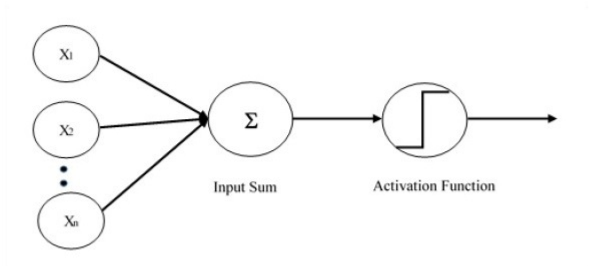
StartOnAI tutorial: Perceptrons, Deep Neural networks, and Hyperparameter optimization

Aurko Routh and Ayush Karupakula

July 2020

1 Single layer perceptrons

In neural networks, the most basic units are what we call single layer perceptrons. A common application of a perceptron is binary classification, where we try to classify our data into two categories. Let's dive in to the structure of a perceptron.



Essentially we define an input vector x which stores our input features for a single training example. Also, as depicted in the figure above with the lines, we have associated weights for each of the input features. We store this in vector w. We then proceed to calculate a weighted sum, including a bias term b as shown below:

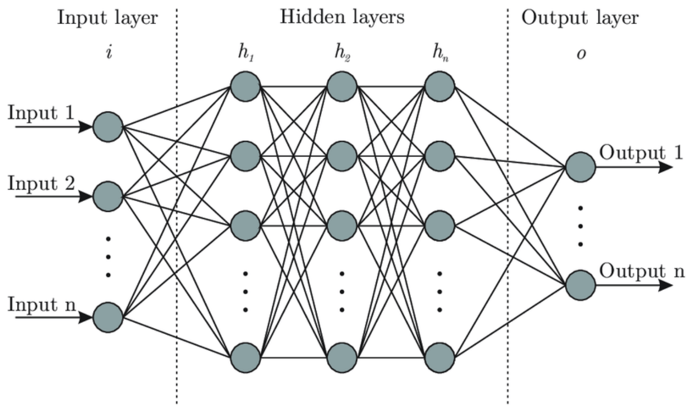
$$z = x^T w + b \tag{1}$$

After this we pass our input through an activation function g to achieve our final output a or $\hat{y}$. The equation below shows the process:

$$\hat{y} = a = g(x^T w + b) \tag{2}$$

2 Forward propagation for deep neural networks

When looking at single-layer perceptrons, we are referring to two layers, the input layer and the output layer. However, they are referred to as single-layer perceptrons because we are not including the input layer. However, we can also have multi-layer perceptrons called *deep neural networks*, which have layers in addition to the input and output layers called hidden layers. With a hidden layer, we can predict connections between inputs automatically. Surprisingly, we can increase the accuracy of our models greatly by incorporating hidden layers.



Looking at a single-layer perceptron, we can see that every neuron in the input layer has an arrow pointing to multiple neurons in the hidden layer. But in deep neural networks, the important thing to see is that each neuron actually points to every other neuron in the hidden layer. Because of this, we can compute the values for each of the neuron's in the hidden layer with all the neurons in the input layer if we know the weights. Every arrow that connects a neuron in the input layer and hidden layer in this figure has a weight associated with it. If we multiply each neuron's value with its corresponding arrow's weight (the arrow pointing to the neuron in the hidden layer), and sum all of these values, we can *almost* find the value of that neuron in the hidden layer. We must also factor in the bias that is associated with all the neurons in the hidden layer.

Using mathematical terms, we can describe the structure of a deep neural network and its associated data set. A training data set consists of m training examples and of course, m training labels for each of the training examples. As we have m training labels, we will use a vector of that length to store the labels, $Y = [y^{(1)}, y^{(2)}, ..., y^{(m)}]$.

We will define the number of layers in our deep neural network to be l. Keep in mind that this number will not include our input layer, hence the number of layers in a single-layer perceptron with an input and output layer being 1. By convention, we will refer to our input layer as the *0th* layer. Note that each layer will have a fixed number of neurons and this number does not have to be equivalent to the amount of neurons in the other layers. This is because all neurons in a previous layer will correspond to a neuron in the next layer. We do not have to have a one-to-one correspondence. The notation that we will sue to refer to the number of neurons in a given layer is $n^{[1]}$ neurons for the first layer, $n^{[2]}$ neurons for the second layer, and our final layer which has $n^{[l]}$ neurons. We will use brackets to refer to a specific layer. We can refer to the input layer as either n_x or $n^{[0]}$. We can use x because the letter x is often used to denote input.

We will define our input matrix to be $X \in \mathbb{R}^{n_x \text{x} m}$, where:

$$X = \begin{bmatrix} {x_1}^{(1)} & {x_1}^{(2)} & ... & {x_1}^{(m)} \\ {x_2}^{(1)} & {x_2}^{(2)} & ... & {x_2}^{(m)} \\ \vdots & \vdots & ... & \vdots \\ {x_{n_x}}^{(1)} & {x_{n_x}}^{(2)} & ... & {x_{n_x}}^{(m)} \end{bmatrix} \tag{3}$$

The subscripts refer to the inputs and the superscripts in parentheses refer to which training example the inputs correspond to. Hence, the superscripts go from training example 1 all the way through m and the subscripts go from 1 all the way through the number of input features.

Now, let's see how we will store the values of the neurons in the hidden and output layers. Let's look at a specific training example and the values of the neurons in the *ith* layer. We will use the vector $a^{[i]}$ to store the values of these neurons. In order refer to the value of a specific one of these neurons, we will use subscripts in the following manner $a_1^{[i]}$, $a_2^{[i]}$, $a_3^{[i]}$, ..., $a_{n^{[l]}}^{[i]}$. In order to refer to the values of the neurons across multiple training examples, we will store them in matrix $A^{[i]} \in \mathbb{R}^{n^{[i]} \text{x} m}$ where:

$$A^{[i]} = \begin{bmatrix} {a_1}^{[i](1)} & {a_1}^{[i](2)} & ... & {a_1}^{[i](m)} \\ {a_2}^{[i](1)} & {a_2}^{[i](2)} & ... & {a_2}^{[i](m)} \\ \vdots & \vdots & ... & \vdots \\ {a_{n^{[i]}}}^{[i](1)} & {a_{n^{[i]}}}^{[i](2)} & ... & {a_{n^{[i]}}}^{[i](m)} \end{bmatrix} \tag{4}$$

Notice that we must use two superscripts in order to denote the training example that we are referring to. We use superscripts in parentheses to refer to the training examples and brackets to refer to the layers.

Next, we will define our weights. In single-layer perceptrons, we were able to have a mostly convenient time when storing our weights in simple vectors. However, dealing with multilayer perceptrons is a different story for two particular reasons. When we were using single-layer perceptrons, we were able to use the same vector for each of the neurons because all of them corresponded to only one neuron in the next layer. However, with multilayer perceptrons, each of the neurons in the previous layer will have different weights associated with them because each and every neuron points to each and every neuron in the next layer. As a result, we need a vector of weights for each neuron in the hidden layer. We can store all of these vectors in one matrix. In addition, we will need multiple weight matrices because we can have more than one hidden layer. We denote the weight matrix connecting the input layer and first layer as $W^{[1]}$ and the weight matrix connecting layer $l-1$ and layer l is $W^{[l]}$. For the *ith* layer we have the matrix $W^{[i]} \in \mathbb{R}^{n^{[i]} \text{ x } n^{[i-1]}}$.

When looking at the weights associated with a particular layer stored in the matrix $W^{[i]}$, it's convenient to look at the various vectors that it consists of. Recall that the lengths of these individual vectors will be $n^{[i-1]}$ because there will be one weight for each of the values in the previous layer as each and every arrow connecting to each neuron has an associated weight. In a given layer, the weight vectors are described by $w_1^{[i]}$,$w_2^{[i]}$, $w_3^{[i]}$,..., $w_{n^{[i]}}^{[i]}$. There will be $n^{[i]}$ neurons for each of the vectors in the next layer because we need one vector of weights for each of the neurons in the layer.

The structure of $W^{[i]}$ can be represented by the following:

$$W = \begin{bmatrix} w_1^{[i]T} \\ w_2^{[i]T} \\ \vdots \\ w_{n^{[i]}}^{[i]T} \end{bmatrix} \tag{5}$$

In addition to the weights associated with each of the neurons in a given layer, we will need to define their biases. Similar to how we had a weight vector associated with each of the neurons in the hidden layers and the output layer, we will have a bias for each of the neurons. Instead of having a matrix for each layer like we had with weights, we will have one *vector* for each layer with the biases of each neuron.

The representation can be shown in the following:

$$b^{[i]} = \begin{bmatrix} b_1^{[i]} \\ b_2^{[i]} \\ \vdots \\ b_{n_{[i]}}^{[i]} \end{bmatrix} \tag{6}$$

With all of our matrices and vectors defined for all of the structures involved in forward propagation, we can easily carry through with a vectorized approach. In order to obtain $A^{[1]}$, we will need to perform two steps. First, we will need to define $Z^{[1]}$ so that we can store the product of the input X and weights $W^{[1]}$. The equation is simply:

$$Z^{[1]} = W^{[1]}X + b^{[1]}. \tag{7}$$

Notice how this vectorized approach is so much more efficient than it would have been if we had used an explicit for-loop with many steps in contrast to performing all calculations in one step.

Our second step is to pass this vector through a non-linear activation function. We can denote the activation function for layer i to be $g^{[i]}$. It follows that the equation for $A^{[1]}$ is:

$$A^{[1]} = g^{[1]}(Z^{[1]}) \tag{8}$$

All we have done in this step is apply the activation function element-wise to the vector Z.

A general forward propagation step for any given layer of a deep neural network is shown here:

```
for $i \leftarrow 1$ to $l$ do
    $Z^{[i]} = W^{[i]}A^{[i-1]} + b^{[i]}$
    $A^{[i]} = g^{[i]}(Z^{[i]})$
end
```

Now let's see how we will adjust the weights and biases using back propagation.[10.8]

3 Backward propagation for deep neural networks

In a deep neural network ,we define our cost function as $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, ..., W^{[l]}, b^{[l]})$. As a result, we tend to have much more complex calculations due to the more layered approach.

While this will not always be the case, for now, we will operate on the basis of using a neural network to solve a binary classification problem. As a result, we can make a few assumptions. Firstly, we will be using binary cross entropy for our loss error equation $L(a, y)$. As we are solving a binary classification problem, we can assume that $g^{[l]} = \sigma$ so that our output is a probability.

The cost function of all the parameters is defined as follows:

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, ..., W^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} L(A_i^{[l]}, Y_i) \tag{9}$$

In order to take the partial derivative of a parameter with respect to our cost function J, we can take the mean of the partial derivatives for all of the training examples. As a reminder, taking the partial derivative of any of our parameters with respect to J is simply the average of our partial derivatives with respect to L for all m training examples. In order to find the partial derivatives of all of the parameters with respect to J, we will use the following algorithms:

Algorithm 1: Backward propagation for l-layered neural network

```
$dZ^{[l]} = A^{[l]} - Y$
for $i \leftarrow l$ to 1 do
    if $i$ is less than $l$ then
        $dZ^{[i]} = W^{[i+1]}dZ^{[i+1]} * g^{[i]\prime}(Z^{[i]})$
    $dW^{[i]} = \frac{1}{m}dZ^{[i]}A^{[i-1]T}$
    $db^{[i]} = \frac{1}{m}$np.sum($dZ^{[i]}$,axis = 1, keepdims = True)
end
```

Just as a reminder, we use backpropagation to adjust our parameters by finding their partial derivatives with respect to the cost function and updating on the basis of the learning rate. The first line of the algorithm is dependent on the cost-function we use, in our case binary cross entropy. For different cost functions, the first line will differ. The following code highlights a process that can be generalized to any neural network.

An important thing to note to avoid any confusion is that we use shortened versions of the full derivative notations for convenience, especially while programming so that we can implement backpropagation easily. We use these shortened versions because they are all derivatives with respect to the same cost function anyways, so we will always know what parameter what we are referring to. The shortened versions of the partial derivatives are as follows: all versions of $\frac{\partial J}{\partial Z^{[i]}}$ are represented by $dZ^{[i]}$, all versions of $\frac{lJ}{\partial W^{[i]}}$ are represented by $dW^{[i]}$, and all versions of $\frac{\partial J}{\partial b^{[i]}}$ are represented by $db^{[i]}$.

In order to compute the partial derivative of $db^{[i]}$, we used a NumPy command that you may not be familiar with. Essentially, all we did was take the sum of each column, hence the "axis = 1" and store these values in the column vector $db^{[i]}$. We do this by storing the values row by row.

Algorithm 2: Deep neural network gradient descent

```
Result: Write here the result
while gradient descent has not converged do
    for $i \leftarrow 1$ to $l$ do
        $W^{[i]} := W^{[i]} - \alpha dW^{[i]}$
        $b^{[i]} := b^{[i]} - \alpha db^{[i]}$
    end
end
Here, we use the partial derivatives of our parameters with respect to
  the cost function and update them by also considering a learning rate
  which dictates how big our adjustments will be with each iteration.
```

4 Hyperparameter tuning

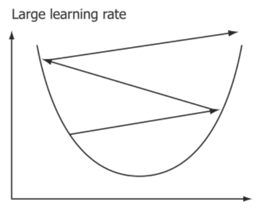
Hyperparameters are parameters set by the programmer which help to control the training of a machine learning model. Some common examples of hyperparemters include the learning rate, the number of layers, the activation functions, and the implemented optimization techniques. A very important step to building a very accurate neural networks is ensuring that the hyperparameters of the model are optimal for the specific problem the neural network is trying to solve. We will be exploring different hyperparemters in order to better understand how to tune a neural network.

5 Learning rate

When training a neural network, it is essential to set an adequate *learning rate*. The learning rate of the neural network, denoted by α, controls the speed at which a neural network learns when carrying out gradient descent. If we set α to 0.01, then we are essentially telling our neural network to adjusts the weights by 1% of the estimated error of the weights for each step in gradient descent.

If we set the learning rate to be too small, then our neural network may learn too slowly. This is primarily because decreasing α results in a more minuscule weight adjustment in our neural network. If the weights of the network aren't changing fast enough, the time it takes for gradient descent to converge to a local minimum significantly increases.

If we set the learning rate to be too large, we may see a divergence of gradient descent. This is because if we overshoot the error of the weights in our neural network, we overstep the local minimum and the cost-function value of our parameters will keep climbing up the gradient. The visual below illustrates this process:



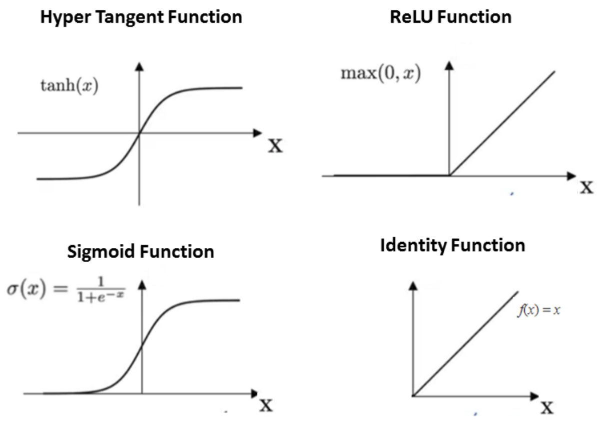
6 Exploring the uses of activation functions

Activation functions are essential to training neural networks. This is because they introduce non-linearity into the computations which allow neural networks to perceive complex, non-linear trends in the data. Applying the right activation functions is crucial to building a high accuracy neural network. Below we will explore some common activation functions:

Sigmoid is a common activation function that is used for an output layer in a binary classifier. This is primarily because the output range for sigmoid is between 0 and 1, allowing us to calculate the $p(x) = 1$ given a data example x. For hidden layers however, a strictly superior activation function is the tanh activation function which we will be talking about next.

The hyperbolic tangent, or tanh activation is similar to the sigmoid activation function but has a slightly different range of values. Tanh has roughly the same shape as sigmoid but with a range of -1 to 1 instead of 0 to 1. Tanh is better than sigmoid for hidden layers since it transforms the weighted sums of the neural networks to both positive and negative values. It also has the advantage of being zero-centered. A disadvantage of both the tanh and the sigmoid activation functions is that they suffer from the vanishing gradient problem as the magnitude of the supplied weighted sums increases. This may significantly slow down learning in the neural network.

The rectified linear unit, or RELU is an activation function with very special properties. Mathematically it is defined as $g(x) = max(0, x)$. The RELU function serves to replicate a biological neuron, where the neuron "fires" only after it reaches a certain threshold, in this case zero. Computing the derivative of RELU is very computationally inexpensive - as it is only 0 or 1 - and speeds up learning in a neural network. As a result, RELU is often the "go-to" activation function for hidden layers. One disadvantage of the RELU activation function is that if the value of the weighted sum being passed into the activation function is less than zero, the derivative is zero. This results in halted learning in zero or negative values.



7 L2 Regularization

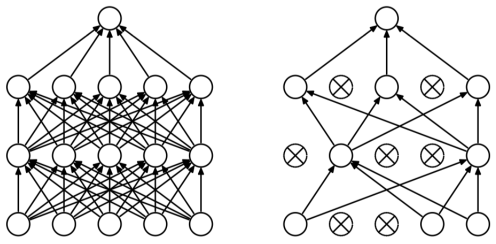
One problem faced by neural networks is the tendency to over-fit on the data. *Over-fitting* is the tendency to be susceptible to random noise and outliers in the training data. Overfitting is usually a consequence of computing a function that is too complex for a given training set. To solve this problem, we use a process known as *regularization.* The type of regularization we will be discussing in this section is L2 regularization. In L2 regularization we write the following for out neural network cost function:

$$Cost = Error + \frac{\lambda}{2m} \sum_{i=1}^{l} \|W^{[i]}\|_F^2 \tag{10}$$

As shown above in the equation above, we essentially add the square of the L2/Frobenius norm of our weight vectors to our cost function. Why does this prevent over-fitting? Since we are adding a term proportional to the square of all our weight parameters, the derivative of our cost-function with respect to the weights of the neural network increase in magnitude. This results in a greater decrease in the magnitude in the weights of our neural network when updating our parameters for gradient descent. Furthermore, with our hyperparameter λ, we can control the degree in which we want to regularize our neural network.

8 Dropout Regularization

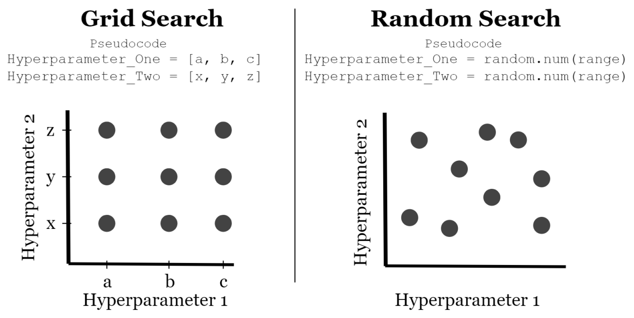
Another form of regularization is *dropout regularization*. In dropout regularization, we essentially "disable" neurons in the process of forward and backward propogation. In a neural networks, disabling a neuron simply equates to setting the activation value of the neuron to zero. To do this, we define a value *keep_prob* which denotes the probability that we keep the activation value of any given neuron in our neural network. In the process of forward propagation, we disable respective neurons using the probability *keep_prob*. Dropout reduces overfitting because it ensures that the network is not dependent on any single neuron or feature, but rather gives every neuron the chance to contribute to the final output of the network. This makes the neural network less susceptible to noise in the training data.



(a) Standard Neural Net (b) After applying dropout.

9 Grid search v.s Random search

For the numerical hyperparameters in our model, it is often hard to tell the optimal value of these hyperparameter. Unfortunatley in these cases, the only solution is to use trial and error for each individual hyperparameter. We can use two approaches to do this: *grid search*, and *random search*. In grid search, we first start off by defining a range of values for each hyperparameter and then we try every possible combination of values. On the other hand, random search is where we randomly try different combinations of hyperparameter values.


Grid Search
Pseudocode
Hyperparameter_One = [a, b, c]
Hyperparameter_Two = [x, y, z]
Hyperparameter 2
z
y
x
a
b
c
Hyperparameter 1
Random Search
Pseudocode
Hyperparameter_One = random.num(range)
Hyperparameter_Two = random.num(range)
Hyperparameter 2
Hyperparameter 1

It turns out that random search is almost always the superior method for searching hyperparameter values because in random search we can tell the general trend of the effects of certain hyperparameter values on the performance of the neural network without iterating over the entire search space. On the flip side, with grid search we may be searching many combinations that don't provide any valuable information on the optimal values of the hyperparameters.

10 Weight initialization techniques

In very deep neural networks a common problem is the vanishing gradient problem. The *vanishing gradient* problem results from very large positive or negative weighted sums being passed in the activation functions of a neural network. Furthermore, the derivatives of the parameters of the cost function exponentially decrease relative to the layer of the specific parameter. Vanishing gradients usually lead to much slower learning and make it difficult to converge to a local minimum in gradient descent.

So how do we solve this problem? One technique is to control the variance of our weights in the neural networks during initialization. Typically, when generating the weights of a neural network we generate it with a standard normal distribution. In a standard normal distribution, the mean is zero and the variance is 1. If the variance of our weights is controlled, we can avoid activation values of large magnitude that lead to the vanishing gradient problem in activation functions such as tanh or sigmoid.

When using the tanh activation function we use a technique called *Xavier initialization.* In Xavier initialization we want the following condition to be true:

$$Var(\text{weight parameter}) = \frac{1}{n^{[\text{previous layer}]}} \tag{11}$$

To set this condition true, we simply multiply our weights by $\sqrt{\frac{1}{n^{[\text{previous layer}]}}}$, which is the desired standard deviation.

For the RELU activation function we use a technique called *He initialization.* In He initialization we want the following condtion to be true:

$$Var(\text{weight parameter}) = \frac{2}{n^{[\text{previous layer}]}} \tag{12}$$

To set this condition true, we similarly multiply our weights by $\sqrt{\frac{2}{n^{[\text{previous layer}]}}}$, which is the desired standard deviation.

11 Batch, Mini-batch, and stochastic gradient descent

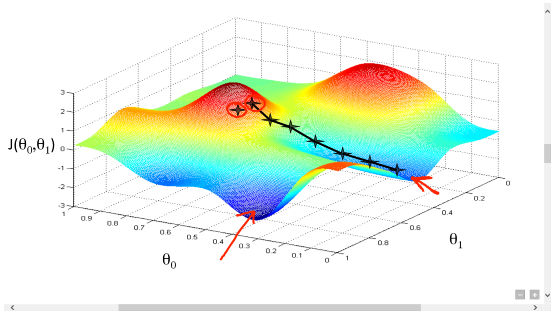
Gradient descent allows us to systematically optimize the cost function of our neural network. Gradient descent is carried out over many iterations, which we call epochs. There are three main types of gradient descent: batch gradient descent, mini-batch gradient descent, and stochastic gradient descent.

In *batch gradient descent*, we evaluate the loss for our neural network with respect to the entire training set using vectorization. As a result, we calculate the derivative for each parameter for the loss of all training examples.

In *stochastic gradient descent*, we compute the cost of each training example individually and update our parameters respectively.

Mini-batch gradient descent partitions the training set into mini-batches and loops through all mini-batches while vectorizing over a mini-batch.

While stochastic gradient descent gives feedback on the optimization quickly, it is unable to take advantage of vectorization since we use an explicit for-loop to go over every training example. On the other hand, we can take advantage of vectorization in batch gradient descent. However, batch gradient descent is often slow since it is very costly to compute cost over an entire training set. Mini-batch gradient descent helps to find a middle ground between the two.



12 References

1. StartOnAI book: A Guide to Machine Learning, Deep Learning, and their Applications
2. TJ Machine Learning club lectures 2019-2020
3. Andrew Ng, Deeplearning.ai Deep learning coursera specialization