```python
# ============================
# importing libraries
# ============================

import tensorflow as tf
from tensorflow.keras.layers import Input,Dense,Flatten,Conv2D,MaxPool2D,Activation,Dropout,Embedding,GRU,RepeatVector
from tensorflow.keras.models import Sequential,Model
from tensorflow.keras import regularizers
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.callbacks import EarlyStopping,ModelCheckpoint
from tensorflow.keras.metrics import RootMeanSquaredError
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers.experimental.preprocessing import TextVectorization
from tensorflow.keras.layers.experimental import preprocessing
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import model_from_json
from sklearn.model_selection import train_test_split

import numpy as np

import unicodedata

import re

import matplotlib.pyplot as plt
%matplotlib inline



# ============================
# problem_set_1
# ============================

BATCH_SIZE = 64
EPOCHS = 50

def vgg16_custom_arch():

  model = Sequential()
  model.add(Conv2D(input_shape=(48,48,1),filters=64,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_1_1'))
  model.add(Conv2D(filters=64,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_1_2'))
  model.add(MaxPool2D(pool_size=(2,2),strides=(2,2),name='max_pool_1'))

  model.add(Conv2D(filters=128,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_2_1'))
  model.add(Conv2D(filters=128,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_2_2'))
  model.add(MaxPool2D(pool_size=(2,2),strides=(2,2),name='max_pool_2'))


  model.add(Conv2D(filters=256,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_3_1'))
  model.add(Conv2D(filters=256,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_3_2'))
  model.add(Conv2D(filters=256,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_3_3'))
  model.add(MaxPool2D(pool_size=(2,2),strides=(2,2),name='max_pool_3'))

  model.add(Conv2D(filters=512,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_4_1'))
  model.add(Conv2D(filters=512,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_4_2'))
  model.add(Conv2D(filters=512,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_4_3'))
  model.add(MaxPool2D(pool_size=(2,2),strides=(2,2),name='max_pool_4'))

  model.add(Conv2D(filters=512,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_5_1'))
  model.add(Conv2D(filters=512,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_5_2'))
  model.add(Conv2D(filters=512,kernel_size=(3,3),strides=(1,1),padding='same',dilation_rate=(1,1),activation='relu',name='conv_5_3'))
  model.add(MaxPool2D(pool_size=(2,2),strides=(2,2),name='max_pool_5'))

  model.add(Flatten(name='flatten_1'))
  model.add(Dense(units=4096,activation='relu',name='fc1'))
  model.add(Dropout(rate=0.5))
  model.add(Dense(units=4096,activation='relu',name='fc2'))
  model.add(Dropout(rate=0.5))
  model.add(Dense(units=1,activation='linear',name='output'))

  model.compile(loss='mean_squared_error', optimizer='adam',metrics=[RootMeanSquaredError()])
  return model

```

```python
vgg16_model = vgg16_custom_arch()

X_tr: np.ndarray = np.load('./facesAndAges/faces.npy')
y_tr: np.ndarray = np.load('./facesAndAges/ages.npy')

shuffling_indices: np.ndarray = np.arange(X_tr.shape[0])
np.random.shuffle(shuffling_indices)
X_tr: np.ndarray = X_tr[shuffling_indices]
y_tr: np.ndarray = y_tr[shuffling_indices]
del shuffling_indices
trainX,trainY = X_tr[0:5250,:],y_tr[0:5250]
valX,valY = X_tr[5250:6000,:],y_tr[5250:6000]
testX,testY = X_tr[6000:7500,:],y_tr[6000:7500]
del X_tr,y_tr

history = vgg16_model.fit(trainX,trainY,batch_size=BATCH_SIZE,epochs=EPOCHS,verbose=1,validation_data=(valX,valY),callbacks=[EarlyStopping(monitor='loss',mode='auto',min_delta=6e-2,patience=6,verbose=0,)])
test_loss, test_rmse = vgg16_model.evaluate(testX,testY)
print(f'Test Loss: {test_loss:.4f}, Test RMSE: {test_rmse:.4f}')
# Test Loss: 141.9804, Test RMSE: 11.9156

fig, axs = plt.subplots(2, 1, figsize=(10,13))
axs[0].plot(history.history['loss'])
axs[0].plot(history.history['val_loss'])
axs[0].title.set_text('Training Loss vs Validation Loss')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Loss')
axs[0].legend(['Train','Val'])
axs[1].plot(history.history['root_mean_squared_error'])
axs[1].plot(history.history['val_root_mean_squared_error'])
axs[1].title.set_text('Training RMSE vs Validation RMSE')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('RMSE')
axs[1].legend(['Train', 'Val'])






# ============================
# problem_set_2
# ============================

X_train: np.ndarray = np.load("./homework5_question2_data/X_train.npy",allow_pickle=True)
y_train: np.ndarray = np.load("./homework5_question2_data/y_train.npy",allow_pickle=True)
X_test: np.ndarray = np.load("./homework5_question2_data/X_test.npy",allow_pickle=True)
y_test: np.ndarray = np.load("./homework5_question2_data/y_test.npy",allow_pickle=True)

hidden_size = 8
epochs = 6
learning_rate = 1e-4

class VanillaRNN:
    def __init__(self, input_size, hidden_size):
        self.input_size = input_size
        self.hidden_size = hidden_size

        self.W_xh = np.random.randn(hidden_size, input_size)
        self.W_hh = np.random.randn(hidden_size, hidden_size)
        self.W_hy = np.random.randn(input_size, hidden_size)
        self.b_h = np.zeros((hidden_size, 1))
        self.b_y = np.zeros((input_size, 1))

    def forward(self, inputs, seq_length):
        hidden_states = np.zeros((seq_length, self.hidden_size))
        outputs = np.zeros((seq_length, self.input_size))

        h_t = np.zeros((self.hidden_size, 1))

        for t in range(seq_length):
            x_t = inputs[t].reshape(-1, 1)
            h_t = np.tanh(np.dot(self.W_xh, x_t) + np.dot(self.W_hh, h_t) + self.b_h)
            y_t = np.dot(self.W_hy, h_t) + self.b_y
```

```
            hidden_states[t] = h_t.flatten()
            outputs[t] = y_t.flatten()

        return hidden_states, outputs

    def backward(self, inputs, hidden_states, outputs, targets, learning_rate):
        seq_length = inputs.shape[0]
        dW_xh, dW_hh, dW_hy, db_h, db_y = (
            np.zeros_like(self.W_xh),
            np.zeros_like(self.W_hh),
            np.zeros_like(self.W_hy),
            np.zeros_like(self.b_h),
            np.zeros_like(self.b_y),
        )
        dh_next = np.zeros((self.hidden_size, 1))

        for t in reversed(range(seq_length)):
            x_t = inputs[t].reshape(-1, 1)
            h_t = hidden_states[t].reshape(-1, 1)
            y_t = outputs[t].reshape(-1, 1)

            if t < len(targets):
                target_t = targets[t].reshape(-1, 1)
                dy = y_t - target_t
            else:
                dy = y_t - 0

            dW_hy += np.dot(dy, h_t.T)
            db_y += dy

            dh = np.dot(self.W_hy.T, dy) + dh_next
            dh_raw = (1 - h_t ** 2) * dh
            db_h += dh_raw

            dW_xh += np.dot(dh_raw, x_t.T)
            dW_hh += np.dot(dh_raw, hidden_states[t - 1].reshape(-1, 1).T) if t > 0 else 0
            dh_next = np.dot(self.W_hh.T, dh_raw)

        for dparam in [dW_xh, dW_hh, dW_hy, db_h, db_y]:
            np.clip(dparam, -5, 5, out=dparam)

        self.W_xh -= learning_rate * dW_xh
        self.W_hh -= learning_rate * dW_hh
        self.W_hy -= learning_rate * dW_hy
        self.b_h -= learning_rate * db_h
        self.b_y -= learning_rate * db_y

def plot_VanillaRNN():
  input_size = X_train[0].shape[1]
  rnn = VanillaRNN(input_size, hidden_size)
  losses = []

  for epoch in range(epochs):
      total_loss = 0
      for i in range(len(X_train)):
          inputs = X_train[i]
          targets = y_train[i]
          seq_length = len(inputs)

          hidden_states, outputs = rnn.forward(inputs, seq_length)
          loss = np.mean(((outputs - targets) ** 2))

          rnn.backward(inputs, hidden_states, outputs, targets, learning_rate)
          total_loss += loss

      average_loss = total_loss / len(X_train)
      losses.append(average_loss)

      print(f"Epoch {epoch + 1}/{epochs}, Loss: {average_loss:.4f}")

  plt.plot(range(1, epochs + 1), losses)
  plt.xlabel('Epochs')
  plt.ylabel('Loss')
```

```python
  plt.title('Training Epoch vs. Loss VanillaRNN')
  plt.show()

  total_loss = 0
  for i in range(len(X_test)):
      inputs = X_test[i]
      targets = y_test[i]
      seq_length = len(inputs)
      _, outputs = rnn.forward(inputs, seq_length)
      total_loss += np.mean(((outputs - targets) ** 2))
  average_loss = total_loss / len(X_test)
  print(f'\nVanillaRNN Test Loss: {average_loss:.4f}\n')
  return None

class VanillaRNNMin:
    def __init__(self, input_size, hidden_size):
        self.input_size = input_size
        self.hidden_size = hidden_size


        self.W_xh = [np.random.randn(hidden_size, input_size) for _ in range(hidden_size)]
        self.W_hh = [np.random.randn(hidden_size, hidden_size) for _ in range(hidden_size)]
        self.W_hy = np.random.randn(input_size, hidden_size)
        self.b_h = [np.zeros((hidden_size, 1)) for _ in range(hidden_size)]
        self.b_y = np.zeros((input_size, 1))

    def forward(self, inputs, seq_length):
        hidden_states = np.zeros((seq_length, self.hidden_size))
        outputs = np.zeros((seq_length, self.input_size))

        h_t = [np.zeros((self.hidden_size, 1)) for _ in range(self.hidden_size)]

        for t in range(seq_length):
            x_t = inputs[t].reshape(-1, 1)
            for i in range(self.hidden_size):
                h_t[i] = np.tanh(np.dot(self.W_xh[i], x_t) + np.dot(self.W_hh[i], h_t[i]) + self.b_h[i])
            y_t = np.dot(self.W_hy, h_t[-1]) + self.b_y

            hidden_states[t] = h_t[-1].flatten()
            outputs[t] = y_t.flatten()

        return hidden_states, outputs

    def backward(self, inputs, hidden_states, outputs, targets, learning_rate=0.01):
        seq_length, input_size = inputs.shape
        dW_xh, dW_hh, dW_hy, db_h, db_y = (
            [np.zeros_like(self.W_xh[0]) for _ in range(self.hidden_size)],
            [np.zeros_like(self.W_hh[0]) for _ in range(self.hidden_size)],
            np.zeros_like(self.W_hy),
            [np.zeros_like(self.b_h[0]) for _ in range(self.hidden_size)],
            np.zeros_like(self.b_y),
        )
        dh_next = np.zeros((self.hidden_size, 1))

        for t in reversed(range(seq_length)):
            x_t = inputs[t].reshape(-1, 1)
            h_t = [hidden_states[t].reshape(-1, 1) for _ in range(self.hidden_size)]
            y_t = outputs[t].reshape(-1, 1)

            if t < len(targets):
                target_t = targets[t].reshape(-1, 1)
                dy = y_t - target_t
            else:
                dy = y_t - 0

            dW_hy += np.dot(dy, h_t[-1].T)
            db_y += dy


            dh = np.dot(self.W_hy.T, dy) + dh_next
            dh_raw = [np.dot(self.W_hh[i].T, dh) for i in range(self.hidden_size)]
            db_h = [dhr + dh for dhr, dh in zip(dh_raw, dh_next)]

            dW_xh = [dwxh + np.dot(dhr, x_t.T) for dwxh, dhr in zip(dW_xh, dh_raw)]
            dW_hh = [dwhh + np.dot(dhr, h.T) for dwhh, dhr, h in zip(dW_hh, dh_raw, h_t)]
            dh_next = np.dot(self.W_hh[-1].T, dh_raw[-1])
```

```
            dh_next = np.dot(self.W_hh[-1].T, dh_raw[-1])

        for i in range(self.hidden_size):
            for dparam in [dW_xh[i], dW_hh[i], db_h[i]]:
                np.clip(dparam, -5, 5, out=dparam)
            self.W_xh[i] -= learning_rate * dW_xh[i]
            self.W_hh[i] -= learning_rate * dW_hh[i]
            self.b_h[i] -= learning_rate * db_h[i]

        for dparam in [dW_hy, db_y]:
            np.clip(dparam, -5, 5, out=dparam)

        self.W_hy -= learning_rate * dW_hy
        self.b_y -= learning_rate * db_y

def plot_VanillaRNNMin():
  min_seq_length = min(set([i.shape[0] for i in X_train]))
  X_train_min_truncated = [i[:min_seq_length] for i in X_train]
  X_test_min_truncated = [i[:min_seq_length] for i in X_test]
  input_size = X_train_min_truncated[0].shape[1]
  rnn = VanillaRNNMin(input_size, hidden_size)
  losses = []

  for epoch in range(epochs):
      total_loss = 0

      for i in range(len(X_train_min_truncated)):
          inputs = X_train_min_truncated[i]
          targets = y_train[i]

          seq_length, input_size = inputs.shape
          hidden_states, outputs = rnn.forward(inputs, seq_length)
          loss = np.mean(((outputs - targets) ** 2))

          rnn.backward(inputs, hidden_states, outputs, targets, learning_rate)
          total_loss += loss

      average_loss = total_loss / len(X_train_min_truncated)
      losses.append(average_loss)

      print(f"Epoch {epoch + 1}/{epochs}, Loss: {average_loss:.4f}")


  plt.plot(range(1, epochs + 1), losses)
  plt.xlabel('Epochs')
  plt.ylabel('Loss')
  plt.title('Training Epoch vs. Loss VanillaRNNMin')
  plt.show()

  total_loss = 0
  for i in range(len(X_test_min_truncated)):
      inputs = X_test_min_truncated[i]
      targets = y_test[i]
      seq_length = len(inputs)
      _, outputs = rnn.forward(inputs, seq_length)
      total_loss += np.mean(((outputs - targets) ** 2))
  average_loss = total_loss / len(X_test_min_truncated)
  print(f'\nVanillaRNNMin Test Loss: {average_loss:.4f}\n')
  return None

class VanillaRNNMax:
    def __init__(self, input_size, hidden_size):
        self.input_size = input_size
        self.hidden_size = hidden_size

        self.W_xh = [np.random.randn(hidden_size, input_size) for _ in range(hidden_size)]
        self.W_hh = [np.random.randn(hidden_size, hidden_size) for _ in range(hidden_size)]
        self.W_hy = np.random.randn(input_size, hidden_size)
        self.b_h = [np.zeros((hidden_size, 1)) for _ in range(hidden_size)]
        self.b_y = np.zeros((input_size, 1))

    def forward(self, inputs, seq_length):
        hidden_states = np.zeros((seq_length, self.hidden_size))
        outputs = np.zeros((seq_length, self.input_size))

        h_t = [np.zeros((self.hidden_size, 1)) for _ in range(self.hidden_size)]
```

```
        h_t = [np.zeros((self.hidden_size, 1)) for _ in range(self.hidden_size)]

        for t in range(seq_length):
            x_t = inputs[t].reshape(-1, 1)
            for i in range(self.hidden_size):
                h_t[i] = np.tanh(np.dot(self.W_xh[i], x_t) + np.dot(self.W_hh[i], h_t[i]) + self.b_h[i])
            y_t = np.dot(self.W_hy, h_t[-1]) + self.b_y

            hidden_states[t] = h_t[-1].flatten()
            outputs[t] = y_t.flatten()

        return hidden_states, outputs

    def backward(self, inputs, hidden_states, outputs, targets, seq_length, max_seq_length, learning_rate):
        criteria_c = max_seq_length - seq_length
        dW_xh, dW_hh, dW_hy, db_h, db_y = (
            [np.zeros_like(self.W_xh[0]) for _ in range(self.hidden_size)],
            [np.zeros_like(self.W_hh[0]) for _ in range(self.hidden_size)],
            np.zeros_like(self.W_hy),
            [np.zeros_like(self.b_h[0]) for _ in range(self.hidden_size)],
            np.zeros_like(self.b_y),
        )
        dh_next = np.zeros((self.hidden_size, 1))

        total_loss = 0

        for t in reversed(range(seq_length)):
            x_t = inputs[t].reshape(-1, 1)
            h_t = [hidden_states[t].reshape(-1, 1) for _ in range(self.hidden_size)]
            y_t = outputs[t].reshape(-1, 1)

            if t < len(targets):
                target_t = targets[t].reshape(-1, 1)
                dy = y_t - target_t

                loss = np.mean(((y_t - target_t) ** 2))
                total_loss += loss

            else:
                dy = y_t - 0

            dW_hy += np.dot(dy, h_t[-1].T)
            db_y += dy

            dh = np.dot(self.W_hy.T, dy) + dh_next
            dh_raw = [np.dot(self.W_hh[i].T, dh) for i in range(self.hidden_size)]
            db_h = [dhr + dh for dhr, dh in zip(dh_raw, dh_next)]

            dW_xh = [dwxh + np.dot(dhr, x_t.T) for dwxh, dhr in zip(dW_xh, dh_raw)]
            dW_hh = [dwhh + np.dot(dhr, h.T) for dwhh, dhr, h in zip(dW_hh, dh_raw, h_t)]
            dh_next = np.dot(self.W_hh[-1].T, dh_raw[-1])

        for i in range(self.hidden_size):
            for dparam in [dW_xh[i], dW_hh[i], db_h[i]]:
                np.clip(dparam, -5, 5, out=dparam)

            self.W_xh[i] -= learning_rate * dW_xh[i]
            self.W_hh[i] -= learning_rate * dW_hh[i]
            self.b_h[i] -= learning_rate * db_h[i]

        for dparam in [dW_hy, db_y]:
            np.clip(dparam, -5, 5, out=dparam)

        self.W_hy -= learning_rate * dW_hy

        self.b_y -= learning_rate * db_y

        average_loss = total_loss / seq_length
        return average_loss

def plot_VanillaRNNMax():
  max_seq_length = max(set([i.shape[0] for i in X_train]))
  input_size = X_train[0].shape[1]
  rnn = VanillaRNNMax(input_size, hidden_size)
  losses = []
```

```python

    for epoch in range(epochs):
        total_loss = 0

        for i in range(len(X_train)):
            inputs = X_train[i]
            targets = y_train[i]
            seq_length = len(X_train[i])
            inputs = np.pad(inputs, ((0, max_seq_length - inputs.shape[0]), (0, 0)), mode='constant')

            hidden_states, outputs = rnn.forward(inputs, max_seq_length)
            loss = rnn.backward(inputs, hidden_states, outputs, targets, seq_length, max_seq_length, learning_rate)
            total_loss += loss

        average_loss = total_loss / len(X_train)
        losses.append(average_loss)

        print(f"Epoch {epoch + 1}/{epochs}, Loss: {average_loss:.4f}")

    plt.plot(range(1, epochs + 1), losses)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training Epoch vs. Loss VanillaRNNMax')
    plt.show()

    total_loss = 0
    for i in range(len(X_test)):
        inputs = X_test[i]
        inputs = np.pad(inputs, ((0, max_seq_length - inputs.shape[0]), (0, 0)), mode='constant')
        targets = y_test[i]
        seq_length = len(inputs)
        _, outputs = rnn.forward(inputs, seq_length)
        total_loss += np.mean(((outputs - targets) ** 2))
    average_loss = total_loss / len(X_test)
    print(f'\nVanillaRNNMax Test Loss: {average_loss:.4f}\n')
    return None

print('Vanilla RNN')
plot_VanillaRNN()
print('Vanilla RNN with Min Sequence Length')
plot_VanillaRNNMin()
print('Vanilla RNN with Max Sequence Length')
plot_VanillaRNNMax()

"""
parameters + output

=============================

hidden_size = 8
epochs = 6
learning_rate = 1e-4

Vanilla RNN
Epoch 1/6, Loss: 4.4596
Epoch 2/6, Loss: 2.1227
Epoch 3/6, Loss: 0.9229
Epoch 4/6, Loss: 0.4005
Epoch 5/6, Loss: 0.1952
Epoch 6/6, Loss: 0.1141
VanillaRNN Test Loss: 0.0877

Vanilla RNN with Min Sequence Length
Epoch 1/6, Loss: 5.1719
Epoch 2/6, Loss: 3.2882
Epoch 3/6, Loss: 2.0529
Epoch 4/6, Loss: 1.2857
Epoch 5/6, Loss: 0.8213
Epoch 6/6, Loss: 0.5385
VanillaRNNMin Test Loss: 0.4455

Vanilla RNN with Max Sequence Length
Epoch 1/6, Loss: 0.4769
Epoch 2/6, Loss: 0.2301
```

```
Epoch 3/6, Loss: 0.0972
Epoch 4/6, Loss: 0.0385
Epoch 5/6, Loss: 0.0161
Epoch 6/6, Loss: 0.0073
VanillaRNNMax Test Loss: 0.0582

=============================

hidden_size = 12
epochs = 8
learning_rate = 3e-6

Vanilla RNN
Epoch 1/8, Loss: 7.9606
Epoch 2/8, Loss: 7.8026
Epoch 3/8, Loss: 7.6467
Epoch 4/8, Loss: 7.4952
Epoch 5/8, Loss: 7.3455
Epoch 6/8, Loss: 7.1980
Epoch 7/8, Loss: 7.0529
Epoch 8/8, Loss: 6.9089
VanillaRNN Test Loss: 6.9403

Vanilla RNN with Min Sequence Length
Epoch 1/8, Loss: 8.6852
Epoch 2/8, Loss: 8.5635
Epoch 3/8, Loss: 8.4432
Epoch 4/8, Loss: 8.3241
Epoch 5/8, Loss: 8.2063
Epoch 6/8, Loss: 8.0898
Epoch 7/8, Loss: 7.9745
Epoch 8/8, Loss: 7.8605
VanillaRNNMin Test Loss: 7.8463

Vanilla RNN with Max Sequence Length
Epoch 1/8, Loss: 0.7396
Epoch 2/8, Loss: 0.7274
Epoch 3/8, Loss: 0.7153
Epoch 4/8, Loss: 0.7034
Epoch 5/8, Loss: 0.6916
Epoch 6/8, Loss: 0.6800
Epoch 7/8, Loss: 0.6685
Epoch 8/8, Loss: 0.6572
VanillaRNNMax Test Loss: 9.2305

=============================

hidden_size = 8
epochs = 6
learning_rate = 3e-4

Vanilla RNN
Epoch 1/6, Loss: 1.5473
Epoch 2/6, Loss: 0.1255
Epoch 3/6, Loss: 0.0368
Epoch 4/6, Loss: 0.0230
Epoch 5/6, Loss: 0.0198
Epoch 6/6, Loss: 0.0189
VanillaRNN Test Loss: 0.0164

Vanilla RNN with Min Sequence Length
Epoch 1/6, Loss: 2.0644
Epoch 2/6, Loss: 0.5714
Epoch 3/6, Loss: 0.1979
Epoch 4/6, Loss: 0.0844
Epoch 5/6, Loss: 0.0442
Epoch 6/6, Loss: 0.0291
VanillaRNNMin Test Loss: 0.0227

Vanilla RNN with Max Sequence Length
Epoch 1/6, Loss: 0.2056
Epoch 2/6, Loss: 0.0158
Epoch 3/6, Loss: 0.0029
```

```
Epoch 4/6, Loss: 0.0019
Epoch 5/6, Loss: 0.0017
Epoch 6/6, Loss: 0.0017
VanillaRNNMax Test Loss: 0.0163



c) Analyze the results and discuss the advantages and disadvantages of each approach in terms of modeling sequences with varying lengths.

=> Based on the above results we can conclude the following based on the 3 architectures.

Vanilla RNN

    Advantages:
        This architecture achieved the lowest test score with hidden_size as 8, epochs as 6 and learning_rate as 3e-4 and highest with hidden_size as 12, epochs as 8, learning_rate as 3e-6.
        It was slightly faster in traning as compared to the rest as the weights were being shared across time steps.
        It is relatively a simple model and achieved good results for the mentioned parameters.
        The sequence length was not padded or truncated during training.
        It requires less computational resources compared to Vanilla RNN with Max Sequence Length architecture.
        This network would be a good choice if we have sequences of varying lengths and need a balance between performance and efficiency.

    Disadvantages:
        The test loss, while low, may still not be sufficient for some applications, depending on the specific problem being solved.

Vanilla RNN with Min Sequence Length

    Advantages:
        This architecture achieved the lowest test score with hidden_size as 8, epochs as 6 and learning_rate as 3e-4 and highest with hidden_size as 12, epochs as 8, learning_rate as 3e-6.
        It was slightly slower in traning as compared to the Vanilla RNN as the weights were not being shared across time steps.
        It has higher test losses compared to Vanilla RNN with the same hidden size.
        The sequence length was truncated during training (minimum length).
        Although it was truncated to minimum length still it provides a reasonable performance considering the sequence length and loss of data.
        This model might be more efficient when working with short sequences.
        It requires less computational resources compared to Vanilla RNN with Max Sequence Length architecture.
        This network would be a good choice if we have short sequences and need a balance between performance and efficiency.


    Disadvantages:
        The test loss is higher than Vanilla RNN for the same hidden size, suggesting that it might struggle with longer sequences.

Vanilla RNN with Max Sequence Length

    Advantages:
        This architecture achieved the lowest test score with hidden_size as 8, epochs as 6 and learning_rate as 3e-4 and highest with hidden_size as 12, epochs as 8, learning_rate as 3e-6.
        It was slightly slower in traning as compared to the Vanilla RNN as the weights were not being shared across time steps.
        It has higher test losses compared to Vanilla RNN with the same hidden size.
        The sequence length was padded during training (maximum length).
        Although it was padded to maximum sequence length still it provides a reasonable performance considering the zero padding.
        This model might be more efficient when working with long sequences and requiring long-term dependencies.
        This network would be a good choice if we have long sequences and need accuracy over efficiency.

    Disadvantages:
        The test loss for shorter sequences is considerably higher, suggesting that it might not generalize well to shorter sequences possibly to the padding.
        It requires more computational resources compared to Vanilla RNN with Min Sequence Length architecture.

"""





# ============================
# problem_set_3
# ============================

start_token = 'sos'
end_token = 'eos'
BATCH_SIZE = 32
EPOCHS = 8
GRU_UNITS = 256

def txt_pre_processing(txt:str)->str:
  txt = txt.lower().strip()
  txt = unicodedata.normalize('NFKD',txt).encode('ascii', 'ignore').decode('utf-8')
  txt = re.sub(pattern=r'[^\sa-z\d\.\?\!\,]',repl='',string=str(txt))
```

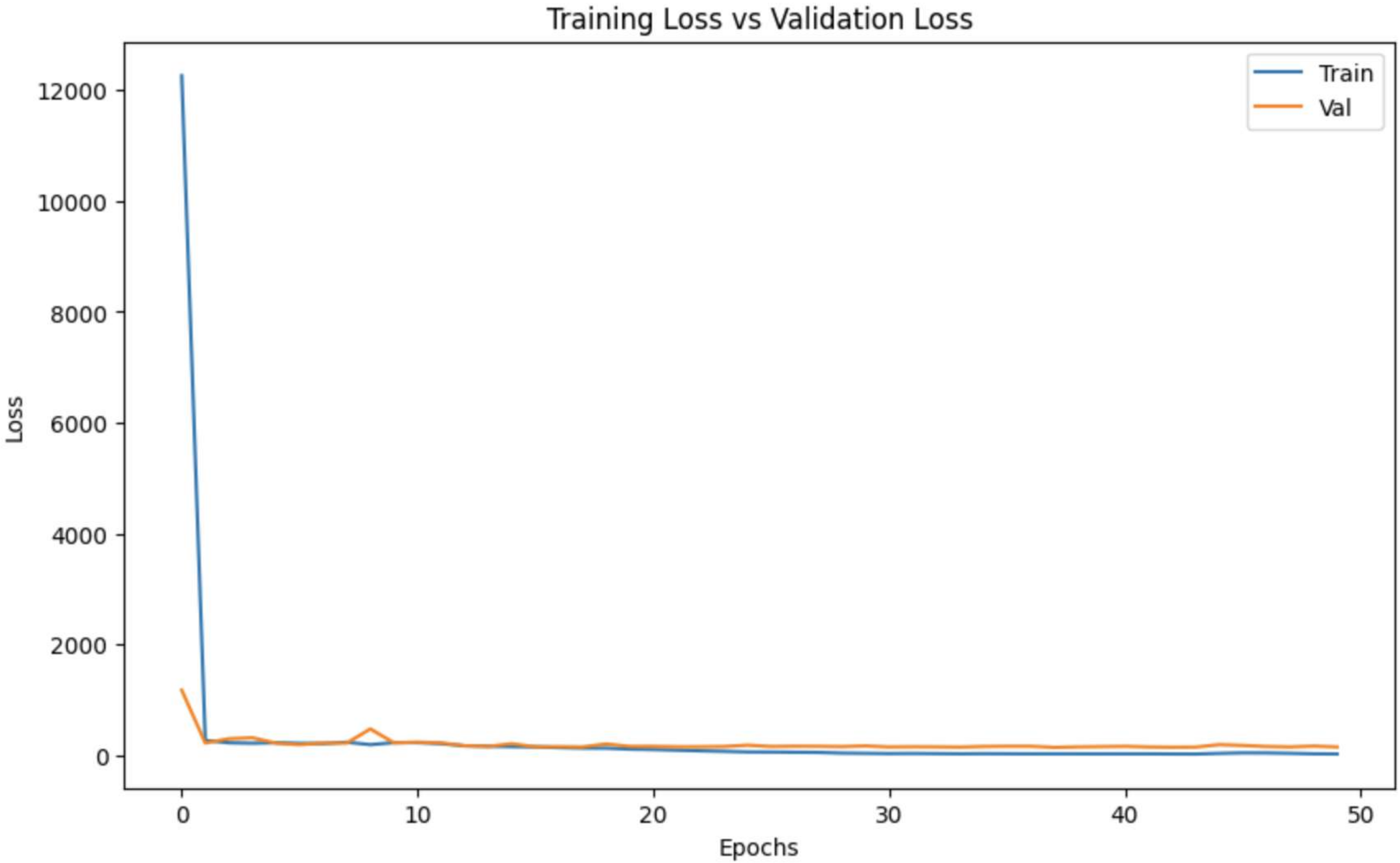
```
  txt = re.sub(pattern=r'([\.\?\!\,])',repl=r' \1 ',string=str(txt)).strip()
  txt = start_token + ' ' + txt + ' ' + end_token
  return txt

def load_data() -> tuple:
  context : list = list()
  target : list = list()
  with open(file='./eng-fra.txt',mode='r',encoding='utf-8') as inputstream:
    for text in inputstream:
      lines = text.replace('\n','').replace('\r','').split('\t')
      eng_txt = lines[0]
      fr_txt = lines[1]
      eng_txt = txt_pre_processing(txt=eng_txt)
      fr_txt = txt_pre_processing(txt=fr_txt)
      context.append(eng_txt)
      target.append(fr_txt)
  context = np.array(context)
  target = np.array(target)
  return context,target

eng_sentences, fr_sentences = load_data()
shuffling_indices = np.arange(len(eng_sentences))
np.random.shuffle(shuffling_indices)
eng_sentences = eng_sentences[shuffling_indices]
fr_sentences = fr_sentences[shuffling_indices]

eng_tokenizer = Tokenizer()
eng_tokenizer.fit_on_texts(eng_sentences)
eng_vocab_size = len(eng_tokenizer.word_index) + 1

fr_tokenizer = Tokenizer()
fr_tokenizer.fit_on_texts(fr_sentences)
fr_vocab_size = len(fr_tokenizer.word_index) + 1

eng_sequences = eng_tokenizer.texts_to_sequences(eng_sentences)
fr_sequences = fr_tokenizer.texts_to_sequences(fr_sentences)

max_seq_length = 52
eng_sequences = pad_sequences(eng_sequences, maxlen=max_seq_length, padding='post')
fr_sequences = pad_sequences(fr_sequences, maxlen=max_seq_length, padding='post')
split_80_20: int = int(eng_sequences.shape[0]*0.8)
X_train, y_train = eng_sequences[:split_80_20,:], fr_sequences[:split_80_20]
X_test, y_test = eng_sequences[split_80_20:,:], fr_sequences[split_80_20:]
y_train = to_categorical(y_train, num_classes=fr_vocab_size)
y_test = to_categorical(y_test, num_classes=fr_vocab_size)

# ============================
# enc + dec
# ============================

encoder_inputs = tf.keras.layers.Input(shape=(None,))
encoder_embedding = Embedding(input_dim=eng_vocab_size, output_dim=GRU_UNITS)(encoder_inputs)
encoder_gru = GRU(GRU_UNITS, return_state=True)
encoder_outputs, encoder_state = encoder_gru(encoder_embedding)
decoder_inputs = tf.keras.layers.Input(shape=(None,))
decoder_embedding = Embedding(input_dim=fr_vocab_size, output_dim=GRU_UNITS)(decoder_inputs)
decoder_gru = GRU(GRU_UNITS, return_sequences=True, return_state=True)
decoder_outputs, _ = decoder_gru(decoder_embedding, initial_state=encoder_state)
decoder_dense = Dense(fr_vocab_size, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit([X_train, X_train], y_train, epochs=EPOCHS, batch_size=BATCH_SIZE, validation_data=([X_test, X_test], y_test),callbacks=[EarlyStopping(patience=6)])
del model,X_train,y_train,X_test,y_test
fig, axs = plt.subplots(2, 1, figsize=(10,13))
axs[0].plot(history.history['loss'])
axs[0].plot(history.history['val_loss'])
axs[0].title.set_text('Enc + Dec Training Loss vs Validation Loss')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Loss')
axs[0].legend(['Train','Val'])
axs[1].plot(history.history['accuracy'])
axs[1].plot(history.history['val_accuracy'])
```

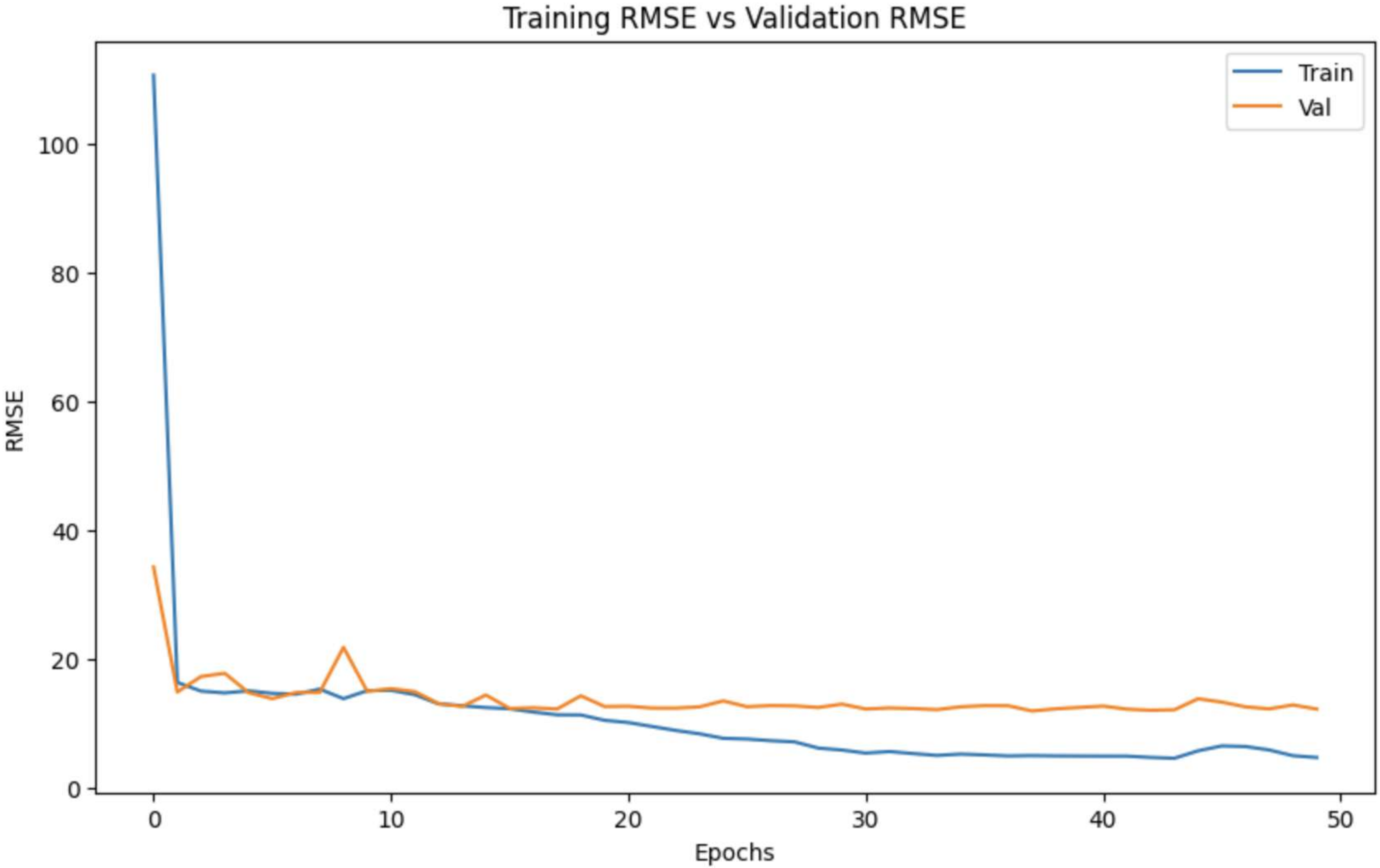
```
axs[1].title.set_text('Enc + Dec Training Accuracy vs Validation Accuracy')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('Accuracy')
axs[1].legend(['Train', 'Val'])

encoder_model = Model(encoder_inputs, encoder_state)
decoder_state_input = Input(shape=(GRU_UNITS,))
decoder_inputs = Input(shape=(1,))
decoder_embedding_inference = Embedding(input_dim=fr_vocab_size, output_dim=GRU_UNITS)(decoder_inputs)
decoder_gru_inference = GRU(GRU_UNITS, return_sequences=True, return_state=True)
decoder_outputs_inference, decoder_state_inference = decoder_gru_inference(decoder_embedding_inference, initial_state=decoder_state_input)
decoder_outputs_inference = decoder_dense(decoder_outputs_inference)
decoder_model = Model([decoder_inputs, decoder_state_input], [decoder_outputs_inference, decoder_state_inference])

def translate_sentence(input_text):
    stop_crit = len(input_text)+3
    input_text = txt_pre_processing(txt=input_text)
    input_seq = eng_tokenizer.texts_to_sequences([input_text])
    input_seq = pad_sequences(input_seq, maxlen=max_seq_length, padding='post')
    input_seq = tf.ragged.constant(input_seq)
    states_value = encoder_model.predict(input_seq)

    target_seq = tf.constant([fr_tokenizer.word_index[start_token]])
    target_text = []
    stop_condition = False
    prev_token_index = None

    while not stop_condition:
      output_tokens, h = decoder_model.predict([target_seq, states_value])
      sampled_token_index = np.argmax(output_tokens[0, -1, :])
      if (sampled_token_index == 0):
        sampled_word = ''

      else:
        sampled_word = fr_tokenizer.index_word[sampled_token_index]

      if (sampled_word != end_token) and (sampled_word != ''):
          target_text.append(sampled_word)

      if sampled_word == end_token or len(target_text) >= stop_crit:
          stop_condition = True

      prev_token_index = sampled_token_index
      target_seq = tf.constant([sampled_token_index])
      states_value = h
    return ' '.join(target_text)
input_text = "I won!"
translation = translate_sentence(input_text)
del decoder_model

# ============================
# only enc
# ============================

autoencoder_inputs = tf.keras.layers.Input(shape=(max_seq_length,))
autoencoder_embedding = Embedding(input_dim=eng_vocab_size, output_dim=GRU_UNITS)(autoencoder_inputs)
autoencoder_gru = GRU(GRU_UNITS, return_state=True)
autoencoder_outputs, autoencoder_state = autoencoder_gru(autoencoder_embedding)
autoencoder_outputs = RepeatVector(max_seq_length)(autoencoder_outputs)
autoencoder_gru = GRU(GRU_UNITS, return_sequences=True, return_state=True)
autoencoder_outputs, _ = autoencoder_gru(autoencoder_outputs, initial_state=autoencoder_state)
autoencoder_dense = Dense(eng_vocab_size, activation='softmax')
autoencoder_outputs = autoencoder_dense(autoencoder_outputs)
autoencoder_model = Model(autoencoder_inputs, autoencoder_outputs)
autoencoder_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
split_80_20: int = int(eng_sequences.shape[0]*0.8)
X_train = eng_sequences[:split_80_20,:]
X_test = eng_sequences[split_80_20:,:]
y_train_autoencoder = to_categorical(X_train, num_classes=eng_vocab_size)
y_test_autoencoder = to_categorical(X_test, num_classes=eng_vocab_size)

f_loss = list()
f_acc = list()
f_val_loss = list()
f_val_acc = list()
```

```python

for epoch in range(EPOCHS):
    history = autoencoder_model.fit(X_train, y_train_autoencoder, epochs=1, batch_size=BATCH_SIZE, verbose=1)
    val_loss, val_accuracy = autoencoder_model.evaluate(X_test, y_test_autoencoder, batch_size=BATCH_SIZE, verbose=1)
    f_loss.append(history.history['loss'])
    f_acc.append(history.history['accuracy'])
    f_val_loss.append(val_loss)
    f_val_acc.append(val_accuracy)

del autoencoder_model,X_train,y_train_autoencoder,X_test,y_test_autoencoder
fig, axs = plt.subplots(2, 1, figsize=(10,13))
axs[0].plot(f_loss)
axs[0].plot(f_val_loss)
axs[0].title.set_text('Encoder Only Training Loss vs Validation Loss')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Loss')
axs[0].legend(['Train','Val'])
axs[1].plot(f_acc)
axs[1].plot(f_val_acc)
axs[1].title.set_text('Encoder Only Training Accuracy vs Validation Accuracy')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('Accuracy')
axs[1].legend(['Train', 'Val'])

# ============================
# save only enc
# ============================

pretrained_encoder_model = Model(autoencoder_inputs, autoencoder_state)

for layer in pretrained_encoder_model.layers:
    layer.trainable = False

pretrained_encoder_model.save_weights('pretrained_encoder_model_weights.h5')
pretrained_encoder_model_json = pretrained_encoder_model.to_json()
with open(file='pretrained_encoder_model.json',mode='w') as json_file:
    json_file.write(pretrained_encoder_model_json)
del pretrained_encoder_model_json

with open(file='pretrained_encoder_model.json',mode='r') as json_file:
    pretrained_encoder_model_json = json_file.read()
pretrained_encoder_model = model_from_json(pretrained_encoder_model_json)
pretrained_encoder_model.load_weights('pretrained_encoder_model_weights.h5')

encoder_model = Model(autoencoder_inputs, autoencoder_state)
decoder_state_input = Input(shape=(GRU_UNITS,))
decoder_inputs = Input(shape=(1,))
decoder_embedding_inference = Embedding(input_dim=fr_vocab_size, output_dim=GRU_UNITS)(decoder_inputs)
decoder_gru_inference = GRU(GRU_UNITS, return_sequences=True, return_state=True)
decoder_outputs_inference, decoder_state_inference = decoder_gru_inference(decoder_embedding_inference, initial_state=decoder_state_input)
decoder_outputs_inference = decoder_dense(decoder_outputs_inference)
decoder_model = Model([decoder_inputs, decoder_state_input], [decoder_outputs_inference, decoder_state_inference])

def translate_sentence(input_text):
    stop_crit = len(input_text)+3
    input_text = txt_pre_processing(txt=input_text)
    input_seq = eng_tokenizer.texts_to_sequences([input_text])
    input_seq = pad_sequences(input_seq, maxlen=max_seq_length, padding='post')
    input_seq = tf.ragged.constant(input_seq)
    states_value = encoder_model.predict(input_seq)

    target_seq = tf.constant([fr_tokenizer.word_index[start_token]])
    target_text = []
    stop_condition = False
    prev_token_index = None

    while not stop_condition:
      output_tokens, h = decoder_model.predict([target_seq, states_value])
      sampled_token_index = np.argmax(output_tokens[0, -1, :])
      if (sampled_token_index == 0):
        sampled_word = ''
      else:
        sampled_word = fr_tokenizer.index_word[sampled_token_index]

```

```
      if (sampled_word != end_token) and (sampled_word != ''):
          target_text.append(sampled_word)

      if sampled_word == end_token or len(target_text) >= stop_crit:
          stop_condition = True

      prev_token_index = sampled_token_index
      target_seq = tf.constant([sampled_token_index])
      states_value = h
    return ' '.join(target_text)
input_text = "I won!"
translation = translate_sentence(input_text)
del decoder_model

# ============================
# only dec
# ============================

translation_decoder_inputs = tf.keras.layers.Input(shape=(None,))
decoder_embedding = Embedding(input_dim=fr_vocab_size, output_dim=GRU_UNITS)(translation_decoder_inputs)
decoder_gru = GRU(GRU_UNITS, return_sequences=True, return_state=True)
decoder_outputs, _ = decoder_gru(decoder_embedding, initial_state=pretrained_encoder_model.output)
decoder_dense = Dense(fr_vocab_size, activation='softmax')
translation_decoder_outputs = decoder_dense(decoder_outputs)
translation_decoder_model = Model(inputs=[pretrained_encoder_model.input, translation_decoder_inputs], outputs=translation_decoder_outputs)
translation_decoder_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
split_80_20: int = int(eng_sequences.shape[0]*0.8)
X_train, y_train = eng_sequences[:split_80_20,:], fr_sequences[:split_80_20]
X_test, y_test = eng_sequences[split_80_20:,:], fr_sequences[split_80_20:]
y_train = to_categorical(y_train, num_classes=fr_vocab_size)
y_test = to_categorical(y_test, num_classes=fr_vocab_size)
history = translation_decoder_model.fit([X_train, X_train], y_train, epochs=EPOCHS, batch_size=BATCH_SIZE, validation_data=([X_test, X_test], y_test),callbacks=[EarlyStopping(patience=6)])
del translation_decoder_model,X_train,y_train,X_test,y_test
fig, axs = plt.subplots(2, 1, figsize=(10,13))
axs[0].plot(history.history['loss'])
axs[0].plot(history.history['val_loss'])
axs[0].title.set_text('Decoder Only Training Loss vs Validation Loss')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Loss')
axs[0].legend(['Train','Val'])
axs[1].plot(history.history['accuracy'])
axs[1].plot(history.history['val_accuracy'])
axs[1].title.set_text('Decoder Only Training Accuracy vs Validation Accuracy')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('Accuracy')
axs[1].legend(['Train', 'Val'])

encoder_model = Model(autoencoder_inputs, autoencoder_state)
decoder_state_input = Input(shape=(GRU_UNITS,))
decoder_inputs = Input(shape=(1,))
decoder_embedding_inference = Embedding(input_dim=fr_vocab_size, output_dim=GRU_UNITS)(decoder_inputs)
decoder_gru_inference = GRU(GRU_UNITS, return_sequences=True, return_state=True)
decoder_outputs_inference, decoder_state_inference = decoder_gru_inference(decoder_embedding_inference, initial_state=decoder_state_input)
decoder_outputs_inference = decoder_dense(decoder_outputs_inference)
decoder_model = Model([decoder_inputs, decoder_state_input], [decoder_outputs_inference, decoder_state_inference])

def translate_sentence(input_text):
    stop_crit = len(input_text)+3
    input_text = txt_pre_processing(txt=input_text)
    input_seq = eng_tokenizer.texts_to_sequences([input_text])
    input_seq = pad_sequences(input_seq, maxlen=max_seq_length, padding='post')
    input_seq = tf.ragged.constant(input_seq)
    states_value = encoder_model.predict(input_seq)

    target_seq = tf.constant([fr_tokenizer.word_index[start_token]])
    target_text = []
    stop_condition = False
    prev_token_index = None

    while not stop_condition:
      output_tokens, h = decoder_model.predict([target_seq, states_value])
      sampled_token_index = np.argmax(output_tokens[0, -1, :])
      if (sampled_token_index == 0):
        sampled_word = ''
```

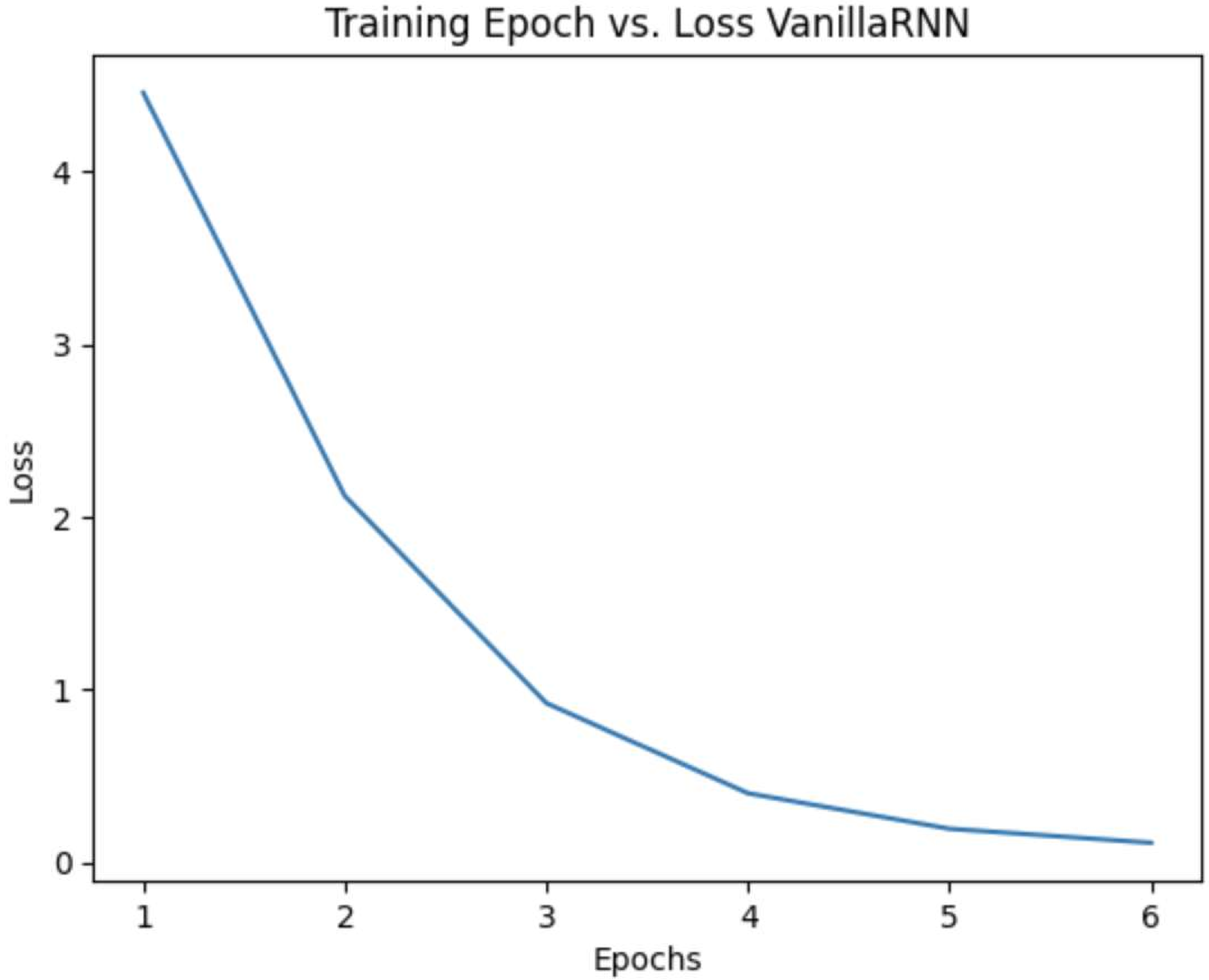
```
        sampled_word =
      else:
        sampled_word = fr_tokenizer.index_word[sampled_token_index]

      if (sampled_word != end_token) and (sampled_word != ''):
          target_text.append(sampled_word)

      if sampled_word == end_token or len(target_text) >= stop_crit:
          stop_condition = True

      prev_token_index = sampled_token_index
      target_seq = tf.constant([sampled_token_index])
      states_value = h
    return ' '.join(target_text)
input_text = "I won!"
translation = translate_sentence(input_text)
del decoder_model
```

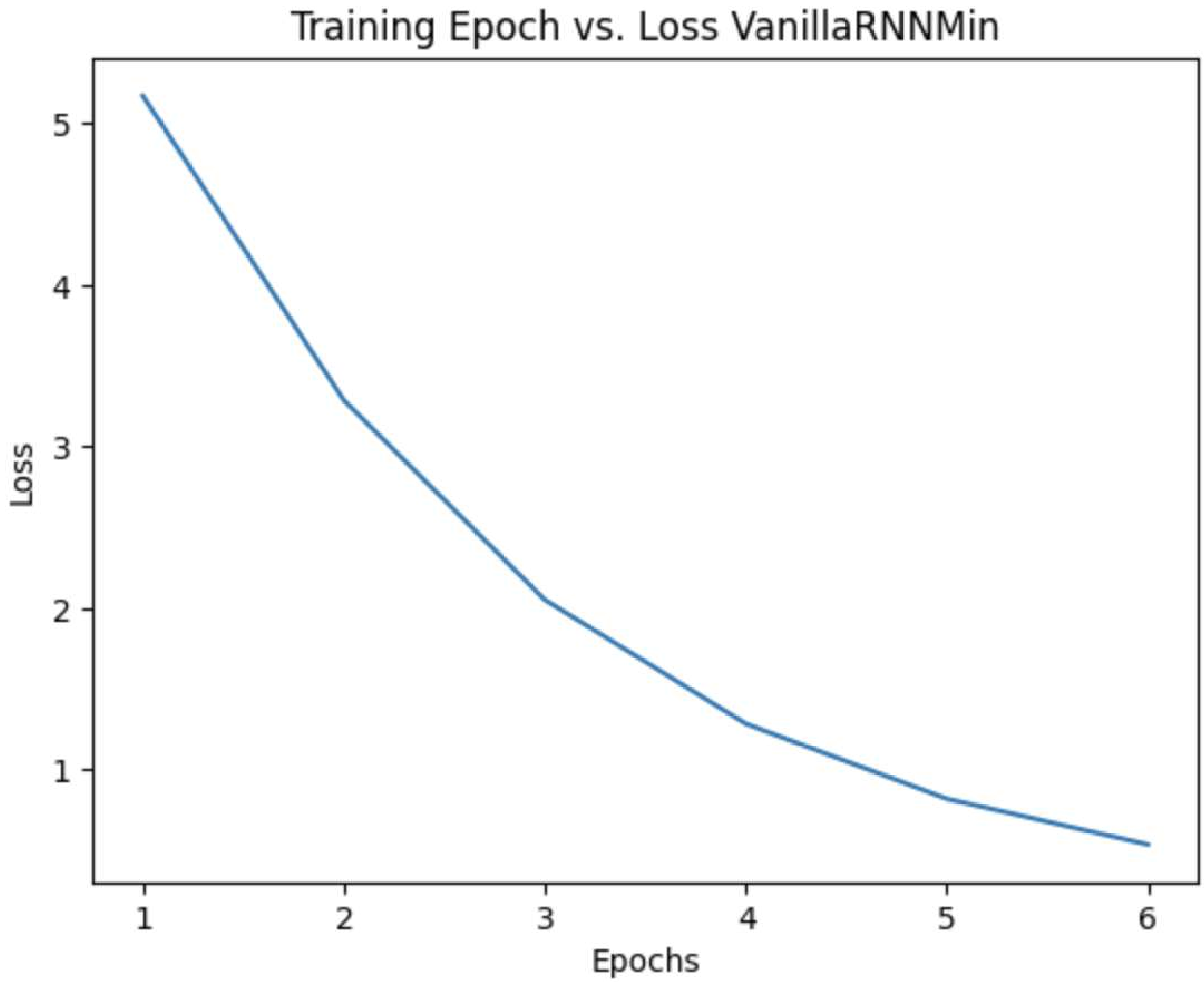

Training Loss vs Validation Loss
Train
Val
Loss
Epochs
12000
10000
8000
6000
4000
2000
0
0
10
20
30
40
50

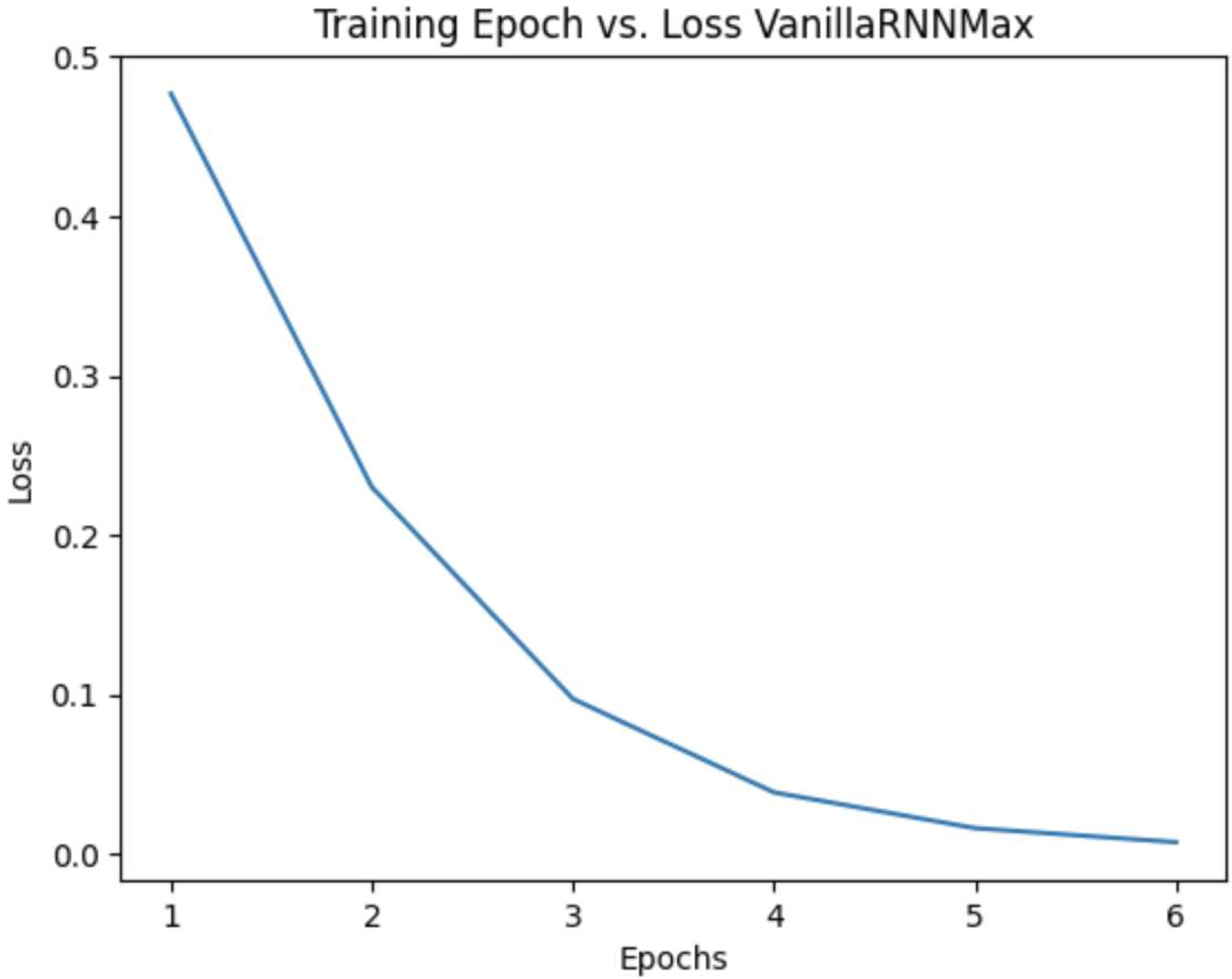


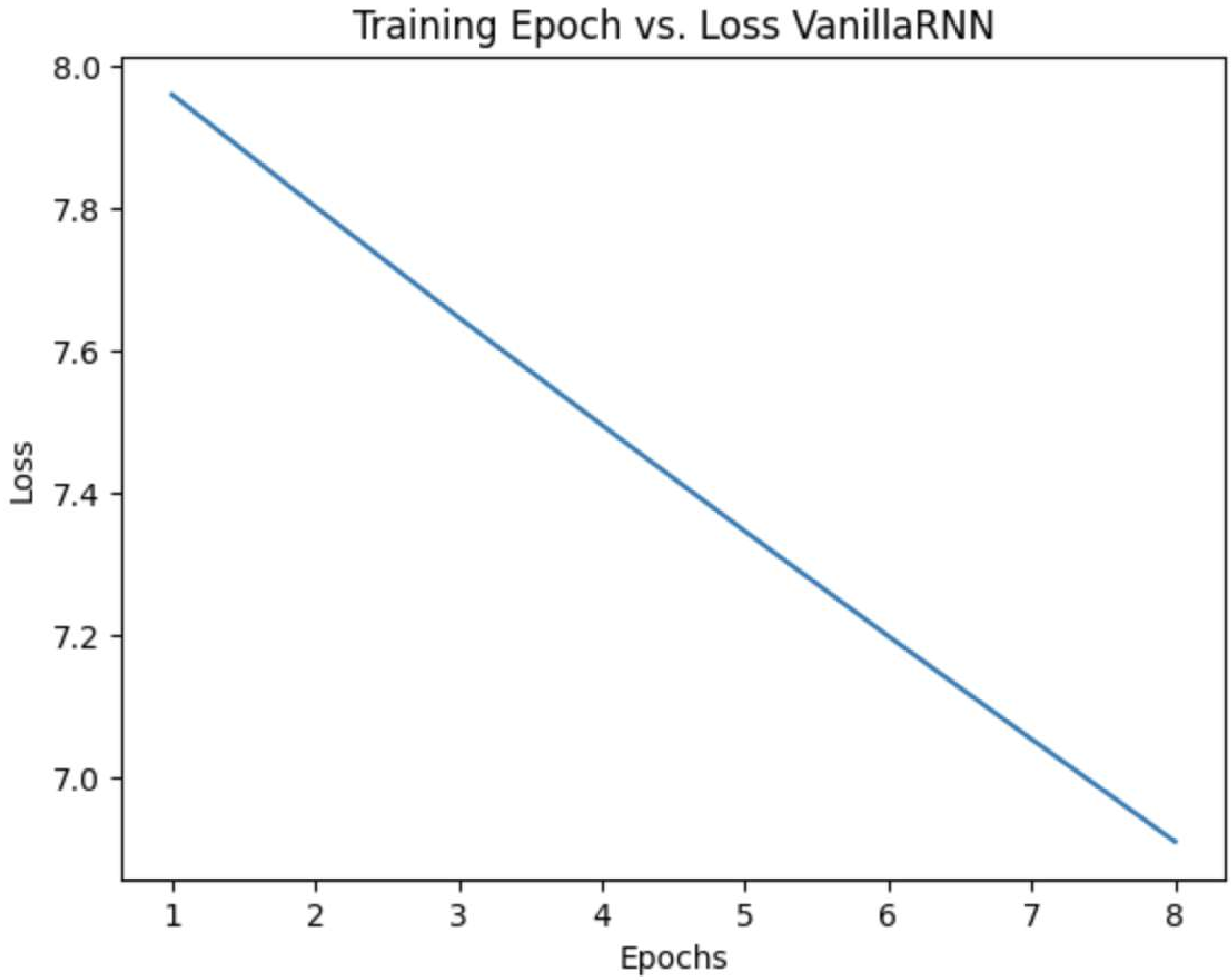
Training RMSE vs Validation RMSE
Train
Val
RMSE
Epochs
0
20
40
60
80
100
0
10
20
30
40
50

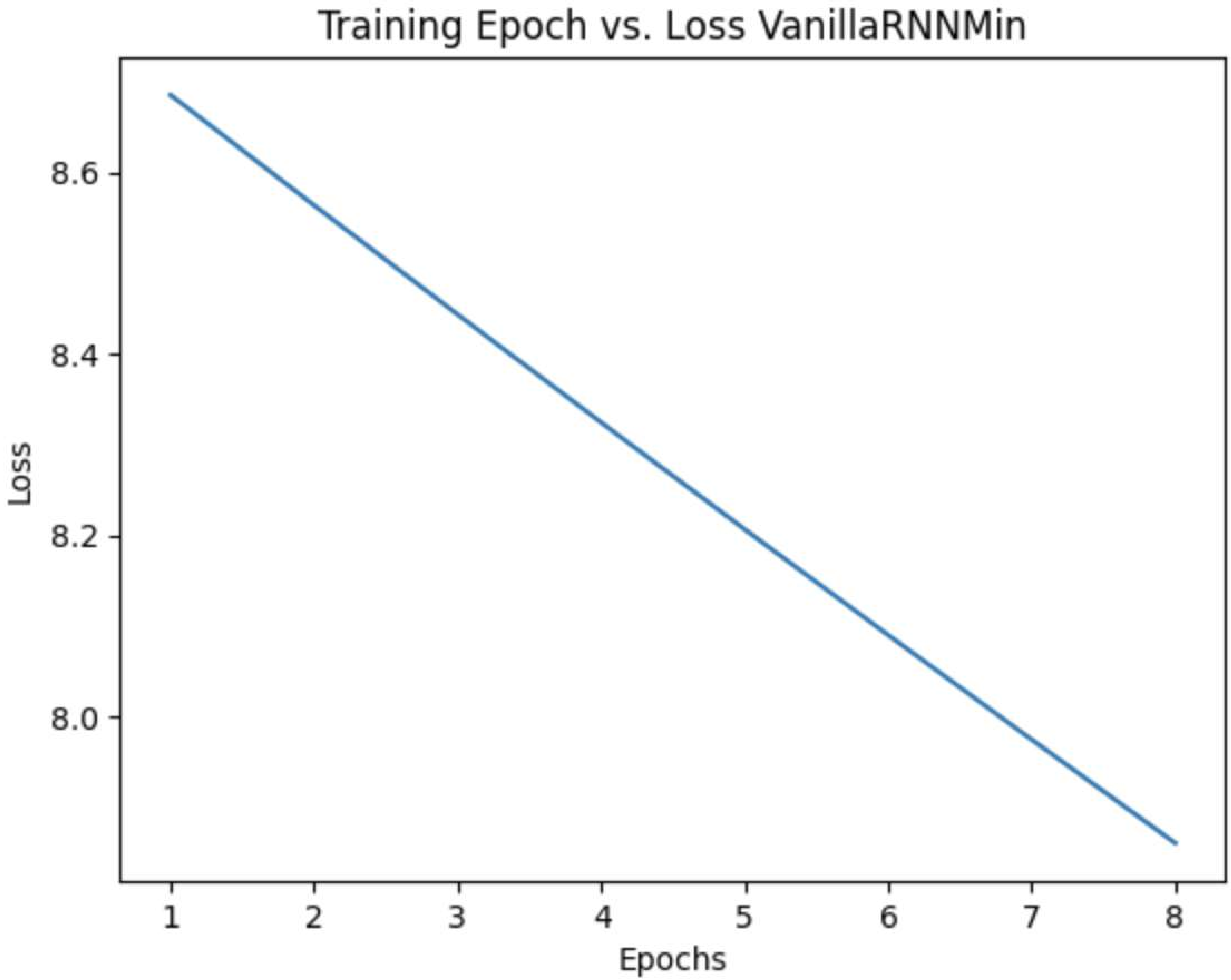
```
Epoch 50/50
165/165 [==============================] - 7s 44ms/step - loss: 21.4251 - root_mean_squared_error: 4.6287 - val_loss: 147.6750 - val_root_mean_squared_error: 12.1522
47/47 [==============================] - 1s 12ms/step - loss: 141.9804 - root_mean_squared_error: 11.9156
Test Loss: 141.9804, Test RMSE: 11.9156
<matplotlib.legend.Legend at 0x78326a3d8a00>
```

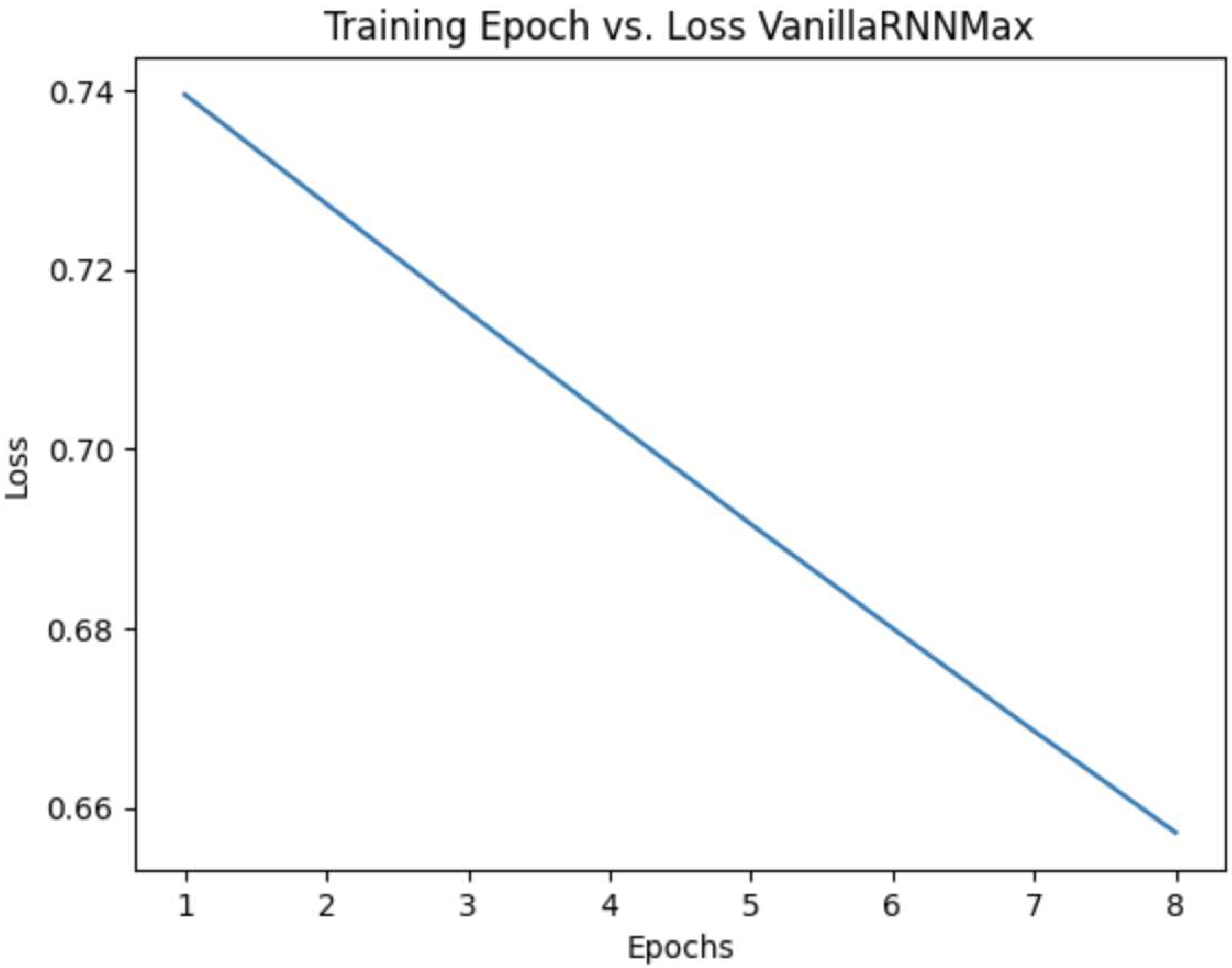

Training Epoch vs. Loss VanillaRNN
Loss
Epochs
0
1
2
3
4
1
2
3
4
5
6

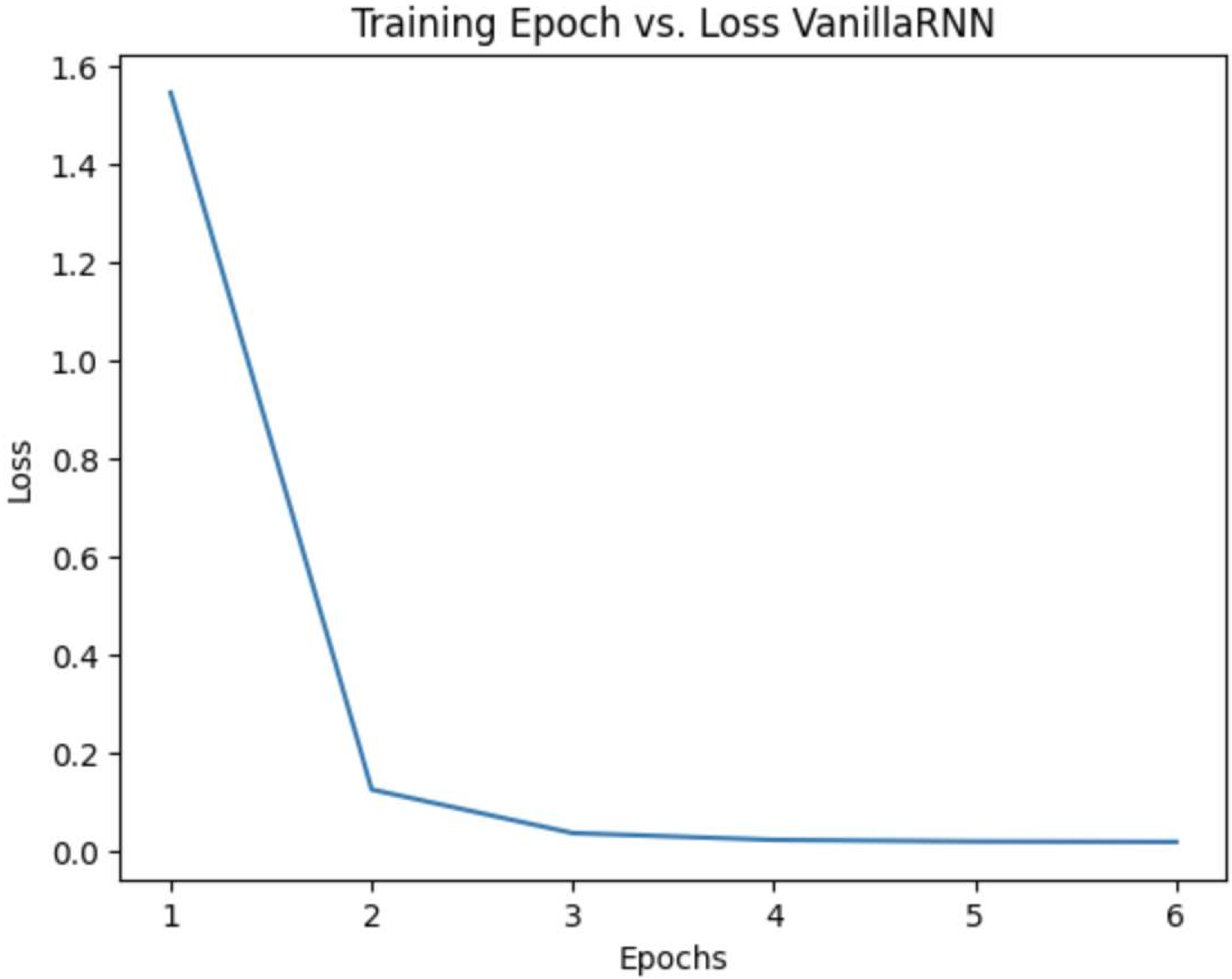

Training Epoch vs. Loss VanillaRNNMin
Loss
5
4
3
2
1
1
2
3
4
5
6
Epochs

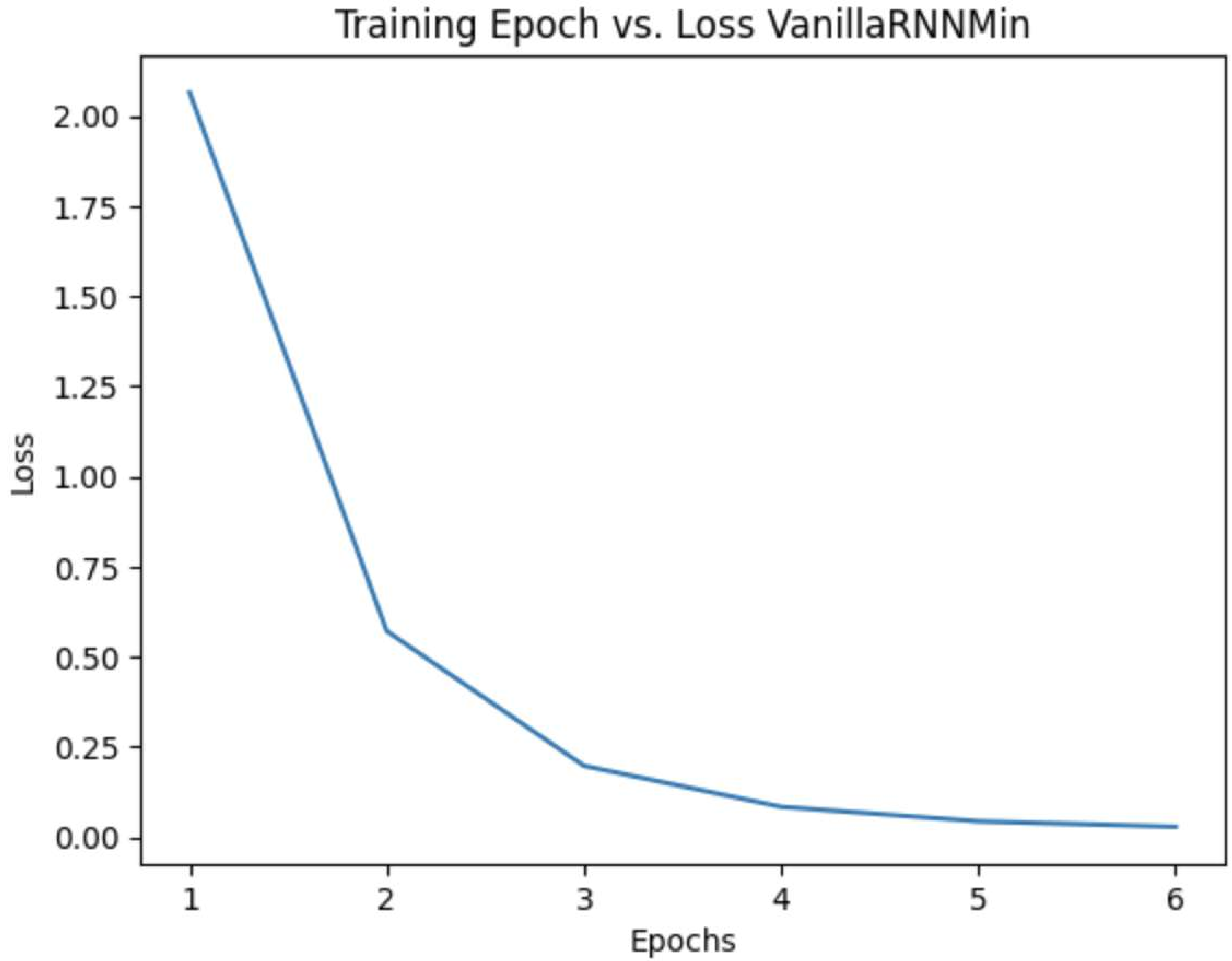

Training Epoch vs. Loss VanillaRNNMax
Loss
Epochs
0.0
0.1
0.2
0.3
0.4
0.5
1
2
3
4
5
6


Training Epoch vs. Loss VanillaRNN
Loss
Epochs
8.0
7.8
7.6
7.4
7.2
7.0
1
2
3
4
5
6
7
8

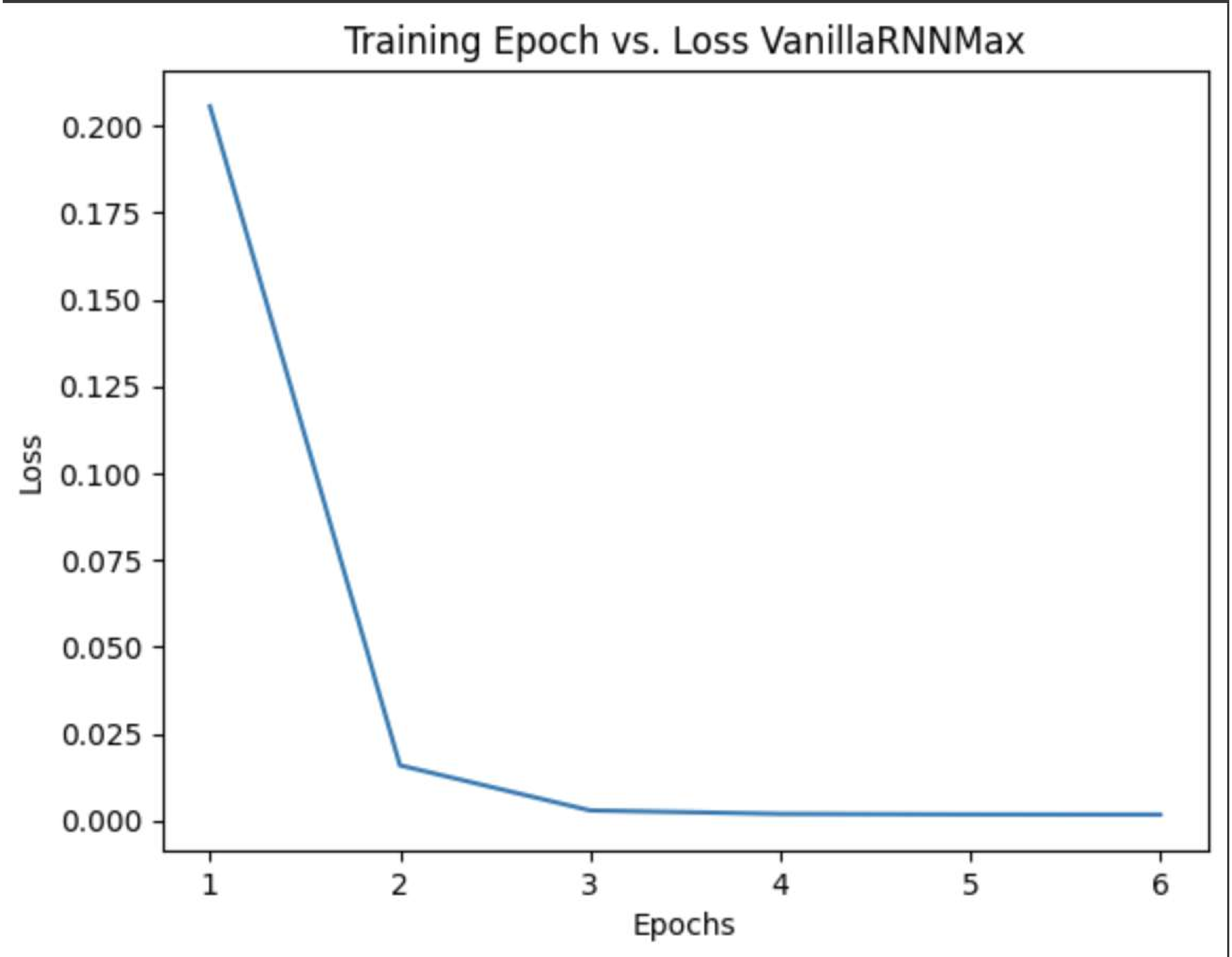

Training Epoch vs. Loss VanillaRNNMin
Loss
8.6
8.4
8.2
8.0
1
2
3
4
5
6
7
8
Epochs


Training Epoch vs. Loss VanillaRNNMax
Loss
0.74
0.72
0.70
0.68
0.66
Epochs
1
2
3
4
5
6
7
8

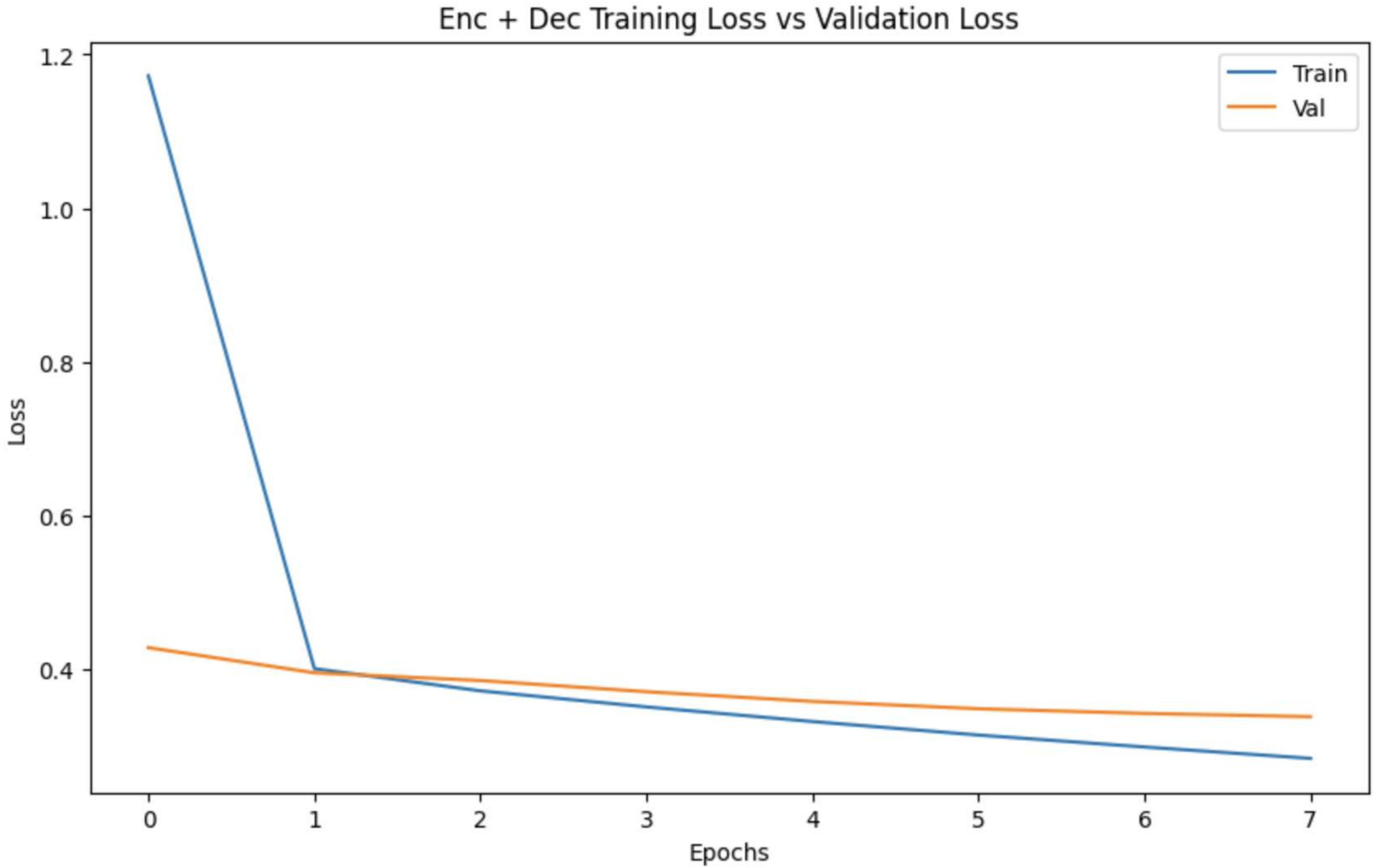

Training Epoch vs. Loss VanillaRNN
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Loss
1
2
3
4
5
6
Epochs



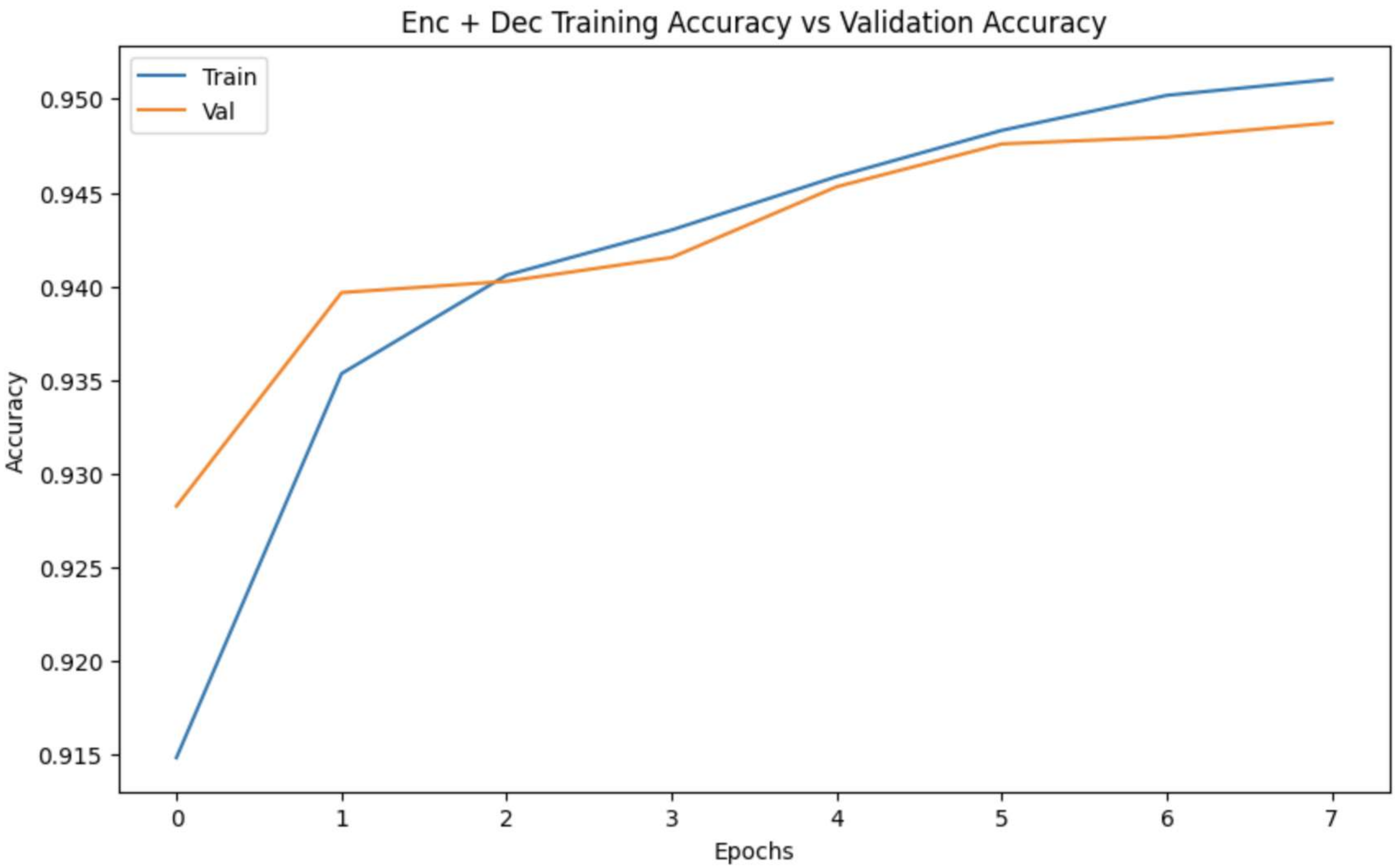
Training Epoch vs. Loss VanillaRNNMin
Loss
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1
2
3
4
5
6
Epochs

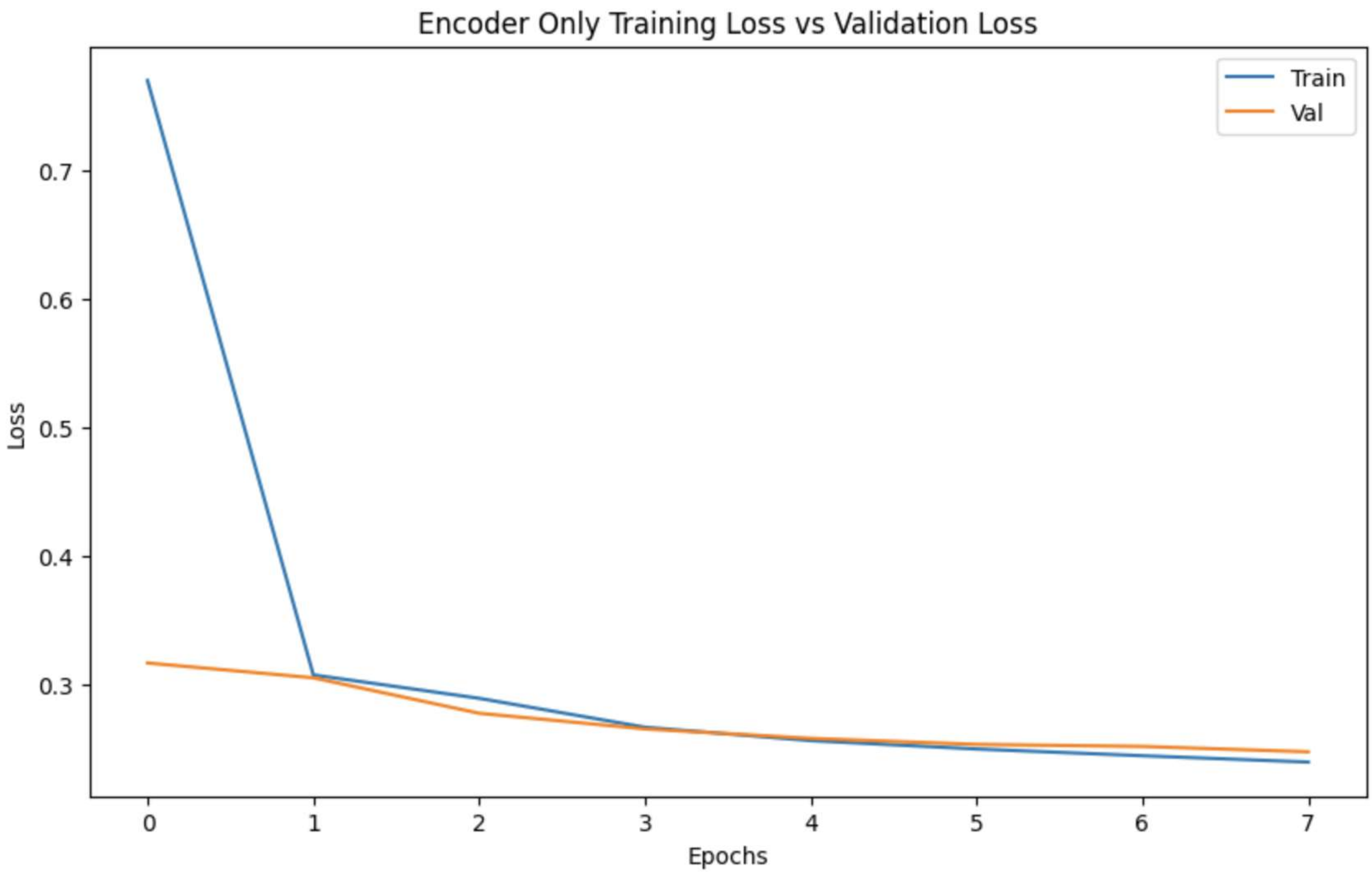


Training Epoch vs. Loss VanillaRNNMax
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
Loss
1
2
3
4
5
6
Epochs

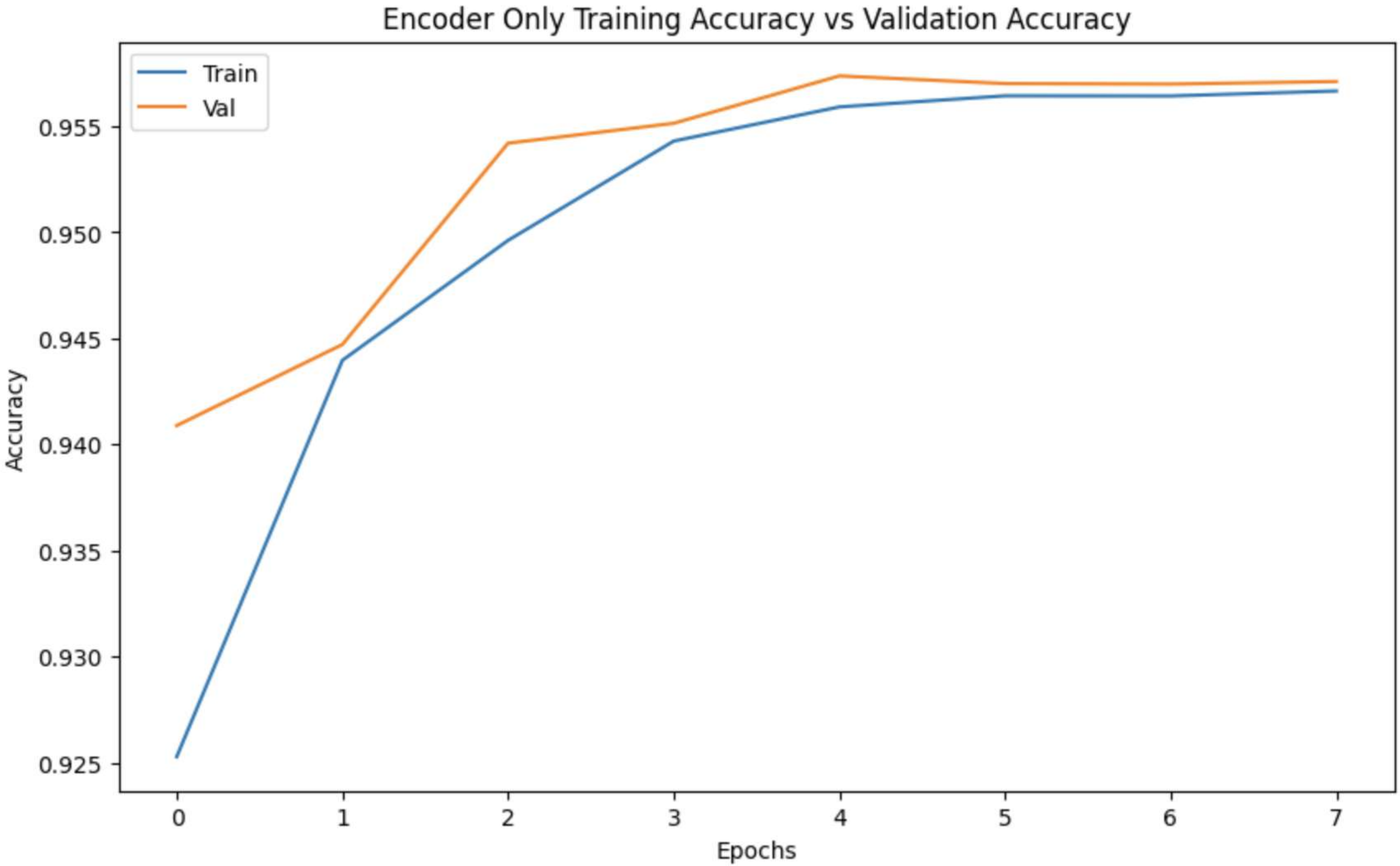


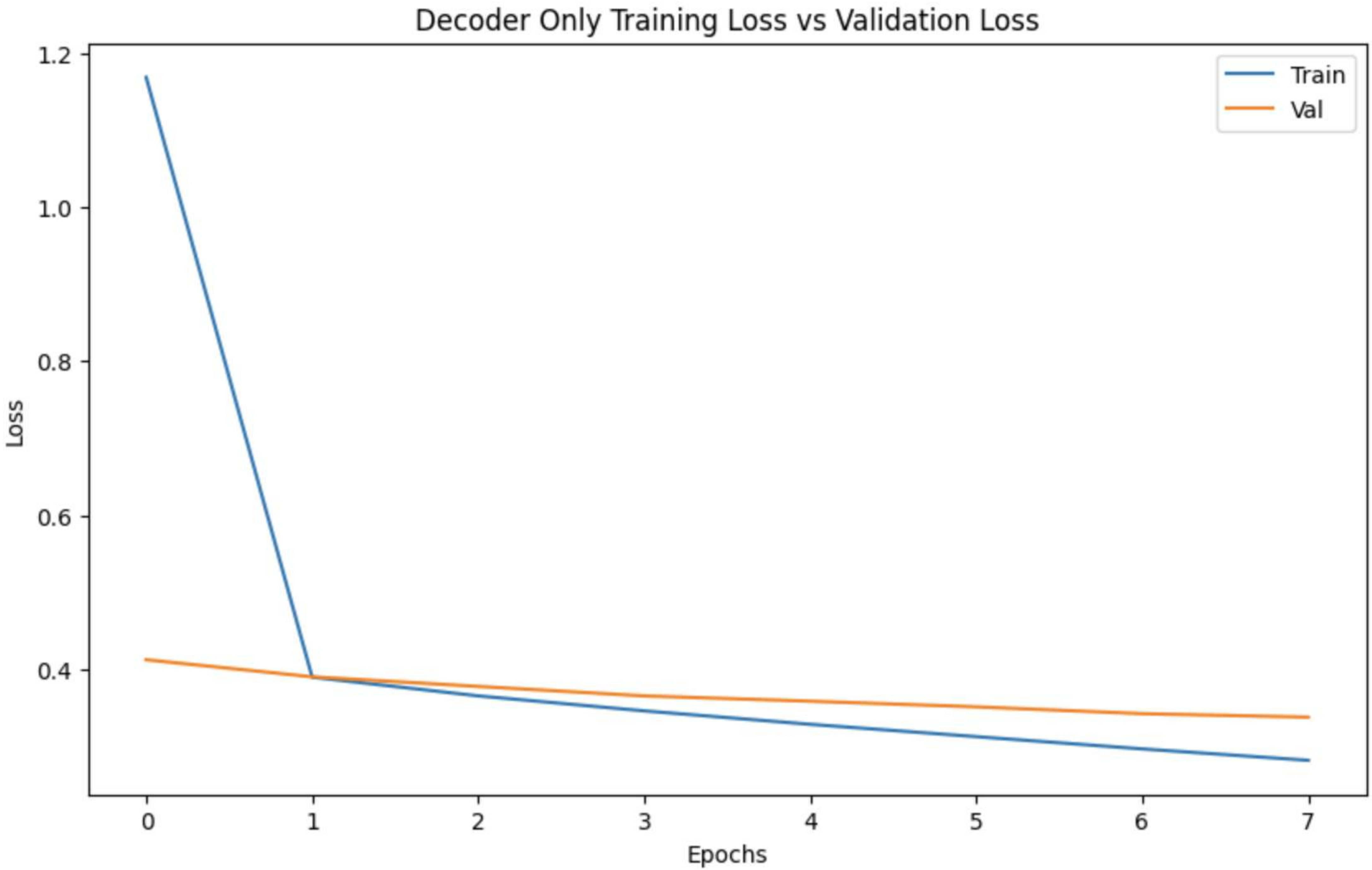
Enc + Dec Training Loss vs Validation Loss
Train
Val
Loss
Epochs
1.2
1.0
0.8
0.6
0.4
0
1
2
3
4
5
6
7

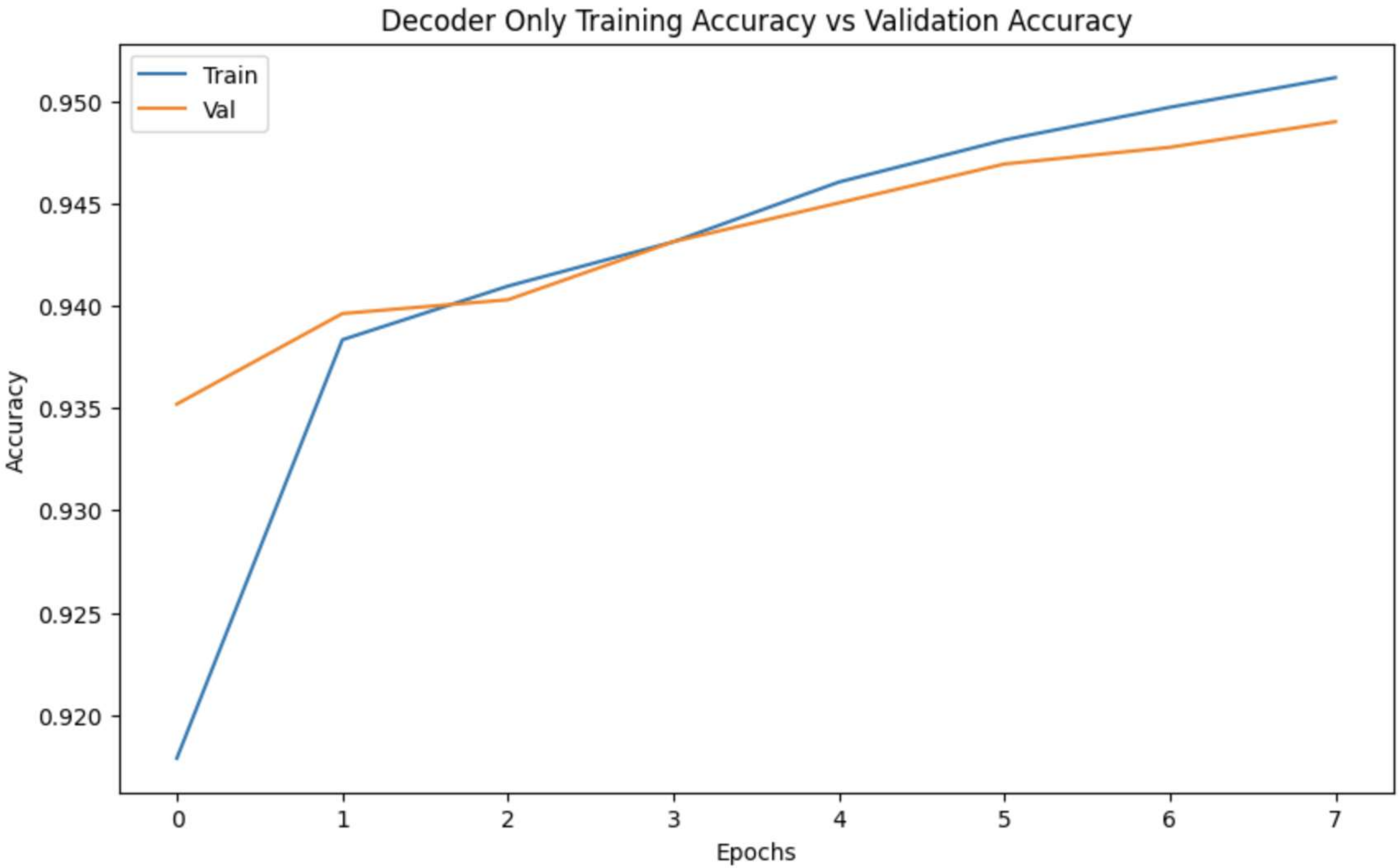

Enc + Dec Training Accuracy vs Validation Accuracy
Train
Val
Accuracy
Epochs
0.950
0.945
0.940
0.935
0.930
0.925
0.920
0.915
0
1
2
3
4
5
6
7



Encoder Only Training Loss vs Validation Loss
Train
Val
Loss
0.7
0.6
0.5
0.4
0.3
0
1
2
3
4
5
6
7
Epochs


Encoder Only Training Accuracy vs Validation Accuracy
Train
Val
0.955
0.950
0.945
0.940
0.935
0.930
0.925
Accuracy
0
1
2
3
4
5
6
7
Epochs


Decoder Only Training Loss vs Validation Loss
Train
Val
Loss
Epochs
1.2
1.0
0.8
0.6
0.4
0
1
2
3
4
5
6
7


Decoder Only Training Accuracy vs Validation Accuracy
Train
Val
Accuracy
Epochs
0.950
0.945
0.940
0.935
0.930
0.925
0.920
0
1
2
3
4
5
6
7