Deep Learning (IT-3030)
Project #2b: Implementing Recurrent Networks

This document describes the second assignment in IT3030, where you will implement recurrent layers for deep neural networks. By implementing both the forward and backward passes of backpropagation for recurrent layers, you will gain a detailed understanding of an essential mechanism in the AI toolbox.

The rules are as follows:

- The task must be solved individually.
- It must be solved in Python (or a similar language) without using software that is dedicated to deep learning, such as Tensorflow, Keras or PyTorch.
- Your solution will be given between 0 and 25 points; the rules for scoring are listed in the last section (Deliverables).
- The deadline for demonstration of this project is the week of March 8th - 12th, during that week’s regularly-scheduled lab hours.
- Code must be properly commented and uploaded to Blackboard prior to or (immediately) after your demonstration session.

1 Introduction

Sequential data poses a different set of challenges to a deep-learning system than do independent slices of information (e.g. simple feature sets, still images, etc). The tight dependencies between each element of a sequence must be recognized and leveraged in order to correctly perform a wide range of tasks. This range includes predicting the next element, classifying the sequence as a whole, interpreting the sequence, and translating the sequence into another.

Leveraging dependence requires the encoding of sequence history into the activations of the individual neurons such that the activation level of node n at time t reflects both the inputs to n at t and the outputs of n from many previous timesteps. Adding these self-loops to neurons is a trivial change to the network architecture, and adjusting your code for the forward pass is no less trivial; but the backward pass becomes quite a bit more complex, and that is the major challenge of this project.

In this project, you will implement recurrent networks of the Elman style: neural layers may directly feed back on themselves. These will be seamlessly integrated into your original implementation of backpropagation such that the user can specify, in the configuration file, that particular layers are recurrent. Your network generator will then connect recurrent layers to each other and to conventional (a.k.a. dense) layers.
The Neural Network

The networks built and run in your system will follow the same basic topology of Figure 1, with an input layer, anywhere from 0 to 5 hidden layers, and a final output layer. The hidden layers could be any combination of dense and recurrent types, and the final layer will generally be a standard dense layer, although traditional recurrent networks also allow softmax'ing of those outputs.

In this project, the task of predicting the next pattern in a sequence will not require softmax, although you are encouraged to include it to enhance the generality of your system. It is also quite possible that reformulating the prediction task as a classification task (by defining each possible pattern as a class) will enable your system to learn better. In short, incorporating softmax into your system is advised but not mandatory. A lot will depend upon the types of sequences that your system attempts to learn.

Figure 1: A typical topology for a recurrent predictor network built by your system. These networks can freely intermix normal (dense) layers with Elman layers (drawn with self-loops), but the final layer is typically a dense layer (with or without a softmax).

As in that earlier project, the weights are typically associated with the layer that they feed into, so a straightforward object-oriented implementation will bundle the units (a.k.a. neurons) of a layer with the incoming weights, all into a layer object. However, the recurrent layers require a second weight matrix to implement a self-loop, in which every unit has a weighted connection to itself and every other unit in that layer. The weighted outputs of a layer at time $t$ will then feed back into that layer at time $t+1$.

You must implement your system in an object-oriented manner. At the very least, your system must include a network object and at least two types of layer objects: dense and recurrent. As in the vanilla backpropagation project, all layers must at least have a forward_pass and backward_pass method.

The differences between recurrent nets and vanilla nets are significant enough that you will probably want to use a separate class for recurrent nets. When processing a configuration file, your system should get
some indication of the network type, either via an explicit global keyword (e.g. \textit{recurrent}) or simply by the presence of any layer whose declared type is \textit{recurrent, elman}, etc. If \textbf{any} layer in a network is recurrent, this affects the behavior of all layers in the network. For example, it forces all layers to cache all of their outputs for each forward pass of a sequence (for later use during the backward phase).

3 The Forward Pass

At the level of individual layers, the forward pass is similar to that of the vanilla backpropagation network, except that the recurrent layers receive their weighted inputs from two sources: their upstream neighbor and themselves. Each neuron in the recurrent layer simply adds together both of these sources and then applies its activation function to that sum.

On the forward pass, the main difference between the vanilla and the recurrent network involves the relationship between individual passes. In the vanilla net, each such pass is independent of the others: each layer implicitly gets a complete reset after each pass. Nets with recurrent layers are designed to handle sequential data, one item at a time. The only reset occurs at the end of each sequence of input-output pairs, and the history of previous items in the current sequence is recorded in the feedback inputs to the recurrent layers. Figure 2 (reprinted from the lecture notes) illustrates the processing of a single sequence consisting of several input-output pairs. Regardless of whether the minibatch size is only 1 (or larger), the network must do forward passes for each pair in the sequence before the backward passes begin. Also, all of the backward passes must finish prior to the updating of weights.

As in the vanilla-network project, you are free to send cases through the network individually or in minibatch chunks, but, as shown in the lecture slides, a minibatch is now organized differently. It is not the case that each of the minibatch items is one sequence of cases. Rather, the kth minibatch item is a collection of the kth input-output pairs from each of the M sequences. Hence, the size of the minibatch equals the length of the sequences (where you can assume that each sequence has the same length). The latter requires a bit more attention to the details of matrix operations, but the added flexibility (and speed) greatly improves your system. There is no point loss or gain for either approach. Once again, the use of numpy’s \texttt{einsum} function can greatly enhance the generality of your code, though it takes awhile to understand it.

To facilitate gradient calculations during the backward phase, the forward phase needs to save the outputs of each layer (whether recurrent or standard) for each of the forward steps. These can be deleted only after all gradients have been calculated for that sequence.

4 The Backward Pass

As with the forward pass, the backward pass can be performed one case at a time or with the entire minibatch in one shot. Regardless, the weights and biases should not be updated until every case in the minibatch has been passed back through the net and its gradients calculated.

As detailed in the lecture notes, there are 5 main types of Jacobian matrices that your code will need to keep track of for each recurrent layer (R) (starred Jacobians also pertain to standard dense layers):

- Output Jacobian$^*$ - storing the derivative of the loss function with respect to R’s outputs.
Figure 2: Processing of a single sequence consisting of three input-output pairs. Note that multiple rounds of forward propagation occur before an equal number of backward rounds. Note that the targets tA, tB and tC could be as simple as the next element in the sequence, e.g. tA = B, tB = C, tC = D, or they could be something more complex, such as a translation or an intermediate state of a sequence parse. For this project, the simplest interpretation (i.e. next element) is assumed.
• Neighbor Jacobian*- housing derivatives of R’s outputs with respect to the outputs of its upstream neighbor.

• Recurrent Jacobian - containing derivatives of R’s outputs at time k+1 w.r.t. its outputs at time k.

• Weight Jacobian* - recording derivatives of the loss function w.r.t. L’s weights. For recurrent layers, there are two such Jacobians to handle the two different weight matrices: input and recurrent.

• Delta Jacobian*- constituting derivatives of the loss function w.r.t. R’s sum of weighted inputs. For recurrent layers, this sum is the combination of sums from the input and recurrent weights. This is a useful intermediate representation that can greatly simplify the calculations of several of the other Jacobians.

The lecture notes provide the key details for implementing the backward phase for this project. The main process is summarized as follows:

• After feeding forward an N-item sequence, do N waves of backpropagation in reverse order: N, N-1, N-2...

• Each wave moves backwards from the output layer to the input layer.

• During each wave (k), each layer $Y_j$ will:
  – Receive Jacobian $\frac{\partial L}{\partial Y_{j,k}}$ from its downstream neighbor $Y_{j+1}$
  – If $Y_j$ is recurrent, modify $\frac{\partial L}{\partial Y_{j,k}}$ to produce $\frac{\partial L}{\partial Y_{j+1,k}}$. Otherwise, just copy the former into the latter.
  – Use $\frac{\partial L}{\partial Y_{j,k}}$ to compute the delta Jacobian, $\delta_j$.
  – If $Y_j$ is recurrent, use $\delta_j$ to modify loss gradients for recurrent weights.
  – Use $\delta_j$ to modify the loss gradients for standard (non-recurrent) weights feeding into layer $Y_j$.
  – Use $\delta_j$ to compute $\frac{\partial L}{\partial Y_{j-1,k}}$ and send it to upstream neighbor $Y_{j-1}$

The explanation above differs slightly from that given in the lecture notes in that this version assumes use of the delta Jacobian and simplifications incurred from it, whereas the lecture notes only mention those simplifications near the end of the slide set.

Although many of the backward pass activities can be achieved with numpy’s tensor operations, such as dot, outer, and einsum, this is not quite as straightforward as with the vanilla backpropagation algorithm. However, the need for storing a complete activation history during the forward pass and then reusing those values during the backward pass does not greatly complicate the overall computational flow, much of which can still be achieved with combinations of matrix operations. In general, it is much easier to realize backpropagation with a majority of numpy matrix operations in recurrent nets (of the Elman style) than to do so with convolution nets.

5 Data Generation

This project requires a different type of data generator than was developed for vanilla neural networks. Here, each data case stems from a sequence that displays various relationships between its elements. Your data generator must first produce sequences and then form cases from pairs of elements.
For example, the individual elements may consist of bit vectors, while the sequence would consist of vectors shifted by a certain amount and in a certain direction. Figure 3 illustrates this concept with two different sequences. You are free to choose the types of patterns, but the sequences must include some critical relationship between the patterns, just as words in sentences have tight interdependencies. You are also free to use dimensions other than 1 and primitives other than binary values, but those alternative choices can greatly increase the difficulty of the task. Binary vectors work well for this assignment.

Furthermore, you need to carefully consider your sequences to insure that some aspect of history actually matters in predicting the next element of a sequence. Otherwise, a vanilla neural network can handle the job. This is best illustrated by sequence A in Figure 3. Consider the 2nd element in that sequence, which is the input of the second case extracted from that sequence. If your sequence generator only and always uses the rule shift right one, then the neural network would learn that this second pattern is always followed by the third case, regardless of what came before the second pattern. In short, the network would not need historical information to determine the next pattern: it would only need the most recent pattern (i.e., the task becomes Markovian).

By simply adding the possibility of having \( n_r \) possible shift rules (that your system randomly chooses among each time it generates a new sequence), the prediction task becomes dependent upon (at least a little) history: to understand how to convert pattern 2 into pattern 3, the network needs to see how pattern 1 was converted into pattern 2. And further along in the sequence, it may be easier for the net to make good predictions if it sees several of the earlier transitions.

The mandatory criteria for the sequence generator are the following:

1. Sequences must not be purely Markovian: the pattern P1 should not always map to P2 across the entire dataset. You need not prove this by analyzing complete datasets, but the principles of sequence generation should clearly enable non-Markovian results.
2. Patterns must contain anywhere from 5 to 20 values (e.g. bits).
3. Sequences must be anywhere from 3 to 20 in length.
4. Datasets must be based on anywhere from 10 to 500 sequences.
5. Employ at least 4 different transition rules when generating any single dataset, but, of course, only use a single rule during the generation of a particular sequence: don’t change rules in the middle of a sequence.

You are free to introduce noise into sequence data, but it is not a requirement of the assignment and should only be done if you are framing the problem as one of regression (i.e. the network predicts the output pattern as opposed to predicting an output class). If you decide to treat this as a classification problem, then adding noise can easily cause major problems.

### 6 Network Architecture Configuration Files

You will extend your format for and parser of configuration files so as to include recurrent layers. See the *Pivotal Parameters* section for more details.
Figure 3: An easy approach to creating data sets for a recurrent network involves a) generating a random pattern, b) shifting it repeatedly using one rule, and c) pairing neighbor vectors to produce cases. Here, each pattern is a 1-dimensional vector of length 10, where filled cells represent 1’s and unfilled cells 0’s. The shift rule for sequence A is \textit{right one}; for sequence B, the rule is \textit{left two}.

6.1 Pivotal Parameters

Each of the following parameters must be specifiable in your architecture configuration files and supported by your deep learning system.

1. The number of sequences that the data-generator shall produce, along with the length of each sequence and the length of each vector in the sequence.

2. The number of layers in the network. This may be indirectly specified by listing the layers themselves. Your system must handle anywhere from 0 to 5 hidden layers.

3. The type of a layer, whether standard (dense) or recurrent.

4. The number of neurons in each layer, including the input and output layers. Sizes from 1 to 100 must be possible.

5. The activation function used in each layer (except the input layer). The options must include: sigmoid, tanh, relu, and linear.

6. If you decide to implement this as a classification network (as opposed to regression), then softmax must be a possible option for the output layer.

7. The loss function. You must include mean-squared error (MSE) if treating sequence-prediction as a regression problem; otherwise, if you treat it as a classification problem, then cross-entropy must be implemented.

8. Initial weight ranges for each (non-input) layer. You may (or may not) support use of the glorot initializer; it is a very effective approach.

9. The dataset to be run. You may choose to pre-generate datasets and simply include the file name in the configuration file, or you may include all of the parameters for the data generator in the configuration file. You only need to provide one of these options to the user, but you must show the generation of data cases during the demonstration session.
You do not need to implement weight regularization for this project.

7 Visualization

Your system must be able to visualize sequences, but it need not display individual input-output pairs derived from the sequence. Feel free to reuse the visualization tools from your earlier project. To display a series of 1-D vectors, simply concatenate them horizontally and use your 2-D visualizer.

Learning progress, as in the earlier project, must also be plotted.

8 Deliverables

The subsystems and procedures listed below will be demonstrated by running the system multiple times and by discussing your code with an instructor or assistant. You must come to the demo session with a few fully-functioning configuration files; the reviewer will then modify these files in various ways and run them through your system. You will assist the reviewer in these modifications in accordance with the legal syntax of your configuration files and any other constraints of your system. However, these constraints should not exclude any of the requirements listed in this document, many of which the reviewer will explore in your system.

If your data-generating system takes several minutes to produce a few hundred sequences, then you should produce and save to file at least one large (200 or more sequences) dataset for use during the demo session. Regardless, you will be asked to demonstrate your data generator during the demonstration session, even if that only involves generating (and displaying) a few sequences.

The point breakdown is as follows:

1. The data generator, image viewer and graphic display of learning progress. (3 points).

2. The (verbal) description of the format for configuration files, along with code to parse those files. (1 point).

3. The forward pass of the deep network, with (some of) the core functions performed with matrix and vector operations in numpy (or a similar package). Our evaluation process includes analysis of your source code and running a few different networks (chosen by the evaluator) for a few minibatches and printing out all of the following: network inputs, network outputs, target values, and error/loss. Be sure that your system has a **verbose** flag that, when True, allows all of this information to be printed out to the command line. (8 points).

4. The backward pass (backpropagation) for modifying the network weights and biases. This must clearly perform the explicit computation of the key Jacobian matrices and vectors, as specified in the lecture notes. Our evaluation process includes analysis of your source code along with running several different networks on several different data sets (many chosen by the evaluator) for many minibatches (depending upon your machine’s computational capacity) and observing learning progress. The actual performance (i.e. error/loss) on the validation and test data sets will not affect your point score, as long as the system runs properly (i.e. no crashes or obvious mistakes) on the validation and test sets, and displays their results. (13 points).
8.1 The Demonstration Session

To insure a smooth demonstration, you should have the following readily available during your session:

1. A configuration file for a network with a) one recurrent layer and at least one dense layer (not including the input layer) b) that runs on a dataset of at least 100 training sequences of length at least 8 with individual vector elements of length at least 10, and c) that has previously shown some learning progress (i.e. loss clearly declines over time / minibatches). All parameters of this network should be tuned to those that have worked well in the past.

2. A configuration file for a network with at least three recurrent layers and one non-input dense layer that runs on the dataset described above and that shows some learning progress. All parameters of this network should be tuned to those that have worked well in the past.

It should be very easy to manipulate these configuration files such that all or some of your recurrent layers can be converted to dense layers, thus enabling a straightforward comparison between recurrent and non-recurrent networks run on the same or similar datasets. Demonstration sessions typically last 20-30 minutes.

9 Important Practical Details

The code for networks, layers and the data generator must be object-oriented. Failure to satisfy this simple criteria will result in considerable point loss.

WARNING: Failure to properly explain ANY portion of your code (or to convince the reviewer that you wrote the code) can result in the loss of 10 to 25 points, depending upon the seriousness of the situation. This is an individual exercise in programming, not in downloading nor copying.

A zip file containing your commented code must be uploaded to BLACKBOARD before or (immediately) after your demonstration. You will not get explicit credit for the code, but it is crucial that we have the code online in the event that you decide to register a formal complaint about your grade (for the entire course).

The 25 total points for this project are 25 of the 100 points that are available for the entire semester.