

GPT in 60 Lines of NumPy

January 30, 2023

In this post, we'll implement a GPT from scratch in just [60 lines of numpy](#). We'll then load the trained GPT-2 model weights released by OpenAI into our implementation and generate some text.

Note:

- This post assumes familiarity with Python, NumPy, and some basic experience training neural networks.
- This implementation is missing tons of features on purpose to keep it as simple as possible while remaining complete. The goal is to provide a **simple yet complete technical introduction to the GPT as an educational tool**.
- The GPT architecture is just one small part of what makes LLMs what they are today.^[1]
- All the code for this blog post can be found at github.com/jaymody/picoGPT.
- [Hacker news thread](#)

EDIT (Feb 9th, 2023): Added a "What's Next" section and updated the intro with some notes.

EDIT (Feb 28th, 2023): Added some additional sections to "What's Next".

Table of Contents

- [Table of Contents](#)
- [What is a GPT?](#)
 - [Input / Output](#)
 - [Input](#)
 - [Output](#)
 - [Generating Text](#)
 - [Autoregressive](#)
 - [Sampling](#)
 - [Training](#)
 - [Prompting](#)
- [Setup](#)
 - [Encoder](#)
 - [Hyperparameters](#)
 - [Parameters](#)
- [Basic Layers](#)
 - [GELU](#)
 - [Softmax](#)
 - [Layer Normalization](#)
 - [Linear](#)
- [GPT Architecture](#)
 - [Embeddings](#)
 - [Token Embeddings](#)
 - [Positional Embeddings](#)
 - [Combined](#)
 - [Decoder Stack](#)
 - [Projection to Vocab](#)
 - [Decoder Block](#)
 - [Position-wise Feed Forward Network](#)
 - [Multi-Head Causal Self Attention](#)
 - [Attention](#)
 - [Self](#)
 - [Causal](#)
 - [Multi-Head](#)
- [Putting it All Together](#)
- [What Next?](#)
 - [GPU/TPU Support](#)
 - [Backpropagation](#)
 - [Batching](#)
 - [Inference Optimization](#)
 - [Training](#)
 - [Evaluation](#)
 - [Architecture Improvements](#)
 - [Stopping Generation](#)
 - [Unconditional Generation](#)
 - [Fine-tuning](#)
 - [Classification Fine-tuning](#)
 - [Generative Fine-tuning](#)
 - [Instruction Fine-tuning](#)
 - [Parameter Efficient Fine-tuning](#)

What is a GPT?

GPT stands for **Generative Pre-trained Transformer**. It's a type of neural network architecture based on the [Transformer](#). [Jay Alammar's How GPT3 Works](#) is an excellent introduction to GPTs at a high level, but here's the tl;dr:

- **Generative:** A GPT *generates* text.
- **Pre-trained:** A GPT is *trained* on lots of text from books, the internet, etc ...
- **Transformer:** A GPT is a decoder-only *transformer* neural network.

Large Language Models (LLMs) like [OpenAI's GPT-3](#), [Google's LaMDA](#), and [Cohere's Command XLarge](#) are just GPTs under the hood. What makes them special is they happen to be **1**) very big (billions of parameters) and **2**) trained on lots of data (hundreds of gigabytes of text).

Fundamentally, a GPT **generates text** given a **prompt**. Even with this very simple API (input = text, output = text), a well-trained GPT can do some pretty awesome stuff like [write your emails](#), [summarize a book](#), [give you instagram caption ideas](#), [explain black holes to a 5 year old](#), [code in SQL](#), and [even write your will](#).

So that's a high-level overview of GPTs and their capabilities. Let's dig into some more specifics.

Input / Output

The function signature for a GPT looks roughly like this:

```
def gpt(inputs: list[int]) -> list[list[float]]:  
    # inputs has shape [n_seq]  
    # output has shape [n_seq, n_vocab]  
    output = # beep boop neural network magic  
    return output
```

Input

The input is some text represented by a **sequence of integers** that map to **tokens** in the text:

```
# integers represent tokens in our text, for example:  
# text  = "not all heroes wear capes":  
# tokens = "not" "all" "heroes" "wear" "capes"  
inputs = [1, 0, 2, 4, 6]
```

Tokens are sub-pieces of the text, which are produced using some kind of **tokenizer**. We can map tokens to integers using a **vocabulary**:

```
# the index of a token in the vocab represents the integer id for that token  
# i.e. the integer id for "heroes" would be 2, since vocab[2] = "heroes"  
vocab = ["all", "not", "heroes", "the", "wear", ".", "capes"]  
  
# a pretend tokenizer that tokenizes on whitespace  
tokenizer = WhitespaceTokenizer(vocab)  
  
# the encode() method converts a str -> list[int]  
ids = tokenizer.encode("not all heroes wear") # ids = [1, 0, 2, 4]  
  
# we can see what the actual tokens are via our vocab mapping  
tokens = [tokenizer.vocab[i] for i in ids] # tokens = ["not", "all", "heroes", "wear"]  
  
# the decode() method converts back a list[int] -> str  
text = tokenizer.decode(ids) # text = "not all heroes wear"
```

In short:

- We have a string.
- We use a tokenizer to break it down into smaller pieces called tokens.
- We use a vocabulary to map those tokens to integers.

In practice, we use more advanced methods of tokenization than simply splitting by whitespace, such as [Byte-Pair Encoding](#) or [WordPiece](#), but the principle is the same:

1. There is a `vocab` that maps string tokens to integer indices
2. There is an `encode` method that converts `str -> list[int]`
3. There is a `decode` method that converts `list[int] -> str` [2]

Output

The output is a **2D array**, where `output[1][j]` is the model's **predicted probability** that the token at `vocab[1]` is the next token, `inputs[1]`. For example:

TOKEN AT `vocab[1]` IS THE NEXT TOKEN `inputs[-1]`, FOR EXAMPLE.

```
vocab = ["all", "not", "heroes", "the", "wear", ".", "capes"]
inputs = [1, 0, 2, 4] # "not" "all" "heroes" "wear"
output = gpt(inputs)
#          [all, not, heroes, the, wear, ., capes]
# output[0] = [0.75  0.1   0.0   0.15  0.0   0.0   0.0 ]
# given just "not", the model predicts the word "all" with the highest probability

#          [all, not, heroes, the, wear, ., capes]
# output[1] = [0.0   0.0   0.8   0.1   0.0   0.0   0.1 ]
# given the sequence ["not", "all"], the model predicts the word "heroes" with the highest probability

#          [all, not, heroes, the, wear, ., capes]
# output[-1] = [0.0   0.0   0.0   0.1   0.0   0.05  0.85 ]
# given the whole sequence ["not", "all", "heroes", "wear"], the model predicts the word "capes"
```

To get a **next token prediction** for the whole sequence, we simply take the token with the highest probability in `output[-1]`:

```
vocab = ["all", "not", "heroes", "the", "wear", ".", "capes"]
inputs = [1, 0, 2, 4] # "not" "all" "heroes" "wear"
output = gpt(inputs)
next_token_id = np.argmax(output[-1]) # next_token_id = 6
next_token = vocab[next_token_id] # next_token = "capes"
```

Taking the token with the highest probability as our prediction is known as **greedy decoding** or **greedy sampling**.

The task of predicting the next logical word in a sequence is called **language modeling**. As such, we can call a GPT a **language model**.

Generating a single word is cool and all, but what about entire sentences, paragraphs, etc ...?

Generating Text

Autoregressive

We can generate full sentences by iteratively getting the next token prediction from our model. At each iteration, we append the predicted token back into the input:

```
def generate(inputs, n_tokens_to_generate):
    for _ in range(n_tokens_to_generate): # auto-regressive decode loop
        output = gpt(inputs) # model forward pass
        next_id = np.argmax(output[-1]) # greedy sampling
        inputs.append(int(next_id)) # append prediction to input
    return inputs[len(inputs) - n_tokens_to_generate : ] # only return generated ids

input_ids = [1, 0] # "not" "all"
output_ids = generate(input_ids, 3) # output_ids = [2, 4, 6]
output_tokens = [vocab[i] for i in output_ids] # "heroes" "wear" "capes"
```

This process of predicting a future value (regression), and adding it back into the input (auto), is why you might see a GPT described as **autoregressive**.

Sampling

We can introduce some **stochasticity** (randomness) to our generations by sampling from the probability distribution instead of being greedy:

```
inputs = [1, 0, 2, 4] # "not" "all" "heroes" "wear"
output = gpt(inputs)
np.random.choice(np.arange(vocab_size), p=output[-1]) # capes
np.random.choice(np.arange(vocab_size), p=output[-1]) # hats
np.random.choice(np.arange(vocab_size), p=output[-1]) # capes
np.random.choice(np.arange(vocab_size), p=output[-1]) # capes
np.random.choice(np.arange(vocab_size), p=output[-1]) # pants
```

This allows us to generate different sentences given the same input. When combined with techniques like **top-k**, **top-p**, and **temperature**, which modify the distribution prior to sampling, the quality of our outputs is greatly increased. These techniques also introduce some hyperparameters that we can play around with to get different generation behaviors (for example, increasing temperature makes our model take more risks and thus be more "creative").

Training

We train a GPT like any other neural network, using **gradient descent** with respect to some **loss function**. In the case of a GPT, we take the **cross entropy loss** over the **language modeling task**:

```

def lm_loss(inputs: list[int], params) -> float:
    # the labels y are just the input shifted 1 to the left
    #
    # inputs = [not, all, heros, wear, capes]
    # x = [not, all, heroes, wear]
    # y = [all, heroes, wear, capes]
    #
    # of course, we don't have a label for inputs[-1], so we exclude it from x
    #
    # as such, for N inputs, we have N - 1 language modeling example pairs
    x, y = inputs[:-1], inputs[1:]

    # forward pass
    # all the predicted next token probability distributions at each position
    output = gpt(x, params)

    # cross entropy loss
    # we take the average over all N-1 examples
    loss = np.mean(-np.log(output[y]))

    return loss

def train(texts: list[list[str]], params) -> float:
    for text in texts:
        inputs = tokenizer.encode(text)
        loss = lm_loss(inputs, params)
        gradients = compute_gradients_via_backpropagation(loss, params)
        params = gradient_descent_update_step(gradients, params)
    return params

```

This is a heavily simplified training setup, but it illustrates the point. Notice the addition of `params` to our `gpt` function signature (we left this out in the previous sections for simplicity). During each iteration of the training loop:

1. We compute the language modeling loss for the given input text example
2. The loss determines our gradients, which we compute via backpropagation
3. We use the gradients to update our model parameters such that the loss is minimized (gradient descent)

Notice, we don't use explicitly labelled data. Instead, we are able to produce the input/label pairs from just the raw text itself. This is referred to as [self-supervised learning](#).

Self-supervision enables us to massively scale train data, just get our hands on as much raw text as possible and throw it at the model. For example, GPT-3 was trained on **300 billion tokens** of text from the internet and books:

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Table 2.2: Datasets used to train GPT-3. "Weight in training mix" refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen up to 3.4 times during training while other datasets are seen less than once.

Table 2.2 from GPT-3 paper

Of course, you need a sufficiently large model to be able to learn from all this data, which is why GPT-3 has **175 billion parameters** and probably cost between [\\$1m-10m in compute cost to train](#).^[3]

This self-supervised training step is called **pre-training**, since we can reuse the "pre-trained" models weights to further train the model on downstream tasks, such as classifying if a tweet is toxic or not. Pre-trained models are also sometimes called **foundation models**.

Training the model on downstream tasks is called **fine-tuning**, since the model weights have already been pre-trained to understand language, it's just being fine-tuned to the specific task at hand.

The "pre-training on a general task + fine-tuning on a specific task" strategy is called [transfer learning](#).

Prompting

In principle, the original [GPT](#) paper was only about the benefits of pre-training a transformer model for transfer learning. The paper showed that pre-training a 117M GPT achieved state-of-the-art performance on various **NLP** (natural language processing) tasks when fine-tuned on labelled datasets.

It wasn't until the [GPT-2](#) and [GPT-3](#) papers that we realized a GPT model pre-trained on enough data with enough parameters was capable of performing any arbitrary task **by itself**, no fine-tuning needed. Just prompt the model, perform autoregressive language modeling, and like voila, the model magically gives us an appropriate response. This is referred to as **in-context learning**, because the model is using just the context of the prompt to perform the task. In-context learning can be zero shot, one shot, or few shot:

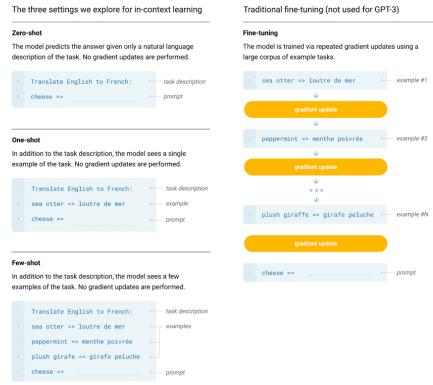


Figure 2.1 from the GPT-3 Paper

Generating text given a prompt is also referred to as **conditional generation**, since our model is generating some output *conditioned* on some input.

GPTs are not limited to NLP tasks. You can condition the model on anything you want. For example, you can turn a GPT into a chatbot (i.e. [ChatGPT](#)) by conditioning it on the conversation history. You can also further condition the chatbot to behave a certain way by prepending the prompt with some kind of description (i.e. "You are a chatbot. Be polite, speak in full sentences, don't say harmful things, etc ..."). Conditioning the model like this can even give your [chatbot a persona](#). However, this is not robust, you can still "[jailbreak](#)" the [model and make it misbehave](#).

With that out of the way, let's finally get to the actual implementation.

Setup

Clone the repository for this tutorial:

```
git clone https://github.com/jaymody/picoGPT
cd picoGPT
```

Then let's install our dependencies:

```
pip install -r requirements.txt
```

Note: This code was tested with [Python 3.9.10](#).

A quick breakdown of each of the files:

- `encoder.py` contains the code for OpenAI's BPE Tokenizer, taken straight from their [gpt-2 repo](#).
- `utils.py` contains the code to download and load the GPT-2 model weights, tokenizer, and hyperparameters.
- `gpt2.py` contains the actual GPT model and generation code, which we can run as a python script.
- `gpt2_pico.py` is the same as `gpt2.py`, but in even fewer lines of code. Why? Because why not.

We'll be reimplementing `gpt2.py` from scratch, so let's delete it and recreate it as an empty file:

```
rm gpt2.py
touch gpt2.py
```

As a starting point, paste the following code into `gpt2.py`:

```
import numpy as np

def gpt2(inputs, wte, wpe, blocks, ln_f, n_head):
    pass # TODO: implement this

def generate(inputs, params, n_head, n_tokens_to_generate):
    from tqdm import tqdm

    for _ in tqdm(range(n_tokens_to_generate), "generating"): # auto-regressive decode
        logits = gpt2(inputs, *params, n_head=n_head) # model forward pass
        next_id = np.argmax(logits[-1]) # greedy sampling
        inputs.append(int(next_id)) # append prediction to input
```

```

        return inputs[len(inputs) - n_tokens_to_generate :] # only return generated ids

def main(prompt: str, n_tokens_to_generate: int = 40, model_size: str = "124M", models_dir: str = "models"):
    from utils import load_encoder_hparams_and_params

    # load encoder, hparams, and params from the released open-ai gpt-2 files
    encoder, hparams, params = load_encoder_hparams_and_params(model_size, models_dir)

    # encode the input string using the BPE tokenizer
    input_ids = encoder.encode(prompt)

    # make sure we are not surpassing the max sequence length of our model
    assert len(input_ids) + n_tokens_to_generate < hparams["n_ctx"]

    # generate output ids
    output_ids = generate(input_ids, params, hparams["n_head"], n_tokens_to_generate)

    # decode the ids back into a string
    output_text = encoder.decode(output_ids)

    return output_text

if __name__ == "__main__":
    import fire

    fire.Fire(main)

```

Breaking down each of the 4 sections:

1. The `gpt2` function is the actual GPT code we'll be implementing. You'll notice that the function signature includes some extra stuff in addition to `inputs`:
 - `wte`, `wpe`, `blocks`, and `ln_f` the parameters of our model.
 - `n_head` is a hyperparameter that is needed during the forward pass.
2. The `generate` function is the autoregressive decoding algorithm we saw earlier. We use greedy sampling for simplicity. `tqdm` is a progress bar to help us visualize the decoding process as it generates tokens one at a time.
3. The `main` function handles:
 1. Loading the tokenizer (`encoder`), model weights (`params`), and hyperparameters (`hparams`)
 2. Encoding the input prompt into token IDs using the tokenizer
 3. Calling the generate function
 4. Decoding the output IDs into a string
4. `fire.Fire(main)` just turns our file into a CLI application, so we can eventually run our code with: `python gpt2.py "some prompt here"`

Let's take a closer look at `encoder`, `hparams`, and `params`, in a notebook, or an interactive python session, run:

```

from utils import load_encoder_hparams_and_params
encoder, hparams, params = load_encoder_hparams_and_params("124M", "models")

```

This will [download the necessary model and tokenizer files to `models/124M`](#) and `load encoder`, `hparams`, and `params` into our code.

Encoder

`encoder` is the BPE tokenizer used by GPT-2:

```

>>> ids = encoder.encode("Not all heroes wear capes.")
>>> ids
[3673, 477, 10281, 5806, 1451, 274, 13]

>>> encoder.decode(ids)
"Not all heroes wear capes."

```

Using the vocabulary of the tokenizer (stored in `encoder.decoder`), we can take a peek at what the actual tokens look like:

```

>>> [encoder.decoder[i] for i in ids]
[ 'Not', 'all', 'heroes', 'wear', 'capes', '.']

```

Notice, sometimes our tokens are words (e.g. `Not`), sometimes they are words but with a space in front of them (e.g. `All`, the `G` represents a space), sometimes there are part of a word (e.g. `capes` is split into `cap` and `es`), and sometimes they are punctuation (e.g. `.`).

One nice thing about BPE is that it can encode any arbitrary string. If it encounters something that is not present in the vocabulary, it just breaks it down into substrings it does understand:

```
>>> [encoder.decoder[i] for i in encoder.encode("zjqf1")]
['z', 'j', 'q', 'f1']
```

We can also check the size of the vocabulary:

```
>>> len(encoder.decoder)
50257
```

The vocabulary, as well as the byte-pair merges which determines how strings are broken down, is obtained by *training* the tokenizer. When we load the tokenizer, we're loading the already trained vocab and byte-pair merges from some files, which were downloaded alongside the model files when we ran `load_encoder_hparams_and_params`. See `models/124M/encoder.json` (the vocabulary) and `models/124M/vocab.bpe` (byte-pair merges).

Hyperparameters

`hparams` is a dictionary that contains the hyper-parameters of our model:

```
>>> hparams
{
    "n_vocab": 50257, # number of tokens in our vocabulary
    "n_ctx": 1024, # maximum possible sequence length of the input
    "n_embd": 768, # embedding dimension (determines the "width" of the network)
    "n_head": 12, # number of attention heads (n_embd must be divisible by n_head)
    "n_layer": 12 # number of layers (determines the "depth" of the network)
}
```

We'll use these symbols in our code's comments to show the underlying shape of things. We'll also use `n_seq` to denote the length of our input sequence (i.e. `n_seq = len(inputs)`).

Parameters

`params` is a nested json dictionary that hold the trained weights of our model. The leaf nodes of the json are NumPy arrays. If we print `params`, replacing the arrays with their shapes, we get:

```
>>> import numpy as np
>>> def shape_tree(d):
>>>     if isinstance(d, np.ndarray):
>>>         return list(d.shape)
>>>     elif isinstance(d, list):
>>>         return [shape_tree(v) for v in d]
>>>     elif isinstance(d, dict):
>>>         return {k: shape_tree(v) for k, v in d.items()}
>>>     else:
>>>         ValueError("uh oh")
>>>
>>> print(shape_tree(params))
{
    "wpe": [1024, 768],
    "wte": [50257, 768],
    "ln_f": {"b": [768], "g": [768]},
    "blocks": [
        {
            "attn": {
                "c_attn": {"b": [2304], "w": [768, 2304]},
                "c_proj": {"b": [768], "w": [768, 768]},
            },
            "ln_1": {"b": [768], "g": [768]},
            "ln_2": {"b": [768], "g": [768]},
            "mlp": {
                "c_fc": {"b": [3072], "w": [768, 3072]},
                "c_proj": {"b": [768], "w": [3072, 768]},
            },
            ...
        } # repeat for n_layers
    ]
}
```

These are loaded from the original OpenAI tensorflow checkpoint:

```
>>> import tensorflow as tf
>>> tf_ckpt_path = tf.train.latest_checkpoint("models/124M")
>>> for name, _ in tf.train.list_variables(tf_ckpt_path):
>>>     arr = tf.train.load_variable(tf_ckpt_path, name).squeeze()
>>>     print(f"(name): {arr.shape}")
model/h0/attn/c_attn/b: (2304,)
model/h0/attn/c_attn/w: (768, 2304)
model/h0/attn/c_proj/b: (768,)
model/h0/attn/c_proj/w: (768, 768)
model/h0/ln_1/b: (768,)
model/h0/ln_1/g: (768,)
model/h0/ln_2/b: (768,)
model/h0/ln_2/g: (768,)
```

```

model/h0/mlp/c_fc/b: (3072,)
model/h0/mlp/c_fc/w: (768, 3072)
model/h0/mlp/c_proj/b: (768,)
model/h0/mlp/c_proj/w: (3072, 768)
model/h1/attn/c_attn/b: (2304,)
model/h1/attn/c_attn/w: (768, 2304)
...
model/h9/mlp/c_proj/b: (768,)
model/h9/mlp/c_proj/w: (3072, 768)
model/ln_f/b: (768,)
model/ln_f/g: (768,)
model/wpe: (1024, 768)
model/wte: (50257, 768)

```

The [following code](#) converts the above tensorflow variables into our `params` dictionary.

For reference, here's the shapes of `params` but with the numbers replaced by the `hparams` they represent:

```

{
    "wpe": [n_ctx, n_embd],
    "wte": [n_vocab, n_embd],
    "ln_f": {"b": [n_embd], "g": [n_embd]},
    "blocks": [
        {
            "attn": {
                "c_attn": {"b": [3*n_embd], "w": [n_embd, 3*n_embd]},
                "c_proj": {"b": [n_embd], "w": [n_embd, n_embd]},
            },
            "ln_1": {"b": [n_embd], "g": [n_embd]},
            "ln_2": {"b": [n_embd], "g": [n_embd]},
            "mlp": {
                "c_fc": {"b": [4*n_embd], "w": [n_embd, 4*n_embd]},
                "c_proj": {"b": [n_embd], "w": [4*n_embd, n_embd]},
            },
            ...
        } # repeat for n_layers
    ]
}

```

You'll probably want to come back to reference this dictionary to check the shape of the weights as we implement our GPT. We'll match the variable names in our code with the keys of this dictionary for consistency.

Basic Layers

Last thing before we get into the actual GPT architecture itself, let's implement some of the more basic neural network layers that are non-specific to GPTs.

GELU

The non-linearity (**activation function**) of choice for GPT-2 is [GELU \(Gaussian Error Linear Units\)](#), an alternative for ReLU:

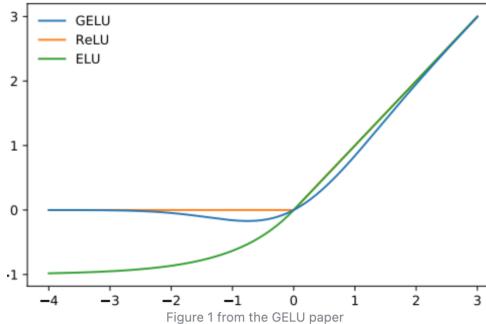


Figure 1 from the GELU paper

It is approximated by the following function:

```

def gelu(x):
    return 0.5 * x * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * x**3)))

```

Like ReLU, GELU operates element-wise on the input:

```

>>> gelu(np.array([[1, 2], [-2, 0.5]]))
array([[ 0.84119,  1.9546 ],
       [-0.0454 ,  0.34571]])

```

Softmax

Good ole softmax:

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

```
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)
```

We use the `max(x)` trick for numerical stability.

Softmax is used to convert a set of real numbers (between $-\infty$ and ∞) to probabilities (between 0 and 1, with the numbers all summing to 1). We apply `softmax` over the last axis of the input.

```
>>> x = softmax(np.array([[2, 100], [-5, 0]]))
>>> x
array([[0.00034, 0.99966],
       [0.26894, 0.73106]])
>>> x.sum(axis=-1)
array([1., 1.])
```

Layer Normalization

Layer normalization standardizes values to have a mean of 0 and a variance of 1:

$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2}} + \beta$$

where μ is the mean of x , σ^2 is the variance of x , and γ and β are learnable parameters.

```
def layer_norm(x, g, b, eps: float = 1e-5):
    mean = np.mean(x, axis=-1, keepdims=True)
    variance = np.var(x, axis=-1, keepdims=True)
    x = (x - mean) / np.sqrt(variance + eps) # normalize x to have mean=0 and var=1
    return g * x + b # scale and offset with gamma/beta params
```

Layer normalization ensures that the inputs for each layer are always within a consistent range, which is supposed to speed up and stabilize the training process. Like [Batch Normalization](#), the normalized output is then scaled and offset with two learnable vectors gamma and beta. The small epsilon term in the denominator is used to avoid a division by zero error.

Layer norm is used instead of batch norm in the transformer for [various reasons](#). The differences between various normalization techniques is outlined [in this excellent blog post](#).

We apply layer normalization over the last axis of the input.

```
>>> x = np.array([[2, 2, 3], [-5, 0, 1]])
>>> x = layer_norm(x, g=np.ones(x.shape[-1]), b=np.zeros(x.shape[-1]))
>>> x
array([[-0.70709, -0.70709, 1.41418],
       [-1.397, 0.508, 0.889]])
>>> x.var(axis=-1)
array([0.99996, 1.]) # floating point shenanigans
>>> x.mean(axis=-1)
array([-0., -0.])
```

Linear

Your standard matrix multiplication + bias:

```
def linear(x, w, b): # [m, in], [in, out], [out] -> [m, out]
    return x @ w + b
```

Linear layers are often referred to as [projections](#) (since they are projecting from one vector space to another vector space).

```
>>> x = np.random.normal(size=(64, 784)) # input dim = 784, batch/sequence dim = 64
>>> w = np.random.normal(size=(784, 10)) # output dim = 10
>>> b = np.random.normal(size=(10,))
>>> x.shape # shape before linear projection
(64, 784)
>>> linear(x, w, b).shape # shape after linear projection
(64, 10)
```

GPT Architecture

The GPT architecture follows that of the [transformer](#):

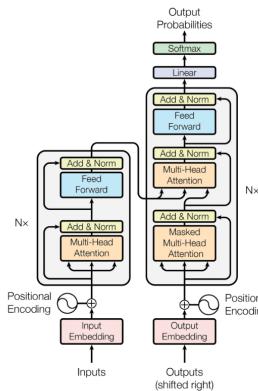
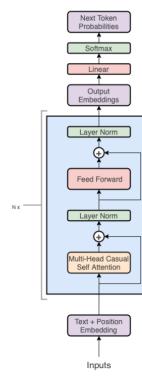


Figure 1 from Attention is All You Need

But uses only the decoder stack (the right part of the diagram):



GPT Architecture

Note, the middle "cross-attention" layer is also removed since we got rid of the encoder.

At a high level, the GPT architecture has three sections:

- Text + positional **embeddings**
- A transformer **decoder stack**
- A **projection to vocab** step

In code, it looks like this:

```
def gpt2(inputs, wte, wpe, blocks, ln_f, n_head):  # [n_seq] -> [n_seq, n_vocab]
    # token + positional embeddings
    x = wte[inputs] + wpe[range(len(inputs))]  # [n_seq] -> [n_seq, n_embd]

    # forward pass through n_layer transformer blocks
    for block in blocks:
        x = transformer_block(x, **block, n_head=n_head)  # [n_seq, n_embd] -> [n_seq, n_embd]

    # projection to vocab
    x = layer_norm(x, **ln_f)  # [n_seq, n_embd] -> [n_seq, n_embd]
    return x @ wte.T  # [n_seq, n_embd] -> [n_seq, n_vocab]
```

Let's break down each of these three sections into more detail.

Embeddings

Token Embeddings

Token IDs by themselves are not very good representations for a neural network. For one, the relative magnitudes of the token IDs falsely communicate information (for example, if `Apple` = 5 and `Table` = 10 in our vocab, then we are implying that $2 * \text{Table} = \text{Apple}$). Secondly, a single number is not a lot of *dimensionality* for a neural network to work with.

To address these limitations, we'll take advantage of [word vectors](#), specifically via a learned embedding matrix:

```
wte[inputs] # [n_seq] -> [n_seq, n_embd]
```

Recall, `wte` is a $[n_{vocab}, n_{embd}]$ matrix. It acts as a lookup table, where the i th row in the matrix corresponds to the learned vector for the i th token in our vocabulary. `wte[inputs]` uses [integer array indexing](#) to retrieve the vectors corresponding to each token in our input.

Like any other parameter in our network, `wte` is learned. That is, it is randomly initialized at the start of training and then updated via gradient descent.

Positional Embeddings

One quirk of the transformer architecture is that it doesn't take into account position. That is, if we randomly shuffled our input and then accordingly unshuffled the output, the output would be the same as if we never shuffled the input in the first place (the ordering of inputs doesn't have any effect on the output).

Of course, the ordering of words is a crucial part of language (duh), so we need some way to encode positional information into our inputs. For this, we can just use another learned embedding matrix:

```
wpe[range(len(inputs))] # [n_seq] -> [n_seq, n_embd]
```

Recall, `wpe` is a $[n_{ctx}, n_{embd}]$ matrix. The i th row of the matrix contains a vector that encodes information about the i th position in the input. Similar to `wte`, this matrix is learned during gradient descent.

Notice, this restricts our model to a maximum sequence length of `n_ctx`.^[4] That is, `len(inputs) <= n_ctx` must hold.

Combined

We can add our token and positional embeddings to get a combined embedding that encodes both token and positional information.

```
# token + positional embeddings
x = wte[inputs] + wpe[range(len(inputs))] # [n_seq] -> [n_seq, n_embd]

# x[i] represents the word embedding for the ith word + the positional
# embedding for the ith position
```

Decoder Stack

This is where all the magic happens and the "deep" in deep learning comes in. We pass our embedding through a stack of `n_layer` transformer decoder blocks.

```
# forward pass through n_layer transformer blocks
for block in blocks:
    x = transformer_block(x, **block, n_head=n_head) # [n_seq, n_embd] -> [n_seq, n_e
```

Stacking more layers is what allows us to control how *deep* our network is. GPT-3 for example, has a [whopping 96 layers](#). On the other hand, choosing a larger `n_embd` value allows us to control how *wide* our network is (for example, GPT-3 uses an embedding size of 12288).

Projection to Vocab

In our final step, we project the output of the final transformer block to a probability distribution over our vocab:

```
# projection to vocab
x = layer_norm(x, **ln_f) # [n_seq, n_embd] -> [n_seq, n_embd]
return x @ wte.T # [n_seq, n_embd] -> [n_seq, n_vocab]
```

Couple things to note here:

1. We first pass `x` through a **final layer normalization** layer before doing the projection to vocab. This is specific to the GPT-2 architecture (this is not present in the original GPT and Transformer papers).
2. We are **reusing the embedding matrix** `wte` for the projection. Other GPT implementations may choose to use a separate learned weight matrix for the projection, however sharing the embedding matrix has a couple of advantages:
 - You save some parameters (although at GPT-3 scale, this is negligible).
 - Since the matrix is both responsible for mapping both to words and *from* words, in theory, it *may* learn a richer representation compared to having two separate matrixes.
3. We **don't apply softmax** at the end, so our outputs will be `logits` instead of probabilities between 0 and 1. This is done for several reasons:
 - `softmax` is **monotonic**, so for greedy sampling `np.argmax(logits)` is equivalent to `np.argmax(softmax(logits))` making `softmax` redundant
 - `softmax` is **irreversible**. meaning we can always do from `logits` to `probabilities`

- by applying `softmax`, but we can't go back to `logits` from `probabilities`, so for maximum flexibility, we output the `logits`
- Numerically stability (for example, to compute cross entropy loss, taking `log(softmax(logits))` is numerically unstable compared to `log_softmax(logits)`)

The projection to vocab step is also sometimes called the **language modeling head**. What does "head" mean? Once your GPT is pre-trained, you can swap out the language modeling head with some other kind of projection, like a **classification head** for fine-tuning the model on some classification task. So your model can have multiple heads, kind of like a [hydra](#).

So that's the GPT architecture at a high level, let's actually dig a bit deeper into what the decoder blocks are doing.

Decoder Block

The transformer decoder block consists of two sublayers:

1. Multi-head causal self attention
2. Position-wise feed forward neural network

```
def transformer_block(x, mlp, attn, ln_1, ln_2, n_head): # [n_seq, n_embd] -> [n_seq, n_embd]
    # multi-head causal self attention
    x = x + mha(layer_norm(x, **ln_1), **attn, n_head=n_head) # [n_seq, n_embd] -> [n_seq, n_embd]

    # position-wise feed forward network
    x = x + ffn(layer_norm(x, **ln_2), **mlp) # [n_seq, n_embd] -> [n_seq, n_embd]

    return x
```

Each sublayer utilizes layer normalization on their inputs as well as a residual connection (i.e. add the input of the sublayer to the output of the sublayer).

Some things to note:

1. **Multi-head causal self attention** is what facilitates the communication between the inputs. Nowhere else in the network does the model allow inputs to "see" each other. The embeddings, position-wise feed forward network, layer norms, and projection to vocab all operate on our inputs position-wise. Modeling relationships between inputs is tasked solely to attention.
2. The **Position-wise feed forward neural network** is just a regular 2 layer fully connected neural network. This just adds a bunch of learnable parameters for our model to work with to facilitate learning.
3. In the original transformer paper, layer norm is placed on the output `layer_norm(x + sublayer(x))` while we place layer norm on the input `x + sublayer(layer_norm(x))` to match GPT-2. This is referred to as **pre-norm** and has been shown to be **important in improving the performance of the transformer**.
4. **Residual connections** (popularized by [ResNet](#)) serve a couple of different purposes:
 1. Makes it easier to optimize neural networks that are deep (i.e. networks that have lots of layers). The idea here is that we are providing "shortcuts" for the gradients to flow back through the network, making it easier to optimize the earlier layers in the network.
 2. Without residual connections, deeper models see a degradation in performance when adding more layers (possibly because it's hard for the gradients to flow all the way back through a deep network without losing information). Residual connections seem to give a bit of an accuracy boost for deeper networks.
 3. Can help with the [vanishing/exploding gradients problem](#).

Let's dig a little deeper into the 2 sublayers.

Position-wise Feed Forward Network

This is just a simple multi-layer perceptron with 2 layers:

```
def ffn(x, c_fc, c_proj): # [n_seq, n_embd] -> [n_seq, n_embd]
    # project up
    a = gelu(linear(x, **c_fc)) # [n_seq, n_embd] -> [n_seq, 4*n_embd]

    # project back down
    x = linear(a, **c_proj) # [n_seq, 4*n_embd] -> [n_seq, n_embd]

    return x
```

Nothing super fancy here, we just project from `n_embd` up to a higher dimension `4*n_embd` and then back down to `n_embd` [5].

Recall, from our `params` dictionary, that our `mlp` params look like this:

```
"mlp": {
    "c_fc": {"b": [4*n_embd], "w": [n_embd, 4*n_embd]},
    "c_proj": {"b": [n_embd], "w": [4*n_embd, n_embd]},
```

```
}
```

Multi-Head Causal Self Attention

This layer is probably the most difficult part of the transformer to understand. So let's work our way up to "Multi-Head Causal Self Attention" by breaking each word down into its own section:

1. Attention
2. Self
3. Causal
4. Multi-Head

Attention

I have another [blog post](#) on this topic, where we derive the scaled dot product equation proposed in the [original transformer paper](#) from the ground up:

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

As such, I'm going to skip an explanation for attention in this post. You can also reference [Lilian Weng's Attention? Attention!](#) and [Jay Alammar's The Illustrated Transformer](#) which are also great explanations for attention.

We'll just adapt our attention implementation from my blog post:

```
def attention(q, k, v): # [n_q, d_k], [n_k, d_k], [n_k, d_v] -> [n_q, d_v]
    return softmax(q @ k.T / np.sqrt(q.shape[-1])) @ v
```

Self

When `q`, `k`, and `v` all come from the same source, we are performing [self-attention](#) (i.e. letting our input sequence attend to itself):

```
def self_attention(x): # [n_seq, n_embd] -> [n_seq, n_embd]
    return attention(q=x, k=x, v=x)
```

For example, if our input is "Jay went to the store, he bought 10 apples.", we would be letting the word "he" attend to all the other words, including "Jay", meaning the model can learn to recognize that "he" is referring to "Jay".

We can enhance self attention by introducing projections for `q`, `k`, `v` and the attention output:

```
def self_attention(x, w_k, w_q, w_v, w_proj): # [n_seq, n_embd] -> [n_seq, n_embd]
    # qkv projections
    q = x @ w_k # [n_seq, n_embd] @ [n_embd, n_embd] -> [n_seq, n_embd]
    k = x @ w_q # [n_seq, n_embd] @ [n_embd, n_embd] -> [n_seq, n_embd]
    v = x @ w_v # [n_seq, n_embd] @ [n_embd, n_embd] -> [n_seq, n_embd]

    # perform self attention
    x = attention(q, k, v) # [n_seq, n_embd] -> [n_seq, n_embd]

    # out projection
    x = x @ w_proj # [n_seq, n_embd] @ [n_embd, n_embd] -> [n_seq, n_embd]

    return x
```

This enables our model to learn a mapping for `q`, `k`, and `v` that best helps attention distinguish relationships between inputs.

We can reduce the number of matrix multiplication from 4 to just 2 if we combine `w_q`, `w_k` and `w_v` into a single matrix `w_fc`, perform the projection, and then split the result:

```
def self_attention(x, w_fc, w_proj): # [n_seq, n_embd] -> [n_seq, n_embd]
    # qkv projections
    x = x @ w_fc # [n_seq, n_embd] @ [n_embd, 3*n_embd] -> [n_seq, 3*n_embd]

    # split into qkv
    q, k, v = np.split(x, 3, axis=-1) # [n_seq, 3*n_embd] -> 3 of [n_seq, n_embd]

    # perform self attention
    x = attention(q, k, v) # [n_seq, n_embd] -> [n_seq, n_embd]

    # out projection
    x = x @ w_proj # [n_seq, n_embd] @ [n_embd, n_embd] = [n_seq, n_embd]

    return x
```

This is a bit more efficient as modern accelerators (GPUs) can take better advantage of one

dimensional memory access patterns when processing 3 parallel vectors at once.

large matrix multiplication rather than 3 separate small ones happening sequentially.

Finally, we add bias vectors to match the implementation of GPT-2, use our `linear` function, and rename our parameters to match our `params` dictionary:

```
def self_attention(x, c_attn, c_proj): # [n_seq, n_embd] -> [n_seq, n_embd]
    # qkv projections
    x = linear(x, **c_attn) # [n_seq, n_embd] -> [n_seq, 3*n_embd]

    # split into qkv
    q, k, v = np.split(x, 3, axis=-1) # [n_seq, 3*n_embd] -> 3 of [n_seq, n_embd]

    # perform self attention
    x = attention(q, k, v) # [n_seq, n_embd] -> [n_seq, n_embd]

    # out projection
    x = linear(x, **c_proj) # [n_seq, n_embd] @ [n_embd, n_embd] = [n_seq, n_embd]

    return x
```

Recall, from our `params` dictionary, our `attn` params look like this:

```
"attn": {
    "c_attn": {"b": [3*n_embd], "w": [n_embd, 3*n_embd]},
    "c_proj": {"b": [n_embd], "w": [n_embd, n_embd]},
},
```

Causal

There is a bit of an issue with our current self-attention setup, our inputs can see into the future! For example, if our input is `["not", "all", "heroes", "wear", "capes"]`, during self attention we are allowing "wear" to see "capes". This means our output probabilities for "wear" will be biased since the model already knows the correct answer is "capes". This is no good since our model will just learn that the correct answer for input i can be taken from input $i + 1$.

To prevent this, we need to somehow modify our attention matrix to *hide* or *mask* our inputs from being able to see into the future. For example, let's pretend our attention matrix looks like this:

	not	all	heroes	wear	capes
not	0.116	0.159	0.055	0.226	0.443
all	0.180	0.397	0.142	0.106	0.175
heroes	0.156	0.453	0.028	0.129	0.234
wear	0.499	0.055	0.133	0.017	0.295
capes	0.089	0.290	0.240	0.228	0.153

Each row corresponds to a query and the columns to a key. In this case, looking at the row for "wear", you can see that it is attending to "capes" in the last column with a weight of 0.295. To prevent this, we want to set that entry to 0.0:

	not	all	heroes	wear	capes
not	0.116	0.159	0.055	0.226	0.443
all	0.180	0.397	0.142	0.106	0.175
heroes	0.156	0.453	0.028	0.129	0.234
wear	0.499	0.055	0.133	0.017	0.
capes	0.089	0.290	0.240	0.228	0.153

In general, to prevent all the queries in our input from looking into the future, we set all positions i, j where $j > i$ to 0:

	not	all	heroes	wear	capes
not	0.116	0.	0.	0.	0.
all	0.180	0.397	0.	0.	0.
heroes	0.156	0.453	0.028	0.	0.
wear	0.499	0.055	0.133	0.017	0.
capes	0.089	0.290	0.240	0.228	0.153

We call this **masking**. One issue with our above masking approach is our rows no longer sum to 1 (since we are setting them to 0 after the `softmax` has been applied). To make sure our rows still sum to 1, we need to modify our attention matrix before the `softmax` is applied.

This can be achieved by setting entries that are to be masked to $-\infty$ prior to the `softmax` [6]:

```
def attention(q, k, v, mask): # [n_q, d_k], [n_k, d_k], [n_k, d_v], [n_q, n_k] -> [n_q, d_v]
    return softmax(q @ k.T / np.sqrt(q.shape[-1]) + mask) @ v
```

where `mask` is the matrix (for `n_seq=5`):

```

0 -1e10 -1e10 -1e10 -1e10
0  0   -1e10 -1e10 -1e10
0  0   0     -1e10 -1e10
0  0   0     0     -1e10
0  0   0     0     0

```

We use `-1e10` instead of `-np.inf` as `-np.inf` can cause `nans`.

Adding `mask` to our attention matrix instead of just explicitly setting the values to `-1e10` works because practically, any number plus `-inf` is just `-inf`.

We can compute the `mask` matrix in NumPy with `(1 - np.tri(n_seq)) * -1e10`.

Putting it all together, we get:

```

def attention(q, k, v, mask): # [n_q, d_k], [n_k, d_k], [n_q, d_v], [n_q, n_k] -> [n_q, n_v]
    return softmax(q @ k.T / np.sqrt(q.shape[-1]) + mask) @ v

def causal_self_attention(x, c_attn, c_proj): # [n_seq, n_embd] -> [n_seq, n_embd]
    # qkv projections
    x = linear(x, **c_attn) # [n_seq, n_embd] -> [n_seq, 3*n_embd]

    # split into qkv
    q, k, v = np.split(x, 3, axis=-1) # [n_seq, 3*n_embd] -> 3 of [n_seq, n_embd]

    # causal mask to hide future inputs from being attended to
    causal_mask = (1 - np.tri(x.shape[0]), dtype=x.dtype) * -1e10 # [n_seq, n_seq]

    # perform causal self attention
    x = attention(q, k, v, causal_mask) # [n_seq, n_embd] -> [n_seq, n_embd]

    # out projection
    x = linear(x, **c_proj) # [n_seq, n_embd] @ [n_embd, n_embd] = [n_seq, n_embd]

    return x

```

Multi-Head

We can further improve our implementation by performing `n_head` separate attention computations, splitting our queries, keys, and values into `heads`:

```

def mha(x, c_attn, c_proj, n_head): # [n_seq, n_embd] -> [n_seq, n_embd]
    # qkv projection
    x = linear(x, **c_attn) # [n_seq, n_embd] -> [n_seq, 3*n_embd]

    # split into qkv
    qkv = np.split(x, 3, axis=-1) # [n_seq, 3*n_embd] -> [3, n_seq, n_embd]

    # split into heads
    qkv_heads = list(map(lambda x: np.split(x, n_head, axis=-1), qkv)) # [3, n_seq, n_head, n_embd]

    # causal mask to hide future inputs from being attended to
    causal_mask = (1 - np.tri(x.shape[0]), dtype=x.dtype) * -1e10 # [n_seq, n_seq]

    # perform attention over each head
    out_heads = [attention(q, k, v, causal_mask) for q, k, v in zip(*qkv_heads)] # [3, n_seq, n_head]

    # merge heads
    x = np.hstack(out_heads) # [n_head, n_seq, n_embd/n_head] -> [n_seq, n_embd]

    # out projection
    x = linear(x, **c_proj) # [n_seq, n_embd] -> [n_seq, n_embd]

    return x

```

There are three steps added here:

1. Split `q, k, v` into `n_head` heads:

```

# split into heads
qkv_heads = list(map(lambda x: np.split(x, n_head, axis=-1), qkv)) # [3, n_seq, n_head, n_embd]

```

2. Compute attention for each head:

```

# perform attention over each head
out_heads = [attention(q, k, v) for q, k, v in zip(*qkv_heads)] # [n_head, 3, n_seq]

```

3. Merge the outputs of each head:

```

# merge heads
x = np.hstack(out_heads) # [n_head, n_seq, n_embd/n_head] -> [n_seq, n_embd]

```

Notice, this reduces the dimension from `n_embd` to `n_embd/n_head` for each attention computation. This is a tradeoff. For reduced dimensionality, our model gets additional subspaces to work when modeling relationships via attention. For example, maybe one attention head is responsible for connecting pronouns to the person the pronoun is referencing. Maybe another might be responsible for grouping sentences by periods. Another could simply be identifying which words are entities, and which are not. Although, it's probably just another neural network black box.

The code we wrote performs the attention computations over each head sequentially in a loop (one at a time), which is not very efficient. In practice, you'd want to do these in parallel. For simplicity, we'll just leave this sequential.

With that, we're finally done our GPT implementation! Now, all that's left to do is put it all together and run our code.

Putting it All Together

Putting everything together, we get [gpt2.py](#), which in its entirety is a mere 120 lines of code ([60 lines if you remove comments and whitespace](#)).

We can test our implementation with:

```
python gpt2.py \
    "Alan Turing theorized that computers would one day become" \
    --n_tokens_to_generate 8
```

which gives the output:

```
the most powerful machines on the planet.
```

It works!!!

We can test that our implementation gives identical results to [OpenAI's official GPT-2 repo](#) using the following [Dockerfile](#) (Note: this won't work on M1 Macbooks because of tensorflow shenanigans and also warning, it downloads all 4 GPT-2 model sizes, which is a lot of GBs of stuff to download):

```
docker build -t "openai-gpt-2" "https://gist.github.com/jaymody/9054ca64ee"
docker run -dt "openai-gpt-2" --name "openai-gpt-2-app"
docker exec -it "openai-gpt-2-app" /bin/bash -c 'python3 src/interactive_conditional_s
# paste "Alan Turing theorized that computers would one day become" when prompted
```

which should give an identical result:

```
the most powerful machines on the planet.
```

What Next?

This implementation is cool and all, but it's missing a ton of bells and whistles:

GPU/TPU Support

Replace NumPy with [JAX](#):

```
import jax.numpy as np
```

That's it. You can now use the code with GPUs and even [TPUs](#)! Just make sure you [install JAX correctly](#).

Backpropagation

Again, if we replace NumPy with [JAX](#):

```
import jax.numpy as np
```

Then computing the gradients is as easy as:

```
def lm_loss(params, inputs, n_head) -> float:
    x, y = inputs[:-1], inputs[1:]
    output = gpt2(x, *params, n_head=n_head)
    loss = -np.mean(-np.log(output[y]))
```

```
    loss = lm_loss(params, inputs, n_head)
    return loss

grads = jax.grad(lm_loss)(params, inputs, n_head)
```

Batching

Once again, if we replace NumPy with [JAX](#)^[7]:

```
import jax.numpy as np
```

Then, making our `gpt2` function batched is as easy as:

```
gpt2_batched = jax.vmap(gpt2, in_axes=[0, None, None, None, None, None])
gpt2_batched(batched_inputs) # [batch, seq_len] -> [batch, seq_len, vocab]
```

Inference Optimization

Our implementation is quite inefficient. The quickest and most impactful optimization you can make (outside of GPU + batching support) would be to implement a [kv cache](#). Also, we implemented our attention head computations sequentially, when we should really be doing it in parallel^[8].

There's many many more inference optimizations. I recommend [Lillian Weng's Large Transformer Model Inference Optimization](#) and [Kiply's Transformer Inference Arithmetic](#) as a starting point.

Training

Training a GPT is pretty standard for a neural network (gradient descent w.r.t a loss function). Of course, you also need to use the standard bag of tricks when training a GPT (i.e. use the Adam optimizer, find the optimal learning rate, regularization via dropout and/or weight decay, use a learning rate scheduler, use the correct weight initialization, batching, etc ...).

The real secret sauce to training a good GPT model is the ability to **scale the data and the model**, which is where the real challenge is.

For scaling data, you'll want a corpus of text that is big, high quality, and diverse.

- Big means billions of tokens (terabytes of data). For example, check out [The Pile](#), which is an open source pre-training dataset for large language models.
- High quality means you want to filter out duplicate examples, unformatted text, incoherent text, garbage text, etc ...
- Diverse means varying sequence lengths, about lots of different topics, from different sources, with differing perspectives, etc ... Of course, if there are any biases in the data, it will reflect in the model, so you need to be careful of that as well.

Scaling the model to billions of parameters involves a cr*p ton of engineering (and money lol). Training frameworks can get [absurdly long and complex](#). A good place to start would be [Lillian Weng's How to Train Really Large Models on Many GPUs](#). On the topic there's also the NVIDIA's Megatron Framework, Cohere's Training Framework, Google's PALM, the open source [mesh-transformer-jax](#) (used to train EleutherAI's open source models), and [many many more](#).

Evaluation

Oh boy, how does one even evaluate LLMs? Honestly, it's really hard problem. [HELM](#) is pretty comprehensive and a good place to start, but you should always be skeptical of [benchmarks and evaluation metrics](#).

Architecture Improvements

I recommend taking a look at [Phil Wang's X-Transformer's](#). It has the latest and greatest research on the transformer architecture. [This paper](#) is also a pretty good summary (see Table 1). Facebook's recent [LLAMA paper](#) is also probably a good reference for standard architecture improvements (as of February 2023).

Stopping Generation

Our current implementation requires us to specify the exact number of tokens we'd like to generate ahead of time. This is not a very good approach as our generations end up being too long, too short, or cutoff mid-sentence.

To resolve this, we can introduce a special **end of sentence (EOS) token**. During pre-training, we append the EOS token to the end of our input (i.e. `tokens = ["not", "all", "heroes", "wear", "capes", ".", "<|EOS|>"]`). During generation, we simply stop whenever we encounter the EOS token (or if we hit some maximum sequence length):

```

def generate(inputs, eos_id, max_seq_len):
    prompt_len = len(inputs)
    while inputs[-1] != eos_id and len(inputs) < max_seq_len:
        output = gpt(inputs)
        next_id = np.argmax(output[-1])
        inputs.append(int(next_id))
    return inputs[prompt_len:]

```

GPT-2 was not pre-trained with an EOS token, so we can't use this approach in our code, but most LLMs nowadays use an EOS token.

Unconditional Generation

Generating text with our model requires us to **condition** it with a prompt. However, we can also make our model perform **unconditional generation**, where the model generates text without any kind of input prompt.

This is achieved by prepending a special **beginning of sentence (BOS) token** to the start of the input during pre-training (i.e. tokens = ["<|BOS|>", "not", "all", "heroes", "wear", "capes", "."]). Then, to generate text unconditionally, we input a list that contains just the BOS token:

```

def generate_unconditioned(bos_id, n_tokens_to_generate):
    inputs = [bos_id]
    for _ in range(n_tokens_to_generate):
        output = gpt(inputs)
        next_id = np.argmax(output[-1])
        inputs.append(int(next_id))
    return inputs[1:]

```

GPT-2 was pre-trained with a BOS token (which is confusingly named `<|endoftext|>`), so running unconditional generation with our implementation is as easy as changing the [following line](#) to:

```
input_ids = encoder.encode(prompt) if prompt else [encoder.encoder["<|endoftext|>"]]
```

And then running:

```
python gpt2.py ""
```

Which generates:

```
The first time I saw the new version of the game, I was so excited. I was so excited t
```

Because we are using greedy sampling, the output is not very good (repetitive) and is deterministic (i.e. same output each time we run the code). To get generations that are both higher quality and non-deterministic, we'd need to sample directly from the distribution (ideally after applying something like top-p).

Unconditional generation is not particularly useful, but it's a fun way of demonstrating the abilities of a GPT.

Fine-tuning

We briefly touched on fine-tuning in the training section. Recall, fine-tuning is when we reuse the pre-trained weights to train the model on some downstream task. We call this process transfer-learning.

In theory, we could use zero-shot or few-shot prompting to get the model to complete our task, however, if you have access to a labelled dataset, fine-tuning a GPT is going to yield better results (results that can scale given additional data and higher quality data).

There are a couple different topics related to fine-tuning, I've broken them down below:

Classification Fine-tuning

In classification fine-tuning, we give the model some text and we ask it to predict which class it belongs to. For example, consider the [IMDB dataset](#), which contains movie reviews that rate the movie as either good, or bad:

```

--- Example 1 ---
Text: I wouldn't rent this one even on dollar rental night.
Label: Bad
--- Example 2 ---
Text: I don't know why I like this movie so well, but I never get tired of watching it
Label: Good
--- Example 3 ---
...

```

To fine-tune our model, we replace the language modeling head with a classification head, which we apply to the last token output:

```
def gpt2(inputs, wte, wpe, blocks, ln_f, cls_head, n_head):
    x = wte[inputs] + wpe[range(len(inputs))]
    for block in blocks:
        x = transformer_block(x, **block, n_head=n_head)
    x = layer_norm(x, **ln_f)

    # project to n_classes
    # [n_embd] @ [n_embd, n_classes] -> [n_classes]
    return x[-1] @ cls_head
```

We only use the last token output `x[-1]` because we only need to produce a single probability distribution for the entire input instead of `n_seq` distributions as in the case of language modeling. We take the last token in particular (instead of say the first token or a combination of all the tokens) because the last token is the only token that is allowed to attend to the entire sequence and thus has information about the input text as a whole.

As per usual, we optimize w.r.t. the cross entropy loss:

```
def single_example_loss_fn(inputs: list[int], label: int, params) -> float:
    logits = gpt(inputs, **params)
    probs = softmax(logits)
    loss = -np.log(probs[label]) # cross entropy loss
    return loss
```

We can also perform **multi-label classification** (i.e. an example can belong to **multiple** classes, not just a single class) by applying `sigmoid` instead of `softmax` and taking the binary cross entropy loss with respect to each class (see [this stack-exchange question](#)).

Generative Fine-tuning

Some tasks can't be neatly categorized into classes. For example, consider the task of summarization. We can fine-tune these types of task by simply performing language modeling on the input concatenated with the label. For example, here's what a single summarization training sample might look like:

```
--- Article ---
This is an article I would like to summarize.
--- Summary ---
This is the summary.
```

We train the model as we do during pre-training (optimize w.r.t language modeling loss).

At predict time, we feed the model the everything up to `--- Summary ---` and then perform auto-regressive language modeling to generate the summary.

The choice of the delimiters `--- Article ---` and `--- Summary ---` are arbitrary. How you choose to format the text is up to you, as long as it is consistent between training and inference.

Notice, we can also formulate classification tasks as generative tasks (for example with IMDB):

```
--- Text ---
I wouldn't rent this one even on dollar rental night.
--- Label ---
Bad
```

However, this will probably perform worse than doing classification fine-tuning directly (loss includes language modeling on the entire sequence, not just the final prediction, so the loss specific to the prediction will get diluted)

Instruction Fine-tuning

Most state-of-the-art large language models these days also undergo an additional **instruction fine-tuning** step after being pre-trained. In this step, the model is fine-tuned (generative) on thousands of instruction prompt + completion pairs that were **human labeled**. Instruction fine-tuning can also be referred to as **supervised fine-tuning**, since the data is human labelled (i.e. **supervised**).

So what's the benefit of instruction fine-tuning? While predicting the next word in a wikipedia article makes the model is good at continuing sentences, it doesn't make it particularly good at following instructions, or having a conversation, or summarizing a document (all the things we would like a GPT to do). Fine-tuning them on human labelled instruction + completion pairs is a way to teach the model how it can be more useful, and make them easier to interact with. This call this **AI alignment**, as we are aligning the model to do and behave as we want it

to. Alignment is an active area of research, and includes more than just following instructions (bias, safety, intent, etc ...).

What does this instruction data look like exactly? Google's [FLAN](#) models were trained on various academic NLP datasets (which are already human labelled):

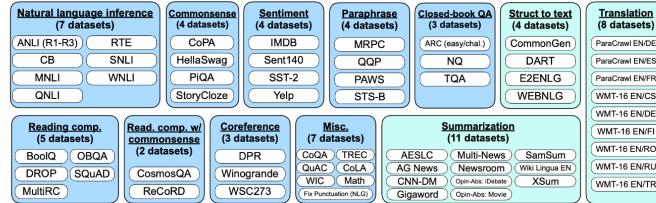


Figure 3: Datasets and task clusters used in this paper (NLU tasks in blue; NLG tasks in teal).

Figure 3 from FLAN paper

OpenAI's [InstructGPT](#) on the other hand was trained on prompts collected from their own API. They then paid workers to write completions for those prompts. Here's a breakdown of the data:

Table 1: Distribution of use case categories from our API prompt dataset.

Use-case	(%)
Generation	45.6%
Open QA	12.4%
Brainstorming	11.2%
Chat	8.4%
Rewrite	6.6%
Summarization	4.2%
Classification	3.5%
Other	3.5%
Closed QA	2.6%
Extract	1.9%

Table 2: Illustrative prompts from our API prompt dataset. These are fictional examples inspired by real usage—see more examples in Appendix A.2.1.

Use-case	Prompt
Brainstorming	List five ideas for how to regain enthusiasm for my career
Generation	Write a short story where a bear goes to the beach, makes friends with a seal, and then returns home.
Rewrite	This is the summary of a Broadway play: "" {summary} "" This is the outline of the commercial for that play: ""

Table 1 and 2 from InstructGPT paper

Parameter Efficient Fine-tuning

When we talk about fine-tuning in the above sections, it is assumed that we are updating all of the model parameters. While this yields the best performance, it is costly both in terms of compute (need to back propagate over the entire model) and in terms of storage (for each fine-tuned model, you need to store a completely new copy of the parameters).

The most simple approach to this problem is to **only update the head** and **freeze** (i.e. make it untrainable) the rest of the model. While this would speed up training and greatly reduce the number of new parameters, it would not perform particularly well since we are losing out on the *depth* in deep learning. We could instead **selectively freeze** specific layers (i.e. freeze all layers except the last 4, or freeze every other layer, or freeze all parameters except multi-head attention parameters), which would help restore the depth. As a result this will perform a lot better, but we become a lot less parameter efficient and we lose out on some of those training speed gains.

Instead, we can utilize **parameter-efficient fine-tuning** methods. This is still an active area of research, and there are [lots of different methods to choose from](#).

As an example, take the [Adapters paper](#). In this approach, we add an additional "adapter" layer after the FFN and MHA layers in the transformer block. The adapter layer is just a simple 2 layer fully connected neural network, where the input and output dimensions are `n_embd`, and the hidden dimension is smaller than `n_embd`:

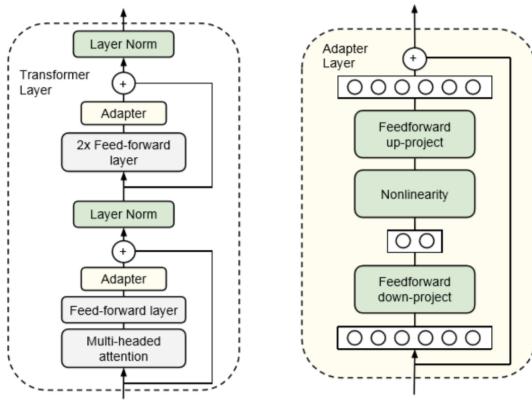


Figure 2 from the Adapters paper

The size of the hidden dimension is a hyper-parameter that we can set, enabling us to tradeoff parameters for performance. For a BERT model, the paper showed that using this approach can reduce the number of trained parameters to 2% while only sustaining a small hit in performance (<1%) when compared to a full fine-tune.

-
1. Training at scale, collecting terabytes of data, making the model fast, evaluating performance, and aligning the models to serve humans is the life's work of the 100s of engineer/researchers required to make LLMs what they are today, not just the architecture. The GPT architecture just happened to be the first neural network architecture that has nice scaling properties, is highly parallelizable on GPUs, and is good at modeling sequences. The real secret sauce comes from scaling the data and model ([as always](#)), GPT just enables us to do that^[9]. It's possible that the transformer has hit the [hardware lottery](#), and some other architecture is still out there waiting to dethrone the transformer. ↵
 2. For certain applications, the tokenizer doesn't require a `decode` method. For example, if you want to classify if a movie review is saying the movie was good or bad, you only need to be able to `encode` the text and do a forward pass of the model, there is no need for `decode`. For generating text however, `decode` is a requirement. ↵
 3. Although, with the [InstructGPT](#) and [Chinchilla](#) papers, we've realized that we don't actually need to train models that big. An optimally trained and instruction fine-tuned GPT at 1.3B parameters can outperform GPT-3 at 175B parameters. ↵
 4. The original transformer paper used a [calculated positional embedding](#) which they found performed just as well as learned positional embeddings, but has the distinct advantage that you can input any arbitrarily long sequence (you are not restricted by a maximum sequence length). However, in practice, your model is only going to be as good as the sequence lengths that it was trained on. You can't just train a GPT on sequences that are 1024 long and then expect it to perform well at 16k tokens long. Recently however, there has been some success with relative positional embeddings, such as [Alibi](#) and [RoPE](#). ↵
 5. Different GPT models may choose a different hidden width that is not `4*n_embd`, however this is the common practice for GPT models. Also, we give the multi-head attention layer a lot of [attention](#) (pun intended) for driving the success of the transformer, but at the scale of GPT-3, [80% of the model parameters are contained in the feed forward layer](#). Just something to think about. ↵
 6. If you're not convinced, stare at the softmax equation and convince yourself this is true (maybe even pull out a pen and paper):

$$\text{softmax}(\vec{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

↵

7. I love JAX ❤️. ↵
 8. Using JAX, this is as simple as `heads = jax.vmap(attention, in_axes=(0, 0, 0, None))(q, k, v, causal_mask)`. ↵
 9. Actually, I might argue that there is something inherently better about the way attention models sequences vs recurrent/convolutional layers, but now we in a footnote inside a footnote, so I digress. ↵
-