Topic 3: Neural network basics

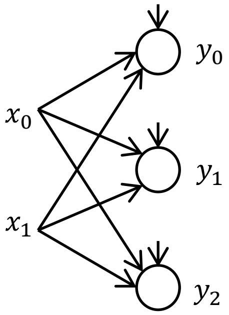
Neural networks are models that can be optimised to turn a tensor of numbers into another tensor of numbers. They are overly simplified models of how neurons in brains work and can be trained just like polynomials, by using gradient descent to optimise their parameters. The basic building block of the neural network is the neural unit, which is defined by the following equation:

$$y = f\left(b + \sum_i \boldsymbol{w}_i \boldsymbol{x}_i\right)$$

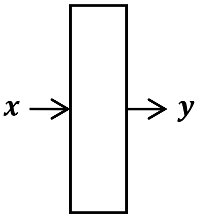
where $\boldsymbol{x}$ is a vector of input numbers, $\boldsymbol{w}$ is a vector of weights, b is a bias scalar, f is a non-linear activation function (a function which cannot be expressed in the form of a linear equation), and y is an activation value. The weighted sum and bias in the activation function is called the net value. This equation gives the output of a single neural unit. Neural units are organised into vectors of neural units called layers. This is the extended equation for a layer:

$$\boldsymbol{y} = f(\boldsymbol{b} + \boldsymbol{x}\boldsymbol{W})$$

where $\boldsymbol{W}$ is a weights matrix (with a row of weights for every activation in the layer), $\boldsymbol{b}$ is a bias vector (with a bias value for every activation in the layer), and $\boldsymbol{y}$ is the activation vector. We usually draw a neural network layer as a bi-partite graph. The following is a diagram representing a layer with 2 inputs and 3 output activations:



The circles are the neural units that output the activations of the layer, the 6 arrows on the left are the weights and the 3 arrows on top of each neural unit are the biases. A diagram can be further abstracted by just representing a layer as a single rectangle:



1. Activation functions

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/03_-_Neural_network_basics/01_-_Activation_functions.py

There are several possible non-linear activation functions. The logistic function, called sigmoid in deep learning literature, is a function that squashes all numbers to be between 0 and 1. It is useful for producing binary numbers, yes/no classifications, and fractions/probabilities. It is defined as:

$$y = \frac{1}{1 + e^{-x}}$$

Its steepest gradient is at $x = 0$, otherwise it becomes almost flat as x grows. The x in the above equation is called a logit. The activation value will never be exactly equal to 1 or 0 (although it does get rounded by your computer system when it gets very close in practice).

The hyperbolic tangent (tanh) is a function that squashes all numbers to be between -1 and 1. It is useful for producing signed fractions as well as representing binary numbers using -1 and 1 instead of 0 and 1. It is defined as:

$$y = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Its steepest gradient is at $x = 0$, otherwise it becomes almost flat as x grows. The activation value will never be exactly equal to 1 or -1 (although it does get rounded by your computer system when it gets very close in practice).

The rectified linear unit function (ReLU) is a function that is zero for all negative inputs and the exact same input for all positive inputs. It is useful for unbounded positive outputs such as prices or other free numeric values. It is defined as:

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

Its gradient is 0 for negative values of x and 1 for positive values of x, with $x = 0$ mathematically having an undefined gradient although Tensorflow sets it to 0.

The softmax function is a function that takes in a vector of arbitrary numbers, called logits, and returns a vector of the same length with positive numbers that sum to 1. It is useful for producing discrete probability distributions such as in multiclass classification and in producing resource distributions (how to distribute something among several slots). It is defined as:

$$\boldsymbol{y}_i = \frac{e^{\boldsymbol{x}_i}}{\sum_j e^{\boldsymbol{x}_j}}$$

An interesting property of softmax is that if you add the same constant to every element in the logits, the output probabilities will remain exactly the same. Also, none of the probabilities will ever be exactly equal to 1 or 0 (although they do get rounded by your computer system when they get very close in practice).

2. Logistic regression

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/03_-_Neural_network_basics/02_-_Logistic_regression.py

A layer with a sigmoid activation function can be used to create logic gates such as AND and OR. Try learning different truth tables by changing the training set. The heat map shows how the output of the layer changes as the two inputs are changed from 0 to 1. Red is for an output of 1 and blue for 0. What do you notice is common among all the heat maps when the truth table is changed? Can you learn an XOR function in with this setup (output 1 for unequal inputs and 0 for equal inputs)?

3. Two layer neural net

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/03_-_Neural_network_basics/03_-_Two_layer_neural_net.py

A single layer neural network can only learn a linearly separated input space. Two layers in series, where the output of the first layer acts as input to the second layer, can create differently shaped separations. In fact, a two layer neural network can learn to divide the input space into any shape, with the shape becoming more complex as the number of neural units in the first layer increases.

The first layer is called a hidden layer, because it is only for internal use. The second layer is called an output layer. You can continue adding more layers, but two layers can approximate any finite region of any binary function if the output layer uses a sigmoid activation function, or any real function if the output layer does not use an activation function.

In practice, however, the number of hidden neural units needed to approximate some desired function could be impractically large. With two hidden layers you can also approximate any desired function but with exponentially fewer parameters.

It is important to realise that each individual neural unit is still just performing a linear separation and that it is their combination that results in interesting functions. The code now trains a neural network with a single hidden layer consisting of two neural units. There are also three heat maps, the top is for the output neural unit and the bottom two are for each hidden neural unit. The bottom heat maps show what each hidden neuron is separating in the input space before the output neuron takes in this information to make a final decision.

4. Dropout

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/03_-_Neural_network_basics/04_-_Dropout.py

An interesting way to regularise a neural network is called dropout. It consists of randomly selecting neural units in a hidden layer or in the input and replacing its activation value with a zero. A probability is used to control how likely each unit is to be selected. This process is done (randomly) for every item in every minibatch during training. Replacing a neural unit with a zero effectively means that it is taken out of the model (for the training set items in which is it replaced with zero) since it will not contribute anything in the weighted sum that is used to calculate the next layer's activation. This has several useful effects that improve performance:

- One source of overfitting in neural networks is neural co-adaptation, which is when two or more neural units act together to perform one function as a team. This results in the layer performing a few complex functions rather than many simple ones. Complex functions are more likely to overfit than simple ones. With dropout, since no neural unit in a layer with dropout can rely on the contribution of any other neural unit in the same layer (as it might be dropped out at any point during training), each unit learns to perform an independent function which is useful in isolation.

- Given that the neural units are constantly being removed and added back, the result will be as if a different neural network is being trained for every training set item (assuming that the number of different combinations of non-dropped neural units is much larger than the training set size). This means that it's like you're training several different neural networks (a very large number of neural networks) and then combining all their outputs together into a single output. In machine learning this is called ensambling and it tends to improve performance.

- It's not just any ensambling that is being performed, it's ensambling of different neural networks that have shared parameters (since the same neural units appear in different 'virtual' neural networks). Also, each virtual network is trained on a different part of the training set, a process called data bagging in machine learning. Shared parameters and data bagging also tend to improve performance.

Of course after training we don't want to keep using dropout when we use the neural network for predictions as we'd like the predictions to be deterministic. On the other hand the neural network would not be optimised to work without dropout as, it turns out, that the weights would be adapted to handle only the typical fraction of present neural units in the layer with dropout. This means that if the layer had a dropout of 0.5, that is, about half of the neural units are randomly dropped out during training, then the next layer would only be able to handle half the neural units and would be overwhelmed with activations if all of the input units are present. Fortunately, this can easily be solved by making the non-dropped neural units output an activation which is twice as large as it should be whilst training and then make all the units output the correct activation after training. This will make the unmodified neural units then output half the activation that the next layer is used to but since there will be twice as many units, it all balances out.

5. Binary cross entropy

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/03_-_Neural_network_basics/05_-_Binary_cross_entropy.py

If the output layer uses a sigmoid activation function, then if you look at the gradient of the square error with respect to the net of the sigmoid you will see that it is only significant when the net is close to zero, otherwise the gradient is approximately zero. This means that if the net is not close to zero then gradient descent will work very slowly or even stagnates. A solution is to use a different error function made especially to fix this situation. It's called the binary cross entropy function and it works by taking the log of the sigmoid. Here is how it's defined:

$$E = -(t \ln y + (1 - t) \ln(1 - y))$$

where E is the error, t is the target output, and y is the actual output. Notice that this error function consists of two simpler error functions, one for when the target is 1 and one for when the target is 0 (or a linear combination of the two if the target is a fraction between 1 and 0). To select between these two sub error functions, a weighted sum is taken where depending on whether the target is 0 or 1, then one of the terms is multiplied by 1 and the other is multiplied by 0, resulting in a choice between one of the two sub functions.

6. Binary cross entropy 2

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/03_-_Neural_network_basics/06_-_Binary_cross_entropy_2.py

Another thing to notice about binary crossentropy is that the error function assumes that the output is never either exactly 1 or exactly 0, otherwise you'd have a log of 0. This is theoretically not a problem with sigmoid as it never equals 1 or 0, but in practice it could still equal one of these two numbers due to precision error. To avoid this problem we can work with the logits instead of the actual sigmoid output, which makes it possible to do some error correction. This is what the built-in Tensorflow binary cross entropy function expects.

7. Categorical cross entropy

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/03_-_Neural_network_basics/07_-_Categorical_cross_entropy.py

Binary cross entropy does not work with softmax, which has its own cross entropy error function called categorical cross entropy. In this case, the target value is the index of the softmax vector element that should have a value of 1 (approximately). Since softmax forces the outputs to sum to 1, increasing the target element also implicitly decreases the rest of the elements. This means that we just need to focus on the target element and let softmax handle the rest. Here is how categorical cross entropy is defined:

$$E = -\ln \boldsymbol{y}_t$$

where E is the error, t is the target index in the output vector, and $\boldsymbol{y}$ is the output vector.

8. Extra theory

- A neural network is just a normal algebraic function that can be written as an expression. This is what a 3 layer neural network looks like:

$$\boldsymbol{y}_i = f^3\left(\boldsymbol{b}_i^3 + \sum_j \boldsymbol{W}_{ji}^3 f^2\left(\boldsymbol{b}_j^2 + \sum_k \boldsymbol{W}_{kj}^2 f^1\left(\boldsymbol{b}_k^1 + \sum_l \boldsymbol{W}_{lj}^1 \boldsymbol{x}_l\right)\right)\right)$$

  where $\boldsymbol{y}_i$ is the ith output of the neural network, f^i is the activation function of the ith layer, $\boldsymbol{b}_j^i$ is the jth bias of the ith layer, $\boldsymbol{W}_{jk}^i$ is the weight connecting the jth neural unit in the (i-1)th layer to the kth neural unit in the ith layer, and x_i is the ith input of the neural network. In general, we can define the neural network as a recurrence relation:

$$\boldsymbol{a}_i^l = f^l\left(\boldsymbol{b}_i^l + \sum_j \boldsymbol{W}_{ji}^l\,\boldsymbol{a}_j^{l-1}\right)$$

where $\boldsymbol{a}_i^l$ is the ith activation of the lth layer and where $\boldsymbol{a}_i^0 = \boldsymbol{x}_i$. We can find the gradient of the output with respect to a weight or bias by using the chain rule:

$$\frac{\partial \boldsymbol{a}_i^l}{\partial \boldsymbol{W}_{jk}^m} = \frac{\partial \boldsymbol{a}_i^l}{\partial \boldsymbol{z}_i^l} \times \sum_n \boldsymbol{W}_{ni}^l \frac{\partial \boldsymbol{a}_n^{l-1}}{\partial \boldsymbol{W}_{jk}^m}$$

$$\frac{\partial \boldsymbol{a}_i^l}{\partial \boldsymbol{W}_{jk}^l} = \frac{\partial \boldsymbol{a}_i^l}{\partial \boldsymbol{z}_i^l} \times \boldsymbol{a}_j^{l-1}$$

where $\boldsymbol{z}_i^l$ is the ith net of lth layer and $\frac{\partial \boldsymbol{a}_i^l}{\partial \boldsymbol{z}_i^l}$ is the partial derivative of activation function in the lth layer with respect to its input. In the above two equations, the first is for the case when the weight you're trying to find the gradient with respect to is deeper than the current layer you're at whilst the second is for the case when it's at the current layer.

It is clear that to find the gradient of a weight you need to know the activation value of the neural unit connected to it. In fact, we can find the gradient by first finding the activations of every layer from input to output and then work out the gradient backwards layer by layer from output to input, using the pre-computed activations throughout. This is called the backpropagation algorithm. This algorithm finds the derivative of all the parameters in a neural network by using a forward pass (computing the activations) followed by a backward pass (computing the gradients using the activations).

- It is well known that gradient descent is vulnerable to local minima. Is this a problem in neural networks? It turns out that it is not. One scientific paper compares optimising neural networks using gradient descent and simulated annealing. Simulated annealing does not use gradients or any local directional guidance to find the minimum and so is not particularly vulnerable to local minima. It turns out that gradient descent always outperforms simulated annealing. Another paper analyses the path taken by gradient descent as the neural network goes from initial parameters to final parameters. Rather than trying to visualise the exact path through parameter-space, the authors simply follow a linear path from initial to final parameters by interpolating a line between the two points in parameter-space. It turns out that the error of the neural network as it changes parameters on this line keeps decreasing all the way until the final parameters are reached, without ever increasing. These results are evidence that there is something particular about neural network parameters that makes them easy to optimise by gradient descent and that local minima are good enough in terms of error.
- The equations of the derivative of the neural network above show that the gradient of the output with respect to a parameter involves a long series of multiplications of the gradient of the activation functions and of the weights at each layer. This means that as you go further backward in a long chain of layers, the gradient with respect to a parameter can become vanishingly small if these factors are fractions (which is the case with sigmoid and tanh), hence making gradient descent impossible, or become explosively large if they are greater than 1, hence resulting in

overflow errors. These two problems are called the vanishing gradient problem and the exploding gradient problem, respectively.

- The vanishing gradient problem can be solved in several ways. The most common solution is to avoid using sigmoid or tanh as an activation function for hidden layers and to instead use ReLU, which has a gradient of either 1 or 0 with nothing in between. This means that as long as there is a chain of 1s in the gradients from output to input layer then the gradient will flow easily, allowing gradient descent to follow. Of course, if there is a single 0 then this will not work and it can also result in dead ReLUs where a large percentage of activations in the neural network never activate for any of the training set items. To fix this we can use leaky ReLUs which replace the flat 0 in ReLU with a shallow line, that is, something like $y = \max(0.01x, x)$, or even parametric ReLU where a separate trainable variable is used to control how shallow the line should be, that is, $y = \max(px, x)$. See this Tensorflow page for functions that compute these activation functions.

  Another solution for the vanishing gradient problem is to use skip connections, which are connections between non-adjacent layers, for example connecting the first and third layers through an extra direct connection. If you just add the other layer's activations to the weighted sum of the later layer then you have created residue connections. Residue connections can be upgraded into highway connections by using an extra layer to provide a gate value. A gate value is a sigmoid fraction which is produced based on the activations of the earlier layer. This gate value is then multiplied by the activations of the same earlier layer prior to including them into the later layer. This allow the neural network to learn to control whether to include the activations from the earlier layer (by making the sigmoid value 1) or to leave them out (by making the sigmoid value 0).

- The exploding gradient problem is usually solved by gradient clipping, that is, clipping the gradient if its value is too large. Usually it is the norm of the gradient that is clipped rather than doing it elementwise. Clipping does not interfere with training as the direction in which the optimiser updates the parameters will still be the same, just at a slower pace than it would move otherwise. This StackOverflow answer shows how to apply it in Tensorflow.