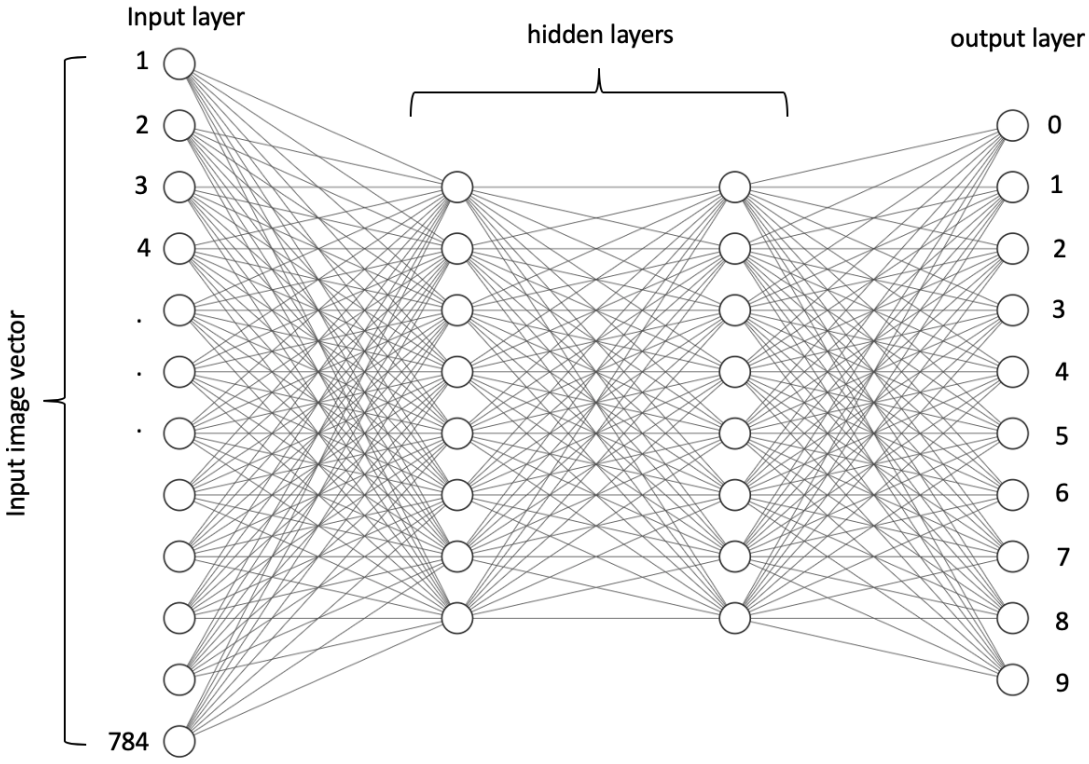
Under the Hood of Deep Learning

A step-by-step tutorial to understand neural networks

Mohamed Gharibi Jan 14, 2020 · 11 min read ★



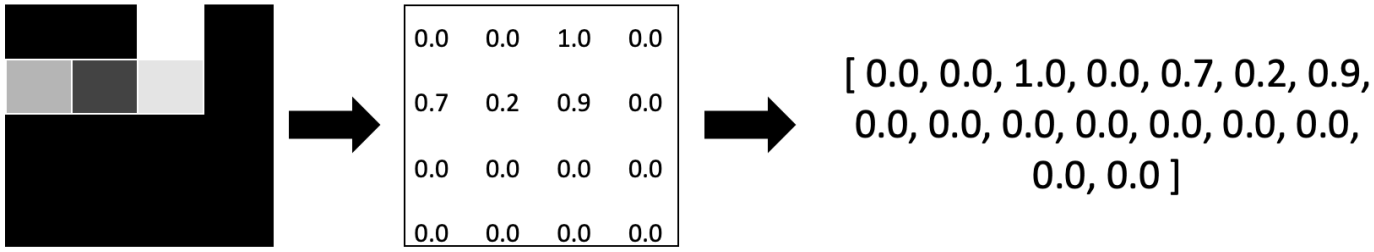
I used Alexlenail website to create this image

The previous image is a simple architecture for a deep neural network. The goal of this post is to understand deep learning details and build your own network, rather than use the existing models as a black box!

written digits (MNIST dataset). Currently, there are various types of neural networks, but for the sake of simplicity, we will start with the vanilla form (aka *"Multilayer Perceptron"*).

Please note that the circles in the previous diagrams called neurons. Each neuron contains a number that ranges between 0 and 1 which is known as "activation". Also, note that the lines between these neurons called weights. Each weight is a randomly generated number that holds a value between -1 and 1.

Each image in the MNIST dataset formed by black and white colors only and the dimension is (28 x 28 = 784 pixels). If you are still wondering why the range of neurons activation between 0 and 1, that is because it holds the greyscale of an image where 0 is completely black and 1 is completely white. Finally, to enter an image in our neural network, we need to convert it into a vector of pixels by taking each line of pixels and append it to the previous one, check the following example:



Layers explanation

As mentioned above, the input of the network will be an image that is converted to a vector of pixels (784 exactly) which is the first layer in our neural network.

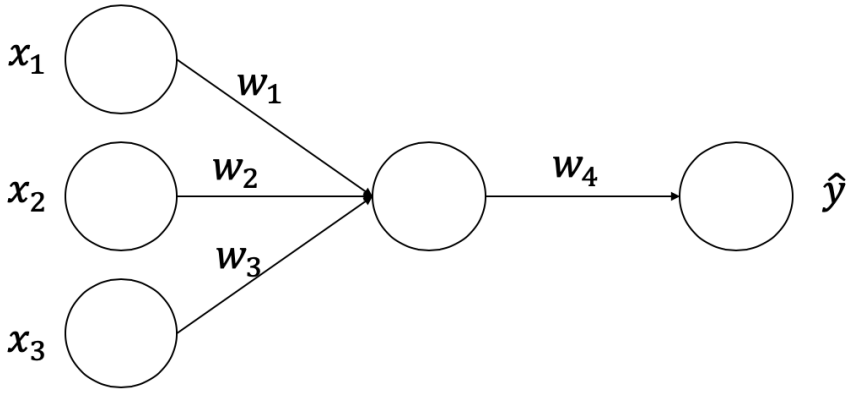
The number of output-layer neurons should be associated with the number of classes we are trying to predict. In our example, the number of classes is 10 (0, 1, 2, …, 9). Therefore, the last layer (output layer) should consist of 10 neurons.

The hidden layers are the layers in between. For now, let us just say that we chose the number of hidden layers and the number of neurons randomly.

What is the goal?

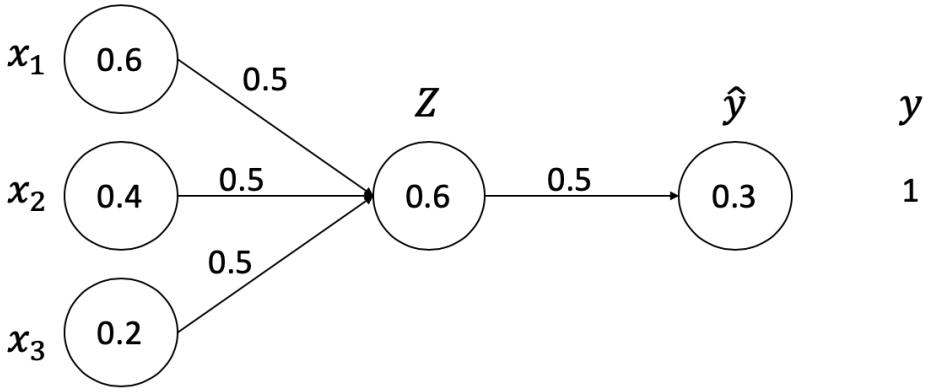
We want to build a model that is capable of predicting the digit of the input image (Image Classification). However, it is difficult to show a step by step example for the

network will look like this



Forward Propagation

Forward propagation is the process of starting with an input, going through the neural network and its calculations and ending up with making a single prediction (denoted by $y^$ and read as y-hat). Let us start with the following vector [0.6, 0.4, 0.2] and the actual output (y) is 1.



Where did all these numbers come from? Well, the input nodes were already given, Z is the result for multiplying the input activations with the weights and add them together, $y^$ is the result for multiplying Z with its weight. Finally, y is also given which is the desired output. Generally, to compute any output, we multiply the activations with the weights and add them together. This the formula for the first layer.

$$\hat{y} = x_1 . w_1 + x_2 . w_2 + x_3 . w_3$$

inputs.

$$\hat{y} = \sum_{i=1}^{n} x_i . w_i$$

The Bias

Each neuron will fire and light at some time. When a neuron fires it means this neuron detected a specific feature in the image. For example, every time an image with the digit 7 enters the network, almost the same neurons activate and fire (it means they triggered a similar event, similar angles, etc.). However, you do not want every neuron to fire when the activation is more than 0 (otherwise all the positive activations will keep firing). You want them to fire up after some threshold, say 10, this is known as the bias (*b*). We add the b to our summation to control when neurons fire.

$$\hat{y} = \sum_{i=1}^{n} x_i . w_i + b$$

Activation Functions

So far, our equation will produce good results. However, the results may be less than 0 or more than 1. As mentioned earlier, each activation should be in the range 0 to 1 since that is the greyscale of each image. Therefore, we need a function *(f)* to squash the result of $y^$ between 0 and 1. This function is known as the activation function, Sigmoid in particular. Calling Sigmoid on $y^$ will end up with the following

$$Z = f\left(\sum_{i=1}^{n} x_i . w_i + b\right)$$

Or you can write it as the following

$$Z = f(\hat{y})$$

we have many different activation functions that we may apply. Here is a list of the commonly used activation functions:

- **Sigmoid:** is a function that transfers the output between 0 and 1 and it is used in probabilities a lot since that is the range of probabilities.

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

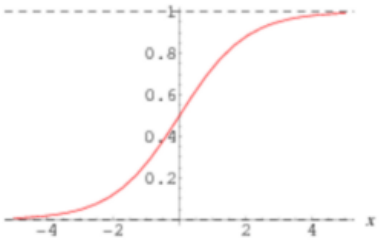


Image by WolframMathWorld

- **Tanh or hyperbolic Tangent:** is similar to Sigmoid somehow and it ranges from -1 to 1.

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

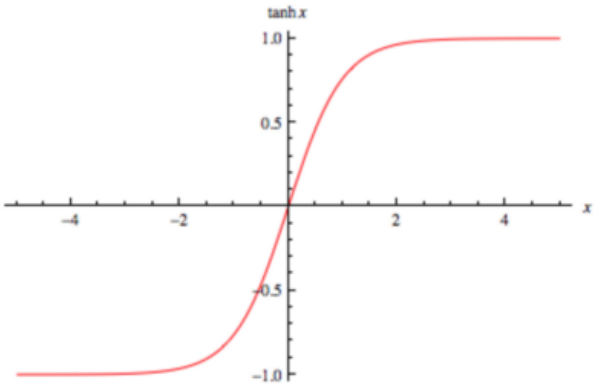


Image by WolframMathWorld

- **Rectified Linear Unit (RELU):** RELU is the most used activation function. Its range is 0 to infinity. If the input is negative, RELU returns 0, otherwise, it returns

$$f(x) = \begin{cases} 0 & for\ x \leq 0 \\ x & for\ x > 0 \end{cases}$$

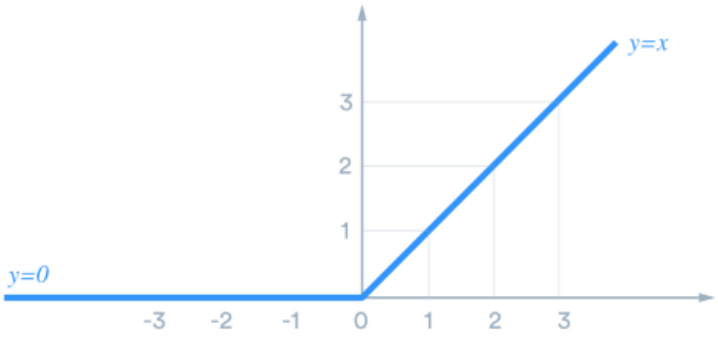


Image by Danqing Liu — Medium: RELU

- **Softmax:** is a unique activation function where it takes a vector of *k* numbers and normalizes it into *k* probabilities. In other words, instead of choosing a single output class, it lists the probability for each class.

$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \quad for\ i = 1, \ldots, J$$

There are many other activation functions such as Leaky RELU, Parametric RELU, SQNL, ArcTan, etc.

Calculating the Loss

What is the loss? In simple words, the loss is how far the model from predicting the correct answer. As seen before, the output is 1, whereas the predicted output is 0.3, so the loss in our case is 0.7. If the model prediction is perfect, then the loss is 0. So it is possible to calculate the loss using the following equation (*L* denotes for Loss)

$$L = (y - \hat{y})$$

Is that it? Basically, yes but there are few things that help to improve the loss in a way to be more beneficial for us in the future:

is -100. Adding these errors and averaging them gives you 0 which means your prediction is 100% correct which is not. Therefore, we are interested in the positive errors only.

$$L = |\, y - \hat{y} \,|$$

- The squared value: Consider having two errors (a small error and a big error). Which error would you pay more attention to? The bigger error of course! Because it affects the results more. Therefore, calculating the squared value for the loss helps us getting rid of the sign and getting the big errors bigger and the small errors smaller. Consider having the errors 0.01 and 100. By squaring these errors we get 0.0001 and 10000. Prioritizing such errors is really important to know what is causing a bad prediction.

$$L = (y - \hat{y})^2$$

- The summation: In our previous example, we calculated the loss between a prediction and an output, but what if we have several output neurons? Therefore, we calculate the summation between all the loss values in the neural network. (D denotes to the dataset that contains many examples).

$$L = \sum_{y \in D} (y - \hat{y})^2$$

- The average: The average of examples in our dataset. In our case, we had a single example. But now, we need to divide it by the number of examples (N) in D.

$$L = \frac{1}{N} \sum_{y \in D} (y - \hat{y})^2$$

This loss function is known as **The Mean Square Error (MSE)** and it is one of the most used loss functions. There are many other loss functions such as Cross-Entropy, Hinge,

for a single training example, whereas the cost function is the average loss over the entire training dataset.

Congratulations! We are done with the forward propagation. However, the prediction that we just made may not be very accurate (consider the output 1, but the model predicted 0.7). How can we make a better prediction? Well, we cannot change the input value, but we can change the weight!! Viola. Now you know the secret of deep learning.

Back Propagation

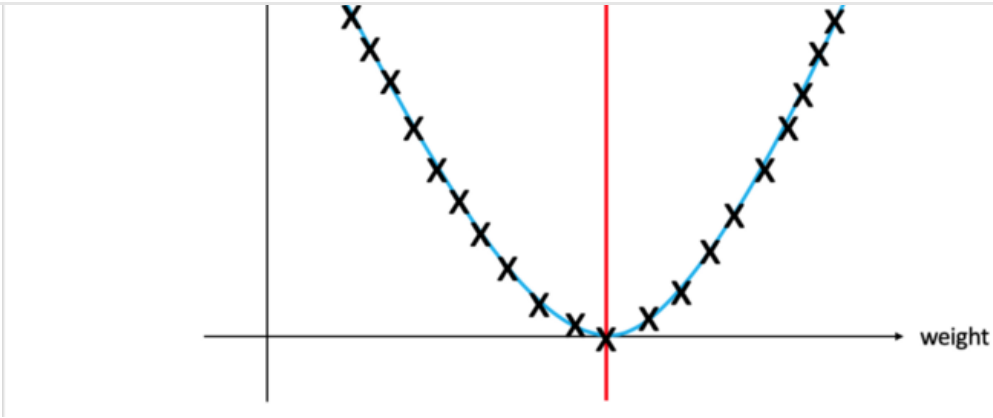
We cannot change the input. However, we can increase the weight, then by multiplying it with the input will give us a larger predicted output, say 0.8. Keep repeating this process (adjusting the weights) and will get better results. Going in the opposite direction to change the weight knows as backpropagation! But how can this be done exactly? Well, this can be done using optimization algorithms. There are different optimizers such as **Gradient Descent, Stochastic Gradient Descent, Adam Optimizer**, etc.

Gradient Descent

Gradient Descent is one of the optimization algorithms that aims to reduce the loss by adjusting the weights. Of course, changing weights manually is impossible (we have tens and hundreds of weights in a single neural network). So how can we automate this process? and how to tell the function which weight to adjust and when to stop?

Let us start adjusting the weights, checking how does that affect the loss and plot all the results (check the bellow plot). As you can see, at a specific point (the red line) is the minimum loss. To the left of the red line, we have to increase the weight to decrease the loss, whereas to the right of the red line, we obviously need to decrease the weight in order to reduce the loss. The main questions remain: How do we know for a given point if it is to the left or to the right of the red line (in order to know if we should increase or decrease the weight)? And how much we should increase or decrease the weight in order to get closer to the minimum loss? Once we answer this question, then we can reduce the loss and get better accuracy.

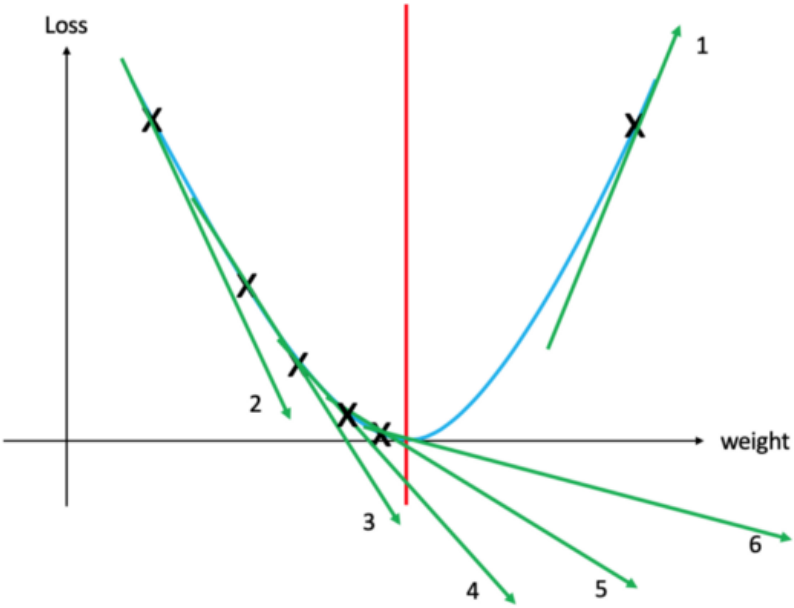
Please note in simple 2D dimensions it is easy to get to the minimum point quickly. However, most of the deep learning models deal with high dimensions.



Luckily, there is a way in Math to answer these questions, derivatives (what you ignored in your high school :)). Using the derivative, we can calculate the instantaneous rate of change for a tangent line on a graph using the derivative of Loss with respect to the Weight.

$$\frac{\partial L}{\partial W}$$

If you are not familiar with derivatives and the previous sentence sounded like gibberish, then just think of it as a way to measure the slope and direction of a line that touches the graph at a specific point. Based on the slope direction for a given point, we can know if the point exists to the left or to the right of the red line.



negative slopes. Also, note that the gradient of slope number 2 is bigger than the gradient of slope number 6. That is how we know how much we need to update the weight. The bigger the gradient is, the further the point is from the minimum point.

Learning Rate

After localizing the point to the left or right of the red line, we need to increase/decrease the weight in order to reduce the loss. However, let us say for a given point that exists to the left of the red line, how much should we increase the weight? Please note if you increase the weight significantly, then the point may pass the minimum loss to the other side. Then you have to reduce the weight, and so on. Adjusting the weight randomly is not efficient. Rather, we add a variable called "learning rate" denoted with "η" to control how the adjustment of weights. Generally, you start with a small learning rate to avoid passing the minimum loss. Why does it call the learning rate? Well, the process of reducing the loss and make better predictions is basically when the model learns. At that time, the learning rate is what controls how fast the model learns.

Adjusting the Weights

Finally, we take the slope amount multiplied with the learning rate and we reduce this amount from the old weight in order to get a new weight. You will understand it better by looking at the bellow equation

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

Stochastic Gradient Descent

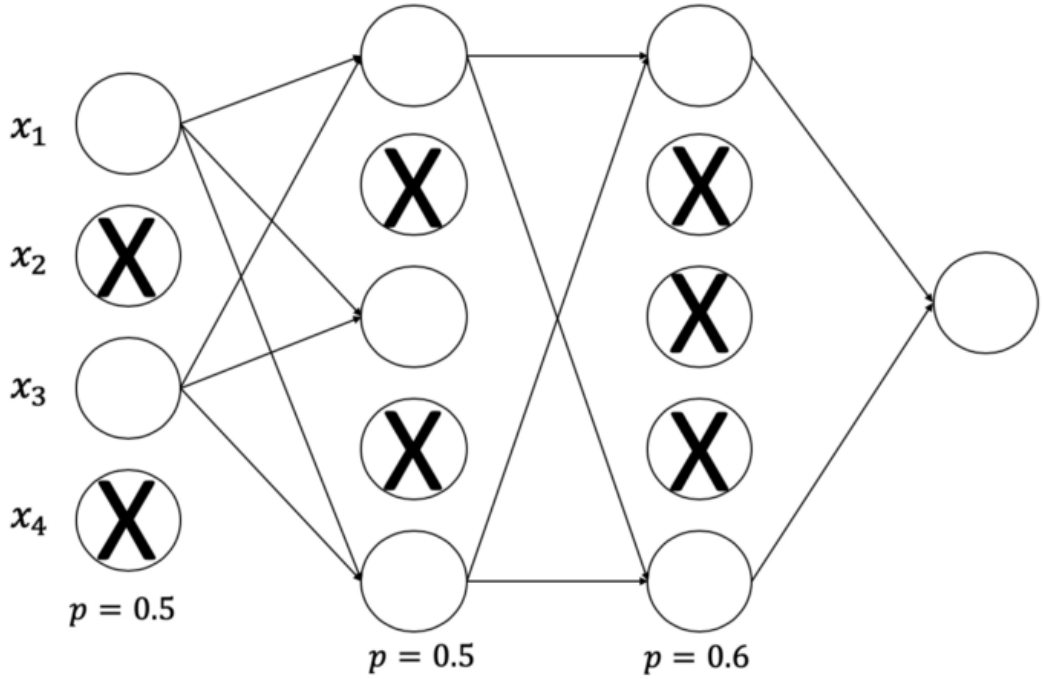
While gradient descent uses the entire dataset to compute the gradient, SGD uses a single example of the training dataset at each iteration. SGD typically reaches convergence much faster than batch or standard gradient descent. Batch Gradient Descent uses a batch of training examples each iteration.

Overfitting

When a neural network is very deep, it has too many weights and biases. When that happens, neural networks tend to overfit their training data. In other words, the model will be so accurate to a specific classification task, without generalization. The model

Dropout

A simple, yet efficient way to avoid overfitting is to use dropout. For each layer, there is a dropout ratio which means deactivate a number of neurons associated with this ration. These neurons will be chosen randomly and will be turned off during that specific iteration. Next iteration, another set of randomly picked neurons will be deactivated, and so on. This helps in generalization the model rather than remembering specific features.



I hope this post was helpful. Please let me know if you have any questions!

Resources

- Deep Learning. By Ian Goodfellow, Yoshua Bengio, and Aaron Courville.
- Grokking Deep Learning. By Andrew Trask.