Hand-in 2

Michael Iversen
Student ID: 201505099

November 9, 2022

PART I: Derivative

When implementing the neural network, we apply the following loss function for a single data point,

$$L(z) = -\sum_{i=1}^{k} y_i \ln(\text{softmax}(z)_i).$$

Let $y_j = 1$ be the correct label and $y_i = 0$ for $i \neq j$. The loss function can be rewritten as,

$$L(z) = -\ln(\text{softmax}(z)_j).$$

In this section, we determine the derivative of the loss function with respect to all entries of z: $\frac{\partial L}{\partial z_i}$. We determine the derivative by direct calculation.

$$\frac{\partial L}{\partial z_i} = \frac{\partial}{\partial z_i}\left[-\ln\left(\frac{e^{z_j}}{\sum_{a=1}^{k} e^{z_a}}\right)\right]$$

Applying the chain rule and quotient rule, we find,

$$\begin{aligned}
&= -\frac{\sum_{a=1}^{k} e^{z_a}}{e^{z_j}} \cdot \frac{\delta_{ij} e^{z_j}\left(\sum_{a=1}^{k} e^{z_a}\right) - e^{z_j}\left(\sum_{a=1}^{k} \delta_{ia} e^{z_a}\right)}{\left(\sum_{a=1}^{k} e^{z_a}\right)^2} \\
&= -\frac{\sum_{a=1}^{k} e^{z_a}}{e^{z_j}} \cdot \frac{\delta_{ij} e^{z_j}\left(\sum_{a=1}^{k} e^{z_a}\right) - e^{z_j} e^{z_i}}{\left(\sum_{a=1}^{k} e^{z_a}\right)^2} \\
&= -\delta_{ij} + \frac{e^{z_i}}{\sum_{a=1}^{k} e^{z_a}} \\
&= -\delta_{ij} + \text{softmax}(z_i).
\end{aligned}$$

In total, the derivative is given by the simple expression, $\frac{\partial L}{\partial z_i} = -\delta_{ij} + \text{softmax}(z_i)$.

PART II: Implementation and test

Implementation

The implementation of the forward pass and backward pass are shown in the code snippet at the end of this section.

In the forward pass, the mean negative log likelihood is computed and saved as the variable "cost". Next, the weight decay is computed and saved as the variable "weight_decay". Finally, the total loss is computed as the sum of the mean negative log likelihood and the weight decay: "L = cost + weight_decay". The intermediate results are stored since these will be reused in the backward pass. I use the notation that "X1_W1" corresponds to $X_1 \cdot W_2$ and "X1_W1_b1" corresponds to $X_1 \cdot W_1 + b_1$. To simplify notation, I denote $\text{relu}(X_1 \cdot W_1 + b_1)$ by "X2". Using this notation, the quantity $\text{relu}(X_1 \cdot W_1 + b_1) \cdot W_2$ is identified with "X2_W2". I also use the notation "z" for $\text{relu}(X_1 \cdot W_1 + b_1) \cdot W_2 + b_2$. I use the variables "W1_squared", "W2_squared" and "W1_W2_sum" to store intermediate values when computing the weight decay.

In the backward pass, I use the notation "d_x" to signify the Jacobian matrix of the cost function with respect to the variable "x". For instance, the variable "d_z" corresponds to $\frac{\partial L}{\partial \vec{z}}$, "d_b2" corresponds to $\frac{\partial L}{\partial \vec{b_2}}$, etc. The backpropagation begins by using the expression from PART I to compute "d_z". Next, the Jacobian of the loss function with respect to W_1, b_1, W_2 and b_2 may be computed by iteratively performing backpropgation. I illustrate this procedure by going through a few steps. First, the Jacobian matrix with respect to "X2_W2" is identical to the Jacobian matrix with respect to "z" because "z" is simply the sum of "X2_W2" and "b2". Similarly, "d_b2" is determined by summing "d_z" over the rows because "b2" is added to each data point in the computational graph. Next, "d_X2" are computed using the following formula "d_X2 = d_X2_W2 @ W2.transpose()" (taken from https://cs231n.github.io/optimization-2/). The remaining quantities are calculated using similar methods.

```
@staticmethod
def cost_grad(X, y, params, c=0.0):
        W1 = params['W1']
        b1 = params['b1']
        W2 = params['W2']
        b2 = params['b2']
        labels = one_in_k_encoding(y, W2.shape[1])

        # ———————— FORWARD PASS ———————— #
        # Cost
        X1_W1 = X @ W1
        X1_W1_b1 = X1_W1 + b1
        X2 = relu(X1_W1_b1)
        X2_W2 = X2 @ W2
        z = X2_W2 + b2
        cost = — np.sum(labels * np.log(softmax(z)))

        # Weight decay
        W1_squared = W1 ** 2
        W2_squared = W2 ** 2
        W1_W2_sum = np.sum(W1_squared) + np.sum(W2_squared)
        weight_decay = c * W1_W2_sum

        # Total loss
        L = cost + weight_decay

        # ———————— BACKWARD PASS ———————— #
        # Cost
        d_z = — labels + softmax(z)
```

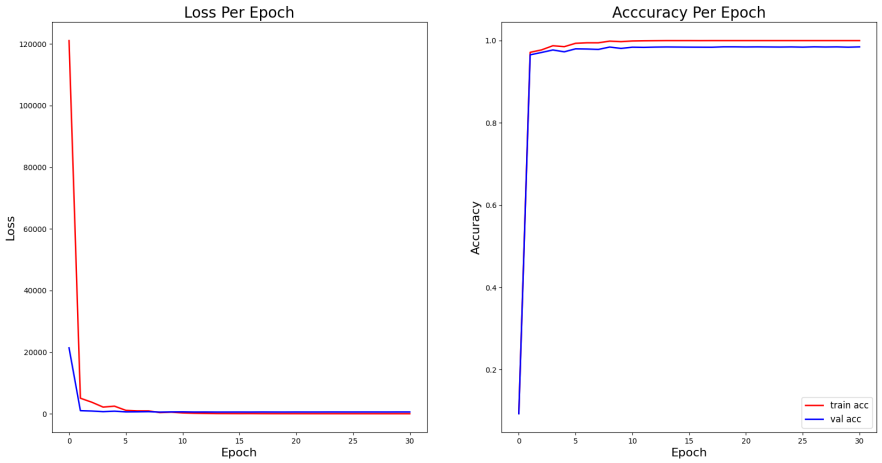

Figure 1: Left: Insample (red) and validation (blue) error as a function of epochs. Right: Insample (red) and validation (blue) accuracy as a function of epochs. The neural network achieves very low loss after a few epochs.

```
d_X2_W2 = d_z
d_b2 = np.sum(d_z, axis=0, keepdims=True)
d_X2 = d_X2_W2 @ W2.transpose()
d_W2 = X2.transpose() @ d_X2_W2
d_X1_W1_b1 = d_X2 * np.heaviside(X1_W1_b1, 0)
d_X1_W1 = d_X1_W1_b1
d_b1 = np.sum(d_X1_W1_b1, axis=0, keepdims=True)
d_W1 = X.transpose() @ d_X1_W1

# Weight decay
d_W1_weight = c * 2 * W1
d_W2_weight = c * 2 * W2

d_w1 = d_W1 + d_W1_weight
d_w2 = d_W2 + d_W2_weight
return L, {'d_w1': d_w1, 'd_b1': d_b1, 'd_w2': d_w2, 'd_b2': d_b2}
```

Tests

The model is trained on the MNIST data set using mini batch stochastic gradient descent. The insample accuracy is $\text{Acc}_{\text{in}} = 0.998$ and the validation accuracy is $\text{Acc}_{\text{val}} = 0.983$. Figure 1 illustrates the insample and validation error as a function of epochs in the left plot and insample and validation accuracy as a function of epochs in the right plot. We see that the neural network achieves very high accuracy after just a few epochs.