Deep Learning, DD2424

# Report on lab assignment 4

## Convolutional Neural Network

Majd Jamal

May 14, 2021

## 1 Main objectives and scope of the assignment

My major goals in the assignment were

- to implement and train a recurrent neural network.
- use deep learning for solving natural language processing problems.
- to try a new variant of training called backpropagation through time.

*I implemented a Recurrent Neural Network and trained it with Backpropagation Through Time (BPTT), using only NumPy.*

## 2 Method

The network was built from scratch using Python 3 and packages: NumPy and Matplotlib. The code and plots are found in Appendix Code.

# 3 Result

## (i) Gradients check

After implementing the required functions, I conducted three experiments to compare the analytic gradients with the numerical ones. The code for these experiments is shown in Appendix Code analyzeGradient(), on page 15. The absolutes difference between numerical and analytic gradients was small, i.e., ($< 1e-6$).

**Few data points & small hidden layer**

With **5 data points** and parameters **m = 5, seq_length = 5**, the difference between numerical and analytical gradients was:

Weight V: 9.95e-10

Weight U: 1.69e-09

Weight W: 6.79e-10

Weight b: 1.80e-09

Weight c: 4.24e-10

**Few data points & big hidden layer**

With **5 data points** and parameters **m = 100, seq_length = 5**, the difference between numerical and analytical gradients was:

Weight V: 6.00e-10

Weight U: 1.63e-09

Weight W: 2.51e-10

Weight b: 1.80e-09

Weight c: 4.28e-10

**Many data points & big hidden layer**

With **100 data points** and parameters **m = 100, seq_length = 100**, the difference between numerical and analytical gradients was:

Weight V: 7.50e-10

Weight U: 1.71e-09

Weight W: 1.29e-10

Weight b: 2.16e-09

Weight c: 6.54e-10

## (ii) Smooth Loss

The network was trained with **10 epochs**, and the smooth loss curve is found in Appendix Figure 1. The minimum loss was **38.87**. We can see from the graph that the learning was quick in the beginning, but slowed down after some 20 update steps.

## (iii) Synthesized text during training

Synthesized text before the first and after every 10 000th update steps until reaching 100 000 update steps.

**Before first step**
*HjJ'FXUÃH}DRWâHTEL1RXpz : um6z"YfOOwXRRbAInID"SjaCMDf))INPoR–rJOxu;)FQTAKLLRErV0TJjRkfoANZ4xv}Vzxy/âO :)uL–YKzAze,GSd)P4KGMloNy LtSQo0/G4"YPOQ4R6?Qv?WLn")GhB.xwD}OuOLMxRqVvNbRnR.OTHJLRS–QIx.RNTo1uA*

**upd. step 10 000**
Comling k Mully've lookly, stat she . Chearen the usly, I s gom tham cevire astuldols shet, veny, to'lryiss Bud doone beburd fare- daid. . Hermed, and of erecore. featredsall oich Harry dack rilly H

**20 000**
reeet begine on, uw if a facnione." Ih 'ir thry think to Syelbsing tadly noury bund," sition the wake with firantair teove non K Keting a hinge and highe win ont bred of sido set ere had lowenereso th

**30 000**
lors in ancerted awming and was dourd lotire tor he laxmy. Hermy! . . Harry

the ggleckned acy stricely. She pusses heve theove mablent in with. Peming. . . tall all the pains ace depch a hap the do

**40 000**
the garted the chamble near;. . . I what him. He pell about whit trouch ulas rest ballen. And to they frace formyion strin anfing Vardis, beliciseedled was stract thermalies dent. Fole my comes intl

**50 000**
d hell oreveny fust Harry thell brobdous Ohe's once themroughus a dome a of tok the stupsesed at whow of the with harly way dont whis Plouge all hald dossly becusdledleotcien frow, wh the realdattenti

**60 000**
oun't over the winds, shough Harry?" said Harry, Pren to then aroy the deshed on.," beately. "But hims. Oh they said, "Haprying a sard blagenturty, yircy uplep." A liten, turty eye startily of enough

**70 000**
endy's of A dow, eldroom upse'r pave said dearidl garll, been paring oren't out clourleas net on an he'd ormy reaided go, sulathing. . There ob amowenand the in winding a crusies said ther, hut at Ron

**80 000**
n deree of the fuctone Dir, for to talk listing hight looked anreling the say to elfoned dubne-. in you was us meren at at fross air cancly?" And to dath-graught walked antulored. "Ced iniven is awad

**90 000**
appro, "Ant. The come Sir foly bef's to boin and her, Workse to Mrobed pealicy to she bet. He was support a that had cethise, micke goughed, was thouldered his walled hostipe who xarll. Wilk someop

**100 000**
t fare lon to whoke voull whis those cask your. "Plort of stupion. Buth Hermaownent akepped wite hood and who demised misted of non with' sirg peenering to bick'ret old Herm it stabe, the Harry to m

## (iv) Test

Synthesized text from a network trained with 100 epochs and a smooth loss of **35.9**.

angwin, what that say, I'h munness temest the gold Listers. And Hermione's it, to wizards. Dumm took al - me else to his you to you doken, feall. . . ." he thouggeth avleand, Bagman," eqsaff we had could loid. "Jout blusured, the Dop his sliside and done shack take died as fourtringing," he'd bey?n the shom dace to wracked last happone had you you no of them then, could chair been drath senmile in the old to breather one in you towalf got the were hourting to masters what, but Ers this got, Rons doress never become ly Vernon leassed at a shair, blacked, then a wairvinit - not to the father flitted that. Was a stard, Crouch in the dour a lazz't cloud.. . we might Prives up, untous bott too nothing inve enor, Fred up. It in you prool. Harry had got gract will let Durkly a - Weall "whing at mine on ronced his," said Bulged anything his plignt are doubtakent give bagged. They was own of even won here. He else quite would keet walk was ristse no by a Art, thougharlid! Goy, Bly said,

## 4 Discussion

The training was quick in the beginning, but it became slower with time. One explanation could be that some parts of the book are harder to learn, and thus some sequences have a big loss. However, the smooth loss followed a downward pattern, indicating that the backpropagation is correctly implemented.

The network learned most characters from the book, such as Hermione, Harry, Ron, and Dumbledore.

## 5 Additional Comments

I'm satisfied with my network, because the network learned the true name of He Who Must Not Be Named. For example:

Update step 931000:
the past I me," saids I was pointuren at it; Ron, **Voldemort** fast. Look it come the eop of you hearder," said offion to drilevered - fifee, there nousy, moints ninamisenhat and shoor to saetately you

# 6 Appendix

## 6.1 Result

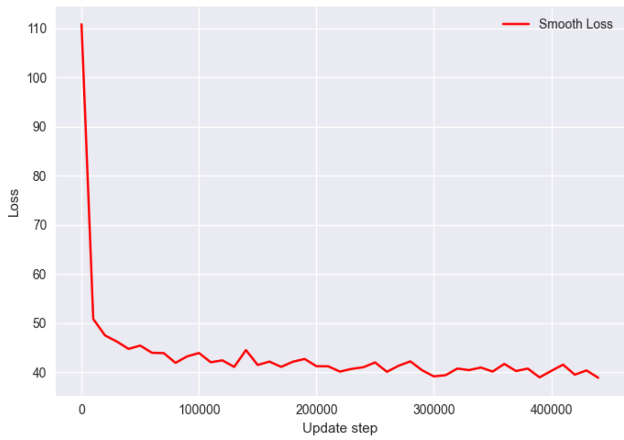


Figure 1: **Title:** Smooth Loss when training with 10 epochs. **Setting was:** m = 100, seq_length = 25, eta = 0.1, sig = 0.1, epochs = 10. **Comments:** The learned became slower after some 20 update steps. However, the loss followed a downward trend, indicating that the backpropagation through time was working properly.

## 6.2 Code

```python
__author__ = 'Majd Jamal'

import numpy as np
import matplotlib.pyplot as plt



#=-=-=-=-=-=-=-=-=
# Data loading
#=-=-=-=-=-=-=-=-=

book_data = open('goblet_book.txt', 'r').read()
unique = Counter(book_data)
chars = unique.items()
chars = list(unique.keys())
NUnique = len(chars)

char_to_ind = {}
ind_to_char = {}

X = np.zeros((NUnique, len(book_data)))
print(chars)
for i in range(len(chars)):
    vec = np.zeros((NUnique, 1))
    char = chars[i]

    vec[i] = 1

    char_to_ind[char] = vec
    ind_to_char[i] = char

for i in range(len(book_data)):
    char = book_data[i]
    vec = char_to_ind[char]

    X[:, i] = vec[:,0]

np.save('book_data.npy', book_data)
np.save('X.npy', X)
np.save('ind_to_char.npy', ind_to_char)
np.save('char_to_ind.npy', char_to_ind)
np.save('NUnique.npy', NUnique)

#=-=-=-=-=-=-=-=-=
# Utils
#=-=-=-=-=-=-=-=-=

```

```
class Data:

    def __init__(self, book_data, X, ind_to_char, char_to_ind, NUnique):
        """ Object to store data
        """
        self.book_data = book_data
        self.X = X
        self.ind_to_char = ind_to_char
        self.char_to_ind = char_to_ind
        self.NUnique = NUnique

class Params:

    def __init__(self, m, seq_length, eta, sig, epochs):
        """ Object to store hyperparamters
        """
        self.m = m # hidden units
        self.seq_length = seq_length
        self.eta = eta  # learning rate
        self.sig = sig  # variance when initializing weights
        self.epochs = epochs


def getData():

    book_data = np.load('data/processed/book_data.npy')
    X = np.load('data/processed/X.npy')
    ind_to_char = np.load('data/processed/ind_to_char.npy', allow_pickle=True)
    char_to_ind = np.load('data/processed/char_to_ind.npy', allow_pickle=True)
    NUnique = np.load('data/processed/NUnique.npy')

    data = Data(book_data, X, ind_to_char, char_to_ind, NUnique)

    return data


def softmax(x):
    """ Standard definition of the softmax function """
    e_x = np.exp(x - np.max(x))
    return e_x / np.sum(e_x, axis=0)

def tanh(x):
    """ Standard definition of the tanH function """
    return np.sinh(x) / np.cosh(x)

def plotter(step, train):
    """ Plots validation loss from a training session.
    :param step: update steps
    :param val: validation loss
    """
```

```
    plt.style.use('seaborn')
    plt.plot(step, train, color = 'red', label = 'Smooth Loss')
    plt.xlabel('Update step')
    plt.ylabel('Loss')
    plt.legend()
    plt.savefig('results/loss')
    plt.close()


#=-=-=-=-=-=-=-=-=
# Model
#=-=-=-=-=-=-=-=-=

class VRNN:
    """ Vanilla Recurrent Neural Network
        used for natural language processing.
    """
    def __init__(self):

        self.W = None
        self.V = None
        self.U = None
        self.b = None
        self.c = None

        self.char_to_ind = None
        self.ind_to_char = None

        self.AdaGradTerm = {}

        self.lossData = [[],[]] # [step, train_loss, val_loss]

    #[a, h, o, p]
    def forward(self, X, h0, V, U, W, b, c):
        """ Computes one pass with
            tanH and softmax activations.
        :param X: vector representation of a character shape = (K, 1)
        :param h0: initial hidden state, shape = (m, 1)
        :param V: Weights in the output layer, shape = (K, m)
        :param U: Second Weights in the input layer, shape = (m, K)
        :param W: First Weights in the input layer, shape = (m, m)
        :param b: bias for the first activation
        :param c: bias for the second activation
        :return a: one pass values, shape = (m, 1)
        :return h: hidden values, shape = (m,1)
        :return o: non-normalized outputs, shape = (K, 1)
        :return p: normalized probabilities, shape = (K, 1)
        """

        a = W @ h0 + U @ X + b
```

```python
        h = tanh(a)
        o = V @ h + c
        p = softmax(o)

        return a, h, o, p

    #[H0, A, H, P]
    def train(self, X, V, U, W, b, c, h0):
        """ Trains the network with a sequence of data points
        :param X: characters, shape = (K, seq_lenght)
        :param V: Weights in the output layer, shape = (K, m)
        :param U: Second Weights in the input layer, shape = (m, K)
        :param W: First Weights in the input layer, shape = (m, m)
        :param b: bias for the first activation
        :param c: bias for the second activation
        :return H0: intial hidden states, used in the backward pass
        :return A: One pass values, used in the backward pass
        :return H: hidden states after training
        :return P: normalized probabilites
        """
        m, _ = self.W.shape
        K, Npts = X.shape

        H0 = np.zeros((m, Npts))    #Initial hidden states
        H = np.zeros((m, Npts))     #Hidden states after one pass
        A = np.zeros((m, Npts))     #One pass values
        P = np.zeros((K, Npts))     #Normalized output probabilities

        for itr in range(Npts):
            x = X[:, itr].reshape(-1,1) #char

            a, h, o, p = self.forward(x, h0,
                V, U, W, b, c)

            H0[:, itr] = h0[:, 0]
            A[:, itr] = a[:, 0]
            H[:, itr] = h[:, 0]
            P[:, itr] = p[:, 0]

            h0 = h

        return H0, A, H, P

    #[dV, dU, dW, db, dc]
    def backward(self, X, H0, Y, P, H, A):
        """
        :param X: matrix representation of a word,
            shape = (K, Npts)
        :param H0: Initial hidden states that were used in the forward pass,
            shape = (m, Npts)
```

```
        :param Y: true labels,
            shape = (K, Npts)
        :param P: normalized probabilities,
            shape = (K, Npts)
        :param H: hidden states,
            shape = (m, Npts)
        :param A: one pass values,
            shape = (m, Npts)
        :return dV: gradients for weights V
        :return dU: gradients for weights U
        :return dW: gradients for weights W
        :return db: gradients for weights b
        :return dc: gradients for weights c
        notations:
                da - dL_da_{t}
                dh - dL_dh_{t}
                g - dL_do_{t}
                g_da - dL_da_{t + 1}
        """

        K, Npts = Y.shape
        m, _ = H.shape

        G = - (Y - P)

        dV = G @ H.T
        dc = G @ np.ones((Npts, 1))

        G_da = np.zeros((m, Npts))

        for t in range(Npts - 1, -1, -1):

            g = G[:, t]
            a_t = A[:, t]
            tanhD = np.diag(1 -  np.square(np.tanh(a_t)))

            if t == Npts - 1:
                dh = self.V.T @ g
            else:
                g_da = G_da[:, t + 1]
                dh = self.V.T @g + self.W.T @ g_da

            da = tanhD.T @ dh
            G_da[:, t] = da

        G = G_da

        dW = G @HO.T
        dU = G @X.T
```

```
        db = G @np.ones((Npts, 1))

        return dV, dU, dW, db, dc

    def exploding(self, grad):
        """ Mitigate exploding gradients
        :param grad: weight gradient
        :return: cliped version of the gradient matrix
        """
        grad = np.where(grad > 5, 5, grad)
        grad = np.where(grad < -5, -5, grad)
        return grad

    def update(self, dL_dV, dL_dU, dL_dW, dL_db, dL_dc, eta, eps = 1e-8):
        """ Updates weights
        :param dL_dV: gradients for weight V
        :param dL_dU: gradients for weight U
        :param dL_dW: gradients for weight W
        :param dL_db: gradients for weight b
        :param dL_dc: gradients for weight c
        :param eta: learning rate
        :param eps: constant to avoid diving with 0
        """
        m_V = self.AdaGradTerm['V'] + np.square(dL_dV)
        m_U = self.AdaGradTerm['U'] + np.square(dL_dU)
        m_W = self.AdaGradTerm['W'] + np.square(dL_dW)
        m_b = self.AdaGradTerm['b'] + np.square(dL_db)
        m_c = self.AdaGradTerm['c'] + np.square(dL_dc)

        self.AdaGradTerm['V'] = m_V
        self.AdaGradTerm['U'] = m_U
        self.AdaGradTerm['W'] = m_W
        self.AdaGradTerm['b'] = m_b
        self.AdaGradTerm['c'] = m_c

        self.V -= eta/np.sqrt(m_V + eps) * self.exploding(dL_dV)
        self.U -= eta/np.sqrt(m_U + eps) * self.exploding(dL_dU)
        self.W -= eta/np.sqrt(m_W + eps) * self.exploding(dL_dW)

        self.b -= eta/np.sqrt(m_b + eps) * self.exploding(dL_db)
        self.c -= eta/np.sqrt(m_c + eps) * self.exploding(dL_dc)


    def loss(self, Y, P):
        """ Compute Cross-Entropy loss
        :param Y: true labels, shape = (K, Npts)
        :param P: normalized probabilities, shape = (K, Npts)
        :return: cross entropy
        """
        return np.sum(-np.log(np.einsum('ij,ji->i', Y.T, P)))
```

```python

    def getLoss(self):
        """ Returns loss
        :return lossData: array with steps and loss, [[step], [loss]]
        """
        return self.lossData

    def getWeigths(self):
        return self.V,self.U, self.W, self.b, self.c

    #[dV_num, dU_num, dW_num, db_num, dc_num]
    def ComputeGradsNumSlow(self, X, Y, h = 1e-4):
        """ Computes numerical gradients
        :param X: data points, shape = (K, Npts)
        :param Y: true labels, shape = (K, Npts)
        :param h: derivative step, const
        :return dV_num: Numerical gradients for weight V
        :return dU_num: Numerical gradients for weight U
        :return dW_num: Numerical gradients for weight W
        :return db_num: Numerical gradients for weight b
        :return dc_num: Numerical gradients for weight c
        """
        m,_ = self.W.shape
        h_init = np.zeros((m,1))
        # V
        dV_num = np.zeros(self.V.shape)
        for i in range(self.V.shape[0]):
            for j in range(self.V.shape[1]):

                V_try = np.array(self.V)
                V_try[i, j] -= h
                _,_, _, P = self.train(X, V_try, self.U, self.W, self.b, self.c, h_init)
                c1 = self.loss(Y, P)

                V_try = np.array(self.V)
                V_try[i, j] += h
                _,_, _, P = self.train(X, V_try, self.U, self.W, self.b, self.c, h_init)
                c2 = self.loss(Y, P)

                dV_num[i,j] = (c2 - c1) / (2 * h)

        # U
        dU_num = np.zeros(self.U.shape)
        for i in range(self.U.shape[0]):
            for j in range(self.U.shape[1]):
                U_try = np.array(self.U)
                U_try[i, j] -= h
                _,_, _, P = self.train(X, self.V, U_try, self.W, self.b, self.c, h_init)
                c1 = self.loss(Y, P)

```

```
                U_try = np.array(self.U)
                U_try[i, j] += h
                _,_, _, P = self.train(X, self.V, U_try, self.W, self.b, self.c, h_init)
                c2 = self.loss(Y, P)

                dU_num[i,j] = (c2 - c1) / (2 * h)

        # W
        dW_num = np.zeros(self.W.shape)
        for i in range(self.W.shape[0]):
            for j in range(self.W.shape[1]):
                W_try = np.array(self.W)
                W_try[i, j] -= h
                _,_, _, P = self.train(X, self.V, self.U, W_try, self.b, self.c, h_init)
                c1 = self.loss(Y, P)

                W_try = np.array(self.W)
                W_try[i, j] += h
                _,_, _, P = self.train(X, self.V, self.U, W_try, self.b, self.c, h_init)
                c2 = self.loss(Y, P)

                dW_num[i,j] = (c2 - c1) / (2 * h)


        db_num = np.zeros(self.b.shape)
        for i in range(self.b.size):
            b_try = np.array(self.b)
            b_try[i] -= h
            _,_, _, P = self.train(X, self.V, self.U, self.W, b_try, self.c, h_init)
            c1 = self.loss(Y, P)

            b_try = np.array(self.b)
            b_try[i] += h
            _,_, _, P = self.train(X, self.V, self.U, self.W, b_try, self.c, h_init)
            c2 = self.loss(Y, P)

            db_num[i] = (c2 - c1) / (2 * h)

        # c
        dc_num = np.zeros(self.c.shape)
        for i in range(self.c.size):
            c_try = np.array(self.c)
            c_try[i] -= h
            _,_, _, P = self.train(X, self.V, self.U, self.W, self.b, c_try,h_init)
            c1 = self.loss(Y, P)

            c_try = np.array(self.c)
            c_try[i] += h
            _,_, _, P = self.train(X, self.V, self.U, self.W, self.b, c_try, h_init)
            c2 = self.loss(Y, P)
```

```python

            dc_num[i] = (c2 - c1) / (2 * h)


        return dV_num, dU_num, dW_num, db_num, dc_num

    def difference(self, grad_num, grad_an):
        """Computes difference between numerical and analytical gradients
        :grad_num: numerical gradients
        :grad_an: analytical gradients
        """
        diff = np.sum(np.abs(np.subtract(grad_an, grad_num)))
        diff /= np.sum(np.add(np.abs(grad_an), np.abs(grad_num)))
        return diff

    def AnalyzeGradients(self, X, Y):
        """ Compares numerical and analytical gradients
        :param X: data points, shape = (K, Npts)
        :param Y: true labels, shape = (K, Npts)
        """
        m, _ = self.W.shape
        h_init = np.zeros((m,1))

        dV_num, dU_num, dW_num, db_num, dc_num = self.ComputeGradsNumSlow(
        X, Y
        )

        H0, A, H, P = self.train(X, self.V, self.U, self.W, self.b, self.c, h_init)

        dV_an, dU_an, dW_an, db_an, dc_an = self.backward(X, H0, Y, P, H, A)

        V_d = self.difference(dV_num, dV_an)
        U_d = self.difference(dU_num, dU_an)
        W_d = self.difference(dW_num, dW_an)
        b_d = self.difference(db_num, db_an)
        c_d = self.difference(dc_num, dc_an)

        print("\x1b[94m =-=-=-=- Numerical vs Analytic gradients -=-=-=-= \x1b[39m")
        print("\x1b[94m =-=- V: \x1b[39m", V_d)
        print("\x1b[94m =-=- U: \x1b[39m", U_d)
        print("\x1b[94m =-=- W: \x1b[39m", W_d)
        print("\x1b[94m =-=- b: \x1b[39m", b_d)
        print("\x1b[94m =-=- c: \x1b[39m", c_d)
        print("\x1b[94m =-=-=-=- @ -=-=-=-= \x1b[39m")

    #[text]
    def synthesize(self, h, x, N):
        """ Generates text based on an initial hidden state and character x.
        :param h: initial hidden state, shape = (m, 1)
        :param x: initial character, shape = (K, 1)
```

```
        :param N: number of characters to generate, const
        :param text: synthesized text, string
        """

        K, _ = x.shape
        m, _ = h.shape
        text = ''

        for n in range(N):

            if n == 0:
                char = np.argmax(x, axis=0)[0]
            else:
                _,h_new,_, prob = self.forward(x, h,
                    self.V, self.U, self.W, self.b, self.c)
                char = np.random.choice(K, 1, p=prob[:,0])[0]

                x = np.zeros((K, 1))
                x[char] = 1
                h = h_new

            char = self.ind_to_char.item().get(char)
            text += char

        return text

    def fit(self, data, params):
        """ Trains the recurrent network with sequences of words.
        :param data: Object containg data, i.e., book_data,
            X, NUnique, char_to_ind, ind_to_char
        :param params: Object containg hyperparameters, i.e., , m,
        seq_length, eta, sig, epochs
        """

        ##
        ## Unpacking data
        ##
        book_data = data.book_data
        X = data.X
        K = data.NUnique
        self.char_to_ind = data.char_to_ind
        self.ind_to_char = data.ind_to_char

        ##
        ##  Unpacking params
        ##
        m = params.m
        seq_length = params.seq_length
        eta = params.eta
        sig = params.sig
```

```
        epochs = params.epochs

        ##
        ##  Initialize weights and
        ##  biases
        ##
        np.random.seed(400)
        self.U = np.random.randn(m, K) * sig
        self.W = np.random.randn(m, m) * sig
        self.V = np.random.randn(K, m) * sig
        self.b = np.zeros((m,1))
        self.c = np.zeros((K,1))



        ##
        ##  AdaGrad initialization
        ##
        self.AdaGradTerm['V'] = 0
        self.AdaGradTerm['U'] = 0
        self.AdaGradTerm['W'] = 0
        self.AdaGradTerm['b'] = 0
        self.AdaGradTerm['c'] = 0

        #self.V,self.U, self.W, self.b, self.c = np.load('weigths.npy', allow_pickle = Tru
        #text = self.synthesize(np.zeros((m,1)), X[:, 1200].reshape(-1,1), 1000)
        #print(text)
        #=-=-=-=-=-=-=-=-=-=-=-=-
        #   Analyze Gradients
        #=-=-=-=-=-=-=-=-=-=-=-=-
        #X_analyze = X[:, 0: seq_length]
        #Y_analyze = X[:, 0 + 1: seq_length + 1]
        #self.AnalyzeGradients(X_analyze, Y_analyze)

        #"""Training
        print('\x1b[91m =-=-=-=- Network parameters -=-=-=-= \x1b[39m')
        print('\x1b[91m =- epochs: \x1b[39m', epochs, '\x1b[91m learning_rate: \x1b[39m',
        print('\x1b[91m =- hidden_units: \x1b[39m', m, '\x1b[91m seq_length: \x1b[39m', se
        print('\x1b[91m =-=-=-=- Starting training -=-=-=-= \n \x1b[39m')

        #update_steps = X.shape[1] - seq_length - 1

        update_steps = round((X.shape[1] - seq_length)/seq_length)
        smooth_loss = 0
        N = 200
        #text = self.synthesize(np.zeros((m,1)), X[:, 0].reshape(-1,1), N)
        for epoch in range(epochs):
            e = 0
            h_init = np.zeros((m,1))

```

```
            for itr in range(update_steps):
            #for itr in range(10000):

                #if e + seq_length + 1 > X.shape[1]:
                #    break
                step = itr + epoch * update_steps
                X_train = X[:, e : e + seq_length]
                Y_train = X[:, e + 1: e + seq_length + 1]

                H0, A, H, P = self.train(X_train, self.V, self.U,
                    self.W, self.b, self.c, h_init)

                grads = self.backward(X_train, H0, Y_train, P, H, A)

                self.update(*grads, eta)

                c = self.loss(P, Y_train)

                if itr == 0 and epoch == 0:
                    smooth_loss = c
                else:
                    smooth_loss = 0.999 * smooth_loss + 0.001 * c

                if step % 10000 == 0:
                    #c = self.loss(P, Y_train)
                    #if itr == 0 and epoch == 0:
                    #    smooth_loss = c
                    #else:
                    #    smooth_loss = 0.999 * smooth_loss + 0.001 * c


                    self.lossData[0].append(step)
                    self.lossData[1].append(smooth_loss)
                    text = self.synthesize(H0[:, 0].reshape(-1,1), X_train[:, 0].reshape(-
                    print('epoch: ', epoch, ' iter: ', step, ' loss: ', smooth_loss)
                    print('Text: \n ', "\x1b[93m" + text + "\x1b[39m \n")

                h_init = H[:, -1].reshape(-1,1)
                e += seq_length

            print('\x1b[36m Epoch: \x1b[39m', epoch, '\n')

        print('\x1b[91m =-=-=-=- Training Complete -=-=-=-= \n \x1b[39m')
        #"""

#=-=-=-=-=-=-=-=-=
# Experiment
#=-=-=-=-=-=-=-=-=
data = getData()
params = Params(
```

```
    m = 100, seq_length = 25, eta = 0.1, sig = 0.1, epochs = 10
    )

vrnn = VRNN()
vrnn.fit(data, params)
loss = vrnn.getLoss()
plotter(loss[0], loss[1])
weigths = vrnn.getWeigths()

np.save('weigths.npy', weigths)
np.save('loss.npy', loss)
```