GPT in 60 Lines of NumPy

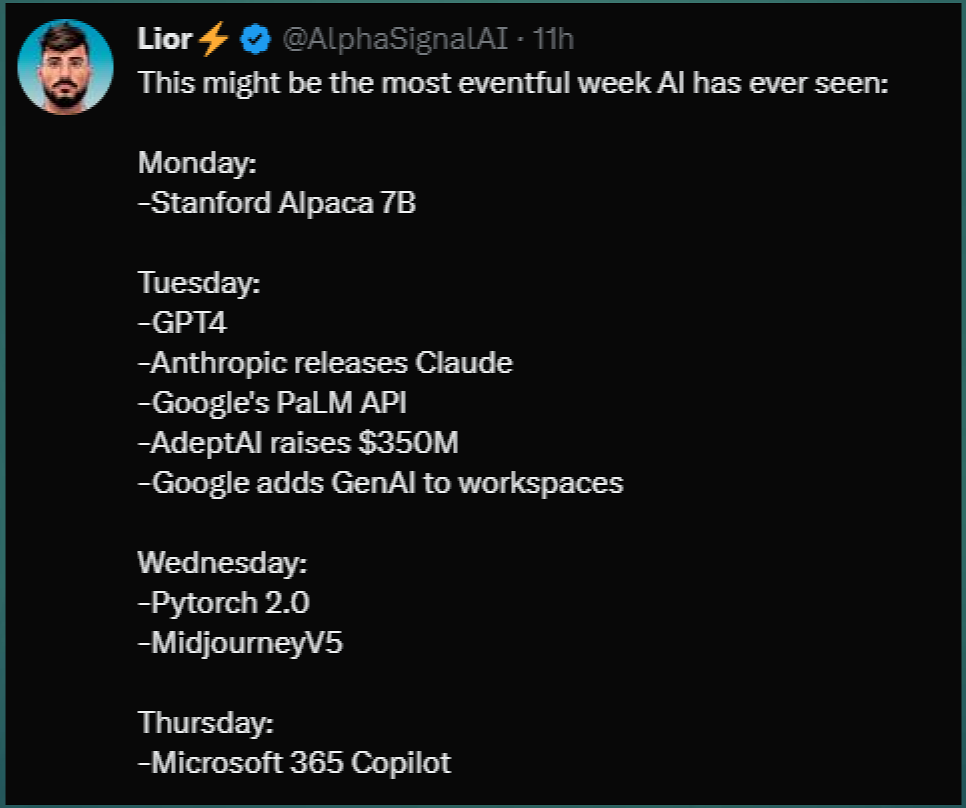
Jay Mody

SUMMARIZED BY: GENE OLAFSEN

Acknowledgement

- https://jaykmody.com/blog/gpt-from-scratch

Things are moving quickly…


Lior⚡ @AlphaSignalAI · 11h
This might be the most eventful week AI has ever seen:

Monday:
-Stanford Alpaca 7B

Tuesday:
-GPT4
-Anthropic releases Claude
-Google's PaLM API
-AdeptAI raises $350M
-Google adds GenAI to workspaces

Wednesday:
-Pytorch 2.0
-MidjourneyV5

Thursday:
-Microsoft 365 Copilot

Imagine...

- Midjourney works similarly to image synthesizers like Stable Diffusion and DALL-E in that it generates images based on text descriptions called "prompts" using an AI model trained on millions of works of human-made art.
- Images such as this are started at prompt labeled "Imagine:"
- ML-generated assets are now being used to construct video-game worlds.

GPT/RNN/LLM

- GPT - Generative Pre-trained Transformer, a type of neural network architecture based on the Transformer.
- Transformers are designed to process sequential input data, such as natural language, similar to recurrent neural networks (RNNs), to support tasks such as translation and text summarization.
- Transformers, unlike RNNs, process the entire input all at once.
- LLM = Large Language Model

What's in a Name?

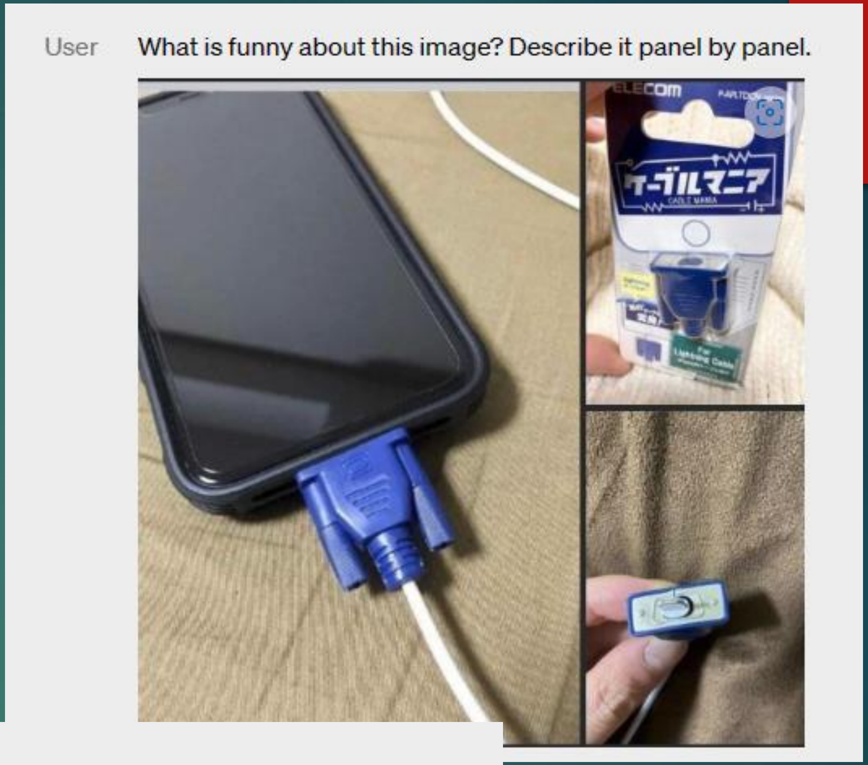
- **G**enerative: A GPT generates text.
- **P**re-trained: A GPT is trained on the text from many books, the Internet, etc ...
- **T**ransformer: A GPT is a decoder-only transformer neural network.
  - Decoder models use only the decoder of a Transformer model. At each stage, for a given word the attention layers can only access the words positioned before it in the sentence.
  - Such models are often called auto-regressive models.
    - (Autoregressive models predict future values based on past values)

GPT High Level

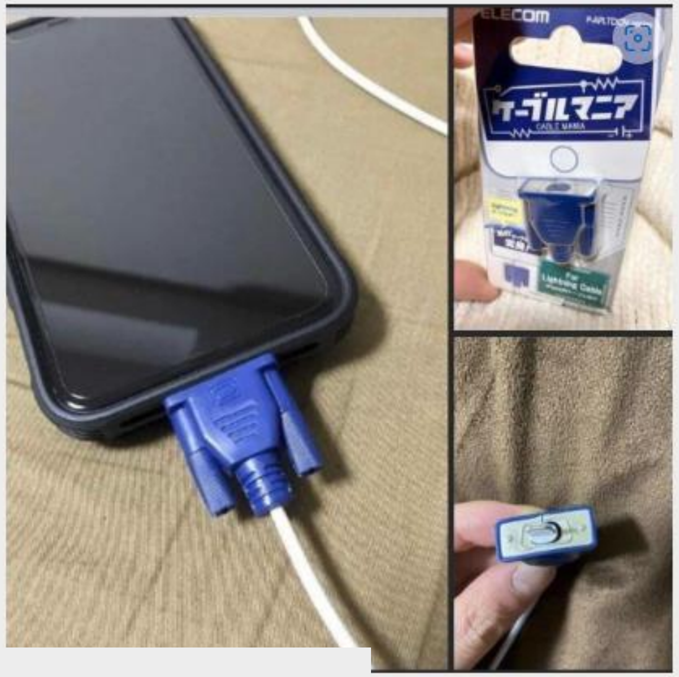
- From an API perspective, GPT is pretty simple. You provide a prompt in the form of text and GPT generates a response and returns text back.

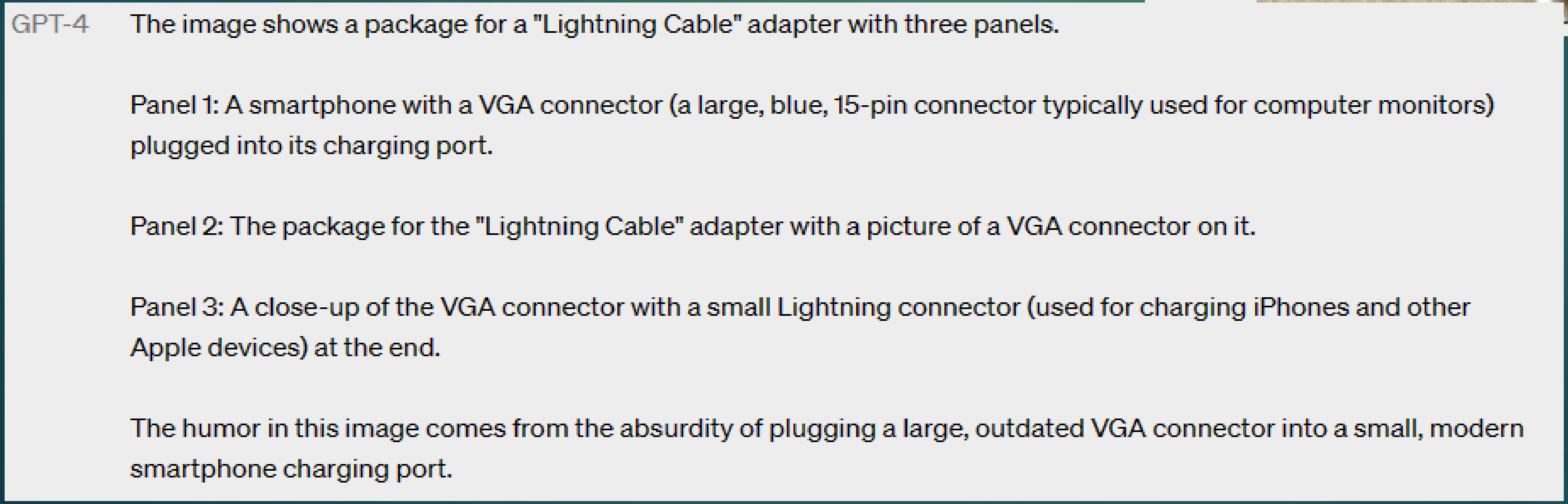
- **Note:** GPT-4 was just released and it accepts text and images as part of the prompt.

GPT-4 Teaser



User What is funny about this image? Describe it panel by panel.





GPT-4 The image shows a package for a "Lightning Cable" adapter with three panels.

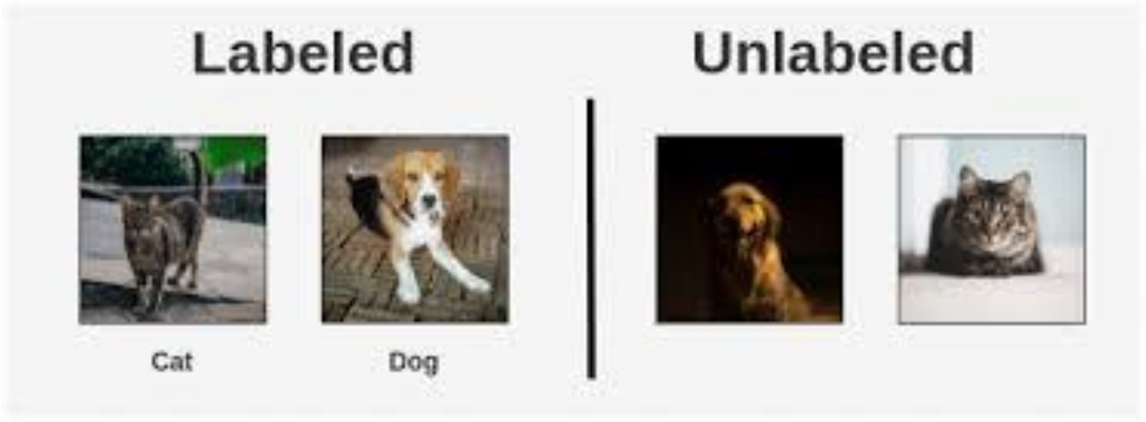
Panel 1: A smartphone with a VGA connector (a large, blue, 15-pin connector typically used for computer monitors) plugged into its charging port.

Panel 2: The package for the "Lightning Cable" adapter with a picture of a VGA connector on it.

Panel 3: A close-up of the VGA connector with a small Lightning connector (used for charging iPhones and other Apple devices) at the end.

The humor in this image comes from the absurdity of plugging a large, outdated VGA connector into a small, modern smartphone charging port.

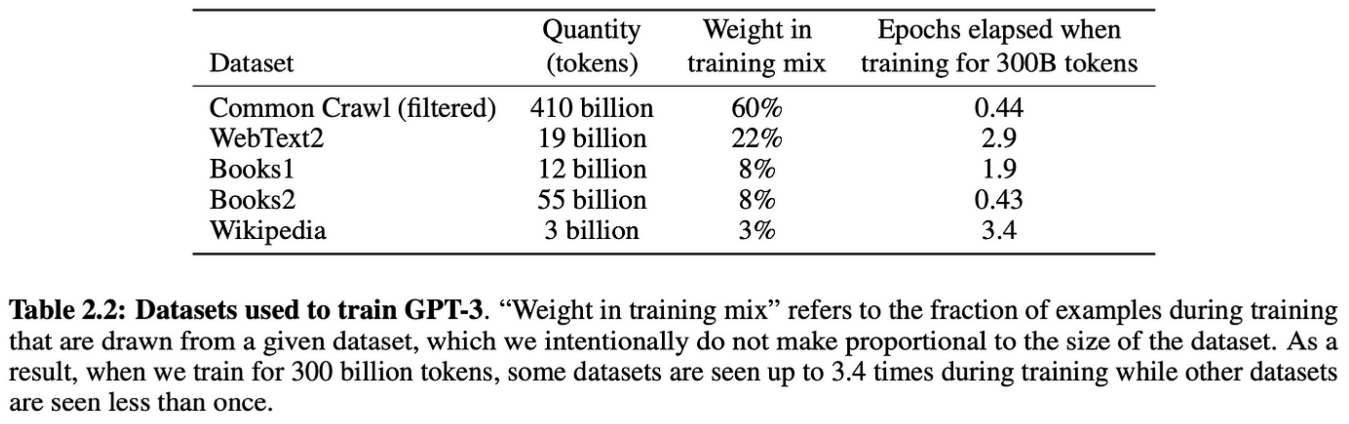
Labeled / Unlabeled



- Labeled data is used in supervised learning.
- Unlabeled data is used in unsupervised/self-supervised learning.
- Labeled data is more difficult to acquire and store (i.e. time consuming and expensive), whereas unlabeled data is easier to acquire and store.

No Labeled Data Required

- Self-supervision enables the massive scaling of training data.
- Just acquire as much raw text as possible and throw it at the model.
- GPT-3 was trained on 300 billion tokens of text from the internet and books:

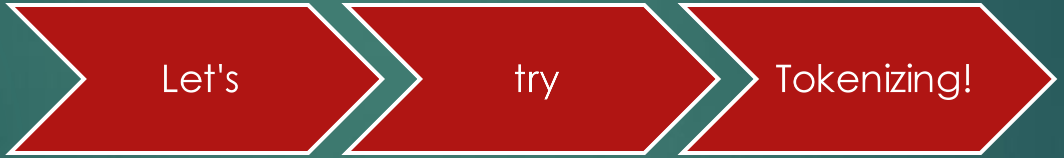


Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

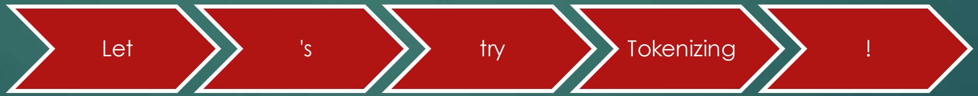
Table 2.2: Datasets used to train GPT-3. "Weight in training mix" refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen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

Tokenizer/Tokenization

- A tokenizer is one of the core components of the NLP pipeline- serving one purpose- to translate text into data that can be processed by the model.
- Models can only process numbers, so a tokenizer need to convert text inputs to numerical values.
- Simple:
  - Split on Spaces



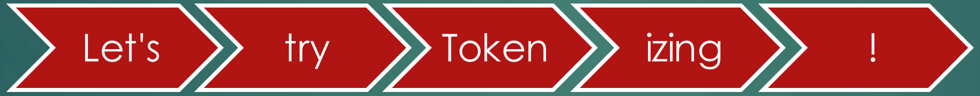
  - Split on Punctuation



- Each word gets assigned a numeric value, starting from 0 and going up to the size of the vocabulary. The model uses these values to identify each word.

"Subword" Tokenization

- Subword tokenization algorithms rely on the principle that frequently used words should not be split into smaller subwords, but rare words should be decomposed into meaningful subwords.
- Subwords provide semantic meaning.



- This approach is especially useful in languages such as Turkish, where (almost) arbitrarily long complex words can be formed by stringing together subwords.
- **Note:** A custom token is often used to represent words that are not in the vocabulary. This is known as the "unknown" token, often represented as "[UNK]"

Other Tokenization Strategies

- There are several other tokenization strategies. Some are well documented, some- such as WordPiece, is used by Google and can only be inferred by published papers and other collateral documentation.
  - Byte-Pair Encoding tokenization (BPE)
  - WordPiece
  - Unigram
- Typically, these tokenizers strive to produce tokens/token counts with just the 'right' amount of granularity to provide corpus coverage without having neither too many tokens or too few tokens (and may unknowns [UNK]s.

Not all heroes wear capes.

- The input to a GPT method is a sequence of integers which correspond to the tokenized 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]
```

Predicting capes.

- The sample method predicts the probability for the 'next' token for the entire vocabulary.
- Selecting the next token based on the highest probability is referred to as 'greedy sampling' or 'greedy decoding'.

```
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" with the highest probability
```

Full sentence prediction

- The generation of full sentences is achieved by iteratively getting the next token prediction from the model.
- After each iteration, the predicted token is appended to the input.
- In the following code fragment, a prompt to two tokens is provided ('not' 'all') and the next three tokens are requested in the response.

```
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"
```

Autoregressive

- This process demonstrated in code on the last slide of predicting a future value (regression), and adding it back into the input (auto), is why GPT is sometimes described as 'autoregressive'.

Take a Walk on the 'Random' Side

Apologies to Lou Reed

- Noun. 1. **stochasticity** - the quality of lacking any predictable order or plan.

```
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
```

Different Sentences, Same Input

- Top-K
  - Sample from a shortlist of the top three tokens.
- Top-P (nucleus sampling)
  - Sample from amongst from the smallest possible set of words whose cumulative probability exceeds the probability p.
- Temperature
  - Temperature is a number used to tune the degree of randomness. A lower temperature means less randomness; a temperature of 0 will always yield the same output. High temperature means more randomness, which can help the model give more creative outputs.

Training

- A GPT AI model like any other neural network- using gradient descent with respect to some loss function.
- A *cross entropy* loss function is used for a GPT.

```
def lm_loss(inputs: list[int], params) -> float:
    # the labels y are just the input shifted 1 to the left
    #
    # inputs = [not,     all,    heroes,    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
```

Training Code Explanation

- A function named 'lm_loss' computes the language modeling loss for the given input text.
- The loss determines the gradients, which are computed by a method named 'compute_gradients_via_backpropagation'.
- The gradients are used to update the model parameters such that the loss is minimized (gradient descent)

Loss Function

- The *loss function* is a method of evaluating how well your algorithm is modeling your dataset.
- This function computes the 'distance' between the current output of the algorithm and the expected output.
- **Note:** If the loss function doesn't penalize wrong output-- appropriate to its magnitude-- it can delay model convergence and affect learning.

Cross Entropy

- Cross entropy loss is a metric used to measure how well a classification model in machine learning performs. Where loss (or error) is measured between 0 and 1-- and a value of 0 represents a perfect model. The goal during training is to get a model as close to 0 as possible.
- Cross entropy measures the difference between two probability distributions for a given random variable/set of events.
- *All possible values for the prediction are stored so, for example, if you were looking for the odds in a coin toss it would store that information at 0.5 and 0.5 (heads and tails)

*https://wandb.ai/sauravmaheshkar/cross-entropy/reports/What-Is-Cross-Entropy-Loss-A-Tutorial-With-Code--VmlldzoxMDA5NTMx

Training GPT-3

- GPT-3 has 175 billion parameters.
- GPT-3 probably cost between 4 and 10 million dollars to train.

Foundation Models, Fine-tuning and Transfer Learning

- 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*.

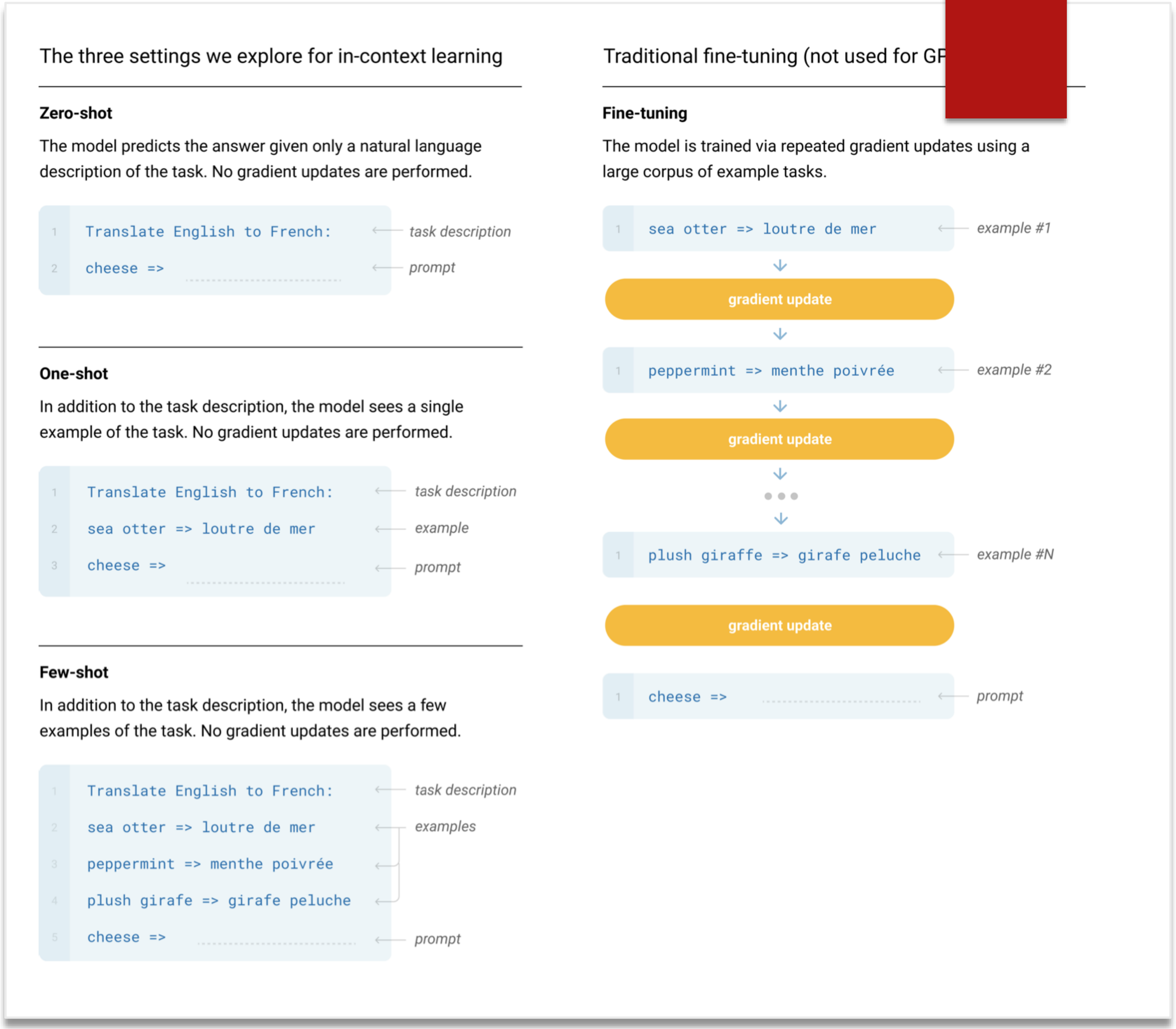
Intent of Original GPT

- The original GPT paper contained information 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.

Prompting

- It wasn't until the GPT-2 and GPT-3 papers that there was a realization that a GPT model pre-trained on enough data, configured with enough parameters-- was capable of performing any arbitrary task "by itself", without the need for fine-tuning.
- 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.

GPT-3 Prompt Examples



The three settings we explore for in-context learning

Zero-shot

The model predicts the answer given only a natural language description of the task. No gradient updates are performed.

```
Translate English to French:      ← task description
cheese => ..................      ← prompt
```

One-shot

In addition to the task description, the model sees a single example of the task. No gradient updates are performed.

```
Translate English to French:      ← task description
sea otter => loutre de mer        ← example
cheese => ..................      ← prompt
```

Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
Translate English to French:      ← task description
sea otter => loutre de mer        ← examples
peppermint => menthe poivrée      ←
plush girafe => girafe peluche    ←
cheese => ..................      ← prompt
```

Traditional fine-tuning (not used for GP

Fine-tuning

The model is trained via repeated gradient updates using a large corpus of example tasks.

```
sea otter => loutre de mer        ← example #1
        ↓
  gradient update
        ↓
peppermint => menthe poivrée      ← example #2
        ↓
  gradient update
        ↓
       ...
        ↓
plush giraffe => girafe peluche   ← example #N

  gradient update

cheese => ..................      ← prompt
```

Configuring an Environment

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

## no pip installed- next two commands fixed that
sudo apt-get update
sudo apt-get install -y python3-pip

pip install -r requirements.txt

## This next command will download the necessary model and tokenizer files to models/124M
and load encoder, hparams, and params into our code.

python3
>>>from utils import load_encoder_hparams_and_params
>>>encoder, hparams, params = load_encoder_hparams_and_params("124M", "models")
```

File Breakdown

- encoder.py
  - Contains the code for OpenAI's BPE Tokenizer, taken straight from their gpt-2 repo.
- utils.py
  - The code to download and load the GPT-2 model weights, tokenizer, and hyperparameters.
- gpt2.py
  - The actual GPT model and generation code, which can be run as a python script. **-- this is the code that will be reimplemented from scratch**
- gpt2_pico.py
  - The same as gpt2.py, but in even fewer lines of code. Why? Because why not.

Code: Main

```python
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)
```

Main Overview

- The main function handles:
  - Loading the tokenizer (encoder), model weights (params), and hyperparameters (hparams)
  - Encoding the input prompt into token IDs using the tokenizer
  - Calling the generate function
  - Decoding the output IDs into a string
- fire.Fire(main) just turns thefile into a CLI application, so it can eventually run with:
  - python gpt2.py "some prompt here"

Code: GPT2, Generate

```python
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 loop
        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
```

GPT2, Generate Overview

- The gpt2 function is the actual GPT code being implemented. The function signature includes some other arguments in addition to inputs:
  - wte, wpe, blocks, and ln_f the parameters of the model.
  - n_head is a hyperparameter that is needed during the forward pass.
- The generate function is the autoregressive decoding algorithm we saw earlier.
  - Greedy sampling for simplicity.
  - tqdm is a progress bar to help us visualize the decoding process as it generates tokens one at a time.

Exercising the Encoder

- Python from terminal
- 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."
```

```
Note to self:

>>> ids = encoder.encode("Not all heroes wear capes.")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'encoder' is not defined

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

Looking at the tokens...

- Using the vocabulary of the tokenizer (stored in encoder.decoder), it is possible to take a peek at what the actual tokens look like:

```
>>> [encoder.decoder[i] for i in ids]
['Not', 'Ġall', 'Ġheroes', 'Ġwear', 'Ġcap', 'es', '.']
```

- Looking at tokens:
  - Sometimes tokens are words (e.g. Not)
  - Other times they are words but with a space in front of them
    - (e.g. Ġall, the Ġ represents a space),
  - They may be part of a word (e.g. capes is split into Ġcap and es), and
  - Finally they are punctuation (e.g. .).

Looking at the tokens...

- BPE can encode an arbitrary string.
- If BPE 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("zjqfl")]
 ['z', 'j', 'q', 'fl']

We can also check the size of the vocabulary:

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

Review

- The vocabulary, as well as the byte-pair merges which determines how strings are broken down, is obtained by training the tokenizer.
- When the tokenizer is loaded, so also are the already trained vocabulary and byte-pair merges from files.
  - These files were downloaded alongside the model files when executing load_encoder_hparams_and_params.

```
## This next command will download the necessary model and tokenizer files to models/124M
and load encoder, hparams, and params into our code.
python3
>>>from utils import load_encoder_hparams_and_params
>>>encoder, hparams, params = load_encoder_hparams_and_params("124M", "models")
```

- See:
  - `/models/124M/encoder.json` (the vocabulary)
  - `/models/124M/vocab.bpe` (byte-pair merges).

Hyperparameters

- hparams is a dictionary that contains the model's hyperparameters

```
>>> 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)
}
```

- Additionally, the parameter n_seq will be used to denote the length of the input sequence (i.e. n_seq = len(inputs)).

Parameters

- params is a nested json dictionary that holds the trained weights of the 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
    ]
}
```

Note: We will come back to reference this dictionary to check the shape of the weights as we implement our GPT. We'll match the v ariable names in our code with the keys of this dictionary for consistency.

```
>>> from notes import shape_tree
>>> print(shape_tree(params))
```

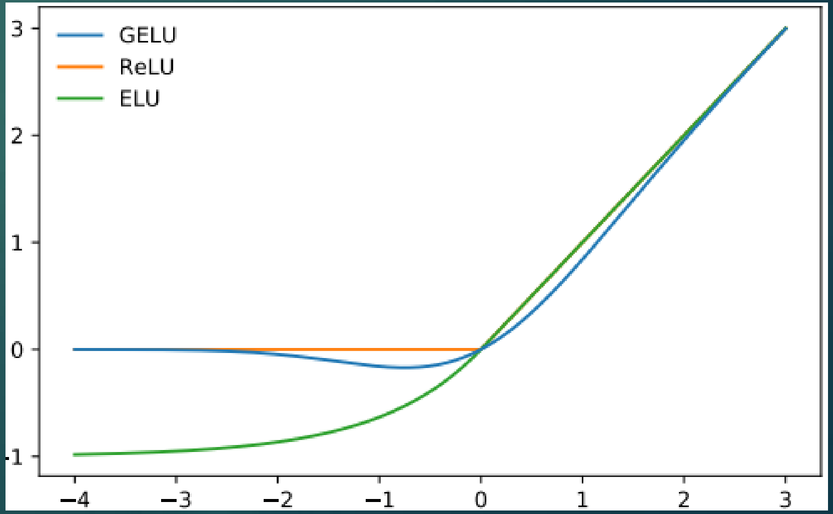
GELU

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

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

- Softmax is used to a convert set of real numbers (between neg. infinity and infinity) to probabilities (between 0 and 1, with the numbers all summing to 1).

```
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)
```

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

- Softmax is applied 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.])
```

AGENDA

Introduction

Primary goals

Areas of growth

Timeline

Summary

Introduction

- At Contoso, we empower organizations to foster collaborative thinking to further drive workplace innovation. By closing the loop and leveraging agile frameworks, we help business grow organically and foster a consumer-first mindset.

PRIMARY GOALS

ANNUAL REVENUE GROWTH

AREAS OF GROWTH

	B2B	Supply chain	ROI	E-commerce
Q1	4.5	2.3	1.7	5.0
Q2	3.2	5.1	4.4	3.0
Q3	2.1	1.7	2.5	2.8
Q4	4.5	2.2	1.7	7.0

“BUSINESS OPPORTUNITIES ARE LIKE BUSES. THERE'S ALWAYS ANOTHER ONE COMING.”

Richard Branson

MEET OUR TEAM

TAKUMA HAYASHI

President

MIRJAM NILSSON

Chief Executiv e Officer

FLORA BERGGREN

Chief Operations Officer

RAJESH SANTOSHI

V P Marketing

MEET OUR Extended TEAM

TAKUMA HAYASHI
President

MIRJAM NILSSON
Chief Executive Officer

FLORA BERGGREN
Chief Operations Officer

RAJESH SANTOSHI
VP Marketing

GRAHAM BARNES
VP Product

ROWAN MURPHY
SEO Strategist

ELIZABETH MOORE
Product Designer

ROBIN KLINE
Content Developer

PLAN FOR PRODUCT LAUNCH

PLANNING	MARKETING	DESIGN	STRATEGY	LAUNCH
Synergize scalable e-commerce	Disseminate standardized metrics	Coordinate e-business applications	Foster holistically superior methodologies	Deploy strategic networks with compelling e-business needs

TIMELINE

SEP 20XX	NOV 20XX	JAN 20XX	MAR 20XX	MAY 20XX
Synergize scalable e-commerce	Disseminate standardized metrics	Coordinate e-business applications	Foster holistically superior methodologies	Deploy strategic networks with compelling e-business needs

AREAS OF FOCUS

B2B MARKET SCENARIOS

- Develop winning strategies to keep ahead of the competition
- Capitalize on low-hanging fruit to identify a ballpark value
- Visualize customer directed convergence

CLOUD-BASED OPPORTUNITIES

- Iterative approaches to corporate strategy
- Establish a management framework from the inside

HOW WE GET THERE

ROI

- Envision multimedia-based expertise and cross-media growth strategies
- Visualize quality intellectual capital
- Engage worldwide methodologies with web-enabled technologies

NICHE MARKETS

- Pursue scalable customer service through sustainable strategies
- Engage top-line web services with cutting-edge deliverables

SUPPLY CHAINS

- Cultivate one-to-one customer service with robust ideas
- Maximize timely deliverables for real-time schemas

SUMMARY

- At Contoso, we believe in giving 110%. By using our next-generation data architecture, we help organizations virtually manage agile workflows. We thrive because of our market knowledge and great team behind our product. As our CEO says, "Efficiencies will come from proactively transforming how we do business."

THANK YOU

MIRJAM NILSSON

MIRJAM@CONTOSO.COM

WWW.CONTOSO.COM