LLMs from Dummies

Topics

1. Language translation
    1. Simplest program: Literal translation
    2. Dealing with missing tokens
2. Literal translation using "ML”
    1. Encoding: One hot
    2. Using matrices
    3. A 'dictionary' using matrices
    4. Decoding: Cosine similarity
3. Attention
    1. Similar tokens: Softmax
    2. Matrix Query: Q
    3. Scaled dot-product attention
4. Attention Head
    1. Weight matrices
    2. Connecting matrices
5. Revisiting tokens & encoding
    1. Tokenizing: BPE
    2. Encoding: Embeddings
6. Transformer Block
    1. Multi-headed attention
    2. Non-linear (feed forward) layer
    3. Stack blocks
    4. Masked attention
7. Transformer
    1. Stacking deep networks
    2. Normalization layers
    3. Skip connections
    4. Dropout

8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models
9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

Basic concepts: Methodology

We will start by creating a trivial program.

Then we’ll transform it into a Neural Network model.

We’ll generalize concepts one by one…

…until we reach an LLM

Topics

1. Language translation
    1. Simplest program: Literal translation
    2. Dealing with missing tokens
2. Literal translation using "ML”
    1. Encoding: One hot
    2. Using matrices
    3. A 'dictionary' using matrices
    4. Decoding: Cosine similarity
3. Attention
    1. Similar tokens: Softmax
    2. Matrix Query: Q
    3. Scaled dot-product attention
4. Attention Head
    1. Weight matrices
    2. Connecting matrices
5. Revisiting tokens & encoding
    1. Tokenizing: BPE
    2. Encoding: Embeddings
6. Transformer Block
    1. Multi-headed attention
    2. Non-linear (feed forward) layer
    3. Stack blocks
    4. Masked attention
7. Transformer
    1. Stacking deep networks
    2. Normalization layers
    3. Skip connections
    4. Dropout

8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models
9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

Topics

1. Language translation
    1. Simplest program: Literal translation
    2. Dealing with missing tokens
2. Literal translation using "ML”
    1. Encoding: One hot
    2. Using matrices
    3. A 'dictionary' using matrices
    4. Decoding: Cosine similarity
3. Attention
    1. Similar tokens: Softmax
    2. Matrix Query: Q
    3. Scaled dot-product attention
4. Attention Head
    1. Weight matrices
    2. Connecting matrices
5. Revisiting tokens & encoding
    1. Tokenizing: BPE
    2. Encoding: Embeddings
6. Transformer Block
    1. Multi-headed attention
    2. Non-linear (feed forward) layer
    3. Stack blocks
    4. Masked attention
7. Transformer
    1. Stacking deep networks
    2. Normalization layers
    3. Skip connections
    4. Dropout

8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models
9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

Language translation

Let’s create the simplest program to translate

“le chat est sous la table” => “the cat is under the table”

Simplest: We’ll use literal (i.e. word by word) translation

We need a dictionary

```
dictionary[ key ] = value
```

```
dictionary = {
    'le': 'the'
    , 'chat': 'cat'
    , 'est': 'is'
    , 'sous': 'under'
    , 'la': 'the'
    , 'table': 'table'
}

dictionary['chat'] = 'cat'
```

1. Split the sentence into words ("tokens")

2. Translate each word

```python
def tokenize(text):
    ''' Split sentences into tokens (words) '''
    return text.split()

def translate(sentence):
    ''' Translate a sentence '''
    out = ''
    for token in tokenize(sentence):
        out += dictionary[token] + ' '
    return out
```

It works

```
translate("le chat est sous la table")
✓ 0.0s
'the cat is under the table '
```

What if a ‘token’ is not in the dictionary?

E.g.: we have a similar word

```
translate("tables")
✓ 0.0s

Traceback
in <module>:1

❱ 1 translate("tables")
  2

in translate:5

  2 │   ''' Translate a sentence '''
  3 │   out = ''
  4 │   for word in tokenize(sentence):
❱ 5 │   │   out += dictionary[word] + ' '
  6 │   return out
  7

KeyError: 'tables'
```

What if a ‘token’ is not in the dictionary?

Let’s improve this by relaxing key matching

- We have a “query” token
- We find the closest “key” in the dictionary

```python
from Levenshtein import distance

def find_closest_key(query):
    ''' Find closest key in dictionary '''
    closest_key, min_dist = None, float('inf')
    for key in dictionary.keys():
        dist = distance(query, key)
        if dist < min_dist:
            min_dist, closest_key = dist, key
    return closest_key


def translate(sentence):
    ''' Translate a sentence '''
    out = ''
    for query in tokenize(sentence):
        key = find_closest_key(query)
        out += dictionary[key] + ' '
    return out
```

Now we can “approximately translate” words that are NOT in our dictionary

```
translate("tables")
✓ 0.0s

'table '
```

Topics

1. Language translation
   1. Simplest program: Literal translation
   2. Dealing with missing tokens
2. Literal translation using "ML”
   1. Encoding: One hot
   2. Using matrices
   3. A 'dictionary' using matrices
   4. Decoding: Cosine similarity
3. Attention
   1. Similar tokens: Softmax
   2. Matrix Query: Q
   3. Scaled dot-product attention
4. Attention Head
   1. Weight matrices
   2. Connecting matrices
5. Revisiting tokens & encoding
   1. Tokenizing: BPE
   2. Encoding: Embeddings
6. Transformer Block
   1. Multi-headed attention
   2. Non-linear (feed forward) layer
   3. Stack blocks
   4. Masked attention
7. Transformer
   1. Stacking deep networks
   2. Normalization layers
   3. Skip connections
   4. Dropout

8. Pre-training, Training, Fine-tunning, Adapting, Instruct
   8. Pre-training numbers (GPU hours, params, etc.)
   9. LORA / PERF: Basic concepts
   10. "Instruct" models
9. Prompts all the way down
   8. "Chat": Just isolated requests with "memory" of conversations
   9. "Context": Just add a sentence to the prompt
   10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

Now let’s use ML

So far we did not use any ML. Let's convert our program to a Neural network.

We cannot use "string" tokens in neural networks, we use vectors instead.

What is the simplest vector representation for our vocabularies?

Let's define "vocabulary":

```python
# Vocabulary: All the words in the dictionary
vocabulary_in = sorted(list(set(dictionary.keys())))
print(f"Vocabulary input ({len(vocabulary_in)}): {vocabulary_in}")

vocabulary_out = sorted(list(set(dictionary.values())))
print(f"Vocabulary output ({len(vocabulary_out)}): {vocabulary_out}")
```

✓ 0.0s

```
Vocabulary input (6): ['chat', 'est', 'la', 'le', 'sous', 'table']
Vocabulary output (5): ['cat', 'is', 'table', 'the', 'under']
```

We can use ”one hot” encoding

Each token is a vector filled with zeros, except in one position where we set it to one.

$$E_{chat} = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$$
$$E_{est} = \begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0\end{bmatrix}$$
$$E_{la} = \begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0\end{bmatrix}$$
$$E_{le} = \begin{bmatrix}0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}$$
$$E_{sous} = \begin{bmatrix}0 & 0 & 0 & 0 & 1 & 0\end{bmatrix}$$
$$E_{table} = \begin{bmatrix}0 & 0 & 0 & 0 & 0 & 1\end{bmatrix}$$

Note: Vector dimension = Vocabulary size

In our example our vocabulary is 6 words, so vector dimension is 6

If we have 10,000 words in our vocabulary, the vector dimension is 10,000

Code for one hot encoding

```
# Convert to one hot encoding
def convert_to_one_hot(vocabulary):
    vocabulary_size = len(vocabulary)
    one_hot = dict()
    LEN = len(vocabulary)
    for i, key in enumerate(vocabulary):
        one_hot_vector = np.zeros(LEN)
        one_hot_vector[i] = 1
        one_hot[key] = one_hot_vector
        print(f"{key}\t: {one_hot[key]}")
```

Code for one hot encoding

```python
# Convert to one hot encoding
def convert_to_one_hot(vocabulary):
    vocabulary_size = len(vocabulary)
    one_hot = dict()
    LEN = len(vocabulary)
    for i, key in enumerate(vocabulary):
        one_hot_vector = np.zeros(LEN)
        one_hot_vector[i] = 1
        one_hot[key] = one_hot_vector
        print(f"{key}\t: {one_hot[key]}")
```

```python
one_hot_in = convert_to_one_hot(vocabulary_in)
```

✓ 0.0s

```
chat    : [1. 0. 0. 0. 0. 0.]
est     : [0. 1. 0. 0. 0. 0.]
la      : [0. 0. 1. 0. 0. 0.]
le      : [0. 0. 0. 1. 0. 0.]
sous    : [0. 0. 0. 0. 1. 0.]
table   : [0. 0. 0. 0. 0. 1.]
```

```python
one_hot_out = convert_to_one_hot(vocabulary_out)
```

✓ 0.0s

```
cat     : [1. 0. 0. 0. 0.]
is      : [0. 1. 0. 0. 0.]
table   : [0. 0. 1. 0. 0.]
the     : [0. 0. 0. 1. 0.]
under   : [0. 0. 0. 0. 1.]
```

Now we need a “dictionary” structure.

Unfortunately, there is no “dictionary” structure in Neural Network…

…but we can create something like that using matrix multiplications.

Let’s create a ”keys matrix” (K), and a “values matrix” (V)

```
K = np.array( [one_hot_in[k] for k in dictionary.keys()] )
K
```

✓ 0.0s

```
array([[0., 0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0.],
       [0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1.]])
```

```
V = np.array( [one_hot_out[k] for k in dictionary.values()] )
V
```

✓ 0.0s

```
array([[0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       [0., 0., 0., 1., 0.],
       [0., 0., 1., 0., 0.]])
```

These matrices are the keys and values of the original "dictionary", but they are just one hot encoded

$$K = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

These matrices are the keys and values of the original “dictionary”, but they are just one hot encoded

$$K = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

```
dictionary = {
    'le': 'the'
    , 'chat': 'cat'
    , 'est': 'is'
    , 'sous': 'under'
    , 'la': 'the'
    , 'table': 'table'
}

dictionary['chat'] = 'cat'
```

Example: In this "dictionary, how do we get the "value" for key *"sous"*(i.e. `dictionary['sous'] = 'under'`)

$$K = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

```
dictionary = {
    'le': 'the'
    , 'chat': 'cat'
    , 'est': 'is'
    , 'sous': 'under'
    , 'la': 'the'
    , 'table': 'table'
}

dictionary['chat'] = 'cat'
```

Example: In this “dictionary, how do we get the “value” for key *“sous”*(i.e. `dictionary[‘sous’] = ‘under’`)

$$q.K^T.V$$

Example: In this “dictionary, how do we get the “value” for key *“sous”*(i.e. `dictionary[‘sous’] = ‘under’`)

$$q = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \; ; \qquad K = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \; ; \qquad V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$q.K^T.V = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} . \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

```python
dictionary = {
    'le': 'the'
    , 'chat': 'cat'
    , 'est': 'is'
    , 'sous': 'under'
    , 'la': 'the'
    , 'table': 'table'
}

dictionary['chat'] = 'cat'
```

Example: In this "dictionary, how do we get the "value" for key *"sous"*(i.e. `dictionary['sous'] = 'under'`)

$$q = \boxed{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}} \; ; \quad K = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \; ; \quad V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

1. Query is "sous" (one hot)

$$q.K^T.V = \boxed{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}} . \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

```
dictionary = {
    'le': 'the'
  , 'chat': 'cat'
  , 'est': 'is'
  , 'sous': 'under'
  , 'la': 'the'
  , 'table': 'table'
}

dictionary['chat'] = 'cat'
```

Example: In this “dictionary, how do we get the “value” for key *“sous”*(i.e. `dictionary[‘sous’] = ‘under’`)

$$q = \boxed{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}} \quad ; \qquad K = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad ; \qquad V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

1. Query is “sous” (one hot)
2. Select the key from K that matches the query

$$\boxed{q.K^T} V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

```python
dictionary = {
    'le': 'the'
    , 'chat': 'cat'
    , 'est': 'is'
    , 'sous': 'under'
    , 'la': 'the'
    , 'table': 'table'
}

dictionary['chat'] = 'cat'
```

Example: In this “dictionary, how do we get the “value” for key *“sous”*(i.e. `dictionary[‘sous’] = ‘under’`)

$$q = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \; ; \qquad K = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \; ; \qquad V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

1. Query is “sous” (one hot)
2. Select the key from K that matches the query
3. Get the value from V

$$q.K^T.V = \begin{bmatrix} 0 & 0 & 0 & \boxed{1} & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

```
dictionary = {
    'le': 'the'
    , 'chat': 'cat'
    , 'est': 'is'
    , 'sous': 'under'
    , 'la': 'the'
    , 'table': 'table'
}
```

```
dictionary['chat'] = 'cat'
```

Example: In this “dictionary, how do we get the “value” for key *“sous”*(i.e. `dictionary[‘sous’] = ‘under’`)

$$q = \boxed{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}} \; ; \quad K = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \; ; \quad V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

1. Query is “sous” (one hot)
2. Select the key from K that matches the query
3. Get the value from V

$$q.K^T.V = \boxed{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \end{bmatrix}}$$

```
dictionary = {
    'le': 'the'
  , 'chat': 'cat'
  , 'est': 'is'
  , 'sous': 'under'
  , 'la': 'the'
  , 'table': 'table'
}

dictionary['chat'] = 'cat'
```

Same example using code

```
q = one_hot_in['sous']
print("Query token     : ", q)
print("Select key (K)  : ", q @ K.T)
print("Select value (V): ", q @ K.T @ V)
```

✓ 0.0s

```
Query token     :  [0. 0. 0. 0. 1. 0.]
Select key (K)  :  [0. 0. 0. 1. 0. 0.]
Select value (V):  [0. 0. 0. 0. 1.]
```

We need a way to ”decode” back from one hot encoding to words

Find the “closest” vector matching the output.

```python
def one_hot_decode(one_hot, vector):
    """ Decode "one hot". Find the best matching 'token' """
    best_key, best_cosine_sim = None, 0
    for k, v in one_hot.items():
        cosine_sim = np.dot(vector, v)
        if cosine_sim > best_cosine_sim:
            best_cosine_sim = cosine_sim
            best_key = k
    return best_key
```

Note: We use an inner product, for vectors of norm 1 this is a “cosine similarity”.

Now we have can write a “translate” function:

```python
def translate(sentence):
    sentence_out = ''
    for token_in in tokenize(sentence):
        q = one_hot_in[token_in]
        out = q @ K.T @ V
        token_out = one_hot_decode(one_hot_out, out)
        sentence_out += token_out + ' '
    return sentence_out
```

```python
translate("le chat est sous la table")
```

✓ 0.0s

```
'the cat is under the table '
```

$$q.K^T.V$$

This works like a “dictionary”

$$q.K^T.V$$

This works like a “dictionary”

With a couple more tweaks, this becomes an “Attention” mechanism, which is the core structure in LLMs

Topics

1. Language translation
    1. Simplest program: Literal translation
    2. Dealing with missing tokens
2. Literal translation using "ML”
    1. Encoding: One hot
    2. Using matrices
    3. A 'dictionary' using matrices
    4. Decoding: Cosine similarity
3. Attention
    1. Similar tokens: Softmax
    2. Matrix Query: Q
    3. Scaled dot-product attention
4. Attention Head
    1. Weight matrices
    2. Connecting matrices
5. Revisiting tokens & encoding
    1. Tokenizing: BPE
    2. Encoding: Embeddings
6. Transformer Block
    1. Multi-headed attention
    2. Non-linear (feed forward) layer
    3. Stack blocks
    4. Masked attention
7. Transformer
    1. Stacking deep networks
    2. Normalization layers
    3. Skip connections
    4. Dropout

8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models
9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

… a few tweaks towards Attention

Let's relax the "one hot" encoding assumption

What if the "query" does not exactly match a key?

We could encode similar tokens using similar vectors

For example, we could encode the token "table" as

$$E_{table} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

…and a similar token "tables"

$$E_{tables} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0.95 \end{bmatrix}$$

Let’s relax the “one hot” encoding assumption

What if the ”query” does not exactly match a key?

We could encode similar tokens using similar vectors

For example, we could encode the token “table” as

$$E_{table} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

…and a similar token “tables”

$$E_{tables} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0.95 \end{bmatrix}$$

Our equation still works, but we need an adjustment:

$$q.K^T.V$$

Let's relax the "one hot" encoding assumption

What if the "query" does not exactly match a key?

We could encode similar tokens using similar vectors

For example, we could encode the token "table" as

$$E_{table} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

…and a similar token "tables"

$$E_{tables} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0.95 \end{bmatrix}$$

Our equation still works, but we need an adjustment:

$$\underbrace{q.K^T}.V$$

This multiplication acts like a "weight" selecting the values from V

Let’s relax the “one hot” encoding assumption

What if the ”query” does not exactly match a key?

We could encode similar tokens using similar vectors

For example, we could encode the token “table” as

$$E_{table} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

…and a similar token “tables”

$$E_{tables} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0.95 \end{bmatrix}$$

Our equation still works, but we need an adjustment:

$$\underbrace{q.K^T}.V$$

This multiplication acts like a “weight” selecting the values from V

So, it should return a vector of non-negative numbers that add to 1

Problem: We want to convert a vector $q.K^T$ into “weights”, i.e. non-negative numbers that add up to 1.

The weight should be “higher” (closest to 1) for the largest number in $q.K^T$

Problem: We want to convert a vector $q.K^T$ into "weights", i.e. non-negative numbers that add up to 1.

The weight should be "higher" (closest to 1) for the largest number in $q.K^T$

Solution: softmax function

$$\sigma(\mathbf{z})_i = \frac{e^{\beta z_i}}{\sum_{j=1}^{K} e^{\beta z_j}}$$

Problem: We want to convert a vector $q.K^T$ into "weights", i.e. non-negative numbers that add up to 1.

The weight should be "higher" (closest to 1) for the largest number in $q.K^T$

Solution: softmax function

$$\sigma(\mathbf{z})_i = \frac{e^{\beta z_i}}{\sum_{j=1}^{K} e^{\beta z_j}}$$

Example: softmax([0, 10])

$$\sigma(0,\, 10) := \sigma_1(0,\, 10) = \left(1/\left(1+e^{10}\right),\, e^{10}/\left(1+e^{10}\right)\right) \approx (0.00005,\, 0.99995)$$

Our new equation is:

$$softmax(q.K^T).V$$

Since softmax() tends to saturate quickly when we use large dimensional vectors, people often adjust using:

$$softmax\left(\frac{q.K^T}{\sqrt{d}}\right).V$$

where ‘d’ is the dimension of the query vector, i.e. d = dim(q)

Improvement: Calculate all the queries in parallel

Previously we had a single query vector ‘q’ for each input token, but we can simply create a matrix ‘Q’ with all the input tokens

$$Q = \begin{array}{cl} \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} & \begin{array}{l} \texttt{--- le} \\ \texttt{--- chat} \\ \texttt{--- est} \\ \texttt{--- sous} \\ \texttt{--- la} \\ \texttt{--- table} \end{array} \end{array}$$

$$softmax\left(\frac{Q.K^T}{\sqrt{d}}\right).V$$

Now our code is simpler, we don't need for loops to create the output and has the additional advantage, it can get calculated in parallel in a GPU:

```
def translate(sentence):
    Q = torch.stack([one_hot_in[token] for token in tokenize(sentence)])
    out = torch.softmax(Q @ K.T, 0) @ V
    return ' '.join([decode_one_hot(one_hot_out, o) for o in out])

translate("le chat est sous la table")
```

✓ 0.0s

```
'the cat is under the table'
```

This is called ”Attention” (or more specifically “Scaled dot-product attention”)

$$Attention(Q, K, V) = softmax\left(\frac{Q.K^T}{\sqrt{d}}\right)V$$

$$Attention(Q, K, V) = softmax\left(\frac{Q.K^T}{\sqrt{d}}\right)V$$

Attention is more than just a “dictionary”, we’ll get to that in a bit...

Published as a conference paper at ICLR 2015

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

ABSTRACT

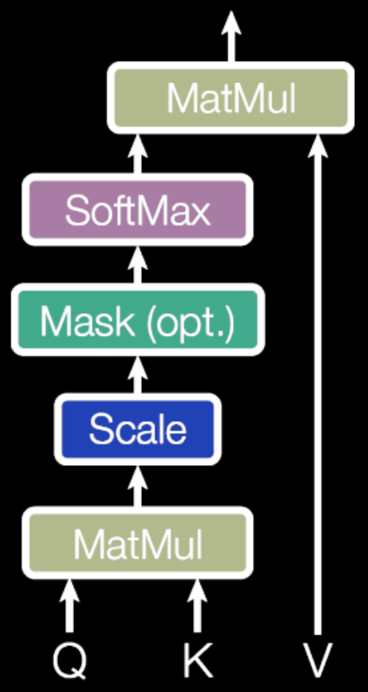
Neural machine translation is a recently proposed approach to machine translation. Unlike the traditional statistical machine translation, the neural machine translation aims at building a single neural network that can be jointly tuned to maximize the translation performance. The models proposed recently for neural machine translation often belong to a family of encoder–decoders and encode a source sentence into a fixed-length vector from which a decoder generates a translation. In this paper, we conjecture that the use of a fixed-length vector is a bottleneck in improving the performance of this basic encoder–decoder architecture, and propose to extend this by allowing a model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word, without having to form these parts as a hard segment explicitly. With this new approach, we achieve a translation performance comparable to the existing state-of-the-art phrase-based system on the task of English-to-French translation. Furthermore, qualitative analysis reveals that the (soft-)alignments found by the model agree well with our intuition.

Topics

1. Language translation
   1. Simplest program: Literal translation
   2. Dealing with missing tokens
2. Literal translation using "ML"
   1. Encoding: One hot
   2. Using matrices
   3. A 'dictionary' using matrices
   4. Decoding: Cosine similarity
3. Attention
   1. Similar tokens: Softmax
   2. Matrix Query: Q
   3. Scaled dot-product attention
4. Attention Head
   1. Weight matrices
   2. Connecting matrices
5. Revisiting tokens & encoding
   1. Tokenizing: BPE
   2. Encoding: Embeddings
6. Transformer Block
   1. Multi-headed attention
   2. Non-linear (feed forward) layer
   3. Stack blocks
   4. Masked attention
7. Transformer
   1. Stacking deep networks
   2. Normalization layers
   3. Skip connections
   4. Dropout
8. Pre-training, Training, Fine-tunning, Adapting, Instruct
   8. Pre-training numbers (GPU hours, params, etc.)
   9. LORA / PERF: Basic concepts
   10. "Instruct" models
9. Prompts all the way down
   8. "Chat": Just isolated requests with "memory" of conversations
   9. "Context": Just add a sentence to the prompt
   10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
   8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
   9. Vector databases
   10. Huggingface
11. Scaling inference & training
   8. Single GPU: Quantization, fp16, etc.
   9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

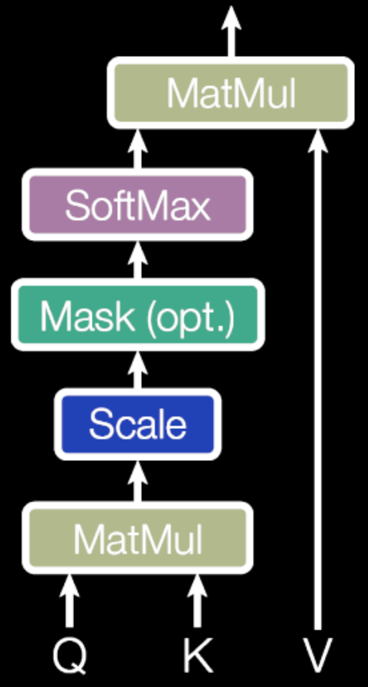
Attention Head

$$Attention(Q, K, V) = softmax\left(\frac{Q.K^T}{\sqrt{d}}\right)V$$



How can we make the Attention mechanism more flexible?

$$Attention(Q, K, V) = softmax\left(\frac{Q.K^T}{\sqrt{d}}\right)V$$

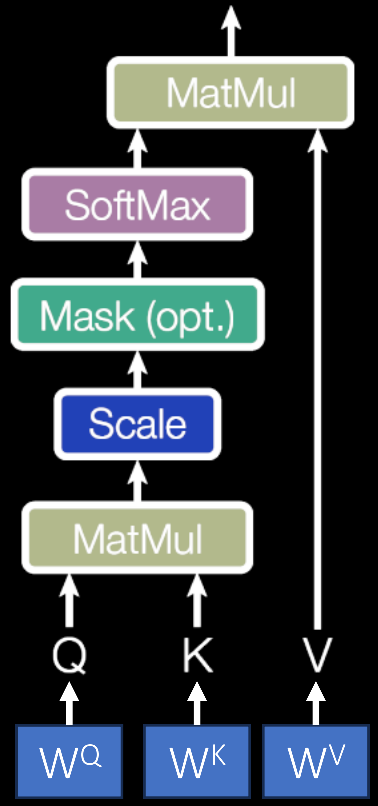


How can we make the Attention mechanism more flexible?

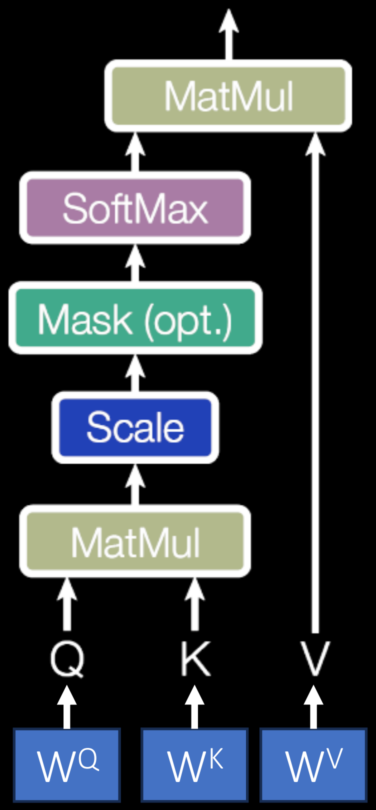
$$Attention(Q, K, V) = softmax\left(\frac{Q.K^T}{\sqrt{d}}\right)V$$

We can add a weight matrices:

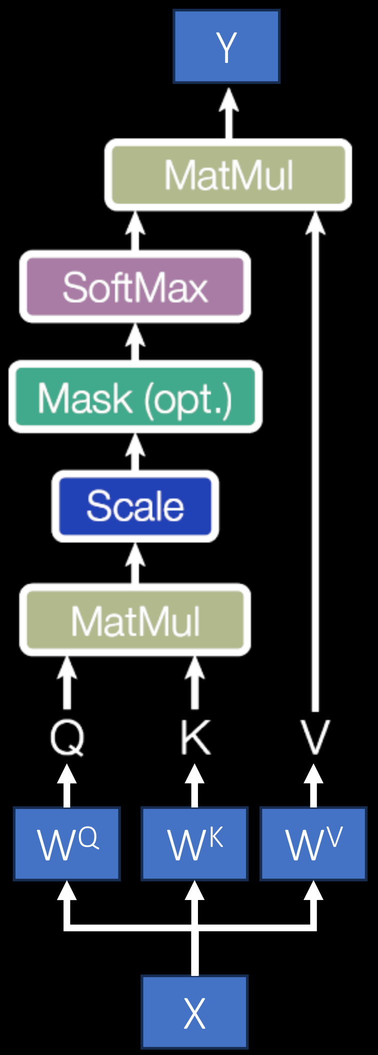
$$Attention(Q, K, V) => Attention(Q.W^Q, K.W^K, V.W^V$$



How do we connect the input?



How do we connect the input?



Coding an “Attention Head”

```python
class Head(nn.Module):
    """ Self attention head """

    def __init__(self, params):
        super().__init__()
        self.key = nn.Linear(params.n_embd, params.head_size, bias=False)
        self.query = nn.Linear(params.n_embd, params.head_size, bias=False)
        self.value = nn.Linear(params.n_embd, params.head_size, bias=False)

    def forward(self, x):
        k = self.key(x)
        q = self.query(x)
        v = self.value(x)
        # Attention score
        w = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5   # Query * Keys / normalization
        w = F.softmax(w, dim=-1)  # Do a softmax across the last dimesion
        # Add weighted values
        out = w @ v
        return out
```

Topics

1. Language translation
    1. Simplest program: Literal translation
    2. Dealing with missing tokens
2. Literal translation using "ML”
    1. Encoding: One hot
    2. Using matrices
    3. A 'dictionary' using matrices
    4. Decoding: Cosine similarity
3. Attention
    1. Similar tokens: Softmax
    2. Matrix Query: Q
    3. Scaled dot-product attention
4. Attention Head
    1. Weight matrices
    2. Connecting matrices
5. Revisiting tokens & encoding
    1. Tokenizing: BPE
    2. Encoding: Embeddings
6. Transformer Block
    1. Multi-headed attention
    2. Non-linear (feed forward) layer
    3. Stack blocks
    4. Masked attention
7. Transformer
    1. Stacking deep networks
    2. Normalization layers
    3. Skip connections
    4. Dropout

8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models
9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

Revisiting token encoding

Tokenizing

- Previously we “tokenized” by splitting a sentence into words

```python
def tokenize(text):
    ''' Split sentences into tokens (words) '''
    return text.split()
```

- We can do better: There are many tokenization methods
- A popular one is “Byte Pair Encoding” (BPE)
- BPE was originally proposed as a simple compression algorithm
- It has the advantage that can detect patterns or composed words (e.g. “breakfast” = “break” + “fast”)

Byte Pair Encoding Data Compression Example

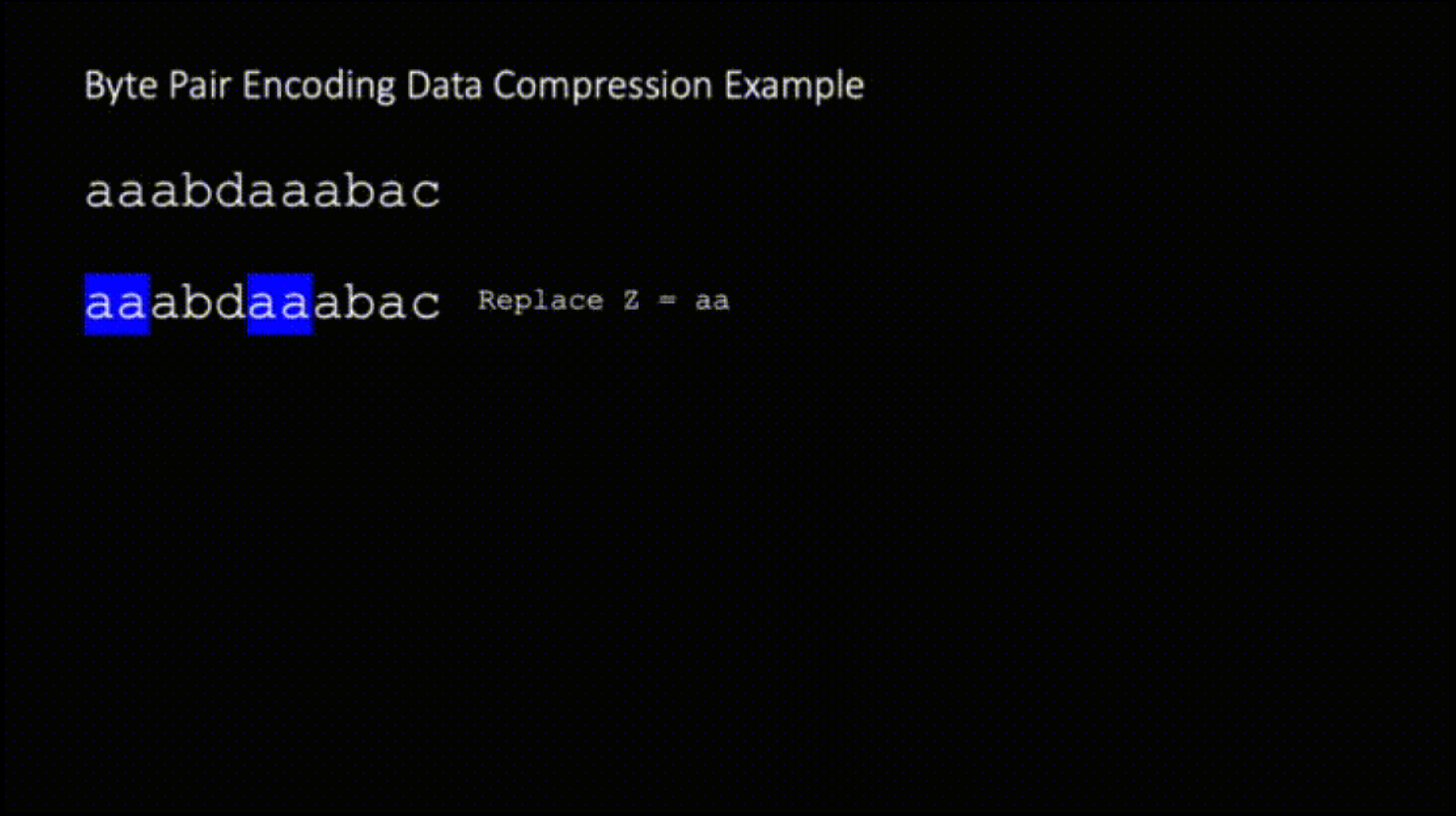
Byte Pair Encoding Data Compression Example

aaabdaaabac

Byte Pair Encoding Data Compression Example

aaabdaaabac

aaabdaaabac


Byte Pair Encoding Data Compression Example
aaabdaaabac
aaabdaaabac
Replace Z = aa

Byte Pair Encoding Data Compression Example

aaabdaaabac

aaabdaaabac Replace Z = aa

ZabdZabac

Byte Pair Encoding Data Compression Example

aaabdaaabac

aaabdaaabac Replace Z = aa

ZabdZabac Replace Y = ab

Byte Pair Encoding Data Compression Example

aaabdaaabac

aaabdaaabac Replace Z = aa

ZabdZabac Replace Y = ab

ZYdZYac

Byte Pair Encoding Data Compression Example

```
aaabdaaabac

aaabdaaabac    Replace Z = aa

ZabdZabac      Replace Y = ab

ZYdZYac        Replace X = ZY
```

Byte Pair Encoding Data Compression Example

```
aaabdaaabac

aaabdaaabac   Replace Z = aa

ZabdZabac     Replace Y = ab

ZYdZYac       Replace X = ZY

XdXac
```

Byte Pair Encoding Data Compression Example

aaabdaaabac

aaabdaaabac   Replace Z = aa

ZabdZabac   Replace Y = ab

ZYdZYac   Replace X = ZY

XdXac   Final compressed string

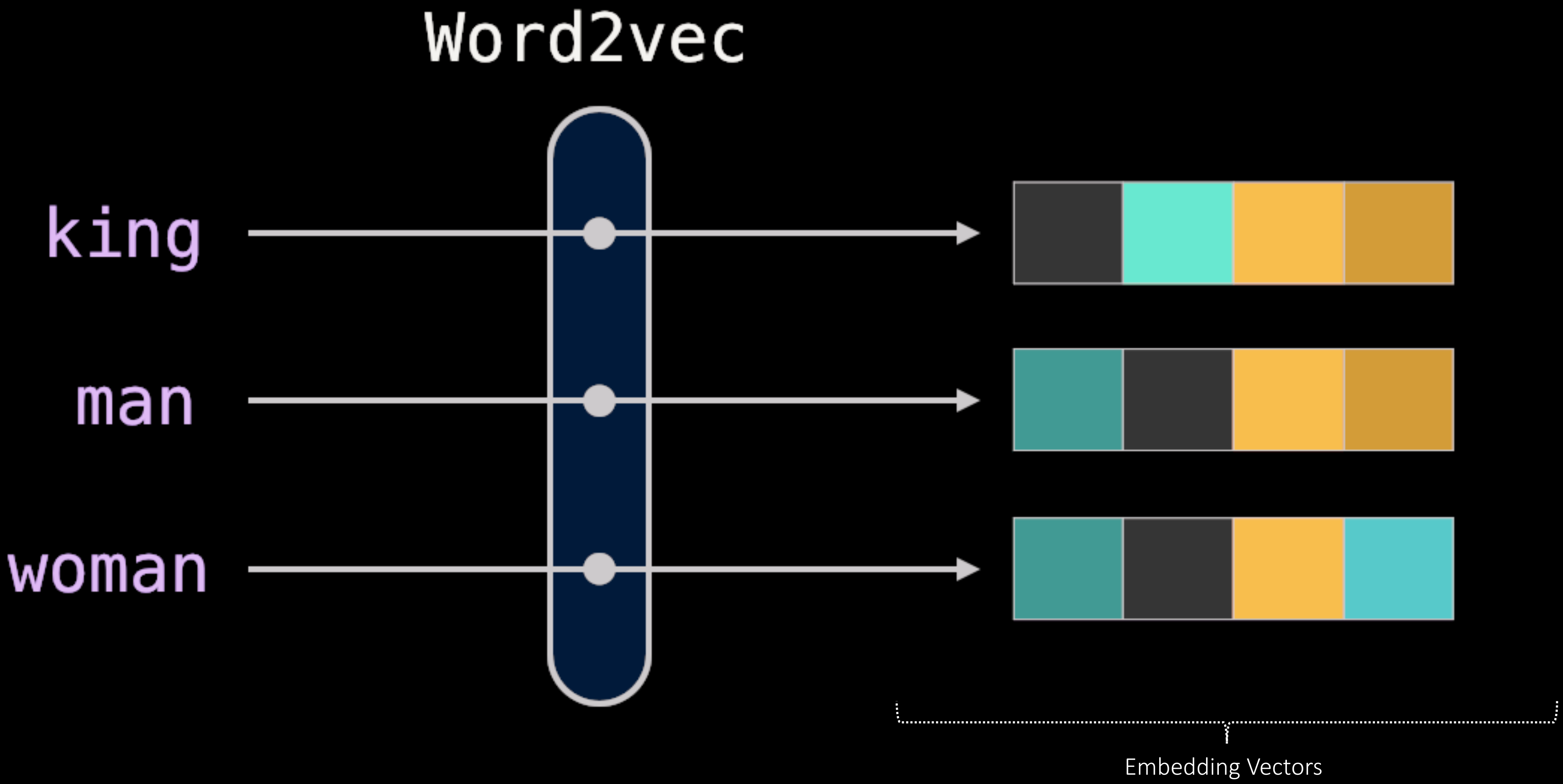
Replacement Table

Byte pair	Replacement
X	ZY
ab	Y
aa	Z

Embeddings: Improving Encoding

- Previously we used “one hot” encoding
- Embeddings are an improvement
- There are many embedding methods
- Examples of embedding are “Word2Vec”, and “GloVE”

Embeddings: Concept



Embeddings: Dataset

Embeddings: Dataset

$\boxed{\text{Thou shalt not}}$ make a machine in the likeness of a human mind

Sliding window across running text

Embeddings: Dataset

Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

thou	shalt	not	make	a	machine	in	the	…

Embeddings: Dataset

Thou shalt not make a machine in the likeness of a human mind

Sliding window across running text

thou	shalt	not	make	a	machine	in	the	…

Dataset

input 1	input 2	output
thou	shalt	not

Embeddings: Dataset

Thou **shalt not make** a machine in the likeness of a human mind

Sliding window across running text

thou	shalt	not	make	a	machine	in	the	…
thou	shalt	not	make	a	machine	in	the	

Dataset

input 1	input 2	output
thou	shalt	not
shalt	not	make

Embeddings: Dataset

Thou shalt not make a machine in the likeness of a human mind

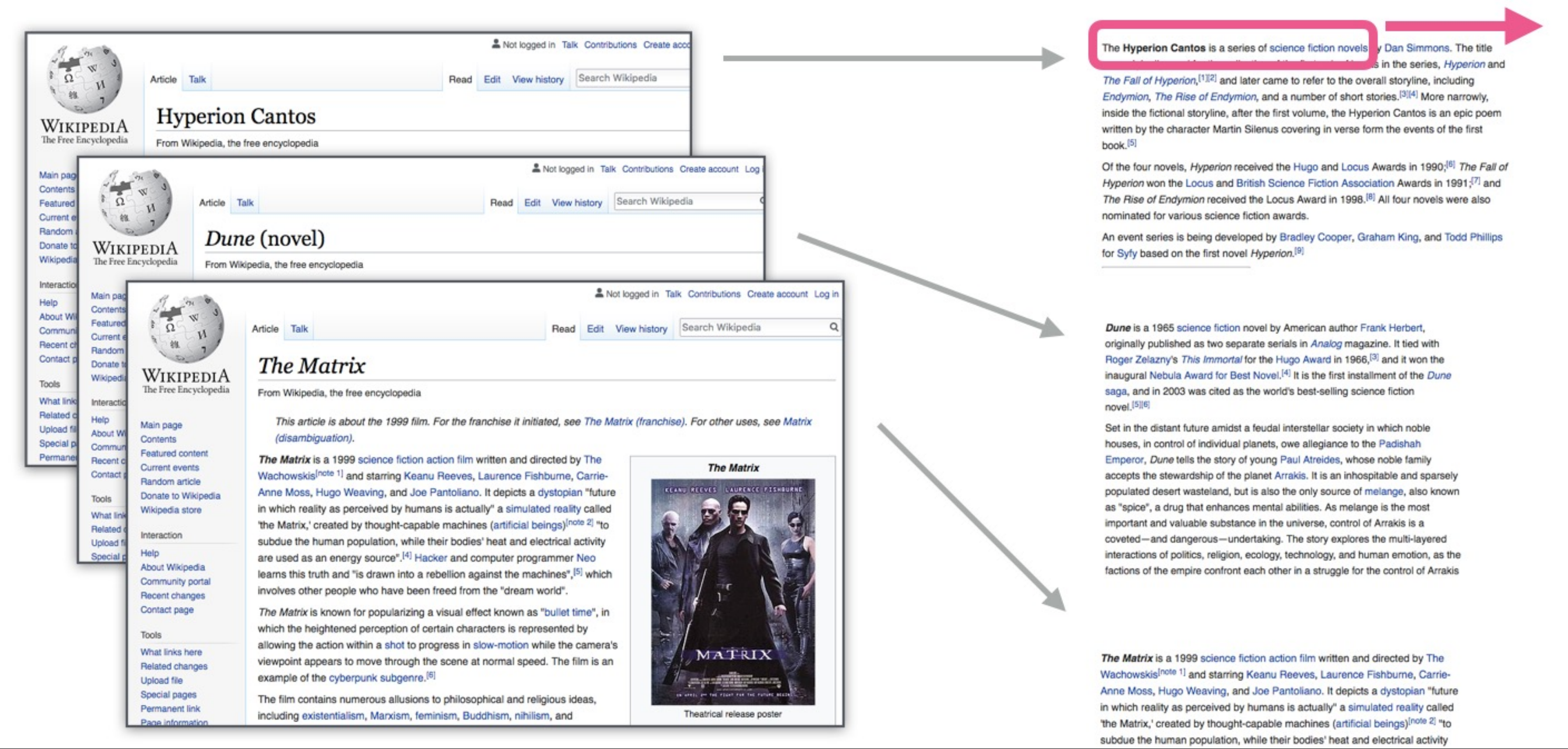
Sliding window across running text

thou	shalt	not	make	a	machine	in	the	…
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	

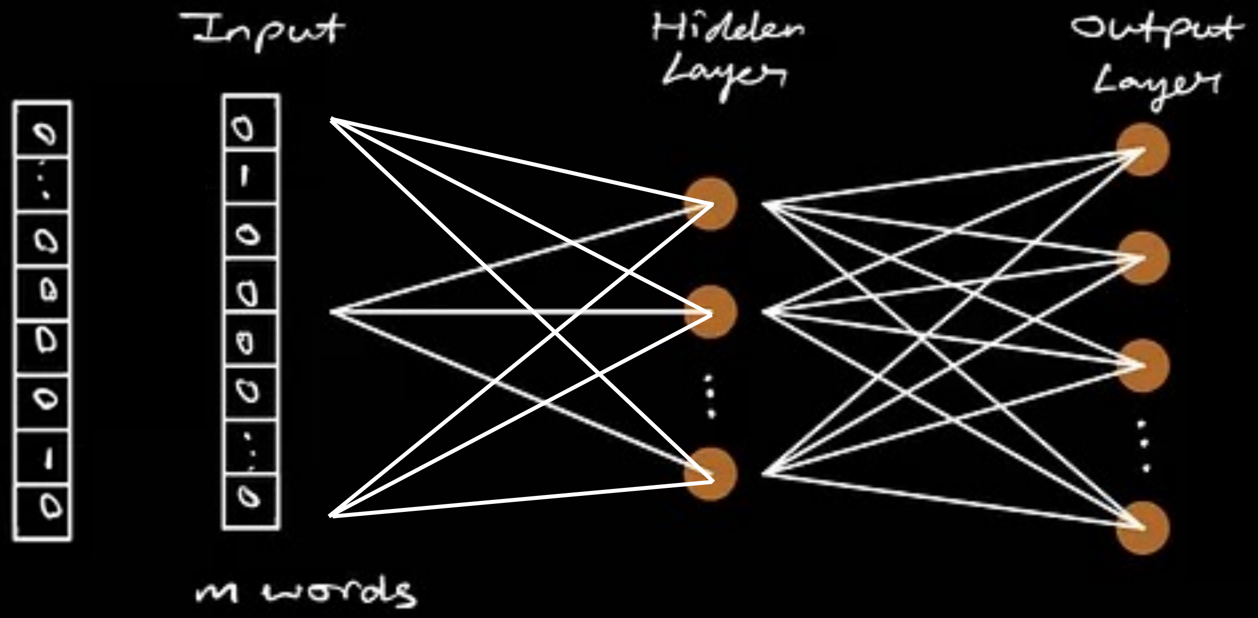
Dataset

input 1	input 2	output
thou	shalt	not
shalt	not	make
not	make	a
make	a	machine
a	machine	in

Embeddings: Dataset

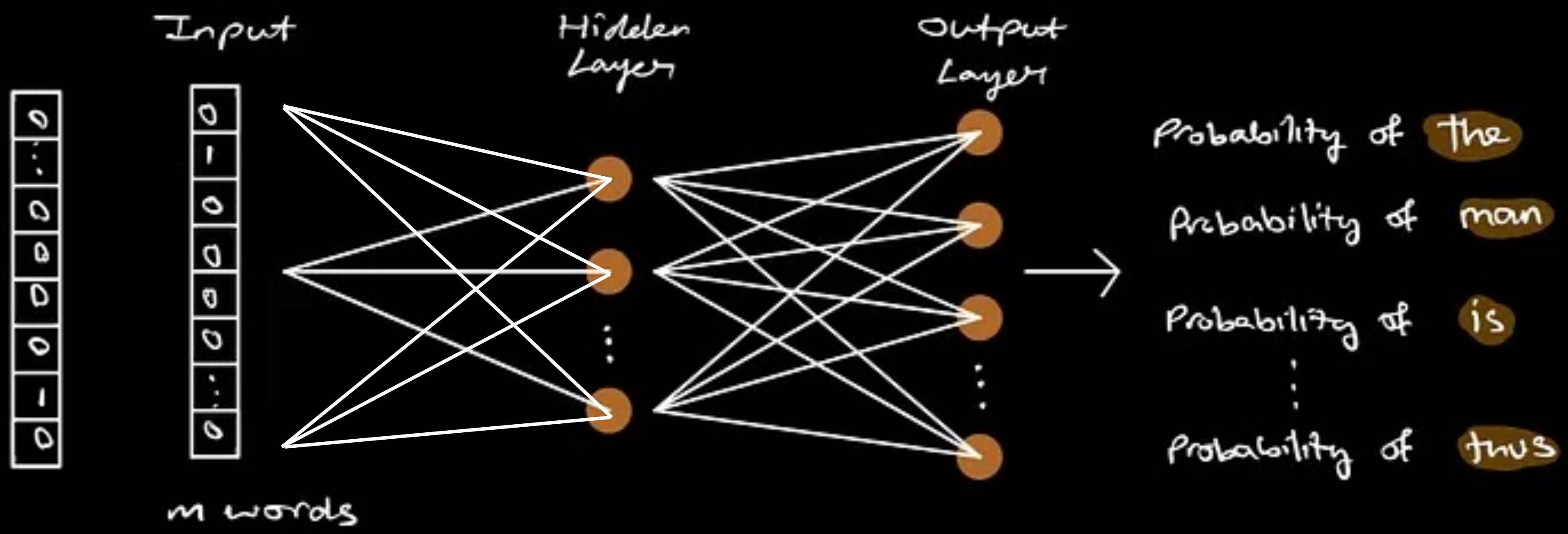


Embeddings: Model



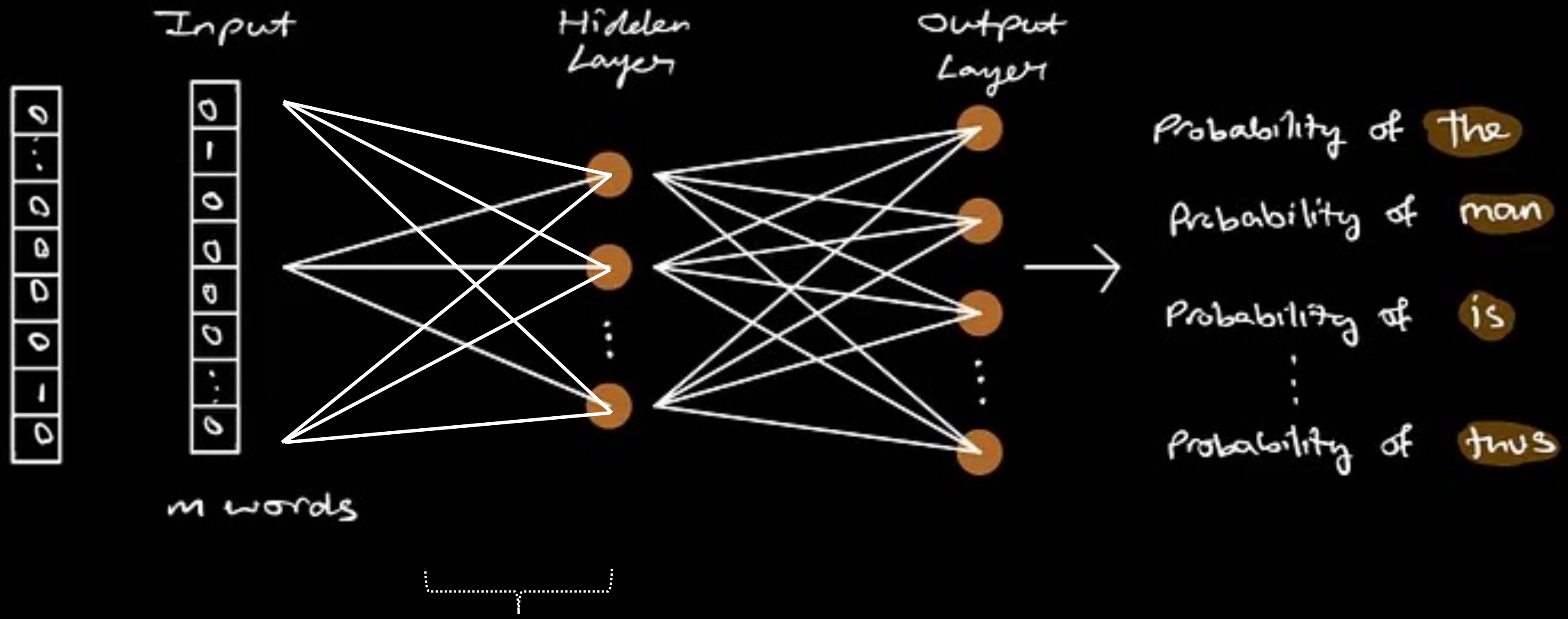
Input: One hot encoded words

Embeddings: Model



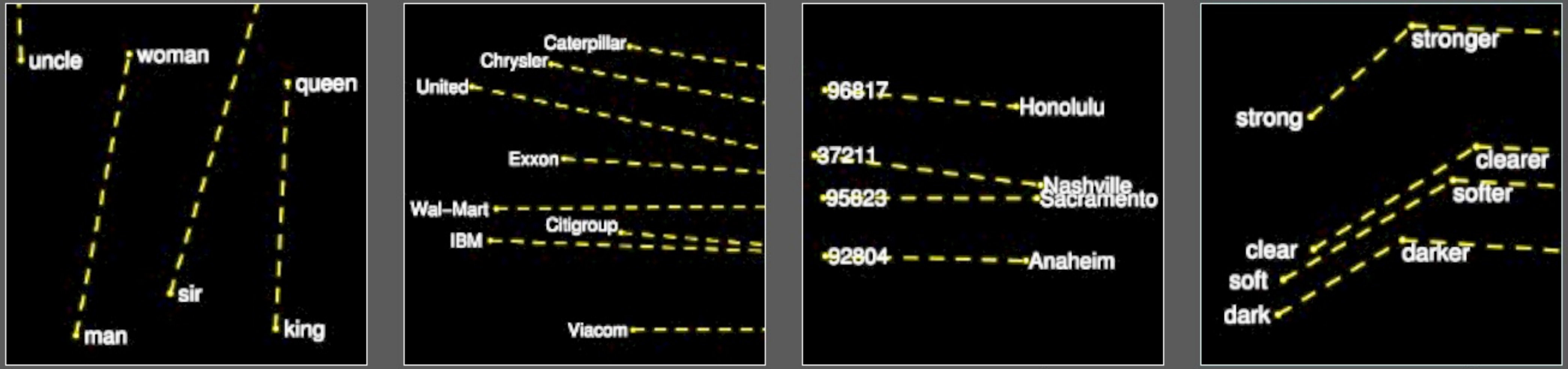
Output layer is a 'softmax'

Embeddings: Model



The Embedding is the hidden layer's weight matrix

Embeddings



man - woman | company - ceo | city - zip code | comparative - superlative

The underlying concept that distinguishes *man* from *woman*, i.e. sex or gender, may be equivalently specified by various other word pairs, such as *king* and *queen* or *brother* and *sister*. To state this observation mathematically, we might expect that the vector differences *man* - *woman*, *king* - *queen*, and *brother* - *sister* might all be roughly equal. This property and other interesting patterns can be observed in the above set of visualizations.

End of Part 1