MNIST handwritten digits classification with CNNs

In this notebook, we'll train a convolutional neural network (CNN, ConvNet) to classify MNIST digits using **Tensorflow** (version $\ge$ 2.0 required) with the **Keras API**.

This notebook builds on the MNIST-MLP notebook, so the recommended order is to go through the MNIST-MLP notebook before starting with this one.

First, the needed imports.

In [ ]:
```
%matplotlib inline

import os
if not os.path.isfile('pml_utils.py'):
  !wget https://raw.githubusercontent.com/csc-training/intro-to-dl/master/day1/pml_
from pml_utils import show_failures
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import plot_model, to_categorical

from packaging.version import Version as LV

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

print('Using Tensorflow version: {}, and Keras version: {}.'.format(tf.__version__
assert(LV(tf.__version__) >= LV("2.0.0"))
```

In [ ]:
```
gpus = tf.config.list_physical_devices('GPU')
if len(gpus) > 0:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
    except RuntimeError:
        pass
    from tensorflow.python.client import device_lib
    for d in device_lib.list_local_devices():
        if d.device_type == 'GPU':
            print('GPU', d.physical_device_desc)
else:
    print('No GPU, using CPU instead.')
```

MNIST data set

In [ ]:
```
from tensorflow.keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
nb_classes = 10
```

```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

# one-hot encoding:
Y_train = to_categorical(y_train, nb_classes)
Y_test = to_categorical(y_test, nb_classes)

print()
print('MNIST data loaded: train:',len(X_train),'test:',len(X_test))
print('X_train:', X_train.shape)
print('y_train:', y_train.shape)
print('Y_train:', Y_train.shape)
```

We'll have to do a bit of tensor manipulations...

In [ ]:
```
# input image dimensions
img_rows, img_cols = 28, 28

X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

print('X_train:', X_train.shape)
```

Initialization

Now we are ready to create a convolutional model.

- The `Conv2D` layer operate on 2D matrices so we input the digit images directly to the model.
- The `MaxPooling2D` layer reduces the spatial dimensions, that is, makes the image smaller.
- The `Flatten` layer flattens the 2D matrices into vectors, so we can then switch to `Dense` layers as in the MLP model.

See https://keras.io/layers/convolutional/, https://keras.io/layers/pooling/ for more information.

In [ ]:
```
inputs = keras.Input(shape=input_shape)

x = layers.Conv2D(32, (3, 3),
                  padding='valid',
                  activation ='relu')(inputs)
x = layers.MaxPooling2D(pool_size=(2,2))(x)

x = layers.Flatten()(x)
x = layers.Dense(units=128, activation ='relu')(x)

outputs = layers.Dense(units=nb_classes,
                       activation='softmax')(x)

model = keras.Model(inputs=inputs, outputs=outputs,
                    name="cnn_model")
```

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

In [ ]:
```
plot_model(model, show_shapes=True)
```

Learning

Now let's train the CNN model.

This is a relatively complex model, so training is considerably slower than with MLPs.

In [ ]:
```
%%time

epochs = 5 # one epoch can take tens of seconds without a GPU

history = model.fit(X_train,
                    Y_train,
                    epochs=epochs,
                    batch_size=128,
                    verbose=2)
```

In [ ]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,3))

ax1.plot(history.epoch,history.history['loss'])
ax1.set_title('loss')
ax1.set_xlabel('epoch')

ax2.plot(history.epoch,history.history['accuracy'])
ax2.set_title('accuracy')
ax2.set_xlabel('epoch');
```

Inference

With enough training epochs, the test accuracy should exceed 99%.

You can compare your result with the state-of-the art here. Even more results can be found here.

In [ ]:
```
%%time
scores = model.evaluate(X_test, Y_test, verbose=2)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

We can now take a closer look at the results using the `show_failures()` helper function.

Here are the first 10 test digits the CNN classified to a wrong class:

In [ ]:
```
predictions = model.predict(X_test)

show_failures(predictions, y_test, X_test)
```

We can use `show_failures()` to inspect failures in more detail. For example, here are failures in which the true class was "6":

In [ ]:
```
show_failures(predictions, y_test, X_test, trueclass=6)
```

Task 1: A more complex CNN model

Your task is to try the same problem as above, but with two convolutional layers. The new model should have the following layers in order:

- Input layer
- Convolutional (`Conv2D`) layer with 32 units and 3x3 kernels, ReLU activation, valid padding
- Another identical convolutional layer
- Max pooling (`MaxPooling2D` layer with 2x2 pooling size
- Dropout with 0.25 rate
- Flatten
- Dense layer with 128 units
- Dropout with 0.5 rate
- Dense output layer (same as in the example above)

You can consult the Keras documentation at https://keras.io/.

The code below is missing the model definition. You can copy any suitable layers from the example above.

In [ ]:
```
# Define the input layer
ex1_inputs = keras.Input(shape=input_shape)

# First convolutional layer
x = layers.Conv2D(32, (3, 3), padding='valid', activation='relu')(ex1_inputs)

# Second convolutional layer
x = layers.Conv2D(32, (3, 3), padding='valid', activation='relu')(x)

# Max pooling layer to reduce spatial dimensions
x = layers.MaxPooling2D(pool_size=(2, 2))(x)

# Dropout layer to prevent overfitting
x = layers.Dropout(0.25)(x)

# Flatten the feature maps into a vector
x = layers.Flatten()(x)

# Dense layer with 128 units and ReLU activation
x = layers.Dense(units=128, activation='relu')(x)

# Dropout layer to prevent overfitting
x = layers.Dropout(0.5)(x)

# Output layer with softmax activation for classification
ex1_outputs = layers.Dense(units=nb_classes, activation='softmax')(x)

# Create the model with input and output layers
ex1_model = keras.Model(inputs=ex1_inputs, outputs=ex1_outputs, name="better_cnn_mo

# Compile the model with loss function, optimizer, and metrics
```

```
ex1_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accu

# Print the summary of the model architecture
print(ex1_model.summary())

# Train the model on the training data
epochs = 5
ex1_history = ex1_model.fit(X_train, Y_train, epochs=epochs, batch_size=128, verbos

# Plot the training progress (loss and accuracy)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 3))
ax1.plot(ex1_history.epoch, ex1_history.history['loss'])
ax1.set_title('loss')
ax1.set_xlabel('epoch')

ax2.plot(ex1_history.epoch, ex1_history.history['accuracy'])
ax2.set_title('accuracy')
ax2.set_xlabel('epoch')

# Evaluate the model on the test data
ex1_scores = ex1_model.evaluate(X_test, Y_test, verbose=2)
print("%s: %.2f%%" % (ex1_model.metrics_names[1], ex1_scores[1] * 100))
```

Task 2: Tune training parameters

Try to improve the classification accuracy, in particular by trying different optimizers and playing with the parameters of the training process.

See optimizers available in Keras here: https://keras.io/api/optimizers/#available-optimizers

The parameters of the `fit()` method are discussed here: https://keras.io/api/models/model_training_apis/#fit-method

You can take the model created in Task 1 as a starting point. Below is a code example which you can modify.

In [25]:
```
# Clone the model from Task 1
ex2_model = keras.models.clone_model(ex1_model)

# Compile the model with a different optimizer and learning rate
ex2_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accu

# Train the model for more epochs
epochs = 10
ex2_history = ex2_model.fit(X_train, Y_train, epochs=epochs, batch_size=128, verbos

# Evaluate the model on the test data
ex2_scores = ex2_model.evaluate(X_test, Y_test, verbose=2)
print("%s: %.2f%%" % (ex2_model.metrics_names[1], ex2_scores[1] * 100))
```

```
Epoch 1/10
469/469 - 21s - loss: 0.2792 - accuracy: 0.9161 - 21s/epoch - 44ms/step
Epoch 2/10
469/469 - 19s - loss: 0.1009 - accuracy: 0.9701 - 19s/epoch - 42ms/step
Epoch 3/10
469/469 - 20s - loss: 0.0720 - accuracy: 0.9779 - 20s/epoch - 42ms/step
Epoch 4/10
469/469 - 20s - loss: 0.0610 - accuracy: 0.9814 - 20s/epoch - 42ms/step
Epoch 5/10
469/469 - 20s - loss: 0.0522 - accuracy: 0.9841 - 20s/epoch - 42ms/step
Epoch 6/10
469/469 - 20s - loss: 0.0465 - accuracy: 0.9857 - 20s/epoch - 42ms/step
Epoch 7/10
469/469 - 20s - loss: 0.0401 - accuracy: 0.9871 - 20s/epoch - 42ms/step
Epoch 8/10
469/469 - 20s - loss: 0.0385 - accuracy: 0.9878 - 20s/epoch - 42ms/step
Epoch 9/10
469/469 - 19s - loss: 0.0350 - accuracy: 0.9891 - 19s/epoch - 41ms/step
Epoch 10/10
469/469 - 20s - loss: 0.0320 - accuracy: 0.9898 - 20s/epoch - 43ms/step
313/313 - 2s - loss: 0.0292 - accuracy: 0.9906 - 2s/epoch - 7ms/step
accuracy: 99.06%
```

Run this notebook in Google Colaboratory using this link.