CIS 522 - University Of Pennsylvania, Spring 2020

Lecture 3: Introduction to Neural Network

23 January 2020

Lecturer: Konrad Kording *Scribe: Chetan, Rohan*

1 History of Neural Networks

Artificial neural networks were first proposed in 1943 by Warren McCulloch and Walter Pitts. Since then, they have undergone much development and transformation to become the powerful tools that they are today. Some of the important developments necessary to make the neural network what it is today include the development of the backpropagation algorithm in the 1970s (which allowed for the practical training of neural networks) and the advent of GPUs and distributed computing in the 2010s (to allow for the training and development of networks with many layers − deep learning!).

Today, neural networks can perform many complex tasks, such as recognizing objects in images by name, interpreting and generating human readable text, and much more.

1.1 Motivation behind the use and growth of the neural networks

Neural networks are designed to model the inner workings of a human brain. A biological neuron receives many input signals from other nodes at its dendrite and then, based on the combination of inputs, decides to “fire" (send an electric signal down its axon) or not. In a similar way, all of the individual nodes that make up an artificial neural network receive many different inputs and outputs a low or high number based on the particular combination of inputs provided.

Neural networks have become extremely popular today due to their ability to approximate extremely complex functions. Since layers can be stacked further and further on top of each other, neural network architects can add more and more layers of analysis to discover and analyze complex patterns within the data, such as the presence of a face within an image or a long range dependency within a sentence. The introduction of GPUs and distributed computing has allowed computer scientists to add more and more layers to create more and more powerful neural networks.

2 Perceptron Model

The simplest type of a neural network is a single layer perceptron, which is used for binary classification. The perceptron takes as input a vector x containing n real numbers $x_1, x_2, ..., x_n$. Each of these real numbers x_i is multiplied by a corresponding weight w_i, which is also a real number. The resulting products are then all added together and added to a real valued bias term b. If this

final sum is greater than or equal to 0, the perceptron outputs 1; otherwise, it outputs -1. Thus, the single layer perceptron can be summed up with the equation

$$y(x) = sign(w \cdot x + b) = sign(\sum_{i=1}^{n}(w_i * x_i) + b)$$

where w is a vector of the weights $w_1, w_2, ..., w_n$ and x is a vector of the real numbers $x_1, x_2, ..., x_n$.

The weights of the perceptron are trained using the following algorithm.

1. Start with all $w_i = 0$
2. For each training example
   (a) Classify the training example x as $+1$ or -1 via the algorithm discussed above
   (b) If the predicted class is correct, do nothing
   (c) If the predicted class is -1 but the correct class is $+1$, let $w = w + x$
   (d) If the predicted class is $+1$ but the correct class is -1, let $w = w - x$

The neat thing about the single layer perceptron is that if the data is linearly separable, the perceptron training algorithm is guaranteed to converge to a solution that classifies all data points correctly!

For example, the perceptron can correctly and completely learn several logical functions, such as AND and OR.

2.1 Multi Layer Perceptron

This generated a lot of excitement about the single layer perceptron and its ability to solve complex problems automatically for humanity.

However, data is very rarely linearly separable in real life. For example, although it only contains 4 data points in two dimensions, the perceptron is unable to learn the logical function XOR since its points are not linearly separable. The perceptron's inability to learn such a simple function as this really dampened enthusiasm around the technology. As can be seen from this example, although the single layer perceptron works surprisingly well in some cases, it doesn't really work for most data sets.

Thus, researchers came up with the idea of the multilayer perceptron to remedy this idea. This perceptron involved multiple iterations of multiplying weights with the "current" input vector. Essentially, the original input vector x would be multiplied by a weight matrix W_1 and added to a bias vector b_1 to produce a new vector x_1. Then, x_1 would be multiplied by another weight matrix W_2 and added to another bias vector b_2 to produce x_2. This process would continue iteratively until a vector x_{i-1} would finally be multiplied with a matrix W_i and added to a bias vector b_i to produce the output. Thus, the multi layer perceptron can be summed up with the equation

$$y(x) = W_i(W_{i-1}(...(W_2(W_1 * x + b_1) + b_2)...) + b_{i-1}) + b_i$$

However, researchers discovered that this process of iterating linear function just produces a more complex linear function, meaning that it was still only able to learn the separate linearly separable data. This necessitated the introduction of an activation function to enable to learning of more complex functions.

3 Activation Function

Activation functions are non-linear functions that are applied after each successive layer of the perceptron. Thus, instead of an iterative process of multiplying by weights and adding a bias, the perceptron is an iterative process of multiplying by weights, adding a bias, and then applying a non-linear function.

As noted above, the purpose of the activation function is to enable the network to learn more complex functions. With this nonlinearity present after each layer, the network is no longer a linear function and thus, can learn higher order or non-polynomial functions.

Assuming that σ denotes a non-linear function, we can now rewrite our multilayer perceptron as

$$y(x) = \sigma(W_i * \sigma(W_{i-1} * \sigma(...\sigma(W_2 * \sigma(W_1 * x + b_1) + b_2)...) + b_{i-1}) + b_i)$$

3.1 Different Activation functions

There are several different types of non-linear functions that one can choose as the activation function in their neural network.

1. **ReLU**

   $$\sigma(z) = max(0, z)$$

   ReLU is a piecewise linear function that is very fast to compute and thus, the most commonly used activation function. Although it is not always differentiable, the main issue with ReLU is actually when the derivative is 0 (whenever $z = 0$). In this case, the ReLU is considered "dead" since we cannot update weights to more accurate quantities when the derivative is 0. However, it is very rare for the derivative to be 0 for all inputs.

2. **Leaky ReLU**

   $$\sigma(z) = max(\epsilon z, z)$$

   where ϵ is some small positive value. This fixes the problem of dead ReLUs but is not used very much in practice since it is rare for dead ReLUs to be a problem and since this is more expensive to compute.

3. **tanh**

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

   tanh smooths out discontinuities in the output by squashing all values to the range between -1 and 1. Since it is a smooth curve, it is always differentiable and never has a derivative

of 0. It used to be much more popular but has largely been replaced by ReLU due to its relatively larger computation requirements.

4. **Logistic Sigmoid**

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

Logistic sigmoid is very similar to tanh in that it smooths out discontinuities and always has a non-zero derivative. It also is rarely used nowadays, as it has largely been replaced by ReLU. Its difference from tanh is that it squashes values to the range between 0 and 1 and is thus good for producing probabilities.

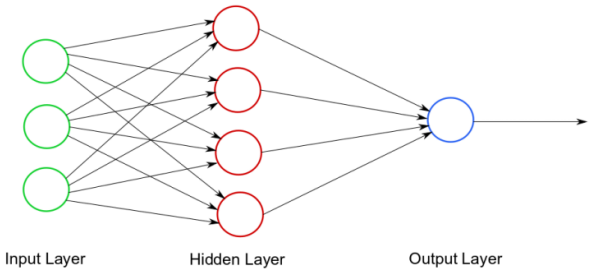
5. **Softmax**

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_j e^{z_i}}$$

Unlike the other non-linear functions, softmax is applied to an entire vector rather than the a single number. It guarantees that the output vector only contains entries between 0 and 1 and that all entries sum to 1. Thus, it is commonly used to predict a probability distribution and is typically used as the final layer in a network for classification tasks.

4 Neural Networks

Neural Networks are an extension to Multi Layer Perceptron. Multi Layer Perceptrons with added bias and non - linearity can be termed as Neural Networks.
Consider a simple neural network architecture with a single hidden layer and a non-linear function acting on the hidden layers.



Given an input feature x $\in$R^d the neural network predicts an output value $\hat{y}$. (We'll ignore the bias terms for simplicity)

The neural network model takes an input feature x with features $x_1, x_2, ..., x_d$ and passes it to the hidden layer to compute the hidden representations(non-linear features of the input) $h^1 \in R^k$. The input is connected to the hidden layer using weight matrix w^1 and each hidden layer node h_j has $w_j^1 \in R^d$ associated with it. Each hidden unit produces a hidden representation of $h_j \in R$ by

computing $h_j = \sigma(w_j^1 \cdot x^T)$ where j represents the j^t node of the hidden layer and σ represents the activation function. The final output is given by $\hat{y} = w^2 \cdot h$ where $\hat{y} \in R$ and w^2 represents the weight matrix connecting the hidden layer to the output.

Question : What are the dimensions of w^2 and w^1?

The prediction of $\hat{y}$ is also called forward pass. Given the input we output a value based on the existing weights and the activation functions. Neural networks or rather Deep Neural Networks have multiple hidden layers and $h^1, h^2, ...$ along with associated weights $w^1, w^2, ...$ for each of the layers. Multiple hidden layers are used to extract more powerful representation of the input features.

Question : What is the requirement to extract more powerful features?
2 **Question :** Can we just keep on increasing the number of hidden layers based on the task or will we face any issues if we just keep increasing the number of hidden layers?

4.1 Regression using neural networks

Regression in Neural Networks is similar to the forward pass. The task is to predict a value $\hat{y} \in R$ based on the input features.
The forward pass for the above neural network model can be written as

$$\hat{y} = w^2(\sigma(w^1 \cdot x^T))^T$$

4.2 Classification using neural networks

Classification in using neural Networks is a bit different as compared to Regression. For Classification the output is usually bounded between certain values. For binary classification the output is either $\{0, 1\}$ or $\{-1, 1\}$. For classification tasks predicting multiple categories(multi class predictions) one predicts $\hat{y} \in R^{Obj}$ where Obj represents the number of categories to predict. Each node in $\hat{y}$ outputs the confidence of the input belonging to a certain category.

One of the ways to interpret the output of a classification task is to predict the probability of the input falling in a particular category. For binary classification task the output is squashed between O and 1 using sigmoid function and for multi class setting softmax squashing function is used.
The forward pass can be written as

$$\hat{y} = \eta(w^2(\sigma(w^1 \cdot x^T))^T)$$

Where η represents the squashing function used to convert the confidence to probability scores

5 Loss Functions

Loss function tells us how well the algorithm performs and is required to estimate the change in the neural network parameters during each time step. We tend to maximize or minimize the loss

function based on the objective at hand.

Question : Can you think of when we would like to maximize the loss function instead of minimizing it?

In general we tend to minimize the loss function and stop updating the parameteres when the global minima is reached. From an optimization perspective, the objective of training a neural network can be written as

$$\min_{\mathbf{w}} L(y, h(x, w))$$

where 'w' represents the learnable parameters or the weights of the network, 'L' is the loss function, 'y' is the label associated with the input ,and h(x, w) is the prediction given the input and the weights.

There are many different loss functions which are being used in practice. Some of the most common one's are:

1. **Mean Squared Error**

$$\mathbf{L} = \frac{1}{s} \sum_{i=1}^{s} \|y_i - h\left(x_i, w\right)\|_2^2$$

   Used to evaluate continuous output and generally used for regression. The predicted value is compared to the ground truth label and averaged over the all the predictions.

2. **Cross-entropy**

$$\mathbf{L} = \frac{1}{s} \sum_{i=1}^{s} \left(- \sum_{j} y_{ij} \log h\left(x_i, w\right)_j \right)$$

   Cross entropy loss is generally used for classification tasks as measure to evaluate the confidence of the predicted output class. The labels are converted to one hot encoding and $y_{i,j}$ represents the value of the encoding for i^{th} data point and j^{th} class. The predictions $h\left(x_i, w\right)_j$ represent the probability score for predicting the j^{th} class for i^{th} data point.

3. **Cosine Similarity**

$$\mathbf{L} = \frac{1}{s} \sum_{i=1}^{s} \frac{y_i \cdot h\left(x_i, w\right)}{\|y_i\|_2 \|N\left(x_i\right)\|_2}$$

   Used to measure the similarity between two vectors. Used in the context of NLP to measure the similarity between two words, etc.

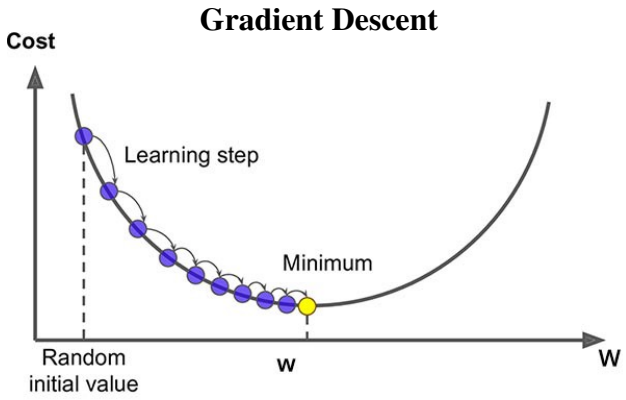
6 Backpropagation

Backpropagation or backward pass is used to update the parameters(weights and biases) of the neural network based on the output loss. Backprop is a technique where the parameters are updated based on their respective gradients.The goal of backprop is to minimize the loss on the samples available to us and can be done using gradient descent.

6.1 Gradient Descent

Gradient Descent minimizes the loss function by iteratively moving in the direction of the steepest descent as defined by the negative gradient. Gradient descent updates the parameters based on all the available data points which can be computationally expensive. Algorithms such as **mini-batch Gradient Descent** and **Stochastic Gradient Descent** consider only a part of the data while updating parameters.

Question : Is there any disadvantage to considering only a part of data as compared to all the data points?



Given a loss/cost function $l(y, \hat{y})$ where 'y' represents the true label and $\hat{y}$ represents the predicted label, the parameter update can be written as

$$w(t+1) = w(t) - lr * \frac{\partial \mathbf{L}(y, \hat{y})}{\partial w(t)}$$

Where w(t) is the current value of the parameter, w(t+1) is the value of the parameter after the update and lr represents the learning rate for the update. Learning rate dictates the size of the steps towards the global minimum. Gradient descent updates will move faster towards the global minimum with higher learning rates and vice versa.

6.2 Backprop equations

Gradient descent starts out with random initialization of the parameter and proceeds with computing the gradient updates based on the loss. Gradient updates are calculated based on the chain rule so as to propagate the derivatives backward through the network from the loss function.

6.2.1 Backprop for Regression

Consider a regression task using the single hidden layer neural network mentioned before. The parameters of this neural network are weights connecting the input layer to the hidden layer (w^1)

and the weights connecting the hidden layer to the output layer (w^2) .The predictions based on the i^{th} data point can be written as

$$\hat{y}_i = w^2(\sigma(w^1 \cdot x_i^T))^T$$

A suitable loss function for regression tasks is to use mean squared loss and we would like to find the suitable parameters which minimize the loss function for all the 'm' data points.

$$\min_{\mathbf{W}} \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$$

The gradient descent update for parameter for layer 'l' can be written as

$$w_{jk}^{(l)} = w_{jk}^{(l)} - lr * \frac{1}{m} \sum_{i=1}^{m} \frac{\partial (\hat{y}_i - y_i)^2}{\partial w_{jk}^{(l)}}$$

Where w_{jk} represents the weight associated with the j^{th} and k^{th} node of two connected layers.

While updating the parameters we need to compute the partial difference w.r.t to the weights connecting the different layers of the neural networks and this is where the partial differentiation comes in handy.
For w^2 the updates can be written as

$$\frac{\partial l(\hat{y}_i, y_i)}{\partial w_{jk}^{(2)}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial l(\hat{y}_i, y_i)}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{jk}^{(2)}}$$

Similarly for w^1 the update can be written as

$$\begin{aligned} \frac{\partial l(\hat{y}_i, y_i)}{\partial w_{jk}^{(1)}} &= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial l(\hat{y}_i, y_i)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}_i}{\partial w_{jk}^{(1)}} \\ &= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial l(\hat{y}_i, y_i)}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial h_{ij}^{(1)}} \cdot \frac{\partial h_{ij}^{(1)}}{\partial w_{jk}^{(1)}} \end{aligned}$$

where h_{ij} represents the value node 'j' of the hidden layer with respect to the i^{th} data point

6.2.2 Backprop for Classification

Gradient updates for classification tasks are similar to regression. The key differences are the squashing function and the loss function.

A squashing function such as sigmoid or softmax is applied to the output of neural network.The predictions based on the i^{th} data point can be written as

$$\hat{y}_i = \eta(w^2(\sigma(w^1 \cdot x_i^T))^T)$$

Generally cross entropy loss is used to measure the performance of the neural network. Cross entropy loss is given by

$$l(y, \widehat{y}) = \begin{cases} -\ln \widehat{y} & \text{if } y = +1 \\ -\ln(1 - \widehat{y}) & \text{if } y = -1 \end{cases}$$

The minimzation of loss function can be written as

$$\min_{W} \frac{1}{m} \sum_{i=1}^{m} (-y_i \ln \widehat{y} - (1 - y_i) \ln (1 - \widehat{y}))$$