Large Language Models
Session 2

Nathanaël Fijalkow
CNRS, LaBRI, Bordeaux
Co-teacher: Marc Lelarge

Outline for today

- Self-evaluation for session 1
- Tokenization
- Fine-tuning

Self-evaluation for Session 1

- (1) What is the difference between nn.embedding and nn.linear?
- (2) In what sense are attention weights "trainable"?
- (3) How many matrix multiplications are there in the computation of a single head?
- (4) Why do use scaled attention instead of non-scaled version?
- (5) What is the difference between normalization layer and batch normalization?
- (6) What are the relationships between the dimensions of keys, queries, and values? Which is the largest (in dimension)?
- (7) What does "auto-regressive" mean?
- (8) Why is "sliding windows" not a great name, and "expanding windows" would be better?
- (9) What are the benefits of the sliding / expanding windows?
- (10) What are the alternatives to cross entropy loss?
- (11) What do "shortcut connections" (also called "residual connections") do?
- (12) What is dropout?

Alternative: Read about KV-cache

https://huggingface.co/blog/kv-cache-quantization

What is the difference between nn.embedding and nn.linear?

Both `nn.Embedding` and `nn.Linear` are modules in PyTorch that deal with transforming inputs, but they serve different purposes and operate differently:

`nn.Embedding`

- **Purpose:** This module is used to represent categorical data, such as words in a vocabulary. It creates a lookup table where each unique category (e.g., word) is assigned a unique vector (embedding).
- **Operation:** It works by looking up the embedding vector corresponding to the given input index. It's essentially a dictionary that maps indices to vectors.

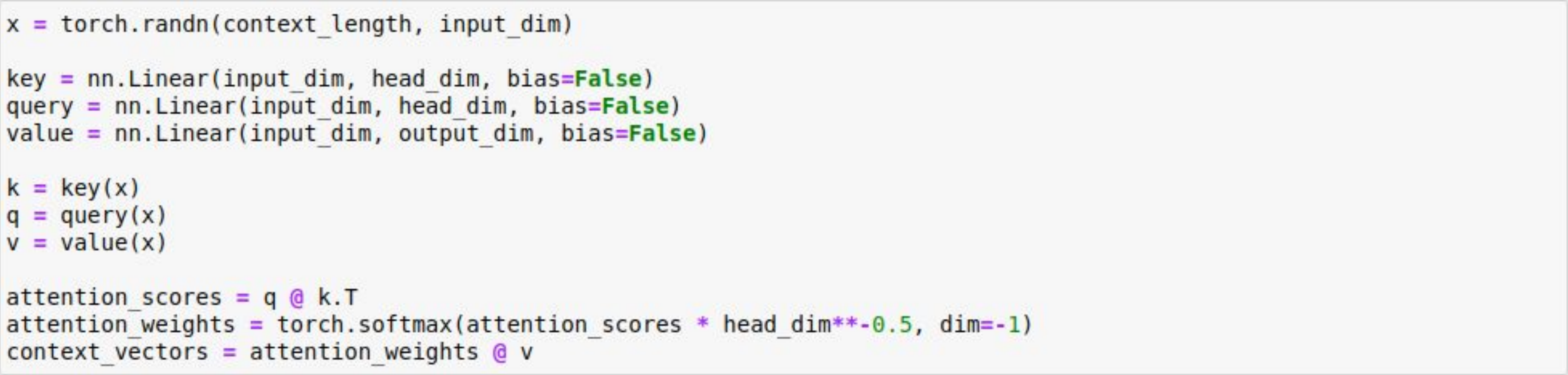
`nn.Linear`

- **Purpose:** This module performs a linear transformation on the input data. It applies a weight matrix and a bias vector to the input.
- **Operation:** It calculates the dot product of the input with the weight matrix and adds the bias vector. This is a fundamental operation in many neural networks.

In Summary:

- Use `nn.Embedding` for representing categorical data as dense vectors.
- Use `nn.Linear` for performing linear transformations in neural networks.

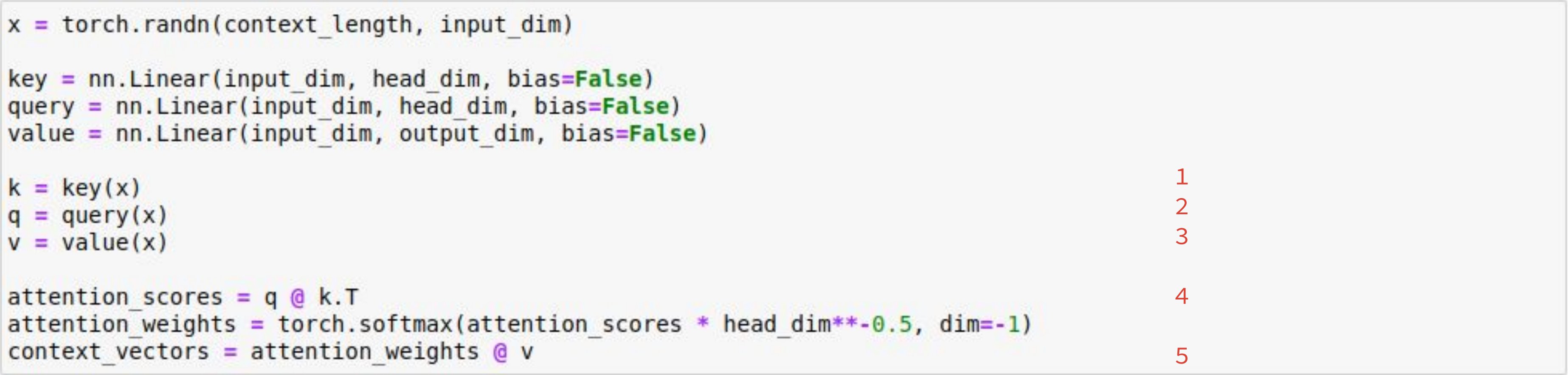
In what sense are attention weights "trainable"?



```
x = torch.randn(context_length, input_dim)

key = nn.Linear(input_dim, head_dim, bias=False)
query = nn.Linear(input_dim, head_dim, bias=False)
value = nn.Linear(input_dim, output_dim, bias=False)

k = key(x)
q = query(x)
v = value(x)

attention_scores = q @ k.T
attention_weights = torch.softmax(attention_scores * head_dim**-0.5, dim=-1)
context_vectors = attention_weights @ v
```

The matrices *key*, *query*, and *value* include trainable parameters

How many matrix multiplications are there in the computation of a single head?



```
x = torch.randn(context_length, input_dim)

key = nn.Linear(input_dim, head_dim, bias=False)
query = nn.Linear(input_dim, head_dim, bias=False)
value = nn.Linear(input_dim, output_dim, bias=False)

k = key(x)                                                                 1
q = query(x)                                                               2
v = value(x)                                                               3

attention_scores = q @ k.T                                                 4
attention_weights = torch.softmax(attention_scores * head_dim**-0.5, dim=-1)
context_vectors = attention_weights @ v                                    5
```

There are 5 matrix multiplications (same with batching)

Let's be a bit more precise

C = context_length

I = input_dim

H = head_dim

O = output_dim

- key(x): (C x I) x (I x H) -> C x H
- query(x): (C x I) x (I x H) -> C x H
- value(x): (C x I) x (I x O) -> C x O
- attention_scores: (C x H) x (H x C) -> C x C
- context_vectors: (C x C) x (C x O) -> C x O

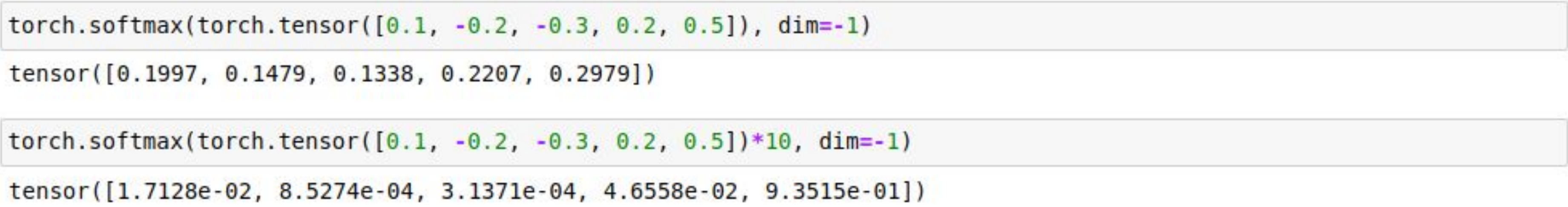
Quadratic in context length!

Why do use scaled attention instead of non-scaled version?

The Problem with Non-Scaled Attention

- **Vanishing Gradients:** The softmax function in attention is used to normalize the attention scores into a probability distribution. However, when the dot products of queries and keys become large, the softmax function can saturate. This means its gradients become extremely small.
- **Unstable Training:** Small gradients hinder the learning process during backpropagation, making it difficult for the model to update its weights effectively. This can lead to slow convergence or even prevent the model from learning altogether.

Illustration of softmax saturation:



```
torch.softmax(torch.tensor([0.1, -0.2, -0.3, 0.2, 0.5]), dim=-1)
```

```
tensor([0.1997, 0.1479, 0.1338, 0.2207, 0.2979])
```

```
torch.softmax(torch.tensor([0.1, -0.2, -0.3, 0.2, 0.5])*10, dim=-1)
```

```
tensor([1.7128e-02, 8.5274e-04, 3.1371e-04, 4.6558e-02, 9.3515e-01])
```

Formalization of the scaling

Assume u,v are vectors of dimension d:

u,v ~ N(0,1)

What is the distribution of u \cdot v?

Answer: Exp[u \cdot v] = 0 but Var(u \cdot v) = d

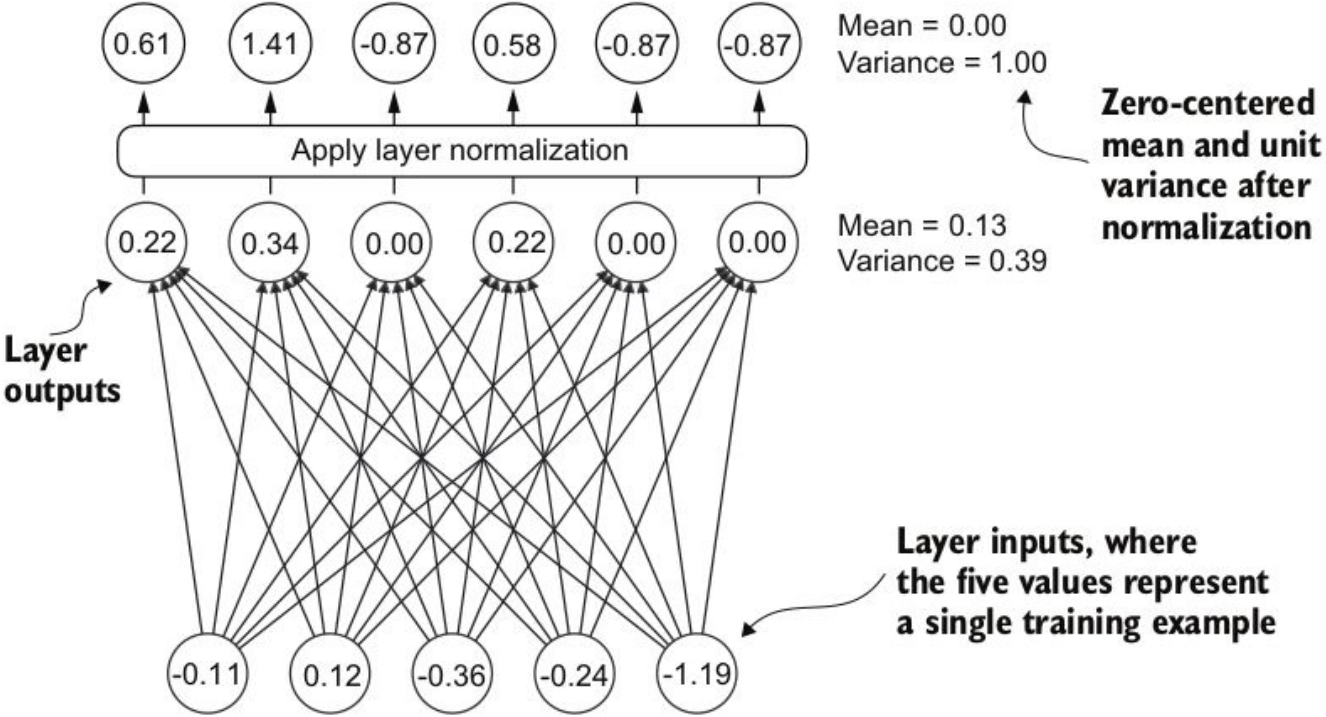
But: Var(u \cdot v / sqrt(d)) = 1

Why not u \cdot v / ||u \cdot v||? **Not differentiable**!

What is the difference between normalization layer and batch normalization?

Normalization Layers (Layer Normalization)

- **Normalization Axis:** Normalizes the activations **across all features within a single training example**. Imagine it as normalizing each row in a batch of data.
- **Effect:** Ensures that the inputs to each layer have a consistent scale and distribution, regardless of the batch size.
- **Benefits:**
  - **Effective for variable batch sizes:** Works well even with small batch sizes or online learning where the batch size is 1.
  - **Suitable for recurrent neural networks (RNNs) and Transformers:** Often preferred in models dealing with sequential data, as it helps stabilize the learning process in the presence of long-term dependencies.
- **Example:** In a layer with 3 features (x1, x2, x3), layer normalization would calculate the mean and standard deviation across these 3 features for each individual training example in the batch and normalize accordingly.



Zero-centered mean and unit variance after normalization

Layer outputs

Layer inputs, where the five values represent a single training example

What is the difference between normalization layer and batch normalization?

Batch Normalization

- **Normalization Axis:** Normalizes the activations **across all training examples within a mini-batch for each individual feature**. Imagine it as normalizing each column in a batch of data.
- **Effect:** Reduces internal covariate shift by keeping the inputs to each layer relatively consistent during training.
- **Benefits:**
  - **Faster training:** Helps the model converge faster.
  - **Improved generalization:** Can lead to better performance on unseen data.
- **Example:** In a batch of 100 training examples with 3 features (x1, x2, x3), batch normalization would calculate the mean and standard deviation of x1 across all 100 examples, then normalize x1 for each example. It would do the same for x2 and x3.

What is the difference between normalization layer and batch normalization?

Feature	Normalization Layers	Batch Normalization
Normalization Axis	Across features within a single example	Across examples within a mini-batch for each feature
Batch Size Dependency	Independent of batch size	Requires sufficiently large batch sizes
Common Use Cases	RNNs, Transformers, small batch sizes	Convolutional Neural Networks (CNNs), large batch sizes

What are the relationships between the dimensions of keys, queries, and values? Which is the largest (in dimension)?

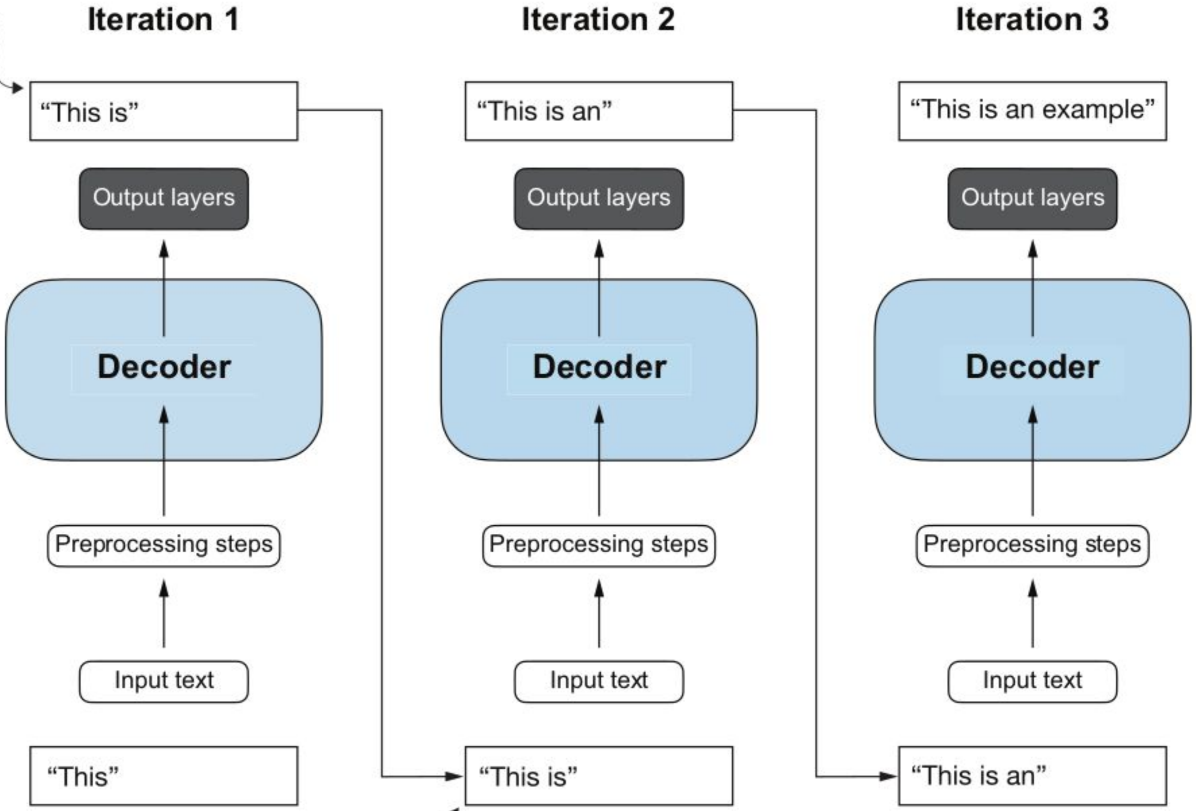
Dimensions: d_k = d_q, independent of d_v

In the original paper:

- d_k = d_v = 64
- h = 8 parallel heads
- d_model = 8 * 64 = 512

What does "auto-regressive" mean?

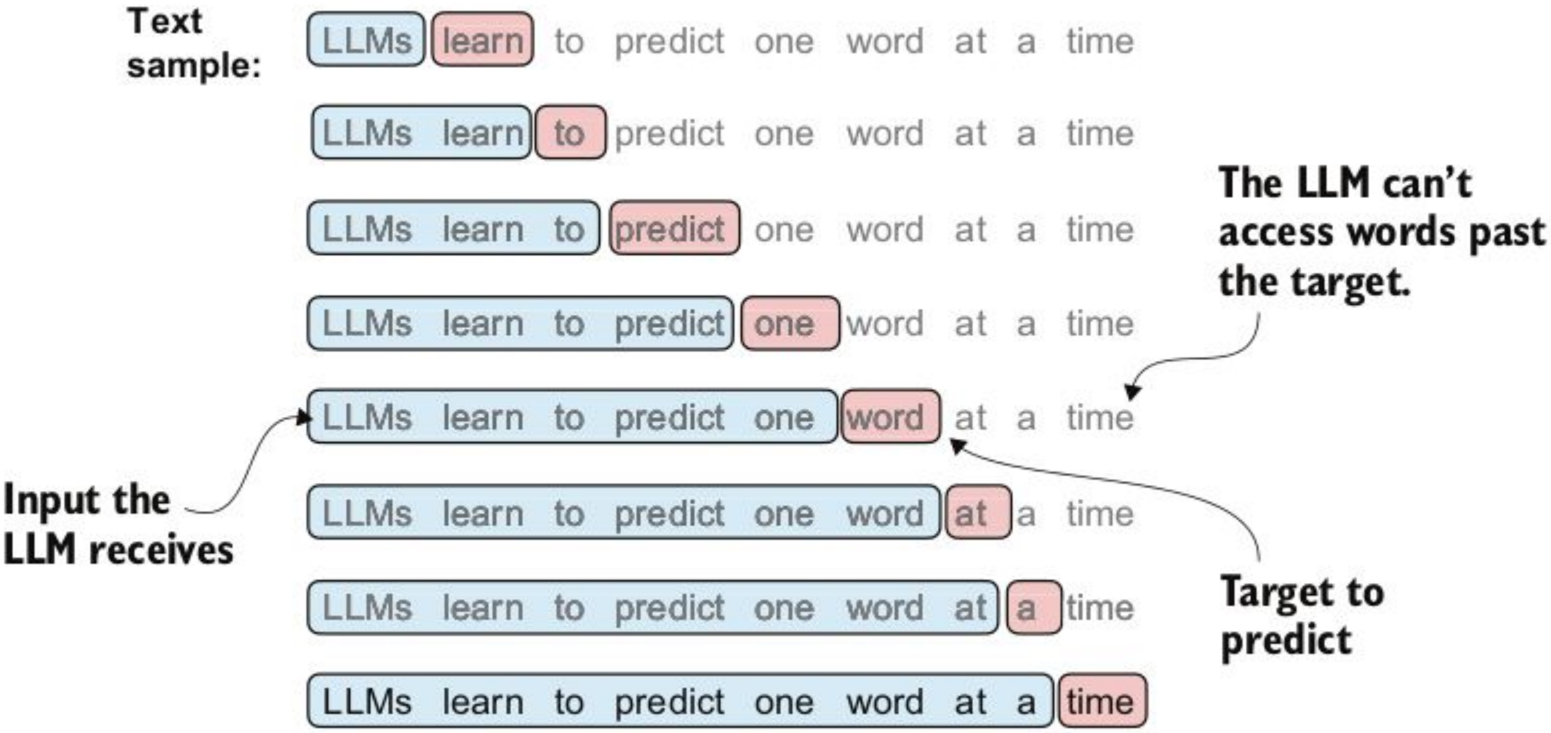
It means that for generating a single new token we feed the model with the input + all tokens generated so far.



Creates the next word based on the input text

The output of the previous round serves as input to the next round.

Why is "sliding windows" not a great name, and "expanding windows" would be better?



Text sample:

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

The LLM can't access words past the target.

Input the LLM receives

Target to predict

What are the benefits of the sliding / expanding windows?

Fix c = context_length

A single data point (meaning, a sequence of c+1 tokens) becomes for free c data points:

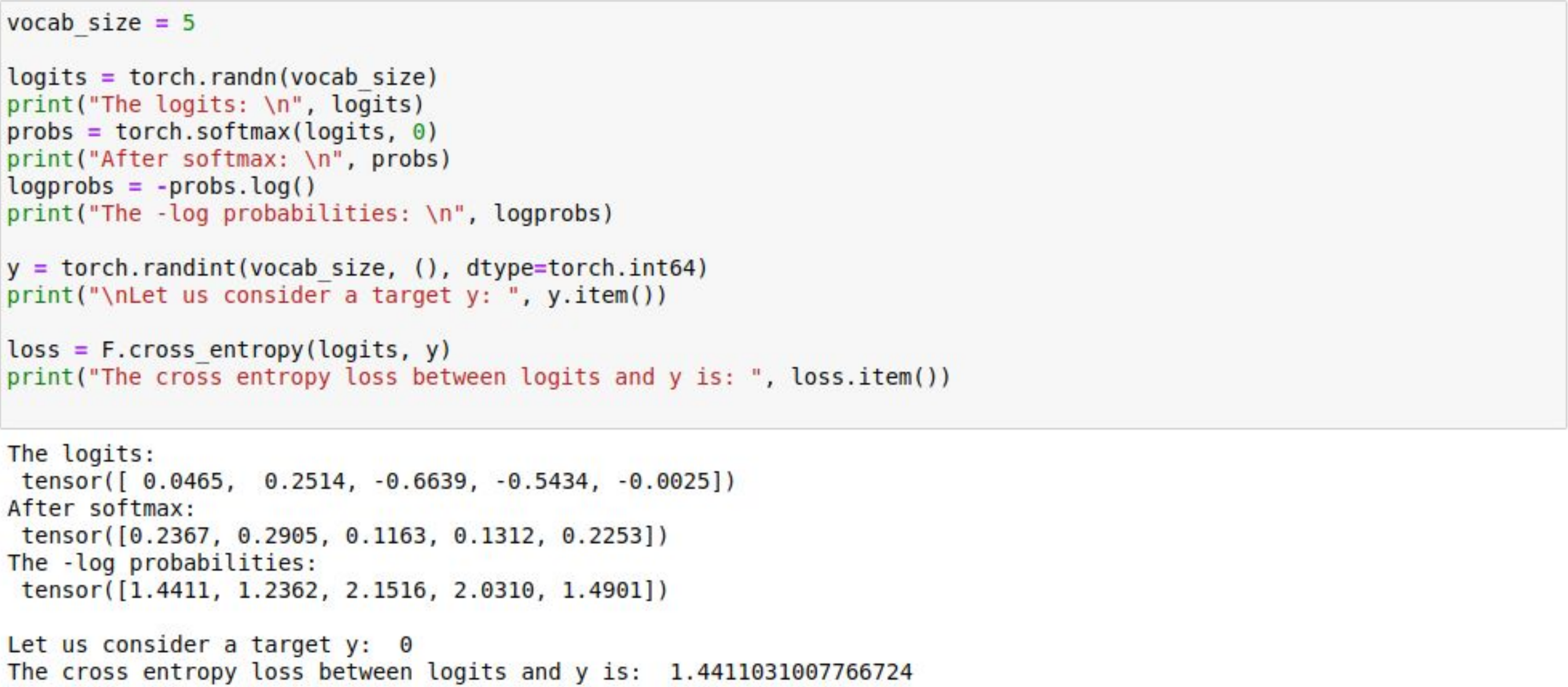
- A single tensor stores all c data points
- Running the model once one the whole sequence yields predictions for all c data points

What are the alternatives to cross entropy loss?

There are many, depending on the task at hand:

- Mean Squared Error (MSE)
- Contrastive Loss
- Connectionist temporal classification (CTC)

Cross entropy loss



```python
vocab_size = 5

logits = torch.randn(vocab_size)
print("The logits: \n", logits)
probs = torch.softmax(logits, 0)
print("After softmax: \n", probs)
logprobs = -probs.log()
print("The -log probabilities: \n", logprobs)

y = torch.randint(vocab_size, (), dtype=torch.int64)
print("\nLet us consider a target y: ", y.item())

loss = F.cross_entropy(logits, y)
print("The cross entropy loss between logits and y is: ", loss.item())
```

```
The logits:
 tensor([ 0.0465,  0.2514, -0.6639, -0.5434, -0.0025])
After softmax:
 tensor([0.2367, 0.2905, 0.1163, 0.1312, 0.2253])
The -log probabilities:
 tensor([1.4411, 1.2362, 2.1516, 2.0310, 1.4901])

Let us consider a target y:  0
The cross entropy loss between logits and y is:  1.4411031007766724
```

What is cross entropy loss?

Cross entropy measures the difference between probability distributions: it quantifies the dissimilarity between the predicted probability distribution and the true probability distribution.

In language modelling we do not have the true distribution of words, it is approximated from a training set:

$$H(T, q) = -\sum_{i=1}^{N} \frac{1}{N} \log_2 q(x_i)$$

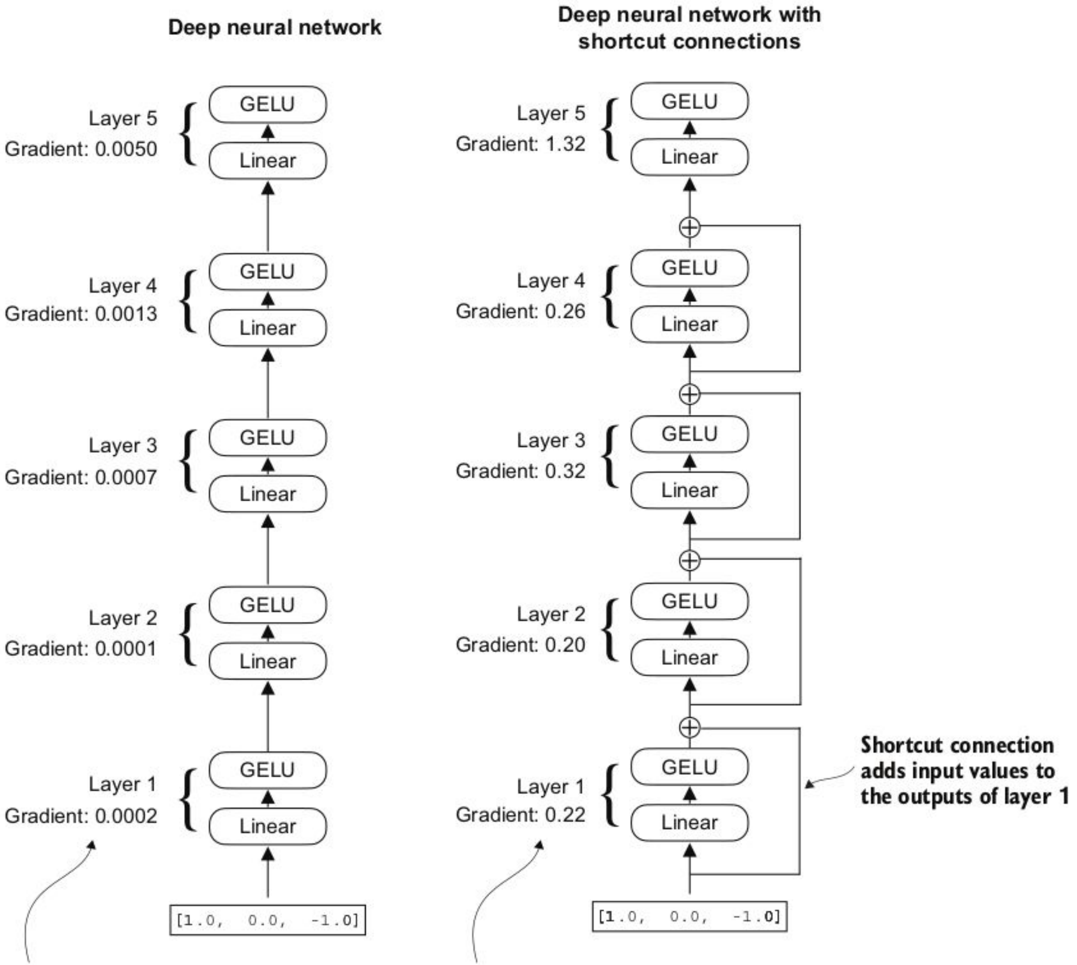
Where N is the number of tokens in the training set and q(x_i) is the probability that the model outputs x_i.

Why is cross entropy loss interesting?

- **Maximum likelihood estimation:** Minimizing cross-entropy is equivalent to maximizing the likelihood of the observed data.
- **Encourages accurate probabilities:** It encourages the model to produce probabilities that closely match the true distribution, not just predict the correct class.
- **Smooth and differentiable:** Cross-entropy loss is a smooth and differentiable function, which is crucial for gradient-based optimization algorithms like gradient descent.
- **Avoids saturation:** Unlike some other loss functions (e.g., mean squared error with sigmoid), cross-entropy with softmax reduces the problem of saturating gradients.

What do "shortcut connections" (also called "residual connections") do?

Shortcut connections, also known as skip connections or residual connections, provide a pathway for the gradient to flow more easily during backpropagation, mitigating the vanishing gradient problem and enabling the training of much deeper networks.



Shortcut connection adds input values to the outputs of layer 1

In very deep networks, the gradient values in early layers become vanishingly small

The shortcut connections help with maintaining relatively large gradient values even in early layers

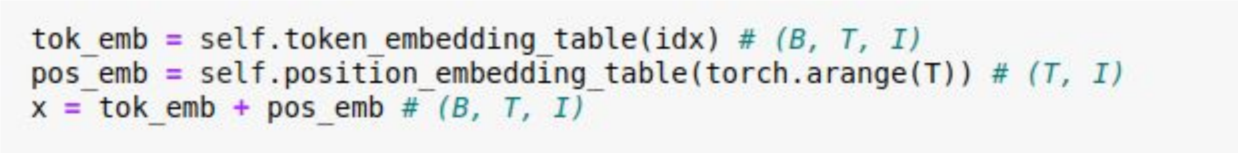
What is dropout?

Dropout is a regularization technique used in neural networks to prevent overfitting. It works by randomly dropping out (setting to zero) a certain proportion of neurons in a layer during each training step.

- **Prevents Overfitting:** By randomly dropping out neurons, dropout prevents the network from learning complex co-adaptations that are specific to the training data. This helps the model generalize better to unseen data.
- **Ensemble Effect:** Dropout can be seen as training an ensemble of multiple smaller networks. Each training step effectively samples a different subnetwork. At test time, the average of these subnetworks is used, which improves the overall performance.
- **Reduces Co-adaptation:** Dropout forces neurons to learn more robust features that are not dependent on the presence of specific other neurons. This leads to better feature representations.

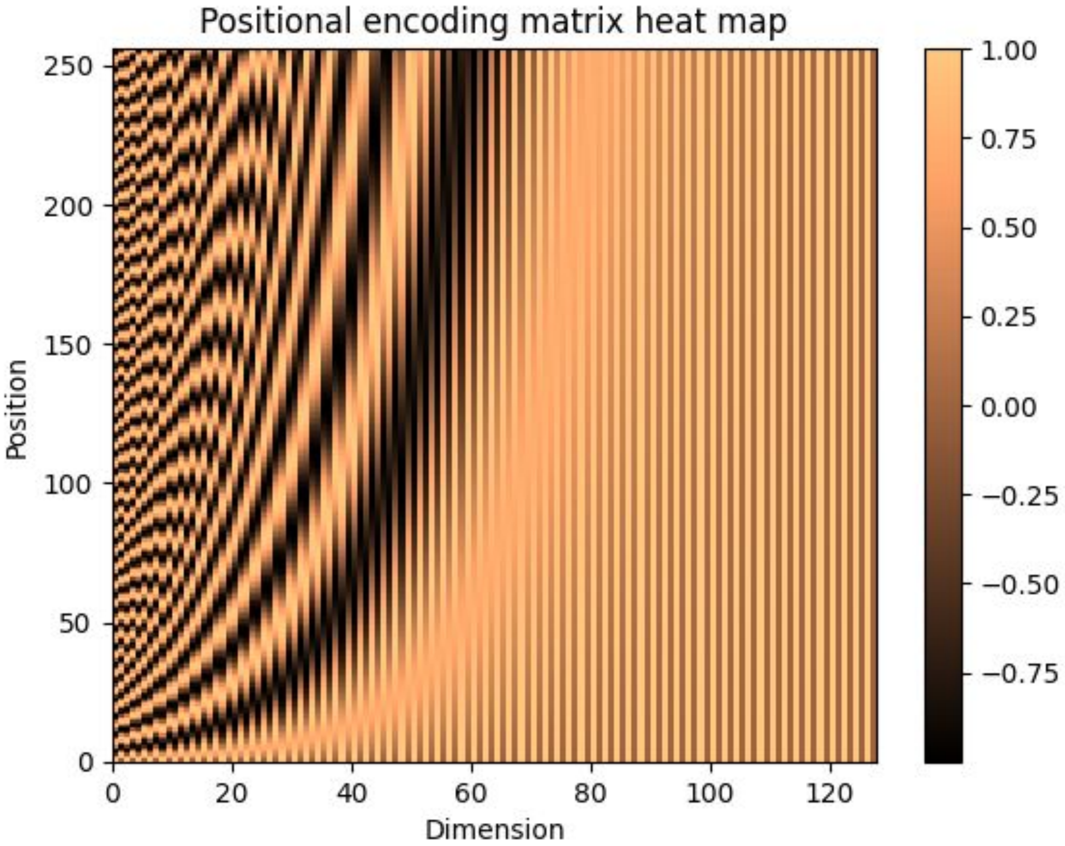
Positional embeddings

Attention scores are computed in the same way for all other tokens. But sometimes it is useful to be aware of *relative* positions (just before, just after,...), or absolute positions (at the beginning, at the end).



```
tok_emb = self.token_embedding_table(idx) # (B, T, I)
pos_emb = self.position_embedding_table(torch.arange(T)) # (T, I)
x = tok_emb + pos_emb # (B, T, I)
```

Some visualization



Tokenization

- Basics of encoding
- Pre-tokenization
- Byte-Pair Encoding (BPE)
- WordPiece

Credits

Images and contents from Chapter 6 of Hugging Face’s course on NLP:

https://huggingface.co/learn/nlp-course/chapter6

Basics

A **bit** = 0 or 1

A **byte** = typically an octet, meaning 8 bits

Character encodings:

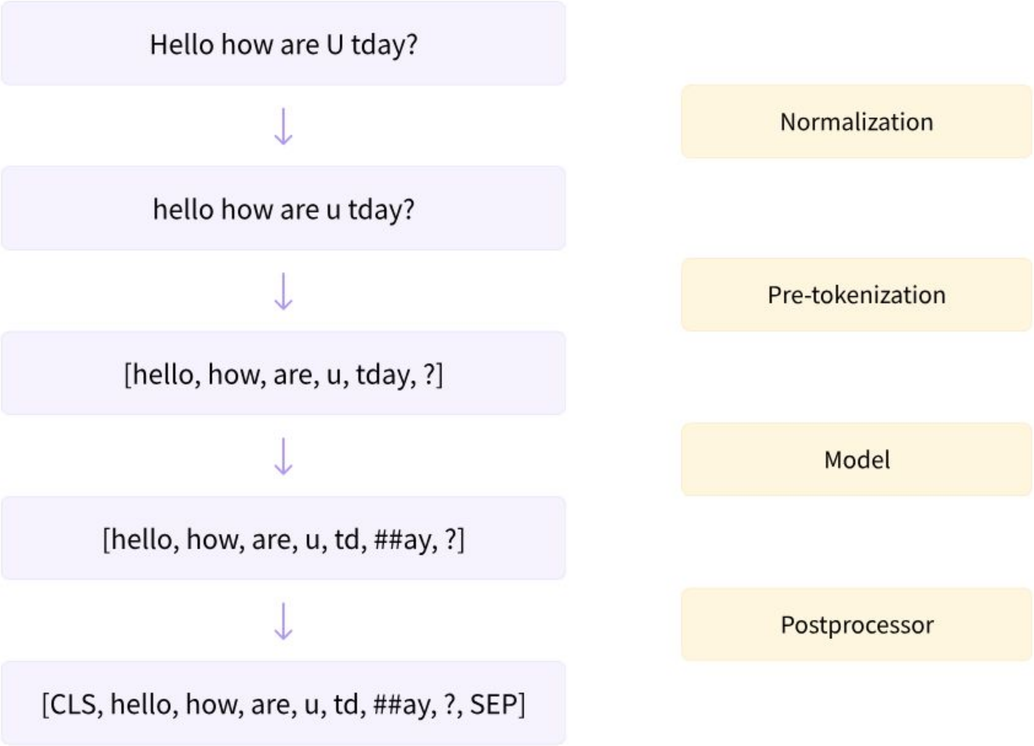
- ASCII (code unit: 7 bits)
- Unicode: UTF-8, UTF-16, UTF-32 (code unit: 8,16,32 bits)

98% of WWW is UTF-8. Technically UTF is variable-length (so infinite…)

ATTENTION

We are only considering “subword tokenization algorithms” but there are other tokenization algorithms…

The full tokenization pipeline



Hello how are U tday?

Normalization

hello how are u tday?

Pre-tokenization

[hello, how, are, u, tday, ?]

Model

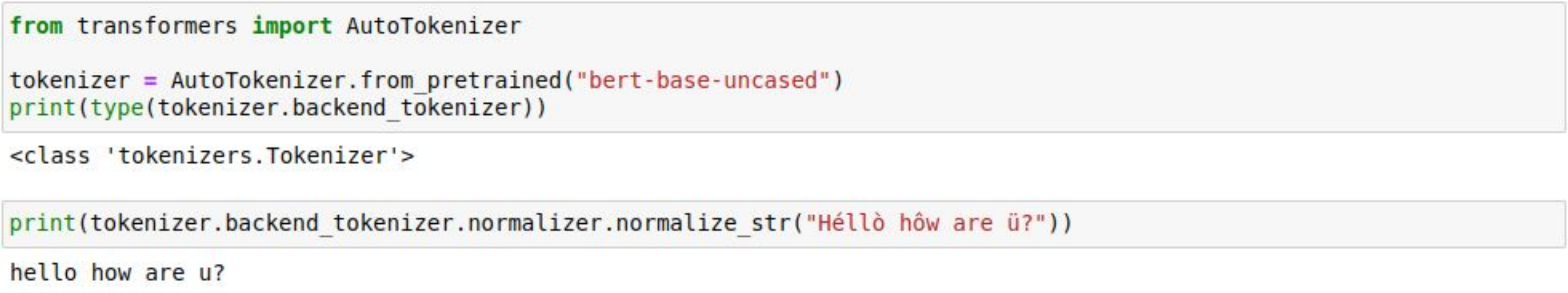
[hello, how, are, u, td, ##ay, ?]

Postprocessor

[CLS, hello, how, are, u, td, ##ay, ?, SEP]

Normalization

The normalization step involves some general cleanup, such as removing needless whitespace, lowercasing, and/or removing accents.



```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
print(type(tokenizer.backend_tokenizer))
```

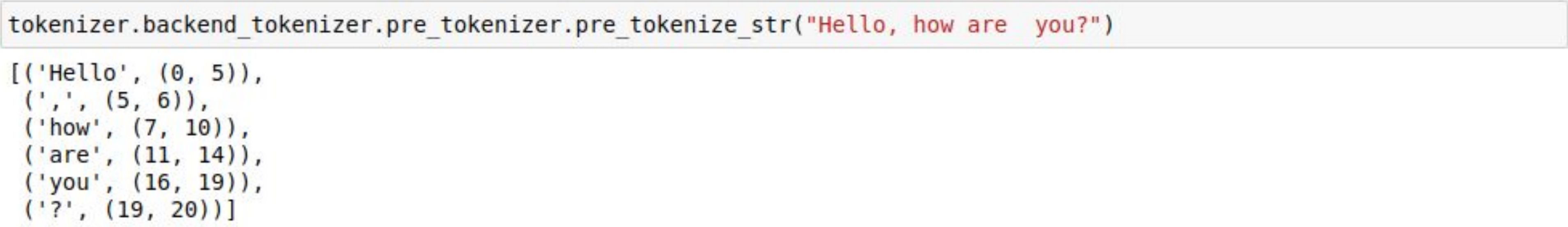
```
<class 'tokenizers.Tokenizer'>
```

```
print(tokenizer.backend_tokenizer.normalizer.normalize_str("Héllò hôw are ü?"))
```

```
hello how are u?
```

Pre-tokenization

Breaks a text into words (keeping the offsets):



```
tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str("Hello, how are  you?")
```

```
[('Hello', (0, 5)),
 (',', (5, 6)),
 ('how', (7, 10)),
 ('are', (11, 14)),
 ('you', (16, 19)),
 ('?', (19, 20))]
```

Pre-tokenization

Again there are many variants…

SentencePiece is a simple pre-tokenization algorithm:

- Treats everything as Unicode characters
- Replaces spaces with “_”

Tokenization algorithms

Two components:

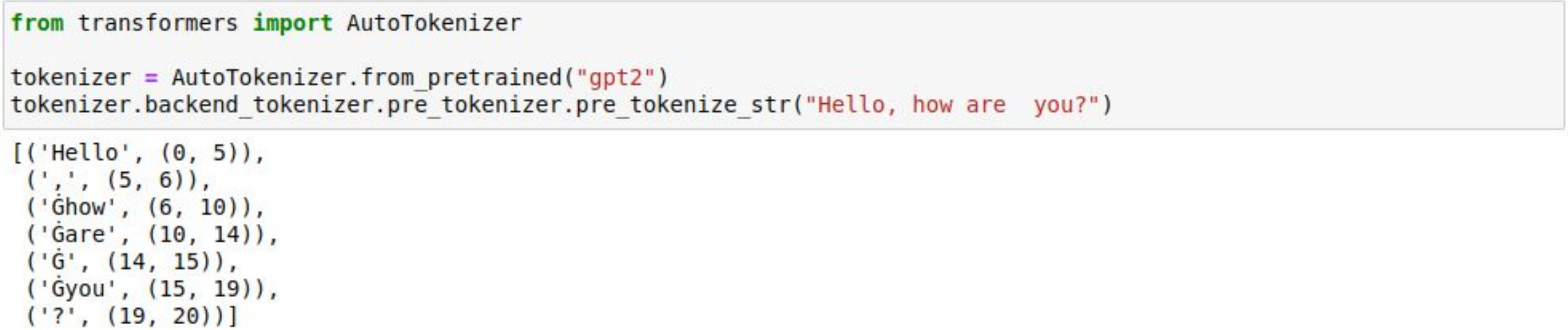
- The *training* algorithm: preprocessing on a training set, to determine what will be the tokens

- The *tokenization* algorithm: at run time, transforming text inputs into sequences of tokens

Byte-Pair Encoding

Developed by OpenAI for GPT-2

Pre-tokenization adds “Ġ” before each word except the first:



```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str("Hello, how are  you?")
```

```
[('Hello', (0, 5)),
 (',', (5, 6)),
 ('Ġhow', (6, 10)),
 ('Ġare', (10, 14)),
 ('Ġ', (14, 15)),
 ('Ġyou', (15, 19)),
 ('?', (19, 20))]
```

BPE in one slide

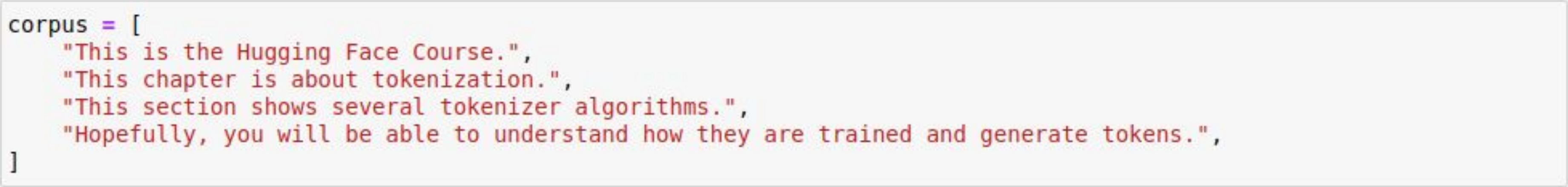
The goal is to learn merge rules, of the form:

(“Amer”, “ica”) -> “America”

Training: starting from characters, we create rules by merging the most frequent pairs, until we reach the budget number of tokens

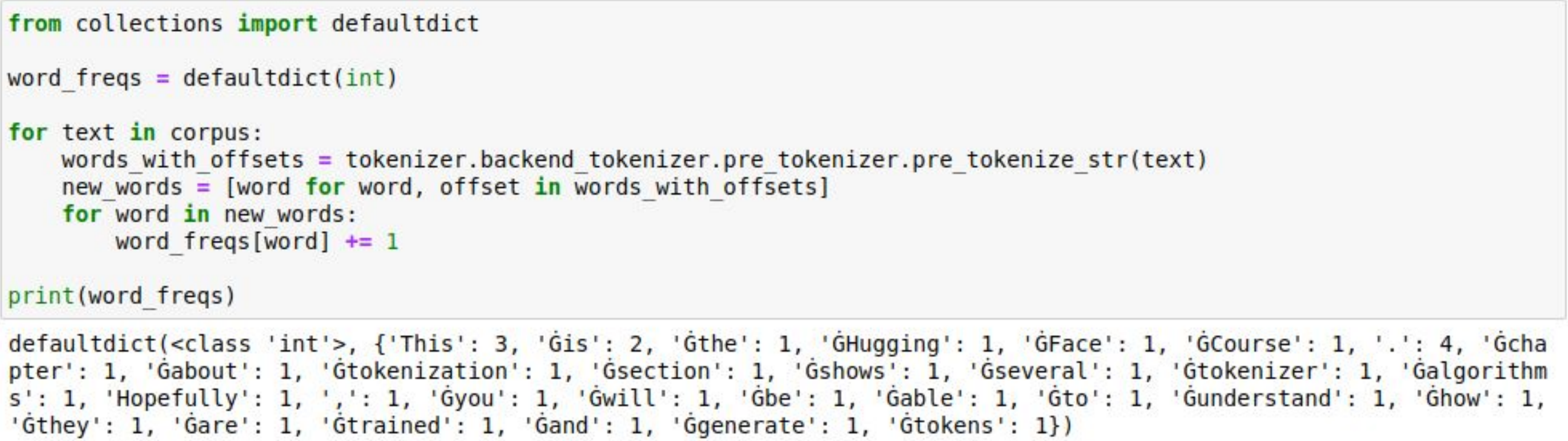
Processing: to process an input text we apply rules greedily

Example corpus



```python
corpus = [
    "This is the Hugging Face Course.",
    "This chapter is about tokenization.",
    "This section shows several tokenizer algorithms.",
    "Hopefully, you will be able to understand how they are trained and generate tokens.",
]
```

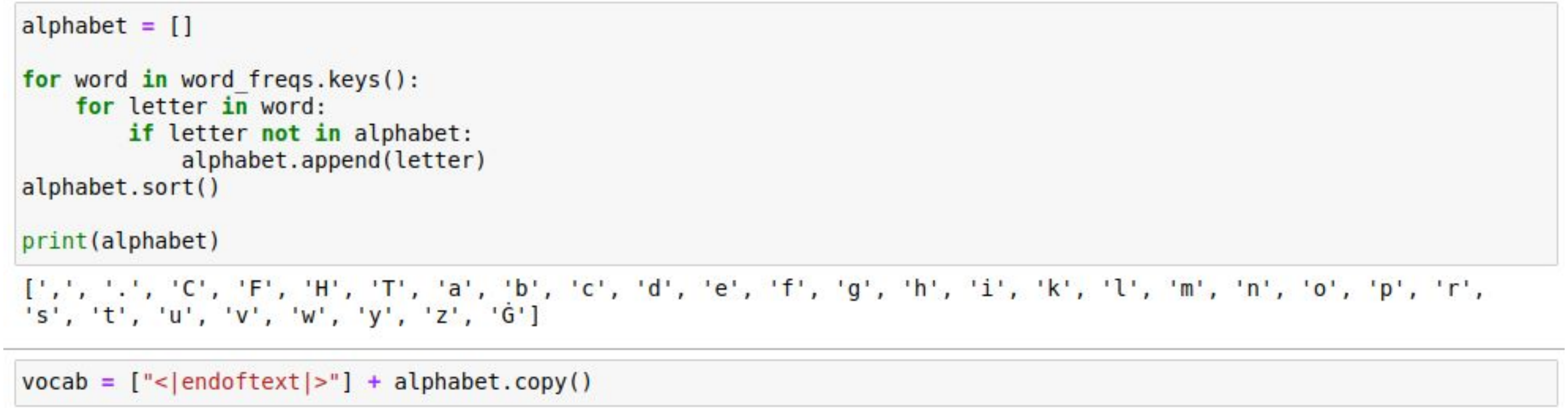
BPE training algorithm, step 0: Compute frequencies



```
from collections import defaultdict

word_freqs = defaultdict(int)

for text in corpus:
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    new_words = [word for word, offset in words_with_offsets]
    for word in new_words:
        word_freqs[word] += 1

print(word_freqs)
```

```
defaultdict(<class 'int'>, {'This': 3, 'Ġis': 2, 'Ġthe': 1, 'ĠHugging': 1, 'ĠFace': 1, 'ĠCourse': 1, '.': 4, 'Ġcha
pter': 1, 'Ġabout': 1, 'Ġtokenization': 1, 'Ġsection': 1, 'Ġshows': 1, 'Ġseveral': 1, 'Ġtokenizer': 1, 'Ġalgorithm
s': 1, 'Hopefully': 1, ',': 1, 'Ġyou': 1, 'Ġwill': 1, 'Ġbe': 1, 'Ġable': 1, 'Ġto': 1, 'Ġunderstand': 1, 'Ġhow': 1,
'Ġthey': 1, 'Ġare': 1, 'Ġtrained': 1, 'Ġand': 1, 'Ġgenerate': 1, 'Ġtokens': 1})
```

BPE training algorithm, step 1: collect characters



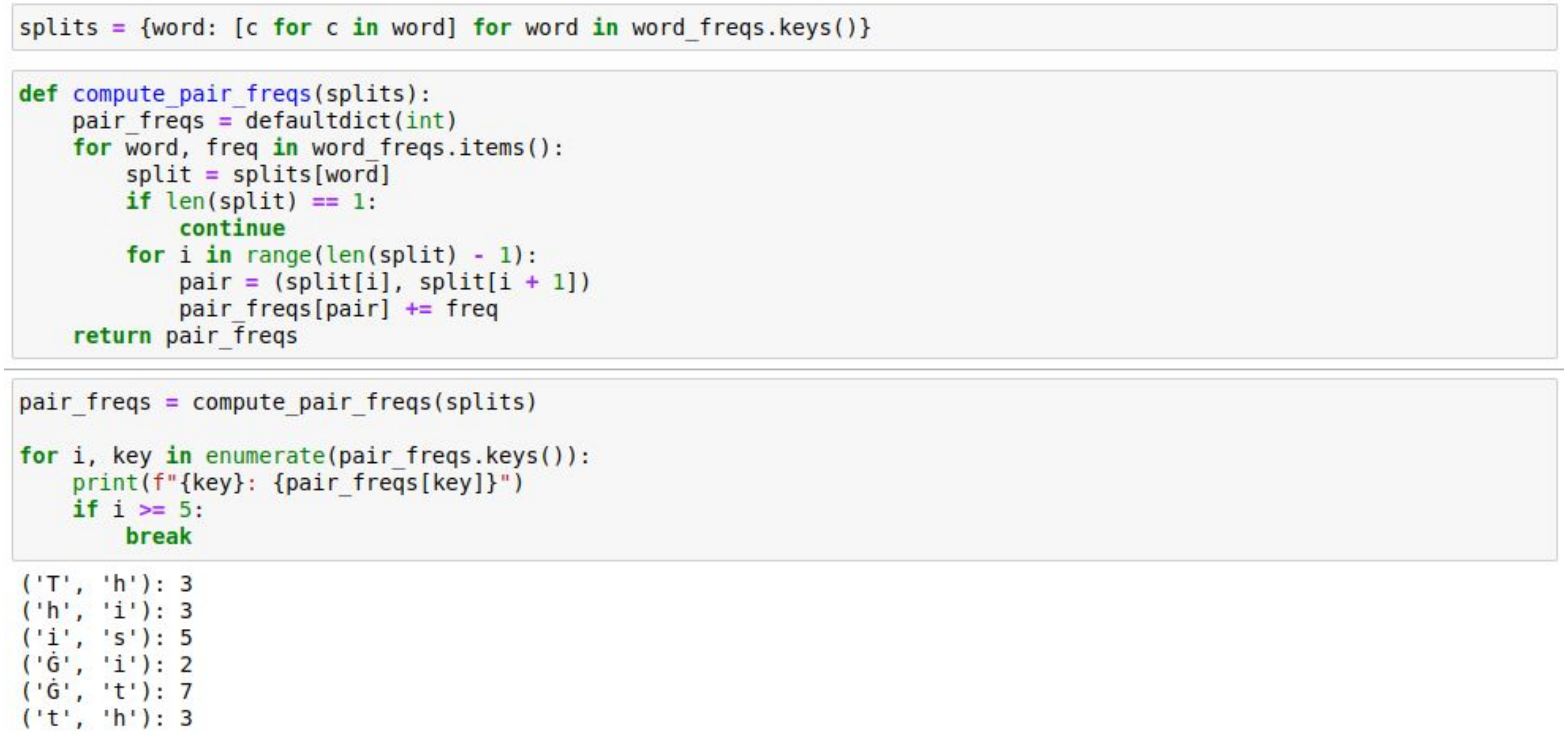
```python
alphabet = []

for word in word_freqs.keys():
    for letter in word:
        if letter not in alphabet:
            alphabet.append(letter)
alphabet.sort()

print(alphabet)
```

```
[',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'r',
's', 't', 'u', 'v', 'w', 'y', 'z', 'Ġ']
```

```python
vocab = ["<|endoftext|>"] + alphabet.copy()
```

“<|endoftext|>” is a special token

BPE training algorithm, step 2: Compute pair frequencies

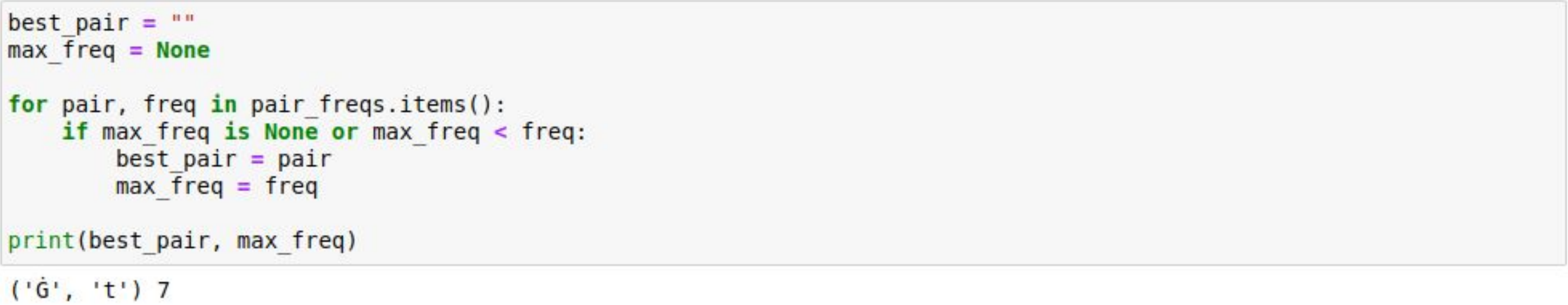


```
splits = {word: [c for c in word] for word in word_freqs.keys()}
```

```
def compute_pair_freqs(splits):
    pair_freqs = defaultdict(int)
    for word, freq in word_freqs.items():
        split = splits[word]
        if len(split) == 1:
            continue
        for i in range(len(split) - 1):
            pair = (split[i], split[i + 1])
            pair_freqs[pair] += freq
    return pair_freqs
```

```
pair_freqs = compute_pair_freqs(splits)

for i, key in enumerate(pair_freqs.keys()):
    print(f"{key}: {pair_freqs[key]}")
    if i >= 5:
        break
```

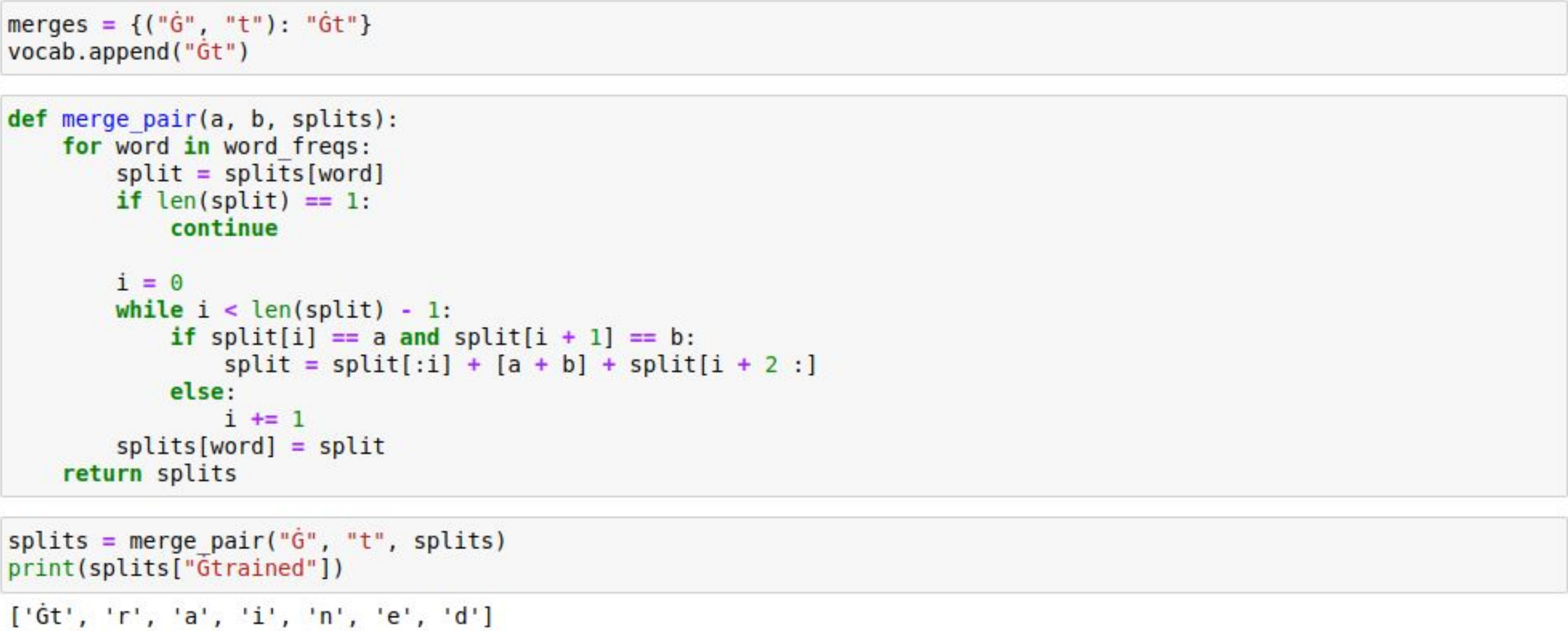
```
('T', 'h'): 3
('h', 'i'): 3
('i', 's'): 5
('Ġ', 'i'): 2
('Ġ', 't'): 7
('t', 'h'): 3
```



```
best_pair = ""
max_freq = None

for pair, freq in pair_freqs.items():
    if max_freq is None or max_freq < freq:
        best_pair = pair
        max_freq = freq

print(best_pair, max_freq)
```

```
('Ġ', 't') 7
```

BPE training algorithm, step 3: Add a merge rule



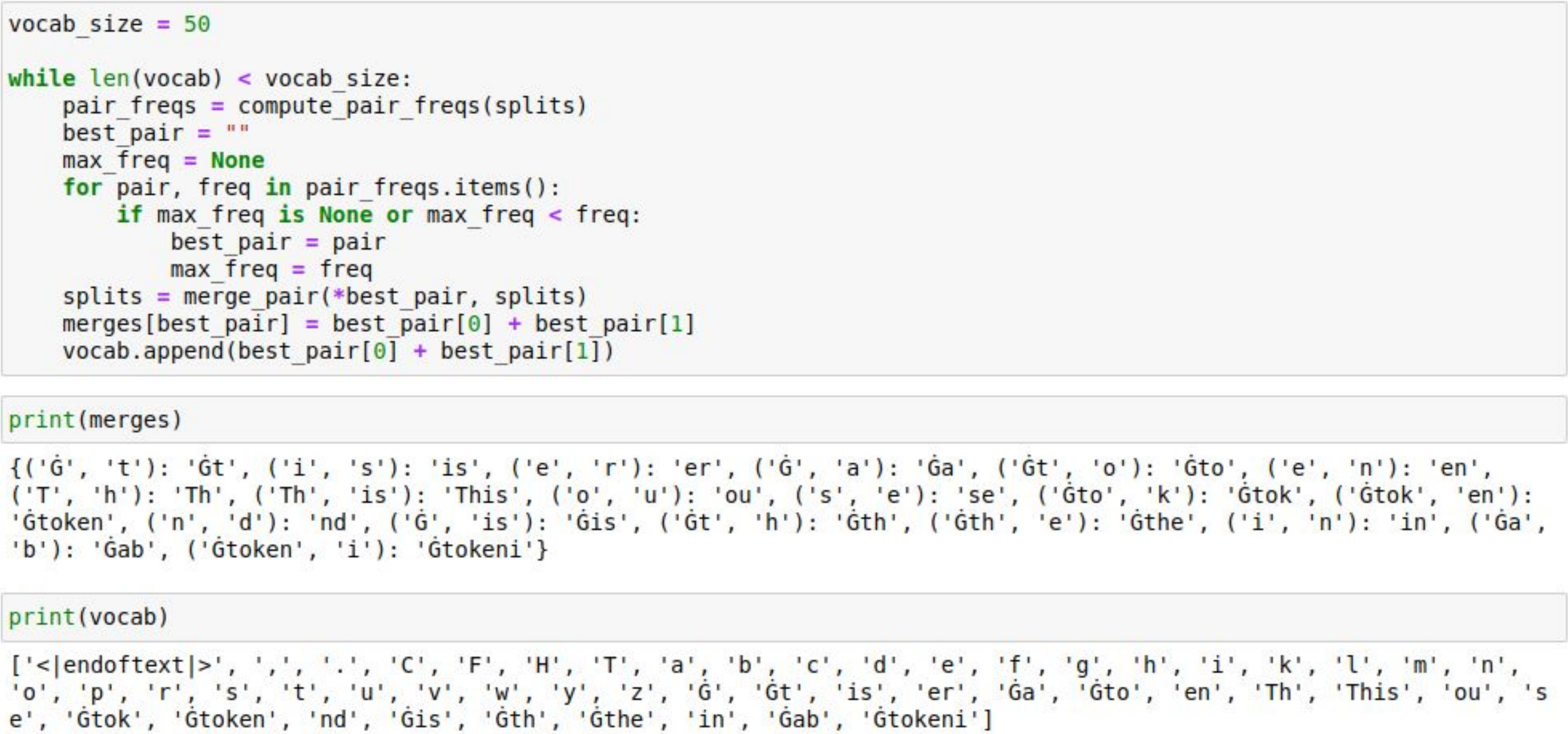
```python
merges = {("Ġ", "t"): "Ġt"}
vocab.append("Ġt")
```

```python
def merge_pair(a, b, splits):
    for word in word_freqs:
        split = splits[word]
        if len(split) == 1:
            continue

        i = 0
        while i < len(split) - 1:
            if split[i] == a and split[i + 1] == b:
                split = split[:i] + [a + b] + split[i + 2 :]
            else:
                i += 1
        splits[word] = split
    return splits
```

```python
splits = merge_pair("Ġ", "t", splits)
print(splits["Ġtrained"])
```

```
['Ġt', 'r', 'a', 'i', 'n', 'e', 'd']
```

BPE training algorithm: the loop



```
vocab_size = 50

while len(vocab) < vocab_size:
    pair_freqs = compute_pair_freqs(splits)
    best_pair = ""
    max_freq = None
    for pair, freq in pair_freqs.items():
        if max_freq is None or max_freq < freq:
            best_pair = pair
            max_freq = freq
    splits = merge_pair(*best_pair, splits)
    merges[best_pair] = best_pair[0] + best_pair[1]
    vocab.append(best_pair[0] + best_pair[1])
```

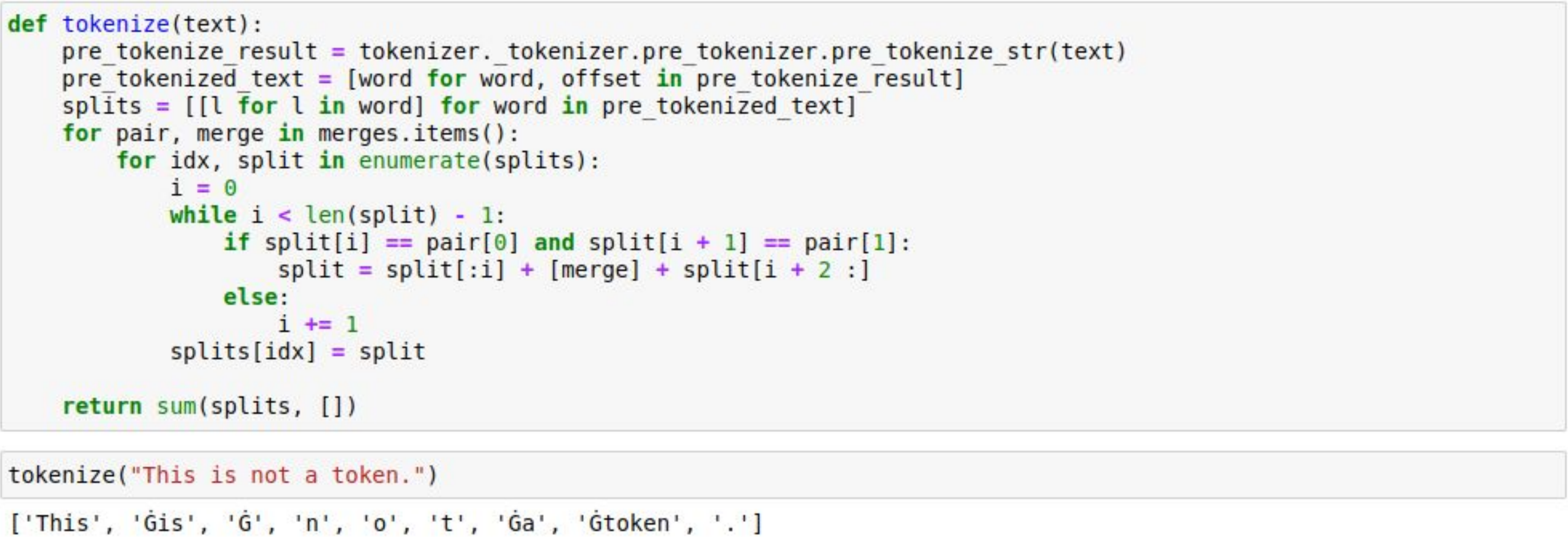
```
print(merges)
```

```
{('Ġ', 't'): 'Ġt', ('i', 's'): 'is', ('e', 'r'): 'er', ('Ġ', 'a'): 'Ġa', ('Ġt', 'o'): 'Ġto', ('e', 'n'): 'en',
('T', 'h'): 'Th', ('Th', 'is'): 'This', ('o', 'u'): 'ou', ('s', 'e'): 'se', ('Ġto', 'k'): 'Ġtok', ('Ġtok', 'en'):
'Ġtoken', ('n', 'd'): 'nd', ('Ġ', 'is'): 'Ġis', ('Ġt', 'h'): 'Ġth', ('Ġth', 'e'): 'Ġthe', ('i', 'n'): 'in', ('Ġa',
'b'): 'Ġab', ('Ġtoken', 'i'): 'Ġtokeni'}
```

```
print(vocab)
```

```
['<|endoftext|>', ',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n',
'o', 'p', 'r', 's', 't', 'u', 'v', 'w', 'y', 'z', 'Ġ', 'Ġt', 'is', 'er', 'Ġa', 'Ġto', 'en', 'Th', 'This', 'ou', 's
e', 'Ġtok', 'Ġtoken', 'nd', 'Ġis', 'Ġth', 'Ġthe', 'in', 'Ġab', 'Ġtokeni']
```

BPE Tokenization algorithm



```python
def tokenize(text):
    pre_tokenize_result = tokenizer._tokenizer.pre_tokenizer.pre_tokenize_str(text)
    pre_tokenized_text = [word for word, offset in pre_tokenize_result]
    splits = [[l for l in word] for word in pre_tokenized_text]
    for pair, merge in merges.items():
        for idx, split in enumerate(splits):
            i = 0
            while i < len(split) - 1:
                if split[i] == pair[0] and split[i + 1] == pair[1]:
                    split = split[:i] + [merge] + split[i + 2 :]
                else:
                    i += 1
            splits[idx] = split

    return sum(splits, [])
```

```python
tokenize("This is not a token.")
```

```
['This', 'Ġis', 'Ġ', 'n', 'o', 't', 'Ġa', 'Ġtoken', '.']
```

BPE Tokenization algorithm can fail?

What happens if there's an unknown character? This code would fail…

In actual (byte-level) implementations, it cannot happen.

In practice

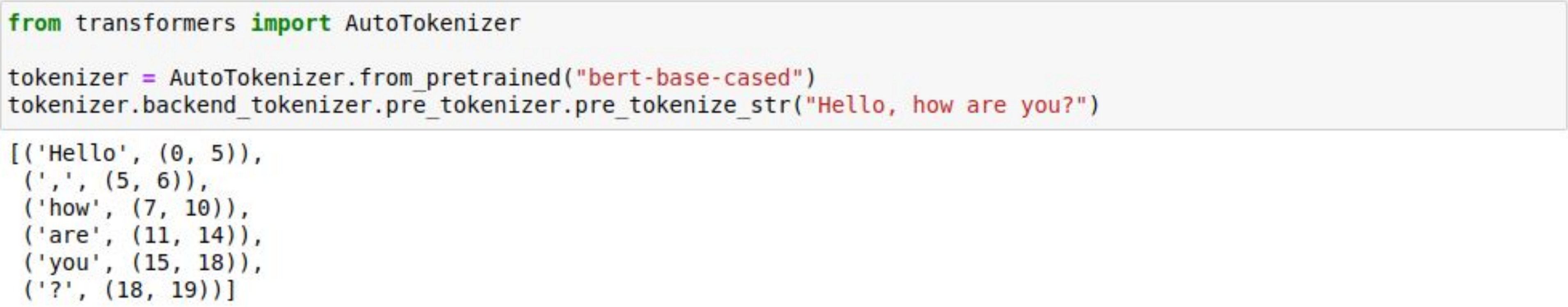
Tiktoken implements BPE:

https://github.com/openai/tiktoken

Wordpiece

Developed by Google for BERT (but never open sourced!)

The pre-tokenizer feels a lot more civilized:



```python
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str("Hello, how are you?")
```

```
[('Hello', (0, 5)),
 (',', (5, 6)),
 ('how', (7, 10)),
 ('are', (11, 14)),
 ('you', (15, 18)),
 ('?', (18, 19))]
```

Wordpiece in one slide

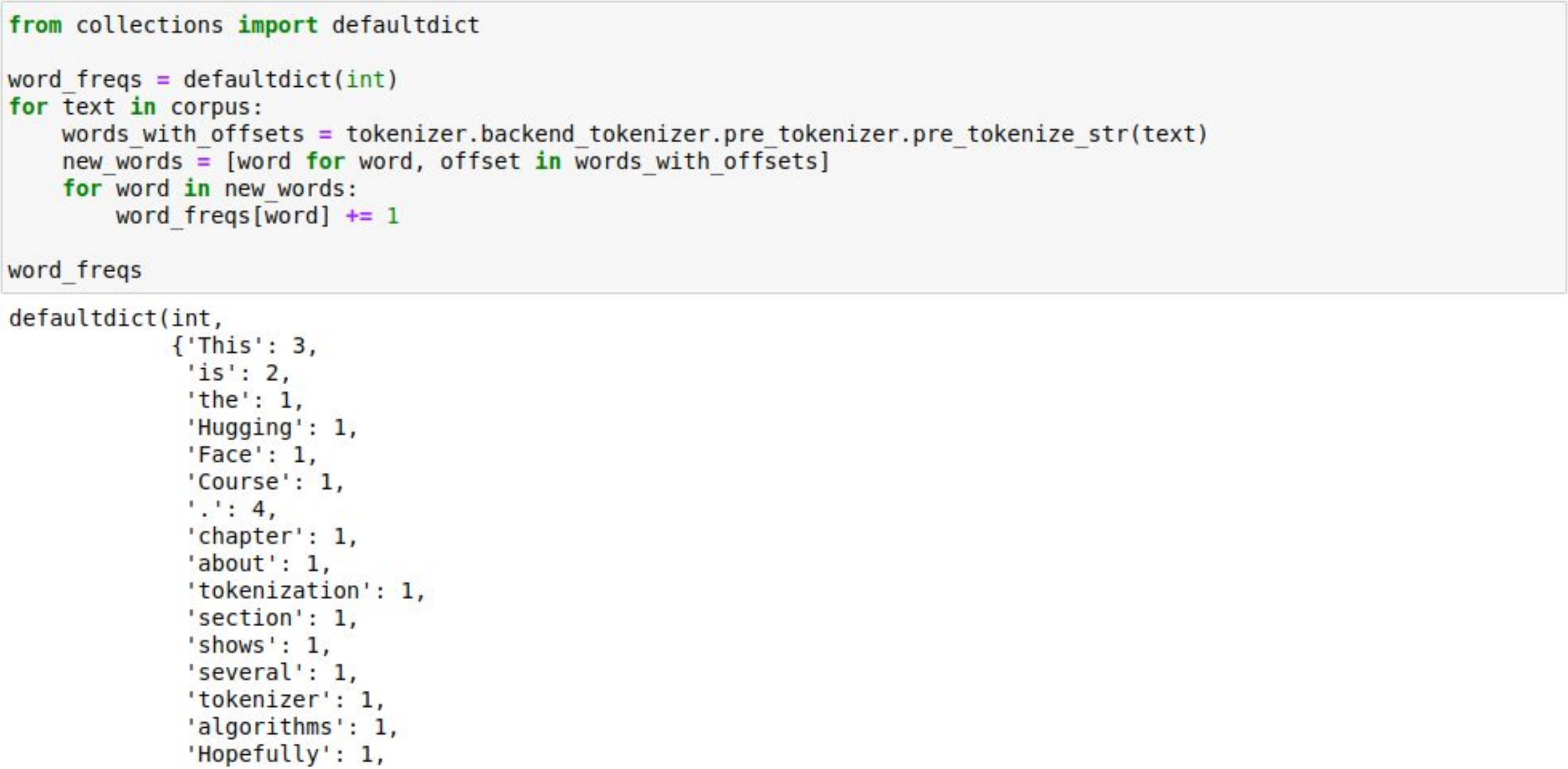
The goal is to learn merge rules, of the form:

(“Amer”, “ica”) -> “America”

Training: starting from characters, we create tokens by merging pairs with highest score, until we reach the budget number of tokens

Processing: to process an input text we look for the longest token and continue recursively (not using rules!)

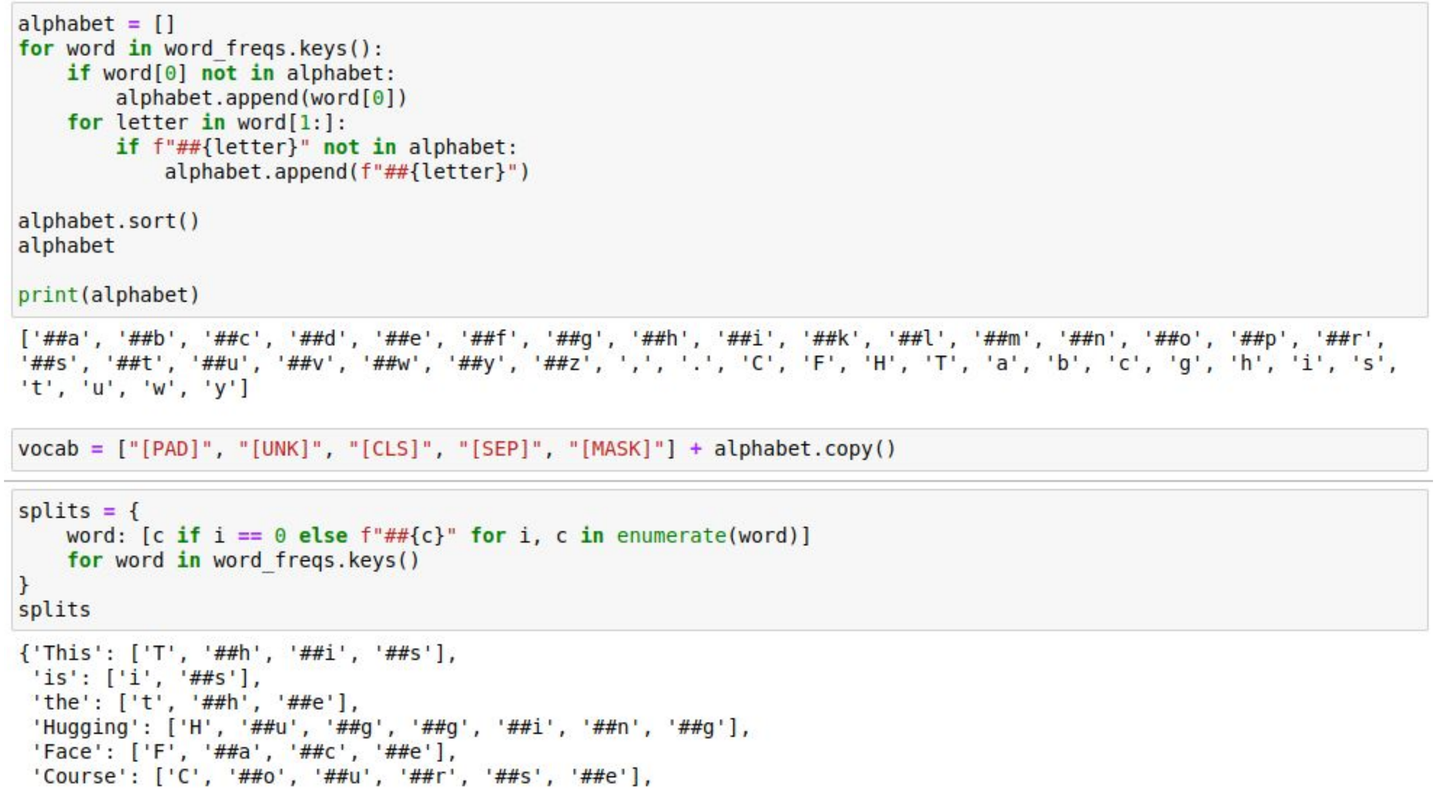
Wordpiece training algorithm, step 0: compute characters



```
from collections import defaultdict

word_freqs = defaultdict(int)
for text in corpus:
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    new_words = [word for word, offset in words_with_offsets]
    for word in new_words:
        word_freqs[word] += 1

word_freqs
```

```
defaultdict(int,
            {'This': 3,
             'is': 2,
             'the': 1,
             'Hugging': 1,
             'Face': 1,
             'Course': 1,
             '.': 4,
             'chapter': 1,
             'about': 1,
             'tokenization': 1,
             'section': 1,
             'shows': 1,
             'several': 1,
             'tokenizer': 1,
             'algorithms': 1,
             'Hopefully': 1,
```

Wordpiece training algorithm, step 1: compute frequencies



```
alphabet = []
for word in word_freqs.keys():
    if word[0] not in alphabet:
        alphabet.append(word[0])
    for letter in word[1:]:
        if f"##{letter}" not in alphabet:
            alphabet.append(f"##{letter}")

alphabet.sort()
alphabet

print(alphabet)
```

```
['##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '##i', '##k', '##l', '##m', '##n', '##o', '##p', '##r',
'##s', '##t', '##u', '##v', '##w', '##y', '##z', ',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'g', 'h', 'i', 's',
't', 'u', 'w', 'y']
```

```
vocab = ["[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]"] + alphabet.copy()
```

```
splits = {
    word: [c if i == 0 else f"##{c}" for i, c in enumerate(word)]
    for word in word_freqs.keys()
}
splits
```

```
{'This': ['T', '##h', '##i', '##s'],
 'is': ['i', '##s'],
 'the': ['t', '##h', '##e'],
 'Hugging': ['H', '##u', '##g', '##g', '##i', '##n', '##g'],
 'Face': ['F', '##a', '##c', '##e'],
 'Course': ['C', '##o', '##u', '##r', '##s', '##e'],
```

Wordpiece training algorithm, step 2: compute scores

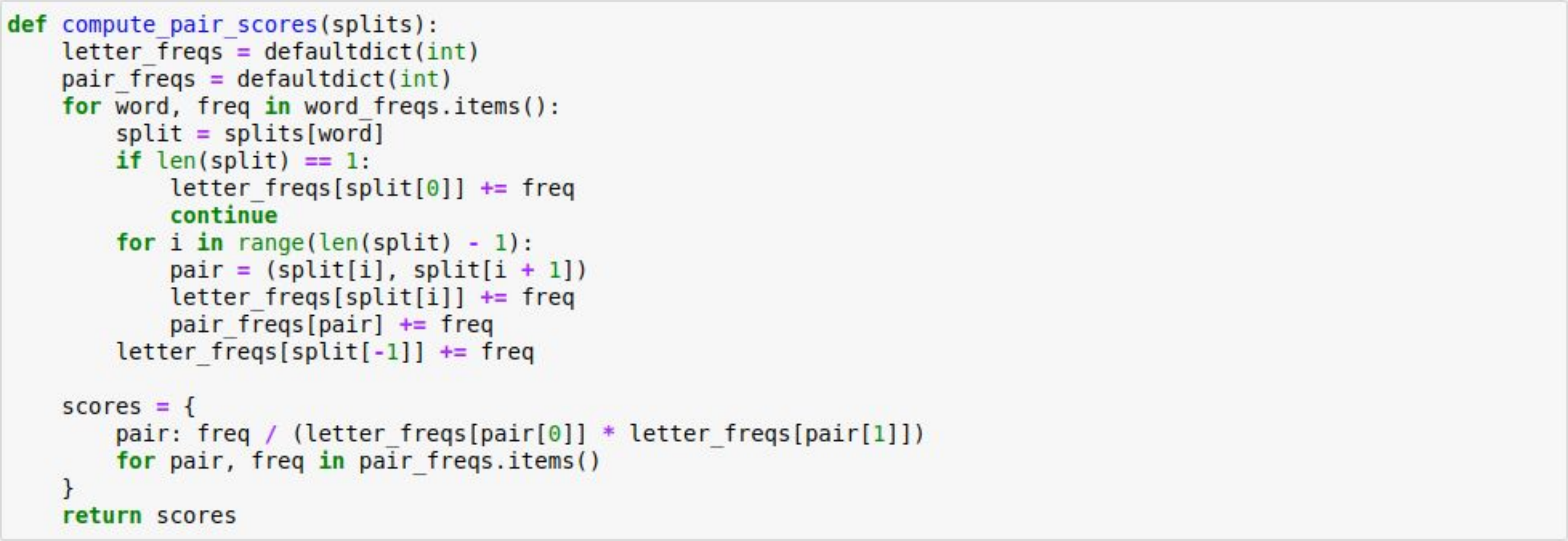
WordPiece computes a score for each pair, using the following formula:

freq_of_pair / (freq_of_first_element × freq_of_second_element)

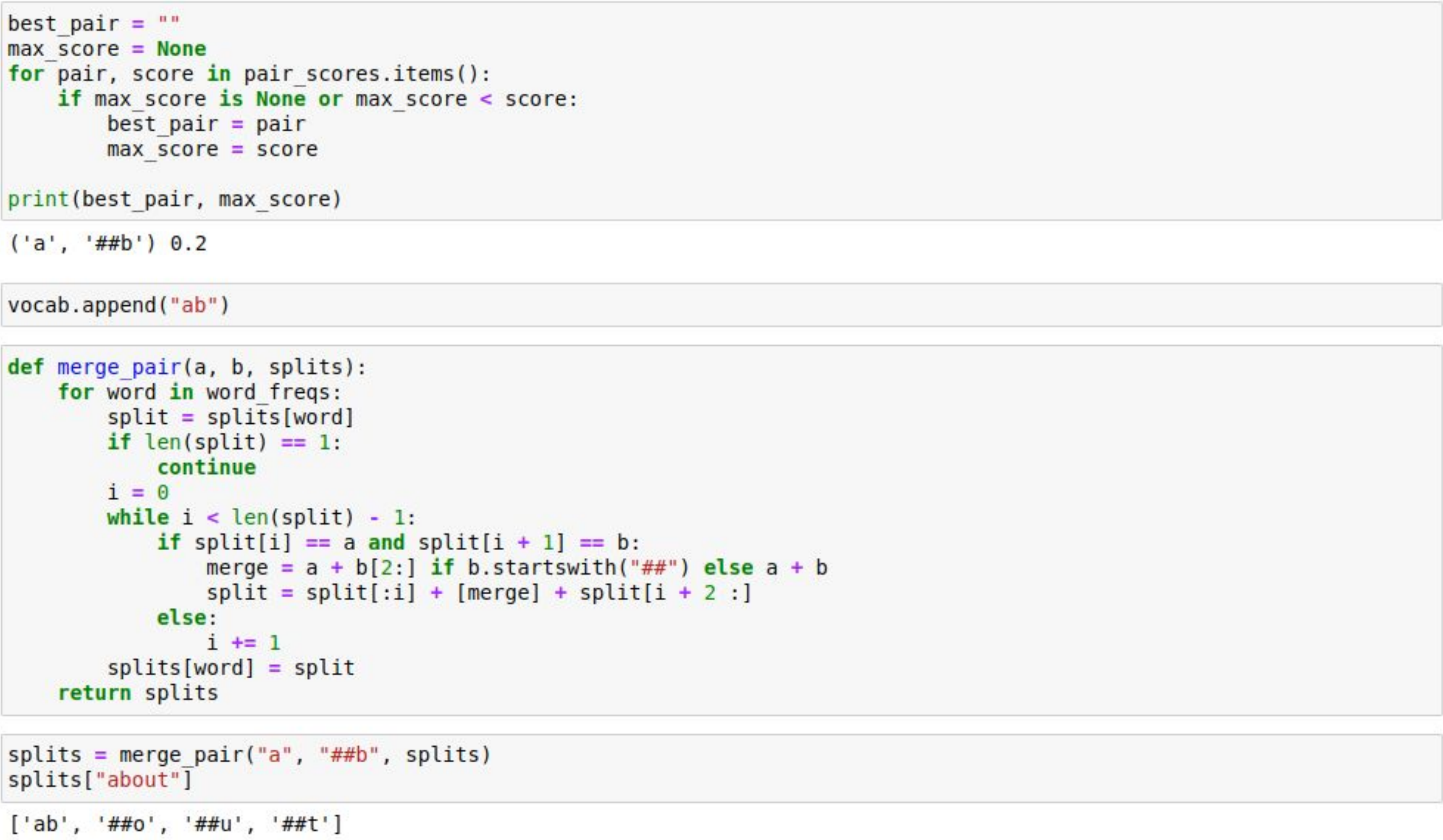
The algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary:

- It won’t necessarily merge `("un", "##able")` even if that pair occurs very frequently in the vocabulary, because the two pairs `"un"` and `"##able"` will likely each appear in a lot of other words and have a high frequency.
- In contrast, a pair like `("hu", "##gging")` will probably be merged faster (assuming the word “hugging” appears often in the vocabulary) since `"hu"` and `"##gging"` are likely to be less frequent individually.

Wordpiece training algorithm, step 2: compute scores



```python
def compute_pair_scores(splits):
    letter_freqs = defaultdict(int)
    pair_freqs = defaultdict(int)
    for word, freq in word_freqs.items():
        split = splits[word]
        if len(split) == 1:
            letter_freqs[split[0]] += freq
            continue
        for i in range(len(split) - 1):
            pair = (split[i], split[i + 1])
            letter_freqs[split[i]] += freq
            pair_freqs[pair] += freq
        letter_freqs[split[-1]] += freq

    scores = {
        pair: freq / (letter_freqs[pair[0]] * letter_freqs[pair[1]])
        for pair, freq in pair_freqs.items()
    }
    return scores
```



```
best_pair = ""
max_score = None
for pair, score in pair_scores.items():
    if max_score is None or max_score < score:
        best_pair = pair
        max_score = score

print(best_pair, max_score)
```

```
('a', '##b') 0.2
```

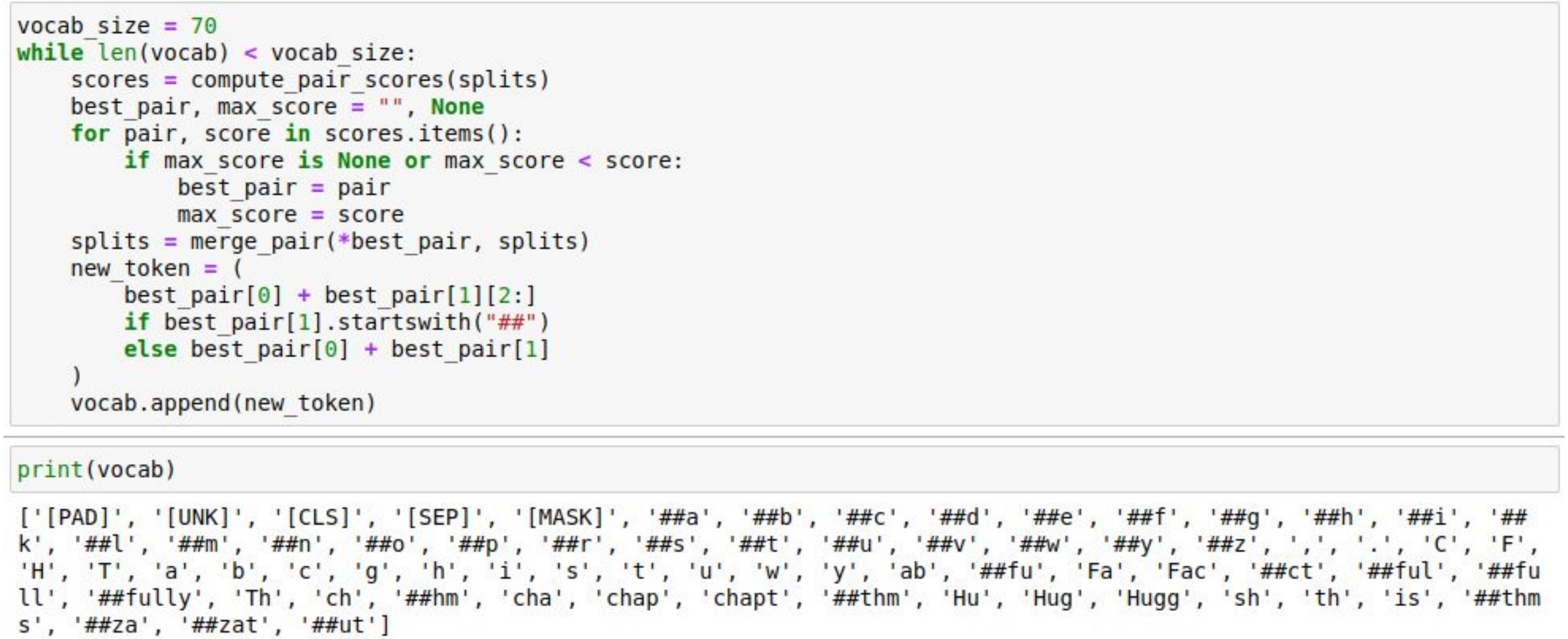
```
vocab.append("ab")
```

```
def merge_pair(a, b, splits):
    for word in word_freqs:
        split = splits[word]
        if len(split) == 1:
            continue
        i = 0
        while i < len(split) - 1:
            if split[i] == a and split[i + 1] == b:
                merge = a + b[2:] if b.startswith("##") else a + b
                split = split[:i] + [merge] + split[i + 2 :]
            else:
                i += 1
        splits[word] = split
    return splits
```

```
splits = merge_pair("a", "##b", splits)
splits["about"]
```

```
['ab', '##o', '##u', '##t']
```

Wordpiece training algorithm: the Loop

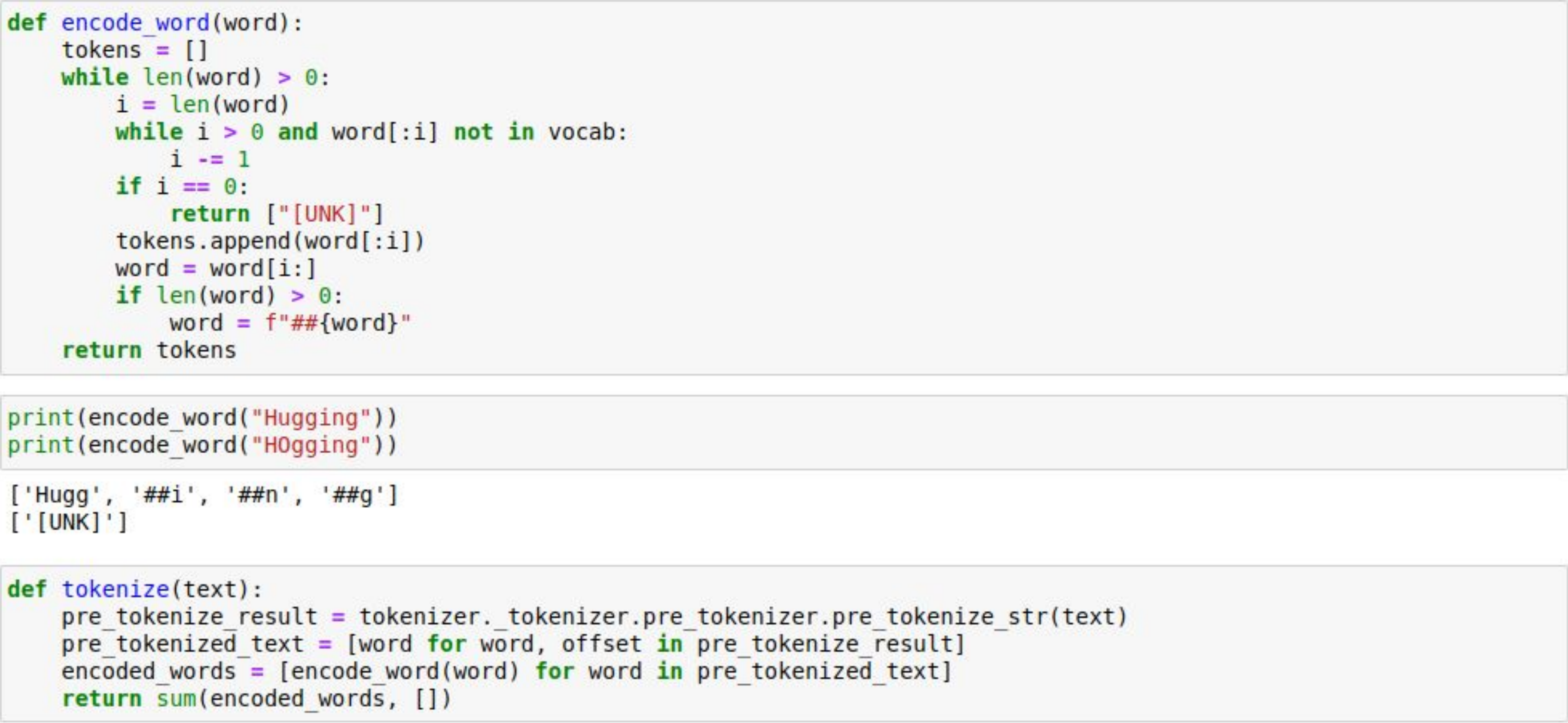


```python
vocab_size = 70
while len(vocab) < vocab_size:
    scores = compute_pair_scores(splits)
    best_pair, max_score = "", None
    for pair, score in scores.items():
        if max_score is None or max_score < score:
            best_pair = pair
            max_score = score
    splits = merge_pair(*best_pair, splits)
    new_token = (
        best_pair[0] + best_pair[1][2:]
        if best_pair[1].startswith("##")
        else best_pair[0] + best_pair[1]
    )
    vocab.append(new_token)
```

```python
print(vocab)
```

```
['[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]', '##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '##i', '##
k', '##l', '##m', '##n', '##o', '##p', '##r', '##s', '##t', '##u', '##v', '##w', '##y', '##z', ',', '.', 'C', 'F',
'H', 'T', 'a', 'b', 'c', 'g', 'h', 'i', 's', 't', 'u', 'w', 'y', 'ab', '##fu', 'Fa', 'Fac', '##ct', '##ful', '##fu
ll', '##fully', 'Th', 'ch', '##hm', 'cha', 'chap', 'chapt', '##thm', 'Hu', 'Hug', 'Hugg', 'sh', 'th', 'is', '##thm
s', '##za', '##zat', '##ut']
```

Wordpiece tokenization algorithm



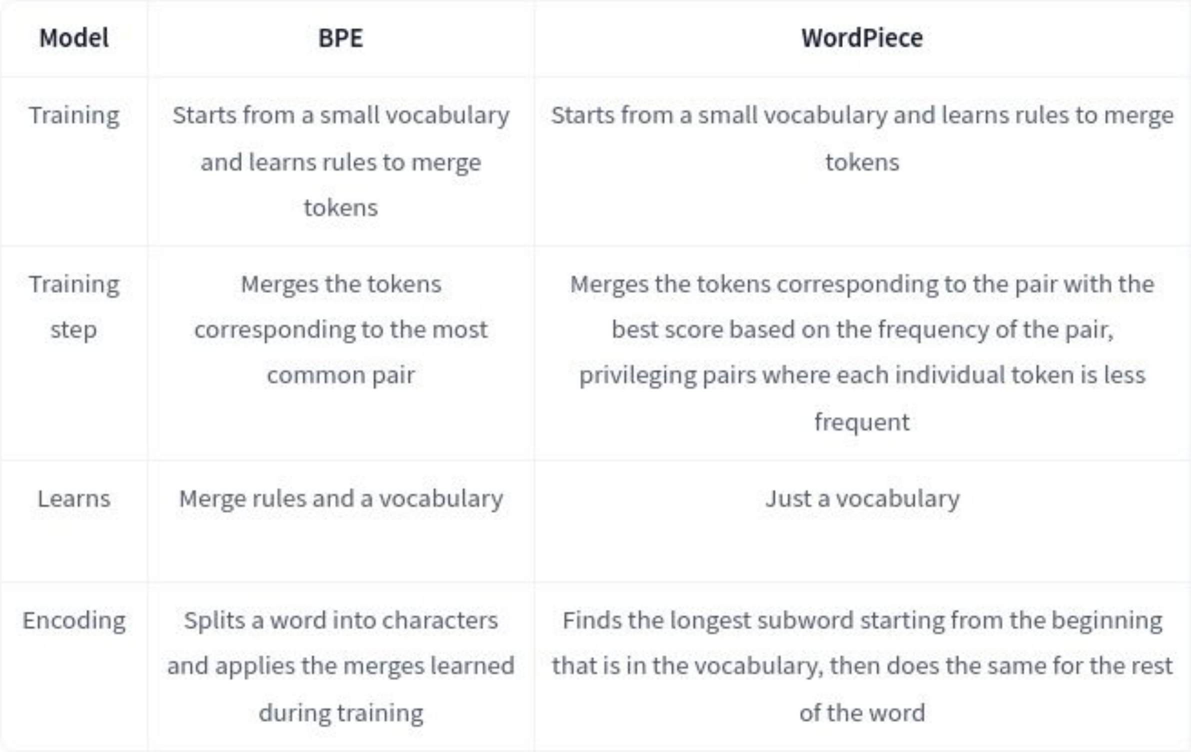
```python
def encode_word(word):
    tokens = []
    while len(word) > 0:
        i = len(word)
        while i > 0 and word[:i] not in vocab:
            i -= 1
        if i == 0:
            return ["[UNK]"]
        tokens.append(word[:i])
        word = word[i:]
        if len(word) > 0:
            word = f"##{word}"
    return tokens
```

```python
print(encode_word("Hugging"))
print(encode_word("HOgging"))
```

```
['Hugg', '##i', '##n', '##g']
['[UNK]']
```

```python
def tokenize(text):
    pre_tokenize_result = tokenizer._tokenizer.pre_tokenizer.pre_tokenize_str(text)
    pre_tokenized_text = [word for word, offset in pre_tokenize_result]
    encoded_words = [encode_word(word) for word in pre_tokenized_text]
    return sum(encoded_words, [])
```

Summary for the two algorithms



Model	BPE	WordPiece
Training	Starts from a small vocabulary and learns rules to merge tokens	Starts from a small vocabulary and learns rules to merge tokens
Training step	Merges the tokens corresponding to the most common pair	Merges the tokens corresponding to the pair with the best score based on the frequency of the pair, privileging pairs where each individual token is less frequent
Learns	Merge rules and a vocabulary	Just a vocabulary
Encoding	Splits a word into characters and applies the merges learned during training	Finds the longest subword starting from the beginning that is in the vocabulary, then does the same for the rest of the word

Short practical session:
Train a tokenizer on code

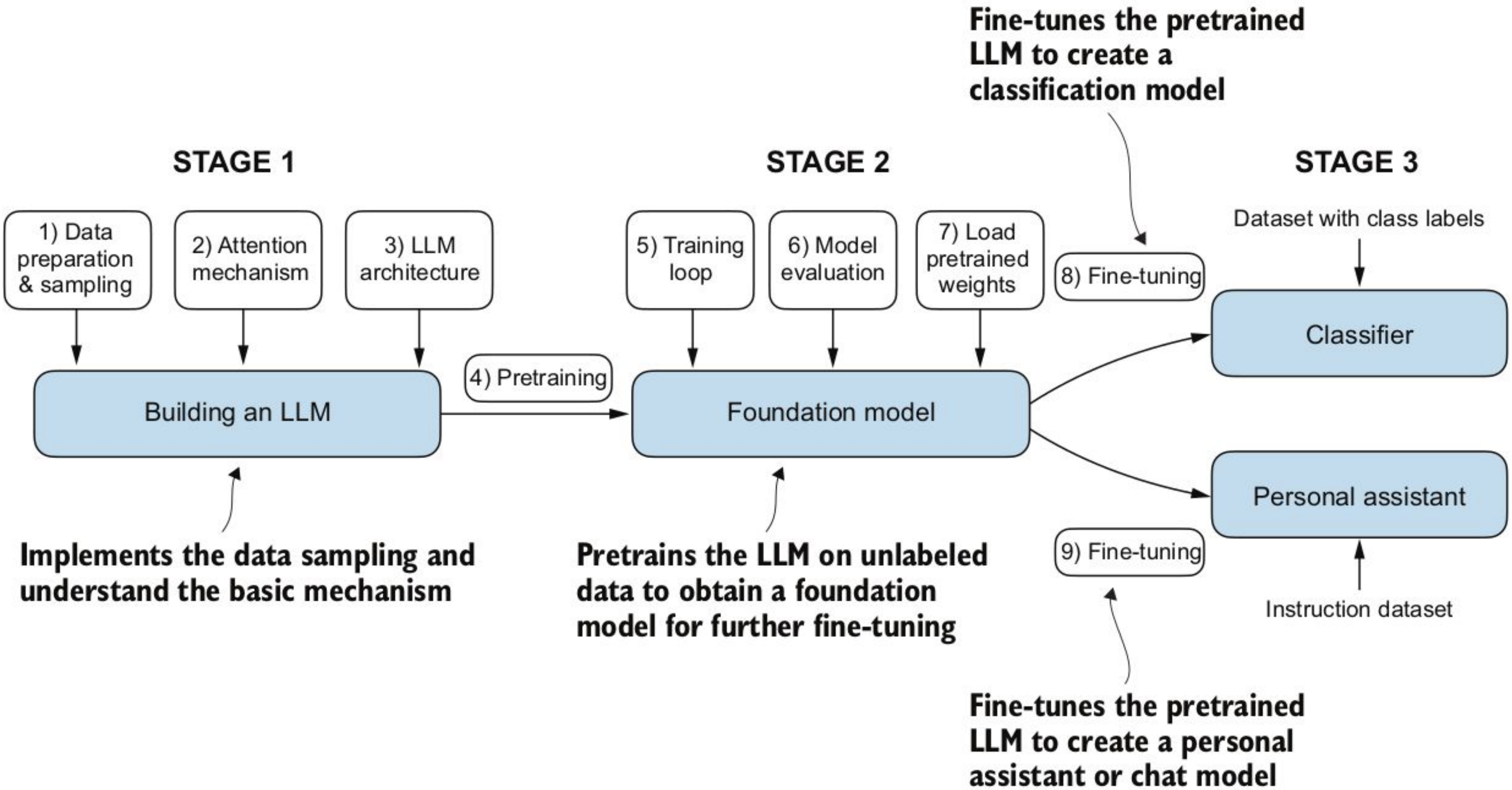
Fine-tuning

- General overview
- LoRA

FOUNDATION MODELS

Language Models are not very useful, they randomly generate texts… But this means that they somehow capture some information from natural language! They are also called *foundation models*.

Fine-tuning is about making Language Models solve concrete tasks, like classification, question answering, name entity recognition…



Fine-tunes the pretrained LLM to create a classification model

Implements the data sampling and understand the basic mechanism

Pretrains the LLM on unlabeled data to obtain a foundation model for further fine-tuning

Fine-tunes the pretrained LLM to create a personal assistant or chat model

Training is expensive

We often cannot afford updating the ***whole*** model!

Most of us will not train foundation models… Rather fine-tune existing ones.

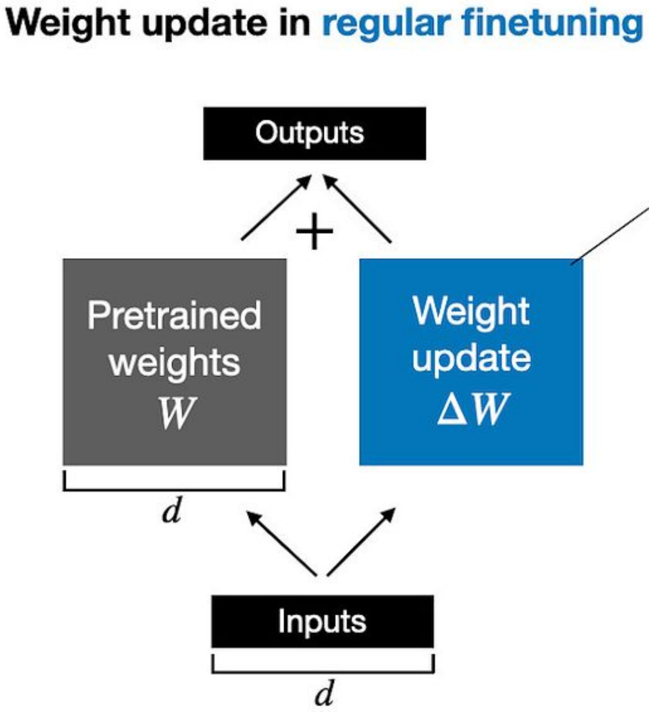
Low-rank Adaptation (LoRA)

Two key ideas:

(1) We only store the changes, not a new model
(2) We only update a small number of parameters

Idea: storing weight updates

Say we consider a linear layer with matrix W. We keep the matrix W fixed and store ΔW

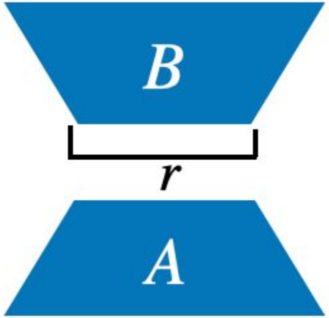


Rank approximations

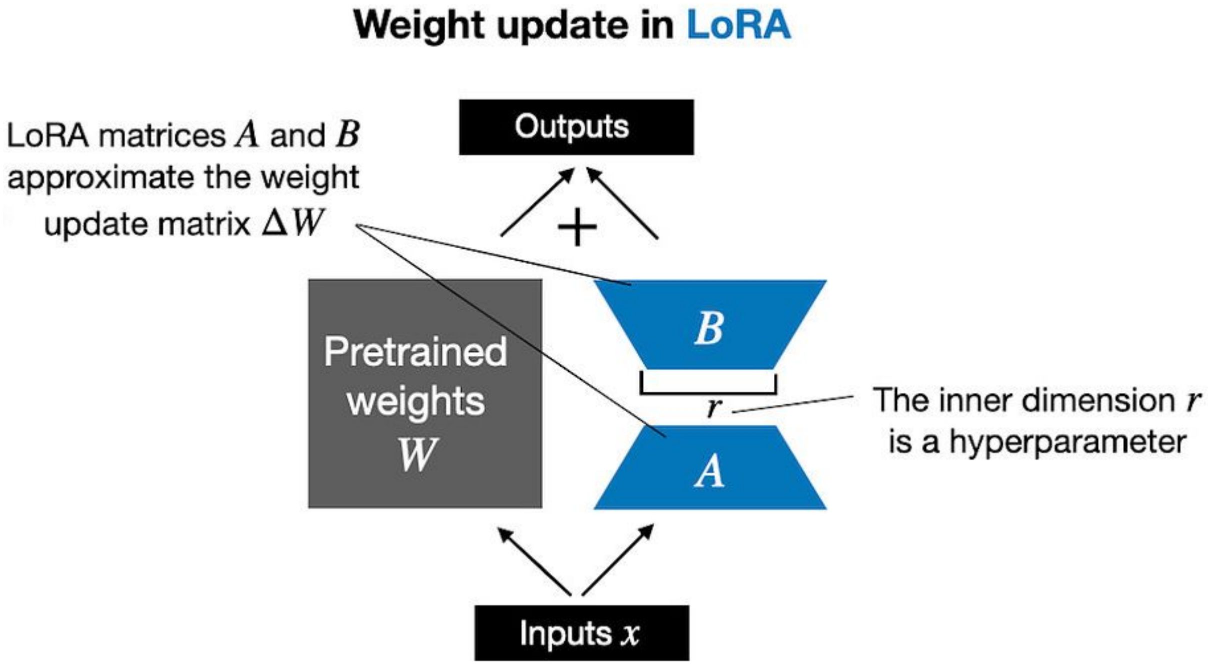
A matrix W of dimension dxd contains dxd parameters. It can be ***rank-r approximated*** by two matrices AxB with:

- A of dimension dxr
- B of dimension rxd

Instead of dxd parameters we now have 2xdxr parameters.

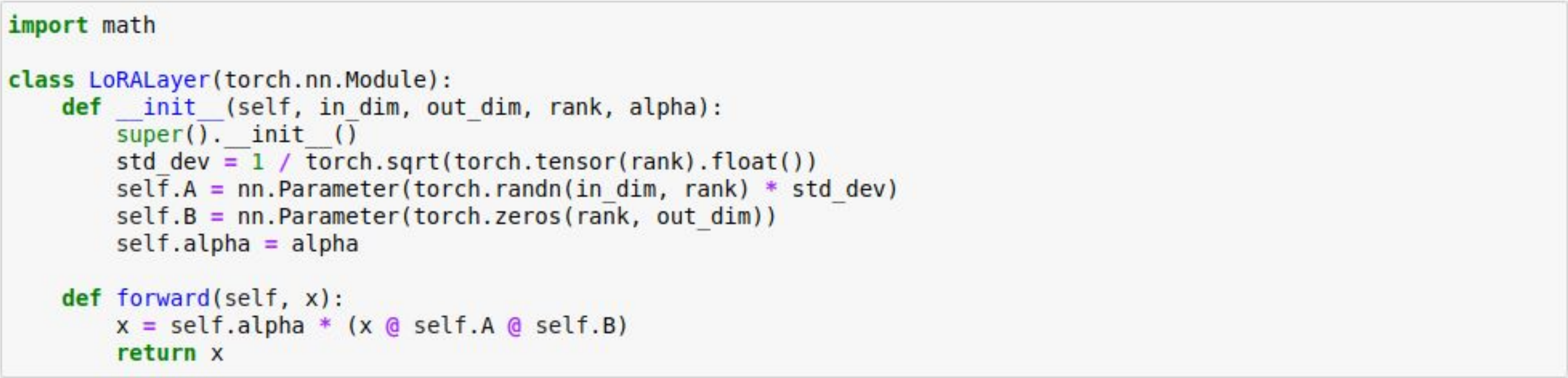


Weight update



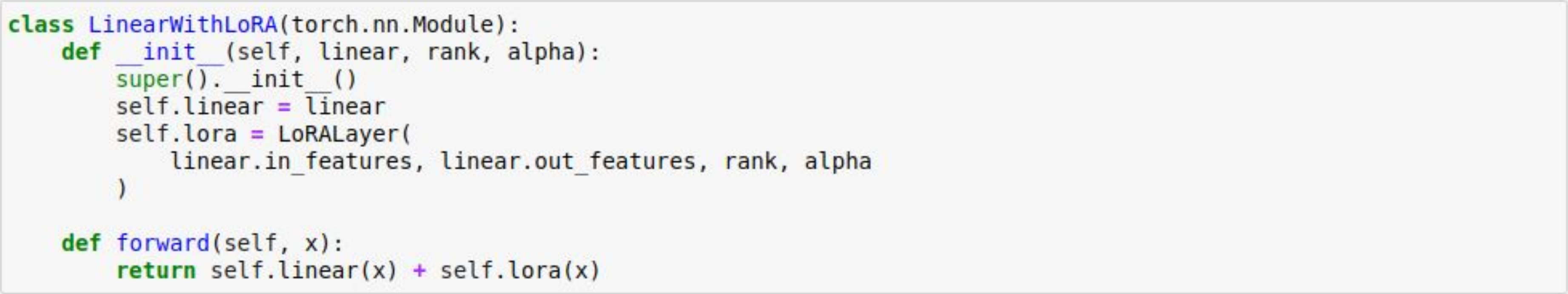
Weight update in LoRA
Outputs
LoRA matrices A and B approximate the weight update matrix ΔW
+
Pretrained weights W
B
r
The inner dimension r is a hyperparameter
A
Inputs x

LoRA Layer



```python
import math

class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        std_dev = 1 / torch.sqrt(torch.tensor(rank).float())
        self.A = nn.Parameter(torch.randn(in_dim, rank) * std_dev)
        self.B = nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

Adding the LoRA layer



```python
class LinearWithLoRA(torch.nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(
            linear.in_features, linear.out_features, rank, alpha
        )

    def forward(self, x):
        return self.linear(x) + self.lora(x)
```

```python
def replace_linear_with_lora(model, rank, alpha):
    for name, module in model.named_children():
        if isinstance(module, torch.nn.Linear):
            # Replace the Linear layer with LinearWithLoRA
            setattr(model, name, LinearWithLoRA(module, rank, alpha))
        else:
            # Recursively apply the same function to child modules
            replace_linear_with_lora(module, rank, alpha)
```

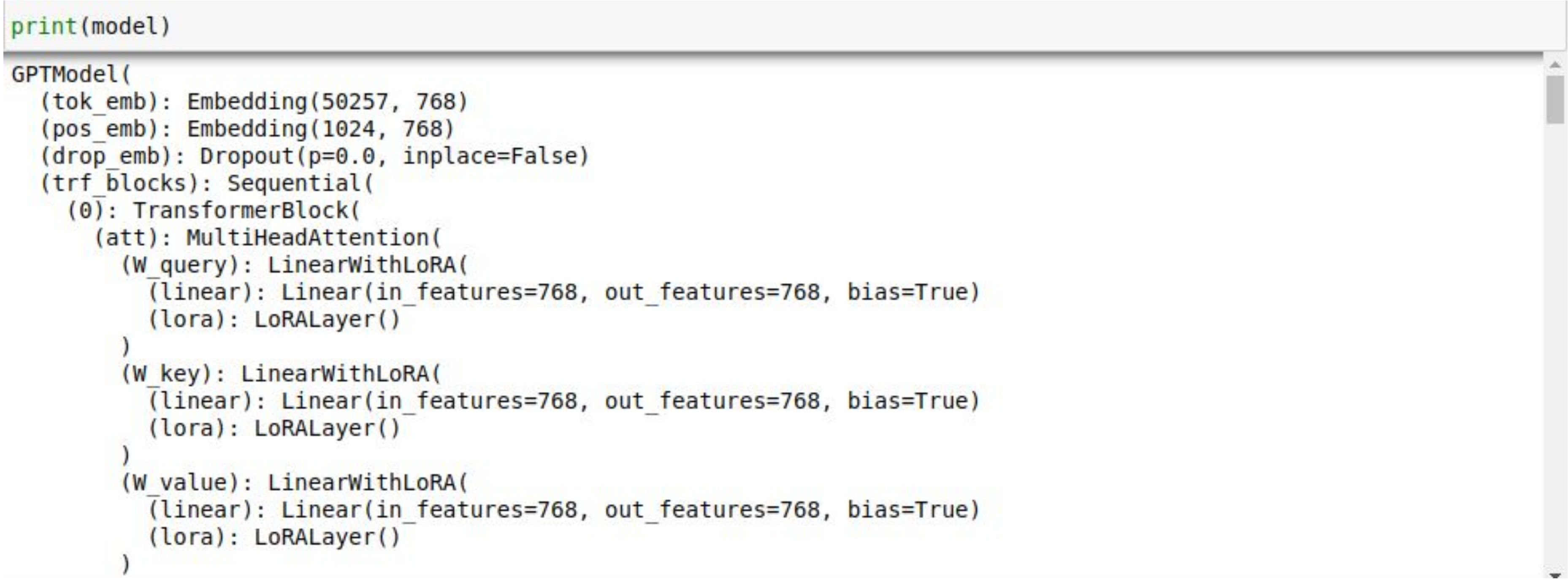
- We then freeze the original model parameter and use the `replace_linear_with_lora` to replace the said `Linear` layers using the code below
- This will replace the `Linear` layers in the LLM with `LinearWithLoRA` layers

```python
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")

for param in model.parameters():
    param.requires_grad = False

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
```

```
Total trainable parameters before: 124,441,346
Total trainable parameters after: 0
```

```python
replace_linear_with_lora(model, rank=16, alpha=16)

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
```

```
Total trainable LoRA parameters: 2,666,528
```



```
print(model)
```

```
GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    (0): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_key): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_value): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
```

Short project:
Teach arithmetic to an LLM

Step 1: construct a tokenizer for arithmetic operations

Step 2: train an LLM on generated arithmetic tasks

Step 3: profit!