50.039 Theory and Practice of Deep Learning

W9-S2 The Embedding Problem

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this week (Week 9)

1. Why are **embeddings** an essential component of Neural Networks (NNs)?
2. Why are **good embeddings** difficult to produce?
3. What are the **conventional approaches to embeddings in NLP?** What can we **learn from these approaches?**
4. What are the **typical issues with embeddings** and how do we address them?
5. **State-of-the-art** of current embedding problems, and **open questions** in research.

About this week (Week 9)

6. How do we evaluate the **quality/performance** of an embedding?
7. Can embeddings be **biased**?
8. Can we help the neural networks **identify the important parts of the context** to focus on?
9. What is **attention** in Neural Networks? What are **transformers** in Neural Networks?
10. What are the typical **uses for attention** these days?
11. What are the **limits of attention** and the **current research directions** on this topic?

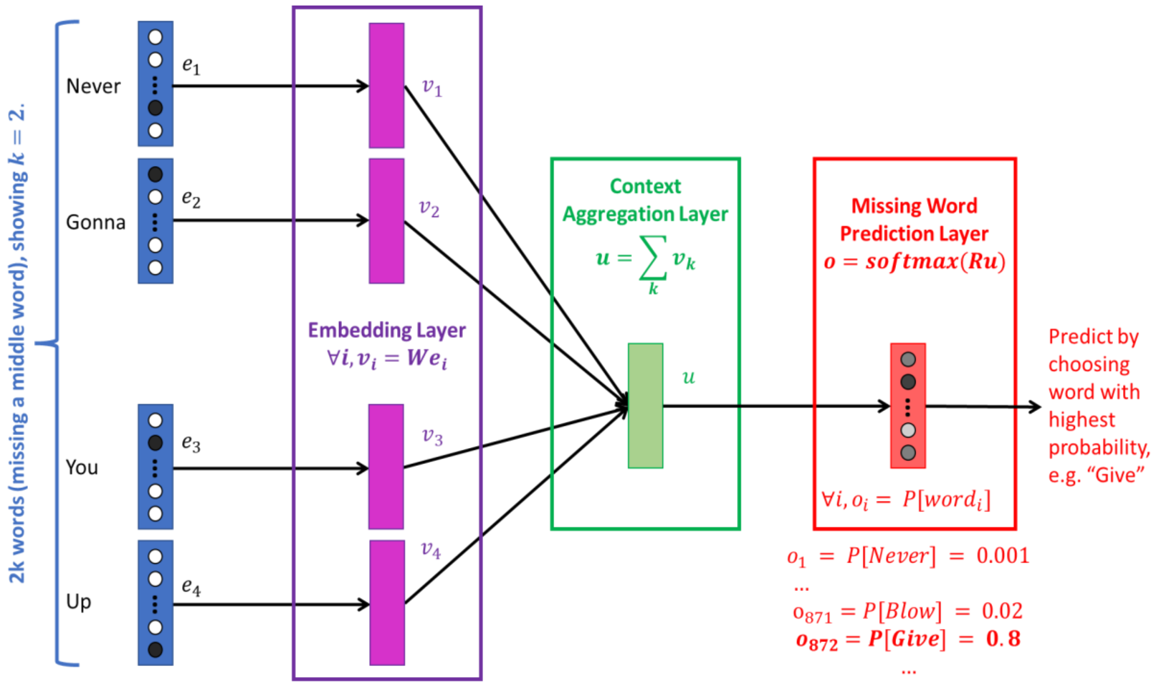
Continuous Bag of Words (CBoW)

Definition (CBoW):

CBoW (introduced in [Mikolov2013]) is a first feature representation model, which can be used for word embedding.

Using a large text corpus for training, it attempts to learn how to predict the word in the middle (with index k) of a sequence of $2k + 1$ words.

Here, k is called the **span** of the language model.

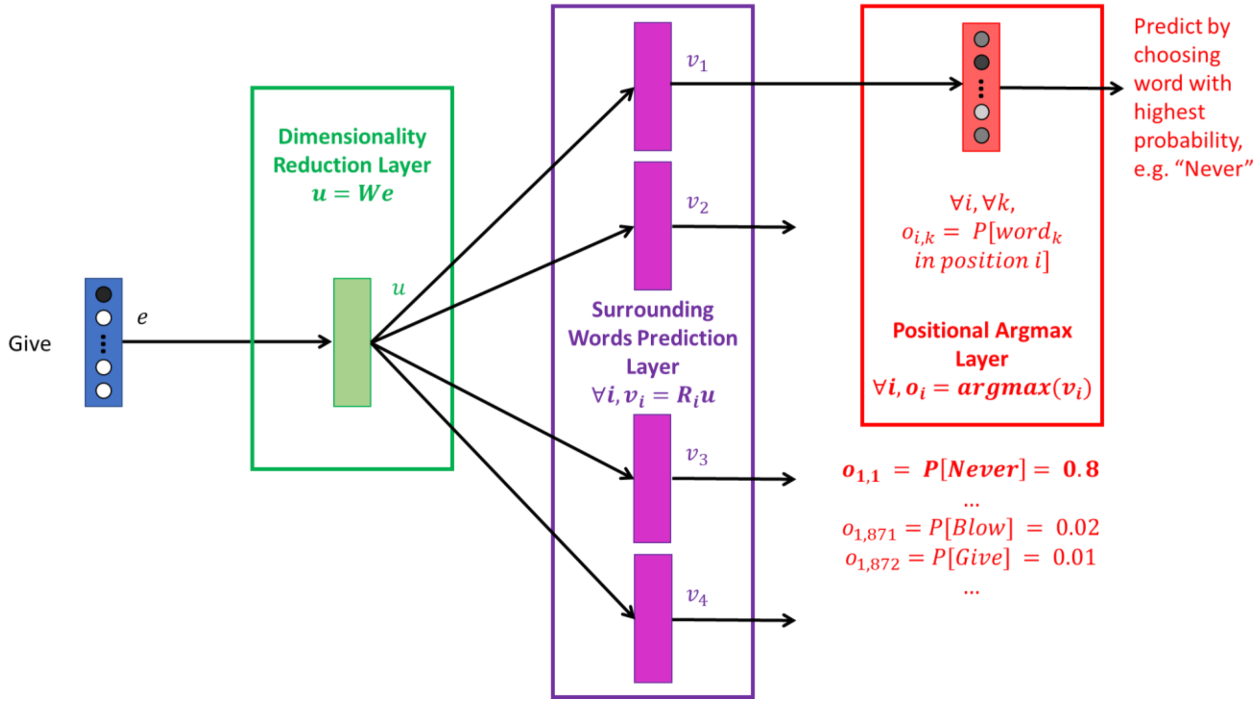


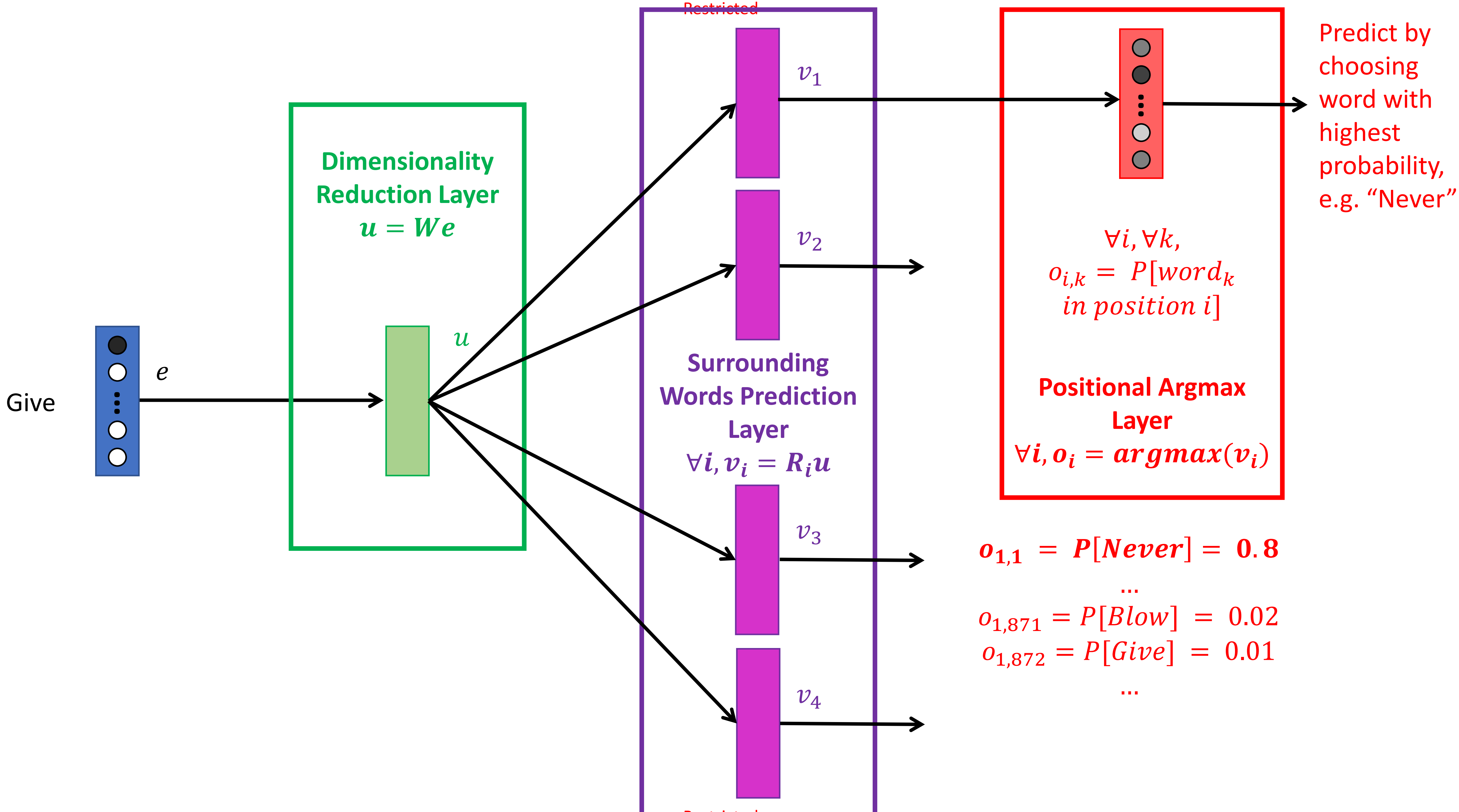
SkipGram (SG)

Definition (SkipGram):

Another interesting learning task one could use to train an embedding, would consist of using a single work and attempt to predict some possible surrounding words. In other words, take a middle word k, and try to predict the $2k$ surrounding words.

This is what **SkipGram** attempts to reproduce.




Give
e
Dimensionality
Reduction Layer
$\boldsymbol{u = We}$
u
v_1
v_2
Surrounding
Words Prediction
Layer
$\forall \boldsymbol{i}, \boldsymbol{v_i} = \boldsymbol{R_i u}$
v_3
v_4
$\forall i, \forall k,$
$o_{i,k} = P[word_k$
$in\ position\ i]$
Positional Argmax
Layer
$\forall \boldsymbol{i}, \boldsymbol{o_i} = \boldsymbol{argmax}(\boldsymbol{v_i})$
Predict by
choosing
word with
highest
probability,
e.g. "Never"
$\boldsymbol{o_{1,1} = P[Never] = 0.8}$
…
$o_{1,871} = P[Blow] = 0.02$
$o_{1,872} = P[Give] = 0.01$
…

Restricted

SkipGram (SG)

- Consider the text: "I have a dream that one day this nation will rise up and live out the true meaning of its creed: We hold these truths to be self-evident, that all men are created equal. I have a dream that one day on the red hills of Georgia, the sons of former slaves and the sons of former slave owners will be able to sit down together at the table of brotherhood."

0. We will use a sliding window, with e.g. size $k = 2$, to generate pairs of (x, y) values to train our SkipGram on.

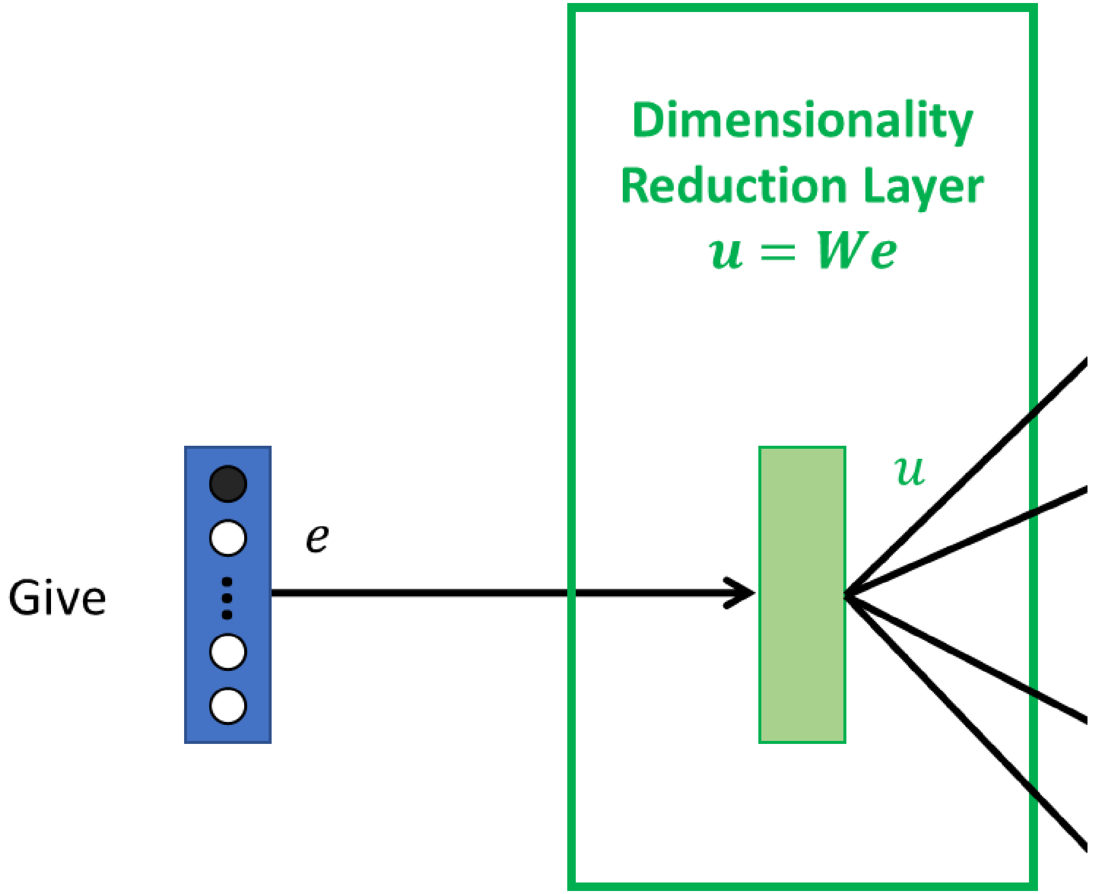
$$y_1 = (I, have, dream, that),$$
$$x_1 = a$$
$$y_2 = (have, a, that, one),$$
$$x_2 = dream$$
$$y_3 = (the, sons, former, slave),$$
$$x_3 = of$$

SkipGram (SG)

1. Then, build a simple NN, which takes a middle word, as one-hot embedding $e \in \mathbb{R}^{|V|}$.

   Add one fully-connected layer with matrix $\mathrm{W} \in \mathbb{R}^{D \times |V|}$. Here, D denotes the size of the new word embedding and is often chosen such that $D \ll |\mathrm{V}|$.

   After training, $u = We$ will be used as the new word embedding to replace any e!



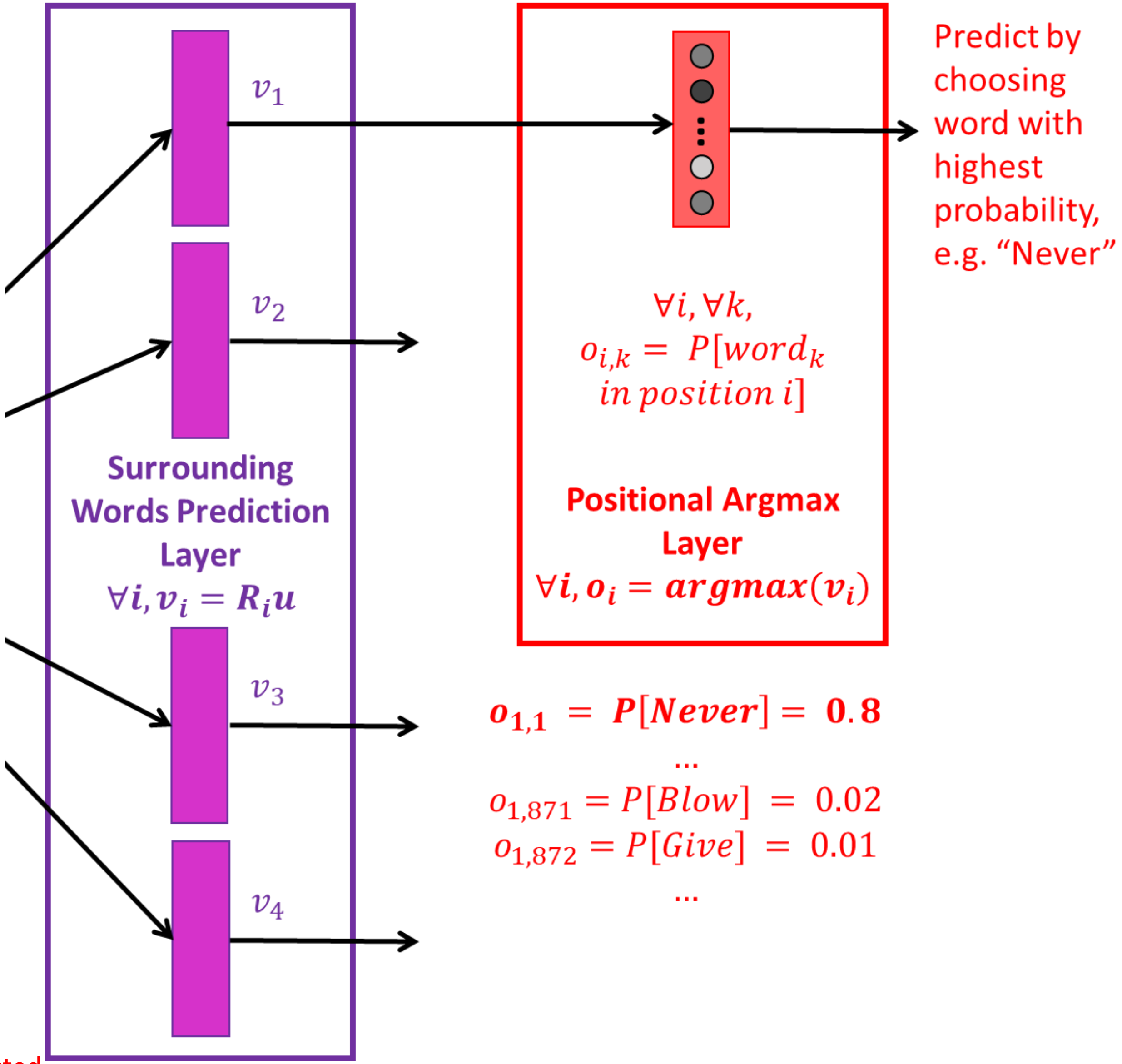
SkipGram (SG)

2. The final layer is a Linear layer (or an Embedding Layer), trainable and produces $2k$ output vectors v_i as:

$$v_i = R_i u$$

With $\forall i, v_i \in \mathbb{R}^{|V|}$.

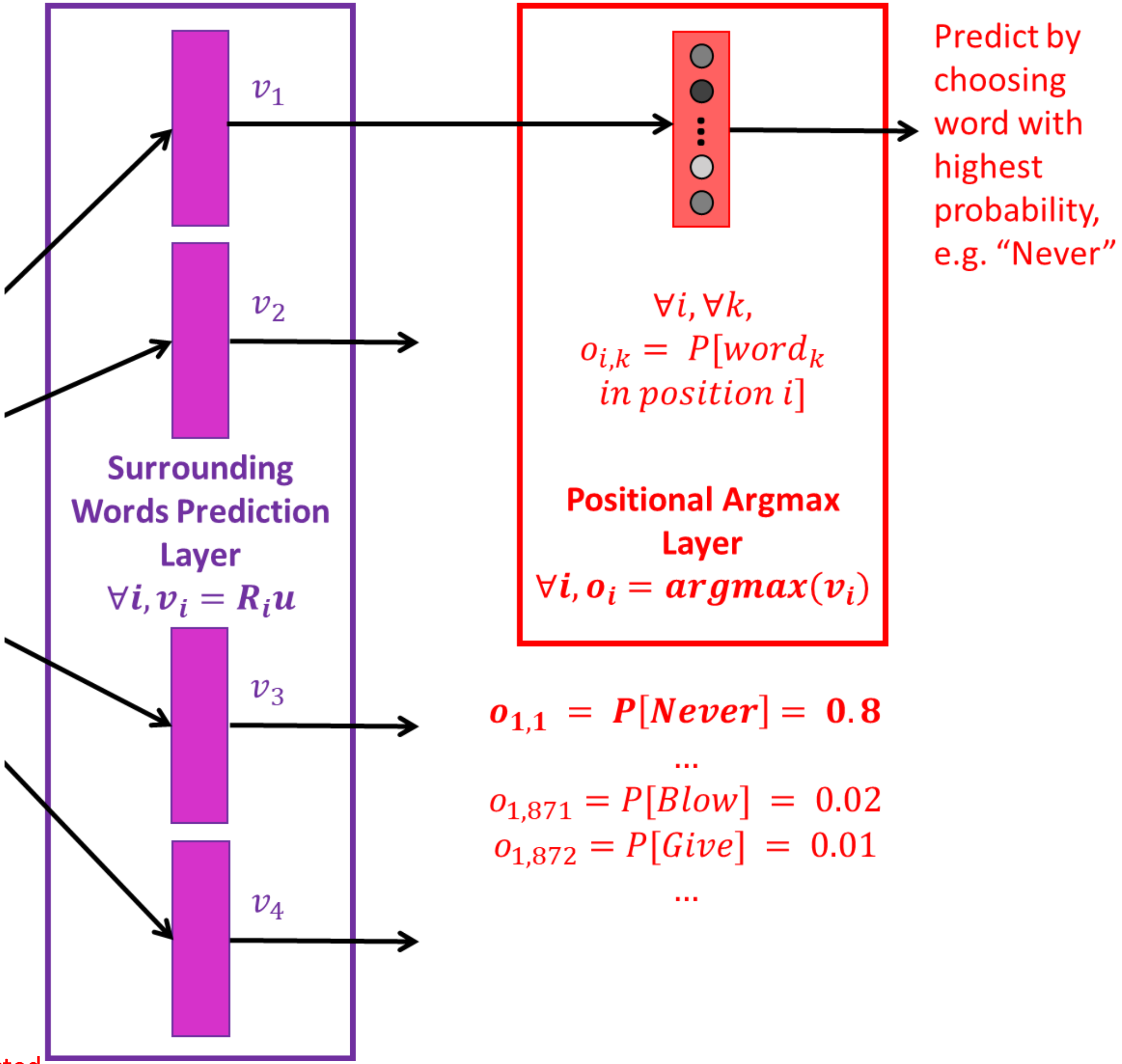
The outputs v_i will then pass through a softmax and will give probabilities over which word should be predicted in position i.



SkipGram (SG)

Important Note

- **The outputs $\boldsymbol{v_i}$ could be the result of $\boldsymbol{2k}$ linear layers, in parallel, each one producing a vector for each position $\boldsymbol{i}$.**
- This is easily done by using $2k$ Linear layers in PyTorch, which all have a single training parameter $R_i \in \mathbb{R}^{D\times|V|}$.

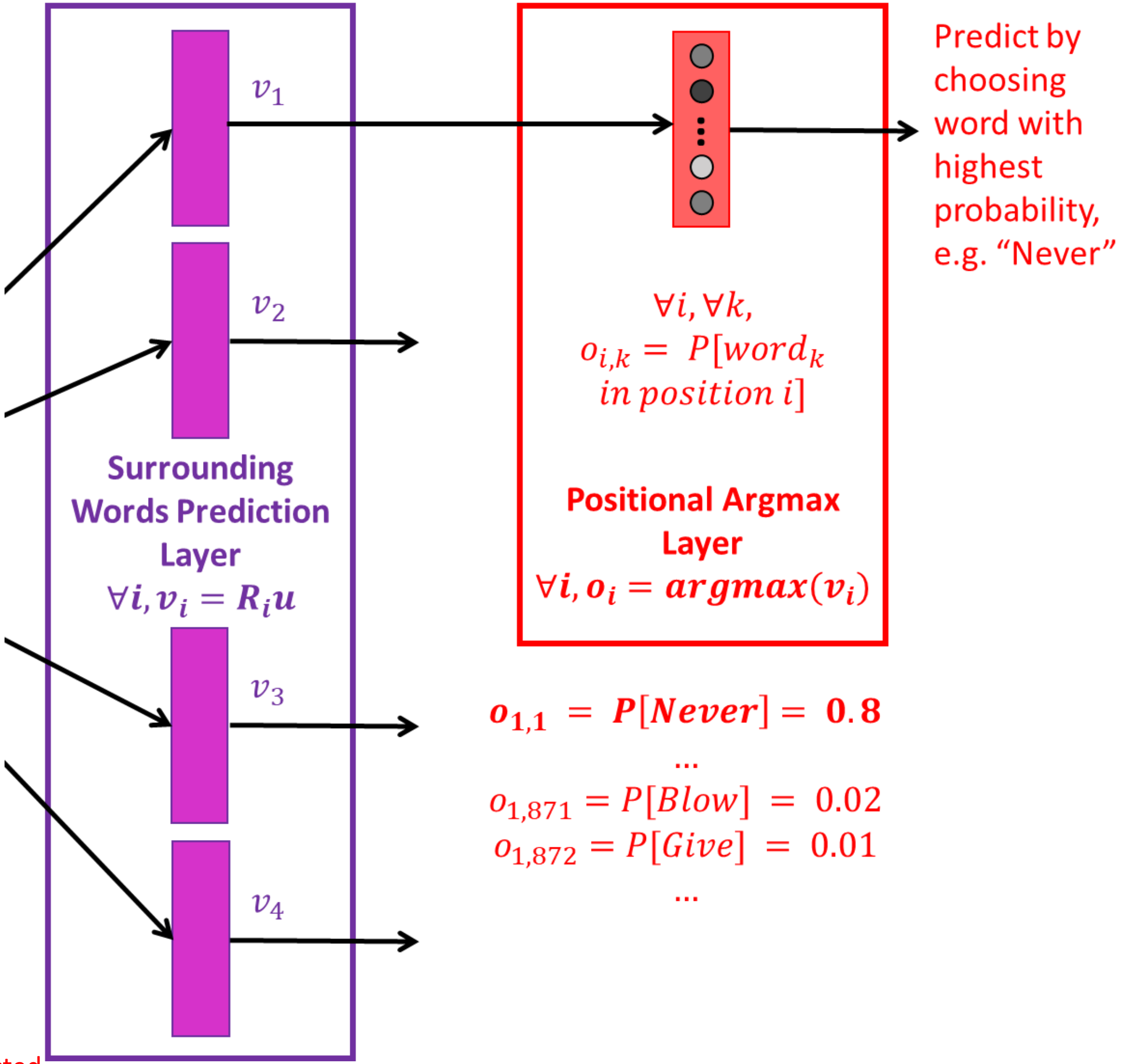


Restricted

SkipGram (SG)

Important Note

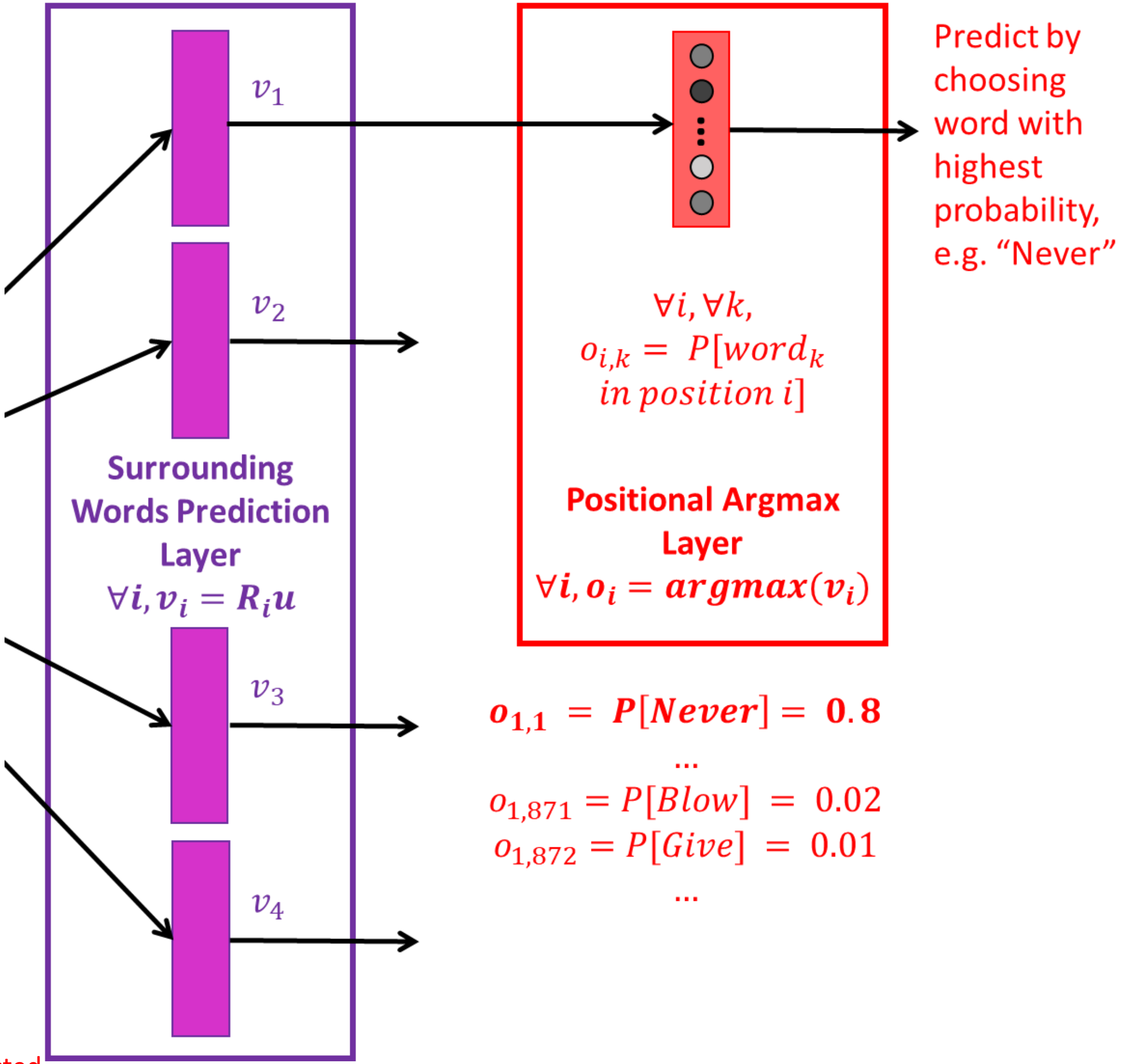
- **As an alternative, v_i could also be the same vector for each position i.** This is easily done by using an Embedding layer in PyTorch, which then only has a single training parameter $R \in \mathbb{R}^{D \times |V|}$.



SkipGram (SG)

Important Note

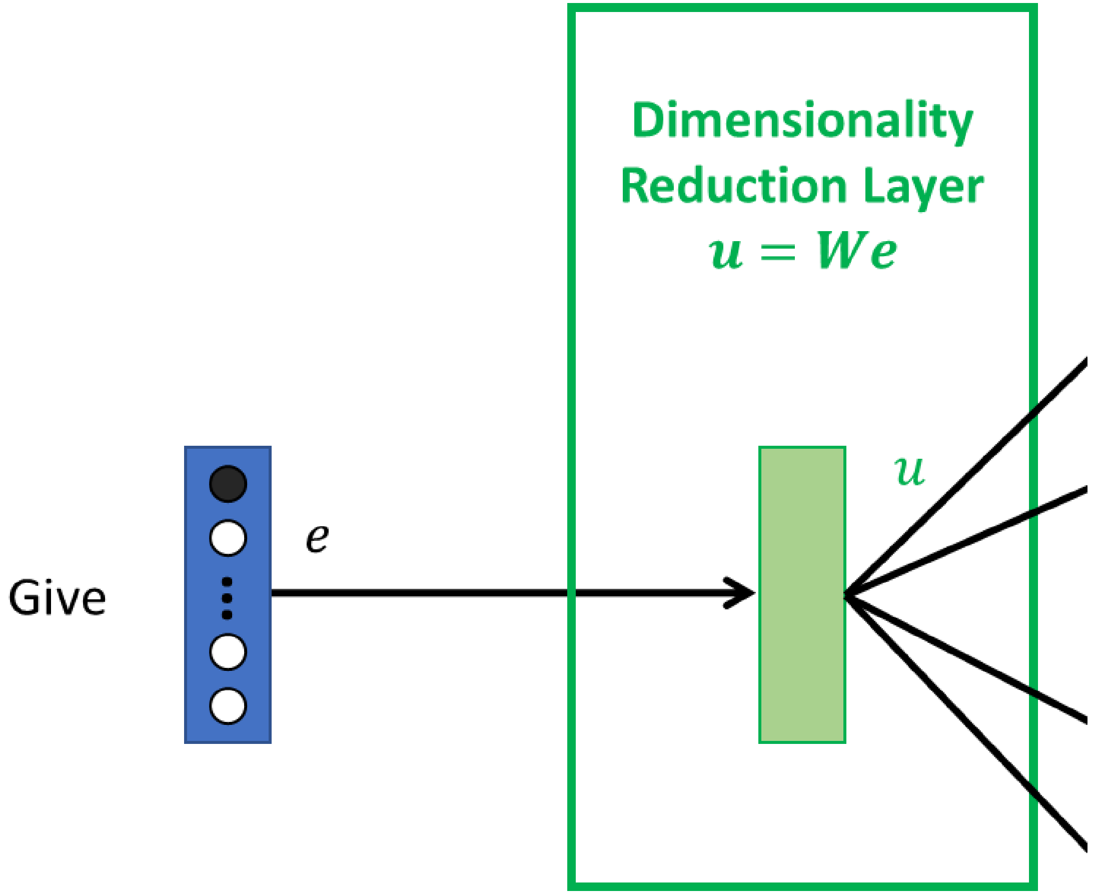
- **Choosing the Linear layers or Embedding layer route has little to no importance here (<u>as long as you correctly implement the loss function you need to use!</u>).**



SkipGram (SG)

Important Note

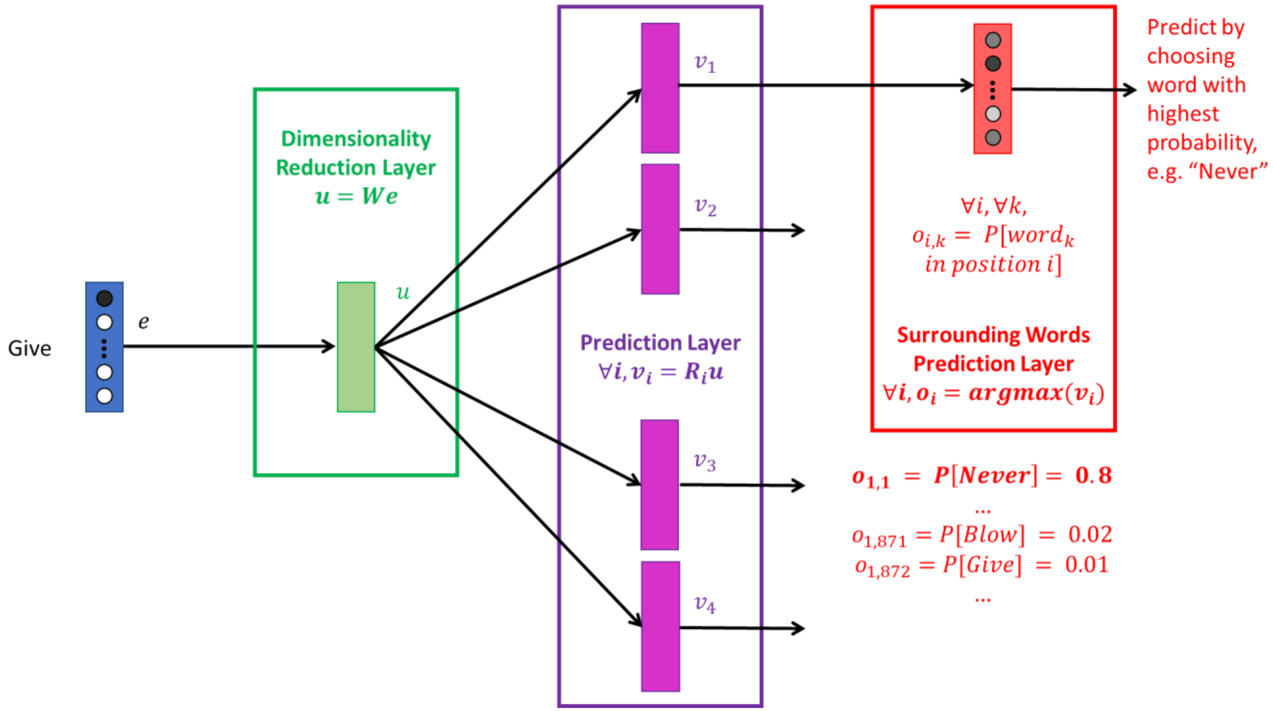
- **Choosing the Linear layers or Embedding layer route has little to no importance here (<u>as long as you correctly implement the loss function you need to use!</u>).**
- **All we want is to train a good Embedding W (!); the order of output words does not matter.** We only care that the probability is high for the right $2k$ output words which are around e.



SkipGram (SG)

Important Note

- This also means that training a SkipGram is far more difficult than training a CBoW!
- It is however producing better encodings, as it produces embedding that are able to describe any word by (somewhat) predicting the surrounding words around it.

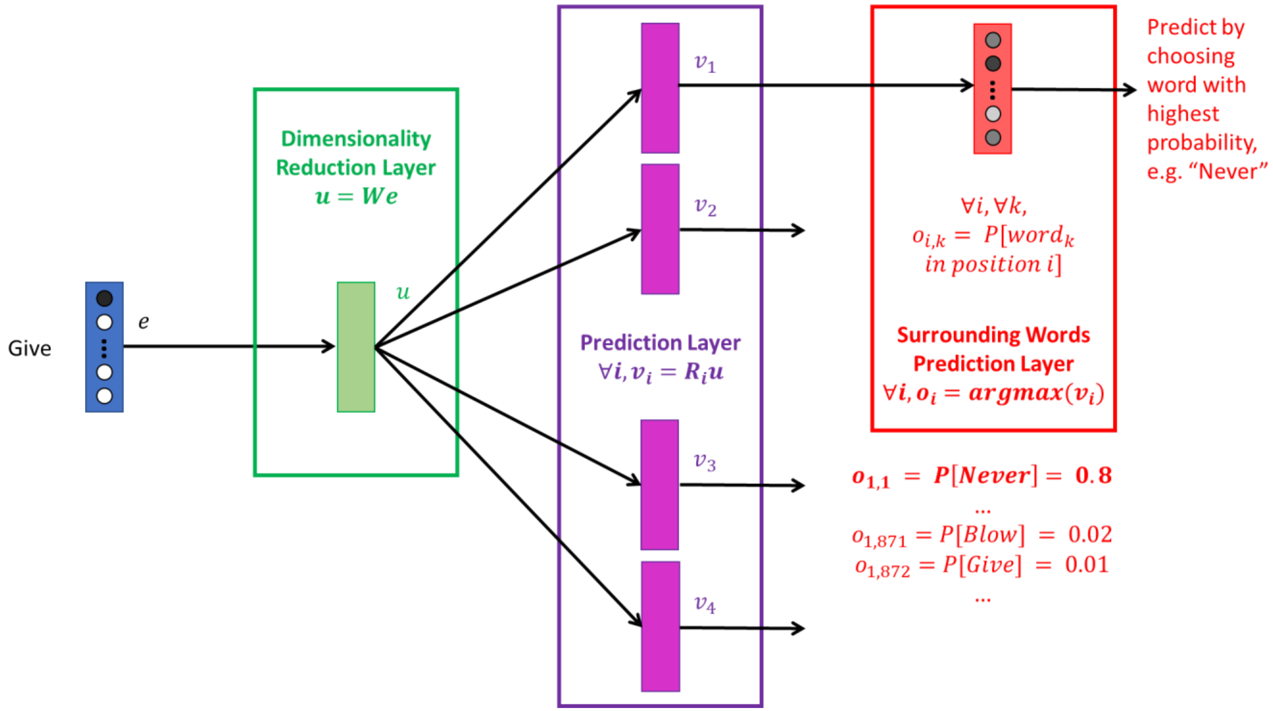


SkipGram (SG)

3. During the training, we browse through all the pairs (x, y) we generated on Step 0.

   Just like before, it is advisable to use some sort of a negative logarithm as the loss function, as it is a classification task.
   It is used on each word in each position i!

   And sum over all the pairs (x, y).



Homework this week will require to train a SkipGram (correctly!)

HW9!

Word2Vec and Glove

- The two approaches (CBoW and SkipGram) are commonly referred to as **Word2Vec** approaches.
- Another option, often referred to in literature is **GloVe** (Global Vectors for Word Representation).
- Another “simple” count-based embedding [GloVe2014] and was considered a good challenger to Word2Vec.

Choose a Pre-Trained Embedding If

- Your dataset is composed of more “general” language and you don’t have that big of a dataset, to begin with.
- Since these embeddings have been trained on a lot of words from different sources, pre-trained models might do well if your data is generalized as well.
- Also, you will save on time and computing resources with pre-trained embeddings.

Word2Vec and Glove

Choose to Train Your Own Embedding If

- Your data (and project) is based on a niche industry, such as medicine, finance or any other non-generic and highly specific domains.
- In such cases, a general word embedding representation might not work out for you and some words might be altogether missing from the pre-trained embeddings.

Word2Vec and Glove

Choose to Train Your Own Embedding If

- Your data (and project) is based on a niche industry, such as medicine, finance or any other non-generic and highly specific domains.
- In such cases, a general word embedding representation might not work out for you and some words might be altogether missing from the pre-trained embeddings.

Keep in mind, however,…

- On the downside, a lot of data is needed to ensure that the word embeddings being learned do a proper job of representing the various words and the semantic relationships between them, unique to your domain.
- Also, it takes a lot of computing resources to go through your corpus and build word embeddings.

Word2Vec and Glove

- Both Word2Vec and GloVe approaches are commonly referred to as **unsupervised (or semi-supervised) approaches to embedding,** both relying on the **distributional hypothesis** we mentioned on the previous lecture.
- Unsupervised representation learning of sentences had been the norm for quite some time.
- **More advanced versions of unsupervised approaches exist!**

- For instance: **FastText** and **ELMo**.

FastText

- **FastText** was developed by the team of Mikolov (him again!), triggering the explosion of research on **universal word embeddings** [Mikolov2017, Bojanovski2017].

Universal embeddings?

- A huge trend in DL/NLP is the quest for **Universal Embeddings.**

Definition (Universal Embedding):

Universal Embeddings are **embeddings that are pre-trained** on a large corpus and can be plugged in a variety of downstream task models to automatically improve their performance, by incorporating some general word/sentence representations learned on larger datasets.

In a sense, it is the ultimate form of **transfer learning** for language embeddings, the holy grail, i.e. a unique **universal embedding** that everyone can use on any task related to language.

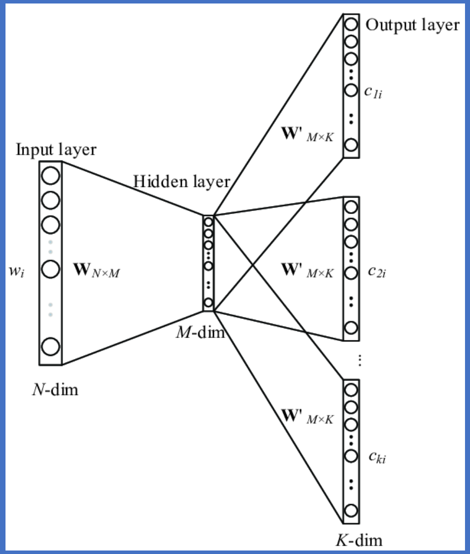
Open question: Are universal embeddings even possible or good? Should not they be task-specific?

FastText

- **FastText** was developed by the team of Mikolov (him again!), triggering the explosion of research on **universal word embeddings** [Mikolov2017, Bojanovski2017].
- Inclusion of character **n-grams**, which allows computing word representations for words that did not appear in the training data (something called **"out-of-vocabulary"** words).

Input: Concatenation of word + its n-grams.
(e.g. **"eating"** + "ea" + "eat" + "ati" + "tin" + "ing")

Model: Skip-Gram Architecture
(Predict context words based on single word + its n-grams)



Output: 2k words of context.

"Out of vocabulary" words

Definition (out-of-vocabulary):

Embeddings are often subject to the **out-of-vocabulary** issue.

This is a simple concept: what should the embedding function be doing if it is asked to encode a word that is not contained in the original dictionary V?

(Think new words, typos, etc.)

Word $w \notin V$ → Embedding function $f(w)$ → ???

"Out of vocabulary" words

Definition (out-of-vocabulary):

Embeddings are often subject to the **out-of-vocabulary** issue.

This is a simple concept: what should the embedding function be doing if it is asked to encode a word that is not contained in the original dictionary V?

(Think new words, typos, etc.)

In the case of Word2Vec (CBoW and SkipGram algorithms), this would be very invalidating, as we would have no way to one-hot encode the missing word.

Word $w \notin V$ → Embedding function $f(w)$ → ???

“Out of vocabulary” words

Definition (out-of-vocabulary):

Embeddings are often subject to the **out-of-vocabulary** issue.

This is a simple concept: what should the embedding function be doing if it is asked to encode a word that is not contained in the original dictionary V?

(Think new words, typos, etc.)

In the case of Word2Vec (CBoW and SkipGram algorithms), this would be very invalidating, as we would have no way to one-hot encode the missing word.

A good embedding function should then be able to, at least partially, operate on out-of-vocabulary words.

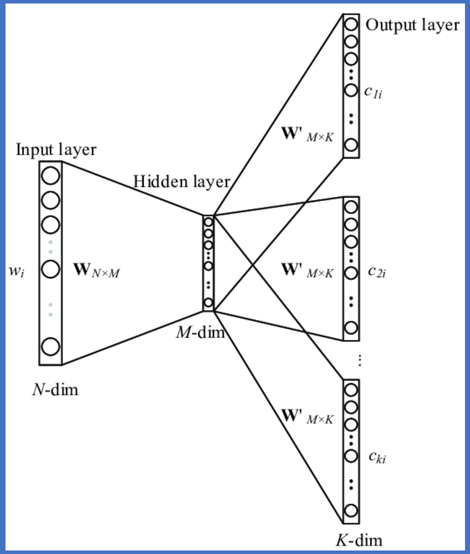
Word $w \notin V$ → Embedding function $f(w)$ → ???

Core idea behind FastText (out-of-class)

- FastText achieves really good performance for word representations, especially in the case of rare words by making use of character level information.
- Each word is represented as a bag of characters **n-grams** in addition to the word itself.
- For example, for the word "computer", with **n = 3**, the FastText representations for the character **n-grams** is <co, com, omp, mpu, put, ute, ter, er>.

Input: Concatenation of word + its n-grams.
(e.g. **"eating"** + "ea" + "eat" + "ati" + "tin" + "ing")

Model: Skip-Gram Architecture
(Predict context words based on single word + its n-grams)



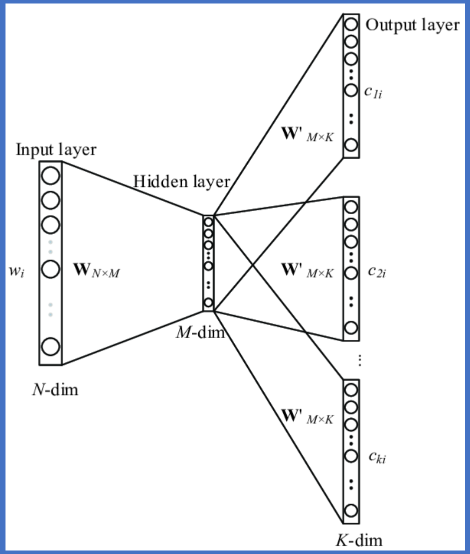
Output: 2k words of context.

Core idea behind FastText (out-of-class)

- Then, FastText uses the same approach and logic as in SkipGram, to try and predict some context words, by using the word and its **n-grams**.
- This typically allows to identify **etymology** in words, and **cover for potential typos**!
- Overall reinforces the performance, allows to **address** the **out of vocabulary issue**!

Input: Concatenation of word + its n-grams.
(e.g. **“eating”** + “ea” + “eat” + “ati” + “tin” + “ing”)

Model: Skip-Gram Architecture
(Predict context words based on single word + its n-grams)



Output: 2k words of context.

FastText - Recap

- **FastText** was developed by the team of Mikolov (him again!), triggering the explosion of research on **universal word embeddings** [Mikolov2017, Bojanovski2017].
- Inclusion of character **n-grams**, which allows computing word representations for words that did not appear in the training data (something called **"out-of-vocabulary"** words).
- FastText vectors are super-fast to train and are available in 157 languages. They are a great baseline, not sacrificing too much performance.
- Ready-to-use FastText embeddings from: https://github.com/facebookresearch/fastText
- Official (and good) tutorial: https://fasttext.cc/docs/en/unsupervised-tutorial.html

FastText - Demo

Download the model

This command will download a pre-trained english language model and save it to file.

Note: heavy model, takes a while.

```
# Load model
lang = 'en'
fasttext.util.download_model(lang, if_exists = 'ignore')  # English
model = fasttext.load_model('cc.en.300.bin')
```

Getting a vector embedding for word

The command below can be used to get the word embedding for any word.

```
# Get vector embedding for word
word = "hello"
v = model.get_word_vector(word)
print(v)
```

```
# Search for the closest word, i.e. the one with the highest cosine
# similarity scores with our given word vector
max_val = 0
best_match = model.words[0]
for index, word in enumerate(model.words):
    vec2 = model.get_word_vector(word)
    val = cos_sim(vec, vec2)
    if(val > max_val):
        max_val = val
        best_match = word
    if(index % 50000 == 0):
        pct = round(index/2000000*100, 1)
        print("- Progress {}/{} [{}%]".format(index, 2000000, pct)
print("Studying vector: ", vec)
print("Best match is: ", best_match)
print("Cosine similarity with best match: ", max_val)
```

Have a look at Notebook 2, to see how we may **download and reuse a pre-trained language model** in Python, to **produce** or **decode word embeddings**.

ELMo

The **Deep Contextualized Word Representations (a.k.a. Embeddings for Language Model - ELMo)** have recently improved the state of the art in word embeddings by a noticeable amount.

They were developed by the Allen institute for AI and were presented at NAACL 2018 in early June [Peters2018].

- Ready-to-use ELMo embeddings https://allennlp.org/elmo
- Official and good tutorial: https://guide.allennlp.org/

ELMo can take context into account

- So far, all the language models we have seen produced embeddings for words…
- **But the embeddings would not change based on context (surrounding words in sentence).**
- This means that homonyms will have identical embeddings, despite having very different meanings.

ELMo can take context into account - A blast from the past (W9S1)

Problem #1: Identical words can have multiple (and sometimes very different) meanings.

- And since the meaning of the word depends on context…
- This means that **our embedding function f should not just take a single word x as input to produce an embedding vector x' for said word x.**

*I ate a **club** sandwich yesterday.*

*I broke my golf **club** yesterday night.*

- Maybe it would be preferable to have the word in question, plus some surrounding words for context instead as inputs for the embedding function.

$$g: V^n \rightarrow \mathbb{R}^m$$

$$g(my, golf, \mathbf{club}, yesterday, night) = (0, 0, \ldots, 0,1,0, \ldots, 0) = \mathrm{e_k}$$

$$g(ate, a, \mathbf{club}, sandwich, yesterday) = (0, 0, 1, 0, \ldots, 0) = \mathrm{e_{k'}}$$

With $k \neq k'$

Restricted

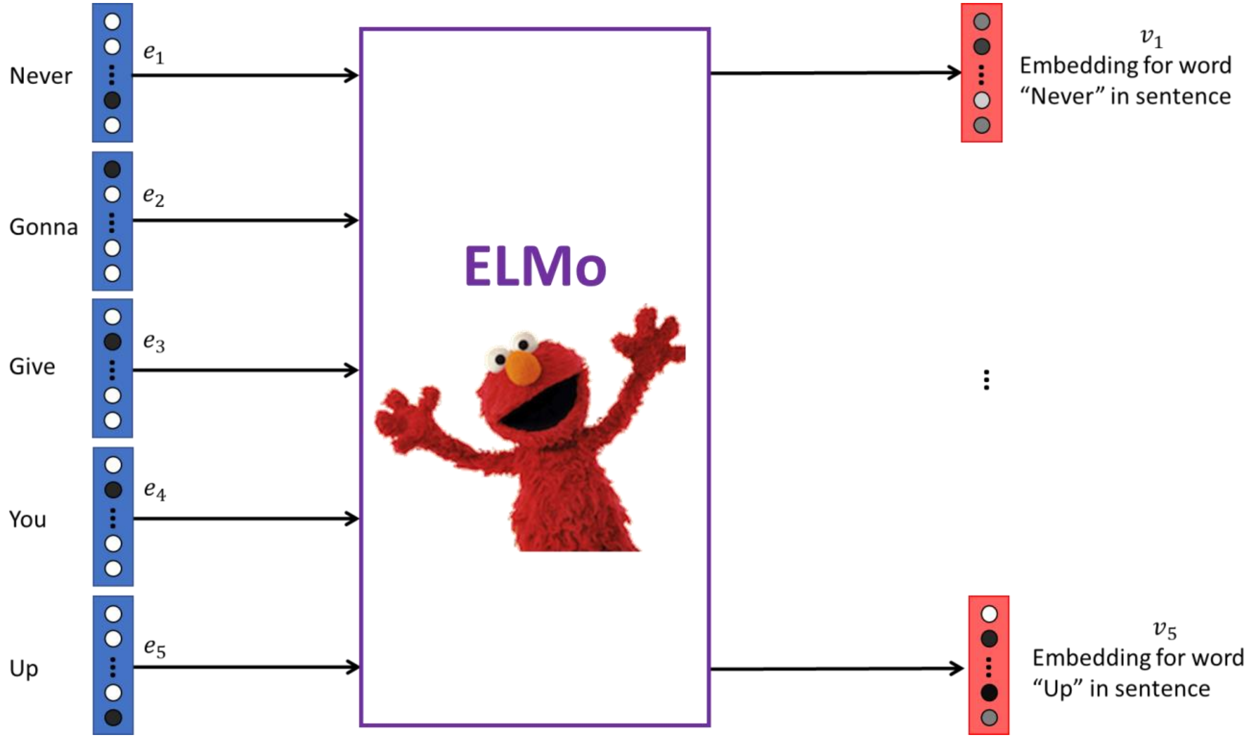
ELMo can take context into account

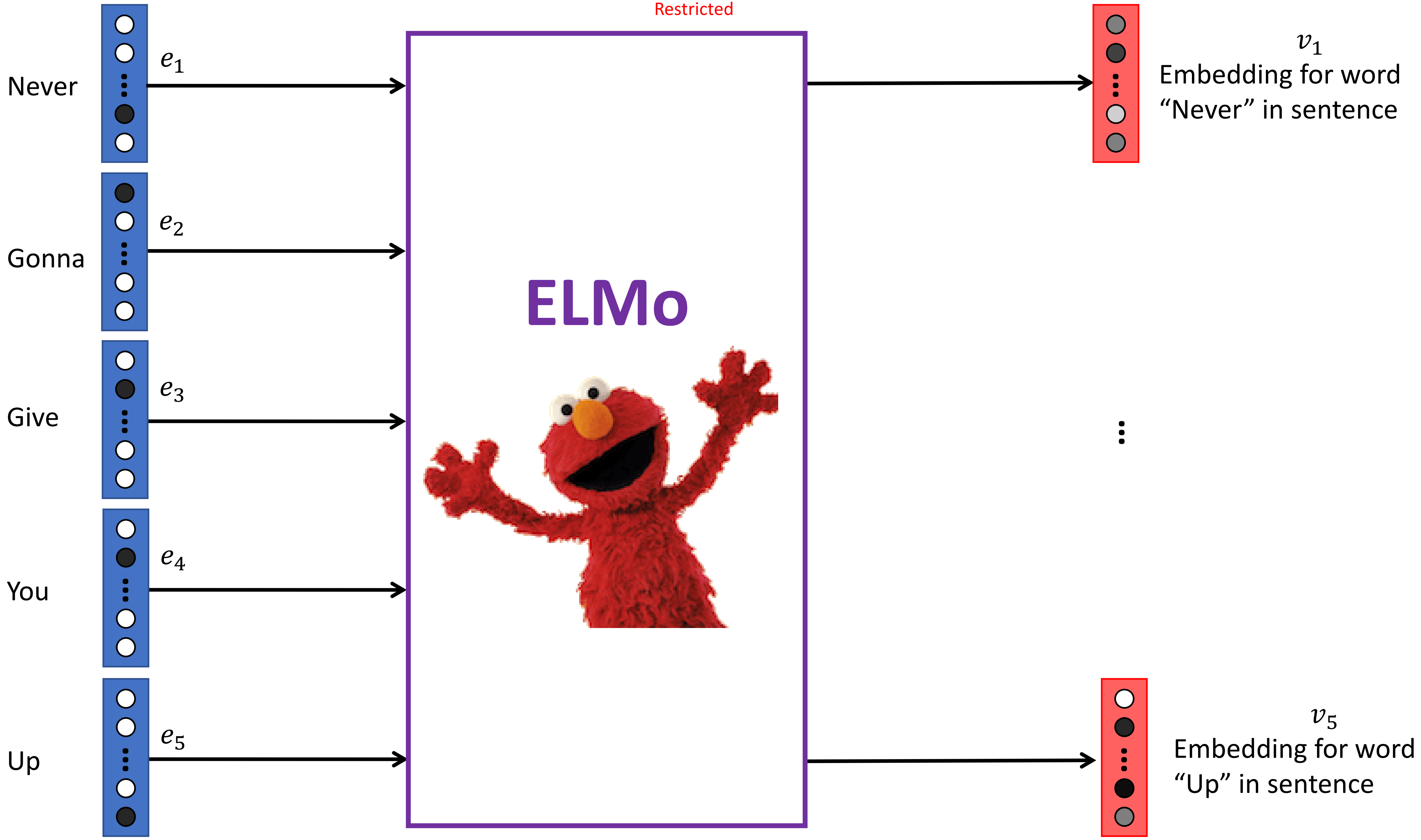
- So far, all the language models we have seen produced embeddings for words…
- **But the embeddings would not change based on context (surrounding words in sentence).**
- This means that homonyms will have identical embeddings, despite having very different meanings.

ELMo – Key takeaways

1. ELMo expects inputs of entire sentences and will then produce embeddings for all words in the sentence separately.

   To get an embedding of a word, input the whole sentence for context, then **extract only the embedding vector which corresponds to that word**.




Never
e_1
Gonna
e_2
Give
e_3
You
e_4
Up
e_5
ELMo
v_1
Embedding for word
“Never” in sentence
v_5
Embedding for word
“Up” in sentence

ELMo – Key takeaways

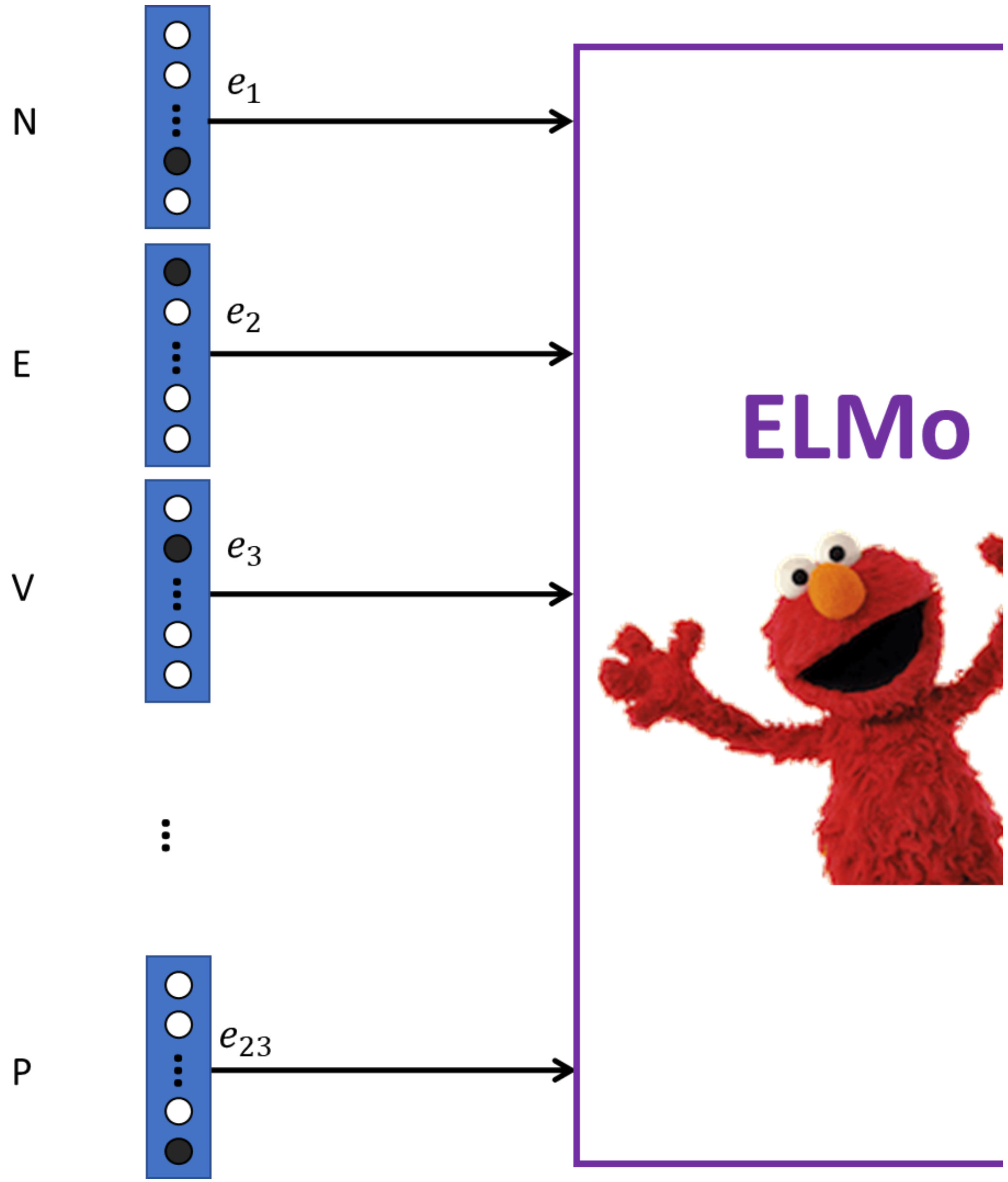
2. ELMo does not take in words inputs as one-hot vectors, instead, it will decompose them into characters.

ELMo – Key takeaways

2. ELMo does not take in words inputs as one-hot vectors, instead, it will decompose them into characters.

   The characters will then be represented as one-hot vectors and fed as inputs.

   (Which is nice, lower dimensionality!)



ELMo has lower dimensionality on inputs - Another blast from the past (W9S1)

Problem #4: Using $OH_{\mathbb{R}^{|V|}}$ to represent embeddings is seriously problematic, if we consider the issue of memory space…

- Languages contain millions (trillions if we include the typos, conjugations, acronyms, etc?) of possible words…
- We need a representation with lower dimensionality!

Rome Paris word V

```
Rome   = [1, 0, 0, 0, 0, 0, …, 0]
Paris  = [0, 1, 0, 0, 0, 0, …, 0]
Italy  = [0, 0, 1, 0, 0, 0, …, 0]
France = [0, 0, 0, 1, 0, 0, …, 0]
```

What is the size of this vector?!
How many zeros in there?!

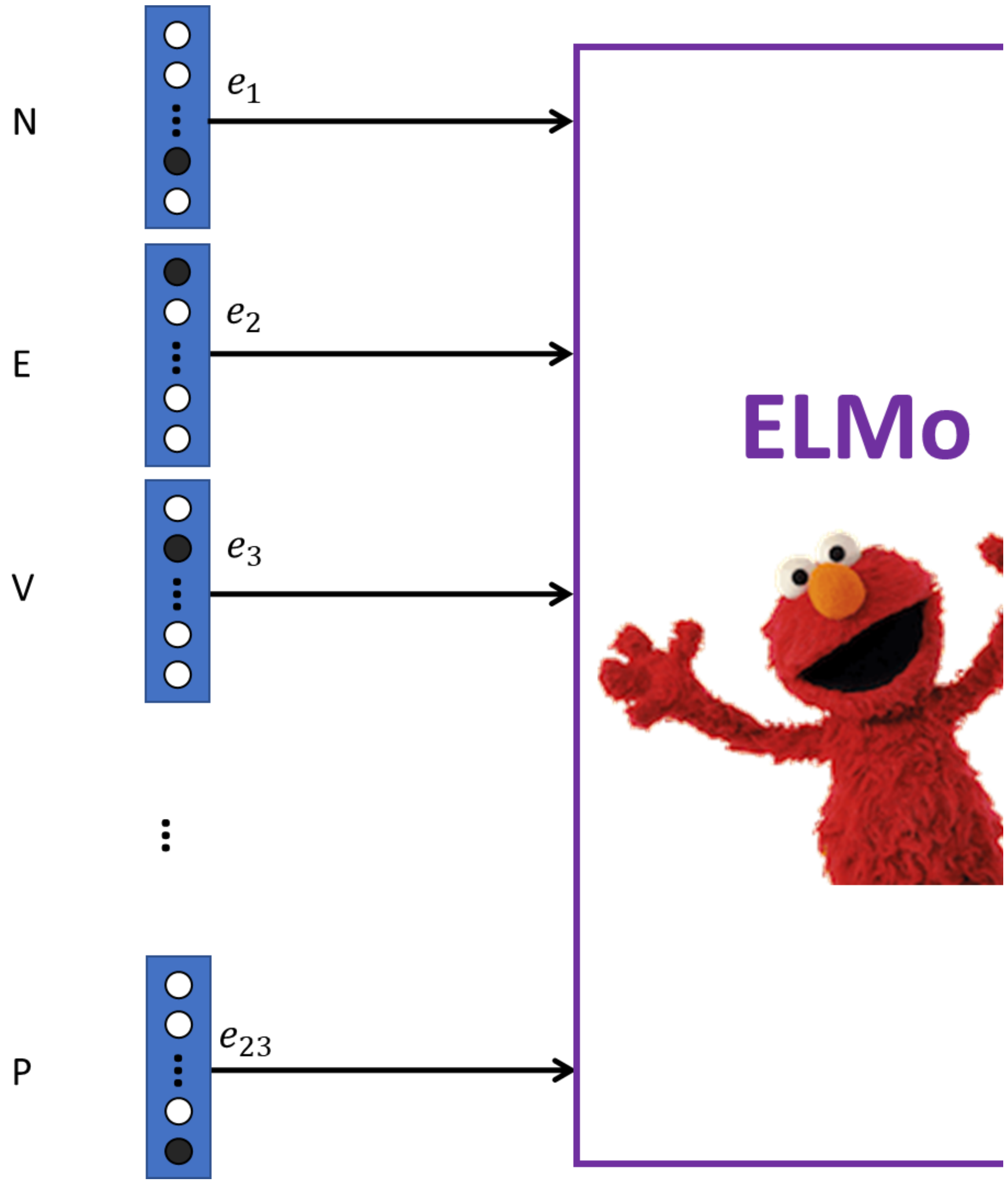
→ Characters, on the other end, can be represented as one-hot vectors, with much lower dimensionality!

ELMo – Key takeaways

2. ELMo does not take in words inputs as one-hot vectors, instead, it will decompose them into characters.

   The characters will then be represented as one-hot vectors and fed as inputs.

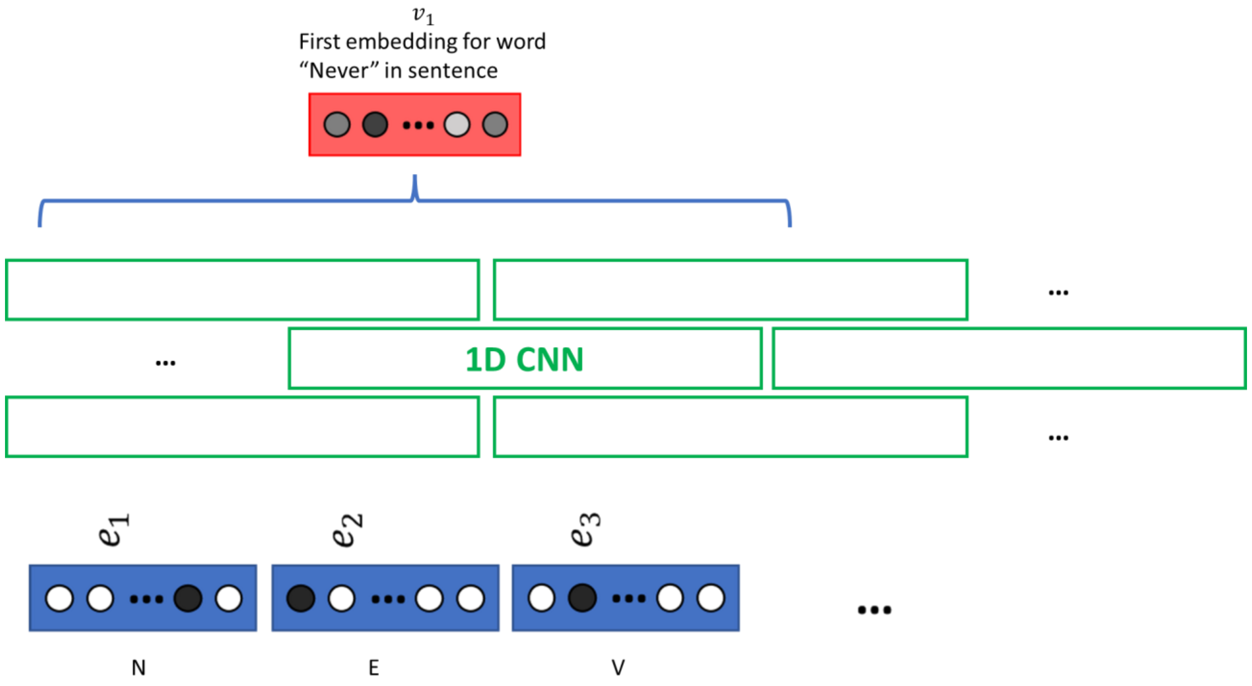
   (Which is nice, lower dimensionality!)

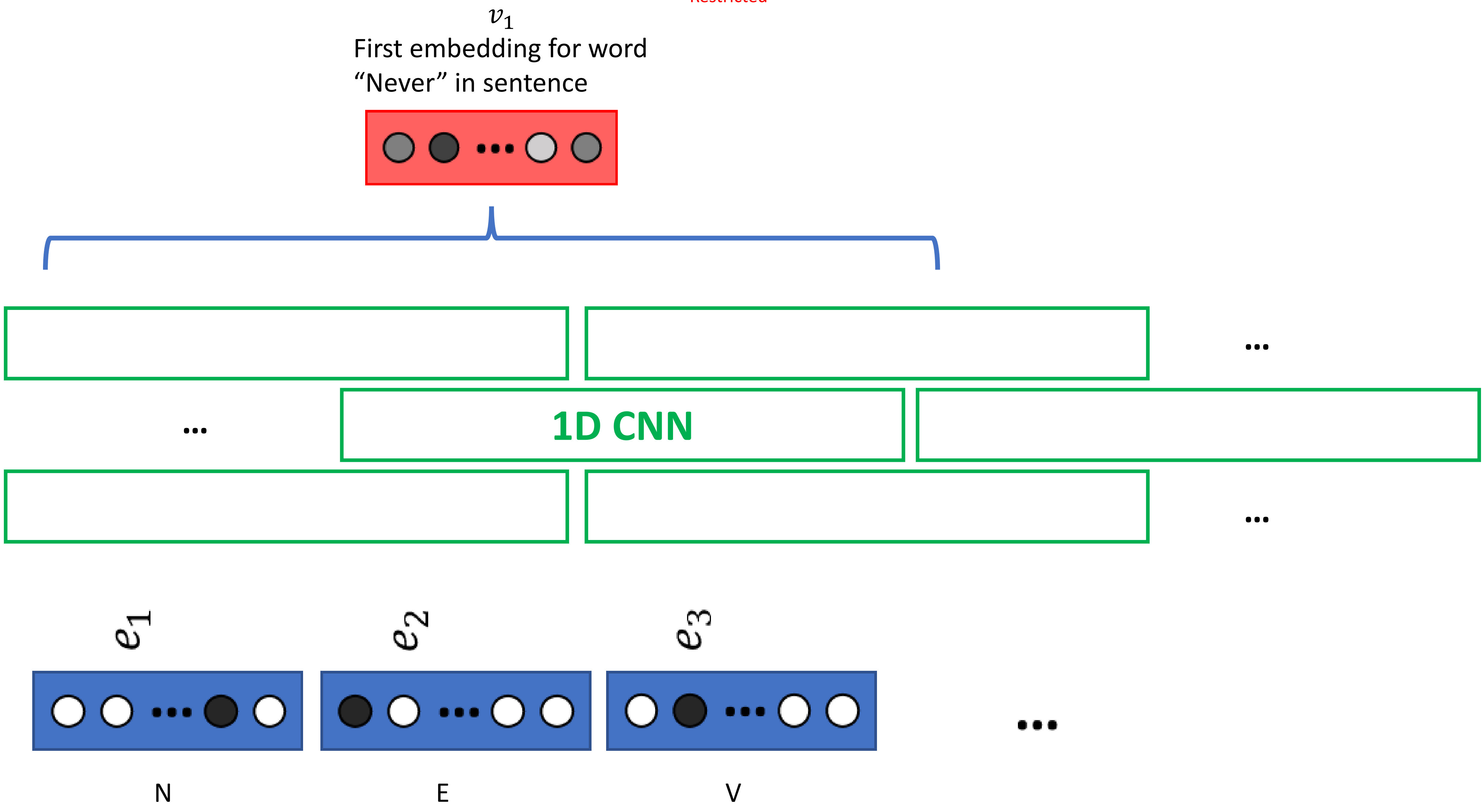


ELMo – Key takeaways

3. The bottom layer of ELMo will combine and process characters together.
   It consists of a succession of several (1-D) CNN layers.

   Intuition: combining characters allows to extract meaning through **etymology**! Also, it solves the **out-of-dictionary problem**!




v_1
First embedding for word
"Never" in sentence
1D CNN
e_1
e_2
e_3
N
E
V

ELMo – Key takeaways

Technicalities for 1D CNNs

- Character-level vectors goes through (1D) convolutional layers with different kernel sizes.
- The original "small" ELMo model used kernels of size 1, 2, 3, 4, 5, 6, 7 with 32, 32, 64, 128, 256, 512, 1024 channels, respectively.
- Outputs from each convolutional layers are then max-pooled and concatenated.

- The final concatenated vector, of size 2048, can be used as a first word embedding. **<u>But it does not benefit from context yet</u>**.

- **Remember the convolution layers lectures:** convolutional layers are well known for their feature-extracting properties… This first operation can therefore be regarded as a character-level context extraction process!

ELMo – Key takeaways

From Notebook 3

- In our implementation, kernel sizes are slightly different, and lead to a 128-length vector in the end.
- Interleaving with Max-Pool as suggested in the "original" ELMo model.
- Adding an embedding layer at the beginning.

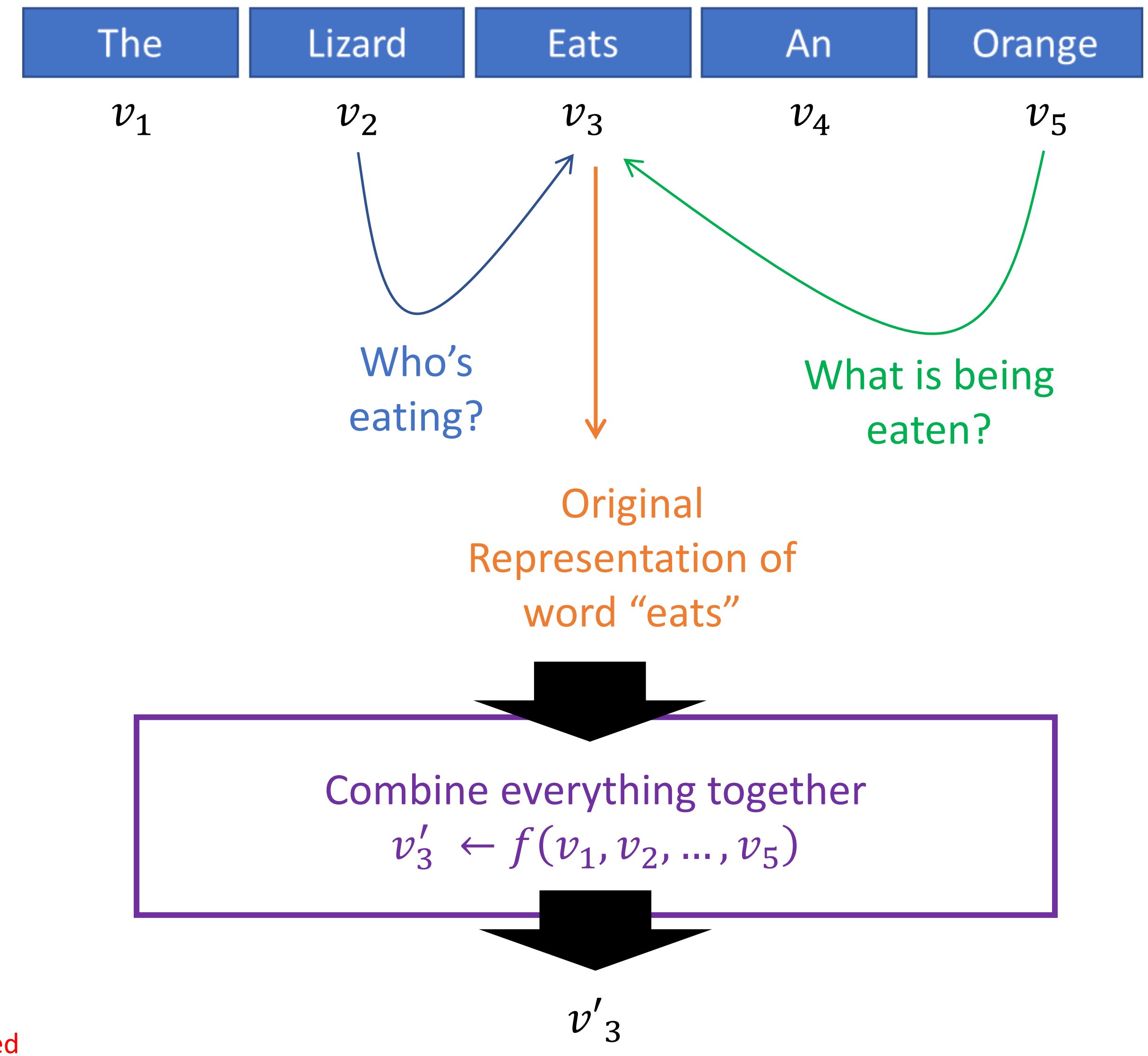
```
class CharConv(nn.Module):

    def __init__(self):
        super().__init__()

        # Embedding layer to start with
        self.char_embedding = nn.Embedding(CHAR_VOCAB_SIZE, CHAR_EMBED_DIM)

        # Some convolution layers
        self.conv1 = nn.Conv2d(CHAR_EMBED_DIM, 2, 1)
        self.conv2 = nn.Conv2d(CHAR_EMBED_DIM, 2, (1, 2))
        self.conv3 = nn.Conv2d(CHAR_EMBED_DIM, 4, (1, 3))
        self.conv4 = nn.Conv2d(CHAR_EMBED_DIM, 8, (1, 4))
        self.conv5 = nn.Conv2d(CHAR_EMBED_DIM, 16, (1, 5))
        self.conv6 = nn.Conv2d(CHAR_EMBED_DIM, 32, (1, 6))
        self.conv7 = nn.Conv2d(CHAR_EMBED_DIM, 64, (1, 7))
        self.convs = [self.conv1, self.conv2, self.conv3, self.conv4,
                      self.conv5, self.conv6, self.conv7,]


    def forward(self, x):
        # Character-level convolution
        # Starts with embeddings and some reshaping
        x = self.char_embedding(x).permute(0,3,1,2)
        # Go through all convolution layers
        x = [conv(x) for conv in self.convs]
        # Max Pooling
        x = [F.max_pool2d(x_c, kernel_size=(1, x_c.shape[3])) for x_c in x]
        # Concatenate/Squeeze into final vector
        # Final vector will be of size (1, n_batch, concat_length)
        x = [torch.squeeze(x_p, dim=3) for x_p in x]
        x = torch.hstack(x)
        return x
```

ELMo – Key takeaways

4. The top layer of ELMo will combine and transfer context from words to each other

What we want:

- a process or function that will transform the first word embedding we produced from the character-level CNN,
- Into a new word vector, which has transferred context from words with each other.

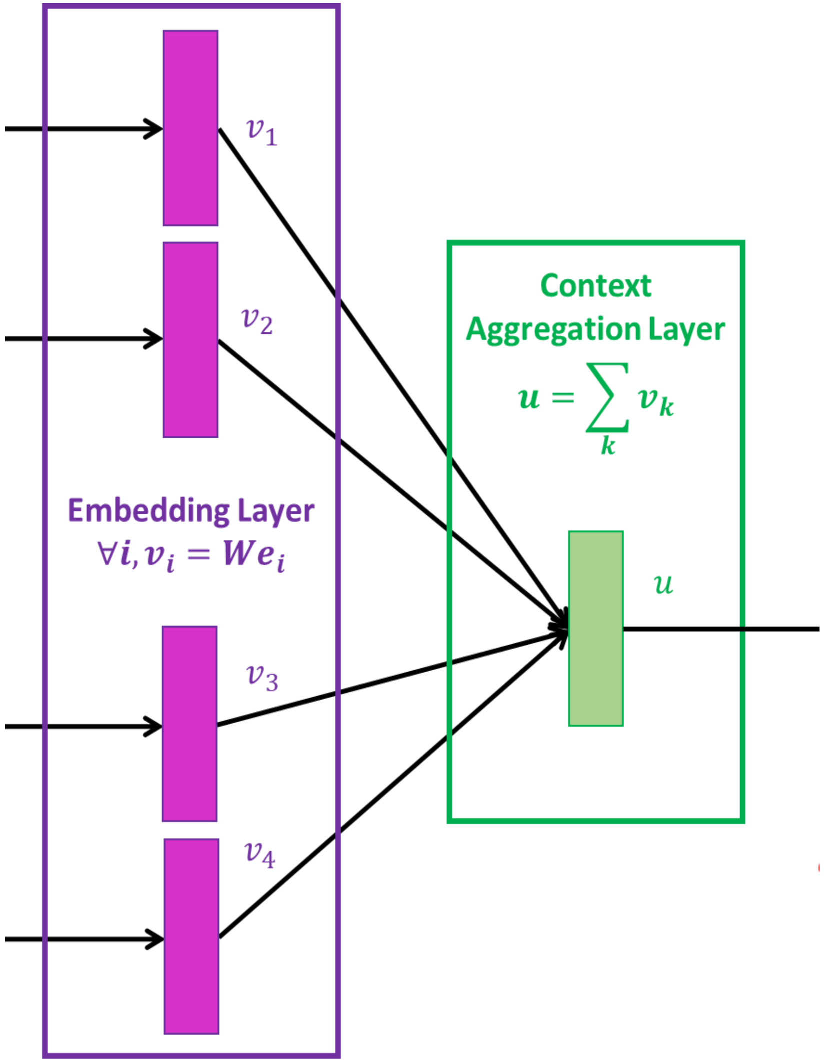


Context Aggregation in CBoW (Reminder)

- In CBoW, we would aggregate the context of the surrounding 2k words, by summing their word representation vectors into a single one.

$$u = \sum_k v_k$$

- Then, supposedly, we assumed u contained the information of the context and could be used to predict the missing word.



→ **Can we do better?**

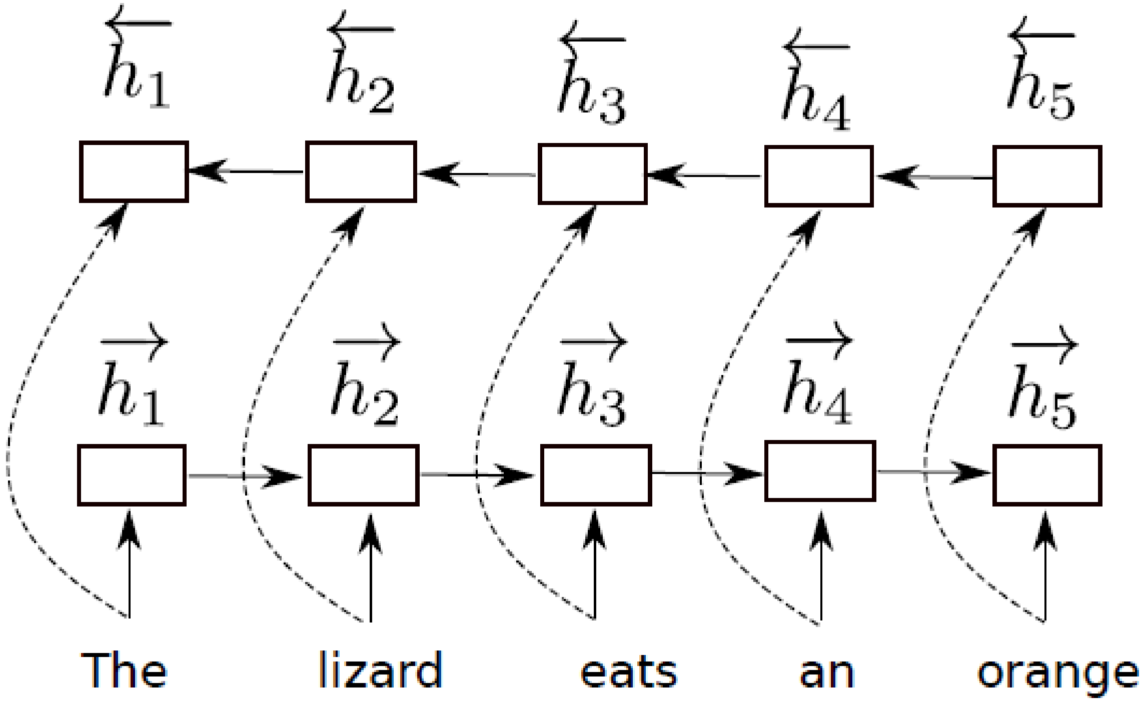
Context aggregation using Bi-LSTMs

- **Transfer information using RNNs? Yes, bidirectional LSTMs!**
- Two hidden states, going in two opposite directions, denoted $\overrightarrow{\dots}$ and $\overleftarrow{\dots}$ respectively.

$$\overrightarrow{h_k} = RNN_r(w_1, w_2 \dots, w_k)$$
$$\overleftarrow{h_k} = RNN_l(w_L, w_{L-1} \dots, w_k)$$
$$h_k = [\overleftarrow{h_k}, \overrightarrow{h_k}]$$

With L being the length of the sentence

$$h_k = [\overleftarrow{h}_k, \overrightarrow{h}_k]$$



Context aggregation using Bi-LSTMs

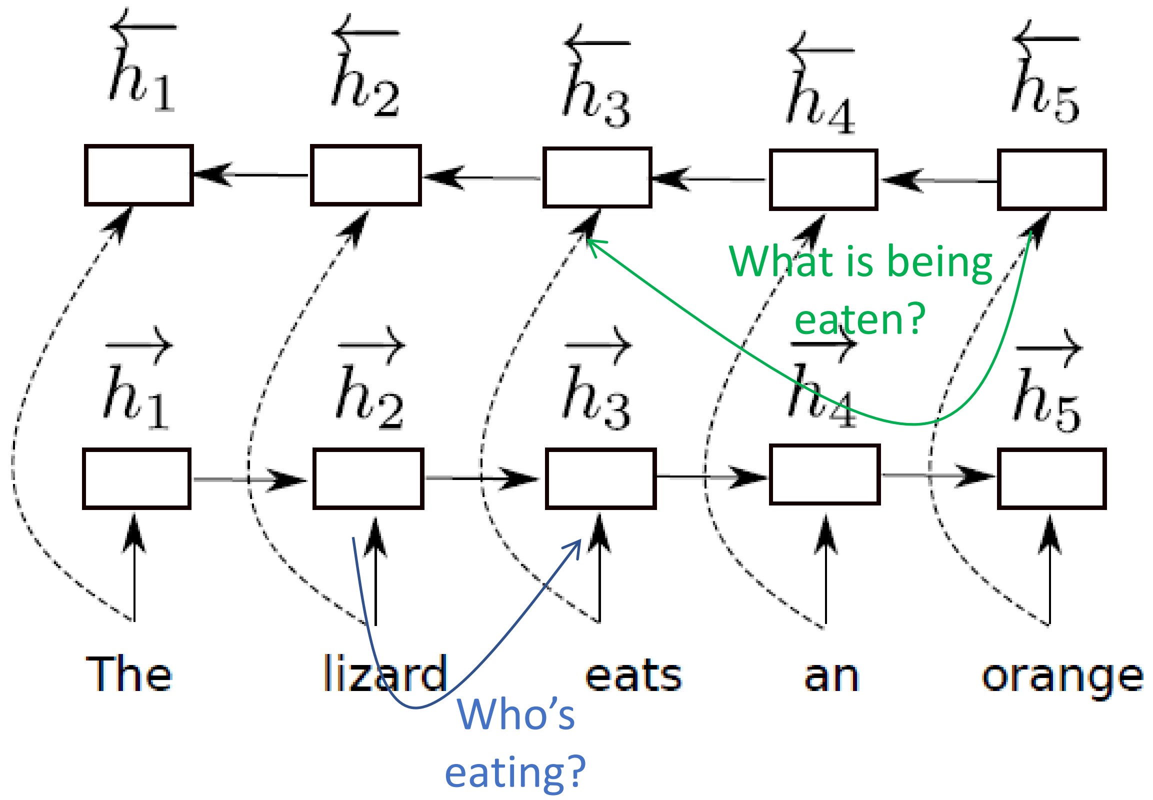
$$h_k = [\overleftarrow{h_k}, \overrightarrow{h_k}]$$

- **Transfer information using RNNs? Yes, bidirectional LSTMs!**
- Two hidden states, going in two opposite directions, denoted $\overrightarrow{\dots}$ and $\overleftarrow{\dots}$ respectively.

$$\overrightarrow{h_k} = RNN_r(w_1, w_2 \dots, w_k)$$
$$\overleftarrow{h_k} = RNN_l(w_L, w_{L-1} \dots, w_k)$$
$$h_k = [\overleftarrow{h_k}, \overrightarrow{h_k}]$$

With L being the length of the sentence

- Use them to **propagate context**!



Context aggregation using Bi-LSTMs

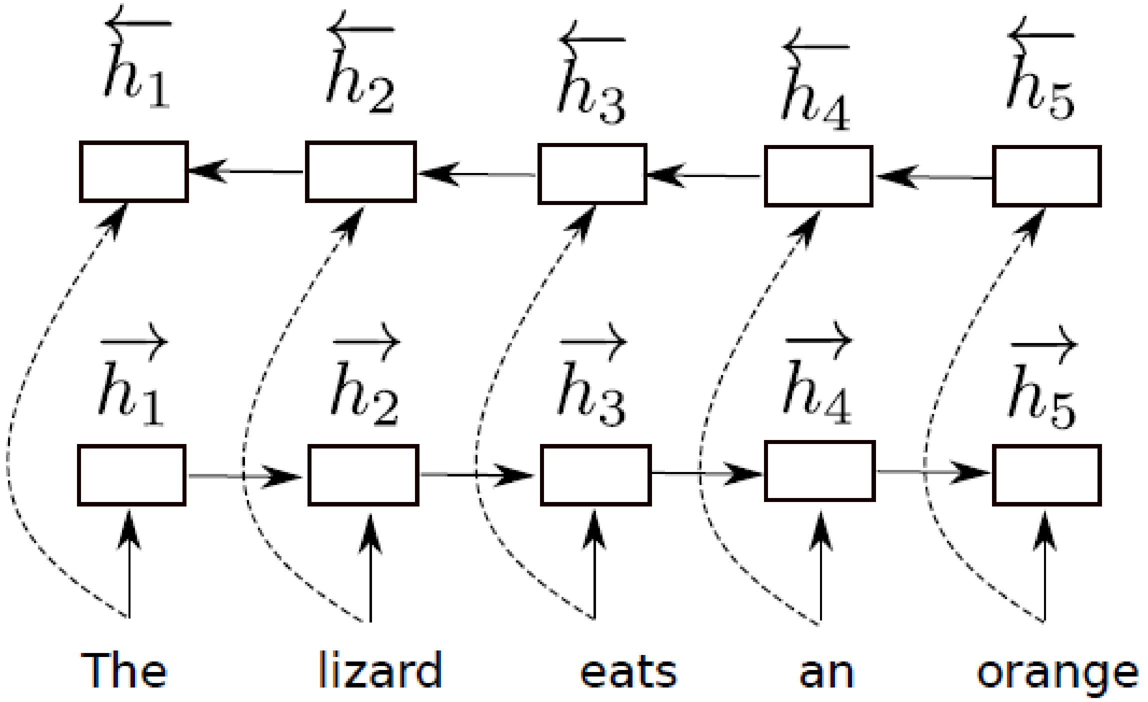
$$h_k = [\overleftarrow{h_k}, \overrightarrow{h_k}]$$

- **Transfer information using RNNs? Yes, bidirectional LSTMs!**
- Two hidden states, going in two opposite directions, denoted $\overrightarrow{\ldots}$ and $\overleftarrow{\ldots}$ respectively.

$$\overrightarrow{h_k} = RNN_r(w_1, w_2 \ldots, w_k)$$
$$\overleftarrow{h_k} = RNN_l(w_L, w_{L-1} \ldots, w_k)$$
$$h_k = [\overleftarrow{h_k}, \overrightarrow{h_k}]$$

With L being the length of the sentence

- Use them to propagate context!



→ **Supposedly better (more sophisticated and trainable), but heavier.**

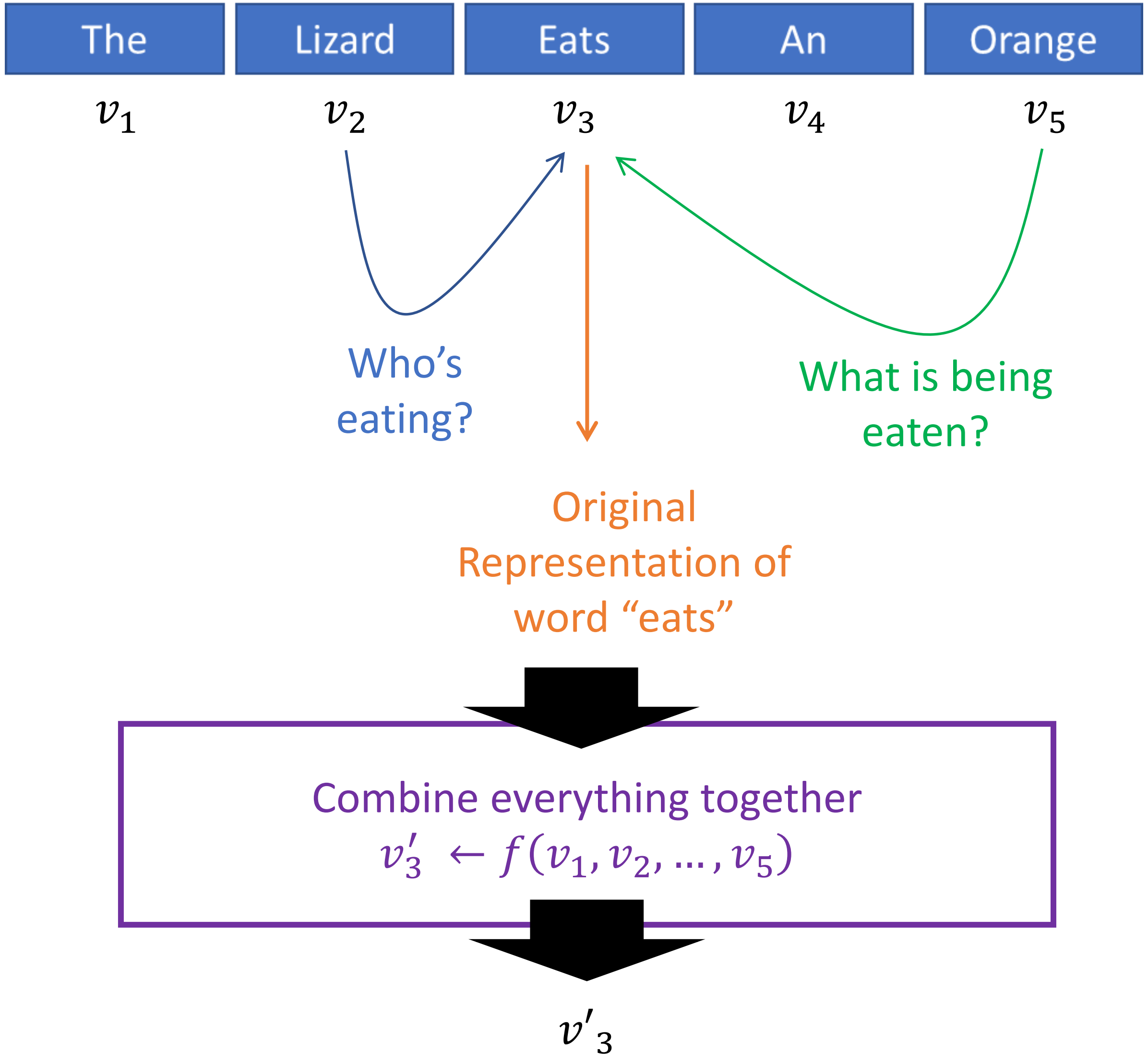
ELMo – Key takeaways

4. The top layer of ELMo will combine and transfer context from words to each other

What we want:

- a process or function that will transform the first word embedding we produced from the character-level CNN,
- Into a new word vector, which has transferred context from words with each other.

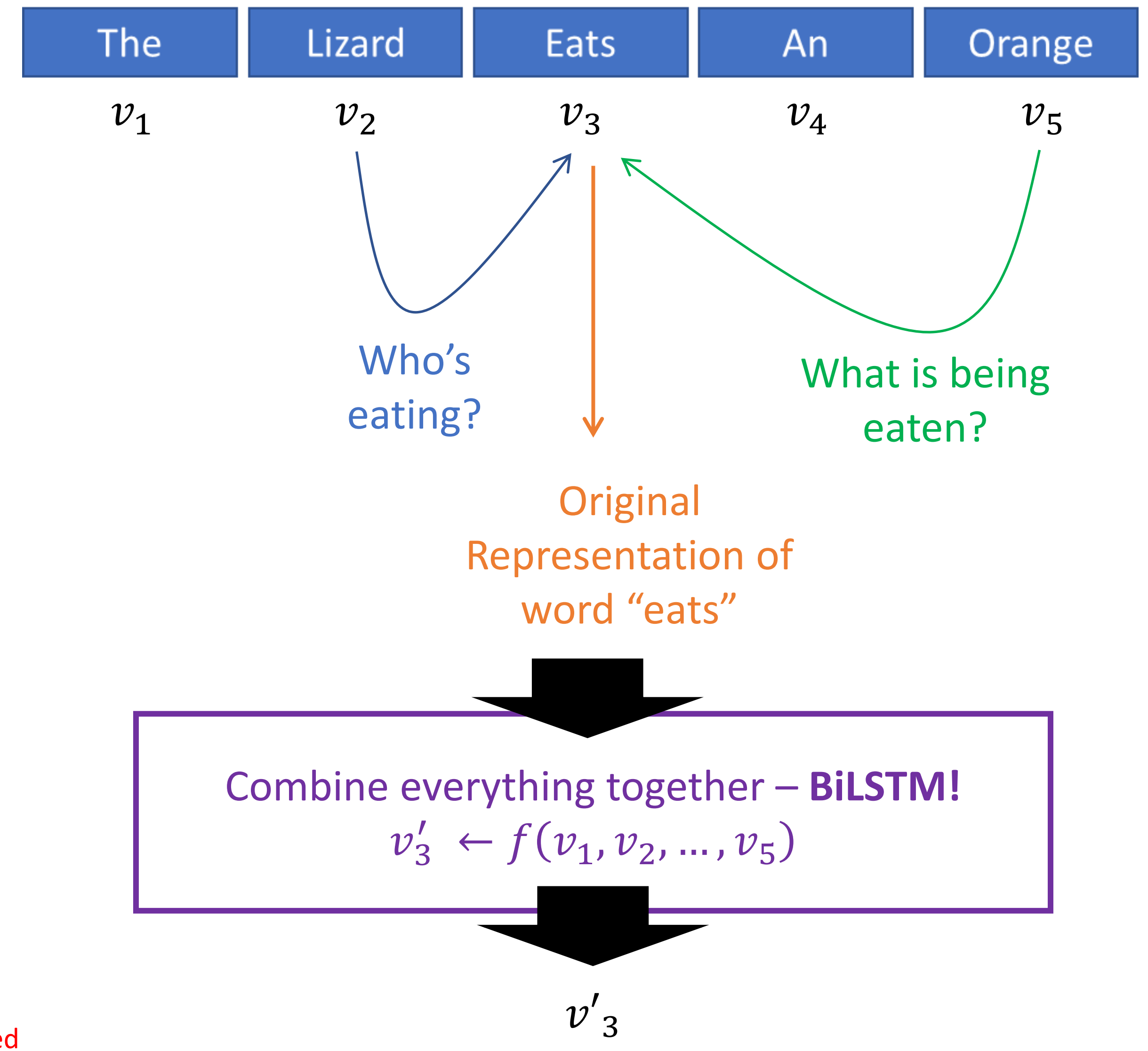
→ Use a bi-LSTM!



ELMo – Key takeaways

4. **Top layer of ELMo:** use a **bidirectional LSTM**, to propagate **some context using** for the word embeddings we had produced earlier.

   Final word embedding will be stored in $h_k = [\overleftarrow{h_k}, \overrightarrow{h_k}]$!



ELMo – Key takeaways

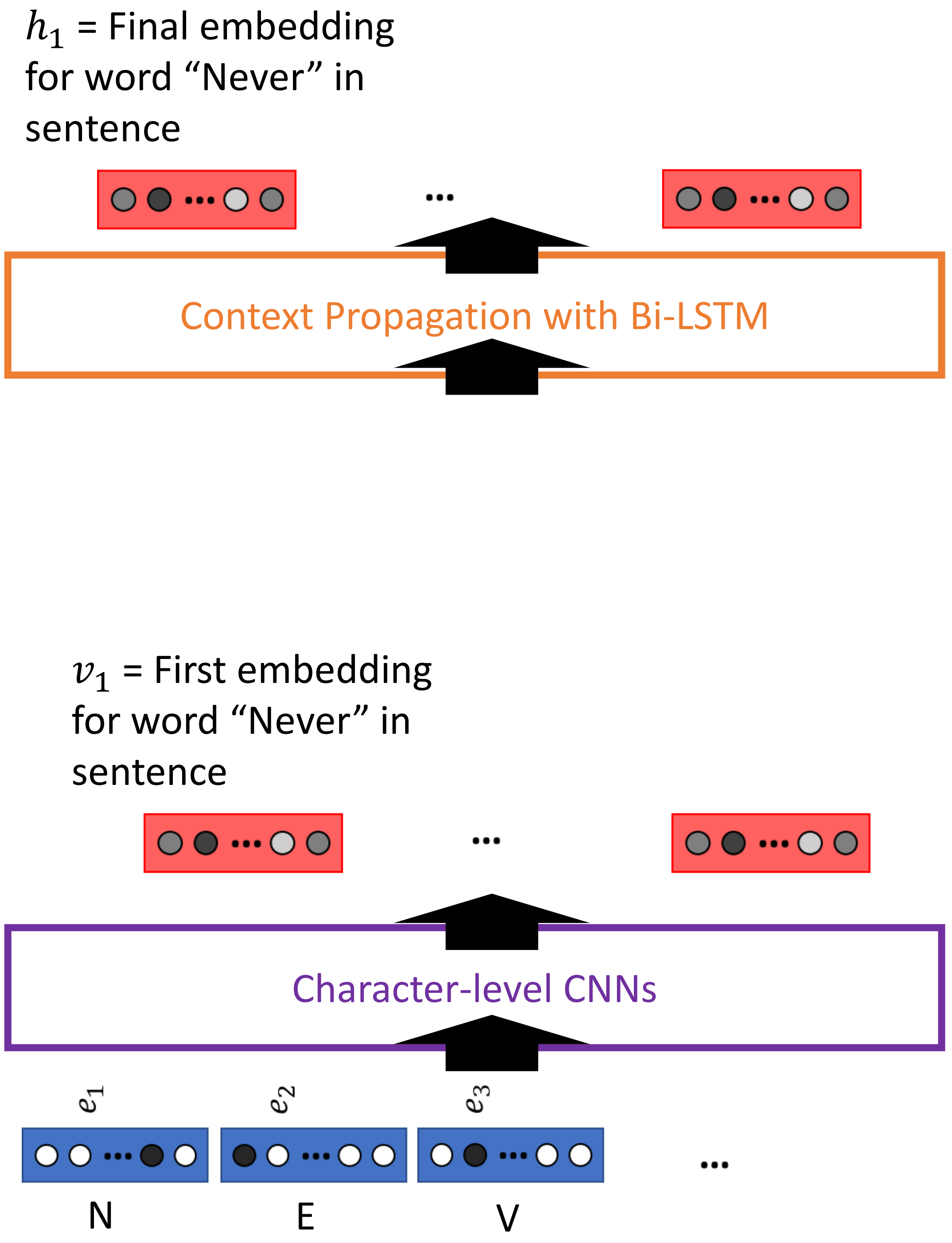
4. **Top layer of ELMo:** use a **bidirectional LSTM**, to propagate **some context using** for the word embeddings we had produced earlier.

   Final word embedding will be stored in $h_k = [\overleftarrow{h_k}, \overrightarrow{h_k}]$!

```python
class BiLSTM(nn.Module):

    def __init__(self):

        super().__init__()
        # To build a bi-directional LSTM, we will need a few LSTM layers
        self.lstm_f1 = nn.LSTM(128, 128)
        self.lstm_r1 = nn.LSTM(128, 128)
        self.dropout = nn.Dropout(0.1)
        self.proj = nn.Linear(128, 64, bias = False)
        self.lstm_f2 = nn.LSTM(64, 128)
        self.lstm_r2 = nn.LSTM(64, 128)

    def forward(self, x):
        # Note: we expect word embeddings of size 128 (as the result of the
        # previous character-level CNN network!)
        # input shape is then: (seq_len, batch_size, 128)

        # 1st LSTM layer - Forward feed LSTM + Dropout
        x_f = x
        o_f1, (h_f1, __) = self.lstm_f1(x_f)
        o_f1 = self.dropout(o_f1)

        # 2nd LSTM layer - Backward feed LSTM + Dropout
        x_r = x.flip(dims=[0])
        o_r1, (h_r1, __) = self.lstm_r1(x_r)
        o_r1 = self.dropout(o_r1)
        h1 = torch.stack((h_f1, h_r1)).squeeze(dim = 1)

        # Assemble
        x2_f = self.proj(o_f1 + x_f)
        x2_r = self.proj(o_r1 + x_r)

        # If we want, we can repeat the bi-directional LSTM
        # a second time (or more, if needed), as such.
        _, (h_f2, __) = self.lstm_f2(x2_f)
        _, (h_r2, __) = self.lstm_r2(x2_r)
        h2 = torch.stack((h_f2, h_r2)).squeeze(dim = 1)

        # Return both
        return h1, h2
```

ELMo – Key takeaways

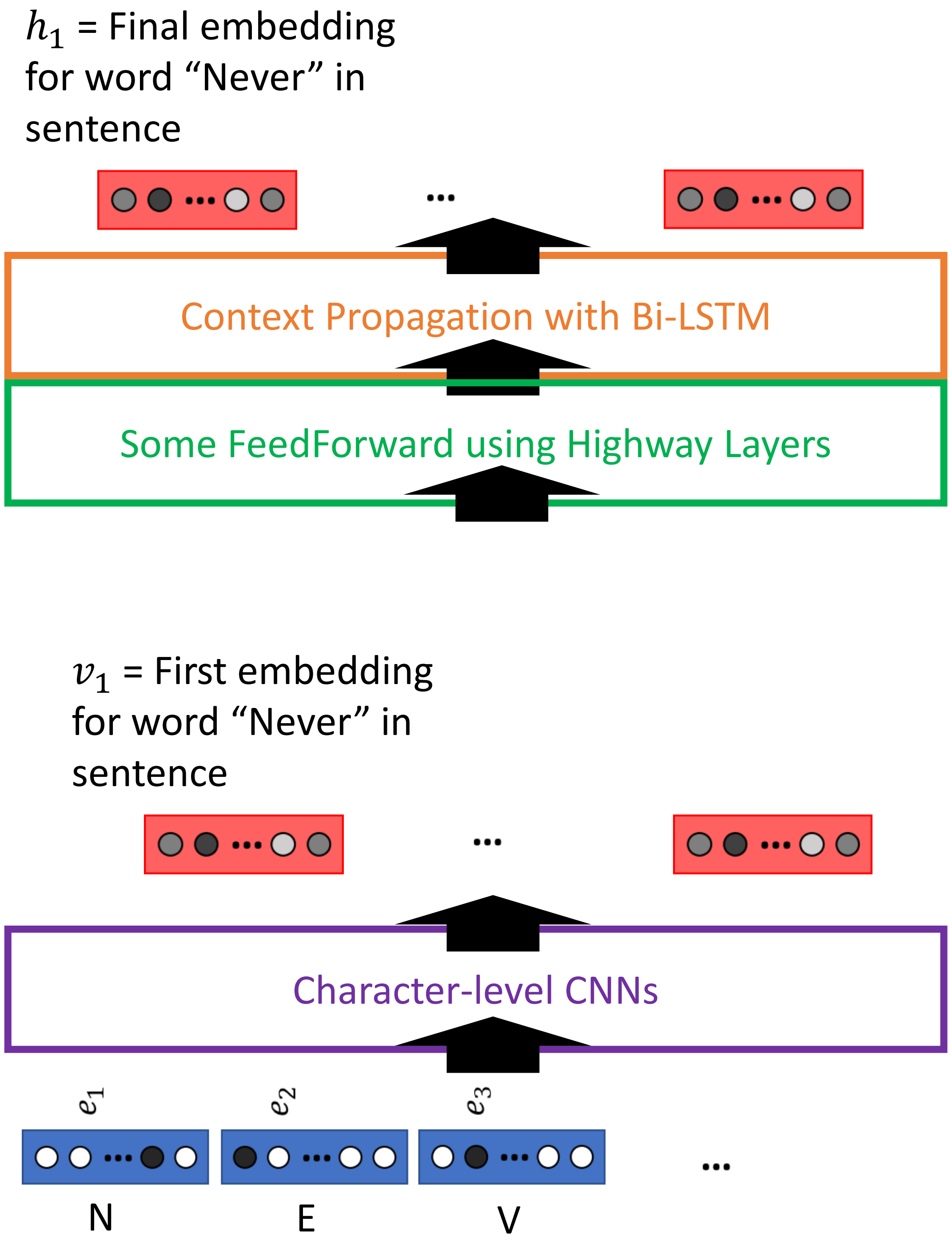
5. So far, we could connect both models and they technically would work just fine…



ELMo – Key takeaways

5. So far, we could connect both models and they technically would work just fine…

   In practice, however, we like to add a few extra Linear layers in the middle to allow for a smoother transition.

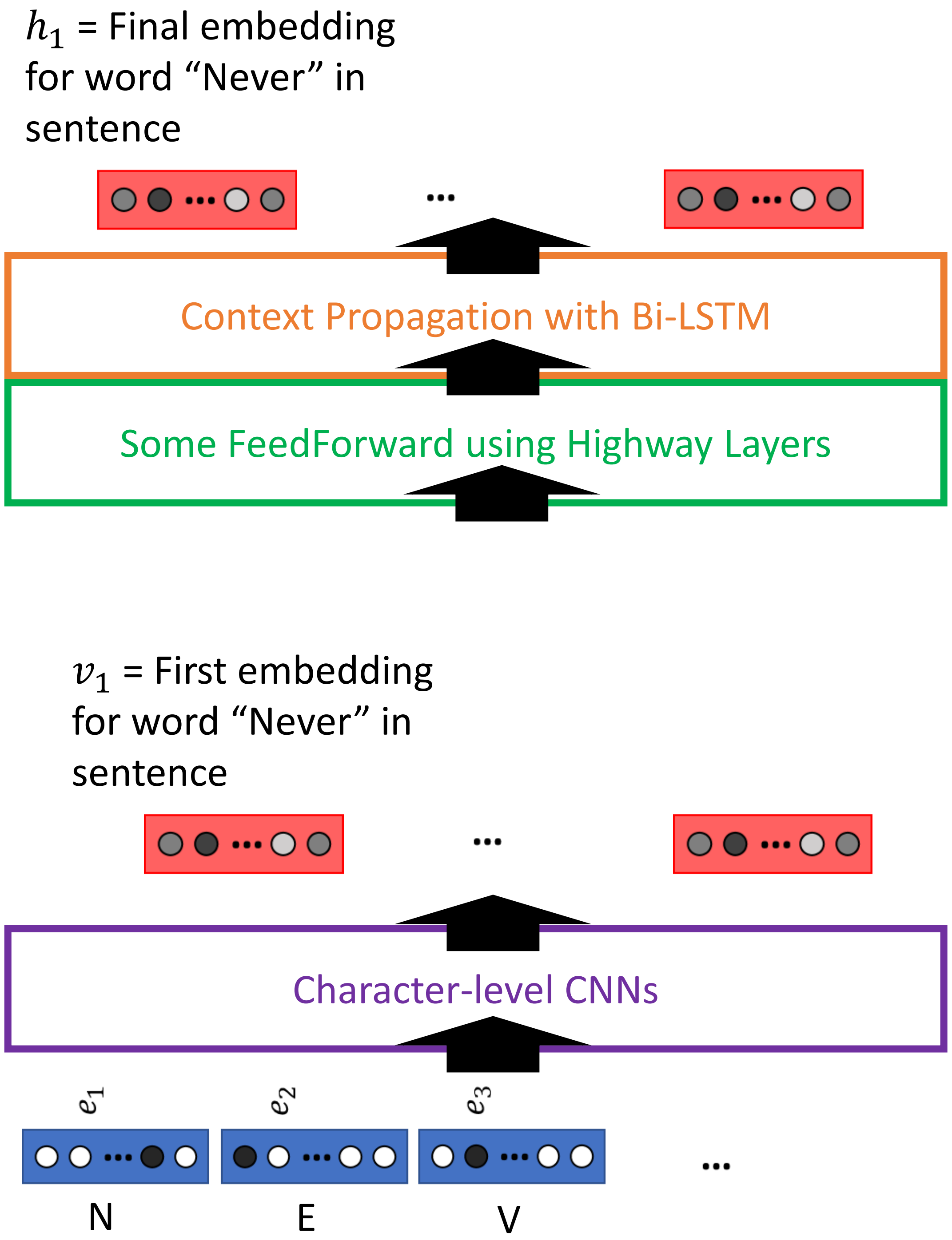


ELMo – Key takeaways

5. **In the middle:**
   Add some **Feedforward layers**

   This allows for a **smoother transition from the character-level CNNs output to the Bi-LSTM input**.

   Can be simply done with a **succession** of **Linear layers**.
   Or a variation of Linear layers, called a **highway network**.



A note on the Highway Layer

Definition (Highway Layer):

The **Highway Layer** is another **variant of the fully connected (or Linear layer)**, with an additional gated residual connection.

You only need to know that it produces a standard mapping with a non-linear activation function $y = g(W_1 x + b_1)$ like any standard fully connected layer.

However, its full propagation formula will include a gated residual, as shown below. It consists of a transform gate and a carry gate.

$$z = t \odot g(W_1 x + b_1) + (1 - \mathrm{t}) \odot x$$

With

$$t = \sigma(W_2 x + b_2)$$

Similar to residual connections, this layer helps with information flow.

A note on the Highway Layer

```
# Blocks to be used for the highway connection
self.highway = nn.Linear(128, 128)
self.transform = nn.Linear(128, 128)
```

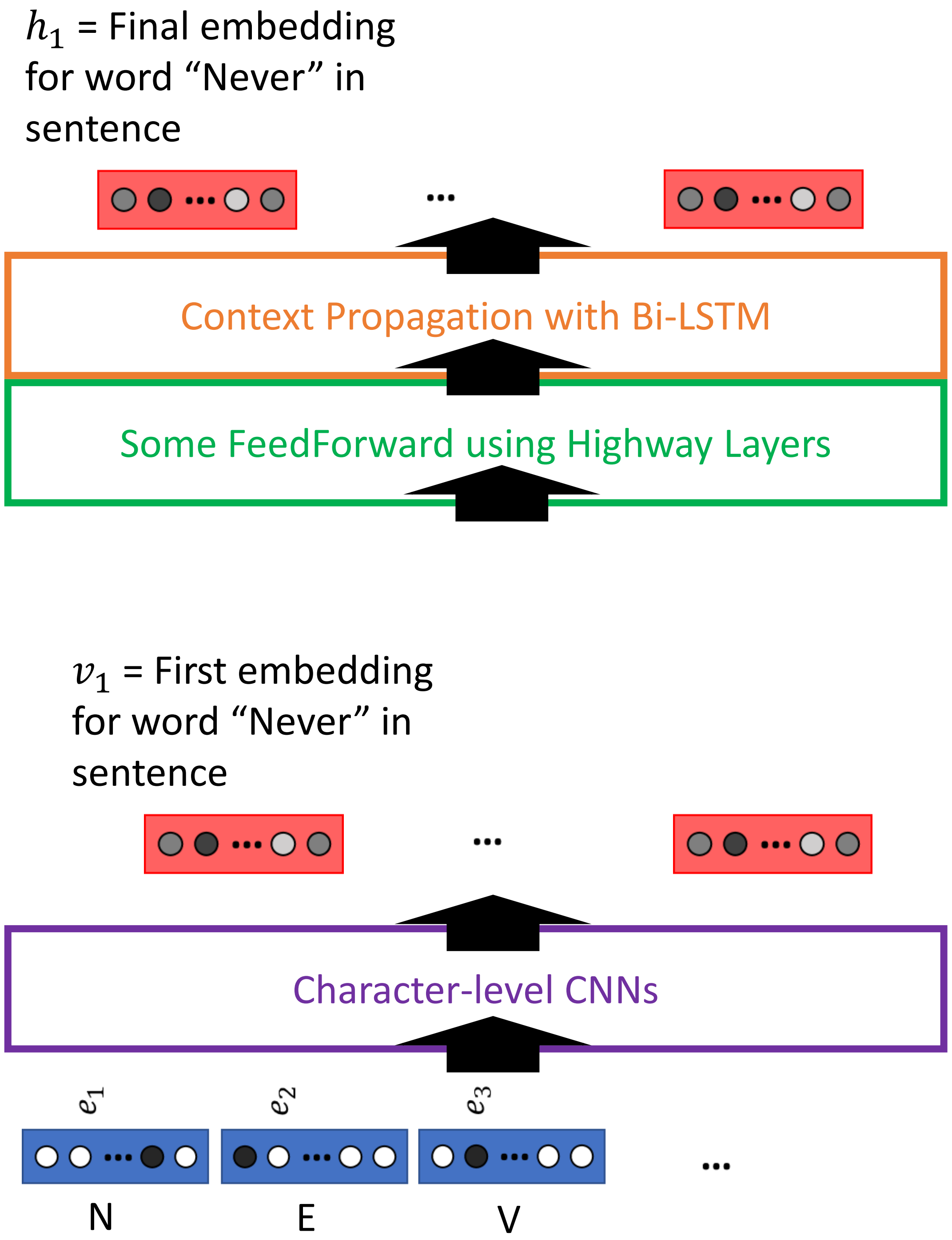
```
# 2 Some Highway layers
h = self.highway(x)
t_gate = torch.sigmoid(self.transform(x))
c_gate = 1 - t_gate
x_ = h * t_gate + x * c_gate
```

ELMo

Final Recap

1. Input of words, into a character-level network (inputs will be a list of characters!)
2. To get an embedding of a word, input the whole sentence for context, then take only the vector which corresponds to that word.
3. On top of character-level network, use multiple layers of RNNs on word-level.
4. Some highway layers for transition in the middle.
5. **Train the whole network for the task of predicting the next word in sentence, almost as in CBoW.**
6. **After training, extract embedding layer for feature representation**

```python
class BiLangModel(nn.Module):

    def __init__(self, char_cnn, bi_lstm):

        super(BiLangModel, self).__init__()
        # Blocks to be used for the highway connection
        self.highway = nn.Linear(128, 128)
        self.transform = nn.Linear(128, 128)

        # Character level CNN model
        self.char_cnn = char_cnn

        # Bi-LSTM model
        self.bi_lstm = bi_lstm

    def forward(self, x):

        # 1. Character-level convolution
        x = self.char_cnn(x).permute(2, 0, 1)

        # 2 Some Highway layers
        h = self.highway(x)
        t_gate = torch.sigmoid(self.transform(x))
        c_gate = 1 - t_gate
        x_ = h * t_gate + x * c_gate

        # 3. Bi-LSTM
        x1, x2 = self.bi_lstm(x_)

        # Feel free to play around and have a look
        # at the x, x1 and x2 vectors!
        return x, x1, x2
```



ELMo

Key takeaways on ELMo

- ELMo is a pretty recent word embedding (2018!), which has, for a long time, held the **state-of-the-art performance for word embeddings**.
- These days (2021-2022), the state of the art is the **BERT embedding.** (to be discussed on the next lecture, after the lesson on transformers!)
- It is **able to operate on out-of-vocabulary words**, as it **takes characters as inputs**, and can process context to address Problem #1 (as opposed to CBoW/SG & FastText)
- **Its main downside is the computational cost** of the architecture. Preferable to use **FastText** if you need speed!

Conclusion (W9S2)

We have seen a few approaches to embeddings.

- Train an AI to figure out embeddings? Basic approaches.
  - (CBoW)
  - SkipGram
- Out of dictionary entries?
  - Use FastText!
- Need to incorporate context words for embedding?
  - Use ELMo!

A few more problems are still open at the moment for these word embeddings.

- How de we evaluate embeddings and decide which one is best?
- Can these embedding be biased? Yes, unfortunately.
- Can we help the network in computing context? BERT, later.

Learn more about these topics

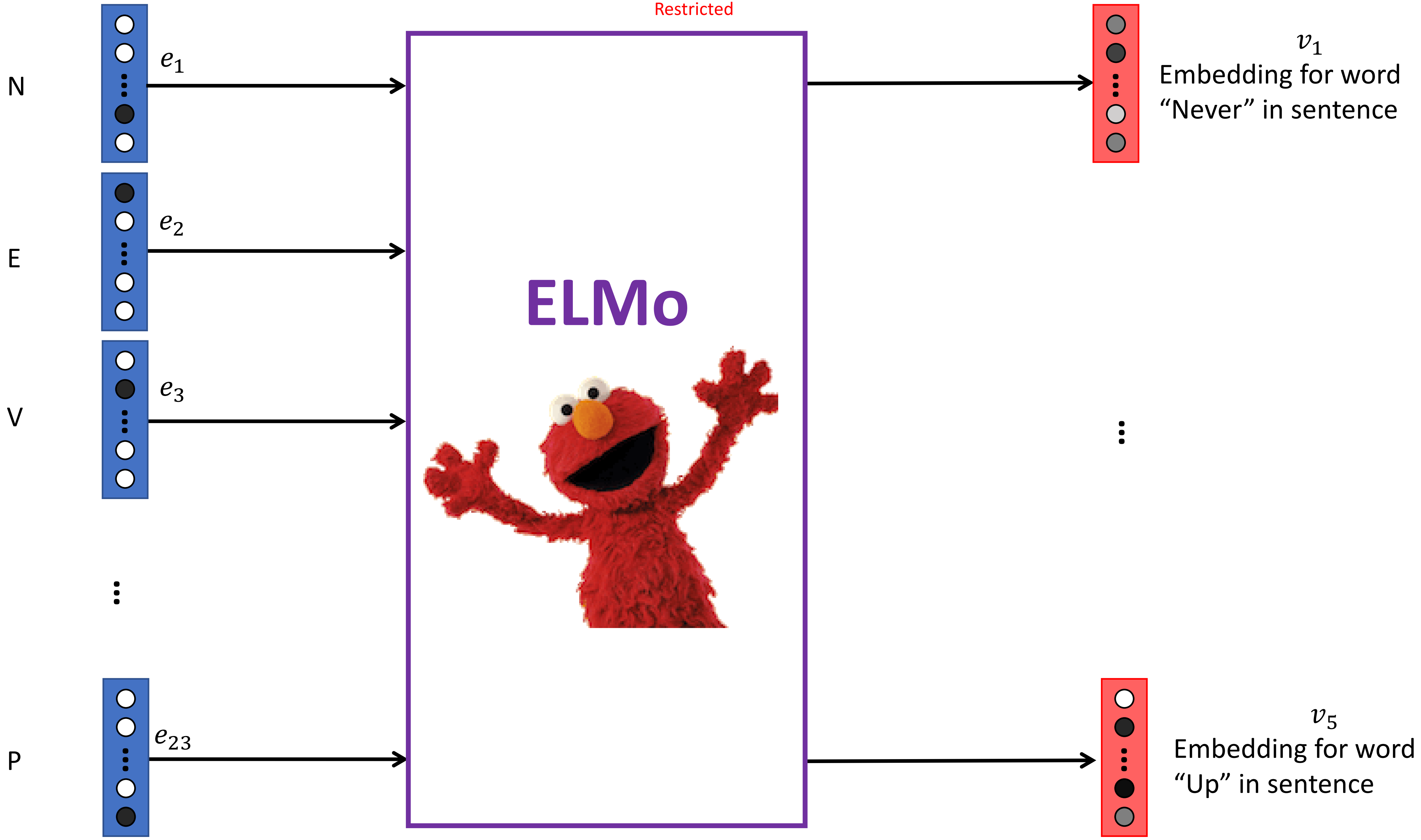
Out of class, for those of you who are curious

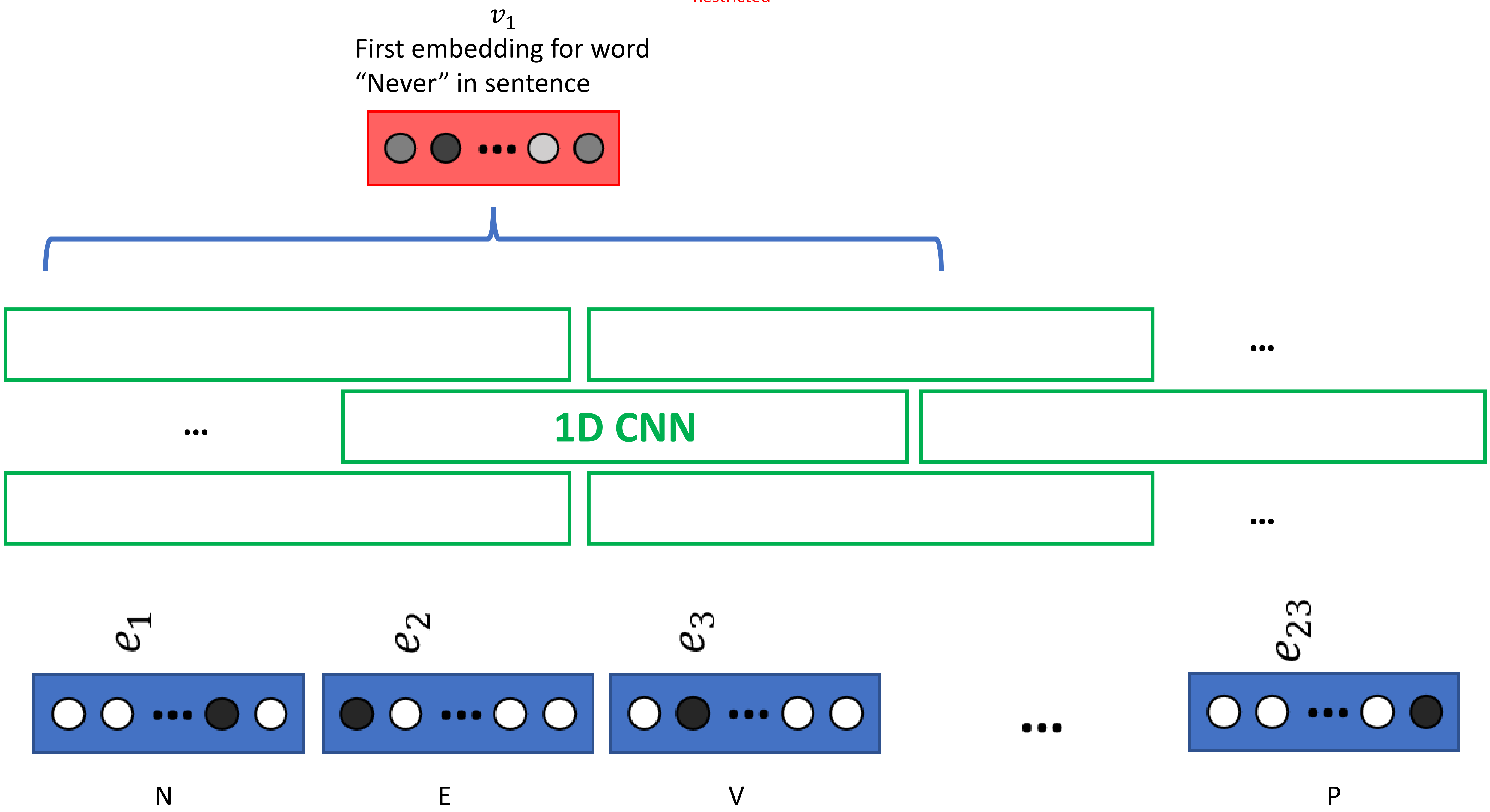
- [Mikolov2013] **Mikolov** et al., "Efficient Estimation of Word Representations in Vector Space", 2013. https://arxiv.org/abs/1301.3781
- [Mikilov2014] **Mikolov** et al., "Distributed Representations of Words and Phrases and their Compositionality", 2014. https://arxiv.org/abs/1310.4546
- [GloVe2014] Pennington et al., "GloVe: Global Vectors for Word Representation", 2014. https://nlp.stanford.edu/projects/glove/

Learn more about these topics

Out of class, for those of you who are curious

- [Mikolov2017] **Mikolov** et al., “Advances in Pre-Training Distributed Word Representations”, 2017. https://arxiv.org/abs/1301.3781
- [Bojanovski2017] Bojanowski et al., “Enriching Word Vectors with Subword Information”, 2017. https://arxiv.org/abs/1607.04606
- [Peters2018] **Peters** et al., “Deep contextualized word representations”, 2018. https://arxiv.org/abs/1802.05365


N
E
V
P
e_1
e_2
e_3
e_{23}
ELMo
v_1
Embedding for word
"Never" in sentence
v_5
Embedding for word
"Up" in sentence


v_1
First embedding for word
“Never” in sentence
...
1D CNN
...
...
e_1
e_2
e_3
e_{23}
...
N
E
V
P