

## ✓ Hands On #9

### RNNs, Attention, and Transformers

File name convention: For group 42 and members Richard Stallman and Linus

Torvalds it would be

"09\_group42\_Stallman\_Torvalds.pdf".

Submission via blackboard (UA).

Feel free to answer free text questions in text cells using markdown and possibly *L<sup>A</sup>T<sub>E</sub>X* if you want to.

**You don't have to understand every line of code here and it is not intended for you to try to understand every line of code.**

**Big blocks of code are usually meant to just be clicked through.**

```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

import torch
from torch import nn
from torch.utils.data import DataLoader, Dataset
import torchvision

import numpy as np
import os

np.random.seed(42)
torch.manual_seed(42)
torch.cuda.manual_seed_all

%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10)) # wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5) # + noise
    return series[:, np.newaxis].astype(np.float32)

np.random.seed(42)

n_steps = 50
total_samples = 100_000
series = generate_time_series(total_samples, n_steps + 1)

# Split the dataset:
X_train, y_train = series[:80_000, :n_steps], series[:80_000, -1]
X_valid, y_valid = series[80_000:90_000, :n_steps], series[80_000:90_000, -1]
X_test, y_test = series[90_000:, :n_steps], series[90_000:, -1]
```

## ✓ Loading and Preparing the Dataset

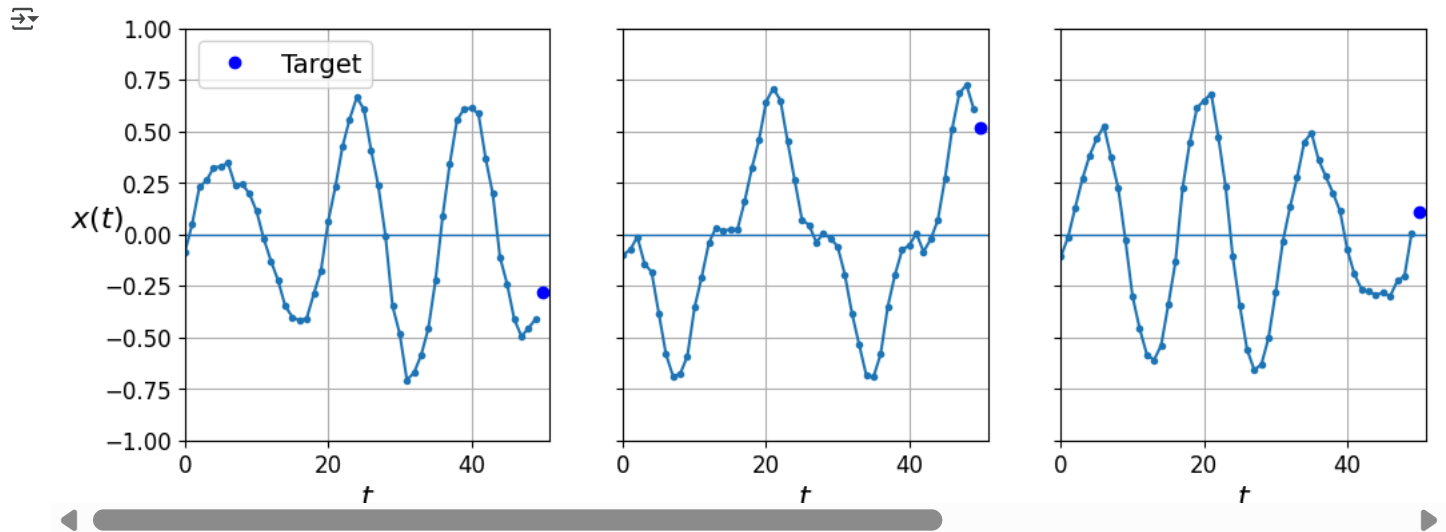
```
X_train.shape, y_train.shape
```

```
((80000, 50, 1), (80000, 1))
```

```
def plot_series(series, y=None, y_pred=None, x_label="$t$", y_label="$x(t)$", legend=True):
    plt.plot(series, "-")
    if y is not None:
        plt.plot(n_steps, y, "bo", label="Target")
    if y_pred is not None:
        plt.plot(n_steps, y_pred, "rx", markersize=10, label="Prediction")
    plt.grid(True)
    if x_label:
        plt.xlabel(x_label, fontsize=16)
    if y_label:
        plt.ylabel(y_label, fontsize=16, rotation=0)
    plt.hlines(0, 0, 100, linewidth=1)
    plt.axis([0, n_steps + 1, -1, 1])
    if legend and (y or y_pred):
        plt.legend(fontsize=14, loc="upper left")
```

```
fig, axes = plt.subplots(nrows=1, ncols=3, sharey=True, figsize=(12, 4))
for col in range(3):
    plt.sca(axes[col])
    plot_series(X_valid[col, :, 0], y_valid[col, 0],
                y_label=("$x(t)$" if col==0 else None),
                legend=(col == 0))
```

```
plt.show()
```



```
class TimeSeries(Dataset):
    def __init__(self, X, y):
        self.X = torch.from_numpy(X.copy()).float()
        self.y = torch.from_numpy(y.copy()).float()
    def __len__(self):
        return len(self.X)
    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

```
train_data = TimeSeries(X_train, y_train)
valid_data = TimeSeries(X_valid, y_valid)
test_data = TimeSeries(X_test, y_test)
```

```
batch_size = 16
```

```
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_data, batch_size=1, shuffle=False)
valid_loader = DataLoader(valid_data, batch_size=batch_size, shuffle=False)
```

```
def train_and_validate(train_loader, val_loader, model, optimizer, criterion, num_epochs, metric=None, scheduler=None, device='cpu'):
    history = {
        'epoch': [],
        'train_loss': [],
        'train_metric': [],
```

```

        'val_loss': [],
        'val_metric': [],
        'learning_rate': []
    } # Initialize a dictionary to store epoch-wise results

model.to(device) # Move the model to the specified device

for epoch in range(num_epochs):
    model.train() # Set the model to training mode
    epoch_loss = 0.0 # Initialize the epoch loss and metric values
    epoch_metric = 0.0

    # Training loop
    for X, y in train_loader:
        X = X.to(device)
        y = y.to(device)
        y = y
        optimizer.zero_grad() # Clear existing gradients
        outputs = model(X) # Make predictions
        loss = criterion(outputs, y) # Compute the loss
        loss.backward() # Compute gradients
        optimizer.step() # Update model parameters

        epoch_loss += loss.item()

    # THESE LINES HAVE BEEN UPDATED TO ACCOUNT FOR DEFAULT ARGUMENTS
    if metric is not None:
        epoch_metric += metric(outputs, y)
    else:
        epoch_metric += 0.0

    # Average training loss and metric
    epoch_loss /= len(train_loader)
    epoch_metric /= len(train_loader)

    # Validation loop
    model.eval() # Set the model to evaluation mode
    with torch.no_grad(): # Disable gradient calculation
        val_loss = 0.0
        val_metric = 0.0
        for X_val, y_val in val_loader:
            X_val = X_val.to(device)
            y_val = y_val.to(device)
            y_val = y_val
            outputs_val = model(X_val) # Make predictions
            val_loss += criterion(outputs_val, y_val).item() # Compute loss
            if metric is not None:
                val_metric += metric(outputs_val, y_val)
            else:
                val_metric += 0.0

        val_loss /= len(val_loader)
        val_metric /= len(val_loader)

    # Append epoch results to history
    history['epoch'].append(epoch)
    history['train_loss'].append(epoch_loss)
    history['train_metric'].append(epoch_metric)
    history['val_loss'].append(val_loss)
    history['val_metric'].append(val_metric)
    history['learning_rate'].append(optimizer.param_groups[0]['lr'])

    print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {epoch_loss:.4f}, '
          f'Train Metric: {epoch_metric:.4f}, Val Loss: {val_loss:.4f}, '
          f'Val Metric: {val_metric:.4f}')

    if scheduler is not None:
        scheduler.step()

return history, model

def test_model(model, data_loader, criterion, metric=None, device='cpu'):
    model.to(device) # Move the model to the specified device

    model.eval() # Set the model to evaluation mode

```

```

total_loss = 0.0    # Initialize the total loss and metric values
total_metric = 0.0

with torch.no_grad():    # Disable gradient tracking
    for batch in data_loader:
        X, y = batch
        X = X.to(device)
        y = y.to(device)
        # Pass the data to the model and make predictions
        outputs = model(X)

        # Compute the loss
        loss = criterion(outputs, y)

        # Add the loss and metric for the batch to the total values
        total_loss += loss.item()

        if metric is not None:
            total_metric += metric(outputs, y)
        else:
            total_metric += 0.0

# Average loss and metric for the entire dataset
avg_loss = total_loss / len(data_loader)
avg_metric = total_metric / len(data_loader)

print(f'Test Loss: {avg_loss:.4f}, Test Metric: {avg_metric:.4f}')

return avg_loss, avg_metric

```

## Recurrent Neural Networks

### ✓ The Simple Recurrent Neural Network RNN

The simplest form of **RNN** one can build **takes an output and passes it into the next input**. The idea is that, if your data has some form of **sequential** property such as being elements of a **time series** or a sentence or even generally related (not explicitly sequential) values, you should attempt to **inform your model via explicit neural connections** that those values are linked. By passing the output of one neuron to the next neuron in a sequence, you can explicitly inform your model of the relationship of an input to its neighboring inputs.

### ✓ Task 1: Simple RNN

Build a simple RNN using the `SimpleRNN` module. Your model should be a **Sequential model called `simplernn` of the following form:**

```
input_size=1, hidden_size=16, output_size=1, seq_len=50
```

Train the model using the `train_and_validate` function on `train_loader` and `valid_loader` using `MSELoss` as criterion, `L1Loss` as metric, and the Adam optimizer with `lr=0.001`. Train for 6 epochs.

↓↓ your code goes below

```

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, seq_len):
        super(SimpleRNN, self).__init__()
        self.rnn1 = nn.RNN(input_size, hidden_size, batch_first=True)
        self.rnn2 = nn.RNN(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size * seq_len, output_size)

    def forward(self, x):
        x, hn = self.rnn1(x)
        x, _ = self.rnn2(x, hn)
        x = x.reshape(-1, x.size(1) * x.size(2))

```

```
simplernn = SimpleRNN (input_size=1, hidden_size=16, output_size=1, seq_len=50)
criterion= nn.MSELoss()
metric = nn.L1Loss()
optimizer = torch.optim.Adam(simplernn.parameters(), lr=0.001)
```

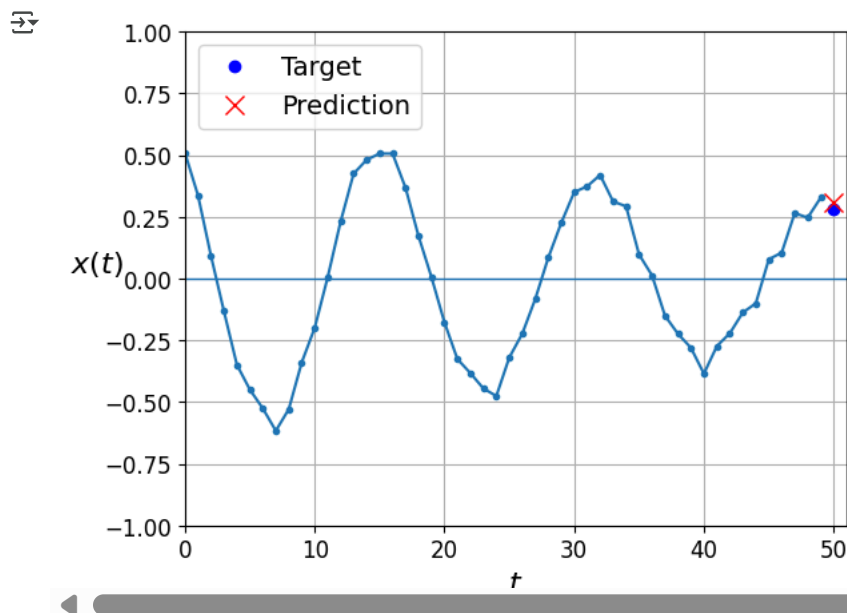
Epoch	Train Loss:	Train Metric:	Val Loss:	Val Metric:
Epoch [1/6],	Train Loss: 0.0046,	Train Metric: 0.0506,	Val Loss: 0.0032,	Val Metric: 0.0457
Epoch [2/6],	Train Loss: 0.0032,	Train Metric: 0.0459,	Val Loss: 0.0030,	Val Metric: 0.0444
Epoch [3/6],	Train Loss: 0.0032,	Train Metric: 0.0455,	Val Loss: 0.0035,	Val Metric: 0.0480
Epoch [4/6],	Train Loss: 0.0031,	Train Metric: 0.0452,	Val Loss: 0.0031,	Val Metric: 0.0447
Epoch [5/6],	Train Loss: 0.0031,	Train Metric: 0.0450,	Val Loss: 0.0031,	Val Metric: 0.0447
Epoch [6/6],	Train Loss: 0.0031,	Train Metric: 0.0449,	Val Loss: 0.0031,	Val Metric: 0.0447

```
test_model(simplernn, test_loader, criterion, metric=metric, device=device)
```

```
def plot_learning_curves(loss, val_loss):
    plt.plot(np.arange(len(loss)) + 0.5, loss, "b.-", label="Training loss")
    plt.plot(np.arange(len(val_loss)) + 1, val_loss, "r.-", label="Validation loss")
    plt.gca().xaxis.set_major_locator(mpl.ticker.MaxNLocator(integer=True))
    plt.axis([1, 10, 0.002, 0.01])
    plt.legend(fontsize=14)
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.grid(True)
```

Epoch	Training loss	Validation loss
1	0.0040	0.0032
2	0.0032	0.0030
3	0.0032	0.0035
4	0.0031	0.0030
5	0.0031	0.0030
6	0.0031	0.0030

```
y_pred = simplernn(torch.from_numpy(X_test).to(device)).to('cpu').detach().numpy()
plot_series(X_test[0, :, 0], y_test[0, 0], y_pred[0, 0])
plt.show()
```



```
print(y_pred.shape)
```

```
(10000, 1)
```

```
np.random.seed(43) # not 42, as it would give the first series in the train set
```

```
new_series = generate_time_series(1, n_steps + 10)
```

```
X_new, Y_new = new_series[:, :n_steps], new_series[:, n_steps:]
```

```
X = X_new
```

```
for step_ahead in range(10):
```

```
    y_pred_one = simplernn(torch.from_numpy(X[:, step_ahead:]).to(device)).to('cpu').detach().numpy()[0, np.newaxis, :]
```

```
    X = np.concatenate([X, y_pred_one], axis=1)
```

```
y_pred_new = X[:, n_steps:]
```

```
y_pred_new.shape
```

```
(1, 10, 1)
```

```
def plot_multiple_forecasts(X, Y, Y_pred):
```

```
    n_steps = X.shape[1]
```

```
    ahead = Y.shape[1]
```

```
    plot_series(X[0, :, 0])
```

```
    plt.plot(np.arange(n_steps, n_steps + ahead), Y[0, :, 0], "bo-", label="Actual")
```

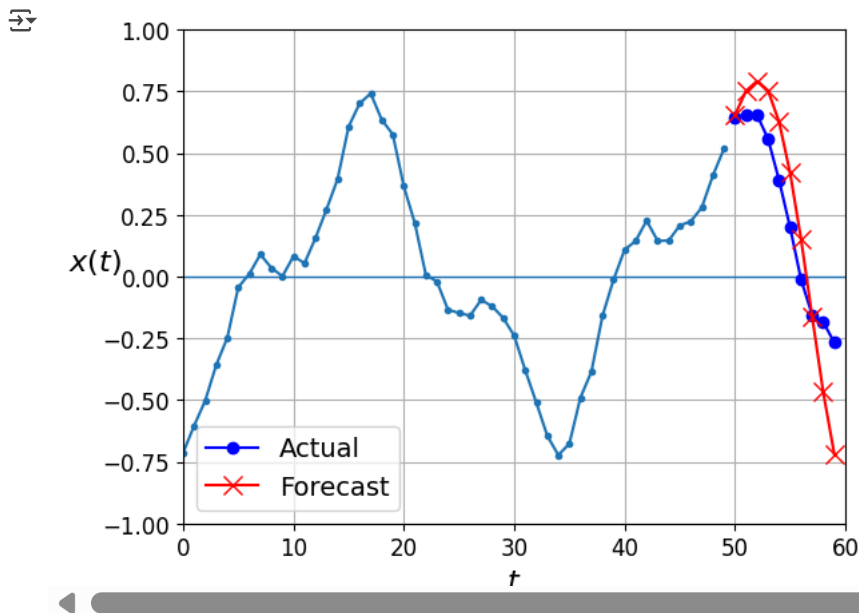
```
    plt.plot(np.arange(n_steps, n_steps + ahead), Y_pred[0, :, 0], "rx-", label="Forecast", markersize=10)
```

```
    plt.axis([0, n_steps + ahead, -1, 1])
```

```
    plt.legend(fontsize=14)
```

```
plot_multiple_forecasts(X_new, Y_new, y_pred_new)
```

```
plt.show()
```



## ✓ The Gated Recurrent Unit (GRU)

GRU models are **actually very new** in the timeline of RNNs having been first **proposed in 2014** by Kyunghyun Cho et al. The main feature that this adds to the RNN architecture is that it replaces the simple densely connected unit with a new unit that adds a "**forget gate**". You might remember the **characteristic equation for a standard neuron** is  $a_i = \sigma(w_i x_i + b_i)$ . For the GRU you add a term which depends on a new weight we can call u which is applied to the output of the previous neuron:  $a_i = \sigma(w_i x_i + u_i h_{i-1} + b_i)$  where h is the final output of the previous neuron. Additionally, it takes that initial activation and applies two steps:

- 1)  $\hat{h}_i = \phi(w_{2i} x_i + u_{2i}(a_i \odot h_{i-1}) + b_{2i})$  where  $\phi$  is tanh and  $w_2$  is a second neuron weight meaning that the simplest **GRU will have 4 weights and 2 biases**.
- 2)  $h_i = (1 - a_i) \odot h_{i-1} + a_i \odot \hat{h}_i$

If we consider what those equations mean then you can probably work out that **(1), our forget gate**, is checking how correlated our activation is with our previous neuron's activation. Then, **if our output is highly correlated**, our final activation **(2) will have a large contribution from our previous neuron** and if uncorrelated, it will have a small contribution.

The modern standard implementation of GRU also now includes a "reset gate" and an "update gate" which you can read about further [here](https://pytorch.org/docs/stable/generated/torch.nn.GRU.html).

## ✓ Task 2: Simple GRU Model

Build an identical model to task 1 but substituting GRU layers instead of RNN layers and instead call the model `simplegru`. The model and training should be otherwise identical to the training from above. Be sure to pass `simplegru`'s parameters to the optimizer and not `simplernn`'s parameters.

<https://pytorch.org/docs/stable/generated/torch.nn.GRU.html>

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

```
# class SimpleGRU(

class SimpleGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, seq_len):
```





LSTM's were among the state of the art for language processing tasks for quite some time. Interestingly though, they were **first proposed in 1995** by Hochreiter and Schmidhuber almost **20 years before GRU**. In fact, LSTM models **outperform GRU models** on many tasks and by many metrics. However, **not without costs**. LSTM cells require **more storage space** and are **slower** than GRU cells. This is due to the fact that they have an additional **"long-term memory"** value which they can use to preserve information even between iterations. Whereas GRU will only "remember" information passed by adjacent neurons, **LSTM will remember** that and also information passed by **previous batches or iterations of data**.

## ✓ Task 3: Long Short-Term Memory (LSTM) Model

Build an identical model to task 1 but substituting LSTM layers instead of RNN layers and instead call the model `simplelstm`. The model and training should be otherwise identical to the training from above. Be sure to pass `simplelstm`'s parameters to the optimizer and not `simplernn`'s parameters.

<https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html>

```
# class SimpleLSTM(

class SimpleLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, seq_len):
        super(SimpleLSTM, self).__init__()
        self.rnn1 = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.rnn2 = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size * seq_len, output_size)

    def forward(self, x):
        x, hn = self.rnn1(x)
        x, _ = self.rnn2(x, hn)
        x = x.reshape(-1, x.size(1) * x.size(2))
        x = self.fc(x)
        return x

# simplelstm = SimpleLSTM(
# criterion =
# metric =
# optimizer =
# history, simplelstm = train_and_validate(
simplelstm = SimpleLSTM(input_size=1, hidden_size=16, output_size=1, seq_len=50)
criterion = nn.MSELoss()
metric = nn.L1Loss()
optimizer = torch.optim.Adam(simplelstm.parameters(), lr=0.001)

history, simplelstm = train_and_validate(
    train_loader,
    valid_loader,
    model=simplelstm,
    optimizer=optimizer,
    criterion=criterion,
    num_epochs=6,
    metric=metric,
    device=device
)

🔄 Epoch [1/6], Train Loss: 0.0065, Train Metric: 0.0571, Val Loss: 0.0031, Val Metric: 0.0453
Epoch [2/6], Train Loss: 0.0033, Train Metric: 0.0462, Val Loss: 0.0031, Val Metric: 0.0450
Epoch [3/6], Train Loss: 0.0032, Train Metric: 0.0452, Val Loss: 0.0030, Val Metric: 0.0443
Epoch [4/6], Train Loss: 0.0030, Train Metric: 0.0440, Val Loss: 0.0028, Val Metric: 0.0427
Epoch [5/6], Train Loss: 0.0027, Train Metric: 0.0420, Val Loss: 0.0027, Val Metric: 0.0415
Epoch [6/6], Train Loss: 0.0024, Train Metric: 0.0399, Val Loss: 0.0023, Val Metric: 0.0388
```

↑ your code goes above this

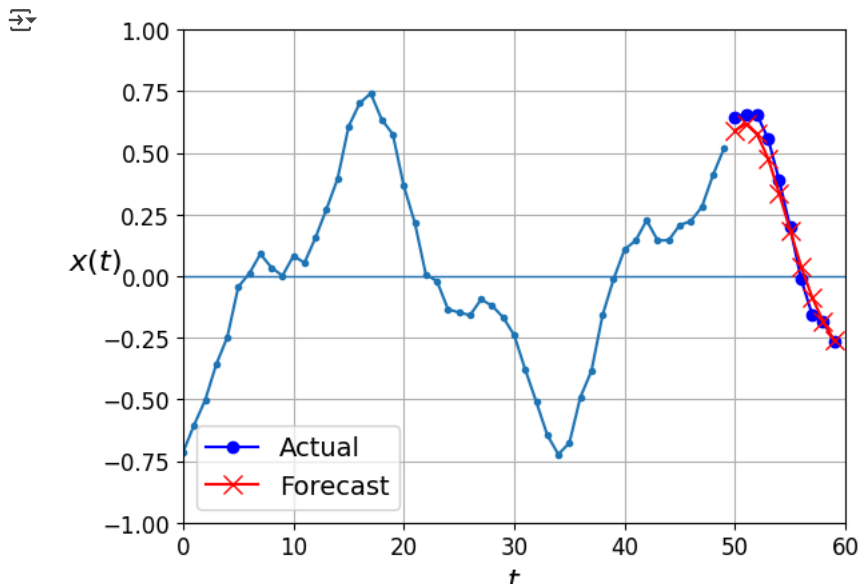
```
test_model(simplelstm, test_loader, criterion, metric=metric, device=device)
```

Test Loss: 0.0023, Test Metric: 0.0385  
(0.002273392732921585, tensor(0.0385, device='cuda:0'))

```
X = X_new
for step_ahead in range(10):
    y_pred_one = simplelstm(torch.from_numpy(X[:, step_ahead:]).to(device)).to('cpu').detach().numpy()[0, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

y_pred_new = X[:, n_steps:]

plot_multiple_forecasts(X_new, Y_new, y_pred_new)
plt.show()
```



### ✓ Task 3: Analyzing Model Performance

Question: Compare the behavior of LSTM to GRU and simple RNN. Which region of future forecasting is most different? Describe characteristics about the various models that would lead to that difference.

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Task 3) answer: The later forecasting steps, especially after step 5, is most different. LSTM performs better than simple RNN and GRU in capturing long-term dependencies. While all models predict well for the first few steps, the simple RNN quickly diverges. GRU holds better over time but still drifts after several steps. LSTM maintains more accurate forecasting in later steps. This is because LSTM has both short-term and long-term memory, allowing it to retain past information more effectively. In contrast, GRU and RNN rely only on recent hidden states, which limits their long-term memory.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this

### ✓ The Luong Attention Mechanism

Attention is an operation built around applying a **dot product** across **1) outputs from the head of a model** which has "encoded" the data by transforming with weights and biases and **2) hidden states (weights) from the tail of a model** which is attempting to "decode" the data and turn it into your desired output.

That is a long sentence but what it breaks down to is that you're **reweighting data** part way through a model so that the back half of your model knows what's most important from what the first half did. When Luong, Pham and Manning published [this paper](#) in 2015, they achieved **state of the art results** and laid the

foundation for the **Third Wave of AI** driven largely by attention-based  
Transformer models.

```
class AttentionLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, seq_len):
        super(AttentionLSTM, self).__init__()
        self.seq_len = seq_len

        # Encoder LSTM: processes the input sequence.
        self.encoder = nn.LSTM(input_size, hidden_size, batch_first=True)

        # Decoder LSTM: will receive the original input concatenated with the context.
        # Since we concatenate the context (of size hidden_size) with the original input,
        # the decoder input dimension becomes input_size + hidden_size.
        self.decoder = nn.LSTM(input_size + hidden_size, hidden_size, batch_first=True)

        # Final fully connected layer to map flattened decoder outputs to the desired output size.
        self.fc = nn.Linear(hidden_size * seq_len, output_size)

        # Softmax applied over the sequence dimension.
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        # x shape: (batch, seq_len, input_size)
        # -----
        # Encoder: process the input sequence.
        encoder_outputs, (h_enc, c_enc) = self.encoder(x)
        # encoder_outputs shape: (batch, seq_len, hidden_size)
        # h_enc shape: (num_layers, batch, hidden_size)
        # -----

        # Attention: Compute Luong-style dot-product attention.
        # Use the final encoder hidden state as the query.
        query = h_enc[-1] # shape: (batch, hidden_size)

        # To compute dot products between the query and each time step in encoder_outputs:
        # Expand query to shape (batch, hidden_size, 1)
        query = query.unsqueeze(2)
        # Compute scores: dot product between each encoder output and the query.
        # encoder_outputs: (batch, seq_len, hidden_size)
        # Resulting scores shape: (batch, seq_len, 1) -> squeeze to (batch, seq_len)
        scores = torch.bmm(encoder_outputs, query).squeeze(2)

        # Normalize the scores with softmax to get attention weights.
        attn_weights = self.softmax(scores) # shape: (batch, seq_len)

        # Compute the context vector as a weighted sum of encoder outputs.
        # attn_weights unsqueezed to shape (batch, 1, seq_len) for bmm.
        context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs).squeeze(1)
        # context shape: (batch, hidden_size)

        # Decoder: Incorporate the context into the decoder input.
        # Here we repeat the context vector along the time dimension and concatenate it with x.
        context_repeated = context.unsqueeze(1).repeat(1, self.seq_len, 1) # shape: (batch, seq_len, hidden_size)
        decoder_input = torch.cat((x, context_repeated), dim=2) # shape: (batch, seq_len, input_size + hidden_size)

        decoder_outputs, _ = self.decoder(decoder_input)

        # decoder_outputs shape: (batch, seq_len, hidden_size)
        decoder_outputs = decoder_outputs.reshape(decoder_outputs.size(0), -1)
        out = self.fc(decoder_outputs)
        return out

attlstm = AttentionLSTM(1, 16, 1, 50)
criterion = nn.MSELoss()
metric = nn.L1Loss()
optimizer = torch.optim.Adam(attlstm.parameters(), lr=0.001)
history, attlstm = train_and_validate(train_loader, valid_loader, attlstm, optimizer, criterion, num_epochs=6, metric=metric, scheduler=None, c
```

🔄 Epoch [1/6], Train Loss: 0.0072, Train Metric: 0.0590, Val Loss: 0.0032, Val Metric: 0.0461  
Epoch [2/6], Train Loss: 0.0032, Train Metric: 0.0458, Val Loss: 0.0032, Val Metric: 0.0459  
Epoch [3/6], Train Loss: 0.0029, Train Metric: 0.0434, Val Loss: 0.0026, Val Metric: 0.0411  
Epoch [4/6], Train Loss: 0.0025, Train Metric: 0.0407, Val Loss: 0.0024, Val Metric: 0.0397  
Epoch [5/6], Train Loss: 0.0024, Train Metric: 0.0399, Val Loss: 0.0023, Val Metric: 0.0385  
Epoch [6/6], Train Loss: 0.0024, Train Metric: 0.0394, Val Loss: 0.0022, Val Metric: 0.0384

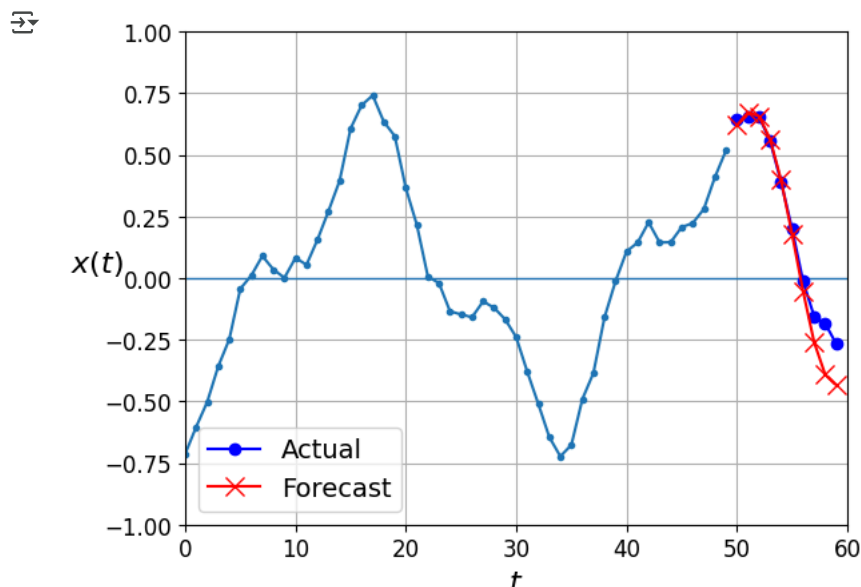
```
test_model(attlstm, test_loader, criterion, metric=metric, device=device)
```

```
Test Loss: 0.0022, Test Metric: 0.0382
(0.0022278609848382777, tensor(0.0382, device='cuda:0'))
```

```
X = X_new
for step_ahead in range(10):
    y_pred_one = attlstm(torch.from_numpy(X[:, step_ahead:]).to(device)).to('cpu').detach().numpy()[0, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)
```

```
y_pred_new = X[:, n_steps:]
```

```
plot_multiple_forecasts(X_new, Y_new, y_pred_new)
plt.show()
```



## ✓ The Transformer Model

The transformer architecture was first proposed in a paper called [Attention is All You Need \(Vaswani et. al 2017\)](#).

The characteristic operation of the transformer is the **self-attention** operation. In essence, it is using **dot product** operations along different axes of the data to amplify the importance of relevant data. You can think of this as effectively building **a model that reweights its inputs** to pick out the most important and relevant inputs. Then it does typical mathematical operations of **densely connected layers** on those reweighted inputs.

Researchers figured out several years before this that dot-product attention could be used in combination with other forms of models to improve performance by reweighting input data but the **Attention is All You Need** paper showed that, as the name would imply, you can get even **better performance** by eliminating all other characteristic layers and instead **using only attention and dense layers**.

```
print(X_train.shape)
```

```
(80000, 50, 1)
```

## ✓ Task 5: Defining model dimensions

Define two variables:

1) `max_len` should be equal to the size of the dimension from `X_train` that

describes the length of the time series. **Hint: see above plots**

2) embedding\_dim should be equal to the fourth root ( $^{1/4}$ ) of the total number of unique datapoints (e.g. number of samples \* time series length).

↓↓ your code goes below

```
max_len = X_train.shape[1]
# # Note: embedding_dim must be an even number
embedding_dim = int((X_train.shape[0] * X_train.shape[1]) ** 0.25)
if embedding_dim % 2 != 0:
    embedding_dim += 1
print(max_len)
print(embedding_dim)
```

↔ 50  
44

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above this

It's typical with transformers to embed our inputs into a higher dimensional space. For discrete data such as integers or words, it's typical to use an embedding layer from the pytorch toolkit. However, for continuous data, it's more common to embed using linear or non-linear transformations.

```
class LinearEmbedding(nn.Module):
    def __init__(self, in_features, out_features, **kwargs):
        super().__init__(**kwargs)
        self.linear = nn.Linear(in_features=in_features, out_features=out_features)
    def forward(self, x):
        return self.linear(x)
```

Another interesting feature of transformers is that they are not explicitly aware of the order of data in a relative sequence, especially after embedding. We therefore can inform it of positions by, for example, adding a small amount related to its position in the tensor by some interpretable form like periodic functions.

```
class SinusoidalPositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_len=512):
        super(SinusoidalPositionalEncoding, self).__init__()
        self.d_model = d_model
        self.max_seq_len = max_seq_len

        pe = torch.zeros(max_seq_len, d_model)
        position = torch.arange(0, max_seq_len, dtype=torch.float).unsqueeze(1)
        #Note: The value of 100.0 below corresponds to an encoding wavelength.
        #For LLMs this is often set much larger to, say 10k or ~10x the embed dim
        #The value of 100.0 was designed for very low-dimensional embedding data.
        div_term = torch.exp(torch.arange(0, d_model, 2, dtype=torch.float32) * (-torch.log(torch.tensor(100.0)) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        batch_size, seq_len, d_model = x.size()
        pe = self.pe[:, :seq_len, :]
        pe = pe.expand(batch_size, -1, -1)
        x = x + pe / torch.norm(pe, p=2, dim=-1, keepdim=True)
        return x
```

Transformer models use **multi-head attention (MHA)** which is very similar to the attention we saw before with LSTM. Here, we treat our data as having the three dimensions (**batch\_size, sequence\_length, features**). We use the linear embedding layer we saw before to **add additional information** to the features dimension in our data. In transformer notation, typically data is

expressed as having the dimensions (**q,k,v**) or (**query, key value**). That means each item in the **batch is a query** that try get get information for. We use the dot product operation for that query item with the other **keys and values** or **sequences and features** in the data and then add that information back to the query and then normalize to avoid exploding values.

### Short summary:

What this means is that at their core, transformers use dot product attention to **extract additional information** about how items in the sequence are **related** to one another, **add that information** to the original data, and then pass the modified data to a dense neural network.

```
class NextTokenPredictionTransformer(nn.Module):
    def __init__(self, embed_dim, latent_dim, num_heads, num_encoders, num_decoders, dropout, sequence_length, device):
        super(NextTokenPredictionTransformer, self).__init__()
        self.seq_len = sequence_length
        self.trans = nn.Transformer(d_model=embed_dim, nhead=num_heads,
                                     num_encoder_layers=num_encoders, num_decoder_layers=num_decoders,
                                     dim_feedforward=latent_dim, dropout=dropout, batch_first=True)

        self.sinemb = SinusoidalPositionalEncoding(embed_dim, sequence_length)
        self.linemb = LinearEmbedding(1, embed_dim)
        self.lin1 = nn.Linear(embed_dim, latent_dim)
        self.relu = nn.ReLU()
        self.lin2 = nn.Linear(latent_dim, 1)

    def forward(self, src, tgt):
        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = self.create_mask(src, tgt)
        src_padding_mask = src_padding_mask.squeeze(-1)
        tgt_padding_mask = tgt_padding_mask.squeeze(-1)
        src = self.linemb(src)
        tgt = self.linemb(tgt)
        src = self.sinemb(src)
        tgt = self.sinemb(tgt)
        out = self.trans(src, tgt, src_mask=src_mask, tgt_mask=tgt_mask,
                         src_key_padding_mask=src_padding_mask,
                         tgt_key_padding_mask=tgt_padding_mask)

        output = self.lin1(out)
        output = self.relu(output)
        output = self.lin2(output)
        return output

    def generate_square_subsequent_mask(self, sz, device):
        mask = (torch.triu(torch.ones((sz, sz), device=device)) == 1).transpose(0, 1)
        mask = ~mask
        return mask

    def create_mask(self, src, tgt):
        src_seq_len = src.shape[1]
        tgt_seq_len = tgt.shape[1]

        tgt_mask = self.generate_square_subsequent_mask(tgt_seq_len, device)
        src_mask = torch.zeros((src_seq_len, src_seq_len), device=device).type(torch.bool)

        src_padding_mask = (src == 0)
        tgt_padding_mask = (tgt == 0)
        return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask
```

**Note: Transformers are often much larger in dimensionality than RNNs so we may need to use different training procedure like schedules for best results.**

```
def transformer_train_and_validate(train_loader, val_loader, model, optimizer, criterion, num_epochs, metric=None, scheduler=None, device='cpu'):
    history = {
        'epoch': [],
        'train_loss': [],
        'train_metric': [],
        'val_loss': [],
        'val_metric': [],
        'learning_rate': []
    } # Initialize a dictionary to store epoch-wise results
```

```

model.to(device)    # Move the model to the specified device

for epoch in range(num_epochs):
    model.train()    # Set the model to training mode
    epoch_loss = 0.0    # Initialize the epoch loss and metric values
    epoch_metric = 0.0

    # Training loop
    for X, y in train_loader:
        X = X.to(device)
        y = y.to(device).unsqueeze(-1)
        optimizer.zero_grad()    # Clear existing gradients
        y_input = torch.cat((X[:, 1:], y[:, -1:]), dim=1)
        outputs = model(X[:,1:], y_input[:, :-1])
        # Reshape outputs and targets to match the shape expected by the loss function
        outputs = outputs.view(-1)
        y_input = y_input[:,1:].contiguous().view(-1)
        loss = criterion(outputs, y_input)    # Compute the loss
        loss.backward()    # Compute gradients
        optimizer.step()

        epoch_loss += loss.item()

    # THESE LINES HAVE BEEN UPDATED TO ACCOUNT FOR DEFAULT ARGUMENTS
    if metric is not None:
        epoch_metric += metric(outputs, y_input)
    else:
        epoch_metric += 0.0

# Average training loss and metric
epoch_loss /= len(train_loader)
epoch_metric /= len(train_loader)

# Validation loop
model.eval()    # Set the model to evaluation mode
with torch.no_grad():    # Disable gradient calculation
    val_loss = 0.0
    val_metric = 0.0
    for X_val, y_val in val_loader:
        X_val = X_val.to(device)
        y_val = y_val.to(device).unsqueeze(-1)
        y_input = torch.cat((X_val[:, 1:], y_val[:, -1:]), dim=1)
        outputs_val = model(X_val[:, 1:], y_input[:, :-1])
        outputs_val = outputs_val.view(-1)
        y_input = y_input[:,1:].contiguous().view(-1)
        val_loss += criterion(outputs_val, y_input).item()
        if metric is not None:
            val_metric += metric(outputs_val, y_input).item()
        else:
            val_metric += 0.0

    val_loss /= len(val_loader)
    val_metric /= len(val_loader)

# Append epoch results to history
history['epoch'].append(epoch)
history['train_loss'].append(epoch_loss)
history['train_metric'].append(epoch_metric)
history['val_loss'].append(val_loss)
history['val_metric'].append(val_metric)
history['learning_rate'].append(optimizer.param_groups[0]['lr'])

print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {epoch_loss:.4f}, '
      f'Train Metric: {epoch_metric:.4f}, Val Loss: {val_loss:.4f}, '
      f'Val Metric: {val_metric:.4f}')

if scheduler is not None:
    scheduler.step()

return history, model

```

## ✎ Task 6: Build a train a NextTokenPredictionTransformer

The transformer parameters should be:

```
embed_dim = embedding_dim
latent_dim = 16
num_heads = 1
num_encoders = 1
num_decoders = 1
dropout = 0.0
sequence_length = max_len
```

Transformers are a bit less stable to train so we'll need to use a slightly more complicated training procedure.

Use Adam optimizer with `lr=0.001` and train for 6 epochs using MSELoss as your criterion and L1Loss as your metric.

```
torch.manual_seed(42)
transformer = NextTokenPredictionTransformer(
    embed_dim=embedding_dim,
    latent_dim=16,
    num_heads=1,
    num_encoders=1,
    num_decoders=1,
    dropout=0.0,
    sequence_length=max_len,
    device=device
)
criterion = nn.MSELoss()
metric = nn.L1Loss()
optimizer = torch.optim.Adam(transformer.parameters(), lr=0.001)
history, transformer = transformer_train_and_validate(
    train_loader=train_loader,
    val_loader=val_loader,
    model=transformer,
    optimizer=optimizer,
    criterion=criterion,
    num_epochs=6,
    metric=metric,
    scheduler=None,
    device=device
)
# transformer = NextTokenPredictionTransformer(
# criterion =
# metric =
# optimizer =
# history, transformer = transformer_train_and_validate(

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/transformer.py:385: UserWarning: enable_nested_tensor is True, but self.use_nested_tensor is
warnings.warn(
Epoch [1/6], Train Loss: 0.0025, Train Metric: 0.0323, Val Loss: 0.0002, Val Metric: 0.0081
Epoch [2/6], Train Loss: 0.0001, Train Metric: 0.0058, Val Loss: 0.0001, Val Metric: 0.0035
Epoch [3/6], Train Loss: 0.0001, Train Metric: 0.0048, Val Loss: 0.0001, Val Metric: 0.0072
Epoch [4/6], Train Loss: 0.0001, Train Metric: 0.0040, Val Loss: 0.0001, Val Metric: 0.0027
Epoch [5/6], Train Loss: 0.0001, Train Metric: 0.0037, Val Loss: 0.0001, Val Metric: 0.0026
Epoch [6/6], Train Loss: 0.0001, Train Metric: 0.0035, Val Loss: 0.0001, Val Metric: 0.0043

def generate_next_step(model, input_sequence):
    model.eval()
    with torch.no_grad():
        input_sequence = input_sequence.to(device)
        output = model(input_sequence[:, 1:], input_sequence[:, :-1])
        next_token_prediction = output[:, -1, :]
    return next_token_prediction


X = X_new
for step_ahead in range(10):
    y_pred_one = generate_next_step(transformer, torch.from_numpy(X[:, step_ahead:]).to(device)).to('cpu').detach().numpy()[0, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

y_pred_new = X[:, n_steps:]
```





```
y_pred_trans = get_transformer_predictions(transformer, X_test)
mse_trans = mean_squared_error(y_test.reshape(-1), y_pred_trans)
mae_trans = mean_absolute_error(y_test.reshape(-1), y_pred_trans)
```

 LSTM - MSE: 0.002273392863571644 MAE: 0.03849470615386963  
 GRU - MSE: 0.0029578946996480227 MAE: 0.04383033886551857  
 Transformer - MSE: 0.006900328677147627 MAE: 0.06721284240484238

Task 7) answer: We compared how well three models—LSTM, GRU, and Transformer—performed on the test set. To evaluate them, we used two common metrics: Mean Squared Error (MSE), which gives more weight to large errors, and Mean Absolute Error (MAE), which reflects the average size of the errors regardless of direction. Among the three, the LSTM model had the lowest error rates on both metrics. This shows it's good at capturing both short-term patterns and longer-term dependencies in the data. The GRU model came in close, performing slightly worse than LSTM, which makes sense given that GRUs are simpler and don't have a separate long-term memory. The Transformer had the highest errors. This could be because Transformers are more sensitive to small datasets and usually need more training or larger data to perform well. Still, they tend to shine in bigger or more complex tasks. To sum it up, LSTM performed best, followed by GRU, and then Transformer, at least for this specific forecasting task on a relatively small dataset.

[illegible]

- ✓ ChatGPT

GPT stands for "**generative, pre-trained transformer**". ChatGPT is a version of a gpt **created by OpenAI** which made waves in late 2022 because of how well it mimics human conversation and interaction. Below, we'll look at how to access **ChatGPT in colab using a public API**. It is not possible to house the full model in a typical colab notebook because the model has an astounding **175 billion model parameters**. For comparison, two fully connected dense layers with 10,000 neurons each would only have around 100 million parameters.

In March 2023, OpenAI also released GPT4 which has over **1 TRILLION!** parameters. The model extended the capabilities of previous GPT iterations to include vision. It's so far shown incredible ability, even able to score in the **90th percentile on the Bar exam**, the final examination to become a lawyer in the United States.

```
!pip install openai
```

```

Requirement already satisfied: openai in /usr/local/lib/python3.11/dist-packages (1.70.0)
Requirement already satisfied: anyio<5,>=3.5.0 in /usr/local/lib/python3.11/dist-packages (from openai) (4.9.0)
Requirement already satisfied: distro<2,>=1.7.0 in /usr/local/lib/python3.11/dist-packages (from openai) (1.9.0)
Requirement already satisfied: httpx<1,>=0.23.0 in /usr/local/lib/python3.11/dist-packages (from openai) (0.28.1)
Requirement already satisfied: jiter<1,>=0.4.0 in /usr/local/lib/python3.11/dist-packages (from openai) (0.9.0)
Requirement already satisfied: pydantic<3,>=1.9.0 in /usr/local/lib/python3.11/dist-packages (from openai) (2.11.1)
Requirement already satisfied: sniffio in /usr/local/lib/python3.11/dist-packages (from openai) (1.3.1)
Requirement already satisfied: tqdm>4 in /usr/local/lib/python3.11/dist-packages (from openai) (4.67.1)
Requirement already satisfied: typing-extensions<5,>=4.11 in /usr/local/lib/python3.11/dist-packages (from openai) (4.13.0)
Requirement already satisfied: idna>=2.8 in /usr/local/lib/python3.11/dist-packages (from anyio<5,>=3.5.0->openai) (3.10)
Requirement already satisfied: certifi in /usr/local/lib/python3.11/dist-packages (from httpx<1,>=0.23.0->openai) (2025.1.31)
Requirement already satisfied: httptools==1.* in /usr/local/lib/python3.11/dist-packages (from httpx<1,>=0.23.0->openai) (1.0.7)
Requirement already satisfied: h11<0.15,>=0.13 in /usr/local/lib/python3.11/dist-packages (from httpx==1.*->httpx<1,>=0.23.0->openai) (0.14.0)
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.11/dist-packages (from pydantic<3,>=1.9.0->openai) (0.7.0)
Requirement already satisfied: pydantic-core==2.33.0 in /usr/local/lib/python3.11/dist-packages (from pydantic<3,>=1.9.0->openai) (2.33.0)
Requirement already satisfied: typing-inspection>=0.4.0 in /usr/local/lib/python3.11/dist-packages (from pydantic<3,>=1.9.0->openai) (0.4.0)

```

```
import openai
```

```
# Create an account at https://platform.openai.com
# Replace YOUR_API_KEY with your actual API key for the ChatGPT service:
# https://platform.openai.com/account/api-keys
my_api_key = "sk-proj-DvsYd12TZ1A3JeaNeX_MuihhDHUF-XSZwY2YwWmTotB879kGMQ_y64exTXU-N63EN8ZvEnc3QT3B1bkFJhSUU2pR0EFUhwvqjicwuE7FHeArcBef94-DRzm0i6tXUOP8Y"

client = openai.OpenAI(api_key=my_api_key)

response = client.chat.completions.create(
    messages=[
        {"role": "user", "content": "What is dot-product attention?"}
    ]
)
```

```

    ],
    model="gpt-3.5-turbo",
    max_tokens=300,
    n=1,
    stop=None,
    temperature=0.0,
)

message = response.choices[0].message.content.strip()
print(message)

```

ferent parts of a sequence of data. It is commonly used in transformer models, such as the popular BERT and GPT-3 models.

unction to normalize the scores. This allows the model to focus on different parts of the input sequence based on their relevance to the task at hand.

ar choice for many state-of-the-art models in natural language processing.

## Task 8 (Bonus): ChatGPT for Education

Using ChatGPT, generate an interesting piece of educational content about the current topic of RNNs and/or Transformers. **Hint: better prompt=better response**

**IF YOU EXCEED YOUR FREE OPENAI QUOTA JUST RUN THE CODE TO GENERATE A RESPONSE AND LEAVE THE ERROR MESSAGE IN THE OUTPUT CELL**

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

```

response = client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Explain in simple terms how Transformers are different from RNNs in processing sequences.",
        }
    ],
    model="gpt-3.5-turbo",
    max_tokens=300,
    temperature=0.5,
)

message = response.choices[0].message.content.strip()
print(message)

```

Transformers and RNNs are both types of neural networks that are used to process sequences of data, such as sentences or time series data. However, t

RNNs process sequences one element at a time, moving sequentially from one element to the next. This means that RNNs have a fixed order in which they

On the other hand, Transformers process all elements of a sequence simultaneously. This means that Transformers can capture dependencies between elem

In summary, Transformers are different from RNNs in how they process sequences because they can capture long-range dependencies in the data more effe

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above this

## Task 9: Fine-tuning LLMs using Hugging Face

This example is from:

[https://github.com/huggingface/notebooks/blob/main/examples/language\\_modeling.ipynb](https://github.com/huggingface/notebooks/blob/main/examples/language_modeling.ipynb)

which is an official Hugging Face demo by GitHub user Sylvain Gugger (sgugger)

Hugging Face is a private company that develops AI tools. Most notably the **transformers** and **datasets** packages. When new LLM models are released **open-source** they are often published through the transformers library.

### Creating a huggingface account and API key

- (1) Go to [huggingface.co](https://huggingface.co) and create an account. You can also link this to your github account for easier deployment of models.
- (2) Navigate to <https://huggingface.co/settings/tokens> and click: Create new token and create a FINEGRAINED token
- (3) You should give the token the following permissions

## Repositories




- ☒ Read access to contents of all repos under your personal namespace
- ☐ Read access to contents of all public gated repos you can access
- ☒ Write access to contents/settings of all repos under your personal namespace

- (4) If you forgot to give the above permissions when creating the token then click on 'Edit permissions' to add them.

## Permissions

FINEGRAINED



-  Edit permissions
-  Invalidate and refresh
-  Delete

- (5) Login using your API key

```
from huggingface_hub import notebook_login
```

```
notebook_login()
```



```
import transformers
```

```
print(transformers.__version__)
```



```
4.50.3
```

```
!pip install datasets
```

```

Requirement already satisfied: datasets in /usr/local/lib/python3.11/dist-packages (3.5.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from datasets) (3.18.0)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.11/dist-packages (from datasets) (2.0.2)
Requirement already satisfied: pyarrow>=15.0.0 in /usr/local/lib/python3.11/dist-packages (from datasets) (18.1.0)
Requirement already satisfied: dill<0.3.9,>=0.3.0 in /usr/local/lib/python3.11/dist-packages (from datasets) (0.3.8)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from datasets) (2.2.2)
Requirement already satisfied: requests>=2.32.2 in /usr/local/lib/python3.11/dist-packages (from datasets) (2.32.3)
Requirement already satisfied: tqdm>=4.66.3 in /usr/local/lib/python3.11/dist-packages (from datasets) (4.67.1)
Requirement already satisfied: xxhash in /usr/local/lib/python3.11/dist-packages (from datasets) (3.5.0)
Requirement already satisfied: multiprocessing<0.70.17 in /usr/local/lib/python3.11/dist-packages (from datasets) (0.70.16)
Requirement already satisfied: fsspec<=2024.12.0,>=2023.1.0 in /usr/local/lib/python3.11/dist-packages (from fsspec[http]<=2024.12.0,>=2023.1.0->data)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.11/dist-packages (from datasets) (3.11.15)
Requirement already satisfied: huggingface-hub>=0.24.0 in /usr/local/lib/python3.11/dist-packages (from datasets) (0.30.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from datasets) (24.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-packages (from datasets) (6.0.2)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp->datasets) (2.6.1)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.11/dist-packages (from aiohttp->datasets) (1.3.2)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp->datasets) (25.3.0)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.11/dist-packages (from aiohttp->datasets) (1.5.0)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.11/dist-packages (from aiohttp->datasets) (6.3.1)
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp->datasets) (0.3.1)
Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp->datasets) (1.18.3)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.11/dist-packages (from huggingface-hub>=0.24.0->datasets) (4.13.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests>=2.32.2->datasets) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests>=2.32.2->datasets) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests>=2.32.2->datasets) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests>=2.32.2->datasets) (2025.1.31)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas->datasets) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->datasets) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->datasets) (2025.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas->datasets) (1.17.0)

```

We're going to use the wikitext-2 dataset for model finetuning a LLM

```

from datasets import load_dataset
datasets = load_dataset('wikitext', 'wikitext-2-raw-v1')

```

```

/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret 'HF_TOKEN' does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(

```

README.md: 100%	10.5k/10.5k [00:00<00:00, 266kB/s]
test-00000-of-00001.parquet: 100%	733k/733k [00:00<00:00, 6.88MB/s]
train-00000-of-00001.parquet: 100%	6.36M/6.36M [00:00<00:00, 50.8MB/s]
validation-00000-of-00001.parquet: 100%	657k/657k [00:00<00:00, 18.9MB/s]
Generating test split: 100%	4358/4358 [00:00<00:00, 9146.73 examples/s]
Generating train split: 100%	36718/36718 [00:00<00:00, 192046.89 examples/s]
Generating validation split: 100%	3760/3760 [00:00<00:00, 77046.56 examples/s]

This is an example of what the data looks like. Notice it's just a **raw string**.

```
datasets["train"][10]
```

```

{'text': 'The game\'s battle system, the BliTz system, is carried over directly from Valkyria Chronicles. During missions, players select each unit using a top @-@ down perspective of the battlefield map: once a character is selected, the player moves the character around the battlefield in third @-@ person. A character can only act once per @-@ turn, but characters can be granted multiple turns at the expense of other characters\' turns. Each character has a field and distance of movement limited by their Action Gauge. Up to nine characters can be assigned to a single mission. During gameplay, characters will call out if something happens to them, such as their health points (HP) getting low or being knocked out by enemy attacks. Each character has specific "Potentials", skills unique to each character. They are divided into "Personal Potential", which are innate skills that remain unaltered unless otherwise dictated by the story and can either help or impede a character, and "Battle Potentials", which are grown throughout the game and always grant boons to a character. To learn Battle Potentials, each character has a unique "Masters Table", a grid @-@ based skill table that can be used to acquire and link different skills. Characters also have Special Abilities that grant them temporary boosts on the battlefield: Kurt can activate "Direct Command" and move around the battlefield without depleting his Action Point gauge, the character Reila can shift into her "Valkyria Form" and become invincible, while Imca can target multiple enemy units with her heavy weapon.
\n'}

```

```

from datasets import ClassLabel
import random

```

```
import pandas as pd
from IPython.display import display, HTML

def show_random_elements(dataset, num_examples=10):
    assert num_examples <= len(dataset), "Can't pick more elements than there are in the dataset."
    picks = []
    for _ in range(num_examples):
        pick = random.randint(0, len(dataset)-1)
        while pick in picks:
            pick = random.randint(0, len(dataset)-1)
        picks.append(pick)

    df = pd.DataFrame(dataset[picks])
    for column, typ in dataset.features.items():
        if isinstance(typ, ClassLabel):
            df[column] = df[column].transform(lambda i: typ.names[i])
    display(HTML(df.to_html()))

show_random_elements(datasets["train"])
```



text

0

1 Major restoration and foundation work began in the 1990s to stabilize the building . Engineers excavated under the cathedral between 1993 and 1998 . They dug shafts under the cathedral and placed shafts of concrete into the soft ground to give the edifice a more solid base to rest on . These efforts have not stopped the sinking of the complex , but they have corrected the tilting towers and ensured that the cathedral will sink uniformly . \n

2 American rapper Jay @-@ Z became involved late in the song 's production . Around three in the morning , he came to the studio and recorded a rap verse , which he improvised in about ten minutes . The recording of " Crazy in Love " took place nearly three months following the meeting of Beyoncé with Harrison . \n

3 The origins of the Georgian script are to this date poorly known , and no full agreement exists among Georgian and foreign scholars as to its date of creation , who designed the script and the main influences on that process . \n

4 The novel was written for Balliett classroom intended to deal with real @-@ world issues . Balliett values children 's ideas and wrote the book specifically to highlight that . Chasing Vermeer has won several awards , including the Edgar and the Agatha . In 2006 , the sequel entitled The Wright 3 was published , followed by The Calder Game in 2008 . . \n

5 Group Two , under the command of Captain Burn , would land at the old entrance to the St Nazaire basin . Their objectives were to destroy the anti @-@ aircraft positions in the area and the German headquarters , to blow up the locks and bridges at the old entrance into the basin and then to guard against a counter @-@ attack from the submarine base . Group Three was under the command of Major William ' Bill ' Copland , who was also the Commandos ' second in command .

6 They were to secure the immediate area around Campbeltown , destroy the dock 's water @-@ pumping and gate @-@ opening machinery and the nearby underground fuel tanks . All three groups were subdivided into assault , demolition and protection teams . The assault teams would clear the way for the other two . The demolition teams carrying the explosive charges only had sidearms for self @-@ defence ; the protection teams , armed with Thompson submachine guns , were to defend them while they completed their tasks . \n

6

7 The 1920s brought several favorable conditions that helped the land and population boom , one of which was an absence of any severe storms . The last severe hurricane , in 1906 , had struck the Florida Keys . Many homes were constructed hastily and poorly as a result of this lull in storms . However , on September 18 , 1926 , a storm that became known as the 1926 Miami Hurricane struck with winds over 140 miles per hour ( 230 km / h ) , and caused massive devastation . The storm surge was as high as 15 feet ( 4 @-@ 6 m ) in some places . Henry Flagler 's opulent Royal Palm Hotel was destroyed along with many other hotels and buildings . Most people who died did so when they ran out into the street in disbelief while the eye of the hurricane passed over , not knowing the wind was coming in from the other direction . " The lull lasted 35 minutes , and during that time the streets of the city became crowded with people " , wrote Richard Gray , the local weather chief . " As a result , many lives were lost during the second phase of the storm . " In Miami alone , 115 people were counted dead — although the true figure may have been as high as 175 , because death totals were racially segregated . More than 25 @-@ 000 people were homeless in the city . The town of Moore Haven , bordering Lake Okeechobee , was hardest hit . A levee built of muck collapsed , drowning almost 400 of the town 's entire 1 @-@ 200 residents .

8 The tops of Lake Okeechobee levees were only 18 to 24 inches ( 46 to 61 cm ) above the lake itself and the engineers were aware of the danger . Two days before the hurricane , an engineer predicted , " [ i ] f we have a blow , even a gale , Moore Haven is going under water " . The engineer lost his wife and daughter in the flood . \n

9 = Coldrum Long Barrow = \n

8

◀  ▶

We'll also need an LLM starting point. Here we'll use distilgpt2 which is a reduced size version of GPT-2, a much smaller predecessor to ChatGPT.

```
model_checkpoint = "distilgpt2"
```

We'll also need to **chunk our data strings** into smaller tokens so that it can be passed to the model like a **time-series**. To do that we'll use a prebuilt **tokenizer** from distilgpt2.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, use_fast=True)
```

```

tokenizer_config.json: 100% 26.0/26.0 [00:00<00:00, 1.18kB/s]
config.json: 100% 762/762 [00:00<00:00, 51.0kB/s]
vocab.json: 100% 1.04M/1.04M [00:00<00:00, 16.4MB/s]
merges.txt: 100% 456k/456k [00:00<00:00, 9.82MB/s]
tokenizer.json: 100% 1.26M/1.26M [00:00<00:00, 14.1MB/s]

```

```

def tokenize_function(examples):
    return tokenizer(examples["text"])

```

```

tokenized_datasets = datasets.map(tokenize_function, batched=True, num_proc=4, remove_columns=["text"])

```

```

Map (num_proc=4): 100% 4358/4358 [00:01<00:00, 3000.18 examples/s]
Map (num_proc=4): 100% 36718/36718 [00:08<00:00, 7679.84 examples/s]
Map (num_proc=4): 100% 3760/3760 [00:01<00:00, 5300.40 examples/s]

```

Notice that now the training dataset is abstracted way into **tokens** matching where in the **total vocabulary** a particular word is. There's also an **attention mask** value stored which can be used to tell the model to **ignore** a certain when computing losses and constructing attention maps.

```

datasets["train"][1]

```

```

{'text': ' = Valkyria Chronicles III = \n'}

```

```

tokenized_datasets["train"][1]

```

```

{'input_ids': [796, 569, 18354, 7496, 17740, 6711, 796, 220, 198],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1]}

```

```

block_size = tokenizer.model_max_length

```

```

def group_texts(examples):
    # Concatenate all texts.
    concatenated_examples = {k: sum(examples[k], []) for k in examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    # We drop the small remainder, we could add padding if the model supported it instead of this drop, you can
    # customize this part to your needs.
    total_length = (total_length // block_size) * block_size
    # Split by chunks of max_len.
    result = {
        k: [t[i : i + block_size] for i in range(0, total_length, block_size)]
        for k, t in concatenated_examples.items()
    }
    result["labels"] = result["input_ids"].copy()
    return result

```

```

lm_datasets = tokenized_datasets.map(
    group_texts,
    batched=True,
    batch_size=1000,
    num_proc=4,
)

```

```

Map (num_proc=4): 100% 4358/4358 [00:02<00:00, 2504.38 examples/s]
Map (num_proc=4): 100% 36718/36718 [00:12<00:00, 3041.14 examples/s]
Map (num_proc=4): 100% 3760/3760 [00:01<00:00, 3814.16 examples/s]

```

```

tokenizer.decode(lm_datasets["train"][1]["input_ids"])

```

```
→ ' attributes shared by the entire squad , a feature differing from early games \' method of distributing to different unit types . \n =  
= Plot == \n The game takes place during the Second European War . Gallian Army Squad 422 , also known as " The Nameless " , are a pena  
l military unit composed of criminals , foreign deserters , and military offenders whose real names are erased from the records and the  
reon officially referred to by numbers . Ordered by the Gallian military to perform the most dangerous missions that the Regular Army a  
nd Militia will not do , they are nevertheless up to the task , exemplified by their motto , Altaha Abilia , meaning " Always Ready . "  
The three main characters are No.7 Kurt Irving , an army officer falsely accused of treason who wishes to redeem himself ; Ace No.1 Imc
```

```
from transformers import AutoModelForCausalLM  
model = AutoModelForCausalLM.from_pretrained(model_checkpoint)
```

```
→ Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better perfo  
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to r  
model.safetensors: 100% 353M/353M [00:03<00:00, 177MB/s]  
generation_config.json: 100% 124/124 [00:00<00:00, 7.74kB/s]
```

```
from transformers import Trainer, TrainingArguments
```

```
model_name = model_checkpoint.split("/")[-1]  
training_args = TrainingArguments(  
    f"{model_name}-finetuned-wikitext2",  
    evaluation_strategy = "epoch",  
    learning_rate=2e-5,  
    weight_decay=0.01,  
    push_to_hub=True,  
)
```

If you want to **store model training history** you can create an account on **Weights and Biases** or wandb at <https://wandb.ai/>. You can also link this to your github account just like with the huggingface account. To acces your API key go to <https://wandb.ai/authorize> and copy your access token down and put it into the prompt below.

```
trainer = Trainer(  
    model=model,  
    args=training_args,  
    train_dataset=lm_datasets["train"],  
    eval_dataset=lm_datasets["validation"],  
)
```

Perplexity is a **numerical metric** that evaluates how good a model is at **predicting the next token** in a sequence. We'll compare perplexity before and after fine tuning on the `wikitext-2` dataset.

```
import math  
eval_results = trainer.evaluate()  
print(f"Perplexity: {math.exp(eval_results['eval_loss']):.2f}")
```

```
→ [30/30 00:12]  
wandb: WARNING If you're specifying your api key in code, ensure this code is not shared publicly.  
wandb: WARNING Consider setting the WANDB_API_KEY environment variable, or running `wandb login` from the command line.  
wandb: No netrc file found, creating one.  
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc  
wandb: Currently logged in as: yangyu01-sdu (yangyu01-sdu-the-university-of-alabama) to https://api.wandb.ai. Use `wandb login --relogir  
Tracking run with wandb version 0.19.9  
Run data is saved locally in /content/wandb/run-20250406_011134-puyp6oi1  
Syncing run distilgpt2-finetuned-wikitext2 to Weights & Biases (docs)  
View project at https://wandb.ai/yangyu01-sdu-the-university-of-alabama/huggingface  
View run at https://wandb.ai/yangyu01-sdu-the-university-of-alabama/huggingface/runs/puyp6oi1  
Perplexity: 46.51
```

```
trainer.train()
```



[870/870 23:03, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Model Preparation Time
1	No log	3.395194	0.001500
2	3.551700	3.366650	0.001500
3	3.551700	3.360828	0.001500

[30/30 07:47]

```
TrainOutput(global_step=870, training_loss=3.505157681169181, metrics={'train_runtime': 1384.6631, 'train_samples_per_second': 5.013,
eval_results = trainer.evaluate()
print(f"Perplexity: {math.exp(eval_results['eval_loss']):.2f}")
```

[30/30 00:12]

#

The model's perplexity improved meaning it's now better at doing next token prediction on the `wikitext-2` dataset. You can also now view your training model fine tuning training history on wandb.



```
trainer.push_to_hub()
```

events.out.tfevents.1743903311.33510d931a67.283.2: 100% 425/425 [00:00<00:00, 2.55kB/s]

Upload 2 LFS files: 100% 2/2 [00:00<00:00, 2.46it/s]

events.out.tfevents.1743901891.33510d931a67.283.1: 100% 7.30k/7.30k [00:00<00:00, 51.4kB/s]

CommitInfo(commit\_url='https://huggingface.co/yyu57/distilgpt2-finetuned-wikitext2/commit/135e10dc5ec72b218d8af2dfd5f4e328f6c6344e', commit\_message='End of training', commit\_description='', oid='135e10dc5ec72b218d8af2dfd5f4e328f6c6344e', pr\_url=None, repo\_url=RepoUrl('https://huggingface.co/yyu57/distilgpt2-finetuned-wikitext2', endpoint='https://huggingface.co', repo\_type='model', repo\_id='yyu57/distilgpt2-finetuned-wikitext2'), pr\_revision=None, pr\_num=None)

#