שכבת תשומת לב ושימושה במפרש הנשנה

אחד הקשיים הגדולים בשימוש ברשתות נוירונים נשנות הוא למידת קשרים ארוכי טווח, בעיה בה דנו כבר ביחידה 6. בבואנו לבצע תרגום מכונה בעיה זו מקבלת משנה תוקף. חשבו לדוגמה על זוג המשפטים הבאים:

"I have a dog, and I love it"

"יש לי כלב, ואני אוהבת אותו"

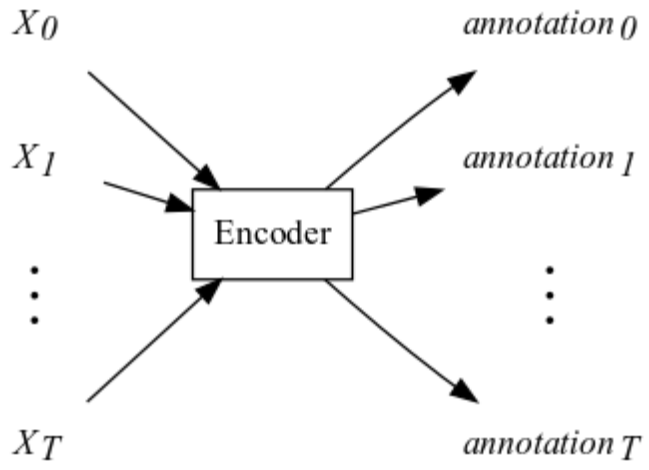
על מנת לתרגם נכונה מאנגלית לעברית את המילה האחרונה בזוג זה, יהיה על הרשת לזהות שהמילה "אותו" מתייחסת ל-"dog ", ולדעת שמילה זו בעברית היא ממין זכר. ברשת שבנינו בפרק הקודם, קשר זה בהכרח חייב להישמר דרך כל החישובים המבוצעים במקודד לאחר הזנת המילה "dog", וכן דרך כל החישובים המבוצעים במפרש עד להזנת המילה "כלב", שכן המקודד מעבד את משפט המקור מתחילתו לסופו, מייצר את וקטור ההקשר ולבסוף המפרש מייצר את המשפט המתורגם, שוב מההתחלה אל הסוף.

על מנת להקל על הרשת ללמוד קשרים מסוג זה, נרצה לספק למפרש גישה מיידית לכל המילים במשפט המקור, כך שבדוגמת התרגום הקודמת היא תוכל (באופן אידיאלי) להבין בעת עבודתה שהמילה האחרונה במשפט צריכה להיות כינוי גוף, ועל מנת להחליט אם מדובר ב"הוא", "היא" או "זה", היא תוכל לחפש את שם העצם המתאים במשפט המקור ישירות, מבלי להסתמך על וקטור ההקשר שהוזן לתא הנשנה באיטרציה הראשונה שלו.

פונקציונליות חיפוש זו, הקרויה שכבת תשומת לב (attention layer), היא הבסיס לארכיטקטורת רשת מוצלחת במיוחד, ה-Transformer, ומוצאת שימוש במשימות רבות מעבר לתרגום מכונה ואף מחוץ לתחום של עיבוד שפה טבעית. בפרק הנוכחי נכיר צורה פשוטה שלה, ונראה כיצד לשלבה בין המפרש למקודד ברשת תרגום המכונה על מנת להקל על הרשת ללמוד הקשרים של מילה במשפט היעד למילים במשפט המקור.

המקודד

ראשית, עלינו לשנות את פלט המקודד שכתבנו בפרק הקודם: ברצוננו לייצר כעת ייצוג חבוי של **כל טוקן** במשפט המקור, כך שבמקום וקטור הקשר יחיד המסכם את המשפט כולו, המקודד יעביר כפלט וקטור אנוטציה (annotation) **עבור כל אחד** מהטוקנים במשפט המקורי. לאחר אימון מוצלח, וקטורים אלו יהוו סיכום של הטוקן בהקשרו בתוך המשפט הנתון. נאייר זאת להלן.



למזלנו, על מנת לממש פונקציונליות זו בקוד יש לשנות רק שורה אחת במקודד מהפרק הקודם. זכרו שלאחר שפלט המודול `nn.LSTM` הוא היסטוריית המצבים החבויים של השכבה העליונה בערימת התאים. עד כה העברנו הלאה רק את המצב החבוי האחרון, השייך לטוקן הקלט האחרון, אך כעת

נשתמש בכל ההיסטוריה: המצב החבוי h_t יהיה האנוטציה של הטוקן X_t. ראו את המקודד החדש בקטע הקוד הבא, כאשר השורות הנבדלות מהמקודד הקודם מסומנות ב-#.

```
class Encoder(nn.Module):
    def __init__(self, embed_dim, hidden_dim, RNNlayers):
        super().__init__()
        self.src_embedding  = nn.Embedding(len(src_vocab),
                                           embed_dim)
        self.rnn_stack      = nn.LSTM(embed_dim,
                                      hidden_dim,
                                      RNNlayers)
    def forward(self, src_tokens):
      all_embeddings          = self.src_embedding(src_tokens)
      all_embeddings          = all_embeddings.unsqueeze(1)
      annotations, _= self.rnn_stack(all_embeddings)          #
      return annotations.squeeze()                            #
```

המפרש

שוב נשנה את הקוד שכתבנו בפרק הקודם, ונאפשר למפרש כעת גישה ישירה של אל האנוטציות: לפני כל צעד עדכון של התא הנשנה נחשב **וקטור הקשר פרטי** בעזרת שכבת תשומת הלב, שאת מימושה נדחה להמשך. לעת עתה נסתפק בתובנה ששכבה זו תקבל כקלט את המצב החבוי הנוכחי של המפרש, ותחשב עבורו ממוצע משוקלל של האנוטציות, כאשר טוקני משפט המקור הרלוונטיים ביותר לצעד המפרש הנוכחי יקבלו בכורה בממוצע זה. בנוסחה,

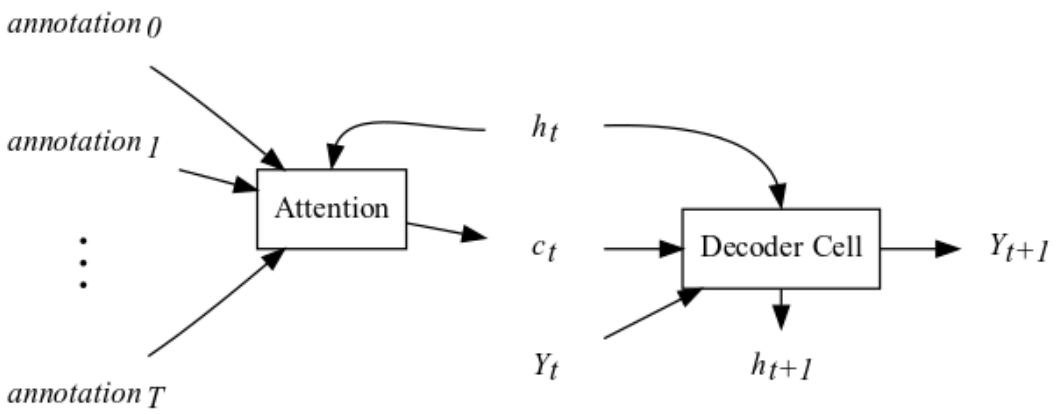
$$c_t = \sum_{k=0}^{T} \alpha\left(h_t, annotation_k\right) \cdot annotation_k .$$

α היא הפונקציה לפיה מתקבלת ההחלטה האם האנוטציה ה-k רלוונטית ליצירת הטוקן הבא בתור במשפט הפלט. אם זהו המצב, על משקלה של אנוטציה זו בוקטור ההקשר הנוכחי להיות גבוה. שימו לב שלפי נוסחה זו מימד c_t זהה לזה של האנוטציות, שמימדן כמימד המרחב החבוי של המקודד.

לאחר קבלת וקטור ההקשר הפרטי, התא הנשנה ישלבו בצעד עדכון המצב החבוי כלהלן,

$$h_{t+1} = \tanh\left(W_{input}X + b_{input} + W_{hidden}h_t + b_{hidden} + W_c c_t + b_c\right) .$$

מלבד תוספת זו, החישוב בתא הנשנה בעל מנגנון תשומת הלב נשאר זהה. נסיים דיון זה באיור סכמתי של התהליך החישובי של איטרציה יחידה במפרש, ומימוש התא החדש בקוד.



```
class AttnDecoderRNNCell(nn.Module):
    def __init__(self, embed_dim, hidden_dim):
        super().__init__()
        self.hidden_state  = torch.zeros(hidden_dim)
        self.RNNcell       = ContextRNNCell(embed_dim,        #
                                            hidden_dim)
        self.output_linear = nn.Linear(in_features=hidden_dim,
                                       out_features=len(tgt_vocab))
        self.logsoftmax    = nn.LogSoftmax(dim=0)

    def forward(self, one_embedded_token, attn_context):
        new_state           = self.RNNcell(one_embedded_token,
                                           self.hidden_state,
                                           attn_context)      #
        tgt_token_scores    = self.output_linear(new_state)
        tgt_token_logprobs = self.logsoftmax(tgt_token_scores)
        self.hidden_state  = new_state
        return tgt_token_logprobs
```

השוו תא זה לתא הנשנה `DecoderRNNCell` מהפרק הקודם וראו כי החידוש בא לידי ביטוי בשורות המסומנות ב-# בלבד. בראשונה בהן אנו מאתחלים תא אלמן מסוג `ContextRNNCell`, אשר מבצע את עדכון המצב החבוי על סמך שלושה וקטורי קלט, החדש בהם הוא ההקשר הפרטי, c_t. ניתן לראות זאת גם בשורה השניה המסומנת, שם התא מקבל קלט נוסף: פלט שכבת תשומת הלב.

על מנת לשלב תא זה בתוך המקודד ולייצר את טוקני המשפט המתורגם, אחד לאחר השני, יהיה עלינו לחשב ולהעביר לו את וקטור ההקשר בכל איטרציה, כפי שניכר מהפרמטרים של מתודת ה-`forward` שלו. זאת נעשה בעזרת שכבת תשומת הלב, אשר כעת הזמן לדון בה לפרטים.

תשומת לב

מנגנון תשומת הלב שואב השראה מחיפוש במאגר נתונים, ובהתאם נתחיל את הדיון בו בדוגמה זו. הניחו כי ברשותנו מילון המורכב זוגות של מפתחות וערכים, למשל,

```
values = ["woman","man","bicycle","queen","house"]
keys   = values
my_dict = dict(zip(keys,values))
pprint(my_dict)
```

פלט:

```
{'bicycle': 'bicycle',
 'house': 'house',
 'man': 'man',
 'queen': 'queen',
 'woman': 'woman'}
```

ראו כי במילון זה השתמשנו בערכים עצמם בתור המפתחות לצורך פשטות הדוגמה.

כעת, בבואנו לחפש מילה במילון זה מתבצעת השוואה של שאילתת החיפוש למפתחות (למעשה ההשוואה מתבצעת לאחר מעבר בפונקציית גיבוב, איך אין זה רלוונטי עבורנו), ואם קיים ערך במילון בעל מפתח זה בדיוק, הוא יוחזר. אם אין במילון מפתח **זהה** לשאילתה לא יוחזר דבר. ראו דוגמה לכך להלן.

```
print(my_dict.get("woman"),my_dict.get("girl"))
```

פלט:

```
woman None
```

למטרותינו, נרצה להפוך את החיפוש המילוני ל"רך": למשל ביודענו שהמילה "girl" בעלת קשר סמנטי הדוק למילה "woman", נצפה לראותה כתוצאת החיפוש על אף שמפתח זהה בדיוק לשאילתה "girl" לא קיים במילון. בבואנו לממש רצון זה עומדים לפנינו שני אתגרים:

1. המרת המפתחות והשאילתות לייצוג המביא לידי ביטוי את הקשר הסמנטי בין מילים.
2. מדידת הדימיון בין שתי מילים, על סמך ייצוג זה, שכן נרצה להחזיר כפלט החיפוש את המילה מהמילון הדומה ביותר לשאילתה.

המשימה הראשונה נראית מאתגרת על פניה, אך לאחר מחשבה נוכל להסיק שכבר ראינו את פתרונה – אלו הם שיכוני המילים המאומנים מראש של GloVe, בהם נתקלנו לראשונה כאשר עסקנו בעיבוד מקדים של נתוני טקסט ביחידה 6. נייבא אותם שוב, וכעת נגדיר את המפתחות המילון שלנו להיות וקטורי השיכון של המילים המתאימות.

```
from torchtext.vocab import GloVe
glove_embedder=GloVe(name='6B',dim=50)
keys = glove_embedder.get_vecs_by_tokens(values)
print(keys.size(),keys.dtype)
```

פלט:

```
torch.Size([5, 50]) torch.float32
```

שימו לב שכעת מפתחות הטוקנים הם וקטורים ממשיים במרחב בעל 50 מימדים.

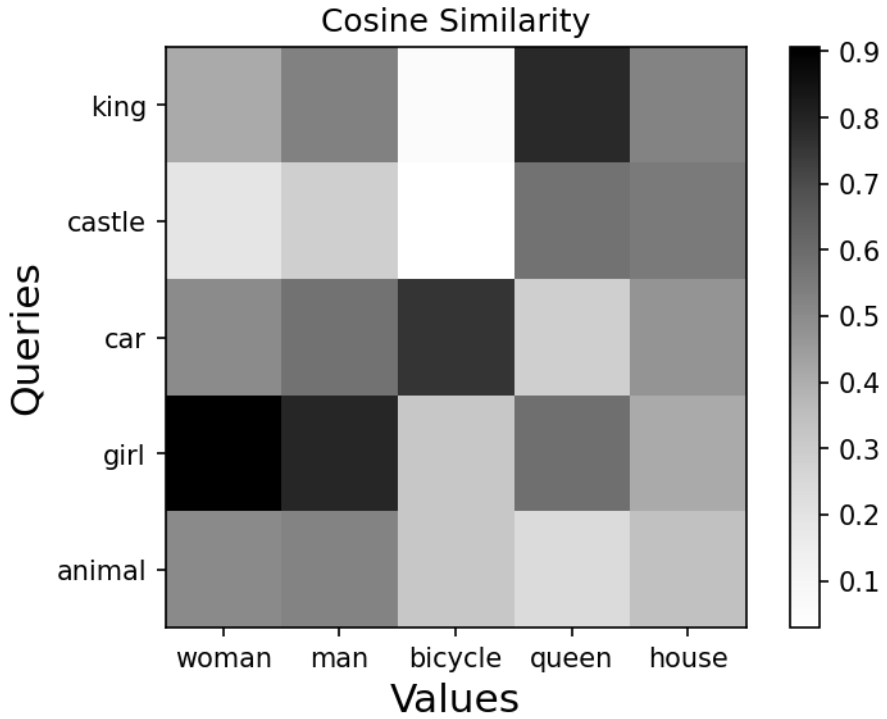
פתרון הבעיה השניה פשוט יותר בהנתן השיכון – כל מדד של דימיון/קרבה בין שני וקטורים יבצע את העבודה בצורה סבירה. כנהוג בשימושי עיבוד שפה טבעית, נבחר לעת עתה את דימיון הקוסינוס (cosine similarity) כמדד לדימיון הסמנטי. בהנתן השיכונים של שתי מילים, שניהם וקטורים של מספרים ממשיים באותו מרחב, פונקציית דימיון זו מחשבת את **קוסינוס הזווית** בין שני הוקטורים, לפי הנוסחה הבאה:

$$S_C(q,k)=\frac{q\cdot k}{\|q\|\|k\|}.$$

הפונקציה מחשבת את המכפלה הפנימית של וקטורי הקלט ומנרמלת אותם כך שיתקבל פלט בטווח $[-1,1]$. ככל שהערך $S_C(q,k)$ גבוה יותר, כך הזווית בין הוקטורים קרובה יותר לאפס, ובהתאם השאילתה q דומה יותר למפתח k. בעזרת פונקציה זו נממש את החיפוש הרך במילון בקוד.

```
def soft_search(query, keys, values):
  dict_size       = len(values)
  cos_similarity = torch.zeros(dict_size)
  q_norm          = query.norm()
  for idx in range(dict_size):
    dot                  = (query*keys[idx]).sum()
    k_norm               = keys[idx].norm()
    cos_similarity[idx] = dot/(q_norm*k_norm)
  best_match_idx = cos_similarity.argmax()
  return values[best_match_idx], cos_similarity
```

ראו כי פונקציה זו מצפה לקבל את השאילתה המשוכנת על ידי GloVe בפרמטר `query` וסדרה של מפתחות בפרמטר `keys`, אלו יהיו שיכוני המילים הנמצאות בפרמטר `values`. תוצאות החישוב עבור מספר שאילתות לדוגמה מאוירות להלן, כאשר הערך הכהה ביותר מצביע על תוצאת החיפוש עבור השאילתה בשורה זו.



תוך מתן מבט באיור זה, נסיק שלעתים ישנן מספר תשובות הולמות עבור שאילתה נתונה, למשל "king" בעל קשר סמנטי חזק ל"queen", כמובן, אך בנוסף גם ל"man". במימוש הנוכחי, תוצאת החיפוש אינה מביאה זאת בחשבון, שכן התוצאה הדומה ביותר בלבד היא שמוחזרת כפלט. כעת נתקן זאת על ידי חישוב ממוצע משוקלל עבור הפלט, ועל הדרך גם נוותר על חישוב הנורמות, על מנת לייעל את זמן הריצה של הפונקציה. התוצאה המתקבלת היא פונקציית תשומת לב המכפלה הפנימית (dot product attention).

```
def dot_attention(q, K, V):
  dict_size  = K.size(0)
  similarity = torch.zeros(dict_size)
  for idx in range(dict_size):
    similarity[idx] = (q*K[idx,:]).sum()
  attn_weights = F.softmax(similarity,dim=0)
  weighted_V   = torch.zeros_like(V)
  for idx in range(dict_size):
    weighted_V[idx,:] = attn_weights[idx]*V[idx,:]
  output = weighted_V.sum(dim=0)
  return output, attn_weights
```

ננתח פעולת פונקציה זו לפרטים:

- ראשית, כמו קודם, אנו מחשבים מדד דימיון בין השאילתה `q` לבין כל אחד מהמפתחות, הם עמודותיה של המטריצה `K`.
- אחרי כן אנו מנרמלים את מדד הדימיון בעזרת פונקציית ה-Softmax, שכן ברצוננו להשתמש בהם כמשקלים לחישוב ממוצע משוקלל.

- התוצאה המתקבלת ב-`attn_weights` היא וקטור אשר סכום כל איבריו הוא 1, וככל שהשאילתה דומה יותר למפתח מסויים, כך משקלו של וקטור זה בתוצאת החיפוש עולה.
- לבסוף ניכר כי פלט פונקציית חיפוש זו הוא וקטור יחיד, הממוצע המשוקלל של עמודות המטריצה `V`. אלו הם הערכים במילון המתאימים למפתחות ב-`K`.

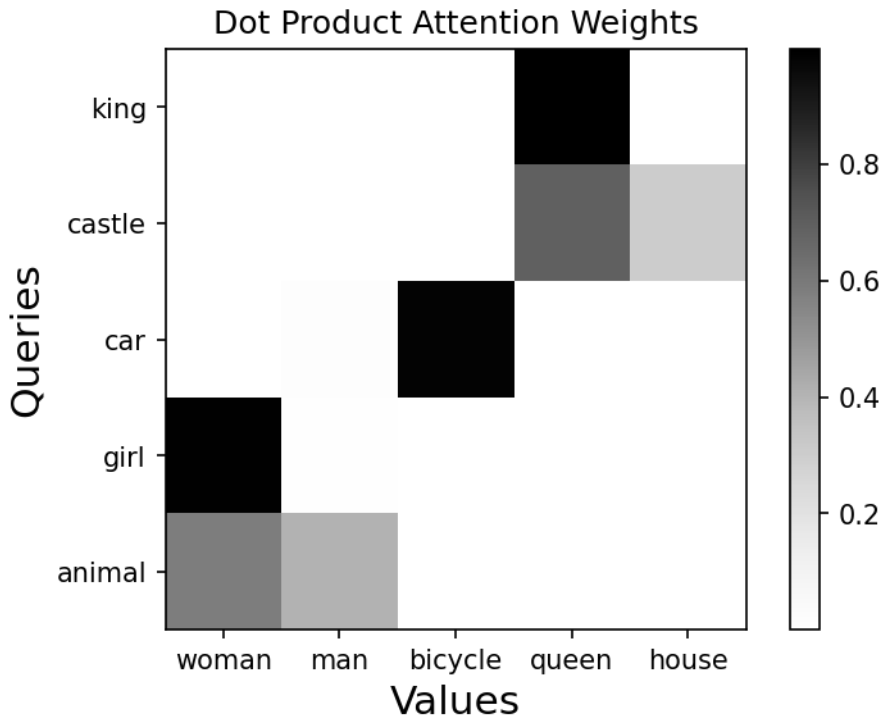
שימו לב שכעת גם על ערכי המילון להיות וקטורים ממשיים, על מנת לאפשר את חישוב הממוצע. בהתאם, עבור דוגמת המילון שלנו, נעביר לפונקציית תשומת הלב את המילים לאחר השיכון על ידי GloVe, כך שהמטריצות `K` ו-`V` יהיו זהות. בקטע הקוד הבא נדגים את השימוש בפונקציה.

```
src_words  = ["woman","man","bicycle","queen","house"]
values     = glove_embedder.get_vecs_by_tokens(src_words)
keys       = values
query      = glove_embedder.get_vecs_by_tokens("king")
search_result, attn_weights = dot_attention(query,keys,values)
print(search_result.size(),search_result.dtype)
print(attn_weights)
```

פלט:

```
torch.Size([50]) torch.float32
tensor([[7.0899e-05, 9.9381e-04, 1.3337e-09, 9.9831e-01,
6.2366e-04]])
```

ראו כי תוצאת החיפוש גם היא וקטור באותו מרחב שיכון בן 50 מימדים. במבט למשקלי הממוצע ניכר כי למילה "queen" ההשפעה הרבה ביותר על הערך המתקבל, באופן המתאים לדמיון הסמנטי ביניהם. נאייר מספר תוצאות נוספות של שימוש בפונקציית תשומת לב המכפלה הפנימית להלן.



מתוצאות אלו ניכר למשל שתוצאת החיפוש עבור השאילתה "castle" היא בעיקרה ממוצע של שיכוני המילים "queen" ו-"house", כצפוי.

חישוב תשומת הלב במפרש

כעת ברשותנו כל הדרוש על מנת לתאר ולממש את החישוב המבוצע במפרש. נשאיר מאחור את דוגמת חיפוש המילים במילון וניזכר שעלינו לייצר עתה בכל איטרציה וקטור הקשר עבור התא

הנשנה. וקטור זה יהיה תוצאת החיפוש של שכבת תשומת לב המכפלה הפנימית עם הקלט הבא: הערכים והמפתחות של השכבה יהיו וקטורי האנוטציה אשר יצר המקודד והשאילתה תהיה המצב החבוי הנוכחי. אם כן, חישוב תשומת הלב המבוצע לפני יצירת הטוקן ה- t במשפט המתורגם הוא:

$$c_t = \sum_{k=0}^{T} \alpha\left(h_t, annotation_k\right) \cdot annotation_k$$

כאשר

$$.\alpha\left(h_t, annotation_k\right) = \text{Softmax}\begin{pmatrix} h_t \cdot annotation_0 \\ h_t \cdot annotation_1 \\ \vdots \\ h_t \cdot annotation_T \end{pmatrix}$$

יש לשים לב שעל מנת לבצע חישוב זה ללא תקלה, מימד המצב החבוי של המפרש, h_t, צריך להיות זהה לזה של המקודד, זהו מימד וקטורי האנוטציות. תוך שימוש בתשומת לב אנו נותנים לרשת הזדמנות לייצג את המצב החבוי h_t בצורה דומה לאנוטציות הרלוונטיות לתרגום הטוקן הבא בתור. כמובן, כדי לנצל הזדמנות זו יהיה על הרשת ללמוד לעשות זאת בתהליך האימון. קוד המפרש מופיע בנספח בסוף הפרק, התבוננו בו וראו כי הוא דומה ברובו למפרש ללא תשומת הלב, אותו כתבנו בפרק הקודם.

לבסוף נחבר את המפרש למקודד על מנת לקבל את הרשת המלאה, כלהלן.

```
class Translator(nn.Module):
      def __init__(self, embed_dim, hidden_dim,encoder_layers):
        super().__init__()
        self.encoder = Encoder(embed_dim,
                               hidden_dim,
                               encoder_layers)
        self.decoder = AttnDecoder(embed_dim, hidden_dim)
      def forward(self,src_tokens, tgt_tokens):
        annotations = self.encoder(src_tokens)
        return self.decoder(annotations, tgt_tokens)
```

נסיים פרק זה בהסתייגות: במימוש ארכיטקטורת הרשת שמנו דגש על בהירות המימוש ופשטות החישוב, וזאת על פני יעילותו ואפקטיביות המודל הנלמד. ביתר פירוט:

- השתמשנו באוסף נתונים אשר נבנה על ידי מתנדבים, בו ידוע שקיימות טעויות.
- בעת העיבוד המקדים ביצענו טוקניזציה בצורה הפשוטה ביותר האפשרית.
- השתמשנו במפרש בעל שכבה יחידה של RNN המבוססת על תא אלמן, התא הנשנה הפשוט ביותר.
- מימשנו את החישוב הרקורסיבי בתא הנשנה של המפרש וכן את החישוב המבוצע בשכבת תשומת הלב באמצעות לולאות פייתון, אשר השימוש בהן אינו יעיל.
- השתמשנו בצורה הפשוטה ביותר של שכבת תשומת הלב, לה גרסאות רבות ומתוחכמות יותר.

התוצאה המתקבלות מהחלטות אלו היא שהרשת שלנו אומנם מסוגלת ללמוד, אך היא עושה זאת באיטיות רבה ואין היא מספקת תוצאות טובות במיוחד. על כן, לאחר קריאת פרק זה וקודמו עליכם

להבין את עקרון הפעולה של רכיביה ויחד עם זאת לקחת בחשבון שדרושים שיפורים טכניים ותיאורטיים נוספים על מנת למצות את מלוא יכולתם.

שאלות לתרגול

1. כתבו את המחלקה `ContextRNNCell` המממשת תא אלמן בעל שלושה וקטורי קלט שונים: המצב החבוי הקודם, טוקן הקלט הנוכחי ווקטור ההקשר הפרטי.
רמז: השתמשו בשכבות ליניאריות לכל אחד מוקטורי הקלט ולבסוף הפעילו עליהם את פונקציית האקטיבציה הדרושה. השקיעו מחשבה במימדי השכבות הליניאריות.
2. כתבו את פונקציית תשומת לב המכפלה הפנימית בצורה וקטורית, ללא שימוש בלולאות.
3. תכננו וכתבו את מצב החיזוי של המפרש עם תשומת הלב.
רמז: קחו השראה ממצב החיזוי של המקודד מהפרק הקודם.
4. בהנתן משפט קלט באורך T_1 טוקנים, ובהנחה שאובייקט `Translator` מייצר T_2 טוקנים עבור קלט זה, כמה פעמים המפרש קורא לפונקציית תשומת הלב? כמה פעמים מבוצע החישוב $q \cdot k$ עבור שאילתה ומפתח יחידים?

נספח: קוד המפרש בעל שכבת תשומת לב

```
class AttnDecoder(nn.Module):
    def __init__(self, embed_dim, hidden_dim):
        super().__init__()
        self.hidden_dim      = hidden_dim
        self.tgt_embedding   = nn.Embedding(len(tgt_vocab),
                                            embed_dim)
        self.RNNcell         = AttnDecoderRNNCell(embed_dim,
                                                  hidden_dim)
        self.attn_layer      = dot_attention
    def forward(self, annotations, tgt_tokens):
      self.RNNcell.hidden_state = torch.zeros(self.hidden_dim)
      keys   = annotations
      values = annotations
      translated_tokens = [START_Token]
      sentence_loss = 0
      for idx in range(len(tgt_tokens)-1):
        previous_token  = translated_tokens[idx]
        embedded_token  = self.tgt_embedding(previous_token)
        query           = self.RNNcell.hidden_state
        attn_context,_  = self.attn_layer(query, keys, values)
        logprobs        = self.RNNcell(embedded_token,
                                       attn_context)
        predicted_token = logprobs.argmax()
        translated_tokens.append(predicted_token.detach())

        correct_token   = tgt_tokens[idx+1]
        token_loss      = -logprobs[correct_token]
        sentence_loss  += token_loss

        if predicted_token == END_Token:
          break
      return translated_tokens, sentence_loss
```