Introduction to Deep Learning

Justin Sirignano*

September 9, 2018

Abstract

These are supplementary notes for the course *IE 534/CS 598: Deep Learning* at the University of Illinois at Urbana-Champaign. **Do not distribute these lecture notes to other people who are not in this course.**

*Department of Industrial & Systems Engineering, University of Illinois at Urbana Champaign

"The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves."

– John McCarthy, one of the original founders of the field of Artificial Intelligence, 1955.

"Most of all, there is a shortage of [deep learning] talent, and the big companies are trying to land as much of it as they can. Solving tough A.I. problems is not like building the flavor-of-the-month smartphone app. In the entire world, fewer than 10,000 people have the skills necessary to tackle serious artificial intelligence research [...]"

– New York Times, October 2017.

Contents

1 Introduction

Deep learning uses multi-layer neural networks to statistically model data. Neural networks are not new; they have been used as far back as the 1980s. However, they fell out of favor in the 1990s and were largely ignored by the machine learning community until the late 2000s. Then, suddenly, in just a few short years deep learning has come to dominate some of the most important areas of machine learning, such as image, text, and speech recognition. It is not an understatement to say that deep learning has revolutionized these fields, and is poised to have a significant impact on many other applications across science, engineering, medicine, and finance.

What precipitated the rapid rise of deep learning? The success of deep learning has been fueled by several interrelated factors. Deep learning uses "deep" neural networks, i.e. multi-layer neural networks with many layers. At a high level, deep neural networks are stacks of nonlinear operations, typically with millions of parameters. This yields a far more powerful and flexible model than traditional statistical and machine learning models. Over the past decade, researchers have continuously advanced the state of the art by developing larger and deeper neural network models.

It has been long understood that *larger* and *deeper* neural network models have a greater capacity to learn complex relationships. However, these deep networks, whose form is specified by parameters, must still be statistically estimated from data. Until recently, the statistical training of deep networks had been thwarted by technical challenges. Neural networks, which are highly non-convex with many local minima, present a challenging optimization landscape. Due to the complexity of the model, there is a constant danger of overfitting. Paradoxically, although networks with more layers have a greater *capacity* to learn complex functions, they are more challenging to statistically train due to the "vanishing gradient problem". Researchers have recently developed models and methods to help address these challenges.

Due to the size of deep neural networks and the amount of data they are typically trained on, significant computational resources are required. Fortunately, training can be highly parallelized on graphics processing units (GPUs). The increasing availability of such computational resources has facilitated the implementation and development of deep learning.

Deep learning is a departure from traditional statistics, which emphasizes hypothesis tests, confidence intervals, and other statistical properties. Despite the lack of such statistical guarantees, deep learning has had remarkable success. Although there are ad hoc aspects of deep learning, many of its advances are the result of careful and thoughtful design of network architectures and training methods. Examples include convolution networks for image recognition and long short-term memory (LSTM) networks for text recognition.

This course aims to cover the fundamental concepts underpinning deep learning and provide the computational methods to implement deep learning models.

1.1 Brief Review of Machine Learning

Machine learning is used to estimate models directly from data. For example, let

$$Y = f(X) + \epsilon, \tag{1.1}$$

where ϵ is a mean-zero random variable, i.e. $\mathbb{E}[\epsilon] = 0$. The random variable ϵ is called a "noise term". Given observations $(x^i, y^i)_{i=1}^N$ of (X, Y), we would like to estimate a model $f(x;\theta)$ for the function $f(x)$. The parameters θ must be learned from the data.

Machine learning falls under two broad categories: supervised learning and unsupervised learning. Supervised learning addresses the case where both input data (the X) and output data (the Y) are available. Unsupervised learning is concerned with the case when only the input data (the X) is available. We will mainly focus on supervised learning.

Different machine learning algorithms make different choices for the model $f(x;\theta)$. They may also use different methods to estimate θ. Deep learning uses multi-layer neural networks.

The success of a particular choice of model for $f(x;\theta)$ depends upon the underlying relationship $f(x)$ and the amount of data N which is available. For example, if $f(x;\theta)$ is a linear model, i.e. $f(x;\theta) = \theta^\top x$, and $f(x)$ is a quadratic, i.e. $f(x) = x^\top x$, then there is little hope that $f(x;\theta)$ can accurately predict Y given X. Complex, nonlinear relationships $f(x)$ require machine learning models $f(x;\theta)$ which are able to capture such nonlinearities.

It is clearly advantageous to choose a model $f(x;\theta)$ with a functional form close to that of $f(x)$. For example, if $f(x)$ is a quadratic, it would be helpful to choose $f(x;\theta)$ to be of the quadratic form $x^\top \theta x$. Although in real-world applications the exact functional form of $f(x)$ is unknown, we frequently do have a priori insight into certain properties of $f(x)$. Deep learning has been able to embed such properties into its model architectures, and this has been an important factor in its success. An example is the convolution network, which is used for image recognition.

Even if we choose the correct model, i.e. $f(x;\theta) = x^\top \theta x$ and $f(x) = x^\top x$, it may be challenging to accurately estimate θ if the number of data samples N is very small. For example, suppose that X is d-dimensional. Then, even in this simple setting, the model has $d \times d$ degrees of freedom (θ is a $d \times d$ matrix) and a large number of data samples N is required to accurately estimate θ. If the number of data samples $N \ll d$, the model $f(x;\theta)$ will be inaccurate. Inaccuracy due to the dataset being much smaller than the model's degrees of freedom is called "overfitting". Deep neural networks, which can have many millions of parameters, have a large number of degrees of freedom. Therefore, large datasets and careful methods for controlling overfitting are required.

1.2 Supervised Learning

Supervised learning estimates a model $f(x;\theta)$ for $f(x)$ from data $(x^i, y^i)_{i=1}^N$. At a high level, supervised learning "searches" for a θ such that, on average across the data samples, $f(x_i;\theta)$ is "close" to y_i. That is,

$$\theta = \arg\min_{\theta' \in \Theta} \sum_{i=1}^{N} \rho\big(f(x_i;\theta'), y_i\big), \tag{1.2}$$

where $\rho(z, y)$ is a measure of how close the prediction z is to y. (Recall that $\arg\min_{z \in \mathcal{Z}} g(z) = \{z : z \in \mathcal{Z},\ \ g(z) \leq g(z')\ \ \forall\ \ z' \in \mathcal{Z}\}$ and $\arg\max_{z \in \mathcal{Z}} g(z) = \{z : z \in \mathcal{Z},\ \ g(z) \geq g(z')\ \ \forall\ \ z' \in \mathcal{Z}\}$.)

The choice of ρ depends upon the application. For example, suppose that $Y \in \mathbb{R}$, $X \in \mathbb{R}^d$, and $f(x;\theta) : \mathbb{R}^d \to \mathbb{R}$. Then, a suitable choice for $\rho(z, y)$ would be the squared error $(z - y)^2$

and θ is the estimator from the familiar least-squares problem

$$\theta = \arg\min_{\theta' \in \mathbb{R}^d} \sum_{i=1}^{N} \big(f(x_i;\theta') - y_i\big)^2. \tag{1.3}$$

(1.3) is an example of a *regression* problem.

Another common class of applications are *classification* problems where $\mathcal{Y}$ is a categorical variable. For instance, suppose one is trying to classify whether an image contains a car, airplane, or neither object. Then, $Y \in \mathcal{Y}$ where the set $\mathcal{Y} = \{\text{car}, \text{airplane}, \text{neither}\}$. Y is often called the "label." For the purposes of numerical implementation, 'cars' could be encoded with the integer 0, 'airplanes' with the integer 1, and 'neither' with the integer 2. In this case, $\mathcal{Y} = \{0, 1, 2\}$.

There is also information $X \in \mathbb{R}^d$ which is available to use for predicting Y. For example, X may be the values for the image pixels, which can be used to help predict whether the image contains a car, airplane, or neither object. We would like to estimate a model $f(x;\theta)$ from data $(x^i, y^i)_{i=1}^N$ which is "close" to the exact (but unknown) conditional probability of Y given $X = x$.

More precisely, let $\mathcal{Y} = \{0, 1, 2, \ldots, M-1\}$ and set

$$f(x) = \Big(\mathbb{P}\big[Y = 0 | X = x\big], \mathbb{P}\big[Y = 1 | X = x\big], \ldots, \mathbb{P}\big[Y = M-1 | X = x\big]\Big). \tag{1.4}$$

If we knew the function $f(x)$, we could automatically predict the most likely label by simply taking the outcome y with the highest probability. Let $f_y(x)$ be the y-th element of the vector output of $f(x)$. Then, if $X = x$, our prediction y^* for Y would be:

$$y^* = \arg \max_{y \in \{0,1,\ldots,M-1\}} f_y(x). \tag{1.5}$$

Of course, we do not know $f(x)$. Instead, we seek to estimate a model $f(x;\theta)$ for $f(x)$ from data. Similar to $f(x)$, the model $f(x;\theta)$ generates a conditional probability distribution for Y given $X = x$, i.e. $f(x;\theta) : \mathbb{R}^d \to \mathcal{P}(\mathcal{Y})$ where

$$\mathcal{P}(\mathcal{Y}) = \Big\{ p \in \mathbb{R}^M : \sum_{i=0}^{M-1} p_i = 1, p \geq 0 \Big\}. \tag{1.6}$$

A natural measure for the accuracy of the model $f(x;\theta)$ is the "likelihood". Assuming the data samples are independent, the likelihood is

$$\mathcal{G}(\theta) = \prod_{i=1}^{N} f_{y^i}(x^i;\theta). \tag{1.7}$$

The likelihood is simply the probability that we would observe the data $(x^i, y^i)_{i=1}^N$ if the model $f(x;\theta)$ were the true model. The larger the probabilities $f_{y^i}(x^i;\theta)$ that the model assigns to the data points (x^i, y^i), the more accurate the model is. Therefore, the optimal choice for θ is

$$\theta = \arg\max_{\theta' \in \mathbb{R}^d} \mathcal{G}(\theta'). \tag{1.8}$$

Note that $0 < f_y(x;\theta) < 1$. As a result, $\mathcal{G}(\theta)$ is difficult to deal with in practice since it involves the multiplication of many small numbers. Numerical underflow issues can easily occur. Therefore, we consider the log-likelihood

$$\mathcal{V}(\theta) = \log\big(\mathcal{G}(\theta)\big) = \sum_{i=1}^{N} \log\big(f_{y^i}(x^i;\theta)\big). \tag{1.9}$$

The log-likelihood involves the some of the log-probabilities of the observations. Since $\log(\cdot)$ is a monotonic function,

$$\arg\max_{\theta' \in \mathbb{R}^d} \mathcal{V}(\theta') = \arg\max_{\theta' \in \mathbb{R}^d} \mathcal{G}(\theta'). \tag{1.10}$$

Since $\arg\max_x f(x) = \arg\min_x -f(x)$, it is equivalent to say that θ is the minimizer of the negative log-likelihood

$$\mathcal{L}(\theta) = -\mathcal{V}(\theta). \tag{1.11}$$

That is,

$$\theta = \arg\min_{\theta' \in \mathbb{R}^d} \mathcal{L}(\theta'). \tag{1.12}$$

The negative log-likelihood $\mathcal{L}(\theta)$ is frequently referred to as the "cross-entropy error" in machine learning. The more negative $\mathcal{L}(\theta)$ is, the more accurate the model $f(x;\theta)$ is on the data points $(x^i, y^i)_{i=1}^N$. Given a trained model $f(x;\theta)$, we can make predictions on new data points. If $X = x$, our prediction $\hat{y}$ for Y would be:

$$\hat{y} = \arg\max_{y \in \{0,1,\ldots,M-1\}} f_y(x;\theta). \tag{1.13}$$

1.3 Overfitting

Overfitting occurs when a model is trained to closely match an observed dataset, yet is inaccurate on new data points not in the training dataset. This is the result of a model with many more degrees of freedom d than the number of data samples N in the dataset.

In the regime where $d \gg N$, there are typically many different models which exactly fit the data samples. These models can vary considerably on new data points. For example, consider the model $f(x;\theta) = ax^2 + bx + c$ where $\theta = (a, b, c)$ and a dataset with a single data point (x^0, y^0). Then, there are an infinite number of models $f(x;\theta)$ which exactly fit the data point (x^0, y^0), i.e. choose $c = y^0$, $b = -ax^0$, and let a be any real number. The vast majority of these models will be inaccurate on new data points.

Complex models with large numbers of parameters, such as neural networks, are particularly susceptible to overfitting. The best way to reduce overfitting is to have a larger dataset, which will help fully resolve the degrees of freedom of the model. The deep learning field has also developed a number of techniques to control and reduce overfitting (dropout, data augmentation, transfer learning, penalties, etc.), which will be discussed in later chapters.

1.4 Notation

We briefly review some of the basic notation that will be used in the lecture notes.

We will use the standard notation for functions $f(x;\theta) : \mathcal{X} \to \mathcal{Y}$, which indicates that f's input x takes values in the space $\mathcal{X}$ and f's output takes values in the space $\mathcal{Y}$. Note that the function also depends upon the choice of parameters θ, although we will suppress this dependence in the formal definition (i.e., for notational convenience, we will not write the full mapping $f(x;\theta) : \mathcal{X} \times \Theta \to \mathcal{Y}$).

The notation $\mathbb{E}_{X,Y}\big[f(X,Y,Z)\big]$ indicates the expectation is only taken with respect to the random variables (X,Y).

The notation $x \in \mathbb{R}^d$ simply means that x is a d-dimensional vector with real values. Similarly, if $x \in \mathbb{R}^{d_1 \times d_2}$, x is a $d_1 \times d_2$ matrix. If $x \in \mathbb{R}^{d_1 \times d_2 \times d_3}$, it is a tensor (i.e., a "three-dimensional matrix") with d_1 rows in the first dimension, d_2 columns in the second dimension, and d_3 indices in the third dimension. Equivalently, x can be viewed as a collection of d_3 two-dimensional $d_1 \times d_2$ matrices.

$x \odot y$ indicates the element-wise multiplication of x and y. For example, if $x, y \in \mathbb{R}^d$ and $z = x \odot y$, then $z \in \mathbb{R}^d$ and $z_i = x_i y_i$. Similarly, if $x, y \in \mathbb{R}^{d_1 \times d_2}$, then $z \in \mathbb{R}^{d_1 \times d_2}$ and $z_{i,j} = x_{i,j} y_{i,j}$.

The indicator function $\mathbf{1}_{y=k}$ is defined as

$$\mathbf{1}_{y=k} = \begin{cases} 1 & y = k \\ 0 & y \neq k. \end{cases}$$

Vectors $x = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$ will be understood as column vectors.

2 Stochastic Gradient Descent

Stochastic gradient descent is the method of choice for training deep learning models. This section presents the "standard" stochastic gradient descent algorithm. There are many other variants (e.g., RMSprop and ADAM) which we will cover later.

2.1 Objective Function

Machine learning estimates a statistical model for the relationship between an input X and an output Y. Formally, suppose there is data $(X, Y) \in \mathbb{R}^d \times \mathcal{Y}$ and a statistical model $f(x;\theta) : \mathbb{R}^d \to \mathbb{R}^K$. $\theta \in \Theta$ are the parameters in the model and must be estimated. We wish to find a model $f(x;\theta)$ such that $f(X;\theta)$ is "an accurate prediction" for Y.

To make this statement precise, first define a function $\rho(z, y) : \mathbb{R}^K \times \mathcal{Y} \to \mathbb{R}$. The function $\rho(z, y)$ measures the distance between a prediction z and the actual observed outcome y. Then, define the objective function

$$\mathcal{L}(\theta) = \mathbb{E}_{(X,Y)}\big[\rho(f(X;\theta), Y)\big]. \tag{2.1}$$

(2.1) is a natural objective function for estimating the parameter θ. The quantity $\rho(f(X;\theta), Y)$ measures the distance between the model prediction $f(X;\theta)$ and Y for a single realization of the data (X, Y). This distance is then averaged over the distribution $\mathbb{P}_{(X,Y)}$ of the data (X, Y). The goal is of course to find a parameter θ such that, on average, the distance between the model prediction $f(X;\theta)$ and the outcome Y is small.

The best model, within the class of models $\{f(x;\theta')\}_{\theta' \in \Theta}$, is the model $f(x;\theta)$ where θ satisfies

$$\theta = \arg\min_{\theta' \in \Theta} \mathcal{L}(\theta'). \tag{2.2}$$

Typically, the distribution $\mathbb{P}_{(X,Y)}$ is unknown. Instead, i.i.d. data samples $(x^n, y^n)_{n=1}^N$ are available from the distribution $\mathbb{P}_{(X,Y)}$.[1] Then, (2.1) can be approximated as

$$\mathcal{L}^N(\theta) = \frac{1}{N} \sum_{n=1}^{N} \rho(f(x^n;\theta), y^n). \tag{2.3}$$

As the number of data samples $N \to \infty$, $\mathcal{L}^N(\theta) \to \mathcal{L}(\theta)$. In this case, we select the model corresponding to the θ which minimizes (2.3), i.e.

$$\theta = \arg\min_{\theta' \in \Theta} \mathcal{L}^N(\theta'). \tag{2.4}$$

For some simple models, (2.2) and (2.4) can be exactly calculated. However, for more complicated models such as neural networks, they cannot be exactly calculated. Instead,

[1] Independent and identically distributed (i.i.d.) samples.

numerical methods are used to minimize the objective function (2.1). When (2.1) is convex, these numerical methods may converge to the exact solution (2.2). However, when (2.1) is non-convex, the numerical methods are not guaranteed to converge to the exact solution. Neural networks are non-convex. In the non-convex case, numerical methods are only guaranteed to converge to a point which satisfies certain optimization properties. The same statement holds for the objective function (2.3) and the exact solution (2.4). We will discuss these important mathematical points later. The most widely-used numerical method for minimizing (2.1) and (2.3) is stochastic gradient descent, which is the topic of this chapter.

Example 2.1. Consider a logistic regression model for classification where $\mathcal{Y} = \{0, 1, \ldots, K-1\}$ and $\Theta = \mathbb{R}^{K \times d}$. Given an input $x \in \mathbb{R}^d$, the model $f(x;\theta)$ produces a probability of each possible outcome in $\mathcal{Y}$:

$$\begin{aligned} f(x;\theta) &= F_{\text{softmax}}(\theta x), \\ F_{\text{softmax}}(z) &= \frac{1}{\sum_{k=0}^{K-1} e^{z_k}} \left(e^{z_0}, e^{z_1}, \ldots, e^{z_{K-1}} \right), \end{aligned} \tag{2.5}$$

where z_k is the k-th element of the vector z. The set of probabilities on $\mathcal{Y}$ is $\mathcal{P}(\mathcal{Y}) \coloneqq \{p \in \mathbb{R}^K : \sum_{k=0}^{K-1} p_k = 1, p_k \geq 0 \quad \forall \quad k = 0, \ldots, K-1\}$. The function $F_{\text{softmax}}(z) : \mathbb{R}^K \to \mathcal{P}(\mathcal{Y})$ is called the "softmax function" and is frequently used in deep learning. $F_{\text{softmax}}(z)$ takes a K-dimensional input and produces a probability distribution on $\mathcal{Y}$. That is, the output of $F_{\text{softmax}}(z)$ is a vector of probabilities for the events $0, 1, \ldots, K-1$. The softmax function can be thought of as a smooth approximation to the arg max function since it pushes its smallest inputs towards 0 and its largest input towards 1.

The objective function is the negative log-likelihood (commonly referred to in machine learning as the "cross-entropy error"):

$$\begin{aligned} \mathcal{L}(\theta) &= \mathbb{E}_{(X,Y)} \big[\rho(f(X;\theta), Y) \big], \\ \rho(z, y) &= -\sum_{k=0}^{K-1} \mathbf{1}_{y=k} \log z_k, \end{aligned} \tag{2.6}$$

where z_k is the k-th element of the vector z and $\mathbf{1}_{y=k}$ is the indicator function

$$\mathbf{1}_{y=k} = \begin{cases} 1 & y = k \\ 0 & y \neq k \end{cases}$$

2.2 Stochastic gradient descent algorithm

The objective function (2.1) can be minimized via the well-known method of gradient descent:

$$\theta^{(\ell+1)} = \theta^{(\ell)} - \alpha^{(\ell)} \nabla_\theta \mathcal{L}(\theta^{(\ell)}). \tag{2.7}$$

Gradient descent repeatedly takes steps in the direction of *steepest descent*. The *negative* gradient of the objective function $\mathcal{L}(\theta)$ is the direction of *steepest descent*. The negative gradient is the direction in which $\mathcal{L}(\theta)$ is decreasing. Gradient descent repeatedly takes small

steps in the direction of the steepest descent. The magnitude of these steps is governed by the "learning rate" $\alpha^{(\ell)}$, which is a positive scalar which may depend upon the iteration number ℓ.

We can show that if the learning rate $\alpha^{(\ell)}$ is sufficiently small, the ℓ-th step of the gradient descent algorithm (2.7) is guaranteed to decrease the objective function. Assuming $\theta \in \mathbb{R}^d$ and using a Taylor expansion,

$$\begin{aligned}\mathcal{L}(\theta^{(\ell+1)}) - \mathcal{L}(\theta^{(\ell)}) &= \nabla_\theta \mathcal{L}(\theta^{(\ell)})(\theta^{(\ell+1)} - \theta^{(\ell)}) + \frac{1}{2}(\theta^{(\ell+1)} - \theta^{(\ell)})^\top \nabla_{\theta\theta}\mathcal{L}(\bar{\theta})(\theta^{(\ell+1)} - \theta^{(\ell)}) \\ &= -\alpha^{(\ell)}\Big(\nabla_\theta \mathcal{L}(\theta^{(\ell)})\Big)^\top \nabla_\theta \mathcal{L}(\theta^{(\ell)}) + \frac{1}{2}\Big(\alpha^{(\ell)}\Big)^2 \nabla_\theta \mathcal{L}(\theta^{(\ell)})^\top \nabla_{\theta\theta}\mathcal{L}(\bar{\theta}^{(\ell)}) \nabla_\theta \mathcal{L}(\theta^{(\ell)}),\end{aligned} \tag{2.8}$$

where $\bar{\theta}^{(\ell)}$ is a point on the line segment connecting $\theta^{(\ell+1)}$ and $\theta^{(\ell)}$. As long as we are not already at a stationary point $\nabla_\theta \mathcal{L}(\theta^{(\ell)}) = 0$, there is a choice of $\alpha^{(\ell)}$ such that the objective function will decrease, i.e. $\mathcal{L}(\theta^{(\ell+1)}) - \mathcal{L}(\theta^{(\ell)}) < 0$. It is also clear that if $\alpha^{(\ell)}$ is too large, the objective function may *increase* due to the second-order term. In practice, a careful choice of the learning rate is very important. The gradient descent algorithm uses only the first derivative $\nabla_\theta \mathcal{L}(\theta)$ to update the parameter θ. If it takes too large of a step, the first derivative no longer accurately describes the change in the objective function.

Gradient descent requires computing the gradient $\nabla_\theta \mathcal{L}(\theta^{(\ell)})$, which can be computationally costly since it involves an integral over (x, y):

$$\begin{aligned}\nabla_\theta \mathcal{L}(\theta^{(\ell)}) &= \nabla_\theta \mathbb{E}_{(X,Y)}\big[\rho(f(X;\theta^{(\ell)}), Y)\big] \\ &= \mathbb{E}_{(X,Y)}\big[\nabla_\theta \rho(f(X;\theta^{(\ell)}), Y)\big].\end{aligned} \tag{2.9}$$

Stochastic gradient descent is a computationally efficient scheme for minimizing (2.1). It follows a *noisy* (but unbiased) descent direction:

$$\theta^{(\ell+1)} = \theta^{(\ell)} - \alpha^{(\ell)} \nabla_\theta \rho(f(x^{(\ell)};\theta^{(\ell)}), y^{(\ell)}), \tag{2.10}$$

where $(x^{(\ell)}, y^{(\ell)})$ are i.i.d. samples from the distribution $\mathbb{P}_{(X,Y)}$. In particular, note that the *average* descent direction in (2.10) equals the descent direction in (2.7) since

$$\mathbb{E}\left[\nabla_\theta \rho(f(x^{(\ell)};\theta^{(\ell)}), y^{(\ell)})\middle|\theta^{(\ell)}\right] = \mathbb{E}\left[\nabla_\theta \rho(f(X;\theta^{(\ell)}), Y)\middle|\theta^{(\ell)}\right] = \nabla_\theta \mathcal{L}(\theta^{(\ell)}). \tag{2.11}$$

Stochastic gradient descent is computationally efficient since it only requires the gradient of the loss from a *single data sample*. It can therefore perform many more iterations than gradient descent, given the same amount of time. In practice, stochastic gradient descent (2.10) typically converges much more rapidly than gradient descent (2.7).

The distribution $\mathbb{P}_{(X,Y)}$ is usually unknown. Instead, data samples $(x_n, y_n)_{n=1}^N$ are available from the distribution $\mathbb{P}_{(X,Y)}$. Then, (2.1) can be approximated as

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{n=1}^{N} \rho(f(x^n;\theta), y^n). \tag{2.12}$$

The gradient descent algorithm for (2.12) is

$$\theta^{(\ell+1)} = \theta^{(\ell)} - \alpha^{(\ell)} \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \rho(f(x^n; \theta^{(\ell)}), y^n). \tag{2.13}$$

The stochastic gradient descent algorithm for (2.12) is:

- Randomly initialize the parameter $\theta^{(0)}$.
- For $\ell = 0, 1, \ldots, L$:
  - Select a data sample $(x^{(\ell)}, y^{(\ell)})$ at random from the dataset $(x_n, y_n)_{n=1}^N$.
  - Calculate the gradient for the loss from the data sample $(x^{(\ell)}, y^{(\ell)})$:

$$G^{(\ell)} = \nabla_\theta \rho(f(x^{(\ell)}; \theta^{(\ell)}), y^{(\ell)}) \tag{2.14}$$

  - Update the parameters:

$$\theta^{(\ell+1)} = \theta^{(\ell)} - \alpha^{(\ell)} G^{(\ell)}, \tag{2.15}$$

  where $\alpha^{(\ell)}$ is the learning rate.

The gradient descent algorithm (2.13) converges slowly since in order to take a single step, it must calculate the gradients for every data sample in the dataset. In contrast, the stochastic gradient descent algorithm (2.44) can rapidly take many steps since each step only requires calculating the gradient for a single data sample. For this reason, stochastic gradient descent is typically superior to gradient descent in practice. Stochastic gradient descent is especially advantageous when the size of the dataset N is large.

The gradient $G^{(\ell)}$ in (2.44) determines the direction of the step. The learning rate $\alpha^{(\ell)}$ determines the *size* of the step. In order for (2.10) to converge, the learning rate must decay as $\ell \to \infty$. The decaying learning rate is required to average out the noise in the stochastic gradient descent step.

In fact, the learning rate must satisfy the following conditions in order for (2.10) to converge:

$$\begin{aligned} \sum_{\ell=0}^{\infty} \alpha^{(\ell)} &= \infty, \\ \sum_{\ell=0}^{\infty} \left(\alpha^{(\ell)}\right)^2 &< \infty. \end{aligned} \tag{2.16}$$

A learning rate which satisfies these conditions is

$$\alpha^{(\ell)} = \frac{C_0}{C_1 + \ell}, \tag{2.17}$$

where C_0 and C_1 are positive constants.

Example 2.2. We will derive the stochastic gradient descent algorithm for the logistic regression model stated in Example 2.1. The logistic regression model $f(x;\theta)$ is estimated from the dataset $(x_n, y_n)_{n=1}^N$ where $(x_n, y_n) \sim \mathbb{P}_{X,Y}$.

The gradient of the loss function for a generic data sample (x, y) is

$$\nabla_\theta \rho(f(x;\theta,y) = -\nabla_\theta \log F_{\text{softmax},y}(\theta x), \tag{2.18}$$

where $F_{\text{softmax},k}(z)$ is the k-th element of the vector output of the function $F_{\text{softmax},k}(z)$. Let $\theta_{k,:}$ be the k-th row of the matrix θ. If $k \neq y$,

$$\begin{aligned}
\nabla_{\theta_{k,:}} \log F_{\text{softmax},y}(\theta x) &= -\frac{e^{\theta_{y,:}x}}{F_{\text{softmax},y}(\theta x)} \frac{e^{\theta_{k,:}x}}{\left(\sum_{m=0}^{K-1} e^{\theta_{m,:}x}\right)^2} x \\
&= -\frac{e^{\theta_{k,:}x}}{\sum_{m=0}^{K-1} e^{\theta_{m,:}x}} x \\
&= -F_{\text{softmax},k}(\theta x) x.
\end{aligned} \tag{2.19}$$

If $k = y$,

$$\begin{aligned}
\nabla_{\theta_{k,:}} \log F_{\text{softmax},y}(\theta x) &= x - \frac{e^{\theta_{y,:}x}}{F_{\text{softmax},y}(\theta x)} \frac{e^{\theta_{k,:}x}}{\left(\sum_{m=0}^{K-1} e^{\theta_{m,:}x}\right)^2} x \\
&= x - F_{\text{softmax},k}(\theta x) x.
\end{aligned} \tag{2.20}$$

Combining equations (2.19) and (2.20), we have that, for any k,

$$\nabla_{\theta_{k,:}} \log F_{\text{softmax},y}(\theta x) = \big(\mathbf{1}_{y=k} - F_{\text{softmax},k}(\theta x)\big) x. \tag{2.21}$$

Therefore,

$$\nabla_\theta \log F_{\text{softmax},y}(\theta x) = \Big(e(y) - F_{\text{softmax}}(\theta x)\Big) x^\top, \tag{2.22}$$

where

$$e(y) = \big(\mathbf{1}_{y=0}, \ldots, \mathbf{1}_{y=K-1}\big). \tag{2.23}$$

The stochastic gradient descent algorithm is:

- Select a data sample $(x^{(\ell)}, y^{(\ell)})$ at random from the dataset $(x_n, y_n)_{n=1}^N$.
- Calculate the gradient for the loss from the data sample $(x^{(\ell)}, y^{(\ell)})$:

$$G^{(\ell)} = -\left(e(y^{(\ell)}) - F_{\text{softmax}}(\theta^{(\ell)} x^{(\ell)})\right)^\top (x^{(\ell)})^\top. \tag{2.24}$$

- Update the parameters:

$$\theta^{(\ell+1)} = \theta^{(\ell)} - \alpha^{(\ell)} G^{(\ell)}, \tag{2.25}$$

2.2.1 Why must the learning rate decay? (Optional Reading)

To better understand why the learning rate (2.16) must decay, let us return to the stochastic gradient descent algorithm (2.10).

$$\theta^{(\ell+1)} = \theta^{(\ell)} - \alpha^{(\ell)} \nabla_\theta \rho(f(x^{(\ell)}; \theta^{(\ell)}), y^{(\ell)}). \tag{2.26}$$

The parameter at the $(\ell+1)$-th iteration can be re-written as:

$$\begin{aligned}
\theta^{(\ell+1)} &= \theta^{(\ell)} - \alpha^{(\ell)} \mathbb{E}_{X,Y}\big[\nabla_\theta \rho(f(X; \theta^{(\ell)}), Y)\big] \\
&+ \alpha^{(\ell)} \Big(\mathbb{E}_{X,Y}\big[\nabla_\theta \rho(f(X; \theta^{(\ell)}), Y)\big] - \nabla_\theta \rho(f(x^{(\ell)}; \theta^{(\ell)}), y^{(\ell)}) \Big) \\
&= \theta^{(\ell)} - \alpha^{(\ell)} \nabla_\theta \mathcal{L}(\theta^{(\ell)}) \\
&+ \alpha^{(\ell)} \Big(\mathbb{E}_{X,Y}\big[\nabla_\theta \rho(f(X; \theta^{(\ell)}), Y)\big] - \nabla_\theta \rho(f(x^{(\ell)}; \theta^{(\ell)}), y^{(\ell)}) \Big).
\end{aligned} \tag{2.27}$$

The first term $\nabla_\theta \mathcal{L}(\theta^{(\ell)})$ is a deterministic descent direction given the current parameter location $\theta^{(\ell)}$. This first term exactly matches the dynamics of gradient descent in (2.7). The second term is a "noise term" which randomly fluctuates around the descent direction $\nabla_\theta \mathcal{L}(\theta^{(\ell)})$. The second term represents the randomness in the stochastic gradient descent algorithm.

In order for the stochastic gradient descent algorithm to converge to a stationary point of the objective function $\mathcal{L}(\theta)$, its dynamics must asymptotically follow the descent direction $\nabla_\theta \mathcal{L}(\theta^{(\ell)})$. In order for this to occur, the "contribution" from the noise term

$$N^{(\ell)} = \alpha^{(\ell)} \Big(\mathbb{E}_{X,Y}\big[\nabla_\theta \rho(f(X; \theta^{(\ell)}), Y)\big] - \nabla_\theta \rho(f(x^{(\ell)}; \theta^{(\ell)}), y^{(\ell)}) \Big) \tag{2.28}$$

must vanish as $\ell \to \infty$. It turns out that these contributions from the noise term will disappear if $\alpha^{(\ell)}$ appropriately decays as $\ell \to \infty$. We briefly provide some insights, and refer the interested reader to [1] and [2] for a rigorous mathematical treatment.

Assume $\rho(f(x;\theta), y) \in C_b^2$.[2] For $\ell > L$, (2.27) can be re-written via a telescoping series and a Taylor expansion as

$$\begin{aligned}
\mathcal{L}(\theta^{(\ell)}) - \mathcal{L}(\theta^{(L)}) &= -\sum_{m=L}^{\ell-1} \alpha^{(m)} \Big(\nabla_\theta \mathcal{L}(\theta^{(m)}) \Big)^\top \nabla_\theta \mathcal{L}(\theta^{(m)}) \\
&+ \sum_{m=L}^{\ell-1} \Big(\nabla_\theta \mathcal{L}(\theta^{(m)}) \Big)^\top N^{(m)} \\
&+ \sum_{m=L}^{\ell-1} \big[\big(\alpha^{(m)}\big)^2 \times \text{Higher order terms} \big].
\end{aligned} \tag{2.29}$$

[2] C_b^k is the space functions with bounded and continuous derivatives up to order k.

Recall that the learning rate must satisfy

$$\sum_{m=0}^{\infty} \left(\alpha^{(m)}\right)^2 < C, \tag{2.30}$$

for some finite constant C.

Then, for any $\epsilon > 0$, there exists an L such that

$$\sum_{m=L}^{\infty} \left(\alpha^{(m)}\right)^2 < \epsilon. \tag{2.31}$$

Since $\rho(f(x;\theta), y) \in C_b^2$, the higher order terms are bounded. Then, for any any $\epsilon > 0$, there exists an L such that for all $\ell > L$

$$\left| \sum_{m=L}^{\ell-1} \left[\left(\alpha^{(m)}\right)^2 \times \text{Higher order terms} \right] \right| < K \sum_{m=L}^{\ell-1} \left(\alpha^{(m)}\right)^2 < \epsilon. \tag{2.32}$$

Next, we show that there exists an L such that the noise term is arbitrarily small for $\ell \geq L$:

$$\begin{aligned}
& \mathbb{E}\left[\left(\sum_{m=L}^{\ell-1} \left(\nabla_\theta \mathcal{L}(\theta^{(m)})\right)^\top N^{(m)} \right)^2 \right] \\
= & \sum_{m=L}^{\ell-1} \left(\alpha^{(m)}\right)^2 \mathbb{E}\left[\left(\left(\nabla_\theta \mathcal{L}(\theta^{(m)})\right)^\top \left(\mathbb{E}_{X,Y}\left[\nabla_\theta \rho(f(X;\theta^{(m)}), Y)\right] - \nabla_\theta \rho(f(x^{(m)};\theta^{(m)}), y^{(m)}) \right) \right)^2 \right] \\
+ & \sum_{m=L}^{\ell-1} \sum_{\substack{L \leq j \leq \ell-1, \\ j \neq m}}^{\ell-1} \alpha^{(m)} \alpha^{(j)} \mathbb{E}\Big[\left(\nabla_\theta \mathcal{L}(\theta^{(m)})\right)^\top \left(\mathbb{E}_{X,Y}\left[\nabla_\theta \rho(f(X;\theta^{(m)}), Y)\right] - \nabla_\theta \rho(f(x^{(m)};\theta^{(m)}), y^{(m)}) \right) \\
\times & \left(\nabla_\theta \left(\mathcal{L}(\theta^{(j)})\right)^\top \left(\mathbb{E}_{X,Y}\left[\nabla_\theta \rho(f(X;\theta^{(j)}), Y)\right] - \nabla_\theta \rho(f(x^{(j)};\theta^{(j)}), y^{(j)}) \right) \Big] \\
\leq & K \sum_{m=L}^{\ell-1} \left(\alpha^{(m)}\right)^2 \\
\leq & K \sum_{m=L}^{\infty} \left(\alpha^{(m)}\right)^2 \\
\leq & \epsilon,
\end{aligned} \tag{2.33}$$

where we have again used the fact that $\rho(f(x;\theta), y) \in C_b^2$ and the term on lines 3-4 vanishes due to iterated expectations combined with the fact that

$$\mathbb{E}\left[\mathbb{E}_{X,Y}\left[\nabla_\theta \rho(f(X;\theta^{(j)}), Y)\right] - \nabla_\theta \rho(f(x^{(j)};\theta^{(j)}), y^{(j)}) \middle| \mathcal{F}_j \right] = 0. \tag{2.34}$$

$\mathcal{F}_j$ is the filtration at step j (i.e., all of the information on the variables at steps $\tau = 0, 1, \ldots, j$).

Recall Markov's inequality, which says that we have the following inequality for any random variable $Z \in \mathbb{R}_+$ and any $\epsilon > 0$:

$$\mathbb{P}[Z \geq \epsilon] \leq \frac{\mathbb{E}[Z]}{\epsilon}. \tag{2.35}$$

Therefore, using equation (2.33) and Markov's inequality we have that, for any $\epsilon > 0$ and any $\delta > 0$, there exists an L such that for any $\ell \geq L$,

$$\mathbb{P}\left[\left(\sum_{m=L}^{\ell-1} \left(\nabla_\theta \mathcal{L}(\theta^{(m)})\right)^\top N^{(m)}\right)^2 \geq \epsilon\right] < \delta. \tag{2.36}$$

In fact, a stronger result can be proven using the Borel-Cantelli lemma, namely that, as $L \to \infty$,

$$\left|\sum_{m=L}^{\infty} \left(\nabla_\theta \mathcal{L}(\theta^{(m)})\right)^\top N^{(m)}\right| \overset{a.s.}{\to} 0. \tag{2.37}$$

Consequently, the contributions of the noise term and higher-order terms in equation (2.29) become small as L become large. The dominating term for large L is the first term in (2.29). That is, for large L, we have that

$$\mathcal{L}(\theta^{(\ell)}) - \mathcal{L}(\theta^{(L)}) \approx -\sum_{m=L}^{\ell-1} \alpha^{(m)} \left(\nabla_\theta \mathcal{L}(\theta^{(m)})\right)^\top \nabla_\theta \mathcal{L}(\theta^{(m)}). \tag{2.38}$$

Note that this is a sum of non-negative terms, which implies that $\mathcal{L}(\theta^{(\ell)}) - \mathcal{L}(\theta^{(L)}) < 0$, i.e. it decreases the objective function. Furthermore, we can easily see that the algorithm should converge to a stationary point $\nabla_\theta \mathcal{L}(\theta) = 0$. If $\left(\nabla_\theta \mathcal{L}(\theta^{(m)})\right)^\top \nabla_\theta \mathcal{L}(\theta^{(m)}) \geq \delta > 0$, then $\mathcal{L}(\theta^{(\ell)}) - \mathcal{L}(\theta^{(L)}) \leq -\delta \sum_{m=L}^{\ell-1} \alpha^{(m)}$. Since $\sum_{m=L}^{\ell-1} \alpha^{(m)} = \infty$, $\mathcal{L}(\theta^{(\ell)}) \to -\infty$. However, since $\mathcal{L}(\theta)$ is bounded, this cannot be true, and therefore there must be some $m > L$ such that $\left(\nabla_\theta \mathcal{L}(\theta^{(m)})\right)^\top \nabla_\theta \mathcal{L}(\theta^{(m)}) < \delta$.

There are two important points in the above analysis. We required that the sum of the squares of the learning rates must be finite in order to show that the noise term and higher order term in equation (2.29) vanishes. That is, it is necessary that $\sum_{m=0}^{\infty} \left(\alpha^{(m)}\right)^2 < \infty$. Secondly, we require that the sum of the learning rates diverges, i.e. $\sum_{m=0}^{\infty} \alpha^{(m)} = \infty$. This is necessary to show that the first term in (2.29) will drive the algorithm towards a stationary point $\nabla_\theta \mathcal{L}(\theta) = 0$.

We emphasize that these calculations are only meant to provide some intuition on stochastic gradient descent and its learning rate. A rigorous mathematical treatment of stochastic gradient descent can be found in [1] and [2].

2.2.2 Learning rates in practice

Although the conditions (2.16) are mathematically required for convergence, it is often sufficient in practice to simply use a piecewise learning rate schedule for $\ell = 0, 1, \ldots, K_4$ such as

$$\alpha^{(\ell)} = \begin{cases} C & \ell \leq K_1 \\ C \times 10^{-1} & K_1 < \ell \leq K_2 \\ C \times 10^{-2} & K_2 < \ell \leq K_3 \\ C \times 10^{-3} & K_3 < \ell \leq K_4 \end{cases}$$

If the learning rate is too small, convergence may be very slow. However, if the learning rate is too large, the algorithm may oscillate and make no progress. Stochastic gradient descent takes unbiased but noisy steps. Therefore, too large of a learning rate may "amplify" this noise and cause oscillations. The larger the noise, the smaller the learning rate that is required. For this reason, gradient descent can use a larger learning rate than stochastic gradient descent. We will partially address this later by developing "mini-batch" stochastic gradient descent which uses small batches of random samples to reduce the noise. In the end, the optimal learning rate heavily depends upon the specific problem and dataset.

2.3 Convergence

There is a large literature on the mathematical analysis of stochastic gradient descent. Nevertheless, this literature does not address many of the challenges of neural networks. To demonstrate, we present one of the strongest theorems which is available regarding convergence of the stochastic gradient descent algorithm (2.10):

Theorem 2.3. *Suppose that $\nabla_\theta \mathcal{L}(\theta)$ is globally Lipschitz and bounded. Furthermore, assume that the condition (2.16) holds and $\mathcal{L}(\theta)$ is bounded. Then,*

$$\mathbb{P}\left[\lim_{\ell \to \infty} \nabla_\theta \mathcal{L}(\theta^{(\ell)}) = 0\right] = 1. \tag{2.39}$$

Proof. The result is an immediate consequence of Proposition 3 in [1]. □

Theorem 2.3 states that, provided certain technical conditions, the parameter estimate $\theta^{(\ell)}$ will converge to a stationary point of the objective function $\mathcal{L}(\theta)$. Theorem 2.3 is powerful since it covers *non-convex* objective functions. The type of convergence in (2.39) is called *almost sure convergence* since with probability 1 the convergence occurs. For example, this is a stronger type of convergence than convergence in probability.

Let us now examine the conditions necessary for Theorem 2.3 to hold. Recall that the function $\nabla_\theta \mathcal{L}(\theta)$ is Globally Lipschitz if

$$|\nabla_\theta \mathcal{L}(\theta) - \nabla_\theta \mathcal{L}(\theta')| \leq K \|\theta - \theta'\|, \tag{2.40}$$

for any $\theta, \theta' \in \Theta = \mathbb{R}^d$. Although there are a wide class of models which satisfy this Global Lipschitz condition, neural networks typically do not. In fact, the gradient of a fully-connected neural network with a single hidden layer will not be Globally Lipschitz. It will also not be globally bounded. Therefore, Theorem 2.3 does not cover neural networks.

This discussion demonstrates some of the mathematical challenges of neural networks. The analysis of stochastic gradient descent algorithms for neural network models remains an interesting problem. Nevertheless, stochastic gradient descent has proven very effective in practice and is the fundamental building block of nearly all approaches for training deep learning models.

2.4 Local Minima

Stochastic gradient descent is not guaranteed to converge to the global minimum of the objective function $\mathcal{L}(\theta)$ if the model $f(x;\theta)$ is a neural network. In fact, it is very unlikely to do so. A global minimum is a parameter θ^* such that

$$\mathcal{L}(\theta^*) \leq \mathcal{L}(\theta), \tag{2.41}$$

for any $\theta \in \Theta$. The global minimum for neural networks is typically not unique (i.e., there are multiple global minima).

Neural networks typically have many local minima. The point θ is a local minimum if there exists a $\delta > 0$ such that

$$\mathcal{L}(\theta') \geq \mathcal{L}(\theta) \quad \forall \quad \|\theta' - \theta\| < \delta. \tag{2.42}$$

Stochastic gradient descent may converge to a local minimum and not a global minimum. This is one of the challenges of non-convex optimization. Neural networks are non-convex and, as a consequence, the objective function $\mathcal{L}(\theta)$ is also non-convex. Some simple examples of the non-convexity and existence of local minima for neural networks are presented in Examples 3.1 and 3.2, respectively.

2.5 Mini-batch Gradient Descent

The stochastic gradient descent algorithm we presented earlier only uses a *single* data sample for computing the update direction. Although the update is unbiased, it may be very noisy (i.e., a large variance) since there is a large amount of randomness in the single data sample that is drawn. Very noisy updates can cause oscillations and slow down convergence.

The noise in stochastic gradient descent can be easily reduced by computing the gradient on a small *mini-batch* of data samples instead of just a single data sample. More data samples reduces the variance in the update. The number of data samples M in the mini-batch is still small compared to the size of the dataset N though, and therefore mini-batch stochastic gradient descent still converges much more rapidly than gradient descent. Frequently, mini-batch stochastic gradient descent and (1 sample) stochastic gradient descent have the same computational speed since computations can be efficiently vectorized for moderate-sized M ($\sim 100 - 1,000$) .

Since the noise in the updates is reduced, mini-batch stochastic gradient descent can use a larger learning rate than stochastic gradient descent. Typically, the larger the mini-batch size M, the larger the learning rate can be.

The mini-batch stochastic gradient descent algorithm for (2.12) is:

- Randomly initialize the parameter $\theta^{(0)}$.
- For $\ell = 0, 1, \ldots, L$:
  - Select M data samples $(x^{(\ell,m)}, y^{(\ell,m)})_{m=1}^{M}$ at random from the dataset $(x_n, y_n)_{n=1}^{N}$, where $M \ll N$.
  - Calculate the gradient for the loss from the data samples:

$$G^{(\ell)} = \frac{1}{M} \sum_{m=1}^{M} \nabla_\theta \rho(f(x^{(\ell,m)}; \theta^{(\ell)}), y^{(\ell,m)}) \tag{2.43}$$

  - Update the parameters:

$$\theta^{(\ell+1)} \;=\; \theta^{(\ell)} - \alpha^{(\ell)} G^{(\ell)}, \tag{2.44}$$

  where $\alpha^{(\ell)}$ is the learning rate.

The mini-batch update $G^{(\ell)}$ is clearly still an unbiased estimate for the gradient $\nabla_\theta \mathcal{L}(\theta^{(\ell)}$. Furthermore, it is less noisy than the stochastic gradient descent update with a single sample, i.e.

$$\begin{aligned} \mathrm{Var}\left[G^{(\ell)} \middle| \theta^{(\ell)} \right] &= \mathrm{Var}\left[\frac{1}{M} \sum_{m=1}^{M} \nabla_\theta \rho(f(x^{(\ell,m)}; \theta^{(\ell)}), y^{(\ell,m)}) \middle| \theta^{(\ell)} \right] \\ &= \frac{1}{M} \mathrm{Var}\left[\nabla_\theta \rho(f(x^{(\ell)}; \theta^{(\ell)}), y^{(\ell)}) \middle| \theta^{(\ell)} \right]. \end{aligned} \tag{2.45}$$

The conditional variance of a mini-batch update is smaller by a factor of $\frac{1}{M}$ than stochastic gradient descent with a single sample, where M is the mini-batch size.

Although we have differentiated here between "mini-batch stochastic gradient descent" and "stochastic gradient descent", the former is also often referred to as "stochastic gradient descent". In practice, the term "stochastic gradient descent" is frequently used with the implicit assumption that it is in fact mini-batch stochastic gradient descent. The term "batch" is also often used interchangeably with "mini-batch".

Example 2.4. Here is an implementation in Python for training the logistic regression model from Example 2.1 using stochastic gradient descent. The model is trained to correctly label handwritten numbers in the MNIST dataset. The input is a 28×28 array of pixels from the grayscale image and the output is a conditional probability that the image is a $0, 1, 2, \ldots,$ or 9.

The dataset is divided into a "training set" and "test set". The model is estimated *only* from the training set. Once the model has been estimated, its performance is evaluated on the "test set". The logistic regression model achieves $\sim 92\%$ accuracy on the test set. We will soon train neural networks that can easily achieve $\sim 98\%$ accuracy.

```
import numpy as np

import h5py
import time
import copy
from random import randint

#load MNIST data
MNIST_data = h5py.File('MNISTdata.hdf5', 'r')
x_train = np.float32(MNIST_data['x_train'][:] )
y_train = np.int32(np.array(MNIST_data['y_train'][:,0]))
x_test = np.float32( MNIST_data['x_test'][:] )
y_test = np.int32( np.array( MNIST_data['y_test'][:,0] ) )

MNIST_data.close()



################################################################################

#Implementation of stochastic gradient descent algorithm


#number of inputs
num_inputs = 28*28

#number of outputs
num_outputs = 10

model = {}
model['W1'] = np.random.randn(num_outputs,num_inputs) / np.sqrt(num_inputs)

model_grads = copy.deepcopy(model)


def softmax_function(z):
    ZZ = np.exp(z)/np.sum(np.exp(z))
    return ZZ

def forward(x,y, model):
    Z =  np.dot(model['W1'], x)
    p = softmax_function(Z)
    return p

def backward(x,y,p, model, model_grads):
    dZ = -1.0*p
    dZ[y] = dZ[y] + 1.0
```

```
    for i in range(num_outputs):
        model_grads['W1'][i,:] = dZ[i]*x

    return model_grads


import time

time1 = time.time()
LR = .01
num_epochs = 20

for epochs in range(num_epochs):

    #Learning rate schedule
    if (epochs > 5):
        LR = 0.001
    if (epochs > 10):
        LR = 0.0001
    if (epochs > 15):
        LR = 0.00001

    total_correct = 0
    for n in range( len(x_train)):
        n_random = randint(0,len(x_train)-1 )
        y = y_train[n_random]
        x = x_train[n_random][:]
        p = forward(x, y, model)
        prediction = np.argmax(p)
        if (prediction == y):
            total_correct += 1
        model_grads = backward(x,y,p, model, model_grads)
        model['W1'] = model['W1'] + LR*model_grads['W1']
    print(total_correct/np.float(len(x_train) ) )

time2 = time.time()
print(time2-time1)


########################################################
#test data
total_correct = 0
for n in range( len(x_test)):
    y = y_test[n]
    x = x_test[n][:]
    p = forward(x, y, model)
```

```
    prediction = np.argmax(p)
    if (prediction == y):
        total_correct += 1

print(total_correct/np.float(len(x_test) ) )
```

2.6 Problems

1. Prove that the softmax function $F_{\text{softmax}} : \mathbb{R}^K \to \mathbb{R}^K$ produces a probability distribution.

2. Assume that $\mathcal{L}(\theta)$ is strongly convex. Prove that the gradient descent update always decreases the objective function. (A function $f(x) : \mathbb{R}^d \to \mathbb{R}$ is strongly convex if there exists a constant $C > 0$ such that $z^\top \nabla_{xx} f z > C z^\top z$ for any non-zero $z \in \mathbb{R}^d$.)

3. Consider a dataset $(x^i, y^i)_{i=1}^N$ where $x \in \mathbb{R}^d$ and $y \in \mathbb{R}$. Recall the least-squares objective function $\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y^i - \theta^\top x^i)^2$ for the linear model $f(x; \theta) = \theta^\top x$. Derive the stochastic gradient descent algorithm for this linear regression model.

4. Let's now consider a nonlinear model $f(x; \theta) = g(\theta^\top x)$ with an objective function $\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left| y^i - g(\theta^\top x^i) \right|$. Derive the stochastic gradient descent algorithm for this model.

5. Derive the mini-batch stochastic gradient descent algorithm for the logistic regression model.

6. Use *mini-batch* stochastic gradient descent to train a logistic regression model for classification of the MNIST dataset. Analyze the effect of different learning rates and different mini-batch sizes.

3 Neural Networks

A fully-connected network with a single hidden layer is a function

$$\begin{aligned} Z &= Wx + b^1 \\ H_i &= \sigma(Z_i), \quad i = 0, \ldots, d_H - 1, \\ f(x;\theta) &= C^\top H + b^2. \end{aligned} \tag{3.1}$$

The neural network $f(x;\theta) : \mathbb{R}^d \to \mathbb{R}^K$ takes an input x of size d and produces an output of size K. The parameters are $C \in \mathbb{R}^{K \times d_H}$, $b^1 \in \mathbb{R}^{d_H}$, $b^2 \in \mathbb{R}^K$, and $W \in \mathbb{R}^{d_H \times d}$. These parameters are collected in $\theta = \{c, b^1, b^2, W\}$.

Let us examine the architecture of the neural network (3.1). First, a linear transformation $Z = Wx + b^1$ of the input x is taken. Then, an element-wise nonlinearity $\sigma(\cdot) : \mathbb{R} \to \mathbb{R}$ is applied to each element of the vector $Z \in \mathbb{R}^{d_H}$. This element-wise transformation of Z produces the "hidden layer" $H \in \mathbb{R}^{d_H}$. The number of units in the hidden layer is d_H. The final output of the neural network is a linear transformation $C^\top H + b^2$ of the hidden layer.

Typical choices for the nonlinearities $\sigma(z)$ are:

- $\tanh(z)$
- Sigmoidal units $\frac{e^z}{1+e^z}$
- Rectified linear units (ReLU) $\max(z, 0)$

In particular, rectified linear units have proven very successful for multi-layer neural networks, and we will discuss them in more detail later.

The neural network model can be used to predict an outcome $Y \in \mathbb{R}^K$ given an input $X \in \mathbb{R}^d$. This is a "regression" problem, and the parameters θ must be chosen to minimize the "distance" between the model prediction $f(X;\theta)$ and the actual outcome Y (e.g., $\rho(z, y) = \|z - y\|^2$). Then, the goal is to select parameters θ that minimize the objective function

$$\mathcal{L}(\theta) = \mathbb{E}_{X,Y}\big[\, \|Y - f(X;\theta)\|^2 \,\big]. \tag{3.2}$$

A global minimum of the objective function (3.2) is

$$\theta^* \in \arg\min_\theta \mathcal{L}(\theta). \tag{3.3}$$

The neural network (3.1) with a single hidden layer is the simplest neural network architecture. However, even in this basic setup, the objective function (3.2) is non-convex. Therefore, it is not guaranteed that stochastic gradient descent will converge to the global minimum.

The neural network (3.1) is a nonlinear model due to the hidden layer H which involves the application of the element-wise nonlinearities $\sigma(\cdot)$. The "approximation power" of the neural network increases with the number of hidden units. First, we will make this statement mathematically precise. Then, we will discuss the practical implications.

Let $Y = f^*(X)$. That is, we would like to learn a model $f(x;\theta)$ for the relationship $y = f^*(x)$ by observing data samples (X, Y). Under mild technical conditions, for any $\epsilon > 0$, there exists a neural network with d_H hidden units such that [3]

$$\mathbb{E}_{X,Y}\left[\, \|Y - f(X;\theta^*)\|^2 \,\right] < \epsilon. \tag{3.4}$$

This result indicates that the neural network $f(x;\theta)$ can approximate a target function $f^*(x)$ arbitrarily well if it has a sufficiently large number of hidden units.

It is important to understand that the result (3.4) does not necessarily mean that a neural network trained in practice will accurately approximate a target function $f^*(x)$. (3.4) achieves the approximation error ϵ at a *global minimum*. However, numerically solving for the global minimum is intractable in practice. Instead, the objective function $\mathcal{L}(\theta)$ is minimized using stochastic gradient descent, which may converge to a local minimum. Nonetheless, (3.4) implies that greater accuracy can be achieved by increasing the number of hidden units. In practice, increasing the number of hidden units will frequently increase the accuracy (as long as the neural network does not begin to overfit).

Example 3.1. A simple example is presented to show that neural networks are non-convex. Consider a neural network with the squared loss (3.2) and a single hidden unit. The objective function is displayed below and is clearly non-convex. (Recall that a convex function $g : \mathbb{R}^d \to \mathbb{R}$ satisfies $g\big(\alpha x_1 + (1-\alpha)x_2\big) \leq \alpha g(x_1) + (1-\alpha)g(x_2)$ for any $\alpha \in [0,1]$ and all $x_1, x_2 \in \mathbb{R}^d$.)

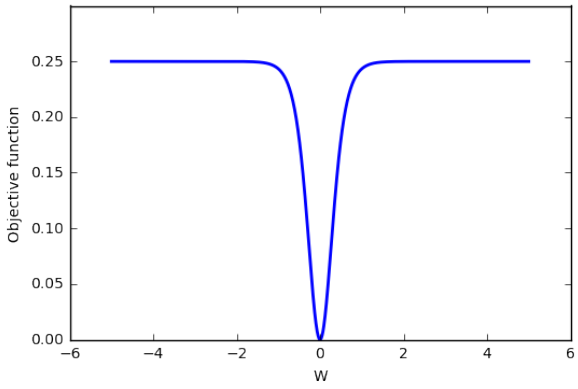

Figure 1: Example of non-convexity in neural networks.

Example 3.2. Neural networks can also have local minima, which are not global minima. As a simple example, consider a one-layer network $f : \mathbb{R} \to \mathbb{R}$ with a single ReLU unit:

$$f(x;\theta) = c\max(Wx + b^1, 0) + b^2. \tag{3.5}$$

Suppose the dataset consists of two data points $\{(x_0, y_0), (x_1, y_1)\}$ and that the loss function is $\rho(z, y) = (z - y)^2$. Any $c, W \in \mathbb{R}$, $b^1 < \min(-Wx_0, -Wx_1)$, and $b^2 = \frac{y_0+y_1}{2}$ is a local minimum.

3.1 Classification with Neural Networks

The example (3.2) considers a regression problem where a model is trained to predict a *real-valued* output given an input. Neural networks can also be used for classification problems where a model is trained to predict a *categorical* outcome given an input. In this case, the outcome is one of a set of discrete values $\mathcal{Y} = \{0, 1, \ldots, K-1\}$. The output of the model will be a vector of probabilities for these potential outcomes.

In order to perform classification, a softmax layer is added to the neural network. The neural network architecture becomes

$$\begin{aligned} Z &= Wx + b^1 \\ H_i &= \sigma(Z_i), \quad i = 0, \ldots, d_H - 1, \\ U &= CH + b^2, \\ f(x;\theta) &= F_{\text{softmax}}(U). \end{aligned} \tag{3.6}$$

The dimensions of the W, C, b^1, and b^2 remain the same as in (3.1).

With the addition of the softmax layer, the neural network now maps the input x to a probability distribution on $\mathcal{Y}$, i.e. $f(x;\theta) : \mathbb{R}^d \to \mathcal{P}(\mathcal{Y})$. The objective function is the negative log-likelihood (commonly also called the cross-entropy error):

$$\begin{aligned} \mathcal{L}(\theta) &= \mathbb{E}_{(X,Y)}\big[\rho(f(X;\theta), Y)\big], \\ \rho(v, y) &= -\sum_{k=0}^{K-1} \mathbf{1}_{y=k} \log v_k. \end{aligned} \tag{3.7}$$

The neural network produces a vector of probabilities for all potential outcomes in $\mathcal{Y}$. In many typical applications, a single prediction is required for the *most likely* outcome. The most likely outcome is

$$\underset{k=0,1,\ldots,K-1}{\arg\max} \; f_k(x;\theta), \tag{3.8}$$

where $f_k(x;\theta)$ is the k-th element of the output of the model $f(x;\theta)$.

3.2 Backpropagation Algorithm

The stochastic gradient descent algorithm requires calculating the gradient of $\rho := \rho(f(X;\theta), Y)$ with respect to the parameters $\theta = \{C, b^2, W, b^1\}$. We will calculate this gradient using the chain rule.

First, similar to our calculations for logistic regression,

$$\begin{aligned} \frac{\partial \rho}{\partial u} &= -\Big(e(Y) - f(X;\theta)\Big), \\ e(y) &= \big(\mathbf{1}_{y=0}, \ldots, \mathbf{1}_{y=K-1}\big). \end{aligned} \tag{3.9}$$

Then, we immediately have the gradient with respect to b^2:

$$\frac{\partial \rho}{\partial b^2} = \frac{\partial \rho}{\partial U} \odot \frac{\partial U}{\partial b^2} = \frac{\partial \rho}{\partial U}, \tag{3.10}$$

Recall that $x \odot y$ denotes the element-wise multiplication of x and y. For example, if $x, y \in \mathbb{R}^d$ and $z = x \odot y$, then $z \in \mathbb{R}^d$ and $z_i = x_i y_i$. Similarly, if $x, y \in \mathbb{R}^{d_1 \times d_2}$, then $z \in \mathbb{R}^{d_1 \times d_2}$ and $z_{i,j} = x_{i,j} y_{i,j}$.

Next, we consider the gradient with respect to C.

$$\frac{\partial \rho}{\partial C_{i,j}} = \sum_{k=0}^{K-1} \frac{\partial \rho}{\partial U_k} \frac{\partial U_k}{\partial C_{i,j}} = \frac{\partial \rho}{\partial U_i} H_j, \tag{3.11}$$

where $C_{i,j}$ is the element of the matrix C corresponding to the i-th row and the j-th column.

In matrix notation,

$$\frac{\partial \rho}{\partial C} = \frac{\partial \rho}{\partial U} H^\top. \tag{3.12}$$

Define

$$\delta := \frac{\partial \rho}{\partial H}. \tag{3.13}$$

We have by chain rule that

$$\begin{aligned} \delta_i &= \sum_{k=0}^{K-1} \frac{\partial \rho}{\partial U_k} \frac{\partial U_k}{\partial H_i} \\ &= \sum_{k=0}^{K-1} \frac{\partial \rho}{\partial U_k} C_{k,i} \\ &= \frac{\partial \rho}{\partial U} \cdot C_{:,i}, \end{aligned} \tag{3.14}$$

where $C:, i$ is the i-th column of the matrix C.

In matrix notation,

$$\delta = C^\top \frac{\partial \rho}{\partial U}. \tag{3.15}$$

Then, we immediately have the gradient with respect to b^1 in terms of δ:

$$\frac{\partial \rho}{\partial b^1} = \delta \odot \sigma'(Z). \tag{3.16}$$

Next, let's consider the gradient with respect to W, which can also be written in terms of δ.

$$\frac{\partial \rho}{\partial W_{i,j}} = \delta_i \sigma'(Z_i) X_j. \tag{3.17}$$

Therefore,

$$\frac{\partial \rho}{\partial W} = \Big(\delta \odot \sigma'(Z)\Big) X^\top, \tag{3.18}$$

where, with a slight abuse of notation, $\sigma'(Z)$ is understood as the element-wise application of $\sigma'(\cdot)$, i.e.

$$\sigma'(Z) = \Big(\sigma'(Z_0), \sigma'(Z_1), \ldots, \sigma'(Z_{d_H-1})\Big). \tag{3.19}$$

Collecting our results, the stochastic gradient descent algorithm for updating θ is:

- Randomly select a new data sample (X, Y).
- Compute the forward step $Z, H, U, f(X;\theta)$, and $\rho\big(f(X;\theta), Y\big)$.
- Calculate the partial derivative

$$\frac{\partial \rho}{\partial U} = -\Big(e(Y) - f(X;\theta)\Big). \tag{3.20}$$

- Calculate the partial derivatives

$$\begin{aligned} \frac{\partial \rho}{\partial b^2} &= \frac{\partial \rho}{\partial U}, \\ \frac{\partial \rho}{\partial C} &= \frac{\partial \rho}{\partial U} H^\top, \\ \delta &= C^\top \frac{\partial \rho}{\partial U}. \end{aligned} \tag{3.21}$$

- Calculate the partial derivatives

$$\begin{aligned} \frac{\partial \rho}{\partial b^1} &= \delta \odot \sigma'(Z), \\ \frac{\partial \rho}{\partial W} &= \Big(\delta \odot \sigma'(Z)\Big) X^\top. \end{aligned} \tag{3.22}$$

- Update the parameters $\theta = \{C, b^2, W, b^1\}$ with a stochastic gradient descent step:

$$\begin{aligned} C^{(\ell+1)} &= C^{(\ell)} - \alpha^\ell \frac{\partial \rho}{\partial U} H^\top, \\ b^{2,(\ell+1)} &= b^{2,(\ell)} - \alpha^\ell \frac{\partial \rho}{\partial U}, \\ b^{1,(\ell+1)} &= b^{1,(\ell)} - \alpha^\ell \delta \odot \sigma'(Z), \\ W^{(\ell+1)} &= W^{(\ell)} - \alpha^\ell \Big(\delta \odot \sigma'(Z)\Big) X^\top. \end{aligned} \tag{3.23}$$

where α^ℓ is the learning rate.

The stochastic gradient descent algorithm described above is frequently referred to as the *backpropagation algorithm*. It is composed of a forward step and a backward step. In the forward step, the output $f(X;\theta)$ and intermediary network values (Z, H, and U) are calculated. In the backward step, the gradient with respect to the parameters θ is calculated. The backward step relies upon the values calculated in the forward step.

The backward step is constructed in an efficient manner. For example, when calculating the gradient with respect to W, it re-uses some of the calculations from the gradients for C and b^1. Essentially, a large number of the steps in the chain rule are shared across the different parameters, which avoids costly re-calculations. In particular, the calculation for gradients of parameters in lower layers re-uses portions of the chain rule which have already been evaluated for parameters in higher layers. We will discuss this again in more detail when multi-layer neural networks are presented.

A numerical implementation of the backpropagation algorithm can be verified by using finite differences. That is, one can numerically estimate the gradient

$$\frac{\partial f}{\partial \theta}(x;\theta) = \frac{f(x;\theta+\Delta) - f(x;\theta-\Delta)}{2\Delta}, \tag{3.24}$$

and compare this against the result from the backpropagation algorithm.

The neural network architecture requires selecting a number of *hyperparameters* such as the number of hidden units, the type of hidden unit, and parameter initialization. Frequently, different choices of hyperparameters have to be tested in order to find the most optimal configuration (i.e., the network architecture which has the best performance).

The neural network becomes a more complex model as the number of hidden units is increased. That is, its approximation power will increase and it will be able to fit more complex relationships. However, the model will also become more likely to overfit as the number of hidden units is increased.

3.3 PyTorch and TensorFlow

PyTorch and TensorFlow are software libraries which can perform automatic differentiation of deep learning models. In summary, these libraries will automatically calculate the backpropagation algorithm, even for complex models. This can significantly accelerate the development and testing of deep learning models.

PyTorch has a define-by-run framework while TensorFlow is a define-and-run framework. The define-by-run framework in PyTorch has certain modeling advantages and, in general, PyTorch is more seamlessly integrated with Python than TensorFlow. Both frameworks are widely used. Google developed TensorFlow, while Facebook is the main developer behind PyTorch.

TensorFlow specifies the model and the computational graph before training begins. (The computational graph is the forward and backward chain of relationships in the backpropagation algorithm.) Hence, it is called a "define-and-run" framework. The model is static and cannot be easily changed during training. By using a static model, in principle, TensorFlow can achieve certain computational efficiencies by a priori optimizing some of the procedures. Since the model does not change during training, the backpropagation algorithm also remains

the same throughout training. In practice, certain "workarounds" can be used to modify the model during training; however, these are not necessarily straightforward to implement.

PyTorch allows the model and loss function to dynamically change during training and testing. PyTorch builds the computational graph on the fly. Hence, it is called a "define-by-run" framework. Thus, changes to the model which in turn cause changes to the back-propagation algorithm can be effortlessly handled, even when they occur during the training process. This produces a highly flexible computational framework for training and testing deep learning models. PyTorch's seamless integration with Python is also not to be underestimated. Typical applications will use Python for data pre-processing purposes, and better integration with the deep learning framework allows for a faster (and easier) development process.

Section 3.4 presents an example of PyTorch code for training a neural network on the MNIST dataset. Section 3.5 provides an example to demonstrate the flexibility of PyTorch's define-by-run framework.

3.4 PyTorch Implementation for a Neural Network on the MNIST Dataset

PyTorch code is provided below for training a one-layer neural network on the MNIST dataset. Note that the training is divided into a sequence of "epochs", where in each epoch the model is trained on the data from the entire training set. At the beginning of each epoch, the dataset is "randomly shuffled" so that the model is trained on a sequence of i.i.d. data samples.

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

import h5py
import time

#load MNIST data
MNIST_data = h5py.File('MNISTdata.hdf5', 'r')
x_train = np.float32(MNIST_data['x_train'][:] )
y_train = np.int32(np.array(MNIST_data['y_train'][:,0]))
x_test = np.float32( MNIST_data['x_test'][:] )
y_test = np.int32( np.array( MNIST_data['y_test'][:,0] ) )

MNIST_data.close()


#number of hidden units
H = 100
```

```
#Model architecture
class MnistModel(nn.Module):
    def __init__(self):
        super(MnistModel, self).__init__()
        # input is 28x28
        #These variables store the model parameters.

        self.fc1 = nn.Linear(28*28, H)
        self.fc2 = nn.Linear(H, 10)

    def forward(self, x):

        #Here is where the network is specified.

        x = F.tanh(self.fc1( x ))
        x = self.fc2( x )

        return F.log_softmax(x, dim=1)


model = MnistModel()

#Stochastic gradient descent optimizer
optimizer = optim.SGD(model.parameters(), lr=0.1)


batch_size = 100
num_epochs = 100
L_Y_train = len(y_train)
model.train()
train_loss = []

#Train Model
for epoch in range(num_epochs):

    #Randomly shuffle data every epoch
    I_permutation = np.random.permutation(L_Y_train)
    x_train = x_train[I_permutation,:]
    y_train = y_train[I_permutation]
    train_accu = []

    for i in range(0, L_Y_train, batch_size):
        x_train_batch = torch.FloatTensor( x_train[i:i+batch_size,:] )
        y_train_batch = torch.LongTensor( y_train[i:i+batch_size] )
        data, target = Variable(x_train_batch), Variable(y_train_batch)
```

```
        #PyTorch "accumulates gradients", so we need to set the stored gradients
            to zero when there's a new batch of data.

        optimizer.zero_grad()

        #Forward propagation of the model, i.e. calculate the hidden units and
            the output.
        output = model(data)

        #The objective function is the negative log-likelihood function.
        loss = F.nll_loss(output, target)

        #This calculates the gradients (via backpropagation)
        loss.backward()
        train_loss.append(loss.data[0])

        #The parameters for the model are updated using stochastic gradient
            descent.
        optimizer.step()

        #Calculate accuracy on the training set.
        prediction = output.data.max(1)[1] # first column has actual prob.
        accuracy = ( float( prediction.eq(target.data).sum() ) /float(batch_size)
            )*100.0
        train_accu.append(accuracy)
    accuracy_epoch = np.mean(train_accu)
    print(epoch, accuracy_epoch)

#Calculate accuracy of trained model on the Test Set
model.eval()
test_accu = []
for i in range(0, len(y_test), batch_size):
    x_test_batch = torch.FloatTensor( x_test[i:i+batch_size,:] )
    y_test_batch = torch.LongTensor( y_test[i:i+batch_size] )
    data, target = Variable(x_test_batch), Variable(y_test_batch)
    optimizer.zero_grad()
    output = model(data)
    loss = F.nll_loss(output, target)
    prediction = output.data.max(1)[1] # first column has actual prob.
    accuracy = ( float( prediction.eq(target.data).sum() ) /float(batch_size)
        )*100.0
    test_accu.append(accuracy)
accuracy_test = np.mean(test_accu)
print(accuracy_test)
```

3.5 An Example illustrating PyTorch's Define-by-Run Framework

PyTorch's define-by-run framework allows significant flexibility when training models. The model architecture and data input can be *dynamically* changed during training. A simple example is provided below to illustrate this. The number of layers in the neural network is increased during training if certain criterion, which are only known during training, are satisfied.

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
#from torchvision import datasets, transforms
from torch.autograd import Variable

import h5py
import time

#load MNIST data
MNIST_data = h5py.File('MNISTdata.hdf5', 'r')
x_train = np.float32(MNIST_data['x_train'][:] )
y_train = np.int32(np.array(MNIST_data['y_train'][:,0]))
x_test = np.float32( MNIST_data['x_test'][:] )
y_test = np.int32( np.array( MNIST_data['y_test'][:,0] ) )

MNIST_data.close()


#number of hidden units
H = 50


batch_size = 100
num_epochs = 100
L_Y_train = len(y_train)
epoch_accuracy_list = []
epoch_accuracy_list.append(0.0)

#learning rate
LR0 = 0.1

W_list = []
W0 = torch.autograd.Variable(torch.randn((H,28*28)), requires_grad=True)
W_list.append(W0)

C = torch.autograd.Variable(torch.randn((10,H)), requires_grad=True)
```

```
Number_of_layers = 1
Max_Number_of_Layers = 3


#Train Model
for epoch in range(num_epochs):

    LR = LR0/float(Number_of_layers)
    #Set gradients to zero
    #W0.grad[:] = W0.grad[:]*0.0


    #Randomly shuffle data every epoch
    I_permutation = np.random.permutation(L_Y_train)
    x_train = x_train[I_permutation,:]
    y_train = y_train[I_permutation]
    train_accu = []

    for i in range(0, L_Y_train, batch_size):
        x_train_batch = torch.FloatTensor( x_train[i:i+batch_size,:] )
        y_train_batch = torch.LongTensor( y_train[i:i+batch_size] )
        #data, target = Variable(x_train_batch).cuda(),
            Variable(y_train_batch).cuda()
        data, target = Variable(x_train_batch), Variable(y_train_batch)

        Z = torch.t( data )

        for i in range(len(W_list)):
            Z = torch.tanh( torch.mm(W_list[i], Z ) )

        V = torch.mm( C , Z)


        output = F.log_softmax( torch.t( V) , dim=1)
        loss = F.nll_loss(output, target)

        loss.backward()  # calculate gradients

        with torch.no_grad():
            for i in range(len(W_list)):
                W_list[i] -= LR * W_list[i].grad
            C -= LR * C.grad

        # Set the gradients to zero
        for i in range(len(W_list)):
            W_list[i].grad.zero_()
```

```
        C.grad.zero_()

        #calculate accuracy
        prediction = output.data.max(1)[1] # first column has actual prob.
        accuracy = ( float( prediction.eq(target.data).sum() ) /float(batch_size)
            )*100.0
        train_accu.append(accuracy)
    accuracy_epoch = np.mean(train_accu)

    epoch_accuracy_list.append(accuracy_epoch)


    #Increase the number of layers in the neural network model if certain
        criteria are satisfied.

    if ((epoch > 1) & ( epoch_accuracy_list[-1] < epoch_accuracy_list[-2] + 0.1 )
        & (Number_of_layers < Max_Number_of_Layers) & (epoch > 20) ):
        W_list.append( torch.autograd.Variable(torch.randn((H,H)),
            requires_grad=True) )

    Number_of_layers = len(W_list)

    print(epoch, accuracy_epoch, Number_of_layers)
```

3.6 Accelerating Computations on Graphics Processing Units

The backpropagation algorithm for training neural networks is composed of a series of (large) matrix multiplications that can be efficiently parallelized on graphics processing units (GPUs). GPUs have thousands of cores which allows for highly parallelized computations. A drawback is that GPUs have significantly lower memory than CPUs. They can also be slower for sequential tasks. GPUs can provide up to a 10× speedup versus CPUs for deep learning, although performance can of course vary.

It is straightforward to train models on GPUs with PyTorch. It only requires a couple of modifications of the code from Section 3.5, which are highlighted in red below.

```
model = MnistModel()
model.cuda()

optimizer = optim.SGD(model.parameters(), lr=0.1)


batch_size = 100
num_epochs = 100
L_Y_train = len(y_train)
model.train()
```

```
train_loss = []

#Train Model
for epoch in range(num_epochs):

    I_permutation = np.random.permutation(L_Y_train)
    x_train = x_train[I_permutation,:]
    y_train = y_train[I_permutation]
    train_accu = []

    for i in range(0, L_Y_train, batch_size):
        x_train_batch = torch.FloatTensor( x_train[i:i+batch_size,:] )
        y_train_batch = torch.LongTensor( y_train[i:i+batch_size] )
       data, target = Variable(x_train_batch).cuda(),
           Variable(y_train_batch).cuda()

        optimizer.zero_grad()

        output = model(data)

        loss = F.nll_loss(output, target)

        loss.backward()
        train_loss.append(loss.data[0])

        optimizer.step()

        prediction = output.data.max(1)[1]
        accuracy = ( float( prediction.eq(target.data).sum() ) /float(batch_size)
            )*100.0
        train_accu.append(accuracy)

    accuracy_epoch = np.mean(train_accu)
    print(epoch, accuracy_epoch)
```

3.7 Problems

1. Show that the global minimum of a neural network is not unique.

2. Derive the backpropagation algorithm for a neural network with an ℓ^1 loss function, i.e.

$$\rho(z, y) = \sum_{k=0}^{K-1} \big| z_k - y_k \big|. \tag{3.25}$$

3. Derive the mini-batch stochastic gradient descent algorithm for a neural network with a single hidden layer.

4. Consider clipped ReLU units

$$\sigma(z) = \min\big(\max(z, 0), t\big), \tag{3.26}$$

where t is a hyperparameter. Derive the backpropagation algorithm for a neural network with clipped ReLU units.

5. Consider a one-layer neural network $f(x; \theta) : \mathbb{R} \to \mathbb{R}$ with ReLU units. (Note that the input x is one-dimensional.) Prove that $f(x; \theta)$ can approximate any continuous function $g(x) : \mathbb{R} \to \mathbb{R}$ on an interval $[a, b]$ arbitrarily well. That is, for any $\epsilon > 0$, there exists a network with some number of hidden units d_H and some choice of parameters θ such that

$$\sup_{x \in [a,b]} \Big| f(x; \theta) - g(x) \Big| < \epsilon. \tag{3.27}$$

6. Show that the gradient (with respect to θ) of the objective function for a one-layer neural network with $\rho(z, y) = (z - y)^2$ is not globally Lipschitz.

7. Show that neural networks are non-convex.

4 Multi-layer Neural Networks

We now consider a multi-layer neural network. A fully-connected, multi-layer neural network has multiple layers, where in each layer an element-wise nonlinearity is applied to the linear combination of the output from the previous layer.

$$\begin{aligned} Z^1 &= W^1 x + b^1, \\ H^1 &= \sigma(Z^1), \\ Z^\ell &= W^\ell H^{\ell-1} + b^\ell, \quad \ell = 2, \ldots, L, \\ H^\ell &= \sigma(Z^\ell), \quad \ell = 2, \ldots, L, \\ U &= W^{L+1} H^L + b^{L+1}, \\ f(x;\theta) &= F_{\text{softmax}}(U). \end{aligned} \tag{4.1}$$

The neural network has L hidden layers followed by a softmax function. Each layer of the neural network has d_H hidden units. The ℓ-th hidden layer is $H^\ell \in \mathbb{R}^{d_H}$. H^ℓ is produced by applying an element-wise nonlinearity to the input $Z^\ell \in \mathbb{R}^{d_H}$. Using a slight abuse of notation,

$$\sigma(Z^\ell) = \Big(\sigma(Z_0^\ell), \sigma(Z_1^\ell), \ldots, \sigma(Z_{d_H-1}^\ell)\Big). \tag{4.2}$$

The parameters are $\theta = \{W^1, \ldots, W^{L+1}, b^1, \ldots, b^{L+1}\}$. The input is $x \in \mathbb{R}^d$ and the input layer has parameters $W^1 \in \mathbb{R}^{d_H \times d}$ and $b^1 \in \mathbb{R}^{d_H}$. The parameters in the layers $\ell = 2, \ldots, L$ have dimensions $W^\ell \in \mathbb{R}^{d_H \times d_H}$ and $b^\ell \in \mathbb{R}^{d_H}$. The softmax layer has parameters $W^{L+1} \in \mathbb{R}^{K \times d_H}$ and $b^{L+1} \in \mathbb{R}^K$.

Similar to before, the error (sometimes called the "loss") for a data sample (x, y) is given by

$$\rho\big(f(x;\theta), y\big) = -\sum_{k=0}^{K-1} \mathbf{1}_{y=k} \log\big(f(x;\theta)\big)_k. \tag{4.3}$$

Let $\rho := \rho\big(f(X;\theta), Y\big)$ and define

$$\delta^\ell := \frac{\partial \rho}{\partial H^\ell}. \tag{4.4}$$

By chain rule, for $\ell = 1, \ldots, L-1$,

$$\begin{aligned} \delta_i^\ell &= \sum_{j=1}^{d_H} \delta_j^{\ell+1} \frac{\partial H_j^{\ell+1}}{\partial H_i^\ell} \\ &= \sum_{j=1}^{d_H} \delta_j^{\ell+1} \sigma'(Z_j^{\ell+1}) W_{j,i}^{\ell+1} \\ &= \big(\delta^{\ell+1} \odot \sigma'(Z^{\ell+1})\big)^\top W_{:,i}^{\ell+1}. \end{aligned} \tag{4.5}$$

Therefore, for $\ell = 1, \ldots, L-1$,

$$\delta^\ell = \big(W^{\ell+1}\big)^\top \big(\delta^{\ell+1} \odot \sigma'(Z^{\ell+1})\big). \tag{4.6}$$

Consequently, for $\ell = 1, \ldots, L-1$,

$$\begin{aligned} \frac{\partial \rho}{\partial b^\ell} &= \delta^\ell \odot \sigma'(Z^\ell), \\ \frac{\partial \rho}{\partial W^\ell} &= \big(\delta^\ell \odot \sigma'(Z^\ell)\big)\big(H^{\ell-1}\big)^\top, \end{aligned} \tag{4.7}$$

where $H^0 := x$.

Finally, we have that

$$\delta^L = \big(W^{L+1}\big)^\top \frac{\partial \rho}{\partial U}, \tag{4.8}$$

and

$$\begin{aligned} \frac{\partial \rho}{\partial b^{L+1}} &= \frac{\partial \rho}{\partial U}, \\ \frac{\partial \rho}{\partial W^{L+1}} &= \frac{\partial \rho}{\partial U}\big(H^L\big)^\top. \end{aligned} \tag{4.9}$$

Collecting our results, the stochastic gradient descent algorithm for updating θ is:

- Randomly select a new data sample (X, Y).
- Compute the forward step $Z^1, H^1, \ldots, Z^L, H^L, U, f(X;\theta)$, and $\rho := \rho\big(f(X;\theta), Y\big)$.
- Calculate the partial derivative

$$\frac{\partial \rho}{\partial U} = -\bigg(e(Y) - f(X;\theta)\bigg). \tag{4.10}$$

- Calculate the partial derivatives $\frac{\partial \rho}{\partial b^{L+1}}$, $\frac{\partial \rho}{\partial W^{L+1}}$, and δ^L.
- For $\ell = L-1, \ldots, 1$:
  - Calculate δ^ℓ via the formula

$$\delta^\ell = \big(W^{\ell+1}\big)^\top \big(\delta^{\ell+1} \odot \sigma'(Z^{\ell+1})\big). \tag{4.11}$$

  - Calculate the partial derivatives with respect to W^ℓ and b^ℓ.
- Update the parameters θ with a stochastic gradient descent step.

The backpropagation algorithm is computationally efficient since it does not re-compute the chain rule for the parameters in different layers. Instead, layer ℓ re-uses the gradient computed in the previous layer $\ell + 1$ via the variable $\delta^{\ell+1}$. Furthermore, only δ^ℓ and $\delta^{\ell+1}$ need to be retained in memory in order to calculate the gradients for the parameters in layer ℓ.

4.1 Computational Cost

The computational cost of the backpropagation algorithm for multi-layer neural networks depends upon a number of factors, including: the number of layers L, the number of units in each layer d_H, the size of the input d, and the number of classes K. The number of arithmetic operations required for the forward step (i.e., make a prediction) for the multi-layer neural network (4.1) is

$$2(L-1)(d_H^2 + d_H) + 2d_H(1 + d + K) + 2K. \tag{4.12}$$

The cost increases linearly in the number of layers L and quadratically in the number of hidden units d_H. The number of arithmetic operations required for the backward step (i.e., a stochastic gradient descent step on a single data sample) is

$$(L-1)(3d_H^2 + d_H) + d_H(2d_H + d + 3K + 1) + 2K. \tag{4.13}$$

The backpropagation step is more costly than the forward step. Note that in the cost estimate for the backpropagation step, we assume that we have stored all of the relevant values from the forward step. The number of arithmetic operations includes all addition, multiplication, and algebraic operations. If one is using mini-batch stochastic gradient descent with a batch-size of N, each backpropagation step has $N\big[(L-1)(3d_H^2 + d_H) + d_H(2d_H + d + 3K + 1) + 2K\big]$ arithmetic operations.

There is also a memory cost for the parameters θ. Large neural network models can require significant amounts of memory. The memory required to store the parameters for the multi-layer network (4.1) is

$$(L-1)(d_H^2 + d_H) + d_H(d + K) + K. \tag{4.14}$$

The backpropagation algorithm also requires δ^ℓ and $\delta^{\ell+1}$, which has size $2Nd_H$ if the batch-size is N. Therefore, the total memory required for backpropagation is

$$(L-1)(d_H^2 + d_H) + d_H(d + K) + K + 2Nd_H. \tag{4.15}$$

As an example, consider a neural network with 5 layers, 500 units per layer, 100 classes, an input vector of size $1,000$, and a batch-size of $1,000$. Each parameter is stored as a 32-bit floating point number. The memory required for such a neural network is approximately 0.08 GB, which is relatively small for neural networks. More sophisticated models (such as convolution networks) can require more memory. Since GPUs have smaller memory than CPU, it can become a challenge to train large models with large batch-sizes on the GPU. The batch-size can be reduced to address this. Alternatively, there are also approaches for distributing the storage of the model across multiple GPUs or machines.

4.2 PyTorch Implementation

A multi-layer network is easily implemented in PyTorch. It only requires modifying the definition of the model in the code in Section 3.4.

```
#Multi-layer model architecture
#H is the number of units in each hidden layer
class MnistModel(nn.Module):
    def __init__(self):
        super(MnistModel, self).__init__()
        # input is 28x28
        #These variables store the model parameters.

        self.fc1 = nn.Linear(28*28, H)
        self.fc2 = nn.Linear(H, H)
        self.fc3 = nn.Linear(H, H)
        self.fc4 = nn.Linear(H, H)

        self.fc5 = nn.Linear(H, 10)

    def forward(self, x):

        #Here is where the network is specified.

        x = F.relu(self.fc1( x ))
        x = F.relu(self.fc2( x ))
        x = F.relu(self.fc3( x ))
        x = F.relu(self.fc4( x ))

        x = self.fc5( x )

        return F.log_softmax(x, dim=1)
```

The remainder of the code remains exactly the same. That is, PyTorch is set up at a high level of abstraction where the user only needs to define the (A) network architecture and (B) the objective function. Once these are defined, PyTorch will automatically calculate the backpropagation rule and train the model.

4.3 Vanishing Gradient Problem

The neural network model (4.1) becomes more complex as more layers are included. In principle, this means that the neural network can more accurately fit more complex nonlinear relationships. However, the numerical estimation of the neural network with stochastic gradient descent suffers a limitation called the *vanishing gradient problem* as the number of layers is increased.

As the number of layers L is increased, the magnitude of the gradient with respect to the parameters in the lower layers becomes small (e.g., $\frac{\partial \rho}{\partial W^\ell}$ for $\ell \ll L$). This leads to (stochastic) gradient descent converging extremely slowly. Essentially, the lower layers take an impractically long amount of time to train.

For a fixed ℓ, the magnitude of the gradient $\frac{\partial \rho}{\partial W^\ell}$ will decrease as the total number of layers L increases. Thus, although increasing L leads to a more complex model, the

numerical estimation of this model in practice becomes increasingly difficult. "Deep learning" is interested in models and methods with large numbers of layers L, i.e. very complex models, and has developed several approaches for overcoming the challenge of the vanishing gradient problem. We will discuss some of these approaches in this Chapter as well as later in the course.

Example 4.1. Let us consider a simple case where we can analytically study the vanishing gradient problem. Consider the multi-layer network

$$\begin{aligned} Z^1 &= W^1 x + b^1, \\ H^1 &= \sigma(Z^1), \\ Z^\ell &= W^\ell H^{\ell-1} + b^\ell, \quad \ell = 2, \ldots, L, \\ H^\ell &= \sigma(Z^\ell), \quad \ell = 2, \ldots, L, \\ f(x;\theta) &= W^{L+1} H^L + b^{L+1}, \end{aligned} \tag{4.16}$$

where each hidden layer has a single unit (i.e., $d_H = 1$) and $\sigma(\cdot)$ is a sigmoid function. Let's initialize $b^\ell = 0$ and $W^\ell = \frac{1}{2}$. The input dimension $d = 1$ and the output is also one-dimensional. Assume $x = 1$ and let the loss function be $\rho(z, y) = (y - z)^2$.

$H^\ell = \sigma(\frac{1}{2} H^{\ell-1})$ where we define $H^0 = x = 1$. Since $\sigma(\cdot)$ is a sigmoid function, $0 < \frac{1}{2} H^\ell \leq \frac{1}{2}$ for $\ell = 0, \ldots, L-1$. Therefore, for $\ell = 1, \ldots, L$,

$$0 < H^\ell \leq \sigma\big(\frac{1}{2}\big) < 1, \tag{4.17}$$

since $\sigma(\cdot)$ is a monotonically increasing function. Then, for $1 \leq \ell < L$,

$$\begin{aligned} \delta^\ell &= \frac{\partial \rho}{\partial H^\ell} = \delta^{\ell+1} \sigma'(Z^{\ell+1}) W^{\ell+1} \\ &= -2\big(y - f(x;\theta)\big) W^{L+1} \prod_{j=\ell+1}^{L} \sigma'(Z^j) W^j. \end{aligned} \tag{4.18}$$

The derivative of a sigmoid function is $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, which implies that $|\sigma'(z)| \leq 1$. Therefore, we have that

$$\big|\delta^\ell\big| \leq \big|\big(y - f(x;\theta)\big)\big| \times 2^{-(L-\ell)}. \tag{4.19}$$

The gradient with respect to the parameter W^ℓ is

$$\frac{\partial \rho}{\partial W^\ell} = \frac{\partial \rho}{\partial H^\ell} \frac{\partial H^\ell}{\partial W^\ell} = \delta^\ell \sigma'(Z^\ell) H^{\ell-1}. \tag{4.20}$$

Consequently, since $0 < H^\ell \leq 1$,

$$\left|\frac{\partial \rho}{\partial W^\ell}\right| \leq \big|\delta^\ell\big| \leq C 2^{-(L-\ell)}, \tag{4.21}$$

where C is a positive constant which may depend upon (x, y).

The bound (4.21) shows that the gradient with respect to the parameters in the ℓ-th layer decreases in magnitude as the total number of layers L increases. In fact, in this simple case, the magnitude decreases at an exponential rate in the total number of layers L. For large L, the gradient is so small that the lower layers in the network take an impractically long amount of time to train.

The vanishing gradient problem can also occur due to *saturation*. Saturation occurs when the inputs to the hidden units have very large magnitudes. For example, recall that if $\sigma(\cdot)$ is a sigmoidal function, then its derivative is

$$\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big). \tag{4.22}$$

Since $\lim_{\|z\| \to \infty} \sigma(z) \to 0$,

$$\lim_{\|z\| \to \infty} \sigma'(z) = 0. \tag{4.23}$$

Therefore, if the magnitudes of the inputs to the nonlinearities $\sigma(\cdot)$ are very large, the backpropagation rule will lead to very small gradients for parameters in the lower layers.

4.4 Problems

1. Prove that the gradient of a multi-layer neural network is not Globally Lipschitz and is not bounded.

2. Implement the backpropagation algorithm for a multi-layer neural network on the MNIST dataset from scratch in Python (i.e., no PyTorch).

3. Construct an example of a multi-layer neural network which has local minima which are not global minima.

5 Convolution Networks

We first consider a convolution network with a single hidden layer. Let the input image be $X \in \mathbb{R}^{d \times d}$ and a filter $K \in \mathbb{R}^{k_y \times k_x}$. Convolutions will be taken with a stride of $s = 1$ and there is only a single channel. Generalizations to the case of stride size $s > 1$, multiple channels, and multiple hidden layers will be considered later. We also do not include the Bias term, to further simplify calculations.

We define a convolution of the matrix X with the filter K as the map $X * K : \mathbb{R}^{d \times d} \times \mathbb{R}^{k_y \times k_x} \to \mathbb{R}^{(d-k_y+1) \times (d-k_x+1)}$ where

$$(X * K)_{i,j} = \sum_{m=0}^{k_y-1} \sum_{n=0}^{k_x-1} K_{m,n} X_{i+m,j+n}. \tag{5.1}$$

The hidden layer applies an element-wise nonlinearity $\sigma : \mathbb{R} \to \mathbb{R}$ to each element of the matrix $X * K$. We define the variable $Z \in \mathbb{R}^{(d-k_y+1)\times(d-k_x+1)}$ and the hidden layer $H \in \mathbb{R}^{(d-k_y+1)\times(d-k_x+1)}$ where

$$\begin{aligned} H_{i,j} &= \sigma\big((Z)_{i,j}\big), \\ Z &= X * K. \end{aligned} \tag{5.2}$$

Y is the label for the image X and takes values in the set $\mathcal{Y} = \{0, 1, \ldots, K-1\}$.[3] The output of the network is simply the softmax function applied to a linear function of the hidden layer H:

$$\begin{aligned} f(x;\theta) &= F_{\text{softmax}}(U), \\ U_k &= W_{k,:,:} \cdot H + b_k, \end{aligned} \tag{5.3}$$

where $W \in \mathbb{R}^{K\times(d-k_y+1)\times(d-k_x+1)}$, $b \in \mathbb{R}^K$, $U \in \mathbb{R}^K$, and $W_{k,:,:} \cdot H = \sum_{i,j} W_{k,i,j} H_{i,j}$.

The collection of parameters is $\theta = \{K, W, b\}$. The cross-entropy error for a single data sample (X, Y) is

$$\rho \;:= \rho(f(X;\theta), Y) = -\log\Big(f_Y(X;\theta)\Big). \tag{5.4}$$

In order to implement the stochastic gradient descent algorithm, we must calculate $\nabla_\theta \rho$. We will next derive the backpropagation rule for single-layer convolution networks.

First, define

$$\delta_{i,j} := \frac{\partial \rho}{\partial H_{i,j}} = \sum_{k=0}^{K-1} \frac{\partial \rho}{\partial U_k} W_{k,i,j} = \frac{\partial \rho}{\partial U_k} \cdot W_{:,i,j}. \tag{5.5}$$

Recall that

$$\begin{aligned} \frac{\partial \rho}{\partial U} &= -\big(e(Y) - f(X;\theta)\big), \\ e(y) &= \big(\mathbf{1}_{y=0}, \ldots, \mathbf{1}_{y=K-1}\big). \end{aligned} \tag{5.6}$$

This of course immediately yields

$$\begin{aligned} \frac{\partial \rho}{\partial W_{k,:,:}} &= \frac{\partial \rho}{\partial U_k} H, \\ \frac{\partial \rho}{\partial b} &= \frac{\partial \rho}{\partial U}. \end{aligned} \tag{5.7}$$

We must next derive the gradient with respect to the filter K. By chain rule,

[3] For example, in the MNIST dataset, $\mathcal{Y} = \{0, 1, \ldots, 9\}$.

$$\begin{aligned}\frac{\partial \rho}{\partial K_{i,j}} &= \sum_{m=0}^{d-k_y}\sum_{n=0}^{d-k_x}\delta_{m,n}\frac{\partial H_{m,n}}{\partial K_{i,j}} \\ &= \sum_{m=0}^{d-k_y}\sum_{n=0}^{d-k_x}\delta_{m,n}\sigma'(Z_{m,n})X_{i+m,j+n} \\ &= X_{i:i+d-k_y,j:j+d-k_x}\cdot\big(\sigma'(Z)\odot\delta\big), \end{aligned} \tag{5.8}$$

where, with a slight abuse of notation, $\sigma'(Z)$ is the element-wise application of the nonlinearity $\sigma(\cdot)$, i.e.

$$\sigma'(Z) = \begin{bmatrix} \sigma'(Z_{0,0}) & \sigma'(Z_{0,1}) & \dots & \sigma'(Z_{0,d-k_x}) \\ \sigma'(Z_{1,0}) & \sigma'(Z_{1,1}) & \dots & \sigma'(Z_{1,d-k_x}) \\ \vdots & \vdots & \ddots & \vdots \\ \sigma'(Z_{d-k_y,0}) & \sigma'(Z_{d-k_y,1}) & \dots & \sigma'(Z_{d-k_y,d-k_x}) \end{bmatrix}$$

In particular, we have that

$$\Big(X * \big(\sigma'(V)\odot\delta\big)\Big)_{i,j} = \sum_{m=0}^{d-k_y}\sum_{n=0}^{d-k_x}\big(\sigma'(V)\odot\delta\big)_{m,n}X_{i+m,j+n} = \frac{\partial\rho}{\partial K_{i,j}}. \tag{5.9}$$

Therefore, from the definition of a convolution,

$$\frac{\partial\rho}{\partial K} = X * \big(\sigma'(V)\odot\delta\big). \tag{5.10}$$

This is a very nice result since the gradient with respect to the parameters also involves a convolution. That is, both the backward and forward steps in the backpropagation algorithm can be written in terms of a convolution.

Collecting our results, the stochastic gradient descent algorithm for updating θ is:

- Randomly select a new data sample (X, Y).
- Compute the forward step (Z, H, U, ρ).
- Calculate the partial derivatives $(\frac{\partial\rho}{\partial U}, \delta, \frac{\partial\rho}{\partial K})$.
- Update the parameters $\theta = \{K, W, b\}$ with a stochastic gradient descent step:

$$\begin{aligned} b^{(\ell+1)} &= b^{(\ell)} - \alpha^{(\ell)}\frac{\partial\rho}{\partial U}, \\ W^{(\ell+1)}_{k,\cdot,\cdot} &= W^{(\ell)}_{k,\cdot,\cdot} - \alpha^{(\ell)}\frac{\partial\rho}{\partial U_k}H, \\ K^{(\ell+1)} &= K^{(\ell)} - \alpha^{(\ell)}\bigg(X * \big(\sigma'(V)\odot\delta\big)\bigg), \end{aligned} \tag{5.11}$$

where $\alpha^{(\ell)}$ is the learning rate.

5.1 Convolution Networks with Multiple Channels

We again study a single layer convolution network, but now with multiple channels. The hidden layer now contains C "feature maps". The number of feature maps C is often called the "number of channels". By having multiple feature maps (instead of a single feature map), the network will be able represent more complex relationships in the data.

The feature maps for the hidden layer are represented by a variable $H \in \mathbb{R}^{(d-k_y+1)\times(d-k_x+1)\times C}$. Each of the feature maps is produced by a convolution of with a filter. The convolution layer has an array (or "stack") of C filters where each filter is of size $k_y \times k_x$. The filters are given by the variable $K \in \mathbb{R}^{d_y \times d_x \times C}$.

The hidden layer H is given by:

$$H_{i,j,p} = \sigma\left(\sum_{m=0}^{k_y^\ell-1}\sum_{n=0}^{k_x^\ell-1} K^\ell_{m,n,p,p'} X_{i+m,j+n}\right). \tag{5.12}$$

Therefore,

$$\begin{aligned} H_{:,:,p} &= \sigma\Big(Z_{:,:,p}\Big), \\ Z_{:,:,p} &= X_{:,:} * K_{:,:,p}. \end{aligned} \tag{5.13}$$

The output of the network is simply the softmax function applied to a linear function of the hidden layer H:

$$\begin{aligned} f(x;\theta) &= F_{\text{softmax}}(U), \\ U_k &= W_{k,:,:,:} \cdot H + b_k, \end{aligned} \tag{5.14}$$

where $W \in \mathbb{R}^{K\times(d-k_y+1)\times(d-k_x+1)\times C}$, $b \in \mathbb{R}^K$, $U \in \mathbb{R}^K$, and $W_{k,:,:,:} \cdot H = \sum_{i,j,m} W_{k,i,j,p} H_{i,j,p}$.

The collection of parameters is $\theta = \{K, W, b\}$.

Define

$$\begin{aligned} \delta_{i,j,p} &:= \frac{\partial \rho}{\partial H_{i,j,p}} = \sum_{k=0}^{K-1} \frac{\partial \rho}{\partial U_k} W_{k,i,j,p} \\ &= \frac{\partial \rho}{\partial U} \cdot W_{:,i,j,p}. \end{aligned} \tag{5.15}$$

The backpropagation algorithm is essentially the same as before, with

$$\nabla_{K_{:,:,p}} \rho = X * \big(\sigma'(Z_{:,:,p}) \odot \delta_{:,:,p}\big), \tag{5.16}$$

and

$$\begin{aligned} \frac{\partial \rho}{\partial b} &= \frac{\partial \rho}{\partial U}, \\ \frac{\partial \rho}{\partial W_{k,:,:,:}} &= \frac{\partial \rho}{\partial U_k} H. \end{aligned} \tag{5.17}$$

References

[1] D. Bertsekas and J. Tsitsiklis. Gradient Convergence in Gradient Methods with Errors. SIAM Journal on Optimization, 10(3), 627-642, 2000.

[2] A. Benveniste, M. Metivier, and P. Priouret. Adaptive Algorithms and Stochastic Approximations. Springer, 1992.

[3] K. Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. Neural networks, 3.5, 551-560, 1990.