

Neural Network Structure

Dr. Amjad Hawash



The problem

- Suppose we have some information about obesity, smoking habits, and exercise habits of five people.
- We also know whether these people are diabetic or not. Our dataset looks like this:

Person	Smoking	Obesity	Exercise	Diabetic
Person 1	0	1	0	1
Person 2	0	0	1	0
Person 3	1	0	0	0
Person 4	1	1	0	1
Person 5	1	1	1	1

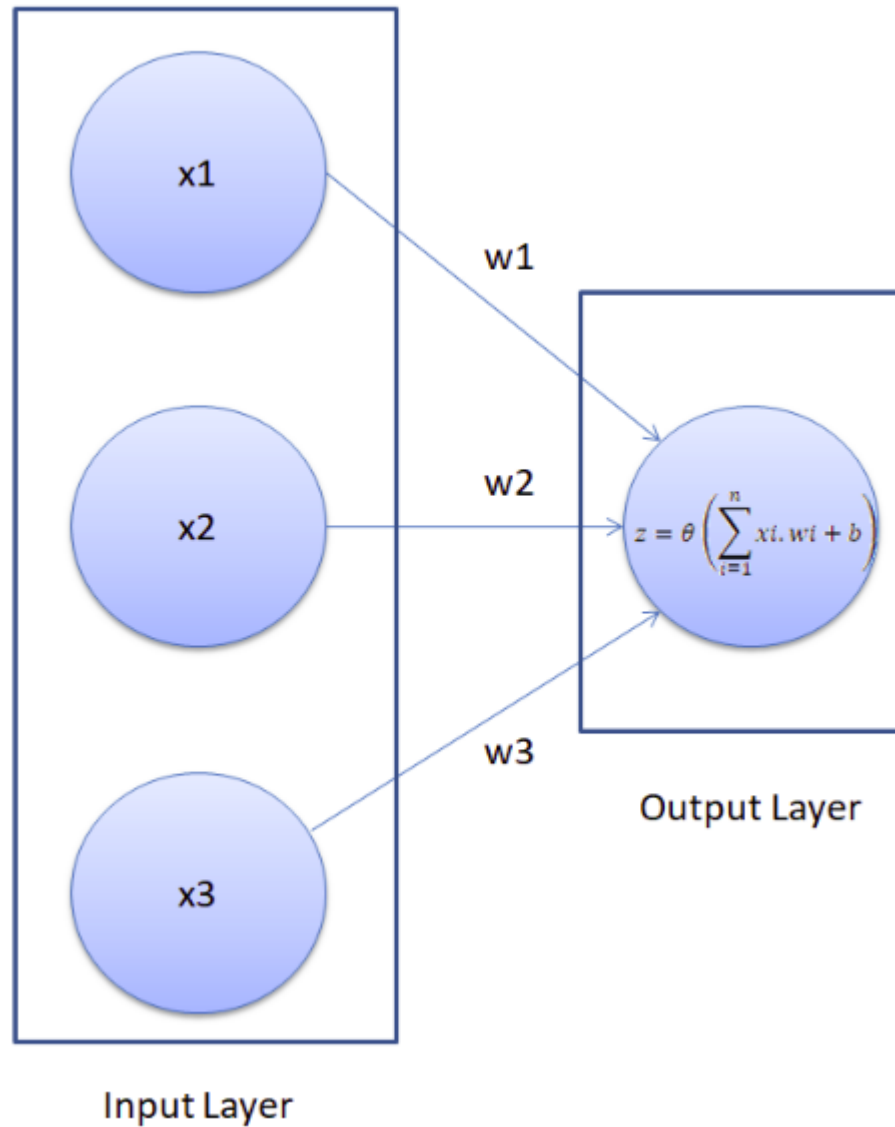


The problem

- It is clearly evident from the dataset that a person's obesity is indicative of him being diabetic.
- Our task is to create a neural network that is able to predict whether an unknown person is diabetic or not given data about his exercise habits, obesity, and smoking habits.
- This is a type of supervised learning problem where we are given inputs and corresponding correct outputs and our task is to find the mapping between the inputs and the outputs.



The Solution



$$X.W = x_1w_1 + x_2w_2 + x_3w_3 + b$$




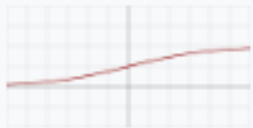
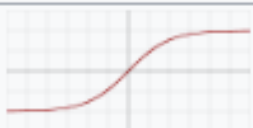
The problem

- The result from Step 1 can be a set of any values.
- However, in our output we have the values in the form of 1 and 0.
- We want our output to be in the same format.
- To do so we need an activation function, which squashes input values between 1 and 0.
- One such activation function is the sigmoid function.



Activation Functions (Sample)

- The sigmoid function returns 0.5 when the input is 0.
- It returns a value close to 1 if the input is a large positive number.
- In case of negative input, the sigmoid function outputs a value close to zero.

Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic (a.k.a. Sigmoid or Soft step)		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \text{ [1]}$
TanH		$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$



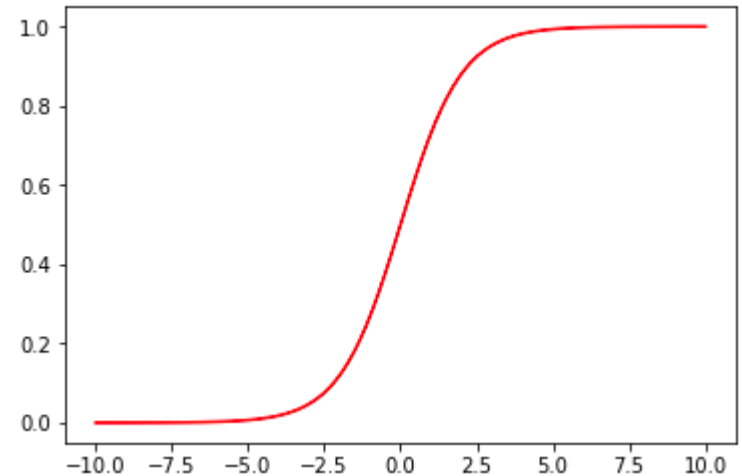
The problem

- Mathematically, the sigmoid function can be represented as: $\theta_{X.W} = \frac{1}{1 + e^{-X.W}}$
-
- `input = np.linspace(-10, 10, 100)`
- `def sigmoid(x):`
- `return 1/(1+np.exp(-x))`
- `from matplotlib import pyplot as plt`
- `plt.plot(input, sigmoid(input), c="r")`



The problem

- You can see that if the input is a negative number, the output is close to zero, otherwise if the input is positive the output is close to 1.
- However, the output is always between 0 and 1.
- This is what we want.



The problem

- This sums up the feedforward part of our neural network.
- It is pretty straightforward.
- First we have to find the dot product of the input feature matrix with the weight matrix.
- Next, pass the result from the output through an activation function, which in this case is the sigmoid function.
- The result of the activation function is basically the predicted output for the input features.



Back Propagation

- In the beginning, before you do any training, the neural network makes random predictions which are far from correct.
- We then compare the predicted output of the neural network with the actual output.
- Next, we fine-tune our weights and the bias in such a manner that our predicted output becomes closer to the actual output, which is basically known as "training the neural network".



The problem

- The first step in the back propagation section is to find the "cost" of the predictions.
- The cost of the prediction can simply be calculated by finding the difference between the predicted output and the actual output.
- The higher the difference, the higher the cost will be.



The problem

- There are several other ways to find the cost, but we will use the mean squared error cost function.
- A cost function is simply the function that finds the cost of the given predictions.
- The mean squared error cost function can be mathematically represented as:

$$MSE = \frac{1}{n} \sum_{i=1}^n (\text{predicted} - \text{observed})^2$$



The problem

- Our ultimate purpose is to fine-tune the knobs of our neural network in such a way that the cost is minimized.
- If you look at our neural network, you'll notice that we can only control the weights and the bias.
- Everything else is beyond our control.
- We cannot control the inputs, we cannot control the dot products, and we cannot manipulate the sigmoid function.



The problem

- In order to minimize the cost, we need to find the weight and bias values for which the cost function returns the smallest value possible.
- The smaller the cost, the more correct our predictions are.
- This is an optimization problem where we have to find the function minima.



The problem

- To find the minima of a function, we can use the **gradient decent algorithm**.
- The gradient decent algorithm can be mathematically represented as follows:

repeat until convergence : $\left\{ w_j := w_j - \alpha \frac{\partial}{\partial w_j} J(w_0, w_1, \dots, w_n) \right\} \dots \dots \dots (1)$



The problem

- Here in the above equation, J is the cost function.
- Basically what the above equation says is: find the partial derivative of the cost function with respect to each weight and bias and subtract the result from the existing weight values to get the new weight values.



The problem

- The derivative of a function gives us its slope at any given point.
- To find if the cost increases or decreases, given the weight value, we can find the derivative of the function at that particular weight value.
- If the cost increases with the increase in weight, the derivative will return a positive value which will then be subtracted from the existing value.

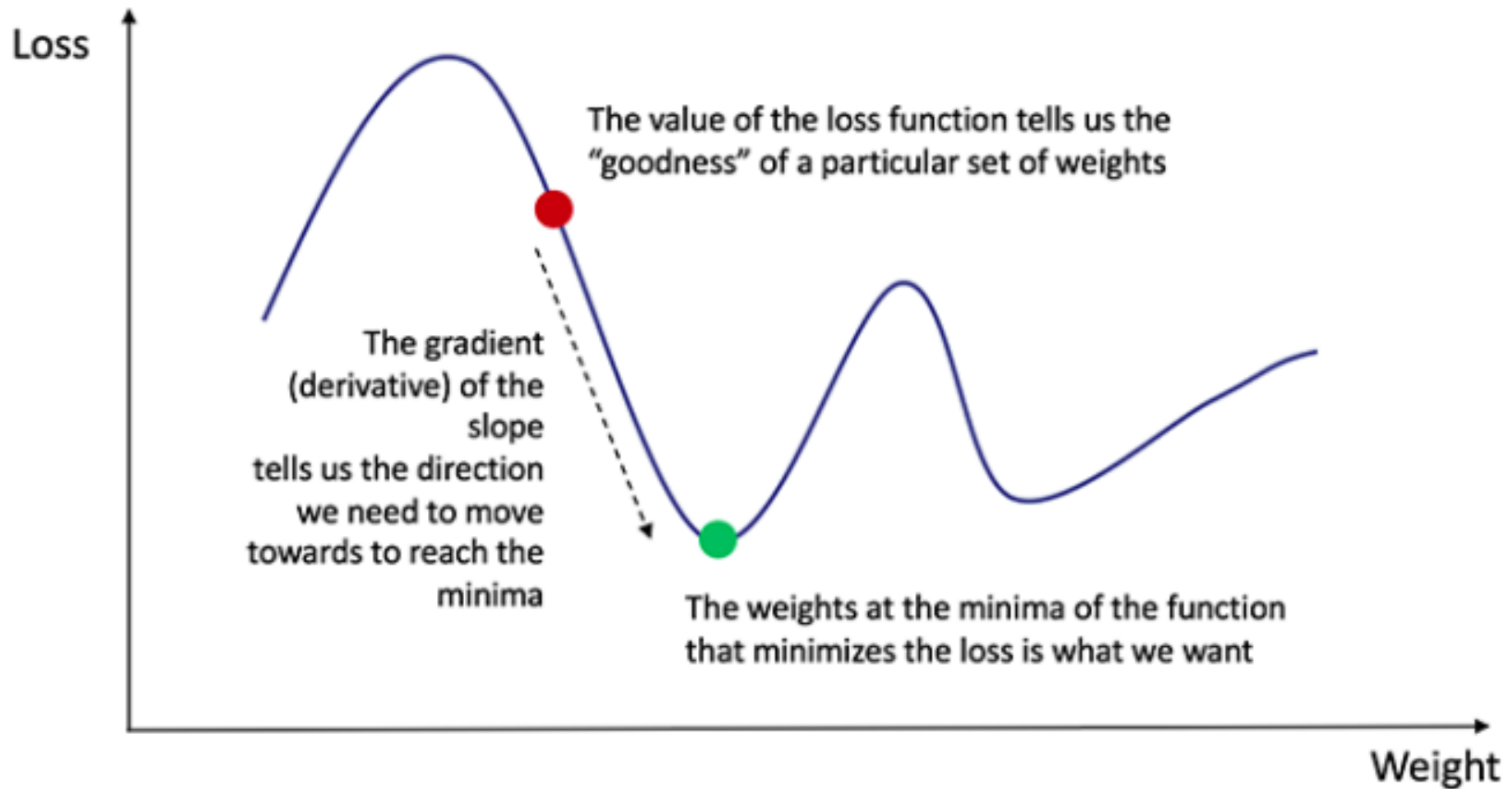


The problem

- On the other hand, if the cost is decreasing with an increase in weight, a negative value will be returned, which will be added to the existing weight value since negative into negative is positive.
- In Equation 1, we can see there is an alpha symbol, which is multiplied by the gradient. This is called the learning rate.
- The learning rate defines how fast our algorithm learns.



The problem



The problem

- We need to repeat the execution of Equation 1 for all the weights and bias until the cost is minimized to the desirable level.
- In other words, we need to keep executing Equation 1 until we get such values for bias and weights, for which the cost function returns a value close to zero.



The implementation

- `import numpy as np`
- `feature_set = np.array([[0,1,0],[0,0,1],[1,0,0],[1,1,0],[1,1,1]])`
- `labels = np.array([[1,0,0,1,1]])`
- `labels = labels.reshape(5,1)`
- In the above script, we create our feature set.
- It contains five records.
- Similarly, we created a labels set which contains corresponding labels for each record in the feature set.
- The labels are the answers we're trying to predict with the neural network.



The implementation

- The next step is to define hyper parameters for our neural network.
- Execute the following script to do so:
 - `np.random.seed(42)`
 - `weights = np.random.rand(3,1)`
 - `bias = np.random.rand(1)`
 - `lr = 0.05`
- In the script above we used the `random.seed` function so that we can get the same random values whenever the script is executed.



The implementation

- In the next step, we initialize our weights with normally distributed random numbers.
- Since we have three features in the input, we have a vector of three weights.
- We then initialize the bias value with another random number.
- Finally, we set the learning rate to 0.05.



The implementation

- Next, we need to define our activation function and its derivative (I'll explain in a moment why we need to find the derivative of the activation).
- Our activation function is the sigmoid function, which we covered earlier.
- The following Python script creates this function:
 - `def sigmoid(x):`
 - `return 1/(1+np.exp(-x))`



The implementation

- And the method that calculates the derivative of the sigmoid function is defined as follows:
 - `def sigmoid_der(x):`
 - `return sigmoid(x)*(1-sigmoid(x))`
- The derivative of sigmoid function is simply $\text{sigmoid}(x) * \text{sigmoid}(1-x)$.



The implementation

- Now we are ready to train our neural network that will be able to predict whether a person is obese or not.
 - inputs = feature_set
 -
 - # feedforward step1
 - $XW = \text{np.dot}(\text{feature_set}, \text{weights}) + \text{bias}$
 - #feedforward step2
 - $z = \text{sigmoid}(XW)$ # predicted outputs
 - # backpropagation step 1
 - $\text{error} = z - \text{labels}$ #amount of error
 - $\text{print}(\text{error.sum}())$
 - # backpropagation step 2
 - $\text{dcost_dpred} = \text{error}$
 - $\text{dpred_dz} = \text{sigmoid_der}(z)$
 - $z_delta = \text{dcost_dpred} * \text{dpred_dz}$
 - $\text{inputs} = \text{feature_set.T}$
 - $\text{weights} -= \text{lr} * \text{np.dot}(\text{inputs}, z_delta)$
 - for num in z_delta:
 - $\text{bias} -= \text{lr} * \text{num}$



The implementation

- In the first step, we define the number of epochs.
- An epoch is basically the number of times we want to train the algorithm on our data.
- We will train the algorithm on our data 20,000 times.
- The error is pretty much minimized after 20,000 iterations.
- You can try with a different number.
- The ultimate goal is to minimize the error.



The implementation

- Next we store the values from the `feature_set` to the input variable.
- We then execute the following line:
 - $XW = \text{np.dot}(\text{feature_set}, \text{weights}) + \text{bias}$
- Here we find the dot product of the input and the weight vector and add bias to it.
- This is Step 1 of the feedforward section.
- In this line:
 - $z = \text{sigmoid}(XW)$



The implementation

- We pass the dot product through the sigmoid activation function, as explained in Step 2 of the feedforward section.
- This completes the feed forward part of our algorithm.
- Now is the time to start backpropagation.
- The variable z contains the predicted outputs.
- The first step of the backpropagation is to find the error.
- We do so in the following line:
 - $\text{error} = z - \text{labels}$
- We then print the error on the screen.



The implementation

- Now is the time to execute Step 2 of backpropagation, which is the core of this code.
- We know that our cost function is:

$$MSE = \frac{1}{n} \sum_{i=1}^n (\text{predicted} - \text{observed})^2$$



The implementation

- We need to differentiate this function with respect to each weight.
- We will use the chain rule of differentiation for this purpose.
- Let's suppose "d_cost" is the derivate of our cost function with respect to weight "w", we can use chain rule to find this derivative, as shown below:

$$\frac{d_cost}{dw} = \frac{d_cost}{d_pred} \frac{d_pred}{dz} \frac{dz}{dw}$$



The implementation

- Here, $\frac{d_cost}{d_pred}$
- can be calculated as: $2(predicted - observed)$
- Here, 2 is constant and therefore can be ignored.
- This is basically the error which we already calculated. In the code, you can see the line:
 - `dcost_dpred = error # (2)`



The implementation

- Next we have to find: $\frac{d_pred}{dz}$
- Here "d_pred" is simply the sigmoid function and we have differentiated it with respect to input dot product "z".
- In the script, this is defined as:
 - `dpred_dz = sigmoid_der(z) # (3)`
- Finally, we have to find: $\frac{d_z}{dw}$
- We know that: $z = x_1w_1 + x_2w_2 + x_3w_3 + b$



The implementation

- Therefore, derivative with respect to any weight is simply the corresponding input.
- Hence, our final derivative of the cost function with respect to any weight is:
 - $\text{slope} = \text{input} \times \text{dcost_dpred} \times \text{dpred_dz}$
- Take a look at the following three lines:
 - $\text{z_delta} = \text{dcost_dpred} * \text{dpred_dz}$
 - $\text{inputs} = \text{feature_set.T}$
 - $\text{weights} -= \text{lr} * \text{np.dot}(\text{inputs}, \text{z_delta})$



The implementation

- Here we have the z_delta variable, which contains the product of $dcost_dpred$ and $dpred_dz$.
- Instead of looping through each record and multiplying the input with corresponding z_delta , we take the transpose of the input feature matrix and multiply it with the z_delta .
- Finally, we multiply the learning rate variable lr with the derivative to increase the speed of convergence.



The implementation

- We then looped through each derivative value and update our bias values, as well as shown in this script.
- Once the loop starts, you will see that the total error starts decreasing as shown below:

```
0.001700995120272485
0.001700910187124885
0.0017008252625468727
0.0017007403465365955
0.00170065543909367
0.0017005705402162556
0.0017004856499031988
0.0017004007681529695
0.0017003158949647542
0.0017002310303364868
0.0017001461742678046
0.0017000613267565308
0.0016999764878018585
0.0016998916574025129
0.00169980683555691
0.0016997220222637836
0.0016996372175222992
0.0016995524213307602
0.0016994676336875778
0.0016993828545920908
0.0016992980840424554
0.0016992133220379794
0.0016991285685766487
0.0016990438236577712
0.0016989590872797753
0.0016988743594415108
0.0016987896401412066
0.0016987049293782815
```



The implementation

- You can see that error is extremely small at the end of the training of our neural network.
- At this point of time our weights and bias will have values that can be used to detect whether a person is diabetic or not, based on his smoking habits, obesity, and exercise habits.



The implementation

- You can now try and predict the value of a single instance.
- Let's suppose we have a record of a patient that comes in who smokes, is not obese, and doesn't exercise.
- Let's find if he is likely to be diabetic or not.
- The input feature will look like this: `[1,0,0]`.



The implementation

- Execute the following script:
 - `single_point = np.array([1,0,0])`
 - `result = sigmoid(np.dot(single_point, weights) + bias)`
 - `print(result)`
- In the output you will see: `[0.00707584]`
- You can see that the person is likely not diabetic since the value is much closer to 0 than 1.



The implementation

- Now let's test another person who doesn't, smoke, is obese, and doesn't exercises.
- The input feature vector will be `[0,1,0]`. Execute this script:
 - `single_point = np.array([0,1,0])`
 - `result = sigmoid(np.dot(single_point, weights) + bias)`
 - `print(result)`
- In the output you will see the following value:
`[0.99837029]`
- You can see that the value is very close to 1, which is likely due to the person's obesity.

