LECTURE 17
Neural Networks

Artificial neural networks have been an important component of machine learning and computational neuroscience for many years before they exploded in the last 5-10 years as an automated tool for prediction modeling, specifically classification and regression. The key modern workhorse are deep convolutional neural networks. In this lecture our goal is to understand the basic concepts underlying deep convolutional networks and the ideas that led to deep convolutional networks. The main idea behind deep neural networks is that the learning or inference problem is framed as learning a function or map $f : X \to Y$ where X are the inputs and Y are the outputs or our predictions.

17.1. Earlier models

The ideas that gave rise to deep convolutional networks that we will cover are the perceptron algorithm, the idea of backpropogation, and universal approximators. The perceptron is the most primitive version of a neural network that still does something useful. Backpropogation is the algorithm that is the conceptual basis for training modern neural networks. The idea of a universal approximator is one of the main arguments for while deep neural networks are successful.

17.1.1. The perceptron algorithm

The perceptron algorithm was proposed by Rosenblatt as a pattern recognition machine. The idea was to take a p-dimensional input $(x_1, ..., x_p)$ vector as input and output one of two classes $\{0, 1\}$. Sometimes the input vector is appended with a 1 so the input is $(1, x_1, ..., x_p)$. The perceptron performs three operations:
1) each input element is multiplied by a weight: $\{u_1 = w_1 \times 1, u_2 = w_2 \times x_1, u_3 = w_3 \times x_2, ...\}$
2) the products are then added $h = \sum_{j=1}^{p+1} u_j$
3) an activation function is applied to the sum, the specific activation function was the unit step function

$$y = \begin{cases} 0 \text{ if } h \leq 0, \\ 1 \text{ o.w.} \end{cases}$$

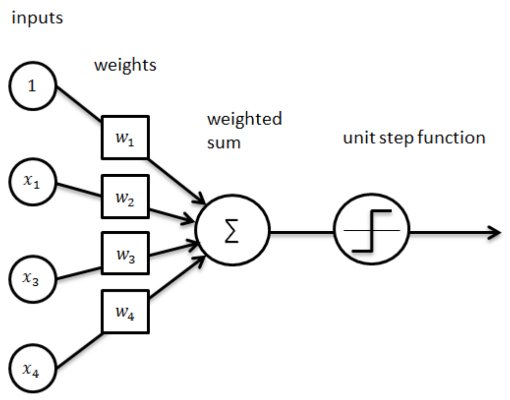

Figure 1. Network architecture for the perceptron.

The perceptron came with the following training algorithm which given n obeservations $(x_i, y_i)_{i=1}^n$ outputs a set of weights for which the perceptron would correctly classifies all of the inputs $(x_i)_{i=1}^n$:

```
1) Initialize weights $w$ randomly
2) While any input is incorrectly classified do:
    i) Select an input-output pair $k \in \{1, ..., n\}$
    ii) If $y_k = 1$ and $w^T x_k < 0$ then
            $w = w + x$
        Else if $y_k = 0$ and $w^T x_k > 0$ then
            $w = w - x$
```

The idea above is that the perceptron algorithm only updates the weights when an error is made on the observations. If the observations can be perfectly classified by a linear function then an upper bound on the number of steps that the perceptron algorithm will take to correctly classify all the points is

$$O\left(\frac{R^2}{\gamma^2}\right)$$

where R is the largest norm of any observation $R = \max_{i=1,...,n}\{\|x_i\|\}$ and γ is the margin satisfied (recall the SVM) or the assumption that there exists a γ such that for some weights w all points will be classified correctly with margin γ or for all $i = 1, ...n$

$$y_i w^T x_i > \gamma.$$

People were excited by the perceptron and it was used in classification. The Minsky and Papert put a kibosh on the perceptron via what was called the x-or (exclusive or problem). Consider the inputs as bi-variate boolean values (and we don't append the 1) and our function $f : X \to Y$ is the exclusive or function

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

The problem with the x-or case is that the perceptron will spin for ever, it can never separate the positive from negative instances.

The way around this was the multi-layer perceptron which could be thought of as nesting perceptrons.

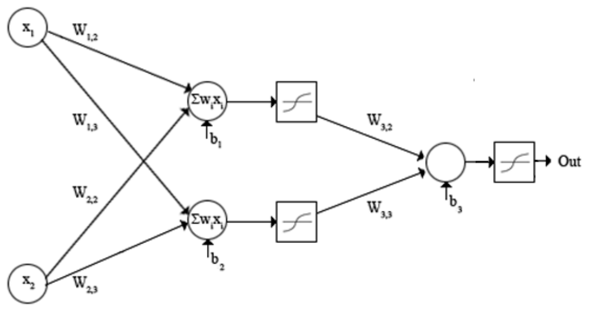

Figure 2. Network architecture for a multi-layer perceptron.

So given a p dimensional input (again appended with a 1) and two hidden layers the multi-perceptron architecture would be

$$y_i = \mathrm{U}\left(w_2^T \left[\mathrm{U}(w_{11}^T x_i)\ \mathrm{U}(w_{12}^T x_i)\right]\right).$$

Here $h_1 = \mathrm{U}(w_{11}^T x_i)$ is the output of the first hidden layer and $h_2 = \mathrm{U}(w_{12}^T x_i)$ hidden layer (both are perceptron operations) then we concatenate the two scalars $h = [h_1\, h_2]$ into a vector and apply the perceptron operations to the vector h.

One can be creative in terms of the number of hidden units and the amount of nesting, the more nesting the deeper the network. The choices of the amount of nesting and the number of hidden units are called the neural network architecture. Once an architecture is specified the weights in each layer need to be learned. The perceptron algorithm cannot be used here so a novel method was needed.

17.1.2. Backpropogation

The idea behind the backpropogation algorithm was to use derivatives and the chain rule to optimize the weights or update weights. The multi-layer perceptron is a feedforward architecture which we consider as a composition of functions. Consider a network with depth d (this is the amount of nesting) and at each layer j we have τ_j hidden units, we can think of the ultimate function $f : X \to Y$ as a composition of functions $f_d \circ f_{d-1} \circ \cdots \circ f_1$ where each function $f_i : \mathbb{R}^{\tau_i} \to \mathbb{R}^{\tau_{i+1}}$ upto the last layer which is $f_d : \mathbb{R}^{\tau_d} \to \mathbb{R}$. At each layer j we also have the weight parameters w_j which are the weights for all the hidden units. So we can think of indexing the function at each layer as f_{w_i} and the multi-layer perceptron can be written as

$$y_i = f_{w_d}(f_{w_{d-1}}(f_{w_{d-2}}(\cdots f_{w_1}(x_i)))) = f_{w_d} \circ f_{w_{d-1}} \circ \cdots \circ f_{w_1}(x_i).$$

We can further refine our notation for the weights at a layer k as $w_k = \{w_{k,1}, ..., w_{k,\tau_k}\}$ where $w_{k,1}$ are the weights for the first hidden unit in layer k and w_{k,τ_k} are the weights for the last hidden unit in layer k. Given this feedforward architecture we can now consider solving the following optimization problem to learn the weights

$$\min_{w_1,...,w_d} \left[L(w_1, ..., w_d) = \frac{1}{2} \sum_{i=1}^{n} (y_i - f(x_i))^2 \right],$$

where $f = f_{w_d} \circ f_{w_{d-1}} \circ \cdots \circ f_{w_1}(x_i)$. Now at any layer j one can update the weights for a hidden unit i based on derivatives with respect to $w_{j,i}$ just as we did for Newton-Raphson

$$w_{j,i}^{(t+1)} = w_{j,i}^{(t)} - \alpha \frac{\partial L}{\partial w_{j,i}}. \tag{17.1}$$

note the compositional structure of the feedforward neural network which suggests using the chain rule. The problem with the derivative operation above is that our activation functions are unit step functions which are non-differentiable. A solution to this is to replace the step functions with a sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}},$$

which is differentiable.

The idea behind the backpropogation algorithm is implement equation (17.1) recursively using the sigmoid as the activation function. The chain rule is the key idea behind backpropogation, so the chain rule for a composition is $f(g(x))' = f'(g(x))\, g'(x)$.

Before we write out the general formula for backpropogation we consider updating the weights at the final layer d. We also consider the case where there is only one observation y_i. We'll extend both these cases soon. Recall that we want to compute over all hidden states $k = 1, ..., \tau_d$ the partial derivative of L with respect to each $w_{d,k}$. We denote $f(x_i)$ as $f_d(x_i)$ (this makes explicit we are looking at the function value after layer d with input value x_i) so by the chain rule we have

$$\frac{\partial L}{\partial w_{d,k}} = \frac{\partial L}{\partial f_d(x_i)} \frac{\partial f_d(x_i)}{\partial w_{d,k}}$$

and $\frac{\partial L}{\partial f_d(x_i)} = (f_d(x_i) - y_i)$. Also we know that

$$f_d(x_i) = \sigma\left(\sum_k f_{d-1,k}(x_i)\, w_{d,k}\right) = \sigma(z_d(x_i))$$

where $w_{d,k}$ are the weights for the hidden units and $f_{d-1,k}(x_i)$ are the function values for x_i as an input for the k-th hidden unti after layer $d-1$. now we can write again using the chain rule

$$\frac{\partial f_d(x_i)}{\partial w_{d,k}} = \frac{\partial f_d(x_i)}{\partial z_d(x_i)} \frac{\partial z_d(x_i)}{\partial w_{d,k}},$$

and

$$\frac{\partial f_d(x_i)}{\partial z_d(x_i)} = \sigma(z_d(x_i))(1-\sigma(z_d(x_i))), \quad \frac{\partial z_d(x_i)}{\partial w_{d,k}} = f_{d-1,k}.$$

Now we put all the terms together with some notation:

1. the error signal: $\delta_e = f_d(x_i) - y_i$
2. the change with respect to the nonlinearity $\delta_d = \sigma(z_d(x_i))(1-\sigma(z_d(x_i)))$
3. the function value $f_{d-1,k}$ feeding into hidden unit k at layer d

so together

$$\frac{\partial L}{\partial w_{d,k}} = \delta_e \delta_d f_{d-1,k}.$$

We now compute the partial derivatives if we go back one layer further

$$\frac{\partial L}{\partial w_{d-1,k}} = \frac{\partial L}{\partial f_d(x_i)} \frac{\partial f_d(x_i)}{\partial w_{d-1,k}} = \frac{\partial L}{\partial f_d(x_i)} \frac{\partial f_d(x_i)}{\partial f_{d-1,k}(x_i)} \frac{\partial f_{d-1,k}(x_i)}{\partial w_{d-1,k}}.$$

We already have the computation for $\frac{\partial L}{\partial f_d(x_i)}$. The computation of $\frac{\partial f_{d-1,k}(x_i)}{\partial w_{d-1,k}}$ is identical to that of $\frac{\partial f_d(x_i)}{\partial w_{d,k}}$ and

$$\frac{\partial f_{d-1,k}(x_i)}{\partial w_{d-1,k}} = \sigma(z_{d-1,k}(x_i)(1-\sigma(z_{d-1,k}(x_i)))f_{d-2,k},$$

with $f_{d-2,k}$ is the function value from layer $d-2$ that feeds into hidden unit k at layer $d-1$

$$z_{d-1,k}(x_i) = \sum_{k' \sim k} f_{d-2,k'}(x_i)\, w_{d-1,k'},$$

as the weighted sum of all the function values at layer $d-2$ that feed into unit k at layer $d-1$ and the notation $k' \sim k$ simply means there is a feedforward edge between unit k at layer $d-1$ and unit k' at layer $d-1$. A similar computation as above gives us

$$\frac{\partial f_d(x_i)}{\partial f_{d-1,k}(x_i)} = \sigma(z_{d-1,k}(x_i)(1-\sigma(z_{d-1,k}(x_i)))w_{d,k}.$$

We can now recurse as many layers back as we want via

$$\begin{aligned}\frac{\partial L}{\partial w_{d-\tau,k}} &= \frac{\partial L}{\partial f_d(x_i)} \frac{\partial f_d(x_i)}{\partial f_{d-1,k}(x_i)} \frac{\partial f_{d-1,k}(x_i)}{\partial f_{d-2,k}(x_i)} \\ &\quad \cdots \frac{\partial f_{d-\tau+1,k}(x_i)}{\partial f_{d-\tau,k}(x_i)} \frac{\partial f_{d-\tau,k}(x_i)}{\partial w_{d-\tau,k}}.\end{aligned}$$

To extend to the case where we have more than one observation

$$\frac{\partial L}{\partial w_{d,k}} = \frac{\partial L}{\partial f_d} \frac{\partial f_d}{\partial w_{d,k}},$$

where $\frac{\partial L}{\partial f_d} = \sum_{i=1}^n (f_d(x_i) - y_i)$ and

$$\frac{\partial f_d}{\partial w_{d,k}} = \sum_{i=1}^n \sigma(z_d(x_i))(1-\sigma(z_d(x_i)))f_{d-1,k}.$$

The reason this algorithm is called backpropogation is that the chain rule is taking the error at the output and passing it back through the network recursively to update the weights.

17.1.2.1. *Choices of activation functions and architecture.* The big breakthrough in deep neural networks came in easy training of neural networks or software that implemented variants of backpropogation easily. So one could change the architecture of the neural network but not have to recode all the backpropogation updates.

Just as one can choose the type of network architecture one can choose the different activation functions. Typically the output activation function is some type of nonlinear function that will either output a probability $y \in [0, 1]$ or a continuous value between $[-1, 1]$ so the logistic function or hyperbolic tangent function are popular

$$\frac{1}{1+e^{-x}} \quad \frac{2}{1+e^{-2x}} - 1.$$

The hyperbolic tangent is often preferred as it does not saturate as easily as the logistic. Now in modern neural network architectures one does not use the above activation functions at the hidden units instead one uses a rectified linear unit (ReLU) or leaky ReLU

$$f(X) = \begin{cases} 0 \text{ if } x < 0, \\ x \text{ if } x \geq 0 \end{cases} \quad , \quad f(X) = \begin{cases} 0 \text{ if } \alpha x < 0, \ \alpha = .01 \\ x \text{ if } x \geq 0 \end{cases} \quad .$$

17.1.3. Universal approximator

Mathematical arguments for why neural networks work well often take the perspective of a neural network as a universal approximator. A a universal approximator is a function $f : \mathbb{R}^p \to \mathbb{R}$ that can approximate any continuous function on a compact subset of $\mathbb{R}^p$ arbitrarily well.

The idea of universal approximators goes back to Hilbert's 13-th conjecture that he presented in the Second International Congress of Mathematicians in 1900. The conjecture was that there exist continuous multivariable functions, which cannot be decomposed as the finite superposition of continuous functions of less variables. The specific example Hilbert gave was that there existed at least one such continuous function f of three variables, for example

$$f(x, y, z)^7 + xf(x, y, z)^3 + yf(x, y, z)^2 + zf(x, y, z) + 1 = 0,$$

which could not be decomposed as the finite superposition of continuous bivariate functions. This conjecture was proven false by Arnold and Kolmogorov in 1957, which meant that one can approximate arbitrary multivariate continuous functions as finite superpositions of univariate functions. The most cited version of an approximation result is that of Cybenko..

Theorem. *Consider the unit hypercube $I_m = [0, 1]^m$ and denote the set of real-valued continuous functions on I_m as $C(I_m)$. The function $\phi : \mathbb{R} \to \mathbb{R}$ is a nonconstant, bounded, and continuous function (the sigmoid). Then given any $\varepsilon > 0$ and any function $f \in C(I_m)$ there exists an integer N real constants v_i, b_i and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, .., N$ such that*

$$F(x) = \sum_{i=1}^{N} v_i \phi(w_i^T x + b_i)$$

and

$$\sup_{x \in I_m} |F(x) - f(x)| < \varepsilon,$$

or $F(x)$ is dense in $C(I_m)$.

17.1.3.1. *Width and depth.* The above theorem states that a two layer neural network with a sigmoid activation function can approximate any continuous function. The network we consider above is not deep, there is one layer of hidden units. So one question is how many units do we need or what should N be in the above theorem. One can also consider deep neural networks, ones where there are several layers of hidden units. A basic question in architecture of neural networks is should one consider shallow networks that are wide (N is very large) or deep networks that are not wide, this means one has many layers but the number of hidden units at each layer is small.

Although feed-forward networks with a single hidden layer are universal approximators, the width of such networks has to be exponentially large or N grows exponentially with m. Deep but not so wide networks tend to be more efficient approximators and results such as the following by Montufar et al., 2014 state that: The number of linear regions carved out by a deep rectifier network with m inputs, depth ℓ and n units per hidden layer is

$$O\left(\binom{n}{p}^{p(\ell-1)} n^p\right).$$

The importance of the above result is that the approximation power of the network increases exponentially with the depth ℓ of the network.

17.1.4. Convolutional neural networks