ECE1513 Winter 2019 Assignment 2

April 10, 2019

1 Neural Networks using Numpy

In this part, we implement a neural network to classify the letters using Numpy and Gradient Descent with Momentum. The neural network has three layers, including 1 input layer, 1 hidden layer with ReLU activation function and 1 output layer with Softmax function. The loss function used is the Cross Entropy Loss.

1.1 Helper Functions

The following helper functions are needed for the implementation of the neural network. Since all the functions need to handle vector / matrix inputs in practice, we implemented the functions in vectorized form. Also a snippet of Python code is provided below for each of the functions.

1. ReLU(). This function accepts one argument and returns a Numpy array with ReLU activation. For scalar ReLU is simple: $ReLU(x) = \max(x, 0)$. The matrix form involves the comparison between $\mathbf{x}$ and $\mathbf{0}$, i.e. $ReLU(\mathbf{X}) = \max(\mathbf{X}, \mathbf{0})$, where $\mathbf{0}$ is zero matrix or vector with the same shape as $\mathbf{X}$.

```
def relu(x):
    """ReLU function.
    :param x: A vector of scalar numbers.
    """
    return np.maximum(x, np.zeros(x.shape))
```

2. softmax(). This function accepts one argument and returns a Numpy array with softmax activations of each of the inputs. The equation form given below can cause overflow when z_k is large enough.

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

, for j = 1..K, for K classes.

To avoid overflow, we can transform the equation into a new form by multiplying it with a constant C.

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

In practice, we use $C = e^{-z_{\max}} = e^{-\max \mathbf{z}}$, thus

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

To handle multiple data points, we need the matrix form of the softmax function.

$$\sigma(Z) = \begin{bmatrix} e^{z_{1,1}}/\Sigma_1 & e^{z_{1,2}}/\Sigma_1 & \cdots & e^{z_{1,K}}/\Sigma_1 \\ e^{z_{2,1}}/\Sigma_2 & e^{z_{2,2}}/\Sigma_2 & \cdots & e^{z_{2,K}}/\Sigma_2 \\ \vdots & \vdots & \ddots & \vdots \\ e^{z_{N,1}}/\Sigma_N & e^{z_{N,2}}/\Sigma_N & \cdots & e^{z_{N,K}}/\Sigma_N \end{bmatrix}$$

where Σ_n is the row-wise summation, i.e. $\Sigma_n = \sum_{k=1}^{K} e^{z_{n,k}}$.

```
def softmax(x):
    """softmax function.
    :param x: A 2-D vector, shape: (N, K), N = # of data points
    """
    e = np.exp(x - np.max(x, axis=1)[:,None])
    s = np.sum(e, axis=1)[:,None]    # row-wise sum
    return e / s                     # Divide each elem with corresponding row
```

3. compute(). This function accepts three arguments, a weight, an input and a bias matrix. It returns the product of the weights and input, and then plus the bias, which is simply $X^T W + b$.

```
def computeLayer(X, W, b):
    return np.matmul(X, W) + b
```

4. averageCE(). This function accepts two parameters, the target labels and the predictions respectively. It returns the cross entropy loss as a number. The cross entropy loss function for a single example in the case of K different classification classes can be denoted as

$$H(\mathbf{t}, \mathbf{s}) = \sum_{k=1}^{K} \big(- t_k \log s_k \big) = - \log \mathbf{s} \cdot \mathbf{t}$$

, where $\mathbf{t}$ is the target matrix, and $\mathbf{s}$ is the prediction matrix, which contains the softmax scores of the example.

The averageCE function computes the average of the cross entropy loss across all examples, thus divides the sum of $H(\mathbf{t}, \mathbf{s})$ by the number of examples N.

```python
def CE(target, prediction):
    """Cross entropy loss.
    CE = -1/N sum{n=1..N} sum{k=1..K} t_k^(n) * log(s_k^(n))
    :param target: N * K
    :param prediction: N * K
    """
    N = target.shape[0]
    return -np.sum(target * np.log(prediction)) / N
```

5. gradCE(). This function accepts two arguments, the target labels and the predictions. It returns the gradient of the cross entropy loss with respect to the softmax scores. The analytical expression for the gradient of a single example can be expressed as:

$$\frac{\partial H(s,t)}{\partial s_j} = \frac{\partial}{\partial s_j}\sum_{k=1}^{K}(-t_k \log s_k) = -\mathbf{t}/\mathbf{s}_j. \tag{1}$$

The gradCE function computes the average gradient across all examples, which is thus $-\frac{1}{N}\sum_n(\mathbf{t}^n/\mathbf{s}^n)$.

```python
def gradCE(target, prediction):
    """Gradient of CE.
    """
    return -np.mean(target / prediction, axis=0)
```

1.2 Back propagation Derivation

In this part, we derive analytical expressions for quantities necessary for the implementation of the back propagation algorithm in order to train the neural network. For convenience, we compute the derivatives of the softmax and the cross entropy loss with softmax before computing the derivations of the back propagation algorithm.

First, we compute the derivative of the softmax function. To simplify the notation, we denote $\sum_{k=1}^{K} e^{z_k}$ as Σ.

$$\begin{aligned}\frac{\partial \sigma(z_j)}{\partial z_i} = \frac{\partial}{\partial z_i}\frac{e^{z_j}}{\Sigma} &= \Big(\frac{\partial}{\partial z_i}e^{z_j}\cdot\Sigma - e^{z_j}\frac{\partial}{\partial z_i}\Sigma\Big)/\Sigma^2 \\ &= \Big(\frac{\partial}{\partial z_i}e^{z_j}\cdot\Sigma - e^{z_j}e^{z_i}\Big)/\Sigma^2.\end{aligned}$$

When $i = j$,

$$\frac{\partial \sigma(z_j)}{\partial z_i} = \big(\frac{\partial e^{z_j}}{\partial z_j}\cdot\Sigma - e^{z_j}e^{z_j}\big)/\Sigma^2 = e^{z_j}/\Sigma - (e^{z_j}/\Sigma)^2 = \sigma(z_j)(1-\sigma(z_j)).$$

When $i \neq j$,

$$\frac{\partial \sigma(z_j)}{\partial z_i} = \Big(\frac{\partial}{\partial z_i}e^{z_j}\cdot\Sigma - e^{z_j}e^{z_i}\Big)/\Sigma^2 = -(e^{z_j}/\Sigma)(e^{z_i}/\Sigma) = -\sigma(z_i)\sigma(z_j).$$

Thus,

$$\frac{\partial \sigma(z_i)}{\partial z_j} = \begin{cases} \sigma(z_j)(1-\sigma(z_j)) & , i = j, \\ -\sigma(z_i)\sigma(z_j) & , i \neq j. \end{cases} \tag{2}$$

Next, we express the cross entropy loss with softmax $H(t, \sigma(z))$ as a function of z directly $H_s(t, z)$. Then

$$H_s(t, z) = \sum_{k=1}^{K} \big(- t_k \log \sigma(z_k) \big)$$

From **Equation (1)** and **(2)** we have

$$\begin{aligned} \frac{\partial H_s(\mathbf{t}, \mathbf{z})}{\partial z_j} &= \frac{\partial}{\partial z_j} \sum_{k=1}^{K} \big(- t_k \log \sigma(z_k) \big) \\ &= -\frac{t_j}{\sigma(z_j)} \frac{\partial \sigma(z_j)}{\partial z_j} - \sum_{k \neq j} \frac{t_k}{\sigma(z_k)} \frac{\partial \sigma(z_k)}{\partial z_j} \\ &= -\frac{t_j}{\sigma(z_j)} \sigma(z_j)(1-\sigma(z_j)) - \sum_{k \neq j} \frac{t_k}{\sigma(z_k)} \big(- \sigma(z_k)\sigma(z_j) \big) \\ &= -t_j + t_j \sigma(z_j) + \sum_{k \neq j} t_k \sigma(z_j) \\ &= \sigma(z_j) \sum_k t_k - t_j. \end{aligned}$$

Note that $\mathbf{t}$ is one-hot encoded, which means $\sum_k t_k = 1$. Therefore,

$$\frac{\partial H_s(\mathbf{t}, \mathbf{z})}{\partial z_j} = \sigma(z_j) - t_j. \tag{3}$$

Now we continue to solve the derivations required by the back propagation algorithm. Here we use superscripts $^{(o)}$,$^{(h)}$, and $^{(i)}$ to indicate variables in the output layer, the hidden layer, and the input layer respectively. See **Figure 1** for the network architecture and the variable naming convention.

The gradient of the loss with respect to the outer layer weights and biases can be found by:

$$\begin{aligned} \frac{\partial L}{\partial W_{i,j}^{(o)}} &= \frac{\partial L}{\partial s_j^{(o)}} \cdot \frac{\partial s_j^{(o)}}{\partial W_{i,j}^{(o)}} = \frac{\partial L}{\partial s_j^{(o)}} \cdot x_i^{(h)}, \\ \frac{\partial L}{\partial b_j^{(o)}} &= \frac{\partial L}{\partial s_j^{(o)}} \cdot \frac{\partial s_j^{(o)}}{\partial b_j} = \frac{\partial L}{\partial s_j^{(o)}}. \end{aligned}$$

Denote $\frac{\partial L}{\partial s_j^{(o)}}$ as $\delta_j^{(o)}$, we have

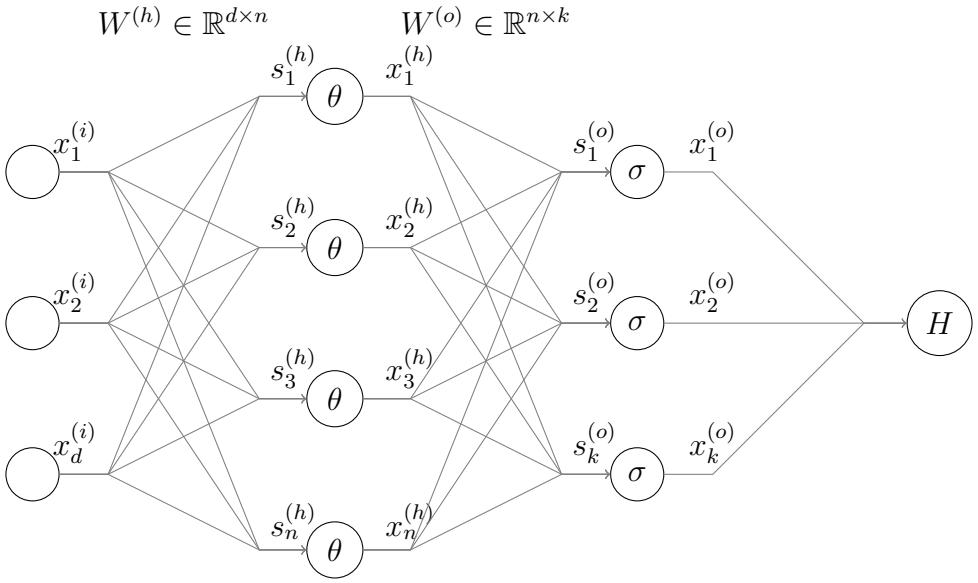

Figure 1: Architecture of the Neural Network.

$$\frac{\partial L}{\partial W_{i,j}^{(o)}} = \delta_j^{(o)} x_i^{(h)}, \tag{4}$$

$$\frac{\partial L}{\partial b_j^{(o)}} = \delta_j^{(o)}. \tag{5}$$

where $\delta_j^{(o)} = x_j^{(o)} - y_j$ (from **Equation (3)**).

The matrix form of **Equation (4)** and **Equation (5)** is:

$$\frac{\partial L}{\partial \mathbf{W}^{(o)}} = \mathbf{x}^{(h)} \delta^{(o)}, \tag{6}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(o)}} = \delta^{(o)}. \tag{7}$$

where

$$\delta^o = \mathbf{x}^{(o)} - \mathbf{y}. \tag{8}$$

Next we solve for the derivatives for the hidden layer. The gradient of the loss with respect to the hidden layer weights can be found by:

$$\begin{aligned}\frac{\partial L}{\partial W_{i,j}^{(h)}} &= \frac{\partial L}{\partial s_j^{(h)}} \cdot \frac{\partial s_j^{(h)}}{\partial w_{i,j}^{(h)}} = \frac{\partial L}{\partial s_j^{(h)}} \cdot x_i^{(i)}, \\ &= \frac{\partial L}{\partial x_j^{(h)}} \cdot \frac{\partial x_j^{(h)}}{\partial s_j^{(h)}} \cdot x_i^{(i)} = \frac{\partial L}{\partial x_j^{(h)}} \cdot \theta'(s_j^{(h)}) \cdot x_i^{(i)},\end{aligned}$$

where $\theta(\cdot)$ is the ReLU activation function in the hidden layer, and

$$\theta'(z) = \begin{cases} 1 & , z > 0 \\ 0 & , \text{otherwise} \end{cases}. \tag{9}$$

Since we have

$$\frac{\partial L}{\partial x_j^{(h)}} = \sum_k \Big(\frac{\partial L}{\partial s_k^{(o)}} \cdot \frac{\partial s_k^{(o)}}{\partial x_j^{(h)}} \Big) = \sum_k \Big(\delta_k^{(o)} \cdot W_{j,k}^{(o)} \Big),$$

Let $\delta_j^{(h)}$ denote $\frac{\partial L}{\partial s_j^{(h)}} = \frac{\partial L}{\partial x_j^{(h)}} \cdot \theta'(s_j^{(h)}) = \sum_k \left(\delta_k^{(o)} \cdot W_{j,k}^{(o)} \right) \cdot \theta'(s_j^{(h)})$, then

$$\frac{\partial L}{\partial W_{i,j}^{(h)}} = \delta_j^{(h)} x_i^{(i)}. \tag{10}$$

Similarly, the gradient of the loss with respect to the hidden layer bias is

$$\frac{\partial L}{\partial b_j^{(h)}} = \delta_j^{(h)} \frac{\partial s_j^h}{\partial b_j^h} = \delta_j^{(h)}. \tag{11}$$

Equation (10) and **Equation (11)** can be written in matrix form as:

$$\frac{\partial L}{\partial \mathbf{W}^{(h)}} = \mathbf{x}^{(i)} \delta^{(h)}, \tag{12}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(h)}} = \delta^{(h)}, \tag{13}$$

where

$$\delta^{(h)} = \delta^{(o)} \mathbf{W}^{(o)\mathsf{T}} \cdot \theta'(s^h). \tag{14}$$

1.3 Learning

A neural network is constructed and trained according to the required specifications, and the detailed implementation can be referred to in the source code submitted separately. Due to memory constraints, Stochastic Gradient Descent is utilized here instead of Full Gradient Descent. With batch size = 100, the neural network is trained for 200 epochs with a hidden unit size of 1000, the resulting loss and accuracy curves for the training, validation and test data sets are shown in **Figure 2**.

As observed in the figure, the learning process appears to be correctly implemented, since the overall trend of the training loss is monotonically decreasing, corresponding to the training accuracy increasing with increased epochs. The local roughness in the curves can be contributed to the fact that Stochastic Gradient Descent is implemented here. It can also be inferred from the figure that this model starts to overfit the data after around 53 epochs, since the test and the validation losses start to increase even though the test loss continues to decrease.

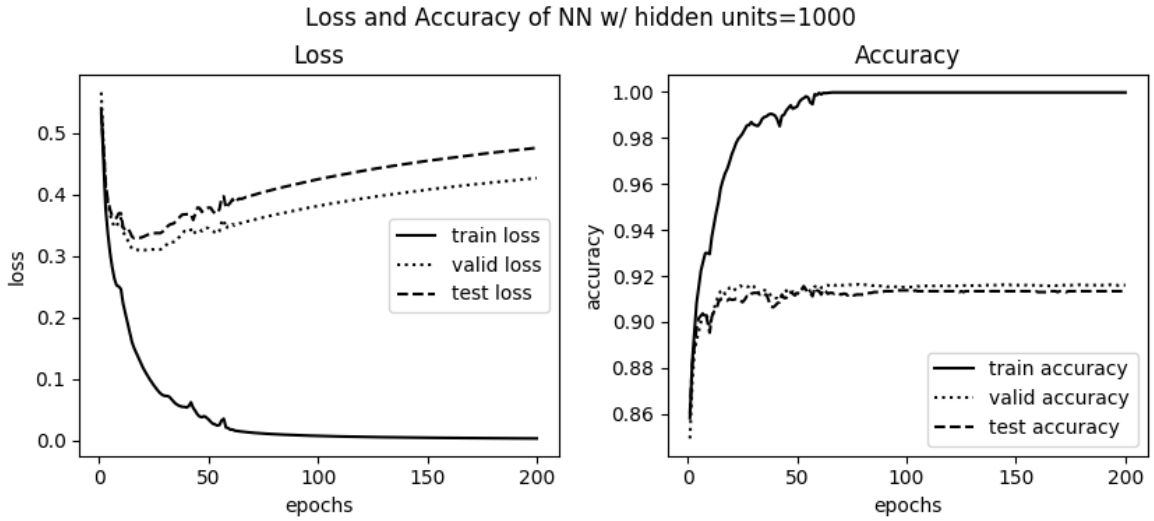

Figure 2: Loss(left) and accuracy(right) curves for training, validation and test data sets, of the neural network with 1000 hidden units trained for 200 epochs. Batch size = 100 and learning rate $\alpha = 1 \times 10^{-5}$ is used for Stochastic Gradient Descent.

1.4 Hyperparameter Investigation

1.4.1 Number of hidden units

To investigate the effects of the number of hidden units on the classification accuracy of neural networks, three different hidden unit sizes [100, 500, 2000] are used in the model instead of 1000 from the previous part. **Figure 3** shows the resulting training and test accuracy curves.

As shown in the figure, larger hidden unit sizes tend to achieve higher accuracy and converge faster in terms of training accuracy. Although after sufficient number of epochs, the final training accuracy that different hidden unit sizes can reach appear to become about the same, indicating the ultimate accuracy the neural network can achieve in fitting the training set.

In the case of test accuracy, larger hidden unit sizes appear to achieve better accuracy. Specifically, the model with 100 hidden units has significantly lower test accuracy than models with 500 and 2000 hidden units, indicating that for this problem, the model with less hidden units does not have enough complexity to well generalize the model to unseen data, i.e. underfitting. Generally speaking, more hidden units allow for higher complexity in the model which may contribute to the model with better generalization to test data.

1.4.2 Early stopping

From **Figure 2** we find that the validation and test accuracies become stabilized after around 50 epochs. The test accuracy reaches its peak at about 53 epochs. After that, both validation loss and test loss start to increase even though the training loss is continuously decreased.

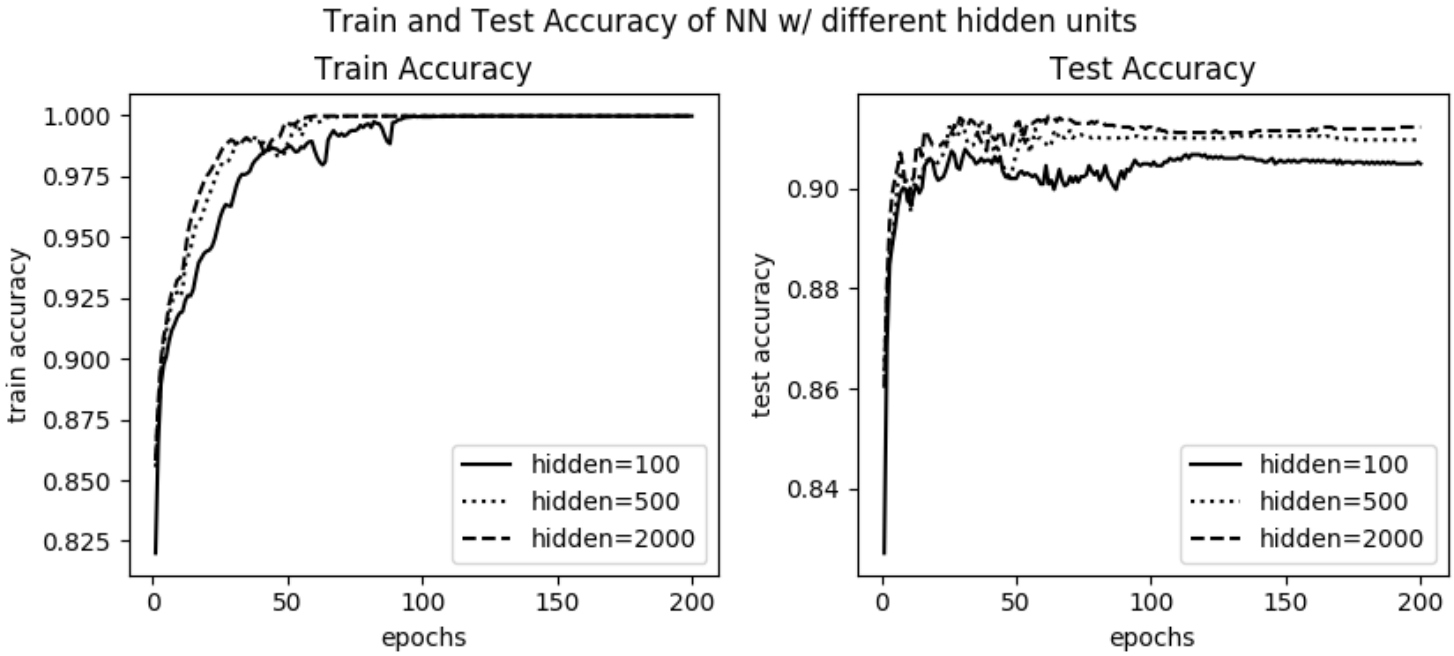

Figure 3: Training(left) and test(right) accuracy curves for neural networks with 100, 500 and 2000 hidden units respectively.

This suggests the sign of overfitting and signifies a reasonable early stopping point can be chosen at epoch 53. The classification accuracy data after epoch 53 is listed below:

- Training Accuracy: 99.73%
- Validation Accuracy: 91.47%
- Test Accuracy: 91.56%

2 Neural Networks in Tensorflow

2.1 Model implementation

In this part, a Convolutional Neural Network is implemented in Tensorflow for image recognition. The network architecture is implemented as required and a snippet of Python code is given below:

```
def build_graph(self, image_size=28, num_of_class=10, conv_params=(3, 32, 1),
                pool_size=2, alpha=1e-4):
  """Build a convolutional neuron network.

  :param image_size: The width/height of the input image. Image must be a square
                     with size of (image_size, image_size).
```

```
:param num_of_class: Number of output classes.
:param conv_params: A 3-element tuple
, specifying (kernel_size, num_of_filters,
                    stride) for conv layer.
:param pool_size: Size of max_pool layer.
:param alpha: Learning rate.
"""
with tf.name_scope('input'):
  # 1. Input layer
  self.X = tf.placeholder(tf.float32, shape=(None, image_size, image_size),
                          name='X')
  X_reshaped = tf.reshape(self.X, shape=(-1, image_size, image_size, 1),
                          name='X_reshaped')

  # Target
  self.y = tf.placeholder(tf.int64, shape=(None,), name='y')
  y_onehot = tf.one_hot(self.y, num_of_class)

with tf.name_scope('conv_pool'):

  # 2. Conv layer
  kernel_size, num_of_filters, stride = conv_params
  W_conv = tf.get_variable('W_conv', dtype=tf.float32,
                           shape=(kernel_size, kernel_size, 1, num_of_filters),
                           initializer=tf.contrib.layers.xavier_initializer())
  b_conv = tf.get_variable('b_conv', shape=(num_of_filters,),
                           initializer=tf.contrib.layers.xavier_initializer())
  conv = tf.nn.conv2d(X_reshaped, W_conv, strides=(1, stride, stride, 1),
                      padding='SAME') + b_conv

  # 3. ReLU activation
  h_conv = tf.nn.relu(conv)

  # 4. Bath normalization
  h_mean, h_var = tf.nn.moments(h_conv, axes=[0])
  batch_norm = tf.nn.batch_normalization(h_conv, h_mean, h_var,
                                         None, None, 1e-9)

  # 5. Max pooling
  h_pool = tf.nn.max_pool(batch_norm, ksize=[1, pool_size, pool_size, 1],
                          strides=[1, pool_size, pool_size, 1],
                          padding='SAME', name='h_pool')

  # 6. Flatten
  feature_count = int(image_size * image_size / pool_size / pool_size *
                      num_of_filters)
  h_flat = tf.reshape(h_pool, shape=[-1, feature_count], name='h_flat')

with tf.name_scope('fc1'):

  # 7. FC1 layer
  fc_size = image_size * image_size
  W_fc1 = tf.get_variable('W_fc1', shape=(feature_count, fc_size),
                          dtype=tf.float32,
```

```
                            initializer=tf.contrib.layers.xavier_initializer())
    b_fc1 = tf.get_variable('b_fc1', shape=(fc_size,), dtype=tf.float32,
                            initializer=tf.contrib.layers.xavier_initializer())
    h_fc1 = tf.matmul(h_flat, W_fc1) + b_fc1

    # Dropout layer
    self.p_dropout = tf.placeholder(tf.float32, shape=(), name='p_dropout')
    h_fc1 = tf.nn.dropout(h_fc1, keep_prob=self.p_dropout)

    # 8. ReLU
    h_fc1 = tf.nn.relu(h_fc1)

  with tf.name_scope('fc2'):
    # 9. FC2 layer
    W_fc2 = tf.get_variable('W_fc2', shape=(fc_size, num_of_class),
                            dtype=tf.float32,
                            initializer=tf.contrib.layers.xavier_initializer())
    b_fc2 = tf.get_variable('b_fc2', shape=(num_of_class,), dtype=tf.float32,
                            initializer=tf.contrib.layers.xavier_initializer())
    h_fc2 = tf.matmul(h_fc1, W_fc2) + b_fc2

  with tf.name_scope('output'):

    # 10. Softmax
    h_softmax = tf.nn.softmax(h_fc2)

  # 11. Cross Entropy Loss
  self.loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(
    labels=y_onehot, logits=h_fc2))

  # L2 regularization
  self.l2_lambda = tf.placeholder(tf.float32, shape=(), name='l2_lambda')
  reg = self.l2_lambda * (tf.nn.l2_loss(W_conv) + tf.nn.l2_loss(W_fc1) +
        tf.nn.l2_loss(W_fc2))
  self.loss = self.loss + reg

  # Optimizer
  self.opt_op = tf.train.AdamOptimizer(learning_rate=alpha).minimize(self.loss)

  # Predict
  self.predict = tf.argmax(h_softmax, 1)
  self.accuracy = tf.reduce_mean(
    tf.cast(tf.equal(self.predict, self.y), tf.float32)
  )
```

2.2 Model Training

The Convolutional Neural Network is trained using SGD for a batch size of 32, 50 epochs, and the learning rate α is set to 10^{-4} in the Adam optimizer. The training, validation and test loss and accuracy curves are plotted in **Figure 4**. As shown in the figure, the training accuracy very quickly converges to nearly 1 after the training commences. The validation and test accuracies appear to stablize after around 20 epochs. After that, validation and

test losses start increasing even though the training loss keeps decreasing, indicating the occurrence of overfitting.

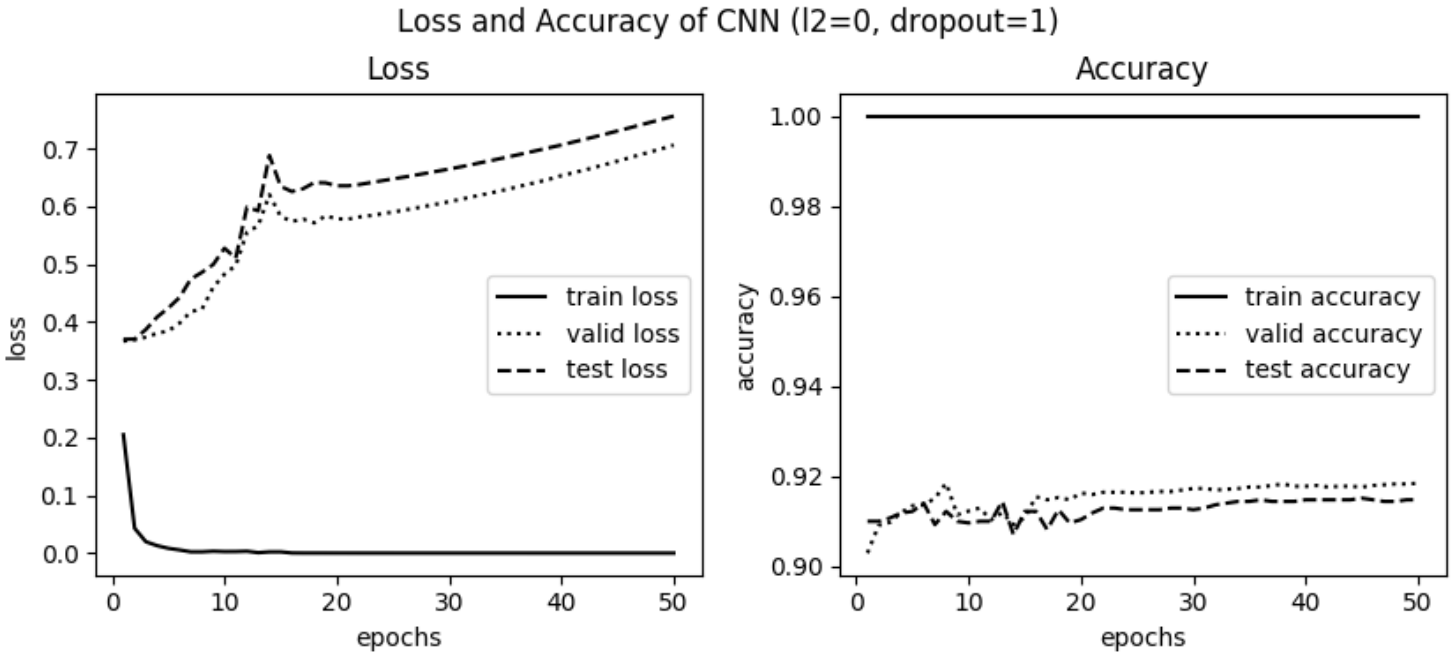

Figure 4: Training, validation and test loss(left) and accuracy(right) curves for the Convolutional Neural Network with no regularization or dropout.

2.3 Hyperparameter Investigation

2.3.1 L2 Normalization

L2 regularization is performed on the weights in the model using three different weight decay coefficients $\lambda = [0.01, 0.1, 0.5]$. The training, validation and test accuracy curves under various λ settings are presented in **Figure 5**. The final accuracy results after 50 epochs are summarized in **Table 1**.

As we can see from the table, $\lambda = 0.1$ achieves the highest final validation and test accuracy when compared to the others, which indicates that it is the better choice out of the three λ values. When $\lambda = 0.5$, the final training, validation and test accuracy scores are significantly lower than the other two cases.

This shows the bias-variance trade off. If λ is too large, it will excessively penalize the weights, resulting in high bias and low variance, leading to under-fitting of the model. Conversely, when $\lambda = 0.01$, the training accuracy reaches its highest 100%, but the validation and test accuracies are lower than those with $\lambda = 0.1$. This is consistent with the fact that small λ can result in low bias and high variance in the model, which may possibly cause overfitting of the model.

weight decay coefficient λ	training accuracy	validation accuracy	test accuracy
0.01	100%	92.37%	92.11%
0.1	93.75%	92.50%	92.69%
0.5	87.5%	89.02%	90.12%

Table 1: The final training, validation and test accuracies after 50 epochs under three different weight decay coefficient settings $\lambda = [0.01, 0.1, 0.5]$.

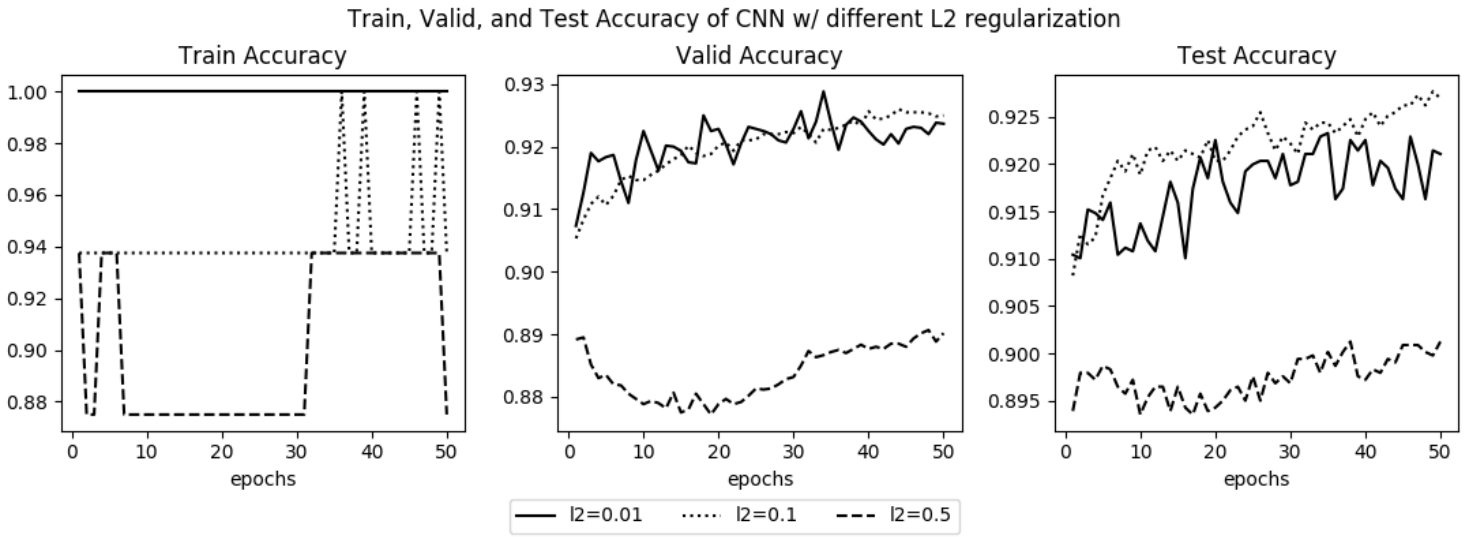

Figure 5: The training, validation and test accuracy curves under three different weight decay coefficient settings $\lambda = [0.01, 0.1, 0.5]$.

2.3.2 Dropout

To investigate the influence of dropout on model performance, three probability values $p = [0.9, 0.75, 0.5]$ are applied to the dropout layer in the model. The training, validation and test accuracy curves are plotted in **Figure 6**. After examining the figure, we can see that higher dropout probability tends to converge faster in terms of training accuracy, while the final training accuracies reached under different probability scenarios are all very close to 1. Meanwhile, $p = 0.5$ generally achieves the best validation and test accuracies among the three, followed by $p = 0.9$, with $p = 0.75$ having the poorest overall validation and test classification performance. This confirms that $p = 0.5$ is the better choice for dropout probability in order to control overfitting and enhance model accuracy performance.

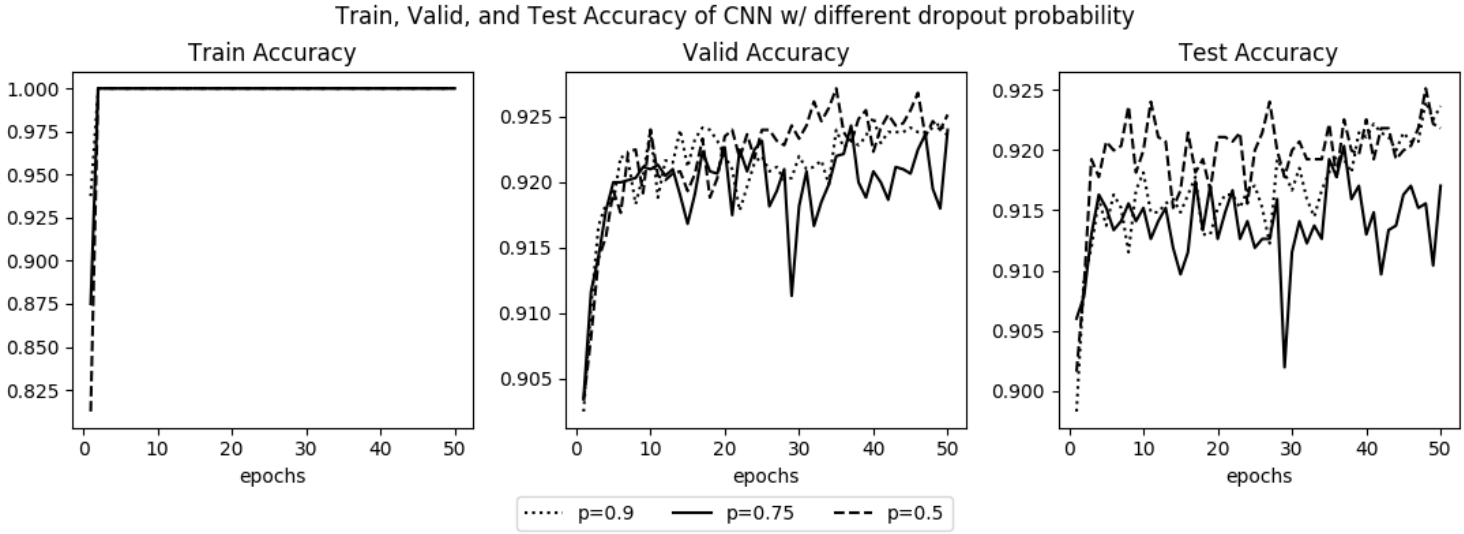

Figure 6: Training, validation and test accuracy curves with three different dropout probabilities $p = [0.9, 0.75, 0.5]$.