The Author

Mike Chyson (Li Mingming)

The Big Book of

Deep Learning

Interpretation on deep learning

Nov, 16, 2020

Website

https://chyson.net
https://chyson.fun

Dedication

I benifit a lot from Goodfellow's "DEEP LEARNING". The book explain machine learning and deep learning on theory deeply.

"deep learning for computer vision with python starter", "deep learning for computer vision with python practitioner" and "deep learning for computer vision with python imagenet" explain computer vision by practice. You can learn machine learning and deep learning by practicing.

"Case Studies" is about the basis of the OpenCV, and can be used as the introduction to OpenCV.

Having read several books on machine learning and deep learning, I decide to write down some important concepts and my interpretation on them. You may find that it is difficult to understand some sentences, because I cite them from the book "DEEP LEARNING". It is recommend to read the book "DEEP LEARNING" yourself if you find these sentences difficult to understand. Here is the hyperlinks: `https://www.deeplearningbook.org/`

After the theory interpretation, I write some programs to implement the deep learning traning and predict.

Contents

Contents **1**

I Theory **5**

1 Introduction **7**

2 Machine Learning Basics **9**

2.1 Learning Algorithms . . . 9
2.2 Capacity, Overfitting and Underfitting . . . 9
2.2.1 The No Free Lunch Theorem . . . 11
2.2.2 Regularization . . . 12
2.3 Hyperparameters and validation sets . . . 13
2.4 Estimators, bias and variance . . . 13
2.4.1 Point estimation . . . 13
2.4.2 Bias . . . 14
2.4.3 Variance and standard error . . . 14
2.4.4 Consistency . . . 14
2.5 Maximum likelihood estimation (*) . . . 15
2.5.1 Properties of maximum likelihood . . . 16
2.6 Bayesian statistics . . . 16
2.7 Stochastic Gradient Descent . . . 17
2.8 Building a machine learning algorithm . . . 18
2.9 Challenges Motivating Deep Learning . . . 18
2.9.1 The Curse of dimensionality . . . 18
2.9.2 Local constancy and smoothness regularization . . . 18
2.10 Manifold Learning . . . 19

3 Deep Feedforward Networks **21**
3.1 Gradient-Based Learning 21

4 Regularization for Deep Learning **23**
4.1 Parameter Norm Penalties 24
4.1.1 L^2 Parameter Regularization 24
4.1.2 L^1 Regularization 24
4.2 Dataset Augmentation 24
4.3 Noise Robustness 24
4.4 Semi-Supervised Learning 25
4.5 Multitask Learning 25
4.6 Early Stopping 25
4.7 Parameter Tying and Parameter Sharing 25
4.8 Sparse Representations 26
4.9 Bagging and Other Emsemble Methods 26
4.10 Adversarial Training 26

5 Optimization for Training Deep Models **29**

6 Convolutional Networks **31**
6.1 The Convolution Operation 31
6.2 Motivation 32
6.2.1 Sparse interaction 33
6.2.2 Parameter sharing 33
6.2.3 Equivariant representations 33
6.3 Pooling 34
6.4 Convolution and Pooling as an Infinitely Strong Prior 35
6.5 Structured Outputs 35

7 Practical Methodology **37**

II Computer Vision Practice **39**

8 Classification **41**
8.1 train.py 41
8.2 models/LeNet.py 43
8.3 Error and Anylysis 44

9 Object Detection **49**
9.1 train.py 50
9.2 ssd_predict.py 52

10 Segmentation **55**

11 Generative Model **57**

III NLP Practice **59**

12 Classificatrion **61**

13 Chat **63**

Part I

Theory

Chapter 1

Introduction

In the early days of the artificial intelligence, the field rapidly tackled and solved problems that are intellectually difficult for human beings but realtively straightforward for computers – problems that can be described by a list of formal, mathematical rules. The true challenge to artificial intelligence proved to be solving the tasks that are easy for every people to perform but hard for people to describe formally – problems that we solve intuitively, but feel automatic, like recognizing spoken words or faces in images.

The solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts. By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all the knowledge that the computer needs. The hiearachy of concepts enables the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these concepts are built on top of the each other, the graph is deep, with many layers. For this reason, we call this approach to AI **deep learning**.

The first artificial intelligence projects are called the **knowledge base**. Knowledge about the world is hard-coded in a formal languages. A computer can reason automatically about statements in these formal languages using logical inference rules. However none of these project has led to a major success.

The difficulty faced by systems relying on hard-coded knowledge suggest that AI systems need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as **machine learning**.

However, the performance of machine learning algorithms depends heavily on the **representation** of the data. Each piece of information included in the representation is known as a **feature**.

For many tasks, however, it is difficult to know what features should be extracted. One solution to this problem is to use machine learning to discover not only the mapping from representation to output but also the representation itself. This approach is known as **representation learning**.

Of course, it can be very difficult to extract high-level, abstract features from raw data. When it is nearly difficult to obtain a representation as to solve the original problem, representation learning does not seem to help us.

Deep learing solves this central problem in representaion learing by introducing representaions that are expressed in terms of other, simpler representaions.

The evolution of deep learning is show in Figure 1.1:

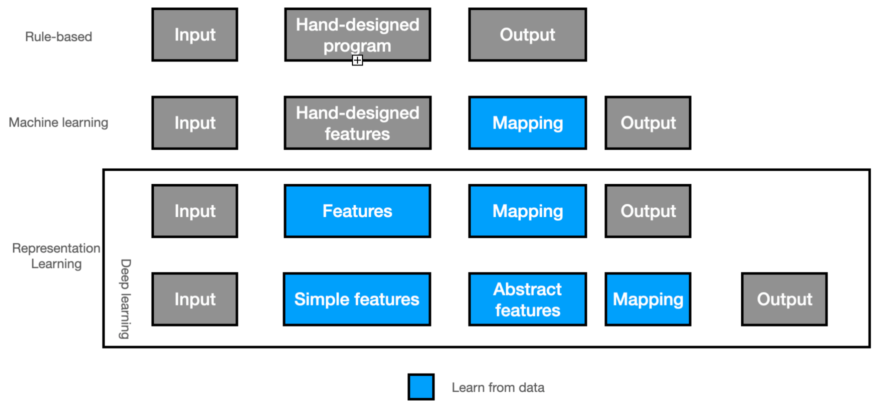


Figure 1.1: Evolution of deep learning

Chapter 2

Machine Learning Basics

Machine learning is essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions.

2.1 Learning Algorithms

A machine learning algorithm is an algorithm that is able to learn from data. But what do we mean by learning? Mitchell provides a succinct definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" Now, how does the learn happen? The model or algorithm learns by adjusting the parameters contained in it.

2.2 Capacity, Overfitting and Underfitting

The central challenge in machine learning is that out algorithm must perform well on *new, previously* – not just on which our model was trained. The ability to perform on previously unobserved inputs is called **generalization**.

Typically, when training a machine learning model, we have access to a training set; we can compute some error measure on the training set, called the **training error**; and we reduce this training error. We simply call this an optimization problem. What separate machine learning from optimization is that we want the **generalization error**, also called the **test error** to be low as well. The generalization error is defined as the expected value of the error on a new input.

How can we affect performance on the test set when we get to observe only the training set? The field of **statistical learning theory** provides some answers. If the training and the test set are collected arbitrarily, there is indeed little we can do. If we are allowed to make some **assumptions** about how the training and test set are collected, then we can make some progress.

The training and test data are generated by a probability distribution over datasets called the **data-generating process**. We typically make a set of assumptions known as the **i.i.d. assumptions**. These assumptions are that the examples in each dataset are independent from each other, and that the training set and test set are identically distributed, drawn from the same probability distributed as each other. This assumption enables us to descirbe the data-generating process with a probability distribution over a single example. The same distribution is then used to generate every train example and every test example. We call that shared underlying distribution the **data-generating distribution**, denoted p_{data}. This probability framework and the i.i.d. assumptions enables us to mathematically study the relationship between training error and test error.

One connection between training error and test error is that expected training error of a randomly selected model is equal to the expected test error of that model. When we use a machine learning algorithm, we sample the training set, the use it to choose the parameters to reduce training set error, the sample the test set. Under this process, the expected test error is greater than or equal to the expected value of training error. The factors determining how well a machine learning algorithm will perform are its ability to

1. Make the training error small.

2. Make the gap between training and test error small.

These two factors correspond to the two central challenges: **underfitting** and **overfitting**. Underfitting occurs when the model is not able to obtain a sufficient low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large.

We can control whether a model is more likely to overfit or underfit by altering its **capacity**. Informally, a model's capacity is its ability to fit a wide variety of functions.

One way to control the capacity of a learning algorithm is by choosing its hypothesis space, the set of functions that the learning algorithm is allowed to selected as being the solution. Capacity is not determined only by the choice of model. The model specifies which family of functions the learning algorithm can choose from. This is called the **representation capacity** of the model. In many cases, finding the best function within this family is a difficult optimization problem. In practice, the learning algorithm does not actually find the best function, but merely one that sinificantly reduces the training error. This additional limitations, such as the inperfection of the optimization algorithm, mean that the learning algorithm's **effective capacity** may be less than the representational capacity of the model family.

Machine learning algorithm will generally perform best when their capacity is appropriate for the true complexity of the task and the amount of training data.

The ideal model is an oracle that simply knows the true probability distribution that generate the data. Even such a model will still incur some error on many problems, because there may still be some noise in the distribution. In the case of supervised learning, the mapping from $\boldsymbol{x}$ to y may be inherently stocastic, or y may be a deterministic function that involves other variables besides those included in $\boldsymbol{x}$. The error incurred by an oracle making predictions from the true distribution $p(\boldsymbol{x}, y)$ is called the **Bayes error**.

2.2.1 The No Free Lunch Theorem

Learning theory claims that a machine learning algorithm can generalize well from a finite training set of examples. However, to logically infer a rule describing every memeber of a set, one must have information about every member of that set.

In part, machine learning avoids this problem by offering only probabilistic rules, rather than the entirely certain rules used in purely logical reasoning. Machine learning promises to find rules that are *probably* correct about *most* memebers of the set they can concern.

Unfortunately, even this does not resovle the entire problem. The **no free lunch theorem** for machine learning states that, averaged over all possible data-generating distribution, every classification algorithm has the same error when classicifying previously unobserved points. In other words, in some sense, no machine learning algorithm is universally any better than any other.

Fortunately, these results holds only when we average over *all* possible data-generating distribution. If we make assumptions about the kinds of probability distributions we encounter in real-world application, then we can design learning algorithms that perform well on these distributions.

This means that the goal of machine learning research is not to seek a universal learning algorithm or the absolute best learning algorithm. Instead, our goal is to understand what kinds of distributions are relevant to the "real world" that an AI agent experiences, and what kinds of machine learning algorithm perform well on data drawn from the kinds of data-generating distributions we care about.

2.2.2 Regularization

The no free lunch theorem implies that we must design our machine learning algorithms to perform well on a specific task. We do so by building a set of preferences into the learning algorithm.

The behavior of our algorithm is strongly affacted not just by how large we make the set of functions allowed in its hypothesis space, but by the specific identity of those functions. For example, liner regression would not perform well if we tried to use it to predict $\sin(x)$ from x. We can thus control the performance of our algorithm by choosing what kind of functions we allow them to draw solutions from, as well as by controlling the amount of these functions.

Generally, we can regularize a model that learns a function $f(\boldsymbol{x};\theta)$ by adding a penalty called a **regularizer** to the cost function. In the case of weight decay, the relularizer is $\Omega(w) = w^{\top}w$.

Expressing preferences for one function over anohter is a more general way of controlling a model's capacity.

Regularization is any modification we make to a learnining algorithm that is intended to reduce its generalization error but not its training error.

The no free lunch theorem has made it clear that there is no best machine learning algorithm, and in particular, no best form of regularization. Instead we must choose a form of regularization that is well suited to the particular task we want to solve. The philosophy of deep learning is that a wide range of tasks may all be solved effectively using very general-purpose forms of regularization.

2.3 Hyperparameters and validation sets

Most machine learning algorithms have hyperparameters, setting that we can use to control the algorithm's behavior. The values of hyperparameters are not adapted by the learning algorithm itself.

Sometimes a setting is chosen to be a hyperparameter that the learning algorithm does not learn because the setting is difficult to optimize. Most frequently, the setting must be hyperparameter because it is not appropriate to learn that hyperparameter on the training set. This applies ot all hyperparameters that control model capacity. If learned on the trainning set, sunch hyperparameters would always choose the maximum possible capacity, resulting in overfitting.

To solve this problem, we need a **validation set** of examples that the training algorithm does not observe. It is important that the test examples are not used in any way to make choice about the model, including hyperparameters. Therefore, we always construct the validation set from the training data. Specifically, we split the training data into tow disjoint subsets. The subset of data used to learn the parameters is still typically called the training set. The subset of data used to guide the selection of hyperparameters is called the validation set. Typically, one uses about 80 percent of the training data for training and 20 percent for validation.

2.4 Estimators, bias and variance

Foundational concepts such as parameter estimation, bias and variance are useful to formally characterize notions of generalization, underfitting and overfitting.

2.4.1 Point estimation

Point estimation is the attempt to provide the single "best" prediction of some quantity of interest. To distinguish estimators of parameters from their true value, our convention will be to denote a point estimate of a parameter $\boldsymbol{\theta}$ by $\hat{\boldsymbol{\theta}}$.

Let $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ be a set of m independent and identically distributed (i.i.d.) data points. A **point estimator** or **statistic** is any function of the data:

$$\hat{\boldsymbol{\theta}} = g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}). \tag{2.1}$$

While almost any function thus qualifies as an estimator, a good estimator is a function whose output is close to the true underlying $\boldsymbol{\theta}$ that generated the training data.

2.4.2 Bias

The bias of an estimator is defined as

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}, \tag{2.2}$$

where the expectation is over the data (seen as samples from a random variable) and $\boldsymbol{\theta}$ is the true underlying value of $\boldsymbol{\theta}$ used to define the data-generating distribution. An estimator $\hat{\boldsymbol{\theta}}$ is said to be **unbiased** if $\text{bias}(\hat{\boldsymbol{\theta}}_m) = 0$. An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be **asymptotically unbiased** if $\lim_{m\to\infty} \text{bias}(\hat{\boldsymbol{\theta}}_m) = 0$.

2.4.3 Variance and standard error

The variance of an estimator is simply the variance

$$\text{Var}(\hat{\boldsymbol{\theta}}) \tag{2.3}$$

where the random variable is the training set. Alternately, the square root of the variance is called the **standard error**, denoted $\text{SE}(\hat{\boldsymbol{\theta}})$.

The variance, or the standard error, of an estimator provides a measure of how we would expect the estimate we compute from data to vary as we independently resample the dataset from the underlying data-generating process.

When we compute any statistic using a finite number of samples, our estimate of the true underlying parameter is uncertain.

2.4.4 Consistency

We usually wish that, as the number of data points m in our dataset increase, our point estimates converge to the true value of the corresponding parameters. More formally, we would like that

$$\text{plim}_{m\to\infty}\hat{\boldsymbol{\theta}}_m = \boldsymbol{\theta}. \tag{2.4}$$

The symbol plim indicates convergence in probability, meaning that for any $\epsilon > 0$, $P(|\hat{\boldsymbol{\theta}}_m - \boldsymbol{\theta}| > \epsilon) \to 0$ as $m \to \infty$. The condition described by equation 2.4 is known as **consistency**.

2.5 Maximum likelihood estimation (*)

Rather than guessing that some function might make a good estimator and then analyzing its bias and variance, we would like to have some principle from which we can derive specific functions that are good estimators for different models.

The most common such principle is the maximum likelihood estimation.

Consider a set of m examples $\mathbb{X} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ drawn independently from the true but unknown data-generating distribution $p_{data}(\mathbf{x})$.

Let $p_{model}(\mathbf{x}; \theta)$ be a parametric family of probability distributions over the same space indexed by θ.

The maximum likelihood estimator for θ is then defined as:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}) = \arg\max\nolimits_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\text{model}}(x^{(i)}; \boldsymbol{\theta}) \tag{2.5}$$

To get a convenient but equivalent optimization problem:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{model}}(x^{(i)}; \boldsymbol{\theta}) \tag{2.6}$$

Because the arg max does not change when we rescale the cost function, we can divide by m to obtain a version of the criterion that is expressed as expectation with respect to the empirical distribution $\hat{p}_{data}$ defined by the training data:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{x\sim\hat{p}_{data}} \log p_{\text{model}}(x; \boldsymbol{\theta}) \tag{2.7}$$

One way to interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution $\hat{p}_{data}$, defined by the training set and the model distribution, with the degree of dissimilarity between the two measured by the KL divergence. The KL divergence is given by:

$$D_{KL}(\hat{p}_{data}||p_{data}) = \mathbb{E}_{\mathrm{x}\sim\hat{p}_{data}}[\log \hat{p}_{data}(\boldsymbol{x}) - \log p_{model}(\boldsymbol{x})] \tag{2.8}$$

2.5.1 Properties of maximum likelihood

The main appeal of the maximum likelihood estimator is that it can be shown to be the best estimator asymptotically, as the number of examples $m \to \infty$, in terms of its rate of convergence as m increases.

Under appropriate conditions, the maximum likelihood estimator has the property of consistency. These conditions are as follows:

- The true distribution p_{data} must lie within the model family $p_{model}(\cdot;\theta)$.
- The true distribution p_{data} must correspond to exactly one value of θ.

2.6 Bayesian statistics

In frequentist perspective, the true parameter value θ is fixed but unknown, while the point estimator $\hat{\theta}$ is a random variable on account of it being a function of the dataset. The Bayesian perspective on statistic is quite different. The Bayesian uses probability to reflect degrees of certainty in states of knowledge. The dataset is directly observed and so is not random. On the other hand, the true parameter θ is unknown or uncertain and thus is represented as a random variable.

Before observing the data, we represent out **knowledge** of θ using the **prior probability distribution**, $p(\theta)$. Generally, the machine learning practitioner selects a prior distribution that is quite broad (i.e. with high entropy) to reflect a high degree of uncertainty in the value of θ before observing any data. Now consider that we have a set of data samples $\{x^{(1)}, \cdots, x^{(m)}\}$. We can recover the effect of data on our belief about θ by combining the data likelihood $p(x^{(1)}, \cdots, x^{(m)}|\theta)$ with the prior via Bayes' rule:

$$p(\theta|x^{(1)}, \cdots, x^{(m)}) = \frac{p(x^{(1)}, \cdots, x^{(m)}|\theta)p(\theta)}{p(x^{(1)}, \cdots, x^{(m)})} \tag{2.9}$$

2.7 Stochastic Gradient Descent

Nearly all of deep learning is powered by one very important algorithm: **stochastic gradient descent** (SGD). A recurring problem in machine learning is that large training sets are neccessary for good generalization, but large training sets are also more computationally expensive.

The cost function used by a machining learning algorithm often decomposes as a sum over training examples of some per-example loss function. For example, the negative conditional log-likelihood of the training data can be written as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathrm{x},y\sim\hat{p}_{data}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}), \tag{2.10}$$

where L is the per-example loss $L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y|\boldsymbol{x}; \boldsymbol{\theta})$.

For these additive cost function, gradient requires computing

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \tag{2.11}$$

The computational cost of this operation is $O(m)$. As the training set size grows to billions of examples, the time to take a single gradient step becomes prohibitively long.

The insight of SGD is that the gradient is an expectation. The expectation may be approximately estimated using a small set of samples. Specifically, on each step of the algorithm, we can sample a **minibatch** of examples $\mathbb{B} = \{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(m)}\}$ drawn uniformly from the training set. The minibatch size m' is typically chosen to be a relatively small number of examples, ranging from one to a few hundred.

The estimator of the gradient is formed as

$$\boldsymbol{g} = \frac{1}{m'}\nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) \tag{2.12}$$

using examples from the minibatch $\mathbb{B}$. The stocastic gradient descent algorithm then follows the estimated gradient donwhill:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}, \tag{2.13}$$

where ϵ is the learning rate.

2.8 Building a machine learning algorithm

Nearly all deep learning algorithms can be described as particular instances of a fairly simple recipe:

- a specification of a dataset
- a cost function
- an optimization procedure
- a model

2.9 Challenges Motivating Deep Learning

The development of deep learning was motivated in part by the failure of traditional algorithms to generalize well on AI tasks, such as recognizing speech or recognizing objects.

2.9.1 The Curse of dimensionality

Many machine learning problems become exceedingly difficult when the number of dimenions in the data is high. This phonomenon is known as the **curse of dimensionality**. One challenge posed by the curse of dimensionality is a statistical challenge. A statistical challenge arises becuase the number of possible configurations of x is much large than the number of training examples.

2.9.2 Local constancy and smoothness regularization

To generalize well, machine learning algorithms need to be guided by prior beliefs about what kind of function they should learn. Among the most widely used implicit "priors" is the **smoothness prior**, or local constancy prior. This prior states that the function we learn should not change very much within a small region.

There are many different ways to implicitly or explicitly express a prior belief that the learned function should be smooth or locally constant. All these different methods are designed to encourage the learning process to learn a function f^* that satisfies the condition

$$f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \epsilon) \tag{2.14}$$

2.10 Manifold Learning

An important concept underlying many ideas in machine learning is that of a manifold. A **manifold** is a connected region. Mathematically, it is a set of points associated with a neighborhood around each point. The concept of a neighborhood surrounding each point implies the existence of transformations that can be applied to move on the manifold from one position to a neighboring one.

Many machine learning problems seem hopeless if we expect the machine learning algorithm to learn function with interesting variations across all of $\mathbb{R}^n$. **Manifold learning** algorithms surmount this obstacle by assuming the most of $\mathbb{R}^n$ consists of invalid inputs,and that interesting inputs occurs only along a collection of manifolds contaning a small subset of points, with interesting variations in the output of the learned function occuring only along directions that lie on the manifold, or with interesting variations happening only when we move from one manifold to another.

The assumption that the data lies along a low-dimensional manifold may not always be correct or useful. We argue that in the context of AI tasks, such as those that involve processing images, sounds, or text, the manifold assumptions is at least approximately correct.

The evidence in favor of this assumptions consists of tow categories of observations:

1. The probability distribution over images, text strings, and sounds that occur in real life is highly concentrated.

2. We can imagine such neighborhoods and transformations, at least informally.

Chapter 3

Deep Feedforward Networks

Deep feedforward networks, also called **feedforward neural networks**, or **multilayer perceptions** (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f^*. A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the value of the parameter $\boldsymbol{\theta}$ that result in the best function approximation.

These models are called **feedforward** because information flows through the function being evaluated from $\boldsymbol{x}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{y}$. Feedforward neural network are called **networks** becuase they are typically represented by composing together many different functions.

3.1 Gradient-Based Learning

The nonlinearity of a neural network causes most intersting loss functions to become nonconvex. This means that neural networks are usually trained by using iterative, gradient-based optimizer that merely drive the cost function to a very low value. Convex optimization converges starting from any initial parameters. Stochastic gradient descent applied to nonconvex loss functions has no such convergence guarantee and is sensitive to the values of the initial parameters.

Chapter 4

Regularization for Deep Learning

A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strateries used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strateries are known collectively as **regularization**.

In practice, an overly complex model family does not necessarily include the target function or the true data generating process, or even a close approximation of either. We almost never have access to the true data generating process so we can never know for sure if the model family being estimated includes the generating process or not. However, most applications of deep learning algorithms are to domains where the true data generating process is almost certainly outside the model family. Deep learning algorithms are typically applied to extremely complicated domains such as images, audio sequences and text, for which the true generation process essentially involves simulating the entire universe. To some extent, we are always trying to fit a square peg (the data generating process) into a round hole (our model family).

What this means is that 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, we might find – and indeed in practical deep learning scenarios, we almost always do find – that the best fitting model is a larger model that has been regularized appropriately.

4.1 Parameter Norm Penalties

Many regularization approaches are based on limiting the capacity of models by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J.

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}) \tag{4.1}$$

4.1.1 L^2 Parameter Regularization

The L^2 parameter norm penalty commonly known as **weight decay**. This regularization strategy derives the weights closer to the original by adding a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{w}||_2^2$ to the objective function.

4.1.2 L^1 Regularization

L^1 regularization on the model parameter $\boldsymbol{w}$ is defined as

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |w_i|. \tag{4.2}$$

4.2 Dataset Augmentation

The best way to make a machine learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set.

Dataset augmentation has been a particular effective technique for a specific classification problem: object recognition.

4.3 Noise Robustness

For some models, the addition of noise with infinitesimal variance at the input of hte model is equivalent to imposing a penalty on the norm of the weights. In the general case, it is important to remember that noise injection can be much more powerful than simply shrinking the parameters, especially when the noise is added to the hidden units.

4.4 Semi-Supervised Learning

In the paradigm of semi-supervised learning, both unlabeled examples from $P(\mathrm{x})$ and labeled examples from $P(\mathrm{x}, \mathrm{y})$ are used to estimate $P(\mathrm{y}|\mathrm{x})$ or predict y from x.

In the context of deep learning, semi-supervised learning usually refers to learning a representation $\boldsymbol{h} = f(\boldsymbol{x})$. The goal is to learn a representation so that examples from the same class have similar representations.

4.5 Multitask Learning

Multitask learning is a way to improve generalization by pooling the examples (which can be seen as soft constaints imposed on the parameters) arising out of several tasks. In the same way that additional tranining examples put more pressure on the parameters of the model toward values that generalize well, when part of a model is shared across tasks, that part of the model is more constrained toward good values (assuming the sharing is justified), often yielding better generalization.

4.6 Early Stopping

Early stopping is used to avoid overfit.

The algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations. This strategy is known as **early stopping**. It is probably the most commonly used form of regularization in deep learning.

Early stopping is a unobtrusive form of regularization, in that it requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values. Early stopping requires a validation set, which means some traning data is not fed to the model.

4.7 Parameter Tying and Parameter Sharing

Sometimes we want to express our prior knowledge about suitable values of the model parameters. Sometimes we might not know precisely what values the parameters should take, but we know, from knowledge of the domain and model archtecture, that there should be some dependencies between the model parameters.

A common type of dependency that we foten want to express is that certain parameters should be close to one another. For example: We have model A with parameters $\boldsymbol{w}^{(A)}$ and model B with parameters $\boldsymbol{w}^{(B)}$. We believe the model parameters should be close to each other: $\forall i w_i^{(A)}$ should be close to $w_i^{(B)}$. We can use a parameter norm penalty of the form $\Omega(\boldsymbol{w}^{(A)}, \boldsymbol{w}^{(B)} = ||\boldsymbol{w}^{(A)} - \boldsymbol{w}^{(B)}||_2^2$. Here we use an L^2 penalty, but other choices are also possible. This is called **parameter typing**.

While a parameter norm penalty is onw way to regularize parameters to be close to one another, the more popular way is to use constraints: *to force stes of parameters to be equal*. This method of regularization is often referred to as **parameters sharing**, because we interpret the various models or model components as sharing a unique set of parameters.

4.8 Sparse Representations

Weightdecay acts by placing a penalty directly on the model parameter. Another strategy is to place a penalty on the activation of the units in a neural network, encouraging their activation to be sparse. This indirectly imposes a complicated penalty on the model parameters.

4.9 Bagging and Other Emsemble Methods

Bagging (short for bootstrap aggregating) is a technique for reducing generalization erry by combining several models. The idea is to train several different models separately, then have all the models vote on the output for test examples. This is a exampe of a general strategy in machine learning called model averaging. Techniques employing this strategy are known as **ensemble methods**.

The reason that model averaging works is that different models will usually not make all the same error on the test set.

4.10 Adversarial Training

Szegedy et al. (2014b) found that even neural network that perform at human level accuracy have a nearly 100 percent error rate on examples that are intentionally constructed by using an optimization procedure to search for an input $x^{'}$ near a data point x such that the model output is very different at $x^{'}$. In many cases, $x^{'}$ can be so similar to x that a human observe cannot tell the difference between the original example and the **adversarial example**, but the network can make highly different predictions.

We can reduce the error rate on the original i.i.d. test set via adversarial training – training on adversarially perturbed examples from the training set.

Chapter 5

Optimization for Training Deep Models

Chapter 6

Convolutional Networks

Convolutional networks, also known as **convolutional neural networks** or CNNs, are specialized kind of neural network for processing data that has a known **grid-like** topology. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

6.1 The Convolution Operation

$$s(t) = \int x(a)w(t-a)da. \tag{6.1}$$

This operation is called **convolution**. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t). \tag{6.2}$$

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the **input**, and the second argument (int this example, the function w) as the **kernel**. The output is sometimes referred to as the **feature map**.

If we assume that x and w are defined only on integer t, we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{6.3}$$

We often use convolutions over more than one axis at a time. For example, if we use a two-dimensinal image I as our input, we probably also want to use a two-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{6.4}$$

Convolution is commulative, meaning we can equivalently write

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n). \tag{6.5}$$

Usually the latter formula is more stratghtforward to implement in a machine learning library, because there is less variation in the range of valid values of m and n.

The commulative property of convolution arises becuase we have fipped the kernel relative to the input, in the sense that as m increase, the index into the input increase, but the index into the kernel decrease. The only reason to flip the kernel is to obtain the commulative property. While the commulative property is useful for writting proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the **cross-correlation**, which is the same as convolution but without flippling the kernel:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n). \tag{6.6}$$

Many machine learning libraries implement cross-correlation but call it convolution.

6.2 Motivation

Convolution leverages three important ideas:

- sparse interaction.
- parameter sharing.
- equivariant representations.

6.2.1 Sparse interaction

Tradition neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. Convolutional networks typically have sparse interactions. This is accomplished by making the kernel smaller than the input.

6.2.2 Parameter sharing

Parameter sharing refers to use the same parameter for more than one function in a model. In a traditional nerual net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. In a convolutional neural net. each member of the kernel is used at every postion of the input. The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set.

6.2.3 Equivariant representations

In the case of convolution, the particular form of a parameter sharing causes the layer to have a property called **equivariance** to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translate the input, that is, shifts it, then the convolution function is equivalent to g.

For images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representations will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input location. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network, The same edges appear more or less everywhere in the image, so it is practical to share parameters across the intire image.

Convolution is not naturally equivalent to some other transformation, such as changes in the scale or ratation of a image.

6.3 Pooling

A typical layer of a convolutional network consists of three stages:

1. convolution stage: affine transform
2. detector stage: nonlinearty
3. pooling stage

In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activations is run through a nonlinear activation function, such as the rectified linear activation function. In the third stage, we use a pooling function to modify the output of the layer further.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling oeration reports the maximum otuput within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L^2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

In all cases, pooling helps to make the representation approximately invariant to small translations of the input. Invariant to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Invariance to local translation can be a useful property if we care about whether some feature is present than exactly where it is. For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. In ohter example, it is more important to perserve the location of a feature. For example, if we want to find a corner defined by two edges meeting at a specified orientation, we need to perserve the location of the edge well enough to test whether they meet.

The use of pooling can be viewed as adding a infinitly strong prior that the function the layer learns must be invariant to small translations. When this assumption is correct, it can greatly improve the statistical efficiency of the network.

6.4 Convolution and Pooling as an Infinitely Strong Prior

A probability distribution over the parameters of a model encodes our beliefs about what models are reasonable, before we see any data.

Prior can be considered weak or strong depending on how concertrated the probability density in the prior is. A weak prior is a prior distribution with high entropy. A strong prior is a prior distribution with low entropy. An infinitely strong prior places zero probability on some parameters and says that these parameter values are completely forbidden, regardless of how much support the data give to these values.

We can imagine a convolutional net as being similar to a fully connected net, but with an infinitely strong prior over its weights. This infinitely strong prior says that the weights for one hidden unit must be identical to the weights of its neighbor, but shifted in space. The prior also says that the weights must be zero, except for in the small, spatially contiguous receptive field assigned to that hidden unit. Overall, we can think of the use of convolution as introducing an infinitely strong prior probability distribution over the parameters of a layer. This prior says that the function the layer should learn contains only local interactions and is equivariant to translation. Likewise, the use of pooling is an infinitely strong prior that each unit should be invariant to small translations.

6.5 Structured Outputs

Convolutional networks can be used to output a high-dimensional, structured object, rather than just predicting a class label for a classification task or a real value for a regression task. Typically this object is just a tensor, emitted by a standard convolutional layer. For example, the model might emit a tensor S, where $S_{i,j,k}$ is the probability that pixel (j, k) of the input to the network belongs to class i. This allows the model to label every pixel in an image and draw precise masks that follow the outlines of individual objects. This the basis of the segmentation model.

In the kinds of architectures typically used for classification of a single object in an image, the greatest reduction in the spatial dimensions of the network comes from using pooling layers with large stride. To produce an output map of similar size as the input, one can:

- avoid pooling altogether
- emit a lower-resolution grid of labels
- use a pooling operator with unit stride

Chapter 7

Practical Methodology

Successfully applying deep learning techniques requires more than a good knowledge of what algorithms exist and the principles that explain how they work. A good machine learning practitioner also needs to know how to choose an algorithm for a particular application and how to monitor and respond to feedback obtained from experiments in order to improve a machine learning system. Practitioners need to decide whether to gather more data, increase or decrease model capacity, add or remove regularizing features, improve the optimization of a model, improve approximate inference in a model, or debug the software implementation of the model.

- a good knowledge of what algorithms exist
- the principles that explain how they work
- how to choose an algorithm for a particular application
- how to monitor and respond to feedback
  - whether to gather more data
  - increase or decrease model capacity
  - add or remove regularizing features
  - improve the optimization
  - improve approximate inference
  - debug the software

Some practical process:

1. Determine your goal – what error metric to use, and your target value for this error metric. These goals and error metrics should be driven by the problem that the application is intended to solve.

2. Establish a working end-to-end pipeline as soon as possible, including the estimation of the appropriate performance metrics.

3. Instrument the system well to determine bottlenecks in performance. Diagnose which components are performing worse than expected and whether poor performance is due to overfitting, underfitting, or a defect in the data or software.

4. Repeatedly make incremental changes such as gathering new data, adjusting hyperparameters, or changing algorithms, based on specific findings from your instrumentation.

Part II

Computer Vision Practice

Chapter 8

Classification

The project overview is show in Figure 8.1:

8.1 train.py

```
#!/usr/bin/env python3
"""
@project: cv_classification
@file: train
@author: mike
@time: 2021/3/3

@function:
"""
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer
from keras.optimizers import SGD
from keras import backend as K
from models.LeNet import LeNet
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
import numpy as np

# Step 1: load the dataset
print('[INFO] loading MNIST...')
dataset = datasets.load_digits()
# data
# target
# frame
# feature_names
# target_names
# images
# DESCR
data = dataset.data
target = dataset.target

# Step 2: process
# Step 2-1: scale the input arrange
data = data / 16.0 # from [0-16] to [0-1]
target = target.astype('int')

# Step 2-2: change shape
```

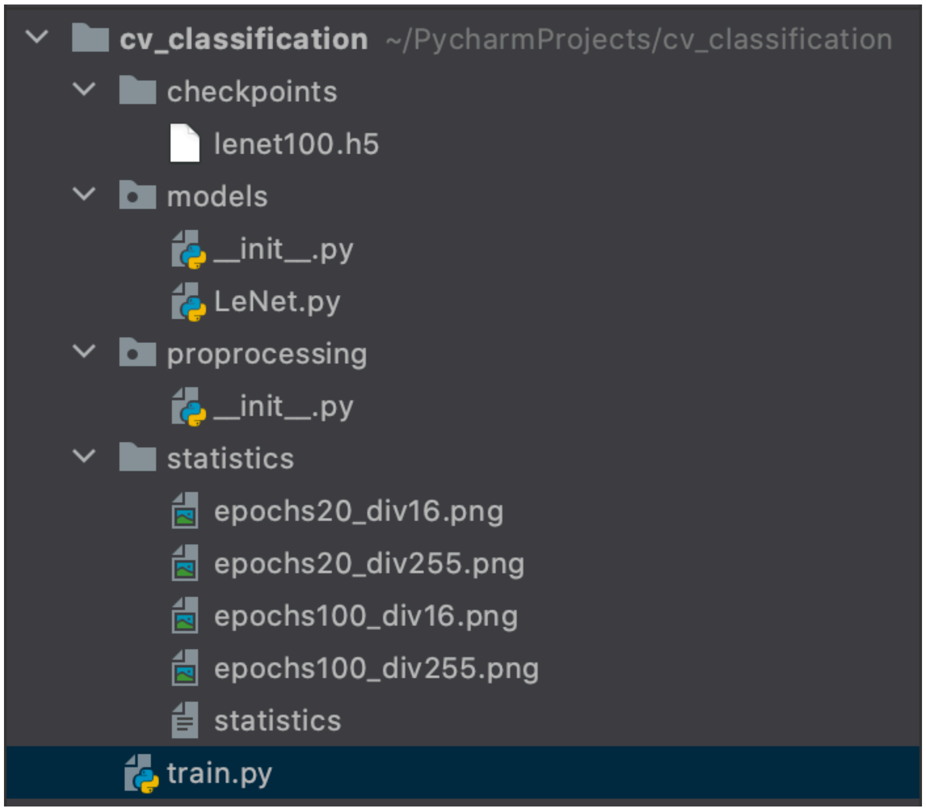

```
cv_classification ~/PycharmProjects/cv_classification
    checkpoints
        lenet100.h5
    models
        __init__.py
        LeNet.py
    proprocessing
        __init__.py
    statistics
        epochs20_div16.png
        epochs20_div255.png
        epochs100_div16.png
        epochs100_div255.png
        statistics
    train.py
```

Figure 8.1: Compute Version Classification Project Overview

```
if K.image_data_format() == "channels_first":
    data = data.reshape(data.shape[0], 1, 8, 8)
else:
    data = data.reshape(data.shape[0], 8, 8, 1)

# Step 2-3: train, test split (this is optional)
# If the train test is split before the loading,
# there is no more split
train_x, test_x, train_y, test_y = train_test_split(data, target, test_size=0.2, random_state=16)

# Step 2-4: convert the labels from integers to vectors
lb = LabelBinarizer()
train_y = lb.fit_transform(train_y)
test_y = lb.transform(test_y)

# Step 3: initialize the optimizer and model
print('[INFO] compiling model...')
opt = SGD(learning_rate=0.01)
model = LeNet(width=8, height=8, depth=1, classes=10)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=['accuracy'])

# Step 4: train the model
print('[INFO] training network...')
epochs = 100
# fit method can only be used for small dataset, for large dataset use the fit_generate instead
```

```
H = model.fit(train_x, train_y, batch_size=128, epochs=epochs, verbose=2, validation_data=(test_x, test_y))

# Step 5: evaluate the network
print("[INFO] evaluating network...")
predictions = model.predict(test_x, batch_size=128)

# step 6: save the model
model.save('checkpoints/lenet100.h5')

# Step 7: plot the train process
plt.style.use('ggplot')
plt.figure()
plt.plot(np.arange(0, epochs), H.history['loss'], label='train_loss')
plt.plot(np.arange(0, epochs), H.history['val_loss'], label='val_loss')
plt.plot(np.arange(0, epochs), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, epochs), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()

# Step 8: write the statistics to a file (or just print)
statistics = classification_report(test_y.argmax(axis=1),
                                    predictions.argmax(axis=1),
                                    target_names=[str(x) for x in lb.classes_])
with open('statistics/statistics', 'w') as fh:
    for line in statistics:
        fh.write(line)
```

8.2 models/LeNet.py

```
#!/usr/bin/env python3
"""
@project: cv_classification
@file: LeNet
@author: mike
@time: 2021/3/3

@function:
"""
from keras.models import Sequential
from keras import backend as K
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.layers.core import Activation, Flatten, Dense


def LeNet(width, height, depth, classes):
    model = Sequential()
    input_shape = (height, width, depth)

    if K.image_data_format() == 'channels_first':
        input_shape = (depth, height, width)

    model.add(Conv2D(20, (3, 3), padding='same', input_shape=input_shape))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    model.add(Conv2D(50, (3, 3), padding='same'))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    model.add(Flatten())

    model.add(Dense(500))
    model.add(Activation('relu'))

    model.add(Dense(classes))
    model.add(Activation('softmax'))

    return model
```

8.3 Error and Anylysis

At first, the division value used is 255.0 (train.py line 35). Normally, this should make sense, becuase the value of image points lay in [0,255]. The output is as shown in Figure 8.2 (epochs=20) and 8.3 (epochs=100).

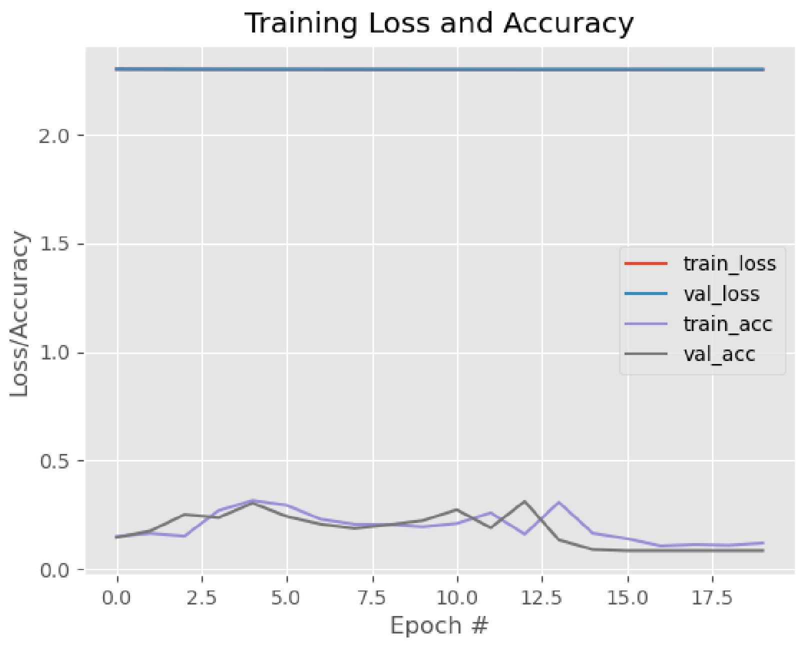


Figure 8.2: Divide 255 and epochs=20

After diving into the dataset, I found that the maximum value is 16. After changing the division to 16, the result is shown in Figure 8.4.

From the Figue 8.4 we can see that the epochs is too small. After chaning the epochs to 100, the result is shown inf Figure 8.5.

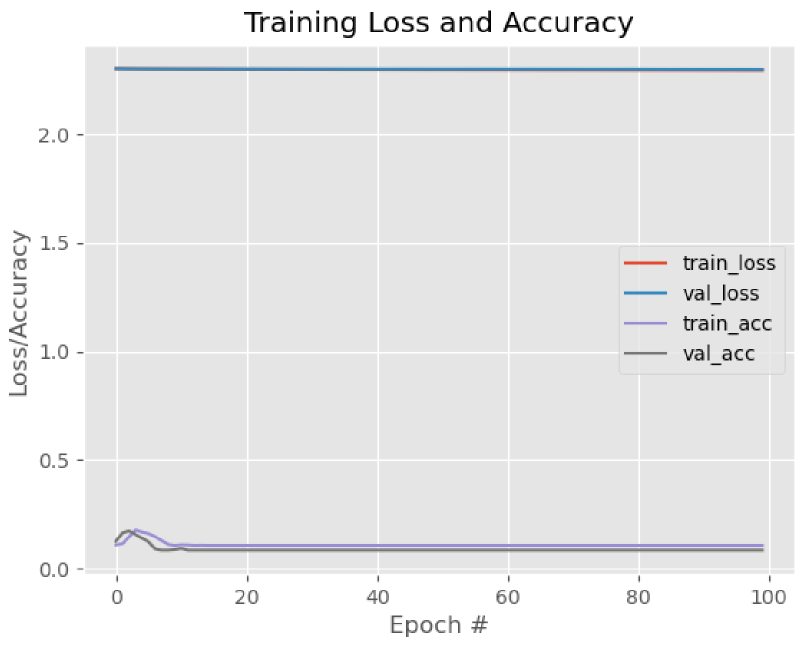


Figure 8.3: Divide 255 and epochs=100

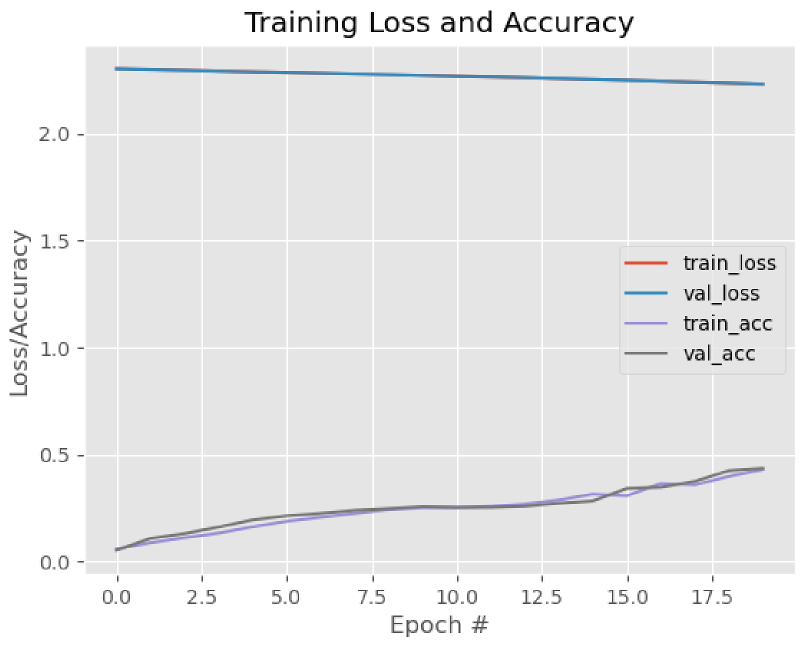


Figure 8.4: Divide 16 and epochs=20

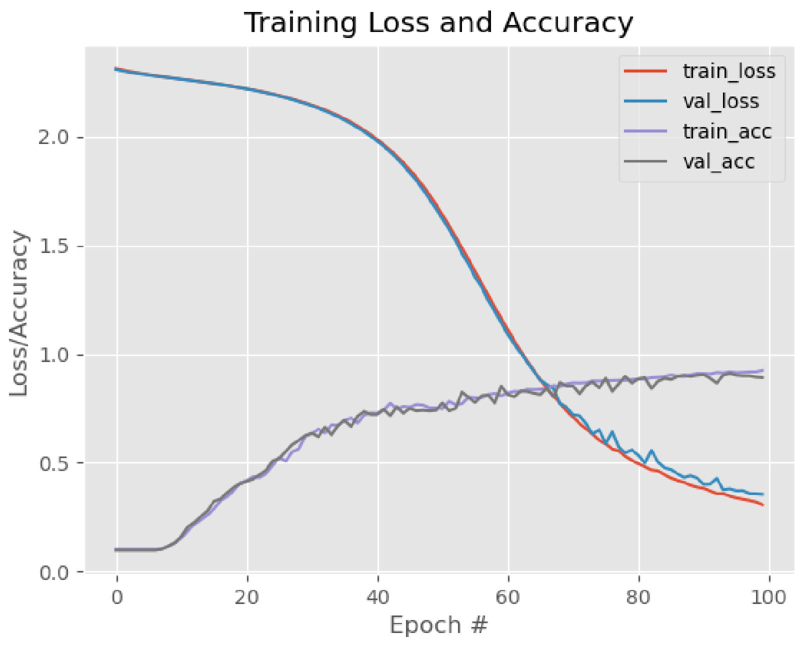


Figure 8.5: Divide 16 and epochs=100

Chapter 9

Object Detection

The object dection model used here is the SSD[1].

The project is the rewrite and simplify the SSD project on github. Here is the project structure show in Figure 9.1:

The project is on the github: https://github.com/mikechyson/object_detection.

The overview is that:

1. Use 7 convolution layers to extract feature maps.
2. From layr 4, 5, 6 and 7, extract classes layers, boxes layers and anchors layers.
3. Reshape classes layers, boxes layers and anchors layers.
4. Concatenate classes layers from 4, 5, 6 and 7 convolution layers.
5. Concatenate boxes layers from 4, 5, 6 and 7 convolution layers.
6. Concatenate anchors layers from 4, 5, 6 and 7 convolution layers.
7. Concatenate the contenated classes, boxes and anchors into a whole tensor.
8. Encoder the lables to the same tensor shape with the whole tensor.
9. In training process, compute the loss between the whole tensor and encoded label tensor.
10. In predicting process, decode the whole tensor into classes and boxes tensor.

[1]https://arxiv.org/pdf/1512.02325.pdf

9.1 train.py

```
import keras.backend as K
from models.ssd7 import build_ssd7
from keras.optimizers import Adam
from loss.loss import SSDLoss
from generator.data_generator import DataGenerator
from augmentation.constant_input_size_chain import DataAugmentationConstantInputSize
from encoder_decoder.input_encoder import SSDInputEncoder
from keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau, CSVLogger
import math
import matplotlib.pyplot as plt
import numpy as np
from encoder_decoder.output_decoder import decode_detections

################################################################################
# Set the configs
img_height = 300
img_width = 480
img_channels = 3
# Set this to your preference (maybe ‘None‘).
# The current settings transform the input pixel values to the interval ‘[−1,1]‘.
intensity_mean = 127.5
intensity_range = 127.5
n_classes = 5 # Number of positive classes
# An explicit list of anchor box scaling factors.
# If this is passed, it will override ‘min_scale‘ and ‘max_scale‘.
scales = [0.08, 0.16, 0.32, 0.64, 0.96]
aspect_ratios = [0.5, 1.0, 2.0] # The list of aspect ratios for the anchor boxes
two_boxes_for_ar1 = True # Whether or not you want to generate two anchor boxes for aspect ratio 1
steps = None # In case you’d like to set the step sizes for the anchor box grids manually; not recommended
offsets = None # In case you’d like to set the offsets for the anchor box grids manually; not recommended
clip_boxes = False # Whether or not to clip the anchor boxes to lie entirely within the image boundaries
variances = [1.0, 1.0, 1.0, 1.0] # The list of variances by which the encoded target coordinates are scaled
normalize_coords = True # Whether or not the model is supposed to use coordinates relative to the image size
batch_size = 16

################################################################################
# Create the model
K.clear_session() # Clear previous models from memory.
model = build_ssd7(image_size=(img_height, img_width, img_channels),
                   n_classes=n_classes,
                   mode=’training’,
                   l2_regularization=0.0005,
                   scales=scales,
                   aspect_ratios_global=aspect_ratios,
                   aspect_ratios_per_layer=None,
                   two_boxes_for_ar1=two_boxes_for_ar1,
                   steps=steps,
                   offsets=offsets,
                   clip_boxes=clip_boxes,
                   variances=variances,
                   normalize_coords=normalize_coords,
                   subtract_mean=intensity_mean,
                   divide_by_stddev=intensity_range)
print(model.summary())

################################################################################
# Compile the model
adam = Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e−08, decay=0.0)
ssd_loss = SSDLoss(neg_pos_ratio=3, alpha=1.0)
model.compile(optimizer=adam, loss=ssd_loss.compute_loss)

################################################################################
# Set up the data generators for the training
train_dataset = DataGenerator(load_images_into_memory=True,
                              hdf5_dataset_path=’dataset/dataset_udacity_traffic_train.h5’)
```

```
val_dataset = DataGenerator(load_images_into_memory=True,
                                hdf5_dataset_path='dataset/dataset_udacity_traffic_val.h5')

# Get the number of samples in the training and validation datasets.
trail_dataset_size = train_dataset.get_dataset_size()
val_dataset_size = val_dataset.get_dataset_size()
print(f'Number of images in the training dataset : \t{trail_dataset_size:>6}')
print(f'Number of images in the validation dataset: \t{val_dataset_size:>6}')

###################################################################################################
# Instantiate an encoder that can encode ground truth labels into the format needed by the SSD loss function.
# The encoder constructor needs the spatial dimensions of the model predictor layers to create the anchor boxes.
predictor_sizes = [
    model.get_layer('classes4').output_shape[1:3],
    model.get_layer('classes5').output_shape[1:3],
    model.get_layer('classes6').output_shape[1:3],
    model.get_layer('classes7').output_shape[1:3]
]
print(f'predictor_sizes={predictor_sizes}')

ssd_input_encoder = SSDInputEncoder(img_height=img_height,
                                    img_width=img_width,
                                    n_classes=n_classes,
                                    predictor_sizes=predictor_sizes,
                                    aspect_ratios_global=aspect_ratios,
                                    two_boxes_for_ar1=two_boxes_for_ar1,
                                    steps=steps,
                                    offsets=offsets,
                                    clip_boxes=clip_boxes,
                                    variances=variances,
                                    matching_type='multi',
                                    pos_iou_threshold=0.5,
                                    neg_iou_limit=0.3,
                                    normalize_coords=normalize_coords)

###################################################################################################
# Create the generator handles that will be passed to Keras' 'fit_generator()' function.
train_generator = train_dataset.generate(batch_size=batch_size,
                                         shuffle=True,
                                         transformations=[],
                                         label_encoder=ssd_input_encoder,
                                         returns={'processed_images', 'encoded_labels'},
                                         keep_images_without_gt=False)
val_generator = val_dataset.generate(batch_size=batch_size,
                                     shuffle=False,
                                     transformations=[],
                                     label_encoder=ssd_input_encoder,
                                     returns={'processed_images', 'encoded_labels'},
                                     keep_images_without_gt=False)

###################################################################################################
# Define model callbacks.
model_checkpoint = ModelCheckpoint(
    filepath='checkpoints/ssd7_epoch-{epoch:02d}_loss-{loss:.4f}_val_loss-{val_loss:.4f}.h5',
    monitor='val_loss',
    verbose=1,
    save_best_only=True,
    save_weights_only=False,
    mode='auto',
    period=1)
csv_logger = CSVLogger(filename='logs/ssd_training_log.csv',
                       separator=',',
                       append=True)
early_stopping = EarlyStopping(monitor='val_loss',
                               min_delta=0.0,
                               patience=10,
                               verbose=1)
reduce_learning_rate = ReduceLROnPlateau(monitor='val_loss',
                                         factor=0.2,
                                         patience=8,
136                                          verbose=1,
137                                          epsilon=0.001,
138                                          cooldown=0,
139                                          min_lr=0.00001)
140 callbacks = [model_checkpoint, csv_logger, early_stopping, reduce_learning_rate]
141 
142 ####################################################################################################################
143 # Set the epochs and train.
144 initial_epoch = 0
145 final_epoch = 1000
146 steps_per_epoch = math.ceil(trail_dataset_size / batch_size)
147 
148 history = model.fit_generator(generator=train_generator,
149                               steps_per_epoch=steps_per_epoch,
150                               epochs=final_epoch,
151                               callbacks=callbacks,
152                               validation_data=val_generator,
153                               validation_steps=math.ceil(val_dataset_size / batch_size),
154                               initial_epoch=initial_epoch,
155                               verbose=1)
156 
157 ####################################################################################################################
158 # Plot the training process.
159 plt.figure(figsize=(20, 12))
160 plt.plot(history.history['loss'], label='loss')
161 plt.plot(history.history['val_loss'], label='val_loss')
162 plt.legend(loc='upper right', prop={'size': 24})
163 plt.show()
```

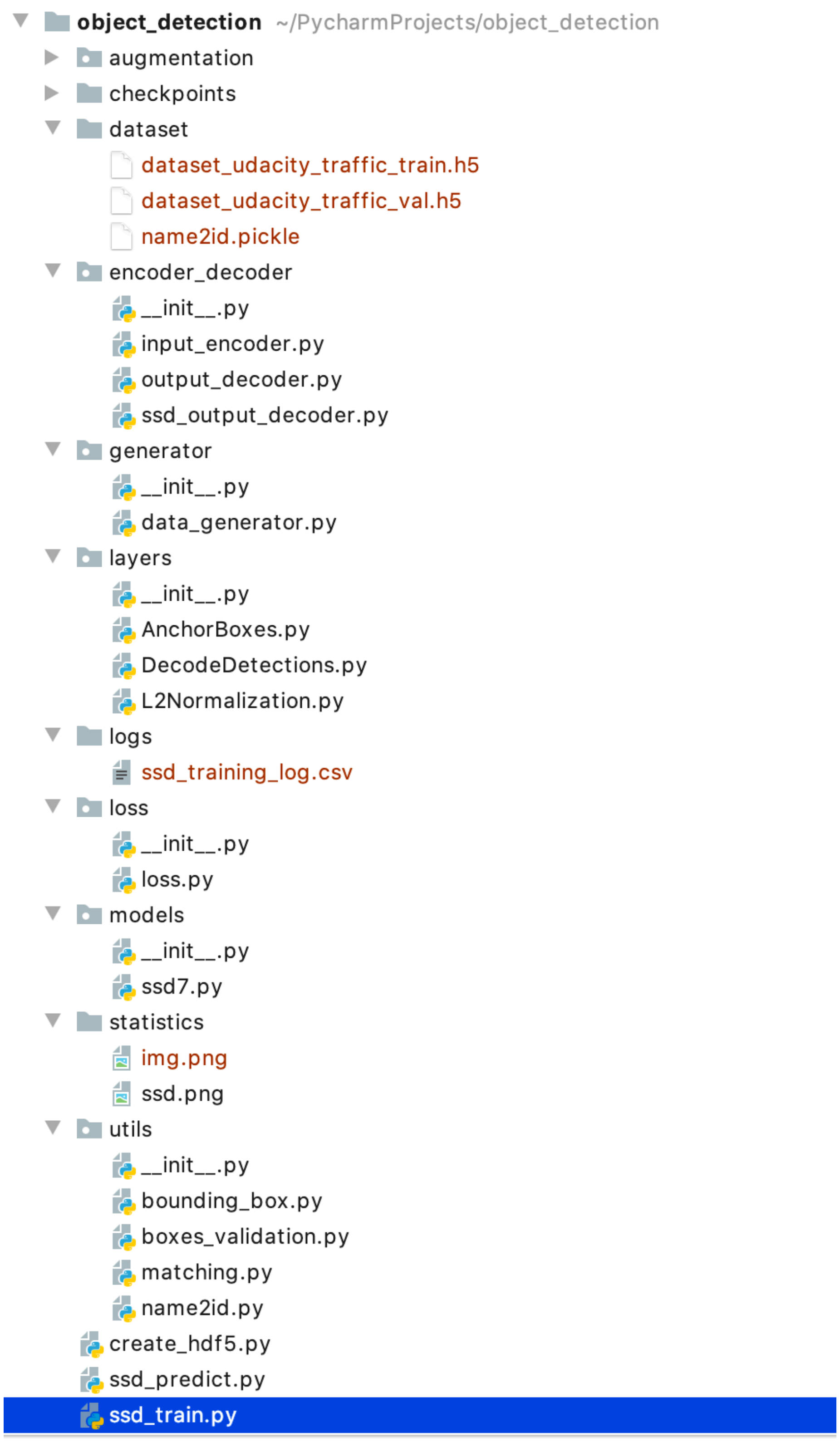
The data is stored on netdisk `https://pan.baidu.com/s/1cbqyymAHODCiM37DINhwbQ` The paasword is `snuc`.

The code to convert the data into h5 file is show in `create_hdf5.py`.

Line 148 use the `fit_generator` to fit the data into model and train the model. This is suitable for large dataset. For small dataset, you can also you the `fit` method.

9.2 ssd_predict.py

This is used to predict the object in a image with the trained model.



```
object_detection ~/PycharmProjects/object_detection
    augmentation
    checkpoints
    dataset
        dataset_udacity_traffic_train.h5
        dataset_udacity_traffic_val.h5
        name2id.pickle
    encoder_decoder
        __init__.py
        input_encoder.py
        output_decoder.py
        ssd_output_decoder.py
    generator
        __init__.py
        data_generator.py
    layers
        __init__.py
        AnchorBoxes.py
        DecodeDetections.py
        L2Normalization.py
    logs
        ssd_training_log.csv
    loss
        __init__.py
        loss.py
    models
        __init__.py
        ssd7.py
    statistics
        img.png
        ssd.png
    utils
        __init__.py
        bounding_box.py
        boxes_validation.py
        matching.py
        name2id.py
    create_hdf5.py
    ssd_predict.py
    ssd_train.py
```

Figure 9.1: SSD structure

Chapter 10

Segmentation

Chapter 11

Generative Model

Part III

NLP Practice

Chapter 12

Classificatrion

Chapter 13

Chat