CLASSIFICATION OF MNIST DIGIT IMAGES VIA NEURAL NETWORK MODELS

Steve Desilets

MSDS 458: Artificial Intelligence and Deep Learning

October 8, 2023

1. Abstract

Machine learning algorithms capable of consuming and interpreting image data have existed for decades and have delivered fantastic results for many organizations, such as the United States Postal Service (USPS). In this paper, we leverage Modified National Institute of Standards and Technology Database (MNIST) data to conduct five rounds of experimentation in which we create ten artificial neural network models to classify observations as corresponding to the digits zero through nine. Throughout the construction of these models, we examine the important impacts that model architecture has on model performance and we delve into the features extracted by these models via analyses of their hidden nodes' activated values. Like previous researchers who paved the way for this study at organizations like USPS, we find that artificial neural networks can serve as excellent tools for classifying images into their corresponding numerical digit classes.

2. Introduction

The United States Postal Service (USPS) processes and delivers 23.8 million packages and 421.4 million pieces of mail while relying on a self-funded operational model that has forced the USPS to innovate to keep up with the rising flow of mail throughout its 248-year existence (United States Postal Service 2023; Madrigal 2011). One of the most significant breakthroughs in data science that empowered the USPS to accelerate its mail delivery systems was the creation of Optical Character Recognition (OCR) models capable of automatically reading and classifying digits, such as those included in zip codes. After such OCR algorithms began enjoying commercial success in the mid-20th Century, the USPS quickly adopted such algorithms to streamline operations in its bulk mail centers (Wikipedia 2023; National Postal Museum 2023).

In this paper, we will construct and examine the effectiveness of ten image classification models similar to those employed by the USPS to recognize digits in zip codes so the USPS can maximize its mail sorting and delivery efficiency. The underlying data we leverage (the MNIST dataset) consists of 60,000 28 pixel by 28 pixel training images of the digits 0 through 9, along with a testing dataset of 10,000 additional images, which were all compiled by the National Institute of Standards and Technology

(Keras 2022). We construct ten single-layer artificial neural network models via varying architecture and development methodologies so that we can gain a better understanding of the impacts of model design on performance and so we can delve into the features extracted by the nodes in the model hidden layers and examine their efficacy at discriminating between digits.

3. Literature Review

Though the origins of optical character recognition algorithms date back to as early as 1870 when Fournier d'Albe and Tauschek began work on developing devices to assist visually impaired individuals read, progress in the field accelerated rapidly in the 20th Century (Wikipedia 2023). One important breakthrough in the field of digit image classification occurred in 1957 when Frank Rosenblatt invented the perceptron capable of carrying out binary classification tasks (Martinez Ojeda 2022). Since then, researchers have developed even more sophisticated methods for classifying data into mutli-class categories, such as the digits 0 through 9 – on which we focus in this study. In fact, Javier Martinez Ojeda provides an excellent tutorial on how to construct single hidden layer classifiers of digit image data and how to examine the model's extracted features, though we will try to see whether we can outperform his testing accuracy rate of 99.72% via experimentation with model architecture (Martinez Ojeda 2022). This MNIST digit classification exercise has become such a popular tool for introducing data scientist to neural networks that the rolling Kaggle MNIST digit classification competition leaderboard (which only included the last two months of entrants) includes 4,049 entries from 1,203 competitors (AstroDave and Cukierski 2012). This study aims to build upon the findings of all these previous data scientists by researching the impacts of model architecture on MNIST digit classification neural networks and via thorough examination of the features extracted by the hidden nodes in these models.

4. Methods

For this project, we constructed 10 neural network models throughout the five primary rounds of experimentation with MNIST image classification models. At the beginning of this process, we first imported the MNIST data and conducted data cleaning exercises, like examining the shape of the data, visualizing a subset of the images, creating barplots of the number of observations corresponding to each

digit in the training and testing datasets, and reshaping the 28 x 28 matrices of data into 1 x 784 vectors. We then constructed our first artificial neural network model, which consisted of a 784-dimensional dense input layer, followed by a hidden layer with one node and a ReLu activation function, and an output layer that classified the observations into one of 10 classes via a softmax activation function. Prior to creating this model, we rescaled all the input data to be between 0 and 1 (instead of being between 0 and 255) and during the creation of this model and we reserved 5,000 of the 60,000 original training observations for a separate validation dataset. After constructing this model, we examined the model's performance on the testing dataset and examined how effective the hidden node's activation values indicated it was performing at discriminating between digits via activation value boxplots and visualizations corresponding to the maximally activated node.

For the second phase of the experiment, we created a very similar artificial neural network model to classify reshaped and rescaled MNIST image data as corresponding to the digits 0 through 9, though there were slight differences in our methodology as well. One such difference is that the hidden layer in the second experiment's neural network model included two hidden nodes instead of one node. In this analysis, we also leveraged scatterplots to visualize how well the activation values of these two nodes appeared to be clustering digits together appropriately. While these differences between experiments 1 and 2 did arise, we also conducted many of the same model performance assessment processes in this round as well.

For the third round of MNIST image classification experimentation, we created six artificial neural network models to classify reshaped and rescaled MNIST data as corresponding to our 10 digit classes. Like the previous models, these models all began with a 784-dimensional dense input layer, followed by a single hidden layer that used a ReLu activation function and an output layer that leveraged a softmax activation function to classify observations into our 10 digit classes. To allow us to continue gaining a better understanding of the impacts of model architecture on model performance, we leveraged different numbers of nodes in the hidden layer for each experiment – with models 3, 4, 5, 6, 7, and 8 containing 10, 30, 80, 130, 200, and 500 nodes, respectively. After creating these models, we conducted

performance assessment and feature extraction evaluation exercises, such as creating confusion matrices and constructing T-SNE plots that displayed how well the high-dimensional activation value data clustered together in a visually intuitive two-dimensional space.

For the fourth round of experimentation, we first leveraged principal components analysis (PCA) to reduce the dimensionality of the MNIST data from 784 dimensions to 154 dimensions, which still preserved 95% of the underlying variation in the original image data. Since the model architecture from our first 8 experiments that had resulted in the highest testing accuracy had included 130 nodes in the hidden layer, we applied a similar architecture to this ninth neural network model. The PCA-reduced data was leveraged inserted into a neural network that began with a 154-dimensional dense input layer, followed by a 130-dimensional dense hidden layer with a ReLu activation function, and a dense output layer that leveraged a softmax activation function to classify MNIST data into our 10 digit classes. After constructing this model, we examined appropriate performance metrics derived from the training, validation, and testing datasets.

For the final round of experimentation, we again reduce the dimensionality of the data prior to creating our artificial neural network classification model, but this time, we employ Random Forest Classification to assist us with that dimensionality reduction instead of Principal Components Analysis. The first step of this process was to leverage our 784-dimensional MNIST data as inputs for a Random Forest Classification Model, which informed us which 70 of those pixels were the strongest predictors of digit classes. We then leveraged just those 70 pixels as the input data for a neural network classifier with one hidden layer with 130 nodes followed by a dense output layer that utilized the softmax activation function to classify observations into our 10 digit classes. After constructing this Random Forest Classifier-informed neural network model, we examined the appropriate performance metrics derived from application of the neural network classifier to our training, validation, and testing datasets.

5. Results

Though the first experiment leveraged a neural network with just one single node in its single hidden layer, even this very basic model yielded moderate success at classifying images into digit classes.

As displayed in Appendix A, the training, validation, and testing accuracies of this model were 39%, 40%, and 38%, respectively. Given that a random number generator selecting digits between 0 and 9 would likely only achieve around 10% accuracy, it's amazing that such a simple model would boost the testing accuracy by 28 percentage points to 38%. Despite this success, our model performance analyses also highlight opportunities for improvement. For example, the first model's confusion matrix (displayed in Appendix B) indicates that the model predicted zero images would be 5's and that very few images would be 0, 2, or 8, despite each of these digits comprising roughly 10% of the sample. Furthermore, the boxplot (provided in Appendix E) displaying the hidden node activation values for each observation (disaggregated by digit) conveys that a lot of overlap exists between the resulting activation values across digits, which means that the model could be better at discriminating between digit classes. Also, we visualize the pattern that maximizes the activation value of the hidden node and see that it resembles a zero, yet the model indicates that it would predict this shape is a 6. These findings suggest that while a neural network with a single hidden node may perform better than expected, it does retain significant opportunities for improvement.

The results of the second experiment, for which we added a second node to the neural network hidden layer, convey strong improvements over the results of the first experiment while still maintaining some room for improvement. For example, while the accuracy of this model was not perfect, the testing accuracy did jump from 38% in the first experiment up to 67% in this second experiment (as displayed in Appendix A). The confusion matrix for this second model (in Appendix B) also conveys that this two-hidden-node model begins predicting each digit in relatively proportionate frequencies – which is another major improvement over the first experiment's model. The scatterplots of the model's nodes' activation values color-coded by digit and visualized in Appendix E suggest that this model is actually performing quite well at clustering digits together. Though all these successes are promising, an examination of a sample of some of the most commonly misclassified digits from this model, as shown in Appendix C, conveys that this model does misclassify many images that humans would easily be able to sort correctly.

The six neural networks constructed during the third round of experimentation provide fascinating results. Perhaps the most striking result is that the testing accuracies of these models (as conveyed in Appendix A) were very high – ranging from 93% to 97%. These high accuracy rates are further reflected in the corresponding confusion matrices in Appendix B, which display near perfect categorization of images with the heatmap strongly highlighting the correct predictions in a diagonal line. Furthermore, examinations of some of the incorrectly classified digits in Appendix C reveals that a notable portion of the misclassified digits would likely also be misclassified by humans. Perhaps the most exciting finding is that the T-SNE Plots in Appendix E, which visualize high-dimensional clusters via a user-friendly two-dimensional image, convey that these plots perform well at clustering observations together by digit. One final notable finding is that the performance of these models seems to initially increase as more nodes are added to the single hidden layer – but only up until a point around 130 nodes (which were leveraged in our sixth model). Beyond that point, the testing accuracy begins to slightly decline – likely because we begin overfitting to the training data and increasing the model's variance.

The fourth experiment, for which we conducted principal components analysis prior to creating a single-hidden-layer neural network with 130 hidden nodes, yielded the best results of any of the ten neural network models constructed! In fact, as stated in Appendix A, the training, validation, and testing accuracies for this model were 100%, 97.9%, and 97.8%, respectively. These incredible results indicate that we successfully dramatically reduced the dimensionality of the data (from 784 dimensions to 154 dimensions) without sacrificing on model predictive performance at all. While the accuracy and loss curves by epochs displayed in Appendix D indicate that this model may be overfitting to the training data slightly, the model still performs remarkably well and would be our choice for the best of the ten models if we had to pick a single top-performing neural network model.

The fifth experiment, in which we leveraged a Random Forest Classification model to identify the 70 most important predictor pixels and subsequently inserted just the data from those 70 pixels into a single-hidden-layer neural network model with 130 hidden nodes, yielded moderately strong results. As conveyed in Appendix A, the testing accuracy of this model was 94.2%. Examination of the plots

displaying the training and validation accuracy across epochs during training suggests that this model could potentially be improved if we applied regularization to address a bit of overfitting that may be unnecessarily increasing the predictive variance. Perhaps one of the most interesting parts of this model's development process was the part (in Appendix F) in which we visualized the importance of the pixels via a heatmap and via plotting the pixels against the backdrop of one of the actual digit images from our sample. These visualizations suggest that the Random Forest Classifier correctly picked out some of the most important pixels, given that the most important pixels selected are concentrated close to the center of the images.

6. Conclusions

The ten neural network models constructed throughout our five experiments for this study suggest that artificial neural networks – even ones with just a single hidden layer – can serve as powerful tools for classifying images into correct classes of digits. The fact that the training accuracies of these models ranged from 38% to 98% also confirms that model architecture plays a highly significant role in model performance for these types of models. Furthermore, these experiments suggest that data scientists may be able to take advantage of certain dimensionality reduction techniques, like PCA, without sacrificing model predictive performance. If I were the data scientist tasked with advising an organization like the USPS about whether to proceed with implementing any of these models, I might first encourage them to consider experimenting with more advanced techniques like convolutional neural networks before proceeding. That said, if the client absolutely needed to move forward with the implementation of one of these 10 models, I would advise that they select the ninth model developed during experiment 4 since it achieved such stunning levels of predictive accuracy.

References

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

Keras. 2022. “MNIST digits classification dataset.” *Keras*. https://keras.io/api/datasets/mnist/

Madrigal, Alexis C. 2011. “Tech has saved the Postal Service for 200 Years – Today, It Won’t.” *The Atlantic*. https://www.theatlantic.com/technology/archive/2011/12/tech-has-saved-the-postal-service-for-200-years-today-it-wont/249946/

Martinez Ojeda, Javier. 2022. “Digit Classification with Single-Layer Perceptron. *Medium*. https://towardsdatascience.com/digit-classification-with-single-layer-perceptron-9a4e7d4d9628

National Postal Museum. 2023. “Systems At Work – 1988.” *Smithsonian - National Postal Museum.* https://postalmuseum.si.edu/exhibition/systems-at-work-the-exhibition/1988

United States Postal Service. 2023. “Postal Facts.” *USPS*. https://facts.usps.com/one-day/#:~:text=On%20average%2C%20the%20Postal%20Service%20processes%20and%20delivers%20162.1%20million,First%2DClass%20Mail%20each%20day.&text=On%20average%2C%20the%20Postal%20Service%20processes%20421.4%20million%20mail%20pieces,minute%20and%204%2C877%20each%20second.

Wikipedia. 2023. “Timeline of Optical Character Recognition.” *Wikipedia*. https://en.wikipedia.org/wiki/Timeline_of_optical_character_recognition

Appendix A – Accuracy Results From Experiments

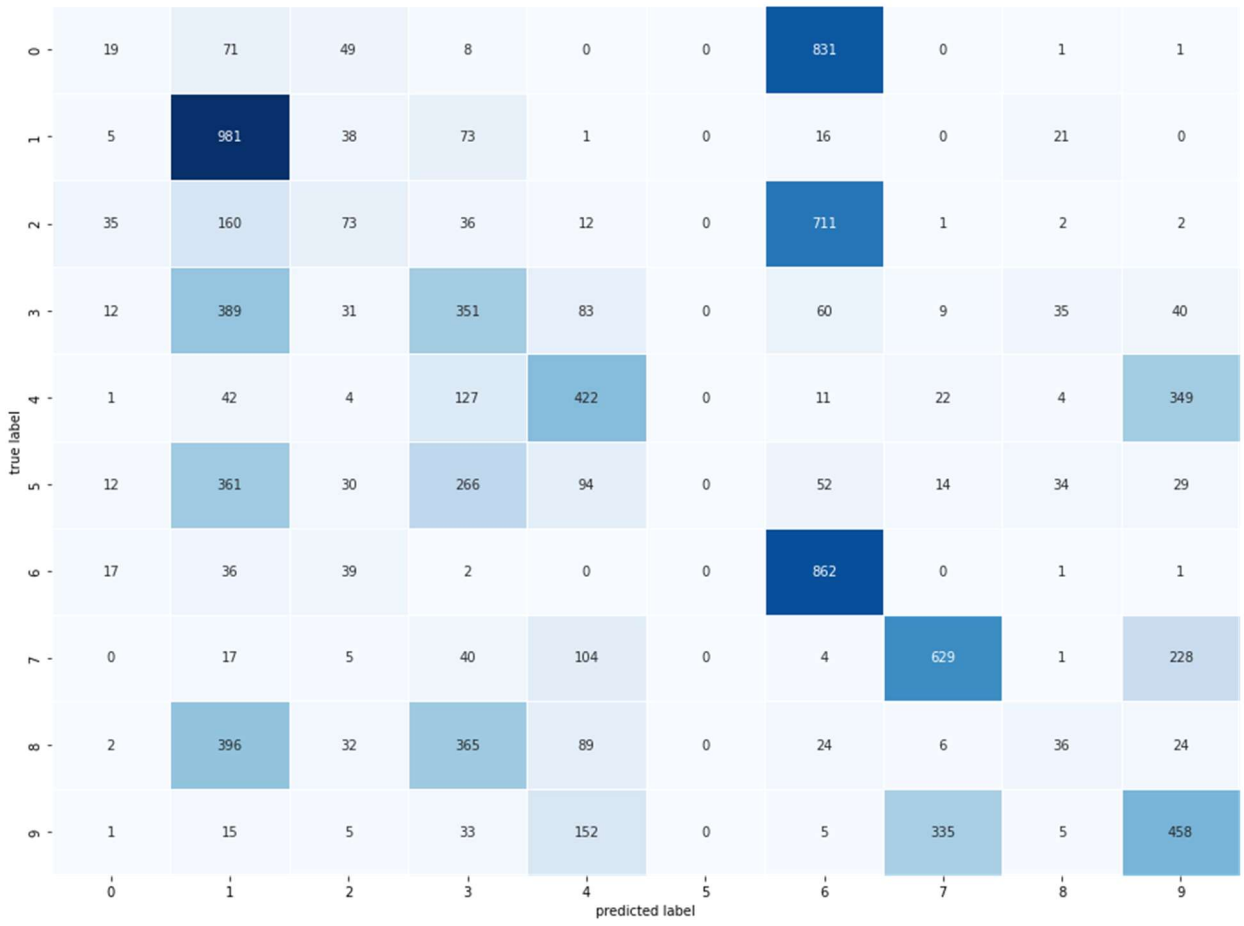
The table below displays the training, validation, and testing accuracy of each of the 10 neural network models developed for classification of MNIST images as digits.

Experiment Number	Model Inputs	Number of Hidden Layers	Number of Nodes in Hidden Layer	Training Accuracy	Validation Accuracy	Testing Accuracy
1	(Preprocessed) MNIST Data	1	1	39.4%	40.4%	38.3%
2	(Preprocessed) MNIST Data	1	2	66.6%	69.4%	66.9%
3	(Preprocessed) MNIST Data	1	10	92.5%	93.7%	92.7%
4	(Preprocessed) MNIST Data	1	30	96.3%	96.8%	96.5%
5	(Preprocessed) MNIST Data	1	80	97.0%	97.0%	96.5%
6	(Preprocessed) MNIST Data	1	130	97.0%	97.3%	97.1%
7	(Preprocessed) MNIST Data	1	200	96.8%	96.3%	96.5%
8	(Preprocessed) MNIST Data	1	500	96.3%	96.9%	96.5%
9	PCA-Reduced MNIST Data	1	130	100.0%	97.9%	97.8%
10	MNIST Data Deemed Most Important By Random Forest Classifier	1	130	96.1%	94.3%	94.2%

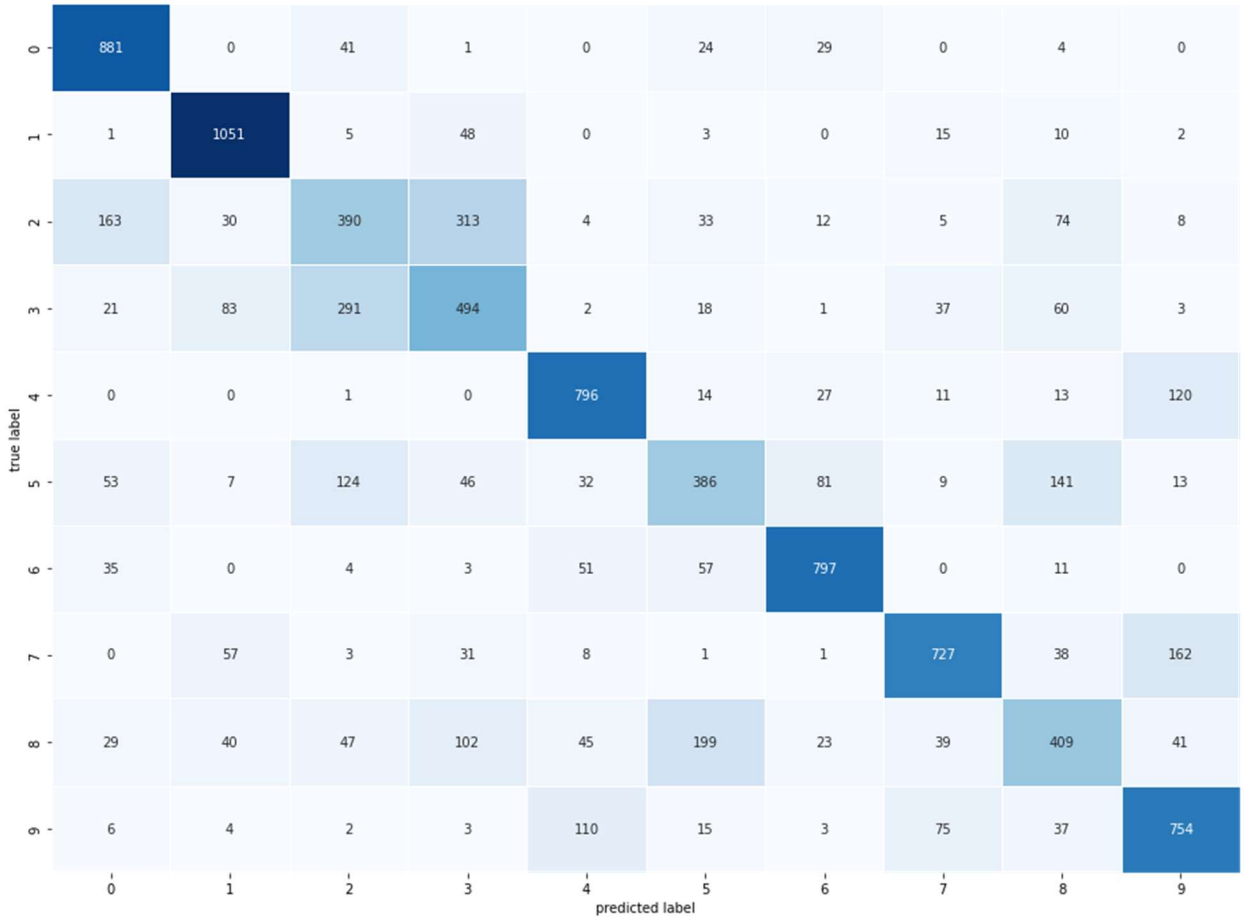
Appendix B – Confusion Matrices Resulting From Experiments

The images below display the confusion matrices resulting from the application of each MNIST neural network classification model the testing dataset. For ease of interpretation, these confusion matrices are color coded as heat maps. Larger versions of these images and the code leveraged to generate these confusion matrices are available in Appendix G.

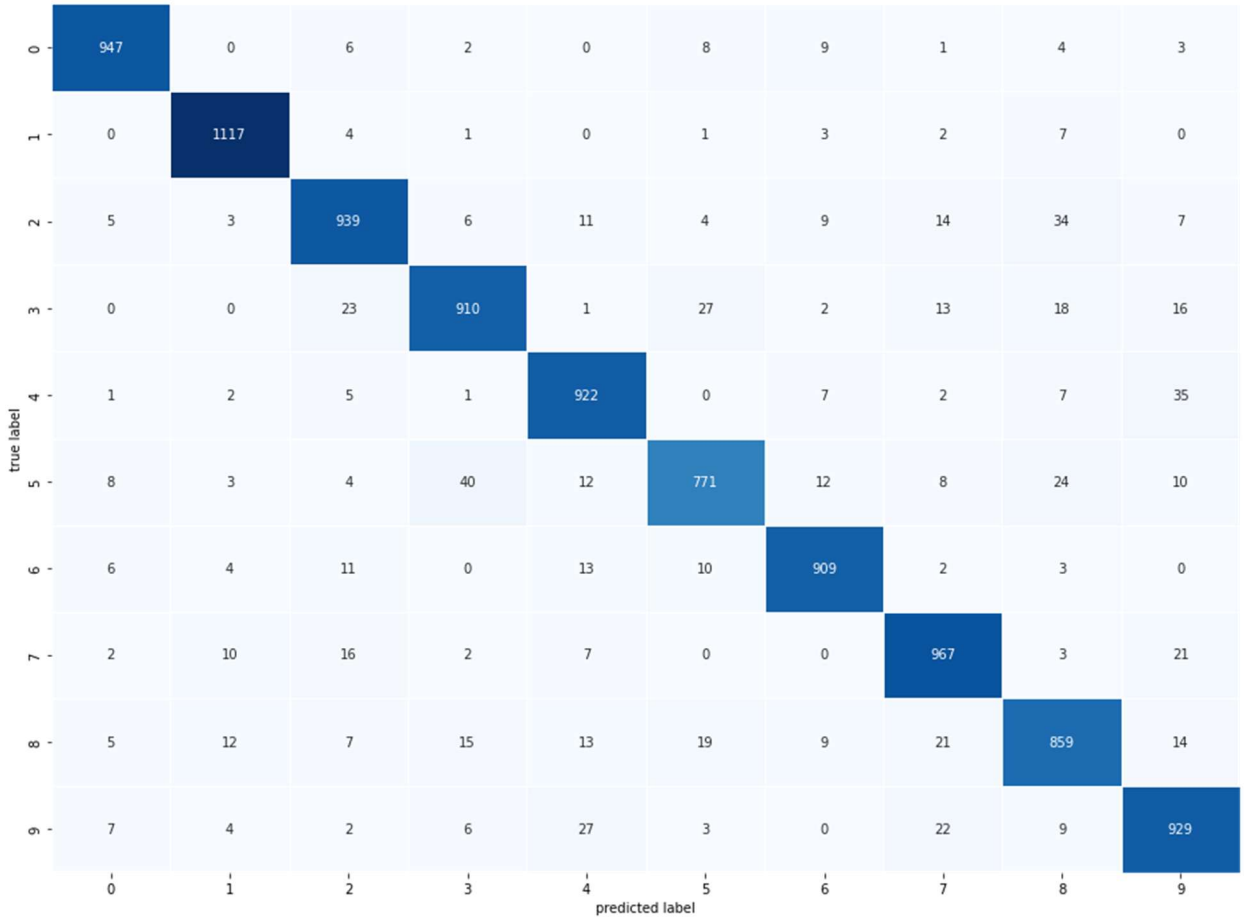
Experiment 1 – Confusion Matrix



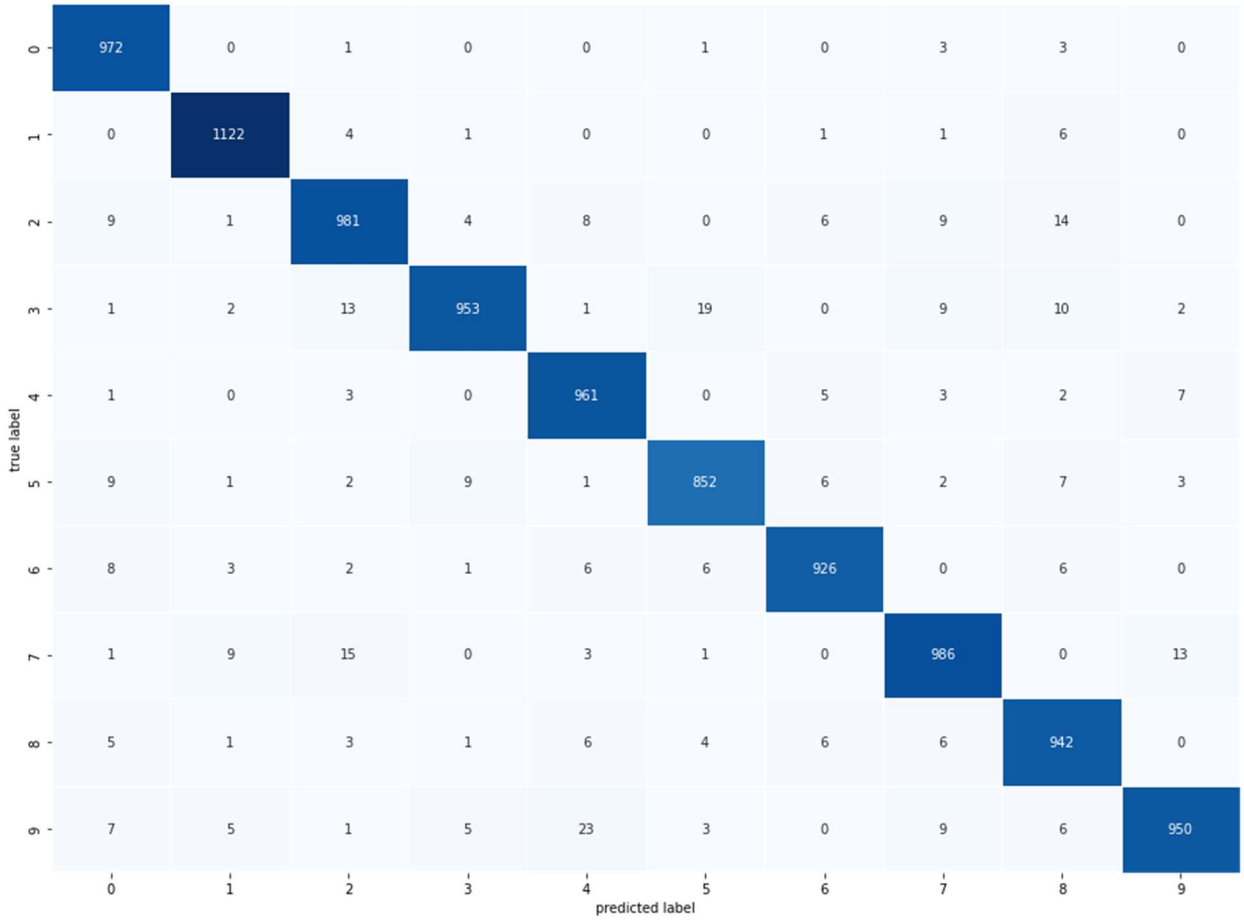
Experiment 2 – Confusion Matrix



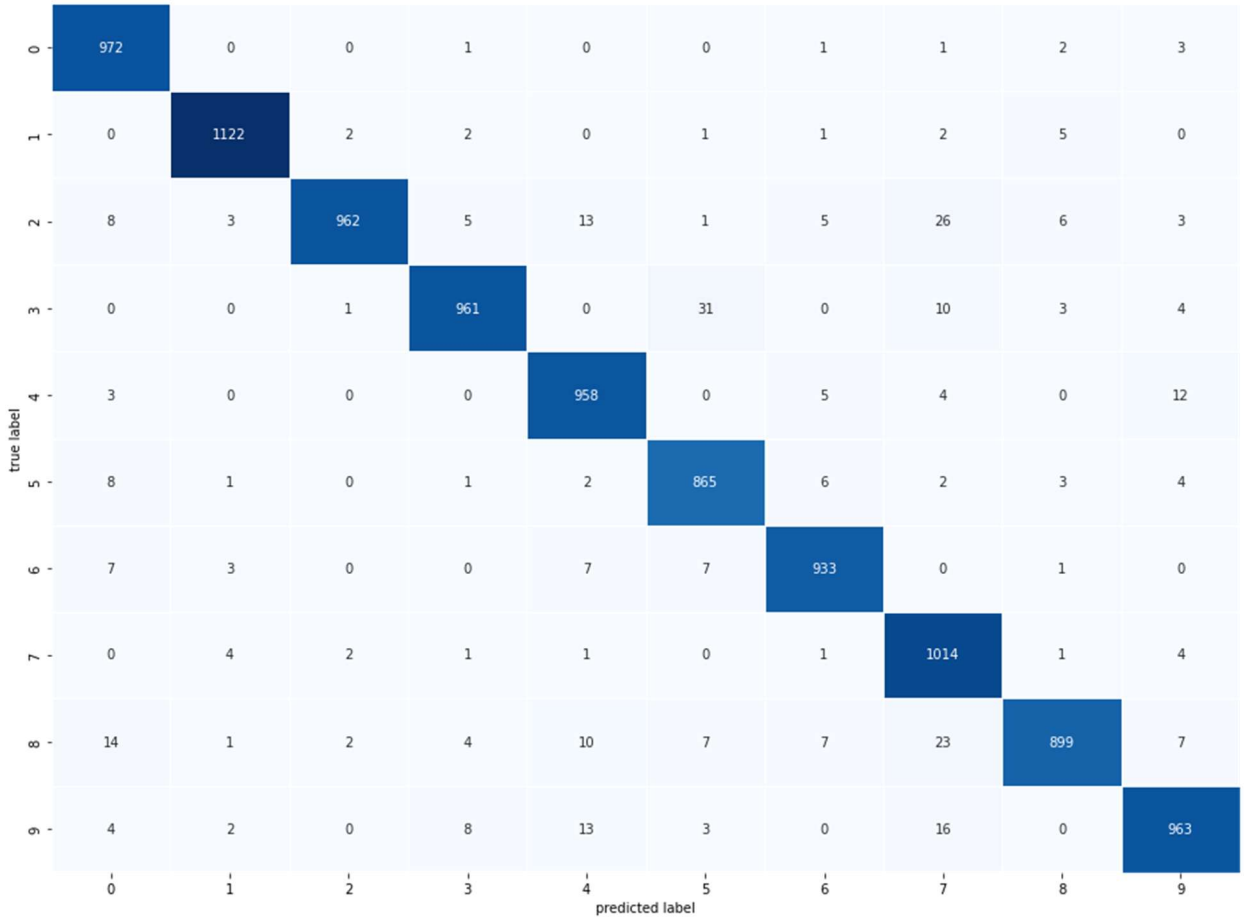
Experiment 3 – Confusion Matrix



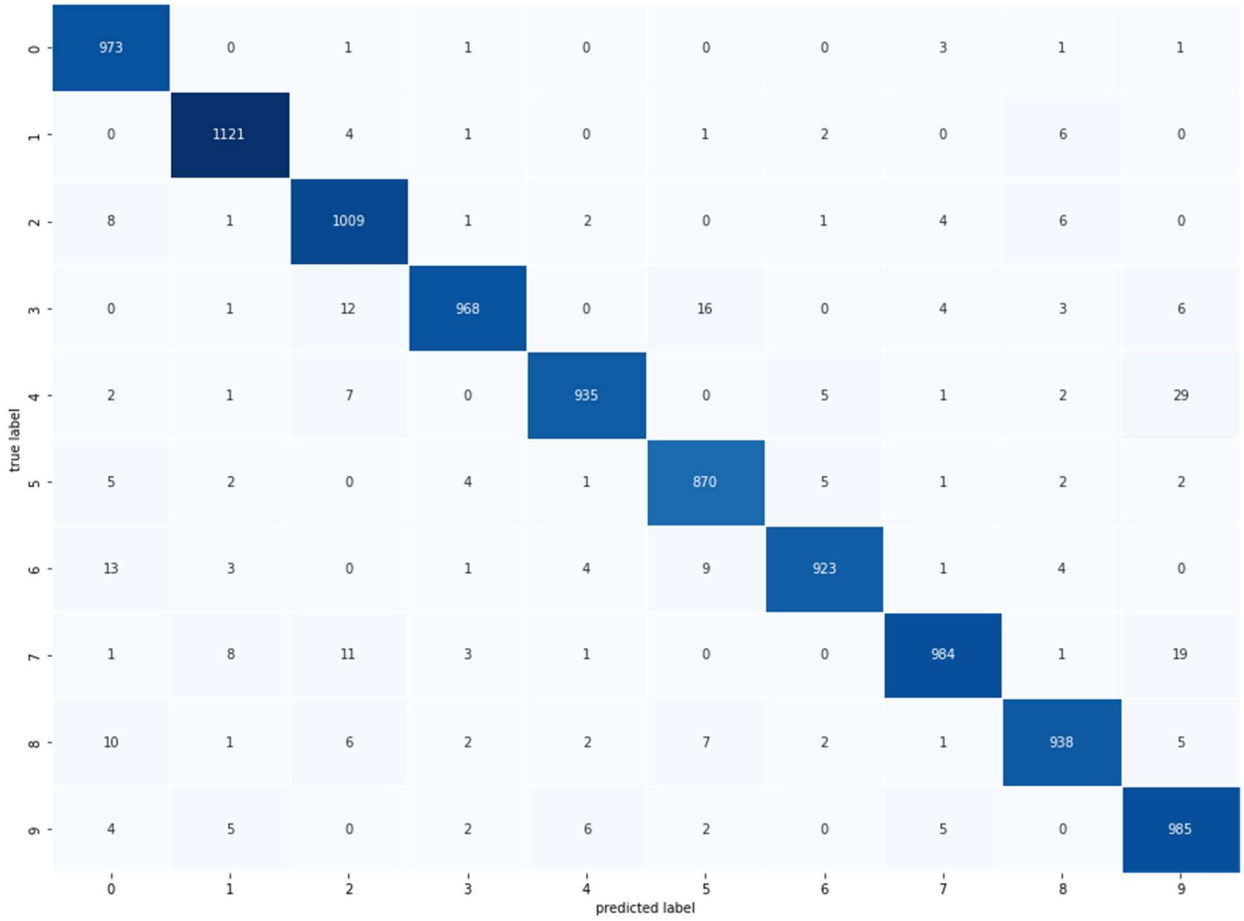
Experiment 4 – Confusion Matrix



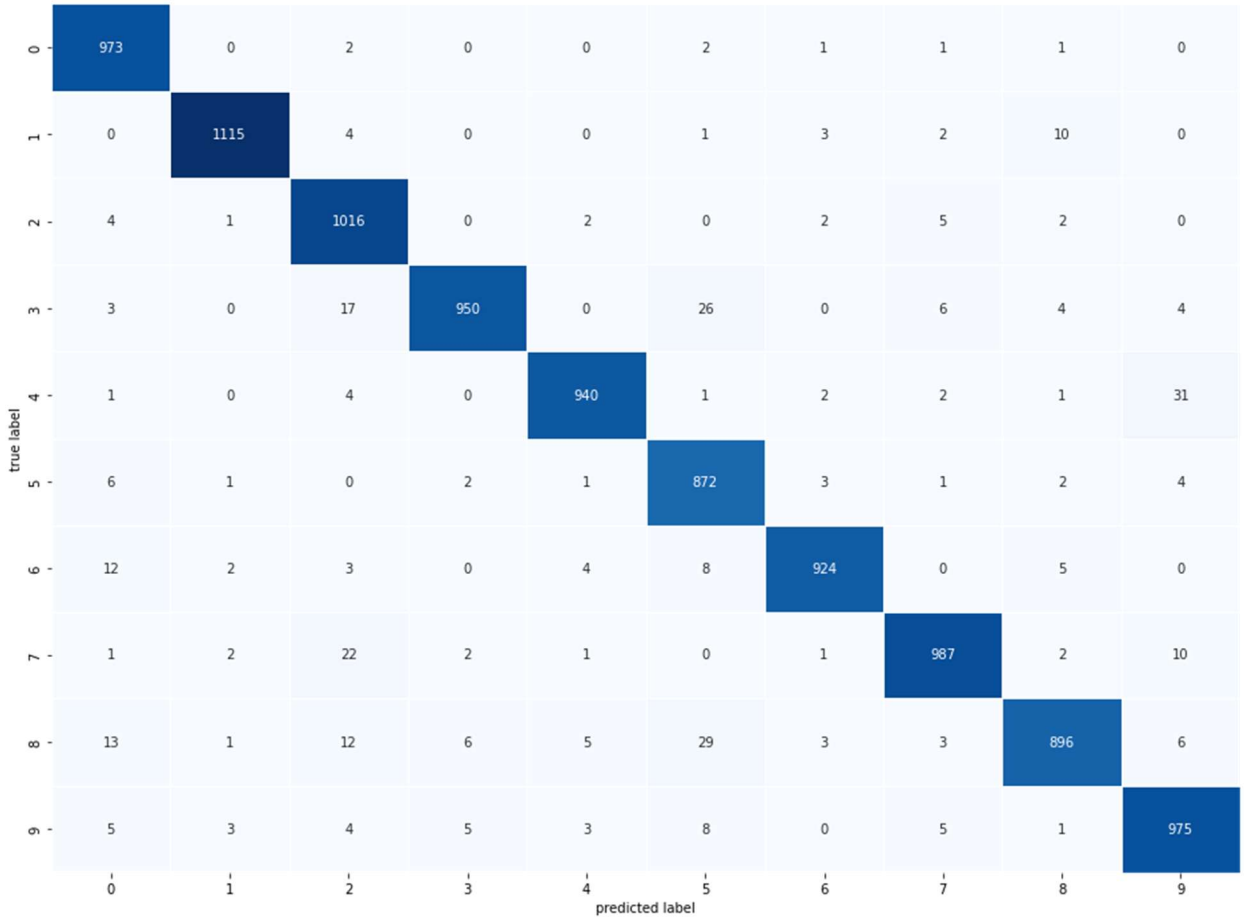
Experiment 5 – Confusion Matrix



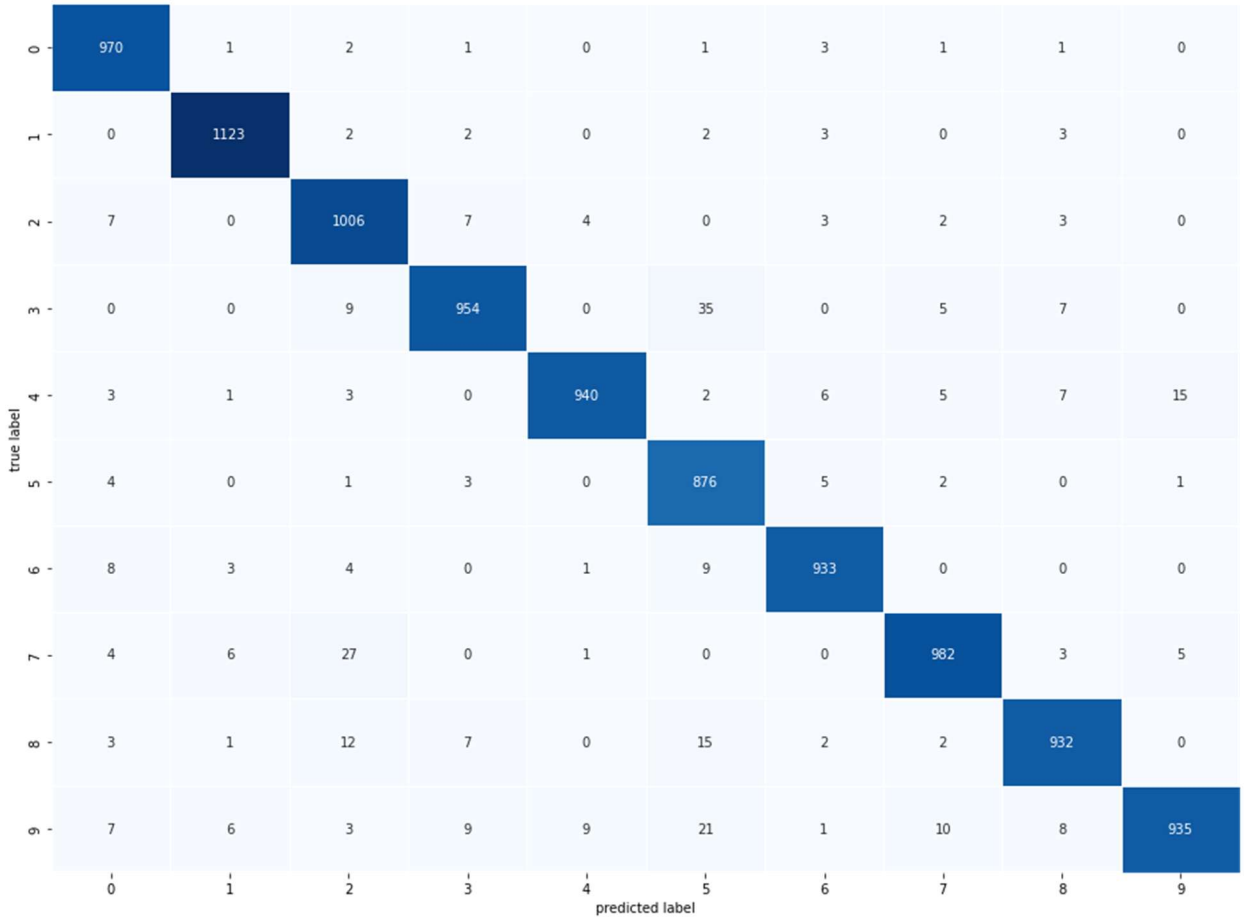
Experiment 6 – Confusion Matrix



Experiment 7 – Confusion Matrix



Experiment 8 – Confusion Matrix

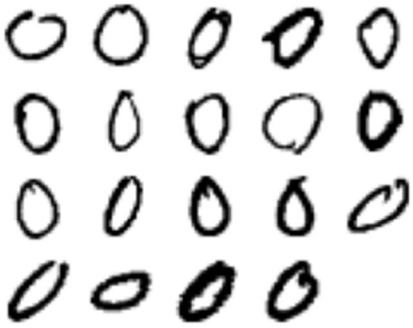


Appendix C – Most Commonly Misclassified Digit Combinations Resulting From Experiments

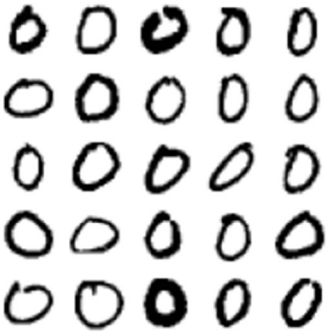
The images below display examples of some of the most frequently misclassified digits resulting from each of the MNIST image digit classification models. Larger versions of these images and the code leveraged to generate them are available in Appendix G.

Examples of Most Frequently Misclassified Digits From Experiment 1

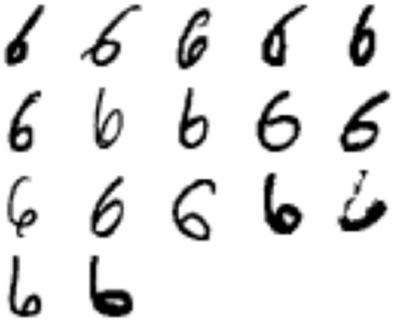
0's classified as 0's



0's classified as 6's



6's classified as 0's



6's classified as 6's

Examples of Most Frequently Misclassified Digits From Experiment 2

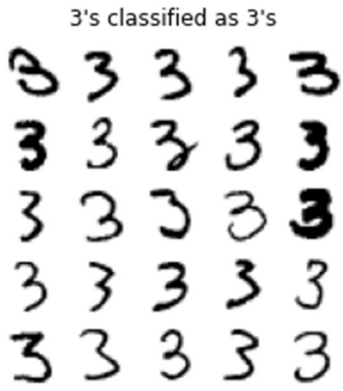
2's classified as 2's

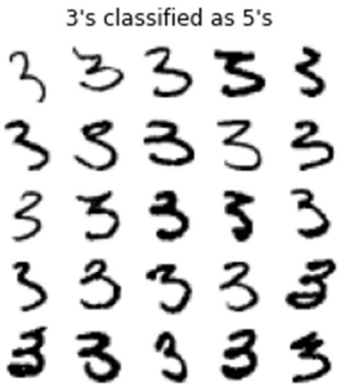
2's classified as 3's

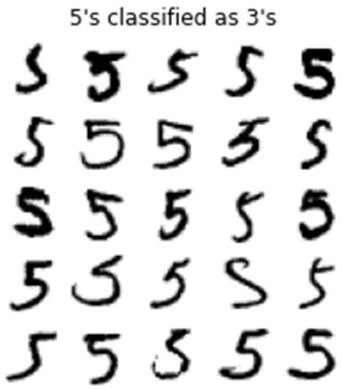
3's classified as 2's

3's classified as 3's

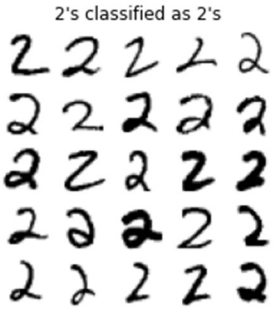
Examples of Most Frequently Misclassified Digits From Experiment 3

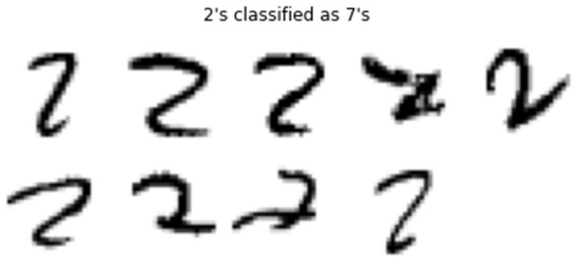


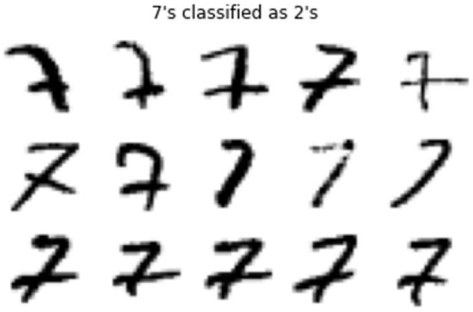




Examples of Most Frequently Misclassified Digits From Experiment 4







Examples of Most Frequently Misclassified Digits From Experiment 5

3's classified as 3's

3's classified as 5's

5's classified as 3's

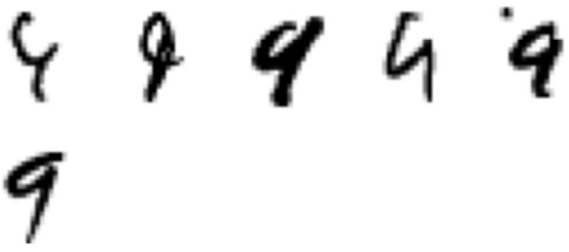
5's classified as 5's

Examples of Most Frequently Misclassified Digits From Experiment 6

4's classified as 4's

4's classified as 9's

9's classified as 4's



9's classified as 9's

Examples of Most Frequently Misclassified Digits From Experiment 7

4's classified as 4's

4's classified as 9's

9's classified as 4's

9's classified as 9's

Examples of Most Frequently Misclassified Digits From Experiment 8

3's classified as 3's

3's classified as 5's

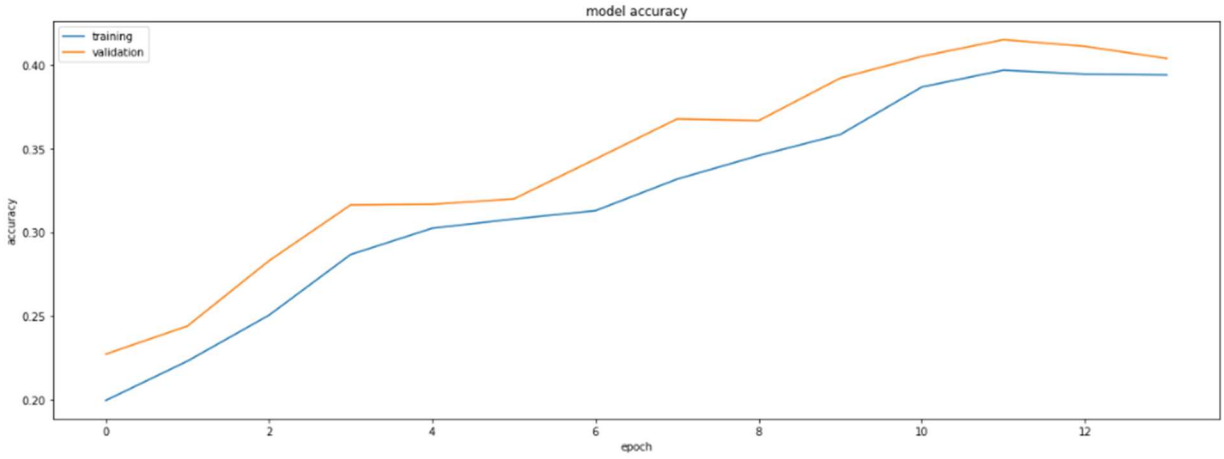
5's classified as 3's

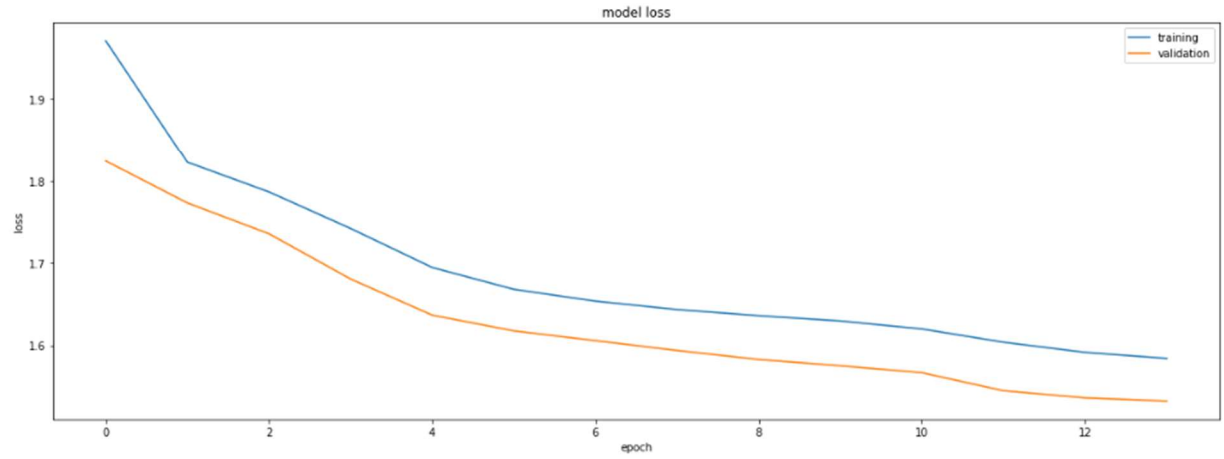
5's classified as 5's

Appendix D – Accuracy and Loss Trends By Epoch Resulting From Experimental Model Fitting

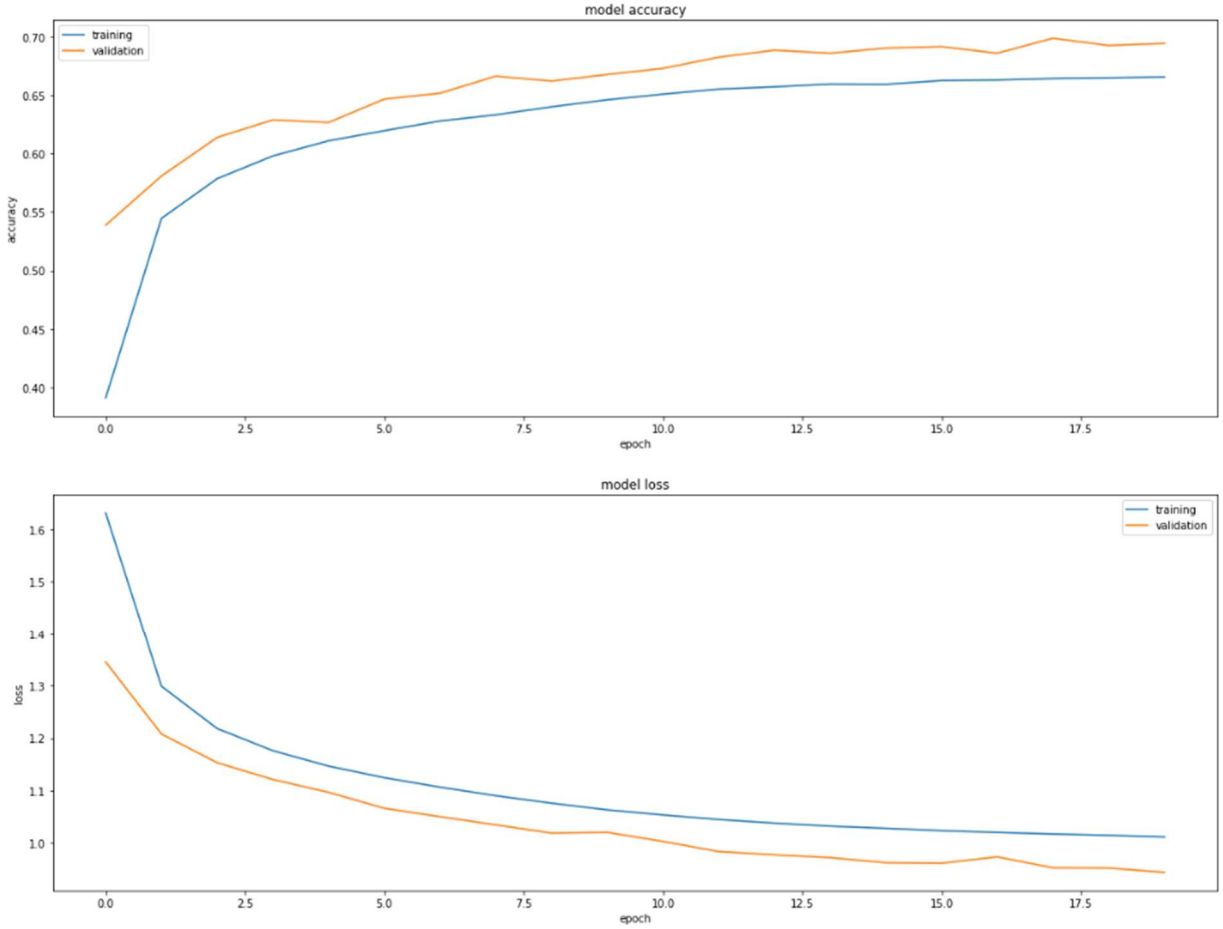
The graphs below display the training and validation accuracies generated throughout each epoch of training each of the MNIST image digit classification models. Larger versions of these images and the code leveraged to generate these graphs are available in Appendix G.

Experiment 1 – Accuracy and Loss Trends During Model Fitting

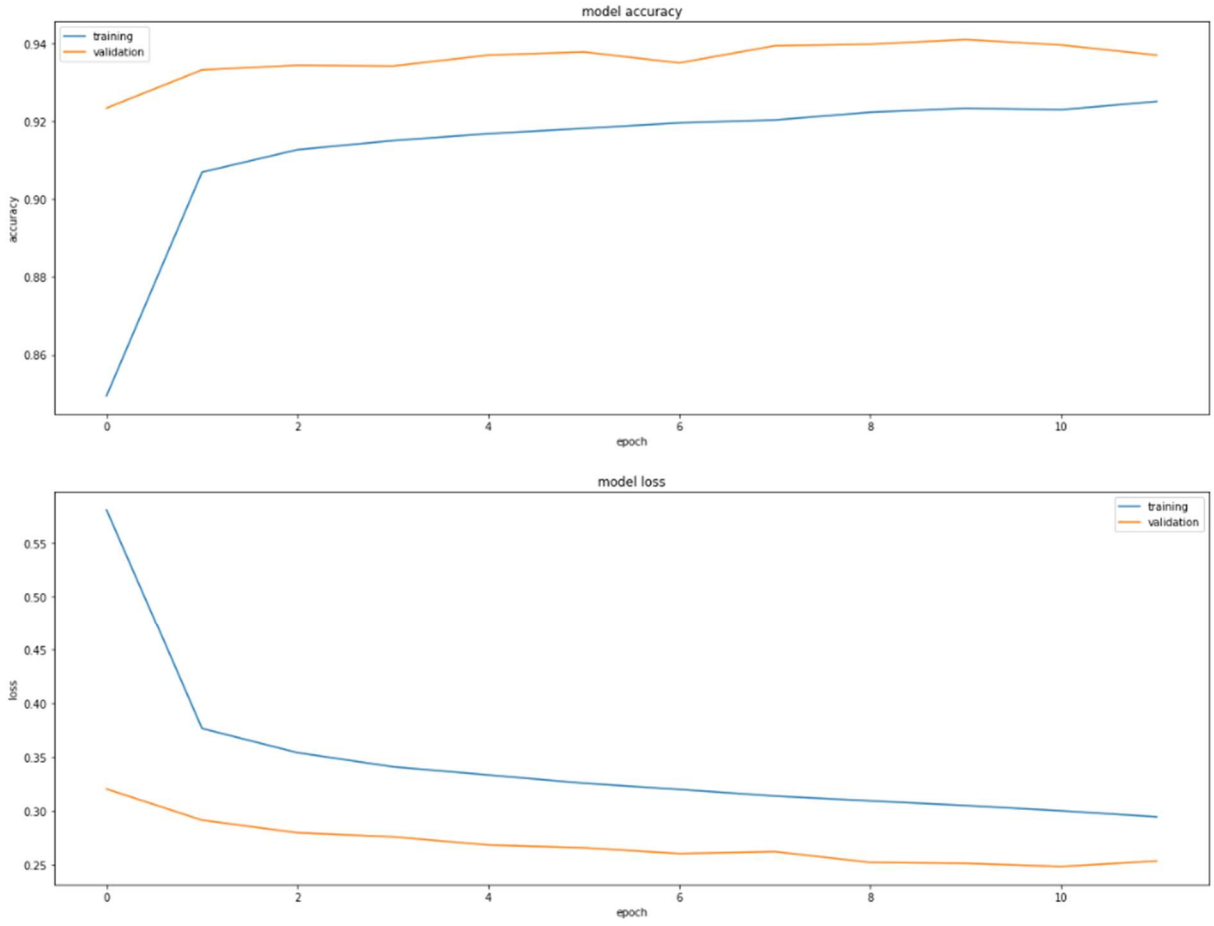




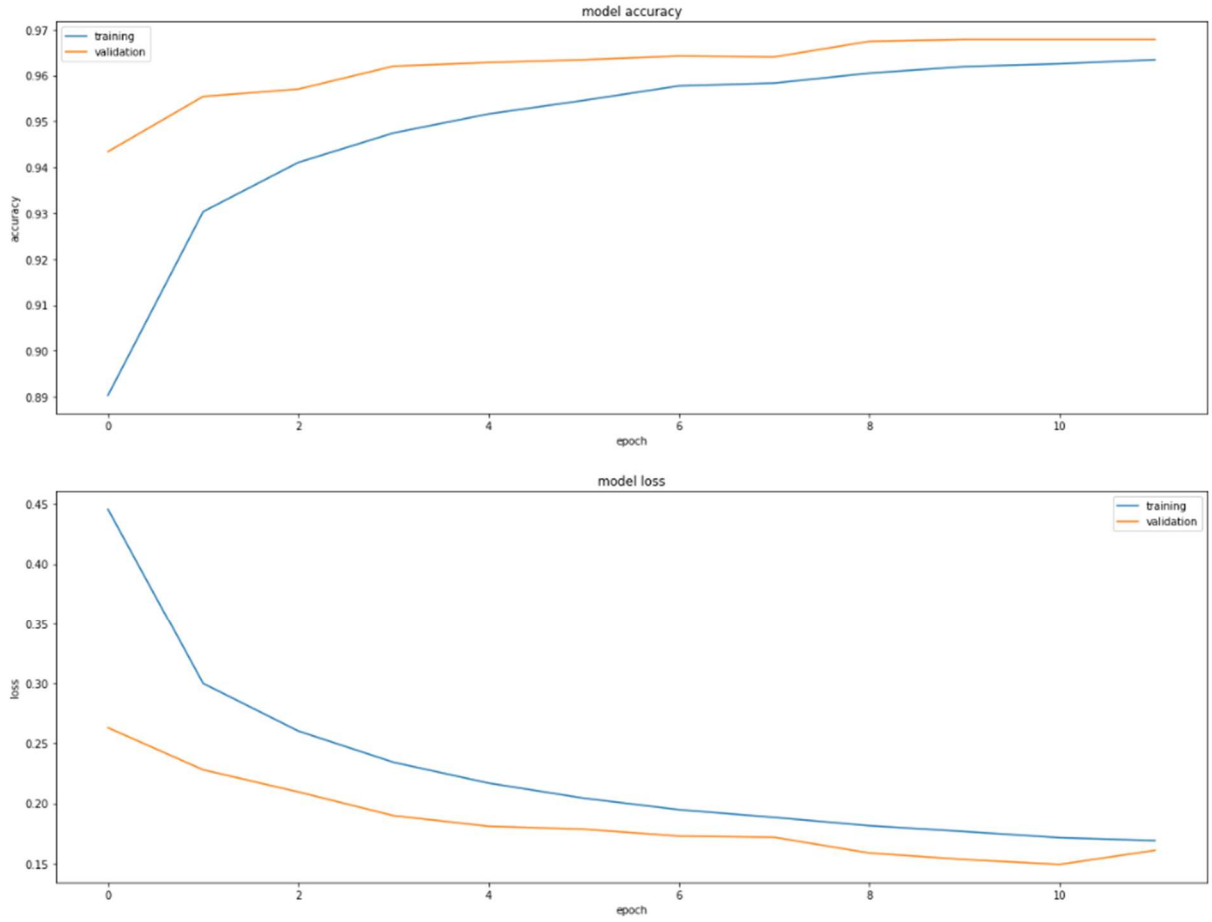
Experiment 2 – Accuracy and Loss Trends During Model Fitting



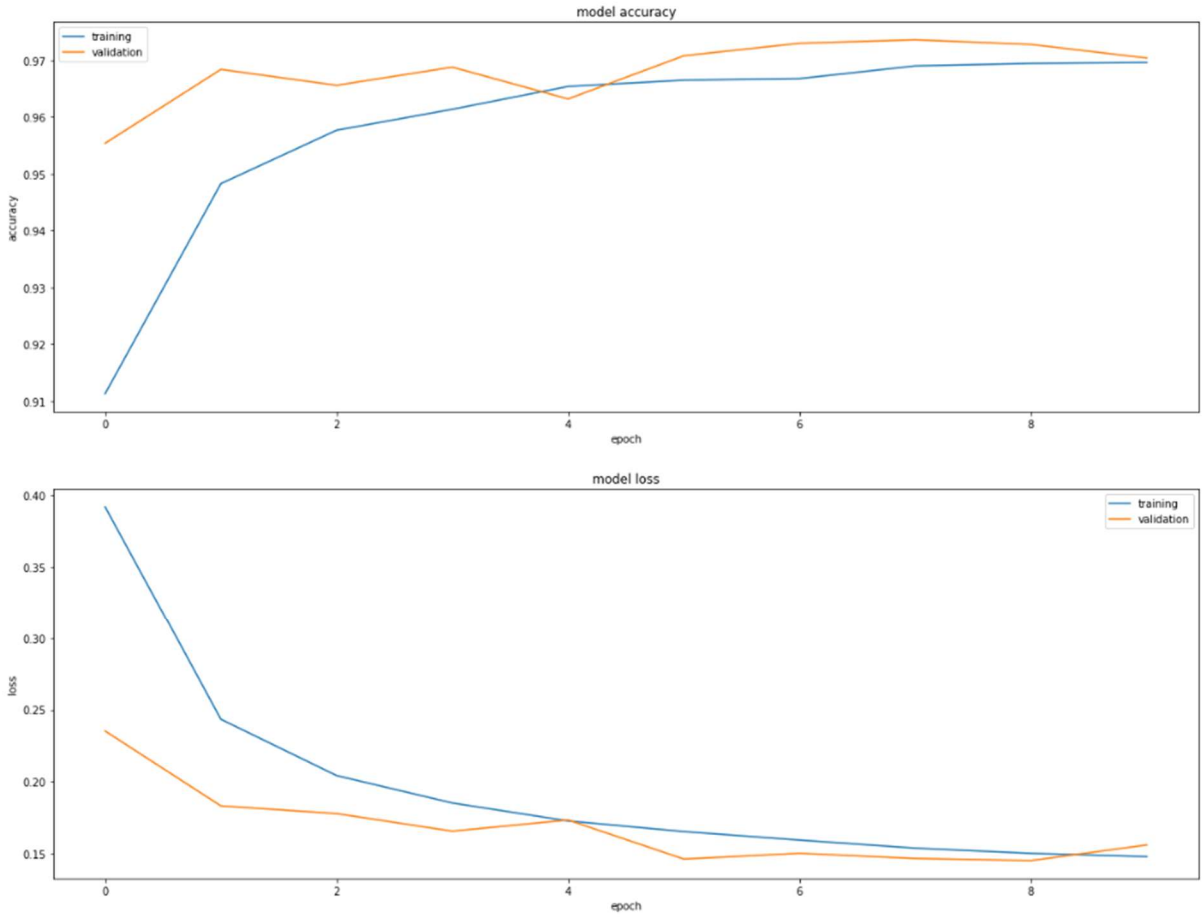
Experiment 3 – Accuracy and Loss Trends During Model Fitting



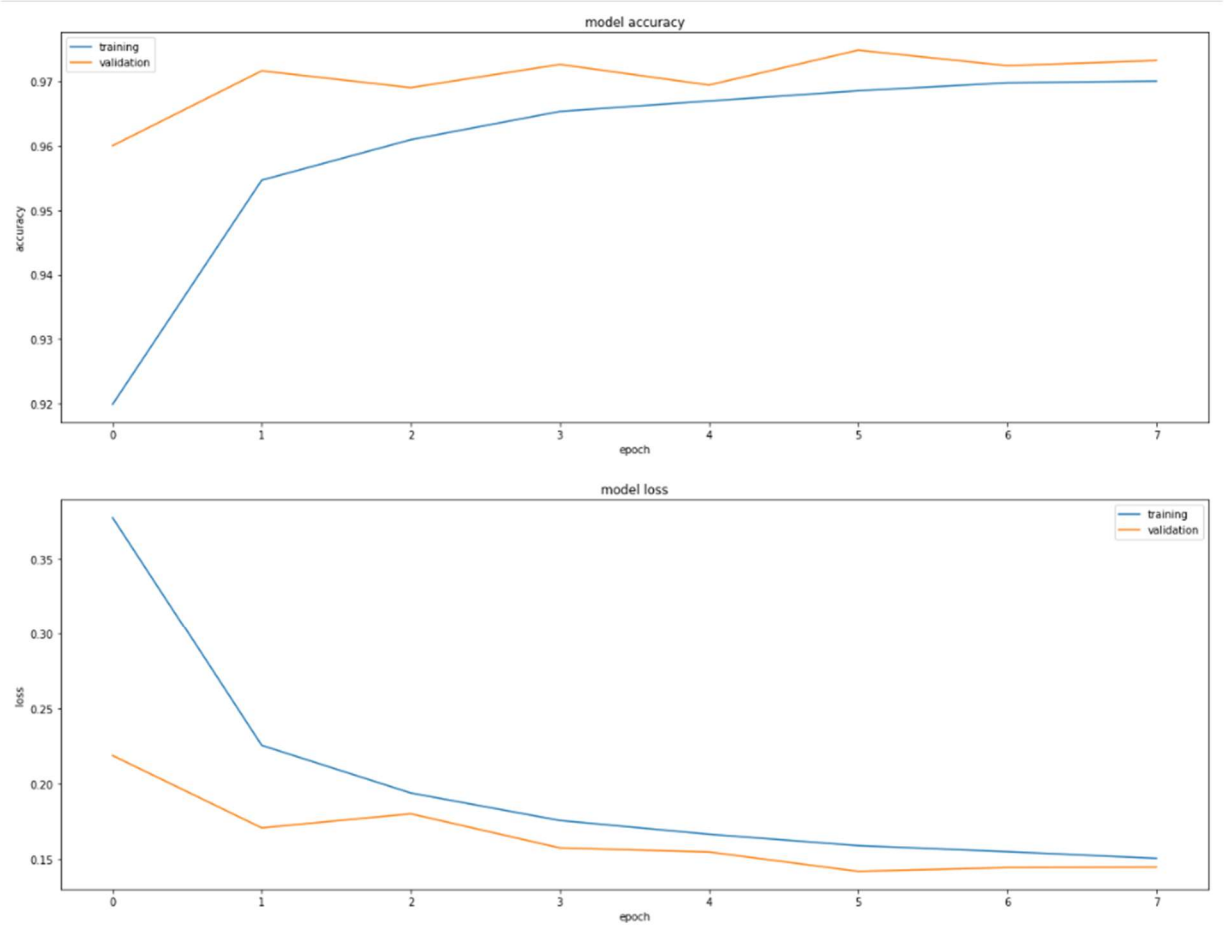
Experiment 4 – Accuracy and Loss Trends During Model Fitting



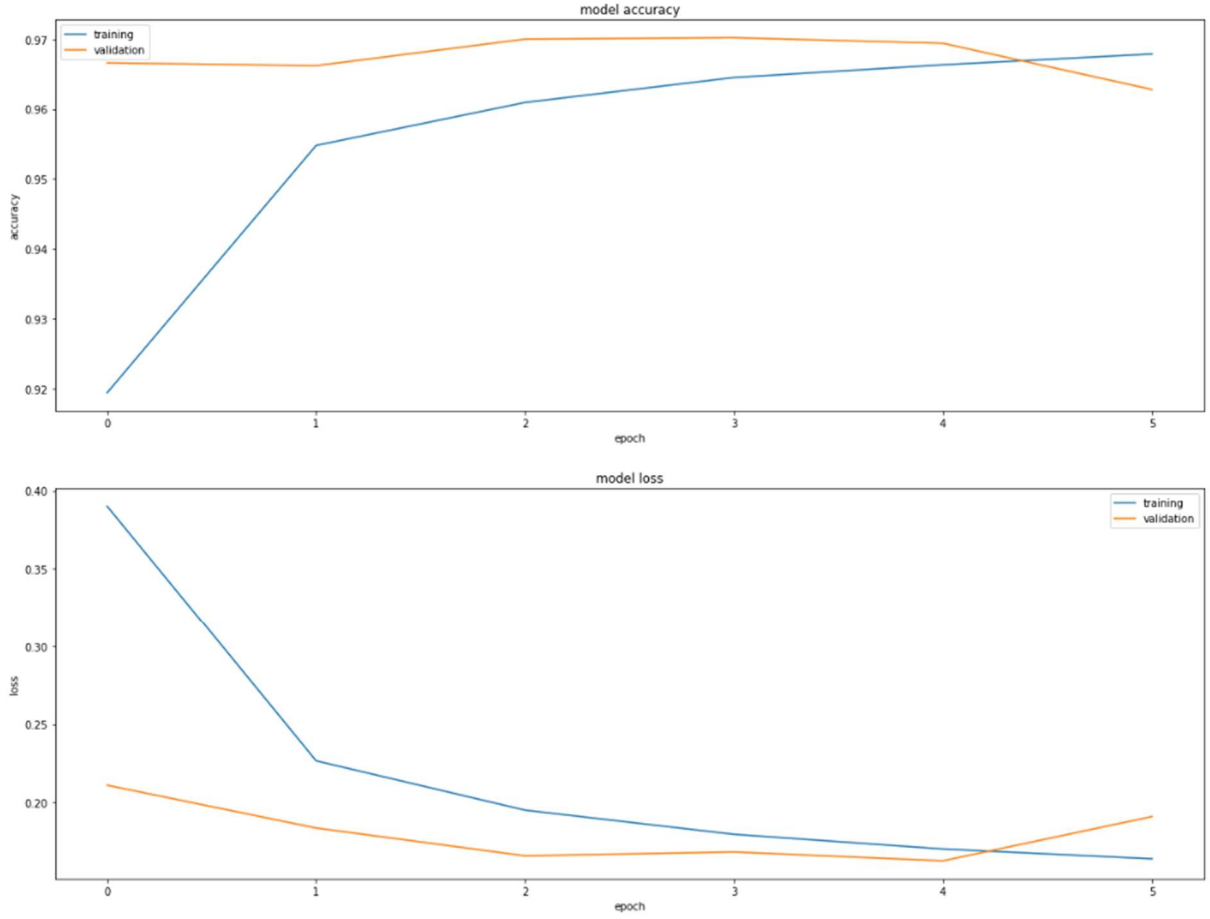
Experiment 5 – Accuracy and Loss Trends During Model Fitting



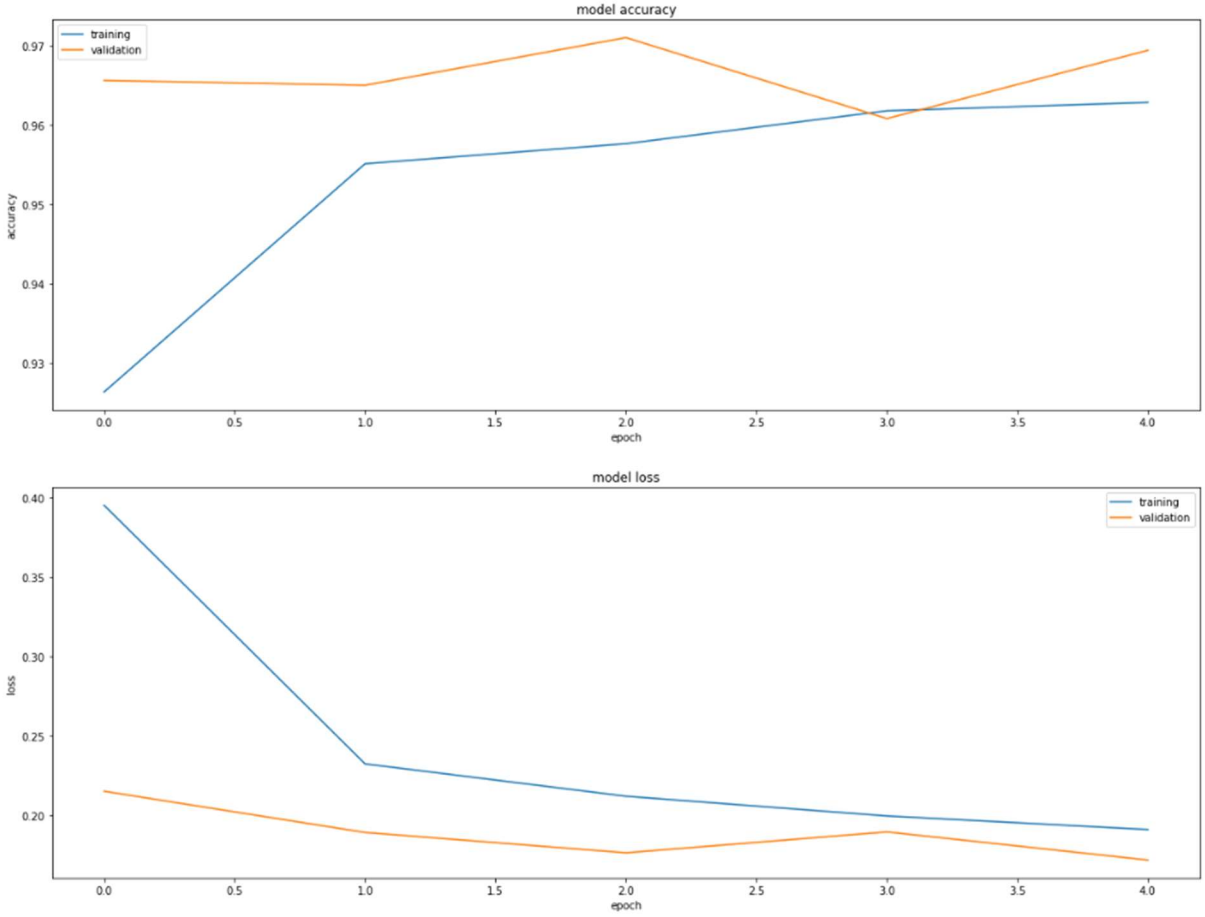
Experiment 6 – Accuracy and Loss Trends During Model Fitting



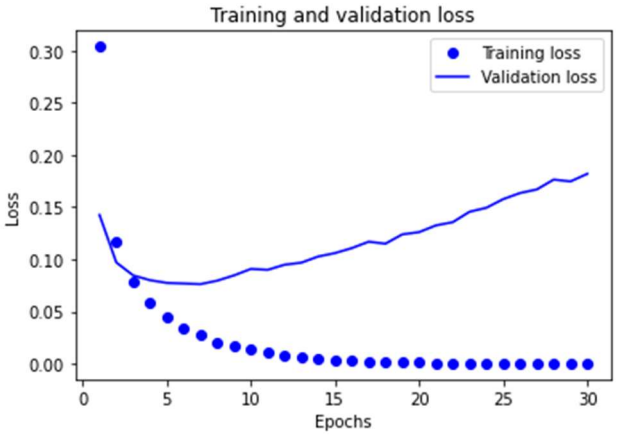
Experiment 7 – Accuracy and Loss Trends During Model Fitting

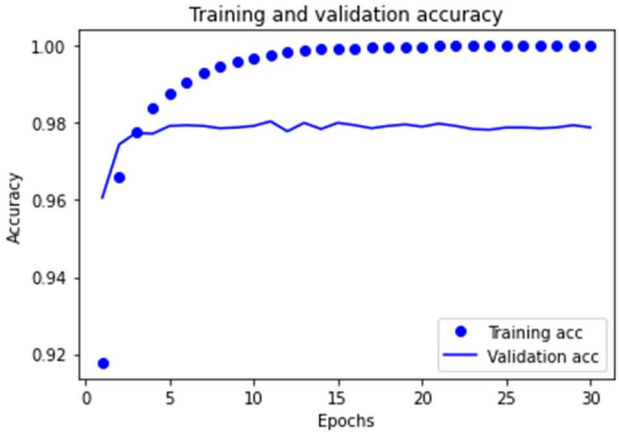


Experiment 8 – Accuracy and Loss Trends During Model Fitting

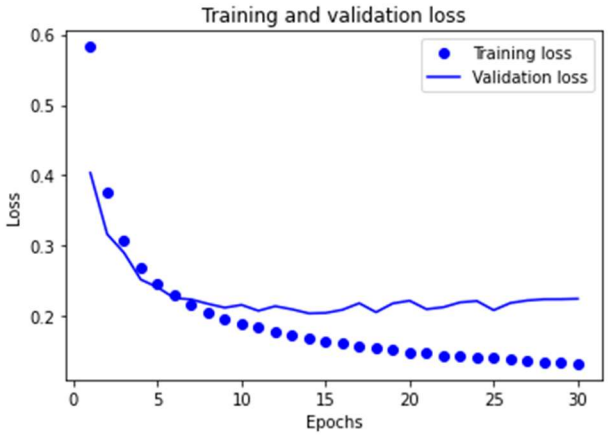


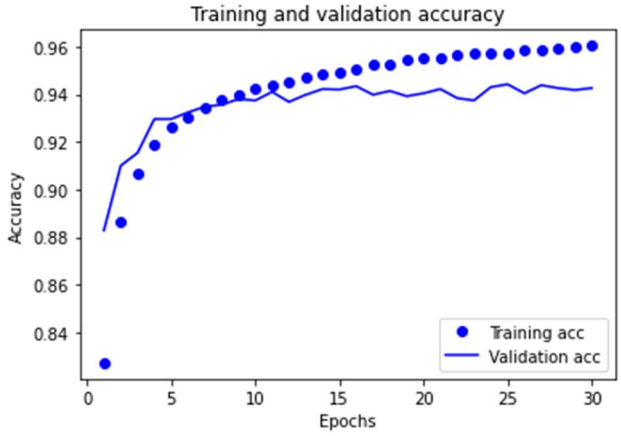
Experiment 9 – Accuracy and Loss Trends During Model Fitting





Experiment 10 – Accuracy and Loss Trends During Model Fitting

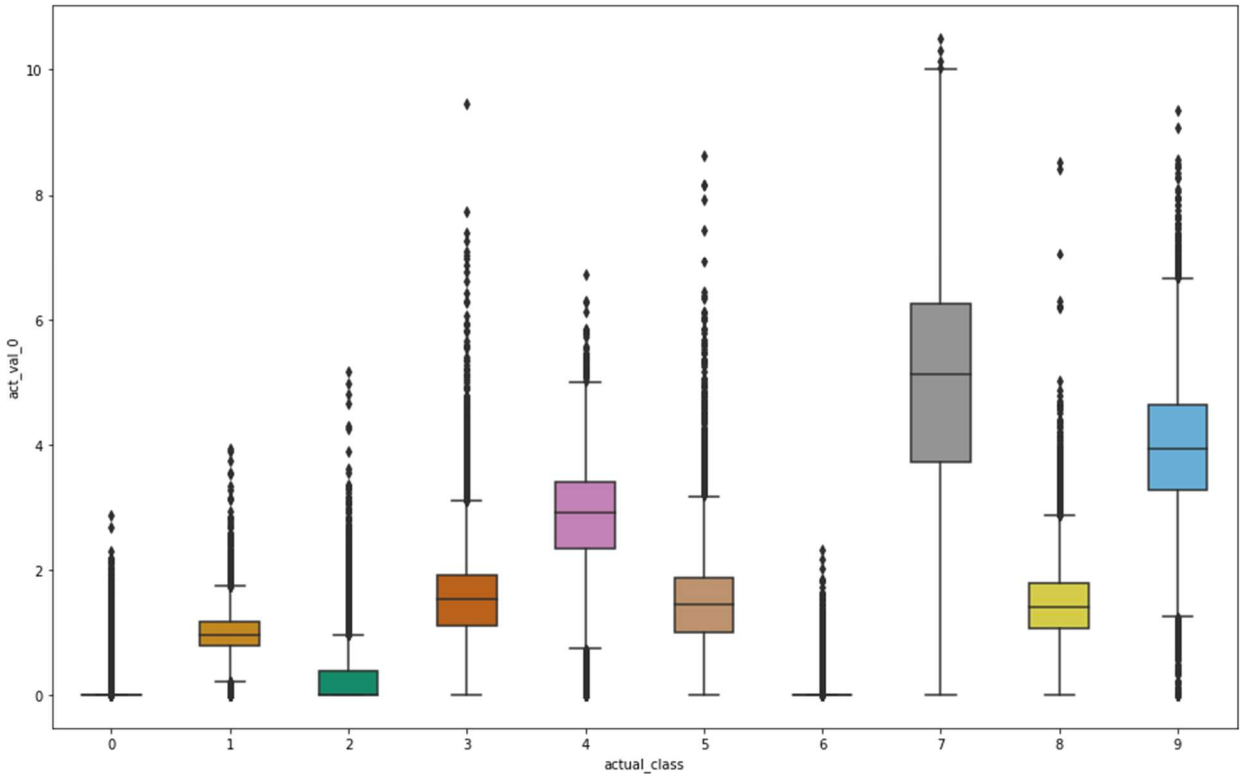




Appendix E – Activation Value Analyses From Experiments

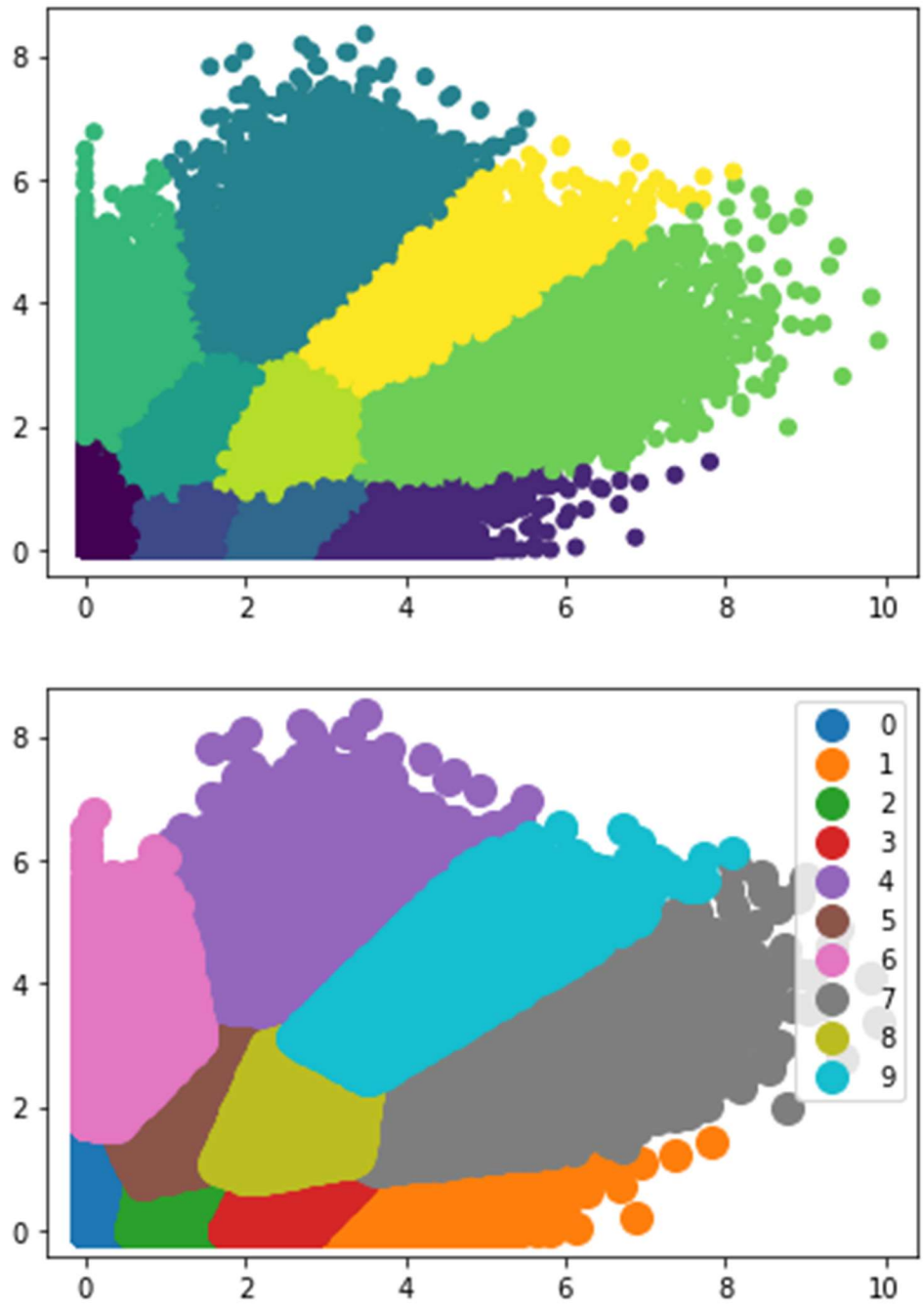
The images below visualize the results of activation value experiments designed to provide insight into features extracted by the neural network model and how well the models are discriminating between digits using these extracted features. Larger versions of these images and the Python code leveraged to generate them are available in Appendix G.

Experiment 1 – Boxplot of Hidden Node Activation Values By Digit



Experiment 1 – Pattern For Which Hidden Node Maximally Responds

Experiment 2 – Activation Values of the Two Hidden Nodes Disaggregated by Digit

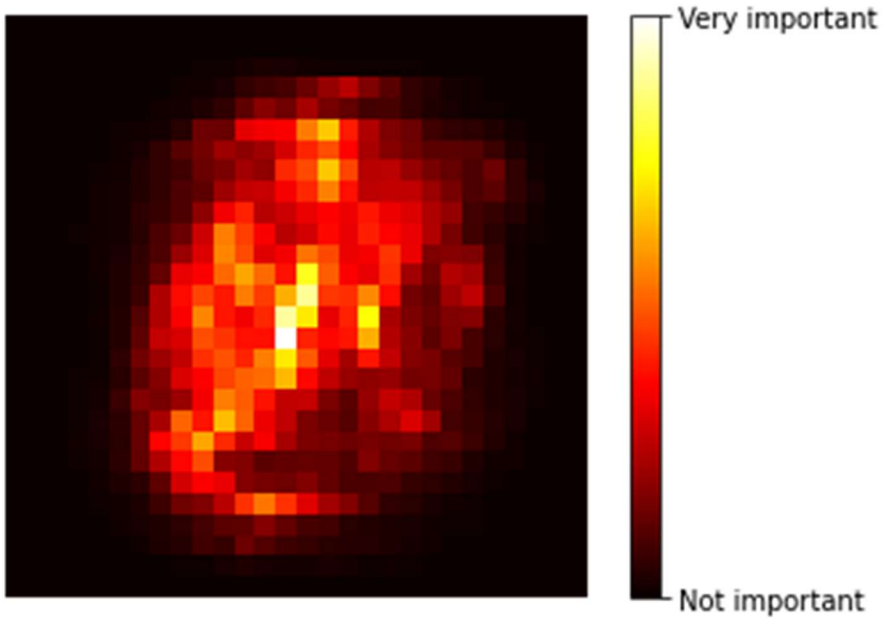


Experiment 3 – T-SNE Plot Visualizing Higher Dimensional Clusters of Activation Values In Two Dimensions Disaggregated By Digit

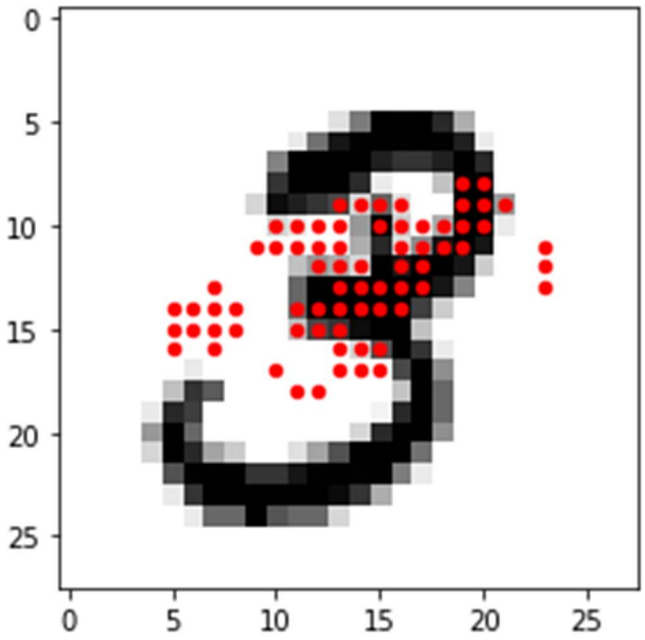
Appendix F – Feature Importance Analyses From Random Forest Classification Model

Below we see two visualizations emphasizing the importance of relative features as determined by the Random Forest Classification Model that served as an input for the tenth neural network classification model. Larger versions of both visualizations and the code leveraged to generate these images are available in Appendix G.

The first image displays a heatmap of the relative importances of each of the MNIST image pixels as determined by the Random Forest Classification Model.



This second image displays the 70 most important pixels identified by the Random Forest Classification Model against a background of one of the MNIST images included in the sample. These 70 pixels were then included in the tenth neural network classification model.



Appendix G - MNIST Digit Classification Project - Python Code

Steve Desilets

October 8, 2023

1) Introduction

For this project, we will be constructing models capable of classifying images of hand-drawn digits as the appropriate number (from 0 throuh 9). To achieve this objective, we will construct several artificial neural network (ANN) models, an ANN model that leverages input data derived from principal components analysis (PCA), and a Random Forest Classifier model. For each of these models, we'll examine performance metrics as well as the key features that the models extracted during the training process.

2) Importing Data, Conducting Exploratory Data Analysis, and Cleaning Data

2.1) Notebook Set-Up and Data Importation

First, let's download the necessary packages.

In [2]:
```
import datetime
from packaging import version
from collections import Counter
import numpy as np
import pandas as pd

import matplotlib as mpl  # EA
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.metrics import confusion_matrix, classification_report
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import mean_squared_error as MSE
from sklearn.metrics import accuracy_score

import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from tensorflow import keras
from tensorflow.keras import models
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
```

```
from tensorflow.keras.datasets import mnist
import tensorflow.keras.backend as k
```

In [3]:
```
%matplotlib inline
np.set_printoptions(precision=3, suppress=True)
```

In [4]:
```
tf.compat.v1.disable_eager_execution() # neccessary for K.gradient to work in TensorFl
```

In [5]:
```
print("This notebook requires TensorFlow 2.0 or above")
print("TensorFlow version: ", tf.__version__)
assert version.parse(tf.__version__).release[0] >=2
```

```
This notebook requires TensorFlow 2.0 or above
TensorFlow version:  2.12.0
```

Now let's mount to the Google Colab environment.

In [ ]:
```
#from google.colab import drive
#drive.mount('/content/gdrive')
```

Let's define functions that will be useful throughout the model development and assessment process.

In [6]:
```
def print_validation_report(test_labels, predictions):
    print("Classification Report")
    print(classification_report(test_labels, predictions))
    print('Accuracy Score: {}'.format(accuracy_score(test_labels, predictions)))
    print('Root Mean Square Error: {}'.format(np.sqrt(MSE(test_labels, predictions))))

def plot_confusion_matrix(y_true, y_pred):
    mtx = confusion_matrix(y_true, y_pred)
    fig, ax = plt.subplots(figsize=(16,12))
    sns.heatmap(mtx, annot=True, fmt='d', linewidths=.75,  cbar=False, ax=ax,cmap='Blu
    #  square=True,
    plt.ylabel('true label')
    plt.xlabel('predicted label')

def plot_history(history):
  losses = history.history['loss']
  accs = history.history['accuracy']
  val_losses = history.history['val_loss']
  val_accs = history.history['val_accuracy']
  epochs = len(losses)

  plt.figure(figsize=(16, 4))
  for i, metrics in enumerate(zip([losses, accs], [val_losses, val_accs], ['Loss', 'Ac
    plt.subplot(1, 2, i + 1)
    plt.plot(range(epochs), metrics[0], label='Training {}'.format(metrics[2]))
    plt.plot(range(epochs), metrics[1], label='Validation {}'.format(metrics[2]))
    plt.legend()
  plt.show()

def plot_digits(instances, pos, images_per_row=5, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
```

```
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    pos.imshow(image, cmap = 'binary', **options)
    pos.axis("off")

def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = 'hot',
               interpolation="nearest")
    plt.axis("off")
```

Let's load the MNIST dataset into this Python notebook.

The MNIST dataset of handwritten digits has a training set of 60,000 images, and a test set of 10,000 images. It comes prepackaged as part of `tf.Keras`. We will load this dataset and the corresponding labels as Numpy arrays.

- Tuples of Numpy arrays: `(x_train, y_train)`, `(x_test, y_test)`
- `x_train`, `x_test`: uint8 arrays of grayscale image data with shapes (num_samples, 28, 28).
- `y_train`, `y_test`: uint8 arrays of digit labels (integers in range 0-9)

In [7]:
```
(x_train, y_train), (x_test, y_test)= tf.keras.datasets.mnist.load_data()
```

2.2) Exploratory Data Analysis

Let's perform exploratory data analysis on the imported MNIST data.

Let's examine the shape of our datasets.

In [8]:
```
print('x_train:\t{}'.format(x_train.shape))
print('y_train:\t{}'.format(y_train.shape))
print('x_test:\t\t{}'.format(x_test.shape))
print('y_test:\t\t{}'.format(y_test.shape))
```

```
x_train:        (60000, 28, 28)
y_train:        (60000,)
x_test:         (10000, 28, 28)
y_test:         (10000,)
```

Let's print the first 10 labels in our training and test datasets.

In [9]:
```
print("First ten labels training dataset:\n {}\n".format(y_train[0:10]))
```

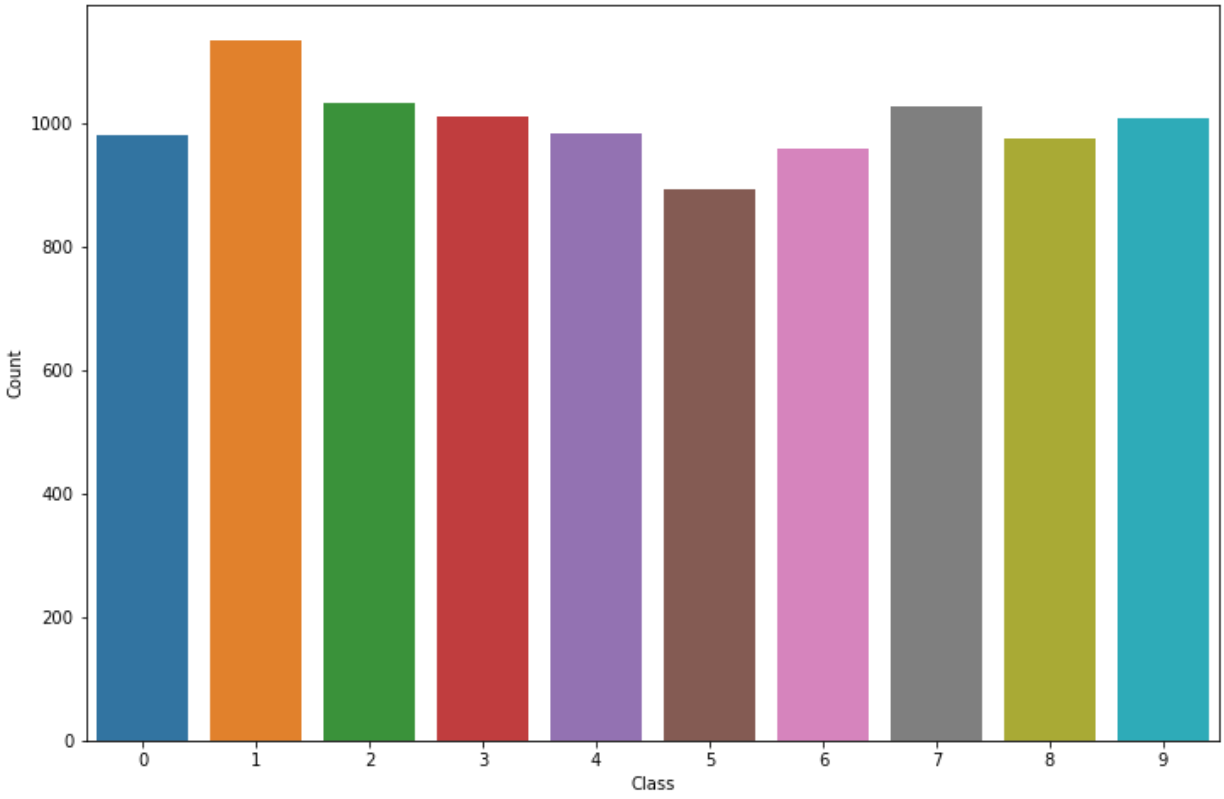
```
First ten labels training dataset:
 [5 0 4 1 9 2 1 3 1 4]
```

In [10]:
```
print("First ten labels training dataset:\n {}\n".format(y_test[0:10]))
```

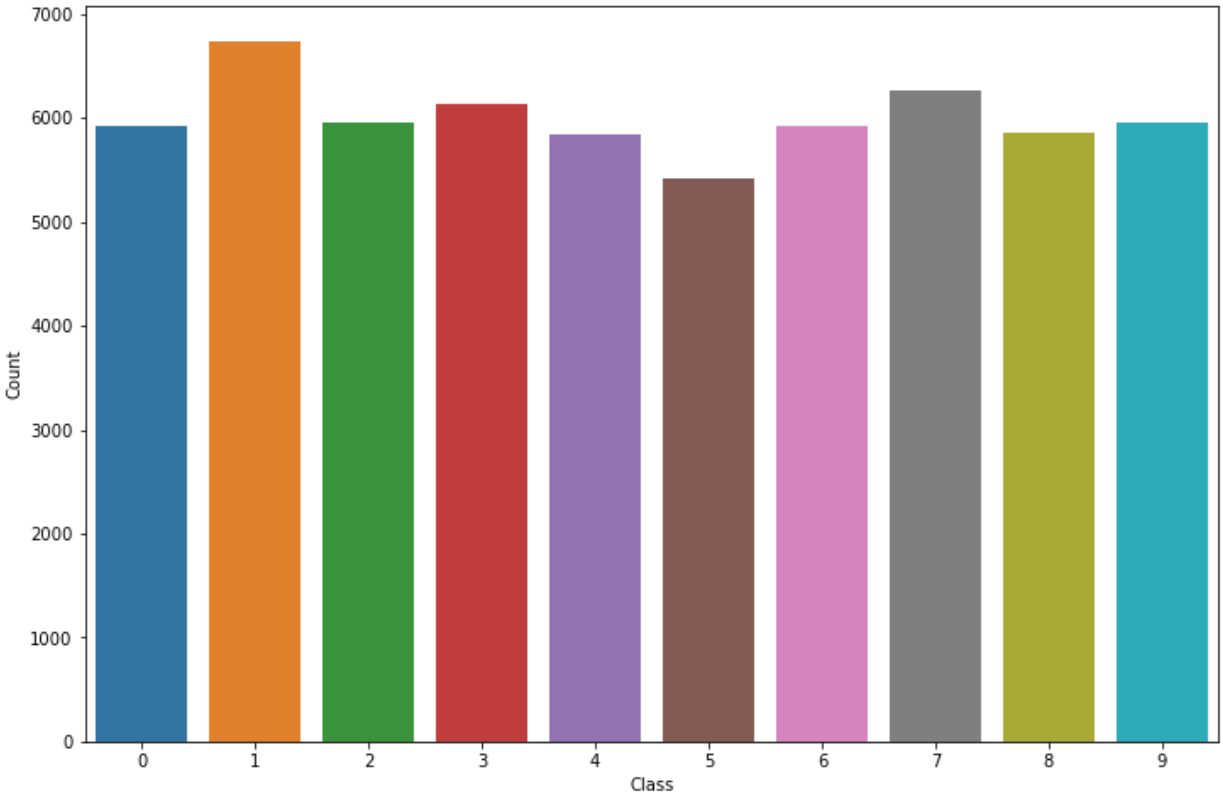
```
First ten labels training dataset:
 [7 2 1 0 4 1 4 9 5 9]
```

Let's examine the distribution of label values in our training and test datasets.

In [11]:
```
plt.figure(figsize = (12 ,8))
items = [{'Class': x, 'Count': y} for x, y in Counter(y_test).items()]
distribution = pd.DataFrame(items).sort_values(['Class'])
sns.barplot(x=distribution.Class, y=distribution.Count);
```



In [12]:
```
plt.figure(figsize = (12 ,8))
items = [{'Class': x, 'Count': y} for x, y in Counter(y_train).items()]
distribution = pd.DataFrame(items).sort_values(['Class'])
sns.barplot(x=distribution.Class, y=distribution.Count);
```



In [13]:
```
Counter(y_train).most_common()
```

Out[13]:
```
[(1, 6742),
 (7, 6265),
 (3, 6131),
 (2, 5958),
 (9, 5949),
 (0, 5923),
 (6, 5918),
 (8, 5851),
 (4, 5842),
 (5, 5421)]
```

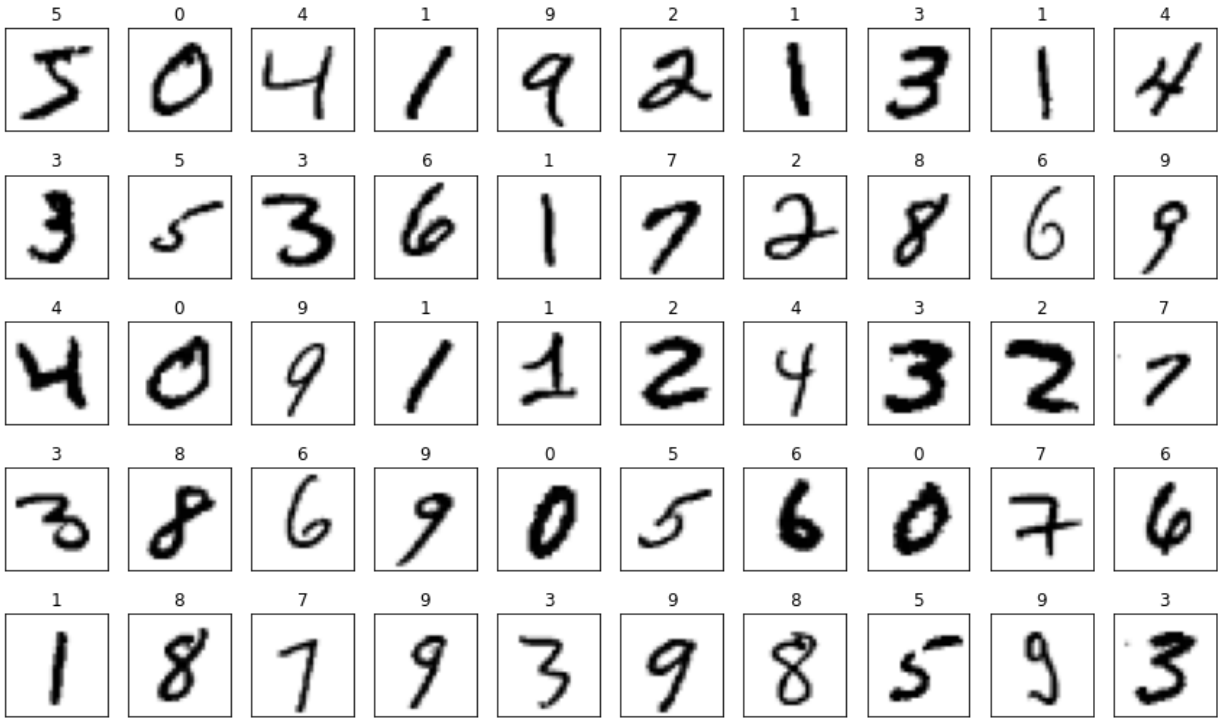
In [14]:
```
Counter(y_test).most_common()
```

Out[14]:
```
[(1, 1135),
 (2, 1032),
 (7, 1028),
 (3, 1010),
 (9, 1009),
 (4, 982),
 (0, 980),
 (8, 974),
 (6, 958),
 (5, 892)]
```

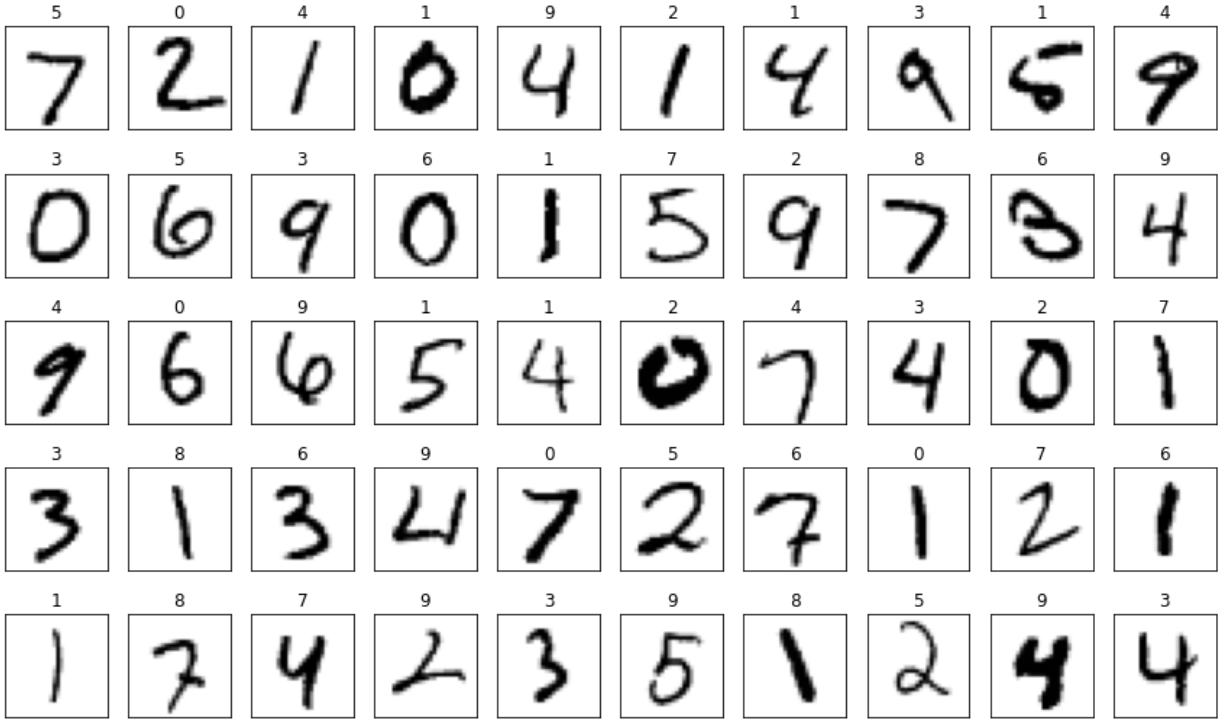
Let's examine the first fifty images in the training and test datasets.

In [15]:
```
fig = plt.figure(figsize = (15, 9))

for i in range(50):
    plt.subplot(5, 10, 1+i)
    plt.title(y_train[i])
    plt.xticks([])
```

```
    plt.yticks([])
    plt.imshow(x_train[i].reshape(28,28), cmap='binary')
```



In [16]:
```
fig = plt.figure(figsize = (15, 9))

for i in range(50):
    plt.subplot(5, 10, 1+i)
    plt.title(y_train[i])
    plt.xticks([])
    plt.yticks([])
    plt.imshow(x_test[i].reshape(28,28), cmap='binary')
```



2.3) Data Cleaning and Pre-Processing

- Before we build our model, we need to prepare the data into the shape the network expected
- More specifically, we will convert the labels (integers 0 to 9) to 1D numpy arrays of shape (10,) with elements 0s and 1s.
- We also reshape the images from 2D arrays of shape (28,28) to 1D *float32* arrays of shape (784,) and then rescale their elements to values between 0 and 1.

Let's apply one-hot coding to the labels.

We will change the way the labels are represented from numbers (0 to 9) to vectors (1D arrays) of shape (10,) with all the elements set to 0 except the one which the label belongs to - which will be set to 1. For example:

original label	one-hot encoded label
5	[0 0 0 0 0 1 0 0 0 0]
7	[0 0 0 0 0 0 0 1 0 0]
1	[0 1 0 0 0 0 0 0 0 0]

In [17]:
```
y_train_encoded = to_categorical(y_train)
y_test_encoded = to_categorical(y_test)

print("First ten entries of y_train:\n {}\n".format(y_train[0:10]))
print("First ten rows of one-hot y_train:\n {}".format(y_train_encoded[0:10,]))

print("First ten entries of y_test:\n {}\n".format(y_train[0:10]))
print("First ten rows of one-hot y_test:\n {}".format(y_train_encoded[0:10,]))
```

```
First ten entries of y_train:
 [5 0 4 1 9 2 1 3 1 4]

First ten rows of one-hot y_train:
 [[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
First ten entries of y_test:
 [5 0 4 1 9 2 1 3 1 4]

First ten rows of one-hot y_test:
 [[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

In [18]:
```
print('y_train_encoded shape: ', y_train_encoded.shape)
print('y_test_encoded shape: ', y_test_encoded.shape)
```

```
y_train_encoded shape:  (60000, 10)
y_test_encoded shape:  (10000, 10)
```

Reshape the images to 1D arrays

Reshape the images from shape (28, 28) 2D arrays to shape (784,) vectors (1D arrays).

In [19]:
```
# Before reshape:
print('x_train:\t{}'.format(x_train.shape))
print('x_test:\t\t{}'.format(x_test.shape))
```

```
x_train:        (60000, 28, 28)
x_test:         (10000, 28, 28)
```

In [20]:
```
np.set_printoptions(linewidth=np.inf)
print("{}".format(x_train[2020]))
```

```
[[  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   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   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 167 208  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  13 235 254  99   0   0   0   0   0
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  74 254 234   4   0   0   0   0   0
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 154 254 145   0   0   0   0   0   0
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 224 254  92   0   0   0   0   0   0
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  51 245 211  13   0   0   0   0   0   0
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   2 169 254 101   0   0   0   0   0   0   0
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0  27 254 254  88   0   0   0   0   0   0   0
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0  72 255 241  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  88 254 153   0   0  33  53 155 156 102  15
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0 130 254  31   0 128 235 254 254 254 254 186
10   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0 190 254  51 178 254 246 213 111 109 186 254
145   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0 192 254 229 254 216  90   0   0   0  57 254
234   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0 235 254 254 247  85   0   0   0   0  32 254
234   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0 235 254 254 118   0   0   0   0   0 107 254
201   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0 235 255 254 102  12   0   0   0   8 188 248
119   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0 207 254 254 238 107   0   0  39 175 254 148
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0  84 254 248  74  11  32 115 238 254 176  11
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0  21 214 254 254 254 254 254 254 132   6   0
0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  14  96 176 254 254 214  48  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   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   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   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   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   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]
 [  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   0   0   0]]
```

In [21]:
```
# Reshape the images:
x_train_reshaped = np.reshape(x_train, (60000, 784))
x_test_reshaped = np.reshape(x_test, (10000, 784))
```

```
# After reshape:
print('x_train_reshaped shape: ', x_train_reshaped.shape)
print('x_test_reshaped shape: ', x_test_reshaped.shape)
```

```
x_train_reshaped shape:  (60000, 784)
x_test_reshaped shape:  (10000, 784)
```

1. Each element in an image is a pixel value
2. Pixel values range from 0 to 255
3. 0 = White
4. 255 = Black

Let's review thew unique values using the set from the first image in the training dataset.

In [22]:
```
print(set(x_train_reshaped[0]))
```

```
{0, 1, 2, 3, 9, 11, 14, 16, 18, 23, 24, 25, 26, 27, 30, 35, 36, 39, 43, 45, 46, 49, 5
5, 56, 64, 66, 70, 78, 80, 81, 82, 90, 93, 94, 107, 108, 114, 119, 126, 127, 130, 13
2, 133, 135, 136, 139, 148, 150, 154, 156, 160, 166, 170, 171, 172, 175, 182, 183, 18
6, 187, 190, 195, 198, 201, 205, 207, 212, 213, 219, 221, 225, 226, 229, 238, 240, 24
1, 242, 244, 247, 249, 250, 251, 252, 253, 255}
```

Rescale the elements of the reshaped images

Let's rescale the elements of the x_train_reshaped and x_test_reshaped datasets to be between 0 and 1.

In [23]:
```
x_train_norm = x_train_reshaped.astype('float32') / 255
x_test_norm = x_test_reshaped.astype('float32') / 255
```

In [24]:
```
# Take a look at the first reshaped and normalized training image:
print(set(x_train_norm[0]))
print(set(x_test_norm[0]))
```

```
{0.0, 0.011764706, 0.53333336, 0.07058824, 0.49411765, 0.6862745, 0.101960786, 0.6509
804, 1.0, 0.96862745, 0.49803922, 0.11764706, 0.14117648, 0.36862746, 0.6039216, 0.66
66667, 0.043137256, 0.05490196, 0.03529412, 0.85882354, 0.7764706, 0.7137255, 0.94509
804, 0.3137255, 0.6117647, 0.41960785, 0.25882354, 0.32156864, 0.21960784, 0.8039216,
0.8666667, 0.8980392, 0.7882353, 0.52156866, 0.18039216, 0.30588236, 0.44705883, 0.35
29412, 0.15294118, 0.6745098, 0.88235295, 0.99215686, 0.9490196, 0.7647059, 0.250980
4, 0.19215687, 0.93333334, 0.9843137, 0.74509805, 0.7294118, 0.5882353, 0.50980395,
0.8862745, 0.105882354, 0.09019608, 0.16862746, 0.13725491, 0.21568628, 0.46666667,
0.3647059, 0.27450982, 0.8352941, 0.7176471, 0.5803922, 0.8117647, 0.9764706, 0.98039
216, 0.73333335, 0.42352942, 0.003921569, 0.54509807, 0.67058825, 0.5294118, 0.007843
138, 0.31764707, 0.0627451, 0.09411765, 0.627451, 0.9411765, 0.9882353, 0.95686275,
0.83137256, 0.5176471, 0.09803922, 0.1764706}
{0.0, 0.32941177, 0.5921569, 0.7254902, 0.62352943, 0.23529412, 0.14117648, 0.8705882
4, 0.99607843, 0.94509804, 0.7764706, 0.6666667, 0.20392157, 0.2627451, 0.44705883,
0.28235295, 0.3254902, 0.99215686, 0.81960785, 0.07058824, 0.08627451, 0.9137255, 1.
0, 0.9411765, 0.9490196, 0.08235294, 0.9254902, 0.41568628, 0.88235295, 0.8901961, 0.
6392157, 0.98039216, 0.8980392, 0.54901963, 0.06666667, 0.25882354, 0.05490196, 0.231
37255, 0.003921569, 0.8117647, 0.29411766, 0.9843137, 0.85882354, 0.3019608, 0.505882
4, 0.93333334, 0.9764706, 0.52156866, 0.73333335, 0.019607844, 0.8039216, 0.972549,
0.03529412, 0.7137255, 0.8666667, 0.011764706, 0.8784314, 0.07450981, 0.12156863, 0.4
509804, 0.17254902, 0.24313726, 0.22745098, 0.22352941, 0.13725491, 0.6509804, 0.1490
1961, 0.49411765, 0.23921569, 0.15686275, 0.4745098, 0.79607844}
```

3) Experiment 1 - ANN with 1 Hidden Layer with 1 Node

3.1) Construct Model

Now that we've finished preprocessing our MNIST image data, let's construct our first artificial neural network to classify images as integers.

In [25]:
```
k.clear_session()

model = Sequential([
    Dense(input_shape=[784], units=1, activation = tf.nn.relu,kernel_regularizer=tf.ke
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])
```

In [26]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 1)                 785

 output_layer (Dense)        (None, 10)                20

=================================================================
Total params: 805
Trainable params: 805
Non-trainable params: 0
_________________________________________________________________
```

In [27]:
```
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
```

```
                metrics=['accuracy'])
```

In [28]:
```
#tf.keras.model.fit
#https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

#tf.keras.callbacks.EarlyStopping
#https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping


history = model.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=(5000/60000)
    ,callbacks=[tf.keras.callbacks.ModelCheckpoint("DNN_model.h5",save_best_only=True,
                ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/200
54016/55000 [============================>.] - ETA: 0s - loss: 1.9729 - accuracy: 0.1
990
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2335: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates
```

```
55000/55000 [==============================] - 2s 41us/sample - loss: 1.9710 - accura
cy: 0.1995 - val_loss: 1.8240 - val_accuracy: 0.2272
Epoch 2/200
55000/55000 [==============================] - 2s 34us/sample - loss: 1.8226 - accura
cy: 0.2230 - val_loss: 1.7729 - val_accuracy: 0.2440
Epoch 3/200
55000/55000 [==============================] - 2s 45us/sample - loss: 1.7863 - accura
cy: 0.2505 - val_loss: 1.7355 - val_accuracy: 0.2830
Epoch 4/200
55000/55000 [==============================] - 2s 40us/sample - loss: 1.7419 - accura
cy: 0.2867 - val_loss: 1.6806 - val_accuracy: 0.3164
Epoch 5/200
55000/55000 [==============================] - 2s 35us/sample - loss: 1.6946 - accura
cy: 0.3025 - val_loss: 1.6368 - val_accuracy: 0.3168
Epoch 6/200
55000/55000 [==============================] - 2s 37us/sample - loss: 1.6681 - accura
cy: 0.3080 - val_loss: 1.6177 - val_accuracy: 0.3200
Epoch 7/200
55000/55000 [==============================] - 2s 33us/sample - loss: 1.6537 - accura
cy: 0.3130 - val_loss: 1.6058 - val_accuracy: 0.3438
Epoch 8/200
55000/55000 [==============================] - 2s 41us/sample - loss: 1.6435 - accura
cy: 0.3318 - val_loss: 1.5939 - val_accuracy: 0.3678
Epoch 9/200
55000/55000 [==============================] - 2s 33us/sample - loss: 1.6361 - accura
cy: 0.3459 - val_loss: 1.5829 - val_accuracy: 0.3668
Epoch 10/200
55000/55000 [==============================] - 2s 38us/sample - loss: 1.6297 - accura
cy: 0.3585 - val_loss: 1.5753 - val_accuracy: 0.3922
Epoch 11/200
55000/55000 [==============================] - 1s 25us/sample - loss: 1.6200 - accura
cy: 0.3869 - val_loss: 1.5669 - val_accuracy: 0.4052
Epoch 12/200
55000/55000 [==============================] - 2s 31us/sample - loss: 1.6039 - accura
cy: 0.3969 - val_loss: 1.5450 - val_accuracy: 0.4152
Epoch 13/200
55000/55000 [==============================] - 1s 24us/sample - loss: 1.5916 - accura
cy: 0.3946 - val_loss: 1.5364 - val_accuracy: 0.4112
Epoch 14/200
55000/55000 [==============================] - 1s 24us/sample - loss: 1.5842 - accura
cy: 0.3941 - val_loss: 1.5325 - val_accuracy: 0.4040
```

3.2) Evaluate Model Performance on Testing Dataset

Now that we've fit our model, let's apply the model to the test dataset and subsequently evaluate its performance.

In [29]:
```
model = tf.keras.models.load_model("DNN_model.h5")
print(f"Test acc: {model.evaluate(x_test_norm, y_test_encoded)[1]:.3f}")
```

```
Test acc: 0.383
```

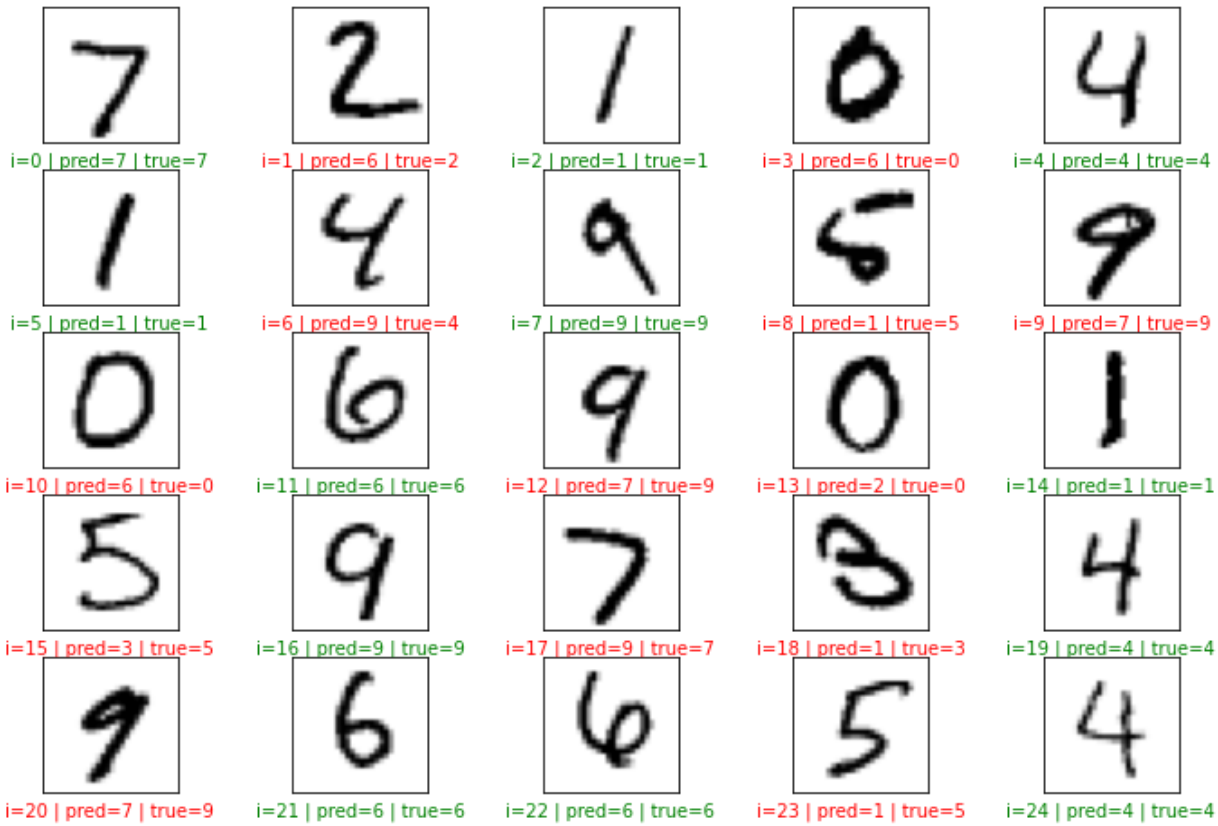
In [30]:
```
# loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
# print('test set accuracy: ', accuracy * 100)
```

In [31]:
```
preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
shape of preds:  (10000, 10)
```

As part of our model evaluation, let's look at the first 25 images by plotting the test set images along with their predicted and actual labels to understand how the trained model actually performed on specific example images.

In [32]:
```
plt.figure(figsize = (12, 8))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```



Let's use `Matplotlib` to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

In [33]:
```
history_dict = history.history
history_dict.keys()
```

Out[33]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

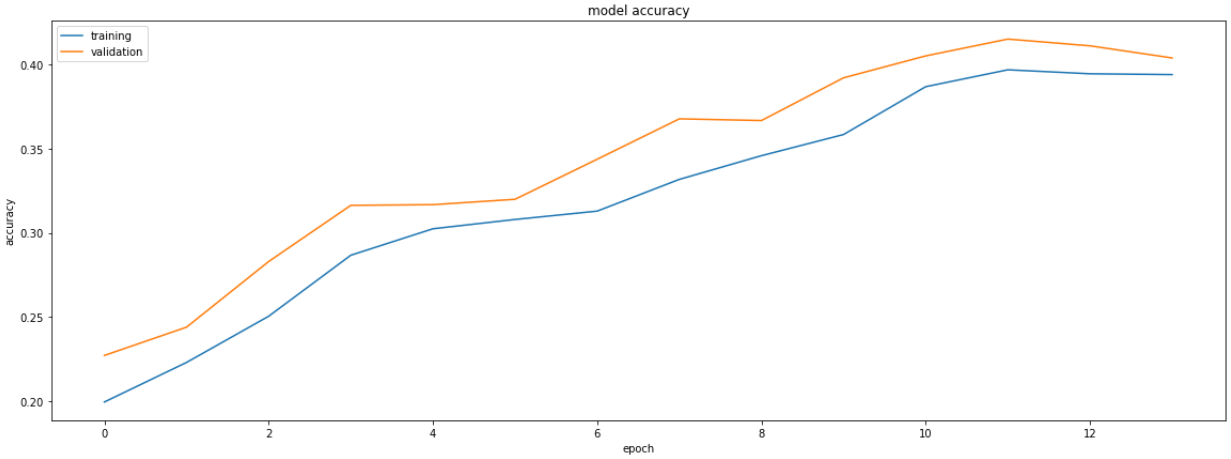
In [34]:
```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)
```

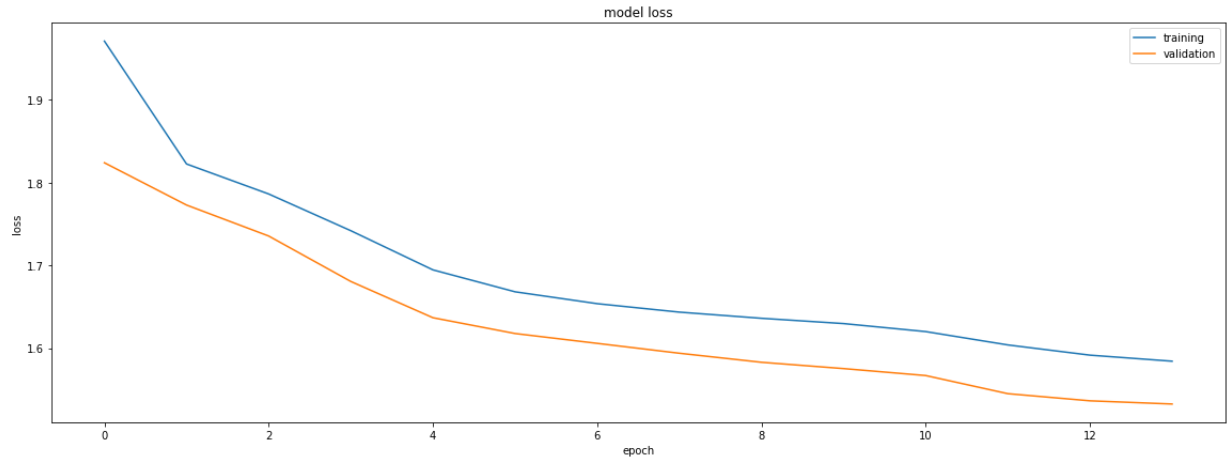
In [35]:
```
history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)
```

Out[35]:

	loss	accuracy	val_loss	val_accuracy
9	1.630	0.358	1.575	0.392
10	1.620	0.387	1.567	0.405
11	1.604	0.397	1.545	0.415
12	1.592	0.395	1.536	0.411
13	1.584	0.394	1.532	0.404

In [36]:
```
def display_training_curves(training, validation, title, subplot):
  ax = plt.subplot(subplot)
  ax.plot(training)
  ax.plot(validation)
  ax.set_title('model '+ title)
  ax.set_ylabel(title)
  ax.set_xlabel('epoch')
  ax.legend(['training', 'validation'])

plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```





Let's examine precision and recall performance metrics for each of the prediction classes.

In [37]:
```
pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)
```

In [38]:
```
print_validation_report(y_test, pred1)
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.18      0.02      0.04       980
           1       0.40      0.86      0.54      1135
           2       0.24      0.07      0.11      1032
           3       0.27      0.35      0.30      1010
           4       0.44      0.43      0.44       982
           5       0.00      0.00      0.00       892
           6       0.33      0.90      0.49       958
           7       0.62      0.61      0.62      1028
           8       0.26      0.04      0.06       974
           9       0.40      0.45      0.43      1009

    accuracy                           0.38     10000
   macro avg       0.31      0.37      0.30     10000
weighted avg       0.32      0.38      0.31     10000

Accuracy Score: 0.3831
Root Mean Square Error: 3.3357907608241857
```

```
C:\Users\steve\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1469: U
ndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in
labels with no predicted samples. Use `zero_division` parameter to control this behav
ior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\steve\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1469: U
ndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in
labels with no predicted samples. Use `zero_division` parameter to control this behav
ior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\steve\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1469: U
ndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in
labels with no predicted samples. Use `zero_division` parameter to control this behav
ior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Let's create a table that visualizes the model output for each of the first 20 images. These outputs can be thought of as the model's expression of the probability that each image corresponds to each digit class.

In [39]:
```
# Get the predicted classes:
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning
pred_classes = np.argmax(model.predict(x_test_norm), axis=-1)
pred_classes;
```

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

In [40]:
```
conf_mx = tf.math.confusion_matrix(y_test, pred_classes)
conf_mx;
```

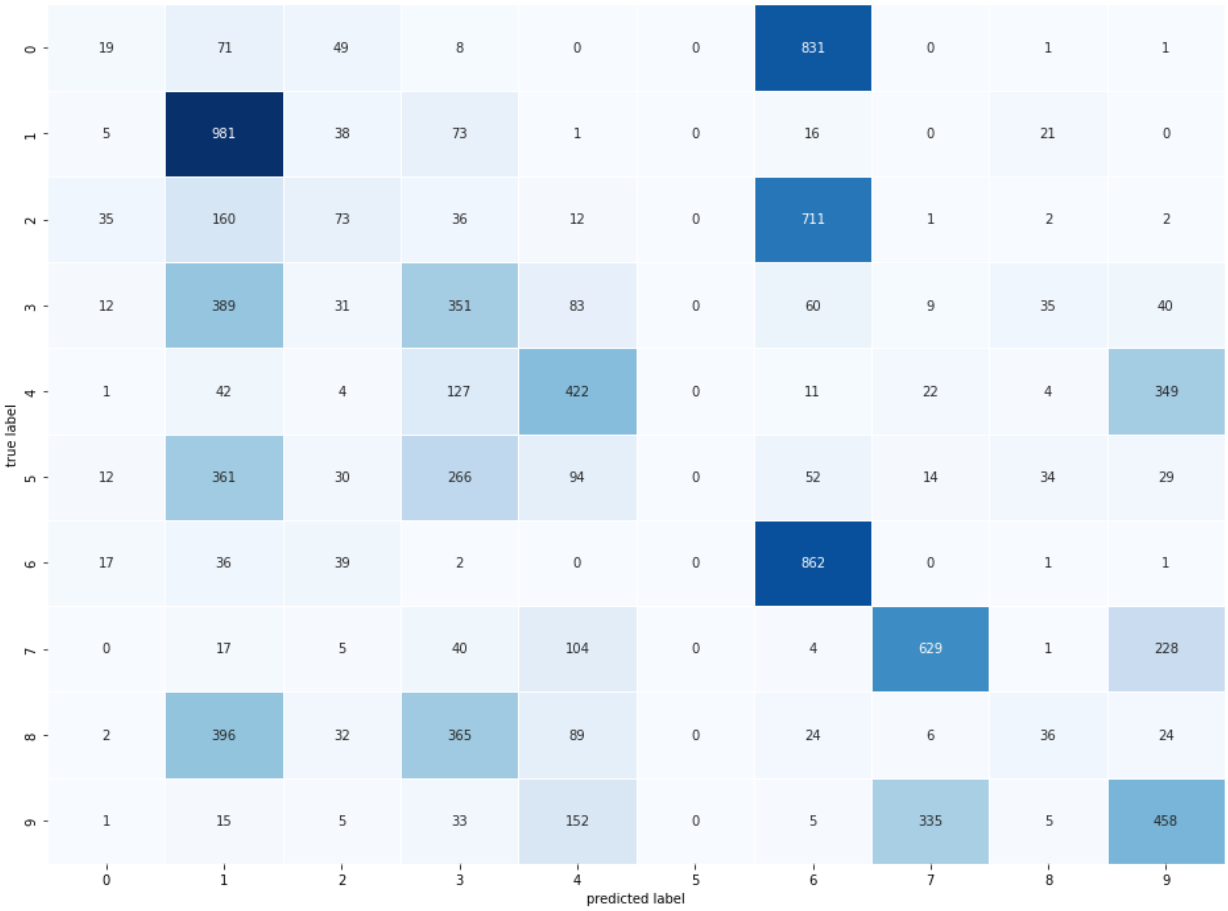
In [41]:
```
cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8',
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

Out[41]:

	0	1	2	3	4	5	6	7	8	9
0	0.00%	0.00%	0.00%	0.18%	6.95%	0.12%	0.00%	60.09%	0.12%	32.55%
1	27.85%	9.33%	21.94%	1.68%	0.19%	1.65%	35.37%	0.01%	1.95%	0.03%
2	6.39%	27.22%	13.02%	15.44%	4.45%	14.17%	1.40%	0.35%	16.27%	1.30%
3	27.85%	9.33%	21.94%	1.68%	0.19%	1.65%	35.37%	0.01%	1.95%	0.03%
4	0.04%	6.24%	0.30%	18.70%	21.64%	15.56%	0.00%	6.46%	17.15%	13.92%
5	4.17%	25.90%	9.79%	17.41%	5.78%	15.81%	0.70%	0.52%	18.08%	1.83%
6	0.00%	0.33%	0.00%	5.25%	24.35%	3.96%	0.00%	27.55%	4.19%	34.37%
7	0.00%	1.01%	0.01%	9.14%	26.70%	7.13%	0.00%	19.38%	7.64%	29.00%
8	15.21%	25.42%	21.85%	9.46%	1.92%	8.90%	6.36%	0.11%	10.32%	0.46%
9	0.00%	0.00%	0.00%	0.27%	8.24%	0.18%	0.00%	57.22%	0.18%	33.91%
10	27.85%	9.33%	21.94%	1.68%	0.19%	1.65%	35.37%	0.01%	1.95%	0.03%
11	27.85%	9.33%	21.94%	1.68%	0.19%	1.65%	35.37%	0.01%	1.95%	0.03%
12	0.00%	0.03%	0.00%	1.38%	15.92%	0.98%	0.00%	43.10%	1.01%	37.59%
13	24.86%	16.73%	25.42%	4.13%	0.59%	3.98%	19.47%	0.02%	4.67%	0.12%
14	7.18%	27.42%	14.04%	14.81%	4.10%	13.63%	1.69%	0.31%	15.66%	1.17%
15	0.26%	12.57%	1.31%	21.30%	15.32%	18.33%	0.01%	2.90%	20.48%	7.53%
16	0.00%	0.77%	0.01%	8.04%	26.37%	6.23%	0.00%	21.37%	6.65%	30.57%
17	0.00%	0.08%	0.00%	2.40%	19.35%	1.74%	0.00%	37.25%	1.81%	37.37%
18	4.95%	26.52%	10.99%	16.68%	5.24%	15.21%	0.92%	0.45%	17.42%	1.61%
19	0.01%	4.19%	0.13%	16.54%	24.11%	13.55%	0.00%	8.98%	14.82%	17.67%

Let's create a confusion matrix that visualizes the model's performance on the testing data.

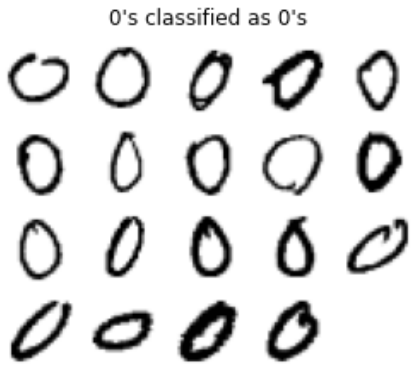
In [42]:
```
plot_confusion_matrix(y_test,pred_classes)
```

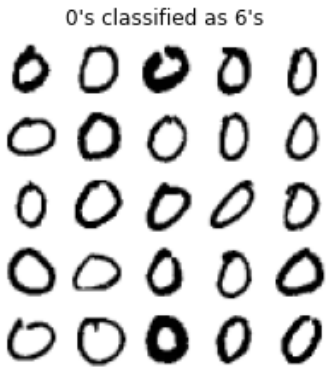


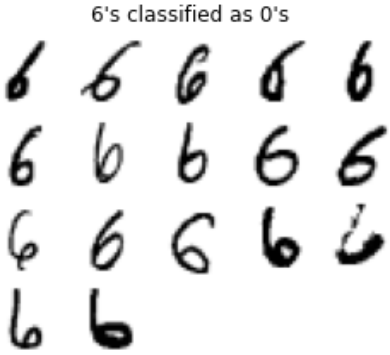
It looks like 831 zeroes were misclassified as sixes (and 17 sixes were misclassified as zeroes). We display some of these misclassfications along with an examination of zeroes and sixes that were correctly identified.

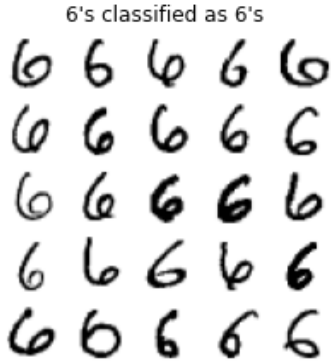
In [43]:
```
cl_a, cl_b = 0, 6
X_aa = x_test_norm[(y_test == cl_a) & (pred_classes == cl_a)]
X_ab = x_test_norm[(y_test == cl_a) & (pred_classes == cl_b)]
X_ba = x_test_norm[(y_test == cl_b) & (pred_classes == cl_a)]
X_bb = x_test_norm[(y_test == cl_b) & (pred_classes == cl_b)]

plt.figure(figsize=(16,8))

p1 = plt.subplot(221)
p2 = plt.subplot(222)
p3 = plt.subplot(223)
p4 = plt.subplot(224)

plot_digits(X_aa[:25], p1, images_per_row=5);
plot_digits(X_ab[:25], p2, images_per_row=5);
plot_digits(X_ba[:25], p3, images_per_row=5);
plot_digits(X_bb[:25], p4, images_per_row=5);


p1.set_title(f"{cl_a}'s classified as {cl_a}'s")
p2.set_title(f"{cl_a}'s classified as {cl_b}'s")
p3.set_title(f"{cl_b}'s classified as {cl_a}'s")
p4.set_title(f"{cl_b}'s classified as {cl_b}'s")

# plt.savefig("error_analysis_digits_plot_EXP1_valid")
```

```
plt.show()
```









3.3) Examine Model Activation Values

To get the activation values of the hidden nodes, we need to create a new model, `activation_model`, that takes the same input as our current model but outputs the activation value of the hidden layer, i.e. of the hidden node. Then use the `predict` function to get the activation values.

In [44]:
```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs; # description of the layers
```

```
There are 2 layers
```

In [45]:
```
# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation val
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
```

Out[45]:
```
(60000, 1)
```

In [46]:
```
output_layer_activations.shape
```

Out[46]:
```
(60000, 10)
```

In [47]:
```
print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")
```

```
The maximum activation value of the hidden nodes in the hidden layer is 10.4958934783
93555
```

In [48]:
```
# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in s
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum
```

```
The output node has shape (60000, 10)
The output for the first image are [0.104 0.273 0.177 0.125 0.03  0.116 0.032 0.002
0.134 0.008]
The sum of the probabilities is (approximately) 0.9999999403953552
```

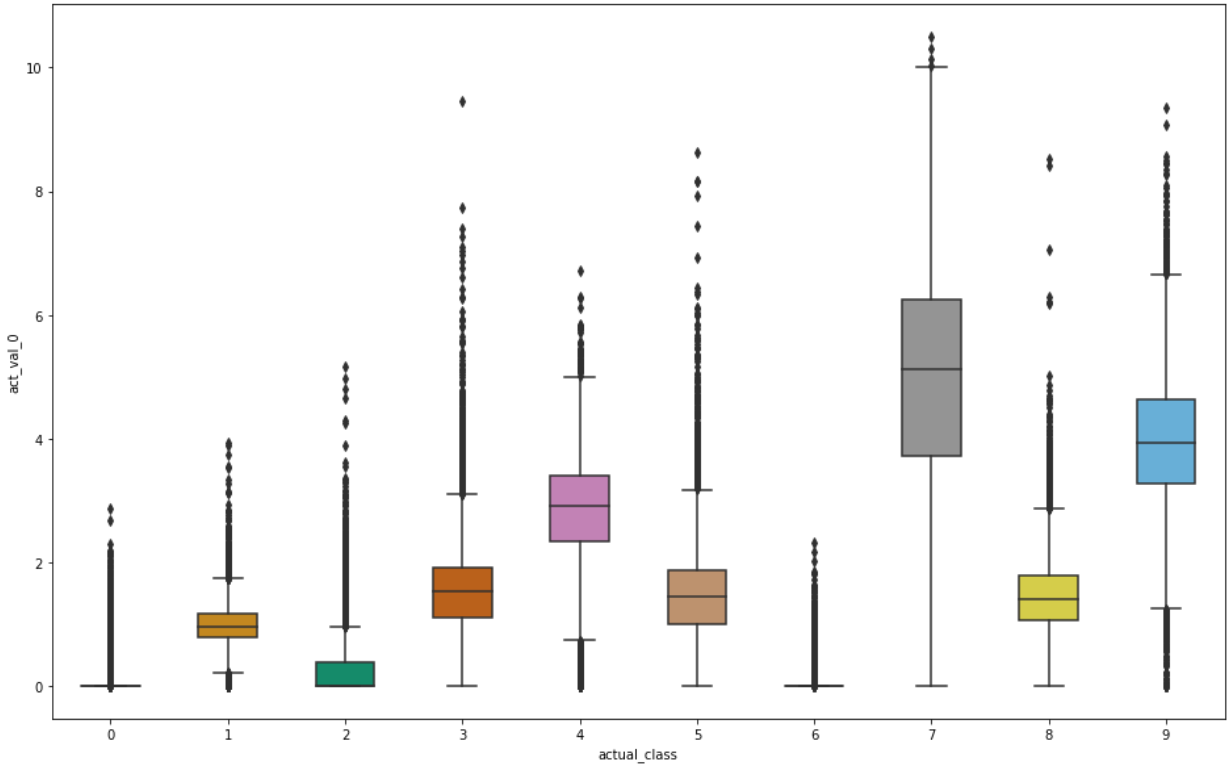
In [49]:
```
#Get the dataframe of all the node values
activation_data = {'actual_class':y_train}
for k in range(0,1):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head(15).round(3).T
```

Out[49]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
actual_class	5.00	0.0	4.000	1.000	9.000	2.000	1.00	3.000	1.00	4.000	3.000	5.000	3.000	6.0	1.0
act_val_0	0.79	0.0	3.088	0.868	4.732	0.914	1.13	1.358	0.95	2.957	2.012	1.035	1.207	0.0	0.7

Let's visualize these activiation values via boxplots.

In [50]:
```
# To see how closely the hidden node activation values correlate with the class labels
# Let us use seaborn for the boxplots this time.
plt.figure(figsize=(16,10))
bplot = sns.boxplot(y='act_val_0', x='actual_class',
                 data=activation_df[['act_val_0','actual_class']],
                 width=0.5,
                 palette="colorblind")
```



In [51]:
```
activation_df.groupby("actual_class")["act_val_0"].apply(lambda x: [round(min(x.tolist
 round(max(x.tolist()),2)]).reset_index().rename(columns={"act_val_0": "range_of_act_v
```

Out[51]:

	actual_class	range_of_act_values
0	0	[0.0, 2.88]
1	1	[0.0, 3.94]
2	2	[0.0, 5.16]
3	3	[0.0, 9.45]
4	4	[0.0, 6.72]
5	5	[0.0, 8.64]
6	6	[0.0, 2.33]
7	7	[0.0, 10.5]
8	8	[0.0, 8.52]
9	9	[0.0, 9.36]

Let's get activation values of the pixel values.

We can create a dataframe with the pixel values and class labels

In [52]:
```
#Get the dataframe of all the pixel values
pixel_data = {'actual_class':y_train}
for k in range(0,784):
    pixel_data[f"pix_val_{k}"] = x_train_norm[:,k]
pixel_df = pd.DataFrame(pixel_data)
pixel_df.head(15).round(3).T
```

Out[52]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
actual_class	5.0	0.0	4.0	1.0	9.0	2.0	1.0	3.0	1.0	4.0	3.0	5.0	3.0	6.0	1.0
pix_val_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	0.0	0.0	0.0
pix_val_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	0.0	0.0	0.0
pix_val_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	0.0	0.0	0.0
pix_val_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	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
pix_val_779	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pix_val_780	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pix_val_781	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pix_val_782	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pix_val_783	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

785 rows × 15 columns

In [53]:
```
pixel_df.pix_val_77.value_counts()
```

Out[53]:
```
0.000000    59720
1.000000       25
0.996078       13
0.992157        9
0.050980        6
            ...
0.894118        1
0.690196        1
0.725490        1
0.517647        1
0.819608        1
Name: pix_val_77, Length: 150, dtype: int64
```

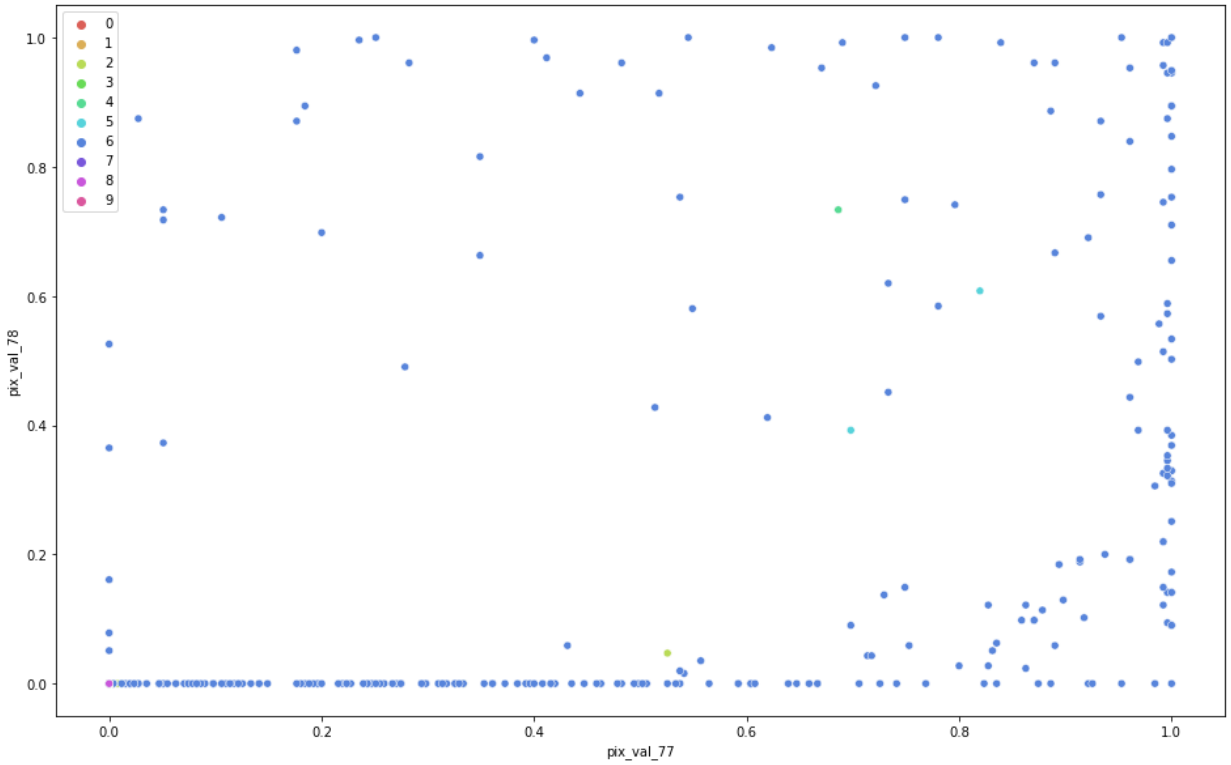
In [54]:
```
pixel_df.pix_val_78.value_counts()
```

Out[54]:
```
0.000000    59862
1.000000        6
0.960784        4
0.992157        4
0.141176        4
            ...
0.556863        1
0.584314        1
0.427451        1
0.078431        1
0.501961        1
Name: pix_val_78, Length: 97, dtype: int64
```

Use a scatter plot to visualize the predicive power of the pixel values at two fixed locations in the image, i.e. how well the pixel values at two fixed locations in the image "predict" the class labels.

We use a scatter plot to determine the correlation between the `pix_val_77` and `pix_val_78` values and the `actual_class` values.

In [55]:
```
plt.figure(figsize=(16, 10))
color = sns.color_palette("hls", 10)
sns.scatterplot(x="pix_val_77", y="pix_val_78", hue="actual_class",  palette=color, da
plt.legend(loc='upper left');
```



Let's find the pattern to which the hidden node maximally responds.

In [56]:
```
def deprocess_image(x):
    # normalize tensor: center on 0., ensure std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    # clip to [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)

    # convert to RGB array
    x *= 255
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

In [57]:
```
def generate_pattern(layer_name, size=28):
    # Build a loss function that maximizes the activation
    # of the nth filter of the layer considered.
    K = tf.keras.backend
    layer_output = model.get_layer(layer_name).output
    loss = K.mean(layer_output)

    # Compute the gradient of the input picture wrt this loss
```

```
    grads = K.gradients(loss, model.input)[0]

    # Normalization trick: we normalize the gradient
    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)

    # This function returns the loss and grads given the input picture
    iterate = K.function([model.input], [loss, grads])

    # We start from a gray image with some noise
    input_img_data = np.random.random((1, size*size)) * 20 + 128.

    # Run gradient ascent for 1000 steps
    step = 1.
    for i in range(1000):
        loss_value, grads_value = iterate([input_img_data])
        input_img_data += grads_value * step


    img = input_img_data[0]
    return deprocess_image(img)
```

In [58]:
```
# Find and display the pattern that maximizes the activation value
max_img = generate_pattern('dense')
plt.imshow(max_img.reshape(28,28), cmap="binary")
plt.axis('off')
plt.show()
```

In [59]:
```
 # print activation value for max_img
max_img_act_value = activation_model.predict(max_img.reshape(-1,784))[0].item(0)
max_img_class = activation_model.predict(max_img.reshape(-1,784))[1].argmax()
print(f"The activation value for max_img is {max_img_act_value}.")
print(f"The model thinks this is an image of a {max_img_class}.")   # check this!
```

```
The activation value for max_img is 3831.807861328125.
The model thinks this is an image of a 7.
```

4) Experiment 2 - ANN with 1 Hidden Layer with 2 Nodes

4.1) Construct Model

Let's construct our an artificial neural network to classify images as integers.

In [62]:
```
tf.keras.backend.clear_session()

model = Sequential([
    Dense(input_shape=[784], units=2, activation = tf.nn.relu,kernel_regularizer=tf.ke
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])
```

In [63]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 2)                 1570

 output_layer (Dense)        (None, 10)                30

=================================================================
Total params: 1,600
Trainable params: 1,600
Non-trainable params: 0
_________________________________________________________________
```

In [64]:
```
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])
```

In [65]:
```
#tf.keras.model.fit
#https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

#tf.keras.callbacks.EarlyStopping
#https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping


history = model.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=(5000/60000)
    ,callbacks=[tf.keras.callbacks.ModelCheckpoint("DNN_model.h5",save_best_only=True,
                ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/200
55000/55000 [==============================] - ETA: 0s - loss: 1.6312 - accuracy: 0.3
911
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2335: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates
```

```
55000/55000 [==============================] - 5s 85us/sample - loss: 1.6312 - accura
cy: 0.3911 - val_loss: 1.3449 - val_accuracy: 0.5388
Epoch 2/200
55000/55000 [==============================] - 2s 40us/sample - loss: 1.2989 - accura
cy: 0.5446 - val_loss: 1.2080 - val_accuracy: 0.5808
Epoch 3/200
55000/55000 [==============================] - 3s 48us/sample - loss: 1.2184 - accura
cy: 0.5785 - val_loss: 1.1533 - val_accuracy: 0.6138
Epoch 4/200
55000/55000 [==============================] - 2s 38us/sample - loss: 1.1761 - accura
cy: 0.5980 - val_loss: 1.1213 - val_accuracy: 0.6288
Epoch 5/200
55000/55000 [==============================] - 1s 26us/sample - loss: 1.1468 - accura
cy: 0.6109 - val_loss: 1.0969 - val_accuracy: 0.6266
Epoch 6/200
55000/55000 [==============================] - 2s 33us/sample - loss: 1.1248 - accura
cy: 0.6195 - val_loss: 1.0665 - val_accuracy: 0.6466
Epoch 7/200
55000/55000 [==============================] - 2s 42us/sample - loss: 1.1068 - accura
cy: 0.6279 - val_loss: 1.0506 - val_accuracy: 0.6516
Epoch 8/200
55000/55000 [==============================] - 3s 53us/sample - loss: 1.0905 - accura
cy: 0.6331 - val_loss: 1.0350 - val_accuracy: 0.6662
Epoch 9/200
55000/55000 [==============================] - 2s 36us/sample - loss: 1.0763 - accura
cy: 0.6400 - val_loss: 1.0193 - val_accuracy: 0.6620
Epoch 10/200
55000/55000 [==============================] - 1s 27us/sample - loss: 1.0635 - accura
cy: 0.6460 - val_loss: 1.0207 - val_accuracy: 0.6676
Epoch 11/200
55000/55000 [==============================] - 1s 26us/sample - loss: 1.0539 - accura
cy: 0.6508 - val_loss: 1.0036 - val_accuracy: 0.6728
Epoch 12/200
55000/55000 [==============================] - 2s 29us/sample - loss: 1.0452 - accura
cy: 0.6551 - val_loss: 0.9841 - val_accuracy: 0.6824
Epoch 13/200
55000/55000 [==============================] - 2s 28us/sample - loss: 1.0383 - accura
cy: 0.6571 - val_loss: 0.9780 - val_accuracy: 0.6884
Epoch 14/200
55000/55000 [==============================] - 1s 27us/sample - loss: 1.0326 - accura
cy: 0.6594 - val_loss: 0.9727 - val_accuracy: 0.6858
Epoch 15/200
55000/55000 [==============================] - 1s 27us/sample - loss: 1.0284 - accura
cy: 0.6592 - val_loss: 0.9629 - val_accuracy: 0.6902
Epoch 16/200
55000/55000 [==============================] - 3s 46us/sample - loss: 1.0240 - accura
cy: 0.6624 - val_loss: 0.9620 - val_accuracy: 0.6914
Epoch 17/200
55000/55000 [==============================] - 2s 40us/sample - loss: 1.0206 - accura
cy: 0.6629 - val_loss: 0.9741 - val_accuracy: 0.6858
Epoch 18/200
55000/55000 [==============================] - 2s 29us/sample - loss: 1.0176 - accura
cy: 0.6641 - val_loss: 0.9533 - val_accuracy: 0.6986
Epoch 19/200
55000/55000 [==============================] - 2s 30us/sample - loss: 1.0152 - accura
cy: 0.6648 - val_loss: 0.9528 - val_accuracy: 0.6924
Epoch 20/200
55000/55000 [==============================] - 2s 42us/sample - loss: 1.0120 - accura
cy: 0.6655 - val_loss: 0.9447 - val_accuracy: 0.6942
```

4.2) Evaluate Model Performance on Testing Dataset

Let's apply the model to the testing dataset and evaluate its performance.

In [66]:
```
model = tf.keras.models.load_model("DNN_model.h5")
print(f"Test acc: {model.evaluate(x_test_norm, y_test_encoded)[1]:.3f}")
```

```
Test acc: 0.669
```

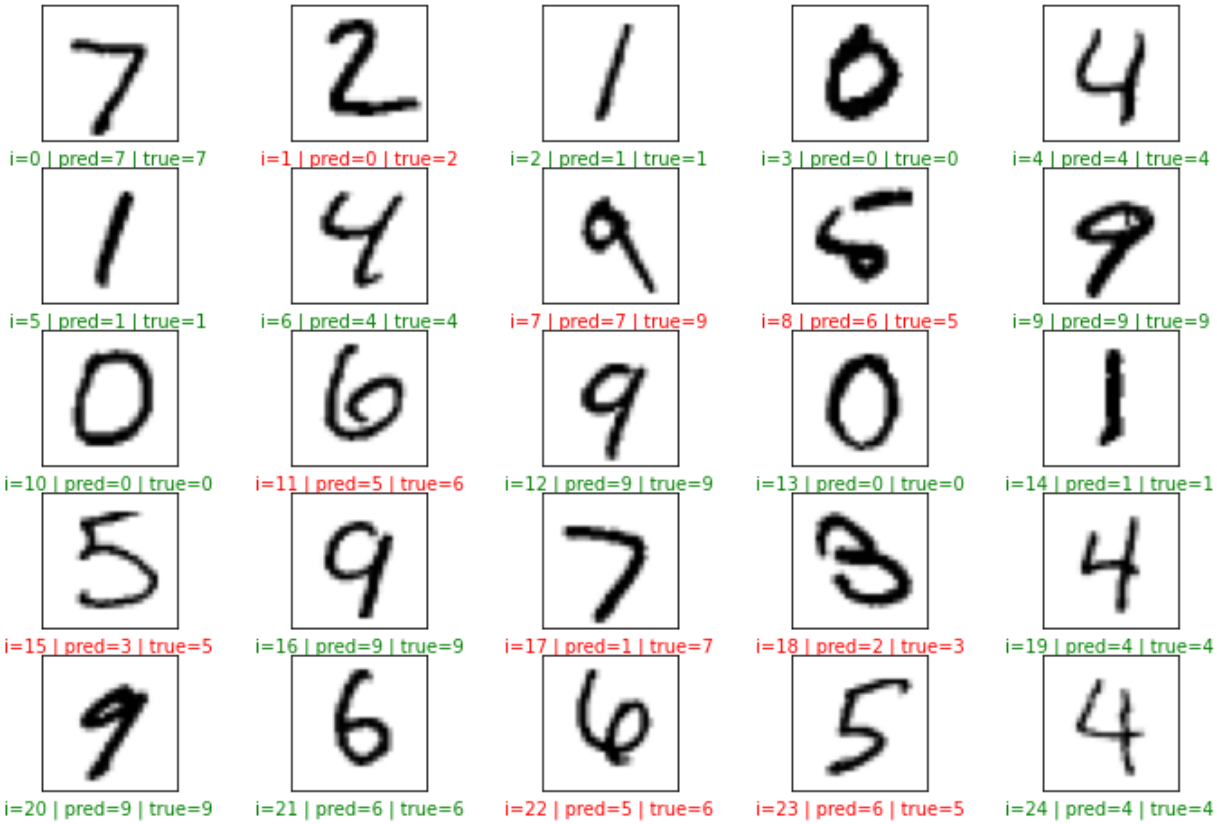
In [67]:
```
# loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
# print('test set accuracy: ', accuracy * 100)
```

In [68]:
```
preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
shape of preds:  (10000, 10)
```

As part of our model evaluation, let's look at the first 25 images by plotting the test set images along with their predicted and actual labels to understand how the trained model actually performed on specific example images.

In [69]:
```
plt.figure(figsize = (12, 8))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```



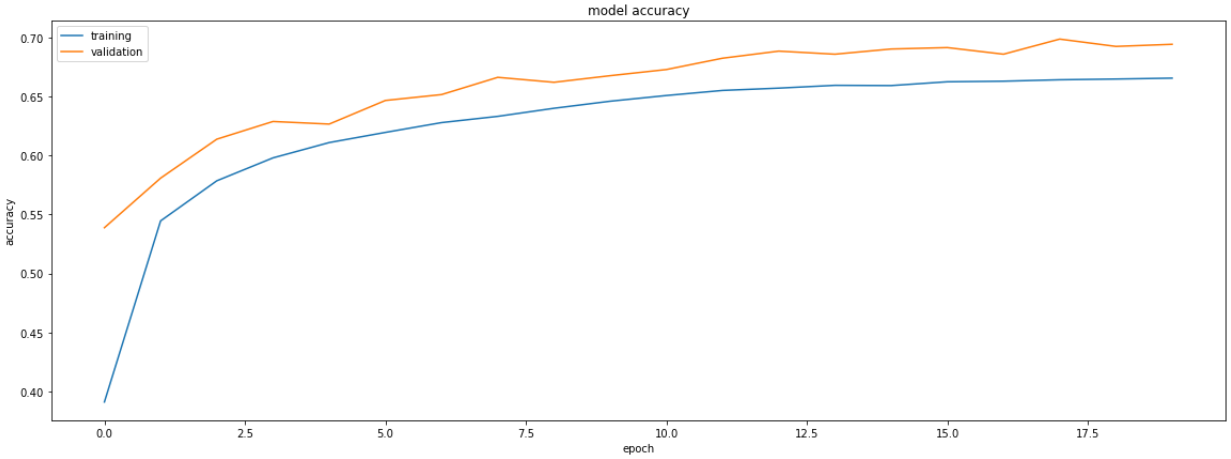
Let's use `Matplotlib` to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

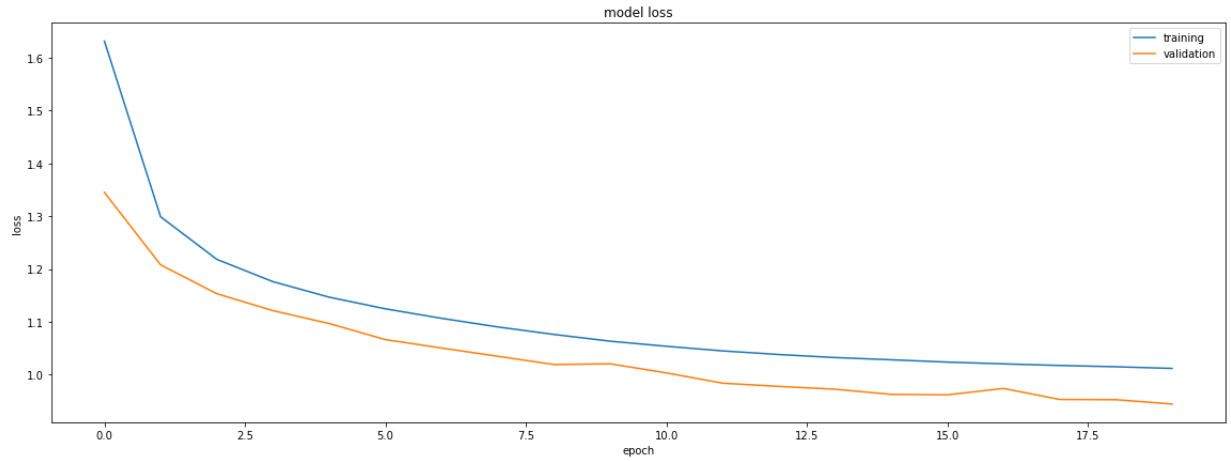
In [70]:
```
history_dict = history.history
history_dict.keys()
```

Out[70]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [71]:
```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)
```

In [72]:
```
history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)

def display_training_curves(training, validation, title, subplot):
  ax = plt.subplot(subplot)
  ax.plot(training)
  ax.plot(validation)
  ax.set_title('model '+ title)
  ax.set_ylabel(title)
  ax.set_xlabel('epoch')
  ax.legend(['training', 'validation'])

plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```





Let's examine precision and recall performance metrics for each of the prediction classes.

In [73]:
```
pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)

print_validation_report(y_test, pred1)
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.74      0.90      0.81       980
           1       0.83      0.93      0.87      1135
           2       0.43      0.38      0.40      1032
           3       0.47      0.49      0.48      1010
           4       0.76      0.81      0.78       982
           5       0.51      0.43      0.47       892
           6       0.82      0.83      0.83       958
           7       0.79      0.71      0.75      1028
           8       0.51      0.42      0.46       974
           9       0.68      0.75      0.71      1009

    accuracy                           0.67     10000
   macro avg       0.66      0.66      0.66     10000
weighted avg       0.66      0.67      0.66     10000

Accuracy Score: 0.6685
Root Mean Square Error: 1.9782315334661917
```

Let's create a table that visualizes the model output for each of the first 20 images. These outputs can be thought of as the model's expression of the probability that each image corresponds to each digit class.

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

In [74]:
```
# Get the predicted classes:
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning
pred_classes = np.argmax(model.predict(x_test_norm), axis=-1)
pred_classes;
```

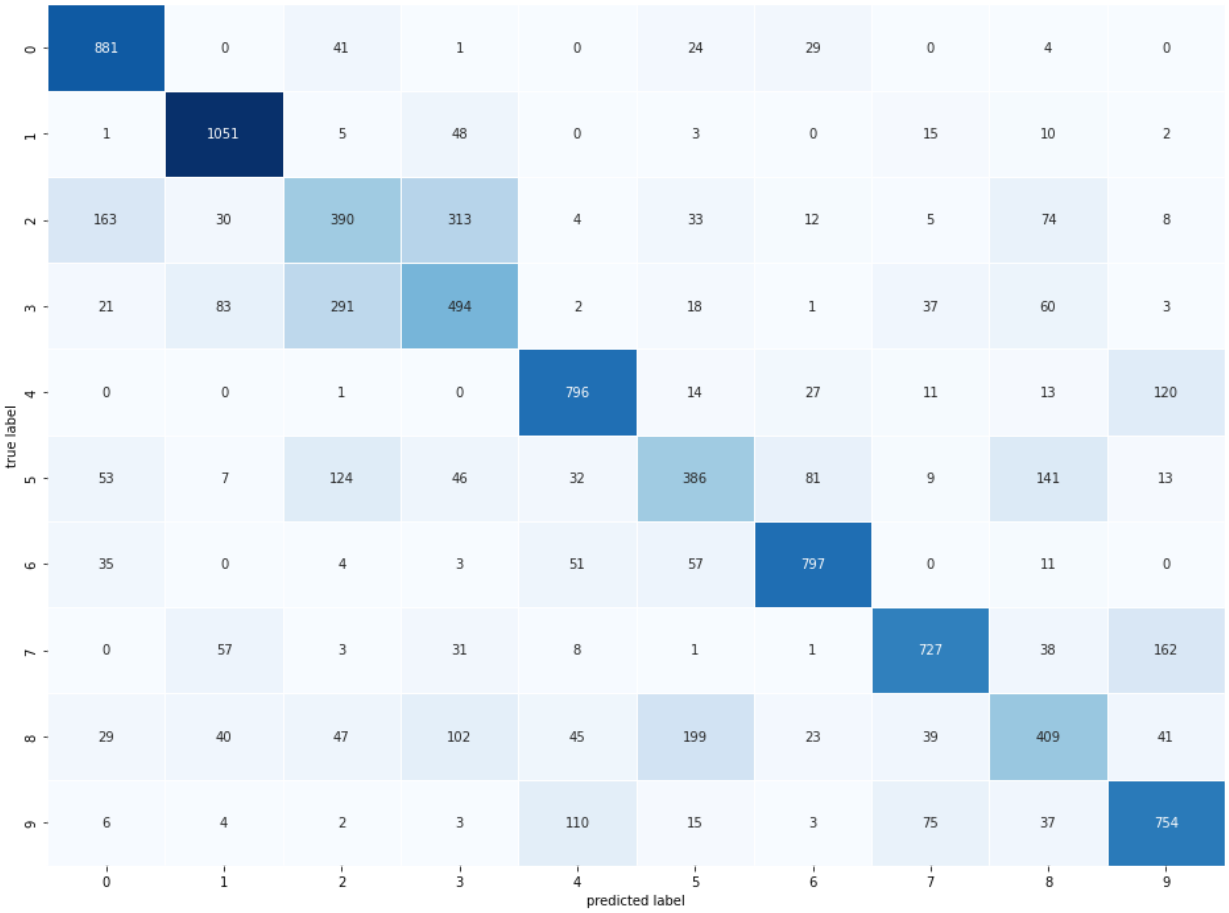
In [75]:
```
conf_mx = tf.math.confusion_matrix(y_test, pred_classes)
conf_mx;

cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8',
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

Out[75]:

	0	1	2	3	4	5	6	7	8	9
0	0.00%	7.24%	0.02%	0.57%	0.00%	0.02%	0.00%	86.58%	1.10%	4.47%
1	81.92%	0.01%	10.44%	1.86%	0.00%	4.24%	0.48%	0.00%	1.04%	0.00%
2	0.00%	62.79%	3.11%	19.09%	0.00%	0.33%	0.00%	9.75%	4.53%	0.40%
3	81.92%	0.01%	10.44%	1.86%	0.00%	4.24%	0.48%	0.00%	1.04%	0.00%
4	0.00%	0.00%	0.10%	0.07%	55.35%	7.01%	1.36%	1.23%	8.74%	26.14%
5	0.00%	89.87%	0.42%	5.16%	0.00%	0.02%	0.00%	3.94%	0.54%	0.05%
6	0.00%	0.00%	0.00%	0.00%	82.77%	0.45%	0.29%	0.19%	0.62%	15.67%
7	0.00%	0.36%	0.48%	1.78%	1.35%	1.99%	0.00%	43.40%	17.16%	33.47%
8	0.06%	0.00%	0.14%	0.02%	37.16%	12.39%	44.31%	0.03%	3.57%	2.31%
9	0.00%	0.00%	0.00%	0.00%	4.21%	0.02%	0.00%	20.84%	0.52%	74.41%
10	81.92%	0.01%	10.44%	1.86%	0.00%	4.24%	0.48%	0.00%	1.04%	0.00%
11	0.76%	0.03%	12.45%	5.68%	2.04%	41.44%	2.42%	0.76%	31.88%	2.54%
12	0.00%	0.00%	0.01%	0.03%	12.78%	0.49%	0.00%	13.56%	3.59%	69.53%
13	81.72%	0.00%	9.20%	1.52%	0.01%	5.40%	0.93%	0.00%	1.22%	0.00%
14	0.00%	92.84%	0.18%	3.00%	0.00%	0.01%	0.00%	3.65%	0.28%	0.03%
15	0.08%	2.68%	24.31%	29.65%	0.09%	12.05%	0.03%	2.87%	27.05%	1.19%
16	0.00%	0.00%	0.06%	0.07%	36.35%	3.37%	0.15%	4.20%	8.42%	47.37%
17	0.00%	65.48%	0.65%	7.51%	0.00%	0.09%	0.00%	23.13%	2.47%	0.67%
18	0.51%	3.06%	37.50%	35.32%	0.01%	8.65%	0.02%	0.65%	14.08%	0.19%
19	0.00%	0.00%	0.04%	0.02%	70.89%	4.65%	2.25%	0.44%	4.43%	17.28%

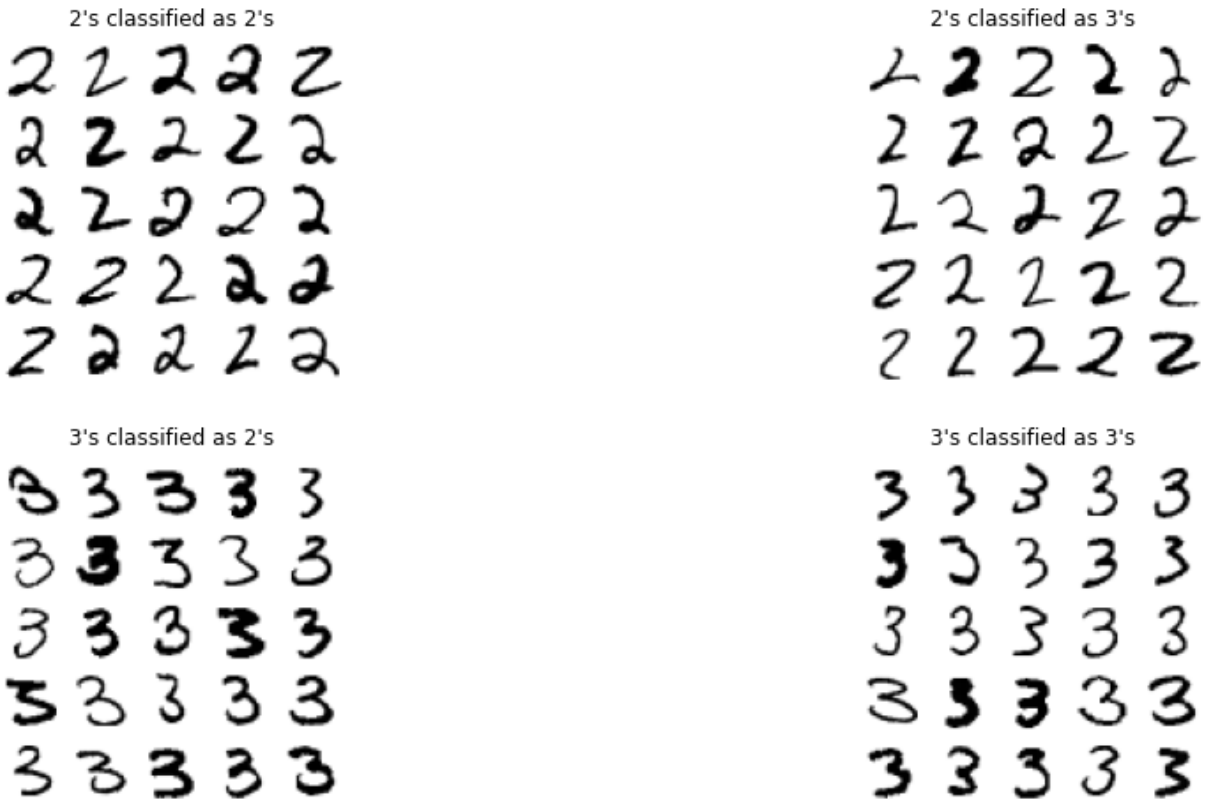
Let's create a confusion matrix that visualizes the model performance on testing data.

In [76]:
```
plot_confusion_matrix(y_test,pred_classes)
```



In [77]:
```
cl_a, cl_b = 2, 3
X_aa = x_test_norm[(y_test == cl_a) & (pred_classes == cl_a)]
X_ab = x_test_norm[(y_test == cl_a) & (pred_classes == cl_b)]
X_ba = x_test_norm[(y_test == cl_b) & (pred_classes == cl_a)]
X_bb = x_test_norm[(y_test == cl_b) & (pred_classes == cl_b)]

plt.figure(figsize=(16,8))

p1 = plt.subplot(221)
p2 = plt.subplot(222)
p3 = plt.subplot(223)
p4 = plt.subplot(224)

plot_digits(X_aa[:25], p1, images_per_row=5);
plot_digits(X_ab[:25], p2, images_per_row=5);
plot_digits(X_ba[:25], p3, images_per_row=5);
plot_digits(X_bb[:25], p4, images_per_row=5);


p1.set_title(f"{cl_a}'s classified as {cl_a}'s")
p2.set_title(f"{cl_a}'s classified as {cl_b}'s")
p3.set_title(f"{cl_b}'s classified as {cl_a}'s")
p4.set_title(f"{cl_b}'s classified as {cl_b}'s")

# plt.savefig("error_analysis_digits_plot_EXP1_valid")
```

```
plt.show()
```



4.3) Examine Model Activation Values

To get the activation values of the hidden nodes, we need to create a new model, `activation_model`, that takes the same input as our current model but outputs the activation value of the hidden layer, i.e. of the hidden node. Then use the `predict` function to get the activation values.

In [87]:
```
from tensorflow.keras import models

# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs # description of the layers


# Get the output of the hidden node for each of the 55000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
hidden_layer_activation.shape   #  2 hidden node each has one activation value per tra



hidden_node1_activation = hidden_layer_activation[:,0] # get activation values of the
hidden_node2_activation = hidden_layer_activation[:,1] # get activation values of the
```

```
print(f"The maximum activation value of the first hidden node is {hidden_node1_activat
print(f"The maximum activation value of the second hidden node is {hidden_node2_activa
```

```
# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in s
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum
```

```
There are 2 layers
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
The maximum activation value of the first hidden node is 9.918289184570312
The maximum activation value of the second hidden node is 8.35073471069336
The output node has shape (60000, 10)
The output for the first image are [0.053 0.003 0.391 0.186 0.001 0.203 0.003 0.002
0.155 0.002]
The sum of the probabilities is (approximately) 0.9999998807907104
```

We combine the activation values of the two hidden nodes together with the corresponding predicted classes into a DataFrame. We use both matplotlib and seaborn to create boxplots from the DataFrame.

In [92]:
```
print(len(hidden_node1_activation))
```

```
60000
```

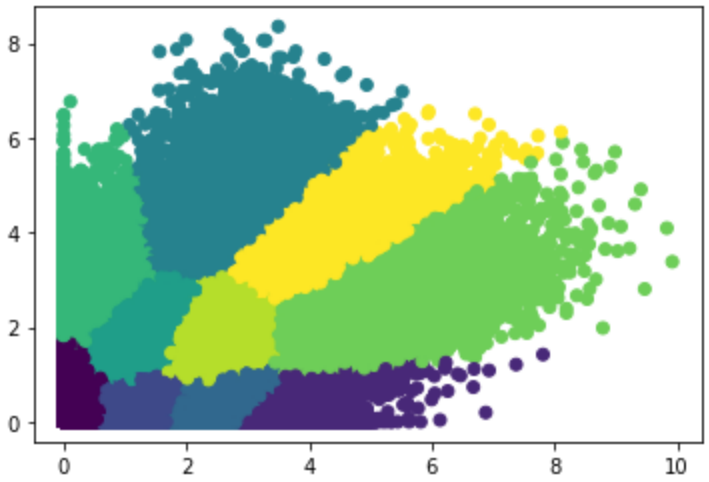
In [93]:
```
print(len(hidden_node2_activation))
```

```
60000
```

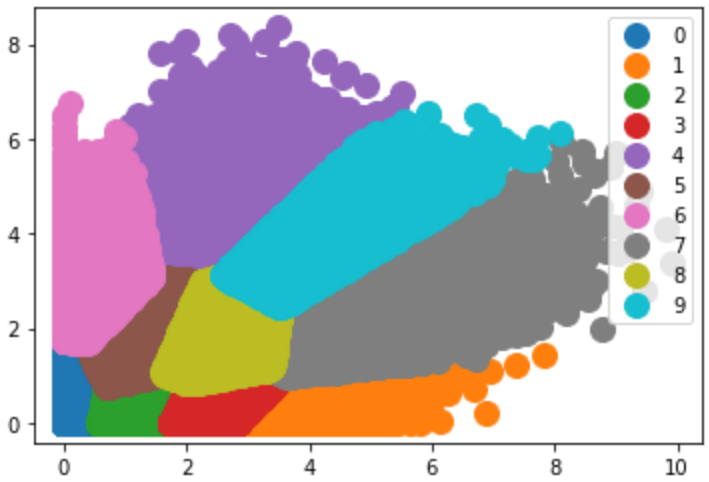
In [97]:
```
pred_classes_train = np.argmax(model.predict(x_train_norm), axis=-1)


print(len(pred_classes_train))
```

```
60000
```

In [98]:
```
scatterPlot_df =  pd.DataFrame({'act_value_h1':hidden_node1_activation,
                                'act_value_h2':hidden_node2_activation,
                                'pred_class':pred_classes_train})



#plt.legend(loc='upper left', prop={'size':6}, bbox_to_anchor=(1,1),ncol=1)
plt.scatter(scatterPlot_df.act_value_h1,
            scatterPlot_df.act_value_h2,
            c=scatterPlot_df.pred_class,
            label=scatterPlot_df.pred_class)
plt.show()
```



In [99]:
```
groups = scatterPlot_df.groupby('pred_class')

# Plot
fig, ax = plt.subplots()
ax.margins(0.05) # Optional, just adds 5% padding to the autoscaling
for name, group in groups:
    ax.plot(group.act_value_h1, group.act_value_h2, marker='o', linestyle='', ms=12, l
ax.legend()

plt.show()
```



5) Experiment 3 - ANN with 1 Hidden Layer with 10 Nodes

5.1) Construct Model

Let's construct our an artificial neural network to classify images as integers.

In [102…
```
tf.keras.backend.clear_session()

model = Sequential([
    Dense(input_shape=[784], units=10, activation = tf.nn.relu,kernel_regularizer=tf.k
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])
```

In [103…
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                7850

 output_layer (Dense)        (None, 10)                110

=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
_________________________________________________________________
```

In [104…
```
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])
```

In [105…
```
#tf.keras.model.fit
#https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

#tf.keras.callbacks.EarlyStopping
#https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping


history = model.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=(5000/60000)
    ,callbacks=[tf.keras.callbacks.ModelCheckpoint("DNN_model.h5",save_best_only=True,
                ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/200
54688/55000 [============================>.] - ETA: 0s - loss: 0.5816 - accuracy: 0.8
492
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2335: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates
```

```
55000/55000 [==============================] - 5s 100us/sample - loss: 0.5808 - accur
acy: 0.8494 - val_loss: 0.3204 - val_accuracy: 0.9234
Epoch 2/200
55000/55000 [==============================] - 2s 29us/sample - loss: 0.3766 - accura
cy: 0.9069 - val_loss: 0.2915 - val_accuracy: 0.9332
Epoch 3/200
55000/55000 [==============================] - 2s 36us/sample - loss: 0.3541 - accura
cy: 0.9127 - val_loss: 0.2798 - val_accuracy: 0.9344
Epoch 4/200
55000/55000 [==============================] - 2s 34us/sample - loss: 0.3410 - accura
cy: 0.9150 - val_loss: 0.2760 - val_accuracy: 0.9342
Epoch 5/200
55000/55000 [==============================] - 2s 34us/sample - loss: 0.3334 - accura
cy: 0.9168 - val_loss: 0.2684 - val_accuracy: 0.9370
Epoch 6/200
55000/55000 [==============================] - 2s 32us/sample - loss: 0.3258 - accura
cy: 0.9182 - val_loss: 0.2659 - val_accuracy: 0.9378
Epoch 7/200
55000/55000 [==============================] - 2s 38us/sample - loss: 0.3200 - accura
cy: 0.9196 - val_loss: 0.2605 - val_accuracy: 0.9350
Epoch 8/200
55000/55000 [==============================] - 2s 37us/sample - loss: 0.3139 - accura
cy: 0.9203 - val_loss: 0.2623 - val_accuracy: 0.9394
Epoch 9/200
55000/55000 [==============================] - 2s 39us/sample - loss: 0.3094 - accura
cy: 0.9223 - val_loss: 0.2523 - val_accuracy: 0.9398
Epoch 10/200
55000/55000 [==============================] - 2s 30us/sample - loss: 0.3051 - accura
cy: 0.9233 - val_loss: 0.2514 - val_accuracy: 0.9410
Epoch 11/200
55000/55000 [==============================] - 2s 39us/sample - loss: 0.3002 - accura
cy: 0.9230 - val_loss: 0.2485 - val_accuracy: 0.9396
Epoch 12/200
55000/55000 [==============================] - 2s 38us/sample - loss: 0.2946 - accura
cy: 0.9251 - val_loss: 0.2536 - val_accuracy: 0.9370
```

5.2) Evaluate Model Performance on Testing Dataset

Let's apply the model to the testing dataset and evaluate its performance.

In [106…

```
model = tf.keras.models.load_model("DNN_model.h5")
print(f"Test acc: {model.evaluate(x_test_norm, y_test_encoded)[1]:.3f}")
```

```
Test acc: 0.927
```

In [ ]:

```
# loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
# print('test set accuracy: ', accuracy * 100)
```

In [107…

```
preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
shape of preds:  (10000, 10)
```

As part of our model evaluation, let's look at the first 25 images by plotting the test set images along with their predicted and actual labels to understand how the trained model actually performed on specific example images.

In [108…

```
plt.figure(figsize = (12, 8))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```

Let's use `Matplotlib` to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

In [109…

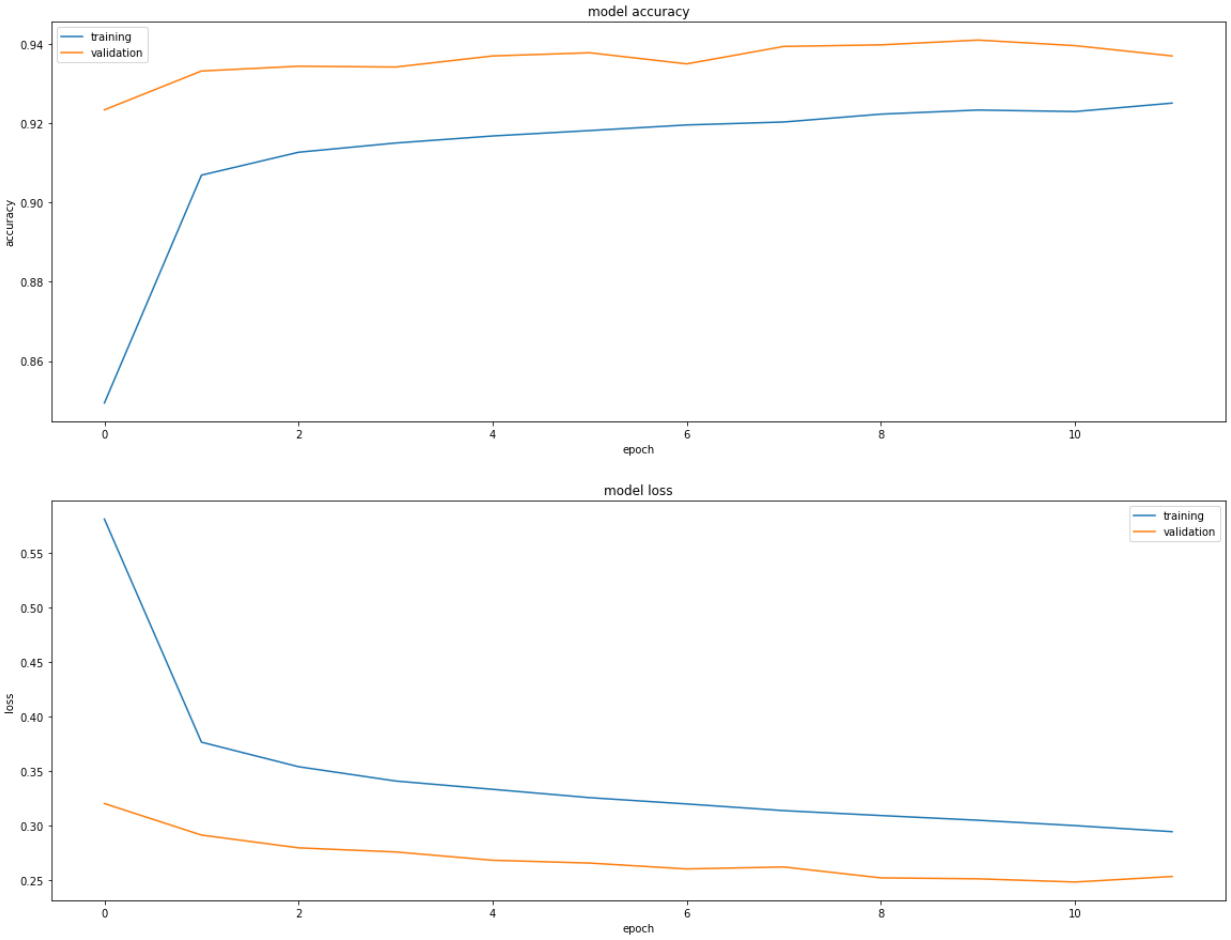
```
history_dict = history.history
history_dict.keys()
```

Out[109]:

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [110…
```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)
```

In [111…
```
history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)

def display_training_curves(training, validation, title, subplot):
  ax = plt.subplot(subplot)
  ax.plot(training)
  ax.plot(validation)
  ax.set_title('model '+ title)
  ax.set_ylabel(title)
  ax.set_xlabel('epoch')
  ax.legend(['training', 'validation'])

plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```



Let's examine precision and recall performance metrics for each of the prediction classes.

In [112…
```
pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)
```

```
print_validation_report(y_test, pred1)
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.97      0.97      0.97       980
           1       0.97      0.98      0.98      1135
           2       0.92      0.91      0.92      1032
           3       0.93      0.90      0.91      1010
           4       0.92      0.94      0.93       982
           5       0.91      0.86      0.89       892
           6       0.95      0.95      0.95       958
           7       0.92      0.94      0.93      1028
           8       0.89      0.88      0.88       974
           9       0.90      0.92      0.91      1009

    accuracy                           0.93     10000
   macro avg       0.93      0.93      0.93     10000
weighted avg       0.93      0.93      0.93     10000

Accuracy Score: 0.927
Root Mean Square Error: 1.126765281680262
```

Let's create a table that visualizes the model output for each of the first 20 images. These outputs can be thought of as the model's expression of the probability that each image corresponds to each digit class.

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

In [113…

```
# Get the predicted classes:
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning
pred_classes = np.argmax(model.predict(x_test_norm), axis=-1)
pred_classes;
```

In [114…

```
conf_mx = tf.math.confusion_matrix(y_test, pred_classes)
conf_mx;

cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8',
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

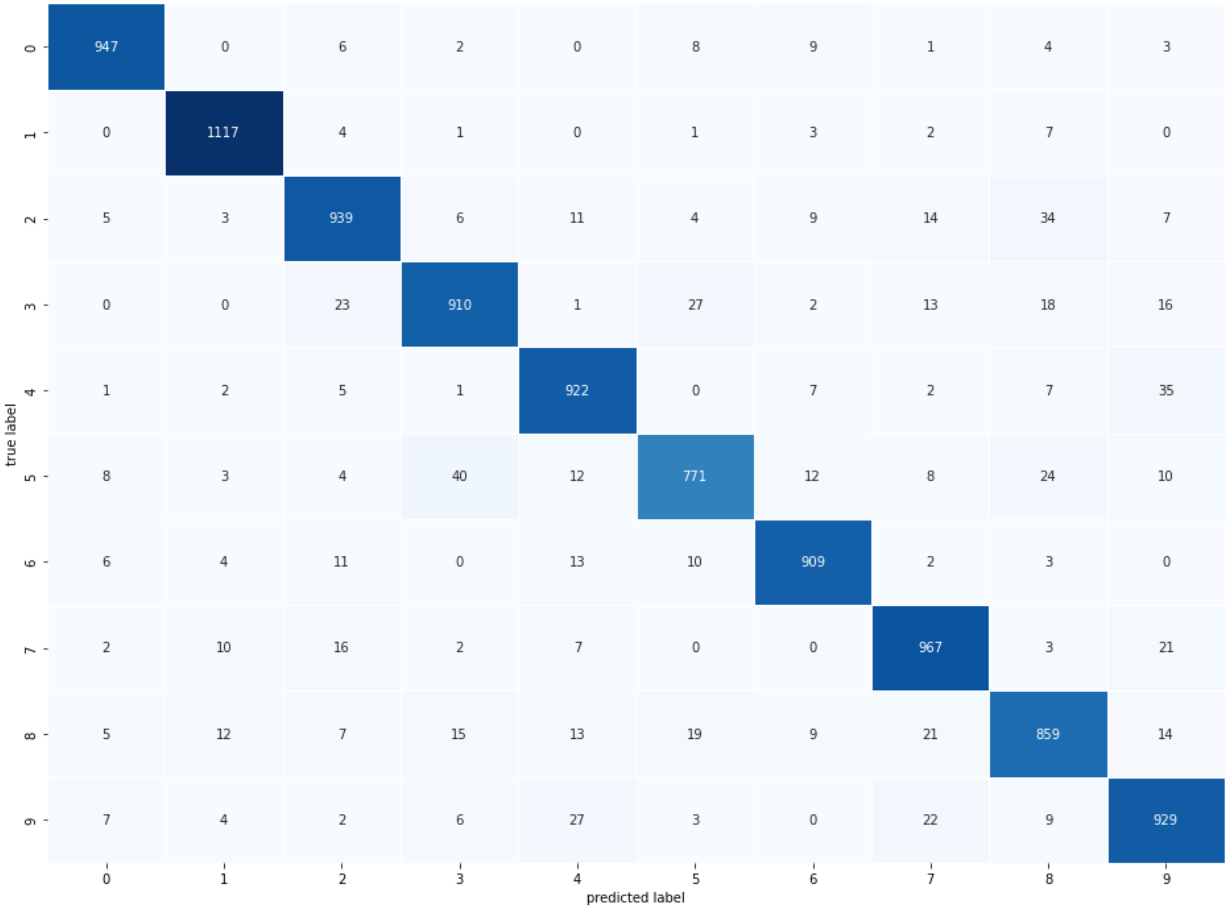
Out[114]:

	0	1	2	3	4	5	6	7	8	9
0	0.01%	0.00%	0.09%	0.16%	0.00%	0.00%	0.00%	99.67%	0.01%	0.08%
1	0.02%	0.00%	99.82%	0.06%	0.00%	0.07%	0.00%	0.00%	0.04%	0.00%
2	0.00%	98.57%	0.43%	0.17%	0.01%	0.02%	0.08%	0.50%	0.20%	0.01%
3	99.87%	0.00%	0.08%	0.00%	0.00%	0.02%	0.02%	0.00%	0.00%	0.00%
4	0.01%	0.00%	0.22%	0.00%	95.19%	0.00%	0.06%	0.35%	0.07%	4.09%
5	0.00%	98.24%	0.22%	0.12%	0.00%	0.00%	0.00%	1.24%	0.16%	0.01%
6	0.00%	0.00%	0.00%	0.00%	99.52%	0.01%	0.00%	0.02%	0.16%	0.29%
7	0.00%	0.02%	0.08%	0.34%	0.58%	1.48%	0.00%	0.06%	0.73%	96.70%
8	0.01%	0.00%	0.08%	0.00%	0.32%	0.02%	99.54%	0.00%	0.03%	0.00%
9	0.00%	0.00%	0.00%	0.00%	2.96%	0.00%	0.00%	7.03%	0.28%	89.73%
10	97.57%	0.00%	0.10%	0.02%	0.00%	2.05%	0.00%	0.00%	0.25%	0.00%
11	0.08%	0.01%	3.20%	0.04%	0.42%	0.02%	95.53%	0.00%	0.66%	0.04%
12	0.00%	0.00%	0.02%	0.04%	3.03%	0.01%	0.00%	0.94%	0.17%	95.77%
13	98.00%	0.00%	0.02%	0.00%	0.00%	0.54%	0.00%	0.02%	0.52%	0.90%
14	0.00%	99.73%	0.04%	0.17%	0.00%	0.00%	0.00%	0.01%	0.04%	0.00%
15	0.07%	0.00%	0.77%	2.17%	0.00%	95.38%	0.02%	0.00%	1.58%	0.01%
16	0.01%	0.00%	0.06%	0.00%	1.97%	0.00%	0.00%	5.09%	0.26%	92.59%
17	0.01%	0.00%	0.06%	0.06%	0.00%	0.00%	0.00%	99.86%	0.00%	0.00%
18	0.00%	1.86%	9.28%	83.00%	0.01%	1.59%	0.31%	0.14%	3.79%	0.02%
19	0.00%	0.00%	0.02%	0.02%	99.14%	0.04%	0.01%	0.01%	0.03%	0.73%

Let's create a confusion matrix that visualizes model performance on testing data.

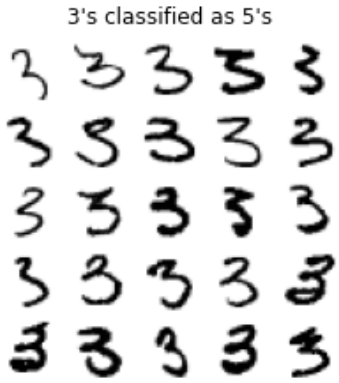
In [115…

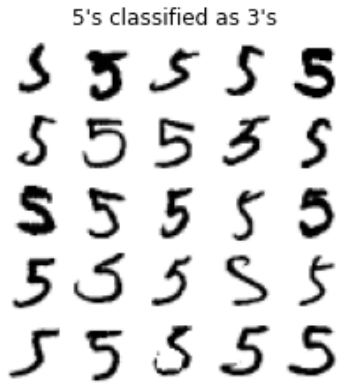
```
plot_confusion_matrix(y_test,pred_classes)
```

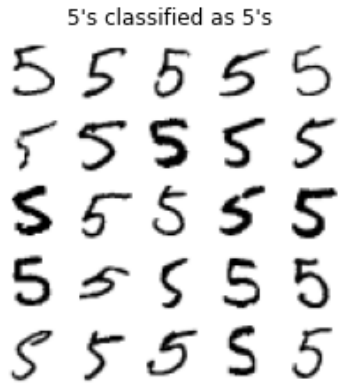


In [116…

```
cl_a, cl_b = 3, 5
X_aa = x_test_norm[(y_test == cl_a) & (pred_classes == cl_a)]
X_ab = x_test_norm[(y_test == cl_a) & (pred_classes == cl_b)]
X_ba = x_test_norm[(y_test == cl_b) & (pred_classes == cl_a)]
X_bb = x_test_norm[(y_test == cl_b) & (pred_classes == cl_b)]

plt.figure(figsize=(16,8))

p1 = plt.subplot(221)
p2 = plt.subplot(222)
p3 = plt.subplot(223)
p4 = plt.subplot(224)

plot_digits(X_aa[:25], p1, images_per_row=5);
plot_digits(X_ab[:25], p2, images_per_row=5);
plot_digits(X_ba[:25], p3, images_per_row=5);
plot_digits(X_bb[:25], p4, images_per_row=5);


p1.set_title(f"{cl_a}'s classified as {cl_a}'s")
p2.set_title(f"{cl_a}'s classified as {cl_b}'s")
p3.set_title(f"{cl_b}'s classified as {cl_a}'s")
p4.set_title(f"{cl_b}'s classified as {cl_b}'s")

# plt.savefig("error_analysis_digits_plot_EXP1_valid")

plt.show()
```







5.3) Examine Model Activation Values via T-SNE Plot

To get the activation values of the hidden nodes, we need to create a new model, `activation_model`, that takes the same input as our current model but outputs the activation value of the hidden layer, i.e. of the hidden node. Then use the `predict` function to get the activation values. We will use this to create t-SNE plots that visualize how much overlap exists between the predicted classes of images.

In [117…

```
# Extracts the outputs of the 2 layers:
layer_outputs = [layer.output for layer in model.layers]

# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

print(f"There are {len(layer_outputs)} layers")
layer_outputs; # description of the layers
```

```
There are 2 layers
```

In [118…

```
# Get the outputs of all the hidden nodes for each of the 60000 training images
activations = activation_model.predict(x_train_norm)
hidden_layer_activation = activations[0]
output_layer_activations = activations[1]
hidden_layer_activation.shape   #  each of the 128 hidden nodes has one activation val
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
```

Out[118]: (60000, 10)

In [119…
```
output_layer_activations.shape
```

Out[119]: (60000, 10)

In [120…
```
print(f"The maximum activation value of the hidden nodes in the hidden layer is \
{hidden_layer_activation.max()}")
```

```
The maximum activation value of the hidden nodes in the hidden layer is 12.191492080688477
```

In [121…
```
# Some stats about the output layer as an aside...
np.set_printoptions(suppress = True)  # display probabilities as decimals and NOT in s
ouput_layer_activation = activations[1]
print(f"The output node has shape {ouput_layer_activation.shape}")
print(f"The output for the first image are {ouput_layer_activation[0].round(4)}")
print(f"The sum of the probabilities is (approximately) {ouput_layer_activation[0].sum
```

```
The output node has shape (60000, 10)
The output for the first image are [0.    0.    0.009 0.66  0.    0.325 0.    0.001
0.005 0.   ]
The sum of the probabilities is (approximately) 1.0
```

In [122…
```
#Get the dataframe of all the node values
activation_data = {'actual_class':y_train}
for k in range(0,10):
    activation_data[f"act_val_{k}"] = hidden_layer_activation[:,k]

activation_df = pd.DataFrame(activation_data)
activation_df.head(15).round(3).T
```

Out[122]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	
actual_class	5.000	0.000	4.000	1.000	9.000	2.000	1.000	3.000	1.000	4.000	3.000	5.000	3.000	6.
act_val_0	5.147	6.765	0.654	1.204	1.474	4.278	0.949	4.477	1.297	0.000	4.551	0.000	3.257	2.
act_val_1	4.647	0.205	0.450	4.242	2.828	3.969	7.170	3.126	5.299	1.937	4.117	3.122	4.055	4.
act_val_2	3.728	1.765	2.473	0.387	0.859	0.325	1.768	3.976	0.880	3.533	3.019	0.777	5.016	3.
act_val_3	0.000	1.734	1.906	0.020	0.482	3.988	0.000	0.000	0.000	2.539	0.000	1.298	0.000	3.
act_val_4	2.296	0.243	2.370	2.110	4.239	3.983	4.048	6.959	2.575	1.303	6.383	0.000	5.742	0.
act_val_5	1.253	0.943	0.594	4.183	0.000	2.769	3.646	2.523	2.452	0.391	3.130	3.146	3.297	0.
act_val_6	0.787	2.743	0.297	1.593	3.100	2.710	0.431	2.462	0.329	2.798	1.536	1.583	0.000	3.
act_val_7	1.237	0.000	2.701	1.082	2.200	0.479	2.326	0.568	1.095	3.189	0.428	0.000	1.505	3.
act_val_8	4.163	2.628	0.394	0.594	0.000	1.613	0.559	5.986	0.000	0.000	3.794	0.242	5.228	0.
act_val_9	2.145	2.760	1.552	0.316	4.092	3.291	0.000	1.878	0.453	3.616	0.838	3.307	1.489	0.

In [123…
```
activation_df.shape
```

Out[123]: (60000, 11)

In [124…
```
N=60000
activation_df_subset = activation_df.iloc[:N].copy()
activation_df_subset.shape
```

Out[124]:
```
(60000, 11)
```

In [126…
```
features = [*activation_data][1:]
data_subset = activation_df_subset[features].values
data_subset.shape
```

Out[126]:
```
(60000, 10)
```

In [127…
```
%%time
tsne = TSNE(n_components=2
            ,init='pca'
            ,learning_rate='auto'
            ,verbose=1
            ,perplexity=40, n_iter=300)
tsne_results = tsne.fit_transform(data_subset)
```

```
[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 60000 samples in 0.248s...
[t-SNE] Computed neighbors for 60000 samples in 33.151s...
[t-SNE] Computed conditional probabilities for sample 1000 / 60000
[t-SNE] Computed conditional probabilities for sample 2000 / 60000
[t-SNE] Computed conditional probabilities for sample 3000 / 60000
[t-SNE] Computed conditional probabilities for sample 4000 / 60000
[t-SNE] Computed conditional probabilities for sample 5000 / 60000
[t-SNE] Computed conditional probabilities for sample 6000 / 60000
[t-SNE] Computed conditional probabilities for sample 7000 / 60000
[t-SNE] Computed conditional probabilities for sample 8000 / 60000
[t-SNE] Computed conditional probabilities for sample 9000 / 60000
[t-SNE] Computed conditional probabilities for sample 10000 / 60000
[t-SNE] Computed conditional probabilities for sample 11000 / 60000
[t-SNE] Computed conditional probabilities for sample 12000 / 60000
[t-SNE] Computed conditional probabilities for sample 13000 / 60000
[t-SNE] Computed conditional probabilities for sample 14000 / 60000
[t-SNE] Computed conditional probabilities for sample 15000 / 60000
[t-SNE] Computed conditional probabilities for sample 16000 / 60000
[t-SNE] Computed conditional probabilities for sample 17000 / 60000
[t-SNE] Computed conditional probabilities for sample 18000 / 60000
[t-SNE] Computed conditional probabilities for sample 19000 / 60000
[t-SNE] Computed conditional probabilities for sample 20000 / 60000
[t-SNE] Computed conditional probabilities for sample 21000 / 60000
[t-SNE] Computed conditional probabilities for sample 22000 / 60000
[t-SNE] Computed conditional probabilities for sample 23000 / 60000
[t-SNE] Computed conditional probabilities for sample 24000 / 60000
[t-SNE] Computed conditional probabilities for sample 25000 / 60000
[t-SNE] Computed conditional probabilities for sample 26000 / 60000
[t-SNE] Computed conditional probabilities for sample 27000 / 60000
[t-SNE] Computed conditional probabilities for sample 28000 / 60000
[t-SNE] Computed conditional probabilities for sample 29000 / 60000
[t-SNE] Computed conditional probabilities for sample 30000 / 60000
[t-SNE] Computed conditional probabilities for sample 31000 / 60000
[t-SNE] Computed conditional probabilities for sample 32000 / 60000
[t-SNE] Computed conditional probabilities for sample 33000 / 60000
[t-SNE] Computed conditional probabilities for sample 34000 / 60000
[t-SNE] Computed conditional probabilities for sample 35000 / 60000
[t-SNE] Computed conditional probabilities for sample 36000 / 60000
[t-SNE] Computed conditional probabilities for sample 37000 / 60000
[t-SNE] Computed conditional probabilities for sample 38000 / 60000
[t-SNE] Computed conditional probabilities for sample 39000 / 60000
[t-SNE] Computed conditional probabilities for sample 40000 / 60000
[t-SNE] Computed conditional probabilities for sample 41000 / 60000
[t-SNE] Computed conditional probabilities for sample 42000 / 60000
[t-SNE] Computed conditional probabilities for sample 43000 / 60000
[t-SNE] Computed conditional probabilities for sample 44000 / 60000
[t-SNE] Computed conditional probabilities for sample 45000 / 60000
[t-SNE] Computed conditional probabilities for sample 46000 / 60000
[t-SNE] Computed conditional probabilities for sample 47000 / 60000
[t-SNE] Computed conditional probabilities for sample 48000 / 60000
[t-SNE] Computed conditional probabilities for sample 49000 / 60000
[t-SNE] Computed conditional probabilities for sample 50000 / 60000
[t-SNE] Computed conditional probabilities for sample 51000 / 60000
[t-SNE] Computed conditional probabilities for sample 52000 / 60000
[t-SNE] Computed conditional probabilities for sample 53000 / 60000
[t-SNE] Computed conditional probabilities for sample 54000 / 60000
[t-SNE] Computed conditional probabilities for sample 55000 / 60000
[t-SNE] Computed conditional probabilities for sample 56000 / 60000
[t-SNE] Computed conditional probabilities for sample 57000 / 60000
```

```
[t-SNE] Computed conditional probabilities for sample 58000 / 60000
[t-SNE] Computed conditional probabilities for sample 59000 / 60000
[t-SNE] Computed conditional probabilities for sample 60000 / 60000
[t-SNE] Mean sigma: 0.681261
[t-SNE] KL divergence after 250 iterations with early exaggeration: 96.393295
[t-SNE] KL divergence after 300 iterations: 4.104528
CPU times: total: 8min 37s
Wall time: 2min 20s
```

In [128…
```
tsne_results = (tsne_results - tsne_results.min()) / (tsne_results.max() - tsne_result
```

In [129…
```
cmap = plt.cm.tab10
plt.figure(figsize=(16,10))
plt.scatter(tsne_results[:,0],tsne_results[:,1], c=y_train, s=10, cmap=cmap)

image_positions = np.array([[1., 1.]])
for index, position in enumerate(tsne_results):
    dist = np.sum((position - image_positions) ** 2, axis=1)
    if np.min(dist) > 0.02: # if far enough from other images
        image_positions = np.r_[image_positions, [position]]
        imagebox = mpl.offsetbox.AnnotationBbox(
            mpl.offsetbox.OffsetImage(x_train[index], cmap="binary"),
            position, bboxprops={"edgecolor": cmap(y_train[index]), "lw": 2})
        plt.gca().add_artist(imagebox)
plt.axis("off")
plt.show()
```

6) Experiment 4 - ANN with 1 Hidden Layer with 30 Nodes

6.1) Construct Model

Let's construct our an artificial neural network to classify images as integers.

In [130…
```
tf.keras.backend.clear_session()

model = Sequential([
    Dense(input_shape=[784], units=30, activation = tf.nn.relu,kernel_regularizer=tf.k
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])
```

In [131…
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 30)                23550

 output_layer (Dense)        (None, 10)                310

=================================================================
Total params: 23,860
Trainable params: 23,860
Non-trainable params: 0
_________________________________________________________________
```

In [132…
```
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])
```

In [133…
```
#tf.keras.model.fit
#https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

#tf.keras.callbacks.EarlyStopping
#https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping


history = model.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=(5000/60000)
    ,callbacks=[tf.keras.callbacks.ModelCheckpoint("DNN_model.h5",save_best_only=True,
                ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/200
54528/55000 [============================>.] - ETA: 0s - loss: 0.4466 - accuracy: 0.8
901
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2335: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates
```

```
55000/55000 [==============================] - 3s 52us/sample - loss: 0.4456 - accura
cy: 0.8903 - val_loss: 0.2629 - val_accuracy: 0.9434
Epoch 2/200
55000/55000 [==============================] - 2s 45us/sample - loss: 0.2997 - accura
cy: 0.9303 - val_loss: 0.2279 - val_accuracy: 0.9554
Epoch 3/200
55000/55000 [==============================] - 3s 57us/sample - loss: 0.2602 - accura
cy: 0.9410 - val_loss: 0.2095 - val_accuracy: 0.9570
Epoch 4/200
55000/55000 [==============================] - 3s 52us/sample - loss: 0.2342 - accura
cy: 0.9475 - val_loss: 0.1898 - val_accuracy: 0.9620
Epoch 5/200
55000/55000 [==============================] - 3s 47us/sample - loss: 0.2168 - accura
cy: 0.9516 - val_loss: 0.1810 - val_accuracy: 0.9628
Epoch 6/200
55000/55000 [==============================] - 2s 32us/sample - loss: 0.2043 - accura
cy: 0.9545 - val_loss: 0.1785 - val_accuracy: 0.9634
Epoch 7/200
55000/55000 [==============================] - 2s 39us/sample - loss: 0.1947 - accura
cy: 0.9577 - val_loss: 0.1728 - val_accuracy: 0.9642
Epoch 8/200
55000/55000 [==============================] - 3s 59us/sample - loss: 0.1884 - accura
cy: 0.9583 - val_loss: 0.1718 - val_accuracy: 0.9640
Epoch 9/200
55000/55000 [==============================] - 2s 40us/sample - loss: 0.1814 - accura
cy: 0.9605 - val_loss: 0.1587 - val_accuracy: 0.9674
Epoch 10/200
55000/55000 [==============================] - 2s 35us/sample - loss: 0.1766 - accura
cy: 0.9619 - val_loss: 0.1533 - val_accuracy: 0.9678
Epoch 11/200
55000/55000 [==============================] - 2s 43us/sample - loss: 0.1715 - accura
cy: 0.9625 - val_loss: 0.1492 - val_accuracy: 0.9678
Epoch 12/200
55000/55000 [==============================] - 2s 34us/sample - loss: 0.1691 - accura
cy: 0.9634 - val_loss: 0.1609 - val_accuracy: 0.9678
```

6.2) Evaluate Model Performance on Testing Dataset

Let's apply the model to the testing dataset and evaluate its performance.

In [134…
```
model = tf.keras.models.load_model("DNN_model.h5")
print(f"Test acc: {model.evaluate(x_test_norm, y_test_encoded)[1]:.3f}")
```

```
Test acc: 0.965
```

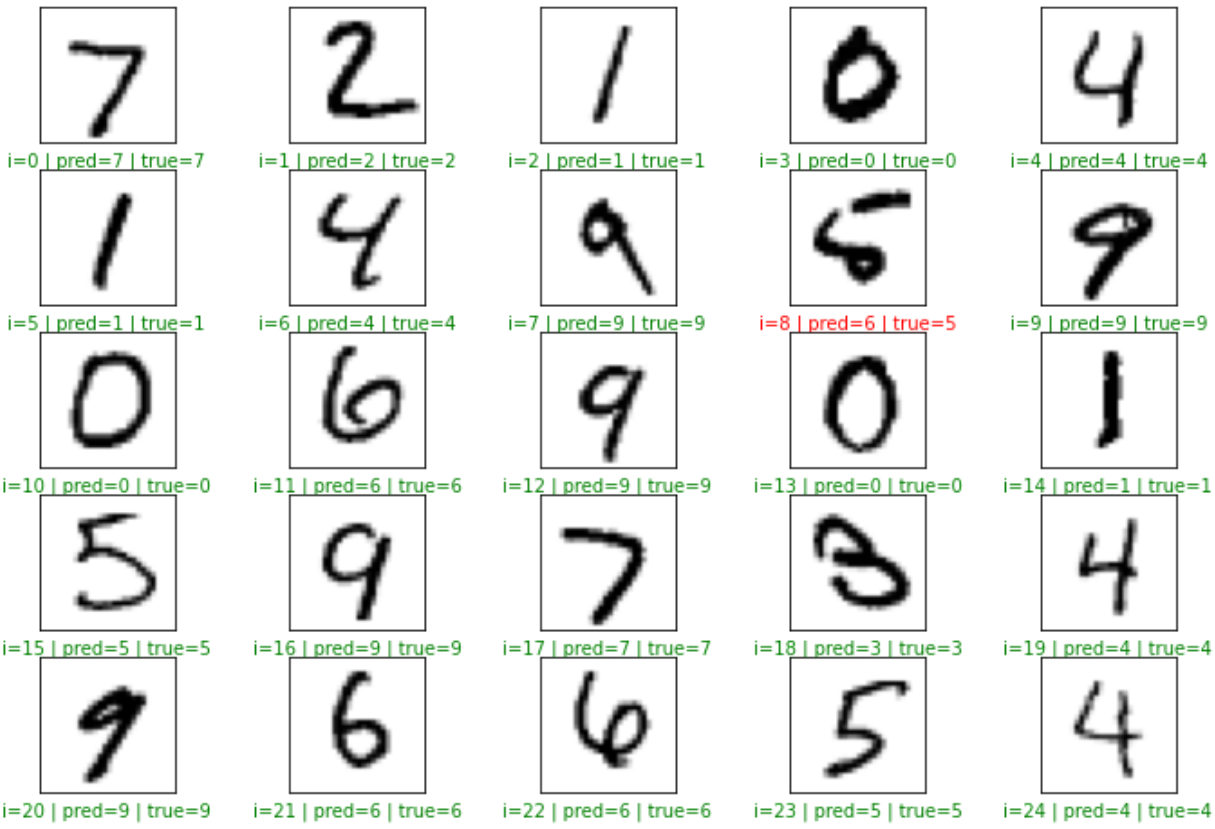
In [ ]:
```
# loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
# print('test set accuracy: ', accuracy * 100)
```

In [135…
```
preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
shape of preds:  (10000, 10)
```

As part of our model evaluation, let's look at the first 25 images by plotting the test set images along with their predicted and actual labels to understand how the trained model actually performed on specific example images.

In [136…
```
plt.figure(figsize = (12, 8))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```



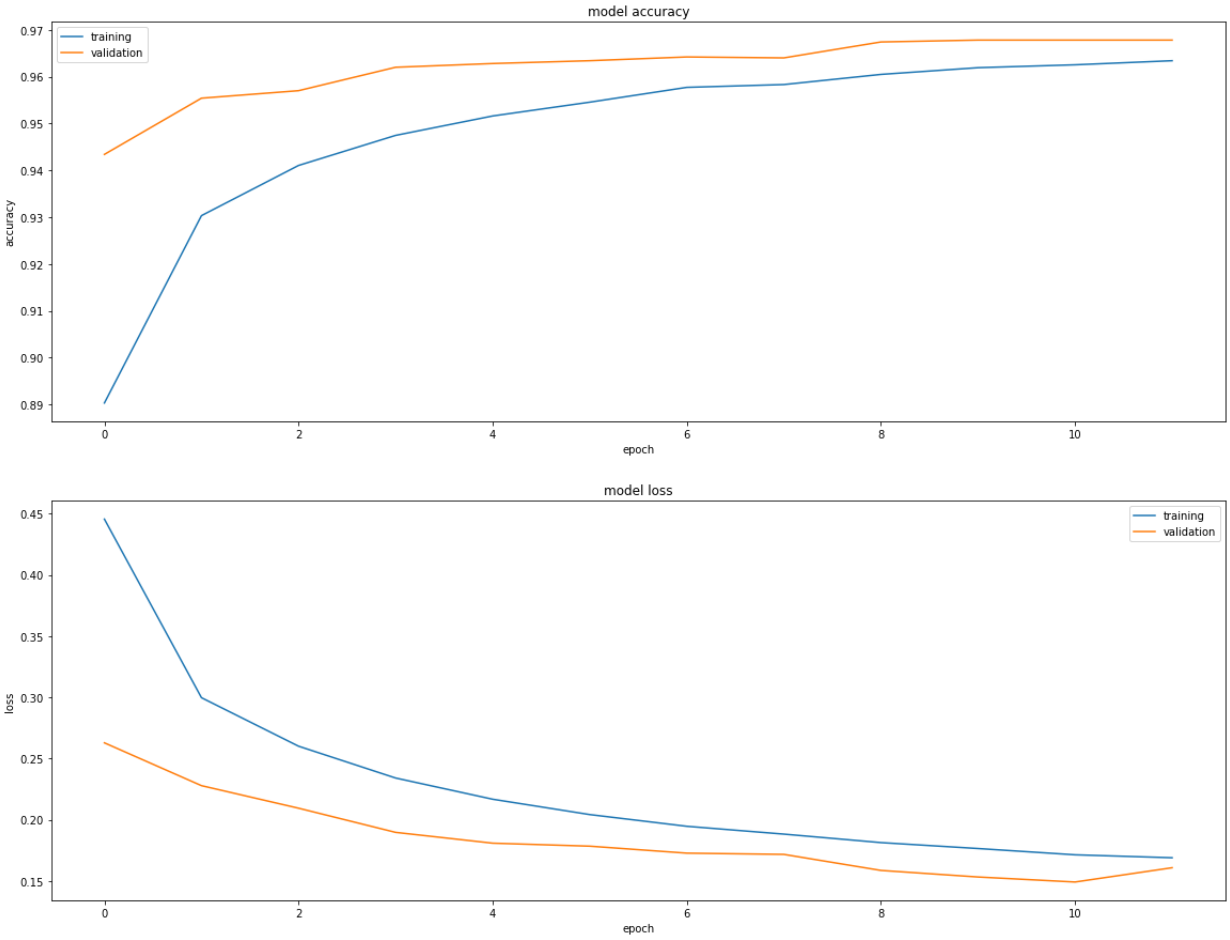
Let's use `Matplotlib` to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

In [137…
```
history_dict = history.history
history_dict.keys()
```

Out[137]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [138…
```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)
```

In [139…
```
history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)

def display_training_curves(training, validation, title, subplot):
  ax = plt.subplot(subplot)
  ax.plot(training)
  ax.plot(validation)
  ax.set_title('model '+ title)
  ax.set_ylabel(title)
  ax.set_xlabel('epoch')
  ax.legend(['training', 'validation'])

plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```



Let's examine precision and recall performance metrics for each of the prediction classes.

In [140…
```
pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)
```

```
print_validation_report(y_test, pred1)
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.96      0.99      0.98       980
           1       0.98      0.99      0.98      1135
           2       0.96      0.95      0.95      1032
           3       0.98      0.94      0.96      1010
           4       0.95      0.98      0.97       982
           5       0.96      0.96      0.96       892
           6       0.97      0.97      0.97       958
           7       0.96      0.96      0.96      1028
           8       0.95      0.97      0.96       974
           9       0.97      0.94      0.96      1009

    accuracy                           0.96     10000
   macro avg       0.96      0.96      0.96     10000
weighted avg       0.96      0.96      0.96     10000

Accuracy Score: 0.9645
Root Mean Square Error: 0.8148005891014071
```

Let's create a table that visualizes the model output for each of the first 20 images. These outputs can be thought of as the model's expression of the probability that each image corresponds to each digit class.

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

In [141…
```
# Get the predicted classes:
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning
pred_classes = np.argmax(model.predict(x_test_norm), axis=-1)
pred_classes;
```

In [142…
```
conf_mx = tf.math.confusion_matrix(y_test, pred_classes)
conf_mx;

cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8',
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

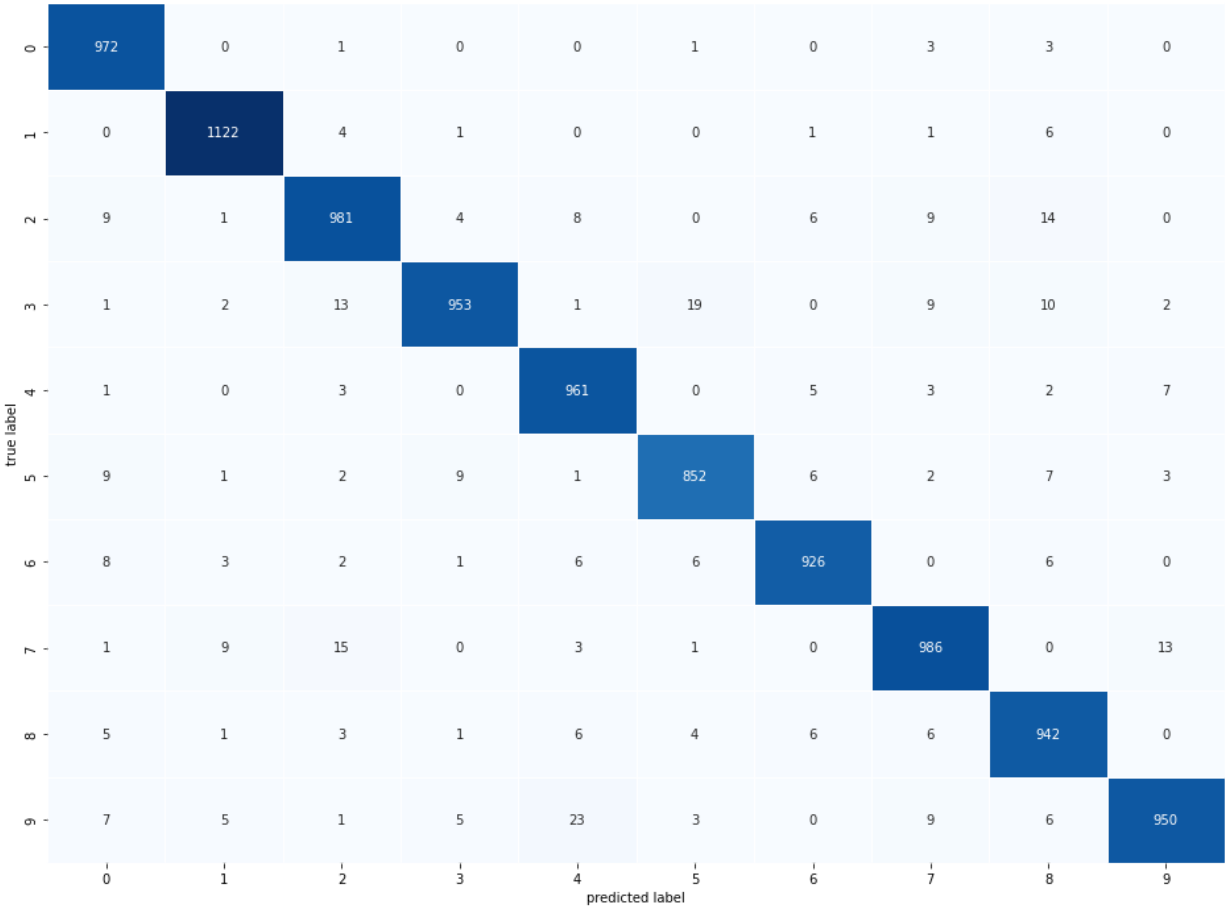
Out[142]:

	0	1	2	3	4	5	6	7	8	9
0	0.00%	0.00%	0.00%	0.14%	0.00%	0.00%	0.00%	99.81%	0.00%	0.04%
1	0.02%	0.00%	98.79%	0.67%	0.00%	0.12%	0.24%	0.00%	0.17%	0.00%
2	0.01%	99.07%	0.26%	0.10%	0.01%	0.02%	0.05%	0.10%	0.38%	0.00%
3	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
4	0.00%	0.00%	0.00%	0.00%	99.62%	0.00%	0.02%	0.01%	0.00%	0.34%
5	0.00%	99.74%	0.03%	0.04%	0.00%	0.00%	0.00%	0.12%	0.06%	0.00%
6	0.00%	0.00%	0.00%	0.00%	92.49%	0.02%	0.00%	0.29%	1.93%	5.26%
7	0.00%	0.00%	0.06%	0.24%	0.56%	0.06%	0.00%	0.03%	0.01%	99.04%
8	0.31%	0.00%	1.04%	0.00%	0.02%	1.48%	96.89%	0.00%	0.11%	0.15%
9	0.00%	0.00%	0.00%	0.00%	3.12%	0.01%	0.00%	13.11%	3.47%	80.29%
10	99.91%	0.00%	0.08%	0.00%	0.00%	0.01%	0.00%	0.00%	0.00%	0.00%
11	0.03%	0.00%	0.14%	0.00%	0.34%	0.02%	98.53%	0.00%	0.94%	0.00%
12	0.00%	0.00%	0.00%	0.08%	2.10%	0.02%	0.00%	0.20%	0.06%	97.55%
13	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
14	0.00%	99.90%	0.01%	0.02%	0.00%	0.00%	0.00%	0.00%	0.06%	0.00%
15	0.03%	0.00%	0.11%	1.95%	0.00%	97.42%	0.00%	0.01%	0.47%	0.00%
16	0.00%	0.00%	0.01%	0.00%	0.48%	0.01%	0.00%	0.43%	0.03%	99.04%
17	0.00%	0.00%	0.01%	0.34%	0.00%	0.00%	0.00%	99.57%	0.00%	0.08%
18	0.00%	0.00%	0.14%	99.32%	0.00%	0.20%	0.00%	0.00%	0.31%	0.02%
19	0.00%	0.00%	0.00%	0.00%	99.77%	0.01%	0.01%	0.02%	0.01%	0.18%

Let's create a confusion matrix that visualizes model performance on testing data.

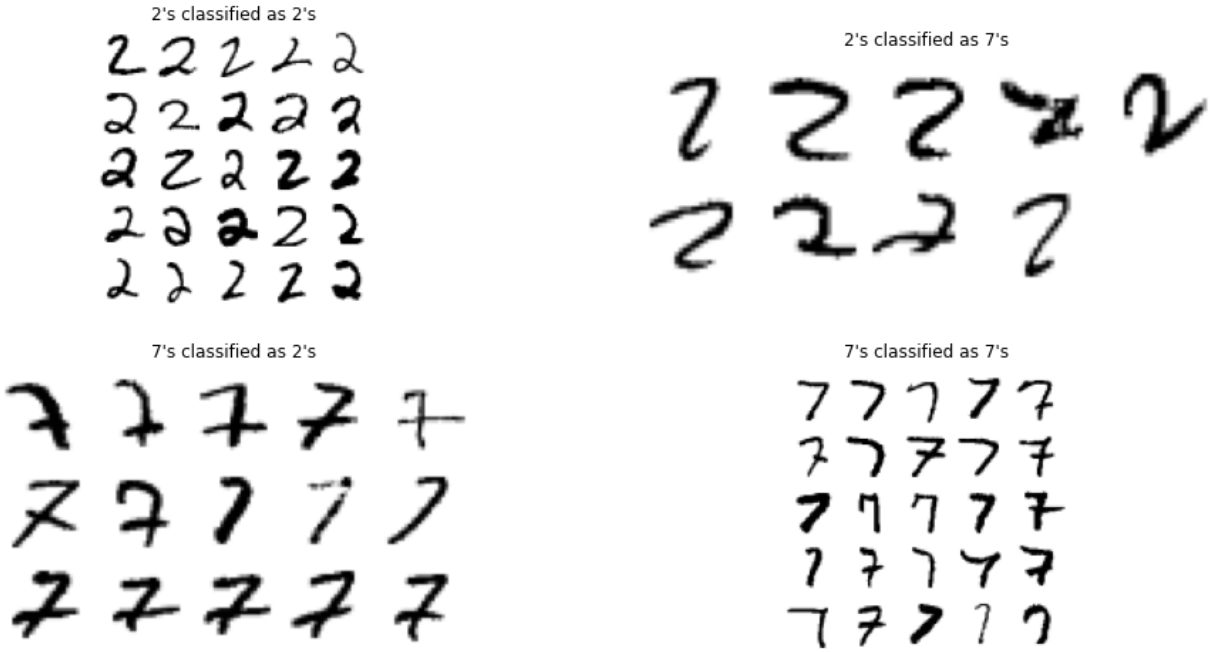
In [143…

```
plot_confusion_matrix(y_test,pred_classes)
```



In [144…

```
cl_a, cl_b = 2, 7
X_aa = x_test_norm[(y_test == cl_a) & (pred_classes == cl_a)]
X_ab = x_test_norm[(y_test == cl_a) & (pred_classes == cl_b)]
X_ba = x_test_norm[(y_test == cl_b) & (pred_classes == cl_a)]
X_bb = x_test_norm[(y_test == cl_b) & (pred_classes == cl_b)]

plt.figure(figsize=(16,8))

p1 = plt.subplot(221)
p2 = plt.subplot(222)
p3 = plt.subplot(223)
p4 = plt.subplot(224)

plot_digits(X_aa[:25], p1, images_per_row=5);
plot_digits(X_ab[:25], p2, images_per_row=5);
plot_digits(X_ba[:25], p3, images_per_row=5);
plot_digits(X_bb[:25], p4, images_per_row=5);


p1.set_title(f"{cl_a}'s classified as {cl_a}'s")
p2.set_title(f"{cl_a}'s classified as {cl_b}'s")
p3.set_title(f"{cl_b}'s classified as {cl_a}'s")
p4.set_title(f"{cl_b}'s classified as {cl_b}'s")

# plt.savefig("error_analysis_digits_plot_EXP1_valid")

plt.show()
```



7) Experiment 5 - ANN with 1 Hidden Layer with 80 Nodes

7.1) Construct Model

Let's construct our an artificial neural network to classify images as integers.

In [166…
```
tf.keras.backend.clear_session()

model = Sequential([
    Dense(input_shape=[784], units=80, activation = tf.nn.relu,kernel_regularizer=tf.k
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])
```

In [167…
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 80)                62800

 output_layer (Dense)        (None, 10)                810

=================================================================
Total params: 63,610
Trainable params: 63,610
Non-trainable params: 0
_________________________________________________________________
```

In [168…
```
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])
```

In [169…
```
#tf.keras.model.fit
#https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

#tf.keras.callbacks.EarlyStopping
#https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping


history = model.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=(5000/60000)
    ,callbacks=[tf.keras.callbacks.ModelCheckpoint("DNN_model.h5",save_best_only=True,
                ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/200
55000/55000 [==============================] - 2s 42us/sample - loss: 0.3918 - accura
cy: 0.9113 - val_loss: 0.2351 - val_accuracy: 0.9554
Epoch 2/200
55000/55000 [==============================] - 2s 43us/sample - loss: 0.2433 - accura
cy: 0.9483 - val_loss: 0.1831 - val_accuracy: 0.9684
Epoch 3/200
55000/55000 [==============================] - 4s 67us/sample - loss: 0.2042 - accura
cy: 0.9577 - val_loss: 0.1778 - val_accuracy: 0.9656
Epoch 4/200
55000/55000 [==============================] - 3s 56us/sample - loss: 0.1851 - accura
cy: 0.9614 - val_loss: 0.1655 - val_accuracy: 0.9688
Epoch 5/200
55000/55000 [==============================] - 3s 56us/sample - loss: 0.1727 - accura
cy: 0.9654 - val_loss: 0.1733 - val_accuracy: 0.9632
Epoch 6/200
55000/55000 [==============================] - 2s 40us/sample - loss: 0.1653 - accura
cy: 0.9665 - val_loss: 0.1462 - val_accuracy: 0.9708
Epoch 7/200
55000/55000 [==============================] - 2s 36us/sample - loss: 0.1593 - accura
cy: 0.9668 - val_loss: 0.1502 - val_accuracy: 0.9730
Epoch 8/200
55000/55000 [==============================] - 2s 42us/sample - loss: 0.1537 - accura
cy: 0.9690 - val_loss: 0.1466 - val_accuracy: 0.9736
Epoch 9/200
55000/55000 [==============================] - 2s 37us/sample - loss: 0.1501 - accura
cy: 0.9695 - val_loss: 0.1451 - val_accuracy: 0.9728
Epoch 10/200
55000/55000 [==============================] - 2s 37us/sample - loss: 0.1479 - accura
cy: 0.9696 - val_loss: 0.1560 - val_accuracy: 0.9704
```

7.2) Evaluate Model Performance on Testing Dataset

Let's apply the model to the testing dataset and evaluate its performance.

In [170…
```
model = tf.keras.models.load_model("DNN_model.h5")
print(f"Test acc: {model.evaluate(x_test_norm, y_test_encoded)[1]:.3f}")
```

```
Test acc: 0.965
```

In [171…
```
# loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
# print('test set accuracy: ', accuracy * 100)
```

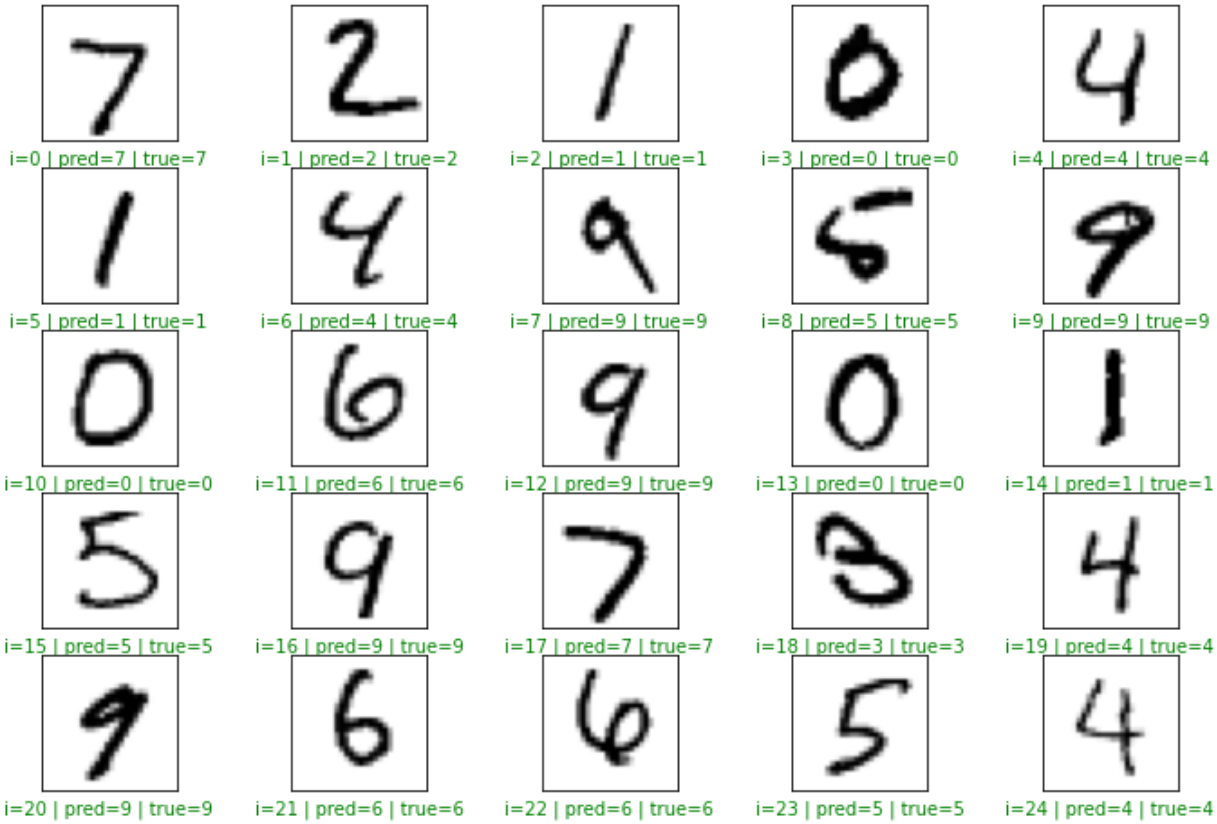
In [172…
```
preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)
```

```
shape of preds:  (10000, 10)
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
```

As part of our model evaluation, let's look at the first 25 images by plotting the test set images along with their predicted and actual labels to understand how the trained model actually performed on specific example images.

In [173…
```
plt.figure(figsize = (12, 8))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```



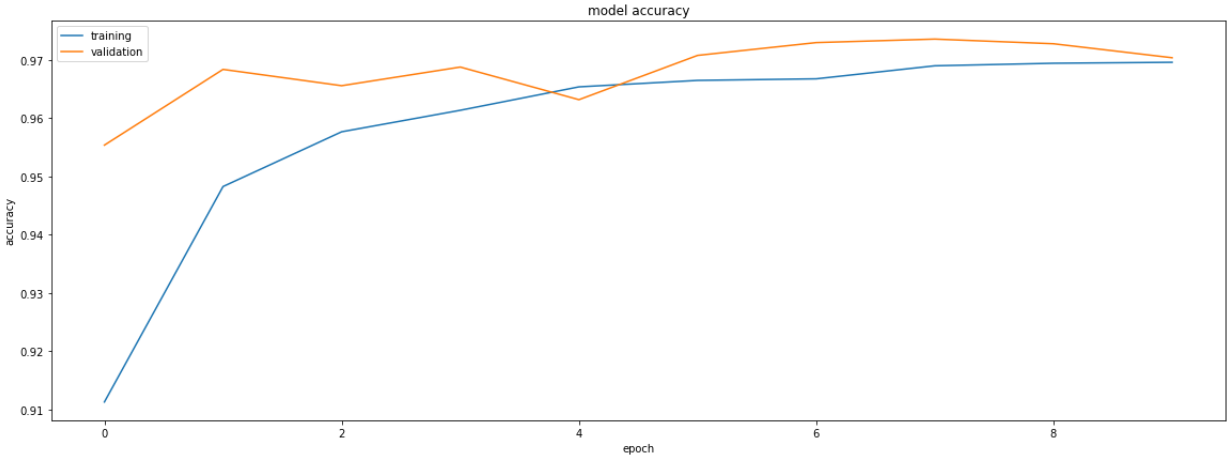
Let's use `Matplotlib` to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

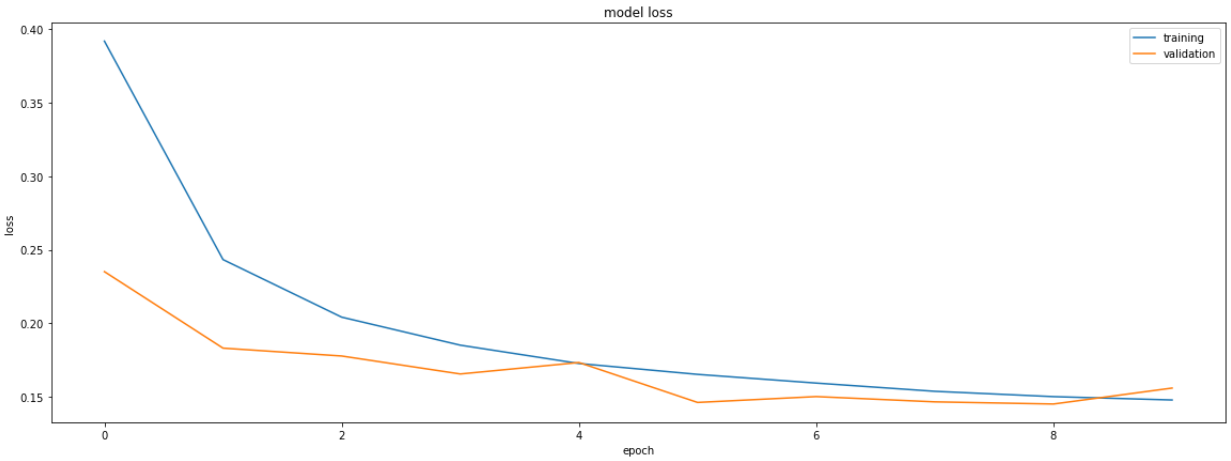
In [174…
```
history_dict = history.history
history_dict.keys()
```

Out[174]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [175…
```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)
```

In [176…
```
history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)

def display_training_curves(training, validation, title, subplot):
  ax = plt.subplot(subplot)
  ax.plot(training)
  ax.plot(validation)
  ax.set_title('model '+ title)
  ax.set_ylabel(title)
  ax.set_xlabel('epoch')
  ax.legend(['training', 'validation'])

plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```





Let's examine precision and recall performance metrics for each of the prediction classes.

In [177…

```
pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)

print_validation_report(y_test, pred1)
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.96      0.99      0.97       980
           1       0.99      0.99      0.99      1135
           2       0.99      0.93      0.96      1032
           3       0.98      0.95      0.96      1010
           4       0.95      0.98      0.96       982
           5       0.95      0.97      0.96       892
           6       0.97      0.97      0.97       958
           7       0.92      0.99      0.95      1028
           8       0.98      0.92      0.95       974
           9       0.96      0.95      0.96      1009

    accuracy                           0.96     10000
   macro avg       0.97      0.96      0.96     10000
weighted avg       0.97      0.96      0.96     10000

Accuracy Score: 0.9649
Root Mean Square Error: 0.8024961059095552
```

Let's create a table that visualizes the model output for each of the first 20 images. These outputs can be thought of as the model's expression of the probability that each image corresponds to each digit class.

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

In [178…
```
# Get the predicted classes:
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning
pred_classes = np.argmax(model.predict(x_test_norm), axis=-1)
pred_classes;
```

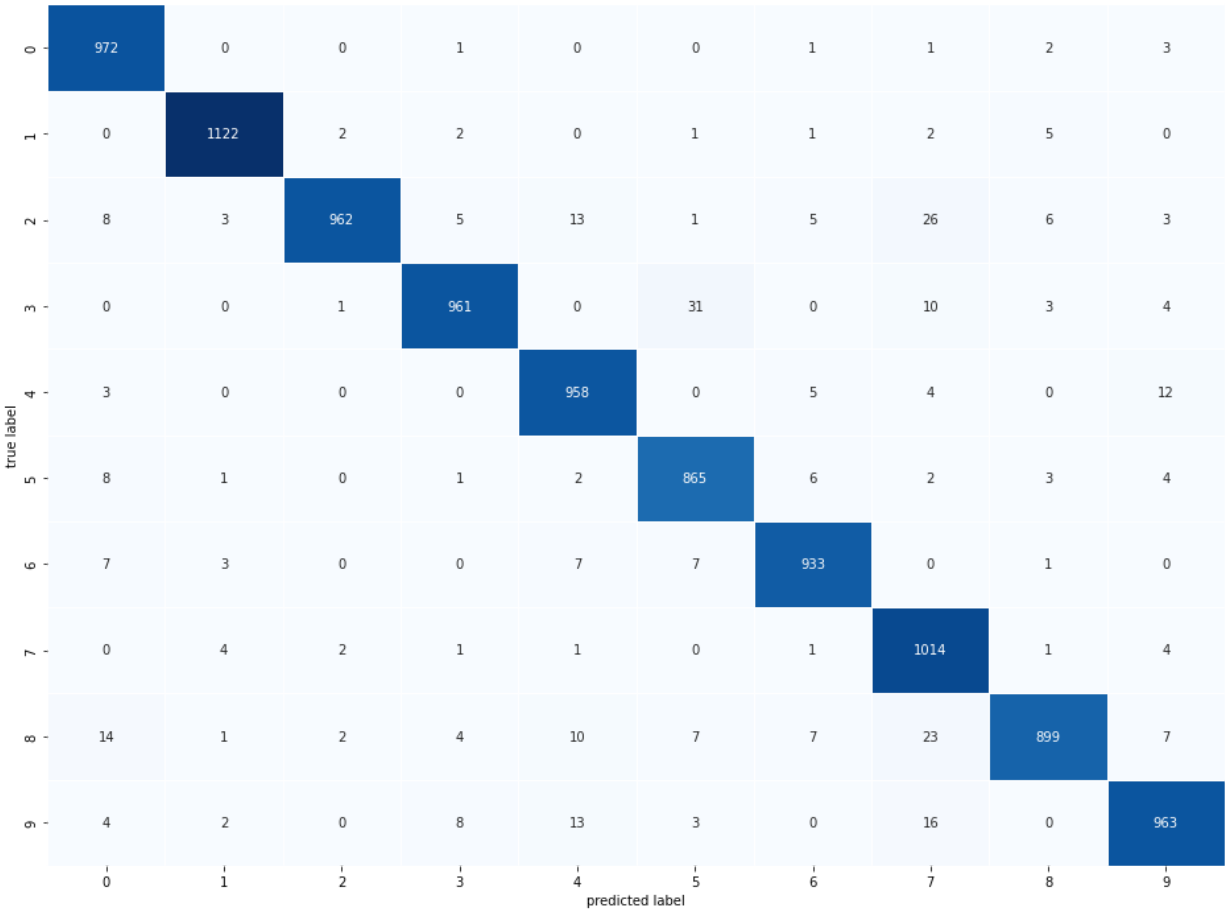
In [179…
```
conf_mx = tf.math.confusion_matrix(y_test, pred_classes)
conf_mx;

cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8',
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

Out[179]:

	0	1	2	3	4	5	6	7	8	9
0	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	99.99%	0.00%	0.00%
1	0.01%	0.06%	97.23%	2.37%	0.00%	0.30%	0.01%	0.00%	0.02%	0.00%
2	0.00%	99.48%	0.11%	0.05%	0.03%	0.01%	0.03%	0.17%	0.12%	0.01%
3	99.96%	0.00%	0.00%	0.00%	0.01%	0.01%	0.01%	0.01%	0.00%	0.00%
4	0.00%	0.00%	0.00%	0.00%	99.64%	0.00%	0.01%	0.08%	0.00%	0.27%
5	0.00%	99.81%	0.01%	0.01%	0.01%	0.00%	0.00%	0.14%	0.02%	0.00%
6	0.00%	0.00%	0.00%	0.00%	98.80%	0.00%	0.01%	0.87%	0.09%	0.23%
7	0.00%	0.00%	0.00%	1.17%	0.05%	0.00%	0.00%	0.21%	0.00%	98.57%
8	0.05%	0.00%	0.39%	0.00%	0.12%	66.34%	26.60%	0.02%	0.81%	5.67%
9	0.00%	0.00%	0.00%	0.00%	0.23%	0.00%	0.00%	1.23%	0.03%	98.51%
10	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
11	0.03%	0.00%	0.00%	0.00%	0.02%	0.02%	99.85%	0.00%	0.08%	0.00%
12	0.00%	0.00%	0.00%	0.00%	0.09%	0.00%	0.00%	0.07%	0.00%	99.84%
13	99.99%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
14	0.00%	99.82%	0.00%	0.14%	0.00%	0.00%	0.00%	0.00%	0.03%	0.00%
15	0.00%	0.00%	0.00%	0.49%	0.00%	99.48%	0.00%	0.00%	0.02%	0.00%
16	0.00%	0.00%	0.00%	0.00%	0.02%	0.00%	0.00%	0.31%	0.00%	99.67%
17	0.00%	0.00%	0.01%	0.00%	0.00%	0.00%	0.00%	99.99%	0.00%	0.00%
18	0.01%	0.00%	1.45%	96.90%	0.00%	0.03%	0.00%	0.02%	1.58%	0.01%
19	0.00%	0.00%	0.00%	0.00%	99.97%	0.00%	0.00%	0.01%	0.00%	0.02%

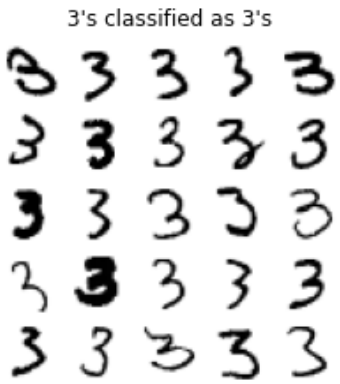
Let's create a confusion matrix that visualizes model performance on testing data.

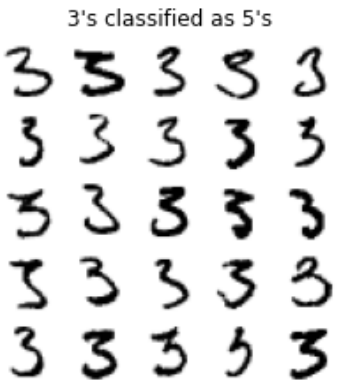
In [180…
```
plot_confusion_matrix(y_test,pred_classes)
```

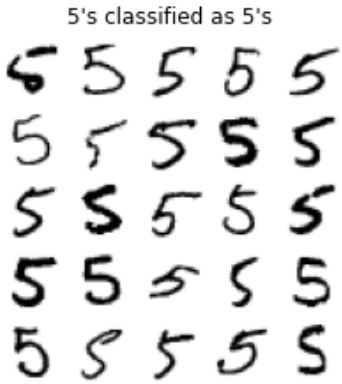


In [181…
```
cl_a, cl_b = 3, 5
X_aa = x_test_norm[(y_test == cl_a) & (pred_classes == cl_a)]
X_ab = x_test_norm[(y_test == cl_a) & (pred_classes == cl_b)]
X_ba = x_test_norm[(y_test == cl_b) & (pred_classes == cl_a)]
X_bb = x_test_norm[(y_test == cl_b) & (pred_classes == cl_b)]

plt.figure(figsize=(16,8))

p1 = plt.subplot(221)
p2 = plt.subplot(222)
p3 = plt.subplot(223)
p4 = plt.subplot(224)

plot_digits(X_aa[:25], p1, images_per_row=5);
plot_digits(X_ab[:25], p2, images_per_row=5);
plot_digits(X_ba[:25], p3, images_per_row=5);
plot_digits(X_bb[:25], p4, images_per_row=5);


p1.set_title(f"{cl_a}'s classified as {cl_a}'s")
p2.set_title(f"{cl_a}'s classified as {cl_b}'s")
p3.set_title(f"{cl_b}'s classified as {cl_a}'s")
p4.set_title(f"{cl_b}'s classified as {cl_b}'s")

# plt.savefig("error_analysis_digits_plot_EXP1_valid")
```

```
plt.show()
```







8) Experiment 6 - ANN with 1 Hidden Layer with 130 Nodes

8.1) Construct Model

Let's construct our an artificial neural network to classify images as integers.

In [191…

```
tf.keras.backend.clear_session()

model = Sequential([
    Dense(input_shape=[784], units=130, activation = tf.nn.relu,kernel_regularizer=tf.
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])
```

In [192…

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 130)               102050

 output_layer (Dense)        (None, 10)                1310

=================================================================
Total params: 103,360
Trainable params: 103,360
Non-trainable params: 0
_________________________________________________________________
```

In [193…

```
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
              metrics=['accuracy'])
```

In [194…

```
#tf.keras.model.fit
#https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

#tf.keras.callbacks.EarlyStopping
#https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping


history = model.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=(5000/60000)
    ,callbacks=[tf.keras.callbacks.ModelCheckpoint("DNN_model.h5",save_best_only=True,
                ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/200
54496/55000 [============================>.] - ETA: 0s - loss: 0.3786 - accuracy: 0.9
197
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2335: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates
```

```
55000/55000 [==============================] - 4s 75us/sample - loss: 0.3776 - accura
cy: 0.9199 - val_loss: 0.2189 - val_accuracy: 0.9600
Epoch 2/200
55000/55000 [==============================] - 3s 51us/sample - loss: 0.2257 - accura
cy: 0.9547 - val_loss: 0.1709 - val_accuracy: 0.9716
Epoch 3/200
55000/55000 [==============================] - 3s 51us/sample - loss: 0.1940 - accura
cy: 0.9609 - val_loss: 0.1802 - val_accuracy: 0.9690
Epoch 4/200
55000/55000 [==============================] - 3s 52us/sample - loss: 0.1758 - accura
cy: 0.9653 - val_loss: 0.1576 - val_accuracy: 0.9726
Epoch 5/200
55000/55000 [==============================] - 3s 47us/sample - loss: 0.1666 - accura
cy: 0.9669 - val_loss: 0.1549 - val_accuracy: 0.9694
Epoch 6/200
55000/55000 [==============================] - 3s 58us/sample - loss: 0.1591 - accura
cy: 0.9685 - val_loss: 0.1420 - val_accuracy: 0.9748
Epoch 7/200
55000/55000 [==============================] - 3s 58us/sample - loss: 0.1551 - accura
cy: 0.9697 - val_loss: 0.1445 - val_accuracy: 0.9724
Epoch 8/200
55000/55000 [==============================] - 2s 45us/sample - loss: 0.1507 - accura
cy: 0.9700 - val_loss: 0.1449 - val_accuracy: 0.9732
```

8.2) Evaluate Model Performance on Testing Dataset

Let's apply the model to the testing dataset and evaluate its performance.

In [195…
```
model = tf.keras.models.load_model("DNN_model.h5")
print(f"Test acc: {model.evaluate(x_test_norm, y_test_encoded)[1]:.3f}")
```

```
Test acc: 0.971
```

In [196…
```
# loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
# print('test set accuracy: ', accuracy * 100)
```

In [197…
```
preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)
```

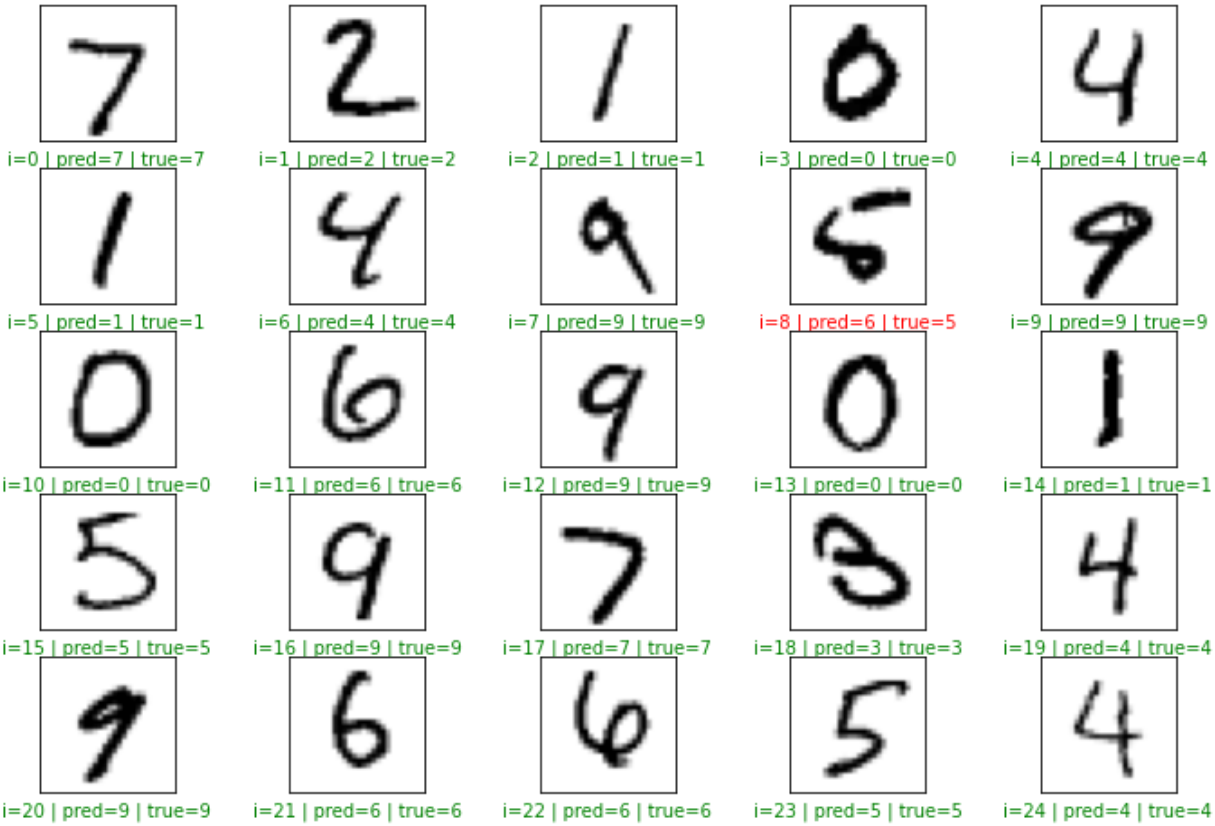
```
shape of preds:  (10000, 10)
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
```

As part of our model evaluation, let's look at the first 25 images by plotting the test set images along with their predicted and actual labels to understand how the trained model actually performed on specific example images.

In [198…
```
plt.figure(figsize = (12, 8))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
```

```
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```



Let's use `Matplotlib` to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

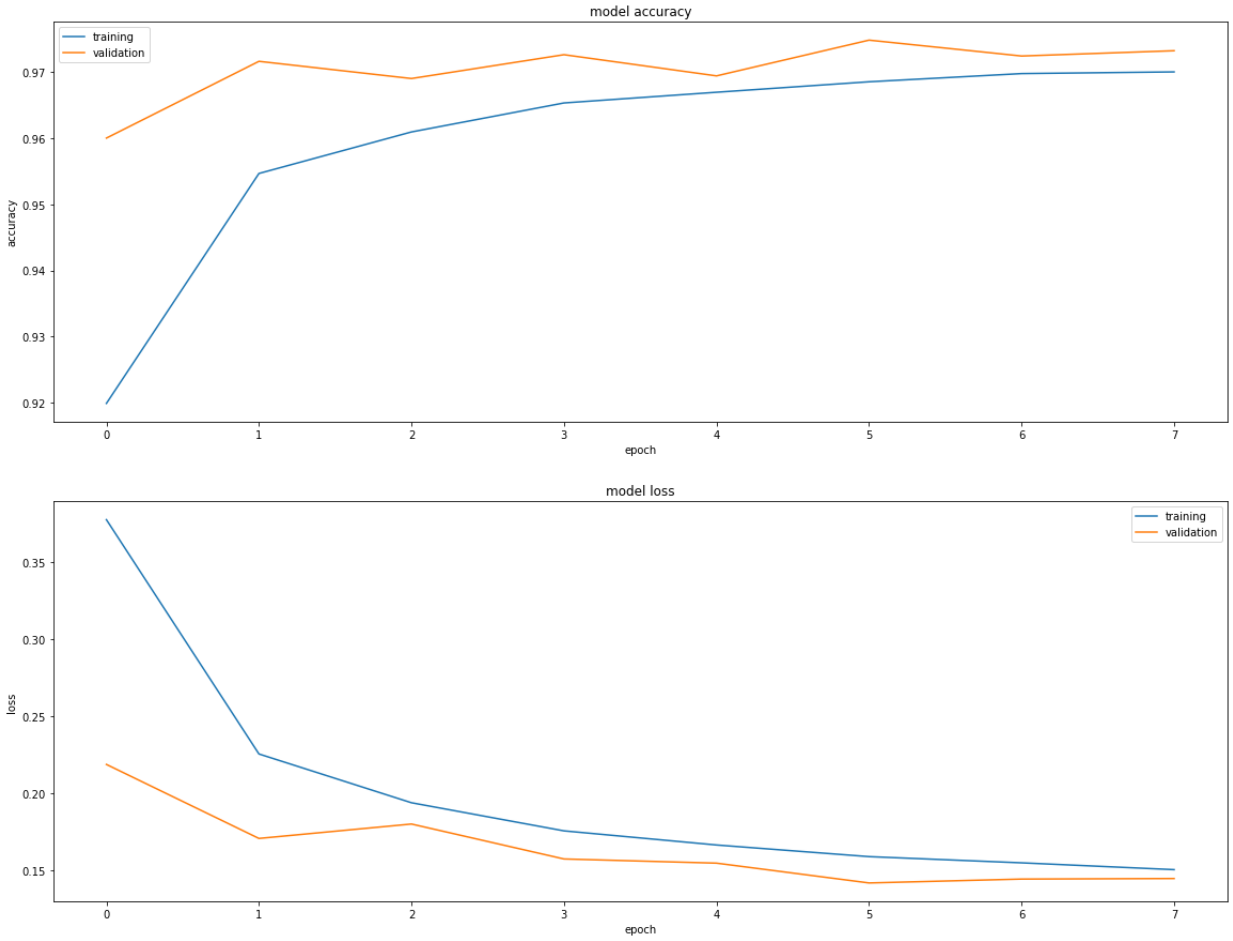
In [199…
```
history_dict = history.history
history_dict.keys()
```

Out[199]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [200…
```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)
```

In [201…
```
history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)

def display_training_curves(training, validation, title, subplot):
  ax = plt.subplot(subplot)
  ax.plot(training)
  ax.plot(validation)
```

```
  ax.set_title('model '+ title)
  ax.set_ylabel(title)
  ax.set_xlabel('epoch')
  ax.legend(['training', 'validation'])

plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```



Let's examine precision and recall performance metrics for each of the prediction classes.

In [202…

```
pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)

print_validation_report(y_test, pred1)
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.96      0.99      0.97       980
           1       0.98      0.99      0.98      1135
           2       0.96      0.98      0.97      1032
           3       0.98      0.96      0.97      1010
           4       0.98      0.95      0.97       982
           5       0.96      0.98      0.97       892
           6       0.98      0.96      0.97       958
           7       0.98      0.96      0.97      1028
           8       0.97      0.96      0.97       974
           9       0.94      0.98      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000

Accuracy Score: 0.9706
Root Mean Square Error: 0.7675936424958195
```

Let's create a table that visualizes the model output for each of the first 20 images. These outputs can be thought of as the model's expression of the probability that each image corresponds to each digit class.

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

In [203…
```
# Get the predicted classes:
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning
pred_classes = np.argmax(model.predict(x_test_norm), axis=-1)
pred_classes;
```

In [204…
```
conf_mx = tf.math.confusion_matrix(y_test, pred_classes)
conf_mx;

cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8',
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

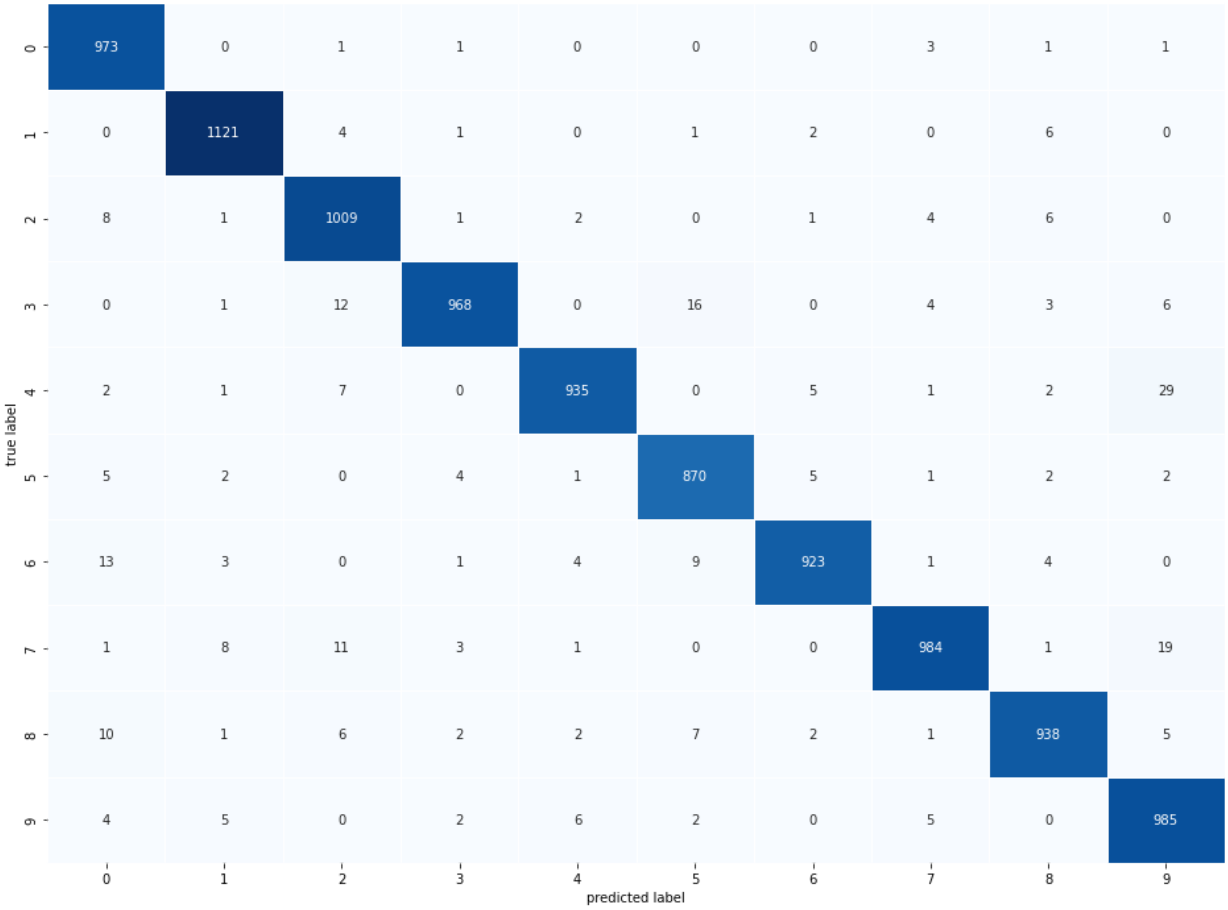
Out[204]:

	0	1	2	3	4	5	6	7	8	9
0	0.00%	0.00%	0.02%	0.05%	0.00%	0.00%	0.00%	99.92%	0.00%	0.01%
1	0.00%	0.00%	99.96%	0.03%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
2	0.00%	99.15%	0.17%	0.04%	0.01%	0.01%	0.02%	0.33%	0.27%	0.01%
3	99.91%	0.00%	0.03%	0.00%	0.00%	0.04%	0.00%	0.02%	0.00%	0.00%
4	0.01%	0.00%	0.04%	0.00%	68.79%	0.00%	0.02%	0.20%	0.01%	30.91%
5	0.00%	99.71%	0.01%	0.01%	0.00%	0.00%	0.00%	0.20%	0.06%	0.00%
6	0.00%	0.00%	0.00%	0.00%	99.18%	0.01%	0.00%	0.04%	0.48%	0.29%
7	0.00%	0.00%	0.02%	0.03%	0.06%	0.06%	0.00%	0.03%	0.01%	99.79%
8	0.09%	0.00%	3.38%	0.00%	0.02%	28.77%	66.84%	0.00%	0.13%	0.76%
9	0.00%	0.00%	0.00%	0.00%	0.61%	0.00%	0.00%	0.04%	0.31%	99.03%
10	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
11	0.37%	0.00%	0.02%	0.00%	0.00%	0.27%	99.00%	0.00%	0.33%	0.00%
12	0.00%	0.00%	0.00%	0.03%	0.51%	0.01%	0.00%	0.03%	0.01%	99.41%
13	99.98%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%	0.00%
14	0.00%	99.98%	0.00%	0.01%	0.00%	0.00%	0.00%	0.00%	0.02%	0.00%
15	0.00%	0.00%	0.01%	0.27%	0.00%	99.63%	0.00%	0.00%	0.09%	0.00%
16	0.00%	0.00%	0.00%	0.00%	0.03%	0.00%	0.00%	0.03%	0.00%	99.93%
17	0.00%	0.00%	0.01%	0.01%	0.00%	0.00%	0.00%	99.97%	0.00%	0.00%
18	0.08%	0.00%	6.17%	91.52%	0.00%	0.21%	0.00%	0.02%	1.89%	0.11%
19	0.00%	0.00%	0.00%	0.00%	99.96%	0.00%	0.00%	0.00%	0.00%	0.04%

Let's create a confusion matrix that visualizes model performance on testing data.

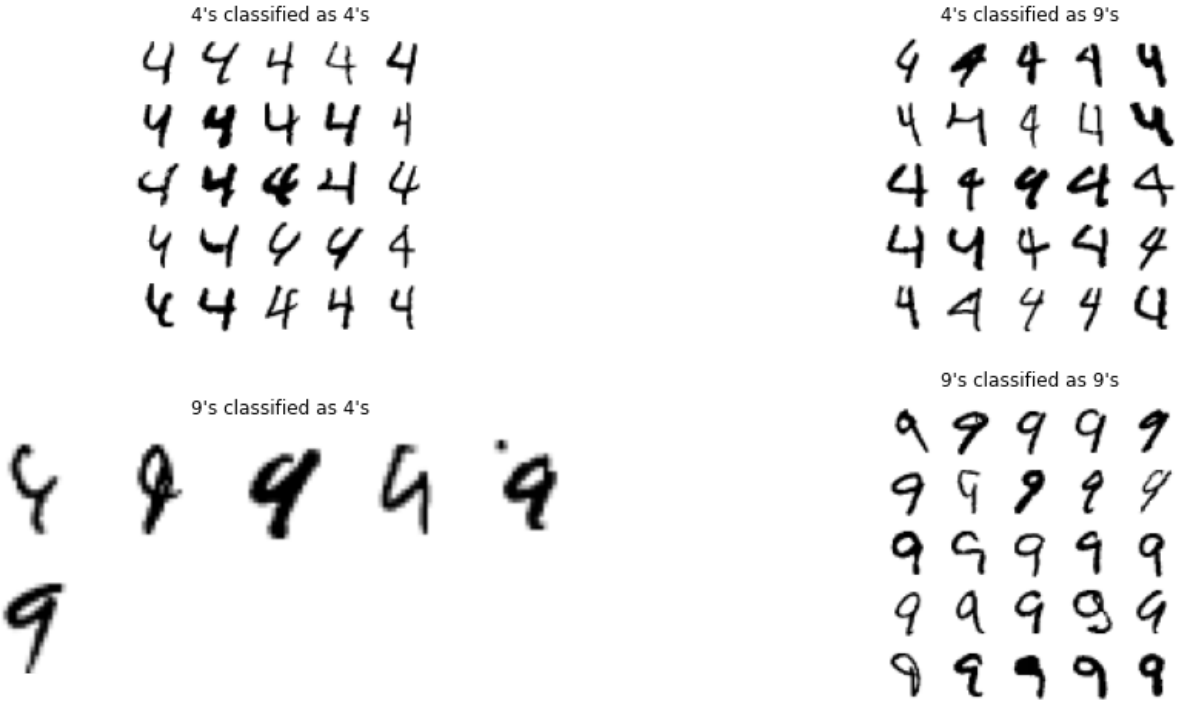
In [205…

```
plot_confusion_matrix(y_test,pred_classes)
```



In [206…

```
cl_a, cl_b = 4, 9
X_aa = x_test_norm[(y_test == cl_a) & (pred_classes == cl_a)]
X_ab = x_test_norm[(y_test == cl_a) & (pred_classes == cl_b)]
X_ba = x_test_norm[(y_test == cl_b) & (pred_classes == cl_a)]
X_bb = x_test_norm[(y_test == cl_b) & (pred_classes == cl_b)]

plt.figure(figsize=(16,8))

p1 = plt.subplot(221)
p2 = plt.subplot(222)
p3 = plt.subplot(223)
p4 = plt.subplot(224)

plot_digits(X_aa[:25], p1, images_per_row=5);
plot_digits(X_ab[:25], p2, images_per_row=5);
plot_digits(X_ba[:25], p3, images_per_row=5);
plot_digits(X_bb[:25], p4, images_per_row=5);


p1.set_title(f"{cl_a}'s classified as {cl_a}'s")
p2.set_title(f"{cl_a}'s classified as {cl_b}'s")
p3.set_title(f"{cl_b}'s classified as {cl_a}'s")
p4.set_title(f"{cl_b}'s classified as {cl_b}'s")

# plt.savefig("error_analysis_digits_plot_EXP1_valid")

plt.show()
```



9) Experiment 7 - ANN with 1 Hidden Layer with 200 Nodes

9.1) Construct Model

Let's construct our an artificial neural network to classify images as integers.

In [215…
```
tf.keras.backend.clear_session()

model = Sequential([
    Dense(input_shape=[784], units=200, activation = tf.nn.relu,kernel_regularizer=tf.
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])
```

In [216…
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 200)               157000

 output_layer (Dense)        (None, 10)                2010

=================================================================
Total params: 159,010
Trainable params: 159,010
Non-trainable params: 0
_________________________________________________________________
```

In [217…
```
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
```

```
                metrics=['accuracy'])
```

In [218…

```
#tf.keras.model.fit
#https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

#tf.keras.callbacks.EarlyStopping
#https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping


history = model.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=(5000/60000)
    ,callbacks=[tf.keras.callbacks.ModelCheckpoint("DNN_model.h5",save_best_only=True,
                ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/200
55000/55000 [==============================] - 3s 58us/sample - loss: 0.3901 - accura
cy: 0.9195 - val_loss: 0.2110 - val_accuracy: 0.9666
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2335: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates
Epoch 2/200
55000/55000 [==============================] - 3s 52us/sample - loss: 0.2266 - accura
cy: 0.9548 - val_loss: 0.1837 - val_accuracy: 0.9662
Epoch 3/200
55000/55000 [==============================] - 4s 66us/sample - loss: 0.1951 - accura
cy: 0.9609 - val_loss: 0.1659 - val_accuracy: 0.9700
Epoch 4/200
55000/55000 [==============================] - 4s 64us/sample - loss: 0.1797 - accura
cy: 0.9645 - val_loss: 0.1684 - val_accuracy: 0.9702
Epoch 5/200
55000/55000 [==============================] - 3s 62us/sample - loss: 0.1703 - accura
cy: 0.9663 - val_loss: 0.1627 - val_accuracy: 0.9694
Epoch 6/200
55000/55000 [==============================] - 3s 55us/sample - loss: 0.1640 - accura
cy: 0.9679 - val_loss: 0.1910 - val_accuracy: 0.9628
```

9.2) Evaluate Model Performance on Testing Dataset

Let's apply this model to the testing dataset and evaluate its performance.

In [219…

```
model = tf.keras.models.load_model("DNN_model.h5")
print(f"Test acc: {model.evaluate(x_test_norm, y_test_encoded)[1]:.3f}")
```

```
Test acc: 0.965
```

In [220…

```
# loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
# print('test set accuracy: ', accuracy * 100)
```

In [221…

```
preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)
```

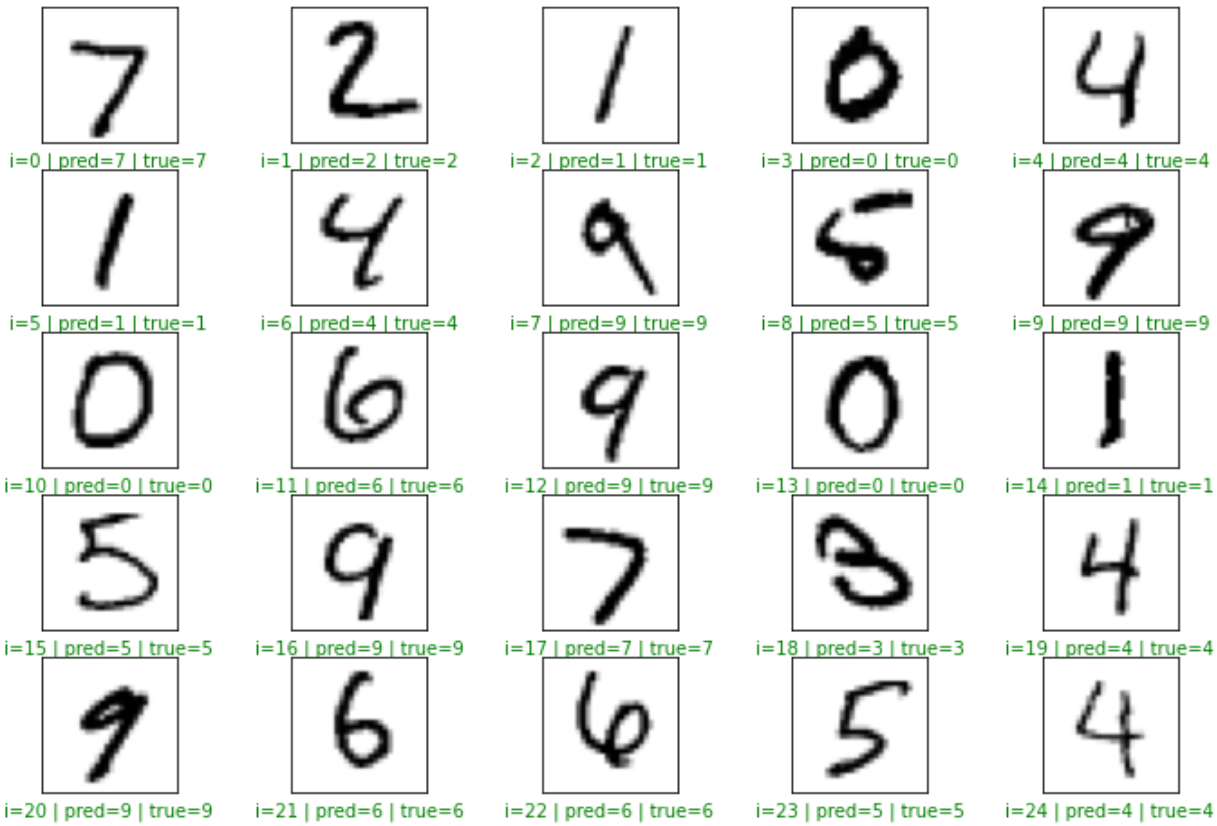
```
shape of preds:  (10000, 10)
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
```

As part of our model evaluation, let's look at the first 25 images by plotting the test set images along with their predicted and actual labels to understand how the trained model actually performed on specific example images.

In [222…

```
plt.figure(figsize = (12, 8))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```



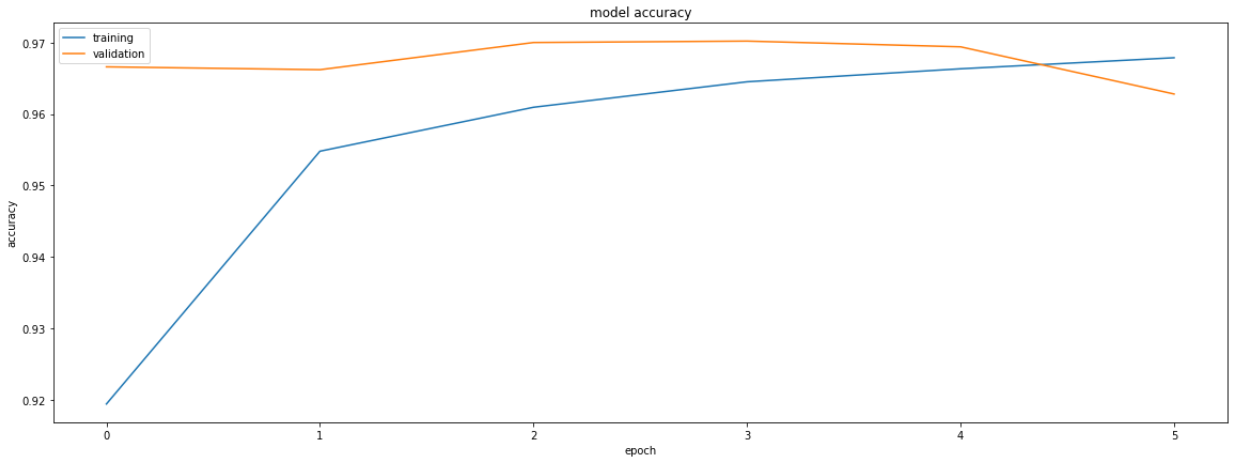
Let's use `Matplotlib` to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

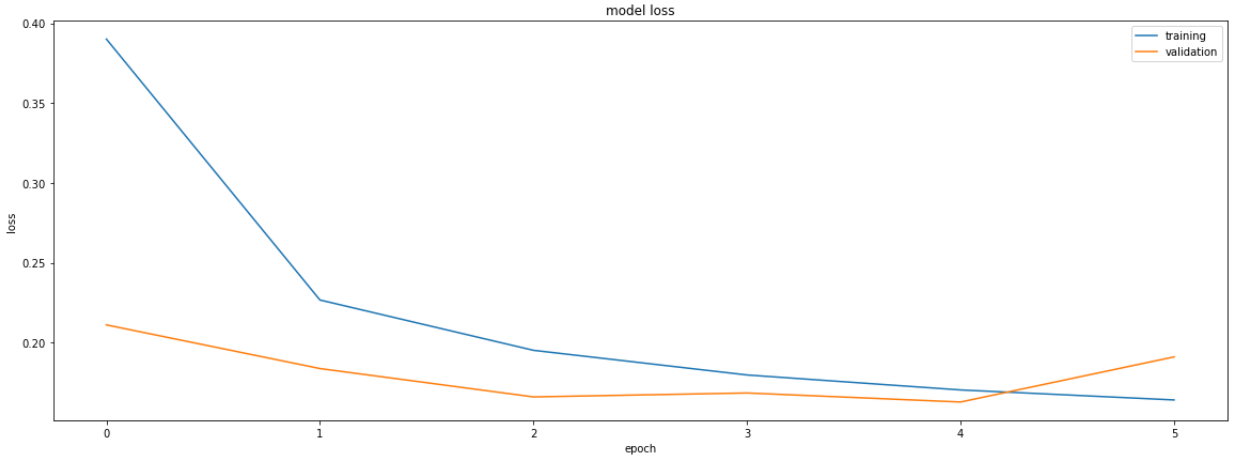
In [223…
```
history_dict = history.history
history_dict.keys()
```

Out[223]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [224…
```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)
```

In [225…
```
history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)

def display_training_curves(training, validation, title, subplot):
  ax = plt.subplot(subplot)
  ax.plot(training)
  ax.plot(validation)
  ax.set_title('model '+ title)
  ax.set_ylabel(title)
  ax.set_xlabel('epoch')
  ax.legend(['training', 'validation'])

plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```





Let's examine precision and recall performance metrics for each of the prediction classes.

In [226…
```
pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)

print_validation_report(y_test, pred1)
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.96      0.99      0.97       980
           1       0.99      0.98      0.99      1135
           2       0.94      0.98      0.96      1032
           3       0.98      0.94      0.96      1010
           4       0.98      0.96      0.97       982
           5       0.92      0.98      0.95       892
           6       0.98      0.96      0.97       958
           7       0.98      0.96      0.97      1028
           8       0.97      0.92      0.94       974
           9       0.95      0.97      0.96      1009

    accuracy                           0.96     10000
   macro avg       0.96      0.96      0.96     10000
weighted avg       0.97      0.96      0.96     10000

Accuracy Score: 0.9648
Root Mean Square Error: 0.8408923831264021
```

Let's create a table that visualizes the model output for each of the first 20 images. These outputs can be thought of as the model's expression of the probability that each image corresponds to each digit class.

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

In [227…
```
# Get the predicted classes:
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning
pred_classes = np.argmax(model.predict(x_test_norm), axis=-1)
pred_classes;
```

In [228…
```
conf_mx = tf.math.confusion_matrix(y_test, pred_classes)
conf_mx;

cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8',
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

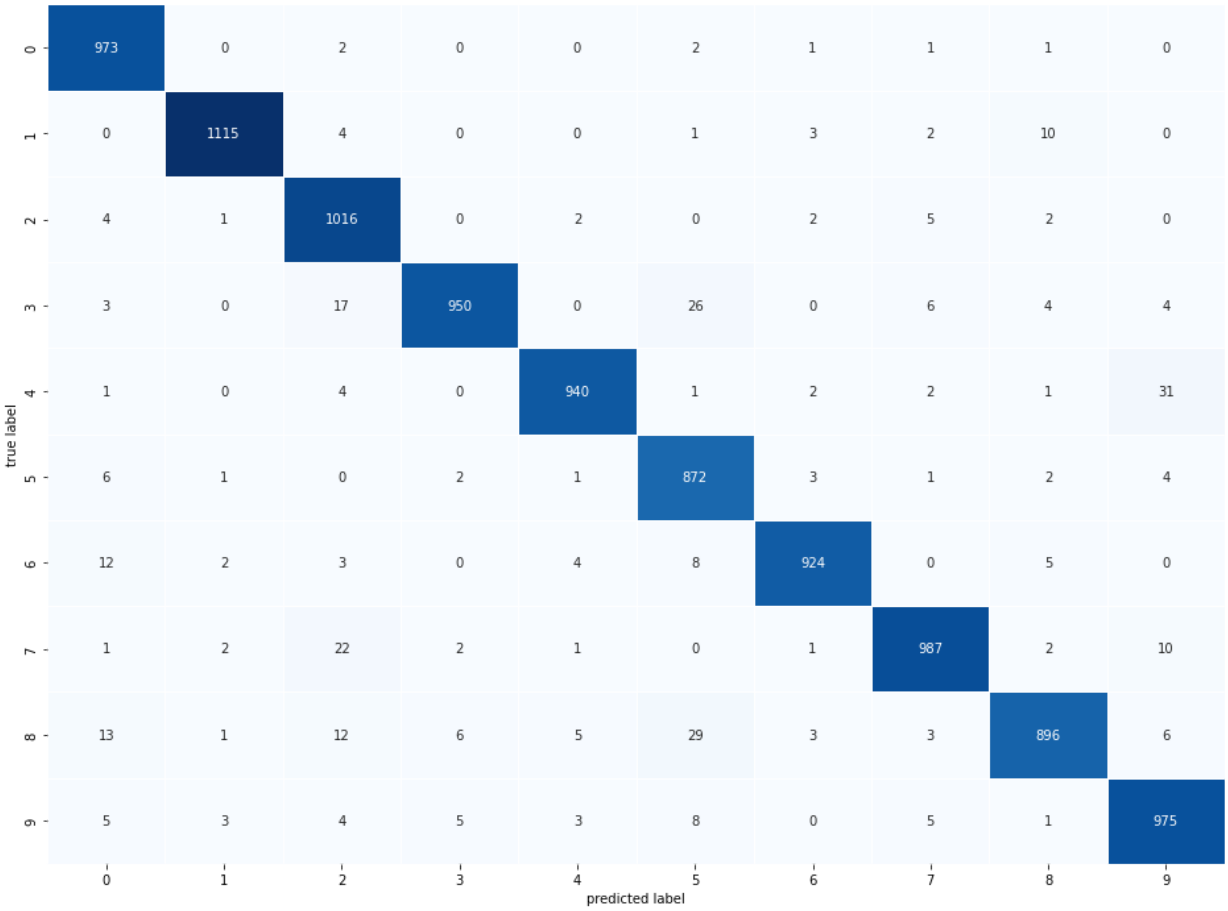
Out[228]:

	0	1	2	3	4	5	6	7	8	9
0	0.00%	0.00%	0.01%	0.01%	0.00%	0.00%	0.00%	99.98%	0.00%	0.00%
1	0.07%	0.01%	99.80%	0.09%	0.00%	0.00%	0.02%	0.00%	0.00%	0.00%
2	0.00%	99.23%	0.28%	0.02%	0.01%	0.02%	0.02%	0.24%	0.18%	0.01%
3	99.97%	0.00%	0.03%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
4	0.00%	0.00%	0.00%	0.00%	99.72%	0.00%	0.00%	0.01%	0.00%	0.26%
5	0.00%	99.50%	0.04%	0.01%	0.00%	0.00%	0.00%	0.39%	0.06%	0.00%
6	0.00%	0.00%	0.00%	0.00%	99.67%	0.00%	0.00%	0.07%	0.12%	0.13%
7	0.00%	0.00%	0.04%	0.62%	0.14%	0.01%	0.00%	0.07%	0.00%	99.11%
8	0.04%	0.00%	0.94%	0.00%	0.01%	71.28%	27.56%	0.00%	0.04%	0.14%
9	0.00%	0.00%	0.00%	0.00%	0.46%	0.00%	0.00%	0.03%	0.02%	99.48%
10	99.98%	0.00%	0.02%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
11	0.12%	0.00%	0.00%	0.00%	0.00%	0.27%	99.54%	0.00%	0.07%	0.00%
12	0.00%	0.00%	0.00%	0.01%	0.07%	0.00%	0.00%	0.04%	0.00%	99.88%
13	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
14	0.00%	99.94%	0.00%	0.03%	0.00%	0.00%	0.00%	0.00%	0.02%	0.00%
15	0.02%	0.00%	0.04%	1.89%	0.00%	98.01%	0.00%	0.00%	0.03%	0.00%
16	0.00%	0.00%	0.03%	0.00%	0.16%	0.00%	0.00%	0.21%	0.01%	99.59%
17	0.00%	0.00%	0.01%	0.00%	0.00%	0.00%	0.00%	99.99%	0.00%	0.00%
18	1.03%	0.00%	0.87%	93.29%	0.00%	0.69%	0.09%	0.00%	4.01%	0.02%
19	0.00%	0.00%	0.01%	0.00%	99.56%	0.00%	0.00%	0.05%	0.00%	0.39%

Let's create a confusion matrix that visualizes model performance on testing data.

In [229…

```
plot_confusion_matrix(y_test,pred_classes)
```



In [230…

```
cl_a, cl_b = 4, 9
X_aa = x_test_norm[(y_test == cl_a) & (pred_classes == cl_a)]
X_ab = x_test_norm[(y_test == cl_a) & (pred_classes == cl_b)]
X_ba = x_test_norm[(y_test == cl_b) & (pred_classes == cl_a)]
X_bb = x_test_norm[(y_test == cl_b) & (pred_classes == cl_b)]

plt.figure(figsize=(16,8))

p1 = plt.subplot(221)
p2 = plt.subplot(222)
p3 = plt.subplot(223)
p4 = plt.subplot(224)

plot_digits(X_aa[:25], p1, images_per_row=5);
plot_digits(X_ab[:25], p2, images_per_row=5);
plot_digits(X_ba[:25], p3, images_per_row=5);
plot_digits(X_bb[:25], p4, images_per_row=5);


p1.set_title(f"{cl_a}'s classified as {cl_a}'s")
p2.set_title(f"{cl_a}'s classified as {cl_b}'s")
p3.set_title(f"{cl_b}'s classified as {cl_a}'s")
p4.set_title(f"{cl_b}'s classified as {cl_b}'s")

# plt.savefig("error_analysis_digits_plot_EXP1_valid")

plt.show()
```

10) Experiment 8 - ANN with 1 Hidden Layer with __ Nodes

10.1) Construct Model

Let's construct our an artificial neural network to classify images as integers.

In [239…
```
tf.keras.backend.clear_session()

model = Sequential([
    Dense(input_shape=[784], units=500, activation = tf.nn.relu,kernel_regularizer=tf.
    Dense(name = "output_layer", units = 10, activation = tf.nn.softmax)
])
```

In [240…
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 500)               392500

 output_layer (Dense)        (None, 10)                5010

=================================================================
Total params: 397,510
Trainable params: 397,510
Non-trainable params: 0
_________________________________________________________________
```

In [241…
```
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
```

```
                metrics=['accuracy'])
```

In [242…

```
#tf.keras.model.fit
#https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

#tf.keras.callbacks.EarlyStopping
#https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping


history = model.fit(
    x_train_norm
    ,y_train_encoded
    ,epochs = 200
    ,validation_split=(5000/60000)
    ,callbacks=[tf.keras.callbacks.ModelCheckpoint("DNN_model.h5",save_best_only=True,
                ,tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=2)]
    )
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/200
54784/55000 [============================>.] - ETA: 0s - loss: 0.3957 - accuracy: 0.9
263
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2335: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates
55000/55000 [==============================] - 9s 169us/sample - loss: 0.3952 - accur
acy: 0.9264 - val_loss: 0.2152 - val_accuracy: 0.9656
Epoch 2/200
55000/55000 [==============================] - 9s 171us/sample - loss: 0.2324 - accur
acy: 0.9551 - val_loss: 0.1893 - val_accuracy: 0.9650
Epoch 3/200
55000/55000 [==============================] - 10s 178us/sample - loss: 0.2121 - accu
racy: 0.9576 - val_loss: 0.1765 - val_accuracy: 0.9710
Epoch 4/200
55000/55000 [==============================] - 10s 174us/sample - loss: 0.1997 - accu
racy: 0.9618 - val_loss: 0.1896 - val_accuracy: 0.9608
Epoch 5/200
55000/55000 [==============================] - 9s 167us/sample - loss: 0.1911 - accur
acy: 0.9628 - val_loss: 0.1719 - val_accuracy: 0.9694
```

10.2) Evaluate Model Performance on Testing Dataset

Let's apply this model to the testing dataset and evaluate its performance.

In [243…

```
model = tf.keras.models.load_model("DNN_model.h5")
print(f"Test acc: {model.evaluate(x_test_norm, y_test_encoded)[1]:.3f}")
```

```
Test acc: 0.965
```

In [244…

```
# loss, accuracy = model.evaluate(x_test_norm, y_test_encoded)
# print('test set accuracy: ', accuracy * 100)
```

In [245…

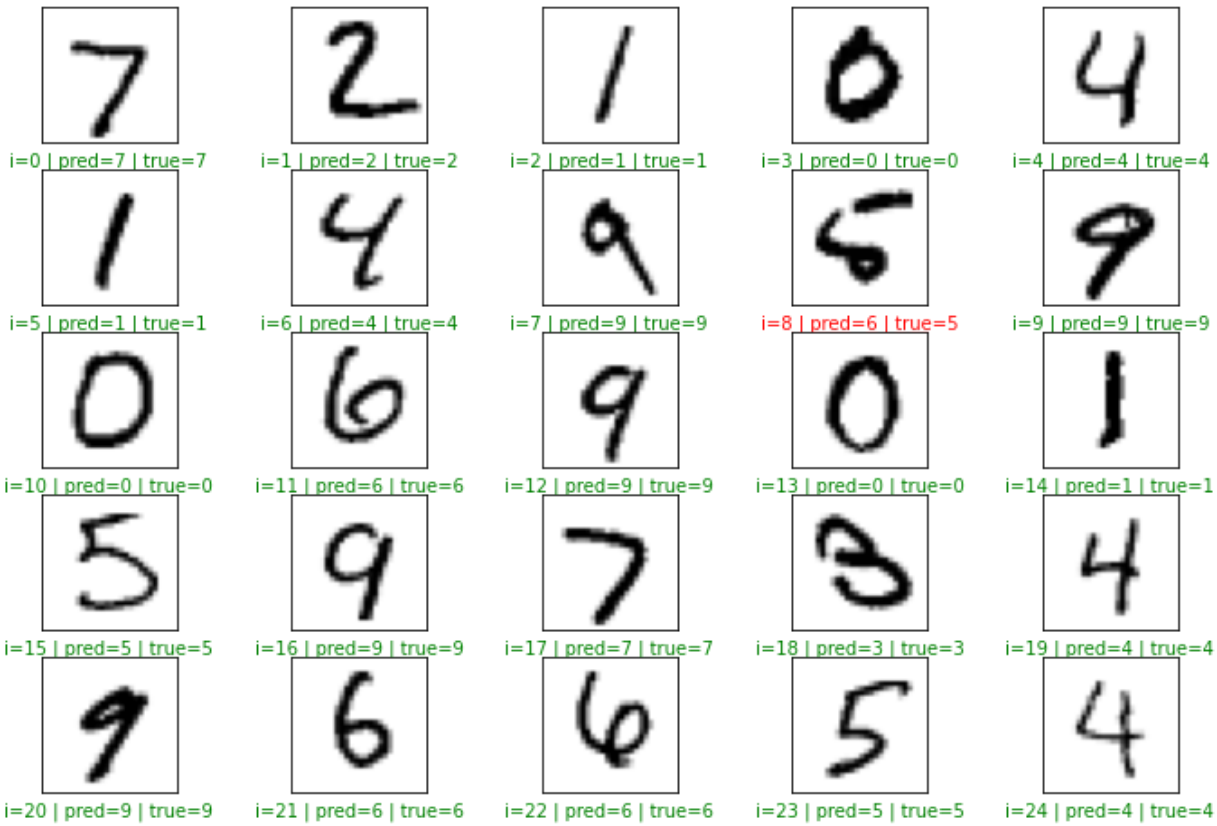
```
preds = model.predict(x_test_norm)
print('shape of preds: ', preds.shape)
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2359: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates=self.state_updates,
shape of preds:  (10000, 10)
```

As part of our model evaluation, let's look at the first 25 images by plotting the test set images along with their predicted and actual labels to understand how the trained model actually performed on specific example images.

In [246…

```
plt.figure(figsize = (12, 8))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```



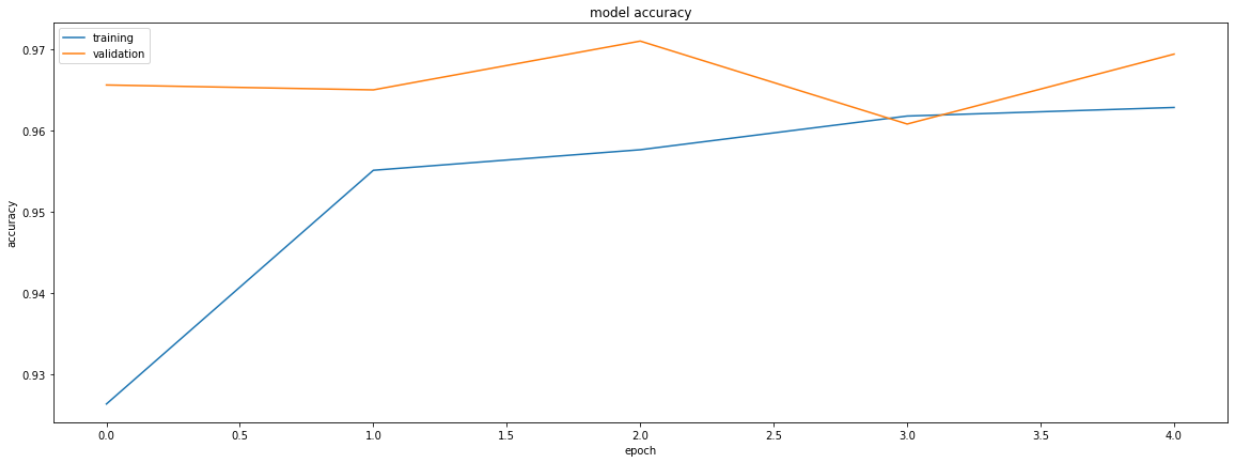
Let's use `Matplotlib` to create 2 plots--displaying the training and validation loss (resp. accuracy) for each (training) epoch side by side.

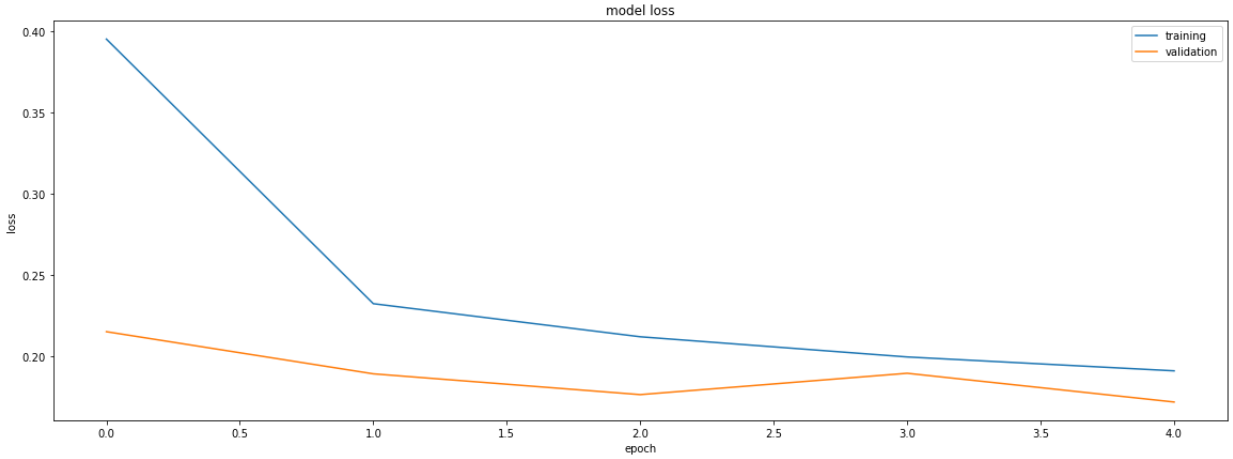
In [247…
```
history_dict = history.history
history_dict.keys()
```

Out[247]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [248…
```
losses = history.history['loss']
accs = history.history['accuracy']
val_losses = history.history['val_loss']
val_accs = history.history['val_accuracy']
epochs = len(losses)
```

In [249…
```
history_df=pd.DataFrame(history_dict)
history_df.tail().round(3)

def display_training_curves(training, validation, title, subplot):
  ax = plt.subplot(subplot)
  ax.plot(training)
  ax.plot(validation)
  ax.set_title('model '+ title)
  ax.set_ylabel(title)
  ax.set_xlabel('epoch')
  ax.legend(['training', 'validation'])

plt.subplots(figsize=(16,12))
plt.tight_layout()
display_training_curves(history.history['accuracy'], history.history['val_accuracy'],
display_training_curves(history.history['loss'], history.history['val_loss'], 'loss',
```





Let's examine precision and recall performance metrics for each of the prediction classes.

In [250…
```
pred1= model.predict(x_test_norm)
pred1=np.argmax(pred1, axis=1)

print_validation_report(y_test, pred1)
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.96      0.99      0.98       980
           1       0.98      0.99      0.99      1135
           2       0.94      0.97      0.96      1032
           3       0.97      0.94      0.96      1010
           4       0.98      0.96      0.97       982
           5       0.91      0.98      0.95       892
           6       0.98      0.97      0.97       958
           7       0.97      0.96      0.96      1028
           8       0.97      0.96      0.96       974
           9       0.98      0.93      0.95      1009

    accuracy                           0.97     10000
   macro avg       0.96      0.97      0.96     10000
weighted avg       0.97      0.97      0.97     10000

Accuracy Score: 0.9651
Root Mean Square Error: 0.809814793641114
```

Let's create a table that visualizes the model output for each of the first 20 images. These outputs can be thought of as the model's expression of the probability that each image corresponds to each digit class.

Correlation matrix that measures the linear relationships
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

In [251…
```
# Get the predicted classes:
# pred_classes = model.predict_classes(x_train_norm)# give deprecation warning
pred_classes = np.argmax(model.predict(x_test_norm), axis=-1)
pred_classes;
```

In [252…
```
conf_mx = tf.math.confusion_matrix(y_test, pred_classes)
conf_mx;

cm = sns.light_palette((260, 75, 60), input="husl", as_cmap=True)
df = pd.DataFrame(preds[0:20], columns = ['0', '1', '2', '3', '4', '5', '6', '7', '8',
df.style.format("{:.2%}").background_gradient(cmap=cm)
```

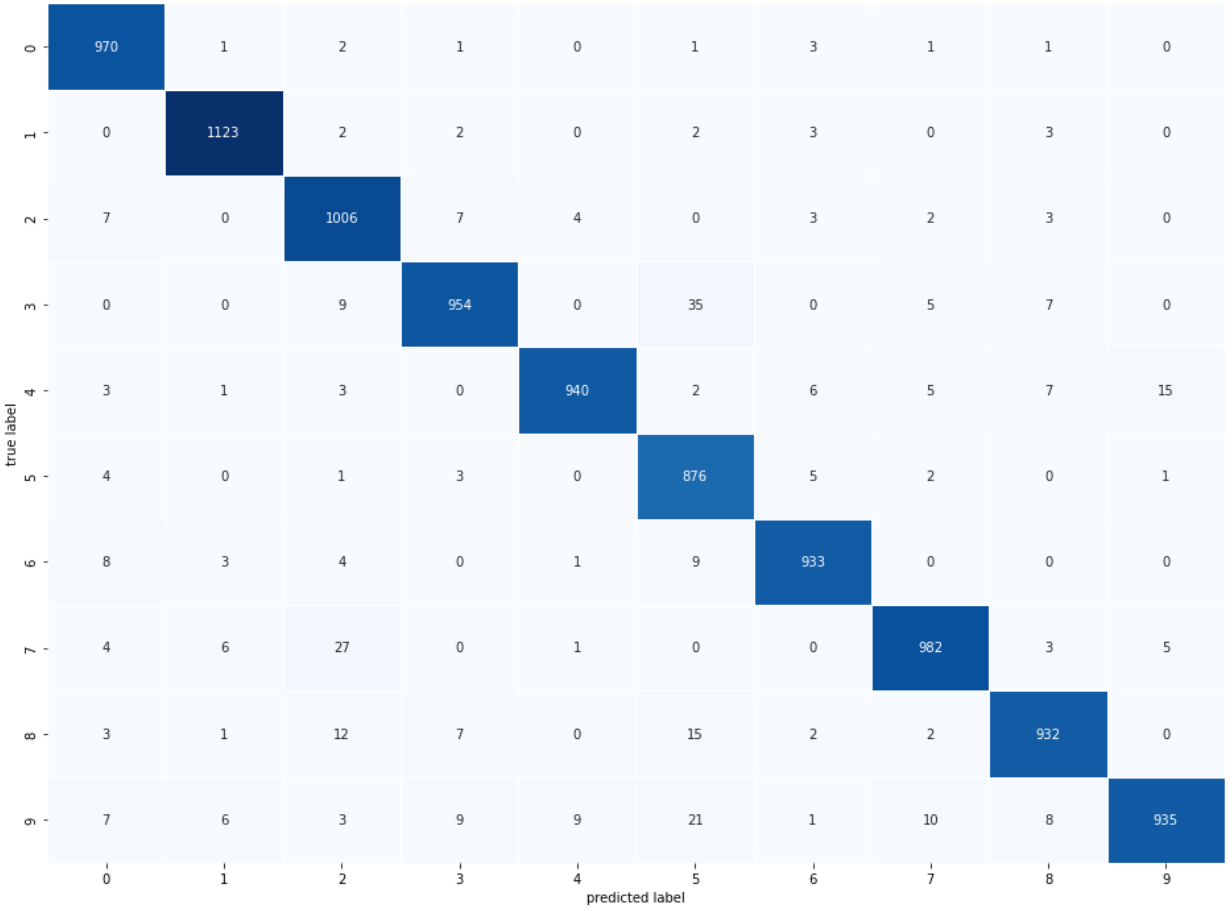
Out[252]:

	0	1	2	3	4	5	6	7	8	9
0	0.00%	0.00%	0.03%	0.02%	0.00%	0.00%	0.00%	99.94%	0.00%	0.00%
1	0.00%	0.00%	99.97%	0.01%	0.00%	0.00%	0.01%	0.00%	0.00%	0.00%
2	0.00%	99.31%	0.11%	0.01%	0.02%	0.04%	0.09%	0.34%	0.07%	0.00%
3	99.99%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
4	0.01%	0.00%	0.22%	0.00%	97.27%	0.00%	0.02%	0.49%	0.11%	1.88%
5	0.00%	99.72%	0.00%	0.00%	0.01%	0.00%	0.00%	0.26%	0.01%	0.00%
6	0.00%	0.00%	0.00%	0.01%	95.35%	0.25%	0.01%	0.99%	3.18%	0.21%
7	0.00%	0.01%	0.16%	0.66%	0.21%	0.12%	0.00%	0.80%	0.11%	97.93%
8	0.01%	0.00%	0.68%	0.00%	0.05%	3.92%	95.17%	0.00%	0.17%	0.00%
9	0.00%	0.00%	0.00%	0.01%	0.41%	0.00%	0.00%	1.41%	0.26%	97.90%
10	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
11	0.29%	0.00%	0.02%	0.00%	0.00%	0.12%	99.43%	0.00%	0.13%	0.00%
12	0.00%	0.00%	0.01%	0.08%	0.07%	0.07%	0.00%	0.03%	0.08%	99.67%
13	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
14	0.00%	99.99%	0.00%	0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
15	0.00%	0.00%	0.00%	0.69%	0.00%	99.28%	0.00%	0.00%	0.03%	0.00%
16	0.02%	0.00%	0.21%	0.02%	0.05%	0.01%	0.00%	0.30%	1.02%	98.38%
17	0.00%	0.00%	0.04%	0.03%	0.00%	0.00%	0.00%	99.93%	0.00%	0.00%
18	0.01%	0.00%	4.67%	94.20%	0.00%	0.70%	0.08%	0.01%	0.27%	0.05%
19	0.00%	0.00%	0.00%	0.00%	99.92%	0.00%	0.00%	0.02%	0.00%	0.05%

Let's create a confusion matrix that visualizes model performance on testing data.

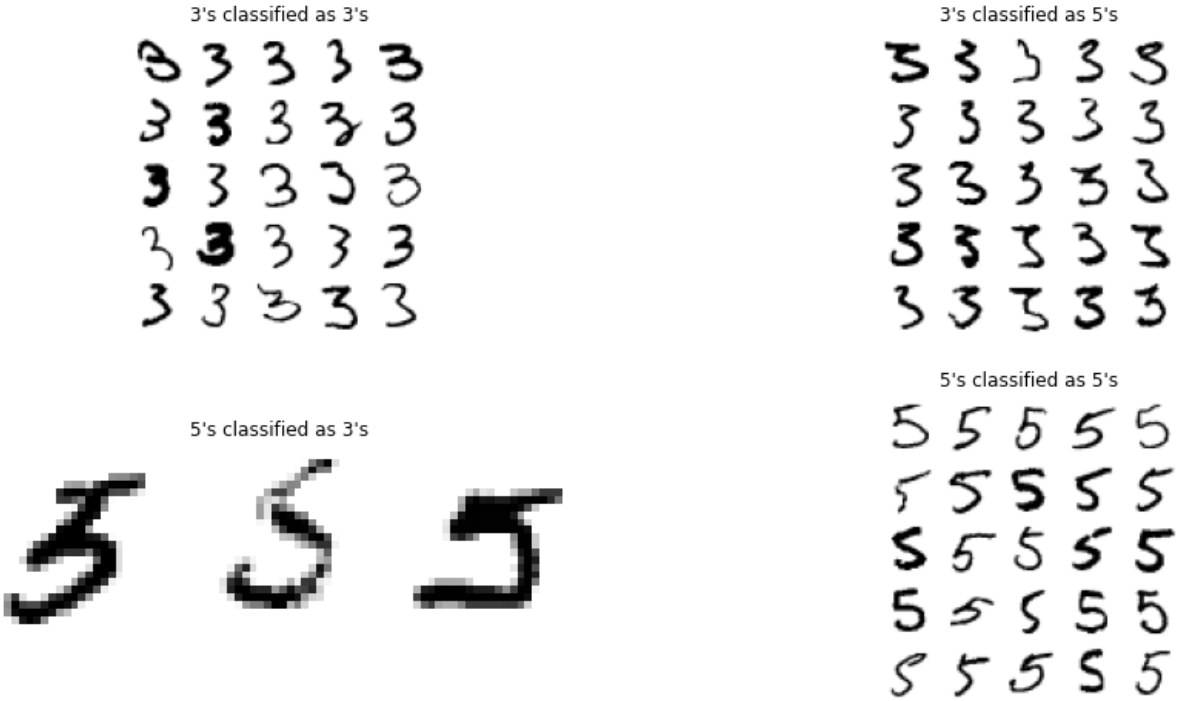
In [253…

```
plot_confusion_matrix(y_test,pred_classes)
```



In [254…

```
cl_a, cl_b = 3, 5
X_aa = x_test_norm[(y_test == cl_a) & (pred_classes == cl_a)]
X_ab = x_test_norm[(y_test == cl_a) & (pred_classes == cl_b)]
X_ba = x_test_norm[(y_test == cl_b) & (pred_classes == cl_a)]
X_bb = x_test_norm[(y_test == cl_b) & (pred_classes == cl_b)]

plt.figure(figsize=(16,8))

p1 = plt.subplot(221)
p2 = plt.subplot(222)
p3 = plt.subplot(223)
p4 = plt.subplot(224)

plot_digits(X_aa[:25], p1, images_per_row=5);
plot_digits(X_ab[:25], p2, images_per_row=5);
plot_digits(X_ba[:25], p3, images_per_row=5);
plot_digits(X_bb[:25], p4, images_per_row=5);


p1.set_title(f"{cl_a}'s classified as {cl_a}'s")
p2.set_title(f"{cl_a}'s classified as {cl_b}'s")
p3.set_title(f"{cl_b}'s classified as {cl_a}'s")
p4.set_title(f"{cl_b}'s classified as {cl_b}'s")

# plt.savefig("error_analysis_digits_plot_EXP1_valid")

plt.show()
```



11) Experiment 9 - Principal Components Analysis Followed By Artificial Neural Network

Now we will run an experiment where we execute principal components analysis to reduce the dimensionality of the input data prior to the creation of an artificial neural network to classify the PCA-reduced observations as digits (from 0 to 9).

11.1) Reduce Data Dimensionality Via Principal Components Analysis

First, we import the relevant packages.

In [263…
```
from sklearn.decomposition import PCA
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
keras.__version__
```

Out[263]:
```
'2.12.0'
```

In [264…
```
tf.__version__
```

Out[264]:
```
'2.12.0'
```

In [265…
```
# To get consistint results each time we rerun the code.
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)
```

Second, we will re-import and preprocess the data.

In [266…
```
from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

In [267…
```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

In [268…
```
val_images, train_images = train_images[:5000], train_images[5000:]
val_labels, train_labels = train_labels[:5000], train_labels[5000:]
```

In [269…
```
pca = PCA(n_components=0.95)
train_images_red = pca.fit_transform(train_images)
val_images_red = pca.transform(val_images)
test_images_red = pca.transform(test_images)
```

In [270…
```
test_images_red.shape, train_images_red.shape, val_images_red.shape
```

Out[270]:
```
((10000, 154), (55000, 154), (5000, 154))
```

11.2) Construct Artificial Neural Network Model

In [271…
```
from tensorflow.keras import models
from tensorflow.keras import layers

model = models.Sequential()
model.add(layers.Dense(130, activation='relu', input_shape=(154,)))
model.add(layers.Dense(10, activation='softmax'))

# For use with non-categorical labels
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_images_red, train_labels, epochs=30,
                    validation_data=(val_images_red, val_labels))
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [==============================] - 2s 28us/sample - loss: 0.3037 - accura
cy: 0.9178 - val_loss: 0.1425 - val_accuracy: 0.9606
Epoch 2/30
 3552/55000 [>.............................] - ETA: 1s - loss: 0.1252 - accuracy: 0.9
651
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2335: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates
```

```
55000/55000 [==============================] - 1s 26us/sample - loss: 0.1167 - accura
cy: 0.9661 - val_loss: 0.0970 - val_accuracy: 0.9744
Epoch 3/30
55000/55000 [==============================] - 1s 25us/sample - loss: 0.0789 - accura
cy: 0.9775 - val_loss: 0.0846 - val_accuracy: 0.9774
Epoch 4/30
55000/55000 [==============================] - 1s 25us/sample - loss: 0.0580 - accura
cy: 0.9837 - val_loss: 0.0800 - val_accuracy: 0.9772
Epoch 5/30
55000/55000 [==============================] - 1s 25us/sample - loss: 0.0445 - accura
cy: 0.9876 - val_loss: 0.0774 - val_accuracy: 0.9792
Epoch 6/30
55000/55000 [==============================] - 1s 27us/sample - loss: 0.0343 - accura
cy: 0.9906 - val_loss: 0.0768 - val_accuracy: 0.9794
Epoch 7/30
55000/55000 [==============================] - 2s 28us/sample - loss: 0.0271 - accura
cy: 0.9931 - val_loss: 0.0762 - val_accuracy: 0.9792
Epoch 8/30
55000/55000 [==============================] - 2s 28us/sample - loss: 0.0208 - accura
cy: 0.9947 - val_loss: 0.0795 - val_accuracy: 0.9786
Epoch 9/30
55000/55000 [==============================] - 2s 31us/sample - loss: 0.0169 - accura
cy: 0.9961 - val_loss: 0.0846 - val_accuracy: 0.9788
Epoch 10/30
55000/55000 [==============================] - 2s 30us/sample - loss: 0.0135 - accura
cy: 0.9967 - val_loss: 0.0907 - val_accuracy: 0.9792
Epoch 11/30
55000/55000 [==============================] - 2s 28us/sample - loss: 0.0106 - accura
cy: 0.9977 - val_loss: 0.0899 - val_accuracy: 0.9804
Epoch 12/30
55000/55000 [==============================] - 1s 25us/sample - loss: 0.0084 - accura
cy: 0.9983 - val_loss: 0.0948 - val_accuracy: 0.9778
Epoch 13/30
55000/55000 [==============================] - 2s 30us/sample - loss: 0.0063 - accura
cy: 0.9987 - val_loss: 0.0968 - val_accuracy: 0.9800
Epoch 14/30
55000/55000 [==============================] - 1s 26us/sample - loss: 0.0051 - accura
cy: 0.9991 - val_loss: 0.1026 - val_accuracy: 0.9784
Epoch 15/30
55000/55000 [==============================] - 1s 25us/sample - loss: 0.0039 - accura
cy: 0.9993 - val_loss: 0.1059 - val_accuracy: 0.9800
Epoch 16/30
55000/55000 [==============================] - 1s 25us/sample - loss: 0.0030 - accura
cy: 0.9994 - val_loss: 0.1107 - val_accuracy: 0.9794
Epoch 17/30
55000/55000 [==============================] - 1s 25us/sample - loss: 0.0022 - accura
cy: 0.9995 - val_loss: 0.1168 - val_accuracy: 0.9786
Epoch 18/30
55000/55000 [==============================] - 1s 25us/sample - loss: 0.0017 - accura
cy: 0.9997 - val_loss: 0.1150 - val_accuracy: 0.9792
Epoch 19/30
55000/55000 [==============================] - 1s 25us/sample - loss: 0.0014 - accura
cy: 0.9999 - val_loss: 0.1238 - val_accuracy: 0.9796
Epoch 20/30
55000/55000 [==============================] - 1s 24us/sample - loss: 0.0012 - accura
cy: 0.9999 - val_loss: 0.1260 - val_accuracy: 0.9790
Epoch 21/30
55000/55000 [==============================] - 1s 25us/sample - loss: 8.5560e-04 - ac
curacy: 0.9999 - val_loss: 0.1324 - val_accuracy: 0.9798
Epoch 22/30
```

```
55000/55000 [==============================] - 1s 25us/sample - loss: 7.0583e-04 - ac
curacy: 1.0000 - val_loss: 0.1355 - val_accuracy: 0.9792
Epoch 23/30
55000/55000 [==============================] - 1s 26us/sample - loss: 5.7645e-04 - ac
curacy: 1.0000 - val_loss: 0.1455 - val_accuracy: 0.9784
Epoch 24/30
55000/55000 [==============================] - 1s 26us/sample - loss: 5.9500e-04 - ac
curacy: 0.9999 - val_loss: 0.1493 - val_accuracy: 0.9782
Epoch 25/30
55000/55000 [==============================] - 1s 25us/sample - loss: 4.9598e-04 - ac
curacy: 0.9999 - val_loss: 0.1576 - val_accuracy: 0.9788
Epoch 26/30
55000/55000 [==============================] - 1s 25us/sample - loss: 4.8918e-04 - ac
curacy: 1.0000 - val_loss: 0.1634 - val_accuracy: 0.9788
Epoch 27/30
55000/55000 [==============================] - 1s 25us/sample - loss: 4.0942e-04 - ac
curacy: 1.0000 - val_loss: 0.1669 - val_accuracy: 0.9786
Epoch 28/30
55000/55000 [==============================] - 1s 26us/sample - loss: 3.9493e-04 - ac
curacy: 1.0000 - val_loss: 0.1763 - val_accuracy: 0.9788
Epoch 29/30
55000/55000 [==============================] - 1s 26us/sample - loss: 3.1013e-04 - ac
curacy: 1.0000 - val_loss: 0.1746 - val_accuracy: 0.9794
Epoch 30/30
55000/55000 [==============================] - 1s 26us/sample - loss: 3.0066e-04 - ac
curacy: 1.0000 - val_loss: 0.1818 - val_accuracy: 0.9788
```

11.3) Evalaute the Performance of the Model

In [272…
```
hist_dict = history.history
hist_dict.keys()
```

Out[272]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [273…
```
print(f'''acc: {hist_dict['accuracy'][-1]:.4}, val acc: {hist_dict['val_accuracy'][-1]
loss: {hist_dict['loss'][-1]:.4}, val loss: {hist_dict['val_loss'][-1]:.4}''')
```

```
acc: 1.0, val acc: 0.9788,
loss: 0.0003007, val loss: 0.1818
```

In [274…
```
test_loss, test_acc = model.evaluate(test_images_red, test_labels)
```

In [275…
```
print(f'test acc: {test_acc}, test loss: {test_loss}')
```

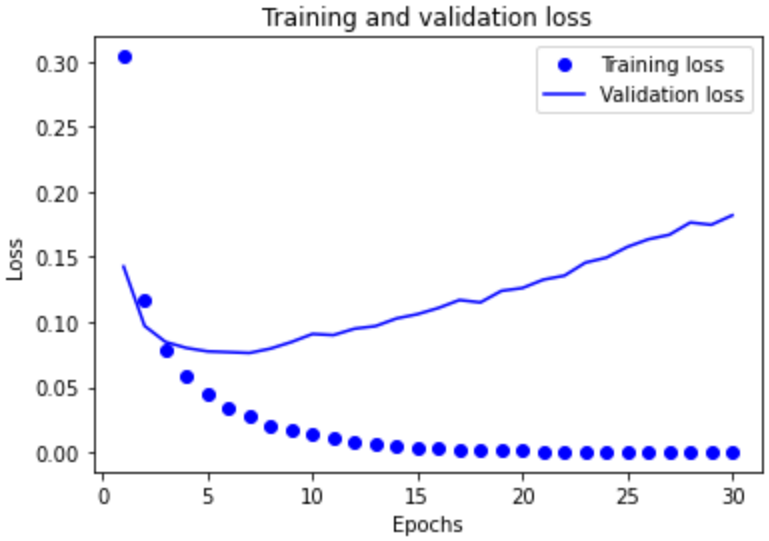
```
test acc: 0.9778000116348267, test loss: 0.1833842407467074
```

In [276…
```
history_dict = history.history
history_dict.keys()
```

Out[276]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

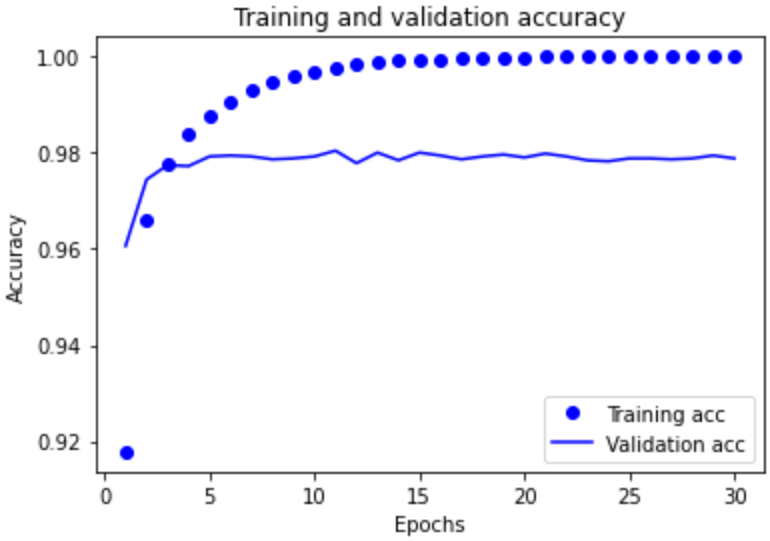
In [277…
```
import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
```

```
epochs = range(1, len(acc) + 1)

# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



In [278…

```
plt.clf()   # clear figure
acc_values = history_dict['accuracy']
# val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')

plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



12) Experiment 10 - Random Forest Classifier-Informed Artificial Neural Network Model

In this experiment, we will create a Random Forest Classifier Model to inform us which of the pixels are the 70 most important predictors for digit classification. Then, we will create an artificial neural network model that solely relies on those 70 pixels as its input data.

12.1) Create a Random Forest Classifier Model

First, we import the necessary packages and preprocess the MNSIT data.

In [279…
```
from sklearn.ensemble import RandomForestClassifier
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
keras.__version__
```

Out[279]:
```
'2.12.0'
```

In [280…
```
tf.__version__
```

Out[280]:
```
'2.12.0'
```

In [281…
```
tf.compat.v1.disable_eager_execution() # neccessary for K.gradient to work in TensorFl
```

In [282…
```
# To get consistint results each time we rerun the code.
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)
```

In [283…
```
from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

In [284…
```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

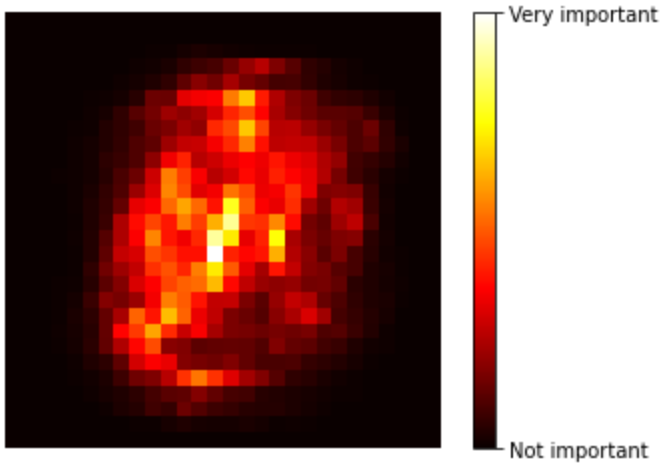
In [285…
```
val_images, train_images = train_images[:5000], train_images[5000:]
val_labels, train_labels = train_labels[:5000], train_labels[5000:]
```

We create a Random Forrest Classifier (with the default 100 trees) and use it to find the relative importance of the 784 features (pixels) in the training set. We produce a heat map to visual the relative importance of the features (using code from Hands On Machine Learning by A. Geron). Finally, we select the 70 most important feature (pixels) from the training, validation and test images to test our 'best' model on.

In [286…
```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
rnd_clf.fit(train_images,train_labels)
```

Out[286]:
```
RandomForestClassifier
RandomForestClassifier(random_state=42)
```

In [287…
```
# https://github.com/ageron/handson-ml2/blob/master/07_ensemble_learning_and_random_fo
import matplotlib as mpl
import matplotlib.pyplot as plt

def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = mpl.cm.hot,
               interpolation="nearest")
    plt.axis("off")

plot_digit(rnd_clf.feature_importances_)

cbar = plt.colorbar(ticks=[rnd_clf.feature_importances_.min(), rnd_clf.feature_importa
cbar.ax.set_yticklabels(['Not important', 'Very important'])

# plt.savefig("mnist_feature_importance_plot")
plt.show()
```



In [288…
```
# https://stackoverflow.com/questions/6910641/how-do-i-get-indices-of-n-maximum-values
n = 70
imp_arr = rnd_clf.feature_importances_
idx = (-imp_arr).argsort()[:n]          # get the indices of the 70 "most important" f
len(idx)
```

Out[288]:
```
70
```

In [289…
```
# Create training, validation and test images using just the 70 pixel locations obtain
train_images_sm = train_images[:,idx]
val_images_sm = val_images[:,idx]
test_images_sm = test_images[:,idx]
train_images_sm.shape, val_images.shape, test_images_sm.shape # the reduced images hav
```

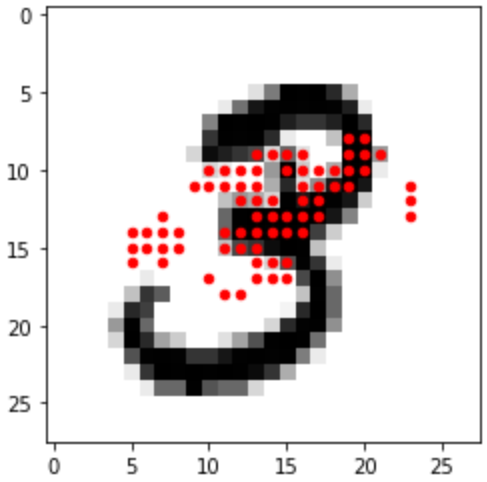
Out[289]:
```
((55000, 70), (5000, 784), (10000, 70))
```

Let's visualize the 70 pixels that we'll be using for our artificial neural network model.

In [290…
```
# to convert an index n, 0<= n < 784
def pair(n,size):
    x = n//size
    y = n%size
    return x,y
```

In [291…
```
plt.imshow(train_images[1].reshape(28,28),cmap='binary')
x, y = np.array([pair(k,28) for k in idx]).T
plt.scatter(x,y,color='red',s=20)
```

Out[291]:
```
<matplotlib.collections.PathCollection at 0x24f5a259790>
```



12.2) Construct the Artificial Neural Network Classification Model

Let's create an artificial neural network model that only uses the 70 most important pixels identified by the Random Forest Classifier model.

In [293…

```
from tensorflow.keras import models
from tensorflow.keras import layers

model = models.Sequential()
model.add(layers.Dense(130, activation='relu', input_shape=(70,)))
model.add(layers.Dense(10, activation='softmax'))

# For use with non-categorical labels
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_images_sm, train_labels, epochs=30,
                    validation_data=(val_images_sm, val_labels))
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [==============================] - 1s 24us/sample - loss: 0.5826 - accura
cy: 0.8272 - val_loss: 0.4034 - val_accuracy: 0.8828
Epoch 2/30
 1664/55000 [..............................] - ETA: 1s - loss: 0.4242 - accuracy: 0.8
666
```

```
C:\Users\steve\anaconda3\lib\site-packages\keras\engine\training_v1.py:2335: UserWarn
ing: `Model.state_updates` will be removed in a future version. This property should
not be used in TensorFlow 2.0, as `updates` are applied automatically.
  updates = self.state_updates
```

```
55000/55000 [==============================] - 1s 22us/sample - loss: 0.3757 - accura
cy: 0.8867 - val_loss: 0.3164 - val_accuracy: 0.9100
Epoch 3/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.3067 - accura
cy: 0.9069 - val_loss: 0.2901 - val_accuracy: 0.9154
Epoch 4/30
55000/55000 [==============================] - 1s 23us/sample - loss: 0.2690 - accura
cy: 0.9188 - val_loss: 0.2519 - val_accuracy: 0.9296
Epoch 5/30
55000/55000 [==============================] - 1s 24us/sample - loss: 0.2454 - accura
cy: 0.9260 - val_loss: 0.2408 - val_accuracy: 0.9296
Epoch 6/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.2289 - accura
cy: 0.9305 - val_loss: 0.2256 - val_accuracy: 0.9324
Epoch 7/30
55000/55000 [==============================] - 1s 21us/sample - loss: 0.2154 - accura
cy: 0.9345 - val_loss: 0.2237 - val_accuracy: 0.9348
Epoch 8/30
55000/55000 [==============================] - 1s 21us/sample - loss: 0.2047 - accura
cy: 0.9377 - val_loss: 0.2174 - val_accuracy: 0.9356
Epoch 9/30
55000/55000 [==============================] - 1s 21us/sample - loss: 0.1968 - accura
cy: 0.9400 - val_loss: 0.2120 - val_accuracy: 0.9380
Epoch 10/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1898 - accura
cy: 0.9421 - val_loss: 0.2159 - val_accuracy: 0.9374
Epoch 11/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1842 - accura
cy: 0.9437 - val_loss: 0.2075 - val_accuracy: 0.9410
Epoch 12/30
55000/55000 [==============================] - 1s 21us/sample - loss: 0.1778 - accura
cy: 0.9451 - val_loss: 0.2141 - val_accuracy: 0.9368
Epoch 13/30
55000/55000 [==============================] - 1s 21us/sample - loss: 0.1725 - accura
cy: 0.9474 - val_loss: 0.2096 - val_accuracy: 0.9398
Epoch 14/30
55000/55000 [==============================] - 1s 21us/sample - loss: 0.1687 - accura
cy: 0.9487 - val_loss: 0.2039 - val_accuracy: 0.9422
Epoch 15/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1640 - accura
cy: 0.9494 - val_loss: 0.2045 - val_accuracy: 0.9420
Epoch 16/30
55000/55000 [==============================] - 1s 21us/sample - loss: 0.1607 - accura
cy: 0.9507 - val_loss: 0.2089 - val_accuracy: 0.9434
Epoch 17/30
55000/55000 [==============================] - 1s 21us/sample - loss: 0.1577 - accura
cy: 0.9525 - val_loss: 0.2183 - val_accuracy: 0.9398
Epoch 18/30
55000/55000 [==============================] - 2s 28us/sample - loss: 0.1553 - accura
cy: 0.9527 - val_loss: 0.2055 - val_accuracy: 0.9414
Epoch 19/30
55000/55000 [==============================] - 1s 23us/sample - loss: 0.1515 - accura
cy: 0.9544 - val_loss: 0.2180 - val_accuracy: 0.9392
Epoch 20/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1491 - accura
cy: 0.9549 - val_loss: 0.2218 - val_accuracy: 0.9404
Epoch 21/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1474 - accura
cy: 0.9553 - val_loss: 0.2098 - val_accuracy: 0.9422
Epoch 22/30
```

```
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1443 - accura
cy: 0.9563 - val_loss: 0.2126 - val_accuracy: 0.9384
Epoch 23/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1424 - accura
cy: 0.9569 - val_loss: 0.2195 - val_accuracy: 0.9374
Epoch 24/30
55000/55000 [==============================] - 1s 21us/sample - loss: 0.1403 - accura
cy: 0.9576 - val_loss: 0.2214 - val_accuracy: 0.9430
Epoch 25/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1403 - accura
cy: 0.9573 - val_loss: 0.2084 - val_accuracy: 0.9442
Epoch 26/30
55000/55000 [==============================] - 1s 21us/sample - loss: 0.1379 - accura
cy: 0.9585 - val_loss: 0.2187 - val_accuracy: 0.9404
Epoch 27/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1362 - accura
cy: 0.9589 - val_loss: 0.2222 - val_accuracy: 0.9438
Epoch 28/30
55000/55000 [==============================] - 1s 23us/sample - loss: 0.1347 - accura
cy: 0.9596 - val_loss: 0.2239 - val_accuracy: 0.9426
Epoch 29/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1342 - accura
cy: 0.9599 - val_loss: 0.2240 - val_accuracy: 0.9418
Epoch 30/30
55000/55000 [==============================] - 1s 22us/sample - loss: 0.1322 - accura
cy: 0.9605 - val_loss: 0.2246 - val_accuracy: 0.9426
```

12.3) Evaluate the Performance of the Model

In [294…
```
hist_dict = history.history
hist_dict.keys()
```

Out[294]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [295…
```
print(f'''acc: {hist_dict['accuracy'][-1]:.4}, val acc: {hist_dict['val_accuracy'][-1]
loss: {hist_dict['loss'][-1]:.4}, val loss: {hist_dict['val_loss'][-1]:.4}''')
```

```
acc: 0.9605, val acc: 0.9426,
loss: 0.1322, val loss: 0.2246
```

In [296…
```
test_loss, test_acc = model.evaluate(test_images_sm, test_labels)
```

In [297…
```
print(f'test acc: {test_acc}, test loss: {test_loss}')
```

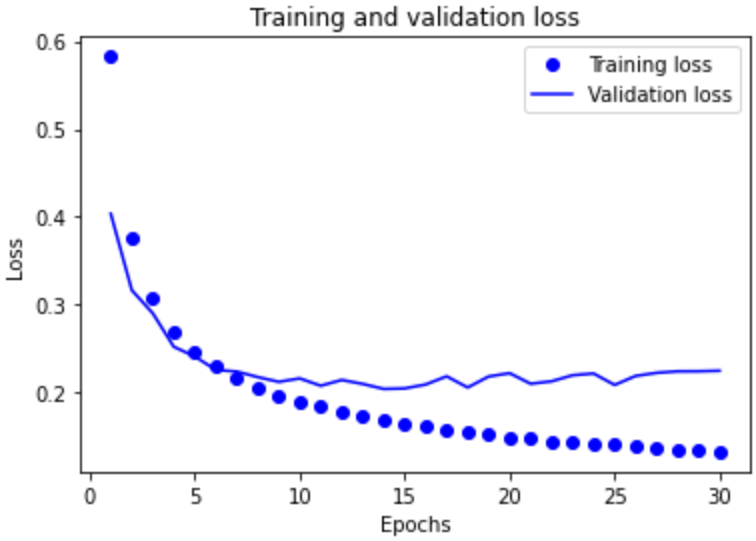
```
test acc: 0.9420999884605408, test loss: 0.22947177608385683
```

In [298…
```
history_dict = history.history
history_dict.keys()
```

Out[298]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

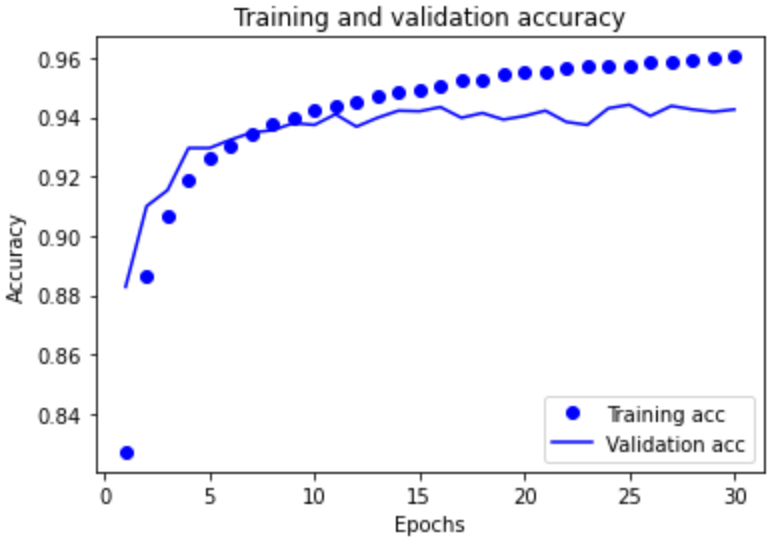
In [299…
```
import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
```

```
epochs = range(1, len(acc) + 1)

# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



In [300…

```
plt.clf()   # clear figure
acc_values = history_dict['accuracy']
# val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')

plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```


Training and validation accuracy
Accuracy
Epochs
Training acc
Validation acc
0.96
0.94
0.92
0.90
0.88
0.86
0.84
0
5
10
15
20
25
30