Practical 3 : Convolutional Neural Networks (CNNs)

In this practical, we will build a convolutional neural network to classify handwritten digits. More specifically, we will use the MNIST dataset to implement and evaluate a neural network-based model for classification of handwritten digits. The main aim is for you to learn, enjoy, and hopefully benefit from it.

You could implement and test your solution in any software platform/release you wish. PyTorch, TensorFlow 2 and Keras, and jax and flax, are installed on the departmental machines. The practical instructions include some details about each if these, for information. It would, however, be difficult to cover them exhaustively during the practicals. It is important and useful to know these different options exist, so that you could explore them more in the future, if you wish. For information, the `jax` and `flax` visions are described at jax and flax, respectively.
Potentially useful examples are: training a convolutional classifier in PyTorch, A CNN in tensorflow/keras, a jax neural network example, a flax annotated MNIST example.

Signing off:

- A correct and complete implementation will be marked with an S+. Intermediate, but not fully working solutions that are conceptually correct, will be discussed with the students, and will be marked with an S.

Convolution and Maxpool

Read about convolution and how 2-D convolution is implemented in (one or more of):

- PyTorch;
- TensorFlow;
- Keras;
- jax and flax.

and max-pooling, at:

- in PyTorch;
- in TensorFlow;
- in Keras;
- in flax.

It would help you to understand how the size of the next layer is computed; depending on the software used, you may need to specify these as parameters, rather than them being inferred based on the model specification. It will help to understand how the sizes are affected by strides, padding, etc., in order to implement the network properly.

Loading the MNIST data

In order to obtain the MNIST data you could run the following:

In `PyTorch`:

```
import torch
from torchvision import datasets, transforms

transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
    ])

dataset = datasets.MNIST('./mnist', train=True, download=True, transform=transform)
dataset_train, dataset_valid = torch.utils.data.random_split(dataset, [50000, 10000])
dataset_test = datasets.MNIST('./mnist', train=False, transform=transform)
```

This would automatically download the MNIST data to the folder `./mnist`, create training and validation sets (`dataset_train` and `dataset_valid`) from the MNIST training data by randomly splitting it into 50k and 10k portions, and also load the MNIST test data (`dataset_test`). It also normalizes the data using the pre-computed mean and standard deviation values for the training set in the [0, 1] range.

In `jax` and `flax`: please look at the code in a jax neural network example and a flax annotated mnist example.

Visualising the MNIST data

To visualise the first 16 data points from the MNIST training data, you could use the code below, where `train_images` is defined as above; the `imshow` function is in `matplotlib.pyplot`.

```
fig = plt.figure()
for i in range(16):
    ax = fig.add_subplot(4, 4, i + 1)
    ax.set_xticks(())
    ax.set_yticks(())
    ax.imshow(dataset_train[i][0].reshape(28, 28), cmap='Greys_r')
```

This code will result in an output like in Figure 1.

Convolutional Neural Network

Below is a suggested CNN configuration, which achieves about 98% accuracy in the training, validation and testing. You could use this suggested configuration, ar amend it, as you wish.

- The inputs to the first layer should be 1x28x28 images (where the dimensions represent the number of channels, image height, image width, respectively. The number of channels is one because MNIST images are grayscale), and this is the expected input of the model. Note that in practice there will be an extra dimension that represents the different images in a given minibatch, so that the input data shape will be Bx1x28x28, where B is the minibatch size.

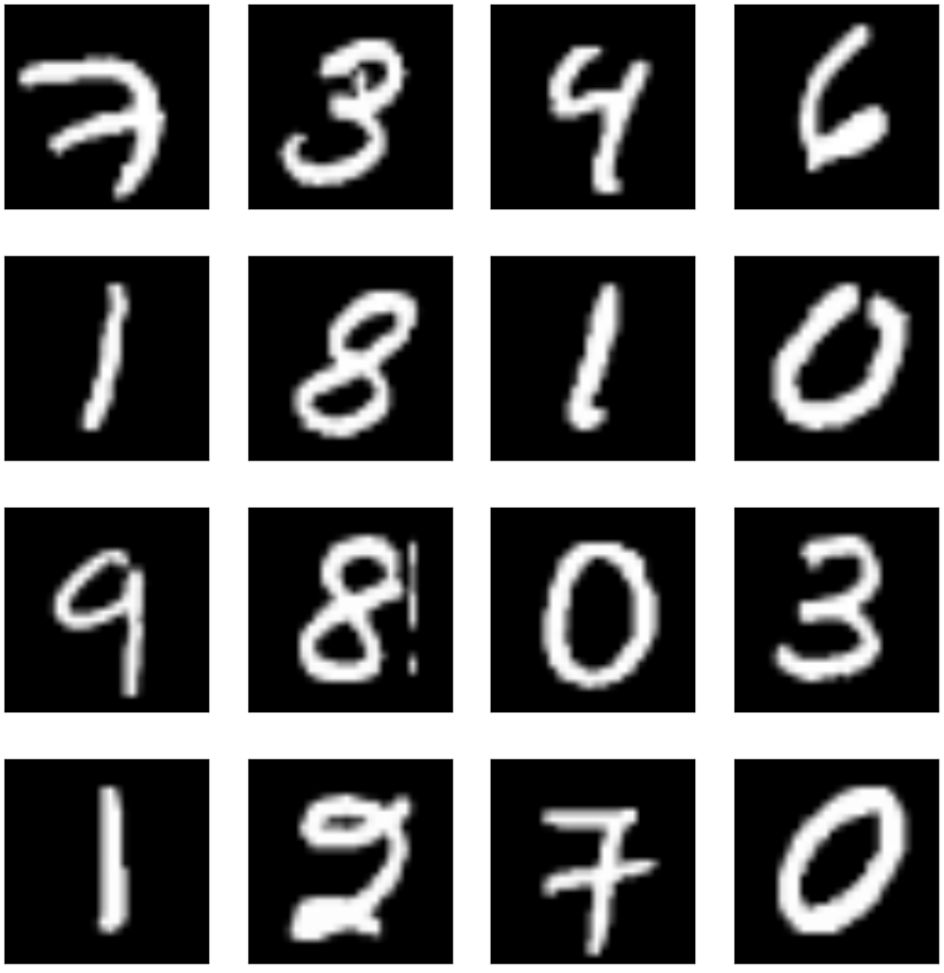

Figure 1: Digits

- Where applicable, we'll initialise weights as Gaussian random variables with mean 0 and variance 0.0025. For biases we'll initialise everything with a constant 0.1. This is because we're mainly going to be using ReLU non-linearities. In many systems including PyTorch, you can also rely on the default initialisation of the convolutional layers.
- Add a convolutional layer with 25 filters of size 12x12x1 (followed by the ReLU non-linearity). Use a stride of 2 in both directions, and ensure that there is no padding.
- Add a second convolutional layer with 64 filters of size 5x5x25 that maintains the same width and height. Use stride of 1 in both directions and add padding as necessary, and add ReLU non-linearity after the convolution.
- Add a max pooling layer with pool size 2x2.
- Add a flattening operation to transform the 2d image data at the output of the convolutions into 1d vectors that will be fed to fully-connected layers. See, for example: PyTorch flatten.
- Add a fully connected layer with 1024 units. Each unit in the output of max pool should be connected to these 1024 units. Add the ReLU non-linearity following the layer.
- Add a dropout layer to reduce overfitting, with a 0.2 rate.
- Add another fully connected layer to get 10 output units, which will be used with a softmax function representing the probabilities of each class. Note that depending on the framework and loss function implementation you use (cross-entropy or negative-log-likelihood), you may or may not need to add an actual softmax operation to the model's output. (E.g., PyTorch's cross-entropy loss function expects to receive the raw output of the model's last layer (without a softmax) and it internally applies a log-softmax and negative-log-likelihood loss to compute the loss.)

In `keras`, you could use:

```
model.summary()
```

to check the outputs of your CNN model.

Loss Function, Accuracy and Training Algorithm

- We'll use the cross entropy loss function.
- Accuracy is simply defined as the fraction of data correctly classified.
- For training you could use the Adam optimizer (read the documentation) and set the learning rate to be 1e-4. You are welcome, and in fact encouraged, to experiment with other optimisation procedures and learning rates.

Training

Train and evaluate your model on the MNIST dataset. It would help to use minibatches (e.g. of size 64). Try about 1000 - 5000 iterations. You may want to start out with fewer iterations to make sure your code is making good progress. Once you are sure your code is correct, you can let it run for more iterations - it will take a bit of time for the model to finish training. Keep track of the training and validation accuracy every 100 iterations or so; however, do not touch the test dataset. Once you are sure your optimisation is working properly, you should run the resulting model on the test data and report the test error.

Handin

- Report your code defining the model
- Report the accuracy on the test data (you should get about 98%-ish accuracy)

Possible extensions (for fun)

- Run experiments with different optimisers, learning and dropout rates, and model structures, and analyse the impact of different parameter values on the overall number of parameters to be learned, on the computational cost, and on the model's accuracy.
- Plot the accuracy and loss during training and validation.
- Use a different dataset.