CS281: Advanced Machine Learning: Assignment 3

Cedric Flamant

October 21, 2019

Exercise 1 and 2 - Neural Stack

The goal of this exercise is to implement and train a neural network to reverse strings. Crucially, the strings in the test data will be longer than those in the training data. Can the neural network actually learn a string reversal policy that generalizes to these longer strings?

The training data should have the following format:

$$S_1 a_1 a_2 a_3 \dots a_{k-1} a_k S_2 a_k a_{k-1} \dots a_3 a_2 a_1 S_3 \tag{1}$$

where S_1, S_2, S_3 are distinct fixed symbols of your choice and used to make explicit respectively the beginning of the input, the separation between the input and the output, and finally the end of the output. Each of the a_i should be chosen uniformly at random from a vocabulary of 128 meaningless symbols. The value k is sampled i.i.d. for each string uniformly at random **between 8 and 64**.

The test set should have the following format:

$$S_1 a_1 a_2 a_3 \dots a_{k-1} a_k S_2 \tag{2}$$

where k is sampled i.i.d. for each string uniformly at random **between 65 and 128**.

The goal in this task is for the neural network to learn to reverse a sequence. That is, given the following input from the test set:

$$S_1 a_1 a_2 a_3 \dots a_{k-1} a_k S_2 \tag{3}$$

the neural network should output

$$a_k a_{k-1} \dots a_3 a_2 a_1 S_3 \tag{4}$$

To that end, we will consider different seq2seq models all of which will take as input, one by one, the characters in the sequence from S_1 to S_2. After receiving S_2 as an input, the neural network will start ouputting symbols one by one, until it outputs the symbol S_3.

Exercise 1 Implementation. Using an automatic differentiation tool, "implement" the seq2seq model for the following neural architectures:

- A vanilla recurrent neural network (RNN)
- A Long Short-Term Memory network (LSTM)
- An LSTM equipped with a neural stack

You will find the specification of one possible architecture for a neural stack in section 3.1 of the paper "Learning to Transduce with Unbounded Memory" by Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, Phil Blunsom (`https://arxiv.org/abs/1506.02516`)

My general code organization followed a modular strategy. A `Reverser` class handled the operations we would expect a reverser to be able to do, such as `reverse()`. It also handles setting up the training sets, calculating

accuracy scores, and converting data types (from integer-encoded to one-hot encoded). A `Reverser` has a backend that actually does the learning and the reversing, and it can take any of the backends `RNN_Reverser`, `LSTM_Reverser`, `LSTMandStack_Reverser`. This common interface between all the backends allowed for easier parallel runs. The neural stack was also created as a separate class, `NeuralStack`, since it can be operated independently in principle, and attached to other neural controllers. I also wrote my code to be GPU-compatible, which turned out to be important since some of the larger networks were pretty computationally-intensive. The code is also heavily commented for easier interpretation and to reduce the chances of errors. This code was fairly involved to write, so I found it invaluable to plan out the structure and keep careful documentation of the types and dimensions of all input and output variables.

Vanilla RNN

I used an Elman RNN architecture, with an encoder and a decoder part. For the encoder, I used an `RNN` for better efficiency, since the RNN layer is more optimized than looping through `RNNCell`s manually with Python loops. For the decoder, I used an `RNNCell` to be able to handle the conditional checks that have to occur, like whether the terminal symbol ¡EOS¿ was output. In my code I never make explicit reference to a symbol dictionary; it is assumed that whatever dictionary of symbols is integer-encoded before it is passed to the reverse method. The `reverse` method of the `Reverser` takes the integer-encoded sequence and converts it to one-hot encoding for the backend. In the RNN backend, the encoder reads in the starting character, ¡SOS¿, and takes in input until right before the separator character ¡sep¿ is read. The hidden state is then passed to the decoder, which takes as its first input the separator character ¡sep¿, and then immediately makes its first prediction. The correct answers are fed in with one step delay (teacher forcing), to simulate inference mode when the network has to take back in its own guess as its next input. On the output of the decoder, I connected a linear layer which converts the hidden size to the output size. This is then passed to a log-softmax layer to prepare for cross-entropy loss minimization during training. During inference, the maximum index of the log-softmax output is converted into a one-hot vector which encodes the output character.

I also added the possibility of stacking layers of RNNs as a hyperparameter. This allowed for "deep" training, and as I will show this did have some benefits, particularly in extrapolation to the longer sequences in the test set.

LSTM

The LSTM backend has nearly an identical overall structure to the RNN. It consists of encoder and decoder sub-modules, which also can be stacked for deep learning.

LSTM with Neural Stack

The LSTM with neural stack backend is more intricate than the previous networks. The neural stack was connected to a LSTMCell in the way described in the referenced paper, with the same setup (like training the initial hidden state h_0 as a parameter, and keeping the initial value vector v and value vector stack V empty. All additional trainable matrices in the neural stack were Xavier-initialized, which seemed to play a role in convergence as I'll show in the next section. Biases were zero-initialized.

My initial implementation of the neural stack was very slow, since I was using Python lists of tensors. I found that by vectorizing the operations described in the paper, the stack updates could be sped up by a factor of 25.

Specifically, here are the vectorized stack updates:

$$\mathbf{V}_t = [\mathbf{V}_{t-1}, \mathbf{v}_t] \qquad \text{(tensor concatenation)} \tag{5}$$

$$\mathbf{s}_t[:t-1,:] = \text{relu}\big[\mathbf{s}_{t-1} - \text{relu}\big(u - A_{(t-1\times t-1)}\mathbf{s}_{t-1}\big)\big] \tag{6}$$

$$\mathbf{s}_t[-1] = d_t \tag{7}$$

$$\mathbf{r}_t = \min\big[\mathbf{s}_t, \text{relu}\big(1 - A_{(t\times t)}\mathbf{s}_t\big)\big]\mathbf{V}_t, \tag{8}$$

where $A_{t\times t}$ is a strictly upper-diagonal matrix of ones (i.e. the main diagonal is zero).

Exercise 2 Training. To train the neural networks, you can sample as many training examples as you see fit, using any strategy. For example, you may decide to start training on smaller examples and increase their length as training proceeds (curriculum learning). You should consider using different optimization strategies (SGD, Adagrad, etc...) and different parameters for these optimizers (learning rate, etc...). To test the neural networks, you should sample once and for all 1,000 examples from the test space. You need not obtain perfect accuracy, however, you should write your code so that you can run many experiments, potentially in parallel and continuously to try to find a good combination of hyper-parameters.

You must report on your attempts to train the networks (what optimizers, what hyper-parameters, etc). You should also produce plots showing the test error as a function of the training time for some of your best trials. Analyze what works and what doesn't, and try to explain why.

Training was performed locally and on AWS and Google Colab. Locally I ran models on an NVIDIA GTX 1060 6GB, on AWS I used a `g4dn.xlarge` instance with an NVIDIA T4 Tensor Core GPU, and on Colab I used NVIDIA K80 GPUs. Independent scripts that created `Reverser` objects with various hyperparameters made it easy to launch many runs simultaneously, and try a few hyperparameter choices. The networks were all pretty significant to train though, taking about 2 hours on average to approach convergence.

For the following plots, I used a shorthand to designate the hyperparameter settings:

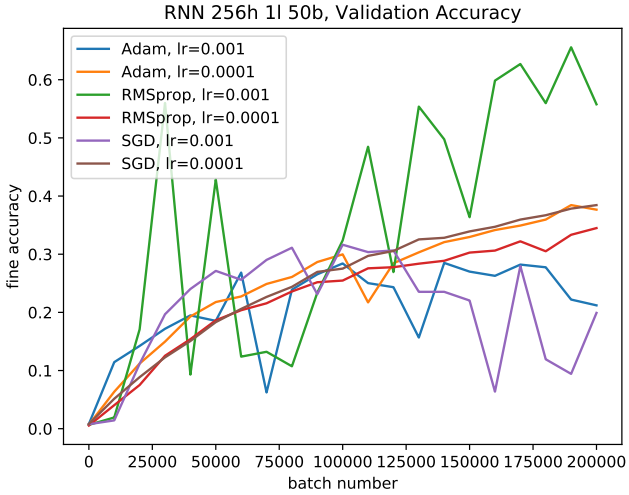
- **h** - This is the size of the hidden dimension. For RNNs, this is all there is. For LSTMs, this dimension is the size of both the hidden state and the cell state.
- **e** - This is the embedding dimension. This hyperparameter only exists in the neural stack. It describes the dimension of v, the vector that can be pushed onto the stack.
- **l** - This stands for the number of layers. Some of the RNNs and LSTMs were stacked to explore deep learning. While the LSTM with a Neural Stack was also given the option to have additional layers, I found that it was not necessary to test (and the single-layer LSTM + neural stack was already quite slow).
- **b** - This is the minibatch size. All networks were designed to handle a minibatch dimension for faster/more stable training.

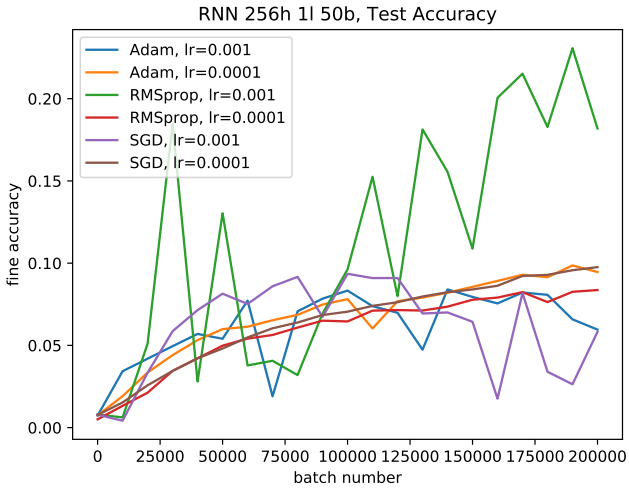
Below I present plots of the fine-grained accuracy,

$$\text{fine} = \frac{1}{\#\text{seqs}} \sum_{i=1}^{\#\text{seqs}} \frac{\#\text{correct}_i}{|\text{target}_i|}, \tag{9}$$

where #correct and #seqs are the number of correctly predicted sequences (end-to-end) and the total number of sequences in the test batch (1000). I show the accuracy on both a validation set, which consists of a fixed 1000 sequences of lengths in the range 8 to 64 (i.e. the same lengths as training data) and a test set, which consists of a fixed 1000 sequences of lengths in the range 65 to 128 (which will test the ability of the networks to generalize to longer sequences).

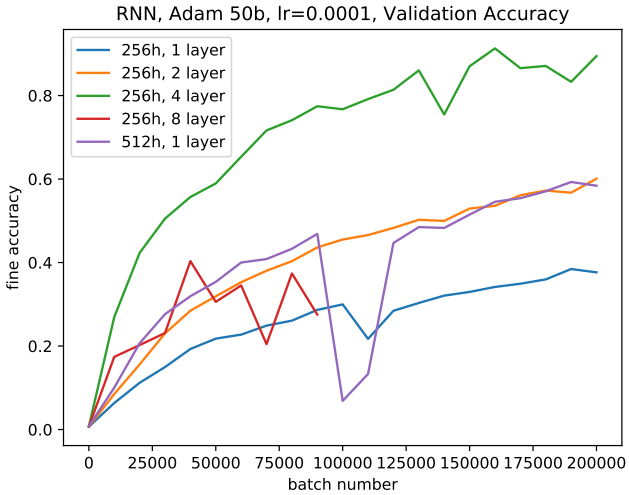
RNN

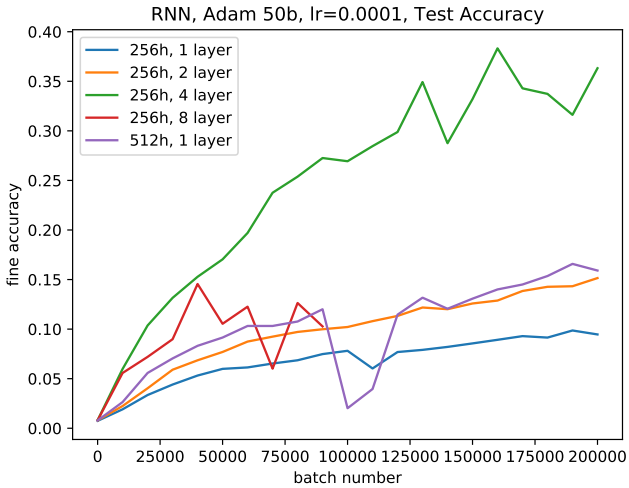




Before running for longer periods of time, I first tried out three different optimization methods, Adam, SGD, and RMSprop. For the most part in the above plot the optimizers seem to perform similarly for a given learning rate. The main surprise is that RMSprop, while it shows the same large variance as Adam and SGD for a learning rate 0.001, tends to increase accuracy faster. I did not think it would be a good choice for longer runs though since I imagine the large fluctuations would hamper convergence as the accuracy got closer to 1.

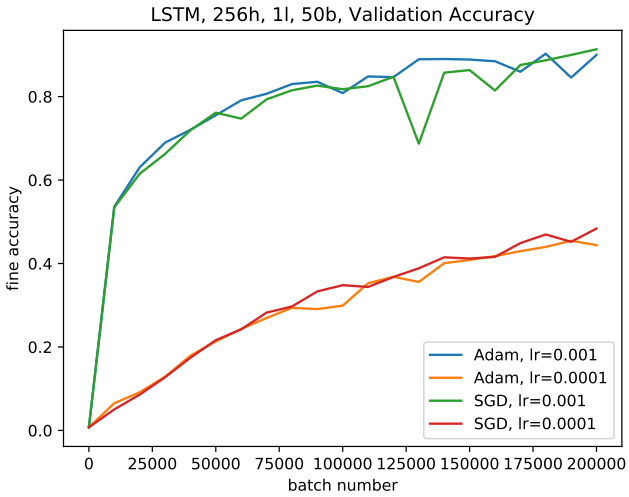
It is also interesting to note that performance on the validation set and the test set seem quite correlated, despite the fact that the possible sequence lengths are completely disjoint.

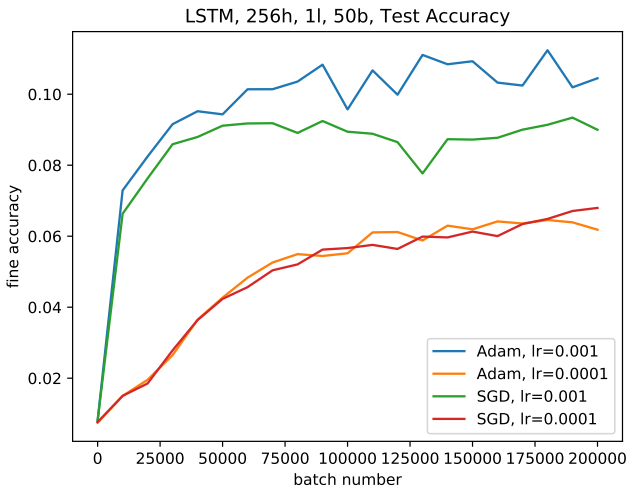




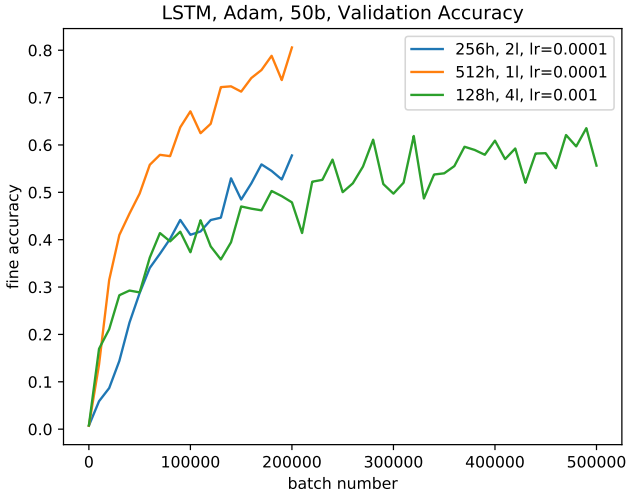
Next I investigated the effect of adding more layers and increasing the hidden state size. Interestingly, the deeper networks tended to gain accuracy faster on both the train and test sets, with exception of the deepest network, the 8 layer one. This however could be due to limitations of SGD-type methods catching up with the complexity. It is also interesting to note that the 4 layer RNN worked the best by far, with all other hyperparameters held constant. It learns fairly quickly, and it also seems to have learned a concept that is a little more general, since the gap is even more pronounced in the test data. This could be due to the extra layers helping establish more abstraction to the learning so it starts to work on a technique that is more nuanced than straight memorization.

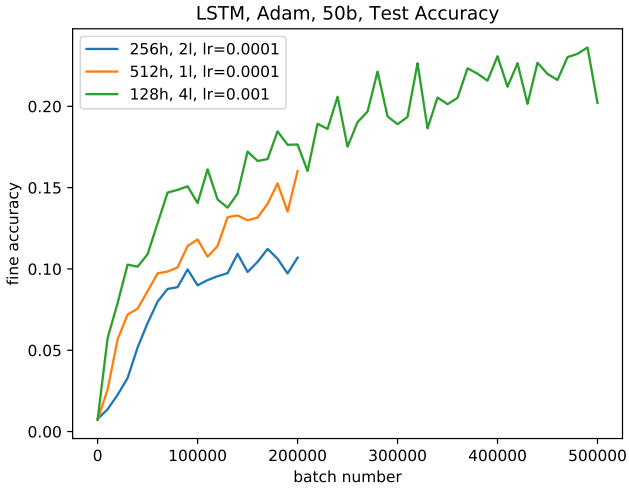
LSTM





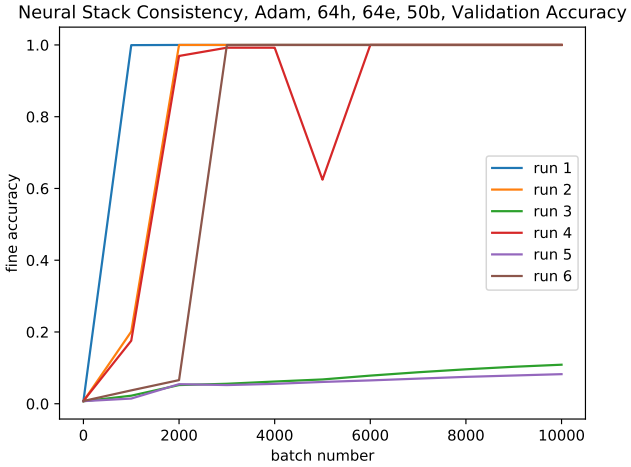
The LSTM learns very quickly relative to the RNNs seen earlier, as seen by the intensity of the initial slope. Again, we do not see much of a difference between the Adam and SGD methods, with the differences seen in the plot being due solely to the learning rate. Interestingly, the better learning rate for these networks was 0.001, which is a larger rate than the one preferred by the RNNs, 0.0001. Also, notice that while the LSTMs learn the training data well (remember that the validation data comes from the same distribution as the training data), it performs quite poorly on the test data, especially compared to the above RNNs. This suggests that the LSTM, true to its name, is better at the memorization task that is learning from the training data, but it does not pick up on more abstract concepts that would generalize to longer unseen sequences.

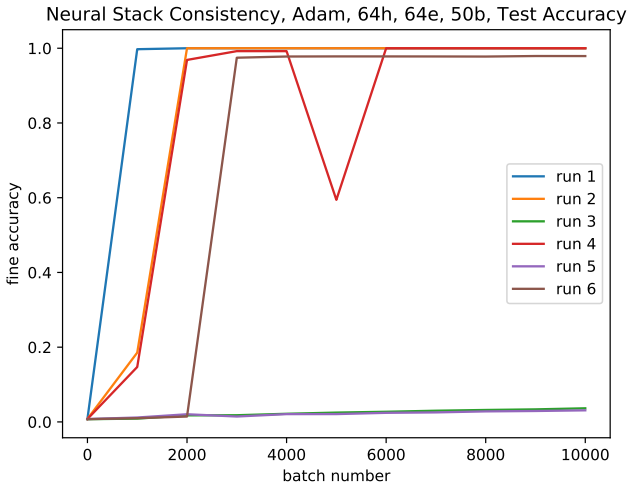


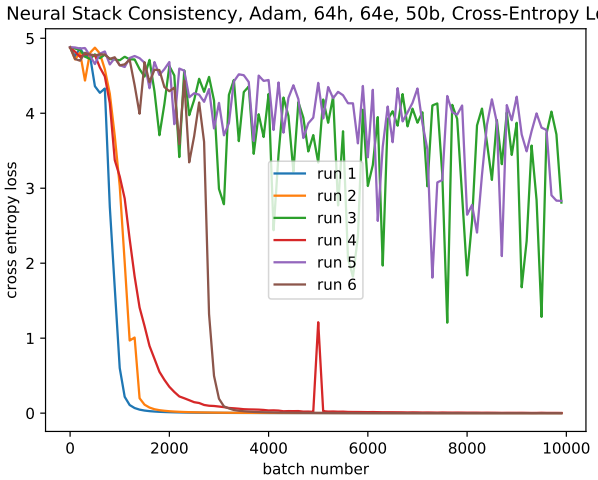


Again I performed a check to see how the layers and hidden dimension affect learning and accuracy. It is interesting to note that while the network with the most memory learned training data the best, it did not perform as well on the test data compared to the deep 4 layer network of much smaller individual hidden dimensions. This again supports my hunch that deep learning is helping with generalization in this problem. The large memory allows the 512h 1l network to memorize well, but it does not generalize concepts as well as the 128h 4l network.

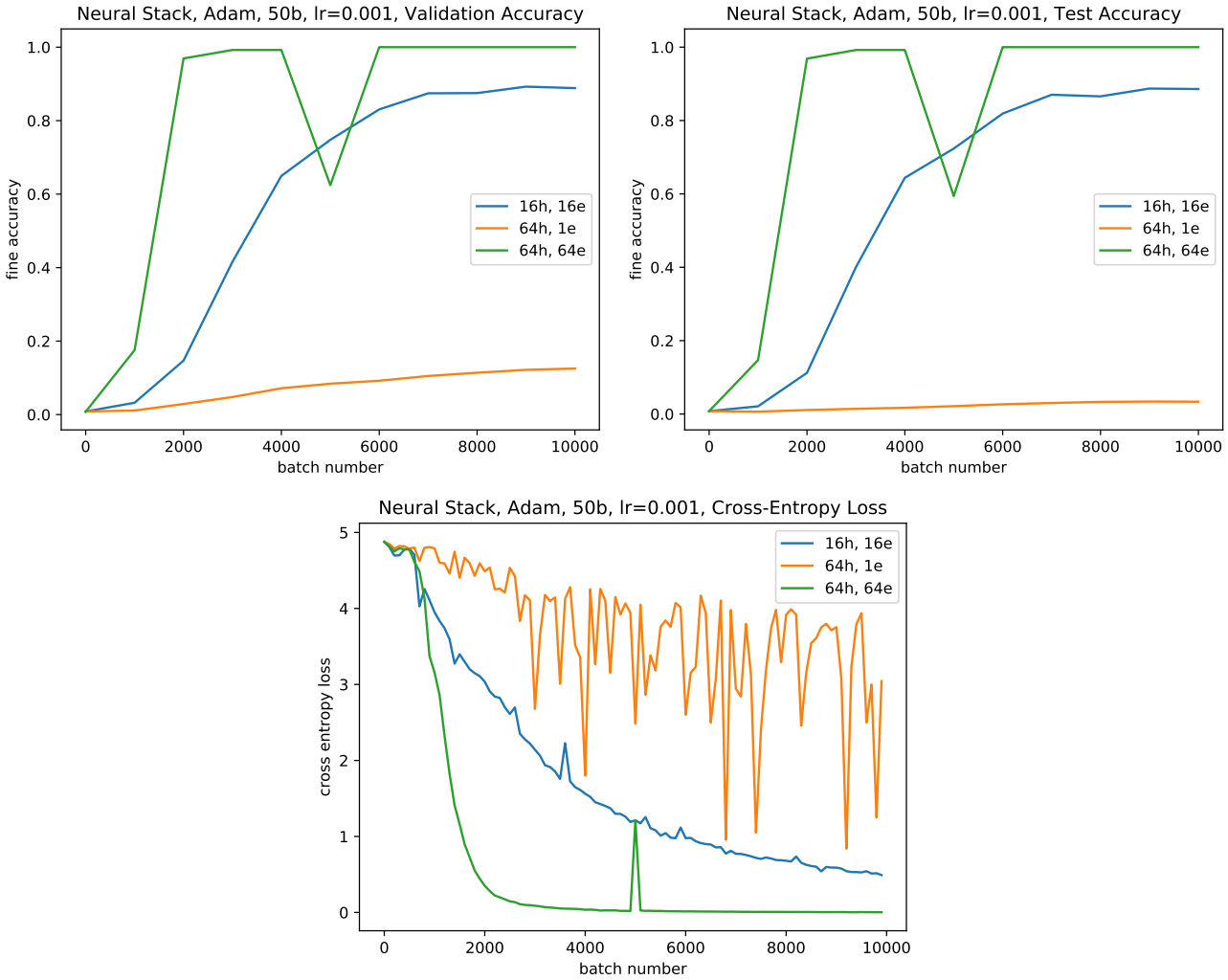
LSTM with Neural Stack





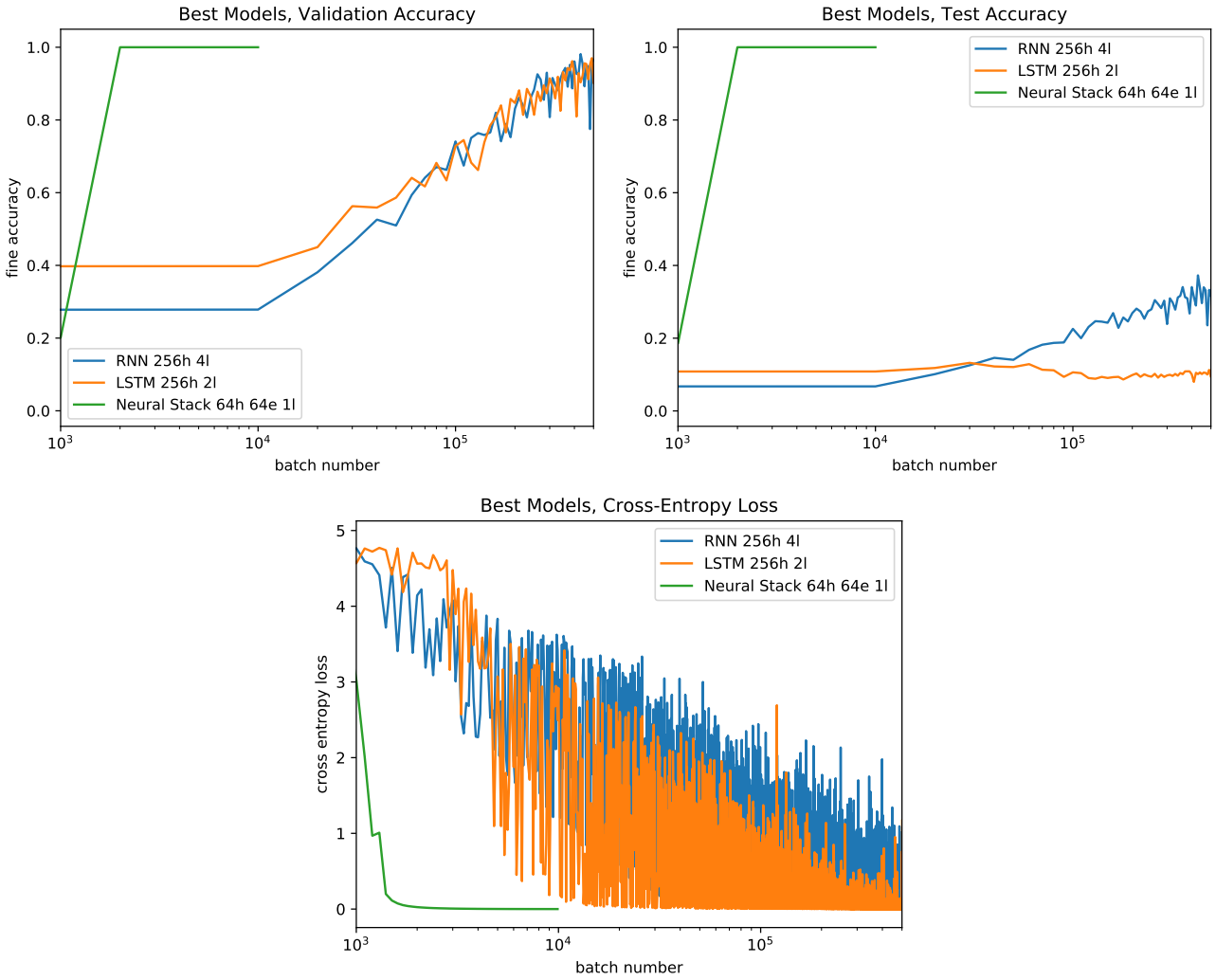


The LSTM equipped with a neural stack can perform extremely well when it converges. However, as the above plots show, even with the exact same hyperparameters, it might not figure out how to use the stack. This indicates that the random initializations of the weight matrices plays a role in whether convergence is attained. Notice that when it does figure out how to use the stack in training, it almost immediately gains a similar amount of accuracy on the test set. This shows that it has grasped the concept of pushing and popping the stack in a smart way. The networks that do not figure it out exhibit substantially slower learning, probably more in line with the bare LSTM from before. Also note that the neural stack networks achieve high accuracy much faster than the RNNs and LSTMS, after seeing 500 times less data! I also plotted the cross-entropy loss since that is computed more frequently than the accuracy scores. Notice that for the networks that figure out how to use the stack, the convergence is far more monotone. From the 4 networks that attained a perfect accuracy score on the validation data, 3 achieved perfect scores on the test data.



I also briefly explored some stranger stack configurations, containing either a small hidden dimension or a small embedding. For comparison I left one of the more standard 64h 64e runs. Notice that with less memory available to it, the 16h 16e network still performs fairly well on both the validation and test data. This shows that it is properly using the stack, but probably struggling to fit 131 dimensions worth of character data onto the size 16 embedding. When the embedding size is severely limited, like in the 64h 1e case, as expected the network fails to achieve a performance that is substantially different than a bare LSTM.

Comparison of Best Models



In the above figure I included the best model from each type of network. As expected, the neural-stack-equipped architecture vastly outperforms the other two in all metrics (especially noting how it needs to see two orders of magnitude less data to achieve this better performance!) It as also interesting to notice that despite the similar performance of the LSTM and RNN on the validation data which contains sizes the networks are familiar with, there is a substantial difference in how this performance generalizes to the longer sequences of the test data. This seems to suggest that the LSTM is memorizing without much abstraction of the task at hand, while the RNN, despite being a "less complex" network seems to benefit from its additional flexibility.

It is also interesting to look at the coarse accuracy of the networks, which is defined as the fraction of sequences from the sets that the network gets *exactly* right, all or nothing.

Coarse Accuracy

Network	Validation	Test
RNN	0.741	0.0
LSTM	0.505	0.0
NeuralStack	1.0	1.0

Notice that neither the RNN nor the LSTM could get a single full sequence right from the test set!