CARTE-Enbridge Bootcamp

Lab 5-0

Building a GPT model from scratch

In this notebook, we are going to build a very simple version of GPT. Our GPT will have a small vocabulary and a small number of layers. Let's begin by importing the necessary libraries.

In [1]:
```
import tensorflow as tf
from tensorflow import keras
import numpy as np
```

```
2023-11-13 09:50:52.777897: I tensorflow/core/util/port.cc:110] oneDNN custo
m operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn t
hem off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-11-13 09:50:52.806822: I tensorflow/core/platform/cpu_feature_guard.cc:
182] This TensorFlow binary is optimized to use available CPU instructions i
n performance-critical operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 AVX512F AVX512_
VNNI FMA, in other operations, rebuild TensorFlow with the appropriate compi
ler flags.
```

In [2]:
```
# Check that we are using a GPU
if tf.test.gpu_device_name():
    print('Default GPU Device: {}'.format(tf.test.gpu_device_name()))
else:
    raise SystemError('GPU device not found! Enable GPU by going to Runtime
```

```
Default GPU Device: /device:GPU:0
```

```
2023-11-13 09:50:54.012523: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.031040: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.031203: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.087398: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.087546: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.087631: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.087704: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device /device:GPU:0 with 10392 MB memory:  -> device: 0, name: NVIDIA GeForce RTX 3060, pci bus id: 0000:02:00.0, compute capability: 8.6
2023-11-13 09:50:54.088448: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.088536: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.088610: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.088700: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.088776: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
```

```
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:50:54.088827: I tensorflow/core/common_runtime/gpu/gpu_device.
cc:1639] Created device /device:GPU:0 with 10392 MB memory:  -> device: 0, n
ame: NVIDIA GeForce RTX 3060, pci bus id: 0000:02:00.0, compute capability:
8.6
```

At its core, GPT is a model to predict the next word in a sequence. In order to be able to learn, we need to first convert words into values that can be fed into the model.

We are going to load a dataset of samples from Simple English Wikipedia. This is a version of Wikipedia that aims to cover the same content, but using a reduced vocabulary and simpler grammar. This makes it easier for language learners to understand. We will use this dataset to train our model.

In [3]:
```python
import requests
import zipfile
from tqdm import tqdm

url = "https://raw.githubusercontent.com/alexwolson/carte_workshop_datasets/

# Stream the download so we can track its progress
response = requests.get(url, stream=True)

# Total size in bytes.
total_size = int(response.headers.get('content-length', 0))
block_size = 1024  # 1KB
progress_bar = tqdm(total=total_size, unit='iB', unit_scale=True)

with open('corpus.txt.zip', 'wb') as file:
    for data in response.iter_content(block_size):
        progress_bar.update(len(data))
        file.write(data)
progress_bar.close()

if total_size != 0 and progress_bar.n != total_size:
    print("ERROR, something went wrong")

# Now we will extract the zip file
z = zipfile.ZipFile('corpus.txt.zip')
z.extractall()

# Read the first 1000 characters from the corpus
with open('corpus.txt', 'r') as f:
    corpus = f.read()

print(corpus[:1000])
```

```
100%|
|
|
| 11.9M/11.9M [00:00<00:00, 42.8MiB/s]
```

```
April
April is the fourth month of the year with 30 days. The name April comes fro
m that Latin word "aperire" which means "to open". This probably refers to g
rowing plants in spring. April begins on the same day of week as "July" in a
ll years and also "January" in leap years.
April's flower is the Sweet Pea and its birthstone is the Diamond. The meani
ng of the Diamond is Innocence.
April in poetry.
Poets use "April" to mean the end of winter. For example: "April showers bri
ng May flowers."

August
August is the eighth month of the year. It has 31 days.
This month was first called "Sextilis" in Latin, because it was the sixth mo
nth in the old Roman calendar. The Roman calendar began in March about 735 B
C with Romulus. It was the eighth month when January or February were added
to the start of the year by King Numa Pompilius about 700 BC. Or, when those
two months were moved from the end to the beginning of the year by the decem
virs about 450 BC (Roman writers disagree).
August is named
```

Because we want our model to be very simple, we are going to determine the 500 most common words, and use those as our vocabulary.

In [4]:

```
from collections import Counter
from re import sub

# Strip out all punctuation and numbers
corpus = sub(r'[^\w\s]', '', corpus)
corpus = sub('\n', ' ', corpus)
corpus = sub(r'\d+', '', corpus)
corpus = sub('  ', ' ', corpus)

words = corpus.lower().split(' ')
word_counts = Counter(words)

vocab_size = 500
most_common_words = word_counts.most_common(vocab_size+1)
most_common_words = [word for word, count in most_common_words if word != ''
print(most_common_words)
```

['the', 'of', 'in', 'a', 'and', 'is', 'to', 'was', 'it', 'that', 'for', 'ar
e', 'as', 'he', 'on', 'by', 'with', 'or', 'from', 'they', 'an', 'this', 'a
t', 'be', 'his', 'people', 'also', 'has', 'not', 'were', 'which', 'have', 'o
ne', 'river', 'but', 'can', 'many', 'called', 'other', 'there', 'city', 'the
ir', 'when', 'first', 'who', 'some', 'used', 'its', 'about', 'had', 'most',
'found', 'into', 'after', 'made', 'united', 'very', 'states', 'she', 'more',
'all', 'time', 'because', 'two', 'france', 'new', 'like', 'part', 'her', 'be
en', 'music', 'region', 'only', 'world', 'known', 'these', 'means', 'north',
'name', 'commune', 'them', 'than', 'became', 'may', 'years', 'such', 'ofte
n', 'so', 'up', 'different', 'where', 'department', 'born', 'during', 's',
'between', 'over', 'if', 'him', 'then', 'th', 'use', 'will', 'make', 'usuall
y', 'war', 'out', 'do', 'state', 'south', 'would', 'american', 'later', 'are
a', 'no', 'famous', 'each', 'same', 'before', 'small', 'year', 'three', 'eas
t', 'english', 'number', 'located', 'sometimes', 'romania', 'important', 'wo
n', 'district', 'well', 'town', 'around', 'work', 'country', 'main', 'bein
g', 'way', 'person', 'life', 'century', 'named', 'language', 'now', 'populat
ion', 'history', 'what', 'group', 'did', 'county', 'government', 'live', 'co
untries', 'second', 'started', 'system', 'flows', 'through', 'west', 'long',
'another', 'british', 'things', 'de', 'both', 'large', 'until', 'since', 'ex
ample', 'national', 'while', 'word', 'could', 'much', 'however', 'water', 't
ributary', 'series', 'place', 'family', 'band', 'played', 'any', 'province',
'said', 'old', 'capital', 'movie', 'how', 'germany', 'several', 'game', 'alb
um', 'popular', 'get', 'died', 'football', 'season', 'even', 'early', 'kin
g', 'still', 'good', 'england', 'day', 'released', 'iowa', 'wrote', 'germa
n', 'great', 'book', 'back', 'team', 'under', 'show', 'i', 'asteroid', 'sept
ember', 'including', 'written', 'us', 'near', 'high', 'school', 'form', 'fou
r', 'end', 'see', 'best', 'church', 'president', 'la', 'big', 'against', 'un
iversity', 'together', 'french', 'power', 'become', 'just', 'league', 'buil
t', 'times', 'lot', 'common', 'championship', 'island', 'august', 'parts',
'today', 'body', 'does', 'own', 'play', 'go', 'came', 'song', 'using', 'crea
ted', 'games', 'october', 'last', 'john', 'went', 'television', 'began', 'pl
ayer', 'america', 'largest', 'death', 'you', 'money', 'given', 'march', 'w
e', 'tropical', 'london', 'sea', 'left', 'europe', 'rock', 'pakistan', 'indi
a', 'party', 'home', 'land', 'based', 'teams', 'took', 'thought', 'hurrican
e', 'million', 'kingdom', 'man', 'include', 'january', 'down', 'july', 'agai
n', 'help', 'every', 'children', 'those', 'november', 'major', 'municipalit
y', 'god', 'modern', 'st', 'songs', 'december', 'company', 'members', 'south
ern', 'although', 'union', 'storm', 'type', 'story', 'april', 'june', 'makin
g', 'house', 'food', 'few', 'comes', 'ii', 'take', 'western', 'period', 'mak
es', 'special', 'animals', 'computer', 'performed', 'set', 'something', 'rom
an', 'greek', 'republic', 'white', 'moved', 'words', 'father', 'term', 'cent
ral', 'human', 'young', 'living', 'put', 'line', 'japan', 'international',
'law', 'northwest', 'similar', 'northern', 'say', 'without', 'next', 'austra
lia', 'order', 'red', 'general', 'others', 'age', 'works', 'come', 'lived',
'black', 'species', 'got', 'off', 'empire', 'february', 'should', 'switzerla
nd', 'change', 'member', 'light', 'third', 'away', 'days', 'single', 'york',
'men', 'along', 'list', 'energy', 'changed', 'books', 'video', 'five', 'thou
gh', 'less', 'ancient', 'middle', 'must', 'army', 'seen', 'c', 'short', 'are
as', 'little', 'film', 'meaning', 'uses', 'kentucky', 'top', 'air', 'japanes
e', 'km', 'too', 'china', 'killed', 'right', 'son', 'formed', 'groups', 'cit
ies', 'think', 'florida', 'languages', 'earth', 'control', 'european', 'type
s', 'eastern', 'public', 'side', 'class', 'head', 'wanted', 'free', 'politic
al', 'almost', 'always', 'worked', 'married', 'places', 'islands', 'former',
'instead', 'center', 'done', 'kind', 'women', 'better', 'version', 'overvie
w', 'canton', 'find', 'gave', 'local', 'love', 'never', 'italy', 'playing',
'held', 'valea', 'hard', 'current', 'point', 'plays', 'late', 'players', 'mo

```
stly', 'once', 'park', 'département', 'building', 'know', 'shows', 'give',
'movies', 'lost', 'force', 'metal', 'mother', 'someone', 'bad', 'division',
'original', 'opera', 'lake', 'founded', 'especially', 'sun', 'study', 'roya
l', 'coast', 'illinois', 'considered', 'certain', 'character', 'want', 'soun
d', 'art', 'taken']
```

With our reduced vocabulary, we will now take our dataset and strip out all words that are not in our vocabulary. Because we are dropping a LOT of words, we are going to keep only segments that are at least 6 words long.

In [5]:
```
context_length = 5
new_corpus = []
phrase = []
for word in tqdm(words):
    if word in most_common_words:
        phrase.append(word)
    elif len(phrase) >= context_length+1:
        new_corpus.append(' '.join(phrase))
        phrase = []
    else:
        phrase = []

    if len(phrase) >= context_length+1:
        new_corpus.append(' '.join(phrase))
        phrase = []
```

```
100%|
|| 5411292/5411292 [00:08<00:00, 666958.42it/s]
```

In [6]:
```
# Remove duplicates
new_corpus = list(set(new_corpus))
```

In [7]:
```
print(new_corpus[:10])
```

```
['of the band they have had', 'to the southern part of these', 'states the t
erm is also used', 'to the black sea by km', 'it down so that the music', 'm
ade the music sound like the', 'many of the art works considered', 'of the r
egion population the municipality', 'in that many if not most', 'however it
is best known for']
```

Fantastic! We now have a dataset of grammatical six-word phrases. Next, we need to encode our words into values, so that we can feed them into our model:

In [8]:
```
words_to_int = {word: i for i, word in enumerate(most_common_words)}
int_to_words = {i: word for i, word in enumerate(most_common_words)}
```

Now we can encode any sentence (as long as it's made up of words in our vocabulary) into a sequence of integers:

In [9]:
```
def encode(sentence):
    return [words_to_int[word] for word in sentence.split(' ')]
```

```
def encode_one_hot(word):
    return [1 if i == words_to_int[word] else 0 for i in range(vocab_size)]

def decode(sequence):
    return ' '.join([int_to_words[i] for i in sequence])

def decode_one_hot(word):
    return int_to_words[np.argmax(word)]

encoded = encode('all of the people')
print(encoded)
print(decode(encoded))
```

```
[60, 1, 0, 25]
all of the people
```

Now that we have a way to convert words into integers, we can create our training data. We will use the first 5 words in a sequence to predict the 6th word. For example, given the sequence "all of the people in the", we will use "all of the people in" to predict "the". We will do this for every sequence in our dataset.

In [10]:
```
from sklearn.model_selection import train_test_split

X = []
y = []

for sentence in new_corpus:
    words = sentence.split(' ')
    for i in range(len(words)-context_length):
        X.append(encode(' '.join(words[i:i+context_length])))
        y.append(encode_one_hot(words[i+context_length]))

X = np.array(X)
y = np.array(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

for i in range(10):
    print(decode(X_train[i]), '->', decode_one_hot(y_train[i]))
```

```
in season overview it was -> performed
in the study of life -> and
in from to the city -> was
at which is also the -> home
is a man that works -> at
at the first years of -> the
so it can use it -> up
some of the music about -> the
with them until and could -> not
version of the team name -> they
```

Now let's use Keras to build our model. At its simplest, GPT has the following structure:

1. An embedding layer that converts each word into a vector

2. A transformer block
3. A linear layer that converts the output of the transformer blocks into a vector of probabilities for each word in the vocabulary

---

Optional math:

The transformer is a key concept in GPT. At its core, you can think of a single transformer block as an equivalent to a layer of neurons, but with a more complex architecture. The transformer block is made up of two parts:

1. Multi-head attention
2. A standard fully-connected layer

Multi-head attention is a way of combining information from different parts of the input. "Multi-head" really just means that we do this multiple times and combine the results. Attention can be thought of as a replacement for a standard neuron - instead of taking in all the inputs and combining them based on a single set of weights, we instead learn three different sets of weights and combine them in a more complex way. So instead of our neuron working like this:

$$y = activation(WX)$$

It works like this:

$$y = activation(\frac{W_1X * W_2X}{\sqrt{size(X)}}) * W_3X$$

If that seems confusing, don't worry - it's a very new concept in deep learning and we aren't explaining it in much detail. The real takeaway is that we are replacing our standard neuron with its one set of parameters, with a more complex neuron that has three sets of parameters.

Each word in our vocabulary will be transformed into a vector of size `8` that will be learned by the model. We will use two transformer blocks, each with two heads. The feedforward layer will have 32 neurons. The output of the feedforward layer will be flattened, and then fed into a linear layer that will output a vector of size `vocab_size`. This vector will be a probability distribution over the words in our vocabulary. We will use the `softmax` activation function to ensure that the output is a valid probability distribution.

In [11]:
```
vocab_size = len(words_to_int)
embedding_size = 8
num_heads = 2
num_transformer_blocks = 2
feedforward_dim = 32
```

```
inputs = keras.layers.Input(shape=(context_length,)) # Take in three words
embedding_layer = keras.layers.Embedding(vocab_size, embedding_size)(inputs)
transformer_block = keras.layers.MultiHeadAttention(num_heads, embedding_siz
transformer_block = keras.layers.MultiHeadAttention(num_heads, embedding_siz
transformer_block = keras.layers.Dense(feedforward_dim, activation='relu')(t
transformer_flattened = keras.layers.Flatten()(transformer_block) # Flatten
outputs = keras.layers.Dense(vocab_size, activation='softmax')(transformer_f


model = keras.Model(inputs=inputs, outputs=outputs)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['a
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   Connecte
d to
=================================================================
 input_1 (InputLayer)        [(None, 5)]               0         []

 embedding (Embedding)       (None, 5, 8)              4000      ['input_
1[0][0]']

 multi_head_attention (Mult  (None, 5, 8)              568       ['embedd
ing[0][0]',
 iHeadAttention)                                                  'embedd
ing[0][0]']

 multi_head_attention_1 (Mu  (None, 5, 8)              568       ['multi_
head_attention[0][0]',
 ltiHeadAttention)                                                'multi_
head_attention[0][0]']

 dense (Dense)               (None, 5, 32)             288       ['multi_
head_attention_1[0][0]
                                                                  ']

 flatten (Flatten)           (None, 160)               0         ['dense
[0][0]']

 dense_1 (Dense)             (None, 500)               80500     ['flatte
n[0][0]']

=================================================================
Total params: 85924 (335.64 KB)
Trainable params: 85924 (335.64 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
2023-11-13 09:51:08.959677: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:51:08.959909: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:51:08.959988: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:51:08.960174: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:51:08.960257: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:51:08.960330: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:51:08.960437: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:51:08.960512: I tensorflow/compiler/xla/stream_executor/cuda/cuda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:51:08.960566: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1639] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 10392 MB memory:  -> device: 0, name: NVIDIA GeForce RTX 3060, pci bus id: 0000:02:00.0, compute capability: 8.6
```

This is our own microscopic GPT! Our model has 85,924 trainable parameters - GPT 3.5 has 154 billion. Let's train our model on our dataset.

In [12]:
```
model.fit(
    X_train,
    y_train,
    epochs=1000,
    batch_size=1024,
    validation_data=(X_test, y_test),
    callbacks=[
```

```
        keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)
    ]
)
```

```
Epoch 1/1000
2023-11-13 09:51:10.472089: I tensorflow/compiler/xla/stream_executor/cuda/cuda_blas.cc:606] TensorFloat-32 will be used for the matrix multiplication. This will only be logged once.
2023-11-13 09:51:10.510603: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x7f13cbba0d90 initialized for platform CUDA (this does not guarantee that XLA will be used). Devices:
2023-11-13 09:51:10.510627: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): NVIDIA GeForce RTX 3060, Compute Capability 8.6
2023-11-13 09:51:10.514408: I tensorflow/compiler/mlir/tensorflow/utils/dump_mlir_util.cc:255] disabling MLIR crash reproducer, set env var `MLIR_CRASH_REPRODUCER_DIRECTORY` to enable.
2023-11-13 09:51:10.527415: I tensorflow/compiler/xla/stream_executor/cuda/cuda_dnn.cc:432] Loaded cuDNN version 8800
2023-11-13 09:51:10.571305: I tensorflow/tsl/platform/default/subprocess.cc:304] Start cannot spawn child process: No such file or directory
2023-11-13 09:51:10.596899: I ./tensorflow/compiler/jit/device_compiler.h:186] Compiled cluster using XLA!  This line is logged at most once for the lifetime of the process.
```

```
82/82 [==============================] - 5s 30ms/step - loss: 5.5730 - accur
acy: 0.0528 - val_loss: 5.1231 - val_accuracy: 0.1129
Epoch 2/1000
82/82 [==============================] - 1s 12ms/step - loss: 5.0815 - accur
acy: 0.1187 - val_loss: 5.0995 - val_accuracy: 0.1129
Epoch 3/1000
82/82 [==============================] - 1s 7ms/step - loss: 5.0598 - accura
cy: 0.1187 - val_loss: 5.0838 - val_accuracy: 0.1129
Epoch 4/1000
82/82 [==============================] - 1s 7ms/step - loss: 5.0484 - accura
cy: 0.1187 - val_loss: 5.0814 - val_accuracy: 0.1129
Epoch 5/1000
82/82 [==============================] - 1s 7ms/step - loss: 5.0448 - accura
cy: 0.1187 - val_loss: 5.0782 - val_accuracy: 0.1129
Epoch 6/1000
82/82 [==============================] - 1s 9ms/step - loss: 5.0424 - accura
cy: 0.1187 - val_loss: 5.0776 - val_accuracy: 0.1129
Epoch 7/1000
82/82 [==============================] - 1s 10ms/step - loss: 5.0396 - accur
acy: 0.1187 - val_loss: 5.0735 - val_accuracy: 0.1129
Epoch 8/1000
82/82 [==============================] - 1s 6ms/step - loss: 5.0312 - accura
cy: 0.1187 - val_loss: 5.0529 - val_accuracy: 0.1128
Epoch 9/1000
82/82 [==============================] - 1s 8ms/step - loss: 4.9811 - accura
cy: 0.1215 - val_loss: 4.9561 - val_accuracy: 0.1349
Epoch 10/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.8438 - accura
cy: 0.1419 - val_loss: 4.8244 - val_accuracy: 0.1415
Epoch 11/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.7581 - accura
cy: 0.1459 - val_loss: 4.7862 - val_accuracy: 0.1433
Epoch 12/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.7204 - accura
cy: 0.1479 - val_loss: 4.7604 - val_accuracy: 0.1455
Epoch 13/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.6798 - accura
cy: 0.1514 - val_loss: 4.7190 - val_accuracy: 0.1479
Epoch 14/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.6309 - accura
cy: 0.1526 - val_loss: 4.6817 - val_accuracy: 0.1500
Epoch 15/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.5883 - accura
cy: 0.1543 - val_loss: 4.6599 - val_accuracy: 0.1523
Epoch 16/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.5539 - accura
cy: 0.1555 - val_loss: 4.6345 - val_accuracy: 0.1517
Epoch 17/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.5264 - accura
cy: 0.1580 - val_loss: 4.6158 - val_accuracy: 0.1552
Epoch 18/1000
82/82 [==============================] - 1s 9ms/step - loss: 4.5021 - accura
cy: 0.1592 - val_loss: 4.5999 - val_accuracy: 0.1578
Epoch 19/1000
82/82 [==============================] - 1s 9ms/step - loss: 4.4818 - accura
cy: 0.1612 - val_loss: 4.5875 - val_accuracy: 0.1581
```

```
Epoch 20/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.4640 - accura
cy: 0.1626 - val_loss: 4.5732 - val_accuracy: 0.1573
Epoch 21/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.4486 - accura
cy: 0.1636 - val_loss: 4.5673 - val_accuracy: 0.1592
Epoch 22/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.4327 - accura
cy: 0.1644 - val_loss: 4.5588 - val_accuracy: 0.1615
Epoch 23/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.4214 - accura
cy: 0.1656 - val_loss: 4.5473 - val_accuracy: 0.1615
Epoch 24/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.4105 - accura
cy: 0.1665 - val_loss: 4.5413 - val_accuracy: 0.1624
Epoch 25/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.3989 - accura
cy: 0.1675 - val_loss: 4.5342 - val_accuracy: 0.1616
Epoch 26/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.3888 - accura
cy: 0.1684 - val_loss: 4.5275 - val_accuracy: 0.1634
Epoch 27/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.3797 - accura
cy: 0.1693 - val_loss: 4.5237 - val_accuracy: 0.1636
Epoch 28/1000
82/82 [==============================] - 1s 8ms/step - loss: 4.3702 - accura
cy: 0.1698 - val_loss: 4.5159 - val_accuracy: 0.1656
Epoch 29/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.3610 - accura
cy: 0.1708 - val_loss: 4.5151 - val_accuracy: 0.1650
Epoch 30/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.3529 - accura
cy: 0.1719 - val_loss: 4.5065 - val_accuracy: 0.1654
Epoch 31/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.3439 - accura
cy: 0.1721 - val_loss: 4.5051 - val_accuracy: 0.1656
Epoch 32/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.3349 - accura
cy: 0.1730 - val_loss: 4.4983 - val_accuracy: 0.1661
Epoch 33/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.3267 - accura
cy: 0.1729 - val_loss: 4.4967 - val_accuracy: 0.1665
Epoch 34/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.3194 - accura
cy: 0.1745 - val_loss: 4.4908 - val_accuracy: 0.1671
Epoch 35/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.3102 - accura
cy: 0.1748 - val_loss: 4.4856 - val_accuracy: 0.1680
Epoch 36/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.3014 - accura
cy: 0.1753 - val_loss: 4.4837 - val_accuracy: 0.1682
Epoch 37/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.2928 - accura
cy: 0.1764 - val_loss: 4.4780 - val_accuracy: 0.1692
Epoch 38/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.2845 - accura
```

```
cy: 0.1770 - val_loss: 4.4703 - val_accuracy: 0.1685
Epoch 39/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.2746 - accura
cy: 0.1780 - val_loss: 4.4679 - val_accuracy: 0.1705
Epoch 40/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.2658 - accura
cy: 0.1778 - val_loss: 4.4673 - val_accuracy: 0.1728
Epoch 41/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.2564 - accura
cy: 0.1788 - val_loss: 4.4601 - val_accuracy: 0.1734
Epoch 42/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.2475 - accura
cy: 0.1791 - val_loss: 4.4520 - val_accuracy: 0.1728
Epoch 43/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.2374 - accura
cy: 0.1799 - val_loss: 4.4470 - val_accuracy: 0.1729
Epoch 44/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.2284 - accura
cy: 0.1805 - val_loss: 4.4507 - val_accuracy: 0.1728
Epoch 45/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.2199 - accura
cy: 0.1809 - val_loss: 4.4372 - val_accuracy: 0.1742
Epoch 46/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.2105 - accura
cy: 0.1830 - val_loss: 4.4325 - val_accuracy: 0.1751
Epoch 47/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.2011 - accura
cy: 0.1827 - val_loss: 4.4290 - val_accuracy: 0.1761
Epoch 48/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1929 - accura
cy: 0.1838 - val_loss: 4.4315 - val_accuracy: 0.1758
Epoch 49/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1856 - accura
cy: 0.1849 - val_loss: 4.4280 - val_accuracy: 0.1775
Epoch 50/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1766 - accura
cy: 0.1852 - val_loss: 4.4223 - val_accuracy: 0.1775
Epoch 51/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1695 - accura
cy: 0.1859 - val_loss: 4.4222 - val_accuracy: 0.1782
Epoch 52/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1625 - accura
cy: 0.1861 - val_loss: 4.4159 - val_accuracy: 0.1777
Epoch 53/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1551 - accura
cy: 0.1870 - val_loss: 4.4146 - val_accuracy: 0.1777
Epoch 54/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.1477 - accura
cy: 0.1868 - val_loss: 4.4121 - val_accuracy: 0.1782
Epoch 55/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1409 - accura
cy: 0.1885 - val_loss: 4.4114 - val_accuracy: 0.1784
Epoch 56/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1352 - accura
cy: 0.1888 - val_loss: 4.4054 - val_accuracy: 0.1793
Epoch 57/1000
```

```
82/82 [==============================] - 1s 6ms/step - loss: 4.1287 - accura
cy: 0.1895 - val_loss: 4.4062 - val_accuracy: 0.1788
Epoch 58/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1229 - accura
cy: 0.1893 - val_loss: 4.3996 - val_accuracy: 0.1797
Epoch 59/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.1147 - accura
cy: 0.1901 - val_loss: 4.3999 - val_accuracy: 0.1806
Epoch 60/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1091 - accura
cy: 0.1900 - val_loss: 4.4109 - val_accuracy: 0.1793
Epoch 61/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.1024 - accura
cy: 0.1906 - val_loss: 4.4012 - val_accuracy: 0.1794
Epoch 62/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0969 - accura
cy: 0.1914 - val_loss: 4.3934 - val_accuracy: 0.1806
Epoch 63/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0904 - accura
cy: 0.1917 - val_loss: 4.4014 - val_accuracy: 0.1811
Epoch 64/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0852 - accura
cy: 0.1916 - val_loss: 4.3939 - val_accuracy: 0.1819
Epoch 65/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0783 - accura
cy: 0.1920 - val_loss: 4.3943 - val_accuracy: 0.1822
Epoch 66/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0723 - accura
cy: 0.1929 - val_loss: 4.4000 - val_accuracy: 0.1826
Epoch 67/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0658 - accura
cy: 0.1932 - val_loss: 4.3990 - val_accuracy: 0.1827
Epoch 68/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0600 - accura
cy: 0.1938 - val_loss: 4.3898 - val_accuracy: 0.1825
Epoch 69/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0543 - accura
cy: 0.1939 - val_loss: 4.3871 - val_accuracy: 0.1840
Epoch 70/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0481 - accura
cy: 0.1949 - val_loss: 4.3913 - val_accuracy: 0.1846
Epoch 71/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0427 - accura
cy: 0.1959 - val_loss: 4.3932 - val_accuracy: 0.1847
Epoch 72/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0359 - accura
cy: 0.1959 - val_loss: 4.3823 - val_accuracy: 0.1848
Epoch 73/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0301 - accura
cy: 0.1968 - val_loss: 4.3792 - val_accuracy: 0.1849
Epoch 74/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0238 - accura
cy: 0.1979 - val_loss: 4.3851 - val_accuracy: 0.1856
Epoch 75/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.0186 - accura
cy: 0.1978 - val_loss: 4.3766 - val_accuracy: 0.1855
```

```
Epoch 76/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0124 - accura
cy: 0.1983 - val_loss: 4.3752 - val_accuracy: 0.1871
Epoch 77/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0065 - accura
cy: 0.1995 - val_loss: 4.3764 - val_accuracy: 0.1872
Epoch 78/1000
82/82 [==============================] - 1s 6ms/step - loss: 4.0001 - accura
cy: 0.1998 - val_loss: 4.3747 - val_accuracy: 0.1864
Epoch 79/1000
82/82 [==============================] - 1s 8ms/step - loss: 3.9941 - accura
cy: 0.2010 - val_loss: 4.3893 - val_accuracy: 0.1881
Epoch 80/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9884 - accura
cy: 0.2007 - val_loss: 4.3774 - val_accuracy: 0.1892
Epoch 81/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9822 - accura
cy: 0.2020 - val_loss: 4.3727 - val_accuracy: 0.1880
Epoch 82/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9760 - accura
cy: 0.2021 - val_loss: 4.3739 - val_accuracy: 0.1888
Epoch 83/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9699 - accura
cy: 0.2037 - val_loss: 4.3685 - val_accuracy: 0.1899
Epoch 84/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9644 - accura
cy: 0.2037 - val_loss: 4.3641 - val_accuracy: 0.1894
Epoch 85/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9571 - accura
cy: 0.2045 - val_loss: 4.3639 - val_accuracy: 0.1902
Epoch 86/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.9525 - accura
cy: 0.2057 - val_loss: 4.3652 - val_accuracy: 0.1914
Epoch 87/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9467 - accura
cy: 0.2064 - val_loss: 4.3631 - val_accuracy: 0.1903
Epoch 88/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9399 - accura
cy: 0.2066 - val_loss: 4.3676 - val_accuracy: 0.1924
Epoch 89/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9338 - accura
cy: 0.2068 - val_loss: 4.3623 - val_accuracy: 0.1920
Epoch 90/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9272 - accura
cy: 0.2084 - val_loss: 4.3597 - val_accuracy: 0.1918
Epoch 91/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.9214 - accura
cy: 0.2096 - val_loss: 4.3579 - val_accuracy: 0.1913
Epoch 92/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9166 - accura
cy: 0.2090 - val_loss: 4.3609 - val_accuracy: 0.1934
Epoch 93/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9102 - accura
cy: 0.2117 - val_loss: 4.3567 - val_accuracy: 0.1928
Epoch 94/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9033 - accura
```

```
cy: 0.2115 - val_loss: 4.3543 - val_accuracy: 0.1936
Epoch 95/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8993 - accura
cy: 0.2123 - val_loss: 4.3615 - val_accuracy: 0.1943
Epoch 96/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8922 - accura
cy: 0.2134 - val_loss: 4.3607 - val_accuracy: 0.1944
Epoch 97/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8855 - accura
cy: 0.2136 - val_loss: 4.3469 - val_accuracy: 0.1961
Epoch 98/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8802 - accura
cy: 0.2149 - val_loss: 4.3508 - val_accuracy: 0.1967
Epoch 99/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8736 - accura
cy: 0.2158 - val_loss: 4.3492 - val_accuracy: 0.1984
Epoch 100/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8686 - accura
cy: 0.2168 - val_loss: 4.3469 - val_accuracy: 0.1968
Epoch 101/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8628 - accura
cy: 0.2171 - val_loss: 4.3452 - val_accuracy: 0.1975
Epoch 102/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8559 - accura
cy: 0.2175 - val_loss: 4.3435 - val_accuracy: 0.1983
Epoch 103/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8506 - accura
cy: 0.2176 - val_loss: 4.3520 - val_accuracy: 0.1990
Epoch 104/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8435 - accura
cy: 0.2193 - val_loss: 4.3348 - val_accuracy: 0.2001
Epoch 105/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8390 - accura
cy: 0.2201 - val_loss: 4.3396 - val_accuracy: 0.2011
Epoch 106/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8327 - accura
cy: 0.2205 - val_loss: 4.3345 - val_accuracy: 0.2002
Epoch 107/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8273 - accura
cy: 0.2208 - val_loss: 4.3273 - val_accuracy: 0.1995
Epoch 108/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8213 - accura
cy: 0.2218 - val_loss: 4.3407 - val_accuracy: 0.2016
Epoch 109/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8141 - accura
cy: 0.2234 - val_loss: 4.3361 - val_accuracy: 0.2017
Epoch 110/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8087 - accura
cy: 0.2231 - val_loss: 4.3355 - val_accuracy: 0.2007
Epoch 111/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8038 - accura
cy: 0.2236 - val_loss: 4.3345 - val_accuracy: 0.2031
Epoch 112/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7997 - accura
cy: 0.2248 - val_loss: 4.3310 - val_accuracy: 0.2033
Epoch 113/1000
```

```
82/82 [==============================] - 1s 6ms/step - loss: 3.7939 - accura
cy: 0.2252 - val_loss: 4.3399 - val_accuracy: 0.2021
Epoch 114/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7887 - accura
cy: 0.2255 - val_loss: 4.3271 - val_accuracy: 0.2026
Epoch 115/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7836 - accura
cy: 0.2264 - val_loss: 4.3376 - val_accuracy: 0.2048
Epoch 116/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7769 - accura
cy: 0.2265 - val_loss: 4.3389 - val_accuracy: 0.2028
Epoch 117/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.7715 - accura
cy: 0.2271 - val_loss: 4.3318 - val_accuracy: 0.2039
Epoch 118/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7674 - accura
cy: 0.2278 - val_loss: 4.3356 - val_accuracy: 0.2041
Epoch 119/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7621 - accura
cy: 0.2285 - val_loss: 4.3411 - val_accuracy: 0.2050
Epoch 120/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7574 - accura
cy: 0.2286 - val_loss: 4.3277 - val_accuracy: 0.2035
Epoch 121/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7527 - accura
cy: 0.2291 - val_loss: 4.3366 - val_accuracy: 0.2048
Epoch 122/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7482 - accura
cy: 0.2295 - val_loss: 4.3369 - val_accuracy: 0.2051
Epoch 123/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7416 - accura
cy: 0.2308 - val_loss: 4.3245 - val_accuracy: 0.2075
Epoch 124/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7375 - accura
cy: 0.2310 - val_loss: 4.3459 - val_accuracy: 0.2082
Epoch 125/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7330 - accura
cy: 0.2317 - val_loss: 4.3324 - val_accuracy: 0.2062
Epoch 126/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7280 - accura
cy: 0.2326 - val_loss: 4.3445 - val_accuracy: 0.2062
Epoch 127/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.7222 - accura
cy: 0.2326 - val_loss: 4.3409 - val_accuracy: 0.2076
Epoch 128/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7190 - accura
cy: 0.2332 - val_loss: 4.3463 - val_accuracy: 0.2090
Epoch 129/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7138 - accura
cy: 0.2336 - val_loss: 4.3365 - val_accuracy: 0.2070
Epoch 130/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7100 - accura
cy: 0.2337 - val_loss: 4.3423 - val_accuracy: 0.2085
Epoch 131/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7049 - accura
cy: 0.2343 - val_loss: 4.3355 - val_accuracy: 0.2083
```

```
Epoch 132/1000
82/82 [==============================] - 1s 9ms/step - loss: 3.7004 - accura
cy: 0.2347 - val_loss: 4.3426 - val_accuracy: 0.2089
Epoch 133/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.6961 - accura
cy: 0.2347 - val_loss: 4.3443 - val_accuracy: 0.2092
```

Out[12]: <keras.src.callbacks.History at 0x7f14b00f0050>

As you can see, our accuracy is not very good. It picks the right word something like 1 in 5 times. That's still a lot better than random, which would be 1 in 500, but it's not enough to be useful in practice. This is because our model is very small, and so is our dataset. However, the principles we are using here are exactly the same as in GPT - just on a smaller scale. Let's see how our model performs on some sample sentence:

In [13]:
```
sample_sentence = 'the united states is one'
sample_sentence_encoded = encode(sample_sentence)
print(sample_sentence_encoded)
predictions = model.predict(np.array([sample_sentence_encoded]))
print(decode_one_hot(predictions[0]))
```

```
[0, 55, 57, 5, 32]
1/1 [==============================] - 0s 84ms/step
of
```

When we talk about models like GPT-3.5, we talk about the 'context window'. This is the number of tokens we feed into the model to get one word out. In our case, our context window is 5. In GPT-3.5, the context window is 4096, or 16384, depending on the model. The latest version of GPT-4 supports a context window of up to 128,000 tokens - as much as 300 pages of text. No matter what the context is, we are only getting one word out - if we want to produce a larger sequence, we have to successively feed the output back into the model. This is called 'autoregressive generation'. We can use our model to generate a sequence of words like this:

In [14]:
```
def generate_sequence(model, context, length):
    result = context
    for i in range(length):
        predictions = model.predict(np.array([context]))
        context = np.append(context, np.argmax(predictions[0]))
        result = np.append(result, np.argmax(predictions[0]))
        context = context[1:]
    return result

print(decode(generate_sequence(model, encode('in the way of the'), 10)))
```

```
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
in the way of the united states the united states and the united states the
```

This is precisely how models like ChatGPT generate text. They take in the context (which, as we discussed, is typically much longer than 5 words) and generate the next word. However, unlike our case, where we choose a fixed number of words to generate, ChatGPT keeps generating words until it reaches a special token that marks the end of a sequence (often `<end>`). This is how it can generate text of arbitrary length.

Now we have created our very own GPT model. But this is not the same as ChatGPT. Models like ChatGPT go one step further, to make the model more useful for conversation. This is done using a technique called Reinforcement Learning from Human Feedback (RLHF).

RLHF expands on the training process we've seen above by adding a second model, called the discriminator or the adversary. The role of the adversary is to rate the quality of a response based on some conditions that we care about. In the case of ChatGPT, the adversary is looking for things like whether the response fits the conversational style, and whether it avoids sensitive topics. The adversary is trained using *human feedback* - humans rate the quality of responses, and the adversary learns to predict the human rating. The adversary is then used to train the generator (the GPT model) - the generator is rewarded for producing responses that the adversary rates highly. This is called adversarial training, and it is a very powerful technique for training models.

We are going to make our own extremely simple adversary. Our adversary will assign a score to the response based on how many times the letter 'e' appears in the response. We will then use this score to train our generator. This is a very simple example, but it demonstrates the principle of adversarial training.

Because we can directly calculate how many 'e's appear in each of our vocabulary words, we don't need to 'train' our adversary - we can just use it directly. We will use the following function to calculate the adversary score of a word:

In [15]:
```
adversary_scores = [word.count('e') for word in most_common_words]
def adversary_score(y_true, y_pred):
    return tf.reduce_sum(y_pred * adversary_scores, axis=-1)
```

In [16]:
```
sample_sentence = "in the same way i"
sample_sentence_encoded = encode(sample_sentence)
prediction = model.predict(np.array([sample_sentence_encoded]))[0]
print(decode_one_hot(prediction))
print(adversary_score(None, prediction))
```

```
1/1 [==============================] - 0s 12ms/step
the
tf.Tensor(0.5640126533411156, shape=(), dtype=float64)
```

As you should be able to see, the adversary assigns a score greater than zero even if the predicted word doesn't have any 'e's in it. This is because the predicted word is not a one-hot vector - it is a probability distribution. The adversary is assigning a score to the entire distribution, not just the most likely word. This is valuable because we typically avoid methods which can produce zero as an error - in a nutshell, if the error is zero, the model doesn't know how to change things in order to improve. Our approach instead will incentivize the model to consider all words containing 'e's more strongly.

In [17]:
```
sample_sentence = "in the same way i"
sample_sentence_encoded = encode(sample_sentence)
prediction = model.predict(np.array([sample_sentence_encoded]))[0]
print("Top 10 most likely words")
print("Word       | Chance \t| e count | Adversary score")
for i in sorted(zip(prediction, most_common_words), reverse=True)[:10]:
    print(f'{i[1]:10} | {i[0]*100:.0f}% \t| {i[1].count("e")}       | {i[0]*
```

```
1/1 [==============================] - 0s 11ms/step
Top 10 most likely words
Word       | Chance     | e count | Adversary score
the        | 12%        | 1       | 0.1155
important  | 4%         | 0       | 0.0000
people     | 4%         | 2       | 0.0780
a          | 4%         | 0       | 0.0000
was        | 3%         | 0       | 0.0000
in         | 2%         | 0       | 0.0000
is         | 2%         | 0       | 0.0000
we         | 2%         | 1       | 0.0199
means      | 2%         | 1       | 0.0183
to         | 2%         | 0       | 0.0000
```

In [18]:
```
def combined_loss(y_true, y_pred):
    adversary_weight = 0.75 # Modify this to increase or decrease the influe
    return tf.losses.categorical_crossentropy(y_true, y_pred) - adversary_we

# Duplicate the model
model_adversary = keras.models.clone_model(model)

model_adversary.compile(optimizer='adam',
```

```
              loss=combined_loss,
              metrics=['accuracy', adversary_score])
```

In [19]:

```
model_adversary.fit(
    X_train,
    y_train,
    epochs=1000,
    batch_size=1024,
    validation_data=(X_test, y_test),
    callbacks=[
        keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=10)
    ]
)
```

```
Epoch 1/1000
82/82 [==============================] - 4s 29ms/step - loss: 4.9615 - accur
acy: 0.0851 - adversary_score: 1.1056 - val_loss: 4.4430 - val_accuracy: 8.1
422e-04 - val_adversary_score: 1.6533
Epoch 2/1000
82/82 [==============================] - 1s 8ms/step - loss: 4.4063 - accura
cy: 6.9449e-04 - adversary_score: 1.6625 - val_loss: 4.4291 - val_accuracy:
7.1843e-04 - val_adversary_score: 1.6659
Epoch 3/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.3953 - accura
cy: 0.0011 - adversary_score: 1.6432 - val_loss: 4.4231 - val_accuracy: 7.18
43e-04 - val_adversary_score: 1.6301
Epoch 4/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.3897 - accura
cy: 0.0011 - adversary_score: 1.6196 - val_loss: 4.4204 - val_accuracy: 7.18
43e-04 - val_adversary_score: 1.6099
Epoch 5/1000
82/82 [==============================] - 1s 8ms/step - loss: 4.3869 - accura
cy: 0.0011 - adversary_score: 1.6085 - val_loss: 4.4189 - val_accuracy: 7.18
43e-04 - val_adversary_score: 1.6180
Epoch 6/1000
82/82 [==============================] - 1s 10ms/step - loss: 4.3845 - accur
acy: 0.0011 - adversary_score: 1.6076 - val_loss: 4.4170 - val_accuracy: 7.1
843e-04 - val_adversary_score: 1.6160
Epoch 7/1000
82/82 [==============================] - 1s 8ms/step - loss: 4.3771 - accura
cy: 0.0018 - adversary_score: 1.6176 - val_loss: 4.3965 - val_accuracy: 0.01
29 - val_adversary_score: 1.6254
Epoch 8/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.3356 - accura
cy: 0.0233 - adversary_score: 1.6706 - val_loss: 4.3470 - val_accuracy: 0.01
56 - val_adversary_score: 1.7029
Epoch 9/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.2907 - accura
cy: 0.0151 - adversary_score: 1.6733 - val_loss: 4.2993 - val_accuracy: 0.00
85 - val_adversary_score: 1.6733
Epoch 10/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.1785 - accura
cy: 0.0157 - adversary_score: 1.6726 - val_loss: 4.1530 - val_accuracy: 0.01
06 - val_adversary_score: 1.6292
Epoch 11/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.0808 - accura
cy: 0.0131 - adversary_score: 1.6737 - val_loss: 4.1092 - val_accuracy: 0.01
61 - val_adversary_score: 1.6783
Epoch 12/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.0427 - accura
cy: 0.0101 - adversary_score: 1.6741 - val_loss: 4.0852 - val_accuracy: 0.00
49 - val_adversary_score: 1.6617
Epoch 13/1000
82/82 [==============================] - 1s 7ms/step - loss: 4.0175 - accura
cy: 0.0190 - adversary_score: 1.6710 - val_loss: 4.0675 - val_accuracy: 0.02
51 - val_adversary_score: 1.6714
Epoch 14/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9962 - accura
cy: 0.0350 - adversary_score: 1.6685 - val_loss: 4.0501 - val_accuracy: 0.02
51 - val_adversary_score: 1.6555
```

```
Epoch 15/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9783 - accura
cy: 0.0351 - adversary_score: 1.6634 - val_loss: 4.0376 - val_accuracy: 0.02
98 - val_adversary_score: 1.6395
Epoch 16/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9634 - accura
cy: 0.0376 - adversary_score: 1.6598 - val_loss: 4.0279 - val_accuracy: 0.03
01 - val_adversary_score: 1.6944
Epoch 17/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.9486 - accura
cy: 0.0368 - adversary_score: 1.6580 - val_loss: 4.0098 - val_accuracy: 0.04
42 - val_adversary_score: 1.6297
Epoch 18/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.9326 - accura
cy: 0.0305 - adversary_score: 1.6614 - val_loss: 3.9938 - val_accuracy: 0.04
35 - val_adversary_score: 1.6628
Epoch 19/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.9142 - accura
cy: 0.0390 - adversary_score: 1.6604 - val_loss: 3.9792 - val_accuracy: 0.05
23 - val_adversary_score: 1.6685
Epoch 20/1000
82/82 [==============================] - 1s 8ms/step - loss: 3.8928 - accura
cy: 0.0432 - adversary_score: 1.6631 - val_loss: 3.9581 - val_accuracy: 0.05
61 - val_adversary_score: 1.6573
Epoch 21/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8614 - accura
cy: 0.0420 - adversary_score: 1.6672 - val_loss: 3.9250 - val_accuracy: 0.04
71 - val_adversary_score: 1.6553
Epoch 22/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8286 - accura
cy: 0.0447 - adversary_score: 1.6676 - val_loss: 3.9061 - val_accuracy: 0.03
30 - val_adversary_score: 1.6814
Epoch 23/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.8062 - accura
cy: 0.0447 - adversary_score: 1.6691 - val_loss: 3.8900 - val_accuracy: 0.04
60 - val_adversary_score: 1.6447
Epoch 24/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.7893 - accura
cy: 0.0432 - adversary_score: 1.6700 - val_loss: 3.8788 - val_accuracy: 0.03
01 - val_adversary_score: 1.6638
Epoch 25/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.7732 - accura
cy: 0.0397 - adversary_score: 1.6699 - val_loss: 3.8693 - val_accuracy: 0.02
90 - val_adversary_score: 1.6771
Epoch 26/1000
82/82 [==============================] - 1s 8ms/step - loss: 3.7603 - accura
cy: 0.0383 - adversary_score: 1.6709 - val_loss: 3.8612 - val_accuracy: 0.04
23 - val_adversary_score: 1.6723
Epoch 27/1000
82/82 [==============================] - 1s 6ms/step - loss: 3.7473 - accura
cy: 0.0380 - adversary_score: 1.6713 - val_loss: 3.8512 - val_accuracy: 0.03
93 - val_adversary_score: 1.6625
Epoch 28/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.7347 - accura
cy: 0.0379 - adversary_score: 1.6718 - val_loss: 3.8420 - val_accuracy: 0.03
28 - val_adversary_score: 1.6429
```

```
Epoch 29/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.7223 - accura
cy: 0.0374 - adversary_score: 1.6708 - val_loss: 3.8326 - val_accuracy: 0.03
94 - val_adversary_score: 1.6905
Epoch 30/1000
82/82 [==============================] - 1s 7ms/step - loss: 3.7113 - accura
cy: 0.0416 - adversary_score: 1.6707 - val_loss: 3.8268 - val_accuracy: 0.02
79 - val_adversary_score: 1.6857
```

Out[19]: <keras.src.callbacks.History at 0x7f15000826d0>

Now let's compare the predictions of our original model and our adversary model:

In [20]:
```
original_predictions = model.predict(X_test)
adversary_predictions = model_adversary.predict(X_test)

print(f'Adversary score for original model: {np.mean(adversary_score(None, c
print(f'Adversary score for adversary model: {np.mean(adversary_score(None,

for i in range(10):
    true_word = decode_one_hot(y_test[i])
    original_word = decode_one_hot(original_predictions[i])
    adversary_word = decode_one_hot(adversary_predictions[i])
    print(f'{true_word:10} | {original_word:10} | {adversary_word:10}')
```

```
653/653 [==============================] - 1s 856us/step
653/653 [==============================] - 1s 869us/step
Adversary score for original model: 0.5046143530021954
Adversary score for adversary model: 1.6856868990073963
the        | he         | september
city       | group      | between
species    | time       | between
a          | the        | the
way        | of         | released
league     | in         | between
movies     | the        | september
band       | first      | september
roman      | countries  | between
living     | years      | between
```

As we can see, introducing the adversary has dramatically increased the model's likelihood to choose words with lots of 'e's in them. Of course, in this setting, that's at a cost to the model's accuracy. However, in a real-world setting, we would use a more sophisticated adversary, and we would use a more sophisticated metric than just accuracy.

So there you have it! We have built our own GPT model, and we have seen how we can use an adversary to obtain specific behaviour. Of course, there are many more details that go into building a model like ChatGPT, but this is the core of it. I hope you enjoyed this tutorial!