Oxford Internet Institute, University of Oxford

Assignment Cover Sheet

Candidate Number *Please note, your OSS number is NOT your candidate number*	1033256
Assignment *e.g. Online Social Networks*	Machine Learning
Term *Term assignment issued, e.g. MT or HT*	MT
Title/Question *Provide the full title, or If applicable, note the question number and the FULL question from the assigned list of questions*	An Approximation to the MNIST Dataset
Word Count	3714

By placing a tick in this box ☑ I hereby certify as follows:

(a) This thesis or coursework is entirely my own work, except where acknowledgments of other sources are given. I also confirm that this coursework has not been submitted, wholly or substantially, to another examination at this or any other University or educational institution;

(b) I have read and understood the Education Committee's information and guidance on academic good practice and plagiarism at https://www.ox.ac.uk/students/academic/guidance/skills?wssl=1.

(c) I agree that my work may be checked for plagiarism using Turnitin software and have read the Notice to Candidates which can be seen at: http://www.admin.ox.ac.uk/proctors/turnitin2w.shtml and that I agree to my work being screened and used as explained in that Notice;

(d) I have clearly indicated (with appropriate references) the presence of all material I have paraphrased, quoted or used from other sources, including any diagrams, charts, tables or graphs.

(e) I have acknowledged appropriately any assistance I have received in addition to that provided by my [tutor/supervisor/adviser].

(f) I have not sought assistance from a professional agency;

(g) I understand that any false claims for this work will be reported to the Proctors and may be penalized in accordance with the University regulations.

Please remember:

- To attach a second relevant cover sheet if you have a disability such as dyslexia or dyspraxia. These are available from the Higher Degrees Office, but the Disability Advisory Service will be able to guide you.

Machine Learning Summative Assessment: An Approximation to the MNIST Dataset

Candidate Number: 1033256

January 12, 2019

We start downloading the MNIST dataset. This modified version of the NIST dataset was famously used to train the LeNet-5, a pioneering Convolutional Neural Network that is, in part, responsible for the popularity of CNNs during the last years. It contains 6000 training and 1000 testing images of labeled handwritten digits. Each greyscale image is composed of 28x28 pixels. Pixel values are 0 to 255, but we will scale them dividing by 255 so they are 0 to 1. This scaling operation tends to speed up and stabilize model optimization.

```
In [1]: import numpy as np
        import pandas as pd
        from matplotlib import pyplot as plt
        %matplotlib
        import tensorflow as tf


        # Loading data
        mnist = tf.keras.datasets.mnist
        (x_train, y_train),(x_test, y_test) = mnist.load_data()

        # Scaling
        x_train, x_test = x_train / 255.0, x_test / 255.0

        plt.imshow(X_train[:,i].reshape(28,28), cmap = matplotlib.cm.binary)
        plt.axis("off")
        plt.show()
        print(y_train[:,i])
```

Right now, each image consists of a 28 by 28 Numpy array. Initially, it would be helpful to reshape the data to have everything in one dimension – that is, the 784 pixels as a vector for each digit.

```
In [2]: train_x_r = np.reshape(x_train,
                               (x_train.shape[0],
```

```
                              x_train.shape[1] * x_train.shape[2]))
test_x_r = np.reshape(x_test,
                      (x_test.shape[0],
                       x_test.shape[1] * x_test.shape[2]))
```

1 Unsupervised learning

In this first section we will perform PCA on the MNIST dataset. We will start with a two component PCA.

In [3]:
```
from sklearn import decomposition

mod = decomposition.PCA(n_components=2)
mod.fit(train_x_r)

print(mod.explained_variance_ratio_)
print(mod.explained_variance_ratio_.sum())
```

```
[0.09704664 0.07095924]
0.16800588410864079
```

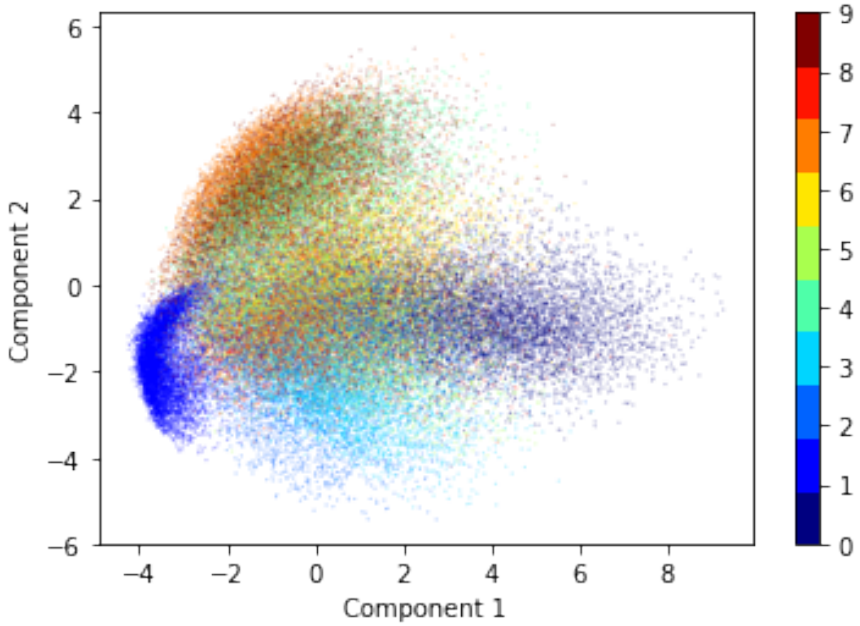
The first component explains almost 10% of the variance, and the second one explains an additional 7%. In total, the model is therefore explaining 16% of the variance. Although these numbers may intuitively seem quite low, it is surprising that we can explain 16% of the variance of a dataset with 784 features with just two components.

Below we include a scatter plot that shows how these two components reflect, in some way, the label in the images. For instance, pictures labeled as 0 have a score close to -4 in Component 1 and around -2 in Component 2. However, it is clear that just two components doesn't allow to distinguish clearly between labels in most cases.

In [4]:
```
train_x_r_t = mod.transform(train_x_r)

# Scatterplot colored by label
plt.scatter(train_x_r_t[:,0],
            train_x_r_t[:,1],
            c = list(y_train),
            cmap=plt.get_cmap('jet', 10), s = 0.01)

plt.xlabel("Component 1")
plt.ylabel("Component 2")
```

```
        plt.colorbar()
```

```
Out[4]: <matplotlib.colorbar.Colorbar at 0x1f60015aa58>
```



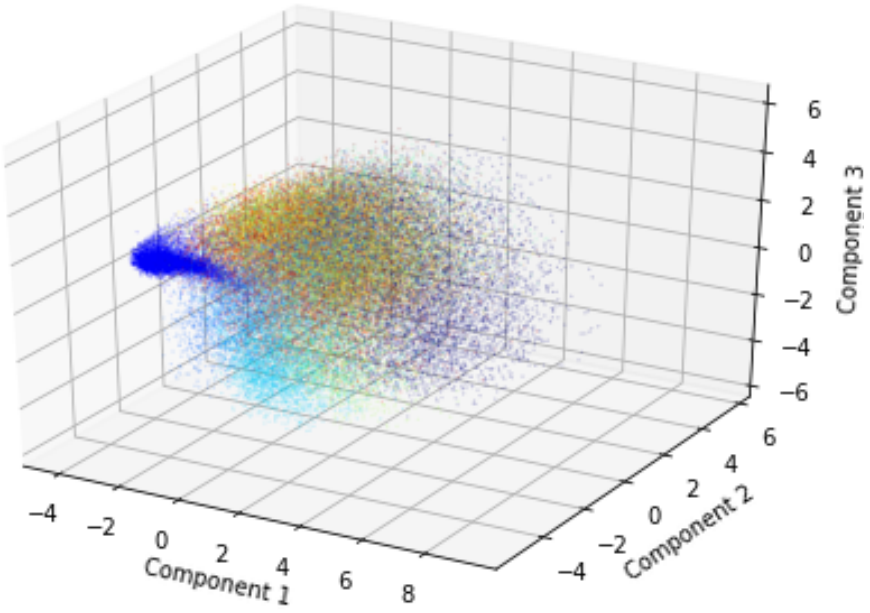
We can also try performing a PCA with three components.

```
In [5]: mod = decomposition.PCA(n_components=3)
          mod.fit(train_x_r)

          print(mod.explained_variance_ratio_)
          print(mod.explained_variance_ratio_.sum())
```

```
[0.09704664 0.07095924 0.06169089]
0.229696769254128
```

```
In [6]: from mpl_toolkits.mplot3d import Axes3D

          train_x_r_t = mod.transform(train_x_r)

          # Same scatterplot as above, but with three dimensions.
          fig = plt.figure()
          ax = Axes3D(fig)
```

```
        ax.scatter(train_x_r_t[:,0],
                   train_x_r_t[:,1],
                   train_x_r_t[:,2],
                   cmap=plt.get_cmap('jet', 10),
                   c = list(y_train), s = 0.01)
        ax.set_xlabel('Component 1')
        ax.set_ylabel('Component 2')
        ax.set_zlabel('Component 3')
        plt.show()
```



We are now explaining 23% of the variance. Compared to the previous plot, it may be easier to distinguish how images with similar labels tend to have similar values in the three components.

```
In [7]: mod = decomposition.PCA(n_components=100)
        mod.fit(train_x_r)
```

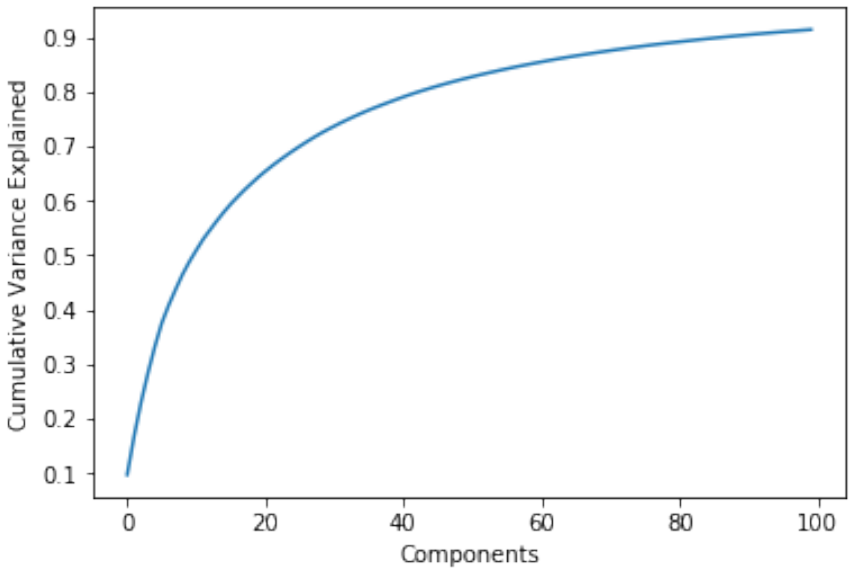
```
Out[7]: PCA(copy=True, iterated_power='auto', n_components=100, random_state=None,
          svd_solver='auto', tol=0.0, whiten=False)
```

Lastly, we will explore how much variance is explained by a different number of components. In order to do that, we will perform a PCA with 784 components, plotting the cumulative variance explained by models with an increasing number of components.

```
In [8]: # Plot cumulative variance explained by component
        plt.plot(mod.explained_variance_ratio_.cumsum())
```

```
plt.xlabel("Components")
plt.ylabel("Cumulative Variance Explained")
```

```
Out[8]: Text(0, 0.5, 'Cumulative Variance Explained')
```



It is obvious that models with more components explain more variance in our data. In fact, just 50 components explain around 80% of the variance.

Note that using much more than these 50 will not make much sense, as the variance explained increase will be much smaller for each component we add.

This is, therefore, much better that the previous PCA models we trained with just two and three components. In this case, we only explained a small part of the total variance, and therefore threw away a lot of the signal. However, note that using much more than these 50 will not make much sense, as the variance explained increase will be much smaller for each component we add. Additionally, take into account that using too many components does not just unnecessarily add more complexity without a clear advantage in variance explained, but it may reintroduce noise that was otherwise removed – preventing overfitting if we later fit a model with the PCA results. That is, it is possible that almost all the signal was explained with the previous components, and adding more just brings noise that with less components will be keep apart.

2 Support Vector Machines

Support Vector Machines (SVM) are a set of supervised learning models that are highly popular for their good accuracy and efficiency. It has been used in face and speech recognition, text catego-

rization, and fraud detection. In the first section, we will introduce the basics of the SVM and give an intuition about the kernel trick, which allows these models to perform non-linear classification very efficiently. In the second section, we will implement SVM in Python.

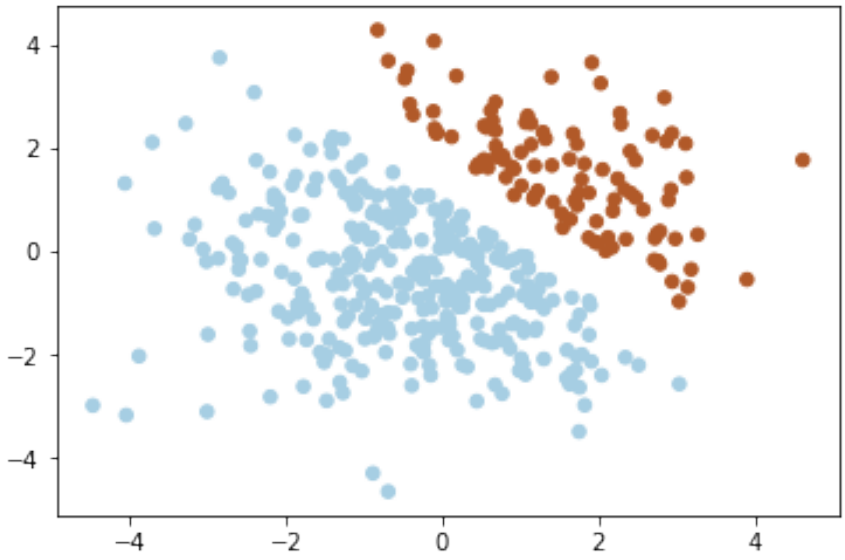
2.1 The Intuition behind the Kernel Trick

2.1.1 An Introduction to Support Vector Machines

The most fundamental idea behind SVM is that they are used to create decision boundary that maximizes the separating margin between separable data. We have anticipated that this method can be used for non-linearly separable data, but we will start understanding how this works with linearly separable classes.

We will start considering the simple case of constructing a decision boundary between two linearly separable classes. First, we generate the random data.

```
In [9]: from sklearn import svm

        X1 = np.random.normal(0,1.5,size = 500)
        X2 = np.random.normal(0,1.5,size = 500)
        y = X1 + X2 > 2

        condition = X1 + X2 < 1

        X1 = X1[(y + condition)]
        X2 = X2[(y + condition)]
        y = y[(y + condition)]

        X = np.column_stack((X1,X2))

        plt.scatter(X[:, 0], X[:, 1], c=y, s=30,
                    cmap=plt.cm.Paired)

        plt.show()
```



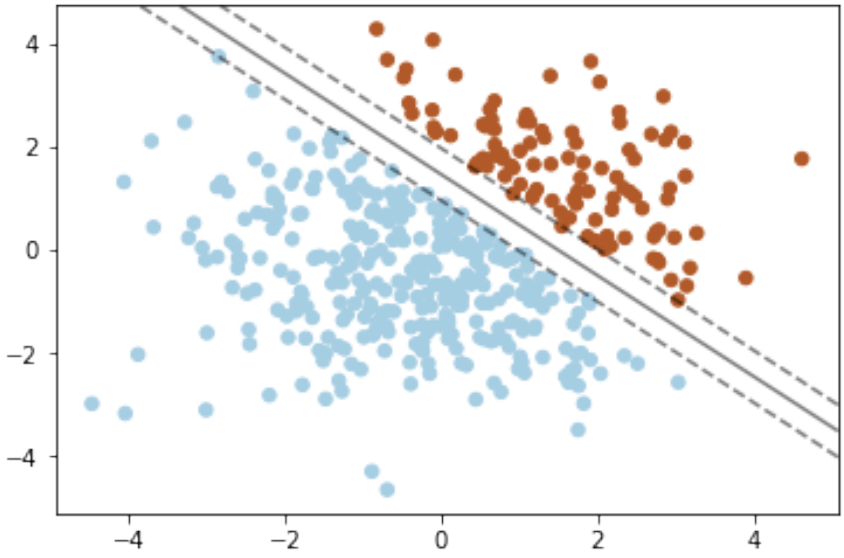
Although there are many possible separators, it is intuitive to consider that the best one will be the decision boundary that maximizes its margin between the two classes. This is called the *maximal margin classifier*, and it is obtained considering:

$$\begin{aligned} &\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{maximize}} \quad M \\ &\text{subject to} \quad \sum_{j=1}^{p} \beta_j^2 = 1, \\ &\qquad\qquad y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) \geq M \;\; \forall\, i = 1, \ldots, n. \end{aligned}$$

That is, we want to create an hyperplane defined by $\beta_0, \beta_1, \ldots, \beta_p$ that maximizes the margin M such as each observation i is in the correct side of the decision boundary and at least a distance M from the hyperplane.

```
In [10]: from sklearn import svm

         clf = svm.SVC(kernel='linear', C=1000)
         clf.fit(X, y)
         plt.scatter(X[:, 0], X[:, 1], c=y, s=30,
                     cmap=plt.cm.Paired)

         ax = plt.gca()
         xlim = ax.get_xlim()
         ylim = ax.get_ylim()

         xx = np.linspace(xlim[0], xlim[1], 30)
         yy = np.linspace(ylim[0], ylim[1], 30)
```

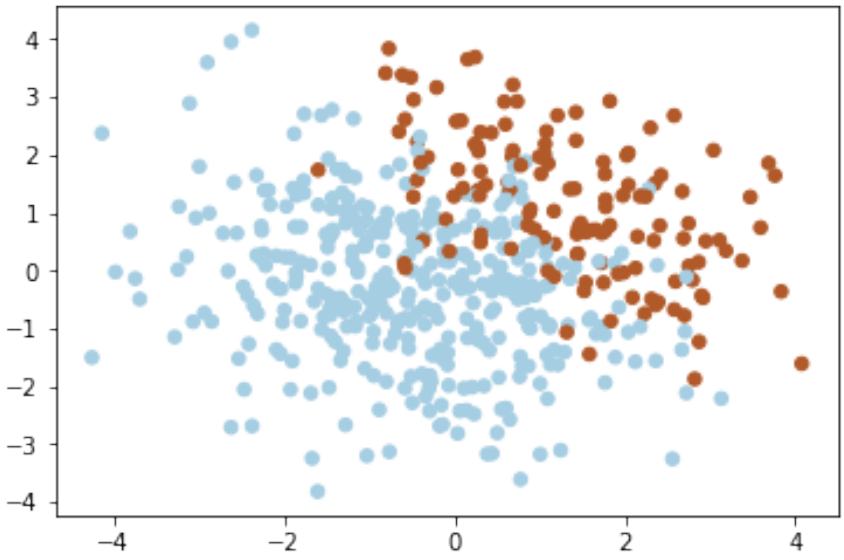
```
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = clf.decision_function(xy).reshape(XX.shape)

ax.contour(XX, YY, Z, colors='k',
           levels=[-1, 0, 1], alpha=0.5,
           linestyles=['--', '-', '--'])
plt.show()
```



A more complex case involves data that cannot be separated perfectly by an hyperplane.

```
In [11]: X1 = np.random.normal(0,1.5,size = 500)
         X2 = np.random.normal(0,1.5,size = 500)
         X = np.column_stack((X1,X2))
         y = np.round((X1+X2)+np.random.normal(0,1,size = 500))>1.5

         plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)
```

```
Out[11]: <matplotlib.collections.PathCollection at 0x1eafe184438>
```



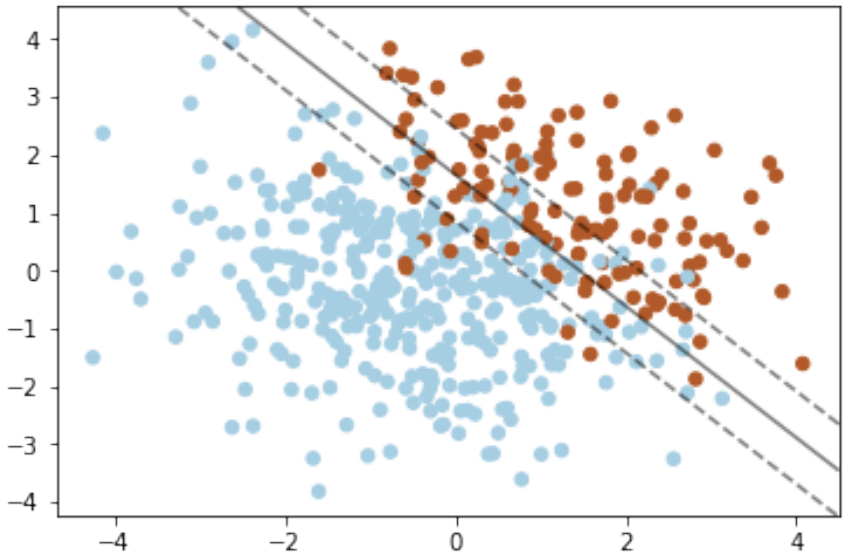
In this case, we can extend the previous model to allow some observations to be inside or even on the wrong side of the margin. This extension is called the *support vector classifier*, and fulfills:

$$
\begin{aligned}
&\underset{\beta_0,\beta_1,\ldots,\beta_p,\epsilon_1,\ldots,\epsilon_n}{\text{maximize}} && M \\
&\text{subject to} && \sum_{j=1}^{p} \beta_j^2 = 1, \\
& && y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) \geq M(1-\epsilon_i) \;\; \forall\, i = 1,\ldots,n, \\
& && \epsilon_i \geq 0, \\
& && \sum_{i=1}^{n} \epsilon_i \leq C.
\end{aligned}
$$

In this formula, ϵ_i indicates if the variable i is on the correct side of the margin (i.e. $\epsilon_i = 0$), on the wrong one (i.e. $0 < \epsilon_i < 1$), or on the wrong side of the hyperplane ($\epsilon_i > 1$). The tuning parameter C determines the severity and number of the violations we will tolerate, and it's usually determined using cross validation.

```
In [12]: clf = svm.SVC(kernel='linear', C=1000)
         clf.fit(X, y)
         plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

         ax = plt.gca()
         xlim = ax.get_xlim()
         ylim = ax.get_ylim()

         xx = np.linspace(xlim[0], xlim[1], 30)
```

```
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = clf.decision_function(xy).reshape(XX.shape)

ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
           linestyles=['--', '-', '--'])
plt.show()
```



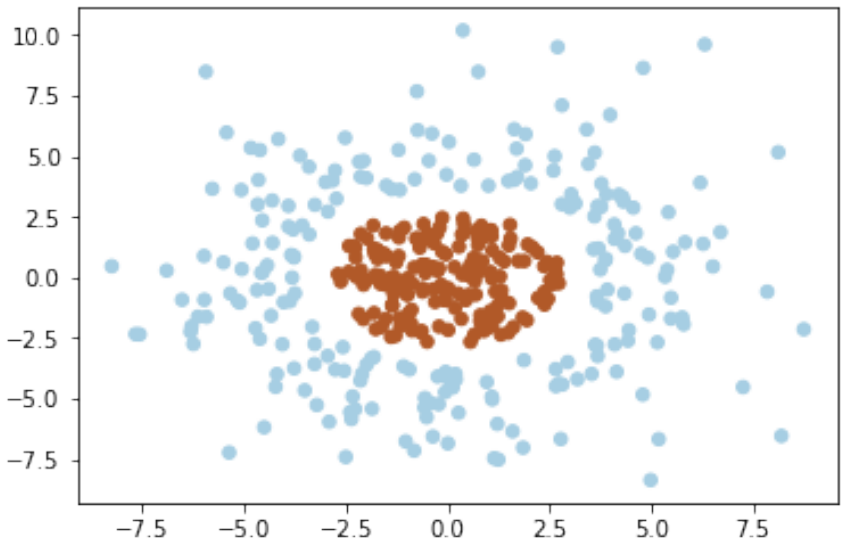
2.1.2 Extending to Non-linearly Separable Data: The Kernel Trick

The final and most interesting case is when the data cannot be separated using a linear decision boundary.

```
In [13]: X1 = np.random.normal(0,3,size = 500)
         X2 = np.random.normal(0,3,size = 500)

         y = (X1**2+X2**2) < 8

         condition = X1**2 + X2**2 > 14

         X1 = X1[(y + condition)]
         X2 = X2[(y + condition)]
         y = y[(y + condition)]
```

```
X = np.column_stack((X1,X2))

plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)
```

```
Out[13]: <matplotlib.collections.PathCollection at 0x1f600d32400>
```

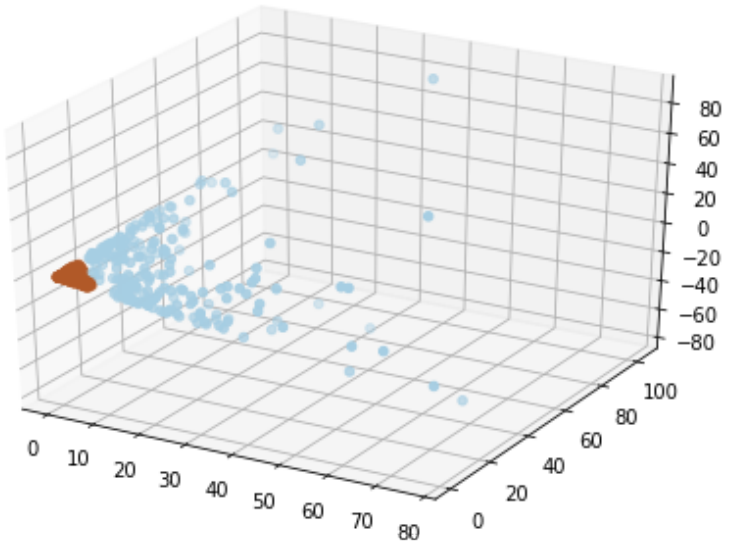


In this case, we will convert the linear classifier described above into one that serves to produce a non-linear separator.

An intuitive approach to how that is possible starts addressing how non-linearly separable data can be linearly separable in a higher dimensional space. In our case we could define a function $\phi : \mathbb{R}^2 \to \mathbb{R}^3$, for example $\phi(x_1, x_2) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$, that mapping our data into a higher dimensional space will allow us to define a linear hyperplane that can separate the data in this higher dimensional space. This linear separator can be projected to our previous lower dimensional space as a a non-linear classifier.

```
In [14]: def phi(X1,X2):
             a = X1**2
             b = X2**2
             c = np.sqrt(2)*X1*X2
             return a,b,c

         from mpl_toolkits.mplot3d import Axes3D

         a,b,c = phi(X1,X2)
         fig = plt.figure()
         ax = Axes3D(fig)
         ax.scatter(a,b,c, c = y, cmap=plt.cm.Paired)
```

```
Out[14]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1f600d40f28>
```



In summary, what we have done is use a function ϕ to transform our data into a 3-D space and calculate the hyperplane that separates the data in this higher dimensional space. Importantly, the computation used to obtain this separation boundary in ϕ space only requires calculating the inner product. For instance, if we have two vectors $\underline{x}$ and $\underline{y}$, obtaining the decision boundary in ϕ space will require:

$$\begin{aligned} \langle \phi(\underline{x}), \phi(\underline{y}) \rangle &= \langle \phi(x_1, x_2), \phi(y_1, y_2) \rangle \\ &= \langle (x_1^2, x_2^2, \sqrt{2}x_1x_2), (y_1^2, y_2^2, \sqrt{2}y_1y_2) \rangle \\ &= x_1^2y_1^2 + x_2^2y_2^2 + 2x_1x_2y_1y_2 \end{aligned} \tag{1}$$

This is clearly computationally expensive, especially considering that in many cases we have to calculate the dot product between thousands of data points, which will imply transforming each of them into the ϕ space and then computing the dot product. Furthermore, it is common that the hyperplane that is able to separate the conditions is not possibly found in just the third dimension, and therefore the problem requires mapping each point into a much higher dimensional space. In this context, finding a solution that allows to compute the inner product efficiently is the only way in which SVM could be applied to problems with more than a handful of data points that are non-linearly separable. This solution is called the kernel trick.

The kernel trick implies that it is not necessary to transform two vectors into the ϕ space in order to compute the dot product in that space. We can find a kernel function $k(\underline{x}, \underline{y}) = \langle \phi(\underline{x}), \phi(\underline{y}) \rangle$ that does not need to compute $\phi(\underline{x})$ nor $\phi(\underline{y})$. For the previously defined ϕ, the kernel function will be:

$$
\begin{aligned}
k(\underline{x}, \underline{y}) &= \langle x, y \rangle^2 \\
&= \langle (x_1, x_2), (y_1, y_2) \rangle^2 \\
&= (x_1 y_1 + x_2 y_2)^2 \\
&= x_1^2 y_1^2 + x_2^2 y_2^2 + 2 x_1 x_2 y_1 y_2
\end{aligned}
\tag{2}
$$

Suprisingly as it may seem, we have obtained the dot product of $\underline{x}$ and $\underline{y}$ in the 3D without explicitly visiting this space.

There are kernels with different properties that allow to obtain the dot product in 4, 5, 6, and more dimensional spaces. Furthermore, the radial basis function kernel $k(\mathbf{x}, \mathbf{y}) = \exp\big(-\|\underline{x} - \underline{y}\|^2/(2\sigma^2)\big)$ corresponds to calculating the dot product in a infinite-dimensional space, something that for obvious reasons would not be possible to calculate if we needed to visit that space.

In sum, with the kernel truck we can find a linear boundary in the higher dimensional space without requiring to visit that space, making this operation computationally feasible.

2.2 SVMs in Python

We have mentioned that SVM is a commonly used technique due to its high accuracy and flexibility to separate linearly and non-linearly separable data. However, the basic algorithm with RBF kernel approaches $O(n^3)$ with a large C, although it converges faster using linear models. There are different solutions for this, but for the sake of simplicity we will use a subset of the training dataset.

```
In [15]: # Randomly subset 10000 training examples
         selector = np.random.randint(train_x_r.shape[0], size=10000)
         train_x_r_subset = train_x_r[selector, :]
         y_train_subset = y_train[selector]
```

First, we will fit four models with four different kernels.

- Linear kernel: $\langle x, y \rangle$
- Polynomial kernel: $(\gamma \langle x, y \rangle + r)^d$
- Sigmoid kernel: $\tanh(\gamma \langle x, y \rangle + r)$
- Radial Basis Function: $\exp(-\gamma \|x - y\|^2)$

The first kernel is a linear one, which is simpler, but also faster. RBF is, in turn, the most complex of the four, but it tends to perform better than the others. For now, we will leave the default hyperparameters $C = 1$ and γ.

```
In [16]: from sklearn import svm

         # Initiate the four models
         linearmodel = svm.SVC(kernel='linear', gamma = 'auto')
         polymodel = svm.SVC(kernel='poly', degree=2, gamma = 'auto')
         sigmoidkernel = svm.SVC(kernel='sigmoid', gamma = 'auto')
         expmodel = svm.SVC(kernel='rbf', gamma = 'auto')

         # Train them
         linearmodel.fit(train_x_r_subset, y_train_subset)
         polymodel.fit(train_x_r_subset, y_train_subset)
         sigmoidkernel.fit(train_x_r_subset, y_train_subset)
         expmodel.fit(train_x_r_subset, y_train_subset)
```

```
Out[16]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False)
```

```
In [17]: from sklearn.metrics import roc_auc_score

         # Get area under the curve for each of the models

         print(roc_auc_score(pd.get_dummies(y_train_subset),
                             pd.get_dummies(linearmodel.predict(train_x_r_subset))))
         print(roc_auc_score(pd.get_dummies(y_train_subset),
                             pd.get_dummies(polymodel.predict(train_x_r_subset))))
         print(roc_auc_score(pd.get_dummies(y_train_subset),
                             pd.get_dummies(sigmoidkernel.predict(train_x_r_subset))))
         print(roc_auc_score(pd.get_dummies(y_train_subset),
                             pd.get_dummies(expmodel.predict(train_x_r_subset))))
```

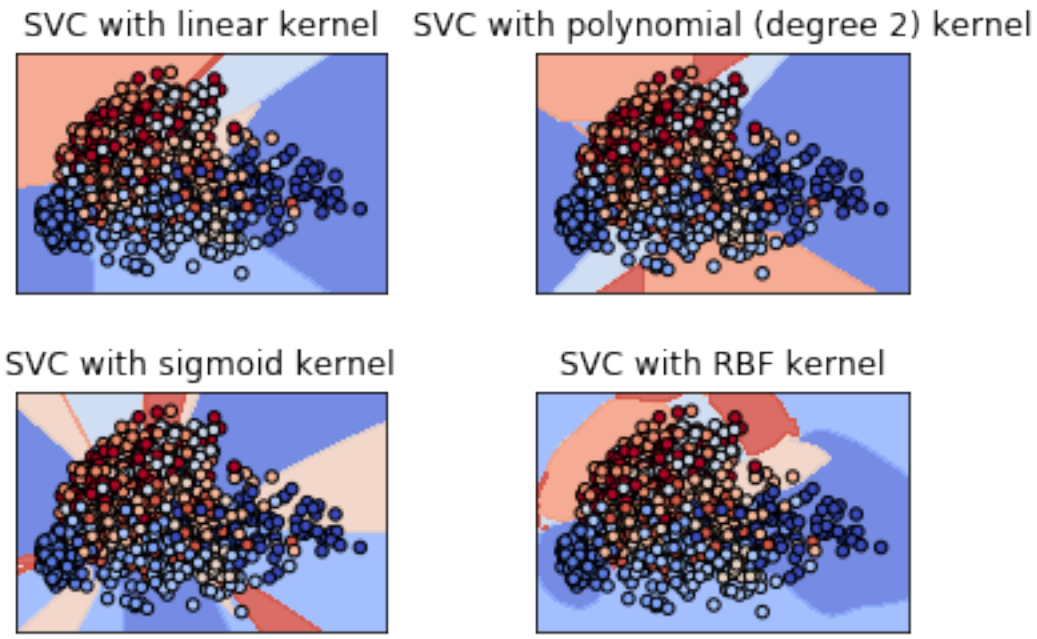
```
0.954208592263859
0.9586914528291149
0.9547996640383098
0.9682198311718968
```

The model that used the RBF kernel performed better than the others. To visualize the results we can perform a PCA to convert our data to just two dimensions and then train a SVM. This allows us to plot the decision boundaries in a two-dimensional plot in which the data points are also visible. However, note that this SVM is not the same as the one we presented above, as we are training it on the two components that result from the PCA.

```
In [18]: # These following plots are inspired by this tutorial:
# https://scikit-learn.org/stable/auto_examples/svm/plot_iris.html
```

```
# We take a sample of just 1000 data points
        selector = np.random.randint(train_x_r.shape[0], size=1000)
        train_x_r_subset = train_x_r[selector, :]
        y_train_subset = y_train[selector]

        # Perform PCA on the subset, with just two components
        # so we can plot the points in a 2D scatterplot

        mod = decomposition.PCA(n_components=2)
        mod.fit(train_x_r_subset)
        train_x_r_subset_t = mod.transform(train_x_r_subset)

        # We then fit the models

        models = (svm.SVC(kernel='linear', gamma = 'auto'),
                  svm.SVC(kernel='poly', degree=2, gamma = 'auto'),
                  svm.SVC(kernel='sigmoid', gamma = 'auto'),
                  svm.SVC(kernel='rbf', gamma = 'auto'))

        models = (clf.fit(train_x_r_subset_t, y_train_subset) for clf in models)

        # And lastly plot the results with the decision boundaries

        def make_meshgrid(x, y, h=.02):
            x_min, x_max = x.min() - 1, x.max() + 1
            y_min, y_max = y.min() - 1, y.max() + 1
            xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                                 np.arange(y_min, y_max, h))
            return xx, yy


        def plot_contours(ax, clf, xx, yy, **params):
            Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)
            out = ax.contourf(xx, yy, Z, **params)
            return out

        titles = ('SVC with linear kernel',
                  'SVC with polynomial (degree 2) kernel',
                  'SVC with sigmoid kernel',
                  'SVC with RBF kernel')

        fig, sub = plt.subplots(2, 2)
        plt.subplots_adjust(wspace=0.4, hspace=0.4)
```

```
X0, X1 = train_x_r_subset_t[:, 0], train_x_r_subset_t[:, 1]
xx, yy = make_meshgrid(X0, X1)

for clf, title, ax in zip(models, titles, sub.flatten()):
    plot_contours(ax, clf, xx, yy,
                  cmap=plt.cm.coolwarm, alpha=0.8)
    ax.scatter(X0, X1, c=y_train_subset,
               cmap=plt.cm.coolwarm, s=20, edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(title)

plt.show()
```



We have seen that the model based on a RBF kernel performs better than the others. However, right now we are not sure what values of C and γ will increase the model's performance. Using grid-search cross validation allows us to fit the model with different hyperparameter values to see which ones improve its performance. In this search, we will use accuracy as the scoring parameter.

```
In [19]: from sklearn.model_selection import GridSearchCV

         # Reload the data taking just 500 data points
         selector = np.random.randint(train_x_r.shape[0], size=500)
         train_x_r_subset = train_x_r[selector, :]
```

```
        y_train_subset = y_train[selector]

        # Define the parameters values we want to test

        tuned_parameters = [{'kernel': ['rbf'],
                             'gamma': [0.0001, 0.0001, 0.001, 0.01, 0.1, 1],
                             'C': [0.01, 0.1, 1, 10, 100, 1000, 10000]}]

        # Perform grid search

        clf = GridSearchCV(svm.SVC(), tuned_parameters, cv=5,
                                scoring='accuracy')
        clf.fit(train_x_r_subset, y_train_subset)

        # Print the parameters that maximized accuracy
        print(clf.best_params_)

        # We will also print all the results
        means = clf.cv_results_['mean_test_score']
        stds = clf.cv_results_['std_test_score']
        for mean, std, params in zip(means, stds, clf.cv_results_['params']):
            print("%0.3f (+/-%0.03f) for %r"
                  % (mean, std * 2, params))
```

```
{'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.01, 'gamma': 0.0001, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.01, 'gamma': 0.0001, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.01, 'gamma': 0.001, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.01, 'gamma': 0.01, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.01, 'gamma': 0.1, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.01, 'gamma': 1, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.1, 'gamma': 0.001, 'kernel': 'rbf'}
0.454 (+/-0.083) for {'C': 0.1, 'gamma': 0.01, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 0.1, 'gamma': 1, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.116 (+/-0.004) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.692 (+/-0.023) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}
0.864 (+/-0.045) for {'C': 1, 'gamma': 0.01, 'kernel': 'rbf'}
0.578 (+/-0.145) for {'C': 1, 'gamma': 0.1, 'kernel': 'rbf'}
0.124 (+/-0.017) for {'C': 1, 'gamma': 1, 'kernel': 'rbf'}
0.706 (+/-0.031) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.706 (+/-0.031) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.860 (+/-0.058) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
0.880 (+/-0.061) for {'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}
```

```
0.598 (+/-0.116) for {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
0.124 (+/-0.017) for {'C': 10, 'gamma': 1, 'kernel': 'rbf'}
0.856 (+/-0.055) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
0.856 (+/-0.055) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
0.864 (+/-0.056) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}
0.880 (+/-0.061) for {'C': 100, 'gamma': 0.01, 'kernel': 'rbf'}
0.598 (+/-0.116) for {'C': 100, 'gamma': 0.1, 'kernel': 'rbf'}
0.124 (+/-0.017) for {'C': 100, 'gamma': 1, 'kernel': 'rbf'}
0.860 (+/-0.055) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.860 (+/-0.055) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.864 (+/-0.056) for {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
0.880 (+/-0.061) for {'C': 1000, 'gamma': 0.01, 'kernel': 'rbf'}
0.598 (+/-0.116) for {'C': 1000, 'gamma': 0.1, 'kernel': 'rbf'}
0.124 (+/-0.017) for {'C': 1000, 'gamma': 1, 'kernel': 'rbf'}
0.860 (+/-0.055) for {'C': 10000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.860 (+/-0.055) for {'C': 10000, 'gamma': 0.0001, 'kernel': 'rbf'}
0.864 (+/-0.056) for {'C': 10000, 'gamma': 0.001, 'kernel': 'rbf'}
0.880 (+/-0.061) for {'C': 10000, 'gamma': 0.01, 'kernel': 'rbf'}
0.598 (+/-0.116) for {'C': 10000, 'gamma': 0.1, 'kernel': 'rbf'}
0.124 (+/-0.017) for {'C': 10000, 'gamma': 1, 'kernel': 'rbf'}
```

We obtain better performance with $C = 10$ and $\gamma = 0.01$. Now, we can train our final model with 3000 training examples and test its accuracy.

```
In [20]: # Select half the dataset to reduce computation time
         selector = np.random.randint(train_x_r.shape[0], size=30000)
         train_x_r_subset = train_x_r[selector, :]
         y_train_subset = y_train[selector]

         # Train the model
         expmodel = svm.SVC(kernel='rbf', gamma = 0.01, C = 10)
         expmodel.fit(train_x_r_subset, y_train_subset)
```

```
Out[20]: SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False)
```

```
In [21]: # Test the model's performance

         print(roc_auc_score(pd.get_dummies(y_test),
                             pd.get_dummies(expmodel.predict(test_x_r))))
```

```
0.9849186619681941
```

3 Neural Networks

Neural Networks are a big part of the current explosion of machine learning development. In this section, we will start exploring the basics of neural networks focusing in understanding the backpropagation algorithm. We will also implement a Neural Network from scratch in order to show how backpropagation can be implemented. After that, we will build a Neural Network using Keras, a high level API that allows to use TensorFlow – arguably the most popular deep learning library – in a fast and easy way.

3.1 Backpropagation Algorithm

3.1.1 Feedforward

We define a network with I input units and J output units with a hidden layer with H units. In this case, the activation of the units in the hidden layer will be defined as:

$$\underset{H\times 1}{z^{(1)}} = \underset{H\times I}{W^{(1)}} \times \underset{I\times 1}{x} + \underset{H\times 1}{b^{(1)}}$$

$$\underset{H\times 1}{a^{(1)}} = \sigma\left(\underset{H\times 1}{z^{(1)}}\right)$$

Where $W^{(1)}$ and $b^{(1)}$ correspond to the weights and the biases of the hidden layer, and σ is an activation function – for simplicity, we will use the sigmoid function. The activation on the output layer will be:

$$\underset{J\times 1}{z^{(2)}} = \underset{J\times H}{W^{(2)}} \times \underset{H\times 1}{a^{(1)}} + \underset{J\times 1}{b^{(2)}}$$

$$\underset{J\times 1}{a^{(2)}} = softmax\left(\underset{J\times 1}{z^{(2)}}\right)$$

Instead of any other activation function, the last layer uses a $softmax$ function that guarantees that all the activations will sum to one, and therefore can be interpreted probabilistically. The $softmax$ function can be expressed as:

$$a_j^{(2)} = \frac{e^{z_j^{(2)}}}{\sum_{j=0}^{9} e^{z_j^{(2)}}}$$

3.1.2 Backpropagation

When we define a neural network, we start with random weights and biases. Obviously, the performance of the network on this initial state is remarkably poor. We need to train it according to different training examples and update the weights and biases in a way that its predictions improve. However, first we need to define a loss function that accounts for how well (or how bad) the network is doing. In other words, we need to measure the inconsistency between the predictions and the actual values in order to tweak different parameters in the network for it to perform better. There are different loss functions, but in classifiers it is common to use the computed using the cross-entropy loss. In a binary classifier, this is:

$$loss(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

Note that $loss(0, 1) = -1$ while $loss(1, 1) = 0$. The same idea can be applied for multiclass classifiers with j classes -- as is the case with the MNIST dataset. Having $y = (y_1, y_2, \ldots, y_J)$ and $\hat{y} = (\hat{y_1}, \hat{y_2}, \ldots, \hat{y_J})$:

$$loss(y, \hat{y}) = -\sum_{j=0}^{J} y_j \log(\hat{y}_j)$$

In practice, we will have hundreds, thousands, or millions of training examples. Therefore, we will average the above formula over K training examples:

$$loss(Y, \hat{Y}) = -\frac{1}{K} \sum_{k=0}^{K} \sum_{j=0}^{J} y_{k,j} \log(\hat{y}_{k,j})$$

Now that we have defined a cost function, we need to find the weights and biases that result in making predictions as close to the true values as possible. That is, this can be understood as an optimization problem in which we want to minimize the loss function. For doing that, we use a technique called gradient descent. That is, we start with randomized weights and biases and try to slowly follow the slope of the surface until we find a local minimum.

$$W^{(l)} := W^{(l)} - \eta \frac{\partial C}{\partial W^{(l)}}$$

$$b^{(l)} := b^{(l)} - \eta \frac{\partial C}{\partial b^{(l)}}$$

Backpropagation is the method we use to calculate the two gradients that are used in the formulas above. The main idea is that C is a function of the activation in the output layer, which is in turn a function of the activation in the previous layers and, ultimately, a function of the input. We can, therefore, compute $\frac{\partial C}{\partial W^{(l)}}$ and $\frac{\partial C}{\partial b^{(l)}}$ using the chain rule for all the layers l. In the neural network

we defined previously we only have two layers, but the same reasoning could be extended to any number of layers:

$$\frac{\partial C}{\partial W^{(2)}} = \frac{\partial C}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}}$$

$$\frac{\partial C}{\partial W^{(1)}} = \frac{\partial C}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$$

$$\frac{\partial C}{\partial b^{(2)}} = \frac{\partial C}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}}$$

$$\frac{\partial C}{\partial b^{(1)}} = \frac{\partial C}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial b^{(1)}}$$

Now, we will develop each of those partial derivatives to finally arrive at the corresponding expressions to calculate these partial derivatives.

Calculating $\frac{\partial C}{\partial W^{(2)}}$ Although we will not develop the derivation here, it can be proven (see Dahal, 2017) that:

$$\underset{J\times 1}{\frac{\partial C}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}}} = \underset{J\times 1}{\hat{y} - y}$$

And:

$$\underset{H\times 1}{\frac{\partial z^{(2)}}{\partial W^{(2)}}} = \underset{H\times 1}{a^{(1)}}$$

Therefore:

$$\frac{\partial C}{\partial W^{(2)}} = \frac{\partial C}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}} = \underset{J\times 1}{\hat{y} - y} \times \underset{1\times H}{\left(a^{(1)}\right)^T}$$

Calculating $\frac{\partial C}{\partial W^{(1)}}$

$$\underset{J\times H}{\frac{\partial z^{(2)}}{\partial a^{(1)}}} = \underset{J\times H}{W^{(2)}}$$

Considering the derivative of the sigmoid function $\sigma(x)$ is $\sigma'(x) = \sigma(x)(1 - \sigma(x))$:

$$\frac{\partial a^{(1)}}{\partial z^{(1)}} = \underset{H\times 1}{\sigma\big(z^{(1)}\big)} \odot \big(1 - \underset{H\times 1}{\sigma\big(z^{(1)}\big)}\big)$$

Lastly:

$$\frac{\partial z^{(1)}}{\partial W^{(1)}} = \underset{I\times 1}{X}$$

Therefore:

$$\frac{\partial C}{\partial W^{(1)}} = \frac{\partial C}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial a^{(1)}}\frac{\partial a^{(1)}}{\partial z^{(1)}}\frac{\partial z^{(1)}}{\partial W^{(1)}} = \underset{H\times J}{\big(W^{(2)}\big)^T} \times \underset{J\times 1}{\big(\hat{y} - y\big)} \odot \Big(\underset{H\times 1}{\sigma\big(z^{(1)}\big)} \odot \big(1 - \underset{H\times 1}{\sigma\big(z^{(1)}\big)}\big)\Big) \times \underset{1\times I}{X^T}$$

Calculating $\frac{\partial C}{\partial b^{(2)}}$

$$\frac{\partial C}{\partial b^{(2)}} = \frac{\partial C}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial b^{(2)}} = \underset{J\times 1}{\hat{y} - y}$$

Calculating $\frac{\partial C}{\partial b^{(1)}}$

$$\frac{\partial C}{\partial b^{(1)}} = \frac{\partial C}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial a^{(1)}}\frac{\partial a^{(1)}}{\partial z^{(1)}}\frac{\partial z^{(1)}}{\partial b^{(1)}} = \underset{H\times J}{\big(W^{(2)}\big)^T} \times \underset{J\times 1}{\big(\hat{y} - y\big)} \odot \Big(\underset{H\times 1}{\sigma\big(z^{(1)}\big)} \odot \big(1 - \underset{H\times 1}{\sigma\big(z^{(1)}\big)}\big)\Big)$$

Summarizing, we use gradient descent to tweak the weights and biases using the following formulas:

$$W^{(2)} := W^{(2)} - \eta \left[\underset{J\times 1}{\hat{y} - y} \times \underset{1\times H}{\big(a^{(1)}\big)^T}\right]$$

$$b^{(2)} := b^{(2)} - \eta \left[\underset{J\times 1}{\hat{y} - y}\right]$$

$$W^{(1)} := W^{(1)} - \eta \left[\underset{H\times J}{\big(W^{(2)}\big)^T} \times \underset{J\times 1}{\big(\hat{y} - y\big)} \odot \Big(\underset{H\times 1}{\sigma\big(z^{(1)}\big)} \odot \big(1 - \underset{H\times 1}{\sigma\big(z^{(1)}\big)}\big)\Big) \times \underset{1\times I}{X^T}\right]$$

$$b^{(l)} := b^{(1)} - \eta \left[\underset{H\times J}{\big(W^{(2)}\big)^T} \times \underset{J\times 1}{\big(\hat{y} - y\big)} \odot \Big(\underset{H\times 1}{\sigma\big(z^{(1)}\big)} \odot \big(1 - \underset{H\times 1}{\sigma\big(z^{(1)}\big)}\big)\Big)\right]$$

With this, we can now implement backpropagation from scratch.

3.2 A Neural Network from Scratch

We will start opening the data again, also using scaling. This time, we will one-hot code the response variable and transpose it so it is a column vector. We will also transpose the dataset containing the images so each column correspond to an image and each row is the activation of a certain pixel.

```
In [22]: mnist = tf.keras.datasets.mnist

         # Load dataset
         (x_train, y_train),(x_test, y_test) = mnist.load_data()
         x_train, x_test = x_train / 255.0, x_test / 255.0

         # Reshaping to have 784 features instead of 28x28
         x_train_r = np.reshape(x_train, (x_train.shape[0], x_train.shape[1] * x_train.shape[2]))
         x_test_r = np.reshape(x_test, (x_test.shape[0], x_test.shape[1] * x_test.shape[2]))

         # One hot encoding
         y_train_r = np.array(pd.get_dummies(y_train))
         y_test_r = np.array(pd.get_dummies(y_test))

         # Transpose so each column is a training/testing example
         X_train = x_train_r.T
         X_test = x_test_r.T
         Y_train = y_train_r.T
         Y_test = y_test_r.T
```

Now, we define the sigmoid function and the multiclass loss as presented in the previous section.

```
In [23]: def sigmoid(z):
             s = 1 / (1 + np.exp(-z))
             return s

         def multiclass_loss(Y, Y_hat, s):
             C = -(1/s) * np.sum(Y * np.log(Y_hat))
             return C
```

We create a class, 'NeuralNetwork', which initially offers the user the possibility of initializing a neural network with a I number of input nodes, H nodes in the hidden layer, and J nodes in the output layer. The weights are sampled randomly from a normal distribution with mean of 0 and standard deviation of 1, while the biases are all initiated at 0.

```
In [24]: class NeuralNetwork:
             def __init__(self, I, H, J):
```

```
        self.I = I
        self.H = H
        self.J = J

        self.W1 = np.random.randn(self.H, self.I)
        self.b1 = np.zeros((self.H, 1))
        self.W2 = np.random.randn(self.J, self.H)
        self.b2 = np.zeros((self.J, 1))

    # Feedforward, as described in the previous section
    def feedforward(self, X):
        z1 = (self.W1 @ X) + self.b1
        a1 = sigmoid(z1)
        z2 = (self.W2 @ a1) + self.b2
        a2 = np.exp(z2) / np.sum(np.exp(z2), axis=0)
        self.output = a2

    # Backpropagation, as described in the previous section. The
    # only change is that we use minibatchsize.
    def backpropagation(self, X, Y, minibatchsize, epochs, eta):
        s = minibatchsize
        K = X.shape[1]
        for i in range(0,epochs):
            c = 0
            for mb in range(int(K//(s+1))):
                Xs = X[:,c:c+s]
                Ys = Y[:,c:c+s]
                c += s + 1

                z1 = (self.W1 @ Xs) + self.b1
                a1 = sigmoid(z1)
                z2 = (self.W2 @ a1) + self.b2
                a2 = np.exp(z2) / np.sum(np.exp(z2), axis=0)

                cost = multiclass_loss(Ys, a2, s)
                a2minusy = (a2-Ys)
                dW2 = (1/(s+1)) * (a2minusy @ a1.T)
                db2 = (1/(s+1)) * np.sum(a2minusy, axis = 1, keepdims=True)

                derivative = self.W2.T @ (a2-Ys) * (sigmoid(z1)*(1-sigmoid(z1)))
                dW1 = (1/(s+1)) * (derivative @ Xs.T)
                db1 = (1/(s+1)) * np.sum(derivative, axis=1, keepdims=True)

                self.W2 = self.W2 - eta * dW2
                self.b2 = self.b2 - eta * db2

                self.W1 = self.W1 - eta * dW1
                self.b1 = self.b1 - eta * db1
```

The class offers a method, 'feedforward', that allows the user to input some data (the number of rows should be equal the I that is defined in the neural network) and output the prediction of the network.

```
In [25]: nn = NeuralNetwork(784,100,10)
         nn.feedforward(X_test)
         # The predicted and true y are one-hot encoded. Using np.argmax
         # outputs the predicted or true label
         print(np.argmax(nn.output, axis = 0))
         print(np.argmax(Y_test, axis = 0))
```

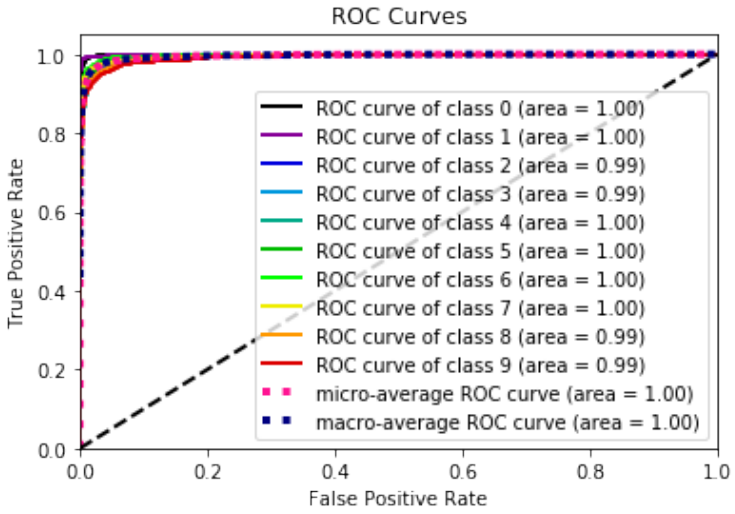
```
[0 7 3 ... 2 7 7]
[7 2 1 ... 4 5 6]
```

Unsurprisingly, the network is performing poorly. That is because we need to use backpropagation first to allow the network to adjust the weights and biases in order to minimize the cost function that we defined previously. The backpropagation method asks for a training dataset and a response variable. It also allows to define the minibatchsize, the number of epochs, and the learning rate. In this case, we will use a minibathsize of 100, an epoch of 1, and a learning rate $\eta = 1$.

```
In [26]: nn = NeuralNetwork(784,100,10)
         # Train the network
         nn.backpropagation(X_train,Y_train, minibatchsize = 50, epochs = 2, eta = 1)
```

```
In [27]: import scikitplot as skplt

         # Calculate AUC score
         nn.feedforward(X_test)
         print(roc_auc_score(Y_test.T, nn.output.T))

         # Plot ROC curves for each class
         labels = np.argmax(Y_test, axis = 0)
         skplt.metrics.plot_roc(labels, nn.output.T)
         plt.show()
```

```
0.9956811036672173
```



The ROC curve clearly shows the network has been trained correctly and is performing even better than the SVM trained in the previous section. Now, we will train a more advanced Neural Network using Keras.

3.3 Building a Neural Network with Keras

TensorFlow is probably the de facto industry standard to build complex models. Importantly, it allows to build neural networks of different types and architectures in an efficient way without requiring us to build the code from scratch, as we did before. Keras is a library that runs in top of TensorFlow (and other libraries), making the process of building and training a machine learning model much faster an easier for the user. However, using TensorFlow may be recommended when extra control about the model and training process is required.

Using Keras and TensorFlow, we will build a 2 layer neural network with 1200 for each hidden layer. There are, however, three important differences between this network and the one we built from scratch in the previous section:

- We are using ReLU (Rectified Linear Unit) instead of the sigmoid function as an activation function, as it can be computed faster and tends to converge quicker than other activation functions, for that has become the standard for neural networks. The last layer will still use the softmax function.
- We will use a regularization technique called dropout in order to avoid overfitting. This dropout means that for each training step some neurons are ignored in order to 'teach' the neural network not to depend on specific units. We have selected a dropout rate of 0.4. * We use Adam to adapt the learning rate for each step in the training process. This makes the gradient descent work faster while solving some problems that may emerge from using other optimizers.

```
In [28]: import tensorflow as tf
         import tensorflow.keras as keras
         from tensorflow.python.keras.models import Sequential
         from tensorflow.python.keras.layers import Dense, Dropout, Activation
         from tensorflow.python.keras.optimizers import SGD

         # We reload the original dataset, rescale and one-hot code.

         mnist = tf.keras.datasets.mnist

         (x_train, y_train),(x_test, y_test) = mnist.load_data()
         x_train, x_test = x_train / 255.0, x_test / 255.0
         y_train_hot = pd.get_dummies(y_train)
         y_test_hot = pd.get_dummies(y_test)


In [29]: model = keras.Sequential([
         # Flatten the input
         keras.layers.Flatten(input_shape=(28, 28)),
         # Hidden layer with 1200 units with Relu activation
         keras.layers.Dense(1200, activation=tf.nn.relu),
         # Moderately high dropout rate that will avoid overfitting
         keras.layers.Dropout(0.4),
         # Second hidden layer, also with Relu
         keras.layers.Dense(1200, activation=tf.nn.relu),
         # Output layer with 10 units and softmax activation
         keras.layers.Dense(10, activation=tf.nn.softmax)])

         # We use categorical crossentropy as a loss function as we did
         # in the network we built from scratch, and additionally we
         # use Adam optimizer.
         model.compile(optimizer=tf.train.AdamOptimizer(),
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])

         # Ten epochs with a batch size of 62
         model.fit(x_train, y_train_hot, epochs=10, batch_size = 62)


Epoch 1/10
60000/60000 [========================] - 35s 591us/step - loss: 0.2181 - acc: 0.9329
Epoch 2/10
60000/60000 [========================] - 33s 547us/step - loss: 0.1139 - acc: 0.9652
Epoch 3/10
60000/60000 [========================] - 35s 582us/step - loss: 0.0882 - acc: 0.9723
Epoch 4/10
60000/60000 [========================] - 37s 618us/step - loss: 0.0740 - acc: 0.9770
Epoch 5/10
```

```
60000/60000 [==========================] - 35s 579us/step - loss: 0.0667 - acc: 0.9788
Epoch 6/10
60000/60000 [==========================] - 38s 627us/step - loss: 0.0582 - acc: 0.9820
Epoch 7/10
60000/60000 [==========================] - 31s 521us/step - loss: 0.0524 - acc: 0.9838
Epoch 8/10
60000/60000 [==========================] - 30s 504us/step - loss: 0.0485 - acc: 0.9850
Epoch 9/10
60000/60000 [==========================] - 29s 484us/step - loss: 0.0429 - acc: 0.9867
Epoch 10/10
60000/60000 [==========================] - 30s 504us/step - loss: 0.0425 - acc: 0.9875
```

```
Out[29]: <tensorflow.python.keras.callbacks.History at 0x2a9fe5921d0>
```

```
In [30]: model.evaluate(x_test, y_test_hot, verbose=0)
```

```
Out[30]: [0.07052413852757454, 0.9812]
```

The performance of the model is more than 98%. Note that this is a different way of measuring the accuracy than the AUC shown for the previous model. It is important to know that this model performs better than the one built from scratch.

3.4 Data Science Challenge: Building a Convolutional Neural Network

For the data science challange I will build a CNN. We will use a similar architecture as the classic LeNet 5, but we will include a dropout rate and ReLU activation functions, and the Adam optimizer. The training will be done using the Adam optimizer with 5 epochs and a mini-batch size of 128. A 20% of the training data will be set apart and not used for training, and just used to evaluate the loss of the model at the end of each epoch.

```
In [31]: # It is necessary to add one dimension to the data. In general, CNN deal with
         # images with various color channels and expect a fourth dimension.
         x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], x_train.shape[2], 1)
         x_test = x_test.reshape(x_test.shape[0], x_test.shape[1], x_test.shape[2], 1)
         y_train_hot = pd.get_dummies(y_train)
```

```
In [32]: model = tf.keras.Sequential()
         model.add(tf.keras.layers.Conv2D(6, kernel_size=(3, 3),
```

```
                    activation='relu',
                    input_shape=(28,28,1)))
        model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
        model.add(tf.keras.layers.Conv2D(16, (3, 3), activation='relu'))
        model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
        model.add(tf.keras.layers.Flatten())
        model.add(tf.keras.layers.Dense(120, activation='relu'))
        model.add(tf.keras.layers.Dropout(0.5))
        model.add(tf.keras.layers.Dense(84, activation='relu'))
        model.add(tf.keras.layers.Dense(10, activation='softmax'))

        model.compile(optimizer=tf.train.AdamOptimizer(),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])


        model.fit(x_train, y_train_hot,
                  batch_size=128,
                  epochs=5,
                  verbose=1,
                  validation_split = 0.2,
                  shuffle=True)
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/5
48000/48000 [========================] - 232s 5ms/step - loss: 0.5382 - acc: 0.8308
- val_loss: 0.1147 - val_acc: 0.9665
Epoch 2/5
48000/48000 [========================] - 198s 4ms/step - loss: 0.1652 - acc: 0.9488
- val_loss: 0.0871 - val_acc: 0.9732
Epoch 3/5
48000/48000 [========================] - 173s 4ms/step - loss: 0.1262 - acc: 0.9608
- val_loss: 0.0667 - val_acc: 0.9792
Epoch 4/5
48000/48000 [========================] - 191s 4ms/step - loss: 0.1015 - acc: 0.9698
- val_loss: 0.0561 - val_acc: 0.9833
Epoch 5/5
48000/48000 [========================] - 186s 4ms/step - loss: 0.0887 - acc: 0.9728
- val_loss: 0.0505 - val_acc: 0.9858
```

```
Out[32]: <tensorflow.python.keras.callbacks.History at 0x2a98e260c50>
```

```
In [33]: model.evaluate(x_test, y_test_hot, verbose=0)
```

```
Out[33]: [0.04359599726048182, 0.9859]
```

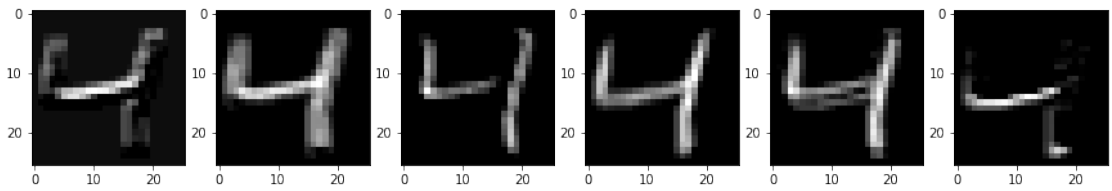
It trains slowly compared with the previous neural network, but with enough epochs it outperforms it.

Lastly, we will visualize the convolutional and max pooling layers for a certain input. For that, we will define a function visualizelayer:

```
In [34]: def visualizelayer(layer, image = 0):

             img_to_visualize = x_train[image]
             img_to_visualize = np.expand_dims(img_to_visualize, axis=0)

             inputs = [tf.keras.backend.learning_phase()] + model.inputs
             _convout1_f = tf.keras.backend.function(inputs, [model.layers[layer].output])

             def convout1_f(X):
                 return _convout1_f([0]+[X])

             convolutions = convout1_f(img_to_visualize)
             convolutions = np.squeeze(convolutions)
             n = convolutions.shape[0]

             if convolutions.shape[2] == 6:
                 a = 6
                 b = 1
             elif convolutions.shape[2] == 16:
                 a = 8
                 b = 2

             fig, axs = plt.subplots(b,a, figsize=(15, 4))
             axs = axs.ravel()
             for i in range(convolutions.shape[2]):
                 axs[i].imshow(convolutions[:,:,i], cmap='gray')
```

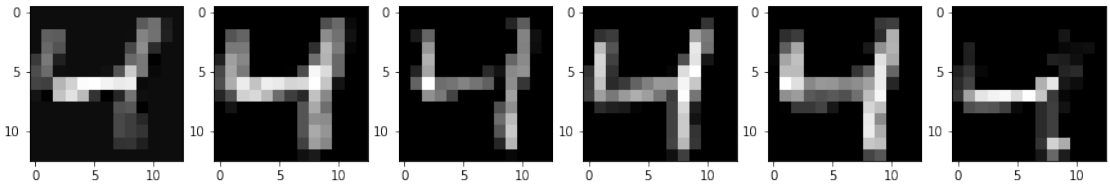
First convolutional layer:

```
In [35]: visualizelayer(0, 2)
```



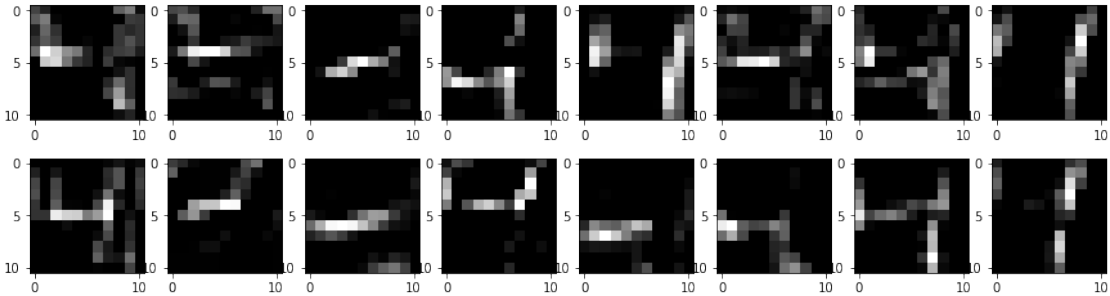
First max pooling layer:

```
In [36]: visualizelayer(1, 2)
```



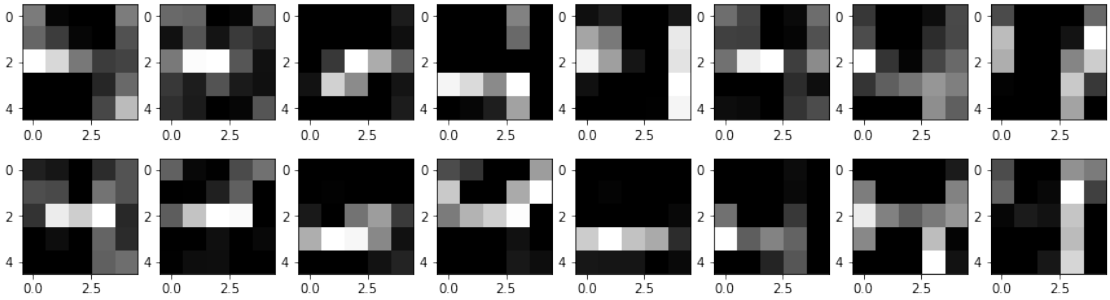
Second convolutional layer:

```
In [37]: visualizelayer(2, 2)
```



Second max pooling layer:

```
In [38]: visualizelayer(3, 2)
```



4 References

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.

Dahal, P. (2017) Classification and Loss Evaluation - Softmax and Cross Entropy Loss. Retrieved from https://deepnotes.io/softmax-crossentropy