System Project Notebook

Tabel of Contents

1. **Overview**
2. **Decomposition and Domain Expertise**
3. **Data**

- Data Source
- Model Vocabulary

4. **Design**

- Hyperparameters
- Pipeline

5. **Training**

- PyTorch and PyTorch Lightning

6. **Visualizations**

- Tensorboard
- Weights and Biases

7. **Diagnosis/Evaluation**

- Default Model
- Comparing Models

8. **Conclusion and Takeaways**
9. **References**

In [ ]:
```
import os
import pickle

from model import Collater, SimpleDataset, Transformer, evaluate, load_model, pairs
from data import PolynomialLanguage, train_test_split, load_file

import torch
import pytorch_lightning as pl
from torch.utils.data import DataLoader
from utils import get_device
```

In [ ]:
```
print('pyTorch VERSION:', torch.__version__)
print('pyTorch Lightning VERSION:', pl.__version__)
print('-' * 50)

print('CUDA VERSION', torch.version.cuda)
print ('Available devices ', torch.cuda.device_count())
print('Active CUDA Device: GPU', torch.cuda.current_device())

device = get_device()
```

```
pyTorch VERSION: 1.13.0
pyTorch Lightning VERSION: 1.7.7
--------------------------------------------------
CUDA VERSION 11.6
Available devices  1
Active CUDA Device: GPU 0
```

1. Overview

My system project will tackle sequence-to-sequence prediction using a Transformer AI model. This falls under the category of neural machine translation where a deep neural network will be used. This is a common task among natural language processing applications.

To be clear, this problem does not *need* neural machine translation to solve. It is likely that there are hardcoded solutions for distributed mutiplication of this type. However, applying a machine learning perspective to this problem is a fun and interesting challenge.

My project with not be using human language, but rather algebraic polynomial expansion. An example can be shown here:

In [ ]:
```
from data import load_file

inputs, outputs = load_file("inputs/test_set.txt") # loading test set

print(f"{'SOURCE (src)':^30}{'TARGET (trg) ':^36}")
print("-" * 66)
for i in range(5):
    print(f"{inputs[i]:^30} | {outputs[i]:^30}")
    print("-" * 66)
```

```
         SOURCE (src)                    TARGET (trg)
------------------------------------------------------------------
         -5*h*(5-2*h)          |          10*h**2-25*h
------------------------------------------------------------------
          s*(8*s-21)           |          8*s**2-21*s
------------------------------------------------------------------
       (21-t)*(-6*t-4)         |        6*t**2-122*t-84
------------------------------------------------------------------
       (21-5*c)*(3*c-7)        |       -15*c**2+98*c-147
------------------------------------------------------------------
          4*n*(n+22)           |          4*n**2+88*n
------------------------------------------------------------------
```

The input to my model is a factored form of a polynomial sequence. The output is the expanded form of the polynomial sequence.

2. Decomposition/Domain Expertise

Predicting a word based on previous words is a fundamental task for NLP. As human beings, we extrapolate and form sequences of thought almost intuititively. If we want to be able to communicate with intelligence machines, we are going to need to teach them language that we already understand. This is the first step in creating a system that can learn to communicate with us.

If we can build a model with natural language ability, this would reduce the workload for many communication-based tasks and greatly increase the speed of communication. In a TED talk by Sam Harris about the safety of AI, he mentions, "Electronic circuits function about a million times faster than biochemical ones. (...) The AI should think about a million times faster than the people who built it" https://youtu.be/8nt3edWLgIg. The speed of computational processing would allow us to achieve new insight into the world around us. This would allow us to explore new ideas and concepts that we have never been able to before.

Natural language processing is a continously developing field of study, and I only have a grasp on the basics. Using polynomial expansion allows exploration of the deep learning process with a greatly reduced "vocabulary".

3. Data

The data for this project was provided by a machine learning company for an interview application assessment. It is a set of 1,000,000 factored and expanded polynomials, line-by-line, located in a .txt file. I have split the original data file into a training set and a test set. The training set is 80% of the original data and the test set is 20% of the original data.

Link to source data: https://scale-static-assets.s3-us-west-2.amazonaws.com/ml-interview/expand/train.txt

The data exploration and analysis process is located in the Python Notebook file: exploratory_data_analysis.ipynb.

In order for the model to process this textual data, a number of process must take place which are outlined below.

In [ ]:
```
# Setting path variables
train_set_path = "inputs/train_set.txt"
test_set_path = "inputs/test_set.txt"
model_path = "models/"
```

In [ ]:
```
train_set_pairs = PolynomialLanguage.load_pairs(train_set_path) # loading train set

# print the first 3 pairs in the training set
print('First 3 pairs: ')
for pair in train_set_pairs[:3]:
    print(pair)
```

```
First 3 pairs:
['6*i*(-4*i-16)', '-24*i**2-96*i']
['-4*s*(-4*s-27)', '16*s**2+108*s']
['-6*o*(20-4*o)', '24*o**2-120*o']
```

In [ ]:
```
# Hyperparameter for train vs validation split
ratio = 0.95

src_lang, trg_lang = PolynomialLanguage.create_vocabs(train_set_pairs) # creating s
train_pairs, val_pairs = train_test_split(train_set_pairs, ratio=ratio) # split for
train_tensors = pairs_to_tensors(train_pairs, src_lang, trg_lang) # converting trai
val_tensors = pairs_to_tensors(val_pairs, src_lang, trg_lang) # converting val pair


save_to_pickle = {"src_lang.pickle": src_lang, "trg_lang.pickle": trg_lang,}

for k, v in save_to_pickle.items(): # saving source and target language objects
    with open(os.path.join(model_path, k), "wb") as file_out:
        pickle.dump(v, file_out)
```

```
creating vocabs: 100%|██████████| 800000/800000 [00:06<00:00, 126573.69it/s]
creating tensors: 100%|██████████| 760000/760000 [00:12<00:00, 60114.24it/s]
creating tensors: 100%|██████████| 40000/40000 [00:00<00:00, 69714.98it/s]
```

In [ ]:
```
# printing the vocabulary
print('Source (src) Vocabulary: ')
print(src_lang.word2index)
print('-'*100)
print('Target (trg) Vocabulary: ')
print(trg_lang.word2index)
print('-'*100)
print()
# print the first 2 tensors in the training set
print('First 2 tensors: (src, trg)')
for tensor in train_tensors[:2]:
    print(tensor)
    print('-'*100)
```

```
Source (src) Vocabulary:
{'<pad>': 0, '<sos>': 1, '<eos>': 2, '<unk>': 3, '6': 4, '*': 5, 'i': 6, '(': 7, '-
': 10, ')': 11, 's': 12, '2': 13, '7': 14, 'o': 15, '0': 16, 'n': 17, '3': 18, 'z':
'+': 21, 'y': 22, '8': 23, 'cos': 24, 'c': 25, 'j': 26, '9': 27, 't': 28, 'k': 29,
'h': 32, 'sin': 33, '**': 34, 'tan': 35}
----------------------------------------------------------------------------------
Target (trg) Vocabulary:
{'<pad>': 0, '<sos>': 1, '<eos>': 2, '<unk>': 3, '-': 4, '2': 5, '4': 6, '*': 7, 'i
': 10, '6': 11, '1': 12, 's': 13, '+': 14, '0': 15, '8': 16, 'o': 17, 'n': 18, '7':
': 21, 'z': 22, 'y': 23, 'cos': 24, '(': 25, 'c': 26, ')': 27, 'j': 28, 't': 29, 'k
x': 32, 'h': 33, 'sin': 34, 'tan': 35}
----------------------------------------------------------------------------------

First 2 tensors: (src, trg)
(tensor([ 1,  7, 12,  8, 10, 27, 11,  5,  7, 12, 21, 18, 16, 11,  2]), tensor([ 1,
2, 12,  7, 13,  4, 20, 19, 15,  2]))
----------------------------------------------------------------------------------
(tensor([ 1,  8, 18,  5, 15,  5,  7, 15, 21, 10, 20, 11,  2]), tensor([ 1,  4, 21,
4,  6, 20,  7, 17,  2]))
----------------------------------------------------------------------------------
```

Looking above at each source and target tensor, we can see all tensors start with a 1 and end with a 2. This is used to indicate the start and end of a sequence.

4. Design

Hyperparameters

Rules for Transformer Models:

- The hidden_dim must be divisible by the number of heads.
- The batch_size should ideally be as large as possible for a given GPU. This greatly increases the speed of training due to parallelization. I have 12GB of VRAM on my GPU, so I can use a batch size larger than the standard values of 32 or 64.

In [ ]:
```
# Specifying hyperparameters for model
hid_dim=256 # default 256
enc_layers=3 # number of encoder layers, default 3
dec_layers=3 # number of decoder layers, default 3
enc_heads=8 # number of self-attention heads in encoder, default 8
dec_heads=8 # number of self-attention heads in decoder, default 8
enc_pf_dim=512 # position-wise feedforward dimension in encoder, default 512
dec_pf_dim=512 # position-wise feedforward dimension in decoder, default 512
enc_dropout=0.1 # dropout in encoder, default 0.1
dec_dropout=0.1 # dropout in decoder, default 0.1

# For batch processing
num_workers=8 # basically the number of parallel processes for *loading* data, defa
batch_size=512 # number of examples in a batch

# Specifying hyperparameters for training
val_check_interval=0.2 # how often to check validation loss (as a proportion of tra
max_epochs=10 # maximum number of epochs to train for
clip=1 # gradient clipping
fast_dev_run=True # Show the output of the training loop without having to actually
```

Pipeline

Collating and Batching:

PyTorch has a method to batch data that helps to take advantage of parallelization. This allows for the input data to be passed to the model in smaller pieces.

The Transformer model class does allow for variable length input and target sequences, but not within individual batches. To fix the input and target lengths and minimize the amount of padding, a custom collator class and corresponding functions are used to pad the input and target sequences to the maximum length of the batch. Using a padding value of 0 allows us to extend the length of the vectors without affecting the dot product calculations.

The PyTorch DataLoader class is used to create the batches. This function returns an iterator data structure that can be used to iterate over the batches. The DataLoader requires a custom dataset class with three methods: `__init__`, `__len__`, and `__getitem__`. The `__init__` method initializes the dataset and the `__len__` method returns the length of the dataset. The `__getitem__` method returns the input and target sequences for a given index.

In [ ]:
```
collate_fn = Collater(src_lang, trg_lang) # initializing collate function

# the data is loaded in batches and then iterated over
train_dataloader = DataLoader(SimpleDataset(train_tensors), # SimpleDataset is a cu
                              batch_size=batch_size, # batch size
                              collate_fn=collate_fn, # collate function
                              num_workers=num_workers) # number of parallel process

val_dataloader = DataLoader(SimpleDataset(val_tensors),
                            batch_size=batch_size,
                            collate_fn=collate_fn,
                            num_workers=num_workers)
```

In [ ]:
```
print('Number of batches in training set: ', len(train_dataloader))

i = 0

for src, trg in train_dataloader:
    print('Batch: ', i)
    print('Source Length: ', src.shape[1], 'Target Length: ', trg.shape[1]) # the l
    i += 1
    print('-'*50)
    if i == 3:
        break
```

```
Number of batches in training set:  1485
Batch:  0
Source Length:  25 Target Length:  24
--------------------------------------------------
Batch:  1
Source Length:  25 Target Length:  25
--------------------------------------------------
Batch:  2
Source Length:  27 Target Length:  25
--------------------------------------------------
```

In [ ]:
```
# showing the padding within the tensors in src
print('Source (src) tensor with padding "0": ')
print(src[:1])
```

```
Source (src) tensor with padding "0":
tensor([[ 1,  7,  8, 23,  5, 17,  8, 13, 16, 11,  5,  7, 17,  8,  9, 11,  2,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0]])
```

The source tensor is padded to the maximum length sequence of the batch.

In [ ]:
```
# Initializing the model using the hyperparameters
model = Transformer(src_lang=src_lang, trg_lang=trg_lang, hid_dim=hid_dim,
                    enc_layers=enc_layers, dec_layers=dec_layers, enc_heads=enc_hea
                    dec_heads=dec_heads, enc_pf_dim=enc_pf_dim, dec_pf_dim=dec_pf_d
                    enc_dropout=enc_dropout, dec_dropout=dec_dropout)
```

5. Training

PyTorch and PyTorch Lightning

For the purposed of this overview, the model has already been fully trained.

During training, the below cell: model = train(...) was run. PyTorch Lightning produces a number of files to help keep track of the model training process. The model checkpoint is located in the models folder. Basic Checkpoint Documentation

This .ckpt file is updated after every epoch of training, in case the training process is interrupted. The model checkpoint is also used to load the model for inference.

To show what the output of PyTorch Lighting looks like, I have passed in a parameter called "fast_dev_run". This parameter is generally used for debugging purposes, but it is useful for showing the output PyTorch Lightning produces during training.

During training, there are a number of progess bars that update with each batch and validation pass. Because the model is in fast_dev_run mode, only one step is shown.

DO NOT RUN THE CODE IN THE NEXT CELL WITH FAST_DEV_RUN = FALSE UNLESS YOU WANT TO TRAIN THE MODEL

In [ ]:
```
# ignoring the warning from pytorch lightning about the model file already existing
import warnings
warnings.filterwarnings("ignore")

# Initializing the trainer with the hyperparameters and training the model
model = train(model, train_dataloader, val_dataloader, val_check_interval, max_epoc
```

```
Using 16bit native Automatic Mixed Precision (AMP)
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
Running in `fast_dev_run` mode: will run the requested loop using 1 batch(es). Log
ging and checkpointing is suppressed.
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name      | Type             | Params
-----------------------------------------------
0 | encoder   | Encoder          | 1.6 M
1 | decoder   | Decoder          | 2.4 M
2 | criterion | CrossEntropyLoss | 0
-----------------------------------------------
4.0 M     Trainable params
0         Non-trainable params
4.0 M     Total params
8.065     Total estimated model params size (MB)
Training: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
`Trainer.fit` stopped: `max_steps=1` reached.
```

Visualizations

TensorBoard / Weights and Biases (Wandb)

Tensorboard is nice to track individual training runs and also visualizing the model graph. (https://www.tensorflow.org/tensorboard)

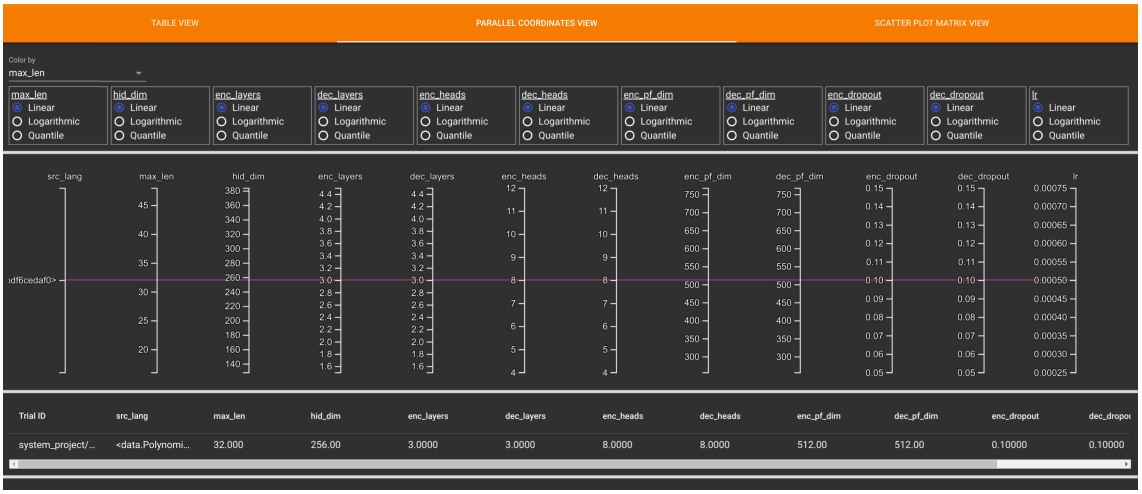
Wandb is a nice tool to track multiple training runs and compare the results. (https://wandb.ai/site)

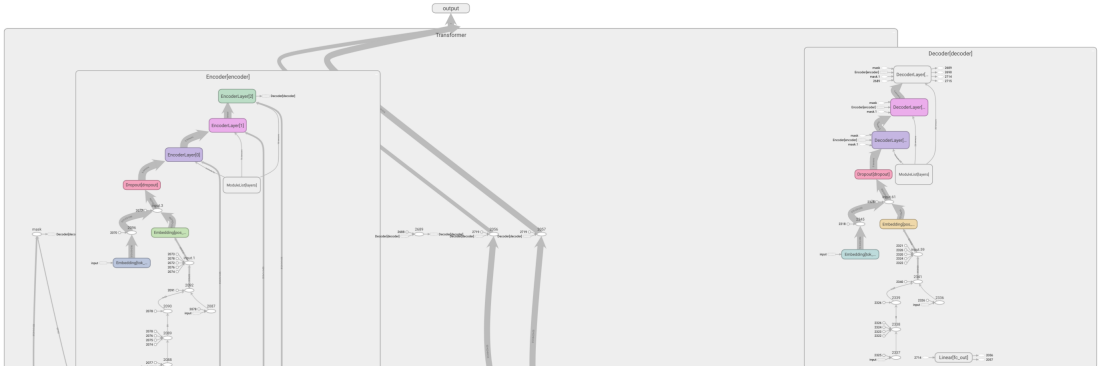
I found that being able to tangibly see the differences between the model training configurations in Wandb is more valuable, and I am able to share the results with others more easily.

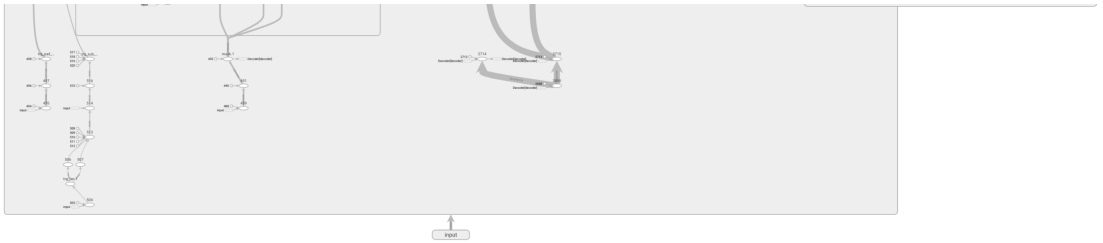
To view the interactive Wandb dashboard, click here: https://wandb.ai/chrisvaisnor/system_project?workspace=user-chrisvaisnor

Below are some screenshots from each dashboard.

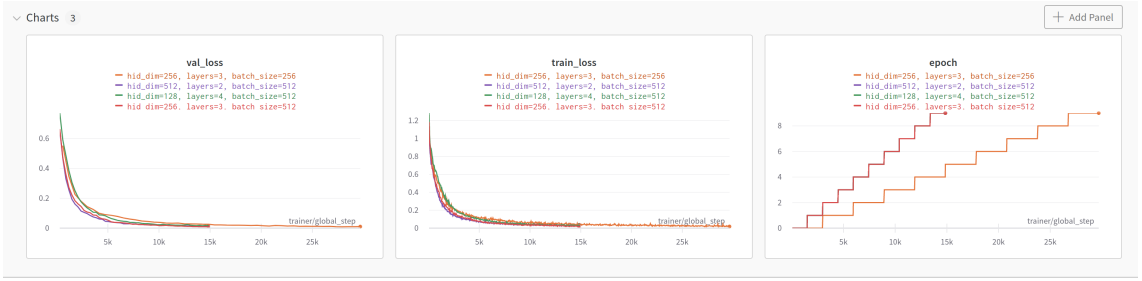
TensorBoard







Weights and Biases (Wandb)



Name (4 visualized)	Notes	User	Tags	Sweep	dec_dropou	dec_heads	dec_layers	dec_pf_dim	device	enc_dropou	enc_heads	enc_layers	enc_pf_dim	hid_dim
hid_dim=256, layers=3, batch_size=256	Add notes	chrisva		-	0.1	8	3	512	cuda	0.1	8	3	512	256
hid_dim=512, layers=2, batch_size=512	Add notes	chrisva		-	0.1	8	2	512	cuda	0.1	8	2	512	512
hid_dim=128, layers=4, batch_size=512	Add notes	chrisva		-	0.1	8	4	512	cuda	0.1	8	4	512	128
hid_dim=256, layers=3, batch_size=512	Add notes	chrisva		-	0.1	8	3	512	cuda	0.1	8	3	512	256

In [ ]:
```
test_set_pairs = PolynomialLanguage.load_pairs(test_set_path) # loading test set

# Loading the saved model checkpoint located in /models.
model_red = load_model(model_path, model_ckpt="model_red_default.ckpt") # default m
model_orange = load_model(model_path, model_ckpt="model_orange.ckpt") # for compari
```

Diagnosis/Evaluation

Default Model

For simplicity, the evaluate function calls the model.predict() function which transforms and batches the data, implicitly. This is in contrast to earlier in the notebook when the data processessing was shown explicitly. The saved model is passed into the function as a parameter, in addition to the test set pairs and batch size.

Depending on the hardware used, this cell can take between 5 minutes to over an hour. The test set is 200,000 lines long.

Scoring is based on if the model's prediction is exactly equal to the target.

A (1) is returned if the prediction is correct and a (0) is returned if the prediction is incorrect. The average of these values is the accuracy score.

In [ ]:
```
evaluate(model_red, test_set_pairs, batch_size=batch_size, print_examples=True)
```

```
creating prediction tensors: 100%|██████████| 200000/200000 [00:01<00:00, 134501.3
9it/s]
predict batch num: 100%|██████████| 391/391 [04:16<00:00,  1.52it/s]
scoring: 100%|██████████| 200000/200000 [00:00<00:00, 3127661.84it/s]
```

```
---- Test Case 0 ----
Input = -5*h*(5-2*h)
Target = 10*h**2-25*h
Predicted = 10*h**2-25*h
score = 1

---- Test Case 1 ----
Input = s*(8*s-21)
Target = 8*s**2-21*s
Predicted = 8*s**2-21*s
score = 1

---- Test Case 2 ----
Input = (21-t)*(-6*t-4)
Target = 6*t**2-122*t-84
Predicted = 6*t**2-122*t-84
score = 1

---- Test Case 3 ----
Input = (21-5*c)*(3*c-7)
Target = -15*c**2+98*c-147
Predicted = -15*c**2+98*c-147
score = 1

---- Test Case 4 ----
Input = 4*n*(n+22)
Target = 4*n**2+88*n
Predicted = 4*n**2+88*n
score = 1

---- Test Case 5 ----
Input = (k+2)*(5*k+29)
Target = 5*k**2+39*k+58
Predicted = 5*k**2+39*k+58
score = 1

---- Test Case 6 ----
Input = (k-15)*(2*k+29)
Target = 2*k**2-k-435
Predicted = 2*k**2+k-435
score = 0

---- Test Case 7 ----
Input = (i-20)*(i+24)
Target = i**2+4*i-480
Predicted = i**2+4*i-480
score = 1

---- Test Case 8 ----
Input = -4*c*(c+2)
Target = -4*c**2-8*c
Predicted = -4*c**2-8*c
score = 1

---- Test Case 9 ----
Input = -6*j*(-7*j-11)
Target = 42*j**2+66*j
Predicted = 42*j**2+66*j
score = 1
--------------------------------------------------
Number Correct:  177508
Number of Examples:  200000
Final Score = 0.8875
```

Out[ ]: 0.88754

Based on the current training configuration, the model is able to achieve a score of 88.7% on the test set! In addition, test case 6 was very close to correct.

In the future, a more specific evaluation metric could be used to determine how close the model's prediction is to the target. This would more accurately reflect the model's performance and especially help with natural language processing tasks.

Comparing performance with two different configurations

- Red = (hidden_dim=256, layers=3, batch_size=512) **DEFAULT**
- Orange = (hidden_dim=256, layers=3, batch_size=256)

In [ ]:
```
evaluate(model_red, test_set_pairs, batch_size=512, print_examples=False)
print('-'*100)
evaluate(model_orange, test_set_pairs, batch_size=256, print_examples=False)
```

```
creating prediction tensors: 100%|██████████| 200000/200000 [00:01<00:00, 128078.8
6it/s]
predict batch num: 100%|██████████| 391/391 [04:18<00:00,  1.51it/s]
scoring: 100%|██████████| 200000/200000 [00:00<00:00, 3233601.11it/s]
--------------------------------------------------
Number Correct:  177489
Number of Examples:  200000
Final Score = 0.8874
----------------------------------------------------------------------------------------------------
creating prediction tensors: 100%|██████████| 200000/200000 [00:01<00:00, 131581.8
0it/s]
predict batch num: 100%|██████████| 782/782 [04:34<00:00,  2.85it/s]
scoring: 100%|██████████| 200000/200000 [00:00<00:00, 3350149.96it/s]
--------------------------------------------------
Number Correct:  174827
Number of Examples:  200000
Final Score = 0.8741
```

Out[ ]:
```
0.874135
```

In the case of the two evaluations above, the models have a very similar accuracy score. The model that used a batch size of 512 was able to train and evaluate 5-10% faster.

The batch size is the number of training examples that are passed through the model at once. The larger the batch size, the more memory is required. There is a concensus that larger batch sizes can reduce model accuracy. The thought is that there is less opportunity for the model to learn from each example. In my case, increasing batch size did NOT reduce model accuracy. This is likely due to a number of factors.

- The dataset is large enough that the model has plenty of examples to learn from. If the dataset was under 100k lines, then accuracy may have been reduced.
- There is enough steps per epoch, and total epochs, to allow the model to converge to an accurate solution.

Conclusion and Takeaways

The Transformer is a powerful model for sequence-to-sequence tasks and this project was a great introduction to the model. Going in-depth into the model's architecture and the training process was a great learning experience. My skills in PyTorch have grown exponentially and I was introduced to PyTorch Lightning, TensorBoard, and Weights and Biases, all of which I will continue to use and develop with in the future.

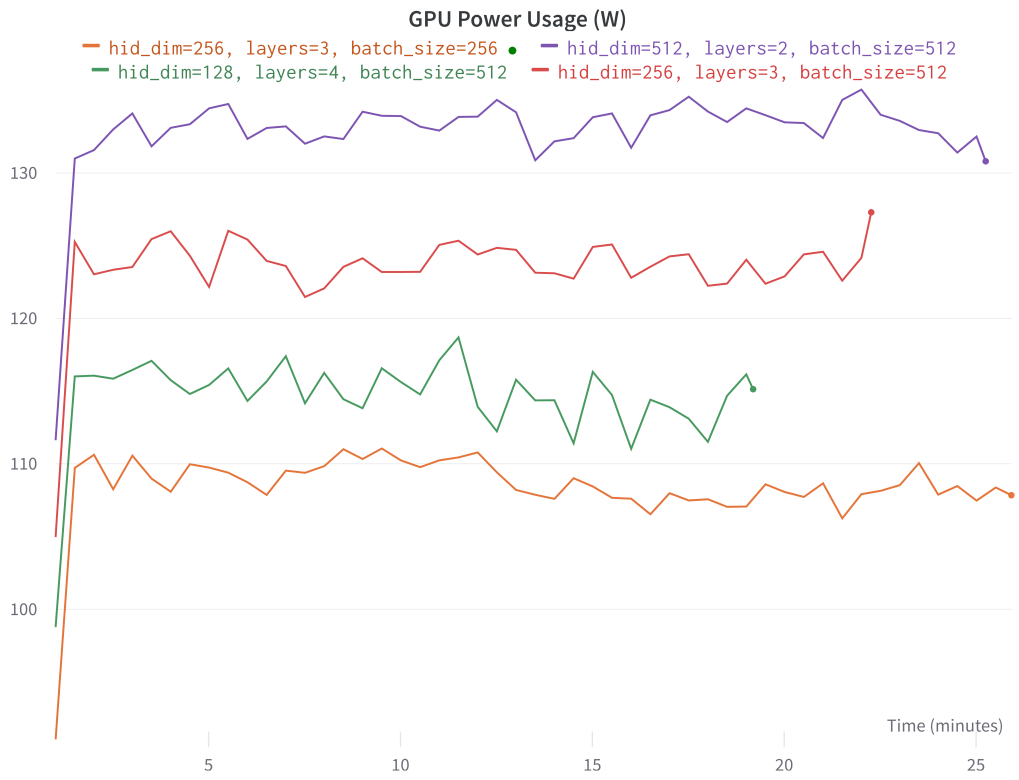
Taking a look at the system section of the Weights and Biases dashboard, I also learned how GPU wattage, utilization, and memory usage can be tracked.

To compare two of the training configurations, lets use the following two runs evaluated earlier:

- hid_dim=256, layers=3, batch_size=256 (orange)
- hid_dim=256, layers=3, batch_size=512 (red) (default)

Looking at the GPU power usage in watts and the training time in minutes, there are a few things to learn.

- The larger batch size (red) trained in 22 minutes and averaged 123 watts.
  - 22m x 123w = 162,360 joules
- The smaller batch size (orange) trained in 27 minutes and averaged 108 watts.
  - 27m x 108w = 174,960 joules



- This leads to a relative difference of 12.9% in power usage and 20.4% in training time.

In this case, having used a larger batch size, even if it included more VRAM and GPU watts, was cost effective. Because the data set was large enough, there was no penalty to accuracy. This is useful to know when training models in the future.

There is new software and hardware being developed every day and I am excited to see what the future holds for machine learning and artificial intelligence. This project has given me a backbone to

future holds for machine learning and artificial intelligence. This project has given me a backbone to build on and continously reference from. The next generation of transformer models, such as Generative Pre-Training Transformers (GPT) and Bidirecitonal Encoder Representations from Transformer (BERT), are now the cutting edge of natural language processing. With the foundational skills I have gained from this project, I am excited to dig deeper into these models and the future of NLP.

References

- Attention Is All You Need (Original Paper) https://arxiv.org/abs/1706.03762
- The Illustrated Transformer (Jay Lammar) http://jalammar.github.io/illustrated-transformer/
- The Annotated Transformer (Harvard Team) http://nlp.seas.harvard.edu/annotated-transformer/
- PyTorch https://pytorch.org/
- PyTorch Lightning https://www.pytorchlightning.ai/
- TensorBoard https://www.tensorflow.org/tensorboard
- Transformer Models (PyTorch) https://pytorch.org/tutorials/beginner/transformer_tutorial.html
- Mastering Transformers (O'Reilly Book) https://learning.oreilly.com/library/view/mastering-transformers/9781801077651/
- Mastering PyTorch (O'Reilly Book) https://learning.oreilly.com/library/view/mastering-pytorch/9781789614381/
- Transformers for Natural Language Processing (O'Reilly Book) https://learning.oreilly.com/library/view/transformers-for-natural/9781803247335/
- Seq-to-Seq Models https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
- PyTorch-seq2seq https://github.com/bentrevett/pytorch-seq2seq
- PyTorch Datasets and DataLoaders https://pytorch.org/tutorials/beginner/basics/data_tutorial.html
- Seq2Seq (https://github.com/jaymody/seq2seq-polynomial)
- TensorFlow Transformer Tutorial (Lilian Weng) https://github.com/lilianweng/transformer-tensorflow
- PyTorch Tutorials https://pytorch.org/tutorials/
- Cuda Toolkit https://developer.nvidia.com/cuda-toolkit
- Anaconda https://www.anaconda.org/