Backpropagate a Two-Layered Neural Network from Scratch

This post is a note taken after reading the Stanford CS231n class note and one of its supplement material. The former provides theory and program implementation of the foward and backward pass from scratch but touches little on how the derivatives are actually computed. The latter provides a small (but concrete) example of how to backpropagate a linear layer of NN without explicit considering the bias term. Neither of the above derive formulas for backpropagating through general activation functions.

In this post we will derive backpropagation rules for a two-layered neural networks. We isolate the bias term and apply activation functions to aid the completeness for BP rules. Additional math notations originally not included in the class note will be introduced to simply notations. Although it's a toy example, it will prove useful when we go deeper to derive formulas for backpropagating deep neural networks.

Shorthand and Notations.

NN: neural network

BP: backpropagation

$\doteq$: to denote "defined as"

Backpropagate a Two-Layered Neural Network from Scratch

- ***Network***
- ***Backpropagation: The Theory***
  - *Gradients*
  - *Hardamard Product*
  - *Matrix Dot Product*
  - *Algorithm*
  - Special Cases: ReLU activation and MSE loss function
- ***Implementation***
  - *Define Network*
  - *Generate Data*
  - *Forward Pass and Loss Compute*
  - *Backpropagation*
  - *Parameter Update*

Network

(I know this not the standard computational graph laguage nor the usual way of visualizing a neural network, but I haven't figured out a way to plot computational graphs. I hope the illustration below is clear.)

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

It's a two-layered network with the first layer as hidden layer and the second layer as output layer. The big parenthesis' with subsripts indicate the operations associating with the layer.

- X : is the data (design) matrix containing N training examples, each with dimension D. That is, X is of shape $N \times D$, each row of X is a training example. This can be thought of we are training a neural network on *mini-batches* rather than inputting one training exmaple at a time. Note that X can be thought of an extra "input layer". But when talking about the number of layers of a neural network, input layer is usually not counted as a layer.
- The first (hidden) layer does the following things:
  1. Takes the data matrix X, multiply the weight matrix W^1, add the bias vector 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 does the following things (pretty much the same as the first layer):
  1. Take the output of the first layer H^1, multiply by its weight W^2, add the bias term b^2
  2. Note, for output layers, we usually use raw outputs without applying activation functions. Hence the final output of the network is $\hat{y} = H^1W^2 + b^2$.
- L is the loss function. It's a scalar-valued differentiable (w.r.t to the output of the network) function that we want to compute the gradient on. Once we have the gradient of the loss function w.r.t. model parameters, we know how to adjust them to (in this case) decrease the loss (hence, *gradient descent*).

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

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

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

Note now the gradient becomes a 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 used frequently in the chain rule when deriving BP rules.

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 due to the fact that we don't know the exactly 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

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

The purpose of computing the gradient w.r.t. $\boldsymbol{H^1}$ is to enable the gradient to keep flowing backward through the neural network. Let's look at them term by term. First, consider the term $\frac{\partial L}{\partial H^1}$. Intuitively, since we know the relation between $\boldsymbol{H^1}$ and $\hat{\boldsymbol{y}}$ (which is just an affine function), we want to apply the 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 above equation. On the right-hand side, $\frac{\partial L}{\partial \hat{y}}$ is a matrix of the same shape as $\hat{\boldsymbol{y}}$. However, the term $\frac{\partial \hat{y}}{\partial H^1}$, according to our definition above, is a matrix of shape $|\hat{\boldsymbol{y}}| \times |\boldsymbol{H^1}|$. Hence, how can two matrices of two different shapes multiply? To answer this question, we examine the equation for one element of $\boldsymbol{H^1}$:

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

The above equality holds by applying chain rule to one element of $\boldsymbol{H^1}$. Oberserve that we are considering the effect of each element of $\hat{\boldsymbol{y}}$ on $\boldsymbol{H^1_{ij}}$ and the sum up all the effect. Note that the operation is just the *dot product* of the two matrices of shape as $\hat{\boldsymbol{y}}$. Now that we have an expression for each term of $\boldsymbol{H_1}$, we know

$$\frac{\partial L}{\partial H^1} = \left(\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial H^1_{ij}} \right)_{ij}$$

That is, what the chain rule really does is "broadcast" the term $\frac{\partial L}{\partial \hat{y}}$ to each element of $\frac{\partial \hat{y}}{\partial H^1}$ with matrix dot product. Now we are clear about what the chain rule really does, for brevity, we will leave the notation to be

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

But one should bare in mind the true meaning of the notation. In the remainder of the post, we will adapt this notation if not state otherwise.

After equipping ourselves with the chain rule, we are ready to derive the formula for $\frac{\partial L}{\partial H^1}$. Again, we look at each element of the derivative

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

Hence,

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

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

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

Therefore,

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

The term $\frac{\partial L}{\partial b^2}$ is a bit different. Again, use the same trick by examing one element at a time (note that ther is only one subscipt for b^2 since it's 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 $\frac{\partial L}{\partial \hat{y}}$, hence collapse the matrix, reducing the dimension from $N \times \_\_$ to $1 \times \_\_$. We write the derivative as

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

Note. the reason for writing a 0 under the summation is that column-wise summing the matrix is equivalent of summing the matrix along *axis 0* in Numpy.

After all the work, we finally finishing backpropagating through the output layer. But we are almost done! Since the first layer is almost exactly as the second layer — except for an additional activation function we need to backpropagate through. Hence once we obtain the derivative $\frac{\partial L}{\partial Y^1}$, copying steps we have taken above, all other gradients can be otained in a similar way. Again,

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

Note in the last equality, the second term is the derivative of the activation function f on each element of Y_{ij}. Hence 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 the backpropagation 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. Note that we also backpropagated the gradient on $\boldsymbol{X}$, but as for learning purpose, it's not necessary to do so since there is no way we can "update" the data to decrease the loss.

Special Cases: ReLU activation and MSE loss function

Here we further derive the two explicit derivatives for the Rectified Linear Unit (ReLU) activation and the mean squared error (MSE) loss function. They are one of the most common activations and loss function and will serve as our activation and loss function later in the numerical example.

The MSE loss function is defined as

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

which is the mean squared 2-norm of the difference between predicted value and the target value. Since it's a differentiable function of the $\hat{y}$, we can differentiate it directly

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

The ReLU activation function is defined as

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

hence a "rectified" linear function. If we plug in a matrix into f, we simply apply f element-wise to the matrix. The derivative of f is easy to compute

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

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

Plug the above derivative to the backpropagation rule, we have

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

The result tells us that only the neurons getting activated during the forward pass (those with positive activations) are backpropagated with nonzero gradients.

Implementation

In this section we compute a complete foward and backward pass by implementing a small neural network with Python. For completeness, also include the parameter update step at the end. A whole round of forward, backward pass and parameter update is called an "epoch" of training. Since the numerical values here are all randomly generated, we don't include them in the post. Those who are interested are welcome to check out my github, where I put a Jupyter notebook implement of the code below.

Define Network

As above, this will be a two-layered neural network. $\boldsymbol{X}$ will represent a mini-batch of data of size 2, data dimension 3. The hidden layer contains 4 neurons and the output layer contains 2 neurons (hence the output is of dimension 2). In conclusion, the network looks like follows and we have matrices of the following dimensions:

$$X \to \left(Y^1 = XW^1 + b^1 \to H^1 = f(Y^1)\right)_1 \to \left(\hat{y} = H^1W^2 + b^2\right)_2 \to L = \frac{1}{N}\sum_{i,j}(\hat{y}_{ij} - y_{ij})^2$$

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

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

```
N = 2  # # 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 data as well as 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 element-wise 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" in front of each variable 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)
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
```