Inteligencia Computacional

Práctica 1
Redes neuronales

Autor
Abdullah Taher Saadoon AL-Musawi

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
—

Granada, noviembre de 2019

Content Index

1- Description of Practice…… ……3
2- Hardware and Software of the work……..……..3
3- Introduction to Neural networks……..……..4
4- Types of NNs ……..……..5
    4.1 Convolutional NNs ……..……..5
    4.2 Multi-layer Perceptron……5
5- Description “MINST” DataSet ……7
6- Implementation ……..……..7
    6.1 BaseLine model with Multi-layer Perceptron ……..……..9
    6.2 Simple CNNs model ……..……..12
    6.3 Large CNNs model ……15
    6.4 increase depth of CNNs ……17
    6.5 increase depth of CNNs with less batch ……17
7- Summary of result ……..……..18
8- Conclusion ……..……..19
9- Bibliography…….…….. ……..…….. ……..19

1- Description

The purpose of this Practice is solving a pattern recognition problem using artificial neural networks, we will evaluate by using several types of neural networks to solve a Handwritten Digit Recognition problem. using MNIST database.

2- Hardware and Software

The following hardware has been used for training and execution of the neural network:

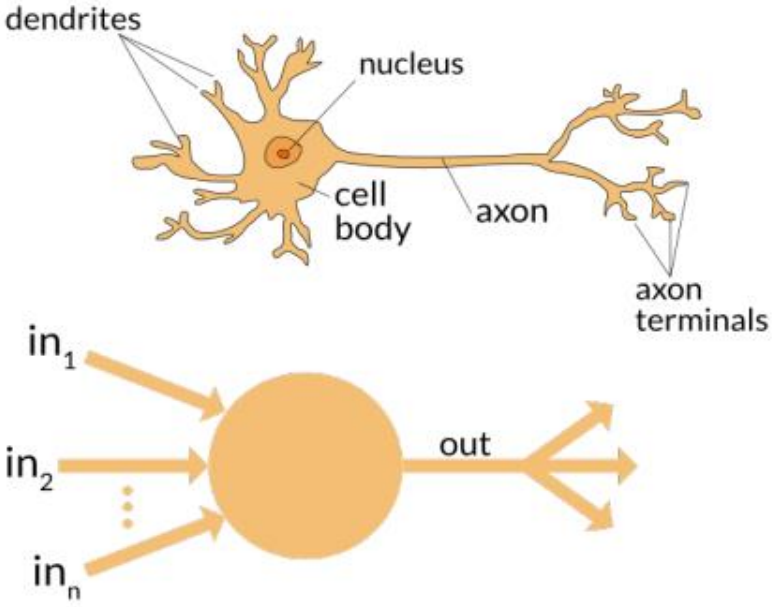
- **Processor:** Intel(R) Core(TM) i7-4702MQ CPU@ 2.20GHz 2.20GHz
- **Installed memory (RAM):** 8.00 GB

The following was used to develop the necessary software:

- **Programming language**: Python 3.6
- **Operating system**: Ubuntu 18.04 x64
- **IDE:** Visual studio Code
- **Frameworks and libraries**
  - Keras: https://keras.io/
  - Matplotlib: https://matplotlib.org/
  - MNIST: https://github.com/sorki/python-mnist
  - Numpy: https://www.numpy.org/
  - Tensorflow: https://www.tensorflow.org/?hl=es

3- Introduction to Neural networks

Neural networks are used as a method of deep learning, one of the many subfields of artificial intelligence. They were first proposed around 70 years ago as an attempt at simulating the way the human brain works, though in a much more simplified form. Individual 'neurons' are connected in layers, with weights assigned to determine how the neuron responds when signals are propagated through the network. Previously, neural networks were limited in the number of neurons they were able to simulate, and therefore the complexity of learning they could achieve.



But in recent years, due to advancements in hardware development, we have been able to build very deep networks, and train them on enormous datasets to achieve breakthroughs in machine intelligence.

4- Types of Neural Networks

4.1 Convolutional Neural Network

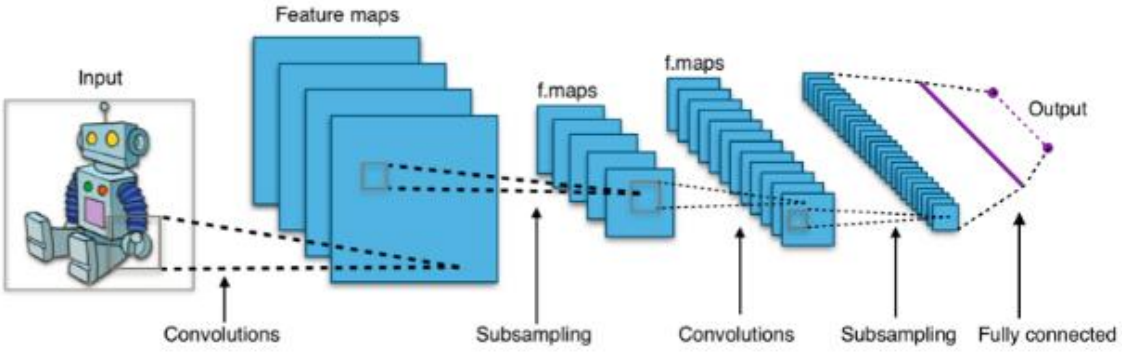
A convolutional neural network (CNN) uses a variation of the multilayer perceptrons. A CNN contains one or more than one convolutional layer. These layers can either be completely interconnected or pooled.

Before passing the result to the next layer, the convolutional layer uses a convolutional operation on the input. Due to this convolutional operation, the network can be much deeper but with much fewer parameters.

Due to this ability, convolutional neural networks show very effective results in image and video recognition, natural language processing, and recommender systems.

Convolutional neural networks also show great results in semantic parsing and paraphrase detection. They are also applied in signal processing and image classification.

CNNs are also being used in image analysis and recognition in agriculture where weather features are extracted from satellites like LSAT to predict the growth and yield of a piece of land. Here's an image of what a Convolutional Neural Network looks like.

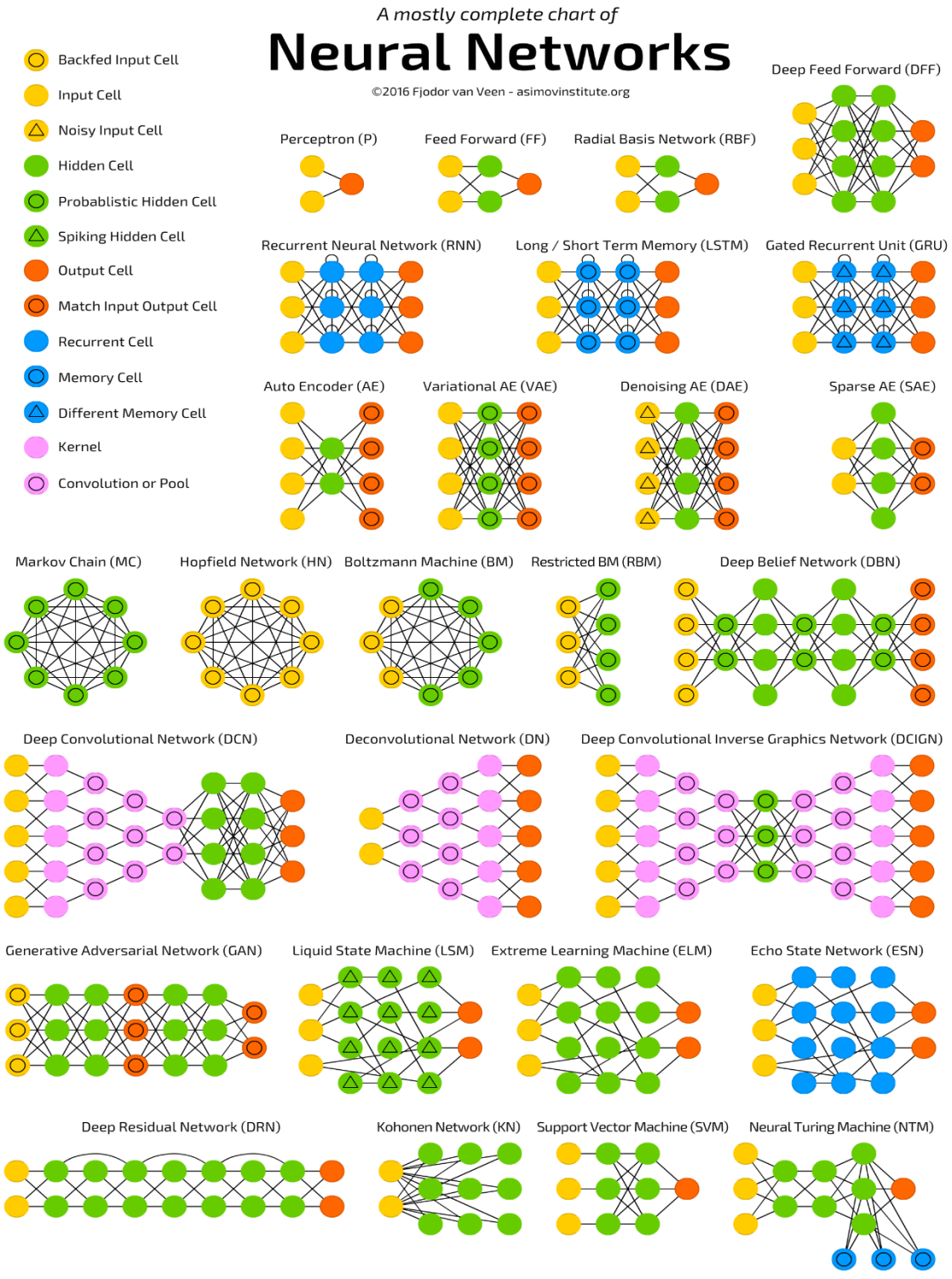


4.2 Multilayer Perceptron

A multilayer perceptron has three or more layers. It is used to classify data that cannot be separated linearly. It is a type of artificial neural network that is fully connected. This is because every single node in a layer is connected to each node in the following layer.A multilayer perceptron uses a nonlinear activation function (mainly hyperbolic tangent or logistic function).

Here's what a multilayer perceptron looks like.

Different types of neural networks use different principles in determining their own rules. There are many types of artificial neural networks, each with their unique strengths.



5- Description of 'MINST' DataSet

The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset.

It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9. The task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively. It is a widely used and deeply understood dataset and, for the most part, is “solved.” Top-performing models are deep learning convolutional neural networks that achieve a classification accuracy of above 99%, with an error rate between 0.4 %and 0.2% on the hold out test dataset.

6- Implementation

Now, we will create a simple program by python3 using Keras API,Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

The first thing we have to do it is Loading the database.

```
# example of loading the mnist dataset
from keras.datasets import mnist
from matplotlib import pyplot
# load dataset
(trainX, trainy), (testX, testy) = mnist.load_data()
```

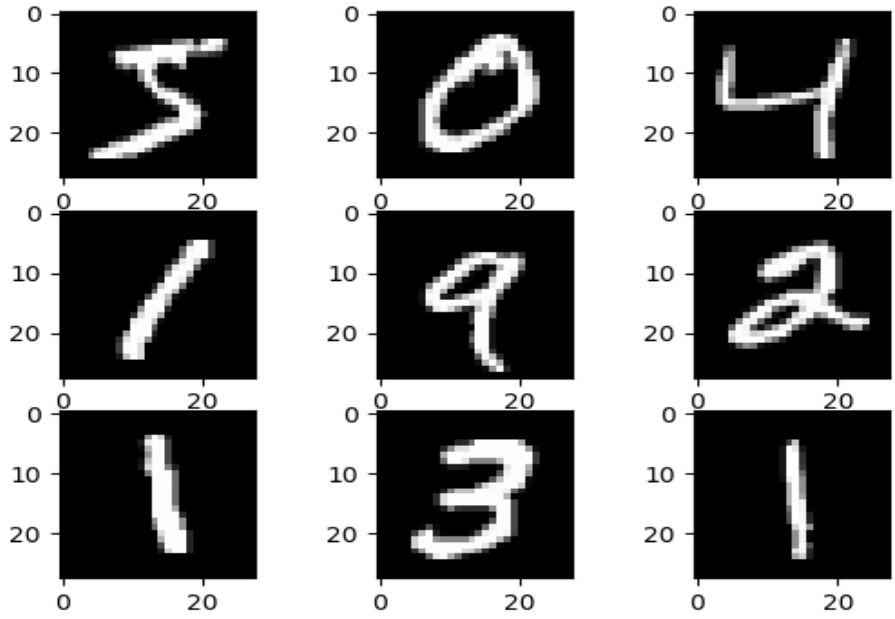
We can show the database summary using.

```
# summarize loaded dataset
print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))
print('Test: X=%s, y=%s' % (testX.shape, testy.shape))
```

```
(base) abdullah@abdullah-Lenovo-IdeaPad-Z510:~/Documents/Practical-IC/Practical-IC

Using TensorFlow backend.
Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)
```

A plot of the first nine images in the dataset is also we can create showing the natural handwritten nature of the images to be classified.



6.1 Baseline Model with Multi-Layer Perceptrons

We can get very good results using a very simple neural network model with a single hidden layer. In this section we will create a simple multi-layer perceptron model that achieves an error rate of 1.75%. We will use this as a baseline for comparing more complex convolutional neural network models.

The training dataset is structured as a 3-dimensional array of instance, image width and image height. For a multi-layer perceptron model we must reduce the images down into a vector of pixels. In this case the 28×28 sized images will be 784 pixel input values.

- **Prepare Pixel Data**

  We know that the pixel values for each image in the dataset are unsigned integers in the range between black and white, or 0 and 255.

  We do not know the best way to scale the pixel values for modeling, but we know that some scaling will be required.

  A good starting point is to normalize the pixel values of grayscale images, e.g. rescale them to the range [0,1]. This involves first converting the data type from unsigned integers to floats, then dividing the pixel values by the maximum value.

```
# flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape((X_train.shape[0],
num_pixels)).astype('float32')
X_test = X_test.reshape((X_test.shape[0], num_pixels)).astype('float32')

...
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
```

Finally, the output variable is an integer from 0 to 9. This is a multi-class classification problem. As such, it is good practice to use a one hot encoding of the class values, transforming the vector of class integers into a binary matrix.

We can easily do this using the built-in np_utils.to_categorical() helper function in Keras.

```
...
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

We are now ready to create our simple neural network model. We will define our model in a function. This is handy if you want to extend the example later and try and get a better score

```
...
# define baseline model
def baseline_model():
	# create model
	model = Sequential()
	model.add(Dense(num_pixels, input_dim=num_pixels,
kernel_initializer='normal', activation='relu'))
	model.add(Dense(num_classes, kernel_initializer='normal',
activation='softmax'))
	# Compile model
	model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
	return model
```

The model is a simple neural network with one hidden layer with the same number of neurons as there are inputs (784). A rectifier activation function is used for the neurons in the hidden layer.

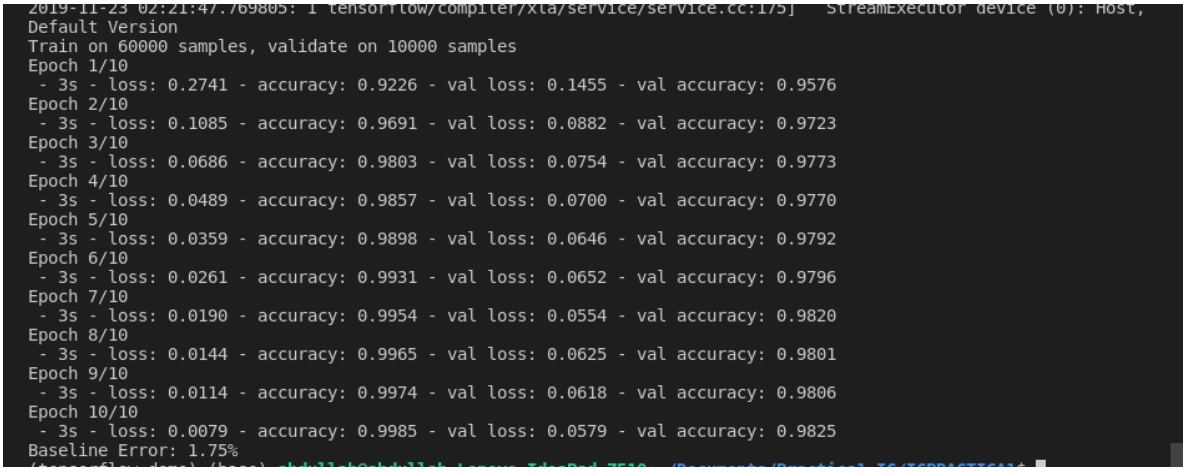
A softmax activation function is used on the output layer to turn the outputs into probability-like values and allow one class of the 10 to be selected as the model's output prediction. Logarithmic loss is used as the loss function (called categorical_crossentropy in Keras and the efficient ADAM gradient descent algorithm is used to learn the weights.

We can now fit and evaluate the model. The model is fit over 10 epochs with updates every 200 images. The test data is used as the validation dataset, allowing you to see the skill of the model as it trains. A verbose value of 2 is used to reduce the output to one line for each training epoch.

Finally, the test dataset is used to evaluate the model and a classification error rate is printed.

```
...
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10,
batch_size=200, verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

we see the output below, this very simple network defined in very few lines of code achieves a respectable error rate of 1.75%, and each epoch takes 3 seconds to run, the total execution time be 30 seconds.



6.2 Simple Convolutional Neural Network

In this section we will create a simple CNN that demonstrates how to use all of the aspects of a modern CNN implementation, including Convolutional layers, Pooling layers and Dropout layers.

- **Prepare Pixel Data**

After Loading database,we can load the images and reshape the data arrays to have a single color channel.becasue we know we know that that the images all have the same square size of 28×28 pixels, and that the images are grayscale.

```
# reshape dataset to have a single channel
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))
```

As before, it is a good idea to normalize the pixel values to the range 0 and 1 and one hot encode the output variables.

```
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Next, we define our neural network model

- **network architecture**

➔ The first hidden layer is a convolutional layer called a Convolution2D. The layer has 32 feature maps, which with the size of 5×5 and a rectifier activation function. This is the input layer, expecting images with the structure outline above [pixels][width][height].

➔ Next, we define a pooling layer that takes the max called MaxPooling2D. It is configured with a pool size of 2×2.

➔ The next layer is a regularization layer using dropout called Dropout. It is configured to randomly exclude 20% of neurons in the layer in order to reduce overfitting.

➔ Next is a layer that converts the 2D matrix data to a vector called Flatten. It allows the output to be processed by standard fully connected layers.

➔ next a fully connected layer with 128 neurons and rectifier activation function.

➔ Finally, the output layer has 10 neurons for the 10 classes and a softmax activation function to output probability-like predictions for each class.

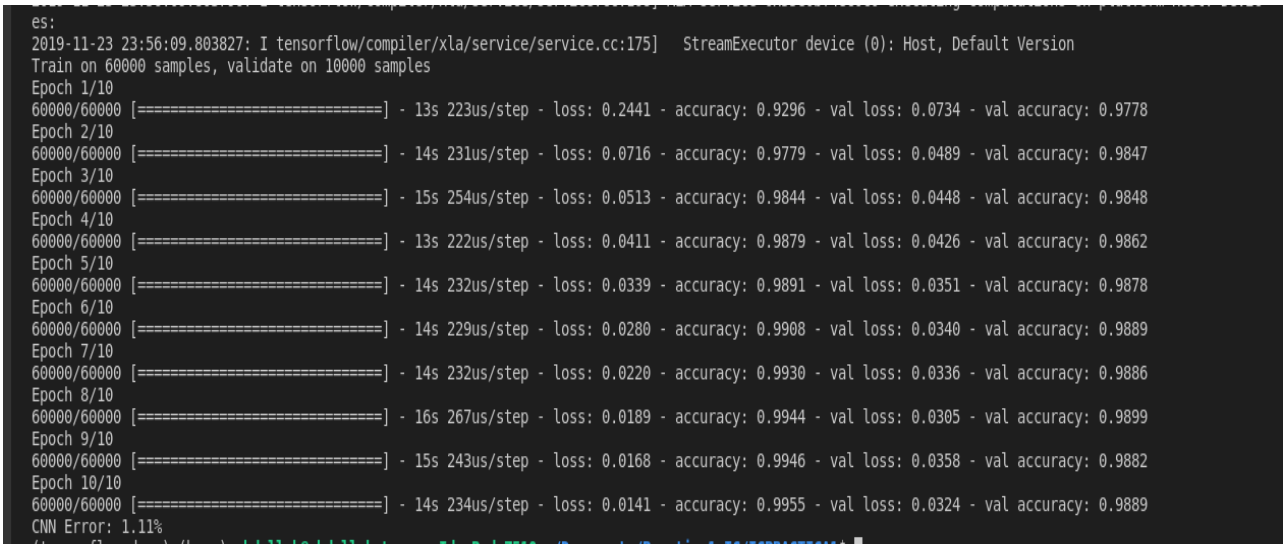
As before, the model is trained using logarithmic loss and the ADAM gradient descent algorithm.

```
...
def baseline_model():
	# create model
	model = Sequential()
	model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28), 
activation='relu'))
	model.add(MaxPooling2D(pool_size=(2, 2)))
	model.add(Dropout(0.2))
	model.add(Flatten())
	model.add(Dense(128, activation='relu'))
	model.add(Dense(num_classes, activation='softmax'))
	# Compile model
	model.compile(loss='categorical_crossentropy', 
optimizer='adam', metrics=['accuracy'])
	return model
```

We evaluate the model the same way as before with the multi-layer perceptron. The CNN is fit over 10 epochs with a batch size of 200.

```
...
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
epochs=10, batch_size=200, verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

Each Epoch may take between 13 and 16 seconds, totally Epochs may take about 150 seconds to run it, we can see that the network achieves an error rate of 1.11%, which is better than our simple multi-layer perceptron model above.



6.3 Larger Convolutional Neural Network

We import classes and function then load and prepare the data the same as in the previous CNN example.

his time we define a large CNN architecture with additional convolutional, max pooling layers and fully connected layers. The network topology can be summarized as follows.

- Convolutional layer with 30 feature maps of size 5×5.
- Pooling layer taking the max over 2*2 patches.
- Convolutional layer with 15 feature maps of size 3×3.
- Pooling layer taking the max over 2*2 patches.
- Dropout layer with a probability of 20%.
- Flatten layer.
- Fully connected layer with 128 neurons and rectifier activation.
- Fully connected layer with 50 neurons and rectifier activation.
- Output layer.

```
...
# define the larger model
def larger_model():
	# create model
	model = Sequential()
	model.add(Conv2D(30, (5, 5), input_shape=(1, 28, 28), 
activation='relu'))
	model.add(MaxPooling2D(pool_size=(2, 2)))
	model.add(Conv2D(15, (3, 3), activation='relu'))
	model.add(MaxPooling2D(pool_size=(2, 2)))
	model.add(Dropout(0.2))
	model.add(Flatten())
	model.add(Dense(128, activation='relu'))
	model.add(Dense(50, activation='relu'))
	model.add(Dense(num_classes, activation='softmax'))
	# Compile model
	model.compile(loss='categorical_crossentropy', 
optimizer='adam', metrics=['accuracy'])
	return model
```

the model is fit over 10 epochs with a batch size of 200.

```
...
# build the model
model = larger_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
epochs=10, batch_size=200)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Large CNN Error: %.2f%%" % (100-scores[1]*100))
```

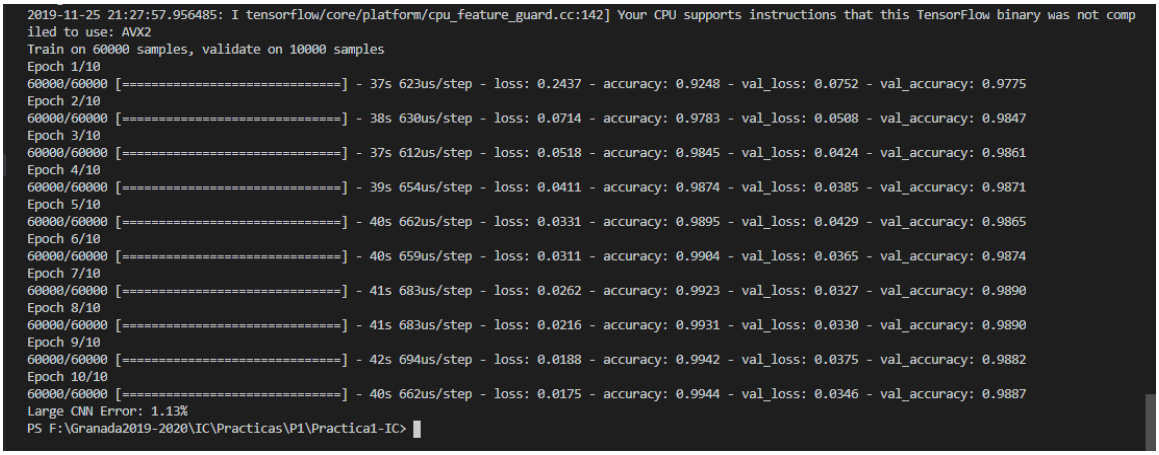
Each Epoch may take between 21 and 23 seconds, totally Epochs may take about 220 seconds to run it, we can see that the network achieves an error rate of 0.80 %, which is better than our simple multi-layer perceptron model above.

```
2019-11-25 17:54:15.195455: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this Tenso
rFlow binary was not compiled to use: AVX2
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 23s 387us/step - loss: 0.3800 - accuracy: 0.8852 - val_loss: 0.0839 - val_accura
cy: 0.9729
Epoch 2/10
60000/60000 [==============================] - 21s 349us/step - loss: 0.0919 - accuracy: 0.9726 - val_loss: 0.0539 - val_accura
cy: 0.9827
Epoch 3/10
60000/60000 [==============================] - 21s 354us/step - loss: 0.0684 - accuracy: 0.9782 - val_loss: 0.0386 - val_accura
cy: 0.9872
Epoch 4/10
60000/60000 [==============================] - 21s 354us/step - loss: 0.0546 - accuracy: 0.9830 - val_loss: 0.0365 - val_accura
cy: 0.9878
Epoch 5/10
60000/60000 [==============================] - 21s 357us/step - loss: 0.0459 - accuracy: 0.9856 - val_loss: 0.0302 - val_accura
cy: 0.9897
Epoch 6/10
60000/60000 [==============================] - 21s 343us/step - loss: 0.0413 - accuracy: 0.9872 - val_loss: 0.0278 - val_accura
cy: 0.9901
Epoch 7/10
60000/60000 [==============================] - 22s 365us/step - loss: 0.0364 - accuracy: 0.9886 - val_loss: 0.0275 - val_accuracy: 0.9912
Epoch 8/10
60000/60000 [==============================] - 21s 355us/step - loss: 0.0346 - accuracy: 0.9889 - val_loss: 0.0266 - val_accuracy: 0.9916
Epoch 9/10
60000/60000 [==============================] - 22s 368us/step - loss: 0.0309 - accuracy: 0.9901 - val_loss: 0.0281 - val_accuracy: 0.9904
Epoch 10/10
60000/60000 [==============================] - 22s 368us/step - loss: 0.0284 - accuracy: 0.9910 - val_loss: 0.0243 - val_accuracy: 0.9920
Large CNN Error: 0.80%
PS F:\Granada2019-2020\IC\Practicas\P1\Practica1-IC>
```

6.4 Increase depth Convolutional Neural Network

- Convolutional layer with 32 feature maps of size 3×3.
- Pooling layer taking the max over 2*2 patches.
- Convolutional layer with 64 feature maps of size 3×3.
- Convolutional layer with 64 feature maps of size 3×3.
- Pooling layer taking the max over 2*2 patches.
- Flatten layer.
- Output layer.
- the model is fit over 10 epochs with a batch size of 200.

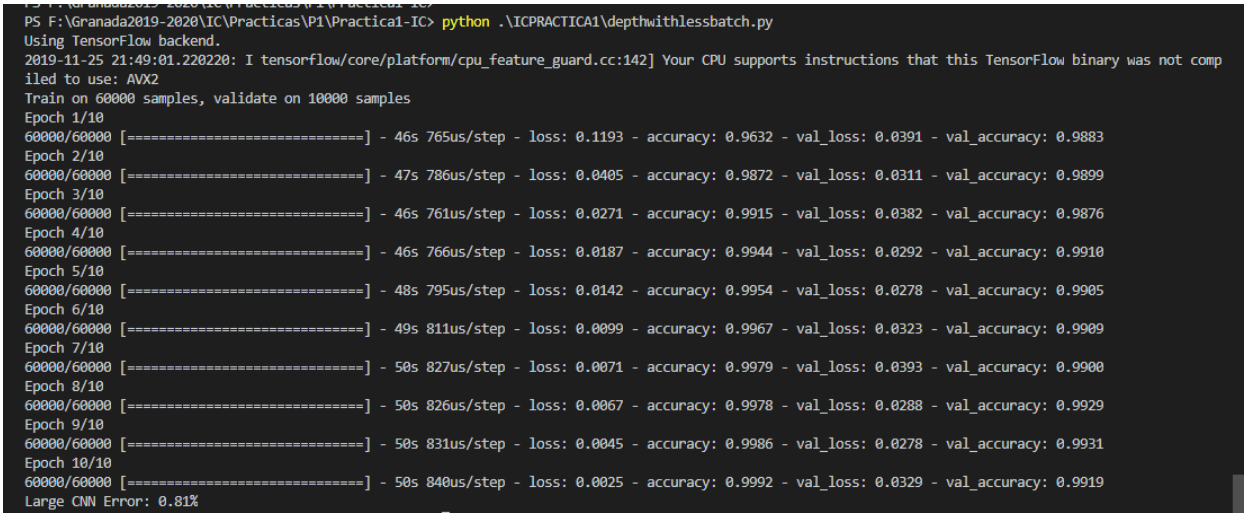
Each Epoch may take between 37 and 42 seconds, totally Epochs may take about 390 seconds to run it, we can see that the network achieves an error rate of 1.13 %, which is better than our simple multi-layer perceptron model above.



6.5 Increase depth Convolutional Neural Network with less batches size

- Convolutional layer with 32 feature maps of size 3×3.
- Pooling layer taking the max over 2*2 patches.
- Convolutional layer with 64 feature maps of size 3×3.
- Convolutional layer with 64 feature maps of size 3×3.
- Pooling layer taking the max over 2*2 patches.
- Flatten layer.
- Output layer.
- the model is fit over 10 epochs with a batch size of 28.

Each Epoch may take between 46 and 50 seconds, totally Epochs may take about 490 seconds to run it, we can see that the network achieves an error rate of 0.81 %, which is better than our simple multi-layer perceptron model above.



7- Summary of result

Models	Rate errors train	Accuracy train	Rate errors test	Accuracy test	Run Time
BaseLine	0.10%	99.36%	1.75%	93.97%	30s
Simple CNN	0.24%	99.15%	1.11%	96.62%	142s
Large CNN	0.41%	98.68%	0.80%	97.75%	215s
Increase depth	0.44%	98.49%	1.13%	95.46%	390s
More depth with less batch	0.11%	99.67%	0.81%	96.69%	495s

8- Conclusion

In this tutorial you successfully trained a neural network to classify the MNIST dataset with around 0.8% rate error with large CNNs, 1.11% simple CNNs with less run time and with BaseLine model, we achieved 1.75% rate error with less time than CNNs models,

So, we can use more complex network architectures involving convolutional layers to achieve less rate error and a good accuracy, Current state-of-the-art research achieves around 99% accuracy on this same problem.

9- Bibliography

1- https://www.digitalocean.com/community/tutorials/how-to-build-a-neural-network-to-recognize-handwritten-digits-with-tensorflow

2- https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/

3- https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

4- https://medium.com/@himanshubeniwal/handwritten-digit-recognition-using-machine-learning-ad30562a9b64

5- http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#4d4e495354

6- https://www.allaboutcircuits.com/news/artificial-neural-network-how-ANN-change-research-engineering/