Week 5: Neural Networks

Instructor: Sergey Levine

1 Backpropagation Base Case Summary

First, let's quickly summarize how a neural network works. The equation from last time that describes how we compute activations is given by:

$$h^{(\ell)} = \sigma(\mathbf{W}^{(\ell)} h^{(\ell-1)} + \mathbf{b}^{(\ell)}).$$

We have L layers, and by convention $h^{(0)} = \mathbf{x}$. In the binary classification case, we have $p(y = 1|\mathbf{x}) = h^{(L)}$, and $p(y = 0|\mathbf{x}) = 1 - h^{(L)}$. If $L = 1$, we simply recover logistic regression. Each layer has a matrix $\mathbf{W}^{(\ell)}$ and bias vector $\mathbf{b}^{(\ell)}$. The size of each hidden activation vector $h^{(\ell)}$ is M_ℓ, and M_0 is the number of attributes and $M_L = 1$ for binary classification. Each matrix $\mathbf{W}^{(\ell)}$ is $M_\ell \times M_{\ell-1}$, and each bias vector $\mathbf{b}^{(\ell)}$ has size M_ℓ. For convenience, we also introduce

$$z^{(\ell)} = \mathbf{W}^{(\ell)} h^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

such that

$$h^{(\ell)} = \sigma(z^{(\ell)}) = \sigma(\mathbf{W}^{(\ell)} h^{(\ell-1)} + \mathbf{b}^{(\ell)}).$$

In order to optimize the conditional log-likelihood with respect to the parameters $\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}\}$, we use gradient ascent, which requires computing the gradient with respect to each of the parameters. In the last lecture, we saw how to do this in the "base case" where we want the derivatives with respect to the last layer weights $\mathbf{W}^{(L)}$ and $\mathbf{b}^{(L)}$. This derivation was for the (simple) case where we only have one datapoint, so $N = 1$. We'll see what happens when $N > 1$ today, and extend this to the recursive case to compute the gradient with respect to all datapoints.

If we have one datapoint, we have

$$\mathcal{L}(h^{(L)}) = \begin{cases} y^1 = 0 : \log(1 - h^{(L)}) \\ y^1 = 1 : \log(h^{(L)}) \end{cases}$$

so we can differentiate with respect to the log likelihood according to

$$\frac{d\mathcal{L}}{dh^{(L)}} = \begin{cases} y^1 = 0 : \frac{-1}{1-h^{(L)}} \\ y^1 = 1 : \frac{1}{h^{(L)}} \end{cases}$$

We then apply the chain rule to get $\frac{d\mathcal{L}}{dz^{(L)}}$, according to:

$$\frac{d\mathcal{L}}{dz^{(L)}} = \frac{d\mathcal{L}}{dh^{(L)}} \frac{dh^{(L)}}{dz^{(L)}} = \frac{d\mathcal{L}}{dh^{(L)}} \circ \sigma(z^{(L)}) \circ (1 - \sigma(z^{(L)})),$$

where $\circ$ denotes an elementwise product, since σ is applied to each element of $z^{(L)}$ independently, and therefore $z_i^{(L)}$ only affects $h_j^{(L)}$ if $i = j$.

Once we have $\frac{d\mathcal{L}}{dz^{(L)}}$, we can again apply chain rule to get derivatives with respect to $\mathbf{W}^{(L)}$ and $\mathbf{b}^{(L)}$, by using the fact that

$$z^{(L)} = \mathbf{W}^{(L)} h^{(L-1)} + \mathbf{b}^{(L)}.$$

For the bias, we have:

$$\frac{d\mathcal{L}}{d\mathbf{b}_i^{(L)}} = \sum_{j=1}^{M_1} \frac{d\mathcal{L}}{dz_j^{(L)}} \frac{dz_j^{(L)}}{d\mathbf{b}_i^{(L)}} = \frac{d\mathcal{L}}{dz_i^{(L)}}.$$

And for the weights matrix, we have:

$$\frac{d\mathcal{L}}{d\mathbf{W}_{i,j}^{(L)}} = \sum_{k=1}^{M_1} \frac{d\mathcal{L}}{dz_k^{(L)}} \frac{dz_k^{(L)}}{d\mathbf{W}_{i,j}^{(L)}} = \frac{d\mathcal{L}}{dz_i^{(L)}} h_j^{(L-1)}.$$

This is all for the case where there is exactly one datapoint, and therefore $\frac{d\mathcal{L}}{dh^{(L)}}$ has one column. If we have multiple datapoints (so $N > 1$), we'll simply treat the datapoints as one huge matrix, just like we did in linear regression. So $\mathbf{x}$ will be $M_0 \times N$, and $h^{(\ell)}$ will be $M_\ell \times N$. The log-likelihood can then be written as

$$\mathcal{L}(h^{(L)}) = \sum_{i=1}^{N} \begin{cases} y^i = 0 : \log(1 - h_{1,i}^{(L)}) \\ y^i = 1 : \log(h_{1,i}^{(L)}) \end{cases}$$

We can therefore fill the $1 \times N$ matrix $\frac{d\mathcal{L}}{dh^{(L)}}$ according to:

$$\frac{d\mathcal{L}}{dh_{1,i}^{(L)}} = \begin{cases} y^i = 0 : \frac{-1}{1-h_{1,i}^{(L)}} \\ y^i = 1 : \frac{1}{h_{1,i}^{(L)}} \end{cases}$$

Now, we can actually repeat the entire derivation above for the matrix case, using matrix multiplication instead of vector multiplication, and all of the math is exactly the same, except that we have to sum over the datapoints when we compute the gradients of the parameters, so we get:

$$\frac{d\mathcal{L}}{d\mathbf{b}_j^{(L)}} = \sum_{i=1}^{N} \frac{d\mathcal{L}}{dz_{j,i}^{(L)}}.$$

and

$$\frac{d\mathcal{L}}{d\mathbf{W}_{k,j}^{(L)}} = \sum_{i=1}^{N} \frac{d\mathcal{L}}{dz_{k,i}^{(L)}} h_{j,i}^{(L-1)}.$$

If we write this in matrix notation, we simply get

$$\begin{aligned} \frac{d\mathcal{L}}{d\mathbf{b}^{(L)}} &= \frac{d\mathcal{L}}{dz^{(L)}} \vec{1}_N^T \\ \frac{d\mathcal{L}}{d\mathbf{W}^{(L)}} &= \frac{d\mathcal{L}}{dz^{(L)}} (h^{(L-1)})^T, \end{aligned}$$

where $\vec{1}_N$ is simply a vector of length N where each entry is 1.

2 Backpropagation Recursive Case

How can we get the derivatives with respect to the weights in the preceding layers? Well, we note that the previous layers only affect $\mathcal{L}$ via $h^{(L-1)}$, so we simply need to know $\frac{d\mathcal{L}}{dh^{(L-1)}}$. Since we know that

$$z^{(L)} = \mathbf{W}^{(L)} h^{(L-1)} + \mathbf{b}^{(L)}$$

and therefore

$$z_i^{(L)} = \sum_{j=1}^{M_{L-1}} \mathbf{W}_{i,j}^{(L)} h_j^{(L-1)} + \mathbf{b}_i^{(L)},$$

we can derive

$$\frac{d\mathcal{L}}{dh_j^{(L-1)}} = \sum_{i=1}^{M_L} \frac{d\mathcal{L}}{dz_i^{(L)}} \frac{dz_i^{(L)}}{dh_j^{(L-1)}} = \sum_{i=1}^{M_L} \frac{d\mathcal{L}}{dz_i^{(L)}} \mathbf{W}_{i,j}^{(L)} = \sum_{i=1}^{M_L} \left((\mathbf{W}^{(L)})^T \right)_{j,i} \frac{d\mathcal{L}}{dz_i^{(L)}}$$

in matrix notation, this simply becomes

$$\frac{d\mathcal{L}}{dh^{(L-1)}} = (\mathbf{W}^{(L)})^T \frac{d\mathcal{L}}{dz^{(L)}}.$$

In the case where we have more than one datapoint, the same exact derivation can be repeated for each column of $\frac{d\mathcal{L}}{dh^{(L-1)}}$ (recall that there are N columns), and the math is exactly the same in matrix notation. Now that we have $\frac{d\mathcal{L}}{dh^{(L-1)}}$, we simply apply the equations in the previous section to get the gradients with respect to $\mathbf{W}^{(L-1)}$ and $\mathbf{b}^{(L-1)}$, and then repeat and compute the derivative with respect to $h^{(L-2)}$.

3 Algorithm Summary

The full backpropagation algorithm is shown below. We assume that each $h^{(\ell)}$ and $z^{(\ell)}$ is arranged into a M_ℓ by N matrix.

Algorithm 1 Backpropagation

```
1: $\ell \leftarrow L$
2: compute $\frac{d\mathcal{L}}{dh^{(L)}}$ by differentiating the log-likelihood
3: **while** $\ell > 0$ **do**
4:     $\frac{d\mathcal{L}}{dz^{(\ell)}} \leftarrow \frac{d\mathcal{L}}{dh^{(L)}} \circ \sigma(z^{(\ell)})(1 - \sigma(z^{(\ell)}))$
5:     $\frac{d\mathcal{L}}{d\mathbf{b}^{(\ell)}} \leftarrow \frac{d\mathcal{L}}{dz^{(\ell)}}$
6:     $\frac{d\mathcal{L}}{d\mathbf{W}^{(\ell)}} \leftarrow \frac{d\mathcal{L}}{dz^{(\ell)}} (h^{(\ell-1)})^T$
7:     $\frac{d\mathcal{L}}{dh^{(\ell-1)}} \leftarrow (\mathbf{W}^{(\ell)})^T \frac{d\mathcal{L}}{dz^{(\ell)}}$
8:     $\ell \leftarrow \ell - 1$
9: **end while**
```

Note that we only ever need to keep track of one partial derivative with respect to either post-synaptic or pre-synaptic activations, so in practice it is often convenient to denote this quantity as δ, which is sometimes referred to as the error, diff, or backward signal. Then, the algorithm looks like this:

Algorithm 2 Backpropagation

```
1: $\ell \leftarrow L$
2: $\delta \leftarrow \frac{d\mathcal{L}}{dh^{(L)}}$
3: while $\ell > 0$ do
4:     $\delta \leftarrow \delta \circ \sigma(z^{(\ell)})(1 - \sigma(z^{(\ell)}))$
5:     $\frac{d\mathcal{L}}{d\mathbf{b}^{(\ell)}} \leftarrow \delta$
6:     $\frac{d\mathcal{L}}{d\mathbf{W}^{(\ell)}} \leftarrow \delta(h^{(\ell-1)})^T$
7:     $\delta \leftarrow (\mathbf{W}^{(\ell)})^T \delta$
8:     $\ell \leftarrow \ell - 1$
9: end while
```

4 Comparison with Logistic Regression

We discussed on Monday how logistic regression compares with naïve Bayes, and saw how naïve Bayes introduces additional assumptions (feature independence) to reduce overfitting at the expense of introducing bias. So logistic regression has less bias (meaning lower training error, it fits the data better), but because it is more expressive (meaning it can capture more interesting and complex functions), it can overfit more when the dataset is small or the input is too high-dimensional. Neural networks are even more extreme in this regard: neural networks can represent many more functions that logistic regression, especially if we use many layers and many hidden units. This means they have less bias, which means they can fit really complicated functions better. However, they are more vulnerable to overfitting, and therefore work best when data is very plentiful.

5 Stochastic Gradient Ascent (or Descent)

Since neural networks work best when there is a lot of data, the computational cost of gradient ascent can be quite high, since we have to keep track of matrices with N columns. Each iteration takes $O(N)$ time, and we might need many iterations. There is a simple trick we can use to substantially accelerate gradient ascent on the neural network log-likelihood (or gradient descent on the negative log-likelihood) called "stochastic gradient descent" (SGD). In these notes, I'll actually describe "stochastic gradient ascent," but it's typically called SGD in practice, and performs descent on the negative log-likelihood. The math is exactly the same in both cases, it's just a matter of whether or not you put the negative sign on $\mathcal{L}$ (and then go in the direction of the negative gradient) or leave the negative sign off and go in the direction of the positive gradient.

The idea in SGD is to sample a small batch from your training data at each iteration. You can think of this as choosing a different tiny "training set" each time. There is a formal reason why this is still guaranteed to converge to a local optimum, but the intuition is that our training set gives us a sample-based estimate of the gradient of the "true" likelihood (which, as you recall, requires integrating over *all* possible datapoints in existence). So if the training set is a big sample-based estimate, using a smaller sample-based estimate is just as "correct," but since we have fewer samples, the estimate of the "true" gradient is more noisy. But much faster. So SGD works like this:

Algorithm 3 SGD

```
1: Initialize $\theta^{(i)}$ (e.g. randomly)
2: while not converged do
3:     Sample a batch $\mathcal{D}^{(i)}$ from the dataset $\mathcal{D}$ by randomly picking $B$ points
4:     Compute $\nabla\mathcal{L}_{\mathcal{D}^{(i)}}(\theta^{(i)})$ using the datapoints in the batch $\mathcal{D}^{(i)}$
5:     $\theta^{(i+1)} \leftarrow \theta^{(i)} + \alpha\nabla\mathcal{L}_{\mathcal{D}^{(i)}}(\theta^{(i)})$
6: end while
```

We typically need to use a lower learning rate α if we use smaller batches with smaller values of B, because our gradient is less exact with fewer samples. Reasonable values of B tend to vary, but something on the order of 50 to 100 is reasonable. There are also many tricks and modifications to the SGD algorithm that people employ to improve convergence and stability, most of which are centered around cleverly setting α. This tends to be a bit difficult, especially for very large networks, and you may need to try a few α settings to get a good answer. Often, if the algorithm seems to convergence, decreasing α a bit will improve the training likelihood further.

6 Non-Binary Outputs

We can handle non-binary categorical outputs $y \in \{1, \dots, L_y\}$ in the same way as we did for logistic regression, by using

$$p(y = j|\mathbf{x}, \theta) = \frac{\exp\left(\sum_{i=1}^{M_{L-1}} \mathbf{W}_{j,i}^{(L)} h_i^{(L-1)} + \mathbf{b}_j^{(L)}\right)}{\sum_{j'=1}^{L_y} \exp\left(\sum_{i=1}^{M_{L-1}} \mathbf{W}_{j',i}^{(L)} h_i^{(L-1)} + \mathbf{b}_j^{(L)}\right)}.$$

We can write this as

$$p(y = j|\mathbf{x}, \theta) = h_j^{(L)},$$

where the last layer has a different nonlinearity. So instead of $h^{(L)} = \sigma(z^{(L)})$, we use $h^{(L)} = \text{softmax}(z^{(L)})$, where softmax$(z)$ is given by

$$\text{softmax}(z) = \begin{bmatrix} \exp(z_1)/\sum_{j=1}^{M_L} \exp(z_j) \\ \exp(z_2)/\sum_{j=1}^{M_L} \exp(z_j) \\ \cdots \\ \exp(z_{M_L})/\sum_{j=1}^{M_L} \exp(z_j) \end{bmatrix}$$

which we can write more concisely as following (assuming exp is applied elementwise):

$$\text{softmax}(z) = \exp(z)/\sum_{j=1}^{M_L} \exp(z_j).$$

Note that all previous layers still use σ as the nonlinearity, so if the network has, for example, 3 hidden layers, we can unroll it as:

$$h^{(3)} = \text{softmax}(\mathbf{W}^{(3)}\sigma(\mathbf{W}^{(2)}\sigma(\mathbf{W}^{(1)}h^{(0)} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}) + \mathbf{b}^{(3)}).$$

We can compute gradients in this network exactly the same as we did before, except that the gradient of the softmax function is a little different. Deriving this gradient is left as an exercise.

We can also handle real-valued outputs. This requires us a different model for the log-likelihood. A convenient model is the same one we used in linear regression, where $p(y|\mathbf{x}, \theta) \sim \mathcal{N}(h^{(L)}, v^2)$ (I used v^2 here for the variance, since we're already using σ for the nonlinearity function). As in linear regression, we assume v^2 is some constant, since the optimal solution will still be the same. The conditional log-likelihood is

$$\mathcal{L}(\theta) = -\frac{1}{2}\sum_{i=1}^{N}(y^i - h_{1,i}^{(L)})^2.$$

So we want each entry in $h^{(L)}$ to be in $\mathbb{R}$. That means that again we have to change the output nonlinearity. This time, instead of using σ or softmax, we simply use nothing at all. So we get

$$h^{(3)} = \mathbf{W}^{(3)}\sigma(\mathbf{W}^{(2)}\sigma(\mathbf{W}^{(1)}h^{(0)} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}) + \mathbf{b}^{(3)}.$$

In the same way that a neural network for classification is a generalization of logistic regression with additional hidden layers, neural networks for regression are a generalization of linear regression with additional hidden layers.

Finally, just like in logistic regression, we can switch from MLE to MAP by putting a prior on the weights. The quadratic penalty is the most popular prior, and corresponds simply to adding the following terms to the objective:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \log p(y^i|\mathbf{x}^i, \theta) - \lambda \sum_{\ell=1}^{L}\sum_{i,j}(\mathbf{W}_{i,j}^{(\ell)})^2.$$

Note that the bias terms are typically not regularized, and λ is sometimes called the "weight decay" because it encourages the weights to take on smaller values.