Going Deep: Neural Networks, Part I
Statistics 650/750
Week 14 Tuesday

Christopher Genovese and Alex Reinhart

27 Nov 2018

Announcements

- Out of town as of Wednesday afternoon, office Hours Wednesday
- Notes on line in `documents` repository `weekET.*`.
- Plan
  - Today: Artificial Neural Networks and Back Propagation
  - Thursday: Neural Net Activity
  - Tuesday: What's Deep about Deep Learning?

Neural Nets: Biological and Statistical Motivation

Cognitive psychologists, neuroscientists, and others trying to understand complex information processing algorithms have long been inspired by the human brain.

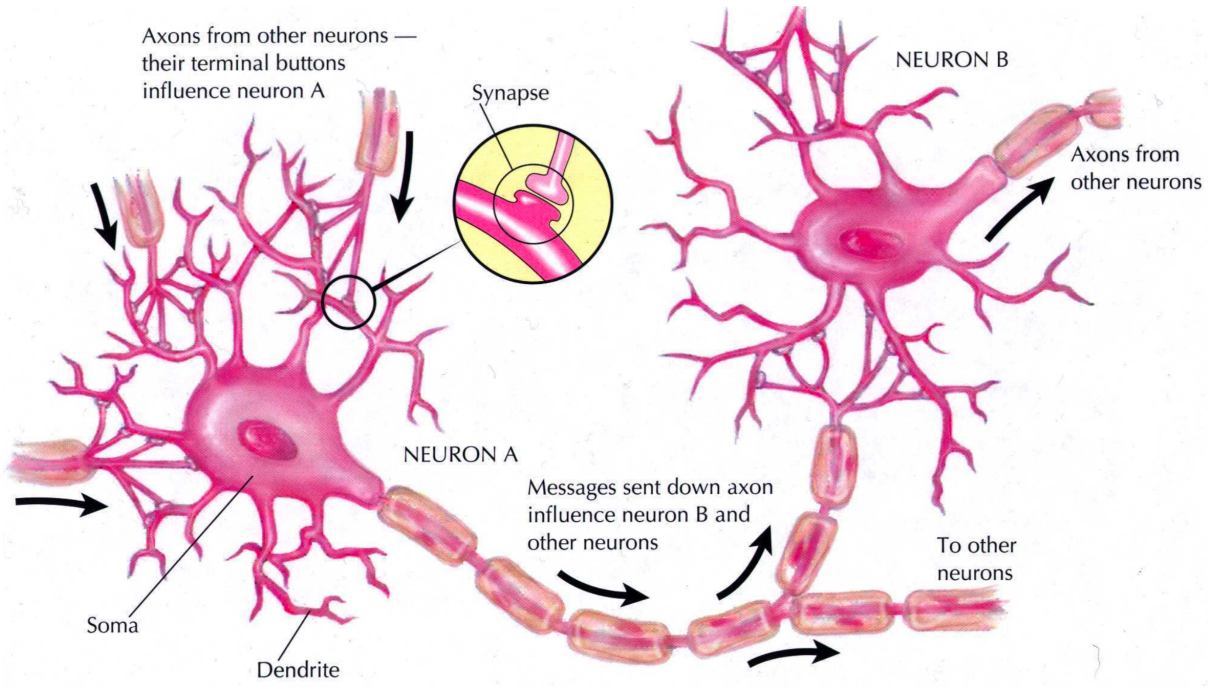


Image Credit: David Plaut

The (real) picture is, not surprisingly, complicated. But a few key themes emerge:

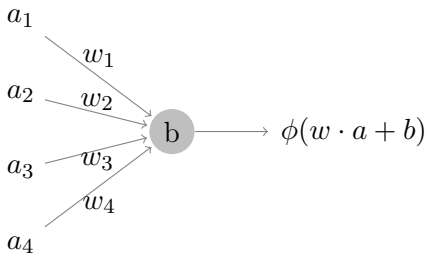
- The firing of neurons produces an **activation** that flows through a network of neurons.
- Each neuron receives input from **multiple other neurons** and contributes output in turn to *multiple other neurons*.
- Local structures in the network can embody **representations** of particular semantic information, though information can be (and is) distributed throughout the network.
- The network can **learn** by changing the connections (and the strength of connections) among neurons.

Artificial Neural Networks

An **Artificial Neural Network** is a computational model comprised of interconnected nodes that are inspired by biological neurons.

Artificial Neurons

Attempts to capture some of the apparent power of biological neural networks in the brain led to various idealized, computational models of neurons. The most common models used today look like the following:



Here, the inputs to this neuron – the a_i's – are the outputs from other neurons and are called **activations**.

The w_i's on the edges describe te strength of these connections and are called the **weights**.

The output of the network is determined in three steps: (i) take a weighted sum of the activations $w \cdot a$, (ii) shift it by a node-parameter b, called a **bias**, and (iii) pass the result through a function ϕ, called the **activation function** (aka *transfer function*).

There are several different types of nodes, characterized by the form of the activation function ϕ.

- Linear $\phi(z) = z$
- Perceptron/Heaviside $\phi(z) = 1_{(0,\infty)}(z)$.
- Sigmoidal $\phi(z)$ is a sigmoidal (S-shaped) function, that is smooth and bounded.

  A common choice is the *logistic function*

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

  but the Gaussian cdf and a shifted and scaled version of tanh are also used.
- Rectified Linear Unit (aka ReLU) $\phi(z) = \max(z, 0)$.
- Bias nodes that always output 1
- Pass through nodes with $\phi(z) = z$

- ...

Different kinds of nodes can be used and combined in any network.
Keep in mind that this *neuron* language is only a metaphor.

Networks

We now combine these units into a computational network. This can be done in various ways, but today we will look at the most basic, a **multi-layer feed-forward neural network** (MLFFNN).

These networks are designed in *layers*, each comprised of one or more artificial neurons, or nodes.

- The **input layer** takes in the data and usually consists of pass-through or purely linear nodes.
- The **output layer** converts the computation into a form needed in how the network is used.

  For two-class classification problems, for example, a single binary output node does the trick.

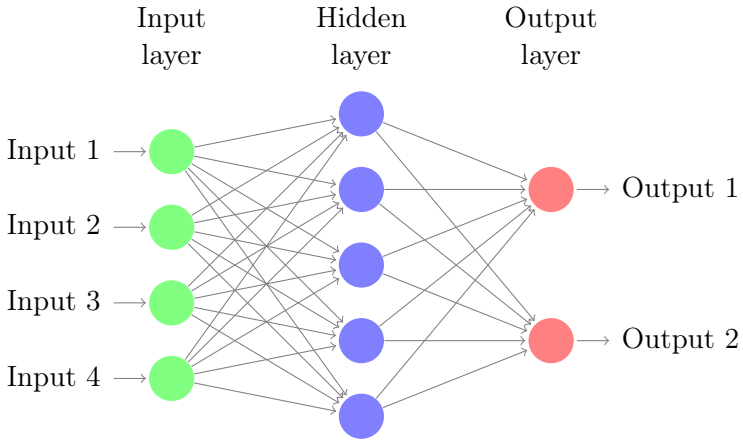
  Another common example is to produce a probability distribution via the **softmax** combination. This takes the outputs $z_1, \ldots, z_p$ of several output nodes and transforms the j^{th} output to

$$\frac{e^{z_j}}{\sum_k e^{z_k}}.$$

- In between input and output are one or more **hidden layers**. A network with more than one hidden layer is often called a *deep network*.

As we will see, next week, some networks have additional filters (e.g., convolution) or tranformation (e.g., pooling) between layers.

Putting these layers together, a multi-layer feed-forward neural network looks like:



These networks will be the focus of our study today.

The Statistical Problem

A MLFFNN represents a parametrized function of its inputs, with parameters consisting of all the weights and biases. We can use these functions to fit complex data.

Such networks provide an *efficiently trainable* family that can *approximate* a broad class of functions.

A principle of **universality** indicates that a class of functions can be approximated arbitrarily well by a specified family of functions. An MLFFNN's have such a property.

For example:

- An arbitrary (Borel measurable) real-valued function on the real numbers can be approximated arbitrary closely by the family of simple functions (finite, linear combination of indicators of measurable sets).
- An arbitrary (computable) function can be approximated by a family of binary circuits comprised of NAND (not-and) gates.

Reflection 1

Construct a specific (e.g., including the weights and biases) single-hidden-layer network with one linear input, one linear output, and two perceptron hidden nodes that gives the indicator function of the interval $[1, 2]$ (don't worry about the endpoints).

If you modify your network to use sigmoidal nodes in the hidden layer (using, say, the logistic function), how well can you approximate the target function?

How does this result lead to a universality theorem for single-hidden layer, feed-forward neural nets? Roughly what does the theorem say?

Reflection 2

A (two-input) NAND gate is a binary function that takes two binary inputs and returns the complement of the inputs' logical and.

a	b	(NAND a b)
0	0	1
0	1	1
1	0	1
1	1	0

Construct a single-node neural network with binary inputs that is equivalent to a NAND gate.

Optional Exercise (for later)

Combine several NAND gates from Reflection 2 to construct a network with two binary inputs and two binary outputs that computes the binary sum of its inputs. (The two binary outputs corresponds to the two binary digits of the sum.) Hint: this network may have some connections *within* layer.

Exercise

Design a data structure or class that represents a MLFFNN, along with a function to construct such a network with random Normal weights and biases.

For simplicity, you may assume that all nodes in that layer have the same type.

Keep this lightweigt and brief, we will build on this as we go.

Mathematical Setup and Forward Propagation

To do computations with these networks, it will be helpful to define the quantities involved carefully. In particular, we will express the computations in terms of matrices and vectors associated with each layer. This will not only make the equations easier to work with, but it will also enable us to use high-performance linear algebra algorithms in our calculations.

At layer ℓ in the network, for $\ell = 1, \ldots, L$, define

- n_ℓ be the number of nodes in the layer.

- Weight matrix W_ℓ, where $W_{\ell,jk}$ is the weight from node j in layer $\ell - 1$ to node k in layer ℓ.
- Bias vector b_ℓ, where $b_{\ell,j}$ is the bias parameter for node j in layer ℓ.
- Activation vector a_ℓ, where $a_{\ell,j}$ is the activation *produced* by node j in layer ℓ. The *input vector* x is labeled a_0.
- The *weighted input* vector $z_\ell = W_\ell^T a_{\ell-1} + b_\ell$, which will be convenient for some calculations.

We thus have the recurrence relation:

$$\begin{aligned} a_\ell &= \phi(W_\ell^T a_{\ell-1} + b_\ell) \\ &= \phi(z_\ell) \\ a_0 &= x. \end{aligned}$$

for layers $\ell = 1, \ldots, L$.

Question: What is W_1 in the typical case where the input layer simply reads in one input value per node?

Activity

Define a function `forward(ffnetwork, inputs, ...)` that takes a network and a vector of inputs and produces a vector of network outputs.

```
# Here, we use a simple way to access the network's attributes,
# but other (possibly better) ways are possible.

forward <- function(network, input) {
    L <- network$L
    z <- vector("list", L)
    a <- vector("list", L)

    activations <- input
    for ( ell in 1:L ) {
        z <- network$W[ell,,] %*% activations + network$b[ell,]
        activations <- network$phi[ell](z)

        z[[ell]] <- z
        a[[ell]] <- activations
    }
    return( list(output=a[[L]], a=a, z=z, input=input, L=L) )
}
```

Learning: Back Propagation and Stochastic Gradient Descent

Our next goal is to help a neural network **learn** how to match the output of a desired function (empirical or otherwise).

In a typical supervised-learning situation, we **train** the network, fitting the model parameters $\theta = (b_1, \ldots, b_L, W_1, \ldots, W_L)$, to minimize a cost function $C(y, \theta)$ that compares expected outputs on some *training sample* $\mathcal{T}$ of size n to the network's predicted outputs.

In general, we will *assume* that

$$C(y, \theta) = \frac{1}{n} \sum_{x \in \mathcal{T}} C_x(y, \theta),$$

where C_x is the cost function for that training sample. We also *assume* that the \$\theta\$-dependence of $C(y, \theta)$ is only through a_L.

But for now, we will consider a more specific case:

$$C(y, \theta) = \frac{1}{2n} \sum_{x \in \mathcal{T}} \|y(x) - a^L(x, \theta)\|^2.$$

There are other choices to consider in practice; an issue we will return to later.

Henceforth, we will treat the dependence of a^L on the weights and biases as implicit. Moreover, for the moment, we can ignore the sum over the training sample and consider a single point x, treating x and y as fixed. (The extension to the full training sample will then be straightforward.) The cost function can then be written as $C(\theta)$, which we want to *minimize*.

Interlude: Gradient Descent

Suppose we have a real-valued function $C(\theta)$ on a multi-dimensional parameter space that we would like to *minimize*.

For small enough changes in the parameter, we have

$$\begin{aligned} \Delta C &\approx \sum_k \frac{\partial C}{\partial \theta_k} \Delta \theta_k \\ &= \frac{\partial C}{\partial \theta} \cdot \Delta \theta, \end{aligned}$$

where $\Delta\theta$ is a vector $(\Delta\theta)_k = \Delta\theta_k$ and where $\frac{\partial C}{\partial \theta} \equiv \nabla C$ is the **gradient** of C with respect to θ, a vector whose k^{th} component is $\frac{\partial C}{\partial \theta_k}$.

We would like to choose the $\Delta\theta$ to reduce C. If, for small $\eta > 0$, we take $\Delta\theta = -\eta \frac{\partial C}{\partial \theta}$, then $\Delta C = -\eta \|\frac{\partial C}{\partial \theta}\|^2 \leq 0$, as desired.

The **gradient descent** algorithm involves repeatedly taking

$$\theta' \leftarrow \theta - \eta \frac{\partial C}{\partial \theta}$$

until the values of C converge. (We often want to adjust η along the way, typically reducing it as we get closer to convergence.)

This reduces C like a ball rolling down the surface of the functions graph until the ball ends up in a local minimum. When we have a well-behaved function C or start close enough to the solution, we can find a global minimum as well.

For neural networks, the step-size parameter η is called the **learning rate**.

So, finding the partial derivative of our cost function C with respect to the weights and biases gives one approach neural network learning.

Unfortunately, this is costly because calculating the gradient requires calculating the cost function *many times*, each of which in turn requires a forward pass through the network. This tends to be slow.

Instead, we will consider an algorithm that computes all the partial derivatives we need using only one forward pass and one backward pass through the network. This method, **back propagation**, is much faster than naive gradient descent.

Back Propagation

The core of the **back propagation** algorithm involves a recurrence relationship that lets us compute the gradient of C. The derivation is relatively straightforward, but we will not derive these equations today. A nice development is given here if you'd like to see it, which motivates the form below.

To start, we will define two specialized products. First, the *Hadamard product* of two vectors (or matrices) of the same dimension to be the elementwise product, $(u \star v)_i = u_i v_i$ (and similarly for matrices). Second, the *outer product* of two vectors, $u \odot v = uv^T$, is the matrix with i, j element $u_i v_j$.

We will also assume that the activation function ϕ and its derivative ϕ' are *vectorized*.

Also, it will be helpful to define the intermediate values $\delta_{\ell,j} = \frac{\partial C}{\partial z_{\ell,j}}$, where z_ℓ is the weighted input vector. Having the vectors δ_ℓ makes the main equations easier to express.

The four main backpropagation equations are:

$$\begin{aligned}
\delta_L &= \frac{\partial C}{\partial a_L} \star \phi'(z_L) && (1)\\
&= (y - a_L(x)) \star \phi'(z_L) \\
\delta_{\ell-1} &= (W_\ell \delta_\ell) \star \phi'(z_{\ell-1}) && (2)\\
\frac{\partial C}{\partial b_\ell} &= \delta_\ell && (3)\\
\frac{\partial C}{\partial W_\ell} &= a_{\ell-1}\delta_\ell^T. && (4)
\end{aligned}$$

For the last two equations, note that the gradients with respect to a vector or matrix are vectors or matrices of the same shape (with corresponding elements, i.e., $\partial C/\partial W_{\ell,jk} = a_{\ell-1,j}\delta_{\ell,k}$).

In the back propagation algorithm, we think of δ_L as a measure of output error. For our mean-squared error cost function, it is just a scaled residual. We propagate this error backward through the network via the recurrence relation above to find all the gradients.

The algorithm is as follows:

1. **Initialize**. Set $a_0 = x$, the input to the network.
2. **Feed forward**. Find a_L by the recurrence $a_\ell = \phi(W_\ell^T a_{\ell-1} + b_\ell)$.
3. **Compute the Output Error**. Initialize the backward steps by computing δ_L using equation (1).
4. **Back Propagate Error**. Compute successive δ_ℓ for $L-1, \ldots, 1$ by the recursion (2).
5. **Compute Gradient**. Gather the gradients with respect to each layer's weights, biases via equations (3) and (4).

While we have used the same symbol ϕ for the activation function in each layer, the equations and algorithm above allow for ϕ to differ *across layers*.

Above we computed the gradient of a loss function based on a single sample. But given any training sample, the resulting loss function and corresponding gradients are just the *average* of what we get for a single training point. (Equations 1-4 work in both cases.)

We can now use the gradients from backward propagation for gradient descent. But there is a more efficient approach to the same goal...

Stochastic Gradient Descent

In practice, using the entire training sample (which may be quite large) to compute *each* gradient is wasteful. We can in fact *approximate* the gradient over the entire training set with only a subset, such as a random sample or even a single instance. This approximation is called **stochastic gradient descent**.

In practice, this is usually done as follows:

- Training proceeds in a series of *epochs*, each of which comprises a pass through the entire training set.
- During an epoch, the training set is divided into non-overlapping subsets, called *mini-batches*, each of which is used in a pass of stochastic gradient descent.
- After each epoch, the training set is randomly *shuffled*, so that the mini-batches used across epochs are different.
- Across epochs, the *learning rate* is adjusted, either adaptively or according to a reduction schedule.
- Before training, the network is usually initialized with random weights and biases.

Pseudo-code:

```
0. Select initial values for the parameters theta. This is
   often done by choosing them randomly.

   Initialize the learning rate \eta and the
   schedule for changing it as the algorithm proceeds.

1. Until stopping conditions are met, do:

   a. Shuffle the training set and divide into m mini-batches
      of designated size.

   b. For b = 1, ..., m, set

      theta = theta - eta * gradient(C_b(theta))

      where C_b is the cost function for mini-batch b

   c. Make any scheduled adjustments to the learning rate
```

Activity

Sketch a function `backprop(network, input, ...)` to implement the back propagation algorithm. This should use your `forward()` function and the data structures you designed earlier.

Suggestion: Start `backprop` by treating all the quantities you need as local variables in your function. Once you get the recurrence to your liking, you can rearrange things.

```
backprop <- function(network, input, observed) {
    predicted <- forward(network, input)
    residual <- (...)
    L <- network$L
    a <- predicted$a
    z <- predicted$z
    delta <- vector("list", L)

    ...  # assume you have phi'[ell] in network

    return(dCdb=..., dCdW=...)
}
```