Fashion MNIST Classification using Artificial Neural Networks

Seong Hyun Han

March 13, 2020

Abstract

In this assignment, artificial neural networks are used to develop a machine learning algorithm that can classify different fashion item images based on their respective features. After developing two different neural network models, Fashion-MNIST datasets will be used to train, validate, and test the developed models. Their respective hyperparameters such as layer width and depth will be optimized in order to increase their accuracies and minimize the loss function value. Later the results of the two neural networks will be analyzed and compared. This assignment will also give informative background on developing neural networks.

Sec. I Introduction and Overview

Artificial neural networks, which were modeled after biological neural networks comprised of neurons and synapses, can be used as machine learning tools to carry out tasks such as classification, dimensionality reduction, and regression. As such, the main goal for this assignment is to develop artificial neural networks that can correctly classify 10 different fashion item images from the Fashion-MNIST dataset. The Fashion-MNIST dataset contains 60,000 training images and 10,000 test images with their respective labels. Each image size is 28 x 28 with label values from 0 to 9, which corresponds to 10 different fashion items. Before beginning to develop the neural networks, the dataset will be preprocessed, allocating 5,000 images from the training dataset to be used as validation data, using the rest of the 55,000 images as training data. Furthermore, the datatype of each image will be converted from uint8 values to floats. The first neural network will be a feedforward multilayer perceptron that is fully connected (dense layers). The hyperparameters for this neural network, such as learning rate and width of each of the layers, will be adjusted in order to obtain an architecture with sufficient training and validation accuracies along with low training and learning categorical cross entropy losses. Another neural network will be developed as a convolutional neural network with different hyperparameters, such as the number of filters used for each layer and kernel sizes. The hyperparameters will again be optimized in a similar fashion as the before. Lastly, the results of both neural networks will be compared.

Sec. II. Theoretical Background

In order to approach this assignment, several key concepts pertaining to artificial neural networks must be understood. An artificial neural network is a model (a complicated function) that gives the relationships between the independent variables (x) and dependent variables (y). This is similar to a linear regression where the model is a linear function. Based on a biological neural network, the artificial neural network is made up of building blocks called threshold logic units or linear threshold units. Each unit is comprised of nodes ('neurons') that inputs and outputs a number that is scaled by the weight (w) of their connections to another node. The input layer of neurons just outputs the same number that was inputted. Given the inputs and weights as a vector, the weighted sum of the inputs can be written as a projection. After taking the weighted sum of the inputs, the output neuron will be active if the sum is greater than a certain threshold and inactive if the sum is less than the threshold. Furthermore, to account for the threshold shifting, an extra bias neuron (b) can be implemented in the input layer with an output of 1 (Eqn. 1).

$$z = \mathrm{w}^{\mathrm{T}}x = w_1x_1 + w_2x_2 + w_3x_3 + b \qquad \text{(Eqn. 1)}$$

The process of determining a node's output is carried out by what is known as the activation function, which can be denoted as Eqn. 2. There are various activation functions that can be used such as sigmoid functions and the rectified linear unit (ReLU) which zeros out all inputs that are negative and retain the positive entries alone. This function has the benefit of being continuous which is necessary to carry out gradient descent. Additionally, this process of taking the weighted sum and classifying the results based on a threshold is similar to linear discriminant analysis where the data is projected onto an optimal *w* vector and classified into different categories.

$$\mathrm{y} = \sigma(\mathrm{z}) = \begin{cases} 0, & z < threshold \\ 1, & z \geq threshold \end{cases} \qquad \text{(Eqn. 2)}$$

A layer of these linear threshold units is known as a perceptron. However, in order to solve more complex and difficult classification or regression problems, multiple perceptrons can be strung together, forming a multiplayer perceptron (MLP). An MLP first consists of an input layer and lastly an output layer. Any layers in between are referred to as hidden layers, where more hidden layers allow for solving more complex problems. The number of neurons per layer and the number of layers are called the width of the layer and depth of the network, respectively. The MLP is also referred to as a feed-forward, fully connected neural network if every neuron in a given layer is connected to every neuron in the previous layer. The MLP with one hidden layer can then be written as

$$\mathrm{y} = \sigma(\mathrm{A}_2\sigma(\mathrm{A}_1\mathrm{x}_1 + \mathrm{b}_1) + \mathrm{b}_2) \qquad \text{(Eqn. 3)}$$
$$\mathrm{x}_2 = \sigma(A_1x_1 + b_1) \qquad \text{(Eqn. 4)}$$
$$y = \sigma(A_2x_2 + b_2) \qquad \text{(Eqn. 5)}$$

where x_1 is the input layer, x_2 is the hidden layer, y_1 is the output layer, b_1 and A_1 are the bias and weight matrix between the input and hidden layer, respectively. b_2 and A_2 are the bias and weight matrix between the hidden and output layer, respectively. For classification purposes, it is common to use a softmax function (Eqn. 6), which is essentially a multiclass logistic function, for the output layer, where the linear output values (Eqn. 3) are rescaled as probability values that fall between 0 and 1.

$$\mathrm{p} = \sigma(y) = \frac{1}{\sum_{j=1}^{1} e^{y_j}} \begin{pmatrix} e^{y_1} \\ e^{y_2} \\ \vdots \\ e^{y_\mathrm{m}} \end{pmatrix} \qquad \text{(Eqn. 6)}$$

Afterwards, the error of the model is computed using a loss function (or objective function) that outputs a loss value. The loss function used for this assignment is a type of categorical cross-entropy function (log loss) that measures the performance of a model that outputs probabilities (Eqn. 7). Taking a logarithm is also desirable as this leads to the optimization problem being convex with only one local minimum.

$$\mathrm{Cross-entropy\ loss} = -\frac{1}{\mathrm{C}}\sum_{\mathrm{c}=1}^{\mathrm{C}} \mathrm{y}_{\mathrm{o,c}} \log(p_{o,c}) + \lambda\|w\|^2 \qquad \text{(Eqn. 7)}$$

where *C* is the number of classes, *y* is a binary (0 and 1) indicator if class label *c* is the correct classification for observation *o*, and *p* is the predicted probability observation *o* is of class *c*. The right side of Eqn. 7 is the L2 regularization to decrease overfitting, where λ is a regularization constant and *w* is the weights. We can then find the weights and biases that minimizes this loss function through training the network (optimization). To do this, we take the partial derivative of the loss function with respect to every weight and biases of the network and set it equal to zero as you would do in calculus for minimization. This process is most commonly carried out using the technique called gradient descent, where you start with initialized random guesses for the parameters and iterate towards the direction of the steepest gradient (which direction and how fast) by a distance set by the learning rate (δ - how long) until you reach the lowest point. This point is reached when the gradient is very close to zero. There are various ways to start this technique but the method we will be focusing is called Mini-batch (Stochastic) gradient descent which uses batches of randomly selected data, performing one step of gradient descent. Furthermore, the partial derivatives (gradients) can be computed using the chain rule.

Another type of neural network is called the convolution neural network. Although the architecture is similar to the feed-forward, fully connected neural network as previously mentioned, the input of the neural network is different in that they are not flattened (e.g., it takes in the full 2D image instead of flattening it). The general idea for a convolutional neural network is that there are a hierarchy of layers of neurons, with the lowest layer having a receptive field that looks for simple features of the original image, and the layer above that has its receptive field that looks for more detailed features in the previous layer and so on. This process continues to extract higher level features of the original image. There are two main layer types, the convolutional and pooling layers. For a convolutional layer, each neuron uses a n x n receptive field where a matrix of weights (filter/convolutional kernel) is used to take a matrix multiplication with the original image or previous layer. Moreover, an activation function can be used after each convolution. This process is done with all neurons in the layer with their receptive layers sliding by a defined stride length. All in all, this produces a feature map with certain features extracted from the previous layer. Furthermore, you can retain the same feature map size by using zero padding to allow the center of the receptor fields at the edges of the original image. One inefficiency of convolutional neural networks is that there are multiple feature maps in a convolutional layer, leading to high ram cost. To ameliorate this, we can take subsamples of each layer, reducing the dimensionality. This is called the pooling layer (e.g., max pooling or average pooling), where either a max pixel value or an average of the pixel values of a set grid can be used. After enough of the high-level details are extracted, the image is flattened and inputted into dense layers for classification using the previously mentioned training procedure.

Sec. III. Algorithm Implementation and Development

Starting with **Part I** in **Appendix B. Python codes**, packages and modules needed for this assignment, such as numpy, tensorflow, matplotlib.pyplot, pandas, and sklearn.metrics, were imported. Afterwards, the Fashion-MNIST dataset was loaded in from the Keras application programming interface (API), which is a library that can run on top of TensorFlow. The Fashion-MNIST dataset contains 60,000 training images (*X_train_full*) and 10,000 test images (*X_test*) with their respective labels (*y_train_full* and *y_test*). Each image size is 28 x 28 with label values from 0 to 9, which corresponds to 10 different fashion items. Before beginning to develop the neural networks, the dataset will be preprocessed, allocating the first 5,000 images from the training dataset to be used as validation data (*X_valid*), using the rest of the 55,000 images as training data (*X_train*) with their respective labels (*y_valid* and *y_train*). Furthermore, the datatype of each image was converted from uint8 values to floats between 0 and 1 by dividing each of the pixel values by 255.0.

The first model constructed was a feed-forward, fully connected neural network and its architecture was created as mentioned in the above section. The model itself was created using the *tf.keras.models.Sequential* function which creates linear stacks of layers. There are five layers within this model. The input layer is created by using the function *tf.keras.layers.Flatten* with input shape 28 x 28. This creates 784 nodes for the input layer that is fully connected to three hidden layers that were created using the *tf.keras.layers.Dense* function with the ReLU function as the activation function and the L2 regularization constant equal to 0.00001. The three hidden dense layers had 500, 250, and 125 nodes sequentially. The last hidden layer was fully connected to the output layer that has 10 nodes that stores the 10 probabilities corresponding to the 10 possible fashion item outputs and uses the softmax function (Eqn. 6) as its activation function. After creating the model architecture, *model.compile* was used to set the loss function to a sparse (list of ints) categorical cross-entropy function and the optimizer that implements the Adam algorithm, a stochastic gradient descent method, with learning rate of 0.0001. The probability of how many predictions were right were reported as an accuracy metric. *Model.fit* was used to obtain training history with training and validation data as inputs along with 20 epochs that defines the number iterations the training process takes.

The hyperparameters defined above regarding the depth of the network, width of the layers, learning rate, and the regularization parameter were adjusted in order to yield high and similar training and validation accuracies while minimizing their respective loss. The width of each layer was formed in a pyramid shape, which is a commonly used method, which resulted in the best validation accuracy when using aforementioned width values. The have a very small regularization constant and a smaller learning rate also lead to higher accuracies. 20 epochs were used to fully train the model.

The resultant training history was then plotted using *pd.DataFrame.plot*. Afterwards, the model was evaluated using the function *model.evaluate* that outputs the loss value and accuracy of the test data using the model. Lastly, a confusion matrix for the training data was developed using the *confusion_matrix* with rows representing predictions (using the *model.predict_classes* with the test data) and columns representing the real labels.

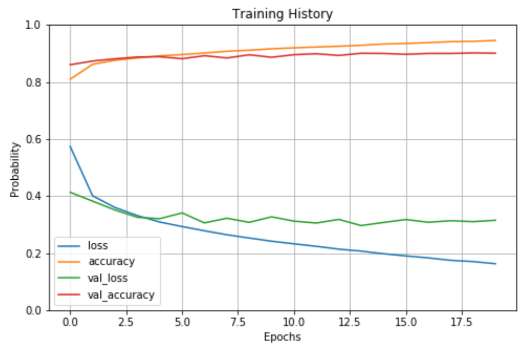
For **Part II** in **Appendix B. Python codes**, the same procedure was repeated but with a convolutional neural network. Furthermore, the training, validation, and testing data was formatted to 28 x 28 x 1 as using *np.newaxis* since TensorFlow requires this format. The convolutional neural network architecture was developed by using the *tf.keras.models.Sequential* function with eight layers ordered as convolutional input, max pooling, convolutional, max pooling, convolutional, flattened, dense, and a last dense output layer. The convolutional layers were made using the *tf.keras.layers.Conv2D* function with ReLU activation function and same padding, which conserves the size of the output feature map from the input feature map. The input convolutional layer had an input shape of 28 x 28 x 1, with 16 features that have the size 3 x 3. The subsequent convolutional layers were the same but with 32 features and 64 features, in that order. All of the max pooling layers were created using *tf.keras.layers.MaxPooling2D* with a pool size of 2 x 2. The dense layers were the same as above but with a L2 regularization constant of 0.0001 and width of 64 and 10, in that order. The learning rate was also set to the default value of 0.001. The flattened layer was created using the *tf.keras.layers.Flatten* function.

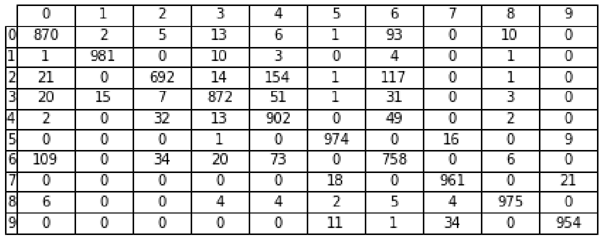
Along with the previously mentioned hyperparameters, the number of convolutional layers, kernel sizes, padding options, strides, and pool sizes. Through various iterative trials, increasing the feature number for the convolutional layers by twice its previous layer was deemed to have higher accuracies and lower loss values. Furthermore, a 3 x 3 kernel size, same padding options for all convolutional layers, and a 2 x 2 pool size seem to result in sufficient accuracy and loss values.

Sec. IV. Computational Results

Part I) From Figure 1, (left) it is evident that the training accuracy and validation accuracies were similar, but the training accuracies were a bit higher by 3 – 4 % towards the latter epochs. This shows that there was some overfitting of the data. Furthermore, the training loss values were almost 10 – 15 % lower than that of the validation data set. The optimal epoch had**: training loss: 0.2073 | training accuracy: 0.9283 | val_loss: 0.2966 | val_accuracy: 0.9004.** (right)The confusion matrix showed that the actual label of 4 (coat) was being confused the most with the predicted value of 2 (pullover), which is reasonable. Testing this model, **the loss: 0.346 and accuracy: 0.8939.**

Part 2) From Figure 2, (left) it is evident that the training accuracy and validation accuracies were more similar and higher than the previous model. However, the training accuracies were a bit higher by 2 – 2.5 % towards the latter epochs. This shows that there was some overfitting of the data. Furthermore, the training loss values were almost 10 – 15 % lower than that of the validation data set. The optimal epoch had**: training loss: 0.2077 | training accuracy: 0.9354 | val_loss: 0.2482 | val_accuracy: 0.9232.** (right) The confusion matrix showed that the actual label of 0 (T-shirt) was being confused the most with the predicted value of 6 (shirt), which is reasonable. Testing this model, **the loss: 0.295 and accuracy: 0.9161.**



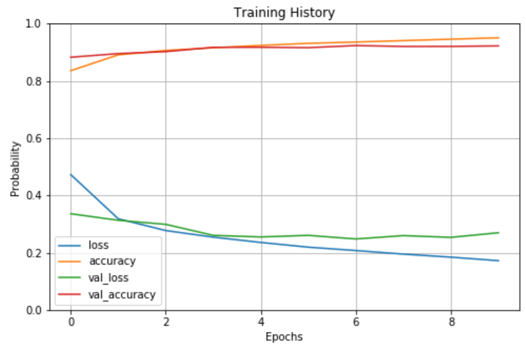


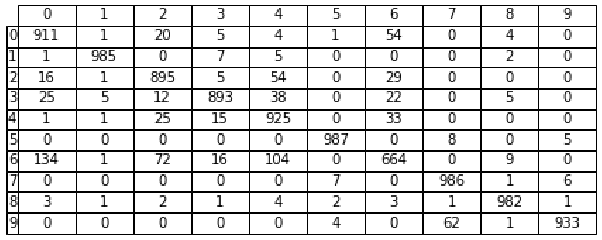
	0	1	2	3	4	5	6	7	8	9
0	870	2	5	13	6	1	93	0	10	0
1	1	981	0	10	3	0	4	0	1	0
2	21	0	692	14	154	1	117	0	1	0
3	20	15	7	872	51	1	31	0	3	0
4	2	0	32	13	902	0	49	0	2	0
5	0	0	0	1	0	974	0	16	0	9
6	109	0	34	20	73	0	758	0	6	0
7	0	0	0	0	0	18	0	961	0	21
8	6	0	0	4	4	2	5	4	975	0
9	0	0	0	0	0	11	1	34	0	954

Figure 1. (*left*) Plot of the training history of 20 epochs using the feed-forward, fully connected neural network, showing training loss and accuracy and validation loss and accuracy. The y axis are values from 0 to 1 (probability). (*right*) Confusion matrix with rows equal to predictions, columns equal to actual labels, and diagonal values showing the number of correctly classified images.

Sec. V. Summary and Conclusions

Looking at the computational results, it seems that the convolutional neural network was slightly better at correctly classifying the fashion item images than the feed-forward, fully connected neural network. Although I was not able to achieve a validation accuracy above 95%, I was able to obtain a ~ 92% validation accuracy using a convolutional neural network, which is ~ 2% higher than that of the first model. Furthermore, for both models, overfitting the data seemed to be a slight problem. In terms of losses, the regularization number seem to work sufficiently to get a downward curving line. However, the there seemed to be a wider variation in the loss values in the first model than the second model. Furthermore, the testing accuracy for the convolutional neural network was about 2% higher than that of the first model. The increased accuracy of the second model could be due to the models architecture of successive iterations of convolutional and pooling layers that are able to extract higher levels of detail in the image than the first data, allowing for better and more accurate classification. All in all, this assignment allowed for the development of two different neural network models to classify fashion item images. The hyper parameters were adjusted in order to optimize loss and accuracy values. However, with more test data, better combinations of hyperparameters may lead to better results.





	0	1	2	3	4	5	6	7	8	9
0	911	1	20	5	4	1	54	0	4	0
1	1	985	0	7	5	0	0	0	2	0
2	16	1	895	5	54	0	29	0	0	0
3	25	5	12	893	38	0	22	0	5	0
4	1	1	25	15	925	0	33	0	0	0
5	0	0	0	0	0	987	0	8	0	5
6	134	1	72	16	104	0	664	0	9	0
7	0	0	0	0	0	7	0	986	1	6
8	3	1	2	1	4	2	3	1	982	1
9	0	0	0	0	0	4	0	62	1	933

Figure 2. (*left*) Plot of the training history of 10 epochs using the convolutional neural network, showing training loss and accuracy and validation loss and accuracy. The y axis are values from 0 to 1 (probability). (*right*) Confusion matrix with rows equal to predictions, columns equal to actual labels, and diagonal values showing the number of correctly classified images.

Appendix A. Python functions used and brief implementation explanation

- **Flatten:** vectorizes 2D images into a single vector with size equal to the total number of pixels.
- **Dense:** generates fully connected layers of neurons.
- **MaxPool2D:** reduces the dimensionality of a feature map based on the pool size.
- **Conv2D:** generates a feature map based on a set filter
- **Sequential:** establish ordered sequence of neural layers.
- **Compile:** builds the models and assign the loss function along with optimizer
- **Fit:** trains the model using training data with specific number of epochs.
- **Predict_classes:** runs the given data through the model and makes predicted values.
- **Confusion_matrix:** creates matrix where rows are real labels and columns are predicted labels.

Appendix B. Python codes

```
# Part I
# Fully-connected Neural Network
# Load in packages and modules
import numpy as np
# Tensorflow 2 with Keras built in - Training platform for neural networks
import tensorflow as tf
# used for plotting
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.metrics import confusion_matrix


# In[2]:


# Load in fashion MNIST dataset
# Keras - does not do all of the computations - it is an api that sits ontop
# of Tensorflow and adds easy to use functionalities
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()


# In[3]:


# Check X_train_full size
X_train_full.shape


# In[4]:


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[5]:


y_train_full[:9]
```

```
# In[6]:



# Preprocessing
# 5000 images for validation data as floats
X_valid = X_train_full[:5000] / 255.0
# 55000 images for training as floats
X_train = X_train_full[5000:] / 255.0
# 10000 test images floats
X_test = X_test / 255.0

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


# In[7]:



# Create our model
# Hyperparameters - parameters about network that you adjust before
# training procedure
from functools import partial

# Hyperparamter - regularization: help prevent overfitting - adds something
# to loss function - constant times the two norm of your weight matrices
# where the weights are relatively small - motivation:
my_dense_layer = partial(tf.keras.layers.Dense, activation='relu',
                         kernel_regularizer=tf.keras.regularizers.l2(0.00001))

# From one layer, feeds into the next layer: feed-forward
model = tf.keras.models.Sequential([
    # convert image into flattened list
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    # can increase layer width and depth
    my_dense_layer(500),
    my_dense_layer(250),
    my_dense_layer(125),
    # output layer of 10 possible fashion item outputs; get ten probabilities
    my_dense_layer(10, activation='softmax')
])


# In[8]:



# Options for training our model
# cross entropy loss for classification; sparse_categorical = list of int
model.compile(loss='sparse_categorical_crossentropy',
             # learning rate = determines how much you will change each of
             # weight and biases per step of the optimization
             optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
             # how many you got right as probability
             metrics=['accuracy'])


# In[9]:


# epochs = how many optimization steps you want to use
history = model.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_valid))


# In[14]:


pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.title('Training History')
plt.xlabel('Epochs')
plt.ylabel('Probability')
plt.savefig('training_history_1.png')
plt.show()
# no over-fitting when training loss goes below validation loss
# want validation loss at the bottom


# In[11]:


# Make confusion matrix for training data
y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)
# row represent predictions, columns represent actual labels
# Diagonals shows which values we got right
# Useful for seeing where our algorithm went wrong - see off diagonals


# In[12]:
```

```
model.evaluate(X_test, y_test)
# returns loss, accuracy


# In[13]:


y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)
print(conf_test)


# In[15]:


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='cent
er')
fig.tight_layout()
plt.savefig('conf_mat_1.png')


# In[ ]:


# Part II Convolutional neural network - CNN


# In[16]:


fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()


# In[17]:


# Tensorflow wants 55000,28,28,1 for each convolutional layers (1 for grayscale)
```

```
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[18]:


# Create CNN Model
# Don't have to flatten each image
from functools import partial

my_dense_layer = partial(tf.keras.layers.Dense, activation="relu", kernel_regularizer=tf.keras.regularizer
s.l2(0.0001))
my_conv_layer = partial(tf.keras.layers.Conv2D, activation="relu", padding="same")
# zero padding = size of the feature map is the same size as the input
model = tf.keras.models.Sequential([
    # my_conv_layer(number of filters, size of filter)
    my_conv_layer(16,3,padding="same",input_shape=[28,28,1]),
    # 2 x 2 pool size with stride 2
    tf.keras.layers.MaxPooling2D(2),
    my_conv_layer(32,3),
    tf.keras.layers.MaxPooling2D(2),
    my_conv_layer(64,3),
    tf.keras.layers.Flatten(),
    my_dense_layer(64),
    my_dense_layer(10, activation="softmax")
])


# In[19]:


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

```
# In[20]:


history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid,y_valid))


# In[23]:


pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.title('Training History')
plt.xlabel('Epochs')
plt.ylabel('Probability')
plt.savefig('training_history_2.png')
plt.show()


# In[24]:


y_pred = model.predict_classes(X_train)
conf_train = confusion_matrix(y_train, y_pred)
print(conf_train)


# In[25]:


model.evaluate(X_test,y_test)


# In[26]:


y_pred = model.predict_classes(X_test)
conf_test = confusion_matrix(y_test, y_pred)
print(conf_test)


# In[27]:


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='cent
er')
fig.tight_layout()
plt.savefig('conf_mat_2.png')
```