Assignment 3

NLP

PREPARED BY

Mohsen Fayyaz - 810100524

Spring 2022

فهرست مطالب

1 تعیین نقش کلمات

الف)

ابتدا دیتاست را با این دو حالت بارگذاری و تفاوت این دو را نشان می‌دهیم.

```
1  nltk.corpus.treebank.tagged_sents(tagset='universal')[0]
```

```
[('Pierre', 'NOUN'),
 ('Vinken', 'NOUN'),
 (',', '.'),
 ('61', 'NUM'),
 ('years', 'NOUN'),
 ('old', 'ADJ'),
 (',', '.'),
 ('will', 'VERB'),
 ('join', 'VERB'),
 ('the', 'DET'),
 ('board', 'NOUN'),
 ('as', 'ADP'),
 ('a', 'DET'),
 ('nonexecutive', 'ADJ'),
 ('director', 'NOUN'),
 ('Nov.', 'NOUN'),
 ('29', 'NUM'),
 ('.', '.')]
```

```
1  nltk.corpus.treebank.tagged_sents()[0]
```

```
[('Pierre', 'NNP'),
 ('Vinken', 'NNP'),
 (',', ','),
 ('61', 'CD'),
 ('years', 'NNS'),
 ('old', 'JJ'),
 (',', ','),
 ('will', 'MD'),
 ('join', 'VB'),
 ('the', 'DT'),
 ('board', 'NN'),
 ('as', 'IN'),
 ('a', 'DT'),
 ('nonexecutive', 'JJ'),
 ('director', 'NN'),
 ('Nov.', 'NNP'),
 ('29', 'CD'),
 ('.', '.')]
```

همانطور که دیده می‌شود، تگهای universal کلی تر و ساده تر هستند. مثلا در حالت عادی یک کلمه ممکن است NNP یا NN بخورد، اما در مثال می بینیم که هر دو در حالت universal که ساده تر است شده اند NOUN. در مجموع چندین دیتاست این کتابخانه از حالت ساده و universal پشتیبانی می‌کنند که یعنی مثلا تمام انواع تگهای اسم در NOUN تجمیع می‌شوند. لیست تگهای universal که کم و ساده هستند در این لینک قابل مشاهده است.

https://universaldependencies.org/u/pos

در ادامه همانطور که گفته شده است برای سادگی از حالت universal استفاده می‌کنیم.

ب)

برای این بخش یک کلاس مجزا برای مدیریت دیتاست می‌سازیم.

```
class DatasetManager:
    def __init__(self):
        dataset = nltk.corpus.treebank.tagged_sents(tagset='universal')
        train_set, test_set = train_test_split(dataset, test_size=0.15, random_state=42)
        train_set, dev_set = train_test_split(dataset, test_size=0.15, random_state=42)
        self.dataset = dict()
        self.dataset["x_train"] = [[d[0] for d in data] for data in train_set]
        self.dataset["y_train"] = [[d[1] for d in data] for data in train_set]
        self.dataset["x_dev"] = [[d[0] for d in data] for data in dev_set]
        self.dataset["y_dev"] = [[d[1] for d in data] for data in dev_set]
        self.dataset["x_test"] = [[d[0] for d in data] for data in test_set]
        self.dataset["y_test"] = [[d[1] for d in data] for data in test_set]
        self.vocab = set([word for sentence in self.dataset["x_train"] for word in sentence])
        self.tags = set([tag for sentence in self.dataset["y_train"] for tag in sentence])

dataset_manager = DatasetManager()
```

15 درصد کل دیتا را به تست اختصاص می‌دهیم و از 85 درصد باقیمانده دوباره 15 درصد را برای validation و باقی را برای آموزش می‌گذاریم. چون دیتاست نسبتا کوچک است، این نسبت ها عددهای مناسبی هستند.

پ)

ابتدا سودوکد الگوریتم viterbi را طبق صفحه 12 فصل 8 کتاب مرجع می‌آوریم.

function VITERBI(*observations* of len *T*,*state-graph* of len *N*) **returns** *best-path*, *path-prob*

create a path probability matrix *viterbi[N,T]*
for each state *s* **from** 1 **to** *N* **do** ; initialization step
  $viterbi[s,1] \leftarrow \pi_s * b_s(o_1)$
  $backpointer[s,1] \leftarrow 0$
for each time step *t* **from** 2 **to** *T* **do** ; recursion step
  for each state *s* **from** 1 **to** *N* **do**
    $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
    $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
$bestpathprob \leftarrow \max_{s=1}^{N} viterbi[s,T]$; termination step
$bestpathpointer \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T]$; termination step
bestpath ← the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
return *bestpath*, *bestpathprob*

Figure 8.10 Viterbi algorithm for finding the optimal sequence of tags. Given an observation sequence and an HMM $\lambda = (A,B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence.

برای این منظور، کلاسی خاص برای الگوریتم ویتربی می‌نویسیم. در ابتدا ماتریس های مورد نیاز که transition تگ ها و emission کلمات و تگها را مشخص می‌کند با استفاده از smoothing می‌سازیم تا کلماتی که دیده نشده اند نیز 0 نشوند. در تابع predict الگوریتم ویتربی اجرا می‌شود و به صورت dynamic programming مسیرهای ممکن بررسی و بهترین آنها با استفاده از back pointer که در طول اجرا می سازیم پیدا می‌شود.

```
class ViterbiHMM:
    def __init__(self, dataset_manager: DatasetManager):
        self.dataset_manager = dataset_manager
        self.START = "<s>"
        self.UNKNOWN = "<unk>"
        # emission:   [word][tag]  -> p(w|t) = c(w,t) / c(t)
        self.emission_matrix = {w: {t: 0 for t in self.dataset_manager.tags} for w in self.dataset_manager.vocab}
        self.emission_matrix[self.UNKNOWN] = {t: 0 for t in self.dataset_manager.tags}
        # transition: [tag1][tag2] -> p(t2|t1) = c(t1, t2) / c(t1)
        self.transition_matrix = {t1: {t2: 0 for t2 in self.dataset_manager.tags} for t1 in self.dataset_manager.tags}
        self.transition_matrix[self.START] = {t: 0 for t in self.dataset_manager.tags}
        self.tag_count = dict()

    def __fill_tag_count(self):
        self.tag_count = {k: 0 for k in self.dataset_manager.tags}
        self.tag_count[self.START] = 0
        for sentence_tags in tqdm(self.dataset_manager.dataset["y_train"], desc="tag count"):
            for tag in sentence_tags:
                self.tag_count[tag] += 1
            self.tag_count[self.START] += 1

    def __fill_emission_matrix(self, smoothing=True):
        for data_idx in tqdm(range(len(self.dataset_manager.dataset["x_train"])), desc="emission"):
            words = self.dataset_manager.dataset["x_train"][data_idx]
```

```
            tags = self.dataset_manager.dataset["y_train"][data_idx]
            for word, tag in zip(words, tags):
                self.emission_matrix[word][tag] += 1

        for tag in self.dataset_manager.tags:
            for word in self.dataset_manager.vocab:
                if smoothing:
                    self.emission_matrix[word][tag] = (self.emission_matrix[word][tag] + 1) / (self.tag_count[tag] +
len(self.dataset_manager.vocab))
                    self.emission_matrix[self.UNKNOWN][tag] = (0 + 1) / (self.tag_count[tag] + len(self.dataset_manager.vocab))
                else:
                    self.emission_matrix[word][tag] = self.emission_matrix[word][tag] / self.tag_count[tag]
                    self.emission_matrix[self.UNKNOWN][tag] = 0

    def __fill_transition_matrix(self, smoothing=True):
        for sentence_tags in tqdm(self.dataset_manager.dataset["y_train"], desc="transition"):
            for i in range(0, len(sentence_tags) - 1):
                tag1 = sentence_tags[i]
                tag2 = sentence_tags[i + 1]
                self.transition_matrix[tag1][tag2] += 1
                if i == 0:
                    self.transition_matrix[self.START][tag1] += 1

        for tag1 in self.transition_matrix.keys():
            for tag2 in self.transition_matrix[tag1].keys():
                if smoothing:
                    self.transition_matrix[tag1][tag2] = (self.transition_matrix[tag1][tag2] + 1) / (self.tag_count[tag1] +
len(self.dataset_manager.tags))
                else:
                    self.transition_matrix[tag1][tag2] = self.transition_matrix[tag1][tag2] / self.tag_count[tag1]

    def train(self):
        self.__fill_tag_count()
        self.__fill_transition_matrix()
        self.__fill_emission_matrix()

    def predict(self, sentence: list):  # Viterbi Algorithm
        viterbi = [{t: 0 for t in self.dataset_manager.tags} for w_idx in range(len(sentence))]
        back_pointer = [{t: 0 for t in self.dataset_manager.tags} for w_idx in range(len(sentence))]
        for tag in self.dataset_manager.tags:
            word = sentence[0]
            emission_prob = self.emission_matrix[self.UNKNOWN][tag] if word not in self.emission_matrix else
self.emission_matrix[word][tag]
            viterbi[0][tag] = self.transition_matrix[self.START][tag] * emission_prob

        for word_idx in range(1, len(sentence)):
            word = sentence[word_idx]
            prev_word = sentence[word_idx - 1]
            for tag in self.dataset_manager.tags:
                viterbi[word_idx][tag] = 0
                for prev_tag in self.dataset_manager.tags:
                    emission_prob = self.emission_matrix[self.UNKNOWN][tag] if word not in self.emission_matrix else
self.emission_matrix[word][tag]
                    new_viterbi = viterbi[word_idx - 1][prev_tag] * self.transition_matrix[prev_tag][tag] * emission_prob
                    if new_viterbi > viterbi[word_idx][tag]:
                        viterbi[word_idx][tag] = new_viterbi
                        back_pointer[word_idx][tag] = prev_tag

        last_tag_prob = 0
        for tag in self.dataset_manager.tags:
            if viterbi[-1][tag] >= last_tag_prob:
                last_tag_prob = viterbi[-1][tag]
                last_tag = tag

        best_tags = [last_tag]
        for word_idx in range(len(sentence) - 1, 0, -1):
            best_tags.append(back_pointer[word_idx][last_tag])
            last_tag = back_pointer[word_idx][last_tag]
        return list(reversed(best_tags))

    def evaluate(self):
        tp = 0
        total = 0
        for data_idx in tqdm(range(len(self.dataset_manager.dataset["x_test"]))):
            tags = self.predict(self.dataset_manager.dataset["x_test"][data_idx])
            for t_pred, t_true in zip(tags, self.dataset_manager.dataset["y_test"][data_idx]):
                total += 1
                if t_pred == t_true:
                    tp += 1
        return tp / total


viterbi = ViterbiHMM(dataset_manager)
viterbi.train()
viterbi.evaluate()
```

در انتها و با استفاده از تابع evaluate مقدار دقت به دست آمده روی داده تست 0.9026 شد که نشان می‌دهد ای الگوریتم به خوبی توانسته عمل کند و HMM برای این مسئله که نسبتا ساده است می‌تواند به نتایج نسبتا خوبی برسد.

ت)

مثالهای خطا:

```
['Moscow', 'has', 'settled', 'pre-1917', 'debts', 'with', 'other', 'countries', 'in',
'recent', 'years', 'at', 'less', 'than', 'face', 'value', '.']
pred: ['NOUN', 'VERB', 'VERB', 'DET', 'NOUN', 'ADP', 'ADJ', 'NOUN', 'ADP', 'ADJ',
'NOUN', 'ADP', 'ADJ', 'ADP', 'NOUN', 'NOUN', '.']
true: ['NOUN', 'VERB', 'VERB', 'ADJ', 'NOUN', 'ADP', 'ADJ', 'NOUN', 'ADP', 'ADJ',
'NOUN', 'ADP', 'ADJ', 'ADP', 'NOUN', 'NOUN', '.']
```

در این مثال دیده می‌شود که کلمه pre-1917 در اصل ADJ بوده است ولی DET تشخیص داده شده. علت آن می‌تواند باشد که این کلمه خاص است و به احتمال زیاد اصلا در داده آموزشی وجود نداشته. بنابراین با استفاده از smoothing جلوی 0 شدنش گرفته شده اما تنها اطلاعات کمکی این بوده که در transition چقدر مثلا از VERB به DET یا ADJ محتمل است. و می‌توان گفت که معمولا پس از فعل بیشتر DET می‌آید تا صفت. همچنان که در این جمله هم اگر جای این کلمه the قرار دهیم باز معنی به خوبی منتقل می‌شود و مشکلی پیش نمی‌آید. بنابراین حدس مدل طبق اطلاعاتی که داشته منطقی است.

همین استدلال را می‌توان در مثال های دیگر مانند مثال زیر دید که کلمه خاصی به کار رفته و مدل ندیده است و براساس جایگاه و ترتیب تگها هم DET محتمل تر است.

```
['Mortgage-Backed', 'Issues']
pred: ['DET', 'NOUN']
true: ['NOUN', 'NOUN']
['champagne', 'and', 'dessert', 'followed', '.']
pred: ['NOUN', 'CONJ', 'ADJ', 'NOUN', '.']
true: ['NOUN', 'CONJ', 'NOUN', 'VERB', '.']
```

برای اطمینان چک هم شد که کلمات مانند dessert در داده آموزش هستند یا نه که نبودند. بنابراین اکثر اشتباهات ظاهرا از جاهایی است که کلمه دیده نشده و فقط براساس ترتیب تگها باید تصمیم گرفت.

دیتاست بزرگتر در این مسئله قطعا مفید خواهد بود. همچنین استفاده از BPE نیز شاید بتواند کلمات ناشناخته را بشکند و کمکی بکند. البته دیتاست بزرگتر قطعا تاثیر بهتری خواهد داشت.

ث)

برای برخورد با کلمات ناشناخته در تست، ماتریسهایی که ساختیم را با smoothing ساختیم تا 0 در آنها نباشد. به این ترتیب در مسیرهای viterbi هیچگاه 0 ضرب نخواهد شد تا همه چیز 0 شود. و بنابراین بیشتر تمرکز مدل می‌تواند روی ترتیب تگهایی که می‌زند باشد و از آن اطلاعات استفاده کند.

همانطور که در بخشهای قبل هم گفته شد، مهمترین و تاثیرگذارترین راه حل، افزایش اندازه دیتاست است تا کلمات ناشناخته به حداقل برسند. در غیر اینصورت مثلا می‌توان از BPE استفاده کرد تا کلمات ناشناخته به اجزای شناخته شده تر شکسته شوند و کاملا ناشناخته نمانند. از طرفی همین روش laplace smoothing نیز یک راه حل این مسئله است تا احتمال 0 نشود. همچنین در مورد کلاسهای closed و open در درس توضیح داده شد و در اینجا اگر کلمه ای را نمیشناسیم احتمال اینکه از closed ها باشد قطعا کمتر است چون کلماتشان بسته است و اضافه شدن کلمه جدید به آن کم است. بنابراین می‌توان با ضریبی این اطلاعات را در مدل تاثیر داد و مثلا emission کلمه دیده نشده با تگ های closed را بسیار کوچک کرد. روشهای بیشتری در مقاله https://cl.lingfil.uu.se/~nivre/statmet/haulrich.pdf ذکر شده است که می‌توان به آن مراجعه کرد.

ج)

برای پیاده سازی RNN ها کلاس آموزش زیر پیاده سازی شد.

```
class Trainer:
    def __init__(self, dataset_manager: DatasetManager, units=64, rnn="SimpleRNN") -> None:
        self.dataset_manager = dataset_manager
        self.rnn = rnn
        self.tokenizer = Tokenizer(num_words=15000, oov_token="[UNK]")
        self.tokenizer.fit_on_texts(dataset_manager.dataset["x_train"])
        self.tokenized_x_train = pad_sequences(self.tokenizer.texts_to_sequences(dataset_manager.dataset["x_train"]), maxlen=100,
padding="post")
        self.tokenized_x_dev = pad_sequences(self.tokenizer.texts_to_sequences(dataset_manager.dataset["x_dev"]), maxlen=100,
padding="post")
        self.tokenized_x_test = pad_sequences(self.tokenizer.texts_to_sequences(dataset_manager.dataset["x_test"]), maxlen=100,
padding="post")

        self.tag_tokenizer = Tokenizer()
        self.tag_tokenizer.fit_on_texts(dataset_manager.dataset["y_train"])
        self.tokenized_y_train = to_categorical(pad_sequences(self.tag_tokenizer.texts_to_sequences(dataset_manager.dataset["y_train"]),
maxlen=100, padding="post"), num_classes = len(self.dataset_manager.tags) + 1)
        self.tokenized_y_dev = to_categorical(pad_sequences(self.tag_tokenizer.texts_to_sequences(dataset_manager.dataset["y_dev"]),
maxlen=100, padding="post"), num_classes = len(self.dataset_manager.tags) + 1)
        self.tokenized_y_test = to_categorical(pad_sequences(self.tag_tokenizer.texts_to_sequences(dataset_manager.dataset["y_test"]),
maxlen=100, padding="post"), num_classes = len(self.dataset_manager.tags) + 1)

        self.model = self.build_model(units)
        self.history = None

    def build_model(self, units):
        rnn_map = {
```

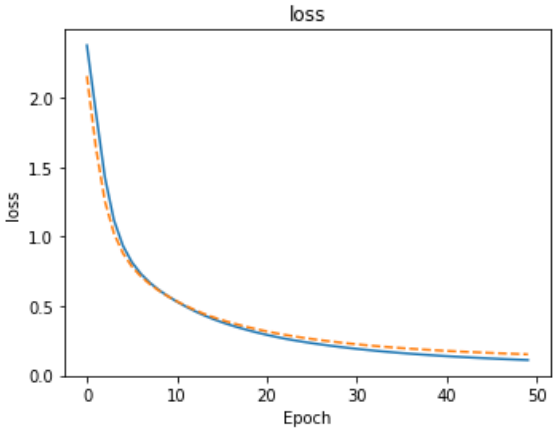
```
            "SimpleRNN": tf.keras.layers.SimpleRNN(units, return_sequences=True),
            "GRU": tf.keras.layers.GRU(units, return_sequences=True),
            "LSTM": tf.keras.layers.LSTM(units, return_sequences=True),
        }
        model = tf.keras.Sequential([
            tf.keras.layers.Embedding(input_dim=15000, output_dim=32, input_length=100),
            rnn_map[self.rnn],
            tf.keras.layers.TimeDistributed(
                tf.keras.layers.Dense(len(self.dataset_manager.tags) + 1,
                                      activation='softmax')
            )
        ])
        model.compile(
            optimizer='adam',
            loss='categorical_crossentropy',
            metrics=['accuracy']
        )
        return model

    def train(self, batch_size=128, epochs=50):
        early_stopping = tf.keras.callbacks.EarlyStopping(
            monitor='val_loss',
            verbose=1,
            patience=5,
            mode='min',
            restore_best_weights=True
        )
        self.history = self.model.fit(
            self.tokenized_x_train, self.tokenized_y_train,
            batch_size=batch_size,
            epochs=epochs,
            verbose=1,
            validation_data=(self.tokenized_x_dev, self.tokenized_y_dev),
            callbacks=[early_stopping],
        )

    def plot_history(self):
        fig = plt.figure(figsize=(12, 4))
        metrics = ['loss', 'accuracy']
        for n, metric in enumerate(metrics):
            plt.subplot(1, 2, n+1)
            plt.plot(self.history.epoch, self.history.history[metric], label='Train')
            plt.plot(self.history.epoch, self.history.history[f"val_{metric}"], linestyle="--", label='Validation')
            plt.xlabel('Epoch')
            plt.ylabel(metric)
            plt.title(metric)
        plt.legend()
        plt.show()

    def evaluate(self):
        [test_loss, test_acc] = self.model.evaluate(self.tokenized_x_test, self.tokenized_y_test)
        print("Test Loss:", test_loss, "Test Accuracy (w/ padding):", test_acc)
        def flatten(t):
            return [item for sublist in t for item in sublist]
        test_preds = np.argmax(self.model.predict(self.tokenized_x_test), axis=-1)
        test_preds = flatten([test_preds[i][:len(dataset_manager.dataset["x_test"][i])] for i in range(len(test_preds))])
        y_test = np.argmax(self.tokenized_y_test, axis=-1)
        y_test = flatten([y_test[i][:len(dataset_manager.dataset["y_test"][i])] for i in range(len(y_test))])
        self.plot_cm(y_test, test_preds)
        print("Test Accuracy (w/o padding):", accuracy_score(y_test, test_preds))

    def plot_cm(self, y_true, preds):
        cm = confusion_matrix(y_true, preds)
        plt.figure(figsize=(7, 5))
        ax = sns.heatmap(cm, annot=True, fmt="d")
        bottom, top = ax.get_ylim()
        ax.set_ylim(bottom + 0.5, top - 0.5)
        plt.title('Confusion Matrix')
        plt.ylabel('True Label')
        plt.xlabel('Predicted Label')
        plt.show()
```

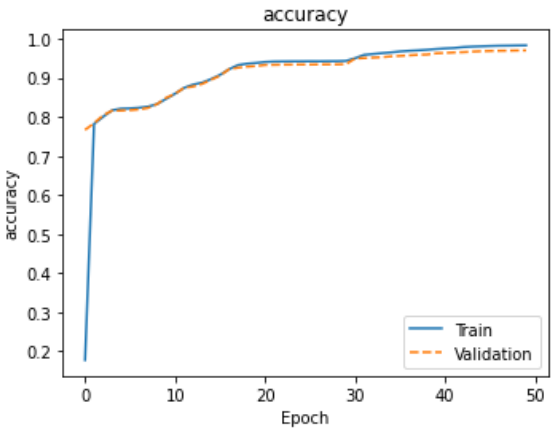
سپس برای اجرای آن با اندازه hidden-layer های مختلف با کد زیر آنرا اجرا میکنیم.

```
for units in [4, 16, 64]:
    print(f"### UNITS={units} ###")
    trainer = Trainer(dataset_manager, units=units)
    trainer.train(batch_size=64, epochs=50)
    print(trainer.model.summary())
    trainer.evaluate()
    trainer.plot_history()
```

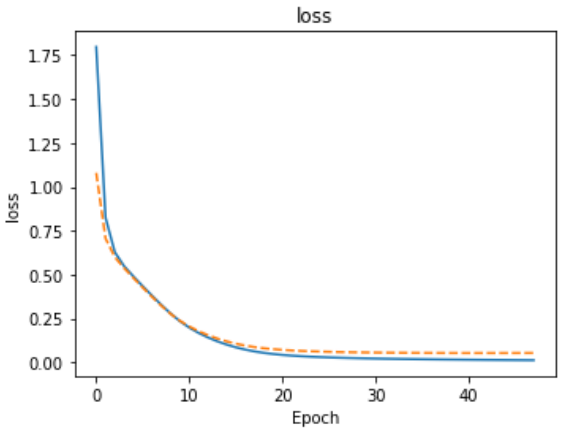
نتایج سه حالت بررسی شده در ادامه می‌آید.

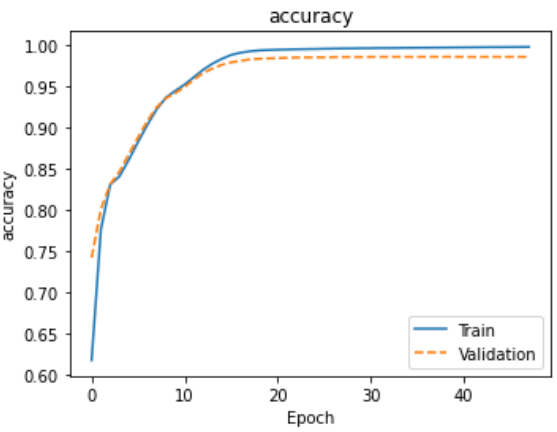
- اندازه 4
  - 480,213 پارامتر
  - val_loss: 0.1523 - val_accuracy: 0.9697



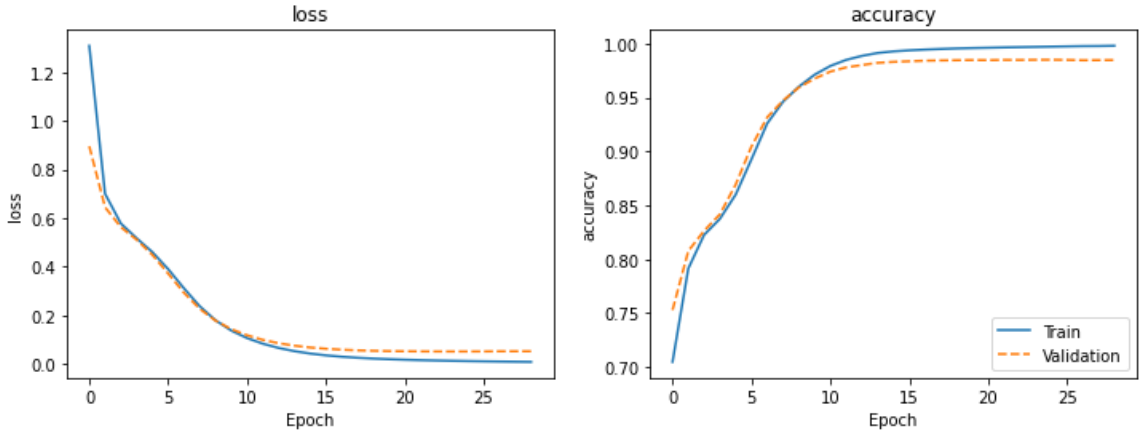


- اندازه 16
  - 481,005 پارامتر
  - val_loss: 0.0533 - val_accuracy: 0.9849





- اندازه 64
  - 487,053 پارامتر
  - val_loss: 0.0523 - val_accuracy: 0.9851



لازم به ذکر است که برای توقف مدل از early stopping استفاده شده است و هر گاه به overfitting نزدیک شویم توقف می‌کنیم. همانطور که دیده می‌شود هر چه units را بیشتر کردیم مدل توانست بهتر روی داده validation عمل کند که نشان می‌دهد قدرت یادگیری مدل افزایش یافته. البته باید ذکر کرد که افزایش بیش از حد این اندازه باعث پیچیدگی بیش از حد مدل می‌تواند شود و باید از آن اجتناب کرد که البته به علت محدودیت زمان و منابع اعداد خیلی بزرگ را آزمایش نکردیم. اما طبق این آزمایشها بهترین نتیجه برای تعداد 64 است.

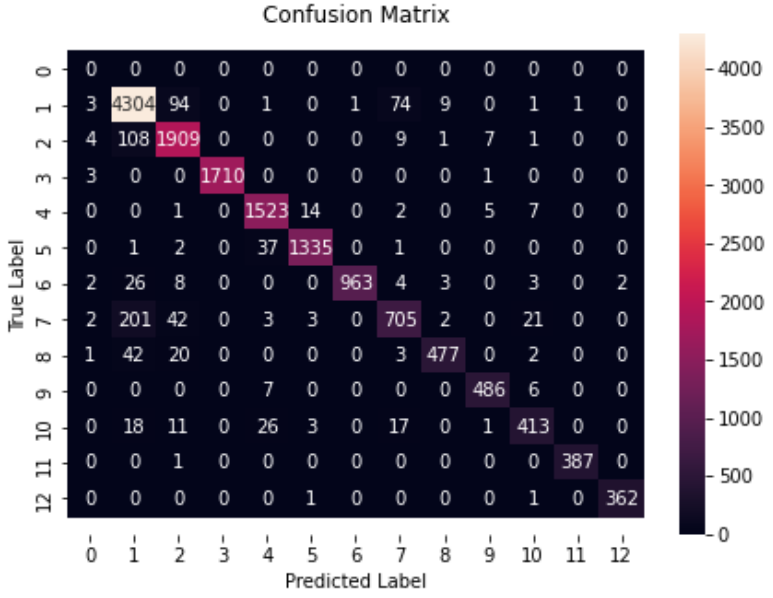
اهمیت استفاده از داده validation به جای تست برای تنظیم هایپرپارامترها این است که داده تست باید کاملا جدا نگه داشته شود و در طول آموزش مدل اصلا دیده نشود. از این داده تنها در انتهای کار استفاده می‌کنیم و روی آن تست نهایی را انجام می‌دهیم. اگر از داده تست برای تنظیم هایپرپارامترها استفاده شود، در اصل انگار که داریم به طور خاص مسئله را برای آن داده تنظیم می‌کنیم و این تنظیم خاص از generalization مدل کم می‌کند و روی داده واقعی که بخواهد کار کند ممکن است نتیجه بد شود. به همین دلیل روی validation تنظیمات هایپرپارامتر را انجام می‌دهیم و در انتها روی تست هم دقت را به دست می‌آوریم تا مطمئن شویم تنظیم خاص برای داده validation باعث نشده باشد generalization کم شود و هنوز روی داده تست هم بتواند نتیجه خوبی بدهد. اگر دقت validation خوب شود و بعد در داده تست دقت کم شود نشانه بدی است که مدل تعمیم پذیری را از دست داده و باید فکری کرد.

ج)

حالا باید دقت روی داده تست را گزارش کنیم. در این بخش باید توجه داشت که ما در ابتدای آموزش مدل، داده‌ها را با استفاده از padding هم اندازه کردیم (برای محدودیت های tensorflow و امکان اجرای موازی روی GPU). یعنی ابتدا یا انتهای جمله‌ها یک تگ جدید به عنوان padding اضافه کردیم که معنای خاصی ندارد. در keras و تابع evaluate آن، این پدینگ ها هم در محاسبه دقت در نظر گرفته می‌شوند و به دقت آن کمک می‌کنند. مثلا اعدادی که در بخش قبل حدود 98 درصد اعلام شد در اصل انقدر نباید بالا باشند و با در نظر گرفتن تعداد زیادی توکن با تگ padding این عدد به دست آمده است. (البته برای تنظیم هایپرپارامترها اشکالی ندارد چون شرایط یکسان است) برای اینکه این مشکل حل شود در کدی که قبلا نشان داده شد، ابتدا خروجی های مدل را می‌گیریم که هر کدام به طول 100 هستند. (به صورت خاص تنظیم کردیم که padding در انتهای جمله ها باشد و طول همه جمله ها را به 100 برساند.) سپس نتایج هر جمله را به اندازه ای که واقعا جمله بوده کوتاه میکنیم تا padding ها حذف شوند. و سپس دقت را محاسبه می‌کنیم.

نتایج این دقت برای بهترین مدل قسمت قبل در زیر آمده است.

```
Test Loss: 0.05234244093298912 Test Accuracy (w/ padding): 0.9851020574569702
```




```
Test Accuracy (w/o padding): 0.9436674436674437
```

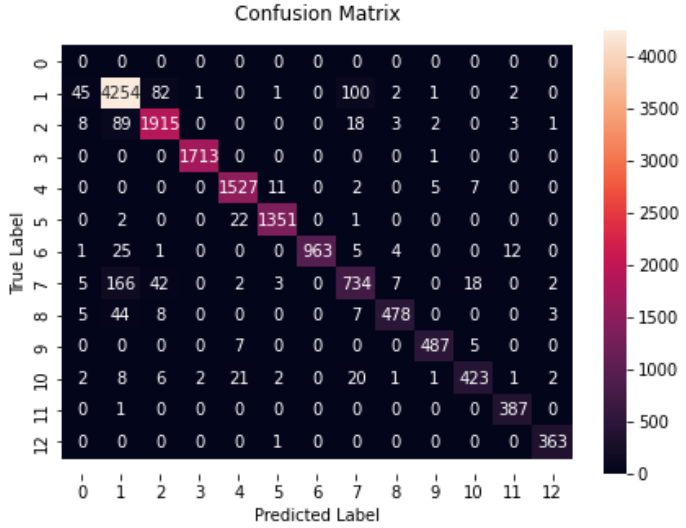
همانطور که دیده می‌شود اگر پدینگ هم در نظر گرفته شود (تابع خود keras) دقت حدود 98 است، اما پس از اینکه پدینگ ها حذف شوند (کلاس 0 در ماتریس) دیده می‌شود که دقت در اصل 94.37 است.

ج)

برای GRU دقت validation برای تعداد 4 و 16 و 64 به ترتیب برابر 0.9748 و 0.9857 و 0.9854 شد که در این مورد ظاهرا 16 بهترین نتیجه است.

بنابراین دقت تست با استفاده از 16 به شکل زیر برابر 94.50 است.

```
19/19 [==============================] - 0s 4ms/step - loss: 0.0509 - accuracy: 0.9855
Test Loss: 0.05094771459698677 Test Accuracy (w/ padding): 0.9855272173881531
```

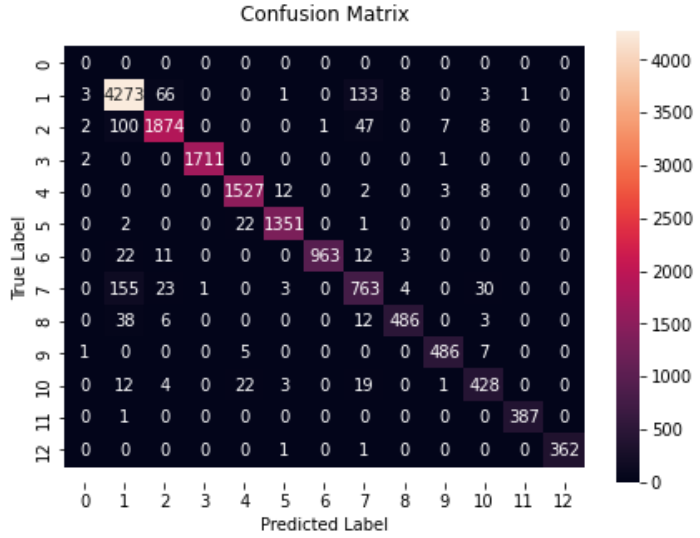



```
Test Accuracy (w/o padding): 0.945027195027195
```

برای LSTM دقت validation برای تعداد 4 و 16 و 64 به ترتیب برابر 0.9696 و 0.9852 و 0.9858 شد که در این مورد ظاهرا 64 بهترین نتیجه را دارد.

بنابراین دقت تست به شکل زیر برابر 96.61 است.

```
19/19 [==============================] - 0s 5ms/step - loss: 0.0469 - accuracy: 0.9858
Test Loss: 0.046879108995199203 Test Accuracy (w/ padding): 0.985765278339386
```



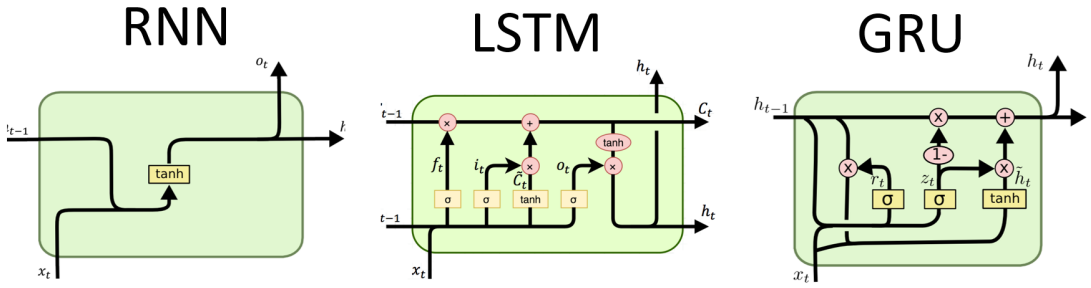

```
Test Accuracy (w/o padding): 0.9460631960631961
```

برای مقایسه این سه مدل باید دوباره ذکر کرد که به ترتیب RNN و GRU و LSTM توانستند دقت تست 94.37 و 94.50 و 94.61 به دست آورند. طبق نتایج مشخصا RNN عادی ضعیف تر از دو مورد دیگر است. علت این است که simpleRNN ساختار بسیار ساده و با کمترین میزان ضرب ماتریسی عمل را انجام می‌دهد و همین سادگی باعث می‌شود قدرت خیلی زیادی نشود انتظار داشت، همچنان که دو مدل دیگر نشان داده شده می‌توانند توانایی های بیشتری داشته باشند چون گیت های خاصی برای یادآوری و فراموشی دارند که می‌تواند اطلاعات در طول زمان منتقل یا فراموش کند در حالیکه مورد اول همچین توانایی ندارد. اما در مورد GRU و LSTM دقتها به هم نزدیکتر است و لزوما با همین یک تست نمی‌توان قطعی گفت که LSTM بهتر است و باید با سیدهای متفاوت تست شود و انحراف معیار نیز گزارش شود تا دید می‌توان از نظر آماری تصمیمی گرفت یا نه. البته می‌توان در کل گفت که ساختار GRU ساده تر از LSTM است و همانطور که در درس گفته شد فقط دو گیت ریست و اپدیت دارد، اما LSTM سه گیت دارد برای ورودی، خروجی و فراموشی. بنابراین همان استدلال ساختار ساده تر و قدرت نسبتا کمتر را

می‌توان در مورد GRU آورد اگرچه که اختلاف زیاد نیست. در قسمت بعدی در مورد ساختار بیشتر بحث می‌کنیم.

غ)

در شکل زیر ساختار معماری RNN و GRU و LSTM آمده است.



معماری LSTM سه گیت دارد. گیت سمت چپ forget gate است که یک تابع سیگموید دارد و با ضرب خروجی آن که بین 0 و 1 است، مشخص می‌کند که محتویات cell state گذشته چقدر نگهداری شود. (توجه داشته باشید که این مقدار یک عدد نیست و به ازای هر بعد این عدد وجود دارد و می‌تواند بعضی ابعاد را فراموش و بعضی را نگه دارد.) گیت وسط input gate است که مشخص می‌کند چه مقدار از ورودی جدید وارد cell state شود که با ضرب سیگموید در مقدار ورودی که از tanh رد شده و در انتها جمع آن با cell state انجام می‌شود. در سمت راست output gate وجود دارد که ترکیب cell state و hidden state را مشخص می‌کند که در hidden state خروجی قرار گیرد که به زمان بعدی و همچنین به عنوان خروجی این زمان داده می‌شود.

معماری GRU مهمترین تفاوتی که دارد این است که به جای دو خط cell state و hidden state فقط یک خط انتقال hidden state دارد. گیت سمت چپ reset gate است که با ضرب خروجی بعد از سیگموید در h مرحله قبلی تاثیر آن را در ترکیب با ورودی جدید مشخص می‌کند. سمت راست نیز update gate است که خروجی بعد از سیگموید را همزمان استفاده می‌کند تا میزان اضافه کردن مقدار جدید بر روی h را مشخص کند و همچنین با استفاده از معکوس آن و ضربش در h قبلی تاثیر آن را کاهش می‌دهد. در مجموع GRU نسبت به LSTM محاسبات کمتری دارد و با سرعت بیشتری می‌تواند اجرا شود، و در مقابل آن، پیچیدگی های بیشتر LSTM پتانسیل یادگیری بیشتر و انعطاف بیشتر در انتقال اطلاعات را دارد.

د)

بهترین نتیجه یعنی نتیجه LSTM برابر 94.61 بود (بدون padding) در حالیکه نتیجه قسمت پ برابر 90.26 بود. بنابراین مشخصا استفاده از یک مدل RNN می‌تواند دقت خیلی بهتری ارائه کند. در مورد معماری این مدلها توضیح داده شد و همین معماری امکان یادگیری dependency های long و short را فراهم می‌کند در حالیکه در حالت HMM و ویتربی صرفا داشتیم به دو کلمه کنار هم توجه می‌کردیم ولی در LSTM می‌توان به باقی بخشها نیز توجه کرد. (همچنین اگر از bidirectional LSTM استفاده می‌کردیم تا در هر لحظه کل جمله در نظر گرفته می‌شد می‌توانستیم احتمالا نتایج بهتری هم بگیریم.)

همچنین بازنمایی هایی که از کلمات توسط لایه embedding و همچنین درون LSTM ساخته می‌شود قدرت شبکه های عصبی را دارد که نشان داده شده چقدر پیشرفت می‌توانند ایجاد کنند و به صورت discriminative این کار را انجام دهند. بنابراین هم توجه به context بیشتر و هم پیچیدگی‌های شبکه عصبی توانستند این پیشرفت را برای ما ایجاد کنند.

2 تشخیص گروه‌های اسمی

الف)

برای به دست آوردن تگ های NER به صورت IOB کد زیر پیاده سازی شد.

```
class DatasetManager:
    def __init__(self):
        dataset = nltk.corpus.treebank.tagged_sents()
        dataset = [tree2conlltags(nltk.ne_chunk(d)) for d in dataset]
        train_set, test_set = train_test_split(dataset, test_size=0.35, random_state=41)
        train_set, dev_set = train_test_split(dataset, test_size=0.01, random_state=41)
        self.dataset = dict()
        self.dataset["x_train"] = [[d[0].lower() for d in data] for data in train_set]
        self.dataset["y_train"] = [[d[2] for d in data] for data in train_set]
        self.dataset["x_dev"] = [[d[0].lower() for d in data] for data in dev_set]
        self.dataset["y_dev"] = [[d[2] for d in data] for data in dev_set]
        self.dataset["x_test"] = [[d[0].lower() for d in data] for data in test_set]
        self.dataset["y_test"] = [[d[2] for d in data] for data in test_set]
        self.vocab = set([word for sentence in self.dataset["x_train"] for word in sentence])
        self.tags = set([tag for sentence in self.dataset["y_train"] for tag in sentence])

dataset_manager = DatasetManager()
```

```
print(dataset_manager.dataset["x_train"][1])
print(dataset_manager.dataset["y_train"][1])
```

```
['the', 'philadelphia', 'fed', ',', 'for', 'instance', ',', 'reported', 'that', 'manufacturing', 'act
['O', 'B-ORGANIZATION', 'I-ORGANIZATION', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
```

ب)

کد بخش قبل به صورت مناسبی نوشته شده بود که نیازی به تغییر برای اجرا روی این دیتاست جدید ندارد. تنها کافیست تا ورودی smoothing برای ساخت ماتریس transition برابر false گذاشته شود تا توالی های غیر ممکن که در داده آموزش نیامده در تستها نیز 0 شود و امکان پیشنهادش نباشد. (همچنین می‌توان smooth کرد و سپس به صورت دستی توالی های I و B را 0 کرد. اما در نهایت مهم این است که توالیهای نامناسب transition برابر صفر داشته باشند تا کل مسیر 0 شود.)

	B-FACILITY	I-PERSON	B-PERSON	I-ORGANIZATION	I-GPE	B-GPE	B-LOCATION	B-ORGANIZATION	I-FACILITY	O
B-FACILITY	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.916667	0.083333
I-PERSON	0.000000	0.083333	0.027778	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.888889
B-PERSON	0.000000	0.222973	0.020270	0.000000	0.000000	0.000000	0.000000	0.087838	0.000000	0.668919
I-ORGANIZATION	0.000000	0.000000	0.000000	0.200000	0.000000	0.000000	0.000000	0.006452	0.000000	0.793548
I-GPE	0.000000	0.000000	0.000000	0.000000	0.013333	0.000000	0.000000	0.000000	0.000000	0.986667
B-GPE	0.000000	0.000000	0.000000	0.000000	0.120521	0.001629	0.000000	0.068404	0.000000	0.809446
B-LOCATION	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
B-ORGANIZATION	0.000000	0.000000	0.000000	0.328042	0.000000	0.000000	0.000000	0.010582	0.000000	0.661376
I-FACILITY	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
O	0.000143	0.000000	0.001289	0.000000	0.000000	0.006038	0.000036	0.003735	0.000000	0.949069

همانطور که دیده می‌شود تمام I به B ها 0 هستند و مشکلی پیش نمی‌آید. اما برای دقیق بودن پاسخ روشی که smooth بکنیم و حالتهای خاص را دستی 0 کنیم پیش می‌بریم.

```
        for tag1 in self.transition_matrix.keys():
            for tag2 in self.transition_matrix[tag1].keys():
                if "I-" in tag1 and "B-" in tag2 and tag1[2:] == tag2[2:]:
                    self.transition_matrix[tag1][tag2] = 0
                if "O" == tag1 and "I-" in tag2:
                    self.transition_matrix[tag1][tag2] = 0
                if "I-" in tag1 and "I-" in tag2 and tag1[2:] != tag2[2:]:
                    self.transition_matrix[tag1][tag2] = 0
```

و نتیجه به شکل زیر است که فقط خانه های خاصی 0 شده اند.

	B-FACILITY	I-PERSON	B-PERSON	I-ORGANIZATION	I-GPE	B-GPE	B-LOCATION	B-ORGANIZATION	I-FACILITY	O
B-FACILITY	0.045455	0.045455	0.045455	0.045455	0.045455	0.045455	0.045455	0.045455	0.545455	0.090909
I-PERSON	0.021739	0.086957	0.000000	0.000000	0.000000	0.021739	0.021739	0.021739	0.000000	0.717391
B-PERSON	0.006329	0.215190	0.025316	0.006329	0.006329	0.006329	0.006329	0.088608	0.006329	0.632911
I-ORGANIZATION	0.006061	0.000000	0.006061	0.193939	0.000000	0.006061	0.006061	0.000000	0.000000	0.751515
I-GPE	0.011765	0.000000	0.011765	0.000000	0.023529	0.000000	0.011765	0.011765	0.000000	0.882353
B-GPE	0.001603	0.001603	0.001603	0.001603	0.120192	0.003205	0.001603	0.068910	0.001603	0.798077
B-LOCATION	0.076923	0.076923	0.076923	0.076923	0.076923	0.076923	0.076923	0.076923	0.076923	0.307692
B-ORGANIZATION	0.002577	0.002577	0.002577	0.322165	0.002577	0.002577	0.002577	0.012887	0.002577	0.646907
I-FACILITY	0.000000	0.000000	0.047619	0.000000	0.000000	0.047619	0.047619	0.047619	0.047619	0.571429
O	0.000155	0.000000	0.001301	0.000000	0.000000	0.006049	0.000048	0.003747	0.000000	0.948968

پ)

عملکرد مدل به شکل زیر است.

	precision	recall	f1-score	support
0	0.00	0.00	0.00	0
B-FACILITY	0.75	0.60	0.67	15
B-GPE	0.92	0.58	0.71	651
B-GSP	0.00	0.00	0.00	10
B-LOCATION	0.00	0.00	0.00	10
B-ORGANIZATION	0.84	0.25	0.39	564
B-PERSON	0.91	0.58	0.71	720
I-FACILITY	0.73	0.73	0.73	15
I-GPE	0.90	0.78	0.84	79
I-LOCATION	0.00	0.00	0.00	8
I-ORGANIZATION	0.75	0.38	0.50	338
I-PERSON	0.73	0.81	0.77	458
O	0.97	1.00	0.98	32304
accuracy			0.96	35172
macro avg	0.58	0.44	0.48	35172
weighted avg	0.96	0.96	0.95	35172

همانطور که دیده می‌شود اکثر موارد در این دیتاست O هستند و تعداد خیلی کمتری به صورت تگهای دیگر هستند. به همین دلیل O را به خوبی تشخیص داده‌ایم ولی از بقیه دسته ها که داده کمی داشتیم فقط دسته هایی مثل B-GPE که تعداد بیشتری داشته آموخته شده و بقیه انقدر کم بودند که اصلا امکان یادگیریشان و تعمیم پذیری نبوده است. اما از نظر دقت 96 درصد و weighted avg f1 برابر 95 و precision برابر 96 می‌توان گفت مدل به خوبی دیتاست را آموخته است، اما باید توجه داشت که خود دیتاست به اندازه کافی بزرگ نبوده و می‌توانست تعداد و تنوع بیشتری داشته باشد تا بهتر یادگیری انجام شود.

ث)

قطعا می‌توان از مدلهای بازگشتی برای NER استفاده کرد. چالش این است که این مدلها خروجی خودشان را می‌دهند احتمالا بر روی بعضی داده ها ممکن است توالی‌های غیر مجاز را نیز بسازند که مناسب نیست. برای حل این مشکل اولا می‌توان به صورت ساده اگر توالی غیر مجاز ایجاد شده بود تبدیل به O کرد. اما این روش خیلی ساده است. برای انجام دقیقتر این کار می‌توان از CRF استفاده کرد مانند مقاله https://arxiv.org/pdf/1508.01991.pdf. با این کار می‌توان مسیرهای غیر مجاز را 0 کرد و

تنها توالی هایی که مناسب هستند را خروجی داد. چیزی مشابه کاری که با HMM و ویتربی در بخشهای قبلی انجام دادیم. در مجموع همه این روش ها سعی می‌کنند با قرار دادن لایه ای اضافه روی نتیجه مدل های بازگشتی این مشکل را برطرف کنند.