

```

1 # =====
2 # importing libraries
3 # =====
4
5 import tensorflow as tf
6 from tensorflow.keras.layers import Input, Dense, Flatten, Conv2D, MaxPool2D, Activation, Dropout, Embedding, GRU, RepeatVector
7 from tensorflow.keras.models import Sequential, Model
8 from tensorflow.keras import regularizers
9 from tensorflow.keras.optimizers import SGD
10 from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
11 from tensorflow.keras.metrics import RootMeanSquaredError
12 from tensorflow.keras.preprocessing.text import Tokenizer
13 from tensorflow.keras.preprocessing.sequence import pad_sequences
14 from tensorflow.keras.layers.experimental.preprocessing import TextVectorization
15 from tensorflow.keras.layers.experimental import preprocessing
16 from tensorflow.keras.utils import to_categorical
17 from tensorflow.keras.models import model_from_json
18 from sklearn.model_selection import train_test_split
19
20 import numpy as np
21
22 import unicodedata
23
24 import re
25
26 import matplotlib.pyplot as plt
27 %matplotlib inline
28
29
30
31 # =====
32 # problem_set_1
33 # =====
34
35 BATCH_SIZE = 64
36 EPOCHS = 50
37
38 def vgg16_custom_arch():
39
40     model = Sequential()
41     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'))
42     model.add(Conv2D(filters=64, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_1_2'))
43     model.add(MaxPool2D(pool_size=(2,2), strides=(2,2), name='max_pool_1'))
44
45     model.add(Conv2D(filters=128, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_2_1'))
46     model.add(Conv2D(filters=128, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_2_2'))
47     model.add(MaxPool2D(pool_size=(2,2), strides=(2,2), name='max_pool_2'))
48
49     model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_3_1'))
50     model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_3_2'))
51     model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_3_3'))
52     model.add(MaxPool2D(pool_size=(2,2), strides=(2,2), name='max_pool_3'))
53
54     model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_4_1'))
55     model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_4_2'))
56     model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_4_3'))
57     model.add(MaxPool2D(pool_size=(2,2), strides=(2,2), name='max_pool_4'))
58
59     model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_5_1'))
60     model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_5_2'))
61     model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), padding='same', dilation_rate=(1,1), activation='relu', name='conv_5_3'))
62     model.add(MaxPool2D(pool_size=(2,2), strides=(2,2), name='max_pool_5'))
63
64     model.add(Flatten(name='flatten_1'))
65     model.add(Dense(units=4096, activation='relu', name='fc1'))
66     model.add(Dropout(rate=0.5))
67     model.add(Dense(units=4096, activation='relu', name='fc2'))
68     model.add(Dropout(rate=0.5))
69     model.add(Dense(units=1, activation='linear', name='output'))
70
71     model.compile(loss='mean_squared_error', optimizer='adam', metrics=[RootMeanSquaredError()])
72     return model
73

```

```

74 vgg16_model = vgg16_custom_arch()
75
76 X_tr: np.ndarray = np.load('./facesAndAges/faces.npy')
77 y_tr: np.ndarray = np.load('./facesAndAges/ages.npy')
78
79 shuffling_indices: np.ndarray = np.arange(X_tr.shape[0])
80 np.random.shuffle(shuffling_indices)
81 X_tr: np.ndarray = X_tr[shuffling_indices]
82 y_tr: np.ndarray = y_tr[shuffling_indices]
83 del shuffling_indices
84 trainX, trainY = X_tr[0:5250,:], y_tr[0:5250]
85 valX, valY = X_tr[5250:6000,:], y_tr[5250:6000]
86 testX, testY = X_tr[6000:7500,:], y_tr[6000:7500]
87 del X_tr, y_tr
88
89 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)])
90 test_loss, test_rmse = vgg16_model.evaluate(testX, testY)
91 print(f'Test Loss: {test_loss:.4f}, Test RMSE: {test_rmse:.4f}')
92 # Test Loss: 141.9804, Test RMSE: 11.9156
93
94 fig, axs = plt.subplots(2, 1, figsize=(10,13))
95 axs[0].plot(history.history['loss'])
96
97 axs[0].plot(history.history['val_loss'])
98 axs[0].title.set_text('Training Loss vs Validation Loss')
99 axs[0].set_xlabel('Epochs')
100 axs[0].set_ylabel('Loss')
101 axs[0].legend(['Train', 'Val'])
102 axs[1].plot(history.history['root_mean_squared_error'])
103 axs[1].plot(history.history['val_root_mean_squared_error'])
104 axs[1].title.set_text('Training RMSE vs Validation RMSE')
105 axs[1].set_xlabel('Epochs')
106 axs[1].set_ylabel('RMSE')
107 axs[1].legend(['Train', 'Val'])
108
109
110
111
112
113 # =====
114 # problem_set_2
115 # =====
116
117 X_train: np.ndarray = np.load("./homework5_question2_data/X_train.npy", allow_pickle=True)
118 y_train: np.ndarray = np.load("./homework5_question2_data/y_train.npy", allow_pickle=True)
119 X_test: np.ndarray = np.load("./homework5_question2_data/X_test.npy", allow_pickle=True)
120 y_test: np.ndarray = np.load("./homework5_question2_data/y_test.npy", allow_pickle=True)
121
122 hidden_size = 8
123 epochs = 6
124 learning_rate = 1e-4
125
126 class VanillaRNN:
127     def __init__(self, input_size, hidden_size):
128         self.input_size = input_size
129         self.hidden_size = hidden_size
130
131         self.W_xh = np.random.randn(hidden_size, input_size)
132         self.W_hh = np.random.randn(hidden_size, hidden_size)
133         self.W_hy = np.random.randn(input_size, hidden_size)
134         self.b_h = np.zeros((hidden_size, 1))
135         self.b_y = np.zeros((input_size, 1))
136
137     def forward(self, inputs, seq_length):
138         hidden_states = np.zeros((seq_length, self.hidden_size))
139         outputs = np.zeros((seq_length, self.input_size))
140
141         h_t = np.zeros((self.hidden_size, 1))
142
143         for t in range(seq_length):
144             x_t = inputs[t].reshape(-1, 1)
145             h_t = np.tanh(np.dot(self.W_xh, x_t) + np.dot(self.W_hh, h_t) + self.b_h)
146             y_t = np.dot(self.W_hy, h_t) + self.b_y

```

```

147
148         hidden_states[t] = h_t.flatten()
149         outputs[t] = y_t.flatten()
150
151     return hidden_states, outputs
152
153 def backward(self, inputs, hidden_states, outputs, targets, learning_rate):
154     seq_length = inputs.shape[0]
155     dW_xh, dW_hh, dW_hy, db_h, db_y = (
156         np.zeros_like(self.W_xh),
157         np.zeros_like(self.W_hh),
158         np.zeros_like(self.W_hy),
159         np.zeros_like(self.b_h),
160         np.zeros_like(self.b_y),
161     )
162     dh_next = np.zeros((self.hidden_size, 1))
163
164     for t in reversed(range(seq_length)):
165         x_t = inputs[t].reshape(-1, 1)
166         h_t = hidden_states[t].reshape(-1, 1)
167         y_t = outputs[t].reshape(-1, 1)
168
169         if t < len(targets):
170             target_t = targets[t].reshape(-1, 1)
171             dy = y_t - target_t
172         else:
173             dy = y_t - 0
174
175         dW_hy += np.dot(dy, h_t.T)
176         db_y += dy
177
178         dh = np.dot(self.W_hy.T, dy) + dh_next
179         dh_raw = (1 - h_t ** 2) * dh
180         db_h += dh_raw
181
182         dW_xh += np.dot(dh_raw, x_t.T)
183         dW_hh += np.dot(dh_raw, hidden_states[t - 1].reshape(-1, 1).T) if t > 0 else 0
184         dh_next = np.dot(self.W_hh.T, dh_raw)
185
186     for dparam in [dW_xh, dW_hh, dW_hy, db_h, db_y]:
187         np.clip(dparam, -5, 5, out=dparam)
188
189     self.W_xh -= learning_rate * dW_xh
190     self.W_hh -= learning_rate * dW_hh
191     self.W_hy -= learning_rate * dW_hy
192
193     self.b_h -= learning_rate * db_h
194     self.b_y -= learning_rate * db_y
195
196 def plot_VanillaRNN():
197     input_size = X_train[0].shape[1]
198     rnn = VanillaRNN(input_size, hidden_size)
199     losses = []
200
201     for epoch in range(epochs):
202         total_loss = 0
203         for i in range(len(X_train)):
204             inputs = X_train[i]
205             targets = y_train[i]
206             seq_length = len(inputs)
207
208             hidden_states, outputs = rnn.forward(inputs, seq_length)
209             loss = np.mean(((outputs - targets) ** 2))
210
211             rnn.backward(inputs, hidden_states, outputs, targets, learning_rate)
212             total_loss += loss
213
214         average_loss = total_loss / len(X_train)
215         losses.append(average_loss)
216
217     print(f"Epoch {epoch + 1}/{epochs}, Loss: {average_loss:.4f}")
218
219 plt.plot(range(1, epochs + 1), losses)
220 plt.xlabel('Epochs')
221 plt.ylabel('Loss')
222

```

```

221 plt.title('Training Epoch vs. Loss VanillaRNN')
222 plt.show()
223
224 total_loss = 0
225 for i in range(len(X_test)):
226     inputs = X_test[i]
227     targets = y_test[i]
228     seq_length = len(inputs)
229     _, outputs = rnn.forward(inputs, seq_length)
230     total_loss += np.mean((outputs - targets) ** 2)
231 average_loss = total_loss / len(X_test)
232 print(f'\nVanillaRNN Test Loss: {average_loss:.4f}\n')
233 return None
234
235 class VanillaRNNMin:
236     def __init__(self, input_size, hidden_size):
237         self.input_size = input_size
238         self.hidden_size = hidden_size
239
240         self.W_xh = [np.random.randn(hidden_size, input_size) for _ in range(hidden_size)]
241         self.W_hh = [np.random.randn(hidden_size, hidden_size) for _ in range(hidden_size)]
242         self.W_hy = np.random.randn(input_size, hidden_size)
243         self.b_h = [np.zeros((hidden_size, 1)) for _ in range(hidden_size)]
244         self.b_y = np.zeros((input_size, 1))
245
246     def forward(self, inputs, seq_length):
247         hidden_states = np.zeros((seq_length, self.hidden_size))
248         outputs = np.zeros((seq_length, self.input_size))
249
250         h_t = [np.zeros((self.hidden_size, 1)) for _ in range(self.hidden_size)]
251
252         for t in range(seq_length):
253             x_t = inputs[t].reshape(-1, 1)
254             for i in range(self.hidden_size):
255                 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])
256             y_t = np.dot(self.W_hy, h_t[-1]) + self.b_y
257
258             hidden_states[t] = h_t[-1].flatten()
259             outputs[t] = y_t.flatten()
260
261         return hidden_states, outputs
262
263     def backward(self, inputs, hidden_states, outputs, targets, learning_rate=0.01):
264         seq_length, input_size = inputs.shape
265         dW_xh, dW_hh, dW_hy, db_h, db_y = (
266             [np.zeros_like(self.W_xh[0]) for _ in range(self.hidden_size)],
267             [np.zeros_like(self.W_hh[0]) for _ in range(self.hidden_size)],
268             np.zeros_like(self.W_hy),
269             [np.zeros_like(self.b_h[0]) for _ in range(self.hidden_size)],
270             np.zeros_like(self.b_y),
271         )
272         dh_next = np.zeros((self.hidden_size, 1))
273
274         for t in reversed(range(seq_length)):
275             x_t = inputs[t].reshape(-1, 1)
276             h_t = [hidden_states[t].reshape(-1, 1) for _ in range(self.hidden_size)]
277             y_t = outputs[t].reshape(-1, 1)
278
279             if t < len(targets):
280                 target_t = targets[t].reshape(-1, 1)
281                 dy = y_t - target_t
282             else:
283                 dy = y_t - 0
284
285             dW_hy += np.dot(dy, h_t[-1].T)
286             db_y += dy
287
288             dh = np.dot(self.W_hy.T, dy) + dh_next
289             dh_raw = [np.dot(self.W_hh[i].T, dh) for i in range(self.hidden_size)]
290             db_h = [dhr + dh for dhr, dh in zip(dh_raw, dh_next)]
291
292             dW_xh = [dwxh + np.dot(dhr, x_t.T) for dwxh, dhr in zip(dW_xh, dh_raw)]
293             dW_hh = [dwhh + np.dot(dhr, h.T) for dwhh, dhr, h in zip(dW_hh, dh_raw, h_t)]
294             dh_next = np.dot(self.W_hh[-1].T, dh_raw[-1])

```

```

294         out_next = np.dot(self.W_hh[-1], out_dwh[-1])
295
296     for i in range(self.hidden_size):
297         for dparam in [dW_xh[i], dW_hh[i], db_h[i]]:
298             np.clip(dparam, -5, 5, out=dparam)
299             self.W_xh[i] -= learning_rate * dW_xh[i]
300             self.W_hh[i] -= learning_rate * dW_hh[i]
301             self.b_h[i] -= learning_rate * db_h[i]
302
303     for dparam in [dW_hy, db_y]:
304         np.clip(dparam, -5, 5, out=dparam)
305
306     self.W_hy -= learning_rate * dW_hy
307     self.b_y -= learning_rate * db_y
308
309 def plot_VanillaRNNMin():
310     min_seq_length = min([i.shape[0] for i in X_train])
311     X_train_min_truncated = [i[:min_seq_length] for i in X_train]
312     X_test_min_truncated = [i[:min_seq_length] for i in X_test]
313     input_size = X_train_min_truncated[0].shape[1]
314     rnn = VanillaRNNMin(input_size, hidden_size)
315     losses = []
316
317     for epoch in range(epochs):
318         total_loss = 0
319
320         for i in range(len(X_train_min_truncated)):
321             inputs = X_train_min_truncated[i]
322             targets = y_train[i]
323
324             seq_length, input_size = inputs.shape
325             hidden_states, outputs = rnn.forward(inputs, seq_length)
326             loss = np.mean(((outputs - targets) ** 2))
327
328             rnn.backward(inputs, hidden_states, outputs, targets, learning_rate)
329             total_loss += loss
330
331         average_loss = total_loss / len(X_train_min_truncated)
332         losses.append(average_loss)
333
334         print(f"Epoch {epoch + 1}/{epochs}, Loss: {average_loss:.4f}")
335
336     plt.plot(range(1, epochs + 1), losses)
337     plt.xlabel('Epochs')
338     plt.ylabel('Loss')
339     plt.title('Training Epoch vs. Loss VanillaRNNMin')
340     plt.show()
341
342     total_loss = 0
343     for i in range(len(X_test_min_truncated)):
344         inputs = X_test_min_truncated[i]
345         targets = y_test[i]
346         seq_length = len(inputs)
347         _, outputs = rnn.forward(inputs, seq_length)
348         total_loss += np.mean(((outputs - targets) ** 2))
349     average_loss = total_loss / len(X_test_min_truncated)
350     print(f'\nVanillaRNNMin Test Loss: {average_loss:.4f}\n')
351     return None
352
353 class VanillaRNNMax:
354     def __init__(self, input_size, hidden_size):
355         self.input_size = input_size
356         self.hidden_size = hidden_size
357
358         self.W_xh = [np.random.randn(hidden_size, input_size) for _ in range(hidden_size)]
359         self.W_hh = [np.random.randn(hidden_size, hidden_size) for _ in range(hidden_size)]
360         self.W_hy = np.random.randn(input_size, hidden_size)
361         self.b_h = [np.zeros((hidden_size, 1)) for _ in range(hidden_size)]
362         self.b_y = np.zeros((input_size, 1))
363
364     def forward(self, inputs, seq_length):
365         hidden_states = np.zeros((seq_length, self.hidden_size))
366         outputs = np.zeros((seq_length, self.input_size))
367
368         h_t = [np.zeros((self.hidden_size, 1)) for _ in range(self.hidden_size)]

```

```

368 h_t = [np.zeros((self.hidden_size, 1)) for _ in range(self.hidden_size)]
369
370 for t in range(seq_length):
371     x_t = inputs[t].reshape(-1, 1)
372     for i in range(self.hidden_size):
373         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])
374     y_t = np.dot(self.W_hy, h_t[-1]) + self.b_y
375
376     hidden_states[t] = h_t[-1].flatten()
377     outputs[t] = y_t.flatten()
378
379 return hidden_states, outputs
380
381 def backward(self, inputs, hidden_states, outputs, targets, seq_length, max_seq_length, learning_rate):
382     criteria_c = max_seq_length - seq_length
383     dW_xh, dW_hh, dW_hy, db_h, db_y = (
384         [np.zeros_like(self.W_xh[0]) for _ in range(self.hidden_size)],
385         [np.zeros_like(self.W_hh[0]) for _ in range(self.hidden_size)],
386         np.zeros_like(self.W_hy),
387         [np.zeros_like(self.b_h[0]) for _ in range(self.hidden_size)],
388         np.zeros_like(self.b_y),
389     )
390     dh_next = np.zeros((self.hidden_size, 1))
391
392     total_loss = 0
393
394     for t in reversed(range(seq_length)):
395         x_t = inputs[t].reshape(-1, 1)
396         h_t = [hidden_states[t].reshape(-1, 1) for _ in range(self.hidden_size)]
397         y_t = outputs[t].reshape(-1, 1)
398
399         if t < len(targets):
400             target_t = targets[t].reshape(-1, 1)
401             dy = y_t - target_t
402
403             loss = np.mean(((y_t - target_t) ** 2))
404             total_loss += loss
405
406         else:
407             dy = y_t - 0
408
409         dW_hy += np.dot(dy, h_t[-1].T)
410         db_y += dy
411
412         dh = np.dot(self.W_hy.T, dy) + dh_next
413         dh_raw = [np.dot(self.W_hh[i].T, dh) for i in range(self.hidden_size)]
414         db_h = [dhr + dh for dhr, dh in zip(dh_raw, dh_next)]
415
416         dW_xh = [dwxh + np.dot(dhr, x_t.T) for dwxh, dhr in zip(dW_xh, dh_raw)]
417         dW_hh = [dwhh + np.dot(dhr, h.T) for dwhh, dhr, h in zip(dW_hh, dh_raw, h_t)]
418         dh_next = np.dot(self.W_hh[-1].T, dh_raw[-1])
419
420     for i in range(self.hidden_size):
421         for dparam in [dW_xh[i], dW_hh[i], db_h[i]]:
422             np.clip(dparam, -5, 5, out=dparam)
423
424         self.W_xh[i] -= learning_rate * dW_xh[i]
425         self.W_hh[i] -= learning_rate * dW_hh[i]
426         self.b_h[i] -= learning_rate * db_h[i]
427
428     for dparam in [dW_hy, db_y]:
429         np.clip(dparam, -5, 5, out=dparam)
430
431     self.W_hy -= learning_rate * dW_hy
432     self.b_y -= learning_rate * db_y
433
434     average_loss = total_loss / seq_length
435     return average_loss
436
437 def plot_VanillaRNNMax():
438     max_seq_length = max([i.shape[0] for i in X_train])
439     input_size = X_train[0].shape[1]
440     rnn = VanillaRNNMax(input_size, hidden_size)
441     losses = []

```

```

442
443     for epoch in range(epochs):
444         total_loss = 0
445
446         for i in range(len(X_train)):
447             inputs = X_train[i]
448             targets = y_train[i]
449             seq_length = len(X_train[i])
450             inputs = np.pad(inputs, ((0, max_seq_length - inputs.shape[0]), (0, 0)), mode='constant')
451
452             hidden_states, outputs = rnn.forward(inputs, max_seq_length)
453             loss = rnn.backward(inputs, hidden_states, outputs, targets, seq_length, max_seq_length, learning_rate)
454             total_loss += loss
455
456         average_loss = total_loss / len(X_train)
457         losses.append(average_loss)
458
459         print(f"Epoch {epoch + 1}/{epochs}, Loss: {average_loss:.4f}")
460
461     plt.plot(range(1, epochs + 1), losses)
462     plt.xlabel('Epochs')
463     plt.ylabel('Loss')
464     plt.title('Training Epoch vs. Loss VanillaRNNMax')
465     plt.show()
466
467     total_loss = 0
468     for i in range(len(X_test)):
469         inputs = X_test[i]
470         inputs = np.pad(inputs, ((0, max_seq_length - inputs.shape[0]), (0, 0)), mode='constant')
471         targets = y_test[i]
472         seq_length = len(inputs)
473         _, outputs = rnn.forward(inputs, seq_length)
474         total_loss += np.mean((outputs - targets) ** 2)
475     average_loss = total_loss / len(X_test)
476     print(f"\nVanillaRNNMax Test Loss: {average_loss:.4f}\n")
477     return None
478
479 print('Vanilla RNN')
480
481 plot_VanillaRNN()
482 plot_VanillaRNNMin()
483 plot_VanillaRNNMax()
484
485 """
486 parameters + output
487
488 =====
489
490 hidden_size = 8
491 epochs = 6
492 learning_rate = 1e-4
493
494 Vanilla RNN
495 Epoch 1/6, Loss: 4.4596
496 Epoch 2/6, Loss: 2.1227
497 Epoch 3/6, Loss: 0.9229
498 Epoch 4/6, Loss: 0.4005
499 Epoch 5/6, Loss: 0.1952
500 Epoch 6/6, Loss: 0.1141
501 VanillaRNN Test Loss: 0.0877
502
503 Vanilla RNN with Min Sequence Length
504 Epoch 1/6, Loss: 5.1719
505 Epoch 2/6, Loss: 3.2882
506 Epoch 3/6, Loss: 2.0529
507 Epoch 4/6, Loss: 1.2857
508 Epoch 5/6, Loss: 0.8213
509 Epoch 6/6, Loss: 0.5385
510 VanillaRNNMin Test Loss: 0.4455
511
512 Vanilla RNN with Max Sequence Length
513 Epoch 1/6, Loss: 0.4769
514 Epoch 2/6, Loss: 0.2301

```

```
516 Epoch 3/6, Loss: 0.0972
517 Epoch 4/6, Loss: 0.0385
518 Epoch 5/6, Loss: 0.0161
519 Epoch 6/6, Loss: 0.0073
520 VanillaRNNMax Test Loss: 0.0582
521
522 =====
523
524 hidden_size = 12
525 epochs = 8
526 learning_rate = 3e-6
527
528 Vanilla RNN
529 Epoch 1/8, Loss: 7.9606
530 Epoch 2/8, Loss: 7.8026
531 Epoch 3/8, Loss: 7.6467
532 Epoch 4/8, Loss: 7.4952
533 Epoch 5/8, Loss: 7.3455
534 Epoch 6/8, Loss: 7.1980
535 Epoch 7/8, Loss: 7.0529
536 Epoch 8/8, Loss: 6.9089
537 VanillaRNN Test Loss: 6.9403
538
539 Vanilla RNN with Min Sequence Length
540 Epoch 1/8, Loss: 8.6852
541 Epoch 2/8, Loss: 8.5635
542 Epoch 3/8, Loss: 8.4432
543 Epoch 4/8, Loss: 8.3241
544 Epoch 5/8, Loss: 8.2063
545 Epoch 6/8, Loss: 8.0898
546 Epoch 7/8, Loss: 7.9745
547 Epoch 8/8, Loss: 7.8605
548 VanillaRNNMin Test Loss: 7.8463
549
550 Vanilla RNN with Max Sequence Length
551 Epoch 1/8, Loss: 0.7396
552 Epoch 2/8, Loss: 0.7274
553 Epoch 3/8, Loss: 0.7153
554 Epoch 4/8, Loss: 0.7034
555 Epoch 5/8, Loss: 0.6916
556 Epoch 6/8, Loss: 0.6800
557 Epoch 7/8, Loss: 0.6685
558 Epoch 8/8, Loss: 0.6572
559 VanillaRNNMax Test Loss: 9.2305
560
561 =====
562
563 hidden_size = 8
564 epochs = 6
565 learning_rate = 3e-4
566
567 Vanilla RNN
568 Epoch 1/6, Loss: 1.5473
569 Epoch 2/6, Loss: 0.1255
570 Epoch 3/6, Loss: 0.0368
571 Epoch 4/6, Loss: 0.0230
572 Epoch 5/6, Loss: 0.0198
573 Epoch 6/6, Loss: 0.0189
574 VanillaRNN Test Loss: 0.0164
575
576 Vanilla RNN with Min Sequence Length
577 Epoch 1/6, Loss: 2.0644
578 Epoch 2/6, Loss: 0.5714
579 Epoch 3/6, Loss: 0.1979
580 Epoch 4/6, Loss: 0.0844
581 Epoch 5/6, Loss: 0.0442
582 Epoch 6/6, Loss: 0.0291
583 VanillaRNNMin Test Loss: 0.0227
584
585 Vanilla RNN with Max Sequence Length
586 Epoch 1/6, Loss: 0.2056
587 Epoch 2/6, Loss: 0.0158
588 Epoch 3/6, Loss: 0.0029
```



```

589 Epoch 4/6, Loss: 0.0019
590 Epoch 5/6, Loss: 0.0017
591 Epoch 6/6, Loss: 0.0017
592 VanillaRNNMax Test Loss: 0.0163
593
594
595
596 c) Analyze the results and discuss the advantages and disadvantages of each approach in terms of modeling sequences with varying lengths.
597
598 => Based on the above results we can conclude the following based on the 3 architectures.
599
600 Vanilla RNN
601
602 Advantages:
603     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.
604     It was slightly faster in training as compared to the rest as the weights were being shared across time steps.
605     It is relatively a simple model and achieved good results for the mentioned parameters.
606     The sequence length was not padded or truncated during training.
607     It requires less computational resources compared to Vanilla RNN with Max Sequence Length architecture.
608     This network would be a good choice if we have sequences of varying lengths and need a balance between performance and efficiency.
609
610 Disadvantages:
611     The test loss, while low, may still not be sufficient for some applications, depending on the specific problem being solved.
612
613 Vanilla RNN with Min Sequence Length
614
615 Advantages:
616     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.
617     It was slightly slower in training as compared to the Vanilla RNN as the weights were not being shared across time steps.
618     It has higher test losses compared to Vanilla RNN with the same hidden size.
619     The sequence length was truncated during training (minimum length).
620     Although it was truncated to minimum length still it provides a reasonable performance considering the sequence length and loss of data.
621     This model might be more efficient when working with short sequences.
622     It requires less computational resources compared to Vanilla RNN with Max Sequence Length architecture.
623     This network would be a good choice if we have short sequences and need a balance between performance and efficiency.
624
625 Disadvantages:
626     The test loss is higher than Vanilla RNN for the same hidden size, suggesting that it might struggle with longer sequences.
627
628 Vanilla RNN with Max Sequence Length
629
630 Advantages:
631     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.
632     It was slightly slower in training as compared to the Vanilla RNN as the weights were not being shared across time steps.
633     It has higher test losses compared to Vanilla RNN with the same hidden size.
634     The sequence length was padded during training (maximum length).
635     Although it was padded to maximum sequence length still it provides a reasonable performance considering the zero padding.
636     This model might be more efficient when working with long sequences and requiring long-term dependencies.
637     This network would be a good choice if we have long sequences and need accuracy over efficiency.
638
639 Disadvantages:
640     The test loss for shorter sequences is considerably higher, suggesting that it might not generalize well to shorter sequences possibly to the padding.
641     It requires more computational resources compared to Vanilla RNN with Min Sequence Length architecture.
642
643 """
644
645
646
647
648
649 # =====
650 # problem_set_3
651 # =====
652
653 start_token = 'eos'
654 end_token = 'eos'
655 BATCH_SIZE = 32
656 EPOCHS = 8
657 GRU_UNITS = 256
658
659 def txt_pre_processing(txt:str)->str:
660     txt = txt.lower().strip()
661     txt = unicodedata.normalize('NFKD',txt).encode('ascii', 'ignore').decode('utf-8')
662     txt = re.sub(pattern=r'[^\sa-z\d\.\?\!\,\,]', repl='', string=txt)

```

```

663 txt = re.sub(pattern=r'([\.\?!\,\,])', repl=r' \1 ', string=txt).strip()
664 txt = start_token + ' ' + txt + ' ' + end_token
665 return txt
666
667 def load_data() -> tuple:
668     context : list = list()
669     target : list = list()
670     with open(file='./eng-fra.txt', mode='r', encoding='utf-8') as inputstream:
671         for text in inputstream:
672             lines = text.replace('\n', '').replace('\r', '').split('\t')
673             eng_txt = lines[0]
674             fr_txt = lines[1]
675             eng_txt = txt_pre_processing(txt=eng_txt)
676             fr_txt = txt_pre_processing(txt=fr_txt)
677             context.append(eng_txt)
678             target.append(fr_txt)
679     context = np.array(context)
680     target = np.array(target)
681     return context, target
682
683 eng_sentences, fr_sentences = load_data()
684 shuffling_indices = np.arange(len(eng_sentences))
685 np.random.shuffle(shuffling_indices)
686 eng_sentences = eng_sentences[shuffling_indices]
687 fr_sentences = fr_sentences[shuffling_indices]
688
689 eng_tokenizer = Tokenizer()
690 eng_tokenizer.fit_on_texts(eng_sentences)
691 eng_vocab_size = len(eng_tokenizer.word_index) + 1
692
693 fr_tokenizer = Tokenizer()
694 fr_tokenizer.fit_on_texts(fr_sentences)
695 fr_vocab_size = len(fr_tokenizer.word_index) + 1
696
697 eng_sequences = eng_tokenizer.texts_to_sequences(eng_sentences)
698 fr_sequences = fr_tokenizer.texts_to_sequences(fr_sentences)
699
700 max_seq_length = 52
701 eng_sequences = pad_sequences(eng_sequences, maxlen=max_seq_length, padding='post')
702 fr_sequences = pad_sequences(fr_sequences, maxlen=max_seq_length, padding='post')
703 split_80_20: int = int(eng_sequences.shape[0]*0.8)
704 X_train, y_train = eng_sequences[:split_80_20,:], fr_sequences[:split_80_20]
705 X_test, y_test = eng_sequences[split_80_20,:], fr_sequences[split_80_20:]
706 y_train = to_categorical(y_train, num_classes=fr_vocab_size)
707 y_test = to_categorical(y_test, num_classes=fr_vocab_size)
708
709 # =====
710 # enc + dec
711 # =====
712
713 encoder_inputs = tf.keras.layers.Input(shape=(None,))
714 encoder_embedding = Embedding(input_dim=eng_vocab_size, output_dim=GRU_UNITS)(encoder_inputs)
715 encoder_gru = GRU(GRU_UNITS, return_state=True)
716 encoder_outputs, encoder_state = encoder_gru(encoder_embedding)
717 decoder_inputs = tf.keras.layers.Input(shape=(None,))
718 decoder_embedding = Embedding(input_dim=fr_vocab_size, output_dim=GRU_UNITS)(decoder_inputs)
719 decoder_gru = GRU(GRU_UNITS, return_sequences=True, return_state=True)
720 decoder_outputs, _ = decoder_gru(decoder_embedding, initial_state=encoder_state)
721 decoder_dense = Dense(fr_vocab_size, activation='softmax')
722 decoder_outputs = decoder_dense(decoder_outputs)
723 model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
724 model.compile(optimizer='adam', loss='Categorical_crossentropy', metrics=['accuracy'])
725 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)])
726 del model, X_train, y_train, X_test, y_test
727 fig, axs = plt.subplots(2, 1, figsize=(10,13))
728 axs[0].plot(history.history['loss'])
729 axs[0].plot(history.history['val_loss'])
730 axs[0].title.set_text('Enc + Dec Training Loss vs Validation Loss')
731 axs[0].set_xlabel('Epochs')
732 axs[0].set_ylabel('Loss')
733 axs[0].legend(['Train', 'Val'])
734 axs[1].plot(history.history['accuracy'])
735 axs[1].plot(history.history['val_accuracy'])

```

```

736 axs[1].title.set_text('Enc + Dec Training Accuracy vs Validation Accuracy')
737 axs[1].set_xlabel('Epochs')
738 axs[1].set_ylabel('Accuracy')
739 axs[1].legend(['Train', 'Val'])
740
741 encoder_model = Model(encoder_inputs, encoder_state)
742 decoder_state_input = Input(shape=(GRU_UNITS,))
743 decoder_inputs = Input(shape=(1,))
744 decoder_embedding_inference = Embedding(input_dim=fr_vocab_size, output_dim=GRU_UNITS)(decoder_inputs)
745 decoder_gru_inference = GRU(GRU_UNITS, return_sequences=True, return_state=True)
746 decoder_outputs_inference, decoder_state_inference = decoder_gru_inference(decoder_embedding_inference, initial_state=decoder_state_input)
747 decoder_outputs_inference = decoder_dense(decoder_outputs_inference)
748 decoder_model = Model([decoder_inputs, decoder_state_input], [decoder_outputs_inference, decoder_state_inference])
749
750 def translate_sentence(input_text):
751     stop_crit = len(input_text)+3
752     input_text = txt_pre_processing(txt=input_text)
753     input_seq = eng_tokenizer.texts_to_sequences([input_text])
754     input_seq = pad_sequences(input_seq, maxlen=max_seq_length, padding='post')
755     input_seq = tf.nn.constant(input_seq)
756     states_value = encoder_model.predict(input_seq)
757
758     target_seq = tf.constant([fr_tokenizer.word_index[start_token]])
759     target_text = []
760     stop_condition = False
761     prev_token_index = None
762
763     while not stop_condition:
764         output_tokens, h = decoder_model.predict([target_seq, states_value])
765         sampled_token_index = np.argmax(output_tokens[0, -1, :])
766         if (sampled_token_index == 0):
767             sampled_word = ''
768         else:
769             sampled_word = fr_tokenizer.index_word[sampled_token_index]
770
771         if (sampled_word != end_token) and (sampled_word != ''):
772             target_text.append(sampled_word)
773
774         if sampled_word == end_token or len(target_text) >= stop_crit:
775             stop_condition = True
776
777         prev_token_index = sampled_token_index
778         target_seq = tf.constant([sampled_token_index])
779         states_value = h
780     return ' '.join(target_text)
781 input_text = "I won!"
782 translation = translate_sentence(input_text)
783 del decoder_model
784
785 # =====
786 # only enc
787 # =====
788
789 autoencoder_inputs = tf.keras.layers.Input(shape=(max_seq_length,))
790 autoencoder_embedding = Embedding(input_dim=eng_vocab_size, output_dim=GRU_UNITS)(autoencoder_inputs)
791 autoencoder_gru = GRU(GRU_UNITS, return_state=True)
792 autoencoder_outputs, autoencoder_state = autoencoder_gru(autoencoder_embedding)
793 autoencoder_outputs = RepeatVector(max_seq_length)(autoencoder_outputs)
794 autoencoder_gru = GRU(GRU_UNITS, return_sequences=True, return_state=True)
795 autoencoder_outputs, _ = autoencoder_gru(autoencoder_outputs, initial_state=autoencoder_state)
796 autoencoder_dense = Dense(eng_vocab_size, activation='softmax')
797 autoencoder_outputs = autoencoder_dense(autoencoder_outputs)
798 autoencoder_model = Model(autoencoder_inputs, autoencoder_outputs)
799 autoencoder_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
800 split_80_20: int = int(eng_sequences.shape[0]*0.8)
801 X_train = eng_sequences[:split_80_20,:]
802 X_test = eng_sequences[split_80_20:,:]
803 y_train_autoencoder = to_categorical(X_train, num_classes=eng_vocab_size)
804 y_test_autoencoder = to_categorical(X_test, num_classes=eng_vocab_size)
805
806 f_loss = list()
807 f_acc = list()
808 f_val_loss = list()
809 f_val_acc = list()
810

```

```

810
811 for epoch in range(EPOCHS):
812     history = autoencoder_model.fit(X_train, y_train_autoencoder, epochs=1, batch_size=BATCH_SIZE, verbose=1)
813     val_loss, val_accuracy = autoencoder_model.evaluate(X_test, y_test_autoencoder, batch_size=BATCH_SIZE, verbose=1)
814     f_loss.append(history.history['loss'])
815     f_acc.append(history.history['accuracy'])
816
817     f_val_loss.append(val_loss)
818     f_val_acc.append(val_accuracy)
819
820 del autoencoder_model, X_train, y_train_autoencoder, X_test, y_test_autoencoder
821 fig, axs = plt.subplots(2, 1, figsize=(10,13))
822 axs[0].plot(f_loss)
823 axs[0].plot(f_val_loss)
824 axs[0].title.set_text('Encoder Only Training Loss vs Validation Loss')
825 axs[0].set_xlabel('Epochs')
826 axs[0].set_ylabel('Loss')
827 axs[0].legend(['Train', 'Val'])
828 axs[1].plot(f_acc)
829 axs[1].plot(f_val_acc)
830 axs[1].title.set_text('Encoder Only Training Accuracy vs Validation Accuracy')
831 axs[1].set_xlabel('Epochs')
832 axs[1].set_ylabel('Accuracy')
833 axs[1].legend(['Train', 'Val'])
834
835 # =====
836 # save only enc
837 # =====
838 pretrained_encoder_model = Model(autoencoder_inputs, autoencoder_state)
839
840 for layer in pretrained_encoder_model.layers:
841     layer.trainable = False
842
843 pretrained_encoder_model.save_weights('pretrained_encoder_model_weights.h5')
844 pretrained_encoder_model_json = pretrained_encoder_model.to_json()
845 with open(file='pretrained_encoder_model.json', mode='w') as json_file:
846     json_file.write(pretrained_encoder_model_json)
847 del pretrained_encoder_model_json
848
849 with open(file='pretrained_encoder_model.json', mode='r') as json_file:
850     pretrained_encoder_model_json = json_file.read()
851 pretrained_encoder_model = model_from_json(pretrained_encoder_model_json)
852 pretrained_encoder_model.load_weights('pretrained_encoder_model_weights.h5')
853
854 encoder_model = Model(autoencoder_inputs, autoencoder_state)
855 decoder_state_input = Input(shape=(GRU_UNITS,))
856 decoder_inputs = Input(shape=(1,))
857 decoder_embedding_inference = Embedding(input_dim=fr_vocab_size, output_dim=GRU_UNITS)(decoder_inputs)
858 decoder_gru_inference = GRU(GRU_UNITS, return_sequences=True, return_state=True)
859 decoder_outputs_inference, decoder_state_inference = decoder_gru_inference(decoder_embedding_inference, initial_state=decoder_state_input)
860 decoder_outputs_inference = decoder_dense(decoder_outputs_inference)
861 decoder_model = Model([decoder_inputs, decoder_state_input], [decoder_outputs_inference, decoder_state_inference])
862
863 def translate_sentence(input_text):
864     stop_crit = len(input_text)+3
865     input_text = txt_pre_processing(txt=input_text)
866     input_seq = eng_tokenizer.texts_to_sequences([input_text])
867     input_seq = pad_sequences(input_seq, maxlen=max_seq_length, padding='post')
868     input_seq = tf.ragged.constant(input_seq)
869     states_value = encoder_model.predict(input_seq)
870
871     target_seq = tf.constant([fr_tokenizer.word_index[start_token]])
872     target_text = []
873     stop_condition = False
874     prev_token_index = None
875
876     while not stop_condition:
877         output_tokens, h = decoder_model.predict([target_seq, states_value])
878         sampled_token_index = np.argmax(output_tokens[0, -1, :])
879         if (sampled_token_index == 0):
880             sampled_word = ''
881         else:
882             sampled_word = fr_tokenizer.index_word[sampled_token_index]
883

```

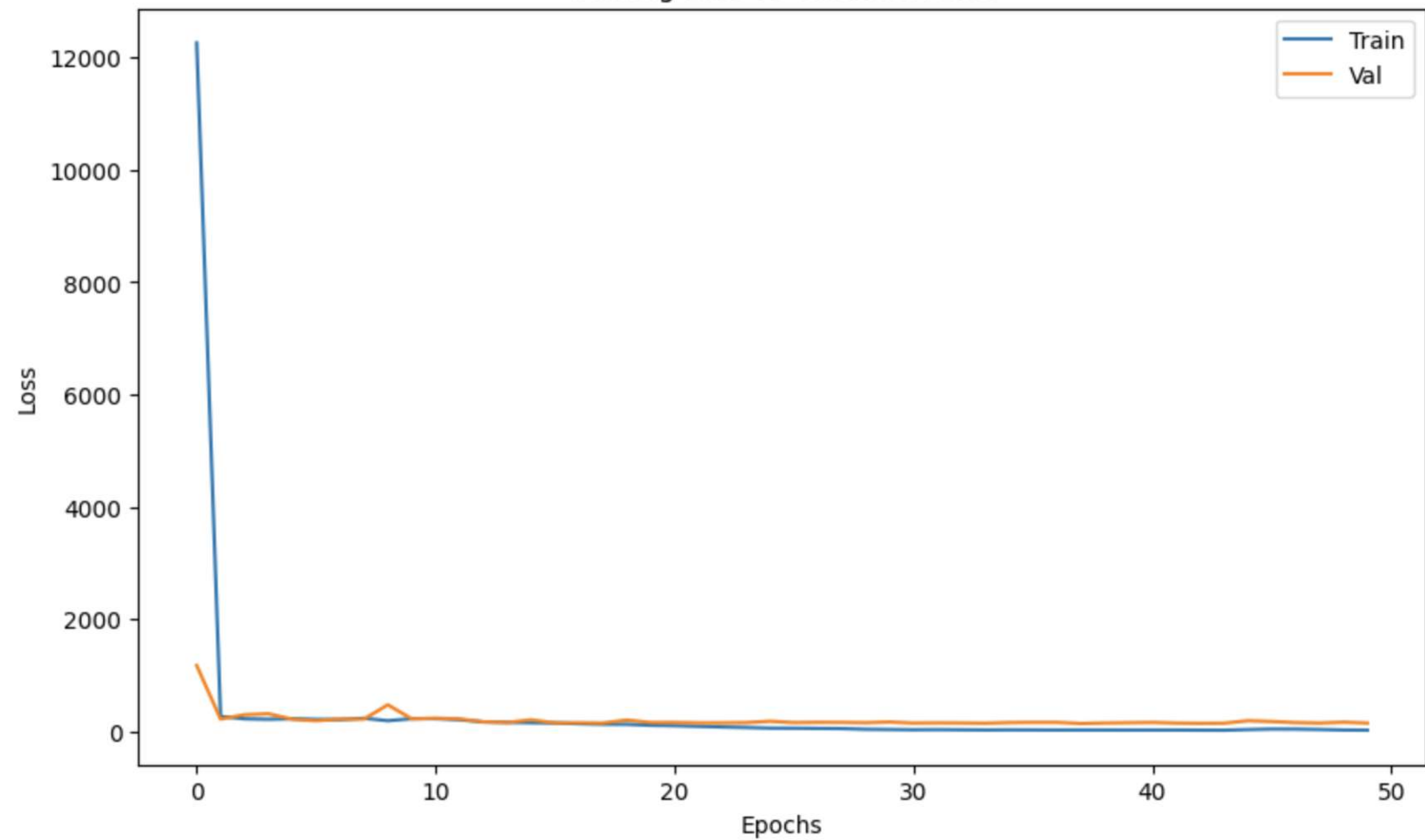
```

884     if (sampled_word != end_token) and (sampled_word != ' '):
885         target_text.append(sampled_word)
886
887     if sampled_word == end_token or len(target_text) >= stop_crit:
888         stop_condition = True
889
890     prev_token_index = sampled_token_index
891     target_seq = tf.constant([sampled_token_index])
892     states_value = h
893     return ' '.join(target_text)
894 input_text = "I won!"
895 translation = translate_sentence(input_text)
896 del decoder_model
897
898 # =====
899 # only dec
900 # =====
901
902 translation_decoder_inputs = tf.keras.layers.Input(shape=(None,))
903 decoder_embedding = Embedding(input_dim=fr_vocab_size, output_dim=GRU_UNITS)(translation_decoder_inputs)
904 decoder_gru = GRU(GRU_UNITS, return_sequences=True, return_state=True)
905 decoder_outputs, _ = decoder_gru(decoder_embedding, initial_state=pretrained_encoder_model.output)
906 decoder_dense = Dense(fr_vocab_size, activation='softmax')
907 translation_decoder_outputs = decoder_dense(decoder_outputs)
908 translation_decoder_model = Model(inputs=[pretrained_encoder_model.input, translation_decoder_inputs], outputs=translation_decoder_outputs)
909 translation_decoder_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
910 split_80_20: int = int(eng_sequences.shape[0]*0.8)
911 X_train, y_train = eng_sequences[:split_80_20,:], fr_sequences[:split_80_20]
912
913 X_test, y_test = eng_sequences[split_80_20:,:], fr_sequences[split_80_20:]
914 y_train = to_categorical(y_train, num_classes=fr_vocab_size)
915 y_test = to_categorical(y_test, num_classes=fr_vocab_size)
916 history = translation_decoder_model.fit([X_train, X_test], y_train, epochs=EPOCHS, batch_size=BATCH_SIZE, validation_data=([X_test, X_test], y_test), callbacks=[EarlyStopping(patience=6)])
917 fig, axs = plt.subplots(2, 1, figsize=(10,13))
918 axs[0].plot(history.history['loss'])
919 axs[0].plot(history.history['val_loss'])
920 axs[0].title.set_text('Decoder Only Training Loss vs Validation Loss')
921 axs[0].set_xlabel('Epochs')
922 axs[0].set_ylabel('Loss')
923 axs[0].legend(['Train', 'Val'])
924 axs[1].plot(history.history['accuracy'])
925 axs[1].plot(history.history['val_accuracy'])
926 axs[1].title.set_text('Decoder Only Training Accuracy vs Validation Accuracy')
927 axs[1].set_xlabel('Epochs')
928 axs[1].set_ylabel('Accuracy')
929 axs[1].legend(['Train', 'Val'])
930
931 encoder_model = Model(autoencoder_inputs, autoencoder_state)
932 decoder_state_input = Input(shape=(GRU_UNITS,))
933 decoder_inputs = Input(shape=(1,))
934 decoder_embedding_inference = Embedding(input_dim=fr_vocab_size, output_dim=GRU_UNITS)(decoder_inputs)
935 decoder_gru_inference = GRU(GRU_UNITS, return_sequences=True, return_state=True)
936 decoder_outputs_inference, decoder_state_inference = decoder_gru_inference(decoder_embedding_inference, initial_state=decoder_state_input)
937 decoder_outputs_inference = decoder_dense(decoder_outputs_inference)
938 decoder_model = Model([decoder_inputs, decoder_state_input], [decoder_outputs_inference, decoder_state_inference])
939
940 def translate_sentence(input_text):
941     stop_crit = len(input_text)+3
942     input_text = txt_pre_processing(txt=input_text)
943     input_seq = eng_tokenizer.texts_to_sequences([input_text])
944     input_seq = pad_sequences(input_seq, maxlen=max_seq_length, padding='post')
945     input_seq = tf.ragged.constant(input_seq)
946     states_value = encoder_model.predict(input_seq)
947
948     target_seq = tf.constant([fr_tokenizer.word_index[start_token]])
949     target_text = []
950     stop_condition = False
951     prev_token_index = None
952
953     while not stop_condition:
954         output_tokens, h = decoder_model.predict([target_seq, states_value])
955         sampled_token_index = np.argmax(output_tokens[0, -1, :])
956         if (sampled_token_index == 0):
957             sampled_word = ' '

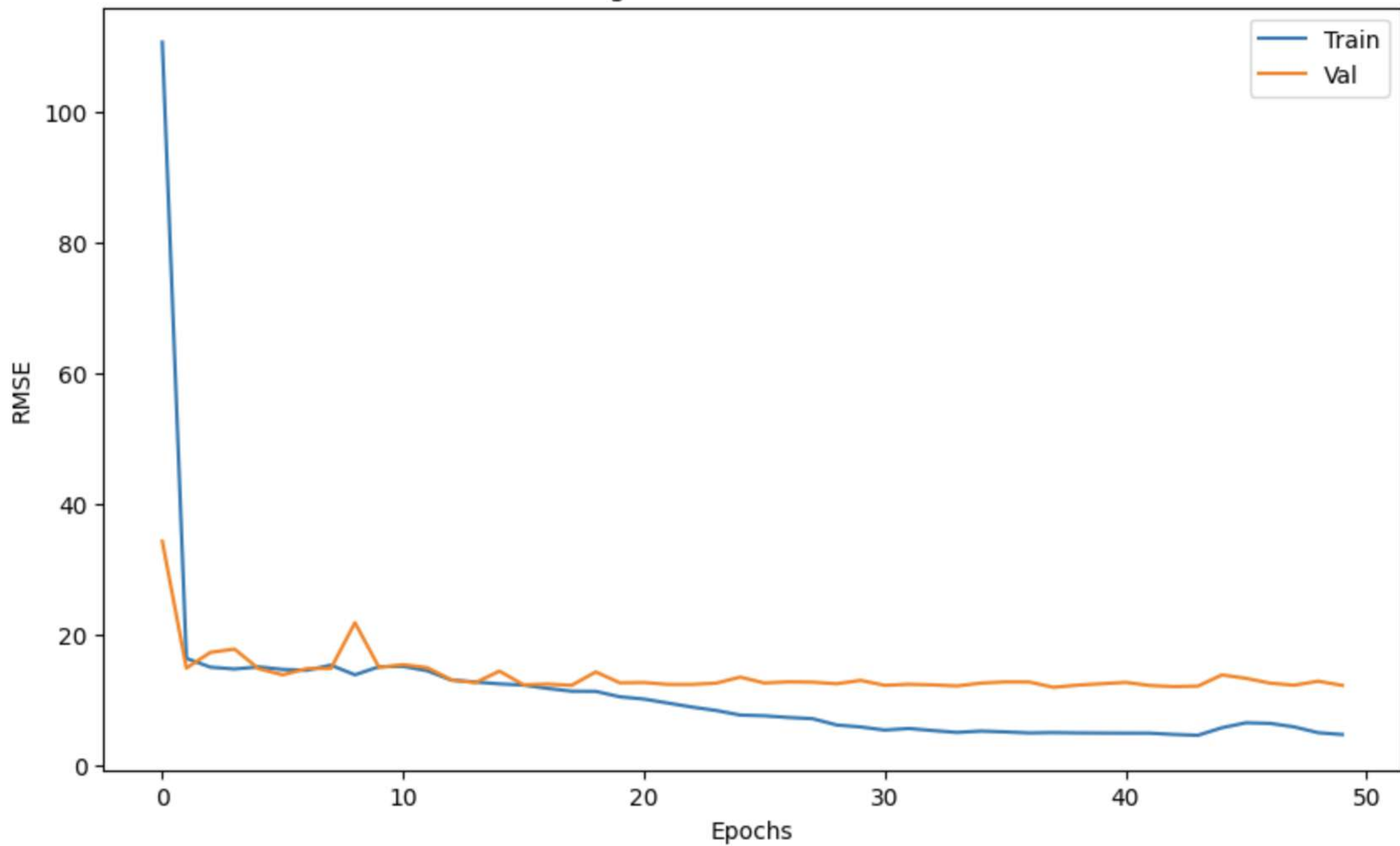
```

```
957         sampled_word =  
958     else:  
959         sampled_word = fr_tokenizer.index_word[sampled_token_index]  
  
960  
961     if (sampled_word != end_token) and (sampled_word != ' '):  
962         target_text.append(sampled_word)  
963  
964     if sampled_word == end_token or len(target_text) >= stop_crit:  
965         stop_condition = True  
966  
967     prev_token_index = sampled_token_index  
968     target_seq = tf.constant([sampled_token_index])  
969     states_value = h  
970     return ''.join(target_text)  
971 input_text = "I won!"  
972 translation = translate_sentence(input_text)  
973 del decoder_model
```

Training Loss vs Validation Loss

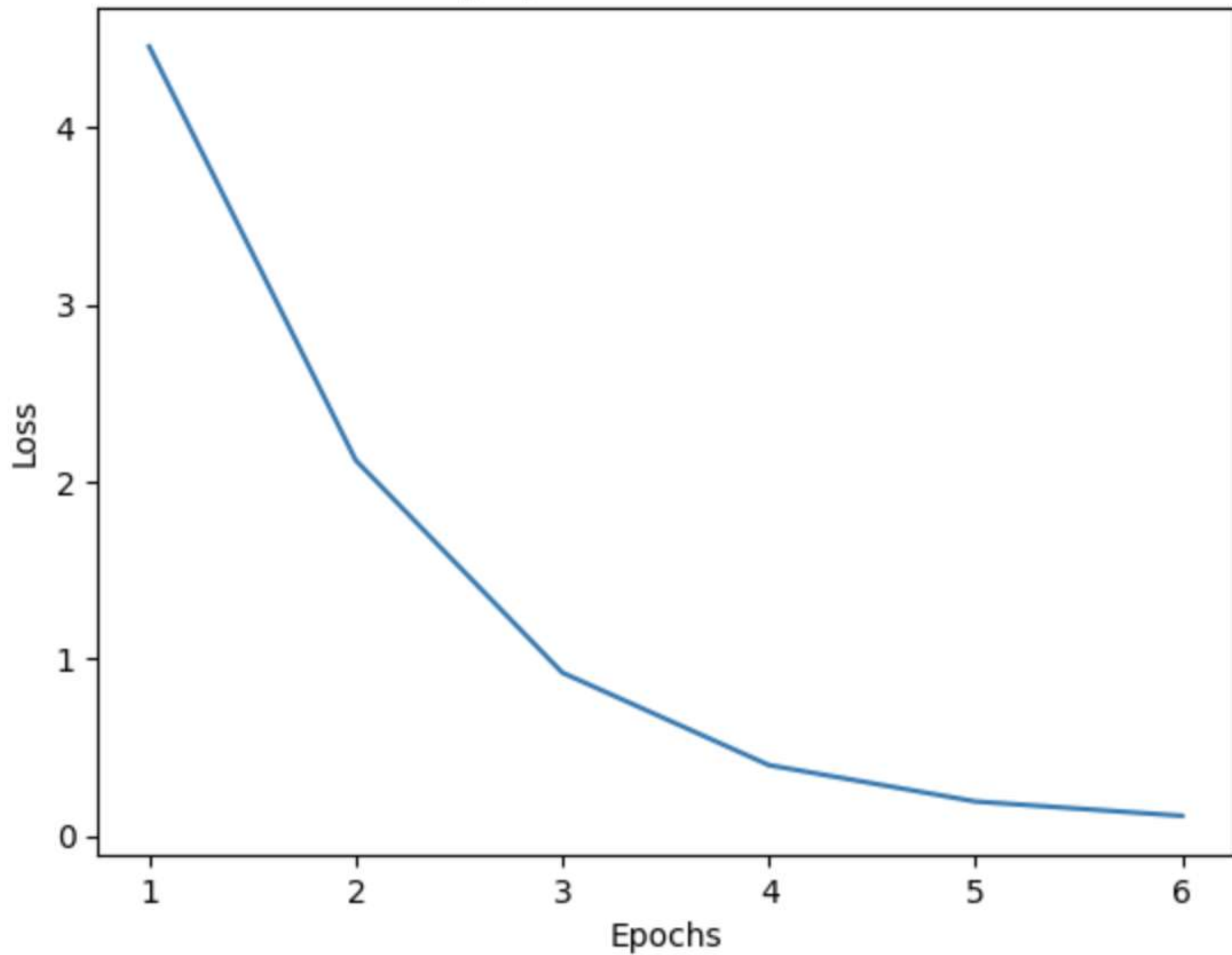


Training RMSE vs Validation RMSE

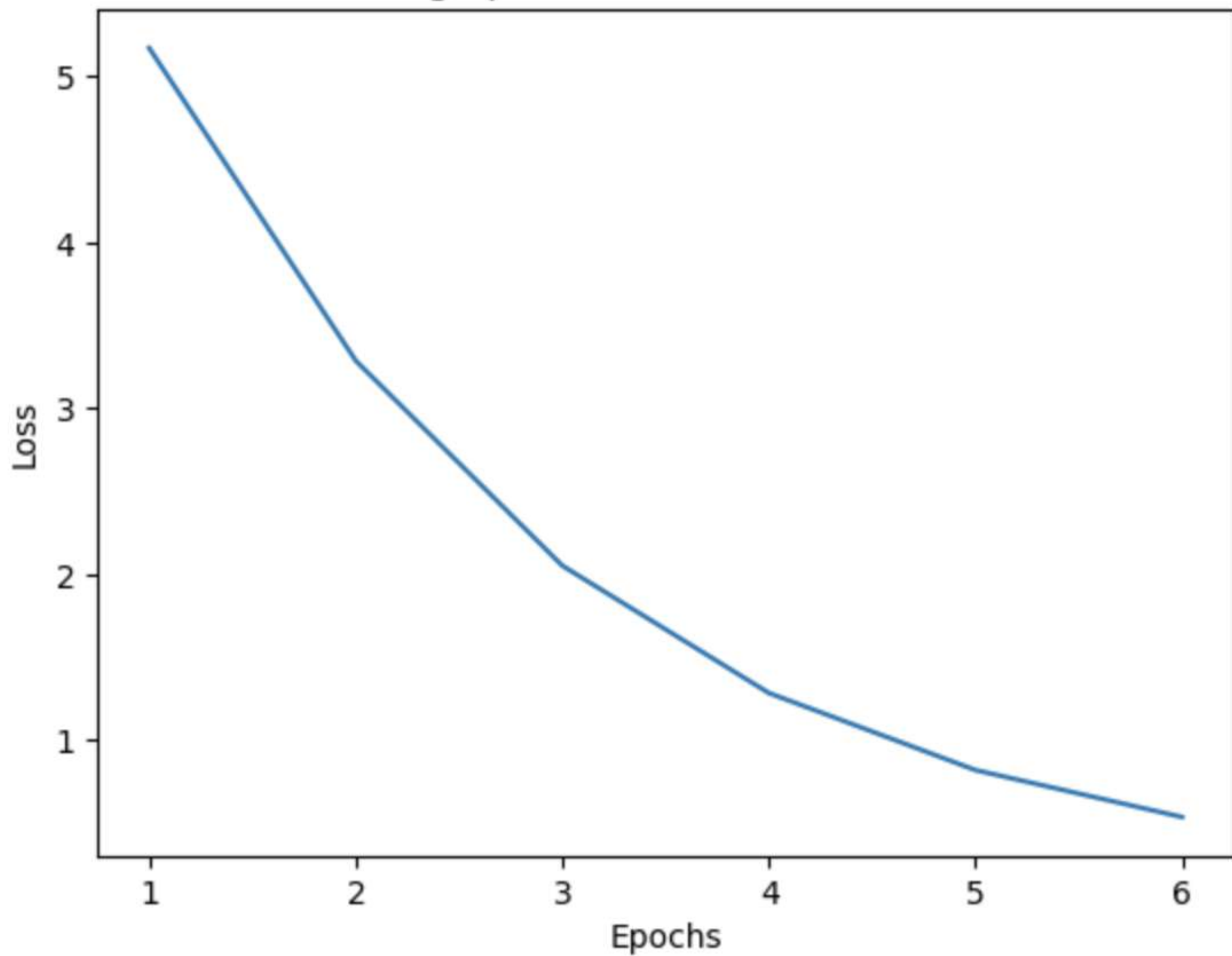



```
Epoch 50/50
165/165 [=====] - 7s 44ms/step - loss: 21.4251 - root_mean_squared_error: 4.6287 - val_loss: 147.6750 - val_root_mean_squared_error: 12.1522
47/47 [=====] - 1s 12ms/step - loss: 141.9804 - root_mean_squared_error: 11.9156
Test Loss: 141.9804, Test RMSE: 11.9156
<matplotlib.legend.Legend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

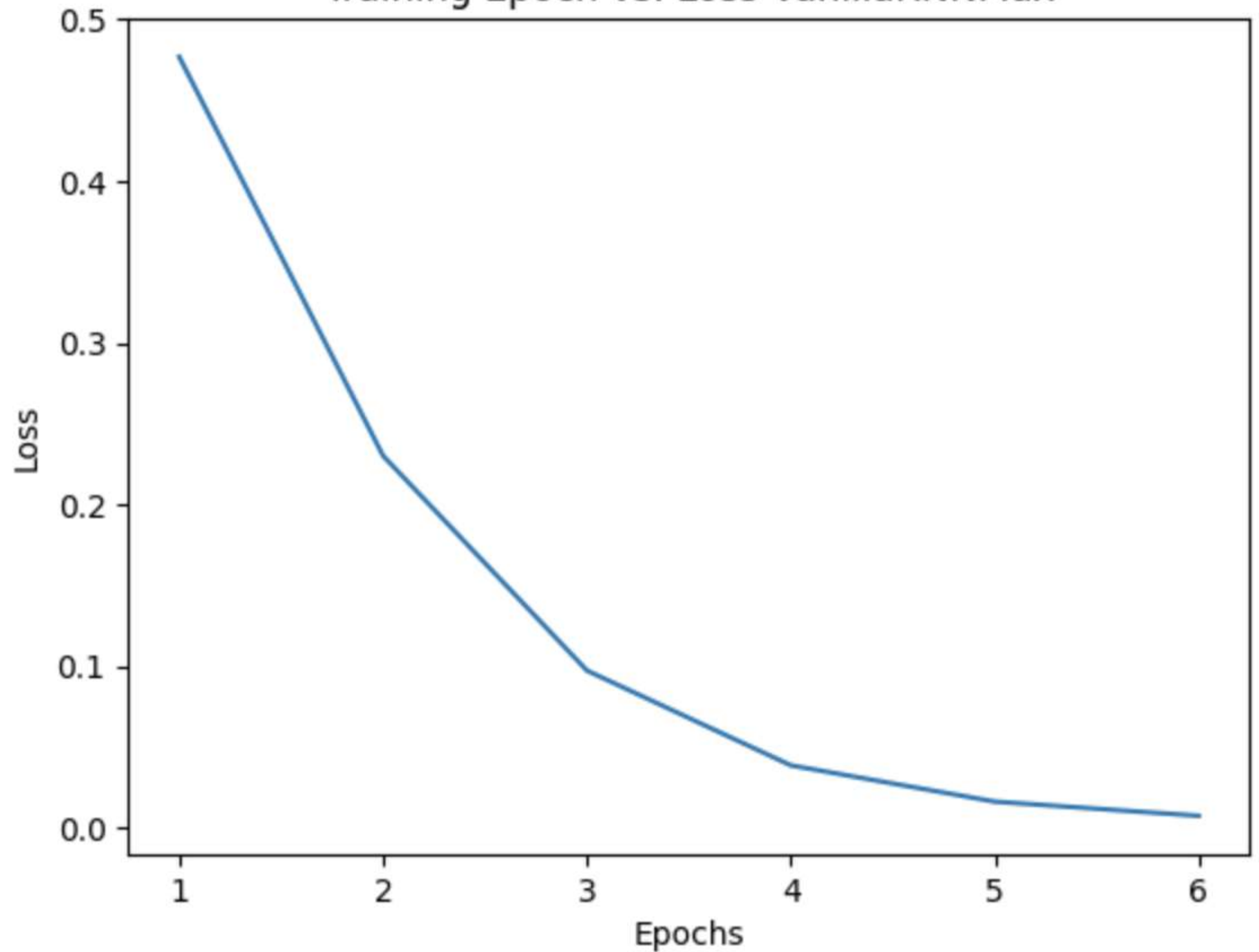
Training Epoch vs. Loss VanillaRNN



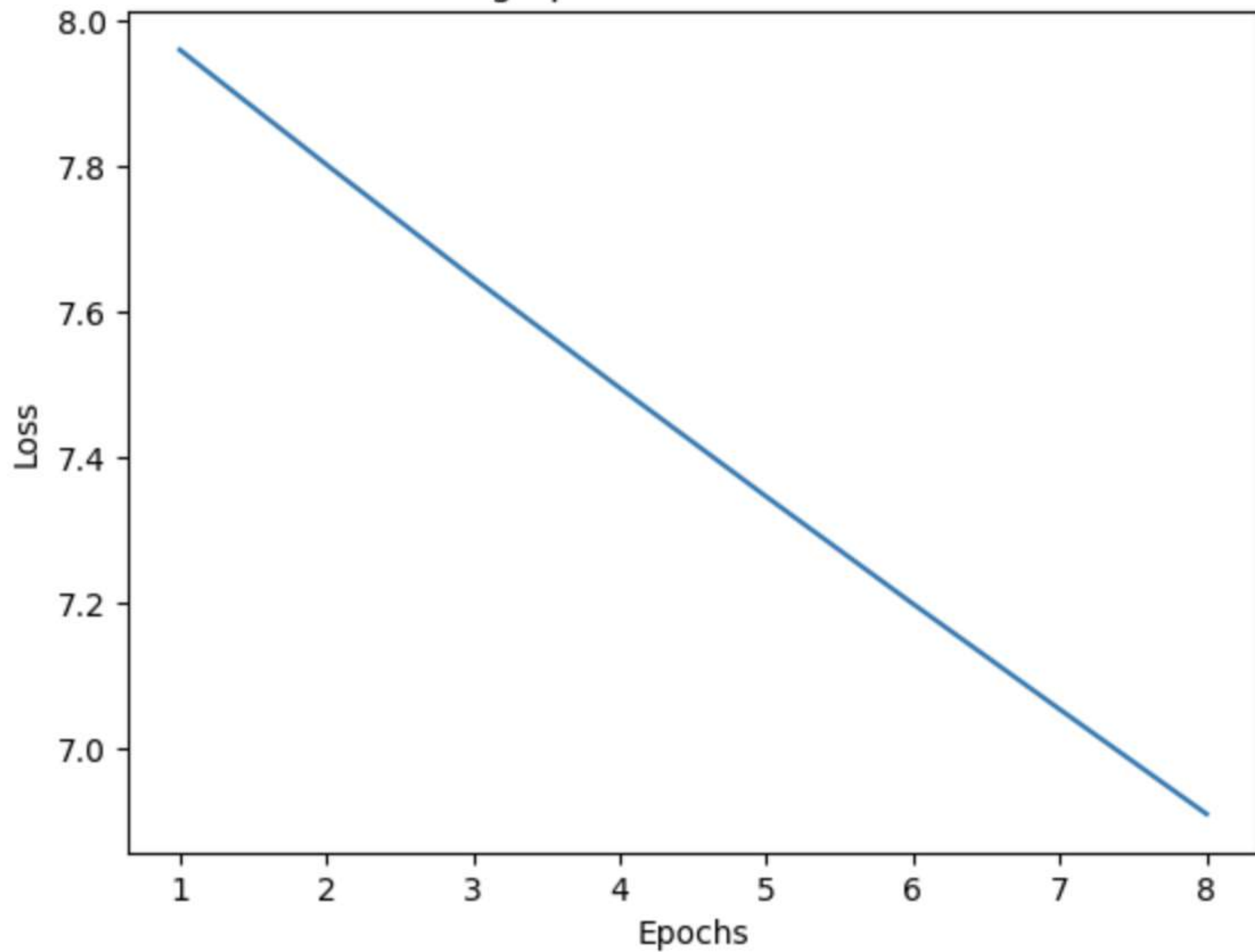
Training Epoch vs. Loss VanillaRNNMin



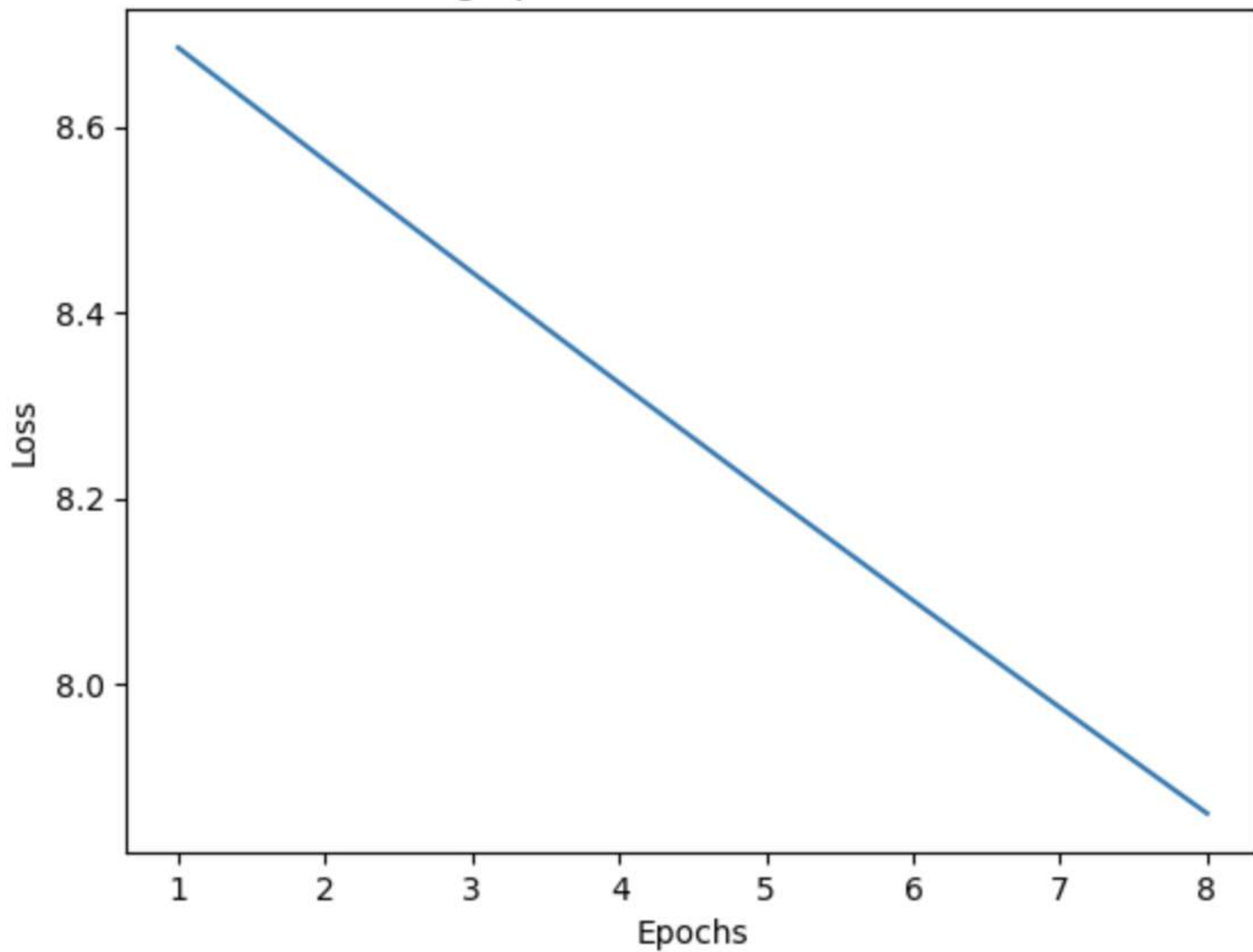
Training Epoch vs. Loss VanillaRNNMax



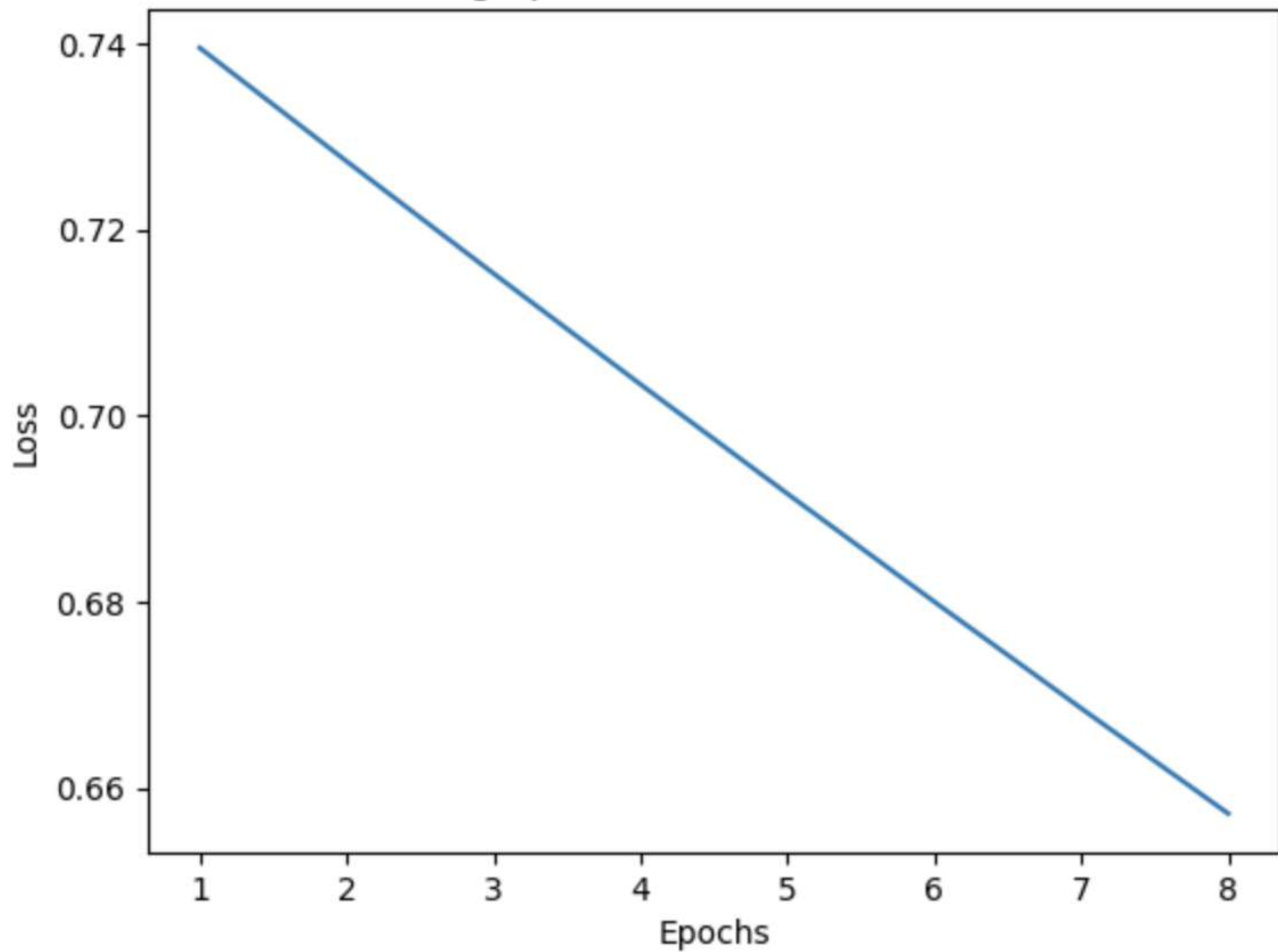
Training Epoch vs. Loss VanillaRNN



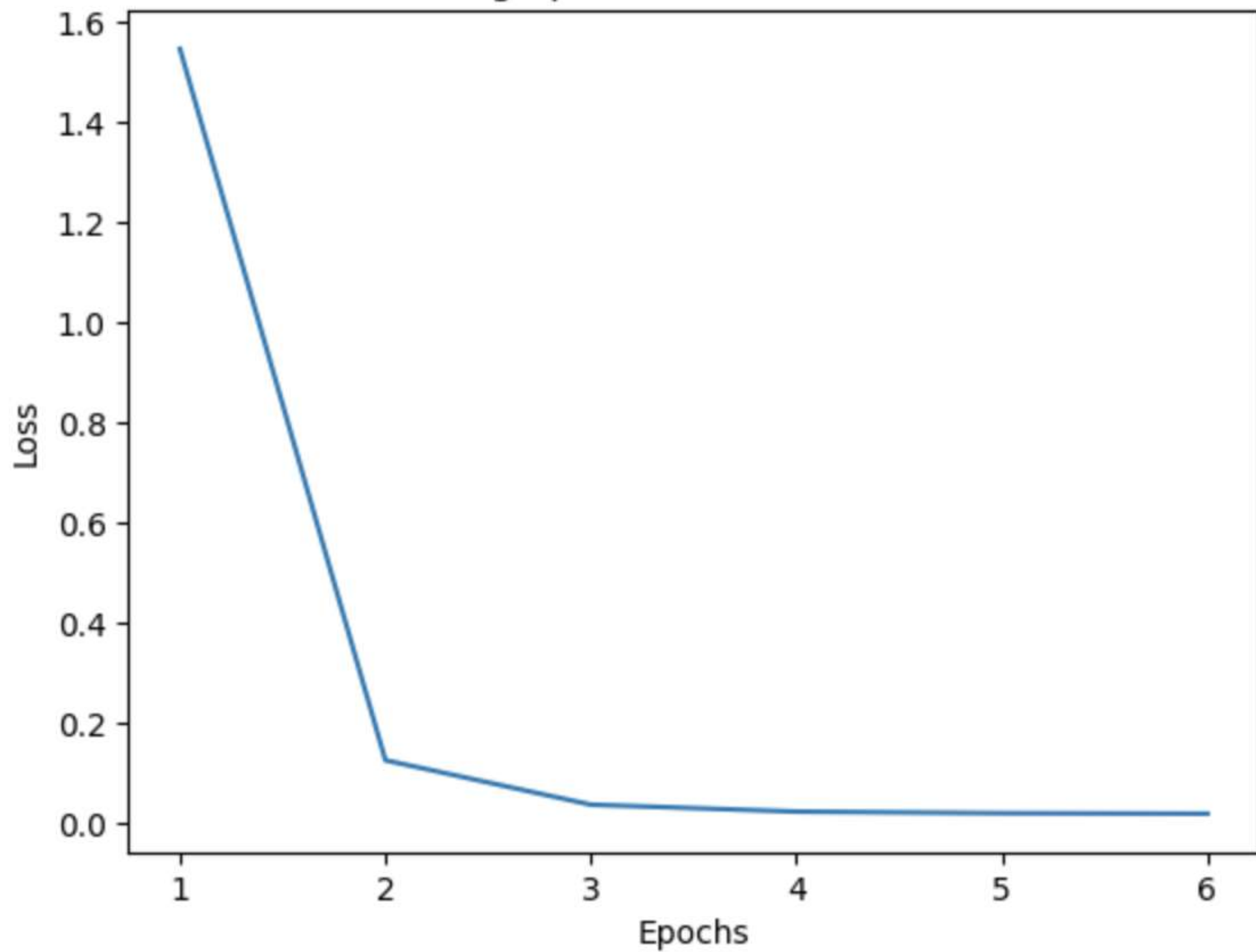
Training Epoch vs. Loss VanillaRNNMin



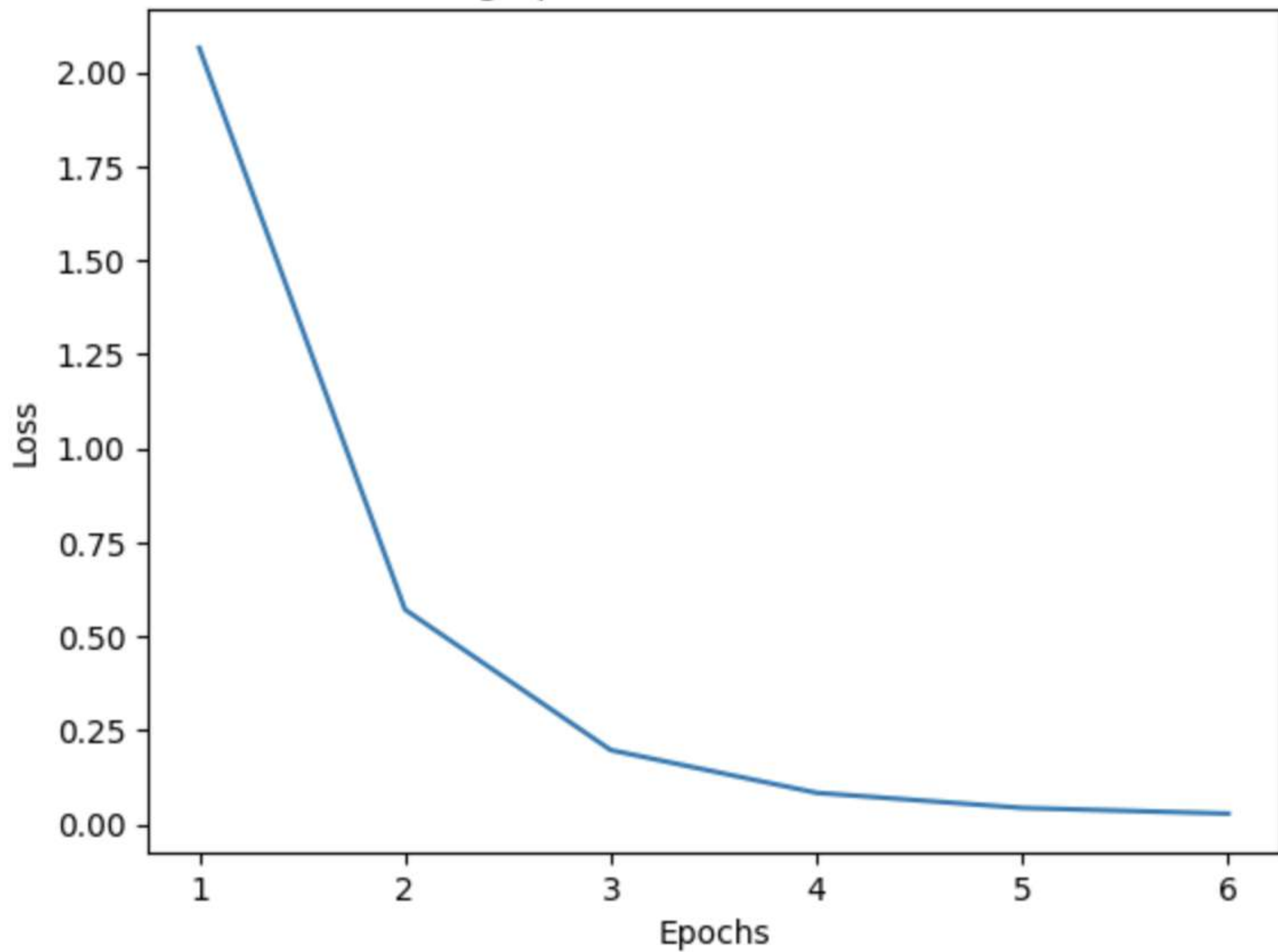
Training Epoch vs. Loss VanillaRNNMax



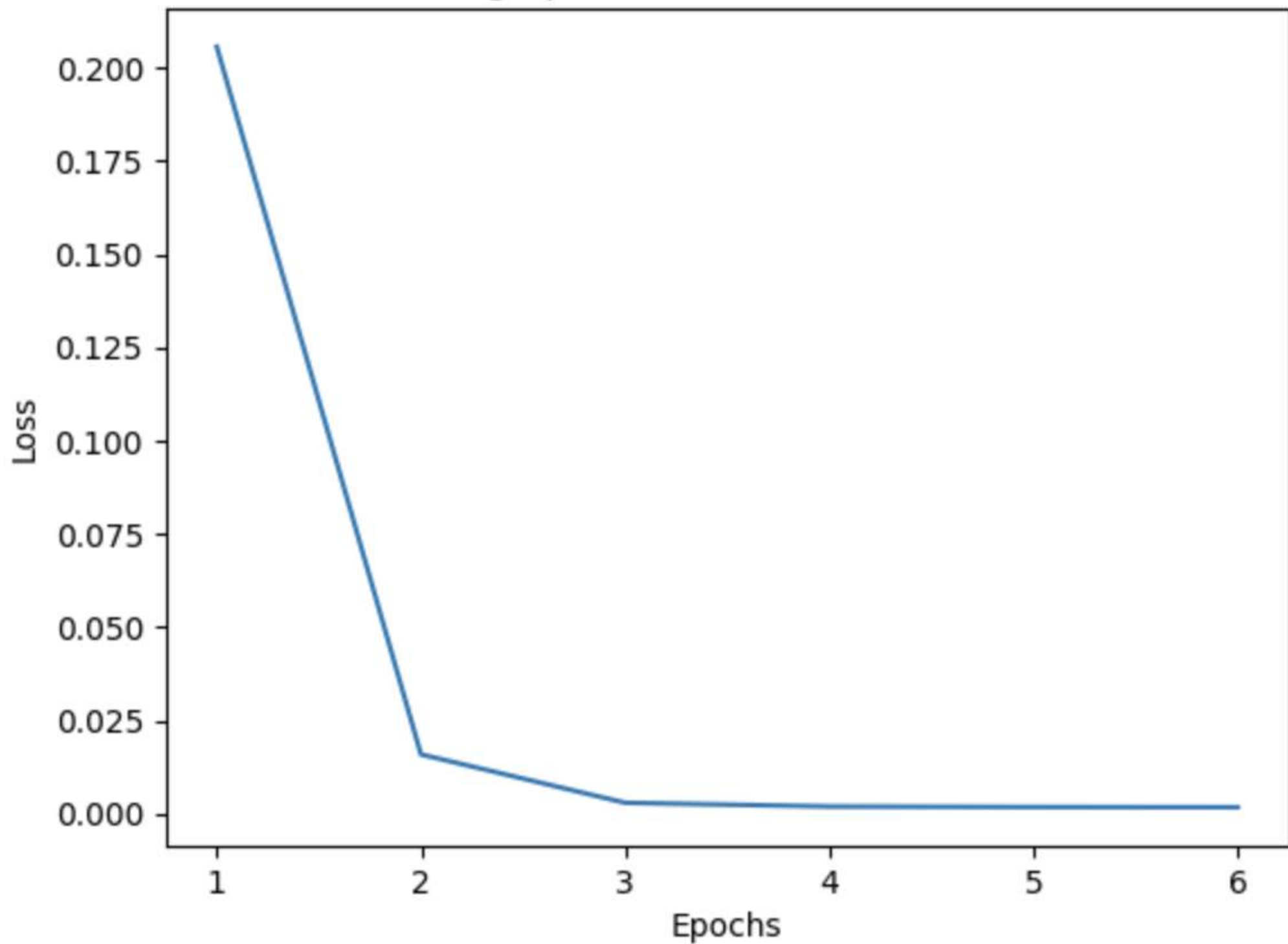
Training Epoch vs. Loss VanillaRNN



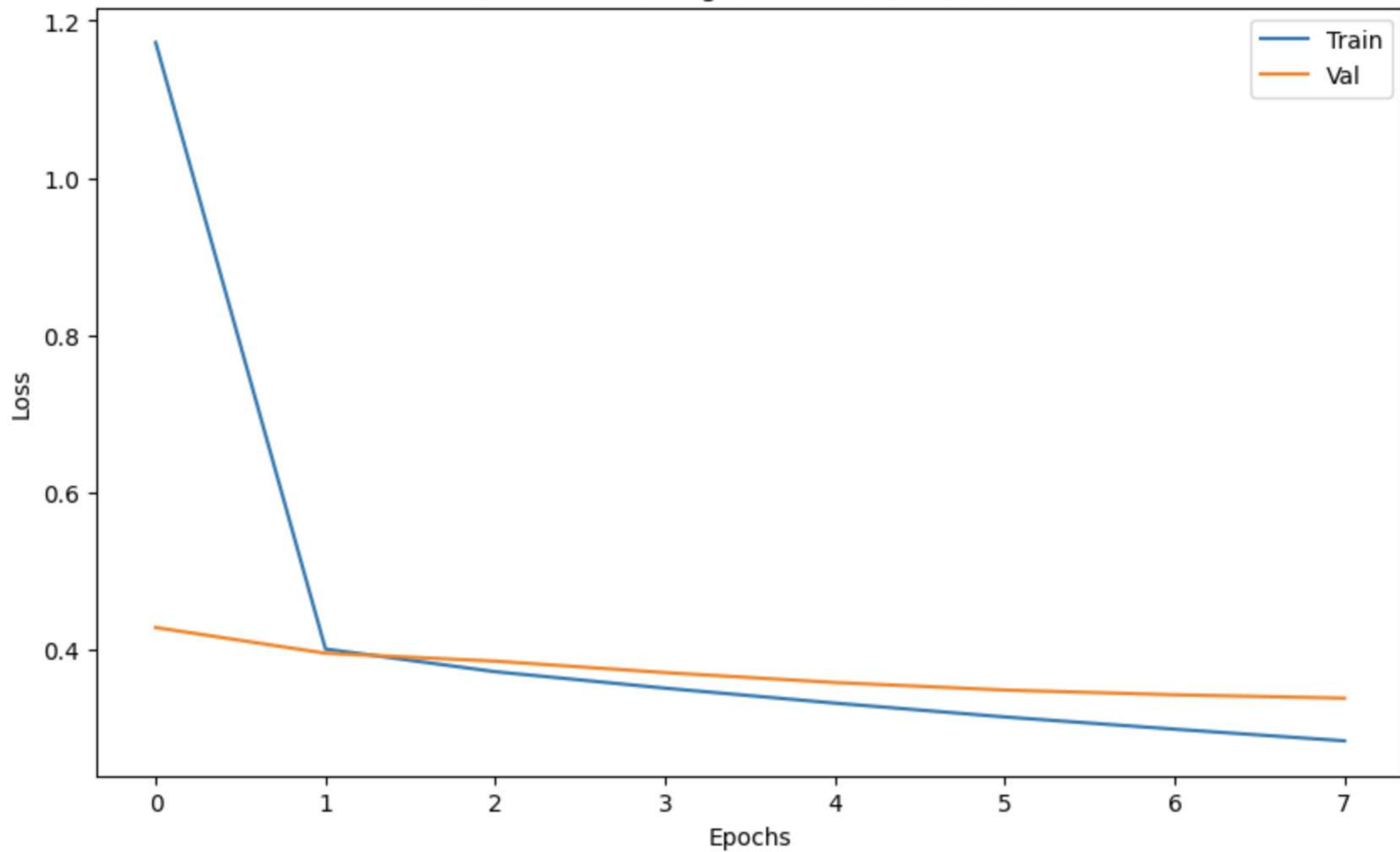
Training Epoch vs. Loss VanillaRNNMin



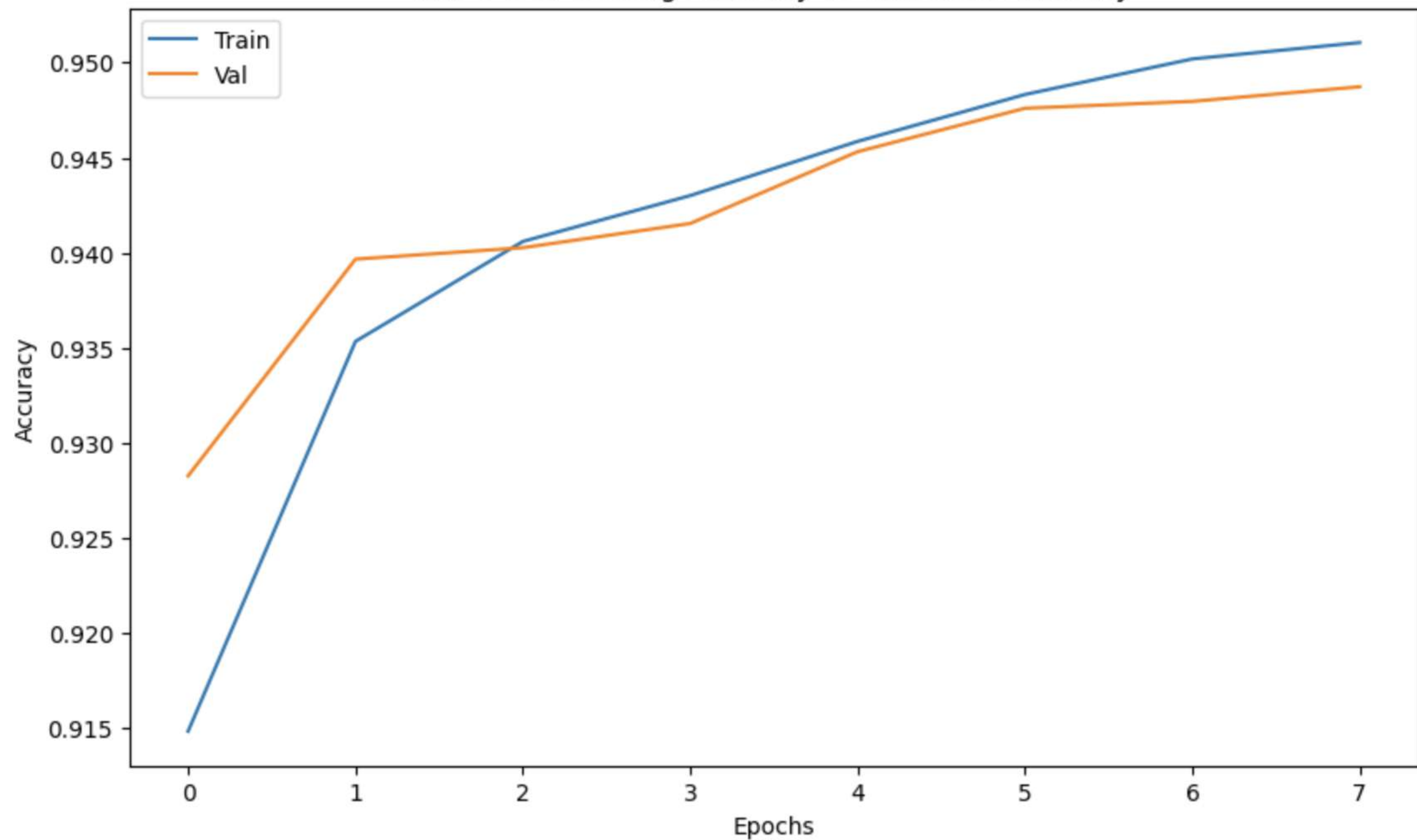
Training Epoch vs. Loss VanillaRNNMax



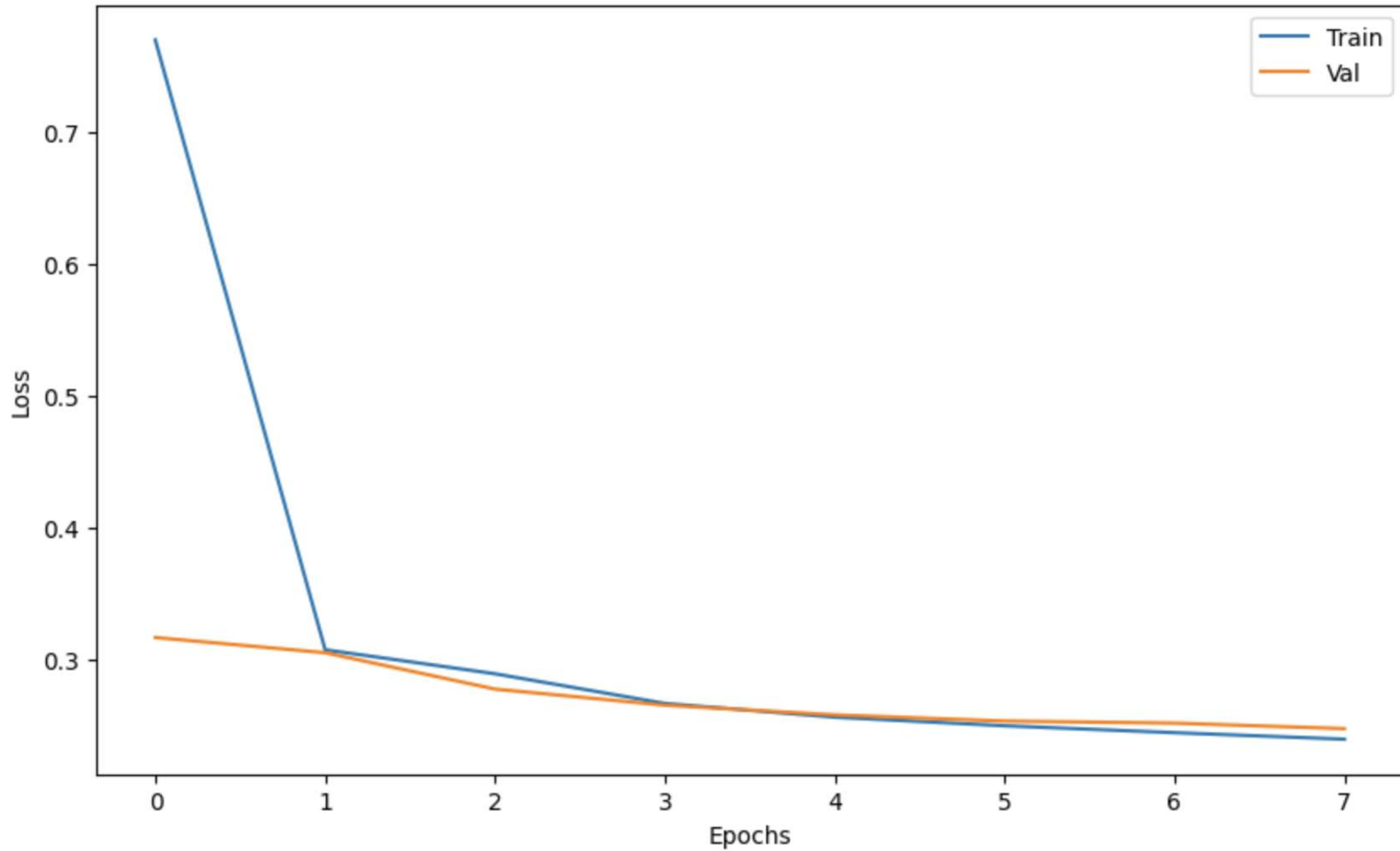
Enc + Dec Training Loss vs Validation Loss



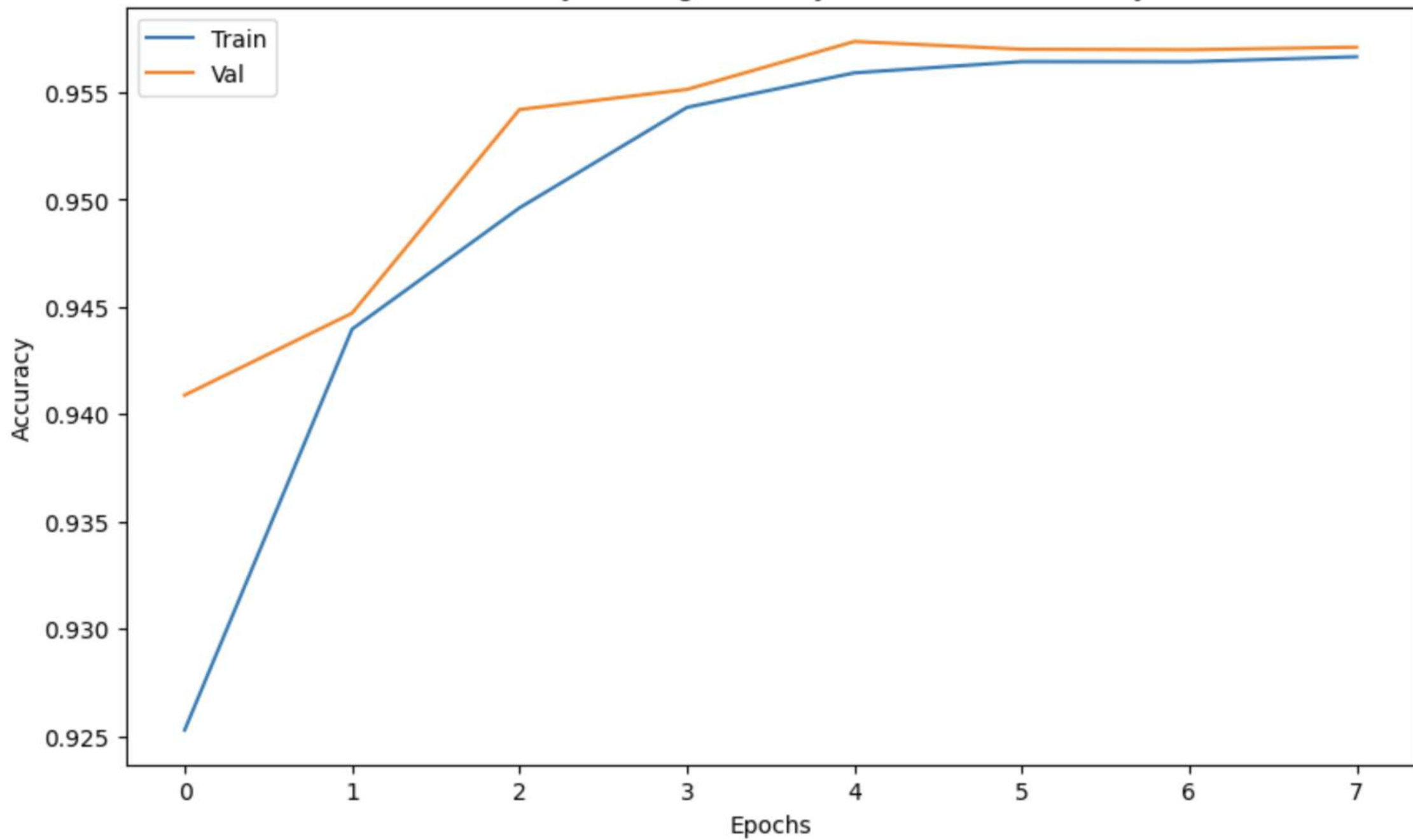
Enc + Dec Training Accuracy vs Validation Accuracy



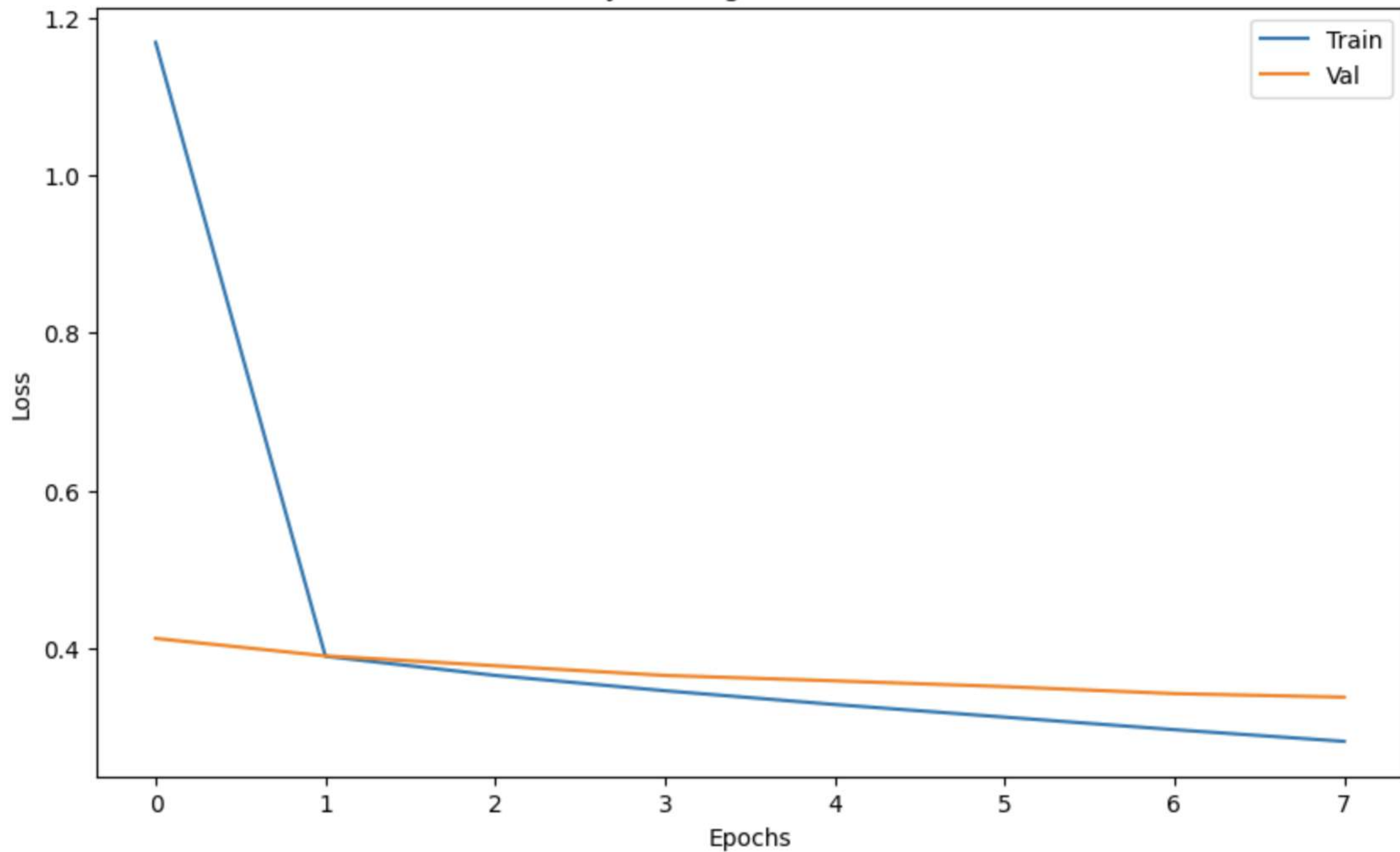
Encoder Only Training Loss vs Validation Loss



Encoder Only Training Accuracy vs Validation Accuracy



Decoder Only Training Loss vs Validation Loss



Decoder Only Training Accuracy vs Validation Accuracy

