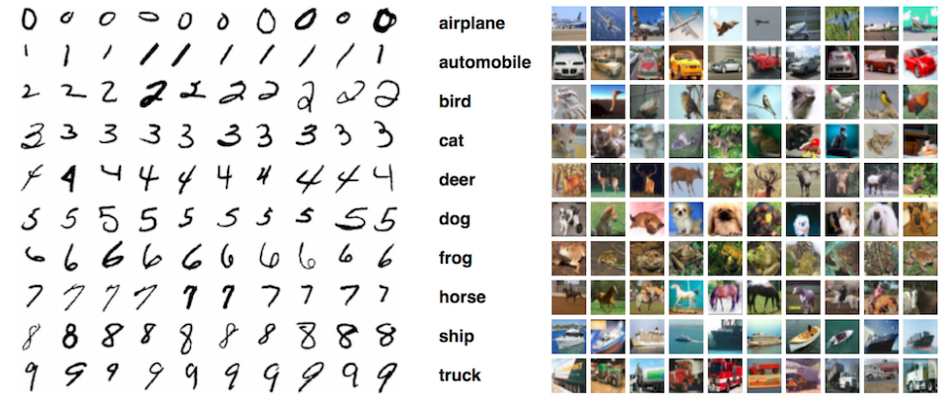
Deep learning for complete beginners: neural network fine-tuning techniques

[Edited on 20 March 2017, to account for API changes introduced by the release of Keras 2]

Introduction

Welcome to the third (and final) in a series of blog posts that is designed to get you quickly up to speed with *deep learning*; from first principles, all the way to discussions of some of the intricate details, with the purposes of achieving respectable performance on two established machine learning benchmarks: MNIST (http://yann.lecun.com/exdb/mnist/) (classification of handwritten digits) and CIFAR-10 (https://www.cs.toronto.edu/~kriz/cifar.html) (classification of small images across 10 distinct classes: plane, car, bird, cat, deer, dog, frog, horse, ship & truck).

MNIST CIFAR-10



Last time around (https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html), I have introduced the *convolutional neural network* model, and illustrated how, combined with a simple but effective regularisation method of *dropout*, it may quickly achieve an accuracy level of 78.6% on CIFAR-10, leveraging the Keras (https://keras.io) deep learning framework.

By now, you have acquired the fundamental skills necessary to apply deep learning to most problems of interest (a notable exception, outside of the scope of these tutorials, is the problem of processing *time-series of arbitrary length*, for which a *recurrent neural network* (RNN) model is often preferable). In this tutorial, I will wrap up with an important but often overlooked aspect of tutorials such as this one – the tips and tricks for properly *fine-tuning* a model, to make it generalise better than the initial baseline you started out with.

This tutorial will, for the most part, assume familiarity with the previous two in the series.

Hyperparameter tuning and the baseline model

Typically, the design process for neural networks starts off by designing a simple network, either directly applying architectures that have shown successes (http://rodrigob.github.io/are_we_there_yet/build/) for similar

problems, or trying out hyperparameter values that generally seem effective. Eventually, we will hopefully attain performance values that seem like a nice baseline starting point, after which we may look into modifying every fixed detail in order to extract the maximal performance capacity out of the network. This is commonly known as *hyperparameter tuning*, because it involves modifying the components of the network which need to be specified before training.

While the methods described here can yield far more tangible improvements on CIFAR-10, due to the relative difficulty of rapid prototyping on it without a GPU, we will focus specifically on improving performance on the MNIST benchmark. Of course, I do invite you to have a go at applying methods like these to CIFAR-10 and see the kinds of gains you may achieve compared to the basic CNN approach, should your resources allow for it.

We will start off with the baseline CNN given below. If you find any aspects of this code unclear, I invite you to familiarise yourself with the previous two tutorials in the series – all the relevant concepts have already been introduced there.

```
from keras.datasets import mnist # subroutines for
fetching the MNIST dataset
from keras.models import Model # basic class for
specifying and training a neural network
from keras.layers import Input, Dense, Flatten,
Convolution2D, MaxPooling2D, Dropout
from keras.utils import np_utils # utilities for
one-hot encoding of ground truth values

batch_size = 128 # in each iteration, we consider 128
training examples at once
num_epochs = 12 # we iterate twelve times over the
entire training set
kernel_size = 3 # we will use 3x3 kernels throughout
pool_size = 2 # we will use 2x2 pooling throughout
conv_depth = 32 # use 32 kernels in both
convolutional layers
drop_prob_1 = 0.25 # dropout after pooling with
probability 0.25
drop_prob_2 = 0.5 # dropout in the FC layer with
probability 0.5
hidden_size = 128 # there will be 128 neurons in both
hidden layers

num_train = 60000 # there are 60000 training examples
in MNIST
num_test = 10000 # there are 10000 test examples in
MNIST

height, width, depth = 28, 28, 1 # MNIST images are
28x28 and greyscale
num_classes = 10 # there are 10 classes (1 per digit)

(X_train, y_train), (X_test, y_test) =
```

```
mnist.load_data() # fetch MNIST data

X_train = X_train.reshape(X_train.shape[0], height,
width, depth)
X_test = X_test.reshape(X_test.shape[0], height,
width, depth)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255 # Normalise data to [0, 1] range
X_test /= 255 # Normalise data to [0, 1] range

Y_train = np_utils.to_categorical(y_train,
num_classes) # One-hot encode the labels
Y_test = np_utils.to_categorical(y_test, num_classes)
# One-hot encode the labels

inp = Input(shape=(height, width, depth)) # N.B.
TensorFlow back-end expects channel dimension last
# Conv [32] -> Conv [32] -> Pool (with dropout on the
pooling layer)
conv_1 = Convolution2D(conv_depth, (kernel_size,
kernel_size), padding='same', activation='relu')(inp)
conv_2 = Convolution2D(conv_depth, (kernel_size,
kernel_size), padding='same',
activation='relu')(conv_1)
pool_1 = MaxPooling2D(pool_size=(pool_size,
pool_size))(conv_2)
drop_1 = Dropout(drop_prob_1)(pool_1)
flat = Flatten()(drop_1)
hidden = Dense(hidden_size, activation='relu')(flat)
# Hidden ReLU layer
drop = Dropout(drop_prob_2)(hidden)
out = Dense(num_classes, activation='softmax')(drop)
# Output softmax layer
```

```
model = Model(inputs=inp, outputs=out) # To define a
model, just specify its input and output layers

model.compile(loss='categorical_crossentropy', #
using the cross-entropy loss function
              optimizer='adam', # using the Adam
optimiser
              metrics=['accuracy']) # reporting the
accuracy

model.fit(X_train, Y_train, # Train the model using
the training set...
          batch_size=batch_size, epochs=num_epochs,
          verbose=1, validation_split=0.1) #
...holding out 10% of the data for validation
model.evaluate(X_test, Y_test, verbose=1) # Evaluate
the trained model on the test set!
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/12
54000/54000 [==============================] - 4s -
loss: 0.3010 - acc: 0.9073 - val_loss: 0.0612 -
val_acc: 0.9825
Epoch 2/12
54000/54000 [==============================] - 4s -
loss: 0.1010 - acc: 0.9698 - val_loss: 0.0400 -
val_acc: 0.9893
Epoch 3/12
54000/54000 [==============================] - 4s -
loss: 0.0753 - acc: 0.9775 - val_loss: 0.0376 -
val_acc: 0.9903
Epoch 4/12
54000/54000 [==============================] - 4s -
loss: 0.0629 - acc: 0.9809 - val_loss: 0.0321 -
val_acc: 0.9913
Epoch 5/12
54000/54000 [==============================] - 4s -
loss: 0.0520 - acc: 0.9837 - val_loss: 0.0346 -
val_acc: 0.9902
Epoch 6/12
54000/54000 [==============================] - 4s -
loss: 0.0466 - acc: 0.9850 - val_loss: 0.0361 -
val_acc: 0.9912
Epoch 7/12
54000/54000 [==============================] - 4s -
loss: 0.0405 - acc: 0.9871 - val_loss: 0.0330 -
val_acc: 0.9917
Epoch 8/12
54000/54000 [==============================] - 4s -
loss: 0.0386 - acc: 0.9879 - val_loss: 0.0326 -
val_acc: 0.9908
Epoch 9/12
```

```
54000/54000 [==============================] - 4s -
loss: 0.0349 - acc: 0.9894 - val_loss: 0.0369 -
val_acc: 0.9908
Epoch 10/12
54000/54000 [==============================] - 4s -
loss: 0.0315 - acc: 0.9901 - val_loss: 0.0277 -
val_acc: 0.9923
Epoch 11/12
54000/54000 [==============================] - 4s -
loss: 0.0287 - acc: 0.9906 - val_loss: 0.0346 -
val_acc: 0.9922
Epoch 12/12
54000/54000 [==============================] - 4s -
loss: 0.0273 - acc: 0.9909 - val_loss: 0.0264 -
val_acc: 0.9930
 9888/10000 [============================>.] - ETA:
0s


[0.026324689089493085, 0.99119999999999997]
```

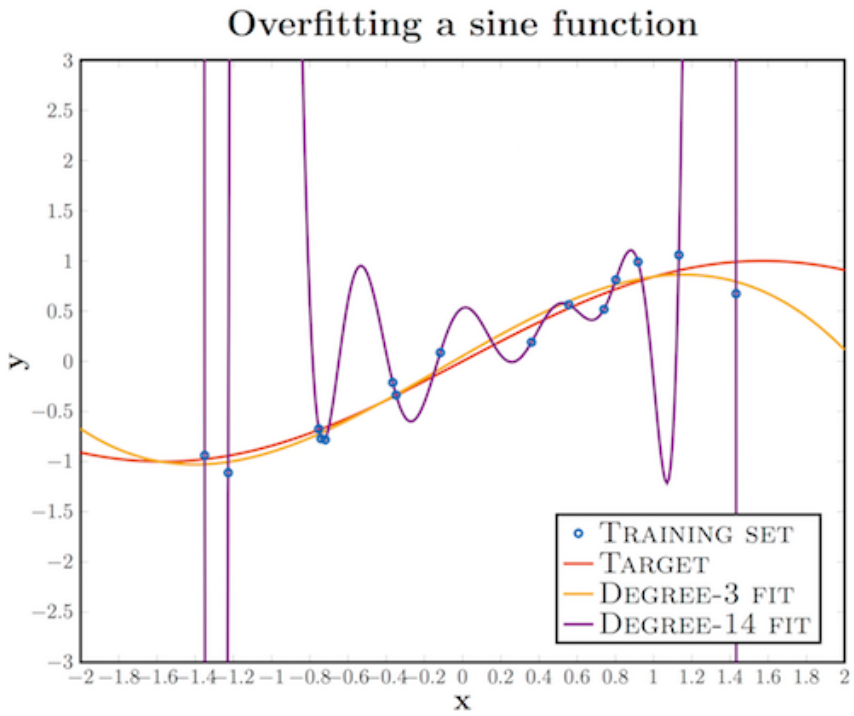
As can be seen, our model achieves an accuracy level of 99.12% on the test set. This is slightly better than the MLP model explored in the first tutorial, but it should be easy to do even better!

The core of this tutorial will explore common ways in which a baseline neural network such as this one can often be improved (we will keep the basic CNN architecture *fixed*), after which we will evaluate the relative gains we have achieved.

L_2 regularisation

As already thoroughly explained in the previous tutorial, one of the primary pitfalls of machine learning is **overfitting**, when the model sometimes catastrophically sacrifices generalisation performance for the pur-

poses of minimising training loss.



Previously, we introduced *dropout* as a very simple way to keep overfitting in check.

There are several other common regularisers that we can apply to our networks. Arguably the most popular out of them is L_2 regularisation (sometimes also called *weight decay*), which takes a more direct approach than dropout for regularising. Namely, a common underlying cause for overfitting is that our model is too complex (in terms of parameter count) for the problem and training set size at hand. A regulariser, in this sense, aims to decrease complexity of the model while maintaining parameter count the same. L_2 regularisation does so by *penalising weights with large magnitudes*, by minimising their L_2 norm, using a hyperparameter λ to specify the relative importance of minimising the norm to minimising the loss on the training set. Introducing this regulariser effectively adds a cost of $\frac{\lambda}{2}||\vec{w}||^2 = \frac{\lambda}{2}\sum_{i=0}^{W} w_i^2$ (the $1/2$ factor is there just for nicer backpropagation updates) to the loss function $\mathcal{L}(\vec{\hat{y}}, \vec{y})$ (in our case, this was the *cross-entropy* loss).

Note that choosing λ properly is important. For too low values, the effect

of the regulariser will be negligible, and for too high values, the optimal model will set all the weights to zero. We will set $\lambda = 0.0001$ here; to add this regulariser to our model, we need an additional import, after which it's as simple as adding a `kernel_regularizer` parameter to each layer we want to regularise:

```
from keras.regularizers import l2 # L2-regularisation
l2_lambda = 0.0001
```

```
conv_1 = Convolution2D(conv_depth, (kernel_size,
kernel_size),
    padding='same',
    kernel_regularizer=l2(l2_lambda),
    activation='relu')(inp)
```

Network initialisation

One issue that was completely overlooked in previous tutorials (and a lot of other existing tutorials as well!) is the policy used for *assigning initial weights* to the various layers within the network. Clearly, this is a very important issue: simply initialising all weights to zero, for example, would significantly impede learning given that no weight would initially be active. Uniform initialisation between ± 1 is also not typically the best way to go – in fact, sometimes (depending on problem and model complexity) choosing the proper initialisation for each layer could mean the difference between superb performance and achieving no convergence at all! Even if the problem does not pose such issues, initialising the weights in an appropriate way can be significantly influential to how easily the network learns from the training set (as it effectively preselects the *initial position* of the model parameters with respect to the loss function to optimise).

Here I will mention two schemes that are particularly of interest:

- *Xavier* (sometimes *Glorot*) (http://jmlr.org/proceedings/papers

/v9/glorot10a/glorot10a.pdf) initialisation: The key idea behind this initialisation scheme is to make it easier for a signal to pass through the layer during forward as well as backward propagation, *for a linear activation* (this also works nicely for *sigmoid* activations because the interval where they are *unsaturated* is roughly linear as well). It draws weights from a probability distribution (uniform or normal) with variance equal to: $\mathbf{Var}(W) = \frac{2}{n_{\text{in}}+n_{\text{out}}}$, where n_{in} and n_{out} are the numbers of neurons in the previous and next layer, respectively.

- *He* (https://arxiv.org/pdf/1502.01852.pdf) initialisation: This scheme is a version of the Xavier initialisation more suitable *for ReLU activations*, compensating for the fact that this activation is zero for half of the possible input space. Namely, $\mathbf{Var}(W) = \frac{2}{n_{\text{in}}}$ in this case.

In order to derive the required variance for the Xavier initialisation, consider what happens to the variance of the output of a linear neuron (ignoring the bias term), based on the variance of its inputs, assuming that the weights and inputs are *uncorrelated*, and are both *zero-mean*:

$$\mathbf{Var}\left(\sum_{i=1}^{n_{\text{in}}} w_i x_i\right) = \sum_{i=1}^{n_{\text{in}}} \mathbf{Var}(w_i x_i) = \sum_{i=1}^{n_{\text{in}}} \mathbf{Var}(W)\mathbf{Var}(X) = n_{\text{in}}\mathbf{Var}(W)\mathbf{Var}(X)$$

This implies that, in order to preserve variance of the input after passing through the layer, it must hold that $\mathbf{Var}(W) = \frac{1}{n_{\text{in}}}$. We may apply a similar argument to the backpropagation update to get that $\mathbf{Var}(W) = \frac{1}{n_{\text{out}}}$. As we cannot typically satisfy these two constraints simultaneously, we set the variance of the weights to their average (i.e., $\mathbf{Var}(W) = \frac{2}{n_{\text{in}}+n_{\text{out}}}$) which usually works quite well in practice.

These two schemes will be sufficient for most examples you will encounter (although the *orthogonal* (http://arxiv.org/pdf/1312.6120v3.pdf) initialisation is also worth investigating in some cases, particularly when initialising recurrent neural networks). Adding a custom initialisation to a layer is simple: you only need to specify an `kernel_initializer` parameter for it, as described below. We will be using the uniform He initialisation

(`he_uniform`) for all ReLU layers and the uniform Xavier initialisation (`glorot_uniform`) for the output softmax layer (as it is effectively a generalisation of the logistic function for multiple inputs).

```
# Add He initialisation to a layer
conv_1 = Convolution2D(conv_depth, (kernel_size,
kernel_size),
    padding='same',
    kernel_initializer='he_uniform',
    kernel_regularizer=l2(l2_lambda),
    activation='relu')(inp)
# Add Xavier initialisation to a layer
out = Dense(num_classes,
    kernel_initializer='glorot_uniform',
    kernel_regularizer=l2(l2_lambda),
    activation='softmax')(drop)
```

Batch normalisation

If there's one technique I would like you to pick up and readily use upon reading this tutorial, it has to be *batch normalisation*, a method for speeding up deep learning pioneered by Ioffe and Szegedy (https://arxiv.org/abs/1502.03167) in early 2015, already accumulating 1200 citations on arXiv (March 2017)! It is based on a really simple failure mode that impedes efficient training of deep networks: as the signal propagates through the network, even if we normalised it in our input, in an intermediate hidden layer it may well end up completely skewed in both mean and variance properties (an effect called the *internal covariance shift* by the original authors), meaning that there will be potentially severe discrepancies between gradient updates across different layers. This requires us to be *more conservative* with our learning rate, and apply stronger regularisers, significantly slowing down learning.

Batch normalisation's answer to this is really quite simple: *normalise* the

activations to a layer to zero mean and unit variance, across the current batch of data being passed through the network (this means that, during training, we normalise across `batch_size` examples, and during testing, we normalise across statistics derived from the *entire training set* – as the testing data cannot be seen upfront). Namely, we compute the mean and variance statistics for a particular batch of activations $\mathcal{B} = \{x_1, \ldots, x_m\}$ as follows:

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

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

We then use these statistics to transform the activations so that they have zero mean and unit variance across the batch, as follows:

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

where $\varepsilon > 0$ is a small "fuzz" parameter designed to protect us from dividing by zero (in an event where the batch standard deviation is very small or even zero). Finally, to obtain the final activations $\boldsymbol{y}$, we need to make sure that we haven't lost any generalisation properties by performing the normalisation – and since the operations we performed on the original data were a scale and shift, we allow for an arbitrary scale and shift on the normalised values to obtain the final acivations (this allows the network, for example, to fall back to the original values if it finds this to be more useful):

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

where β and γ are *trainable* parameters of the batch normalisation operation (can be optimised via gradient descent on the training data). This generalisation also means that batch normalisation can often be usefully applied directly to the *inputs* of a neural network (given that the presence of

these parameters allows the network to assume a different input statistic to the one we selected through manual preprocessing of the data).

This method, when applied to the layers within a deep convolutional neural network almost always achieves significant success with its original design goal of speeding up training. Even further, it acts as a great *regulariser*, allowing us to be far more careless with our choice of learning rate, L_2 regularisation strength and use of dropout (sometimes making it completely *unnecessary*). This regularisation occurs as a consequence of the fact that the output of the network for a single example is *no longer deterministic* (it depends on the entire batch within which it is contained), helping the network generalise easier.

A final piece of analysis: while the authors of batch normalisation suggest performing it *before* applying the activation function of the neuron (on the computed linear combinations of the input data), recently published results (http://arxiv.org/abs/1511.06422) suggest that it might be more beneficial (and at least as good) to do it *after*, which is what we will be doing within this tutorial.

Adding batch normalisation to our network is simple through Keras; expressed by a `BatchNormalization` layer, to which we may provide a few parameters, the most important of which is `axis` (along which axis of the data should the statistics be computed). Specially, when dealing with the inputs to convolutional layers, we would usually like to normalise across individual *channels*, and therefore we set `axis = 3` (which Keras already does by default).

```
from keras.layers.normalization import
BatchNormalization # batch normalisation
# ...
inp_norm = BatchNormalization()(inp) # apply BN to
the input (N.B. need to rename here)
# conv_1 = Convolution2D(...)(inp_norm)
conv_1 = BatchNormalization()(conv_1) # apply BN to
the first conv layer
```

Data augmentation

While the previously discussed methods have all tuned the *model specification*, it is often useful to consider *data-driven* fine-tuning as well – especially when dealing with image recognition tasks.

Imagine that we trained a neural network on handwritten digits which all, roughly, had the same bounding box, and were nicely oriented. Now consider what happens when someone presents the network with a slightly shifted, scaled and rotated version of a training image to test on: its confidence in the correct class is bound to drop. We would ideally want to instruct the model to remain invariant under feasible levels of such *distortions*, but our model can only learn from the samples we provided to it, given that it performs a kind of statistical analysis and extrapolation from the training set!

Luckily, there is a very simple remedy to this problem which is often quite effective, especially on image recognition tasks: artificially *augment* the data with distorted versions during training! This means that, prior to feeding an example to the network for training, we will apply any transformations to it that we find appropriate, and therefore allow the network to directly observe the effects of applying them on data and instructing it to behave better on such examples. For illustration purposes, here are a few shifted/scaled/sheared/rotated examples of MNIST digits:

shift	shift	shear	shift & scale	rotate & scale
				

Keras provides a really nice interface to image data augmentation by way of the `ImageDataGenerator` class. We initialise the class by providing it with the kinds of transformations we want performed to every image, and then feed our training data through the generator, by way of performing a call to its `fit` method followed by its `flow` method, returning an (infinitely-extending) iterator across augmented batches. There is even a custom `model.fit_generator` method which will directly perform training of our model using this iterator, simplifying the code significantly! A slight downside is that we now lose the `validation_split` parameter, meaning we have to separate the validation dataset ourselves, but this adds only four extra lines of code.

For this tutorial, we will apply random horizontal and vertical shifts to the data. `ImageDataGenerator` also provides us with methods for applying random rotations, scales, shears and flips. These should all also be sensible transformations to attempt, except for the flips, due to the fact that we are not ever expecting to receive flipped handwritten digits from a person.

```
from keras.preprocessing.image import
ImageDataGenerator # data augmentation
# ... after model.compile(...)
# Explicitly split the training and validation sets
X_val = X_train[54000:]
Y_val = Y_train[54000:]
X_train = X_train[:54000]
Y_train = Y_train[:54000]

datagen = ImageDataGenerator(
            width_shift_range=0.1, # randomly shift
images horizontally (fraction of total width)
            height_shift_range=0.1) # randomly shift
images vertically (fraction of total height)
datagen.fit(X_train)

# fit the model on the batches generated by
datagen.flow() - most parameters similar to model.fit
model.fit_generator(datagen.flow(X_train, Y_train,
                        batch_size=batch_size),

steps_per_epoch=X_train.shape[0],
                        epochs=num_epochs,
                        validation_data=(X_val,
Y_val),
                        verbose=1)
```

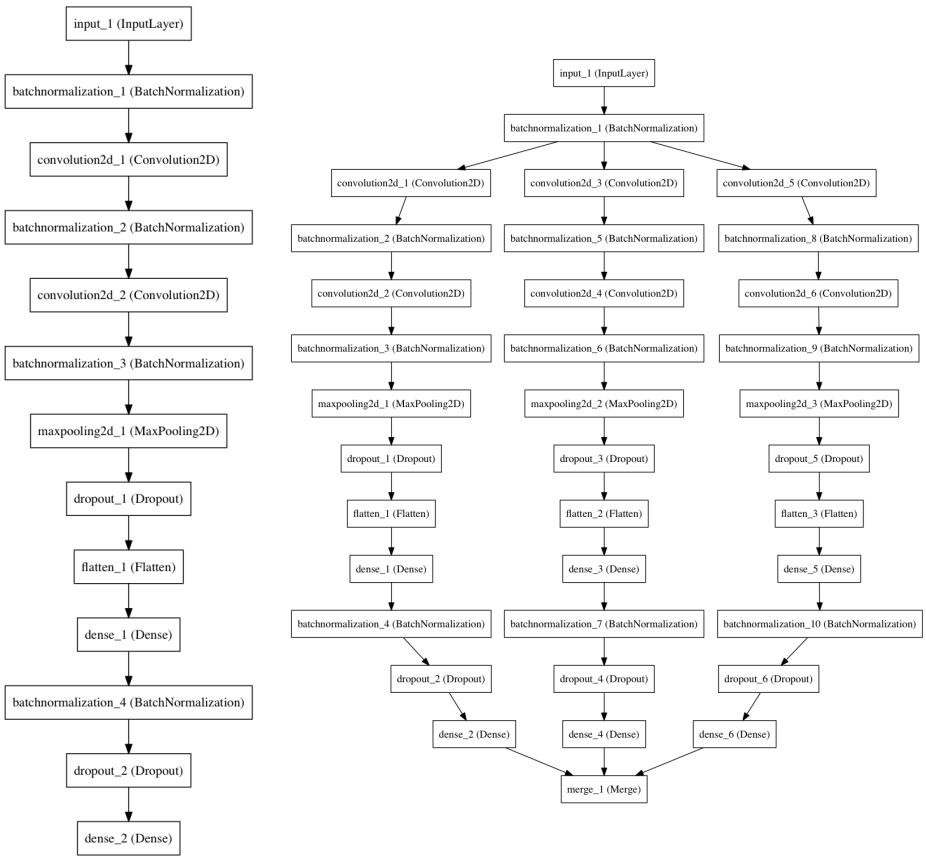
Ensembles

One interesting aspect of neural networks that you might observe when using them for classification (on more than two classes) is that, when trained from different initial conditions, they will express better *discriminative properties* for different classes, and will tend to get more confused on

others. On the MNIST example, you might find that a single trained network becomes very good at distinguishing a three from a five, but as a consequence does not learn to distinguish ones from sevens properly; while another trained network could well do the exact opposite.

This discrepancy may be exploited through an *ensemble* method – rather than building just *one* model, build several *copies* of it (with different initial values), and average their predictions on a particular input to obtain the final answer. Here we will perform this across three separate models. The difference between the two architectures can be easily visualised by a diagram such as the one below (plotted within Keras (https://keras.io/visualization/)!):

Baseline (with BN) Ensemble



Keras once again provides us with a way to effectively do this at minimial expense to code length - we may wrap the constituent models' construc-

tions in a loop, extracting just their outputs for a final three-way `average` layer.

```
from keras.layers import average # for averaging
predictions in an ensemble
# ...
ens_models = 3 # we will train three separate models
on the data
# ...
inp_norm = BatchNormalization()(inp) # Apply BN to
the input (N.B. need to rename here)

outs = [] # the list of ensemble outputs
for i in range(ens_models):
    # conv_1 = Convolution2D(...)(inp_norm)
    # ...
    outs.append(Dense(num_classes,
        kernel_initializer='glorot_uniform',
        kernel_regularizer=l2(l2_lambda),
        activation='softmax')(drop)) # Output softmax
layer

out = average(outs) # average the predictions to
obtain the final output
```

Early stopping

I will discuss one further method here, as an introduction to a much wider area of *hyperparameter optimisation*. Namely, thus far we have utilised the validation dataset just to monitor training progress, which is arguably wasteful (given that we don't do anything constructive with this data, other than observe successive losses on it). In fact, validation data represents the primary platform for evaluating hyperparameters of the network (such as depth, neuron/kernel numbers, regularisation factors, etc.). We may

imagine running our network with different combinations of hyperparameters we wish to optimise, and then basing our decisions on their performance on the validation set. Keep in mind that we *may not observe the test dataset* until we have *irrevocably committed* ourselves to all hyperparameters, given that otherwise features of the test set may inadvertently flow into the training procedure! This is sometimes known as the *golden rule of machine learning*, and breaking it was a common failure of many early approaches.

Perhaps the simplest use of the validation set is for tuning the *number of epochs*, through a procedure known as *early stopping*; simply stop training once the validation loss hasn't decreased for a fixed number of epochs (a parameter known as *patience*). As this is a relatively small benchmark which saturates quickly, we will have a patience of five epochs, and increase the upper bound on epochs to 50 (which will likely never be reached).

Keras supports early stopping through an `EarlyStopping` *callback* class. Callbacks are methods that are called after each epoch of training, upon supplying a `callbacks` parameter to the `fit` or `fit_generator` method of the model. As usual, this is very concise, adding only a single line to our program.

```
from keras.callbacks import EarlyStopping
# ...
num_epochs = 50 # we iterate at most fifty times over
the entire training set
# ...
# fit the model on the batches generated by
datagen.flow() - most parameters similar to model.fit
model.fit_generator(datagen.flow(X_train, Y_train,
                        batch_size=batch_size),

steps_per_epoch=X_train.shape[0],
                        epochs=num_epochs,
                        validation_data=(X_val,
Y_val),
                        verbose=1,

callbacks=[EarlyStopping(monitor='val_loss',
patience=5)]) # adding early stopping
```

Just show me the code! (also, how well does it do?)

With these six techniques applied to our original baseline, the final version of your code should look something like the following:

```
from keras.datasets import mnist # subroutines for
fetching the MNIST dataset
from keras.models import Model # basic class for
specifying and training a neural network
from keras.layers import Input, Dense, Flatten,
Convolution2D, MaxPooling2D, Dropout, average
from keras.utils import np_utils # utilities for
one-hot encoding of ground truth values
from keras.regularizers import l2 # L2-regularisation
from keras.layers.normalization import
BatchNormalization # batch normalisation
from keras.preprocessing.image import
ImageDataGenerator # data augmentation
from keras.callbacks import EarlyStopping # early
stopping

batch_size = 128 # in each iteration, we consider 128
training examples at once
num_epochs = 50 # we iterate at most fifty times over
the entire training set
kernel_size = 3 # we will use 3x3 kernels throughout
pool_size = 2 # we will use 2x2 pooling throughout
conv_depth = 32 # use 32 kernels in both
convolutional layers
drop_prob_1 = 0.25 # dropout after pooling with
probability 0.25
drop_prob_2 = 0.5 # dropout in the FC layer with
probability 0.5
hidden_size = 128 # there will be 128 neurons in both
hidden layers
l2_lambda = 0.0001 # use 0.0001 as a
L2-regularisation factor
ens_models = 3 # we will train three separate models
on the data
```

```
num_train = 60000 # there are 60000 training examples
in MNIST
num_test = 10000 # there are 10000 test examples in
MNIST

height, width, depth = 28, 28, 1 # MNIST images are
28x28 and greyscale
num_classes = 10 # there are 10 classes (1 per digit)

(X_train, y_train), (X_test, y_test) =
mnist.load_data() # fetch MNIST data

X_train = X_train.reshape(X_train.shape[0], height,
width, depth)
X_test = X_test.reshape(X_test.shape[0], height,
width, depth)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

Y_train = np_utils.to_categorical(y_train,
num_classes) # One-hot encode the labels
Y_test = np_utils.to_categorical(y_test, num_classes)
# One-hot encode the labels

# Explicitly split the training and validation sets
X_val = X_train[54000:]
Y_val = Y_train[54000:]
X_train = X_train[:54000]
Y_train = Y_train[:54000]

inp = Input(shape=(height, width, depth)) # N.B.
TensorFlow back-end expects channel dimension last
inp_norm = BatchNormalization()(inp) # Apply BN to
```

```
    the input (N.B. need to rename here)

    outs = [] # the list of ensemble outputs
    for i in range(ens_models):
        # Conv [32] -> Conv [32] -> Pool (with dropout on
    the pooling layer), applying BN in between
        conv_1 = Convolution2D(conv_depth, (kernel_size,
    kernel_size),
            padding='same',
            kernel_initializer='he_uniform',
            kernel_regularizer=l2(l2_lambda),
            activation='relu')(inp_norm)
        conv_1 = BatchNormalization()(conv_1)
        conv_2 = Convolution2D(conv_depth, (kernel_size,
    kernel_size),
            padding='same',
            kernel_initializer='he_uniform',
            kernel_regularizer=l2(l2_lambda),
            activation='relu')(conv_1)
        conv_2 = BatchNormalization()(conv_2)
        pool_1 = MaxPooling2D(pool_size=(pool_size,
    pool_size))(conv_2)
        drop_1 = Dropout(drop_prob_1)(pool_1)
        flat = Flatten()(drop_1)
        hidden = Dense(hidden_size,
            kernel_initializer='he_uniform',
            kernel_regularizer=l2(l2_lambda),
            activation='relu')(flat) # Hidden ReLU layer
        hidden = BatchNormalization()(hidden)
        drop = Dropout(drop_prob_2)(hidden)
        outs.append(Dense(num_classes,
            kernel_initializer='glorot_uniform',
            kernel_regularizer=l2(l2_lambda),
            activation='softmax')(drop)) # Output softmax
```

```
layer

out = average(outs) # average the predictions to
obtain the final output

model = Model(inputs=inp, outputs=out) # To define a
model, just specify its input and output layers

model.compile(loss='categorical_crossentropy', #
using the cross-entropy loss function
              optimizer='adam', # using the Adam
optimiser
              metrics=['accuracy']) # reporting the
accuracy

datagen = ImageDataGenerator(
        width_shift_range=0.1,  # randomly shift
images horizontally (fraction of total width)
        height_shift_range=0.1)  # randomly shift
images vertically (fraction of total height)
datagen.fit(X_train)

# fit the model on the batches generated by
datagen.flow() - most parameters similar to model.fit
model.fit_generator(datagen.flow(X_train, Y_train,
                        batch_size=batch_size),

steps_per_epoch=X_train.shape[0],
                        epochs=num_epochs,
                        validation_data=(X_val,
Y_val),
                        verbose=1,

callbacks=[EarlyStopping(monitor='val_loss',
```

```
patience=5)]) # adding early stopping

model.evaluate(X_test, Y_test, verbose=1) # Evaluate
the trained model on the test set!
```

```
Epoch 1/50
54000/54000 [==============================] - 30s -
loss: 0.3487 - acc: 0.9031 - val_loss: 0.0579 -
val_acc: 0.9863
Epoch 2/50
54000/54000 [==============================] - 30s -
loss: 0.1441 - acc: 0.9634 - val_loss: 0.0424 -
val_acc: 0.9890
Epoch 3/50
54000/54000 [==============================] - 30s -
loss: 0.1126 - acc: 0.9716 - val_loss: 0.0405 -
val_acc: 0.9887
Epoch 4/50
54000/54000 [==============================] - 30s -
loss: 0.0929 - acc: 0.9757 - val_loss: 0.0390 -
val_acc: 0.9890
Epoch 5/50
54000/54000 [==============================] - 30s -
loss: 0.0829 - acc: 0.9788 - val_loss: 0.0329 -
val_acc: 0.9920
Epoch 6/50
54000/54000 [==============================] - 30s -
loss: 0.0760 - acc: 0.9807 - val_loss: 0.0315 -
val_acc: 0.9917
Epoch 7/50
54000/54000 [==============================] - 30s -
loss: 0.0740 - acc: 0.9824 - val_loss: 0.0310 -
val_acc: 0.9917
Epoch 8/50
54000/54000 [==============================] - 30s -
loss: 0.0679 - acc: 0.9826 - val_loss: 0.0297 -
val_acc: 0.9927
Epoch 9/50
54000/54000 [==============================] - 30s -
```

```
loss: 0.0663 - acc: 0.9834 - val_loss: 0.0300 -
val_acc: 0.9908
Epoch 10/50
54000/54000 [==============================] - 30s -
loss: 0.0658 - acc: 0.9833 - val_loss: 0.0281 -
val_acc: 0.9923
Epoch 11/50
54000/54000 [==============================] - 30s -
loss: 0.0600 - acc: 0.9844 - val_loss: 0.0272 -
val_acc: 0.9930
Epoch 12/50
54000/54000 [==============================] - 30s -
loss: 0.0563 - acc: 0.9857 - val_loss: 0.0250 -
val_acc: 0.9923
Epoch 13/50
54000/54000 [==============================] - 30s -
loss: 0.0530 - acc: 0.9862 - val_loss: 0.0266 -
val_acc: 0.9925
Epoch 14/50
54000/54000 [==============================] - 31s -
loss: 0.0517 - acc: 0.9865 - val_loss: 0.0263 -
val_acc: 0.9923
Epoch 15/50
54000/54000 [==============================] - 30s -
loss: 0.0510 - acc: 0.9867 - val_loss: 0.0261 -
val_acc: 0.9940
Epoch 16/50
54000/54000 [==============================] - 30s -
loss: 0.0501 - acc: 0.9871 - val_loss: 0.0238 -
val_acc: 0.9937
Epoch 17/50
54000/54000 [==============================] - 30s -
loss: 0.0495 - acc: 0.9870 - val_loss: 0.0246 -
val_acc: 0.9923
```

```
Epoch 18/50
54000/54000 [==============================] - 31s -
loss: 0.0463 - acc: 0.9877 - val_loss: 0.0271 -
val_acc: 0.9933
Epoch 19/50
54000/54000 [==============================] - 30s -
loss: 0.0472 - acc: 0.9877 - val_loss: 0.0239 -
val_acc: 0.9935
Epoch 20/50
54000/54000 [==============================] - 30s -
loss: 0.0446 - acc: 0.9885 - val_loss: 0.0226 -
val_acc: 0.9942
Epoch 21/50
54000/54000 [==============================] - 30s -
loss: 0.0435 - acc: 0.9890 - val_loss: 0.0218 -
val_acc: 0.9947
Epoch 22/50
54000/54000 [==============================] - 30s -
loss: 0.0432 - acc: 0.9889 - val_loss: 0.0244 -
val_acc: 0.9928
Epoch 23/50
54000/54000 [==============================] - 30s -
loss: 0.0419 - acc: 0.9893 - val_loss: 0.0245 -
val_acc: 0.9943
Epoch 24/50
54000/54000 [==============================] - 30s -
loss: 0.0423 - acc: 0.9890 - val_loss: 0.0231 -
val_acc: 0.9933
Epoch 25/50
54000/54000 [==============================] - 30s -
loss: 0.0400 - acc: 0.9894 - val_loss: 0.0213 -
val_acc: 0.9938
Epoch 26/50
54000/54000 [==============================] - 30s -
```

```
loss: 0.0384 - acc: 0.9899 - val_loss: 0.0226 -
val_acc: 0.9943
Epoch 27/50
54000/54000 [==============================] - 30s -
loss: 0.0398 - acc: 0.9899 - val_loss: 0.0217 -
val_acc: 0.9945
Epoch 28/50
54000/54000 [==============================] - 30s -
loss: 0.0383 - acc: 0.9902 - val_loss: 0.0223 -
val_acc: 0.9940
Epoch 29/50
54000/54000 [==============================] - 31s -
loss: 0.0382 - acc: 0.9898 - val_loss: 0.0229 -
val_acc: 0.9942
Epoch 30/50
54000/54000 [==============================] - 31s -
loss: 0.0379 - acc: 0.9900 - val_loss: 0.0225 -
val_acc: 0.9950
Epoch 31/50
54000/54000 [==============================] - 30s -
loss: 0.0359 - acc: 0.9906 - val_loss: 0.0228 -
val_acc: 0.9943
10000/10000 [==============================] - 2s


[0.017431972888592554, 0.9947000000000003]
```

Our updated model achieves an accuracy of 99.47% on the test set, a significant improvement from the baseline performance of 99.12%. Of course, for such a small and (comparatively) simple problem such as MNIST, the gains might not immediately seem that important. Applying these techniques to problems like CIFAR-10 (provided you have sufficient resources) can yield much more tangible benefits.

I invite you to work on this model even further: specifically, make advan-

tage of the validation data to do more than just early stopping: use it to evaluate various kernel sizes/counts, hidden layer sizes, optimisation strategies, activation functions, number of networks in the ensemble, etc. and see how this makes you compare to the best of the best (http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#4d4e495354) (at the time of writing this post, the top-ranked model achieves an accuracy of 99.79% on the MNIST test set).

Conclusion

Throughout this post we have covered six methods that can help further fine-tune the kinds of deep neural networks discussed in the previous two tutorials:

- L_2 regularisation
- Initialisation
- Batch normalisation
- Data augmentation
- Ensemble methods
- Early stopping

and successfully applied them to a baseline deep CNN model within Keras, achieving a significant improvement on MNIST, all in under 90 lines of code.

This is also the final topic of the series. I hope that what you've learnt here is enough to provide you with an initial drive that, combined with appropriate targeted resources, should see you become a bleeding-edge deep learning engineer in no time!

If you have any feedback on the series as a whole, wishes for future tutorials, or would just like to say hello, feel free to email me: *petar [dot] velickovic [at] cl [dot] cam [dot] ac [dot] uk*. Hopefully, there will be more tutorials to come from me on this subject, perhaps focusing on topics such as recurrent neural networks or deep reinforcement learning.

Thank you!

ABOUT THE AUTHOR

Petar Veličković

Petar is currently a Research Assistant in Computational Biology within the Artificial Intelligence Group of the Cambridge University Computer Laboratory, where he is working on developing machine learning algorithms on complex networks, and their applications to bioinformatics. He is also a PhD student within the group, supervised by Dr Pietro Liò and affiliated with Trinity College. He holds a BA degree in Computer Science from the University of Cambridge, having completed the Computer Science Tripos in 2015.

 (https://www.facebook.com/cambridgespark) 

(https://twitter.com/CambridgeSpark) 

(https://www.youtube.com/channel/UCahjykidQbxUbEVbfN-pd7A)  (https://www.linkedin.com/company/10996616/)

/COM.CAMBRIDGESPARK.PDF/CAMBRIDGESPARKTCS.PDF) | PRIVACY POLICY (HTTPS://S3-EU-WEST-1.AMAZONAWS.COM /COM.CAMBRIDGESPARK.PDF /CAMBRIDGESPARKPRIVACYPOLICY.PDF)