Feed Forward Neural Networks

Thomas Hubert

Abstract

This document is a quick summary of the mathematics behind this project's implementation of Neural Networks. This document will be updated as new features are added to the code. The reader should be familiar with the basic theory of machine learning and neural networks.

Contents

1 General Theory

1.1 Introduction

A common type of machine learning problem is to learn a function $f : X \to Y$ parametrized by θ taking input $x \in X$ and computing: $y = f(x; \theta) \in Y$. In this context, learning means modifying the parameters θ such that f computes the "right" function.

The rightness of the function is usually defined in terms of a cost function C. Given an output $y = f(x; \theta)$ and a target output $\bar{y}$, $C(y, \bar{y})$ quantifies the error between y and $\bar{y}$. Learning then means minimizing the cost function over all possible points $(y = f(x; \theta), \bar{y})$ by modifying θ.

Machine learning algorithms usually differ by how the parametric function f is defined.

1.2 Neural Network Architecture

A neural network is a functional structure composed of a series of layers which successively process the output of the previous layer. Each layer l implements a function f^l parametrized by $\theta^l = (W^l, b^l)$ of the form

$$f^l(., \theta^l) = \sigma^l \left(g^l(., \theta^l) \right)$$

The function $g^l(., \theta^l)$ is in general a linear combination of its input and its parameters θ^l and the function $\sigma^l(.)$ is a non-linear function of its input. $\sigma^l(.)$ is called the activation function of layer l.

Different g^l and $\theta^l = (W^l, b^l)$ define different types of layers.

1.3 Neural Network Learning

In the supervised learning context, a set of N data points $\{(x_0, \bar{y}_0), ..., (x_{N-1}, \bar{y}_{N-1})\}$ are given and learning generally takes the form of minimizing the average error over known points:

$$C\left((f(x_0), \bar{y}_0), ..., (f(x_{N-1}), \bar{y}_{N-1}) \right) = \frac{1}{N} \sum_{d=0}^{N-1} C(f(x_d), \bar{y}_d)$$

Neural Networks have the property that it is relatively easy and efficient to calculate the gradients of the cost function with respect to the parameters θ thanks to the back propagation algorithm. The usual steps, described below, are to forward propagate, then backward propagate and infer the gradients from these steps after processing a batch of data of size D possibly much smaller than N

$$\frac{\partial C}{\partial \theta} = \frac{1}{D} \sum_{d=0}^{D-1} \frac{\partial C}{\partial \theta}(f(x_d), \bar{y_d}) \approx \frac{1}{N} \sum_{d=0}^{N-1} \frac{\partial C}{\partial \theta}(f(x_d), \bar{y_d})$$

Neural Networks then learn using gradient information. Many optimization methods have been adapted for Neural Networks but in the simplest case, a gradient descent algorithm can be used to minimize the cost function and learn the parameters θ

$$\theta \leftarrow \theta - \alpha \frac{\partial C}{\partial \theta}$$

We describe below the steps which allow to efficiently compute the gradients of the cost function with respect to the parameters θ.

1.3.1 Forward Propagation

Forward propagation is the procedure which computes $y = f(x; \theta)$ successively applying the layers' computation one after the other.

More formally, given a neural network composed of L layers indexed by $0 \leq l < L$, $y = f(x; \theta)$ is computed by:

$$
\begin{aligned}
&a^0 = x \\
&\text{For } l = 1 \text{ to } L-1 \\
&\quad z^l = g^l(a^{l-1}; \theta^l) \\
&\quad a^l = \sigma^l(z^l) \\
&y = a^{L-1}
\end{aligned}
$$

a^l is called the activation of layer l and is stored at this stage.

We observe that we only need the specification of g^l and σ^l to forward propagate.

1.3.2 Backward Propagation

Backward propagation is the procedure which computes $\delta^l = \frac{\partial C}{\partial z^l}$ for every l, starting from $l = L-1$ down to $l = 1$. δ^l is the key component to derive the gradients of the cost function with respect to the parameters θ^l.

More formally, δ^l is computed in the following way:

$$
\begin{aligned}
&\delta^{L-1} = \frac{\partial C}{\partial z^{L-1}} = \frac{\partial C}{\partial a^{L-1}} \frac{\partial a^{L-1}}{\partial z^{L-1}} = \frac{\partial C}{\partial y} da^{L-1} \\
&\text{For } l = L-1 \text{ to } 2 \\
&\quad \delta^{l-1} = \frac{\partial C}{\partial z^{l-1}} = \frac{\partial C}{\partial z^l} \frac{\partial z^l}{\partial a^{l-1}} \frac{\partial a^{l-1}}{\partial z^{l-1}} = \delta^l (g^l)'(a^{l-1}; \theta^l) da^{l-1}
\end{aligned}
$$

where we have used the short hand notation $da^l = \frac{\partial a^l}{\partial z^l} = (\sigma^l)'(z^l)$.

We observe that we need to be able to calculate the derivatives of g^l and σ^l (and therefore implicitly a^l) to backward propagate.

1.3.3 Gradient Calculation

Gradient calculation is the procedure which computes $\frac{\partial C}{\partial \theta^l}$ for every $l > 0$.

$$
\frac{\partial C}{\partial \theta^l} = \frac{\partial C}{\partial z^l} \frac{\partial z^l}{\partial \theta^l} = \delta^l \partial_{\theta^l} g^l(a^{l-1}; \theta^l)
$$

We observe that once we have a hold of δ^l, we need to be able to calculate the derivative of g^l with respect to the parameters θ to calculate the gradients.

2 Layers

In this section, we go over the different types of layers that are implemented in this Neural Network project and describe for each the forward propagation, backward propagation and gradient calculation procedures in detail.

We will use the index i to go over the input layer and the index o to go over the output layer.

2.1 Fully Connected Layer

A fully connected layer input and output are basically a vector. As its name suggests, it links every output neuron to every input neuron. There is no restriction on the type of layer preceding a fully connected layer.

2.1.1 Forward Propagation

A fully connected layer implements the following g^l function:

$$g^l : \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l}$$
$$\theta^l = (W^l, b^l) \in (\mathbb{R}^{N_l \times N_{l-1}}, \mathbb{R}^{N_l})$$

The output neuron at position o, is computed by:

$$z_o^l = g^l(a^{l-1}; \theta)_o = \sum_{i=0}^{N^{l-1}-1} W_{oi}^l a_i^{l-1} + b_o^l$$
$$a_o^l = \sigma^l(z_o^l)$$

We can rewrite this in vector form:

$$z^l = g^l(a^{l-1}; \theta) = a^{l-1}(W^l)^T + b^l$$
$$a^l = \sigma^l(z^l)$$

This can be further generalized to process a batch of D input vectors at the same time by having:

$$g^l : \mathbb{R}^{D \times N_{l-1}} \to \mathbb{R}^{D \times N_l}$$
$$\theta^l = (W^l, B^l) \in (\mathbb{R}^{N_l \times N_{l-1}}, \mathbb{R}^{D \times N_l})$$
$$Z^l = g^l(A^{l-1}; \theta) = A^{l-1}(W^l)^T + B^l$$
$$A^l = \sigma^l(Z^l)$$

where $A^l = (a_0^l, ..., a_{D-1}^l)^T$ and $B^l = (b^l, ..., b^l)^T$.

2.1.2 Backward Propagation

We start by calculating

$$\frac{\partial z_o^l}{\partial z_i^{l-1}} = \frac{\partial}{\partial z_i^{l-1}} \left(\sum_{i=0}^{N^{l-1}-1} W_{oi}^l a_i^{l-1} + b_o^l \right) = W_{oi}^l da_i^{l-1}$$

We deduce that

$$\delta_i^{l-1} = \frac{\partial C}{\partial z_i^{l-1}} = \sum_{o=0}^{N^l-1} \frac{\partial C}{\partial z_o^l} \frac{\partial z_o^l}{\partial z_i^{l-1}} = da_i^{l-1} \sum_{o=0}^{N^l-1} \delta_o^l W_{oi}^l$$

Using the symbol $\odot$ for element-wise multiplication, the backward propagation can therefore be written in vector form:

$$\delta^{l-1} = da^{l-1} \odot \left(\delta^l W^l \right)$$

This can be further generalized to process a batch of D input vectors at the same time by having:

$$\Delta^{l-1} = dA^{l-1} \odot \left(\Delta^l W^l \right)$$

where $dA^l = (da_0^l, ..., da_{D-1}^l)^T$ and $\Delta^l = (\delta_0^l, ..., \delta_{D-1}^l)^T$.

2.1.3 Gradients Calculation

Gradients calculation give:

$$\frac{\partial C}{\partial b_o^l} = \frac{\partial C}{\partial z_o^l}\frac{\partial z_o^l}{\partial b_o^l} = \delta_o^l$$
$$\frac{\partial C}{\partial W_{oi}^l} = \frac{\partial C}{\partial z_o^l}\frac{\partial z_o^l}{\partial W_{oi}^l} = \delta_o^l a_i^{l-1}$$

Therefore in vector form, we get:

$$\frac{\partial C}{\partial b^l} = \delta^l$$
$$\frac{\partial C}{\partial W^l} = (\delta^l)^T a^{l-1}$$

When a batch of D input vectors are processed at the same time, the gradients are averaged over each input data. This gives

$$\frac{\partial C}{\partial b^l} = \frac{1}{D}\sum_{d=0}^{D-1}\delta_d^l$$
$$\frac{\partial C}{\partial W^l} = \frac{1}{D}\sum_{d=0}^{D-1}\left((\delta^l)^T a^{l-1}\right)_d$$

2.2 Convolutional Layer

A convolutional layer's input and output are 3 dimensional volumes. Thought for visual recognition, convolutional layer takes advantage of the 3 dimensionality of space. In particular, the activation of a neuron only depends on the neurons that are spatially close to it in the previous layer. The type of layer preceding a convolutional layer must be a 3 dimensional volume.

2.2.1 Convolution and Correlation

We start with a reminder on convolution and correlation operations. In one dimension, for vectors $x, y \in \mathbb{R}^n$, the convolution and correlation of x, y denoted respectively as $x * y$ and $x \bar{*} y$ are another vector $z \in \mathbb{R}^n$ such that

$$z_i = (x * y)_i = \sum_{k=0}^{n-1} x_k y_{i-k}$$
$$z_i = (x \bar{*} y)_i = \sum_{k=0}^{n-1} x_k y_{i+k}$$

where the indices are taken modulo n.

Similarly, we can define the same operations in two dimensions:

$$z_{i,j} = (x * y)_{i,j} = \sum_{k_1=0}^{n_1-1}\sum_{k_2=0}^{n_2-1} x_{k_1,k_2} y_{i-k_1,j-k_2}$$
$$z_{i,j} = (x \bar{*} y)_{i,j} = \sum_{k_1=0}^{n_1-1}\sum_{k_2=0}^{n_2-1} x_{k_1,k_2} y_{i+k_1,j+k_2}$$

Now, suppose we have a vector $x \in \mathbb{R}^{n_1} \times \mathbb{R}^{n_2}$ and a smaller vector $w \in \mathbb{R}^{m_1} \times \mathbb{R}^{m_2}$ where $m_i < n_i$. We define the vector $z \in \mathbb{R}^{n_1-m_1+1} \times \mathbb{R}^{n_2-m_2+1}$ as a weighted average of neighbouring points of x such that

$$z_{i,j} = \sum_{k_1=0}^{m_1-1}\sum_{k_2=0}^{m_2-1} w_{k_1,k_2} x_{i+k_1,j+k_2}$$

for $0 \leq i \leq n_1 - m_1$ and $0 \leq j \leq n_2 - m_2$

By abuse of notation, we will write $z_{i,j} = (w \bar{*} x)_{i,j}$ even though the operation we are doing is not exactly a correlation.

Similarly, if we define z such that

$$z_{i,j} = \sum_{k_1=0}^{m_1-1} \sum_{k_2=0}^{m_2-1} w_{k_1,k_2} x_{i+m_1-1-k_1,j+m_2-1-k_2}$$

for $0 \leq i \leq n_1 - m_1$ and $0 \leq j \leq n_2 - m_2$, we will use the same abuse of notation and write $z_{i,j} = (w * x)_{i,j}$ even though the operation we are doing is not exactly a convolution.

2.2.2 Forward Propagation

A convolutional layer implements the following g^l function:

$$\begin{aligned} g^l &: \mathbb{R}^{D^{l-1} \times H^{l-1} \times W^{l-1}} \to \mathbb{R}^{D^l \times H^l \times W^l} \\ \theta^l &= (W^l, b^l) \in (\mathbb{R}^{D^l \times D^{l-1} \times wH^l \times wW^l}, \mathbb{R}^{D^l}) \end{aligned}$$

with $H^l = H^{l-1} - wH^l + 1$ and $W^l = W^{l-1} - wW^l + 1$.

The letters D, H, W are shorts for depth, height, width, wH, wW are shorts for weight's height and weight's width, with $wH^l < H^{l-1}$ and $wW^l < W^{l-1}$.

In this representation:

- a neuron in layer l, at depth d is connected to only its neighbouring neurons in layer $l-1$ which are in a volume $D^{l-1} \times wH^l \times wW^l$.
- all neurons in layer l at depth d share the same weights and the same bias

More precisely, a neuron in layer l at position od, oh, ow is connected to neurons in layer $l-1$ at position $id \in [0, D_{l-1}[, ih \in \left[oh, oh + wH^l\right[, iw \in \left[ow, ow + wW^l\right[$. Mathematically, this is written:

$$\begin{aligned} z^l_{od,oh,ow} &= \sum_{id=0}^{D^{l-1}-1} \sum_{wh=0}^{wH^l-1} \sum_{ww=0}^{wW^l-1} W^l_{od,id,wh,ww} a^{l-1}_{id,oh+wh,ow+ww} + b^l_{od} \\ &= \sum_{id=0}^{D_{l-1}-1} (W^l_{od,id} \bar{*} a^{l-1}_{id})_{oh,ow} + b^l_{od} \end{aligned}$$

for $0 \leq od < D^l$, $0 \leq oh \leq H^{l-1} - wH^l$ and $0 \leq ow \leq W^{l-1} - wW^l$.

An important note for computational purpose is that the above correlation operation is done for every od for each id. It is more efficient to do all of them at once before switching to the next id.

2.2.3 Backward Propagation

We start by calculating

$$\begin{aligned} \frac{\partial z^l_{od,oh,ow}}{\partial z^{l-1}_{id,ih,iw}} &= \sum_{wh=0}^{wH^l-1} \sum_{ww=0}^{wW^l-1} W^l_{od,id,wh,ww} \frac{\partial a^{l-1}_{id,oh+wh,ow+ww}}{\partial z^{l-1}_{id,ih,iw}} \\ &= \sum_{wh=0}^{wH^l-1} \sum_{ww=0}^{wW^l-1} W^l_{od,id,wh,ww} da^{l-1}_{id,oh+wh,ow+ww} \mathbb{1}_{ih=oh+wh} \mathbb{1}_{iw=ow+ww} \\ &= da^{l-1}_{id,ih,iw} W^l_{od,id,ih-oh,iw-ow} \mathbb{1}_{0 \leq ih-oh < wH^l} \mathbb{1}_{0 \leq iw-ow < wW^l} \end{aligned}$$

We deduce that

$$\begin{aligned}
\delta^{l-1}_{id,ih,iw} &= \sum_{od=0}^{D^l-1}\sum_{oh=0}^{H^l-1}\sum_{ow=0}^{W^l-1}\delta^l_{od,oh,ow}\frac{\partial z^l_{od,oh,ow}}{\partial z^{l-1}_{id,ih,iw}} \\
&= da^{l-1}_{id,ih,iw}\sum_{od=0}^{D^l-1}\sum_{oh=0}^{H^l-1}\sum_{ow=0}^{W^l-1} W^l_{od,id,ih-oh,iw-ow}\delta^l_{od,oh,ow}\mathbb{1}_{0\leq ih-oh<wH^l}\mathbb{1}_{0\leq iw-ow<wW^l} \\
&= da^{l-1}_{id,ih,iw}\sum_{od=0}^{D^l-1}\sum_{wh=0}^{wH^l-1}\sum_{ww=0}^{wW^l-1} W^l_{od,id,wh,ww}\delta^l_{od,ih-wh,iw-ww}\mathbb{1}_{0\leq ih-wh<H^l}\mathbb{1}_{0\leq iw-ww<W^l}
\end{aligned}$$

We almost recover a convolution operation but the $\mathbb{1}_{0\leq ih-ww<H^l}\mathbb{1}_{0\leq iw-ww<W^l}$ terms are cumbersome.

To simplify the task, we pad every matrix $\delta^l_{od,:,:}$ of size $W^l \times H^l$ with zeros to a size of $W^l + 2(wW^l-1) \times H^l + 2(wH^l-1) = W^{l-1} + wW^l - 1 \times H^{l-1} + wH^l - 1$ such that

$$\delta^{p,l}_{od,oh+wH^l-1,ow+wW^l-1} = \delta^l_{od,oh,ow}\mathbb{1}_{0\leq oh<H^l}\mathbb{1}_{0\leq ow<W^l}$$

for $oh \in \left[-wH^l+1, H^l-1+wH^l-1\right]$ and $ow \in \left[-wW^l+1, W^l-1+wW^l-1\right]$

We therefore get:

$$\begin{aligned}
\delta^{l-1}_{id,ih,iw} &= da^{l-1}_{id,ih,iw}\sum_{od=0}^{D^l-1}\sum_{wh=0}^{wH^l-1}\sum_{ww=0}^{wW^l-1} W^l_{od,id,wh,ww}\delta^{p,l}_{od,ih-wh+wH^l-1,iw-ww+wW^l-1} \\
&= da^{l-1}_{id,ih,iw}\sum_{od=0}^{D^l-1}(W^l_{od,id} * \delta^{p,l}_{od})_{ih,iw}
\end{aligned}$$

Similarly to the forward propagation it is more efficient to do the convolution operations for all id's before switching to the next od.

2.2.4 Gradients Calculation

Gradients calculation give:

$$\begin{aligned}
\frac{\partial C}{\partial b^l_{od}} &= \sum_{od=0}^{D^l-1}\sum_{oh=0}^{H^l-1}\sum_{ow=0}^{W^l-1}\frac{\partial C}{\partial z^l_{od,oh,ow}}\frac{\partial z^l_{od,oh,ow}}{\partial b^l_{od}} = \sum_{oh=0}^{H^l-1}\sum_{ow=0}^{W^l-1}\delta^l_{od,oh,ow} \\
\frac{\partial C}{\partial W^l_{od,id,wh,ww}} &= \sum_{od=0}^{D^l-1}\sum_{oh=0}^{H^l-1}\sum_{ow=0}^{W^l-1}\frac{\partial C}{\partial z^l_{od,oh,ow}}\frac{\partial z^l_{od,oh,ow}}{\partial W^l_{od,id,wh,ww}} \\
&= \sum_{oh=0}^{H^l-1}\sum_{ow=0}^{W^l-1}\delta^l_{od,oh,ow}a^{l-1}_{id,oh+wh,ow+ww} \\
&= (\delta^l_{od} \bar{*} a^{l-1}_{id})_{wh,ww}
\end{aligned}$$

Similarly to the forward propagation it is more efficient to do the correlation operations for all od's before switching to the next id.

2.3 Pool Layer

A pool layer's input and output are also 3 dimensional volumes. One of the goal of a pool layer is to summarize the information of the input layer in a more concise way without loosing too much information. One way of doing so would be to output the average or the maximum of a small part of the input. In the following, we will only describe the max pool layer.

2.3.1 Forward Propagation

A pool layer implements the following g^l function:

$$g^l : \mathbb{R}^{D^{l-1} \times H^{l-1} \times W^{l-1}} \to \mathbb{R}^{D^l \times H^l \times W^l}$$

with $H^l = \frac{H^{l-1} - mH^l}{s} + 1$, $W^l = \frac{W^{l-1} - mW^l}{s} + 1$ and $D^l = D^{l-1}$.

The letters D, H, W are shorts for depth, height and width similarly to the convolutional layers. mH^l, mW^l are shorts for map's height and width and finally s is short for stride. Note that a pool layer does not have any parameters θ.

A neuron in layer l, at depth d is connected to only its neighbouring neurons in layer $l-1$ at the same depth d in an area of size $mH^l \times mW^l$ defined by $ih \in \big[oh \times s, oh \times s + mH^l\big[$, $iw \in \big[ow \times s, ow \times s + mW^l\big[$. A pool operation $P : \mathbb{R}^{mH^l \times mW^l} \to \mathbb{R}$ is applied on the input neurons to produce the output neuron.

Mathematically, this is written:

$$z^l_{od,oh,ow} = P_{mh,mw}(a^{l-1}_{id,oh \times s + mh, ow \times s + mw})$$

with $mh \in [0, mH[$, $mw \in [0, mW]$ and for $0 \leq od = id < D^l$, $0 \leq oh \leq \frac{H^{l-1} - mH^l}{s}$ and $0 \leq ow \leq \frac{W^{l-1} - mW^l}{s}$.

We concentrate on the max pool layer for which $P = \max$. We therefore get in this special case:

$$z^l_{od,oh,ow} = \max_{mh,mw}(a^{l-1}_{id,oh \times s + mh, ow \times s + mw}) = a^{l-1}_{id,oh \times s + \bar{mh}, ow \times s + \bar{mw}}$$

where $\bar{mh}$ and $\bar{mw}$ are the indexes where the max realizes.

2.3.2 Backward Propagation

We start by calculating

$$\begin{aligned}
\frac{\partial z^l_{od,oh,ow}}{\partial z^{l-1}_{id,ih,iw}} &= \frac{\partial}{\partial z^{l-1}_{id,ih,iw}} P_{mh,mw}(a^{l-1}_{id,oh \times s + mh, ow \times s + mw}) \\
&= da^{l-1}_{id,ih,iw} \frac{\partial}{\partial a^{l-1}_{id,ih,iw}} P_{mh,mw}(a^{l-1}_{id,oh \times s + mh, ow \times s + mw})
\end{aligned}$$

In the particular case of the max pool layer for which $P = \max$, we get:

$$\begin{aligned}
\frac{\partial z^l_{od,oh,ow}}{\partial z^{l-1}_{id,ih,iw}} &= da^{l-1}_{id,ih,iw} \frac{\partial}{\partial a^{l-1}_{id,ih,iw}} a^{l-1}_{id,oh \times s + \bar{mh}, ow \times s + \bar{mw}} \\
&= da^{l-1}_{id,ih,iw} \mathbb{1}_{ih = oh \times s + \bar{mh}} \mathbb{1}_{iw = ow \times s + \bar{mw}}
\end{aligned}$$

We deduce that

$$\begin{aligned}
\delta^{l-1}_{id,ih,iw} &= \sum_{oh=0}^{H^l - 1} \sum_{ow=0}^{W^l - 1} \delta^l_{id,oh,ow} \frac{\partial z^l_{od,oh,ow}}{\partial z^{l-1}_{id,ih,iw}} \\
&= da^{l-1}_{id,ih,iw} \sum_{oh=0}^{H^l - 1} \sum_{ow=0}^{W^l - 1} \delta^l_{id,oh,ow} \mathbb{1}_{ih = oh \times s + \bar{mh}} \mathbb{1}_{iw = ow \times s + \bar{mw}}
\end{aligned}$$

This last equation can be interpreted as, every output neuron at od, oh, ow, contributes to the input neuron at $id, ih, iw = od, oh \times s + \bar{mh}, ow \times s + \bar{mw}$ by the amount $da^{l-1}_{id,ih,iw} \delta^l_{od,oh,ow}$.

2.3.3 Gradients Calculation

As a pool layer does not have any parameters, there are no gradients to calculate.

3 Activation Functions

The activation functions are the non-linear functions that link z to a by $a = \sigma(z)$. Without activation functions, a neural network would just be a linear function of its input, irrespective of how many layers it is composed of. The non-linearity of the activation functions give the representational power of the neural networks.

We describe some possible activation functions by detailing σ and its derivative σ'.

3.1 Identity

We have for the Identity activation function:

$$\sigma(z) = z$$
$$\sigma'(z) = 1$$

3.2 Sigmoid

We have for the Sigmoid activation function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$
$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

3.3 TanH

We have for the TanH activation function:

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$
$$\sigma'(z) = 1 - \sigma(z)^2$$

3.4 Relu

We have for the Relu activation function:

$$\sigma(z) = max(0, z)$$
$$\sigma'(z) = \mathbb{1}_{z \geq 0} = \mathbb{1}_{\sigma(z) \geq 0}$$

4 Cost Functions

Cost functions express the error between the output of the neural network and the true answer. More formally, given an input x and a target output $\bar{y}$, cost functions C quantify the error between $y = f(x; \theta)$ and $\bar{y}$.

In the supervised classification problem, when the data belongs to class k, $\bar{y}$ is a vector full of 0 except one entry with value 1 at position k. We describe some possible cost functions having the supervised classification problem in mind.

4.1 Mean Squared Error

We have for the mean squared error cost function:

$$C(y, \bar{y}) = \frac{1}{2} \sum_i (y_i - \bar{y}_i)^2$$

$$\frac{\partial}{\partial y_j} C(y, \bar{y}) = (y_j - \bar{y}_j)$$

4.2 Cross Entropy

We have for the cross entropy error cost function:

$$C(y, \bar{y}) = -\sum_i \bar{y}_i \log(y_i) + (1 - \bar{y}_i) \log(1 - y_i)$$

$$\frac{\partial}{\partial y_j} C(y, \bar{y}) = \frac{y_j - \bar{y}_j}{y_j(1 - y_j)}$$

4.3 Softmax

We have for the softmax cost function:

$$C(y, \bar{y}) = -\log(\frac{e^{y_k}}{\sum_i e^{y_i}}) = -y_k - \log(\sum_i e^{y_i})$$

$$\frac{\partial}{\partial y_j} C(y, \bar{y}) = -\mathbb{1}_{k=j} - \frac{e^{y_j}}{\sum_i e^{y_i}}$$

4.4 Support Vector Machine

We have for the support vector machine cost function:

$$C(y, \bar{y}) = \sum_i \max(y_i - y_k + 1, 0) - 1$$

$$\frac{\partial}{\partial y_j} C(y, \bar{y}) = \mathbb{1}_{y_j \geq y_k - 1} - \mathbb{1}_{j=k} \sum_i \mathbb{1}_{y_i \geq y_k - 1}$$

5 Optimizers

Thanks to the back propagation procedure, it is easy and efficient to calculate the gradients $\nabla_\theta C(\theta)$ of the cost function with respect to the parameters θ. Neural Networks learn using gradient information and we describe here several optimizers that make good use of the gradient information in order to find good parameters θ.

5.1 Gradient Descent

Gradient descent is the simplest optimizer. It has one parameter α the learning rate and its update rule is:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta C(\theta_t)$$

It is often a good idea to decrease the learning rate α with time.

5.2 Nesterov Momentum

Momentum optimizers introduce the notion of speed to make gradient descent more robust. They have two parameters: α the learning rate and γ the friction. The friction parameter γ is usually set close to 1 around 0.9.

Simple momentum update rule is :

$$\begin{aligned} v_t &= \gamma v_{t-1} - \alpha \nabla_\theta C(\theta_t) \\ \theta_{t+1} &= \theta_t + v_t \end{aligned}$$

Nesterov momentum is a refined version of the momentum taking into account the fact that θ is going to change by γv and it is therefore better to compute the gradient at $\theta_t + \gamma v_t$ instead of at θ_t. After a change of variable, Nesterov momentum update rule can be written:

$$\begin{aligned} v_t &= \gamma v_{t-1} - \alpha \nabla_\theta C(\theta_t) \\ \theta_{t+1} &= \theta_t + v_t + \gamma(v_t - v_{t-1}) \end{aligned}$$

5.3 RMSProp

RMSProp has also two main parameters: α the learning rate and γ the friction. It also has a less important parameter ϵ set close to 0 to avoid division by 0. The friction parameter γ is usually set close to 1 around 0.9.

RMSProp keeps track of v^2 the moving average of the squared gradients and use this to scale the learning rate α. RMSProp update is written:

$$\begin{aligned} v_t^2 &= \gamma v_{t-1}^2 + (1-\gamma)(\nabla_\theta C(\theta_t))^2 \\ \theta_{t+1} &= \theta_t - \frac{\alpha}{\sqrt{v_t^2 + \epsilon}} \nabla_\theta C(\theta_t) \end{aligned}$$

5.4 AdaDelta

AdaDelta is similar to RMSProp but tries to do without the learning rate parameter α. It has a main parameter γ the friction set close to 1 and a less important parameter ϵ set close to 0.

In addition to keeping track of the moving average of the squared gradients, it also keeps track of x^2 the moving average of the parameter updates. AdaDelta update rule is written:

$$\begin{aligned} v_t^2 &= \gamma v_{t-1}^2 + (1-\gamma)(\nabla_\theta C(\theta_t))^2 \\ \theta_{t+1} &= \theta_t - \sqrt{\frac{x_{t-1}^2 + \epsilon}{v_t^2 + \epsilon}} \nabla_\theta C(\theta_t) \\ x_t^2 &= \gamma x_{t-1}^2 + (1-\gamma)(\theta_{t+1} - \theta_t)^2 \end{aligned}$$

References

[1] Michael Nielsen, http://neuralnetworksanddeeplearning.com

[2] Stanford University, http://vision.stanford.edu/teaching/cs231n/