IT8303

AI Human Interface

Lab 4  
Convolutional Neural Network (CNN)



What you will learn / do in this lab

1. Implement a simple Neural Network (NN)
2. Implement a Convolutional Neural Network (CNN)

TAble of  
Contents

1. Overview 1

Introduction CNN 1

Applications of CNN 1

2. Simple NN 2

MNIST problem 2

Data Preparation 3

Training and evaluation of model 4

3. CNN Model 7

Data Preparation 7

CNN training and Evaluation 7

# 1. Overview

In this practical we will use python with Keras to implement Convolutional Neural Networks (CNN). A popular demonstration of the capability of deep learning techniques is object recognition in image data.

## Introduction CNN

A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, pooling layers, fully connected layers and normalization layers.

CNN are able to handle image data or any data where there information in neighboring regions are more important that those further away.

## Applications of CNN

Applications of CNN includes:

* Image Recognition
* Video Analysis

# 2. Simple NN

In this section we will implement a simple NN model for the MNIST problem.

## MNIST problem

The MNIST problem is a dataset developed by Yann LeCun, Corinna Cortes and Christopher Burges for evaluating machine learning models on the handwritten digit classification problem.

The dataset was constructed from a number of scanned document dataset available from the National Institute of Standards and Technology (NIST). This is where the name for the dataset comes from, as the Modified NIST or MNIST dataset.

Images of digits were taken from a variety of scanned documents, normalized in size and centered. This makes it an excellent dataset for evaluating models, allowing the developer to focus on the machine learning with very little data cleaning or preparation required.

Each image is a 28 by 28 pixel square (784 pixels total). A standard spit of the dataset is used to evaluate and compare models, where 60,000 images are used to train a model and a separate set of 10,000 images are used to test it.

It is a digit recognition task. As such there are 10 digits (0 to 9) or 10 classes to predict. Results are reported using prediction error, which is nothing more than the inverted classification accuracy.

Excellent results achieve a prediction error of less than 1%.

## Data Preparation

The Keras deep learning library provides a convenience method for loading the MNIST dataset. The dataset is downloaded automatically the first time this function is called and is stored in your home directory in ~/.keras/datasets/mnist.pkl.gz as a 15MB file.

This is very handy for developing and testing deep learning models.

# Plot mnist instances

from keras.datasets import mnist

import matplotlib.pyplot as plt

# load (downloaded if needed) the MNIST dataset

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

# plot 4 images as gray scale

plt.subplot(221)

plt.imshow(X\_train[0], cmap=plt.get\_cmap('gray'))

plt.subplot(222)

plt.imshow(X\_train[1], cmap=plt.get\_cmap('gray'))

plt.subplot(223)

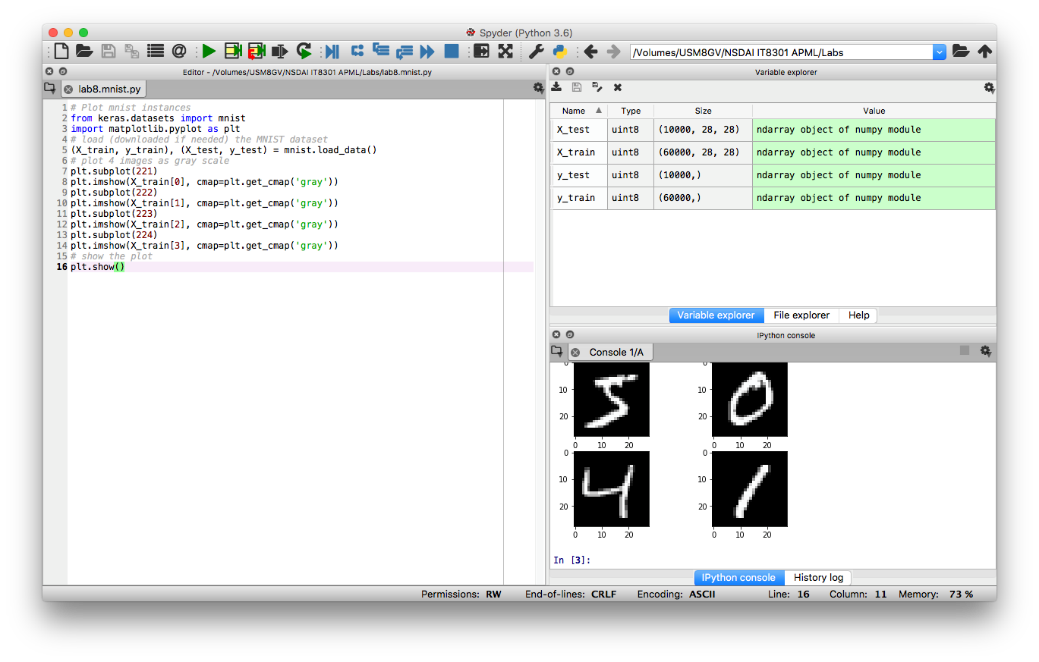
plt.imshow(X\_train[2], cmap=plt.get\_cmap('gray'))

plt.subplot(224)

plt.imshow(X\_train[3], cmap=plt.get\_cmap('gray'))

# show the plot

plt.show()



## Training and evaluation of model

You 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. We will use this as a baseline for comparing more complex convolutional neural network models.

Initialize the random number generator seed to a constant to ensure that the results of your script are reproducible.

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.

We can do this transform easily using the reshape() function on the NumPy array. We can also reduce our memory requirements by forcing the precision of the pixel values to be 32 bit.

The pixel values are gray scale between 0 and 255. It is almost always a good idea to perform some **scaling** of input values when using neural network models. Because the scale is well known and well behaved, we can very quickly **normalize** the pixel values to the range 0 and 1 by dividing each value by the maximum of 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.

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.

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.

# Normal Neural Network

import numpy

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Dropout

from keras.utils import np\_utils

# fix random seed for reproducibility

seed = 88

numpy.random.seed(seed)

# load data

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

# 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

# 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]

# build the 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'])

# 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("Baseline Error: %.2f%%" % (100-scores[1]\*100))

# 3. CNN Model

In this section we will implement a more complex CNN model for the MNIST problem.

## Data Preparation

We always initialize the random number generator to a constant seed value for reproducibility of results.

Next we need to load the MNIST dataset and reshape it so that it is suitable for use training a CNN. In Keras, the layers used for two-dimensional convolutions expect pixel values with the dimensions [pixels][width][height].

In the case of RGB, the first dimension pixels would be 3 for the red, green and blue components and it would be like having 3 image inputs for every color image. *In the case of MNIST where the pixel values are gray scale, the pixel dimension is set to 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.

## CNN training and Evaluation

Convolutional neural networks are more complex than standard multi-layer perceptrons, so we will start by using a simple structure to begin with that uses all of the elements for state of the art results. Below summarizes the 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.

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

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.

Running the example, the accuracy on the training and validation test is printed each epoch and at the end of the classification error rate is printed.

You can see that the network achieves an error rate of 0.92, which is better than our simple multi-layer perceptron model above.

# baseline CNN Model

import numpy

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Dropout

from keras.layers import Flatten

from keras.layers.convolutional import Conv2D

from keras.layers.convolutional import MaxPooling2D

from keras.utils import np\_utils

from keras import backend as K

# fix random seed for reproducibility

seed = 1

numpy.random.seed(seed)

# load data

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

# reshape to be [samples][pixels][width][height]

X\_train = X\_train.reshape(X\_train.shape[0], 28, 28, 1).astype('float32')

X\_test = X\_test.reshape(X\_test.shape[0], 28, 28, 1).astype('float32')

# 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]

# build the model

# create model

model = Sequential()

model.add(Conv2D(32, (5, 5), input\_shape=(28, 28, 1), 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'])

# 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("CNN Error: %.2f%%" % (100-scores[1]\*100))

**More Complex CNN Model**

Now that we have seen how to create a simple CNN, let’s take a look at a model capable of close to state of the art results.

This 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.

Running the example prints accuracy on the training and validation datasets each epoch and a final classification error rate.

The model takes about 100 seconds to run per epoch. This slightly larger model achieves the respectable classification error rate of 0.88%.

# Larger CNN Model

import numpy

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Dropout

from keras.layers import Flatten

from keras.layers.convolutional import Conv2D

from keras.layers.convolutional import MaxPooling2D

from keras.utils import np\_utils

from keras import backend as K

# fix random seed for reproducibility

seed = 88

numpy.random.seed(seed)

# load data

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

# reshape to be [samples][pixels][width][height]

X\_train = X\_train.reshape(X\_train.shape[0], 28, 28, 1).astype('float32')

X\_test = X\_test.reshape(X\_test.shape[0], 28, 28, 1).astype('float32')

# 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]

# build the model

# create model

model = Sequential()

model.add(Conv2D(30, (5, 5), input\_shape=(28, 28, 1), 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'])

# 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("CNN Error: %.2f%%" % (100-scores[1]\*100))