Osvald Nitski
Will Morley

Assignment2

ECE 421
March 13, 2019

1 Neural Networks using Numpy [14 pts.]

1.1 Helper Functions [4 pts.]

1. ReLU() [0.5 pts.]:

Code Snippet 1: ReLU Helper Function

```
def relu(x):
    #return max(0,x) element-wise on array x
    return x.clip(min=0)
```

2. softmax() [0.5 pts.]:

Code Snippet 2: Softmax Helper Function

```
def softmax(x):
    #vectorized softmax
    # x: (N, C) array of N examples and C classes
    # theta: (N, C) softmax activations for each example/class
    numerator = np.exp(x)
    denominator = np.sum(np.exp(x), axis=1, keepdims=True)
    theta = numerator/denominator
    return theta
```

3. compute() [0.5 pts.]:

Code Snippet 3: Compute Helper Function

```
def computeLayer(X, W, b):
    # X: (N, dim(l-1)) array of N examples
    # W: (dim(l-1), dim(l)) array of weights
    # b: (1, dim(l)) bias term
    # y_hat: (N, dim(l)) array output
    y_hat = np.matmul(X, W) + b
    return y_hat
```

4. averageCE() [0.5 pts.]:

Code Snippet 4: Average Cross Entropy Helper Function

```
def CE(target, prediction):
    # return average cross entropy loss
    # target: (N, K) one-hot encoding of labels
    # prediction (N, K) probability outputs
    N = target.shape[0]
    loss = -np.sum(np.multiply(target, np.log(prediction))) / N
    return loss
```

5. gradCE() [2 pts.]:

Code Snippet 5: Gradient of Cross Entropy Helper Function

```
def gradCE(target, prediction):
    # return total cross entropy loss
    # target: (N, K) one-hot encoding of labels
    # prediction (N, K) probability outputs
    N = len(target)
    grad = np.divide(target, np.log(prediction)).reshape(N, -1, 1)
    return grad
```

Analytic Derivation of Gradient:

$$\begin{aligned}\frac{\partial L}{\partial s_o} &= \frac{\partial}{\partial s_o}\left(\frac{-1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^n log(s_k^n)\right)\\ &= \left(\frac{-1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\frac{\partial\, t_k^n log(s_k^n)}{\partial s_o}\right)\\ &= \left(\frac{-1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\frac{t_k^n}{s_k^n}\right)\end{aligned}$$

1.2 Backpropagation Derivation [4 pts.]

1. $\frac{\partial L}{\partial W_o} = \frac{\partial L}{\partial x_o}\frac{\partial x_o}{\partial z_o}\frac{\partial z_o}{\partial W_o}$

$$\frac{\partial L}{\partial x_o} = \left(\frac{-1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\frac{t_k^n}{s_k^n}\right)$$

$$\frac{\partial x_o}{\partial z_o} = \begin{cases} if\ i=j & \frac{\partial x_{oi}}{\partial z_{oj}} = \frac{e^{zi}\sum_{k=1}^{K}e^{zk} - e^{zi}e^{zj}}{(\sum_{k=1}^{K}e^{zk})^2} = \frac{e^{zi}}{\sum_{k=1}^{K}e^{zk}}\frac{\sum_{k=1}^{K}e^{zk} - e^{zj}}{\sum_{k=1}^{K}e^{zk}} = s_i(1-s_i) \\ if\ i\neq j & \frac{\partial x_{oi}}{\partial z_{oj}} = \frac{0 - e^{zi}e^{zj}}{\sum_{k=1}^{K}e^{zk}} = \frac{e^{zi}}{\sum_{k=1}^{K}e^{zk}}\frac{e^{zj}}{\sum_{k=1}^{K}e^{zk}} = -s_i s_j \end{cases}$$

$$\frac{\partial z_o}{\partial W_o} = \frac{\partial x_h W_o + b_o}{\partial W_o} = x_h$$

$$\implies \frac{\partial L}{\partial W_o} = \left(\frac{-1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\frac{t_k^n}{s_k^n}\right)\begin{cases} s_i(1-s_i) & if\ i=j \\ -s_i s_j & if\ i\neq j \end{cases} x_h$$

$$\implies \frac{\partial L}{\partial W_o} = (p_i - y_i)x_h$$

2. $\dfrac{\partial L}{\partial b_o} = \dfrac{\partial L}{\partial x_o}\ \dfrac{\partial x_o}{\partial z_o}\ \dfrac{\partial z_o}{\partial W_o}$

$$\frac{\partial z_o}{\partial b_o} = \frac{\partial x_h W_o + b_o}{\partial W_o} = 1$$

$$\implies \frac{\partial L}{\partial b_o} = p_i - y_i$$

3. $\dfrac{\partial L}{\partial W_h} = \dfrac{\partial L}{\partial x_o}\ \dfrac{\partial x_o}{\partial z_o}\ \dfrac{\partial z_o}{\partial x_h}\ \dfrac{\partial x_h}{\partial z_h}\ \dfrac{\partial z_h}{\partial W_h}$

$$\frac{\partial z_o}{\partial x_h} = \frac{\partial x_h W_o + b_o}{\partial x_h} = W_o$$

$$\frac{\partial x_h}{\partial z_h} = \begin{cases} 0 & if\ z_h \leq 0 \\ 1 & if\ z_h > 0 \end{cases} = (x_h \geq 0)$$

$$\frac{\partial z_h}{\partial W_h} = \frac{\partial x_{in} W_h + b_h}{\partial W_h} = x_{in}$$

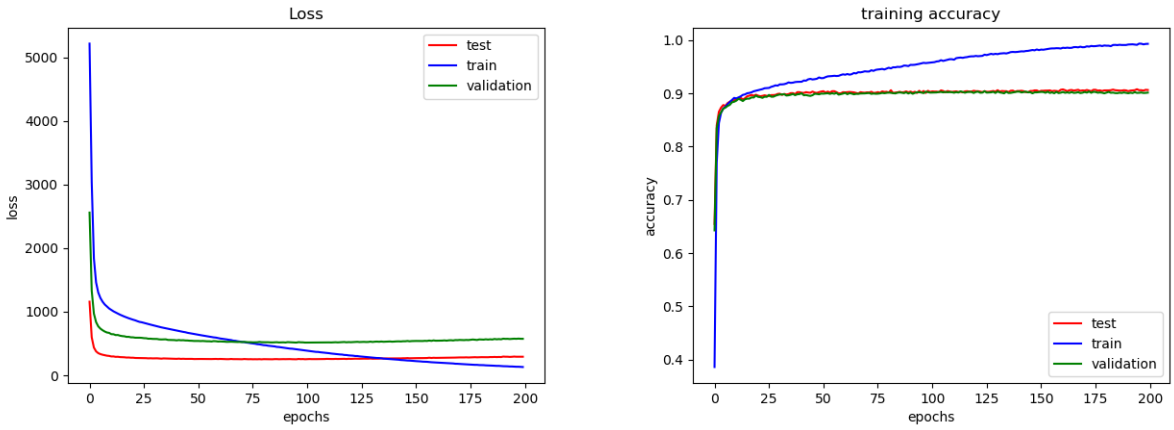
$$\implies \frac{\partial L}{\partial W_h} = (p_i - y_i)\ [W_o]\ (x_h \geq 0)\ x_{in}$$

4. $\dfrac{\partial L}{\partial b_h} = \dfrac{\partial L}{\partial x_o}\ \dfrac{\partial x_o}{\partial z_o}\ \dfrac{\partial z_o}{\partial x_h}\ \dfrac{\partial x_h}{\partial z_h}\ \dfrac{\partial z_h}{\partial b_h}$

$$\frac{\partial z_h}{\partial b_h} = \frac{\partial x_{in} W_h + b_h}{\partial b_h} = 1$$

$$\implies \frac{\partial L}{\partial b_h} = (p_i - y_i)\ [W_o]\ (x_h \geq 0)$$

1.3 Learning [6 pts.]



(a) Loss (b) Accuracy

Figure 1: Training and Validation Loss and Accuracy Curves for Numpy Model

1.4 Hyperparameter Investigation [4 pts.]

1. Number of hidden units [2 pts.]:

Hidden Units	Test Acc. (%)
k = 100	90.7
k = 500	91.3
k = 1000	92.1
k = 2000	91.9

As the number of hidden units increased the training time increased significantly. with 100 hidden units the model ran within an hour while with 2000 hidden units it took multiple hours. Increasing the number of hidden units slightly increased the accuracy up to 1000 units; increasing beyond that decreased test accuracy slightly - likely because the model overfit the training data or perhaps due to a bug in the model.

2. Early stopping [2 pts.]:
From the graph in 1.3 a potential stopping point occurs at 50 epochs at which point

s

validation and test accuracy stop improving
Alternatively, Early stopping could be done at 20 epochs when training accuracy starts to diverge for test/validation accuracy - this option would be appropriate if minimizing training time is a priority.
In the case of overfitting the early stopping point would be at the epoch where validation loss is at a minimum - if it begins to increase after a certain number of epochs the model has begun to overfit to the training data.

2 Neural Networks in Tensorflow [14 pts.]

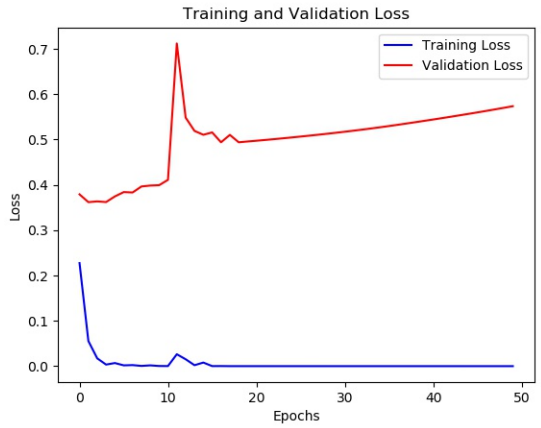
2.1 Model implementation [4 pts.]

Code Snippet 6: CNN model graph

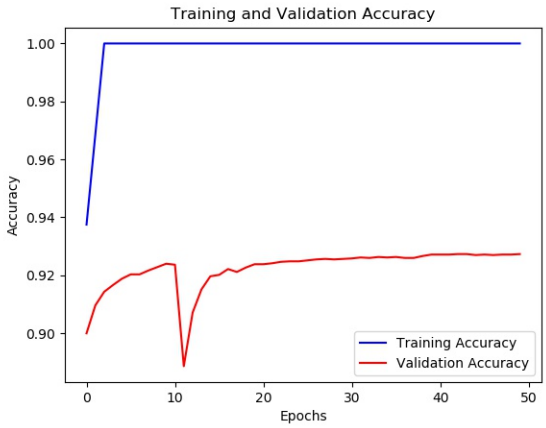
```
def model(data):
    """Construct the CNN model.

    Args:
        data: batch of 28x28 images to evaluate.

    Returns:
        softmax of logits.
    """
    # conv1:
        # 3x3 conv layer, 32 filters, stride of 1
    kernel = tf.get_variable('kernel', shape=[3, 3, 1, 32],
        dtype=tf.float64,
        initializer=tf.contrib.layers.xavier_initializer_conv2d())
    conv = tf.nn.conv2d(data, kernel, [1, 1, 1, 1], 'SAME')
    bias1 = tf.get_variable('bias1', shape=[32], dtype=tf.float64,
        initializer=tf.constant_initializer(0.0))
    conv1 = tf.nn.bias_add(conv, bias1)

    # Relu
    conv1 = tf.nn.relu(conv1)

    # batch normalization
    batch_mean1, batch_var1 = tf.nn.moments(conv1, [0])
    batch1 = tf.nn.batch_normalization(conv1, batch_mean1, batch_var1,
        None, None, 1e-6)

    # 2x2 max pooling
    pool1 = tf.nn.max_pool(batch1, [1, 2, 2, 1], [1, 2, 2, 1], 'SAME')

    # flatten
    reshape = tf.reshape(pool1, [-1,
        pool1.shape[1]*pool1.shape[2]*pool1.shape[3]])

    # fc layer with 784 output units
    fc_weights1 = tf.get_variable('fc_weights1', shape=[reshape.shape[1],
        784], dtype=tf.float64,
        initializer=tf.contrib.layers.xavier_initializer())
    fc_bias1 = tf.get_variable('fc_bias1', shape=[784], dtype=tf.float64,
        initializer=tf.constant_initializer(0.1)) #0.1 init so it doesn't
        die in relu backprop

    # Dropout and Relu Activation
```

```
    fc1 = tf.nn.dropout(tf.matmul(reshape, fc_weights1) + fc_bias1,
        dropout)
    fc1 = tf.nn.relu(fc1)

    # fc layer with 10 output units (1 for each class)
    fc_weights2 = tf.get_variable('fc_weights2', shape=[784, 10],
        dtype=tf.float64,
        initializer=tf.contrib.layers.xavier_initializer())
    fc_bias2 = tf.get_variable('fc_bias2', shape=[10], dtype=tf.float64,
        initializer=tf.constant_initializer(0.0))

    # output
    out = tf.matmul(fc1, fc_weights2) + fc_bias2

    return out, kernel, fc_weights1, fc_weights2
```

2.2 Model Training [4 pts.]



(a) Loss



(b) Accuracy

```
Final test accuracy is  0.9232745961820852
```

(c) Test Accuracy

Figure 2: Training and Validation Loss and Accuracy Curves for Tensorflow Model

2.3 Hyperparameter Investigation [6 pts.]

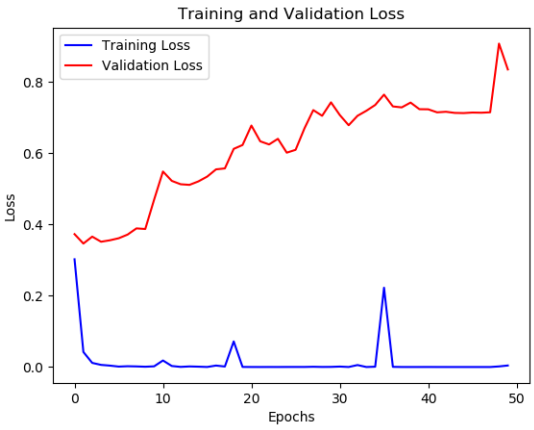
1. L2 Normalization [3 pts.]:

Weight Decay Coefficient	Training Acc. (%)	Validation Acc. (%)	Test Acc. (%)
$\lambda = 0.01$	100	92.21	91.45
$\lambda = 0.1$	93.75	92.48	92.33
$\lambda = 0.5$	84.56	89.18	89.39

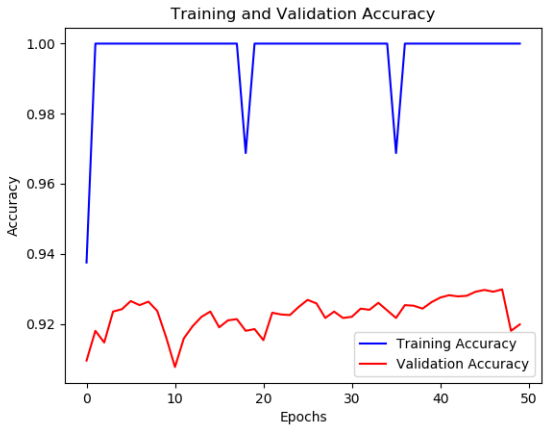
What is impact of L2 regularization on the final test and validation accuracies, if any? There is very little impact when the weight decay is small. When $\lambda = 0.1$, the test accuracy was nearly identical while for $\lambda = 0.01$, the test accuracy was only slightly smaller (approx. 1%). The effect was noticeable only when lambda got large with a decrease of approximately 3% when $\lambda = 0.5$.

2. Dropout [3 pts.]:

Dropout Probability	Training Acc. (%)	Validation Acc. (%)	Test Acc. (%)
p = 0.9	100	91.97	91.41
p = 0.75	100	92.91	92.29
p = 0.5	100	93.13	92.55



(a) Loss

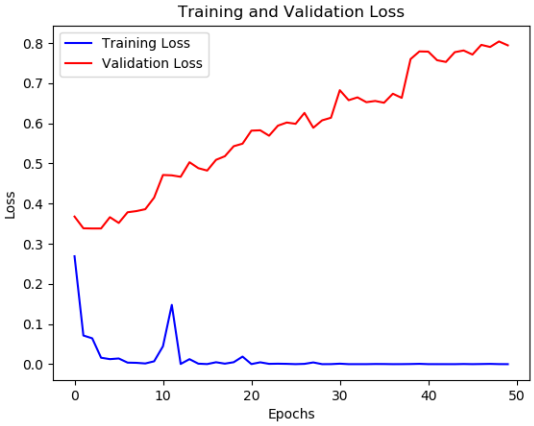


(b) Accuracy

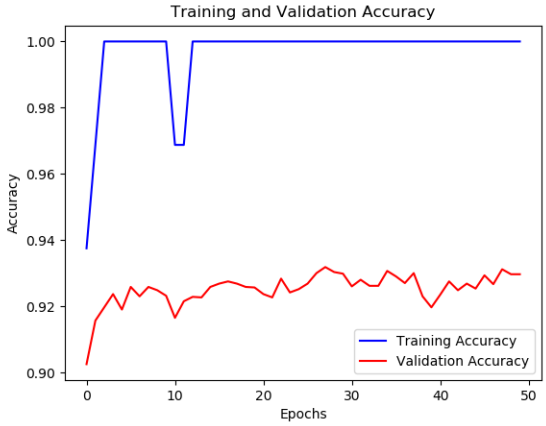
```
Final test accuracy is  0.9140969162995595
```

(c) Test Accuracy

Figure 3: Training and Validation Loss and Accuracy Curves with dropout p = 0.9



(a) Loss

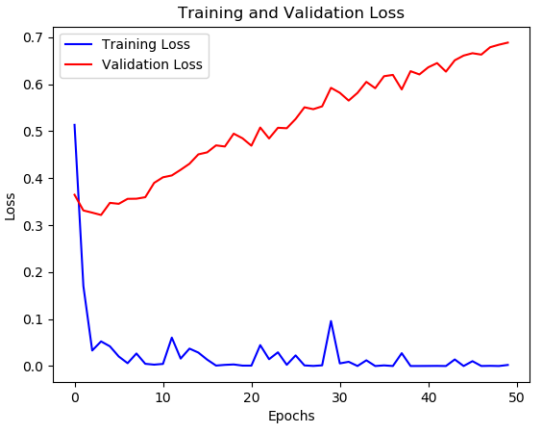


(b) Accuracy

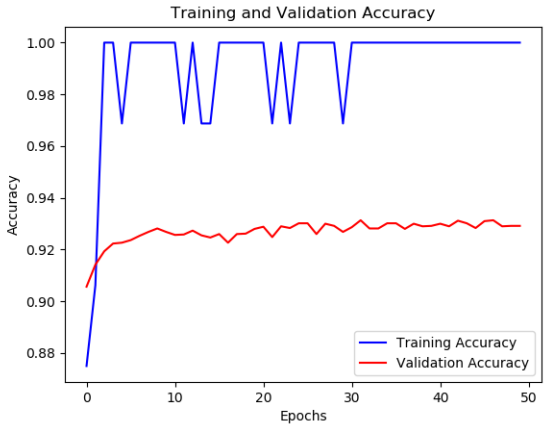
```
Final test accuracy is  0.9229074889867841
```

(c) Test Accuracy

Figure 4: Training and Validation Loss and Accuracy Curves with dropout p = 0.75



(a) Loss



(b) Accuracy

```
Final test accuracy is  0.9254772393538914
```

(c) Test Accuracy

Figure 5: Training and Validation Loss and Accuracy Curves with dropout p = 0.5