CS 3600: Neural Network Project Hints

Osama Sakhi
Siyan Li
Christopher Scherban
Nitya Tarakad

Georgia Institute of Technology

Contents

1 Notation:

Here we'll define the terms used within the project and how they may relate to concepts from the course.

1.1 `inActs`

These are the inputs to a single Perceptron, and they take the form:

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_n \end{bmatrix}^T$$

1.2 `weights`

These are the weights belonging to a single Perceptron. They take the form

$$\mathbf{w} = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & \dots & w_n \end{bmatrix}^T$$

Note that this $\mathbf{w}$ has one more entry in it than `inActs`. This discrepancy will be explained in Section 2.1.

2 Components of a Neural Network

2.1 Perceptron

This is analogous to a "neuron", and it is a single unit within our overall network that takes as input all of the outputs from the previous layer, and applies some non-linear activation function to the weighted sum of its inputs.

Suppose we have chosen some non-linear activation function (described in Section 3), and we call it $g(x)$. This $g(x)$ takes in a weighted sum of the inputs and a bias term. This is where our weights vector gets that extra weight w_0, and it's called the bias. Think of the bias term as the y-intercept in the equation $y = mx + b$.

The output of the Perceptron can be define as a, where:

$$a = g(w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n)$$

This is often expressed of as a dot-product between the vectors $\mathbf{w}, \mathbf{x}$, where we define $x_0 = 1$ for all inputs:

$$a = g(w_0x_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n) = g(\mathbf{w}^T\mathbf{x})$$

It's important to be able to recognize the Perceptron output in this form since Neural Network computations are often expressed as vector and matrix computations, including their update rules.

2.2 Layers

Neural networks tend to have multiple different layers. Each network has an input layer at the beginning of the network and an output layer at the end. All layers that are not input or output layers are called the hidden layers. Every layer contains multiple Perceptrons, and are fully connected to the previous layer. This is to say that every Perceptron in a layer needs to process the entire combined output of the previous layer. See the Figure 1 for an illustration.

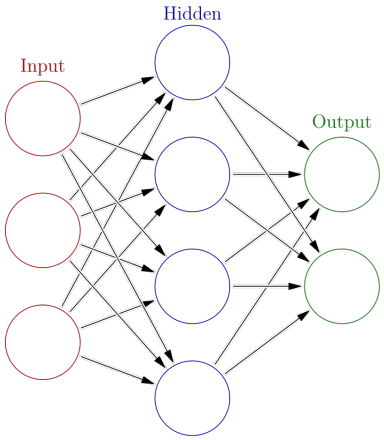


Figure 1: Layers in a Neural network. Note how every output in a previous layer is taken as input by every Perceptron in the next layer. Image taken from Wikipedia.

2.3 Depth as Expressiveness

The depth of a network, meaning the number of hidden layers used, corresponds to the expressiveness of a network. Assuming you have a reasonable number of Perceptrons in each of those hidden layers, the deeper your network, the "higher level" of a concept your network can learn. This has implications for training and testing accuracies as well as for overfitting and underfitting. You'll be tasked with running several experiments and seeing how the hidden layers affect performance of the network.

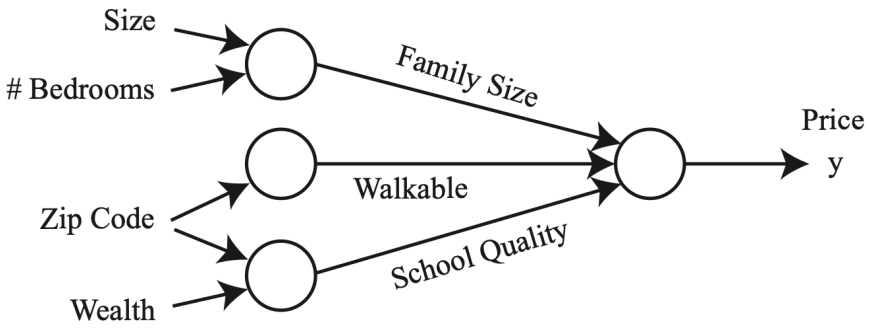


Figure 2: An example of "higher-level" features the intermediate layers of a network may learn from inputs about a real-estate dataset. If we add more layers, we may be able to extract other high level features such as credit score, occupation, and education levels. Image taken from Andrew Ng's CS229 Lecture Notes.

2.4 Loss Function

To train a network to improve performance, we need some measure of the performance. This measure is called a Loss Function, and there are many different types of these functions, each suited for different types of tasks. These functions are continuously defined and they are differentiable to allow back propagation. This means if we denote all the parameters of our network as W (meaning every weight and bias term in every Perceptron), we can compute the Loss using those parameters and call it $\mathcal{L}(W)$. Knowing that this function is differentiable, we can update the individual weights w_i throughout the network in the following way:

$$w_i = w_i - \alpha \frac{\partial \mathcal{L}(W)}{\partial w_i} \tag{1}$$

The expression above is updating each w_i by taking a "step" in the direction that will decrease the loss, $\mathcal{L}(W)$. This technique is called Gradient Descent, and it's used extensively in Machine Learning. In Section 5 we'll see how this partial derivative component is computed for every weight in our network.

The parameter alpha above is the size of the "step", and it's also referred to as the "learning rate".

3 Non-Linear Activation Functions

Activation functions in neural networks were inspired by how actual neurons fire in the brain. When a neuron receives a level of stimulus that surpasses a certain threshold, the neuron would "fire". The three most common choices for activation functions for Neural Networks are the Sigmoid, Tanh, and ReLU functions. For this project, you only need to implement Sigmoid and its derivative.

In order to use these functions in our network, they must be differentiable (otherwise we have no way of updating our network). We'll show the derivatives of each of these functions as well.

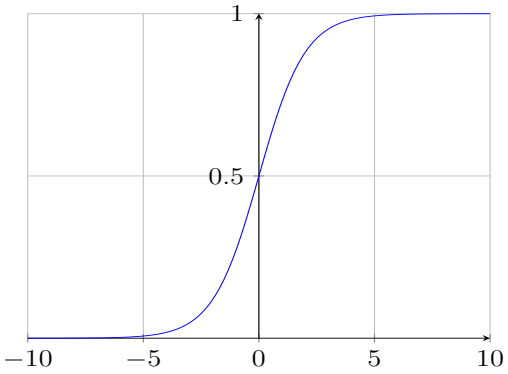
3.1 Sigmoid

3.1.1 Function

The sigmoid activation function is defined as

$$g(x) = \frac{1}{1 + e^{-x}}$$

As can be seen in the plot below, the range of the function conveniently is bounded by 0 and 1.



3.1.2 Derivative

The derivative of the sigmoid function is simply

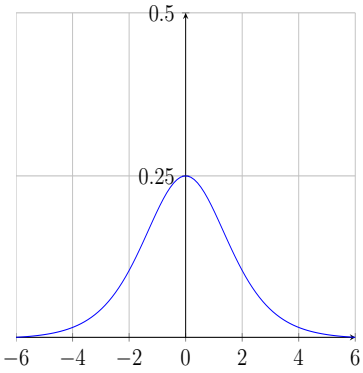
$$g'(x) = \frac{d}{dx}\frac{1}{1 + e^{-x}} = \frac{e^{-x}}{(1 + e^{-x})^2} \tag{2}$$

Using some algebra, we can simplify this derivative a bit:

$$\begin{aligned}
g'(x) &= \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{e^{-x}}{1 + e^{-x}} \cdot \frac{1}{1 + e^{-x}} \\
&= \frac{1 + e^{-x} - 1}{1 + e^{-x}} \cdot \frac{1}{1 + e^{-x}} \\
&= \left(\frac{1 + e^{-x}}{1 + e^{-x}} - \frac{1}{1 + e^{-x}}\right) \cdot \frac{1}{1 + e^{-x}} \\
&= \left(1 - \frac{1}{1 + e^{-x}}\right) \cdot \frac{1}{1 + e^{-x}} \\
&= g(x)(1 - g(x))
\end{aligned}$$

Thus, we have a convenient expression for the derivative: $g'(x) = g(x)(1 - g(x))$.

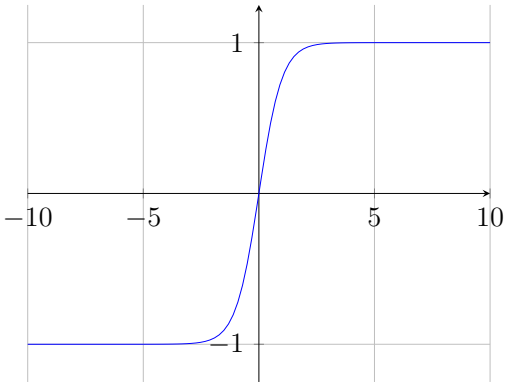
As shown in the plot below, the derivative is bell-shaped, as would be expected from the first visualization



3.2 Tanh

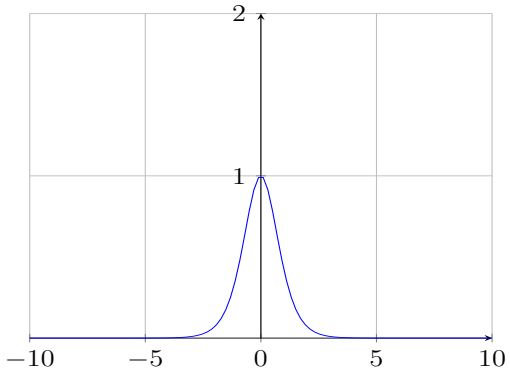
3.2.1 Function

$$g(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



3.2.2 Derivative

$$g'(x) = \frac{1}{cosh^2(x)} = \frac{4}{(e^x - e^{-x})^2} \tag{3}$$

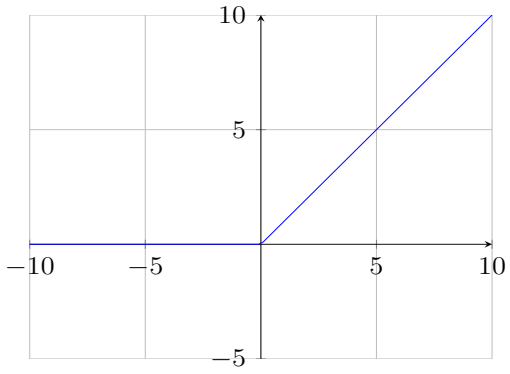


3.3 ReLU

3.3.1 Function

This function is defined as the max of the input x and the value 0.

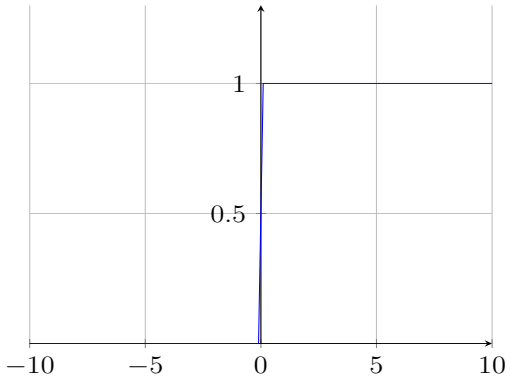
$$g(x) = \max(x, 0)$$



3.3.2 Derivative

Max functions technically don't have a derivative since the max function is not differentiable. Despite that, from the plot above, we can see that the slope of $g(x)$ is 0 when $x \leq 0$, and that it is 1 when $x > 0$. As a result, researchers use the following as the derivative for ReLU:

$$g'(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$$



4 Forward Propagation

When an input is fed into the neural network through the input layer, our goal is to generate a prediction at the output layer via a series of computations. As mentioned before, every perceptron in a layer must process all of the outputs from the previous layer. Every output from the previous layers has a corresponding weight in every perceptron in the current layer. The perceptrons in the current layer compute the weighted sum of the outputs from the previous layer, and then apply the activation function, resulting in outputs for this layer. This continues for all layers going forward.

5 Backpropagation

With backpropagation, we iterate backwards one layer at a time, changing the weights of each Perceptron in terms of its relation to the overall error, in other words, $\frac{\partial \mathcal{L}(W)}{\partial w_i}$.

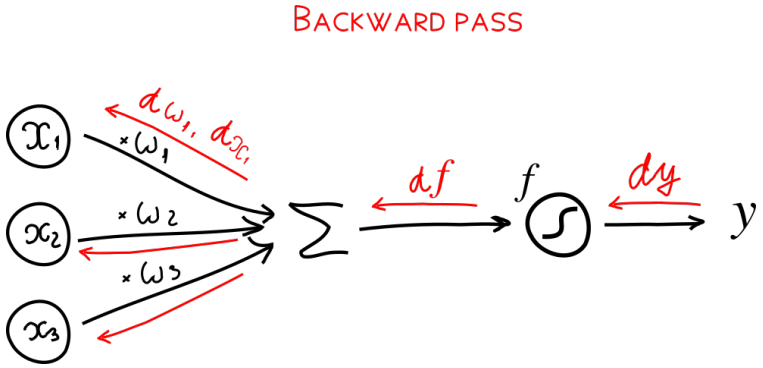


Figure 3: Backpropagation visualized. Image taken from Bogdan Penkovsky.

Since the output layer of the network relies on the composites of functions of the input, the chain rule is used to obtain $\frac{\partial \mathcal{L}(W)}{\partial w_i}$. For simplicity, we can use the psuedocode from Figure 4 to update the weights backwards from the end of the network to the start.

```
repeat
    for each weight w_{i,j} in network do
        w_{i,j} ← a small random number
    for each example (x, y) in examples do
        /* Propagate the inputs forward to compute the outputs */
        for each node i in the input layer do
            a_i ← x_i
        for ℓ = 2 to L do
            for each node j in layer ℓ do
                in_j ← Σ_i w_{i,j} a_i
                a_j ← g(in_j)
        /* Propagate deltas backward from output layer to input layer */
        for each node j in the output layer do
            Δ[j] ← g'(in_j) × (y_j − a_j)
        for ℓ = L − 1 to 1 do
            for each node i in layer ℓ do
                Δ[i] ← g'(in_i) Σ_j w_{i,j} Δ[j]
        /* Update every weight in network using deltas */
        for each weight w_{i,j} in network do
            w_{i,j} ← w_{i,j} + α × a_i × Δ[j]
until some stopping criterion is satisfied
return network
```

Figure 4: The psuedocode for forward and backward propagations. This snippet is taken from the Russel Norvig textbook.

At first glance, this is a pretty dense algorithm, however it can be thought of in four key stages.

5.1 Stage 1: Feed Forward

In this stage, we just run the feed forward process. Anywhere we see the term in_j, we're really just keeping the weighted some of some perceptron j, and $a_j = g(in_j)$corresponds to the output of that perceptron. This is important to have access to for the remaining three stages.

5.2 Stage 2: Compute Deltas for Output Layer Only

In this stage, we compute $\Delta[j]$ for every Perceptron j in the output layer. Remember how we talked about Loss Functions in Section 2.4? The expression here is the derivative of some chosen Loss function with respect to the output nodes.

$$\Delta[j] = g'(in_j)(y_j - a_j)$$

5.3 Stage 3: Compute Deltas for All Other Layers

In this stage, we work back through the network, from the 2nd-to-last layer down to the first hidden layer. At each layer, we'll rely on the next layer's deltas and weights. Let's dissect

this expression a bit further:

$$\Delta[i] = g'(in_i) \times \sum_j w_{i,j} \ \ \Delta[j]$$

Here, we're looking at some Perceptron i in one layer and trying to compute the value $\Delta[i]$. That summation we're computing looks a bit funky, but it's just a slight change of notation from what we've used so far. The $w_{i,j}$ term is the weight Perceptron j **in the next layer** has for the $i-th$ input it receives, which actually corresponds to the output of the Perceptron i in the current layer. That weight is then multiplied by the factor $\Delta[j]$, which has already been computed for the next layer.

5.4 Stage 4: Update All Weights in the Network

Lastly, we're applying updates to the weights in our network using the same concept as what we showed in Equation 1, but now with the Δs we computed in the previous stages.

$$w_{i,j} = w_{i,j} + \alpha \times a_i \times \Delta[j]$$

Remember, in this notation, $w_{i,j}$ represents the weight in the j-th Perceptron in some layer that is multiplied with the output of the i-th Perceptron in the previous layer.

6 Analysis

The goal of training a neural network on a training set is to make accurate predictions when incomplete pieces of information are given. However, a trained neural network may deviate from this goal in two ways. A network can become *overfitted* to the data. For example, if the neural network is trained on a dataset with only grey cats for too many times, the network would not classify white and black cats as cats since the network is overfitted. An overfitted network would have a high accuracy on the training set but a low accuracy on the test set because it does not know how to generalize. On the contrary, an *underfitted* neural network would have low accuracy on both the training and the testing sets, because it has not learned for enough epochs to "understand" all the different features.

7 Further Reading

Lastly, here are some links for understanding Neural Networks beyond the scope of just this project:

- Andrew Ng's CS229 lecture notes
- Perceptrons
- Neural Network Learning Algorithms
- Loss Functions for Neural Networks