Imperial College
London

NOTES

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Machine Learning: Deep Learning I

Author:
W (CID: your college-id number)

Date: January 6, 2020

1 Introduction

Machine Learning can be categorized into two main categories:

1. Supervised Learning: Given input x and output y, the algorithm learns the mapping function $f(x) = y$.

2. Unsupervised Learning: Given only input x, the algorithm learns the underlying structure.

Artificial Neural Networks (ANN) is an information processing paradigm use to solve a specific problem imitating the human brain.

Deep Learning is buzzword to describe a subset of neural networks that are used to make predictions. In deep learning, logistic regression is seen as one layer of **neurons** and we build **neural networks** using a network of neurons to solve more complex problems.

2 Neural Networks

Neural network is one of the many algorithms for supervised learning.

Neural networks is used for multi-class classification. Unlike logistic regression, neural networks can classify non-linearly separable data and overcome the **Donut Problem** and **XOR Problem**.

2.1 Architecture

A neural network consists of **input layer**, **hidden layer** and **output layer** where each layer is made of **neurons**.

All neurons in one layer is connected to every neurons in the next layer. The input signals are propagated through the hidden layers until it reaches the output layer, which makes the prediction.

To train a neural network is to find the **weights** for each connection such that the output layer makes the correct prediction.

2.2 One Hot Encoding

One hot encoding is a process of converting categorical data to a form that can be used in machine learning algorithms.

Given a total of K classes. The class k is expressed as a $K \times 1$ column vector with the k^{th} row set to 1 and other rows set to 0.

2.3 Softmax Function

In binary classification, we only need one output. This is because we only need one number to represent $P(Y = 1 \mid x)$, and find the probability of the other class by

$$P(Y = 0 \mid x) = 1 - P(Y = 1 \mid x)$$

We can actually use two outputs nodes and normalize them so they sum to 1. Notice we use exponents to make sure the output is positive, also w is no longer a vector but a $D \times 2$ matrix.

$$P(Y = 0 \mid x) = \frac{e^{w_0^T x}}{e^{w_0^T x} + e^{w_1^T x}} \qquad P(Y = 1 \mid x) = \frac{e^{w_1^T x}}{e^{w_0^T x} + e^{w_1^T x}}$$

We can extend the idea to K classes to get the softmax function. Notice that w is now a $D \times K$ matrix. (w_i is the i^{th} column)

$$P(Y = k \mid x) = \frac{e^{w_k^T x}}{\sum_{i=1}^{K} e^{w_i^T x}}$$

2.3.1 Softmax vs Sigmoid

Sigmoid function is a Softmax function with two classes.

$$\begin{aligned} P(Y = 1 \mid x) &= \frac{e^{w_1^T x}}{e^{w_0^T x} + e^{w_1^T x}} = \frac{1}{\frac{e^{w_0^T x}}{e^{w_1^T x}} + 1} \\ &= \frac{1}{1 + e^{-(w_1 - w_0)^T x}} \end{aligned}$$

This means having two weights is redundant since we can use $w = w_1 - w_0$.

From software design perspective it is always safer two use softmax as it can handle any value of $K \geq 2$

2.4 Forward Propagation

To understand forward propagation, we start with one sample x that is a column vector with D features.

We use the notations:

- $W^{(l)}$ as the weights connecting layer $l - 1$ and layer l.
- $W_{ij}^{(l)}$ as the weight between neuron i in layer $l - 1$ and neuron j in layer l.
- $b^{(l)}$ as the bias connecting layer $l - 1$ and l.
- $b_i^{(l)}$ as the bias for neuron i in layer l.

- $A^{(l)}$ as the outputs at layer l
- $A_i^{(l)}$ as the outputs for neuron i at layer l
- $f^{(l)}$ as the activation function at layer l

For the input layer, $A^{(0)} = x$.

Assume there is only one hidden layer with M neurons. Hence $W^{(1)}$ is a $D \times M$ matrix. $b^{(1)}$ is a column vector of size $M \times 1$.

The output layer has K neurons. Hence $W^{(2)}$ is a $M \times K$ matrix. $b^{(2)}$ is a column vector of size $K \times 1$.

The first step of forward propagation is to use the input layer to calculate the output for each neuron in the hidden layer

$$A_j^{(1)} = f^{(1)}\left(\sum_{i=1}^{D} W_{ij}^{(1)} x_i + b_j^{(1)}\right)$$
$$A^{(1)} = f^{(1)}\left((W^{(1)})^T x + b^{(1)}\right)$$

The next step is to use the output from hidden layer as the inputs of the output layer. And apply the softmax function.

$$A_j^{(2)} = softmax\left(\sum_{i=1}^{M} W_{ij}^{(2)} A_i^{(1)} + b_j^{(2)}\right)$$
$$A^{(2)} = softmax\left((W^{(2)})^T A^{(1)} + b^{(2)}\right)$$

The output is a $K \times 1$ vector, where the k^{th} row is $P(y = k \mid x)$. We then predict x to the class with the highest probability.

2.4.1 Multiple Samples

When performing forward propagation for N samples, we use $X_{N \times D}$ to represent all samples. Instead of performing forward propagation for each sample, we can combine the operations. The output of the hidden layer is

$$f^{(1)}\left(XW^{(1)} + b^{(1)}\right)$$

Similarly the output of the output layer is

$$softmax\left(A^{(1)}W^{(2)} + b^{(2)}\right) = softmax\left(f^{(1)}\left(XW^{(1)} + b^{(1)}\right)W^{(2)} + b^{(2)}\right)$$

Notice the output is now a $N \times K$ matrix where each row is the predictions of one sample. Also b is no longer a column vector but is duplicate N times to get a matrix of N rows.

3 Training a Neural Network

Training means finding the right weight for the neural network such that it makes predictions with high accuracy.

In multi-class classification, given a set of input X belonging to class $y \in \{1...K\}$. The cost function we want to minimize is **Categorical Cross Entropy**:

$$J = -\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk}\log(\hat{y}_{nk})$$

y_{nk} is 1 if the sample n belongs to class k and 0 otherwise. Notice that in $\sum^K$, only one term is positive while all other terms are 0.

$\hat{y}_{nk}$ is the output of the neural network, indicating the $P(y = k \mid x)$ for sample n. (A value between 0 and 1)

3.1 Multi-class Logistic Regression

To minimize the loss function we want to find the derivative of J with respect to each of the weights, then update the weights using gradient descent.

The method used to find the derivatives is **back propagation**. To understand how it works, we start with the simplest case using one output layer of K neurons with identity activation function. Given a set of inputs X that belongs to class $y \in \{1...K\}$, we can calculate $P(y = k \mid x)$ for each class by:

$$Z = W^T x + b$$
$$\hat{Y} = softmax(Z)$$

We can apply chain rule to find the derivative of J with respect to each weight

$$\frac{\partial J}{\partial W_{dk}} = \sum_{n=1}^{N}\sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial \hat{Y}_{nk'}}\frac{\partial \hat{Y}_{nk'}}{\partial Z_{nk}}\frac{\partial Z_{nk}}{W_{dk}}$$
$$\frac{\partial J}{\partial b_{k}} = \sum_{n=1}^{N}\sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial \hat{Y}_{nk'}}\frac{\partial \hat{Y}_{nk'}}{\partial Z_{nk}}\frac{\partial Z_{nk}}{b_{k}}$$

To make this easier to visualize:

$$X = \begin{bmatrix} x_{11} & x_{12} & ... & x_{1D} \\ ... & ... & ... & ... \\ x_{N1} & x_{N2} & ... & x_{ND} \end{bmatrix} \qquad W = \begin{bmatrix} w_{11} & w_{12} & ... & w_{1K} \\ ... & ... & ... & ... \\ w_{D1} & w_{D2} & ... & w_{DK} \end{bmatrix} \qquad b = \begin{bmatrix} b_1 & b_2 & ... & b_K \end{bmatrix}$$

$$Z = \begin{bmatrix} z_{11} & z_{12} & ... & z_{1K} \\ ... & ... & ... & ... \\ z_{N1} & z_{N1} & ... & z_{NK} \end{bmatrix} \qquad \hat{Y} = \begin{bmatrix} \hat{y}_{11} & \hat{y}_{12} & ... & \hat{y}_{1K} \\ ... & ... & ... & ... \\ \hat{y}_{N1} & \hat{y}_{N1} & ... & \hat{y}_{NK} \end{bmatrix} \qquad J = \begin{bmatrix} j_{11} & j_{12} & ... & j_{1K} \\ ... & ... & ... & ... \\ j_{N1} & j_{N1} & ... & j_{NK} \end{bmatrix}$$

The way to understand the chain rule is to assume there is only one sample. To find the derivative of J w.r.t w_{11} for example, is to find out how w_{11} affects the outcomes j_{11}, j_{12}, ..., j_{1K}.

Each component of J depends on their corresponding $\hat{Y}$, meaning j_{11} depends on $\hat{y}_{11}$, j_{12} depends on $\hat{y}_{12}$... Hence the first derivative.

Each component of $\hat{Y}$ depends on all of the Z, meaning $\hat{y}_{11}$ depends on z_{11}, z_{12}, ..., z_{1K}, $\hat{y}_{12}$ depends on z_{11}, z_{12}, ..., z_{1K}... Hence the second derivative uses k' to iterate through all the $\hat{y}$.

Each component of Z depends on a set of weights and bias. The last derivative is used to find how a particular weight and bias affects a component of Z.

3.1.1 Solving the Derivatives

The next step is to solve the derivatives: ($\delta_{kk'}$ is the Kronecker Delta Function)

$$
\begin{aligned}
J_{nk'} &= -Y_{nk'} \log(\hat{Y}_{nk'}) \\
\frac{\partial J_{nk'}}{\partial \hat{Y}_{nk'}} &= -\frac{Y_{nk'}}{\hat{Y}_{nk'}}
\end{aligned}
$$

$$
\hat{Y}_{nk'} = e^{Z_{nk'}} \left(\sum_{i=0}^{K} e^{Z_{ni}} \right)^{-1}
$$

$$
\begin{aligned}
\frac{\partial \hat{Y}_{nk'}}{\partial Z_{nk}} &= \begin{cases} e^{Z_{nk}}(-1)\left(\sum_{i=0}^{K} e^{Z_{ni}}\right)^{-2} e^{Z_{nk}} + \left(\sum_{i=0}^{K} e^{Z_{ni}}\right)^{-1} e^{Z_{nk}}, & k' = k \\ e^{Z_{nk'}}(-1)\left(\sum_{i=0}^{K} e^{Z_{ni}}\right)^{-2} e^{Z_{nk}}, & k' \neq k \end{cases} \\
&= \begin{cases} e^{Z_{nk}}\left(\sum_{i=0}^{K} e^{Z_{ni}}\right)^{-1} - e^{2Z_{nk}}\left(\sum_{i=0}^{K} e^{Z_{ni}}\right)^{-2}, & k' = k \\ -e^{Z_{nk'}}\left(\sum_{i=0}^{K} e^{Z_{ni}}\right)^{-1} e^{Z_{nk}}\left(\sum_{i=0}^{K} e^{Z_{ni}}\right)^{-1}, & k' \neq k \end{cases} \\
&= \begin{cases} \hat{Y}_{nk}(1 - \hat{Y}_{nk}), & k' = k \\ -\hat{Y}_{nk'}\hat{Y}_{nk}, & k' \neq k \end{cases} \\
\because \delta_{kk'} &= \begin{cases} 1, & k' = k \\ 0, & k' \neq k \end{cases} \\
\therefore \frac{\partial \hat{Y}_{nk'}}{\partial Z_{nk}} &= \begin{cases} \hat{Y}_{nk'}(1 - \hat{Y}_{nk}), & k' = k \\ \hat{Y}_{nk'}(0 - \hat{Y}_{nk}), & k' \neq k \end{cases} \\
&= \hat{Y}_{nk'}(\delta_{kk'} - \hat{Y}_{nk})
\end{aligned}
$$

$$
Z_{nk} = X_{n1} W_{1k} + X_{n2} W_{2k} + ... + X_{nD} W_{Dk} + b_k
$$

$$\frac{\partial Z_{nk}}{W_{dk}} = X_{nd}$$
$$\frac{\partial Z_{nk}}{b_k} = 1$$

3.1.2 Combining the Derivatives

Combing the derivatives we get

$$\begin{aligned}
\frac{\partial J}{\partial W_{dk}} &= \sum_{n=1}^{N}\sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial \hat{Y}_{nk'}} \frac{\partial \hat{Y}_{nk'}}{\partial Z_{nk}} \frac{\partial Z_{nk}}{W_{dk}} \\
&= \sum_{n=1}^{N}\sum_{k'=1}^{K} -\frac{Y_{nk'}}{\hat{Y}_{nk'}} \hat{Y}_{nk'}(\delta_{kk'} - \hat{Y}_{nk}) X_{nd} \\
&= \sum_{n=1}^{N}\sum_{k'=1}^{K} -Y_{nk'}(\delta_{kk'} - \hat{Y}_{nk}) X_{nd} \\
&= \sum_{n=1}^{N} X_{nd} \left(\hat{Y}_{nk} \sum_{k'=1}^{K} Y_{nk'} - \sum_{k'=1}^{K} Y_{nk'}\delta_{kk'} \right) \\
&= \sum_{n=1}^{N} X_{nd} \left(\hat{Y}_{nk} \sum_{k'=1}^{K} Y_{nk'} - Y_{nk} \right) && \delta_{kk'} = 0 \text{ when } k' \neq k \\
&= \sum_{n=1}^{N} X_{nd} \left(\hat{Y}_{nk} - Y_{nk} \right) && \sum Y_{nk'} = 1 \text{ since only one target is 1} \\
\frac{\partial J}{\partial W} &= X^T(\hat{Y} - Y)
\end{aligned}$$

$$\begin{aligned}
\frac{\partial J}{\partial b_k} &= \sum_{n=1}^{N}\sum_{k'=1}^{K} \frac{\partial J_{nk'}}{\partial \hat{Y}_{nk'}} \frac{\partial \hat{Y}_{nk'}}{\partial Z_{nk}} \frac{\partial Z_{nk}}{b_k} \\
&= \sum_{n=1}^{N} \left(\hat{Y}_{nk} - Y_{nk} \right) \\
\frac{\partial J}{\partial b} &= \hat{Y} - Y
\end{aligned}$$

3.2 Back Propagation

Given a neural network of input X, belonging to class $y \in \{1...K\}$, with D features; one hidden layer with M neurons; and an output layer with K neurons.

$$A^{(0)} = X \qquad Z^{(1)} = A^{(0)}W^{(1)} + b^{(1)}$$
$$A^{(1)} = f^{(1)}\left(Z^{(1)}\right) \qquad Z^{(2)} = A^{(1)}W^{(2)} + b^{(2)}$$

$$\hat{Y} = softmax(Z^{(2)}) \qquad J = -\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk}\log(\hat{y}_{nk})$$

From *Multi-class Logistic Regression*, we saw how to find the derivative of J w.r.t $W^{(2)}$ and $b^{(2)}$ in the **output layer**. The only difference is instead of X, we treat $A^{(1)}$ as the input.

$$\frac{\partial J}{\partial W^{(2)}} = A^{(1)T}(\hat{Y} - Y) \qquad \frac{\partial J}{\partial b^{(2)}} = \hat{Y} - Y$$

Finding the derivative of J for the **hidden layers** requires more steps. Notice that to find the derivative of J w.r.t. $W^{(l)}$ and $b^{(l)}$ only requires the derivative of J w.r.t $Z^{(l)}$.

$$\frac{\partial J}{\partial W^{(l)}} = \frac{\partial J}{\partial Z^{(l)}}\frac{\partial Z^{(l)}}{\partial W^{(l)}} = A^{(l-1)T}\frac{\partial J}{\partial Z^{(l)}}$$
$$\frac{\partial J}{\partial b^{(l)}} = \frac{\partial J}{\partial Z^{(l)}}\frac{\partial Z^{(l)}}{\partial b^{(l)}} = \frac{\partial J}{\partial Z^{(l)}}$$

From *Multi-class Logistic Regression*, saw that the derivative of J w.r.t. the output layer $Z^{(2)}$ is

$$\frac{\partial J}{\partial Z^{(2)}} = \sum_{n=1}^{N}\sum_{k'=1}^{K}\frac{\partial J_{nk'}}{\partial \hat{Y}_{nk'}}\frac{\partial \hat{Y}_{nk'}}{\partial Z_{nk}} = \hat{Y} - Y$$

The derivative J w.r.t. $Z^{(l)}$ can be calculated from the derivative J w.r.t $Z^{(l+1)}$. Therefore this derivative is back propagated through the layers to update the weights.

$$\frac{\partial J}{\partial Z^{(l)}} = \frac{\partial J}{\partial Z^{(l+1)}}\frac{\partial Z^{(l+1)}}{\partial A^{(l)}}\frac{\partial A^{(l)}}{\partial Z^{(l)}}$$

In our example,

$$\begin{aligned}
\frac{\partial J}{\partial Z^{(1)}} &= \frac{\partial J}{\partial Z^{(2)}}\frac{\partial Z^{(2)}}{\partial A^{(1)}}\frac{\partial A^{(1)}}{\partial Z^{(1)}} \\
&= \left(\hat{Y} - Y\right) W^{(2)T}\left(f^{(1)'}(Z^{(1)})\right) \\
\frac{\partial J}{\partial W^{(1)}} &= A^{(0)T}\frac{\partial J}{\partial Z^{(1)}} \\
&= X^T\left(\left(\hat{Y} - Y\right) W^{(2)T}\left(f^{(1)'}(Z^{(1)})\right)\right) \\
\frac{\partial J}{\partial b^{(1)}} &= \frac{\partial J}{\partial Z^{(1)}} \\
&= \left(\hat{Y} - Y\right) W^{(2)T}\left(f^{(1)'}(Z^{(1)})\right)
\end{aligned}$$

4 Neural Network for Linear Regression

Neural Networks can be used in both classification and regression problems. For different problems, we keep the hidden layers the same and only apply different functions to the output layer.

- Binary classification: Apply sigmoid to output layer, $\hat{Y} = \sigma(Z)$ (One neuron)
- Multi-class classification: Apply softmax to output layer, $\hat{Y} = softmax(Z)$
- Regression: Apply identity to output layer, $\hat{Y} = Z$ (One neuron)

5 Hyperparameters

Hyperparameters are settings that can be tuned to control the behavior of a machine learning algorithm.

In deep learning, some possible hyperparameters are:

- Learning rate
- Regularization term
- Number of hidden layers
- Number of neurons per hidden layer
- Activation function on hidden layer

There is no precise way to choose the best hyperparameters. In practice we can use **K Fold Cross Validation** to find the best configuration.

5.1 Activation Functions

We can apply different activation functions to the neurons in the hidden layers. The activation function for the output layer is dependent on the type of problem we are trying to solve.

5.1.1 Sigmoid

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

Sigmoid is a theoretical interest because it has range $(0, 1)$ so it act just like a neuron.

One problem with sigmoid is the **vanishing gradient problem**, most part of the sigmoid is flat and has a gradient of zero making it difficult to update weights in gradient descent.

Another problem with sigmoid is that it **offsets the data**. Even if the data is normally standardized with 0 mean and variance 1, the output of the sigmoid $\sigma(0) = 0.5$

5.1.2 Hyperbolic Tangent

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

$tanh$ is typically better than sigmoid because it does not offset the data, $tanh(0) = 0$. However it still has the **vanishing gradient problem**.

5.1.3 Rectifier Linear Unit (ReLU)

$$relu(z) = max(0, z)$$
$$relu'(z) = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}$$

ReLU is the most commonly used activation function because it trains more easily compare to sigmoid and hyperbolic tangent.

One problem with ReLU is **dead neurons**. Since the gradient is zero for inputs less than equal to zero, the weights won't be update again once z reaches zero. As a result this neuron will always output 0. The solution is to add more hidden units and hope not many of them become dead neurons.

5.2 Learning Rate

Learning rate is usually a small number around less than 0.1.

A learning rate that is too high will make the lost goes to infinity, a learning rate that is too low if the error cost is converging too slowly.

Increase and decrease learning rates by factor of 10.

6 A Deeper Look into Cross Entropy

Minimizing the multi-class cross entropy function is the same as maximizing the likelihood function of a Categorical distribution.

The likelihood function is the joint probability distribution of the random samples. Since all samples are independent, the likelihood is the product of individual probabilities.

Given a set of inputs X that belongs to class $y \in \{1...K\}$. We can express $P(Y = k \mid x)$ as $\hat{y}$

$$\text{argmax}\, L = \text{argmax} \left(\prod_{n=1}^{N} \prod_{k=1}^{K} \hat{y}_{nk}^{y_{nk}} \right)$$

$\hat{y}_{nk}$ is the probability of X_n belonging to class k. This is a value between 0 to 1.

y_{nk} is 1 if X_n belongs to class k and 0 otherwise. Therefore in $\prod^K$ only one term is a valid and the other terms are 1.

$$\operatorname{argmax} L = \operatorname{argmax}\left(\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk}\log(\hat{y}_{nk})\right)$$

Maximizing an error function $-E$ is equivalent of minimizing E.

$$\operatorname{argmax}\left(\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk}\log(\hat{y}_{nk})\right) = \operatorname{argmin}\left(-\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk}\log(\hat{y}_{nk})\right)$$

Which is minimizing the cross entropy function.