CST Part III/MPhil in Advanced Computer Science 2016-2017

Machine Learning and Algorithms for Data Mining
Practical 2: Neural Networks

Demonstrators: Petar Veličković*, Duo Wang
Lecturers: Mateja Jamnik, Pietro Lio', Thomas Sauerwald
University of Cambridge Computer Laboratory
{mateja.jamnik,pietro.lio,thomas.sauerwald}@cl.cam.ac.uk

1 Introduction

The purpose of this practical exercise is leveraging the Keras framework to build, prototype and deploy deep neural networks on the notMNIST[1] dataset (classifying 28×28 grayscale glyphs into ten classes (A–J)) in order to maximise accuracy. Some examples of the glyphs belonging to the `A` class are given below.

The reasons for using this dataset are twofold: its relatively small input size allows for efficient utilisation of CPU resources (for even slightly larger datasets, a GPU is almost always a necessity), but it is a definite ramp-up in complexity compared to the MNIST[2] handwritten digit dataset, which is more of a "sanity check" benchmark (building models which achieve over 99% accuracy on MNIST is now a nearly-trivial task).

2 Environment setup

Deploying the practical scripts is dependent on TensorFlow[3] and Keras. You may install them via `pip install` (Or `pip3` if you are using Python 3), as follows:

```
$ pip install tensorflow
$ pip install keras
```

Once again, it is recommended to use a virtual environment if you are deploying the practical scripts on a Lab-managed machine (and in general!).

3 Deployment

To begin, you need to clone the "starter pack" repository, containing all the files you will need:

```
$ git clone https://github.com/PetarV-/L42-Starter-Pack.git
$ cd L42-Starter-Pack
```

Once this is completed, you need to extract the dataset:

*With many thanks to Nenad Bauk.
[1]`http://yaroslavvb.blogspot.co.uk/2011/09/notmnist-dataset.html`
[2]`http://yann.lecun.com/exdb/mnist/`
[3]We use the TensorFlow back-end for Keras here, as it is significantly better-optimised for CPU usage compared to Theano. Different tasks may benefit the Theano back-end.

```
$ tar xzvf notMNIST_small.tar.gz
```

Afterwards, whenever you want to launch the training subroutine with your latest model, all you need to do is run the `main.py` script:

```
$ python main.py
```

Before starting the training, the summary of the deployed neural network (along with its trainable parameter count) will be pretty-printed for convenience. You will then get a detailed trace of the training progress, followed by a final accuracy level on the (separate) validation dataset.

Your objective is to maximise this accuracy. To do this, you will likely want to modify the following two files:

- `model.py`, which contains the specification of the neural network;
- `main.py`, which will fetch the pre-processed dataset and train the model with the specified optimisation hyperparameters.

The dataset is loaded and preprocessed (normalised to $[0, 1]$ range, labels one-hot encoded, and training/validation split performed) using a separate script, `data.py`. While it will be useful to examine it in the context of any future data cleanup work you might need to do, we recommend not modifying for the purposes of this practical.

4 Optimisation

4.1 Baseline model

The key part of `model.py` is the model specification, contained in the following code section:

```
1 inp = Input(shape=input_shape)
2 flat = Flatten()(inp)
3 hidden_1 = Dense(hidden_size, activation='sigmoid')(flat)
4 hidden_2 = Dense(hidden_size, activation='sigmoid')(hidden_1)
5 out = Dense(nb_classes, activation='softmax')(hidden_2)
6
7 model = Model(input=inp, output=out)
```

We exploit Keras' *functional API* for specifying the model. In this API, each neural network layer is defined as a function, which may either be an `Input`, or consume another layer's output (there are also special "merge" layers that can combine several layers). A Keras `Model` is then specified by determining its input and output layers (as exemplified by line 7).

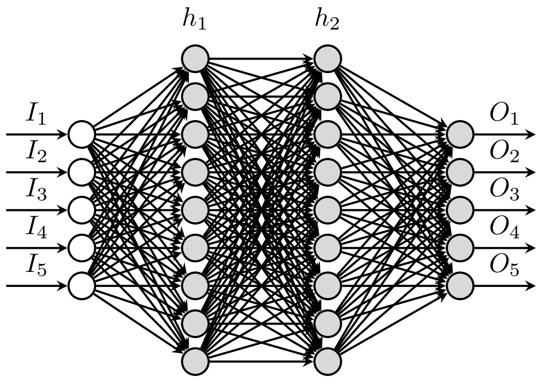
Analysing this model, we may see that it performs the following transformations to the input:

- The input is first *flattened* (from a $28 \times 28 \times 1$ tensor into a 784-dimensional vector) by a `Flatten` layer.
- This is followed by two *fully-connected* (`Dense`) neural network layers of `hidden_size` neurons. This means that every neuron in the layer will receive its own copy of the previous layer's output. Typically, a nonlinear activation is applied before passing the output onwards; in this case, the *logistic sigmoid* function is used:

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{1}$$

- Finally, there is an *output* `Dense` layer, giving us one output per class. In order to perform classification, we need to convert these outputs into probabilities, and for that we exploit a *softmax* activation function, capable of turning any real-valued vector, $\vec{z}$, into a probability distribution, monotonically:

$$\mathbb{P}(\text{class } i) = \text{softmax}(\vec{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{2}$$



Overall, the network's shape can be roughly envisioned as the following (with 784 inputs, 10 outputs, and h_1 and h_2 both having `hidden_size` neurons):
This network, as given in the starter pack, is reminiscent of the kinds of neural networks you might have been able to see in the '90s:

- No *spatial structure* is exploited in the image, as it is immediately flattened into a vector and pixels are treated *independently* from one another.
- All hidden neurons apply the *logistic sigmoid* activation, which suffers from the *vanishing gradient* problem: once its input becomes sufficiently large in magnitude, the output is *saturated* close to ± 1, making the gradients "vanish" and impeding any further learning;
- No *regularisation* is present, making the model prone to overfitting.

All of the above represent possible avenues for improving the network's performance.

4.2 Useful Keras layers

While the Keras documentation (`http://keras.io`) is a superb source of information, here I will highlight the layers that are most likely to be useful to you, along with only their parameters that will most likely be usefully tunable.

4.2.1 Core layers (`keras.layers.*`)

$\sim$ `Flatten()` Flattens the input to a vector representation.

$\sim$ `Dense(h, activation='linear')` A fully-connected layer with `h` neurons. It is fully specified by two trainable parameters:

- a matrix $\mathbf{W}$ of neuron *weights*, of size $h \times n$, where n is the input dimensionality, and h is the number of neurons.
- an h-dimensional vector $\vec{b}$ of neuron *biases*.

For a particular input $\vec{x}$, the layer's output is then $\sigma\left(\mathbf{W}\vec{x} + \vec{b}\right)$, where σ is the activation function. Useful parameters for `activation` include:

- `'linear'` – the *identity* function:
$$\sigma(x) = x \tag{3}$$
- `'sigmoid'` – the *logistic sigmoid* function:
$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{4}$$
- `'tanh'` – the *hyperbolic tangent* function:
$$\sigma(x) = \tanh x \tag{5}$$

- `'relu'` – the *rectified linear* (ReLU) function:

$$\sigma(x) = \max(0, x) \tag{6}$$

- `'softmax'` – the *softmax* function (typically only used for the output layer):

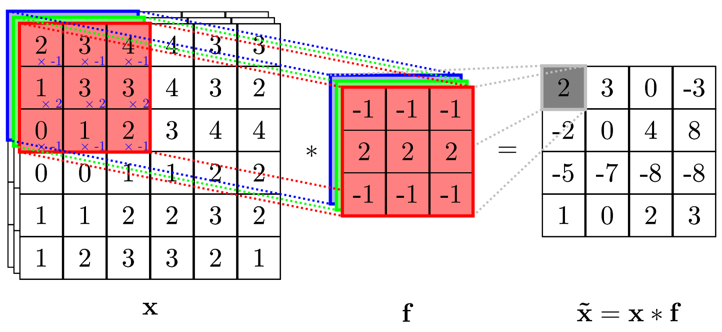
$$\sigma(\vec{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{7}$$

∼ `Convolution2D(d, h, w, activation='linear', border_mode='valid')` A convolutional layer in two dimensions (extending into the third dimension to handle multiple input channels). It generates a two-dimensional output with `d` channels; a single channel is generated as follows:

- For an input, $\mathbf{x}$, with c channels, define a *filter tensor*, $\mathbf{f}$, of size $h \times w \times c$ (where `h` and `w` are passed to the layer constructor, and are typically small—e.g. `h = w = 3`).
- Convolve the input with the filter, adding a *bias* value, b, to each pixel, and then applying an activation function (specified by `activation`) to each pixel.
- To compute the pixel at position (x, y) of this output channel:

$$\tilde{\mathbf{x}}_{xy} = \sigma\left(b + \sum_{i=1}^{h}\sum_{j=1}^{w}\sum_{k=1}^{d} \mathbf{x}_{x+i-1,y+j-1,k} \cdot \mathbf{f}_{ijk}\right) \tag{8}$$

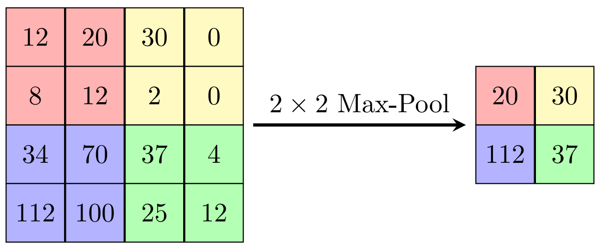
  Or, diagrammatically:



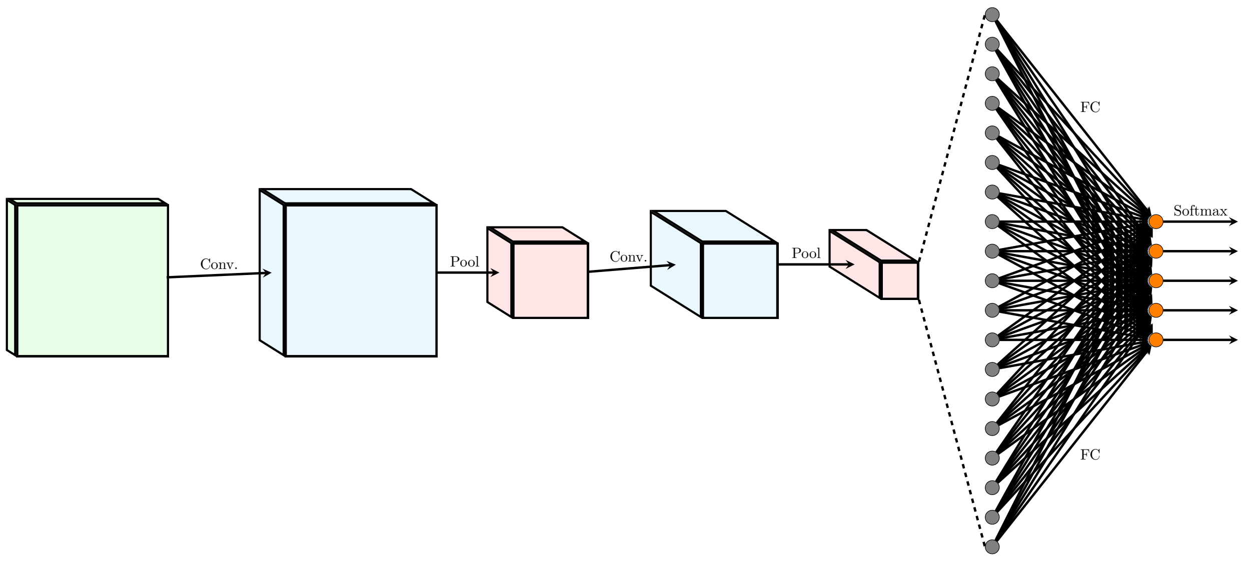
This is repeated (with separate parameters) `d` times in order to produce the final output.

Note that, unless `h = w = 1`, this layer will reduce the height and width of the input. This might be undesirable—setting `border_mode='same'` will *pad* the input with sufficiently many zeroes to ensure the output remains of the same height and width. By default, `border_mode='valid'`, which does not apply any padding.

∼ `MaxPooling2D(pool_size=(2, 2))` Applies an $n \times m$ max-pooling operation to all channels of the input. Typically, $n = m = 2$. This will partition the input into disjoint and $n \times m$ rectangles, and will replace each of them with the maximal value within them, effectively downsampling each input channel. Diagrammatically, for a single input channel:



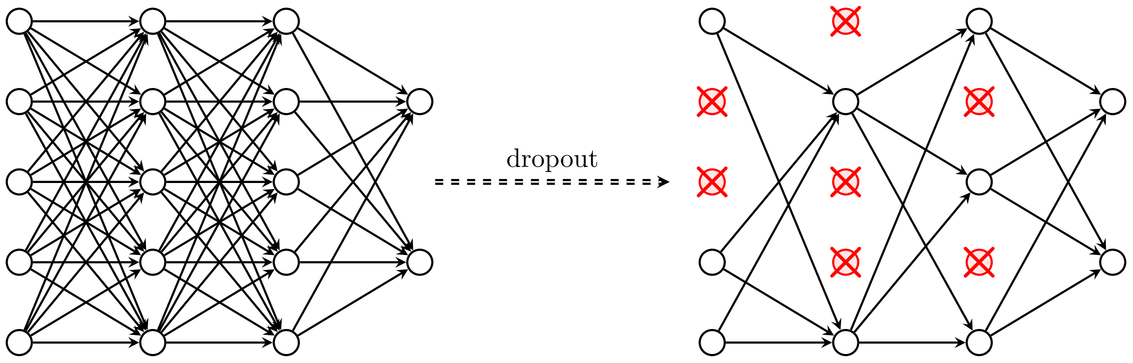
Using these layers, it is possible to construct a common convolutional neural network (CNN). It alternates convolutional and pooling layers, as a very parameter-effective means of extracting useful features from an image (exploiting *spatial structure* of its pixels), until the input is sufficiently downsampled to be flattened, and fed to a traditional MLP for final-stage processing.



4.2.2 Regularisation layers

Convolutional neural networks are extremely prone to *overfitting*. Luckily, two regularisation layers have been particularly effective at overcoming this problem (and are very simple to deploy):

$\sim$ `Dropout(p)` Dropout is a technique that, when applied to a layer, “kills” each of its neurons (by setting their outputs to zero) independently with probability p, during training only. Diagrammatically:



While potentially unintuitive at first sight, this method is a great way of preventing the layer from becoming over-reliant on a particular neuron, and is a very cheap way to average many (smaller) models during training. Good choices lie within $0 < p \leq 0.5$.

$\sim$ `BatchNormalization()` Probably the greatest advance towards training extremely deep networks, this technique (pioneered by Ioffe and Szegedy) was published in February 2015 and has *nearly 1000 citations* at the time of writing this document. It addresses the issue of *internal covariance shift* in deep networks—input statistical properties becoming significantly different as it propagates through the network, making gradient updates difficult to backpropagate. The solution, in the form of a `BatchNormalization` layer, is rather simple—*renormalise* the data within the current training batch, $\mathcal{B} = \{x_1, \ldots, x_m\}$, to have zero mean and unit variance:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{9}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \tag{10}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}} \tag{11}$$

Here, $\varepsilon > 0$ is a small "fuzz" parameter designed to prevent us from dividing by zero (should the sample variance be small). Finally, because this transformation might impede the *generalisation* properties of the network, the final output of the batch normalisation layer is produced by allowing an arbitrary *scale and shift* to the normalised values:

$$y_i = \gamma \hat{x}_i + \beta \tag{12}$$

Here, γ and β are *trainable* parameters! Note that this allows the network to "revert" to the original values, should it find them to be more useful than the normalised ones.

4.3 Model training

Once you have produced a desirable neural network architecture, it can be extremely useful to optimise the *training schedule*. This primarily concerns the following section of `main.py`:

```python
model.compile(loss='categorical_crossentropy', optimizer=SGD(), metrics=['
    accuracy'])

model.fit(X_train, y_train, batch_size=batch_size, nb_epoch=nb_epoch, verbose
    =1, validation_data=(X_test, y_test))
```

Before analysing these lines, it is useful to step back and recall what the training procedure is trying to achieve:

- We have a notion of a *loss function*, $\mathcal{L}$, which measures the "level of error" on the training set, under the current network parameters. This function is differentiable by the weights.
- At each iteration, a mini-batch of examples, $\mathcal{B} = \{(x_1, y_1), \dots, (x_m, y_m)\}$ (where m is the *batch size*), is sampled from the training set. The weights and biases, $\vec{w}$, are then updated according to the *gradient* of the loss function on this minibatch:

$$\vec{w} \leftarrow \vec{w} - \alpha \sum_{i=1}^{m} \frac{\partial \mathcal{L}(y_i, f(x_i; \mathcal{W}))}{\partial \vec{w}} \tag{13}$$

  Here, α is the *learning rate* parameter, and correctly choosing it for each iteration is *critical*! If chosen properly, it will lead to proper decrease of the loss function.
- Typically, the mini-batches are sampled by *shuffling* the training set upfront, and then iterating the aforementioned update going through it, m examples at a time. Covering the entire training set in this way constitutes a single *epoch*. A typical training run will consist of several such epochs.

The first line (`model.compile`) prepares the model for training, by specifying the *loss function* to optimise, the *optimisation algorithm* to use, and the *additional metrics* to report. As we are doing probabilistic classification, the *cross-entropy loss* is the natural loss function to use. It is defined as follows, for a k-class classification problem, network output $\vec{y}$ and ground-truth $\vec{\hat{y}}$:

$$\mathcal{L}(\vec{\hat{y}}, \vec{y}) = \sum_{i=1}^{k} \hat{y}_i \log y_i \tag{14}$$

The optimisation algorithm is responsible for adjusting the value of α at each time-step. The basic SGD optimiser used here applies a fixed learning rate of $\alpha = 0.01$ throughout the entire training run, which is unlikely to be appropriate. In practice, it is usually advisable to make advantage of *adaptive optimisers*, that will take care of adjusting α based on historical updates. In particular, the `'adam'` and `'rmsprop'` algorithms are recommended choices that usually work well without further tuning[4].

The second line (`model.fit`) actually runs the training algorithm. The key parameters to tune here are `batch_size` (specifying the size of a single mini-batch, m) and `nb_epoch` (specifying the number of epochs to train for).

N.B. Keep in mind that, generally, your network **<u>must not observe the test data</u>** until you have *irrevocably committed* yourself to all network parameters and hyperparameters (including architecture and training schedule). Here, `X_test` is used as a *validation* set—a dataset which can be used to influence hyperparameter choices, based on the performance they show on it. This is appropriate for this exercise, as your models will be ultimately tested on an additional dataset, which is *fully hidden* from you!

[4] An excellent overview of these algorithms is given at:
`http://sebastianruder.com/optimizing-gradient-descent/index.html`

4.4 Further hints

While the preceding subsections have likely already given you a lot of ideas to optimise your models, here are a few other interesting aspects which you may wish to look into:

- *L2-regularisation*: an older way to regularise neural networks, by penalising large weights (sometimes also called *weight decay*). It effectively adds a factor of $\frac{\lambda}{2}||\vec{w}||^2$ to the loss function (where λ is a hyperparameter). You may add this method through the `W_regularizer` and `b_regularizer` parameters of any appropriate layers.
- *Network initialisation*: choosing *initial* weight values. For deeper networks, this may mean the difference between having a great model and not converging at all. Keras chooses a very reasonable initialiser by default (Xavier initialisation: `'glorot_uniform'`), but it is not the most appropriate for all kinds of layers—you may look into tweaking the `init` parameter of any layers you deployed.
- *Advanced activations*: While not all research has reached a consensus about the significance of their benefits, several activation functions have been proposed in recent years that offer potential further benefits. These are offered within `keras.layers.advanced_activations.*` and they involve the leaky ReLU (`LeakyReLU`), parametric ReLU (`PReLU`) and the exponential linear unit (`ELU`) functions, among others.

Best of luck!