Backpropagate a Two-Layered Neural Network: Theory and Implementation

This post serves as a note after working through the Stanford CS231n class note and its supplement. The former provides the theory and Python implementation on foward and backward propagating a neural netowrk but touches little on how the derivatives are actually computed. The latter provides a small (but concrete) example on how to backprop a linear layer in a neural network without explicitly isolating the bias term. Mathematical techinques used in this post are mainly taken from the latter.

In this post we will derive complete rules for backpropagating a two-layered neural network. We isolate the bias term and apply activation functions to the linear layer to aid completeness. Additional math notations are introduced. Although it's a toy example, it will serve us well when we go deeper to derive BP rules for deep neural networks or even recurrent or convolutional neural networks.

Define the Network

In this section we define a two-layered neural network. Specifying the weight matrix and the bias terms for the each layer. For the first layer, the weight and bias are superscripted with 1. The matrices are superscripted with 2 for the second layer.

$$X \longrightarrow \left[XW^1 + b^1 = Y^1 \rightarrow f(Y^1) = H^1\right]_1 \longrightarrow \left[H^1W^2 + b^2 = \hat{y}\right]_2 \longrightarrow L$$

The bracket with subscript indicates the layer and encloses everything inside it, the parameters (W and b) and the *operations* (linear and nonliner). Note for the hidden layer (subscripted 1), we apply activation function f after the linear transformation while for the output layer, we don't. It's possible to also apply activation functions to the output layer if you want the output to be in a certain range. An example would be *logistic regression*, which uses logistic function as activation for the output.

*Note we assign a intermediate variable to each step of the computation. These variables will be useful in deriving BP rules later.

We examine the network structure term by term:

- X : is the data matrix containing N training examples, each with dimension D. Hence X is of shape $N \times D$ with each row being a training example. This can be thought of we are training the neural network on *mini-batches* rather than feeding one training exmaple at a time.
- The first / hidden layer
    1. Takes the data matrix X; multiply the weight matrix W^1; add bias vecto b^1.
    2. Pass the whole term $Y^1 = XW^1 + b^1$ into the *activation function* f to produce the output of the layer $H^1 = f(Y^1) = f(XW^1 + b^1)$.
- The second / output layer
    1. Take the output of the first layer H^1; multiply by its weight W^2; add the bias term b^2.
    2. No activation for the output layer. Hence the final output would be $\hat{y} = H^1W^2 + b^2$. Here we use $\hat{y}$ to denote the prediction of the model.
- L is the loss function. It's a scalar-valued, differentiable (with respect to the output of the network) function that we want to compute the gradient on. Once we have the gradient of the loss function with respect to every model parameter, we can perform gradient descent and parameter update on them.

  The above procedure of computing output and loss from the input is called the *forward-propagation / forward pass*. In contrast to the *back-propagation (backward pass)* that we will mention below.

Backpropagation: The Theory

Gradients

First, we define the gradient of a *scalar* function $f : \mathbb{R}^{m \times n} \to \mathbb{R}$ to be

$$\nabla_A f(A) \doteq \left(\frac{\partial f}{\partial A_{ij}}\right)_{ij}$$

i.e. the gradient of f is of the same shape as A, with each entry being the partial derivative of f w.r.t. the ij-th element of A.

The definition of gradient also extends to functions that takes *both* input and output as matrix. Formally, for $\mathbf{F} : \mathbb{R}^{m \times n} \to \mathbb{R}^{p \times q}$, we define the gradient of $\mathbf{F}$ to be

$$\nabla_A \mathbf{F}(A) \doteq \left(\frac{\partial \mathbf{F}}{\partial A_{ij}} \right)_{ij}$$

The gradient now becomes a *block* matrix with each element being the matrix of shape $p \times q$, or equivalently, a big matrix of shape $mp \times nq$ (each element is a matrix of shape $p \times q$ and there are $m \times n$ of them). This form will be frequently used in the chain rule as a generalization of vector-vector Jacobian matrix.

Hardamard Product

Element-wise product of two equal-shaped matrices $\mathbf{A}, \mathbf{B}$ is called the *Hadamard product*. Denoted by $\odot$,

$$\mathbf{A} \odot \mathbf{B} \doteq \left(\mathbf{A}_{ij} \mathbf{B}_{ij} \right)_{ij}$$

Note that the resulting matrix shares the same shape as $\mathbf{A}$ and $\mathbf{B}$.

Matrix Dot Product

Dot product is usually defined for vectors only. One way to define dot product is to multiply corresponding elements in the two vectors and sum up the products to produce a scalar value. Here we borrow the concept and define an analogous operation on matrices: if $\mathbf{A}, \mathbf{B}$ are two equal-shaped matrices, the *dot product* of $\mathbf{A}$ and $\mathbf{B}$ is defined as

$$\mathbf{A} \cdot \mathbf{B} \doteq \sum_{i,j} \mathbf{A}_{ij} \mathbf{B}_{ij}$$

Algorithm

The sole purpose of backpropagation is to compute the gradient of the loss function with model parameters. Hence the following terms are of interest:

$$\frac{\partial L}{\partial W^1}, \frac{\partial L}{\partial b^1}, \frac{\partial L}{\partial W^2}, \frac{\partial L}{\partial b^2}$$

Backpropagating means that we are computing the gradient backward. Starting from the very end, i.e. the output of the network. Since the loss function is differentiable function; the derivative $\frac{\partial L}{\partial \hat{y}}$ exists. We leave it that way since we cannot further simply the term since we don't know the exact formula for the loss function. An explicit example of such loss fucntion will be computed in later sections.

Now that we have the gradient on the output, there are three gradients related directly to output $\hat{y}$ that we can compute immediately. They are

$$\frac{\partial L}{\partial H^1}, \frac{\partial L}{\partial W^2}, \frac{\partial L}{\partial b^2}$$

Computing the gradient of $\boldsymbol{H^1}$ enables us to flow the gradient backward through the neural network. So let's start there. Since we know the relation between $\boldsymbol{H^1}$ and $\hat{\boldsymbol{y}}$ (which is just a linear function), the natural choice is to apply chain rule to obtain the derivative of $\boldsymbol{H^1}$. Formally,

$$\frac{\partial L}{\partial H^1} = \frac{\partial L}{\partial \hat{y}} \, ? \, \frac{\partial \hat{y}}{\partial H^1}$$

But we must be very careful with the meaning of the righthand side of the equation. $\frac{\partial L}{\partial \hat{\boldsymbol{y}}}$ is a matrix of the same shape as $\hat{\boldsymbol{y}}$; however, the term $\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{H^1}}$, according to our definition, is a *block* matrix with each element being a matrix of the shape $\hat{\boldsymbol{y}}$. How can these two matrices (of different shapes) multiply? To answer this question, we examine the chain rule for one element of $\boldsymbol{H^1}$:

$$\frac{\partial L}{\partial H^1_{ij}} = \sum_{m,n} \frac{\partial L}{\partial \hat{y}_{mn}} \frac{\partial \hat{y}_{mn}}{\partial H^1_{ij}} = \frac{\partial L}{\partial \hat{\boldsymbol{y}}} \cdot \frac{\partial \hat{\boldsymbol{y}}}{\partial H^1_{ij}}$$

The equality holds for every element of $\boldsymbol{H^1}$. Intuitively speaking, we just consider all the intermediate effects of $\hat{\boldsymbol{y}}$ and sum up all the effects. Note that the operation is just the *dot product* between two matrices of the same shape as $\hat{\boldsymbol{y}}$. Now that we have an expression for each term of $\boldsymbol{H_1}$,

$$\begin{aligned} \frac{\partial L}{\partial H^1} &= \left(\frac{\partial L}{\partial \hat{\boldsymbol{y}}} \cdot \frac{\partial \hat{\boldsymbol{y}}}{\partial H^1_{ij}} \right)_{ij} \\ &= \frac{\partial L}{\partial \hat{\boldsymbol{y}}} \textbf{ dot with every element of } \frac{\partial \hat{\boldsymbol{y}}}{\partial H^1} \end{aligned}$$

That is, what the chain rule really does is "broadcast" the term $\frac{\partial L}{\partial \hat{\boldsymbol{y}}}$ to each element of $\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{H^1}}$ with matrix dot product. It can be shown that this form reduces to the usual Jacobian matrix when reducing $\hat{\boldsymbol{y}}$ and $\boldsymbol{H^1}$ to vectors (Try this yourself!). Now we are clear about what the chain rule really does; for brevity, we will leave the notation to be

$$\frac{\partial L}{\partial H^1} = \frac{\partial L}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial H^1} \, .$$

But one should always keep in mind the true meaning of the notation. In the remainder of the post, we will adapt this notation if not state otherwise. Although we introduced the chain rule for the generalized Jacobian, in practice we almost always use the element-wise chain rule in deriving BP rules.

Now let's compute the exact form for $\frac{\partial L}{\partial \boldsymbol{H^1}}$. Chain rule again:

$$
\begin{aligned}
\frac{\partial L}{\partial H^1_{ij}} &= \sum_{m,n} \frac{\partial L}{\partial \hat{y}_{mn}} \frac{\partial \hat{y}_{mn}}{\partial H^1_{ij}} && \text{chain rule} \\
&= \sum_{m,n} \frac{\partial L}{\partial \hat{y}_{mn}} \frac{\partial}{\partial H^1_{ij}} \sum_k H^1_{mk} W^2_{kn} + b^2_n && \because \hat{y} = H^1 W^2 + b^2 \\
&= \sum_n \frac{\partial L}{\partial \hat{y}_{in}} \underbrace{\frac{\partial}{\partial H^1_{ij}} \sum_k H^1_{ik} W^2_{kn} + b^2_n}_{= W^2_{jn}} && \text{nonzero for } m = i \\
&= \sum_n \frac{\partial L}{\partial \hat{y}_{in}} {W^2}^T_{nj} && \text{transpose of } W^2 \\
&= \left(\frac{\partial L}{\partial \hat{y}} {W^2}^T \right)_{ij}
\end{aligned}
$$

Therefore we have

$$
\boxed{\frac{\partial L}{\partial H^1} = \frac{\partial L}{\partial \hat{y}} {W^2}^T}
$$

which is just the usual matrix multiplication (you should check that the dimension of both sides of the equation really matches). Similar procedure can be carried out for $\frac{\partial L}{\partial W^2}$,

$$
\begin{aligned}
\frac{\partial L}{\partial W^2_{ij}} &= \sum_{m,n} \frac{\partial L}{\partial \hat{y}_{mn}} \frac{\partial \hat{y}_{mn}}{\partial W^2_{ij}} \\
&= \sum_{m,n} \frac{\partial L}{\partial \hat{y}_{mn}} \frac{\partial}{\partial W^2_{ij}} \sum_k H^1_{mk} W^2_{kn} + b^2_n \\
&= \sum_m \frac{\partial L}{\partial \hat{y}_{mj}} \underbrace{\frac{\partial}{\partial W^2_{ij}} \sum_k H^1_{mk} W^2_{kj} + b^2_j}_{= H^1_{mi}} && \text{nonzero for } n = j \\
&= \sum_m {H^1}^T_{im} \frac{\partial L}{\partial \hat{y}_{mj}} && \text{transpose of } H^1 \\
&= \left({H^1}^T \frac{\partial L}{\partial \hat{y}} \right)_{ij}
\end{aligned}
$$

Hence

$$
\boxed{\frac{\partial L}{\partial W^2} = {H^1}^T \frac{\partial L}{\partial \hat{y}}}
$$

The term $\frac{\partial L}{\partial b^2}$ is a bit different since the bias term is a row vector instead of a matrix. Well, the same trick still applies (note here we use only one subscipt since b^2 is a *row vector*.)

$$
\begin{aligned}
\frac{\partial L}{\partial b_j^2} &= \sum_{m,n} \frac{\partial L}{\partial \hat{y}_{mn}} \frac{\partial \hat{y}_{mn}}{\partial b_j^2} \\
&= \sum_{m} \frac{\partial L}{\partial \hat{y}_{mj}} \underbrace{\frac{\partial}{\partial b_j^2} \sum_k H_{mk}^1 W_{kj}^2 + b_j^2}_{=1} \\
&= \sum_{m} \frac{\partial L}{\partial \hat{y}_{mj}} \\
&= \textbf{sum of the j-th column of } \frac{\partial L}{\partial \hat{y}}
\end{aligned}
$$

That is, the j-th element of $\frac{\partial L}{\partial b^2}$ is just the sum of the j-th column of $\frac{\partial L}{\partial \hat{y}}$. The gradient collapses the matrix, reducing the dimension from $N \times \_\_$ to $1 \times \_\_$; therefore matches the dimension of b^2. We write the derivative as

$$
\boxed{\frac{\partial L}{\partial b^2} = \sum_{0} \frac{\partial L}{\partial \hat{y}}}
$$

The reason for writing a 0 under the summation is that column-wise summing a matrix is equivalent to summing a matrix along *axis 0* in Numpy, Python.

After all these work, we finally finish backpropagating through the output layer. But we are almost done! Since the first layer is almost exactly the same as the second layer — except for an additional activation function we need to backpropagate through. Once we obtain the derivative $\frac{\partial L}{\partial Y^1}$ and copying steps from above, all other gradients of the hidden layer will automatically fall in place. Again,

$$
\begin{aligned}
\frac{\partial L}{\partial Y_{ij}^1} &= \sum_{m,n} \frac{\partial L}{\partial H_{mn}^1} \frac{\partial H_{mn}^1}{\partial Y_{ij}^1} \\
&= \frac{\partial L}{\partial H_{ij}^1} \frac{\partial H_{ij}^1}{\partial Y_{ij}^1} \\
&= \frac{\partial L}{\partial H_{ij}^1} \frac{\partial f(Y_{ij}^1)}{\partial Y_{ij}^1}
\end{aligned}
$$

In the last equality, the second term is the derivative of the activation function f evaluate at each element of Y_{ij}, i.e. the operation is *element-wise*. This is where the Hadamard product comes in; we can write the derivative compactly as

$$
\boxed{\frac{\partial L}{\partial Y^1} = \frac{\partial L}{\partial H^1} \odot \nabla f(Y)}
$$

where $\nabla f(Y)$ is the gradient of f *evaluates* at matrix Y. Continue backpropagating to W^1, b^1, we have

$$\boxed{\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y^1} {W^1}^T}$$

$$\boxed{\frac{\partial L}{\partial W^1} = X^T \frac{\partial L}{\partial Y^1}}$$

$$\boxed{\frac{\partial L}{\partial b^1} = \sum_0 \frac{\partial L}{\partial Y^1}}$$

as desired. We can also backpropagated to $\boldsymbol{X}$, but as for the learning task, it's unnecessary since there is no way we can "train" the data to decrease the loss.

Special Cases: ReLU activation and MSE loss function

Here we use the formula derived ealier to calculate the gradient for the Rectified Linear Unit (ReLU) activation and the mean squared error (MSE) loss function. They are one of the most commonly-used activation and loss functions and will serve as a running example for later use.

The MSE loss function is defined as

$$L = \frac{1}{N} \sum_{i,j} (\hat{y}_{ij} - y_{ij})^2$$

which is the mean of squared 2-norm of the error: difference between predicted value and the target value. It's differentiable function with respect to $\hat{\boldsymbol{y}}$; we can compute

$$\begin{aligned}
\frac{\partial L}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}} \frac{1}{N} \sum_{i,j} (\hat{y}_{ij} - y_{ij})^2 \\
&= \left(\frac{1}{N} \frac{\partial}{\partial \hat{y}_{ij}} \sum_{i,j} (\hat{y}_{ij} - y_{ij})^2 \right)_{ij} \\
&= \left(\frac{1}{N} \frac{\partial}{\partial \hat{y}_{ij}} (\hat{y}_{ij} - y_{ij})^2 \right)_{ij} \\
&= \left(\frac{2}{N} (\hat{y}_{ij} - y_{ij}) \right)_{ij} \\
&= \boxed{\frac{2}{N} (\hat{y} - y)}
\end{aligned}$$

The ReLU activation function is defined as

$$f(x) = \max(x, 0)$$

hence a "rectified" linear function. If we plug in a matrix to $\boldsymbol{f}$, we mean applying $\boldsymbol{f}$ element-wise to the matrix. The derivative of $\boldsymbol{f}$ can be easily computed

$$f'(x) = \begin{cases} 0 & \text{for} \quad x \leq 0 \\ 1 & \text{for} \quad x > 0 \end{cases} = 1(x > 0)$$

where $1(\cdot)$ is the indicator function outputing 1 if the inside condition holds and 0 otherwise.

Plug the above derivative to the backpropagation rule, we have

$$\begin{aligned}\frac{\partial L}{\partial Y^1} &= \frac{\partial L}{\partial H^1} \odot \nabla f(Y^1) \\ &= \boxed{\frac{\partial L}{\partial H^1} \odot 1(Y^1 > 0)}\end{aligned}$$

The result tells us that only the neurons getting activated during the forward pass (those with positive linear output Y^1) are being backpropagated with nonzero gradients, while others will not be backpropagated at all.

Implementation

In this section we perform a complete foward and backward pass by implementing a small neural network in Python using Numpy. For completeness, we also include the parameter update step at the end. One whole round of forward, backward pass and parameter update is called an "epoch" of training. Since the numerics are all randomly generated, we don't include them in the post. If interested, I uploaded a Jupyter notebook file to my github; you are welcome to check it out.

Define Network

As above, this will be a two-layered neural network. X will represent a mini-batch of data of size 2 with data dimension 3. The hidden layer contains 4 neurons and the output layer contains 2 neurons (hence the output is two-dimensional). The network looks like the following and we have matrices being the following shape:

$$X \longrightarrow \left[XW^1 + b^1 = Y^1 \rightarrow f(Y^1) = H^1\right]_1 \longrightarrow \left[H^1W^2 + b^2 = \hat{y}\right]_2 \longrightarrow L$$

$$L = \frac{1}{N}\sum_{i,j}(\hat{y}_{ij} - y_{ij})^2$$

$X : 2 \times 3$
$W^1 : 3 \times 4$
$b^1 : 1 \times 4$
$W^2 : 4 \times 2$
$b^2 : 1 \times 2$

Here we simply initialize the weight randomly from a standard normal distribution and set the biases to zero. More sophiscated methods are possible (check out the CS231n class note) but this will do for now.

```
N = 2  # num of training examples in a mini-batch
D = 3  # data dimension
K = 2  # output dimension

# Model parameters
W1 = np.random.randn(D, 4)
b1 = np.zeros((1, 4))
W2 = np.random.randn(4, K)
b2 = np.zeros((1, K))
```

Generate Data

We need to generate the training example; both data and the target value. For illustration purpose, we just generate them from a standard normal distribution.

```
X = np.random.randn(N, D)  # data matrix
y = np.random.randn(2, K)  # target matrix
```

Forward Pass and Loss Compute

The forward pass is just a series of matrix multiplication, addition and possibly applying activations.

```
# Hidden layer
Y1 = X.dot(W1) + b1
H1 = np.maximum(Y1, 0)

# Output layer
y_pred = H1.dot(W2) + b2

# Loss function
L = 2 * np.mean(y_pred - y)
```

Backpropagation

Once we have the loss function, we can backpropagate it back through the network.

```
# We add a "d" up front to denote the gradient
# We will backprop in the following order:
#    dy_pred -> dH1 & dW2 & db2 -> dY1 -> dW1 & db1

# Output Layer
dy_pred = 2*(y_pred - y)
dH1 = dy_pred.dot(W2.T)
dW2 = H1.T.dot(dy_pred)
db2 = dy_pred.sum(axis=0)

# Hidden Layer
dY1 = dH1 * (H1 > 0)  # BP thru relu
dW1 = X.T.dot(dY1);
db1 = dY1.sum(axis=0)
```

Parameter Update

To complete the learning process, we update the model parameters.

```
# Parameter update
lr = 1e-6  # learning rate

W1 -= lr * dW1
b1 -= lr * db1
W2 -= lr * dW2
b2 -= lr * db2
```

The above code is for one epoch of training in the last section. The training procedure often takes thousands of (if not more) epochs to complete; usually a simple for-loop will do the job.

Conclusion

The post illustrates the forward and backward pass of a two-layered feedforward neural network. Generalizing to deeper neural networks is easy: simply repeat the procedure for the hidden layer. After reading this post, you may wonder how this process really looks like in practice in training neural networks to do various things. My other post on training a neural net to classify a 5-spiral dataset is a good place to start since the code is practically the same as above. I intended to do so to demonstrate the usefulness of the implementation in this post.