MODULE 7: Digit Recognizer using Neural Networks

Claire Markey, Julia Granito, Manny Hurtado, and Steve Desilets

MSDS 422: Practical Machine Learning

May 14th, 2023

Introduction

Identifying and constructing handwritten images constitutes a highly studied application for classification algorithms. Building on a previous analysis using tree-based and clustering methods, large scale data consisting of single digits were classified as their corresponding number using neural networks. Varying experimental designs were used to assess the impact of neural network architecture on classification accuracy.

Methods

Kaggle data containing images of hand-drawn digits from zero through nine were downloaded and analyzed using Jupyter Notebooks (AstroDave and Cukierski, 2012). Neural Network methods were implemented using the original pixel data to construct models that classified digits given the input pixel data. Design of experiments (NxM) was used to study the differences and results from neural network architectures containing different numbers of layers and nodes per layer. Subsequent experiments examined the impact of learning rates and initial learning rates on model accuracy and computation time. Principal component analysis and Kernel PCA were used to reduce the dimensionality of the data before re-running the experiment. Model performance metrics were examined to assess these modifications.

Results and Insights

First, examination of the training and test data confirmed that there was no data missingness to handle using data imputation methods. An exploratory data analysis (EDA) revealed that the training and testing datasets contain numeric data for 784 pixels (28x28 images) for each of the hand-drawn digits in the data. Pixel-values ranged from 0 to 255, where each value indicates the lightness or darkness of that pixel.

Then, four sequential models (two hidden layers, 20 nodes each) were explored. Two models were created using a hyperbolic tangent activation function and two models used a rectified linear unit (ReLU) activation function. Within these sets, one uses a RMSprop optimization method, and the other uses an Adagrad method. All models employed a sparse categorical cross-entropy loss function and measured model accuracy. This approach facilitated an exploration of the optimal learning rates and epoch values. The findings suggest that an RMSprop optimization method better reduces loss compared to Adagrad. The two models that utilized RMSprop differed by activation functions and learning rates. Model 1 uses hyperbolic tangent activation with a learning rate of .01, and Model 2 uses ReLu activation with a learning rate of .001. Model 1 has a testing accuracy rate of .939 and loss of .225. A visual inspection of these values over 30 epochs revealed greatest efficacy for three epochs. Model 2 exhibited a testing accuracy rate of .952 and loss of .205; likewise, three epochs appears ideal based on a visual inspection of the results.

Next, a 3x2 experimental design was conducted to evaluate the performance of our network architectures. Our design uses a Multi Layer Processor classification algorithm with 2, 3, and 5 layers that each contain 10 or 20 nodes. The following parameters are used: activation function (ReLU), default optimization solver (sg-based, adam), and constant learning rate (.001) to isolate the impact of node and layer sizes on performance metrics. The accuracy of these models in predicting digits for the training dataset ranged from .907 to .943, with the optimal model containing 5 layers with 20 nodes each. A confusion matrix is included in the appendix. We then applied the model to the testing dataset, for which the model achieved an accuracy of .9421 in Kaggle.

A follow-up experiment with a 2x2 design was conducted to assess whether setting different initial learning rates or learning rate schedules for weight updates improved predictive ability. For this experiment, all models contained constant or adaptive learning rate weight update schedules and initial learning rates of .01 or .001. Consistent with the results of our previous experiment, each of the neural networks in this experiment utilized that layer and node architecture consisting of 5 hidden layers with 20 nodes each. The four neural networks were constructed and tested using a five-fold cross-validation design. Model computation time and performance metrics (training dataset prediction accuracy, testing dataset prediction accuracy, precision and recall, and confusion matrices) were measured and analyzed. The model that resulted in the best training and testing dataset prediction accuracies - .9995 and .9440, respectively - was the neural network leveraging adaptive learning rates and an initial learning rate of .01. Notably, this model is constructed using the second least compute time of 6 minutes and 39 seconds. (The model constructed most quickly was the model with constant learning rates and an initial learning rate of .01, which achieved a computation time, training dataset prediction accuracy, and testing dataset prediction accuracy of 3 minutes and 16 seconds, .9920, and .9405, respectively). These results suggest that a neural network with 5 layers of 20 nodes each with adaptive learning rates and an initial learning rate of .01 performs well. Building on the results of our previous analysis with this dataset, this method performed better than our prior K-Means analysis, which yielded an accuracy of only .921.

In the final stage of our analysis, transformations of features using principal components were considered (334 principal components, which explain 95% of the variance of our dataset). In addition, Kernel PCA was considered, in the case that preserving the distance between points rather than variance of the dataset impacted the results. In line with previous experiments, PCA-driven classifiers were conducted using a 5-layer, 20-node architecture. Our PCA-based model yielded an accuracy score of .995.

These experimental results suggest that neural networks are effective models for classifying digits, and that strategic design of neural net architecture may improve the models' predictive abilities. High accuracies of .99 and .94 on the training and testing data demonstrate the strength of these models. The fact that the structure of the neural network resulted in accuracy ranges from .907 to .995 suggests that architectural choices (layer count, node count, initial learning rate, learning rate schedules for weight updates, and principal component analysis inclusion) significantly impact the accuracy of neural networks. A further analysis could explore varying activation functions in greater depth.

References

AstroDave, and Will Cukierski. 2012. “Digit Recognizer.” *Kaggle.* https://www.kaggle.com/c/digit-recognizer

Appendix 1 - Python Code and Outputs

Data Preparation

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Import Training Data

In [2]:
```
import numpy as np
import pandas as pd
# load training data
digit_training_data = pd.read_csv('train.csv')

# show first rows of the data
digit_training_data.head(100)
# show number of columns and rows
digit_training_data.shape
```

Out[2]: (42000, 785)

In [3]:
```
digit_training_data.head(10)
```

Out[3]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775
0	1	0	0	0	0	0	0	0	0	0	...	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0
3	4	0	0	0	0	0	0	0	0	0	...	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0
5	0	0	0	0	0	0	0	0	0	0	...	0	0
6	7	0	0	0	0	0	0	0	0	0	...	0	0
7	3	0	0	0	0	0	0	0	0	0	...	0	0
8	5	0	0	0	0	0	0	0	0	0	...	0	0
9	3	0	0	0	0	0	0	0	0	0	...	0	0

10 rows × 785 columns

Investigation of Missing Data and Outliers in Training Data

In [3]:
```
# find null counts, percentage of null values, and column type
null_count = digit_training_data.isnull().sum()
null_percentage = digit_training_data.isnull().sum() * 100 / len(digit_training_data)
```

```
column_type = digit_training_data.dtypes

# show null counts, percentage of null values, and column type for columns with more t
null_summary = pd.concat([null_count, null_percentage, column_type], axis=1, keys=['Mi
null_summary_only_missing = null_summary[null_count != 0].sort_values('Percentage Miss
null_summary_only_missing
```

Out[3]:

Missing Count	Percentage Missing	Column Type

The above analysis displays that there is no missing data in the digit recognizer training dataset.

Import Testing Data

In [4]:
```
# import test dataset
digit_testing_data = pd.read_csv('test.csv')

# show first ten rows of the data
digit_testing_data.head(10)
# show number of columns and rows
digit_testing_data.shape
```

Out[4]:

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775
0	0	0	0	0	0	0	0	0	0	0	...	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0
5	0	0	0	0	0	0	0	0	0	0	...	0	0
6	0	0	0	0	0	0	0	0	0	0	...	0	0
7	0	0	0	0	0	0	0	0	0	0	...	0	0
8	0	0	0	0	0	0	0	0	0	0	...	0	0
9	0	0	0	0	0	0	0	0	0	0	...	0	0

10 rows × 784 columns

Out[4]:
```
(28000, 784)
```

Investigation of Missing Data and Outliers in Training Data

In [5]:
```
# find null counts, percentage of null values, and column type
null_count = digit_testing_data.isnull().sum()
null_percentage = digit_testing_data.isnull().sum() * 100 / len(digit_training_data)
column_type = digit_testing_data.dtypes

# show null counts, percentage of null values, and column type for columns with more t
null_summary = pd.concat([null_count, null_percentage, column_type], axis=1, keys=['Mi
```

```
null_summary_only_missing = null_summary[null_count != 0].sort_values('Percentage Miss
null_summary_only_missing
```

Out[5]:

Missing Count	Percentage Missing	Column Type

The above analysis displays that there is no missing data in the digit recognizer test dataset.

3 x 2 Experimental Design

We used MLP Classifier and GridSearch cross-validation (5-folds) to find the optimal number of layers (2, 3, or 5) and nodes (10 or 20). We used the default activation function (relu), the default optimization solver, the adam solver and the default learning rate 0.001 (as recommended by sklearn's documentation).

In [7]:

```
# Import libraries
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, (

# Extract predictors and outcome (label variable)
X_train = digit_training_data.copy(deep=True)
X_train.drop(['label'], axis=1, inplace=True)
y_train = digit_training_data['label']


# Standardize the features
xscaler = StandardScaler()
X_train = xscaler.fit_transform(X_train)

# Initialize MLP Classifier
mlp_class = MLPClassifier(random_state=1)

# Create paramater grid with hyperparameters to tune, use default adam solver so doen
param_grid = {
    'hidden_layer_sizes': [(10,10), (20,20), (10,10,10), (20,20,20), (10,10,10,10,10),
}

# Kfold cv with 5 splits for GridSearch
cv = KFold(n_splits=5, shuffle=True, random_state=1)

# Create the GridSearchCV with kfold=5 object and fit it to the training data
grid_search = GridSearchCV(mlp_class, param_grid, cv=cv, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Print the best hyperparameters found
print("Best Hyperparameters:", grid_search.best_params_)

# Save the best estimator
best_model = grid_search.best_estimator_

# Save dictionary of mean accuracy scores from models into 'scores' variable
dict_results = grid_search.cv_results_
scores = dict_results['mean_test_score']
```

```
# Use the best model to predict using training data
y_pred_train = best_model.predict(X_train)

# evaluate the model on the training data
accuracy_train = accuracy_score(y_train, y_pred_train)
print("Training Accuracy:", accuracy_train)
print("Training Classification Report:", classification_report(y_train, y_pred_train))

# Create the confusion matrix of the predictions
cm = confusion_matrix(y_train, y_pred_train)
ConfusionMatrixDisplay(confusion_matrix=cm).plot();

# Save layer and node data
Layers = (2,2,3,3,5,5)
Nodes = (10,20,10,20,10,20)

# create dataframe with MLP details and scores for each mode and layer count tested
MLP_Scores = pd.DataFrame({'Layers' : Layers, 'Nodes' : Nodes, 'Mean Training Accuracy
MLP_Scores
```

```
C:\Users\cmark\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\cmark\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\cmark\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\cmark\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\cmark\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\cmark\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\cmark\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\cmark\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\cmark\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:686: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
```

Out[7]:

```
▸ GridSearchCV
▸ estimator: MLPClassifier
  ▸ MLPClassifier
```

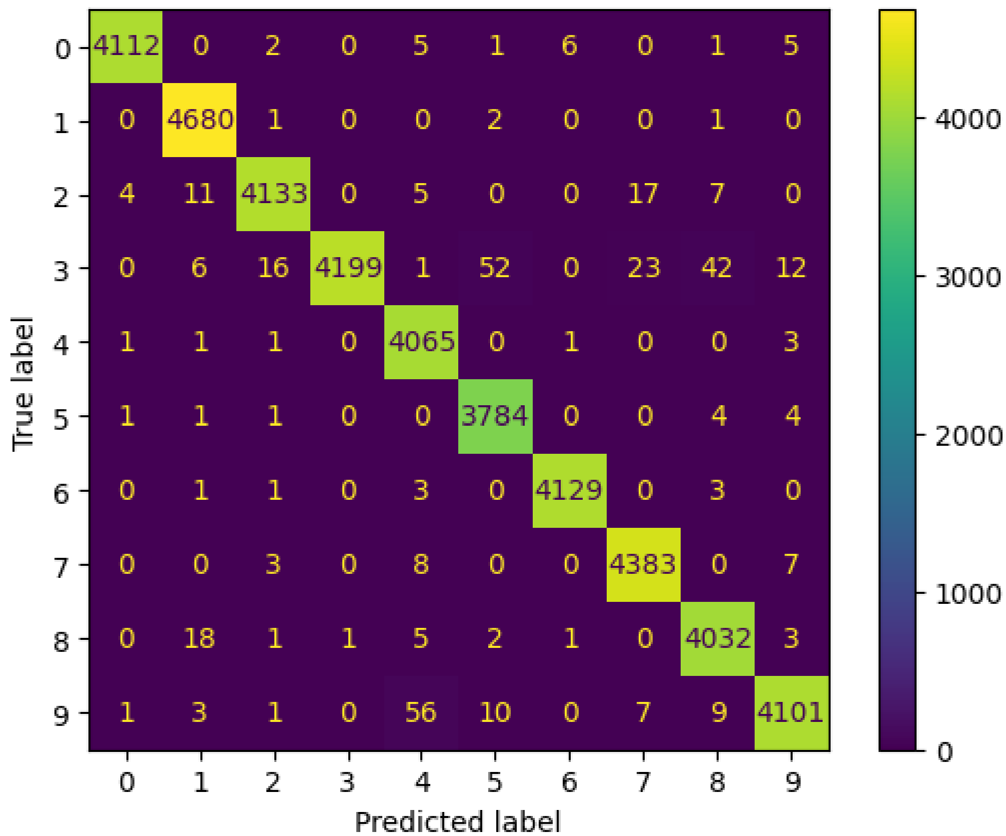
```
Best Hyperparameters: {'hidden_layer_sizes': (20, 20, 20, 20, 20)}
Training Accuracy: 0.990904761904762
Training Classification Report:               precision    recall  f1-score   support

           0       1.00      1.00      1.00      4132
           1       0.99      1.00      1.00      4684
           2       0.99      0.99      0.99      4177
           3       1.00      0.97      0.98      4351
           4       0.98      1.00      0.99      4072
           5       0.98      1.00      0.99      3795
           6       1.00      1.00      1.00      4137
           7       0.99      1.00      0.99      4401
           8       0.98      0.99      0.99      4063
           9       0.99      0.98      0.99      4188

    accuracy                           0.99     42000
   macro avg       0.99      0.99      0.99     42000
weighted avg       0.99      0.99      0.99     42000
```

Out[7]:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x2566eb85bb0>
```

Out[7]:

	Layers	Nodes	Mean Training Accuracy
0	2	10	0.912214
1	2	20	0.939976
2	3	10	0.910690
3	3	20	0.938667
4	5	10	0.907024
5	5	20	0.943143



We see that a 5 layer and 20 node model yields the highest mean training accuracy score (0.943).

Apply MLPClassifier to Test Data

In [8]:
```
# Create a dataframe for predictor variables in the test dataframe for mlpclass model
mlpclass_testing_x = digit_testing_data.copy(deep=True)

# Standardize the features using same scaler as training data
mlpclass_testing_xscale = xscaler.transform(mlpclass_testing_x)

# Apply the mlpclass model to the test dataset
mlpclass_test_ypred = best_model.predict(mlpclass_testing_xscale)

# Put the kmeans predictions into a Pandas dataframe
prediction_df_mlpclass = pd.DataFrame(mlpclass_test_ypred, columns=['Label'])

# Add the ID column to the front of the mlpclass predictions dataframe
ImageId_series = pd.Series(range(1,28001))
prediction_df_mlpclass.insert(0, 'ImageId', ImageId_series)

# Output predictions to csv
#prediction_df_mlpclass.to_csv('test_predictions_mlpclass_v2.csv', index=False)
```

Let's display the Kaggle results from the application of the MLP Classifier model on the test dataset

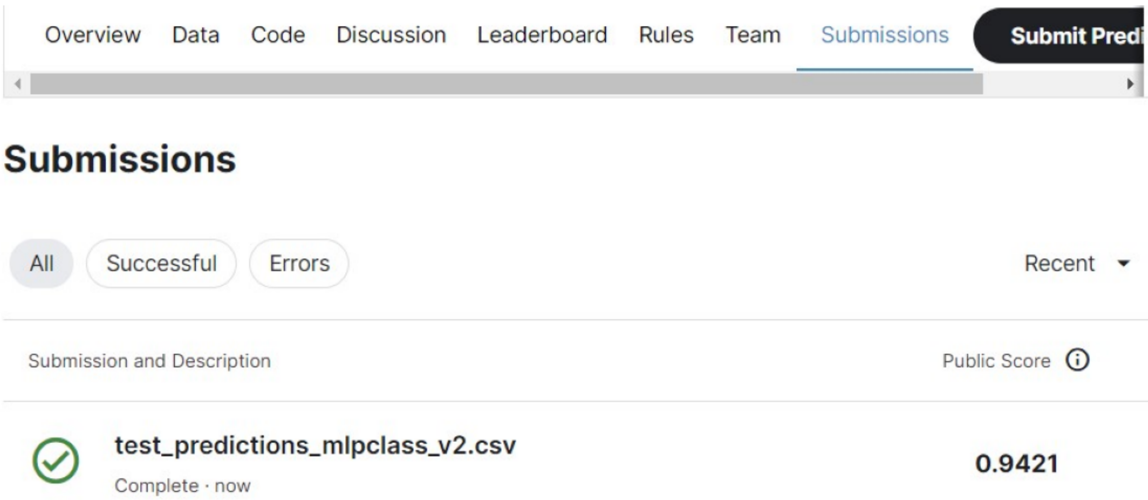
In [10]:
```
# Display the kaggle results associated with the MLP Classifier Model
import matplotlib.pyplot as plt
plt.figure(figsize = (15, 15))
```

```
kaggle_results = plt.imread('Digit_mlpclass_v2.jpg')
plt.imshow(kaggle_results)
plt.axis("off")
plt.show()
```

Out[10]: <Figure size 1500x1500 with 0 Axes>

Out[10]: <matplotlib.image.AxesImage at 0x1847d9f22b0>

Out[10]: (-0.5, 984.5, 414.5, -0.5)



2 x 2 Experimental Design for Learning Rate Tuning

First, let's load the required packages.

In [6]:
```
#pip install tensorflow
import tensorflow as tf
from tensorflow import keras
```

Next let's split the training data into training and validation sets

In [7]:
```
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# Create a copy of the training dataframe
nn_train_df = digit_training_data.copy(deep=True)

sc = MinMaxScaler() #Initialize scaling of data

nn_train_df.drop(['label'], axis=1, inplace=True) #drop the label column from the df

nn_train_x = nn_train_df #set df without label as x
nn_train_y = digit_training_data['label'] #set y a the label column

sc.fit(nn_train_x)
normalized = sc.transform(nn_train_x)

# Convert scaled data from numpy array into dataframe
nn_training_features = list(nn_train_df.columns.values)
nn_training_scaled_df = pd.DataFrame(normalized, columns=nn_training_features)
```

```
# Split the Kaggle training data into training and validation components
nn_x_train, nn_x_validation, nn_y_train, nn_y_validation = train_test_split(nn_trainir
                                                                    nn_train_y,
                                                                         test_size=
                                                                        random_stat
```

Out[7]: MinMaxScaler()

In [8]:
```
nn_x_train.shape

nn_x_validation.shape

nn_y_train.shape

nn_y_validation.shape
```

Out[8]: (31500, 784)

Out[8]: (10500, 784)

Out[8]: (31500,)

Out[8]: (10500,)

In [9]:
```
nn_training_scaled_df.describe()
nn_training_scaled_df.shape
nn_training_scaled_df.head(20)
```

Out[9]:

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	
count	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	...	42
mean	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
std	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
25%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
50%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
75%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
max	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	

8 rows × 784 columns

Out[9]: (42000, 784)

Out[9]:

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
11	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
14	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
15	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
17	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
18	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
19	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0

20 rows × 784 columns

Repeat Scaling for test data:

In [10]:

```
# Create a copy of the training dataframe
nn_test_df = digit_testing_data.copy(deep=True)

sc = MinMaxScaler() #Initialize scaling of data

nn_test_x = nn_test_df #set df without label as x

sc.fit(nn_test_x)
normalized = sc.transform(nn_test_x)

# Convert scaled data from numpy array into dataframe
nn_test_features = list(nn_test_df.columns.values)
nn_test_scaled_df = pd.DataFrame(normalized, columns=nn_test_features)
```

Out[10]:

```
MinMaxScaler()
```

In [11]:
```
nn_test_scaled_df.describe()
nn_test_scaled_df.shape
nn_test_scaled_df.head(20)
```

Out[11]:

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	
count	28000.0	28000.0	28000.0	28000.0	28000.0	28000.0	28000.0	28000.0	28000.0	28000.0	...	28
mean	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
std	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
25%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
50%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
75%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	
max	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	

8 rows × 784 columns

Out[11]:
```
(28000, 784)
```

Out[11]:

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
11	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
14	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
15	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
17	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
18	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0
19	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0

20 rows × 784 columns

Create a model using Sequential API utilizing hyperbolic tangent activation method

In [12]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[784,]),
    keras.layers.Dense(20, activation="tanh"),
    keras.layers.Dense(20, activation="tanh"),
    keras.layers.Dense(10, activation="softmax")
])
```

In [13]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 20)                15700

 dense_1 (Dense)             (None, 20)                420

 dense_2 (Dense)             (None, 10)                210

=================================================================
Total params: 16,330
Trainable params: 16,330
Non-trainable params: 0
_________________________________________________________________
```

In [14]:

```
model.layers
```

Out[14]:

```
[<keras.layers.reshaping.flatten.Flatten at 0x19cb9d21550>,
 <keras.layers.core.dense.Dense at 0x19cb9d21d30>,
 <keras.layers.core.dense.Dense at 0x19cb9d21970>,
 <keras.layers.core.dense.Dense at 0x19cb9dda4f0>]
```

Examine the weights and biases of Hidden Layer 1

In [15]:

```
weights, biases = model.layers[1].get_weights()

weights

weights.shape

biases

biases.shape
```

Out[15]:

```
array([[ 0.07326624,  0.04950942, -0.0630369 , ...,  0.02167481,
        -0.00245464,  0.07073659],
       [-0.08012948, -0.05141045,  0.08416586, ...,  0.02029417,
        -0.06791637,  0.05525993],
       [ 0.01694722,  0.04899427,  0.07274768, ..., -0.04581188,
         0.05995984,  0.00395826],
       ...,
       [-0.03618319,  0.00040896, -0.06808969, ...,  0.00653894,
         0.08358839, -0.04321414],
       [ 0.06863168,  0.03469124,  0.00118482, ...,  0.02018707,
        -0.06759581, -0.06775653],
       [ 0.03377858, -0.06785498, -0.07941253, ..., -0.07023329,
        -0.0207302 ,  0.07253633]], dtype=float32)
```

Out[15]:

```
(784, 20)
```

Out[15]:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0.], dtype=float32)
```

Out[15]:

```
(20,)
```

Examine the weights and biases of Hidden Layer 2

In [16]:
```
weights, biases = model.layers[2].get_weights()

weights

weights.shape

biases

biases.shape
```

Out[16]:
```
array([[-0.10111153,  0.09760392, -0.33700863, -0.01380053,  0.12705648,
         0.11644924,  0.185121  , -0.27162716, -0.11037886, -0.10487276,
         0.05030358, -0.15251108,  0.1423583 ,  0.27406353,  0.3503375 ,
        -0.04738346, -0.21044031,  0.15255487, -0.38132113, -0.17750156],
       [ 0.27691275, -0.03286865, -0.25426352,  0.33897436,  0.05676907,
        -0.15932441,  0.1299097 , -0.1729231 ,  0.3532918 ,  0.07407343,
        -0.0419105 , -0.3826936 , -0.0787648 ,  0.12646747,  0.14670545,
        -0.09610766, -0.3743185 ,  0.33871996,  0.33835232,  0.15350795],
       [-0.03372225, -0.16610663, -0.09282297,  0.1484366 , -0.03761545,
         0.31965107,  0.02658424,  0.16245556,  0.11692977,  0.04515153,
         0.19763869, -0.34725642,  0.2885599 ,  0.30835772,  0.33098644,
        -0.1257908 , -0.20100398,  0.31614578, -0.0337902 ,  0.06840962],
       [-0.21065176,  0.11045173, -0.3308252 , -0.06850389,  0.32132518,
        -0.08534351,  0.2601953 ,  0.2395891 ,  0.06594491, -0.3059823 ,
         0.08540481,  0.16726011,  0.14869606, -0.35130447, -0.0720765 ,
         0.34508985, -0.08676931, -0.23423693, -0.08622646,  0.23349237],
       [ 0.24407393, -0.13525215, -0.13917999,  0.3331762 ,  0.22026259,
        -0.04093623, -0.18133704, -0.08685768,  0.3258884 ,  0.20131063,
         0.16538835, -0.08359635,  0.07564485, -0.01465791,  0.37609422,
        -0.3142687 ,  0.28564572,  0.24114388,  0.21923995, -0.08999667],
       [ 0.18997687, -0.32838914,  0.13766903,  0.36137468, -0.14573514,
         0.00661647,  0.2517867 ,  0.3096199 , -0.20541161,  0.02566114,
         0.09162045,  0.32977015, -0.21452022, -0.1578287 ,  0.02607575,
         0.14219183, -0.11301404, -0.02808383,  0.2912758 ,  0.27010208],
       [-0.17156756, -0.0995304 ,  0.38275313,  0.2993217 ,  0.03866959,
        -0.1766427 , -0.3103913 , -0.1130012 ,  0.10880476, -0.02079746,
         0.29598796,  0.32831   ,  0.29044193,  0.09322631, -0.1283223 ,
         0.12148142, -0.2975244 , -0.33381072,  0.06333226, -0.03032592],
       [ 0.15905589,  0.024966  ,  0.04103199, -0.03465784, -0.21646146,
         0.32019395, -0.28985488,  0.15453988, -0.23766622, -0.13498613,
         0.19592494, -0.15406209, -0.17189084, -0.2039098 , -0.20484298,
         0.23120266, -0.22221881,  0.3646415 , -0.01598731, -0.2661074 ],
       [-0.1553385 , -0.3195664 , -0.16256303, -0.11805528,  0.2872395 ,
         0.0537107 ,  0.27575535,  0.3315776 , -0.26548404, -0.02339911,
         0.37016684, -0.32706416, -0.10795173, -0.10626397, -0.05499256,
        -0.3347077 , -0.26275262, -0.05077055,  0.24232048,  0.14758736],
       [-0.25257522,  0.13376606, -0.21144864,  0.37848264, -0.12353811,
         0.01333117, -0.18407711,  0.37045234, -0.3872074 ,  0.00348395,
        -0.00508317, -0.35632688,  0.233114  ,  0.1172235 ,  0.29343885,
         0.21179402,  0.07607618, -0.25807297, -0.31019795, -0.19959332],
       [-0.31011632, -0.14694183,  0.14171565, -0.08359027,  0.30723476,
         0.22921056,  0.19840288, -0.330117  ,  0.15694124, -0.05306426,
         0.38329583, -0.07528242, -0.33658153, -0.37380058, -0.28540176,
        -0.08235043, -0.01384422, -0.09962413,  0.01670626, -0.08901343],
       [-0.06747931,  0.211734  ,  0.2265063 ,  0.14301556,  0.02635422,
        -0.25727618, -0.02984363,  0.00281829,  0.08872312,  0.33195478,
        -0.2374964 ,  0.24085337,  0.09390861,  0.1659252 ,  0.10133961,
        -0.17886318,  0.36936212,  0.00984475, -0.07069373, -0.14992891],
       [-0.21084788, -0.11169034, -0.17838386, -0.00068313, -0.16244861,
        -0.12498969,  0.08436415,  0.10639998,  0.00127891, -0.30879164,
        -0.11117086, -0.33672154,  0.3634963 ,  0.17624766,  0.19704497,
         0.21287435, -0.37998897, -0.25199062, -0.27589074, -0.17490673],
       [-0.30839762, -0.28260702, -0.01769292,  0.3119195 , -0.30392507,
        -0.09546056,  0.193739  , -0.3229484 , -0.07149467, -0.33627674,
         0.2778437 ,  0.28125572,  0.27764714,  0.16912973, -0.24110733,
        -0.30959487,  0.18371534, -0.02376282,  0.34793067, -0.26427105],
       [-0.36621308, -0.28679737, -0.27529812, -0.01110876, -0.18854263,
         0.02578986,  0.06842855, -0.23607235, -0.05325049,  0.2164644 ,
         0.19711733,  0.04214191,  0.2866007 , -0.29593727, -0.05848935,
         0.04491413,  0.14391595, -0.36737037,  0.3619147 ,  0.19894582],
```

```
       [-0.14433335,  0.1492241 , -0.16637681,  0.34656757, -0.35513553,
        -0.03279573, -0.32343727, -0.37269178,  0.37484354,  0.08988652,
         0.23501998,  0.21060681, -0.17095767,  0.37182933,  0.3228122 ,
         0.16822952,  0.13964725, -0.1962704 ,  0.02926382, -0.04845014],
       [-0.0723187 ,  0.2998283 , -0.2686603 , -0.2523256 ,  0.15023583,
         0.3206362 , -0.08584121,  0.21270603, -0.19684687, -0.3768993 ,
        -0.21380404, -0.10910255, -0.32748005,  0.2060051 ,  0.18456769,
        -0.06213805, -0.29152155,  0.34180903,  0.24649686, -0.06128982],
       [ 0.19512653,  0.15382522, -0.22229332, -0.13317075, -0.364324  ,
         0.12743437, -0.3278275 ,  0.06389359,  0.18111873,  0.2263555 ,
         0.149432  , -0.00474909,  0.2417537 , -0.3855004 , -0.13714263,
        -0.1558073 , -0.13479647, -0.16378781, -0.06301442,  0.22099108],
       [ 0.3659451 , -0.27746522, -0.21314381, -0.17721826,  0.28723562,
         0.043446  ,  0.14069188, -0.36108863, -0.3197291 ,  0.03790575,
         0.18246228,  0.341267  ,  0.2341758 , -0.08884686, -0.34836373,
        -0.13199767, -0.24667251, -0.02006668,  0.2611764 ,  0.3808717 ],
       [-0.17494367, -0.22739229,  0.01788497, -0.21983877,  0.11236554,
         0.18649954, -0.33687323, -0.18060507,  0.14969659,  0.0381619 ,
         0.00843436, -0.35075495,  0.12642306, -0.30686396, -0.14162041,
        -0.29686493, -0.3298985 , -0.10973313, -0.01031187, -0.04776999]],
      dtype=float32)
```

Out[16]:
```
(20, 20)
```

Out[16]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0.], dtype=float32)
```

Out[16]:
```
(20,)
```

Examine the weights and biases of Layer 3

In [17]:
```
weights, biases = model.layers[3].get_weights()

weights

weights.shape

biases

biases.shape
```

Out[17]:
```
array([[ 0.04408306,  0.36542046,  0.26690948, -0.24347757,  0.1313414 ,
        -0.11363956,  0.2218203 , -0.3421546 ,  0.22415304, -0.19604874],
       [-0.2608027 , -0.25291914,  0.14113396, -0.20771736, -0.43182316,
        -0.34950632, -0.08048141, -0.35588285, -0.13631114,  0.15477705],
       [ 0.15132159, -0.279961  ,  0.15711945, -0.05491522, -0.09607232,
         0.4219101 ,  0.13201994,  0.259125  ,  0.3111174 , -0.14609912],
       [-0.35370272, -0.02998248, -0.4066808 , -0.3775913 ,  0.26864576,
         0.07168204,  0.42556882,  0.22688591,  0.17318076, -0.1925419 ],
       [-0.43113425,  0.10641855, -0.32746786,  0.21406245, -0.13901949,
        -0.09051248,  0.11447072,  0.17184532, -0.3263547 , -0.00372022],
       [ 0.33084285,  0.02857012, -0.06847629, -0.05130854, -0.43838   ,
        -0.06801739,  0.24508888, -0.11380729, -0.38612655, -0.08503214],
       [-0.20522684, -0.25851178, -0.14190081, -0.23618096,  0.20544672,
         0.37163848, -0.04718006, -0.24027874,  0.30906266, -0.29113328],
       [ 0.37699997,  0.15887266, -0.02589408,  0.05094624,  0.34929574,
        -0.02723593, -0.36894387,  0.00566569, -0.02734521,  0.04455924],
       [ 0.26035947,  0.28463948,  0.32263875, -0.28184152,  0.4268434 ,
         0.3802809 ,  0.17371911, -0.19764033, -0.13101247,  0.09644711],
       [-0.17840526,  0.09079665,  0.00115475,  0.3041911 ,  0.07941818,
         0.00703219,  0.0076167 ,  0.08913714,  0.34177744,  0.18726027],
       [-0.40107214, -0.06476036, -0.01972991, -0.38601288, -0.4433038 ,
        -0.23705027,  0.08473635, -0.25968707,  0.38902557,  0.37679696],
       [ 0.10957122,  0.08155972, -0.01689959,  0.42389053,  0.34935558,
        -0.02481547, -0.11064598,  0.09124064, -0.166659  , -0.32461163],
       [-0.2652998 , -0.270276  ,  0.20211983,  0.01124948,  0.07969344,
         0.195229  , -0.02088776,  0.24470782, -0.17473033, -0.11801937],
       [ 0.05997312,  0.0523155 ,  0.4453079 ,  0.22290605, -0.22224414,
        -0.17254359,  0.03676131, -0.18473017, -0.18753174,  0.2566887 ],
       [ 0.12799788,  0.44124562, -0.40005848,  0.30466068,  0.26889515,
         0.167499  , -0.2768575 , -0.4257474 ,  0.06224191,  0.15646863],
       [ 0.08631289,  0.39535242, -0.32197213,  0.18419343,  0.27626145,
         0.3609218 , -0.1957162 , -0.4367896 ,  0.08756447,  0.33267665],
       [-0.1037811 ,  0.33112282, -0.04860264, -0.19427931,  0.04562399,
        -0.26178962,  0.38280886,  0.4124936 ,  0.17754477, -0.06265697],
       [ 0.27104193, -0.329284  , -0.2725336 , -0.41052288, -0.22730783,
         0.16526008, -0.26558608, -0.21583144,  0.30662322, -0.09565529],
       [-0.13057616,  0.22945243,  0.088875  , -0.24371055, -0.00927597,
        -0.38451207,  0.4456089 ,  0.11315358,  0.06784016,  0.25650328],
       [-0.14560279, -0.18970388,  0.29883015, -0.37701756, -0.34611142,
         0.31694365, -0.37443203, -0.11145517, -0.395399  ,  0.29043126]],
      dtype=float32)
```

Out[17]:
```
(20, 10)
```

Out[17]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

Out[17]:
```
(10,)
```

Compile the Sequential API model and specify the loss function and optimizer:

- Loss function: **sparse categorical cross-entropy**
- Optimization method: **RMS prop, learning rate set to 0.01**
- Evaluation metric: **accuracy**

In [18]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.RMSprop(learning_rate=0.01),
              metrics=["accuracy"])
```

Now the model is ready to be trained

In [19]:
```
history = model.fit(nn_x_train, nn_y_train, epochs=30,
                    validation_data=(nn_x_validation, nn_y_validation))
```

```
Epoch 1/30
985/985 [==============================] - 2s 2ms/step - loss: 0.3878 - accuracy: 0.8
847 - val_loss: 0.3442 - val_accuracy: 0.8929
Epoch 2/30
985/985 [==============================] - 2s 2ms/step - loss: 0.2638 - accuracy: 0.9
217 - val_loss: 0.2338 - val_accuracy: 0.9319
Epoch 3/30
985/985 [==============================] - 2s 2ms/step - loss: 0.2341 - accuracy: 0.9
319 - val_loss: 0.2545 - val_accuracy: 0.9271
Epoch 4/30
985/985 [==============================] - 2s 2ms/step - loss: 0.2150 - accuracy: 0.9
373 - val_loss: 0.2204 - val_accuracy: 0.9360
Epoch 5/30
985/985 [==============================] - 2s 2ms/step - loss: 0.2053 - accuracy: 0.9
386 - val_loss: 0.2381 - val_accuracy: 0.9285
Epoch 6/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1856 - accuracy: 0.9
448 - val_loss: 0.2402 - val_accuracy: 0.9332
Epoch 7/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1805 - accuracy: 0.9
477 - val_loss: 0.2231 - val_accuracy: 0.9371
Epoch 8/30
985/985 [==============================] - 3s 3ms/step - loss: 0.1767 - accuracy: 0.9
488 - val_loss: 0.2033 - val_accuracy: 0.9422
Epoch 9/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1696 - accuracy: 0.9
511 - val_loss: 0.2072 - val_accuracy: 0.9385
Epoch 10/30
985/985 [==============================] - 3s 3ms/step - loss: 0.1683 - accuracy: 0.9
513 - val_loss: 0.2219 - val_accuracy: 0.9377
Epoch 11/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1664 - accuracy: 0.9
510 - val_loss: 0.2123 - val_accuracy: 0.9413
Epoch 12/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1583 - accuracy: 0.9
551 - val_loss: 0.2106 - val_accuracy: 0.9407
Epoch 13/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1524 - accuracy: 0.9
564 - val_loss: 0.2115 - val_accuracy: 0.9390
Epoch 14/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1498 - accuracy: 0.9
579 - val_loss: 0.2056 - val_accuracy: 0.9429
Epoch 15/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1477 - accuracy: 0.9
568 - val_loss: 0.2111 - val_accuracy: 0.9443
Epoch 16/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1500 - accuracy: 0.9
569 - val_loss: 0.2164 - val_accuracy: 0.9412
Epoch 17/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1472 - accuracy: 0.9
573 - val_loss: 0.2091 - val_accuracy: 0.9421
Epoch 18/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1406 - accuracy: 0.9
591 - val_loss: 0.2085 - val_accuracy: 0.9439
Epoch 19/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1406 - accuracy: 0.9
591 - val_loss: 0.2131 - val_accuracy: 0.9420
Epoch 20/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1365 - accuracy: 0.9
612 - val_loss: 0.2201 - val_accuracy: 0.9374
```

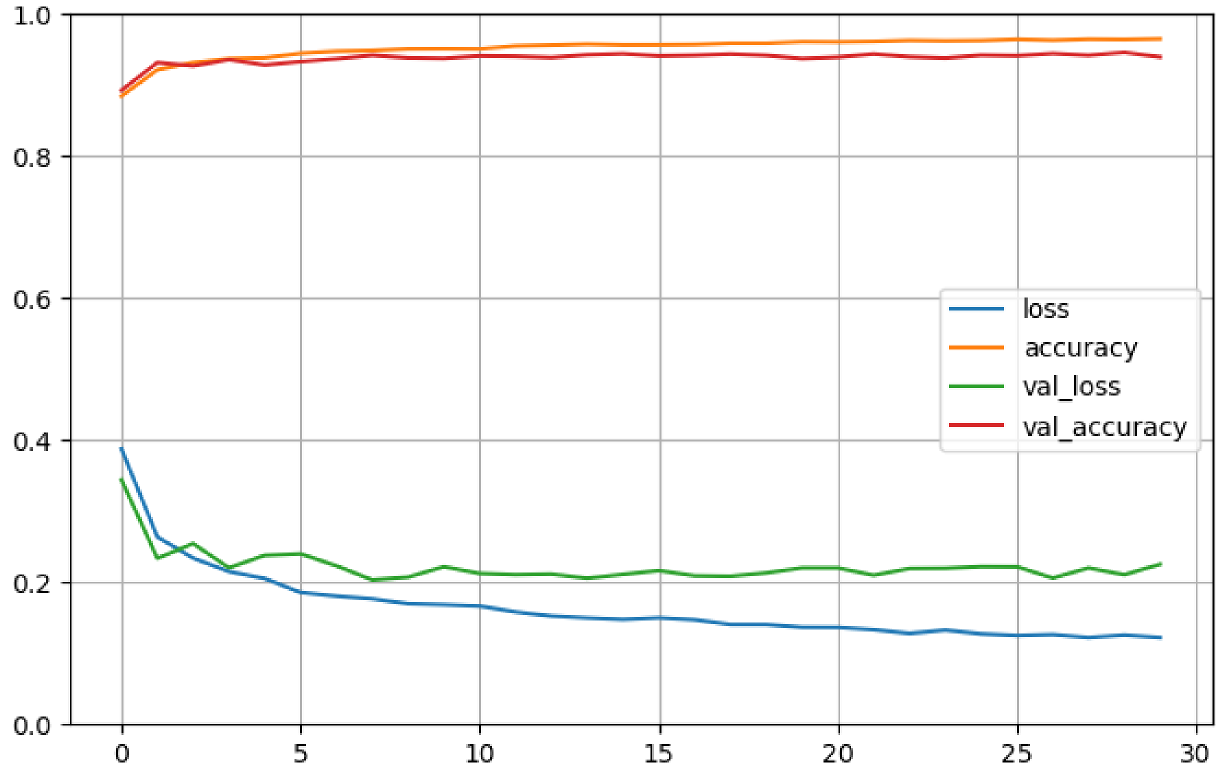
```
Epoch 21/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1363 - accuracy: 0.9
609 - val_loss: 0.2201 - val_accuracy: 0.9395
Epoch 22/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1332 - accuracy: 0.9
615 - val_loss: 0.2098 - val_accuracy: 0.9440
Epoch 23/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1280 - accuracy: 0.9
630 - val_loss: 0.2193 - val_accuracy: 0.9400
Epoch 24/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1326 - accuracy: 0.9
624 - val_loss: 0.2196 - val_accuracy: 0.9382
Epoch 25/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1274 - accuracy: 0.9
628 - val_loss: 0.2221 - val_accuracy: 0.9423
Epoch 26/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1251 - accuracy: 0.9
646 - val_loss: 0.2218 - val_accuracy: 0.9415
Epoch 27/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1263 - accuracy: 0.9
633 - val_loss: 0.2059 - val_accuracy: 0.9446
Epoch 28/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1224 - accuracy: 0.9
648 - val_loss: 0.2201 - val_accuracy: 0.9421
Epoch 29/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1257 - accuracy: 0.9
644 - val_loss: 0.2106 - val_accuracy: 0.9461
Epoch 30/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1224 - accuracy: 0.9
653 - val_loss: 0.2254 - val_accuracy: 0.9397
```

In [20]:

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```

Out[20]: <AxesSubplot:>

Out[20]: (0.0, 1.0)



We can then use the model to predict

The array below produces one probability per class (digit)

In [21]:
```
ypred = model.predict(nn_test_scaled_df)

y_proba = ypred.round(2)

y_proba
```

```
875/875 [==============================] - 1s 990us/step
```

Out[21]:
```
array([[0.  , 0.  , 1.  , ..., 0.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.01, 0.  , ..., 0.05, 0.21, 0.39],
       ...,
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 0.99],
       [0.  , 0.  , 0.99, ..., 0.  , 0.  , 0.  ]], dtype=float32)
```

Below are two ways to show the class with the highest estimated probability:

In [22]:
```
classes_x=np.argmax(ypred,axis=1)
classes_x
```

Out[22]:
```
array([2, 0, 9, ..., 3, 9, 2], dtype=int64)
```

In [23]:
```
class_names = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
np.array(class_names)[classes_x]
```

Out[23]:
```
array(['2', '0', '9', ..., '3', '9', '2'], dtype='<U1')
```

Accuracy and loss values for training data

In [24]:
```
loss, accuracy = model.evaluate(nn_x_train, nn_y_train)
print("Accuracy:", accuracy)
print("Loss:", loss)
```

```
985/985 [==============================] - 1s 1ms/step - loss: 0.1198 - accuracy: 0.9
654
Accuracy: 0.9653650522232056
Loss: 0.11980774998664856
```

Accuracy and loss values for validation data

In [25]:
```
loss, accuracy = model.evaluate(nn_x_validation, nn_y_validation)
print("Accuracy:", accuracy)
print("Loss:", loss)
```

```
329/329 [==============================] - 0s 1ms/step - loss: 0.2254 - accuracy: 0.9
397
Accuracy: 0.9397143125534058
Loss: 0.2254227101802826
```

Compile the Sequential API model and specify the loss function and optimizer:

- Loss function: **sparse categorical cross-entropy**
- Optimization method: **Adagrad, learning rate set to 0.1**
- Evaluation metric: **accuracy**

In [27]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.Adagrad(learning_rate=0.1),
              metrics=["accuracy"])
```

Now the model is ready to be trained

In [28]:
```
history = model.fit(nn_x_train, nn_y_train, epochs=30,
                    validation_data=(nn_x_validation, nn_y_validation))
```

```
Epoch 1/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0915 - accuracy: 0.9
742 - val_loss: 0.2013 - val_accuracy: 0.9449
Epoch 2/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0755 - accuracy: 0.9
784 - val_loss: 0.1951 - val_accuracy: 0.9470
Epoch 3/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0682 - accuracy: 0.9
807 - val_loss: 0.2012 - val_accuracy: 0.9474
Epoch 4/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0633 - accuracy: 0.9
819 - val_loss: 0.2025 - val_accuracy: 0.9474
Epoch 5/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0597 - accuracy: 0.9
837 - val_loss: 0.2056 - val_accuracy: 0.9458
Epoch 6/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0570 - accuracy: 0.9
845 - val_loss: 0.2073 - val_accuracy: 0.9461
Epoch 7/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0548 - accuracy: 0.9
849 - val_loss: 0.2102 - val_accuracy: 0.9448
Epoch 8/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0525 - accuracy: 0.9
863 - val_loss: 0.2120 - val_accuracy: 0.9459
Epoch 9/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0513 - accuracy: 0.9
864 - val_loss: 0.2114 - val_accuracy: 0.9472
Epoch 10/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0494 - accuracy: 0.9
871 - val_loss: 0.2152 - val_accuracy: 0.9448
Epoch 11/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0484 - accuracy: 0.9
871 - val_loss: 0.2157 - val_accuracy: 0.9450
Epoch 12/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0473 - accuracy: 0.9
878 - val_loss: 0.2170 - val_accuracy: 0.9462
Epoch 13/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0463 - accuracy: 0.9
876 - val_loss: 0.2176 - val_accuracy: 0.9459
Epoch 14/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0454 - accuracy: 0.9
883 - val_loss: 0.2201 - val_accuracy: 0.9450
Epoch 15/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0449 - accuracy: 0.9
886 - val_loss: 0.2197 - val_accuracy: 0.9453
Epoch 16/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0442 - accuracy: 0.9
885 - val_loss: 0.2214 - val_accuracy: 0.9465
Epoch 17/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0436 - accuracy: 0.9
887 - val_loss: 0.2234 - val_accuracy: 0.9462
Epoch 18/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0431 - accuracy: 0.9
888 - val_loss: 0.2230 - val_accuracy: 0.9460
Epoch 19/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0425 - accuracy: 0.9
890 - val_loss: 0.2255 - val_accuracy: 0.9458
Epoch 20/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0422 - accuracy: 0.9
892 - val_loss: 0.2263 - val_accuracy: 0.9456
```

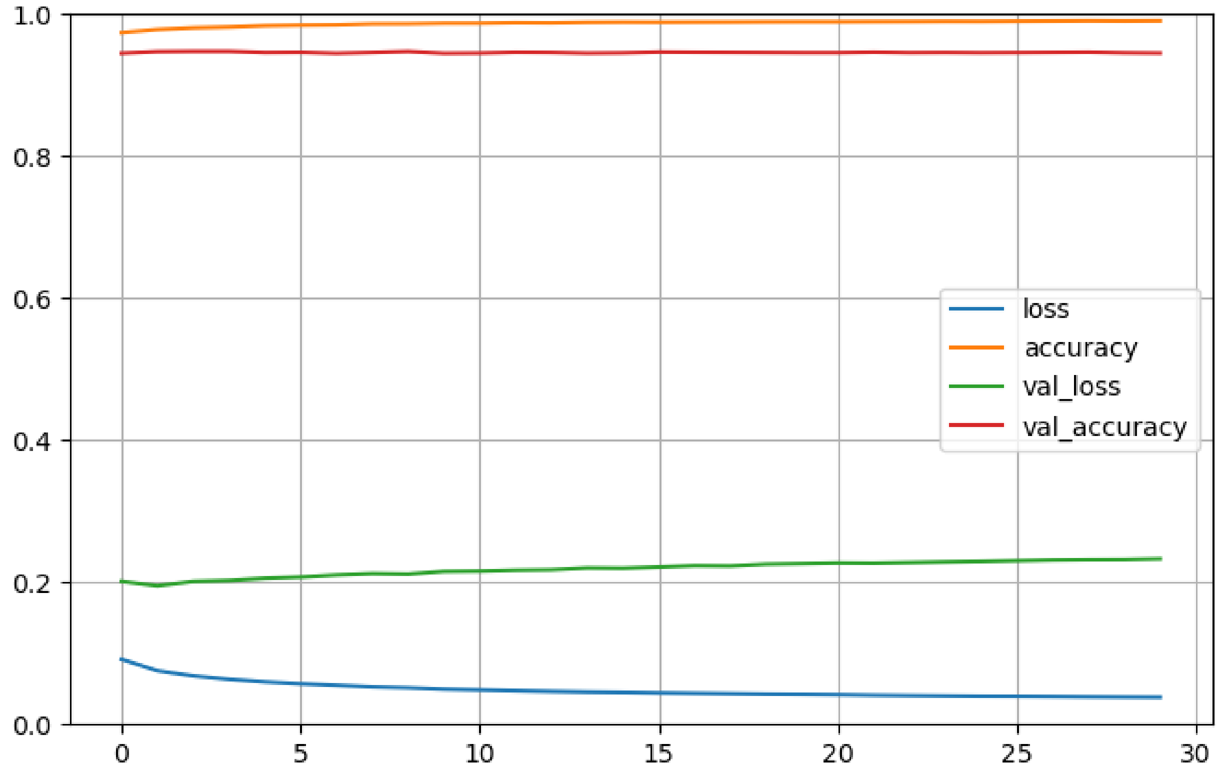
```
Epoch 21/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0417 - accuracy: 0.9
892 - val_loss: 0.2271 - val_accuracy: 0.9456
Epoch 22/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0410 - accuracy: 0.9
894 - val_loss: 0.2268 - val_accuracy: 0.9463
Epoch 23/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0405 - accuracy: 0.9
895 - val_loss: 0.2276 - val_accuracy: 0.9456
Epoch 24/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0401 - accuracy: 0.9
898 - val_loss: 0.2286 - val_accuracy: 0.9458
Epoch 25/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0397 - accuracy: 0.9
897 - val_loss: 0.2294 - val_accuracy: 0.9455
Epoch 26/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0396 - accuracy: 0.9
900 - val_loss: 0.2305 - val_accuracy: 0.9457
Epoch 27/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0392 - accuracy: 0.9
903 - val_loss: 0.2315 - val_accuracy: 0.9460
Epoch 28/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0388 - accuracy: 0.9
906 - val_loss: 0.2319 - val_accuracy: 0.9463
Epoch 29/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0385 - accuracy: 0.9
904 - val_loss: 0.2321 - val_accuracy: 0.9454
Epoch 30/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0383 - accuracy: 0.9
906 - val_loss: 0.2331 - val_accuracy: 0.9450
```

In [29]:

```python
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```

Out[29]: <AxesSubplot:>

Out[29]: (0.0, 1.0)



We can then use the model to predict

The array below produces one probability per class (digit)

In [30]:
```
ypred = model.predict(nn_test_scaled_df)

y_proba = ypred.round(2)

y_proba
```

```
875/875 [==============================] - 1s 1ms/step
```

Out[30]:
```
array([[0.  , 0.  , 1.  , ..., 0.  , 0.  , 0.  ],
       [0.99, 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.01, 0.  , ..., 0.  , 0.  , 0.89],
       ...,
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  , ..., 0.  , 0.  , 0.  ]], dtype=float32)
```

Below are two ways to show the class with the highest estimated probability:

In [31]:
```
classes_x=np.argmax(ypred,axis=1)
classes_x
```

Out[31]:
```
array([2, 0, 9, ..., 3, 9, 2], dtype=int64)
```

In [32]:
```
class_names = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
np.array(class_names)[classes_x]
```

Out[32]:
```
array(['2', '0', '9', ..., '3', '9', '2'], dtype='<U1')
```

Accuracy and loss values for training data

In [33]:
```
loss, accuracy = model.evaluate(nn_x_train, nn_y_train)
print("Accuracy:", accuracy)
print("Loss:", loss)
```

```
985/985 [==============================] - 1s 1ms/step - loss: 0.0370 - accuracy: 0.9
908
Accuracy: 0.9908254146575928
Loss: 0.03702748194336891
```

Accuracy and loss values for validation data

In [34]:
```
loss, accuracy = model.evaluate(nn_x_validation, nn_y_validation)
print("Accuracy:", accuracy)
print("Loss:", loss)
```

```
329/329 [==============================] - 0s 1ms/step - loss: 0.2331 - accuracy: 0.9
450
Accuracy: 0.9450476169586182
Loss: 0.23309732973575592
```

Create a Sequential API model utilizing RELU activation method

In [36]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[784,]),
    keras.layers.Dense(20, activation="relu"),
    keras.layers.Dense(20, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

In [37]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 dense_3 (Dense)             (None, 20)                15700

 dense_4 (Dense)             (None, 20)                420

 dense_5 (Dense)             (None, 10)                210

=================================================================
Total params: 16,330
Trainable params: 16,330
Non-trainable params: 0
_________________________________________________________________
```

In [38]:
```
model.layers
```

Out[38]:
```
[<keras.layers.reshaping.flatten.Flatten at 0x19cbd003220>,
 <keras.layers.core.dense.Dense at 0x19cbd003790>,
 <keras.layers.core.dense.Dense at 0x19cbd003040>,
 <keras.layers.core.dense.Dense at 0x19cbd003130>]
```

Examine the weights and biases of Hidden Layer 1

In [39]:
```
weights, biases = model.layers[1].get_weights()

weights

weights.shape

biases

biases.shape
```

Out[39]:
```
array([[-0.03979247,  0.04281363,  0.05254355, ..., -0.03678064,
         0.04354841, -0.07401411],
       [-0.03818677, -0.01641221, -0.0067735 , ...,  0.03408412,
        -0.06768759, -0.06182778],
       [ 0.01679847, -0.0574919 , -0.04084401, ..., -0.07583979,
        -0.00075091, -0.06602461],
       ...,
       [ 0.02339515, -0.01881402,  0.06827503, ..., -0.04605354,
         0.02058062,  0.08486018],
       [ 0.03087426,  0.02670233, -0.02150353, ...,  0.04234841,
         0.06985018,  0.03585381],
       [ 0.01518469,  0.07592402,  0.03056587, ..., -0.05865112,
         0.0355487 , -0.0219681 ]], dtype=float32)
```

Out[39]:
```
(784, 20)
```

Out[39]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0.], dtype=float32)
```

Out[39]:
```
(20,)
```

Examine the weights and biases of Hidden Layer 2

In [40]:
```
weights, biases = model.layers[2].get_weights()

weights

weights.shape

biases

biases.shape
```

Out[40]:
```
array([[ 0.00764143, -0.34899175, -0.3054614 , -0.14467518, -0.10968152,
         0.19741613,  0.36031878,  0.23396176,  0.2269885 ,  0.15340674,
         0.07932097,  0.15411556,  0.3755085 , -0.26812539, -0.09384489,
         0.03225303,  0.32333255,  0.08503556, -0.11971831, -0.08072177],
       [-0.15443736, -0.019137  ,  0.1889407 ,  0.24003631, -0.2219754 ,
        -0.07670453,  0.01307401, -0.11175692,  0.1704126 ,  0.01831067,
        -0.17758088, -0.12256902, -0.12644634,  0.10247388, -0.16840522,
        -0.17573187,  0.09597895,  0.17649102,  0.32489216, -0.19727486],
       [ 0.22062975, -0.19756787, -0.19619228,  0.29492617, -0.12241232,
        -0.3304566 ,  0.18560153, -0.38271657,  0.18384588,  0.21007574,
        -0.01481998,  0.14551252, -0.03626481,  0.30620694, -0.10323033,
         0.36442107, -0.3561025 ,  0.33644563,  0.18293315,  0.27569914],
       [ 0.2783491 ,  0.32727146,  0.19030058, -0.01746833, -0.37611747,
         0.14141828,  0.30225688,  0.05925244,  0.29479313, -0.01081014,
         0.22819114, -0.3780769 ,  0.1591897 , -0.38458866,  0.19366837,
        -0.26125064,  0.10099092,  0.1496346 ,  0.24375808, -0.08256051],
       [-0.1890891 , -0.28564414, -0.30769622,  0.20602536,  0.23465717,
        -0.32170767,  0.3626883 , -0.34407356, -0.07141969, -0.00073224,
        -0.19136   , -0.07150355,  0.1135909 ,  0.12470555, -0.1560196 ,
        -0.19816336,  0.02481511,  0.34028804,  0.32786655, -0.12143001],
       [-0.24197264,  0.13331813, -0.29655522, -0.00373337,  0.17505836,
         0.38693315, -0.24067518,  0.3709606 , -0.09004644,  0.0893296 ,
        -0.18385577,  0.15214008, -0.19331509, -0.09010074, -0.22337812,
        -0.15278533, -0.30511025,  0.32785058,  0.11130464, -0.23749982],
       [ 0.00856179,  0.1998601 , -0.26802355, -0.16199107, -0.33317554,
        -0.06663319,  0.17265946,  0.15535879, -0.01905242, -0.25464267,
         0.14260578, -0.33512795,  0.16446066, -0.18066159,  0.21852976,
         0.11806625,  0.3137591 , -0.08495829, -0.27221638,  0.23414302],
       [-0.10787407,  0.28761375,  0.11728251, -0.07585806, -0.20456217,
        -0.31148377, -0.37693226,  0.35867792,  0.32707942, -0.24693236,
         0.16620624,  0.20765537,  0.27830058,  0.03143898, -0.3265826 ,
         0.18699759, -0.11101878, -0.08446139, -0.14812331, -0.20310165],
       [ 0.35562307,  0.11299926,  0.11934525, -0.20102873,  0.38657337,
         0.31041044,  0.3179719 ,  0.35414964, -0.33214962,  0.15791345,
        -0.02004305, -0.09869418,  0.3332939 , -0.04325193, -0.03967738,
         0.24376827, -0.1658684 ,  0.37041527,  0.33358526,  0.21927404],
       [-0.2584777 ,  0.2638774 , -0.11692572, -0.22816074, -0.33313313,
        -0.06926596, -0.05568159, -0.36150876,  0.30971074,  0.18527299,
         0.08241719,  0.36464167,  0.22432005,  0.36092758, -0.3455181 ,
        -0.27346444, -0.11765176,  0.21416634,  0.3027388 , -0.21678105],
       [-0.3002386 , -0.01549193,  0.08711523,  0.27932   ,  0.11984086,
         0.19068414, -0.16582121,  0.29655755, -0.02407151, -0.27988496,
         0.0162167 ,  0.19098914,  0.02866113, -0.25819236,  0.10429114,
        -0.3733447 , -0.18785489, -0.3777297 , -0.10126933,  0.02370197],
       [-0.25945115,  0.30352515,  0.16930276,  0.01254687,  0.09269768,
         0.30373347,  0.18670923,  0.00048089,  0.18468875, -0.34093267,
         0.1042951 , -0.12074402, -0.2554791 , -0.16179273,  0.02960208,
        -0.1813099 ,  0.09989238, -0.35504678, -0.3804936 , -0.3124719 ],
       [-0.12619463,  0.21546274, -0.3663477 , -0.043751  ,  0.17313087,
        -0.09180698, -0.23982252, -0.34722114, -0.31810477,  0.29923046,
         0.16121554, -0.2172963 , -0.1476762 ,  0.29117376, -0.04951501,
        -0.2092373 , -0.3732977 , -0.12409511,  0.05738157,  0.1680333 ],
       [ 0.12570173,  0.08288202, -0.35427898, -0.33633545,  0.35164803,
         0.3198167 ,  0.3850072 , -0.02097863, -0.35521382,  0.29358244,
         0.10632083,  0.2238698 , -0.25506884, -0.21300557,  0.15855253,
        -0.3029819 , -0.02602485, -0.24205647, -0.32401714,  0.2174918 ],
       [-0.3834297 ,  0.32775015, -0.2956041 ,  0.23184884,  0.21468067,
         0.23259383, -0.14470363,  0.14087957,  0.26015013,  0.0124546 ,
         0.2883193 ,  0.15279317, -0.12321731, -0.20495555,  0.36603928,
         0.09090778, -0.07579213, -0.34663802, -0.02983367, -0.37571153],
```

```
        [ 0.2698729 ,  0.24932283,  0.2650481 , -0.04477099,  0.16907573,
         -0.07414064,  0.21925664,  0.34392536, -0.3610574 , -0.24059364,
          0.3226905 , -0.21211533, -0.19025081, -0.2954911 ,  0.11205417,
         -0.00239408,  0.12517512, -0.2929199 , -0.10160092,  0.3790518 ],
        [ 0.21871138, -0.2863329 , -0.30405822, -0.081976  , -0.00105479,
         -0.0524092 ,  0.32507038, -0.06196129, -0.24514449,  0.23013419,
          0.08737487, -0.14899038, -0.34000373,  0.28265458,  0.09568086,
          0.2514432 , -0.02778393,  0.18127924,  0.11546248,  0.10781249],
        [-0.30179787,  0.2405625 , -0.17674807,  0.1609624 , -0.31117165,
          0.24775195, -0.22001708, -0.00594619, -0.27679816, -0.17989646,
          0.26868182,  0.35418618, -0.34396192, -0.2536925 , -0.23078622,
          0.22637951, -0.07543755, -0.19965915,  0.13530588,  0.16948283],
        [ 0.268642  , -0.04199186, -0.09822944, -0.05944782, -0.3369667 ,
         -0.09784919, -0.2983728 , -0.00234044, -0.2942207 ,  0.11806554,
          0.27266896,  0.1290794 , -0.2538867 , -0.36522532,  0.16608113,
          0.04052809, -0.17704143,  0.12688035, -0.18797937,  0.03557974],
        [-0.00047037,  0.21756166,  0.26817507,  0.1062105 ,  0.10636607,
         -0.31875724, -0.01811427, -0.29735026,  0.33950067, -0.3283641 ,
          0.06639126, -0.18222035, -0.10648355, -0.15830444,  0.18900943,
          0.10567316, -0.2051635 ,  0.2960223 , -0.20399281,  0.08382037]],
      dtype=float32)
```

Out[40]:
```
(20, 20)
```

Out[40]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0.], dtype=float32)
```

Out[40]:
```
(20,)
```

Examine the weights and biases of Layer 3

In [41]:
```
weights, biases = model.layers[3].get_weights()

weights

weights.shape

biases

biases.shape
```

Out[41]:
```
array([[-1.10205948e-01,  3.38503242e-01,  1.52845144e-01,
         3.10721695e-01,  1.88794076e-01,  1.62791193e-01,
        -1.17220432e-01,  8.14648867e-02,  4.07818973e-01,
        -2.86190510e-01],
       [ 5.85073829e-02,  3.77504528e-01, -2.98117042e-01,
         2.88956463e-01,  4.07948554e-01, -8.56868029e-02,
         1.40638292e-01,  7.75764585e-02,  2.05348790e-01,
        -2.90280193e-01],
       [ 3.27653587e-01,  3.60073745e-01, -1.22879297e-01,
        -1.42733842e-01, -4.31077868e-01, -3.33747506e-01,
        -3.53607059e-01,  7.37733841e-02, -4.26989794e-02,
         1.49662197e-01],
       [ 2.60838211e-01,  1.13839626e-01,  1.68418646e-01,
        -4.11001295e-01, -1.51070148e-01, -1.37736917e-01,
        -3.06604624e-01, -1.61015093e-02,  3.71852040e-01,
         5.64956069e-02],
       [ 1.92616522e-01,  4.35427487e-01,  2.01382697e-01,
         7.24178553e-03, -2.70731747e-01,  1.02239311e-01,
         4.33010221e-01, -1.40573531e-01, -4.23397094e-01,
        -3.44616860e-01],
       [ 7.52208829e-02,  3.37415934e-02,  3.24965417e-01,
        -5.12090623e-02, -1.79583758e-01,  2.72836804e-01,
         2.48477280e-01,  4.67743576e-02, -2.35933378e-01,
         2.71788836e-01],
       [ 3.71943772e-01,  2.38551021e-01,  8.30721855e-02,
        -1.27231896e-01, -7.76597261e-02,  4.34239030e-01,
         2.76573122e-01, -3.45402360e-01, -3.58421475e-01,
         3.04992914e-01],
       [ 3.07343543e-01, -1.82100296e-01,  4.46379483e-02,
        -3.33831400e-01, -4.18546766e-01, -3.88078719e-01,
         2.11300135e-01,  6.65894747e-02,  1.70640111e-01,
         3.06712747e-01],
       [ 3.86252820e-01,  1.64274991e-01, -3.45127165e-01,
         2.55590916e-01,  3.06391060e-01, -1.39881879e-01,
        -1.13652378e-01, -6.61798418e-02,  2.96327055e-01,
        -3.96916062e-01],
       [ 2.59282887e-01, -2.80145466e-01, -3.86537045e-01,
         1.04100347e-01, -2.94995636e-01, -1.32264227e-01,
         5.89243770e-02, -3.85222286e-01, -4.43362117e-01,
        -3.03706497e-01],
       [ 2.20716000e-04, -1.32341743e-01,  2.39339471e-03,
         4.43200588e-01,  3.85064185e-02,  1.72161222e-01,
         2.84343243e-01, -2.60837793e-01,  4.34171140e-01,
         2.21128643e-01],
       [ 1.96405172e-01,  3.36833537e-01, -3.88042897e-01,
        -5.26454151e-02,  2.55570412e-02,  1.96398914e-01,
         8.25458765e-02,  4.16368306e-01, -1.64896727e-01,
         2.39781022e-01],
       [ 2.63751090e-01,  7.62267709e-02, -4.03370529e-01,
        -1.50919169e-01, -7.39949346e-02, -1.20963782e-01,
         5.01401722e-02, -4.85670269e-02, -2.63506472e-02,
         3.21253598e-01],
       [ 2.57392228e-01, -2.26779073e-01,  4.28586066e-01,
         1.04247272e-01, -6.78274930e-02,  1.42304718e-01,
        -1.88588053e-01,  2.62142420e-01, -3.18287313e-01,
         2.37607837e-01],
       [ 8.44530463e-02, -4.12862003e-02, -5.62485456e-02,
         3.98679733e-01,  2.32878089e-01, -1.24650955e-01,
        -3.69564205e-01, -6.32541776e-02,  4.16353405e-01,
         1.53099298e-02],
```

```
        [ 4.23958898e-01,  4.03609037e-01, -2.57029891e-01,
          4.52671349e-02, -3.48214805e-02, -9.93135571e-02,
         -3.18932831e-02,  2.83375859e-01,  3.41472745e-01,
         -1.27606362e-01],
        [-4.34163034e-01, -1.11080378e-01, -1.61027223e-01,
         -2.13499904e-01, -2.61558354e-01,  9.66523290e-02,
          4.08594608e-01, -3.54254603e-01, -1.74433500e-01,
         -1.02025449e-01],
        [-2.77305216e-01,  1.84592962e-01, -1.90234870e-01,
         -3.80549729e-02,  2.73194849e-01, -3.75434726e-01,
          2.39870071e-01,  2.92896330e-01,  2.82933891e-01,
          3.66562605e-03],
        [ 2.71426201e-01,  2.03330815e-02,  2.71110117e-01,
          3.41413260e-01,  2.86218226e-01, -4.84232903e-02,
          2.39132524e-01,  1.15943611e-01,  3.97716701e-01,
          1.09464526e-02],
        [ 7.80220032e-02, -1.19469881e-01,  2.36166120e-01,
          3.28730226e-01, -1.66809976e-01,  1.94371343e-01,
         -5.66865504e-02, -7.70777762e-02,  3.07431102e-01,
         -4.48100269e-02]], dtype=float32)
```

Out[41]:
```
(20, 10)
```

Out[41]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

Out[41]:
```
(10,)
```

Compile the Sequential API model and specify the loss function and optimizer:

- Loss function: **sparse categorical cross-entropy**
- Optimization method: **RMS prop, learning rate set to 0.001**
- Evaluation metric: **accuracy**

In [42]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.RMSprop(learning_rate=0.001),
              metrics=["accuracy"])
```

Now the model is ready to be trained

In [43]:
```
history = model.fit(nn_x_train, nn_y_train, epochs=30,
                    validation_data=(nn_x_validation, nn_y_validation))
```

```
Epoch 1/30
985/985 [==============================] - 2s 2ms/step - loss: 0.5625 - accuracy: 0.8
340 - val_loss: 0.3042 - val_accuracy: 0.9088
Epoch 2/30
985/985 [==============================] - 2s 2ms/step - loss: 0.2726 - accuracy: 0.9
227 - val_loss: 0.2401 - val_accuracy: 0.9319
Epoch 3/30
985/985 [==============================] - 2s 2ms/step - loss: 0.2285 - accuracy: 0.9
349 - val_loss: 0.2261 - val_accuracy: 0.9359
Epoch 4/30
985/985 [==============================] - 2s 2ms/step - loss: 0.2021 - accuracy: 0.9
417 - val_loss: 0.2092 - val_accuracy: 0.9376
Epoch 5/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1821 - accuracy: 0.9
480 - val_loss: 0.1969 - val_accuracy: 0.9436
Epoch 6/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1676 - accuracy: 0.9
525 - val_loss: 0.1794 - val_accuracy: 0.9465
Epoch 7/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1553 - accuracy: 0.9
553 - val_loss: 0.1747 - val_accuracy: 0.9497
Epoch 8/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1451 - accuracy: 0.9
584 - val_loss: 0.1665 - val_accuracy: 0.9524
Epoch 9/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1364 - accuracy: 0.9
606 - val_loss: 0.1594 - val_accuracy: 0.9544
Epoch 10/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1296 - accuracy: 0.9
632 - val_loss: 0.1663 - val_accuracy: 0.9519
Epoch 11/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1219 - accuracy: 0.9
651 - val_loss: 0.1751 - val_accuracy: 0.9502
Epoch 12/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1185 - accuracy: 0.9
664 - val_loss: 0.1614 - val_accuracy: 0.9536
Epoch 13/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1126 - accuracy: 0.9
671 - val_loss: 0.1652 - val_accuracy: 0.9533
Epoch 14/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1085 - accuracy: 0.9
693 - val_loss: 0.1874 - val_accuracy: 0.9475
Epoch 15/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1055 - accuracy: 0.9
696 - val_loss: 0.1751 - val_accuracy: 0.9533
Epoch 16/30
985/985 [==============================] - 2s 2ms/step - loss: 0.1012 - accuracy: 0.9
711 - val_loss: 0.1623 - val_accuracy: 0.9538
Epoch 17/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0997 - accuracy: 0.9
710 - val_loss: 0.1707 - val_accuracy: 0.9550
Epoch 18/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0956 - accuracy: 0.9
733 - val_loss: 0.1711 - val_accuracy: 0.9531
Epoch 19/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0933 - accuracy: 0.9
737 - val_loss: 0.1735 - val_accuracy: 0.9546
Epoch 20/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0897 - accuracy: 0.9
742 - val_loss: 0.1755 - val_accuracy: 0.9549
```

```
Epoch 21/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0885 - accuracy: 0.9
749 - val_loss: 0.1703 - val_accuracy: 0.9546
Epoch 22/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0841 - accuracy: 0.9
763 - val_loss: 0.1961 - val_accuracy: 0.9499
Epoch 23/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0830 - accuracy: 0.9
762 - val_loss: 0.1740 - val_accuracy: 0.9539
Epoch 24/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0798 - accuracy: 0.9
776 - val_loss: 0.1707 - val_accuracy: 0.9559
Epoch 25/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0783 - accuracy: 0.9
778 - val_loss: 0.1776 - val_accuracy: 0.9565
Epoch 26/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0783 - accuracy: 0.9
774 - val_loss: 0.1813 - val_accuracy: 0.9550
Epoch 27/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0748 - accuracy: 0.9
800 - val_loss: 0.1762 - val_accuracy: 0.9568
Epoch 28/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0733 - accuracy: 0.9
798 - val_loss: 0.1980 - val_accuracy: 0.9517
Epoch 29/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0707 - accuracy: 0.9
798 - val_loss: 0.1851 - val_accuracy: 0.9556
Epoch 30/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0682 - accuracy: 0.9
802 - val_loss: 0.2057 - val_accuracy: 0.9526
```

In [44]:

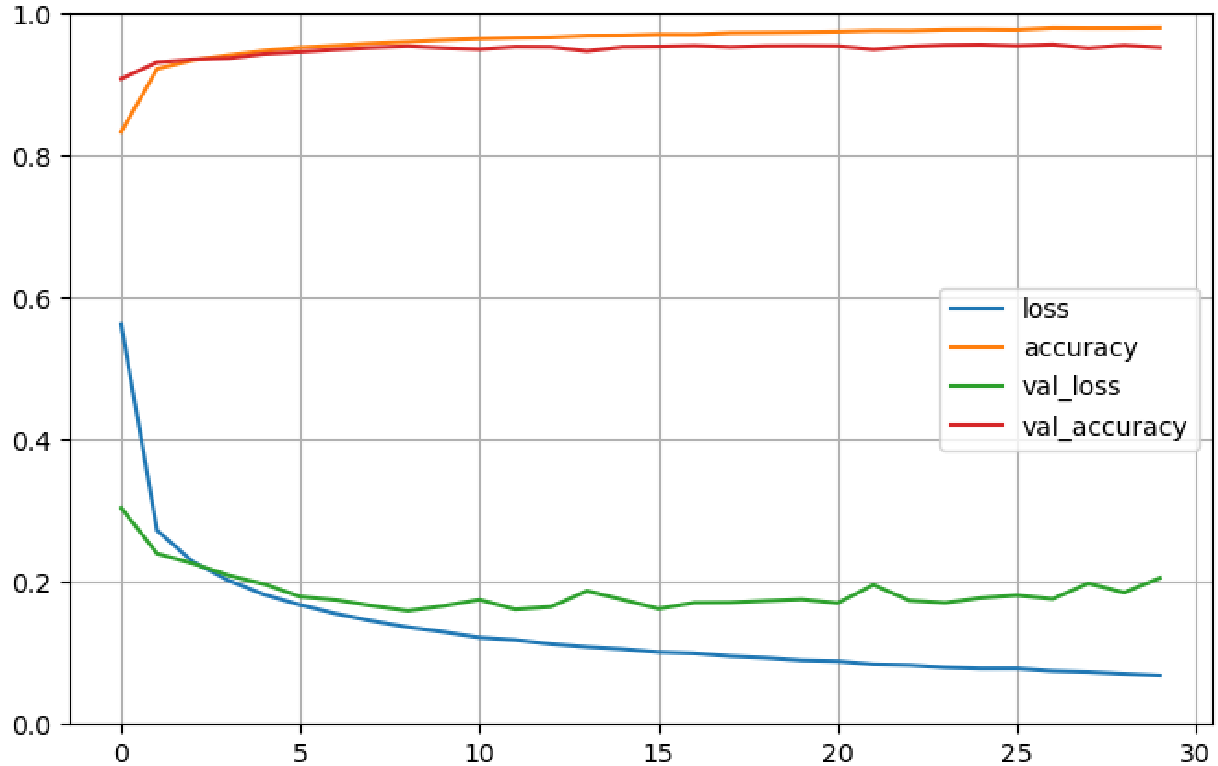
```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```

Out[44]:

```
<AxesSubplot:>
```

Out[44]:

```
(0.0, 1.0)
```



We can then use the model to predict

The array below produces one probability per class (digit)

In [45]:
```
ypred = model.predict(nn_test_scaled_df)

y_proba = ypred.round(2)

y_proba
```

```
875/875 [==============================] - 1s 938us/step
```

Out[45]:
```
array([[0.  , 0.  , 1.  , ..., 0.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , ..., 0.02, 0.  , 0.91],
       ...,
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  , ..., 0.  , 0.  , 0.  ]], dtype=float32)
```

Below are two ways to show the class with the highest estimated probability:

In [46]:
```
classes_x=np.argmax(ypred,axis=1)
classes_x
```

Out[46]:
```
array([2, 0, 9, ..., 3, 9, 2], dtype=int64)
```

In [47]:
```
class_names = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
np.array(class_names)[classes_x]
```

Out[47]:
```
array(['2', '0', '9', ..., '3', '9', '2'], dtype='<U1')
```

Accuracy and loss values for training data

In [48]:
```
loss, accuracy = model.evaluate(nn_x_train, nn_y_train)
print("Accuracy:", accuracy)
print("Loss:", loss)
```

```
985/985 [==============================] - 1s 1ms/step - loss: 0.0689 - accuracy: 0.9
801
Accuracy: 0.9800634980201721
Loss: 0.06894040107727051
```

Accuracy and loss values for validation data

In [49]:
```
loss, accuracy = model.evaluate(nn_x_validation, nn_y_validation)
print("Accuracy:", accuracy)
print("Loss:", loss)
```

```
329/329 [==============================] - 0s 1ms/step - loss: 0.2057 - accuracy: 0.9
526
Accuracy: 0.952571451663971
Loss: 0.2056991308927536
```

Compile the Sequential API model and specify the loss function and optimizer:

- Loss function: **sparse categorical cross-entropy**
- Optimization method: **AdaGrad, learning rate set to 0.005**
- Evaluation metric: **accuracy**

In [51]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.Adagrad(learning_rate=0.005),
              metrics=["accuracy"])
```

Now the model is ready to be trained

In [52]:
```
history = model.fit(nn_x_train, nn_y_train, epochs=30,
                    validation_data=(nn_x_validation, nn_y_validation))
```

```
Epoch 1/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0522 - accuracy: 0.9
862 - val_loss: 0.1754 - val_accuracy: 0.9572
Epoch 2/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0472 - accuracy: 0.9
879 - val_loss: 0.1786 - val_accuracy: 0.9570
Epoch 3/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0456 - accuracy: 0.9
886 - val_loss: 0.1757 - val_accuracy: 0.9566
Epoch 4/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0446 - accuracy: 0.9
890 - val_loss: 0.1770 - val_accuracy: 0.9558
Epoch 5/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0436 - accuracy: 0.9
893 - val_loss: 0.1758 - val_accuracy: 0.9568
Epoch 6/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0431 - accuracy: 0.9
897 - val_loss: 0.1766 - val_accuracy: 0.9567
Epoch 7/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0424 - accuracy: 0.9
896 - val_loss: 0.1780 - val_accuracy: 0.9565
Epoch 8/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0420 - accuracy: 0.9
897 - val_loss: 0.1770 - val_accuracy: 0.9561
Epoch 9/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0416 - accuracy: 0.9
894 - val_loss: 0.1775 - val_accuracy: 0.9564
Epoch 10/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0411 - accuracy: 0.9
901 - val_loss: 0.1771 - val_accuracy: 0.9561
Epoch 11/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0408 - accuracy: 0.9
900 - val_loss: 0.1782 - val_accuracy: 0.9563
Epoch 12/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0404 - accuracy: 0.9
897 - val_loss: 0.1771 - val_accuracy: 0.9562
Epoch 13/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0401 - accuracy: 0.9
903 - val_loss: 0.1778 - val_accuracy: 0.9557
Epoch 14/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0398 - accuracy: 0.9
902 - val_loss: 0.1772 - val_accuracy: 0.9561
Epoch 15/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0396 - accuracy: 0.9
903 - val_loss: 0.1775 - val_accuracy: 0.9554
Epoch 16/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0393 - accuracy: 0.9
904 - val_loss: 0.1780 - val_accuracy: 0.9558
Epoch 17/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0391 - accuracy: 0.9
903 - val_loss: 0.1781 - val_accuracy: 0.9563
Epoch 18/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0388 - accuracy: 0.9
905 - val_loss: 0.1782 - val_accuracy: 0.9562
Epoch 19/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0386 - accuracy: 0.9
907 - val_loss: 0.1775 - val_accuracy: 0.9554
Epoch 20/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0383 - accuracy: 0.9
905 - val_loss: 0.1783 - val_accuracy: 0.9553
```

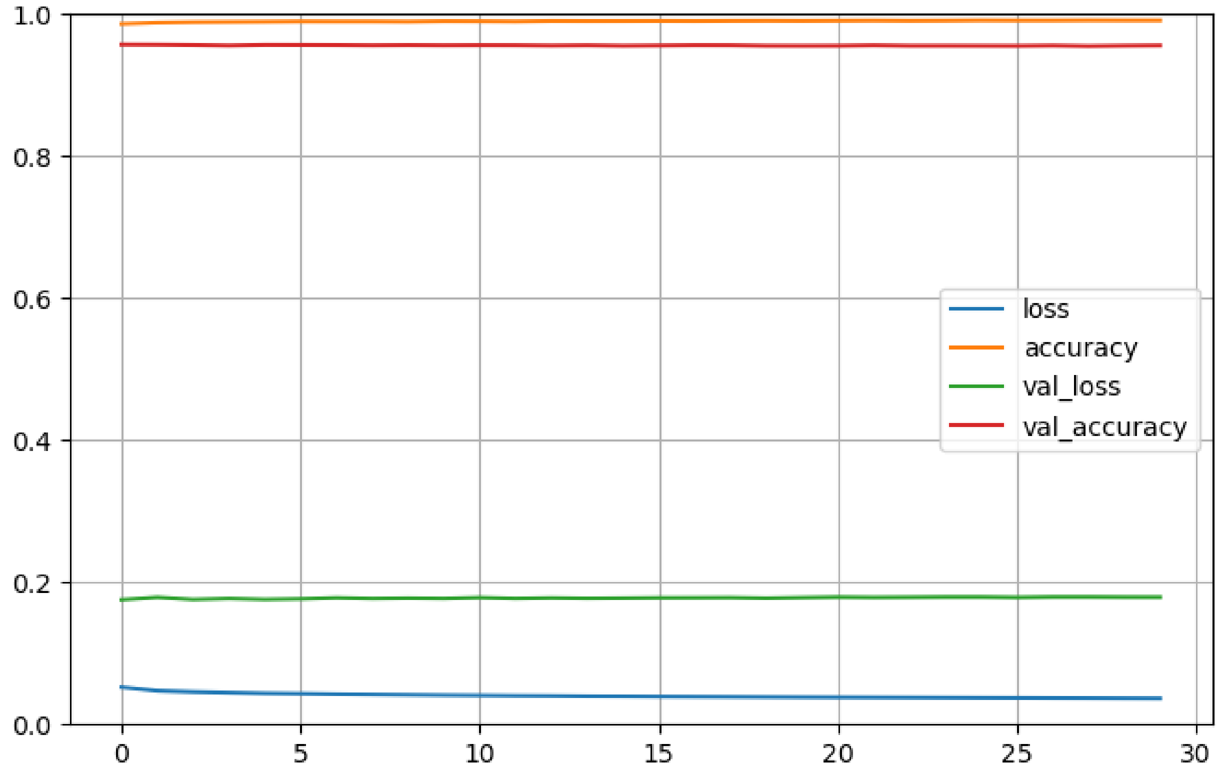
```
Epoch 21/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0381 - accuracy: 0.9
908 - val_loss: 0.1790 - val_accuracy: 0.9554
Epoch 22/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0379 - accuracy: 0.9
910 - val_loss: 0.1786 - val_accuracy: 0.9561
Epoch 23/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0378 - accuracy: 0.9
907 - val_loss: 0.1788 - val_accuracy: 0.9554
Epoch 24/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0376 - accuracy: 0.9
909 - val_loss: 0.1791 - val_accuracy: 0.9554
Epoch 25/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0374 - accuracy: 0.9
912 - val_loss: 0.1791 - val_accuracy: 0.9553
Epoch 26/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0372 - accuracy: 0.9
911 - val_loss: 0.1786 - val_accuracy: 0.9552
Epoch 27/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0370 - accuracy: 0.9
911 - val_loss: 0.1792 - val_accuracy: 0.9557
Epoch 28/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0369 - accuracy: 0.9
913 - val_loss: 0.1792 - val_accuracy: 0.9550
Epoch 29/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0367 - accuracy: 0.9
912 - val_loss: 0.1790 - val_accuracy: 0.9555
Epoch 30/30
985/985 [==============================] - 2s 2ms/step - loss: 0.0365 - accuracy: 0.9
912 - val_loss: 0.1789 - val_accuracy: 0.9559
```

In [53]:

```python
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```

Out[53]: <AxesSubplot:>

Out[53]: (0.0, 1.0)



We can then use the model to predict

The array below produces one probability per class (digit)

In [54]:
```
ypred = model.predict(nn_test_scaled_df)

y_proba = ypred.round(2)

y_proba
```

```
875/875 [==============================] - 1s 984us/step
```
Out[54]:
```
array([[0.  , 0.  , 1.  , ..., 0.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.01, 0.  , ..., 0.04, 0.  , 0.93],
       ...,
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , ..., 0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  , ..., 0.  , 0.  , 0.  ]], dtype=float32)
```

Below are two ways to show the class with the highest estimated probability:

In [55]:
```
classes_x=np.argmax(ypred,axis=1)
classes_x
```

Out[55]:
```
array([2, 0, 9, ..., 3, 9, 2], dtype=int64)
```

In [56]:
```
class_names = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
np.array(class_names)[classes_x]
```

Out[56]:
```
array(['2', '0', '9', ..., '3', '9', '2'], dtype='<U1')
```

Accuracy and loss values for training data

In [57]:
```
loss, accuracy = model.evaluate(nn_x_train, nn_y_train)
print("Accuracy:", accuracy)
print("Loss:", loss)
```

```
985/985 [==============================] - 1s 1ms/step - loss: 0.0358 - accuracy: 0.9
916
Accuracy: 0.9915555715560913
Loss: 0.03575896471738815
```

Accuracy and loss values for validation data

In [58]:
```
loss, accuracy = model.evaluate(nn_x_validation, nn_y_validation)
print("Accuracy:", accuracy)
print("Loss:", loss)
```

```
329/329 [==============================] - 0s 1ms/step - loss: 0.1789 - accuracy: 0.9
559
Accuracy: 0.9559047818183899
Loss: 0.17890749871730804
```

2 x 2 Experiment to Tune Learning Rates with MLPClassifier

In this section, we will conduct a 2x2 experiment that compares the accuracy of neural nets with constant and adaptive learning rates and with initial learning rates of 0.001 and 0.01.

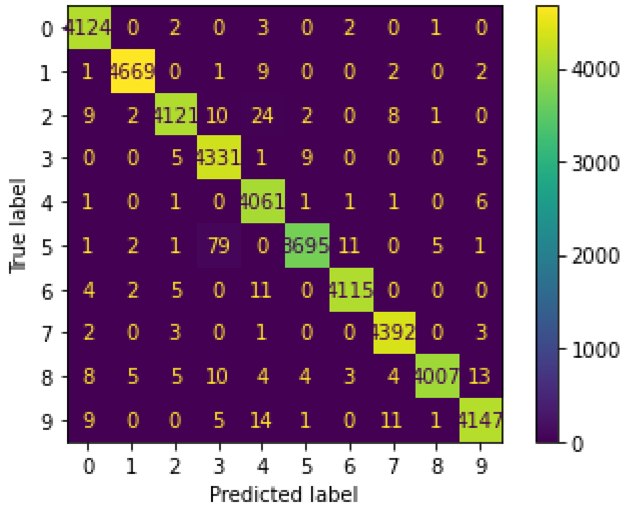
Build a Neural Net with Constant Learning Rate and Initial Learning Rate of 0.01

Build the initial model using the testing data

In [6]:
```
# Import libraries
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, C
import datetime


# Extract predictors and outcome (label variable)
X_train_SD = digit_training_data.copy(deep=True)
X_train_SD.drop(['label'], axis=1, inplace=True)
y_train_SD = digit_training_data['label']


# Standardize the features
xscaler = StandardScaler()
X_train_SD = xscaler.fit_transform(X_train_SD)

# Initialize MLP Classifier
mlp_class = MLPClassifier(random_state=1, hidden_layer_sizes =(20, 20, 20, 20, 20), sc

# Create paramater grid with hyperparameters to tune, use default adam solver so doen
param_grid = {
    'learning_rate': ['constant'],
```

```
    'learning_rate_init': [0.01]
}

# Kfold cv with 5 splits for GridSearch
cv = KFold(n_splits=5, shuffle=True, random_state=1)

# Create the GridSearchCV with kfold=5 object and fit it to the training data
nn_start = datetime.datetime.now()
grid_search = GridSearchCV(mlp_class, param_grid, cv=cv, scoring='accuracy')
grid_search.fit(X_train_SD, y_train_SD)

# Print the time to fit the neural net model
nn_end = datetime.datetime.now()
nn_runtime = nn_end - nn_start
print(f"The total run time for the Principal Components Analysis was {nn_runtime}.")

# Print the best hyperparameters found
print("Best Hyperparameters:", grid_search.best_params_)

# Save the best estimator
best_model = grid_search.best_estimator_

# Save dictionary of mean accuracy scores from models into 'scores' variable
#dict_results = grid_search.cv_results_
#scores = dict_results['mean_test_score']

# Use the best model to predict using training data
y_pred_train_SD = best_model.predict(X_train_SD)

# evaluate the model on the training data
accuracy_train_SD = accuracy_score(y_train_SD, y_pred_train_SD)
print("Training Accuracy:", accuracy_train_SD)
print("Training Classification Report:", classification_report(y_train_SD, y_pred_trai

# Create the confusion matrix of the predictions
cm = confusion_matrix(y_train_SD, y_pred_train_SD)
ConfusionMatrixDisplay(confusion_matrix=cm).plot();
```

```
The total run time for the Principal Components Analysis was 0:03:16.223515.
Best Hyperparameters: {'learning_rate': 'constant', 'learning_rate_init': 0.01}
Training Accuracy: 0.9919523809523809
Training Classification Report:               precision    recall  f1-score   support

           0       0.99      1.00      0.99      4132
           1       1.00      1.00      1.00      4684
           2       0.99      0.99      0.99      4177
           3       0.98      1.00      0.99      4351
           4       0.98      1.00      0.99      4072
           5       1.00      0.97      0.98      3795
           6       1.00      0.99      1.00      4137
           7       0.99      1.00      1.00      4401
           8       1.00      0.99      0.99      4063
           9       0.99      0.99      0.99      4188

    accuracy                           0.99     42000
   macro avg       0.99      0.99      0.99     42000
weighted avg       0.99      0.99      0.99     42000
```



Apply MLP Classifier to test data

In [7]:
```
# Create a dataframe for predictor variables in the test dataframe for mlpclass model
mlpclass_testing_x_SD = digit_testing_data.copy(deep=True)

# Standardize the features using same scaler as training data
mlpclass_testing_xscale_SD = xscaler.transform(mlpclass_testing_x_SD)

# Apply the mlpclass model to the test dataset
mlpclass_test_ypred_SD = best_model.predict(mlpclass_testing_xscale_SD)

# Put the kmeans predictions into a Pandas dataframe
prediction_df_mlpclass_SD = pd.DataFrame(mlpclass_test_ypred_SD, columns=['Label'])

# Add the ID column to the front of the mlpclass predictions dataframe
ImageId_series = pd.Series(range(1,28001))
prediction_df_mlpclass_SD.insert(0, 'ImageId', ImageId_series)

# Output predictions to csv
prediction_df_mlpclass_SD.to_csv('test_predictions_mlpclass_constant_01.csv', index=Fa
```

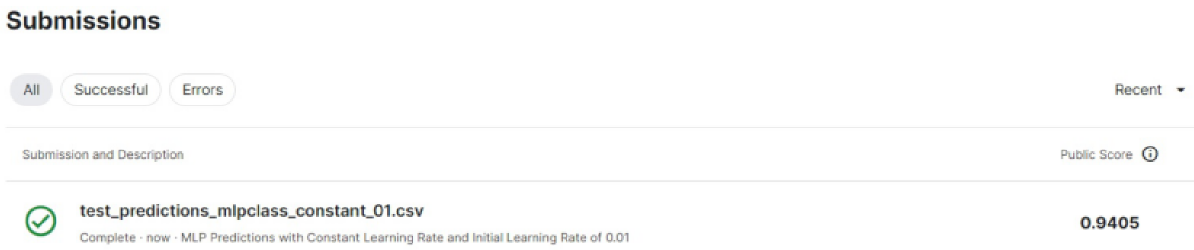
Let's display the Kaggle results from the application of the MLP Classifier model on the test dataset

In [8]:
```
# Display the kaggle results associated with the MLP Classifier Model
import matplotlib.pyplot as plt
plt.figure(figsize = (15, 15))
kaggle_results = plt.imread('Kaggle_results_mlpclass_constant_01.jpg')
plt.imshow(kaggle_results)
plt.axis("off")
plt.show()
```

Out[8]: <Figure size 1080x1080 with 0 Axes>

Out[8]: <matplotlib.image.AxesImage at 0x227b91e6fd0>

Out[8]: (-0.5, 1481.5, 314.5, -0.5)



Build a Neural Net with Constant Learning Rate and Initial Learning Rate of 0.001

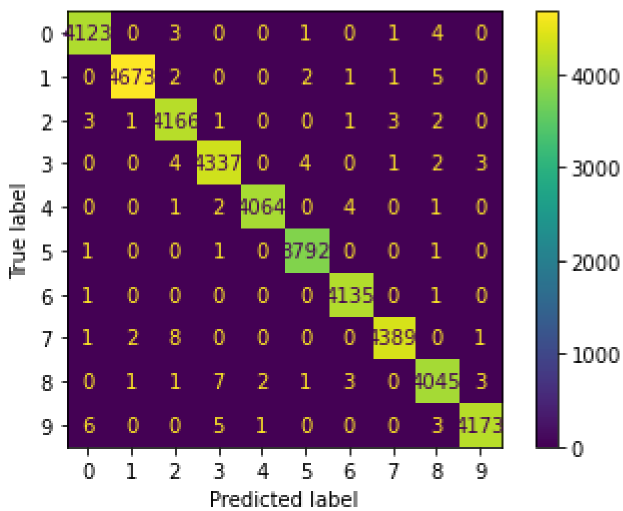
Build the initial model using the testing data

In [9]:
```
# Import libraries
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, (
import datetime


# Extract predictors and outcome (label variable)
X_train_SD = digit_training_data.copy(deep=True)
X_train_SD.drop(['label'], axis=1, inplace=True)
y_train_SD = digit_training_data['label']


# Standardize the features
xscaler = StandardScaler()
X_train_SD = xscaler.fit_transform(X_train_SD)

# Initialize MLP Classifier
mlp_class = MLPClassifier(random_state=1, hidden_layer_sizes =(20, 20, 20, 20, 20), sc

# Create paramater grid with hyperparameters to tune, use default adam solver so doen
param_grid = {
    'learning_rate': ['constant'],
    'learning_rate_init': [0.001]
}

# Kfold cv with 5 splits for GridSearch
cv = KFold(n_splits=5, shuffle=True, random_state=1)

# Create the GridSearchCV with kfold=5 object and fit it to the training data
nn_start = datetime.datetime.now()
grid_search = GridSearchCV(mlp_class, param_grid, cv=cv, scoring='accuracy')
grid_search.fit(X_train_SD, y_train_SD)

# Print the time to fit the neural net model
nn_end = datetime.datetime.now()
nn_runtime = nn_end - nn_start
print(f"The total run time for the Principal Components Analysis was {nn_runtime}.")

# Print the best hyperparameters found
print("Best Hyperparameters:", grid_search.best_params_)
```

```
# Save the best estimator
best_model = grid_search.best_estimator_

# Save dictionary of mean accuracy scores from models into 'scores' variable
#dict_results = grid_search.cv_results_
#scores = dict_results['mean_test_score']

# Use the best model to predict using training data
y_pred_train_SD = best_model.predict(X_train_SD)

# evaluate the model on the training data
accuracy_train_SD = accuracy_score(y_train_SD, y_pred_train_SD)
print("Training Accuracy:", accuracy_train_SD)
print("Training Classification Report:", classification_report(y_train_SD, y_pred_trai

# Create the confusion matrix of the predictions
cm = confusion_matrix(y_train_SD, y_pred_train_SD)
ConfusionMatrixDisplay(confusion_matrix=cm).plot();
```

```
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
```

```
The total run time for the Principal Components Analysis was 0:08:33.066414.
Best Hyperparameters: {'learning_rate': 'constant', 'learning_rate_init': 0.001}
Training Accuracy: 0.997547619047619
Training Classification Report:               precision    recall  f1-score   support

           0       1.00      1.00      1.00      4132
           1       1.00      1.00      1.00      4684
           2       1.00      1.00      1.00      4177
           3       1.00      1.00      1.00      4351
           4       1.00      1.00      1.00      4072
           5       1.00      1.00      1.00      3795
           6       1.00      1.00      1.00      4137
           7       1.00      1.00      1.00      4401
           8       1.00      1.00      1.00      4063
           9       1.00      1.00      1.00      4188

    accuracy                           1.00     42000
   macro avg       1.00      1.00      1.00     42000
weighted avg       1.00      1.00      1.00     42000
```



Apply MLP Classifier to test data

In [10]:
```python
# Create a dataframe for predictor variables in the test dataframe for mlpclass model
mlpclass_testing_x_SD = digit_testing_data.copy(deep=True)

# Standardize the features using same scaler as training data
mlpclass_testing_xscale_SD = xscaler.transform(mlpclass_testing_x_SD)

# Apply the mlpclass model to the test dataset
mlpclass_test_ypred_SD = best_model.predict(mlpclass_testing_xscale_SD)

# Put the kmeans predictions into a Pandas dataframe
prediction_df_mlpclass_SD = pd.DataFrame(mlpclass_test_ypred_SD, columns=['Label'])

# Add the ID column to the front of the mlpclass predictions dataframe
ImageId_series = pd.Series(range(1,28001))
prediction_df_mlpclass_SD.insert(0, 'ImageId', ImageId_series)

# Output predictions to csv
prediction_df_mlpclass_SD.to_csv('test_predictions_mlpclass_constant_001.csv', index=F
```

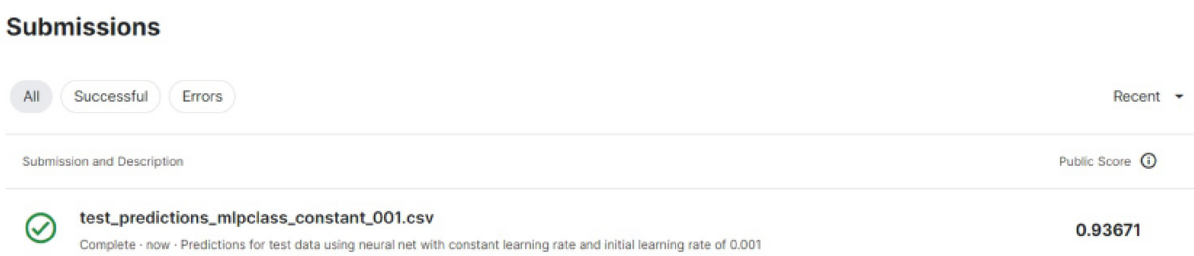
Let's display the Kaggle results from the application of the MLP Classifier model on the test dataset

In [11]:
```
# Display the kaggle results associated with the MLP Classifier Model
import matplotlib.pyplot as plt
plt.figure(figsize = (15, 15))
kaggle_results = plt.imread('Kaggle_results_mlpclass_constant_001.jpg')
plt.imshow(kaggle_results)
plt.axis("off")
plt.show()
```

Out[11]:
```
<Figure size 1080x1080 with 0 Axes>
```

Out[11]:
```
<matplotlib.image.AxesImage at 0x2279a4e8bb0>
```

Out[11]:
```
(-0.5, 1483.5, 317.5, -0.5)
```



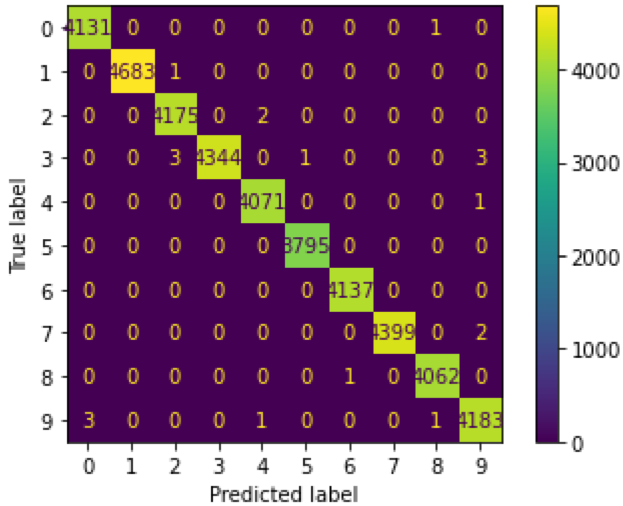
Build a Neural Net with Adaptive Learning Rate and Initial Learning Rate of 0.01

Build the initial model using the testing data

In [12]:
```
# Import libraries
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, C
import datetime


# Extract predictors and outcome (label variable)
X_train_SD = digit_training_data.copy(deep=True)
X_train_SD.drop(['label'], axis=1, inplace=True)
y_train_SD = digit_training_data['label']


# Standardize the features
xscaler = StandardScaler()
X_train_SD = xscaler.fit_transform(X_train_SD)

# Initialize MLP Classifier
mlp_class = MLPClassifier(random_state=1, hidden_layer_sizes =(20, 20, 20, 20, 20), sc

# Create paramater grid with hyperparameters to tune, use default adam solver so doen
param_grid = {
    'learning_rate': ['adaptive'],
    'learning_rate_init': [0.01]
```

```
}

# Kfold cv with 5 splits for GridSearch
cv = KFold(n_splits=5, shuffle=True, random_state=1)

# Create the GridSearchCV with kfold=5 object and fit it to the training data
nn_start = datetime.datetime.now()
grid_search = GridSearchCV(mlp_class, param_grid, cv=cv, scoring='accuracy')
grid_search.fit(X_train_SD, y_train_SD)

# Print the time to fit the neural net model
nn_end = datetime.datetime.now()
nn_runtime = nn_end - nn_start
print(f"The total run time for the model creation was {nn_runtime}.")

# Print the best hyperparameters found
print("Best Hyperparameters:", grid_search.best_params_)

# Save the best estimator
best_model = grid_search.best_estimator_

# Save dictionary of mean accuracy scores from models into 'scores' variable
#dict_results = grid_search.cv_results_
#scores = dict_results['mean_test_score']

# Use the best model to predict using training data
y_pred_train_SD = best_model.predict(X_train_SD)

# evaluate the model on the training data
accuracy_train_SD = accuracy_score(y_train_SD, y_pred_train_SD)
print("Training Accuracy:", accuracy_train_SD)
print("Training Classification Report:", classification_report(y_train_SD, y_pred_trai

# Create the confusion matrix of the predictions
cm = confusion_matrix(y_train_SD, y_pred_train_SD)
ConfusionMatrixDisplay(confusion_matrix=cm).plot();
```

```
The total run time for the model creation was 0:06:39.432408.
Best Hyperparameters: {'learning_rate': 'adaptive', 'learning_rate_init': 0.01}
Training Accuracy: 0.9995238095238095
Training Classification Report:               precision    recall  f1-score   support

           0       1.00      1.00      1.00      4132
           1       1.00      1.00      1.00      4684
           2       1.00      1.00      1.00      4177
           3       1.00      1.00      1.00      4351
           4       1.00      1.00      1.00      4072
           5       1.00      1.00      1.00      3795
           6       1.00      1.00      1.00      4137
           7       1.00      1.00      1.00      4401
           8       1.00      1.00      1.00      4063
           9       1.00      1.00      1.00      4188

    accuracy                           1.00     42000
   macro avg       1.00      1.00      1.00     42000
weighted avg       1.00      1.00      1.00     42000
```



Apply MLP Classifier to test data

In [13]:

```
# Create a dataframe for predictor variables in the test dataframe for mlpclass model
mlpclass_testing_x_SD = digit_testing_data.copy(deep=True)

# Standardize the features using same scaler as training data
mlpclass_testing_xscale_SD = xscaler.transform(mlpclass_testing_x_SD)

# Apply the mlpclass model to the test dataset
mlpclass_test_ypred_SD = best_model.predict(mlpclass_testing_xscale_SD)

# Put the kmeans predictions into a Pandas dataframe
prediction_df_mlpclass_SD = pd.DataFrame(mlpclass_test_ypred_SD, columns=['Label'])

# Add the ID column to the front of the mlpclass predictions dataframe
ImageId_series = pd.Series(range(1,28001))
prediction_df_mlpclass_SD.insert(0, 'ImageId', ImageId_series)

# Output predictions to csv
prediction_df_mlpclass_SD.to_csv('test_predictions_mlpclass_adaptive_01.csv', index=Fa
```

Let's display the Kaggle results from the application of the MLP Classifier model on the test dataset

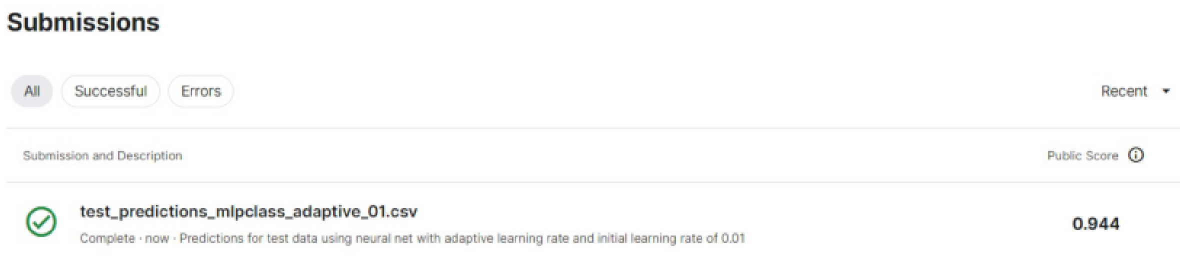
In [14]:

```
# Display the kaggle results associated with the MLP Classifier Model
import matplotlib.pyplot as plt
plt.figure(figsize = (15, 15))
kaggle_results = plt.imread('Kaggle_results_mlpclass_adaptive_01.jpg')
plt.imshow(kaggle_results)
plt.axis("off")
plt.show()
```

Out[14]: <Figure size 1080x1080 with 0 Axes>

Out[14]: <matplotlib.image.AxesImage at 0x227ae947730>

Out[14]: (-0.5, 1501.5, 331.5, -0.5)



Build a Neural Net with Adaptive Learning Rate and Initial Learning Rate of 0.001

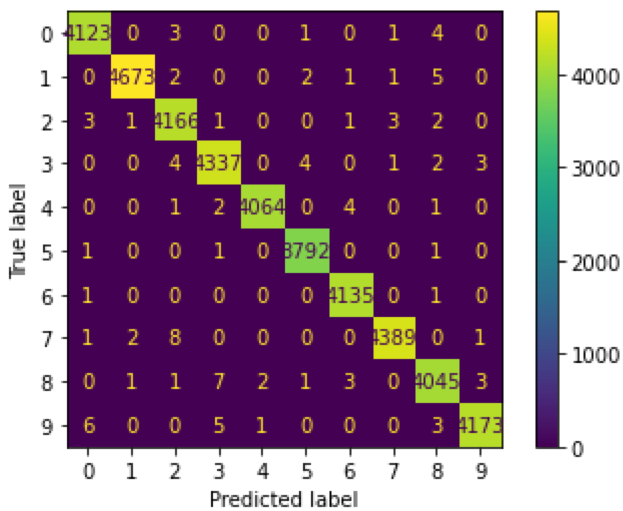
Build the initial model using the testing data

In [15]:
```
# Import libraries
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, (
import datetime


# Extract predictors and outcome (label variable)
X_train_SD = digit_training_data.copy(deep=True)
X_train_SD.drop(['label'], axis=1, inplace=True)
y_train_SD = digit_training_data['label']


# Standardize the features
xscaler = StandardScaler()
X_train_SD = xscaler.fit_transform(X_train_SD)

# Initialize MLP Classifier
mlp_class = MLPClassifier(random_state=1, hidden_layer_sizes =(20, 20, 20, 20, 20), sc

# Create paramater grid with hyperparameters to tune, use default adam solver so doen
param_grid = {
    'learning_rate': ['adaptive'],
    'learning_rate_init': [0.001]
}

# Kfold cv with 5 splits for GridSearch
cv = KFold(n_splits=5, shuffle=True, random_state=1)

# Create the GridSearchCV with kfold=5 object and fit it to the training data
nn_start = datetime.datetime.now()
grid_search = GridSearchCV(mlp_class, param_grid, cv=cv, scoring='accuracy')
grid_search.fit(X_train_SD, y_train_SD)

# Print the time to fit the neural net model
nn_end = datetime.datetime.now()
nn_runtime = nn_end - nn_start
print(f"The total run time for the model creation was {nn_runtime}.")

# Print the best hyperparameters found
print("Best Hyperparameters:", grid_search.best_params_)
```

```
# Save the best estimator
best_model = grid_search.best_estimator_

# Save dictionary of mean accuracy scores from models into 'scores' variable
#dict_results = grid_search.cv_results_
#scores = dict_results['mean_test_score']

# Use the best model to predict using training data
y_pred_train_SD = best_model.predict(X_train_SD)

# evaluate the model on the training data
accuracy_train_SD = accuracy_score(y_train_SD, y_pred_train_SD)
print("Training Accuracy:", accuracy_train_SD)
print("Training Classification Report:", classification_report(y_train_SD, y_pred_trai

# Create the confusion matrix of the predictions
cm = confusion_matrix(y_train_SD, y_pred_train_SD)
ConfusionMatrixDisplay(confusion_matrix=cm).plot();
```

```
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
C:\Users\steve\anaconda3\lib\site-packages\sklearn\neural_network\_multilayer_percept
ron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (200) reache
d and the optimization hasn't converged yet.
  warnings.warn(
```

```
The total run time for the model creation was 0:08:21.102617.
Best Hyperparameters: {'learning_rate': 'adaptive', 'learning_rate_init': 0.001}
Training Accuracy: 0.997547619047619
Training Classification Report:               precision    recall  f1-score   support

           0       1.00      1.00      1.00      4132
           1       1.00      1.00      1.00      4684
           2       1.00      1.00      1.00      4177
           3       1.00      1.00      1.00      4351
           4       1.00      1.00      1.00      4072
           5       1.00      1.00      1.00      3795
           6       1.00      1.00      1.00      4137
           7       1.00      1.00      1.00      4401
           8       1.00      1.00      1.00      4063
           9       1.00      1.00      1.00      4188

    accuracy                           1.00     42000
   macro avg       1.00      1.00      1.00     42000
weighted avg       1.00      1.00      1.00     42000
```



Apply MLP Classifier to test data

In [16]:
```python
# Create a dataframe for predictor variables in the test dataframe for mlpclass model
mlpclass_testing_x_SD = digit_testing_data.copy(deep=True)

# Standardize the features using same scaler as training data
mlpclass_testing_xscale_SD = xscaler.transform(mlpclass_testing_x_SD)

# Apply the mlpclass model to the test dataset
mlpclass_test_ypred_SD = best_model.predict(mlpclass_testing_xscale_SD)

# Put the kmeans predictions into a Pandas dataframe
prediction_df_mlpclass_SD = pd.DataFrame(mlpclass_test_ypred_SD, columns=['Label'])

# Add the ID column to the front of the mlpclass predictions dataframe
ImageId_series = pd.Series(range(1,28001))
prediction_df_mlpclass_SD.insert(0, 'ImageId', ImageId_series)

# Output predictions to csv
prediction_df_mlpclass_SD.to_csv('test_predictions_mlpclass_adaptive_001.csv', index=
```

Let's display the Kaggle results from the application of the MLP Classifier model on the test dataset

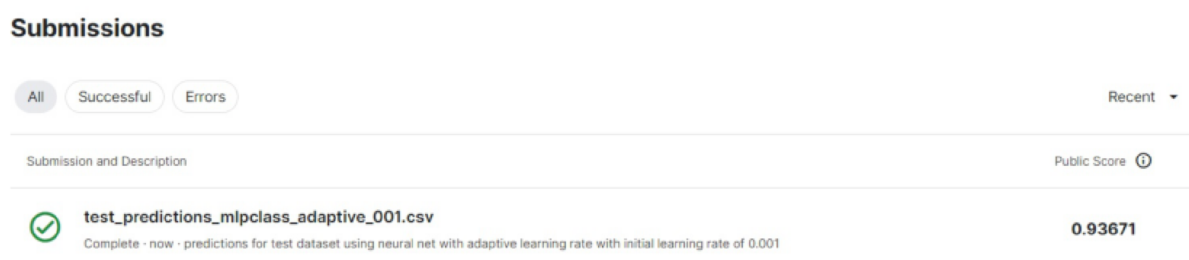
In [17]:

```
# Display the kaggle results associated with the MLP Classifier Model
import matplotlib.pyplot as plt
plt.figure(figsize = (15, 15))
kaggle_results = plt.imread('Kaggle_results_mlpclass_adaptive_001.jpg')
plt.imshow(kaggle_results)
plt.axis("off")
plt.show()
```

Out[17]: <Figure size 1080x1080 with 0 Axes>

Out[17]: <matplotlib.image.AxesImage at 0x2279a34f400>

Out[17]: (-0.5, 1496.5, 331.5, -0.5)



Compile Results From Each of the Four Trials

In [18]:

```
# Save layer and node data
Learning_Rate = ('Constant', 'Constant', 'Adaptive', 'Adaptive')
Initial_Learning_Rate = (0.01, 0.001, 0.01, 0.001)
Time = ('3 minutes and 16 seconds', '8 minutes and 33 seconds', '6 minutes and 39 secc
Training_Accuracy = (0.9920, 0.9975, 0.9995, 0.9975)
Testing_Accuracy = (0.9405, 0.9367, 0.9440, 0.9367)

# create dataframe with MLP details and scores for each mode and layer count tested
MLP_Scores_SD = pd.DataFrame({'Learning Rate' : Learning_Rate,
                             'Initial Learning Rate' : Initial_Learning_Rate,
                             'Time' : Time,
                             'Training Accuracy': Training_Accuracy,
                             'Testing Accuracy': Testing_Accuracy})
MLP_Scores_SD
```

Out[18]:

	Learning Rate	Initial Learning Rate	Time	Training Accuracy	Testing Accuracy
0	Constant	0.010	3 minutes and 16 seconds	0.9920	0.9405
1	Constant	0.001	8 minutes and 33 seconds	0.9975	0.9367
2	Adaptive	0.010	6 minutes and 39 seconds	0.9995	0.9440
3	Adaptive	0.001	8 minutes and 21 seconds	0.9975	0.9367

MLP Classifier with Kernel PCA features

In [5]:
```
digit_training_data.head()
```

Out[5]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775
0	1	0	0	0	0	0	0	0	0	0	...	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0
3	4	0	0	0	0	0	0	0	0	0	...	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0

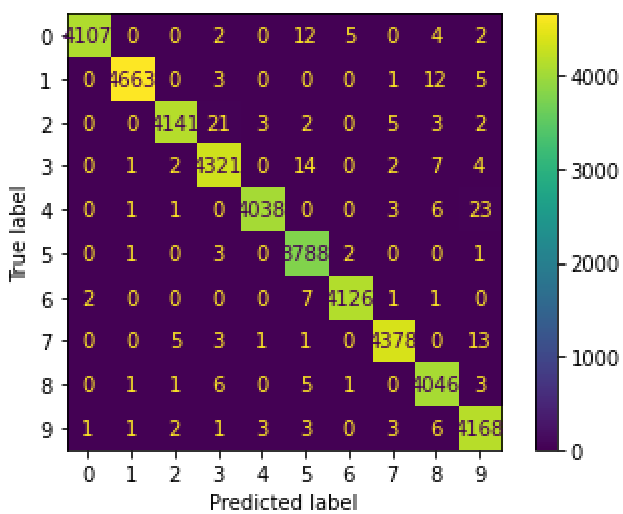
5 rows × 785 columns

In [18]:
```
# Scale PCA dataframe's data
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA, KernelPCA

sc = StandardScaler()
pca_scaled = sc.fit_transform(digit_training_data.drop(columns = ['label'])) # normali

# Convert scaled data from numpy array into dataframe
#pca_features = list(pca_df.columns.values)
pca_scaled_df = pd.DataFrame(pca_scaled)


# Applying PCA function on training and testing set of X component
from sklearn.decomposition import PCA
pca = PCA(n_components=334)
principal_components_digits = pca.fit_transform(pca_scaled_df)


# Create a Cumulative Scree plot to help us determine how many principal components to
import matplotlib.pyplot as plt
import numpy as np
```

In [19]:
```
### PCA ###

from sklearn.preprocessing import StandardScaler

sc = StandardScaler() #Initialize scaling of data
#nn_train_df.drop(['label'], axis=1, inplace=True) #drop the label column from the df

nn_train_x = principal_components_digits #set df without label as x
y_train = digit_training_data['label'] #set y a the label column
nn_train_df = pd.DataFrame(nn_train_x)
y_train = digit_training_data['label'] #set y a the label column

#sc.fit(nn_train_df)
#normalized = sc.transform(nn_train_df)

# Convert scaled data from numpy array into dataframe
```

```
nn_training_features = list(nn_train_df.columns)
nn_training_scaled_df = pd.DataFrame(nn_train_df, columns=nn_training_features)

# Import libraries
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, C

# Initialize MLP Classifier
mlp_class = MLPClassifier(random_state=1, hidden_layer_sizes=(20,20,20,20,20),
                        max_iter = 300,activation = 'relu',
                        solver = 'adam')

mlp_class.fit(nn_train_df, y_train)
y_pred_train = mlp_class.predict(nn_train_df)

# evaluate the model on the training data
accuracy_train = accuracy_score(y_train, y_pred_train)
print("Training Accuracy:", accuracy_train)
print("Training Classification Report:", classification_report(y_train, y_pred_train))

print("Training Accuracy:", accuracy_train)
print("Training Classification Report:", classification_report(y_train, y_pred_train))

# Create the confusion matrix of the predictions
cm = confusion_matrix(y_train, y_pred_train)
ConfusionMatrixDisplay(confusion_matrix=cm).plot();
```

```
Training Accuracy: 0.9946666666666667
Training Classification Report:               precision    recall  f1-score   support

           0       1.00      0.99      1.00      4132
           1       1.00      1.00      1.00      4684
           2       1.00      0.99      0.99      4177
           3       0.99      0.99      0.99      4351
           4       1.00      0.99      0.99      4072
           5       0.99      1.00      0.99      3795
           6       1.00      1.00      1.00      4137
           7       1.00      0.99      1.00      4401
           8       0.99      1.00      0.99      4063
           9       0.99      1.00      0.99      4188

    accuracy                           0.99     42000
   macro avg       0.99      0.99      0.99     42000
weighted avg       0.99      0.99      0.99     42000

Training Accuracy: 0.9946666666666667
Training Classification Report:               precision    recall  f1-score   support

           0       1.00      0.99      1.00      4132
           1       1.00      1.00      1.00      4684
           2       1.00      0.99      0.99      4177
           3       0.99      0.99      0.99      4351
           4       1.00      0.99      0.99      4072
           5       0.99      1.00      0.99      3795
           6       1.00      1.00      1.00      4137
           7       1.00      0.99      1.00      4401
           8       0.99      1.00      0.99      4063
           9       0.99      1.00      0.99      4188

    accuracy                           0.99     42000
   macro avg       0.99      0.99      0.99     42000
weighted avg       0.99      0.99      0.99     42000
```



In [ ]:
```
### KERNEL ###

kernel_pca = KernelPCA(
    n_components=100, kernel="rbf")

kernel_data = kernel_pca.fit_transform(pca_scaled_df)
```

```
kernel_digits_df = pd.DataFrame(kernel_data)

#kernel_data.eigenvalues_

# Confirm scaling transformation was a success
#kernel_digits_df.shape
#kernel_digits_df.head(10)
#kernel_digits_df.describe()

#from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall
#from sklearn.model_selection import train_test_split
#from sklearn.preprocessing import MinMaxScaler

# Initialize MLP Classifier
mlp_class = MLPClassifier(hidden_layer_sizes=(20,20,20,20,20),
                        max_iter = 300,activation = 'relu',
                        solver = 'adam')

mlp_class.fit(kernel_df, y_train)
y_pred_train = mlp_class.predict(nn_train_df)

# evaluate the model on the training data
accuracy_train = accuracy_score(y_train, y_pred_train)
print("Training Accuracy:", accuracy_train)
print("Training Classification Report:", classification_report(y_train, y_pred_train))

print("Training Accuracy:", accuracy_train)
print("Training Classification Report:", classification_report(y_train, y_pred_train))

# Create the confusion matrix of the predictions
cm = confusion_matrix(y_train, y_pred_train)
ConfusionMatrixDisplay(confusion_matrix=cm).plot();
```

```
In [ ]:
```