AI539 Natural Language Processing with Deep Learning – Homework 2

Processing Sequences with Recurrent Neural Networks

Overview and Objectives. In this assignment, we'll get some hands-on experience with training recurrent neural networks for sequence classification tasks – a simple parity problem and a part-of-speech tagging application. As for general PyTorch skills, we'll need to figure out how to handle batches of inputs with different lengths and use some additional tools from `wandb` to analyze our hyperparameter choices.

How to Submit. This assignment includes both coding questions and short response questions. Submit a zip file of your code and a PDF of your responses to Canvas. Do not include model checkpoints or training data.

Advice. Start early and be careful about shapes. For this assignment, we recommend getting on to the university high-performance cluster (HPC). See Canvas page for details.

1 A Simple Sequence Classification Problem: Parity

Let's probe LSTMs a bit by making one solve a fairly simple problem – determining if a binary sequence has an even or odd number of 1's. For context, there exists an LSTM with a 1-d hidden dimension that can perfectly solve this problem – it is a relatively fun puzzle to try to deduce it parameters manually. No guarantee of fun is implied. Professor not liable for any frustration.

1.1 Parity: Examining Generalization To Longer Sequences

To get started on this problem, I've provided a class in `datasets/ParityData.py` that generates binary strings of certain lengths. During training, it provides random binary strings up to `max_length` and during testing it returns `samples=1000` binary strings of exactly `max_length`. The bulk of this class is shown below:

```
class Parity(Dataset):

    def __init__(self, training=True, max_length=4, samples=1000):
      super().__init__()
      self.training = training
      self.max_length = max_length
      if self.training:
        self.data = torch.FloatTensor(list(itertools.product([0,1],
                                      repeat=max_length)))
      else:
        self.data = torch.randint(low=0, high=2, size=(samples,max_length))
        self.data = self.data.to(dtype=torch.float32)


    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        if self.training and self.max_length > 1:
           l = torch.randint(low=1, high=self.max_length, size=(1,1)).item()
           x = self.data[idx][:l]
        else:
           x = self.data[idx]

        y = x.sum() % 2  # Compute parity of the binary sequence
        return x,y
```

The file also provides a convenient `getParityDataloader` function shown below. However, our dataset contains strings of variable length so we need to pass a custom function to our data loader describing how to combine elements in a sampled batch together – in PyTorch, the `collate_fn` argument of the dataloader serves this purpose.

```
def getParityDataloader(training=True, max_length=4, batch_size=1000):
    dataset = Parity(training, max_length)
    loader = DataLoader(dataset, batch_size=batch_size, shuffle=training,
                        collate_fn=dataset.pad_collate, drop_last=False)
    return loader
```

The code is currently set up to use a static method from the `Parity` dataset class to handle this – specifically, the `Parity.pad_collate` function which has the following form.

```
class Parity(Dataset):
    . . .
    # Function to enable batch loader to stack binary strings of different
    lengths into one tensor padding them
    @staticmethod
    def pad_collate(batch):
        #TODO
        return xx, yy, x_lens
```

For a batch size of B, the input `batch` will be a list of B `x`,`y` tuples returned from the dataset's `__getitem__` function where each `x` will be a Tensor of arbitrary length and each `y` will be a Tensor of length 1. Note that if `num_workers` in the DataLoader is greater than 1, this list will be the result of multiple dataloader threads pooling their fetched data to form a batch, but this is transparent to us.

▶ TASK Q1 Batches from Random Length Strings [1 pt]. Implement the `Parity.pad_collate` function to produce the following outputs:

- `xx` – a $B \times T_{\max} \times 1$ FloatTensor representing the B `x` inputs of the batch where $T_{\max}$ is the length of the longest batch item. Any items shorter than $T_{\max}$ should be padded to this length by adding zeros. See the Pytorch pad_sequence function for a convenient way to do this.
- `yy` – a B LongTensor representing the B `y` labels of the batch.
- `x_lens` – a B LongTensor representing the length of each of the B `x` inputs of the batch.

Note that the requirement for `yy` and `x_lens` to be LongTensor types is to avoid errors when using the labels or lengths later on when computing cross entropy loss or when indexing into tensors based on lengths.

With that behind us, the `pairty.py` file in the root directory would be our main runner file that trains the LSTM model.With that behind us, the `pairty.py` file in the root directory implements a fairly standard training loop like we've seen before – except that the dataloader now has three outputs corresponding to `xx`, `yy`, and `x_lens` returned from our `pad_collate` function. The model being trained in an `ParityLSTM`; however, it has not yet been defined completely.

▶ TASK Q2 Parity Recurrent Neural Network Implementation [8 pt] Implement the unfinished `ParityLSTM` model class in `models/ParityLSTM.py`.

```
class ParityLSTM(nn.Module) :

    def __init__(self, hidden_dim=16):
        super().__init__()
        #TODO

    def forward(self, x, x_lens):
        #TODO
```

- **__init__**. For the model structure itself, there should be a one layer unidirectional LSTM with hidden dimension of `hidden_dim` and a single linear layer to map an output hidden state from the LSTM to two scalars corresponding to the score for predicting an even or odd parity.
- **forward**. During the forward pass, the input tensor should be processed by the LSTM to produce a $B \times T \times$ `hidden_dim` tensor of output hidden states through time. For each batch element i, the hidden state at time `x_lens[i]-1` should be collected – resulting in a $B \times$ `hidden_dim` tensor that can then be passed through the linear layer to produce a $B \times 2$ output that is then returned. PyTorch tensors use similar rules for indexing as numpy, so collecting the appropriate hidden states should be doable without doing any looping.

Once you get the `ParityLSTM` implemented, running `parity.py` should train your model on binary strings up to length 10 and log several things to wandb. This process should take only a couple of minutes on the GPU.

1.2 Utilities for Logging Experiments

Like most things in the world, deep learning models do not train in the first attempt and often require running extensive search on hyperparameters to find the best performing model. To be able to figure out where things are going wrong, logging desired values such as accuracy, loss, gradient norms, image/video visualizations etc. are of immense help. We've set up `wandb` logging for this parity problem and walk through the details here.

Weights&Biases. We can make use of experiment management tools to help us monitor our training and keep track of hyperpameters for us. In this section, we'll explore one such tool – Weights&Biases (aka `wandb`). *Why this tool?* Because all my graduate students swear by it and I haven't used a better tool so far. (Not a paid endorsement.)

Account Creation. Weights&Biases uses a web-interface for tracking experiment runs. This can either be a locally hosted server for personal projects or through their website. If you follow this link, you should be able to sign up for a free account. Once you've created your account, going to http://wandb.ai/authorize will let you copy your personal API key – *do not share this or leave it in your code.* The API key will be all we need to send information from our training scripts to the web interface.

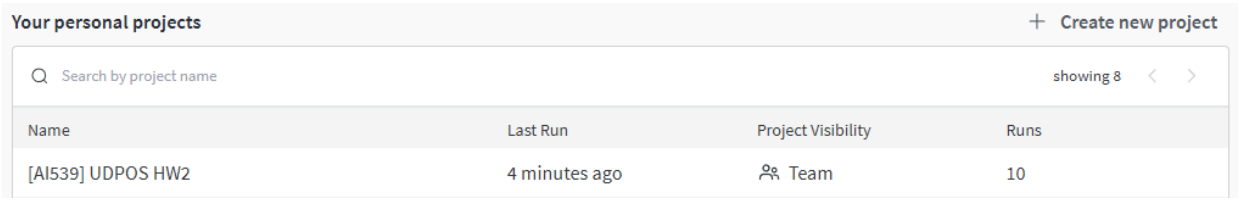
```
import wandb
wandb.login(key=YOUR_API_KEY_HERE) # Remember to remove this key before submitting.
```

Alternatively, calling just `wandb.login()` will produce a terminal prompt asking for your key. After providing it once, it will typically be cached for future runs – the `parity.py` code follows this route.

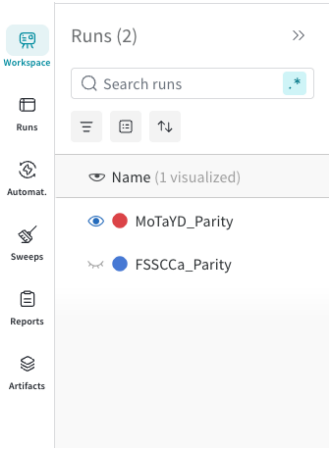
Before we train a new model, we can tell `wandb` what project it is a part of, what to name the current training run, and pass it our configuration dictionary to keep track of our hyperparameters for us. This will let us sort, filter, or group runs by these hyperparameters in the interface. An example call to `wandb.init` is shown below. The `generateRunName()` function is implemented in the skeleton code to generate a random string – e.g. `qX3Ax2_PARITY`.

```
wandb.init(
    # Set the project where this run will be logged in the web interface
    project="[AI539] Parity HW2",
    # We pass a run name to identify this experiment
    name=generateRunName(),
    # Track hyperparameters and run metadata
    config=config)
```

On the website side, we will see a new project added to our profile that will store our runs.



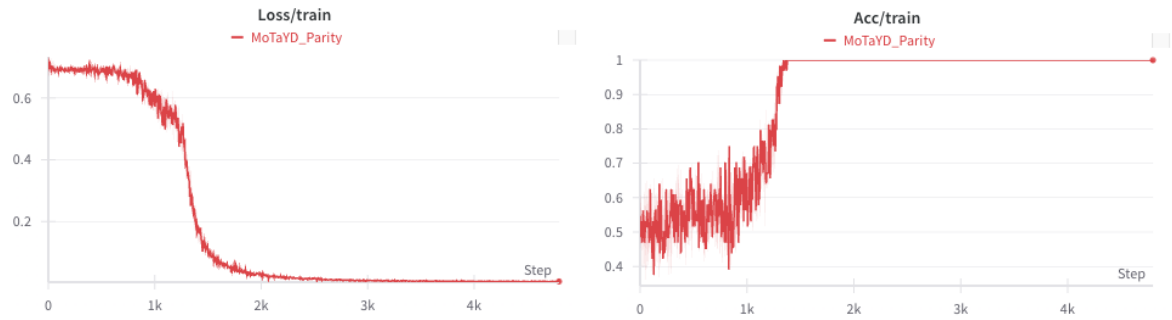
Clicking into that project, I can see information about prior runs I've logged to it.



Logging Values. After a Python script executes `login` and `init`, recording information about our training run to the web platform is quite easy. The primary command is `wandb.log` and it's most common arguments are a dictionary of named values to be logged and a `step` argument indicating how many batches of data have been processed in the training run so far. For instance, the line below logs the training loss and accuracy.

```
loss = ... #Compute loss
acc = ... #Compute accuracy
wandb.log({"Loss/train": loss, "Acc/train": acc}, step=iteration)
```

If the above logging command is called at each batch of training, the web interface will aggregate the results over time and present the following plots inside the `[AI539] PARITY HW2` project for this run:

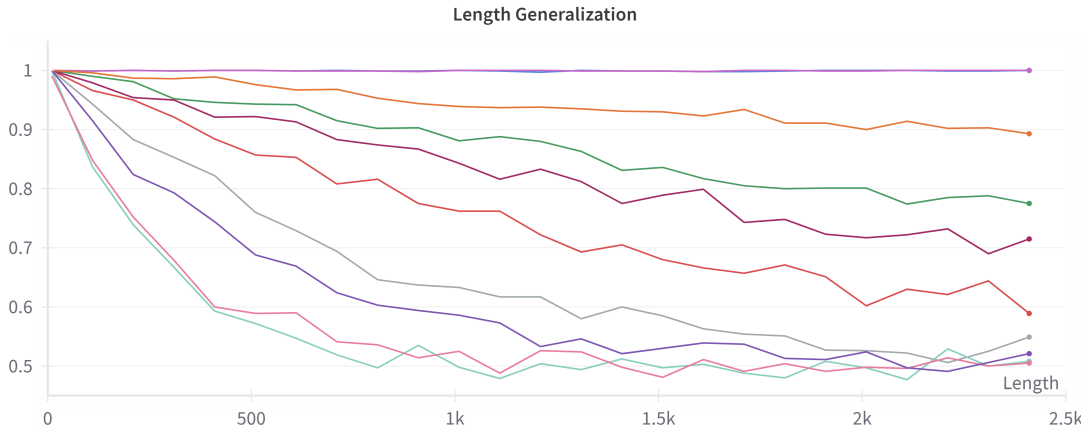


If we later log the test loss with the name "Loss/test", `wandb` will automatically group the two loss plots.

Wrapping Up. When we've finished our training run, we can notify `wandb` that it is over by calling `wandb.finish()`. This will ensure all logged data is successfully flushed to the website before our code exits.

1.3 Evaluating Length Generalization

After training, the code also executes the `runParityExperiment` function which evaluates the accuracy of your trained model on binary strings of increasing length up to 2500 – measuring generalization of your model to strings up to 250x longer than it saw during training! The results are sent to `wandb` as a plot and via two log variables `Text/acc` and `Test/length`. The plot below shows the results of this test for 10 different runs of the code. All runs achieve 100% training accuracy and are identical networks – only differing by the randomness of initialization and batch loading.



Some models (like the purple line at the top) generalize remarkably well, maintaining an accuracy near 100% across a wide range of lengths. Other models degrade as the input strings get longer. There is also substantial variance between different users – with some students reporting *all* runs behaving nearly identical and others replicating results like those shown above.

▶ `TASK Q3 Evaluating Generalization [2 pt].` With all of this randomness, its worth learning how to compare the effect of our hyperparameter choices over multiple runs using `wandb` by Grouping. First, let's generate some data from multiple training runs:

- Run the code 10 times with the default hyperparameters.
- Make a non-trivial change to a hyperparameter in `config` and then run the code 10 more times.

Run plot. To get a plot like the one above, you'll need to modify the x-axis of your `wandb` plot for `Text/acc` to be `Text/length` (see the cog wheel icon when mousing over the plot). Provide this plot in your report.

Grouped plot. By modifying the `Group` settings for this plot in `wandb`, group the runs by your changed hyperparameter and then provide the resulting plot. It should produce a mean and variance for both groups shown by solid and shaded regions. Be sure the plot's legend is enabled. Provide this plot in your report.

▶ `TASK Q3 REPORT.` As shown in Figure 1, there are two run plots: the left run plot is based on the default setting, and the right run plot is based on the parameter setting where batch size is 128 and hidden dimension is 64. Figure 2 shows the group plots based on the batch size and hidden dimension parameters. As a result of the increase in both batch size and hidden dimension, the performance is reduced until reaching around 80%.

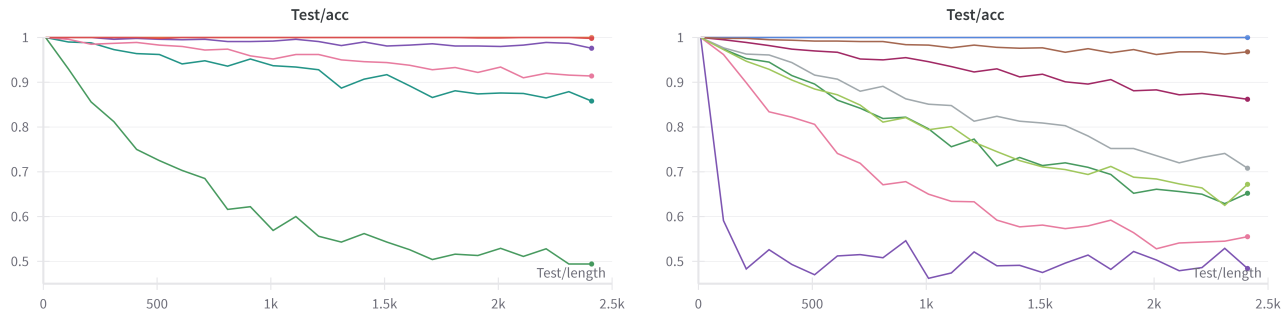


Figure 1: Run Plots - 10 default (left) and 10 non-trivial change (right)

2 Part-of-Speech Tagging with Bidirectional GRUs

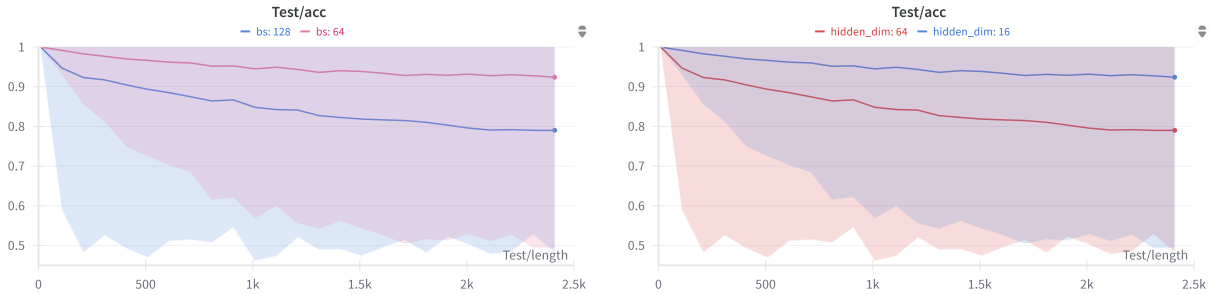


Figure 2: Group Plots - batch size 128 vs 64 (left) and hidden_dim 64 vs 16 (right)

UDPOS Dataset. Now lets get to work on a real problem – part-of-speech tagging in English. This is a many-to-many sequence problem where each word in an input must be labeled with a corresponding part-of-speech tag. The goal of this section will be to train a bidirectional GRU model for this task. The dataset we will be using is the Universal Dependencies English Part-Of-Speech (UDPOS) benchmark which has these parts-of-speech annotated. The dataset has `train`, `dev`, and `test` sets made of approximately 12.5k, 2k, and 2k sentences each. These are largely news and email snippets. Example code visualizing an example from the training set is shown below:

```
> x, y, lens = next(iter(train_loader))

> words = vocab.denumeralizeSentence(x[0,:lens[0]].flatten().numpy())
> labels = vocab.denumeralizeLabels(y[0,:lens[0]].flatten().numpy())

> visualizeSentenceWithTags(words, labels)

Token                POS Tag
---------------------------------
they                 PRON
are                  AUX
doing                VERB
it                   PRON
deliberately         ADV
.                    PUNCT
```

Unfortunately, words are not natively mathematical objects. We'll need to do some tokenization (breaking sentences down into discrete units called tokens) and numeralization (turning tokens into numbers) before we can have our models take them as input. The skeleton code provides a basic version of this functionality in `datasets/PoSData.py` via the `Vocab` class (which you may recognized from the previous assignment). Given a data split from UDPOS, the Vocab class finds all unique words and labels then builds dictionaries to quickly map between a word (or label) and its assigned numerical identifier. We expose this functionality through the numeralize / denumaralize functions demonstrated in the code snippet below.

```
> words = ['in', 'the', 'eastern', 'city', 'of', 'baqubah', ',']
> word_ids = vocab.numeralizeSentence(words)
[16, 12, 3007, 404, 18, 0, 11]

> words_back = vocab.denumeralizeSentence(word_ids)
['in', 'the', 'eastern', 'city', 'of', '<UNK>', ',']

> labels = ['ADP', 'DET', 'ADJ', 'NOUN', 'ADP', 'PROPN', 'PUNCT']
> label_ids = vocab.numeralizeLabels(labels)
[6, 5, 2, 3, 6, 0, 1]
```

Numeralizing an input converts it to corresponding word (or label) ID integers – e.g., the word 'in' being mapped to 16. These IDs are what our model will actually see rather than the words. Note that when we denumeralize `word_ids` above, the word `baqubah` is replaced with `<UNK>`. This happens because `baqubah` did not appear in our training set where the `vocab` was built, so it does not have a numerical ID. As such, it was mapped to a special ID of 0 when numeralized to indicate an unknown input. All such words will be mapped to this UNK ID and appear identical to our models. Note that this means our model is only useful if we are encoding text with the same vocabulary that was used during training! Further, we also include a special `<PAD>` token the in vocabulary to denote when batched inputs have been padded.

The skeleton code provides a convenient function to get dataloaders for all three splits and the training vocabulary. Note how the vocabulary from `train_data` is passed to the validation and test dataset constructors.

```
def getUDPOSDataloaders(batch_size=128):
    train_data = UDPOSDataset(split='train')
    val_data = UDPOSDataset(split='dev', vocab=train_data.vocab)
    test_data = UDPOSDataset(split='test', vocab=train_data.vocab)

    train_loader = DataLoader(dataset=train_data, batch_size=batch_size,
                              shuffle=True, num_workers=8, drop_last=True,
                              collate_fn=UDPOSDataset.pad_collate)

    val_loader = DataLoader(dataset=val_data, batch_size=batch_size,
                              shuffle=False, num_workers=8, drop_last=False,
                              collate_fn=UDPOSDataset.pad_collate)

    test_loader = DataLoader(dataset=test_data, batch_size=batch_size,
                              shuffle=False, num_workers=8, drop_last=False,
                              collate_fn=UDPOSDataset.pad_collate)

    return train_loader, val_loader, test_loader, train_data.vocab
```

And in the `UDPOSDataset` dataset class, we use the vocabulary to numeralize the inputs and labels into tensors of integers in the `__getitem__` function below:

```
class UDPOSDataset(Dataset):

    . . .

    def __getitem__(self, idx):
        x, y = self.data[idx]

        x = torch.LongTensor(self.vocab.numeralizeSentence(x))
        y = torch.LongTensor(self.vocab.numeralizeLabels(y))

        return x, y
```

Of course, both `x` and `y` can be different for each element in our batch – we once again need a function for our dataloaders to handle batching our variable-length inputs! In this case though, we need to think a bit more about how to pad things appropriately.

▶ TASK Q4 Batches from Variable Length Sequences (Again) [1 pt]. Implement the `pad_collate` static method in the `UDPOSDataset` dataset class. Given a list of (`x`,`y`) pairs, return three tensors:

- `xx` – a $B \times T_{\max}$ LongTensor representing the B `x` inputs of the batch where $T_{\max}$ is the length of the longest batch item. Any items shorter than $T_{\max}$ should be padded to this length by adding 1's. Note that we are choosing 1's as this corresponds to the ID of the special "<PAD>" token in our vocabulary.
- `yy` – a $B \times T_{\max}$ LongTensor representing the B `y` labels of the batch.Any items shorter than $T_{\max}$ should be padded to this length by adding -1's. Note that we are choosing -1's here so we can identify these positions as `not` having labels (which have non-negative IDs).
- `x_lens` – a B LongTensor representing the length of each of the B batch sequences.

Bidirectional GRU Model. In this task, we need to predict a distribution over part-of-speech tags for each word in our batch. That is to say: for a $B \times T$ input of word IDs we need to produce a $B \times T \times |L|$ output where $|L|$ is the number of labels in our task. We will also need to configure our calculations for loss and accuracy accordingly. We will design a model that uses a bidirectional GRU as its primary sentence representation module.

We've consider our word representations as integer IDs so far; however, directly inputting these to our model is probably a bad idea. Two words having consecutive IDs (e.g., 16 and 17) is not semantically meaningful, but would look more similar to our model as input than words with distant IDs (e.g., 16 and 2045). To avoid this, it is useful to have each word represented by a learnable vector so that similarities and differences can be decided based on the data. For example, if the word "`in`" maps to ID 16, we would like some d_{embed}-dimensional vector e_{16} to represent it. These are commonly referred to as word embeddings or embedding vectors. For a $B \times T$ input of word IDs, we might collect the corresponding word embeddings for each word to produce a $B \times T \times d_{\text{embed}}$ tensor. PyTorch offers a useful nn.Embedding layer that does this efficiently for tensors of integer IDs.

▶ TASK Q5 Implement PoSGRU [12 pt]. In `models/PoSGRU.py` complete the `PoSGRU` class shown below by implementing the following:

```
class PoSGRU(nn.Module) :

    def __init__(self, vocab_size=1000, embed_dim=16, hidden_dim=16,
    num_layers=2, output_dim=10, residual=True, embed_init=None):
        super().__init__()

    def forward(self, x):
```

- **`__init__`**. The model will consist of an Embedding layer of dimension `embed_dim` and size `vocab_size`, a Linear layer mapping from `embed_dim` to `hidden_dim`, `num_layers` bidirectional GRUs with hidden dimension of `hidden_dim`/2, and a classifier module consisting of a `hidden_dim` Linear layer, a GELU activation, and a `output_dim` Linear layer. Note that a bidirectional GRU will output a concatenation of the forward and backward hidden states, such that setting its dimensionality to `hidden_dim`/2 results in an `hidden_dim` dimensional output per time step. For the embedding layer, setting the appropriate `padding_idx` is good practice.
- **forward**. During the forward pass, the $B \times T$ input tensor of word IDs should be embedded to a tensor of word embeddings that are then mapped to `hidden_dim` with our linear layer. Then each GRU layer should be applied in sequence (with a residual connection after each layer if `residual` is `True`). The result of this process should be a $B \times T \times$ `hidden_dim` tensor that is then mapped to $B \times T \times$ `output_dim` by the classifier and then returned.

Structure of PyTorch Training Code. Generally speaking, a standard training script in PyTorch will consist of the following steps: (1) construction of a dataset and dataloader to provide batches of training data, (2) definition of a model to be trained, (3) registration of model parameters to an optimizer, and (4) a training loop that takes batches of data, computes the model's loss on the batch, calculates gradients via backpropagation, and updates parameters via the optimizer. Note that we have a variable number of outputs and labels for each element of our batch! This requires some extra work to get right with the standard CrossEntropyLoss function.

▶ TASK Q6 API Hunt! Setup optimizer and learning rate scheduler. [2 pt] To complete the `train` function in `train_pos.py` we'll need to implement an optimizer for our model parameters. Look for initializations and parameters of the optimizers at `torch.optim` pytorch documentation. Using a combination of these function, create the following optimizer setup – (a) an AdamW optimizer with `config['lr']` learning rate and `config['l2reg']` weight decay, (b) a sequential learning rate schedule to perform linear warmup (0.25x the learning rate at the start up to 1x the learning rate after 10% of the training epochs) followed by a cosine annealed learning rate for the remainder of the training epochs.

▶ TASK Q7 API Hunt! Implement Loss and Accuracy Calculations [2 pt] Write code to do the following in the training and evaluation loop inside `train_pos.py`:

Forward pass. Get the data from the `data_loader`, move it to the GPU and perform a forward pass through the `PoSGRU` model that you have implemented.
Compute cross entropy loss. Review the PyTorch documentation for CrossEntropyLoss to understand the required shapes to compute our loss in this problem. Further, we want to ignore any losses coming from padded tokens (which we denoted with -1 labels in `UDPOSDataset.pad_collate` earlier).

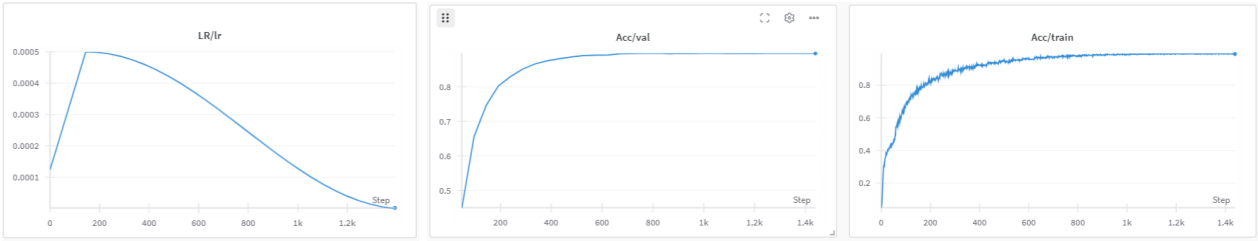
Compute accuracy. We also need to implement an accuracy calculation that likewise ignores padding and only considers our performance on actual input tokens. Implement one!

▶ TASK Q8 API Hunt! Implement Evaluation Function and Model Checkpointing [2 pt] In `train_pos.py`, complete the `evaluate` function to return the validation loss and accuracy. Then, use validation accuracy to perform model checkpointing. By checkpointing, we just mean saving model parameters each time you see a new highest validation accuracy. As our model depends on the specific vocabulary used during training, you'll also want to save that out. The easiest way is to save the entire vocabulary class from `train_loader.dataset.vocab` using the pickle package.

▶ TASK Q9 Integrate wandb logging [1 pt]. Use `wandb` to log the following:

- Learning rate as `LR/lr` once per epoch. You may need to look into the `get_last_lr` function for your learning rate scheduler to get the correct value.
- Training loss and accuracy per batch as `Loss/train` and `Acc/train` respectively.
- Validation loss and accuracy once per epoch as `Loss/val` and `Acc/val` respectively.

Once these are implemented, `train_pos.py` should be able to be run to train your model. You can expect GPU-based training times well under 5 minutes for this problem. `wandb` logging has been setup and you should see training accuracies near 100% and validation accuracies over 85% for default parameters. Sample plots for learning rate and accuracy are shown below:



2.1 Ablations & Experimentation

▶ TASK Q10 Justifying our GRU [1 pt] When `layers` is set to 0 in the `config` in `train_pos.py`, there are no GRUs at all in our model. Instead, each word is embedded and processed by a few linear layers independently. Using your new knowledge about `Grouping` in `wandb`, run multiple trials and use the graphed means and variance to argue for the choice of a GRU here over this word-independent `layers=0` baseline. Run at least 10 random seeds for each setting. Report the resulting plot to justify your argument.

▶ TASK Q10 REPORT. To evaluate the effectiveness of using a GRU, two models are trained, one with a 2-layer bidirectional GRU and another with 0-layer (no GRU), across 10 random seeds each. The GRU model achieved a higher mean validation accuracy (88%) than the baseline (83%) with lower variance. Figure 3 shows that the GRU captures sequential dependencies more effectively and is more stable and accurate.

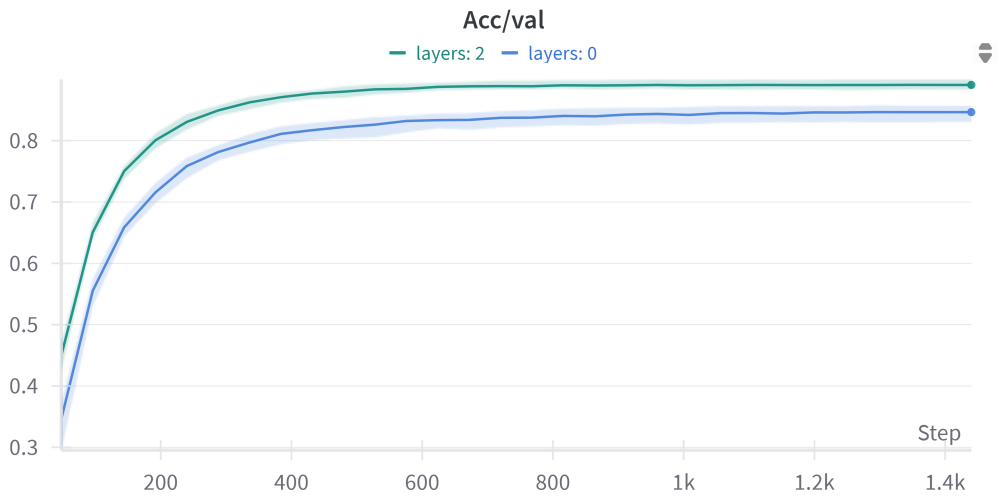


Figure 3: Performance vs. the number of layers (0, 2) in GRU

Using pretrained GloVe vectors: Instead of training the word embeddings from scratch, let's use GloVe vectors that we learned about in Week 1 as an initialization. For this, we will make use of the `gensim` library and you will have to `pip install gensim`.

```
import  gensim.downloader
glove_wv = gensim.downloader.load('glove-wiki-gigaword-100')
```

Now, you can use the downloaded word vectors to query the learned embedding:

```
>> glove_wv['attention'] # look at the GloVe embedding for the word "attention"
array([ -3.3414e-01,  4.6667e-01, ..., 6.7652e-01], dtype=float32)
>> glove_wv['attention'].shape # look at the shape
(100,)
>> glove_wv['GRU'] # Let's see if we can get the embedding for "GRU"
KeyError: "Key 'GRU' not present"
```

As you can see there will be some words (such as "GRU" in the above example) that do not exist in the vocabulary of the corpus on which the GloVe embeddings were trained on. So how can we ensure that there exists a representation for them? You can randomly initialize them (normally distributed with mean 0 and variance 1). Note that even though you initialize the world vectors with GloVe embeddings, you still would like to learn (in machine learning lingo – *finetune*) the word representation.

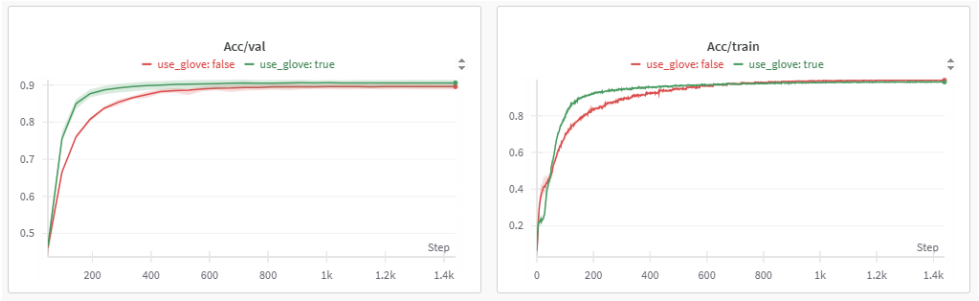
▶ TASK Q11 Use GloVe word vector initializations. [2 pt] To have our model make use of pre-trained GLoVe embeddings, you'll need to:

- Modify `main()` in the `train_pos.py` to build a `vocab_size` $\times$ 100 dimension tensor initialized by GLoVe vectors for our vocabulary words. Pass this as the `embed_init` argument to the `PoSGru` constructor. You may use the above boilerplate code for loading the GloVe embeddings.
- Modify the PoSGRU `__init__` function to initialize the `nn.Embedding` layer with `embed_init` if it is not `None`. The `nn.Embedding.from_pretrained` function can be useful for this, but you'll want to ensure the parameters are not frozen and that no normalization of the vectors occurs – check the API for appropriate arguments.

After you've made these modifications, train the model and answer:

1. How many words did you encounter that did not have GloVe embeddings? List 2-3 of these words.
2. Show the loss and accuracy plots from *wandb* comparing the difference in your model with and without GLoVe embeddings. Describe your observations.

In our experiments with a 2-layer GRU and otherwise default hyperparameters, we see initializing from GLoVe vectors leads to faster improvement on the validation set and some gains in accuracy over the random initialization. The plot below is grouped by GLoVe vector usage and consists of 10 split between the two settings.



▶ TASK Q11 REPORT. As shown in Figure 4, out of 16358 total vocabulary words, 15022 were found in the Glove vectors, meaning that 1336 were missing words. Some of the missing words include: "'akkab", 'jubur', 'batawi.'

To evaluate the impact of using pretrained word embeddings, two models trained, one with $use\_glove = True$ and another with $use\_glove = False$, are compared.

Figure 5 shows the train and validation loss respectively. The validation loss plot indicates that models initialized with GloVe converge faster and achieve lower final validation loss (0.40823) compared to other models (0.42807). Additionally, the training loss curves are almost identical for both settings, implying that using GloVe benefits generalization to the validation sets. These results support the effectiveness of using Glove.

As can be seen in Figure 6, the validation accuracy results further support the benefits of GloVe initialization. Models trained with GloVe converged faster and achieved slightly higher validation accuracy than those trained without GloVe. Specifically, the GloVe-based models reached a final average validation accuracy of around 91%, compared to 89% for the non-GloVe models. Meanwhile, training accuracy was nearly identical in both settings.

```
Word vectors loaded: 15022 / 16358
Example missing words: ["'akkab", 'jubur', 'batawi']
```

Figure 4: The number of Missing words and examples

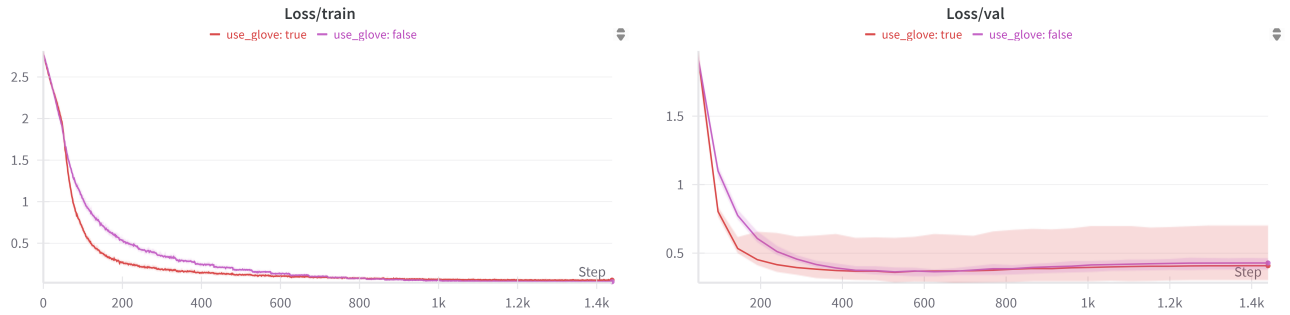


Figure 5: Train and Validation Loss vs. the use of Glove

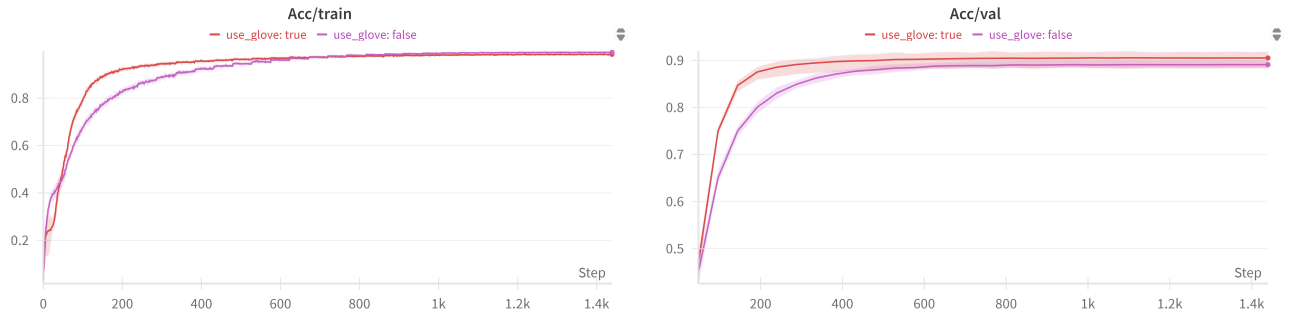


Figure 6: Train and Validation Accuracy vs. the use of Glove

2.2 Inference

Finally, let's make use of your trained model on some example sentences. To do so, we'll need to build a driver program that loads your trained model, applies the same tokenization/numeralization scheme as from model training, and then runs inference. *Hint: Be careful with capitalization and whitespace here!*

▶ `TASK Q12 Build driver script and run inference. [6 pt]` Implement a script named `tag.py` that prompts the user for an input sentence when run and returns the part-of-speech tags predicted by your trained model. This will require you to tokenize/numeralize the sentence, pass it through your network, and then print the denumeralized results. Run your script on the following sentences and report the output:

1. The old man the boat
2. The complex houses married and single soldiers and their families
3. The man who hunts ducks out on weekends

▶ `TASK Q12 REPORT.` A script named 'tag.py' that loads the trained PoSGRU model and vocabulary is implemented. It prompts the user to input a sentence, tokenizes and numeralizes it, passes it through the model, and denumeralizes the predictions to return POS tags.

As can be seen in Figure 7, there are inference results of three sentences. For the first sentence "The old man the boat", the model likely interpreted "man" as a noun instead of a verb. In the case of "The complex houses married and single soldiers and their families", the model did a good job capturing the structure of the sentence, identifying "houses" as a noun and "married" as a verb. Lastly, the inference result of the sentence "The man who hunts ducks out on weekends" implies that the model correctly tagged "hunts" as a verb, but "ducks" as a verb, which may be derived from ambiguous case (e.g. ducks out).

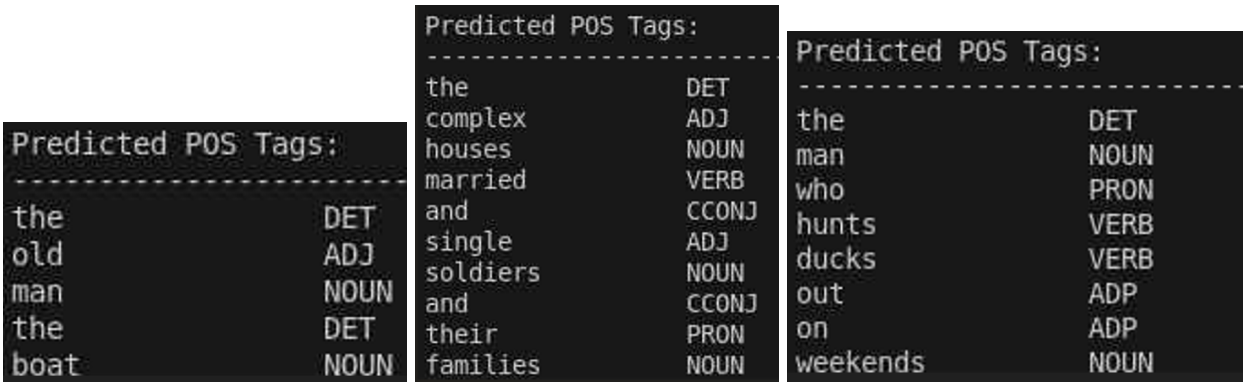


Figure 7: Inference results of three sentences

3 Debriefing (required in your report)

1. Approximately how many hours did you spend on this assignment?

   I spent on this assignment about four days

2. Would you rate it as easy, moderate, or difficult?

   For me, the assignment is to some extent between moderate and difficulty.

3. Did you work on it mostly alone or did you discuss the problems with others?

   I did it mostly alone.

4. How deeply do you feel you understand the material it covers (0%–100%)?

   nearly 80%

5. Any other comments?

   Thanks for providing good quality of assignments. I learn so many things that make me understand the concept of NLP with deep learning :)