TASK

# Convolutional Neural Networks

Visit our website

## Introduction

### WELCOME TO THE CONVOLUTIONAL NEURAL NETWORK TASK!

In this task, we will cover the basics of convolutional neural networks, "ConvNets" or “CNNs”. Convolutional neural networks have had tremendous success especially in solving problems in computer vision, speech processing pipelines and, more recently, machine translation.

Get in touch

**Connect for support**

Remember that with our courses, you’re not alone! You can contact an expert code reviewer to get support on any aspect of your course.

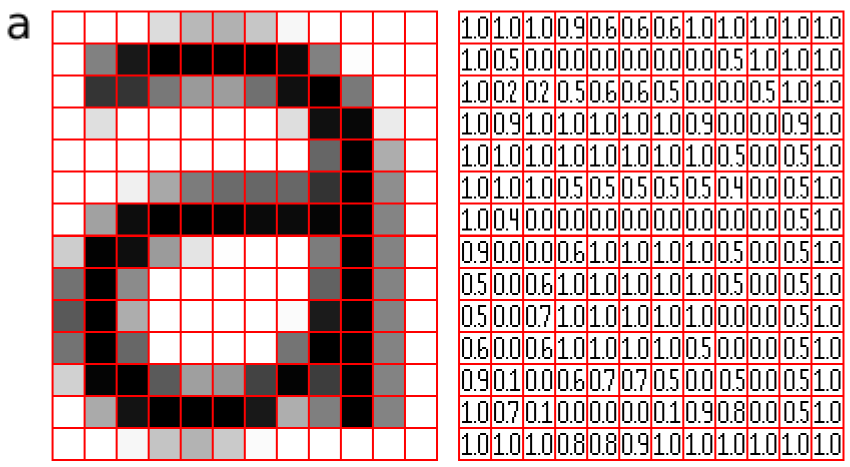
The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

## WHAT IS A CONVOLUTIONAL NEURAL NETWORK?

Convolutional neural networks are very similar to the neural networks we discussed in the previous task: they are formed by neurons that have parameters in the form of weights and biases that can be learned. But in regular neural networks, the hidden layers are connected directly to the input, while in convolutional neural networks the input is connected to the rest of the network in a special way, which we will explain shortly. The technique was developed for inputs that are an image, as a way to let the system 'see' the image in a way that mimics how humans recognise objects in images.

Just like how a toddler learns to recognise objects, we need to show an algorithm millions of pictures before it is able to generalise the input and make predictions for images it has never seen before. However, computers 'see' in a different way than we do. Essentially, a computer 'sees' every image as a 2-dimensional array of numbers known as pixels. The values of the pixels denote how bright and what colour each pixel should be.



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 1.0 | 1.0 | 0.9 | 0.6 | 0.6 | 0.6 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 1.0 | 0.5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 1.0 | 1.0 | 1.0 |
| 1.0 | 0.2 | 0.2 | 0.5 | 0.6 | 0.6 | 0.5 | 0.0 | 0.0 | 0.5 | 1.0 | 1.0 |
| 1.0 | 0.9 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.9 | 0.0 | 0.0 | 0.9 | 1.0 |
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.0 | 0.5 | 1.0 |
| 1.0 | 1.0 | 1.0 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.4 | 0.0 | 0.5 | 1.0 |
| 1.0 | 0.4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 1.0 |
| 0.9 | 0.0 | 0.0 | 0.6 | 1.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.0 | 0.5 | 1.0 |
| 0.5 | 0.0 | 0.6 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.0 | 0.5 | 1.0 |
| 0.5 | 0.0 | 0.7 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.5 | 1.0 |
| 0.6 | 0.0 | 0.6 | 1.0 | 1.0 | 1.0 | 1.0 | 0.5 | 0.0 | 0.0 | 0.5 | 1.0 |
| 0.9 | 0.1 | 0.0 | 0.6 | 0.7 | 0.7 | 0.5 | 0.0 | 0.5 | 0.0 | 0.5 | 1.0 |
| 1.0 | 0.7 | 0.1 | 0.0 | 0.0 | 0.0 | 0.1 | 0.9 | 0.8 | 0.0 | 0.5 | 1.0 |
| 1.0 | 1.0 | 1.0 | 0.8 | 0.8 | 0.9 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

*Representation of an image as a grid of pixels (**Source**)*

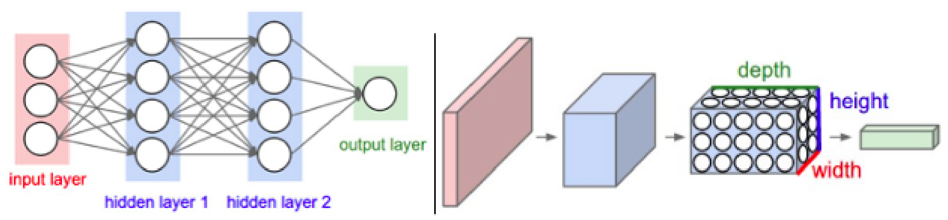
The human brain processes a huge amount of information the second we see an image. Each neuron works in its own receptive field and is connected to other neurons so that they cover the entire visual field. Some neurons respond to very basic elements, like horizontal or vertical lines, others to less basic elements, such as a combination of lines that make up an eye, and others respond to complex objects, such as an entire face or human body.

Just as each neuron responds to stimuli only in the restricted region of the visual field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well. The layers are arranged in such a way that they detect simple patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along. (Adapted from **"Convolutional Neural Networks, Explained"**).

## THE ARCHITECTURE OF CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks have a different architecture than regular Neural Networks. Regular Neural Networks transform input by putting them through a series of fully connected hidden layers, where each layer of neurons is connected to all the neurons in the layer before. The last fully-connected layer, the output layer, represents the predictions.

Convolutional Neural Networks are a bit different. First of all, the layers are organised in three dimensions. They have a dimension each for width, height, and depth, where the depth dimension corresponds to the colour channels red, green and blue. Secondly, the neurons in one layer do not connect to all the neurons in the next layer — only a small region of it. Lastly, the final output will be reduced to a single vector of probability scores, organised along the depth dimension.



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

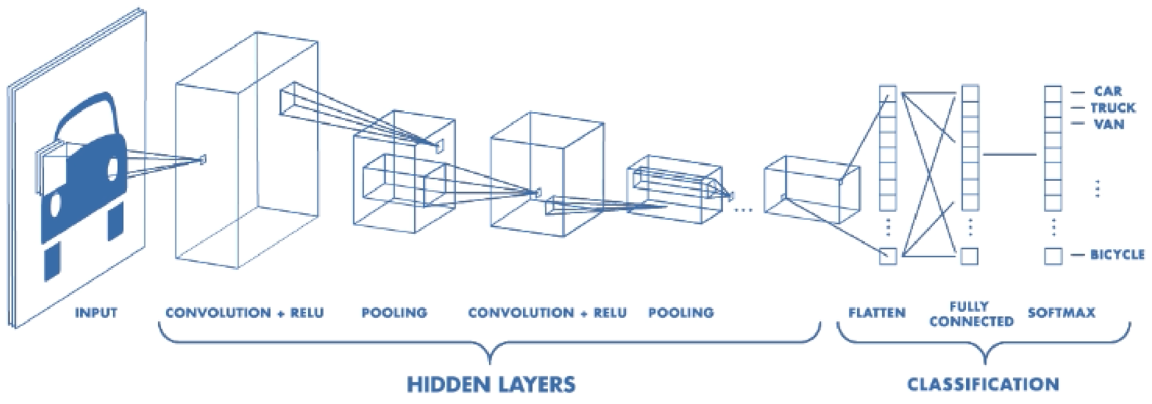
*Normal NN vs CNN (**Source**)*

CNNs have two components:

- **Feature extraction component**
  In this component, the network will identify and extract features that are necessary for classification. This is done using the earlier hidden layers, which themselves consist of convolutional layers and pooling layers — more on these shortly.

- **Classification component:**
  This component represents the later hidden layers and output layer where regular fully connected layers will serve as a classifier on top of the features

extracted by the feature extraction component. They will assign a probability for the object on the image of what the algorithm predicts it to be.



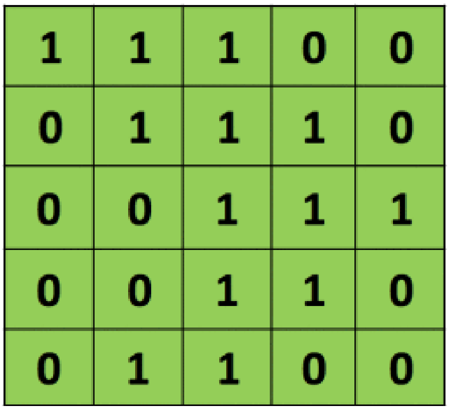
*The architecture of a CNN (**Source**)*

## FEATURE EXTRACTION

The benefit of using convolutional layers is that not every single input node needs to be fully connected to all the nodes in the next layer, as we saw with the regular feedforward network. Remember that all inputs were weighted in the regular feedforward network. This means that for large images, vast numbers of weights would need to be calculated, which is associated with many other issues. Instead, by mixing regions of the input with a small 2-dimensional array of weights called a **filter** or **kernel** in a mathematical process called **convolution**, only the weights in the filter need to be learnt. The output of the convolution is called a **feature map**, which is essentially the weighted sums of the input features around a point.

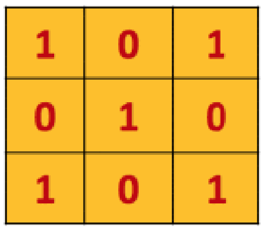
Let’s have a closer look at how this process works using an example:

Imagine we have an input image of size 5 x 5, i.e. it is 5 pixels highl and 5 pixels wide. As discussed previously, this is represented using a 2-dimensional array of numbers, as shown below. This is a very simple image and only has pixel values 0 and 1.



| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

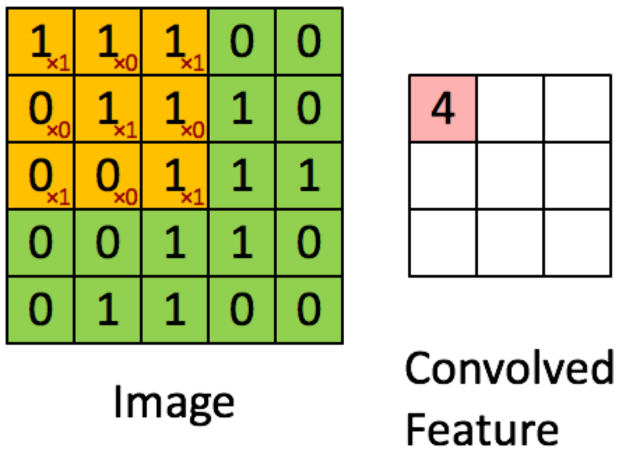
We also have another 3 x 3 matrix, our filter:



| | | |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

We slide this filter over the input matrix by moving it one pixel at a time from left to right and from top to bottom until the entire image has been covered by it at some point. The choice of how many pixels to shift the filter at a time is called **stride**. Here we used a stride of 1, which is the common stride size.

At each position, we perform element-wise multiplication between the filter and the portion of the input image that it covers. The output of this is then added up to produce a single number, which is then stored in a new matrix in a position corresponding to the position in the input image. The process performed at each step is called **convolution** and is demonstrated in the animation below.

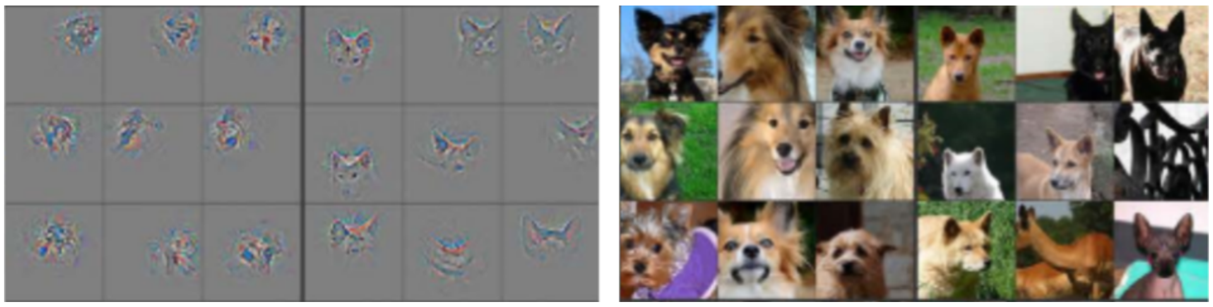


*The Convolution operation (**Source**)*

We have already mentioned that the output of a convolution is called a feature map. However, to be completely accurate, a feature map is actually the output of a convolution that has been passed through an **activation function**, which is needed to make the output non-linear for the backpropagation process, just like with regular neural networks. A common activation function used with CNNs is the **ReLU** activation function.

The reason the output of a convolution is called a feature map is that the process of convolution extracts features from the input image. These features are things like lines, which define the edges of objects, and blobs. Different features can be extracted from the same input image by applying a different filter, i.e. filters with different weights extract different features. This makes intuitive sense: the network adjusts these weights and thus during the training process it determines which features are important.

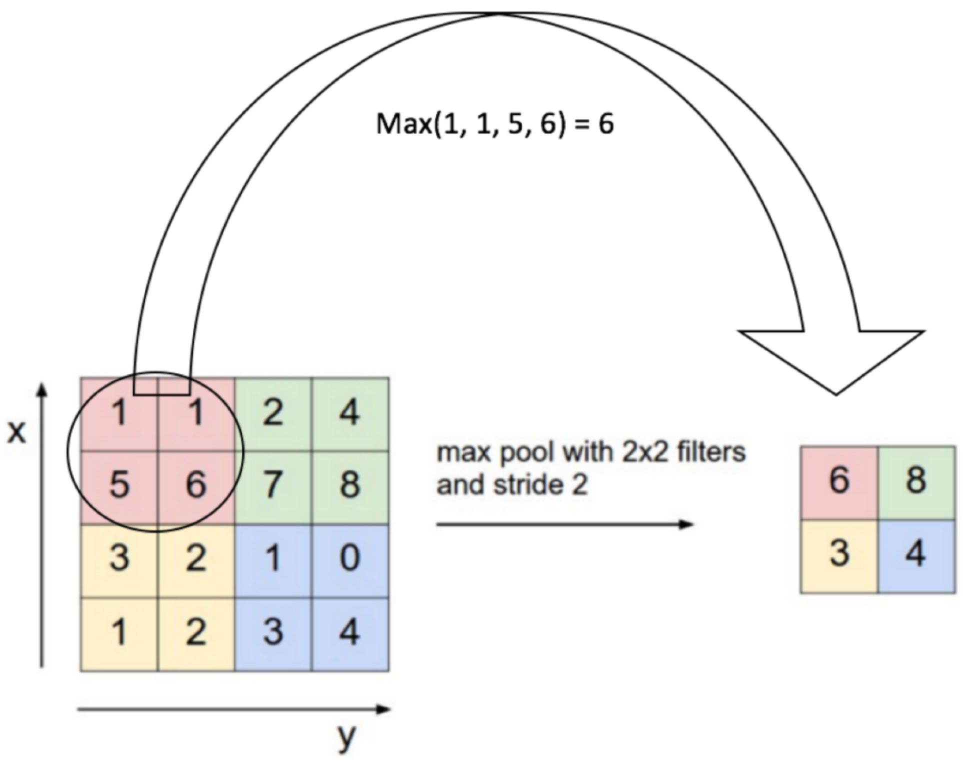
By performing multiple convolutions on an image, different features are extracted each time. Thus, by having many convolutions, the network is able to observe a wide range of features. These feature maps can be combined to produce a final output. For example, see the below image, which is extracted from a paper by Zeiler and Fergus (2014).


*A visualisation the combined features learnt by a fully trained model after five convolutional layers (**Source**)*

In our simple convolution example above, we performed convolution in two dimensions, whereas convolution is often applied in 3 dimensions, as the input image has dimensions of width, height, and depth (the number of colour channels).

This means that in the dog example, some of the filters may have learnt the lines which describe the edges of the dogs, while others may have learnt colour blobs (e.g., the dogs' eyes and noses). These are all then combined to produce the final feature map we see in the image.

In a feature map produced by a convolutional layer, there will be high values if a feature is present and low values if it is not. For example, in a dog nose detecting filter, if a dog nose is present, then the output of the convolution will have a high value, whereas if it is not present, it will have a low value. We say that the filter "activated" if a feature is present.  As you can imagine, as the filter moves over the image, the activation for the dog's nose will only occur in a very small part of the image. By making use of something called **pooling**, we can reduce the dimensionality of the problem by extracting the areas of the feature map in which the filter activated. The most commonly used type of pooling is called **max pooling**. In this type of pooling, we run another filter (which belongs to the pooling layer) over the feature map produced by the convolution and extract only the largest value at each position. This is illustrated in the image below, where a 2 x 2 pooling filter is applied along with a stride of 2. This reduces the dimensions of the feature map from 16 to 4 features.



*Max Pooling (**Source**)*

As an input moves through the layers in a CNN, its feature map becomes smaller and smaller. Sometimes we do not wish this to happen as it is possible to lose information about the corners of the image. This is because when sliding the convolution filter over the image, values in the middle fall within the filter many times, whereas values at the edges only fall in the filter a few times, and values in the corners are only included once. We can overcome this issue by making use of an approach called **padding**. The most common type of padding is **zero-padding**, in which extra pixels with the value zero are introduced to surround the image. Padding can improve the performance of the network and can be used to ensure that the filter and stride size fit the input well.

## CLASSIFICATION

Once an input image has moved through all of the convolutional and pooling layers in a CNN, the feature map will contain high-level features of the image. The final component of the CNN is the classification component, which takes these learnt features and uses them to make a classification.

This part of the network consists of **fully connected layers**, also called "dense" layers. Like in a regular neural network, in these layers every neuron is connected to every other neuron. This part of the network is responsible for deciding how to use the extracted features it has received to classify the input as one of a set of classes (those defined by the training labels).

Unlike the convolutional and pooling layers, the fully connected layers can only accept one-dimensional data. To convert our three-dimensional data to one-dimensional data we use the `flatten()` function in Python.

The final fully connected layer in the network will have the same number of neurons as there are classes. It is common practice to use the softmax activation function in this final layer to adjust the output of the layer to the range [0,1]. These outputs represent the probabilities of the input image belonging to each class.

## TRAINING

Training a CNN works in the same way as training a regular neural network, using backpropagation and gradient descent. However, the convolution operations need to be factored in, making the process more mathematically complex.

## IMPLEMENTATION OF A BASIC MODEL IN KERAS

In this section, we will demonstrate how to build a CNN using Keras, using the **MNIST** dataset. This famous dataset consists of 70 000 images of handwritten digits, in the range 0 - 9, which we will build a CNN to recognise.
First, we need to make sure we import everything we need:

```
# TensorFlow and Keras

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

#Helper libraries
import matplotlib.pyplot as plt
import numpy as np
```

We define a variable, **`num_classes`**, which we will use later, as well as defining the class names, so we can show these along with a selection of predicted images.

```
num_classes = 10

class_names    =    ["Zero","One","Two","Three",    "Four",    "Five","Six",
"Seven","Eight","Nine"]
```

We can now download the MNIST dataset and split it into a train and test set. There are 60 000 images in the train set and 10 000 images in the test set.

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

To get an idea of what the training images look like, we can have a look at one of these:

```
plt.figure()
plt.imshow(X_train[0])
plt.colorbar()
plt.grid(False)
plt.show()
```

It is necessary to scale our image data to the range [0,1]. We do this by dividing by 255.0. This also implicitly converts the type to float.

```
X_train = X_train / 255.0
X_test = X_test / 255.0
```

We now need to reshape the dataset to the shape that the model expects. Remember that images have three dimensions: width, height, and depth (which represents the colour channels). A depth of 1 corresponds to a grayscale image, while a depth of three corresponds to a colour image. The images in our dataset are grayscale images of size 28 x 28 (you can verify this using `.shape`). Therefore, we explicitly need to reshape our data so that Keras knows the image only has a depth of 1.

```
X_train = X_train.reshape(X_train.shape[0],28,28,1)
X_test = X_test.reshape(X_test.shape[0],28,28,1)
```

The images for both training and testing are now ready. The labels for the images now need to be converted from numbers in the range [0,9] to binary variables in a process called **"one-hot encoding"**. This means that an array will be created of length equal to the number of classes, with all values equal to zero except for a single element corresponding to the input label, which will have the value 1.

For example, for the label 5, the one-hot encoding is [0,0,0,0,1,0,0,0,0] and for the label 9 the encoding is [0,0,0,0,0,0,0,0,1]. We make use of Keras's `.to_categorical` function to do this.

```
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

We are now ready to define the model. You will notice that the early layers of the network consist of convolutional and pooling layers. This is the feature extraction portion of the network. You will notice that in the first convolutional layer we specify the input shape of the images; this is because this layer receives our images. The `kernel_size` is the size of our convolutional filters and the number before this, 32, indicates the number of filters that should be used. The `pool_size` is the size of the pooling filters and the `Dropout` layer is used to help prevent overfitting.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(28,28,1)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

The `Flatten` layer indicates the beginning of the classification part of the network, which is made up of dense layers (fully connected layers). Note that the final layer has the same number of neurons as the number of classes, and uses `softmax` activation. The final prediction of the network will be the class with the highest probability.

```
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Different CNN architectures can be built by stacking different numbers of convolutional and pooling layers in the feature extraction portion, and different numbers of dense layers in the classification portion. The filter size, pool size, and the number of filters can also be changed, amongst other hyperparameters. The more layers you stack, the 'deeper' the network becomes.

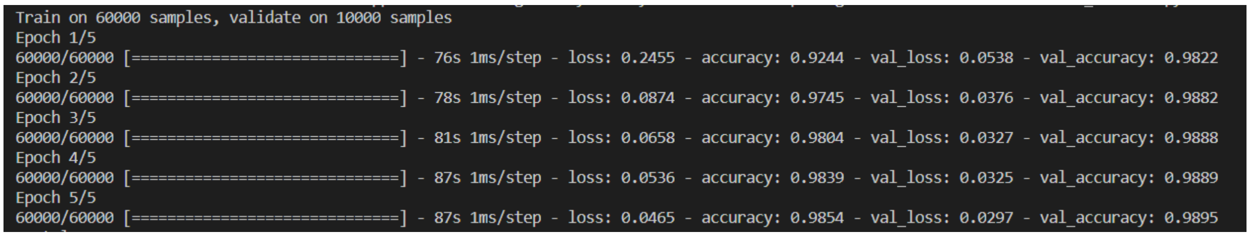
The model is then compiled. We will use the most common loss function for classification, `categorical_crossentropy`, along with the optimizer `adam`, which controls the learning rate. Finally, we wish to keep track of the accuracy score on the validation set (the test set), so we specify this.

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

We can now train our model using the `.fit` method. We specify the `batch_size` (which is how many images to push through the network before updating the weights) and the number of `epochs` to train for (how many times the entire dataset should be passed through the network). The more epochs we run our training for, the better the model will become at predicting the training data (up to a point). We also tell the network what data to use for validation — this lets us see how well our model is generalising to unseen data.

```
model.fit(X_train, y_train,
          batch_size=128,
          epochs=5,
          verbose=1,
          validation_data=(X_test, y_test))
```

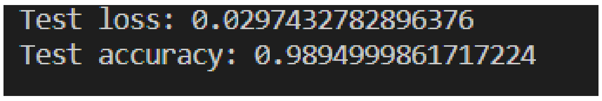
You can now watch the network training. You will see it achieves excellent accuracy in just a few epochs. This will take a bit of time! If your computer is quite slow, reduce the number of epochs to 3.



```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 76s 1ms/step - loss: 0.2455 - accuracy: 0.9244 - val_loss: 0.0538 - val_accuracy: 0.9822
Epoch 2/5
60000/60000 [==============================] - 78s 1ms/step - loss: 0.0874 - accuracy: 0.9745 - val_loss: 0.0376 - val_accuracy: 0.9882
Epoch 3/5
60000/60000 [==============================] - 81s 1ms/step - loss: 0.0658 - accuracy: 0.9804 - val_loss: 0.0327 - val_accuracy: 0.9888
Epoch 4/5
60000/60000 [==============================] - 87s 1ms/step - loss: 0.0536 - accuracy: 0.9839 - val_loss: 0.0325 - val_accuracy: 0.9889
Epoch 5/5
60000/60000 [==============================] - 87s 1ms/step - loss: 0.0465 - accuracy: 0.9854 - val_loss: 0.0297 - val_accuracy: 0.9895
```

We can also check what the performance was like on the whole validation (test) set:

```python
score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```



```
Test loss: 0.0297432782896376
Test accuracy: 0.9894999861717224
```

Congratulations — you have built your first CNN!

The code for this task was adapted from Keras' code for training on the MNIST model, found **here**.

## Instructions

The exercises in these tasks require users to run Jupyter Notebooks through the Anaconda environment. Please make sure that you download the Anaconda Distribution from **https://www.anaconda.com/distribution/**.

## Compulsory Task

Launch Jupyter Notebook via your Anaconda environment and upload the file named **Convolutional Neural Networks.ipynb** from your task folder and follow the instructions.

Submit your Task in .ipynb extension format.

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

### Things to look out for:

1. Make sure that you have installed and set up all programs correctly. You have set up **Dropbox** correctly if you are reading this, but **Python or Notepad++** may not be installed correctly.
2. If you are not using Windows, please ask a reviewer for alternative instructions.

## Completed the task(s)?

Ask an expert to review your work!

**Review work**

Rate us

## Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

**Click here** to share your thoughts anonymously.

**References:**

Zeiler, M. D., and Fergus, R. (2014), *Visualizing and understanding convolutional networks*, pp 818–833, European Conference on Computer Vision. Cham: Springer International Publishing.