MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.036—Introduction to Machine Learning
Spring 2017

Project 2: Handwritten Digit Recognition Issued: Mon., 3/20 Due: Fri., 4/7 at 9am

Project Submission: Please submit a .zip file of your project on the Stellar web site by 9am, 4/7. The zip file should contain: a PDF of your project report and your Python code.

TAs: Alex, Aliaa, Guadalupe, Mohammad, Nathan.

Introduction

This project requires you to use Python 3. It will not run with Python 2

For this project we will use the MNIST[1] (Mixed National Institute of Standards and Technology) database, which contains binary images of handwritten digits commonly used to train image processing systems. The digits were collected from among Census Bureau employees and high school students. The database contains $60,000$ training digits and $10,000$ testing digits, all of which have been size-normalized and centered in a fixed-size image of 28×28 pixels. Many methods have been tested with this dataset and in this project, you will get a chance to experiment with the task of classifying these images.

You will first implement multinomial logistic regression (aka softmax regression) with gradient descent and run your model on a test dataset. Next, you will use PCA to reduce high-dimensional vectors of pixels to a low-dimensional vector of features. You will also experiment with the reverse by increasing the dimension using kernels and then apply the regression model again, except to these new features. Lastly, you will apply neural network models to the same task.

Other than your code, you will submit a PDF with relevant plots and discussion. What to put in the PDF for each part is indicated with **"include"** annotations below.

Point distribution: Even though the total amount of points is doubled than that of project 1, both projects have the same weight on your final grade.

Problem	1	2	3	4	5	**Total**
Points	40	40	40	40	40	200

[1] <http://yann.lecun.com/exdb/mnist/

0. Setup

As with the last project, please use Python's NumPy numerical library for handling arrays and array operations; use `matplotlib` for producing figures and plots.

Download `project2.zip` from Stellar and unzip it into a working directory. The zip file contains the image dataset in `mnist.pkl.gz`, several relevant Python files, and a `main.py` file where you will be running your code.

1. Multinomial/Softmax Regression and Gradient Descent [40pts]

In this section, you will implement multinomial regression and the gradient descent algorithm to learn a set of parameters used to classify images as digits from 0-9. You will work with the raw pixel values of each image. That is, each pixel in the original image corresponds to a feature in our feature vector.

To get warmed up to the MNIST data set run `main.py`. This file provides code that reads the data from `mnist.pkl.gz` by calling the function `getMNISTData` that is provided for you in `utils.py`.
The call to `getMNISTData` returns Numpy arrays:

- `trainX` : A matrix of the training data. Each row of `trainX` contains the features of one image, which are simply the raw pixel values flattened out into a vector of length $784 = 28^2$. The pixel values are float values between 0 and 1 (0 stands for black, 1 for white, and various shades of gray in-between).
- `trainY` : The labels for each training datapoint, aka the digit shown in the corresponding image (a number between 0-9).
- `testX` A matrix of the test data, formatted like `trainX`.
- `testY` The labels for the test data, which should *only* be used to evaluate the accuracy of different classifiers in your report.

Next, we call the function `plotImages` to display the first 20 images of the training set. Look at these images and get a feel for the data (don't include these in your write-up).

We initialize the temperature parameter found in the softmax equation to 1.

The main function which you will call to run the code you will implement in this section is `runSoftmaxOnMNIST` in `main.py` (already implemented). In the appendix, we describe a number of the methods that are already implemented for you in `softmax_skeleton.py` that will be useful.

In order for the regression to work, you will need to implement three methods. Below we describe what the functions should do, for detailed description of the inputs and outputs, consult the appendix. We have included some methods in a file called `softmax_verification.py` to help you verify that the methods you have implemented are behaving sensibly.

1. Write a function `computeProbabilities` [5pts] that computes, for each data point $x^{(i)}$, the probability that $x^{(i)}$ is labeled as j for $j = 0, 1, ..., k - 1$.

   See the appendix for a hint about dealing with overflow errors.

   `verifyFirstIterationProbabilities` and `verifySecondIterationProbabilities` are available to help you reason that your code is correct. Note that `verifySecond-`

`IterationProbabilities` will not pass until you implement `runGradientDescentIteration`.

2. Write a function `computeCostFunction` [5pts] that computes the total cost over every data point.

   For a reminder on the cost function formula, consult the appendix.

   Use `verifyFirstIterationCostFunction` and `verifySecondIterationCostFunction` to help you reason that your code is correct. Note that `verifySecondIterationCostFunction` will not pass until you implement `runGradientDescentIteration`.

3. Write a function `runGradientDescentIteration` [5pts] that runs one step of batch gradient descent.

   For a hint on the gradient descent step, consult the appendix.

4. Finally, in your report **include** the final test error [5pts].

   If you have implemented everything correctly, the error on the test set should be around 0.1, which implies the linear softmax regression model is able to recognize MNIST digits with around 90% accuracy.

Temperature Parameter.

We will now explore the effects of the temperature parameter in our algorithm.

5. In your report, explain how the temperature parameter affects the probability of a sample $x(i)$ being assigned a label that has a large θ. What about a small θ? [3pts]

6. Set the temperature parameter to be .5, 1, and 2; re-run `runSoftmaxOnMNIST` for each one of these (add your code to the specified part in `main.py`). There should be a trend in the error rate. What is it? What does this suggest about our probability distribution? (Make sure to add the error rates and answers to these questions in your report). [5pts]

   If you have implemented everything correctly, one of the new error rates should be around .08 and the other around .13

Return the `tempParameter` to be 1, and re-run `runSoftmaxOnMNIST`.

Changing labels

We now wish to classify the digits by their (mod 3) value, such that the new label $y^{(i)}$ of sample i is the old $y^{(i)} \pmod 3$.

7. Given that we already classified every $x^{(i)}$ as a digit, we could use the model we already trained and just get the (mod 3) of our estimations.

   Implement `updateY`, which changes the old digit labels for the training and test set for the new (mod 3) labels (this should just be one line of code), and `computeTestErrorMod3`, which takes the test points X, their correct labels Y (digits (mod 3) from 0-2), theta, and the tempParameter, and returns the error. [5pts]

   Example:

	Estimated Y	Estimated Y (mod 3)	Correct Y	Correct Y (mod 3)
x_1	9	0	8	2
x_2	6	0	6	0
x_3	5	2	8	2

The error of the regression with the original labels would be .66667

However, the error of the regression when comparing the (mod 3) of the labels would be .33333

Find the error rate of the new labels (call these two functions at the end of `runSoftmaxOnMNIST`), and add these to your report. Compare it to the error of our first classification problem. Why did it change the way it did? Or why did it stay the same?

See the appendix for detailed explanations of the inputs and outputs of the functions

Now suppose that instead we want to retrain our classifier with the new labels.

8. Implement `runSoftmaxOnMNISTMod3` in `main.py` to perform this new training; report the new error rate. Use the same parameters as `runSoftmaxOnMNIST`. [5pts]

   See the appendix for more details about this function.

9. Compare the error rates of (7.) and (8.). Give an explanation for this difference. [2pts]

Important: To make sure you receive credit for section 1, run the file `softmax_checker.py` which will try to check that your functions work with the specified input and return the specified output. If it errors that means there is an issue in your code. If it does not error it will say whether the simple tests in place to check types pass or fail.

2. Classification using manually-crafted features [40pts]

The performance of most learning algorithms depends heavily on the representation of the training data. In this section, we will try representing each image using different features in place of the raw pixel values. Subsequently, we will investigate how well our regression model from the previous section performs when fed different representations of the data.

Dimensionality Reduction via PCA. Principal Components Analysis[2] (PCA) is the most popular method for linear dimension reduction of data and is widely used in data analysis. Briefly, this method finds (orthogonal) directions of maximal variation in the data. By projecting an $n \times d$ dataset X onto $k \ll d$ of these directions, we get a new dataset of lower dimension that reflects more variation in the original data than any other k-dimensional linear projection of X. By going through some linear algebra, it can be proven that these directions are equal to the k eigenvectors corresponding to the k largest eigenvalues of the covariance matrix $\widetilde{X}^T\widetilde{X}$, where $\widetilde{X}$ is a centered version of our original data.

Remark: The best implementations of PCA actually use the Singular Value Decomposition of $\widetilde{X}$ rather than the more straightforward approach outlined here, but these concepts are beyond the scope of this course.

See the appendix for descriptions of some of the functions that we have provided for you to use in this part (all located in `features.py`):

Below are the steps you are responsible for in this section:

1. Fill in function `projectOntoPC` in `features.py` that implements PCA dimensionality reduction of dataset X [10 pts]. Refer to the appendix for details of the inputs and outputs of this function.

   Note that to project a given $n \times d$ dataset X into its k-dimensional PCA representation, one can use matrix multiplication (after first centering X): $\widetilde{X}V$, where V is the $d \times k$ matrix whose columns are the top k eigenvectors of $\widetilde{X}^T\widetilde{X}$. This is because the eigenvectors are of unit-norm, so there is no need to divide by their length.

2. Use `projectOntoPC` to compute a 18-dimensional PCA representation of the MNIST training and test datasets, as illustrated in `main.py`.

3. Retrain your softmax regression model (using the same settings the previous section) on the MNIST training dataset and report its error on the test data, this time using these 18-dimensional PCA-representations rather than the raw pixel values (**Include** the error in your report [5 pts]).

   If your PCA implementation is correct, the model should perform nearly as well when only given 18 numbers encoding each image as compared to the 784 in the original data (error on the test set using PCA features should be around 0.15). This is because PCA ensures these 18 feature values capture the maximal amount of variation in the original 784-dimensional data.

   Remark: Note that we only use the training dataset to determine the principal components. It is *improper* to use the test dataset for anything except evaluating the accuracy of our predictive model. If the test data is used for other purposes such as selecting good features, it is possible to overfit the test set and obtain overconfident estimates of a model's performance.

[2] In-depth exposition: `www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf`

4. Use the call to `plotPC` in `main.py` to visualize the first 100 MNIST images, as represented in the space spanned by the first 2 principal components of the training data.
   Include this plot in your report [5 pts].

   Remark: Two dimensional PCA plots offer a nice way to visualize some global structure in high-dimensional data, although approaches based on nonlinear dimension reduction may be more insightful in certain cases. Notice that for our data, the first 2 principal components are insufficent for fully separating the different classes of MNIST digits.

5. Use the calls to `plotImages()` and `reconstructPC` in `main.py` to plot the reconstructions of the first two MNIST images (from their 18-dimensional PCA-representations) alongside the originals.
   Include these plots in your report [5 pts].

Cubic Features.

In this section, we will work with a *cubic feature* mapping which maps an input vector $x = [x_1, \ldots, x_d]$ into a new feature vector $\phi(x)$, defined so that for any $x, x' \in \mathbb{R}^d$:

$$\phi(x)^T \phi(x') = (x^T x' + 1)^3$$

6. In 2-D, let $x = [x_1, x_2]$. Write down the explicit cubic feature mapping $\phi(x)$ as a vector (**include** in your report).
   Hint: $\phi(x)$ should be a 10-dimensional vector.

   Now, fill in function `cubicFeatures` in `features.py`. Given an input dataset (with d- dimensional features where d is not necessarily equal to 2), this function returns a new dataset where each observation is now represented using cubic features [10 pts].

   Run `cubicFeatures_checker.py` to run some simple tests on your `cubicFeatures` implementation.

7. If we explicitly apply the cubic feature mapping to the original 784-dimensional raw pixel features, the resulting representation would be of massive dimensionality. Instead, we will apply the quadratic feature mapping to the 10-dimensional PCA representation of our training data which we will have to calculate just as we calculated the 18-dimensional representation in the previous problem. After applying the cubic feature mapping to the PCA representations for both the train and test datasets, retrain the softmax regression model using these new features and report the resulting test set error (**Include** this error in your report [5 pts]).

   Important: You will probably get a runtime warning for getting the log of 0, ignore. Your code should still run and perform correctly.

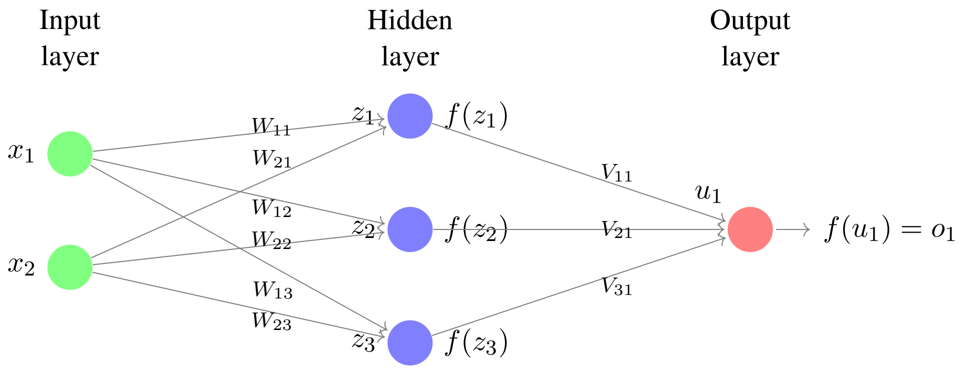
   If you have done everything correctly, softmax regression should perform better (on the test set) using these features than either the 18-dimensional principal components or raw pixels. The error on the test set using cubic features should only be around 0.08, demonstrating the power of nonlinear classification models.

Remark: Note that the feature mapping we have been working with actually corresponds to the cubic kernel, which is defined as: $K(x, x') = (x^T x' + 1)^3$. Recall, that the feature mapping ϕ associated with a kernel K satisfies $K(x, x') = \phi(x)^T \phi(x')$ for any $x, x' \in \mathbb{R}^d$.

In practice, this kernel function is all that is required to perform nonlinear kernel regression and one never has to actually compute the feature mapping (recall the kernel trick). Nevertheless, we have explicitly worked with the feature mapping corresponding to this kernel for educational purposes. Note that working with the explicit representations induced by the feature mapping can vastly increase in the dimensionality of the problem. This is why the kernel trick is a key component of kernel methods, allowing them to run efficiently in practice!

3. Neural Network Basics [40pts]

Good programmers can use neural nets. Great programmers can make them. This section will guide you through the implementation of a simple neural net with an architecture as shown in the figure below. You will implement the net from scratch (you will probably never do this again, don't worry) so that you later feel confident about using libraries. We provide some skeleton code in `neural_nets.py` for you to fill in.



1. **Activation Functions**

   The first step is to design the activation functions for each neuron. In this problem, we will initialize the network weights to 1, use **ReLU** for the activation function of the hidden layers, use an identity function for the output neuron. The hidden layer has a bias but the output layer does not. Complete the helper functions in `neural_networks.py`, including `rectified_linear_unit` and `rectified_linear_unit_derivative`, for you to use in the `Neural_Network` class.

2. **Training the Network**

   Forward propagation is simply the summation of the previous layer's output multiplied by the weight of each wire, while back-propagation works by computing the partial derivatives of the cost function with respect to **any** weight or bias in the network. In back propagation, the network gets better at minimizing the error and predicting the output of the data being used for training by incrementally updating their weights and biases using stochastic gradient descent.

   We are trying to estimate a continuous-valued function, thus we will use squared loss as our cost function and an identity function as the output activation function. $f(x)$ is the activation function that is called on the input to our final layer output node, and $\hat{a}$ is the predicted value, while y is the actual value of the input.

$$C = \frac{1}{2}(y - \hat{a})^2 \tag{1}$$

$$f(x) = x \tag{2}$$

   When you're done implementing the function `train`, run the script and see if the errors are decreasing. If your errors are all under 0.15 after the last training iteration then you have implemented the neural network training correctly.

3. **Predict Test Data**

   Now fill in the code for the function `predict`, which will use your trained neural network in order to label new data.

   When you're done, run the script and make sure that all of your predictions pass the test cases, then turn in your code.

   Answer the following questions in your report:

4. How can you improve the learning rate in order to improve the efficiency of your network in terms of training and accuracy? Describe one of the many methods.

5. What is the danger to having too many hidden units in your network?

6. What would happen if you run the code for more epochs in terms of training and testing accuracy? Describe the trajectory of these accuracies as the network runs through the epochs. Feel free to use a graph.

7. How do you optimize for the amount of epochs used in training in order to have a better neural network? Describe one method.

4. Classification for MNIST using deep neural networks [40pts]

In this section, we are going to use deep neural networks to perform the same classification task as in previous sections. We will use Keras, a python library built upon the deep learning framework Theano. **Refer to the Appendix for instructions on setting up the necessary deep learning environment.**

1. **Fully-Connected Neural Networks**

   First, we will employ the most basic form of a deep neural network, in which the neurons in adjacent layers are fully connected to one another.

   (a) We have provided a toy example `mnist_nnet_fc.py` in which we have implemented for you a simple neural network. This network has one hidden layer of 128 neurons with a rectified linear unit (ReLU) nonlinearity, as well as an output layer of 10 neurons (one for each digit class). Finally, a softmax function normalizes the activations of the output neurons so that they specify a probability distribution. Reference the Keras website[3] and read through the documentation in order to gain a better understanding of the code. Then, try running the code on your computer with the command `python3 mnist_nnet_fc.py`. This will train the network with 10 epochs, where an epoch is a complete pass through the training dataset. Total training time of your network should take no more than a couple of minutes; if you find the training to be going slowly on your machine, try using an Athena[4] cluster machine. At the end of training, your model should have an accuracy of more than $\%85$ on test data.
   Report the test accuracy level achieved in your write-up. [2 pts]

   (b) Modify your model to reach over 98% **test accuracy** after 10 epochs of training. Things you can, but don't have to, tweak include the number of fully-connected layers, the number of neurons in each layer, or the parameters of the stochastic gradient descent (SGD) optimizer (such as the learning rate).
   Submit your code, and include a description of your final model architecture in your report. Give its accuracy on the test data. What did you try that worked, and what did you try that didn't work? [8 pts]

2. **Convolutional neural networks**

   Next, we are going to apply convolutional neural networks to the same task. These networks have demonstrated great performance on many deep learning tasks, especially in computer vision.

   (a) We provide a skeleton code `mnist_nnet_cnn_skeleton.py` which includes examples of some (**not all**) of the new layers you will need in this part. Using the Keras documentation website[5], complete the parts of the code called "Design the model here" and "Import the layer types needed" to implement a convolutional neural network with following layers in order:

   - A convolutional layer with 32 filters of size 3×3
   - A ReLU nonlinearity
   - A max pooling layer with size 2×2
   - A convolutional layer with 64 filters of size 3×3
   - A ReLU nonlinearity

[3] `http://keras.io/`

[4] `http://ist.mit.edu/athena`

[5] `http://keras.io/`

- A max pooling layer with size 2 × 2
- A flatten layer
- A fully connected layer with 128 neurons
- A dropout layer with drop probability 0.5
- A fully-connected layer with 10 neurons
- A softmax layer

Without GPU acceleration, you will likely find that this network takes quite a long time to train. For that reason, we don't expect you to actually train this network until convergence. Implementing the layers and verifying that you get approximately 93% **training accuracy** and 98% **validation accuracy** after one training epoch (this should take less than 10 minutes) is enough for this project. If you are curious, you can let the model train for a few hours; if implemented correctly, your model should achieve >99% **test accuracy** after 10 epochs of training. If you have access to a CUDA compatible GPU, you could even try configuring Keras to use your GPU.

Include the code for your CNN in your submission. [10 pts]

(b) (**Optional**) To understand why convolutional neural networks work so well, we often need to visualize the intermediate filters to see what's going on. blog[6] and try visualizing the features learned in each layer of your CNN network.

REMEMBER Submit a .zip file containing your source code and your report in PDF format to Stellar.

[6] http://blog.keras.io/how-convolutional-neural-networks-see-the-world.html

5. Overlapping, multi-digit MNIST [40pts]

In this problem, we are going to go beyond the basic MNIST. We will train a few neural networks to solve the problem of hand-written digit recognition using a multi-digit version of MNIST.

In your project folder, look at the `Problem 5` subfolder. There you can find the files `mlp.py` and `conv.py`. Your main task here is to complete the code inside the method `main` in these files.

Do the following steps: [4pts]

- Look at `main` method in each file. Identify the training and test data and labels. How many images are inside the train and test data? What is the size of each image? (Include these numbers in your report.)
- Familiarize yourself with the *Functional API* in Keras (https://keras.io/getting-started/functional-api-guide/). This is a more powerful version of the `Sequential` model that you used in the previous problem and, unlike that model, is not restricted to networks which have only a single input and output.
- Look at the lines 27 and 28 of in `mlp.py` (`model.compile(...)` and `model.fit(...)`). Try to make sense of what those lines are trying to achieve. What is `y_train[0]` and `y_train[1]`? Include a brief statement about the arguments to each function (i.e. `model.fit` and `model.compile`) in your report and what they mean.

Now given the intuition you have built with the above steps, complete the following task.

1. Complete the code `main` in `mlp.py` to build a fully-connected model with a single hidden layer with 64 units. For this, you need to make use of `Input`, `Flatten`, and `Dense` layers in Keras.[7] [12 pts]
2. Complete the code `main` in `conv.py` to build a convolutional model. For this, you need to make use of `Convolutional2D` layers and `MaxPooling2d` layers (and perhaps Dropout) in Keras. Make sure that the last layer of the neural network is a fully connected (dense) layer. A suggested specific setting of parameters for the model is provided in the code. [12pts]
3. Next, change the parameters and settings of the models above. Train your model with various parameter settings. Some things you can try is to modify the learning algorithm from `sgd` to more sophisticated ones (such as `adam`) or you can modify the network architecture (number of layers or unit per layers, activation function, etc.). Choose three of the extra models that you selected and report their architecture and report their performance and training time. [12pts] (**Optional**: Explain what you observed. What parameters or settings were more conductive to get a better model with greater generalization capability and lower error? Did extra training for a model always help or did the training accuracy plateau or even get worse after some point?)

[7] Hint: Note that your model must have two outputs (corresponding to the first and second digits) to be compatible with `model.fit`.

For each of the models above (exactly 5) you must include a description of model architecture, number of epochs trained, training time on your machine, and final evaluation scores (on test set) in your report.

(**) For this problem, we provide two versions of the multi-digit MNIST dataset. The two only differ in size, and otherwise are completely equivalent. For achieving better accuracy, we recommend working with the larger (full) dataset. However, if your machine cannot handle the large dataset, you have the option of working with the `mini` dataset. Your grade will not be affected by this choice.[8]

[8] The default setting of the code is to use the mini dataset. To switch to full-dataset, simply change the variable `use_mini_dataset` to `False`.

Appendix: Background and further details

Multinomial/Softmax Regression- Given functions

1. `augmentFeatureVector`: adds the $x_0^{(i)} = 1$ feature for each data point $x^{(i)}$ for $i = 1, 2, ..., n$. The inputs are:

   - `X`, an $n \times (d-1)$ Numpy array of n data points, each with $d-1$ features.

   The function returns:

   - `X_augment`, an $n \times d$ Numpy array of n data points, each with d features.

   The motivation for doing this is to form a compact representation of $\theta \cdot x + \theta_0$. For example, if θ and x were both $(d-1)$ dimensional vectors, we could define a d dimensional vector $\theta' = [\theta_0, \theta_1, \theta_2, \ldots, \theta_{d-1}]$ and $x' = [1, x_1, x_2, \ldots x_{d-1}]$. Then $\theta \cdot x + \theta_0 = \theta' \cdot x'$.

2. `softmaxRegression`: runs batch gradient descent for a specified number of iterations on a dataset, with θ initialized to the all-zeros array. Here, θ is a $k \times d$ Numpy array, where row j represents the parameters of our model for label j for $j = 0, 1, ..., k-1$. The inputs are:

   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - `Y`, an $n \times 1$ Numpy array containing the labels (a number between 0-9) for each data point.
   - `tempParameter`, the temperature parameter of softmax function (scalar).
   - α, the learning rate (scalar).
   - λ, the regularization constant (scalar).
   - `k`, the number of labels (scalar).
   - `numIterations`, the number of iterations to run gradient descent (scalar).

   The function returns:

   - θ', a $k \times d$ Numpy array that is the final value of θ.
   - `costFunctionProgression`, a Python list containing the cost calculated at each step of gradient descent.

3. `getClassification`: makes predictions by classifying a given dataset. The inputs are:

   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - θ, a $k \times d$ Numpy array, where row j represents the parameters of our model for label j.
   - `tempParameter`, the temperature parameter of softmax function (scalar).

   The function returns:

   - `Ŷ`, an $n \times 1$ Numpy array, containing the predicted label (a number between 0-9) for each data point.

4. `runSoftmaxOnMNIST` in `main.py`: runs the above `softmaxRegression` on the MNIST training set and computes the test error using the test set. It uses the following values for parameters: $\alpha = 0.3$, $\lambda = 10^{-4}$, and $numIterations = 150$. It also plots the cost function over the number of iterations. Once softmax regression has run, you will get the final model parameters θ.

We have called the function `writePickleData` that will save the model parameters to a file called `theta.pkl.gz` in the current directory. Rather than rerunning softmax regression, you can read in your model parameters by calling the function `readPickleData` that is provided in `utils.py`. **This model must be saved with this exact name and will be used during grading**.

Multinomial/Softmax Regression- Functions to be implemented

1. `computeProbabilities` that computes, for each data point $x^{(i)}$, the probability that $x^{(i)}$ is labeled as j for $j = 0, 1, ..., k - 1$. The inputs are:

   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - θ, a $k \times d$ Numpy array, where row j represents the parameters of our model for label j.
   - `tempParameter`, the temperature parameter of softmax function (scalar).

   The function returns:

   - `H`, a $k \times n$ Numpy array, where entry (j, i) is the probability that $x^{(i)}$ is labeled as j.

   See the next section in the appendix for a hint about dealing with overflow errors.

2. `computeCostFunction` that computes the total cost over every data point. The inputs are:

   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - `Y`, an $n \times 1$ Numpy array containing the labels (a number between 0-9) for each data point.
   - θ, a $k \times d$ Numpy array, where row j represents the parameters of our model for label j.
   - λ, the regularization constant (scalar).
   - `tempParameter`, the temperature parameter of softmax function (scalar).

   The function returns:

   - `c`, the cost value (scalar).

   The cost function formula can be found in the next section of the appendix.

3. `runGradientDescentIteration` that runs one step of batch gradient descent. The inputs are:

   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - `Y`, an $n \times 1$ Numpy array containing the labels (a number between 0-9) for each data point.
   - θ, a $k \times d$ Numpy array, where row j represents the parameters of our model for label j.
   - α, the learning rate (scalar).
   - λ, the regularization constant (scalar).
   - `tempParameter`, the temperature parameter of softmax function (scalar).

   The function returns:

   - θ', a $k \times d$ Numpy array that is the next value of θ after one step of gradient descent.

   For a hint on the gradient descent step, consult the next section of the appendix.

Multinomial/Softmax Regression – Dealing with overflow errors

The softmax function h for a particular vector x requires computing

$$h(x) = \frac{1}{\sum_{j=1}^{k} e^{\theta_j \cdot x/\tau}} \begin{bmatrix} e^{\theta_1 \cdot x/\tau} \\ e^{\theta_2 \cdot x/\tau} \\ \vdots \\ e^{\theta_k \cdot x/\tau} \end{bmatrix},$$

where $\tau > 0$ is the *temperature parameter*. When computing the output probabilities (they should always be in the range $[0, 1]$), the terms $e^{\theta_j \cdot x/\tau}$ may be very large or very small, due to the use of the exponential function. This can cause numerical or overflow errors. To deal with this, we can simply subtract some fixed amount c from each exponent to keep the resulting number from getting too large. Since

$$h(x) = \frac{e^{-c}}{e^{-c}\sum_{j=1}^{k} e^{\theta_j \cdot x/\tau}} \begin{bmatrix} e^{\theta_1 \cdot x/\tau} \\ e^{\theta_2 \cdot x/\tau} \\ \vdots \\ e^{\theta_k \cdot x/\tau} \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{[\theta_j \cdot x/\tau]-c}} \begin{bmatrix} e^{[\theta_1 \cdot x/\tau]-c} \\ e^{[\theta_2 \cdot x/\tau]-c} \\ \vdots \\ e^{[\theta_k \cdot x/\tau]-c} \end{bmatrix},$$

subtracting some fixed amount c from each exponent will not change the final probabilities. A suitable choice for this fixed amount is $c = \max_j \theta_j \cdot x/\tau$.

Multinomial/Softmax Regression – Cost Function

The cost function $J(\theta)$ is given by: (Use natural log)

$$J(\theta) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{j=1}^{k}[[y^{(i)} == j]] \log \frac{e^{\theta_j \cdot x^{(i)}/\tau}}{\sum_{l=1}^{k} e^{\theta_l \cdot x^{(i)}/\tau}}\right] + \frac{\lambda}{2}\sum_{i=1}^{k}\sum_{j=0}^{d-1}\theta_{ij}^2$$

Multinomial/Softmax Regression – Gradient

The partial derivative of $J(\theta)$ wrt a particular θ_j is given by:

$$\nabla_{\Theta_j} J(\theta) = -\frac{1}{\tau n}\sum_{i=1}^{n}[x^{(i)}([[y^{(i)} == j]] - p(y^{(i)} = j|x^{(i)}, \theta))] + \lambda\theta_j$$

Multinomial/Softmax Regression- Functions for (mod 3) classification

1. `updateY` changes the old digit labels for the training and test set for the new (mod 3) labels. The inputs are:

   - `trainY`, an $n \times 1$ Numpy array containing the labels (a number between 0-9) for each data point in the training set.
   - `testY`, an $n \times 1$ Numpy array containing the labels (a number between 0-9) for each data point in the test set.

   The function returns:

- `trainYMod3`, an $n \times 1$ Numpy array containing the new labels (a number between 0-2) for each data point in the training set.
- `testYMod3`, an $n \times 1$ Numpy array containing the new labels (a number between 0-2) for each data point in the test set.

2. `computeTestErrorMod3` returns the error of these new labels when the classifier predicts the digit. The inputs are:

   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - `Y`, an $n \times 1$ Numpy array containing the labels (a number between 0-9) for each data point.
   - θ, a $k \times d$ Numpy array, where row j represents the parameters of our model for label j.
   - `tempParameter`, the temperature parameter of softmax function (scalar).

   The function outputs:

   - `testError` the error rate of the classifier (scalar).

3. `runSoftmaxOnMNISTMod3` in `main.py`: runs `softmaxRegression` on MNIST training set with the new `trainYMod3` and computes the test error using the test set and `testYMod3` as the ground truth. It uses the same parameters as `runSoftmaxOnMNIST`. It also plots the cost function over the number of iterations. Save the new model to a file called `thetaM.pkl.gz`.

PCA- Given functions

1. `centerData` centers the data (by subtracting off the mean of each feature).
   The input is: X, an $n \times d$ Numpy array of n data points, each with d features.
   The function returns: $n \times d$ Numpy array $\widetilde{X}$, where for each $i = 1, \ldots, n$ and $j = 1, \ldots, d$:

$$\widetilde{X}_{i,j} = X_{i,j} - \mu_j \quad \text{with } \mu_j = \frac{1}{n}\sum_{i=1}^{n} X_{i,j}$$

2. `principalComponents` computes the principal component directions of an input data matrix X ($n \times d$ Numpy array). This function first calculates the covariance matrix, $\widetilde{X}^T\widetilde{X}$ and then find its eigenvectors.
   The function returns: $d \times d$ matrix (Numpy array) whose columns are the principal component directions sorted in descending order by the amount of variation each direction (these are equivalent to the d eigenvectors of $\widetilde{X}^T\widetilde{X}$ sorted in descending order of eigenvalues, so the first column corresponds the eigenvector with largest eigenvalue).

3. `plotPC` produces a scatter-plot of data after it has first been projected down to its 2-D representation in the first 2 principal components.

4. `reconstructPC` tries to reconstruct an original data sample from its lower-dimensional PCA-representation. Note that to reconstruct an observation from its representation in the top k principal components, $x_{\text{pca}} \in \mathbb{R}^{1\times k}$ we can use matrix algebra: $x_{\text{pca}} \cdot V^T + \mu$, where V is the $d \times k$ matrix whose columns are the top k eigenvectors of $\widetilde{X}^T\widetilde{X}$, and μ is a $1 \times d$ vector containing the mean of each feature (see the definition of its jth element μ_j above).

PCA- Functions to be implemented

1. `projectOntoPC` in `features.py` that implements PCA dimensionality reduction of dataset X. Your input should be:

    - `X`, a $n \times d$ Numpy array of n data points, each with d features.
    - `pcs`, a $d \times d$ matrix whose columns are the d eigenvectors of $\widetilde{X}^T \widetilde{X}$, ordered in *descending* order of eigenvalues (i.e. the output of `principalComponents`).
    - `n_components`, the number of principal components to use in the PCA representation.

    Your function should return a $n \times$`n_components` Numpy array, whose rows represent low-dimensional feature vectors corresponding to the projection of dataset X onto its first n principal components.

Image features

Many other feature representations of images have been developed besides the approaches considered here. Some examples of widely used image features include HOG (Histogram of Oriented Gradients), SIFT (Scale-Invariant Feature Transform), and GIST. All of them are variations on the same theme: abstract away from raw pixels, and form more general representations of image content. This can be done manually by histogramming and averaging measurements in different image regions. These measurements are often pixel intensities (grayscale or color) or gradients (edges). A histogram over an image region just specifies how much of each measurement there is in a region (e.g. how many horizontal edges, vertical edges, etc.) without specifying the exact pixel location. This allows for a more general representation that will still match images that have similar distributions of features but are not identical at the pixel level.

Recently, deep learning approaches have outperformed complex manually-engineered features like HOG/SIFT in large image datasets. The original appeal of neural networks was a general purpose model that could learn useful features directly from raw pixels. However, in the past few years, the state of the art for computer vision tasks like object recognition has been significantly improved by spending effort to develop complex CNN architectures[9] in place of feature-engineering.

[9] GoogleNet architecture: `http://homes.cs.washington.edu/~jmschr/lectures/googlenet.png`
For more famous CNN architectures, see: `http://cs231n.github.io/convolutional-networks/#case`

Keras / Theano Environment setup

To set up the computing environment for Keras, you can consider the following solutions:

Solution 1 - Athena

These instructions describe how to install Tensorflow and/or Theano on an MIT Athena machine.[10]

They install all required packages in your user account, so you should ensure you have enough free space available in your home directory, for both the packages and the datasets.

You can check your available space using the ”quota -v” command.

Please run the following commands:

```
wget https://bootstrap.pypa.io/get-pip.py
python3 get-pip.py --user
python3 -m pip install keras --user
python3 -m pip install h5py --user
python3 -m pip install tensorflow --user
python3 -m pip install theano --user
```

Choosing a backend: Theano vs. Tensorflow.

You can edit the file `~/.keras/keras.json` file to choose your preferred backend. The ”backend” line should be either ”theano” or ”tensorflow”, depending on which backend you want to use.

For example, this is what the full file `~/.keras/keras.json` looks like in our machine.

```
----------------------------
"floatx": "float32",
"epsilon": 1e-07,
"backend": "tensorflow,
"image_dim_ordering": "th"
-----------------------------
```

REMARK:
Note that Tensorflow is only available for 64-bit OSes. If your machine is running a 32-bit OS, please use Theano only.

Memory limitations in Athena dialup (remote)

If you use an Athena dialup machine **remotely** (e.g., via athena.dialup.mit.edu), there’s a 1GB memory limit

[10] You can access Athena remotely via SSH by following the instructions at `http://web.mit.edu/dialup/www/ssh.html`.

per session. This will make it impossible to train a network on the full dataset, and your python/keras code will likely abort with a “MemoryError” condition.

For this reason, we also provide a ”mini” dataset, which should be usable under these restricted conditions. If you want to use the full dataset, just use any Athena computer on campus (by physically logging-in to it), since in this case there are no per-session memory limitations.

Furthermore, you should be aware of your hard disk limits as you work with Athena as you try to install new packages such as Keras or Tensorflow. You can check your available space using the ”quota -v” command.

Solution 2 - Mac and Anaconda

- Download and install Anaconda for **Python 3.6** from `https://www.continuum.io/downloads#osx`.
- We have provided you with a conda environment file (environment.yml) that you can use to install the required packages and all their dependencies.Among the packages are Keras 1.2.2, Theano 0.8.2, Tensorflow 1.0.1 and python 3.5.3. (The python 3.5.3 installation will not affect other versions of python you may have on your computer. It will only be used when you activate this environment.) In the project directory, open terminal and type the following commands:

  ```
  - PATH=$PATH:$HOME/anaconda/bin
  ```
  [11]

  ```
  - conda env create -f environment.yml
  - source activate project2
  ```

  You will see a (project2) prefix in the beginning of every line in your terminal now.

- The ”project2” environment should install both Theano and Tensorflow. To switch between them, you can modify the file `~/.keras/keras.json` before running your code.

  ```
  - cd ~/.keras
  - open keras.json
  ```

  You can edit the file `~/.keras/keras.json` file to choose your preferred backend. The backend line should be either ”theano” or ”tensorflow”, depending on which you want to use. Recall that, as noted above, Tensorflow is only available for 64-bit OSes. If your machine is running a 32-bit OS, please use Theano only.

- To run a project file, for example, mnist_nnet_fc.py, return to the directory that contains your project files and type: `python mnist_nnet_fc.py`
- Every time you close and reopen the terminal, you will have to redo `source activate project2` to enter this environment again.

[11] Note that the path may be different depending on where you chose to install Anaconda. For example:HOME/anaconda3/bin

Solution 3 - Mac/Linux

Run the following commands to install all the necessary packages. Note that this solution works for Mac/Linux only, but may be more prone to trouble. It's essentially the same as solution 1, but simplified assuming you have a working Python3/pip installation (as you should), and sudo access (so you can install packages globally, not just on your account). If you encounter a package version conflict and don't know how to solve it, we suggest you choose solution 1 or 2.

```
sudo pip3 install numpy
sudo pip3 install scipy
sudo pip3 install yaml
sudo pip3 install cython
sudo pip3 install h5py
sudo pip3 install theano
sudo pip3 install keras
```