Homework 2 - SN 13064947 John Goodacre

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

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(*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
```

Model 1 (10 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 [0]:

```
"""Implementation of the activation functions and their derivatives -
together with the cost function"""
def sigmoid(x): return 1/(1 + np.exp(-x))
def sigmoid_prime(x): return sigmoid(x) * (1 - sigmoid(x))

def relu(x): return np.maximum(x, 0)
def relu_prime(x): return np.greater(x, 0).astype(int)

def softmax(x):
    expx = np.exp(x)
    return expx / expx.sum(axis=1, keepdims=True)

def cost(y_truth, y_pred):
    #y_pred.shape[0] --- sum wanted , not mean in the cost function
    return -((y_truth * np.log(y_pred)).sum())
```

In [0]:

```
"""Implements a forward pass given input data X, weights and biases
Activation functions are also passed in - results are stored for use
in the backwards pass

This is for a fully connected network.
"""
def forward(X, weights, biases, act):
    a = [X]

    for w,b,f in zip(weights, biases, act):
        a.append(f[0](a[-1].dot(w) + b))
    return a
```

In [0]:

```
"""Returns the gradients of all the weights and biases of a fully
connected network

Given input data X and target data Y, and weights, biases and activation
functions implements a forward pass. The output layer error is calculated
specifically for cross-entropy

In the backward pass the derivatives of the activation functions are used
(thus act is a tuple where the first index has functions for the forward
pass and the second index their derivatives)"""
def gradients(X, Y, weights, biases, act):
    #forward
    a = forward(X, weights, biases, act)

    #backprop
    gradw = np.empty_like(weights)
    grad_bias = np.empty_like(biases)

    #output layer
    delta = a[-1] - Y
    gradw[-1] = a[-2].T.dot(delta)
    grad_bias[-1] = np.sum(delta, axis=0)

    for i in range(len(a)-2,0,-1):
        delta = act[i-1][1](a[i]) * delta.dot(weights[i].T)
        grad_bias[i-1] = np.sum(delta, axis=0)
        gradw[i-1] = a[i-1].T.dot(delta)

    grad_bias = grad_bias / len(X)
    gradw = gradw / len(X)

    return gradw, grad_bias
```

In [0]:

```
# 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 [0]:

```
print('Training Model 1')
"""Implementation of the first Model xavier initialisation and simply
a linear layer and softmax I use an arbitrary 20% of the training set
for reporting purposes"""
# Train Model 1 with the different hyper-parameter settings.
for (num_epochs, learning_rate) in settings:

    #xavier initialisation


    weights = [np.random.uniform(-np.sqrt(6./(w[0] + w[1])), np.sqrt(6./(w[0] + w[1]
                                                                  for w in [(784, 10)]]
    biases = [np.random.uniform(-np.sqrt(3./(b[0])), np.sqrt(3./(b[0])), b)
                                                                  for b in [ (10,)]]
    activations = [(softmax, None)]

    batch_size= 100

    tr_loss = []
    ts_loss = []
    tr_acc = []
    ts_acc = []

    mnist = get_data()
    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]

    log_period_updates = int(log_period_samples / batch_size)

    i, train_accuracy, test_accuracy = 0, [], []

    while mnist.train.epochs_completed < num_epochs:

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

        #The guts of the training - a batch is passed in and the gradients for weigl
        #and biases passed back which is then updated using batch stochastic gradier
        #with the learning rate provided
        gradw, grad_bias = gradients(batch_xs, batch_ys, weights, biases, activatior
        biases -= learning_rate * grad_bias
        weights -= learning_rate * gradw

        if i % log_period_updates == 0:
            #Forward pass to grab predictions for train and test
            trpreds = forward(eval_train_images, weights, biases, activations)[-1]
            tstpreds = forward(eval_mnist.test.images, weights, biases, activations)

            #One hot encoding of predictions
            trpred_onehot = np.argmax(trpreds, axis=1)
            tstpred_onehot = np.argmax(tstpreds, axis=1)

            #Accuracy evaluation
            train_acc = np.mean(trpred_onehot == np.argmax(eval_train_labels, axis=1
```

```
            test_acc = np.mean(tstpred_onehot == np.argmax(eval_mnist.test.labels, a

            train_accuracy.append(train_acc)
            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 (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 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 [0]:

```
print('Training Model 2')
"""Implementation of the second Model xavier initialisation, uses the same
functions I defined above just with another layer - relu and the derivative of relu
also passed in followed by a linear layer and softmax I use an arbitrary 20% of the
training set for reporting purposes

Given the functions I wrote above are quite general for fully connected layer the or
real change is the set of weights, activations and derivatives I pass in"""
for (num_epochs, learning_rate) in settings:


    #xavier initialisation


    weights = [np.random.uniform(-np.sqrt(6./(w[0] + w[1])), np.sqrt(6./(w[0] + w[1]
                                                          for w in [(784, 32),
    biases = [np.random.uniform(-np.sqrt(3./(b[0])), np.sqrt(3./(b[0])), b)
                                                          for b in [(32,), (10,

    activations = [ (relu, relu_prime), (softmax, None)]

    batch_size= 100

    tr_loss = []
    ts_loss = []
    tr_acc = []
    ts_acc = []

    mnist = get_data()
    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]

    log_period_updates = int(log_period_samples / batch_size)

    i, train_accuracy, test_accuracy = 0, [], []

    while mnist.train.epochs_completed < num_epochs:
        i += 1
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)

        #The guts of the training - a batch is passed in and the gradients for weigl
        #and biases passed back which is then updated using batch stochastic gradier
        #with the learning rate provided
        gradw, grad_bias = gradients(batch_xs, batch_ys, weights, biases, activation
        biases -= learning_rate * grad_bias
        weights -= learning_rate * gradw

        if i % log_period_updates == 0:
            #Forward pass to grab predictions for train and test
            trpreds = forward(eval_train_images, weights, biases, activations)[-1]
            tstpreds = forward(eval_mnist.test.images, weights, biases, activations)

            #One hot encoding of predictions
            trpred_onehot = np.argmax(trpreds, axis=1)
            tstpred_onehot = np.argmax(tstpreds, axis=1)
```

```
            #Accuracy evaluation
            train_acc = np.mean(trpred_onehot == np.argmax(eval_train_labels, axis=1
            test_acc = np.mean(tstpred_onehot == np.argmax(eval_mnist.test.labels, a
            tr_acc.append(train_acc)
            ts_acc.append(test_acc)

            train_accuracy.append(train_acc)
            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 (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 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 [0]:

```
print('Training Model 3')
"""Implementation of the third Model, xavier initialisation. Again I
repeat given the functions I wrote  above are quite general for fully connected laye
the only real change is the set of weights, activations and derivatives I pass in
- clearly this goes one layer deeper"""
for (num_epochs, learning_rate) in settings:

    #xavier initialisation


    weights = [np.random.uniform(-np.sqrt(6./(w[0] + w[1])), np.sqrt(6./(w[0] + w[1]
                                                                  for w in [(784, 32),
    biases = [np.random.uniform(-np.sqrt(3./(b[0])), np.sqrt(3./(b[0])), b)
                                                                  for b in [(32,),(32,)
    activations = [ (relu, relu_prime), (relu, relu_prime), (softmax, None)]

    batch_size= 100

    tr_loss = []
    ts_loss = []
    tr_acc = []
    ts_acc = []

    mnist = get_data()
    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]

    log_period_updates = int(log_period_samples / batch_size)

    i, train_accuracy, test_accuracy = 0, [], []

    while mnist.train.epochs_completed < num_epochs:

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

        #The guts of the training - a batch is passed in and the gradients for weigl
        #and biases passed back which is then updated using batch stochastic gradier
        #with the learning rate provided
        gradw, grad_bias = gradients(batch_xs, batch_ys, weights, biases, activation
        biases -= learning_rate * grad_bias
        weights -= learning_rate * gradw

        if i % log_period_updates == 0:

            #Forward pass to grab predictions for train and test
            trpreds = forward(eval_train_images, weights, biases, activations)[-1]
            tstpreds = forward(eval_mnist.test.images, weights, biases, activations)

            #One hot encoding of predictions
            trpred_onehot = np.argmax(trpreds, axis=1)
            tstpred_onehot = np.argmax(tstpreds, axis=1)

            #Accuracy evaluation
            train_acc = np.mean(trpred_onehot == np.argmax(eval_train_labels, axis=1
```

```
            test_acc = np.mean(tstpred_onehot == np.argmax(eval_mnist.test.labels, a

            train_accuracy.append(train_acc)
            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 [0]:

```
"""Please note in the interests of honesty and of course non-plagiarism
much of the conv net module implementation in this cell is not my original code -
all previous code is my own from scratch and the code after this cell is my
implementation. So my marks for this part of the conv implementation should
of course reflect that.

--------------------------------------------------------------------------------
I found that I was able to implement a conv net but that this was tortuously
slow. Basically a bunch of nested loops. So I tried another path, clearly its
about vectorisation and python skills.

After searching on the internet I found that within matlab in image processing
there is a function im2col, basically enabling a convolution with some reshaping
to be a straight matrix multiplication.

This definitely felt the way to go. I then referred to the Stanford course CS231.
They provided their students with implementations of im2col within python as
part of their exercises.

The functions in this cell were then grabbed and adapted by myself because I still
wanted to actually go through the process of understanding and attempt to putting
the conv net together.  And walk away with useful code and understanding. However as
I said it is not my own - I believe I can and could implement all of this now. But
not without having seen the code below first unfortunately.


To be specific as some parts in this cell are quite straightforward - it is the im2c
stuff and a little bit combining that with the conv-forward and backward (although t
part I completely  understand). I was really initially a bit confused by dimensional
and reshaping. I guess my numpy needs work.
--------------------------------------------------------------------------------
I could of course have spent time refactoring this code and pretending to make it my
But I think thats  pretty pathetic from a learning point of view frankly and prefer
open about where I struggled in the time I had.
"""

def conv_forward(X, W, b, stride=1, padding=1):
    cache = W, b, stride, padding
    n_filters, d_filter, h_filter, w_filter = W.shape
    n_x, d_x, h_x, w_x = X.shape
    h_out = (h_x - h_filter + 2 * padding) / stride + 1
    w_out = (w_x - w_filter + 2 * padding) / stride + 1

    if not h_out.is_integer() or not w_out.is_integer():
        raise Exception('Invalid output dimension!')

    h_out, w_out = int(h_out), int(w_out)

    X_col = im2col_indices(X, h_filter, w_filter, padding=padding, stride=stride)
    W_col = W.reshape(n_filters, -1)

    out = W_col @ X_col + b
    out = out.reshape(n_filters, h_out, w_out, n_x)
    out = out.transpose(3, 0, 1, 2)

    cache = (X, W, b, stride, padding, X_col)

    return out, cache
```

```
def conv_backward(dout, cache):
    X, W, b, stride, padding, X_col = cache
    n_filter, d_filter, h_filter, w_filter = W.shape

    db = np.sum(dout, axis=(0, 2, 3))
    db = db.reshape(n_filter, -1)

    dout_reshaped = dout.transpose(1, 2, 3, 0).reshape(n_filter, -1)
    dW = dout_reshaped @ X_col.T
    dW = dW.reshape(W.shape)

    W_reshape = W.reshape(n_filter, -1)
    dX_col = W_reshape.T @ dout_reshaped
    dX = col2im_indices(dX_col, X.shape, h_filter, w_filter, padding=padding, stride

    return dX, dW, db

def maxpool_forward(X, size=2, stride=2):
    def maxpool(X_col):
        max_idx = np.argmax(X_col, axis=0)
        out = X_col[max_idx, range(max_idx.size)]
        return out, max_idx

    return _pool_forward(X, maxpool, size, stride)


def maxpool_backward(dout, cache):
    def dmaxpool(dX_col, dout_col, pool_cache):
        dX_col[pool_cache, range(dout_col.size)] = dout_col
        return dX_col

    return _pool_backward(dout, dmaxpool, cache)


def _pool_forward(X, pool_fun, size=2, stride=2):
    n, d, h, w = X.shape
    h_out = (h - size) / stride + 1
    w_out = (w - size) / stride + 1

    if not w_out.is_integer() or not h_out.is_integer():
        raise Exception('Invalid output dimension!')

    h_out, w_out = int(h_out), int(w_out)

    X_reshaped = X.reshape(n * d, 1, h, w)
    X_col = im2col_indices(X_reshaped, size, size, padding=0, stride=stride)

    out, pool_cache = pool_fun(X_col)

    out = out.reshape(h_out, w_out, n, d)
    out = out.transpose(2, 3, 0, 1)

    cache = (X, size, stride, X_col, pool_cache)

    return out, cache


def _pool_backward(dout, dpool_fun, cache):
    X, size, stride, X_col, pool_cache = cache
    n, d, w, h = X.shape
```

```
    dX_col = np.zeros_like(X_col)
    dout_col = dout.transpose(2, 3, 0, 1).ravel()

    dX = dpool_fun(dX_col, dout_col, pool_cache)

    dX = col2im_indices(dX_col, (n * d, 1, h, w), size, size, padding=0, stride=stri
    dX = dX.reshape(X.shape)

    return dX

def fc_forward(X, W, b):
    out = X @ W + b
    cache = (W, X)
    return out, cache

def fc_backward(dout, cache):
    W, h = cache

    dW = h.T @ dout
    db = np.sum(dout, axis=0)
    dX = dout @ W.T

    return dX, dW, db

def relu_forward(X):
    out = np.maximum(X, 0)
    cache = X
    return out, cache

def relu_backward(dout, cache):
    dX = dout.copy()
    dX[cache <= 0] = 0
    return dX

def get_im2col_indices(x_shape, field_height, field_width, padding=1, stride=1):
    # First figure out what the size of the output should be
    N, C, H, W = x_shape
    assert (H + 2 * padding - field_height) % stride == 0
    assert (W + 2 * padding - field_height) % stride == 0
    out_height = int((H + 2 * padding - field_height) / stride + 1)
    out_width = int((W + 2 * padding - field_width) / stride + 1)

    i0 = np.repeat(np.arange(field_height), field_width)
    i0 = np.tile(i0, C)
    i1 = stride * np.repeat(np.arange(out_height), out_width)
    j0 = np.tile(np.arange(field_width), field_height * C)
    j1 = stride * np.tile(np.arange(out_width), out_height)
    i = i0.reshape(-1, 1) + i1.reshape(1, -1)
    j = j0.reshape(-1, 1) + j1.reshape(1, -1)

    k = np.repeat(np.arange(C), field_height * field_width).reshape(-1, 1)

    return (k.astype(int), i.astype(int), j.astype(int))


def im2col_indices(x, field_height, field_width, padding=1, stride=1):
    """ An implementation of im2col based on some fancy indexing """
    # Zero-pad the input
    p = padding
    x_padded = np.pad(x, ((0, 0), (0, 0), (p, p), (p, p)), mode='constant')
```

```
    k, i, j = get_im2col_indices(x.shape, field_height, field_width, padding, stride

    cols = x_padded[:, k, i, j]
    C = x.shape[1]
    cols = cols.transpose(1, 2, 0).reshape(field_height * field_width * C, -1)
    return cols


def col2im_indices(cols, x_shape, field_height=3, field_width=3, padding=1,
                   stride=1):
    """ An implementation of col2im based on fancy indexing and np.add.at """
    N, C, H, W = x_shape
    H_padded, W_padded = H + 2 * padding, W + 2 * padding
    x_padded = np.zeros((N, C, H_padded, W_padded), dtype=cols.dtype)
    k, i, j = get_im2col_indices(x_shape, field_height, field_width, padding, stride
    cols_reshaped = cols.reshape(C * field_height * field_width, -1, N)
    cols_reshaped = cols_reshaped.transpose(2, 0, 1)
    np.add.at(x_padded, (slice(None), k, i, j), cols_reshaped)
    if padding == 0:
        return x_padded
    return x_padded[:, :, padding:-padding, padding:-padding]
```

In [0]:

```
"""This is the implementation of a forward backward pass for the conv net
As the name suggests it does one forward and one backward pass of the conv net
+ fully connected and returns the gradients for all the weights"""
def forward_backward(x_image, batch_ys, weights):
    #forward

    #Given the modular approach above this is a cleaner implementation and indeed co
    #be generalised further - (for pedagogical purposes I wanted to see each layer)
    h1, h1_cache = conv_forward(x_image, weights[0],weights[1])
    hpool1, hpool_cache1 = maxpool_forward(h1)

    h2, h2_cache = conv_forward(hpool1, weights[2],weights[3])
    hpool2, hpool_cache2 = maxpool_forward(h2)

    h3 = hpool2.ravel().reshape(x_image.shape[0],-1) #reshape for the fully connecte

    h4, h4_cache = fc_forward(h3, weights[4], weights[5])
    h4, n4_cache = relu_forward(h4)

    # Softmax
    score, score_cache = fc_forward(h4, weights[6], weights[7])

    #Predictions
    probs = softmax(score)

    #backward pass
    #print(cost(batch_ys,probs)) - check its learning !
    dh4, dW4, db4 = fc_backward(probs - batch_ys, score_cache) #Hard coded for xent
    dh4 = relu_backward(dh4, n4_cache) #relu backward pass

    dh3, dW3, db3 = fc_backward(dh4, h4_cache)

    dh3 = dh3.ravel().reshape(hpool2.shape) #end of the fully connected layer now re
    dpool2 = maxpool_backward(dh3, hpool_cache2)

    dh2, dW2, db2 = conv_backward(dpool2, h2_cache)

    dpool1 = maxpool_backward(dh2, hpool_cache1)
    dh1, dW1, db1 = conv_backward(dpool1, h1_cache)


    gradients = np.array([dW1,db1,dW2,db2,dW3,db3,dW4,db4]) #Returns all the gradien

    return gradients
```

In [0]:

```
"""A bit of repetition but this is the implementation of the forward predictions -
here the inputs are the reshaped input images and the network weights

Outputs are prediction probabilities"""
def forward_pred_conv(x_image, weights):
    #forward

    h1, h1_cache = conv_forward(x_image, weights[0],weights[1]) #forward through co
    hpool1, hpool_cache1 = maxpool_forward(h1) #forward through pooling module

    h2, h2_cache = conv_forward(hpool1, weights[2],weights[3])
    hpool2, hpool_cache2 = maxpool_forward(h2)

    h3 = hpool2.ravel().reshape(x_image.shape[0],-1) #reshape for fully connected la

    h4, h4_cache = fc_forward(h3, weights[4], weights[5]) #fully connected layer
    h4, n4_cache = relu_forward(h4)

    # Softmax
    score, score_cache = fc_forward(h4, weights[6], weights[7])
    probs = softmax(score)
    return probs
```

In [0]:

```
# 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)] #5,10,20
```

In [0]:

```
print('Training Model 4')
"""Implementation of the fourth Model, xavier initialisation. The skeleton is
again very similar to the previous models - albeit with different modules and
forward and forward-backward passes defined"""

# 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.

  #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]

  eval_train_image = np.reshape(eval_train_images, [-1, 1, 28, 28])
  eval_test_image = np.reshape(eval_mnist.test.images, [-1, 1, 28, 28])

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

  n_input = 784
  n_classes = 10

  #[filter_height, filter_width, input_depth, output_depth

  #xavier initialisation of weights
  #not very elegant - manually input and calculate fan in and fan out for this examp

  wconv1 = np.random.uniform(-np.sqrt(6./(8*3*3*1 +8)), np.sqrt(6./(8*3*3*1 +8)), (8
  bconv1 = np.random.uniform(-np.sqrt(3./8), np.sqrt(3./8), (8,1))

  wconv2 = np.random.uniform(-np.sqrt(6./(8*3*3*8 +8)), np.sqrt(6./(8*3*3*8 +8)), (8
  bconv2 = np.random.uniform(-np.sqrt(3./8), np.sqrt(3./8), (8,1))

  wfull1 = np.random.uniform(-np.sqrt(6./(7*7*8 +  32)), np.sqrt(6./(7*7*8 +  32)),
  bdense1 = np.random.uniform(-np.sqrt(3./32), np.sqrt(3./32), (1,32))

  wout = np.random.uniform(-np.sqrt(6./(n_classes +  32)), np.sqrt(6./(n_classes +
  bout = np.random.uniform(-np.sqrt(3./32), np.sqrt(3./n_classes), (1,n_classes))


  weights = np.array([wconv1, bconv1,wconv2,bconv2,wfull1,bdense1,wout,bout])


  # 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)
    x_image = np.reshape(batch_xs, [-1, 1, 28, 28])


    #################
```

```
    # Training step #
    ##################
    #The guts of the training - a batch is passed in and the gradients for weights
    #and biases passed back which is then updated using batch stochastic gradient de
    #with the learning rate provided - the structure is slightly cleaner than the pr
    gradients = forward_backward(x_image, batch_ys, weights)
    weights -= learning_rate*gradients

    # Periodically evaluate.
    if i % log_period_updates == 0:
      ######################################
      # Compute and store train accuracy. #
      ######################################

      #Forward pass to grab predictions for train and test
      trpreds = forward_pred_conv(eval_train_image, weights)
      tstpreds = forward_pred_conv(eval_test_image, weights)

      #One hot encoding of predictions
      trpred_onehot = np.argmax(trpreds, axis=1)
      tstpred_onehot = np.argmax(tstpreds, axis=1)

      #Accuracy evaluation
      train_acc = np.mean(trpred_onehot == np.argmax(eval_train_labels, axis=1))
      test_acc = np.mean(tstpred_onehot == np.argmax(eval_mnist.test.labels, axis=1)

      train_accuracy.append(train_acc)

      ######################################
      # Compute and store test accuracy.  #
      ######################################
      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
```

```
/Users/johngoodacre/anaconda/envs/tensorflow/lib/python3.6/site-packag
es/ipykernel_launcher.py:9: RuntimeWarning: overflow encountered in ex
p
  if __name__ == '__main__':
/Users/johngoodacre/anaconda/envs/tensorflow/lib/python3.6/site-packag
es/ipykernel_launcher.py:10: RuntimeWarning: invalid value encountered
in true_divide
  # Remove the CWD from sys.path while we load stuff.
/Users/johngoodacre/anaconda/envs/tensorflow/lib/python3.6/site-packag
es/ipykernel_launcher.py:145: RuntimeWarning: invalid value encountere
d in less_equal
```

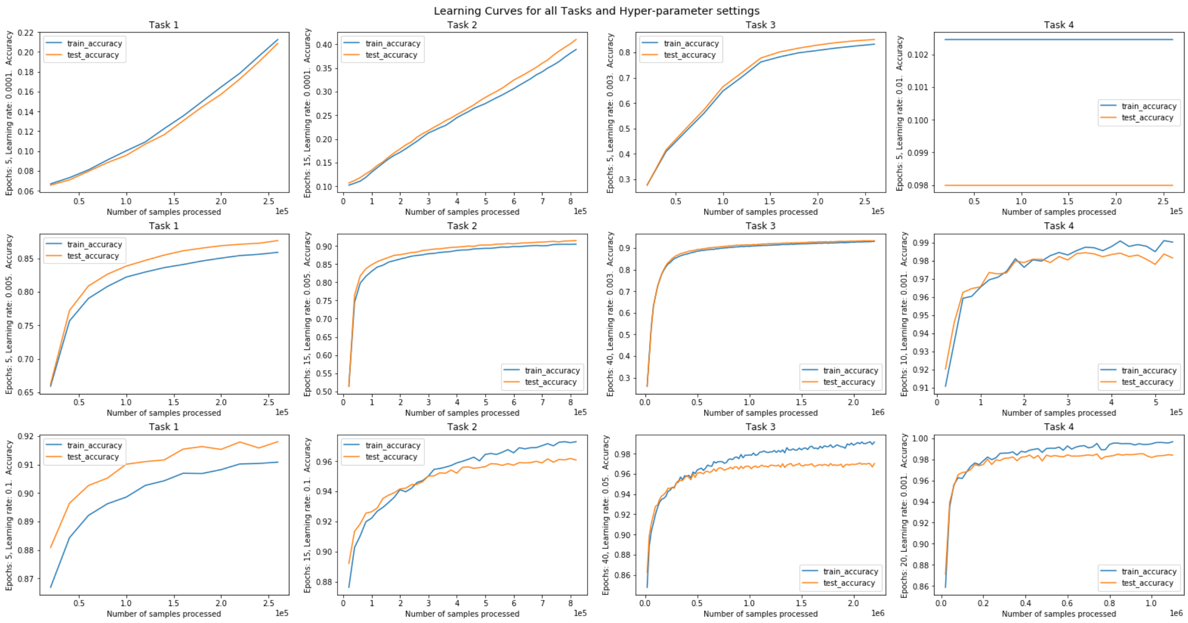
```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

```
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 [0]:

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



In [0]:

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

	Setting 1	Setting 2	Setting 3
Model 1	0.2086	0.8766	0.918
Model 2	0.4102	0.9148	0.9608
Model 3	0.8507	0.9344	0.9703
Model 4	0.098	0.9816	0.9839

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

1)

Apologies for the formatting - I tend to use Sharelatex but this is the first time I have used Latex in IPython notebook cells (it doesn't seem to automatically be as pretty). Given the cross-entropy loss above, compute the derivative of the loss function with respect to the scores z (the input to the softmax layer).

Let $\mathscr{L} = XentLogits(t, y)$, where given k classes $\mathscr{L} = -\sum_k t_k log y_k$

and $y_k = \frac{e^{z_k}}{\sum_m e^{z_m}}$. Clearly this is an individual loss, for the total loss we will sum over all the individual losses for the mini-batch. Taking the derivative of the softmax, we will have two cases depending upon whether $i = k$.

$\therefore \frac{\partial y_k}{\partial z_i} = \frac{\sum_m e^{z_m} e^{z_i} - e^{z_k} e^{z_i}}{(\sum_m e^{z_m})^2}$, in the case where $i = k$, (Equation 1a). We also have that:

$\frac{\partial y_k}{\partial z_i} = \frac{-e^{z_k} e^{z_i}}{(\sum_m e^{z_m})^2}$, when $i \neq k$. (Equation 1b)

Thus $\frac{\partial y_k}{\partial z_i} = y_i(1 - y_k)$, if $i = k$ and $\frac{\partial y_k}{\partial z_i} = -y_i y_k$ otherwise.

We are looking for $\frac{\partial \mathscr{L}}{\partial z_i} = -\sum_k t_k \frac{\partial}{\partial z_i} \log y_k$, so $\frac{\partial \mathscr{L}}{\partial z_i} = -\sum_k \frac{t_k}{y_k} \frac{\partial y_k}{\partial z_i}$. Using equations 1a, 1b and separating out $k = i$ we have that:

$\frac{\partial \mathscr{L}}{\partial z_i} = \sum_{k \neq i} \frac{t_k}{y_k} y_i y_k - t_i(1 - y_i t_i)$

$\therefore \frac{\partial \mathscr{L}}{\partial z_i} = \sum_k t_k y_i - t_i = y_i - t_i$. Therefore in vector form we have that $\frac{\partial \mathscr{L}}{\partial z_i} = y - t$, where in the case of say MNIST this would be a 1x10 dimension object (where errors are added for mini-batches).

2)

Consider the first model (M1: linear + softmax). Compute the derivative of the loss with respect to the inputs, weights and bias.

2a)

$\frac{\partial \mathscr{L}}{\partial x_j} = \sum_i \frac{\partial \mathscr{L}}{\partial z_i} \frac{\partial z_i}{\partial x_j}$

but given $z_i = \sum_j W_{i,j} x_j + b_j$,we have that $\frac{\partial \mathscr{L}}{\partial x_j} = \sum_i \frac{\partial \mathscr{L}}{\partial z_i} W_{i,j}$

Therefore in vector form $\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial y} W = (y - t)W$

2b)

$\frac{\partial \mathscr{L}}{\partial W_{j,k}} = \sum_i \frac{\partial \mathscr{L}}{\partial z_i} \frac{\partial z_i}{\partial W_{j,k}} = \sum_i \frac{\partial \mathscr{L}}{\partial z_i} x_k \delta_{ij} = \frac{\partial \mathscr{L}}{\partial z_j} x_k$

In vector form $\frac{\partial \mathscr{L}}{\partial W} = \left(\frac{\partial \mathscr{L}}{\partial z}\right)^T x^T = (y - t)^T x^T$

2c)

$\frac{\partial z_i}{\partial b_j} = \delta_{ij}$, therefore $\frac{\partial \mathscr{L}}{\partial b} = \frac{\partial \mathscr{L}}{\partial z} = y - t$

Again note that depending upon whethere we do stochastic gradient descent and use one example or a mini-batch we would sum over the mini-batch samples.

---

3)

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.

Notation can be a little painful here, so I will start by defining some terms. I am interpreting the question as being purely about the convolution layer. Thus the input x is post any activation function (in the examples given it was linear, but this would not usually be the case). Again I assume the output y from the convolution layer is prior to the next module which would likely be a non-linear activation function followed by pooling.

I am using $\mathscr{L}$ as my loss and presuming that through back-prop we will know the derivative of the loss with respect to the output Y of the convolution layer. From this point I will calculate the derivative with respect to the weights W and input X.

In terms of coordinates - I will use i,j to find my way around the kernel (one channel to begin with), the output coordinates I will use i', j'. The height and width of the kernel are H,W (not to be mistaken with my choice of W for the weights).

One Channel

3a) One channel - $\frac{\partial \mathscr{L}}{\partial W}$

For a forward pass through the convolution layer we thus have:

$Y_{i',j'} = \sum_i^H \sum_j^W W_{i,j} X_{i'+i-1,j'+j-1} + b,$

$\frac{\partial \mathscr{L}}{\partial W_{i,j}} = \sum_{i'}^H \sum_{j'}^W \frac{\partial \mathscr{L}}{\partial Y_{i',j'}} \frac{\partial Y_{i',j'}}{\partial W_{i,j}} = \sum_{i'}^H \sum_{j'}^W \frac{\partial \mathscr{L}}{\partial Y_{i',j'}} X_{i'+i-1,j'+j-1}$, and we know $\frac{\partial \mathscr{L}}{\partial Y_{i',j'}}$ through backprop to this layer. Note that this itself is another convolution.

3b) One channel - $\frac{\partial \mathscr{L}}{\partial X}$

Similarly $\frac{\partial \mathscr{L}}{\partial X_{i,j}} = \sum_{i',j'} \frac{\partial \mathscr{L}}{\partial Y_{i',j'}} \frac{\partial Y_{i',j'}}{\partial X_{i,j}}$, therefore we have that:

$\frac{\partial \mathscr{L}}{\partial X_{i,j}} = \sum_{i',j'} \frac{\partial \mathscr{L}}{\partial Y_{i',j'}} \frac{\partial}{\partial X_{i,j}} \left(\sum_{m,n} W_{m,n} X_{i'+m-1,j'+n-1} + b \right)$, running out of notation so using m,n for the double sum indices.

However this derivative only has non-zero values when $i = i' + m - 1$ and $j = j' + n - 1'$

Therefore, $m = i - i' + 1$ and $n = j - j' + 1$ and

$\frac{\partial \mathscr{L}}{\partial X_{i,j}} = \sum_{i',j'} \frac{\partial \mathscr{L}}{\partial Y_{i',j'}} W_{i-i'+1,j-j'+1}$, again a convolution.

Multiple Channels - adding another dimension

Keeping the same notational format let d be the number of channels in the previous layer and d' the number of layers in the next. This is similar to the prior reasoning but there are more sub-scripts to keep track of.

$Y_{i',j',d'} = \sum_i^H \sum_j^W \sum_d^D W_{i,j,d,d'} X_{i'+i-1,j'+j-1,d} + b_{d'}$, again taking derivatives with respect to the weights.

3c) Multiple channels - $\frac{\partial \mathscr{L}}{\partial W}$

$\frac{\partial \mathscr{L}}{\partial W_{i,j,d,d'}} = \sum_{i',j',d'} \frac{\partial \mathscr{L}}{\partial Y_{i',j',d'}} \frac{\partial Y_{i',j',d'}}{\partial W_{i,j,d,d'}}$

$\frac{\partial \mathscr{L}}{\partial W_{i,j,d,d'}} = \sum_{i',j',d'} \frac{\partial \mathscr{L}}{\partial Y_{i',j',d'}} X_{i'+i-1,j'+j-1,d}$, as before again a convolution.

3d) Multiple channels - $\frac{\partial \mathscr{L}}{\partial X}$

$\frac{\partial \mathscr{L}}{\partial X_{i,j,d}} = \sum_{i',j',d'} \frac{\partial \mathscr{L}}{\partial Y_{i',j',d'}} \frac{\partial Y_{i',j',d'}}{\partial X_{i,j,d}}$

therefore we have that

$\frac{\partial \mathscr{L}}{\partial X_{i,j,d}} = \sum_{i',j',d'} \frac{\partial \mathscr{L}}{\partial Y_{i',j',d'}} \frac{\partial}{\partial X_{i,j,d}} \left(\sum_{m,n,e} W_{m,n,e,d} X_{i'+m-1,j'+n-1,e} + b_e \right)$

So if we again take only non-zero derivatives i.e. where $i = i' + m - 1, j = j' + n - 1$, then we have $m = i - i' + 1$ and $n = j - j' + 1$. Giving us finally that:

$\frac{\partial \mathscr{L}}{\partial X_{i,j,d}} = \sum_{i',j',d'} \frac{\partial \mathscr{L}}{\partial Y_{i',j',d'}} W_{i-i'+1,j-j'+1,d,d'}$

For a mini-batch we simply pass through another dimension forwards and have a batch of errors which are summed and passed back.

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. Results

My implementation of the first three models was very similar. These were all effectively fully connected models with a varying number of layers. My implementation of stochastic gradient descent was the very simplest. A simple fixed step size...no decay, no momentum etc.

Compared to tensorflow my models learned slower. Not just in terms of run-time but the accuracy for a given number of epochs. I didn't do any %timeit's. But the fact tensorflow was quicker in clock time was not a surprise to me. It is optimised for speed. I was however slightly surprised to see that it had better accuracy for the same number of epochs on the under-fitting examples. At first I believed my implementation was inaccurate. But I tested, varied the number of epochs and am satisfied my implementations are correct. Thus the difference is presumably in the optimisation itself.

The difference in speed with my CNN implementation was considerable. My first implementation had multiple nested loops and was totally unworkable. After some research and finding the Stanford material and looking up Teoplitz matrices, some speed up was enabled through turning the convolutions into matrix multiplications. But, my code is still very slow compared to tensorflow and needs more work on vectorisation and re-factoring.

There was also a small difference in stability, where most of my implementations learned and were stable, with only the high learning rate on the CNN being unstable.

In conclusion the design of the first three models was pretty standard - matrix multiplications, and either relu or softmax. The optimiser was very, very basic. Simple fixed step stochastic gradient descent. Tensorflow was faster, and also for the under-trained examples had better predictive power for a given number of epochs. The big difference was the last model, where tensorflow was much, much faster in terms of run-time.

2. Under-fitting

Under-fitting is where the model either lacks enough power to address the problem (usually where is has a high bias and could benefit from more expressive power) or where the model as is could improve its training and test accuracy simply by training for longer. I will take the models as given and address the question in the latter context.

Model 1 is under-fitting in scenarios one and two. This is the same model, with the same number of epochs and simply a different learning rate. The capacity of the model seems to be an accuracy of around 0.92. The learning rate is very low in the first scenario and the model would benefit from many more epochs of training. In the second scenario there is slight under-fitting the model would still benefit from more training. One can also see from the training curves that the model is still improving.

Model 2 appears to be a similar situation. Same number of epochs and different learning rates. The model in the third scenario has an accuracy of 0.96. Thus again I would argue there is under-fitting in the first two scenarios and the model would benefit from more training.

Model 3 - Again the model is capable of generalising to at least an accuracy of 0.97. Scenario one is clearly underfitting, the learning rate is low and the model needs more training. Scenario two the model is underfitting. This is less evident by simply looking at the training curves. But one can see that the same model with the same learning rate, trained for more epochs generalises better.

Model 4 For the convolutional neural net, I wouldn't exactly call scenario one under-fitting. There is no fitting at all ! The learning rate is too high.

3. Over-fitting

Over-fitting in a neural net context is typically when the model has been trained for too long. The training accuracy continues to climb and the test set accuracy flattens or even becomes worse. An expressive enough model can effectively 'learn' the training set. This can also include any noise! However, this does not aid generalisation. One should also again consider the expressiveness of the model itself. The first model (the linear layer plus softmax) lacks expressive power. In contrast neural nets typically have many parameters and need techniques to avoid over-fitting. (Watching training curves and thus manually implementing esrly stopping as we are here, is one unsophisticated method, others might be techniques such as dropout).

I would argue that the final scenario (most number of epochs) for models two, three and four show signs of over-fitting based upon the fact that the training accuracy some time earlier continued to climb whereas the test set accuracy is not improving. As mentioned earlier I would tend to implement dropout, and also a lookback period for early stopping to avoid this in a more systematic way.

In [0]: