Model Persistence - Loading

Auxiliary materials

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
import pandas as pd
import matplotlib.pyplot as plt
```

Iris Net

In [2]:

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

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

    def validation_step(self, val_batch, batch_idx):
        x, y_true = val_batch
        y_pred = self(x)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_true)
        loss = self.cce_loss(y_pred.permute(*perm),y_true)
        self.log('val_acc', acc, on_step=False, on_epoch=True)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        return loss

    def test_step(self, test_batch, batch_idx):
        x, y_true = test_batch
        y_pred = self(x)
        perm = (0,-1) + tuple(range(y_pred.ndim))[1:-1]
        acc = self.mc_acc(y_pred.permute(*perm),y_true)
        loss = self.cce_loss(y_pred.permute(*perm),y_true)
        self.log('test_acc', acc, on_step=False, on_epoch=True)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        return loss
```

In [3]:

```
class MultiLayerNetwork(MultiClassModule):
    def __init__(self,
                 input_size,
                 output_size,
                 hidden_size = 64,
                 **kwargs):
        super().__init__(output_size=output_size,
                         **kwargs)
        self.save_hyperparameters()
        self.hidden = torch.nn.Linear(input_size,
                                      hidden_size)
        self.output = torch.nn.Linear(hidden_size,
                                      output_size)

    def forward(self, x):
        y = x
        y = torch.tanh(self.hidden(y))
        y = self.output(y)
        return y
```

In [4]:

```
data = np.loadtxt("https://www.cs.mtsu.edu/~jphillips/courses/CSCI4850-5850/publ
np.random.seed(0)
np.random.shuffle(data)
X = data[:,:-1]
Y = data[:,-1]
split_point = int(X.shape[0] * 0.8)
# The dataloaders handle shuffling, batching, etc...
xy_train = torch.utils.data.DataLoader(list(zip(torch.tensor(X[:split_point]).fl
                                                torch.tensor(Y[:split_point]).lo
                                       shuffle=True, batch_size=32,num_workers=4
xy_val = torch.utils.data.DataLoader(list(zip(torch.tensor(X[split_point:]).floa
                                              torch.tensor(Y[split_point:]).long
                                     shuffle=False, batch_size=32,num_workers=
```

```
/opt/conda/lib/python3.11/site-packages/torch/utils/data/dataloader.py:557: UserW
arning: This DataLoader will create 4 worker processes in total. Our suggested ma
x number of worker in current system is 2, which is smaller than what this DataLo
ader is going to create. Please be aware that excessive worker creation might get
DataLoader running slow or even freeze, lower the worker number to avoid potentia
l slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
```

In [5]:

```
iris_net = MultiLayerNetwork.load_from_checkpoint("iris_net.ckpt")

# Or use the one made by Lightning automatically...
# iris_net = MultiLayerNetwork.load_from_checkpoint("logs/persistence/iris_net-0
```

In [6]:

```
x = torch.Tensor(X[0:1]).to(iris_net.device)
x
```

Out[6]:

```
tensor([[6.4000, 2.9000, 4.3000, 1.3000]])
```

In [7]:

```
iris_net.predict(x).cpu().detach().numpy()
```

Out[7]:

```
array([[0.02313578, 0.9282432 , 0.048621  ]], dtype=float32)
```

In [8]:

```
Y[0:1]
```

Out[8]:

```
array([1.])
```

Just validation...

In [9]:

```
logger = pl.loggers.CSVLogger("logs",
                              name="persistence")
```

In [10]:

```
trainer = pl.Trainer(logger=logger,
                     max_epochs=100,
                     enable_progress_bar=True,
                     log_every_n_steps=0,
                     enable_checkpointing=True, # Notice this!
                     callbacks=[pl.callbacks.TQDMProgressBar(refresh_rate=50)])
```

```
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
```

This result should match the previous result which indicates successful loading of the saved model.

In [11]:

```
trainer.validate(iris_net, xy_val)
```

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
SLURM auto-requeueing enabled. Setting signal handlers.
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃       Validate metric        ┃         DataLoader 0         ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│           val_acc            │             1.0              │
│           val_loss           │     0.12450059503316879      │
└──────────────────────────────┴──────────────────────────────┘
```

Out[11]:

```
[{'val_acc': 1.0, 'val_loss': 0.12450059503316879}]
```

Encoder-Decoder

In [12]:

```
# Tokenization Tools
import os
os.environ["TOKENIZERS_PARALLELISM"] = "false"
import tokenizers
import io
```

In [13]:

```
import urllib
data = []
my_url = "https://raw.githubusercontent.com/luisroque/deep-learning-articles/mai
with urllib.request.urlopen(my_url) as raw_data:
    for line in raw_data:
        data.append(line.decode("utf-8").split('\t')[0:2])
data = np.array(data)
```

In [14]:

```
data.shape
```

Out[14]:

```
(170304, 2)
```

In [15]:

```
# 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.seed(0)
np.random.shuffle(data) # In-place modification
```

In [16]:

```
data[0]
```

Out[16]:

```
array(['These are real.', 'Estas são autênticas.'], dtype='<U184')
```

In [17]:

```
eng = data[:,0]
por = data[:,1]
```

In [18]:

```
eng.shape
```

Out[18]:

```
(10000,)
```

In [19]:

```
eng[0:5]
```

Out[19]:

```
array(['These are real.', "I'm sorry.", 'I have wine.', "I won't do it.",
       'I eat bread.'], dtype='<U184')
```

In [20]:

```
por.shape
```

Out[20]:

```
(10000,)
```

In [21]:

```
por[0:5]
```

Out[21]:

```
array(['Estas são autênticas.', 'Desculpe!', 'Tenho vinho.',
       'Eu não irei fazer isso.', 'Eu como pão.'], dtype='<U184')
```

Pretrained tokenizer from the persisted JSON file

In [22]:

```
eng_tokenizer = tokenizers.Tokenizer(tokenizers.models.BPE()).from_file("eng_tra
por_tokenizer = tokenizers.Tokenizer(tokenizers.models.BPE()).from_file("por_tra
```

In [23]:

```
eng_tokenizer.encode("Here is a test.").tokens
```

Out[23]:

```
['Here', 'is', 'a', 'test', '.']
```

In [24]:

```
temp = eng_tokenizer.encode("Here is a test.").ids
temp
```

Out[24]:

```
[443, 78, 46, 3165, 9]
```

In [25]:
```
temp = eng_tokenizer.decode(temp + [0,0])
temp
```

Out[25]:
```
'Here is a test .'
```

In [26]:
```
eng_tokenizer.get_vocab_size()
```

Out[26]:
```
3694
```

In [27]:
```
por_tokenizer.get_vocab_size()
```

Out[27]:
```
6459
```

In [28]:
```
eng_recoded = np.array([eng_tokenizer.decode(eng_tokenizer.encode(s).ids) for s
por_recoded = np.array([por_tokenizer.decode(por_tokenizer.encode(s).ids) for s
```

In [29]:
```
eng_recoded[0]
```

Out[29]:
```
'These are real .'
```

In [30]:
```
eng[0]
```

Out[30]:
```
'These are real.'
```

In [31]:
```
por_recoded[0]
```

Out[31]:
```
'Estas são autênticas .'
```

In [32]:
```
por[0]
```

Out[32]:
```
'Estas são autênticas.'
```

In [33]:
```
def encode_seq(x,tokenizer,max_length=0):
    # String to integer
    x = tokenizer.encode("<START>"+x+"<STOP>").ids
    x += [0]*(max_length-len(x))
    return x

def decode_seq(x,tokenizer):
    return tokenizer.decode(x)
```

In [34]:
```
temp = encode_seq(eng_recoded[0],eng_tokenizer,20)
temp
```

Out[34]:
```
[1, 425, 140, 442, 9, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

In [35]:
```
len(temp)
```

Out[35]:
```
20
```

In [36]:
```
decode_seq(temp,eng_tokenizer)
```

Out[36]:
```
'These are real .'
```

In [37]:
```
temp = encode_seq(por_recoded[0],por_tokenizer,20)
```

```
temp
```

Out[37]:

```
[1, 862, 229, 6063, 8, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

In [38]:

```
decode_seq(temp,por_tokenizer)
```

Out[38]:

```
'Estas são autênticas .'
```

In [39]:

```
max_eng = np.max([len(encode_seq(i,eng_tokenizer)) for i in eng_recoded])
max_eng
```

Out[39]:

```
12
```

In [40]:

```
max_por = np.max([len(encode_seq(i,por_tokenizer)) for i in por_recoded])
max_por
```

Out[40]:

```
11
```

In [41]:

```
X = np.vstack([encode_seq(x,eng_tokenizer,max_eng) for x in eng_recoded])
Y = np.vstack([encode_seq(x,por_tokenizer,max_por) for x in por_recoded])
```

In [42]:

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

Out[42]:

```
array([[   1,  425,  140, ...,    0,    0,    0],
       [   1,   30,    6, ...,    0,    0,    0],
       [   1,   30,  152, ...,    0,    0,    0],
       ...,
       [   1,   90,    6, ...,    0,    0,    0],
       [   1,   30,    6, ...,    0,    0,    0],
       [   1, 1592,   21, ...,    0,    0,    0]])
```

In [43]:

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

Out[43]:

```
array([[   1,  862,  229, ...,    0,    0,    0],
       [   1, 3279,    3, ...,    0,    0,    0],
       [   1,  352,  719, ...,    0,    0,    0],
       ...,
       [   1,  141,  416, ...,    0,    0,    0],
       [   1,  188,  850, ...,    0,    0,    0],
       [   1, 3776,   19, ...,    0,    0,    0]])
```

In [44]:

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

Out[44]:

```
array([[ 862,  229, 6063, ...,    0,    0,    0],
       [3279,    3,    2, ...,    0,    0,    0],
       [ 352,  719,    8, ...,    0,    0,    0],
       ...,
       [ 141,  416, 3597, ...,    0,    0,    0],
       [ 188,  850,    8, ...,    0,    0,    0],
       [3776,   19,    2, ...,    0,    0,    0]])
```

In [45]:

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

```
(8000, 12)
(8000, 10)
(8000, 10)
```

In [46]:
```
print(enc_x_val.shape)
print(dec_x_val.shape)
print(dec_y_val.shape)
```

```
(2000, 12)
(2000, 10)
(2000, 10)
```

In [47]:
```
batch_size = 256
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=batch_size,
                                       num_workers=4)
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=batch_size,
                                     num_workers=4)
```

Model Loading...

In [48]:
```
class RecurrentResidual(pl.LightningModule):
    def __init__(self,
                 latent_size = 256,
                 bidirectional = False,
                 **kwargs):
        super().__init__(**kwargs)
        self.layer_norm = torch.nn.LayerNorm(latent_size)
        self.rnn_layer = torch.nn.LSTM(latent_size,
                                       latent_size // 2 if bidirectional else la
                                       bidirectional=bidirectional,
                                       batch_first=True)
    def forward(self, x):
        return x + self.rnn_layer(self.layer_norm(x))[0]
```

In [49]:
```
class EncoderNetwork(pl.LightningModule):
    def __init__(self,
                 num_tokens,
                 latent_size = 256, # Use something divisible by 2
                 n_layers = 8,
                 **kwargs):
        super().__init__(**kwargs)
        self.embedding = torch.nn.Embedding(num_tokens,
                                            latent_size,
                                            padding_idx=0)
        self.dropout = torch.nn.Dropout1d(0.05) # 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
```

In [50]:

```
class DecoderNetwork(pl.LightningModule):
    def __init__(self,
                 num_tokens,
                 latent_size = 256, # Use something divisible by 2
                 n_layers = 8,
                 **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
```

In [51]:

```
class EncDecLightningModule(pl.LightningModule):
    def __init__(self,
                 output_size,
                 **kwargs):
        super().__init__(**kwargs)
        self.mc_acc = torchmetrics.classification.Accuracy(task='multiclass',
                                                           num_classes=output_si
                                                           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)
```

```python
        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

    def predict_step(self, predict_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)
        return y_pred
```

In [52]:

```python
class EncDecNetwork(EncDecLightningModule):
    def __init__(self,
                 num_enc_tokens,
                 num_dec_tokens,
                 latent_size = 256, # Use something divisible by 2
                 n_layers = 8,
                 **kwargs):
        super().__init__(output_size=num_dec_tokens,
                         **kwargs)
        self.save_hyperparameters()
        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)
```

In [53]:

```python
enc_dec_net = EncDecNetwork.load_from_checkpoint("enc_dec_net.ckpt")
```

Test without Teacher Forcing

In [54]:

```
# Complete max_length cycles with the decoder
i = 0
enc_dec_net.to("cpu")
context = enc_dec_net.enc_net(torch.Tensor(enc_x_val[i:i+1]).long())
token = torch.zeros((1,dec_y_val.shape[1])).long()
token[0,0] = 1

for x in range(dec_y_val.shape[1]-1):
    result = enc_dec_net.dec_net(context,token).argmax(-1)
    token[0,x+1] = result[0,x]
    if result[0,x] == 2:
        break
result = token.cpu().detach().numpy()[0]
result
```

Out[54]:

```
array([   1, 5549,  514,   19,    2,    0,    0,    0,    0,    0])
```

English input...

In [55]:

```
decode_seq(enc_x_val[i],eng_tokenizer)
```

Out[55]:

```
'Can you swim ?'
```

Portuguese translation from network...

In [56]:

```
decode_seq(result,por_tokenizer)
```

Out[56]:

```
'Sabes nadar ?'
```

Target translation from the data set...

In [57]:

```
decode_seq(dec_y_val[i],por_tokenizer)
```

Out[57]:

```
'Sabe nadar ?'
```

In [58]:

```
result.shape
```

Out[58]:

```
(10,)
```

In [59]:

```
dec_y_val.shape
```

Out[59]:

```
(2000, 10)
```

Accuracy **without** teacher forcing...

In [60]:

```
logger = pl.loggers.CSVLogger("logs",
                              name="persistence")
```

In [61]:

```
trainer = pl.Trainer(logger=logger,
                     max_epochs=30,
                     enable_progress_bar=True,
                     log_every_n_steps=0,
                     enable_checkpointing=True, # Notice this here!
                     callbacks=[pl.callbacks.TQDMProgressBar(refresh_rate=50)])
```

```
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
```

This result should match the previous result which indicates successful loading of the saved model.

In [62]:

```
trainer.test(enc_dec_net, xy_val)
```

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
SLURM auto-requeueing enabled. Setting signal handlers.
Testing: |          | 0/? [00:00<?, ?it/s]
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃        Test metric        ┃       DataLoader 0        ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│         test_acc          │    0.4213317930698395     │
│         test_loss         │     6.924007892608643     │
└───────────────────────────┴───────────────────────────┘
```

Out[62]:

```
[{'test_acc': 0.4213317930698395, 'test_loss': 6.924007892608643}]
```

In [ ]: