UNIVERSITY OF OXFORD

Machine Learning
Michaelmas Term 2020
Week 8

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. 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: A CNN in tensorflow/keras, a jax neural network example, and 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.
- The week 9 session will be held 11am-1pm, on Thursday 10th December.
- You are welcome to ask questions via the MS Teams chat during weeks 8 and 9, and to sign off during any of the three ML practical sessions in week 8 and the additional signing off session in week 9.

Convolution and Maxpool

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

- tensorflow;
- keras;
- jax and flax.

and Max-pooling, at:

- 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. Also understand what the difference between `VALID` and `SAME`

padding is. It'll 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 `tensorflow`:

```
import tensorflow as tf
from tensorflow.keras import datasets
import tensorflow_datasets as tfds

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

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

To normalise pixel values, you could use:

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

You also need to encode the labels using one-hot encoding (e.g. for tensorflow).

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(train_images[i].reshape(28, 28), cmap='Greys_r')
```

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

If you wish, you could read more about model building using `tensorflow 2.0` and `keras`, and about result plotting, at:

- https://www.tensorflow.org/api_docs/python/tf/keras/Model, and
- https://www.tensorflow.org/api_docs/python/tf/keras/utils/plot_model.

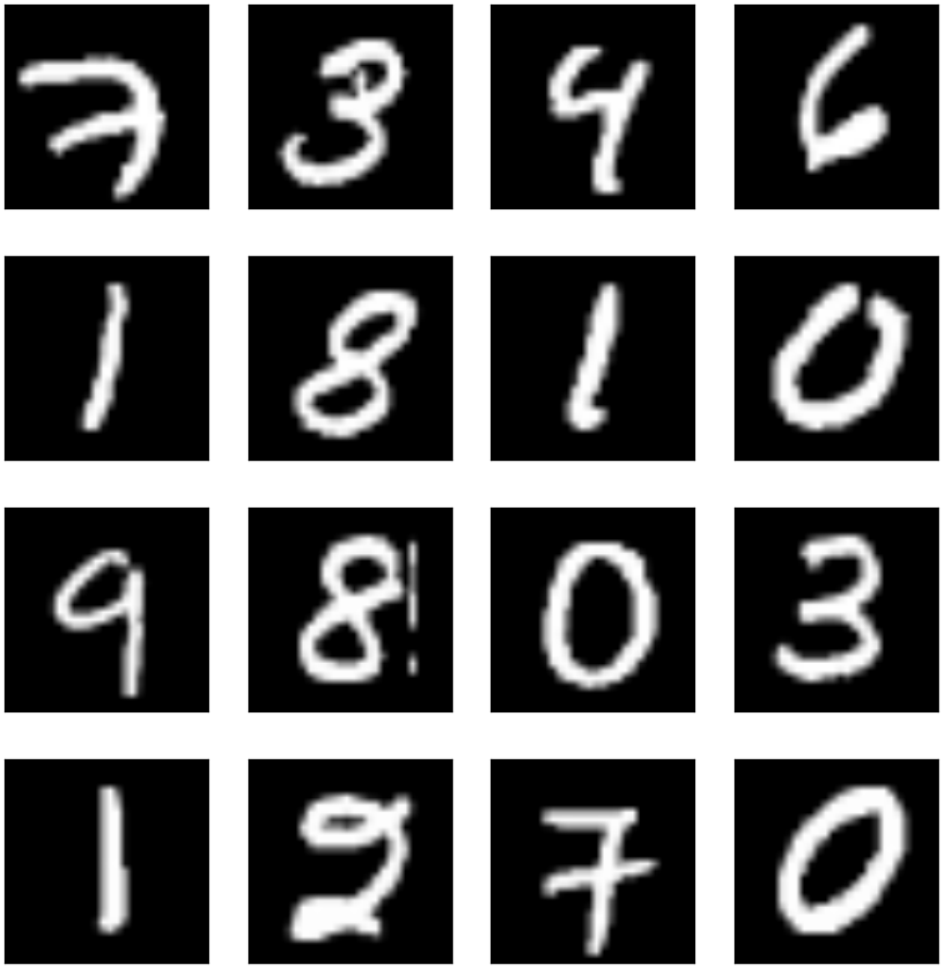

Figure 1: Digits

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 28x28x1 images (1 because these are grey scale), and this is the expected input of the model.
- 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.
- Add a convolutional layer with 25 filters of size 12x12x1 (and 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 use the ReLU non-linearity.
- Add a max_pooling layer with pool size 2x2.
- Add a Flatten-type layer: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Flatten.
- Add a fully connected layer with 1024 units. Each unit in the max_pool should be connected to these 1024 units. Add the ReLU non-linearity to these units.
- Add a Dropout layer to reduce overfitting, with a 0.2 rate.
- Add another fully connected layer to get 10 output units, and a softmax activation function.

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 AdamOptimizer (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 50). 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.