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.

It all started in 1943 when McCulloch and Pitts published a paper with a primitive description of how neurons worked. Computers were still too basic to run neural networks and it took until 1958 for Rosenblatt to create the perceptron, a computational model of a single neuron that could perform tasks such as shape recognition. Unfortunately, there was a lot of hype on what these models would do, such as claiming that they could recognise tanks from trucks. This turned out was only possible because the images use of the tanks were taken in sunny weather whilst those of the trucks were taken in cloudy weather (which means that pixel intensity was all that was needed to distinguish the two).

It all came crashing down in 1969 when Minsky and Papert published their mathematical proof that perceptrons were not even able to classify whether two bits were the same or not. They knew that connecting several perceptrons in a network could solve this problem, but since perceptrons were not differentiable, there was no efficient algorithm available for optimising networks of perceptrons. This led to funding for research in neural networks being slashed and consequently very few advances were made. Thankfully, eventually differentiable neural units were developed and in 1986 Rumelhart, Hinton, and Williams published the Backpropagation algorithm, which is an efficient algorithm to calculate gradients in neural networks and thus be able to use gradient descent to optimise complex networks.

Even so, performing more complex tasks on realistically sized inputs would require neural networks that were too large and slow to run on computers at the time. It took several decades for computer hardware to be powerful enough to meet these requirements but eventually they did, thanks to GPUs, and in 2012, Krizhevsky, Sutskever, and Hinton made neural networks popular again with the neural network AlexNet which performed object recognition on the ImageNet dataset better than every other model by a wide margin.

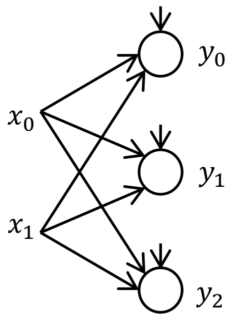
So how do neural networks work? The basic building block of the neural network is the neural unit, which is defined by the following equation:

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

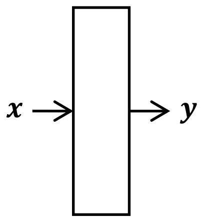
where $\boldsymbol{x}$ is a vector of input numbers, $\boldsymbol{w}$ is a vector of weights, b is a bias scalar, f is a non-linear activation function (a function which cannot be expressed in the form of a linear equation), and y is an activation value. The weighted sum and bias in the activation function 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 four most commonly used ones are sigmoid, tanh, ReLU, and softmax. The provided script plots each of these functions as a graph.

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 due to precision error 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 due to precision error in practice).

The rectified linear unit function (ReLU) is a function that is zero for all negative inputs and the input itself for all positive inputs. It is useful for unbounded positive outputs such as prices. 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:

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

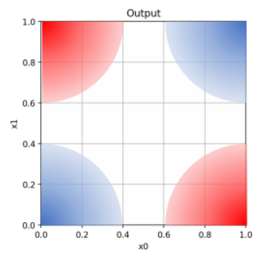
An interesting property of softmax is that if you add the same constant to every element in the logits, the output probabilities will remain exactly the same. Also, none of the probabilities will ever be exactly equal to 1 or 0 (although it does get rounded due to precision error in practice).

2. Logistic regression

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

A single neural unit with a sigmoid activation function can be used to create logic gates such as AND (output a 1 when both inputs are 1) and OR (output a 1 when at least one input is a 1). The given script trains a neural unit to perform an OR function. Change the training set to make the neural unit learn AND instead.

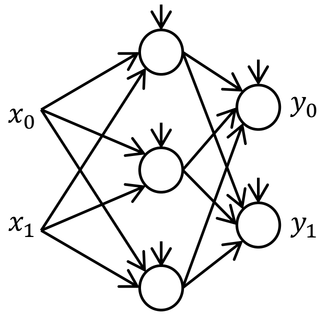
During training, a heat map is shown which displays how the output changes as the two inputs of the neural unit (horizontal and vertical axis) are changed between 0 and 1. The colour changes from red, when the output is 1, to blue, when the output is 0. Note how all the heat maps display a linear separation behaviour. It can be proven that a single sigmoid neural unit can only learn linearly separable classes, that is, whatever parameters you use for the neural unit, you can always draw a straight line on the output heat map which separates the reds from the blues. Can you learn an XOR function (output 1 for unequal inputs and 0 for equal inputs) with this setup? In other words, if the colours had to be setup as shown below, can you draw a single line that has all the blues on one side and all the reds on the other side?



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 neural unit can only learn a linearly separated input space. Two layers worth of neural units in series, where the output of the first group of neural units acts as input to the second group of units, 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. This is what a two layer network looks like:



The first layer is called a hidden layer, because it is only meant for internal use. The second layer is called an output layer. The equation of a two layer neural network is as follows:

$$\boldsymbol{y} = f^o\left(\boldsymbol{b^o} + f^h\left(\boldsymbol{b^h} + \boldsymbol{x}\boldsymbol{W^h}\right)\boldsymbol{W^o}\right)$$

where f^o, $\boldsymbol{b^o}$, and $\boldsymbol{W^o}$ are the activation function, bias vector, and weight matrix of the hidden layer respectively whilst f^h, $\boldsymbol{b^h}$, and $\boldsymbol{W^h}$ are the activation function, bias vector, and weight matrix of the output layer respectively.

You can continue adding more hidden 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 a non-linear activation function.

It is possible to leave out the non-linear activation function for the output layer, but not for the hidden layer. The reason for this is that two linear layers connected in series are equivalent to a single layer. Here's what happens if you leave out the non-linear activation function:

$$\boldsymbol{y} = \boldsymbol{b^o} + \left(\boldsymbol{b^h} + \boldsymbol{x}\boldsymbol{W^h}\right)\boldsymbol{W^o}$$

$$\boldsymbol{y} = \boldsymbol{b^o} + \boldsymbol{b^h}\boldsymbol{W^o} + \boldsymbol{x}\boldsymbol{W^h}\boldsymbol{W^o}$$

$$\boldsymbol{y} = \left(\boldsymbol{b^o} + \boldsymbol{b^h}\boldsymbol{W^o}\right) + \boldsymbol{x}\left(\boldsymbol{W^h}\boldsymbol{W^o}\right)$$

The first term results into a new bias vector of the same size as the output bias vector whilst the second term is the input vector multiplied by two matrices multiplied together which results into the input vector multiplied by a new weight matrix of the same size as the output weight matrix. The non-linear activation function is when prevents this from happening as it keeps the equation from being simplified and keeps the two layer network more expressive than the single layer one.

Although two layers are enough to learn any function, in practice the number of hidden neural units needed to approximate complex functions could be impractically large. It turns out that with three layers you can also approximate any function but with exponentially fewer parameters. This means that adding more layers makes the neural network's parameters more efficient as they can learn more complex functions with less 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 given script 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 units. 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.

See what the output heat map looks like when you use 3 hidden neural units and the following training set:

```
train_x = [ [0.0,0.0],[0.0,0.3],[0.0,0.7],[0.0,1.0],
[0.3,0.0],[0.3,0.3],[0.3,0.7],[0.3,1.0],
[0.7,0.0],[0.7,0.3],[0.7,0.7],[0.7,1.0],
[1.0,0.0],[1.0,0.3],[1.0,0.7],[1.0,1.0] ]
train_y = [    [0],      [0],      [0],      [0],       [0],      [1],
[1],      [0],       [0],      [1],      [1],      [0],       [0],      [0],
[0],      [0] ]
```

4. Dropout

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

In the previous topic was saw weight decay as a way to regularise polynomials. Neural networks have an interesting way to be regularised called dropout. It consists of randomly selecting neural units in a hidden layer or input and dropping it from the neural network by replacing its activation value with a zero. A probability is used to control how likely each unit is to be selected. This process is repeated (independently 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 since it will not contribute anything in the weighted sum that is used to calculate the next layer's activation.

Dropout 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 (think of complex vs simple polynomials). With dropout, since no neural unit in a layer would be able to 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 (with some shared parameters). 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 average output. In machine learning, this is called ensemble learning and it tends to improve performance.

- It's not just any ensemble learning that is being performed, it's ensembling of different neural networks that have shared parameters. Also, each dropped version of the neural 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 a fraction of the 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 doubling the activation values of the non-dropped neural units 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.

In the given script, notice how noisy the error graph is. This is due to the randomness introduced by dropout.

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 of a neural network uses a sigmoid activation function, then if you look at the shape of the graph of the square error, you will see that it is mostly flat with a little curve near 0. This means that when the output is wrong, the gradient of the error will be close to zero, making gradient descent work very slowly. There is another error function which does not suffer from flat gradients called the binary cross entropy function. It basically works by taking the logarithm of the output. 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 added together, one for when the target is 1 and one for when the target is 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-error functions.

The given script plots this error function side-by-side with the square error to see where the flat and steep regions of the error functions are.

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 cross entropy 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 errors. To avoid this problem we can work with the logits of the sigmoid (the numbers fed to sigmoid) 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. Note that the cross entropy function (which might not be binary) is abbreviated to XE.

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 can also be used on each individual element of a softmax output, but it wouldn't be as efficient as a special cross entropy function for softmax called categorical cross entropy. In this case, rather than the target value being what you want the output to be, the target value is instead the index of the softmax element that you want to have a value of 1 (with everything else being 0). 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.

The given script trains a neural network to decode binary numbers into one hot vectors, that is, a vector of all 0s except one 1. 00 maps to 0001, 01 to 0010, 10 to 0100, and 11 to 1000.

8. Extra theory

- A neural network is just a normal algebraic function that can be written as a single 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 as discussed in the previous topic. 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 point in a function 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 all local minima are generally 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 allows 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. It will just progress at a slower pace than it would move otherwise. This StackOverflow answer shows how to apply it in Tensorflow.