Abstract:

This document, written by Zain Syed, the VP Tech at WE AutoPilot, will guide you through building (potentially) your first Machine Learning model! We will be using the MNIST dataset to make an AI tool to recognize handwritten numbers. The principle is, by submitting thousands of handwritten images, we can teach our computer to recognize patterns, making it capable of recognizing numbers. **WHY CARE?** Well, although it may seem like a trivial task to read handwritten numbers (unless it's my handwriting), for a computer it is really huge. If I get a calculator, or a computer to type out the number "8", they do not actually understand what 8 is, or what it represents. It is merely a symbol among others to a computer. If we can train a computer to **visually interpret and understand** a number, that is incredible news! It is like the natural next step when it comes to computer programming. Although this machine learning is only used to identify handwritten numbers, we can take it one step further, and do all sorts of incredible things with AI! Such as autonomous driving.

Keep in mind: All commands in this color are for the terminal
All commands in this color are for our actual files

Stage 1) Getting Started, downloading libraries & programs:

Firstly, we need to ensure we have Python and Pip installed, as well as an IDE (preferably VSCode to make the rest of the process much easier). You guys most likely have all of this stuff though, so no need to worry!

After that, run the following command: `pip install tensorflow numpy matplotlib seaborn`
This should install all the required dependencies, if any issues persist, refer to the FAQ or feel free to ask one of the workshop mentors.

Now we are able to start working on the code itself! Firstly, we need to make a file and import all the libraries we just downloaded. Create a file with any name you like, just make sure it has the .py extension, many people like to do main.py for the training application

The following code will be able for us to import our libraries into the code itself!

```
import tensorflow
Import numpy
Import matplotlib.pyplot
```

Optionally, you can use the "as" keyword to provide the imports with aliases (think of it as making a variable, or a library type object in OOP). We can refer to tensorflow as tf if we would like! This is called creating a ***namespace***. I did not use namespaces to maintain readability

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plot
```

Stage 2) Data Preprocessing:

Preprocessing is simply the idea of manipulating our dataset before we send it to the code to train our machine learning model. This can be for many reasons, but generally it's making sure our files are consistent, and not too large to use.

The next step is to download the MNIST dataset (from TensorFlow), and then loading it. We can add the following line of code:

```
mnist = tensorflow.keras.datasets.mnist
```
(replace tensorflow with your namespace name)

Then we can add this line to load our data.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Here is what everything does:
mnist.load_data() is the function that loads data with handwritten images and their labels
x_train is the dataset of 28x28 pixel images of handwritten numbers that the model will train on
y_train is the dataset of labels corresponding to the images in x_train *(54000 images in train set)*
x_test is the dataset of 28x28 pixel images of handwritten numbers that the model will test on
y_test is the dataset of labels corresponding to the images in x_test *(6000 images in train set)*

We can optionally use these lines of code to test our dataset (not necessary today, but recommended when training new, less “reliable” datasets)

```
seaborn.countplot(y_train)
print(numpy.isnan(x_train).any())
```
(replace numpy with your namespace name)
```
print(numpy.isnan(x_test).any())
```

Stage 3) Normalization:

We have our data, but we need to make it compatible with tensorflow, and make it optimized for when we train our model. The form our data is in right now is not compatible with how we intend to train it, so the following lines of code will aim to bring it to standard.

We start with our inputs, we know that our images are all 28x28 pixels, but the shape of the input images has to be explicitly defined for the model to be able to process it, we can define it with the following line of code. The 1 represents 1 color channel, meaning a grayscale image

```
input_shape = (28, 28, 1)
```

The next step is to reshape the data from `(60000, 28, 28)` to `(60000, 28, 28, 1)` to match TensorFlow’s expected input format. Then, we normalize pixel values by dividing by 255.0, scaling them from **0-255** to **0-1**, making the model process them more efficiently.

```
x_train = (x_train.reshape(x_train.shape[0], x_train.shape[1], x_train.shape[2], 1))/255.0
x_test = (x_test.reshape(x_test.shape[0], x_test.shape[1], x_test.shape[2], 1))/255.0
```

The final step for this stage is to encode our labels from integers to one-hot encoded form

```
y_train = tensorflow.one_hot(y_train.astype(numpy.int32), depth=10)
y_test = tensorflow.one_hot(y_test.astype(numpy.int32), depth=10)
```

Stage 4) Train Convolutional Neural Network:

FINALLY! Whether you read through all the information on this document, or sat through Zain yapping nonstop in the workshop, you finally have made it to the fun part. This is where we will be using all the data we have prepared, and using it to actually train the CNN! If the stars align and nothing goes wrong, a lot of you officially will have built your first AI model after this final step!

Firstly, we will define our batch size (how many images processed together), number of classes (how many neurons our model has), and epochs (how many times the model trains from the same dataset)

```
batch_size = 64
num_classes = 10
epochs = 5
```

Now for the fun part, the training! We can start by defining the following:

```
layer = tensorflow.keras.layers
model = tensorflow.keras.models.Sequential([])
```

Now the **ENTIRETY** of this code will go **INSIDE** the square brackets of Sequential([]) that we defined above. It is best to format each line of code on a separate line in your text editor, it enhances readability and helps us visualize the sequential order of the layers we are creating. After that we can compile the model as well!

```
([
layer.Conv2D(32, (5,5), padding='same', activation='relu', input_shape=input_shape),
layer.Conv2D(32, (5,5), padding='same', activation='relu'),     layer.MaxPool2D(),
layer.Dropout(0.25),
layer.Conv2D(64, (3,3), padding='same', activation='relu'),
layer.Conv2D(64, (3,3), padding='same', activation='relu'),
layer.MaxPool2D(strides=(2,2)),
layer.Dropout(0.25),
layer.Flatten(),
layer.Dense(128, activation='relu'),
layer.Dropout(0.5), # Final dropout, drops 50% of neurons at random
layer.Dense(numClasses, activation='softmax')
])

model.compile(optimizer=tensorflow.keras.optimizers.RMSprop(epsilon=1e-08),
loss='categorical_crossentropy', metrics=['acc'])
```

This may seem like a lot, but it's simpler than it looks! Each image passes through multiple layers, where the model extracts patterns to train itself. The next page explains each layer and shows how they transform the input. However, we must avoid **overfitting**, where the model **MEMORIZES** instead of **UNDERSTANDING** the data. A little uncertainty helps it generalize better to new inputs!

Conv2D: Creates a convolutional layer, which has a specified amount of filters of specific sizes. These filters go over our image pixel by pixel and aim to find patterns. Sometimes convolutional layers are doubled up to enhance our pattern recognition on a given image
MaxPool2D: The max pooling layer takes the maximum value from each 2x2 block, essentially halving the size of the image. It reduces computation and shrinks the image
Dropout: The dropout layer essentially deactivates a specified percentage of the neurons during training, we do this to prevent overfitting and too much training
Flatten: The flatten layer converts at 2D feature map into a 1D vector, preparing it for the final few layers
Dense: The fully connected dense layer condenses our neurons into a specified amount, it adds non-linearity to learn more complex patterns. The last Dense layer creates the output layer, which converts the outputs into probabilities for each digit. Our model will read an image, and give each digit 0-9 a probability value, and determine what the inputted number is based on the highest probability value
padding: A function to ensure the output shape remains the same when going through the layer
relu: The ReLU activation function allows our model to learn complex patterns by reducing linearity of the inputs
softmax: The softmax activation function is what converts the output of our model into a list of probabilities for each digit as defined above in the output layer

Effects of Each Sequential Layer (in order)			
Layer Type	**Filters/Units**	**Activation**	**Output Shape**
Conv2D	32 (5x5)	ReLU	(28, 28, 32)
Conv2D	32 (5x5)	ReLU	(28, 28, 32)
MaxPooling2D	-	-	(14, 14, 32)
Dropout	25%	-	(14, 14, 32)
Conv2D	64 (3x3)	ReLU	(14, 14, 64)
Conv2D	64 (3x3)	ReLU	(14, 14, 64)
MaxPooling2D	-	-	(7, 7, 64)
Dropout	25%	-	(7, 7, 64)
Flatten	-	-	(3136) (Neurons)
Dense	128	ReLU	(128) (Neurons)
Dropout	50%	-	(128) (Neurons)
Dense	10	Softmax	(10) (Neurons)

The next step is to create a callback function. If at any point our model becomes 99.5% accurate (or more), we will stop training, there won't be the need to train anymore. This could potentially end the code early, before the 5 epochs end.

```
class myCallback(tensorflow.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs={}):
    if(logs.get('acc')>0.995):
      print("\nReached 99.5% accuracy so cancelling training!")
      self.model.stop_training = True

callbacks = myCallback()
```

The final step of this stage is to train our model, using predefined values. Keep in mind that we will be training our data with 90% of the dataset, and the remaining 10% of the images will be reserved for the testing phase. That is why validation_split = 0.1

```
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_split=0.1,
                    callbacks=[callbacks]
                )
```

Stage 5) Evaluate our Model (optional):

Although for the sake of this workshop it is not necessary, and won't be gone over I would highly recommend you guys do your research, and try out the code. The code for the evaluation won't be on this document, but you can find it on the main.py file on the GitHub Repository. It is good to know how to do these when training your own models, so you can easily identify errors for improvement.

Essentially what we are aiming to do in this step is visualize the loss and accuracy of our model. We will start by plotting loss and accuracy curves, these will visualize how our loss decreases over time, and how our accuracy increases. After that, we can create a confusion matrix to visualize where and why our model went wrong. Feel free to reach out to mentors or Zain for more information.

Stage 6) Create .h5 File & Run training file

Now that all our code is written, our next step is to add our finishing few lines of code and then execute our python file. The first thing to do is add these lines at the bottom of our code:

```
model.save("mnist_model.h5")
model.summary()
```

Finally, we can double check that all our syntax is correct, resolve any remaining errors, and then run our python file. The best way to do this is by running this command in the terminal:

```
python .\main.py
```

Replace “main.py” with the name of your python file, and make sure your in the right directory

Congratulations! You have completely finished the scope of this workshop! All that's left is to continue to expand your machine learning knowledge! Feel free to ask us questions, and even experiment with your current training model! There is also a GUI tool you can use to visualize your model! The files should be available on the GitHub repository!

FAQs and Common Problems:

Error: Could not install packages due to OSError:[WinError 2], system cannot find file specified
This error can easily be solved, it likely means Python isn't installed, or the path is not correct.
To solve it: Hit **Win+R**, and type **sysdm.cpl**, then navigate to:
Advanced > Environment Variables > User variables for [username] > Path
Now you can click **New**, and add the following to your list of Paths,
C:\Python310\
C:\Python310\Scripts
Be careful before you add these to path, you may have another version of python installed, let a mentor double check everything before you do it.
After the path is added, restart your computer, open a terminal and type where python it should return some path, which means you did it correctly, and the error shouldn't persist.