1 Neural Networks using Numpy

1.1 Helper Functions

1.1.1 Activation Function

The function ***np.maximum()*** *performs an element-wise maximum comparison, which provides is more efficient and simplistic than the other comparison functions such as* ***np.max()*** *and* ***np.amax()****.*

```
def relu(x):
    return np.maximum(0, x)
```

$$\textbf{ReLU}(x) = \max(x, 0)$$

1.1.2 Softmax Function

To avoid overflows while computing exponentials, the maximum of the input array is subtracted from each value before performing the softmax.

```
def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    return np.exp(x) / np.sum(np.exp(x), axis=1, keepdims=True)
```

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

1.1.3 Compute Prediction

The inputs comes first in the matrix multiplication due to the way in which the inputs values are formatted in the matrix: the row vectors represent the input values of one training sample.

```
def compute_layer(x, w, b):
    return np.matmul(x, w) + b
```

$$z = Wx + b$$

1.1.4 Average Cross-Entropy Loss

```
def average_ce(target, prediction):
    return -np.mean(np.sum((target * np.log(prediction)), axis=1))
```

$$\textbf{CE}_{avg} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{(n)} \log\left(p_k^{(n)}\right)$$

1.1.5 Gradient of Average Cross-Entropy Loss

It is important that that gradient computed is of the average cross-entropy loss, and not the normal cross-entropy loss. Although this will not result in major changes, the gradient will be the average of gradient of the cross-entropy loss.

To make the derivation simpler, we will let the output of the softmax for the k^{th} class $\sigma(z)_k$ be represented by p_k, the input of the softmax $\mathbf{o}$ be represented by z, and the average cross-entropy loss function $\mathbf{CE}_{avg}$ be represented by simply L. Furthermore, the derivative can be broken down into constituent parts as such.

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial z} \tag{1}$$

To calculate the full gradient, we must consider all class outputs of the softmax function derived with respect to the logits of every class. The gradient below requires the quotient derivative rule.

$$\begin{aligned}
\frac{\partial p_i}{\partial z_j} &= \frac{\partial}{\partial z_j}\frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \\
&= \begin{cases} \frac{e^{z_i}\left(\sum_{k=1}^{K} e^{z_k}\right) - e^{z_j}e^{z_i}}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2} & \text{if } i = j \\ \frac{0 - e^{z_j}e^{z_i}}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2} & \text{otherwise} \end{cases} \\
&= \begin{cases} \frac{e^{z_i}\left[\left(\sum_{k=1}^{K} e^{z_k}\right) - e^{z_j}\right]}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2} \\ \frac{-e^{z_j}e^{z_i}}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2} \end{cases} \\
&= \begin{cases} \frac{e^{z_i}}{\left(\sum_{k=1}^{K} e^{z_k}\right)}\frac{\left(\sum_{k=1}^{K} e^{z_k}\right) - e^{z_j}}{\left(\sum_{k=1}^{K} e^{z_k}\right)} \\ \frac{-e^{z_j}}{\left(\sum_{k=1}^{K} e^{z_k}\right)}\frac{e^{z_i}}{\left(\sum_{k=1}^{K} e^{z_k}\right)} \end{cases} \\
&= \begin{cases} p_i(1 - p_j) \\ -p_j p_i \end{cases}
\end{aligned} \tag{2}$$

Now we can derive the gradient of the average cross-entropy by substituting the derivation we got above equation 2 into the full gradient through the chain rule we derived from equation 1.

$$
\begin{aligned}
\frac{\partial L}{\partial z} &= \frac{\partial L}{\partial p}\frac{\partial p}{\partial z} \\
&= \frac{\partial}{\partial p}\left[-\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{(n)} \log\left(p_k^{(n)}\right)\right]\frac{\partial p}{\partial z} \\
&= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{(n)}\frac{1}{p_k^{(n)}}\frac{\partial p}{\partial z} \\
&= -\frac{1}{N}\sum_{n=1}^{N}\left[y_i^{(n)}\frac{1}{p_i^{(n)}}\left(p_i^{(n)}(1-p_i^{(n)})\right) + \sum_{k\neq i} y_k^{(n)}\frac{1}{p_k^{(n)}}\left(-p_k^{(n)}p_i^{(n)}\right)\right] \\
&= -\frac{1}{N}\sum_{n=1}^{N}\left[y_i^{(n)}(1-p_i^{(n)}) - \sum_{k\neq i} y_k^{(n)}p_i^{(n)}\right] \\
&= -\frac{1}{N}\sum_{n=1}^{N}\left[y_i^{(n)} - y_i^{(n)}p_i^{(n)} - \sum_{k\neq i} y_k^{(n)}p_i^{(n)}\right] \\
&= -\frac{1}{N}\sum_{n=1}^{N}\left[y_i^{(n)} - p_i^{(n)}\left(y_i^{(n)} + \sum_{k\neq i} y_k^{(n)}\right)\right] \\
&= -\frac{1}{N}\sum_{n=1}^{N}\left[y_i^{(n)} - p_i^{(n)}\sum_{k=1}^{K} y_k^{(n)}\right] \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(p_i^{(n)} - y_i^{(n)}\right) \qquad (3)
\end{aligned}
$$

As we can see from equation 3, the gradient comes to a lovely reduction to the average of the differences between logits of the predictions and the true labels.

```
def grad_ce(target, logits):
    return np.mean(softmax(logits) - target, axis=0, keepdims=True)
```

1.2 Backpropagation Derivation

Since we are taking the gradient with respect to the weights at the output and hidden layer, we will need to incorporate the activations from the input layer $\mathbf{a}_i$, hidden layer $\mathbf{a}_h$, and output layer $\mathbf{a}_o$. The respective values for each unit i is denoted as $a_i^{(i)}$, $a_i^{(h)}$, and $a_i^{(o)}$. For simplicity, the total cross-entropy loss is used both in calculating the gradients and in the `Numpy` implementation.

1.2.1 Output Layer Weights

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}_o} &= \frac{\partial L}{\partial \mathbf{z}_o}\frac{\partial \mathbf{z}_o}{\partial \mathbf{W}_o} \\ &= (\mathbf{a}_o - \mathbf{y})\frac{\partial}{\partial \mathbf{W}_o}\left[\mathbf{W}_o\mathbf{a}_h + \mathbf{b}_o\right] \\ &= \mathbf{a}_h^T(\mathbf{a}_o - \mathbf{y})\end{aligned}$$

1.2.2 Output Layer Biases

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{b}_o} &= \frac{\partial L}{\partial \mathbf{z}_o}\frac{\partial \mathbf{z}_o}{\partial \mathbf{b}_o} \\ &= (\mathbf{a}_o - \mathbf{y})\frac{\partial}{\partial \mathbf{b}_o}\left[\mathbf{W}_o\mathbf{a}_h + \mathbf{b}_o\right] \\ &= \mathbf{1}^T(\mathbf{a}_o - \mathbf{y})\end{aligned}$$

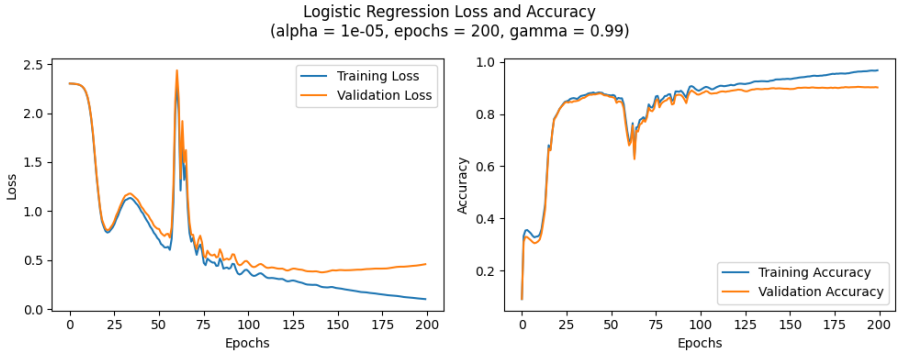
1.2.3 Hidden Layer Weights

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}_h} &= \frac{\partial L}{\partial \mathbf{z}_o}\frac{\partial \mathbf{z}_o}{\partial \mathbf{a}_h}\frac{\partial \mathbf{a}_h}{\partial \mathbf{z}_h}\frac{\partial \mathbf{z}_h}{\partial \mathbf{W}_h} \\ &= \begin{cases}\left(a_i^{(o)} - y_i\right)\left(W_i^{(o)}\right)^T\left(a_i^{(i)}\right) & \text{if } z_i^{(h)} > 0 \\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

1.2.4 Hidden Layer Biases

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}_h} &= \frac{\partial L}{\partial \mathbf{z}_o}\frac{\partial \mathbf{z}_o}{\partial \mathbf{a}_h}\frac{\partial \mathbf{a}_h}{\partial \mathbf{z}_h}\frac{\partial \mathbf{z}_h}{\partial \mathbf{b}_h} \\ &= \begin{cases}\left(a_i^{(o)} - y_i\right)\left(W_i^{(o)}\right)^T & \text{if } z_i^{(h)} > 0 \\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

1.3 Learning



As mentioned in section 1.2, the `Numpy` model was trained using total cross-entropy loss and its gradients for simplicity. The learning rate was scaled down to $\alpha = 10^{-5}$ to adjust for the difference. Despite the large instability spikes between epoch 50 and epoch 75, the network stabilizes afterwards for a final training accuracy of 96.8%, validation accuracy of 90.2%, and testing accuracy of 90.1%.