Transformers for Stock Price Prediction

ECE590 Final Project

Name: Javier Cervantes

net id: jc1010

Abstract

The motivation behind this experiment is to evaluate whether Transformers are more capable of predicting stock prices than traditional machine learning models. Given the Transformer's capabilities to utilize and derive context from the input, this experiment will consist in utilizing the simplest possible inputs: historical price data. This shall lay the foundations for future experiments that can incorporate sentiment, fundamental and technical analysis data.

After demonstrating that Transformers can perform gradient descent in their forward pass on linear and then on non-linear data, we proceeded to add enough complexity that would allow us to tackle the stock prediction problem. We compared the performance of the Transformer model with popular, powerful machine learning models such as the LSTM and XGBoost. The results showed that the Transformer model was able to outperform the LSTM and XGBoost models in predicting future stock prices.

Note that this paper isn't intended to provide a trading strategy or a state of the art prediction tool, but rather to demonstrate the capabilities of Transformers in predicting stock prices compared to other traditional models.

Introduction

To build up to our experiment, we shall first demonstrate how Transformers are able to perform gradient descent in their forward pass using linear self-attention on linear data. Upon successful completion of this task, we shall then add complexity to our model by introducing non linear self-attention mechanisms and evaluating on non-linear data. If this proves successful, we shall escalate the complexity of our model by introducing a MLP on top of the self-attention mechanism and evaluating on stock data.

• For the linear attention examples, we attempted to replicate the work done on this paper: https://arxiv.org/abs/2212.07677

• For the examples with softmax attention, we followed this paper: https://arxiv.org/abs/2208.01066

• Special thanks to `hkproj` for their lecture on building a Transformer from scratch: Lecture & Repo

Linear Self Attention

In this section, we shall demonstrate how a Transformer can perform gradient descent in its forward pass using linear self-attention on linear data.

Data

We implemented the following process for generating contextual data for a linear model: weights $w_m \in \mathbb{R}^{10}$ are drawn for context m as $w_m \sim \mathcal{N}(0_d, I_{10})$ where $\mu \in \mathbb{R}^{10}$ is a fixed mean vector. Covariates $x_i \in \mathbb{R}^{10}$ are drawn as $x_i \sim \mathcal{U}(-1, 1)$. For contextual data $\mathcal{C}_m$ draw one weight vector w_m as above. For a context of length N draw $x_{m,i}, i = 1, \ldots, N$ as above, and for each $x_{m,i}$ constitute a corresponding $y_{m,i} = w_m^T x_{m,i}$.

The contextual data so drawn are represented as $\mathcal{C}_m = (x_{m,1}, y_{m,1}, \ldots, x_{m,N}, y_{m,N})$. Finally, we draw a query associated with $\mathcal{C}_m, x_{m,N+1}$, and the goal is to predict $y_{m,N+1}$ given $x_{m,N+1}$ and $\mathcal{C}_m$.

Weight Initialization

Following the paper, we shall initialize the attention heads as follows:

- $W_Q = W_K = \begin{pmatrix} I_d & 0 \\ 0 & 0 \end{pmatrix}$
- $W_V = \begin{pmatrix} 0_{d \times d} & 0_{d \times 1} \\ 0_{1 \times d} & 1 \end{pmatrix}$
- $P = \begin{pmatrix} 0_{d \times d} & 0_{d \times 1} \\ 0_{1 \times d} & -\alpha \end{pmatrix}$

Where α is the learning rate.

In [ ]:
```
import warnings

warnings.filterwarnings("ignore")
```

In [ ]:
```
from scratch_transformer import MultiHeadAttentionBlock
from data import create_weights, get_reg_data, get_nonlinear_data
import numpy as np

feature_size = 10
output_size = 1
M = 10
N = 1000
lr = 1e-3
```

```
# linear attention params override
la_params = create_weights(feature_size, output_size, N, lr)

# get the data
eval_data = get_reg_data(no_tasks=M, feature_size=feature_size, no_examples=N)


# Create a MultiHeadAttentionBlock
mha = MultiHeadAttentionBlock(
    d_model=feature_size + 1, heads=1, dropout=0.0, softmax_att=False
)  # (batch_size, seq_len, d_model)
```

In [ ]:
```
def compute_loss(preds, targets):
    """Compute the MSE loss."""
    return 0.5 * np.sum((targets - preds) ** 2) / targets.shape[0]
```

Forward Pass

The forward pass of the Transformer model is as follows:

In [ ]:
```
import torch

e_eval = torch.tensor(eval_data[0]).float()

# Forward pass
out = mha(e_eval, e_eval, e_eval)

# Compare the output to the targets
eval_targets = eval_data[1][:, -1]
eval_preds = out[:, -1, -1] * (-1.0)
```

In [ ]:
```
loss = compute_loss(eval_preds.detach().numpy(), eval_targets)
print(f"Loss pre weight override for M: {M}, N: {N} is {loss:.3f}.")
```

```
Loss pre weight override for M: 10, N: 1000 is 15.427.
```

In [ ]:
```
# Now we will override the weights of the model to implement those that perform GD in th
def override_weights(model, new_params, w_name):
    w_name = "Transformer_gd/multi_head_attention/" + w_name
    w_numpy = new_params[w_name]["w"]
    w_tensor = torch.tensor(w_numpy, dtype=model.weight.dtype)
    model.weight.data = w_tensor


# Override the weights of the model
override_weights(mha.w_q, la_params, "query")
override_weights(mha.w_k, la_params, "key")
override_weights(mha.w_v, la_params, "value")
override_weights(mha.w_o, la_params, "linear")
```

In [ ]:
```
e_eval = torch.tensor(eval_data[0]).float()
```

```
# Forward pass
out = mha(e_eval, e_eval, e_eval)

# Compare the output to the targets
eval_targets = eval_data[1][:, -1]
eval_preds = out[:, -1, -1] * (-1.0)
```

In [ ]:
```
loss = compute_loss(eval_preds.detach().numpy(), eval_targets)
print(f"Loss for M: {M}, N: {N} is {loss:.3f}.")
```

```
Loss for M: 10, N: 1000 is 0.350.
```

Above, we performed two forward passes on our Self Attention layer:

1. With random initialization of the weights. This resulted in a very high loss.
2. With the weights initialized as designed to perform GD on the forward pass. The loss decreased substantially to 0.35

Learning the parameters

We shall proceed to perform a few steps of GD on the designed weights to see if we can achieve better performance.

In [ ]:
```
from tqdm import tqdm
from torch.optim.lr_scheduler import StepLR


def train(
    model,
    optimizer,
    criterion,
    eval_data=None,
    training_steps=1000,
    linear_data=False,
    model_type="attn",
    mask=None,
    stocks_train=None,
):
    """
    param model_type: str, "attn" or "transformer"
    """
    assert model_type in [
        "attn",
        "transformer",
    ], "model_type must be 'attn' or 'transformer'"
    if stocks_train is not None:
        assert eval_data is not None, "No stock evaluation data provided."
    eval_losses = []
    lowest_loss = 1e9

    # Move the model to device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
    model.to(device)
    print(f"Training on {device}.")

    # If using stock data, we're predicting 5 outomes
    if stocks_train is not None:
        no_outcomes = 5
    else:
        no_outcomes = 1

    # Get the evaluation data if it is not provided
    if eval_data is None:
        if linear_data:
            eval_data = get_reg_data(
                no_tasks=M, feature_size=feature_size, no_examples=N
            )
        else:
            eval_data = get_nonlinear_data(
                no_tasks=M, feature_size=feature_size, no_examples=N
            )
    assert eval_data is not None, "No evaluation data provided."
    e_eval = torch.tensor(eval_data[0]).float().to(device)
    eval_targets = (
        torch.tensor(eval_data[1][:, -no_outcomes:]).float().to(device)
    )  # change for stocks

    # Define lr scheduler
    scheduler = StepLR(optimizer, step_size=1000, gamma=0.5)

    for step in tqdm(range(training_steps + 1)):
        # Generate train data
        if stocks_train is not None:
            train_data = stocks_train
        elif linear_data:
            train_data = get_reg_data(
                no_tasks=M, feature_size=feature_size, no_examples=N
            )
        else:
            train_data = get_nonlinear_data(
                no_tasks=M, feature_size=feature_size, no_examples=N
            )
        e_train = torch.tensor(train_data[0]).float().to(device)
        targets = (
            torch.tensor(train_data[1][:, -no_outcomes:]).float().to(device)
        )  # change for stocks

        # Forward pass
        optimizer.zero_grad()
        if model_type == "attn":
            out = model(e_train, e_train, e_train, mask)
        else:
            out = model(e_train, mask)
        preds = out[:, -1, -no_outcomes:] * (-1.0)  # change for stocks
        loss = criterion(preds, targets)
        loss.backward()
```

```
        optimizer.step()
        scheduler.step()

        # Evaluate
        if step % 100 == 0:
            model.eval()
            with torch.no_grad():
                if model_type == "attn":
                    ev_preds = model(e_eval, e_eval, e_eval)
                else:
                    ev_preds = model(e_eval, None)  # no mask in evaluation mode
                ev_preds = ev_preds[:, -1, -no_outcomes:] * (-1.0)  # change for stocks
                eval_loss = criterion(ev_preds, eval_targets)
                eval_losses.append(eval_loss)
            model.train()
            if eval_loss < lowest_loss:
                lowest_loss = eval_loss
                if stocks_train is not None:
                    data_type = "stocks"
                elif linear_data:
                    data_type = "lin_data"
                else:
                    data_type = "nonlin_data"
                if model_type == "transformer":
                    att = "transformer"
                elif model.softmax_att:
                    att = "softmax_attn"
                else:
                    att = "linear_attn"
                path = f"models/{att}-{data_type}.pth"
                torch.save(model.state_dict(), path)
            print(f"Step {step}, Train Loss: {loss.item():.3f}")
            print(f"Step {step}, Eval Loss: {eval_loss:.3f}")
```

In [ ]:
```
# Now let's explore training the model
import torch.optim as optim

# Train
optimizer = optim.Adam(mha.parameters(), lr=lr)
criterion = torch.nn.MSELoss()

training_steps = 1000

train(
    mha,
    optimizer,
    criterion,
    eval_data=eval_data,
    training_steps=training_steps,
    linear_data=True,
    model_type="attn",
)
```

```
Training on cuda.
```

```
  2%|▌          | 16/1001 [00:00<00:33, 29.41it/s]
Step 0, Train Loss: 0.774
Step 0, Eval Loss: 0.526
 13%|▍          | 128/1001 [00:01<00:06, 136.61it/s]
Step 100, Train Loss: 0.152
Step 100, Eval Loss: 0.086
 22%|▊          | 224/1001 [00:02<00:05, 152.27it/s]
Step 200, Train Loss: 0.033
Step 200, Eval Loss: 0.039
 32%|▉          | 320/1001 [00:02<00:04, 151.25it/s]
Step 300, Train Loss: 0.033
Step 300, Eval Loss: 0.023
 42%|▋         | 417/1001 [00:03<00:03, 156.26it/s]
Step 400, Train Loss: 0.029
Step 400, Eval Loss: 0.024
 53%|▊        | 532/1001 [00:04<00:02, 159.83it/s]
Step 500, Train Loss: 0.019
Step 500, Eval Loss: 0.023
 63%|▍       | 629/1001 [00:04<00:02, 150.13it/s]
Step 600, Train Loss: 0.012
Step 600, Eval Loss: 0.021
 72%|▏      | 723/1001 [00:05<00:01, 148.19it/s]
Step 700, Train Loss: 0.010
Step 700, Eval Loss: 0.025
 82%|▍     | 816/1001 [00:05<00:01, 143.23it/s]
Step 800, Train Loss: 0.027
Step 800, Eval Loss: 0.036
 93%|▏    | 929/1001 [00:06<00:00, 149.03it/s]
Step 900, Train Loss: 0.008
Step 900, Eval Loss: 0.018
100%|▉   | 1001/1001 [00:07<00:00, 138.58it/s]
Step 1000, Train Loss: 0.007
Step 1000, Eval Loss: 0.026
```

From the previous results we can observe that after just 100 steps of GD, the loss dropped considerably to 0.086. After 600 steps, the model started to overfit.

Non-Linear Data

We shall proceed to perform the same experiment as above with a key modification: we shall introduce non-linear data.

Data

Now we consider contextual data $\mathcal{C}_m = (x_{m,1}, y_{m,1}, \ldots, x_{m,N}, y_{m,N})$ where in each case $y_{m,i} = f_{w_m}(x_{m,i}) = w_m^T x_{m,i}$, where each $w_m \sim \mathcal{N}(\mathbf{0}_d, I_d)$, where $d = 10$. This is as above, but now the manner with which $x_{m,i}$ are drawn is different: Consider two 10-dimensional real-valued

vectors: $v = (v_1, \ldots, v_{10})^T$ and $u = (u_1, \ldots, u_{10})^T$, where $v_j = cos(\frac{j\pi}{5})$ and $u_j = sin(\frac{j\pi}{5})$, for $j = 1, \ldots, 10$. Each $x_{m,i} = \alpha v + \beta u + \epsilon$, where $\alpha \sim \mathcal{N}(0, 1)$, $\beta \sim \mathcal{N}(0, 1)$, and $\epsilon = (\epsilon_1, \ldots, \epsilon_{10})^T$, with $\epsilon_j \sim \mathcal{N}(0, \frac{1}{100})$.

Weight Initialization

The first part of the experiment with non-linear data is to use the same weight construction as we did in the previous exercise. Once we proceed to use softmax attention, we shall drop this construct.

Forward Pass

The forward pass of the Transformer model is as follows:

In [ ]:
```
lr = 5e-4
# Let's do the same but with non linear data
eval_nl_data = get_nonlinear_data(no_tasks=M, feature_size=feature_size, no_examples=N)
e_eval_nl = torch.tensor(eval_nl_data[0]).float()

# Create a MultiHeadAttentionBlock
mha_nl = MultiHeadAttentionBlock(
    d_model=feature_size + 1, heads=1, dropout=0.0, softmax_att=False
)  # (batch_size, seq_len, d_model)

# Forward pass pre override
out_nl = mha_nl(e_eval_nl, e_eval_nl, e_eval_nl)

# Compare the output to the targets
eval_nl_targets = eval_nl_data[1][:, -1]
eval_nl_preds = out_nl[:, -1, -1] * (-1.0)

loss_nl = compute_loss(eval_nl_preds.detach().numpy(), eval_nl_targets)
print(f"Loss pre override for M: {M}, N: {N} is {loss_nl:.3f}.")

# Override the weights of the model
override_weights(mha_nl.w_q, la_params, "query")
override_weights(mha_nl.w_k, la_params, "key")
override_weights(mha_nl.w_v, la_params, "value")
override_weights(mha_nl.w_o, la_params, "linear")

# Forward pass
out_nl = mha_nl(e_eval_nl, e_eval_nl, e_eval_nl)

# Compare the output to the targets
eval_nl_targets = eval_nl_data[1][:, -1]
eval_nl_preds = out_nl[:, -1, -1] * (-1.0)

loss_nl = compute_loss(eval_nl_preds.detach().numpy(), eval_nl_targets)
print(f"Loss with GD weights for M: {M}, N: {N} is {loss_nl:.3f}.")
```

```
Loss pre override for M: 10, N: 1000 is 425.128.
Loss with GD weights for M: 10, N: 1000 is 0.432.
```

Quite surprisingly, using the constructed weights from the previous exercise, we were able to achieve a loss of 0.432 on the non-linear data. This suggests that linear self attention was able to capture some of the non-linear relationships in the data.

In the same manner as before, we shall proceed to perform a few steps of GD on the designed weights to see if we can achieve better performance.

In [ ]:
```python
lr = 5e-4
optimizer = optim.Adam(mha_nl.parameters(), lr=lr)
criterion = torch.nn.MSELoss()

training_steps = 3000

# Now let's explore training the model
train(
    mha_nl,
    optimizer,
    criterion,
    eval_data=eval_nl_data,
    training_steps=training_steps,
    linear_data=False,
)
```

```
Training on cuda.
  0%|          | 1/3001 [00:00<05:49,  8.59it/s]
Step 0, Train Loss: 0.641
Step 0, Eval Loss: 5.404
  3%|▌          | 103/3001 [00:12<04:49, 10.01it/s]
Step 100, Train Loss: 1.100
Step 100, Eval Loss: 1.027
  7%|▊          | 203/3001 [00:23<04:47,  9.74it/s]
Step 200, Train Loss: 1.471
Step 200, Eval Loss: 1.128
 10%|▉          | 303/3001 [00:35<04:35,  9.80it/s]
Step 300, Train Loss: 2.686
Step 300, Eval Loss: 1.232
 13%|█▏         | 402/3001 [00:47<04:45,  9.10it/s]
Step 400, Train Loss: 0.690
Step 400, Eval Loss: 1.062
 17%|█▋         | 503/3001 [00:58<04:03, 10.25it/s]
Step 500, Train Loss: 1.851
Step 500, Eval Loss: 0.996
 20%|█▉         | 602/3001 [01:16<04:07,  9.71it/s]
Step 600, Train Loss: 1.104
Step 600, Eval Loss: 1.753
 23%|██▏        | 701/3001 [01:31<10:57,  3.50it/s]
Step 700, Train Loss: 3.782
Step 700, Eval Loss: 1.005
```

```
 27%|██▊         | 803/3001 [01:46<03:46,  9.72it/s]
Step 800, Train Loss: 1.036
Step 800, Eval Loss: 1.150
 30%|███         | 902/3001 [01:58<03:44,  9.37it/s]
Step 900, Train Loss: 0.997
Step 900, Eval Loss: 1.434
 33%|███▎        | 1002/3001 [02:10<04:03,  8.21it/s]
Step 1000, Train Loss: 1.066
Step 1000, Eval Loss: 1.069
 37%|███▋        | 1102/3001 [02:22<05:31,  5.73it/s]
Step 1100, Train Loss: 0.867
Step 1100, Eval Loss: 1.165
 40%|████        | 1202/3001 [02:33<05:09,  5.82it/s]
Step 1200, Train Loss: 0.582
Step 1200, Eval Loss: 1.082
 43%|████▎       | 1302/3001 [02:44<03:01,  9.38it/s]
Step 1300, Train Loss: 0.462
Step 1300, Eval Loss: 1.036
 47%|████▋       | 1402/3001 [02:55<02:45,  9.66it/s]
Step 1400, Train Loss: 0.371
Step 1400, Eval Loss: 1.039
 50%|█████       | 1503/3001 [03:07<02:27, 10.14it/s]
Step 1500, Train Loss: 1.739
Step 1500, Eval Loss: 0.987
 53%|█████▎      | 1603/3001 [03:18<02:20,  9.98it/s]
Step 1600, Train Loss: 0.744
Step 1600, Eval Loss: 1.002
 57%|█████▋      | 1702/3001 [03:30<02:32,  8.51it/s]
Step 1700, Train Loss: 0.642
Step 1700, Eval Loss: 1.103
 60%|██████      | 1803/3001 [03:41<02:01,  9.85it/s]
Step 1800, Train Loss: 1.527
Step 1800, Eval Loss: 0.970
 63%|██████▎     | 1902/3001 [03:52<02:15,  8.11it/s]
Step 1900, Train Loss: 0.244
Step 1900, Eval Loss: 0.986
 67%|██████▋     | 2002/3001 [04:04<01:44,  9.59it/s]
Step 2000, Train Loss: 0.267
Step 2000, Eval Loss: 1.062
 70%|███████     | 2103/3001 [04:15<01:31,  9.78it/s]
Step 2100, Train Loss: 0.729
Step 2100, Eval Loss: 1.007
 73%|███████▎    | 2201/3001 [04:26<01:50,  7.23it/s]
Step 2200, Train Loss: 0.607
Step 2200, Eval Loss: 0.990
 77%|███████▋    | 2302/3001 [04:37<01:52,  6.21it/s]
Step 2300, Train Loss: 0.905
Step 2300, Eval Loss: 1.007
 80%|████████    | 2402/3001 [04:48<01:06,  9.06it/s]
```

```
Step 2400, Train Loss: 0.412
Step 2400, Eval Loss: 0.993
 83%|████████▎   | 2502/3001 [04:59<00:52,  9.57it/s]
Step 2500, Train Loss: 0.339
Step 2500, Eval Loss: 1.019
 87%|████████▋   | 2601/3001 [05:10<00:39, 10.08it/s]
Step 2600, Train Loss: 0.194
Step 2600, Eval Loss: 1.025
 90%|█████████   | 2702/3001 [05:21<00:29, 10.04it/s]
Step 2700, Train Loss: 1.356
Step 2700, Eval Loss: 0.966
 93%|█████████▎  | 2802/3001 [05:33<00:21,  9.15it/s]
Step 2800, Train Loss: 1.046
Step 2800, Eval Loss: 1.022
 97%|█████████▋  | 2902/3001 [05:44<00:10,  9.09it/s]
Step 2900, Train Loss: 1.530
Step 2900, Eval Loss: 0.982
100%|██████████| 3001/3001 [05:56<00:00,  8.43it/s]
Step 3000, Train Loss: 0.791
Step 3000, Eval Loss: 1.021
```

The results from the training is quite baffling: the model wasn't able to converge into a state that generalized better than the manual weight construction. It immediately left that local optima and never returned. I experimented with smaller learning rates, larger N but the results were the same.

Softmax Attention

We shall now proceed to implement the softmax attention mechanism and evaluate the model on the non-linear data.

In [ ]:
```python
# Finally let's use softmax attention
# Create a MultiHeadAttentionBlock
mha_nl_sa = MultiHeadAttentionBlock(
    d_model=feature_size + 1, heads=1, dropout=0.0, softmax_att=True
)  # (batch_size, seq_len, d_model)

optimizer = optim.Adam(mha_nl_sa.parameters(), lr=lr)
criterion = torch.nn.MSELoss()

training_steps = 1000

# Training the model
train(
    mha_nl_sa,
    optimizer,
    criterion,
    eval_data=None,
    training_steps=training_steps,
```

```
    linear_data=False,
)
```

```
Training on cuda.
  0%|          | 2/1001 [00:00<02:02,  8.15it/s]
Step 0, Train Loss: 1.214
Step 0, Eval Loss: 0.925
 10%|▉         | 102/1001 [00:11<01:43,  8.68it/s]
Step 100, Train Loss: 0.657
Step 100, Eval Loss: 0.816
 20%|██        | 203/1001 [00:22<01:19, 10.06it/s]
Step 200, Train Loss: 0.845
Step 200, Eval Loss: 0.781
 30%|███       | 302/1001 [00:34<01:20,  8.73it/s]
Step 300, Train Loss: 1.154
Step 300, Eval Loss: 0.767
 40%|████      | 402/1001 [00:45<01:41,  5.89it/s]
Step 400, Train Loss: 0.835
Step 400, Eval Loss: 0.775
 50%|█████     | 503/1001 [00:55<00:48, 10.26it/s]
Step 500, Train Loss: 0.927
Step 500, Eval Loss: 0.796
 60%|██████    | 602/1001 [01:06<00:40,  9.80it/s]
Step 600, Train Loss: 1.260
Step 600, Eval Loss: 0.743
 70%|███████   | 703/1001 [01:18<00:30,  9.85it/s]
Step 700, Train Loss: 0.980
Step 700, Eval Loss: 0.664
 80%|████████  | 802/1001 [01:29<00:21,  9.18it/s]
Step 800, Train Loss: 0.829
Step 800, Eval Loss: 0.620
 90%|█████████ | 902/1001 [01:41<00:11,  8.77it/s]
Step 900, Train Loss: 0.726
Step 900, Eval Loss: 0.647
100%|██████████| 1001/1001 [01:53<00:00,  8.86it/s]
Step 1000, Train Loss: 1.069
Step 1000, Eval Loss: 0.662
```

We can observe that the model was able to outperform the learning process of the linear self attention mechanism. The loss dropped to 0.62 before starting to show signs of overfitting. We shall now proceed to evaluate the same experiment using a full Transformer model.

Traditional Transformer

As a final task before moving onto stock prediction, we shall attempt to improve on the previous results using the full Transformer architecture. To our model, we add Layer Normalization, a Multi-Layer Perceptron, and Residual Connections. Note that we shall still continue to use a single attention head. The reason is because of our token construction: 10 dimensions + output = 11

input dimensions. This isn't divisible by any number of heads at the moment. We shall later proceed to make some modifications but we won't be able to directly compare with the previuos experiments once we do.

In [ ]:
```
from scratch_transformer import (
    LayerNormalization,
    FeedForwardBlock,
    ResidualConnection,
    EncoderBlock,
    MultiHeadAttentionBlock,
)
import torch
import torch.nn as nn
from data import get_nonlinear_data

feature_size = 10
output_size = 1
M = 10
N = 1000
lr = 1e-4
dropout = 0.2
mask = None

# get the data
eval_data = get_nonlinear_data(no_tasks=M, feature_size=feature_size, no_examples=N)
e = torch.tensor(eval_data[0]).float()


# MLP dimension usually 4 times the d_model
# Residual connection already contains layer normalizations

# Start with Self Attention
mha = MultiHeadAttentionBlock(
    d_model=feature_size + 1, heads=1, dropout=dropout, softmax_att=True
)  # (batch_size, seq_len, d_model)


# Feed Forward
ff = FeedForwardBlock(
    d_model=feature_size + 1, d_ff=4 * (feature_size + 1), dropout=dropout
)  # (batch_size, seq_len, d_model


# Create an EncoderBlock
eb = EncoderBlock(
    self_attention_block=mha,
    feed_forward_block=ff,
    dropout=dropout,
)
```

In [ ]:
```
training_steps = 1000
optimizer = optim.Adam(eb.parameters(), lr=lr)
criterion = torch.nn.MSELoss()
```

```python
total_params = sum(p.numel() for p in eb.parameters())
print(f"Total number of parameters: {total_params}")

# Training the model
train(
    eb,
    optimizer,
    criterion,
    eval_data=None,
    training_steps=training_steps,
    linear_data=False,
    model_type="transformer",
    mask=mask,
)
```

```
Total number of parameters: 1555
Training on cuda.
  0%|          | 2/1001 [00:01<07:23,  2.25it/s]
Step 0, Train Loss: 18.331
Step 0, Eval Loss: 4.704
 10%|▉         | 102/1001 [00:12<01:40,  8.91it/s]
Step 100, Train Loss: 3.360
Step 100, Eval Loss: 4.210
 20%|██        | 203/1001 [00:24<01:19, 10.01it/s]
Step 200, Train Loss: 2.359
Step 200, Eval Loss: 3.661
 30%|███       | 302/1001 [00:36<01:19,  8.80it/s]
Step 300, Train Loss: 29.277
Step 300, Eval Loss: 3.126
 40%|████      | 403/1001 [00:47<01:01,  9.72it/s]
Step 400, Train Loss: 8.752
Step 400, Eval Loss: 2.586
 50%|█████     | 502/1001 [00:59<00:55,  8.95it/s]
Step 500, Train Loss: 3.398
Step 500, Eval Loss: 2.059
 60%|██████    | 601/1001 [01:10<00:44,  8.97it/s]
Step 600, Train Loss: 8.046
Step 600, Eval Loss: 1.619
 70%|███████   | 701/1001 [01:22<00:31,  9.56it/s]
Step 700, Train Loss: 17.576
Step 700, Eval Loss: 1.217
 80%|████████  | 802/1001 [01:33<00:32,  6.04it/s]
Step 800, Train Loss: 9.904
Step 800, Eval Loss: 0.878
 90%|█████████ | 902/1001 [01:45<00:16,  5.97it/s]
Step 900, Train Loss: 2.503
Step 900, Eval Loss: 0.605
100%|██████████| 1001/1001 [01:56<00:00,  8.62it/s]
Step 1000, Train Loss: 1.350
Step 1000, Eval Loss: 0.442
```

We can quickly observe that the full architecture was able to achieve a loss 33% lower than the previous softmax attention mechanism in the same number of training steps and with a negligible increase in training time.

Given the successful implementation of different components of the Transformer architecture, we proceed to evaluate the model on stock data.

Stock Prediction

We shall now proceed to evaluate the Transformer model on stock data. We shall compare the performance of the Transformer model with popular, powerful machine learning models such as the LSTM and XGBoost.

Data

The SPY ETF is an exchange-traded fund that tracks the S&P 500 index. It is one of the most widely traded ETFs in the world and provides exposure to the U.S. stock market.

We gathered daily historical Open, High, Low, Close and 3 Month Implied Volatility data for the SPY ETF from May 2014 to March 2024.

Data Preprocessing

To begin with, I scaled the prices by 1/200 and performed a log transformation on the prices and implied volatility so that the inputs varied around zero.

We then created 1 day, 5 day, 10 day, 20 day Future Closing prices as well as 1 day Future Open price to serve as our target variables. We proceeded to create context windows $C_m = x_{m,1}, y_{m,1} \ldots x_{m,N}, y_{m,N}$ of size 230 days and an additional query $x_{m,N+1}$ to be used in an attempt to predict $y_{m,N+1}$. Note from this construct that this implies our goal is to predict 5 separate target variables. Otherwise, the construct is similar to that of the previous sections.

Since we're dealing with sequential data, we slide the context window across our entire dataset in order to generate our training and testing data. Using a context window instead of simply separating the entire dataset into two train & test pieces allows our model to learn from different contexts provided by the different market environments.

In addition to $y_{m,N+1}$ and with the purpose of avoiding lookahead bias, we proceeded to separate the following 20 days of each context C_m as the test set. The reason being that one of our target variables is the future 20 day Closing Price.

In [ ]:
```
import pandas as pd
from data import get_stock_data
```

```
path = "data/stocks.csv"
spy_train, spy_eval = get_stock_data(path, ticker="SPY")
```

Model

We now proceed to construct the Transformer model using the full architecture utilized in the last experiment. We now increase the number of attention heads to 5 and utilize a total of 12 layers. The total number of parameters increased 10x with respect to the previous implementation. To help manage the additional complexity, we shall implement a dropout rate of 20% and a Learning Rate Scheduler. Additionally, we shall allow 5,000 steps of training as I expect this model to take longer to train.

In [ ]:
```
from scratch_transformer import (
    LayerNormalization,
    FeedForwardBlock,
    ResidualConnection,
    EncoderBlock,
    MultiHeadAttentionBlock,
    Encoder,
)
import torch
import torch.nn as nn

feature_size = 10
output_size = 1
dropout = 0.2
mask = None
heads = 5
layers = 12

# convert to tensor
e = torch.tensor(spy_eval[0]).float()
et = torch.tensor(spy_train[0]).float()


# MLP dimension usually 4 times the d_model
# Residual connection already contains layer normalizations

# Start with Self Attention
mha = MultiHeadAttentionBlock(
    d_model=feature_size,
    heads=heads,
    dropout=dropout,
    softmax_att=True,
)  # (batch_size, seq_len, d_model)

# out = mha(e, e, e)
out = mha(et, et, et, mask)

# Feed Forward
ff = FeedForwardBlock(
```

```
    d_model=feature_size, d_ff=4 * feature_size, dropout=dropout
)  # (batch_size, seq_len, d_model


# Create an EncoderBlock
eb = EncoderBlock(
    self_attention_block=mha,
    feed_forward_block=ff,
    dropout=dropout,
)

encoder_blocks = []
for _ in range(layers):
    encoder_self_attention_block = MultiHeadAttentionBlock(
        d_model=feature_size, heads=heads, dropout=dropout, softmax_att=True
    )
    encoder_feed_forward_block = FeedForwardBlock(
        d_model=feature_size, d_ff=4 * feature_size, dropout=dropout
    )
    encoder_block = EncoderBlock(
        self_attention_block=encoder_self_attention_block,
        feed_forward_block=encoder_feed_forward_block,
        dropout=dropout,
    )
    encoder_blocks.append(encoder_block)

# Don't worry about Encoder, it's just predefined in scratch_transformer
decoder = Encoder(
    nn.ModuleList(encoder_blocks),
)

# out = decoder(et, mask)

for p in decoder.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)

# Compare the output to the targets
# eval_targets = spy_eval[1]
# eval_preds = out[:, -1, -5:] * (-1.0)

# loss = compute_loss(eval_preds.detach().numpy(), eval_targets)
# print(f"Loss is {loss:.3f}.")
```

In [ ]:
```
import torch.optim as optim

# Train
lr = 1e-3
training_steps = 5000
optimizer = optim.Adam(decoder.parameters(), lr=lr)
criterion = torch.nn.MSELoss()

total_params = sum(p.numel() for p in decoder.parameters())
print(f"Total number of parameters: {total_params}")
```

```python
# Training the model
train(
    decoder,
    optimizer,
    criterion,
    eval_data=spy_eval,
    training_steps=training_steps,
    linear_data=False,
    model_type="transformer",
    mask=mask,
    stocks_train=spy_train,
)
```

```
Total number of parameters: 15530
Training on cuda.
  0%|          | 2/5001 [00:00<05:31, 15.09it/s]
Step 0, Train Loss: 0.865
Step 0, Eval Loss: 0.544
  2%|▏         | 104/5001 [00:04<03:45, 21.76it/s]
Step 100, Train Loss: 0.074
Step 100, Eval Loss: 0.055
  4%|▍         | 203/5001 [00:08<03:40, 21.73it/s]
Step 200, Train Loss: 0.058
Step 200, Eval Loss: 0.045
  6%|▌         | 305/5001 [00:14<03:27, 22.65it/s]
Step 300, Train Loss: 0.040
Step 300, Eval Loss: 0.013
  8%|▊         | 404/5001 [00:18<03:17, 23.26it/s]
Step 400, Train Loss: 0.017
Step 400, Eval Loss: 0.013
 10%|▉         | 504/5001 [00:23<05:21, 13.98it/s]
Step 500, Train Loss: 0.019
Step 500, Eval Loss: 0.008
 12%|█▏        | 603/5001 [00:27<03:15, 22.52it/s]
Step 600, Train Loss: 0.016
Step 600, Eval Loss: 0.008
 14%|█▍        | 705/5001 [00:31<03:08, 22.80it/s]
Step 700, Train Loss: 0.011
Step 700, Eval Loss: 0.007
 16%|█▌        | 805/5001 [00:36<03:12, 21.80it/s]
Step 800, Train Loss: 0.008
Step 800, Eval Loss: 0.007
 18%|█▊        | 904/5001 [00:41<03:07, 21.83it/s]
Step 900, Train Loss: 0.011
Step 900, Eval Loss: 0.006
 20%|██        | 1003/5001 [00:45<02:59, 22.33it/s]
Step 1000, Train Loss: 0.010
Step 1000, Eval Loss: 0.006
 22%|██▏       | 1103/5001 [00:50<02:51, 22.70it/s]
```

```
Step 1100, Train Loss: 0.009
Step 1100, Eval Loss: 0.006
 24%|██▍       | 1205/5001 [00:54<02:59, 21.19it/s]
Step 1200, Train Loss: 0.008
Step 1200, Eval Loss: 0.006
 26%|██▌       | 1305/5001 [01:00<03:25, 17.95it/s]
Step 1300, Train Loss: 0.007
Step 1300, Eval Loss: 0.006
 28%|██▊       | 1405/5001 [01:04<02:46, 21.63it/s]
Step 1400, Train Loss: 0.016
Step 1400, Eval Loss: 0.005
 30%|███       | 1504/5001 [01:09<02:36, 22.31it/s]
Step 1500, Train Loss: 0.020
Step 1500, Eval Loss: 0.005
 32%|███▏      | 1603/5001 [01:14<02:30, 22.63it/s]
Step 1600, Train Loss: 0.006
Step 1600, Eval Loss: 0.005
 34%|███▍      | 1705/5001 [01:18<02:26, 22.43it/s]
Step 1700, Train Loss: 0.006
Step 1700, Eval Loss: 0.005
 36%|███▌      | 1803/5001 [01:22<03:01, 17.57it/s]
Step 1800, Train Loss: 0.016
Step 1800, Eval Loss: 0.007
 38%|███▊      | 1905/5001 [01:27<02:08, 24.10it/s]
Step 1900, Train Loss: 0.005
Step 1900, Eval Loss: 0.006
 40%|████      | 2004/5001 [01:31<02:04, 23.99it/s]
Step 2000, Train Loss: 0.012
Step 2000, Eval Loss: 0.005
 42%|████▏     | 2104/5001 [01:36<02:38, 18.25it/s]
Step 2100, Train Loss: 0.007
Step 2100, Eval Loss: 0.005
 44%|████▍     | 2203/5001 [01:40<02:02, 22.93it/s]
Step 2200, Train Loss: 0.006
Step 2200, Eval Loss: 0.006
 46%|████▌     | 2305/5001 [01:44<01:53, 23.68it/s]
Step 2300, Train Loss: 0.007
Step 2300, Eval Loss: 0.006
 48%|████▊     | 2405/5001 [01:49<01:49, 23.79it/s]
Step 2400, Train Loss: 0.005
Step 2400, Eval Loss: 0.006
 50%|█████     | 2504/5001 [01:53<01:45, 23.56it/s]
Step 2500, Train Loss: 0.005
Step 2500, Eval Loss: 0.006
 52%|█████▏    | 2603/5001 [01:58<02:18, 17.36it/s]
Step 2600, Train Loss: 0.005
Step 2600, Eval Loss: 0.006
 54%|█████▍    | 2705/5001 [02:03<01:38, 23.41it/s]
Step 2700, Train Loss: 0.004
Step 2700, Eval Loss: 0.006
```

```
 56%|█████     | 2804/5001 [02:07<01:33, 23.41it/s]
Step 2800, Train Loss: 0.005
Step 2800, Eval Loss: 0.005
 58%|█████▊    | 2902/5001 [02:12<02:20, 14.91it/s]
Step 2900, Train Loss: 0.008
Step 2900, Eval Loss: 0.006
 60%|██████    | 3005/5001 [02:16<01:22, 24.21it/s]
Step 3000, Train Loss: 0.004
Step 3000, Eval Loss: 0.006
 62%|██████    | 3104/5001 [02:21<01:21, 23.35it/s]
Step 3100, Train Loss: 0.005
Step 3100, Eval Loss: 0.006
 64%|██████▍   | 3205/5001 [02:25<01:16, 23.48it/s]
Step 3200, Train Loss: 0.004
Step 3200, Eval Loss: 0.005
 66%|██████    | 3304/5001 [02:30<01:11, 23.66it/s]
Step 3300, Train Loss: 0.003
Step 3300, Eval Loss: 0.006
 68%|██████▊   | 3403/5001 [02:34<01:08, 23.32it/s]
Step 3400, Train Loss: 0.005
Step 3400, Eval Loss: 0.006
 70%|███████   | 3504/5001 [02:39<01:02, 23.88it/s]
Step 3500, Train Loss: 0.006
Step 3500, Eval Loss: 0.006
 72%|███████   | 3603/5001 [02:43<00:59, 23.61it/s]
Step 3600, Train Loss: 0.005
Step 3600, Eval Loss: 0.005
 74%|███████   | 3703/5001 [02:47<01:12, 17.88it/s]
Step 3700, Train Loss: 0.004
Step 3700, Eval Loss: 0.005
 76%|███████   | 3806/5001 [02:52<00:50, 23.87it/s]
Step 3800, Train Loss: 0.005
Step 3800, Eval Loss: 0.005
 78%|███████▊  | 3905/5001 [02:56<00:51, 21.12it/s]
Step 3900, Train Loss: 0.004
Step 3900, Eval Loss: 0.005
 80%|████████  | 4003/5001 [03:02<00:44, 22.47it/s]
Step 4000, Train Loss: 0.006
Step 4000, Eval Loss: 0.005
 82%|████████  | 4105/5001 [03:07<00:38, 23.23it/s]
Step 4100, Train Loss: 0.003
Step 4100, Eval Loss: 0.005
 84%|████████▍ | 4204/5001 [03:12<00:46, 17.08it/s]
Step 4200, Train Loss: 0.003
Step 4200, Eval Loss: 0.005
 86%|████████▌ | 4305/5001 [03:16<00:29, 23.42it/s]
Step 4300, Train Loss: 0.003
Step 4300, Eval Loss: 0.005
 88%|████████▊ | 4404/5001 [03:20<00:25, 23.43it/s]
```

```
Step 4400, Train Loss: 0.003
Step 4400, Eval Loss: 0.005
 90%|████████▉ | 4505/5001 [03:25<00:25, 19.23it/s]
Step 4500, Train Loss: 0.004
Step 4500, Eval Loss: 0.005
 92%|█████████▏ | 4604/5001 [03:29<00:16, 23.66it/s]
Step 4600, Train Loss: 0.003
Step 4600, Eval Loss: 0.006
 94%|█████████▍ | 4705/5001 [03:34<00:12, 23.39it/s]
Step 4700, Train Loss: 0.004
Step 4700, Eval Loss: 0.005
 96%|█████████▌ | 4805/5001 [03:39<00:08, 23.63it/s]
Step 4800, Train Loss: 0.007
Step 4800, Eval Loss: 0.005
 98%|█████████▊ | 4904/5001 [03:43<00:04, 22.70it/s]
Step 4900, Train Loss: 0.004
Step 4900, Eval Loss: 0.005
100%|██████████| 5001/5001 [03:47<00:00, 21.94it/s]
Step 5000, Train Loss: 0.002
Step 5000, Eval Loss: 0.006
```

Note that the model begins to show signs of overfitting around step 4,500. Given that we're utilizing log-transformed prices, looking at the Eval Loss by itself doesn't provide much information. We shall now proceed to fit the LSTM and XGBoost models to the same data and compare the results.

In [ ]:
```python
# load the transformer-stocks.pth model
decoder.load_state_dict(torch.load("models/transformer-stocks.pth"))
criterion = torch.nn.MSELoss()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
decoder.to(device)
# Evaluate the model
decoder.eval()
with torch.no_grad():
    e_eval = torch.tensor(spy_eval[0]).float()
    eval_preds = decoder(e_eval.to(device), None)[:, -1, -5:] * (-1.0)
    eval_targets = torch.tensor(spy_eval[1]).float()
    eval_loss = criterion(eval_preds, eval_targets.to(device))

print(f"Eval Loss: {eval_loss:.3f}")
```

```
Eval Loss: 0.005
```

Data pre-processing

In order to fit the LSTM and XGBoost models, we shall construct train and test data in the traditional construct of machine learning models. Note that this implies that the dimensions of the input and our testing data will now be 5 each.

In [ ]:
```python
# Now we change the inputs to traditional x, y
train = spy_train[0].reshape(-1, 10)
test = spy_eval[0].reshape(-1, 10)

x_train = train[:, :5]
y_train = train[:, 5:]

x_test = test[:, :5]
y_test = test[:, 5:]
```

XGBoost

We begin by fitting an XGBoost Regressor and calculating its MSE.

In [ ]:
```python
# compare to XGBoost
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error

# Create the model
model = XGBRegressor(
    n_estimators=10000,
    max_depth=100,
    learning_rate=0.01,
    objective="reg:squarederror",
    n_jobs=-1,
)

# Fit the model
model.fit(x_train, y_train)

# Make predictions
y_pred = model.predict(x_test)

# Calculate the MSE
mse = mean_squared_error(y_test, y_pred)
print(f"XGBoost Test MSE: {mse:.3f}")
```

```
XGBoost Test MSE: 0.013
```

As we can observe, at 0.013 the loss is considerably higher than the 0.005 achieved by the Transformer. As a first result, this is very encouraging towards evaluating our initial hypothesis.

We now proceed to train and evaluate an LSTM on the same data.

LSTM

Similar to the traditional construct of the MLP in the Transformer, we shall choose a hidden size that is 4x the input's dimensions. Additionally, we have chosen the same number of layers as that which we used in the Transformer.

In [ ]:
```python
import torch
import torch.nn as nn

feature_size = 5
output_size = 1
hidden_size = 20
num_layers = 12
dropout = 0.2


class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(
            input_size, hidden_size, num_layers, batch_first=True, dropout=dropout
        )
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.unsqueeze(1)
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)

        out, _ = self.lstm(x, (h0, c0))

        out = self.fc(out[:, -1, :])
        return out


model = LSTMModel(feature_size, hidden_size, num_layers, output_size)

# Initialize the weights of the model
for name, param in model.named_parameters():
    if "bias" in name:
        nn.init.constant_(param, 0.0)
    elif "weight" in name:
        nn.init.xavier_uniform_(param)
```

In [ ]:
```python
from torch.utils.data import DataLoader, TensorDataset
import torch.optim as optim

# Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Training on {device}.")


# Convert the data to tensors
x_train_tensor = torch.tensor(x_train).float().to(device)
y_train_tensor = torch.tensor(y_train).float().to(device)
x_test_tensor = torch.tensor(x_test).float().to(device)
y_test_tensor = torch.tensor(y_test).float().to(device)
```

```
# Create a DataLoader
train_data = TensorDataset(x_train_tensor, y_train_tensor)
train_loader = DataLoader(dataset=train_data, batch_size=64, shuffle=False)

# Define a loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters())

model.to(device)

best_val_loss = 1e9
num_epochs = 1000
# Training loop
for epoch in range(num_epochs):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()

    # Print every 100 epochs
    if epoch % 100 == 0:
        model.eval()
        with torch.no_grad():
            val_outputs = model(x_test_tensor)
            val_loss = criterion(val_outputs, y_test_tensor)
        print(
            f"Epoch {epoch + 1}, Training loss: {running_loss / len(train_loader)}, Vali
        )
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict(), "models/lstm-stocks.pth")
        model.train()

print("Finished Training")
```

```
Training on cuda.
Epoch 1, Training loss: 0.2050805888981719, Validation loss: 0.10673071444034576
Epoch 101, Training loss: 0.02208308392035743, Validation loss: 0.02183053269982338
Epoch 201, Training loss: 0.008579795744575175, Validation loss: 0.017444998025894165
Epoch 301, Training loss: 0.00848640529097636, Validation loss: 0.01739390194416046
Epoch 401, Training loss: 0.007310390781514861, Validation loss: 0.019773859530687332
Epoch 501, Training loss: 0.006472167184632593, Validation loss: 0.016421332955360413
Epoch 601, Training loss: 0.005802895442245464, Validation loss: 0.019048817455768585
Epoch 701, Training loss: 0.006011815845026988, Validation loss: 0.02138584852218628
Epoch 801, Training loss: 0.005568012486790018, Validation loss: 0.016140814870595932
Epoch 901, Training loss: 0.005295451493644765, Validation loss: 0.01866176165640354
Finished Training
```

Somewhat underwhelming results from the LSTM: with a loss of 0.016, it was the worst performer of our 3 experiments.

Conclusion

Overall I consider the results to be very encouraging when contemplating whether Transformers can belong in an investment advisor's toolkit.

Of first note, the features utilized in this experiment are as simple as can be. I argue that this was done by construction in order to demonstrate the capabilities of a Transformer to extract and design meaningful features and lay a solid foundation for future work. As it exists today, there is a plethora of financial engineering literature dedicated to creating signals that allow traders to gain the slightest edge in the market. The next step in the evolution of this experiment is to start integrating such features.

Another potentially exciting avenue to explore is the utilization of sentiment analysis data. With the advent of LLMs, literature and tools on sentiment analysis has exploded. Given the Transformer's capacity to gather context from features, I think it could be immensely valuable to integrate sentiment related features to the model.

References

- von Oswald, J., Niklasson, E., Randazzo, E., Sacramento, J., Mordvintsev, A., Zhmoginov, A., & Vladymyrov, M. (2023). Transformers learn in-context by gradient descent. arXiv:2212.07677v2. https://doi.org/10.48550/arXiv.2212.076772
- Garg, S., Tsipras, D., Liang, P., & Valiant, G. (2023). What Can Transformers Learn In-Context? A Case Study of Simple Function Classes. arXiv:2208.01066v3. https://doi.org/10.48550/arXiv.2208.01066
- Gruver, N., Finzi, M., Qiu, S., & Wilson, A. G. (2023). Large Language Models Are Zero-Shot Time Series Forecasters arXiv preprint arXiv:2310.078201. Retrieved from https://doi.org/10.48550/arXiv.2310.07820

- Karpathy, Andrej (2022). nanoGPT, The simplest, fastest repository for training/finetuning medium-sized GPTs. https://github.com/karpathy/nanoGPT/tree/master?tab=MIT-1-ov-file#readme
- hkproj (2023). Coding a Transformer from scratch on PyTorch. Attention is all you need implementation. https://github.com/hkproj/pytorch-transformer?tab=readme-ov-file