradients is more common with recurrent neural networks, such as LSTMs given the accumulation of gradients unrolled over hundreds of input time steps. A common and relatively easy solution to the exploding gradients problem is to change the derivative of the error before propagating it backward through the network and using it to update the weights. Two approaches include rescaling the gradients given a chosen vector norm and clipping gradient values that exceed a preferred range. Together, these methods are referred to as *gradient clipping*. In this tutorial, you will discover the exploding gradient problem and how to improve neural network training stability using gradient clipping. After completing this tutorial, you will know:

- Training neural networks can become unstable, leading to a numerical overflow or underflow referred to as exploding gradients.
- The training process can be made stable by changing the error gradients either by scaling the vector norm or clipping gradient values to a range.
- How to update an MLP model for a regression predictive modeling problem with exploding gradients to have a stable training process using gradient clipping methods.

Let's get started.

8.1 Exploding Gradients and Clipping

Neural networks are trained using the stochastic gradient descent optimization algorithm. This requires first the estimation of the loss on one or more training examples, then the calculation of the derivative of the loss, which is propagated backward through the network in order to update the weights. Weights are updated using a fraction of the back propagated error controlled by the learning rate. It is possible for the updates to the weights to be so large that the weights either overflow or underflow their numerical precision. In practice, the weights can take on the value of an `NaN` (not a number) or `Inf` (infinity) when they overflow or underflow and for

practical purposes the network will be useless from that point forward, forever predicting NaN values as signals flow through the invalid weights.

The difficulty that arises is that when the parameter gradient is very large, a gradient descent parameter update could throw the parameters very far, into a region where the objective function is larger, undoing much of the work that had been done to reach the current solution.

— Page 413, *Deep Learning*, 2016.

The underflow or overflow of weights is generally referred to as an instability of the network training process and is known by the name *exploding gradients* as the unstable training process causes the network to fail to train in such a way that the model is essentially useless. In a given neural network, such as a Convolutional Neural Network or Multilayer Perceptron, this can happen due to a poor choice of configuration. Some examples include:

- Poor choice of learning rate that results in large weight updates.
- Poor choice of data preparation, allowing large differences in the target variable.
- Poor choice of loss function, allowing the calculation of large error values.

Exploding gradients is also a problem in recurrent neural networks such as the Long Short-Term Memory network given the accumulation of error gradients in the unrolled recurrent structure. Exploding gradients can be avoided in general by careful configuration of the network model, such as choice of small learning rate, scaled target variables, and a standard loss function. Nevertheless, exploding gradients may still be an issue with recurrent networks with a large number of input time steps.

One difficulty when training LSTM with the full gradient is that the derivatives sometimes become excessively large, leading to numerical problems. To prevent this, [we] clipped the derivative of the loss with respect to the network inputs to the LSTM layers (before the sigmoid and tanh functions are applied) to lie within a predefined range.

— *Generating Sequences With Recurrent Neural Networks*, 2013.

A common solution to exploding gradients is to change the error derivative before propagating it backward through the network and using it to update the weights. By rescaling the error derivative, the updates to the weights will also be rescaled, dramatically decreasing the likelihood of an overflow or underflow. There are two main methods for updating the error derivative; they are:

- Gradient Scaling.
- Gradient Clipping.

Gradient scaling involves normalizing the error gradient vector such that vector norm (magnitude) equals a defined value, such as 1.0.

... one simple mechanism to deal with a sudden increase in the norm of the gradients is to rescale them whenever they go over a threshold

— *On the difficulty of training Recurrent Neural Networks*, 2013.

Gradient clipping involves forcing the gradient values (element-wise) to a specific minimum or maximum value if the gradient exceeded an expected range. Together, these methods are often simply referred to as *gradient clipping*.

When the traditional gradient descent algorithm proposes to make a very large step, the gradient clipping heuristic intervenes to reduce the step size to be small enough that it is less likely to go outside the region where the gradient indicates the direction of approximately steepest descent.

— Page 289, *Deep Learning*, 2016.

It is a method that only addresses the numerical stability of training deep neural network models and does not offer any general improvement in performance. The value for the gradient vector norm or preferred range can be configured by trial and error, by using common values used in the literature or by first observing common vector norms or ranges via experimentation and then choosing a sensible value.

In our experiments we have noticed that for a given task and model size, training is not very sensitive to this [gradient norm] hyperparameter and the algorithm behaves well even for rather small thresholds.

— *On the difficulty of training Recurrent Neural Networks*, 2013.

It is common to use the same gradient clipping configuration for all layers in the network. Nevertheless, there are examples where a larger range of error gradients are permitted in the output layer compared to hidden layers.

The output derivatives [...] were clipped in the range [-100, 100], and the LSTM derivatives were clipped in the range [-10, 10]. Clipping the output gradients proved vital for numerical stability; even so, the networks sometimes had numerical problems late on in training, after they had started overfitting on the training data.

— *Generating Sequences With Recurrent Neural Networks*, 2013.

8.2 Gradient Clipping Keras API

Keras supports gradient clipping on each optimization algorithm, with the same scheme applied to all layers in the model. Gradient clipping can be used with an optimization algorithm, such as stochastic gradient descent, via including an additional argument when configuring the optimization algorithm. Two types of gradient clipping can be used: gradient norm scaling and gradient value clipping.

8.2.1 Gradient Norm Scaling

Gradient norm scaling involves changing the derivatives of the loss function to have a given vector norm when the L2 vector norm (sum of the squared values) of the gradient vector exceeds a threshold value. For example, we could specify a norm of 1.0, meaning that if the vector norm for a gradient exceeds 1.0, then the values in the vector will be rescaled so that the norm of the vector equals 1.0. This can be used in Keras by specifying the `clipnorm` argument on the optimizer; for example:

```
...
# configure sgd with gradient norm clipping
opt = SGD(lr=0.01, momentum=0.9, clipnorm=1.0)
```

Listing 8.1: Example of gradient norm scaling in Keras.

8.2.2 Gradient Value Clipping

Gradient value clipping involves clipping the derivatives of the loss function to have a given value if a gradient value is less than a negative threshold or more than the positive threshold. For example, we could specify a norm of 0.5, meaning that if a gradient value was less than -0.5, it is set to -0.5 and if it is more than 0.5, then it will be set to 0.5. This can be used in Keras by specifying the `clipvalue` argument on the optimizer, for example:

```
...
# configure sgd with gradient value clipping
opt = SGD(lr=0.01, momentum=0.9, clipvalue=0.5)
```

Listing 8.2: Example of gradient value clipping in Keras.

8.3 Gradient Clipping Case Study

In this section, we will demonstrate how to use gradient clipping to counter the exploding gradients problem with a MLP on a simple classification problem. This example provides a template for exploring gradient clipping with your own neural network for classification and regression problems.

8.3.1 Regression Predictive Modeling Problem

A regression predictive modeling problem involves predicting a real-valued quantity. We can use a standard regression problem generator provided by the scikit-learn library in the `make_regression()` function. This function will generate examples from a simple regression problem with a given number of input variables, statistical noise, and other properties. We will use this function to define a problem that has 20 input features; 10 of the features will be meaningful and 10 will not be relevant. A total of 1,000 examples will be randomly generated. The pseudorandom number generator will be fixed to ensure that we get the same 1,000 examples each time the code is run.

```
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
```

Listing 8.3: Example of generating samples for the regression problem.

Each input variable has a Gaussian distribution, as does the target variable. We can create plots of the target variable showing both the distribution and spread. The complete example is listed below.

```
# regression predictive modeling problem
from sklearn.datasets import make_regression
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# histogram of target variable
pyplot.subplot(121)
pyplot.hist(y)
# boxplot of target variable
pyplot.subplot(122)
pyplot.boxplot(y)
pyplot.show()
```

Listing 8.4: Example of generating samples and plotting their distribution for the regression problem.

Running the example creates a figure with two plots showing a histogram and a box and whisker plot of the target variable. The histogram shows the Gaussian distribution of the target variable. The box and whisker plot shows that the range of samples varies between about -400 to 400 with a mean of about 0.0.

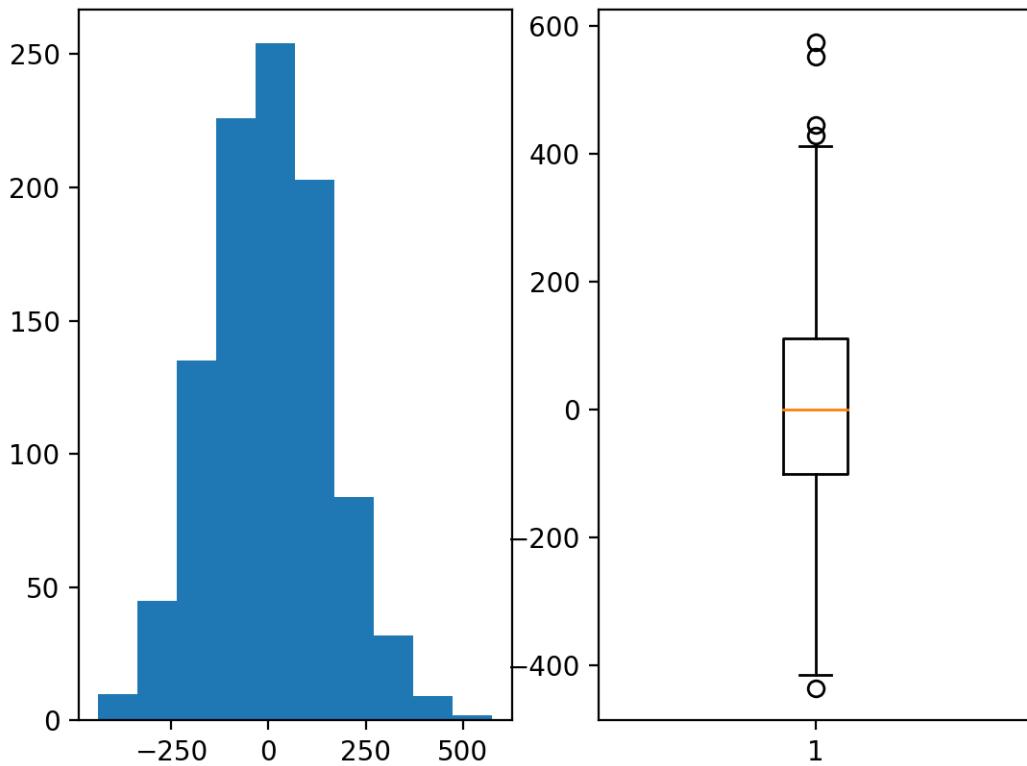


Figure 8.1: Histogram and Box and Whisker Plot of the Target Variable for the Regression Problem.

8.3.2 Multilayer Perceptron With Exploding Gradients

We can develop a Multilayer Perceptron (MLP) model for the regression problem. A model will be demonstrated on the raw data, without any scaling of the input or output variables. This is a good example to demonstrate exploding gradients as a model trained to predict the unscaled target variable will result in error gradients with values in the hundreds or even thousands, depending on the batch size used during training. Such large gradient values are likely to lead to unstable learning or an overflow of the weight values. The first step is to split the data into train and test sets so that we can fit and evaluate a model. We will generate 1,000 examples from the domain and split the dataset in half, using 500 examples for train and test sets.

```
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 8.5: Example of preparing the dataset ready for modeling.

Next, we can define an MLP model. The model will expect 20 inputs for the 20 input variables in the problem. A single hidden layer will be used with 25 nodes and a rectified linear

activation function. The output layer has one node for the single target variable and a linear activation function to predict real values directly.

```
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
```

Listing 8.6: Example of defining the MLP model.

The mean squared error loss function will be used to optimize the model and the stochastic gradient descent optimization algorithm will be used with the sensible default configuration of a learning rate of 0.01 and a momentum of 0.9.

```
# compile model
model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.01, momentum=0.9))
```

Listing 8.7: Example of compiling the MLP model.

The model will be fit for 100 training epochs and the test set will be used as a validation set, evaluated at the end of each training epoch.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
```

Listing 8.8: Example of fitting the MLP model.

The mean squared error is calculated on the train and test datasets at the end of training to get an idea of how well the model learned the problem.

```
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
```

Listing 8.9: Example of evaluating the MLP model.

Finally, learning curves of mean squared error on the train and test sets at the end of each training epoch are graphed using line plots, providing learning curves to get an idea of the dynamics of the model while learning the problem.

```
# plot loss during training
pyplot.title('Mean Squared Error')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 8.10: Example of plotting the learning curves for the MLP model.

Tying these elements together, the complete example is listed below.

```
# mlp with unscaled data for the regression problem
from sklearn.datasets import make_regression
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
```

```

# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mean_squared_error', optimizer=SGD(lr=0.01, momentum=0.9))
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.title('Mean Squared Error')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 8.11: Example of fitting and evaluating an MLP with exploding gradients on the regression problem.

Running the example fits the model and calculates the mean squared error on the train and test sets. In this case, the model is unable to learn the problem, resulting in predictions of NaN values. The model weights exploded during training given the very large errors and in turn error gradients calculated for weight updates.

Train: nan, Test: nan

Listing 8.12: Example output from fitting and evaluating an MLP with exploding gradients on the regression problem.

This demonstrates that some intervention is required with regard to the target variable for the model to learn this problem. A line plot of training history is created but does not show anything as the model almost immediately results in a NaN mean squared error. A traditional solution would be to rescale the target variable using either standardization or normalization, and this approach is recommended for MLPs. Nevertheless, an alternative that we will investigate in this case will be the use of gradient clipping.

8.3.3 MLP With Gradient Norm Scaling

We can update the training of the model in the previous section to add gradient norm scaling. This can be implemented by setting the `clipnorm` argument on the optimizer. For example, the gradients can be rescaled to have a vector norm (magnitude or length) of 1.0, as follows:

```

# compile model
opt = SGD(lr=0.01, momentum=0.9, clipnorm=1.0)
model.compile(loss='mean_squared_error', optimizer=opt)

```

Listing 8.13: Example of updating the optimizer to use gradient norm scaling.

The complete example with this change is listed below.

```
# mlp with unscaled data for the regression problem with gradient norm scaling
from sklearn.datasets import make_regression
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
# compile model
opt = SGD(lr=0.01, momentum=0.9, clipnorm=1.0)
model.compile(loss='mean_squared_error', optimizer=opt)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.title('Mean Squared Error')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 8.14: Example of fitting and evaluating an MLP with gradient norm scaling on the regression problem.

Running the example fits the model and evaluates it on the train and test sets, printing the mean squared error.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that scaling the gradient with a vector norm of 1.0 has resulted in a stable model capable of learning the problem and converging on a solution.

Train: 5.082, Test: 27.433

Listing 8.15: Example output from fitting and evaluating an MLP with gradient norm scaling on the regression problem.

A line plot is also created showing the means squared error loss on the train and test datasets over training epochs. The plot shows how loss dropped from large values above 20,000 down to small values below 100 rapidly over 20 epochs.

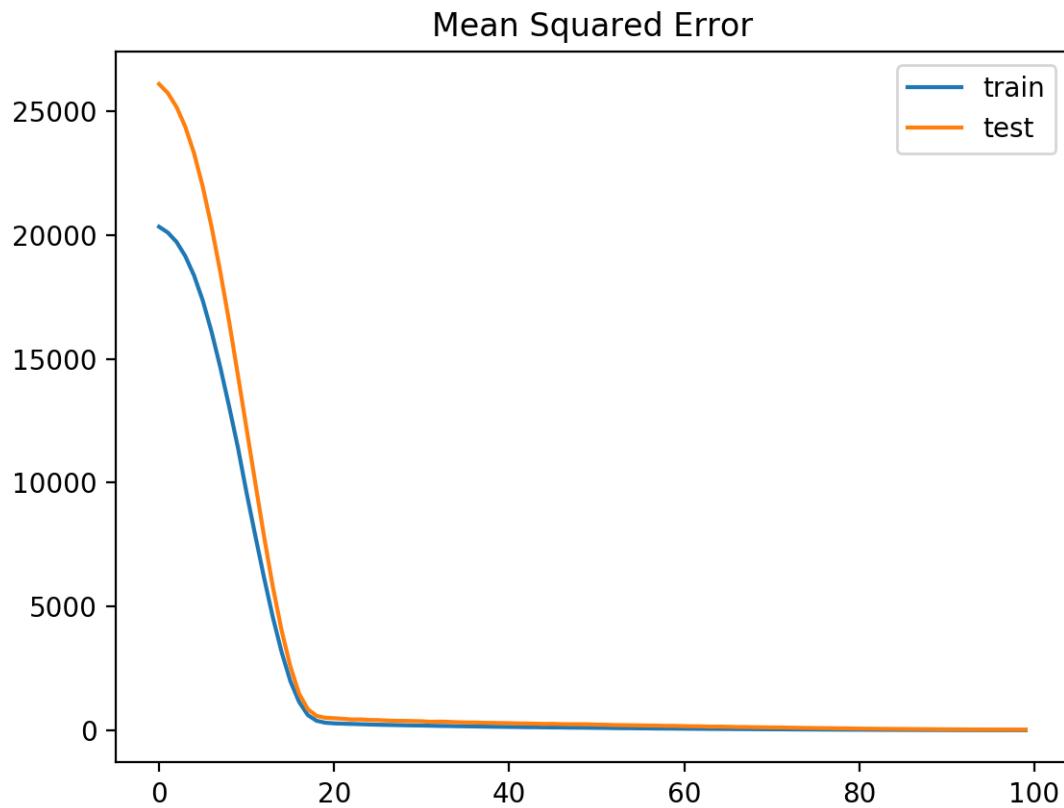


Figure 8.2: Line Plot of Mean Squared Error Loss for the Train (blue) and Test (orange) Datasets Over Training Epochs With Gradient Norm Scaling.

There is nothing special about the vector norm value of 1.0, and other values could be evaluated and the performance of the resulting model compared.

8.3.4 MLP With Gradient Value Clipping

Another solution to the exploding gradient problem is to clip the gradient if it becomes too large or too small. We can update the training of the MLP to use gradient clipping by adding the `clipvalue` argument to the optimization algorithm configuration. For example, the code below clips the gradient to the range [-5 to 5].

```
# compile model
opt = SGD(lr=0.01, momentum=0.9, clipvalue=5.0)
model.compile(loss='mean_squared_error', optimizer=opt)
```

Listing 8.16: Example of updating the optimizer to use gradient value clipping.

The complete example of training the MLP with gradient clipping is listed below.

```
# mlp with unscaled data for the regression problem with gradient clipping
from sklearn.datasets import make_regression
from keras.layers import Dense
from keras.models import Sequential
```

```

from keras.optimizers import SGD
from matplotlib import pyplot
# generate regression dataset
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=20, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
# compile model
opt = SGD(lr=0.01, momentum=0.9, clipvalue=5.0)
model.compile(loss='mean_squared_error', optimizer=opt)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
train_mse = model.evaluate(trainX, trainy, verbose=0)
test_mse = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_mse, test_mse))
# plot loss during training
pyplot.title('Mean Squared Error')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 8.17: Example of fitting and evaluating an MLP with gradient value clipping on the regression problem.

Running this example fits the model and evaluates it on the train and test sets, printing the mean squared error.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that in this case, the model is able to learn the problem without exploding gradients achieving an MSE of below 10 on both the train and test sets.

Train: 9.487, Test: 9.985

Listing 8.18: Example output from fitting and evaluating an MLP with gradient value clipping on the regression problem.

A line plot is also created showing the means squared error loss on the train and test datasets over training epochs. The plot shows that the model learns the problem fast, achieving a sub-100 MSE loss within just a few training epochs.

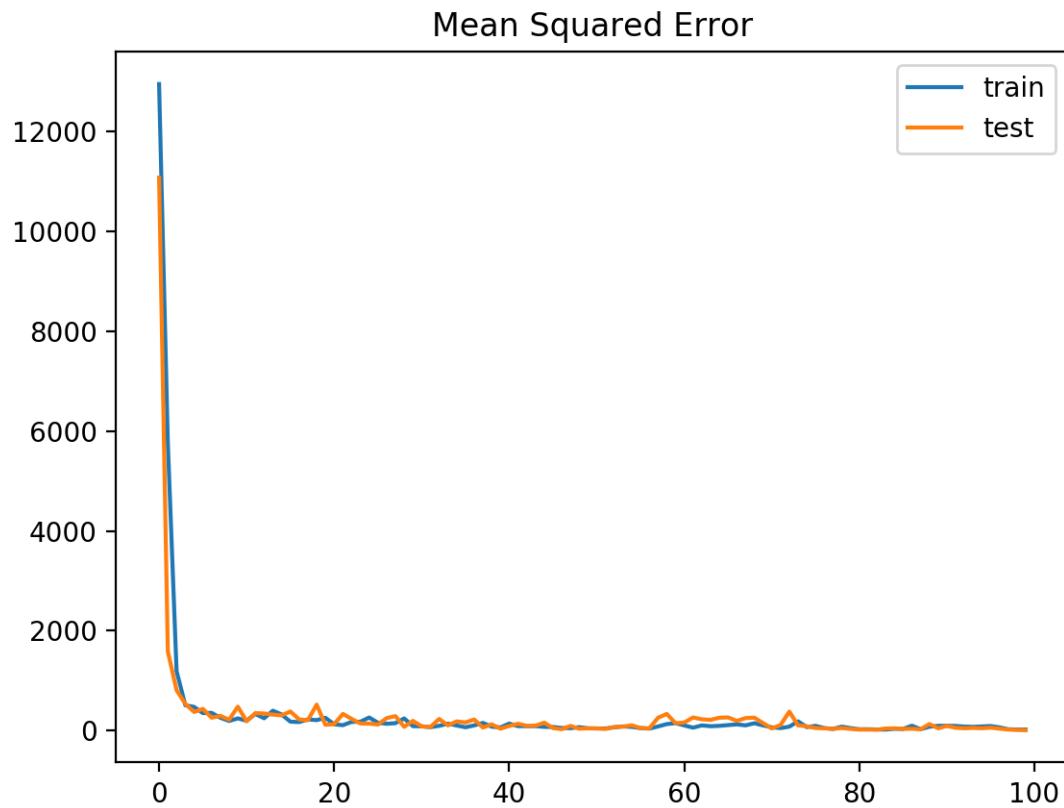


Figure 8.3: Line Plot of Mean Squared Error Loss for the Train (blue) and Test (orange) Datasets Over Training Epochs With Gradient Value Clipping.

A clipped range of $[-5, 5]$ was chosen arbitrarily; you can experiment with different sized ranges and compare performance of the speed of learning and final model performance.

8.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Vector Norm Values.** Update the example to evaluate different gradient vector norm values and compare performance.
- **Vector Clip Values.** Update the example to evaluate different gradient value ranges and compare performance.
- **Vector Norm and Clip.** Update the example to use a combination of vector norm scaling and vector value clipping on the same training run and compare performance.

If you explore any of these extensions, I'd love to know.

8.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

8.5.1 Books

- Section 8.2.4: Cliffs and Exploding Gradients, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- Section 10.11.1: Clipping Gradients, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>

8.5.2 Papers

- *On the difficulty of training Recurrent Neural Networks*, 2013.
<https://arxiv.org/abs/1211.5063>
- *Generating Sequences With Recurrent Neural Networks*, 2013.
<https://arxiv.org/abs/1308.0850>

8.5.3 APIs

- Keras Optimizers API.
<https://keras.io/optimizers/>
- Keras Optimizers Source Code.
<https://github.com/keras-team/keras/blob/master/keras/optimizers.py>
- `sklearn.datasets.make_regression` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html

8.6 Summary

In this tutorial, you discovered the exploding gradient problem and how to improve neural network training stability using gradient clipping. Specifically, you learned:

- Training neural networks can become unstable, leading to a numerical overflow or underflow referred to as exploding gradients.
- The training process can be made stable by changing the error gradients, either by scaling the vector norm or clipping gradient values to a range.
- How to update an MLP model for a regression predictive modeling problem with exploding gradients to have a stable training process using gradient clipping methods.

8.6.1 Next

In the next tutorial, you will discover how to accelerate the training process through the use of batch normalization.

Chapter 9

Accelerate Learning with Batch Normalization

Training deep neural networks with tens of layers is challenging as they can be sensitive to the initial random weights and configuration of the learning algorithm. One possible reason for this difficulty is the distribution of the inputs to layers deep in the network may change after each minibatch when the weights are updated. This can cause the learning algorithm to forever chase a moving target. This change in the distribution of inputs to layers in the network is referred to by the technical name *internal covariate shift*. Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each minibatch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks. In this tutorial, you will discover the batch normalization method used to accelerate the training of deep learning neural networks. After reading this tutorial, you will know:

- Deep neural networks are challenging to train, not least because the input from prior layers can change after weight updates.
- Batch normalization is a technique to standardize the inputs to a network, applied to either the activations of a prior layer or inputs directly.
- Batch normalization accelerates training, in some cases by halving the number of epochs (or better), and provides some regularization effect, reducing generalization error.

Let's get started.

9.1 Batch Normalization

In this section you will discover batch normalization, the effect it has on the training process and the tips for using batch normalization on your own deep learning models.

9.1.1 Problem of Training Deep Networks

Training deep neural networks, e.g. networks with tens of hidden layers, is challenging. One aspect of this challenge is that the model is updated layer-by-layer backward from the output to

the input using an estimate of error that assumes the weights in the layers prior to the current layer are fixed.

Very deep models involve the composition of several functions or layers. The gradient tells how to update each parameter, under the assumption that the other layers do not change. In practice, we update all of the layers simultaneously.

— Page 317, *Deep Learning*, 2016.

Because all layers are changed during an update, the update procedure is forever chasing a moving target. For example, the weights of a layer are updated given an expectation that the prior layer outputs values with a given distribution. This distribution is likely changed after the weights of the prior layer are updated.

Training Deep Neural Networks is complicated by the fact that the distribution of each layer’s inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities.

— *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015.

The authors of the paper introducing batch normalization refer to change in the distribution of inputs during training as *internal covariate shift*.

We refer to the change in the distributions of internal nodes of a deep network, in the course of training, as Internal Covariate Shift.

— *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015.

9.1.2 Standardize Layer Inputs

Batch normalization, or batchnorm for short, is proposed as a technique to help coordinate the update of multiple layers in the model.

Batch normalization provides an elegant way of reparametrizing almost any deep network. The reparametrization significantly reduces the problem of coordinating updates across many layers.

— Page 318, *Deep Learning*, 2016.

It does this by scaling the output of the layer, specifically by standardizing the activations of each input variable per minibatch, such as the activations of a node from the previous layer. Recall that standardization refers to rescaling data to have a mean of zero and a standard deviation of one, e.g. a standard Gaussian.

Batch normalization reparametrizes the model to make some units always be standardized by definition

— Page 319, *Deep Learning*, 2016.

This process is also called *whitening* when applied to images in computer vision.

By whitening the inputs to each layer, we would take a step towards achieving the fixed distributions of inputs that would remove the ill effects of the internal covariate shift.

— *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015.

Standardizing the activations of the prior layer means that assumptions the subsequent layer makes about the spread and distribution of inputs during the weight update will not change, at least not dramatically. This has the effect of stabilizing and speeding-up the training process of deep neural networks.

Batch normalization acts to standardize only the mean and variance of each unit in order to stabilize learning, but allows the relationships between units and the nonlinear statistics of a single unit to change.

— Page 320, *Deep Learning*, 2016.

Normalizing the inputs to the layer has an effect on the training of the model, dramatically reducing the number of epochs required. It can also have a regularizing effect, reducing generalization error much like the use of activation regularization.

Batch normalization can have a dramatic effect on optimization performance, especially for convolutional networks and networks with sigmoidal nonlinearities.

— Page 425, *Deep Learning*, 2016.

Although reducing *internal covariate shift* was a motivation in the development of the method, there is some suggestion that instead batch normalization is effective because it smooths and, in turn, simplifies the optimization function that is being solved when training the network.

... BatchNorm impacts network training in a fundamental way: it makes the landscape of the corresponding optimization problem be significantly more smooth. This ensures, in particular, that the gradients are more predictive and thus allow for use of larger range of learning rates and faster network convergence.

— *How Does Batch Normalization Help Optimization? (No, It Is Not About Internal Covariate Shift)*, 2018.

9.1.3 How to Standardize Layer Inputs

Batch normalization can be implemented during training by calculating the mean and standard deviation of each input variable to a layer per minibatch and using these statistics to perform the standardization. Alternately, a running average of mean and standard deviation can be maintained across minibatches, but may result in unstable training.

It is natural to ask whether we could simply use the moving averages [...] to perform the normalization during training [...]. This, however, has been observed to lead to the model blowing up.

— *Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models*, 2017.

After training, the mean and standard deviation of inputs for the layer can be set as mean values observed over the training dataset. For small minibatch sizes or minibatches that do not contain a representative distribution of examples from the training dataset, the differences in the standardized inputs between training and inference (using the model after training) can result in noticeable differences in performance. This can be addressed with a modification of the method called Batch Renormalization (or BatchRenorm for short) that makes the estimates of the variable mean and standard deviation more stable across minibatches.

Batch Renormalization extends batchnorm with a per-dimension correction to ensure that the activations match between the training and inference networks.

— *Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models*, 2017.

This standardization of inputs may be applied to input variables for the first hidden layer or to the activations from a hidden layer for deeper layers. In practice, it is common to allow the layer to learn two new parameters, namely a new mean and standard deviation, Beta and Gamma respectively, that allow the automatic scaling and shifting of the standardized layer inputs. These parameters are learned by the model as part of the training process.

Note that simply normalizing each input of a layer may change what the layer can represent. [...] These parameters are learned along with the original model parameters, and restore the representation power of the network.

— *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015.

Importantly the backpropagation algorithm is updated to operate upon the transformed inputs, and error is also used to update the new scale and shifting parameters learned by the model. The standardization is applied to the inputs to the layer, namely the input variables or the output of the activation function from the prior layer. Given the choice of activation function, the distribution of the inputs to the layer may be quite non-Gaussian. In this case, there may be benefit in standardizing the summed activation before the activation function in the previous layer.

We add the BN transform immediately before the nonlinearity [...] We could have also normalized the layer inputs u , but since u is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift.

— *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015.

9.1.4 Examples of Using Batch Normalization

This section provides a few examples of milestone papers and popular models that make use of batch normalization. In the 2015 paper that introduced the technique titled *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, the authors Sergey Ioffe and Christian Szegedy from Google demonstrated a dramatic speedup of an Inception-based convolutional neural network for photo classification over a baseline method.

By only using Batch Normalization [...], we match the accuracy of Inception in less than half the number of training steps.

Kaiming He, et al. in their 2015 paper titled *Deep Residual Learning for Image Recognition* used batch normalization after the convolutional layers in their very deep model referred to as ResNet and achieve then state-of-the-art results on the ImageNet dataset, a standard photo classification task.

We adopt batch normalization (BN) right after each convolution and before activation

...

Christian Szegedy, et al. from Google in their 2016 paper titled *Rethinking the Inception Architecture for Computer Vision* used batch normalization in their updated inception model referred to as GoogleNet Inception-v3, achieving then state-of-the-art results on the ImageNet dataset.

BN-auxiliary refers to the version in which the fully connected layer of the auxiliary classifier is also batch-normalized, not just the convolutions.

Dario Amodei from Baidu in their 2016 paper titled *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin* use a variation of batch normalization recurrent neural networks in their end-to-end deep model for speech recognition.

... we find that when applied to very deep networks of RNNs on large data sets, the variant of BatchNorm we use substantially improves final generalization error in addition to accelerating training

9.1.5 Tips for Using Batch Normalization

This section provides tips and suggestions for using batch normalization with your own neural networks.

Use With Different Network Types

Batch normalization is a general technique that can be used to normalize the inputs to a layer. It can be used with most network types, such as Multilayer Perceptrons, Convolutional Neural Networks and Recurrent Neural Networks.

Probably Use Before the Activation

Batch normalization may be used on the inputs to the layer before or after the activation function in the previous layer. It may be more appropriate after the activation function for *s*-shaped functions like the hyperbolic tangent and logistic function. It may be appropriate before the activation function for activations that may result in non-Gaussian distributions like the rectified linear activation function, the modern default for most network types.

The goal of Batch Normalization is to achieve a stable distribution of activation values throughout training, and in our experiments we apply it before the nonlinearity since that is where matching the first and second moments is more likely to result in a stable distribution

— *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015.

Perhaps test both approaches with your network.

Use Large Learning Rates

Using batch normalization makes the network more stable during training. This may require the use of much larger than normal learning rates, that in turn may further speed up the learning process.

In a batch-normalized model, we have been able to achieve a training speedup from higher learning rates, with no ill side effects

— *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015.

The faster training also means that the decay rate used for the learning rate may be increased.

Less Sensitive to Weight Initialization

Deep neural networks can be quite sensitive to the technique used to initialize the weights prior to training. The stability to training brought by batch normalization can make training deep networks less sensitive to the choice of weight initialization method.

Alternate to Data Preparation

Batch normalization could be used to standardize raw input variables that have differing scales. If the mean and standard deviations calculated for each input feature are calculated over the minibatch instead of over the entire training dataset, then the batch size must be sufficiently representative of the range of each variable. It may not be appropriate for variables that have a data distribution that is highly non-Gaussian, in which case it might be better to perform data scaling as a pre-processing step.

Don't Use With Dropout

Batch normalization offers some regularization effect, reducing generalization error, perhaps no longer requiring the use of dropout for regularization.

Removing Dropout from Modified BN-Inception speeds up training, without increasing overfitting.

— *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015.

Further, it may not be a good idea to use batch normalization and dropout in the same network. The reason is that the statistics used to normalize the activations of the prior layer may become noisy given the random dropping out of nodes during the dropout procedure.

Batch normalization also sometimes reduces generalization error and allows dropout to be omitted, due to the noise in the estimate of the statistics used to normalize each variable.

— Page 425, *Deep Learning*, 2016.

9.2 Batch Normalization Keras API

This section demonstrates how to use batch normalization with the Keras API.

9.2.1 Batch Normalization in Keras

Keras provides support for batch normalization via the `BatchNormalization` layer. For example:

```
bn = BatchNormalization()
```

Listing 9.1: Example of creating a batch normalization layer.

The layer will transform inputs so that they are standardized, meaning that they will have a mean of zero and a standard deviation of one. During training, the layer will keep track of statistics for each input variable and use them to standardize the data. Further, the standardized output can be scaled using the learned parameters of Beta and Gamma that define the new mean and standard deviation for the output of the transform. The layer can be configured to control whether these additional parameters will be used or not via the `center` and `scale` attributes respectively. By default, they are enabled.

The statistics used to perform the standardization, e.g. the mean and standard deviation of each variable, are updated for each mini batch and a running average is maintained. A `momentum` argument allows you to control how much of the statistics from the previous mini batch to include when the update is calculated. By default, this is kept high with a value of 0.99. This can be set to 0.0 to only use statistics from the current minibatch, as described in the original paper.

```
bn = BatchNormalization(momentum=0.0)
```

Listing 9.2: Example of creating a batch normalization layer with momentum.

At the end of training, the mean and standard deviation statistics in the layer at that time will be used to standardize inputs when the model is used to make a prediction. The default configuration estimating mean and standard deviation across all mini batches is probably sensible.

9.2.2 BatchNormalization in Models

Batch normalization can be used at most points in a model and with most types of deep learning neural networks.

Input and Hidden Layer Inputs

The `BatchNormalization` layer can be added to your model to standardize raw input variables or the outputs of a hidden layer. Batch normalization is not recommended as an alternative to proper data preparation for your model. Nevertheless, when used to standardize the raw input variables, the layer must specify the `input_shape` argument; for example:

```
...
model = Sequential
model.add(BatchNormalization(input_shape=(2,)))
...
```

Listing 9.3: Example of batch normalization for input.

When used to standardize the outputs of a hidden layer, the layer can be added to the model just like any other layer.

```
...
model = Sequential
...
model.add(BatchNormalization())
...
```

Listing 9.4: Example of batch normalization between hidden layers.

Use Before or After the Activation Function

The `BatchNormalization` layer can be used to standardize inputs before or after the activation function of the previous layer. The original paper that introduced the method suggests adding batch normalization before the activation function of the previous layer, for example:

```
...
model = Sequential
model.add(Dense(32))
model.add(BatchNormalization())
model.add(Activation('relu'))
...
```

Listing 9.5: Example of batch normalization before the activation function.

Some reported experiments suggest better performance when adding the batch normalization layer after the activation function of the previous layer; for example:

```

...
model = Sequential
model.add(Dense(32, activation='relu'))
model.add(BatchNormalization())
...

```

Listing 9.6: Example of batch normalization after the activation function.

If time and resources permit, it may be worth testing both approaches on your model and use the approach that results in the best performance. Let's take a look at how batch normalization can be used with some common network types.

MLP Batch Normalization

The example below adds batch normalization after the activation function between `Dense` hidden layers.

```

# example of batch normalization for an mlp
from keras.layers import Dense
from keras.layers import BatchNormalization
...
model.add(Dense(32, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(1))
...

```

Listing 9.7: Example of batch normalization for an MLP model.

CNN Batch Normalization

The example below adds batch normalization after the activation function between a convolutional and max pooling layers.

```

# example of batch normalization for an cnn
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import BatchNormalization
...
model.add(Conv2D(32, (3,3), activation='relu'))
model.add(Conv2D(32, (3,3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())
model.add(Dense(1))
...

```

Listing 9.8: Example of batch normalization for a CNN model.

RNN Batch Normalization

The example below adds batch normalization after the activation function between an LSTM and `Dense` hidden layers.

```
# example of batch normalization for a lstm
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import BatchNormalization
...
model.add(LSTM(32))
model.add(BatchNormalization())
model.add(Dense(1))
...
```

Listing 9.9: Example of batch normalization for an LSTM model.

9.3 Batch Normalization Case Study

In this section, we will demonstrate how to use batch normalization to accelerate the training of an MLP on a simple binary classification problem. This example provides a template for applying batch normalization to your own neural network for classification and regression problems.

9.3.1 Binary Classification Problem

We will use a standard binary classification problem that defines two two-dimensional concentric circles of observations, one circle for each class. Each observation has two input variables with the same scale and a class output value of either 0 or 1. This dataset is called the *circles* dataset because of the shape of the observations in each class when plotted. We can use the `make_circles()` function to generate observations from this problem. We will add noise to the data and seed the random number generator so that the same samples are generated each time the code is run.

```
# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
```

Listing 9.10: Example of a generating samples for the circles problem.

We can plot the dataset where the two variables are taken as x and y coordinates on a graph and the class value is taken as the color of the observation. The complete example of generating the dataset and plotting it is listed below.

```
# scatter plot of the circles dataset with points colored by class
from sklearn.datasets import make_circles
from numpy import where
from matplotlib import pyplot
# generate circles
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# select indices of points with each class label
for i in range(2):
    samples_ix = where(y == i)
    pyplot.scatter(X[samples_ix, 0], X[samples_ix, 1], label=str(i))
pyplot.legend()
pyplot.show()
```

Listing 9.11: Example of plotting samples from the circles problem.

Running the example creates a scatter plot showing the concentric circles shape of the observations in each class. We can see the noise in the dispersal of the points making the circles less obvious.

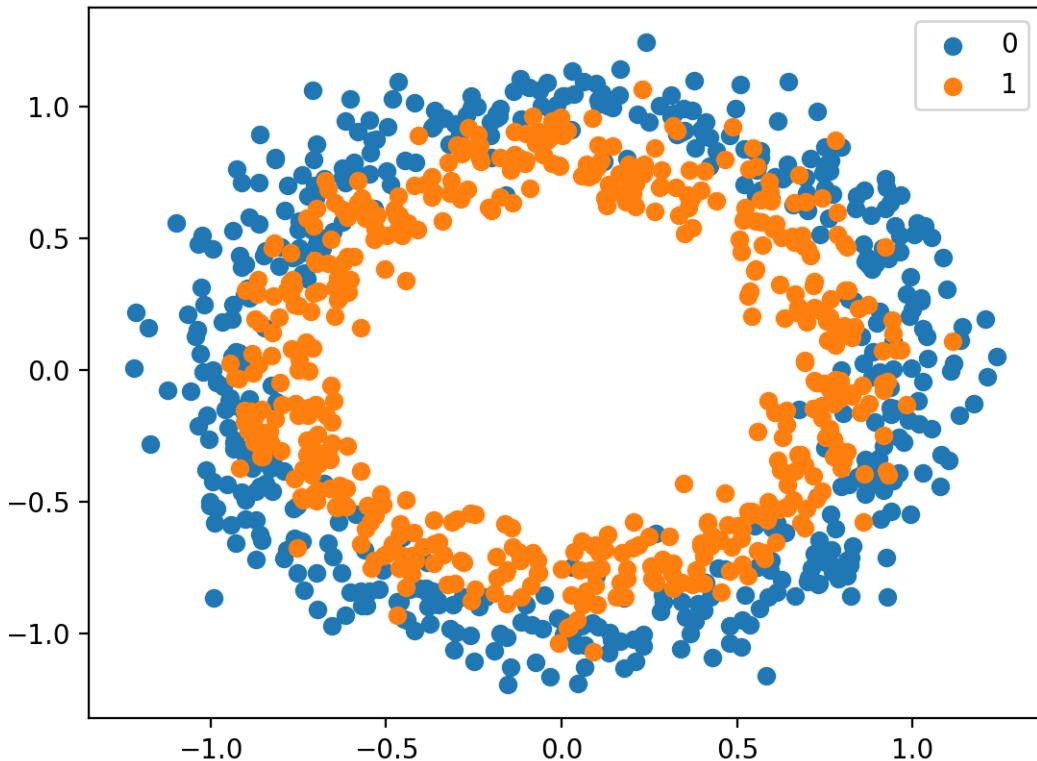


Figure 9.1: Scatter Plot of Circles Dataset With Color Showing the Class Value of Each Sample.

This is a good test problem because the classes cannot be separated by a line, e.g. are not linearly separable, requiring a nonlinear method such as a neural network to address.

9.3.2 Multilayer Perceptron Model

We can develop a Multilayer Perceptron model, or MLP, as a baseline for this problem. First, we will split the 1,000 generated samples into a train and test dataset, with 500 examples in each. This will provide a sufficiently large sample for the model to learn from and an equally sized (fair) evaluation of its performance.

```
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 9.12: Example of preparing the dataset for model.

We will define a simple MLP model. The network must have two inputs in the visible layer for the two variables in the dataset. The model will have a single hidden layer with 50 nodes, chosen arbitrarily, and use the rectified linear activation function and the He random weight initialization method. The output layer will be a single node with the sigmoid activation function, capable of predicting a 0 for the outer circle and a 1 for the inner circle of the problem. The model will be trained using stochastic gradient descent with a modest learning rate of 0.01 and a large momentum of 0.9, and the optimization will be directed using the binary cross-entropy loss function.

```
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 9.13: Example of defining the MLP model.

Once defined, the model can be fit on the training dataset. We will use the holdout test dataset as a validation dataset and evaluate its performance at the end of each training epoch. The model will be fit for 100 epochs, chosen after a little trial and error.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
```

Listing 9.14: Example of fitting the MLP model.

At the end of the run, the model is evaluated on the train and test dataset and the accuracy is reported.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 9.15: Example of evaluating the MLP model.

Finally, line plots are created showing model accuracy on the train and test sets at the end of each training epoch providing learning curves. This plot of learning curves is useful as it gives an idea of how quickly and how well the model has learned the problem.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 9.16: Example of plotting learning curves the MLP model.

Tying these elements together, the complete example is listed below.

```
# mlp for the two circles problem
from sklearn.datasets import make_circles
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='sigmoid'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 9.17: Example evaluating an MLP on the two circles problem.

Running the example fits the model and evaluates it on the train and test sets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved an accuracy of about 84% on the test dataset and achieved comparable performance on both the train and test sets, given the same size and similar composition of both datasets.

Train: 0.826, Test: 0.840

Listing 9.18: Example output from evaluating an MLP on the two circles problem.

A graph is created showing line plots of the classification accuracy on the train (blue) and test (orange) datasets. The plot shows comparable performance of the model on both datasets

during the training process. We can see that performance leaps up over the first 30-to-40 epochs to above 80% accuracy then is slowly refined.

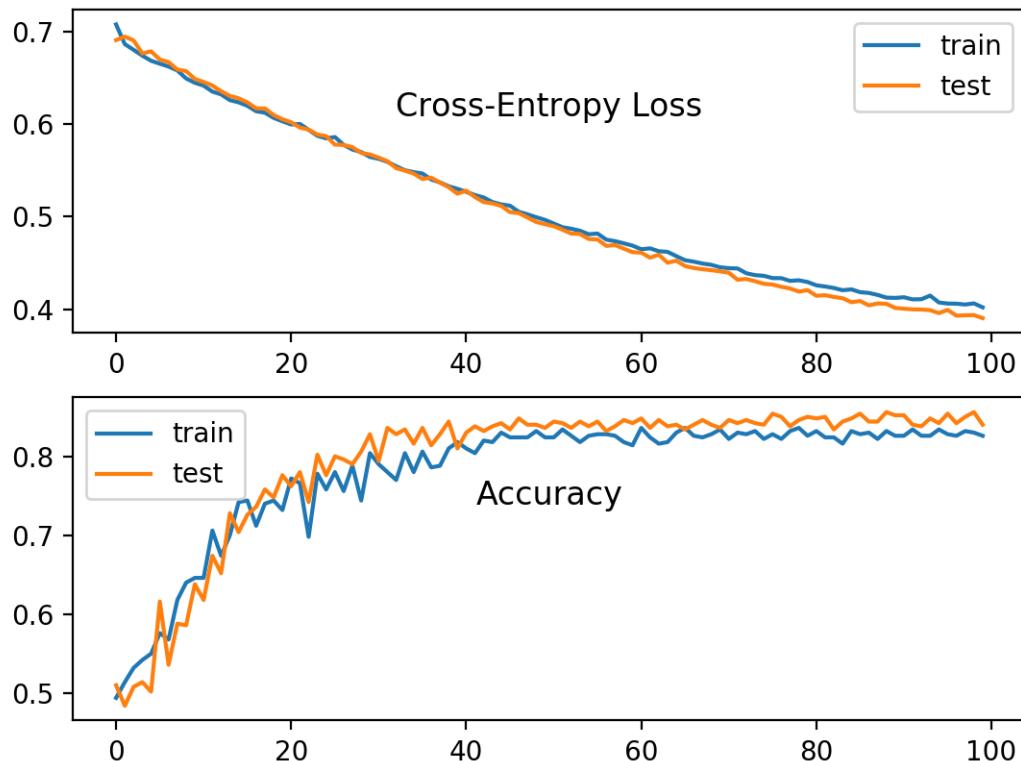


Figure 9.2: Line Plot of MLP Classification Accuracy on Train and Test Datasets Over Training Epochs.

This result, and specifically the dynamics of the model during training, provide a baseline that can be compared to the same model with the addition of batch normalization.

9.3.3 MLP With Batch Normalization

The model introduced in the previous section can be updated to add batch normalization. The expectation is that the addition of batch normalization would accelerate the training process, offering similar or better classification accuracy of the model in fewer training epochs. Batch normalization is also reported as providing a modest form of regularization, meaning that it may also offer a small reduction in generalization error demonstrated by a small increase in classification accuracy on the holdout test dataset. A new `BatchNormalization` layer can be added to the model after the hidden layer before the output layer. Specifically, after the activation function of the prior hidden layer.

```
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
```

```

model.add(BatchNormalization())
model.add(Dense(1, activation='sigmoid'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

```

Listing 9.19: Example of updating the model to use batch normalization.

The complete example with this modification is listed below.

```

# mlp for the two circles problem with batchnorm after activation function
from sklearn.datasets import make_circles
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import BatchNormalization
from keras.optimizers import SGD
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Dense(1, activation='sigmoid'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 9.20: Example evaluating an MLP with batch normalization after activation on the two circles problem.

Running the example first prints the classification accuracy of the model on the train and test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see comparable performance of the model on both the train and test set of about 84% accuracy, very similar to what we saw in the previous section, if not a little bit better.

```
Train: 0.842, Test: 0.846
```

Listing 9.21: Example output from evaluating an MLP with batch normalization after activation on the two circles problem.

A graph of the learning curves is also created showing classification accuracy on both the train and test sets for each training epoch. In this case, we can see that the model has learned the problem faster than the model in the previous section without batch normalization. Specifically, we can see that classification accuracy on the train and test datasets leaps above 80% within the first 20 epochs, as opposed to 30-to-40 epochs in the model without batch normalization. The plot also shows the effect of batch normalization during training. We can see lower performance on the training dataset than the test dataset: scores on the training dataset that are lower than the performance on the test dataset at the end of the training run. This is likely the effect of the input collected and updated each minibatch.

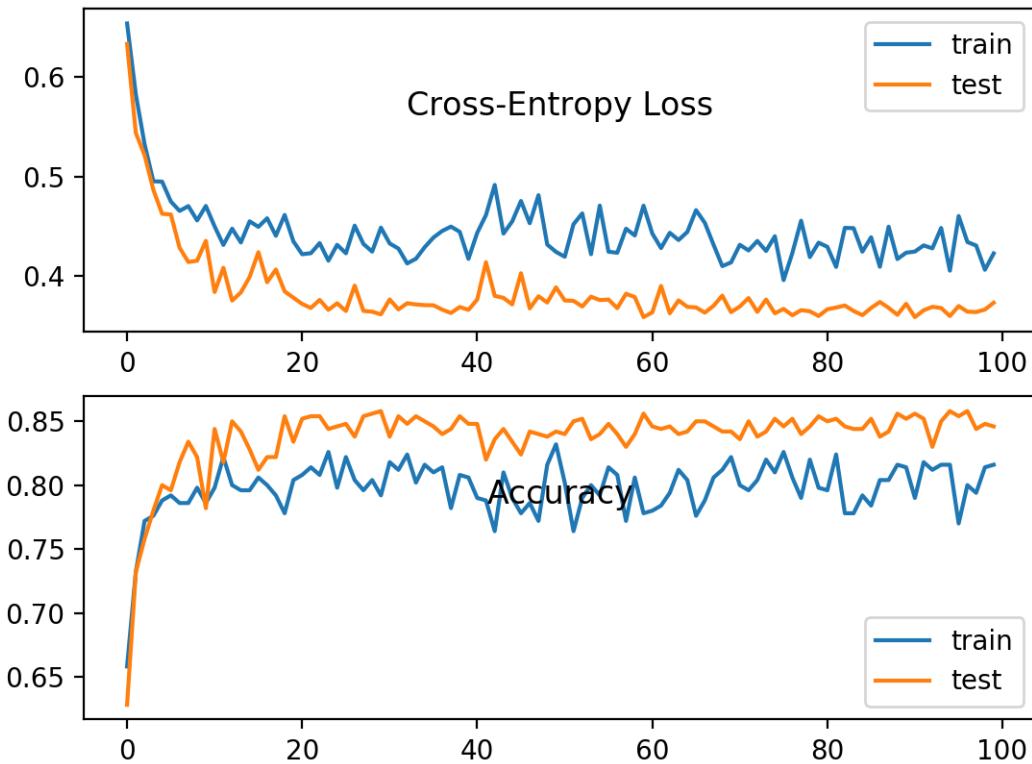


Figure 9.3: Line Plot Classification Accuracy of MLP With Batch Normalization After Activation Function on Train and Test Datasets Over Training Epochs.

We can also try a variation of the model where batch normalization is applied prior to the activation function of the hidden layer, instead of after the activation function.

```
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(1, activation='sigmoid'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 9.22: Example of updating the model to use batch normalization before the activation function.

The complete code listing with this change to the model is listed below.

```
# mlp for the two circles problem with batchnorm before activation function
from sklearn.datasets import make_circles
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Activation
from keras.layers import BatchNormalization
from keras.optimizers import SGD
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(1, activation='sigmoid'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 9.23: Example evaluating an MLP with batch normalization before activation on the two circles problem.

Running the example first prints the classification accuracy of the model on the train and test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see comparable performance of the model on the train and test datasets, but slightly worse than the model without batch normalization.

```
Train: 0.832, Test: 0.834
```

Listing 9.24: Example output from evaluating an MLP with batch normalization before activation on the two circles problem.

The line plot of the learning curves on the train and test sets also tells a different story. The plot shows the model learning perhaps at the same pace as the model without batch normalization, but the performance of the model on the training dataset is much worse, hovering around 70% to 75% accuracy, again likely an effect of the statistics collected and used over each minibatch. At least for this model configuration on this specific dataset, it appears that batch normalization is more effective after the rectified linear activation function.

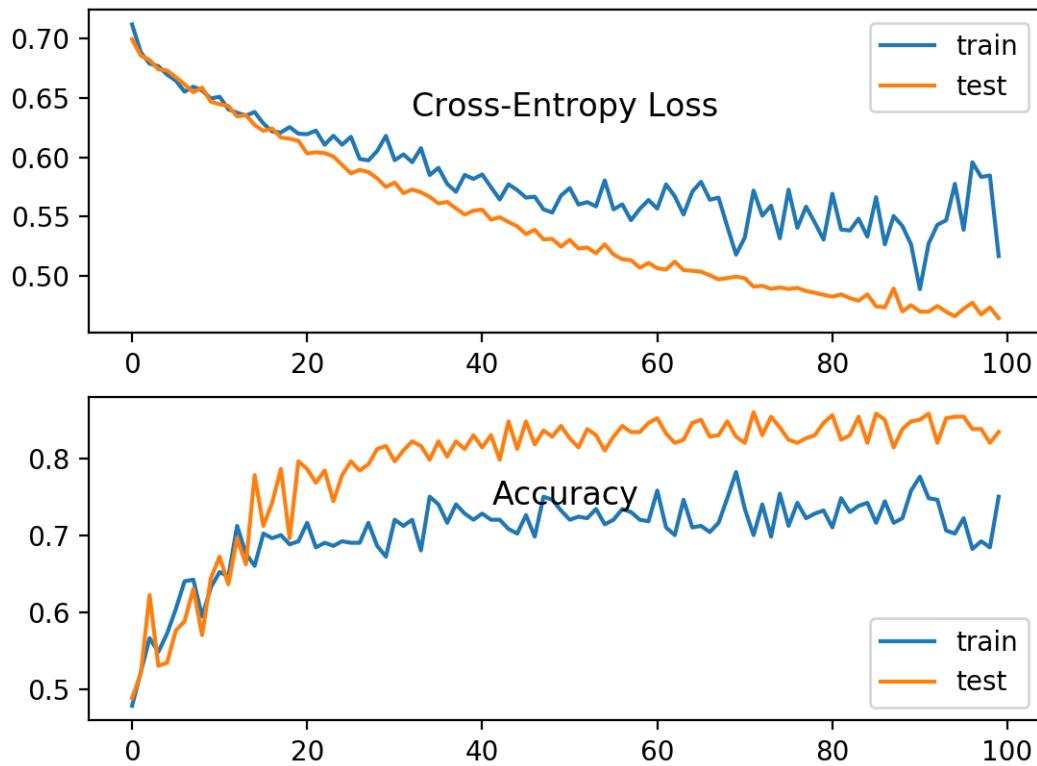


Figure 9.4: Line Plot Classification Accuracy of MLP With Batch Normalization Before Activation Function on Train and Test Datasets Over Training Epochs.

9.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Without Beta and Gamma.** Update the example to not use the beta and gamma parameters in the batch normalization layer and compare results.
- **Without Momentum.** Update the example to not use momentum in the batch normalization layer during training and compare results.
- **Input Layer.** Update the example to use batch normalization after the input to the model and compare results.

If you explore any of these extensions, I'd love to know.

9.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

9.5.1 Books

- Section 8.7.1: Batch Normalization, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- Section 7.3.1: Advanced architecture patterns, *Deep Learning With Python*, 2017.
<https://amzn.to/2Ck4ImT>

9.5.2 Papers

- *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015.
<https://arxiv.org/abs/1502.03167>
- *Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models*, 2017.
<https://arxiv.org/abs/1702.03275>
- *How Does Batch Normalization Help Optimization? (No, It Is Not About Internal Covariate Shift)*, 2018.
<https://arxiv.org/abs/1805.11604>

9.5.3 APIs

- Keras Regularizers API.
<https://keras.io/regularizers/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>
- Keras Recurrent Layers API.
<https://keras.io/layers/recurrent/>
- BatchNormalization Keras API.
<https://keras.io/layers/normalization/>
- sklearn.datasets.make_circles API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html

9.5.4 Articles

- Batch normalization, Wikipedia.
https://en.wikipedia.org/wiki/Batch_normalization
- Batch Normalization before or after ReLU?, Reddit.
https://www.reddit.com/r/MachineLearning/comments/67gonq/d_batch_normalization_before_or_after_relu/

- The Batch Normalization layer of Keras is broken, Vasilis Vryniotis, 2018.
<http://blog.datumbox.com/the-batch-normalization-layer-of-keras-is-broken/>
- Studies of Batch Normalization Before and After Activation Function.
<https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md>

9.6 Summary

In this tutorial, you discovered the batch normalization method used to accelerate the training of deep learning neural networks. Specifically, you learned:

- Deep neural networks are challenging to train, not least because the input from prior layers can change after weight updates.
- Batch normalization is a technique to standardize the inputs to a network, applied to either the activations of a prior layer or inputs directly.
- Batch normalization accelerates training, in some cases by halving the number of epochs (or better), and provides some regularization effect, reducing generalization error.

9.6.1 Next

In the next tutorial, you will discover how to develop deeper neural network models with greedy layer-wise pretraining.

Chapter 10

Deeper Models with Greedy Layer-Wise Pretraining

Training deep neural networks was traditionally challenging as the vanishing gradient meant that weights in layers close to the input layer were not updated in response to errors calculated on the training dataset. An innovation and important milestone in the field of deep learning was greedy layer-wise pretraining that allowed very deep neural networks to be successfully trained, achieving then state-of-the-art performance. In this tutorial, you will discover greedy layer-wise pretraining as a technique for developing deep multilayered neural network models.

After completing this tutorial, you will know:

- Greedy layer-wise pretraining provides a way to develop deep multilayered neural networks whilst only ever training shallow networks.
- Pretraining can be used to iteratively deepen a supervised model or an unsupervised model that can be repurposed as a supervised model.
- Pretraining may be useful for problems with small amounts labeled data and large amounts of unlabeled data.

Let's get started.

10.1 Greedy Layer-Wise Pretraining

Traditionally, training deep neural networks with many layers was challenging. As the number of hidden layers is increased, the amount of error information propagated back to earlier layers is dramatically reduced. This means that weights in hidden layers close to the output layer are updated normally, whereas weights in hidden layers close to the input layer are updated minimally or not at all. Generally, this problem prevented the training of very deep neural networks and was referred to as the vanishing gradient problem. An important milestone in the resurgence of neural networks that initially allowed the development of deeper neural network models was the technique of greedy layer-wise pretraining, often simply referred to as *pretraining*.

The deep learning renaissance of 2006 began with the discovery that this greedy learning procedure could be used to find a good initialization for a joint learning

procedure over all the layers, and that this approach could be used to successfully train even fully connected architectures.

— Page 528, *Deep Learning*, 2016.

Pretraining involves successively adding a new hidden layer to a model and refitting, allowing the newly added model to learn the inputs from the existing hidden layer, often while keeping the weights for the existing hidden layers fixed. This gives the technique the name *layer-wise* as the model is trained one layer at a time. The technique is referred to as *greedy* because of the piecewise or layer-wise approach to solving the harder problem of training a deep network. As an optimization process, dividing the training process into a succession of layer-wise training processes is seen as a greedy shortcut that likely leads to an aggregate of locally optimal solutions, a shortcut to a good enough global solution.

Greedy algorithms break a problem into many components, then solve for the optimal version of each component in isolation. Unfortunately, combining the individually optimal components is not guaranteed to yield an optimal complete solution.

— Page 323, *Deep Learning*, 2016.

Pretraining is based on the assumption that it is easier to train a shallow network instead of a deep network and contrives a layer-wise training process that we are always only ever fitting a shallow model.

... builds on the premise that training a shallow network is easier than training a deep one, which seems to have been validated in several contexts.

— Page 529, *Deep Learning*, 2016.

The key benefits of pretraining are:

- Simplified training process.
- Facilitates the development of deeper networks.
- Useful as a weight initialization scheme.
- Perhaps lower generalization error.

In general, pretraining may help both in terms of optimization and in terms of generalization.

— Page 325, *Deep Learning*, 2016.

There are two main approaches to pretraining; they are:

- Supervised greedy layer-wise pretraining.
- Unsupervised greedy layer-wise pretraining.

Broadly, supervised pretraining involves successively adding hidden layers to a model trained on a supervised learning task. Unsupervised pretraining involves using the greedy layer-wise process to build up an unsupervised autoencoder model, to which a supervised output layer is later added.

It is common to use the word “pretraining” to refer not only to the pretraining stage itself but to the entire two phase protocol that combines the pretraining phase and a supervised learning phase. The supervised learning phase may involve training a simple classifier on top of the features learned in the pretraining phase, or it may involve supervised fine-tuning of the entire network learned in the pretraining phase.

— Page 529, *Deep Learning*, 2016.

Unsupervised pretraining may be appropriate when you have a significantly larger number of unlabeled examples that can be used to initialize a model prior to using a much smaller number of examples to fine tune the model weights for a supervised task.

... we can expect unsupervised pretraining to be most helpful when the number of labeled examples is very small. Because the source of information added by unsupervised pretraining is the unlabeled data, we may also expect unsupervised pretraining to perform best when the number of unlabeled examples is very large.

— Page 532, *Deep Learning*, 2016.

Although the weights in prior layers are held constant, it is common to fine tune all weights in the network at the end after the addition of the final layer. As such, this allows pretraining to be considered a type of weight initialization method.

... it makes use of the idea that the choice of initial parameters for a deep neural network can have a significant regularizing effect on the model (and, to a lesser extent, that it can improve optimization).

— Pages 530-531, *Deep Learning*, 2016.

Greedy layer-wise pretraining is an important milestone in the history of deep learning, that allowed the early development of networks with more hidden layers than was previously possible. The approach can be useful on some problems; for example, it is best practice to use unsupervised pretraining for text data in order to provide a richer distributed representation of words and their interrelationships via Word2Vec.

Today, unsupervised pretraining has been largely abandoned, except in the field of natural language processing [...] the advantage of pretraining is that one can pretrain once on a huge unlabeled set (for example with a corpus containing billions of words), learn a good representation (typically of words, but also of sentences), and then use this representation or fine-tune it for a supervised task for which the training set contains substantially fewer examples.

— Page 535, *Deep Learning*, 2016.

Nevertheless, it is likely that better performance may be achieved using modern methods such as better activation functions, weight initialization, variants of gradient descent, and regularization methods.

Today, we now know that greedy layer-wise pretraining is not required to train fully connected deep architectures, but the unsupervised pretraining approach was the first method to succeed.

— Page 528, Deep Learning, 2016.

10.2 Greedy Layer-Wise Pretraining Case Study

In this section, we will demonstrate how to use greedy layer-wise pretraining to develop deeper MLP models on a multiclass classification problem. This example provides a template for applying greedy layer-wise pretraining to your own neural network for classification and regression problems.

10.2.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate the effect of greedy layer-wise pretraining on model performance. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. The problem will be configured with two input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same data points.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

Listing 10.1: Example of a generating samples for the blobs problem.

The results are the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can plot each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot for each class value
for class_value in range(3):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 10.2: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (separable by a line), causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions.

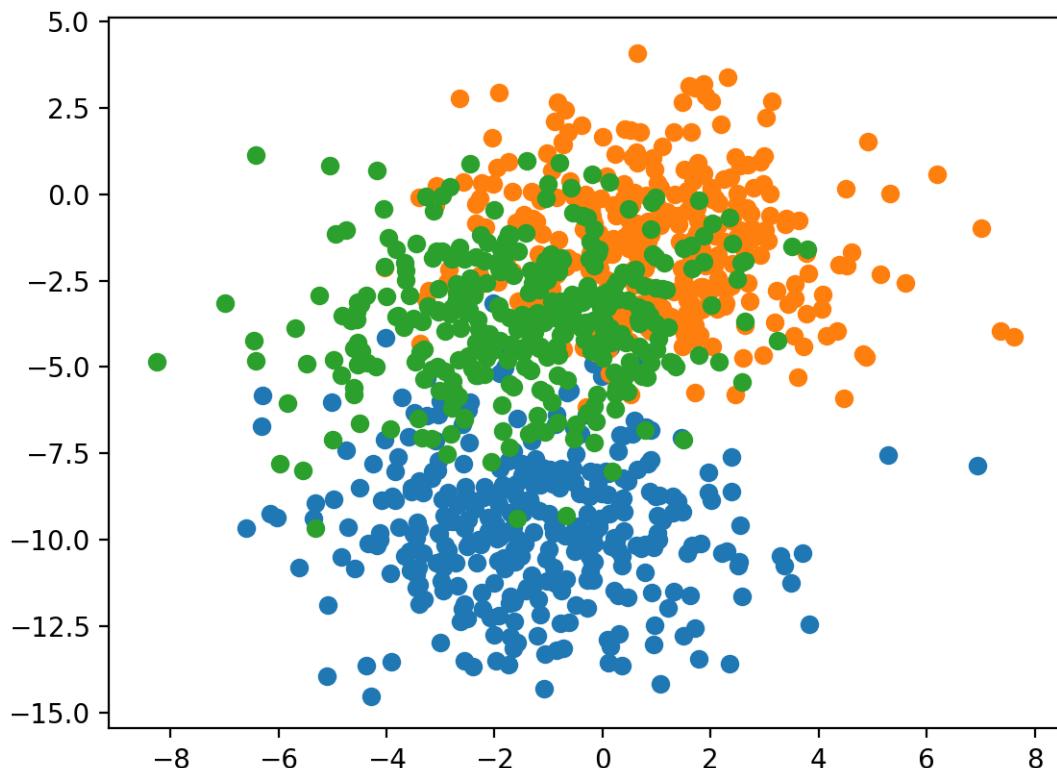


Figure 10.1: Scatter Plot of Blobs Dataset With Three Classes and Points Colored by Class Value.

10.2.2 Supervised Greedy Layer-Wise Pretraining

In this section, we will use greedy layer-wise supervised learning to build up a deep Multilayer Perceptron (MLP) model for the blobs supervised learning multiclass classification problem. Pretraining is not required to address this simple predictive modeling problem. Instead, this is a demonstration of how to perform supervised greedy layer-wise pretraining that can be used as a template for larger and more challenging supervised learning problems. As a first step, we can develop a function to create 1,000 samples from the problem and split them evenly into train and test datasets. The `prepare_data()` function below implements this and returns the train and test sets in terms of the input and output components.

```
# prepare the dataset
```

```
def prepare_data():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, testX, trainy, testy
```

Listing 10.3: Example of a function for preparing the dataset for modeling.

We can call this function to prepare the data.

```
# prepare data
trainX, testX, trainy, testy = prepare_data()
```

Listing 10.4: Example of calling the function to prepare the dataset for modeling.

Next, we can train and fit a base model. This will be an MLP that expects two inputs for the two input variables in the dataset and has one hidden layer with 10 nodes and uses the rectified linear activation function. The output layer has three nodes in order to predict the probability for each of the three classes and uses the softmax activation function.

```
# define model
model = Sequential()
model.add(Dense(10, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))
```

Listing 10.5: Example of defining an MLP model for the blobs problem.

The model is fit using stochastic gradient descent with the sensible default learning rate of 0.01 and a high momentum value of 0.9. The model is optimized using cross-entropy loss.

```
# compile model
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 10.6: Example of compiling an MLP model for the blobs problem.

The model is then fit on the training dataset for 100 epochs with a default batch size of 32 examples.

```
# fit model
model.fit(trainX, trainy, epochs=100, verbose=0)
```

Listing 10.7: Example of fitting an MLP model for the blobs problem.

The `get_base_model()` function below ties these elements together, taking the training dataset as arguments and returning a fit baseline model.

```
# define and fit the base model
def get_base_model(trainX, trainy):
    # define model
    model = Sequential()
    model.add(Dense(10, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
```

```

opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
model.fit(trainX, trainy, epochs=100, verbose=0)
return model

```

Listing 10.8: Example of a function for fitting and returning a model for the blobs problem.

We can call this function to prepare the base model to which we can later add layers one at a time.

```

# get the base model
model = get_base_model(trainX, trainy)

```

Listing 10.9: Example of calling the function to fit the model on the dataset.

We need to be able to easily evaluate the performance of a model on the train and test sets. The `evaluate_model()` function below takes the train and test sets as arguments as well as a model and returns the accuracy on both datasets.

```

# evaluate a fit model
def evaluate_model(model, trainX, testX, trainy, testy):
    _, train_acc = model.evaluate(trainX, trainy, verbose=0)
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    return train_acc, test_acc

```

Listing 10.10: Example of a function to evaluate a fit model.

We can call this function to calculate and report the accuracy of the base model and store the scores away in a dictionary against the number of layers in the model (currently two, one hidden and one output layer) so we can plot the relationship between layers and accuracy later.

```

# evaluate the base model
scores = dict()
train_acc, test_acc = evaluate_model(model, trainX, testX, trainy, testy)
print('> layers=%d, train=%f, test=%f' % (len(model.layers), train_acc, test_acc))

```

Listing 10.11: Example of summarizing the performance of a fit model.

We can now outline the process of greedy layer-wise pretraining. A function is required that can add a new hidden layer and retrain the model but only update the weights in the newly added layer and in the output layer. This requires first storing the current output layer including its configuration and current set of weights.

```

# remember the current output layer
output_layer = model.layers[-1]

```

Listing 10.12: Example of referencing the output layer.

Then removing the output layer from the stack of layers in the model.

```

# remove the output layer
model.pop()

```

Listing 10.13: Example of removing the output layer.

All of the remaining layers in the model can then be marked as non-trainable, meaning that their weights cannot be updated when the `fit()` function is called again.

```
# mark all remaining layers as non-trainable
for layer in model.layers:
    layer.trainable = False
```

Listing 10.14: Example of making all hidden layers not trainable.

We can then add a new hidden layer, in this case with the same configuration as the first hidden layer added in the base model.

```
# add a new hidden layer
model.add(Dense(10, activation='relu', kernel_initializer='he_uniform'))
```

Listing 10.15: Example of adding a new hidden layer.

Finally, the output layer can be added back and the model can be refit on the training dataset.

```
# re-add the output layer
model.add(output_layer)
# fit model
model.fit(trainX, trainy, epochs=100, verbose=0)
```

Listing 10.16: Example of restoring the output layer.

We can tie all of these elements into a function named `add_layer()` that takes the model and the training dataset as arguments.

```
# add one new layer and re-train only the new layer
def add_layer(model, trainX, trainy):
    # remember the current output layer
    output_layer = model.layers[-1]
    # remove the output layer
    model.pop()
    # mark all remaining layers as non-trainable
    for layer in model.layers:
        layer.trainable = False
    # add a new hidden layer
    model.add(Dense(10, activation='relu', kernel_initializer='he_uniform'))
    # re-add the output layer
    model.add(output_layer)
    # fit model
    model.fit(trainX, trainy, epochs=100, verbose=0)
```

Listing 10.17: Example of a function to add and train a new hidden layer to an existing model.

This function can then be called repeatedly based on the number of layers we wish to add to the model. In this case, we will add 10 layers, one at a time, and evaluate the performance of the model after each additional layer is added to get an idea of how it is impacting performance. Train and test accuracy scores are stored in the dictionary against the number of layers in the model.

```
# add layers and evaluate the updated model
n_layers = 10
for i in range(n_layers):
    # add layer
    add_layer(model, trainX, trainy)
    # evaluate model
```

```

train_acc, test_acc = evaluate_model(model, trainX, testX, trainy, testy)
print('> layers=%d, train=%.3f, test=%.3f' % (len(model.layers), train_acc, test_acc))
# store scores for plotting
scores[len(model.layers)] = (train_acc, test_acc)

```

Listing 10.18: Example of evaluating the performance of models with added pre-trained layers.

At the end of the run, a line plot is created showing the number of layers in the model (*x*-axis) compared to the model accuracy on the train and test datasets. We would expect the addition of layers to improve the performance of the model on the training dataset and perhaps even on the test dataset.

```

# plot number of added layers vs accuracy
pyplot.plot(list(scores.keys()), [scores[k][0] for k in scores.keys()], label='train',
            marker='.')
pyplot.plot(list(scores.keys()), [scores[k][1] for k in scores.keys()], label='test',
            marker='.')
pyplot.legend()
pyplot.show()

```

Listing 10.19: Example of plotting the number of pre-trained layers vs model performance.

Tying all of these elements together, the complete example is listed below.

```

# supervised greedy layer-wise pretraining for blobs classification problem
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare the dataset
def prepare_data():
    # generate 2d classification dataset
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, testX, trainy, testy

# define and fit the base model
def get_base_model(trainX, trainy):
    # define model
    model = Sequential()
    model.add(Dense(10, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=100, verbose=0)
    return model

```

```

# evaluate a fit model
def evaluate_model(model, trainX, testX, trainy, testy):
    _, train_acc = model.evaluate(trainX, trainy, verbose=0)
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    return train_acc, test_acc

# add one new layer and re-train only the new layer
def add_layer(model, trainX, trainy):
    # remember the current output layer
    output_layer = model.layers[-1]
    # remove the output layer
    model.pop()
    # mark all remaining layers as non-trainable
    for layer in model.layers:
        layer.trainable = False
    # add a new hidden layer
    model.add(Dense(10, activation='relu', kernel_initializer='he_uniform'))
    # re-add the output layer
    model.add(output_layer)
    # fit model
    model.fit(trainX, trainy, epochs=100, verbose=0)

# prepare data
trainX, testX, trainy, testy = prepare_data()
# get the base model
model = get_base_model(trainX, trainy)
# evaluate the base model
scores = dict()
train_acc, test_acc = evaluate_model(model, trainX, testX, trainy, testy)
print('> layers=%d, train=%.3f, test=%.3f' % (len(model.layers), train_acc, test_acc))
scores[len(model.layers)] = (train_acc, test_acc)
# add layers and evaluate the updated model
n_layers = 10
for i in range(n_layers):
    # add layer
    add_layer(model, trainX, trainy)
    # evaluate model
    train_acc, test_acc = evaluate_model(model, trainX, testX, trainy, testy)
    print('> layers=%d, train=%.3f, test=%.3f' % (len(model.layers), train_acc, test_acc))
    # store scores for plotting
    scores[len(model.layers)] = (train_acc, test_acc)
# plot number of added layers vs accuracy
pyplot.plot(list(scores.keys()), [scores[k][0] for k in scores.keys()], label='train',
            marker='.')
pyplot.plot(list(scores.keys()), [scores[k][1] for k in scores.keys()], label='test',
            marker='.')
pyplot.legend()
pyplot.show()

```

Listing 10.20: Example of supervised pre-training on the blobs problem.

Running the example reports the classification accuracy on the train and test sets for the base model (two layers), then after each additional layer is added (from three to 12 layers).

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the baseline model does reasonably well on this problem. As the layers are increased, we can roughly see an increase in accuracy for the model on the training dataset, likely as it is beginning to overfit the data. We see a rough drop in classification accuracy on the test dataset, likely because of the overfitting.

```
> layers=2, train=0.816, test=0.830
> layers=3, train=0.834, test=0.830
> layers=4, train=0.836, test=0.824
> layers=5, train=0.830, test=0.824
> layers=6, train=0.848, test=0.820
> layers=7, train=0.830, test=0.826
> layers=8, train=0.850, test=0.824
> layers=9, train=0.840, test=0.838
> layers=10, train=0.842, test=0.830
> layers=11, train=0.850, test=0.830
> layers=12, train=0.850, test=0.826
```

Listing 10.21: Example output from supervised pre-training on the blobs problem.

A line plot is also created showing the train (blue) and test set (orange) accuracy as each additional layer is added to the model. In this case, the plot suggests a slight overfitting of the training dataset, but perhaps better test set performance after seven added layers.

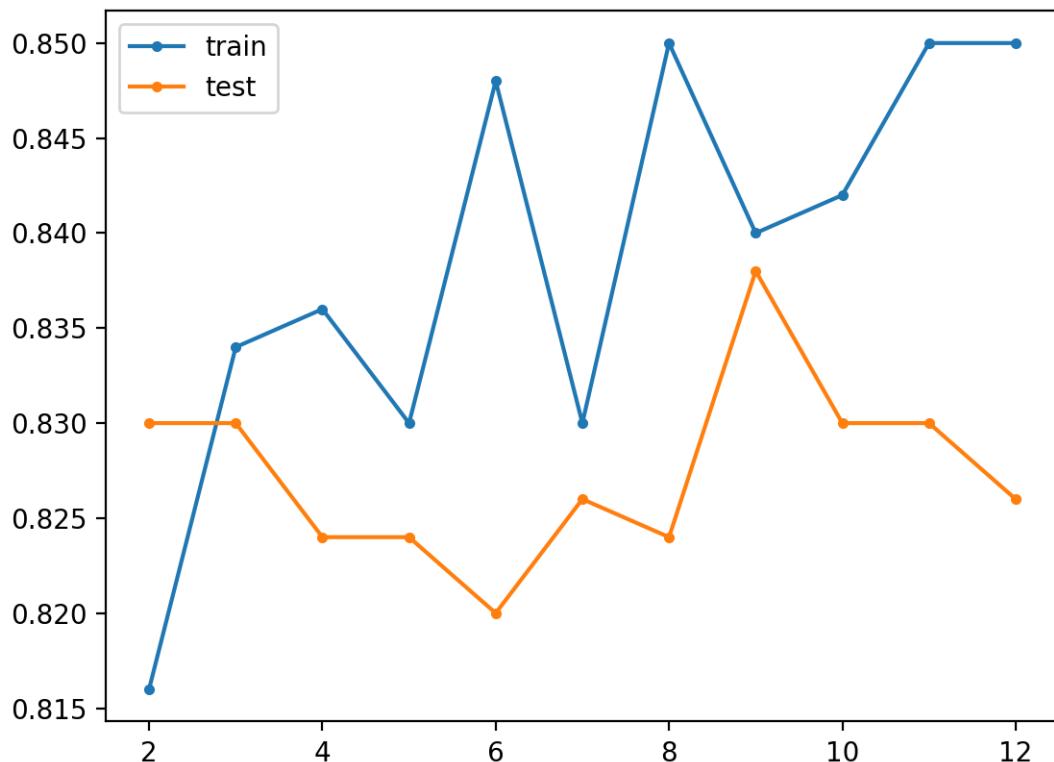


Figure 10.2: Line Plot for Supervised Greedy Layer-Wise Pretraining Showing Model Layers vs Train and Test Set Classification Accuracy on the Blobs Classification Problem.

An interesting extension to this example would be to allow all weights in the model to be fine tuned with a small learning rate for a large number of training epochs to see if this can further reduce generalization error.

10.2.3 Unsupervised Greedy Layer-Wise Pretraining

In this section, we will explore using greedy layer-wise pretraining with an unsupervised model. Specifically, we will develop an autoencoder model that will be trained to reconstruct input data. In order to use this unsupervised model for classification, we will remove the output layer, add and fit a new output layer for classification. This is slightly more complex than the previous supervised greedy layer-wise pretraining, but we can reuse many of the same ideas and code from the previous section.

The first step is to define, fit, and evaluate an autoencoder model. We will use the same two-layer base model as we did in the previous section, except modify it to predict the input as the output and use mean squared error to evaluate how good the model is at reconstructing a given input sample. The `base_autoencoder()` function below implements this, taking the train and test sets as arguments, then defines, fits, and evaluates the base unsupervised autoencoder model, printing the reconstruction error on the train and test sets and returning the model.

```
# define, fit and evaluate the base autoencoder
def base_autoencoder(trainX, testX):
    # define model
    model = Sequential()
    model.add(Dense(10, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(2, activation='linear'))
    # compile model
    model.compile(loss='mse', optimizer=SGD(lr=0.01, momentum=0.9))
    # fit model
    model.fit(trainX, trainX, epochs=100, verbose=0)
    # evaluate reconstruction loss
    train_mse = model.evaluate(trainX, trainX, verbose=0)
    test_mse = model.evaluate(testX, testX, verbose=0)
    print('> reconstruction error train=%f, test=%f' % (train_mse, test_mse))
    return model
```

Listing 10.22: Example of a function fit a base autoencoder.

We can call this function in order to prepare our base autoencoder to which we can add and greedily train layers.

```
# get the base autoencoder
model = base_autoencoder(trainX, testX)
```

Listing 10.23: Example of creating the base autoencoder.

Evaluating an autoencoder model on the blobs multiclass classification problem requires a few steps. The hidden layers will be used as the basis of a classifier with a new output layer that must be trained then used to make predictions before adding back the original output layer so that we can continue to add layers to the autoencoder. The first step is to reference, then remove the output layer of the autoencoder model.

```
# remember the current output layer
output_layer = model.layers[-1]
```

```
# remove the output layer
model.pop()
```

Listing 10.24: Example of removing the output layer of the autoencoder.

All of the remaining hidden layers in the autoencoder must be marked as non-trainable so that the weights are not changed when we train the new output layer.

```
# mark all remaining layers as non-trainable
for layer in model.layers:
    layer.trainable = False
```

Listing 10.25: Example of marking all pre-trained layers as non-trainable.

We can now add a new output layer that predicts the probability of an example belonging to each of the three classes. The model must also be re-compiled using a new loss function suitable for multiclass classification.

```
# add new output layer
model.add(Dense(3, activation='softmax'))
# compile model
model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.01, momentum=0.9),
               metrics=['accuracy'])
```

Listing 10.26: Example of adding a new classification output layer.

The model can then be re-fit on the training dataset, specifically training the output layer on how to make class predictions using the learned features from the autoencoder as input. The classification accuracy of the fit model can then be evaluated on the train and test datasets.

```
# fit model
model.fit(trainX, trainy, epochs=100, verbose=0)
# evaluate model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
```

Listing 10.27: Example of training a new classification output layer.

Finally, we can put the autoencoder back together but removing the classification output layer, adding back the original autoencoder output layer and recompiling the model with an appropriate loss function for reconstruction.

```
# put the model back together
model.pop()
model.add(output_layer)
model.compile(loss='mse', optimizer=SGD(lr=0.01, momentum=0.9))
```

Listing 10.28: Example of restoring the autoencoder output layer.

We can tie this together into an `evaluate_autoencoder_as_classifier()` function that takes the model as well as the train and test sets, then returns the train and test set classification accuracy.

```
# evaluate the autoencoder as a classifier
def evaluate_autoencoder_as_classifier(model, trainX, trainy, testX, testy):
    # remember the current output layer
    output_layer = model.layers[-1]
    # remove the output layer
```

```

model.pop()
# mark all remaining layers as non-trainable
for layer in model.layers:
    layer.trainable = False
# add new output layer
model.add(Dense(3, activation='softmax'))
# compile model
model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.01, momentum=0.9),
    metrics=['accuracy'])
# fit model
model.fit(trainX, trainy, epochs=100, verbose=0)
# evaluate model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
# put the model back together
model.pop()
model.add(output_layer)
model.compile(loss='mse', optimizer=SGD(lr=0.01, momentum=0.9))
return train_acc, test_acc

```

Listing 10.29: Example of a function for evaluating the autoencoder as a classification model.

This function can be called to evaluate the baseline autoencoder model and then store the accuracy scores in a dictionary against the number of layers in the model (in this case two).

```

# evaluate the base model
scores = dict()
train_acc, test_acc = evaluate_autoencoder_as_classifier(model, trainX, trainy, testX,
    testy)
print('> classifier accuracy layers=%d, train=%.3f, test=%.3f' % (len(model.layers),
    train_acc, test_acc))
scores[len(model.layers)] = (train_acc, test_acc)

```

Listing 10.30: Example of calling the function to evaluate an autoencoder as a classification model.

We are now ready to define the process for adding and pretraining layers to the model. The process for adding layers is much the same as the supervised case in the previous section, except we are optimizing reconstruction loss rather than classification accuracy for the new layer. The `add_layer_to_autoencoder()` function below adds a new hidden layer to the autoencoder model, updates the weights for the new layer and the hidden layers, then reports the reconstruction error on the train and test sets input data. The function does re-mark all prior layers as non-trainable, which is redundant because we already did this in the `evaluate_autoencoder_as_classifier()` function, but I have left it in, in case you decide to reuse this function in your own project.

```

# add one new layer and re-train only the new layer
def add_layer_to_autoencoder(model, trainX, testX):
    # remember the current output layer
    output_layer = model.layers[-1]
    # remove the output layer
    model.pop()
    # mark all remaining layers as non-trainable
    for layer in model.layers:
        layer.trainable = False
    # add a new hidden layer
    model.add(Dense(10, activation='relu', kernel_initializer='he_uniform'))

```

```
# re-add the output layer
model.add(output_layer)
# fit model
model.fit(trainX, trainX, epochs=100, verbose=0)
# evaluate reconstruction loss
train_mse = model.evaluate(trainX, trainX, verbose=0)
test_mse = model.evaluate(testX, testX, verbose=0)
print('> reconstruction error train=% .3f, test=% .3f' % (train_mse, test_mse))
```

Listing 10.31: Example of adding a hidden layer to the autoencoder model.

We can now repeatedly call this function, adding layers, and evaluating the effect by using the autoencoder as the basis for evaluating a new classifier.

```
# add layers and evaluate the updated model
n_layers = 5
for _ in range(n_layers):
    # add layer
    add_layer_to_autoencoder(model, trainX, testX)
    # evaluate model
    train_acc, test_acc = evaluate_autoencoder_as_classifier(model, trainX, trainy, testX,
        testy)
    print('> classifier accuracy layers=%d, train=% .3f, test=% .3f' % (len(model.layers),
        train_acc, test_acc))
    # store scores for plotting
    scores[len(model.layers)] = (train_acc, test_acc)
```

Listing 10.32: Example of adding layers and evaluating the autoencoder as a classification model.

As before, all accuracy scores are collected and we can use them to create a line graph of the number of model layers vs train and test set accuracy.

```
# plot number of added layers vs accuracy
keys = list(scores.keys())
pyplot.plot(keys, [scores[k][0] for k in keys], label='train', marker='.')
pyplot.plot(keys, [scores[k][1] for k in keys], label='test', marker='.')
pyplot.legend()
pyplot.show()
```

Listing 10.33: Example of plotting the performance of the number of autoencoder layers vs model performance.

Tying all of this together, the complete example of unsupervised greedy layer-wise pretraining for the blobs multiclass classification problem is listed below.

```
# unsupervised greedy layer-wise pretraining for blobs classification problem
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare the dataset
def prepare_data():
    # generate 2d classification dataset
```

```
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 500
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
return trainX, testX, trainy, testy

# define, fit and evaluate the base autoencoder
def base_autoencoder(trainX, testX):
    # define model
    model = Sequential()
    model.add(Dense(10, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(2, activation='linear'))
    # compile model
    model.compile(loss='mse', optimizer=SGD(lr=0.01, momentum=0.9))
    # fit model
    model.fit(trainX, trainX, epochs=100, verbose=0)
    # evaluate reconstruction loss
    train_mse = model.evaluate(trainX, trainX, verbose=0)
    test_mse = model.evaluate(testX, testX, verbose=0)
    print('> reconstruction error train=% .3f, test=% .3f' % (train_mse, test_mse))
    return model

# evaluate the autoencoder as a classifier
def evaluate_autoencoder_as_classifier(model, trainX, trainy, testX, testy):
    # remember the current output layer
    output_layer = model.layers[-1]
    # remove the output layer
    model.pop()
    # mark all remaining layers as non-trainable
    for layer in model.layers:
        layer.trainable = False
    # add new output layer
    model.add(Dense(3, activation='softmax'))
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.01, momentum=0.9),
                  metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=100, verbose=0)
    # evaluate model
    _, train_acc = model.evaluate(trainX, trainy, verbose=0)
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    # put the model back together
    model.pop()
    model.add(output_layer)
    model.compile(loss='mse', optimizer=SGD(lr=0.01, momentum=0.9))
    return train_acc, test_acc

# add one new layer and re-train only the new layer
def add_layer_to_autoencoder(model, trainX, testX):
    # remember the current output layer
    output_layer = model.layers[-1]
    # remove the output layer
    model.pop()
```

```

# mark all remaining layers as non-trainable
for layer in model.layers:
    layer.trainable = False
# add a new hidden layer
model.add(Dense(10, activation='relu', kernel_initializer='he_uniform'))
# re-add the output layer
model.add(output_layer)
# fit model
model.fit(trainX, trainX, epochs=100, verbose=0)
# evaluate reconstruction loss
train_mse = model.evaluate(trainX, trainX, verbose=0)
test_mse = model.evaluate(testX, testX, verbose=0)
print('> reconstruction error train=%f, test=%f' % (train_mse, test_mse))

# prepare data
trainX, testX, trainy, testy = prepare_data()
# get the base autoencoder
model = base_autoencoder(trainX, testX)
# evaluate the base model
scores = dict()
train_acc, test_acc = evaluate_autoencoder_as_classifier(model, trainX, trainy, testX,
    testy)
print('> classifier accuracy layers=%d, train=%f, test=%f' % (len(model.layers),
    train_acc, test_acc))
scores[len(model.layers)] = (train_acc, test_acc)
# add layers and evaluate the updated model
n_layers = 5
for _ in range(n_layers):
    # add layer
    add_layer_to_autoencoder(model, trainX, testX)
    # evaluate model
    train_acc, test_acc = evaluate_autoencoder_as_classifier(model, trainX, trainy, testX,
        testy)
    print('> classifier accuracy layers=%d, train=%f, test=%f' % (len(model.layers),
        train_acc, test_acc))
    # store scores for plotting
    scores[len(model.layers)] = (train_acc, test_acc)
# plot number of added layers vs accuracy
keys = list(scores.keys())
pyplot.plot(keys, [scores[k][0] for k in keys], label='train', marker='.')
pyplot.plot(keys, [scores[k][1] for k in keys], label='test', marker='.')
pyplot.legend()
pyplot.show()

```

Listing 10.34: Example of unsupervised pre-training on the blobs problem.

Running the example reports both reconstruction error and classification accuracy on the train and test sets for the model for the base model (two layers) then after each additional layer is added (from three to 12 layers).

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that reconstruction error starts low, in fact near-perfect, then slowly increases during training. Accuracy on the training dataset seems to decrease as layers are

added to the encoder, although accuracy test seems to improve as layers are added, at least until the model has five layers, after which performance appears to crash.

```
> reconstruction error train=0.000, test=0.000
> classifier accuracy layers=2, train=0.830, test=0.832
> reconstruction error train=0.001, test=0.002
> classifier accuracy layers=3, train=0.826, test=0.842
> reconstruction error train=0.002, test=0.002
> classifier accuracy layers=4, train=0.820, test=0.838
> reconstruction error train=0.016, test=0.028
> classifier accuracy layers=5, train=0.828, test=0.834
> reconstruction error train=2.311, test=2.694
> classifier accuracy layers=6, train=0.764, test=0.762
> reconstruction error train=2.192, test=2.526
> classifier accuracy layers=7, train=0.764, test=0.760
```

Listing 10.35: Example output from unsupervised pre-training on the blobs problem.

A line plot is also created showing the train (blue) and test set (orange) accuracy as each additional layer is added to the model. In this case, the plot suggests there may be some minor benefits in the unsupervised greedy layer-wise pretraining, but perhaps beyond five layers the model becomes unstable.

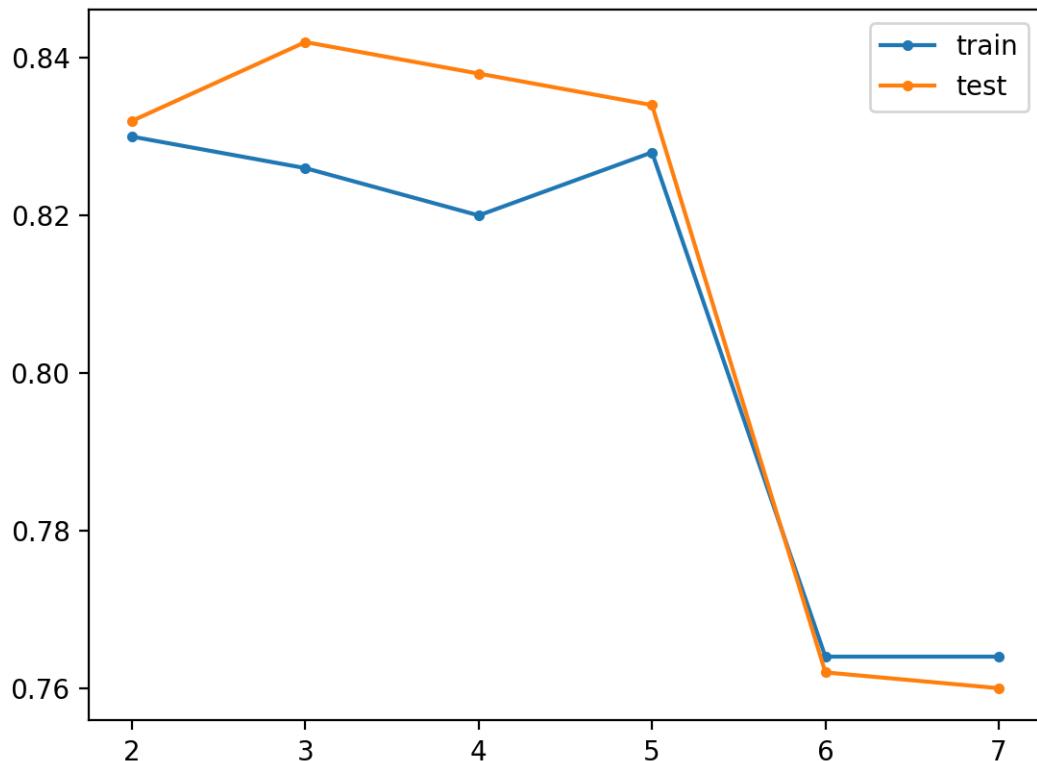


Figure 10.3: Line Plot for Unsupervised Greedy Layer-Wise Pretraining Showing Model Layers vs Train and Test Set Classification Accuracy on the Blobs Classification Problem.

An interesting extension would be to explore whether fine tuning of all weights in the model prior or after fitting a classifier output layer improves performance.

10.3 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Fine Tuning.** Update the example to use a smaller learning rate and fine-tune the pre-trained layers and compare performance.
- **Improve Autoencoder.** Update the example to monitor loss on the autoencoder and tune the model to further reduce the loss.

If you explore any of these extensions, I'd love to know.

10.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

10.4.1 Books

- Section 8.7.4: Supervised Pretraining, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- Section 15.1: Greedy Layer-Wise Unsupervised Pretraining, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>

10.4.2 APIs

- Getting started with the Keras Sequential model.
<https://keras.io/getting-started/sequential-model-guide/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- `sklearn.datasets.make_blobs` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

10.4.3 Papers

- *Greedy Layer-Wise Training of Deep Networks*, 2007.
<https://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks>
- *Why Does Unsupervised Pre-training Help Deep Learning*, 2010.
<http://www.jmlr.org/papers/v11/erhan10a.html>

10.5 Summary

In this tutorial, you discovered greedy layer-wise pretraining as a technique for developing deep multilayered neural network models. Specifically, you learned:

- Greedy layer-wise pretraining provides a way to develop deep multilayered neural networks whilst only ever training shallow networks.
- Pretraining can be used to iteratively deepen a supervised model or an unsupervised model that can be repurposed as a supervised model.
- Pretraining may be useful for problems with small amounts labeled data and large amounts of unlabeled data.

10.5.1 Next

In the next tutorial, you will discover how to reuse models trained on related but different problems called transfer learning for feature extraction and weight initialization.

Chapter 11

Jump-Start Training with Transfer Learning

An interesting benefit of deep learning neural networks is that they can be reused on related problems. Transfer learning refers to a technique for predictive modeling on a different but somehow similar problem that can then be reused partly or wholly to accelerate the training and improve the performance of a model on the problem of interest. In deep learning, this means reusing the weights in one or more layers from a pre-trained network model in a new model and either keeping the weights fixed, fine tuning them, or adapting the weights entirely when training the model. In this tutorial, you will discover how to use transfer learning to improve the performance of deep learning neural networks in Python with Keras. After completing this tutorial, you will know:

- Transfer learning is a method for reusing a model trained on a related predictive modeling problem.
- Transfer learning can be used to accelerate the training of neural networks as either a weight initialization scheme or feature extraction method.
- How to use transfer learning to improve the performance of an MLP for a multiclass classification problem.

Let's get started.

11.1 Transfer Learning

Transfer learning generally refers to a process where a model trained on one problem is used in some way on a second related problem.

Transfer learning and domain adaptation refer to the situation where what has been learned in one setting (i.e., distribution P1) is exploited to improve generalization in another setting (say distribution P2).

— Page 536, *Deep Learning*, 2016.

In deep learning, transfer learning is a technique whereby a neural network model is first trained on a problem similar to the problem that is being solved. One or more layers from the trained model are then used in a new model trained on the problem of interest.

This is typically understood in a supervised learning context, where the input is the same but the target may be of a different nature. For example, we may learn about one set of visual categories, such as cats and dogs, in the first setting, then learn about a different set of visual categories, such as ants and wasps, in the second setting.

— Page 536, *Deep Learning*, 2016.

Transfer learning has the benefit of decreasing the training time for a neural network model, resulting in lower generalization error. There are two main approaches to implementing transfer learning; they are:

- Weight Initialization.
- Feature Extraction.

The weights in re-used layers may be used as the starting point for the training process and adapted in response to the new problem. This usage treats transfer learning as a type of weight initialization scheme. This may be useful when the first related problem has a lot more labeled data than the problem of interest and the similarity in the structure of the problem may be useful in both contexts.

... the objective is to take advantage of data from the first setting to extract information that may be useful when learning or even when directly making predictions in the second setting.

— Page 538, *Deep Learning*, 2016.

Alternately, the weights of the network may not be adapted in response to the new problem, and only new layers after the reused layers may be trained to interpret their output. This usage treats transfer learning as a type of feature extraction scheme. An example of this approach is the re-use of deep convolutional neural network models trained for photo classification as feature extractors when developing photo captioning models. Variations on these usages may involve not training the weights of the model on the new problem initially, but later fine tuning all weights of the learned model with a small learning rate.

11.2 Transfer Learning Case Study

In this section, we will demonstrate how to use transfer learning to develop MLP models on a multiclass classification problem. This example provides a template for applying transfer learning to your own neural network for classification and regression problems.

11.2.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate transfer learning. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. We can configure the problem to have two input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same data points.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=1)
```

Listing 11.1: Example of a generating samples for the blobs problem 1.

The results are the input and output elements of a dataset that we can model. The `random_state` argument can be varied to give different versions of the problem (different cluster centers). We can use this to generate samples from two different problems: train a model on one problem and re-use the weights to better learn a model for a second problem. Specifically, we will refer to `random_state=1` as Problem 1 and `random_state=2` as Problem 2.

- **Problem 1.** Blobs problem with two input variables and three classes with the `random_state` argument set to one.
- **Problem 2.** Blobs problem with two input variables and three classes with the `random_state` argument set to two.

In order to get a feeling for the complexity of the problem, we can plot each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# plot of blobs multiclass classification problems 1 and 2
from sklearn.datasets import make_blobs
from numpy import where
from matplotlib import pyplot

# generate samples for blobs problem with a given random seed
def samples_for_seed(seed):
    # generate samples
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2,
                      random_state=seed)
    return X, y

# create a scatter plot of points colored by class value
def plot_samples(X, y, classes=3):
    # plot points for each class
    for i in range(classes):
        # select indices of points with each class label
        samples_ix = where(y == i)
        # plot points for this class with a given color
        pyplot.scatter(X[samples_ix, 0], X[samples_ix, 1])

# generate multiple problems
n_problems = 2
```

```

for i in range(1, n_problems+1):
    # specify subplot
    pyplot.subplot(210 + i)
    # generate samples
    X, y = samples_for_seed(i)
    # scatter plot of samples
    plot_samples(X, y)
# plot figure
pyplot.show()

```

Listing 11.2: Example of plotting samples from the blobs problems 1 and 2.

Running the example generates a sample of 1,000 examples for Problem 1 and Problem 2 and creates a scatter plot for each sample, coloring the data points by their class value.

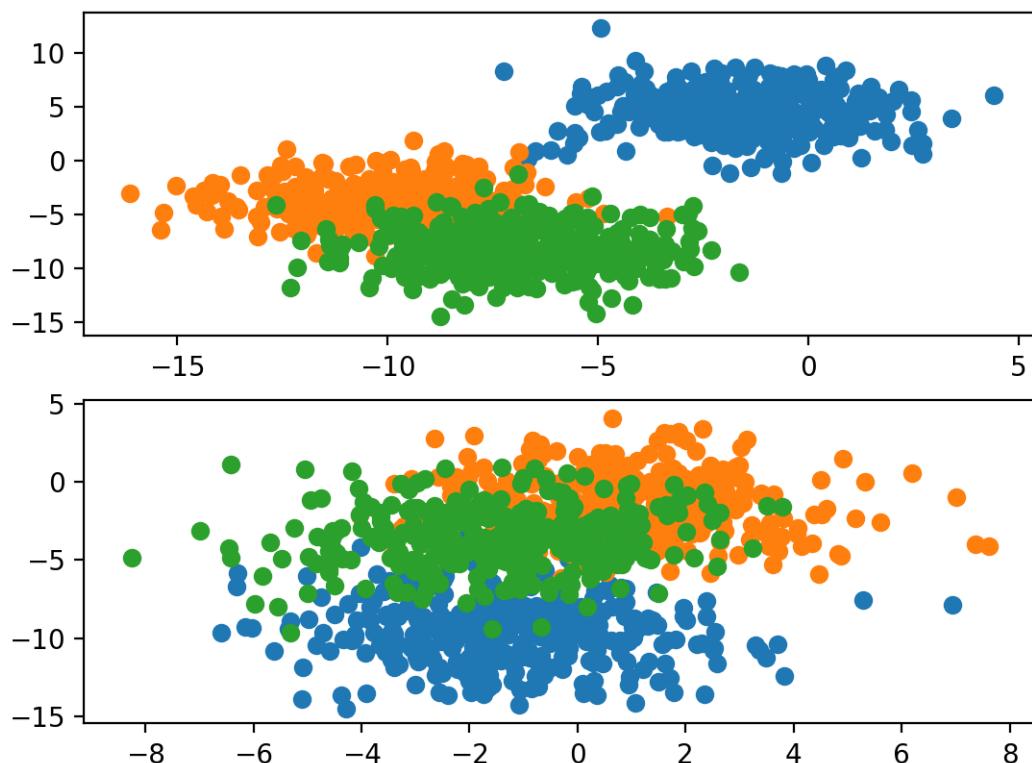


Figure 11.1: Scatter Plots of Blobs Dataset for Problems 1 and 2 With Three Classes and Points Colored by Class Value.

This provides a good basis for transfer learning as each version of the problem has similar input data with a similar scale, although with different target information (e.g. cluster centers). We would expect that aspects of a model fit on one version of the blobs problem (e.g. Problem 1) to be useful when fitting a model on a new version of the blobs problem (e.g. Problem 2).

11.2.2 Multilayer Perceptron Model for Problem 1

In this section, we will develop a Multilayer Perceptron model (MLP) for Problem 1 and save the model to file so that we can reuse the weights later. First, we will develop a function to prepare the dataset for modeling. After the `make_blobs()` function is called with a given random seed (e.g, one in this case for Problem 1), the target variable must be one hot encoded so that we can develop a model that predicts the probability of a given sample belonging to each of the target classes. The prepared samples can then be split in half, with 500 examples for both the train and test datasets. The `samples_for_seed()` function below implements this, preparing the dataset for a given random number generator seed and retuning the train and test sets split into input and output components.

```
# prepare a blobs examples with a given random seed
def samples_for_seed(seed):
    # generate samples
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2,
                       random_state=seed)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy
```

Listing 11.3: Example of generating a dataset for a blobs problem.

We can call this function to prepare a dataset for Problem 1 as follows.

```
# prepare data
trainX, trainy, testX, testy = samples_for_seed(1)
```

Listing 11.4: Example of generating a dataset for Problem 1.

Next, we can define and fit a model on the training dataset. The model will expect two inputs for the two variables in the data. The model will have two hidden layers with five nodes each and the rectified linear activation function. Two layers are probably not required for this function, although we're interested in the model learning some deep structure that we can reuse across instances of this problem. The output layer has three nodes, one for each class in the target variable and the softmax activation function.

```
# define model
model = Sequential()
model.add(Dense(5, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(3, activation='softmax'))
```

Listing 11.5: Example of defining the MLP model.

Given that the problem is a multiclass classification problem, the categorical cross-entropy loss function is minimized and the stochastic gradient descent with the default learning rate and no momentum is used to learn the problem.

```
# compile model
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

Listing 11.6: Example of compiling the MLP model.

The model is fit for 100 epochs on the training dataset and the test set is used as a validation dataset during training, evaluating the performance on both datasets at the end of each epoch so that we can plot learning curves.

```
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
```

Listing 11.7: Example of fitting the MLP model.

The `fit_model()` function ties these elements together, taking the train and test datasets as arguments and returning the fit model and training history.

```
# define and fit model on a training dataset
def fit_model(trainX, trainy, testX, testy):
    # define model
    model = Sequential()
    model.add(Dense(5, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
    return model, history
```

Listing 11.8: Example of defining a function to fit an MLP model.

We can call this function with the prepared dataset to obtain a fit model and the history collected during the training process.

```
# fit model on train dataset
model, history = fit_model(trainX, trainy, testX, testy)
```

Listing 11.9: Example of calling the function to fit an MLP model.

Finally, we can summarize the performance of the model. The classification accuracy of the model on the train and test sets can be evaluated.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 11.10: Example of evaluating a fit model.

The history collected during training can be used to create line plots showing both the loss and classification accuracy for the model on the train and test sets over each training epoch, providing learning curves.

```
# plot loss during training
pyplot.subplot(211)
pyplot.title('Loss')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy during training
pyplot.subplot(212)
pyplot.title('Accuracy')
pyplot.plot(history.history['accuracy'], label='train')
```

```
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 11.11: Example of plotting learning curves for the fit model.

The `summarize_model()` function below implements this, taking the fit model, training history, and dataset as arguments and printing the model performance and creating a plot of model learning curves.

```
# summarize the performance of the fit model
def summarize_model(model, history, trainX, trainy, testX, testy):
    # evaluate the model
    _, train_acc = model.evaluate(trainX, trainy, verbose=0)
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
    # plot loss during training
    pyplot.subplot(211)
    pyplot.title('Loss')
    pyplot.plot(history.history['loss'], label='train')
    pyplot.plot(history.history['val_loss'], label='test')
    pyplot.legend()
    # plot accuracy during training
    pyplot.subplot(212)
    pyplot.title('Accuracy')
    pyplot.plot(history.history['accuracy'], label='train')
    pyplot.plot(history.history['val_accuracy'], label='test')
    pyplot.legend()
    pyplot.show()
```

Listing 11.12: Example of a function for evaluating a fit model.

We can call this function with the fit model and prepared data.

```
# evaluate model behavior
summarize_model(model, history, trainX, trainy, testX, testy)
```

Listing 11.13: Example of calling the function for evaluating a fit model.

At the end of the run, we can save the model to file so that we may load it later and use it as the basis for some transfer learning experiments. Note that saving the model to file requires that you have the `h5py` library installed. This library can be installed via `pip` as follows:

```
sudo pip install h5py
```

Listing 11.14: Example installing the `h5py` library with `pip`.

The fit model can be saved by calling the `save()` function on the model.

```
# save model to file
model.save('model.h5')
```

Listing 11.15: Example of saving the fit model.

Tying these elements together, the complete example of fitting an MLP on Problem 1, summarizing the model's performance, and saving the model to file is listed below.

```
# fit mlp model on problem 1 and save model to file
from sklearn.datasets import make_blobs
```

```
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare a blobs examples with a given random seed
def samples_for_seed(seed):
    # generate samples
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2,
        random_state=seed)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy

# define and fit model on a training dataset
def fit_model(trainX, trainy, testX, testy):
    # define model
    model = Sequential()
    model.add(Dense(5, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
    return model, history

# summarize the performance of the fit model
def summarize_model(model, history, trainX, trainy, testX, testy):
    # evaluate the model
    _, train_acc = model.evaluate(trainX, trainy, verbose=0)
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
    # plot loss during training
    pyplot.subplot(211)
    pyplot.title('Loss')
    pyplot.plot(history.history['loss'], label='train')
    pyplot.plot(history.history['val_loss'], label='test')
    pyplot.legend()
    # plot accuracy during training
    pyplot.subplot(212)
    pyplot.title('Accuracy')
    pyplot.plot(history.history['accuracy'], label='train')
    pyplot.plot(history.history['val_accuracy'], label='test')
    pyplot.legend()
    pyplot.show()

# prepare data
trainX, trainy, testX, testy = samples_for_seed(1)
# fit model on train dataset
model, history = fit_model(trainX, trainy, testX, testy)
# evaluate model behavior
```

```
summarize_model(model, history, trainX, trainy, testX, testy)
# save model to file
model.save('model.h5')
```

Listing 11.16: Example of fitting and saving an MLP model on Problem 1.

Running the example fits and evaluates the performance of the model, printing the classification accuracy on the train and test sets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model performed well on Problem 1, achieving a classification accuracy of about 92% on both the train and test datasets.

```
Train: 0.916, Test: 0.920
```

Listing 11.17: Example output from fitting an MLP on Problem 1.

A figure is also created summarizing the learning curves of the model, showing both the loss (top) and accuracy (bottom) for the model on both the train (blue) and test (orange) datasets at the end of each training epoch. In this case, we can see that the model learned the problem reasonably quickly and well, perhaps converging in about 40 epochs and remaining reasonably stable on both datasets.

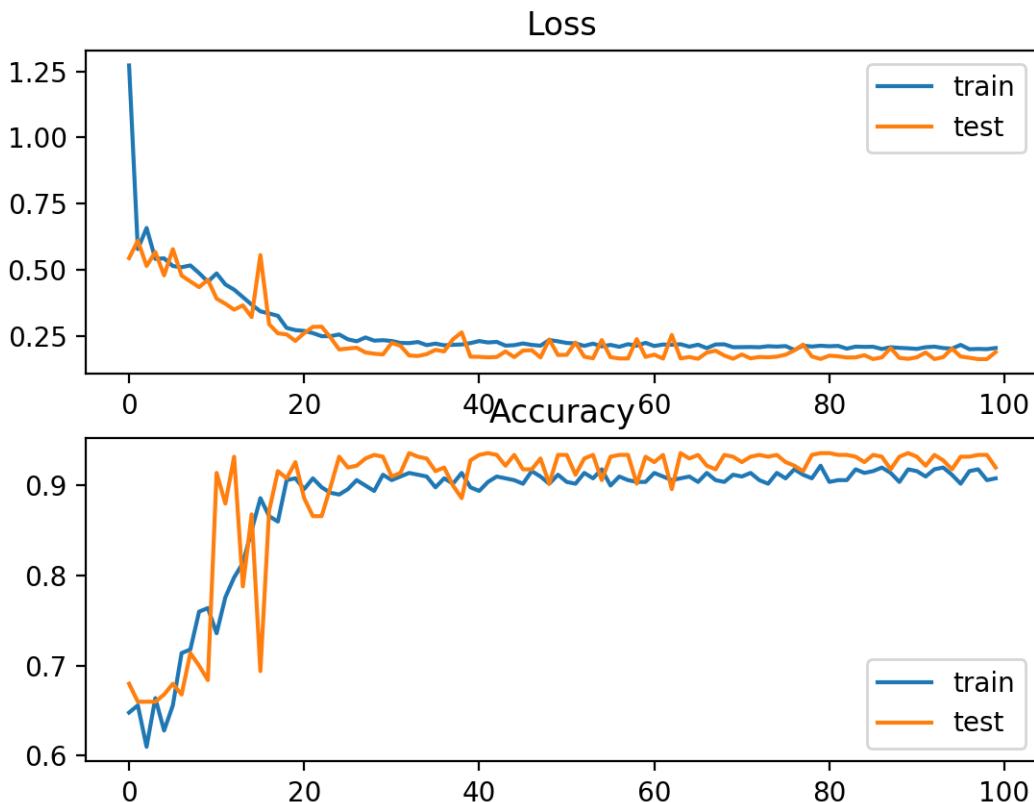


Figure 11.2: Loss and Accuracy Learning Curves on the Train and Test Sets for an MLP on Problem 1.

Now that we have seen how to develop a standalone MLP for the blobs Problem 1, we can look at the doing the same for Problem 2 that can be used as a baseline.

11.2.3 Standalone MLP Model for Problem 2

The example in the previous section can be updated to fit an MLP model to Problem 2. It is important to get an idea of performance and learning dynamics on Problem 2 for a standalone model first as this will provide a baseline in performance that can be used to compare to a model fit on the same problem using transfer learning. A single change is required that changes the call to `samples_for_seed()` to use the pseudorandom number generator seed of two instead of one.

```
# prepare data
trainX, trainy, testX, testy = samples_for_seed(2)
```

Listing 11.18: Example of preparing the dataset for Problem 2.

For completeness, the full example with this change is listed below.

```
# fit mlp model on problem 2 and save model to file
from sklearn.datasets import make_blobs
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from matplotlib import pyplot

# prepare a blobs examples with a given random seed
def samples_for_seed(seed):
    # generate samples
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2,
                       random_state=seed)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy

# define and fit model on a training dataset
def fit_model(trainX, trainy, testX, testy):
    # define model
    model = Sequential()
    model.add(Dense(5, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
    # fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
    return model, history

# summarize the performance of the fit model
def summarize_model(model, history, trainX, trainy, testX, testy):
    # evaluate the model
```

```

_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss during training
pyplot.subplot(211)
pyplot.title('Loss')
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy during training
pyplot.subplot(212)
pyplot.title('Accuracy')
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

# prepare data
trainX, trainy, testX, testy = samples_for_seed(2)
# fit model on train dataset
model, history = fit_model(trainX, trainy, testX, testy)
# evaluate model behavior
summarize_model(model, history, trainX, trainy, testX, testy)

```

Listing 11.19: Example of fitting and saving an MLP model on Problem 2.

Running the example fits and evaluates the performance of the model, printing the classification accuracy on the train and test sets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model performed okay on Problem 2, but not as well as was seen on Problem 1, achieving a classification accuracy of about 79% on both the train and test datasets.

Train: 0.794, Test: 0.794

Listing 11.20: Example output from fitting an MLP on Problem 2.

A figure is also created summarizing the learning curves of the model. In this case, we can see that the model converged more slowly than we saw on Problem 1 in the previous section. This suggests that this version of the problem may be slightly more challenging, at least for the chosen model configuration.

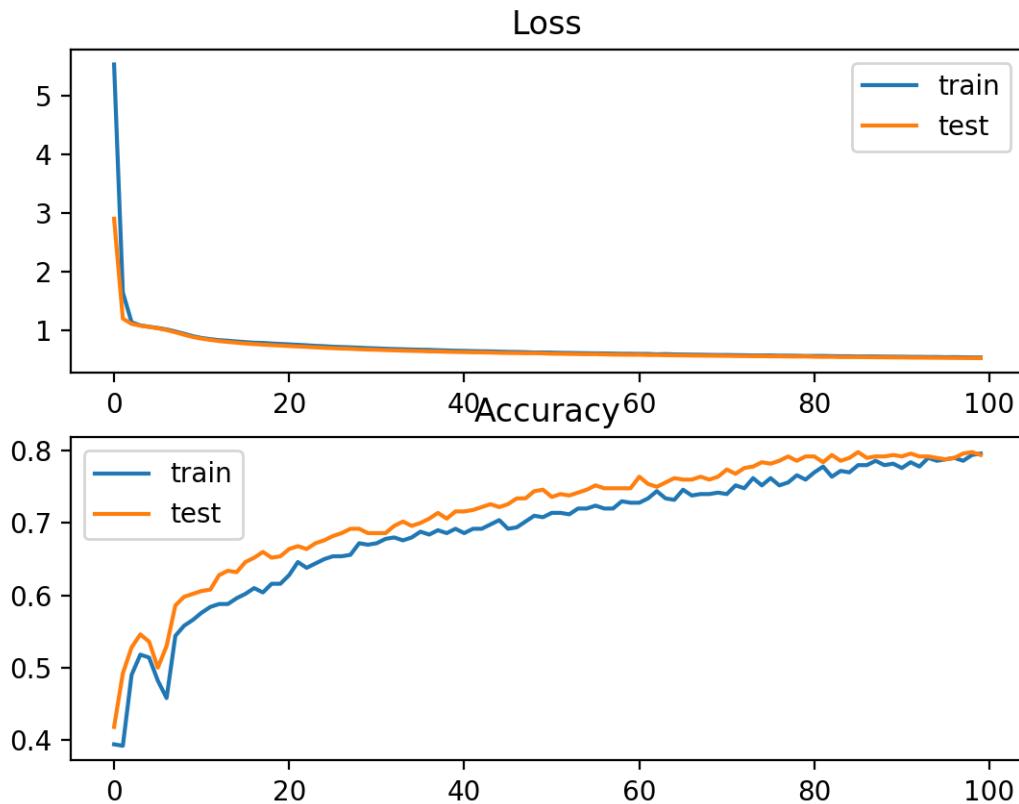


Figure 11.3: Loss and Accuracy Learning Curves on the Train and Test Sets for an MLP on Problem 2.

Now that we have a baseline of performance and learning dynamics for an MLP on Problem 2, we can see how the addition of transfer learning affects the MLP on this problem.

11.2.4 MLP With Transfer Learning for Problem 2

The model that was fit on Problem 1 can be loaded and the weights can be used as the initial weights for a model fit on Problem 2. This is a type of transfer learning where learning on a different but related problem is used as a type of weight initialization scheme. This requires that the `fit_model()` function be updated to load the model and refit it on examples for Problem 2. The model saved in `model.h5` can be loaded using the `load_model()` Keras function.

```
# load model
model = load_model('model.h5')
```

Listing 11.21: Example loading a saved model.

Once loaded, the model can be compiled and fit as per normal. The updated `fit_model()` with this change is listed below.

```
# load and re-fit model on a training dataset
def fit_model(trainX, trainy, testX, testy):
    # load model
```

```

model = load_model('model.h5')
# compile model
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
# re-fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
return model, history

```

Listing 11.22: Example a function for loading and re-fitting the model.

We would expect that a model that uses the weights from a model fit on a different but related problem to learn the problem perhaps faster in terms of the learning curve and perhaps result in lower generalization error, although these aspects would be dependent on the choice of problems and model. For completeness, the full example with this change is listed below.

```

# transfer learning with mlp model on problem 2
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import load_model
from matplotlib import pyplot

# prepare a blobs examples with a given random seed
def samples_for_seed(seed):
    # generate samples
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2,
        random_state=seed)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
    return trainX, trainy, testX, testy

# load and re-fit model on a training dataset
def fit_model(trainX, trainy, testX, testy):
    # load model
    model = load_model('model.h5')
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
    # re-fit model
    history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=100, verbose=0)
    return model, history

# summarize the performance of the fit model
def summarize_model(model, history, trainX, trainy, testX, testy):
    # evaluate the model
    _, train_acc = model.evaluate(trainX, trainy, verbose=0)
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
    # plot loss during training
    pyplot.subplot(211)
    pyplot.title('Loss')
    pyplot.plot(history.history['loss'], label='train')
    pyplot.plot(history.history['val_loss'], label='test')
    pyplot.legend()
    # plot accuracy during training

```

```
pyplot.subplot(212)
pyplot.title('Accuracy')
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

# prepare data
trainX, trainy, testX, testy = samples_for_seed(2)
# fit model on train dataset
model, history = fit_model(trainX, trainy, testX, testy)
# evaluate model behavior
summarize_model(model, history, trainX, trainy, testX, testy)
```

Listing 11.23: Example of transfer learning for Problem 2.

Running the example fits and evaluates the performance of the model, printing the classification accuracy on the train and test sets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved a lower generalization error, achieving an accuracy of about 81% on the test dataset for Problem 2 as compared to the standalone model that achieved about 79% accuracy.

```
Train: 0.786, Test: 0.810
```

Listing 11.24: Example output from transfer learning for Problem 2.

A figure is also created summarizing the learning curves of the model. In this case, we can see that the model does appear to have a similar learning curve, although we do see apparent improvements in the learning curve for the test set (orange line) both in terms of better performance earlier (epoch 20 onward) and above the performance of the model on the training set.

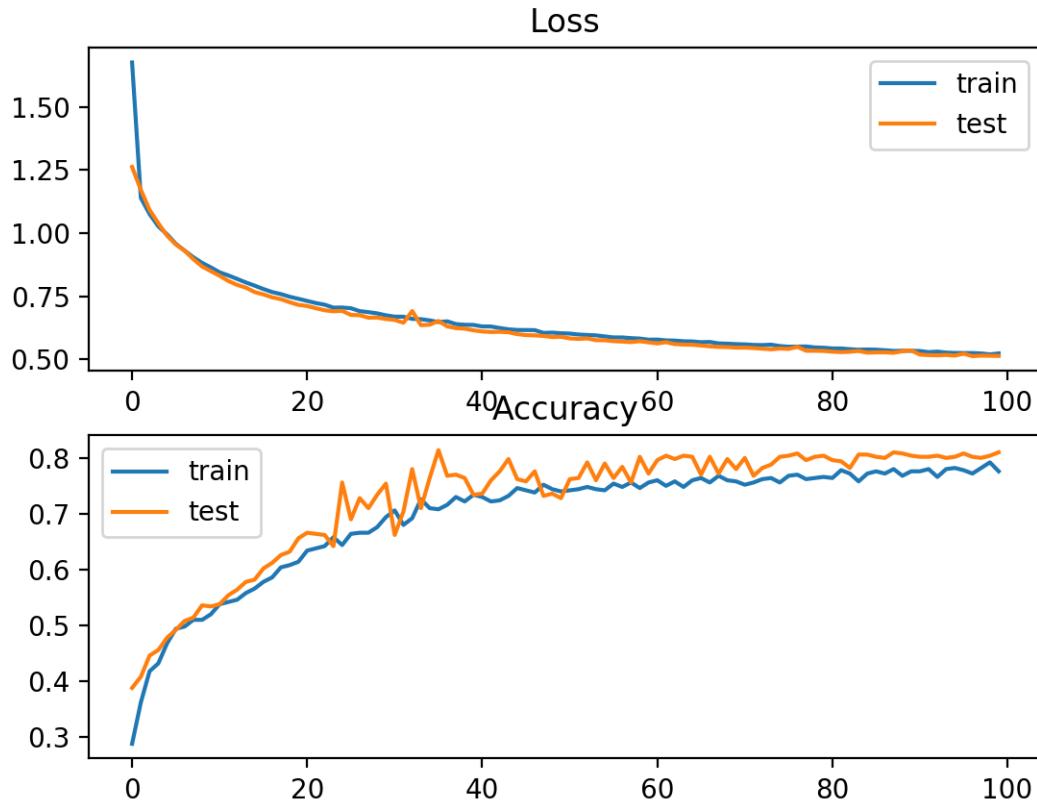


Figure 11.4: Loss and Accuracy Learning Curves on the Train and Test Sets for an MLP With Transfer Learning on Problem 2.

We have only looked at single runs of a standalone MLP model and an MLP with transfer learning. Neural network algorithms are stochastic, therefore an average of performance across multiple runs is required to see if the observed behavior is real or a statistical fluke.

11.2.5 Comparison of Models on Problem 2

In order to determine whether using transfer learning for the blobs multiclass classification problem has a real effect, we must repeat each experiment multiple times and analyze the average performance across the repeats. We will compare the performance of the standalone model trained on Problem 2 to a model using transfer learning, averaged over 30 repeats. Further, we will investigate whether keeping the weights in some of the layers fixed improves model performance. The model trained on Problem 1 has two hidden layers. By keeping the first or the first and second hidden layers fixed, the layers with unchangeable weights will act as a feature extractor and may provide features that make learning Problem 2 easier, affecting the speed of learning and/or the accuracy of the model on the test set. As the first step, we will simplify the `fit_model()` function to fit the model and discard any training history so that we can focus on the final accuracy of the trained model.

```
# define and fit model on a training dataset
```

```
def fit_model(trainX, trainy):
    # define model
    model = Sequential()
    model.add(Dense(5, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=100, verbose=0)
    return model
```

Listing 11.25: Example a function to fit and return a model.

Next, we can develop a function that will repeatedly fit a new standalone model on Problem 2 on the training dataset and evaluate accuracy on the test set. The `eval_standalone_model()` function below implements this, taking the train and test sets as arguments as well as the number of repeats and returns a list of accuracy scores for models on the test dataset.

```
# repeated evaluation of a standalone model
def eval_standalone_model(trainX, trainy, testX, testy, n_repeats):
    scores = list()
    for _ in range(n_repeats):
        # define and fit a new model on the train dataset
        model = fit_model(trainX, trainy)
        # evaluate model on test dataset
        _, test_acc = model.evaluate(testX, testy, verbose=0)
        scores.append(test_acc)
    return scores
```

Listing 11.26: Example a function for the repeated evaluation of a model.

Summarizing the distribution of accuracy scores returned from this function will give an idea of how well the chosen standalone model performs on Problem 2.

```
# repeated evaluation of standalone model
standalone_scores = eval_standalone_model(trainX, trainy, testX, testy, n_repeats)
print('Standalone %.3f (%.3f)' % (mean(standalone_scores), std(standalone_scores)))
```

Listing 11.27: Example a of repeated evaluation of a model.

Next, we need an equivalent function for evaluating a model using transfer learning. In each loop, the model trained on Problem 1 must be loaded from file, fit on the training dataset for Problem 2, then evaluated on the test set for Problem 2. In addition, we will configure 0, 1, or 2 of the hidden layers in the loaded model to remain fixed. Keeping 0 hidden layers fixed means that all of the weights in the model will be adapted when learning Problem 2, using transfer learning as a weight initialization scheme. Whereas, keeping both (2) of the hidden layers fixed means that only the output layer of the model will be adapted during training, using transfer learning as a feature extraction method.

The `eval_transfer_model()` function below implements this, taking the train and test datasets for Problem 2 as arguments as well as the number of hidden layers in the loaded model to keep fixed and the number of times to repeat the experiment. The function returns a list of test accuracy scores and summarizing this distribution will give a reasonable idea of how well the model with the chosen type of transfer learning performs on Problem 2.

```
# repeated evaluation of a model with transfer learning
def eval_transfer_model(trainX, trainy, testX, testy, n_fixed, n_repeats):
    scores = list()
    for _ in range(n_repeats):
        # load model
        model = load_model('model.h5')
        # mark layer weights as fixed or not trainable
        for i in range(n_fixed):
            model.layers[i].trainable = False
        # re-compile model
        model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
        # fit model on train dataset
        model.fit(trainX, trainy, epochs=100, verbose=0)
        # evaluate model on test dataset
        _, test_acc = model.evaluate(testX, testy, verbose=0)
        scores.append(test_acc)
    return scores
```

Listing 11.28: Example a of a function for repeated evaluation of a transfer learning model.

We can call this function repeatedly, setting `n_fixed` to 0, 1, 2 in a loop and summarizing performance as we go; for example:

```
# repeated evaluation of transfer learning model, vary fixed layers
n_fixed = 3
for i in range(n_fixed):
    scores = eval_transfer_model(trainX, trainy, testX, testy, i, n_repeats)
    print('Transfer (fixed=%d) %.3f (%.3f)' % (i, mean(scores), std(scores)))
```

Listing 11.29: Example a reporting performance from evaluating transfer learning models.

In addition to reporting the mean and standard deviation of each model, we can collect all scores and create a box and whisker plot to summarize and compare the distributions of model scores. Tying all of the these elements together, the complete example is listed below.

```
# compare standalone mlp model performance to transfer learning
from sklearn.datasets import make_blobs
from keras.layers import Dense
from keras.models import Sequential
from keras.utils import to_categorical
from keras.models import load_model
from matplotlib import pyplot
from numpy import mean
from numpy import std

# prepare a blobs examples with a given random seed
def samples_for_seed(seed):
    # generate samples
    X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2,
                      random_state=seed)
    # one hot encode output variable
    y = to_categorical(y)
    # split into train and test
    n_train = 500
    trainX, testX = X[:n_train, :], X[n_train:, :]
    trainy, testy = y[:n_train], y[n_train:]
```

```
    return trainX, trainy, testX, testy

# define and fit model on a training dataset
def fit_model(trainX, trainy):
    # define model
    model = Sequential()
    model.add(Dense(5, input_dim=2, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(5, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(3, activation='softmax'))
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=100, verbose=0)
    return model

# repeated evaluation of a standalone model
def eval_standalone_model(trainX, trainy, testX, testy, n_repeats):
    scores = list()
    for _ in range(n_repeats):
        # define and fit a new model on the train dataset
        model = fit_model(trainX, trainy)
        # evaluate model on test dataset
        _, test_acc = model.evaluate(testX, testy, verbose=0)
        scores.append(test_acc)
    return scores

# repeated evaluation of a model with transfer learning
def eval_transfer_model(trainX, trainy, testX, testy, n_fixed, n_repeats):
    scores = list()
    for _ in range(n_repeats):
        # load model
        model = load_model('model.h5')
        # mark layer weights as fixed or not trainable
        for i in range(n_fixed):
            model.layers[i].trainable = False
        # re-compile model
        model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
        # fit model on train dataset
        model.fit(trainX, trainy, epochs=100, verbose=0)
        # evaluate model on test dataset
        _, test_acc = model.evaluate(testX, testy, verbose=0)
        scores.append(test_acc)
    return scores

# prepare data for problem 2
trainX, trainy, testX, testy = samples_for_seed(2)
n_repeats = 30
dists, dist_labels = list(), list()

# repeated evaluation of standalone model
standalone_scores = eval_standalone_model(trainX, trainy, testX, testy, n_repeats)
print('Standalone %.3f (%.3f)' % (mean(standalone_scores), std(standalone_scores)))
dists.append(standalone_scores)
dist_labels.append('standalone')

# repeated evaluation of transfer learning model, vary fixed layers
```

```

n_fixed = 3
for i in range(n_fixed):
    scores = eval_transfer_model(trainX, trainy, testX, testy, i, n_repeats)
    print('Transfer (fixed=%d) %.3f (%.3f)' % (i, mean(scores), std(scores)))
    dists.append(scores)
    dist_labels.append('transfer f=' + str(i))

# box and whisker plot of score distributions
pyplot.boxplot(dists, labels=dist_labels)
pyplot.show()

```

Listing 11.30: Example of evaluating different configurations for transfer learning.

Running the example first reports the mean and standard deviation of classification accuracy on the test dataset for each model.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the standalone model achieved an accuracy of about 78% on Problem 2 with a large standard deviation of 10%. In contrast, we can see that the spread of all of the transfer learning models is much smaller, ranging from about 0.05% to 1.5%.

This difference in the standard deviations of the test accuracy scores shows the stability that transfer learning can bring to the model, reducing the variance in the performance of the final model introduced via the stochastic learning algorithm. Comparing the mean test accuracy of the models, we can see that transfer learning that used the model as a weight initialization scheme (`fixed=0`) resulted in better performance than the standalone model with about 80% accuracy. Keeping all hidden layers fixed (`fixed=2`) and using them as a feature extraction scheme resulted in worse performance on average than the standalone model. It suggests that the approach is too restrictive in this case.

Interestingly, we see best performance when the first hidden layer is kept fixed (`fixed=1`) and the second hidden layer is adapted to the problem with a test classification accuracy of about 81%. This suggests that in this case, the problem benefits from both the feature extraction and weight initialization properties of transfer learning. It may be interesting to see how results of this last approach compare to the same model where the weights of the second hidden layer (and perhaps the output layer) are re-initialized with random numbers. This comparison would demonstrate whether the feature extraction properties of transfer learning alone or both feature extraction and weight initialization properties are beneficial.

```

Standalone 0.787 (0.101)
Transfer (fixed=0) 0.805 (0.004)
Transfer (fixed=1) 0.817 (0.005)
Transfer (fixed=2) 0.750 (0.014)

```

Listing 11.31: Example output from evaluating different configurations for transfer learning.

A figure is created showing four box and whisker plots. The box shows the middle 50% of each data distribution, the orange line shows the median, and the dots show outliers. The box and whisker plot for the standalone model shows a number of outliers, indicating that on average, the model performs well, but there is a chance that it can perform very poorly. Conversely, we see that the behavior of the models with transfer learning are more stable, showing a tighter distribution in performance.

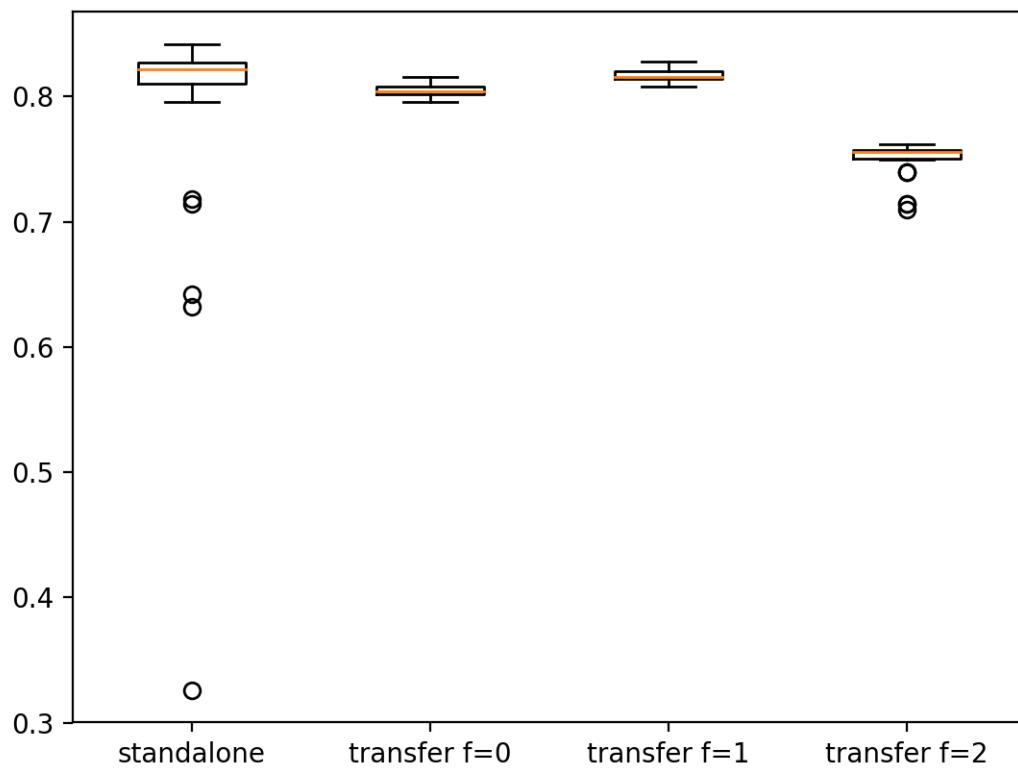


Figure 11.5: Box and Whisker Plot Comparing Standalone and Transfer Learning Models via Test Set Accuracy on the Blobs Multiclass Classification Problem.

11.3 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Reverse Experiment.** Train and save a model for Problem 2 and see if it can help when using it for transfer learning on Problem 1.
- **Add Hidden Layer.** Update the example to keep both hidden layers fixed, but add a new hidden layer with randomly initialized weights after the fixed layers before the output layer and compare performance.
- **Randomly Initialize Layers.** Update the example to randomly initialize the weights of the second hidden layer and the output layer and compare performance.

If you explore any of these extensions, I'd love to know.

11.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

11.4.1 Books

- Section 5.2: Transfer Learning and Domain Adaptation, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>

11.4.2 Papers

- *Deep Learning of Representations for Unsupervised and Transfer Learning*, 2011.
<http://proceedings.mlr.press/v27/bengio12a.html>
- *Domain Adaptation for Large-Scale Sentiment Classification: A Deep Learning Approach*, 2011.
<https://dl.acm.org/citation.cfm?id=3104547>
- *Is Learning The n-th Thing Any Easier Than Learning The First?*, 1996.
<http://papers.nips.cc/paper/1034-is-learning-the-n-th-thing-any-easier-than-learning-the-first.pdf>

11.4.3 APIs

- Getting started with the Keras Sequential model.
<https://keras.io/getting-started/sequential-model-guide/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- `sklearn.datasets.make_blobs` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

11.4.4 Articles

- Transfer learning, Wikipedia.
https://en.wikipedia.org/wiki/Transfer_learning

11.5 Summary

In this tutorial, you discovered how to use transfer learning to improve the performance of deep learning neural networks in Python with Keras. Specifically, you learned:

- Transfer learning is a method for reusing a model trained on a related predictive modeling problem.
- Transfer learning can be used to accelerate the training of neural networks as either a weight initialization scheme or feature extraction method.
- How to use transfer learning to improve the performance of an MLP for a multiclass classification problem.

11.5.1 Next

This was the last tutorial in this Part on learning. In the next part, you will discover tutorials for reducing the overfitting and improve the generalization of neural network models.

Part II

Better Generalization

Overview

In this part you will discover techniques that you can use to reduce overfitting and improve the generalization of your deep learning neural network models. After reading the chapters in this part, you will know:

- How techniques that reduce model complexity have a regularizing effect resulting in less overfitting and better generalization ([Chapter 12](#)).
- How to add a penalty to the loss function to encourage smaller model weights ([Chapter 13](#)).
- How to add a penalty to the loss function to encourage sparse internal representations ([Chapter 14](#)).
- How to add a constraint to the model to force small model weights and lower complexity models ([Chapter 15](#)).
- How to add dropout weights during training to decouple model layers ([Chapter 16](#)).
- How to add noise to the training process to promote model robustness ([Chapter 17](#)).
- How to use early stopping to halt model training at the right time ([Chapter 18](#)).

Chapter 12

Fix Overfitting with Regularization

Training a deep neural network that can generalize well to new data is a challenging problem. A model with too little capacity cannot learn the problem, whereas a model with too much capacity can learn it too well and overfit the training dataset. Both cases result in a model that does not generalize well. A modern approach to reducing generalization error is to use a larger model that may be required to use regularization during training that keeps the weights of the model small. These techniques not only reduce overfitting, but they can also lead to faster optimization of the model and better overall performance. In this tutorial, you will discover the problem of overfitting when training neural networks and how it can be addressed with regularization methods. After reading this tutorial, you will know:

- Underfitting can easily be addressed by increasing the capacity of the network, but overfitting requires the use of specialized techniques.
- Regularization methods like weight decay provide an easy way to control overfitting for large neural network models.
- A modern recommendation for regularization is to use early stopping with dropout and a weight constraint.

Let's get started.

12.1 Problem of Model Generalization and Overfitting

The objective of a neural network is to have a final model that performs well both on the data that we used to train it (e.g. the training dataset) and the new data on which the model will be used to make predictions.

The central challenge in machine learning is that we must perform well on new, previously unseen inputs - not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called generalization.

— Page 110, *Deep Learning*, 2016.

We require that the model learn from known examples and generalize from those known examples to new examples in the future. We use methods like a train/test split or k -fold cross-validation to estimate the ability of the model to generalize to new data. Learning and also generalizing to new cases is hard. Too little learning and the model will perform poorly on the training dataset and on new data. The model will underfit the problem. Too much learning and the model will perform well on the training dataset and poorly on new data, the model will overfit the problem. In both cases, the model has not generalized.

- **Underfit Model.** A model that fails to sufficiently learn the problem and performs poorly on a training dataset and does not perform well on a holdout sample.
- **Overfit Model.** A model that learns the training dataset too well, performing well on the training dataset but does not perform well on a hold out sample.
- **Good Fit Model.** A model that suitably learns the training dataset and generalizes well to the hold out dataset.

A model fit can be considered in the context of the bias-variance trade-off. An underfit model has high bias and low variance. Regardless of the specific samples in the training data, it cannot learn the problem. An overfit model has low bias and high variance. The model learns the training data too well and performance varies widely with new unseen examples or even statistical noise added to examples in the training dataset.

In order to generalize well, a system needs to be sufficiently powerful to approximate the target function. If it is too simple to fit even the training data then generalization to new data is also likely to be poor. [...] An overly complex system, however, may be able to approximate the data in many different ways that give similar errors and is unlikely to choose the one that will generalize best ...

— Page 241, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

We can address underfitting by increasing the capacity of the model. Capacity refers to the ability of a model to fit a variety of functions; more capacity, means that a model can fit more types of functions for mapping inputs to outputs. Increasing the capacity of a model is easily achieved by changing the structure of the model, such as adding more layers and/or more nodes to layers. Because an underfit model is so easily addressed, it is more common to have an overfit model. An overfit model is easily diagnosed by monitoring the performance of the model during training by evaluating it on both a training dataset and on a holdout validation dataset. Graphing line plots of the performance of the model during training, called learning curves, will show a familiar pattern.

For example, line plots of the loss (that we seek to minimize) of the model on train and validation datasets will show a line for the training dataset that drops and may plateau and a line for the validation dataset that drops at first, then at some point begins to rise again.

As training progresses, the generalization error may decrease to a minimum and then increase again as the network adapts to idiosyncrasies of the training data.

— Page 250, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

A learning curve plot tells the story of the model learning the problem until a point at which it begins overfitting and its ability to generalize to the unseen validation dataset begins to get worse.

12.2 Reduce Overfitting by Constraining Complexity

There are two ways to approach an overfit model:

1. Reduce overfitting by training the network on more examples.
2. Reduce overfitting by changing the complexity of the network.

A benefit of very deep neural networks is that their performance continues to improve as they are fed larger and larger datasets. A model with a near-infinite number of examples will eventually plateau in terms of what the capacity of the network is capable of learning. A model can overfit a training dataset because it has sufficient capacity to do so. Reducing the capacity of the model reduces the likelihood of the model overfitting the training dataset, to a point where it no longer overfits. The capacity of a neural network model, it's complexity, is defined by both it's structure in terms of nodes and layers and the parameters in terms of its weights. Therefore, we can reduce the complexity of a neural network to reduce overfitting in one of two ways:

- Change network complexity by changing the network structure (number of weights).
- Change network complexity by changing the network parameters (values of weights).

In the case of neural networks, the complexity can be varied by changing the number of adaptive parameters in the network. This is called structural stabilization. [...] The second principal approach to controlling the complexity of a model is through the use of regularization which involves the addition of a penalty term to the error function.

— Page 332, *Neural Networks for Pattern Recognition*, 1995.

For example, the structure could be tuned such as via grid search until a suitable number of nodes and/or layers is found to reduce or remove overfitting for the problem. Alternately, the model could be overfit and pruned by removing nodes until it achieves suitable performance on a validation dataset. It is more common to instead constrain the complexity of the model by ensuring the parameters (weights) of the model remain small. Small parameters suggest a less complex and, in turn, more stable model that is less sensitive to statistical fluctuations in the input data.

Large weights tend to cause sharp transitions in the [activation] functions and thus large changes in output for small changes in inputs.

— Page 269, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

It is more common to focus on methods that constrain the size of the weights in a neural network because a single network structure can be defined that is under-constrained, e.g. has a much larger capacity than is required for the problem, and regularization can be used during training to ensure that the model does not overfit. In such cases, performance can even be better as the additional capacity can be focused on better learning generalizable concepts in the problem. Techniques that seek to reduce overfitting (reduce generalization error) by keeping network weights small are referred to as regularization methods. More specifically, regularization refers to a class of approaches that add additional information to transform an ill-posed problem into a more stable well-posed problem.

A problem is said to be ill-posed if small changes in the given information cause large changes in the solution. This instability with respect to the data makes solutions unreliable because small measurement errors or uncertainties in parameters may be greatly magnified and lead to wildly different responses. [...] The idea behind regularization is to use supplementary information to restate an ill-posed problem in a stable form.

— Page 266, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

Regularization methods are so widely used to reduce overfitting that the term *regularization* may be used for any method that improves the generalization error of a neural network model.

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error. Regularization is one of the central concerns of the field of machine learning, rivaled in its importance only by optimization.

— Page 120, *Deep Learning*, 2016.

12.3 Regularization Methods for Neural Networks

The simplest and perhaps most common regularization method is to add a penalty to the loss function in proportion to the size of the weights in the model.

- **Weight Regularization:** Penalize the model during training based on the magnitude of the weights (Chapter 13).

This will encourage the model to map the inputs to the outputs of the training dataset in such a way that the weights of the model are kept small. This approach is called weight regularization or weight decay and has proven very effective for decades for both simpler linear models and neural networks.

A simple alternative to gathering more data is to reduce the size of the model or improve regularization, by adjusting hyperparameters such as weight decay coefficients ...

— Page 427, *Deep Learning*, 2016.

Below is a list of five of the most common additional regularization methods.

- **Activity Regularization:** Penalize the model during training based on the magnitude of the activations (Chapter 14).
- **Weight Constraint:** Constrain the magnitude of weights to be within a range or below a limit (Chapter 15).
- **Dropout:** Probabilistically remove inputs during training (Chapter 16).
- **Noise:** Add statistical noise to inputs during training (Chapter 17).
- **Early Stopping:** Monitor model performance on a validation set and stop training when performance degrades (Chapter 18).

Most of these methods have been demonstrated (or proven) to approximate the effect of adding a penalty to the loss function. Each method approaches the problem differently, offering benefits in terms of a mixture of generalization performance, configurability, and/or computational complexity.

12.4 Regularization Recommendations

This section outlines some recommendations for using regularization methods for deep learning neural networks. You should always consider using regularization, unless you have a very large dataset, e.g. big-data scale.

Unless your training set contains tens of millions of examples or more, you should include some mild forms of regularization from the start.

— Page 426, *Deep Learning*, 2016.

A good general recommendation is to design a neural network structure that is under-constrained and to use regularization to reduce the likelihood of overfitting.

... controlling the complexity of the model is not a simple matter of finding the model of the right size, with the right number of parameters. Instead, [...] in practical deep learning scenarios, we almost always do find that the best fitting model (in the sense of minimizing generalization error) is a large model that has been regularized appropriately.

— Page 229, *Deep Learning*, 2016.

Early stopping should almost universally be used in addition to a method to keep weights small during training.

Early stopping should be used almost universally.

— Page 426, *Deep Learning*, 2016.

Some more specific recommendations include:

- **Classical:** use early stopping and weight decay (L2 weight regularization).
- **Alternate:** use early stopping and added noise with a weight constraint.
- **Modern:** use early stopping and dropout, in addition to a weight constraint.

These recommendations would suit Multilayer Perceptrons and Convolutional Neural Networks. Some recommendations for recurrent neural nets include:

- **Classical:** use early stopping with added weight noise and a weight constraint such as maximum norm.
- **Modern:** use early stopping with a backpropagation-through-time-aware version of dropout and a weight constraint.

There are no silver bullets when it comes to regularization and systematic experimentation is strongly encouraged.

12.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

12.5.1 Books

- Chapter 7: Regularization for Deep Learning, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- Section 5.5: Regularization in Neural Networks, *Pattern Recognition and Machine Learning*, 2006.
<https://amzn.to/2Q2rEeP>
- Chapter 16: Heuristics for Improving Generalization, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
<https://amzn.to/2Dxo4XU>
- Chapter 9: Learning and Generalization, *Neural Networks for Pattern Recognition*, 1995.
<https://amzn.to/2I9gNMP>

12.5.2 Articles

- What is overfitting and how can I avoid it? Neural Network FAQ.
ftp://ftp.sas.com/pub/neural/FAQ3.html#A_over
- Regularization (mathematics), Wikipedia.
[https://en.wikipedia.org/wiki/Regularization_\(mathematics\)](https://en.wikipedia.org/wiki/Regularization_(mathematics))

12.6 Summary

In this tutorial, you discovered the problem of overfitting when training neural networks and how it can be addressed with regularization methods. Specifically, you learned:

- Underfitting can easily be addressed by increasing the capacity of the network, but overfitting requires the use of specialized techniques.
- Regularization methods like weight decay provide an easy way to control overfitting for large neural network models.
- A modern recommendation for regularization is to use early stopping with dropout and a weight constraint.

12.6.1 Next

In the next tutorial, you will discover how to add a penalty to the loss function to encourage the training of models with smaller weights and in turn less overfitting.

Chapter 13

Penalize Large Weights with Weight Regularization

Neural networks learn a set of weights that best map inputs to outputs. A network with large network weights can be a sign of an unstable network where small changes in the input can lead to large changes in the output. This can be a sign that the network has overfit the training dataset and will likely perform poorly when making predictions on new data. A solution to this problem is to update the learning algorithm to encourage the network to keep the weights small. This is called weight regularization and it can be used as a general technique to reduce overfitting of the training dataset and improve the generalization of the model. In this tutorial, you will discover weight regularization as an approach to reduce overfitting for neural networks. After reading this tutorial, you will know:

- Large weights in a neural network are a sign of a more complex network that has overfit the training data.
- Penalizing a network based on the size of the network weights during training can reduce overfitting.
- An L1 or L2 vector norm penalty can be added to the optimization of the network to encourage smaller weights.

Let's get started.

13.1 Weight Regularization

In this section you will discover the problem with neural networks that have large weights, a technique that you can use to encourage the development of models with smaller weights called weight regularization and tips for using this technique in your own projects.

13.1.1 Problem With Large Weights

When fitting a neural network model, we must learn the weights of the network (i.e. the model parameters) using stochastic gradient descent and the training dataset. The longer we train the network, the more specialized the weights will become to the training data, overfitting the

training data. The weights will grow in size in order to handle the specifics of the examples seen in the training data. Large weights make the network unstable. Although the weights will be specialized to the training dataset, minor variation or statistical noise on the expected inputs will result in large differences in the output.

Large weights tend to cause sharp transitions in the node functions and thus large changes in output for small changes in the inputs.

— Page 269 *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

Generally, we refer to this model as having a large variance and a small bias. That is, the model is sensitive to the specific examples, the statistical noise, in the training dataset. A model with large weights is more complex than a model with smaller weights. It is a sign of a network that may be overly specialized to training data. In practice, we prefer to choose the simpler models to solve a problem (e.g. Occam's razor). We prefer models with smaller weights.

... given some training data and a network architecture, multiple sets of weight values (multiple models) could explain the data. Simpler models are less likely to over-fit than complex ones. A simple model in this context is a model where the distribution of parameter values has less entropy

— Page 107, *Deep Learning with Python*, 2017.

Another possible issue is that there may be many input variables, each with different levels of relevance to the output variable. Sometimes we can use methods to aid in selecting input variables, but often the interrelationships between variables is not obvious. Having small weights or even zero weights for less relevant or irrelevant inputs to the network will allow the model to focus learning. This too will result in a simpler model.

13.1.2 Encourage Small Weights

The learning algorithm can be updated to encourage the network toward using small weights. One way to do this is to change the calculation of loss used in the optimization of the network to also consider the size of the weights. Remember, that when we train a neural network, we minimize a loss function, such as the log loss in classification or mean squared error in regression. In calculating the loss between the predicted and expected values in a batch, we can add the current size of all weights in the network or add in a layer to this calculation. This is called a penalty because we are penalizing the model proportional to the size of the weights in the model.

Many regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a [...] penalty to the objective function.

— Page 230, *Deep Learning*, 2016.

Larger weights result in a larger penalty, in the form of a larger loss score. The optimization algorithm will then push the model to have smaller weights, i.e. weights no larger than needed to perform well on the training dataset. Smaller weights are considered more regular or less specialized and as such, we refer to this penalty as weight regularization. When this approach of penalizing model coefficients is used in other machine learning models such as linear regression or logistic regression, it may be referred to as shrinkage, because the penalty encourages the coefficients to shrink during the optimization process.

Shrinkage. This approach involves fitting a model involving all p predictors. However, the estimated coefficients are shrunken towards zero [...] This shrinkage (also known as regularization) has the effect of reducing variance

— Page 204, *An Introduction to Statistical Learning: with Applications in R*, 2013.

The addition of a weight size penalty or weight regularization to a neural network has the effect of reducing generalization error and of allowing the model to pay less attention to less relevant input variables.

1) It suppresses any irrelevant components of the weight vector by choosing the smallest vector that solves the learning problem. 2) If the size is chosen right, a weight decay can suppress some of the effect of static noise on the targets.

— *A Simple Weight Decay Can Improve Generalization*, 1992.

13.1.3 How to Penalize Large Weights

There are two parts to penalizing the model based on the size of the weights. The first is the calculation of the size of the weights, and the second is the amount of attention that the optimization process should pay to the penalty.

Calculate Weight Size

Neural network weights are real-values that can be positive or negative, as such, simply adding the weights is not sufficient. There are two main approaches used to calculate the size of the weights, they are:

- Calculate the sum of the absolute values of the weights, called the L1 norm (or L^1).
- Calculate the sum of the squared values of the weights, called the L2 norm (or L^2).

L1 encourages weights to 0.0 if possible, resulting in more sparse weights (weights with more 0.0 values). L2 offers more nuance, both penalizing larger weights more severely, but resulting in less sparse weights. The use of L2 in linear and logistic regression is often referred to as Ridge Regression. This is useful to know when trying to develop an intuition for the penalty or examples of its usage.

In other academic communities, L2 regularization is also known as ridge regression or Tikhonov regularization.

— Page 231, *Deep Learning*, 2016.

The weights may be considered a vector and the magnitude of a vector is called its norm, from linear algebra. As such, penalizing the model based on the size of the weights is also referred to as a weight or parameter norm penalty. It is possible to include both L1 and L2 approaches to calculating the size of the weights as the penalty. This is akin to the use of both penalties used in the Elastic Net algorithm for linear and logistic regression. The L2 approach is perhaps the most used and is traditionally referred to as *weight decay* in the field of neural networks. It is called *shrinkage* in statistics, a name that encourages you to think of the impact of the penalty on the model weights during the learning process.

This particular choice of regularizer is known in the machine learning literature as weight decay because in sequential learning algorithms, it encourages weight values to decay towards zero, unless supported by the data. In statistics, it provides an example of a parameter shrinkage method because it shrinks parameter values towards zero.

— Page 144-145, *Pattern Recognition and Machine Learning*, 2006.

Recall that each node has input weights and a bias weight. The bias weight is generally not included in the penalty because the *input* is constant.

Control Impact of the Penalty

The calculated size of the weights is added to the loss objective function when training the network. Rather than adding each weight to the penalty directly, they can be weighted using a new hyperparameter called alpha (α) or sometimes lambda. This controls the amount of attention that the learning process should pay to the penalty. Or put another way, the amount to penalize the model based on the size of the weights. The alpha hyperparameter has a value between 0.0 (no penalty) and 1.0 (full penalty). This hyperparameter controls the amount of bias in the model from 0.0, or low bias (high variance), to 1.0, or high bias (low variance).

If the penalty is too strong, the model will underestimate the weights and underfit the problem. If the penalty is too weak, the model will be allowed to overfit the training data. The vector norm of the weights is often calculated per-layer, rather than across the entire network. This allows more flexibility in the choice of the type of regularization used (e.g. L1 for inputs, L2 elsewhere) and flexibility in the alpha value, although it is common to use the same alpha value on each layer by default.

In the context of neural networks, it is sometimes desirable to use a separate penalty with a different coefficient for each layer of the network. Because it can be expensive to search for the correct value of multiple hyperparameters, it is still reasonable to use the same weight decay at all layers just to reduce the size of search space.

— Page 230, *Deep Learning*, 2016.

13.1.4 Examples of Weight Regularization

It can be helpful to look at some examples of weight regularization configurations reported in the literature. It is important to select and tune a regularization technique specific to your network and dataset, although real examples can also give an idea of common configurations that may be a useful starting point. Recall that 0.1 can be written in scientific notation as 1e-1 or 1E-1 or as an exponential 10^{-1} , 0.01 as 1e-2 or 10^{-2} and so on.

Examples of MLP Weight Regularization

Weight regularization was borrowed from penalized regression models in statistics. The most common type of regularization is L2, also called simply *weight decay*, with values often on a logarithmic scale between 0 and 0.1, such as 0.1, 0.001, 0.0001, etc.

Reasonable values of lambda [regularization hyperparameter] range between 0 and 0.1.

— Page 144, *Applied Predictive Modeling*, 2013.

The classic text on Multilayer Perceptrons *Neural Smithing: Supervised Learning in Feed-forward Artificial Neural Networks* provides a worked example demonstrating the impact of weight decay by first training a model without any regularization, then steadily increasing the penalty. They demonstrate graphically that weight decay has the effect of improving the resulting decision function.

... net was trained [...] with weight decay increasing from 0 to 1E-5 at 1200 epochs, to 1E-4 at 2500 epochs, and to 1E-3 at 400 epochs. [...] The surface is smoother and transitions are more gradual

This is an interesting procedure that may be worth investigating. The authors also comment on the difficulty of predicting the effect of weight decay on a problem.

... it is difficult to predict ahead of time what value is needed to achieve desired results. The value of 0.001 was chosen arbitrarily because it is a typically cited round number

Examples of CNN Weight Regularization

Weight regularization does not seem widely used in CNN models, or if it is used, its use is not widely reported. L2 weight regularization with very small regularization hyperparameters such as (e.g. 0.0005 or 5×10^{-4}) may be a good starting point. Alex Krizhevsky, et al. from the University of Toronto in their 2012 paper titled *ImageNet Classification with Deep Convolutional Neural Networks* developed a deep CNN model for the ImageNet dataset, achieving then state-of-the-art results reported:

...and weight decay of 0.0005. We found that this small amount of weight decay was important for the model to learn. In other words, weight decay here is not merely a regularizer: it reduces the model's training error.

Karen Simonyan and Andrew Zisserman from Oxford in their 2015 paper titled *Very Deep Convolutional Networks for Large-Scale Image Recognition* develop a CNN for the ImageNet dataset and report:

The training was regularised by weight decay (the L2 penalty multiplier set to 5×10^{-4})

Francois Chollet from Google (and author of Keras) in his 2016 paper titled *Xception: Deep Learning with Depthwise Separable Convolutions* reported the weight decay for both the Inception V3 CNN model from Google (not clear from the Inception V3 paper) and the weight decay used in his improved Xception for the ImageNet dataset:

The Inception V3 model uses a weight decay (L2 regularization) rate of 4e-5, which has been carefully tuned for performance on ImageNet. We found this rate to be quite suboptimal for Xception and instead settled for 1e-5.

Examples of LSTM Weight Regularization

It is common to use weight regularization with LSTM models. An often used configuration is L2 (weight decay) and very small hyperparameters (e.g. 10^{-6}). It is often not reported what weights are regularized (input, recurrent, and/or bias), although one would assume that both input and recurrent weights are regularized only. Gabriel Pereyra, et al. from Google Brain in the 2017 paper titled *Regularizing Neural Networks by Penalizing Confident Output Distributions* apply a seq2seq LSTMs models to predicting characters from the Wall Street Journal and report:

All models used weight decay of 10^{-6}

Barret Zoph and Quoc Le from Google Brain in the 2017 paper titled *Neural Architecture Search with Reinforcement Learning* use LSTMs and reinforcement learning to learn network architectures to best address the CIFAR-10 dataset and report:

weight decay of 1e-4

Ron Weiss, et al. from Google Brain and Nvidia in their 2017 paper titled *Sequence-to-Sequence Models Can Directly Translate Foreign Speech* develop a sequence-to-sequence LSTM for speech translation and report:

L2 weight decay is used with a weight of 10^{-6}

13.1.5 Tips for Using Weight Regularization

This section provides some tips for using weight regularization with your neural network.

Use With All Network Types

Weight regularization is a generic approach. It can be used with most, perhaps all, types of neural network models, not least the most common network types of Multilayer Perceptrons, Convolutional Neural Networks, and Long Short-Term Memory Recurrent Neural Networks. In the case of LSTMs, it may be desirable to use different penalties or penalty configurations for the input and recurrent connections.

Standardize Input Data

It is generally good practice to update input variables to have the same scale. When input variables have different scales, the scale of the weights of the network will, in turn, vary accordingly. This introduces a problem when using weight regularization because the absolute or squared values of the weights must be added for use in the penalty. This problem can be addressed by either normalizing or standardizing input variables.

Use a Larger Network

It is common for larger networks (more layers or more nodes) to more easily overfit the training data. When using weight regularization, it is possible to use larger networks with less risk of overfitting. A good configuration strategy may be to start with larger networks and use weight decay.

Grid Search Parameters

It is common to use small values for the regularization hyperparameter that controls the contribution of each weight to the penalty. Perhaps start by testing values on a log scale, such as 0.1, 0.001, and 0.0001. Then use a grid search at the order of magnitude that shows the most promise.

Use L1 + L2 Together

Rather than trying to choose between L1 and L2 penalties, use both. Modern and effective linear regression methods such as the Elastic Net use both L1 and L2 penalties at the same time and this can be a useful approach to try. This gives you both the nuance of L2 and the sparsity encouraged by L1.

Use on a Trained Network

The use of weight regularization may allow more elaborate training schemes. For example, a model may be fit on training data first without any regularization, then updated later with the use of a weight penalty to reduce the size of the weights of the already well-performing model.

13.2 Weight Regularization Keras API

This section demonstrates how to use weight regularization techniques with the Keras API.

13.2.1 Create Weight Regularizers

Keras provides a weight regularization API that allows you to add a penalty for weight size to the loss function. Three different regularizer instances are provided; they are:

- **L1**: Sum of the absolute weights.
- **L2**: Sum of the squared weights.

- **L1L2:** Sum of the absolute and the squared weights.

The regularizers are provided under `keras.regularizers` module and have the names `l1`, `l2` and `l1_l2`. Each takes the regularizer hyperparameter as an argument. For example:

```
# example of creating an l2 regularizer
from keras.regularizers import l2
reg = l2(0.01)
```

Listing 13.1: Example of creating an L2 regularizer in Keras.

13.2.2 Weight Regularization on Layers

By default, no regularizer is used in any layers. A weight regularizer can be added to each layer when the layer is defined in a Keras model. This is achieved by setting the `kernel_regularizer` argument on each layer. A separate regularizer can also be used for the bias via the `bias_regularizer` argument, although this is less often used. Let's look at some examples.

Weight Regularization for MLPs

The example below sets an `l2` regularizer on a `Dense` fully connected layer:

```
# example of l2 on a dense layer
from keras.layers import Dense
from keras.regularizers import l2
...
model.add(Dense(32, kernel_regularizer=l2(0.01), bias_regularizer=l2(0.01)))
...
```

Listing 13.2: Example of adding weight regularization to an MLP.

Weight Regularization for CNNs

Like the `Dense` layer, the Convolutional layers (e.g. `Conv1D` and `Conv2D`) also use the `kernel_regularizer` and `bias_regularizer` arguments to define a regularizer. The example below sets an `l2` regularizer on a `Conv2D` convolutional layer:

```
# example of l2 on a convolutional layer
from keras.layers import Conv2D
from keras.regularizers import l2
...
model.add(Conv2D(32, (3,3), kernel_regularizer=l2(0.01), bias_regularizer=l2(0.01)))
...
```

Listing 13.3: Example of adding weight regularization to a CNN.

Weight Regularization for RNNs

Recurrent layers like the `LSTM` layer offer more flexibility in regularizing the weights. The input, recurrent, and bias weights can all be regularized separately via the `kernel_regularizer`, `recurrent_regularizer`, and `bias_regularizer` arguments. The example below sets an `l2` regularizer on an `LSTM` recurrent layer:

```
# example of 12 on an lstm layer
from keras.layers import LSTM
from keras.regularizers import l2
...
model.add(LSTM(32, kernel_regularizer=l2(0.01), recurrent_regularizer=l2(0.01),
    bias_regularizer=l2(0.01)))
...
```

Listing 13.4: Example of adding weight regularization to an LSTM.

Now that we know how to use the weight regularization API, let's look at a worked example.

13.3 Weight Regularization Case Study

In this section, we will demonstrate how to use weight regularization to reduce overfitting of an MLP on a simple binary classification problem. This example provides a template for applying weight regularization to your own neural network for classification and regression problems.

13.3.1 Binary Classification Problem

We will use a standard binary classification problem that defines two semi-circles of observations: one semi-circle for each class. Each observation has two input variables with the same scale and a class output value of either 0 or 1. This dataset is called the *moons* dataset because of the shape of the observations in each class when plotted. We can use the `make_moons()` function to generate observations from this problem. We will add noise to the data and seed the random number generator so that the same samples are generated each time the code is run.

```
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
```

Listing 13.5: Example of creating samples for the moons problem.

We can plot the dataset where the two variables are taken as x and y coordinates on a graph and the class value is taken as the color of the observation. The complete example of generating the dataset and plotting it is listed below.

```
# scatter plot of moons dataset
from sklearn.datasets import make_moons
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# scatter plot for each class value
for class_value in range(2):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 13.6: Example of plotting samples from the two moons problem.

Running the example creates a scatter plot showing the semi-circle or moon shape of the observations in each class. We can see the noise in the dispersal of the points making the moons less obvious.

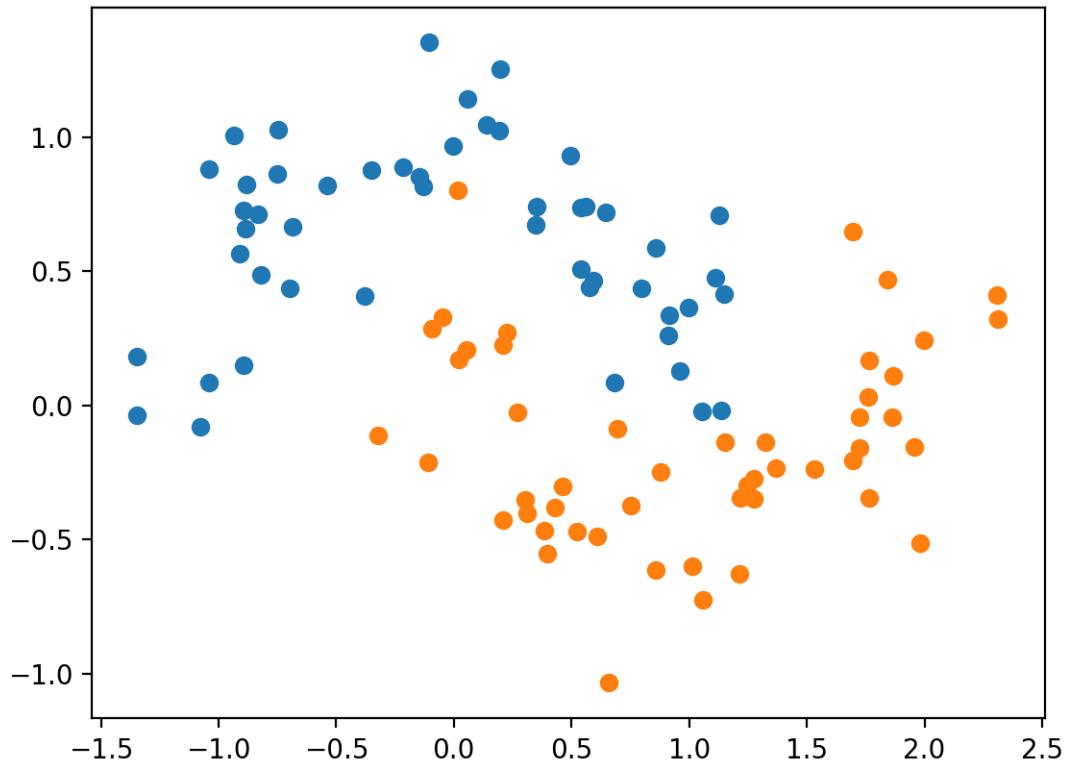


Figure 13.1: Scatter Plot of Moons Dataset With Color Showing the Class Value of Each Sample.

This is a good test problem because the classes cannot be separated by a line, e.g. are not linearly separable, requiring a nonlinear method such as a neural network to address. We have only generated 100 samples, which is small for a neural network, providing the opportunity to overfit the training dataset and have higher error on the test dataset: a good case for using regularization. Further, the samples have noise, giving the model an opportunity to learn aspects of the samples that don't generalize.

13.3.2 Overfit Multilayer Perceptron Model

We can develop an MLP model to address this binary classification problem. The model will have one hidden layer with more nodes that may be required to solve this problem, providing an opportunity to overfit. We will also train the model for longer than is required to ensure the model overfits. Before we define the model, we will split the dataset into train and test sets, using 30 examples to train the model and 70 to evaluate the fit model's performance.

```
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
```

```
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 13.7: Example of preparing the data modeling.

Next, we can define the model. The model uses 500 nodes in the hidden layer and the rectified linear activation function. A sigmoid activation function is used in the output layer in order to predict class values of 0 or 1. The model is optimized using the binary cross-entropy loss function, suitable for binary classification problems and the efficient Adam version of gradient descent.

```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 13.8: Example of defining a model for the moons problem.

The defined model is then fit on the training data for 4,000 epochs and the default batch size of 32. We will use the test set as the validation dataset to get an idea of the model performance on a hold out dataset during training

```
# fit model
history = model.fit(trainX, trainy, epochs=4000, validation_data=(testX, testy), verbose=0)
```

Listing 13.9: Example of fitting a model for the moons problem.

Next, we will evaluate the performance of the model on the test dataset and report the result.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 13.10: Example of evaluating a model for the moons problem.

Finally, we will plot learning curves of model performance in terms of cross-entropy loss and classification accuracy on the train and test datasets for each epoch during training.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 13.11: Example of plotting learning curves of model performance during training.

We can tie all of these pieces together; the complete example is listed below.

```
# overfit mlp for the moons dataset
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test sets
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, epochs=4000, validation_data=(testX, testy), verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 13.12: Example of fitting an MLP for the two moons problem.

Running the example first reports the model performance on the train and test datasets. We can see that the model has better performance on the training dataset than the test dataset, one possible sign of overfitting.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

Train: 1.000, Test: 0.914

Listing 13.13: Example output fitting an MLP on the two moons problem.

Line plots showing learning curves of cross-entropy loss and classification accuracy on the train and test sets for each training epoch are also created. The learning curve for loss shows a clear pattern of overfitting, mirrored in the learning curve for the classification accuracy.

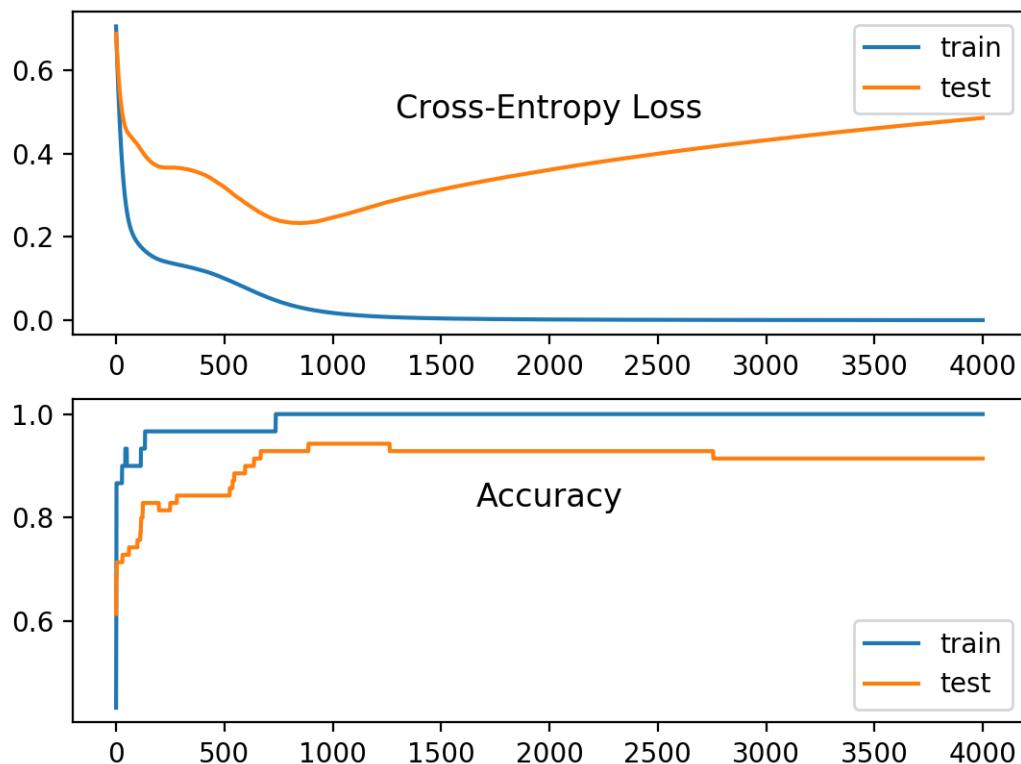


Figure 13.2: Line Plots of Learning Curves for Loss and Accuracy on Train and Test Datasets While Training.

13.3.3 MLP Model With Weight Regularization

We can add weight regularization to the hidden layer to reduce the overfitting of the model to the training dataset and improve the performance on the holdout set. We will use the L2 vector norm also called weight decay with a regularization parameter (called alpha or lambda) of 0.001, chosen arbitrarily. This can be done by adding the `kernel_regularizer` argument to the layer and setting it to an instance of `l2`.

```
model.add(Dense(500, input_dim=2, activation='relu', kernel_regularizer=l2(0.001)))
```

Listing 13.14: Updated layer to use weight regularization.

The updated example of fitting and evaluating the model on the moons dataset with weight regularization is listed below.

```
# mlp with weight regularization for the moons dataset
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l2
from matplotlib import pyplot
# generate 2d classification dataset
```

```

X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test sets
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, epochs=4000, validation_data=(testX, testy), verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 13.15: Example of updated MLP using weight regularization for the two moons problem.

Running the example first reports the performance of the model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we see no change in the accuracy on the training dataset and an improvement on the test dataset.

Train: 1.000, Test: 0.943

Listing 13.16: Example output from updated MLP using weight regularization for the two moons problem.

We would expect that the telltale learning curve for overfitting would also have been changed through the use of weight regularization. Instead of the accuracy of the model on the test set increasing and then decreasing again, we should see it continually rise during training. As expected, we see the learning curves for loss and accuracy on the test dataset plateau, indicating that the model has no longer overfit the training dataset.

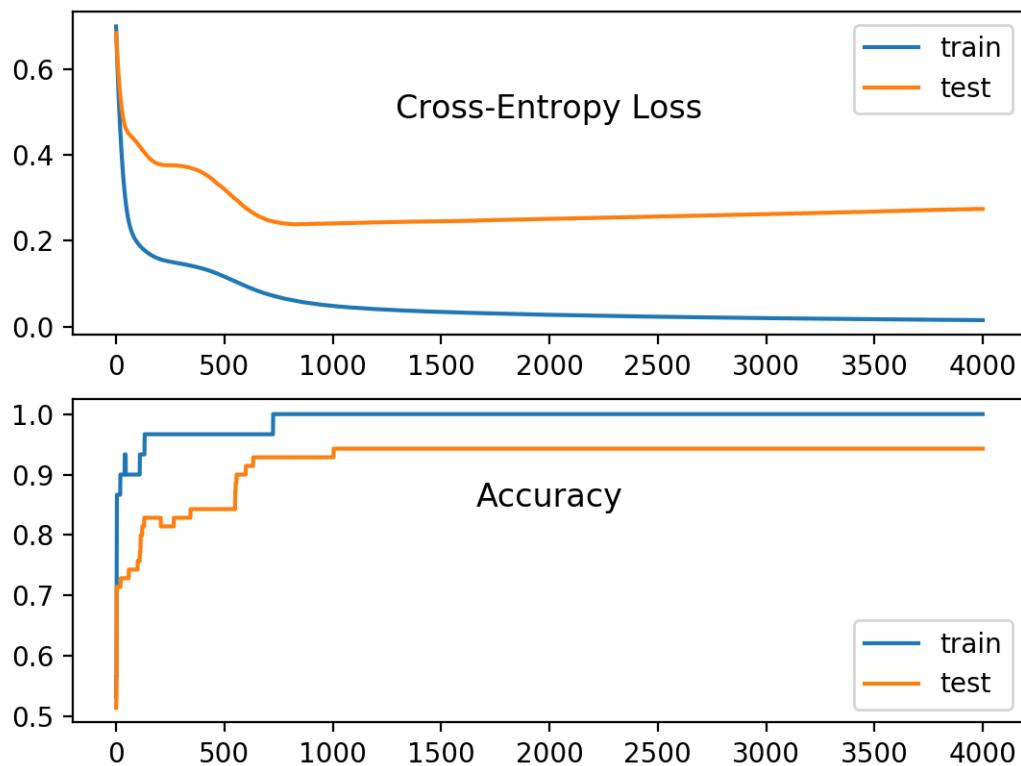


Figure 13.3: Line Plots Learning Curves for Loss and Accuracy on Train and Test Datasets While Training Without Overfitting.

13.3.4 Grid Search Regularization Hyperparameter

Once you can confirm that weight regularization may improve your overfit model, you can test different values of the regularization parameter. It is a good practice to first grid search through some orders of magnitude between 0.0 and 0.1, then once a level is found, to grid search on that level. We can grid search through the orders of magnitude by defining the values to test, looping through each and recording the train and test performance.

```
...
# grid search values
values = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
all_train, all_test = list(), list()
for param in values:
    ...
    model.add(Dense(500, input_dim=2, activation='relu', kernel_regularizer=l2(param)))
    ...
    all_train.append(train_acc)
    all_test.append(test_acc)
```

Listing 13.17: Example of a grid search for regularization values.

Once we have all of the values, we can graph the results as a line plot to help spot any patterns in the configurations to the train and test accuracies. Because parameters jump orders of magnitude (powers of 10), we can create a line plot of the results using a logarithmic scale. The Matplotlib library allows this via the `semilogx()` function. For example:

```
pyplot.semilogx(values, all_train, label='train', marker='o')
pyplot.semilogx(values, all_test, label='test', marker='o')
```

Listing 13.18: Example of plotting the results from grid searching regularization values.

The complete example for grid searching weight regularization values on the moon dataset is listed below.

```
# grid search regularization values for moons dataset
from sklearn.datasets import make_moons
from keras.layers import Dense
from keras.models import Sequential
from keras.regularizers import l2
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# grid search values
values = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
all_train, all_test = list(), list()
for param in values:
    # define model
    model = Sequential()
    model.add(Dense(500, input_dim=2, activation='relu', kernel_regularizer=l2(param)))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=4000, verbose=0)
    # evaluate the model
    _, train_acc = model.evaluate(trainX, trainy, verbose=0)
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    print('Param: %f, Train: %.3f, Test: %.3f' % (param, train_acc, test_acc))
    all_train.append(train_acc)
    all_test.append(test_acc)
# plot train and test means
pyplot.semilogx(values, all_train, label='train', marker='o')
pyplot.semilogx(values, all_test, label='test', marker='o')
pyplot.legend()
pyplot.show()
```

Listing 13.19: Example grid searching weight regularization values for the two moons problem.

Running the example prints the parameter value and the accuracy on the train and test sets for each evaluated model.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the results suggest that 0.01 or 0.001 may be sufficient and may provide good bounds for further grid searching.

```
Param: 0.100000, Train: 0.967, Test: 0.829
Param: 0.010000, Train: 1.000, Test: 0.943
Param: 0.001000, Train: 1.000, Test: 0.943
Param: 0.000100, Train: 1.000, Test: 0.929
Param: 0.000010, Train: 1.000, Test: 0.929
Param: 0.000001, Train: 1.000, Test: 0.914
```

Listing 13.20: Example output from grid searching weight regularization values for the two moons problem.

A line plot of the results is also created, showing the increase in test accuracy with larger weight regularization parameter values, at least to a point. We can see that using the largest value of 0.1 results in a large drop in both train and test accuracy.

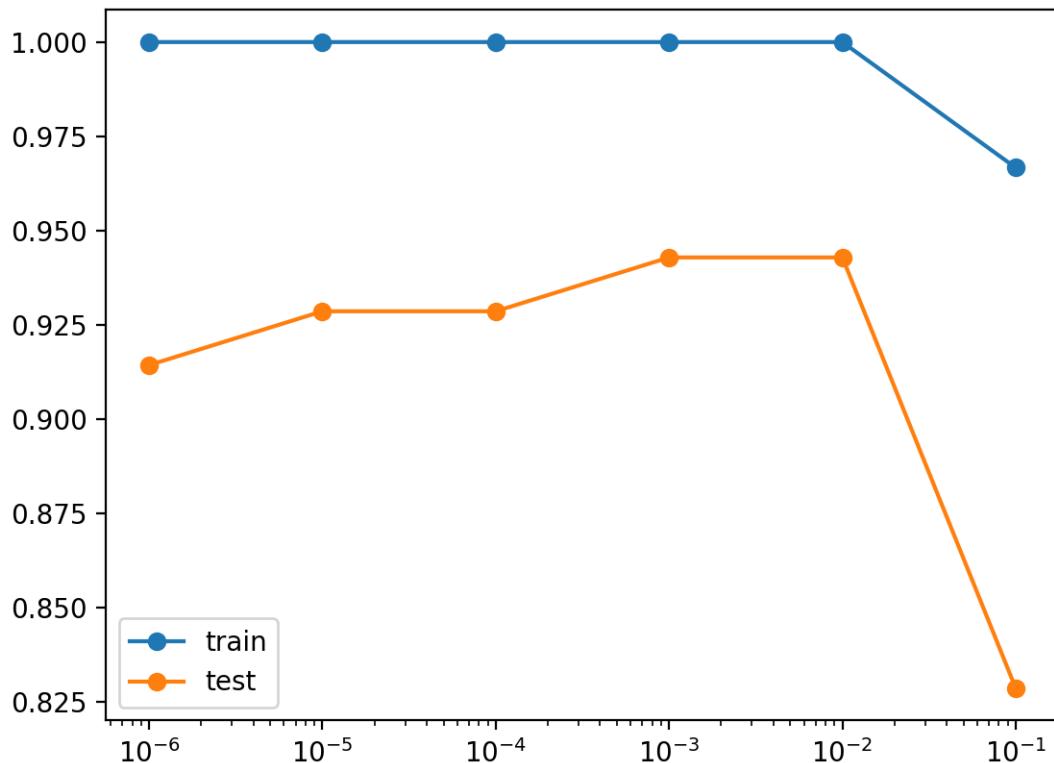


Figure 13.4: Line Plot of Model Accuracy on Train and Test Datasets With Different Weight Regularization Parameters.

13.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Try Alternates.** Update the example to use L1 or the combined L1L2 methods instead of L2 regularization.
- **Report Weight Norm.** Update the example to calculate the magnitude of the network weights and demonstrate that regularization indeed made the magnitude smaller.
- **Regularize Output Layer.** Update the example to regularize the output layer of the model and compare the results.
- **Regularize Bias.** Update the example to regularize the bias weight and compare the results.
- **Repeated Model Evaluation.** Update the example to fit and evaluate the model multiple times and report the mean and standard deviation of model performance.
- **Grid Search Along Order of Magnitude.** Update the grid search example to grid search within the best-performing order of magnitude of parameter values.
- **Repeated Regularization of Model.** Create a new example to continue the training of a fit model with increasing levels of regularization (e.g. 1E-6, 1E-5, etc.) and see if it results in a better performing model on the test set.

If you explore any of these extensions, I'd love to know.

13.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

13.5.1 Books

- Section 7.1: Parameter Norm Penalties, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- Section 5.5: Regularization in Neural Networks, *Pattern Recognition and Machine Learning*, 2006.
<https://amzn.to/2Q2rEeP>
- Section 16.5: Weight Decay, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
<https://amzn.to/2PBsezv>
- Section 4.4.2: Adding weight regularization, *Deep Learning with Python*, 2017.
<https://amzn.to/2wVqZDq>
- Section 6.2: Shrinkage Methods, *An Introduction to Statistical Learning: with Applications in R*, 2013.
<https://amzn.to/2MXGK7I>

13.5.2 Papers

- *A Simple Weight Decay Can Improve Generalization*, 1992.
<https://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf>
- *Note on generalization, regularization and architecture selection in nonlinear learning systems*, 1991.
<https://ieeexplore.ieee.org/abstract/document/239541/>

13.5.3 APIs

- Keras Regularization API.
<https://keras.io/regularizers/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>
- Keras Recurrent Layers API.
<https://keras.io/layers/recurrent/>
- sklearn.datasets.make_moons API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html
- matplotlib.pyplot.semilogx API.
https://matplotlib.org/api/_as_gen/matplotlib.pyplot.semilogx.html

13.5.4 Articles

- Regularization (mathematics), Wikipedia.
[https://en.wikipedia.org/wiki/Regularization_\(mathematics\)](https://en.wikipedia.org/wiki/Regularization_(mathematics))
- Weight Decay in Neural Networks, Metacademy.
https://metacademy.org/graphs/concepts/weight_decay_neural_networks
- Why large weights are prohibited in neural networks?
<https://datascience.stackexchange.com/questions/23287/why-large-weights-are-prohibited-in-neural-networks>

13.6 Summary

In this tutorial, you discovered weight regularization as an approach to reduce overfitting for neural networks. Specifically, you learned:

- Large weights in a neural network are a sign of a more complex network that has overfit the training data.

- Penalizing a network based on the size of the network weights during training can reduce overfitting.
- An L1 or L2 vector norm penalty can be added to the optimization of the network to encourage smaller weights.

13.6.1 Next

In the next tutorial, discover how to update the loss function to encourage the training of models with simpler internal representations.

Chapter 14

Sparse Representations with Activity Regularization

Deep learning models are capable of automatically learning a rich internal representation from raw input data. This is called feature or representation learning. Better learned representations, in turn, can lead to better insights into the domain, e.g. via visualization of learned features, and to better predictive models that make use of the learned features. A problem with learned features is that they can be too specialized to the training data, or overfit, and not generalize well to new examples. Large values in the learned representation can be a sign of the representation being overfit. Activity or representation regularization provides a technique to encourage the learned representations, the output or activation of the hidden layer or layers of the network, to stay small and sparse. In this tutorial, you will discover activation regularization as a technique to improve the generalization of learned features in neural networks. After reading this tutorial, you will know:

- Neural networks learn features from data and models, such as autoencoders and encoder-decoder models, and explicitly seek effective learned representations.
- Similar to weights, large values in learned features, e.g. large activations, may indicate an overfit model.
- The addition of penalties to the loss function that penalize a model in proportion to the magnitude of the activations may result in more robust and generalized learned features.

Let's get started.

14.1 Activity Regularization

In this section you will discover the problem with neural networks that have large activity, a technique that you can use to encourage the development of models with sparse activity called activity regularization and tips for using this technique in your own projects.

14.1.1 Problem With Learned Features

Deep learning models are able to perform feature learning. That is, during the training of the network, the model will automatically extract the salient features from the input patterns or

learn features. These features may be used in the network in order to predict a quantity for regression or predict a class value for classification. These internal representations are tangible things. The output of a hidden layer within the network represent the learned features by the model at that point in the network.

There is a field of study focused on the efficient and effective automatic learning of features, often investigated by having a network reduce an input to a small learned feature before using a second network to reconstruct the original input from the learned feature. Models of this type are called auto-encoders, or encoder-decoders, and their learned features can be useful to learn more about the domain (e.g. via visualization) and in predictive models. The learned features, or *encoded inputs*, must be large enough to capture the salient features of the input but also focused enough to not overfit the specific examples in the training dataset. As such, there is a tension between the expressiveness and the generalization of the learned features.

More importantly, when the dimension of the code in an encoder-decoder architecture is larger than the input, it is necessary to limit the amount of information carried by the code, lest the encoder-decoder may simply learn the identity function in a trivial way and produce uninteresting features.

— *Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition*, 2007.

In the same way that large weights in the network can signify an unstable and overfit model, large output values in the learned features can signify the same problems. It is desirable to have small values in the learned features, e.g. small outputs or activations from the encoder network.

14.1.2 Encourage Small Activations

The loss function of the network can be updated to penalize models in proportion to the magnitude of their activation. This is similar to *weight regularization* where the loss function is updated to penalize the model in proportion to the magnitude of the weights. The output of a layer is referred to as its *activation* or *activity*, as such, this form of penalty or regularization is referred to as *activation regularization* or *activity regularization*.

... place a penalty on the activations of the units in a neural network, encouraging their activations to be sparse.

— Page 254, *Deep Learning*, 2016.

The output of an encoder or, generally, the output of a hidden layer in a neural network may be considered the representation of the problem at that point in the model. As such, this type of penalty may also be referred to as *representation regularization*. The desire to have small activations or even very few activations with mostly zero values is also called a desire for sparsity. As such, this type of penalty is also referred to as *sparse feature learning*.

One way to limit the information content of an overcomplete code is to make it sparse.

— *Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition*, 2007.

The encouragement of sparse learned features in autoencoder models is referred to as *sparse autoencoders*.

A sparse autoencoder is simply an autoencoder whose training criterion involves a sparsity penalty on the code layer, in addition to the reconstruction error

— Page 505, *Deep Learning*, 2016.

Sparsity is most commonly sought when a larger-than-required hidden layer (e.g. overcomplete) is used to learn features that may encourage overfitting. The introduction of a sparsity penalty counters this problem and encourages better generalization. A sparse overcomplete learned feature has been shown to be more effective than other types of learned features offering better robustness to noise and even transforms in the input, e.g. learned features of images may have improved invariance to the position of objects in the image.

Sparse-overcomplete representations have a number of theoretical and practical advantages, as demonstrated in a number of recent studies. In particular, they have good robustness to noise, and provide a good tiling of the joint space of location and frequency. In addition, they are advantageous for classifiers because classification is more likely to be easier in higher dimensional spaces.

— *Sparse Feature Learning for Deep Belief Networks*, 2007.

There is a general focus on sparsity of the representations rather than small vector magnitudes. A study of these representations that is more general than the use of neural networks is known as *sparse coding*.

Sparse coding provides a class of algorithms for finding succinct representations of stimuli; given only unlabeled input data, it learns basis functions that capture higher-level features in the data.

— *Efficient Sparse Coding Algorithms*, 2007.

14.1.3 How to Encourage Small Activations

An activation penalty can be applied per-layer, perhaps only at one layer that is the focus of the learned representation, such as the output of the encoder model or the middle (bottleneck) of an autoencoder model. A constraint can be applied that adds a penalty proportional to the magnitude of the vector output of the layer. The activation values may be positive or negative, so we cannot simply sum the values. Two common methods for calculating the magnitude of the activation are:

- Sum of the absolute activation values, called L1 vector norm.
- Sum of the squared activation values, called the L2 vector norm.

The L1 norm encourages sparsity, e.g. allows some activations to become zero, whereas the L2 norm encourages small activation values in general. Use of the L1 norm may be a more commonly used penalty for activation regularization. A hyperparameter must be specified that indicates the amount or degree that the loss function will weight or pay attention to the penalty. Common values are on a logarithmic scale between 0 and 0.1, such as 0.1, 0.001, 0.0001, etc. Activity regularization can be used in conjunction with other regularization techniques, such as weight regularization.

14.1.4 Examples of Activation Regularization

This section provides some examples of activation regularization in order to provide some context for how the technique may be used in practice. Regularized or sparse activations were originally sought as an approach to support the development of much deeper neural networks, early in the history of deep learning. As such, many examples may make use of architectures like restricted Boltzmann machines (RBMs) that have been replaced by more modern methods. Another big application of weight regularization is in autoencoders with semi-labeled or unlabeled data, so-called sparse autoencoders. Xavier Glorot, et al. at the University of Montreal introduced the use of the rectified linear activation function to encourage sparsity of representation. They used an L1 penalty and evaluate deep supervised MLPs on a range of classical computer vision classification tasks such as MNIST and CIFAR10.

Additionally, an L1 penalty on the activations with a coefficient of 0.001 was added to the cost function during pre-training and fine-tuning in order to increase the amount of sparsity in the learned representations

— *Deep Sparse Rectifier Neural Networks*, 2011.

Stephen Merity, et al. from Salesforce Research used L2 activation regularization with LSTMs on outputs and recurrent outputs for natural language process in conjunction with dropout regularization. They tested a suite of different activation regularization coefficient values on a range of language modeling problems.

While simple to implement, activity regularization and temporal activity regularization are competitive with other far more complex regularization techniques and offer equivalent or better results.

— *Revisiting Activation Regularization for Language RNNs*, 2017.

14.1.5 Tips for Using Activation Regularization

This section provides some tips for using activation regularization with your neural network.

Use With All Network Types

Activation regularization is a generic approach. It can be used with most, perhaps all, types of neural network models, not least the most common network types of Multilayer Perceptrons, Convolutional Neural Networks, and Long Short-Term Memory Recurrent Neural Networks.

Use With Autoencoders and Encoder-Decoders

Activity regularization may be best suited to those model types that explicitly seek an efficient learned representation. These include models such as autoencoders (i.e. sparse autoencoders) and encoder-decoder models, such as encoder-decoder LSTMs used for sequence-to-sequence prediction problems.

Experiment With Different Norms

The most common activation regularization is the L1 norm as it encourages sparsity. Experiment with other types of regularization such as the L2 norm or using both the L1 and L2 norms at the same time, e.g. like the Elastic Net linear regression algorithm.

Use Rectified Linear Activation

The rectified linear activation function, also called `relu`, is an activation function that is now widely used in the hidden layer of deep neural networks. Unlike classical activation functions such as `tanh` (hyperbolic tangent function) and `sigmoid` (logistic function), the `relu` function allows exact zero values easily. This makes it a good candidate when learning sparse representations, such as with the L1 vector norm activation regularization.

Grid Search Parameters

It is common to use small values for the regularization hyperparameter that controls the contribution of each activation to the penalty. Perhaps start by testing values on a log scale, such as 0.1, 0.001, and 0.0001. Then use a grid search at the order of magnitude that shows the most promise.

Standardize Input Data

It is a generally good practice to rescale input variables to have the same scale. When input variables have different scales, the scale of the weights of the network will, in turn, vary accordingly. Large weights can saturate the nonlinear transfer function and reduce the variance in the output from the layer. This may introduce a problem when using activation regularization. This problem can be addressed by either normalizing or standardizing input variables.

Use an Overcomplete Representation

Configure the layer chosen to be the learned features, e.g. the output of the encoder or the bottleneck in the autoencoder, to have more nodes than may be required. This is called an overcomplete representation that will encourage the network to overfit the training examples. This can be countered with a strong activation regularization in order to encourage a rich learned representation that is also sparse.

14.2 Activity Regularization Keras API

This section demonstrates how to use activity regularization techniques with the Keras API.

14.2.1 Activity Regularization in Keras

Keras supports activity regularization. There are three different regularization techniques supported, each provided as a class in the `keras.regularizers` module:

- `L1`: Activity is calculated as the sum of absolute values.

- `l2`: Activity is calculated as the sum of the squared values.
- `l1_l2`: Activity is calculated as the sum of absolute and sum of the squared values.

Each of the `l1` and `l2` regularizers takes a single hyperparameter that controls the amount that each activity contributes to the sum. The `l1_l2` regularizer takes two hyperparameters, one for each of the `l1` and `l2` methods. The regularizer class must be imported and then instantiated; for example:

```
# import regularizer
from keras.regularizers import l1
# instantiate regularizer
reg = l1(0.001)
```

Listing 14.1: Example of creating an L1 regularizer.

14.2.2 Activity Regularization on Layers

Activity regularization is specified on a layer in Keras. This can be achieved by setting the `activity_regularizer` argument on the layer to an instantiated and configured regularizer class. The regularizer is applied to the output of the layer, but you have control over what the *output* of the layer actually means. Specifically, you have flexibility as to whether the layer output means that the regularization is applied before or after the *activation* function. For example, you can specify the function and the regularization on the layer, in which case activation regularization is applied to the output of the activation function, in this case, `relu`.

```
...
model.add(Dense(32, activation='relu', activity_regularizer=l1(0.001)))
...
```

Listing 14.2: Example of activity regularization after activation.

Alternately, you can specify a linear activation function (the default, that does not perform any transform) which means that the activation regularization is applied on the raw outputs, then, the activation function can be added as a subsequent layer.

```
...
model.add(Dense(32, activation='linear', activity_regularizer=l1(0.001)))
model.add(Activation('relu'))
...
```

Listing 14.3: Example of activity regularization before activation.

The latter is probably the preferred usage of activation regularization as described in *Deep Sparse Rectifier Neural Networks* in order to allow the model to learn to take activations to a true zero value in conjunction with the rectified linear activation function. Nevertheless, the two possible uses of activation regularization may be explored in order to discover what works best for your specific model and dataset. Let's take a look at how activity regularization can be used with some common layer types.

MLP Activity Regularization

The example below sets L_1 norm activity regularization on a `Dense` fully connected layer.

```
# example of L1 norm on activity from a dense layer
from keras.layers import Dense
from keras.regularizers import L1
...
model.add(Dense(32, activity_regularizer=L1(0.001)))
...
```

Listing 14.4: Example of activity regularization for an MLP.

CNN Activity Regularization

The example below sets L_1 norm activity regularization on a `Conv2D` convolutional layer.

```
# example of L1 norm on activity from a cnn layer
from keras.layers import Conv2D
from keras.regularizers import L1
...
model.add(Conv2D(32, (3,3), activity_regularizer=L1(0.001)))
...
```

Listing 14.5: Example of activity regularization for a CNN.

RNN Activity Regularization

The example below sets L_1 norm activity regularization on an `LSTM` recurrent layer.

```
# example of L1 norm on activity from an lstm layer
from keras.layers import LSTM
from keras.regularizers import L1
...
model.add(LSTM(32, activity_regularizer=L1(0.001)))
...
```

Listing 14.6: Example of activity regularization for an LSTM.

Now that we know how to use the activity regularization API, let's look at a worked example.

14.3 Activity Regularization Case Study

In this section, we will demonstrate how to use activity regularization to reduce overfitting of an MLP on a simple binary classification problem. Although activity regularization is most often used to encourage sparse learned representations in autoencoder and encoder-decoder models, it can also be used directly within normal neural networks to achieve the same effect and improve the generalization of the model. This example provides a template for applying activity regularization to your own neural network for classification and regression problems.

14.3.1 Binary Classification Problem

We will use a standard binary classification problem that defines two two-dimensional concentric circles of observations, one circle for each class. Each observation has two input variables with the same scale and a class output value of either 0 or 1. This dataset is called the *circles* dataset because of the shape of the observations in each class when plotted. We can use the `make_circles()` function to generate observations from this problem. We will add noise to the data and seed the random number generator so that the same samples are generated each time the code is run.

```
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
```

Listing 14.7: Example of creating samples for the two circles problem.

We can plot the dataset where the two variables are taken as x and y coordinates on a graph and the class value is taken as the color of the observation. The complete example of generating the dataset and plotting it is listed below.

```
# scatter plot of circles dataset
from sklearn.datasets import make_circles
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# scatter plot for each class value
for class_value in range(2):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 14.8: Example of plotting samples from the two circles problem.

Running the example creates a scatter plot showing the concentric circles shape of the observations in each class. We can see the noise in the dispersal of the points making the circles less obvious.

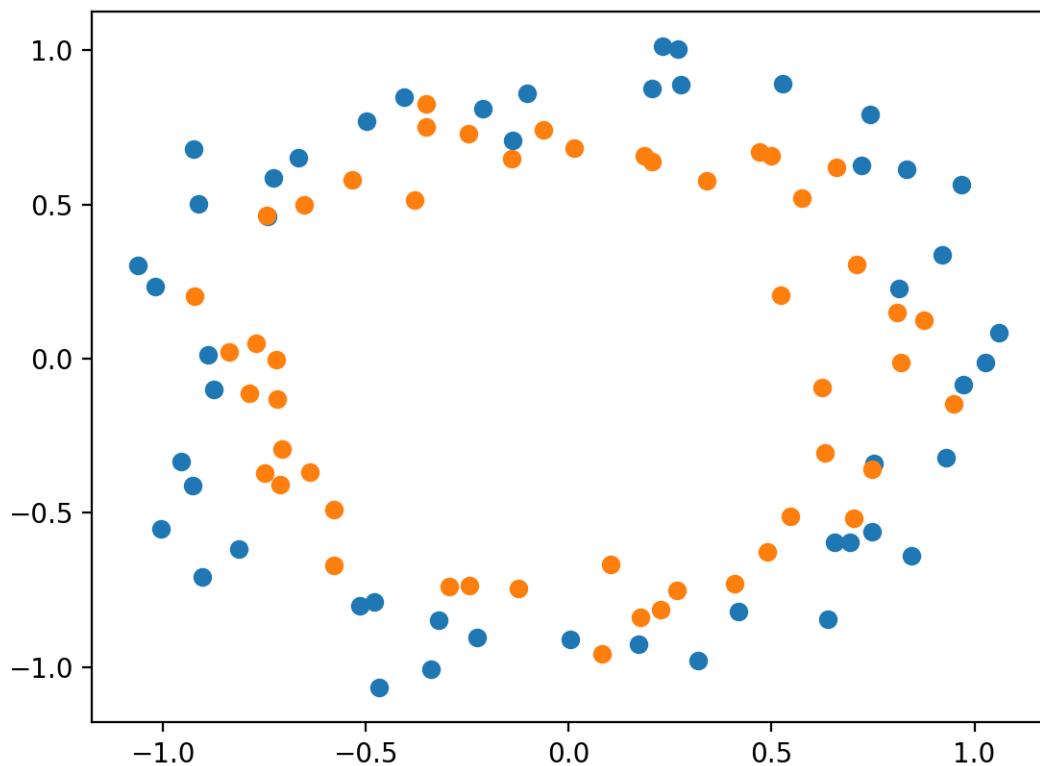


Figure 14.1: Scatter Plot of Circles Dataset with Color Showing the Class Value of Each Sample.

This is a good test problem because the classes cannot be separated by a line, e.g. are not linearly separable, requiring a nonlinear method such as a neural network to address. We have only generated 100 samples, which is small for a neural network, providing the opportunity to overfit the training dataset and have higher error on the test dataset: a good case for using regularization. Further, the samples have noise, giving the model an opportunity to learn aspects of the samples that don't generalize.

14.3.2 Overfit Multilayer Perceptron

We can develop an MLP model to address this binary classification problem. The model will have one hidden layer with more nodes that may be required to solve this problem, providing an opportunity to overfit. We will also train the model for longer than is required to ensure the model overfits. Before we define the model, we will split the dataset into train and test sets, using 30 examples to train the model and 70 to evaluate the fit model's performance.

```
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 14.9: Example of preparing the data for modeling.

Next, we can define the model. The hidden layer uses 500 nodes and the rectified linear activation function. A sigmoid activation function is used in the output layer in order to predict class values of 0 or 1. The model is optimized using the binary cross-entropy loss function, suitable for binary classification problems and the efficient Adam version of gradient descent.

```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 14.10: Example of defining the MLP model.

The defined model is then fit on the training data for 4,000 epochs and the default batch size of 32. We will also use the test dataset as a validation dataset.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
```

Listing 14.11: Example of fitting the MLP model.

We can evaluate the performance of the model on the test dataset and report the result.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 14.12: Example of evaluating the MLP model.

Finally, we will plot the performance of the model on both the train and test set each epoch. If the model does indeed overfit the training dataset, we would expect the line plot of cross-entropy loss and classification accuracy to show the pattern of overfitting. That is improvement on both train and test sets until an inflection point after which performance continues to improve for the train set and begins to get worse for the test set.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 14.13: Example of plotting learning curves for the MLP model.

We can tie all of these pieces together, the complete example is listed below.

```

# mlp overfit on the two circles dataset
from sklearn.datasets import make_circles
from keras.layers import Dense
from keras.models import Sequential
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 14.14: Example of an overfit MLP on the two circles problem.

Running the example reports the model performance on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that the model has better performance on the training dataset than the test dataset, one possible sign of overfitting.

Train: 1.000, Test: 0.757

Listing 14.15: Example output from the overfit MLP on the two circles problem.

A figure is created showing line plots of the model loss and accuracy on the train and test sets. We can see the expected shape of an overfit model where test accuracy increases to a point and then begins to decrease again. The effect is even more dramatic with loss, showing a large increase in test set loss as training continues.

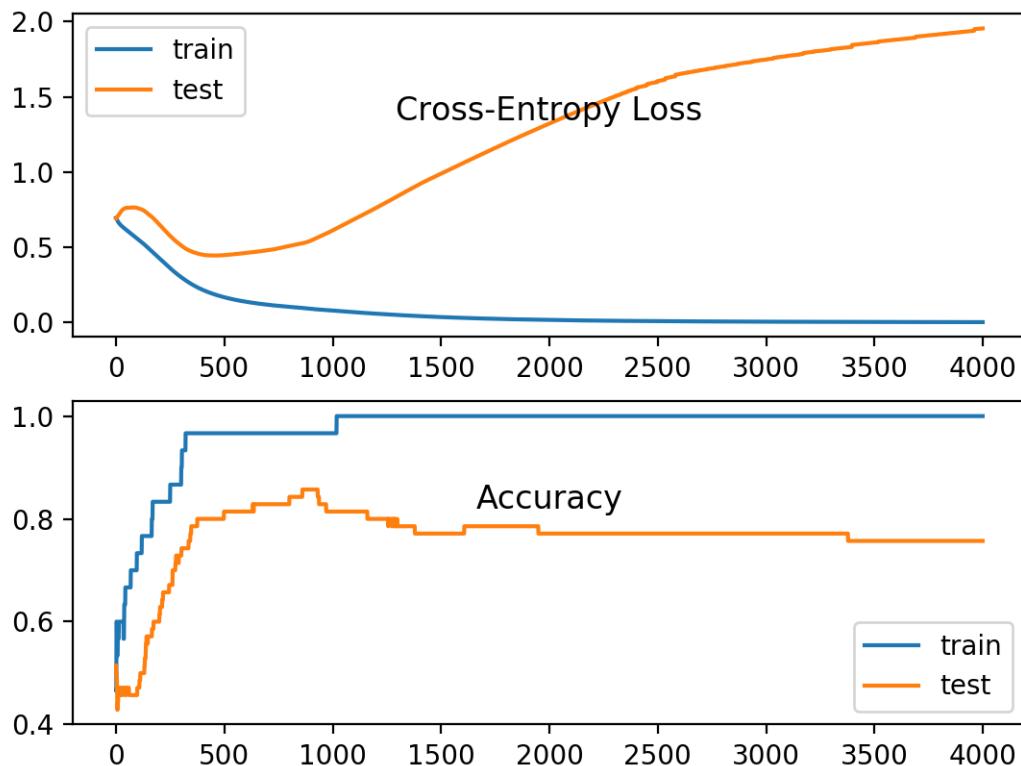


Figure 14.2: Line Plots of Accuracy on Train and Test Datasets While Training Showing an Overfit.

14.3.3 Overfit MLP With Activation Regularization

We can update the example to use activation regularization. There are a few different regularization methods to choose from, but it is probably a good idea to use the most common, which is the L1 vector norm. This regularization has the effect of encouraging a sparse representation (lots of zeros), which is supported by the rectified linear activation function that permits true zero values. We can do this by using the `keras.regularizers.l1` class in Keras.

We will configure the layer to use the linear activation function so that we can regularize the raw outputs, then add a `relu` activation layer after the regularized outputs of the layer. We will set the regularization hyperparameter to 1E-4 or 0.0001, found with a little trial and error.

```
# model.add(Dense(500, input_dim=2, activation='linear', activity_regularizer=l1(0.0001)))
model.add(Activation('relu'))
```

Listing 14.16: Example of adding activity regularization before activation.

The complete updated example with the L1 norm constraint is listed below:

```
# mlp overfit on the two circles dataset with activation regularization before activation
from sklearn.datasets import make_circles
from keras.layers import Dense
from keras.models import Sequential
```

```

from keras.regularizers import l1
from keras.layers import Activation
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='linear', activity_regularizer=l1(0.0001)))
model.add(Activation('relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 14.17: Example of an MLP with activity regularization before activation on the two circles problem.

Running the example reports the model performance on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that activity regularization resulted in a slight drop in accuracy on the training dataset down from 100% to 96% and a lift in accuracy on the test set up from 78% to 81%.

Train: 0.967, Test: 0.814

Listing 14.18: Example output from an MLP with activity regularization before activation on the two circles problem.

Reviewing the line plot of train and test accuracy, we can see that it no longer appears that the model has overfit the training dataset, at least not as strongly. Model accuracy on both the train and test sets continues to increase to a plateau.

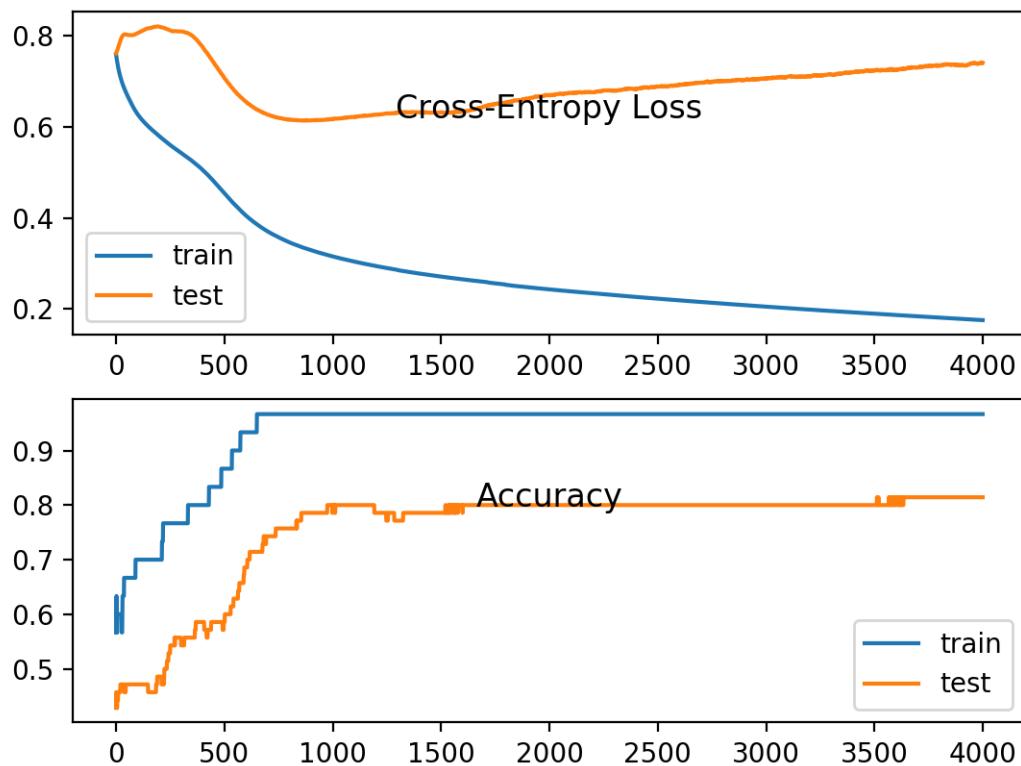


Figure 14.3: Line Plots of Accuracy on Train and Test Datasets While Training With Activity Regularization.

For completeness, we can compare results to a version of the model where activity regularization is applied after the `relu` activation function.

```
model.add(Dense(500, input_dim=2, activation='relu', activity_regularizer=l1(0.0001)))
```

Listing 14.19: Example of adding activity regularization after activation.

The complete example is listed below.

```
# mlp overfit on the two circles dataset with activation regularization after activation
from sklearn.datasets import make_circles
from keras.layers import Dense
from keras.models import Sequential
from keras.regularizers import l1
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
```

```

model.add(Dense(500, input_dim=2, activation='relu', activity_regularizer=l1(0.0001)))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 14.20: Example of an MLP with activity regularization after activation on the two circles problem.

Running the example reports the model performance on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that, at least on this problem and with this model, activation regularization after the activation function did not improve generalization error; in fact, it made it worse.

Train: 1.000, Test: 0.757

Listing 14.21: Example output from an MLP with activity regularization after activation on the two circles problem.

Reviewing the line plot of train and test accuracy, we can see that indeed the model still shows the signs of having overfit the training dataset.

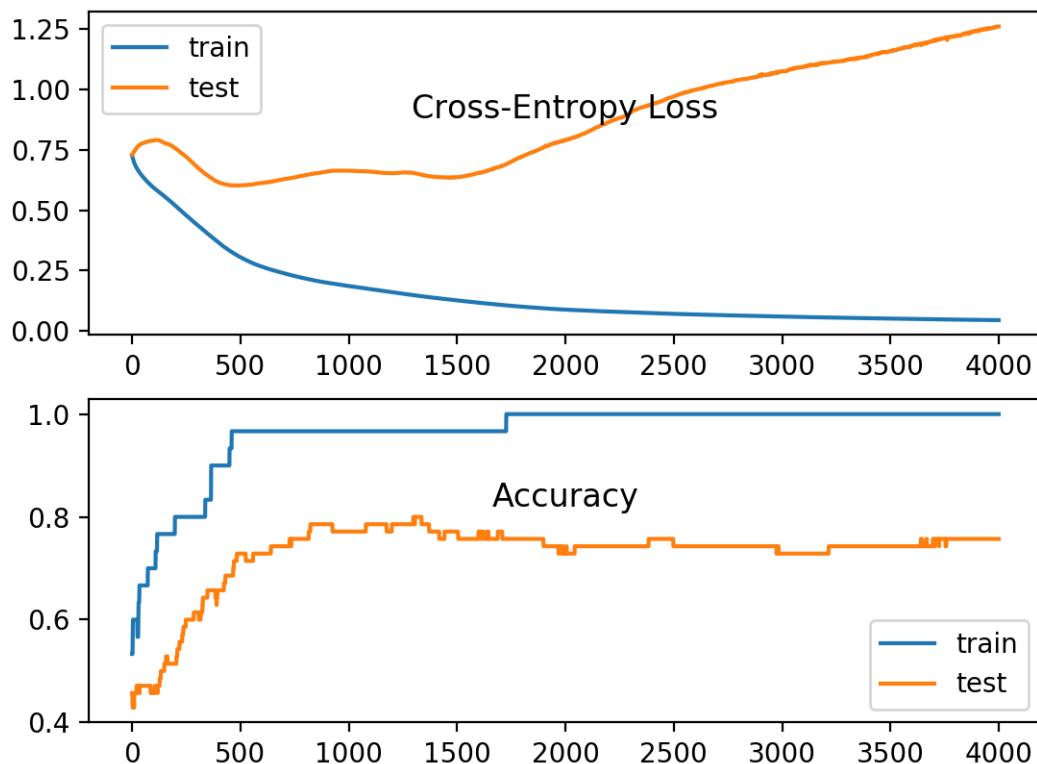


Figure 14.4: Line Plots of Accuracy on Train and Test Datasets While Training With Activity Regularization, Still Overfit.

This suggests that it may be worth experimenting with both approaches for implementing activity regularization with your own dataset, to confirm that you are getting the most out of the method.

14.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Report Activation Mean.** Update the example to calculate the mean activation of the regularized layer and confirm that indeed the activations have been made more sparse.
- **Grid Search.** Update the example to grid search different values for the regularization hyperparameter.
- **Alternate Norm.** Update the example to evaluate the L2 or L1 and L2 vector norm for regularizing the hidden layer outputs.
- **Repeated Evaluation.** Update the example to fit and evaluate the model multiple times and report the mean and standard deviation of model performance.

If you explore any of these extensions, I'd love to know.

14.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

14.5.1 Books

- Section 7.10 Sparse Representations, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>

14.5.2 Papers

- *Deep Sparse Rectifier Neural Networks*, 2011.
<http://proceedings.mlr.press/v15/glorot11a.html>
- *Sparse Feature Learning for Deep Belief Networks*, 2007.
<http://papers.nips.cc/paper/3363-sparse-feature-learning-for-deep-belief-networks>
- *Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition*, 2007.
<https://ieeexplore.ieee.org/document/4270182/>
- *Efficient sparse coding algorithms*, 2007.
<https://dl.acm.org/citation.cfm?id=2976557>
- *Measuring Invariances in Deep Networks*, 2009.
<https://papers.nips.cc/paper/3790-measuring-invariances-in-deep-networks>
- *Sparse deep belief net model for visual area V2*, 2007.
<https://papers.nips.cc/paper/3313-sparse-deep-belief-net-model-for-visual-area-v2>
- *Revisiting Activation Regularization for Language RNNs*, 2017.
<https://arxiv.org/abs/1708.01009>
- *Sparse Activity and Sparse Connectivity in Supervised Learning*, 2013.
<http://jmlr.org/papers/v14/thom13a.html>

14.5.3 APIs

- Keras Regularizers API.
<https://keras.io/regularizers/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>
- Keras Recurrent Layers API.
<https://keras.io/layers/recurrent/>

- `sklearn.datasets.make_circles` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html

14.5.4 Articles

- Sparse coding, Scholarpedia.
http://www.scholarpedia.org/article/Sparse_coding
- Sparse autoencoder, CS294A Lecture notes.
<https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf>

14.6 Summary

In this tutorial, you discovered activation regularization as a technique to improve the generalization of learned features. Specifically, you learned:

- Neural networks learn features from data and models, such as autoencoders and encoder-decoder models, and explicitly seek effective learned representations.
- Similar to weights, large values in learned features, e.g. large activations, may indicate an overfit model.
- The addition of penalties to the loss function that penalize a model in proportion to the magnitude of the activations may result in more robust and generalized learned features.

14.6.1 Next

In the next tutorial, you will discover how to update the model and add a constraint that ensures that model weights are kept small during training.

Chapter 15

Force Small Weights with Weight Constraints

Weight regularization methods like weight decay introduce a penalty to the loss function when training a neural network to encourage the network to use small weights. Smaller weights in a neural network can result in a model that is more stable and less likely to overfit the training dataset, in turn having better performance when making a prediction on new data. Unlike weight regularization, a weight constraint is a trigger that checks the size or magnitude of the weights and scales them so that they are all below a pre-defined threshold. The constraint forces weights to be small and can be used instead of weight decay and in conjunction with more aggressive network configurations, such as very large learning rates. In this tutorial, you will discover the use of weight constraint regularization as an alternative to weight penalties to reduce overfitting in deep neural networks.

After reading this tutorial, you will know:

- Weight penalties encourage but do not require neural networks to have small weights.
- Weight constraints, such as the L2 norm and maximum norm, can be used to force neural networks to have small weights during training.
- Weight constraints can improve generalization when used in conjunction with other regularization methods like dropout.

Let's get started.

15.1 Weight Constraints

In this section you will discover the problem with neural networks that have large weights, a technique that you can use to force the development of models with small weights called weight constraints and tips for using this technique in your own projects.

15.1.1 Alternative to Penalties for Large Weights

Large weights in a neural network are a sign of overfitting. A network with large weights has very likely learned the statistical noise in the training data. This results in a model that is

unstable, and very sensitive to changes to the input variables. In turn, the overfit network has poor performance when making predictions on new unseen data. A popular and effective technique to address the problem is to update the loss function that is optimized during training to take the size of the weights into account.

This is called a penalty, as the larger the weights of the network become, the more the network is penalized, resulting in larger loss and, in turn, larger updates. The effect is that the penalty encourages weights to be small, or no larger than is required during the training process, in turn reducing overfitting. A problem in using a penalty is that although it does encourage the network toward smaller weights, it does not force smaller weights. A neural network trained with weight regularization penalty may still allow large weights, in some cases very large weights.

15.1.2 Force Small Weights

An alternate solution to using a penalty for the size of network weights is to use a weight constraint. A weight constraint is an update to the network that checks the size of the weights (e.g. their vector norm), and if the size exceeds a predefined limit, the weights are rescaled so that their size is below the limit or between a range. You can think of a weight constraint as an if-then rule checking the size of the weights while the network is being trained and only coming into effect and making weights small when required. Note, for efficiency, it does not have to be implemented as an if-then rule and often is not.

Unlike adding a penalty to the loss function, a weight constraint ensures the weights of the network are small, instead of merely encouraging them to be small. It can be useful on those problems or with networks that resist other regularization methods, such as weight penalties. Weight constraints prove especially useful when you have configured your network to use alternative regularization methods to weight regularization and yet still desire the network to have small weights in order to reduce overfitting. One often-cited example is the use of a weight constraint regularization with dropout regularization.

Although dropout alone gives significant improvements, using dropout along with [weight constraint] regularization, [...] provides a significant boost over just using dropout.

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

15.1.3 How to Use a Weight Constraint

A constraint is enforced on each node within a layer. All nodes within the layer use the same constraint, and often multiple hidden layers within the same network will use the same constraint. Recall that when we talk about the vector norm in general, that this is the magnitude of the vector of weights in a node, and by default is calculated as the L2 norm, e.g. the square root of the sum of the squared values in the vector. Some examples of constraints that could be used include:

- Force the vector norm to be 1.0 (e.g. the unit norm).
- Limit the maximum size of the vector norm (e.g. the maximum norm).

- Limit the minimum and maximum size of the vector norm (e.g. the min_max norm).

The maximum norm, also called max-norm or maxnorm, is a popular constraint because it is less aggressive than other norms such as the unit norm, simply setting an upper bound.

Max-norm regularization has been previously used [...] It typically improves the performance of stochastic gradient descent training of deep neural nets ...

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

When using a limit or a range, a hyperparameter must be specified. Given that weights are small, the hyperparameter too is often a small integer value, such as a value between 1 and 4.

... we can use max-norm regularization. This constrains the norm of the vector of incoming weights at each hidden unit to be bound by a constant c . Typical values of c range from 3 to 4.

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

If the norm exceeds the specified range or limit, the weights are rescaled or normalized such that their magnitude is below the specified parameter or within the specified range.

If a weight-update violates this constraint, we renormalize the weights of the hidden unit by division. Using a constraint rather than a penalty prevents weights from growing very large no matter how large the proposed weight-update is.

— *Improving neural networks by preventing co-adaptation of feature detectors*, 2012.

The constraint can be applied after each update to the weights, e.g. at the end of each minibatch.

15.1.4 Example Uses of Weight Constraints

This section provides a few cherry-picked examples from recent research papers where a weight constraint was used. Geoffrey Hinton, et al. in their 2012 paper titled *Improving neural networks by preventing co-adaptation of feature detectors* used a maxnorm constraint on CNN models applied to the MNIST handwritten digit classification task and ImageNet photo classification task.

All layers had L2 weight constraints on the incoming weights of each hidden unit.

Nitish Srivastava, et al. in their 2014 paper titled *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* used a maxnorm constraint with an MLP on the MNIST handwritten digit classification task and with CNNs on the streetview house numbers dataset with the parameter configured via a holdout validation set.

Max-norm regularization was used for weights in both convolutional and fully connected layers.

Jan Chorowski, et al. in their 2015 paper titled *Attention-Based Models for Speech Recognition* use LSTM and attention models for speech recognition with a max norm constraint set to 1.

We first trained our models with a column norm constraint with the maximum norm

1 ...

15.1.5 Tips for Using Weight Constraints

This section provides some tips for using weight constraints with your neural network.

Use With All Network Types

Weight constraints are a generic approach. They can be used with most, perhaps all, types of neural network models, not least the most common network types of Multilayer Perceptrons, Convolutional Neural Networks, and Long Short-Term Memory Recurrent Neural Networks. In the case of LSTMs, it may be desirable to use different constraints or constraint configurations for the input and recurrent connections.

Standardize Input Data

It is a good general practice to rescale input variables to have the same scale. When input variables have different scales, the scale of the weights of the network will, in turn, vary accordingly. This introduces a problem when using weight constraints because large weights will cause the constraint to trigger more frequently. This problem can be done by either normalization or standardization of input variables.

Use a Larger Learning Rate

The use of a weight constraint allows you to be more aggressive during the training of the network. Specifically, a larger learning rate can be used, allowing the network to, in turn, make larger updates to the weights each update. This is cited as an important benefit to using weight constraints. Such as the use of a constraint in conjunction with dropout:

Using a constraint rather than a penalty prevents weights from growing very large no matter how large the proposed weight-update is. This makes it possible to start with a very large learning rate which decays during learning, thus allowing a far more thorough search of the weight-space than methods that start with small weights and use a small learning rate.

— *Improving neural networks by preventing co-adaptation of feature detectors*, 2012.

Try Other Constraints

Explore the use of other weight constraints, such as a minimum and maximum range, non-negative weights, and more. You may also choose to use constraints on some weights and not others, such as not using constraints on bias weights in an MLP or not using constraints on recurrent connections in an LSTM.

15.2 Weight Constraints Keras API

This section demonstrates how to use weight constraints with the Keras API.

15.2.1 Weight Constraints in Keras

The Keras API supports weight constraints. The constraints are specified per-layer, but applied and enforced per-node within the layer. Using a constraint generally involves setting the `kernel_constraint` argument on the layer for the input weights and the `bias_constraint` for the bias weights. Generally, weight constraints are not used on the bias weights. A suite of different vector norms can be used as constraints, provided as classes in the `keras.constraints` module. They are:

- **Maximum norm** (`max_norm`), to force weights to have a magnitude at or below a given limit.
- **Non-negative norm** (`non_neg`), to force weights to have a positive magnitude.
- **Unit norm** (`unit_norm`), to force weights to have a magnitude of 1.0.
- **Min-Max norm** (`min_max_norm`), to force weights to have a magnitude between a range.

For example, a constraint can be imported and instantiated:

```
# import norm
from keras.constraints import max_norm
# instantiate norm
norm = max_norm(3.0)
```

Listing 15.1: Example of creating an max norm constraint.

15.2.2 Weight Constraints on Layers

The weight norms can be used with most layers in Keras. In this section, we will look at some common examples.

MLP Weight Constraint

The example below sets a `max_norm` weight constraint on a `Dense` fully connected layer.

```
# example of max norm on a dense layer
from keras.layers import Dense
from keras.constraints import max_norm
...
model.add(Dense(32, kernel_constraint=max_norm(3), bias_constraint=max_norm(3)))
...
```

Listing 15.2: Example of adding a weight constraint to an MLP.

CNN Weight Constraint

The example below sets a `max_norm` weight constraint on a `Conv2D` convolutional layer.

```
# example of max norm on a cnn layer
from keras.layers import Conv2D
from keras.constraints import max_norm
...
model.add(Conv2D(32, (3,3), kernel_constraint=max_norm(3), bias_constraint=max_norm(3)))
...
```

Listing 15.3: Example of adding a weight constraint to a CNN.

RNN Weight Constraint

Unlike other layer types, recurrent neural networks allow you to set a weight constraint on both the input weights and bias, as well as the recurrent input weights. The constraint for the recurrent weights is set via the `recurrent_constraint` argument to the layer. The example below sets a maximum `max_norm` constraint on a LSTM layer.

```
# example of max norm on an lstm layer
from keras.layers import LSTM
from keras.constraints import max_norm
...
model.add(LSTM(32, kernel_constraint=max_norm(3), recurrent_constraint=max_norm(3),
               bias_constraint=max_norm(3)))
...
```

Listing 15.4: Example of adding a weight constraint to an LSTM.

Now that we know how to use the weight constraint API, let's look at a worked example.

15.3 Weight Constraints Case Study

In this section, we will demonstrate how to use weight constraints to reduce overfitting of an MLP on a simple binary classification problem. This example provides a template for applying weight constraints to your own neural network for classification and regression problems.

15.3.1 Binary Classification Problem

We will use a standard binary classification problem that defines two semi-circles of observations, one semi-circle for each class. Each observation has two input variables with the same scale and a class output value of either 0 or 1. This dataset is called the `moons` dataset because of the shape of the observations in each class when plotted. We can use the `make_moons()` function to generate observations from this problem. We will add noise to the data and seed the random number generator so that the same samples are generated each time the code is run.

```
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
```

Listing 15.5: Example of creating samples for the two moons problem.

We can plot the dataset where the two variables are taken as x and y coordinates on a graph and the class value is taken as the color of the observation. The complete example of generating the dataset and plotting it is listed below.

```
# scatter plot of moons dataset
from sklearn.datasets import make_moons
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# scatter plot for each class value
for class_value in range(2):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 15.6: Example of plotting samples from the two moons problem.

Running the example creates a scatter plot showing the semi-circle or moon shape of the observations in each class. We can see the noise in the dispersal of the points making the moons less obvious.

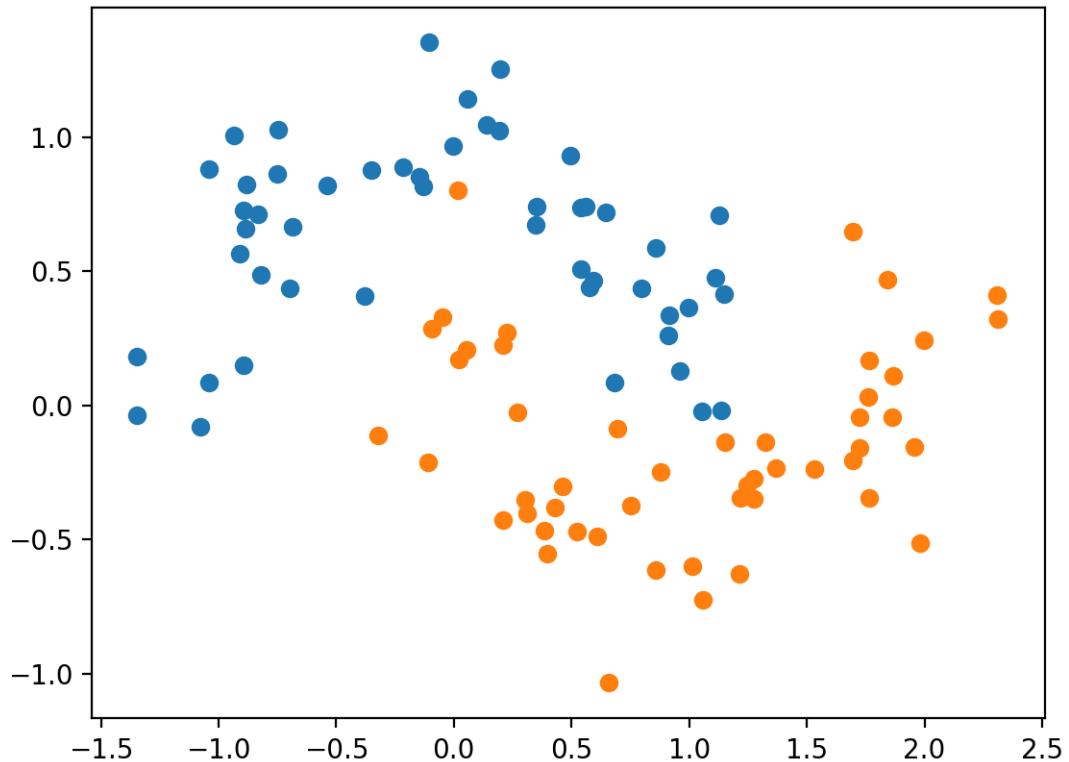


Figure 15.1: Scatter Plot of Moons Dataset With Color Showing the Class Value of Each Sample.

This is a good test problem because the classes cannot be separated by a line, e.g. are not linearly separable, requiring a nonlinear method such as a neural network to address. We have

only generated 100 samples, which is small for a neural network, providing the opportunity to overfit the training dataset and have higher error on the test dataset: a good case for using regularization. Further, the samples have noise, giving the model an opportunity to learn aspects of the samples that don't generalize.

15.3.2 Overfit Multilayer Perceptron

We can develop an MLP model to address this binary classification problem. The model will have one hidden layer with more nodes than may be required to solve this problem, providing an opportunity to overfit. We will also train the model for longer than is required to ensure the model overfits. Before we define the model, we will split the dataset into train and test sets, using 30 examples to train the model and 70 to evaluate the fit model's performance.

```
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 15.7: Example of preparing data samples for modeling.

Next, we can define the model. The hidden layer uses 500 nodes in the hidden layer and the rectified linear activation function. A sigmoid activation function is used in the output layer in order to predict class values of 0 or 1. The model is optimized using the binary cross-entropy loss function, suitable for binary classification problems and the efficient Adam version of gradient descent.

```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 15.8: Example of defining an MLP model.

The defined model is then fit on the training data for 4,000 epochs and the default batch size of 32. We will also use the test dataset as a validation dataset.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
```

Listing 15.9: Example of fitting an MLP model.

We can evaluate the performance of the model on the test dataset and report the result.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 15.10: Example of evaluating an MLP model.

Finally, we will plot the performance of the model on both the train and test set each epoch. If the model does indeed overfit the training dataset, we would expect the line plot of loss and

accuracy on the training set to continue to improve and the test set start to get worse once the model learns statistical noise in the training dataset.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 15.11: Example of plotting learning curves for an MLP model.

We can tie all of these pieces together; the complete example is listed below.

```
# mlp overfit on the moons dataset
from sklearn.datasets import make_moons
from keras.layers import Dense
from keras.models import Sequential
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 15.12: Example of an overfit MLP on the two moons problem.

Running the example reports the model performance on the train and test datasets. We can see that the model has better performance on the training dataset than the test dataset, one possible sign of overfitting.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
Train: 1.000, Test: 0.914
```

Listing 15.13: Example output from the overfit MLP on the two moons problem.

A figure is created showing line plots of the model loss and accuracy on the train and test sets. We can see that expected shape of an overfit model where test accuracy increases to a point and then begins to decrease again.

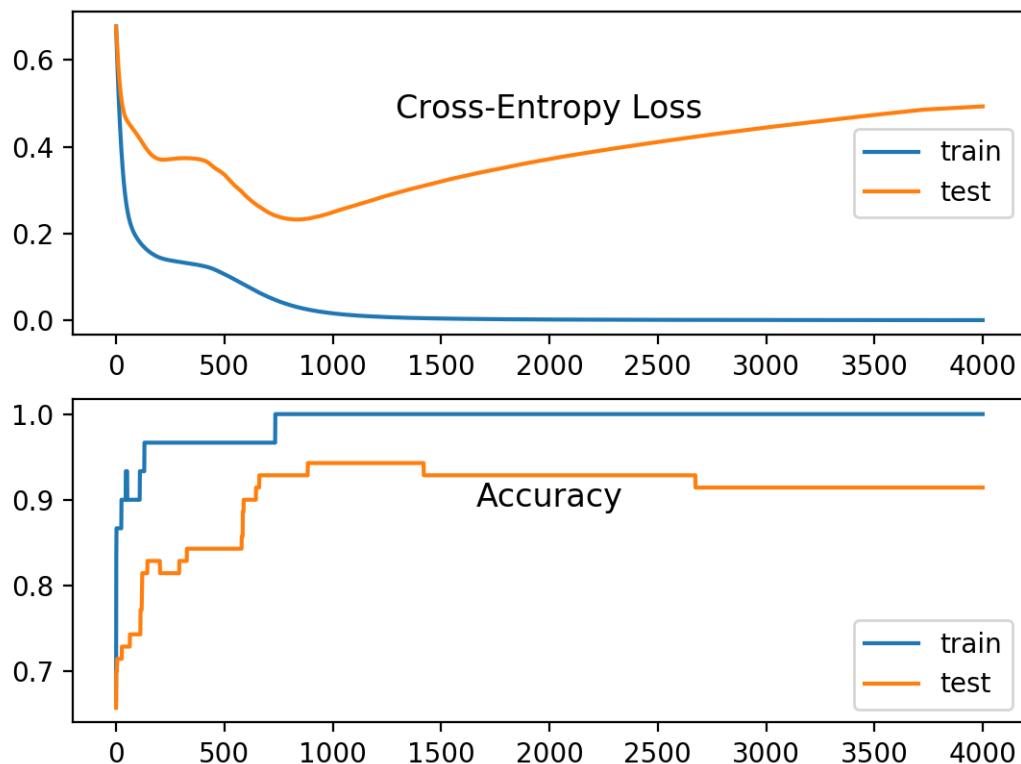


Figure 15.2: Line Plots of Accuracy on Train and Test Datasets While Training Showing an Overfit.

15.3.3 Overfit MLP With Weight Constraint

We can update the example to use a weight constraint. There are a few different weight constraints to choose from. A good simple constraint for this model is to simply normalize the weights so that the norm is equal to 1.0. This constraint has the effect of forcing all incoming weights to be small. We can do this by using the `unit_norm` in Keras. This constraint can be added to the first hidden layer as follows:

```
model.add(Dense(500, input_dim=2, activation='relu', kernel_constraint=unit_norm()))
```

Listing 15.14: Example of adding a unit norm weight constraint.

We can also achieve the same result by using the `min_max_norm` and setting the min and maximum to 1.0, for example:

```
model.add(Dense(500, input_dim=2, activation='relu',
               kernel_constraint=min_max_norm(min_value=1.0, max_value=1.0)))
```

Listing 15.15: Example of adding a min-max norm weight constraint.

We cannot achieve the same result with the maximum norm constraint as it will allow norms at or below the specified limit; for example:

```
model.add(Dense(500, input_dim=2, activation='relu', kernel_constraint=max_norm(1.0)))
```

Listing 15.16: Example of adding a max norm weight constraint.

The complete updated example with the unit norm constraint is listed below:

```
# mlp overfit on the moons dataset with a unit norm constraint
from sklearn.datasets import make_moons
from keras.layers import Dense
from keras.models import Sequential
from keras.constraints import unit_norm
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu', kernel_constraint=unit_norm()))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
```

```
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 15.17: Example of an MLP with a weight constraint on the two moons problem.

Running the example reports the model performance on the train and test datasets. We can see that indeed the strict constraint on the size of the weights has improved the performance of the model on the holdout set without impacting performance on the training set.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
Train: 1.000, Test: 0.943
```

Listing 15.18: Example output from the MLP with a weight constraint on the two moons problem.

Reviewing the line plot of train and test loss and accuracy, we can see that it no longer appears that the model has overfit the training dataset. Model accuracy on both the train and test sets continues to improve to a plateau.

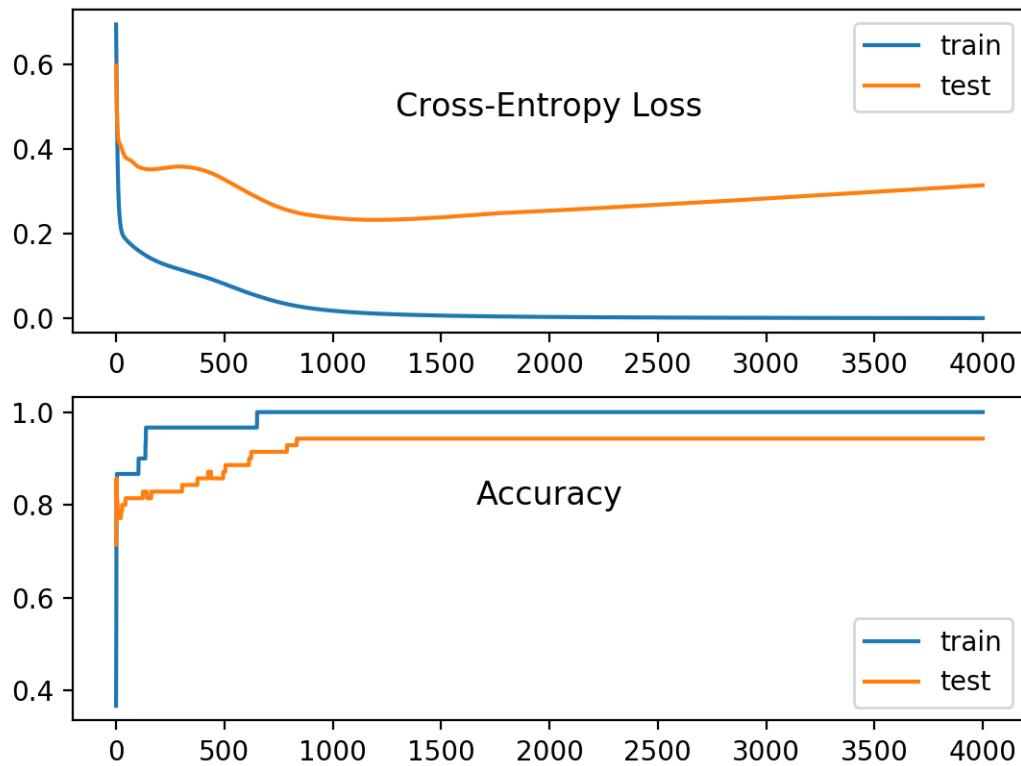


Figure 15.3: Line Plots of Accuracy on Train and Test Datasets While Training With Weight Constraints.

15.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Report Weight Norm.** Update the example to calculate the magnitude of the unit weights and demonstrate that the constraint indeed made the magnitude smaller.
- **Constrain Output Layer.** Update the example to add a constraint to the output layer of the model and compare the results.
- **Constrain Bias.** Update the example to add a constraint to the bias weight and compare the results.
- **Repeated Evaluation.** Update the example to fit and evaluate the model multiple times and report the mean and standard deviation of model performance.

If you explore any of these extensions, I'd love to know.

15.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

15.5.1 Books

- Section 7.2: Norm Penalties as Constrained Optimization, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>

15.5.2 Papers

- *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.
<http://jmlr.org/papers/v15/srivastava14a.html>
- *Rank, Trace-Norm and Max-Norm*, 2005.
https://link.springer.com/chapter/10.1007/11503415_37
- *Improving neural networks by preventing co-adaptation of feature detectors*, 2012.
<https://arxiv.org/abs/1207.0580>

15.5.3 APIs

- Keras Constraints API.
<https://keras.io/constraints/>
- Keras constraints.py.
<https://github.com/keras-team/keras/blob/master/keras/constraints.py>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>
- Keras Recurrent Layers API.
<https://keras.io/layers/recurrent/>
- sklearn.datasets.make_moons API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html

15.5.4 Articles

- Norm (mathematics), Wikipedia.
[https://en.wikipedia.org/wiki/Norm_\(mathematics\)](https://en.wikipedia.org/wiki/Norm_(mathematics))
- Regularization, Neural Networks Part 2: Setting up the Data and the Loss, CS231n Convolutional Neural Networks for Visual Recognition.
<http://cs231n.github.io/neural-networks-2/#reg>

15.6 Summary

In this tutorial, you discovered the use of weight constraint regularization as an alternative to weight penalties to reduce overfitting in deep neural networks. Specifically, you learned:

- Weight penalties encourage but do not require neural networks to have small weights.
- Weight constraints such as the L2 norm and maximum norm can be used to force neural networks to have small weights during training.
- Weight constraints can improve generalization when used in conjunction with other regularization methods, like dropout.

15.6.1 Next

In the next tutorial, you will discover dropout regularization that can decouple the layers in your neural network model.

Chapter 16

Decouple Layers with Dropout

Deep learning neural networks are likely to quickly overfit a training dataset with few examples. Ensembles of neural networks with different model configurations are known to reduce overfitting, but require the additional computational expense of training and maintaining multiple models. A single model can be used to simulate having a large number of different network architectures by randomly dropping out nodes during training. This is called dropout and offers a very computationally cheap and remarkably effective regularization method to reduce overfitting and generalization error in deep neural networks of all kinds. In this tutorial, you will discover the use of dropout regularization for reducing overfitting and improving the generalization of deep neural networks. After reading this tutorial, you will know:

- Large weights in a neural network are a sign of a more complex network that has overfit the training data.
- Probabilistically dropping out nodes in the network is a simple and effective regularization method.
- A large network with more training epochs and the use of a weight constraint are suggested when using dropout.

Let's get started.

16.1 Dropout

In this section you will discover that you can simulate the development of a large ensemble of neural network models in a single model called dropout, how you can use it to reduce overfitting, and tips for using this technique on your own projects.

16.1.1 Problem With Overfitting

Large neural nets trained on relatively small datasets can overfit the training data. This has the effect of the model learning the statistical noise in the training data, which results in poor performance when the model is evaluated on new data, e.g. a test dataset. Generalization error increases due to overfitting. One approach to reduce overfitting is to fit all possible different neural networks on the same dataset and to average the predictions from each model. This is

not feasible in practice, and can be approximated using a small collection of different models, called an ensemble.

With unlimited computation, the best way to *regularize* a fixed-sized model is to average the predictions of all possible settings of the parameters, weighting each setting by its posterior probability given the training data.

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

A problem even with the ensemble approximation is that it requires multiple models to be fit and stored, which can be a challenge if the models are large, requiring days or weeks to train and tune.

16.1.2 Randomly Drop Nodes

Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel. During training, some number of node outputs are randomly ignored or *dropped out*. This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different *view* of the configured layer.

By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs. This conceptualization suggests that perhaps dropout breaks-up situations where network layers co-adapt to correct mistakes from prior layers, in turn making the model more robust.

... units may change in a way that they fix up the mistakes of the other units. This may lead to complex co-adaptations. This in turn leads to overfitting because these co-adaptations do not generalize to unseen data. [...]

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

Dropout simulates a sparse activation from a given layer, which interestingly, in turn, encourages the network to actually learn a sparse representation as a side-effect. As such, it may be used as an alternative to activity regularization for encouraging sparse representations in autoencoder models.

We found that as a side-effect of doing dropout, the activations of the hidden units become sparse, even when no sparsity inducing regularizers are present.

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

Because the outputs of a layer under dropout are randomly subsampled, it has the effect of reducing the capacity or thinning the network during training. As such, a wider network, e.g. more nodes, may be required when using dropout.

16.1.3 How to Dropout

Dropout is implemented per-layer in a neural network. It can be used with most types of layers, such as dense fully connected layers, convolutional layers, and recurrent layers such as the long short-term memory network layer. Dropout may be implemented on any or all hidden layers in the network as well as the visible or input layer. It is not used on the output layer.

The term “dropout” refers to dropping out units (hidden and visible) in a neural network.

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

A new hyperparameter is introduced that specifies the probability at which outputs of the layer are dropped out, or inversely, the probability at which outputs of the layer are retained. The interpretation is an implementation detail that can differ from paper to code library. A common value is a probability of 0.5 for retaining the output of each node in a hidden layer and a value close to 1.0, such as 0.8, for retaining inputs from the visible layer.

In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

Dropout is not used after training when making a prediction with the fit network. The weights of the network will be larger than normal because of dropout. Therefore, before finalizing the network, the weights are first scaled by the chosen dropout rate. The network can then be used as per normal to make predictions.

If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

The rescaling of the weights can be performed at training time instead, after each weight update at the end of the minibatch. This is sometimes called *inverse dropout* and does not require any modification of weights during training. Both the Keras and PyTorch deep learning libraries implement dropout in this way.

At test time, we scale down the output by the dropout rate. [...] Note that this process can be implemented by doing both operations at training time and leaving the output unchanged at test time, which is often the way it’s implemented in practice

— Page 109, *Deep Learning With Python*, 2017.

Dropout works well in practice, perhaps replacing the need for weight regularization (e.g. weight decay) and activation regularization (e.g. representation sparsity).

... dropout is more effective than other standard computationally inexpensive regularizers, such as weight decay, filter norm constraints and sparse activity regularization. Dropout may also be combined with other forms of regularization to yield a further improvement.

— Page 265, *Deep Learning*, 2016.

16.1.4 Examples of using Dropout

This section summarizes some examples where dropout was used in recent research papers to provide a suggestion for how and where it may be used. Geoffrey Hinton, et al. in their 2012 paper that first introduced dropout titled *Improving neural networks by preventing co-adaptation of feature detectors* used the method with a range of different neural networks on different problem types achieving improved results, including handwritten digit recognition (MNIST), photo classification (CIFAR-10), and speech recognition (TIMIT).

... we use the same dropout rates - 50% dropout for all hidden units and 20% dropout for visible units

Nitish Srivastava, et al. in their 2014 journal paper introducing dropout titled *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* used dropout on a wide range of computer vision, speech recognition, and text classification tasks and found that it consistently improved performance on each problem.

We trained dropout neural networks for classification problems on data sets in different domains. We found that dropout improved generalization performance on all data sets compared to neural networks that did not use dropout.

On the computer vision problems, different dropout rates were used down through the layers of the network in conjunction with a max-norm weight constraint.

Dropout was applied to all the layers of the network with the probability of retaining the unit being $p = (0.9, 0.75, 0.75, 0.5, 0.5, 0.5)$ for the different layers of the network (going from input to convolutional layers to fully connected layers). In addition, the max-norm constraint with $c = 4$ was used for all the weights.

A simpler configuration was used for the text classification task.

We used probability of retention $p = 0.8$ in the input layers and 0.5 in the hidden layers. Max-norm constraint with $c = 4$ was used in all the layers.

Alex Krizhevsky, et al. in their famous 2012 paper titled *ImageNet Classification with Deep Convolutional Neural Networks* achieved (at the time) state-of-the-art results for photo classification on the ImageNet dataset with deep convolutional neural networks and dropout regularization.

We use dropout in the first two fully-connected layers [of the model]. Without dropout, our network exhibits substantial overfitting. Dropout roughly doubles the number of iterations required to converge.

George Dahl, et al. in their 2013 paper titled *Improving deep neural networks for LVCSR using rectified linear units and dropout* used a deep neural network with rectified linear activation functions and dropout to achieve (at the time) state-of-the-art results on a standard speech recognition task. They used a Bayesian optimization procedure to configure the choice of activation function and the amount of dropout.

... the Bayesian optimization procedure learned that dropout wasn't helpful for sigmoid nets of the sizes we trained. In general, ReLUs and dropout seem to work quite well together.

16.1.5 Tips for Using Dropout Regularization

This section provides some tips for using dropout regularization with your neural network.

Use With All Network Types

Dropout regularization is a generic approach. It can be used with most, perhaps all, types of neural network models, not least the most common network types of Multilayer Perceptrons, Convolutional Neural Networks, and Long Short-Term Memory Recurrent Neural Networks. In the case of LSTMs, it may be desirable to use different dropout rates for the input and recurrent connections.

Dropout Rate

The default interpretation of the dropout hyperparameter is the probability of training a given node in a layer, where 1.0 means no dropout, and 0.0 means no outputs from the layer. A good value for dropout in a hidden layer is between 0.5 and 0.8. Input layers use a larger dropout (retention) rate, such as of 0.8.

Use a Larger Network

It is common for larger networks (more layers or more nodes) to more easily overfit the training data. When using dropout regularization, it is possible to use larger networks with less risk of overfitting. In fact, a large network (more nodes per layer) may be required as dropout will probabilistically reduce the capacity of the network. A good rule of thumb is to divide the number of nodes in the layer before dropout by the proposed dropout rate and use that as the number of nodes in the new network that uses dropout. For example, a network with 100 nodes and a proposed dropout rate of 0.5 will require 200 nodes ($\frac{100}{0.5}$) when using dropout.

If n is the number of hidden units in any layer and p is the probability of retaining a unit [...] a good dropout net should have at least $\frac{n}{p}$ units

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

Grid Search Parameters

Rather than guess at a suitable dropout rate for your network, test different rates systematically. For example, test values between 1.0 and 0.1 in increments of 0.1. This will both help you discover what works best for your specific model and dataset, as well as how sensitive the model is to the dropout rate. A more sensitive model may be unstable and could benefit from an increase in size.

Use a Weight Constraint

Network weights will increase in size in response to the probabilistic removal of layer activations. Large weight size can be a sign of an unstable network. To counter this effect a weight constraint can be imposed to force the norm (magnitude) of all weights in a layer to be below a specified value. For example, the maximum norm constraint is recommended with a value between 3 and 4.

... we can use max-norm regularization. This constrains the norm of the vector of incoming weights at each hidden unit to be bound by a constant c . Typical values of c range from 3 to 4.

— *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.

This does introduce an additional hyperparameter that may require tuning for the model.

Use With Smaller Datasets

Like other regularization methods, dropout is more effective on those problems where there is a limited amount of training data and the model is likely to overfit the training data. Problems where there is a large amount of training data may see less benefit from using dropout.

For very large datasets, regularization confers little reduction in generalization error. In these cases, the computational cost of using dropout and larger models may outweigh the benefit of regularization.

— Page 265, *Deep Learning*, 2016.

16.2 Dropout Keras API

This section demonstrates how to use dropout with the Keras API.

16.2.1 Dropout in Keras

Keras supports dropout regularization. The simplest form of dropout in Keras is provided by a `Dropout` core layer. When created, the dropout rate can be specified to the layer as the probability of setting each input to the layer to zero. This is different from the definition of dropout rate from the papers, in which the rate refers to the probability of retaining an input. Therefore, when a dropout rate of 0.8 is suggested in a paper (retain 80%), this will, in fact, will be a dropout rate of 0.2 (set 20% of inputs to zero). Below is an example of creating a dropout layer with a 50% chance of setting inputs to zero.

```
layer = Dropout(0.5)
```

Listing 16.1: Example of creating a `Dropout` layer.

16.2.2 Dropout on Layers

The `Dropout` layer is added to a model between existing layers and applies to outputs of the prior layer that are fed to the subsequent layer. For example, given two `Dense` layers:

```
...
model.append(Dense(32))
model.append(Dense(32))
...
```

Listing 16.2: Example of two `Dense` hidden layers.

We can insert a dropout layer between them, in which case the outputs or activations of the first layer have dropout applied to them, which are then taken as input to the next layer.

```
...
model.append(Dense(32))
model.append(Dropout(0.5))
model.append(Dense(32))
...
```

Listing 16.3: Example of a `Dropout` layer between two `Dense` hidden layers.

Dropout can also be applied to the visible layer, e.g. the inputs to the network. This requires that you define the network with the `Dropout` layer as the first layer and add the `input_shape` argument to the layer to specify the expected shape of the input samples.

```
...
model.add(Dropout(0.5, input_shape=(2,)))
...
```

Listing 16.4: Example of a `Dropout` input layer.

Let's take a look at how dropout regularization can be used with some common network types.

MLP Dropout Regularization

The example below adds dropout between two `Dense` fully connected layers.

```
# example of dropout between fully connected layers
from keras.layers import Dense
from keras.layers import Dropout
...
model.add(Dense(32))
model.add(Dropout(0.5))
model.add(Dense(1))
...
```

Listing 16.5: Example of Dropout for an MLP.

CNN Dropout Regularization

Dropout can be used after convolutional layers (e.g. `Conv2D`) and after pooling layers (e.g. `MaxPooling2D`). Often, dropout is only used after the pooling layers, but this is just a rough heuristic.

```
# example of dropout for a CNN
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dropout
...
model.add(Conv2D(32, (3,3)))
model.add(Conv2D(32, (3,3)))
model.add(MaxPooling2D())
model.add(Dropout(0.5))
model.add(Dense(1))
...
```

Listing 16.6: Example of Dropout for a CNN.

In this case, dropout is applied to each element or cell within the feature maps. An alternative way to use dropout with convolutional neural networks is to dropout entire feature maps from the convolutional layer which are then not used during pooling. This is called spatial dropout (or *Spatial Dropout*).

Instead we formulate a new dropout method which we call SpatialDropout. For a given convolution feature tensor [...] [we] extend the dropout value across the entire feature map.

— *Efficient Object Localization Using Convolutional Networks*, 2015.

Spatial Dropout is provided in Keras via the `SpatialDropout2D` layer (as well as 1D and 3D versions).

```
# example of spatial dropout for a CNN
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import SpatialDropout2D
...
model.add(Conv2D(32, (3,3)))
model.add(Conv2D(32, (3,3)))
model.add(SpatialDropout2D(0.5))
model.add(MaxPooling2D())
model.add(Dense(1))
...
```

Listing 16.7: Example of Spatial Dropout for a CNN.

RNN Dropout Regularization

The example below adds dropout between two layers: an LSTM recurrent layer and a `Dense` fully connected layers.

```
# example of dropout between LSTM and fully connected layers
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
...
model.add(LSTM(32))
model.add(Dropout(0.5))
model.add(Dense(1))
...
```

Listing 16.8: Example of Dropout for an LSTM.

This example applies dropout to, in this case, 32 outputs from the LSTM layer provided as input to the Dense layer. Alternately, the inputs to the LSTM layer may be subjected to dropout. In this case, a different dropout mask is applied to each time step within each sample presented to the LSTM.

```
# example of dropout before LSTM layer
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
...
model.add(Dropout(0.5, input_shape=(...)))
model.add(LSTM(32))
model.add(Dense(1))
...
```

Listing 16.9: Example of Dropout for input to an LSTM.

There is an alternative way to use dropout with recurrent layers like the LSTM. The same dropout mask may be used by the LSTM for all inputs within a sample. The same approach may be used for recurrent input connections across the time steps of the sample. This approach to dropout with recurrent models is called a *Variational RNN*.

The proposed technique (Variational RNN [...]) uses the same dropout mask at each time step, including the recurrent layers. [...] Implementing our approximate inference is identical to implementing dropout in RNNs with the same network units dropped at each time step, randomly dropping inputs, outputs, and recurrent connections. This is in contrast to existing techniques, where different network units would be dropped at different time steps, and no dropout would be applied to the recurrent connections

— *A Theoretically Grounded Application of Dropout in Recurrent Neural Networks*, 2016.

Keras supports Variational RNNs (i.e. consistent dropout across the time steps of a sample for inputs and recurrent inputs) via two arguments on the recurrent layers, namely `dropout` for inputs and `recurrent_dropout` for recurrent inputs.

```
# example of variational LSTM dropout
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
...
```

```
model.add(LSTM(32, dropout=0.5, recurrent_dropout=0.5))
model.add(Dense(1))
...
```

Listing 16.10: Example of Dropout for an LSTM over time.

16.3 Dropout Case Study

In this section, we will demonstrate how to use dropout regularization to reduce overfitting of an MLP on a simple binary classification problem. This example provides a template for applying dropout regularization to your own neural network for classification and regression problems.

16.3.1 Binary Classification Problem

We will use a standard binary classification problem that defines two two-dimensional concentric circles of observations, one circle for each class. Each observation has two input variables with the same scale and a class output value of either 0 or 1. This dataset is called the *circles* dataset because of the shape of the observations in each class when plotted. We can use the `make_circles()` function to generate observations from this problem. We will add noise to the data and seed the random number generator so that the same samples are generated each time the code is run.

```
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
```

Listing 16.11: Example of creating samples for the two circles problem.

We can plot the dataset where the two variables are taken as x and y coordinates on a graph and the class value is taken as the color of the observation. The complete example of generating the dataset and plotting it is listed below.

```
# scatter plot of circles dataset
from sklearn.datasets import make_circles
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# scatter plot for each class value
for class_value in range(2):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 16.12: Example of plotting samples from the two circles problem.

Running the example creates a scatter plot showing the concentric circles shape of the observations in each class. We can see the noise in the dispersal of the points making the circles less obvious.

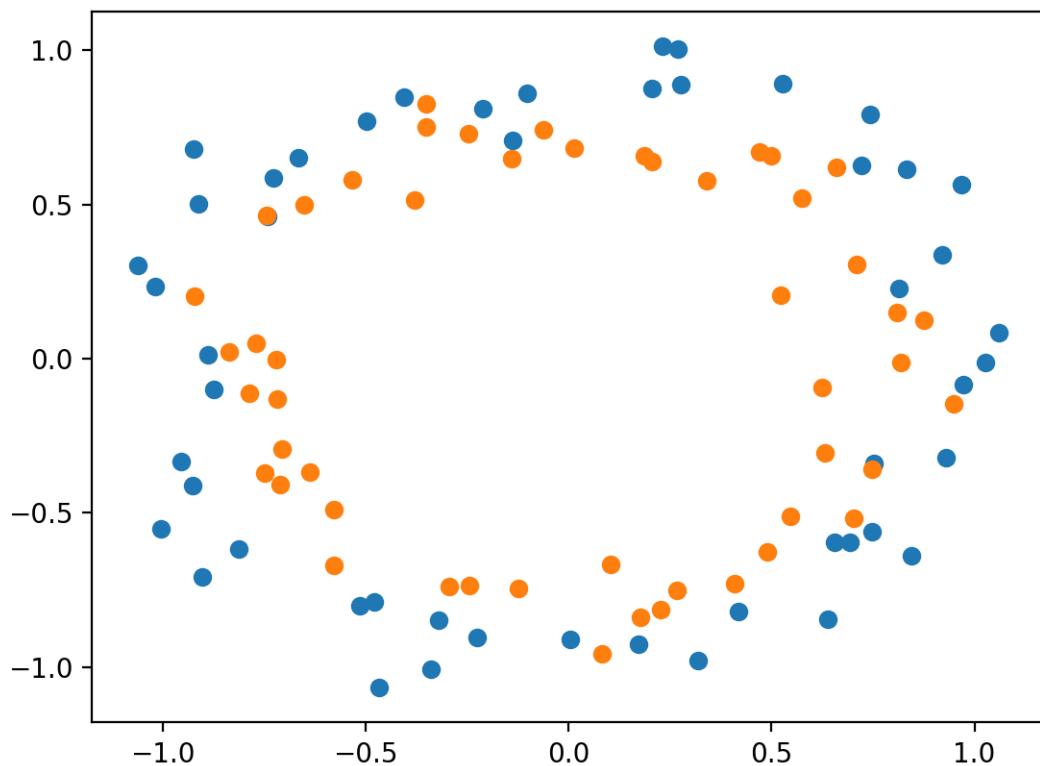


Figure 16.1: Scatter Plot of Circles Dataset with Color Showing the Class Value of Each Sample.

This is a good test problem because the classes cannot be separated by a line, e.g. are not linearly separable, requiring a nonlinear method such as a neural network to address. We have only generated 100 samples, which is small for a neural network, providing the opportunity to overfit the training dataset and have higher error on the test dataset: a good case for using regularization. Further, the samples have noise, giving the model an opportunity to learn aspects of the samples that don't generalize.

16.3.2 Overfit Multilayer Perceptron

We can develop an MLP model to address this binary classification problem. The model will have one hidden layer with more nodes than may be required to solve this problem, providing an opportunity to overfit. We will also train the model for longer than is required to ensure the model overfits. Before we define the model, we will split the dataset into train and test sets, using 30 examples to train the model and 70 to evaluate the fit model's performance.

```
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 16.13: Example of preparing the dataset for modeling.

Next, we can define the model. The hidden layer uses 500 nodes in the hidden layer and the rectified linear activation function. A sigmoid activation function is used in the output layer in order to predict class values of 0 or 1. The model is optimized using the binary cross-entropy loss function, suitable for binary classification problems and the efficient Adam version of gradient descent.

```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 16.14: Example of defining an MLP model.

The defined model is then fit on the training data for 4,000 epochs and the default batch size of 32. We will also use the test dataset as a validation dataset.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
```

Listing 16.15: Example of fitting MLP model.

We can evaluate the performance of the model on the test dataset and report the result.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 16.16: Example of evaluating fit MLP model.

Finally, we will plot the performance of the model on both the train and test set each epoch. If the model does indeed overfit the training dataset, we would expect the line plot of accuracy on the training set to continue to increase and the test set to rise and then fall again as the model learns statistical noise in the training dataset.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 16.17: Example of plotting MLP model performance.

We can tie all of these pieces together; the complete example is listed below.

```

# mlp overfit on the two circles dataset
from sklearn.datasets import make_circles
from keras.layers import Dense
from keras.models import Sequential
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 16.18: Example of MLP fit on the two circles problem.

Running the example reports the model performance on the train and test datasets. We can see that the model has better performance on the training dataset than the test dataset, one possible sign of overfitting.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
Train: 1.000, Test: 0.771
```

Listing 16.19: Example output fitting an MLP on the two circles problem.

A figure is created showing line plots of the model performance on the train and test sets. We can see that expected shape of an overfit model where test loss and accuracy improve to a point and then begin get worse as training continues.

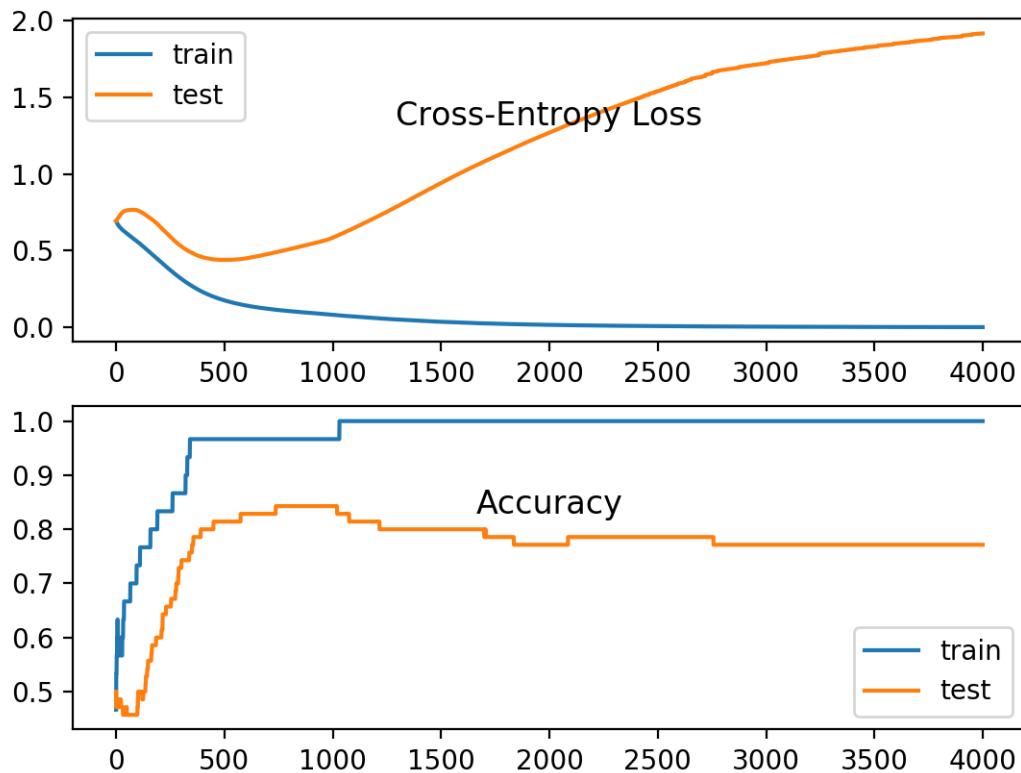


Figure 16.2: Line Plots of Accuracy on Train and Test Datasets While Training Showing an Overfit.

16.3.3 MLP With Dropout Regularization

We can update the example to use dropout regularization. We can do this by simply inserting a new Dropout layer between the hidden layer and the output layer. In this case, we will specify a dropout rate (probability of setting outputs from the hidden layer to zero) to 40% or 0.4.

```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 16.20: Example of MLP with dropout.

The complete updated example with the addition of dropout after the hidden layer is listed below:

```
# mlp with dropout on the two circles dataset
from sklearn.datasets import make_circles
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
```

```

from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 16.21: Example of MLP with dropout fit on the two circles problem.

Running the example reports the model performance on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this specific case, we can see that dropout resulted in a slight drop in accuracy on the training dataset, down from 100% to 96%, and a lift in accuracy on the test set, up from 77% to 81%.

Train: 0.967, Test: 0.814

Listing 16.22: Example output fitting an MLP with dropout on the two circles problem.

Reviewing the line plot of train and test accuracy during training, we can see that it no longer appears that the model has overfit the training dataset. Model accuracy on both the train and test sets continues to increase to a plateau, albeit with a lot of noise given the use of dropout during training.

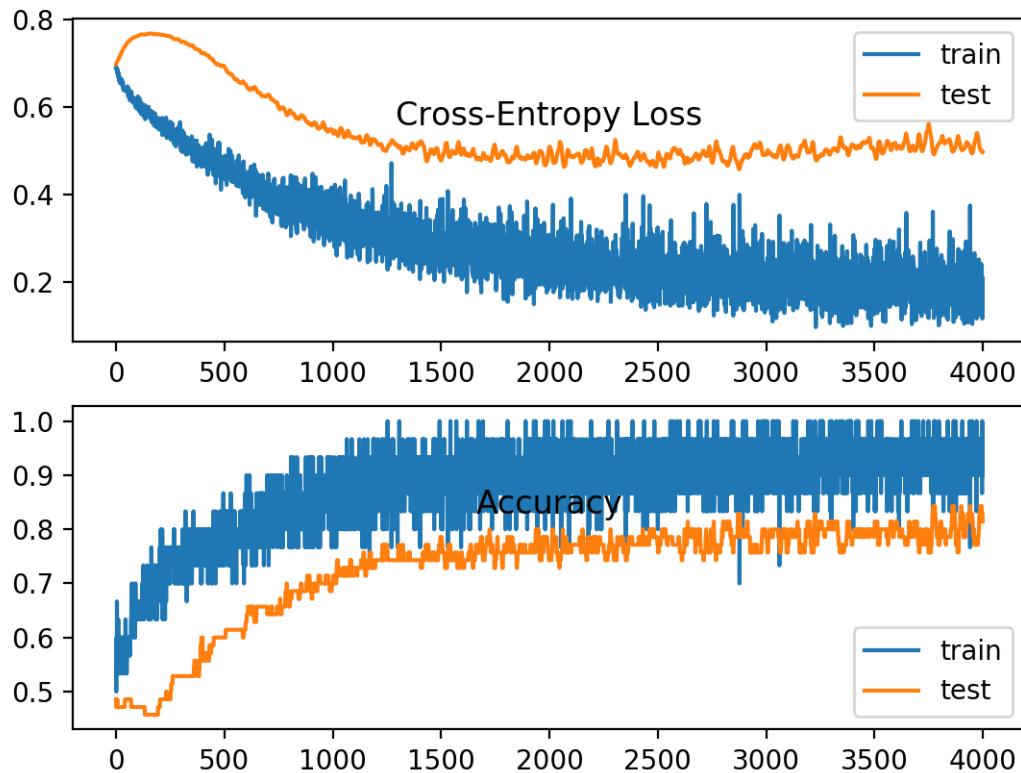


Figure 16.3: Line Plots of Accuracy on Train and Test Datasets While Training With Dropout Regularization.

16.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Input Dropout.** Update the example to use dropout on the input variables and compare results.
- **Weight Constraint.** Update the example to add a max-norm weight constraint to the hidden layer and compare results.
- **Repeated Evaluation.** Update the example to repeat the evaluation of the overfit and dropout model and summarize and compare the average results.
- **Grid Search Rate.** Develop a grid search of dropout probabilities and report the relationship between dropout rate and test set accuracy.

If you explore any of these extensions, I'd love to know.

16.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

16.5.1 Books

- Section 7.12: Dropout, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- Section 4.4.3: Adding dropout, *Deep Learning With Python*, 2017.
<https://amzn.to/2wVqZDq>

16.5.2 Papers

- *Improving neural networks by preventing co-adaptation of feature detectors*, 2012.
<https://arxiv.org/abs/1207.0580>
- *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, 2014.
<http://jmlr.org/papers/v15/srivastava14a.html>
- *Improving deep neural networks for LVCSR using rectified linear units and dropout*, 2013.
<https://ieeexplore.ieee.org/document/6639346/>
- *Dropout Training as Adaptive Regularization*, 2013.
<https://arxiv.org/abs/1307.1493>
- *Efficient Object Localization Using Convolutional Networks*, 2015.
<https://arxiv.org/abs/1411.4280>
- *A Theoretically Grounded Application of Dropout in Recurrent Neural Networks*, 2016.
<https://arxiv.org/abs/1512.05287>

16.5.3 API

- Keras Regularizers API.
<https://keras.io/regularizers/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>
- Keras Recurrent Layers API.
<https://keras.io/layers/recurrent/>
- `sklearn.datasets.make_circles` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html

16.5.4 Articles

- Dropout (neural networks), Wikipedia.
[https://en.wikipedia.org/wiki/Dropout_\(neural_networks\)](https://en.wikipedia.org/wiki/Dropout_(neural_networks))
- Regularization, CS231n Convolutional Neural Networks for Visual Recognition.
<http://cs231n.github.io/neural-networks-2/#reg>
- How was ‘Dropout’ conceived? Was there an ‘aha’ moment?
https://www.reddit.com/r/MachineLearning/comments/4w6tsv/ama_we_are_the_google_brain_team_wed_love_to/d64yyas

16.6 Summary

In this tutorial, you discovered the use of dropout regularization for reducing overfitting and improving the generalization of deep neural networks. Specifically, you learned:

- Large weights in a neural network are a sign of a more complex network that has overfit the training data.
- Probabilistically dropping out nodes in the network is a simple and effective regularization method.
- A large network with more training epochs and the use of a weight constraint are suggested when using dropout.

16.6.1 Next

In the next tutorial, you will discover how to improve neural network model robustness by adding statistical noise.

Chapter 17

Promote Robustness with Noise

Training a neural network with a small dataset can cause the network to memorize all training examples, in turn leading to poor performance on a holdout dataset. Small datasets may also represent a harder mapping problem for neural networks to learn, given the patchy or sparse sampling of points in the high-dimensional input space. One approach to making the input space smoother and easier to learn is to add noise to inputs during training. In this tutorial, you will discover that adding noise to a neural network during training can improve the robustness of the network, resulting in better generalization and faster learning. After reading this tutorial, you will know:

- Small datasets can make learning challenging for neural nets and the examples can be memorized.
- Adding noise during training can make the training process more robust and reduce generalization error.
- Noise is traditionally added to the inputs, but can also be added to weights, gradients, and even activation functions.

Let's get started.

17.1 Noise Regularization

In this section you will discover the brittleness of large network weights and how the addition of statistical noise can provide a regularizing effect, as well as tips to help when adding noise to your own neural network models.

17.1.1 Challenge of Small Training Datasets

Small datasets can introduce problems when training large neural networks. The first problem is that the network may effectively memorize the training dataset. Instead of learning a general mapping from inputs to outputs, the model may learn the specific input examples and their associated outputs. This will result in a model that performs well on the training dataset, and poor on new data, such as a holdout dataset. The second problem is that a small dataset provides less opportunity to describe the structure of the input space and its relationship to the

output. More training data provides a richer description of the problem from which the model may learn. Fewer data points means that rather than a smooth input space, the points may represent a jarring and disjointed structure that may result in a difficult, if not unlearnable, mapping function. It is not always possible to acquire more data. Further, getting a hold of more data may not address these problems.

17.1.2 Add Random Noise During Training

One approach to improving generalization error and to improving the structure of the mapping problem is to add random noise.

Many studies [...] have noted that adding small amounts of input noise (jitter) to the training data often aids generalization and fault tolerance.

— Page 273, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.

At first, this sounds like a recipe for making learning more challenging. It is a counter-intuitive suggestion to improving performance because one would expect noise to degrade performance of the model during training.

Heuristically, we might expect that the noise will ‘smear out’ each data point and make it difficult for the network to fit individual data points precisely, and hence will reduce over-fitting. In practice, it has been demonstrated that training with noise can indeed lead to improvements in network generalization.

— Page 347, *Neural Networks for Pattern Recognition*, 1995.

The addition of noise during the training of a neural network model has a regularization effect and, in turn, improves the robustness of the model. It has been shown to have a similar impact on the loss function as the addition of a penalty term, as in the case of weight regularization methods.

It is well known that the addition of noise to the input data of a neural network during training can, in some circumstances, lead to significant improvements in generalization performance. Previous work has shown that such training with noise is equivalent to a form of regularization in which an extra term is added to the error function.

— *Training with Noise is Equivalent to Tikhonov Regularization*, 2008.

In effect, adding noise expands the size of the training dataset. Each time a training sample is exposed to the model, random noise is added to the input variables making them different every time it is exposed to the model. In this way, adding noise to input samples is a simple form of data augmentation.

Injecting noise in the input to a neural network can also be seen as a form of data augmentation.

— Page 241, *Deep Learning*, 2016.

Adding noise means that the network is less able to memorize training samples because they are changing all of the time, resulting in smaller network weights and a more robust network that has lower generalization error. The noise means that it is as though new samples are being drawn from the domain in the vicinity of known samples, smoothing the structure of the input space. This smoothing may mean that the mapping function is easier for the network to learn, resulting in better and faster learning.

... input noise and weight noise encourage the neural-network output to be a smooth function of the input or its weights, respectively.

— *The Effects of Adding Noise During Backpropagation Training on a Generalization Performance*, 1996.

17.1.3 How and Where to Add Noise

The most common type of noise used during training is the addition of Gaussian noise to input variables. Gaussian noise, or white noise, has a mean of zero and a standard deviation of one and can be generated as needed using a pseudorandom number generator. The addition of Gaussian noise to the inputs to a neural network was traditionally referred to as *jitter* or *random jitter* after the use of the term in signal processing to refer to the uncorrelated random noise in electrical circuits. The amount of noise added (e.g. the spread or standard deviation) is a configurable hyperparameter. Too little noise has no effect, whereas too much noise makes the mapping function too challenging to learn.

This is generally done by adding a random vector onto each input pattern before it is presented to the network, so that, if the patterns are being recycled, a different random vector is added each time.

— *Training with Noise is Equivalent to Tikhonov Regularization*, 2008.

The standard deviation of the random noise controls the amount of spread and can be adjusted based on the scale of each input variable. It can be easier to configure if the scale of the input variables has first been normalized. Noise is only added during training. No noise is added during the evaluation of the model or when the model is used to make predictions on new data. The addition of noise is also an important part of automatic feature learning, such as in the case of autoencoders, so-called denoising autoencoders that explicitly require models to learn robust features in the presence of noise added to inputs.

We have seen that the reconstruction criterion alone is unable to guarantee the extraction of useful features as it can lead to the obvious solution “simply copy the input” or similarly uninteresting ones that trivially maximizes mutual information. [...] we change the reconstruction criterion for a both more challenging and more interesting objective: cleaning partially corrupted input, or in short denoising.

— *Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion*, 2010.

Although additional noise to the inputs is the most common and widely studied approach, random noise can be added to other parts of the network during training. Some examples include:

- **Add noise to activations**, i.e. the outputs of each layer.
- **Add noise to weights**, i.e. an alternative to the inputs.
- **Add noise to the gradients**, i.e. the direction to update weights.
- **Add noise to the outputs**, i.e. the labels or target variables.

The addition of noise to the layer activations allows noise to be used at any point in the network. This can be beneficial for very deep networks. Noise can be added to the layer outputs themselves, but this is more likely achieved via the use of a noisy activation function. The addition of noise to weights allows the approach to be used throughout the network in a consistent way instead of adding noise to inputs and layer activations. This is particularly useful in recurrent neural networks.

Another way that noise has been used in the service of regularizing models is by adding it to the weights. This technique has been used primarily in the context of recurrent neural networks. [...] Noise applied to the weights can also be interpreted as equivalent (under some assumptions) to a more traditional form of regularization, encouraging stability of the function to be learned.

— Page 242, *Deep Learning*, 2016.

The addition of noise to gradients focuses more on improving the robustness of the optimization process itself rather than the structure of the input domain. The amount of noise can start high at the beginning of training and decrease over time, much like a decaying learning rate. This approach has proven to be an effective method for very deep networks and for a variety of different network types.

We consistently see improvement from injected gradient noise when optimizing a wide variety of models, including very deep fully-connected networks, and special-purpose architectures for question answering and algorithm learning. [...] Our experiments indicate that adding annealed Gaussian noise by decaying the variance works better than using fixed Gaussian noise

— *Adding Gradient Noise Improves Learning for Very Deep Networks*, 2015.

Adding noise to the activations, weights, or gradients all provide a more generic approach to adding noise that is invariant to the types of input variables provided to the model. If the problem domain is believed or expected to have mislabeled examples, then the addition of noise to the class label can improve the model's robustness to this type of error. Although, it can be easy to derail the learning process. Adding noise to a continuous target variable in the case of regression or time series forecasting is much like the addition of noise to the input variables and may be a better use case.

17.1.4 Examples of Adding Noise During Training

This section summarizes some examples where the addition of noise during training has been used. Lasse Holmstrom studied the addition of random noise both analytically and experimentally with MLPs in the 1992 paper titled *Using Additive Noise in Back-Propagation Training*. They recommend first standardizing input variables then using cross-validation to choose the amount of noise to use during training.

If a single general-purpose noise design method should be suggested, we would pick maximizing the cross-validated likelihood function. This method is easy to implement, is completely data-driven, and has a validity that is supported by theoretical consistency results

Klaus Gref, et al. in their 2016 paper titled *LSTM: A Search Space Odyssey* used a hyperparameter search for the standard deviation for Gaussian noise on the input variables for a suite of sequence prediction tasks and found that it almost universally resulted in worse performance.

Additive Gaussian noise on the inputs, a traditional regularizer for neural networks, has been used for LSTM as well. However, we find that not only does it almost always hurt performance, it also slightly increases training times.

Alex Graves, et al. in their groundbreaking 2013 paper titled *Speech recognition with deep recurrent neural networks* that achieved then state-of-the-art results for speech recognition added noise to the weights of LSTMs during training.

... weight noise [was used] (the addition of Gaussian noise to the network weights during training). Weight noise was added once per training sequence, rather than at every timestep. Weight noise tends to ‘simplify’ neural networks, in the sense of reducing the amount of information required to transmit the parameters, which improves generalisation.

In a prior 2011 paper that studies different types of static and adaptive weight noise titled *Practical Variational Inference for Neural Networks*, Graves recommends using early stopping in conjunction with the addition of weight noise with LSTMs.

... in practice early stopping is required to prevent overfitting when training with weight noise.

17.1.5 Tips for Adding Noise During Training

This section provides some tips for adding noise during training with your neural network.

Problem Types for Adding Noise

Noise can be added to training regardless of the type of problem that is being addressed. It is appropriate to try adding noise to both classification and regression type problems. The type of noise can be specialized to the types of data used as input to the model, for example, two-dimensional noise in the case of images and signal noise in the case of audio data.

Add Noise to Different Network Types

Adding noise during training is a generic method that can be used regardless of the type of neural network that is being used. It was a method used primarily with Multilayer Perceptrons given their prior dominance, but can be and is used with Convolutional and Recurrent Neural Networks.

Rescale Data First

It is important that the addition of noise has a consistent effect on the model. This requires that the input data is rescaled so that all variables have the same scale, so that when noise is added to the inputs with a fixed variance, it has the same effect. This also applies to adding noise to weights and gradients as they too are affected by the scale of the inputs. This can be achieved via standardization or normalization of input variables. If random noise is added after data scaling, then the variables may need to be rescaled again, perhaps per minibatch.

Test the Amount of Noise

You cannot know how much noise will benefit your specific model on your training dataset. Experiment with different amounts, and even different types of noise, in order to discover what works best. Be systematic and use controlled experiments, perhaps on smaller datasets across a range of values.

Noisy Training Only

Noise is only added during the training of your model. Be sure that any source of noise is not added during the evaluation of your model, or when your model is used to make predictions on new data.

17.2 Noise Regularization Keras API

This section demonstrates how to add noise with the Keras API.

17.2.1 Noise Regularization in Keras

Keras supports the addition of noise to models via the `GaussianNoise` layer. This is a layer that will add noise to inputs of a given shape. The noise has a mean of zero and requires that a standard deviation of the noise be specified as a hyperparameter. For example:

```
# import noise layer
from keras.layers import GaussianNoise
# define noise layer
layer = GaussianNoise(0.1)
```

Listing 17.1: Example of creating a `GaussianNoise` layer.

The output of the layer will have the same shape as the input, with the only modification being the addition of noise to the values.

17.2.2 Noise Regularization in Models

The `GaussianNoise` layer can be used in a few different ways with a neural network model. Firstly, it can be used as an input layer to add noise to input variables directly. This is the traditional use of noise as a regularization method in neural networks. Below is an example of defining a `GaussianNoise` layer as an input layer for a model that takes 2 input variables.

```
...
model.add(GaussianNoise(0.01, input_shape=(2,)))
...
```

Listing 17.2: Example of adding a `GaussianNoise` layer to a model.

Noise can also be added between hidden layers in the model. Given the flexibility of Keras, the noise can be added before or after the use of the activation function. It may make more sense to add it before the activation; nevertheless, both options are possible. Below is an example of a `GaussianNoise` layer that adds noise to the linear output of a `Dense` layer before a rectified linear activation function, perhaps a more appropriate use of noise between hidden layers.

```
...
model.add(Dense(32))
model.add(GaussianNoise(0.1))
model.add(Activation('relu'))
model.add(Dense(32))
...
```

Listing 17.3: Example of adding a `GaussianNoise` layer before activation.

Noise can also be added after the activation function, much like using a noisy activation function. One downside of this usage is that the resulting values may be out-of-range from what the activation function may normally provide. For example, a value with added noise may be less than zero, whereas the `relu` activation function will only ever output values 0 or larger.

```
...
model.add(Dense(32, activation='relu'))
model.add(GaussianNoise(0.1))
model.add(Dense(32))
...
```

Listing 17.4: Example of adding a `GaussianNoise` layer after activation.

Let's take a look at how noise regularization can be used with some common network types.

MLP Noise Regularization

The example below adds noise between two `Dense` fully connected layers.

```
# example of noise between fully connected layers
from keras.layers import Dense
from keras.layers import GaussianNoise
from keras.layers import Activation
...
model.add(Dense(32))
model.add(GaussianNoise(0.1))
model.add(Activation('relu'))
model.add(Dense(1))
```

...

Listing 17.5: Example of adding a Noise to an MLP model.

CNN Noise Regularization

The example below adds noise after a pooling layer in a convolutional network.

```
# example of noise for a CNN
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import GaussianNoise
...
model.add(Conv2D(32, (3,3)))
model.add(Conv2D(32, (3,3)))
model.add(MaxPooling2D())
model.add(GaussianNoise(0.1))
model.add(Dense(1))
...
```

Listing 17.6: Example of adding a Noise to a CNN model.

RNN Noise Regularization

The example below adds noise between an LSTM recurrent layer and a Dense fully connected layer.

```
# example of noise between LSTM and fully connected layers
from keras.layers import Dense
from keras.layers import Activation
from keras.layers import LSTM
from keras.layers import GaussianNoise
...
model.add(LSTM(32))
model.add(GaussianNoise(0.5))
model.add(Activation('relu'))
model.add(Dense(1))
...
```

Listing 17.7: Example of adding a Noise to an LSTM model.

Now that we have seen how to add noise to neural network models, let's look at a case study of adding noise to an overfit model to reduce generalization error.

17.3 Noise Regularization Case Study

In this section, we will demonstrate how to use noise regularization to reduce overfitting of an MLP on a simple binary classification problem. This example provides a template for applying noise regularization to your own neural network for classification and regression problems.

17.3.1 Binary Classification Problem

We will use a standard binary classification problem that defines two two-dimensional concentric circles of observations, one circle for each class. Each observation has two input variables with the same scale and a class output value of either 0 or 1. This dataset is called the *circles* dataset because of the shape of the observations in each class when plotted. We can use the `make_circles()` function to generate observations from this problem. We will add noise to the data and seed the random number generator so that the same samples are generated each time the code is run.

```
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
```

Listing 17.8: Example of creating samples for the two circles problem.

We can plot the dataset where the two variables are taken as x and y coordinates on a graph and the class value is taken as the color of the observation. The complete example of generating the dataset and plotting it is listed below.

```
# scatter plot of circles dataset
from sklearn.datasets import make_circles
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# scatter plot for each class value
for class_value in range(2):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 17.9: Example of plotting samples from the two circles problem.

Running the example creates a scatter plot showing the concentric circles shape of the observations in each class. We can see the noise in the dispersal of the points making the circles less obvious.

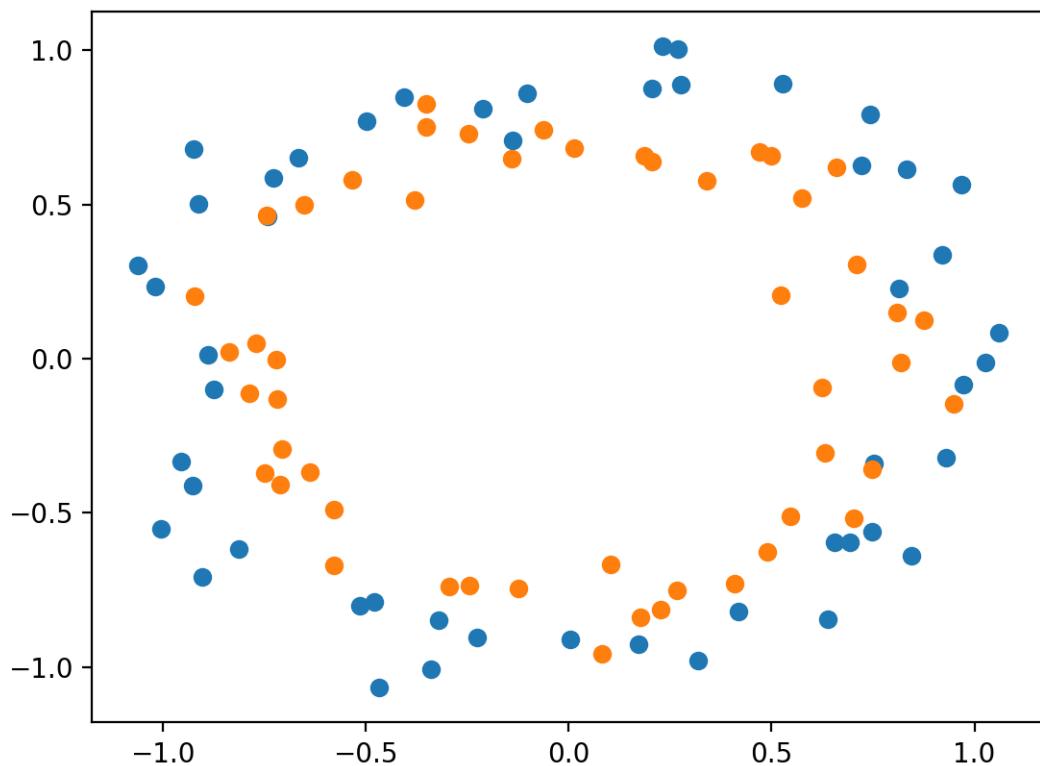


Figure 17.1: Scatter Plot of Circles Dataset with Color Showing the Class Value of Each Sample.

This is a good test problem because the classes cannot be separated by a line, e.g. are not linearly separable, requiring a nonlinear method such as a neural network to address. We have only generated 100 samples, which is small for a neural network, providing the opportunity to overfit the training dataset and have higher error on the test dataset, a good case for using regularization. Further, the samples have noise, giving the model an opportunity to learn aspects of the samples that don't generalize.

17.3.2 Overfit Multilayer Perceptron

We can develop an MLP model to address this binary classification problem. The model will have one hidden layer with more nodes than may be required to solve this problem, providing an opportunity to overfit. We will also train the model for longer than is required to ensure the model overfits. Before we define the model, we will split the dataset into train and test sets, using 30 examples to train the model and 70 to evaluate the fit model's performance.

```
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 17.10: Example of preparing the dataset for modeling.

Next, we can define the model. The hidden layer uses 500 nodes in the hidden layer and the rectified linear activation function. A sigmoid activation function is used in the output layer in order to predict class values of 0 or 1. The model is optimized using the binary cross-entropy loss function, suitable for binary classification problems and the efficient Adam version of gradient descent.

```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 17.11: Example of defining an MLP model.

The defined model is then fit on the training data for 4,000 epochs and the default batch size of 32. We will also use the test dataset as a validation dataset.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
```

Listing 17.12: Example of fitting an MLP model.

We can evaluate the performance of the model on the test dataset and report the result.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 17.13: Example of evaluating an MLP model.

Finally, we will plot the performance of the model on both the train and test set each epoch. If the model does indeed overfit the training dataset, we would expect the line plot of accuracy on the training set to continue to increase and the test set to rise and then fall again as the model learns statistical noise in the training dataset.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()

# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 17.14: Example of plotting learning curves for the fit MLP model.

We can tie all of these pieces together; the complete example is listed below.

```
# mlp overfit on the two circles dataset
from sklearn.datasets import make_circles
from keras.layers import Dense
from keras.models import Sequential
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[: n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 17.15: Example of MLP fit on the two circles problem.

Running the example reports the model performance on the train and test datasets. We can see that the model has better performance on the training dataset than the test dataset, one possible sign of overfitting.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

Train: 1.000, Test: 0.771

Listing 17.16: Example output fitting an MLP on the two circles problem.

A figure is created showing line plots of the model accuracy on the train and test sets. We can see that expected shape of an overfit model where test accuracy increases to a point and then begins to decrease again.

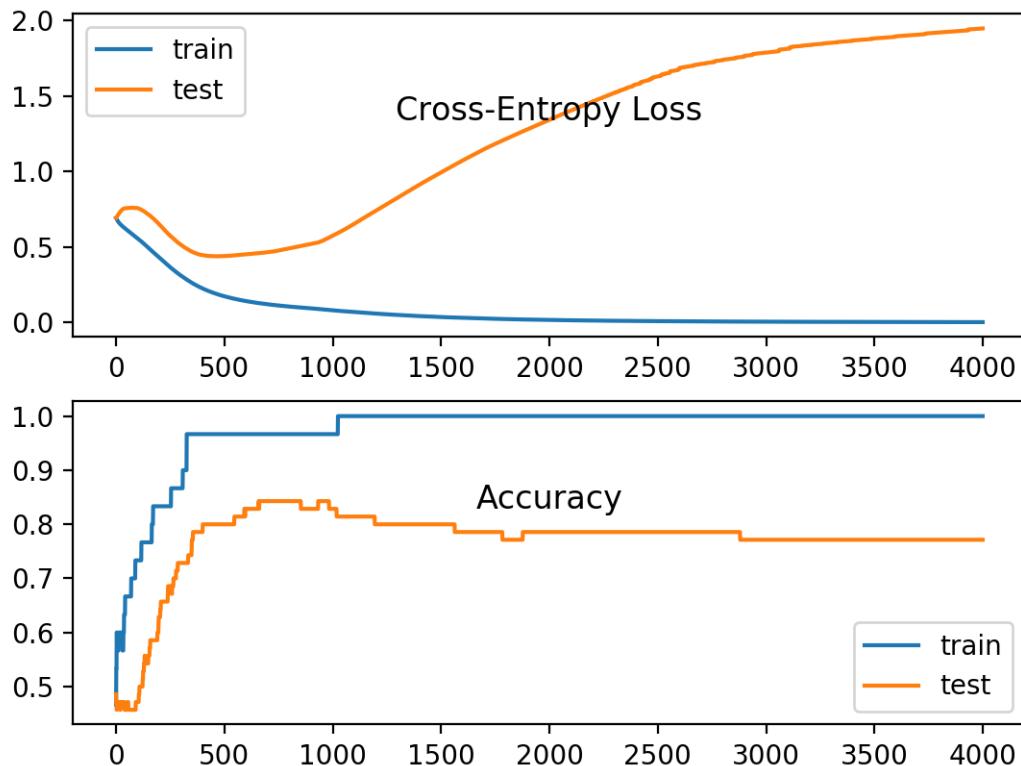


Figure 17.2: Line Plots of Accuracy on Train and Test Datasets While Training Showing an Overfit.

17.3.3 MLP With Input Layer Noise

The dataset is defined by points that have a controlled amount of statistical noise. Nevertheless, because the dataset is small, we may wish to add further noise to the input values. This will have the effect of creating more samples or resampling the domain, making the structure of the input space artificially smoother. This may make the problem easier to learn and improve generalization performance. We can add a `GaussianNoise` layer as the input layer. The amount of noise must be small. Given that the input values are within the range [0, 1], we will add Gaussian noise with a mean of 0.0 and a standard deviation of 0.01, chosen arbitrarily.

```
# define model
model = Sequential()
model.add(GaussianNoise(0.01, input_shape=(2,)))
model.add(Dense(500, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 17.17: Example of updating the MLP model to add noise to the input.

The complete example with this change is listed below.

```
# mlp overfit on the two circles dataset with input noise
```

```

from sklearn.datasets import make_circles
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import GaussianNoise
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(GaussianNoise(0.01, input_shape=(2,)))
model.add(Dense(500, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 17.18: Example of MLP with noise added to input on the two circles problem.

Running the example reports the model performance on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we may see a small lift in performance of the model on the test dataset, with no negative impact on the training dataset.

Train: 1.000, Test: 0.786

Listing 17.19: Example output the MLP with noise added to input on the two circles problem.

We clearly see the impact of the added noise on the evaluation of the model during training as graphed on the line plot. The noise causes the accuracy of the model to jump around during training, possibly due to the noise introducing points that conflict with true points from the training dataset. Perhaps a lower input noise standard deviation would be more appropriate.

The model still shows a pattern of being overfit, with a rise and then fall in test accuracy over training epochs.

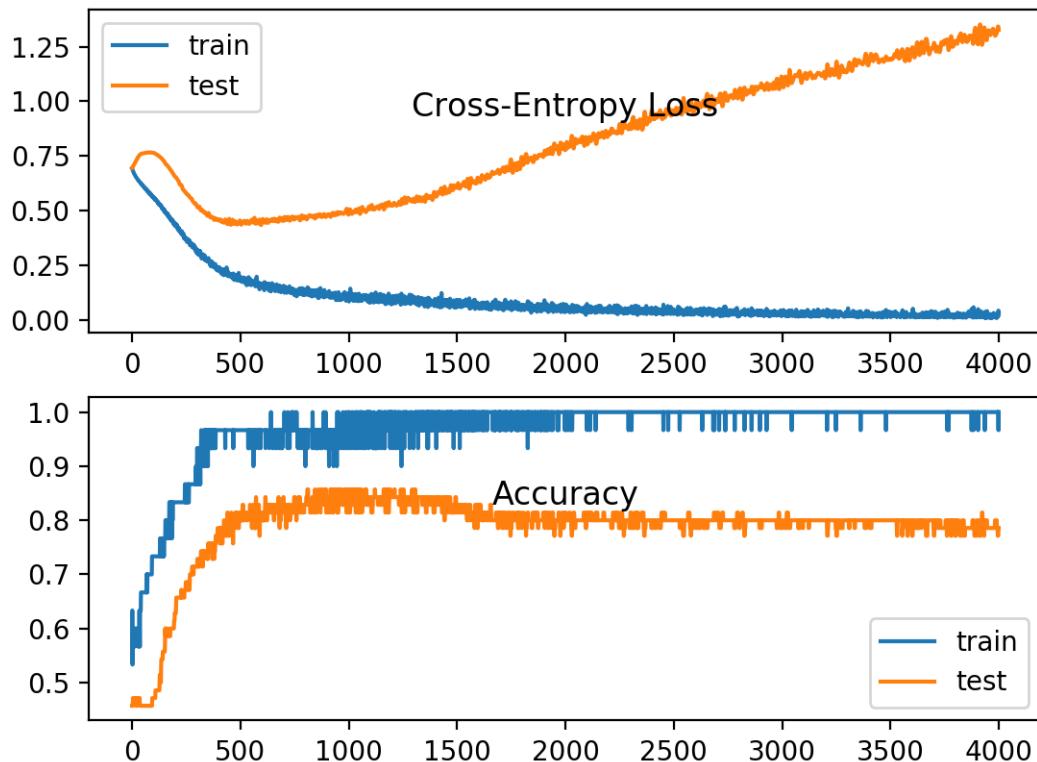


Figure 17.3: Line Plot of Train and Test Accuracy With Input Layer Noise.

17.3.4 MLP With Hidden Layer Noise

An alternative approach to adding noise to the input values is to add noise between the hidden layers. This can be done by adding noise to the linear output of the layer (weighted sum) before the activation function is applied, in this case a rectified linear activation function. We can also use a larger standard deviation for the noise as the model is less sensitive to noise at this level given the presumably larger weights from being overfit. We will use a standard deviation of 0.1, again, chosen arbitrarily.

```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2))
model.add(GaussianNoise(0.1))
model.add(Activation('relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 17.20: Example of updating the MLP model to add noise before activation.

The complete example with Gaussian noise between the hidden layers is listed below.

```
# mlp overfit on the two circles dataset with hidden layer noise
from sklearn.datasets import make_circles
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Activation
from keras.layers import GaussianNoise
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2))
model.add(GaussianNoise(0.1))
model.add(Activation('relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 17.21: Example of MLP with noise added to the hidden layer before activation for the two circles problem.

Running the example reports the model performance on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see a marked increase in the performance of the model on the hold out test set.

Train: 0.933, Test: 0.814

Listing 17.22: Example output the MLP with noise added to the hidden layer before activation for the two circles problem.

We can also see from the line plot of accuracy over training epochs that the model no longer appears to show the properties of being overfit with regard to classification accuracy. The learning curves for loss do still show a pattern of being overfit.

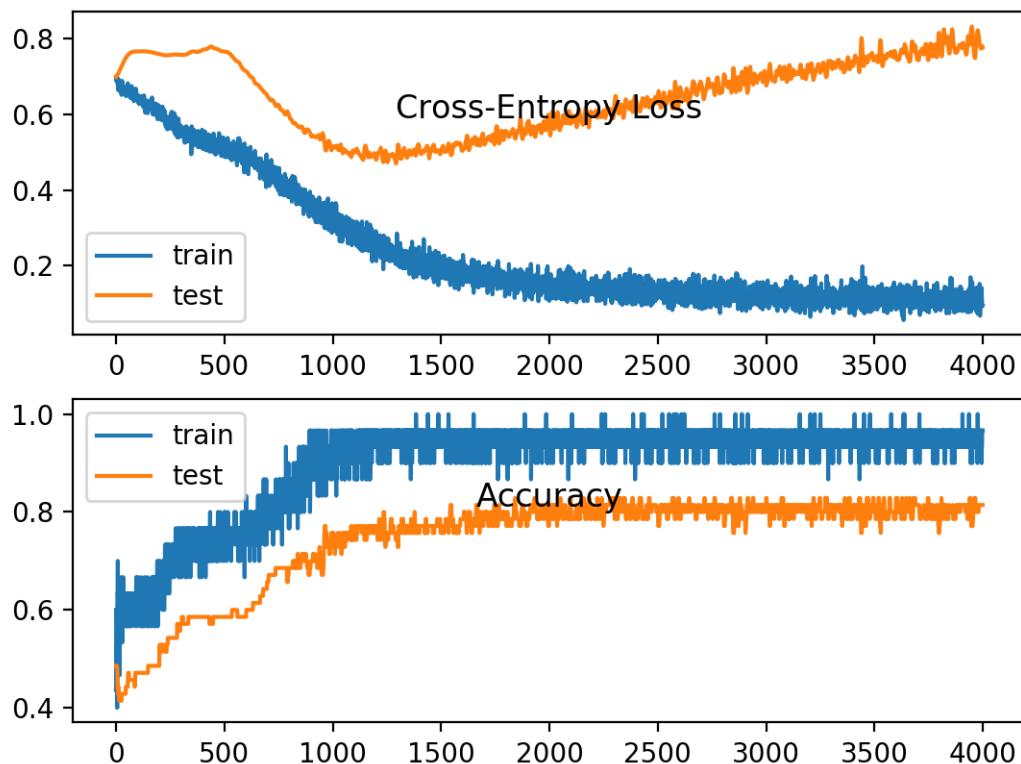


Figure 17.4: Line Plot of Train and Test Accuracy With Hidden Layer Noise.

We can also experiment and add the noise after the outputs of the first hidden layer pass through the activation function.

```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(GaussianNoise(0.1))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 17.23: Example of updating the MLP model to add noise after activation.

The complete example is listed below.

```
# mlp overfit on the two circles dataset with hidden layer noise (alternate)
from sklearn.datasets import make_circles
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import GaussianNoise
from matplotlib import pyplot
```

```

# generate 2d classification dataset
X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(GaussianNoise(0.1))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 17.24: Example of MLP with noise added to the hidden layer after activation for the two circles problem.

Running the example reports the model performance on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

Surprisingly, we see little difference in the performance of the model, perhaps a small lift in performance.

Train: 1.000, Test: 0.829

Listing 17.25: Example output the MLP with noise added to the hidden layer after activation for the two circles problem.

Again, we can see from the line plot of accuracy over training epochs that the model no longer shows sign of overfitting.

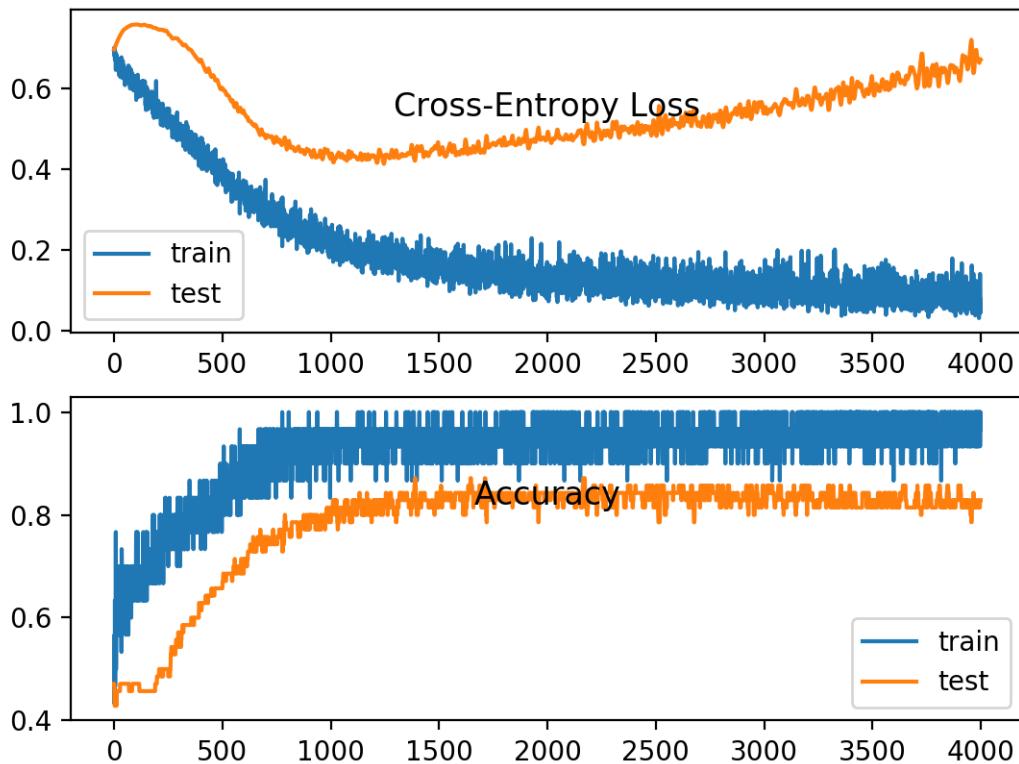


Figure 17.5: Line Plot of Train and Test Accuracy With Hidden Layer Noise (alternate).

17.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Repeated Evaluation.** Update the example to use repeated evaluation of the model with and without noise and report performance as the mean and standard deviation over repeats.
- **Grid Search Standard Deviation.** Develop a grid search in order to discover the amount of noise that reliably results in the best performing model.
- **Input and Hidden Noise.** Update the example to introduce noise at both the input and hidden layers of the model.

If you explore any of these extensions, I'd love to know.

17.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

17.5.1 Books

- Section 7.5: Noise Robustness, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- Chapter 17: Training with Noisy Inputs, *Neural Smithing: Supervised Learning in Feed-forward Artificial Neural Networks*, 1999.
<https://amzn.to/2Dxo4XU>
- Section 9.3: Training with Noise, *Neural Networks for Pattern Recognition*, 1995.
<https://amzn.to/2I9gNMP>

17.5.2 Papers

- *Creating artificial neural networks that generalize*, 1991.
<https://www.sciencedirect.com/science/article/pii/0893608091900332>
- *Deep networks for robust visual recognition*, 2010.
<https://dl.acm.org/citation.cfm?id=3104456>
- *Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion*, 2010.
<http://www.jmlr.org/papers/v11/vincent10a.html>
- *Analyzing noise in autoencoders and deep networks*, 2014.
<https://arxiv.org/abs/1406.1831>
- *The Effects of Adding Noise During Backpropagation Training on a Generalization Performance*, 1996.
<https://ieeexplore.ieee.org/document/6796981/>
- *Training with Noise is Equivalent to Tikhonov Regularization*, 2008.
<https://www.mitpressjournals.org/doi/abs/10.1162/neco.1995.7.1.108>
- *Adding Gradient Noise Improves Learning for Very Deep Networks*, 2016.
<https://arxiv.org/abs/1511.06807>
- *Noisy Activation Functions*, 2016.
<http://proceedings.mlr.press/v48/gulcehre16.html>

17.5.3 APIs

- Keras Regularizers API.
<https://keras.io/regularizers/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>

- Keras Recurrent Layers API.
<https://keras.io/layers/recurrent/>
- Keras Noise API.
<https://keras.io/layers/noise/>
- `sklearn.datasets.make_circles` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html

17.5.4 Articles

- What is jitter? (Training with noise), Neural Network FAQ.
ftp://ftp.sas.com/pub/neural/FAQ3.html#A_jitter
- Jitter, Wikipedia.
<https://en.wikipedia.org/wiki/Jitter>

17.6 Summary

In this tutorial, you discovered that adding noise to a neural network during training can improve the robustness of the network resulting in better generalization and faster learning. Specifically, you learned:

- Small datasets can make learning challenging for neural nets and the examples can be memorized.
- Adding noise during training can make the training process more robust and reduce generalization error.
- Noise is traditionally added to the inputs, but can also be added to weights, gradients, and even activation functions.

17.6.1 Next

In the next tutorial, you will discover how to halt model training at the right time with early stopping.

Chapter 18

Halt Training at the Right Time with Early Stopping

A major challenge in training neural networks is how long to train them. Too little training will mean that the model will underfit the train and the test sets. Too much training will mean that the model will overfit the training dataset and have poor performance on the test set. A compromise is to train on the training dataset but to stop training at the point when performance on a validation dataset starts to degrade. This simple, effective, and widely used approach to training neural networks is called early stopping. In this tutorial, you will discover that stopping the training of a neural network early before it has overfit the training dataset can reduce overfitting and improve the generalization of deep neural networks. After reading this tutorial, you will know:

- The challenge of training a neural network long enough to learn the mapping, but not so long that it overfits the training data.
- Model performance on a holdout validation dataset can be monitored during training and training stopped when generalization error starts to increase.
- The use of early stopping requires the selection of a performance measure to monitor, a trigger to stop training, and a selection of the model weights to use.

Let's get started.

18.1 Early Stopping

In this section discover the problem of training a model for too long and the regularizing effect that halting the training process at the right time can have, as well as tips for using early stopping in your own projects.

18.1.1 The Problem of Training Just Enough

Training neural networks is challenging. When training a large network, there will be a point during training when the model will stop generalizing and start learning the statistical noise in the training dataset. This overfitting of the training dataset will result in an increase in

generalization error, making the model less useful at making predictions on new data. The challenge is to train the network long enough that it is capable of learning the mapping from inputs to outputs, but not training the model so long that it overfits the training data.

However, all standard neural network architectures such as the fully connected multi-layer perceptron are prone to overfitting [10]: While the network seems to get better and better, i.e., the error on the training set decreases, at some point during training it actually begins to get worse again, i.e., the error on unseen examples increases.

— *Early Stopping — But When?*, 2002.

One approach to solving this problem is to treat the number of training epochs as a hyperparameter and train the model multiple times with different values, then select the number of epochs that result in the best performance on the train or a holdout test dataset. The downside of this approach is that it requires multiple models to be trained and discarded. This can be computationally inefficient and time-consuming, especially for large models trained on large datasets over days or weeks.

18.1.2 Stop Training When Generalization Error Increases

An alternative approach is to train the model once for a large number of training epochs. During training, the model is evaluated on a holdout validation dataset after each epoch. If the performance of the model on the validation dataset starts to degrade (e.g. loss begins to increase or accuracy begins to decrease), then the training process is stopped.

... the error measured with respect to independent data, generally called a validation set, often shows a decrease at first, followed by an increase as the network starts to over-fit. Training can therefore be stopped at the point of smallest error with respect to the validation data set

— Page 259, *Pattern Recognition and Machine Learning*, 2006.

The model at the time that training is stopped is then used and is known to have good generalization performance. This procedure is called *early stopping* and is perhaps one of the oldest and most widely used forms of neural network regularization.

This strategy is known as early stopping. It is probably the most commonly used form of regularization in deep learning. Its popularity is due both to its effectiveness and its simplicity.

— Page 247, *Deep Learning*, 2016.

If regularization methods like weight decay that update the loss function to encourage less complex models are considered *explicit* regularization, then early stopping may be thought of as a type of *implicit* regularization, much like using a smaller network that has less capacity.

Regularization may also be implicit as is the case with early stopping.

— *Understanding deep learning requires rethinking generalization*, 2017.

18.1.3 How to Stop Training Early

Early stopping requires that you configure your network to be under constrained, meaning that it has more capacity than is required for the problem. When training the network, a larger number of training epochs is used than may normally be required, to give the network plenty of opportunity to fit, then begin to overfit the training dataset. There are three elements to using early stopping; they are:

- Monitoring model performance.
- Trigger to stop training.
- The choice of model to use.

Monitoring Performance

The performance of the model must be monitored during training. This requires the choice of a dataset that is used to evaluate the model and a metric used to evaluate the model. It is common to split the training dataset and use a subset, such as 30%, as a validation dataset used to monitor performance of the model during training. This validation set is not used to train the model. It is also common to use the loss on a validation dataset as the metric to monitor, although you may also use prediction error in the case of regression, or accuracy in the case of classification.

The loss of the model on the training dataset will also be available as part of the training procedure, and additional metrics may also be calculated and monitored on the training dataset. Performance of the model is evaluated on the validation set at the end of each epoch, which adds an additional computational cost during training. This can be reduced by evaluating the model less frequently, such as every 2, 5, or 10 training epochs.

Early Stopping Trigger

Once a scheme for evaluating the model is selected, a trigger for stopping the training process must be chosen. The trigger will use a monitored performance metric to decide when to stop training. This is often the performance of the model on the holdout dataset, such as the loss. In the simplest case, training is stopped as soon as the performance on the validation dataset decreases as compared to the performance on the validation dataset at the prior training epoch (e.g. an increase in loss). More elaborate triggers may be required in practice. This is because the training of a neural network is stochastic and can be noisy. Plotted on a graph, the performance of a model on a validation dataset may go up and down many times. This means that the first sign of overfitting may not be a good place to stop training.

... the validation error can still go further down after it has begun to increase [...]
Real validation error curves almost always have more than one local minimum.

— *Early Stopping — But When?*, 2002.

Some more elaborate triggers may include:

- No change in metric over a given number of epochs.

- An absolute change in a metric.
- A decrease in performance observed over a given number of epochs.
- Average change in metric over a given number of epochs.

Some delay or *patience* in stopping is almost always a good idea.

... results indicate that “slower” criteria, which stop later than others, on the average lead to improved generalization compared to “faster” ones. However, the training time that has to be expended for such improvements is rather large on average and also varies dramatically when slow criteria are used.

— *Early Stopping — But When?*, 2002.

Model Choice

At the time that training is halted, the model is known to have slightly worse generalization error than a model at a prior epoch. As such, some consideration may need to be given as to exactly which model is saved. Specifically, the training epoch from which weights in the model that are saved to file. This will depend on the trigger chosen to stop the training process. For example, if the trigger is a simple decrease in performance from one epoch to the next, then the weights for the model at the prior epoch will be preferred. If the trigger is required to observe a decrease in performance over a fixed number of epochs, then the model at the beginning of the trigger period will be preferred. Perhaps a simple approach is to always save the model weights if the performance of the model on a holdout dataset is better than at the previous epoch. That way, you will always have the model with the best performance on the holdout set.

Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters.

— Page 246, *Deep Learning*, 2016.

18.1.4 Examples of Early Stopping

This section summarizes some examples where early stopping has been used. Yoon Kim in his seminal application of convolutional neural networks to sentiment analysis in the 2014 paper titled *Convolutional Neural Networks for Sentence Classification* used early stopping with 10% of the training dataset used as the validation hold outset.

We do not otherwise perform any dataset-specific tuning other than early stopping on dev sets. For datasets without a standard dev set we randomly select 10% of the training data as the dev set.

Chiyuan Zhang, et al. from MIT, Berkeley, and Google in their 2017 paper titled *Understanding deep learning requires rethinking generalization* highlight that on very deep convolutional neural networks for photo classification where there is an abundant dataset that early stopping may not always offer benefit, as the model is less likely to overfit such large datasets.

[regarding] the training and testing accuracy on ImageNet [results suggest] a reference of potential performance gain for early stopping. However, on the CIFAR10 dataset, we do not observe any potential benefit of early stopping.

Yarin Gal and Zoubin Ghahramani from Cambridge in their 2015 paper titled *A Theoretically Grounded Application of Dropout in Recurrent Neural Networks* use early stopping as an *unregularized baseline* for LSTM models on a suite of language modeling problems.

Lack of regularisation in RNN models makes it difficult to handle small data, and to avoid overfitting researchers often use early stopping, or small and under-specified models ...

Alex Graves, et al., in their famous 2013 paper titled *Speech recognition with deep recurrent neural networks* achieved state-of-the-art results with LSTMs for speech recognition, while making use of early stopping.

Regularisation is vital for good performance with RNNs, as their flexibility makes them prone to overfitting. Two regularisers were used in this paper: early stopping and weight noise ...

18.1.5 Tips for Early Stopping

This section provides some tips for using early stopping regularization with your neural network.

When to Use Early Stopping

Early stopping is so easy to use, e.g. with the simplest trigger, that there is little reason to not use it when training neural networks. Use of early stopping may be a staple of the modern training of deep neural networks.

Early stopping should be used almost universally.

— Page 425, *Deep Learning*, 2016.

Plot Learning Curves to Select a Trigger

Before using early stopping, it may be interesting to fit an under constrained model and monitor the performance of the model on a train and validation dataset. Plotting the performance of the model in real-time or at the end of a long run will show how noisy the training process is with your specific model and dataset. This may help in the choice of a trigger for early stopping.

Monitor an Important Metric

Loss is an easy metric to monitor during training and to trigger early stopping. The problem is that loss does not always capture what is most important about the model to you and your project.

Sometimes, the loss function we actually care about (say classification error) is not one that can be optimized efficiently. [...] In such situations, one typically optimizes a surrogate loss function instead, which acts as a proxy but has advantages.

— Page 276, *Deep Learning*, 2016.

It may be better to choose a performance metric to monitor that best defines the performance of the model in terms of the way you intend to use it. This may be the metric that you intend to use to report the performance of the model.

Suggested Training Epochs

A problem with early stopping is that the model does not make use of all available training data. It may be desirable to avoid overfitting and to train on all possible data, especially on problems where the amount of training data is very limited. A recommended approach would be to treat the number of training epochs as a hyperparameter and to grid search a range of different values, perhaps using k -fold cross-validation. This will allow you to fix the number of training epochs and fit a final model on all available data.

Early stopping could be used instead. The early stopping procedure could be repeated a number of times. The epoch number at which training was stopped could be recorded. Then, the average of the epoch number across all repeats of early stopping could be used when fitting a final model on all available training data. This process could be performed using a different split of the training set into train and validation steps each time early stopping is run. An alternative might be to use early stopping with a validation dataset, then update the final model with further training on the held out validation set.

Early Stopping With Cross-Validation

Early stopping could be used with k -fold cross-validation, although it is not recommended. The k -fold cross-validation procedure is designed to estimate the generalization error of a model by repeatedly refitting and evaluating it on different subsets of a dataset. Early stopping is designed to monitor the generalization error of one model and stop training when generalization error begins to degrade. They are at odds because cross-validation assumes you don't know the generalization error and early stopping is trying to give you the best model based on knowledge of generalization error.

It may be desirable to use cross-validation to estimate the performance of models with different hyperparameter values, such as learning rate or network structure, whilst also using early stopping. In this case, if you have the resources to repeatedly evaluate the performance of the model, then perhaps the number of training epochs may also be treated as a hyperparameter to be optimized, instead of using early stopping. Instead of using cross-validation with early stopping, early stopping may be used directly without repeated evaluation when evaluating different hyperparameter values for the model (e.g. different learning rates). One possible point of confusion is that early stopping is sometimes referred to as *cross-validated training*. Further, research into early stopping that compares triggers may use cross-validation to compare the impact of different triggers.

Overfit Validation

Repeating the early stopping procedure many times may result in the model overfitting the validation dataset. This can happen just as easily as overfitting the training dataset. One approach is to only use early stopping once all other hyperparameters of the model have been

chosen. Another strategy may be to use a different split of the training dataset into train and validation sets each time early stopping is used.

18.2 Early Stopping Keras API

This section describes how to use early stopping with the Keras API.

18.2.1 Using Callbacks in Keras

A callback is a snippet of code that can be executed at a specific point during training, such as before or after training, an epoch or a batch. They provide a way to execute code and interact with the training model process automatically. Callbacks can be provided to the `fit()` function via the `callbacks` argument. First, callback must be instantiated.

```
...  
cb = Callback(...)
```

Listing 18.1: Example of creating a callback.

Then, one or more callbacks that you intend to use must be added to a Python list.

```
...  
cb_list = [cb, ...]
```

Listing 18.2: Example of creating a list of callbacks.

Finally, the list of callbacks is provided to the `callback` argument when fitting the model.

```
...  
model.fit(..., callbacks=cb_list)
```

Listing 18.3: Example of creating using a list of callbacks when fitting a model.

18.2.2 Evaluating a Validation Dataset in Keras

Early stopping requires that a validation dataset is evaluated during training. This can be achieved by specifying the validation dataset to the `fit()` function when training your model. There are two ways of doing this. The first involves you manually splitting your training data into a train and validation dataset and specifying the validation dataset to the `fit()` function via the `validation_data` argument. For example:

```
...  
model.fit(train_X, train_y, validation_data=(val_x, val_y))
```

Listing 18.4: Example of specifying a validation dataset.

Alternately, the `fit()` function can automatically split your training dataset into train and validation sets based on a percentage split specified via the `validation_split` argument. The `validation_split` is a value between 0 and 1 and defines the percentage amount of the training dataset to use for the validation dataset. For example:

```
...
model.fit(train_X, train_y, validation_split=0.3)
```

Listing 18.5: Example of specifying a validation dataset as a percentage of the training set.

In both cases, the model is not trained on the validation dataset. Instead, the model is evaluated on the validation dataset at the end of each training epoch.

18.2.3 Monitoring Model Performance

The loss function chosen to be optimized for your model is calculated at the end of each epoch. To callbacks, this is made available via the name `loss`. If a validation dataset is specified to the `fit()` function via the `validation_data` or `validation_split` arguments, then the loss on the validation dataset will be made available via the name `val_loss`. Additional metrics can be monitored during the training of the model. They can be specified when compiling the model via the `metrics` argument to the `compile` function. This argument takes a Python list of known metric functions, such as `mse` for mean squared error and `acc` for accuracy. For example:

```
...
model.compile(..., metrics=['accuracy'])
```

Listing 18.6: Example of monitoring accuracy during training.

If additional metrics are monitored during training, they are also available to the callbacks via the same name, such as `acc` for accuracy on the training dataset and `val_acc` for the accuracy on the validation dataset. Or, `mse` for mean squared error on the training dataset and `val_mse` on the validation dataset.

18.2.4 Early Stopping Callback

Keras supports the early stopping of training via a callback called `EarlyStopping`. This callback allows you to specify the performance measure to monitor, the trigger, and once triggered, it will stop the training process. The `EarlyStopping` callback is configured when instantiated via arguments. The `monitor` allows you to specify the performance measure to monitor in order to end training. Recall from the previous section that the calculation of measures on the validation dataset will have the `val_` prefix, such as `val_loss` for the loss on the validation dataset.

```
es = EarlyStopping(monitor='val_loss')
```

Listing 18.7: Example of early stopping monitoring validation loss.

Based on the choice of performance measure, the `mode` argument will need to be specified as whether the objective of the chosen metric is to increase (`maximize` or `max`) or to decrease (`minimize` or `min`). For example, we would seek a minimum for validation loss and a minimum for validation mean squared error, whereas we would seek a maximum for validation accuracy.

```
es = EarlyStopping(monitor='val_loss', mode='min')
```

Listing 18.8: Example of early stopping monitoring a minimized validation loss.

By default, `mode` is set to `auto` and knows that you want to minimize loss or maximize accuracy. That is all that is needed for the simplest form of early stopping. Training will stop

when the chosen performance measure stops improving. To discover the training epoch on which training was stopped, the `verbose` argument can be set to 1. Once stopped, the callback will print the epoch number.

```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1)
```

Listing 18.9: Example of early stopping with verbose output.

Often, the first sign of no further improvement may not be the best time to stop training. This is because the model may coast into a plateau of no improvement or even get slightly worse before getting much better. We can account for this by adding a delay to the trigger in terms of the number of epochs on which we would like to see no improvement. This can be done by setting the `patience` argument.

```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=50)
```

Listing 18.10: Example of early stopping with patience.

The exact amount of patience will vary between models and problems. Reviewing plots of your performance measure can be very useful to get an idea of how noisy the optimization process for your model on your data may be. By default, any change in the performance measure, no matter how fractional, will be considered an improvement. You may want to consider an improvement that is a specific increment, such as 1 unit for mean squared error or 1% for accuracy. This can be specified via the `min_delta` argument.

```
es = EarlyStopping(monitor='val_accuracy', mode='max', min_delta=1)
```

Listing 18.11: Example of early stopping with a minimum delta.

Finally, it may be desirable to only stop training if performance stays above or below a given threshold or baseline. For example, if you have familiarity with the training of the model (e.g. learning curves) and know that once a validation loss of a given value is achieved that there is no point in continuing training. This can be specified by setting the `baseline` argument. This might be more useful when fine tuning a model, after the initial wild fluctuations in the performance measure seen in the early stages of training a new model are past.

```
es = EarlyStopping(monitor='val_loss', mode='min', baseline=0.4)
```

Listing 18.12: Example of early stopping with a baseline.

18.2.5 Model Checkpointing

The `EarlyStopping` callback will stop training once triggered, but the model at the end of training may not be the model with best performance on the validation dataset. An additional callback is required that will save the best model observed during training for later use. This is the `ModelCheckpoint` callback. The `ModelCheckpoint` callback is flexible in the way it can be used, but in this case we will use it only to save the best model observed during training as defined by a chosen performance measure on the validation dataset. Saving and loading models requires that HDF5 support has been installed on your workstation. For example, using the `pip` Python installer, this can be achieved as follows:

```
sudo pip install h5py
```

Listing 18.13: Install the `h5py` library via `pip`.

You can learn more from the `h5py` Installation documentation¹. The callback will save the model to file, which requires that a path and filename be specified via the first argument.

```
mc = ModelCheckpoint('best_model.h5')
```

Listing 18.14: Example of creating a model checkpoint callback.

The preferred loss function to be monitored can be specified via the `monitor` argument, in the same way as the `EarlyStopping` callback. For example, loss on the validation dataset (the default).

```
mc = ModelCheckpoint('best_model.h5', monitor='val_loss')
```

Listing 18.15: Example of model checkpoint that monitors validation loss.

Also, as with the `EarlyStopping` callback, we must specify the `mode` as either minimizing or maximizing the performance measure. Again, the default is `auto`, which is aware of the standard performance measures.

```
mc = ModelCheckpoint('best_model.h5', monitor='val_loss', mode='min')
```

Listing 18.16: Example of model checkpoint that monitors a minimized validation loss.

Finally, we are interested in only the very best model observed during training, rather than the best compared to the previous epoch, which might not be the best overall if training is noisy. This can be achieved by setting the `save_best_only` argument to `True`.

```
mc = ModelCheckpoint('best_model.h5', monitor='val_loss', mode='min', save_best_only=True)
```

Listing 18.17: Example of model checkpoint that only saves the best.

That is all that is needed to ensure the model with the best performance is saved when using early stopping, or in general. It may be interesting to know the value of the performance measure and at what epoch the model was saved. This can be printed by the callback by setting the `verbose` argument to 1.

```
mc = ModelCheckpoint('best_model.h5', monitor='val_loss', mode='min', verbose=1)
```

Listing 18.18: Example of model checkpoint with verbose output.

The saved model can then be loaded and evaluated any time by calling the `load_model()` function.

```
# load a saved model
from keras.models import load_model
saved_model = load_model('best_model.h5')
```

Listing 18.19: Example of loading a checkpoint model.

Now that we know how to use the early stopping and model checkpoint APIs, let's look at a worked example.

18.3 Early Stopping Case Study

In this section, we will demonstrate how to use early stopping to reduce overfitting of an MLP on a simple binary classification problem. This example provides a template for applying early stopping to your own neural network for classification and regression problems.

¹<http://docs.h5py.org/en/latest/build.html>

18.3.1 Binary Classification Problem

We will use a standard binary classification problem that defines two semi-circles of observations, one semi-circle for each class. Each observation has two input variables with the same scale and a class output value of either 0 or 1. This dataset is called the *moons* dataset because of the shape of the observations in each class when plotted. We can use the `make_moons()` function to generate observations from this problem. We will add noise to the data and seed the random number generator so that the same samples are generated each time the code is run.

```
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
```

Listing 18.20: Example of creating samples for the two moons problem.

We can plot the dataset where the two variables are taken as x and y coordinates on a graph and the class value is taken as the color of the observation. The complete example of generating the dataset and plotting it is listed below.

```
# scatter plot of moons dataset
from sklearn.datasets import make_moons
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# scatter plot for each class value
for class_value in range(2):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 18.21: Example of plotting samples from the two moons problem.

Running the example creates a scatter plot showing the semi-circle or moon shape of the observations in each class. We can see the noise in the dispersal of the points making the moons less obvious.

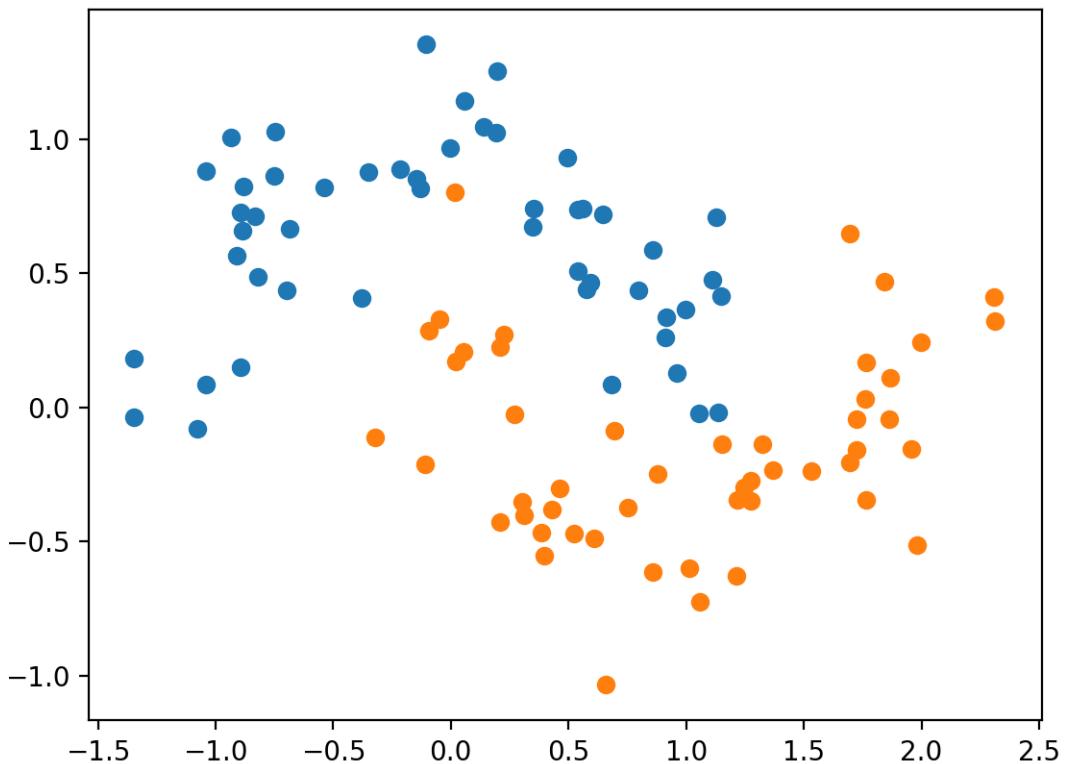


Figure 18.1: Scatter Plot of Moons Dataset With Color Showing the Class Value of Each Sample.

This is a good test problem because the classes cannot be separated by a straight line, e.g. are not linearly separable, requiring a nonlinear method such as a neural network to address. We have only generated 100 samples, which is small for a neural network, providing the opportunity to overfit the training dataset and have higher error on the test dataset: a good case for using regularization. Further, the samples have noise, giving the model an opportunity to learn aspects of the samples that don't generalize.

18.3.2 Overfit Multilayer Perceptron

We can develop an MLP model to address this binary classification problem. The model will have one hidden layer with more nodes than may be required to solve this problem, providing an opportunity to overfit. We will also train the model for longer than is required to ensure the model overfits. Before we define the model, we will split the dataset into train and test sets, using 30 examples to train the model and 70 to evaluate the fit model's performance.

```
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 18.22: Example of creating datasets ready for modeling.

Next, we can define the model. The hidden layer uses 500 nodes and the rectified linear activation function. A sigmoid activation function is used in the output layer in order to predict class values of 0 or 1. The model is optimized using the binary cross-entropy loss function, suitable for binary classification problems and the efficient Adam version of gradient descent.

```
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 18.23: Example of defining the MLP model.

The defined model is then fit on the training data for 4,000 epochs and the default batch size of 32. We will also use the test dataset as a validation dataset. This is just a simplification for this example. In practice, you would split the training set into train and validation and also hold back a test set for final model evaluation.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
```

Listing 18.24: Example of fitting the MLP model.

We can evaluate the performance of the model on the test dataset and report the result.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 18.25: Example of evaluating the MLP model.

Finally, we will plot the loss and accuracy of the model on both the train and test set each epoch. If the model does indeed overfit the training dataset, we would expect the line plot of loss (and accuracy) on the training set to continue to increase and the test set to rise and then fall again as the model learns statistical noise in the training dataset.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()

# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 18.26: Example of plotting learning curves for the MLP model.

We can tie all of these pieces together; the complete example is listed below.

```
# mlp overfit on the moons dataset
from sklearn.datasets import make_moons
from keras.layers import Dense
from keras.models import Sequential
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 18.27: Example of MLP fit on the two moons problem.

Running the example reports the model performance on the train and test datasets. We can see that the model has better performance on the training dataset than the test dataset, one possible sign of overfitting.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

Train: 1.000, Test: 0.914

Listing 18.28: Example output fitting an MLP on the two moons problem.

A figure is created showing line plots of the model loss and accuracy on the train and test sets. We can see that expected shape of an overfit model where test accuracy increases to a point and then begins to decrease again. Reviewing the figure, we can also see flat spots in the ups and downs in the validation loss. Any early stopping will have to account for these behaviors. We would also expect that a good time to stop training might be around epoch 800.

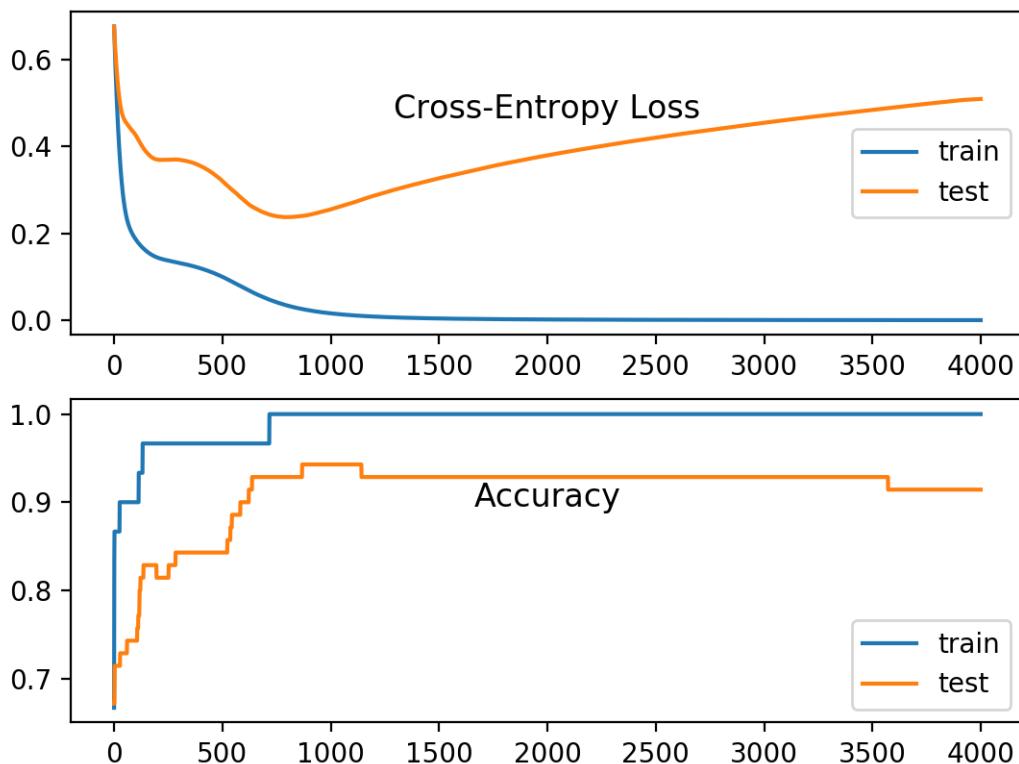


Figure 18.2: Line Plots of Loss on Train and Test Datasets While Training Showing an Overfit Model.

18.3.3 Overfit MLP With Early Stopping

We can update the example and add very simple early stopping. As soon as the loss of the model begins to increase on the test dataset, we will stop training. First, we can define the `EarlyStopping` callback.

```
# simple early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1)
```

Listing 18.29: Example of defining an early stopping callback for the MLP.

We can then update the call to the `fit()` function and specify a list of callbacks via the `callback` argument.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0,
 callbacks=[es])
```

Listing 18.30: Example of fitting the MLP with the early stopping callback.

The complete example with the addition of simple early stopping is listed below.

```
# mlp overfit on the moons dataset with simple early stopping
```

```

from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import EarlyStopping
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# simple early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0,
                     callbacks=[es])
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 18.31: Example of MLP with early stopping fit on the two moons problem.

Running the example reports the model performance on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can also see that the callback stopped training at epoch 219. This is too early as we would expect an early stop to be around epoch 800. This is also highlighted by the classification accuracy on both the train and test sets, which is worse than no early stopping.

```

Epoch 00219: early stopping
Train: 0.967, Test: 0.814

```

Listing 18.32: Example output fitting an MLP with early stopping on the two moons problem.

Reviewing the line plot of train and test loss, we can indeed see that training was stopped at the point when validation loss began to plateau for the first time.

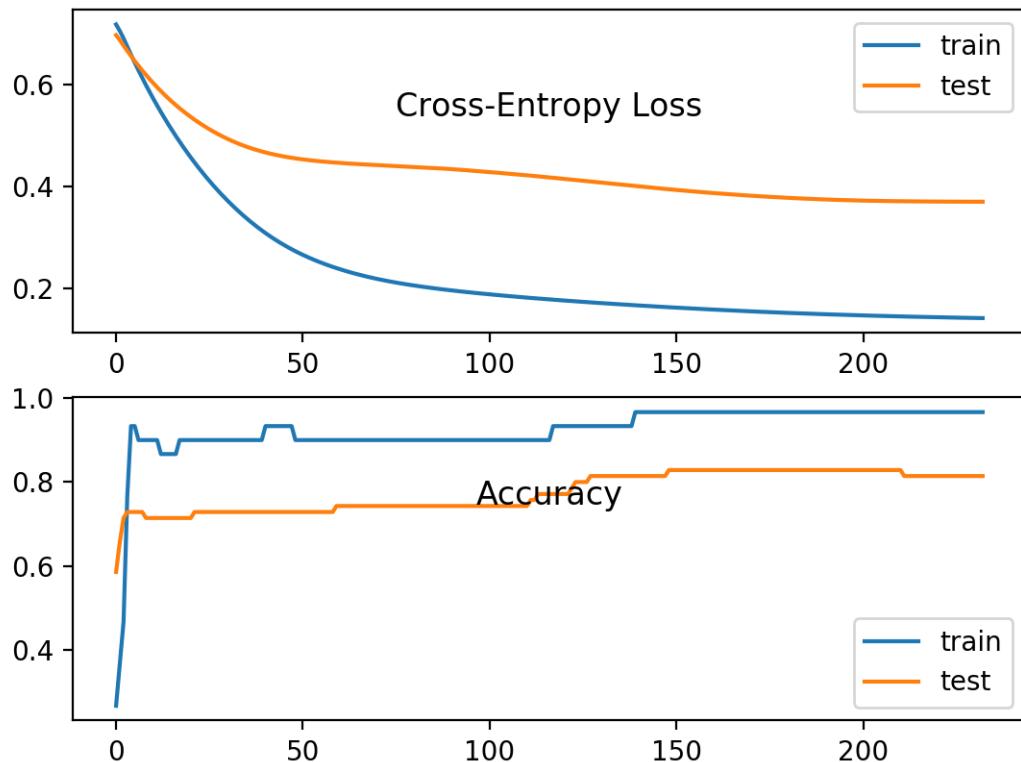


Figure 18.3: Line Plot of Train and Test Loss During Training With Simple Early Stopping.

We can improve the trigger for early stopping by waiting a while before stopping. This can be achieved by setting the `patience` argument. In this case, we will wait 200 epochs before training is stopped. Specifically, this means that we will allow training to continue for up to an additional 200 epochs after the point that validation loss started to degrade, giving the training process an opportunity to get across flat spots or find some additional improvement.

```
# patient early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=200)
```

Listing 18.33: Example of fitting the MLP with patient early stopping.

The complete example with this change is listed below.

```
# mlp overfit on the moons dataset with patient early stopping
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import EarlyStopping
from matplotlib import pyplot
# generate 2d classification dataset
```

```

X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# patient early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=200)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0,
                     callbacks=[es])
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 18.34: Example of MLP with patient early stopping fit on the two moons problem.

Running the example, we can see that training was stopped much later, in this case just before epoch 1,000.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can also see that the performance on the test dataset is better than not using any early stopping.

```

Epoch 00986: early stopping
Train: 1.000, Test: 0.943

```

Listing 18.35: Example output fitting an MLP with patient early stopping on the two moons problem.

Reviewing the line plot of loss during training, we can see that the patience allowed the training to progress past some small flat and bad spots.

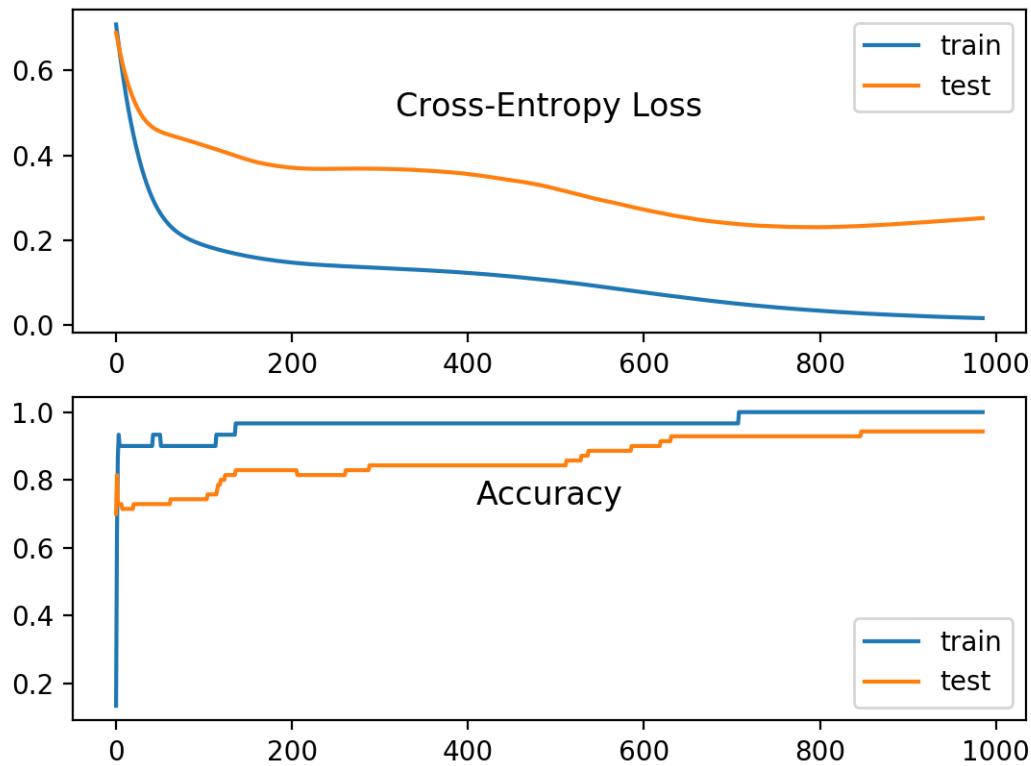


Figure 18.4: Line Plot of Train and Test Loss During Training With Patient Early Stopping.

We can also see that test loss started to increase again in the last approximately 100 epochs. This means that although the performance of the model has improved, we may not have the best performing or most stable model at the end of training. We can address this by using a `ModelCheckpoint` callback. In this case, we are interested in saving the model with the best accuracy on the test dataset. We could also seek the model with the best loss on the test dataset, but this may or may not correspond to the model with the best accuracy.

This highlights an important concept in model selection. The notion of the *best* model during training may conflict when evaluated using different performance measures. Try to choose models based on the metric by which they will be evaluated and presented in the domain. In a balanced binary classification problem, this will most likely be classification accuracy. Therefore, we will use accuracy on the validation in the `ModelCheckpoint` callback to save the best model observed during training.

```
mc = ModelCheckpoint('best_model.h5', monitor='val_accuracy', mode='max', verbose=1,
                     save_best_only=True)
```

Listing 18.36: Example of a model checkpoint callback for the MLP with early stopping.

During training, the entire model will be saved to the file `best_model.h5` only when accuracy on the validation dataset improves overall across the entire training process. A verbose output will also inform us as to the epoch and accuracy value each time the model is saved to the same

file (e.g. overwritten). This new additional callback can be added to the list of callbacks when calling the `fit()` function.

```
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0,
                     callbacks=[es, mc])
```

Listing 18.37: Example of fitting the MLP model with early stopping and model checkpoint callbacks.

We are no longer interested in the line plot of loss during training; it will be much the same as the previous run. Instead, we want to load the saved model from file and evaluate its performance on the test dataset.

```
# load the saved model
saved_model = load_model('best_model.h5')
# evaluate the model
_, train_acc = saved_model.evaluate(trainX, trainy, verbose=0)
_, test_acc = saved_model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 18.38: Example of loading and evaluating the saved model checkpoint.

The complete example with these changes is listed below.

```
# mlp overfit on the moons dataset with patient early stopping and model checkpointing
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import EarlyStopping
from keras.callbacks import ModelCheckpoint
from keras.models import load_model
# generate 2d classification dataset
X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
# split into train and test
n_train = 30
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(500, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# simple early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=200)
mc = ModelCheckpoint('best_model.h5', monitor='val_accuracy', mode='max', verbose=1,
                     save_best_only=True)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=4000, verbose=0,
                     callbacks=[es, mc])
# load the saved model
saved_model = load_model('best_model.h5')
# evaluate the model
_, train_acc = saved_model.evaluate(trainX, trainy, verbose=0)
_, test_acc = saved_model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 18.39: Example of MLP with patient early stopping with model checkpointing fit on the two moons problem.

Running the example, we can see the verbose output from the `ModelCheckpoint` callback for both when a new best model is saved and from when no improvement was observed. We can see that the best model was observed at epoch 879 during this run.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

Again, we can see that early stopping continued patiently until after epoch 1,000. Note that epoch $880 + \text{patience}$ of 200 is not epoch 1,044. Recall that early stopping is monitoring loss on the validation dataset and that the model checkpoint is saving models based on accuracy. As such, the patience of early stopping started at an epoch other than 880.

```
...
Epoch 00878: val_acc did not improve from 0.92857
Epoch 00879: val_acc improved from 0.92857 to 0.94286, saving model to best_model.h5
Epoch 00880: val_acc did not improve from 0.94286
...
Epoch 01042: val_acc did not improve from 0.94286
Epoch 01043: val_acc did not improve from 0.94286
Epoch 01044: val_acc did not improve from 0.94286
Epoch 01044: early stopping
Train: 1.000, Test: 0.943
```

Listing 18.40: Example output fitting an MLP with patient early stopping with model checkpointing on the two moons problem.

In this case, we don't see any further improvement in model accuracy on the test dataset. Nevertheless, we have followed a good practice. *Why not monitor validation accuracy for early stopping?* This is a good question. The main reason is that accuracy is a coarse measure of model performance during training and that loss provides more nuance when using early stopping with classification problems. The same measure may be used for early stopping and model checkpointing in the case of regression, such as mean squared error.

18.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Use Accuracy.** Update the example to monitor accuracy on the test dataset rather than loss, and plot learning curves showing accuracy.
- **Use True Validation Set.** Update the example to split the training set into train and validation sets, then evaluate the model on the test dataset.
- **Regression Example.** Create a new example of using early stopping to address overfitting on a simple regression problem and monitoring mean squared error.

If you explore any of these extensions, I'd love to know.

18.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

18.5.1 Books

- Section 7.8: Early Stopping, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- Section 5.5.2: Early stopping, *Pattern Recognition and Machine Learning*, 2006.
<https://amzn.to/2Q2rEeP>
- Section 16.1: Early Stopping, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
<https://amzn.to/2poq0xc>

18.5.2 Papers

- *Early Stopping — But When?*, 2002.
https://link.springer.com/chapter/10.1007/3-540-49430-8_3
- *Improving model selection by nonconvergent methods*, 1993.
<https://www.sciencedirect.com/science/article/pii/S0893608005801224>
- *Automatic early stopping using cross validation: quantifying the criteria*, 1997.
<https://www.sciencedirect.com/science/article/pii/S0893608098000100>
- *Understanding deep learning requires rethinking generalization*, 2017.
<https://arxiv.org/abs/1611.03530>

18.5.3 APIs

- H5Py Installation Documentation.
<http://docs.h5py.org/en/latest/build.html>
- Keras Regularizers API.
<https://keras.io/regularizers/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- Keras Convolutional Layers API.
<https://keras.io/layers/convolutional/>
- Keras Recurrent Layers API.
<https://keras.io/layers/recurrent/>
- Keras Callbacks API.
<https://keras.io/callbacks/>
- `sklearn.datasets.make_moons` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html

18.5.4 Articles

- Early stopping, Wikipedia.
https://en.wikipedia.org/wiki/Early_stopping

18.6 Summary

In this tutorial, you discovered that stopping the training of neural network early before it has overfit the training dataset can reduce overfitting and improve the generalization of deep neural networks. Specifically, you learned:

- The challenge of training a neural network long enough to learn the mapping, but not so long that it overfits the training data.
- Model performance on a holdout validation dataset can be monitored during training and training stopped when generalization error starts to increase.
- The use of early stopping requires the selection of a performance measure to monitor, a trigger for stopping training, and a selection of the model weights to use.

18.6.1 Next

This is the end of the Part on Generalization. In the next Part, you will discover techniques for improving the predictions from your neural network model.

Part III

Better Predictions

Overview

In this part you will discover how to make better predictions with final models using model ensembles. After reading the chapters in this part, you will know:

- How to reduce the variance of final models and improve the skill of predictions with model ensembles (Chapter [19](#)).
- How to average the predictions from multiple models (Chapter [20](#)).
- How to weight the predictions from multiple models in proportion to the trust in each model (Chapter [21](#)).
- How to develop ensemble members from the resampling methods used to estimate model performance (Chapter [22](#)).
- How to develop ensemble members from contiguous epochs from a single model training run (Chapter [23](#)).
- How to develop ensemble members from the trough of an aggressive cyclical learning rate schedule across a single training run (Chapter [24](#)).
- How to develop a new model to learn how to best combine the predictions from multiple ensemble members (Chapter [25](#)).
- How to combine the model parameters or weights instead of the predictions from multiple ensemble members (Chapter [26](#)).

Chapter 19

Reduce Model Variance with Ensemble Learning

Deep learning neural networks are nonlinear methods. They offer increased flexibility and can scale in proportion to the amount of training data available. A downside of this flexibility is that they learn via a stochastic training algorithm which means that they are sensitive to the specifics of the training data and random initialization, and may find a different set of weights each time they are trained, which in turn produce different predictions. Generally, this is referred to as neural networks having a high variance and it can be frustrating when trying to develop a final model to use for making predictions.

A successful approach to reducing the variance of neural network models is to train multiple models instead of a single model and to combine the predictions from these models. This is called ensemble learning and not only reduces the variance of predictions but also can result in predictions that are better than any single model. In this tutorial, you will discover methods for deep learning neural networks to reduce variance and improve prediction performance. After reading this tutorial, you will know:

- Neural network models are nonlinear and have a high variance, which can be frustrating when preparing a final model for making predictions.
- Ensemble learning combines the predictions from multiple neural network models to reduce the variance of predictions and reduce generalization error.
- Techniques for ensemble learning can be grouped by the element that is varied, such as training data, the model, and how predictions are combined.

Let's get started.

19.1 High Variance of Neural Network Models

Training deep neural networks can be very computationally expensive. Very deep networks trained on millions of examples may take days, weeks, and sometimes months to train.

Google's baseline model [...] was a deep convolutional neural network [...] that had been trained for about six months using asynchronous stochastic gradient descent on a large number of cores.

— *Distilling the Knowledge in a Neural Network*, 2015.

After the investment of so much time and resources, there is no guarantee that the final model will have low generalization error, performing well on examples not seen during training.

... train many different candidate networks and then to select the best, [...] and to discard the rest. There are two disadvantages with such an approach. First, all of the effort involved in training the remaining networks is wasted. Second, [...] the network which had best performance on the validation set might not be the one with the best performance on new test data.

— Pages 364-365, *Neural Networks for Pattern Recognition*, 1995.

Neural network models are a nonlinear method. This means that they can learn complex nonlinear relationships in the data. A downside of this flexibility is that they are sensitive to initial conditions, both in terms of the initial random weights and in terms of the statistical noise in the training dataset. This stochastic nature of the learning algorithm means that each time a neural network model is trained, it may learn a slightly (or dramatically) different version of the mapping function from inputs to outputs, that in turn will have different performance on the training and holdout datasets. As such, we can think of a neural network as a method that has a low bias and high variance. Even when trained on large datasets to satisfy the high variance, having any variance in a final model that is intended to be used to make predictions can be frustrating.

19.2 Reduce Variance Using an Ensemble of Models

A solution to the high variance of neural networks is to train multiple models and combine their predictions. The idea is to combine the predictions from multiple good but different models. A good model has skill, meaning that its predictions are better than random chance. Importantly, the models must be good in different ways; they must make different prediction errors.

The reason that model averaging works is that different models will usually not make all the same errors on the test set.

— Page 256, *Deep Learning*, 2016.

Combining the predictions from multiple neural networks adds a bias that in turn counters the variance of a single trained neural network model. The results are predictions that are less sensitive to the specifics of the training data, choice of training scheme, and the serendipity of a single training run. In addition to reducing the variance in the prediction, the ensemble can also result in better predictions than any single best model.

... the performance of a committee can be better than the performance of the best single network used in isolation.

— Page 365, *Neural Networks for Pattern Recognition*, 1995.

This approach belongs to a general class of methods called *ensemble learning* that describes methods that attempt to make the best use of the predictions from multiple models prepared for the same problem. Generally, ensemble learning involves training more than one network on the same dataset, then using each of the trained models to make a prediction before combining the predictions in some way to make a final outcome or prediction.

In fact, ensembling of models is a standard approach in applied machine learning to ensure that the most stable and best possible prediction is made. For example, Alex Krizhevsky, et al. in their famous 2012 paper titled *Imagenet classification with deep convolutional neural networks* that introduced very deep convolutional neural networks for photo classification (i.e. AlexNet) used model averaging across multiple well-performing CNN models to achieve state-of-the-art results at the time. Performance of one model was compared to ensemble predictions averaged over two, five, and seven different models.

Averaging the predictions of five similar CNNs gives an error rate of 16.4%. [...] Averaging the predictions of two CNNs that were pre-trained [...] with the aforementioned five CNNs gives an error rate of 15.3%.

Ensembling is also the approach used by winners in machine learning competitions.

Another powerful technique for obtaining the best possible results on a task is model ensembling. [...] If you look at machine-learning competitions, in particular on Kaggle, you'll see that the winners use very large ensembles of models that inevitably beat any single model, no matter how good.

— Page 264, *Deep Learning With Python*, 2017.

19.3 How to Ensemble Neural Network Models

Perhaps the oldest and still most commonly used ensembling approach for neural networks is called a *committee of networks*. A collection of networks with the same configuration and different initial random weights is trained on the same dataset. Each model is then used to make a prediction and the actual prediction is calculated as the average of the predictions. The number of models in the ensemble is often kept small both because of the computational expense in training models and because of the diminishing returns in performance from adding more ensemble members. Ensembles may be as small as three, five, or 10 trained models. The field of ensemble learning is well studied and there are many variations on this simple theme. It can be helpful to think of varying each of the three major elements of the ensemble method; for example:

- **Training Data:** Vary the choice of data used to train each model in the ensemble.
- **Ensemble Models:** Vary the choice of the models used in the ensemble.
- **Combinations:** Vary the choice of the way that outcomes from ensemble members are combined.

Let's take a closer look at each element in turn.

19.3.1 Varying Training Data

The data used to train each member of the ensemble can be varied. The simplest approach would be to use k -fold cross-validation to estimate the generalization error of the chosen model configuration. In this procedure, k different models are trained on k different subsets of the training data. These k models can then be saved and used as members of an ensemble. Another popular approach involves resampling the training dataset with replacement, then training a network using the resampled dataset. The resampling procedure means that the composition of each training dataset is different with the possibility of duplicated examples allowing the model trained on the dataset to have a slightly different expectation of the density of the samples, and in turn different generalization error.

This approach is called bootstrap aggregation, or bagging for short, and was designed for use with unpruned decision trees that have high variance and low bias. Typically a large number of decision trees are used, such as hundreds or thousands, given that they are fast to prepare.

... a natural way to reduce the variance and hence increase the prediction accuracy of a statistical learning method is to take many training sets from the population, build a separate prediction model using each training set, and average the resulting predictions. [...] Of course, this is not practical because we generally do not have access to multiple training sets. Instead, we can bootstrap, by taking repeated samples from the (single) training data set.

— Pages 216-317, *An Introduction to Statistical Learning with Applications in R*, 2013.

An equivalent approach might be to use a smaller subset of the training dataset without regularization to allow faster training and some overfitting. The desire for slightly under-optimized models applies to the selection of ensemble members more generally.

... the members of the committee should not individually be chosen to have optimal trade-off between bias and variance, but should have relatively smaller bias, since the extra variance can be removed by averaging.

— Page 366, *Neural Networks for Pattern Recognition*, 1995.

Other approaches may involve selecting a random subspace of the input space to allocate to each model, such as a subset of the hyper-volume in the input space or a subset of input features.

19.3.2 Varying Models

Training the same under-constrained model on the same data with different initial conditions will result in different models given the difficulty of the problem, and the stochastic nature of the learning algorithm. This is because the optimization problem that the network is trying to solve is so challenging that there are many *good* and *different* solutions to map inputs to outputs.

Most neural network algorithms achieve sub-optimal performance specifically due to the existence of an overwhelming number of sub-optimal local minima. If we take a set of neural networks which have converged to local minima and apply averaging we can construct an improved estimate. One way to understand this fact is to consider that, in general, networks which have fallen into different local minima will perform poorly in different regions of feature space and thus their error terms will not be strongly correlated.

— *When networks disagree: Ensemble methods for hybrid neural networks*, 1995.

This may result in a reduced variance, but may not dramatically improve generalization error. The errors made by the models may still be too highly correlated because the models all have learned similar mapping functions. An alternative approach might be to vary the configuration of each ensemble model, such as using networks with different capacity (e.g. number of layers or nodes) or models trained under different conditions (e.g. learning rate or regularization). The result may be an ensemble of models that have learned a more heterogeneous collection of mapping functions and in turn have a lower correlation in their predictions and prediction errors.

Differences in random initialization, random selection of minibatches, differences in hyperparameters, or different outcomes of non-deterministic implementations of neural networks are often enough to cause different members of the ensemble to make partially independent errors.

— Pages 257-258, *Deep Learning*, 2016.

Such an ensemble of differently configured models can be achieved through the normal process of developing the network and tuning its hyperparameters. Each model could be saved during this process and a subset of better models chosen to comprise the ensemble.

Slightly inferiorly trained networks are a free by-product of most tuning algorithms; it is desirable to use such extra copies even when their performance is significantly worse than the best performance found. Better performance yet can be achieved through careful planning for an ensemble classification by using the best available parameters and training different copies on different subsets of the available database.

— *Neural Network Ensembles*, 1990.

In cases where a single model may take weeks or months to train, another alternative may be to periodically save the best model during the training process, called snapshot or checkpoint models, then select ensemble members among the saved models. This provides the benefits of having multiple models trained on the same data, although collected during a single training run.

Snapshot Ensembling produces an ensemble of accurate and diverse models from a single training process. At the heart of Snapshot Ensembling is an optimization process which visits several local minima before converging to a final solution. We take model snapshots at these various minima, and average their predictions at test time.

— *Snapshot Ensembles: Train 1, get M for free*, 2017.

A variation on the Snapshot ensemble is to save models from a range of epochs, perhaps identified by reviewing learning curves of model performance on the train and validation datasets during training. Ensembles from such contiguous sequences of models are referred to as horizontal ensembles.

First, networks trained for a relatively stable range of epoch are selected. The predictions of the probability of each label are produced by standard classifiers [over] the selected epoch[s], and then averaged.

— *Horizontal and vertical ensemble with deep representation for classification*, 2013.

A further enhancement of the snapshot ensemble is to systematically vary the optimization procedure during training to force different solutions (i.e. sets of weights), the best of which can be saved to checkpoints. This might involve injecting an oscillating amount of noise over training epochs or oscillating the learning rate during training epochs. A variation of this approach called Stochastic Gradient Descent with Warm Restarts (SGDR) demonstrated faster learning and state-of-the-art results for standard photo classification tasks.

Our SGDR simulates warm restarts by scheduling the learning rate to achieve competitive results [...] roughly two to four times faster. We also achieved new state-of-the-art results with SGDR, mainly by using even wider [models] and ensembles of snapshots from SGDR’s trajectory.

— *SGDR: Stochastic Gradient Descent with Warm Restarts*, 2016.

A benefit of very deep neural networks is that the intermediate hidden layers provide a learned representation of the low-resolution input data. The hidden layers can output their internal representations directly, and the output from one or more hidden layers from one very deep network can be used as input to a new classification model. This is perhaps most effective when the deep model is trained using an autoencoder model. This type of ensemble is referred to as a vertical ensemble.

This method ensembles a series of classifiers whose inputs are the representation of intermediate layers. A lower error rate is expected because these features seem diverse.

— *Horizontal and vertical ensemble with deep representation for classification*, 2013.

19.3.3 Varying Combinations

The simplest way to combine the predictions is to calculate the average of the predictions from the ensemble members. This can be improved slightly by weighting the predictions from each model, where the weights are optimized using a hold-out validation dataset. This provides a weighted average ensemble that is sometimes called model blending.

... we might expect that some members of the committee will typically make better predictions than other members. We would therefore expect to be able to reduce the error still further if we give greater weight to some committee members than to others. Thus, we consider a generalized committee prediction given by a weighted combination of the predictions of the members ...

— Page 367, *Neural Networks for Pattern Recognition*, 1995.

One further step in complexity involves using a new model to learn how to best combine the predictions from each ensemble member. The model could be a simple linear model (e.g. much like the weighted average), but could be a sophisticated nonlinear method that also considers the specific input sample in addition to the predictions provided by each member. This general approach of learning a new model is called model stacking, or stacked generalization.

Stacked generalization works by deducing the biases of the generalizer(s) with respect to a provided learning set. This deduction proceeds by generalizing in a second space whose inputs are (for example) the guesses of the original generalizers when taught with part of the learning set and trying to guess the rest of it, and whose output is (for example) the correct guess. [...] When used with a single generalizer, stacked generalization is a scheme for estimating (and then correcting for) the error of a generalizer which has been trained on a particular learning set and then asked a particular question.

— *Stacked generalization*, 1992.

There are more sophisticated methods for stacking models, such as boosting where ensemble members are added one at a time in order to correct the mistakes of prior models. The added complexity means this approach is less often used with large neural network models. Another combination that is a little bit different is to combine the weights of multiple neural networks with the same structure. The weights of multiple networks can be averaged, to hopefully result in a new single model that has better overall performance than any original model. This approach is called model weight averaging.

... suggests it is promising to average these points in weight space, and use a network with these averaged weights, instead of forming an ensemble by averaging the outputs of networks in model space

— *Averaging Weights Leads to Wider Optima and Better Generalization*, 2018.

19.4 Summary of Ensemble Techniques

In summary, we can list some of the more common and interesting ensemble methods for neural networks organized by each element of the method that can be varied, as follows:

- Varying Training Data (Chapter 22)
 - k -fold Cross-Validation Ensemble.

- Bootstrap Aggregation (bagging) Ensemble.
- Random Training Subset Ensemble.

- Varying Models

- Multiple Training Run Ensemble (Chapter 20 and Chapter 21).
- Hyperparameter Tuning Ensemble.
- Snapshot Ensemble (Chapter 24).
- Horizontal Epochs Ensemble (Chapter 23).
- Vertical Representational Ensemble.

- Varying Combinations

- Model Averaging Ensemble (Chapter 20).
- Weighted Average Ensemble (Chapter 21).
- Stacked Generalization (stacking) Ensemble (Chapter 25).
- Boosting Ensemble.
- Model Weight Averaging Ensemble (Chapter 26).

There is no single best ensemble method; perhaps experiment with a few approaches or let the constraints of your project guide you.

19.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

19.5.1 Books

- Section 9.6 Committees of networks, *Neural Networks for Pattern Recognition*, 1995.
<https://amzn.to/2I9gNMP>
- Section 7.11 Bagging and Other Ensemble Methods, *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- Section 7.3.3 Model ensembling, *Deep Learning With Python*, 2017.
<https://amzn.to/2NJq1pf>
- Section 8.2 Bagging, Random Forests, Boosting, *An Introduction to Statistical Learning with Applications in R*, 2013.
<https://amzn.to/2zxHR5E>

19.5.2 Papers

- *Neural Network Ensembles*, 1990.
<https://ieeexplore.ieee.org/abstract/document/58871/>
- *Neural Network Ensembles, Cross Validation, and Active Learning*, 1994.
<https://dl.acm.org/citation.cfm?id=2998716>
- *When networks disagree: Ensemble methods for hybrid neural networks*, 1995.
https://www.worldscientific.com/doi/abs/10.1142/9789812795885_0025
- *Snapshot Ensembles: Train 1, get M for free*, 2017.
<https://arxiv.org/abs/1704.00109>
- *SGDR: Stochastic Gradient Descent with Warm Restarts*, 2016.
<https://arxiv.org/abs/1608.03983>
- *Horizontal and vertical ensemble with deep representation for classification*, 2013.
<https://arxiv.org/abs/1306.2759>
- *Stacked generalization*, 1992.
<https://www.sciencedirect.com/science/article/pii/S0893608005800231>
- *Averaging Weights Leads to Wider Optima and Better Generalization*, 2018.
<https://arxiv.org/abs/1803.05407>

19.5.3 Articles

- Ensemble learning, Wikipedia.
https://en.wikipedia.org/wiki/Ensemble_learning
- Bootstrap aggregating, Wikipedia.
https://en.wikipedia.org/wiki/Bootstrap_aggregating
- Boosting (machine learning), Wikipedia.
[https://en.wikipedia.org/wiki/Boosting_\(machine_learning\)](https://en.wikipedia.org/wiki/Boosting_(machine_learning))

19.6 Summary

In this tutorial, you discovered ensemble methods for deep learning neural networks to reduce variance and improve prediction performance. Specifically, you learned:

- Neural network models are nonlinear and have a high variance, which can be frustrating when preparing a final model for making predictions.
- Ensemble learning combines the predictions from multiple neural network models to reduce the variance of predictions and reduce generalization error.
- Techniques for ensemble learning can be grouped by the element that is varied, such as training data, the model, and how predictions are combined.

19.6.1 Next

In the next tutorial, you will discover how to average the predictions made by multiple neural network models.

Chapter 20

Combine Models From Multiple Runs with Model Averaging Ensemble

Deep learning neural network models are highly flexible nonlinear algorithms capable of learning a near infinite number of mapping functions. A frustration with this flexibility is the high variance in a final model. The same neural network model trained on the same dataset may find one of many different possible *good enough* solutions each time it is run. Model averaging is an ensemble learning technique that reduces the variance in a final neural network model, sacrificing spread (and possibly better scores) in the performance of the model for a confidence in what performance to expect from the model. In this tutorial, you will discover how to develop a model averaging ensemble in Keras to reduce the variance in a final model. After completing this tutorial, you will know:

- Model averaging is an ensemble learning technique that can be used to reduce the expected variance of deep learning neural network models.
- How to implement model averaging in Keras for classification and regression predictive modeling problems.
- How to work through a multiclass classification problem and use model averaging to reduce the variance of the final model.

Let's get started.

20.1 Model Averaging Ensemble

Deep learning neural network models are nonlinear methods that learn via a stochastic training algorithm. This means that they are highly flexible, capable of learning complex relationships between variables and approximating any mapping function, given enough resources. A downside of this flexibility is that the models suffer high variance. This means that the models are highly dependent on the specific training data used to train the model and on the initial conditions (random initial weights) and serendipity during the training process. The result is a final model that makes different predictions each time the same model configuration is trained on the same dataset.

This can be frustrating when training a final model for use in making predictions on new data, such as operationally or in a machine learning competition. The high variance of the approach can be addressed by training multiple models for the problem and combining their predictions. This approach is called model averaging and belongs to a family of techniques called ensemble learning.

20.2 Ensembles in Keras

The simplest way to develop a model averaging ensemble in Keras is to train multiple models on the same dataset then combine the predictions from each of the trained models.

20.2.1 Train Multiple Models

Training multiple models may be resource intensive, depending on the size of the model and the size of the training data. You may have to train the models sequentially on the same hardware. For very large models, it may be worth training the models in parallel using cloud infrastructure such as Amazon Web Services.

The number of models required for the ensemble may vary based on the complexity of the problem and model. A benefit of the approach is that you can continue to create models, add them to the ensemble, and evaluate their impact on the performance by making predictions on a holdout test set. For small models, you can train the models sequentially and keep them in memory for use in your experiment. For example:

```
...
# train models and keep them in memory
n_members = 10
models = list()
for _ in range(n_members):
    # define and fit model
    model = ...
    # store model in memory as ensemble member
    models.add(model)
...
...
```

Listing 20.1: Example of training multiple models in memory.

For large models, perhaps trained on different hardware, you can save each model to file.

```
...
# train models and keep them to file
n_members = 10
for i in range(n_members):
    # define and fit model
    model = ...
    # save model to file
    filename = 'model_' + str(i + 1) + '.h5'
    model.save(filename)
    print('Saved: %s' % filename)
...
...
```

Listing 20.2: Example of training multiple models and saving them to file.

Models can then be loaded later. Small models can all be loaded into memory at the same time, whereas very large models may have to be loaded one at a time to make a prediction, then later to have the predictions combined.

```
from keras.models import load_model
...
# load pre-trained ensemble members
n_members = 10
models = list()
for i in range(n_members):
    # load model
    filename = 'model_' + str(i + 1) + '.h5'
    model = load_model(filename)
    # store in memory
    models.append(model)
...
```

Listing 20.3: Example of loading multiple models from file.

20.2.2 Combine Predictions

Once the models have been prepared, each model can be used to make a prediction and the predictions can be combined. In the case of a regression problem where each model is predicting a real-valued output, the values can be collected and the average calculated.

```
...
# make predictions
yhats = [model.predict(testX) for model in models]
yhats = array(yhats)
# calculate average
outcomes = mean(yhats)
```

Listing 20.4: Example of averaging predictions for a regression problem.

In the case of a classification problem, there are two options: to combine the predicted class labels or to combine the predicted probabilities. The class labels can be combined by calculating the statistical mode (most frequent value), for example:

```
...
# make predictions
yhats = [model.predict_classes(testX) for model in models]
yhats = array(yhats)
# calculate mode
outcomes, _ = mode(yhats)
```

Listing 20.5: Example of calculating the mode for a classification problem.

A downside of this approach is that for small ensembles or problems with a large number of classes, the sample of predictions may not be large enough for the mode to be meaningful. In the case of a binary classification problem, a sigmoid activation function is used on the output layer and the average of the predicted probabilities can be calculated much like a regression problem. In the case of a multiclass classification problem with more than two classes, a softmax activation function is used on the output layer and the sum of the probabilities for each predicted class can be calculated before taking the argmax to get the class value, for example:

```

...
# make predictions
yhats = [model.predict(testX) for model in models]
yhats = array(yhats)
# sum across ensembles
summed = numpy.sum(yhats, axis=0)
# argmax across classes
outcomes = argmax(summed, axis=1)

```

Listing 20.6: Example of calculating the argmax for class probabilities.

These approaches for combining predictions of Keras models will work just as well for Multilayer Perceptron, Convolutional, and Recurrent Neural Networks. Now that we know how to average predictions from multiple neural network models in Keras, let's work through a case study.

20.3 Model Averaging Ensemble Case Study

In this section, we will demonstrate how to use the model average ensemble to reduce the variance of an MLP on a simple multiclass classification problem. This example provides a template for applying the model average ensemble to your own neural network for classification and regression problems.

20.3.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate a model averaging ensemble. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. We use this problem with 500 examples, with two input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same 500 points.

```

# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2, cluster_std=2, random_state=2)

```

Listing 20.7: Example of generating samples for the blobs problem.

The results are the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can graph each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```

# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot for each class value
for class_value in range(3):
    # select indices of points with the class label
    row_ix = where(y == class_value)

```

```
# scatter plot for points with a different color
pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 20.8: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (separable by a line) causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions resulting in a high variance.

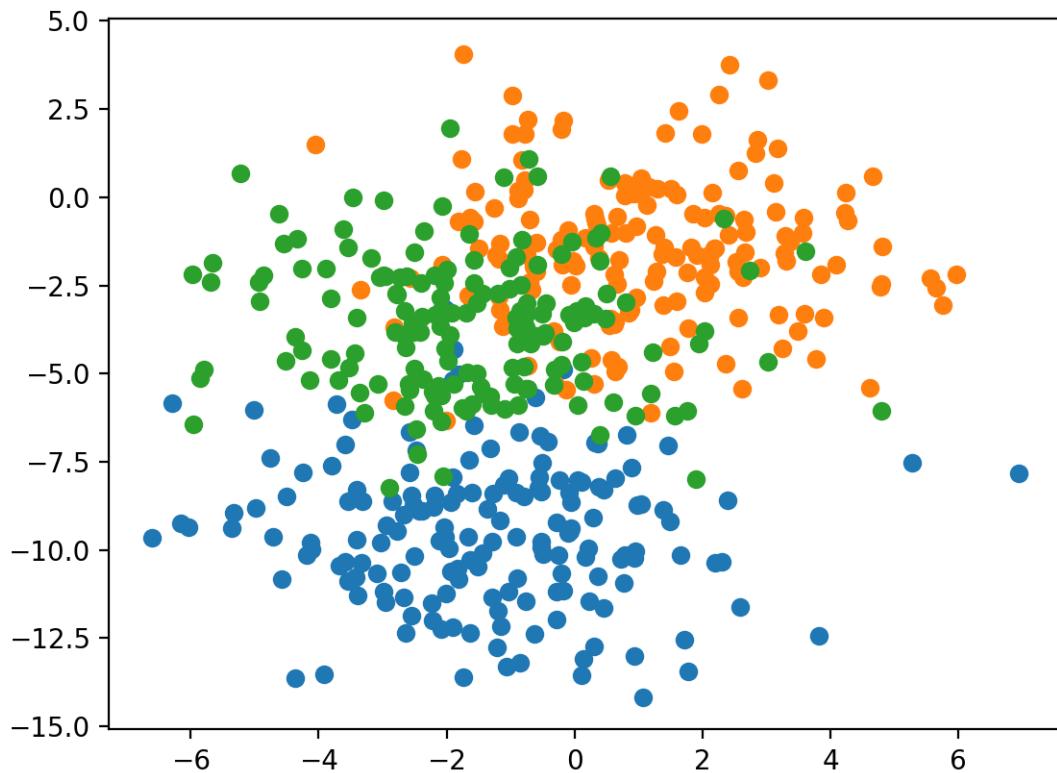


Figure 20.1: Scatter Plot of Blobs Dataset with Three Classes and Points Colored by Class Value.

20.3.2 MLP Model for Multiclass Classification

Now that we have defined a problem, we can define a model to address it. We will define a model that is perhaps under-constrained and not tuned to the problem. This is intentional to demonstrate the high variance of a neural network model seen on truly large and challenging supervised learning problems. The problem is a multiclass classification problem, and we will model it using a softmax activation function on the output layer. This means that the model

will predict a vector with 3 elements with the probability that the sample belongs to each of the 3 classes. Therefore, the first step is to one hot encode the class values.

```
y = to_categorical(y)
```

Listing 20.9: Example of one hot encoding the target variable.

Next, we must split the dataset into training and test sets. We will use the test set both to evaluate the performance of the model and to plot its performance during training with a learning curve. We will use 30% of the data for training and 70% for the test set. This is an example of a challenging problem where we have more unlabeled examples than we do labeled examples.

```
# split into train and test
n_train = int(0.3 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 20.10: Example of preparing the dataset for modeling.

Next, we can define and compile the model. The model will expect samples with two input variables. The model then has a single hidden layer with 15 nodes and a rectified linear activation function, then an output layer with 3 nodes to predict the probability of each of the 3 classes and a softmax activation function. Because the problem is multiclass, we will use the categorical cross-entropy loss function to optimize the model and the efficient Adam flavor of stochastic gradient descent.

```
# define model
model = Sequential()
model.add(Dense(15, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 20.11: Example of defining the MLP model.

The model is fit for 200 training epochs and we will evaluate the model each epoch on the test set, using the test set as a validation set.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
```

Listing 20.12: Example of fitting the MLP model.

At the end of the run, we will evaluate the performance of the model on both the train and the test sets.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 20.13: Example of evaluate the MLP model.

Then finally, we will plot learning curves of the model loss and accuracy over each training epoch on both the training and test dataset.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 20.14: Example of plotting model performance.

The complete example is listed below.

```
# fit high variance mlp on blobs classification problem
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2, cluster_std=2, random_state=2)
y = to_categorical(y)
# split into train and test
n_train = int(0.3 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(15, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 20.15: Example of fitting an MLP on the blobs problem.

Running the example first prints the performance of the final model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved about 84% accuracy on the training dataset and about 76 % accuracy on the test dataset; not terrible.

Train: 0.847, Test: 0.763

Listing 20.16: Example output fitting an MLP on the blobs problem.

A line plot is also created showing the learning curves for the model accuracy on the train and test sets over each training epoch. We can see that the model is not really overfit, but is perhaps a little underfit and may benefit from an increase in capacity, more training, and perhaps some regularization. We intentionally hold back all of these improvements to force high variance for our case study.

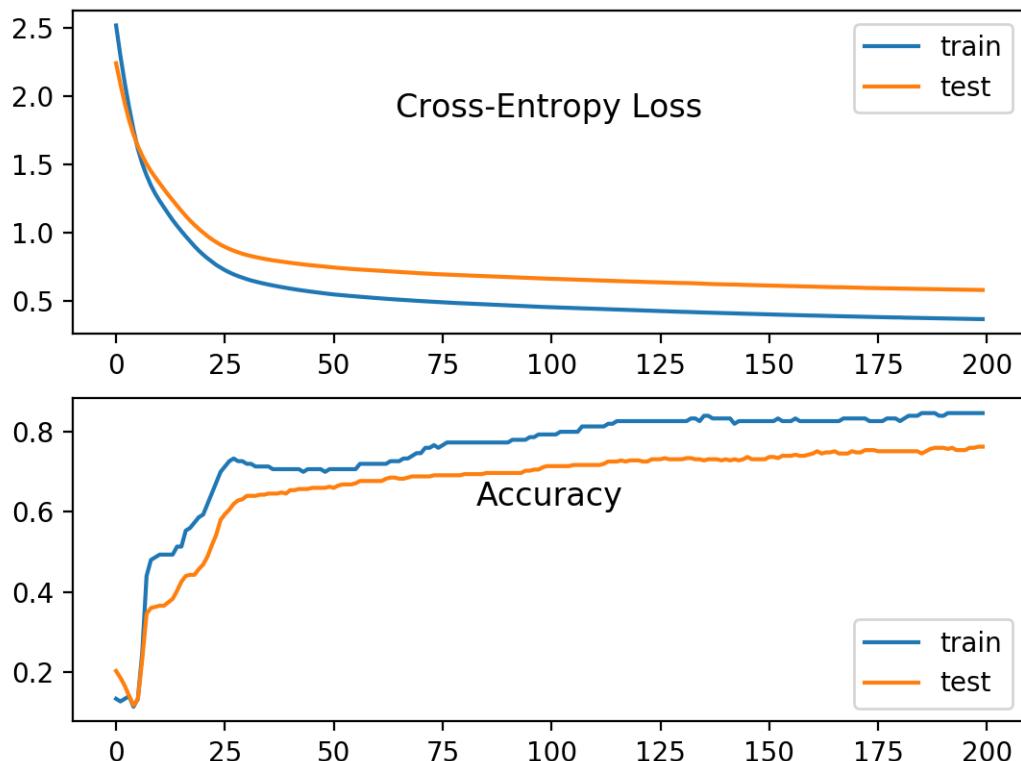


Figure 20.2: Line Plot Learning Curves of Model Accuracy on Train and Test Dataset Over Each Training Epoch.

20.3.3 High Variance of MLP Model

It is important to demonstrate that the model indeed has a variance in its prediction. We can demonstrate this by repeating the fit and evaluation of the same model configuration on the same dataset and summarizing the final performance of the model. To do this, we first split the fit and evaluation of the model out as a function that we can call repeatedly. The `evaluate_model()` function below takes the train and test dataset, fits a model, then evaluates it, returning the accuracy of the model on the test dataset.

```
# fit and evaluate a neural net model on the dataset
def evaluate_model(trainX, trainy, testX, testy):
    # define model
    model = Sequential()
    model.add(Dense(15, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=200, verbose=0)
    # evaluate the model
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    return test_acc
```

Listing 20.17: Example of a function to fit and evaluate an MLP model.

We can call this function 30 times, saving the test accuracy scores.

```
# repeated evaluation
n_repeats = 30
scores = list()
for _ in range(n_repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    print('> %.3f' % score)
    scores.append(score)
```

Listing 20.18: Example of repeated evaluation of a model.

Once collected, we can summarize the distribution scores, first in terms of the mean and standard deviation, assuming the distribution is Gaussian, which is very reasonable.

```
# summarize the distribution of scores
print('Scores Mean: %.3f, Standard Deviation: %.3f' % (mean(scores), std(scores)))
```

Listing 20.19: Example of summarizing scores from repeated evaluation.

We can then summarize the distribution both as a histogram to show the shape of the distribution and as a box and whisker plot to show the spread and body of the distribution.

```
# histogram of distribution
pyplot.hist(scores, bins=10)
pyplot.show()
# boxplot of distribution
pyplot.boxplot(scores)
pyplot.show()
```

Listing 20.20: Example of plotting the distribution of scores from repeated evaluation.

The complete example of summarizing the variance of the MLP model on the chosen blobs dataset is listed below.

```

# demonstrate high variance of mlp model on blobs classification problem
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from numpy import mean
from numpy import std
from matplotlib import pyplot

# fit and evaluate a neural net model on the dataset
def evaluate_model(trainX, trainy, testX, testy):
    # define model
    model = Sequential()
    model.add(Dense(15, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=200, verbose=0)
    # evaluate the model
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    return test_acc

# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2, cluster_std=2, random_state=2)
y = to_categorical(y)
# split into train and test
n_train = int(0.3 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# repeated evaluation
n_repeats = 30
scores = list()
for _ in range(n_repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    print('> %.3f' % score)
    scores.append(score)
# summarize the distribution of scores
print('Scores Mean: %.3f, Standard Deviation: %.3f' % (mean(scores), std(scores)))
# histogram of distribution
pyplot.hist(scores, bins=10)
pyplot.show()
# boxplot of distribution
pyplot.boxplot(scores)
pyplot.show()

```

Listing 20.21: Example of repeated evaluation of an MLP on the blobs problem.

Running the example first prints the accuracy of each model on the test set, finishing with the mean and standard deviation of the sample of accuracy scores.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the average of the sample is 77% with a standard deviation of about 1.4%. Assuming a Gaussian distribution, we would expect 99% of accuracy scores to fall

between about 73% and 81% (i.e. 3 standard deviations above and below the mean). We can take the standard deviation of the accuracy of the model on the test set as an estimate for the variance of the predictions made by the model.

```
...
> 0.751
> 0.789
> 0.791
> 0.766
> 0.766
Scores Mean: 0.770, Standard Deviation: 0.014
```

Listing 20.22: Example output from repeated evaluation an MLP on the blobs problem.

A histogram of the accuracy scores is also created, showing a very rough Gaussian shape, perhaps with a longer right tail. A large sample and a different number of bins on the plot might better expose the true underlying shape of the distribution.

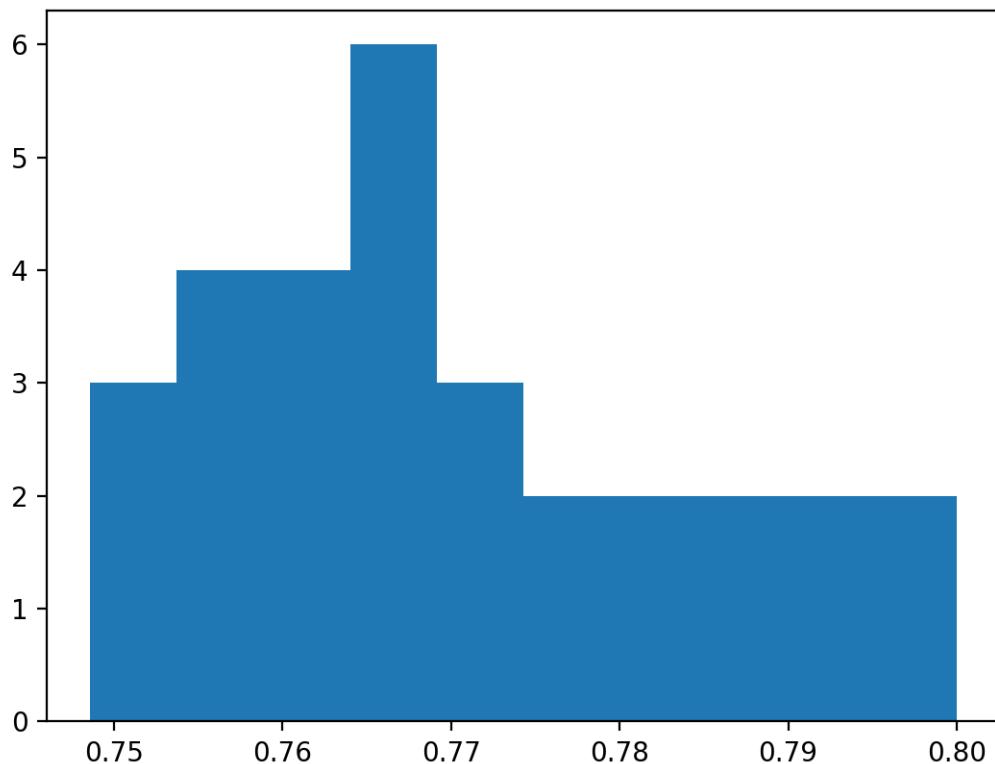


Figure 20.3: Histogram of Model Test Accuracy Over 30 Repeats.

A box and whisker plot is also created showing a line at the median at about 76.5% accuracy on the test set and the interquartile range or middle 50% of the samples between about 78% and 76%.

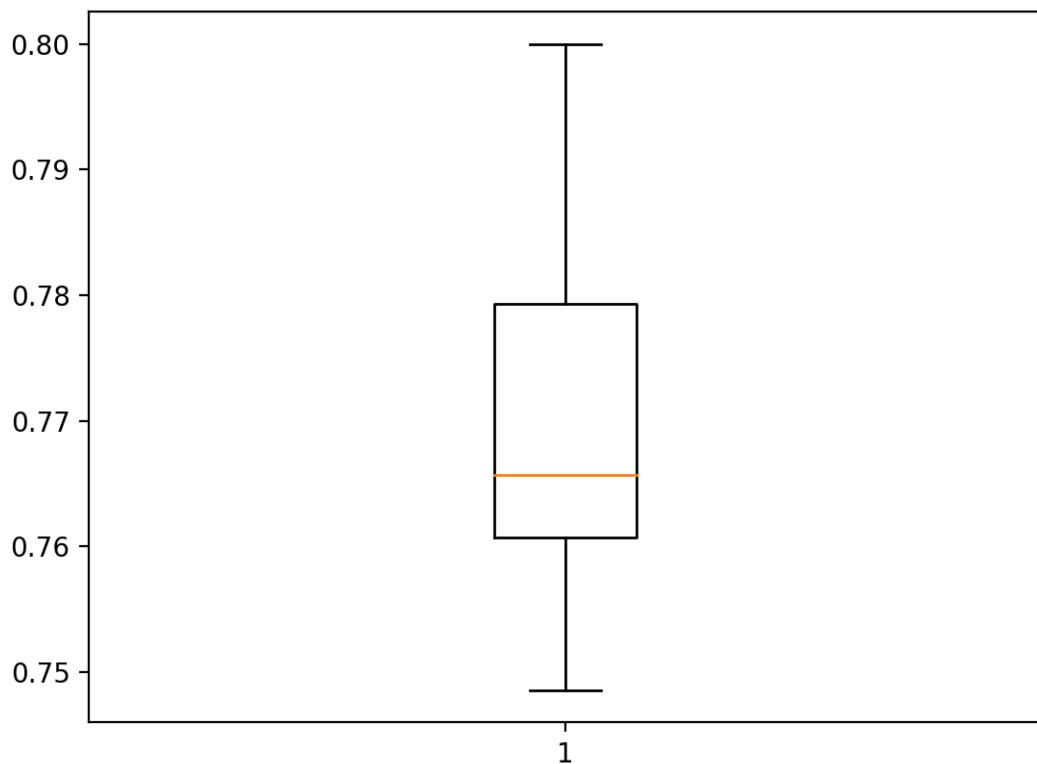


Figure 20.4: Box and Whisker Plot of Model Test Accuracy Over 30 Repeats.

The analysis of the sample of test scores clearly demonstrates a variance in the performance of the same model trained on the same dataset. A spread of likely scores of about 8 percentage points (81% to 73%) on the test set could reasonably be considered large, e.g. a high variance result.

20.3.4 Model Averaging Ensemble

We can use model averaging to both reduce the variance of the model and possibly reduce the generalization error of the model. Specifically, this would result in a smaller standard deviation on the holdout test set and a better performance on the training set. We can check both of these assumptions. First, we must develop a function to prepare and return a fit model on the training dataset.

```
# fit model on dataset
def fit_model(trainX, trainy):
    # define model
    model = Sequential()
    model.add(Dense(15, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=200, verbose=0)
```

```
    return model
```

Listing 20.23: Example of a function to return a fit MLP model.

Next, we need a function that can take a list of ensemble members and make a prediction for an out-of-sample dataset. This could be one or more samples arranged in a two-dimensional array of samples and input features. Tip: you can use this function yourself for testing ensembles and for making predictions with ensembles on new data.

```
# make an ensemble prediction for multiclass classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result
```

Listing 20.24: Example of a function for making ensemble predictions.

We don't know how many ensemble members will be appropriate for this problem. Therefore, we can perform a sensitivity analysis of the number of ensemble members and how it impacts test accuracy. This means we need a function that can evaluate a specified number of ensemble members and return the accuracy of a prediction combined from those members.

```
# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    print(len(subset))
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)
```

Listing 20.25: Example of a function for evaluating predictions from a given number of ensemble members.

Finally, we can create a line plot of the number of ensemble members (x-axis) versus the accuracy of a prediction averaged across that many members on the test dataset (y-axis).

```
# plot score vs number of ensemble members
x_axis = [i for i in range(1, n_members+1)]
pyplot.plot(x_axis, scores)
pyplot.show()
```

Listing 20.26: Example of a line plot of the number of ensemble members vs predictive performance.

The complete example is listed below.

```
# model averaging ensemble and a study of ensemble size on test accuracy
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
```

```
from keras.layers import Dense
from sklearn.metrics import accuracy_score
from matplotlib import pyplot
from numpy import array
from numpy import argmax
import numpy

# fit model on dataset
def fit_model(trainX, trainy):
    # define model
    model = Sequential()
    model.add(Dense(15, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=200, verbose=0)
    return model

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    print(len(subset))
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = int(0.3 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
trainy = to_categorical(trainy)
# fit all models
n_members = 20
members = [fit_model(trainX, trainy) for _ in range(n_members)]
# evaluate different numbers of ensembles
scores = list()
for i in range(1, n_members+1):
    score = evaluate_n_members(members, i, testX, testy)
    print('> %.3f' % score)
    scores.append(score)
# plot score vs number of ensemble members
```

```
x_axis = [i for i in range(1, n_members+1)]
pyplot.plot(x_axis, scores)
pyplot.show()
```

Listing 20.27: Example of a study of the number of ensemble members vs predictive performance on the blobs problem.

Running the example first fits 20 models on the same training dataset, which may take less than a minute on modern hardware. Then, different sized ensembles are tested from 1 member to all 20 members and test accuracy results are printed for each ensemble size.

```
...
16
> 0.760
17
> 0.763
18
> 0.766
19
> 0.763
20
> 0.763
```

Listing 20.28: Example output from a study of the number of ensemble members vs predictive performance on the blobs problem.

Finally, a line plot is created showing the relationship between ensemble size and performance on the test set. We can see that performance improves to about five members, after which performance plateaus around 76% accuracy. This is close to the average test set performance observed during the analysis of the repeated evaluation of the model.

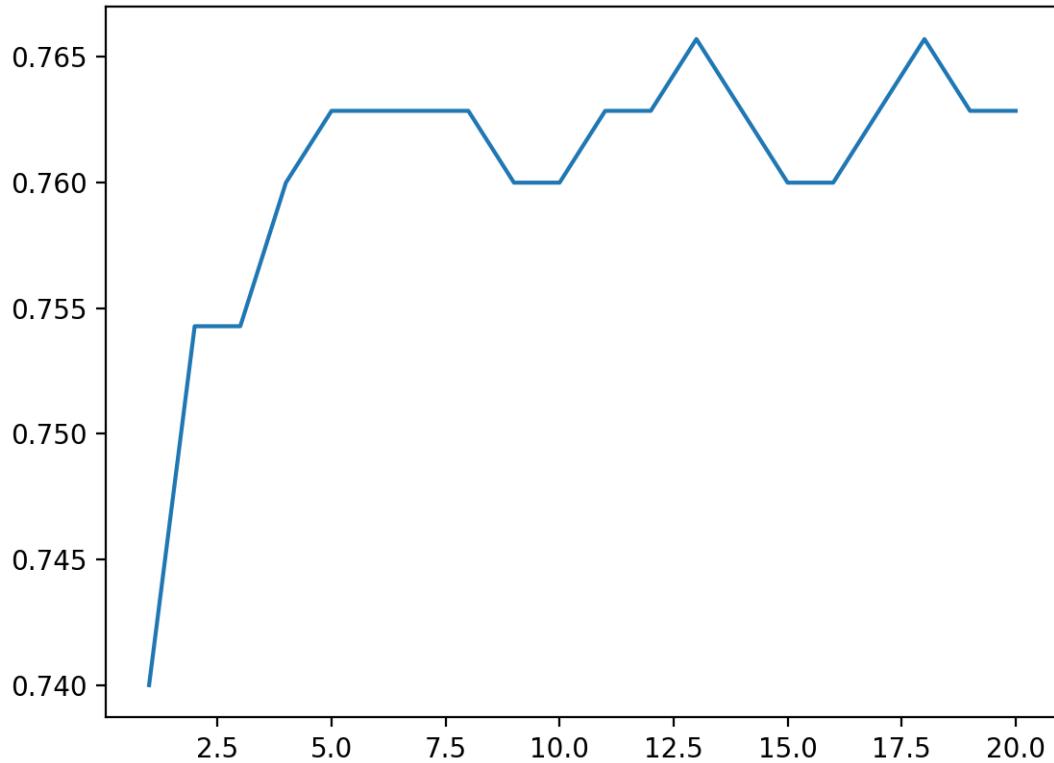


Figure 20.5: Line Plot of Ensemble Size Versus Model Test Accuracy.

Finally, we can update the repeated evaluation experiment to use an ensemble of five models instead of a single model and compare the distribution of scores. The complete example of a repeated evaluated five-member ensemble of the blobs dataset is listed below.

```
# repeated evaluation of model averaging ensemble on blobs dataset
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from numpy import array
from numpy import argmax
from numpy import mean
from numpy import std
import numpy

# fit model on dataset
def fit_model(trainX, trainy):
    # define model
    model = Sequential()
    model.add(Dense(15, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, to_categorical(trainy), epochs=100, verbose=0)
    return model
```

```

model.fit(trainX, trainy, epochs=200, verbose=0)
return model

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

# evaluate ensemble model
def evaluate_members(members, testX, testy):
    # make prediction
    yhat = ensemble_predictions(members, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = int(0.3 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
trainy = to_categorical(trainy)
# repeated evaluation
n_repeats = 30
n_members = 5
scores = list()
for _ in range(n_repeats):
    # fit all models
    members = [fit_model(trainX, trainy) for _ in range(n_members)]
    # evaluate ensemble
    score = evaluate_members(members, testX, testy)
    print('> %.3f' % score)
    scores.append(score)
# summarize the distribution of scores
print('Scores Mean: %.3f, Standard Deviation: %.3f' % (mean(scores), std(scores)))

```

Listing 20.29: Example of a repeated evaluation of a model average ensemble on the blobs problem.

Running the example may take a few minutes as five models are fit and evaluated and this process is repeated 30 times. The performance of each model on the test set is printed to provide an indication of progress. The mean and standard deviation of the model performance is printed at the end of the run.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```

...
> 0.766

```

```
> 0.763
> 0.766
> 0.771
> 0.769
Scores Mean: 0.768, Standard Deviation: 0.006
```

Listing 20.30: Example output from a repeated evaluation of a model average ensemble on the blobs problem.

In this case, we can see that the average performance of a five-member ensemble on the dataset is 76.8%. This is very close to the average of 77% seen for a single model. The important difference is the standard deviation shrinking from 1.4% for a single model to 0.6% with an ensemble of five models. We might expect that a given ensemble of five models on this problem to have a performance fall between about 74% and about 78% with a likelihood of 99%.

Averaging the same model trained on the same dataset gives us a spread for improved reliability, a property often highly desired in a final model to be used operationally. More models in the ensemble will further decrease the standard deviation of the accuracy of an ensemble on the test dataset given the law of large numbers, at least to a point of diminishing returns. This demonstrates that for this specific model and prediction problem, that a model averaging ensemble with five members is sufficient to reduce the variance of the model. This reduction in variance, in turn, also means a better on-average performance when preparing a final model.

20.4 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Average Class Prediction.** Update the example to average the class integer prediction instead of the class probability prediction and compare results.
- **Save and Load Models.** Update the example to save ensemble members to file, then load them from a separate script for evaluation.
- **Sensitivity of Variance.** Create a new example that performs a sensitivity analysis of the number of ensemble members on the standard deviation of model performance on the test set over a given number of repeats and report the point of diminishing returns.

If you explore any of these extensions, I'd love to know.

20.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

20.5.1 APIs

- Getting started with the Keras Sequential model.
<https://keras.io/getting-started/sequential-model-guide/>
- Keras Core Layers API.
<https://keras.io/layers/core/>

- `scipy.stats.mode` API.
<https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mode.html>
- `numpy.argmax` API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html>
- `sklearn.datasets.make_blobs` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

20.6 Summary

In this tutorial, you discovered how to develop a model averaging ensemble in Keras to reduce the variance in a final model. Specifically, you learned:

- Model averaging is an ensemble learning technique that can be used to reduce the expected variance of deep learning neural network models.
- How to implement model averaging in Keras for classification and regression predictive modeling problems.
- How to work through a multiclass classification problem and use model averaging to reduce the variance of the final model.

20.6.1 Next

In the next tutorial, you will discover how to weigh the contributions of ensemble members to a prediction in proportion to the trust in each model.

Chapter 21

Contribute Proportional to Trust with Weighted Average Ensemble

A modeling averaging ensemble combines the prediction from each model equally and often results in better performance on average than a given single model. Sometimes there are very good models that we wish to contribute more to an ensemble prediction, and perhaps less skillful models that may be useful but should contribute less to an ensemble prediction. A weighted average ensemble is an approach that allows multiple models to contribute to a prediction in proportion to their trust or estimated performance. In this tutorial, you will discover how to develop a weighted average ensemble of deep learning neural network models in Python with Keras. After completing this tutorial, you will know:

- Model averaging ensembles are limited because they require that each ensemble member contribute equally to predictions.
- Weighted average ensembles allow the contribution of each ensemble member to a prediction to be weighted proportionally to the trust or performance of the member on a holdout dataset.
- How to implement a weighted average ensemble in Keras and compare results to a model averaging ensemble and standalone models.

Let's get started.

21.1 Weighted Average Ensemble

Model averaging is an approach to ensemble learning where each ensemble member contributes an equal amount to the final prediction. In the case of regression, the ensemble prediction is calculated as the average of the member predictions. In the case of predicting a class label, the prediction is calculated as the mode of the member predictions. In the case of predicting a class probability, the prediction can be calculated as the argmax of the summed probabilities for each class label. A limitation of this approach is that each model has an equal contribution to the final prediction made by the ensemble. There is a requirement that all ensemble members have skill as compared to random chance, although some models are known to perform much better or much worse than other models.

A weighted ensemble is an extension of a model averaging ensemble where the contribution of each member to the final prediction is weighted by the performance of the model. The model weights are small positive values and the sum of all weights equals one, allowing the weights to indicate the percentage of trust or expected performance from each model.

One can think of the weight W_k as the belief in predictor k and we therefore constrain the weights to be positive and sum to one.

— *Learning with ensembles: How over-fitting can be useful*, 1996.

Uniform values for the weights (e.g. $\frac{1}{k}$ where k is the number of ensemble members) means that the weighted ensemble acts as a simple averaging ensemble. There is no analytical solution to finding the weights (we cannot calculate them); instead, the value for the weights can be estimated using either the training dataset or a holdout validation dataset. Finding the weights using the same training set used to fit the ensemble members will likely result in an overfit model. A more robust approach is to use a holdout validation dataset unseen by the ensemble members during training.

The simplest, perhaps most exhaustive approach would be to grid search weight values between 0 and 1 for each ensemble member. Alternately, an optimization procedure such as a linear solver or gradient descent optimization can be used to estimate the weights using a unit norm weight constraint to ensure that the vector of weights sum to one. Unless the holdout validation dataset is large and representative, a weighted ensemble has an opportunity to overfit as compared to a simple averaging ensemble. A simple alternative to adding more weight to a given model without calculating explicit weight coefficients is to add a given model more than once to the ensemble. Although less flexible, it allows a given well-performing model to contribute more than once to a given prediction made by the ensemble.

21.2 Weighted Average Ensemble Case Study

In this section, we will demonstrate how to use the weighted average ensemble to reduce the variance of an MLP on a simple multiclass classification problem. This example provides a template for applying the weighted average ensemble to your own neural network for classification and regression problems.

21.2.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate the weighted averaging ensemble. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. The problem can be configured to have two input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same data points.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

Listing 21.1: Example of creating samples for the blobs problem.

The results are the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can plot each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot for each class value
for class_value in range(3):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 21.2: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (separable by a line) causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions resulting in a high variance.

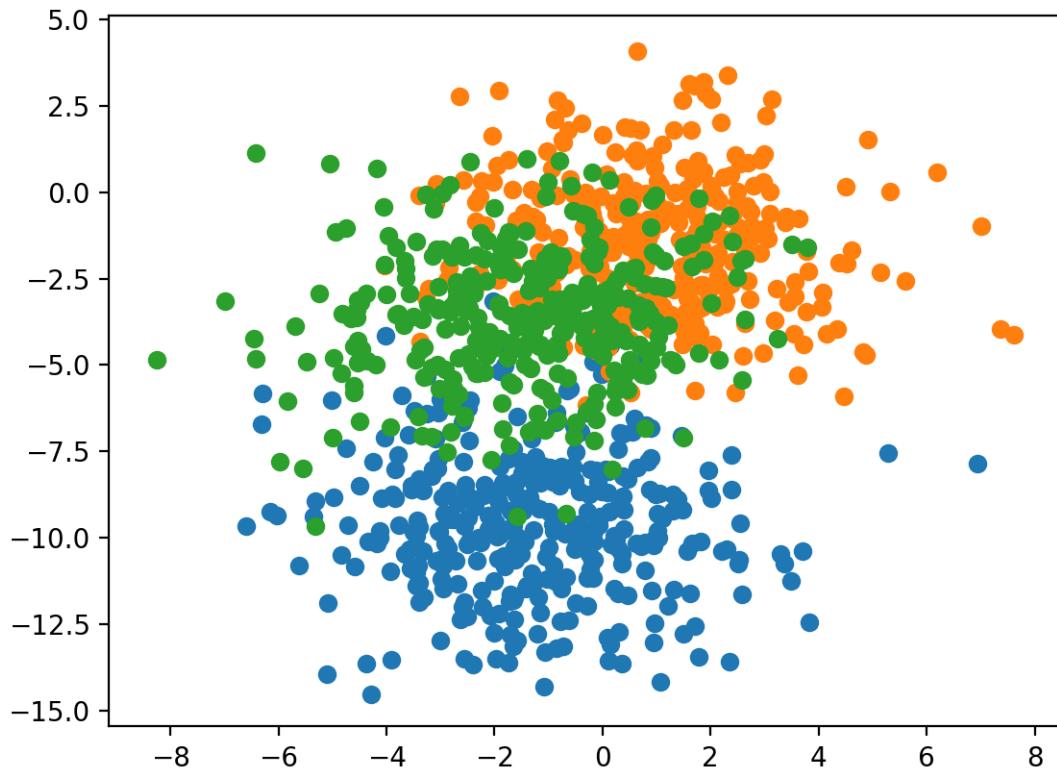


Figure 21.1: Scatter Plot of Blobs Dataset With Three Classes and Points Colored by Class Value.

21.2.2 Multilayer Perceptron Model

Before we define a model, we need to contrive a problem that is appropriate for the weighted average ensemble. In our problem, the training dataset is relatively small. Specifically, there is a 10:1 ratio of examples in the training dataset to the holdout dataset. This mimics a situation where we may have a vast number of unlabeled examples and a small number of labeled examples with which to train a model. We will create 1,100 data points from the blobs problem. The model will be trained on the first 100 points and the remaining 1,000 will be held back in a test dataset, unavailable to the model.

The problem is a multiclass classification problem, and we will model it using a softmax activation function on the output layer. This means that the model will predict a vector with three elements with the probability that the sample belongs to each of the three classes. Therefore, we must one hot encode the class values before we split the rows into the train and test datasets. We can do this using the Keras `to_categorical()` function.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
```

```
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
print(trainX.shape, testX.shape)
```

Listing 21.3: Example of preparing the datasets for modeling.

Next, we can define and compile the model. The model will expect samples with two input variables. The model then has a single hidden layer with 25 nodes and a rectified linear activation function, then an output layer with three nodes to predict the probability of each of the three classes and a softmax activation function. Because the problem is multiclass, we will use the categorical cross-entropy loss function to optimize the model and the efficient Adam flavor of stochastic gradient descent.

```
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 21.4: Example of defining an MLP model.

The model is fit for 500 training epochs and we will evaluate the model each epoch on the test set, using the test set as a validation set.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=500, verbose=0)
```

Listing 21.5: Example of fitting an MLP model.

At the end of the run, we will evaluate the performance of the model on the train and test sets.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 21.6: Example of evaluating an MLP model.

Then finally, we will plot learning curves of the model accuracy over each training epoch on both the training and validation datasets.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 21.7: Example of plotting learning curves for the MLP model.

Tying all of this together, the complete example is listed below.

```
# develop an mlp for blobs dataset
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=500, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 21.8: Example of fitting an MLP on the blobs problem.

Running the example first prints the shape of each dataset for confirmation, then the performance of the final model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved about 85% accuracy on the training dataset, which we know is optimistic, and about 81% on the test dataset, which we would expect to be more realistic.

Train: 0.850, Test: 0.816

Listing 21.9: Example output fitting an MLP on the blobs problem.

A line plot is also created showing the learning curves for the model loss and accuracy on the train and test sets over each training epoch. We can see that training accuracy is more optimistic over most of the run as we also noted with the final scores.

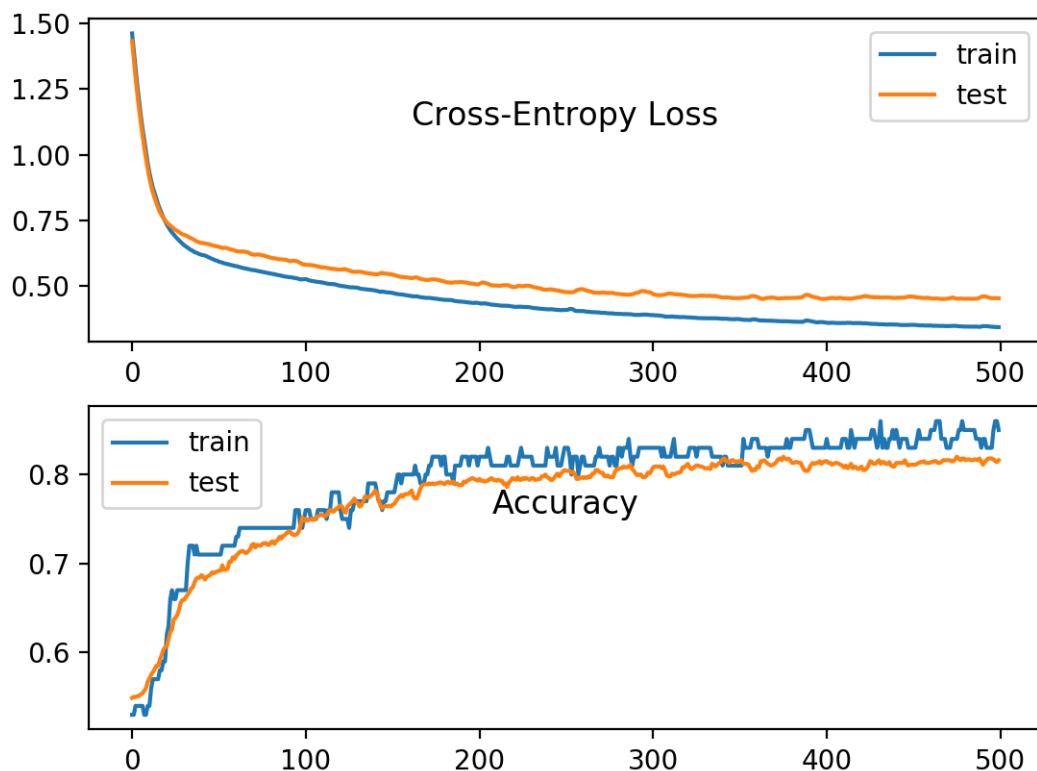


Figure 21.2: Line Plot Learning Curves of Model Accuracy on Train and Test Dataset over Each Training Epoch.

Now that we have identified that the model is a good candidate for developing an ensemble, we can next look at developing a simple model averaging ensemble.

21.2.3 Model Averaging Ensemble

We can develop a simple model averaging ensemble before we look at developing a weighted average ensemble (covered in depth in Chapter 20). The results of the model averaging ensemble can be used as a point of comparison as we would expect a well configured weighted average ensemble to perform better. First, we need to fit multiple models from which to develop an ensemble. We will define a function named `fit_model()` to create and fit a single model on the training dataset that we can call repeatedly to create as many models as we wish.

```
# fit model on dataset
def fit_model(trainX, trainy):
    trainy_enc = to_categorical(trainy)
    # define model
```

```

model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
model.fit(trainX, trainy_enc, epochs=500, verbose=0)
return model

```

Listing 21.10: Example of function for fitting and returning an MLP model.

We can call this function to create a pool of 10 models.

```

# fit all models
n_members = 10
members = [fit_model(trainX, trainy) for _ in range(n_members)]

```

Listing 21.11: Example of creating a list of fit models.

Next, we can develop model averaging ensemble. We don't know how many members would be appropriate for this problem, so we can create ensembles with different sizes from one to 10 members and evaluate the performance of each on the test set. We can also evaluate the performance of each standalone model in the performance on the test set. This provides a useful point of comparison for the model averaging ensemble, as we expect that the ensemble will out-perform a randomly selected single model on average.

Each model predicts the probabilities for each class label, e.g. has three outputs. A single prediction can be converted to a class label by using the `argmax()` function on the predicted probabilities, e.g. return the index in the prediction with the largest probability value. We can ensemble the predictions from multiple models by summing the probabilities for each class prediction and using the `argmax()` on the result. The `ensemble_predictions()` function below implements this behavior.

```

# make an ensemble prediction for multiclass classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

```

Listing 21.12: Example of a function for making ensemble predictions.

We can estimate the performance of an ensemble of a given size by selecting the required number of models from the list of all models, calling the `ensemble_predictions()` function to make a prediction, then calculating the accuracy of the prediction by comparing it to the true values. The `evaluate_n_members()` function below implements this behavior.

```

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # make prediction
    yhat = ensemble_predictions(subset, testX)

```

```
# calculate accuracy
return accuracy_score(testy, yhat)
```

Listing 21.13: Example of a function for evaluating an ensemble with a given number of members.

The scores of the ensembles of each size can be stored to be plotted later, and the scores for each individual model are collected and the average performance reported.

```
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, len(members)+1):
    # evaluate model with i members
    ensemble_score = evaluate_n_members(members, i, testX, testy)
    # evaluate the i'th model standalone
    testy_enc = to_categorical(testy)
    _, single_score = members[i-1].evaluate(testX, testy_enc, verbose=0)
    # summarize this step
    print('> %d: single=%f, ensemble=%f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# summarize average accuracy of a single final model
print('Accuracy %f (%f)' % (mean(single_scores), std(single_scores)))
```

Listing 21.14: Example of a evaluating the performance of different sized ensembles.

Finally, we create a graph that shows the accuracy of each individual model (blue dots) and the performance of the model averaging ensemble as the number of members is increased from one to 10 members (orange line). Tying all of this together, the complete example is listed below.

```
# model averaging ensemble for the blobs dataset
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
from numpy import mean
from numpy import std
import numpy
from numpy import array
from numpy import argmax

# fit model on dataset
def fit_model(trainX, trainy):
    trainy_enc = to_categorical(trainy)
    # define model
    model = Sequential()
    model.add(Dense(25, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy_enc, epochs=500, verbose=0)
    return model

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
```

```

# make predictions
yhats = [model.predict(testX) for model in members]
yhats = array(yhats)
# sum across ensemble members
summed = numpy.sum(yhats, axis=0)
# argmax across classes
result = argmax(summed, axis=1)
return result

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# fit all models
n_members = 10
members = [fit_model(trainX, trainy) for _ in range(n_members)]
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, len(members)+1):
    # evaluate model with i members
    ensemble_score = evaluate_n_members(members, i, testX, testy)
    # evaluate the i'th model standalone
    testy_enc = to_categorical(testy)
    _, single_score = members[i-1].evaluate(testX, testy_enc, verbose=0)
    # summarize this step
    print('> %d: single=%3f, ensemble=%3f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# summarize average accuracy of a single final model
print('Accuracy %.3f (%.3f)' % (mean(single_scores), std(single_scores)))
# plot score vs number of ensemble members
x_axis = [i for i in range(1, len(members)+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()

```

Listing 21.15: Example of a modeling averaging ensemble on the blobs problem.

Running the example first reports the performance of each single model as well as the model averaging ensemble of a given size with 1, 2, 3, etc. members.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

On this run, the average performance of the single models is reported at about 80.4% and

we can see that an ensemble with between five and nine members will achieve a performance between 80.8% and 81%. As expected, the performance of a modest-sized model averaging ensemble out-performs the performance of a randomly selected single model on average.

```
> 1: single=0.803, ensemble=0.803
> 2: single=0.805, ensemble=0.808
> 3: single=0.798, ensemble=0.805
> 4: single=0.809, ensemble=0.809
> 5: single=0.808, ensemble=0.811
> 6: single=0.805, ensemble=0.808
> 7: single=0.805, ensemble=0.808
> 8: single=0.804, ensemble=0.809
> 9: single=0.810, ensemble=0.810
> 10: single=0.794, ensemble=0.808
Accuracy 0.804 (0.005)
```

Listing 21.16: Example output from modeling averaging ensemble on the blobs problem.

Next, a graph is created comparing the accuracy of single models (blue dots) to the model averaging ensemble of increasing size (orange line). On this run, the orange line of the ensembles clearly shows better or comparable performance (if dots are hidden) than the single models.

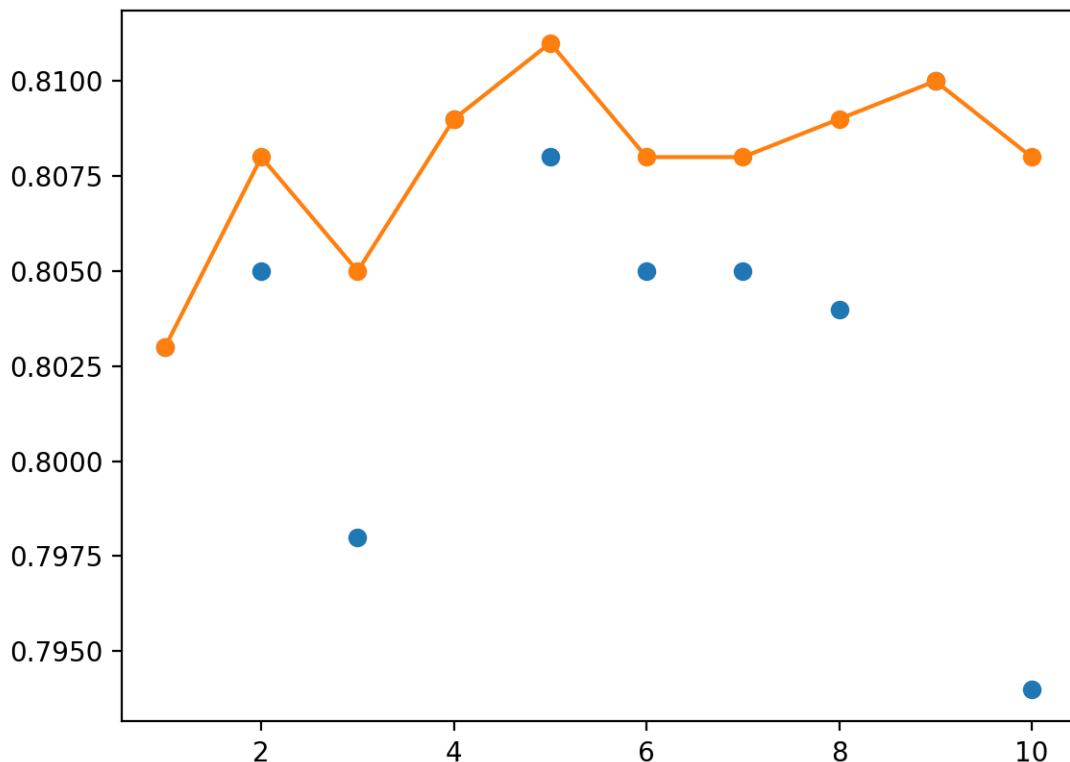


Figure 21.3: Line Plot Showing Single Model Accuracy (blue dots) and Accuracy of Ensembles of Increasing Size (orange line).

Now that we know how to develop a model averaging ensemble, we can extend the approach one step further by weighting the contributions of the ensemble members.

21.2.4 Grid Search Weighted Average Ensemble

The model averaging ensemble allows each ensemble member to contribute an equal amount to the prediction of the ensemble. We can update the example so that instead, the contribution of each ensemble member is weighted by a coefficient that indicates the trust or expected performance of the model. Weight values are small values between 0 and 1 and are treated like a percentage, such that the weights across all ensemble members sum to one. First, we must update the `ensemble_predictions()` function to make use of a vector of weights for each ensemble member. Instead of simply summing the predictions across each ensemble member, we must calculate a weighted sum. We can implement this manually using for loops, but this is terribly inefficient; for example:

```
# calculated a weighted sum of predictions
def weighted_sum(weights, yhats):
    rows = list()
    for j in range(yhats.shape[1]):
        # enumerate values
        row = list()
        for k in range(yhats.shape[2]):
            # enumerate members
            value = 0.0
            for i in range(yhats.shape[0]):
                value += weights[i] * yhats[i,j,k]
            row.append(value)
        rows.append(row)
    return array(rows)
```

Listing 21.17: Example of manually calculating a weighted average for predictions.

Instead, we can use efficient NumPy functions to implement the weighted sum such as `einsum()` or `tensordot()`. Full discussion of these functions is a little out of scope so please refer to the API documentation for more information on how to use these functions as they are challenging if you are new to linear algebra and/or NumPy. We will use `tensordot()` function to apply the tensor product with the required summing; the updated `ensemble_predictions()` function is listed below.

```
# make an ensemble prediction for multiclass classification
def ensemble_predictions(members, weights, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # weighted sum across ensemble members
    summed = tensordot(yhats, weights, axes=((0),(0)))
    # argmax across classes
    result = argmax(summed, axis=1)
    return result
```

Listing 21.18: Example of making weighted ensemble predictions.

Next, we must update `evaluate_ensemble()` to pass along the weights when making the prediction for the ensemble.

```
# evaluate a specific number of members in an ensemble
def evaluate_ensemble(members, weights, testX, testy):
    # make prediction
    yhat = ensemble_predictions(members, weights, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)
```

Listing 21.19: Example of evaluating an ensemble of a given size.

We will use a modest-sized ensemble of five members, that appeared to perform well in the model averaging ensemble.

```
# fit all models
n_members = 5
members = [fit_model(trainX, trainy) for _ in range(n_members)]
```

Listing 21.20: Example of creating a list of ensemble members.

We can then estimate the performance of each individual model on the test dataset as a reference.

```
# evaluate averaging ensemble (equal weights)
weights = [1.0/n_members for _ in range(n_members)]
score = evaluate_ensemble(members, weights, testX, testy)
print('Equal Weights Score: %.3f' % score)
```

Listing 21.21: Example of evaluating an equally weighted ensemble.

Finally, we can develop a weighted average ensemble. A simple, but exhaustive approach to finding weights for the ensemble members is to grid search values. We can define a coarse grid of weight values from 0.0 to 1.0 in steps of 0.1, then generate all possible five-element vectors with those values. Generating all possible combinations is called a Cartesian product, which can be implemented in Python using the `itertools.product()` function from the standard library.

A limitation of this approach is that the vectors of weights will not sum to one (called the unit norm), as required. We can force each generated weight vector to have a unit norm by calculating the sum of the absolute weight values (called the L1 norm) and dividing each weight by that value. The `normalize()` function below implements this hack.

```
# normalize a vector to have unit norm
def normalize(weights):
    # calculate l1 vector norm
    result = norm(weights, 1)
    # check for a vector of all zeros
    if result == 0.0:
        return weights
    # return normalized vector (unit norm)
    return weights / result
```

Listing 21.22: Example of a function for normalizing ensemble member weights.

We can now enumerate each weight vector generated by the Cartesian product, normalize it, and evaluate it by making a prediction and keeping the best to be used in our final weight averaging ensemble.

```
# grid search weights
def grid_search(members, testX, testy):
```

```

# define weights to consider
w = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
best_score, best_weights = 0.0, None
# iterate all possible combinations (cartesian product)
for weights in product(w, repeat=len(members)):
    # skip if all weights are equal
    if len(set(weights)) == 1:
        continue
    # hack, normalize weight vector
    weights = normalize(weights)
    # evaluate weights
    score = evaluate_ensemble(members, weights, testX, testy)
    if score > best_score:
        best_score, best_weights = score, weights
    print('>%s %.3f' % (best_weights, best_score))
return list(best_weights)

```

Listing 21.23: Example of a function for grid searching weights for an ensemble.

Once discovered, we can report the performance of our weight average ensemble on the test dataset, which we would expect to be better than the best single model and ideally better than the model averaging ensemble.

```

# grid search weights
weights = grid_search(members, testX, testy)
score = evaluate_ensemble(members, weights, testX, testy)
print('Grid Search Weights: %s, Score: %.3f' % (weights, score))

```

Listing 21.24: Example of grid searching weights for a weighted ensemble.

The complete example is listed below.

```

# grid search for coefficients in a weighted average ensemble for the blobs problem
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from numpy import array
from numpy import argmax
from numpy import tensordot
from numpy.linalg import norm
from itertools import product

# fit model on dataset
def fit_model(trainX, trainy):
    trainy_enc = to_categorical(trainy)
    # define model
    model = Sequential()
    model.add(Dense(25, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy_enc, epochs=500, verbose=0)
    return model

# make an ensemble prediction for multi-class classification

```

```
def ensemble_predictions(members, weights, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # weighted sum across ensemble members
    summed = tensordot(yhats, weights, axes=((0),(0)))
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

# evaluate a specific number of members in an ensemble
def evaluate_ensemble(members, weights, testX, testy):
    # make prediction
    yhat = ensemble_predictions(members, weights, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# normalize a vector to have unit norm
def normalize(weights):
    # calculate l1 vector norm
    result = norm(weights, 1)
    # check for a vector of all zeros
    if result == 0.0:
        return weights
    # return normalized vector (unit norm)
    return weights / result

# grid search weights
def grid_search(members, testX, testy):
    # define weights to consider
    w = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
    best_score, best_weights = 0.0, None
    # iterate all possible combinations (cartesian product)
    for weights in product(w, repeat=len(members)):
        # skip if all weights are equal
        if len(set(weights)) == 1:
            continue
        # hack, normalize weight vector
        weights = normalize(weights)
        # evaluate weights
        score = evaluate_ensemble(members, weights, testX, testy)
        if score > best_score:
            best_score, best_weights = score, weights
            print('>%s %.3f' % (best_weights, best_score))
    return list(best_weights)

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# fit all models
n_members = 5
members = [fit_model(trainX, trainy) for _ in range(n_members)]
# evaluate each single model on the test set
```

```

testy_enc = to_categorical(testy)
for i in range(n_members):
    _, test_acc = members[i].evaluate(testX, testy_enc, verbose=0)
    print('Model %d: %.3f' % (i+1, test_acc))
# evaluate averaging ensemble (equal weights)
weights = [1.0/n_members for _ in range(n_members)]
score = evaluate_ensemble(members, weights, testX, testy)
print('Equal Weights Score: %.3f' % score)
# grid search weights
weights = grid_search(members, testX, testy)
score = evaluate_ensemble(members, weights, testX, testy)
print('Grid Search Weights: %s, Score: %.3f' % (weights, score))

```

Listing 21.25: Example of a grid searching a weighted average ensemble on the blobs problem.

Running the example first creates the five single models and evaluates their performance on the test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

On this run, we can see that model 2 has the best solo performance of about 81.7% accuracy. Next, a model averaging ensemble is created with a performance of about 80.7%, which is reasonable compared to most of the models, but not all.

```

Model 1: 0.798
Model 2: 0.817
Model 3: 0.798
Model 4: 0.806
Model 5: 0.810
Equal Weights Score: 0.807

```

Listing 21.26: Example output of model performance for ensemble members and equally weighted ensemble.

Next, the grid search is performed. It is pretty slow and may take about twenty minutes on modern hardware. The process could easily be made parallel using libraries such as Joblib. Each time a new top performing set of weights is discovered, it is reported along with its performance on the test dataset. We can see that during the run, the process discovered that using model 2 alone resulted in a good performance, until it was replaced with something better. We can see that the best performance was achieved on this run using the weights that focus only on the first and second models with the accuracy of 81.8% on the test dataset. This out-performs both the single models and the model averaging ensemble on the same dataset.

```

>[0. 0. 0. 0. 1.] 0.810
>[0. 0. 0. 0.5 0.5] 0.814
>[0.          0.          0.33333333 0.66666667] 0.815
>[0. 1. 0. 0. 0.] 0.817
>[0.23076923 0.76923077 0.          0.          ] 0.818
Grid Search Weights: [0.23076923076923075, 0.7692307692307692, 0.0, 0.0, 0.0], Score: 0.818

```

Listing 21.27: Example output for model weights and the performance of the weighted average ensemble.

An alternate approach to finding weights would be a random search, which has been shown to be effective more generally for model hyperparameter tuning.

21.2.5 Weighted Average MLP Ensemble

An alternative to searching for weight values is to use a directed optimization process. Optimization is a search process, but instead of sampling the space of possible solutions randomly or exhaustively, the search process uses any available information to make the next step in the search, such as toward a set of weights that has lower error. The SciPy library offers many excellent optimization algorithms, including local and global search methods.

SciPy provides an implementation of the Differential Evolution method. This is one of the few stochastic global search algorithms that *just works* for function optimization with continuous inputs, and it works well. The `differential_evolution()` SciPy function requires that function is specified to evaluate a set of weights and return a score to be minimized. We can minimize the classification error (1 - accuracy). As with the grid search, we must normalize the weight vector before we evaluate it. The `loss_function()` function below will be used as the evaluation function during the optimization process.

```
# loss function for optimization process, designed to be minimized
def loss_function(weights, members, testX, testy):
    # normalize weights
    normalized = normalize(weights)
    # calculate error rate
    return 1.0 - evaluate_ensemble(members, normalized, testX, testy)
```

Listing 21.28: Example of loss function for global search algorithm.

We must also specify the bounds of the optimization process. We can define the bounds as a five-dimensional hypercube (e.g. 5 weights for the 5 ensemble members) with values between 0.0 and 1.0.

```
# define bounds on each weight
bound_w = [(0.0, 1.0) for _ in range(n_members)]
```

Listing 21.29: Example of parameter bounds for global search algorithm.

Our loss function requires three parameters in addition to the weights, which we will provide as a tuple to then be passed along to the call to the `loss_function()` each time a set of weights is evaluated.

```
# arguments to the loss function
search_arg = (members, testX, testy)
```

Listing 21.30: Example of arguments for evaluating the performance of a model under the global search algorithm.

We can now call our optimization process. We will limit the total number of iterations of the algorithms to 1,000, and use a smaller than default tolerance to detect if the search process has converged.

```
# global optimization of ensemble weights
result = differential_evolution(loss_function, bound_w, search_arg, maxiter=1000, tol=1e-7)
```

Listing 21.31: Example of executing the global search algorithm.

The result of the call to `differential_evolution()` is a dictionary that contains all kinds of information about the search. Importantly, the `x` key contains the optimal set of weights found during the search. We can retrieve the best set of weights, then report them and their performance on the test set when used in a weighted ensemble.

```
# get the chosen weights
weights = normalize(result['x'])
print('Optimized Weights: %s' % weights)
# evaluate chosen weights
score = evaluate_ensemble(members, weights, testX, testy)
print('Optimized Weights Score: %.3f' % score)
```

Listing 21.32: Example of evaluating the weights found via global optimization.

Tying all of this together, the complete example is listed below.

```
# global optimization to find coefficients for weighted ensemble on blobs problem
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from numpy import array
from numpy import argmax
from numpy import tensordot
from numpy.linalg import norm
from scipy.optimize import differential_evolution

# fit model on dataset
def fit_model(trainX, trainy):
    trainy_enc = to_categorical(trainy)
    # define model
    model = Sequential()
    model.add(Dense(25, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy_enc, epochs=500, verbose=0)
    return model

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, weights, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # weighted sum across ensemble members
    summed = tensordot(yhats, weights, axes=((0),(0)))
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

# # evaluate a specific number of members in an ensemble
def evaluate_ensemble(members, weights, testX, testy):
    # make prediction
    yhat = ensemble_predictions(members, weights, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# normalize a vector to have unit norm
def normalize(weights):
    # calculate l1 vector norm
```

```

result = norm(weights, 1)
# check for a vector of all zeros
if result == 0.0:
    return weights
# return normalized vector (unit norm)
return weights / result

# loss function for optimization process, designed to be minimized
def loss_function(weights, members, testX, testy):
    # normalize weights
    normalized = normalize(weights)
    # calculate error rate
    return 1.0 - evaluate_ensemble(members, normalized, testX, testy)

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# fit all models
n_members = 5
members = [fit_model(trainX, trainy) for _ in range(n_members)]
# evaluate each single model on the test set
testy_enc = to_categorical(testy)
for i in range(n_members):
    _, test_acc = members[i].evaluate(testX, testy_enc, verbose=0)
    print('Model %d: %.3f' % (i+1, test_acc))
# evaluate averaging ensemble (equal weights)
weights = [1.0/n_members for _ in range(n_members)]
score = evaluate_ensemble(members, weights, testX, testy)
print('Equal Weights Score: %.3f' % score)
# define bounds on each weight
bound_w = [(0.0, 1.0) for _ in range(n_members)]
# arguments to the loss function
search_arg = (members, testX, testy)
# global optimization of ensemble weights
result = differential_evolution(loss_function, bound_w, search_arg, maxiter=1000, tol=1e-7)
# get the chosen weights
weights = normalize(result['x'])
print('Optimized Weights: %s' % weights)
# evaluate chosen weights
score = evaluate_ensemble(members, weights, testX, testy)
print('Optimized Weights Score: %.3f' % score)

```

Listing 21.33: Example of using global search to find weights for weighted average ensemble on the blobs problem.

Running the example first creates five single models and evaluates the performance of each on the test dataset.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see on this run that models 3 and 4 both perform best with an accuracy of about

82.2%. Next, a model averaging ensemble with all five members is evaluated on the test set reporting an accuracy of 81.8%, which is better than some, but not all, single models.

```
Model 1: 0.814
Model 2: 0.811
Model 3: 0.822
Model 4: 0.822
Model 5: 0.809
Equal Weights Score: 0.818
```

Listing 21.34: Example output of model performance for ensemble members and equally weighted ensemble.

The optimization process is relatively quick. We can see that the process found a set of weights that pays most attention to models 3 and 4, and spreads the remaining attention out among the other models, achieving an accuracy of about 82.4%, out-performing the model averaging ensemble and individual models.

```
Optimized Weights: [0.1660322 0.09652591 0.33991854 0.34540932 0.05211403]
Optimized Weights Score: 0.824
```

Listing 21.35: Example output for model weights and the performance of the weighted average ensemble.

It is important to note that in these examples, we have treated the test dataset as though it were a validation dataset. This was done to keep the examples focused and technically simpler. In practice, the choice and tuning of the weights for the ensemble would be chosen by a validation dataset, and single models, model averaging ensembles, and weighted ensembles would be compared on a separate test set.

21.3 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Parallelize Grid Search.** Update the grid search example to use the Joblib library to parallelize weight evaluation.
- **Implement Random Search.** Update the grid search example to use a random search of weight coefficients.
- **Try a Local Search.** Try a local search procedure provided by the SciPy library instead of the global search and compare performance.
- **Repeat Global Optimization.** Repeat the global optimization procedure multiple times for a given set of models to see if differing sets of weights can be found across the runs.

If you explore any of these extensions, I'd love to know.

21.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

21.4.1 Papers

- *When Networks Disagree: Ensemble Methods for Hybrid Neural Networks*, 1995.
https://www.worldscientific.com/doi/pdf/10.1142/9789812795885_0025
- *Neural Network Ensembles, Cross Validation, and Active Learning*, 1995.
<https://papers.nips.cc/paper/1001-neural-network-ensembles-cross-validation-and-active-learning.pdf>
- *Learning with ensembles: How over-fitting can be useful*, 1996.
<http://papers.nips.cc/paper/1044-learning-with-ensembles-how-overfitting-can-be-useful.pdf>

21.4.2 APIs

- Getting started with the Keras Sequential model.
<https://keras.io/getting-started/sequential-model-guide/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- numpy.argmax API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html>
- sklearn.datasets.make_blobs API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
- numpy.einsum API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.einsum.html>
- numpy.tensordot API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.tensordot.html>
- itertools.product API.
<https://docs.python.org/3/library/itertools.html#itertools.product>
- scipy.optimize API.
<https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html>
- scipy.optimize.differential_evolution API.
https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html

21.4.3 Articles

- Ensemble averaging (machine learning), Wikipedia.
[https://en.wikipedia.org/wiki/Ensemble_averaging_\(machine_learning\)](https://en.wikipedia.org/wiki/Ensemble_averaging_(machine_learning))
- Cartesian product, Wikipedia.
https://en.wikipedia.org/wiki/Cartesian_product

- Implementing a Weighted Majority Rule Ensemble Classifier, 2015.
https://sebastianraschka.com/Articles/2014_ensemble_classifier.html

21.5 Summary

In this tutorial, you discovered how to develop a weighted average ensemble of deep learning neural network models in Python with Keras. Specifically, you learned:

- Model averaging ensembles are limited because they require that each ensemble member contribute equally to predictions.
- Weighted average ensembles allow the contribution of each ensemble member to a prediction to be weighted proportionally to the trust or performance of the member on a holdout dataset.
- How to implement a weighted average ensemble in Keras and compare results to a model averaging ensemble and standalone models.

21.5.1 Next

In the next tutorial, you will discover how you can develop ensemble members from the resampling methods used to estimate the performance of a neural network model.

Chapter 22

Fit Models on Different Samples with Resampling Ensembles

Ensemble learning are methods that combine the predictions from multiple models. It is important in ensemble learning that the models that comprise the ensemble are good, making different prediction errors. Predictions that are good in different ways can result in a prediction that is both more stable and often better than the predictions of any individual member model. One way to achieve differences between models is to train each model on a different subset of the available training data. Models are trained on different subsets of the training data naturally through the use of resampling methods such as cross-validation and the bootstrap, designed to estimate the average performance of the model generally on unseen data. The models used in this estimation process can be combined in what is referred to as a resampling-based ensemble, such as a cross-validation ensemble or a bootstrap aggregation (or bagging) ensemble. In this tutorial, you will discover how to develop a suite of different resampling-based ensembles for deep learning neural network models. After completing this tutorial, you will know:

- How to estimate model performance using random-splits and develop an ensemble from the models.
- How to estimate performance using 10-fold cross-validation and develop a cross-validation ensemble.
- How to estimate performance using the bootstrap and combine models using a bagging ensemble.

Let's get started.

22.1 Resampling Ensembles

Combining the predictions from multiple models can result in more stable predictions, and in some cases, predictions that have better performance than any of the contributing models. Effective ensembles require members that disagree. Each member must have skill (e.g. perform better than random chance), but ideally, perform well in different ways. Technically, we can say that we prefer ensemble members to have low correlation in their predictions, or prediction errors.

One approach to encourage differences between ensembles is to use the same learning algorithm on different training datasets. This can be achieved by repeatedly resampling a training dataset that is in turn used to train a new model. Multiple models are fit using slightly different perspectives on the training data and, in turn, make different errors and often more stable and better predictions when combined. We can refer to these methods generally as data resampling ensembles. A benefit of this approach is that resampling methods may be used that do not make use of all examples in the training dataset. Any examples that are not used to fit the model can be used as a test dataset to estimate the generalization error of the chosen model configuration. There are three popular resampling methods that we could use to create a resampling ensemble; they are:

- **Random Splits.** The dataset is repeatedly sampled with a random split of the data into train and test sets.
- **k -fold Cross-Validation.** The dataset is split into k equally sized folds, k models are trained and each fold is given an opportunity to be used as the holdout set where the model is trained on all remaining folds.
- **Bootstrap Aggregation.** Random samples are collected with replacement and examples not included in a given sample are used as the test set.

Perhaps the most widely used resampling ensemble method is bootstrap aggregation, more commonly referred to as bagging. The resampling with replacement allows more difference in the training dataset, biasing the model and, in turn, resulting in more difference between the predictions of the resulting models. Resampling ensemble models makes some specific assumptions about your project:

- That a robust estimate of model performance on unseen data is required; if not, then a single train/test split can be used.
- That there is a potential for a lift in performance using an ensemble of models; if not, then a single model fit on all available data can be used.
- That the computational cost of fitting more than one neural network model on a sample of the training dataset is not prohibitive; if not, all resources should be put into fitting a single model.

Neural network models are remarkably flexible, therefore the lift in performance provided by a resampling ensemble is not always possible given that individual models trained on all available data can perform so well. As such, the sweet spot for using a resampling ensemble is the case where there is a requirement for a robust estimate of performance and multiple models can be fit to calculate the estimate, but there is also a requirement for one (or more) of the models created during the estimate of performance to be used as the final model (e.g. a new final model cannot be fit on all available training data). Now that we are familiar with resampling ensemble methods, we can work through an example of applying each method in turn.

22.2 Resampling Ensembles Case Study

In this section, we will demonstrate how to use the resampling ensemble to reduce the variance of an MLP on a simple multiclass classification problem. This example provides a template for applying the resampling ensemble to your own neural network for classification and regression problems.

22.2.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate a model resampling ensembles. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. We use this problem with 1,000 examples, with input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same 1,000 points.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

Listing 22.1: Example of creating samples for the blobs problem.

The results are the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can plot each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot for each class value
for class_value in range(3):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 22.2: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (separable by a line) causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions resulting in a high variance.

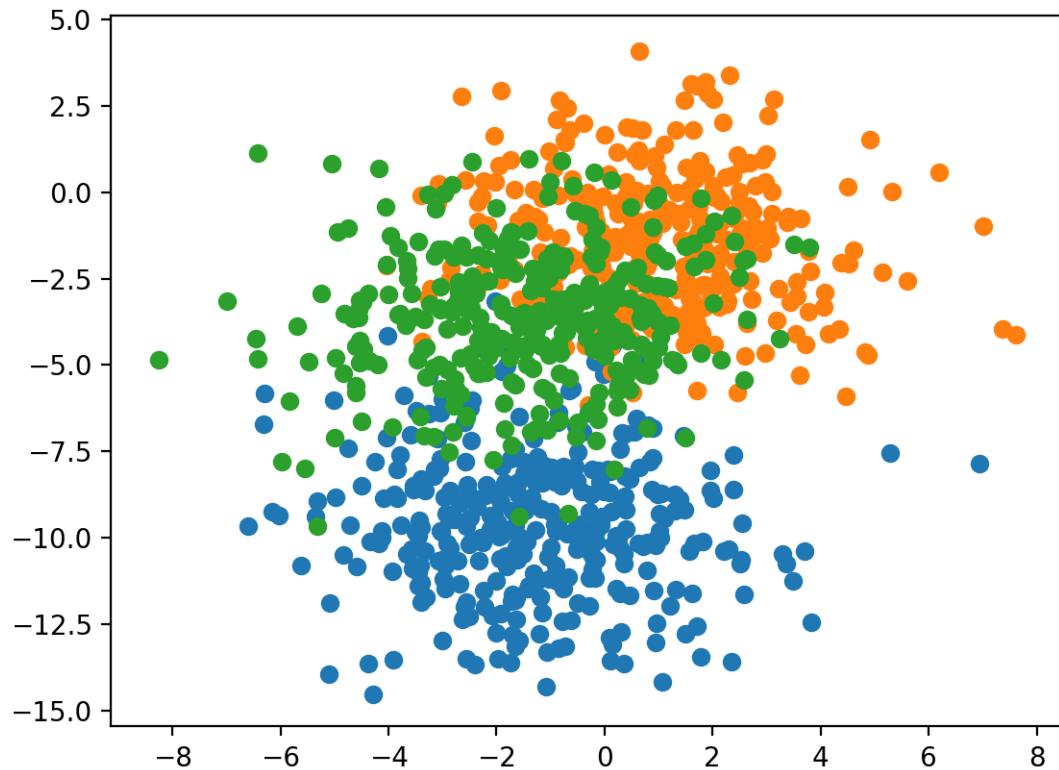


Figure 22.1: Scatter Plot of Blobs Dataset with Three Classes and Points Colored by Class Value.

22.2.2 Single Multilayer Perceptron Model

We will define a Multilayer Perceptron neural network, or MLP, that learns the problem reasonably well. The problem is a multiclass classification problem, and we will model it using a softmax activation function on the output layer. This means that the model will predict a vector with 3 elements with the probability that the sample belongs to each of the 3 classes. Therefore, the first step is to one hot encode the class values.

```
y = to_categorical(y)
```

Listing 22.3: Example of one hot encoding the target variable.

Next, we must split the dataset into training and test sets. We will use the test set both to evaluate the performance of the model and to plot its performance during training with a learning curve. We will use 90% of the data for training and 10% for the test set. We are choosing a large split because it is a noisy problem and a well-performing model requires as much data as possible to learn the complex classification function.

```
# split into train and test
n_train = int(0.9 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 22.4: Example of preparing the dataset for modeling.

Next, we can define and combine the model. The model will expect samples with two input variables. The model then has a single hidden layer with 50 nodes and a rectified linear activation function, then an output layer with 3 nodes to predict the probability of each of the 3 classes, and a softmax activation function. Because the problem is multiclass, we will use the categorical cross-entropy loss function to optimize the model and the efficient Adam flavor of stochastic gradient descent.

```
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 22.5: Example of defining the MLP model.

The model is fit for 50 training epochs and we will evaluate the model each epoch on the test set, using the test set as a validation set.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=50, verbose=0)
```

Listing 22.6: Example of fitting the MLP model.

At the end of the run, we will evaluate the performance of the model on both the train and the test sets.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 22.7: Example of evaluating the MLP model.

Then finally, we will plot learning curves of the model accuracy over each training epoch on both the training and test datasets.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 22.8: Example of plotting learning curves for the MLP model.

The complete example is listed below.

```

# develop an mlp for blobs dataset
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = int(0.9 * X.shape[0])
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=50, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 22.9: Example of fitting an MLP on the blobs problem.

Running the example first prints the performance of the final model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved about 83% accuracy on the training dataset and about 88% accuracy on the test dataset. The chosen split of the dataset into train and test sets means that the test set is small and not representative of the broader problem. In turn, performance on the test set is not representative of the model; in this case, it is optimistically biased.

Train: 0.832, Test: 0.850

Listing 22.10: Example output fitting an MLP on the blobs problem.

A line plot is also created showing the learning curves for the model accuracy on the train and test sets over each training epoch. We can see that the model has a reasonably stable fit.

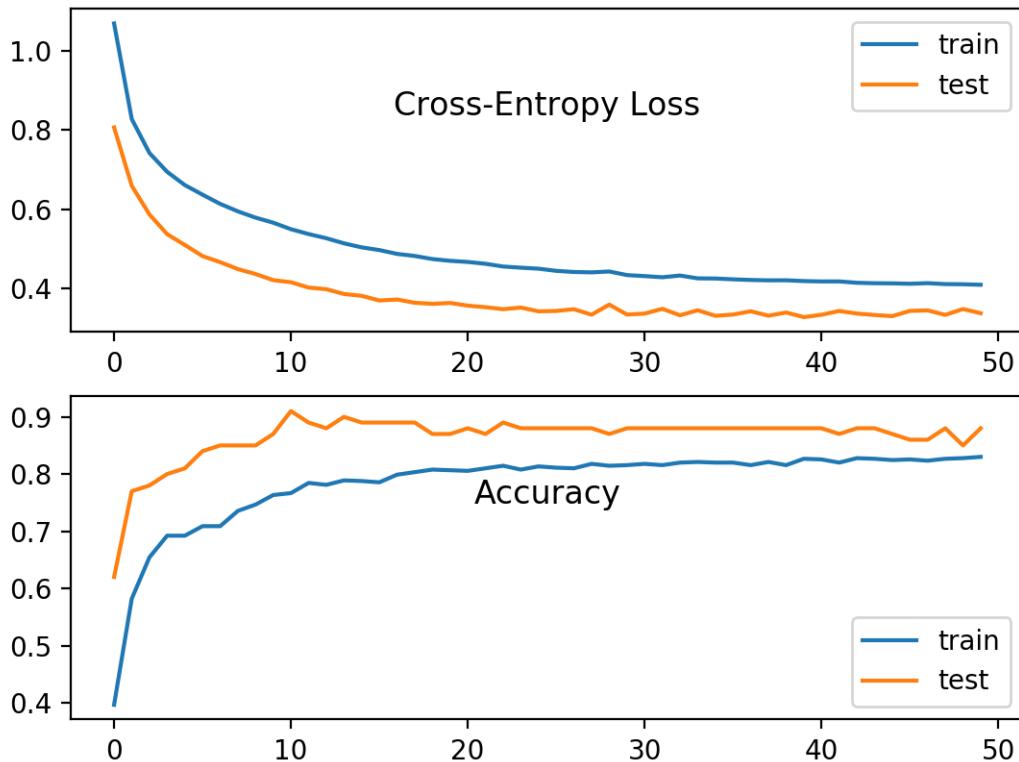


Figure 22.2: Line Plot Learning Curves of Model Accuracy on Train and Test Dataset Over Each Training Epoch.

22.2.3 Random Splits Ensemble

The instability of the model and the small test dataset mean that we don't really know how well this model will perform on new data in general. We can try a simple resampling method of repeatedly generating new random splits of the dataset in train and test sets and fit new models. Calculating the average of the performance of the model across each split will give a better estimate of the model's generalization error. We can then combine multiple models trained on the random splits with the expectation that performance of the ensemble is likely to be more stable and better than the average single model.

We will generate 10 times more sample points from the problem domain and hold them back as an unseen dataset. The evaluation of a model on this much larger dataset will be used as a proxy or a much more accurate estimate of the generalization error of a model for this problem.

This extra dataset is not a test dataset. Technically, it is for the purposes of this demonstration, but we are pretending that this data is unavailable at model training time.

```
# generate 2d classification dataset
dataX, datay = make_blobs(n_samples=55000, centers=3, n_features=2, cluster_std=2,
    random_state=2)
X, newX = dataX[:5000, :], dataX[5000:, :]
y, newy = datay[:5000], datay[5000:]
```

Listing 22.11: Example creating a much larger dataset for modeling.

So now we have 5,000 examples to train our model and estimate its general performance. We also have 50,000 examples that we can use to better approximate the true general performance of a single model or an ensemble. Next, we need a function to fit and evaluate a single model on a training dataset and return the performance of the fit model on a test dataset. We also need the model that was fit so that we can use it as part of an ensemble. The `evaluate_model()` function below implements this behavior.

```
# evaluate a single mlp model
def evaluate_model(trainX, trainy, testX, testy):
    # encode targets
    trainy_enc = to_categorical(trainy)
    testy_enc = to_categorical(testy)
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy_enc, epochs=50, verbose=0)
    # evaluate the model
    _, test_acc = model.evaluate(testX, testy_enc, verbose=0)
    return model, test_acc
```

Listing 22.12: Example defining a function to fit and evaluate an MLP model.

Next, we can create random splits of the training dataset and fit and evaluate models on each split. We can use the `train_test_split()` function from the scikit-learn library to create a random split of a dataset into train and test sets. It takes the X and y arrays as arguments and the `test_size` specifies the size of the test dataset in terms of a percentage. We will use 10% of the 5,000 examples as the test. We can then call the `evaluate_model()` to fit and evaluate a model. The returned accuracy and model can then be added to lists for later use. In this example, we will limit the number of splits, and in turn, the number of fit models to 10.

```
# multiple train-test splits
n_splits = 10
scores, members = list(), list()
for _ in range(n_splits):
    # split data
    trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.10)
    # evaluate model
    model, test_acc = evaluate_model(trainX, trainy, testX, testy)
    print('>%.3f' % test_acc)
    scores.append(test_acc)
    members.append(model)
```

Listing 22.13: Example creating ensemble members from random train/test splits.

After fitting and evaluating the models, we can estimate the expected performance of a given model with the chosen configuration for the domain.

```
# summarize expected performance
print('Estimated Accuracy %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 22.14: Example summarizing the performance of single models.

We don't know how many of the models will be useful in the ensemble. It is likely that there will be a point of diminishing returns, after which the addition of further members no longer changes the performance of the ensemble. Nevertheless, we can evaluate different ensemble sizes from 1 to 10 and plot their performance on the unseen holdout dataset. We can also evaluate each model on the holdout dataset and calculate the average of these scores to get a much better approximation of the true performance of the chosen model on the prediction problem.

```
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, n_splits+1):
    ensemble_score = evaluate_n_members(members, i, newX, newy)
    newy_enc = to_categorical(newy)
    _, single_score = members[i-1].evaluate(newX, newy_enc, verbose=0)
    print('> %d: single=%f, ensemble=%f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
```

Listing 22.15: Example of evaluating ensembles of differing sizes.

Finally, we can compare and calculate a more robust estimate of the general performance of an average model on the prediction problem, then plot the performance of the ensemble size to accuracy on the holdout dataset.

```
# plot score vs number of ensemble members
print('Accuracy %.3f (%.3f)' % (mean(single_scores), std(single_scores)))
x_axis = [i for i in range(1, n_splits+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()
```

Listing 22.16: Example of plotting ensemble performance vs ensemble size.

Tying all of this together, the complete example is listed below.

```
# random-splits mlp ensemble on blobs dataset
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
from numpy import mean
from numpy import std
from numpy import array
from numpy import argmax
```

```
import numpy

# evaluate a single mlp model
def evaluate_model(trainX, trainy, testX, testy):
    # encode targets
    trainy_enc = to_categorical(trainy)
    testy_enc = to_categorical(testy)
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy_enc, epochs=50, verbose=0)
    # evaluate the model
    _, test_acc = model.evaluate(testX, testy_enc, verbose=0)
    return model, test_acc

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# generate 2d classification dataset
dataX, datay = make_blobs(n_samples=55000, centers=3, n_features=2, cluster_std=2,
    random_state=2)
X, newX = dataX[:5000, :], dataX[5000:, :]
y, newy = datay[:5000], datay[5000:]
# multiple train-test splits
n_splits = 10
scores, members = list(), list()
for _ in range(n_splits):
    # split data
    trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.10)
    # evaluate model
    model, test_acc = evaluate_model(trainX, trainy, testX, testy)
    print('>%.3f' % test_acc)
    scores.append(test_acc)
    members.append(model)
# summarize expected performance
print('Estimated Accuracy %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, n_splits+1):
    ensemble_score = evaluate_n_members(members, i, newX, newy)
    newy_enc = to_categorical(newy)
    _, single_score = members[i-1].evaluate(newX, newy_enc, verbose=0)
    print('> %d: single=% .3f, ensemble=% .3f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# plot score vs number of ensemble members
print('Accuracy %.3f (%.3f)' % (mean(single_scores), std(single_scores)))
x_axis = [i for i in range(1, n_splits+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()
```

Listing 22.17: Example of a random splits ensemble on the blobs problem.

Running the example first fits and evaluates 10 models on 10 different random splits of the dataset into train and test sets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

From these scores, we estimate that the average model fit on the dataset will achieve an accuracy of about 83% with a standard deviation of about 1.9%.

```
>0.816
>0.836
>0.818
>0.806
>0.814
>0.824
>0.830
>0.848
>0.868
>0.858
Estimated Accuracy 0.832 (0.019)
```

Listing 22.18: Example output from summarizing single model performance.

We then evaluate the performance of each model on the unseen dataset and the performance of ensembles of models from 1 to 10 models. From these scores, we can see that a more accurate estimate of the performance of an average model on this problem is about 82% and that the estimated performance is optimistic.

```
> 1: single=0.821, ensemble=0.821
> 2: single=0.821, ensemble=0.820
> 3: single=0.820, ensemble=0.820
> 4: single=0.820, ensemble=0.821
> 5: single=0.821, ensemble=0.821
> 6: single=0.820, ensemble=0.821
> 7: single=0.820, ensemble=0.821
> 8: single=0.820, ensemble=0.821
> 9: single=0.820, ensemble=0.820
> 10: single=0.820, ensemble=0.821
```

```
Accuracy 0.820 (0.000)
```

Listing 22.19: Example output from summarizing train/test split ensemble performance.

A lot of the difference between the accuracy scores is happening in the fractions of percent. A graph is created showing the accuracy of each individual model on the unseen holdout dataset as blue dots and the performance of an ensemble with a given number of members from 1-10 as an orange line and dots. We can see that using an ensemble of 4-to-8 members, at least in this case, results in an accuracy that is better than most of the individual runs (orange line is above many blue dots).

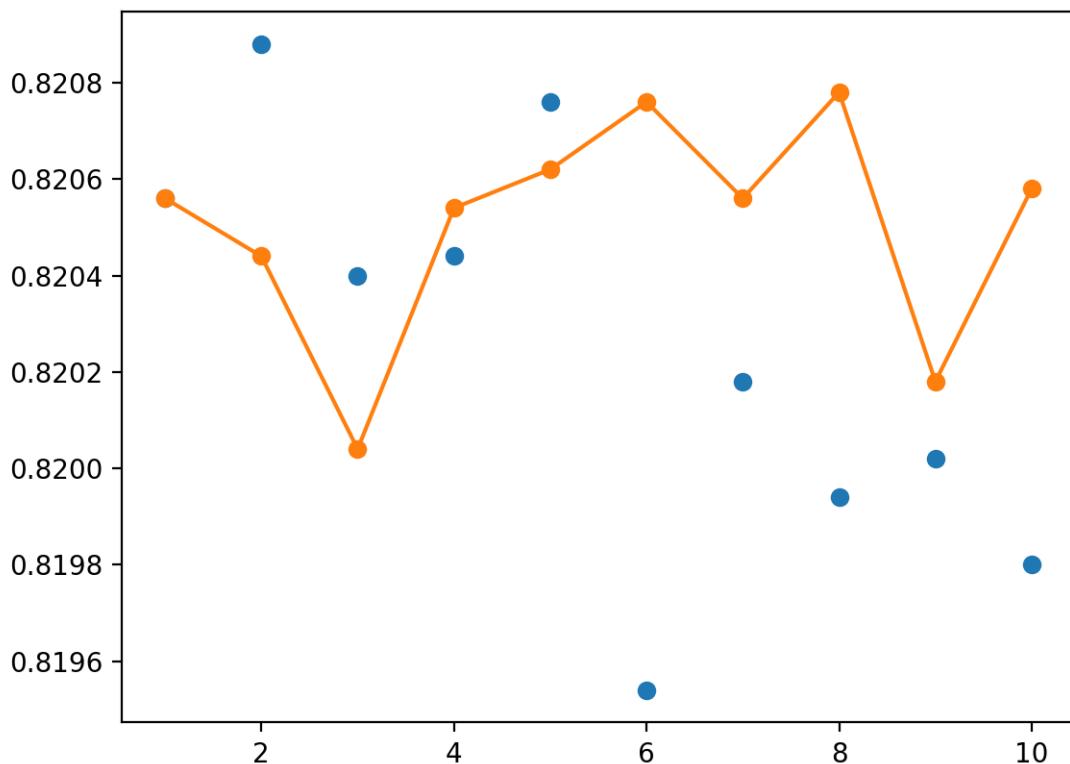


Figure 22.3: Line Plot Showing Single Model Accuracy (blue dots) vs Accuracy of Ensembles of Varying Size for Random-Split Resampling.

The graph does show some individual models can perform better than an ensemble of models (blue dots above the orange line), but we are unable to choose these models. Here, we demonstrate that without additional data (e.g. the out-of-sample dataset) that an ensemble of 4-to-8 members will give better on average performance than a randomly selected train-test model. More repeats (e.g. 30 or 100) may result in a more stable ensemble performance.

22.2.4 Cross-Validation Ensemble

A problem with repeated random splits as a resampling method for estimating the average performance of model is that it is optimistic. An approach designed to be less optimistic and is widely used as a result is the k -fold cross-validation method. The method is less biased because each example in the dataset is only used one time in the test dataset to estimate model performance, unlike random train-test splits where a given example may be used to evaluate a model many times. The procedure has a single parameter called k that refers to the number of groups that a given data sample is to be split into. The average of the scores of each model provides a less biased estimate of model performance. A typical value for k is 10.

Because neural network models are computationally very expensive to train, it is common to use the best performing model during cross-validation as the final model. Alternately, the resulting models from the cross-validation process can be combined to provide a cross-validation ensemble that is likely to have better performance on average than a given single model. We can use the `KFold` class from scikit-learn to split the dataset into k folds. It takes as arguments the number of splits, whether or not to shuffle the sample, and the seed for the pseudorandom number generator used prior to the shuffle.

```
# prepare the k-fold cross-validation configuration
n_folds = 10
kfold = KFold(n_folds, True, 1)
```

Listing 22.20: Example of configuring k -fold cross-validation.

Once the class is instantiated, it can be enumerated to get each split of indexes into the dataset for the train and test sets.

```
# cross validation estimation of performance
scores, members = list(), list()
for train_ix, test_ix in kfold.split(X):
    # select samples
    trainX, trainy = X[train_ix], y[train_ix]
    testX, testy = X[test_ix], y[test_ix]
    # evaluate model
    model, test_acc = evaluate_model(trainX, trainy, testX, testy)
    print('>%.3f' % test_acc)
    scores.append(test_acc)
    members.append(model)
```

Listing 22.21: Example of creating a k -fold cross-validation ensemble.

Once the scores are calculated on each fold, the average of the scores can be used to report the expected performance of the approach.

```
# summarize expected performance
print('Estimated Accuracy %.3f (%.3f)' % (mean(scores), std(scores)))
```

Listing 22.22: Example of estimating the performance of single models.

Now that we have collected the 10 models evaluated on the 10 folds, we can use them to create a cross-validation ensemble. It seems intuitive to use all 10 models in the ensemble, nevertheless, we can evaluate the accuracy of each subset of ensembles from 1 to 10 members as we did in the previous section. The complete example of analyzing the cross-validation ensemble is listed below.

```
# cross-validation mlp ensemble on blobs dataset
from sklearn.datasets import make_blobs
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
from numpy import mean
from numpy import std
from numpy import array
from numpy import argmax
import numpy

# evaluate a single mlp model
def evaluate_model(trainX, trainy, testX, testy):
    # encode targets
    trainy_enc = to_categorical(trainy)
    testy_enc = to_categorical(testy)
    # define model
    model = Sequential()
    model.add(Dense(50, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy_enc, epochs=50, verbose=0)
    # evaluate the model
    _, test_acc = model.evaluate(testX, testy_enc, verbose=0)
    return model, test_acc

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# generate 2d classification dataset
dataX, datay = make_blobs(n_samples=55000, centers=3, n_features=2, cluster_std=2,
    random_state=2)
X, newX = dataX[:5000, :], dataX[5000:, :]
y, newy = datay[:5000], datay[5000:]
```

```

# prepare the k-fold cross-validation configuration
n_folds = 10
kfold = KFold(n_folds, True, 1)
# cross validation estimation of performance
scores, members = list(), list()
for train_ix, test_ix in kfold.split(X):
    # select samples
    trainX, trainy = X[train_ix], y[train_ix]
    testX, testy = X[test_ix], y[test_ix]
    # evaluate model
    model, test_acc = evaluate_model(trainX, trainy, testX, testy)
    print('>%.3f' % test_acc)
    scores.append(test_acc)
    members.append(model)
# summarize expected performance
print('Estimated Accuracy %.3f (%.3f)' % (mean(scores), std(scores)))
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, n_folds+1):
    ensemble_score = evaluate_n_members(members, i, newX, newy)
    newy_enc = to_categorical(newy)
    _, single_score = members[i-1].evaluate(newX, newy_enc, verbose=0)
    print('> %d: single=%.3f, ensemble=%.3f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# plot score vs number of ensemble members
print('Accuracy %.3f (%.3f)' % (mean(single_scores), std(single_scores)))
x_axis = [i for i in range(1, n_folds+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()

```

Listing 22.23: Example of a cross-validation ensemble on the blobs problem.

Running the example first prints the performance of each of the 10 models on each of the folds of the cross-validation.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The average performance of these models is reported as about 82%, which appears to be less optimistic than the random-splits approach used in the previous section.

```

>0.834
>0.870
>0.818
>0.806
>0.836
>0.804
>0.820
>0.830
>0.828
>0.822
Estimated Accuracy 0.827 (0.018)

```

Listing 22.24: Example output from summarizing single model performance.

Next, each of the saved models is evaluated on the unseen holdout set. The average of these scores is also about 82%, highlighting that, at least in this case, the cross-validation estimation of the general performance of the model was reasonable.

```
> 1: single=0.819, ensemble=0.819
> 2: single=0.820, ensemble=0.820
> 3: single=0.820, ensemble=0.820
> 4: single=0.821, ensemble=0.821
> 5: single=0.820, ensemble=0.821
> 6: single=0.821, ensemble=0.821
> 7: single=0.820, ensemble=0.820
> 8: single=0.819, ensemble=0.821
> 9: single=0.820, ensemble=0.821
> 10: single=0.820, ensemble=0.821
Accuracy 0.820 (0.001)
```

Listing 22.25: Example output from summarizing cross-validation ensemble performance.

A graph of single model accuracy (blue dots) and ensemble size vs accuracy (orange line) is created. As in the previous example, the real difference between the performance of the models is in the fractions of percent in model accuracy. The orange line shows that as the number of members increases, the accuracy of the ensemble increases to a point of diminishing returns. We can see that, at least in this case, using four or more of the models fit during cross-validation in an ensemble gives better performance than almost all individual models. We can also see that a default strategy of using all models in the ensemble would be effective.

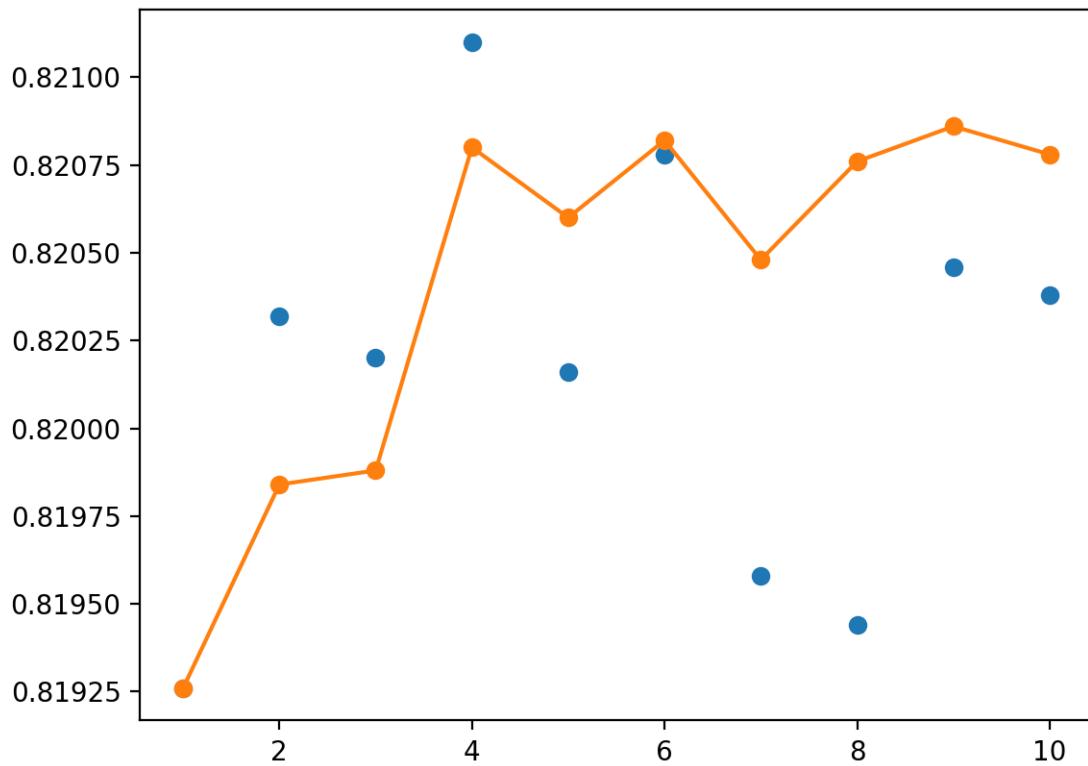


Figure 22.4: Line Plot Showing Single Model Accuracy (blue dots) vs Accuracy of Ensembles of Varying Size for Cross-Validation Resampling.

22.2.5 Bagging Ensemble

A limitation of random splits and k -fold cross-validation from the perspective of ensemble learning is that the models are very similar. The bootstrap method is a statistical technique for estimating quantities about a population by averaging estimates from multiple small data samples. Importantly, samples are constructed by drawing observations from a large data sample one at a time and returning them to the data sample after they have been chosen. This allows a given observation to be included in a given small sample more than once. This approach to sampling is called sampling with replacement.

The method can be used to estimate the performance of neural network models. Examples not selected in a given sample can be used as a test set to estimate the performance of the model. The bootstrap is a robust method for estimating model performance. It does suffer a little from an optimistic bias, but is often almost as accurate as k -fold cross-validation in practice. The benefit for ensemble learning is that each data sample is biased, allowing a given example to appear many times in the sample. This, in turn, means that the models trained on those samples will be biased, importantly in different ways. The result can be ensemble predictions that can be more accurate.

Generally, use of the bootstrap method in ensemble learning is referred to as bootstrap

aggregation or bagging. We can use the `resample()` function from scikit-learn to select a subsample with replacement. The function takes an array to subsample and the size of the resample as arguments. We will perform the selection in rows indices that we can in turn use to select rows in the `X` and `y` arrays. The size of the sample will be 4,500, or 90% of the data, although the test set may be larger than 10% as given the use of resampling, more than 500 examples may have been left unselected.

```
# multiple train-test splits
n_splits = 10
scores, members = list(), list()
for _ in range(n_splits):
    # select indexes
    ix = [i for i in range(len(X))]
    train_ix = resample(ix, replace=True, n_samples=4500)
    test_ix = [x for x in ix if x not in train_ix]
    # select data
    trainX, trainy = X[train_ix], y[train_ix]
    testX, testy = X[test_ix], y[test_ix]
    # evaluate model
    model, test_acc = evaluate_model(trainX, trainy, testX, testy)
    print('>%.3f' % test_acc)
    scores.append(test_acc)
    members.append(model)
```

Listing 22.26: Example of creating a bootstrap ensemble.

It is common to use simple overfit models like unpruned decision trees when using a bagging ensemble learning strategy (e.g. an ensemble averaging used to add bias to a suite of high variance models). Better performance may be seen with over-constrained and overfit neural networks. Nevertheless, we will use the same MLP from previous sections in this example. Additionally, it is common to continue to add ensemble members in bagging until the performance of the ensemble plateaus, as bagging does not overfit the dataset. We will again limit the number of members to 10 as in previous examples. The complete example of bootstrap aggregation for estimating model performance and ensemble learning with a Multilayer Perceptron is listed below.

```
# bagging mlp ensemble on blobs dataset
from sklearn.datasets import make_blobs
from sklearn.utils import resample
from sklearn.metrics import accuracy_score
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
from numpy import mean
from numpy import std
from numpy import array
from numpy import argmax
import numpy

# evaluate a single mlp model
def evaluate_model(trainX, trainy, testX, testy):
    # encode targets
    trainy_enc = to_categorical(trainy)
    testy_enc = to_categorical(testy)
```

```
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
model.fit(trainX, trainy_enc, epochs=50, verbose=0)
# evaluate the model
_, test_acc = model.evaluate(testX, testy_enc, verbose=0)
return model, test_acc

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# generate 2d classification dataset
dataX, datay = make_blobs(n_samples=55000, centers=3, n_features=2, cluster_std=2,
    random_state=2)
X, newX = dataX[:5000, :], dataX[5000:, :]
y, newy = datay[:5000], datay[5000:]
# multiple train-test splits
n_splits = 10
scores, members = list(), list()
for _ in range(n_splits):
    # select indexes
    ix = [i for i in range(len(X))]
    train_ix = resample(ix, replace=True, n_samples=4500)
    test_ix = [x for x in ix if x not in train_ix]
    # select data
    trainX, trainy = X[train_ix], y[train_ix]
    testX, testy = X[test_ix], y[test_ix]
    # evaluate model
    model, test_acc = evaluate_model(trainX, trainy, testX, testy)
    print('>%.3f' % test_acc)
    scores.append(test_acc)
    members.append(model)
# summarize expected performance
print('Estimated Accuracy %.3f (%.3f)' % (mean(scores), std(scores)))
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
```

```

for i in range(1, n_splits+1):
    ensemble_score = evaluate_n_members(members, i, newX, newy)
    newy_enc = to_categorical(newy)
    _, single_score = members[i-1].evaluate(newX, newy_enc, verbose=0)
    print('> %d: single=% .3f, ensemble=% .3f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# plot score vs number of ensemble members
print('Accuracy %.3f (%.3f)' % (mean(single_scores), std(single_scores)))
x_axis = [i for i in range(1, n_splits+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()

```

Listing 22.27: Example of a bagging ensemble on the blobs problem.

Running the example prints the model performance on the unused examples for each bootstrap sample.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that, in this case, the expected performance of the model is less optimistic than random train-test splits and is perhaps quite similar to the finding for k -fold cross-validation.

```

>0.829
>0.820
>0.830
>0.821
>0.831
>0.820
>0.834
>0.815
>0.829
>0.827
Estimated Accuracy 0.825 (0.006)

```

Listing 22.28: Example output from summarizing single model performance.

Perhaps due to the bootstrap sampling procedure, we see that the actual performance of each model is a little worse on the much larger unseen holdout dataset. This is to be expected given the bias introduced by the sampling with replacement of the bootstrap.

```

> 1: single=0.819, ensemble=0.819
> 2: single=0.818, ensemble=0.820
> 3: single=0.820, ensemble=0.820
> 4: single=0.818, ensemble=0.821
> 5: single=0.819, ensemble=0.820
> 6: single=0.820, ensemble=0.820
> 7: single=0.820, ensemble=0.820
> 8: single=0.819, ensemble=0.820
> 9: single=0.820, ensemble=0.820
> 10: single=0.819, ensemble=0.820
Accuracy 0.819 (0.001)

```

Listing 22.29: Example output from summarizing bagging ensemble performance.

The created line plot is encouraging. We see that after about four members that the bagged ensemble achieves better performance on the holdout dataset than any individual model. No doubt, given the slightly lower average performance of individual models.

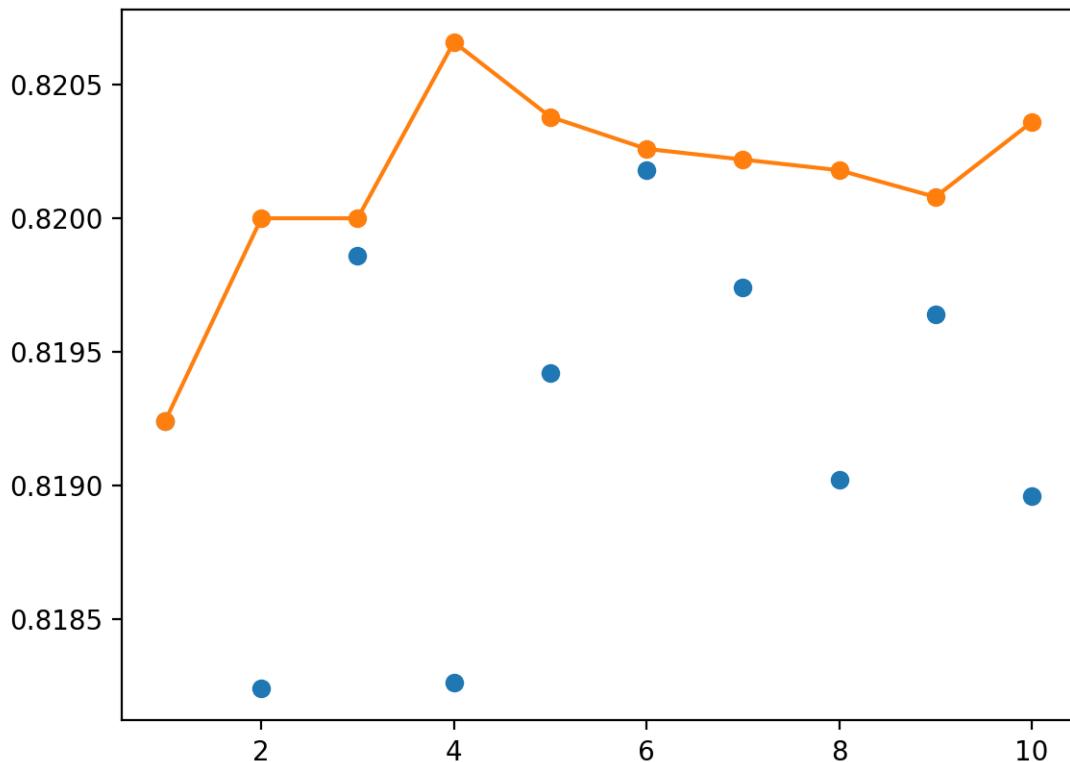


Figure 22.5: Line Plot Showing Single Model Accuracy (blue dots) vs Accuracy of Ensembles of Varying Size for Bagging.

22.3 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Single Model.** Compare the performance of each ensemble to one model trained on all available data.
- **CV Ensemble Size.** Experiment with larger and smaller ensemble sizes for the cross-validation ensemble and compare their performance.
- **Bagging Ensemble Limit.** Increase the number of members in the bagging ensemble to find the point of diminishing returns.

If you explore any of these extensions, I'd love to know.

22.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

22.4.1 Papers

- *Neural Network Ensembles, Cross Validation, and Active Learning*, 1995.
<http://papers.nips.cc/paper/1001-neural-network-ensembles-cross-validation-and-active-learning.pdf>

22.4.2 APIs

- Getting started with the Keras Sequential model.
<https://keras.io/getting-started/sequential-model-guide/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- numpy.argmax API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html>
- sklearn.datasets.make_blobs API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
- sklearn.model_selection.train_test_split API.
http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
- sklearn.model_selection.KFold API.
http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html
- sklearn.utils.resample API.
<http://scikit-learn.org/stable/modules/generated/sklearn.utils.resample.html>

22.4.3 Articles

- Cross-validation (statistics), Wikipedia.
[https://en.wikipedia.org/wiki/Cross-validation_\(statistics\)](https://en.wikipedia.org/wiki/Cross-validation_(statistics))
- Bootstrapping (statistics), Wikipedia.
[https://en.wikipedia.org/wiki/Bootstrapping_\(statistics\)](https://en.wikipedia.org/wiki/Bootstrapping_(statistics))
- Bootstrap aggregating, Wikipedia.
https://en.wikipedia.org/wiki/Bootstrap_aggregating

22.5 Summary

In this tutorial, you discovered how to develop a suite of different resampling-based ensembles for deep learning neural network models. Specifically, you learned:

- How to estimate model performance using random-splits and develop an ensemble from the models.
- How to estimate performance using 10-fold cross-validation and develop a cross-validation ensemble.
- How to estimate performance using the bootstrap and combine models using a bagging ensemble.

22.5.1 Next

In the next tutorial, you will discover how to develop ensemble members from a contiguous block of epochs from a single model training run.

Chapter 23

Models from Contiguous Epochs with Horizontal Voting Ensembles

Predictive modeling problems where the training dataset is small relative to the number of unlabeled examples are challenging. Neural networks can perform well on these types of problems, although they can suffer from high variance in model performance as measured on a training or hold-out validation datasets. This makes choosing which model to use as the final model risky, as there is no clear signal as to which model is better than another toward the end of the training run. The horizontal voting ensemble is a simple method to address this issue, where a collection of models saved over contiguous training epochs towards the end of a training run are saved and used as an ensemble that results in more stable and better performance on average than randomly choosing a single final model. In this tutorial, you will discover how to reduce the variance of a final deep learning neural network model using a horizontal voting ensemble. After completing this tutorial, you will know:

- That it is challenging to choose a final neural network model that has high variance on a training dataset.
- Horizontal voting ensembles provide a way to reduce variance and improve average model performance for models with high variance using a single training run.
- How to develop a horizontal voting ensemble in Python using Keras to improve the performance of a final Multilayer Perceptron model for multiclass classification.

Let's get started.

23.1 Horizontal Voting Ensemble

Ensemble learning combines the predictions from multiple models. A challenge when using ensemble learning when using deep learning methods is that given the use of very large datasets and large models, a given training run may take days, weeks, or even months. Training multiple models may not be feasible. An alternative source of models that may contribute to an ensemble are the state of a single model at different points during training. Horizontal voting is an ensemble method proposed by Jingjing Xie, et al. in their 2013 paper *Horizontal and Vertical Ensemble with Deep Representation for Classification*.

The method involves using multiple models from the end of a contiguous block of epochs before the end of training in an ensemble to make predictions. The approach was developed specifically for those predictive modeling problems where the training dataset is relatively small compared to the number of predictions required by the model. This results in a model that has a high variance in performance during training. In this situation, using the final model or any given model toward the end of the training process is risky given the variance in performance.

... the error rate of classification would first decline and then tend to be stable with the training epoch grows. But when size of labeled training set is too small, the error rate would oscillate [...] So it is difficult to choose a *magic* epoch to obtain a reliable output.

— *Horizontal and Vertical Ensemble with Deep Representation for Classification*, 2013.

Instead, the authors suggest using all of the models in an ensemble from a contiguous block of epochs during training, such as models from the last 200 epochs. The result are predictions by the ensemble that are as good as or better than any single model in the ensemble.

To reduce the instability, we put forward a method called Horizontal Voting. First, networks trained for a relatively stable range of epoch are selected. The predictions of the probability of each label are produced by standard classifiers with top level representation of the selected epoch, and then averaged.

— *Horizontal and Vertical Ensemble with Deep Representation for Classification*, 2013.

As such, the horizontal voting ensemble method provides an ideal method for both cases where a given model requires vast computational resources to train, and/or cases where final model selection is challenging given the high variance of training due to the use of a relatively small training dataset. Now that are we are familiar with horizontal voting, we can implement the procedure.

23.2 Horizontal Voting Ensembles Case Study

In this section, we will demonstrate how to use the horizontal voting ensemble to reduce the variance of an MLP on a simple multiclass classification problem. This example provides a template for applying the horizontal voting ensemble to your own neural network for classification and regression problems.

23.2.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate a horizontal voting ensemble. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. The problem can be configured to have two input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same data points.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

Listing 23.1: Example of creating samples for the blobs problem.

The results are the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can graph each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot for each class value
for class_value in range(3):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 23.2: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (can be separated by a line) causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions resulting in a high variance.

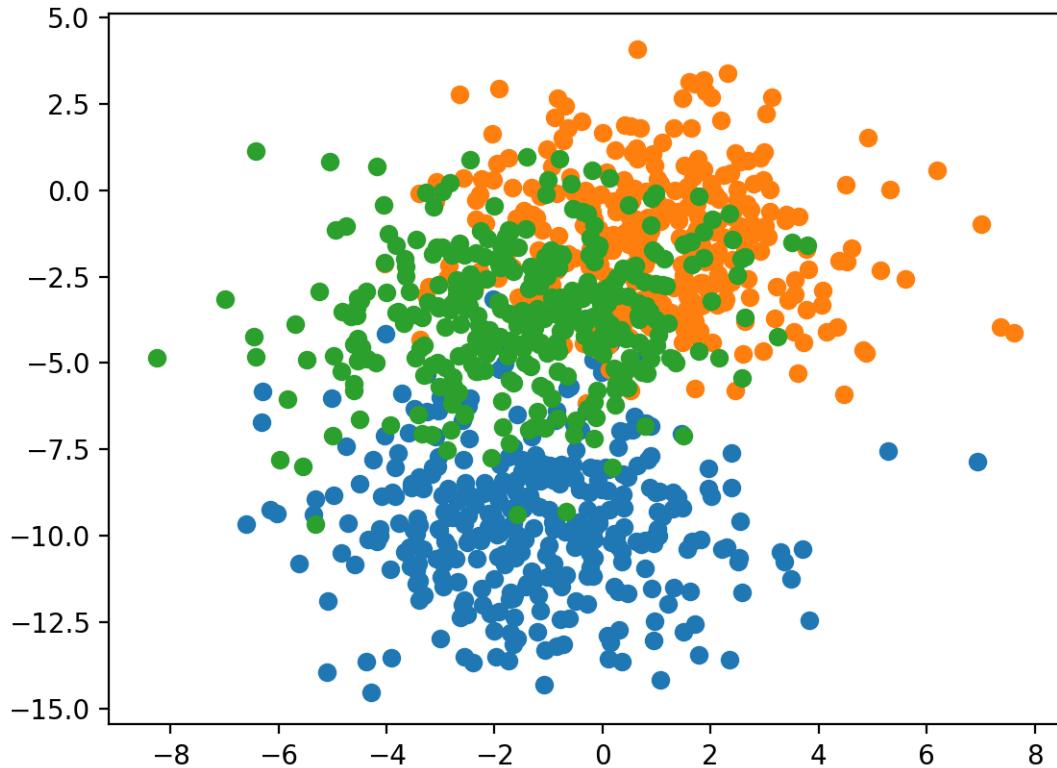


Figure 23.1: Scatter Plot of Blobs Dataset With Three Classes and Points Colored by Class Value.

23.2.2 Multilayer Perceptron Model

Before we define a model, we need to contrive a problem that is appropriate for a horizontal voting ensemble. In our problem, the training dataset is relatively small. Specifically, there is a 10:1 ratio of examples in the training dataset to the holdout dataset. This mimics a situation where we may have a vast number of unlabeled examples and a small number of labeled examples with which to train a model. We will create 1,100 data points from the blobs problem. The model will be trained on the first 100 points and the remaining 1,000 will be held back in a test dataset, unavailable to the model.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
print(trainX.shape, testX.shape)
```

Listing 23.3: Example of preparing dataset for modeling.

The problem is a multiclass classification problem, and we will model it using a softmax activation function on the output layer. This means that the model will predict a vector

with three elements with the probability that the sample belongs to each of the three classes. Therefore, we must one hot encode the class values, ideally before we split the rows into the train, test, and validation datasets so that it is a single function call.

```
y = to_categorical(y)
```

Listing 23.4: Example of one hot encoding the target value.

Next, we can define and combine the model. The model will expect samples with two input variables. The model then has a single hidden layer with 25 nodes and a rectified linear activation function, then an output layer with three nodes to predict the probability of each of the three classes and a softmax activation function. Because the problem is multiclass, we will use the categorical cross-entropy loss function to optimize the model and the efficient Adam flavor of stochastic gradient descent.

```
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 23.5: Example of defining MLP model.

The model is fit for 1,000 training epochs and we will evaluate the model each epoch on the training set, using the test set as a validation set.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=1000, verbose=0)
```

Listing 23.6: Example of fitting MLP model.

At the end of the run, we will evaluate the performance of the model on the train and test sets.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 23.7: Example of evaluating MLP model.

Then finally, we will plot learning curves of the model accuracy over each training epoch on both the training and validation datasets.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 23.8: Example of plotting learning curves for MLP model.

Tying all of this together, the complete example is listed below.

```
# develop an mlp for blobs dataset
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=1000, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 23.9: Example of fitting an MLP on the blobs problem.

Running the example first prints the shape of each dataset for confirmation, then the performance of the final model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved about 86% accuracy on the training dataset that we know is optimistic, and about 81% on the test dataset, which we would expect to be more realistic.

Train: 0.860, Test: 0.814

Listing 23.10: Example output fitting an MLP on the blobs problem.

A line plot is also created showing the learning curves for the model accuracy on the train and test sets over each training epoch. We can see that training accuracy is more optimistic over the whole run as we also noted with the final scores. We can see that the accuracy of the model has high variance on the training dataset as compared to the test set, as we would expect. The variance in the model highlights the fact that choosing the model at the end of the run or any model from about epoch 400 is challenging as accuracy on the training dataset has a high variance. We also see a muted version of the variance on the test dataset.

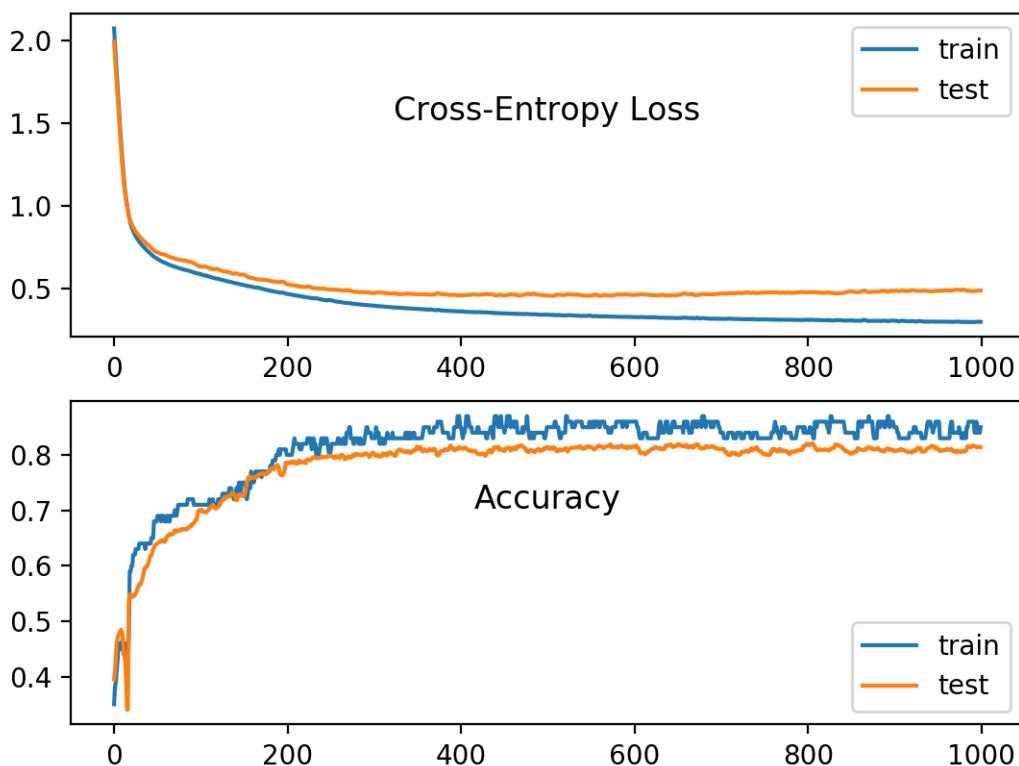


Figure 23.2: Line Plot Learning Curves of Model Accuracy on Train and Test Dataset over Each Training Epoch.

Now that we have identified that the model is a good candidate for a horizontal voting ensemble, we can begin to implement the technique.

23.2.3 Save Horizontal Models

There may be many ways to implement a horizontal voting ensemble. Perhaps the simplest is to manually drive the training process, one epoch at a time, then save models at the end of the epoch if we have exceeded an upper limit on the number of epochs. For example, with our test problem, we will train the model for 1,000 epochs and perhaps save models from epoch 950 onwards (e.g. between and including epochs 950 and 999).

```
# fit model
n_epochs, n_save_after = 1000, 950
for i in range(n_epochs):
    # fit model for a single epoch
    model.fit(trainX, trainy, epochs=1, verbose=0)
    # check if we should save the model
    if i >= n_save_after:
        model.save('models/model_' + str(i) + '.h5')
```

Listing 23.11: Example of fitting and saving horizontal ensemble members.

Models can be saved to file using the `save()` function on the model and specifying a filename that includes the epoch number. To avoid clutter with our source files, we will save all models under a new `models/` folder in the current working directory.

```
# create directory for models
makedirs('models')
```

Listing 23.12: Example of creating folder used to save models.

Note, saving and loading neural network models in Keras requires that you have the `h5py` library installed. You can install this library using `pip` as follows:

```
pip install h5py
```

Listing 23.13: Example installing the `h5py` library with `pip`.

Tying all of this together, the complete example of fitting the model on the training dataset and saving all models from the last 50 epochs is listed below.

```
# save horizontal voting ensemble members during training
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from os import makedirs
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# create directory for models
makedirs('models')
# fit model
n_epochs, n_save_after = 1000, 950
for i in range(n_epochs):
    # fit model for a single epoch
    model.fit(trainX, trainy, epochs=1, verbose=0)
    # check if we should save the model
```

```
if i >= n_save_after:
    model.save('models/model_' + str(i) + '.h5')
```

Listing 23.14: Example of saving horizontal ensemble members to file.

Running the example creates the `models/` folder and saves 50 models into the directory. Note, to re-run this example, you must delete the `models/` directory so that the script can recreate it.

23.2.4 Make Horizontal Ensemble Predictions

Now that we have created the models, we can use them in a horizontal voting ensemble. First, we need to load the models into memory. This is reasonable as the models are small. If you are trying to develop a horizontal voting ensemble with very large models, it might be easier to load models one at a time, make a prediction, then load the next model and repeat the process. The function `load_all_models()` below will load models from the `models/` directory. It takes the start and end epochs as arguments so that you can experiment with different groups of models saved over contiguous epochs.

```
# load models from file
def load_all_models(n_start, n_end):
    all_models = list()
    for epoch in range(n_start, n_end):
        # define filename for this ensemble
        filename = 'models/model_' + str(epoch) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models
```

Listing 23.15: Example of a function for loading saved ensemble members.

We can call the function to load all of the models. We can then reverse the list of models so that the models at the end of the run are at the beginning of the list. This will be helpful later when we test voting ensembles of different sizes, including models sequentially from the end of the run backward through training epochs, in case the best models really were at the end of the run.

```
# load models in order
members = load_all_models(950, 1000)
print('Loaded %d models' % len(members))
# reverse loaded models so we build the ensemble with the last models first
members = list(reversed(members))
```

Listing 23.16: Example of a loading saved ensemble members.

Next, we can evaluate each saved model on the test dataset, as well as a voting ensemble of the last n contiguous models from training. We want to know how well each model actually performed on the test dataset and, importantly, the distribution of model performance on the test dataset, so that we know how well (or poorly) an average model chosen from the end of the run would perform in practice. We don't know how many members to include in the horizontal

voting ensemble. Therefore, we can test different numbers of contiguous members, working backward from the final model.

First, we need a function to make a prediction with a list of ensemble members. Each member predicts the probabilities for each of the three output classes. The probabilities are added and we use an argmax to select the class with the most support. The `ensemble_predictions()` function below implements this voting based prediction scheme.

```
# make an ensemble prediction for multiclass classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result
```

Listing 23.17: Example of a function for making ensemble predictions.

Next, we need a function to evaluate a subset of the ensemble members of a given size. The subset needs to be selected, predictions made, and the performance of the ensemble estimated by comparing the predictions to the expected values. The `evaluate_n_members()` function below implements this ensemble size evaluation.

```
# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)
```

Listing 23.18: Example of a function for evaluating an ensemble of a given size.

We can now enumerate through different sized horizontal voting ensembles from 1 to 50. Each member is evaluated alone, then the ensemble of that size is evaluated and scores are recorded.

```
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, len(members)+1):
    # evaluate model with i members
    ensemble_score = evaluate_n_members(members, i, testX, testy)
    # evaluate the i'th model standalone
    testy_enc = to_categorical(testy)
    _, single_score = members[i-1].evaluate(testX, testy_enc, verbose=0)
    # summarize this step
    print('> %d: single=%3f, ensemble=%3f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
```

Listing 23.19: Example of evaluating ensembles of different given sizes.

At the end of the evaluations, we report the distribution of scores of single models on the test dataset. The average score is what we would expect on average if we picked any of the saved models as a final model.

```
# summarize average accuracy of a single final model
print('Accuracy %.3f (%.3f)' % (mean(single_scores), std(single_scores)))
```

Listing 23.20: Example of summarizing single model performance.

Finally, we can plot the scores. The scores of each standalone model are plotted as blue dots and line plot is created for each ensemble of contiguous models (orange).

```
# plot score vs number of ensemble members
x_axis = [i for i in range(1, len(members)+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()
```

Listing 23.21: Example of plotting single model and ensemble model performance.

Our expectation is that a fair sized ensemble will outperform a randomly selected model and that there is a point of diminishing returns in choosing the ensemble size. The complete example is listed below.

```
# load models and make predictions using a horizontal voting ensemble
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score
from keras.utils import to_categorical
from keras.models import load_model
from matplotlib import pyplot
from numpy import mean
from numpy import std
from numpy import array
from numpy import argmax
import numpy

# load models from file
def load_all_models(n_start, n_end):
    all_models = list()
    for epoch in range(n_start, n_end):
        # define filename for this ensemble
        filename = 'models/model_' + str(epoch) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
```

```

result = argmax(summed, axis=1)
return result

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# load models in order
members = load_all_models(950, 1000)
print('Loaded %d models' % len(members))
# reverse loaded models so we build the ensemble with the last models first
members = list(reversed(members))
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, len(members)+1):
    # evaluate model with i members
    ensemble_score = evaluate_n_members(members, i, testX, testy)
    # evaluate the i'th model standalone
    testy_enc = to_categorical(testy)
    _, single_score = members[i-1].evaluate(testX, testy_enc, verbose=0)
    # summarize this step
    print('> %d: single=%f, ensemble=%f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# summarize average accuracy of a single final model
print('Accuracy %.3f (%.3f)' % (mean(single_scores), std(single_scores)))
# plot score vs number of ensemble members
x_axis = [i for i in range(1, len(members)+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()

```

Listing 23.22: Example of evaluating horizontal ensemble models.

First, the 50 saved models are loaded into memory.

```

...
>loaded models/model_990.h5
>loaded models/model_991.h5
>loaded models/model_992.h5
>loaded models/model_993.h5
>loaded models/model_994.h5
>loaded models/model_995.h5
>loaded models/model_996.h5
>loaded models/model_997.h5
>loaded models/model_998.h5

```

```
>loaded models/model_999.h5
```

Listing 23.23: Example output from loading the saved models.

Next, the performance of each single model is evaluated on the holdout test dataset, and the ensemble of that size (1, 2, 3, etc.) is created and evaluated on the holdout test dataset.

```
> 1: single=0.814, ensemble=0.814
> 2: single=0.816, ensemble=0.816
> 3: single=0.812, ensemble=0.816
> 4: single=0.812, ensemble=0.815
> 5: single=0.811, ensemble=0.815
...
> 46: single=0.817, ensemble=0.818
> 47: single=0.812, ensemble=0.818
> 48: single=0.811, ensemble=0.818
> 49: single=0.810, ensemble=0.818
> 50: single=0.811, ensemble=0.818
```

Listing 23.24: Example output from evaluating single models.

Roughly, we can see that the ensemble appears to outperform most single models, consistently achieving accuracy around 81.8%. Next, the distribution of the accuracy of single models is reported. We can see that picking any of the saved models at random would result in a model with the accuracy of 81.6% on average with a reasonably tight standard deviation of 0.3%. We would require that a horizontal ensemble out-perform this average in order to be useful.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
Accuracy 0.816 (0.003)
```

Listing 23.25: Example output from summarizing the distribution of single model performance.

Finally, a graph is created summarizing the performance of each single model (blue dot) and the ensemble of each size from 1 to 50 members. We can see from the blue dots that there is no structure to the models over the epochs, e.g. if the last models during training were better, there would be a downward trend in accuracy from left to right. We can see that as we add more ensemble members, the better the performance of the horizontal voting ensemble in the orange line. We can see a flattening of performance on this problem perhaps between 23 and 33 epochs; that might be a good choice.

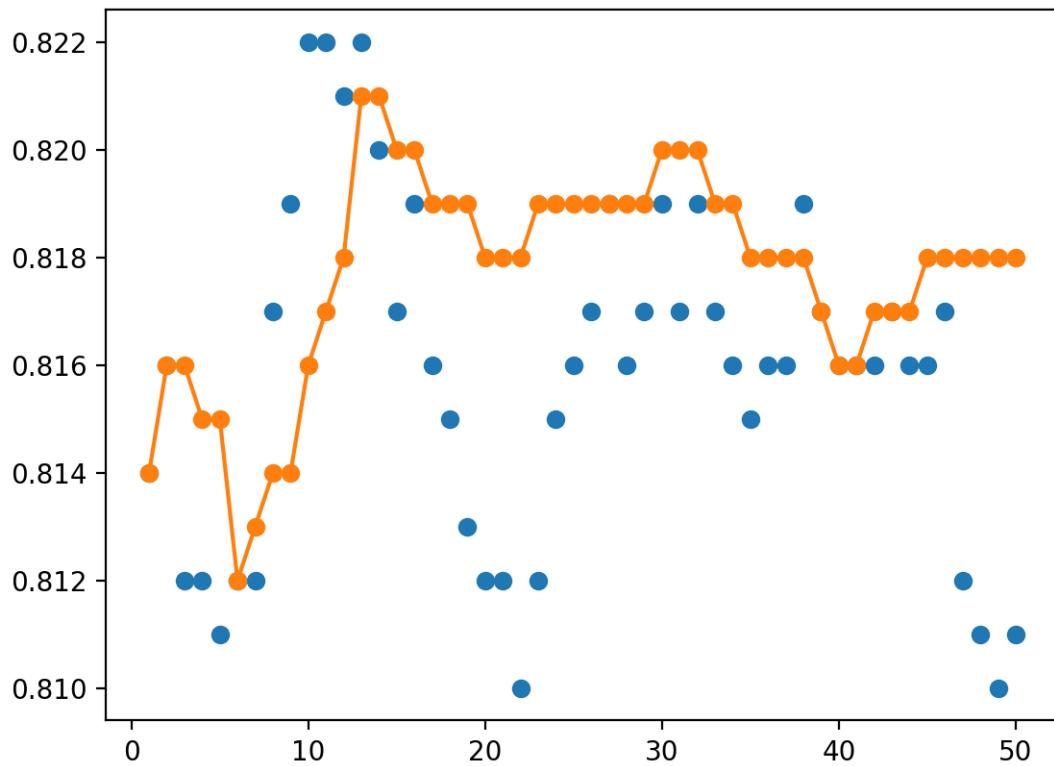


Figure 23.3: Line Plot Showing Single Model Accuracy (blue dots) vs Accuracy of Ensembles of Varying Size With a Horizontal Voting Ensemble.

23.3 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Dataset Size.** Repeat the experiments with a smaller or larger sized dataset with a similar ratio of training to test examples.
- **Larger Ensemble.** Re-run the example with hundreds of final models and report the impact of the large ensemble sizes of accuracy on the test set.
- **Random Sampling of Models.** Re-run the example and compare the performance of ensembles of the same size with models saved over contiguous epochs to a random selection of saved models.

If you explore any of these extensions, I'd love to know.

23.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

23.4.1 Papers

- *Horizontal and Vertical Ensemble with Deep Representation for Classification*, 2013.
<https://arxiv.org/abs/1306.2759>

23.4.2 APIs

- Getting started with the Keras Sequential model.
<https://keras.io/getting-started/sequential-model-guide/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- numpy.argmax API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html>
- sklearn.datasets.make_blobs API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
- How can I save a Keras model?.
<https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model>
- Keras Callbacks API.
<https://keras.io/callbacks>

23.5 Summary

In this tutorial, you discovered how to reduce the variance of a final deep learning neural network model using a horizontal voting ensemble. Specifically, you learned:

- It is challenging to choose a final neural network model that has high variance on a training dataset.
- Horizontal voting ensembles provide a way to reduce variance and improve average model performance for models with high variance using a single training run.
- How to develop a horizontal voting ensemble in Python using Keras to improve the performance of a final Multilayer Perceptron model for multiclass classification.

23.5.1 Next

In the next tutorial, you will discover how to develop ensemble members from the troughs of an aggressive cyclical learning rate schedule over a single model training run.

Chapter 24

Cyclic Learning Rate and Snapshot Ensembles

Model ensembles can achieve lower generalization error than single models but are challenging to develop with deep learning neural networks given the computational cost of training each single model. An alternative is to train multiple model snapshots during a single training run and combine their predictions to make an ensemble prediction. A limitation of this approach is that the saved models will be similar, resulting in similar predictions and predictions errors and not offering much benefit from combining their predictions.

Effective ensembles require a diverse set of skillful ensemble members that have differing distributions of prediction errors. One approach to promoting a diversity of models saved during a single training run is to use an aggressive learning rate schedule that forces large changes in the model weights and, in turn, the nature of the model saved at each snapshot. In this tutorial, you will discover how to develop snapshot ensembles of models saved using an aggressive learning rate schedule over a single training run. After completing this tutorial, you will know:

- Snapshot ensembles combine the predictions from multiple models saved during a single training run.
- Diversity in model snapshots can be achieved through the use of aggressively cycling the learning rate used during a single training run.
- How to save model snapshots during a single run and load snapshot models to make ensemble predictions.

Let's get started.

24.1 Snapshot Ensembles

A problem with ensemble learning with deep learning methods is the large computational cost of training multiple models. This is because of the use of very deep models and very large datasets that can result in model training times that may extend to days, weeks, or even months.

Despite its obvious advantages, the use of ensembling for deep networks is not nearly as widespread as it is for other algorithms. One likely reason for this lack

of adaptation may be the cost of learning multiple neural networks. Training deep networks can last for weeks, even on high performance hardware with GPU acceleration.

— *Snapshot Ensembles: Train 1, get M for free*, 2017.

One approach to ensemble learning for deep learning neural networks is to collect multiple models from a single training run. This addresses the computational cost of training multiple deep learning models as models can be selected and saved during training, then used to make an ensemble prediction. A key benefit of ensemble learning is in improved performance compared to the predictions from single models. This can be achieved through the selection of members that have good skill, but in different ways, providing a diverse set of predictions to be combined. A limitation of collecting multiple models during a single training run is that the models may be good, but too similar.

This can be addressed by changing the learning algorithm for the deep neural network to force the exploration of different network weights during a single training run that will result, in turn, with models that have differing performance. One way that this can be achieved is by aggressively changing the learning rate used during training. An approach to systematically and aggressively changing the learning rate during training to result in very different network weights is referred to as *Stochastic Gradient Descent with Warm Restarts* or SGDR for short, described by Ilya Loshchilov and Frank Hutter in their 2017 paper *SGDR: Stochastic Gradient Descent with Warm Restarts*.

Their approach involves systematically changing the learning rate over training epochs, called cosine annealing. This approach requires the specification of two hyperparameters: the initial learning rate and the total number of training epochs. The *cosine annealing* method has the effect of starting with a large learning rate that is relatively rapidly decreased to a minimum value before being dramatically increased again. The model weights are subjected to the dramatic changes during training, having the effect of using *good weights* as the starting point for the subsequent learning rate cycle, but allowing the learning algorithm to converge to a different solution.

The resetting of the learning rate acts like a simulated restart of the learning process and the re-use of good weights as the starting point of the restart is referred to as a *warm restart*, in contrast to a *cold restart* where a new set of small random numbers may be used as a starting point. The *good weights* at the bottom of each cycle can be saved to file, providing a snapshot of the model. These snapshots can be collected together at the end of the run and used in a model averaging ensemble. The saving and use of these models during an aggressive learning rate schedule is referred to as a *Snapshot Ensemble* and was described by Gao Huang, et al. in their 2017 paper titled *Snapshot Ensembles: Train 1, get M for free* and subsequently also used in an updated version of the Loshchilov and Hutter paper.

... we let SGD converge M times to local minima along its optimization path. Each time the model converges, we save the weights and add the corresponding network to our ensemble. We then restart the optimization with a large learning rate to escape the current local minimum.

— *Snapshot Ensembles: Train 1, get M for free*, 2017.

The ensemble of models is created during the course of training a single model, therefore, the authors claim that the ensemble forecast is provided at no additional cost.

[the approach allows] learning an ensemble of multiple neural networks without incurring any additional training costs.

— *Snapshot Ensembles: Train 1, get M for free*, 2017.

Although a cosine annealing schedule is used for the learning rate, other aggressive learning rate schedules could be used, such as the simpler cyclical learning rate schedule described by Leslie Smith in the 2017 paper titled *Cyclical Learning Rates for Training Neural Networks*. Now that we are familiar with the snapshot ensemble technique, we can look at how to implement it in Python with Keras.

24.2 Snapshot Ensembles Case Study

In this section, we will demonstrate how to use the snapshot ensemble to reduce the variance of an MLP on a simple multiclass classification problem. This example provides a template for applying the snapshot ensemble to your own neural network for classification and regression problems.

24.2.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate the snapshot ensemble. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. The problem can be configured to have two input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same data points.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

Listing 24.1: Example of creating samples for the blobs problem.

The result is the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can plot each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot for each class value
for class_value in range(3):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
```

```

    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()

```

Listing 24.2: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (separable by a line) causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions resulting in a high variance.

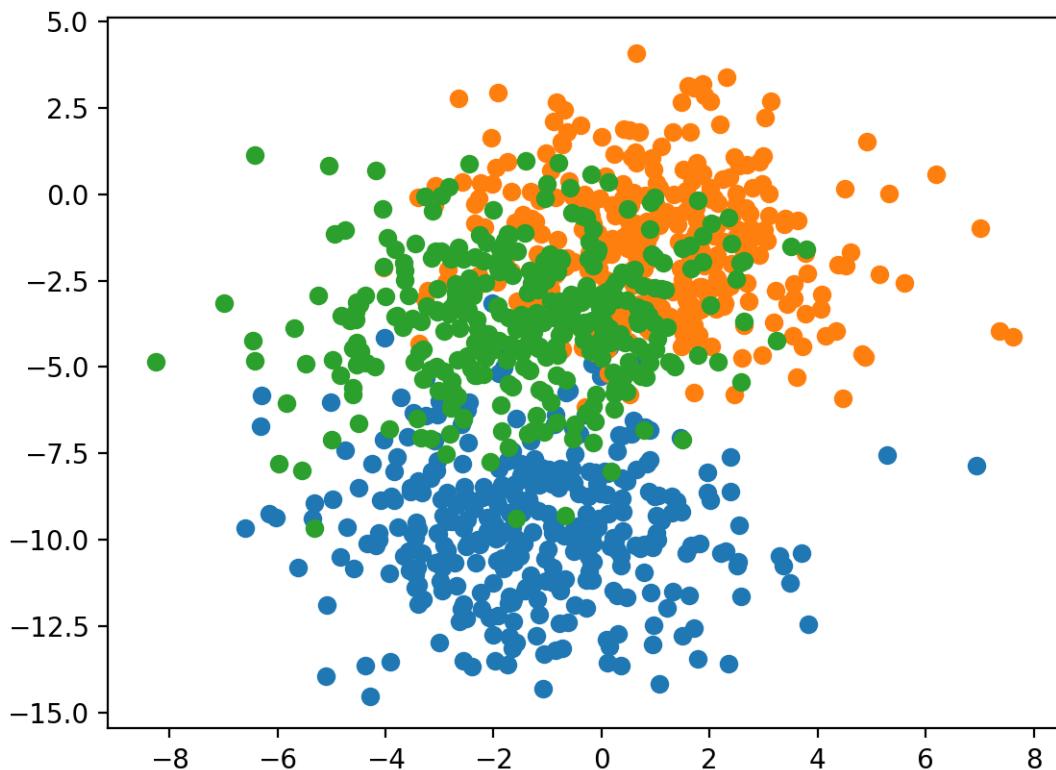


Figure 24.1: Scatter Plot of Blobs Dataset With Three Classes and Points Colored by Class Value.

24.2.2 Multilayer Perceptron Model

Before we define a model, we need to contrive a problem that is appropriate for the ensemble. In our problem, the training dataset is relatively small. Specifically, there is a 10:1 ratio of examples in the training dataset to the holdout dataset. This mimics a situation where we may have a vast number of unlabeled examples and a small number of labeled examples with which to train a model. We will create 1,100 data points from the blobs problem. The model will

be trained on the first 100 points and the remaining 1,000 will be held back in a test dataset, unavailable to the model.

The problem is a multiclass classification problem, and we will model it using a softmax activation function on the output layer. This means that the model will predict a vector with three elements with the probability that the sample belongs to each of the three classes. Therefore, we must one hot encode the class values before we split the rows into the train and test datasets. We can do this using the Keras `to_categorical()` function.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 24.3: Example of preparing dataset for modeling.

Next, we can define and compile the model. The model will expect samples with two input variables. The model then has a single hidden layer with 25 nodes and a rectified linear activation function, then an output layer with three nodes to predict the probability of each of the three classes and a softmax activation function. Because the problem is multiclass, we will use the categorical cross-entropy loss function to optimize the model and stochastic gradient descent with a small learning rate and momentum.

```
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 24.4: Example of defining the MLP model.

The model is fit for 200 training epochs and we will evaluate the model each epoch on the test set, using the test set as a validation set.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
```

Listing 24.5: Example of fitting the MLP model.

At the end of the run, we will evaluate the performance of the model on the train and test sets.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 24.6: Example of evaluate the MLP model.

Then finally, we will plot learning curves of the model accuracy over each training epoch on both the training and validation datasets.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 24.7: Example of plotting learning curves for the MLP.

Tying all of this together, the complete example is listed below.

```
# develop an mlp for blobs dataset
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=200, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
```

```
pyplot.legend()
pyplot.show()
```

Listing 24.8: Example of fitting an MLP on the blobs problem.

Running the example prints the performance of the final model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved about 82% accuracy on the training dataset, which we know is optimistic, and about 79% on the test dataset, which we would expect to be more realistic.

```
Train: 0.820, Test: 0.791
```

Listing 24.9: Example output fitting an MLP on the blobs problem.

A line plot is also created showing the learning curves for the model accuracy on the train and test sets over each training epoch. We can see that training accuracy is more optimistic over most of the run as we also noted with the final scores.

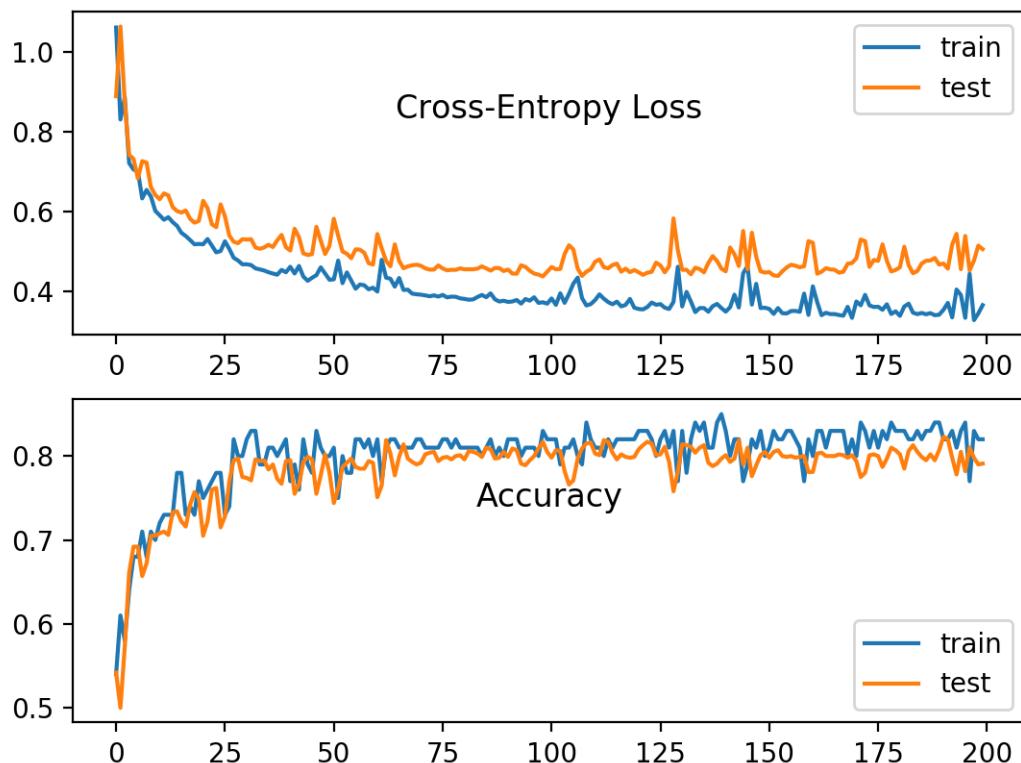


Figure 24.2: Line Plot Learning Curves of Model Accuracy on Train and Test Dataset over Each Training Epoch.

Next, we can look at how to implement an aggressive learning rate schedule.

24.2.3 Cosine Annealing Learning Rate

An effective snapshot ensemble requires training a neural network with an aggressive learning rate schedule. The cosine annealing schedule is an example of an aggressive learning rate schedule where learning rate starts high and is dropped relatively rapidly to a minimum value near zero before being increased again to the maximum. We can implement the schedule as described in the 2017 paper *Snapshot Ensembles: Train 1, get M for free*. The equation requires the total training epochs, maximum learning rate, and number of cycles as arguments as well as the current epoch number. The function then returns the learning rate for the given epoch.

$$\alpha(t) = \frac{\alpha_0}{2} \left(\cos \left(\frac{\pi \text{mod}(t - 1, \lceil T/M \rceil)}{\lceil T/M \rceil} \right) + 1 \right)$$

Figure 24.3: Equation for the Cosine Annealing Learning Rate Schedule.

Where $\alpha(t)$ is the learning rate at epoch t , α_0 is the maximum learning rate, T is the total epochs, M is the number of cycles, mod is the modulo operation, and square brackets indicate a floor operation. The function `cosine_annealing()` below implements the equation.

```
# cosine annealing learning rate schedule
def cosine_annealing(epoch, n_epochs, n_cycles, lrate_max):
    epochs_per_cycle = floor(n_epochs/n_cycles)
    cos_inner = (pi * (epoch % epochs_per_cycle)) / (epochs_per_cycle)
    return lrate_max/2 * (cos(cos_inner) + 1)
```

Listing 24.10: Example of a function for calculating the cosine annealing learning rate schedule.

We can test this implementation by plotting the learning rate over 100 epochs with five cycles (e.g. 20 epochs long) and a maximum learning rate of 0.01. The complete example is listed below.

```
# example of a cosine annealing learning rate schedule
from matplotlib import pyplot
from math import pi
from math import cos
from math import floor

# cosine annealing learning rate schedule
def cosine_annealing(epoch, n_epochs, n_cycles, lrate_max):
    epochs_per_cycle = floor(n_epochs/n_cycles)
    cos_inner = (pi * (epoch % epochs_per_cycle)) / (epochs_per_cycle)
    return lrate_max/2 * (cos(cos_inner) + 1)

# create learning rate series
n_epochs = 100
n_cycles = 5
lrate_max = 0.01
series = [cosine_annealing(i, n_epochs, n_cycles, lrate_max) for i in range(n_epochs)]
# plot series
pyplot.plot(series)
```

```
pyplot.show()
```

Listing 24.11: Example of plotting the cosine annealing learning rate schedule.

Running the example creates a line plot of the learning rate schedule over 100 epochs. We can see that the learning rate starts at the maximum value at epoch 0 and decreases rapidly to epoch 19, before being reset at epoch 20, the start of the next cycle. The cycle is repeated five times as specified in the argument.

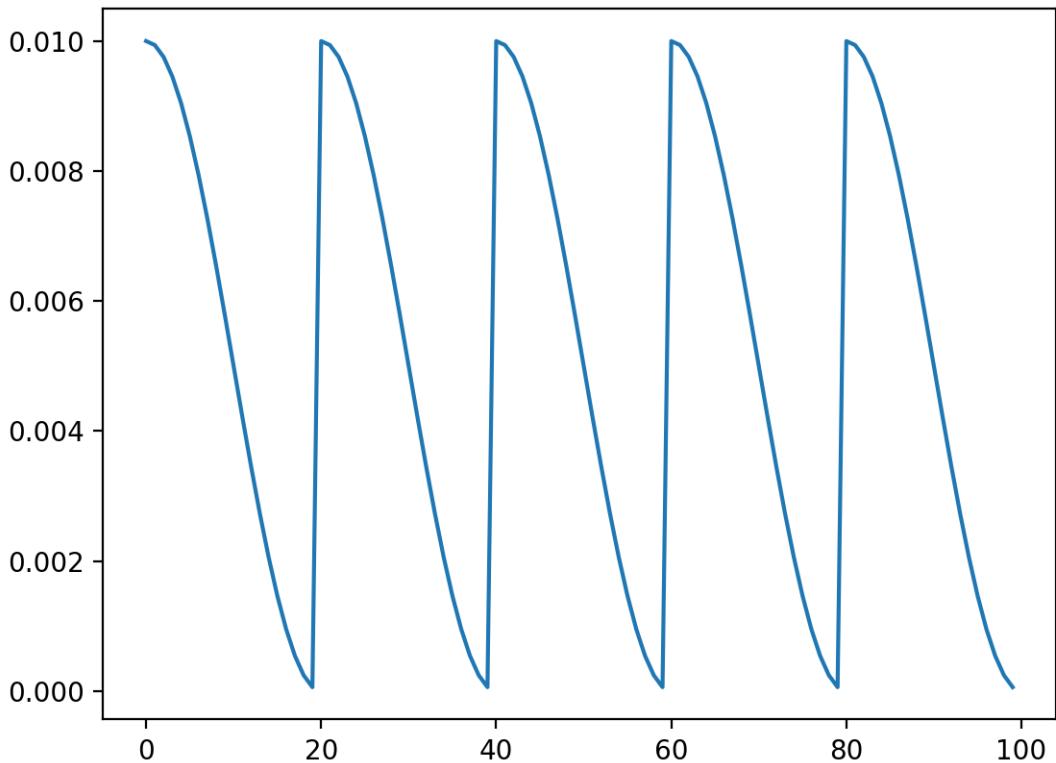


Figure 24.4: Line Plot of Cosine Annealing Learning Rate Schedule.

We can implement this schedule as a custom callback in Keras. This allows the parameters of the schedule to be specified and for the learning rate to be logged so we can ensure it had the desired effect. A custom callback can be defined as a Python class that extends the Keras `Callback` class. In the class constructor, we can take the required configuration as arguments and save them for use, specifically the total number of training epochs, the number of cycles for the learning rate schedule, and the maximum learning rate. We can use our `cosine_annealing()` defined above to calculate the learning rate for a given training epoch. The `Callback` class allows an `on_epoch_begin()` function to be overridden that will be called prior to each training epoch. We can override this function to calculate the learning rate for the current epoch and set it in the optimizer. We can also keep track of the learning rate in an internal list. The complete custom callback is defined below.

```

# define custom learning rate schedule
class CosineAnnealingLearningRateSchedule(Callback):
    # constructor
    def __init__(self, n_epochs, n_cycles, lrate_max, verbose=0):
        self.epochs = n_epochs
        self.cycles = n_cycles
        self.lr_max = lrate_max
        self.lrates = list()

    # calculate learning rate for an epoch
    def cosine_annealing(self, epoch, n_epochs, n_cycles, lrate_max):
        epochs_per_cycle = floor(n_epochs/n_cycles)
        cos_inner = (pi * (epoch % epochs_per_cycle)) / (epochs_per_cycle)
        return lrate_max/2 * (cos(cos_inner) + 1)

    # calculate and set learning rate at the start of the epoch
    def on_epoch_begin(self, epoch, logs=None):
        # calculate learning rate
        lr = self.cosine_annealing(epoch, self.epochs, self.cycles, self.lr_max)
        # set learning rate
        backend.set_value(self.model.optimizer.lr, lr)
        # log value
        self.lrates.append(lr)

```

Listing 24.12: Example of a Keras callback for the cosine annealing learning rate schedule.

We can create an instance of the callback and set the arguments. We will train the model for 400 epochs and set the number of cycles to be 50 epochs long, or $\frac{400}{50}$ cycles, a suggestion made and configuration used throughout the snapshot ensembles paper.

We lower the learning rate at a very fast pace, encouraging the model to converge towards its first local minimum after as few as 50 epochs.

— *Snapshot Ensembles: Train 1, get M for free*, 2017.

The paper also suggests that the learning rate can be set each sample or each minibatch instead of prior to each epoch to give more nuance to the updates, but we will leave this as a future exercise.

... we update the learning rate at each iteration rather than at every epoch. This improves the convergence of short cycles, even when a large initial learning rate is used.

— *Snapshot Ensembles: Train 1, get M for free*, 2017.

Once the callback is instantiated and configured, we can specify it as part of the list of callbacks to the call to the `fit()` function to train the model.

```

# define learning rate callback
n_epochs = 400
n_cycles = n_epochs / 50
ca = CosineAnnealingLearningRateSchedule(n_epochs, n_cycles, 0.01)
# fit model

```

```
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=n_epochs,
verbose=0, callbacks=[ca])
```

Listing 24.13: Example of a using the cosine annealing learning rate schedule callback.

At the end of the run, we can confirm that the learning rate schedule was performed by plotting the contents of the `lrates` list.

```
# plot learning rate
pyplot.plot(ca.lrates)
pyplot.show()
```

Listing 24.14: Example of a plotting the learning rates recorded during training.

Tying these elements together, the complete example of training an MLP on the blobs problem with a cosine annealing learning rate schedule is listed below.

```
# mlp with cosine annealing learning rate schedule on blobs problem
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import Callback
from keras.optimizers import SGD
from keras import backend
from math import pi
from math import cos
from math import floor
from matplotlib import pyplot

# define custom learning rate schedule
class CosineAnnealingLearningRateSchedule(Callback):
    # constructor
    def __init__(self, n_epochs, n_cycles, lrate_max, verbose=0):
        self.epochs = n_epochs
        self.cycles = n_cycles
        self.lrate_max = lrate_max
        self.lrates = list()

    # calculate learning rate for an epoch
    def cosine_annealing(self, epoch, n_epochs, n_cycles, lrate_max):
        epochs_per_cycle = floor(n_epochs/n_cycles)
        cos_inner = (pi * (epoch % epochs_per_cycle)) / (epochs_per_cycle)
        return lrate_max/2 * (cos(cos_inner) + 1)

    # calculate and set learning rate at the start of the epoch
    def on_epoch_begin(self, epoch, logs=None):
        # calculate learning rate
        lr = self.cosine_annealing(epoch, self.epochs, self.cycles, self.lrate_max)
        # set learning rate
        backend.set_value(self.model.optimizer.lr, lr)
        # log value
        self.lrates.append(lr)

    # generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
```

```

y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
opt = SGD(momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# define learning rate callback
n_epochs = 400
n_cycles = n_epochs / 50
ca = CosineAnnealingLearningRateSchedule(n_epochs, n_cycles, 0.01)
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=n_epochs,
                     verbose=0, callbacks=[ca])
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot learning rate
pyplot.plot(ca.lrates)
pyplot.show()
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()

```

Listing 24.15: Example of fitting an MLP with the cosine annealing learning rate schedule.

Running the example first reports the accuracy of the model on the training and test sets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we do not see much difference in the performance of the final model as compared to the previous section.

Train: 0.860, Test: 0.819

Listing 24.16: Example output fitting an MLP with the cosine annealing learning rate schedule.

A line plot of the learning rate schedule is created, showing eight cycles of 50 epochs each.

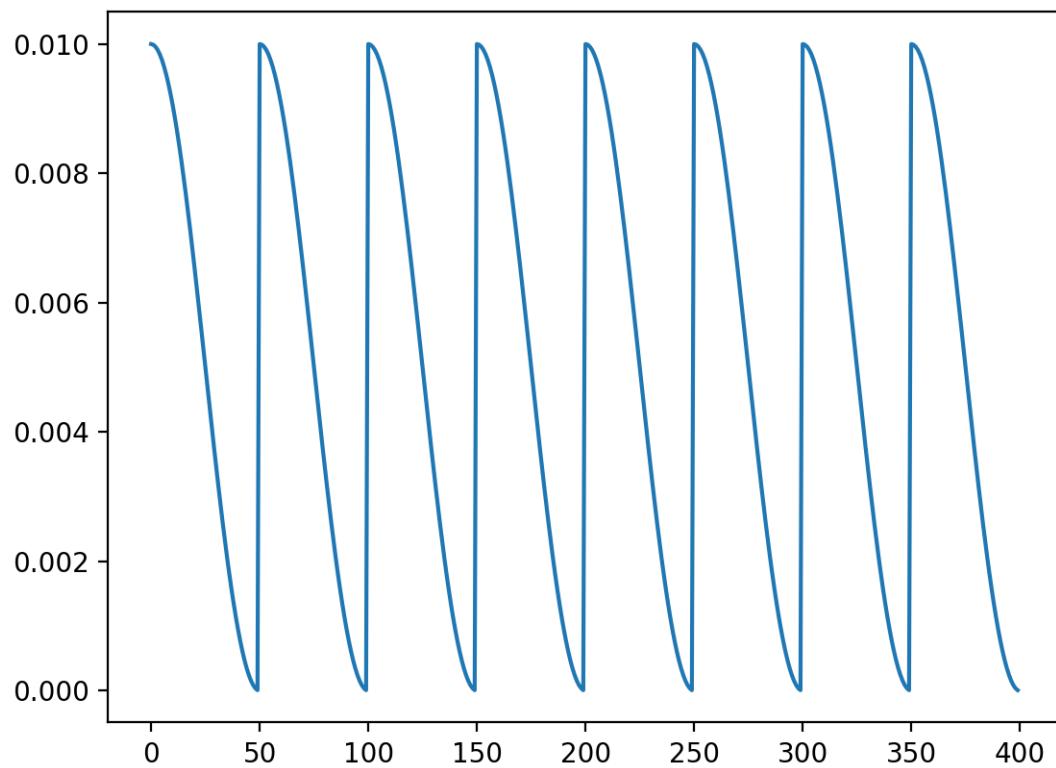


Figure 24.5: Cosine Annealing Learning Rate Schedule While Fitting an MLP on the Blobs Problem.

Finally, a line plot of model loss and accuracy on the train and test sets is created over each training epoch. We can see that although the learning rate was changed dramatically, there was not a dramatic effect on model performance, likely because the chosen classification problem is not very difficult.

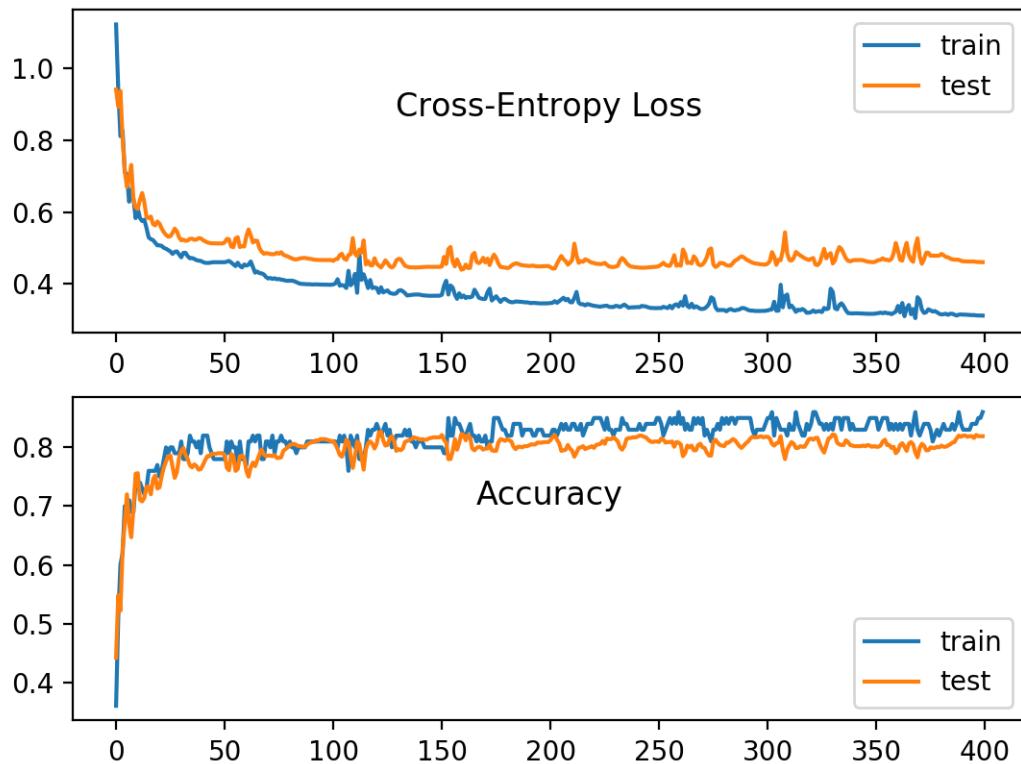


Figure 24.6: Line Plot of Train and Test Set Accuracy on the Blobs Dataset With a Cosine Annealing Learning Rate Schedule.

Now that we know how to implement the cosine annealing learning schedule, we can use it to prepare a snapshot ensemble.

24.2.4 MLP Snapshot Ensemble

We can develop a snapshot ensemble in two parts. The first part involves creating a custom callback to save the model at the bottom of each learning rate schedule. The second part involves loading the saved models and using them to make an ensemble prediction.

Save Snapshot Models During Training

The `CosineAnnealingLearningRateSchedule` can be updated to override the `on_epoch_end()` function called at the end of each training epoch. In this function, we can check if the current epoch that just ended was the end of a cycle. If so, we can save the model to file. Below is the updated callback, named the `SnapshotEnsemble` class. A debug message is printed each time a model is saved as confirmation that models are being saved at the right time. For example, with 50-epoch long cycles, we would expect a model to be saved on epoch 49, 99, etc. and the learning rate reset at epoch 50, 100, etc.

```

# snapshot ensemble with custom learning rate schedule
class SnapshotEnsemble(Callback):
    # constructor
    def __init__(self, n_epochs, n_cycles, lrate_max, verbose=0):
        self.epochs = n_epochs
        self.cycles = n_cycles
        self.lr_max = lrate_max
        self.lrates = list()

    # calculate learning rate for epoch
    def cosine_annealing(self, epoch, n_epochs, n_cycles, lrate_max):
        epochs_per_cycle = floor(n_epochs/n_cycles)
        cos_inner = (pi * (epoch % epochs_per_cycle)) / (epochs_per_cycle)
        return lrate_max/2 * (cos(cos_inner) + 1)

    # calculate and set learning rate at the start of the epoch
    def on_epoch_begin(self, epoch, logs={}):
        # calculate learning rate
        lr = self.cosine_annealing(epoch, self.epochs, self.cycles, self.lr_max)
        # set learning rate
        backend.set_value(self.model.optimizer.lr, lr)
        # log value
        self.lrates.append(lr)

    # save models at the end of each cycle
    def on_epoch_end(self, epoch, logs={}):
        # check if we can save model
        epochs_per_cycle = floor(self.epochs / self.cycles)
        if epoch != 0 and (epoch + 1) % epochs_per_cycle == 0:
            # save model to file
            filename = "snapshot_model_%d.h5" % int((epoch + 1) / epochs_per_cycle)
            self.model.save(filename)
            print('>saved snapshot %s, epoch %d' % (filename, epoch))

```

Listing 24.17: Example of a Keras callback for saving snapshot ensemble members.

We will train the model for 500 epochs, to give 10 models to choose from later when making an ensemble prediction. The complete example of using this new snapshot ensemble to save models to file is listed below.

```

# example of saving models for a snapshot ensemble
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import Callback
from keras.optimizers import SGD
from keras import backend
from math import pi
from math import cos
from math import floor

# snapshot ensemble with custom learning rate schedule
class SnapshotEnsemble(Callback):
    # constructor
    def __init__(self, n_epochs, n_cycles, lrate_max, verbose=0):

```

```

self.epochs = n_epochs
self.cycles = n_cycles
self.lr_max = lrate_max
self.lrates = list()

# calculate learning rate for epoch
def cosine_annealing(self, epoch, n_epochs, n_cycles, lrate_max):
    epochs_per_cycle = floor(n_epochs/n_cycles)
    cos_inner = (pi * (epoch % epochs_per_cycle)) / (epochs_per_cycle)
    return lrate_max/2 * (cos(cos_inner) + 1)

# calculate and set learning rate at the start of the epoch
def on_epoch_begin(self, epoch, logs={}):
    # calculate learning rate
    lr = self.cosine_annealing(epoch, self.epochs, self.cycles, self.lr_max)
    # set learning rate
    backend.set_value(self.model.optimizer.lr, lr)
    # log value
    self.lrates.append(lr)

# save models at the end of each cycle
def on_epoch_end(self, epoch, logs={}):
    # check if we can save model
    epochs_per_cycle = floor(self.epochs / self.cycles)
    if epoch != 0 and (epoch + 1) % epochs_per_cycle == 0:
        # save model to file
        filename = "snapshot_model_%d.h5" % int((epoch + 1) / epochs_per_cycle)
        self.model.save(filename)
        print('>saved snapshot %s, epoch %d' % (filename, epoch))

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(50, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
opt = SGD(momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# create snapshot ensemble callback
n_epochs = 500
n_cycles = n_epochs / 50
ca = SnapshotEnsemble(n_epochs, n_cycles, 0.01)
# fit model
model.fit(trainX, trainy, validation_data=(testX, testy), epochs=n_epochs, verbose=0,
          callbacks=[ca])

```

Listing 24.18: Example of saving snapshot ensemble members.

Running the example reports that 10 models were saved for the 10-ends of the cosine annealing learning rate schedule.

```
>saved snapshot snapshot_model_1.h5, epoch 49
>saved snapshot snapshot_model_2.h5, epoch 99
>saved snapshot snapshot_model_3.h5, epoch 149
>saved snapshot snapshot_model_4.h5, epoch 199
>saved snapshot snapshot_model_5.h5, epoch 249
>saved snapshot snapshot_model_6.h5, epoch 299
>saved snapshot snapshot_model_7.h5, epoch 349
>saved snapshot snapshot_model_8.h5, epoch 399
>saved snapshot snapshot_model_9.h5, epoch 449
>saved snapshot snapshot_model_10.h5, epoch 499
```

Listing 24.19: Example output from saving snapshot ensemble members.

Load Models and Make Ensemble Prediction

Once the snapshot models have been saved to file, they can be loaded and used to make an ensemble prediction. The first step is to load the models into memory. For large models, this could be done one model at a time, make a prediction, and move on to the next model before combining predictions. In this case, the models are relatively small and we can load all 10 from file as a list.

```
# load models from file
def load_all_models(n_models):
    all_models = list()
    for i in range(n_models):
        # define filename for this ensemble
        filename = 'snapshot_model_' + str(i + 1) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models
```

Listing 24.20: Example of loading saved snapshot ensemble members.

We would expect that models saved towards the end of the run may have better performance than models saved earlier in the run. As such, we can reverse the list of loaded models so that the older models are first.

```
# reverse loaded models so we build the ensemble with the last models first
members = list(reversed(members))
```

Listing 24.21: Example of reversing the order of saved snapshot ensemble members.

We don't know how many snapshots are required to make a good prediction for this problem. We can explore the effect of the number of ensemble members on test set accuracy by creating ensembles of increasing size starting with the final model at epoch 499, then adding the model saved at epoch 449, and so on until all 10 models are included. First, we require a function to make a prediction given a list of models. Given that each model predicts the probabilities of each of the output classes, we can sum the predicted probabilities across the models and select the class with the most support via the `argmax()` function. The `ensemble_predictions()` function below implements this functionality.

```
# make an ensemble prediction for multiclass classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result
```

Listing 24.22: Example of a function for making ensemble predictions.

We can then evaluate an ensemble of a given size by selecting the first n members from the list of models, making a prediction by calling the `ensemble_predictions()` function, and then calculating and returning the accuracy of the prediction. The `evaluate_n_members()` function below implements this behavior.

```
# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)
```

Listing 24.23: Example of a function for evaluating an ensemble of a given size.

The performance of each ensemble can also be contrasted with the performance of each standalone model and the average performance of all standalone models.

```
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, len(members)+1):
    # evaluate model with i members
    ensemble_score = evaluate_n_members(members, i, testX, testy)
    # evaluate the i'th model standalone
    testy_enc = to_categorical(testy)
    _, single_score = members[i-1].evaluate(testX, testy_enc, verbose=0)
    # summarize this step
    print('> %d: single=%f, ensemble=%f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# summarize average accuracy of a single final model
print('Accuracy %.3f (%.3f)' % (mean(single_scores), std(single_scores)))
```

Listing 24.24: Example of evaluating different sized snapshot ensembles.

Finally, we can plot the performance of each individual snapshot model (blue dots) compared to the performance of an ensemble that includes all models up to and including each individual model (orange line).

```
# plot score vs number of ensemble members
x_axis = [i for i in range(1, len(members)+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
```

```
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()
```

Listing 24.25: Example of plotting ensemble performance vs ensemble size.

The complete example of making snapshot ensemble predictions with different sized ensembles is listed below.

```
# load models and make a snapshot ensemble prediction
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score
from keras.utils import to_categorical
from keras.models import load_model
from matplotlib import pyplot
from numpy import mean
from numpy import std
from numpy import array
from numpy import argmax
import numpy

# load models from file
def load_all_models(n_models):
    all_models = list()
    for i in range(n_models):
        # define filename for this ensemble
        filename = 'snapshot_model_' + str(i + 1) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models

# make an ensemble prediction for multi-class classification
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = array(yhats)
    # sum across ensemble members
    summed = numpy.sum(yhats, axis=0)
    # argmax across classes
    result = argmax(summed, axis=1)
    return result

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # make prediction
    yhat = ensemble_predictions(subset, testX)
    # calculate accuracy
    return accuracy_score(testy, yhat)

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = 100
```

```

trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
print(trainX.shape, testX.shape)
# load models in order
members = load_all_models(10)
print('Loaded %d models' % len(members))
# reverse loaded models so we build the ensemble with the last models first
members = list(reversed(members))
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, len(members)+1):
    # evaluate model with i members
    ensemble_score = evaluate_n_members(members, i, testX, testy)
    # evaluate the i'th model standalone
    testy_enc = to_categorical(testy)
    _, single_score = members[i-1].evaluate(testX, testy_enc, verbose=0)
    # summarize this step
    print('> %d: single=%f, ensemble=%f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# summarize average accuracy of a single final model
print('Accuracy %f (%f)' % (mean(single_scores), std(single_scores)))
# plot score vs number of ensemble members
x_axis = [i for i in range(1, len(members)+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()

```

Listing 24.26: Example of loading and evaluating different sized snapshot ensembles.

Running the example first loads all 10 models into memory.

```

>loaded snapshot_model_1.h5
>loaded snapshot_model_2.h5
>loaded snapshot_model_3.h5
>loaded snapshot_model_4.h5
>loaded snapshot_model_5.h5
>loaded snapshot_model_6.h5
>loaded snapshot_model_7.h5
>loaded snapshot_model_8.h5
>loaded snapshot_model_9.h5
>loaded snapshot_model_10.h5
Loaded 10 models

```

Listing 24.27: Example output from loading saved snapshot ensemble members.

Next, each snapshot model is evaluated on the test dataset and the accuracy is reported. This is contrasted with the accuracy of a snapshot ensemble that includes all snapshot models working backward from the end of the run including the single model. The results show that as we work backward from the end of the run, the performance of the snapshot models gets worse, as we might expect.

Combining snapshot models into an ensemble shows that performance increases up to and including the last 3-to-5 models, reaching about 82%. This can be compared to the average performance of a snapshot model of about 80% test set accuracy.

Note: Your specific results may vary given the stochastic nature of the learning algorithm.

Consider running the example a few times and compare the average performance.

```
> 1: single=0.813, ensemble=0.813
> 2: single=0.814, ensemble=0.813
> 3: single=0.822, ensemble=0.822
> 4: single=0.810, ensemble=0.820
> 5: single=0.813, ensemble=0.818
> 6: single=0.811, ensemble=0.815
> 7: single=0.807, ensemble=0.813
> 8: single=0.805, ensemble=0.813
> 9: single=0.805, ensemble=0.813
> 10: single=0.790, ensemble=0.813
Accuracy 0.809 (0.008)
```

Listing 24.28: Example output from evaluating standalone models and snapshot ensembles of different sizes.

Finally, a line plot is created plotting the same test set accuracy scores. We can set the performance of each individual snapshot model as a blue dot and the snapshot ensemble of increasing size (number of members) from 1 to 10 members as an orange line. At least on this run, we can see that the snapshot ensemble quickly out-performs the final model at 82.2% with 3 members and all other saved models before performance degrades back down to about the same as the final model at 81.3%.

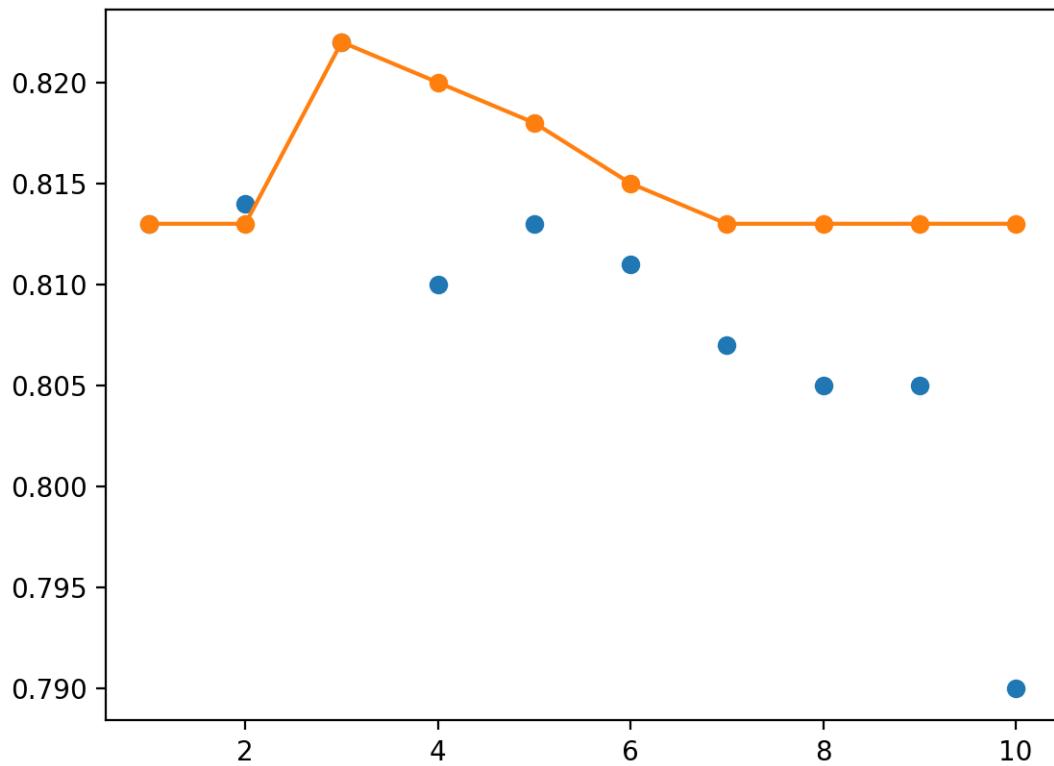


Figure 24.7: Line Plot of Single Snapshot Models (blue dots) vs Snapshot Ensembles of Varied Sized (orange line).

24.3 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Vary Cycle Length.** Update the example to use a shorter or longer cycle length and compare results.
- **Vary Maximum Learning Rate.** Update the example to use a larger or smaller maximum learning rate and compare results.
- **Update Learning Rate Per Batch.** Update the example to calculate the learning rate per-batch instead of per-epoch.
- **Repeated Evaluation.** Update the example to repeat the evaluation of the model to confirm that the approach indeed leads to an improved performance over the final model on the blobs problem.
- **Cyclic Learning Rate.** Update the example to use a cyclic learning rate schedule and compare results.

If you explore any of these extensions, I'd love to know.

24.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

24.4.1 Papers

- *Snapshot Ensembles: Train 1, get M for free*, 2017.
<https://arxiv.org/abs/1704.00109>
- *SGDR: Stochastic Gradient Descent with Warm Restarts*, 2017.
<https://arxiv.org/abs/1608.03983>
- *Cyclical Learning Rates for Training Neural Networks*, 2017.
<https://arxiv.org/abs/1506.01186>

24.4.2 APIs

- Getting started with the Keras Sequential model.
<https://keras.io/getting-started/sequential-model-guide/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- Keras Stochastic Gradient Descent API.
<https://keras.io/optimizers/#sgd>
- Keras Callbacks API.
<https://keras.io/callbacks/>
- numpy.argmax API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html>
- sklearn.datasets.make_blobs API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

24.5 Summary

In this tutorial, you discovered how to develop snapshot ensembles of models saved using an aggressive learning rate schedule over a single training run. Specifically, you learned:

- Snapshot ensembles combine the predictions from multiple models saved during a single training run.
- Diversity in model snapshots can be achieved through the use of aggressively cycling the learning rate used during a single training run.
- How to save model snapshots during a single run and load snapshot models to make ensemble predictions.

24.5.1 Next

In the next tutorial, you will discover how to train a new model to learn how to best combine the predictions from multiple ensemble members.

Chapter 25

Learn to Combine Predictions with Stacked Generalization Ensemble

Model averaging is an ensemble technique where multiple submodels contribute equally to a combined prediction. Model averaging can be improved by weighting the contributions of each submodel to the combined prediction by the expected performance of the submodel. This can be extended further by training an entirely new model to learn how to best combine the contributions from each submodel. This approach is called stacked generalization, or stacking for short, and can result in better predictive performance than any single contributing model. In this tutorial, you will discover how to develop a stacked generalization ensemble for deep learning neural networks. After completing this tutorial, you will know:

- Stacked generalization is an ensemble method where a new model learns how to best combine the predictions from multiple existing models.
- How to develop a stacking model using neural networks as a submodel and a scikit-learn classifier as the meta-learner.
- How to develop a stacking model where neural network submodels are embedded in a larger stacking ensemble model for training and prediction.

Let's get started.

25.1 Stacked Generalization Ensemble

A model averaging ensemble combines the predictions from multiple trained models. A limitation of this approach is that each model contributes the same amount to the ensemble prediction, regardless of how well the model performed. A variation of this approach, called a weighted average ensemble, weighs the contribution of each ensemble member by the trust or expected performance of the model on a holdout dataset. This allows well-performing models to contribute more and less-well-performing models to contribute less. The weighted average ensemble provides an improvement over the model average ensemble.

A further generalization of this approach is replacing the linear weighted sum (e.g. linear regression) model used to combine the predictions of the submodels with any learning algorithm. This approach is called stacked generalization, or stacking for short. In stacking, an algorithm

takes the outputs of submodels as input and attempts to learn how to best combine the input predictions to make a better output prediction. It may be helpful to think of the stacking procedure as having two levels: level 0 and level 1.

- **Level 0:** The level 0 data is the training dataset inputs and level 0 models learn to make predictions from this data.
- **Level 1:** The level 1 data takes the output of the level 0 models as input and the single level 1 model, or meta-learner, learns to make predictions from this data.

Stacked generalization works by deducing the biases of the generalizer(s) with respect to a provided learning set. This deduction proceeds by generalizing in a second space whose inputs are (for example) the guesses of the original generalizers when taught with part of the learning set and trying to guess the rest of it, and whose output is (for example) the correct guess.

— *Stacked Generalization*, 1992.

Unlike a weighted average ensemble, a stacked generalization ensemble can use the set of predictions as a context and conditionally decide to weigh the input predictions differently, potentially resulting in better performance. Interestingly, although stacking is described as an ensemble learning method with two or more level 0 models, it can be used in the case where there is only a single level 0 model. In this case, the level 1, or meta-learner, model learns to correct the predictions from the level 0 model.

... although it can also be used when one has only a single generalizer, as a technique to improve that single generalizer

— *Stacked Generalization*, 1992.

It is important that the meta-learner is trained on a separate dataset to the examples used to train the level 0 models to avoid overfitting. A simple way that this can be achieved is by splitting the training dataset into a train and validation set. The level 0 models are then trained on the train set. The level 1 model is then trained using the validation set, where the raw inputs are first fed through the level 0 models to get predictions that are used as inputs to the level 1 model. A limitation of the hold-out validation set approach to training a stacking model is that level 0 and level 1 models are not trained on the full dataset.

A more sophisticated approach to training a stacked model involves using k -fold cross-validation to develop the training dataset for the meta-learner model. Each level 0 model is trained using k -fold cross-validation (or even leave-one-out cross-validation for maximum effect); the models are then discarded, but the predictions are retained. This means for each model, there are predictions made by a version of the model that was not trained on those examples, e.g. like having holdout examples, but in this case for the entire training dataset. The predictions are then used as inputs to train the meta-learner. Level 0 models are then trained on the entire training dataset and together with the meta-learner, the stacked model can be used to make predictions on new data. In practice, it is common to use different algorithms to prepare each of the level 0 models, to provide a diverse set of predictions.

... stacking is not normally used to combine models of the same type [...] it is applied to models built by different learning algorithms.

— *Practical Machine Learning Tools and Techniques*, Second Edition, 2005.

It is also common to use a simple linear model to combine the predictions. Because use of a linear model is common, stacking is more recently referred to as *model blending* or simply *blending*, especially in machine learning competitions.

... the multi-response least squares linear regression technique should be employed as the high-level generalizer. This technique provides a method of combining level-0 models' confidence

— *Issues in Stacked Generalization*, 1999.

A stacked generalization ensemble can be developed for regression and classification problems. In the case of classification problems, better results have been seen when using the prediction of class probabilities as input to the meta-learner instead of class labels.

... class probabilities should be used instead of the single predicted class as input attributes for higher-level learning. The class probabilities serve as the confidence measure for the prediction made.

— *Issues in Stacked Generalization*, 1999.

Now that we are familiar with stacked generalization, we can work through a case study of developing a stacked deep learning model.

25.2 Stacked Generalization Ensemble Case Study

In this section, we will demonstrate how to use the stacking ensemble to reduce the variance of an MLP on a simple multiclass classification problem. This example provides a template for applying the stacking ensemble to your own neural network for classification and regression problems.

25.2.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate the stacking ensemble. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. The problem can be configured to have two input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same data points.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

Listing 25.1: Example of creating samples for the blobs problem.

The results are the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can graph each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot for each class value
for class_value in range(3):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 25.2: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (separable by a line) causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions, resulting in a high variance.

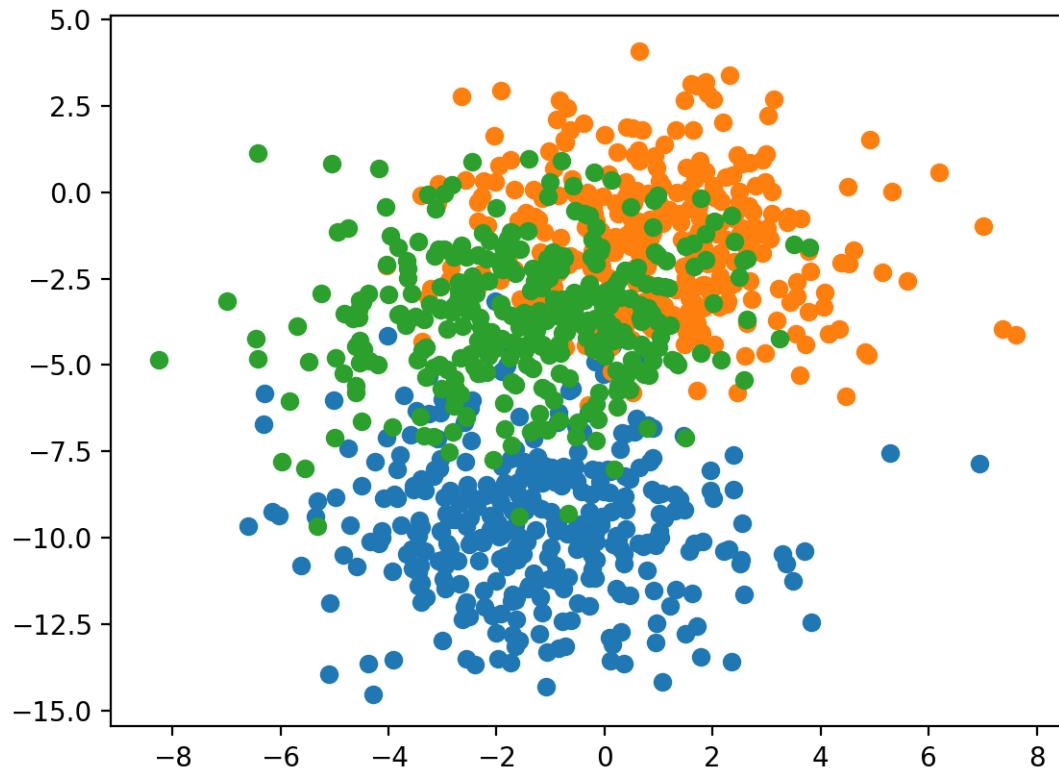


Figure 25.1: Scatter Plot of Blobs Dataset With Three Classes and Points Colored by Class Value.

25.2.2 Multilayer Perceptron Model

Before we define a model, we need to contrive a problem that is appropriate for the stacking ensemble. In our problem, the training dataset is relatively small. Specifically, there is a 10:1 ratio of examples in the training dataset to the holdout dataset. This mimics a situation where we may have a vast number of unlabeled examples and a small number of labeled examples with which to train a model. We will create 1,100 data points from the blobs problem. The model will be trained on the first 100 points and the remaining 1,000 will be held back in a test dataset, unavailable to the model.

The problem is a multiclass classification problem, and we will model it using a softmax activation function on the output layer. This means that the model will predict a vector with three elements with the probability that the sample belongs to each of the three classes. Therefore, we must one hot encode the class values before we split the rows into the train and test datasets. We can do this using the Keras `to_categorical()` function.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
```

```
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
print(trainX.shape, testX.shape)
```

Listing 25.3: Example of preparing the dataset for modeling.

Next, we can define and combine the model. The model will expect samples with two input variables. The model then has a single hidden layer with 25 nodes and a rectified linear activation function, then an output layer with three nodes to predict the probability of each of the three classes and a softmax activation function. Because the problem is multiclass, we will use the categorical cross-entropy loss function to optimize the model and the efficient Adam flavor of stochastic gradient descent.

```
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 25.4: Example of defining the MLP model.

The model is fit for 500 training epochs and we will evaluate the model each epoch on the test set, using the test set as a validation set.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=500, verbose=0)
```

Listing 25.5: Example of fitting the MLP model.

At the end of the run, we will evaluate the performance of the model on the train and test sets.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 25.6: Example of evaluating the MLP model.

Then finally, we will plot learning curves of the model accuracy over each training epoch on both the training and validation datasets.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 25.7: Example of plotting learning curves for the MLP model.

Tying all of this together, the complete example is listed below.

```
# develop an mlp for blobs dataset
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=500, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 25.8: Example of fitting an MLP on the blobs problem.

Running the example first prints the shape of each dataset for confirmation, then the performance of the final model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved about 83% accuracy on the training dataset, which we know is optimistic, and about 82% on the test dataset, which we would expect to be more realistic.

Train: 0.830, Test: 0.821

Listing 25.9: Example output fitting an MLP on the blobs problem.

A line plot is also created showing the learning curves for the model accuracy on the train and test sets over each training epoch. We can see that training accuracy is more optimistic over most of the run as we also noted with the final scores.

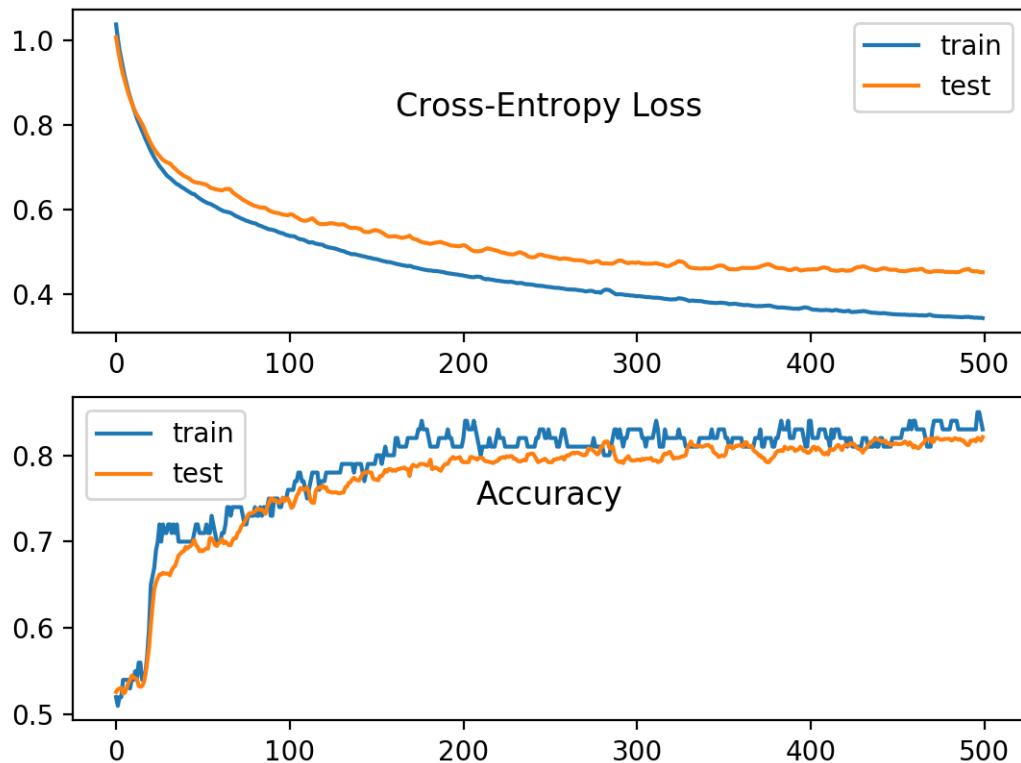


Figure 25.2: Line Plot Learning Curves of Model Accuracy on Train and Test Dataset Over Each Training Epoch.

We can now look at using instances of this model as part of a stacking ensemble.

25.2.3 Train and Save Sub-Models

To keep this example simple, we will use multiple instances of the same model as level-0 or submodels in the stacking ensemble. We will also use a holdout validation dataset to train the level-1 or meta-learner in the ensemble. A more advanced example may use different types of MLP models (deeper, wider, etc.) as submodels and train the meta-learner using k -fold cross-validation. In this section, we will train multiple submodels and save them to file for later use in our stacking ensembles. The first step is to create a function that will define and fit an MLP model on the training dataset.

```
# fit model on dataset
def fit_model(trainX, trainy):
    # define model
    model = Sequential()
```

```

model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
model.fit(trainX, trainy, epochs=500, verbose=0)
return model

```

Listing 25.10: Example of a function for fitting and returning an MLP model.

Next, we can create a sub-directory to store the models. Note, if the directory already exists, you may have to delete it when re-running this code.

```

# create directory for models
makedirs('models')

```

Listing 25.11: Example creating a folder to store saved models.

Finally, we can create multiple instances of the MLP and save each to the `models/` subdirectory with a unique filename. In this case, we will create five submodels, but you can experiment with a different number of models and see how it impacts model performance.

```

# fit and save models
n_members = 5
for i in range(n_members):
    # fit model
    model = fit_model(trainX, trainy)
    # save model
    filename = 'models/model_' + str(i + 1) + '.h5'
    model.save(filename)
    print('>Saved %s' % filename)

```

Listing 25.12: Example fitting and saving ensemble members.

We can tie all of these elements together; the complete example of training the submodels and saving them to file is listed below.

```

# example of saving sub-models for later use in a stacking ensemble
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from os import makedirs

# fit model on dataset
def fit_model(trainX, trainy):
    # define model
    model = Sequential()
    model.add(Dense(25, input_dim=2, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit model
    model.fit(trainX, trainy, epochs=500, verbose=0)
    return model

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable

```

```

y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# create directory for models
makedirs('models')
# fit and save models
n_members = 5
for i in range(n_members):
    # fit model
    model = fit_model(trainX, trainy)
    # save model
    filename = 'models/model_' + str(i + 1) + '.h5'
    model.save(filename)
    print('>Saved %s' % filename)

```

Listing 25.13: Example of fitting and saving MLP ensemble members.

Running the example creates the `models/` subfolder and saves five trained models with unique filenames.

```

>Saved models/model_1.h5
>Saved models/model_2.h5
>Saved models/model_3.h5
>Saved models/model_4.h5
>Saved models/model_5.h5

```

Listing 25.14: Example output from fitting and saving MLP ensemble members.

Next, we can look at training a meta-learner to make best use of the predictions from these submodels.

25.2.4 Separate Stacking Model

We can now train a meta-learner that will best combine the predictions from the submodels and ideally perform better than any single submodel. The first step is to load the saved models. We can use the `load_model()` Keras function and create a Python list of loaded models.

```

# load models from file
def load_all_models(n_models):
    all_models = list()
    for i in range(n_models):
        # define filename for this ensemble
        filename = 'models/model_' + str(i + 1) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models

```

Listing 25.15: Example of a function for loading saved ensemble members.

We can call this function to load our five saved models from the `models/` sub-directory.

```
# load all models
n_members = 5
members = load_all_models(n_members)
print('Loaded %d models' % len(members))
```

Listing 25.16: Example loading ensemble members.

It would be useful to know how well the single models perform on the test dataset as we would expect a stacking model to perform better. We can easily evaluate each single model on the training dataset and establish a baseline of performance.

```
# evaluate standalone models on test dataset
for model in members:
    testy_enc = to_categorical(testy)
    _, acc = model.evaluate(testX, testy_enc, verbose=0)
    print('Model Accuracy: %.3f' % acc)
```

Listing 25.17: Example evaluating standalone model performance.

Next, we can train our meta-learner. This requires two steps:

- Prepare a training dataset for the meta-learner.
- Use the prepared training dataset to fit a meta-learner model.

We will prepare a training dataset for the meta-learner by providing examples from the test set to each of the submodels and collecting the predictions. In this case, each model will output three predictions for each example for the probabilities that a given example belongs to each of the three classes. Therefore, the 1,000 examples in the test set will result in five arrays with the shape [1000, 3]. We can combine these arrays into a three-dimensional array with the shape [1000, 5, 3] by using the `dstack()` NumPy function that will stack each new set of predictions.

As input for a new model, we will require 1,000 examples with some number of features. Given that we have five models and each model makes three predictions per example, then we would have 15 (3×5) features for each example provided to the submodels. We can transform the [1000, 5, 3] shaped predictions from the submodels into a [1000, 15] shaped array to be used to train a meta-learner using the `reshape()` NumPy function and flattening the final two dimensions. The `stacked_dataset()` function implements this step.

```
# create stacked model input dataset as outputs from the ensemble
def stacked_dataset(members, inputX):
    stackX = None
    for model in members:
        # make prediction
        yhat = model.predict(inputX, verbose=0)
        # stack predictions into [rows, members, probabilities]
        if stackX is None:
            stackX = yhat
        else:
            stackX = dstack((stackX, yhat))
    # flatten predictions to [rows, members x probabilities]
    stackX = stackX.reshape((stackX.shape[0], stackX.shape[1]*stackX.shape[2]))
    return stackX
```

Listing 25.18: Example of a function for creating a stacking dataset.

Once prepared, we can use this input dataset along with the output, or y part, of the test set to train a new meta-learner. In this case, we will train a simple logistic regression algorithm from the scikit-learn library. Logistic regression only supports binary classification, although the implementation of logistic regression in scikit-learn in the `LogisticRegression` class supports multiclass classification (more than two classes) using a multinomial scheme. The function `fit_stacked_model()` below will prepare the training dataset for the meta-learner by calling the `stacked_dataset()` function, then fit a logistic regression model that is then returned.

```
# fit a model based on the outputs from the ensemble members
def fit_stacked_model(members, inputX, inputy):
    # create dataset using ensemble
    stackedX = stacked_dataset(members, inputX)
    # fit standalone model
    model = LogisticRegression(solver='lbfgs', multi_class='multinomial')
    model.fit(stackedX, inputy)
    return model
```

Listing 25.19: Example of a function for fitting a stacked model.

We can call this function and pass in the list of loaded models and the training dataset.

```
# fit stacked model using the ensemble
model = fit_stacked_model(members, testX, testy)
```

Listing 25.20: Example of fitting a stacked model.

Once fit, we can use the stacked model, including the members and the meta-learner, to make predictions on new data. This can be achieved by first using the submodels to make an input dataset for the meta-learner, e.g. by calling the `stacked_dataset()` function, then making a prediction with the meta-learner. The `stacked_prediction()` function below implements this.

```
# make a prediction with the stacked model
def stacked_prediction(members, model, inputX):
    # create dataset using ensemble
    stackedX = stacked_dataset(members, inputX)
    # make a prediction
    yhat = model.predict(stackedX)
    return yhat
```

Listing 25.21: Example of a function for making a stacked prediction.

We can use this function to make a prediction on new data; in this case, we can demonstrate it by making predictions on the test set.

```
# evaluate model on test set
yhat = stacked_prediction(members, model, testX)
acc = accuracy_score(testy, yhat)
print('Stacked Test Accuracy: %.3f' % acc)
```

Listing 25.22: Example of making a stacked prediction.

Tying all of these elements together, the complete example of fitting a linear meta-learner for the stacking ensemble of MLP submodels is listed below.

```
# stacked generalization with linear meta model on blobs dataset
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score
```

```
from sklearn.linear_model import LogisticRegression
from keras.models import load_model
from keras.utils import to_categorical
from numpy import dstack

# load models from file
def load_all_models(n_models):
    all_models = list()
    for i in range(n_models):
        # define filename for this ensemble
        filename = 'models/model_' + str(i + 1) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models

# create stacked model input dataset as outputs from the ensemble
def stacked_dataset(members, inputX):
    stackX = None
    for model in members:
        # make prediction
        yhat = model.predict(inputX, verbose=0)
        # stack predictions into [rows, members, probabilities]
        if stackX is None:
            stackX = yhat
        else:
            stackX = dstack((stackX, yhat))
    # flatten predictions to [rows, members x probabilities]
    stackX = stackX.reshape((stackX.shape[0], stackX.shape[1]*stackX.shape[2]))
    return stackX

# fit a model based on the outputs from the ensemble members
def fit_stacked_model(members, inputX, inputy):
    # create dataset using ensemble
    stackedX = stacked_dataset(members, inputX)
    # fit standalone model
    model = LogisticRegression(solver='lbfgs', multi_class='multinomial')
    model.fit(stackedX, inputy)
    return model

# make a prediction with the stacked model
def stacked_prediction(members, model, inputX):
    # create dataset using ensemble
    stackedX = stacked_dataset(members, inputX)
    # make a prediction
    yhat = model.predict(stackedX)
    return yhat

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

```
# load all models
n_members = 5
members = load_all_models(n_members)
print('Loaded %d models' % len(members))
# evaluate standalone models on test dataset
for model in members:
    testy_enc = to_categorical(testy)
    _, acc = model.evaluate(testX, testy_enc, verbose=0)
    print('Model Accuracy: %.3f' % acc)
# fit stacked model using the ensemble
model = fit_stacked_model(members, testX, testy)
# evaluate model on test set
yhat = stacked_prediction(members, model, testX)
acc = accuracy_score(testy, yhat)
print('Stacked Test Accuracy: %.3f' % acc)
```

Listing 25.23: Example of fitting a logistic regression stacking model.

Running the example first loads the submodels into a list and evaluates the performance of each. We can see that the best performing model is the final model with an accuracy of about 81.3%.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
>loaded models/model_1.h5
>loaded models/model_2.h5
>loaded models/model_3.h5
>loaded models/model_4.h5
>loaded models/model_5.h5
Loaded 5 models
Model Accuracy: 0.805
Model Accuracy: 0.806
Model Accuracy: 0.804
Model Accuracy: 0.809
Model Accuracy: 0.813
```

Listing 25.24: Example output from loading and evaluating standalone MLP models.

Next, a logistic regression meta-learner is trained on the predicted probabilities from each submodel on the test set, then the entire stacking model is evaluated on the test set. We can see that in this case, the meta-learner out-performed each of the submodels on the test set, achieving an accuracy of about 82.4%.

```
Stacked Test Accuracy: 0.824
```

Listing 25.25: Example output from evaluating the stacking model.

25.2.5 Integrated Stacking Model

When using neural networks as submodels, it may be desirable to use a neural network as a meta-learner. Specifically, the sub-networks can be embedded in a larger multi-headed neural network that then learns how to best combine the predictions from each input submodel. It

allows the stacking ensemble to be treated as a single large model. The benefit of this approach is that the outputs of the submodels are provided directly to the meta-learner. Further, it is also possible to update the weights of the submodels in conjunction with the meta-learner model, if this is desirable. This can be achieved using the Keras functional interface for developing models.

After the models are loaded as a list, a larger stacking ensemble model can be defined where each of the loaded models is used as a separate input-head to the model. This requires that all of the layers in each of the loaded models be marked as not trainable so the weights cannot be updated when the new larger model is being trained. Keras also requires that each layer has a unique name, therefore the names of each layer in each of the loaded models will have to be updated to indicate to which ensemble member they belong.

```
# update all layers in all models to not be trainable
for i in range(len(members)):
    model = members[i]
    for layer in model.layers:
        # make not trainable
        layer.trainable = False
        # rename to avoid 'unique layer name' issue
        layer._name = 'ensemble_' + str(i+1) + '_' + layer.name
```

Listing 25.26: Example of updating layer names and layers to be non-trainable.

Once the submodels have been prepared, we can define the stacking ensemble model. The input layer for each of the submodels will be used as a separate input head to this new model. This means that k copies of any input data will have to be provided to the model, where k is the number of input models, in this case, 5. The outputs of each of the models can then be merged. In this case, we will use a simple concatenation merge, where a single 15-element vector will be created from the three class-probabilities predicted by each of the 5 models. We will then define a hidden layer to interpret this *input* to the meta-learner and an output layer that will make its own probabilistic prediction. The `define_stacked_model()` function below implements this and will return a stacked generalization neural network model given a list of trained submodels.

```
# define stacked model from multiple member input models
def define_stacked_model(members):
    # update all layers in all models to not be trainable
    for i in range(len(members)):
        model = members[i]
        for layer in model.layers:
            # make not trainable
            layer.trainable = False
            # rename to avoid 'unique layer name' issue
            layer._name = 'ensemble_' + str(i+1) + '_' + layer.name
    # define multi-headed input
    ensemble_visible = [model.input for model in members]
    # concatenate merge output from each model
    ensemble_outputs = [model.output for model in members]
    merge = concatenate(ensemble_outputs)
    hidden = Dense(10, activation='relu')(merge)
    output = Dense(3, activation='softmax')(hidden)
    model = Model(inputs=ensemble_visible, outputs=output)
    # plot graph of ensemble
    plot_model(model, show_shapes=True, to_file='model_graph.png')
    # compile
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
return model
```

Listing 25.27: Example of a function for defining a stacked MLP model.

A plot of the network graph is created when this function is called to give an idea of how the ensemble model fits together.

```
# define ensemble model
stacked_model = define_stacked_model(members)
```

Listing 25.28: Example of defining a stacked MLP model.

Creating the plot requires that `pygraphviz` is installed. If this is a challenge on your workstation, you can comment out the call to the `plot_model()` function.

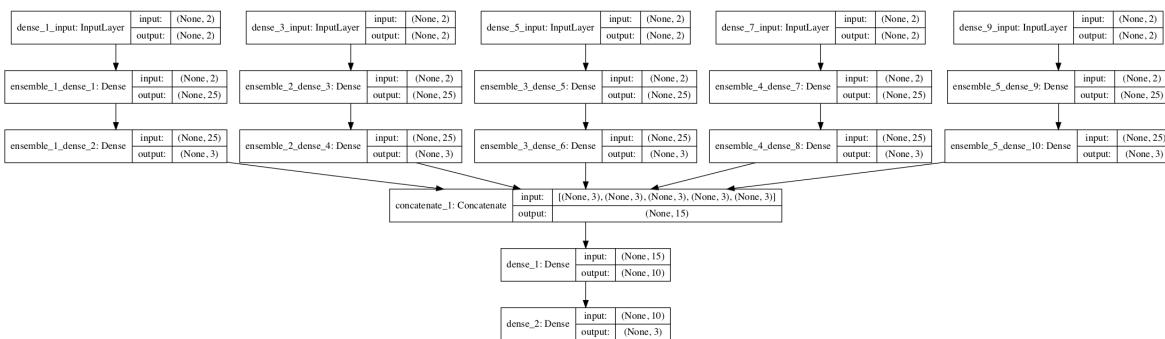


Figure 25.3: Visualization of Stacked Generalization Ensemble of Neural Network Models.

Once the model is defined, it can be fit. We can fit it directly on the holdout test dataset. Because the submodels are not trainable, their weights will not be updated during training and only the weights of the new hidden and output layer will be updated. The `fit_stacked_model()` function below will fit the stacking neural network model on for 300 epochs.

```
# fit a stacked model
def fit_stacked_model(model, inputX, inputy):
    # prepare input data
    X = [inputX for _ in range(len(model.input))]
    # encode output data
    inputy_enc = to_categorical(inputy)
    # fit model
    model.fit(X, inputy_enc, epochs=300, verbose=0)
```

Listing 25.29: Example of a function for fitting the stacked MLP model.

We can call this function providing the defined stacking model and the test dataset.

```
# fit stacked model on test dataset
fit_stacked_model(stacked_model, testX, testy)
```

Listing 25.30: Example of fitting the stacked MLP model.

Once fit, we can use the new stacked model to make a prediction on new data. This is as simple as calling the `predict()` function on the model. One minor change is that we require k copies of the input data in a list to be provided to the model for each of the k submodels. The `predict_stacked_model()` function below simplifies this process of making a prediction with the stacking model.

```
# make a prediction with a stacked model
def predict_stacked_model(model, inputX):
    # prepare input data
    X = [inputX for _ in range(len(model.input))]
    # make prediction
    return model.predict(X, verbose=0)
```

Listing 25.31: Example of a function for making a prediction with a stacked MLP model.

We can call this function to make a prediction for the test dataset and report the accuracy. We would expect the performance of the neural network learner to be better than any individual submodel and perhaps competitive with the linear meta-learner used in the previous section.

```
# make predictions and evaluate
yhat = predict_stacked_model(stacked_model, testX)
yhat = argmax(yhat, axis=1)
acc = accuracy_score(testy, yhat)
print('Stacked Test Accuracy: %.3f' % acc)
```

Listing 25.32: Example of making a prediction with a stacked MLP model.

Tying all of these elements together, the complete example is listed below.

```
# stacked generalization with neural net meta model on blobs dataset
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score
from keras.models import load_model
from keras.utils import to_categorical
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Dense
from keras.layers.merge import concatenate
from numpy import argmax

# load models from file
def load_all_models(n_models):
    all_models = list()
    for i in range(n_models):
        # define filename for this ensemble
        filename = 'models/model_' + str(i + 1) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models

# define stacked model from multiple member input models
def define_stacked_model(members):
    # update all layers in all models to not be trainable
    for i in range(len(members)):
        model = members[i]
        for layer in model.layers:
            # make not trainable
            layer.trainable = False
            # rename to avoid 'unique layer name' issue
```

```

layer._name = 'ensemble_' + str(i+1) + '_' + layer.name
# define multi-headed input
ensemble_visible = [model.input for model in members]
# concatenate merge output from each model
ensemble_outputs = [model.output for model in members]
merge = concatenate(ensemble_outputs)
hidden = Dense(10, activation='relu')(merge)
output = Dense(3, activation='softmax')(hidden)
model = Model(inputs=ensemble_visible, outputs=output)
# plot graph of ensemble
plot_model(model, show_shapes=True, to_file='model_graph.png')
# compile
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
return model

# fit a stacked model
def fit_stacked_model(model, inputX, inputy):
    # prepare input data
    X = [inputX for _ in range(len(model.input))]
    # encode output data
    inputy_enc = to_categorical(inputy)
    # fit model
    model.fit(X, inputy_enc, epochs=300, verbose=0)

# make a prediction with a stacked model
def predict_stacked_model(model, inputX):
    # prepare input data
    X = [inputX for _ in range(len(model.input))]
    # make prediction
    return model.predict(X, verbose=0)

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# load all models
n_members = 5
members = load_all_models(n_members)
print('Loaded %d models' % len(members))
# define ensemble model
stacked_model = define_stacked_model(members)
# fit stacked model on test dataset
fit_stacked_model(stacked_model, testX, testy)
# make predictions and evaluate
yhat = predict_stacked_model(stacked_model, testX)
yhat = argmax(yhat, axis=1)
acc = accuracy_score(testy, yhat)
print('Stacked Test Accuracy: %.3f' % acc)

```

Listing 25.33: Example of fitting an MLP stacking model.

Running the example first loads the five submodels.

Note: Your specific results may vary given the stochastic nature of the learning algorithm.

Consider running the example a few times and compare the average performance.

A larger stacking ensemble neural network is defined and fit on the test dataset, then the new model is used to make a prediction on the test dataset. We can see that, in this case, the model achieved an accuracy of about 83.3%, out-performing the linear model from the previous section.

```
>loaded models/model_1.h5  
>loaded models/model_2.h5  
>loaded models/model_3.h5  
>loaded models/model_4.h5  
>loaded models/model_5.h5  
Loaded 5 models  
Stacked Test Accuracy: 0.833
```

Listing 25.34: Example output from evaluating a stacking MLP model.

25.3 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Alternate Meta-Learner.** Update the example to use an alternate meta-learner classifier model to the logistic regression model.
- **Single Level 0 Model.** Update the example to use a single level-0 model and compare the results.
- **Vary Level 0 Models.** Develop a study that demonstrates the relationship between test classification accuracy and the number of submodels used in the stacked ensemble.
- **Cross-Validation Stacking Ensemble.** Update the example to use k -fold cross-validation to prepare the training dataset for the meta-learner model.
- **Use Raw Input in Meta-Learner.** Update the example so that the meta-learner algorithms take the raw input data for the sample as well as the output from the submodels and compare performance.

If you explore any of these extensions, I'd love to know.

25.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

25.4.1 Books

- Section 8.8 Model Averaging and Stacking, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Second Edition, 2016.
<https://amzn.to/2DYeHAO>

- Section 7.5 Combining multiple models, *Data Mining: Practical Machine Learning Tools and Techniques*, Second Edition, 2005.
<https://amzn.to/2pBxIPN>
- Section 9.8.2 Stacked Generalization, *Neural Networks for Pattern Recognition*, 1995.
<https://amzn.to/2pAQAOR>

25.4.2 Papers

- *Stacked generalization*, 1992.
<https://www.sciencedirect.com/science/article/pii/S0893608005800231>
- *Issues in Stacked Generalization*, 1999.
<https://www.jair.org/index.php/jair/article/view/10228>

25.4.3 APIs

- Getting started with the Keras Sequential model.
<https://keras.io/getting-started/sequential-model-guide/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- numpy.argmax API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html>
- sklearn.datasets.make_blobs API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
- numpy.dstack API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.dstack.html>
- sklearn.linear_model.LogisticRegression API.
http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

25.4.4 Articles

- Stacked Generalization (Stacking) Bibliography.
<http://machine-learning.martinsewell.com/ensembles/stacking/>
- Ensemble learning, Wikipedia.
https://en.wikipedia.org/wiki/Ensemble_learning

25.5 Summary

In this tutorial, you discovered how to develop a stacked generalization ensemble for deep learning neural networks. Specifically, you learned:

- Stacked generalization is an ensemble method where a new model learns how to best combine the predictions from multiple existing models.
- How to develop a stacking model using neural networks as a submodel and a scikit-learn classifier as the meta-learner.
- How to develop a stacking model where neural network submodels are embedded in a larger stacking ensemble model for training and prediction.

25.5.1 Next

In the next tutorial, you will discover how to combine the model parameters or weights from multiple ensemble members, instead of their predictions.

Chapter 26

Combine Model Parameters with Average Model Weights Ensemble

The training process of neural networks is a challenging optimization process that can often fail to converge. This can mean that the model at the end of training may not be a stable or best-performing set of weights to use as a final model. One approach to address this problem is to use an average of the weights from multiple models seen toward the end of the training run. This is called Polyak-Ruppert averaging and can be further improved by using a linearly or exponentially decreasing weighted average of the model weights. In addition to resulting in a more stable model, the performance of the averaged model weights can also result in better performance. In this tutorial, you will discover how to combine the weights from multiple different models into a single model for making predictions. After completing this tutorial, you will know:

- The stochastic and challenging nature of training neural networks can mean that the optimization process does not converge.
- Creating a model with the average of the weights from models observed towards the end of a training run can result in a more stable and sometimes better-performing solution.
- How to develop final models created with the equal, linearly, and exponentially weighted average of model parameters from multiple saved models.

Let's get started.

26.1 Average Model Weight Ensemble

Learning the weights for a deep neural network model requires solving a high-dimensional non-convex optimization problem. A challenge with solving this optimization problem is that there are many *good* solutions and it is possible for the learning algorithm to bounce around and fail to settle in on one. In the area of stochastic optimization, this is referred to as problems with the convergence of the optimization algorithm on a solution, where a solution is defined by a set of specific weight values.

A symptom you may see if you have a problem with the convergence of your model is train and/or test loss value that shows higher than expected variance, e.g. it thrashes or bounces

up and down over training epochs. One approach to address this problem is to combine the weights collected towards the end of the training process. Generally, this might be referred to as temporal averaging and is known as Polyak Averaging or Polyak-Ruppert averaging, named for the original developers of the method.

Polyak averaging consists of averaging together several points in the trajectory through parameter space visited by an optimization algorithm.

— Page 322, *Deep Learning*, 2016.

Averaging multiple noisy sets of weights during the learning process may paradoxically sound less desirable than tuning the optimization process itself, but may prove a desirable solution, especially for very large neural networks that may take days, weeks, or even months to train.

The essential advancement was reached on the basis of the paradoxical idea: a slow algorithm having less than optimal convergence rate must be averaged.

— *Acceleration of Stochastic Approximation by Averaging*, 1992.

Averaging the weights of multiple models from a single training run has the effect of calming down the noisy optimization process that may be noisy because of the choice of learning hyperparameters (e.g. learning rate) or the shape of the mapping function that is being learned. The result is a final model or set of weights that may offer a more stable, and perhaps more accurate result.

The basic idea is that the optimization algorithm may leap back and forth across a valley several times without ever visiting a point near the bottom of the valley. The average of all of the locations on either side should be close to the bottom of the valley though.

— Page 322, *Deep Learning*, 2016.

The simplest implementation of Polyak-Ruppert averaging involves calculating the average of the weights of the models over the last few training epochs.

This can be improved by calculating a weighted average, where more weight is applied to more recent models, which is linearly decreased through prior epochs. An alternative and more widely used approach is to use an exponential decay in the weighted average.

Polyak-Ruppert averaging has been shown to improve the convergence of standard SGD [...] . Alternatively, an exponential moving average over the parameters can be used, giving higher weight to more recent parameter value.

— *Adam: A Method for Stochastic Optimization*, 2014.

Using an average or weighted average of model weights in the final model is a common technique in practice for ensuring the very best results are achieved from the training run. The approach is one of many *tricks* used in the Google Inception V2 and V3 deep convolutional neural network models for photo classification, a milestone in the field of deep learning.

Model evaluations are performed using a running average of the parameters computed over time.

— *Rethinking the Inception Architecture for Computer Vision*, 2015.

26.2 Average Model Weight Ensemble Case Study

In this section, we will demonstrate how to use the average model weight ensemble to reduce the variance of an MLP on a simple multiclass classification problem. This example provides a template for applying the average model weight ensemble to your own neural network for classification and regression problems.

26.2.1 Multiclass Classification Problem

We will use a small multiclass classification problem as the basis to demonstrate the model weight ensemble. The scikit-learn class provides the `make_blobs()` function that can be used to create a multiclass classification problem with the prescribed number of samples, input variables, classes, and variance of samples within a class. The problem can be configured to have two input variables (to represent the x and y coordinates of the points) and a standard deviation of 2.0 for points within each group. We will use the same random state (seed for the pseudorandom number generator) to ensure that we always get the same data points.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
```

Listing 26.1: Example of creating samples for the blobs problem.

The results are the input and output elements of a dataset that we can model. In order to get a feeling for the complexity of the problem, we can plot each point on a two-dimensional scatter plot and color each point by class value. The complete example is listed below.

```
# scatter plot of blobs dataset
from sklearn.datasets import make_blobs
from matplotlib import pyplot
from numpy import where
# generate 2d classification dataset
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=2, random_state=2)
# scatter plot for each class value
for class_value in range(3):
    # select indices of points with the class label
    row_ix = where(y == class_value)
    # scatter plot for points with a different color
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
# show plot
pyplot.show()
```

Listing 26.2: Example of plotting samples from the blobs problem.

Running the example creates a scatter plot of the entire dataset. We can see that the standard deviation of 2.0 means that the classes are not linearly separable (separable by a line) causing many ambiguous points. This is desirable as it means that the problem is non-trivial and will allow a neural network model to find many different *good enough* candidate solutions resulting in a high variance.

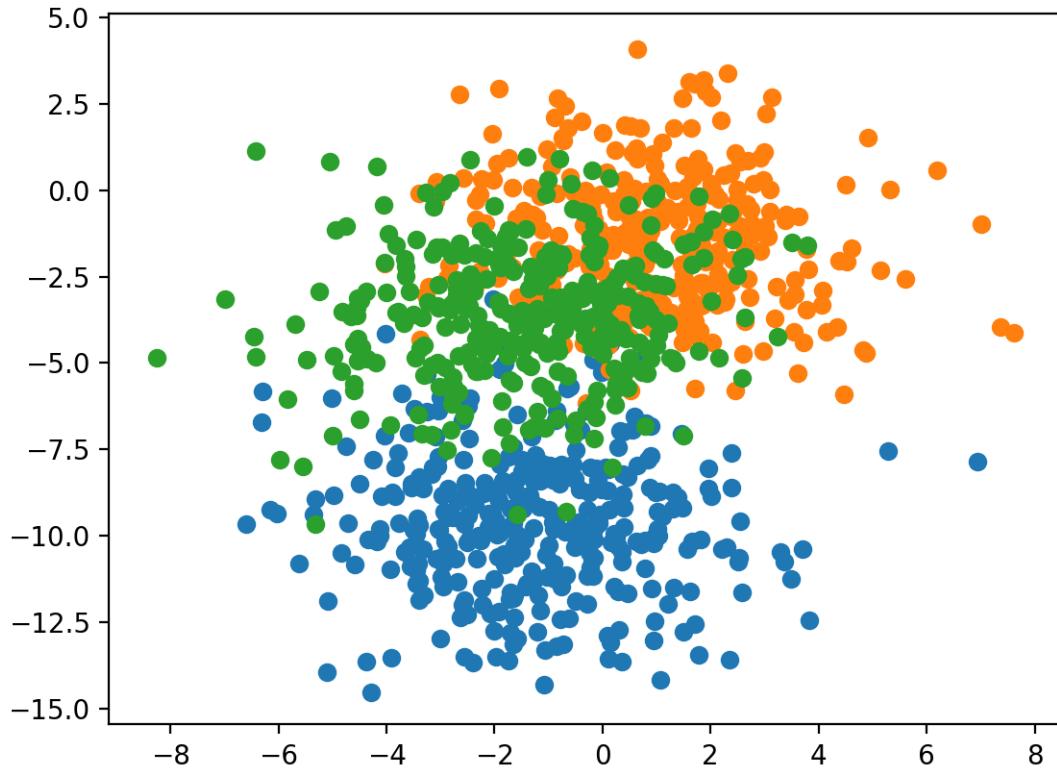


Figure 26.1: Scatter Plot of Blobs Dataset With Three Classes and Points Colored by Class Value.

26.2.2 Multilayer Perceptron Model

Before we define a model, we need to contrive a problem that is appropriate for the ensemble. In our problem, the training dataset is relatively small. Specifically, there is a 10:1 ratio of examples in the training dataset to the holdout dataset. This mimics a situation where we may have a vast number of unlabeled examples and a small number of labeled examples with which to train a model. We will create 1,100 data points from the blobs problem. The model will be trained on the first 100 points and the remaining 1,000 will be held back in a test dataset, unavailable to the model.

The problem is a multiclass classification problem, and we will model it using a softmax activation function on the output layer. This means that the model will predict a vector with three elements with the probability that the sample belongs to each of the three classes. Therefore, we must one hot encode the class values before we split the rows into the train and test datasets. We can do this using the Keras `to_categorical()` function.

```
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
```

```
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
```

Listing 26.3: Example of defining the dataset for modeling.

Next, we can define and compile the model. The model will expect samples with two input variables. The model then has a single hidden layer with 25 nodes and a rectified linear activation function, then an output layer with three nodes to predict the probability of each of the three classes and a softmax activation function. Because the problem is multiclass, we will use the categorical cross-entropy loss function to optimize the model and stochastic gradient descent with a small learning rate and momentum.

```
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Listing 26.4: Example of defining the MLP model.

The model is fit for 500 training epochs and we will evaluate the model each epoch on the test set, using the test set as a validation set.

```
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=500, verbose=0)
```

Listing 26.5: Example of fitting the MLP model.

At the end of the run, we will evaluate the performance of the model on the train and test sets.

```
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

Listing 26.6: Example of evaluating the MLP model.

Then finally, we will plot learning curves of the model accuracy over each training epoch on both the training and validation datasets.

```
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 26.7: Example of plotting learning curves for the MLP model.

Tying all of this together, the complete example is listed below.

```
# develop an mlp for blobs dataset
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
# fit model
history = model.fit(trainX, trainy, validation_data=(testX, testy), epochs=500, verbose=0)
# evaluate the model
_, train_acc = model.evaluate(trainX, trainy, verbose=0)
_, test_acc = model.evaluate(testX, testy, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
# plot loss learning curves
pyplot.subplot(211)
pyplot.title('Cross-Entropy Loss', pad=-40)
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
# plot accuracy learning curves
pyplot.subplot(212)
pyplot.title('Accuracy', pad=-40)
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```

Listing 26.8: Example of fitting an MLP on the blobs problem.

Running the example prints the performance of the final model on the train and test datasets.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved about 85% accuracy on the training dataset, which we know is optimistic, and about 80% on the test dataset, which we would expect to be more realistic.

Train: 0.850, Test: 0.804

Listing 26.9: Example output fitting an MLP on the blobs problem.

A line plot is also created showing the learning curves for the model accuracy on the train and test sets over each training epoch. We can see that training accuracy is more optimistic over most of the run, as we also noted with the final scores. Importantly, we do see a reasonable amount of variance in the accuracy during training on both the train and test datasets, potentially providing a good basis for using model weight averaging.

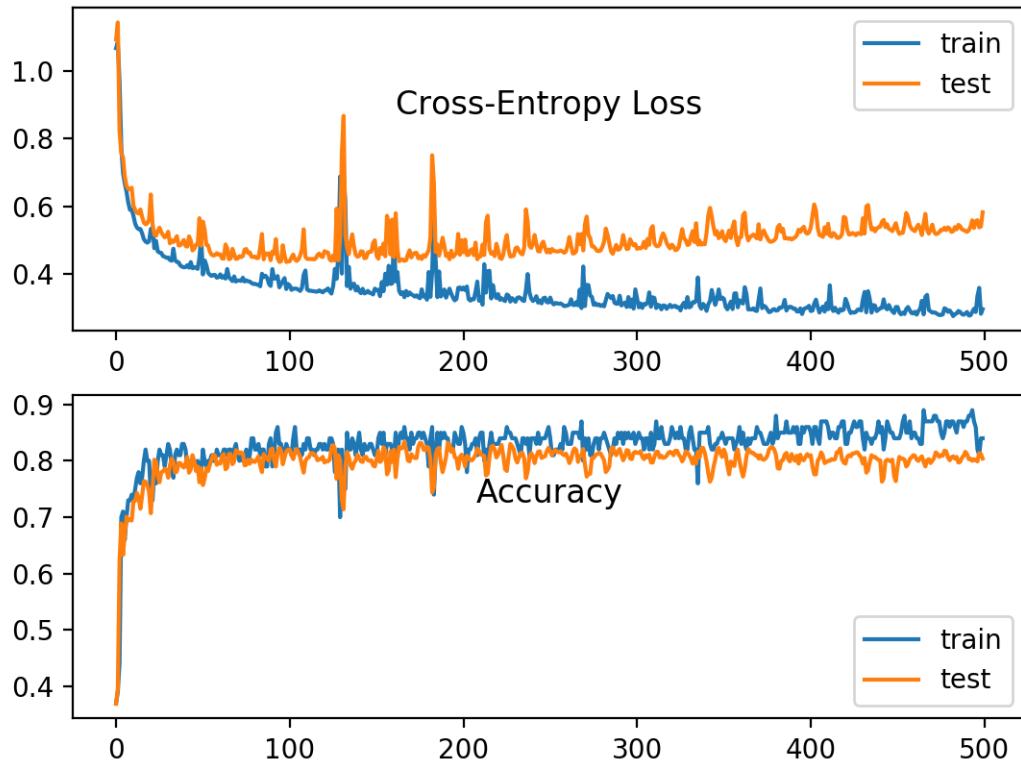


Figure 26.2: Line Plot Learning Curves of Model Accuracy on Train and Test Dataset over Each Training Epoch.

26.2.3 Save Multiple Models to File

One approach to the model weight ensemble is to keep a running average of model weights in memory. There are three downsides to this approach:

- It requires that you know beforehand the way in which the model weights will be combined; perhaps you want to experiment with different approaches.
- It requires that you know the number of epochs to use for training; maybe you want to use early stopping.
- It requires that you keep at least one copy of the entire network in memory; this could be very expensive for large models and fragile if the training process crashes or is killed.

An alternative is to save model weights to file during training as a first step, and later combine the weights from the saved models in order to make a final model. Perhaps the simplest way to implement this is to manually drive the training process, one epoch at a time, then save models at the end of the epoch if we have exceeded an upper limit on the number of epochs. For example, with our test problem, we will train the model for 500 epochs and perhaps save models from epoch 490 onwards (e.g. between and including epochs 490 and 499).

```
# fit model
n_epochs, n_save_after = 500, 490
for i in range(n_epochs):
    # fit model for a single epoch
    model.fit(trainX, trainy, epochs=1, verbose=0)
    # check if we should save the model
    if i >= n_save_after:
        model.save('model_' + str(i) + '.h5')
```

Listing 26.10: Example of fitting and saving multiple MLP models.

Models can be saved to file using the `save()` function on the model and specifying a filename that includes the epoch number. Note, saving and loading neural network models in Keras requires that you have the `h5py` library installed. You can install this library using `pip` as follows:

```
pip install h5py
```

Listing 26.11: Example installing the `h5py` library with `pip`.

Tying all of this together, the complete example of fitting the model on the training dataset and saving all models from the last 10 epochs is listed below.

```
# save models to file toward the end of a training run
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# define model
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
n_epochs, n_save_after = 500, 490
for i in range(n_epochs):
    # fit model for a single epoch
    model.fit(trainX, trainy, epochs=1, verbose=0)
    # check if we should save the model
    if i >= n_save_after:
        model.save('model_' + str(i) + '.h5')
```

Listing 26.12: Example of saving MLP models to file.

Running the example saves 10 models into the current working directory.

26.2.4 New Model With Average Models Weights

We can create a new model from multiple existing models with the same architecture. First, we need to load the models into memory. This is reasonable as the models are small. If you are working with very large models, it might be easier to load models one at a time and average the weights in memory. The `load_model()` Keras function can be used to load a saved model from file. The function `load_all_models()` below will load models from the current working directory. It takes the start and end epochs as arguments so that you can experiment with different groups of models saved over contiguous epochs.

```
# load models from file
def load_all_models(n_start, n_end):
    all_models = list()
    for epoch in range(n_start, n_end):
        # define filename for this ensemble
        filename = 'model_' + str(epoch) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models
```

Listing 26.13: Example of a function for loading saved models.

We can call the function to load all of the models.

```
# load models in order
members = load_all_models(490, 500)
print('Loaded %d models' % len(members))
```

Listing 26.14: Example of loading saved models.

Once loaded, we can create a new model with the weighted average of the model weights. Each model has a `get_weights()` function that returns a list of arrays, one for each layer in the model. We can enumerate each layer in the model, retrieve the same layer from each model, and calculate the weighted average. This will give us a set of weights. We can then use the `clone_model()` Keras function to create a clone of the architecture and call `set_weights()` function to use the average weights we have prepared. The `model_weight_ensemble()` function below implements this.

```
# create a model from the weights of multiple models
def model_weight_ensemble(members, weights):
    # determine how many layers need to be averaged
    n_layers = len(members[0].get_weights())
    # create an set of average model weights
    avg_model_weights = list()
    for layer in range(n_layers):
        # collect this layer from each model
```

```

layer_weights = array([model.get_weights()[layer] for model in members])
# weighted average of weights for this layer
avg_layer_weights = average(layer_weights, axis=0, weights=weights)
# store average layer weights
avg_model_weights.append(avg_layer_weights)
# create a new model with the same structure
model = clone_model(members[0])
# set the weights in the new
model.set_weights(avg_model_weights)
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
return model

```

Listing 26.15: Example of a function for averaging weights from multiple models.

Tying these elements together, we can load the 10 models and calculate the equally weighted average (arithmetic average) of the model weights. The complete listing is provided below.

```

# average the weights of multiple loaded models
from keras.models import load_model
from keras.models import clone_model
from numpy import average
from numpy import array

# load models from file
def load_all_models(n_start, n_end):
    all_models = list()
    for epoch in range(n_start, n_end):
        # define filename for this ensemble
        filename = 'model_' + str(epoch) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models

# create a model from the weights of multiple models
def model_weight_ensemble(members, weights):
    # determine how many layers need to be averaged
    n_layers = len(members[0].get_weights())
    # create an set of average model weights
    avg_model_weights = list()
    for layer in range(n_layers):
        # collect this layer from each model
        layer_weights = array([model.get_weights()[layer] for model in members])
        # weighted average of weights for this layer
        avg_layer_weights = average(layer_weights, axis=0, weights=weights)
        # store average layer weights
        avg_model_weights.append(avg_layer_weights)
    # create a new model with the same structure
    model = clone_model(members[0])
    # set the weights in the new
    model.set_weights(avg_model_weights)
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# load all models into memory

```

```

members = load_all_models(490, 500)
print('Loaded %d models' % len(members))
# prepare an array of equal weights
n_models = len(members)
weights = [1/n_models for i in range(1, n_models+1)]
# create a new model with the weighted average of all model weights
model = model_weight_ensemble(members, weights)
# summarize the created model
model.summary()

```

Listing 26.16: Example of loading models and creating a new model with the average of their weights.

Running the example first loads the 10 models from file.

```

>loaded model_490.h5
>loaded model_491.h5
>loaded model_492.h5
>loaded model_493.h5
>loaded model_494.h5
>loaded model_495.h5
>loaded model_496.h5
>loaded model_497.h5
>loaded model_498.h5
>loaded model_499.h5
Loaded 10 models

```

Listing 26.17: Example output from loading saved models.

A model weight ensemble is created from these 10 models giving equal weight to each model and a summary of the model structure is reported.

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 25)	75
dense_2 (Dense)	(None, 3)	78

Total params: 153
Trainable params: 153
Non-trainable params: 0

Listing 26.18: Example output from summarizing the new model.

26.2.5 Predicting With an Average Model Weight Ensemble

Now that we know how to calculate a weighted average of model weights, we can evaluate predictions with the resulting model. One issue is that we don't know how many models are appropriate to combine in order to achieve good performance. We can address this by evaluating model weight averaging ensembles with the last n models and vary n to see how many models results in good performance. The `evaluate_n_members()` function below will create a new model from a given number of loaded models. Each model is given an equal weight in

contributing to the final model, then the `model_weight_ensemble()` function is called to create the final model that is then evaluated on the test dataset.

```
# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # reverse loaded models so we build the ensemble with the last models first
    members = list(reversed(members))
    # select a subset of members
    subset = members[:n_members]
    # prepare an array of equal weights
    weights = [1.0/n_members for i in range(1, n_members+1)]
    # create a new model with the weighted average of all model weights
    model = model_weight_ensemble(subset, weights)
    # make predictions and evaluate accuracy
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    return test_acc
```

Listing 26.19: Example of evaluating a model created from a given number of members.

Importantly, the list of loaded models is reversed first to ensure that the last n models in the training run are used, which we would assume might have better performance on average.

```
# reverse loaded models so we build the ensemble with the last models first
members = list(reversed(members))
```

Listing 26.20: Example of reversing the list of loaded ensemble members.

We can then evaluate models created from different numbers of the last n models saved from the training run from the last 1-model to the last 10 models. In addition to evaluating the combined final model, we can also evaluate each saved standalone model on the test dataset to compare performance.

```
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, len(members)+1):
    # evaluate model with i members
    ensemble_score = evaluate_n_members(members, i, testX, testy)
    # evaluate the i'th model standalone
    _, single_score = members[i-1].evaluate(testX, testy, verbose=0)
    # summarize this step
    print('> %d: single=%f, ensemble=%f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
```

Listing 26.21: Example of evaluating a suite of different average model weight models.

The collected scores can be plotted, with blue dots for the accuracy of the single saved models and the orange line for the test accuracy for the model that combines the weights the last n models.

```
# plot score vs number of ensemble members
x_axis = [i for i in range(1, len(members)+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()
```

Listing 26.22: Example of plotting model performance vs the number of contributing members.

Tying all of this together, the complete example is listed below.

```
# average of model weights on blobs problem
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import load_model
from keras.models import clone_model
from matplotlib import pyplot
from numpy import average
from numpy import array

# load models from file
def load_all_models(n_start, n_end):
    all_models = []
    for epoch in range(n_start, n_end):
        # define filename for this ensemble
        filename = 'model_' + str(epoch) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models

# # create a model from the weights of multiple models
def model_weight_ensemble(members, weights):
    # determine how many layers need to be averaged
    n_layers = len(members[0].get_weights())
    # create an set of average model weights
    avg_model_weights = []
    for layer in range(n_layers):
        # collect this layer from each model
        layer_weights = array([model.get_weights()[layer] for model in members])
        # weighted average of weights for this layer
        avg_layer_weights = average(layer_weights, axis=0, weights=weights)
        # store average layer weights
        avg_model_weights.append(avg_layer_weights)
    # create a new model with the same structure
    model = clone_model(members[0])
    # set the weights in the new
    model.set_weights(avg_model_weights)
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # prepare an array of equal weights
    weights = [1.0/n_members for i in range(1, n_members+1)]
    # create a new model with the weighted average of all model weights
    model = model_weight_ensemble(subset, weights)
    # make predictions and evaluate accuracy
    _, test_acc = model.evaluate(testX, testy, verbose=0)
    return test_acc
```

```

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# load models in order
members = load_all_models(490, 500)
print('Loaded %d models' % len(members))
# reverse loaded models so we build the ensemble with the last models first
members = list(reversed(members))
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, len(members)+1):
    # evaluate model with i members
    ensemble_score = evaluate_n_members(members, i, testX, testy)
    # evaluate the i'th model standalone
    _, single_score = members[i-1].evaluate(testX, testy, verbose=0)
    # summarize this step
    print('> %d: single=%f, ensemble=%f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# plot score vs number of ensemble members
x_axis = [i for i in range(1, len(members)+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()

```

Listing 26.23: Example the performance of average model weight ensemble models with differing number of contributing members.

Running the example first loads the 10 saved models.

```

>loaded model_490.h5
>loaded model_491.h5
>loaded model_492.h5
>loaded model_493.h5
>loaded model_494.h5
>loaded model_495.h5
>loaded model_496.h5
>loaded model_497.h5
>loaded model_498.h5
>loaded model_499.h5
Loaded 10 models

```

Listing 26.24: Example output from loading saved models.

The performance of each individually saved model is reported as well as an ensemble model with weights averaged from all models up to and including each model, working backward from the end of the training run. The results show that the best test accuracy was about 81.4% achieved by the last two models. We can see that the test accuracy of the model weight ensemble levels out the performance and performs just as well.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
> 1: single=0.814, ensemble=0.814
> 2: single=0.814, ensemble=0.814
> 3: single=0.811, ensemble=0.813
> 4: single=0.805, ensemble=0.813
> 5: single=0.807, ensemble=0.811
> 6: single=0.805, ensemble=0.807
> 7: single=0.802, ensemble=0.809
> 8: single=0.805, ensemble=0.808
> 9: single=0.805, ensemble=0.808
> 10: single=0.810, ensemble=0.807
```

Listing 26.25: Example output from evaluating single models and ensemble models.

A line plot is also created showing the test accuracy of each single model (blue dots) and the performance of the model weight ensemble (orange line). We can see that averaging the model weights does level out the performance of the final model and performs at least as well as the final model of the run.

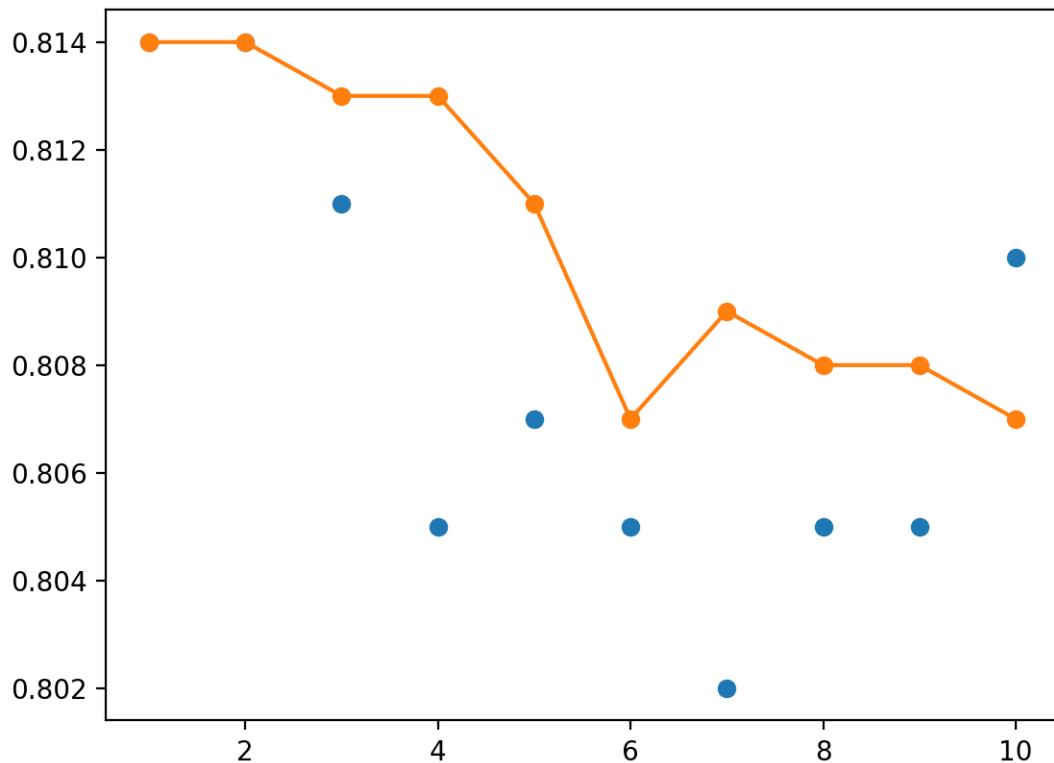


Figure 26.3: Line Plot of Single Model Test Performance (blue dots) and Model Weight Ensemble Test Performance (orange line).

26.2.6 Linearly and Exponentially Decreasing Weighted Average

We can update the example and evaluate a linearly decreasing weighting of the model weights in the ensemble. The weights can be calculated as follows:

```
# prepare an array of linearly decreasing weights
weights = [i/n_members for i in range(n_members, 0, -1)]
```

Listing 26.26: Example of using a linearly decreasing weighting for ensemble members.

This can be used instead of the equal weights in the `evaluate_n_members()` function. The complete example is listed below.

```
# linearly decreasing weighted average of models on blobs problem
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import load_model
from keras.models import clone_model
from matplotlib import pyplot
from numpy import average
from numpy import array

# load models from file
def load_all_models(n_start, n_end):
    all_models = []
    for epoch in range(n_start, n_end):
        # define filename for this ensemble
        filename = 'model_' + str(epoch) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models

# create a model from the weights of multiple models
def model_weight_ensemble(members, weights):
    # determine how many layers need to be averaged
    n_layers = len(members[0].get_weights())
    # create an set of average model weights
    avg_model_weights = []
    for layer in range(n_layers):
        # collect this layer from each model
        layer_weights = array([model.get_weights()[layer] for model in members])
        # weighted average of weights for this layer
        avg_layer_weights = average(layer_weights, axis=0, weights=weights)
        # store average layer weights
        avg_model_weights.append(avg_layer_weights)
    # create a new model with the same structure
    model = clone_model(members[0])
    # set the weights in the new
    model.set_weights(avg_model_weights)
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
```

```

# select a subset of members
subset = members[:n_members]
# prepare an array of linearly decreasing weights
weights = [i/n_members for i in range(n_members, 0, -1)]
# create a new model with the weighted average of all model weights
model = model_weight_ensemble(subset, weights)
# make predictions and evaluate accuracy
_, test_acc = model.evaluate(testX, testy, verbose=0)
return test_acc

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# load models in order
members = load_all_models(490, 500)
print('Loaded %d models' % len(members))
# reverse loaded models so we build the ensemble with the last models first
members = list(reversed(members))
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, len(members)+1):
    # evaluate model with i members
    ensemble_score = evaluate_n_members(members, i, testX, testy)
    # evaluate the i'th model standalone
    _, single_score = members[i-1].evaluate(testX, testy, verbose=0)
    # summarize this step
    print('> %d: single=%f, ensemble=%f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# plot score vs number of ensemble members
x_axis = [i for i in range(1, len(members)+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()

```

Listing 26.27: Example the performance of linearly weighted ensembles.

Running the example reports the performance of each single model again, and this time the test accuracy of each average model weight ensemble with a linearly decreasing contribution of models.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

We can see that, at least in this case, the ensemble achieves a small bump in performance above any standalone model to about 81.5% accuracy.

```

...
> 1: single=0.814, ensemble=0.814
> 2: single=0.814, ensemble=0.815
> 3: single=0.811, ensemble=0.814

```

```
> 4: single=0.805, ensemble=0.813
> 5: single=0.807, ensemble=0.813
> 6: single=0.805, ensemble=0.813
> 7: single=0.802, ensemble=0.811
> 8: single=0.805, ensemble=0.810
> 9: single=0.805, ensemble=0.809
> 10: single=0.810, ensemble=0.809
```

Listing 26.28: Example output from linearly weighted ensembles.

The line plot shows the bump in performance and shows a more stable performance in terms of test accuracy over the different sized ensembles created, as compared to the use of an evenly weighted ensemble.

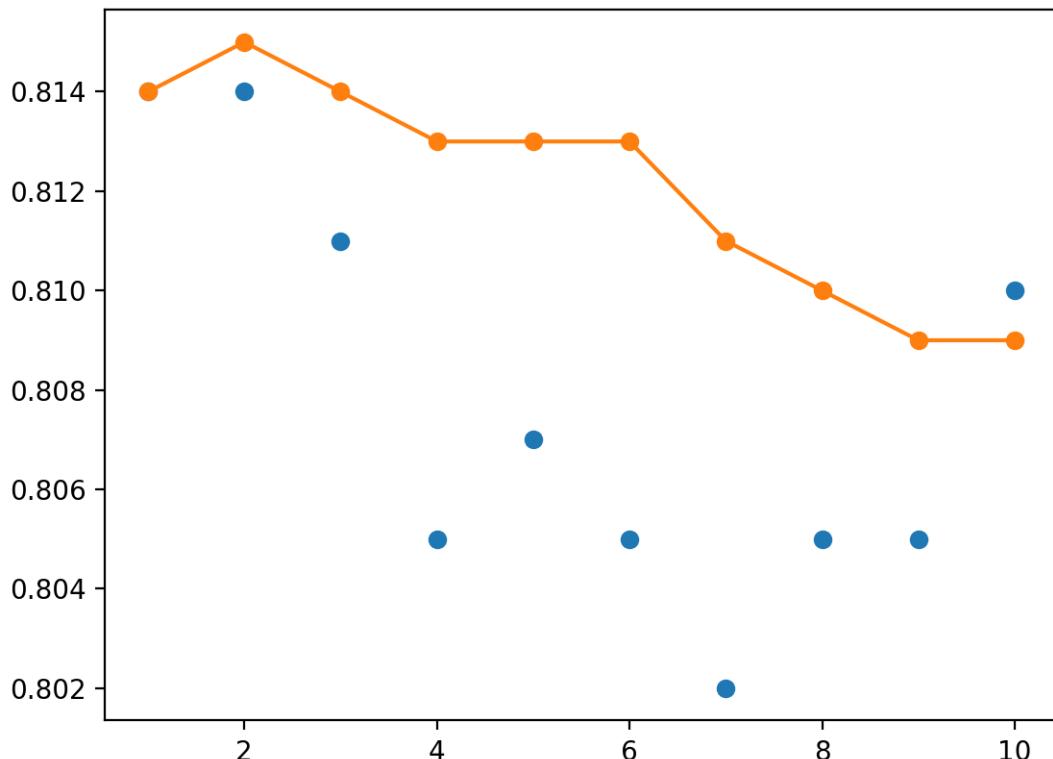


Figure 26.4: Line Plot of Single Model Test Performance (blue dots) and Model Weight Ensemble Test Performance (orange line) With a Linear Decay.

We can also experiment with an exponential decay of the contribution of models. This requires that a decay rate (α) is specified. The example below creates weights for an exponential decay with a decrease rate of 2.

```
# prepare an array of exponentially decreasing weights
alpha = 2.0
weights = [exp(-i/alpha) for i in range(1, n_members+1)]
```

Listing 26.29: Example of using an exponentially decreasing weighting for ensemble members.

The complete example with an exponential decay for the contribution of models to the average weights in the ensemble model is listed below.

```
# exponentially decreasing weighted average of models on blobs problem
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import load_model
from keras.models import clone_model
from matplotlib import pyplot
from numpy import average
from numpy import array
from math import exp

# load models from file
def load_all_models(n_start, n_end):
    all_models = list()
    for epoch in range(n_start, n_end):
        # define filename for this ensemble
        filename = 'model_' + str(epoch) + '.h5'
        # load model from file
        model = load_model(filename)
        # add to list of members
        all_models.append(model)
        print('>loaded %s' % filename)
    return all_models

# create a model from the weights of multiple models
def model_weight_ensemble(members, weights):
    # determine how many layers need to be averaged
    n_layers = len(members[0].get_weights())
    # create an set of average model weights
    avg_model_weights = list()
    for layer in range(n_layers):
        # collect this layer from each model
        layer_weights = array([model.get_weights()[layer] for model in members])
        # weighted average of weights for this layer
        avg_layer_weights = average(layer_weights, axis=0, weights=weights)
        # store average layer weights
        avg_model_weights.append(avg_layer_weights)
    # create a new model with the same structure
    model = clone_model(members[0])
    # set the weights in the new
    model.set_weights(avg_model_weights)
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# evaluate a specific number of members in an ensemble
def evaluate_n_members(members, n_members, testX, testy):
    # select a subset of members
    subset = members[:n_members]
    # prepare an array of exponentially decreasing weights
    alpha = 2.0
    weights = [exp(-i/alpha) for i in range(1, n_members+1)]
```

```

# create a new model with the weighted average of all model weights
model = model_weight_ensemble(subset, weights)
# make predictions and evaluate accuracy
_, test_acc = model.evaluate(testX, testy, verbose=0)
return test_acc

# generate 2d classification dataset
X, y = make_blobs(n_samples=1100, centers=3, n_features=2, cluster_std=2, random_state=2)
# one hot encode output variable
y = to_categorical(y)
# split into train and test
n_train = 100
trainX, testX = X[:n_train, :], X[n_train:, :]
trainy, testy = y[:n_train], y[n_train:]
# load models in order
members = load_all_models(490, 500)
print('Loaded %d models' % len(members))
# reverse loaded models so we build the ensemble with the last models first
members = list(reversed(members))
# evaluate different numbers of ensembles on hold out set
single_scores, ensemble_scores = list(), list()
for i in range(1, len(members)+1):
    # evaluate model with i members
    ensemble_score = evaluate_n_members(members, i, testX, testy)
    # evaluate the i'th model standalone
    _, single_score = members[i-1].evaluate(testX, testy, verbose=0)
    # summarize this step
    print('> %d: single=%f, ensemble=%f' % (i, single_score, ensemble_score))
    ensemble_scores.append(ensemble_score)
    single_scores.append(single_score)
# plot score vs number of ensemble members
x_axis = [i for i in range(1, len(members)+1)]
pyplot.plot(x_axis, single_scores, marker='o', linestyle='None')
pyplot.plot(x_axis, ensemble_scores, marker='o')
pyplot.show()

```

Listing 26.30: Example the performance of exponentially weighted ensembles.

Running the example shows a small improvement in performance much like the use of a linear decay in the weighted average of the saved models.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```

> 1: single=0.814, ensemble=0.814
> 2: single=0.814, ensemble=0.815
> 3: single=0.811, ensemble=0.814
> 4: single=0.805, ensemble=0.814
> 5: single=0.807, ensemble=0.813
> 6: single=0.805, ensemble=0.813
> 7: single=0.802, ensemble=0.813
> 8: single=0.805, ensemble=0.813
> 9: single=0.805, ensemble=0.813
> 10: single=0.810, ensemble=0.813

```

Listing 26.31: Example output from exponentially weighted ensembles.

The line plot of the test accuracy scores shows the stronger stabilizing effect of using the exponential decay instead of the linear or equal weighting of models.

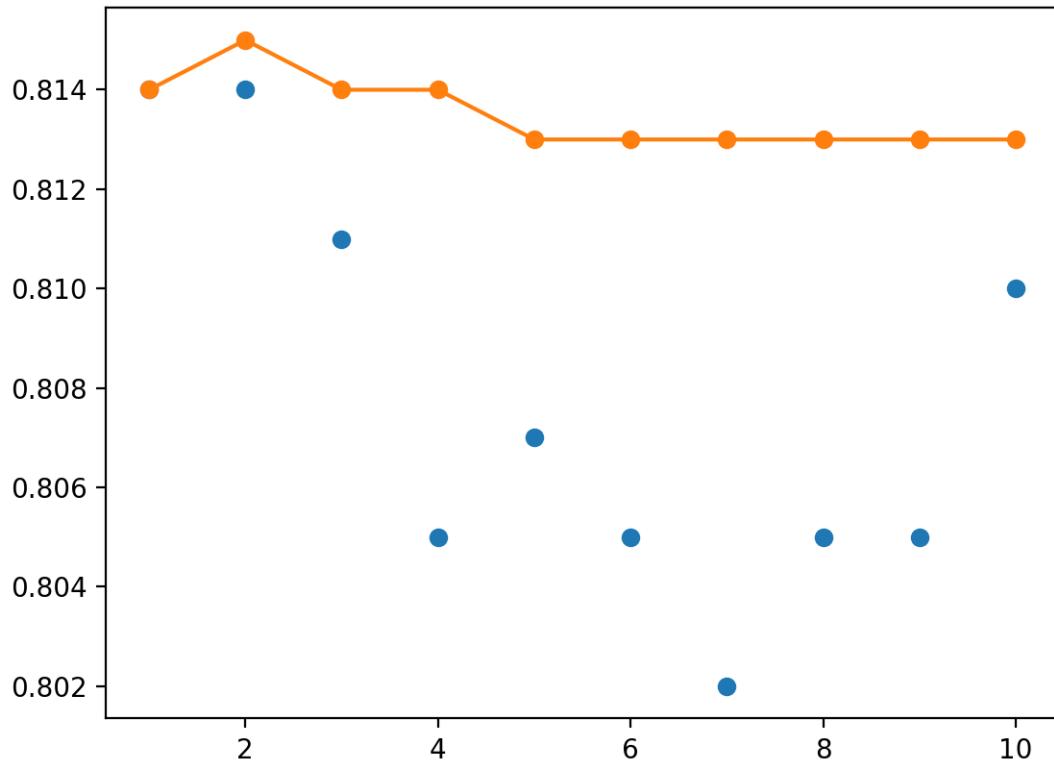


Figure 26.5: Line Plot of Single Model Test Performance (blue dots) and Model Weight Ensemble Test Performance (orange line) With an Exponential Decay.

26.3 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Number of Models.** Evaluate the effect of many more models contributing their weights to the final model.
- **Decay Rate.** Evaluate the effect on test performance of using different decay rates for an exponentially weighted average.

If you explore any of these extensions, I'd love to know.

26.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

26.4.1 Books

- Section 8.7.3 Polyak Averaging, *Deep Learning*, 2016.
<https://amzn.to/2A1v00d>

26.4.2 Papers

- *Acceleration of Stochastic Approximation by Averaging*, 1992.
<https://pubs.siam.org/doi/abs/10.1137/0330046>
- *Efficient estimations from a slowly convergent robbins-monro process*, 1988.
<https://ecommons.cornell.edu/handle/1813/8664>

26.4.3 APIs

- Getting started with the Keras Sequential model.
<https://keras.io/getting-started/sequential-model-guide/>
- Keras Core Layers API.
<https://keras.io/layers/core/>
- `sklearn.datasets.make_blobs` API.
http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
- `numpy.average` API.
<https://docs.scipy.org/doc/numpy/reference/generated/numpy.average.html>

26.5 Summary

In this tutorial, you discovered how to combine the weights from multiple different models into a single model for making predictions. Specifically, you learned:

- The stochastic and challenging nature of training neural networks can mean that the optimization process does not converge.
- Creating a model with the average of the weights from models observed towards the end of a training run can result in a more stable and sometimes better-performing solution.
- How to develop final models created with the equal, linearly, and exponentially weighted average of model parameters from multiple saved models.

26.5.1 Next

This was the last tutorial in this Part. Next you will discover the Appendix and resources that you can use to dive deeper into the topic of getting better performance with neural networks.

Part IV

Appendix

Appendix A

Getting Help

This is just the beginning of your journey with better deep learning in Python. As you start to work on methods or expand your existing knowledge of algorithms you may need help. This chapter points out some of the best sources of help.

A.1 Applied Neural Networks

This section lists some of the best books on the practical considerations of working with neural network models.

- *Deep Learning With Python*, 2017.
<https://amzn.to/2NJq1pf>
- *Deep Learning*, 2016.
<https://amzn.to/2NJW3gE>
- *Neural Networks: Tricks of the Trade*, 2012.
<https://amzn.to/2S83KiB>
- *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
<https://amzn.to/2poq0xc>
- *Neural Networks for Pattern Recognition*, 1995.
<https://amzn.to/2I9gNMP>

A.2 Official Keras Destinations

This section lists the official Keras sites that you may find helpful.

- Keras Official Blog.
<https://blog.keras.io/>
- Keras API Documentation.
<https://keras.io/>
- Keras Source Code Project.
<https://github.com/keras-team/keras>

A.3 Where to Get Help with Keras

This section lists the 9 best places I know where you can get help with Keras.

- Keras Users Google Group.
<https://groups.google.com/forum/#!forum/keras-users>
- Keras Slack Channel (you must request to join).
<https://keras-slack-autojoin.herokuapp.com/>
- Keras on Gitter.
<https://gitter.im/Keras-io/Lobby#>
- Keras tag on StackOverflow.
<https://stackoverflow.com/questions/tagged/keras>
- Keras tag on CrossValidated.
<https://stats.stackexchange.com/questions/tagged/keras>
- Keras tag on DataScience.
<https://datascience.stackexchange.com/questions/tagged/keras>
- Keras Topic on Quora.
<https://www.quora.com/topic/Keras>
- Keras Github Issues.
<https://github.com/keras-team/keras/issues>
- Keras on Twitter.
<https://twitter.com/hashtag/keras>

A.4 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

- Boil your question down to the simplest form. E.g. not something broad like *my model does not work* or *how does x work*.
- Search for answers before asking questions.
- Provide complete code and error messages.
- Boil your code down to the smallest possible working example that demonstrates the issue.

These are excellent resources both for posting unique questions, but also for searching through the answers to questions on related topics.

A.5 Contact the Author

You are not alone. If you ever have any questions about deep learning or this book, please contact me directly. I will do my best to help.

Jason Brownlee

Jason@MachineLearningMastery.com

Appendix B

How to Setup Your Workstation

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning and deep learning software. These instructions are suitable for Windows, macOS, and Linux platforms. I will demonstrate them on macOS, so you may see some mac dialogs and file extensions.

B.1 Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda
4. Install Deep Learning Libraries

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2 Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

- 1. Visit the Anaconda homepage.
<https://www.continuum.io/>
- 2. Click Anaconda from the menu and click **Download** to go to the download page.
<https://www.continuum.io/downloads>

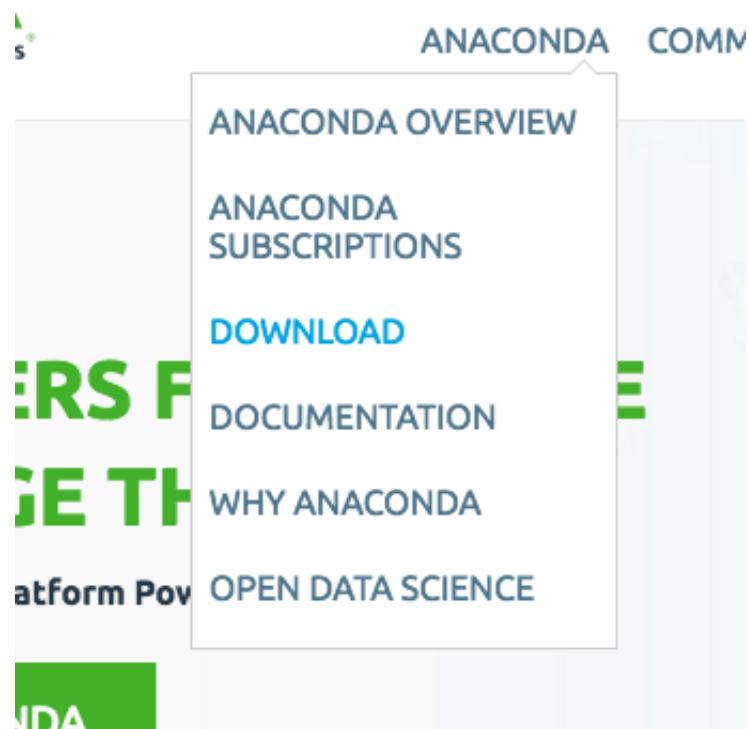


Figure B.1: Click Anaconda and Download.

- 3. Choose the download suitable for your platform (Windows, OSX, or Linux):
 - Choose Python 3.6
 - Choose the Graphical Installer

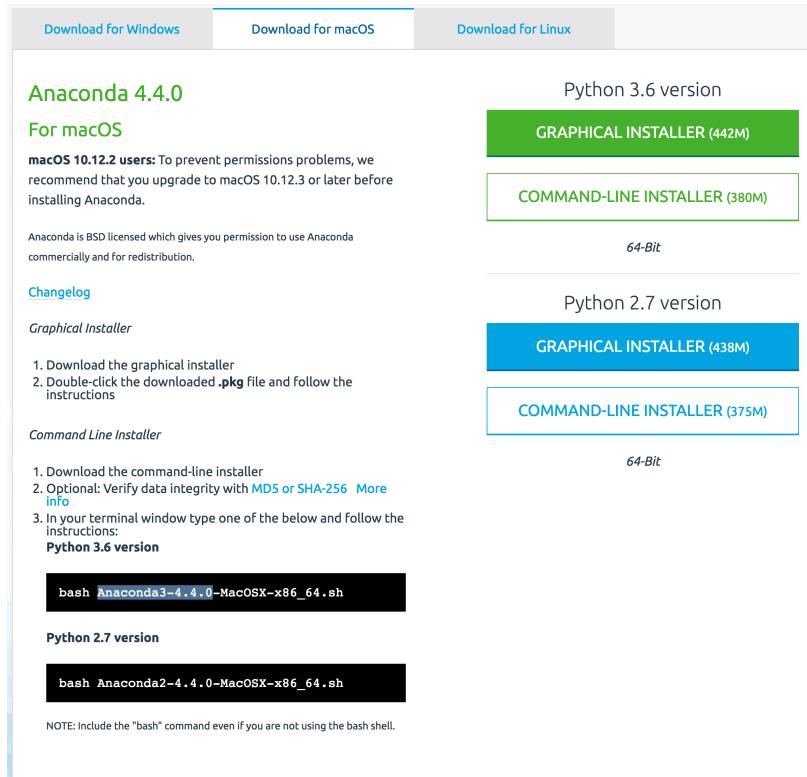


Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on macOS, so I chose the macOS version. The file is about 426 MB. You should have a file with a name like:

```
Anaconda3-4.4.0-MacOSX-x86_64.pkg
```

Listing B.1: Example filename on macOS.

B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

- 1. Double click the downloaded file.
- 2. Follow the installation wizard.

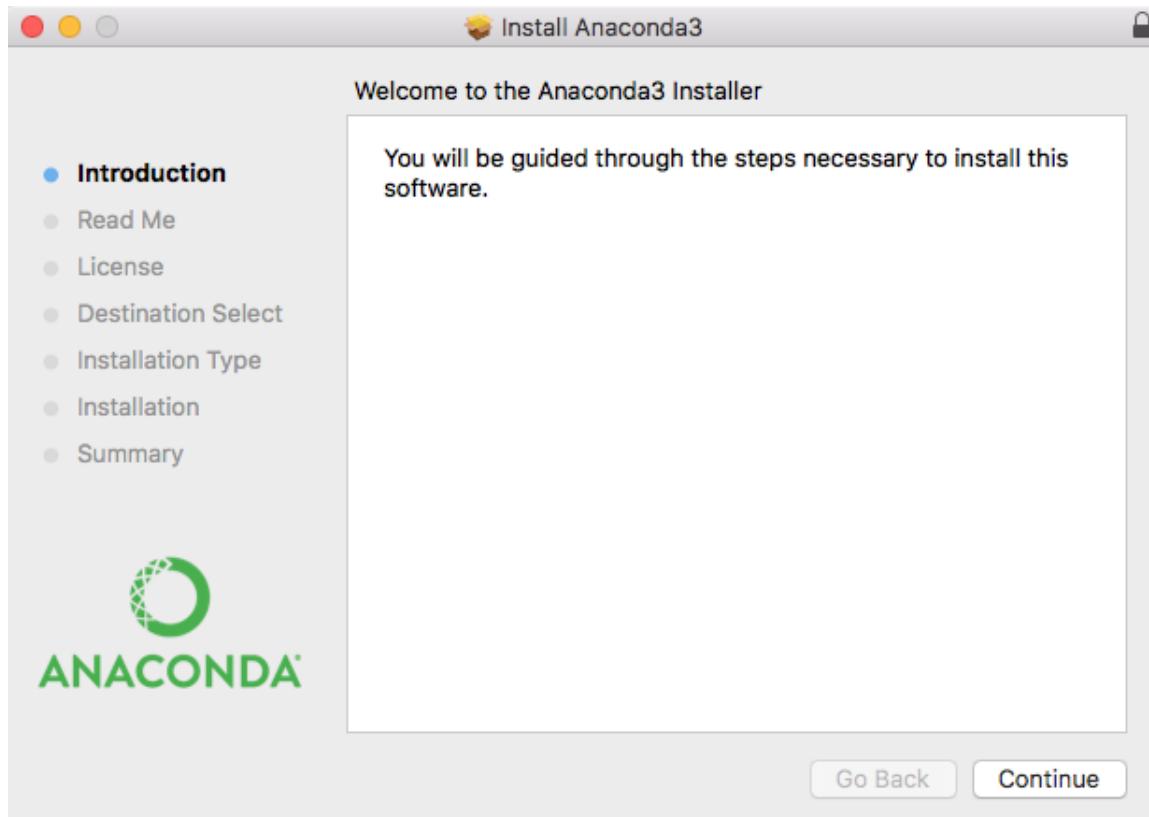


Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.

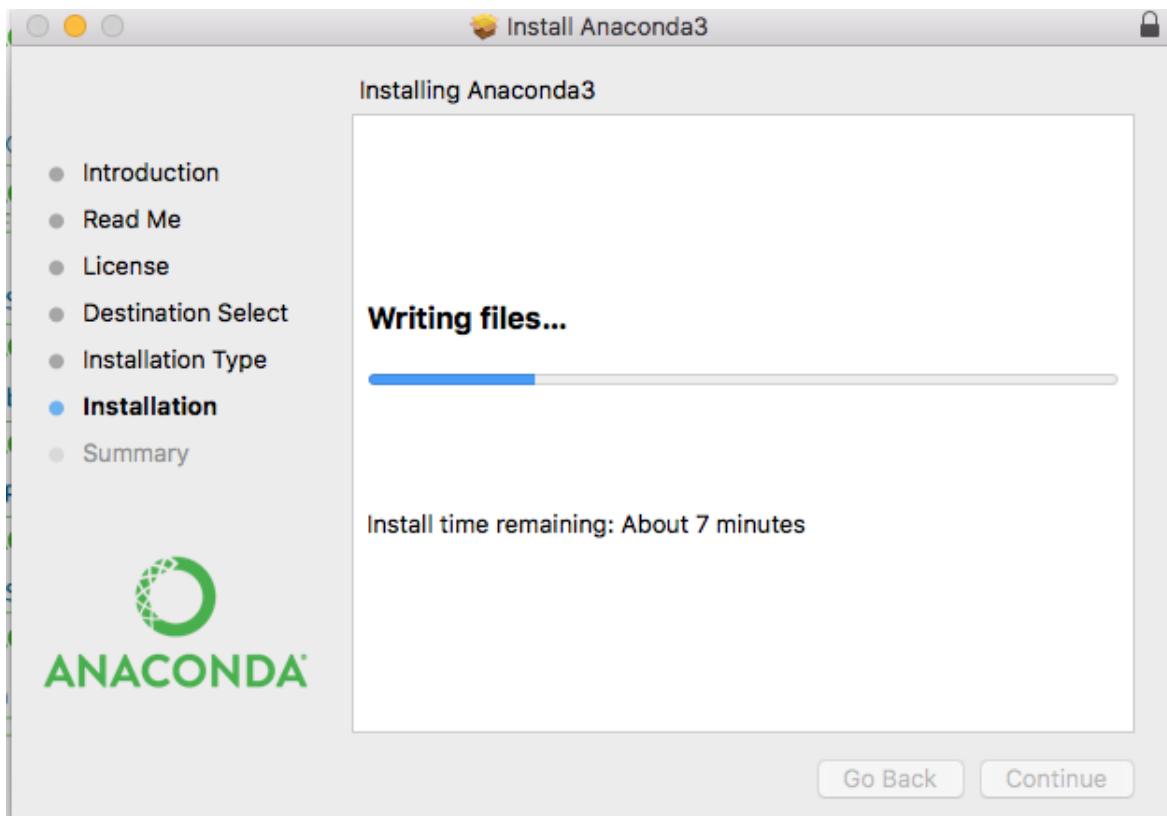


Figure B.4: Anaconda Python Installation Wizard Writing Files.

The installation should take less than 10 minutes and take up a little more than 1 GB of space on your hard drive.

B.4 Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.

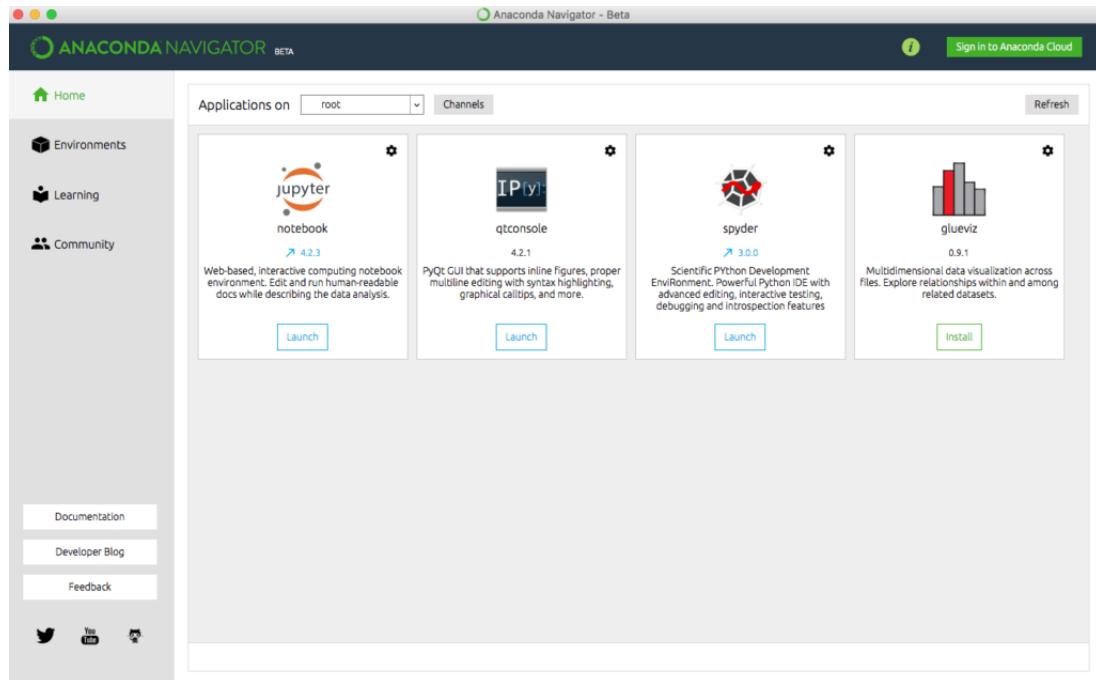


Figure B.5: Anaconda Navigator GUI.

You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

- 1. Open a terminal (command line window).
- 2. Confirm conda is installed correctly, by typing:

```
conda -v
```

Listing B.2: Check the conda version.

You should see the following (or something similar):

```
conda 4.3.21
```

Listing B.3: Example conda version.

- 3. Confirm Python is installed correctly by typing:

```
python -v
```

Listing B.4: Check the Python version.

You should see the following (or something similar):

```
Python 3.6.1 :: Anaconda 4.4.0 (x86_64)
```

Listing B.5: Example Python version.

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the *Further Reading* section.

- 4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

Listing B.6: Update conda and anaconda.

You may need to install some packages and confirm the updates.

- 5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type `python` and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```
# check library version numbers
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.7: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

Listing B.8: Run the script from the command line.

You should see output like the following:

```
scipy: 1.5.2
numpy: 1.19.1
matplotlib: 3.3.0
pandas: 1.1.0
statsmodels: 0.11.1
sklearn: 0.23.2
```

Listing B.9: Sample output of versions script.

B.5 Install Deep Learning Libraries

In this step, we will install Python libraries used for deep learning, specifically: Theano, TensorFlow, and Keras. Note: I recommend using Keras for deep learning and Keras only requires one of Theano or TensorFlow to be installed. You do not need both. There may be problems installing TensorFlow on some Windows machines.

- 1. Install the Theano deep learning library by typing:

```
conda install theano
```

Listing B.10: Install Theano with conda.

- 2. Install the TensorFlow deep learning library by typing:

```
conda install -c conda-forge tensorflow
```

Listing B.11: Install TensorFlow with conda.

Alternatively, you may choose to install using pip and a specific version of TensorFlow for your platform.

- 3. Install Keras by typing:

```
pip install keras
```

Listing B.12: Install Keras with pip.

- 4. Confirm your deep learning environment is installed and working correctly.

Create a script that prints the version numbers of each library, as we did before for the SciPy environment.

```
# check deep learning version numbers
# theano
import theano
print('theano: %s' % theano.__version__)
# tensorflow
import tensorflow
print('tensorflow: %s' % tensorflow.__version__)
# keras
import keras
print('keras: %s' % keras.__version__)
```

Listing B.13: Code to check that key deep learning libraries are installed.

Save the script to a file `deep_versions.py`. Run the script by typing:

```
python deep_versions.py
```

Listing B.14: Run script from the command line.

You should see output like:

```
theano: 1.0.5
tensorflow: 2.3.0
keras: 2.4.3
```

Listing B.15: Sample output of the deep learning versions script.

B.6 Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda homepage.
<https://www.continuum.io/>
- Anaconda Navigator.
<https://docs.continuum.io/anaconda/navigator.html>
- The conda command line tool.
<http://conda.pydata.org/docs/index.html>
- Instructions for installing TensorFlow in Anaconda.
https://www.tensorflow.org/get_started/os_setup#anaconda_installation

B.7 Summary

Congratulations, you now have a working Python development environment for machine learning and deep learning. You can now learn and practice machine learning and deep learning on your workstation.

Part V

Conclusions

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now know:

- A checklist of techniques that you can use to improve the performance of deep learning neural network models on your own predictive modeling problems.
- How to accelerate learning through better configured stochastic gradient descent batch size, loss functions, learning rates, and to avoid exploding gradients via gradient clipping.
- How to accelerate learning through correct data scaling, batch normalization, and use of modern activation functions such as the rectified linear activation function.
- How to accelerate learning through choosing better initial weights with greedy layer-wise pretraining and transfer learning.
- A gentle introduction to the problem of overfitting and a tour of regularization techniques.
- How to reduce overfitting by updating the loss function using techniques such as weight regularization, weight constraints, and activation regularization.
- How to reduce overfitting using techniques such as dropout, the addition of noise, and early stopping.
- A gentle introduction to how to combine the predictions from multiple models and a tour of ensemble learning techniques.
- How to combine the predictions from multiple different models using techniques such as weighted averaging ensembles and stacked generalization ensembles, also known as blending.
- How to combine the predictions from multiple models saved during a single training run with techniques such as horizontal ensembles and snapshot ensembles.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable set of skills for improving the performance of deep learning neural network models. You can now confidently:

- Diagnose poor model training and problems such as premature convergence and accelerate the model training process using one or a combination of modifications to the learning algorithm.

- Diagnose cases of overfitting the training dataset and reduce generalization error using one or a combination of modifications of the model, loss function, or learning algorithm.
- Diagnose high variance in a final model and improve the average predictive skill by combining the predictions from multiple models trained over a single or multiple training runs.

The sky's the limit.

Thank You!

I want to take a moment and sincerely thank you for letting me help you start your journey toward better deep learning. I hope you keep learning and have fun as you continue to master machine learning.

Jason Brownlee
2020