EE6132 Advanced Topics in Signal Processing - Assignment No 3

Name: Atishay Ganesh
Roll Number: EE17B155

October 12, 2019

1 Abstract

The goal of this assignment is the following.

- To experiment with Recurrent Neural Networks.
- To study Vanilla RNN's and LSTM's.
- To explore the ability of RNN's to tackle varied lengths of inputs and outputs.

2 Assignment

2.1 Part 0

Implementation Details: We use Pytorch in Python 3.6 for implementing the Recurrent Neural Networks. We use torch as the Deep Learning Framework, Tensorboard for plotting the graphs and torchvision for the MNIST dataset and the appropriate transforms.

Importing the standard libraries

```
import sys
import torch
import torch.nn as nn
from torchvision import datasets, transforms
import torch.nn.functional as F
import matplotlib.pyplot as plt
from torch.utils.tensorboard import SummaryWriter
from torch.utils.data import RandomSampler
import numpy as np
```

```
from torch.utils.data.sampler import SubsetRandomSampler
from functools import partial
from loadData import loadData
import argparse
```

We write code to load the MNIST Data and split it into train, validation and test sets. (50000-10000-10000 split of images) 0.1307 and 0.3081 are the mean and σ of MNIST respectively. We normalise the images to -1 to 1.

```
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307 ,), (0.3081 ,))
                       ])),batch_size= batch_size,

                       shuffle=False,sampler=train_sampler,**kwargs)
```

2.2 MNIST Classifiaction using RNN

In this question we try and use Vanilla RNN's and LSTM's to classify MNIST Data.
Forward Pass function:

```
class RNN(nn.Module):
    def __init__(self,device,bidir=False,lstm=False):
        super(RNN, self).__init__()
        self.hidden_dim =128
        self.input_dim = 28
        self.device = device
        self.bidir = bidir
        self.lstm = lstm
        self.num_dir = 2 if bidir else 1
        if lstm:
            self.rnn = nn.LSTM(input_size=self.input_dim,
                    hidden_size=self.hidden_dim,
                    batch_first=True,bidirectional=self.bidir)
        else:
            self.rnn = nn.RNN(input_size=self.input_dim,
                    hidden_size=self.hidden_dim,
                    batch_first=True,bidirectional=self.bidir)
        self.fc1 = nn.Linear(self.hidden_dim*self.num_dir,10)
        self.ct = 0
        print('num_dir',self.num_dir)
```

```
    def forward(self, x):
        batch_size = x.size(0)
        out, hidden = self.rnn(torch.squeeze(x))
        out = out[:,-1,:]
        self.ct +=1
        out = self.fc1(out)
        return F.log_softmax(out, dim=1)
```

We train the network using negative log likelihood loss and Adam optimizer with learning rate 1^{-3} and weight decay of 1^{-2}.

```
    def train_epoch(self,epoch):
        for batch_idx, (data, target) in enumerate(self.train_loader):
            data, target = data.to(self.device), target.to(self.device)
            self.optimizer.zero_grad()
            output = self.model(data)
            correct = 0
            loss = F.nll_loss(output, target)
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

            loss.backward()
            self.optimizer.step()
            self.writer.add_scalar('Loss/train', loss.item(),
            self.steps)
            self.writer.add_scalar('Accuracy/train',
            100*correct / len(target),self.steps)
            self.steps +=1

            if batch_idx % self.logging_interval == 0:
                print('Train Epoch: {} [{}/{} ({:.0f})%]\tLoss: {:.6f}\n'.format(
                    epoch+1, batch_idx * len(data), self.size[0],
                    100.*batch_idx*len(data) /self.size[0], loss.item()),
                    self.steps)
                self.test_epoch('valid')
                self.model.train()


def test_epoch(self,type_set):
    def test_epoch(self,type_set):
        self.model.eval()
        curr_loader = self.valid_loader if type_set == 'valid'\\
            else self.test_loader
        size_value = 1 if type_set =='valid' else 2
        correct = 0
```

```
loss = 0
with torch.no_grad():
    for data, target in curr_loader:
        data, target = data.to(self.device), target.to(self.device)
        output = self.model(data)
        loss += F.nll_loss(output, target, reduction='sum').item()
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()
loss /= self.size[size_value]
if type_set== 'valid':

    self.writer.add_scalar('Loss/valid',
             loss,self.steps)
    self.writer.add_scalar('Accuracy/valid',
              100.*correct / self.size[size_value],self.steps)

print('{} set: Average loss: {:.4f},\\
    Accuracy: {}/{} ({:.0f}%)\n'.format(
    type_set,loss, correct, self.size[size_value],
    100.*correct/self.size[size_value]),self.steps)
if type_set !='valid':
    with open('log.txt','a') as fp:
        fp.write(f'{self.tag_str}')
        fp.write(
            f'Test Set Accuracy:{100.*correct/self.size[size_value]}%')
```

We train all 4 combinations (i.e LSTM, Bidirectional LSTM, RNN, Bidirectional RNN).
The results of the same are shown.

LSTM Test Set Accuracy: 95.72%
Bidirectional LSTM Test Set Accuracy: 95.64%
RNN Test Set Accuracy: 95.6%
Bidirectional RNN Test Set Accuracy: 95.53%

From the plots, the RNN seemed to converge slightly slower than the LSTM. The LSTM also seemed to do slightly better. The bidirectional variants took a little longer to train than their unidirectional counterparts but their performance was not significantly different.

2.3 Remembering the number at a particular index

We explore the question of remembering the value of a number at a particular index (2^{nd} Index, in this case), and try and solve it using an LSTM.

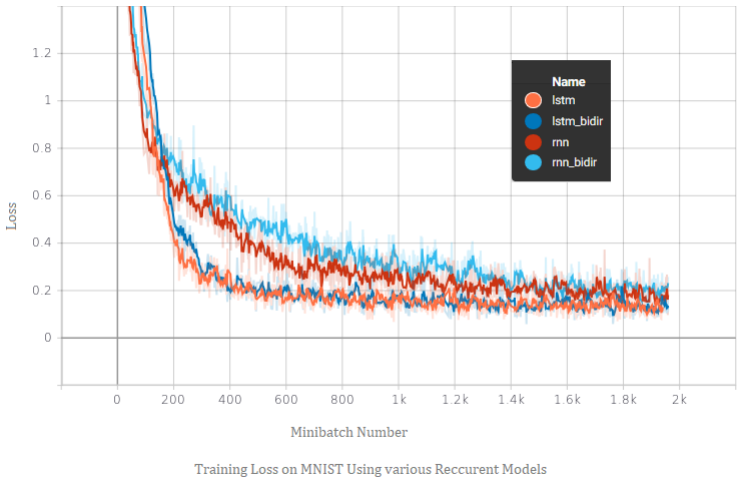

Figure 1: Training Loss

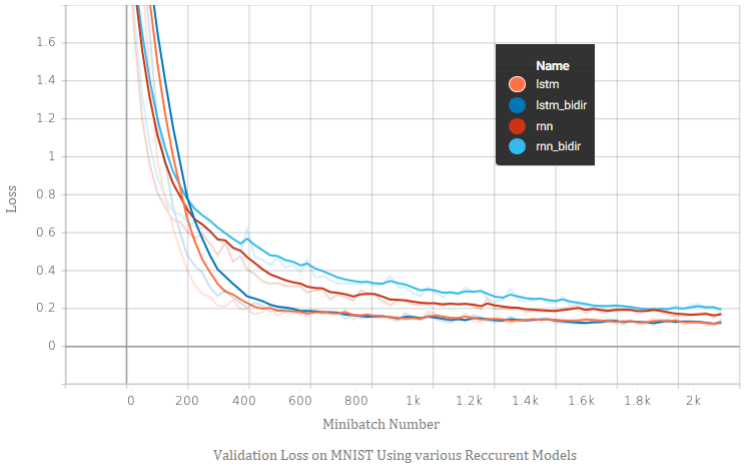

Figure 2: Validation Loss

We vary the memory size between 2, 5 and 10. We use an Adam optimizer with learning rate of 5^{-3} and Weight Decay of 1^{-4}

Generating the training data is as shown.

```
def sample(len,number=1):
    inp_o = np.random.randint(0,10,(number,len))
    n_values = 10
    inp = np.eye(n_values)[inp_o]
    target = inp_o[:,1]
```

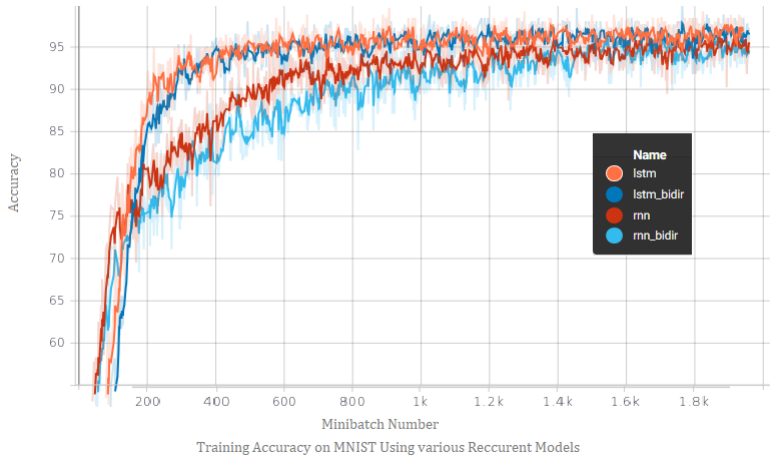

Figure 3: Training Accuracy

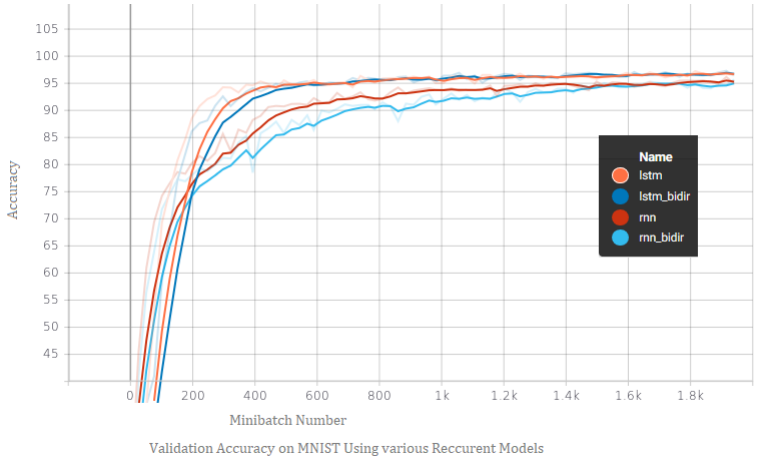

Figure 4: Validation Accuracy

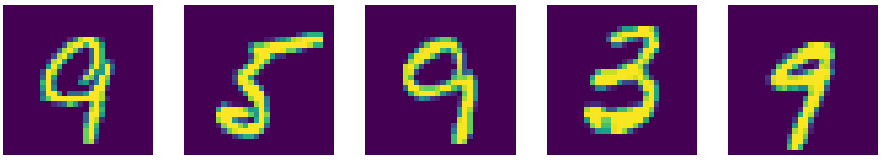

Figure 5: LSTM: True Labels: [9, 5, 9, 3, 9] Predicted Labels:[9, 5, 9, 3, 9] respectively

```
        return inp,target
```

The RNN is very similar to the first case.

```
class RNN(nn.Module):
```

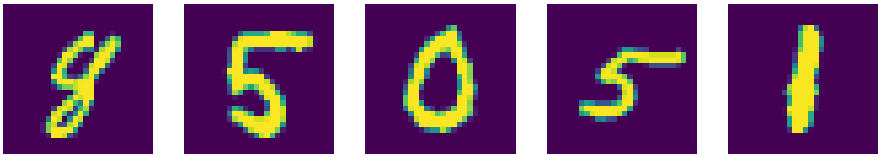

Figure 6: Bidirectional LSTM: True Labels: [8,5,0,5,1] Predicted Labels:[8,5,0,5,1] respectively

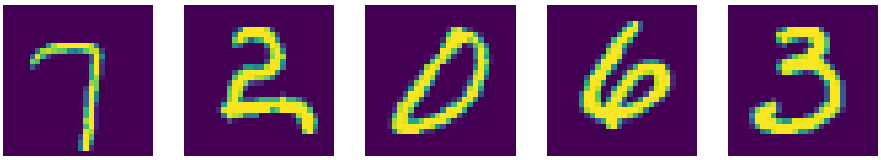

Figure 7: RNN: True Labels: [7,2,0,6,3]Predicted Labels:[7,2,0,6,3] respectively

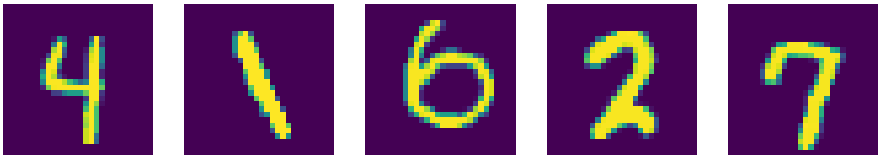

Figure 8: Bidirectional RNN: True Labels: [4,1,6,2,7] Predicted Labels:[4 1 6 2 7] respectively

```
def __init__(self,latent_dim):
    super(RNN, self).__init__()
    self.hidden_dim = latent_dim
    self.input_dim = 10
    self.num_dir = 1
    self.rnn = nn.LSTM(input_size=self.input_dim,hidden_size=self.hidden_dim,
                batch_first=True,bidirectional=False)
    self.fc1 = nn.Linear(self.hidden_dim*self.num_dir,10)
    print('num_dir',self.num_dir)

def forward(self, x):
    batch_size = x.size(0)
    out, hidden = self.rnn(x)
    out = out[:,-1,:]
    out = self.fc1(out)
    return F.log_softmax(out, dim=1)
```

The code for training the network is almost identical, so is omitted for brevity.

The results are as follows.
The Training of the models with memory 5 and 10 were stopped early as they had converged to an accuracy of 100% and hence no room for improvement was there.

Memory Size 2: Acc 25.5%
Memory Size 5: Acc 100%
Memory Size 10: Acc 100%

Using the model with memory size 5 (Since both sizes 5 and 10 have 100% accuracy:

Input Sequence [5 8 6 1 5 4 1]
Prediction for second element: 8
Actual second element: 8

Input Sequence [7 4 5 9 4 3 2 3 3 3]
Prediction for second element: 4
Actual second element: 4

Input Sequence [8 0 1 5 2 4 7 4 9]
Prediction for second element: 0
Actual second element: 0

Input Sequence [7 3 9 9 3 6 1 4 2]
Prediction for second element: 3
Actual second element: 3

Input Sequence [4 1 9 2 3 5]
Prediction for second element: 1
Actual second element: 1

2.4 Addition

We try to use an RNN to learn addition of two binary strings. The idea is the RNN should be able to learn the concept of a carry string and should also be able to scale up. (i.e a model trained on strings of length 5 should be able to work on much longer strings, say of length 20)
We run various types of experiments to understand this better.

- We vary the model size between 3, 5, 10

Figure 9: Training Loss

Figure 10: Prediction Accuracy

- We allow the network to train on fixed input sizes - 3,5, and 10. For comparison, we also show the normal plot.
- We compare Cross-entropy Loss and MSE Loss.

Loading the data is as shown.

```
def sample(types,args,lu=None):
    if types==1:
        lower = 2
        upper = 2**20
    else:
        lower =2**args.fix
        upper =2**(args.fix+1)
    if lu:
        lower = 2**lu
        upper = 2**(lu+1)

    i1 = int(np.random.randint(lower,upper,1))
    i2 = int(np.random.randint(lower,upper,1))
    inp_1 = [int(x) for x in bin(i1)[2:]]
    inp_2 = [int(x) for x in bin(i2)[2:]]
```

```
    o_p = [int(x) for x in bin(i2+i1)[2:]][::-1]
    l = len(o_p)
    inp_1 = np.concatenate((np.array(inp_1[::-1]),np.zeros(l-len(inp_1))))
    inp_2 = np.concatenate((np.array(inp_2[::-1]),np.zeros(l-len(inp_2))))
    inp = np.stack((inp_1,inp_2),axis=-1)
    op = np.expand_dims(np.array(o_p),-1)
    return inp,op

def loadData(type_data):
    train = [],[]
    device = torch.device("cpu")
    if args.fix:
        types = 2,1
    else:
        types = 1,1
    for i in range(2000):
        a = sample(types[0],args)
        train[0].append(a[0])
        train[1].append(a[1])
    for j in range(2,21):
        for i in range(200):
            a = sample(types[1],args,lu=j)
            train[0].append(a[0])
            train[1].append(a[1])

    tensor_trx = ([torch.Tensor([i]) for i in train[0]])
    tensor_try = ([torch.Tensor([i]) for i in train[1]])
    print(tensor_trx[0].shape)
    print(tensor_try[0].shape)
    return tensor_trx[0:2000],tensor_try[0:2000],
            tensor_trx[2000:],tensor_try[2000:],device,(2000,200)
```

The RNN used for addition is as shown

```
class RNN(nn.Module):
    def __init__(self,args):
        super(RNN, self).__init__()
        self.hidden_dim = args.memory
        self.input_dim = 2
        self.num_dir = 1
        self.rnn = nn.LSTM(input_size=self.input_dim,hidden_size=self.hidden_dim,
                    batch_first=True,bidirectional=False)
        self.fc1 = nn.Linear(self.hidden_dim*self.num_dir,1)
        print('num_dir',self.num_dir)
```

```
    def forward(self, x):
        out, hidden = self.rnn(x)
        out = self.fc1(out)
        return torch.sigmoid(out)
```

The training of the RNN is almost identical to the previous cases, so is omitted for brevity.

2.4.1 Results and Analysis

The Results are as follows.
Average bit accuracies for the various Models are as shown.
MSE Loss:

Variable Length Input:
Memory Size 3: 100% accuracy
Memory Size 5: 100% accuracy
Memory Size 10: 100% accuracy

Fixed Length Input and tested on variable lengths, memory Size 5:
Input Length 3: 99.87% accuracy
Input Length 5: 99.96% accuracy
Input Length 10: 100% accuracy

Cross-entropy Loss with memory size 5: 100% accuracy

For the case with Fixed input length, we also plot the bit accuracy vs length, since it is the only case with non perfect outputs.

It is observed that the accuracy is almost 100% in all cases. In cases with variable length input for training, the final accuracy is always 100%. The larger models converge faster in this case though.
On the other hand, if we have fixed length input the model takes longer to train (especially for the case with Input size 3). The accuracy is also not always perfectly 1, and the difference of the loss from 1 increases for greater input sizes.

For the case of Comparing Crossentropy Loss with MSE Loss, we see that they both converge to the same final accuracy of 100% and loss, although the cross entropy case takes slightly longer to converge.

2.4.2 Example

Example of an input and output pair for addition:

Input Sequence (bits paired):

$[0,0],[0,0],[0,0],[1,1],[0,1],[1,0],[1,0],[1,0],[1,0],[0,1],[1,1]$
$,[0,1],[0,0],[0,1],[1,1],[0,0],[1,1],[1,0],[1,0],[1,1],[0,0]$

Prediction (bitwise):

$[0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,0,1,0,0,0,1,1]$

Actual (bitwise):

$[0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,0,1,0,0,0,1,1]$

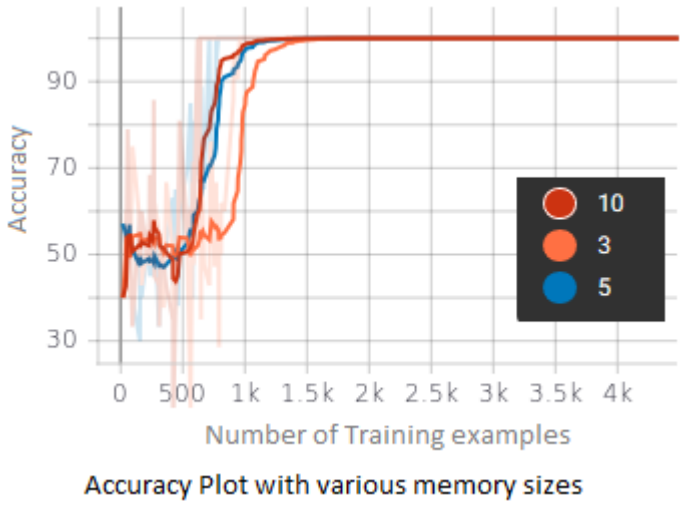

Figure 11: Accuracy while varying Model Size

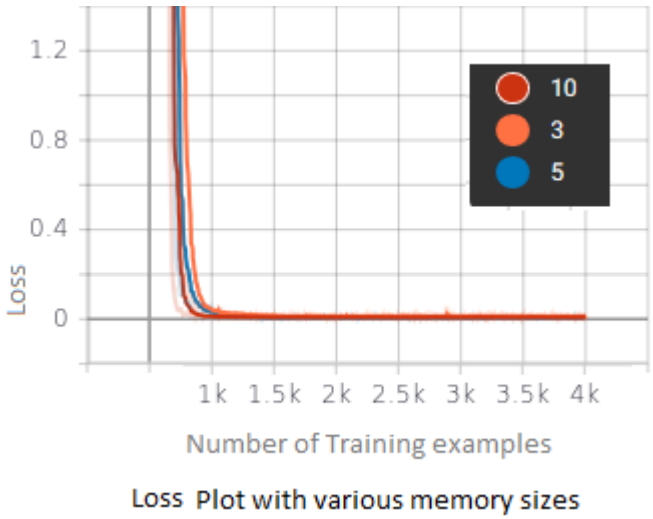

Figure 12: Training Loss while Varying Model Size

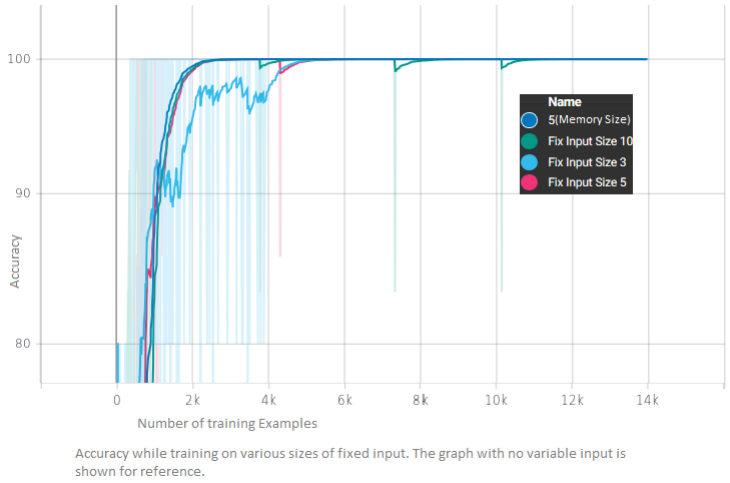

Figure 13: Accuracy while varying size of fixed input

3 Conclusions

- We experimented with Recurrent Neural Networks.
- We compared LSTM's vs Vanilla RNN's
- We studied the capability of RNN's in handling variable length inputs.
- We solved MNIST using RNN's, and also used an RNN to solve addition and memorisation problems.

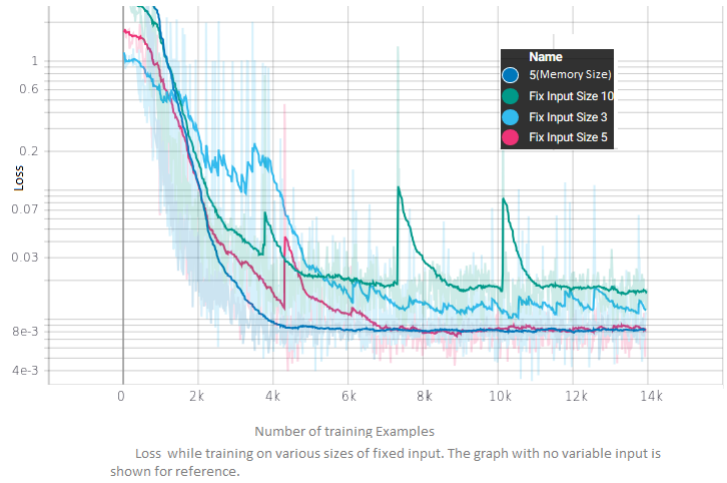

Figure 14: Training Loss while varying size of fixed input

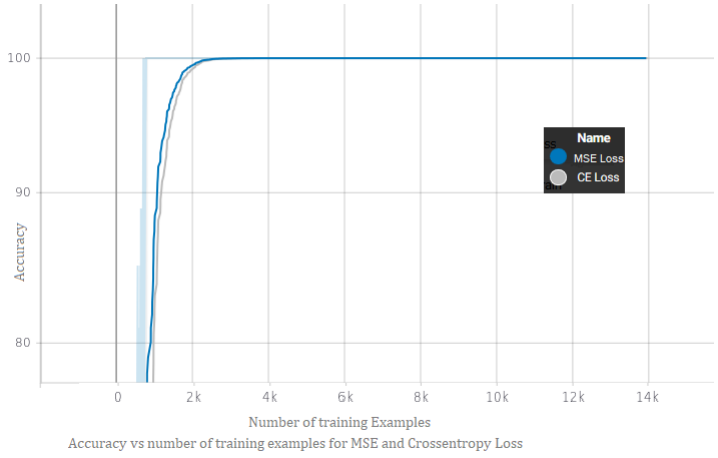

Figure 15: Accuracy while comparing Different Loss types

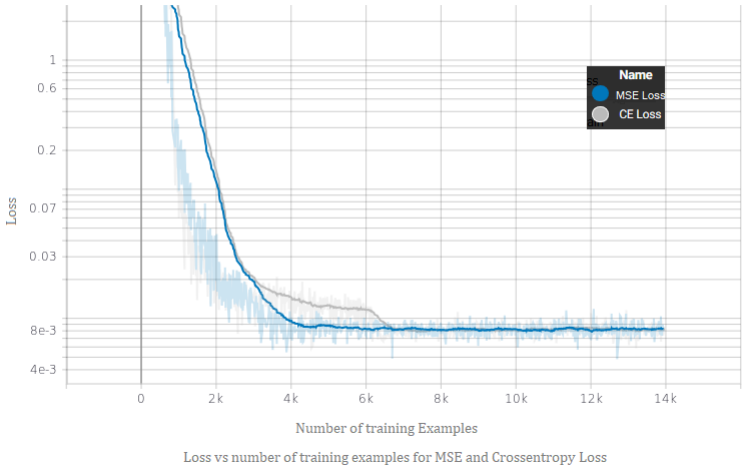

Figure 16: Training Loss while comparing different loss types

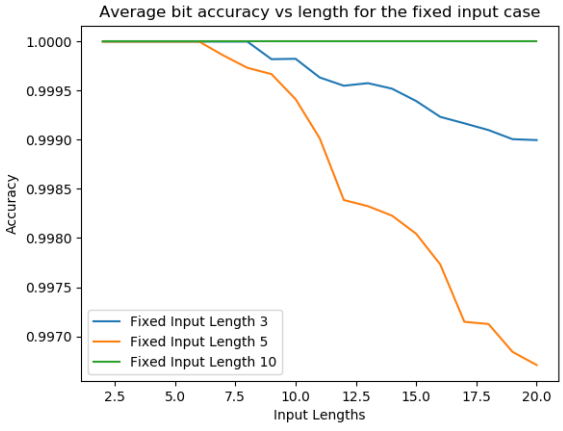

Figure 17: Accuracy vs Length for the fixed length input case.