1 Overview

Idea. Trying to get the maximum from datsets where we have very few training examples, but each example has a very large number of features. Examples of such datasets include medical databases where we have gene activation measurements for very few patients but many different genes.

Method. We design a neural network architecture whose number of parameters is constant with respect to the number of features (which is not the case with a typical linear classifier). The basic idea is that we use a linear classifier whose coefficients for each features are generated by a single neural nework that takes as an input a representation for this feature, which is basically a transformation of the set of values taken by this feature through ale the exampes. More complex (deep) architectures are also experimented.

Datasets.

- ICML 2003 feature selection challenge datasets: Arcene, Dorothea
- AML/ALL Leukemia classification dataset

2 Models & implementation

Notations. We call X the matrix of training examples, where each line of X is an example, and each column r_j of X is the column of all the values taken by a feature throughout the training set. n is the number of lines of X (number of training examples) and d is the number of columns of X (number of features). We call Y the column matrix of training labels y_i.

Code. The code uses Blocks and Fuel frameworks developped at MILA. The `dataset.py` and `datastream.py` files are responsible for loading the datasets into Fuel. The `train.py` script is the main script of the project, that loads the data and the model and does the training. `train.py` expects one argument, which is the name of a model to do the training on. For instances, existing models include `mlp.py`, `mlpfsel.py`, `mlpfsel2.py`, etc. To run the model from `mlp.py`, type:

```
$ ./train.py mlp
```

2.1 First idea

This is a first model for binary classification. It can probably be extended to n-ary classification.

Model. The example x is decomposed in its features x_j. We have a function $f(x_j, r_j)$ which is a MLP that calculates a probability that the example is a positive sample. We then have them vote to produce classification $\hat{y}$:

$$\hat{y} \ = \ \sum_{j=1}^{d} f(x_j, r_j)$$

During training, x is one of the rows of X. During validation and testing, different examples are chosen. The parameters of the model are the parameters of the MLP implemented by f.

Implementation. A simple implementation of this model is in the repo, in `mlp.py`.

2.2 Recurrent neural nets

Model. We do the same as previously, but only the NN is a recurrent net that scans the features successively in a random order. A prediction is output at each timestep.

Implementation. A simple implementation is available in `rnn.py` that uses LSTM. A more complex implementation in `mlprnn.py` first passes (x_j, r_j) through a MLP, which can be usefull for dimension reduction.

2.3 MLP-generated linear classifier

This model is also described for binary classification and can also be extended to n-ary classification.

Model. We have a function $f(r_j)$ which for each features gives a single coefficient in a linear classifier:

$$\hat{y} \;=\; \sigma\left(\sum_{j=1}^{d} f(r_j)\, x_j + b\right)$$

where σ is a sigmoid function. The parameters of the model are the bias b and the parameters defining the MLP f.

Implementation. A simple implementation of this model is in `mlpfsel.py`.

2.4 MLP-generated MLP

This is the model that has been experimented with the most.

Model. The idea is that the function $f(r_j)$ gives us not only one coefficient for each feature, but a bunch of them, producing the matrix W_0 of weights for the first layer of a MLP:

$$\begin{aligned} W_0 \;&=\; \begin{pmatrix} f(r_1) \\ \vdots \\ f(r_d) \end{pmatrix} \\ \hat{y} \;&=\; g(x\, W_0^\top) \end{aligned}$$

Both f and g are MLPs whose coefficients are the parameters of the model.

Implementation. Available in `mlpfsel2.py`

Extensions.

- `mlpfsel3.py` : f is decomposed in two successive MLPs, and a reconstruction penalty is applied so that from the output of the first half of $f(r_j)$ we can reconstruct the full vector r_j (through yet another MLP).

- `mlpfsel4.py` : we divide the features r_j into several (possibly overlapping) subsets, and then have a separate classificaion system $f^{(k)}, g^{(k)}$ for each subset:

$$\begin{aligned} r_j^{(k)} \;&=\; \pi^{(k)}(r_j) \\ W_0^{(k)} \;&=\; \begin{pmatrix} f^{(k)}\Big(r_1^{(k)}\Big) \\ \vdots \\ f^{(k)}\Big(r_d^{(k)}\Big) \end{pmatrix} \\ \hat{y}^{(k)} \;&=\; g^{(k)}\Big(x\, W_0^{(k)\top}\Big) \\ \hat{y} \;&=\; \frac{1}{K}\sum_{k=1}^{K} \hat{y}^{(k)} \end{aligned}$$

- `mlpfsel5.py` : combination of the approaches from `mlpfsel3` and `mlpfsel4`.

- `mlpfsel2ae.py` : a deep auto-encoder is first pre-trained to encode the r_j vectors into a more compact/sparse representation, that is then fed to f.

- In `mlpfsel2.py`, principal component analysis can be done on the r_j vectors so that r_j are replaced by their projectons on the m most present components.

- Various forms of regularization (dropout, noise, L1 penalty) are sometimes implemented in all the models.

3 Current results

Convergence. It is hard to make any of these models converge. The only optimization algorithm that we have managed to use is AdaDelta. A simple SGD momentum does work for some models (`mlp`, `mlprnn`) but it stays on a plateau for a very long time before the costs goes down. RMSProp does not work at all: in all our experiments the cost diverges and the model parameters become meaningless.

Baseline. For the NIPS 2003 feature selection challenge, previous results are available at this URL. An online judge for trying solutions against the undisclosed test set is available here.

Overfitting. A typical behaviour for all the models is overfitting. The training costs and error rate typically go to zero, while the validation cost and error rate diverge. For instance this is what we observe on Arcene, using 100 training examples and 100 validation examples:

- The validation cost starts at 0.6 or 0.7, goes down a bit, then back up. When the model is settled at zero error for the training, the validation cost continues a regular upward progression going often way above 1.

- The validation error rate goes down a bit and stabilizes above 15%, often above 20%. Best hyperparameter choices have converged at 13% validation error. Submissions to the online judge have yielded balanced error rates above 15% for the undisclosed test set, which is far from the best entries that manage to do 7% error rate.

Regularization.

- Noise on weights and on r_j inputs can be used but does not bring us to exceptionnal performance. Too much noise makes the model fluctuate and the validation costs do not converge anymore.

- Dropout does not help us much either, and too much dropout has the same effect as too much noise.

- In the `mlpfsel2` model, applying a L1 penalty on the weights of the first MLP (f) is effective to prevent the validation cost from diverging, but does not improve the error rates.

Hyperparameter search. Given the number of different models and the number of hyperparameters that can be tweaked, the hyperparameter search has been extremely cumbersome and no definitive sweeet spot has yet been identified.

Best solution at the moment. The best solution found at the moment that brings the validation cost to 13% on Arcene is based on the following model:

- `mlpfsel2` model, no hidden layer in f, f outputs 10 coefficients, no hidden layer in g

- pre-processing is a PCA on the vectors r_j and about 50 out of 100 components are kept

- a L1 penalty of 0.01 is applied on the coefficients of f, and a L1 penalty of 0.001 is applied on the input to g

This solution has not yet been tested against the full (undisclosed) test set.

4 Further tasks

- Evaluation of current solutions:
    - Evaluate a standard linear classifier as a baseline ?
    - Submit on Codalab a solution obtained with models attaining a low validation score
- New solutions:
    - Using a deep auto-encoder with pre-training (without layerwise pre-training it won't converge) but that can continue to evolve during the training to better fit the prediction cost
    - Adding to the cost a reconstruction penalty for the deep auto-encoder, trying to balance the two costs
    - Add to that a L1 penalty on the MLP weights
    - Think of new solutions
- Hyperparameter search