Homework 1 - Student Number 13064947 John Goodacre

Start date: *18th Jan 2017*

Due date: *04 February 2017, 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_hw1.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).

The Assignment

Objectives

You will use TensorFlow to implement several neural network models (labelled Model 1-4, and described in the corresponding sections of the Colab).

You will then train these models to classify hand written digits from the Mnist dataset.

Variable Initialization

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

```
initializer = tf.contrib.layers.xavier_initializer()
my_variable = tf.Variable(initializer(shape))
```

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 $z \in \mathbb{R}^{10}$ 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 $\{(x_i, y_i)\}_{i=1}^{N}$.

Note: Sum the loss across the elements of the batch with tf.reduce_sum().

Hint: read about TensorFlow's tf.nn.softmax_cross_entropy_with_logits (https://www.tensorflow.org/api_docs/python/tf/nn/softmax_cross_entropy_with_logits) function.

Optimization

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

Hint: read about TensorFlow's tf.train.GradientDescentOptimizer() (https://www.tensorflow.org/api_docs/python/tf/train/GradientDescentOptimizer).

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

If you need some extra help, familiarizing yourselves with the dataset and the task of building models in TensorFlow, you can check the TF tutorial for MNIST (https://www.tensorflow.org/tutorials/mnist/beginners/).

The tutorial will walk you through the MNIST classification task step-by-step, building and optimizing a model in TensorFlow.

(Please do not copy the provided code, though. Walk through the tutorial, but write your own implementation).

Imports and utility functions (do not modify!)

In [1]:

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

# 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(*setti
      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)
  plt.show() #JG I added this as the table was not showing
```

```
/Users/johngoodacre/anaconda/envs/tensorflow/lib/python3.6/importlib/_
bootstrap.py:205: RuntimeWarning: compiletime version 3.5 of module 't
ensorflow.python.framework.fast_tensor_util' does not match runtime ve
rsion 3.6
  return f(*args, **kwds)
/Users/johngoodacre/anaconda/envs/tensorflow/lib/python3.6/site-packag
es/h5py/__init__.py:34: FutureWarning: Conversion of the second argume
nt of issubdtype from `float` to `np.floating` is deprecated. In futur
e, it will be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

Model 1 (20 pts)

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 [2]:

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

In [3]:

```
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.
  tf.reset_default_graph()
  x, y_ = get_placeholders()
  mnist = get_data()
  eval_mnist = get_data()

  #20% of the training set
  #JG I would probably grab all my data pre doing anything rather than
  #putting it into a loop each time. But really just following the format
  #of the cells given and trying to plonk my code where it is probably expected.

  #In practise I would use train/ validate / test and probably choose a
  #random set to evaluate on for greater robustness. Here I just did a simply
  #grab the first 20 percent.
  twenty_per_cent = int(eval_mnist.train.num_examples*0.2)
  eval_train_images = eval_mnist.train.images[0:twenty_per_cent]
  eval_train_labels = eval_mnist.train.labels[0:twenty_per_cent]


  ########################################################
  # Define model, loss, update and evaluation metric. #
  ########################################################
  n_input = 784
  n_classes = 10

  #JG we were asked for xavier initialisation
  initializer = tf.contrib.layers.xavier_initializer()
  W = tf.Variable(initializer([n_input,n_classes]))
  b = tf.Variable(initializer([n_classes]))

  #simple linear regression gives the logits for the softmax
  y = tf.matmul(x,W) + b

  #make into two lines so it can be read when saved as a pdf
  loss = tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits(
                          labels=y_, logits=y))
  optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

  pred_probs = tf.nn.softmax(y)
  pred_labels = tf.argmax(pred_probs, 1)
  correct_pred = tf.equal(tf.argmax(y_, 1), pred_labels)
  acc = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

  init = tf.global_variables_initializer()

  # Train.
  i, train_accuracy, test_accuracy = 0, [], []
  log_period_updates = int(log_period_samples / batch_size)
  with tf.train.MonitoredSession() as sess:
    sess.run(init)

    while mnist.train.epochs_completed < num_epochs:

      # Update.
```

```
        i += 1

        batch_xs, batch_ys = mnist.train.next_batch(batch_size)


        #################
        # Training step #
        #################
        _, loss_val = sess.run([optimizer, loss], feed_dict={x:batch_xs,
                                                            y_:batch_ys})


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

          #####################################
          # Compute and store train accuracy. #
          #####################################
          train_acc = sess.run(acc, feed_dict={x:eval_train_images,
                                              y_:eval_train_labels})
          train_accuracy.append(train_acc)
          #####################################
          # Compute and store test accuracy.  #
          #####################################
          test_acc = sess.run(acc, feed_dict={x:eval_mnist.test.images,
                                             y_:eval_mnist.test.labels})
          test_accuracy.append(test_acc)

    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
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
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
```

Model 2 (20 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 [4]:

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

In [5]:

```
print('Training Model 2')

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

  # Reset graph, recreate placeholders and dataset.
  tf.reset_default_graph()  # reset the tensorflow graph
  x, y_ = get_placeholders()
  mnist = get_data()  # use for training.
  eval_mnist = get_data()  # use for evaluation.

  #20% of the training set - same as above
  twenty_per_cent = int(eval_mnist.train.num_examples*0.2)
  eval_train_images = eval_mnist.train.images[0:twenty_per_cent]
  eval_train_labels = eval_mnist.train.labels[0:twenty_per_cent]


  ########################################################
  # Define model, loss, update and evaluation metric. #
  ########################################################
  n_input = 784
  n_classes = 10
  n_hidden_1 = 32

  #xavier initialisation same as above
  initializer = tf.contrib.layers.xavier_initializer()

  # Model parameters
  weights = {
      'h1': tf.Variable(initializer([n_input, n_hidden_1])),
      'out': tf.Variable(initializer([n_hidden_1, n_classes]))
  }
  biases = {
      'b1': tf.Variable(initializer([n_hidden_1])),
      'out': tf.Variable(initializer([n_classes]))
  }

  #add a hidden layer with relu activation
  layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
  layer_1 = tf.nn.relu(layer_1)
  out_layer = tf.matmul(layer_1, weights['out']) + biases['out']

  #loss asked for reduce sum not mean
  loss = tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits(
                                                  labels=y_, logits=out_layer))
  optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

  pred_probs = tf.nn.softmax(out_layer)
  pred_labels = tf.argmax(pred_probs, 1)
  correct_pred = tf.equal(tf.argmax(y_, 1), pred_labels)
  acc = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

  init = tf.global_variables_initializer()

  # Train.
  i, train_accuracy, test_accuracy = 0, [], []
  log_period_updates = int(log_period_samples / batch_size)
  with tf.train.MonitoredSession() as sess:
    sess.run(init)
```

```
    while mnist.train.epochs_completed < num_epochs:


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

      #################
      # Training step #
      #################
      _, loss_val = sess.run([optimizer, loss], feed_dict={x:batch_xs,
                                                          y_:batch_ys})

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

        #####################################
        # Compute and store train accuracy. #
        #####################################
        train_acc = sess.run(acc, feed_dict={x:eval_train_images,
                                             y_:eval_train_labels})
        train_accuracy.append(train_acc)

        #####################################
        # Compute and store test accuracy.  #
        #####################################
        test_acc = sess.run(acc, feed_dict={x:eval_mnist.test.images,
                                            y_:eval_mnist.test.labels})
        test_accuracy.append(test_acc)

    experiments_task2.append(
        ((num_epochs, learning_rate), train_accuracy, test_accuracy))
```

```
Training Model 2
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
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
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
```

Model 3 (20 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 [6]:

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

In [7]:

```
print('Training Model 3')

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

  # Reset graph, recreate placeholders and dataset.
  tf.reset_default_graph()  # reset the tensorflow graph
  x, y_ = get_placeholders()
  mnist = get_data()  # use for training.
  eval_mnist = get_data()  # use for evaluation.

  #20% of the training set - same comments as for first model
  twenty_per_cent = int(eval_mnist.train.num_examples*0.2)
  eval_train_images = eval_mnist.train.images[0:twenty_per_cent]
  eval_train_labels = eval_mnist.train.labels[0:twenty_per_cent]


  ######################################################
  # Define model, loss, update and evaluation metric. #
  ######################################################
  n_input = 784
  n_classes = 10
  n_hidden_1 = 32
  n_hidden_2 = 32

  initializer = tf.contrib.layers.xavier_initializer()

  # Model parameters - JG now two hidden layers
  weights = {
      'h1': tf.Variable(initializer([n_input, n_hidden_1])),
      'h2': tf.Variable(initializer([n_hidden_1, n_hidden_2])),
      'out': tf.Variable(initializer([n_hidden_2, n_classes]))
  }
  biases = {
      'b1': tf.Variable(initializer([n_hidden_1])),
      'b2': tf.Variable(initializer([n_hidden_2])),
      'out': tf.Variable(initializer([n_classes]))
  }

  layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
  layer_1 = tf.nn.relu(layer_1)
  layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
  layer_2 = tf.nn.relu(layer_2)
  out_layer = tf.matmul(layer_2, weights['out']) + biases['out']

  loss = tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits(
                                                  labels=y_, logits=out_layer))
  optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

  pred_probs = tf.nn.softmax(out_layer)
  pred_labels = tf.argmax(pred_probs, 1)
  correct_pred = tf.equal(tf.argmax(y_, 1), pred_labels)
  acc = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

  init = tf.global_variables_initializer()


  # Train.
  i, train_accuracy, test_accuracy = 0, [], []
```

```
  log_period_updates = int(log_period_samples / batch_size)

  with tf.train.MonitoredSession() as sess:
    sess.run(init)
    while mnist.train.epochs_completed < num_epochs:

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

      #################
      # Training step #
      #################
      _, loss_val = sess.run([optimizer, loss], feed_dict={x:batch_xs,
                                                           y_:batch_ys})

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

        #####################################
        # Compute and store train accuracy. #
        #####################################
        train_acc = sess.run(acc, feed_dict={x:eval_train_images,
                                             y_:eval_train_labels})
        train_accuracy.append(train_acc)

        #####################################
        # Compute and store test accuracy.  #
        #####################################
        test_acc = sess.run(acc, feed_dict={x:eval_mnist.test.images,
                                            y_:eval_mnist.test.labels})
        test_accuracy.append(test_acc)

    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
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
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
```

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 [8]:

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

In [9]:

```
print('Training Model 4')

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

  # Reset graph, recreate placeholders and dataset.
  tf.reset_default_graph()  # reset the tensorflow graph
  x, y_ = get_placeholders()
  x_image = tf.reshape(x, [-1, 28, 28, 1])
  mnist = get_data()  # use for training.
  eval_mnist = get_data()  # use for evaluation.

  #20% of the training set
  twenty_per_cent = int(eval_mnist.train.num_examples*0.2)
  eval_train_images = eval_mnist.train.images[0:twenty_per_cent]
  eval_train_labels = eval_mnist.train.labels[0:twenty_per_cent]

  ######################################################
  # Define model, loss, update and evaluation metric. #
  ######################################################
  n_input = 784
  n_classes = 10

  initializer = tf.contrib.layers.xavier_initializer()

  #set up weights - two conv layers and dense
  weights = {
      'wconv1': tf.Variable(initializer([3, 3, 1, 8])),
      'wconv2': tf.Variable(initializer([3, 3, 8, 8])),
      'wdense1': tf.Variable(initializer([7*7*8, 32])),
      'out': tf.Variable(initializer([32, n_classes]))
  }

  biases = {
      'bconv1': tf.Variable(initializer([8])),
      'bconv2': tf.Variable(initializer([8])),
      'bdense1': tf.Variable(initializer([32])),
      'out': tf.Variable(initializer([n_classes]))
  }

  #first convolution
  conv1 = tf.nn.conv2d(x_image,weights['wconv1'],
                       strides=[1,1,1,1], padding='SAME')
  conv1 = tf.nn.bias_add(conv1, biases['bconv1'])
  #conv1 = tf.nn.relu(conv1) # JG I expected a relu in the question but wasnt there
  #so leave out
  maxpool1 = tf.nn.max_pool(conv1, ksize=[1,2,2,1],
                            strides=[1,2,2,1], padding='SAME')
  #second convolution
  conv2 = tf.nn.conv2d(maxpool1,weights['wconv2'],
                        strides=[1,1,1,1], padding='SAME')

  conv2 = tf.nn.bias_add(conv2, biases['bconv2'])
  #conv2 = tf.nn.relu(conv2)
  maxpool2 = tf.nn.max_pool(conv2, ksize=[1,2,2,1],
                            strides=[1,2,2,1], padding='SAME')

  #flatten the outputs for the fully connected inputs
  fullc1 = tf.reshape(maxpool2, [-1, weights['wdense1'].get_shape().as_list()[0]])
```

```
  fullc1 = tf.add(tf.matmul(fullc1, weights['wdense1']), biases['bdense1'])

  fullc1 = tf.nn.relu(fullc1)

  # Output, class prediction
  out_layer = tf.add(tf.matmul(fullc1, weights['out']), biases['out'])

  #loss asked for reduce sum not mean
  loss = tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits(labels=y_,
                                                               logits=out_layer))
  optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

  pred_probs = tf.nn.softmax(out_layer)
  pred_labels = tf.argmax(pred_probs, 1)
  correct_pred = tf.equal(tf.argmax(y_, 1), pred_labels)
  acc = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

  init = tf.global_variables_initializer()

  # Train.
  i, train_accuracy, test_accuracy = 0, [], []
  log_period_updates = int(log_period_samples / batch_size)
  with tf.train.MonitoredSession() as sess:
    sess.run(init)
    while mnist.train.epochs_completed < num_epochs:

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

      #################
      # Training step #
      #################
      _, loss_val = sess.run([optimizer, loss], feed_dict={x:batch_xs,y_:batch_ys})

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


        ######################################
        # Compute and store train accuracy. #
        ######################################
        train_acc = sess.run(acc, feed_dict={x:eval_train_images,
                                             y_:eval_train_labels})
        train_accuracy.append(train_acc)

        ######################################
        # Compute and store test accuracy.  #
        ######################################
        test_acc = sess.run(acc, feed_dict={x:eval_mnist.test.images,
                                            y_:eval_mnist.test.labels})
        test_accuracy.append(test_acc)

    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
```

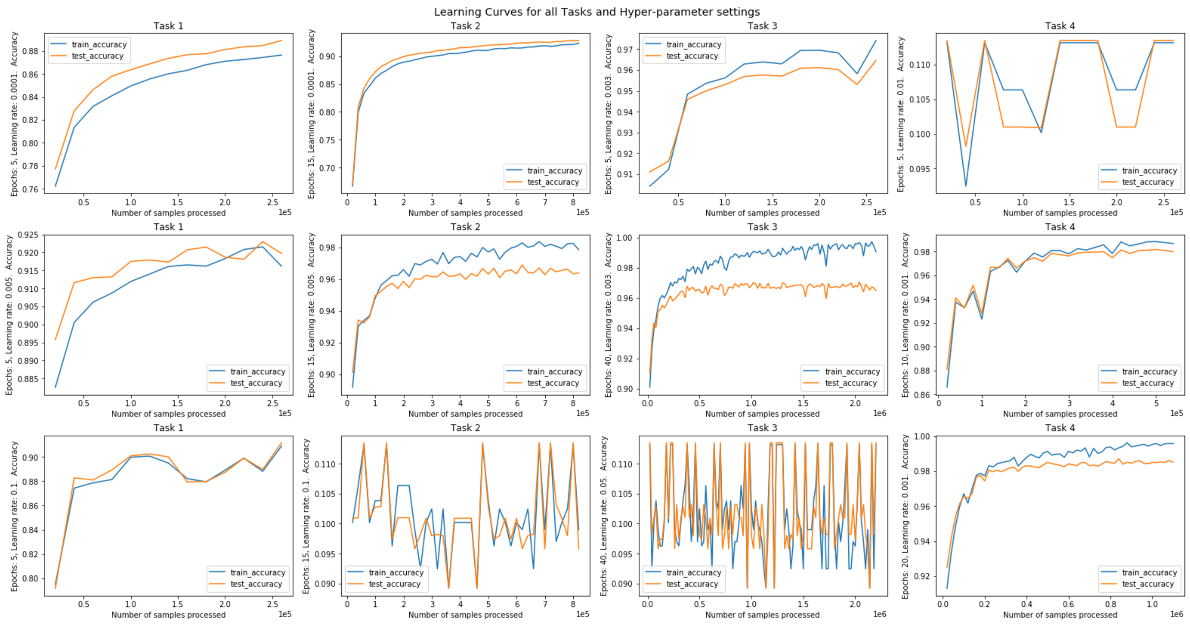
```
Extracting MNIST_data/train-labels-idx1-ubyte.gz

Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
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
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
```

Results

In [10]:

```
plot_learning_curves([experiments_task1, experiments_task2, experiments_task3, expe
```



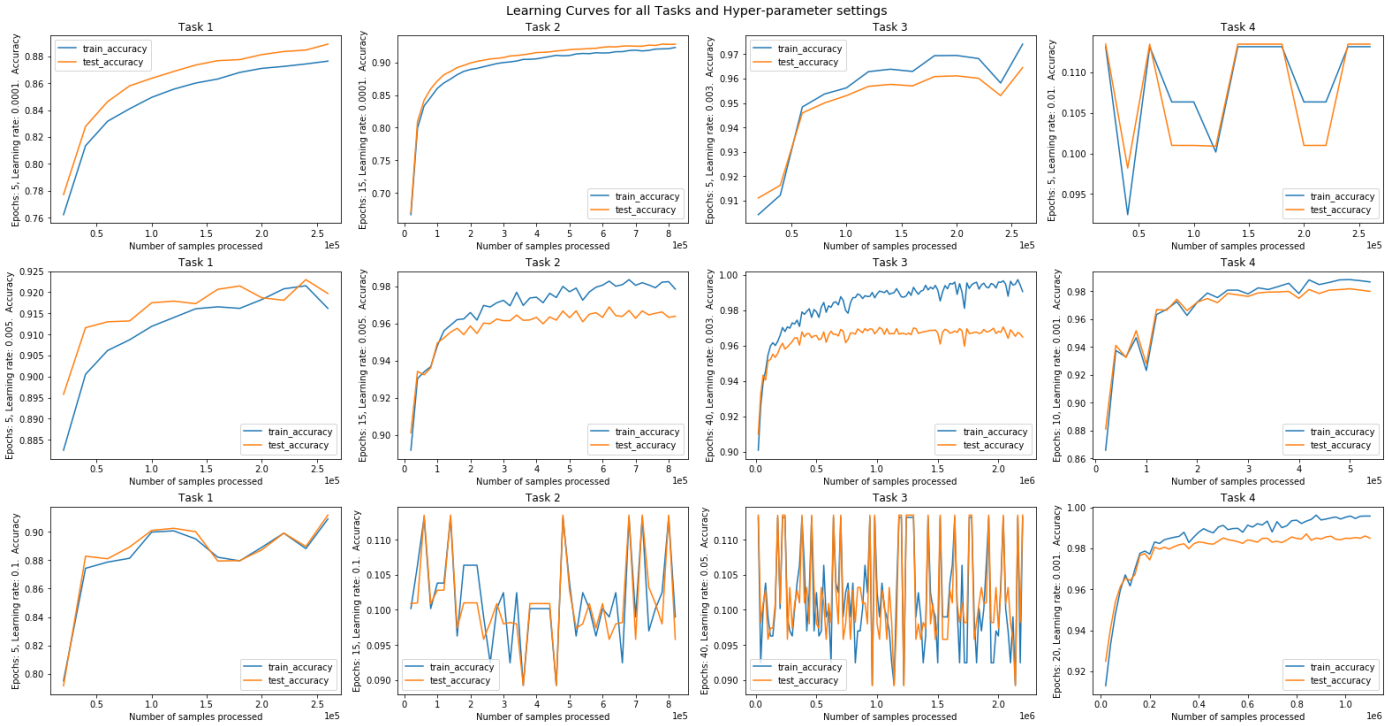
In [11]:

```
plot_summary_table([experiments_task1, experiments_task2, experiments_task3, experir
```

	Setting 1	Setting 2	Setting 3
Model 1	0.889	0.9197	0.9119
Model 2	0.928	0.964	0.0958
Model 3	0.9645	0.9649	0.1135
Model 4	0.1135	0.9799	0.985

Copy of graph and table outputs from previous run using this code

Due the question asking markers not to have to rerun our code in all files to see our outputs (image file saved as graphs.png, table as table.png in same directory as this .ipynb file)



	Setting 1	Setting 2	Setting 3
Model 1	0.889	0.9197	0.9119
Model 2	0.928	0.964	0.0958
Model 3	0.9645	0.9649	0.1135
Model 4	0.1135	0.9799	0.985

Questions

Q1 (5 pts): Indicate which of the previous experiments constitute an example of over-fitting. Why is this happening?

Overview of over/ underfitting

Over-fitting typically occurs in a setting where the model may be relatively complex with respect to the structure of the data, typically with a large number of parameters. If we fit the model to the data, the model may be perfectly capable of 'remembering' all the training data, but actually be very poor in terms of its generalisation capabilities.

This is related to the bias/ variance trade-off and is a factor in all of machine learning. Too simple a model leads to heavy bias. Too complex and we may fit perfectly, including to noise. When we begin to overfit we will typically see the following characteristics.

1) In terms of the data we will find a divergence between our accuracy on the training data and the test data. The training data may be learned to perfection but test accuracy begin to flatten and even reduce.

2) In terms of the characteristics of the model the coefficients of the gradients will have a greater magnitude, so the norm of the gradients will become larger (this is more easily visualised in a simple polynomial regression setting where with enough degrees of our polynomial we can fit the data perfectly, but with tortuously high twists and turns and thus large coeeficients). This latter point, is a clue to one of the anti-dotes to over-fitting.

The over-fitting/under-fitting issue is thus affected by the complexity of our model, the structure of our data, the learning rate, the number of epochs even how we optimise. A complex model with a good learning rate and enough epochs will simply learn the training data, but the test accuracy will have stalled. On the flip side a model not trained for enough epochs relative to its learning rate would benefit from more training and would thus be under-fitting.

To answer the question in this particular MNIST setting. We have 4 models.

Model 1)

This a very simple model akin to a logistic regression with 10 output classes. I did a number of experiments with different learning rates and number of epochs. I struggled to get a test set accuracy of greater than 0.92 for this model. I would argue that there was no over-fitting for model one in the examples given here. It is just not expressive enough. Note that the highest learning rate gave a far more unstable convergence, of which we will see more later. (The slowest learning rate showed signs of under-fitting which I will describe in the next section).

Conclusion: Model too weak - no overfitting

Model 2)

A more expressive fully connected model, but still relatively simple. With 15 epochs and a learning rate of 0.0001 we were clearly under-fitting. With 15 epochs and a learning rate of 0.005, I wasn't sure if we were over-fitting. The training accuracy is a lot higher than the test accuracy. But having run a number of experiments, it seems an accuracy of around 0.97 on the test set was close to as high as this model could manage. So although the training accuracy was a lot higher, the test accuracy hadn't deteriorated.

The final example failed to converge. The learning rate of 0.1 was too high. This is not really over or underfitting but a problem with the optimisation. With two high a learning rate then step sizes can be too high for optimising our error landscape.

Conclusion: (15,0.05) - possible over-fitting given training accuracy far higher than test. But the model possibly still isn't quite expressive enough and the test set accuracy hadn't really deteriorated.

Model 3)

This was the most expressive fully connected model. Again there was some under-fitting evidence and when the learning rate was too high a non-convergence. I therefore more closely examined the setting of 40 epochs and a learning rate of 0.003, to see if this was over-fitting. However I concluded this was not the case - first by adding more training epochs I could improve generalisation and second the results are lower than the simpler model 2, thus the model should be capable of doing slightly better. So I concluded there was no over-fitting for this model.

Conclusion: (40, 0.003) - Examined closely to check for over-fitting. However I concluded this was not the case based on the evidence that further epochs actually improved test set accuracy.

Model 4)

The conv net was more complex and gave the best results. Again we saw the lack of convergence with the high learning rate example. The question as regards overfitting was whether 20 epochs on a learning rate of 0.01 was too much. In my example this gave a test set accuracy of 0.986. Here the model was very expressive and I could achieve a training set accuracy of 1.0 given enough epochs (gross over-fitting). I experimented with 15-30 epochs. And had similar test set accuracies from 15-25 epochs. So by a small margin the model with 20 epochs may be slightly over-fitting, but the test accuracy hadn't deteriorated.

Conclusion: (20, 0.001) - Possible slight over-fitting, examined closely and had similar generalisation results between 15-25 epochs. The test set accuracies had not deteriorated by 20 epochs. But further training simply memorised the training data.

Q2 (5 pts): Indicate which of the previous experiments constitute an example of under-fitting. Why is this happening?

Model 1)

With 5 epochs and a learning rate of 0.0001 I would argue we are underfitting. This model is capable of at least a test set accuracy of 0.92. At this learning rate I increased the number of epochs from 5 to 80 and had a test set accuracy of 0.92. Thus I concluded this was under-fitting.

Conclusion: (5,0.0001) - Underfitting

Model 2)

This is a similar story. With 15 epochs and a learning rate of 0.0001 I achieved a test set accuracy of 0.93. I could continue training this model even to 80 epochs and have an accuracy of 0.954 with this model. Thus the learning rate was too low for the number of epochs.

Conclusion: (15, 0.0001) - Underfitting

Model 3)

With 5 epochs and a learning rate of 0.03, I believe we may be slightly underfitting. I achieved a test set accuracy of 0.97 by simply increasing the number of epochs - even to 80 epochs. One could even argue even 40 epochs may have slight underfitting, however here the changes here are very small and I would wish to do things more thoroughly (i.e. train/ validate etc etc). Or even better just stick dropout in there !

Conclusion: (5, 0.003) - Slight Underfitting

Model 4)

With 10 epochs and a learning rate of 0.001, we have evidence of underfitting. The model does very slightly better by increasing the number of epochs at least to 15 epochs. Again though given accuracy levels are now quite high the improvments are very small.

Conclusion: (10, 0.001) - Very slight underfitting

Q3 (10 pts): How would you prevent over-/under-fitting from happening?

Underfitting:

Given a learning rate which is not too high (i.e. actually converges to a sensible local minima), one can prevent under-ftting by training for more epochs.

Overfitting:

There are various ways of preventing over-fitting.

1) Regularisation

I mentioned the norm of the gradients can get larger as we over-fit, so we can simply add a penalisation term accounting for this in our error functional.

2) Dropout

Originally I was bothered by losing 'half the neurons in my brain' ! when first implementing dropout. However this intuition is not quite correct. The network is forced to learn more robust representations when using dropout. Yarin Gal has also done work showing a bayesian interpretation of dropout. In practise, dropout means when watching your training curves you can train a long time and often not have the validation accuracy reduce and the training set 'learned' - so one can have a bit more confidence that one doesn't have to stop training at some exact epoch !?

3) Early stopping

Clearly my interpretations above are by eye-balling the training accuracy and the test accuracy (i.e. is it increasing to 1, while the test accuracy is stalling?). One can simply code early stopping into the model with a lookback to stop the model when this effect is clearly occuring.

There are other methods for preventing over-fitting but I think these are probably the most important.

I would also mention the optimiser. In optimisation theory - pure gradient descent with fixed step sizes can be very poor (imagine the rosenbrock function where gradient descent jumps back and forth thousands of times across a valley). There are various methods of including curvature (newton or quasi newton given the computational expense) into the step direction to prevent this and also methods of annealing step sizes as one approaches an optimum. There are also various momentum methods one can use.

At the moment we do not have the same convergence guarantees as compared to convex optimisation and so there are heuristics but without total clarity. Although the optimiser choice is only indirectly related to over and under-fitting, it is also an implicit aspect as we need to choose learning rates, which is important then for the number of epochs and different optimisers appear to approach an optimum with varying speeds depending upon the specific problem we are addressing. Also I have found in practise that some optimisers such as adam or rmpprop simply seem to just do better.

However this is peripheral and I believe the most common answers will be the three I gave above.

Extension (Ungraded)

In the previous tasks you have used plain Stohastic Gradient Descent to train the models.

There is a large literatures on variants of Stochastic Gradient Descent, that improve learning speed and robustness to hyper-parameters.

Here (https://www.tensorflow.org/api_docs/python/train/optimizers) you can find the documentation for several optimizers already implemented in TensorFlow, as well as the original papers proposing these methods.*italicized text*.

AdamOptimizer and RMSProp are among the most commonly employed in Deep Learning.

How does replacing SGD with these optimizers affect the previous results?

In [14]:

```
# Feel free to experiment!
```

In [ ]:

```

```