Going Deep: Neural Networks, Part II
Statistics 650/750
Week 15 Tuesday

Christopher Genovese and Alex Reinhart

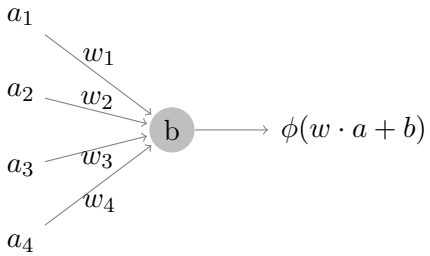
05 Dec 2017

Announcements

- Alex will be out of town again Wednesday 6 Dec, so his office hours will be canceled. He will hold his Friday hours, though.
- I will start my office hours 30 minutes early to help make up for the missing Wednesday office hours.
- As announced in emails, homework must be submitted by 5 Dec (today) to guarantee grading by 10 Dec, giving you time to revise. Homework submitted after 5 Dec cannot be revised.
- Final homework submissions 12 Dec, final Challenge 2 submissions 14 Dec. No revisions after these dates.
- Notes on line in `documents` repository `weekDT.*`.

Neural Nets: Review

(Artificial) Neural networks are computational models comprised of interconnected nodes with a basic structure that looks like:



where the **weights** (w_i) and the **bias** (b) are model parameters we select (and fit) and the **activation function** ϕ is a configuration parameter, often chosen from one of the following

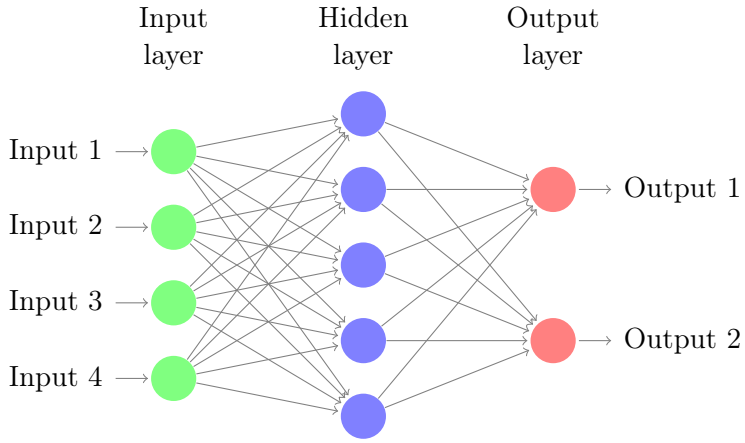
- Linear $\phi(z) = z$
- Perceptron/Heaviside $\phi(z) = 1_{(0,\infty)}(z)$.
- Sigmoidal $\phi(z)$ is a sigmoidal (S-shaped) function, most commonly the *logistic function*

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

  but sometimes the Gaussian cdf or a shifted and scaled version of tanh.

- Rectified Linear Unit (aka ReLU) $\phi(z) = \max(z, 0)$.
- $\cdots$

A common type of network is the **multi-layer feed-forward neural network** which has an *input layer*, an *output layer*, and one or more *hidden layers*.



It is useful to define the parameters (and some intermediate quantities) for this model in matrix-vector terms.

At layer ℓ in the network, define

- 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:

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

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

Last time, you also designed a (rough) data structure for this kind of network and implemented the `forward` operation from inputs to outputs.

We also saw last time that these networks have a **universality property** and that in fact with only *single-hidden-layer feed-forward neural network* (of sufficient size), we can approximate arbitrarily well any simple function and thus as well any (Borel meaurable) real-valued function of a single real argument. Similar more general universality theorems can be proved.

Brief Exercise

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 Exercise 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.

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 loss 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 loss function for that training sample. We also *assume* that the θ-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 loss 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. Why?)

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 loss function C with respect to the weights and biases gives one approach neural network learning.

Question. What does computing the gradient cost?

```
To calcualte the gradient, we (usually) need to compute
the loss functions many time. Each calculation of the
loss function (for a single training point) requires
a feed-forward propogation through the network.
This is, relatively speaking, costly.
```

Unfortunately, for this reason, the naive gradient descent 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, $u \odot v$, is the matrix with element i, j equal to $u_i v_j$.

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:

$$\begin{aligned}
\delta_L &= \frac{\partial C}{\partial a_L} \star \phi'(z_L) && (1)\\
&= (y - a_L(x)) \star \phi'(z_L) \\
\delta_{\ell-1} &= (W_\ell \delta_\ell) \star \phi'(z_{\ell-1}) && (2)\\
\frac{\partial C}{\partial b_\ell} &= \delta_\ell && (3)\\
\frac{\partial C}{\partial W_\ell} &= a_{\ell-1} \odot \delta_\ell. && (4)
\end{aligned}$$

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 loss 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 $a_\ell = \phi(W_\ell^T a_{\ell-1} + b_\ell)$. (This is your `forward` function.)
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, however, using the entire training sample (which may be quite large) to compute *each* gradient is wasteful. For instance, in the early stages of the search, the gradient need not be completely accurate to get us closer to our goal. 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 *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 (interactive):

Activity

This activity works best in groups of two or three. Consider dividing your task among the group members, remaining in discussion as you do so.

1. Write a function `backprop(network, input, output, ...)` to implement the back propagation algorithm. (The `...` represents additional arguments that you need or desire.) Your function should return (at least) the relevant gradient.

   This should use your `forward()` and data structure design from last time.

2. Write a function `trainFFNN()` that uses `backprop` to train a network using stochastic gradient descent. The arguments should specify the number of mini-batches, end conditions on iteration (e.g., tolerance – absolute or proportional – on changes in the cost function) and as well as *some* scheduling of learning rate changes.

3. Generate a simple data set and train the network on a substantial fraction of that data set. How well does the network perform in matching the outputs for the rest of the data?

A look forward: What's Deep about Deep Learning?

We've seen that single-hidden-layer FF neural networks can approximate arbitrarily complicated functions, if given enough nodes.

What would we gain or lose by having more hidden layers? And is there a qualitative difference between say 2-10 hidden layers and 1000, if we control for the total number of model parameters?

And if we do use more hidden layers, what effect will this have on training? Will backpropogation with stochastic gradient descent still work? Will it still be efficient?

We will look at these questions. But as a preview it's worth considering why *depth* per se is viewed as desirable:

- Efficiency of representation

  Universality is not enough

- "Modularity" of representation

  Specialized layers (convolutional, pooling, softmax, ...) help design.

Finally, we need to consider types of networks beyond feed-forward (DAG) style. An important example of that is **recurrent** neural networks, where a node's output at one time can influence that node (or others) at a later time.