ML2016 HW3 Semi-Supervised Learning in Picture Classification

November 20, 2016

1 Supervised learning

In my implementation of supervised learning, I use the CNN sample code from the github of the developer of Keras:

(https://github.com/fchollet/keras/blob/master/examples/cifar10_cnn.py).

I made some minor change from the original code, such as dimention adjustmnet (I use Theano backend so I have to change the dimension of images from `(32, 32, 3)` to `(3, 32, 32)`), and I didn't use data augmentation. That is , I just use all `5000` labeled data as training and tried some combinations of learning rates and batch sizes. I determined the learning rate and batch size by 5-fold cross-validation. After some tuning, it seems that learning rate = `0.005` and batch size = `64` leads to the maximum accuracy on the validation set (averaged over 5 folds).

The final Kaggle private score is `0.54840`.

2 Semi-supervised learning (1): Self-training

First, I concatenate the `45000` unlabeled data with the `10000` test data to make a larger (`55000`) unlabeled data. Second, I split the `5000` labeled data into `4000` training (80%) and `1000` validation (20%) by `train_test_split` function from `sklearn.model_selection`.

I use a `while` loop to add a portion of unlabeled data into labeled data. For each iteration, a temp model is trained on the temp labeled data (initial to the `4000` labeled training data) with the `1000` labeled validation data being the `validation_data` in the fit function call, and then use the trained model to predict the labels of the temp unlabeled data (initial to all the 55000 unlabeled data). Then, I choose those unlabeled data with predicted probability of its "most-probable class" exceed `0.99`, concatenated them with the temp labeled data, and remove them from the temp unlabeled data.

Since the temp labeled data for each iteration will become larger and larger, adaptive learning rate and batch size are used (slightly larger batch_size and smaller learning rate for larger training data). `EarlyStopping` is also used with `monitor = 'val_loss'` and `patience = 2 + count` (also apative), where `count` is just the variable counting number of iterations (start from `1`).

As for the condition for the while loop to stop, I save the history of those temp models and compare their maximum `val_acc`. If the maximum `val_acc` stop increasing, then the **current** temp model is choosed to determine the final setting of the final model. The final model is then trained using all `5000` labeled data + all the unlabeled data that had been choosen previously (using the predicted pseudo labels). In this step, the size of my training data is around `12000`.

Then, the trained final model is used for the final predictionon on the `10000` test data.

The final Kaggle private score is `0.58600`.

3 Semi-supervised learning (2): Autoencoder pretraining

First, I concatenate the `5000` labeled data with the `45000` unlabeled data to make a larger (`5000`) training data.

The autoencoder I impelement is just three DNN (not convolutional autoencoder) with input being flatten images of size `3072`. The hidden layer in the first autoencoder is `512`, the second `256`, and the third `64`. In the first autoencoder, I used the aformentioned `50000` training data. In the second autoencoder, I used the encoded `512`-sized code from the first autoencoder as the training data. In the third autoencoder, I used the encoded `256`-sized code from the second autoencoder as the training data. All autoencoders are trained using `adadelta` optimizer, `binary_crossentropy` loss function, batch_size = `64`, and ran `50` epoch for the first autoencoder and `100` epoch for the second and the third.

After derived the three autoencoders, I re-use their encoder parts to build my final DNN for image classification. This can be done using functional API such as `layer1 = encoder1(input_img)` followed by `layer2 = encoder2(layer1)` and then `layer3 = encoder3(layer2)` as showed in my source code **autoencoder.py**. Finally, I got a `512-256-64-10` DNN for image classification. I fit the model using `sgd` optimizer with a small learning rate (`0.00001`) since there are too many weights in the network. I ran `150` epoch and set a relatively small batch size of `16` to get more updates, but both the training and validation accuracy stop increasing around `0.3`.

Since the results of `val_acc` are bad during training (around `0.3`), so I didn't submit the results to Kaggle.

4 Results comparison and analysis

My best result comes from **self training** (`0.58600`). Since the Kaggle private score only improve about `4%` compared to **supervised learning** (`0.54840`), I think maybe it's the limitation of *naive* self-training. That is, we need more sophisticated image processing, clutering, or pretraining techniques to improve the performance.

As for my **autoencoder pretraining** approach (validation accuracy around `0.3`), the main reason of bad performance might come from insufficient parameter tuning, since the training validation is not good either (around `0.4`). But I think the most fundamental reason is: I should implement **convolutional** autoencoders for image data, as they are more suitable for image detection/classification tasks.

Furthermore, I also tried the `adam` optimizer to replace `sgd` after Kaggle deadline, and the validation accuracy is higher for both **self training** (slightly above `0.6`) and **autoencoder pretraining** (almost `0.4`). It can be seem that the `adam` is really a promising optimizer and yet easy to implement.