Project 4: CIFAR Image Classification

Background

This project is a bit different: You will be using Python and making use of the Keras library, with the Tensorflow backend. Tensorflow is a code library for high performance numerical computation, and is commonly used for deep learning. It can make us of GPUs, but this is not necessary. Keras is a simpler interface for Tensorflow, but it is still a complex library that will take some effort to learn to use.

Setting Things Up

Setting up your system to run this project will require installing Python, the Keras package for Python and the Tensorflow backend to Keras. Note that although Keras supports other backends, it is required that you use Tensorflow. Installing these and making them work on your system (particularly Tensorflow) can take some time depending on your system and what your Python IDE is, so give yourself a few hours and follow online tutorials.

The Task

Project four is a simple image classification task: You need to build and train a convolutional neural network so that it can classify images according to their content. We will use the CIFAR10 dataset. This contains small (32x32) color images, consisting of 10 classes. A data wrapping CIFAR class has been created for use in this project (it is included in the project code file). This class will divide the data into training, validation and test cases. You will train a CNN using the training/validation data and we will evaluate your network's performance on the test data. The division of data into training, validation and test depends on a random seed. When we evaluate your code, we will choose a random seed to make this division.

You will implement the two functions to be passed to the runImageClassification function as the arguments 'getModel' and 'fitModel'. You should call your functions 'myGetModel' and 'myFitModel'.

The getModel function should accept an object of the CIFAR class, and return a compiled Keras CNN model. In this function you will specify the network structure (including regularization) and the optimizer to be used (and its parameters like learning rate), and run compile the model (in the Keras sense of running model.compile).

The fitModel function should accect two arguments. The first is the CNN model you return from your getModel function, and the second is the CIFAR classed data object. It will return a trained Keras CNN model, which will then be applied to the test data. In this function you will train the model, using the Keras model.fit function. You will need to specify all

parameters of the training algorithm (batch size, etc), and the callbacks you will use (EarlyStopping and ModelCheckpoint). You will need to make sure you save and load into the model the weight values of its best performing epoch.

The CIFAR Class

Training, Validation & Test Data

The CIFAR object will automatically split the data into training, validation and test data. These should be used as they are intended. Your task is to train on the training data, and validate this training using the validation data, so as to perform as well as possible on the test data. The test data should never be used in any part of the training algorithm **and it will not be available to you code during evaluation runs**. Note that this split depends on a random seed, and so it will be different when your code is evaluated.

The data is available in the following fields:

Field	Description
x_train	The input features (scaled pixel values) of the training cases.
y_train	The classes of the training cases.
x_validation	The input features (scaled pixel values) of the validation cases.
y_validation	The classes of the validation cases.
x_test	The input features (scaled pixel values) of the test cases. **Note that this field will be blank during your code execution in the evaluation run.** So you cannot peek at the test data when training!
y_test	The classes of the test cases. **Note that this field will be blank during your code execution in the evaluation run.** So you cannot peek at the test data when training!

The data made available is slightly different from that in the CIFAR10 dataset. As noted in the table, the input RGB integer pixel values have been scaled and turned into floating point numbers. Likewise the integer y values have been registered as factors. Essentially I have just made them ready for use in a Keras neural network. Raw values are available in the x_train_raw (etc) fields if you want to look at them.

Viewing Images

The CIFAR class also lets you look at some random images (taken from the cases placed in validation data). These images are labelled with their classes. To do this, use the CIFAR showImages method.

Gradient Descent Options

In Keras, the *optimizer* object controls the gradient descent optimization (and in theory alternative optimization methods). Choosing different optimizers will allow different options for how weights are updated each iteration of the gradient descent algorithm, such as how the learning rate is adjusted over time. You should research these, and try (and tune) different alternatives. The SGD optimizer is the simplest gradient descent optimizer.

Early Stopping

You should use early-stopping, both in the sense of using the model where validation performance peaks, and in the sense of stopping the training once it is clear that the performance has peaked.

Keras unfortunately splits these two things. The early stopping callback can be used to stop training once it is clear validation performance has peaked, but even when used the fitting process will return the weights at the end of training, not from the epoch where validation performance peaked.* But the ModelCheckpoint callback can be used to save the best weights found during the fitting process using a target score (e.g. validation accuracy) in a file. At the end of the fitting process you can then load these weights using load_model.

The end result is you should use both of these callbacks.

Do note that when you set up the early-stopping callback you shouldn't stop immediately when the validation performance deteriorates. Instead run more epochs to check if it starts improving again. This is because the amount of validation is small and the performance will fluctuate rather noisily, and there will be downticks that occur even when the model is improving. You can implement this 'wait and see' approach using the patience argument in Keras' early stopping callback.

When you use the ModelCheckpoint callback, only save the best weights in a file. DO NOT save the weights every time they improve. Call your file "weights-best.hdf5".

* This has actually recently changed, such that the early stopping appears to be able to cause the fitting procedure to return the weights at peak validation performance using a return_best_weights arguments. Sadly this is not included in the latest release version, only on the GitHub master.

Regularization

In addition to early stopping, you should introduce some form of intra-model regularization. Using drop-out is common and easy: Add a number of drop-out layers to your model (after your convolution layers). Experiment with different drop-out values. L1 and L2 regularization are also easy to do with Keras.

Max Epochs

You MUST set the epochs number in the fit function to 100 or less. This will terminate the training after this number of epochs, regardless of what is happening. We cannot cope with extremely long training runs (since we have so many groups to evaluate).

Par Function

The performance of the par functions (with the default seed 7) is .685. You need to match or beat this to pass.

Note that the par function here is designed to not be very good (a common online network was made worse and used) and beating it is not much of an achievement. In fact, just changing the optimizer (again without tuning anything) improves things considerably. You should be able to get results over .75 without a great deal of effort.

So if you have time, instead of thinking of the par function network as a challenge to match, think of it as a starting point: You can use the information below to set it up (you'll have to work out how to do this using Keras) and improve matters from there.

Par Function Network

The par function network uses:

1. The following network structure:
    - Convolution (32 3x3 filters with relu activation and .5 dropout)
    - Convolution (32 3x3 filters with relu activation, followed by 2x2 max pooling and .25 dropout)
    - Convolution (64 3x3 filters with relu activation and .5 dropout)
    - Convolution (64 3x3 filters with relu activation, followed by 2x2 max pooling and .25 dropout). The result is flattened.
    - A fully connected layer with relu activation and .5 dropout
    - A softmax output layer.
2. An SGD optimizer, with an initial learning rate of .1 and decay of 1e-6.
3. Early stopping and model checkpoint callbacks as described above. Both are tracking validation accuracy, and the early stopping has 20 patience