COMP809 – Data Mining and Machine Learning

Lab 11

- **CIFAR10 dataset**

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

"""
 Data preparation
    1 Load data
    2 Check for null and missing values
    3 Normalization
    4 Reshape
    5 Label encoding
    6 Split training and valdiation set

 CNN
    1 Define the model
    2 Set the optimizer and annealer
    3 Data augmentation

 Evaluate the model
    1 Training and validation curves
    2 Confusion matrix

 Prediction
    1 Predict and Submit results
"""

"""
Download and prepare the CIFAR10 dataset
The CIFAR10 dataset contains 60,000 color images in 10 classes,
with 6,000 images in each class. The dataset is divided
into 50,000 training images and 10,000 testing images.
The classes are mutually exclusive and there is no overlap
between them.
"""

(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()
```

```python
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images /
255.0

"""
To verify that the dataset looks correct,
let's plot the first 25 images from the training set and
display the class name below each image:
"""
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i])
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()

"""
The 6 lines of code below define the convolutional base using a
common pattern: a stack of Conv2D and MaxPooling2D layers.

As input, a CNN takes tensors of shape (image_height,
image_width, color_channels),
ignoring the batch size. If you are new to these dimensions,
color_channels refers to (R,G,B).
In this example, you will configure your CNN to process inputs
of shape (32, 32, 3),
which is the format of CIFAR images. You can do this by passing
the argument input_shape to your first layer.
"""
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

```
"""
Let's display the architecture of your model so far:
(describe and interpret your model in your report )
"""
model.summary()

"""
Add Dense layers on top
To complete the model, you will feed the last output tensor
from the convolutional base (of shape (4, 4, 64))
into one or more Dense layers to perform classification.
Dense layers take vectors as input (which are 1D),
while the current output is a 3D tensor. First,
you will flatten (or unroll) the 3D output to 1D,
then add one or more Dense layers on top.
!!!! CIFAR has 10 output classes !!!! so you use a final Dense
layer with 10 outputs.
"""
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))

"""
Compile and train the model
"""
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=
True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))

plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

# evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels,
verbose=2)
print(test_acc)
```

- ## MINIST dataset
  - ### Visualization

```
from tensorflow.keras.datasets import mnist
from matplotlib import pyplot

"""
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.

The example below loads the MNIST dataset using the Keras
API and creates
a plot of the first nine images in the training dataset.


http://yann.lecun.com/exdb/mnist/

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

"""

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

```python
# plot first few images
for i in range(9):
    # define subplot
    pyplot.subplot(330 + 1 + i)
    # plot raw pixel data
    pyplot.imshow(trainX[i], cmap=pyplot.get_cmap('gray'))
# show the figure
pyplot.show()


"""
Although the MNIST dataset is effectively solved,
it can be a useful starting point for developing and 
practicing
a methodology for solving image classification tasks using 
convolutional neural networks.

Instead of reviewing the literature on well-performing 
models on the dataset,
 we can develop a new model from scratch.

The dataset already has a well-defined train and test 
dataset that we can use.
"""
```

-MINIST dataset

- ### Evaluate models and find the best one

```
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.layers import BatchNormalization

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


# load train and test dataset
def load_dataset():
    """
    We know some things about the dataset.

    For example, we know that the images are all pre-aligned
    (e.g. each image only contains a hand-drawn digit),
    that the images all have the same square size of 28×28
pixels,
    and that the images are grayscale.

    Therefore, we can load the images and reshape the data
arrays to have a single color channel.

    # load dataset
    (trainX, trainY), (testX, testY) = mnist.load_data()
    # 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))


    We also know that there are 10 classes and that classes are
represented as unique integers.
```

```
    We can, therefore, use a one hot encoding for the class
element of each sample,
    transforming the integer into a 10 element binary vector
with a 1 for the index of the class value,
    and 0 values for all other classes.
    We can achieve this with the to_categorical() utility
function.
    :return:
    """
    # load dataset
    (trainX, trainY), (testX, testY) = mnist.load_data()
    # 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))
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY


# scale pixels
def prep_pixels(train, test):
    """
    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.

    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.

    :param train:
    :param test:
    :return:
    """
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
```

```
    return train_norm, test_norm


# define cnn model
def define_model():
    """

    Next, we need to define a baseline convolutional neural
network model for the problem.

    The model has two main aspects: the feature extraction front
end comprised
    of convolutional and pooling layers, and the classifier
backend that will make a prediction.

    For the convolutional front-end, we can start with a single
convolutional
    layer with a small filter size (3,3) and a modest number of
filters (32) followed by a max pooling layer. The filter maps
can then be flattened to provide features to the classifier.

    Given that the problem is a multi-class classification task,
    we know that we will require an output layer with 10 nodes
    in order to predict the probability distribution of an image
belonging to each of the 10 classes.
    This will also require the use of a softmax activation
function.
    Between the feature extractor and the output layer, we can
add a dense layer to interpret the features, in this case with
100 nodes.

    All layers will use the ReLU activation function and the He
weight
    initialization scheme, both best practices.

    We will use a conservative configuration
    for the stochastic gradient descent optimizer with a
learning rate of 0.01
    and a momentum of 0.9. The categorical cross-entropy loss
function will be optimized,
    suitable for multi-class classification, and we will monitor
the classification accuracy metric,
    which is appropriate given we have the same number of
examples in each of the 10 classes.

    The define_model() function below will define and return
this model.
```

```
    :return:
    """
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu',
kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(learning_rate=0.01, momentum=0.9)
    model.compile(optimizer=opt,
loss='categorical_crossentropy', metrics=['accuracy'])
    return model


# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
    scores, histories = list(), list()
    # prepare cross validation
    """
    The performance of a model can be taken as the mean
performance across k-folds,
    given the standard deviation, that could be used to estimate
a confidence interval if desired.

    We can use the KFold class from the scikit-learn API to
    implement the k-fold cross-validation evaluation of a given
neural network model.

    After the model is defined, we need to evaluate it.

    The model will be evaluated using five-fold cross-
validation.
    The value of k=5 was chosen to provide a baseline for both
    repeated evaluation and to not be so large as to require a
    long running time. Each test set will be 20% of the training
dataset,
    or about 12,000 examples, close to the size of the actual
test set for this problem.

    The training dataset is shuffled prior to being split,
    and the sample shuffling is performed each time,
    so that any model we evaluate will have the same train
    and test datasets in each fold, providing an apples-to-
apples comparison between models.
```

```
    We will train the baseline model for a modest 10 training
    epochs with a default batch size of 32 examples. The test
set
    for each fold will be used to evaluate the model
     both during each epoch of the training run, so that we can
later create learning curves,
     and at the end of the run, so that we can estimate the
performance of the model.
     As such, we will keep track of the resulting history from
each run, as well as the classification accuracy of the fold.

    The evaluate_model() function below implements these
behaviors,
    taking the training dataset as arguments and returning
     a list of accuracy scores and training histories that can
be later summarized.

    """
    kfold = KFold(n_folds, shuffle=True, random_state=1)
    # enumerate splits
    for train_ix, test_ix in kfold.split(dataX):
        # define model
        model = define_model()
        # select rows for train and test
        trainX, trainY, testX, testY = dataX[train_ix],
dataY[train_ix], dataX[test_ix], dataY[test_ix]
        # fit model
        history = model.fit(trainX,
                            trainY,
                            # please try the epoch at least =
10,
                            # we don't want to waste too much
time on the lab, so use 3
                            epochs=3,
                            batch_size=32,
                            validation_data=(testX, testY),
                            verbose=0)
        # evaluate model
        _, acc = model.evaluate(testX, testY, verbose=0)
        print('> %.3f' % (acc * 100.0))
        # stores scores
        scores.append(acc)
        histories.append(history)
    return scores, histories
```

```
# plot diagnostic learning curves
def summarize_diagnostics(histories):
    """
    Once the model has been evaluated, we can present the
results.

    There are two key aspects to present:
     the diagnostics of the learning behavior of the model
during
     training and the estimation of the model performance.
     These can be implemented using separate functions.

    First, the diagnostics involve creating a line plot showing
model
    performance on the train and test set during each fold of
the k-fold cross-validation.
     These plots are valuable for getting an idea of whether a
model is overfitting,
     underfitting, or has a good fit for the dataset.

    We will create a single figure with two subplots, one for
loss and one for accuracy.
    Blue lines will indicate model performance on the training
dataset and orange
    lines will indicate performance on the hold out test
dataset.
    The summarize_diagnostics() function below creates and shows
this plot given
    the collected training histories.
    :param histories:
    :return:
    """
    for i in range(len(histories)):
        # plot loss
        pyplot.subplot(2, 1, 1)
        pyplot.title('Cross Entropy Loss')
        pyplot.plot(histories[i].history['loss'], color='blue',
label='train')
        pyplot.plot(histories[i].history['val_loss'],
color='orange', label='test')
        # plot accuracy
        pyplot.subplot(2, 1, 2)
        pyplot.title('Classification Accuracy')
        pyplot.plot(histories[i].history['accuracy'],
color='blue', label='train')
        pyplot.plot(histories[i].history['val_accuracy'],
color='orange', label='test')
```

```
    pyplot.show()


# summarize model performance
def summarize_performance(scores):
    # print summary
    print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores) *
100, std(scores) * 100, len(scores)))
    # box and whisker plots of results
    pyplot.boxplot(scores)
    pyplot.show()


# run the test function for evaluating a model
def run_test():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # evaluate model

    # please try the n_fold = 5,
    # we don't want to waste too much time on the lab, so use 3

    scores, histories = evaluate_model(trainX, trainY, 3)
    # learning curves
    summarize_diagnostics(histories)
    # summarize estimated performance
    summarize_performance(scores)


# entry point, run the test harness
run_test()

"""
Please try the other models as follows,

# another model with BatchNormalization
def define_model():
   model = Sequential()
   model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
   model.add(BatchNormalization())
   model.add(MaxPooling2D((2, 2)))
   model.add(Flatten())
   model.add(Dense(100, activation='relu',
```

```
kernel_initializer='he_uniform'))
    model.add(BatchNormalization())
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
    return model


# another model with more layers (Increase in Model Depth)
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_uniform'))
    model.add(Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_uniform'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu',
kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
    return model
"""
```

-MINIST dataset

-Save the best model

```
# save the final model to file
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.optimizers import SGD


# load train and test dataset
def load_dataset():
    # load dataset
    (trainX, trainY), (testX, testY) = mnist.load_data()
    # 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))
    # one hot encode target values
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY


# scale pixels
def prep_pixels(train, test):
    # convert from integers to floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # normalize to range 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # return normalized images
    return train_norm, test_norm


# define cnn model
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_uniform'))
```

```
    model.add(Conv2D(64, (3, 3), activation='relu',
kernel_initializer='he_uniform'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu',
kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = SGD(learning_rate=0.01, momentum=0.9)
    model.compile(optimizer=opt,
loss='categorical_crossentropy', metrics=['accuracy'])
    return model


# run the test harness for evaluating a model
def run_test_harness():
    # load dataset
    trainX, trainY, testX, testY = load_dataset()
    # prepare pixel data
    trainX, testX = prep_pixels(trainX, testX)
    # define model
    model = define_model()
    # fit model
    model.fit(trainX, trainY, epochs=10, batch_size=32,
verbose=0)
    # save model
    model.save('final_model.h5')


# entry point, run the test harness
run_test_harness()
```

-MINIST dataset

-Make prediction

Please take a screenshot of your handwrite digit. Load the image and make prediction

```
# make a prediction for a new image.
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.models import load_model
import numpy as np


# load and prepare the image
def load_image(filename):
    # load the image
    img = load_img(filename, grayscale=True, target_size=(28,
28))
    # convert to array
    img = img_to_array(img)
    # reshape into a single sample with 1 channel
    img = img.reshape(1, 28, 28, 1)
    # prepare pixel data
    img = img.astype('float32')
    img = img / 255.0
    return img


# load an image and predict the class
def run_example():
    # load the image
    img = load_image('sample_image.png')
    # load model
    model = load_model('final_model.h5')
    # predict the class
    predict_x = model.predict(img)
    classes_x = np.argmax(predict_x, axis=1)

    # we only input one image
    # so we get the first element of the array
    digit = classes_x[0]

    print("The prediction of the input handwritten digit is :
", digit, ". \U0001F606")


# entry point, run the example
run_example()
```