Course Introduction

Hey everyone and welcome to our course on image recognition with Tensorflow! Throughout this course, we will learn about how image recognition works and use some of the tools in the Tensorflow library to help us construct programs to recognize handwritten digits. We will start by exploring how we recognize image and how machines try to mimic that process. This will give us insight into how machines process images and some of the challenges that we might face when trying to train machines to recognize and classify images. Next, we will familiarize ourselves with some of the tools that can be of aid. These include from mechanisms to transform and apply filters to images and functions that process the transformed results, all found in Tensorflow. After this, we will build a custom model from scratch using Tensorflow with custom training and testing functionality. Finally, we will build another similar model using Keras functionality.

We chose this topic as it is a great place to start when learning about machine learning and there are a variety of applications for image classifiers. We will build a simple feed-forward network when acts as an extension for a simpler network that we built in the Intro to Machine Learning and Tensorflow Part 1 course. This is a convenient way to expand upon concepts with which we should already be somewhat familiar and a way to comfortably introduce some new concepts without throwing too much at you at once. Image recognition also makes up a component of many larger machine learning systems. Most programs that read their surroundings employ some sort of image recognition from self driving cars, to virtual reality games. Understanding how image recognition works is key to understanding how bigger systems process data and respond to their environment. Lastly, there are many well-known datasets readily available for free that can be used to train image recognition models so it’s a convenient topic to broach.

Intro to Image Recognition

Let's get started by learning a bit about the topic itself. **Image recognition is, at its heart, image classification so we will use these terms interchangeably throughout this course.** We see images or real-world items and we classify them into one (or more) of many, many possible categories. The categories used are entirely up to use to decide. For example, we could divide all animals into mammals, birds, fish, reptiles, amphibians, or arthropods. Alternatively, we could divide animals into carnivores, herbivores, or omnivores. Perhaps we could also divide animals into how they move such as swimming, flying, burrowing, walking, or slithering. There are potentially endless sets of categories that we could use.

Among categories, we divide things based on a set of characteristics. When categorizing animals, we might choose characteristics such as whether they have fur, hair, feathers, or scales. Maybe we look at the shape of their bodies or go more specific by looking at their teeth or how their feet are shaped. Once again, we choose there are potentially endless characteristics we could look for.

Analogies aside, the main point is that **in order for classification to work, we have to determine a set of categories into which we can class the things we see and the set of characteristics we use to make those classifications.** This allows us to then place everything that we see into one of the categories or perhaps say that it belongs to none of the categories. The more categories we have, the more specific we have to be. It's easier to say something is either an animal or not an animal but it's harder to say what group of animals an animal may belong to. However complicated, this classification allows us to not only recognize things that we have seen before, but also to place new things that we have never seen. Good image recognition models will perform well even on data they have never seen before (or any machine learning model, for that matter).

How do we Perform Image Recognition?

We do a lot of this image classification without even thinking about it. For starters, **we choose what to ignore and what to pay attention to**. This actually presents an interesting part of the challenge: picking out what's important in an image. We see everything but only pay attention to some of that so we tend to ignore the rest or at least not process enough information about it to make it stand out. Knowing what to ignore and what to pay attention to depends on our current goal. For example, if we were walking home from work, we would need to pay attention to cars or people around us, traffic lights, street signs, etc. but wouldn't necessarily have to pay attention to the clouds in the sky or the buildings or wildlife on either side of us. On the other hand, if we were looking for a specific store, we would have to switch our focus to the buildings around us and perhaps pay less attention to the people around us.

The same thing occurs when asked to find something in an image. **We decide what features or characteristics make up what we are looking for and we search for those, ignoring everything else.** This is easy enough if we know what to look for but it is next to impossible if we don't understand what the thing we're searching for looks like.

This brings to mind the question: how do we know what the thing we're searching for looks like? There are two main mechanisms: **either we see an example of what to look for and can determine what features are important from that (or are told what to look for verbally) or we have an abstract understanding of what we're looking for should look like already**. For example, if you've ever played "Where's Waldo?", you are shown what Waldo looks like so you know to look out for the glasses, red and white striped shirt and hat, and the cane. To the uninitiated, "Where's Waldo?" is a search game where you are looking for a particular character hidden in a very busy image. I'd definitely recommend checking it out. However, if we were given an image of a farm and told to count the number of pigs, most of us would know what a pig is and wouldn't have to be shown. That's because we've memorized the key characteristics of a pig: smooth pink skin, 4 legs

with hooves, curly tail, flat snout, etc. We don't need to be taught because we already know.

This logic applies to almost everything in our lives. **We learn fairly young how to classify things we haven't seen before into categories that we know based on features that are similar to things within those categories. If we come across something that doesn't fit into any category, we can create a new category**. For example, there are literally thousands of models of cars; more come out every year. However, we don't look at every model and memorize exactly what it looks like so that we can say with certainty that it is a car when we see it. We know that the new cars look similar enough to the old cars that we can say that the new models and the old models are all types of car.

By now, we should understand that image recognition is really image classification; we fit everything that we see into categories based on characteristics, or features, that they possess. We're intelligent enough to deduce roughly which category something belongs to, even if we've never seen it before. If something is so new and strange that we've never seen anything like it and it doesn't fit into any category, we can create a new category and assign membership within that. The next question that comes to mind is: how do we separate objects that we see into distinct entities rather than seeing one big blur?

The somewhat annoying answer is that it depends on what we're looking for. If we look at an image of a farm, do we pick out each individual animal, building, plant, person, and vehicle and say we are looking at each individual component or do we look at them all collectively and decide we see a farm? Okay, let's get specific then. Let's say we aren't interested in what we see as a big picture but rather what individual components we can pick out. How do we separate them all?

The key here is in contrast. **Generally, we look for contrasting colours and shapes; if two items side by side are very different colours or one is angular and the other is smooth, there's a good chance that they are different objects**. Although this is not always the case, it stands as a good starting point for distinguishing between objects.

Coming back to the farm analogy, we might pick out a tree based on a combination of browns and greens: brown for the trunk and branches and green for the leaves. Of course this is just a generality because not all trees are green and brown and trees come in many different shapes and colours but most of us are intelligent enough to be able to recognize a tree as a tree even if it looks different. We could find a pig due to the contrast between its pink body and the brown mud it's playing in. We could recognize a tractor based on its square body and round wheels. This is why colour-camouflage works so well; if a tree trunk is brown and a moth with wings the same shade of brown as tree sits on the tree trunk, it's difficult to see the moth because there is no colour contrast.

Another amazing thing that we can do is determine what object we're looking at by seeing only part of that object. This is really high level deductive reasoning and is hard to program into computers. This is one of the reasons it's so difficult to build a generalized artificial intelligence but more on that later. As long as we can see enough of something to pick out the main distinguishing features, we can tell what the entire object should be. For example, if we see only one eye, one ear, and a part of a nose and mouth, we know that we're looking at a face even though we know most faces should have two eyes, two ears, and a full mouth and nose.

Although we don't necessarily need to think about all of this when building an image recognition machine learning model, it certainly helps give us some insight into the underlying challenges that we might face. If nothing else, it serves as a preamble into how machines look at images. The main problem is that we take these abilities for granted and perform them without even thinking but it becomes very difficult to translate that logic and those abilities into machine code so that a program can classify images as well as we can. This is just the simple stuff; we haven't got into the recognition of abstract ideas such as recognizing emotions or actions but that's a much more challenging domain and far beyond the scope of this course.

How do Machines Interpret Images?

The previous topic was meant to get you thinking about how we look at images and contrast that against how machines look at images. We'll see that there are similarities and differences and by the end, we will hopefully have an idea of how to go about solving image recognition using machine code.

Let's start by examining the first thought: we categorize everything we see based on features (usually subconsciously) and we do this based on characteristics and categories that we choose. The number of characteristics to look out for is limited only by what we can see and the categories are potentially infinite. This is different for a program as programs are purely logical. **As of now, they can only really do what they have been programmed to do which means we have to build into the logic of the program what to look for and which categories to choose between**.

This is a very important notion to understand: as of now, machines can only do what they are programmed to do. If we build a model that finds faces in images, that is all it can do. It won't look for cars or trees or anything else; it will categorize everything it sees into a face or not a face and will do so based on the features that we teach it to recognize. **This means that the number of categories to choose between is finite, as is the set of features we tell it to look for**. We can tell a machine learning model to classify an image into multiple categories if we want (although most choose just one) and for each category in the set of categories, we say that every input either has that feature or doesn't have that feature. **Machine learning helps us with this task by determining membership based on values that it has learned rather than being explicitly programmed** but we'll get into the details later.

Often the inputs and outputs will look something like this:

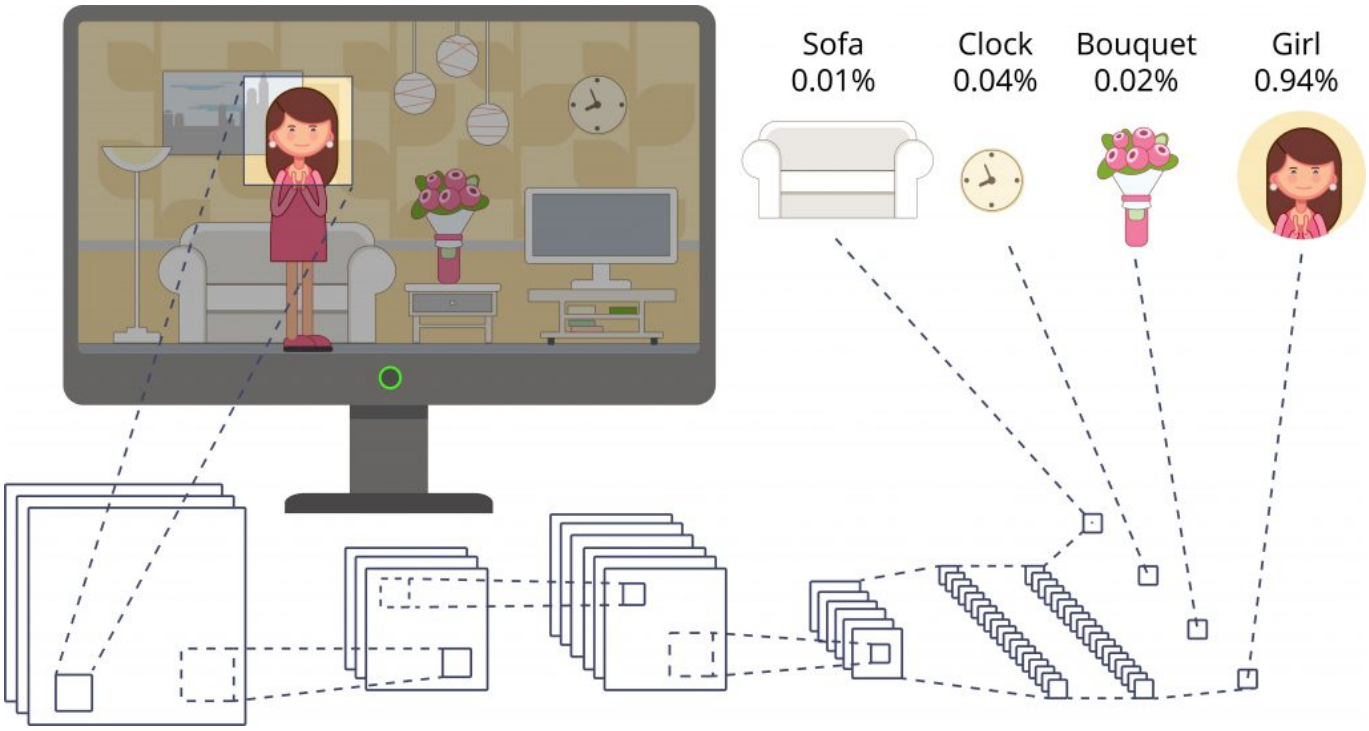
Input: [1 1 0 0 0 1 0 0 1 0]

Output: [0 0 1 0 0]

In the above example, we have 10 features. A 1 means that the object has that feature and a 0 means that it does not so this input has features 1, 2, 6, and 9 (whatever those may be). We can 5 categories to choose between. A 1 in that position means that it is a member of that category and a 0 means that it is not so our object belongs to category 3 based on its features. **This form of input and output is called one-hot encoding and is often seen in classification models**. Realistically, we don't usually see exactly 1s and 0s (especially in the outputs). We should see numbers close to 1 and close to 0 and these represent certainties or percent chances that our outputs belong to those categories. For example, if the above output came from a machine learning model, it may look something more like this:

[0.01 0.02 0.95 0.01 0.01]

This means that there is a 1% chance the object belongs to the 1st, 4th, and 5th categories, a 2% change it belongs to the 2nd category, and a 95% chance that it belongs to the 3rd category. It can be nicely demonstrated in this image:

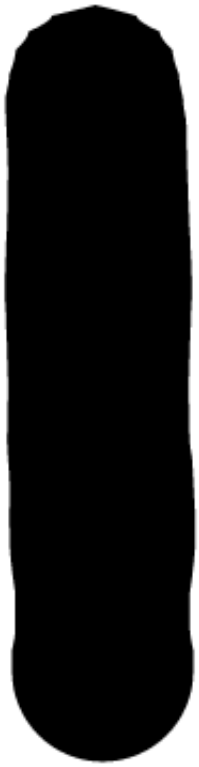


How do Machines Interpret Images?

This provides a nice transition into how computers actually look at images. To a computer, it doesn't matter whether it is looking at a real-world object through a camera in real time or whether it is looking at an image it downloaded from the internet; it breaks them both down the same way. **Essentially, in image is just a matrix of bytes that represent pixel values.** When it comes down to it, all data that machines read whether it's text, images, videos, audio, etc. is broken down into a list of bytes and is then interpreted based on the type of data it represents.

For images, each byte is a pixel value but there are up to 4 pieces of information encoded for each pixel. Grey-scale images are the easiest to work with because each pixel value just represents a certain amount of "whiteness". Because they are bytes, values range between 0 and 255 with 0 being the least white (pure black) and 255 being the most white (pure white). Everything in between is some shade of grey. With colour images, there are additional red, green, and blue values encoded for each pixel (so 4 times as much info in total). Each of those values is between 0 and 255 with 0 being the least and 255 being the most. If an image sees a bunch of pixels with very low values clumped together, it will conclude that there is a dark patch in the image and vice versa.

Below is a very simple example. An image of a 1 might look like this:



And have this as the pixel values:

[[255, 255, 255, 255, 255],

[255, 255, 0, 255, 255],

[255, 255, 0, 255, 255],

[255, 255, 0, 255, 255],

[255, 255, 255, 255, 255]]

This is definitely scaled way down but you can see a clear line of black pixels in the middle of the image data (0) with the rest of the pixels being white (255).

Images have 2 dimensions to them: height and width. These are represented by rows and columns of pixels, respectively. In this way, **we can map each pixel value to a position in the image matrix (2D array so rows and columns)**. Machines don't really care about the dimensionality of the image; most image recognition models flatten an image matrix into one long array of pixels anyway so they don't care about the position of individual pixel values. Rather, they care about the position of pixel values relative to other pixel values. They learn to associate positions of adjacent,

similar pixel values with certain outputs or membership in certain categories. In the above example, a program wouldn't care that the 0s are in the middle of the image; it would flatten the matrix out into one long array and say that, because there are 0s in certain positions and 255s everywhere else, we are likely feeding it an image of a 1. The same can be said with coloured images. If a model sees pixels representing greens and browns in similar positions, it might think it's looking at a tree (if it had been trained to look for that, of course).

This is also how image recognition models address the problem of distinguishing between objects in an image; they can recognize the boundaries of an object in an image when they see drastically different values in adjacent pixels. **A machine learning model essentially looks for patterns of pixel values that it has seen before and associates them with the same outputs**. It does this during training; we feed images and the respective labels into the model and over time, it learns to associate pixel patterns with certain outputs. If a model sees many images with pixel values that denote a straight black line with white around it and is told the correct answer is a 1, it will learn to map that pattern of pixels to a 1.

This is great when dealing with nicely formatted data. If we feed a model a lot of data that looks similar then it will learn very quickly. The problem then comes when an image looks slightly different from the rest but has the same output. Consider again the image of a 1. It could be drawn at the top or bottom, left or right, or center of the image. It could have a left or right slant to it. It could look like this: 1 or this l. This is a big problem for a poorly-trained model because it will only be able to recognize nicely-formatted inputs that are all of the same basic structure but there is a lot of randomness in the world. We need to be able to take that into account so our models can perform practically well. **This is why we must expose a model to as many different kinds of inputs as possible so that it learns to recognize general patterns rather than specific ones**. There are tools that can help us with this and we will introduce them in the next topic.

Hopefully by now you understand how image recognition models identify images and some of the challenges we face when trying to teach these models. Models can only look for features that we teach them to and choose between categories that we program into them. To machines, images are just arrays of pixel values and the job of a model is to recognize patterns that it sees across many instances of similar images and associate them with specific outputs. We need to teach machines to look at images more abstractly rather than looking at the specifics to produce good results across a wide domain. Next up we will learn some ways that machines help to overcome this challenge to better recognize images.

What Tools Help us Solve Image Recognition?

Processing an entire image is a lot of work, even for a computer. Generally speaking, we only use small images in machine learning models (at least the simple ones) but even a small image is a lot of pixels and we usually feed thousands of images in at a time. For example, we will use MNIST images that are 28 x 28 pixels. This is quite tiny for an image but that's still 784 pixels and we examine tens of thousands of images during training and testing. These images are black and white so it would take even more processing power to examine colour images.

The problem lies not just in processing that much data, but in finding consistent patterns in all that data. **It's easy for a model to get confused by all the extra noise in an image and to lose sight of what's really important**. Image recognition models have two mechanisms for dealing with these issues: making the images smaller and more abstract. It does through the use of image convolutions and max pooling. Once it has the smaller more abstract images, it can process them and make a prediction.

A convolution is a way to apply a filter to an image to distort and shrink it. This doesn't technically have anything to do with machine learning; image convolutions are used in all sorts of image filters such as applying blurs and detecting edges. However, **convolutional neural networks get their name from the fact that one of the layers consists of neurons that apply convolutions to the images they receive as input**. Although CNNs are not the only ways to solve image recognition, they have shown great success.

The way a convolution works is by running a smaller matrix called a kernel over the matrix, averaging out the products of the image pixel values with the kernel values, and mapping that average to new pixels. By the end, we have a different set of pixel values and a smaller image. We actually do this several times to produce a bunch of smaller images with the filter applied in different ways. Each output is called a filter and all of the filters will look different, even though they start out as the same image.

Now that's kind of difficult to visualize so here's a very simple example of an image, a kernel, and the result of applying the kernel just to the first 9×9 matrix:

Image:

[[0, 0, 0, 0, 0],

[0, 0, 255, 0, 0],

[0, 0, 255, 0, 0],

[0, 0, 255, 0, 0],

[0, 0, 0, 0, 0]]

Kernel:

[[0, -1, 0],

[-1, 5, -1],

[0, -1, 0]]

Result:

[[, ,],

[, 255,],

[, ,]]

Now this might now be quite what we would expect but it's a weird case because we are applying the convolution to the corner of the image. The way the above example works is that we multiply each pixel of the first 9×9 matrix with the pixel value of the kernel. We then sum up the results and replace the center pixel in the resultant image with that average. In this way, **we average out the pixel values and shrink the image**. We run the kernel over the entire image until we get a series of smaller filters that look quite different from the original image.

Different kernels will have different values within the matrix depending on the kind of effect they want to achieve. Convolutional layers in Tensorflow will have a predetermined kernel to apply so we don't have to create it ourselves. Generally we use a 3×3 or a 5×5 kernel and choose a stride length. That tells the kernel how many pixels to skip over when applying the next filter.

The next step is often to apply a max pool although this is not always necessary. **Max pooling takes a predetermined matrix size, finds the highest pixel value within the image in that matrix, and then gets rid of the other pixel values**. Typically, we do a 2×2 max pool. This cuts down the image size further and focuses on the stand-out pixels. The general goal is to eliminate unnecessary noise in the image. Again, it's difficult to visualize so let's see a quick example:

Image:

[[5, 10, 15, 10],

[15, 20, 10, 12],

[13, 16, 9, 8],

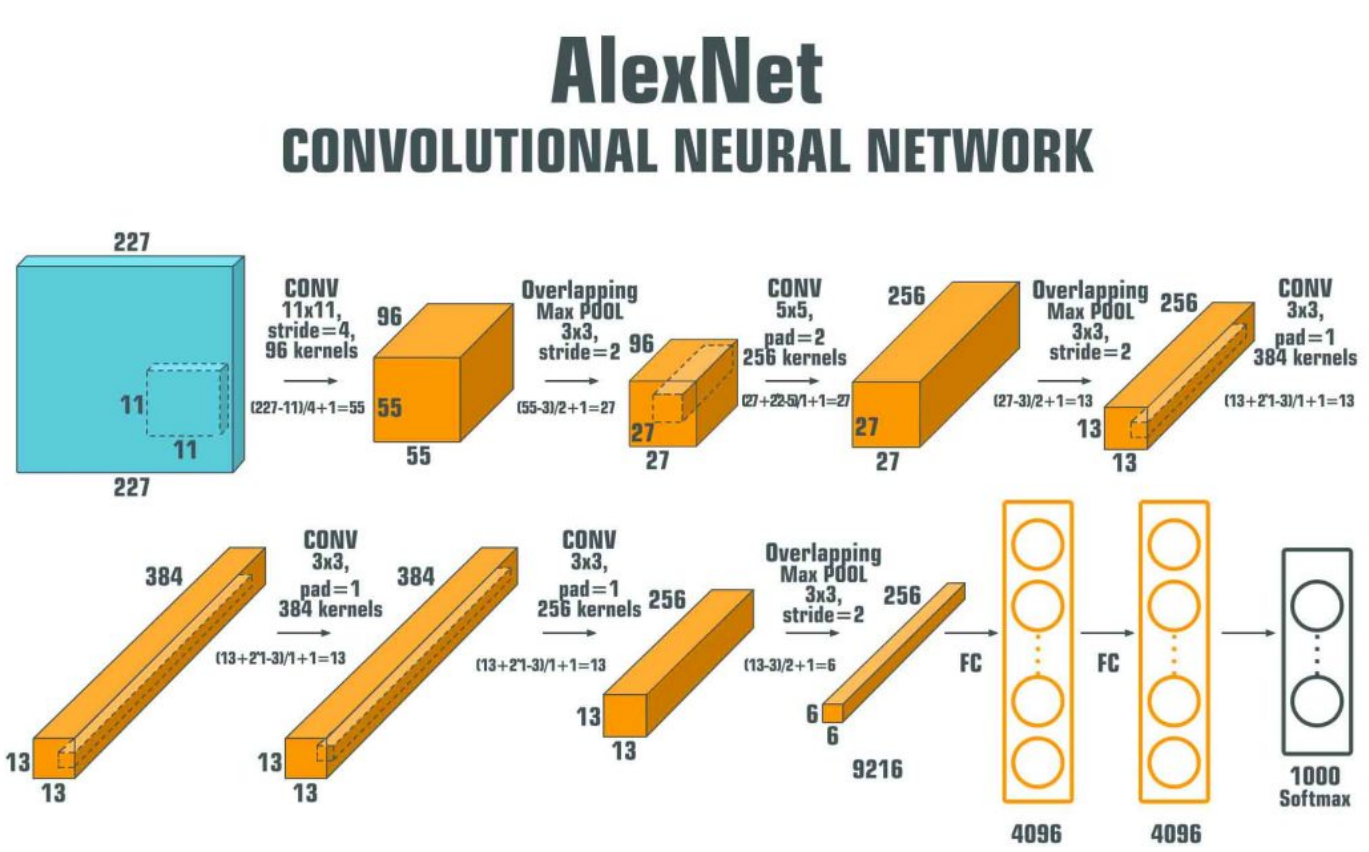
[15, 19 , 25, 20]]

Final result:

[[20, 15],

[19, 25]]

By the end of the convolution and max pool, we get a series of smaller, distorted, more abstract images that the model can more easily process. These usually bear some resemblance to the original digits but wouldn't be recognizable to us. Remember, the image recognition model does not see the images fed into it as specific numbers. Rather it searches for pixel patterns so it doesn't matter that the model is processing images that we wouldn't necessarily recognize as images. Bigger networks will have multiple combinations of these but the network that we will build will only need one convolution. Because we are trying to accomplish a fairly simple task, we won't need anything too complex. This image below nicely sums up what the end results of a series of convolutions and max pooling would look like:



MNIST

Now that we understand, roughly speaking, how image recognition works and what we can do to make the process easier, let's apply our knowledge toward building our model. **Our goal will be to build a model that can recognize handwritten digits**. We should take a second to get to know our data set before launching in, however. Lucky for us, there is a free, open-source dataset called MNIST that contains exactly what we need.

MNIST stands for the Modern National Institute of Standards of Technology. Now, we're not really interested in the institute but we are interested in the dataset that made MNIST famous. The dataset is used to train image recognition models and it serves as a starting point for models everywhere. There is an ongoing national competition to see who can produce the best results. The highest accuracy achieved so far is 99.79% achieved by The Parallel Computing Center using an ensemble of 5 CNNs together.

The dataset itself consists of 70000 labelled images of handwritten digits numbered 0 – 9. Each of the images is 28 x 28 pixels so 784 in total and is black and white (no colour). Each image has a corresponding label (0 – 9) and the whole data set is divided into 60000 training images and 10000 testing images. There is actually a newer version called EMNIST released in 2017 that had 280000 images but that's much more than we need. It's important that the images are handwritten as they provide a more accurate representation of what we might see when analyzing actual text. Each of the looks slightly different from the others but we want that; the more varied the data, the better.

Downloading MNIST

We want to import the MNIST dataset into our project. Of course, the first step is to start a project so go ahead and get a Python environment set up. I will be using a **Jupyter notebook run through Anaconda Navigator** although it doesn't matter too much what you use; as long as you have Tensorflow installed. This is assuming you have used Tensorflow before and have it installed. To start, **import tensorflow with this code:**

```
import tensorflow as tf
```

We can import mnist and load the data like so:

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

This loads the data and sorts it into train images (x_train), train labels (y_train), test images (x_test), and test labels (y_test). The images are matrices or 2d arrays with 28 rows of 28 pixels. Each pixel value is between 0 and 255 but **we want to normalize everything to between 0 and 1** so we will transform them like this:

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Go ahead and **examine the first image and label** by running the code:

```
print(x_train[0])
print(y_train[0])
```

This prints out the first image pixel values and the first label. **If we want to see what the image actually looks like, we can import matplotlib and use the imshow function** like this:

```
import matplotlib.pyplot as plt
plt.imshow(x_train[0],cmap='gray')
```

Next, we want to transform the images from 3 dimensions to 4. Right now, the images are in 3 dimensions as we have an array of images which are themselves arrays of arrays. We can achieve this by putting each of the individual pixel values in an array of its own. Transform the images like so:

```
x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]
```

This turns the 3 dimensions into 4. **Finally, we should shuffle the training images and divide train and test images into batches**. Shuffling helps the model train better as it forces the model

to keep on its toes, so to speak. If we feed in a bunch of the same digits in a row, the model gets really good at identifying that image, but not so good at identifying the others. The batches help us to train and test the model more efficiently by adjusting weights based on a small subset of images at a time. We can accomplish that with this code:

```
train_data = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(10000).ba
tch(32)
test_data = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(32)
```

Note that we don’t need to shuffle the test data because we perform no training during the testing phase so it doesn’t matter if the model sees a bunch of the same images in a row.

That’s all that we need to do to transform our data! It’s ready to be fed into the model so let’s build it.

Building the Model

We will use a subclass of the Keras model class. As Tensorflow 2.0 so closely integrates Keras, it is impractical to build a model that doesn't use Keras functionality at some point. Although we will build a custom training and testing loop, we will use a Keras model and layers. We can start with the imports:

```
from tensorflow.keras.layers import Conv2D, Flatten, Dense
from tensorflow.keras import Model
```

Now before we add the layers, let's talk about what they will do. **We will start with a Conv2D layer. This is a convolutional layer that applies a convolution to the images**. These layers are quite complex but we will only specify the number of filters (outputs), kernel size (width and height), and the type of activation function we will use. Conv2D layers need inputs in 4 dimensions, hence why we converted our original inputs into 4d arrays. For a full list of parameters, check out this link:

https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D

After that, we want a Flatten layer. This is used to turn our 4 dimensional input into a 1 dimensional array. It's nothing special so let's move on.

Our next couple of layers are going to be dense layers. These are your typical neural network layers that take inputs, multiply them by weights, add biases, and feed them through an activation function to either fire and produce an output or not fire. Although these are fairly complex behind the scenes, they are easy to set up with just the number of neurons (and also the number of outputs) and the activation function. For a full list of parameters, check out this link:

https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense

Now, to put it all together, we will need an initializer to create and assign the layers and a call function to feed inputs through the layers and produce an output. Let's write the code and then examine it:

```
class MNISTModel(Model):,

    def __init__(self):,
        super(MNISTModel, self).__init__()
        self.conv1 = Conv2D(32, 3, activation='relu')
        self.flatten = Flatten()
        self.dense1 = Dense(128, activation='relu')
        self.dense2 = Dense(10, activation='softmax')

    def call(self, x):
        x1 = self.conv1(x)
        x2 = self.flatten(x1)
        x3 = self.dense1(x2)
        return self.dense2(x3)
```

The initializer is fairly straightforward at first. We are just creating layers and assigning them to the model. The convolution layer has 32 filters (produces 32 outputs), has a kernel size of 3 (due to smaller image size), and employs a relu activation as it has proved to perform better than sigmoid and tanh functions. Remember, the convolution layer is used to transform the image before processing. The Flatten layer speaks for itself. The first dense layer is the first actual machine learning layer. It will feed the transformed images through the layer and into the next dense layer. The final dense layer has only 10 neurons because it will produce a series of encodings that denote which category the image belongs to. This is the one-hot encoding that we talked about earlier. If the highest value is in the 0th position, the digit is 0; if it's in the 1st position, the digit is 1; and so on. **We use a softmax function as we are essentially calculating probabilities at the end. Classification models often end with a softmax function**.

The call function will take some input x and feed it through each of the layers. Each layer will produce an output that acts as input into the next layer. By the end, we will have the processed output which will represent the model prediction. We create an instance like this:

```
model = MNISTModel()
```

Adding Training and Testing Functionality

Now that our model is built and initialized, we need optimizer and loss functions. We will use Keras functions for this. A sparse categorical crossentropy loss function is what we want as we are assigning the results to one of the 10 possible categories. We can set that up like this:

```
loss_function = tf.keras.losses.SparseCategoricalCrossentropy()
```

We will use an Adam optimizer as it adjusts its learning rate based on the current loss. If loss is decreasing quickly, it will use a higher learning rate. As it needs more fine tuning, it will decrease the learning rate. We can set that up like this:

```
optimizer = tf.keras.optimizers.Adam()
```

Next, we want a way to measure the train and test loss and accuracy. Once again, we can use Keras functions for this:

```
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')
```

We use the mean for the loss functions as we want the average loss per data point rather than the total loss for all points combined. As for the accuracy, we use sparse categorical accuracy as we are categorizing our results into one of 10 categories.

We have a way to keep track of the loss and accuracy but we need a way to actually train the model. This function will look very similar to the function we built for our model in part 1 of our Intro to Tensorflow course. Let's write the function and then talk about it:

```
@tf.function
def train_step(inputs, outputs):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = loss_function(outputs, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    train_loss(loss)
    train_accuracy(outputs, predictions)
```

This starts by taking in the inputs and the labels. The inputs are the images and the labels and the labels assigned. **The GradientTape acts as a way to initialize weight and bias variables and change them to minimize loss**. The variable predictions is the result of feeding the inputs through the model based on its current weights and biases. The loss variable is the difference between model outputs and the labels. Model outputs and labels will be of the same format. We can think of the gradients variable as being the suggestions made by the gradient tape based on the current values of weights and biases and the current loss. The next line actually applies the changes to the model variables to minimize loss. After that, we store the loss and accuracy.

We want a similar function for the test step although we don't need to do any training. We just need the predictions and loss and need to store the loss and accuracy values. We can do so like this:

```
@tf.function
def test_step(inputs, outputs):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = loss_function(outputs, predictions)

    test_loss(loss)
    test_accuracy(outputs, predictions)
```

Training and Testing the Model

Now that we have the means to train and test the model, we should write the actual train/test loop. **This loop will run the train and test steps and print out the current loss and accuracy values**. First we set the number of epochs; this is the number of times we run through the training data. We can do so like this:

```
epochs = 5
```

The loop can be run like this:

```
for epoch in range(epochs):
    for train_inputs, train_labels in train_data:
        train_step(train_inputs, train_labels)

    for test_inputs, test_labels in test_data:
        test_step(test_inputs, test_labels)

    template = 'Epochs: {}, Train loss: {}, Train accuracy: {}, Test loss: {} Test ac
curacy: {}'
    print(template.format(
        epoch + 1,
        train_loss.result(),
        train_accuracy.result(),
        test_loss.result(),
        test_accuracy.result(),
    ))

    train_loss.reset_states()
    train_accuracy.reset_states()
    test_loss.reset_states()
    test_accuracy.reset_states()
```

This runs through the training data 5 times. Each time, it runs through each of the training, then the testing steps. This stores the train loss and accuracy and the test loss and accuracy. It then prints out the results and resets the train and test loss and accuracy. This is a necessary step to reset the values stored. If you run the code, it will train and test the model and print out the progress. Go ahead and run the loop and watch the results. We should end up at over 98% accuracy! That's really quite good, especially considering our model was so simple. Anything over 95% is a very good model.

That's it! We have successfully built a custom machine learning model to recognize handwritten digits. It's performing very well even though the actual model only had 4 layers, one of which was a Flatten layer. This was done using a combination of custom Tensorflow and Keras. There is a simpler way, however. We could use pure Keras to build basically a similar model that produces results almost as good in just a dozen lines of code.

Intro to Keras

First, we should talk about what Keras is. **Keras is simply another machine learning library built on top of Tensorflow**. It's higher level than just Tensorflow meaning it's more abstract, easier to use, but slightly less customizable. It is built so that developers can more rapidly and easily construct and deploy machine learning models. Most users don't need quite the level of accuracy that can be achieved with a complex, well-thought-out, custom model.

The modular approach that Keras provides means that anyone can learn to use it. That's why Keras became so popular; the old Tensorflow was still somewhat difficult to use for the average developer. With the built-in functionality that Keras provides, it's easy to construct a model layer by layer and train and test the model with just a handful of lines of code. Besides its ease of use, the best part is that Keras comes with Tensorflow 2.0. In fact, we've already used it in the model that we built previously. In fact, it's impractical and even discouraged to build a Tensorflow model without using any Keras functionality.

Building a Model with Keras

Now we know that Keras provides a shortcut for building models without much sacrificing customizability and performance so let's give it a whirl! Our goal is to build a similar model to the one previously but just using Keras. It will be simpler but won't perform quite as well. Start up a new program and import Tensorflow:

```
import tensorflow as tf
```

Next, we want to import MNIST as we will be using that to train and test our model. Just like before we want to import and normalize the data:

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

We don't need the other transformation because, weirdly enough, we won't actually be using a convolutional layer. We can launch right into flattening the data to be fed into a dense layer. Besides the lack of a convolutional layer in this model, the other big difference is the addition of a dropout layer. **The dropout layer prevents overfitting by dropping a certain number of neurons. This helps to compensate for the lack of convolutions**. We can build our model like this:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax'),
])
```

The Dropout (0.2) drops 20% of neurons. Notice how for everything, we call tf.keras.whatever. That is to avoid adding the imports but feel free to add those and skip the tf.keras syntax. The sequential model for our purposes acts similar to the model that we build before. It specifically adds one layer after another so that the output of one automatically acts as an input to the next.

Next, we want to add the optimizer and loss functions. We only need one line for that:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['acc
uracy'])
```

Here we specify an adam optimizer, a sparse categorical crossentropy loss function, and add accuracy to the list of metrics to keep track of and print out. Finally, we can run the training and testing loops like this:

```
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test, verbose=2)
```

That’s all we need to run the train and test loops! We specify the training and testing inputs and the number of epochs. The verbose=2 just commands the program to print out various metrics. Go ahead and run the code. You should see that we get comparable accuracy to our other model (about 97%). That’s still really good, especially considering that the entire program was 13 lines of code (including the imports). Once we get to a high enough accuracy, we have to put in exponentially more work to achieve those last few percent. That’s why most people turn to using just Keras; the extra couple of percent accuracy is just not worth the time and knowledge it takes for most models. However, when we really need to be as accurate as possible, we can turn to a more custom solution like in our first model.

Summary

So that’s it from us! We have learned about image recognition, how machines perform this task, and built a digit recognition model a couple of different ways. We started by learning about how we recognize images. Next we compared that to how machines process images and some of the challenges that we face when trying to program a machine to do this. Next, we explored the MNIST data set to get a feel for what our data will look like. After that, we built a fairly simple digit recognition model that performed quite well. Finally, we built a similar model using just Keras. Although it didn’t quite perform as well as the first, it still performed very well and with much fewer lines of code. At the end of the day, it’s up to us to choose between performance and ease. If we are willing to put in the effort, we can achieve better results. If we need a quick and easy model, we can turn to Keras which still performs well but doesn’t quite provide the same level of customization.

From here, you could try to improve the model or start an entirely new one! If image recognition isn’t quite your thing, you could try different kinds of models aimed at solving different kinds of problems. Tensorflow and Keras have pretty much all of the tools you need. As it is open source, even if there is something that doesn’t exist yet in Tensorflow, you could be the first to implement it! Regardless, we hope you enjoyed the course and stay tuned for more content!