Neural networks for undergrads

Xiaochuan Yang

May 20 2023

Everyone knows that a neural network is a universal function approximator that can be used to learn complex relationships between inputs and outputs. It is a learning machine that mimics biological neural networks in the human brain. Mathematically, given an input $x \in \mathbb{R}^{n_x}$, a neural nerwork computes an ouput $\hat{y} \in \mathbb{R}^{n_y}$ as follows,

$$\begin{aligned} a^{[1]} &= \sigma(W^{[1]\top} x + b^{[1]}) \\ a^{[2]} &= \sigma(W^{[2]\top} a^{[1]} + b^{[2]}) \\ &\vdots \\ a^{[L]} &= \sigma(W^{[L]\top} a^{[L-1]} + b^{[L]}) \\ \hat{y} &= \sigma(W^{[L+1]\top} a^{[L]} + b^{[L+1]}) \end{aligned}$$

Here $\sigma : \mathbb{R} \to \mathbb{R}$ is any nonlinear differentiable function (or sufficiently close to be such), L is the number of hidden layers, W and b are weight matrices and bias vectors. We use n_l to denote the number of hidden nodes in the l-th layer. The pre-activation vector at layer l is denoted by $z^{[l]}$.

From the definition we see that $\hat{y}$ is not only a function of x, but also a function of $z^{[1]}, z^{[2]}$ and so forth using sub-networks. It may seem like stating the obvious, but it's a useful fact to keep in mind when we derive backpropagation algorithm for training these networks.

1 Training

How to train a neural network to make good predictions? Well, we first need to specify a computable objective. To achieve this we introduce a loss function that measures how good or bad a predictor is compared to the ground truth. This is called supervised learning. A commonly used loss is the squared loss

$$\ell(u, v) = \frac{1}{2}\|u - v\|^2$$

where $\|\cdot\|$ is the Euclidean norm. Then our goal is to minimize $J = \ell(y, \hat{y})$.

A general-purpose optimization method is gradient descent. In every iteration step, this method updates the parameters (i.e. weights and biases) by moving along the opposite

of the gradient of the loss with respect to the parameters. Due to the characterization of the gradient of a function as being the direction along which the function increases the most (infinitesimally speaking), this method is heuristically justified for minimizing an objective function when the step size (aka learning rate) is not too large.

A crucial point is that we need to compute all the partial derivatives for every gradient descent step! Mathematically this is tedious but not hard at all. Everyone knows that to differentiate a composition of functions, the chain rule is our best friend. Sure, we have multiple compositions, but it does not produce any conceptual complications because we can just apply the chain rule multiple times.

Take weight matrix $W^{[1]}$ as an example. Any small nudge on its value would result in changes in $z^{[1]}$, and once we have the value of $z^{[1]}$, we feed it into the sub-network made of layers 1 to $L+1$ and get the output. Hence $J = g(z^{[1]})$ for some g. By the chain rule,

$$\frac{\partial J}{\partial W^{[1]}} = \sum_i \frac{\partial J}{\partial z_i^{[1]}} \frac{\partial z_i^{[1]}}{\partial W^{[1]}}.$$

The second gradient in the summand is the rather simple because $z^{[1]}$ is a linear function of $W^{[1]}$. However, the first gradient is not explicit because the function g is cumbersome as a composition of compositions of compositions ... What we can do is to apply chain rule again, then

$$\frac{\partial J}{\partial z^{[1]}} = \sum_i \frac{\partial J}{\partial z_i^{[2]}} \frac{\partial z_i^{[2]}}{\partial z^{[1]}}$$

Recalling $z^{[2]} = W^{[2]\top}\sigma(z^{[1]}) + b^{[2]}$, the second gradient is easy to calculate. Hence, the gradient of J with respect to the first layer pre-activation is a linear combination of the gradient with respect to the second layer pre-activation. Applying the chain rule recursively in the forward direction all the way to the output layer, we can express $\frac{\partial J}{\partial z^{[1]}}$ as a multiple sum over $\frac{\partial J}{\partial z^{[L+1]}}$ times multiple products of gradients of consecutive pre-activations.

Following the same recipe, we can compute the gradients with respect to weights and biases of all the layers. In summary, we would need for all $l = 1, \ldots, L+1$:

- $\frac{\partial z^{[l]}}{\partial z^{[l-1]}}$
- $\frac{\partial z^{[l]}}{\partial W^{[l]}}$

and chain them together using multiple sums and products. This seems like a lot of work, even for a computer!

Here comes an important observation. There are lots of redundant computations if we use the recipe just described to compute an explicit form for all the gradients at every layer.

The big idea is to take advantage of the recursive relations between gradients with respect to pre-activations of consecutive layers. To be more precise, let's rewrite both equations at a general layer (we also include an equation for the biases).

$$\frac{\partial J}{\partial W^{[l]}} = \sum_i \frac{\partial J}{\partial z_i^{[l]}} \frac{\partial z_i^{[l]}}{\partial W^{[l]}}. \tag{1}$$

$$\frac{\partial J}{\partial b^{[l]}} = \sum_i \frac{\partial J}{\partial z_i^{[l]}} \frac{\partial z_i^{[l]}}{\partial b^{[l]}}. \tag{2}$$

$$\frac{\partial J}{\partial z^{[l]}} = \sum_i \frac{\partial J}{\partial z_i^{[l+1]}} \frac{\partial z_i^{[l+1]}}{\partial z^{[l]}}. \tag{3}$$

Let $S^{[l]} = \frac{\partial J}{\partial z^{[l]}}$. We use equations (1) and (2), along with $S^{[L+1]}$ (easy to compute), to find the required gradients for updating $W^{[L+1]}$ and $b^{[L+1]}$. Then we use equation (3) to find $S^{[L]}$, which can be plugged back into equations (1) and (2) to get the required gradient for updating $W^{[L]}$ and $b^{[L]}$, and so on.

What we just described is the famous backpropagation. The advantage of this approach is that we compute each basic computation (itemized above) only once for each gradient descent iteration.

When L is large i.e. the network is deep, we can easily write a for loop iterating all $l = 1, ..., L$. For each fixed l, however, it is better to avoid for loops for efficiency purposes and write Equations (1)-(3) in matrix forms.

Squared loss Set $A^{[l]} = \text{diag}(\sigma'(z^{[l]}))$ and $e = y - \hat{y}$. It is easy to see that

$$S^{[L+1]} = -A^{[L+1]}e$$

Equation (3) can be written in matrix form as well

$$S^{[l]} = A^{[l]}W^{[l+1]}S^{[l+1]}$$

Now the gradients

$$dW^{[l]} = a^{[l-1]}S^{[l]\top}$$
$$db^{[l]} = S^{[l]}$$

Cross entropy loss The XE loss is defined for two probability mass functions as follows

$$\ell(u, v) = -\sum_k u_k \log v_k.$$

It is particularly well suited for multiclass classification problem. To ensure that $\hat{y}$ is a probability mass function. We use the softmax activation function at the output layer

$$\text{softmax}(x) = \frac{e^x}{\sum_i e^{x_i}}$$

All we have to do is to change the computation of $S^{[L+1]}$, namely the gradient of the XE loss with respect to the output layer pre-activation, then back propagate using the last three equations in the squared loss case. A routine application of chain rule yields that

$$S^{[L+1]} = -e$$

where e was defined earlier in the squared loss case.

2 Implementation

Here is a python implementation of the training with min-batch SGD, 1 hidden layer, sigmoid activation in the hidden and output layers, and squared loss.

```
for epoch in range(n_epoch):
    error = 0
    # permute the data for SGD
    perm_idx = np.random.permutation(m)
    X_train = X_train[:,perm_idx]
    Y_train = Y_train[:,perm_idx]
    for bat in range(n_batches):
        x_batch = X_train[:,bat*batch_size:(1+bat)*batch_size]
        y_batch = Y_train[:,bat*batch_size:(1+bat)*batch_size]
        # forward pass
        z1 = np.matmul(W1.T,x_batch) + b1
        a1 = sig(z1)
        z2 = np.matmul(W2.T,a1) + b2
        a2 = sig(z2)
        # backward pass
        e = y_batch - a2
        A2 = sig(z2)*(1-sig(z2))
        S2 = - A2*e
        A1 = sig(z1)*(1-sig(z1))
        S1 = A1*np.matmul(W2,S2)
        # gradient descent
        dW2 = np.matmul(a1,S2.T)
        db2 = np.sum(S2, axis = 1, keepdims=True)
        dW1 = np.matmul(x_batch,S1.T)
        db1 = np.sum(S1, axis = 1, keepdims=True)
        W2 -= lr * dW2
        b2 -= lr * db2
        W1 -= lr * dW1
        b1 -= lr * db1
        # compute the error
        error += 0.5*np.sum(e*e)
    # report error of the current epoch
    print("Epoch:", epoch, "XE:", error)
    errors.append(error)
```

Exercise: implement the training with XE loss, more hidden layers, and the tricks we will mention in the next section.

3 Tricks

Training neural networks is hard. The backpropagation is the most fundamental technique but we need more tricks to make the training fast and stable. Deep learning researchers have invented many tricks for this purpose, although some of them are rather heuristic and hard to justify mathematically.

Scale the initialisation For wide neural works, using unscaled weight matrix with iid standard normal or symmetric uniform can lead to saturation of nodes of the next layer activation i.e. becoming close or equal to 1 (sigmoid activation function is used in this discussion). To fix this, one can simply scale the weight matrices by a factor of $1/\sqrt{n}$ where n is the number of nodes in the current layer so as to make the variance of the nodes in the next layer pre-activation of order 1.

Activation function ReLU activation, defined by $x \mapsto x\mathbb{1}(x > 0)$, due to its simplicity and efficiency (e.g. its derivative is the indicator of positive half line), becomes popular over the last decade. One of the relatives of ReLU called GeLU given by $x \mapsto x\mathbb{P}[N < x]$ where N is a standard Gaussian is commonly used now in training large models such as GPT and Bert.