Programming Assignment 3: Recurrent Neural Network Autograd

Due: Thursday 10/23/2025 @11:59pm EST

Disclaimer

These questions are **implementation** question in the Python programming language. In this assignment you will be implementing a symbolic differentiation system that supports Recurrent neural network layers. By the end of this assignment, you will be able to build a sequential neural network comprised of Recurrent layers, Time Distributed layers along with the layers developed in previous assignments. This package will build upon pa2 to add more functionality to our growing autograd capabilities.

Note that you are **NOT** allowed to use any help from LLMs, online solutions, old solutions, etc. when solving these problems. Your solutions are your own. You **are** allowed to chat with your classmates, but not with detail granular enough to copy each others work.

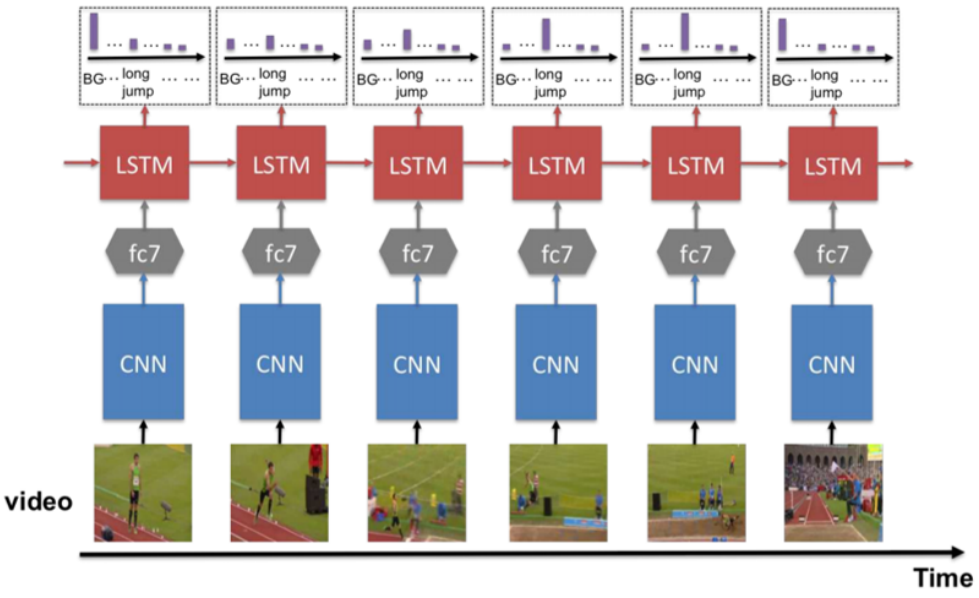
Note that I have organized the modules you will add to your neural network package roughly in order of difficulty (at least to me). Good luck, you can do it! One thing to note here is that the data our Recurrent layers expect has a specific structure. The first dimension of batched data is the batch size, and the second is the sequence size. Any extra dimension after the first two specify the dimensionality of a specific element within a sequence within a batch. Recurrent layers also typically *encapsulate* other layers. This is a different paradigm than what you've probably experienced so far where your layers directly store `Parameter` objects. Recurrent layers on the other hand typically store `Module` objects and then wield those `Modules` to compute their `forward` and `backward` methods. You'll see.

Task 1: Time Distributed Layer (10 points)

In this assignment, all batched data will be sequential. This will take the form of numpy arrays with shape `(num_examples, sequence_size, ...)`. A *Time Distributed* Layer is a way to take layers that are not intended to process sequential data (e.g. convolutional layers, pooling layers, feed-forward layers, activations, etc.) and apply them to each element of the sequence independently (i.e. "distribute" them across time). For example, the code snippet

```
t1 = TimeDistributed(Dense(20, 40))
t2 = TimeDistributed(Conv2d(...)) # ... is a placeholder for the arguments
```

creates two `TimeDistributed` objects. Object `t1` will apply a `Dense` layer to every element of an input sequence independently, while `t2` will apply a `Conv2d` layer to every element of an input sequence. Object `t1` can process batches of data that has shape `(num_examples, seq_size, 20)` while `t2` can process batches of data that has shape `(num_examples, seq_size, img_height, img_width, num_channels)`. For instance, the following example network from lecture (shown below) uses a Time Distributed layer to apply a CNN (and a subsequent fully connected layer) to every frame of a video:



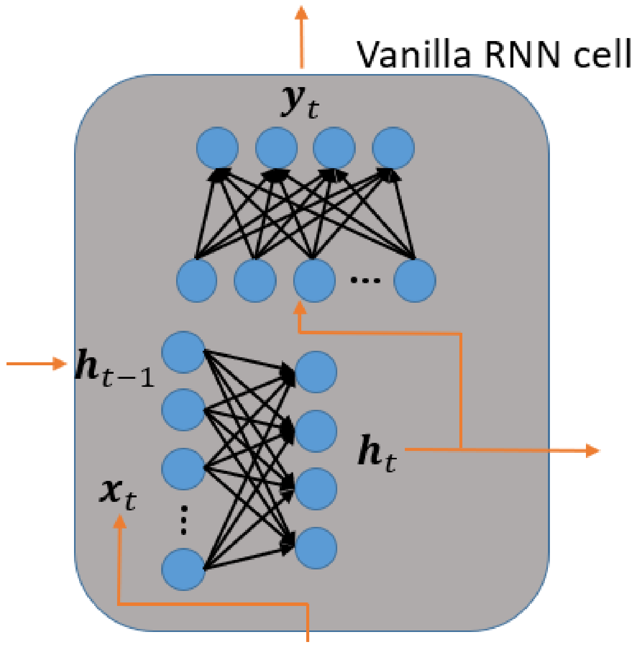
Please create the class `TimeDistributed` in the file `autograd/nn/layers/timedistributed.py`. This class extends the `Module` class like all other layers and only takes a single argument in its constructor `module: Module`. Please save this argument to a field of the same name. This `module` field contains the `Module` that we wish to "distribute" across time (e.g. apply independently to each element of a sequence). Your `forward` method should do just that: given a numpy array where the first dimension is the batch dimension and the second dimension is the sequence size, apply `self.module` to each element of the sequence independently and return the outputs. You might find it convenient to collapse the first and second dimensions into a single dimension using a `reshape` (and then you would have to reshape the output accordingly), or you can iterate over the sequence if you want. The `backward` method is just as straightforward: given the same batched data and `dLoss_dModule` (which has the same dimensions as the output of your `forward` method), use `self.module` to calculate `dLoss_dX` for each element of the sequence independently by calling it's `backward` method. You are again welcome to do some fancy reshaping to implement this or you can do some looping instead.

Don't forget to implement the `parameters` method! The parameters of any `TimeDistributed` layer are just the parameters of the `Module` that is being distributed across time!

Task 2: Vanilla RNN Cell (50 points)

Before we can implement a generic Recurrent Neural Network (RNN), we'll need to first define what cell the RNN is using. A RNN is just a way of *using* a RNN *Cell* to process a sequence: if we change the cell we change the type of RNN. Think of an RNN as a wrapper around a cell: the RNN is responsible for using the cell to process a sequence while the cell itself is responsible for how elements are related to each other. In this assignment we will only be implementing a "Vanilla" RNN Cell but if you want there are many other flavors of RNN cells that exist (such as LSTM cells, GRU cells, etc.). You are welcome to implement more if you want: the software design choices I have made here will hopefully make it easier to implement as many cell flavors as you want!

A Vanill RNN cells looks like this internally:



In lecture we defined a Vanilla RNN cell as first calculating the new hidden state as a sum of the output of two feed-forward layers: one feed-forward layer transforms the input vector into the hidden state vector space, while the other feed-forward layer transforms the previous hidden state. But remember! The sum of feed-forward neural networks is itself a (bigger) feed-forward neural network with concatenated input. So what we have drawn here is the concatenated neural network instead of two discrete networks. This should make your code easier to implement.

Please create class `VanillaRNNCell` in file `autograd/nn/layers/cells/vanilla_rnn_cell.py`. This class should extend `RNNCell` which is in `autograd/nn/layers/cells/rnn_cell.py`. A "Vanilla" RNN cell, as shown above, contains two `Dense` layers along with two activation layers. A Vanilla RNN Cell takes two vectors as input: the previous state which has some *hidden* dimensionality (which is a hyperparameter), and an element from a sequence. These two vectors are first concatenated together and then passed through the first `Dense` layer and activation which produces an updated state. This new state can be used by the RNN Cell to process future sequence elements. The updated state is then passed through the second `Dense` layer and activation which produces a prediction vector (of some other dimensionality) which is the predictions for the sequence up to and including the input.

To help you with this, I have included my versions of `autograd/nn/layers/dense.py` along with other activations you implemented in `pa1`. Feel free to use my versions if you weren't able to get yours working. This class will take some arguments in its constructor which are listed in order:

1. `in_dim: int`. This argument specifies how many features an element of a sequence has (which will be a vector).
2. `hidden_dim: int`. This argument specifies how many features the "state" of this RNN Cell has. This is also the output dimensionality of the first `Dense` layer which projects from `in_dim + hidden_dim` to `hidden_dim`.
3. `out_dim: int`. This argument specifies how many features output vectors from this cell has. This is also the output dimensionality of the second `Dense` layer which projects from `hidden_dim` to `out_dim`.
4. `hidden_activation: Module = None`. This argument specifies the activation function that the first `Dense` layer will use. If `None` this should default to `Tanh`.
5. `output_activation: Module = None`. This argument specifies the activation function that the second `Dense` layer will use. If `None` this should default to `Sigmoid`.

Please store these arguments in fields with the same name as the argument. You will also want to create a field for the first and second `Dense` layers as well.

As a consequence of inheriting from class `RNNCell`, there is a new method that all `RNNCell`s must implement which is called `init_states(batch_size: int) -> np.ndarray`. The purpose of this method is to provide the initial value of all states that a `RNNCell` uses when processing the first element of a sequence. By convention, the values of initial states should be zero. Since you're implementing a Vanilla RNN Cell, which only has a single state, you will want to return a numpy array of shape `(batch_size, self.hidden_dim)` populated with zeros.

Now implement the `forward` method, which looks slightly different now. The new signature is this:

```
def forward(self: VanillaRNNCell,
            H_t_minus_1: np.ndarray,
            X_t: np.ndarray) -> Tuple[np.ndarray, np.ndarray]
```

The first argument `H_t_minus_1` is the current state of the RNN Cell and has shape `(batch_size, self.hidden_dim)`. The second argument `X_t` is the current element of a (batched) sequence and has shape `(batch_size, self.in_dim)`. Your cell should implement the Vanilla RNN forward pass:

1. $\mathbf{Z}_t = f_1([\mathbf{H}_{t-1}, \mathbf{X}_t]; \theta)$ where $[\mathbf{H}_{t-1}, \mathbf{X}_t]$ is concatenating along each row and f_1 is a feed-forward neural network with parameter θ.
2. $\mathbf{H}_t = g(\mathbf{Z}_t)$ where g is an activation function and $\mathbf{H}_t$ is the new batch of states.
3. $\mathbf{R}_t = f_2(\mathbf{H}_t; \phi)$ where f_2 is a feed-forward neural network with parameters ϕ.
4. $\mathbf{A}_t = o(\mathbf{R}_t)$ where o is an activation function and $\mathbf{A}_t$ is a batch of predictions.

Now implement the `backward` method, which also looks slightly different now. The new signature is this:

```
def backward(self: VanillaRNNCell,
             H_t_minus_1: np.ndarray,
             X_t: np.ndarray,
             dLoss_dModule_t: np.ndarray,
             dLoss_dStates_t: np.ndarray) -> Tuple[np.ndarray, np.ndarray]
```

The first two arguments are just like previous `backward` methods: they're whatever data you need to do the forward pass. What is different is that there are now **two** "downstream" gradients instead of just a single `dLoss_dModule`. This is because RNN Cells have potentially two ways that their computations are connected to the loss function:

1. The predictions of this input $\mathbf{A}_t$ from the forward pass could have been used by the loss function. If it was, the term `dLoss_dModule_t` will not be `None`. If the predictions used for this element of a sequence were not used to compute the loss then this value will be `None`.

2. The updated state $\mathbf{H}_t$ from the forward pass could have been used by future RNN Cell computation or something and eventually contributed to the loss. If it was, the term `dLoss_dStates_t` will not be `None`. If $\mathbf{H}_t$ was not used downstream to contribute to the loss then this term will be `None`.

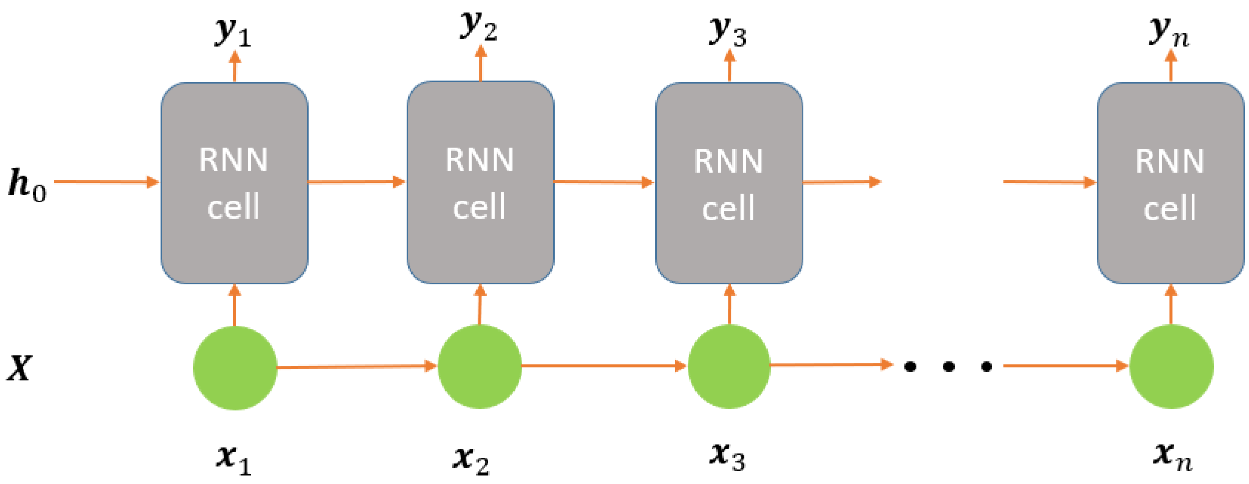
This is probably your first time dealing with a **convergence** of gradients. The two arrows represented by `dLoss_dModule_t` and `dLoss_dStates_t` converge (e.g. intersect) at the value $\mathbf{H}_t$. If `dLoss_dModule_t` is not `None`, then you need to first pass it through the last two steps of the pipeline in the `forward` method until you get to `dLoss_dH_t`. If `dLoss_dStates_t` is not `None`, this also provides a value for `dLoss_dH_t`, so you should add the two if both terms are present, and only use one of them if only one is present. You can then pass `dLoss_dH_t` through the first two steps of the pipeline in the `forward` method to calculate `dLoss_dH_t_minus_1` and `dLoss_dX_t` which are the two values to return in that order. Note that by storing `Dense` objects and other `Module` objects as fields inside your `VanillaRNNCell`, any parameters internal to those objects are getting their gradients set (thanks encapsulation)!

Don't forget to implement the `parameters` method! The parameters of a `VanillaRNNCell` are the parameters of the four `Module` objects contained within. Assuming you call the first `Dense` module `self.hidden_dense` and the second `Dense` module `self.out_dense`, return a list of parameters in the following order:

```
self.hidden_dense.params + self.hidden_activation.params +\
self.out_dense.params + self.out_activation.params
```

Task 3: Recurrent Neural Network Layer (40 points)

Now that the cell is done, we can finally create a Recurrent Neural Network (RNN) layer. Remember, a RNN is just a wrapper around a cell: the cell does the actual computation while the RNN just applies it. In other words, then entire purpose of an RNN is just to perform the unrolling of the cell across data:



Whatever the cell is is what the RNN unrolls. Different cells make different design decisions (for instance an LSTM cell has two hidden states to keep track of). The job of the RNN is just to manage the cell as it is unrolled. Formally, this means the job of the RNN is to initialize hidden states, and unroll the cell across data to compute predictions in the forward pass, and to pass gradients together between cells in the unrolled graph in the backward pass.

Please create class `RNN` in file `autograd/nn/layers/rnn.py`. This class will take some arguments in its constructor which are listed in order:

1. `cell: RNNCell`. This argument provides an instance of a `RNNCell` that this `RNN` should wrap around and use internally.
2. `return_sequences: bool = False`. If `True`, the output of `RNN.forward` will be a sequence: outputs will be returned from the processing of every element of the sequence. If `False`, `RNN.forward` will only produce output from the processing of the last element of the sequence.
3. `return_states: bool = False`. If `True`, `RNN.forward` will produce the states along with the predictions. If `False`, `RNN.forward` will produce only the predictions and not the states.
4. `backprop_through_time_limit: int = None`. If not `None`, this will specify how far to backprop gradients through time. If `None`, use a value of `np.inf` instead.

Your constructor should create fields for all of these quanties and set them appropriately.

Now create method `init_states(batch_size: int) -> np.ndarray`. This method is just an interface and should wrap around the `init_states` of the cell the RNN is using.

Now, finally, create the `forward` method. This method also looks slightly different than in the past. It now should have signature:

```
def forward(self: RNN,
            X: np.ndarray, # a sequence (batch_size, seq_size, ...)
            states_init: np.ndarray = None
            ) -> Union[np.ndarray, Tuple[np.ndarray, np.ndarray]]
```

The first argument `X` is just like normal: its a batched input. The first dimension is the number of examples in a batch, the second dimension is the size of the sequence, and the rest of the dimensions are whatever dimensionality the cell the RNN is using expects. If you're using `VanillaRNNCell` then there should only be a third dimension with a predetermined size. If you eventually expand this library with new cells, you can design your cell to process whatever dimensioned data you want.

The last argument `states_init` is where the inital states for the cell are provided. If this is `None` then you should call your `init_states` method. If they are provided, then use those initial states instead. The `forward` method should apply your cell to each element of the sequence. If `self.return_states` is `True` then you will want to return the states as well as the predictions from each cell, otherwise just the predictions. This combines with `self.return_sequences` to determine if the returned values should be sequences or not. If `self.return_sequences` is `True` then your method should return sequences (either the pair of (state, prediction) sequences or just the sequence of predictions). If `self.return_sequences` is `False` then you should only return the output from processing the last element of the sequence (either a pair of (state, prediction) outputs or just the prediction output).

Now implement the `backward` method. This method also has a slightly different signature:

```
def backward(self: RNN,
             X: np.ndarray,
             dLoss_dModule: np.ndarray,
             states_init: np.ndarray = None) -> np.ndarray:
```

Most of these arguments are as expected, and `states_init` works just like it did in the `forward` method. One thing to watch out for though is the shape of `dLoss_dModule`. If `self.return_sequences` is `True` then `dLoss_dModule` will be a sequence as well. If `self.return_sequences` is `False` then `dLoss_dModule` will be only the error flowing into the predictions of the last element of the input sequence. Your backward method should do a forward pass like normal to cache all intermediary values, and then it should start at the end of the sequence. Working your way from the back of the sequence to the front of the sequence, you need to propagate the `dLoss_dModule` error for each output back through the RNN. For instance, the last output was calculated from the last element of the sequence, which was calculated from the second to last element of the sequence, which was calculated from the third to last element of the sequence, etc. You need to pass this gradient all the way back to the front of the sequence (if the value of `self.backprop_through_time_limit` is `np.inf`, otherwise stop according to the limit). Then you can start with the second to last output and propagate it backwards, then propagate the third to last output, etc. This is called "unrolling" a RNN and this flavor of backpropagation is called **backpropagation through time**. Your method needs to calculate and return `dLoss_dX` which will have the same shape as `X`.

Don't forget to implement the `parameters` method! The only parameters of a `RNN` exist within the cell that it is using!

Task 6: Testing Your Solution

When you are ready to test your solution, you will have to make some testing files to individually test each piece of functionality that you have added to your package. I have provided two testing scripts which tests all of them together, so I would not recommend running this until you are confident in each piece's correctness first. The testing scripts I have provided are called `test_rnn_single_output.py` and `test_rnn_sequence_output.py` and are located in `autograd/`. Good luck!

Task 7: Submitting Your Assignment

Please turn in all of the python files you created for the package! Please note that only the files you were told to create in the instructions above will be moved to the testing area by the autograder, so if you create any additional files they won't be moved to where they need to be by the autograder! You should be able to turn in these files by dragging and dropping them in your web browser.