**Problem Definition**

Recurrent Neural Networks (RNNs) have shown a lot of potential in many natural language processing (NLP) and sequence learning tasks. The main idea behind sequential learning is that they make use of sequential information. In traditional feed-forward neural networks such as Convolutional Neural Networks (CNNs), it is assumed that all inputs are independent of each other, e.g., in an image classification task, the pixels of an image are independent of each other. However, this approach is not valid for sequence learning tasks. For instance, if you want to predict the next word in a sequence (sentence), you need the prior information (i.e., previous words in the sentence) to do so. RNNs can remember previous states (information), i.e., RNNs have a “memory” in the form of hidden states which store information about what has been processed so far. At each input of the sequence, the model not only takes the current input but also remembers the preceding information. Like the human way of processing sequential information, this allows the model to learn long-term dependencies in the sequence, i.e., it considers the entire context when making a prediction.

Prediction of the next alarm can also be modelled as a sequence learning task: given a sorted sequence of alarms based on their start time, predict the upcoming alarm. In this report, we compared a RNN architecture and a Transfomrer architecture in terms of accuracy of predicitng next alarm.

By predicting future alarms in real-time with the help of the AI module, the operator may avert abnormal situations by taking corrective actions or prepare for such situations in advance.

*# # for logging*

**from** **comet\_ml** **import** Experiment

**from** **pytorch\_lightning.loggers** **import** CometLogger

**from** **pytorch\_lightning.loggers** **import** TestTubeLogger

**from** **pytorch\_lightning.loggers** **import** WandbLogger

**import** **wandb**

*# For metrics*

**from** **pytorch\_lightning** **import** metrics

**import** **math**

**import** **torch**

**import** **torch.nn.functional** **as** **F**

**from** **torch.utils.data** **import** Dataset, DataLoader

**import** **io**

**import** **torchtext**

**from** **torchtext.utils** **import** download\_from\_url, extract\_archive

**from** **torchtext.data.utils** **import** get\_tokenizer

**from** **torchtext.vocab** **import** build\_vocab\_from\_iterator

**import** **pytorch\_lightning** **as** **pl**

**from** **pytorch\_lightning.trainer.trainer** **import** Trainer

**from** **pytorch\_lightning.callbacks.early\_stopping** **import** EarlyStopping *# The EarlyStopping callback can be used to monitor a validation metric and stop the training when no improvement is observed.*

*"""*

*To enable it:*

*Import EarlyStopping callback.*

*Log the metric you want to monitor using log() method.*

*Init the callback, and set monitor to the logged metric of your choice.*

*Pass the EarlyStopping callback to the Trainer callbacks flag.*

*"""*

**from** **pytorch\_lightning** **import** seed\_everything

seed\_everything(42)

**Performance Metrics**

**from** **pytorch\_lightning.metrics** **import** Metric

**from** **pytorch\_lightning.metrics.utils** **import** \_input\_format\_classification

**from** **sklearn.metrics** **import** classification\_report

**class** **MyClassificationReport**(Metric):

**def** \_\_init\_\_(self,threshold: float = 0.5,compute\_on\_step: bool = **True**,dist\_sync\_on\_step: bool = **False**):

super().\_\_init\_\_(

compute\_on\_step=compute\_on\_step,

dist\_sync\_on\_step=dist\_sync\_on\_step,

)

self.threshold = threshold

self.add\_state("preds", default=[], dist\_reduce\_fx=**None**)

self.add\_state("target", default=[], dist\_reduce\_fx=**None**)

*# rank\_zero\_warn(*

*# 'Metric `MyClassificationReport` will save all targets and predictions in buffer.'*

*# ' For large datasets this may lead to large memory footprint.'*

*# )*

**def** update(self, preds: torch.Tensor, target: torch.Tensor):

preds = preds.cpu()

target = target.cpu()

y\_hat, y = preds.max(1).indices, target

**assert** y\_hat.shape == y.shape

self.preds.append(y\_hat)

self.target.append(y)

**def** compute(self):

preds = torch.cat(self.preds, dim=0)

target = torch.cat(self.target, dim=0)

**Dataset Preparation**

**from** **sklearn.model\_selection** **import** train\_test\_split

**class** **AlarmDataset**(Dataset):

**def** \_\_init\_\_(self,data,seq\_len,batch\_size):

self.length = len(data)//seq\_len *# how much data i have*

self.data = data

self.seq\_len = seq\_len

self.batch\_size = batch\_size

**def** \_\_getitem\_\_(self, index: int):

x = self.data[index\*self.seq\_len:(index\*self.seq\_len)+self.seq\_len]

y = self.data[1+index\*self.seq\_len:1+(index\*self.seq\_len)+self.seq\_len]

**return** x,y

**def** \_\_len\_\_(self) -> int:

**return** self.length

**class** **MyDataModule**(pl.LightningDataModule):

**def** \_\_init\_\_(self,config):

super().\_\_init\_\_()

self.config = config

dir\_path = self.config['dir-path']

file\_name = 'train.tokens'

self.tokenizer = get\_tokenizer('basic\_english')

self.vocab = build\_vocab\_from\_iterator(map(self.tokenizer,iter(io.open(dir\_path+file\_name,encoding="utf8"))))

train\_data = self.data\_process(iter(io.open(dir\_path +"train.tokens", encoding="utf8")))

val\_data = self.data\_process(iter(io.open(dir\_path +"val.tokens", encoding="utf8")))

test\_data = self.data\_process(iter(io.open(dir\_path +"test.tokens", encoding="utf8")))

self.train\_dataset = AlarmDataset(train\_data, self.config['seq-len'], self.config['batch-size'])

self.valid\_dataset = AlarmDataset(val\_data,self.config['seq-len'], self.config['batch-size'])

self.test\_dataset = AlarmDataset(test\_data, self.config['seq-len'], self.config['batch-size'])

**def** data\_process(self, raw\_text\_iter):

data = [torch.tensor([self.vocab[token] **for** token **in** self.tokenizer(item)],dtype=torch.long) **for** item **in** raw\_text\_iter]

**return** torch.cat(tuple(filter(**lambda** t: t.numel() > 0, data)))

**def** get\_weight\_per\_class(self):

**def** lambdaFun(total,v,num\_classes):

**if** v>0:

**return** total/(v\*num\_classes)

**return** 0

index\_2\_count = {self.vocab.stoi[k]:self.vocab.freqs[k] **for** k **in** list(self.vocab.stoi)}

total = sum(index\_2\_count.values())

index\_2\_ws = {k:lambdaFun(total,v,len(index\_2\_count)) **for** k,v **in** index\_2\_count.items()}

index\_2\_ws[1] = 0.0 *# MANUALLY Setting the weights to zero for the padding*

*# index\_2\_ws[0] = 0.0 # MANUALLY Setting the weights to zero for the padding*

ws = torch.tensor([index\_2\_ws[i] **for** i **in** range(len(index\_2\_ws))])

**return** ws

**def** prepare\_data(self):

*"""*

*Use this method to do things that might write to disk or that need to be done only from a single GPU in distributed settings.*

*e.g., download,tokenize,etc…*

*"""*

**return** **None**

**def** setup(self, stage: **None**):

*"""*

*There are also data operations you might want to perform on every GPU. Use setup to do things like:*

*count number of classes,build vocabulary,perform train/val/test splits,apply transforms (defined explicitly in your datamodule or assigned in init),etc…*

*"""*

**return** **None**

**def** train\_dataloader(self) -> DataLoader:

**return** DataLoader(self.train\_dataset, batch\_size=self.config['batch-size'], shuffle=**False**,num\_workers=8,drop\_last=**True**, pin\_memory=**True**)

**def** val\_dataloader(self) -> DataLoader:

**return** DataLoader(self.valid\_dataset, batch\_size=self.config['batch-size'], shuffle=**False**,num\_workers=8,drop\_last=**True**, pin\_memory=**True**)

**def** test\_dataloader(self) -> DataLoader:

**return** DataLoader(self.test\_dataset, batch\_size=self.config['batch-size'], shuffle=**False**,num\_workers=8,drop\_last=**True**, pin\_memory=**True**)

**Transformer Model**

The alarm modeling task is to assign a probability for the likelihood of a given sequence of words to follow a next alarm. A sequence of tokens are passed to the embedding layer first, followed by a positional encoding layer to account for the order of the word. The nn.TransformerEncoder consists of multiple layers of nn.TransformerEncoderLayer <https://pytorch.org/docs/master/nn.html?highlight=transformerencoderlayer#torch.nn.TransformerEncoderLayer>\_\_. Along with the input sequence, a square attention mask is required because the self-attention layers in nn.TransformerEncoder are only allowed to attend the earlier positions in the sequence. For the language modeling task, any tokens on the future positions should be masked. To have the actual alarms, the output of nn.TransformerEncoder model is sent to the final Linear layer, which is followed by a log-Softmax function.

**Positional Encoding**

PositionalEncoding module injects some information about the relative or absolute position of the tokens (i.e.,) in the sequence. The positional encodings have the same dimension as the embeddings so that the two can be summed. Here, we use sine and cosine functions of different frequencies.

**Loss Function**

CrossEntropyLoss is applied to track the loss and AdamWimplements stochastic gradient descent method as the optimizer.

**class** **PositionalEncoding**(torch.nn.Module):

**def** \_\_init\_\_(self, d\_model, dropout=0.1, max\_len=5000):

super(PositionalEncoding, self).\_\_init\_\_()

self.dropout = torch.nn.Dropout(p=dropout)

pe = torch.zeros(max\_len, d\_model)

position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1)

div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \* (-math.log(10000.0) / d\_model))

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

pe = pe.unsqueeze(0).transpose(0, 1)

self.register\_buffer('pe', pe)

**def** forward(self, x):

x = x + self.pe[:x.size(0), :]

**return** self.dropout(x)

**Transformer Architecture**

**class** **TransformerModel**(pl.LightningModule):

**def** \_\_init\_\_(self, config):

super(TransformerModel, self).\_\_init\_\_()

self.accuraccy\_50\_count = 0

self.config = config

self.lr = self.config["lr"]

self.weight\_decay = self.config["weight-decay"]

self.pos\_encoder = PositionalEncoding(self.config['em-size'], self.config['dropout'])

encoder\_layers = torch.nn.TransformerEncoderLayer(self.config['em-size'], self.config['nhead'], self.config['nhid'], self.config["dropout"])

self.transformer\_encoder = torch.nn.TransformerEncoder(encoder\_layers, self.config['nlayers'])

self.encoder = torch.nn.Embedding(self.config["vocab-size"], self.config['em-size'])

self.decoder = torch.nn.Linear(self.config['em-size'], self.config["vocab-size"])

self.src\_mask = self.generate\_square\_subsequent\_mask(self.config['seq-len'])

self.init\_weights()

self.class\_weight = self.config['weight\_per\_class']

*# self.train\_F1 = metrics.classification.F1(num\_classes=self.config["vocab-size"],average = 'micro')*

*# self.val\_F1 = metrics.classification.F1(num\_classes=self.config["vocab-size"],average = 'micro')*

*# self.test\_F1 = metrics.classification.F1(num\_classes=self.config["vocab-size"],average = 'micro')*

self.val\_CM\_normalized = metrics.classification.ConfusionMatrix(num\_classes=self.config["vocab-size"],normalize ='true')

self.val\_CM\_raw = metrics.classification.ConfusionMatrix(num\_classes=self.config["vocab-size"])

self.train\_CM\_normalized = metrics.classification.ConfusionMatrix(num\_classes=self.config["vocab-size"],normalize ='true')

self.train\_CM\_raw = metrics.classification.ConfusionMatrix(num\_classes=self.config["vocab-size"])

self.test\_CM = metrics.classification.ConfusionMatrix(num\_classes=self.config["vocab-size"],normalize ='true')

self.val\_MCR = MyClassificationReport()

self.test\_MCR = MyClassificationReport()

self.log("Sequence length",self.config['seq-len'])

self.log("lr",self.lr)

self.log("# of tokens/vocab\_size (unique alarms)",self.config['vocab-size'])

self.log("weight\_decay",self.weight\_decay)

self.save\_hyperparameters()

**def** generate\_square\_subsequent\_mask(self, sz):

mask = (torch.triu(torch.ones(sz, sz)) == 1).transpose(0, 1)

mask = mask.float().masked\_fill(mask == 0, float('-inf')).masked\_fill(mask == 1, float(0.0))

**return** mask

**def** init\_weights(self): *# initialize the weights to non zero number*

initrange = 0.1

self.encoder.weight.data.uniform\_(-initrange, initrange)

self.decoder.bias.data.zero\_()

self.decoder.weight.data.uniform\_(-initrange, initrange)

**def** forward(self, src, src\_mask):

src\_mask = src\_mask.to(self.device)

src = self.encoder(src) \* math.sqrt(self.config['em-size'])

src = self.pos\_encoder(src)

output = self.transformer\_encoder(src, src\_mask)

output = self.decoder(output)

**return** output

*# The ReduceLROnPlateau scheduler requires a monitor*

**def** configure\_optimizers(self):

optimizer = torch.optim.AdamW(self.parameters(), lr=self.lr,weight\_decay=self.weight\_decay)

d = {

'optimizer': optimizer,

'lr\_scheduler': torch.optim.lr\_scheduler.ReduceLROnPlateau(optimizer, mode = "min", factor = 0.5, patience=10, verbose=**True**),

'monitor': 'val\_epoch\_loss',

'interval': 'epoch'

}

**return** d

**def** loss\_function(self,logits,y):

**return** F.cross\_entropy(logits,y,weight= self.class\_weight,ignore\_index=1)

**def** myPrintToFile(self,cm\_normal,cm\_raw,f):

cm\_normal = cm\_normal.cpu()

cm\_raw = cm\_raw.cpu()

sum\_of\_each\_class = cm\_raw.sum(axis=1) *# sum along the columns*

print(f" ------ Epoch **{**self.current\_epoch**}** ---------",file=f)

print(f"Total=**{**[v.item() **for** v **in** sum\_of\_each\_class]**}**",file=f)

print(f"Corret=**{**[v.item() **for** v **in** torch.diagonal(cm\_raw,0)]**}**",file=f)

print(f"Accuracy=**{**[round(v.item(),3) **for** v **in** (torch.diagonal(cm\_raw,0)/sum\_of\_each\_class)]**}**",file=f)

accs = [round(v.item(),3) **for** v **in** torch.diagonal(cm\_normal,0)]

source2acc = {self.config['vocab'].itos[i]:accs[i] **for** i **in** range(len(accs))}

source2\_acc50 = {self.config['vocab'].itos[i]:accs[i] **for** i **in** range(len(accs)) **if** accs[i]>=0.5}

print(f"Acc2=**{**accs**}**",file=f)

print(f"source2\_acc= **{**source2acc**}**",file=f)

print(f"source2\_acc50= **{**source2\_acc50**}**",file=f)

a\_50 = len([a **for** a **in** accs **if** a>=0.5])

a\_30 = len([a **for** a **in** accs **if** a>=0.3])

out\_str = f"acc>0.5= **{**a\_50**}**, acc>=0.3= **{**a\_30**}**, Total=**{**len(accs)**}**"

print(out\_str,file=f)

*# if temp> self.accuraccy\_50\_count and train=:*

*# self.accuraccy\_50\_count = temp*

print(out\_str,end=" ")

**def** training\_step(self,batch,batch\_idx):

x,y = batch

x = x.T *# transpose*

y = y.T.reshape(-1)

**if** x.size(0) != self.config['seq-len']:

self.src\_mask = self.generate\_square\_subsequent\_mask(x.size(0))

y\_hat = self(x,self.src\_mask) *# calling forward method*

y\_hat = y\_hat.view(-1, self.config['vocab-size'])

loss = self.loss\_function(y\_hat,y) *# cross entropy itself compute softmax*

self.train\_CM\_normalized(F.softmax(y\_hat),y)

self.train\_CM\_raw(F.softmax(y\_hat),y)

self.log('train\_loss',loss,logger=**True**)

*# self.log('train\_F1',self.train\_F1(F.softmax(y\_hat),y),logger=True)*

**return** loss

**def** validation\_step(self,batch, batch\_idx):

x,y = batch

x = x.T

y = y.T.reshape(-1)

**if** x.size(0) != self.config['seq-len']:

*# print(f">> passed {x.size()}")*

self.src\_mask = self.generate\_square\_subsequent\_mask(x.size(0))

y\_hat = self(x,self.src\_mask)

y\_hat = y\_hat.view(-1, self.config['vocab-size'])

loss = self.loss\_function(y\_hat,y)

self.val\_MCR(F.softmax(y\_hat),y)

self.val\_CM\_normalized(F.softmax(y\_hat),y)

self.val\_CM\_raw(F.softmax(y\_hat),y)

self.log('val\_loss',loss,logger=**True**)

*# self.log('val\_F1',self.val\_F1(F.softmax(y\_hat) ,y),logger=True)*

**return** {'val\_loss':loss}

**def** test\_step(self,batch, batch\_idx):

x,y = batch

x = x.T

y = y.T.reshape(-1)

**if** x.size(0) != self.config['seq-len']:

self.src\_mask = self.generate\_square\_subsequent\_mask(x.size(0))

y\_hat = self(x,self.src\_mask)

y\_hat = y\_hat.view(-1, self.config['vocab-size'])

loss = self.loss\_function(y\_hat,y)

self.test\_MCR(F.softmax(y\_hat),y)

self.log('test\_loss',loss,logger=**True**)

*# self.log('test\_F1', self.test\_F1(F.softmax(y\_hat) ,y),logger=True)*

**return** {'test\_loss':loss}

**def** training\_epoch\_end(self, outputs):

avg\_loss = torch.stack([d['loss'] **for** d **in** outputs]).mean()

*# f1 = self.train\_F1.compute()*

print(f"[**{**self.current\_epoch**}**]E, Avg Training loss = **{**round(avg\_loss.item(),4)**}**",end=" ")

**with** open(self.config["train-file"],'a') **as** f:

self.myPrintToFile(self.train\_CM\_normalized.compute(),self.train\_CM\_raw.compute(),f)

self.log("train\_epoch\_loss",avg\_loss,logger=**True**,prog\_bar=**True**)

*# self.log("train\_epoch\_F1", f1, logger=True,prog\_bar=True)*

**def** validation\_epoch\_end(self, outputs):

avg\_loss = torch.stack([d['val\_loss'] **for** d **in** outputs]).mean()

*# f1 = self.val\_F1.compute()*

*# print(self.val\_MCR.compute(),file=open("val-out.txt",'w'))*

*# print(self.val\_CM.compute(),file=open("val-cm-out.txt",'w'))*

*# if self.current\_epoch%4==0 and self.current\_epoch>0:*

*# self.myPrintToFile(self.val\_CM\_normalized.compute(),self.val\_CM\_raw.compute())*

print(f"::Val Loss = **{**round(avg\_loss.item(),4) **}**",end=" ")

**with** open(self.config["val-file"],'a') **as** f:

self.myPrintToFile(self.val\_CM\_normalized.compute(),self.val\_CM\_raw.compute(),f)

print("")

self.log("val\_epoch\_loss",avg\_loss,logger=**True**)

*# self.log("val\_epoch\_F1",f1,logger=True,prog\_bar=True)*

**def** test\_epoch\_end(self, outputs):

avg\_loss = torch.stack([d['test\_loss'] **for** d **in** outputs]).mean()

*# f1 = self.test\_F1.compute()*

*# print(self.test\_MCR.compute(),file=open("test-out.txt",'w'))*

print(f">Average Test Loss = **{**avg\_loss.item()**}**")

self.log("test\_epoch\_loss",avg\_loss, logger = **True**)

*# self.log("test\_epoch\_F1",f1, logger=True)*

**RNN Architecture**

**class** **AlarmGRU**(pl.LightningModule):

**def** \_\_init\_\_(self,config):

*# super().\_\_init\_\_()*

super(AlarmGRU,self).\_\_init\_\_()

self.config =config

self.lr = self.config['lr']

self.val\_CM\_normalized = metrics.classification.ConfusionMatrix(num\_classes=self.config["vocab-size"],normalize ='true')

self.val\_CM\_raw = metrics.classification.ConfusionMatrix(num\_classes=self.config["vocab-size"])

self.train\_CM\_normalized = metrics.classification.ConfusionMatrix(num\_classes=self.config["vocab-size"],normalize ='true')

self.train\_CM\_raw = metrics.classification.ConfusionMatrix(num\_classes=self.config["vocab-size"])

self.test\_CM = metrics.classification.ConfusionMatrix(num\_classes=self.config["vocab-size"],normalize ='true')

self.val\_MCR = MyClassificationReport()

self.test\_MCR = MyClassificationReport()

*## TODO: define the layers of the model*

self.h = **None**

self.embedding = torch.nn.Embedding(self.config['vocab-size'],self.config['em-size'])

self.gru = torch.nn.GRU(input\_size=self.config['em-size'], hidden\_size=self.config['nhid'], num\_layers=self.config['nlayers'],dropout=self.config['dropout'], batch\_first=**True**)

*# self.droput = torch.nn.Dropout(p=self.drop\_prob)*

self.fc3 = torch.nn.Linear(in\_features=self.config['nhid'], out\_features=self.config['vocab-size'])

self.softmax = torch.nn.LogSoftmax(dim=1)

**def** \_\_init\_hidden(self):

*''' Initializes hidden state '''*

*# Create two new tensors with sizes n\_layers x batch\_size x n\_hidden,*

*# initialized to zero, for hidden state and cell state of GRU*

device = **None**

**if** (torch.cuda.is\_available()):

device = torch.device("cuda")

**else**:

device = torch.device("cpu")

weight = next(self.parameters()).data

hidden = weight.new(self.config['nlayers'], self.config['batch-size'], self.config['nhid']).zero\_().to(device)

**return** hidden

**def** initialize\_hidden(self):

self.h = self.\_\_init\_hidden()

**def** forward(self, x, hidden):

*''' Forward pass through the network.*

*These inputs are x, and the hidden/cell state `hidden`. '''*

*## TODO: Get the outputs and the new hidden state from the lstm*

x = x.long()

embeds = self.embedding(x)

out, hidden = self.gru(embeds,hidden)

out = out.contiguous().view(-1,self.config['nhid'])

out = self.fc3(out)

out = self.softmax(out)

*# return the final output and the hidden state*

**return** out, hidden

**def** configure\_optimizers(self):

optimizer = torch.optim.AdamW(self.parameters(), lr=self.lr,weight\_decay=self.config['weight-decay'])

d = {

'optimizer': optimizer,

'lr\_scheduler': torch.optim.lr\_scheduler.ReduceLROnPlateau(optimizer, mode = "min", factor = 0.5, patience=10, verbose=**True**),

'monitor': 'val\_epoch\_loss',

'interval': 'epoch'

}

**return** d

*# def loss\_function(self,logits,y):*

*# return F.cross\_entropy(logits,y,weight= self.class\_weight,ignore\_index=1)*

**def** myPrintToFile(self,cm\_normal,cm\_raw,f):

cm\_normal = cm\_normal.cpu()

cm\_raw = cm\_raw.cpu()

sum\_of\_each\_class = cm\_raw.sum(axis=1) *# sum along the columns*

print(f" ------ Epoch **{**self.current\_epoch**}** ---------",file=f)

print(f"Total=**{**[v.item() **for** v **in** sum\_of\_each\_class]**}**",file=f)

print(f"Corret=**{**[v.item() **for** v **in** torch.diagonal(cm\_raw,0)]**}**",file=f)

print(f"Accuracy=**{**[round(v.item(),3) **for** v **in** (torch.diagonal(cm\_raw,0)/sum\_of\_each\_class)]**}**",file=f)

accs = [round(v.item(),3) **for** v **in** torch.diagonal(cm\_normal,0)]

source2acc = {self.config['vocab'].itos[i]:accs[i] **for** i **in** range(len(accs))}

source2\_acc50 = {self.config['vocab'].itos[i]:accs[i] **for** i **in** range(len(accs)) **if** accs[i]>=0.5}

print(f"Acc2=**{**accs**}**",file=f)

print(f"source2\_acc= **{**source2acc**}**",file=f)

print(f"source2\_acc50= **{**source2\_acc50**}**",file=f)

a\_50 = len([a **for** a **in** accs **if** a>=0.5])

a\_30 = len([a **for** a **in** accs **if** a>=0.3])

out\_str = f"acc>0.5= **{**a\_50**}**, acc>=0.3= **{**a\_30**}**, Total=**{**len(accs)**}**"

print(out\_str,file=f)

*# if temp> self.accuraccy\_50\_count and train=:*

*# self.accuraccy\_50\_count = temp*

print(out\_str,end=" ")

**def** training\_step(self,batch,batch\_idx):

x,y = batch

y = y.view(self.config['batch-size']\*self.config['seq-len']).long()

self.h = self.h.data *# for GRU*

y\_hat, self.h = self(x,self.h)

*# ignore\_index=self.char2int["NoName"]*

loss = F.nll\_loss(y\_hat,y)

self.train\_CM\_normalized(y\_hat,y)

self.train\_CM\_raw(y\_hat,y)

*# result = pl.TrainResult(loss) # logging*

self.log('train\_loss',loss,logger=**True**)

**return** loss

**def** validation\_step(self,batch, batch\_idx):

x,y = batch

y = y.view(self.config['batch-size']\*self.config['seq-len']).long()

self.h = self.h.data *# for GRU*

y\_hat, self.h = self(x,self.h)

*# ignore\_index=self.char2int["NoName"]*

loss = F.nll\_loss(y\_hat,y)

self.val\_CM\_normalized(y\_hat,y)

self.val\_CM\_raw(y\_hat,y)

self.log('val\_loss',loss,logger=**True**)

**return** {'val\_loss':loss}

**def** test\_step(self,batch, batch\_idx):

x,y = batch

y = y.view(self.config['batch-size']\*self.config['seq-len']).long()

self.h = self.h.data *# for GRU*

y\_hat, self.h = self(x,self.h)

*# ignore\_index=self.char2int["NoName"]*

loss = F.nll\_loss(y\_hat,y)

self.val\_CM\_normalized(y\_hat,y)

self.val\_CM\_raw(y\_hat,y)

self.log('test\_loss',loss,logger=**True**)

**return** {'test\_loss':loss}

**def** training\_epoch\_end(self, outputs):

avg\_loss = torch.stack([d['loss'] **for** d **in** outputs]).mean()

*# f1 = self.train\_F1.compute()*

print(f"[**{**self.current\_epoch**}**]E, Avg Training loss = **{**round(avg\_loss.item(),4)**}**",end=" ")

**with** open(self.config["train-file"],'a') **as** f:

self.myPrintToFile(self.train\_CM\_normalized.compute(),self.train\_CM\_raw.compute(),f)

self.log("train\_epoch\_loss",avg\_loss,logger=**True**,prog\_bar=**True**)

*# self.log("train\_epoch\_F1", f1, logger=True,prog\_bar=True)*

**def** validation\_epoch\_end(self, outputs):

avg\_loss = torch.stack([d['val\_loss'] **for** d **in** outputs]).mean()

*# f1 = self.val\_F1.compute()*

*# print(self.val\_MCR.compute(),file=open("val-out.txt",'w'))*

*# print(self.val\_CM.compute(),file=open("val-cm-out.txt",'w'))*

*# if self.current\_epoch%4==0 and self.current\_epoch>0:*

*# self.myPrintToFile(self.val\_CM\_normalized.compute(),self.val\_CM\_raw.compute())*

print(f"::Val Loss = **{**round(avg\_loss.item(),4) **}**",end=" ")

**with** open(self.config["val-file"],'a') **as** f:

self.myPrintToFile(self.val\_CM\_normalized.compute(),self.val\_CM\_raw.compute(),f)

print("")

self.log("val\_epoch\_loss",avg\_loss,logger=**True**)

*# self.log("val\_epoch\_F1",f1,logger=True,prog\_bar=True)*

**def** test\_epoch\_end(self, outputs):

avg\_loss = torch.stack([d['test\_loss'] **for** d **in** outputs]).mean()

*# f1 = self.test\_F1.compute()*

*# print(self.test\_MCR.compute(),file=open("test-out.txt",'w'))*

print(f">Average Test Loss = **{**avg\_loss.item()**}**")

self.log("test\_epoch\_loss",avg\_loss, logger = **True**)

*# self.log("test\_epoch\_F1",f1, logger=True)*

**Training Transformers**

Note: When monitoring any parameter after the validation epoch end then you should pass check\_val\_every\_n\_epoch=1 not to other. This is very important.

**Finding the learning rate for the Transformer model**

**def** weightCondition(w,avg\_w):

**if** w<avg\_w:

**return** w

**else**:

**return** avg\_w

*# setup data*

config\_data = {

'dir-path' : "../.data/",

'batch-size' :512, *# Batch Size*

'seq-len' :12, *# Sequence length*

}

dm = MyDataModule(config=config\_data)

ws = dm.get\_weight\_per\_class().cuda()

print("Before",[round(w.item(),3) **for** w **in** ws])

*# avg\_w = sum(ws)/len(ws)*

*# ws = torch.tensor([weightCondition(w,avg\_w) for w in ws]).cuda()*

print("After",[round(w.item(),3) **for** w **in** ws])

config\_model = {

'lr' : 0.001,

'dropout' : 0.2,

'weight-decay': 3.1,

'em-size' :256, *# embedding dimension*

'nhid' : 128, *# the dimension of the feedforward network model in nn.TransformerEncoder*

'nlayers' :4, *# the number of nn.TransformerEncoderLayer in nn.TransformerEncoder*

'nhead' : 2, *# the number of heads in the multiheadattention models*

'seq-len': config\_data['seq-len'], *# dont use wandb config*

'vocab-size':len(dm.vocab.stoi), *# the size of vocabulary /also called tokens*

'weight\_per\_class':ws,

"val-file":"val-out.txt",

"train-file":'train-out.txt',

"vocab": dm.vocab

}

**with** open (config\_model["val-file"],'w') **as** f:

f.write(">> Starting")

**with** open (config\_model["train-file"],'w') **as** f:

f.write(">> Starting")

*# setup model - note how we refer to sweep parameters with wandb.config*

model = TransformerModel(config=config\_model)

early\_stop\_callback = EarlyStopping(

monitor='val\_epoch\_loss',

min\_delta=0,

patience=600,

verbose=**True**,

mode='min'

)

trainer = Trainer(auto\_lr\_find=0.0001, precision=16,gpus=-1, num\_nodes=1, max\_epochs=100, check\_val\_every\_n\_epoch=1,deterministic=**True**,gradient\_clip\_val=0.5,enable\_pl\_optimizer=**True**,callbacks=[early\_stop\_callback],progress\_bar\_refresh\_rate=0)

*# Run learning rate finder*

lr\_finder = trainer.tuner.lr\_find(model,dm)

*# Results can be found in*

lr\_finder.results

*# Plot with*

fig = lr\_finder.plot(suggest=**True**)

fig.show()

*# Pick point based on plot, or get suggestion*

new\_lr = lr\_finder.suggestion()

print(f"Suggested lr = **{**new\_lr**}**")

model.hparams.lr = new\_lr/100 *#7.5e-12 # can devide by 10*

trainer.fit(model,dm) *# Fit model*

*# trainer.test(datamodule=dm) # testing*

**Training RNNs**

Finding the learning rate for the RNN model.

*# setup data*

config\_data = {

'dir-path' : "../.data/",

'batch-size' :512, *# Batch Size*

'seq-len' :128, *# Sequence length change to 6*

}

dm = MyDataModule(config=config\_data)

ws = dm.get\_weight\_per\_class().cuda()

print("Before",[round(w.item(),3) **for** w **in** ws])

*# avg\_w = sum(ws)/len(ws)*

*# ws = torch.tensor([weightCondition(w,avg\_w) for w in ws]).cuda()*

print("After",[round(w.item(),3) **for** w **in** ws])

config\_model = {

'lr' : 0.001,

'dropout' : 0.05,

'weight-decay': 3.1, *#3.1,*

'em-size' :256, *# embedding dimension*

'nhid' : 512, *# the dimension of the feedforward network model in nn.TransformerEncoder*

'nlayers' :3, *# the number of nn.TransformerEncoderLayer in nn.TransformerEncoder*

'seq-len': config\_data['seq-len'], *# dont use wandb config*

'vocab-size':len(dm.vocab.stoi), *# the size of vocabulary /also called tokens*

'weight\_per\_class':ws,

"val-file":"val-out-rnn.txt",

"train-file":'train-out-rnn.txt',

"vocab": dm.vocab,

"batch-size":config\_data['batch-size']

}

**with** open (config\_model["val-file"],'w') **as** f:

f.write(">> Starting")

**with** open (config\_model["train-file"],'w') **as** f:

f.write(">> Starting")

model = AlarmGRU(config=config\_model)

model.initialize\_hidden()

early\_stop\_callback = EarlyStopping(

monitor='val\_epoch\_loss',

min\_delta=0,

patience=100,

verbose=**True**,

mode='min'

)

trainer = Trainer(auto\_lr\_find=0.0001, precision=16,gpus=-1, num\_nodes=1, max\_epochs=100, check\_val\_every\_n\_epoch=1,deterministic=**True**,gradient\_clip\_val=0.5,enable\_pl\_optimizer=**True**,callbacks=[early\_stop\_callback],progress\_bar\_refresh\_rate=0)

*# Run learning rate finder*

lr\_finder = trainer.tuner.lr\_find(model,dm)

*# Results can be found in*

lr\_finder.results

*# Plot with*

fig = lr\_finder.plot(suggest=**True**)

fig.show()

*# Pick point based on plot, or get suggestion*

new\_lr = lr\_finder.suggestion()

print(f"Suggested lr = **{**new\_lr**}**")

model.hparams.lr = new\_lr/100 *#7.5e-12 # can devide by 10*

trainer.fit(model,dm) *# Fit model*

*# trainer.test(datamodule=dm) # testing*

**Conclusion**

As we can see from the above outputs, in 100 epochs the Transformers models predict roughly more than 135 alarms sources with more than 50% accuracy while RNN model only able to predict 51 alrarm sources with more than 50% accuracy. Thus, the Transformers performs relative better as compared to RNNs on next alarm prediction task.