My solution for the Galaxy Zoo challenge

Sander Dieleman

April 16, 2014

Abstract

This document describes my solution for the Galaxy Challenge on Kaggle. I had previously documented my solution in a blog post[1]. Many parts of this document are taken from that post. This document additionally provides information on how to use the code, which has been made publicly available under a BSD 3-clause licence in accordance with the requirements for prize eligibility.

Personal details

- Name: Sander Dieleman
- Location: Ghent, Belgium
- Email: sanderdieleman@gmail.com
- Competition: Galaxy Zoo - The Galaxy Challenge[2]

Contents

[1]`http://benanne.github.io/2014/04/05/galaxy-zoo.html`

[2]`http://www.kaggle.com/c/galaxy-zoo-the-galaxy-challenge`

1 Summary

My solution is based around *convolutional neural networks* (convnets). I used fairly large models (the largest network has about 42 million parameters) with 5 to 7 layers, and as a result my main focus during the competition was combatting overfitting. I did this by using data augmentation (creating new datapoints from existing datapoints by transformations which the target values should be invariant to), dropout and weight norm constraints, and modifying the network architecture to increase parameter sharing. I trained all networks with stochastic gradient descent with Nesterov momentum and used GPU acceleration to speed up experiments. My final submission was an ensemble of 17 models, which differed in architecture and input preprocessing. No external data sources were used.

2 Models and training

2.1 Preprocessing and data augmentation

2.1.1 Cropping and downsampling

The data consisted of 424x424 colour JPEG images, along with 37 weighted probabilities that have to be predicted for each image[3]. For almost all of the images, the interesting part was in the center, so I cropped all images to 207x207. I then downsampled them 3x to 69x69, to keep the input size of the network manageable.

2.1.2 Exploiting spatial invariances

Images of galaxies are rotation invariant. They are also scale invariant and translation invariant to a limited extent. All of these invariances could be exploited to do data augmentation. Each training example was perturbed before presenting it to the network by randomly scaling it, rotating it, translating it and optionally flipping it. I used the following parameter ranges:

- rotation: random with angle between $0°$ and $360°$ (uniform)
- translation: random with shift between -4 and 4 pixels (relative to the original image size of 424x424) in the x and y direction (uniform)
- zoom: random with scale factor between 1/1.3 and 1.3 (log-uniform)
- flip: yes or no (bernoulli)

Because both the initial downsampling to 69x69 and the random perturbation are affine transforms, they could be combined into one affine transformation step. This sped up things significantly and reduced information loss.

2.1.3 Colour perturbation

After this, the colour of the images was changed as described by Krizhevsky et al. [4], with two differences: the first component had a much larger eigenvalue than the other two, so only this one was used, and the standard deviation for the scale factor alpha was set to 0.5.

2.1.4 'Realtime' augmentation

Combining downsampling and perturbation into a single affine transform made it possible to do data augmentation in realtime, i.e. during training. This significantly reduced overfitting because the network would never see the exact same image twice. While the network was being trained on a chunk of data on the GPU, the next chunk would be generated on the CPU in multiple processes, to ensure that all the available cores were used.

2.1.5 Centering and rescaling

I experimented with centering and rescaling the galaxy images based on parameters extracted with *sextractor*[4]. Although this didn't improve performance, including a few models that used it in the final ensemble helped to increase variance. I extracted the center of the galaxies, as well as the

[3]for details on the weighting scheme, please refer to `http://www.kaggle.com/c/galaxy-zoo-the-galaxy-challenge/details/the-galaxy-zoo-decision-tree`.

[4]`https://www.astromatic.net/software/sextractor`

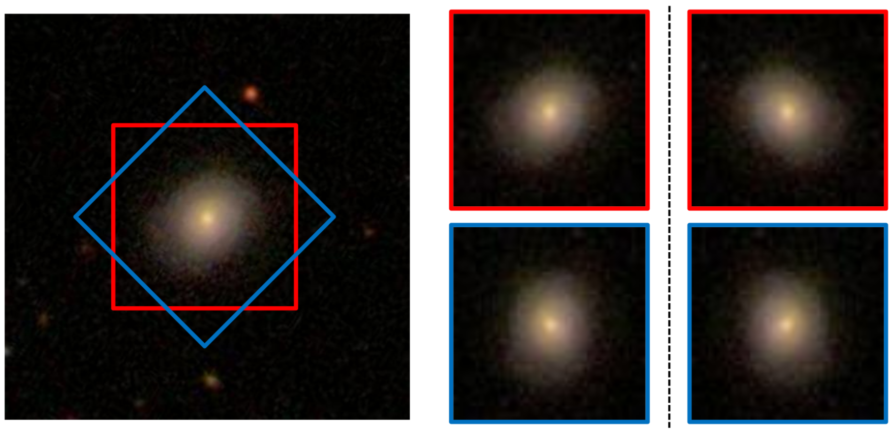

Figure 1: Four different *views* were extracted from each image: a regular view (red), a 45° rotated view (blue), and mirrored versions of both.

Petrosian radius. I then centered each image by shifting the estimated center pixel to (212, 212), and rescaled it so that its Petrosian radius would be equal to 160 pixels. The scale factor was limited to the range (1/1.5, 1.5), because there were some outliers. This rescaling and centering could also be collapsed into the affine transform doing downsampling and perturbation, so it did not slow things down at all.

2.2 Model architecture

2.2.1 Exploiting rotation invariance

I increased parameter sharing in the network by cutting the galaxy images into multiple parts that could be treated in the same fashion, i.e. processed by the same convolutional architecture. For this I exploited the rotation invariance of the images. As mentioned before, the images were cropped to 207x207 and downsampled by a factor of 3. This was done with two different orientations: a regular crop, as well as one that is rotated 45°. Both of these crops were also flipped horizontally, resulting in four 69x69 'views' of the image. This is visualised in Figure 1.

Each of the four views was again split into four partially overlapping 'parts' of size 45x45. Each part was rotated so that they are all aligned, with the galaxy in the bottom right corner. This is visualised in Figure 2. In total, 16 parts were extracted from the original image.

This results in 16 smaller 45x45 images which appear very similar. They can be expected to have the same topological structure due to rotation invariance, so they can be processed by the same convolutional architecture, which results in a 16x increase in parameter sharing, and thus less overfitting. At the top of the network, the features extracted from these 16 parts are concatenated and connected to one or more dense layers, so the information can be aggregated.

Due to the overlap of the parts, a lot of information is available about the center of the galaxy, because it is processed by the convnet in 16 different orientations. This is useful because a few important properties of the galaxies are expected to be in the center of the image (the presence of a bar or a bulge, for example). Reducing this overlap typically resulted in reduced performance. I chose not to make the parts fully overlap, because it would slow down training too much.

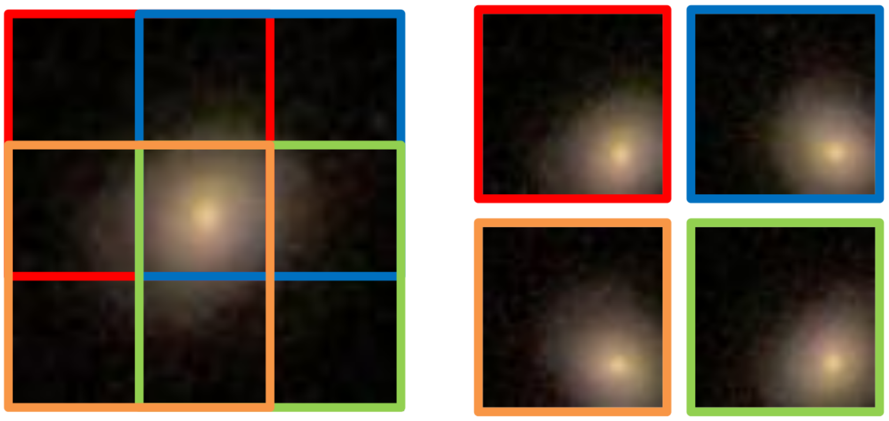

Figure 2: Each view was then split into four partially overlapping *parts*. Each part was rotated so that they are all aligned, with the galaxy in the bottom right corner. In total, 16 parts were extracted from the original image.

2.2.2 Incorporating output constraints

The 37 outputs to be predicted are weighted probabilities, adhering to a number of constraints. Incorporating these constraints into the model turned out to be beneficial. In essence, the answers to each question should form a categorical distribution. Additionally, they are scaled by the probability of the question being asked, i.e. the total probability of answers given that would lead to this question being asked.

Using a softmax output for each question, and then applying the scaling factors didn't make much of a difference. I believe this is because the softmax function has difficulty predicting hard zeros and ones, of which there were quite a few in the training data (its input would have to be very large in magnitude).

In the end I normalised the distribution for each question by adding a rectification nonlinearity in the top layer instead of the softmax functions, and then just using divisive normalisation. For example, if the raw, linear outputs of the top layer of the network for question one were z_1, z_2, z_3, then the actual output for question one, answer one was given by:

$$y_1 = \frac{\max(z_1, 0)}{\max(z_1, 0) + \max(z_2, 0) + \max(z_3, 0) + \epsilon}, \tag{1}$$

where ϵ is a very small constant that prevented division by zero errors. I set it to 1e-12. This approach allowed the network to predict hard zeros more easily.

2.2.3 Architecture of the best model

The best model I found is shown in Figure 3 in the form of a Krizhevsky-style diagram. All other models included in the final ensemble I submitted are slight variations of this model.

The input is presented to the model in the form of RGB coloured 45x45 image parts. The model has 7 layers: 4 convolutional layers and 3 dense layers. All convolutional layers include a ReLU nonlinearity (i.e. $f(x) = \max(x, 0)$). The first, second and fourth convolutional layers are followed by 2x2 max-pooling. The sizes of the layers, as well as the sizes of the filters, are indicated in the figure.

As mentioned before, the convolutional part of the network is applied to 16 different parts of the input image. The extracted features for all these parts are then aggregated and connected to the

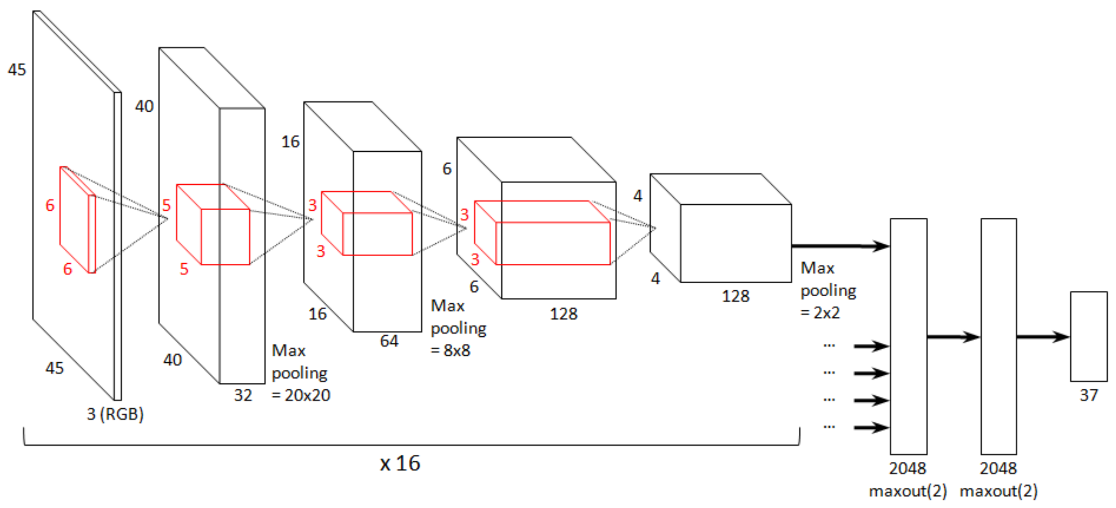

Figure 3: Krizhevsky-style diagram of the architecture of the best performing network.

dense part of the network. The dense part consists of two maxout layers with 2048 units [2], both of which take the maximum over pairs of linear filters (so 4096 linear filters in total). Using maxout here instead of regular dense layers with ReLUs helped to reduce overfitting a lot, compared to dense layers with 4096 linear filters. Using maxout in the convolutional part of the network as well proved too computationally intensive. Training this model took 67 hours on the available hardware (see Section 4).

2.2.4 Variants

Variants of the best model were included in the final ensemble I submitted, to increase variance (see Section 2.4). They include:

- a network with two dense layers instead of three (just one maxout layer);
- a network with one of the dense layers reduced in size and applied individually to each part (resulting in 16-way parameter sharing for this layer as well);
- a network with a different filter size configuration: 8/4/3/3 instead of 6/5/3/3 (from bottom to top);
- a network with centered and rescaled input images;
- a network with a ReLU dense layer instead of maxout;
- a network with 192 filters instead of 128 for the topmost convolutional layer;
- a network with 256 filters instead of 128 for the topmost convolutional layer;
- a network with norm constraint regularisation applied to the two maxout layers (as in Hinton et al. [3]);
- combinations of the above variations.

2.3 Training

2.3.1 Validation

For validation purposes, I split the training set in two parts. I used the first 90% for training, and the remainder for validation. Near the end of the competition I tried retraining a model on the entire training set, including the validation data I split off, but I noticed no increase in performance on the public leaderboard, so I left it at that.

2.3.2 Training algorithm

I trained the networks with stochastic gradient descent (SGD) and Nesterov momentum (with a fixed momentum constant of 0.9). I used a minibatch size of 16 examples. This meant that the effective minibatch size for the convolutional part was 256, because each training image is split into 16 parts (see Section 2.2.1).

The training data was processed into *chunks* of 10000 examples (one chunk was loaded into GPU memory at a time), so each chunk contained 625 minibatches. I trained the networks for 2500 chunks. I used a learning rate schedule with two discrete decreases. Initially it was set to 0.04. It was decreased tenfold to 0.004 after 1800 chunks, and again to 0.0004 after 2300 chunks. For the last few models I trained, the number of chunks was slightly decreased to 2100 to be able to finish training before the competition deadline.

For the first chunk, the divisive normalisation in the output layer described in Section 2.2.2 was disabled. This was necessary to ensure convergence (otherwise it would get stuck at the start sometimes).

2.3.3 Initialisation

Some fiddling with the parameter initialisation was required to get the network to train properly. Most of the layer weights were initialised from a Gaussian distribution with mean zero and a standard deviation of 0.01, with biases initialised to 0.1. For the topmost convolutional layer, I increased the standard deviation to 0.1. For the dense layers, I reduced it to 0.001 and the biases were initialised to 0.01. These modifications were necessary presumably because these layers are much smaller resp. bigger than the others.

2.3.4 Regularisation

Dropout was used in all three dense layers, with a dropout probability of 0.5. Near the very end of the competition I also experimented with norm constraint regularisation for the maxout layers. I chose the maximal norm for each layer based on a histogram of the norms of a network trained without norm constraint regularisation (I chose it so the tail of the histogram would be chopped off). I'm not entirely sure if this helped or not, since I was only able to do two runs with this setup.

2.4 Model averaging

2.4.1 Averaging across transformed images

For each individual model, I computed predictions for 60 affine transformations of the test set images: a combination of 10 rotations, spaced by 36°, 3 rescalings (with scale factors 1/1.2, 1 and 1.2) and flipping / no flipping. These were uniformly averaged. Even though the model architecture already incorporated a lot of invariances, this still helped quite a bit. Computing these averaged test set predictions for a single model took just over 4 hours.

2.4.2 Averaging across architectures

The averaged predictions for each model were then uniformly blended again, across a number of different models (variants of the model described in Section 2.2.4). I also experimented with a weighted blend, optimised on the validation set I split off, but this turned out not to make a significant difference. However, I did use the learned weights to identify sets of predictions that were not contributing at all, and I removed those from the uniform blend as well.

My final submission was a blend of predictions from 17 different models, each of which were themselves blended across 60 transformations of the input. So in the end, I blended 1020 predictions for each test set image.

For comparison: my best single model achieved a score of **0.07671** (public leaderboard) / **0.07693** (private leaderboard). After averaging across transformed images and architectures, I achieved a final score of **0.07467** (public) / **0.07492** (private).

3 Code description

This section describes the implementation. Please note that some of the published code was not used to generate the winning solution, and is not described here. I wanted to make it available anyway in case it proves useful for someone, and taking all the files apart to remove unused code would probably have introduced bugs. In addition, some methods are only used internally by other methods, and were left out of this description to avoid clutter.

3.1 Relevant paths

A number of paths are hardcoded, and the necessary files and directories need to be in place for some scripts to work. These paths are listed below. Section 5 describes step by step where to place the downloaded data files and how to generate the other necessary files.

- `data/`: contains all train and test data
- `data/raw/`: contains the data as it was provided on Kaggle
- `data/raw/training_solutions_rev1.csv`: the provided CSV file with training labels
- `data/raw/images_train_rev1/`: contains the train images in their original format (JPEG files)
- `data/raw/images_test_rev1/`: contains the test images in their original format (JPEG files)
- `/dev/shm/images_*_rev1/`: the JPEG files are copied to RAM (`/dev/shm` on Linux systems) for faster loading using `copy_data_to_shm.py`
- `data/solutions_train.npy`: train labels in a NumPy array, generated by `convert_training_labels_to_npy.py`
- `data/train_ids.npy`: ids of the train set in a NumPy array, generated by `create_train_ids_file.py`

- `data/test_ids.npy`: ids of the test set in a NumPy array, generated by `create_test_ids_file.py`
- `data/pysex_params_*.npy.gz`: these files contain parameters extracted from the image with sextractor. They are generated by `extract_pysex_params_extra.py` and `extract_pysex_params_gen2.py`.
- `analysis/final/*.pkl`: Python pickle files containing Python dictionaries with details about the training process, as well as final parameter values; created by running training scripts (`try_*.py`)
- `predictions/final/*.csv.gz`: test set prediction files in CSV format, created by running training scripts (`try_*.py`)
- `predictions/final/augmented/valid/*.npy.gz`: validation set prediction files in NumPy format, created by running augmented prediction scripts (`predict_augmented_npy_*.py`) (i.e. predictions averaged across transformed images)
- `predictions/final/augmented/test/*.npy.gz`: corresponding test set prediction files in NumPy format
- `predictions/final/blended/*.npy.gz`: test set prediction files in NumPy format containing predictions averaged across multiple architectures

3.2 Python modules

3.2.1 Data loading: `load_data.py`

load_data.train_ids, .test_ids: NumPy arrays containing a list of image IDs for the train and test sets respectively.

load_data.load_image: given an image ID, load a JPEG image from the dataset into a NumPy array and normalise it so all values are in the interval [0, 1].

Inputs:

- img_id: the ID of the image.
- subset='train': the subset the image is part of ('train' or 'test').
- normalise=True: whether to normalise the values to [0, 1].
- from_ram=False: load the image from `/dev/shm` instead of from `data/raw/`.

Outputs:

- img: a NumPy array with shape (424, 424, 3) containing the image data.

load_data.load_gz, .save_gz: extended versions of numpy.load and numpy.save that work with gzipped files.

load_data.buffered_gen_mp: generator that runs a slow source generator in a separate process. This is useful for doing data augmentation in parallel with training.

Inputs:

- source_gen: the generator to run in a separate process.
- buffer_size=2: the number of items to pre-generate and buffer.

- sleep_time=1: number of seconds to sleep between attempts to get a new item from the buffer.

3.2.2 Realtime augmentation: `realtime_augmentation.py`

This module is usually imported as `ra` for convenience.

realtime_augmentation.build_ds_transform: creates a `skimage.transform.AffineTransform` object that represents a downsampling and cropping operation. Note that only the `ds_factor` and `target_size` arguments were used in most cases, the others were typically left to their default values.

Inputs:

- ds_factor=1.0: the downsampling factor.
- orig_size=(424, 424): tuple representing the original image size in pixels.
- target_size=(53, 53): tuple representing the target size of the downsampled image in pixels.
- do_shift=True: include a translation so the center part of the image is cropped.
- subpixel_shift=False: if the downsampling factor is integer, enabling this will ensure that the pixel grid of the subsampled image is aligned with that of the original image.

Outputs:

- tform_ds: a `skimage.transform.AffineTransform` object representing the downsampling and cropping operation.

realtime_augmentation.build_augmentation_transform: creates a `skimage.transform.AffineTransform` object that represents a data augmentation operation.

zoom=1.0, rotation=0, shear=0, translation=(0, 0)

Inputs:

- zoom=1.0: zoom factor.
- rotation=0: rotation angle in degrees.
- shear=0: shear angle in degrees.
- translation=(0,0): tuple representing the shift in the x and y directions in pixels.

Outputs:

- tform_augment: a `skimage.transform.AffineTransform` object representing the data augmentation operation.

realtime_augmentation.random_perturbation_transform: creates a `skimage.transform.AffineTransform` object that representing a random dat augmentation operation.

Inputs:

- zoom_range: range from which the zoom factor is sampled (log-uniform).
- rotation_range: range (in degrees) from which the rotation angle is sampled (uniform).
- shear_range: range (in degrees) from which the shear angle is sampled (uniform).

- translation_range: range (in pixels) from which the shift in both the x and y directions is sampled.
- do_flip=False: if True, a flip is incorporated into the transform half of the time (bernoulli).

Outputs:

- tform_augment: a `skimage.transform.AffineTransform` object representing the random data augmentation operation.

realtime_augmentation.realtime_augmented_data_gen: generator that yields chunks of data to be used for training. Each chunk consists of a bunch of example images, which have been downsampled, cropped and augmented. The preprocessing happens in separate processes to increase parallelism.

Inputs:

- num_chunks=None: the number of chunks to generate. If set to None, it never stops generating chunks, but this may prevent scripts from exiting properly.
- chunk_size=CHUNK_SIZE: the number of examples to include in each chunk. For all experiments this was set to 10000.
- augmentation_params=default_augmentation_params: a dictionary containing the ranges from which to sample the augmentation transform parameters.
- ds_transforms=ds_transforms_default: a list of cropping and downsampling transforms to apply to the source images.
- target_sizes=None: a list of tuples representing the target sizes of each `ds_transform`. If set to None, this defaults to (53, 53) for each `ds_transform`.
- processor_class=LoadAndProcess: a class encapsulating how the data should be loaded and preprocessed. `LoadAndProcess` gives the default behaviour. Alternatives are `LoadAndProcessPysexCentering`, `LoadAndProcessPysexCenteringRescaling` and `LoadAndProcessPysexGen1CenteringRescaling`. All of these incorporate centering and rescaling based on the parameters extracted with `sextractor`. Because the preprocessing happens in separate processes, these cannot be simple functions, they must be encapsulated in a class instead.

Outputs

- target_arrays: a list of arrays containing the preprocessed image data (one array for each `ds_transform`), and the corresponding training labels.
- chunk_size: the number of examples in the chunk. This is always equal to the specified `chunk_size` for this generator, it is included mainly for compatibilty reasons.

realtime_augmentation.post_augment_brightness_gen: generator that wraps `realtime_augmented_data_gen` and does colour augmentation as described in Section 2.1.3.

Inputs:

- data_gen: a data generator, typically an instance of `realtime_augmented_data_gen`.
- std=0.5: the standard deviation of the added 'colour noise'.

realtime_augmentation.realtime_fixed_augmented_data_gen: this is an alternate version of `realtime_augmented_data_gen` which does not perturb images randomly, but according to a fixed set of transformations. This can be used for validation and generating predictions.

Inputs:

- selected_indices: indices of the train/test examples to use.
- subset: which subset to use, i.e. 'train' or 'test'.
- ds_transforms=ds_transforms_default: a list of cropping and downsampling transforms to apply to the source images.
- augmentation_transforms=[tform_identity]: a list of augmentation transforms to apply to the source images. By default, only the identity transform is applied.
- chunk_size=CHUNK_SIZE: the number of examples to include in each chunk.
- target_sizes=None: a list of tuples representing the target sizes of each `ds_transform`. If set to None, this defaults to (53, 53) for each `ds_transform`.
- processor_class=LoadAndProcessFixed: a class encapsulating how the data should be loaded and preprocessed. `LoadAndProcessFixed` gives the default behaviour. Alternatives are `LoadAndProcessFixedPysexCentering`, `LoadAndProcessFixedPysexCenteringRescaling` and `LoadAndProcessFixedPysexGen1CenteringRescaling`. All of these incorporate centering and rescaling based on the parameters extracted with `sextractor`. Because the preprocessing happens in separate processes, these cannot be simple functions, they must be encapsulated in a class instead.

Outputs

- target_arrays: a list of arrays containing the preprocessed image data (one array for each `ds_transform`).
- current_chunk_size: the number of examples in the chunk. This is not always equal to `chunk_size` in this case. The returned arrays always have size `chunk_size`, and are padded with zeros if the number of examples is lower than that. This is to facilitate loading the data into GPU memory at a fixed location (Theano shared variables are not reallocated if their size remains the same when they are updated).

3.2.3 Layers: `layers.py`, `cc_layers.py`, `custom.py`

The networks are built up from a number of different layers, representing by instances of layer classes. All layer classes have `output` and `get_output_shape` methods, except for output layers (placed at the top of a network), which have `error` and `predictions` methods instead. When calling the `output`, `predictions` and `error` methods, the keyword argument `dropout_active` specifies whether dropout should be used. This allows for it to be turned on and off for training and evaluation respectively.

The `layers` module contains default layer implementations, using Theano's own operators. It also includes a bunch of layer classes that are not particularly useful for this problem. This is one of those files that I tend to copy from project to project, that's why there's a lot of other stuff in it as well. Some of the code in this file was contributed by my colleague Aäron van den Oord.

The `cc_layers` module contains all layer classes needed to work with the cuda-convnet convolution and max-pooling implementations. The `custom` module contains some layer classes that are specific to the Galaxy Challenge, i.e. the output layers incorporating the decision tree constraints (see Section 2.2.2).

The `layers` module also includes some utility functions to work with layer classes. `all_parameters` returns a list containing all shared variables representing parameters of the network. `all_bias_parameters` is similar, but includes only the biases. The various `gen_updates_*` methods return a list of tuples representing parameter updates, which can be passed directly to `theano.function`.

`get_param_values` and `set_param_values` are used to serialise and restore the parameter values of a network.

layers.Input2DLayer: provides 2D input data in the form of a 4-tensor with shape `(minibatch_size,num_input_channels,width,height)`. Its property `input_var` symbolically represents the input.

layers.DenseLayer: a dense (fully connected) neural network layer, which flattens its input to a matrix with shape `(minibatch_size,num_features)` if necessary.

layers.FlattenLayer: a layer that simply flattens its input to a matrix with shape `(minibatch_size,num_features)`. It does not have any weights itself.

layers.MultiRotSliceLayer: performs the views/parts slicing as described in Section 2.2.1. It can have multiple input layers, and it slices the input images into parts and stacks them into a larger minibatch.

layers.MultiRotMergeLayer: performs the inverse operation of `MultiRotSliceLayer` and flattens the result into a matrix with shape `(minibatch_size,num_features)`, so `DenseLayers` can be stacked on top.

layers.FeatureMaxPoolingLayer: performs max-pooling across the feature dimension. This is used to implement maxout [2].

cc_layers.ShuffleBC01ToC01BLayer: shuffles the 4-tensor with shape `(minibatch_size, num_input_channels,width,height)` so it has shape `(num_input_channels,width,height, minibatch_size)` instead. This is necessary to interface with the cuda-convnet convolution implementation, which expects the trailing dimension to be the minibatch dimension. Theano on the other hand encourages the convention where leading dimension is the minibatch dimension. Luckily this shuffling does not incur a big performance penalty.

cc_layers.ShuffleC01BToBC01Layer: performs the inverse operation of `ShuffleBC01ToC01BLayer`.

cc_layers.CudaConvnetConv2DLayer: a 2D convolutional layer, using the cuda-convnet implementation.

cc_layers.CudaConvnetPooling2DLayer: a 2D max-pooling layer, using the cuda-convnet implementation.

custom.OptimisedDivGalaxyOutputLayer: an output layer for the Galaxy Challenge, which incorporates the decision tree constraints with rectification and divisive normalisation as described in Section 2.2.2. This is a version of `DivisiveGalaxyOutputLayer`, optimised for performance.

3.3 Python scripts

3.3.1 Training scripts: `try_*.py`

All scripts starting with `try_` are training scripts. A separate script is provided for each of the 17 models used in the final ensemble that I submitted. Although these scripts contain mostly the same code, I opted to create a separate script for each training instance, to be able to keep track of the parameters and code that generated each set of predictions. This is not a particularly modular style of programming, but since the code was written in the context of a research competition, modularity was not very high on the list of requirements.

Each training scripts generates test set predictions, as well as an 'analysis' pickle file which contains the final parameter values and some information about the training process.

3.3.2 Prediction scripts: `predict_augmented_npy_*.py`

All scripts starting with `predict_augmented_npy_` are for generating predictions which are averaged across various transformations of the input. Here too there is a separate script for each of the 17 models used in the final ensemble. The prediction scripts read the network parameters from the corresponding analysis pickle files, so they will not work unless the corresponding training scripts have run to completion.

3.3.3 Model blending: `ensembled_predictions_npy.py`

The model blending script generates three sets of blended predictions, from all prediction files it can find in `predictions/final/augmented/test/` (i.e. generated by the `predict_augmented_npy_*.py` scripts). A uniform blend, a weighted linear blend with weights fitted on the validation set, and another weighted linear blend where the weights are fitted to each question individually. The uniform blend was the best in the end (the weighting apparently lead to overfitting on the validation set in both cases), but the weights from the weighted linear blend were used to identify and remove sets of predictions that were not contributing (and making the uniform blend perform worse).

3.4 Other utilities

copy_data_to_shm.py: this script copies the train and test images to `/dev/shm/`, a RAM drive that is available on modern Linux systems. The `load_data.load_images` method relies on this if its `from_ram` parameter is set to True. Most calls to this method from the `realtime_augmentation` module have this parameter set to True, which means all training scripts rely on the image data being available in `/dev/shm/` by default. If you do not wish to copy the images to RAM, you will have to modify all the calls to `load_data.load_images` in `realtime_augmentation.py` and set the `from_ram` parameter to False.

convert_training_labels_to_npy.py: converts the CSV file with the training labels to a NumPy file for convenience.

create_train_ids_file.py, create_test_ids_file.py: creates two NumPy files containing a list of train and test IDs respectively. These files are useful to ensure that the data is always processed in the same order and to avoid mismatching images and labels.

extract_pysex_params_*.py: these scripts use sextractor and pysex to extract some parameters from the images. These parameters can be used to recenter and rescale the images.

create_submission_from_npy.py: the `predict_augmented_npy_*.py` and `ensembled_predictions_npy.py` scripts generate predictions in `.npy.gz` format. To be able to submit predictions for the competition, these need to be converted to `.csv.gz` format. This script can be used for that purpose. It takes a path to a `.npy.gz` file as its sole command line argument, and generates a corresponding `.csv.gz` file in the same location.

check_label_constraints.py: this script can be used to verify that the provided training labels satisfy the decision tree constraints. It was written because the original training data provided by the contest organisers had some errors, which resulted in some of the constraints being violated.

4 Dependencies

I used **Python 2.7.3**, with the following libraries and modules:

- NumPy 1.6.1
- SciPy 0.10.1
- Theano 0.6
- scikit-image 0.9.3
- pylearn2 (commit f032030bfb047a5616c72151b7dd98fc3b78bb5a[5])
- pandas 0.7.3 (optional - only to convert the training labels to a NumPy array)

For parameter extraction with sextractor, I also used the following (all of these are optional):

- sextractor 2.8.6 (2009-04-09)[6]
- pysex 0.1.0
- pyfits 3.0.6 (pysex dependency)
- asciidata 1.1.1 (pysex dependency)

Note that the training scripts use the cuda-convnet wrappers from pylearn2, which means that **a recent NVIDIA GPU is required to be able to run them**. They will not run on the CPU, contrary to most Theano code. The scripts should work with cards from the 500 series or upwards, with at least 2GB of memory. I used GTX 680 cards with 2 or 4GB RAM[7], and version 5.0 of the CUDA toolkit. For the data augmentation to work properly, a quadcore processor and 16GB of CPU RAM is recommended. You may reduce memory usage by reducing the chunk size (10000 by default), but of course you will need to increase the number of chunks accordingly.

I ran all the code on Linux. It should work on any Linux distribution. If you want to run the code on another OS, some modifications might be required.

[5]`https://github.com/lisa-lab/pylearn2/commit/f032030bfb047a5616c72151b7dd98fc3b78bb5a`

[6]`https://www.astromatic.net/software/sextractor`

[7]If you plan to invest in a GPU, I would recommend getting either a 500 series or a 700 series card. The 600 series cards are known to be less suitable for CUDA applications.

5 Generating the solution

5.1 Install the dependencies

Instructions for installing Theano and getting it to run on the GPU can be found at `http://deeplearning.net/software/theano/install.html`. It should be possible to install NumPy, SciPy, scikit-image and pandas using `pip` or `easy_install`. To install pylearn2, simply run `git clone git://github.com/lisa-lab/pylearn2.git` and add the resulting directory to your `PYTHONPATH`.

The optional dependencies don't have to be installed to reproduce the winning solution: the generated data files are already provided, so they don't have to be regenerated (but of course you can if you want to). If you want to install them, please refer to their respective documentation.

5.2 Download the code

Run `git clone git://github.com/benanne/kaggle-galaxies.git` to download the code, as well as a bunch of data files (extracted sextractor parameters, IDs files, training labels in NumPy format, ...). I decided to include these since generating them is a bit tedious and requires extra dependencies. It's about 20MB in total, so depending on your connection speed it could take a minute. Cloning the repository should also create the directory structure described in Section 3.1.

5.3 Download the training data

Download the data files from `http://www.kaggle.com/c/galaxy-zoo-the-galaxy-challenge/data`. Place and extract the files in the following locations:

- `data/raw/training_solutions_rev1.csv`
- `data/raw/images_train_rev1/*.jpg`
- `data/raw/images_test_rev1/*.jpg`

5.4 Create data files

This step may be skipped. The necessary data files have been included in the git repository. Nevertheless, if you wish to regenerate them (or make changes to how they are generated), here's how to do it.

- create `data/train_ids.npy` by running `python create_train_ids_file.py`.
- create `data/test_ids.npy` by running `python create_test_ids_file.py`.
- create `data/solutions_train.npy` by running `python convert_training_labels_to_npy.py`.
- create `data/pysex_params_extra_*.npy.gz` by running `python extract_pysex_params_extra.py`.
- create `data/pysex_params_gen2_*.npy.gz` by running `python extract_pysex_params_gen2.py`.

5.5 Train the networks

Run `python try_convnet_cc_multirotflip_3x69r45_maxout2048_extradense.py` to train the best single model. On a GeForce GTX 680, this took about 67 hours to run to completion. The prediction file generated by this script, `predictions/final/try_convnet_cc_multirotflip_3x69r45_maxout2048_extradense.csv.gz`, **should get you a score that's good enough to land in the #1 position (without any model averaging)**. You can similarly run the other `try_*.py` scripts to train the other models I used in the winning ensemble.

If you have more than 2GB of GPU memory, I recommend disabling Theano's garbage collector with `allow_gc=False` in your `.theanorc` file or in the `THEANO_FLAGS` environment variable, for a nice speedup. Please refer to `http://deeplearning.net/software/theano/tutorial/using_gpu.html#tips-for-improving-performance-on-gpu` for more information on how to get the most out Theano's GPU support.

5.6 Generate augmented predictions

To generate predictions which are averaged across multiple transformations of the input, run `python predict_augmented_npy_maxout2048_extradense.py`. This takes just over 4 hours on a GeForce GTX 680, and will create two files `predictions/final/augmented/valid/try_convnet_cc_multirotflip_3x69r45_maxout2048_extradense.npy.gz` and `predictions/final/augmented/test/try_convnet_cc_multirotflip_3x69r45_maxout2048_extradense.npy.gz`. You can similarly run the corresponding `predict_augmented_npy_*.py` files for the other models you trained.

5.7 Blend augmented predictions

Run `python ensemble_predictions_npy.py` to generate blended prediction files from all the models for which you generated augmented predictions. The script checks which files are present in `predictions/final/augmented/test/` and uses this to determine the models for which predictions are available. It will create three files:

- `predictions/final/blended/blended_predictions_uniform.npy.gz`: uniform blend.
- `predictions/final/blended/blended_predictions.npy.gz`: weighted linear blend.
- `predictions/final/blended/blended_predictions_separate.npy.gz`: weighted linear blend, with separate weights for each question.

5.8 Convert prediction file to CSV

Finally, in order to prepare the predictions for submission, the prediction file needs to be converted from `.npy.gz` format to `.csv.gz`. Run the following to do so (or similarly for any other prediction file in `.npy.gz` format): `python create_submission_from_npy.py predictions/final/blended/blended_predictions_uniform.npy.gz`

5.9 Submit predictions

Submit the file `predictions/final/blended/blended_predictions_uniform.csv.gz` at `http://www.kaggle.com/c/galaxy-zoo-the-galaxy-challenge/submit` to get it scored. Note that the process of generating this file involves considerable randomness: the weights of the networks are initialised randomly, the training data for each chunk is randomly selected, ... so I cannot guarantee that you will achieve the same score as I did. I did not use fixed random seeds. This might not have made much of a difference though, since different GPUs and CUDA toolkit versions will also introduce different rounding errors.

6 Additional comments and observations

Below are a few things that I tried but ended up not using, either because they didn't help my score, or because they slowed down training too much.

- Adding Gaussian noise to the input images during training to reduce overfitting. This didn't help.
- changing the gamma value of the input images (instead of brightness) didn't help either.
- Extracting crops from the input images at different scales, and training a multiscale convnet on them. It turned out that only the part of the network for the most detailed scale was actually learning anything. The other parts received no gradient and weren't really learning.
- Overlapping pooling. This seemed to help a little bit, but it slowed things down too much.
- Downsampling the input images less (1.5x instead of 3x) and using a strided convolution in the first layer (with stride 2). This did not improve results, and dramatically increased memory usage.
- Adding shearing to the data augmentation step. This didn't help, but it didn't hurt performance either. I assumed that it would hurt performance because question 7 pertains to the ellipticity of the galaxy (shearing would of course change this), but this didn't seem to be the case.
- Stochastic pooling [6] was too slow to use in all convolutional layers. I tried using it only in the topmost convolutional layer, but it didn't make much of a difference that way. If I had a faster implementation, I believe this might have helped to reduce overfitting.
- alternative training algorithms: adagrad [1], rmsprop and adadelta [5]. It might be down to how much time I spent tuning them, but I could not get them to work better than stochastic gradient descent.
- Near the end of the competition I also toyed with a polar coordinate representation of the images. I suppose this could work well because rotations turn into translations in polar space, so the convnet's inherent translation invariance would amount to rotation invariance in the original input space. Unfortunately I didn't have enough time left to properly explore this approach, so I decided to focus on my initial approach instead.

During the contest, I frequently referred to Krizhevsky et al.'s seminal 2012 paper on ImageNet classification [4] for guidance. If you are looking to use convolutional neural networks for a practical application, reading this paper is thoroughly recommended.

References

[1] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[2] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[3] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. Technical report, University of Toronto, 2012.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, 2012.

[5] Matthew D Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[6] Matthew D Zeiler and Rob Fergus. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*, 2013.