National Data Science Bowl model documentation

≋Deep Sea≋

Aäron van den Oord, Iryna Korshunova, Jeroen Burms, Jonas Degrave, Lionel Pigou, Pieter Buteneers, Sander Dieleman

1. Personal details

Team Name: ≋Deep Sea≋
Names: Aäron van den Oord, Iryna Korshunova, Jeroen Burms, Jonas Degrave, Lionel Pigou, Pieter Buteneers, Sander Dieleman
Location: Ghent, Belgium
Emails: aaron.vdo@gmail.com, irene.korshunova@gmail.com, jer.burms@gmail.com, Jonas.Degrave@UGent.be, lionelpigou@gmail.com, pieter.buteneers@gmail.com, sanderdieleman@gmail.com
Competition: National Data Science Bowl

2. Summary

Our solution is based around convolutional neural networks. We used fairly large models, and as a result our main focus during the competition was combatting overfitting. To do this we used data augmentation, dropout, weight decay, pseudo-labeling and modified network architectures to exploit rotation invariance, among other things. All networks were trained with stochastic gradient descent with Nesterov momentum and GPU acceleration was used to speed up experiments. Our final submission was an ensemble consisting of over 40 networks. No external data sources were used.

3. Models and training

3.1. Pre-processing and data augmentation

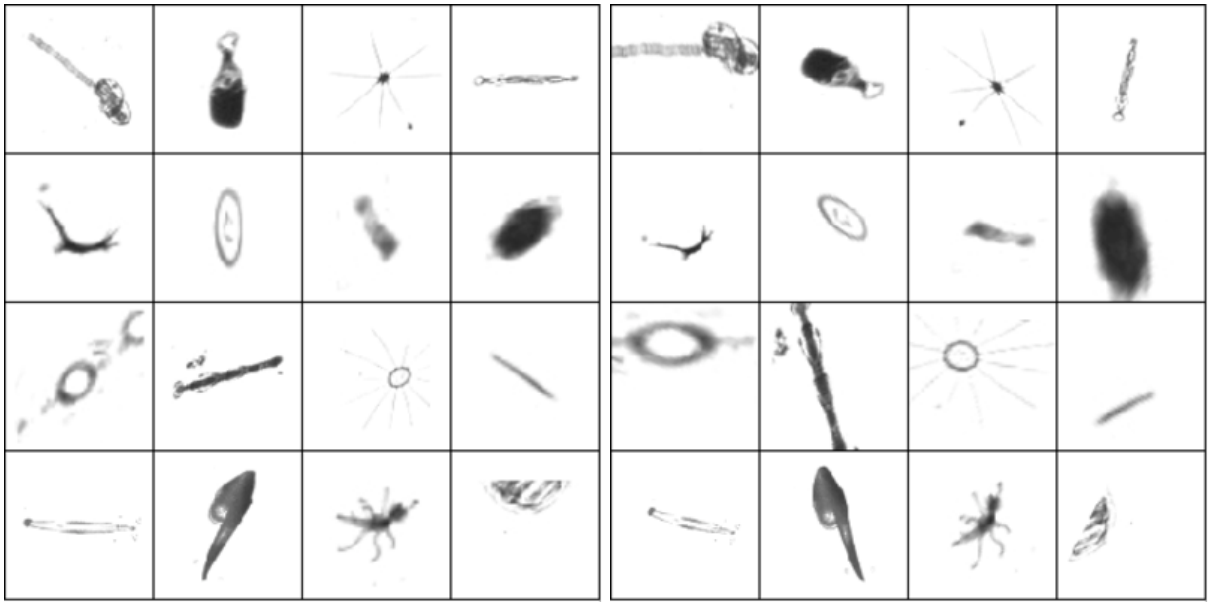
We performed very little pre-processing, other than rescaling the images in various ways and then performing global zero mean unit variance (ZMUV) normalization, to improve the stability of training and increase the convergence speed.

Rescaling the images was necessary because they vary in size a lot: the smallest ones are less than 40 by 40 pixels, whereas the largest ones are up to 400 by 400 pixels. We experimented with various (combinations of) rescaling strategies. For most networks, we simply rescaled the largest side of each image to a fixed length.

We augmented the data to artificially increase the size of the dataset. We used various affine transforms, and gradually increased the intensity of the augmentation as our models started to overfit more. We ended up with some pretty extreme augmentation parameters:

- rotation: random with angle between 0° and 360° (uniform)
- translation: random with shift between -10 and 10 pixels (uniform)
- rescaling: random with scale factor between 1/1.6 and 1.6 (log-uniform)
- flipping: yes or no (bernoulli)
- shearing: random with angle between -20° and 20° (uniform)
- stretching: random with stretch factor between 1/1.3 and 1.3 (log-uniform)

We augmented the data on-demand during training (realtime augmentation), which allowed us to combine the image rescaling and augmentation into a single affine transform. The augmentation
was all done on the CPU while the GPU was training on the previous chunk of data.



Pre-processed images (left) and augmented versions of the same images (right).

3.2. Network architecture

Most of our convnet architectures were strongly inspired by OxfordNet: they consist of lots of convolutional layers with 3x3 filters. We used 'same' convolutions (i.e. the output feature maps are the same size as the input feature maps) and overlapping pooling with window size 3 and stride 2. Most models had untied biases in the convolutional layers (i.e. separate bias parameters for each spatial location on each feature map).

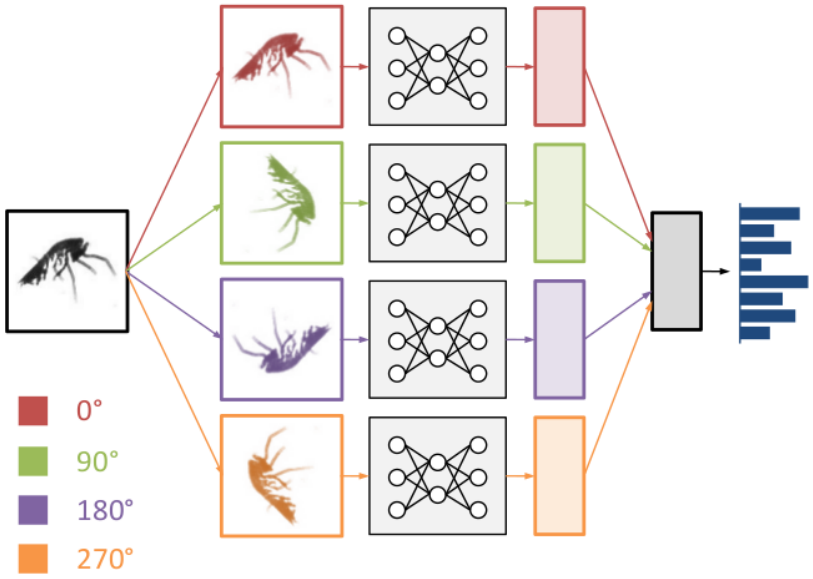
We started with a fairly shallow models by modern standards (~ 6 layers) and gradually added more layers when we noticed it improved performance (it usually did). Near the end of the

competition, we were training models with up to 16 layers. The challenge, as always, was balancing improved performance with increased overfitting.

3.2.1. Cyclic pooling

We exploited the rotational symmetry of the images to share parameters in the network: We applied the same stack of convolutional layers to several rotated versions of the same input image, and then pooled across the resulting feature representations to get rotation invariance. This allowed the network to use the same feature extraction pipeline to "look at" the input from different angles.

In practice, this was implemented as follows: the images in a minibatch occur 4 times, in 4 different orientations. They are processed by the network in parallel, and at the top, the feature maps are pooled together. We decided to call this cyclic pooling, after cyclic groups.



Schematic representation of a convnet with cyclic pooling

The nice thing about 4-way cyclic pooling is that it can be implemented very efficiently: the images are rotated by 0, 90, 180 and 270 degrees. All of these rotations can be achieved simply by transposing and flipping image axes. That means no interpolation is required.

Cyclic pooling also allowed us to reduce the batch size by a factor of 4: instead of having batches of 128 images, each batch now contained 32 images and was then turned into a batch with an effective size of 128 again inside the network, by stacking the original batch in 4 orientations. After the pooling step, the batch size was reduced to 32 again.

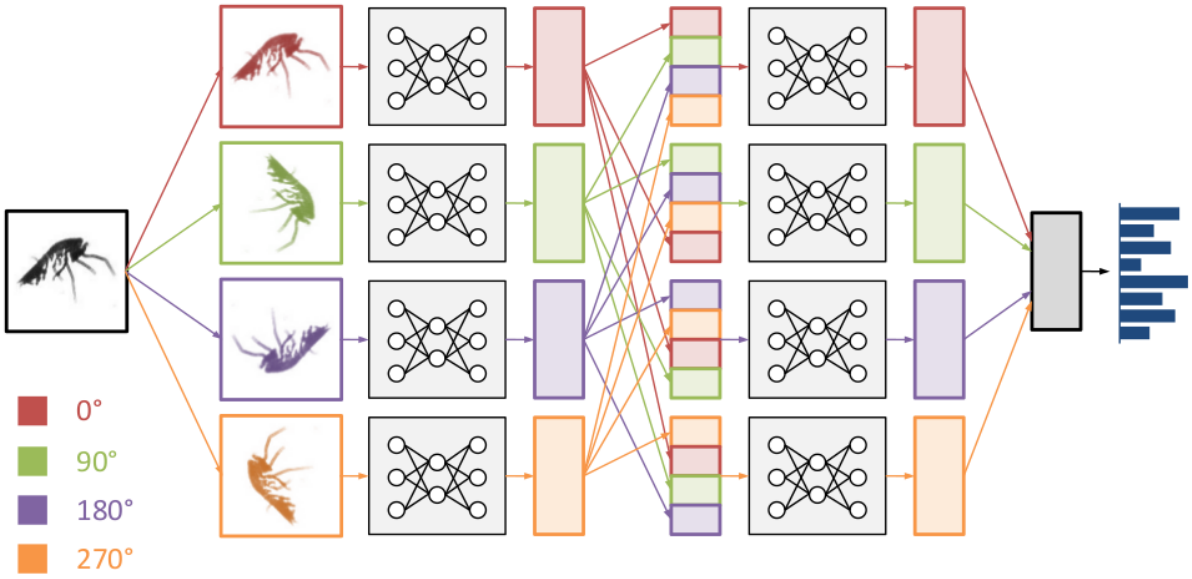
We tried several pooling functions over the course of the competition. It turned out that root-mean-square pooling gave much better results than mean pooling or max pooling.

One of our models pooled over 8 rotations, spaced apart 45 degrees. This required generating the input images at two angles (0 and 45 degrees).

3.2.2. 'Rolling' feature maps

A cyclic pooling convnet extracts features from input images in four different orientations. An alternative interpretation is that its filters are applied to the input images in four different orientations. That means we can combine the stacks of feature maps from the different orientations into one big stack, and then learn the next layer of features on this combined input. As a result, the network then appears to have 4 times more filters than it actually has.

This is cheap to do, since the feature maps are already being computed anyway. We just have to combine them together in the right order and orientation. We named the operation that combines feature maps from different orientations a roll.



Schematic representation of a roll operation inside a convnet with cyclic pooling.

Roll operations can be inserted after dense layers or after convolutional layers. In the latter case, care has to be taken to rotate the feature maps appropriately, so that they are all aligned.

In most of the models we evaluated, we only inserted convolutional roll operations after the pooling layers, because this reduced the size of the feature maps that needed to be copied and stacked together.

3.2.3. Nonlinearities

We experimented with various variants of rectified linear units (ReLUs), as well as maxout units (only in the dense layers). We had great success with (very) leaky ReLUs. Instead of taking the maximum of the input and zero, *y = max(x, 0)*, leaky ReLUs take the maximum of the input and a scaled version of the input, *y = max(x, a*x)*. Here, *a* is a tunable scale parameter.

For fairly deep networks (10+ layers), we found that varying this parameter between 0 and 1/2 did not really affect the predictive performance. However, larger values in this range significantly reduced the level of overfitting. This in turn allowed us to scale up our models further. We eventually settled on *a = 1/3*.

3.2.4. Spatial pooling

Most of our models had 4 spatial max-pooling layers. We started out with the traditional approach of 2x2 max-pooling, but eventually switched to 3x3 max-pooling with stride 2 (which we'll refer to as 3x3s2), mainly because it allowed us to use a larger input size while keeping the same feature map size at the topmost convolutional layer, and without increasing the computational cost significantly.

As an example, a network with 80x80 input and 4 2x2 pooling stages will have feature maps of size 5x5 at the topmost convolutional layer. If we use 3x3s2 pooling instead, we can feed 95x95 input and get feature maps with the same 5x5 shape. This improved performance and only slowed down training slightly.

3.2.5. Multiscale architectures

As mentioned before, the images vary widely in size, so we usually rescaled them using the largest dimension of the image as a size estimate. This is clearly suboptimal, because some species of plankton are larger than others. Size carries valuable information.

To allow the network to learn this, we experimented with combinations of different rescaling strategies within the same network, by combining multiple networks with different rescaled inputs together into 'multiscale' networks. What worked best was to combine a network with inputs rescaled based on image size, and a smaller network with inputs rescaled by a fixed factor.

3.2.6. Additional image features

We experimented with training small neural nets on extracted image features to 'correct' the predictions of our convnets. We referred to this as 'late fusing' because the feature network and the convnet were joined only at the output layer (before the softmax).

We thought this could be useful, because the features can be extracted from the raw (i.e. non-rescaled) images, so this procedure could provide additional information that is missed by the convnets. We ended up using the following types of features:

- Image size in pixels
- Size and shape estimates based on image moments
- Haralick texture features

The features were fed to a neural net with two dense layers of 80 units. The final layer of the model was fused with previously generated predictions of our best convnet-based models. Using this approach, we didn't have to retrain the convnets nor did we have to regenerate predictions.

To deal with variance due to the random weight initialization, we trained each feature network 10 times and blended the copies with uniform weights. This resulted in a consistent validation loss decrease of 0.01 (or 1.81%) on average, which was quite significant near the end of the competition.

Interestingly, late fusion with image size and features based on image moments seems to help just as much for multiscale models as for regular convnets. This is a bit counterintuitive: we expected both approaches to help because they could extract information about the size of the creatures, so the obtained performance improvements would overlap.

3.3. Training

3.3.1. Validation

We split off 10% of the labeled data as a validation set using stratified sampling. Due to the small size of this set, our validation estimates were relatively noisy and we periodically validated some models on the leaderboard as well.

3.3.2. Training algorithm

We trained all of our models with stochastic gradient descent (SGD) with Nesterov momentum. We set the momentum parameter to 0.9 and did not tune it further. Most models took between 24 and 48 hours to train to convergence.

We trained most of the models with about 215000 gradient steps and eventually settled on a discrete learning rate schedule with two 10-fold decreases (following Krizhevsky et al.), after about 180000 and 205000 gradient steps respectively. For most models we used an initial learning rate of 0.003.

3.3.3. Initialization

We used a variant of the orthogonal weight initialization strategy proposed by Saxe et al. everywhere. This allowed us to add as many layers as we wanted without running into any convergence problems.

3.3.4. Regularization

For most models, we used dropout in the fully connected layers of the network, with a dropout probability of 0.5. We discovered near the end of the competition that it was useful to have a

small amount of weight decay to stabilize training of larger models (so not just for its regularizing effect). Models with large fully connected layers and without weight decay would often diverge unless the learning rate was decreased considerably, which slowed things down too much.

3.3.5. Pseudo-labeling

We exploited the information in the test setby a combination of pseudo-labeling and knowledge distillation (Hinton et al.). The initial results from models trained with pseudo-labeling were significantly better than we anticipated, so we ended up investigating this approach quite thoroughly.

Pseudo-labeling entails adding test data to the training set to create a much larger dataset. The labels of the test datapoints (so called pseudo-labels) are based on predictions from a previously trained model or an ensemble of models. This mostly had a regularizing effect, which allowed us to train bigger networks.

We experimented both with hard targets (one-hot coded) and soft targets (predicted probabilities), but quickly settled on soft targets as these gave much better results.

Another important detail is the balance between original data and pseudo-labeled data in the resulting dataset. In most of our experiments 33% of the minibatch was sampled from the pseudo-labeled dataset and 67% from the real training set.

It is also possible to use more pseudo-labeled data points (e.g. 67%). In this case the model is regularized a lot more, but the results will be more similar to the pseudo-labels. As mentioned before, this allowed us to train bigger networks, but in fact this is necessary to make pseudo-labeling work well. When using 67% of the pseudo-labeled dataset we even had to reduce or disable dropout, or the models would underfit.

Our pseudo-labeling approach differs from knowledge distillation in the sense that we use the test set instead of the training set to transfer knowledge between models. Another notable difference is that knowledge distillation is mainly intended for training smaller and faster networks that work nearly as well as bigger models, whereas we used it to train bigger models that perform better than the original model(s).

We think pseudo-labeling helped to improve our results because of the large test set and the combination of data-augmentation and test-time augmentation (see below). When pseudo-labeled test data is added to the training set, the network is optimized (or constrained) to generate predictions similar to the pseudo-labels for all possible variations and transformations of the data resulting from augmentation. This makes the network more invariant to these transformations, and forces the network to make more meaningful predictions.

We saw the biggest gains in the beginning (up to 0.015 improvement on the leaderboard), but even in the end we were able to improve on very large ensembles of (bagged) models (between 0.003 - 0.009).

3.4. Model averaging

We combined several forms of model averaging in our final submissions.

3.4.1. Test-time augmentation (TTA)

For each individual model, we computed predictions across various augmented versions of the input images and averaged them. This improved performance by quite a large margin. When we started doing this, our leaderboard score dropped from 0.7875 to 0.7081. We used the acronym TTA to refer to this operation.

Initially, we used a manually created set of affine transformations which were applied to each image to augment it. This worked better than using a set of transformations with randomly sampled parameters. After a while, we looked for better ways to tile the augmentation parameter space, and settled on a quasi-random set of 70 transformations, using slightly more modest augmentation parameter ranges than those used for training.

Computing model predictions for the test set using TTA could take up to 12 hours, depending on the model.

3.4.2. Combining different models

In total we trained over 300 models, so we had to select how many and which models to use in the final blend. For this, we used cross-validation on our validation set. On each fold, we optimized the weights of all models to minimize the loss of the ensemble on the training part.

We regularly created new ensembles from a different number of top-weighted models, which we further evaluated on the testing part. In the end, this could give an approximate idea of suitable models for ensembling.

Once the models were selected, they were blended uniformly or with weights optimized on the validation set. Both approaches gave comparable results.

The models selected by this process were not necessarily the ones with the lowest TTA score. Some models with relatively poor scores were selected because they make very different predictions than our other models. A few models had poor scores due to overfitting, but were selected nevertheless because the averaging reduces the effect of overfitting.

3.4.3. Bagging

To improve the score of the ensemble further, we replaced some of the models by an average of 5 models (including the original one), where each model was trained on a different subset of the data.

4. Code description

4.1. Relevant paths

- `configurations/*.py`: configuration files for all network architectures, to be used with `train.py` and `predict.py`
- `data/`: contains all train and test data
- `data/train/*/*.jpg`: the training images in their original format (JPEG files, contents of the data file `train.zip`)
- `data/test/*.jpg`: the testing images in their original format (JPEG files, contents of the data file `test.zip`)
- `data/labels_train.npy.gz`: gzipped numpy file containing the training labels (generated by `create_data_files.py`)
- `data/images_train.npy.gz`: gzipped numpy file containing the training images (generated by `create_data_files.py`)
- `data/images_test.npy.gz`: gzipped numpy file containing the test images (generated by `create_data_files.py`)
- `data/image_moment_stats_v1_train.pkl`: pickle file containing image moment statistics for the training data (generated by `compute_image_moment_stats.py`)
- `data/image_moment_stats_v1_test.pkl`: pickle file containing image moment statistics for the test data (generated by `compute_image_moment_stats.py`)
- `data/features_train.pkl`: pickle file containing other image features for the training data (generated by `extract_features.py`)
- `data/features_test.pkl`: pickle file containing other image features for the test data (generated by `extract_features.py`)
- `metadata/*.pkl`: training run metadata files, containing info about the training process as well as model parameters. Created and updated periodically by `train.py` during training.
- `predictions/train--*.npy`: training set predictions, generated using `predict.py`
- `predictions/valid--*.npy`: validation set predictions, generated using `predict.py`
- `predictions/test--*.npy`: test set predictions, generated using `predict.py`
- `splits/*.pkl`: validation splits for bagging, created using `create_bagging_validation_split.py`
- `submissions/*.csv.gz`: submission files in gzipped CSV format, created from prediction files using `create_submission.py`
- `validation_split_v1.pkl`: default validation split, created using `create_validation_split.py`

4.2. Python modules

4.2.1. Data and augmentation: data.py

The `data` module handles loading, pre-processing and augmentation of the images. It loads metadata and training labels automatically and has a `load()` function to load the train and

test image sets. It also has a bunch of functions to create and work with scikit-image's `AffineTransform` objects, which are used in pre-processing and data augmentation.

It features various generators for iteratively creating training data in the right format. The most important of these are `rescaled_patches_gen_augmented()` and `multiscale_patches_gen_augmented()`. They return pre-processed, augmented chunks of data that can be sent to the GPU without further processing.

4.2.2. Threaded generators: buffering.py

This small utility module implements two generators that take another generator as input and run it in a different thread / different process: `buffered_gen_threaded()` and `buffered_gen_mp()` respectively. We use this to ensure that data augmentation and model training happen in parallel.

4.2.3. Cyclic pooling layers: dihedral.py

This module implements neural network layers that perform cyclic pooling and rolling as described in Section 3.2. The layer classes are intended to be used with the Lasagne library.

4.2.4. Faster cyclic roll layers: dihedral_fast.py, dihedral_ops.py

These modules provide alternative, faster implementations for cyclic rolling based on custom CUDA kernels.

4.2.5. Quasi-random test-time augmentation: tta.py, icdf.py

These modules provide tools for generating a quasi-random set of augmentation transforms to use for test-time augmentation. The `icdf` module provides inverse CDFs for common distributions, which are used to convert quasi-random uniform samples.

4.2.6. Data loading: load.py

This file contains a collection of data loading classes. There is a different class for every kind of expected input (features / pre-processed and augmented images / pseudo-labeled data). A data loader class is instantiated in a configuration file, and is used in the `train_convnet.py` script to create data generators.

4.2.7. Additional neural network tools: nn_plankton.py

This module contains various extensions to the Lasagne library that were useful for this competition, including additional layer classes, update functions and initializers.

4.2.8. cuDNN based convolutional and pooling layers: tmp_dnn.py

This module contains cuDNN-based convolutional and pooling layers for Lasagne. In the meantime these have been merged into the main library so they are no longer necessary, but some of our code still depends on them.

4.2.9. Various utility functions: utils.py

This module contains various utility functions that don't fit anywhere else.

4.2.10. Configuration files: configurations/*.py

Configuration files are simply Python modules that specify a model architecture, data loader and various other training parameters. When `train_convnet.py` or `predict_convnet.py` are run with a configuration name as their command line argument, the corresponding configuration module is imported.

To resume training after an interrupted training run, make a copy of the configuration file that specifies the variable `resume_path`. It should be a string containing the path to the metadata file of the interrupted training run. The `train_convnet.py` script will load the parameters from the metadata file and resume training if this variable is present.

4.3. Python scripts

4.3.1. Create data files: create_data_files.py

Script to create the numpy files that contain the training and testing images and the training labels. This enables faster loading of the data.

4.3.2. Model training: train_convnet.py <config_name>

Given a configuration name, this script trains a model.

4.3.3. Generating predictions: predict_convnet.py <config_name or _> <metadata_path> <subset>

Given an optional configuration name, a path to a metadata file containing learned model parameters, and a subset (one of train, valid, test), this script generates predictions with test-time augmentation. The configuration name need not be specified and can be replaced by an underscore instead. In that case, the configuration name is extracted from the metadata file.

4.3.4. Evaluating validation set predictions: eval_predictions.py <predictions_file>

Compute the log-loss as well as top-1 through top-5 classification accuracies on the validation set for a given validation set predictions file.

4.3.5. Generate Kaggle CSV submission file: create_submission.py <predictions_file>

Given a test set predictions file, convert it into a gzipped CSV file which can be submitted to Kaggle.

4.3.6. Extract image features: extract_features.py <subset>, compute_image_moment_stats.py

These two scripts can be used to extract image features from the training and test data. The former requires specification of the subset (either train or test), the latter extracts features for both in one go.

4.3.7. Create validation splits: create_validation_split.py, create_bagging_validation_split.py <seed>

These scripts create pickle files that contain validation split information. The former generates the default split that we used for our experiments. The latter can be used to create different splits for bagging purposes, based on different random seeds.

4.3.8. Train models on image features and fuse them: blend_momentsinfo_haralick.py, train_fuse_blend.py, fuse_features_bagged.py

These scripts can be used to train the smaller networks that 'correct' the convnet predictions based on image features. For details of their usage please refer to section 6.4.

4.3.9. Ensemble predictions: ensemble_predictions.py

This script combines model predictions into a weighted blend by fitting weights on the validation set. All model names have been hardcoded so you will have to generate all the necessary predictions for it to work, or alternatively modify the code.

4.3.10. Merge pseudo-labeling model predictions with the blend: create_final_ensemble.py

After a model has been trained with pseudo-labeling using the ensemble, this script can be used to merge the predictions of this model with the ensemble.

5. Dependencies

5.1. Hardware

All convnets were trained on GPUs. Although our use of Theano technically allows for our code to be run exclusively on CPUs with minor modifications, this will be much too slow in practice. As a result, a recent NVIDIA GPU (Kepler- or Maxwell-based 600, 700, 800 and 900 series) with at least 4GB of memory is strongly recommended.

All machines we used to train the models had at least 16GB of memory, but 6 or 8GB should probably suffice.

5.2. Software

We used Linux machines to develop the code. Most of them ran **Ubuntu 14.04 LTS**, but other distributions should work. Some of the code may not run without modifications on Windows or Mac OS (e.g. code using `os.system()` to run commands).

We used **Python 2.7.6** with the following libraries and modules:

- **numpy** 1.8.2 - http://www.numpy.org/
- **scipy** 0.13.3 - http://www.scipy.org/
- **Theano** * - http://deeplearning.net/software/theano/
- **Lasagne** ** - http://github.com/benanne/Lasagne
- **scikit-learn** 0.15.2 - http://scikit-learn.org/

- **scikit-image** 0.10.1 - http://scikit-image.org/
- **ghalton** 0.6 - https://github.com/fmder/ghalton
- **PyCUDA** 2014.1 - http://mathema.tician.de/software/pycuda/
- **pandas** 0.15.2 - http://pandas.pydata.org/
- **mahotas** 1.2.4 - http://luispedro.org/software/mahotas/

All of these can be installed using `pip`.

* The Theano version should either be the development version from git, or 0.7rc1 or later. Version 0.6 (the current release at the time of writing) will not work.

** Lasagne is under heavy development and has not been released yet. As a result, the interface may change. We recommend installing the following commit, which our code is known to work with: `f445b71`. You can do this with the following command:
`pip install git+git://github.com/benanne/Lasagne.git@f445b71`

The following dependencies should also be installed:
- **openCV** 2.4 - http://opencv.org/
- **CUDA** 6.5 - https://developer.nvidia.com/cuda-zone
- **cuDNN** R2 - https://developer.nvidia.com/cuDNN

If you do not wish to regenerate the image features we extracted, you can use the files we included with the code. In that case, you do not need to install **mahotas** and **OpenCV**.

6. Generating the solution

First checkout the code from the git repository. Next unpack the train and test zip files into the data/train and data/test directories, so that data/train contains 121 directories (one for each class) and data/test contains all the test images.

Next run **create_data_files.py** to create numpy files containing all the images and labels.

The code already comes with a pre-generated validation split file. To regenerate this file, or to create a validation split with a different seed use **create_validation_split.py**.

6.1. Training convolutional neural networks

6.1.1. Basic networks

For every model in the list of basic models (see appendix, section A.1.1.) run the following command:

python train_convnet.py CONFIG_NAME

Example: python train_convnet.py convroll4

Each trained model will be saved in the metadata/ folder.
An example model file could be “metadata/convroll4-desktop1-20150204-164548.pkl”

6.1.2. Models with preinit

Models with “preinit” use trained parameters/layers from another model file (see appendix, section A.1.2.). In our experiments these models use the first couple of layers from the convroll4 model file.
When the convroll4 model is trained edit the configuration file of every model with preinit:

replace CONVROLL4_MODEL_FILE with the absolute path to the trained convroll4 model file (in the metadata folder).

Next train only the following model: convroll5_preinit_resume_drop@420:

python train_convnet.py convroll5_preinit_resume_drop@420

The other 4 models with preinit will be trained in 6.1.3.

6.1.3. Bagged models

The bagged models use different train/validations split files. These are included in the splits directory. Every file is generated with a different seed. To (re)generate these files use **create_bagging_validation_split.py** with seeds 0 through 27 (28 additional splits in total).

For every model in the list of bagged models (see appendix, section A.1.3.) run the following command:

python train_convnet.py CONFIG_NAME

Example: python train_convnet.py bagging_00_convroll4_big_wd_maxout512

6.2. Generating predictions

For every model we have trained we will now generate predictions.
Run the following command for every model file in the metadata/ folder:

python predict_convnet.py _ MODEL_FILE test

Example:
python predict_convnet.py _ metadata/convroll4-desktop1-20150204-164548.pkl test

This will create test-set prediction files in the predictions/ folder.

An example prediction file could be
“predictions/test--convroll4--convroll4-desktop1-20150204-164548--avg-probs.npy”

For every model in basic models (see A.1.1.) and the convroll5_preinit_resume_drop@420 model also run the following commands to generate train and validation set predictions:

```
python predict_convnet.py _ MODEL_FILE train
python predict_convnet.py _ MODEL_FILE valid
```

These predictions are necessary for training feature models, and for the weighted ensembling procedure (Section 6.5).

6.3. Combine predictions for bagging

To combine predictions of all five copies of the bagged models (the one trained on the original validation split as well as the 4 copies with unique validation splits), use the following command:

```
python average_predictions.py 5_PREDICTION_FILES TARGET_FILE

Example: python average_predictions.py predictions/test--cr4_ds--*.npy
predictions/test--bagging_*_cr4_ds--*.npy bagged--test--cr4_ds--avg-probs.npy
```

6.4. Training feature models

We have provided extracted feature files with the code. This means the next step is **optional**. If you wish to recreate the feature files, you can do so as follows (if you make any changes to this procedure, you will have to recreate them):

```
python extract_features.py train
python extract_features.py test
python compute_image_moment_stats.py
```

To fuse features with a model, you will need to generate training, validation and test predictions of the original model. For every fuse, we use different configuration files named `featharalick_MODEL` and `featmomentsinfo_MODEL` (where you replace “MODEL” with the actual name of the model). In each configuration file, change the value of `train_pred_file`, `valid_pred_file` and `test_pred_file` to the path of the above-mentioned predictions.

For every non-bagged model use this command to fuse features:

```
python train_fuse_blend.py CONFIG_NAME
```

Example: python train_fuse_blend.py featharalick_cp8

This will train 10 copies and generate 10 metadata files.

For every bagged model use this command (it will use the metadata of the non-bagged version of the model):

python fuse_features_bagged.py CONFIG_NAME 10_METADATA_FILES

Example:
python fuse_features_bagged.py featharalick_bagged_cp8 metadata/featharalick_cp8 -*

Finally, use this command to blend both “haralick” and “momentsinfo” predictions together:

python blend_momentsinfo_haralick.py 2_PREDICTION_FILES

Example: python blend_momentsinfo_haralick.py predictions/test--blend_feat*_cp8

6.5. Generate weighted ensembles

Once you generated validation and test predictions from A.1.5 and added features as described in 6.3 run this command:

python ensemble_predictions.py

This script will generate a .npy file with test predictions from a weighted blend of 10 models (7 bagged and 3 unbagged).
The script will not proceed unless you have all the prediction files with correct names, e.g. for some **CONFIG_NAME** validation predictions should be in
*valid--blend_featblend_* ***CONFIG_NAME****--featblend_* ***CONFIG_NAME****--avg-prob.npy.*
If this model is bagged its test prediction should be in
*bagged--test--blend_featblend_bagged_****CONFIG_NAME****--avg-prob.npy*
If this one of the unbagged configurations, test predictions should be in
*test--blend_featblend_* ***CONFIG_NAME****--featblend_* ***CONFIG_NAME****--avg-prob.npy.*

6.6. Training Pseudo-labeled convolutional neural networks

After ensembling, there should be a file called weighted_blend.npy in the predictions folder that was generated in the previous step.

For every model in the list of pseudo-labeled models (see appendix, section A.1.4.) run the following command:

python train_convnet.py CONFIG_NAME

Example: python train_convnet.py pl_blend4_convroll4_doublescale_fs5_no_dropout_33_66_sharded_0

6.7. Create final ensemble

Run the following command for the 3 models trained in the previous step.

python predict_convnet.py _ MODEL_FILE test

Example:
python predict_convnet.py _ metadata/pl_blend4_convroll4_doublescale_fs5_no_dropout_33_66_sharded_0-desktop1-20150204-164548.pkl test

Next run **create_final_submission.py**, which will combine these 3 predictions with the weighted ensemble from section 6.4. The final submission can be found in the predictions dir, and is called “final_prediction.npy”. To create the kaggle submission file run

python create_submission.py predictions/final_prediction.npy

A. Appendix

1. A.1. List of all convolutional neural networks

A.1.1. basic models

- *convroll4*
- *convroll4_1024_lesswd*
- *convroll4_big_wd_maxout512*
- *convroll4_big_weightdecay*
- *convroll4_doublescale_fs5*
- *convroll_all_broaden_7x7_weightdecay_resume*
- *cp8*
- *cr4_ds*
- *doublescale_fs5_latemerge_2233*
- *triplescale_fs2_fs5*

A.1.2. models with preinit

- *convroll5_preinit_resume_drop@420*
- *bagging_24_convroll5_preinit_drop@420*

- *bagging_25_convroll5_preinit_drop@420*
- *bagging_26_convroll5_preinit_drop@420*
- *bagging_27_convroll5_preinit_drop@420*

A.1.3. bagged models

- *bagging_00_convroll4_big_wd_maxout512*
- *bagging_01_convroll4_big_wd_maxout512*
- *bagging_02_convroll4_big_wd_maxout512*
- *bagging_03_convroll4_big_wd_maxout512*
- *bagging_04_convroll4_doublescale_fs5*
- *bagging_05_convroll4_doublescale_fs5*
- *bagging_06_convroll4_doublescale_fs5*
- *bagging_07_convroll4_doublescale_fs5*
- *bagging_08_convroll_all_broaden_7x7_weightdecay*
- *bagging_09_convroll_all_broaden_7x7_weightdecay*
- *bagging_10_convroll_all_broaden_7x7_weightdecay*
- *bagging_11_convroll_all_broaden_7x7_weightdecay*
- *bagging_12_convroll4_big_weightdecay*
- *bagging_13_convroll4_big_weightdecay*
- *bagging_14_convroll4_big_weightdecay*
- *bagging_15_convroll4_big_weightdecay*
- *bagging_16_cr4_ds*
- *bagging_17_cr4_ds*
- *bagging_18_cr4_ds*
- *bagging_19_cr4_ds*
- *bagging_20_cp8*
- *bagging_21_cp8*
- *bagging_22_cp8*
- *bagging_23_cp8*
- *bagging_24_convroll5_preinit_drop@420*
- *bagging_25_convroll5_preinit_drop@420*
- *bagging_26_convroll5_preinit_drop@420*
- *bagging_27_convroll5_preinit_drop@420*

A.1.4. pseudo-labeled models

- *pl_blend4_convroll4_doublescale_fs5_no_dropout_33_66_sharded_0*
- *pl_blend4_convroll4_doublescale_fs5_no_dropout_33_66_sharded_1*
- *pl_blend4_convroll4_doublescale_fs5_no_dropout_33_66_sharded_2*

A.1.5. Required model predictions for ensembling

A.1.5.1 Validation set predictions:

- *convroll4_doublescale_fs5*

- *cp8*
- *convroll4_big_wd_maxout512*
- *triplescale_fs2_fs5*
- *cr4_ds*
- *convroll5_preinit_resume_drop@420*
- *doublescale_fs5_latemerge_2233*
- *convroll_all_broaden_7x7_weightdeca*
- *convroll4_1024_lesswd*
- *convroll4_big_weightdecay*

A.1.5.2 Test set predictions from unbagged models:

- *triplescale_fs2_fs5*
- *doublescale_fs5_latemerge_2233*
- *convroll4_1024_lesswd*

A.1.5.3 Test set predictions from bagged models:

- *convroll4_doublescale_fs5*
- *cp8*
- *convroll4_big_wd_maxout512*
- *cr4_ds*
- *convroll5_preinit_resume_drop@420*
- *convroll_all_broaden_7x7_weightdeca*
- *convroll4_big_weightdecay*