Instructors: Parth Shah, Riju Pahwa

Lecture 2 Notes

Outline

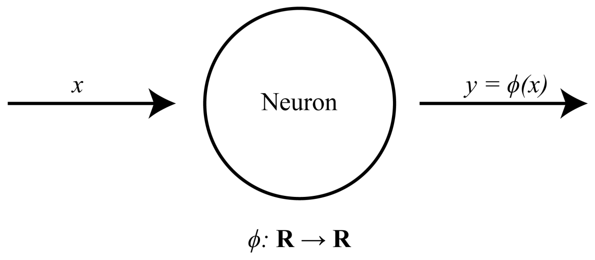
Neural Networks

The Big Idea

We have seen that often times the relationship between the data and labels is complex. For this reason linear classifiers are not enough. But what about multiple linear classifiers? How about a network of classifiers? How about a network of differentiable functions?

Architecture

This is all a neural network is. It is a solution to deal with high dimensional data. But to understand the neural network we have to understand the fundamental unit: the neuron. Note these neurons take an input $\mathbf{x}$ which is a vector of data and then output a real value y. This is similar to the way we defined classifier except here instead of a label it is a value. On the next page is a diagram of a neuron:



The function that transforms the input into the output is known as an **activation function**. The activation functions come in many different flavors. The most popular ones to know are:

Rectified Linear Unit

$$y = \max(0, x) \tag{1}$$

Note in reality there's a set of input values to each neuron which as stated earlier acts as an input vector. x is usually used as the sum of the entries in the vector.

Therefore the formula looks more like:

$$y = \max(0, \sum_i \mathbf{x}[i]) \tag{2}$$

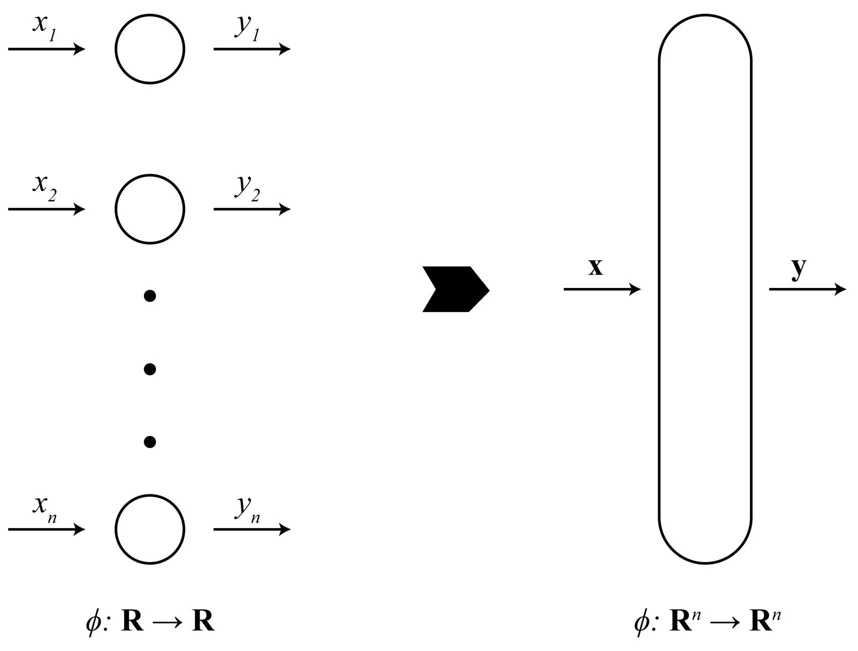
Sigmoid

$$y = \frac{1}{1 + e^{-x}} \tag{3}$$

Tanh

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4}$$

These neurons are then organized into layers. The idea is that at each step the data is passed through some set of neurons which will then pass their results to another set of neurons. These sets are layers. In neural networks there are three types of layers: input layers, hidden layers, and output layers. Every network has one input and one output layer. All layers in between are referred to as hidden layers.

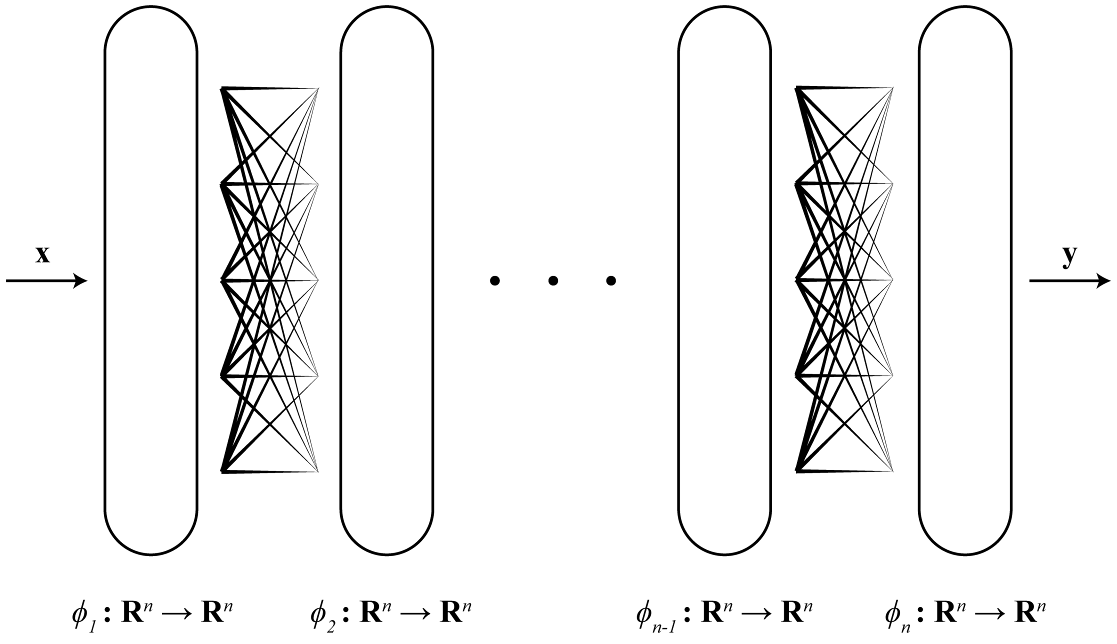

x_1
y_1
x_2
y_2
x_n
y_n
$\phi: \mathbf{R} \to \mathbf{R}$
$\mathbf{x}$
$\mathbf{y}$
$\phi: \mathbf{R}^n \to \mathbf{R}^n$

When designing the architecture we have some decisions to make. I will use the example of MNIST data to explain these decisions. MNIST data comes in the form of 28 by 28 pixel images which are all images of digits. So the input layer is obvious. 28 by 28 implies 784 input neurons.

The output neurons is an interesting question. We could have one output neuron but keep in mind the way our network learns is by propagating error back. So in this case if we output a 9 then its the digit 9. However this is not all to great of an approach because lets say our neural network thinks the image is a 9 with probability 0.5 and 0 with probability 0.5, with 1 output neuron there's no way of knowing this. Let's say it even decides that output somehow instead of outputting the average, training it is a problem. We should penalize the neural network more if the digit was a 9 and it thought it was a 9 with probability with 0.1 than with probability 0.4 (it might output the wrong answer in both cases). So a smarter output layer would be 10 neurons and basically train it to output a 1 if it thinks thats the digit and 0 if not. Turns out this has additional advantages. The MNIST data is special in that it gives you the 28 by 28 box for the digit. But lets say we are not given those boxes. I might want to be able to classify digits that are just written on a page. In this case we do not have boxes. So why does this decision turn out to be better. We can train it to output all 0s if the box does not align. If we only had one output neuron this would not work. Clearly architecture is very important. The key is thinking about how the output layer helps for all tasks and what output layer makes it easier to understand how off the network is (is it easy to define the error). In the MNIST case one output neuron makes it hard to define error. But having an ideally binary output for each digit does.

The hidden layers are more up to you. The more neurons and more layers in your hidden layers the higher dimension of relationships your network can find in the data.

Below is a diagram of the entire network:

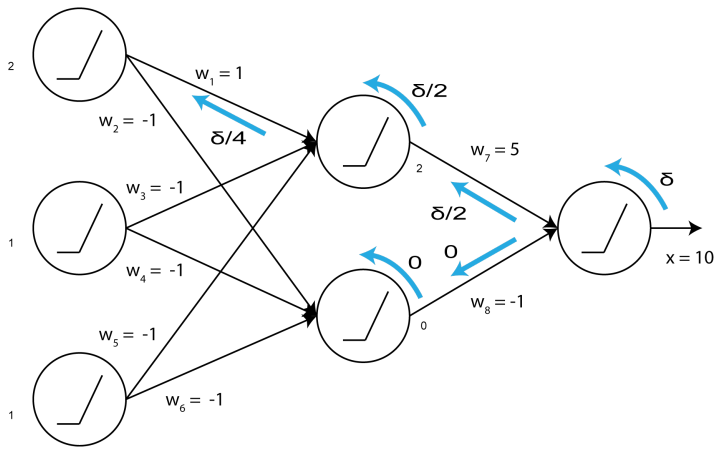

x
y
$\phi_1 : \mathbf{R}^n \rightarrow \mathbf{R}^n$
$\phi_2 : \mathbf{R}^n \rightarrow \mathbf{R}^n$
$\phi_{n-1} : \mathbf{R}^n \rightarrow \mathbf{R}^n$
$\phi_n : \mathbf{R}^n \rightarrow \mathbf{R}^n$

Note the connections between the layers. Each edge has some weight. The weights are initialized randomly and the goal is to learn the weights using some optimization method. This leaves the final section on feed forward neural networks: SGD and backpropagation.

SGD and Backpropagation

We learned about stochastic gradient descent when we covered the passive aggressive algorithm. It is one of the most powerful optimization tools. Although it might not seem we can use it in Neural networks it turns out we can. The idea is to use the chain rule to pass back the derivative. This way we can update all our weights.

Here is how it works:



Convolutional Neural Networks

Intuition

The idea behind convolutional neural networks does not stray too far from feed forward neural networks. The only idea is that the layers are not two dimensional (images). Rather they are three dimensional. What does that mean? Well the idea is to take the image input and try to extract information.

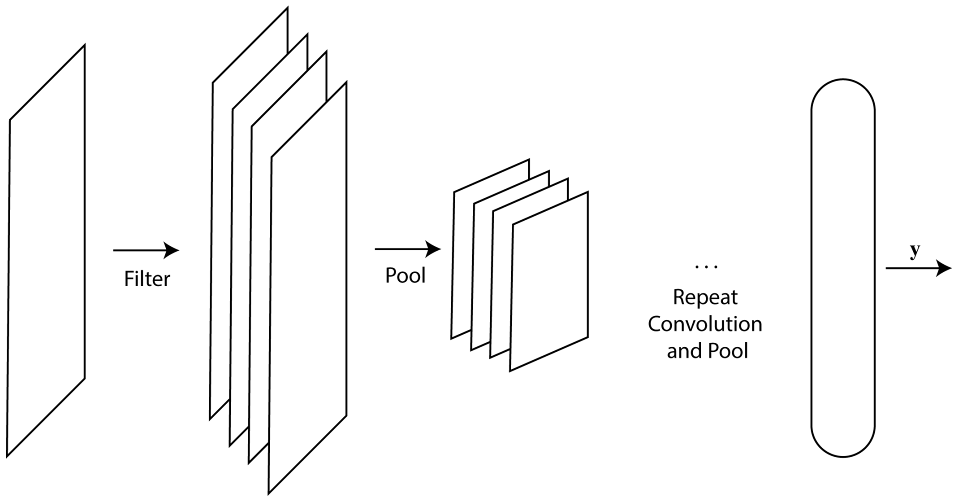
Architecture

The architecture does not differ much from a feedforward neural network. The only difference is that instead of regular layers and connections between layers. The initial layers alternate between filtering and pooling (reducing the size of the layer). The pooling is trivial but the filtering is where the power of the CNNs comes in.

The example I will use to explain filtering is finding edges. To find the edges in an image we could apply a convolution operation. Lets say we take the RGB values of the image a convolution is like applying a small square as an operation. For example a 3×3 matrix is multiplied to each smaller 3×3 square in the original image. So the function leaves us with a $(n-2) \times (n-2)$ matrix of values (if the original data is $n \times n$). So for example to find edges we may want a matrix [[-1 -1 -1][-1 8 -1][-1 -1 -1]]. So if the pixels are similar over the 3 by 3 square then the value comes out to be small. However, only on edges does this take values away from 0.

These convolution matrices are called filters. For all purposes, know that the power of CNNs is in these filters. This is why convolutional neural networks are so good for image classification. The goal of each filter is to extract information from the original image. The rest is just reducing the dimension of the information on each layer and proceeding just as a feed forward neural network.

Below is an image of CNNs:



Recurrent Neural Networks

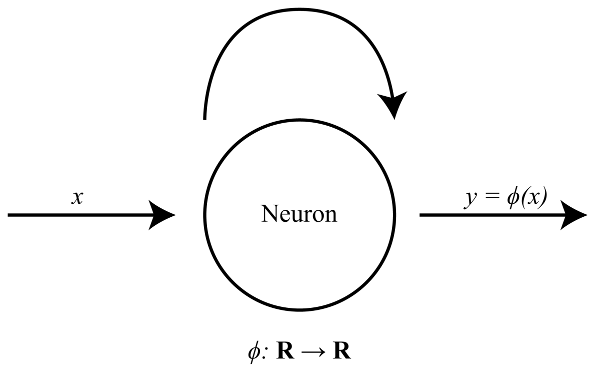
Intuition

As Convolutional Neural Networks are powerful in the field of computer vision for understanding images and video data, Recurrent Neural Networks are particularly powerful for language processing. Why?

To understand this we need to think a little bit about how we process that data. When we see an image we process it all at once. We extract objects and use that to understand what we see. However, when we read sentences or books, we have an internal state. What we read next is affected by what we have read before. Basically we process language with state. However, our neural networks have no concept of state. The data is processed one at a time, so the next data point is not affected by previous data points. This is where we introduce recurrence.

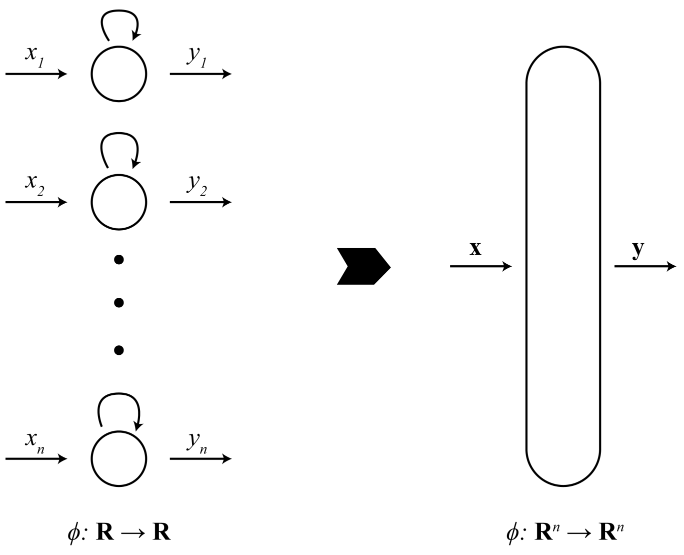
Architecture

The architecture is very similar to the feed forward neural network bar one difference in the neurons. An image is the best way to see the difference. This is the recurrent neuron:



The arrow passed back in gives the state. And this affects future outputs which are passed back in. Therefore this network has memory.

Here is how layers look in recurrent networks:


x_1
y_1
x_2
y_2
x_n
y_n
ϕ: $\mathbf{R} \to \mathbf{R}$
$\mathbf{x}$
$\mathbf{y}$
ϕ: $\mathbf{R}^n \to \mathbf{R}^n$

We will avoid rigorous discussion of the mathematics recurrent neural networks as it is quite in depth. But what we should notice is that running gradient descent is not quite so straightforward. We cannot use chain rule because the value passes into itself. Furthermore, we have this problem of explosion and vanishing. If the value out of a neuron is high the value passed in is high and it becomes self-perpetuating.
