Department of Electrical and Computer Engineering

ELE 888 / EE 8209 - Intelligent Systems (Machine Learning)

Winter 2017 - Lab #3

Neural Networks and Support Vector Machines

Author: Raymond Phan

Edited by: Md Moinuddin Bhuiyan

Objectives

In this lab assignment, you will investigate the use of Neural Networks, one of the most widely used machine learning constructs seen in practice. You will be implementing neural networks for classification for both the binary and multi-class case. You will also explore varying the amount of hidden layer neurons for both the binary and multi-class case and seeing what effect this has in performance. For the multi-class case, we will also deal with regularization and seeing what effect it has in classification accuracy.

For the binary case, you will implement the Stochastic Gradient Descent method for finding the optimal weights for the neural network to solve one of the most famous problems in machine learning: The XOR problem, which is a classification problem that does not have a decision boundary, linear or non-linear, that can separate between the two classes.

For multi-class case, you will use the aid of optimization tools in MATLAB to help you find the optimal weights to solve a car acceptability problem. Given several features about a car, the goal is to predict whether the car is acceptable for driving. This is to be implemented in a One-Vs-All framework and you will determine the accuracy of this multi-class classification framework by classifying the attributes of cars in a test dataset where these examples were not part of the training data. Finally, you will also explore the use of Support Vector Machines (SVMs) that are part of MATLAB's Statistics and Machine Learning Toolbox to re-solve both the binary and multi-class problems as mentioned previously. You will compare the performance between Neural Networks and SVMs in terms of classification accuracy.

List of Files Included

In this lab assignment, you are given some files with code already completed to help you complete this lab assignment. These are primarily for setup so you can concentrate on actually implementing machine learning algorithms rather than being bogged down with minor minutiae. There are additionally other files that are skeletons where it is your task to complete these files or functions and submit them with your lab report.

Demonstration Scripts

These are scripts that help familiarize yourself with the lab and may help you in completing the required portions of this lab.

- `binary_SVM_demo.m`: This is a quick MATLAB demo on the basics of using MATLAB's SVM toolchain that is part of the Statistics and Machine Learning toolbox for performing binary classification. It loads in a predefined dataset in MATLAB, separates it into two classes, trains a binary classification SVM and performs some example predictions. Note that the default kernel function and the one used in this script is **linear**.

- `multi_class_SVM_demo.m`: Like the previous script, this uses MATLAB's SVM toolchain to perform multi-class classification. It also loads in a predefined dataset in MATLAB which contains three classes. SVM classifiers are trained per class and predictions are made using a One-Vs-All prediction scheme. Note that the default kernel function and the one used in this script is **linear**.

Dataset Files

These are MATLAB MAT files that contain datasets that will be used for this lab. These are stored in the `data` directory of the lab and these files are:

- `lab3cardata.mat`: A real dataset created by Mark Bohanec and Biaz Zupan from the Jožef Stefan Institute - the largest research institute in Slovenia: http://www.ijs.si/. This dataset consists of six (6) features to identify the acceptable rating of a car if it is fit to drive and has 1728 examples. This dataset has been post-processed by your instructor to allow ease of algorithm implementation and has been split up into training and testing sets. The original link to the dataset can be found here on the UCI Machine Learning Dataset Repository Database: http://archive.ics.uci.edu/ml/datasets/Car+Evaluation.

Helper Functions

These are scripts that are designed to help you complete the lab. Some other functions are used to help you give a sense of the problems you are attempting to solve in this lab. These are stored in the `helper` directory of the lab.

The following is a list of functions for each part of this lab.

For Part 1

- `plot_XOR_and_regions.m`: This is a function written to plot the XOR problem and plot the decision regions for each class given two neural network weight matrices: One matrix defined between the input layer and hidden layer and one matrix defined between the hidden layer and output layer.

For Part 2

- `fmincg`: This is a function written by Carl Edward Rasmussen from the University of Cambridge. Some modifications were made by Andrew Ng and further modifications were made by Raymond Phan. Essentially, this finds the minimum of a cost function very much like `fminunc` in MATLAB but uses the Conjugate Gradient method for finding the minimum. This function is included because it is especially optimized for handling data with many features and it is also more memory efficient. This will especially be useful when you perform multi-class classification with the car acceptability dataset included with this lab.

For Part 3

None

For Part 4

None

To complete

You are required to complete an informal lab report regarding the completed tasks of this lab. All questions that pertain to results in the lab, specifically any answers to observational questions as well as calculations and figures to be included are expected to be placed in this report. Please consult the Lab Guidelines document on our course website for more details.

In addition, these are the scripts you need to complete for this lab. Each script will either be a skeleton that you need to complete in functionality or is a function that you need to write with a few lines of code to get you started. These scripts are stored in the `tocomplete` directory of the lab.

- `sigmoid.m`: A function that computes the sigmoid term for every element in an input array or matrix. You should have already completed this in Lab #2. This will **not** be marked but you are to ensure that this is included so you can complete this lab.
- `dsigmoid.m`: A function that computes the **derivative** of the sigmoid function for every element in an input array or matrix.
- `forward_propagation.m`: Assuming that we have a neural network that has an input layer, one hidden layer and an output layer, this function performs forward propagation given a data matrix where each row is an example and each column is a feature, as well as the weight matrices from the input layer to the hidden layer and the hidden layer to the output layer. This code **does not predict the output class** of each example but rather returns the raw outputs of the neurons in the output layer where each row denotes the outputs of each neuron in the output layer for an example. We can call this **a score matrix**.
- `predict_class.m`: Given a score matrix where each row is an example and each column denotes the score for a particular class, either from the output of multi-output neuron neural network or from a score matrix of a SVM, this code predicts the class per example.
- `part1.m`: This is a template to be filled out to complete Part 1 of the lab. This code illustrates solving the XOR problem with Stochastic Gradient Descent using Neural Networks. Some code is already in the file to get you started, but the relevant parts to be completed are highlighted.
- `costFunction_NN_reg.m`: This is the cost function that reflects solving the parameters / weights for a neural network with an input layer, a hidden layer, and an output layer. This function also returns the gradients for each of the weights given an input set of weights. This cost function also implements regularization to prevent overfitting. Given a set of input parameters, the function returns the overall cost required to use those weights in the neural network as well as the gradient vector where each weight has its error gradient evaluated at these current weights.
- `part2.m`: This is a template to be filled out to complete Part 2 of the lab. This code is to create a prediction model for predicting car acceptibility using the provided data as mentioned previously using multi-class classification and neural networks. Combined with `fmincg` and `costFunction_NN_reg.m`, you are to compute the optimal neural network weights for accurately predicting car acceptability. You are also to use `forward_propagation.m` combined with `predict_class` to predict the acceptability of the test dataset. You will also calculate the classification accuracy of the training and test datasets.
- `part3.m`: This reimplements Part 1 of the lab but we will be using SVMs to solve the XOR problem instead. This is an exposition on how to use MATLAB's SVM toolchain from the Statistics and Machine Learning toolbox to help solve binary classification problems.
- `part4.m`: This reimplements Part 2 of the lab but we will be using SVMs to solve the car acceptability problem instead. This is an exposition on how to use MATLAB's SVM toolchain to solve multi-class classification problems instead.

First Half - Neural Networks

Warmup

> *This section is strictly for your benefit. There is nothing to write up about in your report here. You don't have to read this section but it's highly encouraged that you do so you can implement the lab assignment with ease.*

Forward Propagation Algorithm Review

Reviewing from class, when trying to find predictions using a neural network, we use a process called **forward propagation**. We will consider only the linear algebra viewpoint for conciseness and for ease of computation. We will use the following diagram to help explain our notation.

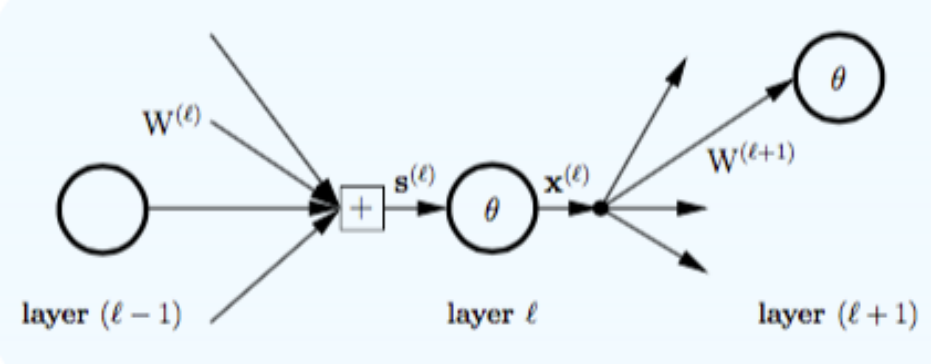

Figure 1 - Diagram to help explain the notation for a neuron in a neural network

This diagram was taken from Yaser Abu Mostafa's *Learning from Data* textbook, Chapter 7 - Neural Networks. Take note that the textbook uses θ to represent the activation function where in our case, we will use g to eliminate confusion.

Common activation functions used in practice are the following:

1. Sigmoid: $g(z) = \frac{1}{1+e^{-z}}$
2. Hyperbolic Tangent (tanh): $g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
3. Perceptron: $g(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$.
4. Linear: $g(z) = z$.

The first three activation functions are commonly used for classification where the last activation function is used for regression. The perceptron activation function was used in the past to mimic the way the human brain worked when neurons fire in the brain but it isn't being used as often anymore.

The notation we will establish is as follows:

1. Let the input layer be represented as a vector of neurons such that $\mathbf{x}^{(0)} = (x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, \dots, x_{d^{(0)}}^{(0)})^T$. The input example has $d^{(0)}$ features and by convention $x_0^{(0)}$ represents the bias unit of the input layer and is traditionally set to 1. Take note that the input example is represented as a **column vector**.

2. Let $W^{(l)}$ be a **matrix** of weights where each row and column entry (i, j) is the weight that connects node i of layer $l-1$ to node j of layer l. Concretely, each element in the matrix $W^{(l)}$ is $w_{ij}^{(l)}$.
3. Let $\mathbf{s}^{(l)}$ represent a column vector of inputs for each neuron of layer l. Therefore, for a particular input into node j at layer l, this can be represented as $s_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$ with the convention that $x_0^{(l-1)} = 1$ to represent the bias unit. Therefore, we can represent every input into the neurons of layer l to be $\mathbf{s}^{(l)} = (s_1^{(l)}, s_2^{(l)}, \dots, s_{d^{(l)}}^{(l)})^T$. Take note that there are no inputs into the bias unit at layer l.
4. Let $\mathbf{x}^{(l)}$ represent a vector of outputs for each neuron of layer l. An output at node j for layer l, $x_j^{(l)}$ is simply applying the activation function to the input into node j or $x_j^{(l)} = g(s_j^{(l)})$. Therefore, we can collect all of these into a vector $x^{(l)}$ such that: $\mathbf{x}^{(l)} = (x_0^{(l)}, x_1^{(l)}, x_2^{(l)}, \dots, x_{d^{(l)}}^{(l)})^T$. Take note that the bias unit sends an output, and by convention $x_0^{(l)} = 1$.

Finally, the forward propagation algorithm is as follows:

1. Initialization: Map the input example $\mathbf{x} = (x_0, x_1, x_2, \dots, x_{d^{(0)}})^T$ to the input layer: $\mathbf{x^{(0)}} \leftarrow \mathbf{x}$. Remember that the input example is formatted as a **column vector**.
2. Perform forward propagation:

$$\begin{aligned} &\text{for } l = 1 \text{ to } L \ \{ \\ &\mathbf{s}^{(l)} = \left(W^{(l)}\right)^T \mathbf{x}^{(l-1)} \\ &\mathbf{x}^{(l)} = \begin{bmatrix} 1 \\ g(\mathbf{s}^{(l)}) \end{bmatrix} \} \end{aligned}$$

3. The prediction can be found at layer L with the vector $\mathbf{x}^{(L)}$. Make sure you remove the bias element, or the first element of $\mathbf{x}^{(L)}$.

When performing predictions with classification, it depends on what scheme of classification you are using. For binary classification, there is only one output neuron and so if you are using the sigmoid function, you threshold the output at 0.5, so any values that are 0.5 or larger, these belong to the positive class while anything else belongs to the negative class. When performing multi-class classification, whichever neuron gives you the highest response, that is the class that is to be assigned to the input.

Backpropagation Algorithm Review

Recall that the cost or error e for a single training example when using a neural network is defined as the following:

$$e = \frac{1}{2}||\mathbf{x}^{(L)} - y||^2 = \frac{1}{2}\left((x_1^{(L)} - y_1)^2 + (x_2^{(L)} - y_2)^2 + \cdots + (x_{d^{(L)}}^{(L)} - y_{d^{(L)}})^2\right)$$

$\mathbf{x}^{(L)}$ is the predicted output of the neural network represented as a vector of features and y is the true output represented as a vector of features. Usually, for the binary classification case, the total number of neurons at the output layer is 1 and for the multi-class classification case, the total number of neurons is equal to the total nubmer of classes expected in the training data.

Reviewing from class, the **backpropagation** algorithm is an efficient way to compute the gradients of the error for a single example. We will consider only the linear algebra viewpoint for conciseness and for ease of computation. We will use the following diagram to help explain our notation. Concretely, we wish to find the error gradient $\partial e \,/\, \partial w_{ij}^{(l)}$ for every weight in our neural network. By the chain rule, the error gradient can be represented as a product of two terms:

$$\frac{\partial e}{\partial w_{ij}^{(l)}} = \frac{\partial e}{\partial s_j^{(l)}} \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} x_i^{(l-1)}$$

The second term originates from the equation of the input node: $s_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$ where differentiating with respect to $w_{ij}^{(l)}$ gives us that single term $x_i^{(l-1)}$. The first term we define as the **sensitivity** which is the error of one example with respect to the corresponding input signal at node j in layer l, or $s_j^{(l)}$. Take note that the sensitivity is **not defined** for the bias unit at each layer because there are no input signals into these bias units so their sensitivities will never change. Therefore, the total number of sensitivities seen at layer l is simply $d^{(l)}$.

If we can represent the output signals of layer $l-1$ as a vector $\mathbf{x}^{(l-1)} = (x_0^{(l-1)}, x_1^{(l-1)}, x_2^{(l-1)}, \dots, x_{d^{(l-1)}}^{(l-1)})^T$ like we did before and the sensitivities of layer l to be a vector $\delta^{(l)} = (\delta_1^{(l)}, \delta_2^{(l)}, \dots, \delta_{d^{(l)}}^{(l)})^T$, we can represent the error gradient of the weights seen at layer l as a **matrix** which is the same size as $W^{(l)}$ such that:

$$\frac{\partial e}{\partial W^{(l)}} = \mathbf{x}^{(l-1)} (\delta^{(l)})^T$$

From class, we know that we can find the sensitivities of the output layer $\delta^{(L)}$ such that:

$$\delta^{(L)} = (\mathbf{x}^{(L)} - y) \otimes g'(\mathbf{s}^{(L)})$$

$\otimes$ stands for the Hadamard or element-wise product. $g'(z)$ is the derivative of the activation function. The derivatives for each of the respective activation functions that we've seen above are:

1. Sigmoid: $g'(z) = g(z)(1 - g(z))$
2. Hyperbolic Tangent (tanh): $(1 - (g(z))^2)$
3. Linear: $g'(z) = 1$.

Depending on what activation function you're using, you can represent the derivative of each using combinations of operations with the original activation function. Take note that the derivative of the perceptron function is undefined at $z = 0$ and so this is another reason why the perceptron function isn't used in neural networks as of late.

We apply the derivative of the activation function to each element in $\mathbf{s}^{(L)}$ individually. We also know that if we want to find the sensitivities in a layer other than the output layer, we use the following recursive relationship:

$$\delta^{(l)} = g'(\mathbf{s}^{(l)}) \otimes (\tilde{W}^{(l+1)} \delta^{(l+1)})$$

$\tilde{W}^{(l+1)}$ is the weight matrix **before performing any updates** at layer $l+1$ **without** the first row. This are the original matrices before any updates are applied. This applies for layers $l = L-1, L-2, \dots, 2, 1$.

Therefore, the final backpropagation algorithm for a single example is thus:

1. Initialization: Map the input example $\mathbf{x} = (x_0, x_1, x_2, \dots, x_{d^{(0)}})^T$ to the input layer: $\mathbf{x^{(0)}} \leftarrow \mathbf{x}$. Remember that the input example is a **column vector**.
2. Perform forward propagation: Remember each $\mathbf{s}^{(l)}$ and $\mathbf{x}^{(l)}$ for $l = 1, 2, \dots, L$.
3. Find the sensitivity vector for output layer L: $\delta^{(L)} = (\mathbf{x}^{(L)} - y) \otimes g'(\mathbf{s}^{(L)})$.
4. Find the sensitivity vectors for the other layers: $\delta^{(l)} = g'(\mathbf{s}^{(l)}) \otimes (\tilde{W}^{(l+1)} \delta^{(l+1)})$ for $l = L-1, L-2, \dots, 2, 1$.
5. Compute the error gradients for the weights: $\partial e \,/\, \partial W^{(l)} = \mathbf{x}^{(l-1)} (\delta^{(l)})^T$ for $l = 1, 2, \dots, L$.

Training a Neural Network - Stochastic Gradient Descent

Now that we have forward and back propagation laid out for a single example, we now lay out the algorithm to train a neural network with Stochastic Gradient Descent.

Given a set of m training examples: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$ and a learning rate α, the algorithm is as follows.

1. Initialization:
    a. Set the weight matrices $W^{(1)}, W^{(2)}, \dots, W^{(L)}$ to have small random values. This is to ensure that all of the neurons in the network will behave properly to classify new inputs. Setting all of the weights to be equal to each other or setting them all to zero will have unintended side effects.
    b. Create an array of costs J and initialize all elements to 0.
2. Randomly shuffle the training examples. The purpose is to ensure that that we escape local minima as we are operating on one input at a time.
3. `for i = 1, 2, ..., K` ← repeat K times. Each of these iterations is known as an **epoch**.
    a. $J[i] \leftarrow 0$
    b. `for j = 1, 2, ... m` ← for each training example j
        a. Set the input layer $\mathbf{x}^{(0)}$ to be training example j, $\mathbf{x}^{(0)} \leftarrow x^{(j)}$
        b. Compute forward propagation and save every $\mathbf{x}^{(l)}$ and $\mathbf{s}^{(l)}$ for layer $l = 1, 2, \dots, L$.
        c. Compute backpropagation and save $\delta^{(l)}$ and $\partial e \,/\, \partial W^{(l)}$ for layer $l = L-1, L-2, \dots, 2, 1$.
        d. Update the weight matrices: $W^{(l)} \leftarrow W^{(l)} - \alpha \frac{\partial e}{\partial W^{(l)}}$ for layer $l = 1, 2, \dots, L$.
        e. Accumulate the cost: $J[i] \leftarrow J[i] + \frac{1}{2m} \|\mathbf{x}^{(L)} - y^{(i)}\|^2$
4. Final weights are stored in $W^{(1)}, W^{(2)}, \dots, W^{(L)}$.

Take note that the costs should generally decrease or perhaps stay the same as the epochs progress. You can then use the new weight matrices to predict new examples using forward propagation.

For the case of binary classification, the output layer only has one neuron so the algorithm above is quite clear. For the case of multi-class classification, the expected vector $y^{(i)}$ is a binary vector that is the same size as the total amount of neurons in the output layer. This amount is the total number of classes expected to be seen in the training data. This binary vector is such that every element is all zero **except** for one element. The position of this non-zero element is the class that the training example belongs to, and this is set to 1. In other words, the expected vector $y^{(i)}$ is a **column vector** such that $y^{(i)} = (0, 0, 0, \dots, 1, \dots, 0)^T$ where every element is zero except for the position $i = k$ where k is the class that training example $x^{(i)}$ belongs to. Take note that the classes range between $[1, N]$ where N is the total number of classes as opposed to $[0, N-1]$ as we have seen in Lab #2. As this is a vector of elements, $\|x^{(L)} - y^{(i)}\|$ is the length of the vector that is the difference between the predicted output of training example $x^{(i)}$ with its expected output $y^{(i)}$ converted into a binary vector.

Training a Neural Network - Using Optimization Tools

To efficiently train a neural network, we can take advantage of using optimization tools such as `fmincg` to allow convergence of the trained weights to be faster. The following algorithm lays out how to train a neural network to be placed in a cost function that is suitable to be optimized with mechanisms such as `fmincg`. Specifically, we need to evaluate the overall cost J to assign an input set of weights to the neural network from the weight matrices $W^{(1)}, W^{(2)}, \dots, W^{(L)}$ as well as the gradient vector of the cost function which contains elements evaluated for every weight in the neural network.

Given a set of m training examples: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$ and the input set of weights as discussed above the algorithm is as follows. Take note that this algorithm **includes** regularization to reduce overfitting so we are also given the regularization parameter λ.

1. Initialization: Set $J = 0$ and create weight update matrices $W^{(l)}_{update}$ such that $W^{(l)}_{update} = 0 \cdot W^{(l)}$, for $l = 1, 2, \ldots, L$. Essentially, we create matrices of all zero that are of the same size as the weight matrices. Also randomly shuffle the training examples.
2. Compute gradient and cost **without regularization**

   `for i = 1, 2, ..., m` ← for each training example i
   a. Assign the input layer to the training example $x^{(i)}$. $\mathbf{x}^{(0)} \leftarrow x^{(i)}$.
   b. `for l = 1, 2, ... L` ← for each layer l, perform forward propagation and save $\mathbf{s}^{(l)}$ and $\mathbf{x}^{(l)}$ for all layers.
      1. $\mathbf{s}^{(l)} = \left(W^{(l)}\right)^T \mathbf{x}^{(l-1)}$
      2. $\mathbf{x}^{(l)} = \begin{bmatrix} 1 \\ g(\mathbf{s}^{(l)}) \end{bmatrix}$
   c. Remove bias from $\mathbf{x}^{(L)}$ or the first element
   d. Find the sensitivity vectors for all of the layers. First, find the sensitivity vector for output layer L: $\delta^{(L)} = (\mathbf{x}^{(L)} - y) \otimes g'(\mathbf{s}^{(L)})$.
   e. Find the sensitivity vectors for the other layers: $\delta^{(l)} = g'(\mathbf{s}^{(l)}) \otimes (\tilde{W}^{(l+1)}\delta^{(l+1)})$ for $l = L-1, L-2, \ldots, 2, 1$. $\tilde{W}^{(l+1)}$ is the weight matrix for layer $l+1$ **before** the update (i.e. from the input of this algorithm) and **removing the first row of the matrix**.
   f. Compute the error gradients for the weights: $\partial e \,/\, \partial W^{(l)} = \mathbf{x}^{(l-1)}(\delta^{(l)})^T$ for $l = 1, 2, \ldots, L$.
   g. Accumulate the gradient and cost **without regularization**.
      1. `for l = 1, 2, ... L` ← for each layer l
         $W^{(l)}_{update} \leftarrow W^{(l)}_{update} + \frac{1}{m}\frac{\partial e}{\partial W^{(l)}}$
      2. $J \leftarrow J + \frac{1}{2m}\|\mathbf{x}^{(L)} - y^{(i)}\|^2$
3. Add in regularization to the cost and gradient. This is done once you loop over all training examples:
   a. `for l = 1, 2, ..., L`

      $$W^{(l)}_{update} \leftarrow W^{(l)}_{update} + \frac{\lambda}{m}\begin{bmatrix} \vec{\mathbf{0}} \\ \tilde{W}^{(l)} \end{bmatrix}$$

      $\tilde{W}^{(l)}$ is the weight matrix for layer l **before** the update (i.e. from the input of this algorithm) and **removing the first row of the matrix**. $\vec{\mathbf{0}}$ is a row vector of all zeroes that has the length equal to the total number of columns in $W^{(l)}$.
   b. $J \leftarrow J + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{d^{(l-1)}}\sum_{j=1}^{d^{(l)}}(w^{(l)}_{ij})^2$

      Recall that $w^{(l)}_{ij}$ is the weight from node i of layer $l-1$ and node j from layer l. This states that we square each of the weights **before** the update and we sum these weights **without the bias nodes**. We never apply regularization to the bias units just like we don't apply regularization to the bias term in linear or logistic regression.
4. The desired output is stored in J and $W^{(l)}_{update}$ for layer $l = 1, 2, \ldots, L$ that is to be sent to the output. Take note that we have to pack all of the weight update matrices into a single vector, but we will talk about that later when you actually implement the algorithm.

Performing Multiple Predictions using Linear Algebra

To wrap things up, in class we derived a way to efficiently compute multiple predictions in a neural network simultaneously, rather than submitting one example at a time. Given a data matrix of examples X where each row is an example and each column is a feature, or corresponding to an input into a neuron in the input layer, the algorithm is as follows:

1. Set $\hat{X}^{(0)} = \begin{bmatrix} \vec{\mathbf{1}} & X \end{bmatrix}$. $\vec{\mathbf{1}}$ is a column vector of ones that has as many rows as there are in X.
2. For each layer $l = 1, 2, \ldots, L$
    a. Compute the matrix $S^{(l)} = \hat{X}^{(l-1)} W^{(l)}$
    b. Compute the matrix $\hat{X}^{(l)} = \begin{bmatrix} \vec{\mathbf{1}} & g(S^{(l)}) \end{bmatrix}$. The activation function is applied to every element in the matrix $S^{(l)}$.
3. The output predictions are in matrix $\hat{X}^{(L)}$. Remove the first column of all ones and each row consists of the predicted values for the corresponding input example. Each column is the output of a neuron in the output layer.
4. If we are performing the regression task, the activation functions are linear and so you would use the raw output directly. If we are performing the classification task, if this is the binary classification problem, we threshold the output where any output values that are 0.5 or larger get classified as being the positive class while anything less than 0.5 is the negative class. This is assuming we are using the sigmoid activation function. For the multi-class classification task, we examine all of the output neuron outputs and determine which neuron yielded the highest output. Whichever neuron gave the highest response will dictate what the predicted class for the input example.

1 - Solving the XOR Problem using Stochastic Gradient Descent

1.1 - Introduction

In this section, you will implement Stochastic Gradient Descent to train a neural network for the purposes of solving the XOR problem. The XOR problem is a set of 4 training examples that uses 2 features with associated outputs where the data is not separable by any known decision boundary that we have seen in the course so far. Concretely, the XOR problem can be represented as the two-input logic gate that mimics the Exclusive OR (hence XOR) operation. Specifically, given two input features x_1 and x_2, and the output class y which will either be 0 or 1, we have the following four possibilities:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Pictorially, if we plot these points on the two-dimensional Cartesian plane and assign different markers to each of the examples that belong to their corresponding classes, we get this image:

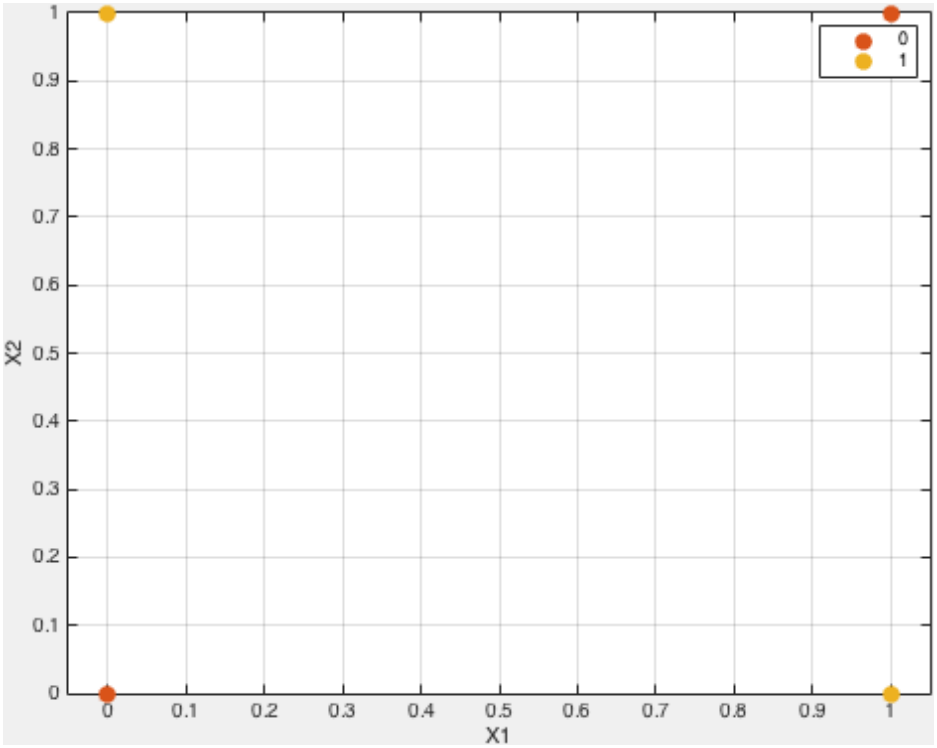

Figure 2 - Graphical representation of the XOR problem

As you can clearly see, there is no decision boundary we know of currently that can separate between the two classes. Therefore, instead of thinking of a decision boundary, we'll need to think of decision **regions** that can successfully separate the two classes. By implementing Stochastic Gradient Descent to train a neural network on the above problem, we will be able to succesfully solve this problem.

Before we do any of that, we will need to set up some preliminary functions to help us complete this task.

1.2 - Implement the Sigmoid Function

Deliverables

The activation function that we will be using in this neural network problem will be the sigmoid function. This file is **not** for marks as you have already implemented this in Lab #2. As such, this part exists just to remind you that you will need to include that implementation in Lab #3. Simply copy and paste your code into the `sigmoid.m` file that is seen in the `tocomplete` directory or simply overwrite this file with your own version from Lab #2 in this directory.

1.3 - Implement the Derivative of the Sigmoid Function

In order to compute backpropagation, we require that the derivative of the activation function be completed.

Deliverables

Since we are choosing the activation function to be the sigmoid function, your task in this part is to complete the `dsigmoid.m` file found in the `tocomplete` directory so that it computes the derivative of the sigmoid function for each element in the input, whether it be a single value, vector or matrix. The file is currently a skeleton and your objective is to complete the file with the previously mentioned behaviour.

Recall that the derivative of the sigmoid function $g'(z)$ can be represented as:

$$g'(z) = g(z)(1 - g(z))$$

Remember that $g(z) = \frac{1}{1+\exp(-z)}$. Therefore, the function has the following input and output specifications.

Inputs

- `z`: A matrix, vector or single value of any size that contains real numbers.

Outputs

- `g`: A matrix, vector or single value that is the same size as the input where the derivative of the sigmoid function is applied to every element in this input individually.

> *Tip: Consider using **element-wise** operators to do this efficiently (i.e. `.*, ./, .^`, etc.) instead of using a `for` loop. Most of the time, MATLAB works best if you avoid using `for` loops.*

1.4 - Implement Forward Propagation using Multiple Inputs with Linear Algebra

There will be several cases in this lab where this kind of computation will be used. Specifically, your task is to implement forward propagation where multiple input examples are provided simultaneously. These examples are provided in a data matrix where each row is an example and each column is a feature. It should be noted that you will **not** perform the final prediction stage here where you choose which class each input belongs to as we have seen in the past. What will be returned instead is the **raw output** of each neuron in the output layer for each training example.

Deliverables

The function that you will write is to be completed in the file `forward_prediction.m` file in the `tocomplete` directory of the lab assignment . The file is currently a skeleton and your objective is to complete the file with the previously mentioned behaviour. In this lab, we will be using only one input layer, one hidden layer and one output layer for all of the neural network architectures for the Neural Networks portion. Therefore, we will only need to have two weight matrices $W^{(1)}$ and $W^{(2)}$. As such, given the weight matrices $W^{(1)}$ and $W^{(2)}$ as well as the data matrix we discussed earlier, your task is to implement forward propagation using linear algebra. Note that the **raw output** of the output neurons is what is expected to be returned, not the actual classes of each input itself. We call this matrix a **score matrix** as this determines how likely each example belongs to each of the possible classes by considering the magnitude of the outputs of each neuron.

The function has the following input and output specifications.

Inputs

- `X`: The data matrix of size `m x n`, where `m` is the number of examples and `n` is the number of features. Each row is an example and each column is a feature. Unlike what we have seen before, it's important that you **do not** append a column of ones to the matrix before you call the function. This behaviour will be implemented internally.
- `W1`: The weight matrix $W^{(1)}$ which defines the weight relationships between the input layer and hidden layer. This matrix is of size `(d0 + 1) x d1` where `d0` are the number of neurons in the input layer **without**

the bias unit and `d1` is the number of neurons in the hidden layer **without the bias unit**. Each row and column pair (i, j) is the weight between neuron i from the input layer to neuron j in the hidden layer.
- `W2`: The weight matrix $W^{(2)}$ which defines the weight relationships between the hidden layer and output layer. This matrix is of size `(d1 + 1) x d2` where `d1` are the number of neurons in the hidden layer **without the bias unit** and `d2` is the number of neurons in the output layer **without the bias unit**. Each row and column pair (i, j) is the weight between neuron i in the hidden layer to neuron j in the output layer.

Outputs

- `Y`: An output prediction matrix of the **raw output** of the output neurons or the **score matrix**. This is of size `m x d2` where `m` is the number of examples and `d2` is the number of neurons in the output layer **without the bias unit**. Therefore, each row is the predicted raw output for each corresponding input example from the data matrix and each column represents the output of an output neuron. Each column j is the output of the neuron $x_j^{(2)}$ for all examples. (i.e. The first row of `Y` is the predicted raw output of the first example of `X` - the first row. The second row of `Y` is the predicted raw output of the second example `X` - the second row, etc.)

Hints

1. Because we have a fixed number of layers and also a very small amount of layers, writing a `for` loop to compute the inputs and outputs at each layer is unnecessary. The algorithm using linear algebra is thus reduced to the following:
    a. Set $\hat{X}^{(0)} = \begin{bmatrix} \vec{\mathbf{1}} & X \end{bmatrix}$ where $\vec{\mathbf{1}}$ is a **column** vector of ones that spans the same size as the total number of examples in `X`.
    b. Compute $S^{(1)} = \hat{X}^{(0)} W^{(1)}$
    c. Compute $\hat{X}^{(1)} = \begin{bmatrix} \vec{\mathbf{1}} & g(S^{(1)}) \end{bmatrix}$ where $g(S^{(1)})$ is applying the activation function (i.e. the sigmoid) to every element in $S^{(1)}$.
    d. Compute $S^{(1)} = \hat{X}^{(1)} W^{(2)}$
    e. Compute $X^{(2)} = Y = g(S^{(2)})$. $X^{(2)}$ contains the desired raw output matrix to be returned from the function. The function's output variable is stored in `Y`, and is also the output seen at the output layer. Therefore, $X^{(2)} = Y$ in this case.
2. Because of the reduced set of steps for the algorithm, the code should only be a few lines long.

1.5 - Implement the Class Prediction Process for Neural Networks using One-Vs-All

Leading from Section 1.4, the function `forward_prediction.m` computes the raw outputs of the output layer neurons for each training example supplied to it in a matrix. Each row of the output tells you what the predicted raw outputs are for each example from the data matrix. This is what is known as a **score matrix**. The goal of this function now is to take this score matrix that is output from `forward_prediction.m` and to predict the class that each input from the data matrix supplied to `forward_prediction.m` belongs to.

Deliverables

The function that you will write is to be completed in the file `predict_class.m` and can be found in the `tocomplete` directory of the lab assignment. The file is currently a skeleton and your objective is to complete the

file with the previously mentioned behaviour. The function takes in the score matrix that is output from the `forward_prediction.m` function and it outputs the predicted class for each training example referencing the data matrix `X` that was supplied to `forward_prediction.m`. This is to be done using the One-Vs-All approach. Remember that for the score matrix, the higher the output, the more likely the input example belongs to that class. Therefore, whichever neuron gives you the highest response for that particular example, the neuron that is trained to classify for that particular class is what we should predict the final class to be.

The function has the following input and output specifications.

Inputs

1. `Y`: The output prediction matrix of the **raw output** of the output neurons for each training example we wanted to use when using the `forward_propagation.m` function. This is of size `m x d2` where `m` is the number of examples and `d2` is the number of neurons in the output layer. Therefore, each row is the predicted raw output for each corresponding input example from the data matrix and each column represents the output of an output neuron. Each column j is the output of the neuron $x_j^{(L)}$ for all examples. (i.e. The first row of `Y` is the predicted raw output of the first example of `X` - the first row. The second row of `Y` is the predicted raw output of the second example `X` - the second row, etc.)

Outputs

1. `classes`: This will be a `m x 1` column vector where each element predicts what the class would be for each training example we wanted to use when using the `forward_propagation.m` function.

Hints

1. You implemented something very similar in Lab #2. Take a look at the `multiclass_predict.m` function you wrote in Lab #2.
2. The score matrix is structured in such a way where each row determines the raw output of the output neurons for the corresponding training example. To determine which class each example belongs to, scan through every column of each row and determine which column has the largest value. Whichever column gives you the largest value for a row, this column number corresponds to the predicted class to be chosen. Have a look at MATLAB's `max` function, especially the two variable output version.

1.6 - Back to the XOR Problem

1.6.1 - Introduction

Now that we have established the majority of functions you will be using to help you implement neural networks for solving the XOR problem, now it's time to go back and finally solve it. Our job in this section is to solve the XOR problem using the method of Stochastic Gradient Descent as we have seen in class, as well as the Warmup section in this lab assignment.

The neural network architecture we will choose to solve this problem is a three layer network consisting of an input layer, one hidden layer and an output layer (as previously mentioned, we will be assuming this architecture for all neural network questions in this lab assignment). The input layer has two neurons with a bias unit, the hidden layer also has two neurons with a bias unit and the output layer has one output neuron. Take note that the input layer and output layer neurons are to remain the same for the purposes of this part. The number of hidden layer neurons will be varied as an experiment that we will perform once you complete `part1.m`.

The initial neural network architecture just described can pictorially be represented in the figure on the next page.

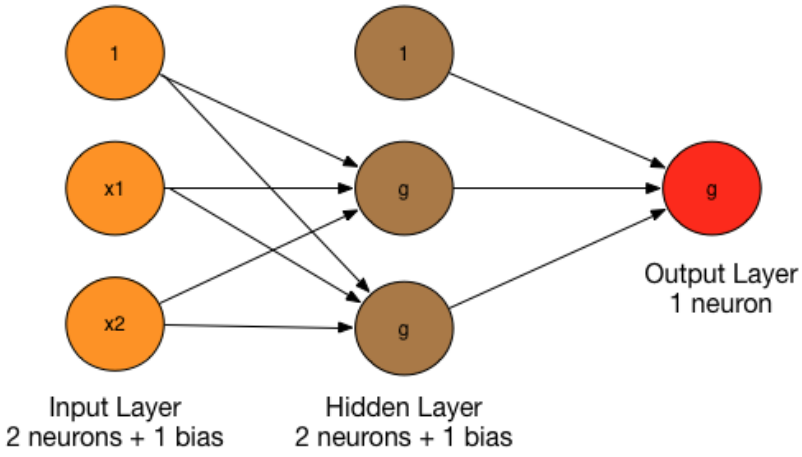

Figure 3 - Neural Network architecture to solve the XOR problem

The activation function we will be choosing in this problem is the sigmoid function.

Deliverables

In this part, you are required to complete the functionality of `part1.m` which will perform the procedure of solving this problem. The code is split up into multiple steps. Some of the steps have already been completed. For the other steps, it will be your task to complete these steps to ultimately solve the problem. The details for each step will be explained in further detail.

For reference purposes, the code listing for `part1.m` is shown below:

```
%%% 1. Clear all variables and close all figures
%%% DON'T CHANGE
clearvars;
close all;
addpath('../helper');

%%% 2. Input training examples
%%% DON'T CHANGE
X = [0 1; 1 1; 1 0; 0 0];
y = [1;0;1;0];

%%% 3. Initialize weight matrices

%%% Number of input neurons
%%% DON'T CHANGE
input_neurons = 2;

%%% Number of hidden layer neurons
%%% This you can change
hidden_neurons = 2;

%%% Number of output layer neurons
%%% DON'T CHANGE
output_neurons = 1;
```

```
%%% DON'T CHANGE
% W1 is a 3 x X matrix - 2 + 1 input neurons, X hidden layer neurons
rng(123);
e_init_1 = sqrt(6) / sqrt(input_neurons + hidden_neurons);
W1 = 2*e_init_1*rand(input_neurons + 1,hidden_neurons) - e_init_1;

% W2 is a (X + 1) x 1 matrix - X + 1 hidden layer neurons, 1 output layer neuron
e_init_2 = sqrt(6) / sqrt(hidden_neurons + output_neurons);
W2 = 2*e_init_2*rand(hidden_neurons + 1,output_neurons) - e_init_1;

%%% 4. Repeat k times
%%% DON'T CHANGE
k = 150;

%%% 5. Some relevant variables
%%% DON'T CHANGE
m = size(X,1);
n = size(X,2);

%%% 6. Initialize cost array
%%% DON'T CHANGE
costs = zeros(k,1);

%%% 7. Set learning rate
%%% DON'T CHANGE
alpha = 5;

%%% 8. Implement Stochastic Gradient Descent
%%% PLACE YOUR CODE HERE

%%% 9. Plot the XOR points as well as the decision regions
%%% PLACE YOUR CODE HERE

%%% 10. Plot the cost per iteration
%%% PLACE YOUR CODE HERE
```

There are 10 parts to this file that implement the process of solving the XOR problem. Parts 1 through 7 are already completed for you. Parts 8 through 10 are what need to be completed by you. The detailed description of each part now follows.

1. ***(Completed for you)*** This is standard procedure where before we start running any code, we clear all of the variables created before the behaviour of this script is run as well as closing all open figure windows. We also add the `helper` and `data` directories to MATLAB's search path so you don't have to copy and paste any functions from those directories into your working directory to access them. Do **not** change any code here.
2. ***(Completed for you)*** We also create the training examples and expected outputs. We create a `4 x 2` matrix `X` that stores the desired input features and the desired output labels are stored in `y`. Remember, in `X` each row is an example and each column is a feature. We have two features and four examples. Note that the order of how the input and outputs are declared are slightly different than what was seen in the table in the introduction of this section. This is to simulate the random shuffling of the input training examples. Do **not** change any code here.
3. ***(Completed for you)*** The first step of the process is to initialize the weight matrices so that the elements are small. Remember, as we have only a three layer neural network, we only have two weight matrices to

consider. Initially, the first matrix $W^{(1)}$ is of size `3 x 2` where each row of $W^{(1)}$ denotes the source node from the input layer and each column denotes the target node to the hidden layer. We have an additional row to account for the bias unit. The second matrix $W^{(2)}$ is initially a column vector of size `3 x 1` which is represents the relationship between the connectivity or neurons between the hidden layer and output layer. The spatial relationships of the elements in this matrix is similar to what was mentioned with the first matrix $W^{(1)}$. In the `part1.m` template, the weight matrices are stored in the variables `W1` and `W2` respectively.

You'll also notice that there are some constants that determine how many input neurons, hidden layer neurons and output neurons there are. The number of input neurons and output neurons should not be touched during this part of the lab. However, the number of hidden layer neurons will be varied in order to assess what happens when this number changes. A good "rule of thumb" to determine the magnitude of how small each of the weights in each weight matrix is can be determined by the following relationship as noted by Yoshua Bengio from the University of Montreal, Canada: http://www.iro.umontreal.ca/~bengioy/ift6266/H12/html.old/mlp_en.html. For a weight matrix $W^{(l)}$, the weights should be randomly initialized and should be restricted to within the following range:

$$W^{(l)} \in \left[-\frac{\sqrt{6}}{\sqrt{d^{(l-1)} + d^{(l)}}}, \frac{\sqrt{6}}{\sqrt{d^{(l-1)} + d^{(l)}}} \right]$$

Recall that $d^{(l)}$ is the number of neurons into layer l without the bias unit. By letting the value $\epsilon_{init}^{(l)} = \frac{\sqrt{6}}{\sqrt{d^{(l-1)}+d^{(l)}}}$, this means that the expected weights should be randomly initialized at layer l to ensure that they are within the range of $[-\epsilon_{init}^{(l)}, \epsilon_{init}^{(l)}]$. This is what this part of the code is doing. We initialize the weight matrices so that they conform to this specification. First the function `rng` is used to **seed** the random function generator. Seeding the random function means that the same sequence of random numbers are generated regardless of which computer you run any random functions on. This is to allow for **reproducible** results, and also to ensure all students get more or less the same results.

The function `rand` in MATLAB creates random matrices of a desired size and have a dynamic range between `[0,1]`. By multiplying each matrix by $2\epsilon_{init}^{(l)}$ and subtracting this result by $\epsilon_{init}^{(l)}$, we would thus achieve a random weight matrix that has its dynamic range between $[-\epsilon_{init}^{(l)}, \epsilon_{init}^{(l)}]$.

It is imperative that you **do not** change any code here for consistency amongst all students.

4. ***(Completed for you)*** Recalling from Stochastic Gradient Descent, we need to have a set number of epochs so we can iterate through all of the training examples and create updates after each example. For consistency, we have set the total number of epochs to 150, or `k = 150`. Do **not** change this variable.

5. ***(Completed for you)*** Some variables to help you with the process of completing this part are extracting the total number of training examples, which is stored in the variable `m` and the total number of features, which is stored in `n`. These should both be 4 and 2 respectively. Do **not** change these variables.
6. ***(Completed for you)*** This creates a cost array called `costs` which is as large as the total number of epochs we have declared. The purpose of this is to record the average error or cost per epoch, or to calculate $J[i]$ as seen in the algorithm description in the Warmup section of this part. You will need to use this array for completing the next parts. Do not **change** this variable.
7. ***(Completed for you)*** We will be assuming a learning rate $\alpha = 5$ for this part of the lab. This is stored in a variable called `alpha` for you to use. Do **not** change this variable.
8. ***(To be completed)*** Up to this point, we have the following relevant MATLAB variables for you to use:

    a. `X`: The training examples for the XOR problem as a `4 x 2` matrix. Each row is an example and each column is a feature.
    b. `y`: The expected labels for the XOR problem as a `4 x 1` column vector

c. `hidden_neurons`: The total number of hidden neurons in the hidden layer
d. `W1`: The initial weight matrix $W^{(1)}$
e. `W2`: The initial weight matrix $W^{(2)}$
f. `k`: The total number of epochs
g. `m`: The total number of training examples
h. `n`: The total number of features
i. `costs`: The cost array to store the average cost / error at each epoch.
j. `alpha`: The learning rate for Stochastic Gradient Descent

Using the above variables as well as referencing the algorithm for Stochastic Gradient Descent, your task for this part is to implement Stochastic Gradient Descent to solve the XOR problem. Ensure that the **number of hidden neurons is set to 2** for this part of the lab. We will vary this later in a later part of the lab. It is expected that the final weight matrices will be updated and stored into the same `W1` and `W2` variables that were created initially. In addition, `costs` should be populated with the average error / cost per epoch.

> ***Hints:***
> ***a.*** *You're going to need at least two* `for` *loops: One to iterate over the epochs and one to iterate over the training examples.*
> ***b.*** *Remember that the training examples in* `X` *are stored per* ***row*** *yet the algorithm assumes each training example is a* ***column****. As such, it may help you to extract out the row of interest from* `X` *and* ***transpose*** *the row to a column so that you can implement the algorithm with ease.*
> ***c.*** *Remember that you are going to have to save each* $\mathbf{x}^{(l)}$ *and* $\mathbf{s}^{(l)}$ *when performing forward propagation. Therefore, using* `forward_propagation.m` *to try and save time is not applicable here so you shouldn't use it.*
> ***d.*** *Remember to use the* `costs` *array to accumulate the costs at each epoch and assigning the accumulation to the right element in* `costs`*.*
> ***e.*** *Remember that you will need to update the weight matrices* `W1` *and* `W2` *at each iteration.*
> ***f.*** *Part of the back propagation algorithm requires that you perform matrix-vector multiplication with the right weight matrices* ***without*** *the first row. To remove the first row from a matrix and leave the rest intact, you can use MATLAB indexing where you sample from the second row to the end. Specifically, you can use* `Wtilde = W(2:end,:);` *where* `W` *is a weight matrix and* `Wtilde` *is the output matrix with the first row removed.*

9. ***(To be completed)*** Once you have the final weight matrices, it's now time to plot the decision **regions**. Our training examples vary only between `[0,1]` and so all of the possible inputs that we should consider are defined within a square such that $0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1$. As such, it will be prudent to create a **square** of features defined with the previous range and predicting the class that each pair of features within this square of features belongs to. By doing this for all pairs of features within this square, we can visualize decision regions that tell us which areas of the square would be considered as the negative class $y = 0$ or the positive class $y = 1$.

   The function `plot_XOR_and_regions.m` is designed to do this for us. You are to use this function in `part1.m` so that it will create a figure with the training examples in the XOR problem plotted as well as the decision regions placed on top of this plot.

   The code listing for this function is shown below:

```
function plot_XOR_and_regions(W1, W2)

%%% 1. Define inputs
X = [0 1; 1 1; 1 0; 0 0];

%%% 2. Define a set of coordinates for each feature between 0 and 1 in
%%% steps of 0.001.
[X1,X2] = meshgrid(0:0.001:1,0:0.001:1);

%%% 3. Convert into column vectors and create a new input matrix that stacks these
%%% columns together
X1 = X1(:);
X2 = X2(:);
XVALS = [X1 X2];

%%% 4. Compute raw outputs from the output layer
%%% NOTE: This requires that forward_propagation be completed successfully
h = forward_propagation(XVALS, W1, W2);

%%% 5. Predict which class each input belongs to
Y = h >= 0.5;

%%% 6. Spawn new figure
figure; hold on;

%%% 7. Plot the actual training examples
% y = 0 are red crosses
% y = 1 are blue circles
plot([X(1,1) X(3,1)], [X(1,2) X(3,2)], 'rx', 'MarkerSize', 16);
plot([X(2,1) X(4,1)], [X(2,2) X(4,2)], 'bo', 'MarkerSize', 16);

%%% 8. Plot decision regions
gscatter(X1, X2, Y, [0.85 0.325 0.098; 0.9290 0.6940 0.1250]);
axis tight;
```

Given the two weight matrices `W1` and `W2` after the algorithm finishes, `plot_XOR_and_regions` takes in these matrices and draws the decision boundaries defined for the XOR problem. Each of the steps that the function takes is outlined below:

a. The training examples for the XOR problem are defined.
b. A set of coordinates / features that are defined within the square of $0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1$, with a small enough step size of 0.001 to ensure that the visualization of the decision regions appear continuous. The function `meshgrid` performs this for us where it takes in two vectors. The first vector is for the first feature x_1 and we define this vector to go from 0 to 1 in steps of 0.001. We do the same thing for x_2 and that is the second input into `meshgrid`. The outputs are matrices `X1` and `X2` where each spatial position `(i,j)` shared between `X1` and `X2` are unique features x_1 and x_2.
c. We transform these matrices so that they are single column vectors by stacking all of the columns in `X1` together into a single column and we do the same for `X2`. We place these columns together in a new data matrix `XVALS` so that each row is an example defined within the square we talked about previously. This data matrix will thus be compatible to use linear algebra for performing multiple predictions.
d. We compute the raw outputs from the output layer given the weight matrices `W1` and `W2` and with the data matrix created previously. *Take note that you **must** complete `forward_propagation.m` in*

order for this code to work.

e. We use the raw outputs from `forward_propagation.m` and threshold these outputs to determine which class each input belongs to.
f. We create a new blank figure and we use `hold on;` so that multiple invocations to `plot` or any of the plotting functions available in MATLAB don't clear the window. These additions will get added on top of the figure.
g. We first plot the actual training examples in this plot and ensure that each class that belongs to each training example get a different marker. The examples where `y = 0` are marked with blue circles crosses and the examples where `y = 1` are marked with red crosses.
h. We finally plot the decision regions. The function `gscatter` in MATLAB takes in an array of `x` coordinates, an array of `y` coordinates as well as which group each corresponding pair of `(x,y)` values belongs to. The fourth argument determines the colour to be placed at the `(x,y)` coordinate depending on which group that coordinate belongs to. Therefore, one colour is assigned to examples where they were classified as the negative class and another colour is assigned to examples where they were classified as the positive class. We ensure that the graph is tight via `axis tight;` to pack the plot as tightly as possible for a compact representation.

If you trained the neural network properly and run this function, you should get something like you seen below:

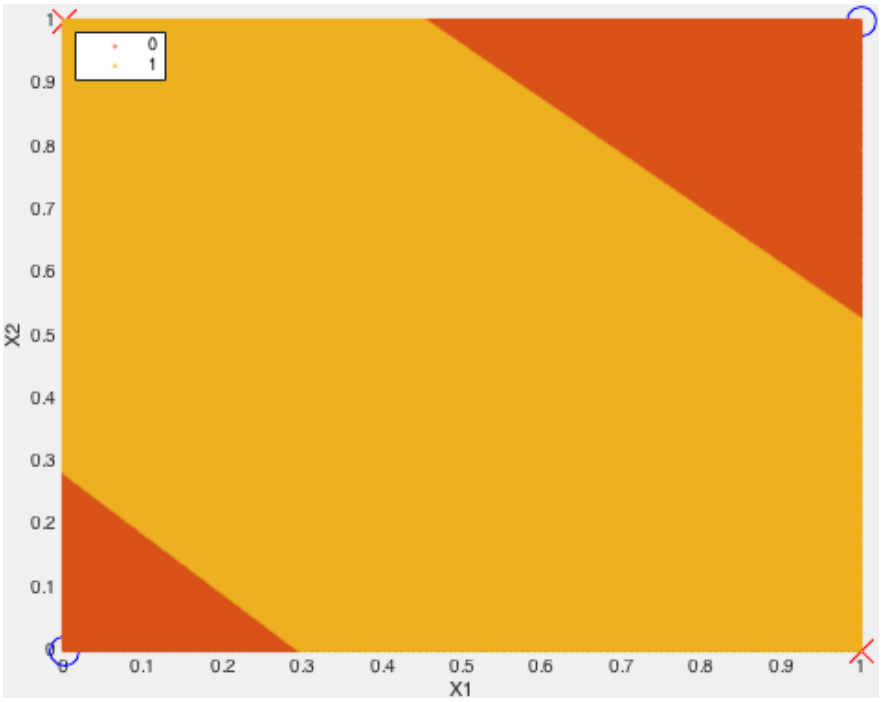

Figure 4 - Example figure showing the decision regions if the neural network is trained correctly

Note that the training examples are separated properly into their corresponding decision regions.

> ***Hint:*** *There's no overthinking this part. It's only one line of code and that is to simply call this function so that the figure above is produced.*

10. **(To be completed)** The last part is to take the `costs` array that was populated from the Stochastic Gradient Descent algorithm and to plot the costs as a function of the epoch. Create a new figure that shows what each cost was at each epoch in the algorithm.

1.7 - Experimentation with the Number of Hidden Neurons in the Hidden Layer

Now that you have a working Stochastic Gradient Descent algorithm to solve the XOR problem, one aspect we have not explored yet is to vary the number of hidden neurons in the hidden layer. If you examine the `part1.m` script that you completed, there is a variable called `hidden_neurons` that you can vary to change the number of hidden neurons in the input layer. Note that the weight matrices will be changed dynamically once `hidden_neurons` changes and so there is no need to perform any additional work once you change this parameter. The code was designed this way so minimal effort is needed on your part.

1.7.1 - Train the Neural Network to Solve the XOR Problem using 1 Hidden Neuron

Deliverables

- Set the `hidden_neurons` variable so that it is equal to 1. This means that we will only have 1 hidden neuron in the hidden layer.
- Run `part1.m` and provide the decision region plot that is generated by `plot_XOR_and_regions.m` as well as the plot showing the cost at each epoch in your report.
- What do you notice about the decision regions? Do these decision regions solve the XOR problem? What phenomenon is happening here?
- Comment on the cost curve that is generated with 1 hidden neuron. Does the cost curve accurately reflect what you are seeing in the decision region plot?

1.7.2. - Train the Neural Network to Solve the XOR Problem using 2 Hidden Neurons

Deliverables

- Set the `hidden_neurons` variable so that it is equal to 2. This means that we will have 2 hidden neurons in the hidden layer.
- Run `part1.m` and provide the decision region plot that is generated by `plot_XOR_and_regions.m` as well as the plot showing the cost at each epoch in your report.
- What do you notice about the decision regions? Do these decision regions solve the XOR problem?
- Comment on the cost curve that is generated with 2 hidden neurons. Does the cost curve accurately reflect what you are seeing in the decision region plot?

1.7.3 - Train the Neural Network to Solve the XOR Problem using 8 Hidden Neurons

- Set the `hidden_neurons` variable so that it is equal to 8. This means that we will have 8 hidden neurons in the hidden layer.
- Run `part1.m` and provide the decision region plot that is generated by `plot_XOR_and_regions.m` as well as the plot showing thes cost at each epoch in your report.
- What do you notice about the decision regions? Do these decision regions solve the XOR problem? What phenomenon is happening here?
- Comment on the cost curve that is generated with 8 hidden neurons. Does the cost curve accurately reflect what you are seeing in the decision region plot?

2 - Creating a Neural Network Prediction Model for Predicting Car Acceptability

2.1 - Introduction

In this section, you will implement the training of a neural network for multi-class prediction. Instead of using Stochastic Gradient Descent, you will use `fmincg` to compute the optimal weights so that these can be found more efficiently with the data that is to be used in this part of the lab assignment.

The data we will be using for this part was collated by Mark Botanic and Biaz Supan from the Jožef Stefan Institute, which is the largest research institute in Slovenia: http://www.ijs.si/. The data consists of 1728 examples and 6 features to identify the acceptability rating of a car. Specifically, whether the car is acceptable for driving. A lot of the data was categorical (i.e. using words to describe each class) and your instructor has post-processed this to convert this categorical data into numerical data for ease of algorithm implementation. This data is stored in the file `lab3cardata.mat` and is found in the `helper` directory of the lab assignment.

The possible acceptability ratings are: *Unacceptable, Acceptable, Good, Very Good*. The original 6 features in the dataset, as well as the possible labels per feature are:

1. Price: Low, Medium, High, Very High
2. Maintenance Price: Low, Medium, High, Very High
3. Number of Doors: 2, 3, 4, 5
4. Number of Seats: 2, 4, 6
5. Trunk Size: Small, Medium, Big
6. Safety Factor: Low, Medium, High

> *In the original data, the trunk of a car was called the Luggage Boot, which is a common way in Europe to describe the trunk: https://en.wikipedia.org/wiki/Trunk_(car)*

As the original data has categorical properties, the data is now transformed so it looks like the following per feature. The original label appears in parentheses while the final numerical label to be used appears to the left. If the feature is already numerical, no processing was done on it:

1. Price: 1 (Low), 2 (Medium), 3 (High), 4 (Very High)
2. Maintenance Price: 1 (Low), 2 (Medium), 3 (High), 4 (Very High)
3. Number of Doors: 2, 3, 4, 5
4. Number of Seats: 2, 4, 6
5. Trunk Size: 1 (Small), 2 (Medium), 3 (Big)
6. Safety Factor: 1 (Low), 2 (Medium), 3 (High)

> *In the original data, the actual labels for the number of seats was 2, 4, and **more than 4**, but it was transformed to 6 for the purposes of this lab assignment.*

The acceptability ratings are also transformed so that Unacceptable is 1, Acceptable is 2, Good is 3 and Very Good is 4.

This transformed data was then further split into training data so that you can use this to train your neural network and testing data so that you can test the accuracy of the neural network on unseen examples. The data was also randomly shuffled so that the training set consists of 80% of the data and the testing set consists of 20% of the data. These training and testing datasets are found in `lab3cardata.mat`. You can use `load lab3cardata.mat` in MATLAB to load in the data and what appears are four variables:

1. `Xtrain`: A `1382 x 6` matrix consisting of 1382 training examples and 6 features. Each row of the matrix is an example and each column of this matrix represents a feature and the order of the features respect the order seen previously. Therefore, the price is the first column of the matrix, the maintenance price is the second column of the matrix and so on.
2. `Ytrain`: A `1382 x 1` column vector that describes the acceptability label for each of the training examples. These labels range between 1 and 4, from Unacceptable to Very Good as mentioned previously.
3. `Xtest`: A `346 x 6` matrix consisting of 346 testing examples and 6 features. These features are (of course) the same as seen in the training set and is structured very much like `Xtrain`.
4. `Ytest`: A `346 x 1` column vector that describes the acceptability level for each of the test examples. This is structured like `Ytrain`.

2.2 - Develop a Multi-Class Neural Network Cost Function

2.2.1 - Introduction and Deliverables

As we will be using `fmincg` to help us train the neural network, we will need to develop a cost function for `fmincg` to minimize so that the optimal weights are obtained. This is the objective of this section, where you are to develop the corresponding cost function required to find the optimal weights to train a neural network that will perform multi-class classification. This function is to be implemented in the `costFunction_NN_reg.m` file found in the `tocomplete` directory of the lab assignment and it is your objective to complete it. This cost function will also implement regularization in order to combat the problem of overfitting.

An intricacy with `fmincg` is that the input (and ultimately the output) parameters / weights into the function are expected to be a single column vector. Because we are dealing with weight matrices, we will have to transform each weight matrix so that the rows are stacked together into a single column vector. These column vectors are stacked into one final column vector that is used as input into the cost function. Inside the cost function, you will need to sample the right positions of this column vector and **reshape** these sampled elements so that you can reconstruct the weight matrices. You would then use the weight matrices to implement the calculation of the cost required to assign the current input weights to the neural network as well as the gradient error updates. The gradient error updates are also expected to be in a column vector to be sent to the output, so one final intricacy is to transform the gradient error updates, which are matrices, to be packed into a single column vector in the same manner that we talked about before.

2.2.2 - Function Specifications

In light of what was discussed above, this function has the following specifications.

Inputs

- `X`: A data matrix of examples of size `m x n` where each row is an example and each column is a feature. There are `m` training examples and `n` features. Take note that you **should not** append a column of ones to this matrix as we have customarily done in the past. This behaviour is expected to be handled inside the function.

- `y`: A column vector of `m x 1` that denotes the expected class of each training example. The dynamic range of `y` is expected to lie in the range $[1, N]$ where N is the total number of possible classes. This is contrary to the range $[0, N-1]$ as we have seen in Lab #2. The reason why the range is shifted is to allow ease of implementation when considering the case of multi-class classification.
- `lambda`: The regularization parameter λ to prevent overfitting.
- `input_neurons`: The total number of input neurons in the input layer.
- `hidden_neurons`: The total number of hidden neurons in the hidden layer.
- `output_neurons`: The total number of output neurons in the output layer.
- `weights`: The vector of weights over all connections between nodes over all layers. This is ultimately a `((input_neurons + 1)*hidden_neurons + (hidden_neurons + 1)*output_neurons) x 1` vector

Outputs

- `cost_val`: The cost function evaluated at the given input set of parameters `weights`, given the input data `X` and output labels `y`.
- `grad`: The derivative (gradient) vector for each parameter / weight in our neural network. Concretely, this computes $\partial J \ / \ \partial w_{ij}^{(l)}$ for all weights in the neural network. The length of `grad` is the same as the length of `weights`.

Some More Explanations on the Inputs

We see some additional inputs that we normally haven't seen in a cost function. Specifically, we are providing the number of input layer, hidden layer and output layer neutons. The reason behind this ties in directly with how the input parameter `weights` is shaped. Because `fmincg` expects the cost function to have the input parameters to be shaped into a single column vector, manually specifying the number of neurons in each layer allows us to accurately sample from the column vector corresponding to the elements for the different weight matrices. We use these matrices to compute our gradient error updates, then use these same input parameters to pack the gradient error update vector, which is also expected to be a column vector in the right positions to be sent to the output.

You will also notice later in the lab that the value of the regularization parameter λ is going to be much less than what we've dealt with in the past with Logistic Regression. We are going to use rather small values but they will achieve the same results of preventing overfitting. This is because the cost function for Logistic Regression is quite different from the cost function used for Neural Networks. The dynamic range for the cost function between Logistic Regression and Neural Networks is not the same between them. Specifically, the Neural Network cost function has a much higher dynamic range and so the range of the regularization parameter has to be smaller in order to compensate.

2.2.3 - Code Listing

The function `costFunction_NN_reg.m` is partially completed in order for you to focus on the algorithm implementation. The file so far looks like the following:

```
function [cost_val, grad] = costFunction_NN_reg(X, y, lambda, input_neurons, ...
                                    hidden_neurons, output_neurons, weights)

%%% 1. Compute the total amount of weights per layer
total_weights_W1 = (input_neurons + 1)*hidden_neurons;
total_weights_W2 = (hidden_neurons + 1)*output_neurons;

%%% 2. Extract out the right portions of the weights vector and reshape
W1 = reshape(weights(1:total_weights_W1), hidden_neurons, input_neurons + 1).';
W2 = reshape(weights(total_weights_W1+1:end), output_neurons, hidden_neurons + 1).';

%%% 3. Get number of training examples
m = size(X, 1);

%%% 4. Initialize total cost and gradient update matrices
cost_val = 0;
W1_update = zeros(size(W1));
W2_update = zeros(size(W2));

%%% 5. Compute total cost and gradient update matrices
%%% PLACE YOUR CODE HERE

%%% 6. Take the updates and pack the output parameter vector
grad = zeros(numel(weights),1);
grad(1:total_weights_W1) = reshape(W1_update.', total_weights_W1, 1);
grad(total_weights_W1+1:end) = reshape(W2_update.', total_weights_W2, 1);
```

We will explain each step in detail. All of the steps have been completed for you, with the exception of Step #5 - the core part of the file which implements computing the total cost and the gradient updates per weight.

1. ***(Completed for you)*** The first step is to compute the total number of weights seen at each layer. Recalling from class, for a given layer l, there are $(d^{(l-1)} + 1) \times d^{(l)}$ total weights between layer $l-1$ and layer l. This is because for each node in layer $l-1$ including the bias, there are weights or connections to every node in layer l without the bias. For each node in layer $l-1$, we have $d^{(l)}$ connections, and thus the result follows. We calculate the total number of weights between the input layer and hidden layer, then again between the hidden layer and output layer. The purpose of this is to extract out the right elements in the parameter vector.
2. ***(Completed for you)*** The way the input parameter vector is going to be structured is that the first $n_1 = (d^{(0)} + 1) \times d^{(1)}$ elements belong to the weight matrix $W^{(1)}$ and the next $n_2 = (d^{(1)} + 1) \times d^{(2)}$ elements belong to the weight matrix $W^{(2)}$. The elements are packed in such a way where each row of a weight matrix is stacked together as a single vector and these elements are packed into input parameter vector in the right positions. We use the function `reshape` in MATLAB to transform a vector into a matrix. The elements are shuffled and rearranged so that a `(m x n)` row or column vector gets transformed into a `m x n` matrix. The input parameters into `reshape` are the vector to reshape, and the desired output dimensions of this newly shaped vector where the second input is the desired number of rows and the third input is the desired number of columns. However, MATLAB operates in **column-major** format. This means that when the final matrix gets created with `reshape`, the **columns** of the matrix get populated first.

   As a concrete example, supposing we had a vector from 1 to 12, and we wish to create a `3 x 4` matrix out of this using `reshape`, this is what happens when we use `reshape` in MATLAB:

```
>> reshape(1:12, 3, 4)

ans =

     1     4     7    10
     2     5     8    11
     3     6     9    12
```

As you can see, the **columns** of the matrix get populated first. However, the way we packed the parameter vector is that the elements should be populated **along the rows** not the columns. Therefore, what you have to end up doing is you have to use `reshape` but creating a matrix that is `n x m` instead of `m x n`, then **transposing** the result. This will have the effect of populating a matrix row-wise. As such, for the example we show above, if we used `reshape` and specified the total number of rows to be 4, the total number of columns to be 3 and then transposing the result, we get this instead:

```
>> reshape(1:12, 4, 3).'

ans =

     1     2     3     4
     5     6     7     8
     9    10    11    12
```

You can see now that the elements are populated row-wise. With this idea, the creation of the weight matrices from the input vector have been done for you. We extract the first n_1 elements in the parameter vector, then reshape the matrix so that it becomes an initial $d^{(1)} \times (d^{(0)} + 1)$ matrix to create $W^{(1)}$, and we transpose this result. We perform the same with the next n_2 elements to create $W^{(2)}$. The weight matrices $W^{(1)}$ and $W^{(2)}$ are both stored in the variables `W1` and `W2` respectively.

3. ***(Completed for you)*** We determine the total number of training examples and store this in a variable called `m`.

4. ***(Completed for you)*** This creates the initial weight matrices that will store our updates. These are created as `W1_update` and `W2_update` and you are to use these variables to calculate the updates required for each of the weights. Once the main algorithm is complete, these variables are to be used for sending the results to the output.

5. ***(To be completed)*** Your task for this file is to complete the algorithm for training a neural network in the viewpoint where you are to use MATLAB's optimization tools. Remember that **regularization** is also added into this algorithm. See the section on ***Training a Neural Network - Using Optimization Tools*** for more details. The relevant variables that you are going to need to use are:

    a. `m`: The total number of training examples
    b. `X`: The data matrix of training examples
    c. `y`: The expected output class for each input training example
    d. `lambda`: The regularization parameter
    e. `W1_update`: Where the gradient updates for the weights in $W^{(1)}$ are to be stored
    f. `W2_update`: Where the gradient updates for the weights in $W^{(2)}$ are to be stored
    g. `cost_val`: The average cost / error incurred when using the current set of input parameters for the neural network

> ***Hints:***
> ***a.*** *You're going to need at least one* `for` *loop to iterate through all training examples*
> ***b.*** *Each example in* `X` *is a row vector, yet the algorithm requires the input into the input layer to be a* ***column*** *vector. As such, it may be prudent to extract out the right example you need then transpose it so it becomes a column vector.*
> ***c.*** *You are going to need to save each* $\mathbf{s}^{(l)}$ *and* $\mathbf{x}^{(l)}$ *at every layer, and so using* `forward_propagation.m` *to save time is not applicable here. Don't use it in your code.*
> ***d.*** *Because there are a set number of layers (i.e. 1 input layer, 1 hidden layer, 1 output layer), the algorithm is simplified quite nicely. There's no need to write another* `for` *loop to go through each layer. Like what we saw with the XOR problem, you can simply compute each* $\mathbf{s}^{(l)}$, $\mathbf{x}^{(l)}$ *and* $\delta^{(l)}$ *individually and it'll take just a few lines of code.*
> ***e.*** *Once you complete the predicted output for a training example using forward propagation, you will need to compute the sensitivity at the output layer first. Because we are using a One-Vs-All scheme, you need to create the appropriate* y *vector where every element is 0 except for the position that corresponds to the right class that the training example belongs to. It may be helpful to use* `zeros` *to create a zero vector, then use the corresponding element in the variable* `y` *to set the right position of this new vector to 1.*
> ***f.*** *Remember that you will need to update the weight matrices* `W1_update` *and* `W2_update` *at each iteration.*
> ***g.*** *Part of the back propagation algorithm requires that you perform matrix-vector multiplication with the right weight matrices* ***without*** *the first row. To remove the first row from a matrix and leave the rest intact, you can use MATLAB indexing where you sample from the second row to the end. You can achieve this by doing* `Wtilde = W(2:end,:);` *where* `W` *is a weight matrix. and* `Wtilde` *is the resulting matrix without the first row.*
> ***h.*** *When computing the cost for each example to be accumulated into the cost variable, because we are dealing with vectors and we wish to find the length, the MATLAB function* `norm` *may be of use. If you're not comfortable with* `norm`*, a simple application of* `sum` *should help.*
> ***i.*** *When you finally add regularization in for the weight updates, only the second row to the last row has regularization applied to each weight matrix. Consider using what you see in hint* ***f.*** *so that you can just apply regularization to only these rows.*
> ***j.*** *When you finally add regularization in for the cost value, remember that you only sum over the second to the last row of each weight matrices. Use the hint* ***f.*** *as a source of inspiration. When it finally comes down to computing the final cost value, there are a few ways to do this efficiently. One approach is to take the modified matrices and convert them into vectors, then sum over the squares of each element in the vector individually.*

6. ***(Completed for you)*** The last step is to take the weight updates and to convert them into column vectors like in the manner we spoke of before. We first initialize a vector of all zeroes that matches in size with the input `weights` vector. This vector `grad` is what is to be sent to the output. After, for the first n_1 elements, we take the matrix `W1_update` and pack these elements into the first n_1 elements of `grad`. What's important here is that we need to **transpose** this matrix, then we use `reshape` so that we sample the rows of the matrix to be packed into a single column vector. We transpose the matrix first then unroll the matrix into a single vector because MATLAB operates in column-major. Therefore, in order to sample the rows of a matrix and pack them into a single vector, transposing transforms the rows into columns and so unpacking the matrix by its columns here is the same as unpacking the matrix by its rows in the original matrix. We do the same for the next n_2 elements, where we take the matrix `W2_update` and pack these elements into the

last part of `grad`. The final value `cost_val` and `grad` are sent to the output.

2.3 - Back to the Car Acceptability Prediction Problem

Deliverables

Now that we have successfully completed the cost function that `fmincg` will operate on, it's time to examine `part2.m`, which is the file used to complete the training of a neural network to perform multi-class classification to predict car acceptability. The code listing is shown below. Some of the parts have already been completed for you, and your objective is to complete the other parts. We will explain each part in detail.

```
%%% 1. Initial cleanup, add paths and load in data
%%% DON'T CHANGE
clearvars;
close all;
addpath('../data');
addpath('../helper');
load lab3cardata.mat;

%%% 2. Declare total number of input neurons, hidden layer neurons and output
%%% neurons

%%% DON'T CHANGE
input_neurons = 6;

%%% This we can change
hidden_neurons = 4;

%%% DON'T CHANGE
output_neurons = 4;

%%% 3. Compute the total weights between the input and hidden layer and
%%% the hidden layer and output layer.  Also compute the total amount
%%% of weights
%%% DON'T CHANGE
total_weights_W1 = (input_neurons + 1)*hidden_neurons;
total_weights_W2 = (hidden_neurons + 1)*output_neurons;
total_weights = total_weights_W1 + total_weights_W2;

%%% 4. Create the initial parameter vector of weights
%%% DON'T CHANGE
rng(123);
e_init_1 = sqrt(6) / sqrt(input_neurons + hidden_neurons);
e_init_2 = sqrt(6) / sqrt(hidden_neurons + output_neurons);
initial_vec = zeros(total_weights,1);
initial_vec(1:total_weights_W1) = 2*e_init_1*rand(total_weights_W1,1) - e_init_1;
initial_vec(total_weights_W1 + 1:end) = 2*e_init_2*rand(total_weights_W2,1) - e_init_2;

%%% 5. Set total number of iterations
%%% DON'T CHANGE
N = 400;

%%% 6. Regularization parameter - This you can change
lambda = 0;
```

```
%%% 7. Declare optimization settings
%%% PLACE YOUR CODE HERE

%%% 8. Find optimal weights
%%% PLACE YOUR CODE HERE
%%% MAKE SURE THE OUTPUT WEIGHT PARAMETER VECTOR IS STORED IN A VARIABLE CALLED weights

%%% 9. Extract out the final weight matrices
%%% DON'T CHANGE
W1 = reshape(weights(1:total_weights_W1), hidden_neurons, input_neurons + 1).';
W2 = reshape(weights(total_weights_W1+1:end), output_neurons, hidden_neurons + 1).';

%%% 10. Compute predictions for training and testing data
%%% PLACE YOUR CODE HERE

%%% 11. Compute classification accuracy for training and testing data
%%% PLACE YOUR CODE HERE
```

1. ***(Completed for you)*** This is standard procedure. Clear all of the current variables in the workspace, close all of the figures, then add the `data` and `helper` directories that are part of the lab to MATLAB's path so you don't have to copy and paste any of the files in these directories to the current working directory when completing this code.
2. ***(Completed for you)*** Like the XOR problem, we declare the total number of input, hidden and output layer neurons as variables. There are 6 features and so we have 6 input neurons. There are 4 expected classes, and so there are 4 output neurons. What can be varied is the total number of hidden layer neurons and this variable will be changed when performing experiments later in the lab.
3. ***(Completed for you)*** We compute the total number of weights between the input layer and hidden layer and the hidden layer and output layer. We've seen something like this in the `costFunction_NN_reg.m` file. We also compute the total number of weights overall so that we can create our initial parameter vector in the next step.
4. ***(Completed for you)*** This part performs the random initialization of weights like we have seen with the XOR problem. However, the creation of these weights is going to be slightly different. Instead of directly creating the weight matrices, we need to create an initial parameters vector that is to be used with `costFunction_NN_reg.m` where the first n_1 elements belong to $W^{(1)}$. A $n_1 \times 1$ vector of random values is created and scaled correctly. Similarly, the next n_2 elements belong to $W^{(2)}$ and a $n_2 \times 1$ vector of random values is created and are scaled correctly. This initial vector is what you would use as part of the input into `fmincg`.
5. ***(Completed for you)*** We set the total number of iterations that `fmincg` is going to take. We will **set this to 400 to be consistent**. Do **not** change this variable.
6. ***(Completed for you)*** The regularization parameter is set here. We will also vary this when performing experiments later in the lab.
7. ***(To be completed)*** You are to create an options structure using `optimset` that `fmincg` will use when finding the optimal weights. Consult Lab #2 on details on how to do that.
8. ***(To be completed)*** It is now your task to use `fmincg` so that you can find the optimal weights using `costFunction_NN_reg.m` that you completed earlier. Remember that you need to create an anonymous function handle to `costFunction_NN_reg.m` that consists of only one input - the weights themselves. Consult Lab #2 on details on how to do that. **You need to make sure that the output optimal parameters (stored as a vector) are stored in a variable called `weights`.**
9. ***(Completed for you)*** We use the optimal parameter vector and extract out the right portions of the vector to reshape these entries into their correct weight matrices. These are stored in variables `W1` and `W2`

respectively.

10. ***(To be completed)*** Once we finally extract out the weight matrices, you are to use `forward_propagation.m` and `predict_class.m` to help determine the predictions for both the training and test data.
11. ***(To be completed)*** Once you compute the predictions for both the training and test data, you are now required to compute the classification accuracy for both the training and test datasets.

> ***Note:*** *Due to the random initialization of the weights, even though the random seed is set, results still may differ from platform to platform. If you are comparing performance between other students, you may not get the same performance measures. However, as you will see with the next section, if you increase the total number of neurons, you should see a clear pattern in terms of performance.*

> ***Hint:*** *The amount of time it takes to train the neural network will vary from machine to machine. This is because we are iterating over each training example and computing an update as opposed to just compute a single update like in Linear or Logistic Regression. Because we are iterating over each training example, expect some time required to compute the optimal parameters. It is estimated to take between 30 seconds to 1 minute depending on the specifications of your machine. When your instructor implemented the training algorithm and when training the neural network was being performed, it took roughly 37 seconds on a MacBook Pro with a Intel 2.3 GHz i7 CPU with 16 GB of RAM.*

2.4 - Experiments with the Hidden Layer

Now that you have a working multi-class classification algorithm that predicts car acceptability, similar to the XOR problem, we will explore varying the number of hidden neurons in the hidden layer. If you examine the `part2.m` script that you completed, there is a variable called `hidden_neurons` that you can vary to change the number of hidden neurons in the input layer. Note that the weight matrices will be changed dynamically once `hidden_neurons` changes and so there is no need to perform any additional work once you change this parameter. The code was designed this way so minimal effort is needed on your part.

2.4.1 - Train the Neural Network to Solve the Car Acceptability Prediction Problem using 2 Hidden Neurons

Deliverables

- Set the `hidden_neurons` variable so that it is equal to 2. This means that we will only have 2 hidden neurons in the hidden layer.
- Run `part2.m` with **no regularization** (i.e. `lambda = 0`) and determine the classification accuracies for both the training and test data in your report. Also provide the final cost function value after convergence.
- Comment on the classification accuracies for the training and test data. Given the number of hidden

neurons, is this what you expect?
- Run `part2.m` again with 2 hidden neurons but set the regularization parameter `lambda = 0.25`. Determine the classification accuracies for both the training and test data in your report. Also provide the final cost function value after convergence.
- Comment on the classification accuracies for the training and test data. Given the number of hidden neurons and the regularization parameter, is this what you expect?

2.4.2 - Train the Neural Network to Solve the Car Acceptability Prediction Problem using 6 Hidden Neurons

Deliverables

- Set the `hidden_neurons` variable so that it is equal to 6. This means that we will have 6 hidden neurons in the hidden layer.
- Run `part2.m` with **no regularization** (i.e. `lambda = 0`) and determine the classification accuracies for both the training and test data in your report. Also provide the final cost function value after convergence.
- Comment on the classification accuracies for the training and test data. Given the number of hidden neurons, is this what you expect?
- Run `part2.m` again with 6 hidden neurons but set the regularization parameter `lambda = 0.25`. Determine the classification accuracies for both the training and test data in your report. Also provide the final cost function value after convergence.
- Comment on the classification accuracies for the training and test data. Given the number of hidden neurons and the regularization parameter, is this what you expect?

2.4.3 - Train the Neural Network to Solve the Car Acceptability Prediction Problem using 10 Hidden Neurons

Deliverables

- Set the `hidden_neurons` variable so that it is equal to 10. This means that we will have 10 hidden neurons in the hidden layer.
- Run `part2.m` with **no regularization** (i.e. `lambda = 0`) and determine the classification accuracies for both the training and test data in your report. Also provide the final cost function value after convergence.
- Comment on the classification accuracies for the training and test data. Given the number of input neurons, is this what you expect?
- Run `part2.m` again with 10 hidden neurons but set the regularization parameter `lambda = 0.25`. Determine the classification accuracies for both the training and test data in your report. Also provide the final cost function value after convergence.
- Comment on the classification accuracies for the training and test data. Given the number of hidden neurons and the regularization parameter, is this what you expect?

Second Half - SVMs

Now that we have examined how to perform both binary and multi-class classification using Neural Networks, we will investigate how to perform the same tasks using Support Vector Machines - another machine learning paradigm that is competing in performance compared to Neural Networks. Specifically, we will focus on using the SVM functionality that is part of MATLAB's Statistics and Machine Learning Toolbox to allow us to train SVMs and

create objects that we can use to predict the classes for new examples. We will examine both the binary classification and multi-class classification case in this part of the lab.

Warmup

> *This section is strictly for your benefit. There is nothing to write up about in your report here. You don't have to read this section but it's highly encouraged that you do so you can implement the lab assignment with ease.*

In class, we covered a quick demo on how to use SVMs for both binary and multi-class classification.

Binary Classification - SVMs

Taking from what we covered in class, the code listing for the binary classification demo using SVMs is shown below. We will explain each part in detail:

```
%%% 1. Load in the data
close all;
clearvars;
load fisheriris;

%%% 2. Create the training example data and expected labels
% This is a 150 training example dataset with 4 features
% stored in meas - 150 x 4
% The labels are stored as strings in a cell array called species
% Get the first 100 examples - 2 classes
% Look at two features for now - The last two
X = meas(1:100,3:4);

% Make the first class positive and the second class negative
y = [ones(50,1); zeros(50,1)];

%%% 3. Show what the data looks like
figure;

% Plots a scatter plot where each training example is highlighted in a
% different colour depending on the class
plot(X(1:50,1), X(1:50,2), 'b.', X(51:100,1), X(51:100,2), 'r.',...
    'MarkerSize', 16);

%%% 4. Train a linear SVM classifier
svm = fitcsvm(X, y, 'ClassNames', [0 1]);

%%% 5. Plot the support vectors as black circles on top of the data
% Get the support vectors
% This is a matrix of points that tells you the training examples that were
% selected as support vectors
sv = svm.SupportVectors;
hold on;
plot(sv(:,1), sv(:,2), 'ko', 'MarkerSize', 10);
legend('Positive Class', 'Negative Class', 'Support Vectors');
```

```
%%% 6. Declare new examples for predicting, then use SVM model to predict
%%% new instances
newX = [1.7 2; 4.5 1.2];
labels = predict(svm, newX);
```

1. The first part is to clear all variables, close all figures and load in the Fisher Iris data. You may recognize this data from the Bayesian Decision Theory portion of Lab #2. However, this is actually built-in to MATLAB so there's no need to re-import the data from Lab #2 for the purposes of this explanation.
2. The next part extracts out 2 features from the dataset and separates the data into 2 classes. There are 150 training examples in this class with 4 features stored as a `150 x 4` matrix and is stored in a variable called `meas` when you load in the Fisher Iris dataset. This means that each row is a training example and each column is a feature. The first 50 examples belong to one class while the next 50 examples belong to another class. Therefore, we extract out the first 100 rows of this data matrix and we concentrate on the third and fourth features of the matrix as our two features of interest. This final matrix gets stored in `X`. We also create a label vector `y` that has 100 elements where the first 50 elements denote the positive class and the last 50 elements denote the negative class. These are the labels 1 and 0 respectively.
3. We spawn a new figure and plot what these features look like as well as colouring them with respect to their corresponding labels. The positive class is blue while the negative class is red. The positive class is the first 50 training examples and the negative class is the last 50. We specifically make the size of each 16 pixels large. The figure looks like so:

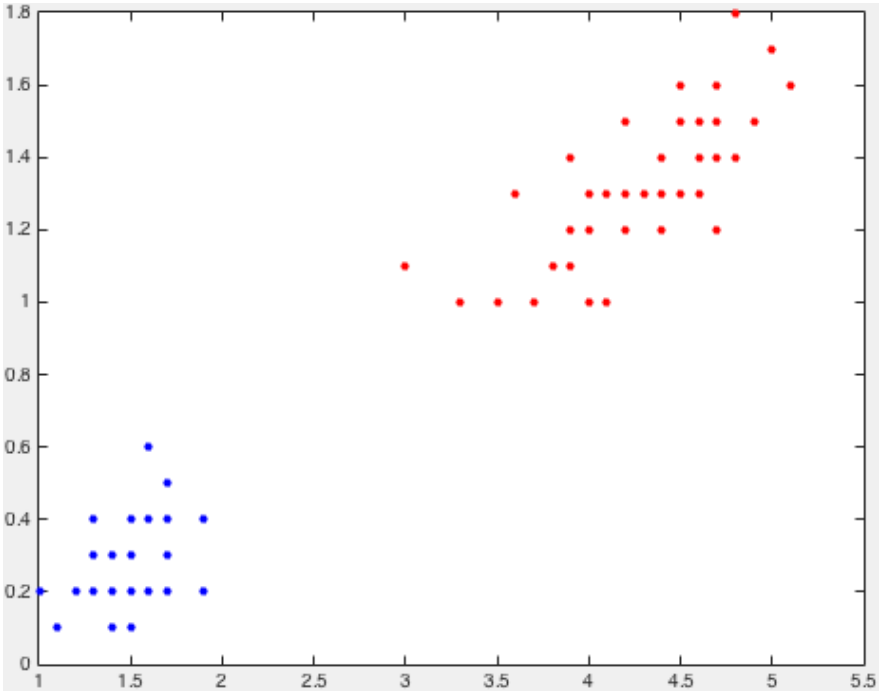

Figure 5 - The training examples to be used in the binary classification demo

We can clearly see that this data is linearly separable.

4. The next part is to train a SVM classifier using the function `fitcsvm`. We specify the training examples and expected labels per example as the first two elements of `fitcsvm`. Take note that the default kernel function is **linear**, implying that the data is expected to be linearly separable. It is also good practice to manually specify which labels in the expected labels vector correspond to the negative and positive class. As such, we specify the flag `ClassNames` and a two element vector where the first position tells `fitcsvm` which labels correspond to negative and the second position denotes which labels correspond to positive. We denoted a negative example to be 0 and a positive example to be 1 from Step #2, and so we set this

input vector to be `[0 1]`. What is output is a SVM object that we can use to perform predictions.

5. On top of the plot that was already created in Step #3, we also plot what training examples were used as the support vectors and we draw these as black circles on top of these training examples. We first access the actual support vectors from the SVM object and it is a matrix of training examples that tell us the actual training examples chosen to be support vectors The first column is the first feature and the second column is the second feature. We finally plot these support vectors and put in a legend in the plot to make things neat. We also add in a legend that denotes what are positive examples, negative examples and the corresponding support vectors. The final graph looks like this:

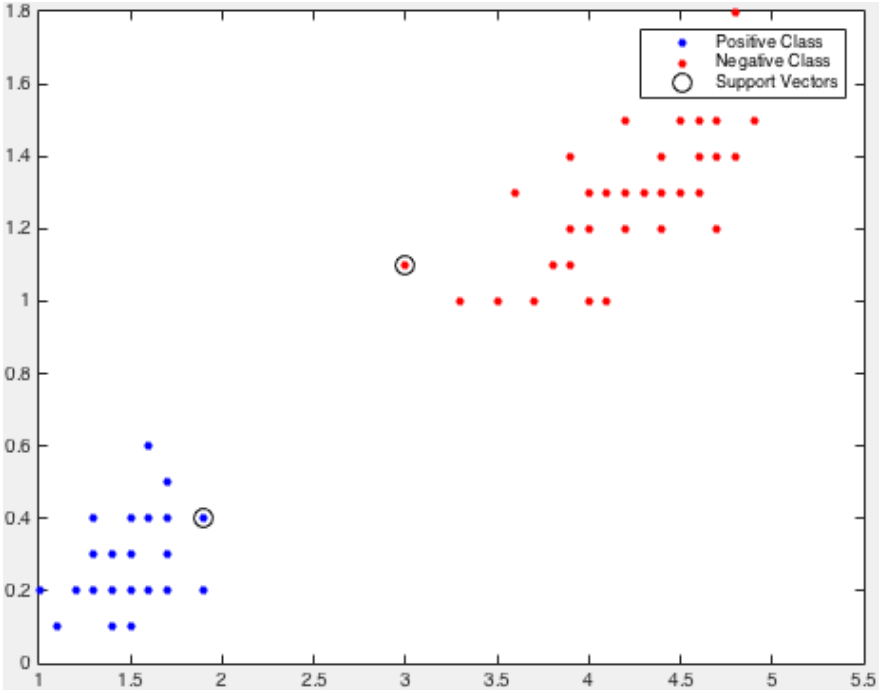

Figure 6 - The training examples to be used in the binary classificationth demo with the support vectors plotted

6. The last part is to create new examples in a data matrix and to predict their classes. Each row is a new example and each column is a feature. In this case, we want to predict the classes for examples $(x_1, x_2) = (1.7, 2)^T$ and $(x_1, x_2) = (4.5, 1.2)^T$. These examples are placed in a matrix stored in the variable `newX` and we use the `predict` function where the first input is the trained SVM object we created from `fitcsvm` and the second input is the matrix of new instances we want to predict the classes of. The output will be a vector that has its length equal to the total number of new examples that tells you what class each input belongs to. Because the `ClassNames` input was set to `[0 1]` we should see either 0 or 1 for each example.

Multi-Class Classification - SVMs

Taking from what we learned in class, the code listing for the multi-class classification demo using SVMs is shown below. We will explain each part in detail:

```
%%% 1. Load in the data
close all;
clearvars;
load fisheriris;

%%% 2. Get only the last two features, but we want all of the data
X = meas(1:150,3:4);

%%% 3. Show data first
figure;
plot(X(1:50,1), X(1:50,2), 'b.', X(51:100,1), X(51:100,2), 'r.',...
    X(101:150,1), X(101:150,2), 'g.', 'MarkerSize', 16);

%%% 4. Train SVMs
svms = cell(1,3); % Stores SVM model for detecting each class
% Cell arrays are flexible arrays where each element can store ANYTHING

% Train first SVM
y1 = false(150,1);
y1(1:50) = true;
svms{1} = fitcsvm(X, y1, 'ClassNames', [false true]);

% Train second SVM
y2 = false(150,1);
y2(51:100) = true;
svms{2} = fitcsvm(X, y2, 'ClassNames', [false true]);

% Train third SVM
y3 = false(150,1);
y3(101:150) = true;
svms{3} = fitcsvm(X, y3, 'ClassNames', [false true]);

%%% 5. Use one-vs-all to predict new instances
newX = [1.7 2; 4.5 1.2; 6 2];

% For each SVM, calculate the matching score for each set of new examples
scores = zeros(size(newX,1), 3);

for ii = 1 : 3
    % This version of predict has a second output argument where
    % it provides a two column matrix of matching scores
    % The first column is the score for the negative class
    % The second column is the score for the positive class
    [~,score] = predict(svms{ii}, newX);

    % Place positive class scores in the right column
    scores(:,ii) = score(:,2);
end

%%% 6. Find the labels
[~,labels] = max(scores,[],2);
```

1. The first part is to clear all variables, close all figures and load in the Fisher Iris data. You may recognize this data from the Bayesian Decision Theory portion of Lab #2. However, this is actually built-in to MATLAB so there's no need to re-import the data from Lab #2 for the purposes of this explanation.

2. The next part extracts out 2 features from the dataset and separates the data into 2 classes. There are 150 training examples in this class with 4 features stored as a `150 x 4` matrix and is stored in a variable called `meas` when you load in the Fisher Iris dataset. This means that each row is a training example and each column is a feature. The first 50 examples belong to one class while the next 50 examples belong to another class and finally the last 50 examples belong to the last class. There are 3 classes in total for this data set. Therefore, we will want to perform multi-class classification and so we need all of the training examples. However, we will still concentrate on just two features which are the third and fourth columns of the data set. This final matrix gets stored in `X`.
3. We show what the data looks like. The first class is marked in blue circles, the second class is marked in red circles and finally the last class is marked in green circles. We set the pixel size of each marker to be 16 just like before. The spawned figure is shown below:

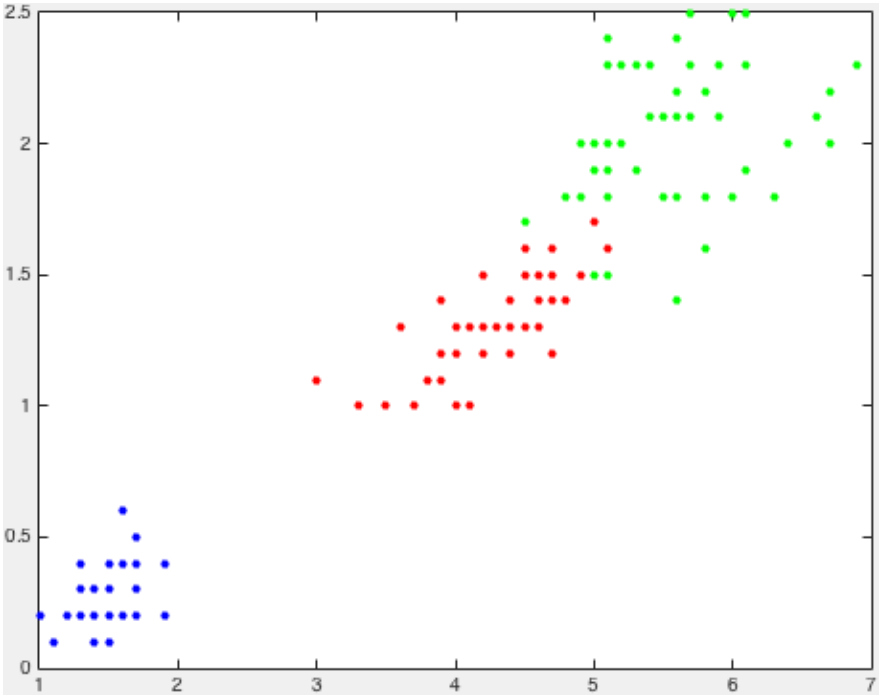

Figure 7 - The training examples to be used in the multi-class demo

The data is more or less linearly separable. The classes dictated by the green and red dots have some ambiguities, but we will still use a linear kernel function regardless.

4. We now train 3 SVMs that are designed to detect each class. Take note that the default kernel function is **linear**, implying that the data is linearly separable. Just like the standard One-Vs-All approach, we create label vectors where the positive label is for the class in consideration while the negative label is for the other classes. Instead of using `0` and `1` like in the binary classification case, we will use `false` and `true` for the `ClassNames` attribute so we can show you how versatile `fitcsvm` is when determining which labels are negative and positive classes. The labels don't have to be integer. They can be Boolean, strings, doubles, etc. as long as you provide a two element vector that appropriately distinguishes between the positive and negative classes for the `ClassNames` attribute. Remember that the first 50 training examples are the first class, the next 50 are the second class and the last 50 are the third class. We also use a **cell array** to store the SVM objects instead of individual SVM objects stored as variable names so that it will allow predictions to be performed in an easier manner.
5. We now create new examples in a data matrix to predict their classes. Each row is a new example and each column is a feature. In this case, we want to predict the classes for examples $(x_1, x_2) = (1.7, 2)^T$, $(x_1, x_2) = (4.5, 1.2)^T$ and $(x_1, x_2) = (6, 2)^T$. These examples are placed in a matrix stored in the variable `newX`. Next what we do is we cycle through each SVM object and determine the **score matrix** for the training examples in `newX` for each SVM. We create a `scores` variable that will contain the score matrix

that we discussed previously. The number of rows should equal the number of examples in the new dataset while the total number of columns equals the total number of expected classes.

There is another version of `predict` which produces two outputs. The first output is the predicted label, which isn't applicable to us, but the second output is what we are interested in - the score matrix. The score matrix is structured such that the total number of rows is equal to the total number of examples that were input into `predict` and the total number of columns is the total number of classes that were used when training the SVM. In our case, this is only two columns. The score matrix contains the score of how likely each training example belongs to the particular class. The larger the score, the better chance the example belongs to that class. The first column is for the negative class and the second column is for the positive class. As such, because we are considering a One-Vs-All scheme, we cycle through each SVM, calculate the score matrices and extract out the **second column** of the score matrix and place this column into the right column in an overall score matrix which we have created called `scores`.

6. Once the `scores` variable has been populated, note that each training example is stored in a row and each element of a row tells you the score of how likely that training example belongs to a particular class. As such, we use `max` to determine what the final class would be per training example.

3 - XOR Problem Revisited - Using SVMs

3.1 - Introduction

We will repeat solving the XOR problem but we will use SVMs to perform this for us instead. What is recommended is that you use the binary classification demo code for SVMs as a basis for solving this problem.

One aspect of the SVM toolchain in MATLAB that we haven't explored is using different kernel functions to transform the potential non-linear relationships of data into linearly separable data for finding the largest margin of separation between the two classes. As seen in the demo code, the kernel function assumed is linear, but in general the data may be non-linearly separable.

When using `fitcsvm`, you may have noticed that we used `'ClassNames'` as an additional flag and you provide the right input that is specific for this flag. `fitcsvm` in general can take in pairs of flags and corresponding inputs. Concretely, we can create a SVM object that conforms to the following MATLAB code:

```
svm = fitcsvm(X, y, 'param1', value1, 'param2', value2, ..., 'paramN', valueN);
```

In addition to the training examples and expected outputs, you can specify a flag as a string and the input that matches that flag. `'param1'`, `'param2'`, ..., `'paramN'` are the different flags you can use as strings and `value1`, `value2`, ... `valueN` are the inputs that are expected for each corresponding flag. You have seen `'ClassNames'` already used. Some other useful input flags are:

1. `'KernelFunction'` - We haven't specified this flag so far but if you do, this specifically tells MATLAB which kernel function you'd like to use to transform your data. The available options are: `'gaussian'`, `'linear'` (the default) and `'polynomial'`. If you don't specify the `'KernelFunction'` flag (as what we have seen previously), the linear kernel function will be used. For the Gaussian kernel, there is no standard deviation associated with the kernel and so this is a parameter that can't be varied. Concretely, the kernel is defined as $\phi(x^{(i)}, x^{(j)}) = \exp\left(-\|x^{(i)} - x^{(j)}\|^2\right)$.
2. `'PolynomialOrder'` - If you are using a polynomial kernel, you can specify what order you want to use. Take note that the polynomial kernel in MATLAB is defined as $\phi(x^{(i)}, x^{(j)}) = (x^{(i)} \cdot x^{(j)})^d$ where d is the order. There is no offset like we have seen in class and so that offset is equal to 0. The default order is 3 and so omitting `PolynomialOrder` assumes the order is 3.
3. `'Standardize'` - You can choose this option to normalize your features so that each feature has zero-

mean and unit variance. The default is `false`, but you can specify `true` to ensure the features are normalized.
4. `'ClassNames'` - We've seen this already in action. You provide a two element vector where the first element is the label in `y` for the negative class and the second label is the label in `y` for the positive class.

3.2 - Solving the XOR Problem with SVMs

3.2.1 - Complete `part3.m`

Deliverables

You are to complete solving the XOR problem using SVMs in the file `part3.m`. Some of the code has been filled out for you but the rest of it you will need to complete. The code listing for this is shown below:

```
%%% 1. Clear variables and close all figures
clearvars;
close all;

%%% 2. Define input training examples
X = [0 1; 1 1; 1 0; 0 0];
y = [1;0;1;0];

%%% 3. Train a non-linear SVM classifier
%%% PLACE YOUR CODE HERE

%%% 4. Compute decision regions
%%% PLACE YOUR CODE HERE
%%% Hint: Use plot_XOR_and_regions as inspiration
```

Part 1 and Part 2 you are quite familiar with so we won't explain what those are doing. Part 3 and 4 are to be completed by you. To start off with things, use the default `'linear'` kernel function when creating the SVM object for Part 3. For Part 4, you are to recreate the decision region plot like you have seen in the past when using `plot_XOR_and_regions.m`

> ***Hint:*** *You can use the majority of the code in* `plot_XOR_and_regions.m` *and leave it untouched. The* ***only*** *thing that needs to be changed is the use of* `forward_propagation.m`*. Replace this with* `predict` *instead. There's also no reason to threshold the output because the* `predict` *function will give you the actual class labels themselves.*

3.2.2 - Using the Linear Kernel Function

Deliverables

- Now that you've completed `part3.m`, let's try the default kernel function - the linear function. Run `part3.m` with the linear kernel function and plot the decision regions. Comment on the performance that the linear function has on the XOR problem. Is it what you expect? What assumption does the linear kernel function make when performing classification?

3.2.3 - Using the Gaussian Kernel Function

Deliverables

- Change the kernel function so that you are using the Gaussian kernel instead. Run `part3.m` with the Gaussian kernel function and plot the decision regions. Comment on the performance that the linear kernel function has on the XOR problem. Is it what you expect?

3.2.4 - Using the Polynomial Kernel Function

Deliverables

- Change the kernel function so that you are using the polynomial kernel instead. Use the default polynomial order of 3. Run `part3.m` with the polynomial kernel function and plot the decision regions. Comment on the performance that the linear function has on the XOR problem. Is it what you expect?
- Out of the linear, Gaussian and polynomial kernel, which one would you choose for the best performance and why?

4 - Car Acceptability Prediction Problem Revisited - SVMs

We will repeat solving the car acceptability prediction problem but we will use SVMs to perform this for us instead. What is recommended is that you use the multi-class classification demo code for SVMs as a basis for solving this problem.

4.1 - Solving the Car Acceptability Prediction Problem using SVMs

4.1.1 - Complete `part4.m`

Deliverables

You are to complete solving the car acceptability prediction problem using SVMs in the file `part4.m`. Some of the code has been filled out for you but the rest of it you will need to complete. The code listing for this is shown below:

```
%%% 1. Initial cleanup, add paths and load in data
%%% DON'T CHANGE
clearvars;
close all;
addpath('../data');
addpath('../helper');
load lab3cardata.mat;

%%% 2. Create helpful variables
mTrain = size(Xtrain, 1); % Total number of training examples
mTest = size(Xtest, 1); % Total number of test examples
n = 4; % Total number of classes
```

```
%%% 3. Train non-linear SVM classifiers - one vs all using the training data
% Create cell arrays to store each SVM classifier
% Use the Gaussian kernel function
%%% PLACE YOUR CODE HERE

%%% 4. Perform One-Vs-All prediction on the training and test dataset
% Determine which class each of the examples in the test datasets are
% Create score matrices for both the training and test datasets
%%% PLACE YOUR CODE HERE

%%% 5. Calculate the classification accuracy for the training and test datasets
%%% PLACE YOUR CODE HERE
```

Step 1 you are familiar with. Step 2 creates helpful variables to be used during the SVM training and prediction process. `mTrain` and `mTest` contain the number of training and testing examples and `n` contains the total number of possible classes - 4 in our case. Step 3 requires that you create non-linear SVM classifiers using the Gaussian kernel functions and there will be 4 of them in total. Step 4 requires that you create score matrices for both the training and test datasets and you are to use these to predict which class each example belongs to for both the training and test datasets. Step 5 finally asks you to calculate the classification accuracy for both the training and test datasets.

> ***Hints**:*
> ***a.** For step 3, it may be prudent to use a `for` loop to iterate over each class and to finally train a SVM suited for detecting that class. The `y` vector to be used in `fitcsvm` can easily be found by creating a suitable logical vector. For example, doing `y == 1` would create a logical vector where the elements are either `true` or `false` where `true` means that a position in `y` is equal to 1 and `false` is when it isn't equal to 1.*
> ***b.** For step 4, using `predict_class.m` may be of use here as the score matrix format is exactly the same as what was seen in Neural Networks.*

Questions to Ask

1. Once you have completed `part4.m`, run it and report the classification accuracy for the training and test datasets.
2. Is there a significant difference in accuracy between these figures and the ones using Neural Networks? Why do you believe there is a difference?

Discussion

These are open ended discussion questions designed to assess whether you have learned all you needed to learn in this lab. There are no right or wrong answers. Answer whatever comes off the top of your head and which makes sense!

1. Now that you have seen both neural networks and SVMs in practice to solve a real world problem, which method would you prefer and why?
2. We have seen in class some heuristics on how many hidden neurons to choose when considering the hidden layer. As mentioned in class, there is no universal measure on determining the total number of

neurons in the hidden layer. Do you believe that there is an automated way to choose the number of hidden neurons? If so, what procedure would you perform to determine this number? If not, what reasons do you have for this?

3. You've also seen the effects of increasing the number of neurons in the hidden layer and varying the regularization parameter. Which do you believe is the better practice - starting off with many hidden layer neurons and using regularization to compensate or using a smaller number of neurons and avoiding regularization? Why do you believe that the method you chose is better?
4. Do you believe that adding more than one hidden layer would increase performance and classification accuracy? Why or why not?
5. What we have not done in this lab is normalize the input data so that there is zero-mean and unit variance for each feature. Do you believe that there is a benefit to normalizing our data before we train our neural network? Why or why not?
6. As we have seen in SVMs, there are multiple kernel functions we can consider. We ran multiple experiments on different kernel functions in the XOR problem. How would you ultimately decide what kernel function you would use in a general problem?
7. One thing we did not do in this lab is use the `'Standardize'` flag for training our SVM so that our data is normalized. Do you believe that there is a merit to normalizing our data when training a SVM? Why or why not?