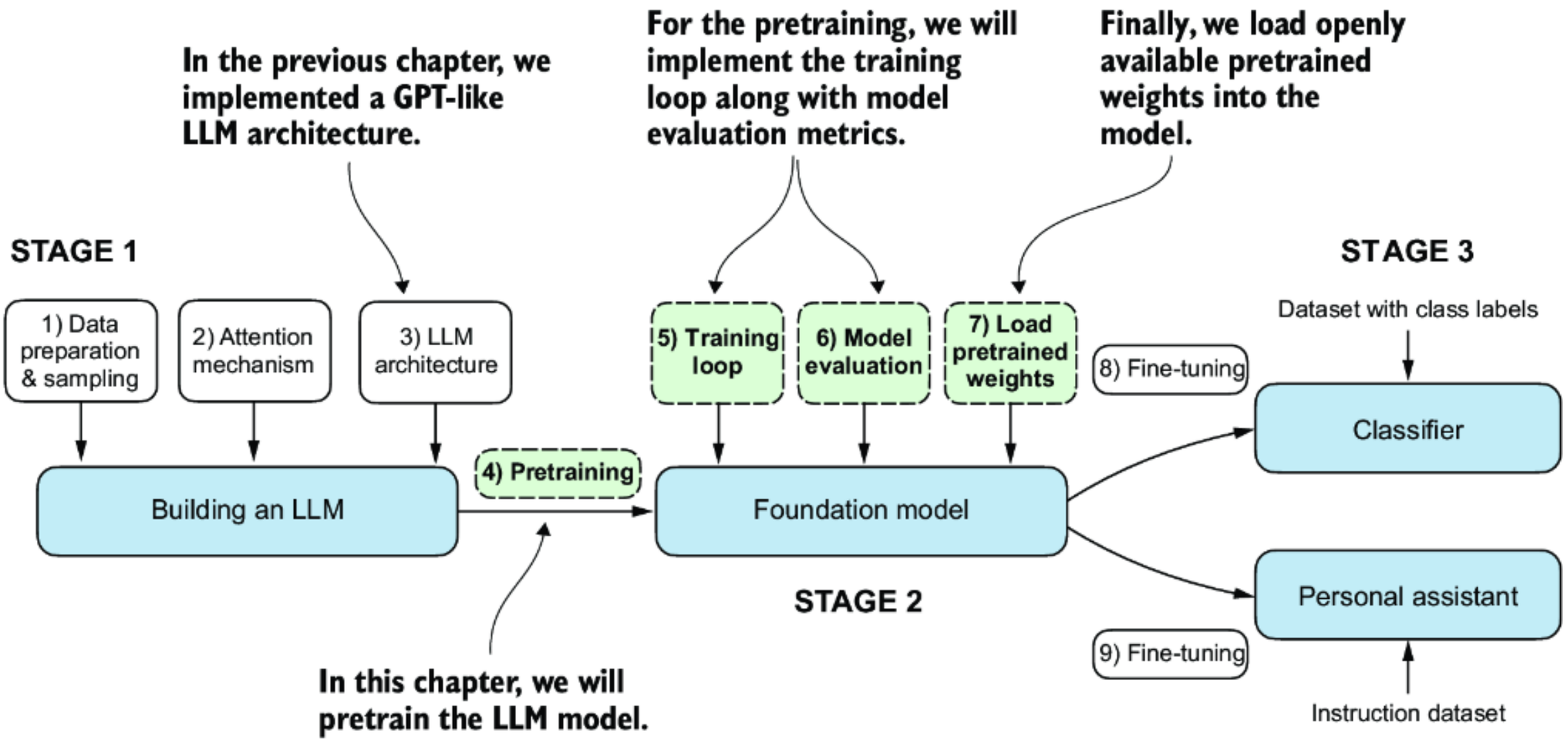
CS 696 Applied Large Language Models
Spring Semester, 2025
Doc 12 Training
Feb 18, 2025

Chapter 5 - Pretraining



Preview

Chapter 5

- Pretraining
- Loading model weights from Open AI

Scripts to do both

gpt_generate.py

Load and use the pretrained model weights from OpenAI

CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves you"

(base) rwhitney@127 01_main-chapter-code % python gpt_generate.py
File already exists and is up-to-date: gpt2/124M/checkpoint
File already exists and is up-to-date: gpt2/124M/encoder.json
File already exists and is up-to-date: gpt2/124M/hparams.json
File already exists and is up-to-date: gpt2/124M/model.ckpt.data-00000-of-00001
File already exists and is up-to-date: gpt2/124M/model.ckpt.index
File already exists and is up-to-date: gpt2/124M/model.ckpt.meta
File already exists and is up-to-date: gpt2/124M/vocab.bpe
Output text:
Every effort moves you toward finding an ideal life. You don't have to accept your problems by trying to remedy them, because that would be foolish

Second run

Note they saved model etc.

gpt_generate.py

Load and use the pretrained model weights from OpenAI

CHOOSE_MODEL = "gpt2-medium (355M)"
INPUT_PROMPT = "Every effort moves you”
It's the same seed as before

(base) rwhitney@127 01_main-chapter-code % python gpt_generate.py
checkpoint: 100%|| 77.0/77.0 [00:00<00:00, 21.3kiB/s]
encoder.json: 100%|| 1.04M/1.04M [00:00<00:00, 1.76MiB/s]
hparams.json: 100%|| 91.0/91.0 [00:00<00:00, 9.01kiB/s]
model.ckpt.data-00000-of-00001: 100%|| 1.42G/1.42G [05:31<00:00, 4.28MiB/s]
model.ckpt.index: 100%|| 10.4k/10.4k [00:00<00:00, 831kiB/s]
model.ckpt.meta: 100%|| 927k/927k [00:00<00:00, 1.76MiB/s]
vocab.bpe: 100%|| 456k/456k [00:00<00:00, 997kiB/s]
Output text:

Every effort moves you toward balance." But it seems that these values have been forgotten by both parties.

If Congress is to fulfill these basic

gpt_train.py

Trains the model on the-verdict.txt

Ep 1 (Step 000000): Train loss 9.812, Val loss 9.846

Ep 1 (Step 000005): Train loss 7.588, Val loss 8.044

Every effort moves you,,,,,,,,,,,,.

Ep 2 (Step 000010): Train loss 6.582, Val loss 6.800

Ep 2 (Step 000015): Train loss 5.920, Val loss 6.592

Every effort moves you, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and

...

Ep 10 (Step 000085): Train loss 0.358, Val loss 6.542

Every effort moves you?" "Yes--quite insensible to the irony. She wanted him vindicated--and by me
He laughed again, and threw back his head to look up at the sketch of the donkey. "There were days
when I

Traceback (most recent call last):

  File "/Users/rwhitney/Courses/696/Spring 2025 LLM/Notebooks/LLMs-from-scratch/ch05/01_main-
chapter-code/gpt_train.py", line 236, in <module>

    plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)

  File "/Users/rwhitney/Courses/696/Spring 2025 LLM/Notebooks/LLMs-from-scratch/ch05/01_main-
chapter-code/gpt_train.py", line 116, in plot_losses

Model to Train

```
import torch
from previous_chapters import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,   # Vocabulary size
    "context_length": 256, # Shortened context length (orig: 1024)
    "emb_dim": 768,        # Embedding dimension
    "n_heads": 12,         # Number of attention heads
    "n_layers": 12,        # Number of layers
    "drop_rate": 0.1,      # Dropout rate
    "qkv_bias": False      # Query-key-value bias
}

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval();  # Disable dropout during inference
```

Training

Convert text to tokens

Need inputs and targets

Determine how “off” model(inputs) are from targets

Use loss function to adjust the weights

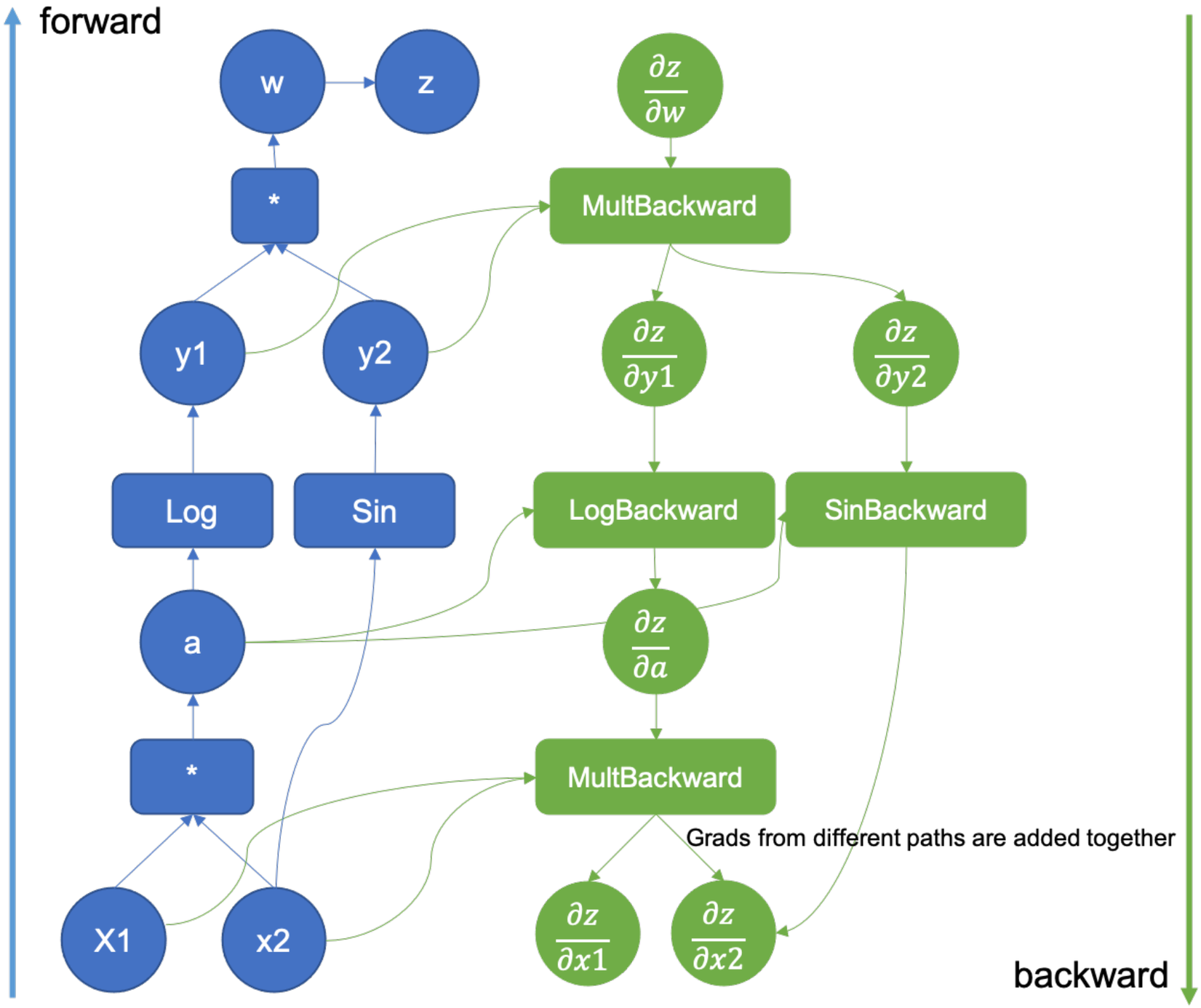
Inputs and Targets

```
inputs = torch.tensor([[16833, 3626, 6100],   # ["every effort moves",
                       [40,    1107, 588]])   #  "I really like"]

targets = torch.tensor([[3626, 6100, 345  ],  # [" effort moves you",
                        [1107,  588, 11311]]) #  " really like chocolate"]
```

```
with torch.no_grad():
    logits = model(inputs)

probas = torch.softmax(logits, dim=-1) # Probability of each token in vocabulary
print(probas.shape) # Shape: (batch_size, num_tokens, vocab_size)
```

torch.Size([2, 3, **50257**])

log(x1*x2) * sin(x2)



PyTorch Autograd Saved Tensors

For some computations, Torch will store

- Input tensors
- Intermediate tensors

Done to make backpropagation more efficient

```
import torch
x = torch.randn(5, requires_grad=True)
y = x.exp()
print(y.equal(y.grad_fn._saved_result))  # True
print(y is y.grad_fn._saved_result)
```

```
True
False
```

requires_grad flag

If true tensors will have gradients accumulated in their .grad field

Defaults to false unless wrapped in an nn.Parameter

requires_grad=False

- Means they will not be part of the backward graph
- So will not be updated backward calculation

```
with torch.no_grad():
    logits = model(inputs)
```

softmax

Converts collection of values to probabilities

Exponential of each value

Normalize result

Logits	Exponential	Normalized
2.5	12.18	0.7856
1.0	2.72	0.1753
-0.5	0.61	0.0391

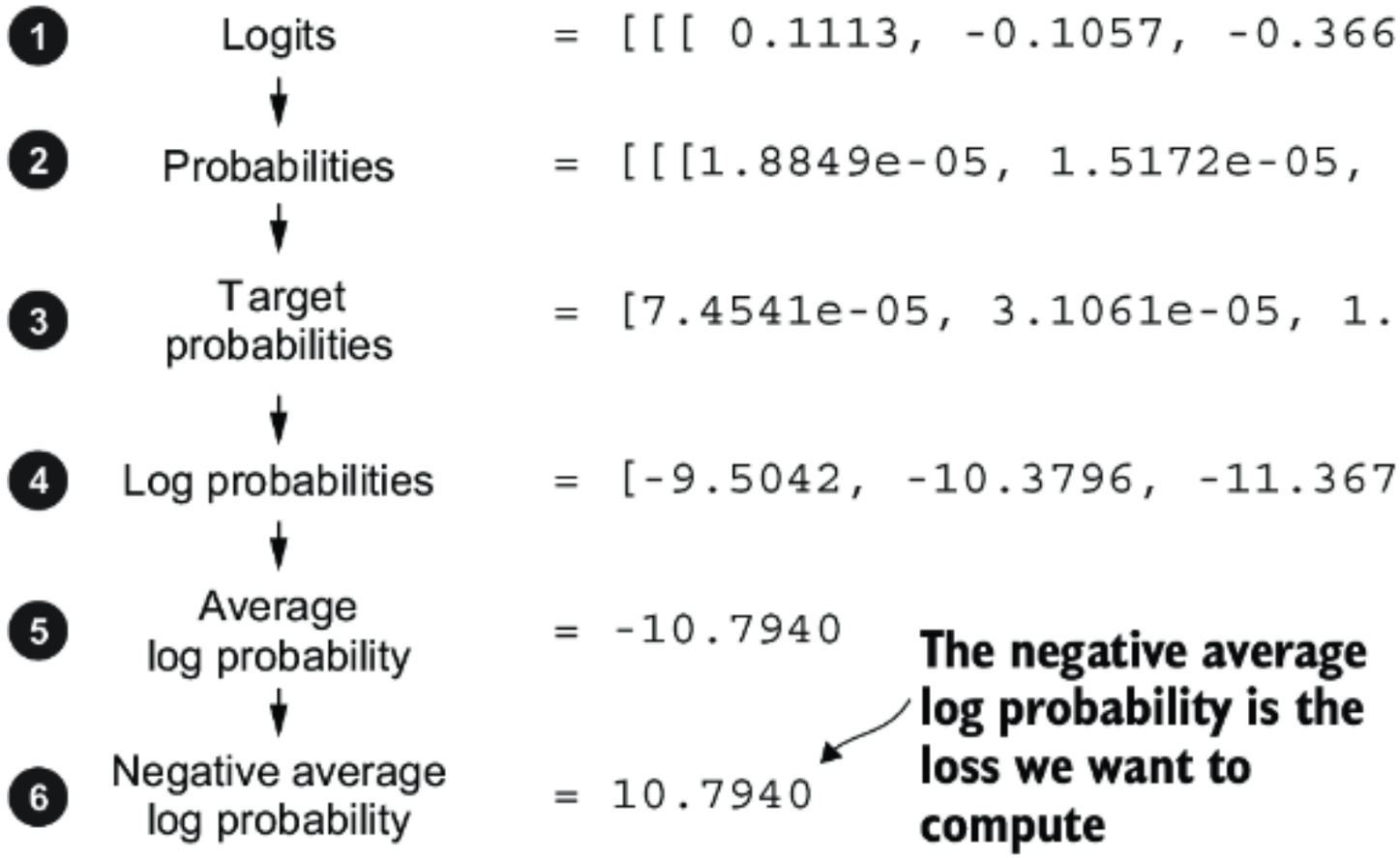
```
import torch

x = torch.tensor([2.5, 1.0, -0.5])
y = torch.softmax(x, dim=0)

print(y)

  tensor([0.7856, 0.1753, 0.0391])
```

Logits	Softmax
5.5	0.9866
1.0	0.0110
-0.5	0.0024

Logits	Softmax
10.5	0.9991
1.0	7.4845E-05
-0.5	1.6700E-05

Extreme values push softmax results to 1 & 0

Backpropagation - cross_entropy



loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)

Perplexity

How well probability distribution given by the model
matches the actual distribution of the words in the dataset

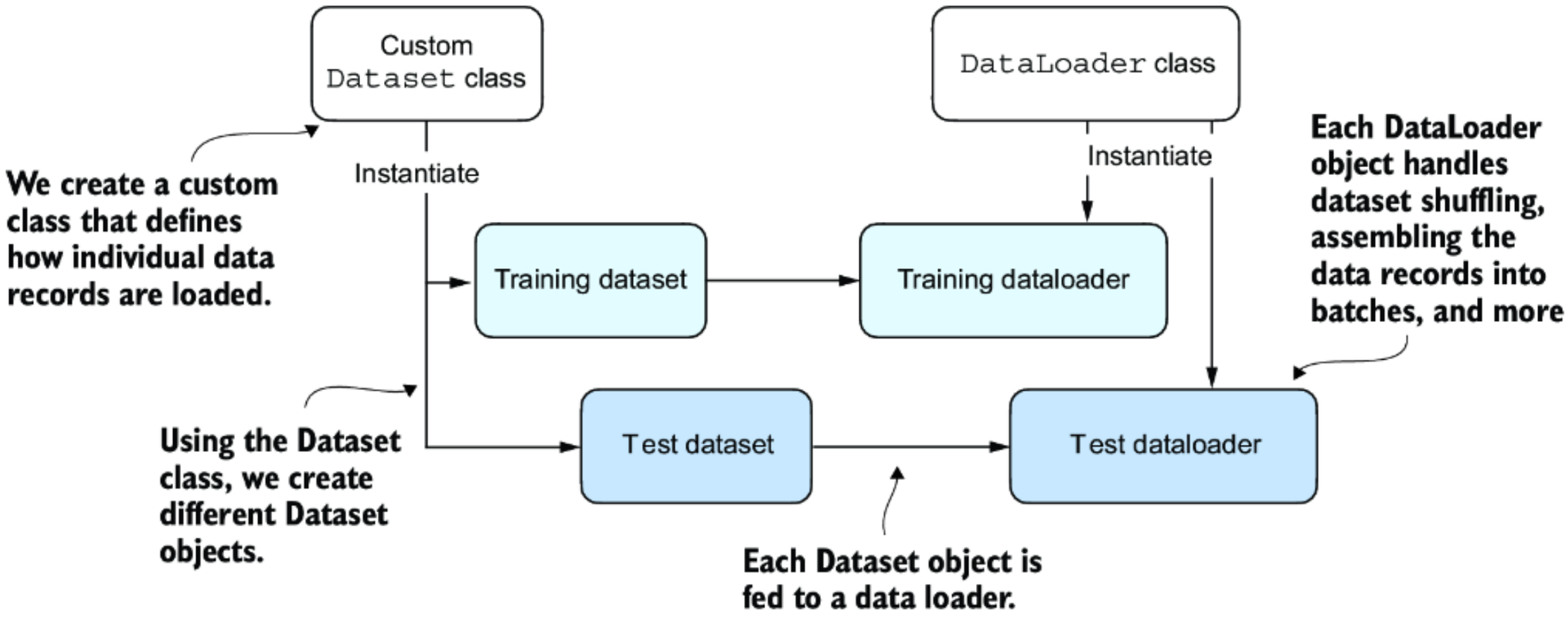
How uncertain a model is about the next word in a sequence.

Effective vocabulary size that the model is uncertain about at each step

torch.exp(loss)

tensor(48725.8203)

Reading Data



Dataset Types

Map-style datasets

- Subclass of torch.utils.data.Datase
- __getitem__()
- __len__()

- Map from (possibly non-integral) indices/keys to data samples

Iterable-style datasets

- subclass of torch.utils.data.IterableDataset
- __iter__()

- Useful when data comes from a stream

Simple Dataset

```
from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):
        one_x = self.features[index]
        one_y = self.labels[index]
        return one_x, one_y

    def __len__(self):
        return self.labels.shape[0]

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
```

From Chapter 2

```
import torch
from torch.utils.data import Dataset, DataLoader
class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.input_ids = []
        self.target_ids = []

        token_ids = tokenizer.encode(txt)

    for i in range(0, len(token_ids) - max_length, stride):
        input_chunk = token_ids[i:i + max_length]
        target_chunk = token_ids[i + 1: i + max_length + 1]
        self.input_ids.append(torch.tensor(input_chunk))
        self.target_ids.append(torch.tensor(target_chunk))

def __len__(self):
    return len(self.input_ids)

def __getitem__(self, idx):
    return self.input_ids[idx], self.target_ids[idx]
```

DataLoader

Batching, shuffling, and parallelizing data loading

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

DataLoader Arguments

dataset:

The Dataset object from which the data is loaded

batch_size:

The number of samples in each batch.

shuffle:

A boolean indicating whether to shuffle the data.

num_workers:

Number of workers to process data in parallel

collate_fn:

The default collate function works for most common use cases.

pin_memory:

Copy Tensors into CUDA pinned memory before returning them.

This can improve data transfer speeds to GPU devices.

drop_last:

Drop the last incomplete batch, if the dataset size is not evenly divisible by the batch size.

Creating the Loader

```
def create_dataloader_v1(txt, batch_size=4, max_length=256,
                         stride=128, shuffle=True, drop_last=True, num_workers=0):
    tokenizer = tiktoken.get_encoding("gpt2")
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)
    dataloader = DataLoader(
        dataset, batch_size=batch_size, shuffle=shuffle, drop_last=drop_last,
        num_workers=num_workers)

return dataloader
```

Reading Data

```
file_path = "the-verdict.txt"
with open(file_path, "r", encoding="utf-8") as file:
    text_data = file.read()


train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```

```
from chapter02 import create_dataloader_v1
torch.manual_seed(123)                                  #To make things consistent

train_loader = create_dataloader_v1(
    train_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    val_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)
```

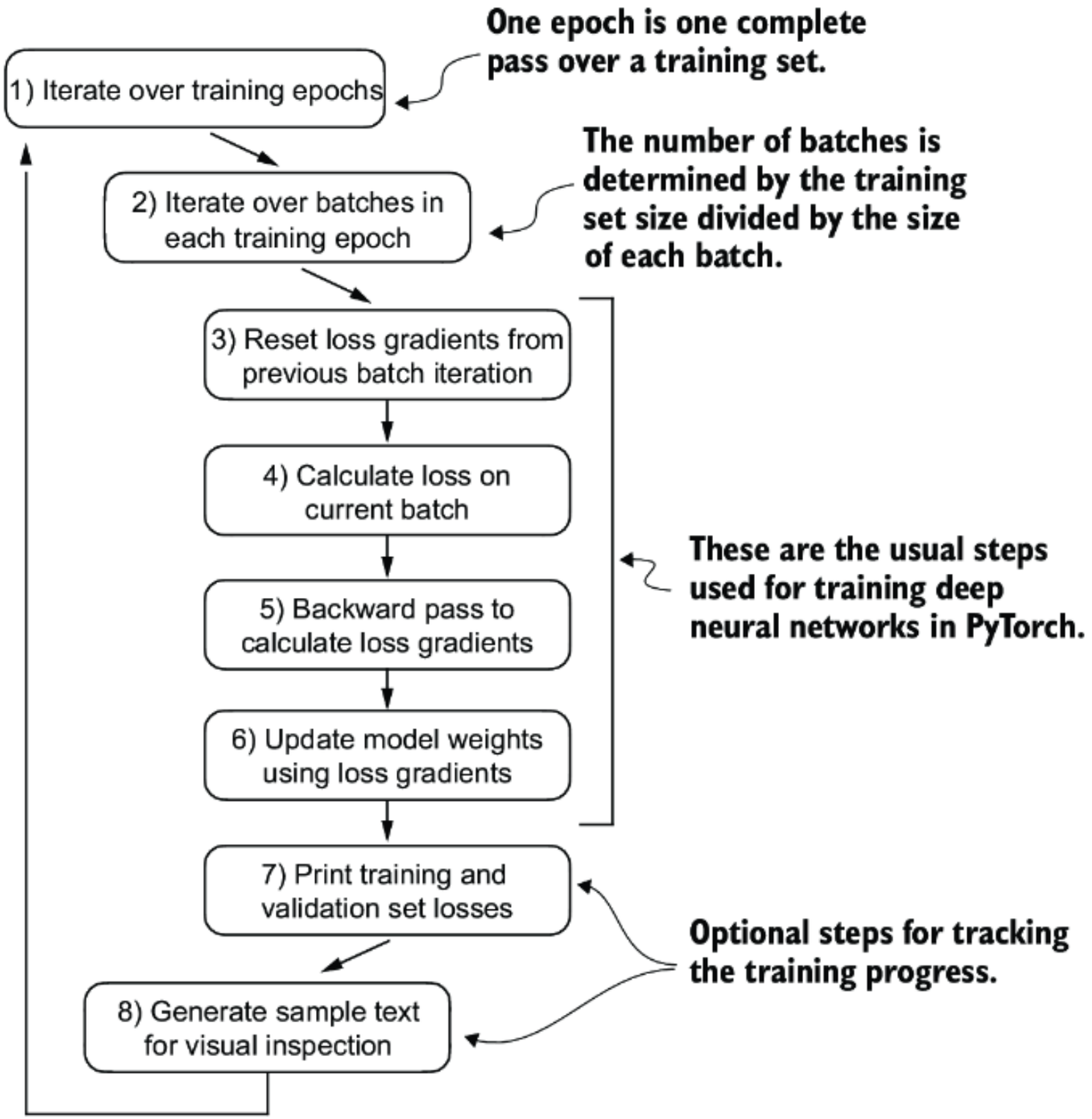
Loss for one Batch

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)        #1
    target_batch = target_batch.to(device)
    logits = model(input_batch)
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1), target_batch.flatten()
    )
    return loss
```

Loss for All Batched

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
for i, (input_batch, target_batch) in enumerate(data_loader):
    if i < num_batches:
        loss = calc_loss_batch(
            input_batch, target_batch, model, device
        )
        total_loss += loss.item()
    else:
        break
return total_loss / num_batches
```

Training an LLM



```
def train_model_simple(model, train_loader, val_loader,
                       optimizer, device, num_epochs,
                       eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward()
            optimizer.step()
            tokens_seen += input_batch.numel()
            global_step += 1

            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}"
                )

        generate_and_print_sample(
            model, tokenizer, device, start_context
        )
    return train_losses, val_losses, track_tokens_seen
```

evaluate_model

```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_loss = calc_loss_loader(
            val_loader, model, device, num_batches=eval_iter
        )
    model.train()
    return train_loss, val_loss
```

```
def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(
        model=model, idx=encoded,
        max_new_tokens=50, context_size=context_size
    )
decoded_text = token_ids_to_text(token_ids, tokenizer)
print(decoded_text.replace("\n", " "))      #1
model.train()
```

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=0.0004, weight_decay=0.1
)
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenizer
)
```

torch.optim.Adam

Adaptive Moment Estimation

Momentum

Accelerate gradient descent
by adding a fraction of the previous update to the current update.

RMSProp (Root Mean Square Propagation)

Adapts the learning rate for each parameter based on the magnitude of recent gradients

Adaptive Learning Rates

Bias Correction

torch.optim.Adam

Adaptive Moment Estimation

Computational Efficiency

- Computationally efficient
- Low memory requirements,
- Suitable for training large neural networks

Robust

- Performs well across a wide range of deep learning tasks and model architectures

Fast Convergence

- Often converges faster than traditional optimization algorithms

torch.optim.AdamW

Effective in training large and complex models

Decouples of weight decay from the gradient-based updates

Weight decay is a separate step
Applying it directly to the weights after the gradient update

gpt_train.py

Trains the model on the-verdict.txt

Ep 1 (Step 000000): Train loss 9.812, Val loss 9.846

Ep 1 (Step 000005): Train loss 7.588, Val loss 8.044

Every effort moves you,,,,,,,,,,,,.

Ep 2 (Step 000010): Train loss 6.582, Val loss 6.800

Ep 2 (Step 000015): Train loss 5.920, Val loss 6.592

Every effort moves you, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and

…

Ep 9 (Step 000075): Train loss 0.762, Val loss 6.329

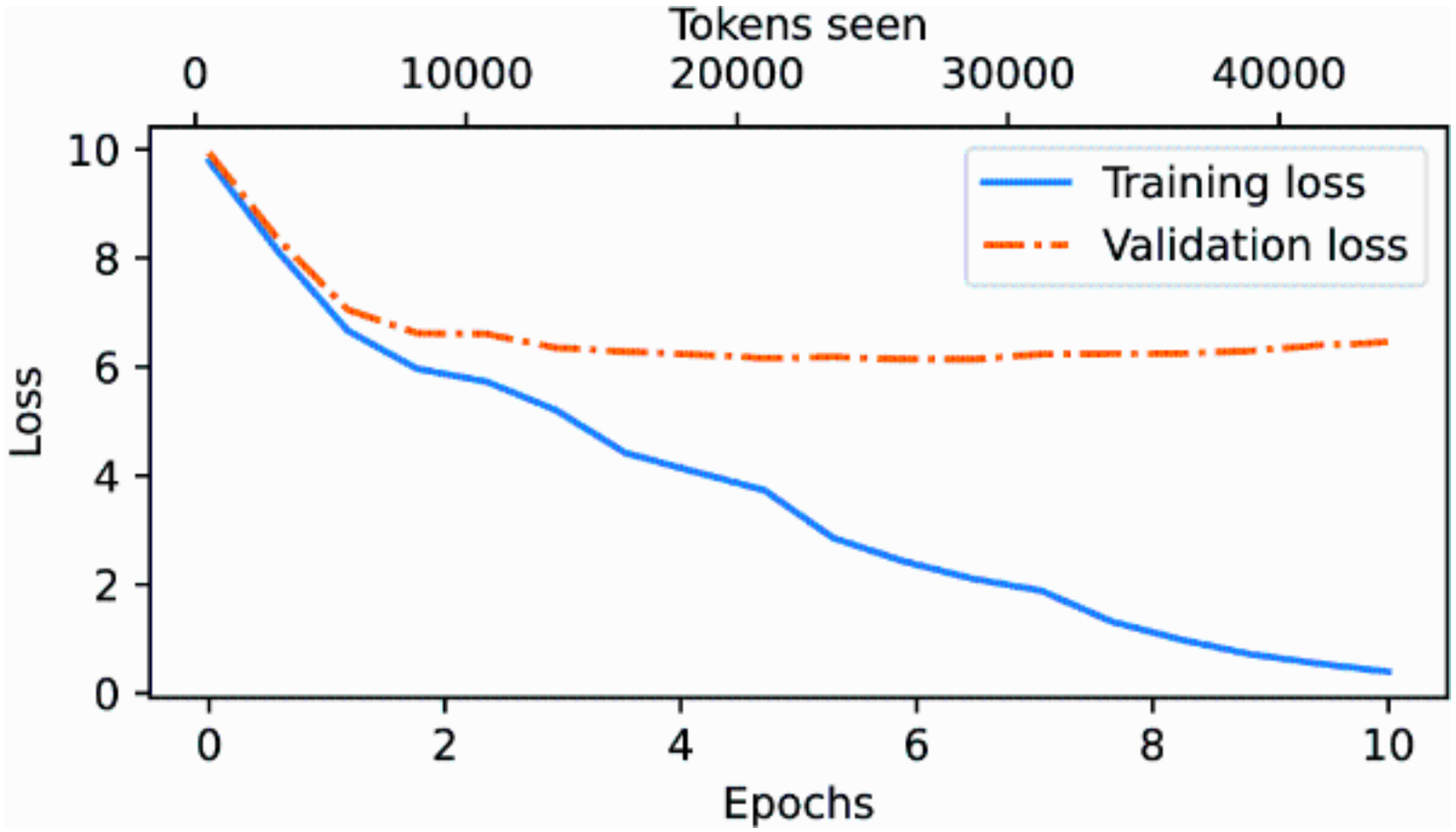
Ep 9 (Step 000080): Train loss 0.563, Val loss 6.462

Every effort moves you?" "Yes--quite insensible to the irony. She wanted him vindicated--and by me!
He laughed again, and threw back the window-curtains, I saw that, and down the room, my eyes

Ep 10 (Step 000085): Train loss 0.358, Val loss 6.542

Every effort moves you?" "Yes--quite insensible to the irony. She wanted him vindicated--and by me!
He laughed again, and threw back his head to look up at the sketch of the donkey. "There were days
when I

Overfitting Past Epoch 2



Always selecting Highest

```
def generate_text_simple(model, idx,
                         max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :]
        probas = torch.softmax(logits, dim=-1)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)

    return idx
```

torch.argmax

Returns the indices of the maximum value of all elements in the input tensor

```
a = torch.randn(4, 4)
a
```

```
tensor([[ 8.9682e-01, -2.1756e+00, -7.1390e-01, -4.4147e-01],
        [ 6.5534e-01,  4.5381e-01,  1.5842e+00, -3.0665e+00],
        [-4.3658e-01,  3.0377e-04,  1.9257e+00,  2.9086e-01],
        [ 3.1287e-01,  4.9243e-02, -5.2300e-01, -1.2458e+00]])
```

```
torch.argmax(a)
```

```
tensor(10)
```

Better Selection of Next Token

Probabilistic Sampling

Temperature Scaling

Top-k Sampling

Probabilistic Sampling

torch.multinomial(input, num_samples, replacement=False, *, generator=None, out=None)

Returns tensor with index selected with probability of the value of that position

```
import numpy as np
import torch

weights = torch.tensor([2.0, 10.0, 8.0, 5.0])

sample = [torch.multinomial(weights, 1).item() for i in range(100)]

sampled_ids = np.bincount(sample)
for i, freq in enumerate(sampled_ids):
        print(f"{freq} x {i}")
```

```
8 x 0
44 x 1
35 x 2
13 x 3
```

Temperature Scaling

Divide logits by a value

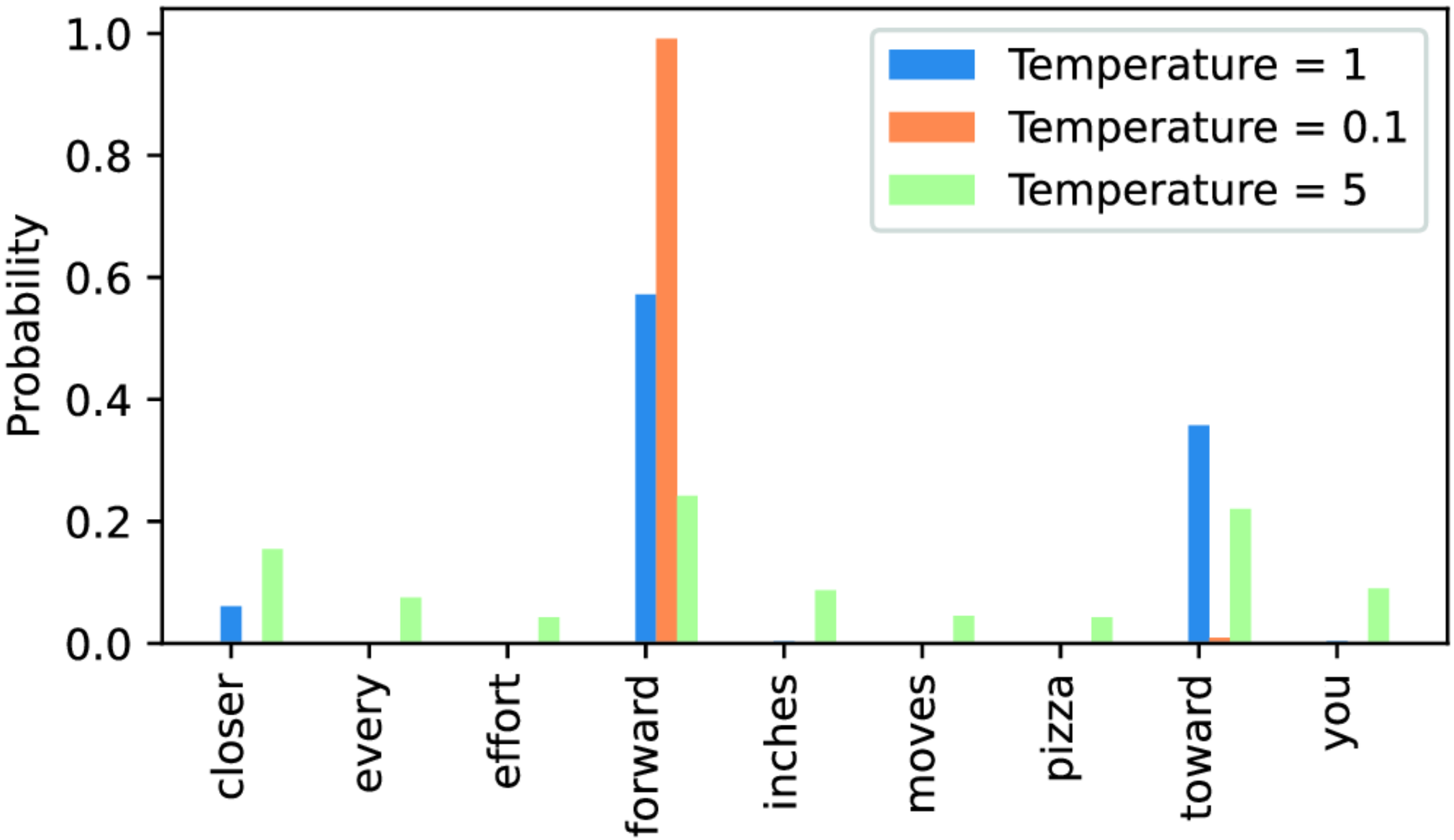
```
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```

Large values push softmax values to extremes

Temperature

Less than 1, makes the model more certain

More than 1 reduces the certainty


Temperature = 1
Temperature = 0.1
Temperature = 5
Probability
1.0
0.8
0.6
0.4
0.2
0.0
closer
every
effort
forward
inches
moves
pizza
toward
you

Top-k Sampling

Only select from the top k items

Benefits

- Enhanced Coherence
- Efficiency

torch.argmax vs Dynamic Token Selection

Every effort moves you know," was one of the axioms he laid down across the Sevres and silver of an exquisitely appointed lun

Every effort moves you stand to work on surprise, a one of us had gone with random-

Saving & Reloading The Model

```
torch.save(model.state_dict(), "model.pth")
```

```
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(torch.load("model.pth", map_location=device))
model.eval()
```

Saving & Reloading The Model

```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    },
    "model_and_optimizer.pth"
)
```

```
checkpoint = torch.load("model_and_optimizer.pth", map_location=device)
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();
```