HW5: Neural Networks for Classifying Fashion MINST

AMATH 482 | Celeste Becker

Abstract

This homework is about exploring neural networks. We will use two different types of neural networks to build a classifier for the MINST Fashion dataset. This dataset contains images of 10 different classes of fashion items. In part one we will build a classifier using a fully-connected neural network, and in part two we will use convolutional neural network. Then we will use the results from each to compare their accuracy.

Sec. I. Introduction and Overview

The goal is to use neural networks to classify the images of different types of fashion items in the MINST Fashion dataset. We will split the Fashion-MINST data into 55,000 training images, and 10,000 test images. We will also use 5,000 images as validation data. Because the images are in grayscale, each image is 28 by 28.

A neural network is a model that is made up of called perceptrons. Each perceptron is made up of linear threshold logic units. There are three layers to a neural network. The input layer, the hidden layer(s), and the output layer. The number of neurons in the output layer represents how many different classes there are. So, we will have ten different neurons in the output layer.

In part one, we will build the classifier using a fully-connected neural network. In part two, we will build the classifier using a convolutional neural network (CNN).

Sec. II. Theoretical Background

The information below is mainly referenced from first and second lecture notes on Neural Networks from class.

Neural Networks: A neural network is a model that is made up of neurons called perceptrons. Each perceptron is made up of linear threshold logic units. Each neuron inputs and outputs a number and each connection has a weight (w) associated with it. Sometimes there are also bias neurons added to a layer, which just outputs a constant number (b). An **input layer**, outputs the same number it gets as input. By feeding these numbers into a neuron, we can determine whether the neuron should fire or not by comparing the weighted sum of the inputs to a threshold value.

Perceptron output (Equation 1): $y = \sigma(Ax + b)$

in the above equation, sigma is the Activation function, A is a matrix containing the weights, X is a column vector of the input data, b is a column vector of the biases, and y is the output.

Activation Function: By plugging (Ax +b) into the activation function, we can set the neuron to be active if the sum is greater than a threshold, and inactive if it is less than the threshold. There are different types of activation functions, but the most popular, and the one we are using, is the **Rectified Linear Unit Function (ReLU)**. Applying ReLU to a layer of neurons means that all entries that are negative become zero, and all positive entries get left alone.

ReLu (Equation 2): $\sigma(x) = \max(0, x)$

In part one, we will build a **Fully-Connected Neural Network**. Fully connected means that each neuron is connected to every other neuron in the previous layer. When each neuron is directly connected to the neuron in the previous layer it is a dense layer. A fully-connected neural network is made up of only dense layers.

In part two, we will build a **Convolutional Neural Network (CNN)**. In a CNN, you pass the data through convolutional layers, which uses a small window that slides through the entire layer and filters the data into a new node. Then you pass the data through a pooling layer. We are using max pooling, which takes the maximum value for all the nodes in a convolutional window. In a CNN you can have multiple convolutional and pooling layers, in combination with dense layers.

Loss function: This function is used to determine which parameters make the model best fit the data. Here we are using the cross-entropy loss function:

Cross-entropy loss (Equation 3): $\frac{1}{N}\sum_{\mathrm{j}-1}^{\mathrm{N}}[\, y_j \ln(p_j) + (1 - y_j)\ln(1 - p_j)]$

Gradient Decent: Training a neural network requires using a gradient decent algorithm for optimization. The Stochastic Gradient Descent Algorithm is an iterative algorithm that finds the minimum of a convex function.

Over Fitting: Training a neural network, we need to be careful of doing much better on training data than on validation data. This means that the model is finding patterns in the noise of the training data, and creating an incorrect model.

The process of building a neural network starts with loading in the dataset, splitting the data into training, validation, and test data, and finally building the model. Once you build the base of the model, you tweak it to get the best accuracy without over fitting by running the training data through it and then testing it on the validation data. Once you have a decent model, you do a final test on the test data.

Sec. III. Algorithm Implementation and Development

This section has code references to the *Appendix B. Python Codes* at the end of this report. Code reference lines are also included. This program has been split up into two files. One for each part.

To build both neural network programs, for part one and part two, the process was very similar. First, we load the data, then we split the Fashion-MINST data into 55,000 training images, and 10,000 test images. We will also use 5,000 images as validation data. Because the images are in grayscale, each image is 28 by 28. To convert the data to a floating-point number we divide by 255 for each set of data.

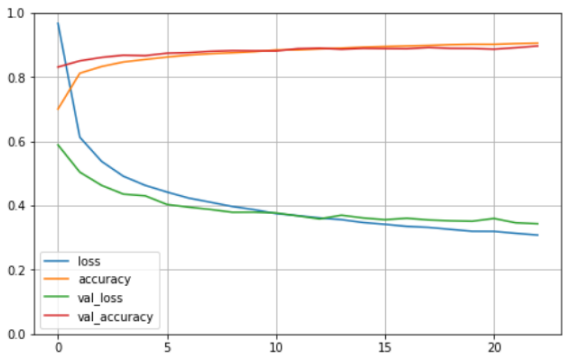
Next, we build the neural network model. For this program we were instructed to experiment with different parameters of the network. Developing the model required a lot of trial and error testing. Here are some main points from the development process:

- **Depth of network:** A very high number of layers seems to produce over-fitting problems. A very low number can cause the network to have a very low accuracy.
- **Width of layers:** Very wide layers can cause over fitting, seeing noise as patters in the training data. However, layers that are too narrow seems to also produce low accuracy
- **Optimizer:** In this program we are using Atom as the optimizer.
- **Learning rate:** By changing the learning rate of the optimizer, you are changing the rate that the program converges to the minima for gradient decent. The learning rate is the amount that the weights are updated during training. Choosing a learning rate that is too big causes unstable training, by taking to big steps to converge to the minima. While taking to small of a learning rate can cause the process to get stuck. The learning rate determines how much to change the weights and biases in each procedure.
- **Activation function:** I chose to use the ReLU (Equation 2) function.

Part 1 – Fully-Connected Neural Network: To build the model, I found that the best results came from setting the dense layer activation function to ReLU. The model has a first initial layer that takes the 28 by 28 image and flattens it into a column. Then I have 3 hidden layers. The first dense hidden layer has 300 neurons. Then I use the tf.keras.layers.Dropout(0.1). Dropout randomly sets the fraction 0.1 of inputs to zero at each update during training. This helps prevent over fitting. Next, there is another dense layer with 32 neurons, and another Dropout of 0.1. Finally there is the last hidden dense layer of 16 neurons, and the final output layer of ten neurons, using the softmax activation function. The learning rate is set at 0.0001. Epoch, the number of times the training dataset is presented to the model, is set to 23. Reference Code: Part 1: #NEURAL NETOWRK

Part 2 – Convolutional Neural Network (CNN): To build the model, we repeat this pattern twice: convolutional layer, pooling layer, and Dropout. Tweaking the number of strides, and filters to get the best results. Then we flatten the data, send it through one dense layer, and then to the output layer. Example reference: Part 2: #NEURAL NETOWRK

Figure 1 - Part 1: Fully-Connected Neural Network

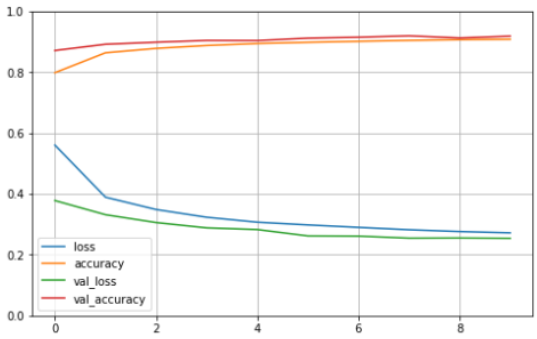



```
training accuracy: 0.9061
test accuracy: 0.8600
```

Figure 2 – Confusion Matrix With Test Data

```
[[843   4  24  25  11   0  77   0  16   0]
 [  0 978   0  14   5   0   2   0   1   0]
 [ 13   2 684   8 255   0  36   0   2   0]
 [ 16  19  10 841  86   0  22   0   6   0]
 [  0   0  38  11 938   0  11   0   2   0]
 [  0   0   0   0   0 935   0  16   4  45]
 [127   4  85  23 219   0 529   0  13   0]
 [  0   0   0   0   0  15   0 895   0  90]
 [  4   1   2   3   8   1   2   4 975   0]
 [  0   0   0   0   0   4   1  13   0 982]]
```

Figure 3 – Part 2: Convolutional Neural Network




```
training accuracy: 0.9086
test accuracy: 0.9062
```

Figure 4 – Confusion Matrix With Test Data

	0	1	2	3	4	5	6	7	8	9
0	882	1	7	24	2	2	79	0	3	0
1	1	975	0	15	5	0	2	0	2	0
2	21	1	803	12	87	0	76	0	0	0
3	19	6	3	926	24	0	22	0	0	0
4	1	1	28	21	901	0	47	0	1	0
5	0	0	0	1	0	964	0	23	0	12
6	137	1	36	31	82	0	706	0	7	0
7	0	0	0	0	0	5	0	966	0	29
8	5	2	2	5	1	2	4	5	974	0
9	0	0	0	0	0	2	1	24	0	973

Sec. IV. Computational Results

Part 1 – Fully-Connected Neural Network: As you can see from Figure 1 above, there is a little bit of overfitting happening when the validation loss (the green line) raises above the loss (the blue line. This will negatively affect the accuracy of our model. The training accuracy was at 90.61% and after running the model on the test data, the test accuracy came out at 86%. Most likely the test accuracy would be higher if the model was not overfitted to the training data. You can see from Figure – 3, the confusion matrix of the test data. The largest number outside the diagonal, representing image data that is mis-categorized, is 255 (in row 3, column 5) corresponding to mis-classifying pullovers as a coat.

Part 2 – Convolutional Neural Network (CNN): Here we can see in Figure – 3 above that there is no overfitting. This leads to a better training accuracy of the model, at 90.86%. And a significantly better test accuracy of 90.62%. We can see from Figure 4 that the largest number off the diagonal is 137 (in row 6, column 0) corresponding to mis-categorizing T-shirts as shirts. As far as mis-categorizations go, this is a pretty good one to have because technically a T-shirt is a shirt! Overall, the CNN performs much better than the Fully-Connected Neural Network in Part 1.

Sec. V. Summary and Conclusions

In conclusion, we were able to successfully use neural networks to classify the images of different types of fashion items in the MINST Fashion dataset. We created two the classifier using two different types of neural networks: A fully-connected neural network and a convolutional neural network. Overall, we saw that the CNN gave significantly better results on test data than the fully-connected neural network.

Appendix A. Python functions used and brief implementation explanation

tf.keras.layers.Dense

- Adds a densely-connected layer to the neural network
- *Implementation*: Part 1 - used in a function to define a dense layer function: my_dense_layer = partial(tf.keras.layers.Dense, activation="relu", kernel_regularizer=tf.keras.regularizers.l2(0.0001))

tf.keras.layers.Flatten

- Flattens the input
- *Implementation*: Part 1 - Used to flatten the image from 28 by 28 to a single column: tf.keras.layers.Flatten(input_shape=[28,28])

tf.keras.layers.Dropout

- Applies dropout to the input. Can set a float between 0 and 1 to set the fraction of the input units to drop out.
- *Implementation*: Part 1 - tf.keras.layers.Dropout(0.1)

tf.keras.models.Sequential

- Creates a linear stack of layers
- *Implementation*: Part 1 - used to define model: model = tf.keras.models.Sequential([…])

tf.keras.layers.Conv2D

- 2D convolution layer (e.g. spatial convolution over images)
- *Implementation*: Part 2 - used to define the convolutional layer function: my_conv_layer = partial(tf.keras.layers.Conv2D, activation="relu", padding="valid")

tf.keras.layers.MaxPooling2D

- max pooling operation for special data. (Takes max number)
- *Implementation*: Part 2: tf.keras.layers.MaxPooling2D(pool_size=2, strides=2)

tf.keras.models.fit

- Trains the model for a fixed number of epochs (iterations on a dataset
- *Implementation*: Part 2 – used to train the model :
  history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid,y_valid))

tf.keras.models.compile

- Configures the model for training
- *Implementation*: Part 2:
  model.compile(loss="sparse_categorical_crossentropy",
  optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
  metrics=["accuracy"])

Appendix B. Python codes

This HW was split into two different files for each part below:

Part 1 – Fully-Connected Neural Network:

In [135]:
```
#imports all nessary librarys
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix
```

In [136]:
```
#loads in fashion data, sets training and test data
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

In [137]:
```
#plot first 9 images in X training data
plt.figure()
for k in range(9):
    plt.subplot(3,3,k+1)
    plt.imshow(X_train_full[k], cmap="gray")
    plt.axis('off')
plt.show()
```

In [138]:
```
#get validation data from training data -> first 5000 images
x_valid = X_train_full[:5000] / 255.0

#Setting training data -> 55000 images after first 5000
x_train = X_train_full[5000:] / 255.0

#Deviding test data by 255.0 to convert to double
x_test = X_test / 255.0

#NOTE the labels are in ytrain -> contains the integer that coresponds
#to the type of fashion

#Spliting data labels into training and valid
y_valid = y_train_full[:5000]
y_train = y_train_full[5000:]
```

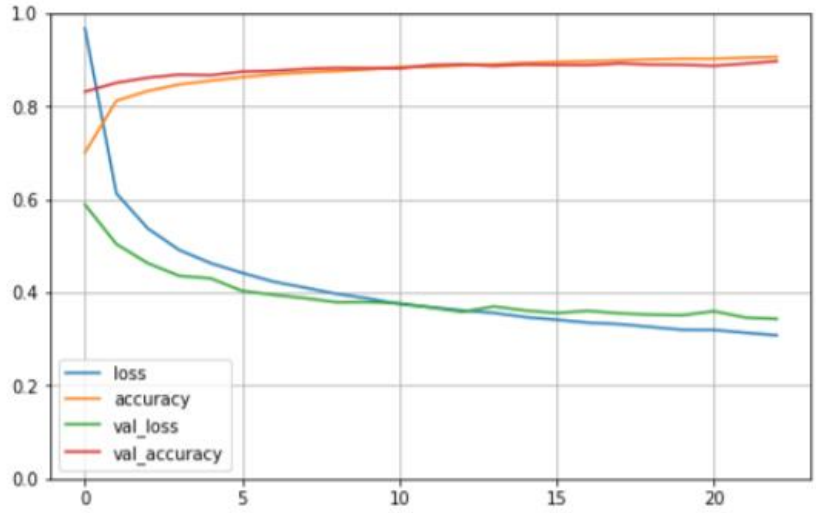
In [139]:
```
#NEURAL NETOWRK --> Build Model
#Sequential ->  each layer feeds into the next layer

from functools import partial

my_dense_layer = partial(tf.keras.layers.Dense, activation="relu", kernel_regu
larizer=tf.keras.regularizers.l2(0.0001))

model = tf.keras.models.Sequential([
    #first layer --  takes input image of 28x28 and flattens it into a column
    tf.keras.layers.Flatten(input_shape=[28,28]),

    #Hidden layers --  (fully connected = Dense)
    #20 neurons, relu activation function
    my_dense_layer(300),
    tf.keras.layers.Dropout(0.1),
    #my_dense_layer(80),
    my_dense_layer(32),
    tf.keras.layers.Dropout(0.1),
    #my_dense_layer(16),
    my_dense_layer(16),

    #output layer -- (fully connected = Dense)
    #10 because there are ten options that we are trying to classify
    my_dense_layer(10, activation="softmax")

])
```

In [140]:
```
#Training options
#loss = loss function
#optimizer -> used to solve for optimal weights and biases
#optimization is iterative, learning rate determines how much
#you are going to change your weights and biases in each iteration

model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate = 0.0001),
              metrics = ["accuracy"])
```

In [141]:
```
#Training Model
#epochs = how many times want optimization to run use lot more than 5
#Save results of training model to history
history = model.fit(x_train, y_train, epochs=23, validation_data=(x_valid,y_va
lid))
```

```
Epoch 23/23
55000/55000 [==============================] - 9s 160us/sample - loss: 0.3081
- accuracy: 0.9061 - val_loss: 0.3433 - val_accuracy: 0.8970
```

In [142]:
```
#Plotting
pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()
```



In [144]:
```
#Plot confusion matrix
y_pred = model.predict_classes(x_train)
conf_train = confusion_matrix(y_train,y_pred)
print(conf_train)
```

```
[[4811    5   90  169   19    1  429    1   18    0]
 [   3 5343    3   81    9    0    5    0    0    0]
 [  26    1 4704   46  489    0  226    0    4    0]
 [  49   16   16 5213  146    0   57    0    2    0]
 [   2    4  306  129 4915    0  153    0    3    0]
 [   0    0    0    0    0 5460    0   40    2    5]
 [ 551    5  384  135  408    0 4016    0    8    0]
 [   0    0    0    0    0   43    0 5359    2   84]
 [   9    1    7   13   14    7   17    8 5434    0]
 [   0    0    0    0    0   11    0  147    0 5336]]
```

In [145]:
```
#This evaluates the model with the test data
model.evaluate(X_test,y_test)
```

```
10000/10000 [==============================] - 1s 96us/sample - loss: 48.2413
- accuracy: 0.8600
```

Out[145]:
```
[48.24127066493034, 0.86]
```

In [146]:
```
#Plot confusion matrix with test data
y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)
print(conf_test)
```

```
[[843   4  24  25  11   0  77   0  16   0]
 [  0 978   0  14   5   0   2   0   1   0]
 [ 13   2 684   8 255   0  36   0   2   0]
 [ 16  19  10 841  86   0  22   0   6   0]
 [  0   0  38  11 938   0  11   0   2   0]
 [  0   0   0   0   0 935   0  16   4  45]
 [127   4  85  23 219   0 529   0  13   0]
 [  0   0   0   0   0  15   0 895   0  90]
 [  4   1   2   3   8   1   2   4 975   0]
 [  0   0   0   0   0   4   1  13   0 982]]
```

Part 2 – CNN:

In [53]:
```
#imports all nessary librarys
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix
```

In [42]:
```
#loads in fashion data, sets training and test data
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

In [43]:
```
X_valid = X_train_full[:5000] / 255.0
X_train = X_train_full[5000:] / 255.0
X_test = X_test / 255.0

y_valid = y_train_full[:5000]
y_train = y_train_full[5000:]

X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]
```

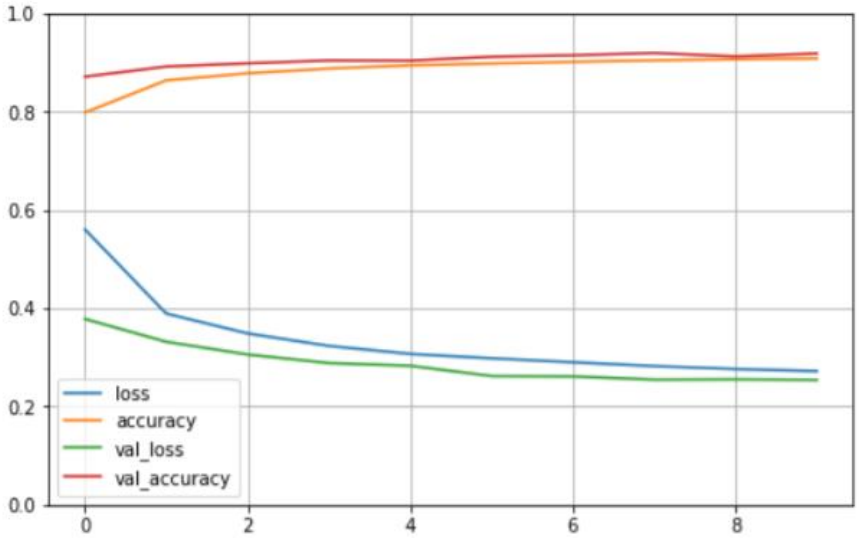
In [86]:
```
from functools import partial

my_dense_layer = partial(tf.keras.layers.Dense, activation="relu", kernel_regu
larizer=tf.keras.regularizers.l2(0.0001))
my_conv_layer = partial(tf.keras.layers.Conv2D, activation="relu", padding="va
lid")

model = tf.keras.models.Sequential([

    my_conv_layer(32,3,padding="same",input_shape=[28,28,1]),
    tf.keras.layers.MaxPooling2D(pool_size=2, strides=2),
    tf.keras.layers.Dropout(0.2),

    my_conv_layer(16,3,padding="same",input_shape=[28,28,1]),
    tf.keras.layers.MaxPooling2D(pool_size=2, strides=2),
    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Flatten(),
    my_dense_layer(32),
    my_dense_layer(10, activation="softmax")


])
```

In [87]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              metrics=["accuracy"])
```

In [88]:
```
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid,y_valid))
```

```
Epoch 10/10
55000/55000 [==============================] - 48s 867us/sample - loss: 0.2721 - accuracy: 0.9086 - val_loss: 0.2542 - val_accuracy: 0.9188
```

In [89]:
```
pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()
```



In [90]:
```
y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)
```

```
[[4867    1   86   91    5    1  482    0   10    0]
 [   1 5378    2   57    3    0    1    0    2    0]
 [  30    3 5043   46  143    0  227    0    4    0]
 [  74   20   12 5181  122    0   89    0    1    0]
 [   3    4  391  122 4557    0  432    0    3    0]
 [   0    0    0    0    0 5445    0   32    2   28]
 [ 461    3  315  118  172    0 4429    0    9    0]
 [   0    0    0    0    0   52    0 5189    1  246]
 [   2    0   10    9   15    3   30    0 5441    0]
 [   0    0    0    0    1   11    0   67    1 5414]]
```

In [91]:
```
model.evaluate(X_test,y_test)
```

```
10000/10000 [==============================] - 2s 222us/sample - loss: 0.2783 - accuracy: 0.9062
```

Out[91]:
```
[0.27826369936466216, 0.9062]
```

In [84]:
```
y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)
print(conf_test)
```

```
[[882   1   7  24   2   2  79   0   3   0]
 [  1 975   0  15   5   0   2   0   2   0]
 [ 21   1 803  12  87   0  76   0   0   0]
 [ 19   6   3 926  24   0  22   0   0   0]
 [  1   1  28  21 901   0  47   0   1   0]
 [  0   0   0   1   0 964   0  23   0  12]
 [137   1  36  31  82   0 706   0   7   0]
 [  0   0   0   0   0   5   0 966   0  29]
 [  5   2   2   5   1   2   4   5 974   0]
 [  0   0   0   0   0   2   1  24   0 973]]
```

In [92]:
```
fig, ax = plt.subplots()

# hide axes
fig.patch.set_visible(False)
ax.axis('off')
ax.axis('tight')

# create table and save to file
df = pd.DataFrame(conf_test)
ax.table(cellText=df.values, rowLabels=np.arange(10), colLabels=np.arange(10),
loc='center', cellLoc='center')
fig.tight_layout()
plt.savefig('conf_mat.pdf')
```

In [92]:
```
fig, ax = plt.subplots()

# hide axes
fig.patch.set_visible(False)
ax.axis('off')
ax.axis('tight')

# create table and save to file
df = pd.DataFrame(conf_test)
ax.table(cellText=df.values, rowLabels=np.arange(10), colLabels=np.arange(10),
loc='center', cellLoc='center')
fig.tight_layout()
plt.savefig('conf_mat.pdf')
```

	0	1	2	3	4	5	6	7	8	9
0	882	1	7	24	2	2	79	0	3	0
1	1	975	0	15	5	0	2	0	2	0
2	21	1	803	12	87	0	76	0	0	0
3	19	6	3	926	24	0	22	0	0	0
4	1	1	28	21	901	0	47	0	1	0
5	0	0	0	1	0	964	0	23	0	12
6	137	1	36	31	82	0	706	0	7	0
7	0	0	0	0	0	5	0	966	0	29
8	5	2	2	5	1	2	4	5	974	0
9	0	0	0	0	0	2	1	24	0	973