Deep Learning with Keras Tutorial

Deep Learning Winter School, University of Hull
January 22-23, 2019
Dr Nina Dethlefs

This tutorial goes through the basic steps involved in training neural networks with Keras [1]. We will train a multi-layer perceptron (MLP) and a convolutional neural network (CNN) for the MNIST digit recognition task.

1. The MNIST digit recognition task

The MNIST task[1] is a digit recognition task in which a neural net learns to identify handwritten digits from images. The training set contains 60,000 examples and the test set contains 10,000 examples. An illustration of some images is shown in Figure 1.[2]

Figure 1: Illustration of MNIST examples images.

The images are represented as matrices that represent 28×28 pixels or a flat vector of length 784. Figure 2 shows an example representation for digit 1.[3]

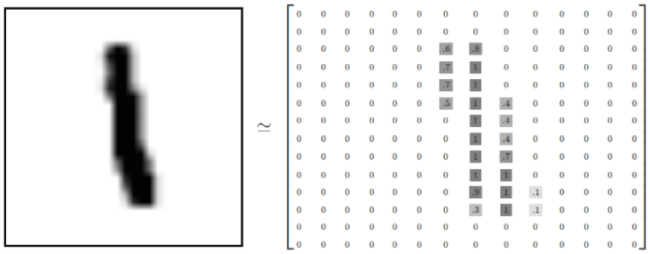

Figure 2: Example of digit representation for input to neural net.

MNIST is one of the most famous benchmarks for deep learning. A simple MLP (as the one we are going to train first) can achieve about 98.4% of classification accuracy after 20 training epochs. The best reported accuracy is 99.9% by [2] using a 2-layered convolutional neural net with DropConnect — a technique comparable to drop-out in neural nets.

[1] `http://yann.lecun.com/exdb/mnist/`
[2] From `https://en.wikipedia.org/wiki/MNIST_database`
[3] From `https://www.tensorflow.org/get_started/mnist/beginners`

Training an MLP for MNIST in Keras

Keras[4] is a Python library that provides a convenient set of functions and implementations to develop deep learning models at a higher level than using other frameworks such as TensorFlow. It also contains a number of very good example implementations of different models. For the first part of this tutorial, we will use *mnist_mlp.py*, a simple MLP implementation for MNIST.

Lets look at the code in detail. The first few lines just import a number of modules: `Sequential` is the model we are going to use. `Dense` and `Dropout` are types of layers. The former, `Dense`, is a normal fully-connected neural net layer and the latter, `Dropout` is a drop-out layer. Drop-out is a regularisation technique that drops a pre-specified number of connections randomly to prevent overfitting [3]. For example, a dropout rate of 0.2 would tell the neural net to lose 20% of connections of the relevant layer. `RMSProp` is a type of optimiser, i.e. an algorithm for updating the weights.

As a next step, we specify the batch size (the number of examples we look at "at the same time" during training), the number of output classes (10 in our case for 10 digits) and the number of training epochs.

```
16  batch_size = 128
17  num_classes = 10
18  epochs = 20
```

Once this is done, we need to prepare our training and test data. Keras comes with an MNIST dataset, so all we need to do is load it (line 21). The data will come in the form of a numpy matrix. You can print it to the console to verify this. Keras expects a matrix m×n, where m is the number of examples and n is the dimension of the input data. We therefore need to flatten `x_train` and `x_test` to a shape of (60000, 784) and (10000, 784), respectively. This is done in lines 23 and 24. Lines 25 and 26 make sure that we pass in floats with 32-digit precision. It would work with other types of floats (change the number to confirm this) but 32 is all we need. Lines 27-28 do data normalisation by ensuring that all data points are in the range of $\{0 \ldots 1\}$.

```
20  # the data, shuffled and split between train and test sets
21  (x_train, y_train), (x_test, y_test) = mnist.load_data()
22
23  x_train = x_train.reshape(60000, 784)
24  x_test = x_test.reshape(10000, 784)
25  x_train = x_train.astype('float32')
26  x_test = x_test.astype('float32')
27  x_train /= 255
28  x_test /= 255
29  print(x_train.shape[0], 'train samples')
30  print(x_test.shape[0], 'test samples')
31
32  # convert class vectors to binary class matrices
33  y_train = keras.utils.to_categorical(y_train, num_classes)
34  y_test = keras.utils.to_categorical(y_test, num_classes)
```

[4] `https://keras.io/`

Lines 33-34 finally convert the test data into a matrix containing entries for each of our 10 output classes. The matrix will contain a 1 for the target digit and 0 for all others. Again, print the matrix to the console to see what is going on.

Once our data is loaded and pre-processed, we can define our neural net. You can consult the Keras documentation[5] to see what input values are possible. In our case, we specify that we want 1024 hidden nodes, a ReLU activation function and we have an input dimension of 784 – all for the first layer in line 37. We then define a dropout layer with a dropout rate of 0.99 on line 38. Line 39 specified a further "normal" layer with 10 hidden units and a ReLU activation, followed by another dropout layer. The output layer has 10 outputs (i.e. the number of classes we aim to predict) and a sigmoid activation function.

Note: these values are not at all tuned towards good performance. It will be your task later on to analyse why these will do badly and improve / correct them to get closer towards the network's possible upper-bound performance of 98%.

```
36  model = Sequential()
37  model.add(Dense(1024, activation='relu', input_shape=(784,)))
38  model.add(Dropout(0.99))
39  model.add(Dense(10, activation='relu'))
40  model.add(Dropout(0.85))
41  model.add(Dense(num_classes, activation='sigmoid'))
```

Finally, we add a loss function to our model (categorical crossentropy in this case to deal with our multiple outputs), an optimiser and an evaluation metric (lines 45-47). Lines 49-53 define the parameters for training the model and line 54 keeps track of the evaluation score, i.e. accuracy as specified.

```
45  model.compile(loss='categorical_crossentropy',
46                optimizer=RMSprop(),
47                metrics=['accuracy'])
48
49  history = model.fit(x_train, y_train,
50                      batch_size=batch_size,
51                      epochs=epochs,
52                      verbose=1,
53                      validation_data=(x_test, y_test))
54  score = model.evaluate(x_test, y_test, verbose=0)
55  print('Test loss:', score[0])
56  print('Test accuracy:', score[1])
```

Exercise 1: Modify the training parameters in the MLP. Using the documentation on `https://keras.io/`, modify some parameters of the neural net and observe the effects on performance. For example,

[5] e.g. for Sequential: `https://keras.io/models/sequential/`

- Experiment with the number of *hidden nodes*, try 20, 200, 500 or 2000. What is the effect on performance and speed?
- Experiment with different *activation functions*, try to change the output layer to a ReLU or softmax function, or use sigmoid on all layers. Does the performance change?
- What happens if we change the dropout rate, e.g. to 0 using no dropout, or to more moderate values such as 0.1 or 0.2?
- Add an additional layer to the MLP definition and use a different number of hidden units on that layer. How does this affect speed and performance?
- Use the Keras documentation to add a learning rate to the optimiser. How does the size of the learning rate change the model's performance and speed?

Training a CNN for MNIST in Keras

Now that we are familiar with the basics of training an MLP, we can look at more complex models such as Convolutional Neural Networks (CNN) that are frequently used for computer vision tasks. Keras includes an example for this too.We are going to use another Keras example. Figure 3 shows an illustration of the model we are going to train.

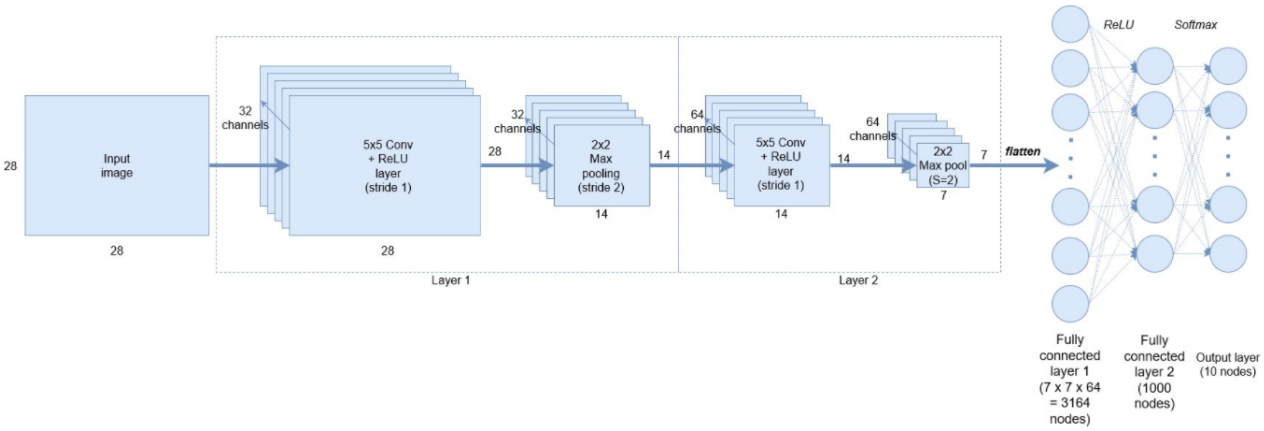

Figure 3: Convolutional neural network with convolutional, pooling and fully-connected layers. The "channels" shown are also known as "feature maps".

```
8  from __future__ import print_function
9  import keras
10 from keras.datasets import mnist
11 from keras.models import Sequential
12 from keras.layers import Dense, Dropout, Flatten
13 from keras.layers import Conv2D, MaxPooling2D
14 from keras import backend as K
15
16 batch_size = 128
17 num_classes = 10
18 epochs = 12
19
20 # input image dimensions
21 img_rows, img_cols = 28, 28
22
23 # the data, shuffled and split between train and test sets
24 (x_train, y_train), (x_test, y_test) = mnist.load_data()
25
26 if K.image_data_format() == 'channels_first':
27     x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
28     x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
29     input_shape = (1, img_rows, img_cols)
30 else:
31     x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
32     x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
33     input_shape = (img_rows, img_cols, 1)
34
35 x_train = x_train.astype('float32')
36 x_test = x_test.astype('float32')
37 x_train /= 255
38 x_test /= 255
39 print('x_train shape:', x_train.shape)
40 print(x_train.shape[0], 'train samples')
41 print(x_test.shape[0], 'test samples')
42
43 # convert class vectors to binary class matrices
44 y_train = keras.utils.to_categorical(y_train, num_classes)
45 y_test = keras.utils.to_categorical(y_test, num_classes)
```

Lines 8-21 (see next page) are very similar to what we saw above, importing functionality and defining parameters. Lines 23-33 load the MNIST data (as before) and format the data.

We can see that the input shape to a CNN is three dimensional — the number of rows, number of columns and the number of channels. You will typically have 3 channels indicating three RGB values. For MNIST we have just one channel. So the code checks whether the number of channels is specified first or last in the MNIST data and then shapes the input accordingly. Lines 35-45 are again similar to operations we needed to do for our MLP example, ensure the data have the correct format and the output targets are represented as a categorical matrix of possible outputs.

Next, we define the CNN (see lines 47-57). Our first hidden layer is a two-dimensional convolutional layer with 32 filters, a ReLU activation and a kernel. The kernel, also called filter, is a 3×3 matrix that specifies the area of the image that we are going to look at with each iteration. This is where our weights are in a CNN and what gets updated during training. You can imagine this kernel matrix "sliding" over the image to inspect different areas consecutively. The second hidden layer is very similar but uses a higher number of filters. The third hidden layer is a two-

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

dimensional max pooling layer. The pooling layer reduces the size of the input matrix by using the largest element of each image region. There are other pooling mechanisms (e.g. average pooling which uses the average of a region, and several others [4]) but max pooling has been shown to work well in practice [5]. See [6] for a theoretical comparison. These are the main specifications that differ between a CNN and an MLP.

Afterwards, we just add on a dropout layer, flatten our intermediate representation, add another fully-connected layer, another dropout layer and finally a fully-connected output layer. The output layer is specified in the same way as the output layer of our MLP above.

Finally, the training setup specifies a loss function, optimiser and evaluation metric, and we train the model for 12 epochs. This should reach an accuracy of about 99%.

Exercise 2: Modify the training parameters for the CNN. Using the documentation on `https://keras.io/`, modify some parameters of the neural net and observe the effects on performance. For example,

- Experiment with the type of pooling layer using the layer specifications on `https://keras.io/layers/pooling/`. How does an average pooling layer affect performance, or global max pooling?
- Experiment with different kernel sizes, e.g. 2×2. What is the effect on speed and performance?
- File *image_operations.py* provides some operations you can do on images. Try to flip a random number of images in the training and test set and see how this affects your performance.

References

[1] F. Chollet, "Keras," https://github.com/fchollet/keras, 2016.

[2] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus, "Regularization of Neural Networks using DropConnect," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, S. Dasgupta and D. Mcallester, Eds., vol. 28 (3). JMLR Workshop and Conference Proceedings, May 2013, pp. 1058–1066.

[3] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.

[4] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. Dasgupta and D. McAllester, Eds., vol. 28, no. 3, Atlanta, Georgia, USA, 2013, pp. 1319–1327.

[5] J. Nagi, F. Ducatelle, G. A. D. Caro, D. Ciresan, U. Meier, A. Giusti, F. Nagi, J. Schmidhuber, and L. M. Gambardella, "Max-pooling convolutional neural networks for vision-based hand gesture recognition," 2011.

[6] Y.-L. Boureau, J. Ponce, and Y. Lecun, "A theoretical analysis of feature pooling in visual recognition," in *27th Internationlal Conference on Machine Learning*, Haifa, Israeal, 2010.