Homework 2

Start date: *29th January 2018*

Due date: *11th February 2018, 11:55 pm*

How to Submit

When you have completed the exercises and everything has finsihed running, click on 'File' in the menu-bar and then 'Download .ipynb'. This file must be submitted to Moodle named as **studentnumber_DL_hw2.ipynb** before the deadline above.

Also send a **sharable link** to the notebook at the following email: ucl.coursework.submit@gmail.com. You can also make it sharable via link to everyone, up to you.

IMPORTANT

Please make sure you submission includes **all results/plots/tables** required for grading. We will not re-run your code.

The Data

Handwritten Digit Recognition Dataset (MNIST)

In this assignment we will be using the MNIST digit dataset (https://yann.lecun.com/exdb/mnist/).

The dataset contains images of hand-written digits ($0 - 9$), and the corresponding labels.

The images have a resolution of 28×28 pixels.

The MNIST Dataset in TensorFlow

You can use the tensorflow build-in functionality to download and import the dataset into python (see *Setup* section below). But note that this will be the only usage of TensorFlow in this assignment.

The Assignment

Objectives

This assignment will be mirroring the first assignment (DL_hw1), but this time you are **not allowed to use any of the Tensorflow functionality for specifing nor optimizing** your neural network models. You will now use your **own implementations** of different neural network models (labelled Model 1-4, and described in the corresponding sections of the Colab).

As before, you will train these models to classify hand written digits from the Mnist dataset.

Keep in mind, the purpose of this exercise is to implement and optimize your own neural networks architectures without the toolbox/library tailored to do so. This also means, in order to train and evaluate your models, you will need to implement your own optimization procedure. You are to use the same cross-entropy loss as before and your own implementation of SGD.

Additional instruction:

Do not use any other libraries than the ones provided in the imports cell. You should be able to do everything via *numpy* (especially for the convolutional layer, rely on the in-built matrix/tensor multiplication that numpy offers).

There are a few questions at the end of the colab. **Before doing any coding, please take look at Question 1** -- this should help you with the implementations, especially the optimization part.

Variable Initialization

Initialize the variables containing the parameters using Xavier initialization (http://proceedings.mlr.press/v9/glorot10a.html).

Hyper-parameters

For each of these models you will be requested to run experiments with different hyper-parameters.

More specifically, you will be requested to try 3 sets of hyper-parameters per model, and report the resulting model accuracy.

Each combination of hyper-parameter will specify how to set each of the following:

- **num_epochs**: Number of iterations through the training section of the dataset [*a positive integer*].
- **learning_rate**: Learning rate used by the gradient descent optimizer [*a scalar between 0 and 1*]

In all experiments use a *batch_size* of 100.

Loss function

All models, should be trained as to minimize the **cross-entropy loss** function:

$$\text{loss} = -\sum_{i=1}^{N} \log p(y_i \,|\, x_i, \theta) = -\sum_{i=1}^{N} \log \underbrace{\left(\frac{\exp(z_i[y_i])}{\sum_{c=1}^{10} \exp(z_i[c])} \right)}_{\text{softmax output}} = \sum_{i=1}^{N} \left(-z_i[y_i] + \log \left(\sum_{c=1}^{10} \exp(z_i[c]) \right) \right)$$

where $z \in \mathrm{R}^{10}$ is the input to the softmax layer and $z[c]$ denotes the c-th entry of vector z. And i is a index for the dataset $\{(x_i, y_i)\}_{i=1}^{N}$.

Optimization

Use **stochastic gradient descent (SGD)** for optimizing the loss function. Sum over the batch.

Training and Evaluation

The tensorflow built-in functionality for downloading and importing the dataset into python returns a Datasets object.

This object will have three attributes:

- train
- validation
- test

Use only the **train** data in order to optimize the model.

Use *datasets.train.next_batch(100)* in order to sample mini-batches of data.

Every 20000 training samples (i.e. every 200 updates to the model), interrupt training and measure the accuracy of the model,

each time evaluate the accuracy of the model both on 20% of the **train** set and on the entire **test** set.

Reporting

For each model i, you will collect the learning curves associated to each combination of hyper-parameters.

Use the utility function `plot_learning_curves` to plot these learning curves,

and the and utility function `plot_summary_table` to generate a summary table of results.

For each run collect the train and test curves in a tuple, together with the hyper-parameters.

```
experiments_task_i = [

   (num_epochs_1, learning_rate_1), train_accuracy_1, test_accuracy_1),

   (num_epochs_2, learning_rate_2), train_accuracy_2, test_accuracy_2),

   (num_epochs_3, learning_rate_3), train_accuracy_3, test_accuracy_3)]
```

Hint:

Remind yourselves of the chain rule and read through the lecture notes on back-propagation (computing the gradients by recursively applying the chain rule). This is a general procedure that applies to all model architectures you will have to code in the following steps. Thus, you are strongly encourage to implement an optimization procedure that generalizes and can be re-used to train all your models. Recall the only things that you will need for each layer are:

(i) the gradients of layer with respect to its input

(ii) the gradients with respect to its parameters, if any.

(See Question 1).

Also from the previous assignment, you should have a good idea of what to expect, both terms of behavior and relative performance. (To keep everything comparable, we kept all the hyperparameters and reporting the same).

Imports and utility functions (do not modify!)

In [0]:

```
# Import useful libraries.
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np

# Global variables.
log_period_samples = 20000
batch_size = 100

# Import dataset with one-hot encoding of the class labels.
def get_data():
  return input_data.read_data_sets("MNIST_data/", one_hot=True)

# Placeholders to feed train and test data into the graph.
# Since batch dimension is 'None', we can reuse them both for train and eval.
def get_placeholders():
  x = tf.placeholder(tf.float32, [None, 784])
  y_ = tf.placeholder(tf.float32, [None, 10])
  return x, y_

# Plot learning curves of experiments
def plot_learning_curves(experiment_data):
  # Generate figure.
  fig, axes = plt.subplots(3, 4, figsize=(22,12))
  st = fig.suptitle(
      "Learning Curves for all Tasks and Hyper-parameter settings",
      fontsize="x-large")
  # Plot all learning curves.
  for i, results in enumerate(experiment_data):
    for j, (setting, train_accuracy, test_accuracy) in enumerate(results):
      # Plot.
      xs = [x * log_period_samples for x in range(1, len(train_accuracy)+1)]
      axes[j, i].plot(xs, train_accuracy, label='train_accuracy')
      axes[j, i].plot(xs, test_accuracy, label='test_accuracy')
      # Prettify individual plots.
      axes[j, i].ticklabel_format(style='sci', axis='x', scilimits=(0,0))
      axes[j, i].set_xlabel('Number of samples processed')
      axes[j, i].set_ylabel('Epochs: {}, Learning rate: {}.  Accuracy'.format(*setting))
      axes[j, i].set_title('Task {}'.format(i + 1))
      axes[j, i].legend()
  # Prettify overall figure.
  plt.tight_layout()
  st.set_y(0.95)
  fig.subplots_adjust(top=0.91)
  plt.show()

# Generate summary table of results.
def plot_summary_table(experiment_data):
  # Fill Data.
  cell_text = []
  rows = []
  columns = ['Setting 1', 'Setting 2', 'Setting 3']
  for i, results in enumerate(experiment_data):
    rows.append('Model {}'.format(i + 1))
    cell_text.append([])
    for j, (setting, train_accuracy, test_accuracy) in enumerate(results):
      cell_text[i].append(test_accuracy[-1])
  # Generate Table.
  fig=plt.figure(frameon=False)
  ax = plt.gca()
  the_table = ax.table(
      cellText=cell_text,
      rowLabels=rows,
      colLabels=columns,
      loc='center')
  the_table.scale(1, 4)
  # Prettify.
  ax.patch.set_facecolor('None')
  ax.xaxis.set_visible(False)
  ax.yaxis.set_visible(False)
```

Model

Train a neural network model consisting of 1 linear layer, followed by a softmax:

(input → linear layer → softmax → class probabilities)

Hyper-parameters

Train the model with three different hyper-parameter settings:

- *num_epochs*=5, *learning_rate*=0.0001
- *num_epochs*=5, *learning_rate*=0.005
- *num_epochs*=5, *learning_rate*=0.1

In [0]:
```
# CAREFUL: Running this CL resets the experiments_task1 dictionary where results should be stored.
# Store results of runs with different configurations in a dictionary.
# Use a tuple (num_epochs, learning_rate) as keys, and a tuple (training_accuracy, testing_accuracy)
experiments_task1 = []
settings = [(5, 0.0001), (5, 0.005), (5, 0.1)]
```

In [16]:

```
print('Training Model 1')

# Train Model 1 with the different hyper-parameter settings.
for (num_epochs, learning_rate) in settings:

  # Reset graph, recreate placeholders and dataset.
  mnist = get_data()
  eval_mnist = get_data()

  ####################################################
  # Define model, loss, update and evaluation metric. #
  ####################################################

  # define the components of the model including loss
  graph = [linear(name='linear1',\
                  shape=(784, 10),\
                  initializer=xavier),\
           cross_entropy_softmax_logits(name='loss1')]

  # define the model
  model = Model(graph)

  # define the optimizer
  optimizer = SGD(name='sgd',\
                  learning_rate=learning_rate,\
                  model=model)

  # since evaluation metric is a function, we just need to call it directly

  # Train.
  i, train_accuracy, test_accuracy = 0, [], []
  log_period_updates = int(log_period_samples / batch_size)
  while mnist.train.epochs_completed < num_epochs:

    # Update.
    i += 1
    batch_xs, batch_ys = mnist.train.next_batch(batch_size)

    #################
    # Training step #
    #################
    optimizer.learning(batch_xs, batch_ys)

    # Periodically evaluate.
    if i % log_period_updates == 0:

      ###################################
      # Compute and store train accuracy. #
      ###################################
      train_data_size = eval_mnist.train.images.shape[0]
      eval_train_xs = eval_mnist.train.images[:int(train_data_size*0.2)]
      eval_train_ys = eval_mnist.train.labels[:int(train_data_size*0.2)]
      _, train_predicts = model.forward_pass(eval_train_xs)
      train_predicts = softmax(train_predicts)
      acc_train = accuracy_evaluation(train_predicts, eval_train_ys)
      print ('accuracy of train data: ', acc_train)
      train_accuracy.append(acc_train)
      ###################################
      # Compute and store test accuracy.  #
      ###################################
      eval_test_xs = eval_mnist.test.images
      eval_test_ys = eval_mnist.test.labels
      _, test_predicts = model.forward_pass(eval_test_xs)
      test_predicts = softmax(test_predicts)
      acc_test = accuracy_evaluation(test_predicts, eval_test_ys)
      print ('accuracy of test data: ', acc_test)
      test_accuracy.append(acc_test)
  del graph
  del model
  del optimizer
  experiments_task1.append(
      ((num_epochs, learning_rate), train_accuracy, test_accuracy))
```

```
Training Model 1
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Module : linear1 is constructed.
Module : loss1 is constructed.
accuracy of train data:  0.7436363636363637
accuracy of test data:  0.7551
accuracy of train data:  0.8075454545454546
accuracy of test data:  0.8187
accuracy of train data:  0.8293636363636364
accuracy of test data:  0.8444
accuracy of train data:  0.845
accuracy of test data:  0.8599
```

Model 2 (15 pts)

1 hidden layer (32 units) with a ReLU non-linearity, followed by a softmax.

(input → non-linear layer → linear layer → softmax → class probabilities)

Hyper-parameters

Train the model with three different hyper-parameter settings:

- *num_epochs*=15, *learning_rate*=0.0001
- *num_epochs*=15, *learning_rate*=0.005
- *num_epochs*=15, *learning_rate*=0.1

In [0]:
```
# CAREFUL: Running this CL resets the experiments_task1 dictionary where results should be stored.
# Store results of runs with different configurations in a dictionary.
# Use a tuple (num_epochs, learning_rate) as keys, and a tuple (training_accuracy, testing_accuracy)
experiments_task2 = []
settings = [(15, 0.0001), (15, 0.005), (15, 0.1)]
```

In [18]:

```
print('Training Model 2')

# Train Model 1 with the different hyper-parameter settings.
for (num_epochs, learning_rate) in settings:

  mnist = get_data()  # use for training.
  eval_mnist = get_data()  # use for evaluation.

  ##################################################
  # Define model, loss, update and evaluation metric. #
  ##################################################

  # define the components of the model including loss
  graph = [linear(name='linear1',\
                  shape=(784, 32),\
                  initializer=xavier),\
           relu(name='relu1'),\
           linear(name='linear2',\
                  shape=(32, 10),\
                  initializer=xavier),\
           cross_entropy_softmax_logits(name='loss1')]

  # define the model
  model = Model(graph)

  # define the optimizer
  optimizer = SGD(name='sgd',\
                  learning_rate=learning_rate,\
                  model=model)

  # Train.
  i, train_accuracy, test_accuracy = 0, [], []
  log_period_updates = int(log_period_samples / batch_size)
  while mnist.train.epochs_completed < num_epochs:

    # Update.
    i += 1
    batch_xs, batch_ys = mnist.train.next_batch(batch_size)

    #################
    # Training step #
    #################
    optimizer.learning(batch_xs, batch_ys)

    # Periodically evaluate.
    if i % log_period_updates == 0:

      ###################################
      # Compute and store train accuracy. #
      ###################################
      train_data_size = eval_mnist.train.images.shape[0]
      eval_train_xs = eval_mnist.train.images[:int(train_data_size*0.2)]
      eval_train_ys = eval_mnist.train.labels[:int(train_data_size*0.2)]
      _, train_predicts = model.forward_pass(eval_train_xs)
      train_predicts = softmax(train_predicts)
      acc_train = accuracy_evaluation(train_predicts, eval_train_ys)
      print ('accuracy of train data: ', acc_train)
      train_accuracy.append(acc_train)
      ##################################
      # Compute and store test accuracy.  #
      ##################################
      eval_test_xs = eval_mnist.test.images
      eval_test_ys = eval_mnist.test.labels
      _, test_predicts = model.forward_pass(eval_test_xs)
      test_predicts = softmax(test_predicts)
      acc_test = accuracy_evaluation(test_predicts, eval_test_ys)
      print ('accuracy of test data: ', acc_test)
      test_accuracy.append(acc_test)
  del graph
  del model
  del optimizer
  experiments_task2.append(
      ((num_epochs, learning_rate), train_accuracy, test_accuracy))
```

```
accuracy of train data:  0.9055454545454545
accuracy of test data:  0.9136
accuracy of train data:  0.9074545454545454
accuracy of test data:  0.9156
accuracy of train data:  0.9068181818181819
accuracy of test data:  0.9163
accuracy of train data:  0.9082727272727272
accuracy of test data:  0.917
accuracy of train data:  0.9094545454545454
accuracy of test data:  0.918
accuracy of train data:  0.9099090909090909
accuracy of test data:  0.9189
accuracy of train data:  0.9118181818181819
accuracy of test data:  0.9197
accuracy of train data:  0.9122727272727272
accuracy of test data:  0.9206
accuracy of train data:  0.9114545454545454
accuracy of test data:  0.9199
accuracy of train data:  0.9134545454545454
```

Model 3 (15 pts)

2 hidden layers (32 units) each, with ReLU non-linearity, followed by a softmax.

(input → non-linear layer → non-linear layer → linear layer → softmax → class probabilities)

Hyper-parameters

Train the model with three different hyper-parameter settings:

- *num_epochs*=5, *learning_rate*=0.003
- *num_epochs*=40, *learning_rate*=0.003
- *num_epochs*=40, *learning_rate*=0.05

In [0]:
```
# CAREFUL: Running this CL resets the experiments_task1 dictionary where results should be stored.
# Store results of runs with different configurations in a dictionary.
# Use a tuple (num_epochs, learning_rate) as keys, and a tuple (training_accuracy, testing_accuracy)
experiments_task3 = []
settings = [(5, 0.003), (40, 0.003), (40, 0.05)]
```

In [20]:

```
print('Training Model 3')

# Train Model 1 with the different hyper-parameter settings.
for (num_epochs, learning_rate) in settings:

  mnist = get_data()  # use for training.
  eval_mnist = get_data()  # use for evaluation.

  ##################################################
  # Define model, loss, update and evaluation metric. #
  ##################################################

  # define the components of the model including loss
  graph = [linear(name='linear1',\
                  shape=(784, 32),\
                  initializer=xavier),\
           relu(name='relu1'),\
           linear(name='linear2',\
                  shape=(32, 32),\
                  initializer=xavier),\
           relu(name='relu2'),\
           linear(name='linear3',\
                  shape=(32, 10),\
                  initializer=xavier),\
           cross_entropy_softmax_logits(name='loss1')]

  # define the model
  model = Model(graph)

  # define the optimizer
  optimizer = SGD(name='sgd',\
                  learning_rate=learning_rate,\
                  model=model)

  # Train.
  i, train_accuracy, test_accuracy = 0, [], []
  log_period_updates = int(log_period_samples / batch_size)
  while mnist.train.epochs_completed < num_epochs:

    # Update.
    i += 1
    batch_xs, batch_ys = mnist.train.next_batch(batch_size)

    #################
    # Training step #
    #################
    optimizer.learning(batch_xs, batch_ys)

    # Periodically evaluate.
    if i % log_period_updates == 0:

      ###################################
      # Compute and store train accuracy. #
      ###################################
      train_data_size = eval_mnist.train.images.shape[0]
      eval_train_xs = eval_mnist.train.images[:int(train_data_size*0.2)]
      eval_train_ys = eval_mnist.train.labels[:int(train_data_size*0.2)]
      _, train_predicts = model.forward_pass(eval_train_xs)
      train_predicts = softmax(train_predicts)
      acc_train = accuracy_evaluation(train_predicts, eval_train_ys)
      print ('accuracy of train data: ', acc_train)
      train_accuracy.append(acc_train)
      ###################################
      # Compute and store test accuracy.  #
      ###################################
      eval_test_xs = eval_mnist.test.images
      eval_test_ys = eval_mnist.test.labels
      _, test_predicts = model.forward_pass(eval_test_xs)
      test_predicts = softmax(test_predicts)
      acc_test = accuracy_evaluation(test_predicts, eval_test_ys)
      print ('accuracy of test data: ', acc_test)
      test_accuracy.append(acc_test)
  del graph
  del model
  del optimizer
  experiments_task3.append(
      ((num_epochs, learning_rate), train_accuracy, test_accuracy))
```

```
Training Model 3
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Module : linear1 is constructed.
Module : relu1 is constructed.
Module : linear2 is constructed.
Module : relu2 is constructed.
Module : linear3 is constructed.
Module : loss1 is constructed.
accuracy of train data:  0.8958181818181818
accuracy of test data:  0.9022
accuracy of train data:  0.931
accuracy of test data:  0.932
```

Model 4 (20 pts)

Model

3 layer convolutional model (2 convolutional layers followed by max pooling) + 1 non-linear layer (32 units), followed by softmax.

(input(28x28) → conv(3x3x8) + maxpool(2x2) → conv(3x3x8) + maxpool(2x2) → flatten → non-linear → linear layer → softmax → class probabilities)

- Use *padding* = '*SAME*' for both the convolution and the max pooling layers.
- Employ plain convolution (no stride) and for max pooling operations use 2x2 sliding windows, with no overlapping pixels (note: this operation will down-sample the input image by 2x2).

Hyper-parameters

Train the model with three different hyper-parameter settings:

- num_epochs=5, learning_rate=0.01
- num_epochs=10, learning_rate=0.001
- num_epochs=20, learning_rate=0.001

In [0]:
```
# CAREFUL: Running this CL resets the experiments_task1 dictionary where results should be stored.
# Store results of runs with different configurations in a dictionary.
# Use a tuple (num_epochs, learning_rate) as keys, and a tuple (training_accuracy, testing_accuracy)
experiments_task4 = []
settings = [(5, 0.01), (10, 0.001), (20, 0.001)]
```

In [22]:

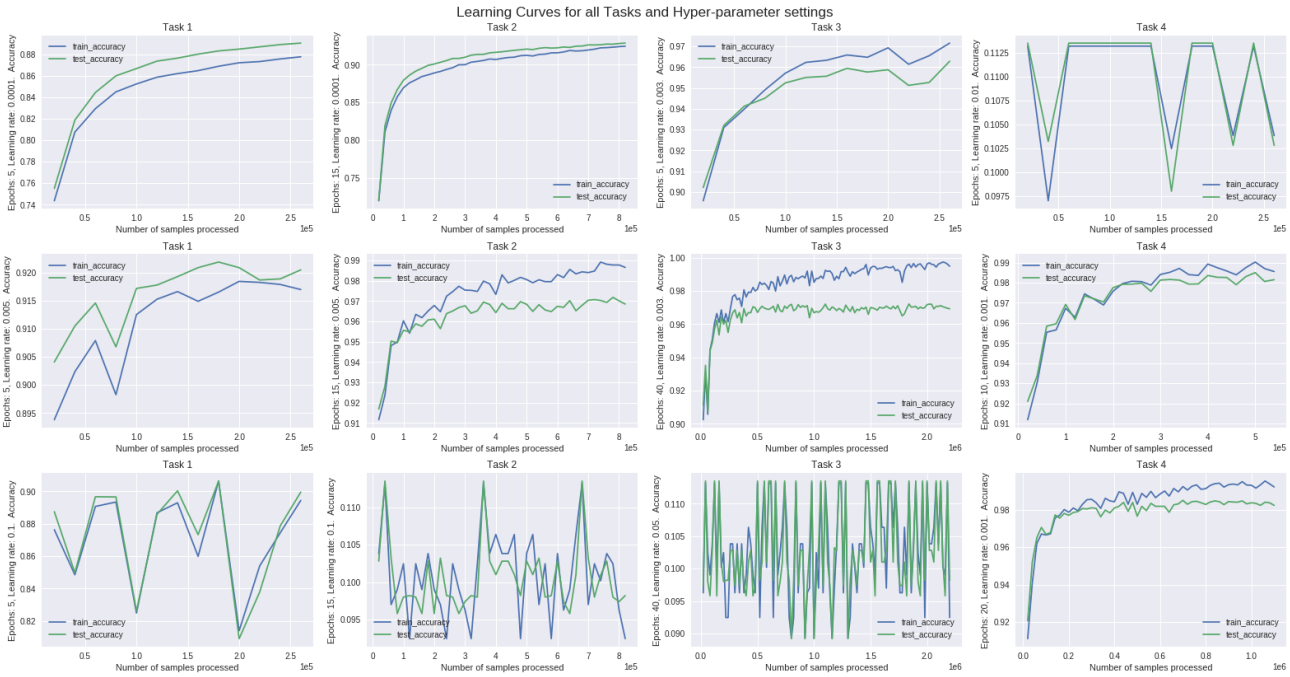
```
print('Training Model 4')

# Train Model 1 with the different hyper-parameter settings.
for (num_epochs, learning_rate) in settings:

  mnist = get_data()  # use for training.
  eval_mnist = get_data()  # use for evaluation.

  ################################################
  # Define model, loss, update and evaluation metric. #
  ################################################

  # define the components of the model including loss
  graph = [convolution(name='conv1',\
                       shape=(3,3,1,8),\
                       initializer=xavier,\
                       padding='SAME',\
                       stride=(1,1)),\
           maxpooling(name='maxpool1',\
                      shape=(2,2),\
                      padding='SAME',\
                      stride=(2,2)),\
           convolution(name='conv2',\
                       shape=(3,3,8,8),\
                       initializer=xavier,\
                       padding='SAME',\
                       stride=(1,1)),\
           maxpooling(name='maxpool2',\
                      shape=(2,2),\
                      padding='SAME',\
                      stride=(2,2)),\
           flatten(name='flatten1'),\
           linear(name='linear1',\
                  shape=(7*7*8, 32),\
                  initializer=xavier),\
           relu(name='relu1'),\
           linear(name='linear2',\
                  shape=(32, 10),\
                  initializer=xavier),\
           cross_entropy_softmax_logits(name='loss1')]

  # define the model
  model = Model(graph)

  # define the optimizer
  optimizer = SGD(name='sgd',\
                  learning_rate=learning_rate,\
                  model=model)

  # Train.
  i, train_accuracy, test_accuracy = 0, [], []
  log_period_updates = int(log_period_samples / batch_size)
  while mnist.train.epochs_completed < num_epochs:

    # Update.
    i += 1
    batch_xs, batch_ys = mnist.train.next_batch(batch_size)

    #################
    # Training step #
    #################
    batch_xs = batch_xs.reshape(-1, 28, 28, 1)
    optimizer.learning(batch_xs, batch_ys)

    # Periodically evaluate.
    if i % log_period_updates == 0:

      ###################################
      # Compute and store train accuracy. #
      ###################################
      train_data_size = eval_mnist.train.images.shape[0]
      eval_train_xs = eval_mnist.train.images[:int(train_data_size*0.2)]
      eval_train_xs = eval_train_xs.reshape(-1, 28, 28, 1)
      eval_train_ys = eval_mnist.train.labels[:int(train_data_size*0.2)]
      _, train_predicts = model.forward_pass(eval_train_xs)
      train_predicts = softmax(train_predicts)
      acc_train = accuracy_evaluation(train_predicts, eval_train_ys)
      print ('accuracy of train data: ', acc_train)
      train_accuracy.append(acc_train)
      ##################################
      # Compute and store test accuracy.  #
      ##################################
      eval_test_xs = eval_mnist.test.images
      eval_test_ys = eval_mnist.test.labels
      eval_test_xs = eval_test_xs.reshape(-1, 28, 28, 1)
      _, test_predicts = model.forward_pass(eval_test_xs)
      test_predicts = softmax(test_predicts)
      acc_test = accuracy_evaluation(test_predicts, eval_test_ys)
      print ('accuracy of test data: ', acc_test)
      test_accuracy.append(acc_test)
  del graph
  del model
  del optimizer
  experiments_task4.append(
      ((num_epochs, learning_rate), train_accuracy, test_accuracy))
```

```
Training Model 4
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
```

```
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Module : conv1 is constructed.
Module : maxpool1 is constructed.
Module : conv2 is constructed.
Module : maxpool2 is constructed.
Module : flatten1 is constructed.
Module : linear1 is constructed.
Module : relu1 is constructed.
Module : linear2 is constructed.
Module : loss1 is constructed.
accuracy of train data:  0.11318181818181818
```

Results

In [23]:
```
plot_learning_curves([experiments_task1, experiments_task2, experiments_task3, experiments_task4])
```



In [24]:
```
plot_summary_table([experiments_task1, experiments_task2, experiments_task3, experiments_task4])
```

	Setting 1	Setting 2	Setting 3
Model 1	0.8904	0.9205	0.8996
Model 2	0.9285	0.9685	0.0982
Model 3	0.9629	0.9693	0.0982
Model 4	0.1028	0.9815	0.9824

Questions

###Q1 (32 pts): Compute the following derivatives Show all intermediate steps in the derivation (in markdown below). Provide the final results in vector/matrix/tensor form whenever appropiate.

1. [5 pts] Give the cross-entropy loss above, compute the derivative of the loss function with respect to the scores z (the input to the softmax layer).

$$\frac{\partial loss}{\partial z} = ?$$

2. [12 pts] Consider the first model (M1: linear + softmax). Compute the derivative of the loss with respect to

   - the input x

$$\frac{\partial loss}{\partial x} = ?$$

   - the parameters of the linear layer: weights W and bias b

$$\frac{\partial loss}{\partial W} = ?$$

$$\frac{\partial loss}{\partial b} = ?$$

3. [10 pts] Compute the derivative of a convolution layer wrt. to its parameters W and wrt. to its input (4-dim tensor). Assume a filter of size H x W x D, and stride 1.

$$\frac{\partial loss}{\partial W} = ?$$

A1: (Your answer here)

1.

$$\text{loss} = -\sum_{i=1}^{N} \log p(y_i | x_i, \theta) = -\sum_{i=1}^{N} \log \underbrace{\left(\frac{\exp(z_i[y_i])}{\sum_{c=1}^{10} \exp(z_i[c])} \right)}_{\text{softmax output}} = \sum_{i=1}^{N} \left(-z_i[y_i] + \log \left(\sum_{c=1}^{10} \exp(z_i[c]) \right) \right)$$

where $z \in \mathrm{R}^{10}$ is the input to the softmax layer and $z[c]$ denotes the c-th entry of vector z. And i is a index for the dataset $\{(x_i, y_i)\}_{i=1}^{N}$.

$$\frac{\partial loss}{\partial z_i} = \begin{cases} -1 + \frac{\exp(z_i[k])}{\sum_{c=1}^{10} \exp(z_i[c])} & \text{if } k = y \\ \frac{\exp(z_i[k])}{\sum_{c=1}^{10} \exp(z_i[c])} & \text{if } k \neq y \end{cases}$$

To reform it more compact, we can define a $\delta_{x,y}$ function such that

$$\delta_{x,y} = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

So, we now can write the derivative as

$$\frac{\partial loss}{\partial z_i} = \left[-\delta_{1,y} + \frac{\exp(z_i[1])}{\sum_{c=1}^{10} \exp(z_i[c])}, \quad -\delta_{2,y} + \frac{\exp(z_i[2])}{\sum_{c=1}^{10} \exp(z_i[c])}, \quad \ldots, \quad -\delta_{9,y} + \frac{\exp(z_i[9])}{\sum_{c=1}^{10} \exp(z_i[c])} \quad -\delta_{10,y} + \frac{\exp(z_i[10])}{\sum_{c=1}^{10} \exp(z_i[c])} \right] \in \mathrm{R}^{1 \times 10}$$

So, we can get

$$\frac{\partial loss}{\partial z} = \begin{bmatrix} \frac{\partial loss}{\partial z_1} \\ \frac{\partial loss}{\partial z_2} \\ \cdots \\ \cdots \\ \cdots \\ \frac{\partial loss}{\partial z_N} \end{bmatrix} \in \mathrm{R}^{N \times 10}$$

2.

(1)

First, we define the definition of the linear function as

$$Z = W \cdot X + b \text{ where } W \in \mathrm{R}^{10 \times K}, b \in \mathrm{R}^{10}, X \in \mathrm{R}^{K \times N}, Z \in \mathrm{R}^{10 \times N}$$

Now, we calculate the derivative of Z w.r.t X such that

$$\frac{\partial Z[i,j]}{\partial X[p,q]} = \frac{\partial}{\partial X[p,q]} \left(\sum_{l}^{K} W[i,l] \cdot X[l,j] + b[j] \right) = W[i,q] \cdot \delta_{i,p} \cdot \delta_{j,q}$$

$$\frac{\partial loss}{\partial X} = \frac{\partial loss}{\partial Z} \cdot \frac{\partial Z}{\partial X} = \frac{\partial loss}{\partial Z} \cdot W \in \mathrm{R}^{N \times K}$$

(2)

Then, we derive the derivative of *loss* w.r.t W such that

$$\frac{\partial z[i,j]}{\partial W[p,q]} = \frac{\partial}{\partial W[p,q]} \left(\sum_{l}^{K} W[i,l] \cdot X[l,j] + b[j] \right) = X[i,q] \cdot \delta_{i,p} \cdot \delta_{j,q}$$

$$\frac{\partial loss}{\partial W} = \frac{\partial Z}{\partial W} \cdot \frac{\partial loss}{\partial Z} = X \cdot \frac{\partial loss}{\partial Z} \in \mathrm{R}^{K \times 10}$$

Finally, we derive the derivative of *loss* w.r.t b such that

$$\frac{\partial Z[i,j]}{\partial b[q]} = \frac{\partial}{\partial b[q]} \left(\sum_{l} W[i,l] \cdot X[l,j] + b[j] \right) = \delta_{q,j}$$

$$\frac{\partial loss}{\partial b} = \sum_{i} \frac{\partial loss}{\partial Z}[i,:] = \mathbf{1}^T \cdot \frac{\partial loss}{\partial Z}$$

where $\mathbf{1} \in \mathrm{R}^{N}$, is a vector of ones

3.

Now, we can define $Y = X_{pad} *_s W + b$, where the covolution of X_{pad} and W (filter) is along the last 3 axes of X_{pad} and the first 3 axes of W.
$W \in \mathrm{R}^{W_h \times W_w \times D \times O}$, $X_{pad} \in \mathrm{R}^{N \times X_h \times X_w \times D}$, $Y \in \mathrm{R}^{N \times Y_h \times Y_w \times O}$, $b \in \mathrm{R}^{O}$, where W_h is the height of the kernel, W_w is the width of the kernel, D is the input channel length, O is the output channel length, N is the batch size, X_h is the height of the input, X_w is the width of the input, Y_h is the height of the output, Y_w is the width of the output, and s is stride where $s_h = 1$ is the stride length for height and $s_w = 1$ is the stride length for width.

So, we can write the derivative of $loss$ w.r.t $W[i, j, d, o]$ and $b[o]$

$$\frac{\partial Y[n,k,l,o]}{\partial W[i,j,d,o]} = X_{pad}[n, k+i, l+j, d, o]$$

$$\frac{\partial Y[n,h,w]}{\partial b} = 1$$

$$\frac{\partial loss}{\partial W[i,j,d,o]} = \sum_{n=0}^{N-1} \sum_{k=0}^{Y_h-1} \sum_{l}^{Y_w-1} \frac{\partial loss}{\partial Y[n,k,l,o]} \cdot \frac{\partial Y[n,k,l,o]}{\partial W[i,j,d,o]} = \sum_{n=0}^{N-1} \sum_{k=0}^{Y_h-1} \sum_{l=0}^{Y_w-1} \frac{\partial loss}{\partial Y[n,k,l,o]} \cdot X_{pad}[n, k+i, l+j, d]$$

$$\frac{\partial loss}{\partial b[o]} = \sum_{n=0}^{N-1} \sum_{k=0}^{Y_h-1} \sum_{l=0}^{Y_w-1} \frac{\partial loss}{\partial Y[n,k,l,o]}$$

We can also write $\frac{\partial loss}{\partial W}$ as

$$\frac{\partial loss}{\partial W} = X_{pad} * \frac{\partial loss}{\partial Y}$$

where the convolution between X_{pad} and $\frac{\partial loss}{\partial Y}$ is along the first 3 axes of X_{pad} and the first 3 axes of $\frac{\partial loss}{\partial Y}$.

We can also write $\frac{\partial loss}{\partial b}$ as

$$\frac{\partial loss}{\partial b} = \frac{\partial loss}{\partial Y} * \mathbf{1}$$

where the convolution between $\frac{\partial loss}{\partial Y}$ and $\mathbf{1}$ is along all of axes of $\mathbf{1}$ and the first 3 axes of $\frac{\partial loss}{\partial Y}$, and $\mathbf{1} \in \mathrm{R}^{N \times Y_h \times Y_w}$ is a 3D tensor of ones.

We can also write the derivative of $loss$ w.r.t $X_{pad}[n, i, j, d]$

$$\frac{\partial Y[n,k,l,o]}{\partial X_{pad}[n,i,j,d]} = W[i-k, j-l, d, o] \cdot (0 \le i-k \le W_h - 1) \cdot (0 \le j-l \le W_w - 1)$$

$$\frac{\partial loss}{\partial X_{pad}[n,i,j,d]} = \sum_{k=i-W_h+1}^{i} \sum_{l=j-W_w+1}^{j} \sum_{o=0}^{O-1} \frac{\partial loss}{\partial Y[n,k,l,o]} \cdot \frac{\partial Y[n,k,l,o]}{\partial X[n,i,j,d]}$$

$$= \sum_{k=i-W_h+1}^{i} \sum_{l=j-W_w+1}^{j} \sum_{o=0}^{O-1} \frac{\partial loss}{\partial Y[n,k,l,o]} \cdot W[i-k, j-l, d, o]$$

We can also write $\frac{\partial loss}{\partial X_{pad}}$ as

$$\frac{\partial loss}{\partial X_{pad}} = \frac{\partial loss}{\partial Y} * W$$

where the convolution between $\frac{\partial loss}{\partial Y}$ and W is along the last 3 axes of $\frac{\partial loss}{\partial Y}$, and the first 2 and the last axis of W.

Now, we can crop $\frac{\partial loss}{\partial X_{pad}}$ so as to get $\frac{\partial loss}{\partial X}$

$$\frac{\partial loss}{\partial X} = \frac{\partial loss}{\partial X_{pad}}[:, p_h : -p_h, p_w : -p_w, :]$$

where p_h is the padding of height of X and p_w is the padding of width of X.

Q2 (8 pts): How do the results compare to the ones you got when implementing these models in TensorFlow?

1. [4 pts] For each of the models, please comment on any differences or discrepancies in results -- runtime, performance and stability in training and final performance. (This would be the place to justify design decisions in the implementation and their effects).
2. [2 pts] Which of the models show under-fitting?
3. [2 pts] Which of the models show over-fitting?

A2: (Your answer here)

1.

For model 1 with all of settings, the performance, stability and runtime of my model written in numpy are almost the same as the model written in tensoflow.

For model 2 with all of settings, the performance, stability and runtime of my model written in numpy are almost the same as the model written in tensoflow.

For model 3 with setting 1, the stability of my model written in numpy is a bit better than the model written in tensorflow, and the performance and runtime are almost the same as tensorflow. The reason may be caused by the different initialization values in our model compared with the ones in the tensorflow model (since the initialization is xavier, which can cause a various selection of initialization values). For model 3 with setting 2 and setting 3, the performance, runtime and stability of my model written in numpy are almost the same as the model written in tensorflow.

For model 4 with all of settings, the stability and performance of my model written in numpy is almost the same as the model written in tensorflow. However, the runtime of my model written in numpy is longer than the model written in tensorflow.

For the whole model, the basic programming logic is to build up each component of the model as a module, which includes a function **forward_pass**, a function **params_gradient** and a function **backward_pass**. Meanwhile, the updated variables and calculated gradients are stored in each module. Then, several components of modules defined before are composed together using a list called **graph**. After that, we also implement a module called **Model** including the list **graph**, which has a function called

forward_pass and a function called **backprop** so as to complete the procedures of forward passing and backpropagation. Finally, we define an optimizer module called **SGD** including the instance of **Model**, which has a function called **learning** that can operate the update of parameters. So, for training we just need to call this **learning** function, which is convenient.

About, the implementation of the loss function, we combine the softmax and cross entropy together and divide the maximal value to each output value, so that the numerical problem can be solved out. For linear module, the implementation should be simple, we just use the matrix multiplication to implement it. For convolution module and maxpooling module, we use two **for loop** to control the sliding window to finish the forward pass, backward pass and paramter gradients calculation, which is the same as what we derive in the above section. The implementation using two **for loop** may be the reason why our model is worse than tensorflow on runtime on CNN. To increase the performance, we set every variable to be **float64** so that the problem of overflow can be mitigated.

2.

Model 1 with setting 1, setting 2 and setting 3 are under-fitting.

Model 2 with setting 1 and setting 3 are under-fitting.

Model 3 with setting 3 is under-fitting.

Model 4 with setting 1 is under-fitting.

3.

Model 2 with setting 2 is over-fitting.

Model 3 with setting 2 is over-fitting.

Model 4 with setting 3 is over-fitting.