Homework 3 Part 1
THE RISE OF RNN!

11-785: Introduction to Deep Learning (Fall 2020)

OUT: **October 18 2020, 12:00 AM EST**
DUE: **November 8 2020, 11:59 PM EST**
(Last Updated: October 17 2020, 11:45PM EST)

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with / work with other students on homework assignments
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.
- **Overview:**
  - **Introduction**
  - **Running/Submitting Code**
  - **RNN**
  - **GRU**
  - **Appendix**
- **Directions:**
  - You are required to do this assignment in the Python (version 3) programming language. Do not use any auto-differentiation toolboxes (PyTorch, TensorFlow, Keras, etc) - you are only permitted and recommended to vectorize your computation using the Numpy library.
  - We recommend that you look through all of the problems before attempting the first problem. However we do recommend you complete the problems in order, as the difficulty increases, and questions often rely on the completion of previous questions.
  - If you haven't done so, use pdb to debug your code effectively.

Introduction

In this assignment, you will continue to develop your own version of PyTorch, which is of course called MyTorch (still a brilliant name; a master stroke. Well done!).

Homework Structure

Below is a list of files that are **directly relevant** to hw3.

IMPORTANT: First, copy the highlighted files/folders from the HW3P1 handout over to the corresponding folders that you used in hw1 and hw2.

NOTE: We recommend you make a backup of your hw3 files before copying everything over, just in case you break code or want to revert back to an earlier version.

```
handout
├── autograder
│   ├── hw3_autograder .......................................... Copy over this entire folder
│   │   ├── runner.py
│   │   ├── test_rnn.py
│   │   ├── test_util.py
│   │   ├── test_gru.py
│   │   └── test_functional.py
│   └── helpers.py
├── mytorch ........................................................... MyTorch library
│   ├── nn ........................................................ Neural Net-related files
│   │   ├── activations.py
│   │   ├── gru.py ........................................................... GRU objects
│   │   ├── rnn.py ........................................................... RNN objects
│   │   ├── util.py ...................................................... pack_sequence etc
│   │   ├── conv.py
│   │   ├── functional.py
│   │   ├── linear.py
│   │   └── sequential.py
│   └── tensor.py
├── hw3
│   └── stubs.py
├── sandbox.py
├── create_tarball.sh
└── grade.sh
```

Next, copy and paste the following code stubs from hw3/stubs.py into the correct files.

1. Copy Slice(Function) into nn/functional.py.
2. Copy Cat(Function) into nn/functional.py.
3. Copy Unsqueeze(Function) into nn/functional.py.

0.1 Running/Submitting Code

This section covers how to test code locally and how to create the final submission.

0.1.1 Running Local Autograder

Run the command below to calculate scores and test your code locally.

```
./grade.sh 3
```

If this doesn't work, converting line-endings may help:

```
sudo apt install dos2unix
dos2unix grade.sh
./grade.sh 3
```

If all else fails, you can run the autograder manually with this:

```
python3 ./autograder/hw3_autograder/runner.py
```

0.1.2 Running the Sandbox

We've provided sandbox.py: a script to test and easily debug basic operations and autograd.

Note: We will not provide new sandbox methods for this homework. You are required to write your own from now onwards.

```
python3 sandbox.py
```

0.1.3 Submitting to Autolab

Note: You can submit to Autolab even if you're not finished yet. You should do this early and often, as it guarantees you a minimum grade and helps avoid last-minute problems with Autolab.

Run this script to gather the needed files into a handin.tar file:

```
./create_tarball.sh
```

If this crashes (with some message about a hw4 folder) use dos2unix on this file too.

You can now upload handin.tar to Autolab .

1 RNN (Total 80 points)

In `mytorch/nn/rnn.py` we will implement a full-fledged Recurrent Neural Network module with the ability to handle variable length inputs in the same batch.

1.1 RNN Unit (15 points)

Follow the starter code available in `mytorch/nn/rnn.py` to get a better sense of the various attributes of the RNNUnit.

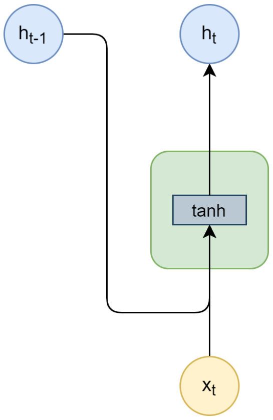


Figure 1: The computation flow for the RNN Unit forward.

$$h'_{t,l} = tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1,l} + b_{hh}) \tag{1}$$

The equation you should follow is given in equation 1.

You are required to implement the forward method of RNNUnit module. The description of the inputs and expected outputs are specified below:

Inputs

- x (effective_batch_size, input_size)
  - Input at the current time step.
  - NOTE: For now interpret effective_batch_size same as regular batch_size. The difference will become apparent later in the homework.
- h (effective_batch_size, hidden_size)
  - Hidden state generated by the previous time step, h_{t-1}

Outputs

- h_prime: (effective_batch_size, hidden_size)
  - New hidden state generated at the current time step, h'_t

NOTE: As a matter of convention, if you apply certain reshape/transpose operations while using $self.weight\_ih$ then please do the same for $self.weight\_hh$. This is important to note because $self.weight\_hh$ is symmetric and is therefore exposed to multiple interpretations on how to use it.

1.2 Detour 1: Cat, Slice, Unsqueeze (15 points)

In the following section we implement some necessary functions which will prove to be extremely handy while completing the rest of the homework. These are namely: Cat, Slice and Unsqueeze.

1.2.1 Cat (7 points)

Concatenates the given sequence of tensors in the given dimension. All tensors must either have the same shape (except in the concatenating dimension) or be empty. Please refer to the PyTorch Documentation for better understanding. If you are not into documentations then please refer back to Recitation 0 where this was covered as well.

First implement the corresponding Cat class in `mytorch/nn/functional.py`. This will be a subclass of Function class.

Next implement a helper function *cat* in the `mytorch/tensor.py` to call the concatenation operation on a list of sequences. This should in turn correctly call the corresponding Function sub-class. Below is a description of the required inputs and outputs.

Inputs

- seq: list of tensors
  - The list basically contains the sequences we want to concatenate
- dim: (int, default=0)
  - The dimension along which we are supposed to concatenate the tensors in the list seq

Outputs

- Tensor:
  - The concatenated tensor

NOTE: You don't need to add anything to the Tensor class in `mytorch/tensor.py` with respect to Cat operation.

1.2.2 Slice (5 points)

Despite being worth only 5 points, implementing this operation takes your Tensor class to a while new level. With this operation up and running, you can index your Tensor class just like you index Numpy arrays/PyTorch tensors, while autograd takes charge of calculating the required gradients (assuming you implemented the backward correctly).

First implement the corresponding Slice class in `mytorch/nn/functional.py`. This will be a subclass of Function class.

Next add the *__getitem__* method in your Tensor class `mytorch_tensor.py`, which in turn calls the Slice function sub-class.

HINT
The implementation of a slicing operation from scratch may appear to be a daunting task but we will employ a cool trick to get this done in just a few lines of code. Whenever we try to slice a segment of a given tensor using the [] notation, python creates a *key* (depending on what you provide within []) and passes this to the *__getitem__* function. This *key* is used by the class to provide the appropriate result to the user. The *key* can be either an integer, a python slice object, a list etc depending on how we slice our object. This is the principle implemented in Numpy for slicing ndarrays. For the purpose of our implementation we will try to leverage this fact to make our task easy. Once we create a *__getitem__* method for our tensor class, everytime we slice our tensors the *__getitem__* method will be invoked with the appropriate *key*. Given that the intention of slicing on our tensor is to get the appropriate segment of the underlying ndarray (data attribute) as a tensor object, can you use this *key* to complete the task.

1.2.3 Unsqueeze (3 points)

Returns a new tensor with a dimension of size one inserted at the specified position. Please refer to the PyTorch Documentation for better understanding. If you are not into documentations then please refer back to Recitation 0 where this was covered as well.

Add the *unsqueeze* method in your Tensor class `mytorch_tensor.py`.

HINT: Why is this only worth 3 points? Well if you look closely, you might be able to use a function you implemented previously without any extra effort. The name of the function you might need begins with 'R' in numpy.

1.3 Detour 2: pack_sequence, unpack_sequence (30 points)

What's all the fuss about handling variable length samples in a batch?
This section is devoted to the construction of objects of class *PackedSequence* which help us create batches in the context of RNNs.

As you might have guessed by now, handling of variable length sequences in a batch requires way more machinery than what exists in a simple RNN. In this section we will create all the utility functions that will help us with the handling of variable length sequence in a single batch.

Before we proceed let's understand an important class that packages our packed sequences in a form which makes it easier for us to deal with them. This will also help you better understand why handling variable length sequences in a batch can be a tedious task.

Refer to Appendix for more details.

Please use the Tensor Slicing and Cat operations you implemented in the previous sections while implementing these functions.

1.3.1 pack_sequence (15 points)

In `mytorch/nn/util.py` implement the method `pack_sequence`.

This function as the name suggests, packs a list of variable length Tensors into an object of class PackedSequence. A list of tensors is given (as shown in Figure 2) which are to be converted into a PackedSequence object (detailed class description below). This object holds the packed 2d tensor which is later fed into the RNN module for training and testing. Refer to Figures 2, 3 and 4 to better understand what pack_sequence does.

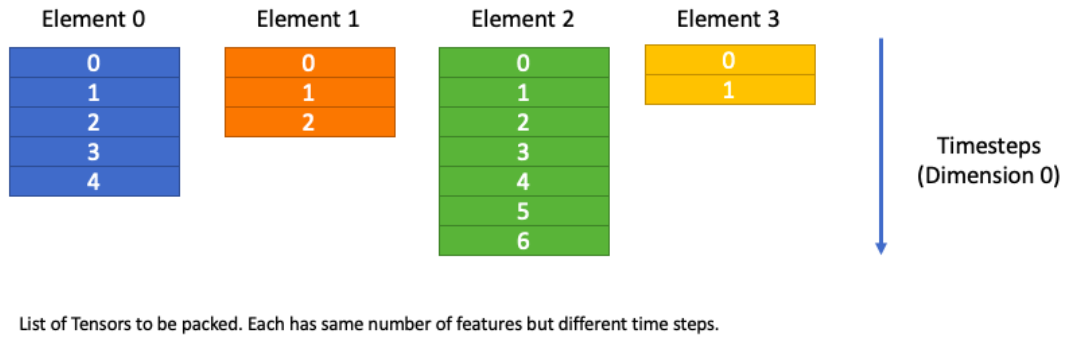

Element 0
0
1
2
3
4
Element 1
0
1
2
Element 2
0
1
2
3
4
5
6
Element 3
0
1
Timesteps
(Dimension 0)
List of Tensors to be packed. Each has same number of features but different time steps.

Figure 2: List of tensors we want to pack

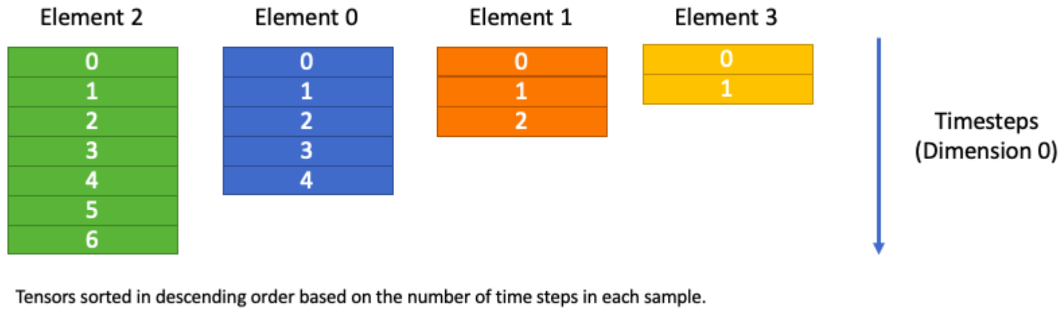

Element 2
0
1
2
3
4
5
6
Element 0
0
1
2
3
4
Element 1
0
1
2
Element 3
0
1
Timesteps
(Dimension 0)
Tensors sorted in descending order based on the number of time steps in each sample.

Figure 3: First we sort the list in a descending order based on number of timesteps in each

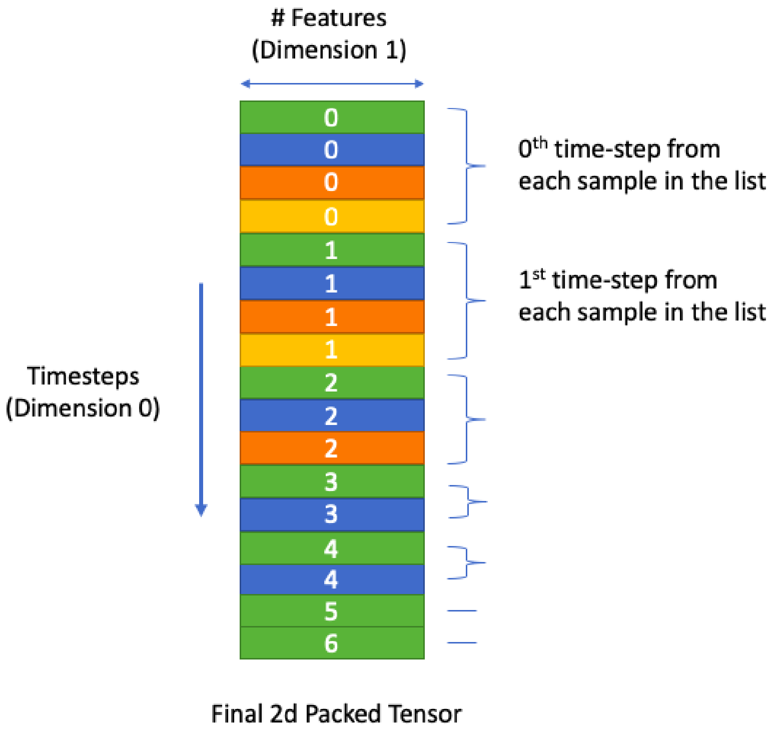


Figure 4: Final Packed 2d Tensor

Class: PackedSequence
Attributes

- data
  - The actual packed tensor. In the context of this homework this is a 2D tensor. Refer to 4
- sorted_indices
  - 1d ndarray (numpy n-dimensional array) of integers containing the indices of elements in the original list when they are sorted based on number of time steps in each sample. Refer to Figure 2 and Figure 3
- sorted_indices
  - 1d ndarray of integers where ith element represents the number of samples in the original list having atleast (i+1) timesteps. Refer to Figure 4 for more clarity.

Function: pack_sequence
Inputs

- seq: list of tensors
  - The list contains tensors representing individual instances, each having a variable length.

Outputs

- PackedSequence:
  - PackedSequence

1.3.2 unpack_sequence (15 points)

In `mytorch/nn/util.py` implement the method `unpack_sequence`.

This function unpacks a given PackedSequence object into a list of tensors that were used to create it.

Function: unpack_sequence
Inputs

- ps: PackedSequence

Outputs

- list of Tensors:

1.4 TimeIterator: (20 points)

In `mytorch/nn/rnn.py` implement the forward method for the `TimeIterator`.

For a given input this class iterates through time by processing the entire sequence of timesteps. Can be thought to represent a single layer for a given basic unit (interpret this as RNNUnit for now) which is applied at each time step.

Working of forward method in TimeIterator

- This module's forward method is tasked with receiving a batch of samples packed in the PackedSequence form.
- The method runs the section of the input corresponding to a single time-step across the batches through an RNNUnit.
- The hidden state returned by the RNNUnit is collected and then the section of the input corresponding to the next time-step across the batches are fed through the RNNUnit along with the previously (last-time step) collected hidden state to generate a new hidden-state.
- This process is done iteratively till all time-steps are exhausted for each sample in the batch.
- Follow Figures 5, 6, 7 and 8 to get a better understanding of how TimeIterator processes and given input PackedSequence.

The inputs and outputs of the forward method are given below for the

Class: PackedSequence
Forward Method
Inputs

- input: PackedSequence
- hidden: Tensor (batch_size,hidden_size)

Outputs

- PackedSequence: hiddens at each timestep for each sample packaged as a packed sequence
- Tensor (batch_size,hidden_size): This is the hidden generated by the last time step for each sample joined together

Finally in `mytorch/nn/rnn.py` create a class *RNN* as a sub-class of TimeIterator which instantiates the base class with basic_unit = RNNUnit. Please look at the code for more details.

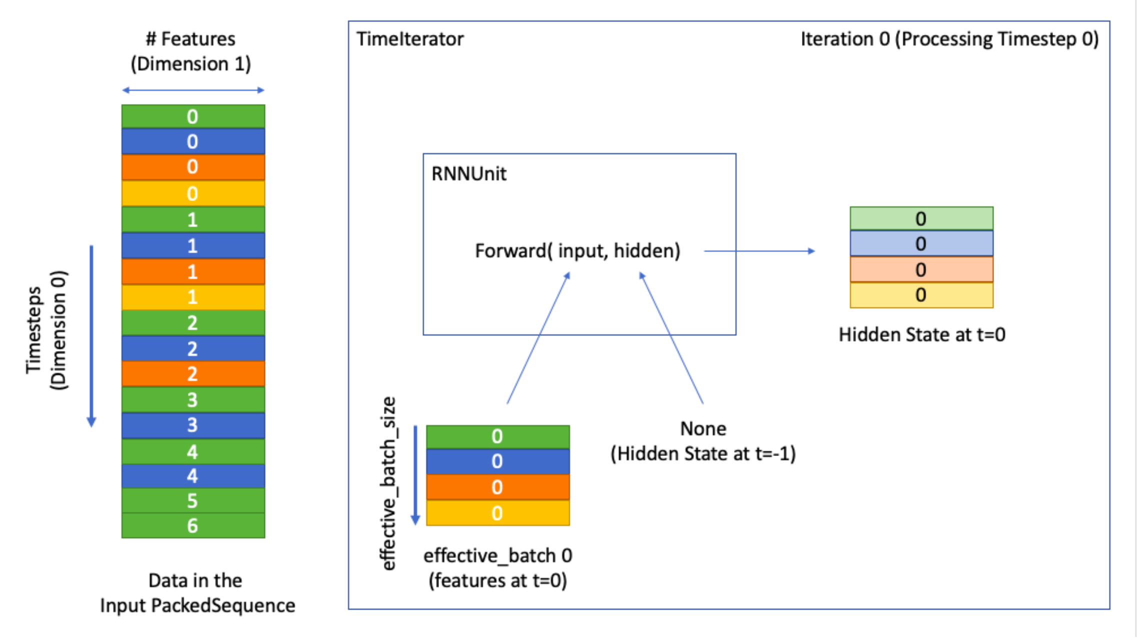


Figure 5: Iteration 0 for the TimeIterator

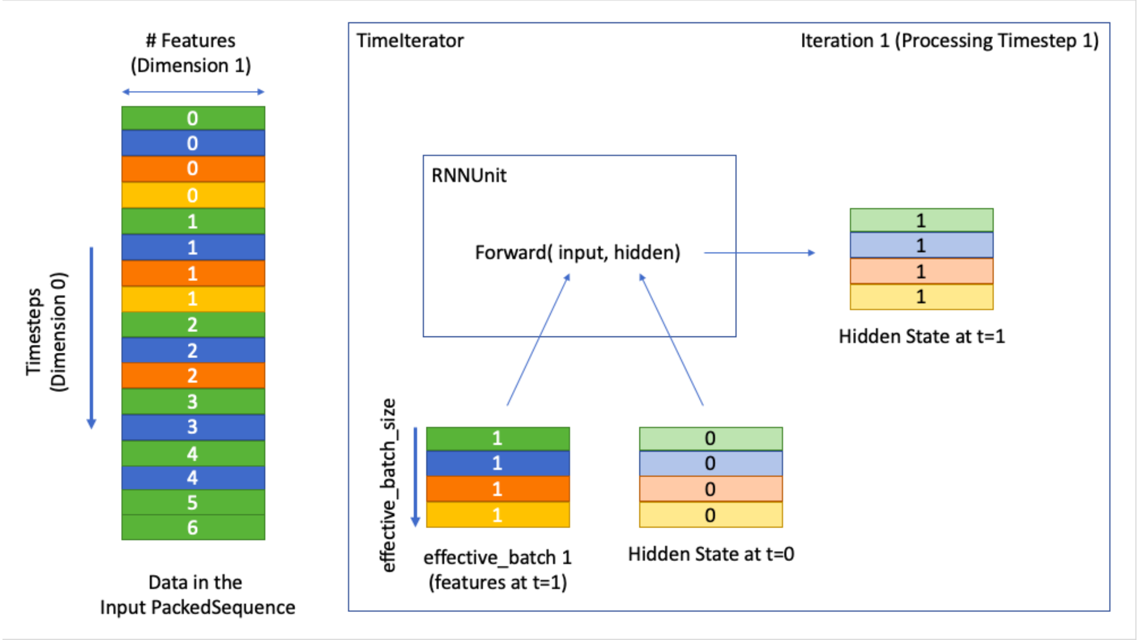
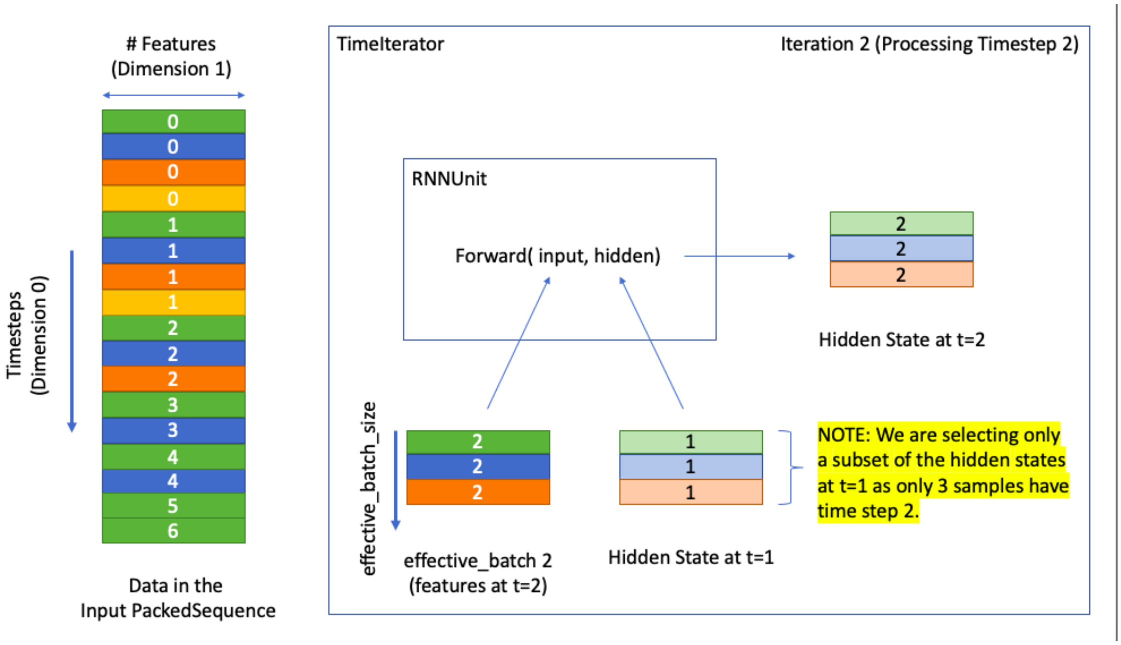


Figure 6: Iteration 1 for the TimeIterator



NOTE: We are selecting only a subset of the hidden states at t=1 as only 3 samples have time step 2.

Figure 7: Iteration 2 for the TimeIterator

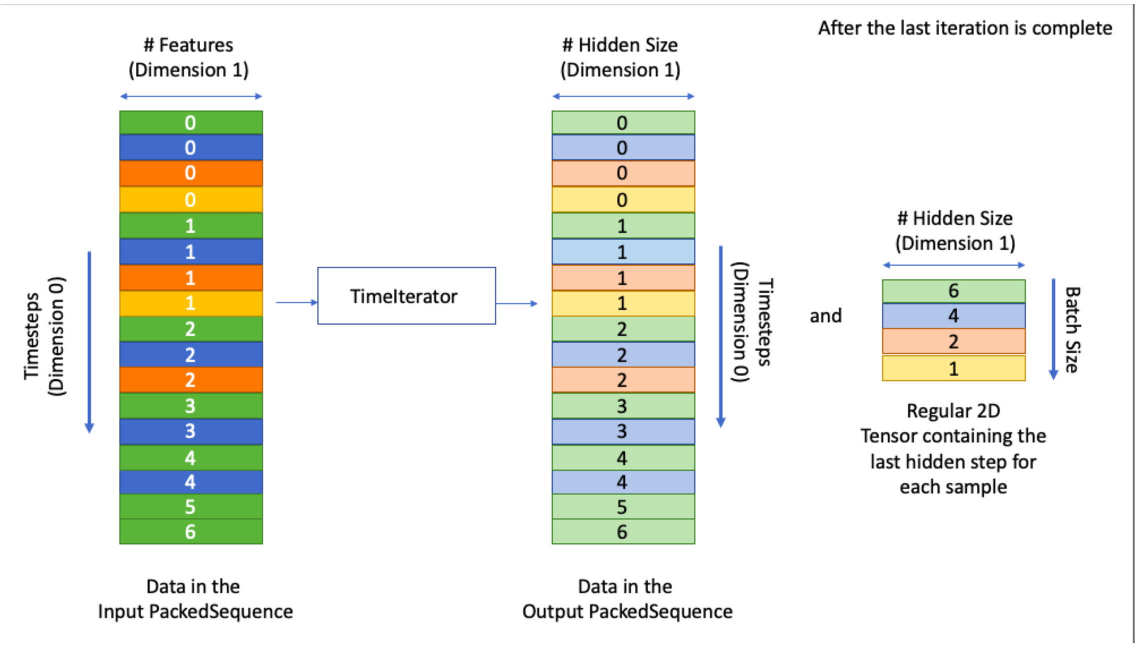


Figure 8: The final output from the TimeIterator

2 GRU (Total 20 points)

In this section we will explore the world of GRU. We will heavily use the machinery built for RNN to make our lives easier and create a full-fledged working GRU.

2.1 GRU Unit (15 points)

In `mytorch/nn/gru.py` implement the forward pass for a GRUUnit (though we follow a slightly different naming convention than the Pytorch documentation.) The equations for a GRU cell are the following:

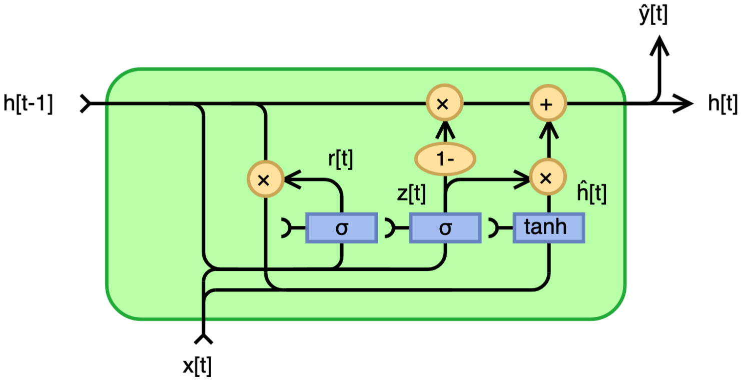


Figure 9: The computation for our GRU

$$\mathbf{z}_t = \sigma(\mathbf{W}_{zh}\mathbf{h}_{t-1} + \mathbf{W}_{zx}\mathbf{x}_t) \tag{2}$$
$$\mathbf{r}_t = \sigma(\mathbf{W}_{rh}\mathbf{h}_{t-1} + \mathbf{W}_{rx}\mathbf{x}_t) \tag{3}$$
$$\tilde{\mathbf{h}}_t = tanh(\mathbf{W}_h(\mathbf{r}_t \otimes \mathbf{h}_{t-1}) + \mathbf{W}_x\mathbf{x}_t) \tag{4}$$
$$\mathbf{h}_t = (1 - \mathbf{z}_t) \otimes \mathbf{h}_{t-1} + \mathbf{z}_t \otimes \tilde{\mathbf{h}}_t \tag{5}$$

Please refer to (and use) the GRUUnit class attributes defined in the init method. You are not expected to define any extra attributes for a working implementation of this class.

The inputs to the GRUCell forward method are x and h represented as x_t and h_{t-1} in the equations above. These are the inputs at time t.

The output of the forward method is h_t in the equations above.

You are required to implement the forward method of this module. The description of the inputs and expected outputs are specified below:

Inputs

- x (effective_batch_size, input_size)
  - Input at the current time step.
  - NOTE: For now interpret effective_batch_size same as regular batch_size. The difference will become apparent later in the homework.
- h (effective_batch_size, hidden_size)
  - Hidden state generated by the previous time step, h_{t-1}

Outputs

- h_prime: (effective_batch_size, hidden_size)
  - New hidden state generated at the current time step, h'_t

2.2 GRU (5 points)

Finally in `mytorch/nn/gru.py` create a class *GRU* as a sub-class of TimeIterator which instantiates the base class with basic_unit = GRUUnit. Please look at the code for more details.

Appendix

A Glossary

- **effective_batch**: i^{th} effective_batch refers to the set of i^{th} timesteps from each sample that are simultaneously fed to the RNNUnit in the $(i-1)^{th}$ iteration inside the TimeIterator
- **effective_batch_size**: number of samples in an effective batch. Effective batch size of the i^{th} effective batch is equal to the number of samples containing atleast (i+1) timesteps

B What's all the fuss about handling variable length samples in one batch?

B.1 Importance of batching in Deep Learning

When training a neural network, we stack together a number of inputs a.k.a batching and pass them together for a single training iteration. From a computational standpoint this helps us make the most of GPUs by parallelizing this computation (done by various frameworks such a PyTorch for you). Moreover ideas like batch normalization fundamentally depend on existence of batches instead of using single samples for training. Therefore there is no denying the importance of batching in the deep learning world.

B.2 Batching for MLPs and CNNs

Life in a world with only MLPs and CNNs was simpler from a batching perspective. Each input sample had exactly the same shape. These input samples in the form of tensors (each with the same shape) could then be stacked together along a new dimension to create a higher-dimensional tensor which then becomes our batch. For example consider a simple 1-D input where each sample has 20 features. We now want to create a batch of 256 such samples. Each sample has a shape (20,) which are then stacked along a new dimension (this now becomes dimension 0) to provide a batch of shape (256,20) where the first dimension represents number of samples in a batch while the second dimension represents the number of features.

B.3 Batching in RNNs

With RNNs, this becomes very tricky. Each sample in this context has a temporal dimension (atleast in the simplest case). The number of feature at each time step remains fixed for all the samples. Let this be K for the purpose of our discussion. Therefore each sample i has T_i time steps where at each time step we have K features. The i^{th} sample can then be represented by a tensor of shape (T_i,K). Given that the first dimension is of variable length there is no simple way for us to create batches as we did for MLPs and CNNs. One might argue for fixed time-step inputs, but that severely limits the power of RNNs/GRUs/LSTMs and reduces them to a large fixed MLP. Therefore batching in RNNs require a special setup which we provide via PackedSequences.