Machine Learning II – Lab Excercise

Lab Excercise Manual

Autor:

Version: 1.0

Datum: 05.01.2022

Table of Content

Abbildungsverzeichnis

1 MLII1 – Principal Components Analysis

The objective of this exercise is to get a profound understanding of principal component analysis (PCA). For this reason, you will implement PCA. You should be able to assess the capabilities of PCA in its main application domains data compression and data visualization. You will use principal component analysis to find a low-dimensional representation of face images and visualize the color distribution in an RGB image.

For this exercise you will need the following files:

- `ex7 pca.m` - Octave/Matlab script for running the exercise on PCA
- `ex7data1.mat` - Example Dataset for PCA
- `ex7faces.mat` - Faces Dataset
- `kMeansResultsOnAceros_small.mat` – bird image from exercise MLI6 and results from kMeans-Clustering
- `displayData.m` - Displays 2D data stored in a matrix
- `drawLine.m` - Draws a line over an exsiting figure
- `featureNormalize.m` – Normalizes feature vectors
- `pca.m`*) - Perform principal component analysis
- `projectData.m`*) - Projects a data set into a lower dimensional space
- `recoverData.m`*) - Recovers the original data from the projection

You will have to modify the files marked with *).

Throughout this exercise, you will be using the script `ex7_pca.m`. This script sets up the dataset for the problems and makes calls to functions that you will write. You are only required to modify functions in other files, by following the instructions in this document.

1.1 Example dataset

In this exercise, you will use principal component analysis (PCA) to perform dimensionality reduction. You will first experiment with an example 2D dataset to get an intuition on how PCA works, and then use it on a bigger dataset of 5000 face images.

To help you understand how PCA works, you will first start with a 2D dataset, which has one direction of large variation and one of smaller variation. The script `ex7_pca.m` will plot the training data (Figure 1). In this part of the exercise, you will visualize what happens when you use PCA to reduce the data from 2D to 1D. In practice, you might want to reduce data from 256 to 50 dimensions, say; but using lower dimensional data in this example allows us to visualize the algorithm much better.

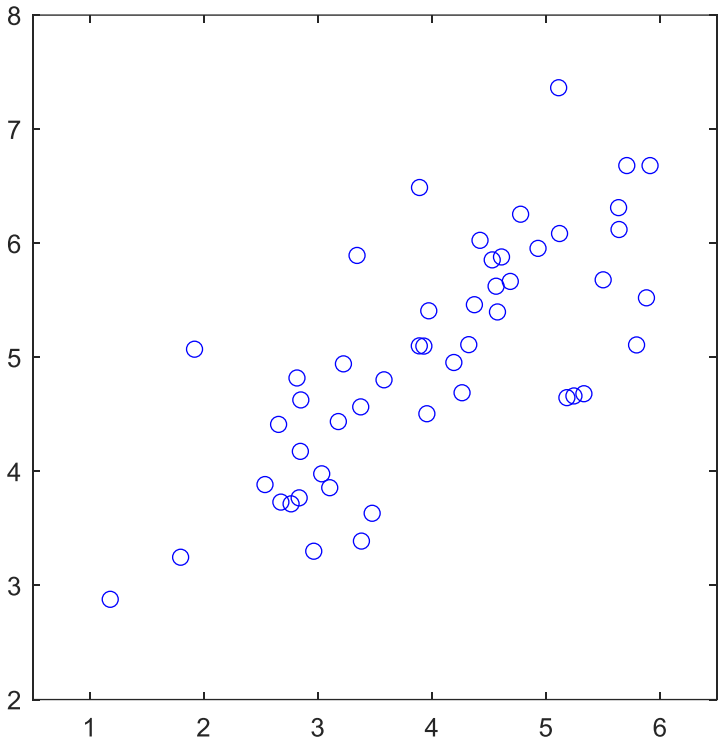

Figure 1: Example Dataset 1

1.2 Implementing PCA

Now, you will implement PCA. PCA consists of two computational steps: First, you compute the covariance matrix of the data. Then, you use Octave/Matlab's SVD function to compute the eigenvectors $\vec{u}^{(1)}, \vec{u}^{(2)}, \ldots, \vec{u}^{(n)}$. These will correspond to the principal components of variation in the data.

Before using PCA, it is important to first normalize the data by subtracting the mean value of each feature from the dataset and scaling each dimension so that they are in the same range. In the provided script `ex7_pca.m`, this normalization has been performed for you using the `featureNormalize` function.

After normalizing the data, you can run PCA to compute the principal components. Your task is to complete the code in `pca.m` to compute the principal components of the dataset. First, you should compute the covariance matrix of the data, which is given by:

$$\Sigma = \frac{1}{m} \hat{X}^T X$$

where $\hat{X}$ is the data matrix with examples in rows and m is the number of examples. Note that Σ is a $n \times n$ matrix and not the summation operator. After computing the covariance matrix, you can run SVD on it to compute the principal components. In Octave/Matlab, you can run SVD with the following command: `[U, S, V] = svd(Sigma)`, where `U` will contain the principal components and `S` will contain a diagonal matrix.

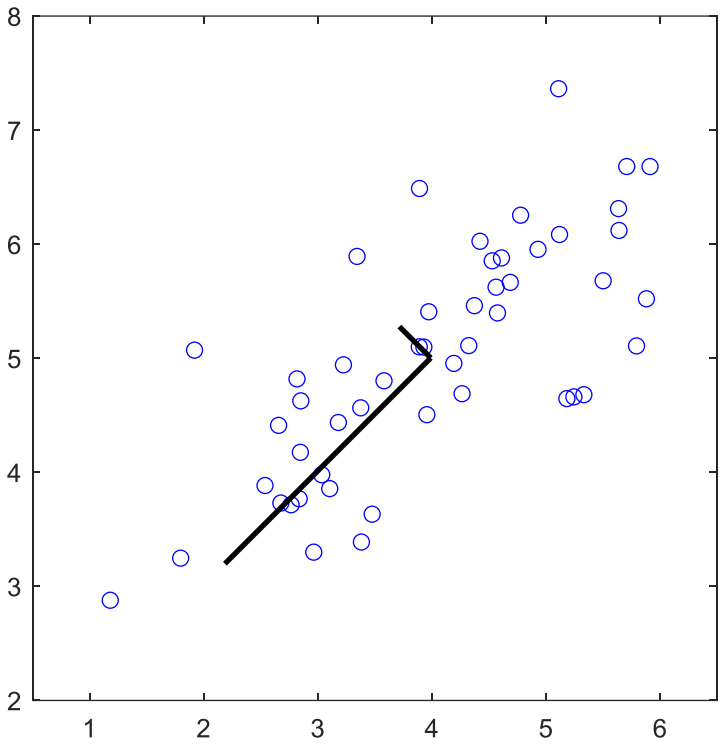

Figure 2: Computed eigenvectors of the Dataset 1

Once you have completed `pca.m`, the `ex7_pca.m` script will run PCA on the example dataset and plot the corresponding eigenvectors of the principal components found (Figure 2). The script will also output the eigenvector of the top principal component. For this you should expect to see an output of about `[-0.707 -0.707]`. (It is possible that Octave/Matlab may instead output the negative of this, since $\vec{u}^{(1)}$ and $-\vec{u}^{(1)}$ are equally valid choices for the first principal component.)

Show your results to an instructor.

1.3 Dimensionality Reduction with PCA

After computing the principal components, you can use them to reduce the feature dimension of your dataset by projecting each example onto a lower dimensional space, $\vec{x}^{(i)} \longrightarrow \vec{z}^{(i)}$ (e.g., projecting the data from 2D to 1D).

In this part of the exercise, you will use the eigenvectors returned by PCA and project the example dataset into a 1-dimensional space. In practice, if you were using a learning algorithm such as linear regression or perhaps neural networks, you could now use the projected data instead of the original data. By using the projected data, you can train your model faster as there are less dimensions in the input.

1.3.1 Projecting the data onto the principal components

You should now complete the code in `projectData.m`. Specifically, you are given a dataset `X`, the principal components `U`, and the desired number of dimensions to reduce to `K`. You should project each example in `X` onto the top `K` components in `U`. Note that the top `K` components in `U` are given by the first `K` columns of `U`, that is `U_reduce = U(:, 1:K)`. Once you have completed the code in `projectData.m`, `ex7_pca.m` will project the first example onto the first dimension and you should see a value of about `1.481` (or possibly `-1.481`, if you got $-\vec{u}^{(1)}$ instead of $\vec{u}^{(1)}$).

Show your results to an instructor.

1.3.2 Reconstructing an approximation of the data

After projecting the data onto the lower dimensional space, you can approximately recover the data by projecting them back onto the original high dimensional space. Your task is to complete `recoverData.m` to project each example in `Z` back onto the original space and return the recovered approximation in `X_rec`.

Once you have completed the code in `projectData.m`, `ex7_pca.m` will recover an approximation of the first example and you should see a value of about `[-1.047 -1.047]`.

Show your results to an instructor.

1.3.3 Visualizing the projections

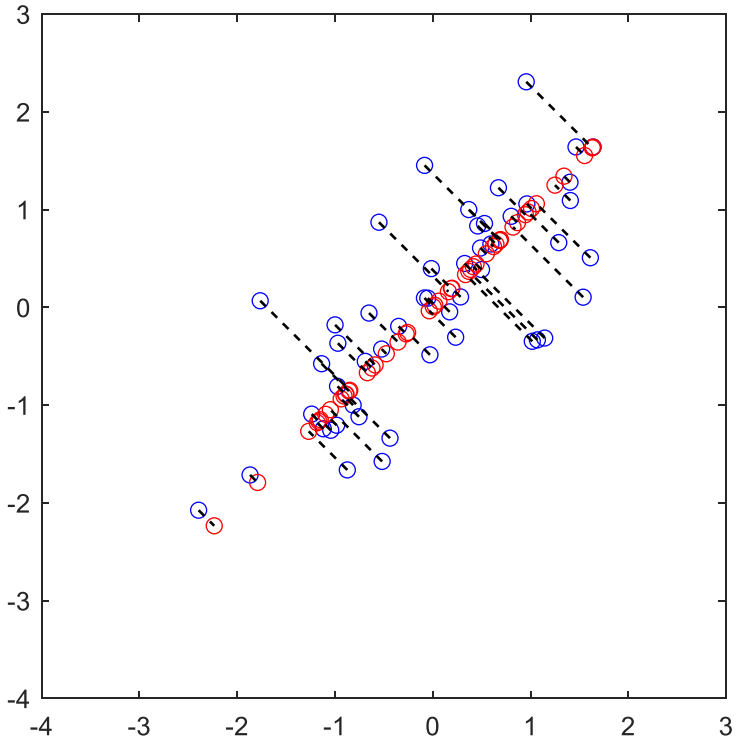

Figure 3: The normalized and projected data

After completing both `projectData` and `recoverData`, `ex7_pca.m` will now perform both the projection and approximate reconstruction to show how the projection affects the data. In Figure 3, the original data points are indicated with the blue circles, while the projected data points are indicated with the red circles. The projection effectively only retains the information in the direction given by $\vec{u}^{(1)}$.

1.4 Face Image Dataset

In this part of the exercise, you will run PCA on face images to see how it can be used in practice for dimension reduction. The dataset `ex7faces.mat` contains a dataset `X` of face images, each 32 × 32 in grayscale. Each row of `X` corresponds to one face image (a row vector of length 1024). The next step in `ex7_pca.m` will load and visualize the first 100 of these face images (Figure 4).

Figure 4: Faces dataset

1.4.1 PCA on Faces

To run PCA on the face dataset, we first normalize the dataset by subtracting the mean of each feature from the data matrix `X`. The script `ex7_pca.m` will do this for you and then run your PCA code. After running PCA, you will obtain the principal components of the dataset. Notice that each principal component in `U` (each row) is a vector of length n (where for the face dataset, $n = 1024$). It turns out that we can visualize these principal components by reshaping each of them into a 32 × 32 matrix that corresponds to the pixels in the original dataset. The script `ex7_pca.m` displays the first 36 principal components that describe the largest variations (Figure 5). If you want, you can also change the code to display more principal components to see how they capture more and more details.

1.4.2 Dimensionality Reduction

Now that you have computed the principal components for the face dataset, you can use it to reduce the dimension of the face dataset. This allows you to use your learning algorithm with a smaller input size (e.g., 100 dimensions) instead of the original 1024 dimensions. This can help speed up your learning algorithm.

Figure 5: Principal components of the face dataset

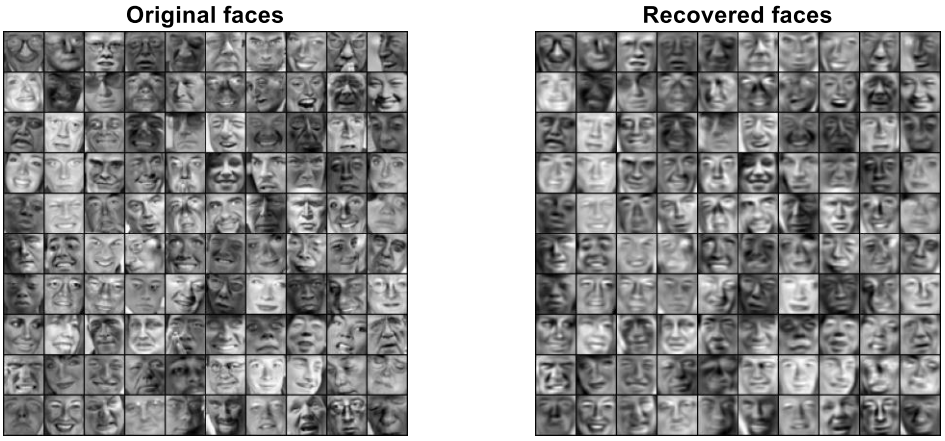

Figure 6: Original images of the faces and ones reconstructed from the first top 100 principal components

The next part in `ex7_pca.m` will project the face dataset onto only the first 100 principal components. Concretely, each face image is now described by a vector $\vec{z}^{(i)} \in \mathbb{R}^{100}$. To understand what is lost in the dimension reduction, you can recover the data using only the projected dataset. In `ex7_pca.m`, an approximate recovery of the data is performed and the original and projected face images are displayed side by side (Figure 6). From the reconstruction, you can observe that the general structure and appearance of the faces are kept while the fine details are lost. This is a remarkable reduction (more than 10x) in the dataset size that can help speed up your learning algorithm significantly. For example, if you were training a neural network to perform person recognition (given a face image, predict the identity of the person), you can use the dimension reduced input of only a 100 dimensions instead of the original pixels.

1.5 PCA for visualization

In the earlier K-means image compression exercise (MLI6), you used the K-means algorithm in the 3-dimensional RGB space of a bird image (Figure 7). In the last part of the `ex7_pca.m` script, we have provided code to visualize the final pixel assignments in this 3D space using the `scatter3` function (Figure 8). Each data point is colored according to the cluster it has been assigned to. You can drag your mouse on the figure to rotate and inspect this data in 3 dimensions.

Figure 7: RGB image

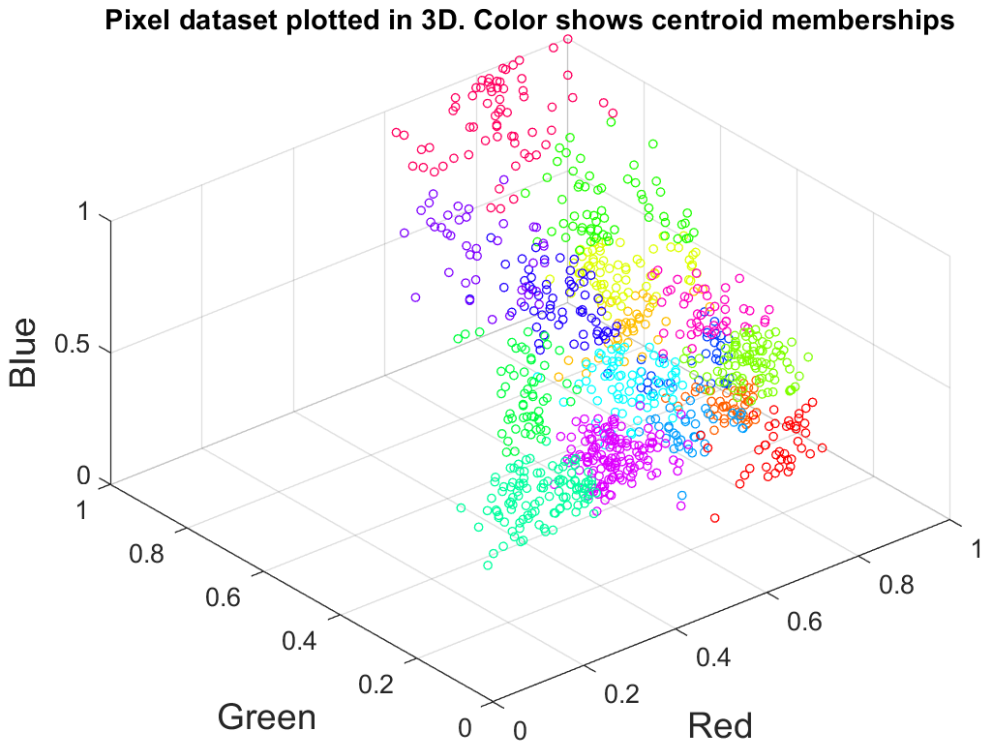

Figure 8:RGB pixel values colored according to cluster membership

It turns out that visualizing datasets in 3 dimensions or greater can be cumbersome. Therefore, it is often desirable to only display the data in 2D even at the cost of losing some information. In practice, PCA is often used to reduce the dimensionality of data for visualization purposes. In the next part of `ex7_pca.m`, the script will apply your implementation of PCA to the 3-dimensional data to reduce it to 2 dimensions and visualize the result in a 2D scatter plot (Figure 9). The PCA projection can be thought of as a rotation that selects the view that maximizes the spread of the data, which often corresponds to the "best" view.

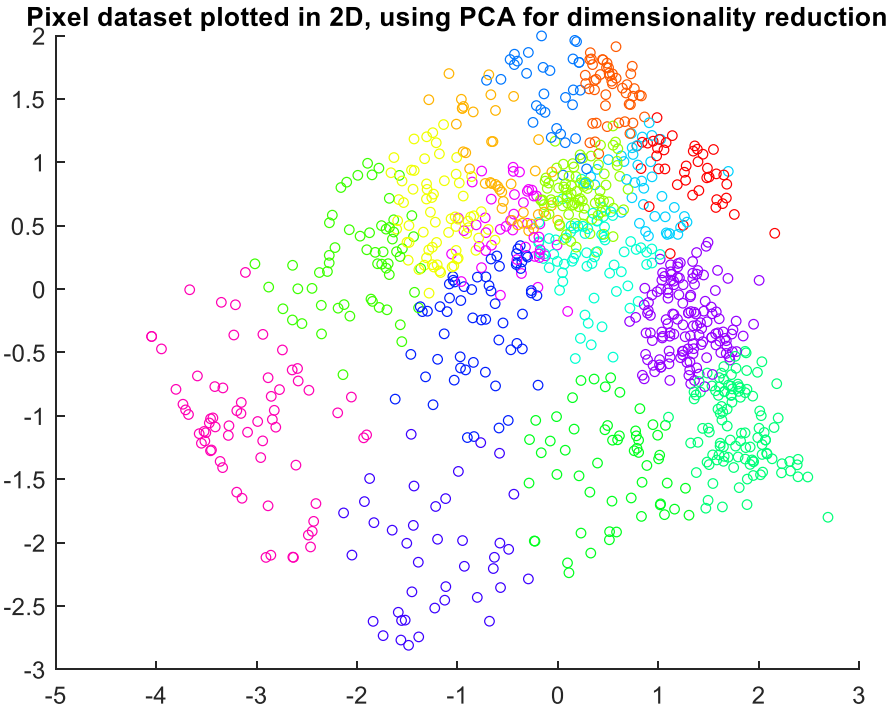

Figure 9: 2D visualization produced by PCA

2 MLII2 – Anomaly Detection

In this exercise, you will implement the anomaly detection algorithm and apply it to detect failing servers on a network. You will recognize that even simple probability calculus, as it is used in this algorithm, can be applied very effectively to provide additional benefit in practical applications. Furthermore, you will once more calculate and apply performance metrics for tuning a machine learning system.

For this exercise you will need the following files:

- `ex8.m` - Octave/Matlab script for running this exercise
- `ex8data1.mat` - First example Dataset for anomaly detection
- `ex8data2.mat` - Second example Dataset for anomaly detection
- `multivariateGaussian.m` - Computes the probability density function for a Gaussian distribution
- `visualizeFit.m` - 2D plot of a Gaussian distribution and a dataset
- `estimateGaussian.m*)` - Estimate the parameters of a Gaussian distribution with a diagonal covariance matrix
- `selectThreshold.m*)` - Find a threshold for anomaly detection

You will have to modify the files marked with `*)`.

Throughout this exercise (anomaly detection), you will be using the script `ex8.m`. This script sets up the dataset for the problems and makes calls to functions that you will write. You are only required to modify functions in other files, by following the instructions in this assignment.

2.1 Example dataset

In this exercise, you will implement an anomaly detection algorithm to detect anomalous behavior in server computers. The features measure the throughput (mb/s) and latency (ms) of response of each server. While your servers were operating, you collected $m = 307$ examples of how they were behaving, and thus have an unlabeled dataset $\{\vec{x}^{(1)}, \dots, \vec{x}^{(m)}\}$. You suspect that the vast majority of these examples are "normal" (non-anomalous) examples of the servers operating normally, but there might also be some examples of servers acting anomalously within this dataset.
You will use a Gaussian model to detect anomalous examples in your dataset. You will first start on a 2D dataset that will allow you to visualize what the algorithm is doing. On that dataset you will fit a Gaussian distribution and then find values that have very low probability and hence can be considered anomalies. After that, you will apply the anomaly detection algorithm to a larger dataset with many dimensions. You will be using `ex8.m` for this part of the exercise.
The first part of `ex8.m` will visualize the dataset as shown in Figure 10.

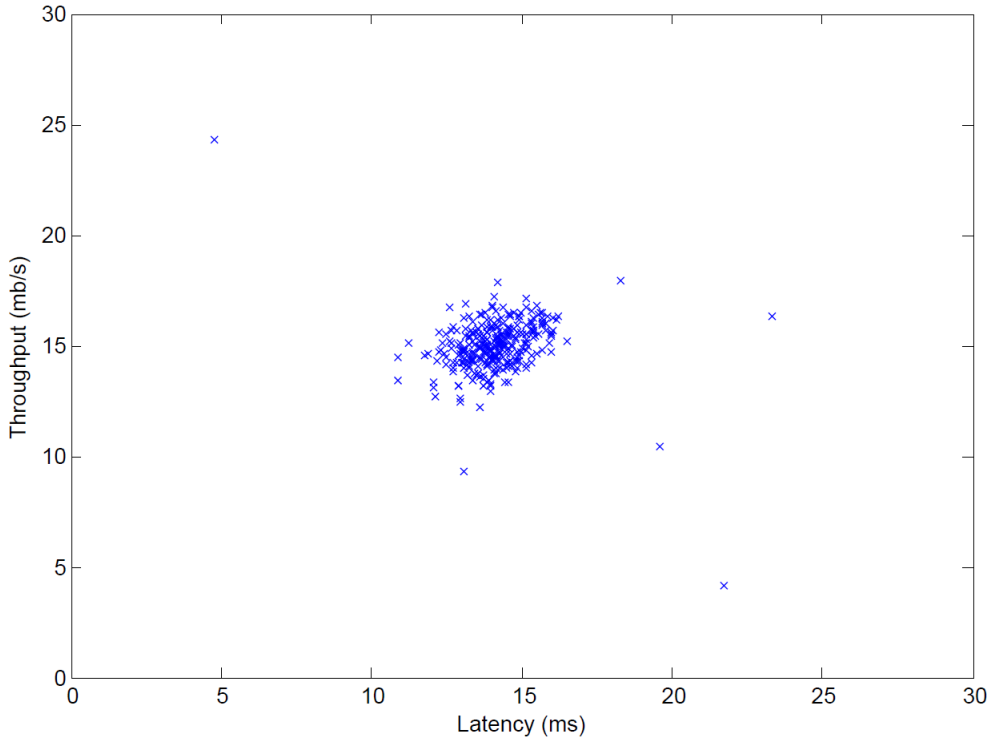

Figure 10: The first dataset

2.2 Gaussian distribution

To perform anomaly detection, you will first need to fit a model to the data's distribution.

Given a training set, $\{\vec{x}^{(1)}, \ldots, \vec{x}^{(m)}\}$ $\left(\text{where } \vec{x}^{(i)} \in \mathbb{R}^n\right)$, you want to estimate the Gaussian distribution for each of the features x_i. For each feature $i = 1 \ldots n$, you need to find parameters μ_i and σ_i^2 that fit the data in the i-th dimension $\left\{x_i^{(1)}, \ldots, x_i^{(m)}\right\}$ (i-th dimension of each example).

The Gaussian distribution is given by

$$p(x, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ^2 controls the variance.

2.3 Estimating parameters for a Gaussian

You can estimate the parameters, (μ_i, σ_i^2), of the i-th feature by using the following equations. To estimate the mean, you will use:

$$\mu_i = \frac{1}{m} \sum_{j=1}^{m} x_i^{(j)},$$

and for the variance you will use:

$$\sigma_i^2 = \frac{1}{m} \sum_{j=1}^{m} \left(x_i^{(j)} - \mu_i\right)^2.$$

Your task is to complete the code in `estimateGaussian.m`. This function takes as input the data matrix `X` and should output an n-dimension vector `mu` that holds the mean of all the n-features and another n-dimension vector `sigma2` that holds the variances of all the features. You can implement this using a for-loop over every feature and every training example (though a vectorized implementation might be more efficient; feel free to use a vectorized implementation if you prefer). Note that in Octave, the `var` function will (by default) use $\frac{1}{m-1}$, instead of $\frac{1}{m}$, when computing σ_i^2.

Once you have completed the code in `estimateGaussian.m`, the next part of `ex8.m` will visualize the contours of the fitted Gaussian distribution. You should get a plot similar to Figure 11. From your plot, you can see that most of the examples are in the region with the highest probability, while the anomalous examples are in the regions with lower probabilities.

You should now show your estimate Gaussian parameters function to an instructor.

2.4 Selecting the threshold, ε

Now that you have estimated the Gaussian parameters, you can investigate which examples have a very high probability given this distribution and which examples have a very low probability.

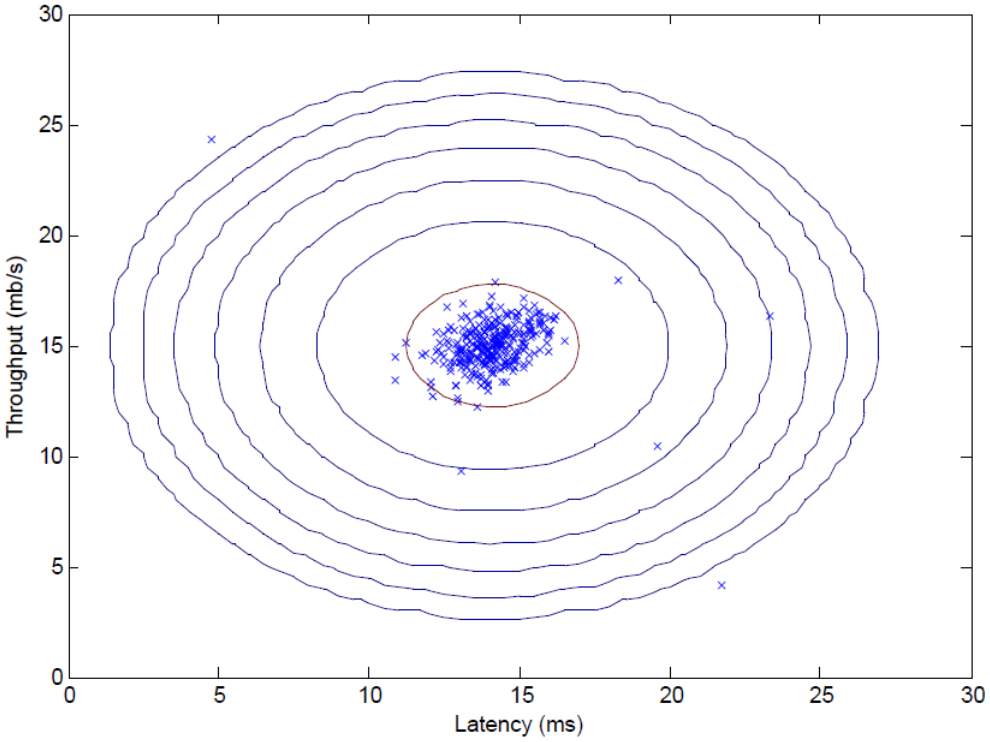

Figure 11: The Gaussian distribution contours of the distribution fit to the dataset

The low probability examples are more likely to be the anomalies in our dataset. One way to determine which examples are anomalies is to select a threshold based on a cross validation set. In this part of the exercise, you will implement an algorithm to select the threshold ε using the F_1 score on a cross validation set.

You should now complete the code in `selectThreshold.m`. For this, we will use a cross validation set $\left\{\left(\vec{x}_{cv}^{(1)}, y_{cv}^{(1)}\right), \ldots, \left(\vec{x}_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})}\right)\right\}$, where the label $y = 1$ corresponds to an anomalous example, and $y = 0$ corresponds to a normal example. For each cross-validation example, we will compute $p\left(\vec{x}_{cv}^{(i)}\right)$. The vector of all of these probabilities $p\left(\vec{x}_{cv}^{(1)}\right), \ldots, p\left(\vec{x}_{cv}^{(m_{cv})}\right)$ is passed to `selectThreshold.m` in the vector `pval`. The corresponding labels $y_{cv}^{(1)}, \ldots, y_{cv}^{(m_{cv})}$ are passed to the same function in the vector `yval`.

The function `selectThreshold.m` should return two values; the first is the selected threshold ε. If an example x has a low probability $p(x) < \varepsilon$, then it is considered to be an anomaly. The function should also return the F_1 score, which tells you how well you're doing on finding the ground truth anomalies given a certain threshold. For many different values of ε, you will compute the resulting F_1 score by computing how many examples the current threshold classifies correctly and incorrectly.

The F_1 score is computed using precision ($prec$) and recall (rec):

$$F_1 = \frac{2 \cdot prec \cdot rec}{prec + rec},$$

You compute precision and recall by:

$$prec = \frac{tp}{tp + fp}$$

$$rec = \frac{tp}{tp + fn},$$

where

- tp is the number of true positives: the ground truth label says it's an anomaly and our algorithm correctly classified it as an anomaly.
- fp is the number of false positives: the ground truth label says it's not an anomaly, but our algorithm incorrectly classified it as an anomaly.
- fn is the number of false negatives: the ground truth label says it's an anomaly, but our algorithm incorrectly classified it as not being anomalous.

In the provided code `selectThreshold.m`, there is already a loop that will try many different values of ε and select the best ε based on the F_1 score.
You should now complete the code in `selectThreshold.m`. You can implement the computation of the `F1` score using a for-loop over all the cross validation examples (to compute the values tp, fp, fn). You should see a value for epsilon of about 8.99e-05.

***Implementation Note**: In order to compute tp, fp and fn, you may be able to use a vectorized implementation rather than loop over all the examples. This can be implemented by Octave's equality test between a vector and a single number. If you have several binary values in an n-dimensional binary vector $\in \{0,1\}^n$, you can find out how many values in this vector are* 0 *by using:* `sum(v == 0)`*. You can also apply a logical* `and` *operator to such binary vectors. For instance, let* `cvPredictions` *be a binary vector of the size of your number of cross validation set, where the i-th element is* 1 *if your algorithm considers $x_{cv}^{(i)}$ an anomaly, and* 0 *otherwise. You can then, for example, compute the number of false positives using:* `fp = sum((cvPredictions == 1) & (yval == 0))`.

Once you have completed the code in `selectThreshold.m`, the next step in `ex8.m` will run your anomaly detection code and circle the anomalies in the plot (Figure 12).

You should now show your select threshold function to an instructor.

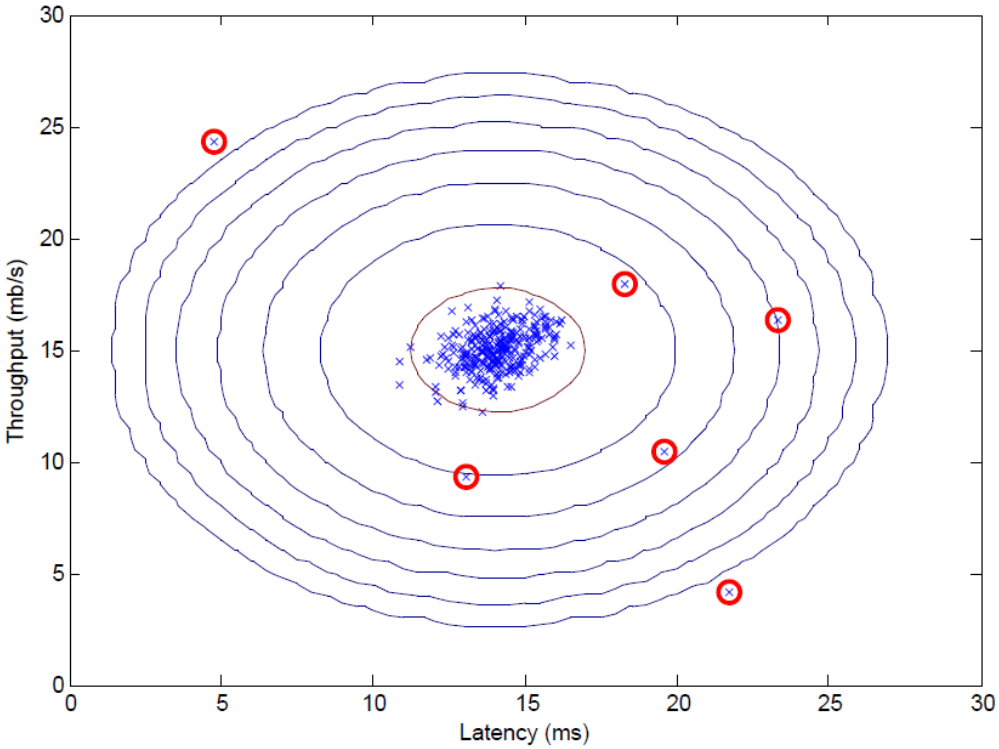

Figure 12: The classified anomalies

2.5 High dimensional dataset

The last part of the script `ex8.m` will run the anomaly detection algorithm you implemented on a more realistic and much harder dataset. In this dataset, each example is described by 11 features, capturing many more properties of your compute servers.

The script will use your code to estimate the Gaussian parameters (μ_i and σ_i^2), evaluate the probabilities for both the training data `X` from which you estimated the Gaussian parameters, and do so for the the cross-validation set `Xval`. Finally, it will use `selectThreshold` to find the best threshold ε. You should see a value epsilon of about 1.38e-18, and 117 anomalies found.

3 MLII3 – Recommender Systems

In this exercise, you will use collaborative filtering to build a recommender system for movies. The implementation of the algorithm will enable you to recognize the similarities with fundamental supervised learning algorithms and how actually working systems can be built by cleverly combing and varying them. By watching a typical application running and understanding the algorithmic background, you are able to assess the relevance of recommender systems for current online business.

For this exercise you will need the following files:

- `ex8_cofi.m` - Octave/Matlab script for running the exercise
- `ex8_movies.mat` - Movie Review Dataset
- `ex8_movieParams.mat` - Parameters provided for debugging
- `checkCostFunction.m` - Gradient checking for collaborative filtering
- `computeNumericalGradient.m` - Numerically compute gradients
- `fmincg.m` - Function minimization routine (similar to fminunc)
- `loadMovieList.m` - Loads the list of movies into a cell-array
- `movie_ids.txt` - List of movies
- `cofiCostFunc.m*)` - Implement the cost function for collaborative filtering

You will have to modify the files marked with `*)`.

Throughout the exercise, you will be using the script `ex8_cofi.m`. This script sets up the dataset for the problems and makes calls to functions that you will write. You are only required to modify functions in other files, by following the instructions in this assignment.

3.1 Movie ratings dataset

In this exercise, you will implement the collaborative filtering learning algorithm and apply it to a dataset of movie ratings.[1] This dataset consists of ratings on a scale of 1 to 5. The dataset has $n_u = 943$ users, and $n_m = 1682$ movies. You will be working with the script `ex8_cofi.m`.
In the next parts of this exercise, you will implement the function `cofiCostFunc.m` that computes the collaborative fitlering objective function and gradient. After implementing the cost function and gradient, you will use `fmincg.m` to learn the parameters for collaborative filtering.
The first part of the `script_ex8 cofi.m` will load the dataset `ex8_movies.mat`, providing the variables `Y` and `R` in your Octave/Matlab environment.
The matrix Y (a `num_movies × num_users` matrix) stores the ratings $y^{(i,j)}$ (from 1 to 5). The matrix R is a binary-valued indicator matrix, where $R(i,j) = 1$ if user j gave a rating to movie i, and $R(i,j) = 0$ otherwise. The objective of collaborative filtering is to predict movie ratings for the movies that users have not yet rated, that is, the entries with $R(i,j) = 0$. This will allow us to recommend the movies with the highest predicted ratings to the user.
To help you understand the matrix `Y`, the script `ex8_cofi.m` will compute the average movie rating for the first movie (Toy Story) and output the average rating to the screen.
Throughout this part of the exercise, you will also be working with the matrices, `X` and `Theta`:

$$\mathrm{X} = \begin{bmatrix} -\left(\vec{x}^{(1)}\right)^T - \\ -\left(\vec{x}^{(2)}\right)^T - \\ \vdots \\ -\left(\vec{x}^{(n_m)}\right)^T - \end{bmatrix}, \qquad \mathrm{Theta} = \begin{bmatrix} -\left(\vec{\theta}^{(1)}\right)^T - \\ -\left(\vec{\theta}^{(2)}\right)^T - \\ \vdots \\ -\left(\vec{\theta}^{(n_u)}\right)^T - \end{bmatrix}.$$

[1] MovieLens 100k Dataset from GroupLens Research.

The i-th row of `X` corresponds to the feature vector $\vec{x}^{(i)}$ for the i-th movie, and the j-th row of `Theta` corresponds to one parameter vector $\vec{\theta}^{(j)}$, for the j-th user. Both $\vec{x}^{(i)}$ and $\vec{\theta}^{(j)}$ are n-dimensional vectors. For the purposes of this exercise, you will use $n = 100$, and therefore, $\vec{x}^{(i)} \in \mathbb{R}^{100}$ and $\vec{\theta}^{(j)} \in \mathbb{R}^{100}$. Correspondingly, `X` is a n_m × 100 matrix and `Theta` is a n_u × 100 matrix.

3.2 Collaborative filtering learning algorithm

Now, you will start implementing the collaborative filtering learning algorithm. You will start by implementing the cost function (without regularization).

The collaborative filtering algorithm in the setting of movie recommendations considers a set of n-dimensional parameter vectors $\vec{x}^{(1)}, \dots, \vec{x}^{(n_m)}$ and $\vec{\theta}^{(1)}, \dots, \vec{\theta}^{(n_u)}$, where the model predicts the rating for movie i by user j as $y^{(i,j)} = \left(\vec{\theta}^{(j)}\right)^T \vec{x}^{(i)}$. Given a dataset that consists of a set of ratings produced by some users on some movies, you wish to learn the parameter vectors $\vec{x}^{(1)}, \dots, \vec{x}^{(n_m)}, \vec{\theta}^{(1)}, \dots, \vec{\theta}^{(n_u)}$ that produce the best fit (minimizes the squared error).

You will complete the code in `cofiCostFunc.m` to compute the cost function and gradient for collaborative filtering. Note that the parameters to the function (i.e., the values that you are trying to learn) are `X` and `Theta`. In order to use an off-the-shelf minimizer such as `fmincg`, the cost function has been set up to unroll the parameters into a single vector `params`. Unrolling means that the elements of the matrix `X` and the elements of the matrix `Theta` are all written one after another into `params(1), params(2),…,params(np)`, where $np = n * n_m + n * n_u$.

3.2.1 Collaborative filtering cost function

The collaborative filtering cost function (without regularization) is given by

$$J\left(\vec{x}^{(1)}, \dots, \vec{x}^{(n_m)}, \vec{\theta}^{(1)}, \dots, \vec{\theta}^{(n_u)}\right) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} \left(\left(\vec{\theta}^{(j)}\right)^T \vec{x}^{(i)} - y^{(i,j)}\right)^2.$$

You should now modify `cofiCostFunc.m` to return this cost in the variable `J`. Note that you should be accumulating the cost for user j and movie i only if $R(i,j) = 1$.

After you have completed the function, the script `ex8_cofi.m` will run your cost function. You should expect to see an output of 22.22.

You should now show your cost function an instructor.

Implementation Note: *We strongly encourage you to use a vectorized implementation to compute J, since it will later be called many times by the optimization package* `fmincg`*. As usual, it might be easiest to first write a non-vectorized implementation (to make sure you have the right answer), and then modify it to become a vectorized implementation (checking that the vectorization steps do not change your algorithm's output). To come up with a vectorized implementation, the following tip might be helpful: You can use the* `R` *matrix to set selected entries to* 0*. For example,* `R .* M` *will do an element-wise multiplication between* `M` *and* `R`*; since* `R` *only has elements with values either* 0 *or* 1*, this has the effect of setting the elements of* `M` *to* 0 *only when the corresponding value in R is* 0*. Hence,* `sum(sum(R.*M))` *is the sum of all the elements of* `M` *for which the corresponding element in* `R` *equals* 1*.*

3.2.2 Collaborative filtering gradient

Now, you should implement the gradient (without regularization). Specifically, you should complete the code in `cofiCostFunc.m` to return the variables `X_grad` and `Theta_grad`. Note that `X_grad` should be a matrix of the same size as `X` and similarly, `Theta_grad` is a matrix of the same size as `Theta`. The gradients of the cost function is given by:

$$\frac{\partial J}{\partial x_k^{(i)}} = \sum_{j:r(i,j)=1} \left(\left(\vec{\theta}^{(j)} \right)^T \vec{x}^{(i)} - y^{(i,j)} \right) \theta_k^{(j)}$$

$$\frac{\partial J}{\partial \theta_k^{(j)}} = \sum_{i:r(i,j)=1} \left(\left(\vec{\theta}^{(j)} \right)^T \vec{x}^{(i)} - y^{(i,j)} \right) x_k^{(i)} .$$

Note that the function returns the gradient for both sets of variables by unrolling them into a single vector. After you have completed the code to compute the gradients, the script `ex8 cofi.m` will run a gradient check (`checkCostFunction`) to numerically check the implementation of your gradients.[2] If your implementation is correct, you should find that the analytical and numerical gradients match up closely.

You should now show your collaborative filtering gradient function to an instructor.

Implementation Note: *You can get full credit for this assignment without using a vectorized implementation, but your code will run much more slowly (a small number of hours), and so we recommend that you try to vectorize your implementation.*
To get started, you can implement the gradient with a for-loop over movies (for computing $\frac{\partial J}{\partial x_k^{(i)}}$ *) and a for-loop over users (for computing* $\frac{\partial J}{\partial \theta_k^{(j)}}$ *). When you first implement the gradient, you might start with an unvectorized version, by implementing another inner for-loop that computes each element in the summation. After you have completed the gradient computation this way, you should try to vectorize your implementation (vectorize the inner for-loops), so that you're left with only two for-loops (one for looping over movies to compute* $\frac{\partial J}{\partial x_k^{(i)}}$ *for each movie, and one for looping over users to compute* $\frac{\partial J}{\partial \theta_k^{(j)}}$ *for each user).*

Implementation Tip: *To perform the vectorization, you might find this helpful: You should come up with a way to compute all the derivatives associated with* $x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)}$ *(i.e., the derivative terms associated with the feature vector* $\vec{x}^{(i)}$ *) at the same time. Let us define the derivatives for the feature vector of the* i*-th movie as:*

$$\left(X_{grad}(i,:) \right)^T = \begin{bmatrix} \frac{\partial J}{\partial x_1^{(i)}} \\ \frac{\partial J}{\partial x_2^{(i)}} \\ \vdots \\ \frac{\partial J}{\partial x_n^{(i)}} \end{bmatrix} = \sum_{j:r(i,j)=1} \left(\left(\vec{\theta}^{(j)} \right)^T \vec{x}^{(i)} - y^{(i,j)} \right) \vec{\theta}^{(j)}$$

To vectorize the above expression, you can start by indexing into `Theta` *and* `Y` *to select only the elements of interests (that is, those with* $r(i,j) = 1$*). Intuitively, when you consider the features for the* i*-th movie, you only need to be concern about the users who had given ratings to the movie, and this allows you to remove all the other users from* `Theta` *and* `Y`*.*
Concretely, you can set `idx = find(R(i, :)==1)` *to be a list of all the users that have rated movie* i*. This will allow you to create the temporary matrices* $Theta_{temp} = Theta(idx,:)$ *and*

[2] This is similar to the numerical check that you will use in the neural networks exercise.

$Y_{temp} = Y(i, idx)$ that index into `Theta` *and* `Y` *to give you only the set of users which have rated the i-th movie. This will allow you to write the derivatives as:*

$$X_{grad}(i,:) = \left(X(i,:) * Theta_{temp}^{T} - Y_{temp}\right) * Theta_{temp} .$$

(Note: The vectorized computation above returns a row-vector instead.)

After you have vectorized the computations of the derivatives with respect to $\vec{x}^{(i)}$, you should use a similar method to vectorize the derivatives with respect to $\vec{\theta}^{(j)}$ as well.

3.2.3 Regularized cost function

The cost function for collaborative filtering with regularization is given by

$$J\left(\vec{x}^{(1)}, \dots, \vec{x}^{(n_m)}, \vec{\theta}^{(1)}, \dots, \vec{\theta}^{(n_u)}\right) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} \left(\left(\vec{\theta}^{(j)}\right)^{T} \vec{x}^{(i)} - y^{(i,j)}\right)^2 + \left(\frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(\theta_k^{(j)}\right)^2\right) + \left(\frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} \left(x_k^{(i)}\right)^2\right).$$

You should now add regularization to your original computations of the cost function, J. After you are done, the script `ex8_cofi.m` will run your regularized cost function, and you should expect to see a cost of about 31.34.

You should now show your regularized cost function to an instructor.

3.2.4 Regularized gradient

Now that you have implemented the regularized cost function, you should proceed to implement regularization for the gradient. You should add to your implementation in `cofiCostFunc.m` to return the regularized gradient by adding the contributions from the regularization terms. Note that the gradients for the regularized cost function is given by:

$$\frac{\partial J}{\partial x_k^{(i)}} = \sum_{j:r(i,j)=1} \left(\left(\vec{\theta}^{(j)}\right)^{T} \vec{x}^{(i)} - y^{(i,j)}\right) \theta_k^{(j)} + \lambda x_k^{(i)}$$

$$\frac{\partial J}{\partial \theta_k^{(j)}} = \sum_{i:r(i,j)=1} \left(\left(\vec{\theta}^{(j)}\right)^{T} \vec{x}^{(i)} - y^{(i,j)}\right) x_k^{(i)} + \lambda \theta_k^{(j)} .$$

This means that you just need to add $\lambda \vec{x}^{(i)}$ to the `X_grad(i,:)` variable described earlier, and add $\lambda \vec{\theta}^{(j)}$ to the `Theta_grad(j,:)` variable described earlier.
After you have completed the code to compute the gradients, the script `ex8_cofi.m` will run another gradient check (`checkCostFunction`) to numerically check the implementation of your gradients.

You should now show the regularized gradient function to an instructor.

3.3 Learning movie recommendations

After you have finished implementing the collaborative filtering cost function and gradient, you can now start training your algorithm to make movie recommendations for yourself. In the next part of the `ex8_cofi.m` script, you can enter your own movie preferences, so that later when the algorithm runs, you can get your own movie recommendations! We have filled out some values according to our own preferences, but you should change this according to your own tastes. The list of all movies and their number in the dataset can be found listed in the file `movie_idx.txt`.

3.3.1 Recommendations

Top recommendations for you:
Predicting rating 4.3 for movie Titanic (1997)
Predicting rating 4.2 for movie Star Wars (1977)
Predicting rating 4.0 for movie Raiders of the Lost Ark (1981)
Predicting rating 3.9 for movie Shawshank Redemption, The (1994)
Predicting rating 3.9 for movie Return of the Jedi (1983)
Predicting rating 3.9 for movie Braveheart (1995)
Predicting rating 3.9 for movie Good Will Hunting (1997)
Predicting rating 3.8 for movie Empire Strikes Back, The (1980)
Predicting rating 3.8 for movie Schindler's List (1993)
Predicting rating 3.8 for movie Air Force One (1997) Original ratings provided:
Rated 4 for Toy Story (1995)
Rated 3 for Twelve Monkeys (1995)
Rated 5 for Usual Suspects, The (1995)
Rated 4 for Outbreak (1995)
Rated 5 for Shawshank Redemption, The (1994)
Rated 3 for While You Were Sleeping (1995)
Rated 5 for Forrest Gump (1994)
Rated 2 for Silence of the Lambs, The (1991)
Rated 4 for Alien (1979)
Rated 5 for Die Hard 2 (1990)
Rated 5 for Sphere (1998)

Figure 13: Movie recommendations

After the additional ratings have been added to the dataset, the script will proceed to train the collaborative filtering model. This will learn the parameters `X` and `Theta`. To predict the rating of movie i for user j, you need to compute $\left(\vec{\theta}^{(j)}\right)^T \vec{x}^{(i)}$. The next part of the script computes the ratings for all the movies and users and displays the movies that it recommends (Figure 13), according to ratings that were entered earlier in the script. Note that you might obtain a different set of the predictions due to different random initializations.

4 MLII4 – Support Vector Machine

Support Vector Machine (SVM) is a very popular and widely used classification algorithm. It´s real strength lies in its ability to effectively deal with highly non-linear separable problems. The objective of this exercise is to get an intuition on how this algorithm works on linear and non-linear separable classification problems and to enable you to apply it on real-world problems. You will try out SVM on some example datasets to study its behaviour and use it to build a spam classifier.

For this exercise you will need the following files:

- `ex6.m` – Octave/Matlab script for the first half of the exercise
- `ex6data1.mat` - Example Dataset 1
- `ex6data2.mat` - Example Dataset 2
- `ex6data3.mat` - Example Dataset 3
- `svmTrain.m` - SVM training function
- `svmPredict.m` - SVM prediction function
- `plotData.m` - Plot 2D data
- `visualizeBoundaryLinear.m` - Plot linear boundary
- `visualizeBoundary.m` - Plot non-linear boundary
- `linearKernel.m` - Linear kernel for SVM
- `gaussianKernel.m*)` - Gaussian kernel for SVM
- `dataset3Params.m*)` - Parameters to use for Dataset 3
- `ex6_spam.m` – Octave/Matlab script for the second half of the exercise
- `spamTrain.mat` - Spam training set
- `spamTest.mat` - Spam test set
- `emailSample1.txt` - Sample email 1
- `emailSample2.txt` - Sample email 2
- `spamSample1.txt` - Sample spam 1
- `spamSample2.txt` - Sample spam 2
- `vocab.txt` - Vocabulary list
- `getVocabList.m` - Load vocabulary list
- `porterStemmer.m` - Stemming function
- `readFile.m` - Reads a file into a character string
- `submit.m` - Submission script that sends your solutions to our servers
- `submitWeb.m` - Alternative submission script
- `processEmail.m*)` - Email preprocessing
- `emailFeatures.m*)` - Feature extraction from emails

You will have to modify the files marked with `*)`.

Throughout the exercise, you will be using the script `ex6.m` and `ex6_spam.m`. These scripts set up the dataset for the problems and make calls to functions that you will write. You are only required to modify functions in other files, by following the instructions in this assignment.

4.1 Understanding SVM

In the first half of this exercise, you will be using support vector machines (SVMs) with various example 2D datasets. Experimenting with these datasets will help you gain an intuition of how SVMs work and how to use a Gaussian kernel with SVMs. In the next half of the exercise, you will be using support vector machines to build a spam classifier.
The provided script, `ex6.m`, will help you step through the first half of the exercise.

4.1.1 Example Dataset 1

We will begin by with a 2D example dataset which can be separated by a linear boundary. The script `ex6.m` will plot the training data (Figure 14). In this dataset, the positions of the positive examples (indicated with $+$) and the negative examples (indicated with o) suggest a natural separation indicated by the gap. However, notice that there is an outlier positive example + on the far left at about (0.1, 4.1). As part of this exercise, you will also see how this outlier affects the SVM decision boundary.

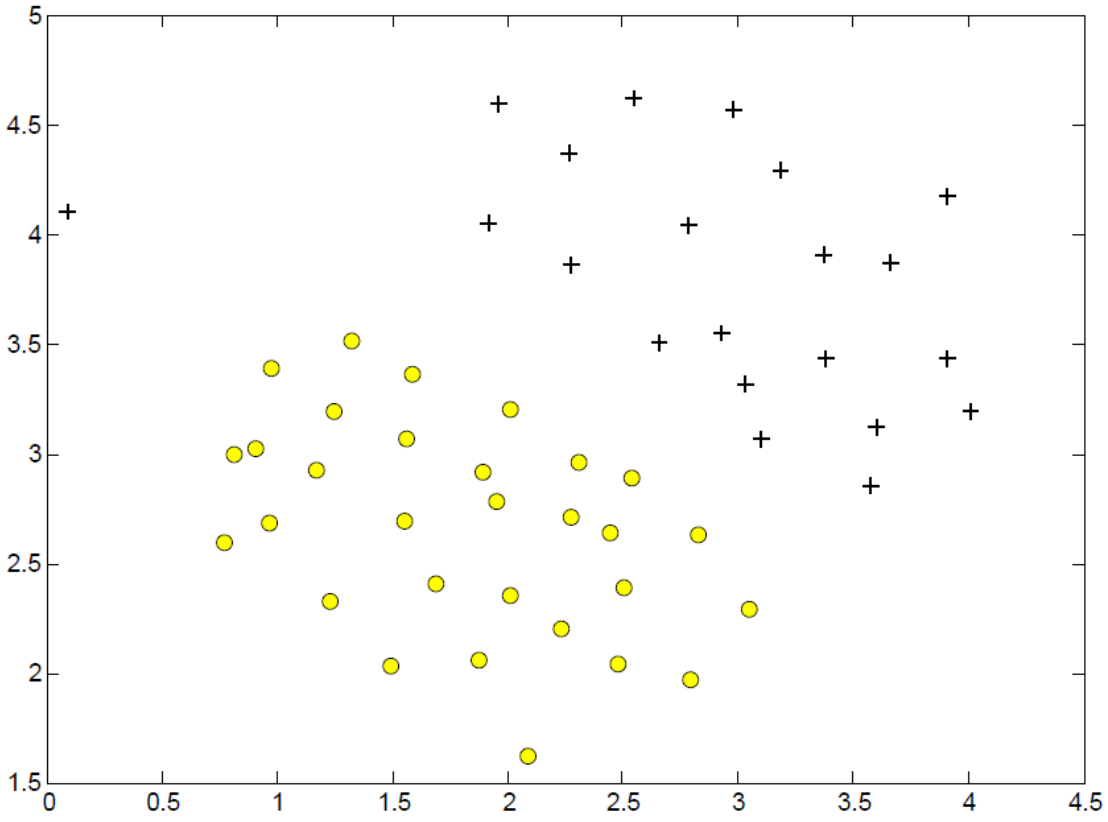

Figure 14: Example Dataset 1

In this part of the exercise, you will try using different values of the C parameter with SVMs. Informally, the C parameter is a positive value that controls the penalty for misclassified training examples. A large C parameter tells the SVM to try to classify all the examples correctly. C plays a role similar to $\frac{1}{\lambda}$, where λ is the regularization parameter that we were using previously for logistic regression.

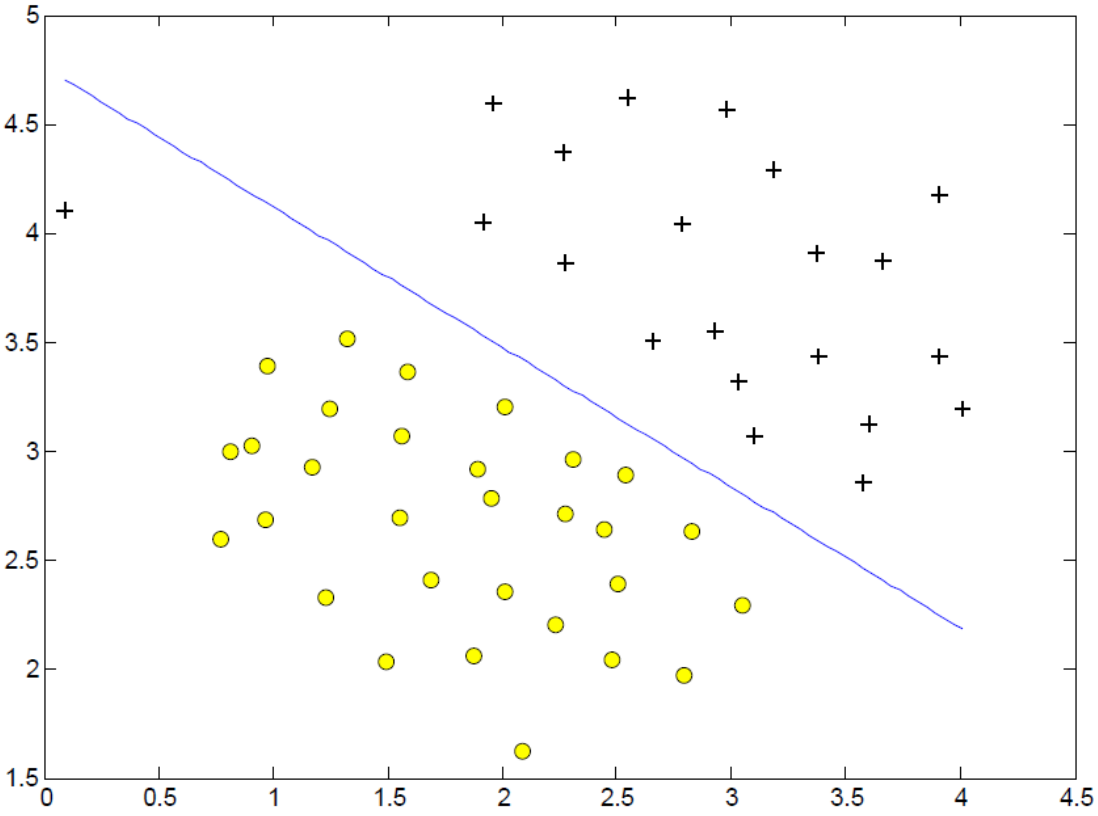

Figure 15: SVM Decision Boundary with C = 1 (Example Dataset 1)

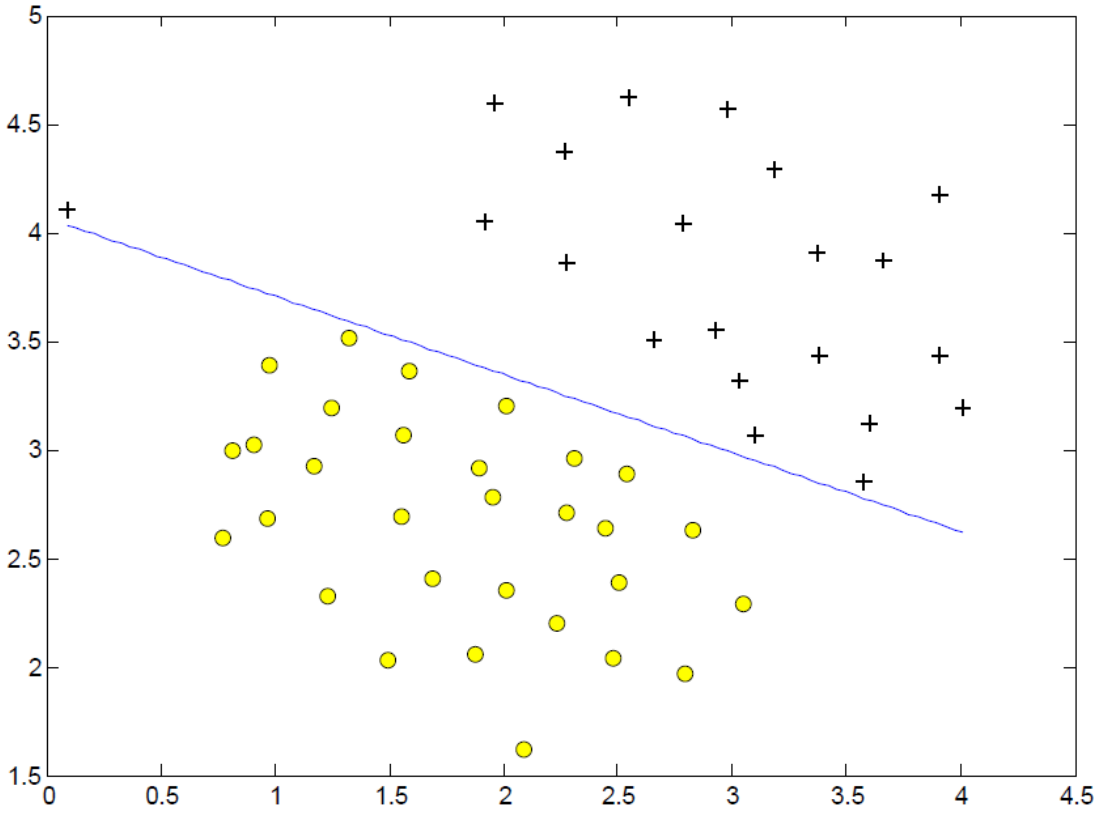

Figure 16: SVM Decision Boundary with C = 100 (Example Dataset 1)

The next part in `ex6.m` will run the SVM training (with $C = 1$) using SVM software that we have included with the starter code, `svmTrain.m.1`.[3] When $C = 1$, you should find that the SVM puts the decision boundary in the gap between the two datasets and *misclassifies* the data point on the far left (Figure 15).

Implementation Note: *Most SVM software packages (including* `svmTrain.m`*) automatically add the extra feature* $x_0 = 1$ *for you and automatically take care of learning the intercept term* θ_0*. So when passing your training data to the SVM software, there is no need to add this extra feature* $x_0 = 1$ *yourself. In particular, in Octave your code should be working with training examples* $\vec{x} \in \mathbb{R}^n$ *(rather than* $\vec{x} \in \mathbb{R}^{n+1}$*); for example, in the first example dataset* $\vec{x} \in \mathbb{R}^2$*.*

Your task is to try different values of C on this dataset. Specifically, you should change the value of C in the script to $C = 100$ and run the SVM training again. When $C = 100$, you should find that the SVM now classifies every single example correctly, but has a decision boundary that does not appear to be a natural fit for the data (Figure 16).

4.2 SVM with Gaussian Kernels

In this part of the exercise, you will be using SVMs to do non-linear classification. In particular, you will be using SVMs with Gaussian kernels on datasets that are not linearly separable.

4.2.1 Gaussian Kernel

To find non-linear decision boundaries with the SVM, we need to first implement a Gaussian kernel. You can think of the Gaussian kernel as a similarity function that measures the "distance" between a pair of examples, $\left(\vec{x}^{(i)}, \vec{x}^{(j)}\right)$. The Gaussian kernel is also parameterized by a bandwidth parameter, σ, which determines how fast the similarity metric decreases (to 0) as the examples are further apart.
You should now complete the code in `gaussianKernel.m` to compute the Gaussian kernel between two examples, $\left(\vec{x}^{(i)}, \vec{x}^{(j)}\right)$. The Gaussian kernel function is defined as:

$$K_{gaussian}\left(x^{(i)}, x^{(j)}\right) = exp\left(-\frac{\left\|\vec{x}^{(i)} - \vec{x}^{(j)}\right\|^2}{2\sigma^2}\right) = exp\left(-\frac{\sum_{k=1}^{n}\left(x_k^{(i)} - x_k^{(j)}\right)^2}{2\sigma^2}\right)$$

Once you've completed the function `gaussianKernel.m`, the script `ex6.m` will test your kernel function on two provided examples and you should expect to see a value of 0.324652.

You should now show your function that computes the Gaussian kernel to an instructor.

[3] In order to ensure compatibility with Octave/Matlab, we have included this implementation of an SVM learning algorithm. However, this particular implementation was chosen to maximize compatibility, and is **not** very efficient. If you are training an SVM on a real problem, especially if you need to scale to a larger dataset, we strongly recommend instead using a highly optimized SVM toolbox such as LIBSVM.

4.2.2 Example Dataset 2

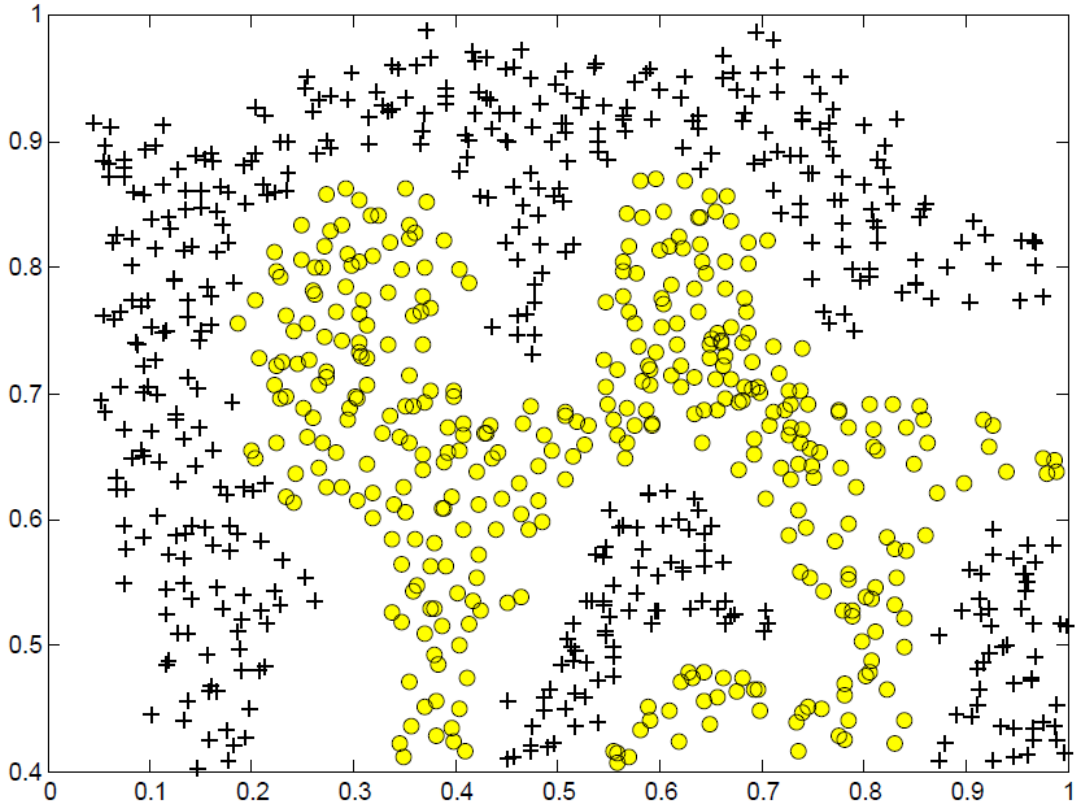

Figure 17: Example Dataset 2

The next part in `ex6.m` will load and plot `dataset` 2 (Figure 17). From the figure, you can obserse that there is no linear decision boundary that separates the positive and negative examples for this dataset. However, by using the Gaussian kernel with the SVM, you will be able to learn a non-linear decision boundary that can perform reasonably well for the dataset.

If you have correctly implemented the Gaussian kernel function, `ex6.m` will proceed to train the SVM with the Gaussian kernel on this dataset.

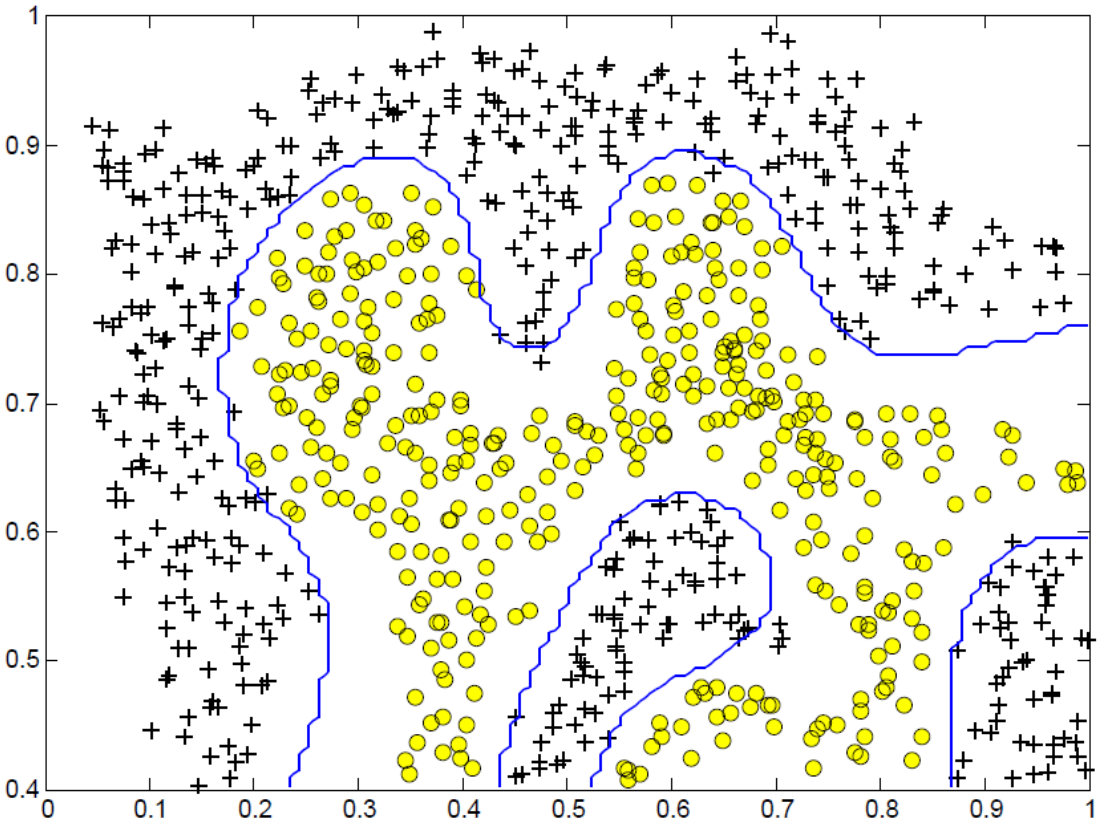

Figure 18: SVM (Gaussian Kernel) Decision Boundary (Example Dataset 2)

Figure 20 shows the decision boundary found by the SVM with a Gaussian kernel. The decision boundary is able to separate most of the positive and negative examples correctly and follows the contours of the dataset well.

4.2.3 Example Dataset 3

In this part of the exercise, you will gain more practical skills on how to use a SVM with a Gaussian kernel. The next part of `ex6.m` will load and display a third dataset (Figure 19). You will be using the SVM with the Gaussian kernel with this dataset.

In the provided dataset, `ex6data3.mat`, you are given the variables `X, y, Xval, yval`. The provided code in `ex6.m` trains the SVM classifier using the training set `(X, y)` using parameters loaded from `dataset3Params.m`.

Your task is to use the cross validation set `Xval, yval` to determine the best C and σ parameter to use. You should write any additional code necessary to help you search over the parameters C and σ. For *both* C and σ, we suggest trying values in multiplicative steps (e.g., 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30). Note that you should try all possible pairs of values for C and σ (e.g., $C = 0.3$ and $\sigma = 0.1$). For example, if you try each of the 8 values listed above for C and for σ^2, you would end up training and evaluating (on the cross validation set) a total of $8^2 = 64$ different models.

After you have determined the best C and σ parameters to use, you should modify the code in `dataset3Params.m`, filling in the best parameters you found. For our best parameters, the SVM returned a decision boundary shown in Figure 20.

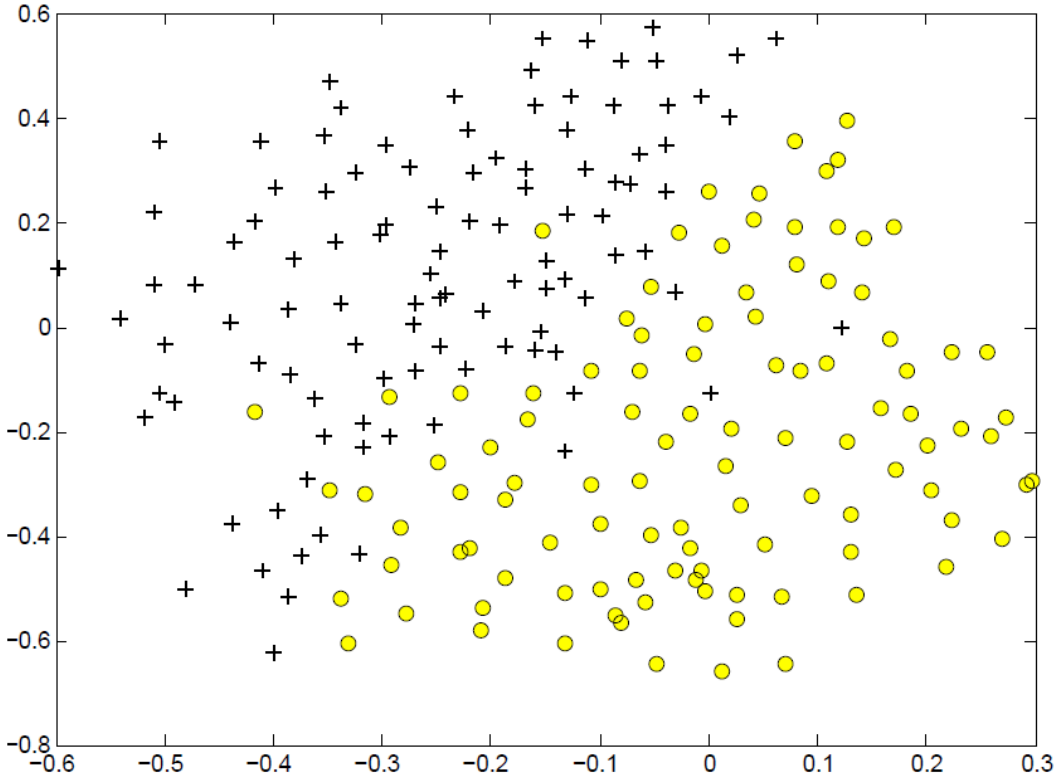

Figure 19: Example Dataset 3

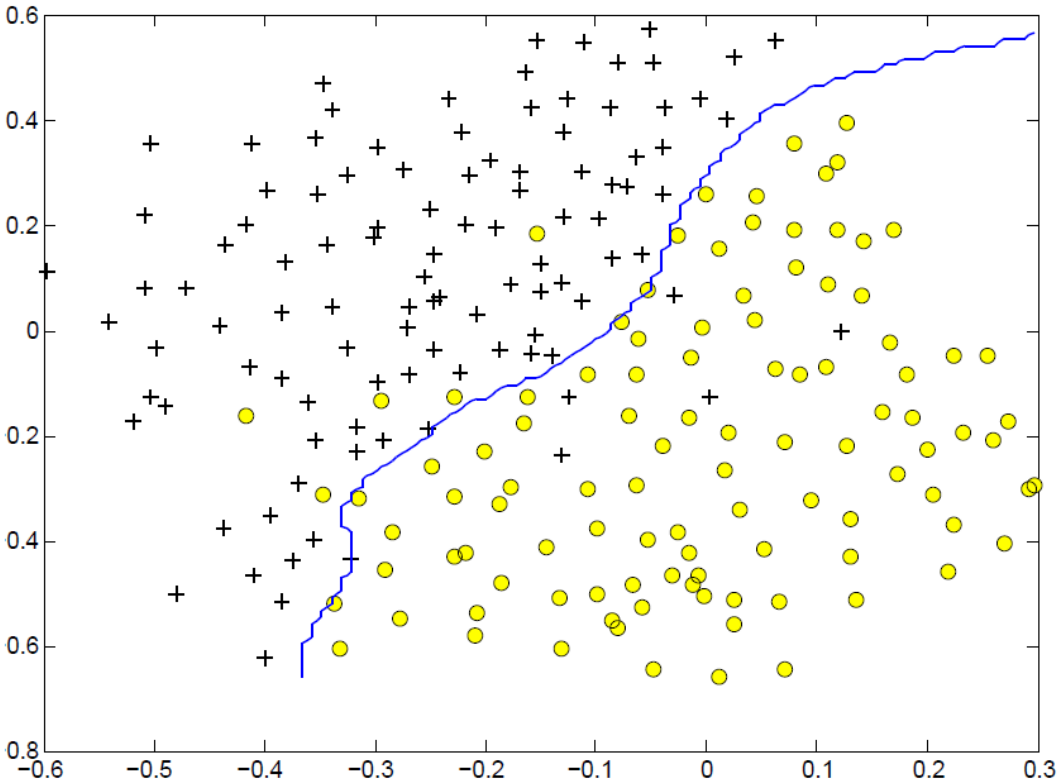

Figure 20: SVM (Gaussian Kernel) Decision Boundary (Example Dataset 3)

Implementation Tip: *When implementing cross validation to select the best* C and σ *parameter to use, you need to evaluate the error on the cross validation set. Recall that for classification, the error is defined as the fraction of the cross validation examples that were classified incorrectly. In*

Octave/Matlab, you can compute this error using `mean(double(predictions ~= yval))`*, where predictions is a vector containing all the predictions from the SVM, and* `yval` *are the true labels from the cross validation set.*
You can use the `svmPredict` *function to generate the predictions for the cross validation set.*

You should now show your best C and σ *values to an instructor.*

4.3 Spam Classification

Many email services today provide spam filters that are able to classify emails into spam and non-spam email with high accuracy. In this part of the exercise, you will use SVMs to build your own spam filter.
You will be training a classifier to classify whether a given email, x, is spam ($y = 1$) or non-spam ($y = 0$). In particular, you need to convert each email into a feature vector $x \in \mathbb{R}^n$. The following parts of the exercise will walk you through how such a feature vector can be constructed from an email.
Throughout the rest of this exercise, you will be using the the script `ex6_spam.m`. The dataset included for this exercise is based on a a subset of the SpamAssassin Public Corpus.[4] For the purpose of this exercise, you will only be using the body of the email (excluding the email headers).

4.4 Preprocessing Emails

> Anyone knows how much it costs to host a web portal ?
>
Well, it depends on how many visitors youre expecting. This can be anywhere from less than 10 bucks a month to a couple of $100. You should checkout http://www.rackspace.com/ or perhaps Amazon EC2 if youre running something big..

To unsubscribe yourself from this mailing list, send an email to:
groupname-unsubscribe@egroups.com

Figure 21: Sample Email

Before starting on a machine learning task, it is usually insightful to take a look at examples from the dataset. Figure 21 shows a sample email that contains a URL, an email address (at the end), numbers, and dollar amounts. While many emails would contain similar types of entities (e.g., numbers, other URLs, or other email addresses), the specific entities (e.g., the specific URL or specific dollar amount) will be different in almost every email. Therefore, one method often employed in processing emails is to "normalize" these values, so that all URLs are treated the same, all numbers are treated the same, etc. For example, we could replace each URL in the email with the unique string "httpaddr" to indicate that a URL was present.
This has the effect of letting the spam classifier make a classification decision based on whether any URL was present, rather than whether a specific URL was present. This typically improves the performance of a spam classifier, since spammers often randomize the URLs, and thus the odds of seeing any particular URL again in a new piece of spam is very small.
In `processEmail.m`, we have implemented the following email preprocessing and normalization steps:

- `Lower-casing`: The entire email is converted into lower case, so that captialization is ignored (e.g., `IndIcaTE` is treated the same as `Indicate`).
- `Stripping HTML`: All HTML tags are removed from the emails. Many emails often come with HTML formatting; we remove all the HTML tags, so that only the content remains.
- `Normalizing URLs`: All URLs are replaced with the text `"httpaddr"`.
- `Normalizing Email Addresses`: All email addresses are replaced with the text `"emailaddr"`.
- `Normalizing Numbers`: All numbers are replaced with the text `"number"`.

[4] http://spamassassin.apache.org/publiccorpus/

- `Normalizing Dollars`: All dollar signs ($) are replaced with the text "`dollar`".
- `Word Stemming`: Words are reduced to their stemmed form. For example, "discount", "discounts", "discounted" and "discounting" are all replaced with "`discount`". Sometimes, the Stemmer actually strips off additional characters from the end, so "include", "includes", "included", and "including" are all replaced with "`includ`".
- `Removal of non-words`: Non-words and punctuation have been removed. All white spaces (tabs, newlines, spaces) have all been trimmed to a single space character.

The result of these preprocessing steps is shown in Figure 22. While preprocessing has left word fragments and non-words, this form turns out to be much easier to work with for performing feature extraction.

anyon know how much it cost to host a web portal well it depend on how mani visitor your expect thi can be anywher from less than number buck a month to a coupl of dollarnumb you should checkout httpaddr or perhap amazon ecnumb if your run someth big to unsubscrib yourself from thi mail list send an email to emailaddr

Figure 22: Preprocessed Sample Email

1 aa
2 ab
3 abil
...
86 anyon
...
916 know
...
1898 zero
1899 zip

Figure 23: Vocabulary List

86 916 794 1077 883
370 1699 790 1822
1831 883 431 1171
794 1002 1893 1364
592 1676 238 162 89
688 945 1663 1120
1062 1699 375 1162
479 1893 1510 799
1182 1237 810 1895
1440 1547 181 1699
1758 1896 688 1676
992 961 1477 71 530
1699 531

Figure 24: Word Indices for Sample Email

4.4.1 Vocabulary List

After preprocessing the emails, we have a list of words (e.g., Figure 22) for each email. The next step is to choose which words we would like to use in our classifier and which we would want to leave out. For this exercise, we have chosen only the most frequently occuring words as our set of words considered (the vocabulary list). Since words that occur rarely in the training set are only in a few emails, they might cause the model to overfit our training set. The complete vocabulary list is in the file `vocab.txt` and also shown in Figure 23. Our vocabulary list was selected by choosing all words which occur at least a 100 times in the spam corpus, resulting in a list of 1899 words. In practice, a vocabulary list with about 10,000 to 50,000 words is often used.

Given the vocabulary list, we can now map each word in the preprocessed emails (e.g., Figure 22) into a list of word indices that contains the index of the word in the vocabulary list. Figure 24 shows the mapping for the sample email. Specifically, in the sample email, the word "anyone" was first normalized to "anyon" and then mapped onto the index 86 in the vocabulary list.

Your task now is to complete the code in `processEmail.m` to perform this mapping. In the code, you are given a string `str` which is a single word from the processed email. You should look up the word in the vocabulary list `vocabList` and find if the word exists in the vocabulary list. If the word exists,

you should add the index of the word into the `word_indices` variable. If the word does not exist, and is therefore not in the vocabulary, you can skip the word.
Once you have implemented `processEmail.m`, the script `ex6_spam.m` will run your code on the email sample and you should see an output similar to Figure 22 & Figure 24.

Octave/Matlab Tip: *In Octave/Matlab, you can compare two strings with the* `strcmp` *function. For example,* `strcmp(str1, str2)` *will return 1 only when both strings are equal. In the provided starter code,* `vocabList` *is a "cellarray" containing the words in the vocabulary. In Octave/Matlab, a cell-array is just like a normal array (i.e., a vector), except that its elements can also be strings (which they can't in a normal Octave/Matlab matrix/vector), and you index into them using curly braces instead of square brackets. Specifically, to get the word at index* `i`*, you can use* `vocabList{i}`*. You can also use* `length(vocabList)` *to get the number of words in the vocabulary.*

You should now show the email preprocessing function to an instructor.

4.5 Extracting Features from Emails

You will now implement the feature extraction that converts each email into a vector in $\mathbb{R}^n$. For this exercise, you will be using $n = \#$ words in vocabulary list. Specifically, the feature $x_i \in \{0,1\}$ for an email corresponds to whether the i-th word in the dictionary occurs in the email. That is, $x_i = 1$ if the i-th word is in the email and $x_i = 0$ if the i-th word is not present in the email.
Thus, for a typical email, this feature would look like:

$$\vec{x} = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^n .$$

You should now complete the code in `emailFeatures.m` to generate a feature vector for an email, given the word indices.
Once you have implemented `emailFeatures.m`, the next part of `ex6_spam.m` will run your code on the email sample. You should see that the feature vector had length 1899 and 45 non-zero entries.

You should now show the email feature extraction function to an instructor.

4.6 Training SVM for Spam Classification

After you have completed the feature extraction functions, the next step of `ex6_spam.m` will load a preprocessed training dataset that will be used to train a SVM classifier. `spamTrain.mat` contains 4000 training examples of spam and non-spam email, while `spamTest.mat` contains 1000 test examples. Each original email was processed using the `processEmail` and `emailFeatures` functions and converted into a vector $\vec{x}^{(i)} \in \mathbb{R}^{1899}$.
After loading the dataset, `ex6_spam.m` will proceed to train a SVM to classify between spam ($y = 1$) and non-spam ($y = 0$) emails. Once the training completes, you should see that the classifier gets a training accuracy of about 99.8% and a test accuracy of about 98.5%.

4.7 Top Predictors for Spam

our click remov guarante visit basenumb dollar will price pleas nbsp most lo ga dollarnumb

Figure 25: Top predictors for spam email

To better understand how the spam classifier works, we can inspect the parameters to see which words the classifier thinks are the most predictive of spam. The next step of `ex6_spam.m` finds the parameters with the largest positive values in the classifier and displays the corresponding words (Figure 25). Thus, if an email contains words such as "guarantee", "remove", "dollar", and "price" (the top predictors shown in Figure 25), it is likely to be classified as spam.

4.8 Optional (ungraded) exercise: Try your own emails

Now that you have trained a spam classifier, you can start trying it out on your own emails. In the starter code, we have included two email examples (`emailSample1.txt` and `emailSample2.txt`) and two spam examples(`spamSample1.txt` and `spamSample2.txt`). The last part of `ex6_spam.m` runs the spam classifier over the first spam example and classifies it using the learned SVM. You should now try the other examples we have provided and see if the classifier gets them right. You can also try your own emails by replacing the examples (plain text files) with your own emails.

You do not need to submit any solutions for this optional (ungraded) exercise.

4.9 Optional (ungraded) exercise: Build your own dataset

In this exercise, we provided a preprocessed training set and test set. These datasets were created using the same functions (`processEmail.m` and `emailFeatures.m`) that you now have completed. For this optional (ungraded) exercise, you will build your own dataset using the original emails from the SpamAssassin Public Corpus.

Your task in this optional (ungraded) exercise is to download the original files from the public corpus and extract them. After extracting them, you should run the `processEmail`[5] and `emailFeatures` functions on each email to extract a feature vector from each email. This will allow you to build a dataset `X, y` of examples. You should then randomly divide up the dataset into a training set, a cross validation set and a test set.

While you are building your own dataset, we also encourage you to try building your own vocabulary list (by selecting the high frequency words that occur in the dataset) and adding any additional features that you think might be useful.

Finally, we also suggest trying to use highly optimized SVM toolboxes such as LIBSVM.

You do not need to submit any solutions for this optional (ungraded) exercise.

[5] The original emails will have email headers that you might wish to leave out. We have included code in process Email that will help you remove these headers.

5 MLII5 – Neural Networks – Forward Propagation

The objective of this exercise is to get a deeper understanding of the building blocks and the functioning of a neural network. You will implement a neural network to recognize hand-written digits, using the same dataset as in a previous exercise on logistic regression in the course ML-I. Automated handwritten digit recognition is widely used today - from recognizing zip codes (postal codes) on mail envelopes to recognizing amounts written on bank checks. This exercise will show you how a neural network can be used for this classification task. This exercise only covers the forward-propagation to make predictions for unknown patterns. It uses a pre-trained parameter set. The next exercise covers the back-propagation, which is used for the training of the parameters.

For this exercise you will need the following files:

- `ex3_nn.m` – Octave/Matlab script that will help step you through the exercise
- `ex3data1.mat` – Training set of hand-written digits
- `ex3weights.mat` – Initial weights for the neural network exercise
- `displayData.m` – Function to help visualize the dataset
- `sigmoid.m` - Sigmoid function
- `predict.m`*) – Neural network prediction function

You will have to modify the files marked with *).

Throughout the exercise, you will be using the script `ex3_nn.m`. This script sets up the dataset for the problems and makes calls to functions that you will write. You do not need to modify the script. You are only required to modify functions in other files, by following the instructions in this assignment.

5.1 Dataset

You are given a data set in `ex3data1.mat` that contains 5000 training examples of handwritten digits.[6] The `.mat` format means that that the data has been saved in a native Octave/Matlab matrix format, instead of a text (ASCII) format like a `csv`-file. These matrices can be read directly into your program by using the load command. After loading, matrices of the correct dimensions and values will appear in your program's memory. The matrix will already be named, so you do not need to assign names to them.

```
% Load saved matrices from file
load('ex3data1.mat');
% The matrices X and y will now be in your Octave environment
```

There are 5000 training examples in `ex3data1.mat`, where each training example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is "unrolled" into a 400-dimensional vector. Each of these training examples becomes a single row in our data matrix X. This gives us a 5000 by 400 matrix $\hat{X}$ where every row is a training example for a handwritten digit image.

$$\hat{X} = \begin{bmatrix} -\left(\vec{x}^{(1)}\right)^T - \\ -\left(\vec{x}^{(2)}\right)^T - \\ \vdots \\ -\left(\vec{x}^{(m)}\right)^T - \end{bmatrix}$$

[6] This is a subset of the MNIST handwritten digit dataset (http://yann.lecun.com/exdb/mnist/).

The second part of the training set is a 5000-dimensional vector $\vec{y}$ that contains labels for the training set. To make things more compatible with Octave/Matlab indexing, where there is no zero index, we have mapped the digit zero to the value ten. Therefore, a "0" digit is labeled as "10", while the digits "1" to "9" are labeled as "1" to "9" in their natural order.

5.2 Visualizing the data

You will begin by visualizing a subset of the training set. In Part 1 of `ex3_nn.m`, the code randomly selects 100 rows from $\hat{X}$ and passes those rows to the `displayData` function. This function maps each row to a 20 pixel by 20 pixel grayscale image and displays the images together. We have provided the `displayData` function, and you are encouraged to examine the code to see how it works. After you run this step, you should see an image like Figure 26.

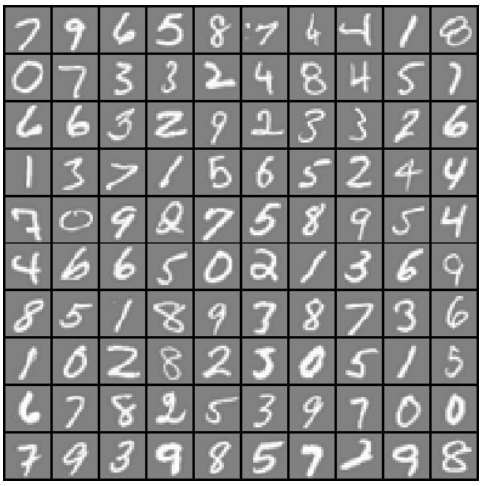

Figure 26: Examples from the dataset

5.3 Model representation

Our neural network is shown in

Figure 27. It has 3 layers – an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20×20, this gives us 400 input layer units (excluding the extra bias unit which always outputs +1). As before, the training data will be loaded into the variables $\hat{X}$ and $\vec{y}$.

You have been provided with a set of network parameters $(\hat{\theta}^{(1)}, \hat{\theta}^{(2)})$ already trained by us. These are stored in `ex3weights.mat` and will be loaded by `ex3_nn.m` into $Theta1$ and $Theta2$. The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes).

```
% Load saved matrices from file
load('ex3weights.mat');
% The matrices Theta1 and Theta2 will now be in your Octave
% environment
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```

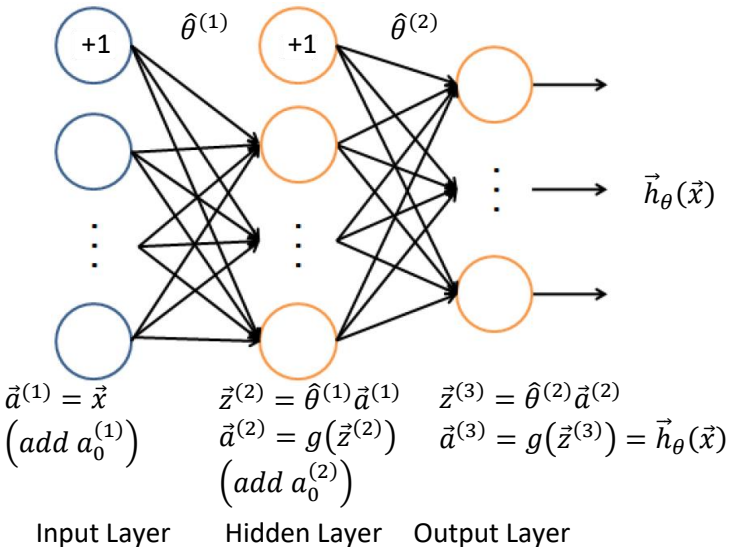

Figure 27: Network Architecture – Forward Propagation

5.4 Feedforward Propagation and Prediction

Now you will implement feedforward propagation for the neural network. You will need to complete the code in `predict.m` to return the neural network's prediction.

You should implement the feedforward computation that computes $h_\theta\left(\vec{x}^{(i)}\right)$ for every example i and returns the associated predictions. Similar to the one-vs-all classification strategy in logistic regression, the prediction from the neural network will be the label that has the largest output $\left(h_\theta(\vec{x})\right)_k$.

Implementation Note: *The matrix $\hat{X}$ contains the examples in rows. When you complete the code in* `predict.m`*, you will need to add the column of 1's to the matrix. The matrices $Theta1$ and $Theta2$ contain the parameters for each unit in rows. Specifically, the first row of $Theta1$ corresponds to the first hidden unit in the second layer. In Octave/Matlab, when you compute $\vec{z}^{(2)} = \hat{\theta}^{(1)}\vec{a}^{(1)}$, be sure that you index (and if necessary, transpose) $\hat{X}$ correctly so that you get $\vec{a}^{(l)}$ as a column vector.*

Once you are done, `ex3_nn.m` will call your predict function using the loaded set of parameters for $Theta1$ and $Theta2$. You should see that the accuracy is about 97.5%. After that, an interactive sequence will launch displaying images from the training set one at a time, while the console prints out the predicted label for the displayed image. To stop the image sequence, press `Ctrl-C`.

You should now show the neural network prediction function to an instructor.

6 MLII6 – Neural Networks – Backward Propagation

The objective of this exercise is to get a deeper understanding of how the training of a neural network is done. By implementing the cost function and the gradient of the cost function using the backpropagation algorithm you should realize that the general training procedure is the same as in the case of linear or logistic regression. The implemented cost function and gradient are used to train a neural network classifier for predicting handwritten characters, using the same dataset as in a previous exercise.

For this exercise you will need the following files:

- `ex4.m` – Octave/Matlab script that will help step you through the exercise
- `ex4data1.mat` – Training set of hand-written digits
- `ex4weights.mat` – Neural network parameters
- `displayData.m` – Function to help visualize the dataset
- `fmincg.m` – Function minimization routine (similar to fminunc)
- `sigmoid.m` – Sigmoid function
- `computeNumericalGradient.m` – Numerically compute gradients
- `checkNNGradients.m` – Function to help check your gradients
- `debugInitializeWeights.m` – Function for initializing weights
- `predict.m` – Neural network prediction function
- `sigmoidGradient.m`*) – Compute the gradient of the sigmoid function
- `randInitializeWeights.m`*) – Randomly initialize weights
- `nnCostFunction.m`*) – Neural network cost function

You will have to modify the files marked with *).

The script `ex4.m` will guide you through the exercise step by step. This script will load necessary datasets and calls the functions you have implemented. The instructions for the implementation of the functions are described in this document.

6.1 Neural Networks

In the previous exercise, you implemented feedforward propagation for neural networks and used it to predict handwritten digits with the weights we provided. In this exercise, you will implement the backpropagation algorithm to learn the parameters for the neural network.

The provided script, ex4.m, will help you step through this exercise.

6.1.1 Visualizing the data

In the first part of `ex4.m`, the code will load the data and display it on a 2-dimensional plot (Figure 28) by calling the function `displayData`.

This is the same dataset that you used in the previous exercise. There are 5000 training examples in `ex4data1.mat`, where each training example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by

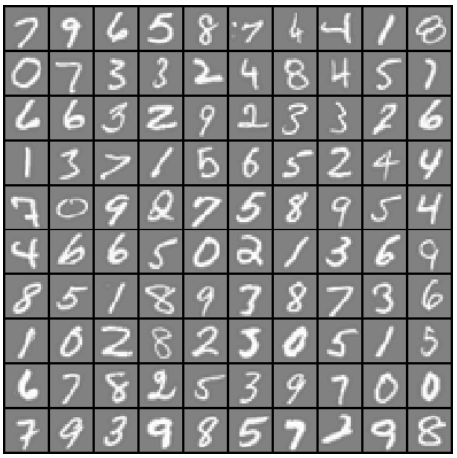

Figure 28: Examples from the dataset

a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is "unrolled" into a 400-dimensional vector. Each of these training examples becomes a single row in our data matrix $\hat{X}$. This gives us a 5000 by 400 matrix $\hat{X}$ where every row is a training example for a handwritten digit image.

$$\hat{X} = \begin{bmatrix} - & \left(\vec{x}^{(1)}\right)^T & - \\ - & \left(\vec{x}^{(2)}\right)^T & - \\ & \vdots & \\ - & \left(\vec{x}^{(m)}\right)^T & - \end{bmatrix}$$

The second part of the training set is a 5000-dimensional vector $\vec{y}$ that contains labels for the training set. To make things more compatible with Octave/Matlab indexing, where there is no zero index, we have mapped the digit zero to the value ten. Therefore, a "0" digit is labeled as "10", while the digits "1" to "9" are labeled as "1" to "9" in their natural order.

6.1.2 Model representation

Our neural network is shown in Figure 29. It has 3 layers – an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20 × 20, this gives us 400 input layer units (not counting the extra bias unit which always outputs +1). The training data will be loaded into the variables `X` and `y` by the `ex4.m` script.

You have been provided with a set of network parameters $\left(\hat{\theta}^{(1)}, \hat{\theta}^{(2)}\right)$ already trained by us. These are stored in `ex4weights.mat` and will be loaded by `ex4.m` into `Theta1` and `Theta2`. The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes).

```
% Load saved matrices from file
load('ex4weights.mat');
% The matrices Theta1 and Theta2 will now be in your workspace
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```

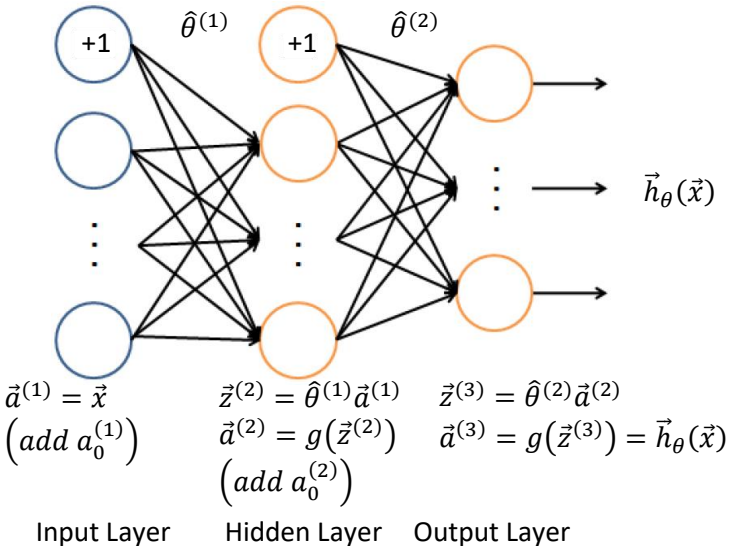

Figure 29: Neural network model.

6.1.3 Feedforward and cost function

Now you will implement the cost function and gradient for the neural network. First, complete the code in `nnCostFunction.m` to return the cost.

Recall that the cost function for the neural network (without regularization) is

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log\left(\left(h_\theta\left(\vec{x}^{(i)}\right)\right)_k\right) - \left(1-y_k^{(i)}\right)\log\left(1-\left(h_\theta\left(\vec{x}^{(i)}\right)\right)_k\right)\right],$$

where $\vec{h}_\theta\left(\vec{x}^{(i)}\right)$ is computed as shown in the Figure 29 and $K = 10$ is the total number of possible labels. Note that $h_\theta\left(\vec{x}^{(i)}\right)_k = a_k^{(3)}$ is the activation (output value) of the $k-th$ output unit. Also, recall that whereas the original labels (in the variable `y`) were 1, 2, ..., 10, for the purpose of training a neural network, we need to recode the labels as vectors containing only values 0 or 1, so that

$$\vec{y} = \begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix}, \quad \begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix}, \quad \ldots or \quad \begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix}.$$

For example, if $\vec{x}^{(i)}$ is an image of the digit 5, then the corresponding $\vec{y}^{(i)}$ (that you should use with the cost function) should be a 10-dimensional vector with $y_5 = 1$, and the other elements equal to 0.

You should implement the feedforward computation that computes $\vec{h}_\theta(\vec{x}^{(i)})$ for every example i and sum the cost over all examples. For implementing the feedforward propagation you might reuse your solution from previous section. However, take care that your code should also work for a dataset of any size, with any number of labels (you can assume that there are always at least $K \geq 3$ labels).

Implementation Note: *The matrix* `X` *contains the examples in rows (i.e.,* `X(i,:)'` *is the* $i-th$ *training example* $\vec{x}^{(i)}$*, expressed as a* $n \times 1$ *vector.) When you complete the code in* `nnCostFunction.m`*, you will need to add the column of 1's to the* `X` *matrix. The parameters for each unit in the neural network is represented in* `Theta1` *and* `Theta2` *as one row. Specifically, the first row of* `Theta1` *corresponds to the first hidden unit in the second layer. You can use a for-loop over the examples to compute the cost.*

Once you are done, `ex4.m` will call your `nnCostFunction` using the loaded set of parameters for `Theta1` and `Theta2`. You should see that the cost is about 0.287629.

You should now show the neural network cost function (feedforward) to an instructor.

6.1.4 Regularized cost function

The cost function for neural networks with regularization is given by

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log\left(\left(h_\theta(\vec{x}^{(i)})\right)_k\right) - \left(1-y_k^{(i)}\right)\log\left(1-\left(h_\theta(\vec{x}^{(i)})\right)_k\right)\right] +$$

$$\frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}\left(\theta_{j,k}^{(1)}\right)^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}\left(\theta_{j,k}^{(2)}\right)^2\right].$$

You can assume that the neural network will only have 3 layers – an input layer, a hidden layer and an output layer. However, your code should work for any number of input units, hidden units and outputs units. While we have explicitly listed the indices above for $\hat{\theta}^{(1)}$ and $\hat{\theta}^{(2)}$ for clarity, do note that **your code should in general work with $\hat{\boldsymbol{\theta}}^{(1)}$ and $\hat{\boldsymbol{\theta}}^{(2)}$ of any size**.

Note that you should not be regularizing the terms that correspond to the bias. For the matrices `Theta1` and `Theta2`, this corresponds to the first column of each matrix. You should now add regularization to your cost function. Notice that you can first compute the unregularized cost function J using your existing `nnCostFunction.m` and then later add the cost for the regularization terms.

Once you are done, `ex4.m` will call your `nnCostFunction` using the loaded set of parameters for `Theta1` and `Theta2`, and $\lambda = 1$. You should see that the cost is about 0.383770.

You should now show the regularized neural network cost function (feed-forward) to an instructor.

6.2 Backpropagation

In this part of the exercise, you will implement the backpropagation algorithm to compute the gradient for the neural network cost function. You will need to complete the `nnCostFunction.m` so that it returns an appropriate value for grad. Once you have computed the gradient, you will be able to train the neural network by minimizing the cost function $J(\Theta)$ using an advanced optimizer such as `fmincg`.

You will first implement the backpropagation algorithm to compute the gradients for the parameters for the (unregularized) neural network. After you have verified that your gradient computation for the unregularized case is correct, you will implement the gradient for the regularized neural network.

6.2.1 Sigmoid gradient

To help you get started with this part of the exercise, you will first implement the sigmoid gradient function. The gradient for the sigmoid function can be computed as

$$g'(z) = \frac{d}{dz} g(z) = g(z)\big(1 - g(z)\big)$$

where

$$sigmoid(z) = g(z) = \frac{1}{1 + e^{-z}}.$$

When you are done, try testing a few values by calling `sigmoidGradient(z)` at the Octave command line. For large values (both positive and negative) of z, the gradient should be close to 0. When $z = 0$, the gradient should be exactly 0.25. Your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid gradient function on every element.

You should now show the sigmoid gradient function to an instructor.

6.2.2 Random initialization

When training neural networks, it is important to randomly initialize the parameters for symmetry breaking. One effective strategy for random initialization is to randomly select values for $\hat{\Theta}^{(1)}$ uniformly in the range $[-\epsilon_{init}, \epsilon_{init}]$. You should use $\epsilon_{init} = 0.12$[7]. This range of values ensures that the parameters are kept small and makes the learning more efficient.

Your job is to complete `randInitializeWeights.m` to initialize the weights for $\hat{\Theta}^{(1)}$. Modify the file and fill in the following code:

```
% Randomly initialize the weights to small values
epsilon_init = 0.12;
W = rand(L_out, 1 + L_in) * 2 * epsilon_init - epsilon_init;
```

You do not need to submit any code for this part of the exercise.

[7] One effective strategy for choosing ϵ_{init} is to base it on the number of units in the network. A good choice of ϵ_{init} is $\epsilon_{init} = \frac{\sqrt{6}}{\sqrt{L_{in} + L_{out}}}$, where $L_{in} = s_l$ and $L_{out} = s_{l+1}$ are the number of units in the layers adjacent to $\Theta^{(1)}$.

6.2.3 Backpropagation

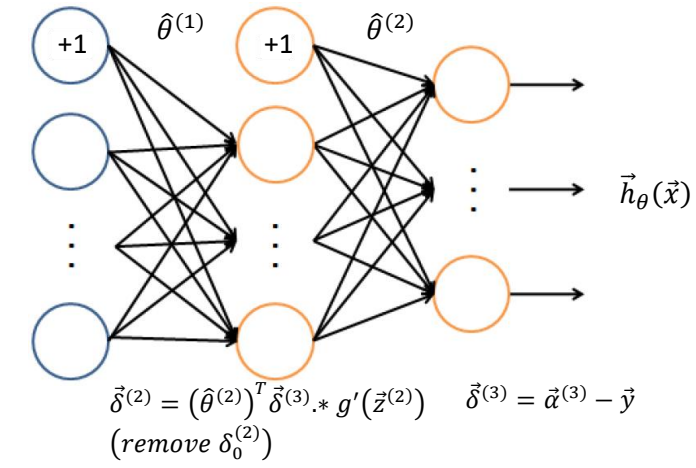

Figure 30: Backpropagation Updates

Now, you will implement the backpropagation algorithm. Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example $(\vec{x}^{(t)}, \vec{y}^{(t)})$, we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $\vec{h}_\theta(\vec{x})$. Then, for each node j in layer l, we would like to compute an "error term" $\delta_j^{(l)}$ that measures how much that node was "responsible" for any errors in our output.

For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_j^{(3)}$ (since layer 3 is the output layer). For the hidden units, you will compute $\delta_j^{(l)}$ based on a weighted average of the error terms of the nodes in layer $(l+1)$.

In detail, here is the backpropagation algorithm (also depicted in Figure 30). You should implement steps 1 to 4 in a loop that processes one example at a time. Concretely, you should implement a for-loop `for t=1:m` and place steps 1-4 below inside the for-loop, with the t^{th} iteration performing the calculation on the t$^{\text{th}}$ training example $(\vec{x}^{(t)}, \vec{y}^{(t)})$. Step 5 will divide the accumulated gradients by m to obtain the gradients for the neural network cost function.

1. Set the input layer's values $(\vec{a}^{(1)})$ to the t-th training example $\vec{x}^{(t)}$. Perform a feedforward pass (Figure 29), computing the activations $(\vec{z}^{(2)}, \vec{a}^{(2)}, \vec{z}^{(3)}, \vec{a}^{(3)})$ for layers 2 and 3. Note that you need to add a +1 term to ensure that the vectors of activations for layers $\vec{a}^{(1)}$ and $\vec{a}^{(2)}$ also include the bias unit. In Octave, a_1 is a column vector, adding one corresponds to `a_1=[1 ; a_1]`.
2. For each output unit k in layer 3 (the output layer), set
$$\delta_k^{(3)} = \left(a_k^{(3)} - y_k\right)$$
where $y_k \in \{0,1\}$ indicates whether the current training example belongs to class k $(y_k = 1)$, or if it belongs to a different class $(y_k = 0)$. You may find logical arrays helpful for this task (explained in the pre-vious programming exercise).
3. For the hidden layer $l = 2$, set

$$\vec{\delta}^{(2)} = \hat{\Theta}^{(2)^T}\vec{\delta}^{(3)} .* g'\left(\vec{z}^{(2)}\right)$$

4. Accumulate the gradient from this example using the following formula. Note that you should skip or remove $\delta_0^{(2)}$. In Octave, removing $\delta_0^{(2)}$ corresponds to delta_2=delta 2(2:end).

$$\hat{\Delta}^{(l)} = \hat{\Delta}^{(l)} + \vec{\delta}^{(l+1)}\left(\vec{a}^{(l)}\right)^T$$

5. Obtain the (unregularized) gradient for the neural network cost function by multiplying the accumulated gradients by $\frac{1}{m}$:

$$\frac{\partial}{\partial\theta_{ij}^{(l)}}J(\theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)}$$

*Octave Tip: You should implement the backpropagation algorithm only after you have successfully completed the feedforward and cost functions. While implementing the backpropagation algorithm, it is often useful to use the size function to print out the **sizes** of the variables you are working with if you run into dimension mismatch errors ("**nonconformant arguments**" errors in Octave).*

After you have implemented the backpropagation algorithm, the script `ex4.m` will proceed to run gradient checking on your implementation. The gradient check will allow you to increase your confidence that your code is computing the gradients correctly.

6.2.4 Gradient checking

In your neural network, you are minimizing the cost function $J(\Theta)$. To perform gradient checking on your parameters, you can imagine "unrolling" the parameters $\hat{\Theta}^{(1)}, \hat{\Theta}^{(2)}$ into a long vector $\vec{\theta}$. By doing so, you can think of the cost function being $J(\theta)$ instead and use the following gradient checking procedure.

Suppose you have a function $f_i\left(\vec{\Theta}\right)$ that purportedly computes $\frac{\partial}{\partial\theta_i}J\left(\vec{\Theta}\right)$; you'd like to check if$f_i$ is outputting correct derivative values.

$$Let \quad \vec{\theta}^{(i+)} = \vec{\theta} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix} \quad and \quad \vec{\theta}^{(i-)} = \vec{\theta} - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}$$

So, $\vec{\theta}^{(i+)}$ is the same as $\vec{\theta}$, except its i-th element has been incremented by ϵ. Similarly, $\vec{\theta}^{(i-)}$ is the corresponding vector with the i-th element decreased by ϵ. You can now numerically verify $f_i\left(\vec{\Theta}\right)$'s correctness by checking, for each i, that:

$$f_i\left(\vec{\Theta}\right) \approx \frac{J\left(\vec{\theta}^{(i+)}\right) - J\left(\vec{\theta}^{(i-)}\right)}{2\,\epsilon}$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\epsilon = 10^{-4}$, you'll usually find that the left- and right-hand sides of the above will agree to at least 4 significant digits (and often many more).

We have implemented the function to compute the numerical gradient for you in `computeNumericalGradient.m`. While you are not required to modify the file, we highly encourage you to take a look at the code to understand how it works.

In the next step of `ex4.m`, it will run the provided function `checkNNGradients.m` which will create a small neural network and dataset that will be used for checking your gradients. If your backpropagation implementation is correct, you should see a relative difference that is less than $1e-9$.

***Practical Tip**: When performing gradient checking, it is much more efficient to use a small neural network with a relatively small number of input units and hidden units, thus having a relatively small number of parameters. Each dimension of Θ requires two evaluations of the cost function and this can be expensive. In the function `checkNNGradients`, our code creates a small random model and dataset which is used with `computeNumericalGradient` for gradient checking. Furthermore, after you are confident that your gradient computations are correct, you should turn off gradient checking before running your learning algorithm.*

***Practical Tip**: Gradient checking works for any function where you are computing the cost and the gradient. Concretely, you can use the same `computeNumericalGradient.m` function to check if your gradient implementations for the other exercises are correct too (e.g., logistic regression's cost function).*

Once your cost function passes the gradient check for the (unregularized) neural network cost function, you should show the neural network gradient function (backpropagation) to an instructor.

6.2.5 Regularized Neural Networks

After you have successfully implemented the backpropagation algorithm, you will add regularization to the gradient. To account for regularization, it turns out that you can add this as an additional term after computing the gradients using backpropagation. Specifically, after you have computed $\Delta_{ij}^{(l)}$ using backpropagation, you should add regularization using

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} \qquad for\ j = 0$$

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\theta_{ij}^{(l)} \qquad for\ j \geq 1$$

Note that you should not be regularizing the first column of $\hat{\theta}^{(l)}$ which is used for the bias term. Furthermore, in the parameters $\theta_{ij}^{(l)}$, i is indexed starting from 1, and j is indexed starting from 0. Thus,

$$\hat{\theta}^{(l)} = \begin{bmatrix} \theta_{1,0}^{(l)} & \theta_{1,1}^{(l)} & \cdots \\ \theta_{2,0}^{(l)} & \theta_{2,1}^{(l)} & \\ \vdots & & \ddots \end{bmatrix}.$$

Somewhat confusingly, indexing in Octave starts from 1 (for both i and j), thus `Theta1(2, 1)` actually corresponds to $\theta_{2,0}^{(l)}$ (i.e., the entry in the second row, first column of the matrix $\hat{\theta}^{(1)}$ shown above).

Now modify your code that computes grad in `nnCostFunction` to account for regularization. After you are done, the `ex4.m` script will proceed to run gradient checking on your implementation. If your code is correct, you should expect to see a relative difference that is less than 1e-9.

You should now show your regularized neural network gradient to an innstructor.

6.2.6 Learning parameters using fmincg

After you have successfully implemented the neural network cost function and gradient computation, the next step of the `ex4.m` script will use `fmincg` to learn a good set parameters.

After the training completes, the `ex4.m` script will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct. If your implementation is correct, you should see a reported training accuracy of about 95.3% (this may vary by about 1% due to the random initialization). It is possible to get higher training accuracies by training the neural network for more iterations. We encourage you to try training the neural network for more iterations (e.g., set `MaxIter` to 400) and also vary the regularization parameter λ. With the right learning settings, it is possible to get the neural network to perfectly fit the training set.

6.3 Visualizing the hidden layer

One way to understand what your neural network is learning is to visualize the representations captured by the hidden units. Informally, given a particular hidden unit, one way to visualize what it computes is to find an input $\vec{x}$ that will cause it to activate (that is, to have an activation value $\left(a_i^{(l)}\right)$ close to 1). For the neural network you trained, notice that the i^{th} row of $\hat{\theta}^{(1)}$ is a 401-dimensional vector that represents the parameter for the i^{th} hidden unit. If we discard the bias term, we get a 400 dimensional vector that represents the weights from each input pixel to the hidden unit.

Thus, one way to visualize the "representation" captured by the hidden unit is to `reshape` this 400 dimensional vector into a 20 × 20 image and display it.[8] The next step of `ex4.m` does this by using the `displayData` function and it will show you an image (similar to Figure 31) with 25 units, each corresponding to one hidden unit in the network.

In your trained network, you should find that the hidden units correspond roughly to detectors that look for strokes and other patterns in the input.

[8] It turns out that this is equivalent to finding the input that gives the highest activation for the hidden unit, given a "norm" constraint on the input $(i.e., \|x\|_2 \leq 1)$.

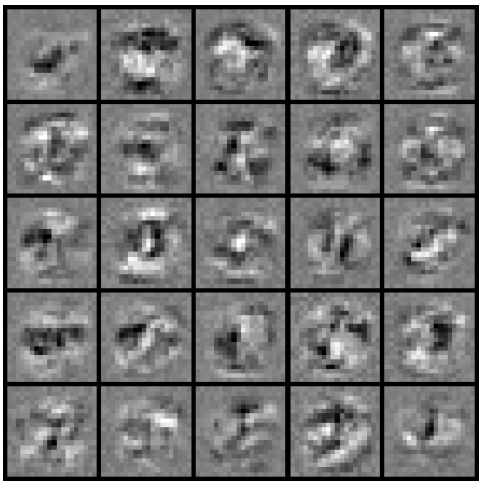

Figure 31: Visualization of Hidden Units.

6.4 Network Performance (optional)

In this part of the exercise, you will get to try out different learning settings for the neural network to see how the performance of the neural network varies with the regularization parameter λ and number of training steps (the `MaxIter` option when using `fmincg`).

Neural networks are very powerful models that can form highly complex decision boundaries. Without regularization, it is possible for a neural network to "overfit" a training set so that it obtains close to 100% accuracy on the training set but does not as well on new examples that it has not seen before. You can set the regularization λ to a smaller value and the `MaxIter` parameter to a higher number of iterations to see this for youself.

You will also be able to see for yourself the changes in the visualizations of the hidden units when you change the learning parameters λ and `MaxIter`.

You do not need to submit any solutions for this optional (ungraded) exercise.

7 MLII7 – Convolutional Neural Network

The objective of this exercise is get a thorough understanding of the fundamental building blocks, the architecture and the functioning of convolutional neural networks (CNN). Furthermore you should be able to explain why CNNs are especially well suited for the classification of images and design and train appropriate networks for this kind of applications. Finally, you should get an impression of the effect on the training time when using GPUs instead of CPUs and be able to assess why the availability of cheap GPUs is a driver for artificial intelligence technologies.

For this exercise you will need the following files:

- `exCNN.m` – Octave/Matlab script that will help step you through the exercise
- `crossCorr1D` – Function to calculate a cross-correlation in one dimension
- `crossCorr2D` – Function to calculate a cross-correlation in one dimension
- `defineONN.m`*) – Function to define the layers of an ordinary neural network
- `defineCNN.m`*) – Function to define the layers of a convolutional neural network
- `DigitTraing3D.mat` – Training set of handwritten digits from the MNIST dataset
- `DigitValidation3D.mat` – Validation set of handwritten digits from the MNIST dataset
- `displayData.m` – Function to help visualize the dataset

You will have to modify the files marked with *) .

The script `exCNN.m` will guide you through the exercise step by step. This script will load necessary datasets and calls the functions you have implemented. The instructions for the implementation of the functions are described in this document.

7.1 One-Dimensional Cross Correlation (Convolution)

Convolutional Neural Networks are named after the mathematical convolution operation that is used to extract spatial features from images. Actually, instead of the convolution the cross-correlation operation is used. The only difference between these operations is a sign prefix, which is totally irrelevant for the purpose of learning features. So in this and the next part we will have an in depth look at the cross-correlation operation, first in one dimension and then in two dimensions. In this part of the exercise, you will implement the cross-correlation in one dimension and examine its behaviour for different kernel or filter functions.

The cross-correlation combines two functions, a signal function $f \in \mathbb{R}$ with a kernel function $g \in \mathbb{R}$.

$$z(x) = (g \star f)(x) = \int_{-\infty}^{\infty} g(x')f(x + x')\,dx'$$

The resulting function $z \in \mathbb{R}$ yields high values at all positions x, where the signal f is similar to the kernel g. So, if g describes a certain characteristic or feature, we can extract this feature from the input signal f and its position. I.e., with a proper chosen g, we can detect positive or negative edges, spikes or any other patterns in f.

In a computer, a function has to be discretized and represented by a **limited** series of values. We will describe the values of the signal function by a vector $\vec{x}$ of size n and the filter by a vector $\vec{\theta}$ of size$f$. The names $\vec{x}$ and $\vec{\theta}$ are wilfully chosen. They express their meaning in neural networks as the input signal (features) and the learnable weights or parameters.

Figure 32 shows an example of a cross-correlation between an input signal $\vec{x}$ with $m = 9$ and a filter $\vec{\theta}$ with $f = 3$. In the discretized form the integral in the cross-correlation becomes a sum and the i-th element of the output vector $\vec{z}$ is calculated by

$$z_i = \sum_{k=1}^{f} x_{k+(i-1)}\,\theta_k = \vec{x}_{i:i+f-1}^{T} \cdot \vec{\theta}$$

In the figure and the formula, two observations are worth to be mentioned: a) The summation just goes over 1 to f. In fact, the sum can be done by a dot product between the filter and a patch of the input signal (with the same size as the filter size) located at position $i + f/2$. b) The size of the output signal is smaller than the size of the input signal ($\vec{z} \in \mathbb{R}^{n-f+1}$). In order to apply the filter to a full patch of the input signal, the center of the first patch lies at $f/2$, whereas the center of the last patch lies at $n - f/2$. This yields to the problem, that the first and the last elements of the input signal are less counted in the cross-correlation than those in the middle are. (In the example, the elements in the middle are considered three times whereas the first one is only considered once.) This problem is addressed by "**zero padding**". We expand the input vector $\vec{x}$ by adding a certain number p of zeros before and after the original values. The dimension of the new, expanded input vector is $n + 2p$. The cross-correlation on the expanded vector can be done in the same way as on the original input vector (see Figure 33). But now, the dimension of the output vector $\vec{z}$ is increased by $2p$ as well, i.e. $\vec{z} \in \mathbb{R}^{n+2p-f+1}$. In principle, the number of padded zeros p can be arbitrary, but often p is chosen so that the output vector has the same size as the original input vector.

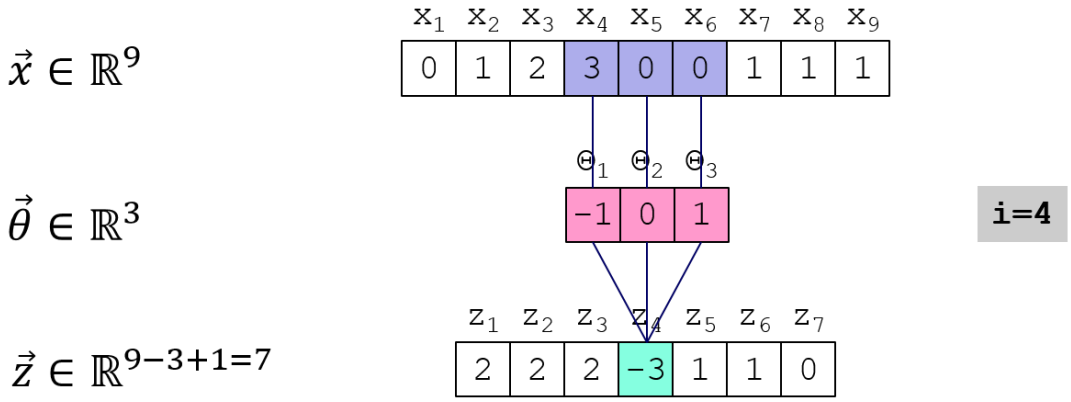

Figure 32: Cross-correlation of two discrete signals in one dimension

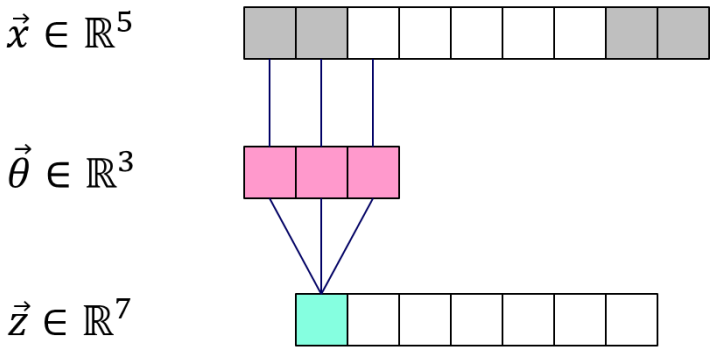

Figure 33: Cross-correlation with Zero-Padding

In normal cross-correlation the kernel is moved over the input vector value by value, i.e. with a step size of one. Sometimes, in CNN´s another step size is used. This is called the **stride** s. When using an arbitrary stride s, the formula for the cross-correlation and the dimension of the output vector change as follows:

$$z_i = \sum_{k=1}^{f} x_{k+s(i-1)}\, \theta_k = \vec{x}^T_{s(i-1)+1:s(i-1)+f} \cdot \vec{\theta}$$

$$\vec{z} \in \mathbb{R}^{(n+2p-f)/s+1}$$

7.1.1 Implement one dimensional cross-correlation

Now you should implement the 1D-cross-correlation in the function `crossCorr1D.m`. The function is provided with the input signal $\vec{x}$, the kernel $\vec{\theta}$, the number p of zeros to be used for padding and the stride s. You should first add p zeros at the beginning and the end of the vector $\vec{x}$. Then determine the size nz of the output vector $\vec{z}$. You can do the calculation of the elements z_i in a `for`-loop. To calculate a single z_i use the dot product (matrix multiplication) for the convolution of the kernel with a patch of the input vector at position i.

The script `exCNN.m` will now call your function with some test data. It should output: `6 -2 -1 -2 -1`.

You should now show your one dimensional cross-correlation to an instructor.

7.1.2 Apply different filters

Now we will apply our `crossCorr1D` function on a test data set to examine the behaviour of different filter functions. Figure 34 shows the test data set. The input signal exhibits certain characteristics like positive and negative edges as well as specific patterns, a sawtooth and a rectangle pattern. By convoluting the input signal with appropriate filter functions, we will try to detect these characteristics.

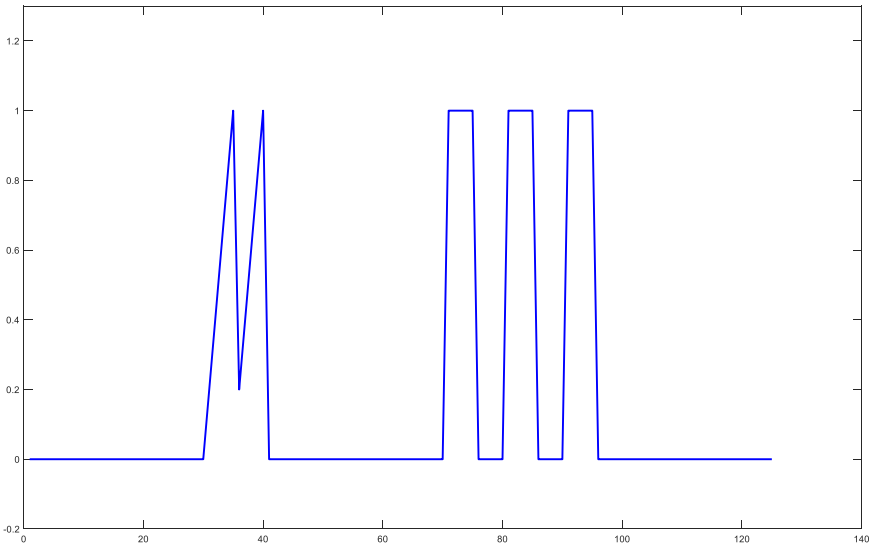

Figure 34: Sample input signal for 1D convolution

7.1.2.1 Edge detection

A kernel with $\vec{\theta}^T = (-1 \quad 0 \quad 1)$ will highlight signal positions x_i where x_{i-1} is small or negative and x_{i+1} is high and positive, i.e. where the signal has a positive edge. Let´s have a look at position 81 in the input signal where we find the values $(0 \quad 1 \quad 1)$, a positive edge, and at the position 84, where we find the values $(1 \quad 1 \quad 1)$, a plateau. The dot product of the kernel at these positions yields:

$$(x_{80} \quad x_{81} \quad x_{82}) \cdot \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix} = (0 \quad 1 \quad 1) \cdot \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} = 0 \cdot -1 + 1 \cdot 0 + 1 \cdot 1 = 1$$

$$(x_{83} \quad x_{84} \quad x_{85}) \cdot \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix} = (1 \quad 1 \quad 1) \cdot \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} = 1 \cdot -1 + 1 \cdot 0 + 1 \cdot 1 = 0$$

In the same way, we can detect negative edges with a kernel $\vec{\theta}^T = (1 \quad 0 \quad -1)$. Figure 35 shows the results of convolving the input signal with the positive edge detection kernel (left) and the negative edge detection kernel (right). The convolution yields the highest results where the edges are most pronounced.

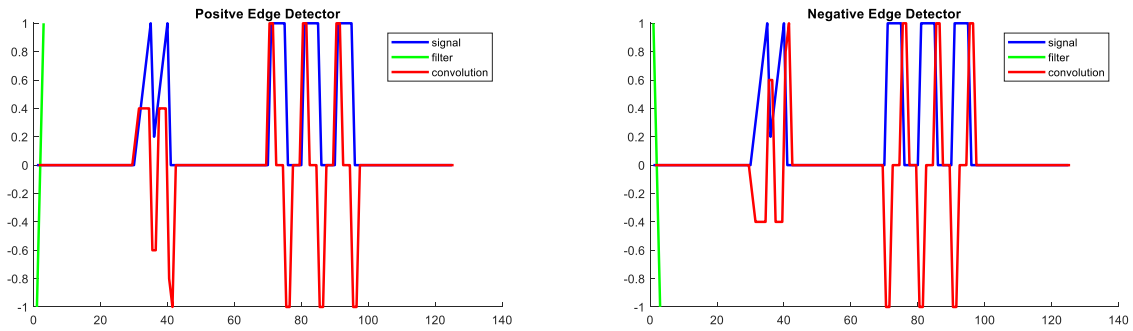

Figure 35: 1D convolution with edge detection kernels

7.1.2.2 Pattern detection

We can also search for very specific patterns in a signal. The cross correlation output is higher the more the input signal matches the kernel pattern. The cross correlation is a measure of similarity of two functions. If our kernel has the shape of a sawtooth and we move it over the input signal, we will get the highest results where the same sawtooth pattern occurs in the input signal. This can be observed in Figure 36 (left). The green curve shows the kernel function and the red curve the outcome of the cross correlation of the blue input signal with the kernel. The right side of Figure 36 depicts the same operation, but now the pattern described by the kernel is a rectangle function. Again, the cross correlation gives the highest peak where the kernel function exactly matches the input signal.

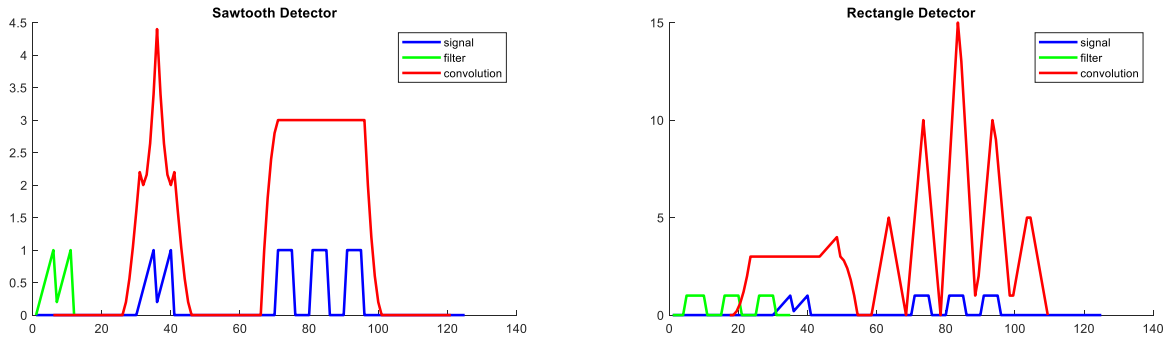

Figure 36: 1D convolution with pattern detection kernels

7.2 Two-Dimensional Cross-Correlation (Convolution)

The 1D cross correlation can easily be enhanced for two dimensions and hence be applied on images. Now in the discrete case, both the input signal and the filter are two-dimensional matrices $\hat{X} \in \mathbb{R}^{n \times n}$ and $\hat{\theta} \in \mathbb{R}^{f \times f}$ respectively. The filter slides over the image column-by-column and line-by-line. At each position a "dot-product" between the filter and the underlying patch of the image is calculated (see Figure 37). A high result means, that a certain characteristic (e.g. an edge), which is described by the filter, is present at this location in the image, a low value means that this characteristic is not present.

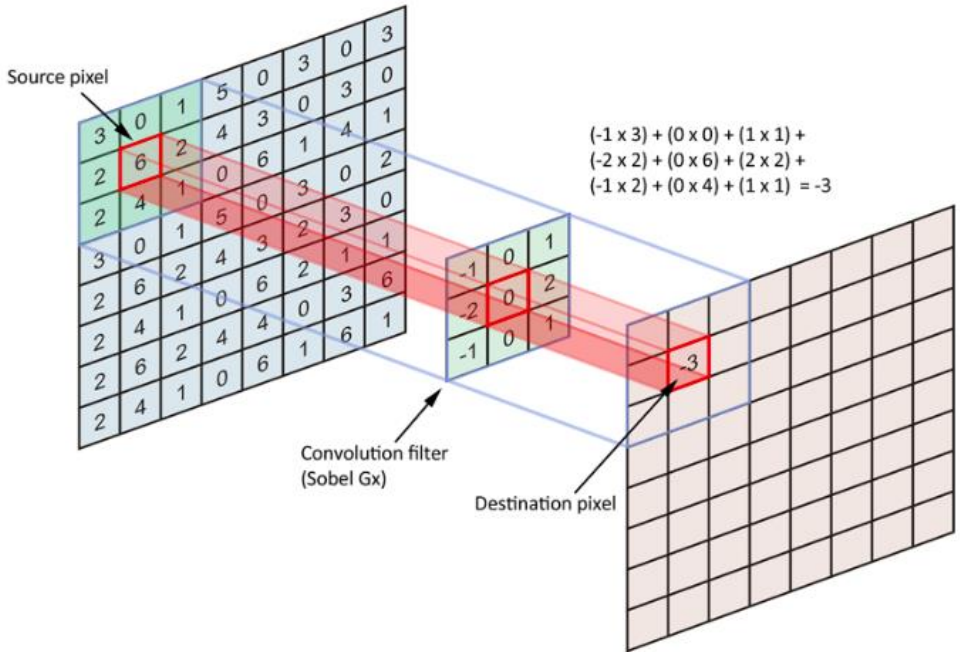

Figure 37: Calculating the cross-correlation in two dimensions

The concepts of "zero-padding" and "stride" can also easily be applied in two dimensions. At each side of the input matrix we can add a number of p zeros and then use this enhanced input matrix $\hat{X}$, having now the dimension $(n + p) \times (n + p)$. Furthermore, we can move the filter both horizontally and vertically with other strides s than 1. The formula for calculating the 2D cross correlation is then given by:

$$\hat{Z} = \hat{X} \star \hat{\theta}: \; z_{i,j} = \sum_{k_1=1}^{f} \sum_{k_2=1}^{f} x_{k_1+s(i-1),k_2+s(j-1)} \, \theta_{k_1,k_2}$$

The dimension of the result matrix $\hat{Z}$ is $(\frac{n+2p-f}{s} + 1) \times (\frac{n+2p-f}{s} + 1)$. It is an abstraction of the original image, describing the image not in terms of pixel values but in terms of more abstract features like e.g. edges, blobs, colours, etc.. The filter or kernel determines the feature that is extracted from the original image.

7.2.1 Implement two dimensional cross-correlation

Now implement the cross correlation for two dimensions in `crossCorr2D.m`. Proceed in the same manner as for the 1D cross correlation. a) Add zeros to all sides of the matrix `X` according the parameter `p`. b) Calculate the dimensions `nz` of the output matrix `Z`. c) Carry out the actual cross correlation. The indexing of the output values `z(i,j)` can be done with two nested `for`-loops. Inside the `for`-loops, perform the "dot-product" between the filter `Theta` and the appropriate patch of the matrix `X` by applying elementwise matrix multiplication and the MATLAB `sum` function.

You should now show your two dimensional cross-correlation to an instructor.

7.2.2 Apply different filters

As in the one-dimensional case, we will examine the behaviour of different filters. For this we use the test image shown in the top left of Figure 38. The convolution operation reveals whether and where certain features are present in the image. The filter parameters determine the kind of the feature.

E.g. with $\hat{\theta}_1 = \begin{pmatrix} -2 & 0 & -2 \\ 0 & 8 & 0 \\ -2 & 0 & -2 \end{pmatrix}$ we can detect edges, independent of their orientation. The result of applying this filter on the test image can be seen in the top right of Figure 38. We can also design more specific filters e.g. for extracting diagonal edges. The filter $\hat{\theta}_2 = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ is doing this for diagonal edges with a positive slope. The result is shown on the bottom left of Figure 38. Accordingly, the filter $\hat{\theta}_3 = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$ extracts top horizontal edges, as shown on the bottom right of Figure 38. The last filter is also well known as the Sobel filter in computer vision. However, for CNN´s the point is, that the filter parameters are not hand designed, they are learnt during the training phase. Thus, the training algorithm determines the features that are relevant for the classification task.

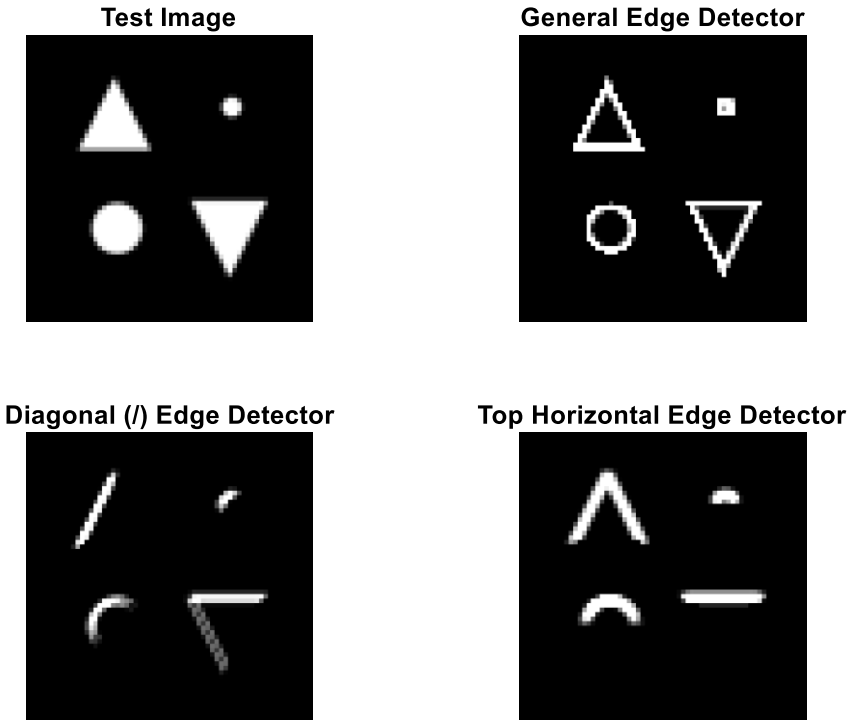

Figure 38: Test image and results of two dimensional convolution with edge detectors

7.3 Image Classification with Ordinary Neural Networks

7.3.1 Visualizing the data

In the remaining parts of this exercise, we will come back to our well-known classification task for handwritten digits. Again, we use a subset of the MNIST[9], but other than in chapter 5 and 6 the images are a little bit larger (28 x 28 pixels) and more important, contain rotated digits. This makes the classification task a bit more complex. One hundred randomly chosen examples are shown in Figure 39. In total, the script `exCNN.m` will load 10000 images and the corresponding labels. This data set will be split into a training set with 7500 examples and a validation set with 2500 examples. In

[9] MNIST handwritten digit dataset (http://yann.lecun.com/exdb/mnist/).

both subsets, the examples are evenly distributed among the 10 classes. Concretely, the training and validation examples are given in the matrices `XTrain3D` and `XValidation3D`, which are of size 28x28x1x7500 and 28x28x1x7500, respectively. The associated label vectors `YTrain` and `YValidation` use the categorical values 1 to 10 to encode the digits 1 to 9 and 0. In this and the next part of the exercise, we build two classifiers with this dataset based on two different models: an ordinary neural network and a convolutional neural network. It will be your task to setup the network architectures, train the models, optimize the hyperparameters and finally compare and discuss the results.

Figure 39: Examples of the MNIST dataset of hand-written digit images

7.3.2 Setting up the network architecture

You will have to setup an ordinary neural network with a nearly identical architecture than we used before in the exercises about neural networks. We will then see how well this network performs on the extended dataset. The network shall have only one hidden layer with 25 neurons. The number of classes determines the number of output neurons (10). For the network implementation, we will follow the way that is given by MATLAB for the definition of deep network architectures. Within the Deep Learning Toolbox, MATLAB provides a variety of already implemented layer types[10]. A network architecture can be defined by instantiating the necessary layer types and put the instances in the desired sequence into an array. For the configuration of the different layer types refer to the MATLAB documentation. In MATLAB Deep Learning Toolbox all deep nets are assumed to process 3D volumes of data. In our case, we start with an input layer (`imageInputLayer`) that takes a 3D volume of size 28x28x1. A deep net in MATLAB must start with at least one input layer. For the

[10] https://de.mathworks.com/help/deeplearning/ug/list-of-deep-learning-layers.html

following layers MATLAB automatically determines the input size by the output size of the preceding layer.

Now define a function `defineONN.m`, which shall return the architecture of a simple ONN with one hidden layer of 25 neurons. The input and output size of the network are configurable by parameters `inputSize` and `numClasses`. The return value is an array of instances of the following layer types:

- `imageInputLayer`: Has to be configured with the input size of the data volume.
- `fullyConnectedLayer`: Configure the number neurons (25).
- `tanhLayer`: A fully connected layer only performs the dot product of the input with the weights. The non-linear activation has to be added separately. Here use the tanh-function.
- `fullyConnectedLayer`: This is the output layer. Configure the number of neurons according the number of classes.
- `softmaxLayer`: For the output layer we take the softmax-function as the non-linear activation. This gives us the probabilities of the classes.
- `classificationLayer`: Finally, this converts the probabilities into a class label.

7.3.3 Tuning the hyperparameter λ

Once the network is setup properly, you can start training it with the MNIST dataset. The script `exCNN.m` will ask you to enter the regularization parameter `lambda`, sets this and other training options and starts the training. You can trace the training progress by observing the learning curves as shown in Figure 40. After the training finished, write down the training and validation accuracy, which are displayed in the Command Window of MATLAB. This procedure will be repeated 6 times. Your task is to find the best value for the hyperparameter λ. Choose λ in the range of 0.01 to 3 with logarithmic step size.

Show your results to an instructor.

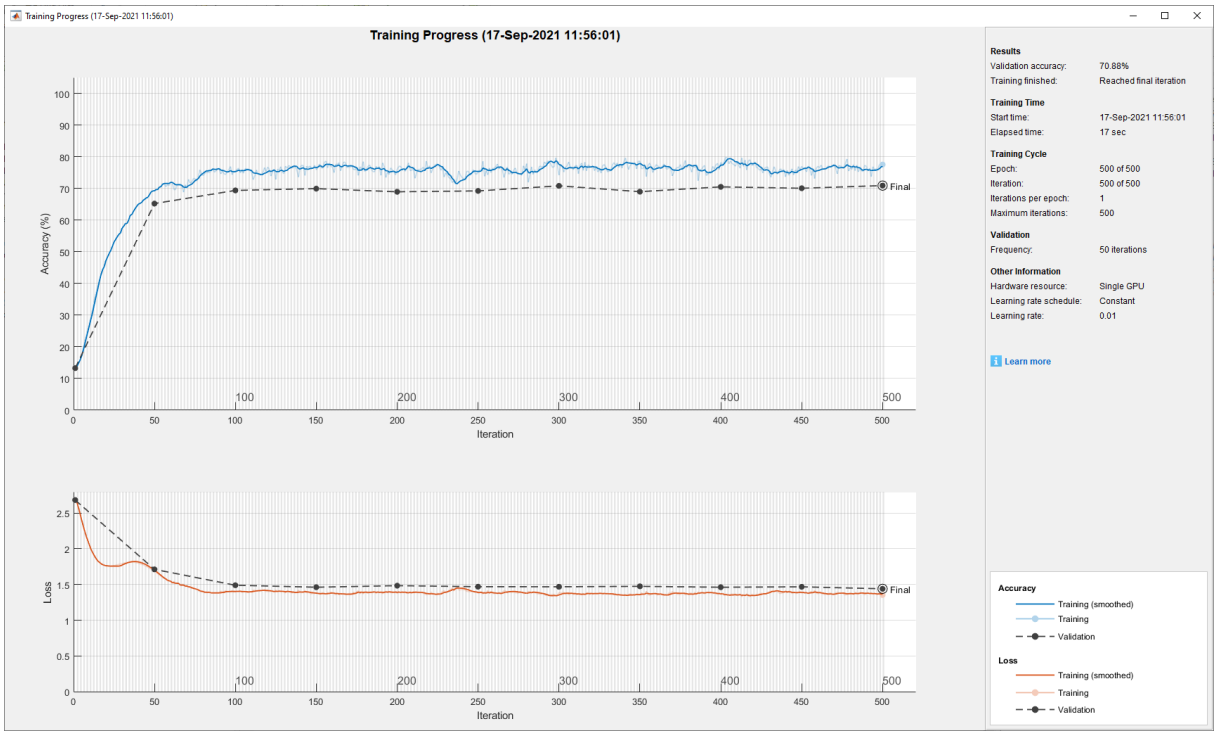

Figure 40: Learning curves of ONN training

7.4 Image Classification with Convolutional Neural Networks

Next, we will do the classification of the MNIST dataset with a convolutional neural network. We will use a very simple architecture. The network consists only of two convolutional layers for extracting relevant features and a single fully connected layer (which is already the output layer) for doing the classification. The architecture is depicted in Figure 41. Since the network is already very shallow, i.e. the number of parameters is small, we will add only little regularization by a dropout layer after the second pooling layer. The dropout rate of this layer is considered as a hyperparameter. It´s your task to the best value of it during the training.

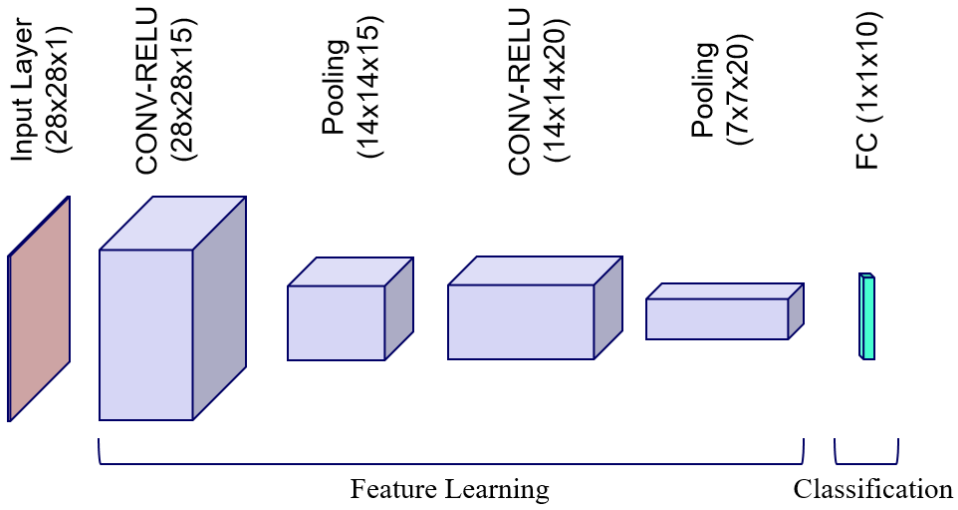

Figure 41: CNN for classification of the MNIST dataset

7.4.1 Setup of the network architecture

To setup the CNN, you will have to implement the function `defineCNN.m`. In the same way as in the previous step for the ordinary neural network, you must define an array of layer instances. Besides the input size and the number of classes, this function defines as an additional parameter for the dropout rate of the second pooling layer. The sequence of layers is as follows:

- `imageInputLayer`: Has to be configured with the input size of the data volume.
- `convolution2dLayer`: The first convolutional layer shall comprise 15 filters of size 3x3 and use a stride of 1 and "same" padding.
- `reluLayer`: The non-linearity has to be added as a separate layer. Use the rectifying linear unit as the activation function.
- `maxPooling2dLayer`: For downsizing the feature map by a factor of two, use a max pooling layer with filter size 2 and stride 2.
- `convolution2dLayer`: The second convolutional layer shall comprise 20 filters of size 3x3 and use a stride of 1 and "same" padding.
- `reluLayer`: Again, use the rectifying linear unit as the activation function.
- `maxPooling2dLayer`: For downsizing the previous feature map by a factor of two, use a max pooling layer with filter size 2 and stride 2.
- `dropoutLayer`: Dropout rates can be set separately for every feature map. Here, configure the dropout rate of the second conv-layer with value given by the parameter `dropoutrate`
- `fullyConnectedLayer`: This is the output layer. Configure the number of neurons according the number of classes.

- `softmaxLayer`: For the output layer we take the softmax-function as non-linear activation. This gives us the probabilities of the classes.
- `classificationLayer`: Finally, this converts the probabilities into a class label.

For the configuration of the layers, refer to the MATLAB documentation.

7.4.2 Training and optimization

Your next task is to train the network with different values of the dropout rate for the second conv-layer. When continuing the script, you will be asked to enter a value for the dropout rate. After each training write down the dropout rate, the training accuracy, the validation accuracy and the time needed for the training. For the six training runs, vary the dropout rate in the range from 0% to 20%.

You can also try to change other hyperparameters and observe the effect on the training as well as the results. Adjust the according `trainingOptions` in the script `exCNN.m`. However, the pre-set values for the initial learning rate (0.0003), the mini batch size (128) and the number of epochs (50) turned out to be most suitable.

Show your results to an instructor.

7.4.3 GPU vs. CPU runtime

In a final step, the script `exCNN.m` asks you to re-enter the dropout rate that yields the best performance. The network will be trained again with this dropout rate - but instead of the GPU the CPU will be used for the training operations. Doing the training on the CPU will lead to a runtime about five times longer than with the GPU. This should give you an impression about the effect of using GPU´s and their importance for large machine learning problems. Note that with the current problem the effect of runtime reduction is very moderate, since the size of the network, the amount of training data and the number of features is too small to really leverage the power a parallel computing.

7.4.4 Predictions

Finally, the script uses the re-trained network to classify consecutive test images. The (intermediate) results of the prediction process is visualized as shown in Figure 42 . The subimages display the output from the four layers (from left to right): 1) the input image, 2) the 15 reduced feature maps of the first convolutional layer, 3) the 20 reduced feature maps of the second convolutional layer and 4) the class probabilities of the fully connected/softmax layer. If you flip through the images, you can observe the good generalization capability of the network even for very distorted and rotated digits.

Determine the number of adaptable parameters in the ONN and the CNN and discuss the observed performance of ONN with CNN against this background in your final report.

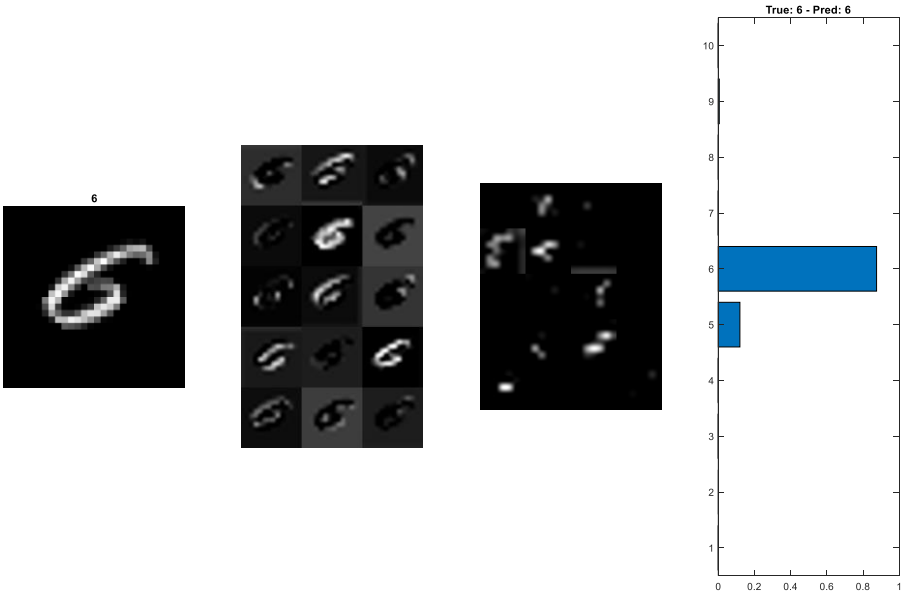

Figure 42: Visualization of the intermediate results of CNN prediction

7.5 References

https://de.mathworks.com/help/deeplearning/examples/create-simple-deep-learning-network-for-classification.html

```
openExample('nnet/ConstructAndTrainAConvolutionalNeuralNetworkExample')
```

http://deeplearning.stanford.edu/tutorial/

8 MLII8 – Object Detection

- Transfer learning
- Tensorflow
-