How to create a CNN with TensorFlow and Keras?

We'll first look at the concept of a classifier, CNNs themselves and their components. We then continue with a real Keras / Python implementation for classifying numbers using the MNIST dataset.

Here we will:

- Understand the basic concepts behind Convolutional Neural Networks.
- Learn how to implement a ConvNet classifier with TensorFlow and Keras.
- See how you can evaluate the CNN after it was trained.

What is a classifier?

Suppose that you work in the field of separating non-ripe tomatoes from the ripe ones. It's an important job, one can argue, because we don't want to sell customers tomatoes they can't process into dinner. It's the perfect job to illustrate what a human classifier would do.

Humans have a perfect eye to spot tomatoes that are not ripe or that have any other defect, such as being rotten. They derive certain characteristics for those tomatoes, e.g. based on color, smell and shape:

- If it's green, it's likely to be unripe (or: not sellable);
- If it smells, it is likely to be unsellable;
- The same goes for when it's white or when fungus is visible on top of it.

If none of those occur, it's likely that the tomato can be sold.
We now have *two classes*: sellable tomatoes and non-sellable tomatoes.
Human classifiers *decide about which class an object (a tomato) belongs to.*
The same principle occurs again in machine learning and deep learning.

Only then, we replace the human with a machine learning model. We're then using machine learning for *classification*, or for deciding about some "model input" to "which class" it belongs.

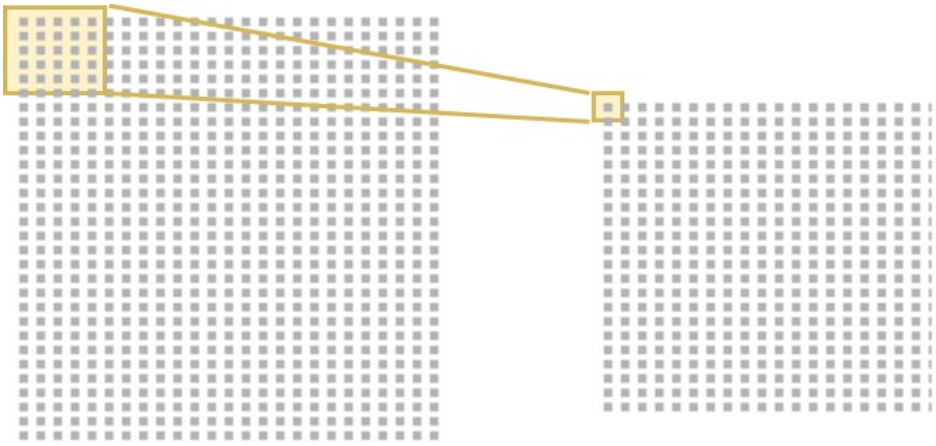
Especially when we deal with image-like inputs, Convolutional Neural Networks can be very good at doing that.

What is a Convolutional Neural Network?

Suppose that you have an image. In the case of the humans classifying tomatoes above this would be the continuous stream of image-like data that is processed by our brain and is perceived with our eyes. In the case of artificial classification with machine learning models, it would likely be input generated from a camera such as a webcam.

You wish to detect certain characteristics from the object in order to classify them. This means that you'll have to make a *summary* of those characteristics that gets more abstract over time. For example, with the tomatoes above, humans translate their continuous stream of observation into a fixed set of intuitive rules about when to classify a tomato as non-sellable; i.e., the three rules specified above.

Machine learning models and especially convolutional neural networks (CNNs) do the same thing.



Summarizing with a convolutional layer, as if you're using a magnifier

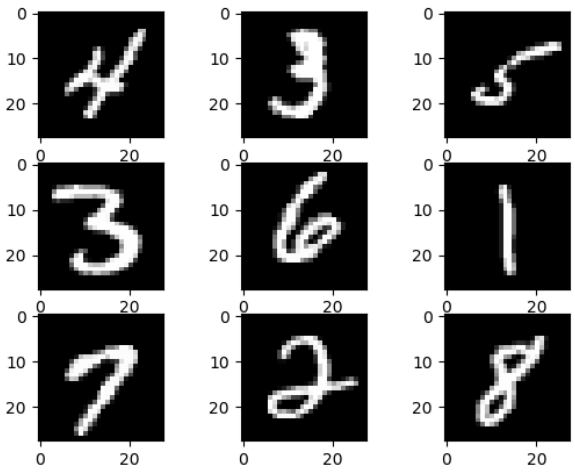
Please note, we left out the often-common layers like max pooling and batch normalization from the description above.

Hence, by creating an abstract summary with a Convolutional Neural Network, it's going to be possible to train a classifier that can *assign an object (the image) into a*

class. Just like humans do. We'll show it next with a simple and clear example using the MNIST dataset in Keras.

Dataset:

For the model that we'll create, we're going to use the MNIST dataset. The dataset, or the Modified National Institute of Standards and Technology database, contains many thousands of 28×28 pixel images of handwritten numbers, like this:



It's a very fine dataset for practicing with CNNs in Keras, since the dataset is already pretty normalized, there is not much noise and the numbers discriminate themselves relatively easily. Additionally, much data is available.

Hence, let's create our CNN!

Creating a CNN with TensorFlow 2.0 and Keras:

Software dependencies:

We always start with listing certain dependencies that you'll need to install before you can run the model on your machine.

Python, Tensorflow, Numpy, Matplotlib.

Model dependencies

Here, we'll first import the dependencies that we require later on:

```
import tensorflow
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
```

Obviously, we need **Keras** since it's the framework we're working with. We import the mnist dataset and benefit from the fact that it comes with Keras by default.

With respect to the layers, we will primarily use the **Conv2D** and **Dense** layers – we would say that these constitute the *core* of your deep learning model. The Conv2D layers will provide these *magnifier* operations, at two dimensions (like on the image above). That means: it slides with a small 2D box over a larger 2D box, being the image. It goes without saying that one can also apply 3D convolutional layers (for analyzing videos, with boxes sliding over a larger box) and 1D convolutional layers (for analyzing e.g. timeseries, with 'pixels' / points on a line sliding over the line).

We use the Dense layers later on for generating predictions (*classifications*) as it's the structure used for that.

However, we'll also use **Dropout**, **Flatten** and **MaxPooling2D**. A max pooling layer is often added after a Conv2D layer and it also provides a magnifier operation, although a different one. In the 2D case, it also slides with a box over the image (or in that case, the 'convolutional maps' generated by the first convolutional layer, i.e. the summarized image) and for every slide picks the maximum value for further propagation. In short, it generates an even stronger summary and can be used to induce sparsity when data is large.

Flatten connects the convolutional parts of the layer with the Dense parts. Those latter ones can only handle flat data, e.g. onedimensional data, but convolutional outputs are anything but onedimensional. Flatten simply takes all dimensions and concatenates them after each other.

With Dropout, we're essentially breaking tiny bits of the magnifier directly in front of it. This way, a little bit of noise is introduced into the summary during training.

Since we're breaking the magnifiers randomly, the noise is somewhat random as well and hence cannot be predicted in advance. Perhaps counterintuitively, it tends to improve model performance and reduce overfitting: the variance between training images increases without becoming too large. This way, a 'weird' slice of e.g. a tomato can perhaps still be classified correctly.

Model configuration

We'll next configure the CNN itself:

```
# Model configuration
img_width, img_height = 28, 28
batch_size = 250
no_epochs = 25
no_classes = 10
validation_split = 0.2
verbosity = 1
```

Since the MNIST images are 28×28 pixels, we define img_width and img_height to be 28. We use a batch size of 250 samples, which means that 250 samples are fed forward every time before a model improvement is calculated. We'll do 25 epochs, or passing *all* data 25 times (in batches of 250 samples, many batches per epoch), and have 10 classes: the numbers 0-9. We also use 20% of the training data, or 0.2, for validation during optimization. Finally, we wish to see as much output as possible, thus configure the training process to be verbose.

Loading and preparing MNIST data

We next load and prepare the MNIST data:

```
# Reshape data
input_train = input_train.reshape(input_train.shape[0],
img_width, img_height, 1)
input_test = input_test.reshape(input_test.shape[0], img_width,
img_height, 1)
input_shape = (img_width, img_height, 1)

# Parse numbers as floats
input_train = input_train.astype('float32')
```

```
input_test = input_test.astype('float32')

# Convert into [0, 1] range.
input_train = input_train / 255
input_test = input_test / 255
```

We first reshape our input data (the feature vectors). As you can see with the `input_shape`, it's the way your data must be built up to be handled correctly by the framework.

We then parse the numbers as floats, especially 32-bit floats. This optimizes the trade-off between memory and number precision over e.g. integers and 64-bit floats.

Finally, we convert the numbers to greyscale by dividing all (numeric!) image samples by 255. This allows them to be converted to the interval [0, 1] – or, greyscale. Why we do this? Because we don't care about the color of a number, only about the number itself.

Preparing target vectors with to_categorical

We next convert our target vectors, which are integers (0-9) into *categorical data*:

```
# Convert target vectors to categorical targets
target_train =
tensorflow.keras.utils.to_categorical(target_train, no_classes)
target_test = tensorflow.keras.utils.to_categorical(target_test,
no_classes)
```

Creating your model architecture

We then create the architecture of the model:

```
# Create the model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(no_classes, activation='softmax'))
```

We first define the `model` itself to be using the `Sequential` API, or, a stack of layers that together compose the Convolutional Neural Network.

We start off with a two-dimensional convolutional layer, or a Conv2D layer. It learns 32 filters, or feature maps, based on the data. The kernel, or the small image that slides over the larger one, is 3×3 pixels. As expected, we use the ReLU activation function for nonlinearity. In the first layer, we also specify the `input_shape` of our data, as determined by reshape.

The Conv2D layer is followed by a MaxPooling2D layer with a pool size of 2 x 2. That is, we further summarize the derivation of the Conv2D layer by applying max pooling with another image sliding over the filters that is 2×2 pixels. For every *slide*, it takes the maximum value (hence max pooling) within the 2×2 box and passes it on. Hence, each 2×2 = 4 pixel wide slide is turned into a one-pixel output. This greatly reduces memory requirements while keeping mostly intact your model performance.

Finally, before repeating the convolutional layers, we add Dropout. Dropout, as said, essentially breaks the magnifiers. Hence, a little bit of random noise is introduced during training. This greatly reduces the odds of overfitting. It does so by converting certain inputs to 0, and does so randomly. The parameter `0.25` is the dropout rate, or the number of input neurons to drop (in this case, 25% of the inputs is converted to 0).

Since we wish to summarize further, we repeat the Conv2D process (although learning *more* filters this time), the MaxPooling2D process and the Dropout process.

It's then likely that the summary is *general* enough to compare new images and assign them one of the classes 0-9. We must however convert the many filters learnt and processed to a *flat* structure before it can be processed by the part that can actually generate the predictions. Hence, we use the Flatten layer. Subsequently, we let the data pass through two Dense layers, of which the first is ReLU-activated and

the second one is Softmax-activated. Softmax activation essentially generates a *multiclass probability distribution*, or computes the probability that the item belongs to one of the classes 0-9, summed to 1 (the maximum probability). *This is also why we must have categorical data: it's going to be difficult to add an 11th class on the fly.*

Note that the number of output neurons is num_classes for the final layer for the same reason: since num_classes probabilities must be computed, we must have num_classes different outputs so that for every class a unique output exists.

Model compilation & starting training

We then compile the model and start the training by *fitting the data*:

```
# Compile the model
model.compile(loss=tensorflow.keras.losses.categorical_crossentr
opy,
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics=['accuracy'])

# Fit data to model
model.fit(input_train, target_train,
          batch_size=batch_size,
          epochs=no_epochs,
          verbose=verbosity,
          validation_split=validation_split)
```

We next *fit the data to the model*, or in plain English start the training process. We do so by feeding the training data (both inputs and targets), specifying the batch size, number of epochs, verbosity and validation split configured before.

Adding test metrics for testing generalization

Here you'll need to add metrics for *testing* as well. After training with the training and validation data, which essentially tells you something about the model's *predictive performance*, you also wish to test it for *generalization* – or, whether it works well when data is used that the model has never seen before. That's why you

created the train / test split in the first place. Now is the time to add a test, or an evaluation step, to the model – which executes just after the training process ends:

```
# Generate generalization metrics
score = model.evaluate(input_test, target_test, verbose=0)
print(f'Test loss: {score[0]} / Test accuracy: {score[1]}')
```

Final model

In the process, altogether you've created this:

```
import tensorflow
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D

# Model configuration
img_width, img_height = 28, 28
batch_size = 250
no_epochs = 25
no_classes = 10
validation_split = 0.2
verbosity = 1

# Load MNIST dataset
(input_train, target_train), (input_test, target_test) =
mnist.load_data()

# Reshape data
input_train = input_train.reshape(input_train.shape[0],
img_width, img_height, 1)
input_test = input_test.reshape(input_test.shape[0], img_width,
img_height, 1)
input_shape = (img_width, img_height, 1)

# Parse numbers as floats
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')

# Convert into [0, 1] range.
```

```
input_train = input_train / 255
input_test = input_test / 255

# Convert target vectors to categorical targets
target_train =
tensorflow.keras.utils.to_categorical(target_train, no_classes)
target_test = tensorflow.keras.utils.to_categorical(target_test,
no_classes)

# Create the model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(no_classes, activation='softmax'))

# Compile the model
model.compile(loss=tensorflow.keras.losses.categorical_crossentr
opy,
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics=['accuracy'])

# Fit data to model
model.fit(input_train, target_train,
          batch_size=batch_size,
          epochs=no_epochs,
          verbose=verbosity,
          validation_split=validation_split)

# Generate generalization metrics
score = model.evaluate(input_test, target_test, verbose=0)
print(f'Test loss: {score[0]} / Test accuracy: {score[1]}')
```

It's a complete Keras model that can now be run in order to find its performance and to see whether it works.

Open your terminal, preferably an Anaconda environment, and ensure that all the necessary dependencies are installed and are in working order.

You should see Keras starting up, running the training process in TensorFlow, and displaying the results of the epochs.