1. Neural Networks using Numpy [14 pts]

1.1 Helper Functions [4 pt.]

- ReLU():

```
def relu(x):
    zeros = np.zeros(x.shape)
    return np.maximum(x, zeros)
```

- softmax():

```
def softmax(x):
    softmax = lambda x:np.exp(x)/np.sum(np.exp(x))
    return softmax
```

- compute():

```
def computeLayer(X, W, b):
    result = []
    traning_data = X.flatten()
    result.append(relu(np.dot(np.transpose(W),traning_data) + b))
    return result
```

- averageCE():

```
def CE(target, prediction):
    return -1/(len(prediction)) * np.sum(target*np.log(prediction))
```

- gradCE():

```
def gradCE(target, prediction):
    return -1/(len(prediction)) * np.sum(target/prediction, axis=0)
```

Analytical expression\:

Every hot encode group has only one value of “1” and others are “0”, so the value “0” is the number that we should ignore them.

$$\frac{\partial \text{ average } CE}{\partial S_k^n} = \begin{cases} -\frac{1}{N} \sum_{N=1}^{N} \{t_k^n = 1\} \frac{1}{S_k^n} & \text{for } t_k^n = 1 \\ 0 & \text{for } t_k^n = 0 \end{cases}$$

1.2 Backpropagation Derivation [4 pts.]

- $\dfrac{\partial L}{\partial W_o}$

$$\frac{\partial L}{\partial W_o} = \frac{\partial\left(-\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^n \log\left(\frac{e^{W_o^{kT}x_n+b}}{\sum_{k=1}^{K} e^{W_o^{kT}x_n+b}}\right)\right)}{\partial W_o^i}$$

$$= \frac{\partial\left(\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^n \left(-W_o^{kT}x_n+b\right)+\log\left(\sum_{k=1}^{K} e^{W_o^{kT}x_n+b}\right)\right)}{\partial W_o^i}$$

when $k=i$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^n \left(-x_n + \frac{e^{W_o^{iT}x_n+b}\cdot x_n}{\sum_{k=1}^{K} e^{W_o^{kT}x_n+b}}\right)$$

$$= \frac{1}{N}\sum_{n=1}^{N}\left(-x_n + \frac{e^{W_o^{iT}x_n+b}\cdot x_n}{\sum_{k=1}^{K} e^{W_o^{kT}x_n+b}}\right)$$

- $\dfrac{\partial L}{\partial b_o}$

$$\frac{\partial L}{\partial b_o} = \frac{\partial\left(-\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^n \log\left(\frac{e^{W_o^{kT}x_n+b}}{\sum_{k=1}^{K} e^{W_o^{kT}x_n+b}}\right)\right)}{\partial b_o^i}$$

$$= \frac{\partial\left(\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^n \left(-W_o^{kT}x_n+b\right)+\log\left(\sum_{k=1}^{K} e^{W_o^{kT}x_n+b}\right)\right)}{\partial b_o^i}$$

when $k=i$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^n \left(1 + \frac{e^{W_o^{iT}x_n+b}}{\sum_{k=1}^{K} e^{W_o^{kT}x_n+b}}\right)$$

$$= \frac{1}{N}\sum_{n=1}^{N}\left(1 + \frac{e^{W_o^{iT}x_n+b}}{\sum_{k=1}^{K} e^{W_o^{kT}x_n+b}}\right)$$

- $\dfrac{\partial L}{\partial W_h}$

$$\frac{\partial L}{\partial W_h} = \frac{\partial L}{\partial X_h} \cdot \frac{\partial X_h}{\partial W_h}$$

$$= \frac{\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{k} t_k^n \left(-W_o^{kT} X_n + b\right) + \log\left(\sum_{k=1}^{k} e^{W_o^{kT} X_n + b}\right)}{\partial X_h} \cdot \frac{\partial X_h}{\partial W_h}$$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{k} t_k^n \left(-W_o + \frac{e^{W_o^{T} X_n + b}}{\sum_{k=1}^{k} e^{W_o^{kT} X_n + b}} W_o \right) \cdot X_{input}$$

$$= \begin{cases} \frac{1}{N}\sum_{n=1}^{N} \left(-W_o + \frac{e^{W_o^{T} X_n + b}}{\sum_{k=1}^{k} e^{W_o^{kT} X_n + b}} W_o \right) \cdot X_{input} & \text{if } X_{hidden} > 0 \\ 0 & \text{otherwise} \end{cases}$$

- $\dfrac{\partial L}{\partial b_h}$

$$\frac{\partial L}{\partial b_h} = \frac{\partial L}{\partial X_h} \cdot \frac{\partial X_h}{\partial b_h}$$

$$= \frac{\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{k} t_k^n \left(-W_o^{kT} X_n + b\right) + \log\left(\sum_{k=1}^{k} e^{W_o^{kT} X_n + b}\right)}{\partial X_h} \cdot \frac{\partial X_h}{\partial b_h}$$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{k} t_k^n \left(-W_o + \frac{e^{W_o^{T} X_n + b}}{\sum_{k=1}^{k} e^{W_o^{kT} X_n + b}} W_o \right)$$

$$= \begin{cases} \frac{1}{N}\sum_{n=1}^{N} \left(-W_o + \frac{e^{W_o^{T} X_n + b}}{\sum_{k=1}^{k} e^{W_o^{kT} X_n + b}} W_o \right) & \text{if } X_{hidden} > 0 \\ 0 & \text{otherwise} \end{cases}$$

1.3 Learning [6 pts.]

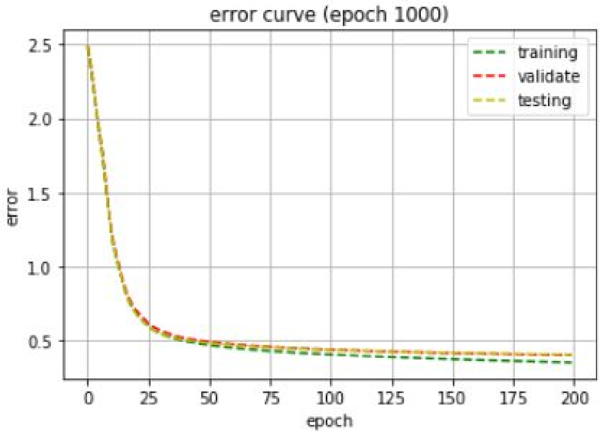


Fig 1.3.1 error curve (1000)

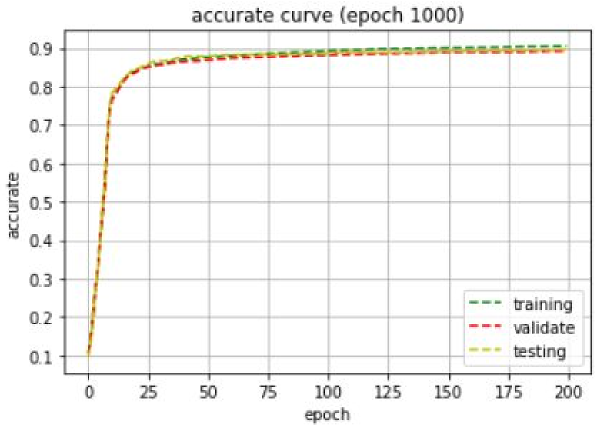


Fig 1.3.2 accurate curve (1000)

1.4 Hyperparameter Investigation [4 pts.]

1. Number of hidden units

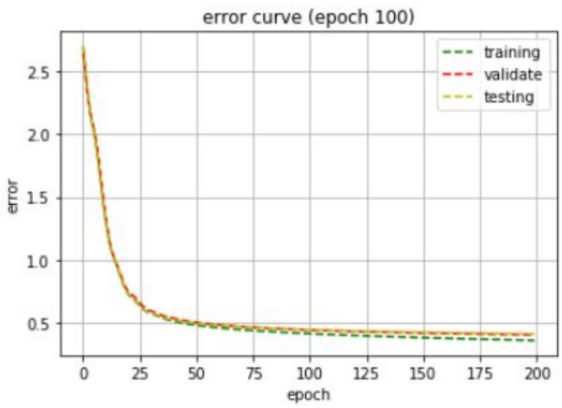


Fig 1.4.1 error curve (100)

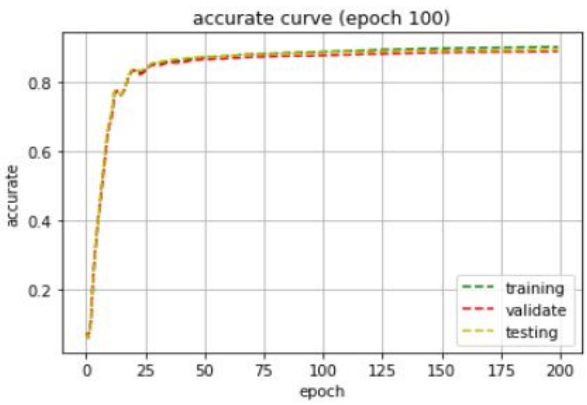


Fig 1.4.2 accurate curve (100)

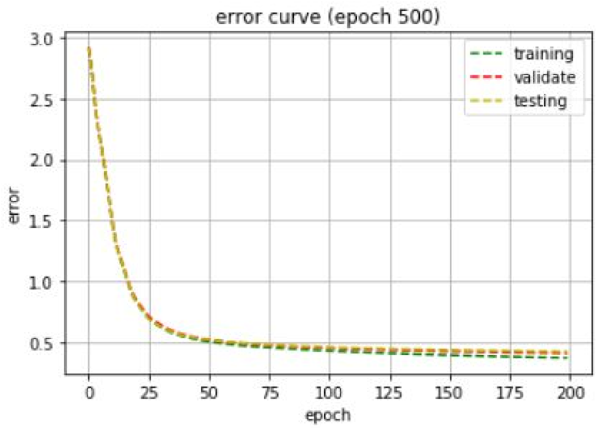


Fig 1.4.3 error curve (500)

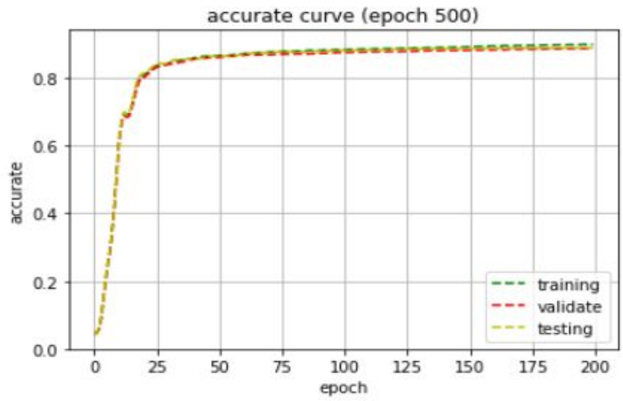


Fig 1.4.4 accurate curve (500)

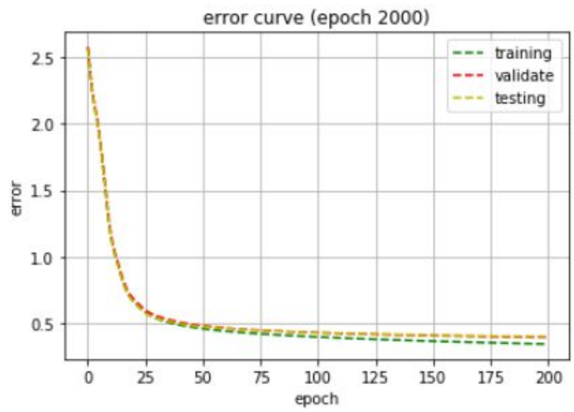


Fig 1.4.5 error curve (2000)

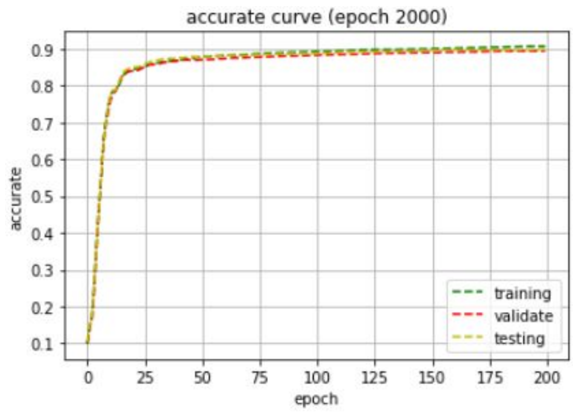


Fig 1.4.6 accurate curve (2000)

Hidden unit#	Accuracy %			Error		
	TrainData	Validation	Testing	TrainData	Validation	Testing
1000	90.57	89.21	89.68	0.3496	0.4003	0.4042
100	89.87	88.68	89.39	0.3731	0.4118	0.4238
500	90.15	88.88	88.91	0.3611	0.4063	0.4135
2000	90.65	89.4	89.83	0.3465	0.3971	0.4018

Table 1.4.7 Accuracy and Error

Discussion:

Based on the observation of the table, the relation between the number of hidden unit and accuracy or error is the larger number of hidden unit in the network model can improve the accuracy and error loss in the result。

2. Early stopping:

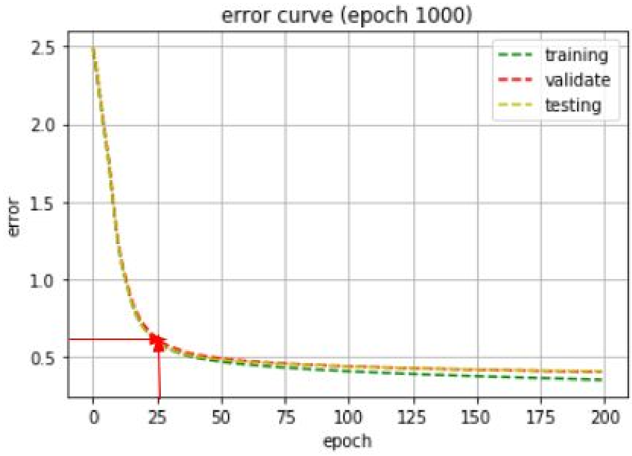


Fig 1.4.8 error curve (1000)

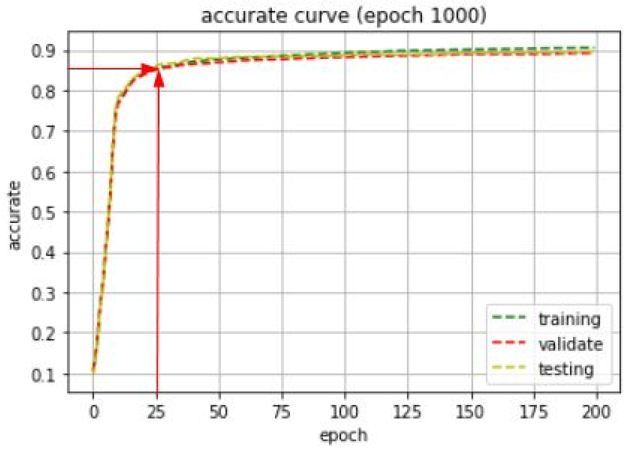


Fig 1.4.9 error curve (1000)

Epoch #	Accuracy %			Error		
	TrainData	Validation	Testing	TrainData	Validation	Testing
200	90.57	89.21	89.68	0.3496	0.4003	0.4042
25	85.06	84.916	85.499	0.6099	0.63153	0.5941

Discussion:
As shown in figure 1.4.8 and figure 1.4.9, after 25 epochs, the trend starts to slow down and coverage to a number 0.9 accuracy. in order to avoid overfitting, it is a good point to stay a margin in the model.

2. Neural Networks in Tensorflow [14 pts]

2.1 Model implementation [4 pts]:

Description:

Following the implementation steps:

1. Input Layer
2. A 3×3 convolutional layer, with 32 filters, using vertical and horizontal strides of 1.
3. ReLU activation
4. A batch normalization layer
5. A 2×2 max pooling layer
6. Flatten layer
7. Fully connected layer (with 784 output units, i.e. corresponding to each pixel)
8. ReLU activation
9. Fully connected layer (with 10 output units, i.e. corresponding to each class)
10. Softmax output
11. Cross Entropy loss

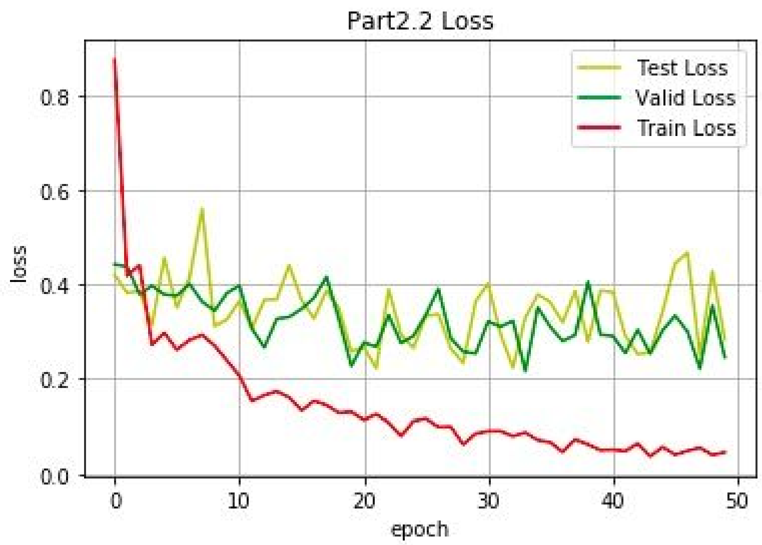
Python Code Snippets:

```python
def nn_init(features, labels):
    initializer=tf.contrib.layers.xavier_initializer()
    #1. input layer
    k_p = tf.placeholder(tf.float32)
    X_r = tf.reshape(features,shape = [-1,28,28,1])
    #Regularizer
    #reg = tf.contrib.layers.l2_regularizer(scale=)
    #2. 3*3 convolution
    W1 = tf.get_variable("W1",[3,3,1,32],dtype='float32',initializer=initializer)
    b1 = tf.get_variable("b1",[32],dtype='float32',initializer=initializer)
    filt_r = tf.nn.conv2d(X_r,W1,strides=[1,1,1,1],padding='SAME')
    #3. Relu activation
    conv_R = tf.nn.relu(filt_r+b1,name='conv_R')
    #4. Batch normalization
    mean, var = tf.nn.moments(conv_R,axes=[0,1,2])
    bn = tf.nn.batch_normalization(x=conv_R,mean=mean,variance=var,offset=None,scale=None,variance_epsilon=0.001)
    #5. 2*2 max pooling
    pool = tf.nn.max_pool(bn,ksize=[1,2,2,1],strides=[1,2,2,1],padding='SAME')
    #6. Flatten 28*28*32
    pool = tf.reshape(pool,[-1,6272])
    #7. Fully connected layer
    W2 = tf.get_variable("W2",[6272,1024],dtype='float32',initializer=initializer)
    b2 = tf.get_variable("b2",[1024],dtype='float32',initializer=initializer)
    #8. Relu activation / dropout
    fully_c1 = tf.matmul(pool,W2)+b2
    if dropout:
        drop = tf.nn.dropout(fully_c1,k_p)
        fully_c1 = tf.nn.relu(drop)
    else:
        fully_c1 = tf.nn.relu(fully_c1)
    #9. Fully connected layer
    W3 = tf.get_variable("W3",[1024,10],dtype='float32',initializer=initializer)
    b3 = tf.get_variable("b3",[10],dtype='float32',initializer=initializer)
    #10. Softmax
    fully_c2 = tf.matmul(fully_c1,W3)+b3
```

```python
    acc,acc_o = tf.metrics.accuracy(labels=tf.argmax(sm,1),predictions=tf.argmax(labels,1))
    #11. Cross Entropy Loss
    en = tf.nn.softmax_cross_entropy_with_logits_v2(labels=labels,logits=fully_c2)
    loss = tf.reduce_mean(en)
    #Regulization
    op = tf.train.AdamOptimizer(0.0001).minimize(loss)
```

2.2 Model Training [4 pts]:

Description: Use batch size of 32, 50 epochs and learning rate of 1e-4 through SGD to output training, validation and test loss and accuracy.



2.3 Hyperparamter Investigation [6 pts]:

- **1. L2 Normalization:**

Description: Use lambda = [0.01, 0.1, 0.5] other remained.

```
reg = tf.contrib.layers.l2_regularizer(scale=regulize)
```

```
reg_var = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
reg_term = tf.contrib.layers.apply_regularization(reg,reg_var)
loss += reg_term
```

p = 0.01

```
Loss in train: 0.3797518407756632
Loss in valid: 0.4402970572312673
Losss in test: 0.3797518407756632
Accuracy in train: 0.9131329330531034
Accuracy in valid: 0.9134248097737631
Accuracy in test: 0.9131329330531034
```

p = 0.1

```
Loss in train: 2.379045529799028
Loss in valid: 2.3024564186731973
Losss in test: 2.379045529799028
Accuracy in train: 0.8837903250347484
Accuracy in valid: 0.8841543594996134
Accuracy in test: 0.8837903250347484
```

p = 0.5

```
Loss in train: 221.29562932794744
Loss in valid: 214.94220225016275
Losss in test: 221.29562932794744
Accuracy in train: 0.6177025274796919
Accuracy in valid: 0.6193171938260397
Accuracy in test: 0.6177025274796919
```

Observation: With different lambda normalization, the weight parameter becomes more effective to the final answer. As shown in the tabulars, when the normalization factor become larger, the loss increases and accuracy decreases. The L2 normalization avoids the perfect matching and results in longer time to reach the high accuracy and low loss.

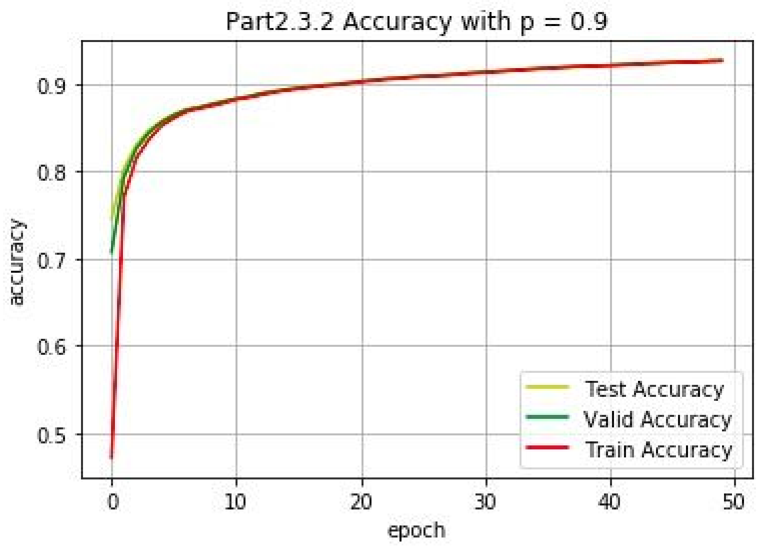
- **2. Dropout:**

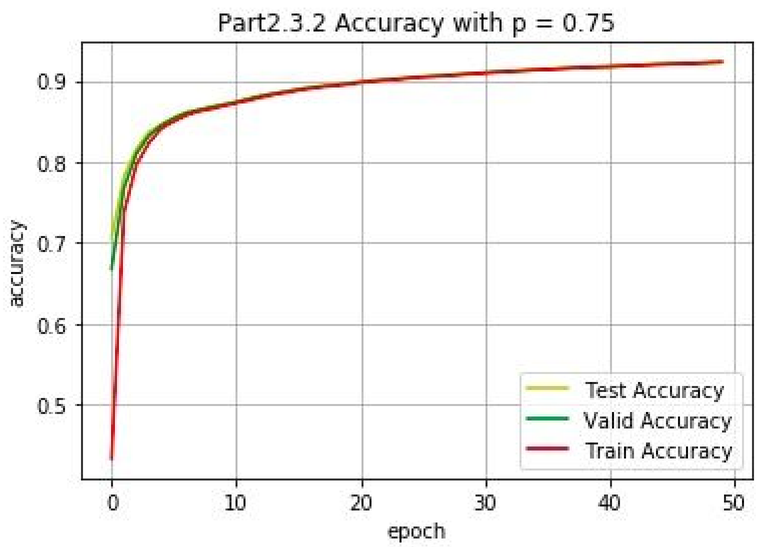
Description: Use dropout probabilities = [0.9, 0.75, 0.5] other remained.

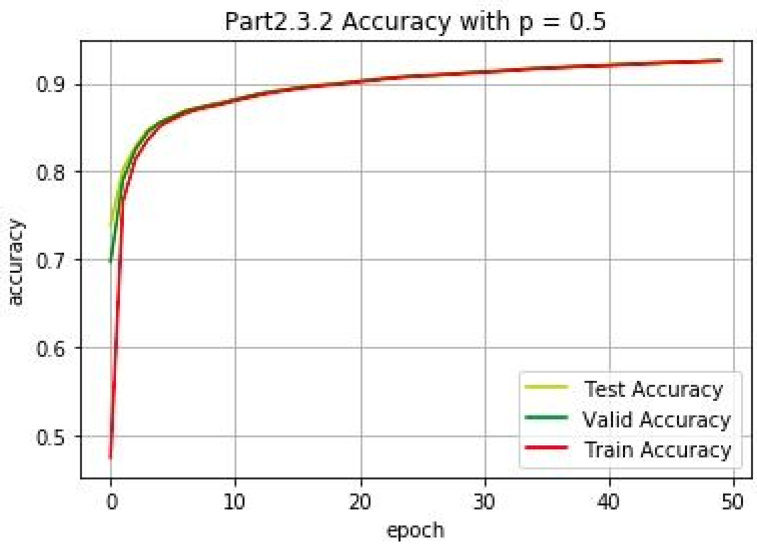
$$
\begin{aligned}
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)}\mathbf{y}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}),
\end{aligned}
$$

$$
\begin{aligned}
r_j^{(l)} &\sim \text{Bernoulli}(p), \\
\widetilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)}\widetilde{\mathbf{y}}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}).
\end{aligned}
$$

```
fc1 = (tf.matmul(pool, W2) + b2)
drop = tf.nn.dropout(fc1,0.9)
fc1 = tf.nn.relu(drop)
```







Observation: As graphs shown above, it indicates that the higher dropout probability will result in less accuracy for the same epoch. As dropout method is used to deal with overfitting and gradient vanishing, the dropout probability with p = 0.5 has higher error-tolerant rate and avoid the incident of missing part in both hidden and visual layers. Meanwhile, lower dropout probability can lower the structure risk.