Coursework 3

Tianyu Dai

March 9, 2021

1 Coursework 3: RNNs

Instructions Please submit on CATe a zip file named *CW3_RNNs.zip* containing a version of this notebook with your answers. Write your answers in the cells below for each question.

1.1 Recurrent models coursework

This coursework is separated into a coding and a theory component.

For the first part, you will use the Google Speech Commands v0.02 subset that you used in the RNN tutorial: http://www.doc.ic.ac.uk/~pam213/co460_files/

1.1.1 Part 1 - Coding

In this part you will have to:

- Implement an LSTM
- Implement a GRU

1.1.2 Part 2 - Theory

Here you will answer some theoretical questions about RNNs – no detailed proofs and no programming.

1.1.3 Part 1: Coding

1.1.4 Dataset

We will be using the Google *Speech Commands* v0.02 [1] dataset.

[1] Warden, P. (2018). Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*.

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[2]:
```
## MAKE SURE THIS POINTS INSIDE THE DATASET FOLDER.
dataset_folder = "/content/drive/MyDrive/data/" # this should change depending␣
 ↪on where you have stored the data files
```

1.1.5 Initial code before coursework questions start:

[3]:
```
import math
import os
import random
from collections import defaultdict

import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.utils.data import Dataset
import numpy as np
from scipy.io.wavfile import read
import librosa
from matplotlib import pyplot as plt

cuda = True if torch.cuda.is_available() else False

Tensor = torch.cuda.FloatTensor if cuda else torch.FloatTensor
```

[4]:
```
def set_seed(seed_value):
    """Set seed for reproducibility.
    """
    random.seed(seed_value)
    np.random.seed(seed_value)
    torch.manual_seed(seed_value)
    torch.cuda.manual_seed_all(seed_value)

set_seed(42)
```

[5]:
```
class SpeechCommandsDataset(Dataset):
    """Google Speech Commands dataset."""

    def __init__(self, root_dir, split):
        """
        Args:
            root_dir (string): Directory with all the data files.
            split    (string): In ["train", "valid", "test"].
        """
        self.root_dir = root_dir
        self.split = split

        self.number_of_classes = len(self.get_classes())

        self.class_to_file = defaultdict(list)

        self.valid_filenames = self.get_valid_filenames()
```

```
        self.test_filenames = self.get_test_filenames()

        for c in self.get_classes():
            file_name_list = sorted(os.listdir(self.root_dir +␣
↪"data_speech_commands_v0.02/" + c))
            for filename in file_name_list:
                if split == "train":
                    if (filename not in self.valid_filenames[c]) and (filename␣
↪not in self.test_filenames[c]):
                        self.class_to_file[c].append(filename)
                elif split == "valid":
                    if filename in self.valid_filenames[c]:
                        self.class_to_file[c].append(filename)
                elif split == "test":
                    if filename in self.test_filenames[c]:
                        self.class_to_file[c].append(filename)
                else:
                    raise ValueError("Invalid split name.")

        self.filepath_list = list()
        self.label_list = list()
        for cc, c in enumerate(self.get_classes()):
            f_extension = sorted(list(self.class_to_file[c]))
            l_extension = [cc for i in f_extension]
            f_extension = [self.root_dir + "data_speech_commands_v0.02/" + c +␣
↪"/" + filename for filename in f_extension]
            self.filepath_list.extend(f_extension)
            self.label_list.extend(l_extension)
        self.number_of_samples = len(self.filepath_list)

    def __len__(self):
        return self.number_of_samples

    def __getitem__(self, idx):
        sample = np.zeros((16000, ), dtype=np.float32)

        sample_file = self.filepath_list[idx]

        sample_from_file = read(sample_file)[1]
        sample[:sample_from_file.size] = sample_from_file
        sample = sample.reshape((16000, ))

        sample = librosa.feature.mfcc(y=sample, sr=16000, hop_length=512,␣
↪n_fft=2048).transpose().astype(np.float32)

        label = self.label_list[idx]
```

```
        return sample, label

    def get_classes(self):
        return ['one', 'two', 'three']

    def get_valid_filenames(self):
        class_names = self.get_classes()

        class_to_filename = defaultdict(set)
        with open(self.root_dir + "data_speech_commands_v0.02/validation_list.
↪txt", "r") as fp:
            for line in fp:
                clean_line = line.strip().split("/")

                if clean_line[0] in class_names:
                    class_to_filename[clean_line[0]].add(clean_line[1])

        return class_to_filename

    def get_test_filenames(self):
        class_names = self.get_classes()

        class_to_filename = defaultdict(set)
        with open(self.root_dir + "data_speech_commands_v0.02/testing_list.
↪txt", "r") as fp:
            for line in fp:
                clean_line = line.strip().split("/")

                if clean_line[0] in class_names:
                    class_to_filename[clean_line[0]].add(clean_line[1])

        return class_to_filename
```

[6]:
```
train_dataset = SpeechCommandsDataset(dataset_folder,
                                      "train")
valid_dataset = SpeechCommandsDataset(dataset_folder,
                                      "valid")

test_dataset = SpeechCommandsDataset(dataset_folder,
                                     "test")

batch_size = 100


num_epochs = 5
valid_every_n_steps = 20
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
```

```
                                           batch_size=batch_size,
                                           shuffle=True)
valid_loader = torch.utils.data.DataLoader(dataset=valid_dataset,
                                           batch_size=batch_size,
                                           shuffle=False)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

1.1.6 Question 1: Finalise the LSTM and GRU cells by completing the missing code

You are allowed to use nn.Linear.

[7]:
```
class LSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size, bias=True):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.bias = bias

        ######################################################################
        ## START OF YOUR CODE - Question 1a) Complete the missing code
        ######################################################################

        self.w_c_x = nn.Linear(input_size, hidden_size, bias = bias)
        self.w_c_h = nn.Linear(hidden_size, hidden_size, bias = bias)

        self.w_i_x = nn.Linear(input_size, hidden_size, bias = bias)
        self.w_i_h = nn.Linear(hidden_size, hidden_size, bias = bias)

        self.w_f_x = nn.Linear(input_size, hidden_size, bias = bias)
        self.w_f_h = nn.Linear(hidden_size, hidden_size, bias = bias)

        self.w_o_x = nn.Linear(input_size, hidden_size, bias = bias)
        self.w_o_h = nn.Linear(hidden_size, hidden_size, bias = bias)

        ######################################################################
        ## END OF YOUR CODE
        ######################################################################
        self.reset_parameters()

    def reset_parameters(self):
        std = 1.0 / math.sqrt(self.hidden_size)
        for w in self.parameters():
            w.data.uniform_(-std, std)
```

```
    def forward(self, input, hx=None):
        if hx is None:
            hx = input.new_zeros(input.size(0), self.hidden_size,␣
↪requires_grad=False)
            hx = (hx, hx)

        # We used hx to pack both the hidden and cell states
        hx, cx = hx

        ##############################################################################
        ## START OF YOUR CODE - Question 1b) Complete the missing code
        ##############################################################################
        c_t = torch.tanh(self.w_c_x(input) + self.w_c_h(hx))
        i_t = torch.sigmoid(self.w_i_x(input) + self.w_i_h(hx))
        f_t = torch.sigmoid(self.w_f_x(input) + self.w_f_h(hx))
        o_t = torch.sigmoid(self.w_o_x(input) + self.w_o_h(hx))
        cy = f_t * cx + i_t * c_t
        hy = o_t * torch.tanh(cy)

        ##############################################################################
        ## END OF YOUR CODE
        ##############################################################################

        return (hy, cy)

class BasicRNNCell(nn.Module):
    def __init__(self, input_size, hidden_size, bias=True, nonlinearity="tanh"):
        super(BasicRNNCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.bias = bias
        self.nonlinearity = nonlinearity
        if self.nonlinearity not in ["tanh", "relu"]:
            raise ValueError("Invalid nonlinearity selected for RNN.")

        self.x2h = nn.Linear(input_size, hidden_size, bias=bias)
        self.h2h = nn.Linear(hidden_size, hidden_size, bias=bias)

        self.reset_parameters()


    def reset_parameters(self):
        std = 1.0 / math.sqrt(self.hidden_size)
        for w in self.parameters():
            w.data.uniform_(-std, std)
```

```
    def forward(self, input, hx=None):
        if hx is None:
            hx = input.new_zeros(input.size(0), self.hidden_size,␣
↪requires_grad=False)

        activation = getattr(nn.functional, self.nonlinearity)
        hy = activation(self.x2h(input) + self.h2h(hx))

        return hy



class GRUCell(nn.Module):
    def __init__(self, input_size, hidden_size, bias=True):
        super(GRUCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.bias = bias

        ######################################################################
        ## START OF YOUR CODE - Question 1c) Complete the missing code
        ######################################################################
        self.w_z_x = nn.Linear(input_size, hidden_size, bias = bias)
        self.w_z_h = nn.Linear(hidden_size, hidden_size, bias = bias)

        self.w_r_x = nn.Linear(input_size, hidden_size, bias = bias)
        self.w_r_h = nn.Linear(hidden_size, hidden_size, bias = bias)

        self.w_n_x = nn.Linear(input_size, hidden_size, bias = bias)
        self.w_n_h = nn.Linear(hidden_size, hidden_size, bias = bias)
        ######################################################################
        ## END OF YOUR CODE
        ######################################################################
        self.reset_parameters()


    def reset_parameters(self):
        std = 1.0 / math.sqrt(self.hidden_size)
        for w in self.parameters():
            w.data.uniform_(-std, std)

    def forward(self, input, hx=None):
        if hx is None:
            hx = input.new_zeros(input.size(0), self.hidden_size,␣
↪requires_grad=False)

        ######################################################################
```

```
        ## START OF YOUR CODE - Question 1d) Complete the missing code
        ################################################################################
        z_t = torch.sigmoid(self.w_z_x(input) + self.w_z_h(hx))
        r_t = torch.sigmoid(self.w_r_x(input) + self.w_r_h(hx))
        n_t = torch.tanh(self.w_n_x(input) + r_t * self.w_n_h(hx))
        hy = torch.mul((1 - z_t), n_t) + torch.mul(z_t, hx)
        ################################################################################
        ## END OF YOUR CODE
        ################################################################################

        return hy
```

1.1.7 Question 2: Finalise the RNNModel and BidirRecurrentModel

[8]:
```
class RNNModel(nn.Module):
    def __init__(self, mode, input_size, hidden_size, num_layers, bias,␣
↪output_size):
        super(RNNModel, self).__init__()
        self.mode = mode
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bias = bias
        self.output_size = output_size

        self.rnn_cell_list = nn.ModuleList()

        if mode == 'LSTM':
        ################################################################################
        ## START OF YOUR CODE - Question 2a) Complete the missing code
        #
        #  Append the appropriate LSTM cells to rnn_cell_list
        ################################################################################
            self.rnn_cell_list.append(LSTMCell(self.input_size, self.
↪hidden_size, self.bias))
            for l in range(1, self.num_layers):
                self.rnn_cell_list.append(LSTMCell(self.hidden_size, self.
↪hidden_size, self.bias))

        ################################################################################
        ## END OF YOUR CODE
       ␣
↪################################################################################

        elif mode == 'GRU':

        ################################################################################
```

```
        ## START OF YOUR CODE - Question 2b) Complete the missing code
        #
        #  Append the appropriate GRU cells to rnn_cell_list
        ################################################################################
            self.rnn_cell_list.append(GRUCell(self.input_size, self.
↪hidden_size, self.bias))
            for l in range(1, self.num_layers):
                self.rnn_cell_list.append(GRUCell(self.hidden_size, self.
↪hidden_size, self.bias))

        ################################################################################
        ## END OF YOUR CODE
        ␣
↪################################################################################    ␣
↪

        elif mode == 'RNN_TANH':


        ################################################################################
        ## START OF YOUR CODE - Question 2c) Complete the missing code
        #
        #  Append the appropriate RNN cells to rnn_cell_list
        ################################################################################
            self.rnn_cell_list.append(BasicRNNCell(self.input_size, self.
↪hidden_size, self.bias, "tanh"))
            for l in range(1, self.num_layers):
                self.rnn_cell_list.append(BasicRNNCell(self.hidden_size, self.
↪hidden_size, self.bias, "tanh"))

        ################################################################################
        ## END OF YOUR CODE
        ################################################################################



        elif mode == 'RNN_RELU':

        ################################################################################
        ## START OF YOUR CODE - Question 2d) Complete the missing code
        #
        #  Append the appropriate RNN cells to rnn_cell_list
        ################################################################################
            self.rnn_cell_list.append(BasicRNNCell(self.input_size, self.
↪hidden_size, self.bias, "relu"))
            for l in range(1, self.num_layers):
```

```
                self.rnn_cell_list.append(BasicRNNCell(self.hidden_size, self.
↪hidden_size, self.bias, "relu"))

        ##############################################################################
        ## END OF YOUR CODE
        ##############################################################################


        else:
            raise ValueError("Invalid RNN mode selected.")


        self.att_fc = nn.Linear(self.hidden_size, 1)
        self.fc = nn.Linear(self.hidden_size, self.output_size)


    def forward(self, input, hx=None):

        outs = []
        h0 = [None] * self.num_layers if hx is None else list(hx)

        # In this forward pass we want to create our RNN from the rnn cells,
        # ..taking the hidden states from the final RNN layer and passing these
        # ..through our fully connected layer (fc).

        # The multi-layered RNN should be able to run when the mode is either
        # .. LSTM, GRU, RNN_TANH or RNN_RELU.

        ##############################################################################
        ## START OF YOUR CODE - Question 2e) Complete the missing code
        #
        #  HINT: You may need a special case for LSTMs
        ##############################################################################

        for seq in range(input.size(1)):
          x = input[:, seq, :]
          for l, cell in enumerate(self.rnn_cell_list):
            h0[l] = cell(x, h0[l])
            if self.mode == 'LSTM':
              x, _ = h0[l]
            else:
              x = h0[l]
          outs.append(x)

        ##############################################################################
        ## END OF YOUR CODE
        ##############################################################################
```

```
        out = outs[-1].squeeze()

        out = self.fc(out)


        return out


class BidirRecurrentModel(nn.Module):
    def __init__(self, mode, input_size, hidden_size, num_layers, bias,␣
 ↪output_size):
        super(BidirRecurrentModel, self).__init__()
        self.mode = mode
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.bias = bias
        self.output_size = output_size

        self.rnn_cell_list = nn.ModuleList()
        self.rnn_cell_list_rev = nn.ModuleList()

        ########################################################################
        ## START OF YOUR CODE - Question 2f) Complete the missing code
        #
        #  Create code for the following 'mode' values:
        # 'LSTM', 'GRU', 'RNN_TANH' and 'RNN_RELU'
        ########################################################################
        if mode == 'LSTM':
            self.rnn_cell_list.append(LSTMCell(self.input_size, self.
 ↪hidden_size, self.bias))
            for l in range(1, self.num_layers):
              self.rnn_cell_list.append(LSTMCell(self.hidden_size, self.
 ↪hidden_size, self.bias))
            self.rnn_cell_list_rev.append(LSTMCell(self.input_size, self.
 ↪hidden_size, self.bias))
            for l in range(1, self.num_layers):
              self.rnn_cell_list_rev.append(LSTMCell(self.hidden_size, self.
 ↪hidden_size, self.bias))

        elif mode == 'GRU':
            self.rnn_cell_list.append(GRUCell(self.input_size, self.
 ↪hidden_size, self.bias))
            for l in range(1, self.num_layers):
```

```
                self.rnn_cell_list.append(GRUCell(self.hidden_size, self.
↪hidden_size, self.bias))
            self.rnn_cell_list_rev.append(GRUCell(self.input_size, self.
↪hidden_size, self.bias))
            for l in range(1, self.num_layers):
                self.rnn_cell_list_rev.append(GRUCell(self.hidden_size, self.
↪hidden_size, self.bias))

        elif mode == 'RNN_TANH':
            self.rnn_cell_list.append(BasicRNNCell(self.input_size, self.
↪hidden_size, self.bias, "tanh"))
            for l in range(1, self.num_layers):
                self.rnn_cell_list.append(BasicRNNCell(self.hidden_size, self.
↪hidden_size, self.bias, "tanh"))
            self.rnn_cell_list_rev.append(BasicRNNCell(self.input_size, self.
↪hidden_size, self.bias, "tanh"))
            for l in range(1, self.num_layers):
                self.rnn_cell_list_rev.append(BasicRNNCell(self.hidden_size,␣
↪self.hidden_size, self.bias, "tanh"))

        elif mode == 'RNN_RELU':
            self.rnn_cell_list.append(BasicRNNCell(self.input_size, self.
↪hidden_size, self.bias, "relu"))
            for l in range(1, self.num_layers):
                self.rnn_cell_list.append(BasicRNNCell(self.hidden_size, self.
↪hidden_size, self.bias, "relu"))
            self.rnn_cell_list_rev.append(BasicRNNCell(self.input_size, self.
↪hidden_size, self.bias, "relu"))
            for l in range(1, self.num_layers):
                self.rnn_cell_list_rev.append(BasicRNNCell(self.hidden_size,␣
↪self.hidden_size, self.bias, "relu"))
        else:
            raise ValueError("Invalid RNN mode selected.")

        # x2 hidden-to-output connections
        self.fc = nn.Linear(self.hidden_size*2, self.output_size)

        ################################################################################
        ## END OF YOUR CODE
        ################################################################################


    def forward(self, input, hx=None):

        # In this forward pass we want to create our Bidirectional RNN from the␣
↪rnn cells,
```

```
        # .. taking the hidden states from the final RNN layer with their␣
↪reversed counterparts
        # .. before concatening these and running them through the fully␣
↪connected layer (fc)

        # The multi-layered RNN should be able to run when the mode is either
        # .. LSTM, GRU, RNN_TANH or RNN_RELU.

        ##########################################################################
        ## START OF YOUR CODE  - Question 2g) Complete the missing code
        ##########################################################################
        outs = []
        outs_rev = []
        h0 = [None] * self.num_layers if hx is None else list(hx)
        h0_rev = [None] * self.num_layers if hx is None else list(hx)

        for seq in range(input.size(1)):
          x = input[:, seq, :]
          x_rev = input[:, -seq-1, :]
          for l, cell in enumerate(self.rnn_cell_list):
            h0[l] = cell(x, h0[l])
            if self.mode == 'LSTM':
              x, _ = h0[l]
            else:
              x = h0[l]
          for l, cell_rev in enumerate(self.rnn_cell_list_rev):
            h0_rev[l] = cell_rev(x_rev, h0_rev[l])
            if self.mode == 'LSTM':
              x_rev, _ = h0_rev[l]
            else:
              x_rev = h0_rev[l]
          outs.append(x)
          outs_rev.append(x_rev)
        ##########################################################################
        ## END OF YOUR CODE
        ##########################################################################

        out = outs[-1].squeeze()
        out_rev = outs_rev[0].squeeze()
        out = torch.cat((out, out_rev), 1)

        out = self.fc(out)
        return out
```

The code below trains a network based on your code above. This should work without error:

[9]:
```
seq_dim, input_dim = train_dataset[0][0].shape
output_dim = 3

hidden_dim = 128
layer_dim = 4
bias = True

### Change the code below to try running different models:
# model = RNNModel("LSTM", input_dim, hidden_dim, layer_dim, bias, output_dim)
model = BidirRecurrentModel("LSTM", input_dim, hidden_dim, layer_dim, bias,␣
 ↪output_dim)

if torch.cuda.is_available():
    model.cuda()

criterion = nn.CrossEntropyLoss()

learning_rate = 0.001
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

loss_list = []
iter = 0
max_v_accuracy = 0
reported_t_accuracy = 0
max_t_accuracy = 0
for epoch in range(num_epochs):
    for i, (audio, labels) in enumerate(train_loader):
        if torch.cuda.is_available():
            audio = Variable(audio.view(-1, seq_dim, input_dim).cuda())
            labels = Variable(labels.cuda())
        else:
            audio = Variable(audio.view(-1, seq_dim, input_dim))
            labels = Variable(labels)

        optimizer.zero_grad()

        outputs = model(audio)

        loss = criterion(outputs, labels)

        if torch.cuda.is_available():
            loss.cuda()

        loss.backward()

        optimizer.step()
```

```
        loss_list.append(loss.item())
        iter += 1

        if iter % valid_every_n_steps == 0:
            correct = 0
            total = 0
            for audio, labels in valid_loader:
                if torch.cuda.is_available():
                    audio = Variable(audio.view(-1, seq_dim, input_dim).cuda())
                else:
                    audio = Variable(audio.view(-1, seq_dim, input_dim))

                outputs = model(audio)

                _, predicted = torch.max(outputs.data, 1)

                total += labels.size(0)

                if torch.cuda.is_available():
                    correct += (predicted.cpu() == labels.cpu()).sum()
                else:
                    correct += (predicted == labels).sum()

            v_accuracy = 100 * correct // total

            is_best = False
            if v_accuracy >= max_v_accuracy:
                max_v_accuracy = v_accuracy
                is_best = True

            if is_best:
                for audio, labels in test_loader:
                    if torch.cuda.is_available():
                        audio = Variable(audio.view(-1, seq_dim, input_dim).
↪cuda())
                    else:
                        audio = Variable(audio.view(-1, seq_dim, input_dim))

                    outputs = model(audio)

                    _, predicted = torch.max(outputs.data, 1)

                    total += labels.size(0)

                    if torch.cuda.is_available():
                        correct += (predicted.cpu() == labels.cpu()).sum()
                    else:
```

```
                    correct += (predicted == labels).sum()

            t_accuracy = 100 * correct // total
            reported_t_accuracy = t_accuracy

        print('Iteration: {}. Loss: {}. V-Accuracy: {}  T-Accuracy: {}'.
↪format(iter, loss.item(), v_accuracy, reported_t_accuracy))
```

```
Iteration: 20. Loss: 0.9298540353775024. V-Accuracy: 58  T-Accuracy: 57
Iteration: 40. Loss: 0.728527307510376. V-Accuracy: 68  T-Accuracy: 69
Iteration: 60. Loss: 0.5860320925712585. V-Accuracy: 73  T-Accuracy: 73
Iteration: 80. Loss: 0.4155048727989197. V-Accuracy: 81  T-Accuracy: 82
Iteration: 100. Loss: 0.4415952265262604. V-Accuracy: 81  T-Accuracy: 80
Iteration: 120. Loss: 0.22947010397911072. V-Accuracy: 84  T-Accuracy: 85
Iteration: 140. Loss: 0.323339581489563. V-Accuracy: 88  T-Accuracy: 88
Iteration: 160. Loss: 0.38627883791923523. V-Accuracy: 90  T-Accuracy: 91
Iteration: 180. Loss: 0.1758715659379959. V-Accuracy: 90  T-Accuracy: 91
Iteration: 200. Loss: 0.31855612993240356. V-Accuracy: 92  T-Accuracy: 92
Iteration: 220. Loss: 0.24535949528217316. V-Accuracy: 92  T-Accuracy: 93
Iteration: 240. Loss: 0.23214882612228394. V-Accuracy: 94  T-Accuracy: 93
Iteration: 260. Loss: 0.3989745080471039. V-Accuracy: 93  T-Accuracy: 93
Iteration: 280. Loss: 0.11613195389509201. V-Accuracy: 94  T-Accuracy: 94
Iteration: 300. Loss: 0.14072883129119873. V-Accuracy: 93  T-Accuracy: 94
Iteration: 320. Loss: 0.12234506011009216. V-Accuracy: 95  T-Accuracy: 95
Iteration: 340. Loss: 0.1554853469133377. V-Accuracy: 94  T-Accuracy: 95
Iteration: 360. Loss: 0.12669718265533447. V-Accuracy: 95  T-Accuracy: 95
Iteration: 380. Loss: 0.1484023630619049. V-Accuracy: 94  T-Accuracy: 95
Iteration: 400. Loss: 0.15945787727832794. V-Accuracy: 95  T-Accuracy: 95
Iteration: 420. Loss: 0.036814115941524506. V-Accuracy: 95  T-Accuracy: 96
Iteration: 440. Loss: 0.07982578128576279. V-Accuracy: 95  T-Accuracy: 96
Iteration: 460. Loss: 0.05860297754406929. V-Accuracy: 96  T-Accuracy: 96
```

1.2 Part 2: Theoretical questions

Theory question 1: What is the *vanishing gradients problem* and why does it occur? Which activation functions are more or less impacted by this, and why?

Your answers(see next cell):

- Your answer here describing vanishing gradients problem
- Two examples of activation functions more impacted by vanishing gradients
- Two examples of activation functions less impacted by vanishing gradients, why are they impacted less?

Answers for question 1:

1) As for neural network like:

In each layer, we can get $y_i = \sigma(z_i) = \sigma(w_i x_i + b_i)$ (σ is activation function).

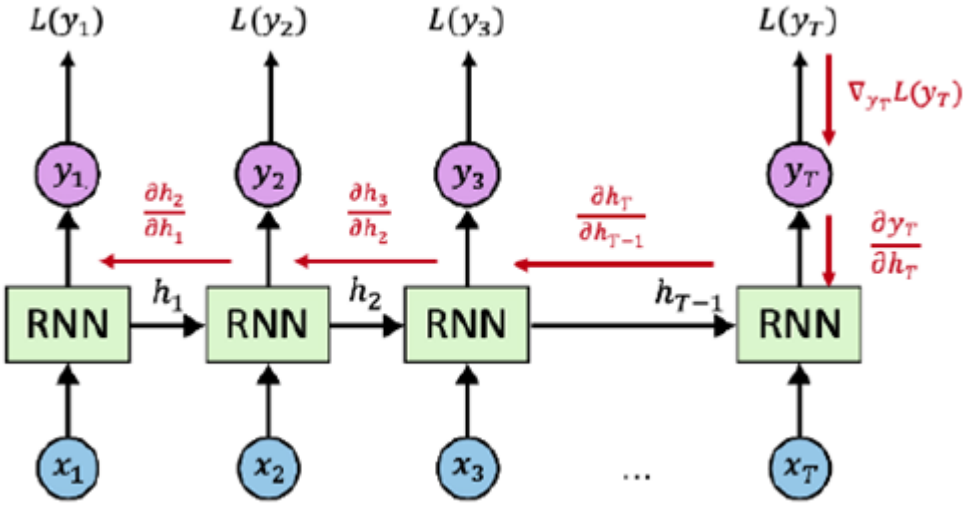
According to Backpropagation, we can get:

$$\frac{\delta C}{\partial b_1} = \frac{\partial C}{\partial y_n} \cdot \prod_{i=2}^{n} \left(\sigma'(z_i) w_i\right) \sigma'(z_1)$$

So, if $\sigma\prime < 1$ and the initial value of $w < 1$, we will find with the deeper of the network, the derivative can be zero, which is called vanishing gradients problem.

So, **sigmoid and tanh activation functions** can be more impacted by vanishing gradients, for the derivative of former is between 0 and 0.25, the latter is between 0 and 1. However, **Relu, Leaky ReLU and ELU activation functions** can be impacted less, because, when $x > 0$ the derivatives are always 1, therefore, the gradient of the deep layers can also be transferred to the shallow layers.

2) As for RNN



We can get:

$$\frac{\partial \text{Loss}_t}{\delta w_h} = \sum_{k=0}^{t} \frac{\delta \text{Loss}_t}{\delta L_t} \frac{\delta L_t}{\delta h_t} \left(\prod_{j=k+1}^{t} \frac{\delta h_j}{\delta h_{j-1}} \right) \frac{\delta h_k}{\delta w_h}$$

$\prod_{j=k+1}^{t} \frac{\delta h_j}{\delta h_{j-1}}$ causes vanishing gradients problem, the reason is the same as 1).

Theory question 2: Why do LSTMs help address the vanishing gradient problem compared to a vanilla RNN?

Answers for question 2:

$$
\begin{aligned}
i_t &= \sigma\left(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}\right) \\
f_t &= \sigma\left(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}\right) \\
o_t &= \sigma\left(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}\right) \\
g_t &= \tanh\left(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}\right) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh\left(c_t\right)
\end{aligned}
$$

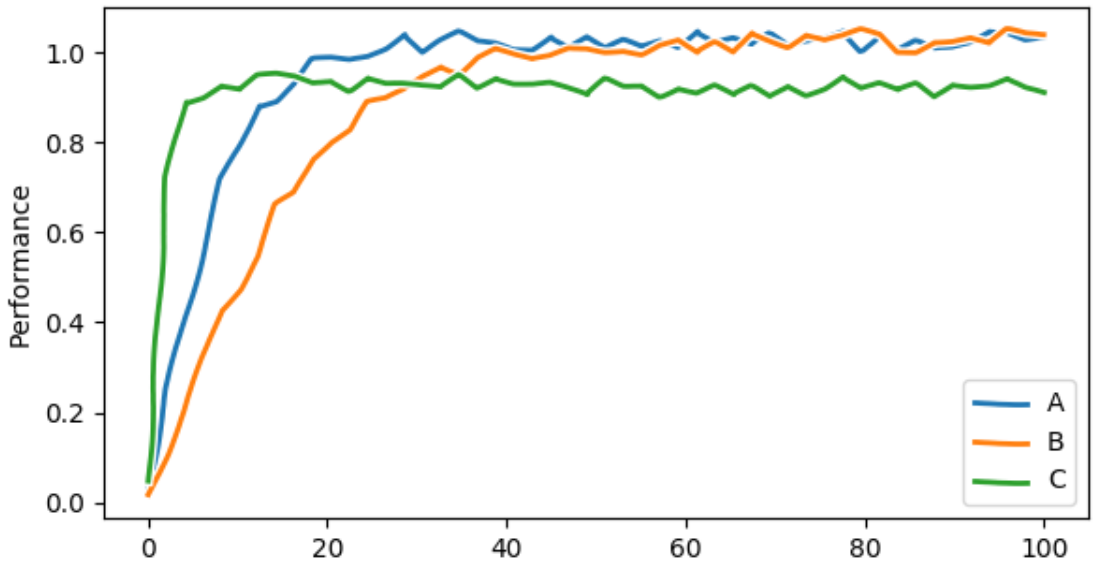
We noticed that the activation function of the first three gates is sigmoid, and this means that the output of these three gates are either close to 0 or close to 1. So as for $\frac{\delta c_t}{\delta c_{t-1}} = f_t + ...$, $\frac{\delta h_t}{\delta h_{t-1}} = o_t + ...$, f_t and o_t are 0 or 1. When the output of gate is 1, the gradient can be well propagated in the LSTM, which greatly reduces the probability of gradient vanishing. When the gate is 0, it means that the information at the previous moment has no effect on the current moment, and it is not necessary to pass the gradient back to update the parameters.

Theory question 3: The plot below shows the training curves for three models A, B, and C, trained on the same dataset up to 100 epochs. The three models are a RNN, a LSTM and a GRU, not necessarily in that order.

- Which could plausibly be which? Why? Please explain your reasoning.

(In the cell below please set the values for A_model, B_model and C_model to be 'RNN', 'LSTM' or 'GRU'. This needs to be exact for the automatic marking.)

[10]:
```
from IPython.display import Image, display
display(Image(filename='Performance by epoch.png', width=550))
```



[ ]:
```
# Answers below:

A_model = 'GRU'
B_model = 'LSTM'
C_model = 'RNN'

# Give your reasons below:
# See next cell.
```

Model C has the worst performance in the three models, so it is RNN, for it is more susceptible to the vanishing gradient problem.

What's more we see that model B takes longer to train to get similar performance as model A, so model B should be LSTM, for LSTMs have more paramteters than GRU.

Theory question 4: When might you choose to use each of the three different types of models?

Your answers:

- Type of problem when best to use vanilla RNN: We use vanilla RNN when the problem is simple and does not include long sequences of inputs.
- Type of problem to use GRU: If the task required a quite deep network, such as speech recognition, then a GRU would make sense for its efficiency, which has less paramteters.
- Type of problem to use LSTM: If we have enough data sets and computation resources, then we can use the LSTM since it contains more parameters and maybe get a better result compared to the GRU.