Grade: 90/100

Generally, you have everything put together correctly, but didn't adjust embedding lengths accordingly...

Open Lab 4

Recurrent Encoder-Decoder

CSCI 7850 - Deep Learning

Due: Oct. 26 @ 11:00pm

Assignment

Here are the details of what you need to do for this assignment:

1. Create one python script (`translation-simple.py`) that solves the ENG-POR problem using an Encoder-Decoder architecture. You will need to construct your network with the following properties:
   - Your code should utilize the top 10,000 sentences, shuffled (for monte-carlo sampling).
   - Use length 100 random embeddings for your encodings (`torch.nn.Embedding()`)
   - You should utilize simple recurrent layers in your model (`torch.nn.RNN()`)
   - Your recurrent layers should use the hyperbolic tangent activation function
   - Utilize the 80/20 validation split rule to train your model for 200 epochs
   - Your script should print the validation accuracy at the end **without teacher forcing**
2. Create one python script (`translation-lstm.py`) that solves the ENG-POR problem using an Encoder-Decoder architecture:
   - You should keep all architecture settings the same, **except** use LSTM recurrent layers (`torch.nn.LSTM()`)
3. Use your scripts to run your models - **perform 10 independent runs for each model**.
   - Compile your validation accuracy data into a single, two-column text file (`translation-results.txt`) that can be read in using `np.loadtxt` .
4. Create one python script (`parity-simple.py`) that solves the parity problem using an Encoder-Decoder architecture. You will need to construct your network with the following properties:
   - Your code should generate 1000 random bit strings (and their corresponding parity strings for targets) varying in length from 10 to 30 bits for training/validation data
   - Use length 20 random embeddings for your encodings (`torch.nn.Embedding()`)
   - You should utilize simple recurrent layers in your model (`torch.nn.RNN()`)
   - Your recurrent layers should use the hyperbolic tangent activation function
   - Utilize the 80/20 validation split rule to train your model for 1000 epochs
   - Your script should print the validation accuracy at the end **without teacher forcing**
5. Create one python script (`parity-lstm.py`) that solves the parity problem using an Encoder-Decoder architecture:
   - You should keep all architecture settings the same, **except** use LSTM recurrent layers (`torch.nn.LSTM()`)
6. Use your scripts to run your models - **perform 10 independent runs for each model**.
   - Compile your validation data into a text file (`parity-results.txt`) that can be read in using `np.loadtxt` .
7. Create an iPython Notebook file named `OL4.ipynb` which reads in the compiled results files that you created above to produce a boxplot comparing the performance of the problems/architectures.

Submission

Create a zip archive which contains the following contents:

- translation-simple.py
- translation-lstm.py
- translation-results.txt
- parity-simple.py
- parity-lstm.py
- parity-results.txt
- OL4.ipynb

Upload your zip archive to the course assignment system by the deadline at the top of this document.

Recurrent Encoder-Decoder

First, I will illustrate creating simple RNNs for this task - this is similar to what was explored in class. We will use random embeddings...

In [1]:
```
import numpy as np
import torch
import lightning.pytorch as pl
import torchmetrics
import torchvision
from torchinfo import summary
from torchview import draw_graph
from IPython.display import display
import sympy as sp
sp.init_printing(use_latex=True)
import pandas as pd
import matplotlib.pyplot as plt
```

In [2]:
```
if torch.cuda.is_available():
    print(torch.cuda.get_device_name())
    print(torch.cuda.get_device_properties("cuda"))
    print("Number of devices:",torch.cuda.device_count())
    device = ("cuda")
else:
    print("Only CPU is available...")
    device = ("cpu")
```

```
NVIDIA GeForce RTX 2080 Ti
_CudaDeviceProperties(name='NVIDIA GeForce RTX 2080 Ti', major=7, minor=5, total_memory=11011MB, mul
ti_processor_count=68)
Number of devices: 1
```

ENG-POR data set

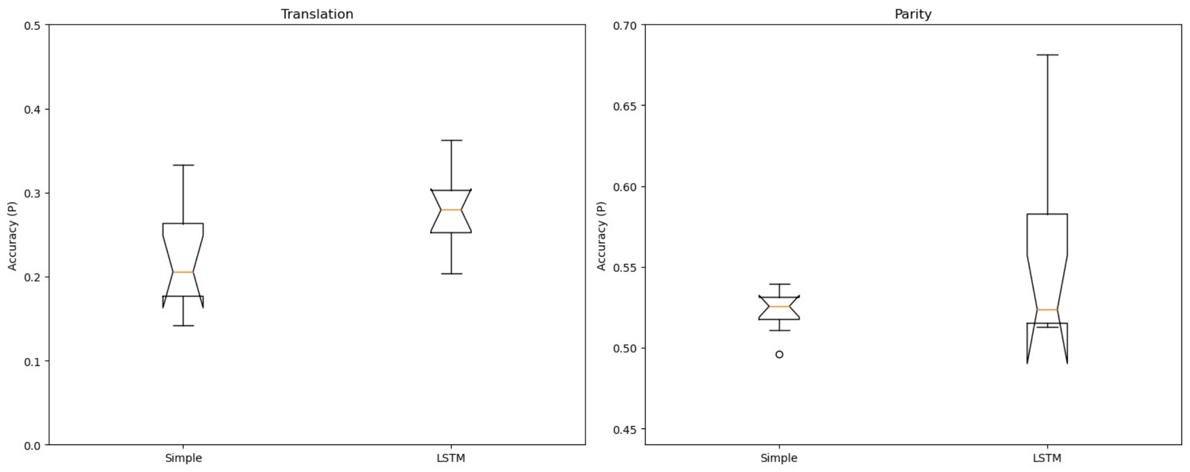
If cut off below...

https://raw.githubusercontent.com/luisroque/deep-learning-articles/main/data/eng-por.txt

In [3]:
```
url = "https://raw.githubusercontent.com/luisroque/deep-learning-articles/main/data/eng-por.txt"
```

In [8]:
```python
# Setup Imports
import numpy as np
import matplotlib.pyplot as plt

# Loading Results from OL4.py app outputs
results = [{
        "title":"Translation",
        "data": np.loadtxt("translation-results.txt"),
        "limits": [0.0, 0.5],
    },{
        "title":"Parity",
        "data": np.loadtxt("parity-results.txt"),
        "limits": [0.44, 0.7],
    }]
```

In [9]:
```python
# Create a 1x2 grid for the Plots (subplots)
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Default Values for all Plots
for i, axis in enumerate(axes.ravel()):

    # Setup Data Layers for Plots
    axis.set_title(results[i]["title"])

    axis.boxplot(results[i]["data"], notch=True)
    axis.set_ylabel('Accuracy (P)')
    axis.set_ylim(results[i]["limits"])
    axis.set_xticks([1, 2], ['Simple', 'LSTM'])


# Adjust spacing between subplots
plt.tight_layout()

# Show the plots
plt.show()
```



```
==> parity-lstm.py <==
#!/usr/bin/env python3

import time
start = time.time() # Start Timing

import numpy as np
import torch
import lightning.pytorch as pl
import torchmetrics
import torchvision
from torchinfo import summary

import pandas as pd


# Config Section
cfg_batch_size  = 20
cfg_max_epochs  = 200
cfg_num_workers = 2

# Array Building Config
cfg_array_count      = 1000
cfg_min_array_length = 10
cfg_max_array_length = 30
cfg_start_int = 1
cfg_stop_int = 2
cfg_empty_int = 0

# Logger Config
cfg_logger_dir     = "logs"
cfg_logger_name    = "OL4"
cfg_logger_version = "parity-lstm"

if torch.cuda.is_available():
    print(torch.cuda.get_device_name())
    print(torch.cuda.get_device_properties("cuda"))
    print("Number of devices:",torch.cuda.device_count())
    device = ("cuda")
else:
    print("Only CPU is available...")
    device = ("cpu")


# Create The data and parity array
# Code it with start and stop integers
def make_arrays():
    # Generate a random length between 10 and 30 (30 + 1)
    length = np.random.randint(cfg_min_array_length, cfg_max_array_length + 1)

    # Create a random array of 0s and 1s with the random length
    data_array = np.random.randint(2, size=length)

    # Initialize an array to store the cumulative sum of random_array
    cumulative_sum = np.cumsum(data_array)

    # Calculate parity for each cumulative sum and save to a separate array
    parity_array = cumulative_sum % 2

    # No We need to convert Array to Encoding 0s = 3 and 1s = 4
    enc_data_array = np.where(data_array == 0, 3, 4)
    enc_parity_array = np.where(parity_array == 0, 3, 4)
    enc_data_array = np.concatenate(([cfg_start_int], enc_data_array, [cfg_stop_int]))
    enc_parity_array = np.concatenate(([cfg_start_int], enc_parity_array, [cfg_stop_int]))

    # Add Padding at the end to make sure ALL Arrays are same length
    enc_data_array = np.pad(enc_data_array, (0, cfg_max_array_length - length), 'constant')
    enc_parity_array = np.pad(enc_parity_array, (0, cfg_max_array_length - length), 'constant'
)

    return np.array([np.array(enc_data_array), np.array(enc_parity_array)])


# Create an array to store 1000 arrays generated by make_arrays
data = np.array([make_arrays() for _ in range(cfg_array_count)])
print(f"data.shape: {data.shape}")

# Print Example of First Row
data_array = data[0][0]
parity_array = data[0][1]
print("\nArray Example:")
print("Encoded:\nKey: 1 = Start, 2 = Stop, 3 = Zero, 4 = One, 0 = Padding / Empty")
print("Data Array:   ", data_array)
print("Parity Array: ", parity_array)
print("")

# Set Split Point
split_point = int(data.shape[0] * 0.8) # Keep 80/20 split
print(f"split_point: {split_point}")


# Setup - Create X & Y
X = data[:,0]
Y = data[:,1]

enc_x_train = X[:split_point]
enc_x_val = X[split_point:]
enc_x_train

dec_x_train = Y[:,0:-1][:split_point]
dec_x_val = Y[:,0:-1][split_point:]
dec_x_train

dec_y_train = Y[:,1:][:split_point]
dec_y_val = Y[:,1:][split_point:]
dec_y_train

print(enc_x_train.shape)
print(dec_x_train.shape)
print(dec_y_train.shape)

print("----")

print(enc_x_val.shape)
print(dec_x_val.shape)
print(dec_y_val.shape)

# Start, Stop, One, Zero, Empty
token_count = 5
print(f"Token Count: {token_count}")


class Lstm(torch.nn.Module):
    def __init__(self,
                 latent_size = 64,
                 bidirectional = False,
                 **kwargs):
        super().__init__(**kwargs)
        self.layer_norm = torch.nn.LayerNorm(latent_size)
        self.lstm_layer = torch.nn.LSTM(latent_size,
                                        latent_size // 2 if bidirectional else latent_size,
                                        bidirectional=bidirectional,
                                        batch_first=True)
    def forward(self, x):
        return x + self.lstm_layer(self.layer_norm(x))[0]

class EncoderNetwork(torch.nn.Module):
    def __init__(self,
                 num_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(**kwargs)
        self.embedding = torch.nn.Embedding(num_tokens,
                                            latent_size,
                                            padding_idx=0)
        self.dropout = torch.nn.Dropout1d(0.1) # Whole token dropped
        self.lstm_layers = torch.nn.Sequential(*[
            Lstm(latent_size, True) for _ in range(n_layers)
        ])

    def forward(self, x):
        y = x
        y = self.embedding(y)
        y = self.dropout(y)
        y = self.lstm_layers(y)[:,-1]
        return y


enc_net = EncoderNetwork(num_tokens=token_count)
summary(enc_net,input_data=torch.Tensor(enc_x_train[0:5]).long())


# Decoder Component
class DecoderNetwork(torch.nn.Module):
    def __init__(self,
                 num_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(**kwargs)
        self.embedding = torch.nn.Embedding(num_tokens,
                                            latent_size,
                                            padding_idx=0)
        self.dropout = torch.nn.Dropout1d(0.1) # Whole token dropped
        self.linear = torch.nn.Linear(latent_size*2, latent_size)
        self.lstm_layers = torch.nn.Sequential(*[
            Lstm(latent_size,False) for _ in range(n_layers)
        ])
        self.output_layer = torch.nn.Linear(latent_size,
                                            num_tokens)

    def forward(self, x_enc, x_dec):
        y_enc = x_enc.unsqueeze(1).repeat(1,x_dec.shape[1],1)
        y_dec = self.embedding(x_dec)
        y_dec = self.dropout(y_dec)
        y = y_enc
        y = torch.concatenate([y_enc,y_dec],-1)
        y = self.linear(y)
        y = self.lstm_layers(y)
        y = self.output_layer(y)
        return y

dec_net = DecoderNetwork(num_tokens=token_count)
summary(dec_net,input_data=[enc_net(torch.Tensor(enc_x_train[0:5]).long()).cpu(), torch.Tensor
(dec_x_train[0:5]).long()])


class EncDecLightningModule(pl.LightningModule):
    def __init__(self,
                 output_size,
                 **kwargs):
        super().__init__(**kwargs)
        self.mc_acc = torchmetrics.classification.Accuracy(task='multiclass',
                                                           num_classes=output_size,
                                                           ignore_index=0)
        self.cce_loss = torch.nn.CrossEntropyLoss(ignore_index=0)

    def predict(self, x):
        return torch.softmax(self(x),-1)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        return optimizer

    def training_step(self, train_batch, batch_idx):
        x_enc, x_dec, y_dec = train_batch
        y_pred = self(x_enc, x_dec)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('train_acc', acc, on_step=False, on_epoch=True)
        self.log('train_loss', loss, on_step=False, on_epoch=True)
        return loss

    # Validate used for Teacher Forcing
    def validation_step(self, val_batch, batch_idx):
        x_enc, x_dec, y_dec = val_batch
        y_pred = self(x_enc, x_dec)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('val_acc', acc, on_step=False, on_epoch=True)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        return loss

    # Test used for Non-Teacher Forcing
    def test_step(self, test_batch, batch_idx):
        x_enc, x_dec, y_dec = test_batch
        context = self.enc_net(x_enc)
        tokens = torch.zeros_like(x_dec).long()
        tokens[:,0] = 1
        for i in range(y_dec.shape[1]-1):
            tokens[:,i+1] = self.dec_net(context, tokens).argmax(-1)[:,i]
        y_pred = self(x_enc, tokens)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('test_acc', acc, on_step=False, on_epoch=True)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        return loss
```

```
class EncDecNetwork(EncDecLightningModule):
    def __init__(self,
                 num_enc_tokens,
                 num_dec_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(output_size=num_dec_tokens, **kwargs)
        self.enc_net = EncoderNetwork(num_enc_tokens,latent_size,n_layers)
        self.dec_net = DecoderNetwork(num_dec_tokens,latent_size,n_layers)

    def forward(self, x_enc, x_dec):
        return self.dec_net(self.enc_net(x_enc), x_dec)


enc_dec_net = EncDecNetwork(num_enc_tokens=token_count,
                            num_dec_tokens=token_count)

summary(enc_dec_net,input_data=[torch.Tensor(enc_x_train[0:1]).long(),
                                torch.Tensor(dec_x_train[0:1]).long()])



xy_train = torch.utils.data.DataLoader(list(zip(torch.Tensor(enc_x_train).long(),
                                                torch.Tensor(dec_x_train).long(),
                                                torch.Tensor(dec_y_train).long())),
                                       shuffle=True,
                                       batch_size=cfg_batch_size,
                                       num_workers=cfg_num_workers)

xy_val = torch.utils.data.DataLoader(list(zip(torch.Tensor(enc_x_val).long(),
                                              torch.Tensor(dec_x_val).long(),
                                              torch.Tensor(dec_y_val).long())),
                                     shuffle=False,
                                     batch_size=cfg_batch_size,
                                     num_workers=cfg_num_workers)


logger = pl.loggers.CSVLogger(cfg_logger_dir,
                              name=cfg_logger_name,
                              version=cfg_logger_version)


trainer = pl.Trainer(logger=logger,
                     max_epochs=cfg_max_epochs,
                     enable_progress_bar=True,
                     log_every_n_steps=0,
                     enable_checkpointing=False,
                     callbacks=[pl.callbacks.TQDMProgressBar(refresh_rate=50)])


# Train Model
trainer.fit(enc_dec_net, xy_train, xy_val)

# Test Model
results = trainer.test(enc_dec_net, xy_val)

# Hanlding Timing
end = time.time()
elapsed = end - start

print("")
print(f"Processing Time: {elapsed:.6f} seconds\n")
print("Test Accuracy:", results[0]['test_acc'])
print("")
print("")
==> parity-simple.py <==
#!/usr/bin/env python3

import time
start = time.time() # Start Timing

import numpy as np
import torch
import lightning.pytorch as pl
import torchmetrics
import torchvision
from torchinfo import summary

import pandas as pd


# Config Section
cfg_batch_size  = 20
cfg_max_epochs  = 200
cfg_num_workers = 2

# Array Building Config
cfg_array_count      = 1000
cfg_min_array_length = 10
cfg_max_array_length = 30
cfg_start_int = 1
cfg_stop_int = 2
cfg_empty_int = 0

# Logger Config
cfg_logger_dir     = "logs"
cfg_logger_name    = "OL4"
cfg_logger_version = "parity-simple"

if torch.cuda.is_available():
    print(torch.cuda.get_device_name())
    print(torch.cuda.get_device_properties("cuda"))
    print("Number of devices:",torch.cuda.device_count())
    device = ("cuda")
else:
    print("Only CPU is available...")
    device = ("cpu")


# Create The data and parity array
# Code it with start and stop integers
def make_arrays():
    # Generate a random length between 10 and 30 (30 + 1)
    length = np.random.randint(cfg_min_array_length, cfg_max_array_length + 1)

    # Create a random array of 0s and 1s with the random length
    data_array = np.random.randint(2, size=length)

    # Initialize an array to store the cumulative sum of random_array
    cumulative_sum = np.cumsum(data_array)

    # Calculate parity for each cumulative sum and save to a separate array
    parity_array = cumulative_sum % 2
    # No We need to convert Array to Encoding 0s = 3 and 1s = 4
    enc_data_array = np.where(data_array == 0, 3, 4)
    enc_parity_array = np.where(parity_array == 0, 3, 4)

    enc_data_array = np.concatenate(([cfg_start_int], enc_data_array, [cfg_stop_int]))
    enc_parity_array = np.concatenate(([cfg_start_int], enc_parity_array, [cfg_stop_int]))

    # Add Padding at the end to make sure ALL Arrays are same length
    enc_data_array = np.pad(enc_data_array, (0, cfg_max_array_length - length), 'constant')
    enc_parity_array = np.pad(enc_parity_array, (0, cfg_max_array_length - length), 'constant'
)

    return np.array([np.array(enc_data_array), np.array(enc_parity_array)])


# Create an array to store 1000 arrays generated by make_arrays
data = np.array([make_arrays() for _ in range(cfg_array_count)])
print(f"data.shape: {data.shape}")

# Print Example of First Row
data_array = data[0][0]
parity_array = data[0][1]
print("\nArray Example:")
print("Encoded:\nKey: 1 = Start, 2 = Stop, 3 = Zero, 4 = One, 0 = Padding / Empty")
print("Data Array:   ", data_array)
print("Parity Array: ", parity_array)
print("")

# Set Split Point
split_point = int(data.shape[0] * 0.8) # Keep 80/20 split
print(f"split_point: {split_point}")


# Setup - Create X & Y
X = data[:,0]
Y = data[:,1]

enc_x_train = X[:split_point]
enc_x_val = X[split_point:]
enc_x_train

dec_x_train = Y[:,0:-1][:split_point]
dec_x_val = Y[:,0:-1][split_point:]
dec_x_train

dec_y_train = Y[:,1:][:split_point]
dec_y_val = Y[:,1:][split_point:]
dec_y_train

print(enc_x_train.shape)
print(dec_x_train.shape)
print(dec_y_train.shape)

print("----")

print(enc_x_val.shape)
print(dec_x_val.shape)
print(dec_y_val.shape)

# Start, Stop, One, Zero, Empty
token_count = 5
print(f"Token Count: {token_count}")
```

```
class RecurrentResidual(torch.nn.Module):
    def __init__(self,
                 latent_size = 64,
                 bidirectional = False,
                 **kwargs):
        super().__init__(**kwargs)
        self.layer_norm = torch.nn.LayerNorm(latent_size)
        self.rnn_layer = torch.nn.RNN(latent_size,
                                      latent_size // 2 if bidirectional else latent_size,
                                      bidirectional=bidirectional,
                                      nonlinearity='tanh',
                                      batch_first=True)
    def forward(self, x):
        return x + self.rnn_layer(self.layer_norm(x))[0]

class EncoderNetwork(torch.nn.Module):
    def __init__(self,
                 num_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(**kwargs)
        self.embedding = torch.nn.Embedding(num_tokens,
                                            latent_size,
                                            padding_idx=0)
        self.dropout = torch.nn.Dropout1d(0.1) # Whole token dropped
        self.rnn_layers = torch.nn.Sequential(*[
            RecurrentResidual(latent_size, True) for _ in range(n_layers)
        ])

    def forward(self, x):
        y = x
        y = self.embedding(y)
        y = self.dropout(y)
        y = self.rnn_layers(y)[:,-1]
        return y


enc_net = EncoderNetwork(num_tokens=token_count)
summary(enc_net,input_data=torch.Tensor(enc_x_train[0:5]).long())


# Decoder Component
class DecoderNetwork(torch.nn.Module):
    def __init__(self,
                 num_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(**kwargs)
        self.embedding = torch.nn.Embedding(num_tokens,
                                            latent_size,
                                            padding_idx=0)
        self.dropout = torch.nn.Dropout1d(0.1) # Whole token dropped
        self.linear = torch.nn.Linear(latent_size*2, latent_size)
        self.rnn_layers = torch.nn.Sequential(*[
            RecurrentResidual(latent_size,False) for _ in range(n_layers)
        ])
        self.output_layer = torch.nn.Linear(latent_size,
                                            num_tokens)

    def forward(self, x_enc, x_dec):
        y_enc = x_enc.unsqueeze(1).repeat(1,x_dec.shape[1],1)
        y_dec = self.embedding(x_dec)
        y_dec = self.dropout(y_dec)
        y = y_enc
        y = torch.concatenate([y_enc,y_dec],-1)
        y = self.linear(y)
        y = self.rnn_layers(y)
        y = self.output_layer(y)
        return y

dec_net = DecoderNetwork(num_tokens=token_count)
summary(dec_net,input_data=[enc_net(torch.Tensor(enc_x_train[0:5]).long()).cpu(), torch.Tensor
(dec_x_train[0:5]).long()])


class EncDecLightningModule(pl.LightningModule):
    def __init__(self,
                 output_size,
                 **kwargs):
        super().__init__(**kwargs)
        self.mc_acc = torchmetrics.classification.Accuracy(task='multiclass',
                                                           num_classes=output_size,
                                                           ignore_index=0)
        self.cce_loss = torch.nn.CrossEntropyLoss(ignore_index=0)

    def predict(self, x):
        return torch.softmax(self(x),-1)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        return optimizer

    def training_step(self, train_batch, batch_idx):
        x_enc, x_dec, y_dec = train_batch
        y_pred = self(x_enc, x_dec)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('train_acc', acc, on_step=False, on_epoch=True)
        self.log('train_loss', loss, on_step=False, on_epoch=True)
        return loss

    # Validate used for Teacher Forcing
    def validation_step(self, val_batch, batch_idx):
        x_enc, x_dec, y_dec = val_batch
        y_pred = self(x_enc, x_dec)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('val_acc', acc, on_step=False, on_epoch=True)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        return loss

    # Test used for Non-Teacher Forcing
    def test_step(self, test_batch, batch_idx):
        x_enc, x_dec, y_dec = test_batch
        context = self.enc_net(x_enc)
        tokens = torch.zeros_like(x_dec).long()
        tokens[:,0] = 1
        for i in range(y_dec.shape[1]-1):
            tokens[:,i+1] = self.dec_net(context, tokens).argmax(-1)[:,i]
        y_pred = self(x_enc, tokens)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('test_acc', acc, on_step=False, on_epoch=True)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        return loss

class EncDecNetwork(EncDecLightningModule):
    def __init__(self,
                 num_enc_tokens,
                 num_dec_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(output_size=num_dec_tokens, **kwargs)
        self.enc_net = EncoderNetwork(num_enc_tokens,latent_size,n_layers)
        self.dec_net = DecoderNetwork(num_dec_tokens,latent_size,n_layers)

    def forward(self, x_enc, x_dec):
        return self.dec_net(self.enc_net(x_enc), x_dec)


enc_dec_net = EncDecNetwork(num_enc_tokens=token_count,
                            num_dec_tokens=token_count)

summary(enc_dec_net,input_data=[torch.Tensor(enc_x_train[0:1]).long(),
                                torch.Tensor(dec_x_train[0:1]).long()])
```

Looks like you didn't adjust the embedding size in either of the above models - everything is 64... (-5)

```
xy_train = torch.utils.data.DataLoader(list(zip(torch.Tensor(enc_x_train).long(),
                                                torch.Tensor(dec_x_train).long(),
                                                torch.Tensor(dec_y_train).long())),
                                       shuffle=True,
                                       batch_size=cfg_batch_size,
                                       num_workers=cfg_num_workers)

xy_val = torch.utils.data.DataLoader(list(zip(torch.Tensor(enc_x_val).long(),
                                              torch.Tensor(dec_x_val).long(),
                                              torch.Tensor(dec_y_val).long())),
                                     shuffle=False,
                                     batch_size=cfg_batch_size,
                                     num_workers=cfg_num_workers)


logger = pl.loggers.CSVLogger(cfg_logger_dir,
                              name=cfg_logger_name,
                              version=cfg_logger_version)


trainer = pl.Trainer(logger=logger,
                     max_epochs=cfg_max_epochs,
                     enable_progress_bar=True,
                     log_every_n_steps=0,
                     enable_checkpointing=False,
                     callbacks=[pl.callbacks.TQDMProgressBar(refresh_rate=50)])


# Train Model
trainer.fit(enc_dec_net, xy_train, xy_val)

# Test Model
results = trainer.test(enc_dec_net, xy_val)
```

```
# Hanlding Timing
end = time.time()
elapsed = end - start

print("")
print(f"Processing Time: {elapsed:.6f} seconds\n")
print("Test Accuracy:", results[0]['test_acc'])
print("")
print("")
==> translation-lstm.py <==
#!/usr/bin/env python3

import time
start = time.time() # Start Timing

import numpy as np
import torch
import lightning.pytorch as pl
import torchmetrics
import torchvision
from torchinfo import summary

import pandas as pd

import urllib


if torch.cuda.is_available():
    print(torch.cuda.get_device_name())
    print(torch.cuda.get_device_properties("cuda"))
    print("Number of devices:", torch.cuda.device_count())
    device = ("cuda")
else:
    print("Only CPU is available...")
    device = ("cpu")

# Config Section
cfg_batch_size  = 20
cfg_max_epochs  = 200
cfg_num_workers = 2
cfg_url         = "https://raw.githubusercontent.com/luisroque/deep-learning-articles/main/dat
a/eng-por.txt"

# Logger Config
cfg_logger_dir     = "logs"
cfg_logger_name    = "OL4"
cfg_logger_version = "translation-lstm"

# Setup Data Objects
data = []
with urllib.request.urlopen(cfg_url) as raw_data:
    for line in raw_data:
        data.append(line.decode("utf-8").split('\t')[0:2])

data = np.array(data)

# Subset? - All of the data will take some time...
n_seq = data.shape[0]
n_seq = 10000
data = data[0:n_seq]
split_point = int(data.shape[0] * 0.8) # Keep 80/20 split
np.random.shuffle(data) # In-place modification
max_length = np.max([len(i) for i in data.flatten()]) + 2 # Add start/stop
```

```
print(f"max_length = ${max_length}")
print(f"data[0] => ${data[0]}")

# Setup
i_to_c_eng = ['','<START>','<STOP>'] + list({char for word in data[:,0] for char in word})
c_to_i_eng = {i_to_c_eng[i]:i for i in range(len(i_to_c_eng))}
i_to_c_eng[1] = i_to_c_eng[2] = ''

i_to_c_por = ['','<START>','<STOP>'] + list({char for word in data[:,1] for char in word})
c_to_i_por = {i_to_c_por[i]:i for i in range(len(i_to_c_por))}
i_to_c_por[1] = i_to_c_por[2] = ''


def encode_seq(x, mapping, max_length=0):
    # String to integer
    return [mapping['<START>']] + \
    [mapping[i] for i in list(x)] + \
    [mapping['<STOP>']] + \
    [0]*(max_length-len(list(x))-2)

def decode_seq(x, mapping):
    # Integer-to-string
    try:
        idx = list(x).index(2) # Stop token?
    except:
        idx = len(list(x)) # No stop token found
    return ''.join([mapping[i] for i in list(x)[0:idx]])



# Setup
X = np.vstack([encode_seq(x, c_to_i_eng, max_length) for x in data[:,0]])
Y = np.vstack([encode_seq(x, c_to_i_por, max_length) for x in data[:,1]])

enc_x_train = X[:split_point]
enc_x_val = X[split_point:]
enc_x_train

dec_x_train = Y[:,0:-1][:split_point]
dec_x_val = Y[:,0:-1][split_point:]
dec_x_train

dec_y_train = Y[:,1:][:split_point]
dec_y_val = Y[:,1:][split_point:]
dec_y_train

print(enc_x_train.shape)
print(dec_x_train.shape)
print(dec_y_train.shape)

print("----")

print(enc_x_val.shape)
print(dec_x_val.shape)
print(dec_y_val.shape)


class Lstm(torch.nn.Module):
    def __init__(self,
                 latent_size = 64,
                 bidirectional = False,
                 **kwargs):
```

```
        super().__init__(**kwargs)
        self.layer_norm = torch.nn.LayerNorm(latent_size)
        self.lstm_layer = torch.nn.LSTM(latent_size,
                                        latent_size // 2 if bidirectional else latent_size,
                                        bidirectional=bidirectional,
                                        batch_first=True)
    def forward(self, x):
        return x + self.lstm_layer(self.layer_norm(x))[0]

class EncoderNetwork(torch.nn.Module):
    def __init__(self,
                 num_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(**kwargs)
        self.embedding = torch.nn.Embedding(num_tokens,
                                            latent_size,
                                            padding_idx=0)
        self.dropout = torch.nn.Dropout1d(0.1) # Whole token dropped
        self.lstm_layers = torch.nn.Sequential(*[
            Lstm(latent_size, True) for _ in range(n_layers)
        ])

    def forward(self, x):
        y = x
        y = self.embedding(y)
        y = self.dropout(y)
        y = self.lstm_layers(y)[:,-1]
        return y

enc_net = EncoderNetwork(num_tokens=len(i_to_c_eng))
summary(enc_net,input_data=torch.Tensor(enc_x_train[0:5]).long())


# Decoder Component
class DecoderNetwork(torch.nn.Module):
    def __init__(self,
                 num_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(**kwargs)
        self.embedding = torch.nn.Embedding(num_tokens,
                                            latent_size,
                                            padding_idx=0)
        self.dropout = torch.nn.Dropout1d(0.1) # Whole token dropped
        self.linear = torch.nn.Linear(latent_size*2, latent_size)
        self.lstm_layers = torch.nn.Sequential(*[
            Lstm(latent_size,False) for _ in range(n_layers)
        ])
        self.output_layer = torch.nn.Linear(latent_size,
                                            num_tokens)

    def forward(self, x_enc, x_dec):
        y_enc = x_enc.unsqueeze(1).repeat(1,x_dec.shape[1],1)
        y_dec = self.embedding(x_dec)
        y_dec = self.dropout(y_dec)
        y = y_enc
        y = torch.concatenate([y_enc,y_dec],-1)
        y = self.linear(y)
        y = self.lstm_layers(y)
        y = self.output_layer(y)
        return y

dec_net = DecoderNetwork(num_tokens=len(i_to_c_por))
summary(dec_net,input_data=[enc_net(torch.Tensor(enc_x_train[0:5]).long()).cpu(), torch.Tensor
(dec_x_train[0:5]).long()])


class EncDecLightningModule(pl.LightningModule):
    def __init__(self,
                 output_size,
                 **kwargs):
        super().__init__(**kwargs)
        self.mc_acc = torchmetrics.classification.Accuracy(task='multiclass',
                                                           num_classes=output_size,
                                                           ignore_index=0)
        self.cce_loss = torch.nn.CrossEntropyLoss(ignore_index=0)

    def predict(self, x):
        return torch.softmax(self(x),-1)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        return optimizer

    def training_step(self, train_batch, batch_idx):
        x_enc, x_dec, y_dec = train_batch
        y_pred = self(x_enc, x_dec)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('train_acc', acc, on_step=False, on_epoch=True)
        self.log('train_loss', loss, on_step=False, on_epoch=True)
        return loss

    # Validate used for Teacher Forcing
    def validation_step(self, val_batch, batch_idx):
        x_enc, x_dec, y_dec = val_batch
        y_pred = self(x_enc, x_dec)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('val_acc', acc, on_step=False, on_epoch=True)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        return loss

    # Test used for Non-Teacher Forcing
    def test_step(self, test_batch, batch_idx):
        x_enc, x_dec, y_dec = test_batch
        context = self.enc_net(x_enc)
        tokens = torch.zeros_like(x_dec).long()
        tokens[:,0] = 1
        for i in range(y_dec.shape[1]-1):
            tokens[:,i+1] = self.dec_net(context, tokens).argmax(-1)[:,i]
        y_pred = self(x_enc, tokens)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('test_acc', acc, on_step=False, on_epoch=True)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        return loss

class EncDecNetwork(EncDecLightningModule):
    def __init__(self,
                 num_enc_tokens,
                 num_dec_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(output_size=num_dec_tokens, **kwargs)
        self.enc_net = EncoderNetwork(num_enc_tokens,latent_size,n_layers)
        self.dec_net = DecoderNetwork(num_dec_tokens,latent_size,n_layers)

    def forward(self, x_enc, x_dec):
        return self.dec_net(self.enc_net(x_enc), x_dec)


enc_dec_net = EncDecNetwork(num_enc_tokens=len(i_to_c_eng),
                            num_dec_tokens=len(i_to_c_por))

summary(enc_dec_net,input_data=[torch.Tensor(enc_x_train[0:1]).long(),
                                torch.Tensor(dec_x_train[0:1]).long()])


xy_train = torch.utils.data.DataLoader(list(zip(torch.Tensor(enc_x_train).long(),
                                                torch.Tensor(dec_x_train).long(),
                                                torch.Tensor(dec_y_train).long())),
                                       shuffle=True,
                                       batch_size=cfg_batch_size,
                                       num_workers=cfg_num_workers)

xy_val = torch.utils.data.DataLoader(list(zip(torch.Tensor(enc_x_val).long(),
                                              torch.Tensor(dec_x_val).long(),
                                              torch.Tensor(dec_y_val).long())),
                                     shuffle=False,
                                     batch_size=cfg_batch_size,
                                     num_workers=cfg_num_workers)


logger = pl.loggers.CSVLogger(cfg_logger_dir,
                              name=cfg_logger_name,
                              version=cfg_logger_version)

trainer = pl.Trainer(logger=logger,
                     max_epochs=cfg_max_epochs,
                     enable_progress_bar=True,
                     log_every_n_steps=0,
                     enable_checkpointing=False,
                     callbacks=[pl.callbacks.TQDMProgressBar(refresh_rate=50)])


# Train Model
trainer.fit(enc_dec_net, xy_train, xy_val)

# Test Model
results = trainer.test(enc_dec_net, xy_val)

# Hanlding Timing
end = time.time()
elapsed = end - start

print("")
print(f"Processing Time: {elapsed:.6f} seconds\n")
print("Test Accuracy:", results[0]['test_acc'])
print("")
print("")
==> translation-simple.py <==
#!/usr/bin/env python3

import time
start = time.time() # Start Timing

import numpy as np
import torch
import lightning.pytorch as pl
import torchmetrics
import torchvision
from torchinfo import summary

import pandas as pd

import urllib


if torch.cuda.is_available():
    print(torch.cuda.get_device_name())
    print(torch.cuda.get_device_properties("cuda"))
    print("Number of devices:", torch.cuda.device_count())
    device = ("cuda")
else:
    print("Only CPU is available...")
    device = ("cpu")

# Config Section
cfg_batch_size  = 20
cfg_max_epochs  = 200
cfg_num_workers = 2
cfg_url         = "https://raw.githubusercontent.com/luisroque/deep-learning-articles/main/dat
a/eng-por.txt"

# Logger Config
cfg_logger_dir     = "logs"
cfg_logger_name    = "OL4"
cfg_logger_version = "translation-simple"

# Setup Data Objects
data = []
with urllib.request.urlopen(cfg_url) as raw_data:
    for line in raw_data:
        data.append(line.decode("utf-8").split('\t')[0:2])

data = np.array(data)

# Subset? - All of the data will take some time...
n_seq = data.shape[0]
n_seq = 10000
data = data[0:n_seq]
split_point = int(data.shape[0] * 0.8) # Keep 80/20 split
np.random.shuffle(data) # In-place modification
max_length = np.max([len(i) for i in data.flatten()]) + 2 # Add start/stop

print(f"max_length = ${max_length}")
print(f"data[0] => ${data[0]}")

# Setup
i_to_c_eng = ['','<START>','<STOP>'] + list({char for word in data[:,0] for char in word})
c_to_i_eng = {i_to_c_eng[i]:i for i in range(len(i_to_c_eng))}
i_to_c_eng[1] = i_to_c_eng[2] = ''

i_to_c_por = ['','<START>','<STOP>'] + list({char for word in data[:,1] for char in word})
```

```
c_to_i_por = {i_to_c_por[i]:i for i in range(len(i_to_c_por))}
i_to_c_por[1] = i_to_c_por[2] = ''


def encode_seq(x, mapping, max_length=0):
    # String to integer
    return [mapping['<START>']] + \
    [mapping[i] for i in list(x)] + \
    [mapping['<STOP>']] + \
    [0]*(max_length-len(list(x))-2)

def decode_seq(x, mapping):
    # Integer-to-string
    try:
        idx = list(x).index(2) # Stop token?
    except:
        idx = len(list(x)) # No stop token found
    return ''.join([mapping[i] for i in list(x)[0:idx]])



# Setup
X = np.vstack([encode_seq(x, c_to_i_eng, max_length) for x in data[:,0]])
Y = np.vstack([encode_seq(x, c_to_i_por, max_length) for x in data[:,1]])

enc_x_train = X[:split_point]
enc_x_val = X[split_point:]
enc_x_train

dec_x_train = Y[:,0:-1][:split_point]
dec_x_val = Y[:,0:-1][split_point:]
dec_x_train

dec_y_train = Y[:,1:][:split_point]
dec_y_val = Y[:,1:][split_point:]
dec_y_train

print(enc_x_train.shape)
print(dec_x_train.shape)
print(dec_y_train.shape)

print("----")

print(enc_x_val.shape)
print(dec_x_val.shape)
print(dec_y_val.shape)


class RecurrentResidual(torch.nn.Module):
    def __init__(self,
                 latent_size = 64,
                 bidirectional = False,
                 **kwargs):
        super().__init__(**kwargs)
        self.layer_norm = torch.nn.LayerNorm(latent_size)
        self.rnn_layer = torch.nn.RNN(latent_size,
                                      latent_size // 2 if bidirectional else latent_size,
                                      bidirectional=bidirectional,
                                      nonlinearity='tanh',
                                      batch_first=True)
    def forward(self, x):
        return x + self.rnn_layer(self.layer_norm(x))[0]
```

```
class EncoderNetwork(torch.nn.Module):
    def __init__(self,
                 num_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(**kwargs)
        self.embedding = torch.nn.Embedding(num_tokens,
                                            latent_size,
                                            padding_idx=0)
        self.dropout = torch.nn.Dropout1d(0.1) # Whole token dropped
        self.rnn_layers = torch.nn.Sequential(*[
            RecurrentResidual(latent_size, True) for _ in range(n_layers)
        ])

    def forward(self, x):
        y = x
        y = self.embedding(y)
        y = self.dropout(y)
        y = self.rnn_layers(y)[:,-1]
        return y

enc_net = EncoderNetwork(num_tokens=len(i_to_c_eng))
summary(enc_net,input_data=torch.Tensor(enc_x_train[0:5]).long())


# Decoder Component
class DecoderNetwork(torch.nn.Module):
    def __init__(self,
                 num_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(**kwargs)
        self.embedding = torch.nn.Embedding(num_tokens,
                                            latent_size,
                                            padding_idx=0)
        self.dropout = torch.nn.Dropout1d(0.1) # Whole token dropped
        self.linear = torch.nn.Linear(latent_size*2, latent_size)
        self.rnn_layers = torch.nn.Sequential(*[
            RecurrentResidual(latent_size,False) for _ in range(n_layers)
        ])
        self.output_layer = torch.nn.Linear(latent_size,
                                            num_tokens)

    def forward(self, x_enc, x_dec):
        y_enc = x_enc.unsqueeze(1).repeat(1,x_dec.shape[1],1)
        y_dec = self.embedding(x_dec)
        y_dec = self.dropout(y_dec)
        y = y_enc
        y = torch.concatenate([y_enc,y_dec],-1)
        y = self.linear(y)
        y = self.rnn_layers(y)
        y = self.output_layer(y)
        return y

dec_net = DecoderNetwork(num_tokens=len(i_to_c_por))
summary(dec_net,input_data=[enc_net(torch.Tensor(enc_x_train[0:5]).long()).cpu(), torch.Tensor
(dec_x_train[0:5]).long()])


class EncDecLightningModule(pl.LightningModule):
    def __init__(self,
                 output_size,
                 **kwargs):
        super().__init__(**kwargs)
        self.mc_acc = torchmetrics.classification.Accuracy(task='multiclass',
                                                           num_classes=output_size,
                                                           ignore_index=0)
        self.cce_loss = torch.nn.CrossEntropyLoss(ignore_index=0)

    def predict(self, x):
        return torch.softmax(self(x),-1)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        return optimizer

    def training_step(self, train_batch, batch_idx):
        x_enc, x_dec, y_dec = train_batch
        y_pred = self(x_enc, x_dec)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('train_acc', acc, on_step=False, on_epoch=True)
        self.log('train_loss', loss, on_step=False, on_epoch=True)
        return loss

    # Validate used for Teacher Forcing
    def validation_step(self, val_batch, batch_idx):
        x_enc, x_dec, y_dec = val_batch
        y_pred = self(x_enc, x_dec)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('val_acc', acc, on_step=False, on_epoch=True)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        return loss

    # Test used for Non-Teacher Forcing
    def test_step(self, test_batch, batch_idx):
        x_enc, x_dec, y_dec = test_batch
        context = self.enc_net(x_enc)
        tokens = torch.zeros_like(x_dec).long()
        tokens[:,0] = 1
        for i in range(y_dec.shape[1]-1):
            tokens[:,i+1] = self.dec_net(context, tokens).argmax(-1)[:,i]
        y_pred = self(x_enc, tokens)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_dec)
        loss = self.cce_loss(y_pred.permute(*perm),y_dec)
        self.log('test_acc', acc, on_step=False, on_epoch=True)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        return loss

class EncDecNetwork(EncDecLightningModule):
    def __init__(self,
                 num_enc_tokens,
                 num_dec_tokens,
                 latent_size = 64, # Use something divisible by 2
                 n_layers = 4,
                 **kwargs):
        super().__init__(output_size=num_dec_tokens, **kwargs)
        self.enc_net = EncoderNetwork(num_enc_tokens,latent_size,n_layers)
        self.dec_net = DecoderNetwork(num_dec_tokens,latent_size,n_layers)
```

Same as with the parity problem - no adjustment of embedding lengths (-5) (100 for translation, 20 for parity)

```
    def forward(self, x_enc, x_dec):
        return self.dec_net(self.enc_net(x_enc), x_dec)


enc_dec_net = EncDecNetwork(num_enc_tokens=len(i_to_c_eng),
                            num_dec_tokens=len(i_to_c_por))

summary(enc_dec_net,input_data=[torch.Tensor(enc_x_train[0:1]).long(),
                                torch.Tensor(dec_x_train[0:1]).long()])


xy_train = torch.utils.data.DataLoader(list(zip(torch.Tensor(enc_x_train).long(),
                                                torch.Tensor(dec_x_train).long(),
                                                torch.Tensor(dec_y_train).long())),
                                       shuffle=True,
                                       batch_size=cfg_batch_size,
                                       num_workers=cfg_num_workers)

xy_val = torch.utils.data.DataLoader(list(zip(torch.Tensor(enc_x_val).long(),
                                              torch.Tensor(dec_x_val).long(),
                                              torch.Tensor(dec_y_val).long())),
                                     shuffle=False,
                                     batch_size=cfg_batch_size,
                                     num_workers=cfg_num_workers)


logger = pl.loggers.CSVLogger(cfg_logger_dir,
                              name=cfg_logger_name,
                              version=cfg_logger_version)

trainer = pl.Trainer(logger=logger,
                     max_epochs=cfg_max_epochs,
                     enable_progress_bar=True,
                     log_every_n_steps=0,
                     enable_checkpointing=False,
                     callbacks=[pl.callbacks.TQDMProgressBar(refresh_rate=50)])


# Train Model
trainer.fit(enc_dec_net, xy_train, xy_val)

# Test Model
results = trainer.test(enc_dec_net, xy_val)

# Hanlding Timing
end = time.time()
elapsed = end - start

print("")
print(f"Processing Time: {elapsed:.6f} seconds\n")
print("Test Accuracy:", results[0]['test_acc'])
print("")
print("")
```