Neural Networks

Alex Sverdlov
alex@theparticle.com

1 Connectionism

In the early days of Artificial Intelligence, researchers got the idea that the shortest path to 'intelligence' was to copy something that appears to already have it. That is, to mimic the functionality of the brain.

From brain tissue observations, it appears the brain is composed of a network of cells, called neurons. Each neuron is connected to many (often thousands) of other neurons, and occasionally 'fires' (releases a burst of electrically charged molecules, which other connected neurons can detect). It is assumed that the neuron firing is mostly determined by a function of the connected neurons (though some neurons may fire due to other factors, such as stimulus from the environment).

Thus, a simplistic computational model emerged. Each artificial neuron computes a linear combination of its inputs (big assumption here), followed by a threshold function. Essentially a weighted sum, followed by a decision on whether to fire or not.

Such artificial neurons can then be arranged in a network to enable function composition (output of neurons used as input to other neurons). While many network architectures are possible, a very common one is arranging neurons in layers, and then composing the layers (output of one layer feeding as input to next layer).

Many of these simplifications may not be biologically sound, but there is hope they capture some basic essence of what goes on inside the brain.

1.1 Perceptrons

The first artificial neurons were perceptrons.

Perceptrons are linear discriminators, meaning that they use a linear function, such as a line, to split the input domain into two classes, the 'in front' class, and 'in back' class. One can easily visualize this by drawing a line on a piece of paper, and noting that the line splits the paper in two halves. Inputs on the same side of the splitting line have the same class. When we train a perceptron using labeled data, we are in effect learning where this splitting line should be.

Perceptrons generally have two parts, the summing part, and the threshold part. The summing part simply performs the inner product of inputs and outputs, while the threshold part applies the *step* function. The perceptron function is essentially:

$$f_{\boldsymbol{w},b}(\boldsymbol{x}) = \mathit{step}(\boldsymbol{w}^T\boldsymbol{x} + b)$$

where both $\boldsymbol{w}$ and $\boldsymbol{x}$ are column vectors, $\boldsymbol{w}$ are the weights by which inputs get scaled, $\boldsymbol{w}$ are the inputs, and b is the 'bias' (the constant term in the linear equation). The *step* function is simply:

$$\mathit{step}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

The general rule to train a perceptron using labeled training data is:

$$w_i = w_i + \eta(t - o)x_i$$

Where t is the target label, x is the input, and η is the learning rate, often set to something small, like 0.01.

The meaning of o depends on the learning method. In *perceptron learning*, o is the final output of the perceptron (after applying the $\mathit{step}(x)$ function). The values of o are either 0 or 1. Notice that w_i will increase when $t = 1$ and $o = 0$, decrease when $t = 0$ and $o = 1$, and stay unchanged when $t = o$.

In *gradient descent*, o is the pre-threshold value ($\boldsymbol{w}^T\boldsymbol{x} + b$), or the inner product of input and weights.

Notice that with perceptron learning, the learner only knows how many errors it makes for the training data (each instance is either right or wrong), while in gradient descent, the learner has a good idea of how well the model fits the training data (not just the number of errors). Gradient descent can use this extra information to improve the fit between the model and the data—in fact, while perceptron learning algorithm will not converge on non-linearly separable training data, gradient descent will still manage to find weights that provide the best fit (with the least number of errors).

1.2 Artificial Neurons

The main problem with perceptrons is that they lack a derivative: the derivative of $\mathit{step}(x)$ is 0 everywhere, except when $x = 0$. To see why this is important, lets back-track a bit to the loss function.

To objectively measure performance, we need to define a fitness function, also known as a loss function (or a cost function, etc.). One function that is often used is the sum of squares:

$$L(D, f) = \frac{1}{2} \sum_{(\boldsymbol{x},t)\in D} (t - f(\boldsymbol{x}))^2$$

Given a labeled dataset D consisting of instances of the form $(\boldsymbol{x}, t)$, where $\boldsymbol{x}$ is the input, and t is the desired label, and a function f, the loss function L returns a number indicating the amount of disagreement between the calculated (via f) and expected results. For a perceptron, the L function would return number of errors that f makes on the dataset D. The $\frac{1}{2}$ multiple is there for convenience, but doesn't change the spirit of the output.

The derivative of L:

$$\frac{\partial L}{\partial f} = \sum_{(\boldsymbol{x},t)\in D} f(\boldsymbol{x}) - t$$

or for any given training instance:

$$f(\boldsymbol{x}) - t$$

The function f often has the form of

$$f_{\boldsymbol{w}}(\boldsymbol{x}) = thresh(\boldsymbol{w}^T\boldsymbol{x}) = y$$

If we know that derivative of *thresh* is *dthresh*, we can write:

$$\frac{\partial f}{\partial thresh} = dthresh(\boldsymbol{w}^T\boldsymbol{x})$$

and

$$\frac{\partial thresh}{\partial w_i} = x_i$$

Combining all of the above, we have:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial thresh}\frac{\partial thresh}{\partial w_i}$$

This is the gradient of the loss function. If we move w_i in the gradient direction, it will increase the loss function. If we want to decrease the loss function, we need to move w_i in the negative of that direction. The weight update rule is:

$$w_i = w_i - \lambda\frac{\partial L}{\partial w_i}$$

where λ is the learning rate, and is often a small constant, such as 0.01.

Notice that if *thresh* function is *step* (which has no derivative), the weight update rule does not work (will always adjust by 0).

Artificial neurons use a variety of threshold functions, all of them with derivatives. The few common ones are: *identity* (no threshold at all), *sigmoid*, or more recently the *ReLU* function:

$$identity(x) = x \qquad sigmoid(x) = \frac{1}{1+e^{-x}} \qquad ReLU(x) = max(0,x)$$

Notice that if *identity* function is used, the weight update rule is identical to the perceptron gradient descent mentioned above, e.g.:

$$w_i = w_i + \eta(t-o)x_i \qquad\qquad w_i = w_i - \lambda(o-t)x_i$$

1.3 Limitations of Neurons

There are certain problems that artificial neurons (a single layer of them) are incapable of dealing with, such as approximating the *XOR* function. Imagine inputs with labels: $((0,0),0)$, $((0,1),1)$, $((1,0),1)$, and $((1,1),0)$, and try to separate them using a single straight line. No such line exists, implying the need for a different approach, such as composing multiple layers, forming artificial neural networks.

1.4 Neural Networks & Backpropagation

The next logical step is to use multiple layers of artificial neurons. This is essentially function composition, where the output of the network is:

$$f(\boldsymbol{x}) = L_N(L_{N-1}(L_{\dots}(L_1(\boldsymbol{x}))))$$

where L_1 is the first layer function, output going into L_2, subsequently going into L_3, and so on, until the last output layer L_N.

Such neural networks are often known as "feed forward", as they do not provide a mechanism for later layers to send information back into the previous layers. It is not hard to imagine more complicated arrangements of neurons; the main problem is how to adjust weights to make the network output desired results.

Let us imagine a 2-layer network with just two weights, w_1, and w_2, and *ReLU* threshold. In this example, $L_1(x)$ computes $ReLU(w_1 x)$, etc., The entire computation of the network (with temp variables added) is:

$$\begin{aligned} a &= w_1 * x \\ b &= ReLU(a) \\ c &= w_2 * b \\ y &= ReLU(c) \end{aligned}$$

We start with the y and work our way backwards applying chain-rule:

$$\begin{aligned} \frac{\partial L}{\partial y} &= y - t \\ \frac{\partial y}{\partial c} &= step(c) \\ \frac{\partial c}{\partial w_2} &= b \\ \frac{\partial c}{\partial b} &= w_2 \\ \frac{\partial b}{\partial a} &= step(a) \\ \frac{\partial a}{\partial w_1} &= x \end{aligned}$$

So now we have a way to adjusting w_1 and w_2. We can do this for as many layers as we have. The $\frac{\partial c}{\partial b}$ is generally known as the "error" that we are back-propagating. It serves the same function as the $\frac{\partial L}{\partial y}$ in the last layer.

In general, backpropagation tends to do very badly on non-trivial problems (such as networks with more than 2 layers), and requires quite a bit of tweaking. Somewhat paradoxically, it also tends to do badly on many trivial problems—simple functions with few dimensions.

The reason is that with low dimensional inputs, there is a high chance of quickly getting stuck in a bad local minima, while with high dimensional inputs, there is a high chance of getting out of local minima via some downward leading dimension.

Some of the improvements involve adding momentum and/or adding regularization. Momentum [**?**] considers previous weight update as part of current weight update—allowing gradient descent to roll over small bumps (local minima).

Regularization [**?**] addresses a problem often associated with learning weights: the learning rule encourages weights to get outrageously huge; over fitting the training set. This can often be avoided by starting with very small weights, and quitting before over-fitting occurs. The regularization technique essentially adds a weight decay value, so with every iteration, weights tend to get smaller—even while the learning rule is trying to make them bigger. One of the problems with regularization is that the weights become 'stuck' near the origin.

Recent progress on neural networks has been by switching to *ReLU* function (away from *sigmoid*). The backpropagated gradient of *sigmoid* activation tends to 0 the deeper into the network it goes—*ReLU* seems immune to that particular problem.

Another way of making practical use of backpropagation is to do unsupervised pre-training on the network prior to applying backpropagation. This hopefully moves the network into the right cost-function valley, from where backpropagation can find the local minimum. Such unsupervised pre-training is often done via auto-encoders or via Restricted Boltzmann machines.

1.5 Recurrent Networks

Recurrent networks are networks that are not strictly layered; more of a general graph structure. These can have outputs going back as inputs, etc. Such networks are considered to be more representative of how the brain might work [**?**], as they can have memory, and do timing tasks [**?**].

The most common way to organize RNNs is to do a linear combination of the current output and previous output.

Backpropagation can be applied in recurrent networks, except instead of layers, we might imagine a predecessor concept—where we backpropagate errors to the predecessor neurons, etc. Obviously these things can fall into infinite loops—so normally one would backpropagate only N "hops" back, etc.

1.6 Hopfield Networks

A Hopfield network is a form of recurrent artificial neural network that can function as content addressable associative memory [**?**, **?**].

Essentially it's a network where correlation in activity corresponds with weight between different neurons. Imagine stimulus m causing neuron m to be activated, and stimulus n causing neuron n to be activated. If both stimulus m and n appear in the environment together, the network will learn the activity correlation. If at a later time the network is presented stimulus m, it can recall n.

1.7 Autoencoders

The idea behind *autoencoders* is to train the network of the form pictured in Figure 4.

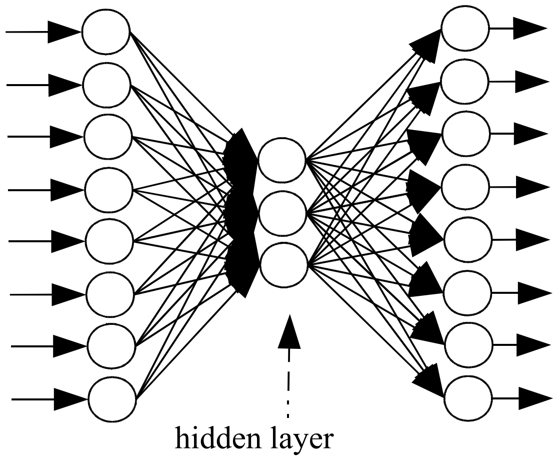

Figure 1: Learning the hidden layer finds a new compact representation of the data.

If we feed this network samples such as: 10000000, 01000000, 00100000, 00010000, 00001000, 00000100, 00000010, 00000001, and use input itself as the target, something remarkable happens: the network learns to equate the input with a binary encoding of the input, such as 001, 010, 011, 100, 101, 110, and 111. In other words, it finds a more compact representation of the data [?].

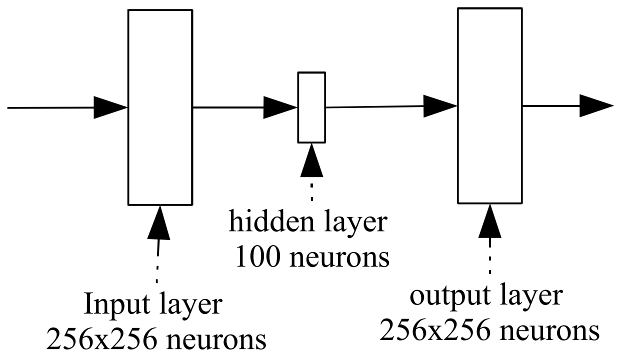

Figure 2: Learning the hidden layer finds the most important features of the input image.

Now consider a more extreme example in Figure 5. The input and output is a 256×256 image, and the goal is to learn the hidden layer. Once trained, the hidden layer will have a compact representation of an image—it will have the most important features of the input image. Trained on a collection of images, it will pick out the most important feature out of all of them [?, ?].

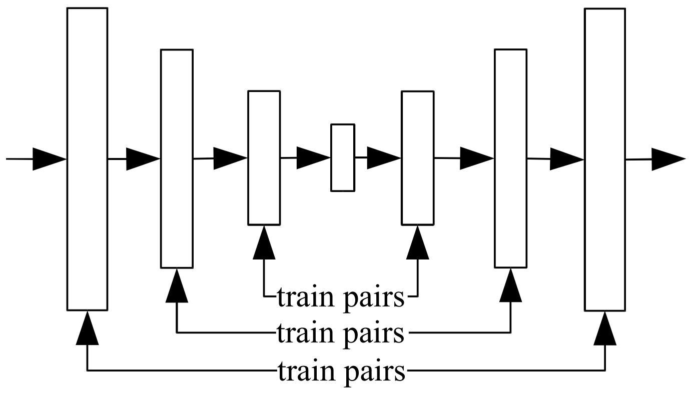

Figure 3: Autoencoders are trained in layers.

We can layer such architectures like onions, and train them in steps, as in Figure 6. First train the outer layer, then unroll, and train the inner layer using the inputs/outputs of the first layer. Then unroll again, etc. Such an approach allows us to build *deep neural networks*.

Applications for autoencoders (or deep belief nets, as they're sometimes referred to) include image processing and character recognition [?, ?, ?]. Essentially a deep network is trained to pull out the most relevant features from many sample patches of writing. In computer vision, autoencoders are used for object recognition [?, ?].

1.8 Convolutional Neural Network

One of the motivations for neural networks is that they mimic the way the brain works. Initially that was easily observed: humans can recognize hand-written characters, and neural networks can recognize hand-written characters.

But then some experiments were done where the input vector was randomly permuted. In other words, all input pixels were. To the human eye, it looked like noise, yet a neural network performed just as well as before. This illustrates that the way our brain works is very different than a neural network.

Our brain has locality. Pixels that are next to each other are processed somewhat together with other near by pixels. This is not the case in the dense layers of a neural network.

The other problem with neural networks was the number of weights. There is a weight between every input and every output. That is on the order of N^2 weights. If our input images are 1000×1000 pixels in size, then a single layer would have 1000000^2 weights. That is not practical.

To overcome the above problems, we can define convolution layers. These are small kernels that apply to the entire image, producing an output. And we can apply multiple kernels.

TODO; more in class.