Remote Processing for Machine Learning

Assignment 3

ENGG 6500 | Intro to Machine Learning

Dr. Graham Taylor

Jesse Knight

2015-11-25

1 Remote High Performance Computing: Motivations

A major utility of machine learning (ML) is its ability to operate on highly complex data. A corollary to this feature is a requirement for high performance computing (HPC). It is therefore fundamental for ML practitioners to understand modern HPC resources; this can increase the efficiency of model development as well as runtime implementations. In this report we examine the use of a remote processing platform, SHARCNET, for training a recurrent neural network (RNN). The task of the RNN is to predict three labels for a given tweet (as in Twitter) related to weather events:

- *Sentiment*: How does the author feel about the weather (positive, neutral, negative, not sure)?
- *When*: When are the author's weather comments referring to (past, current, future, not sure)?
- *Kind*: What kind of weather the author referring to (e.g. sun, rain, wind, etc. – 15 conditions)?

1.1 Memory Distribution

Many HCP platforms are optimized for different tasks. For instance, large machine learning datasets may require large contiguous memory blocks. Given highly efficient modern matrix operation libraries, often it is the reading and writing from memory which bottlenecks algorithm speed; therefore memory allocation specifications should not be taken for granted. Conversely, we may have a small dataset, but wish to train a model a large number of times (e.g. if we are optimizing hyperparameters); this represents an opportunity for parallel processing, as no model instance relies on any other.

In the first case, we require a large *shared* memory capacity. Our matrix operations will require read and write capabilities of the same memory block, potentially at the same time – i.e. multiple CPUs operating on the same memory. In contrast, our parameter optimization should be run using *distributed* memory: a separate block of memory (and likely CPU) for each model instance.

1.2 Enabling Remote Software

On SHARCNET, many software packages are available for interpreting and running user-submitted jobs. However, these must be specifically activated before use. This helps to distinguish software versions and prevent namespace conflicts. To activate a package (e.g. Python 2.7), we use the `module load` command; to deactivate, we use `module unload`. Specific software versions may be specified using a directory-style syntax:

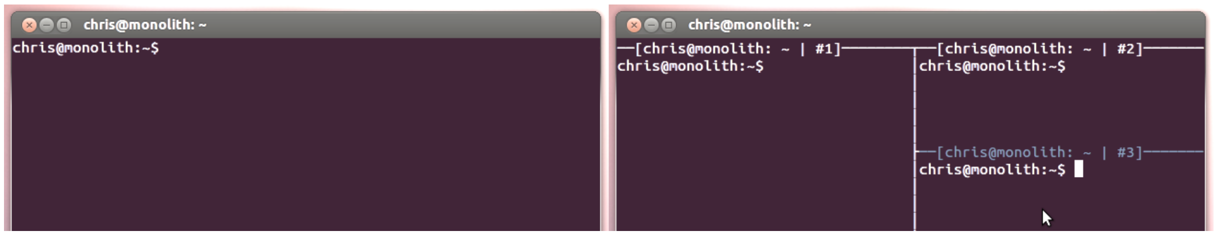
```
module load python/gcc/2.7.8
python python_script_name.py arg1 arg2 arg3
module unload python/gcc/2.7.8
```

2 Practical Remote Computing

2.1 Virtual Screen

SHARCNET has two types of nodes, login and compute nodes. Login nodes have limited computing power, so jobs should be run on the compute notes. While we may submit jobs to the compute nodes, we may also wish to interact with a compute node job. It is therefore useful to employ a virtual screen tool, like the `screen` tool available on SHARCNET. This allows the user to 'enter' a compute node environment (and any terminal-style sub-process therein) to monitor or interact with the running processes.

The virtual `screen` terminal has helpful features like command history and a 'clipboard' for copy-pasting text. However, while `screen` has multiplexing capabilities (connections to multiple applications), a single screen terminal can only show one application at a time. This can be a frustrating if comparing file contents. The updated version of screen, called `tmux`, addresses this limitation by parsing the original terminal into slices as designed by the user (Figure 1). Another similar application is `byobu`.



```
chris@monolith:~$
```

```
─[chris@monolith: ~ | #1]──────────┬─[chris@monolith: ~ | #2]──────
chris@monolith:~$                  │chris@monolith:~$
                                   ├─[chris@monolith: ~ | #3]──────
                                   │chris@monolith:~$
```

Figure 1: `screen` (single) vs. `tmux` (multiple) rendering of applications

2.2 Virtual Screen Parameters

Some useful `screen` options for use on SHARCNET – i.e. as an interactive "job" – include:

- `-D`: do not switch to this screen immediately; this is useful if multiple screens are invoked at once as we cannot switch to all simultaneously.
- `-m`: force create a new (temporary) screen, which terminates after the user exits by default.

2.3 Efficient Queue Use

Jobs on SHARCNET are queued and prioritized for nodes depending on their requirements for runtime and memory. However, some responsibility still falls on the user to specify these parameters when requesting a job using `sqsub`. It is important to estimate these correctly to not abuse the computing resources. Moreover, in the case of an erroneous job submission, or for `screen` jobs (which do not 'complete' unless they reach specified runtime) the user should know how to kill a job, using `sqkill -jobid` (or `-a` for all jobs):

```
 [jxknight@tem-login ~]$ sqsub –mpp=100M –r 5m –o /dev/null python test.py  # requesting screen job
submitted as jobid 12345678
[jxknight@tem-login ~]$ sqjobs -r                                          # checking job exists
   jobid  queue state ncpus  nodes time command
-------- ------ ----- ----- ------ ---- -------
12345678 serial     R     1 tem115   3s python test.py#
[jxknight@tem-login ~]$ sqkill -a                                          # killing all jobs
```

2.4 RNN Implementation

2.4.1 Model Development

The underlying data for our task are tweets: series' of words with constrained total character length (140). However, we have preprocessed the data using *Google*'s `word2vec` algorithm to represent words as vectors in a 20-dimensional feature space, such that similar "words" have a high dot-product in the space. We then use a RNN to make predictions using the tweets – series' of vectors – as short series data are well modeled with RNNs. Below is a summary of the architecture of the RNN employed for this task (default parameters).

- *Hidden Units*: The network employs a single layer of 50 recurrent hidden units, having rectified linear activation functions. Hidden units have weighted inputs from the preprocessed data $X\,W_{X\rightarrow h}$, weighted previous activations of all other hidden units $h^{t-1}W_{h^{t-1}\rightarrow h^{t}}$ (recurrence), and a bias term each b_h.

- *Output Units*: 24 output units are employed corresponding to the possible output labels:
  - 1 – 5 (*sentiment*): softmax functions; relative degrees of certainty about how the author feels.
  - 6 – 9 (*when*): softmax functions; relative degrees of certainty about when the event will occur.
  - 10 – 24 (*what*): sigmoidal functions; predictions (including negative) about whether each type of weather will occur – independent of one another.
- *Objective Function*: The original objective function is simply the mean squared error (MSE) between the predicted labels (probability) and the true binary labels for each of the output unit: $N^{-1}\sum(\hat{Y}-Y)^2$.

2.4.2 Scripting Considerations

Shared Variables

Training most RNN models is be computationally expensive. While our `theano` implementation will improve speed and accuracy using symbolic back propagation, `theano` also offers processor-level memory access boosts using `theano.shared` variables. Namely, multiple functions can read and write from a `shared` variable's memory simultaneously. This means parallelized matrix operations can update an entire variable at the same time. Strictly speaking – there actually is a scheduler for '`borrowing`' operations (e.g. read / write), but this is still much faster overall. In our python script, the entire set of vectorized model parameters, denoted `theta`, is stored as a `shared` variable, as well as the `theta_update` variable, input (`shared_x`) and label (`shared_y`) data.

Differentiable Looping

Loops, like other programming constructs, are not necessarily differentiable and their dependencies are not easily resolved by a compiler; `theano` therefore employs an augmented stand-in function called `scan` which allows a symbolic dependency graph to be constructed around the looped process. In our script, `scan` loops through all inputs (words) in single series (tweet) before defining the output activations for the whole series.

Variable Length Series'

In fact, in terms of error computation, each series is treated as a single input, regardless of length; this allows the loss function gradient with respect to each whole tweet in a mini-batch to be calculated in the same way. The recursive, variable-length nature of the gradient for the hidden weights is then managed symbolically by `theano`'s `tensor.grad`, through chain rule expansion as far back as necessary. For completeness, we also note that full epoch errors are reported as a weighted average of mini-batch errors, depending on the batch size.

Monitoring Training

During model training, we may wish to monitor various model parameters to examine how they change over the training epochs. This can be useful for evaluating hyperparameters' effects on convergence. For instance, we may log the epoch number, mini-batch number, training and validation set losses, and the learning rate:

```
INFO:__main__:epoch 30, mb 500/500, tr loss 0.241607 va loss 0.241457 lr: 0.009714
```

Additionally, during model development, monitoring weight norms and sample output activations can give the designer an intuition about whether the model is training reasonably or is profoundly off track.

```
INFO:__main__:norms: {W: 1.1706, W_in: 1.1469, W_out: 1.8261, h0: 0.1174, bh: 1.4087, by: 3.7892}
INFO:__main__:sample output: [[ 0.06811519  0.17460415  0.34003891  0.21857331  0.19866845  0.68188437
                                0.10939092  0.13640948  0.07231523  0.04187628  0.1212517   0.02842046
                                0.14309179  0.04672093  0.02578327  0.29380726  0.02687019  0.10215085
                                0.13780715  0.05860289  0.20299914  0.17945771  0.03654195  0.06763455]]
```

3 Implementation

3.1 Manual Job Submission

To submit jobs manually on SHARCNET, as noted above, we use the `sqsub` command. For a toy example, we compute the Collatz Conjecture solution (which is always 1) for arbitrary user specified integers using a python script, `cl_example.py`, which takes the integer as a command line argument. In Python, the `sys` module argument parser assigns all space-delimited strings after the `python` command to the `sys.argv` list variable; by default, the first argument (`sys.argv[0]`) is always the name of the python file.

```
[jxknight@tem-login ~]$ sqsub –mpp=100M –r 1m –o collatz_22 python cl_example.py 22
```

3.2 Batched Jobs

For a variety of reasons (e.g. hyperparameter optimization, leave-one out validation, etc.), we may wish to run multiple instances of the same script using different arguments. To do this, we may use a batch script, which invokes a set of command line arguments in series, with looping and variable functionality.

3.2.1 Example results

The output files of a batched set of toy `cl_example.py` jobs, submitted through SHARCNET, contain the results otherwise visible in the command line (e.g. all values converged to 1, in so many steps), as well as some information logged by the job hosting platform, SHARCNET (e.g. percentage use of requested resources).

3.2.2 Output Files

It is good practice during batched job executions to write separate output files for each job. This differentiates output from simultaneous jobs, which might otherwise be confusingly interleaved in the same file. Alternatively, the script may not use the “append” option when opening the output file, so the contents may be overwritten as each new job starts. This also prevents unnecessary queueing (if read/writing through a slow external connection), and ensures that any failed jobs will not affect the results of other jobs.

3.2.3 Scripts Interpreters

There are many executable file types in `Unix` environments, and using filename extensions is a “convention, not a rule”. It is therefore useful to specify the required interpreter for an executable file within the file itself. For example, this first line in a bash file, using the “*shebang*” flag, `#!`, invokes the `bash` interpreter.

```
#!/bin/bash
```

3.2.4 Accessing Variables

The `bash` scripting language is arguably the most hideous of languages; its syntax is as consistent as Charlie Sheen, and piped function sequences, while programmatically elegant, are unintelligible to the untrained reader, among other syntactic quirks. For instance, in the code below, to create a new *standard variable* `foo`, we simply set the name equal to some value `(1)`, yet to access the value, we must use `$foo`, `(5)`. If we require a *list variable*, `bar`, we define it as in `(2)`; however, to iterate through the list, we use “`for b in ${bar[@]}`”, `(3)`, which assigns `b` each value in `bar` during the loop `(6)`. Confusingly, omitting `{}` will “iterate” through the one-element list of the concatenated argument: first element of `bar`, plus the string “`[@]`”, yielding the output at right. Ridiculous. The author understands why knowing `bash` is useful, but is still a little bitter.

```
foo=0
bar=(1 2 3)
for b in ${bar[@]}
do
    echo $foo
    echo $b
done
```

```
OUTPUT with {}:
>> 0
>> 1
>> 0
>> 2
>> 0
>> 3
```

```
OUTPUT without {}:
>> 0
>> 1[@]
```

4 Model Regularization

4.1 Improving Training

We train our model using stochastic gradient descent (SGD); as in any training, we wish to avoid overfitting on the training data. While our dataset here is relatively large, the gamut of tweets the model might encounter during on-line deployment is much more vast. We therefore examine some methods of model regularization. Pertinent edits to the originally supplied python script are summarized in Figure 2.

4.1.1 Early Stopping

One of the more robust methods for preventing overfitting is *early stopping*. This method does not inherently limit the model complexity, but monitors the error J_θ on a validation data set X_V, which is separate from the training data X_T. While the training error will always decrease using gradient descent, an increase in validation set error indicates the onset of overfitting: a good time to stop training.

We employ a moving average of N previous epochs (indexed t) to identify validation error increase (4.1). This is more robust than a comparison with the previous single epoch, as the validation error may be noisy (especially if using injected noise for regularization, below). The initial conditions are ${J_\theta}^i = \infty,\ i \in \mathbb{Z}\,[1, N]$, implying we cannot stop training until at least N epochs.

$$\textbf{if}\ \left({J_\theta}^t(X_V) > \frac{1}{N}\sum_{i=t-1}^{t-N} {J_\theta}^i(X_V)\right) : \boldsymbol{stop\ training} \tag{4.1}$$

4.1.2 Regularization

We implemented two additional regularization methods, weight decay and noise injection, to improve model generalization even further. First, we augmented the loss function for training $J_\theta(X)$ with the L1 Norm of the weights to encourage weight decay (4.2). We employ the hyperparameter λ to control the relative impact of this term on the total $J_\theta(X)$ – i.e. relative to the MSE term. Input, recurrent, and output weight matrices $\{W_{h\to h}, W_{h^{t-1}\to t}, W_{h\to Y}\}$ are all included in the norm without per-matrix weighting, to limit hyperparameters.

$${J_\theta}^t(X) = \frac{1}{N}\sum_{i}^{N}(\hat{Y}_i - Y_i)^2 + \lambda\left(\left\| W_{X\to h} \right\|_1 + \left\| W_{h^{t-1}\to h^t} \right\|_1 + \left\| W_{h\to Y} \right\|_1\right) \tag{4.2}$$

Our second method for improving generalization is addition of Gaussian noise η to the hidden unit activations $h^t(X)$. This has the effect of introducing "new" activations (variations on the original set) to the output classifying units, so that they will not "memorize" the training set. Note that the input and recursive weights are still impacted during error backpropagation, despite the noise being added above their position in the graph. The hyper-parameter of this addition is the standard deviation of the noise, σ.

$$h^t(X) = ReLU\left((X\, W_{X\to h}) + \left(h^{t-1} W_{h^{t-1}\to h^t}\right) + b_h\right) + \eta, \qquad \eta \sim N(0, \sigma^2) \tag{4.3}$$

```
{...}
class RNN(object):
    def __init__(self, n_in=5, n_hidden=50, n_out=5, learning_rate=0.01,
                 n_epochs=100, batch_size=100,
                 reg_lambda=0.05, reg_noise=0.01, es_epochs=10,         # hyperparameters
                 learning_rate_decay=1, activation_type='tanh', final_momentum=0.9,
                 initial_momentum=0.5, momentum_switchover=5, snapshot_every=None,
                 snapshot_path='/tmp', output_types=None):
{...}
        self.reg_lambda = float(reg_lambda)  # L1 regularization coefficient
        self.reg_noise  = float(reg_noise)   # regularization hiddden activation noise SD
        self.es_epochs = int(es_epochs)      # early stopping epochs running average length
{...}
        self.rnd = T.shared_randomstreams.RandomStreams(seed=8497)      # seed the random stream
{...}
        def step(x_t, j, h_tm1, y):
            h_to = self.activation(T.dot(x_t, self.W_in) + \             # temp variable `h_to`
                                   T.dot(h_tm1, self.W) + self.bh)      # (without noise)
            # add normally distributed noise to hidden activations
            h_t = h_to + self.rnd.normal(size=self.n_hidden, std=self.reg_noise, ndim=1)
            y_t = {...}
{...}
            return h_t, y, y_t
{...}
        self.loss = lambda y: (self.mse(y) + (self.reg_lambda * self.regL1()))  # add L1 reg to cost
{...}
    def regL1(self):                           # definition of L1 regularization: sum of all weights
        return T.sum(T.sum(abs(self.W    ))) \
             + T.sum(T.sum(abs(self.W_in ))) \
             + T.sum(T.sum(abs(self.W_out)))
{...}
    def fit(self, X_train, Y_train, X_valid=None, Y_valid=None, validate_every=100,
        optimizer='sgd', nesterov=True, show_norms=True, show_output=True):
{...}
        if optimizer == 'sgd':
{...}
            prev_valid_loss = np.empty(self.es_epochs)  # weight vector of past validation losses
            prev_valid_loss.fill(np.inf)                # initialize these to inf
            early_stopped = False                       # boolean: have we met conditions for E.S.?

            epoch = 0
            while (epoch < self.n_epochs) and (not early_stopped):      # Checking not early stopped
                epoch = epoch + 1
{...}
                if self.interactive:                                   # ensure new validation error
                    if this_valid_loss > np.average(prev_valid_loss):  # compare to past vector avg
                        logger.info('STOPPING EARLY')
                        early_stopped = True
                        break
                    else:
                        prev_valid_loss[:-1] = prev_valid_loss[1:]     # shift history vector data
                        prev_valid_loss[-1]  = this_valid_loss         # append new validation error

{ __main__ }

reg_lambda = sys.argv[1]   # L1 regularization coefficient
reg_noise  = sys.argv[2]   # regularization hiddden activation noise SD
es_epochs  = sys.argv[3]   # early stopping epochs running average length
{...}
model = RNN(n_in=n_in, n_hidden=n_hidden, n_out=n_out,
            learning_rate=0.01, learning_rate_decay=0.999,
            n_epochs=n_epochs, batch_size=batch_size,
            reg_lambda=reg_lambda, reg_noise=reg_noise, es_epochs=es_epochs,     # hyperparameters
            activation_type='relu', output_types=output_types)
{...}
```

Figure 2: Python implementation of regularization methods (edits vs. original A3.py script). Colouring implies edits pertaining to: purple – L1 weight decay; blue – hidden unit activation Gaussian noise; green – early stopping.

4.2 Hyperparameter Optimization

4.2.1 Parameter Grid

To understand the impacts of each of the regularization methods on our model, we performed a simple grid search with the hyperparameters: L1 norm λ, and hidden unit noise σ. We also trained with and without early stopping, using a loss history length of 10 and 5000, respectively (5000 being equal to the number of epochs, making early stopping impossible). Below are our grid variables and an example job submission:

```
L1_lambda=(0.001 0.0001 0.00001 0.000001)  # iterated by L1_iter
Noise_SD=(10 1 0.1 0.01)                   # iterated by NSD_iter
Early_Stopping_Epochs=(10 5000)            # iterated by ESE_iter
{... for ... for ... for ...}
sqsub -f serial -mpp=12G -r 4h -o $outfile python A3_fast.py $L1_iter $NSD_iter $ESE_iter
```

4.2.2 Encouraging Overfitting

We noted that the default model training parameters did not encourage overfitting in less than thousands of epochs. While this is good for model performance, it makes demonstrating the impact of regularization hyperparameters difficult, particularly considering a 4-hour job runtime limit encountered on most SHARCNET compute nodes. We therefore made some modifications to the training conditions to encourage overfitting, including reduced training set size and increased number of hidden units:

```
# Data Preprocessing
train_X = train_X[2500:5000] # Default: train_X[50000:]   Impact: Reduce training clock time, also
train_y = train_y[2500:5000] # Default: train_y[50000:]           increase ease of overfitting:
valid_X = valid_X[:2500]     # Default: valid_X[:50000]           fewer training examples to memorize
valid_y = valid_y[:2500]     # Default: valid_y[:50000]
# Model Training Script
n_hidden = 150               # Defualt: 50                Impact: Increase capacity for overfitting
n_epochs = 5000              # Default: 2000              Impact: Increase epochs available to overfit
```

Using a modified `bash` submission script, we then submitted 4_λ x 4_σ x 2_{ES} = 32 model training instances with varied hyperparameters; the results are discussed below.

4.2.3 Results

Overview

A summary of the best training losses across regularization parameter combinations is available in Table 1. Our best validation data loss was 0.211, achieved without early stopping and the smallest magnitude of L1 weight decay and noise injection. This indicates that despite our modifications, overfitting was still not a significant challenge for training the model. That is, our regularization methods were usually constraining the model unnecessarily. This is most likely due to limited training runtime.

Table 1: Best model performances on each data set, with and without early stopping, and under varied regularization conditions. †Majority (88%) of instances reached at least 1500 epochs before the 4 hour SHARCNET runtime constraint

	Early Stopping		1500+ epochs†	
	X Training	X Validation	X Training	X Validation
Minimum Model Loss J_θ	0.215	0.217	0.156	0.211
L1 λ	1.0 x 10^{-6}	1.0 x 10^{-6}	1.0 x 10^{-6}	1.0 x 10^{-6}
Noise σ^2	0.01	0.01	0.01	**0.1**

Graphical Results

Our graphical results indicate that L1 regularization with $\lambda \in [10^{-5}, 10^{-6}]$ does not inhibit continued descent of training error, but prevents validation error from rebounding. Conversely, increasing hidden unit noise level maintains a narrow gap between the loss functions, but does so by pulling the training error up, rather than the validation error down – which is undesirable. At high noise levels, we see increased instability, while strongly decaying weights will consistently achieve $J_\theta \approx 0.25$; this may be sufficient, and is very reliable.

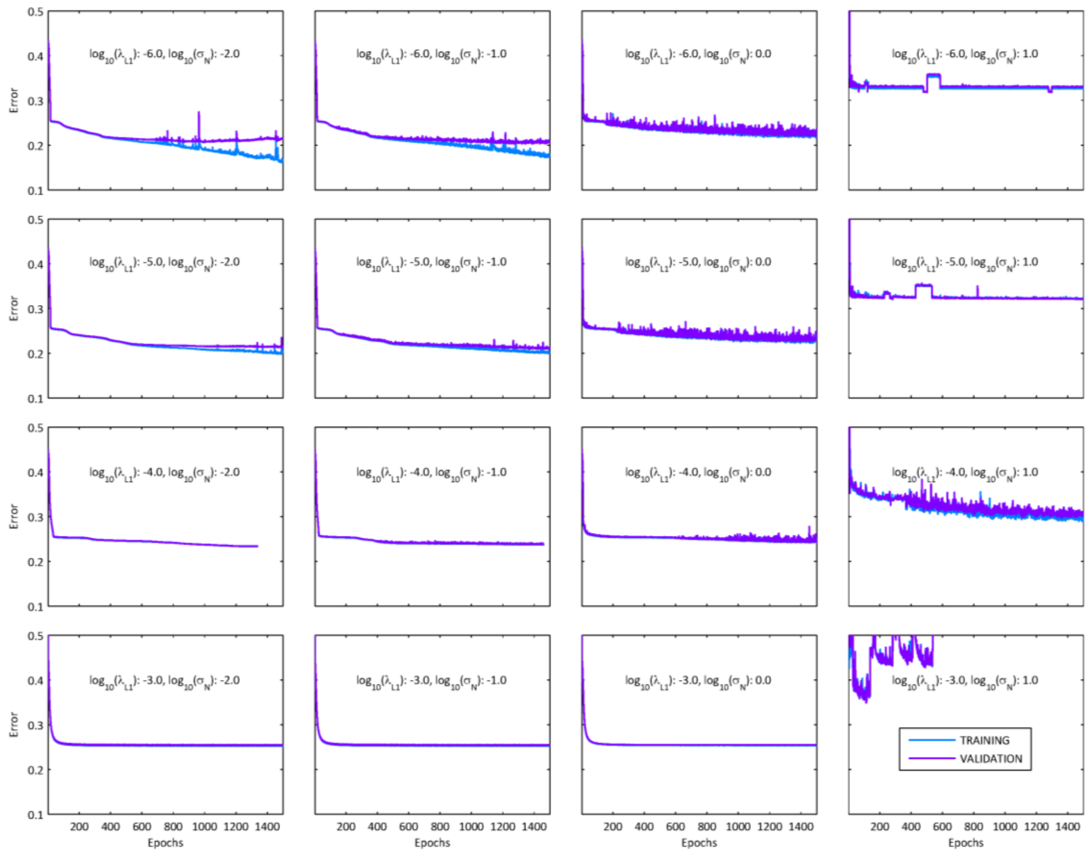


Figure 3: Error convergence without early stopping, using L1 weight decay and hidden unit activation noise (Gaussian); L1 coefficient is varied vertically and the standard deviation of the noise, horizontally, both by factors of 10.

Early Stopping Criteria

Lastly, the mean epoch at which early stopping occurred was only 91; in Figure 3 we observe that most instances do not show evidence of any overfitting at this time. We would therefore require a better early stopping criteria, as we are currently stopping much too early. For example, one alternative would be to store the minimum validation loss achieved over all epochs J_{MIN}, and stop training if, after an additional $N \approx 50$ epochs, we are still above this error by some factor $\alpha \approx 0.05$:

$$\textbf{if}\ \left({J_\theta}^{t}(X_V)\ > (1-\alpha)\cdot {J_{MIN}}^{(t-k)}(X_V)\ \textbf{and}\ (\mathrm{k} > \mathrm{N})\right)\text{: } \boldsymbol{stop\ training} \tag{4.4}$$

Recommendations

Based on our results, we recommend using L1 regularization, with $\lambda \|W_\forall\|_1$ (aggregate term) on the order of 1 (for 150 hidden units, this is $\lambda \approx 10^{-5}$). We do not recommend adding noise to hidden unit activations as it did not appear to improve performance, and has the risk of making convergence noisier. Finally, early stopping criteria should be refined, and any hyperparameters optimized to improve robustness.