CS336 Notes

1 Overview

> CS336 Spring 2025

---

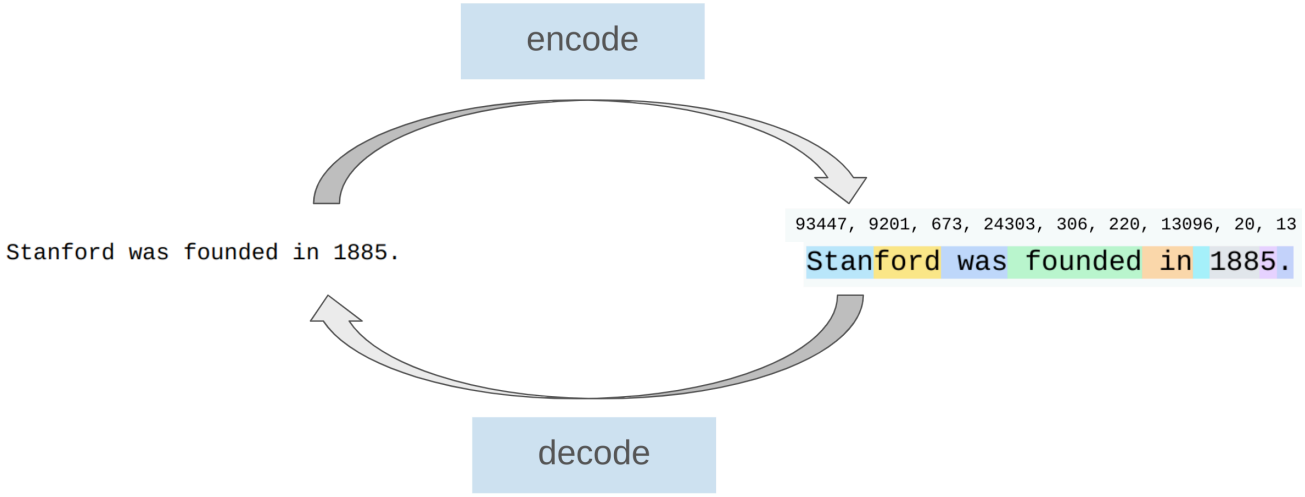
2 Tokenize

A language model places a probability distribution over sequences of tokens (usually integer indices)

> **Example**
>
> `hello, world! 你好!` => `[15496, 11, 995, 0]`

Tokenizer 就是完成将 Strings Encode（编码）到 Tokens 和将 Tokens Decode（解码） 回 Strings 的工具/Class。主要包括两个过程：分词+编码映射

其中字典大小（Vocabulary Size）就是可能出现的所有不同 Tokens 的数量。



> Tokenizer interactive site

2.1 Observations

- A word and its preceding space are part of the same token (e.g., " world").
- A word at the beginning and in the middle are represented differently (e.g., "hello hello").
- Numbers are tokenized into every few digits.

2.2 Character-Base Tokenizer

一个句子（Strings）必然由多个字母组成，对每个字母独立编码，就是 Character-based Tokenization。

Problems:

- 字典会非常大
- characters 的出现频率不同，效率差

2.3 Byte-based Tokenization

将所有字符看作字节（UTF-8）进行编码

- 字典很小（256）
- Problems：压缩太差，Token 序列会很长

2.4 Word-based Tokenization

将句子中每个单词独立映射成 Tokens。

- 需要很大的字典（number of words is huge）
- 很多单词出现频率很低，模型可能不认识他们
- 字典更新很麻烦（出现新的单词 word 不在训练中，模型容易困惑）

2.5 Byte Pair Encoding

在原始文本上训练 Tokenization，自由地处理句子切分
一开始将每个字节作为 Token，后续将经常出现的 Token 组合（pair）合并

---

3 PyTorch & Resource Accounting

Lecture Materials

How long would it take to train a 70B parameter model on 15T tokens on 1024 H100s?

$$\begin{aligned} \text{total} &= 6 \times 70e9 \times 15e12 \\ \text{flops\_per\_day} &= 1979e12/2 \times \text{mfu} \times 1024 \times 60 \times 60 \times 24 \\ \text{days} &= \frac{\text{total}}{\text{flops\_per\_day}} = 143.927 \end{aligned}$$

What's the largest model that can you can train on 8 H100s using AdamW (naively)?

```
h100_bytes = 80e9
bytes_per_parameter = 4 + 4 + (4 + 4) # parameters, gradients, optimizer state @inspect bytes_per_parameter
num_parameters = (h100_bytes * 8) / bytes_per_parameter
```

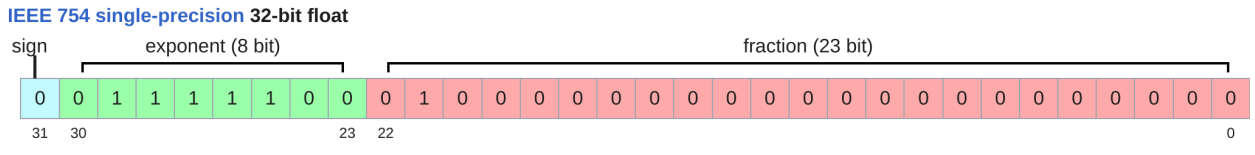
✓ **40, 000, 000, 000 Float32 Parameters**

P.S. Default data type is Float32. Maybe we can try mixed precision train like BF16, but we still need keep an extra float32 copy of the parameters. It has nothing to do with memory, it just speeds up.

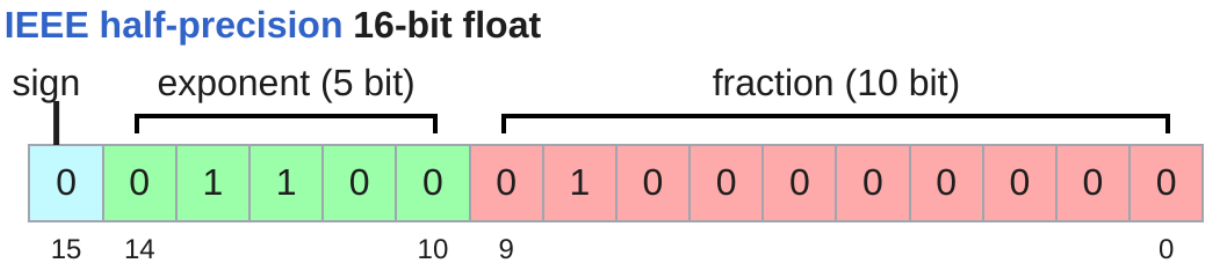
3.1 Floats

- **float32**: default. (fp32, single precision)

  - double precision (float64)

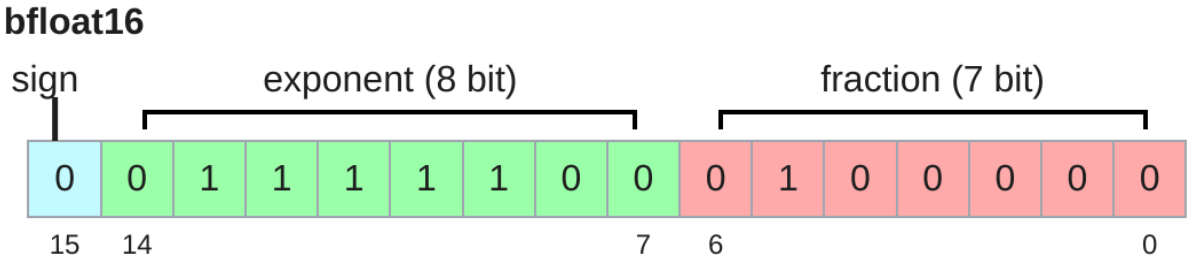


- **float16** (fp16, half precision)

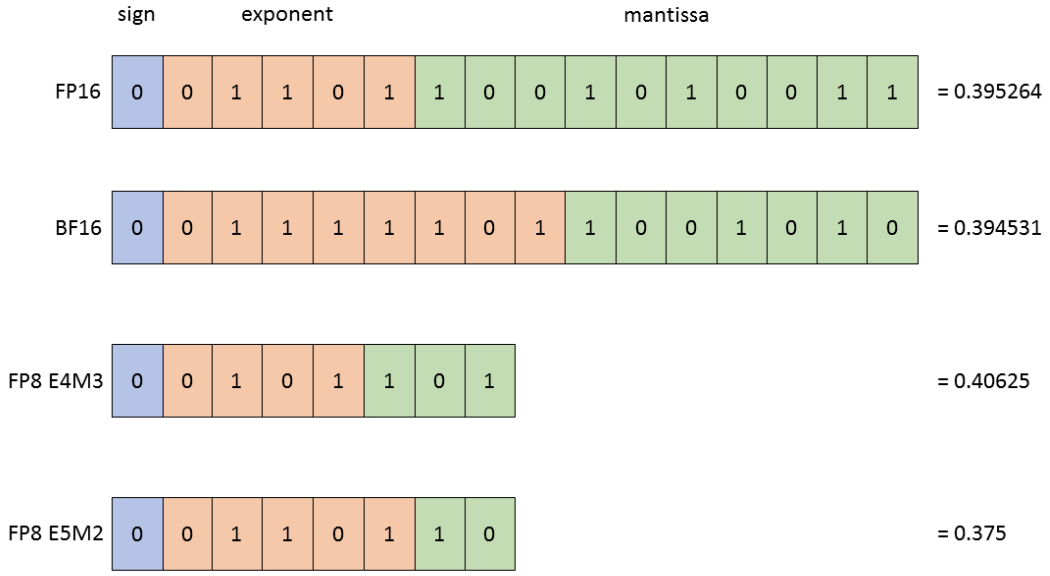


- **bfloat16**: uses the same memory as float16 but has the same dynamic range as float32!

  - The only catch is that the resolution is worse, but this matters less for deep learning.（唯一的问题是精度/分辨率会差一点，但这在深度学习领域并不重要）



- **fp8**

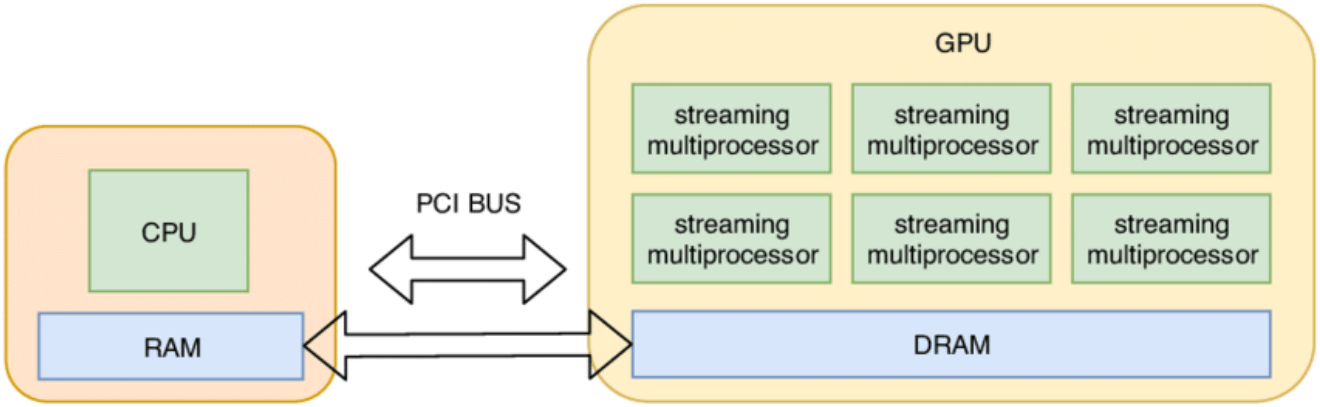


- Mixed Precision Training

3.2 PyTorch Skills

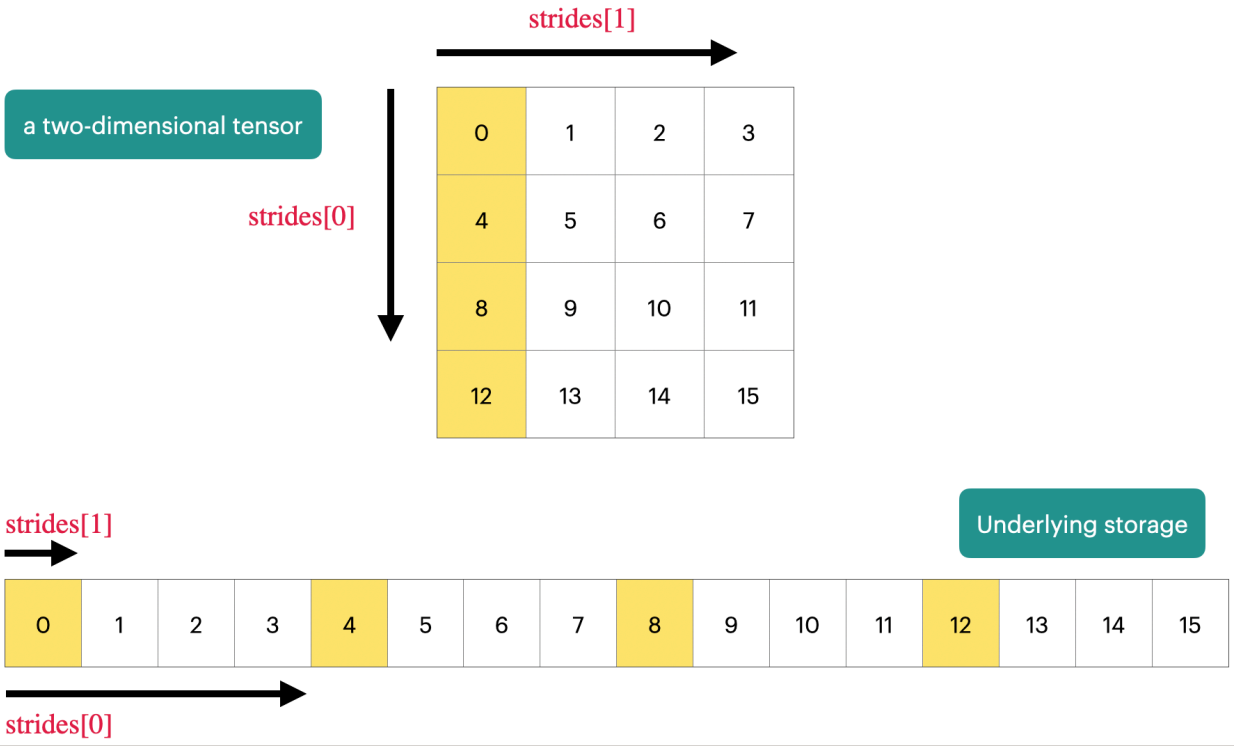
In order to take advantage of the massive parallelism of GPUs, we need to move them to GPU memory.

```
torch.cuda.is_available()
y = x.to("cuda:0")
```



3.2.1 PyTorch Tensors

PyTorch Tensors is pointer into allocated continuous block of memory (row major).

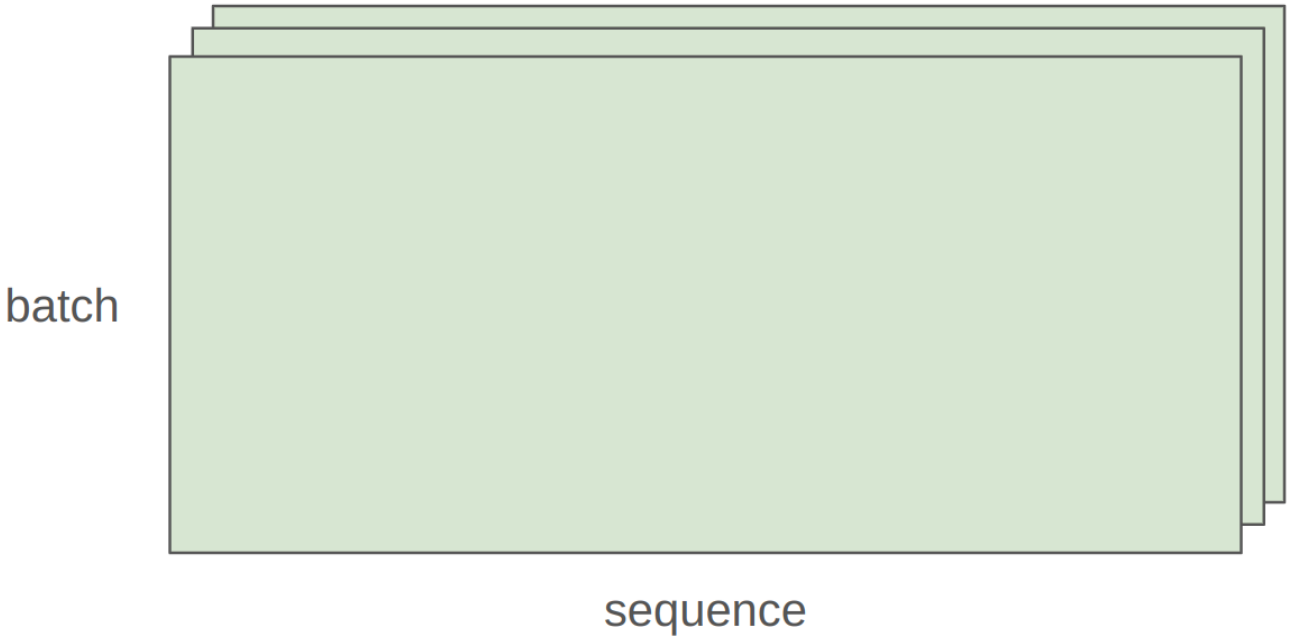


Many operations simply provide a different **view** of the tensor.
Note that some views are non-contiguous entries (**not stored in the logical order you see**), which means that further views aren't possible.

```
x = torch.tensor([[1., 2, 3], [4, 5, 6]]) # @inspect x
y = x.transpose(1, 0) # @inspect y
```

```
assert not y.is_contiguous()
try:
    y.view(2, 3)
    assert False
except RuntimeError as e:
    assert "view size is not compatible with input tensor's
size and stride" in str(e)
```

In general, we perform operations for every example in a batch and token in a sequence.




```
y = x @ w
```

3.2.2 PyTorch Einops

Einops is a library for manipulating tensors where dimensions are named.

```
x = torch.ones(2, 2, 3) # batch, sequence, hidden
y = torch.ones(2, 2, 3) # batch, sequence, hidden
z = x @ y.transpose(-2, -1) # batch, sequence, sequence
```

- jaxtyping

```
# old way
# x = torch.ones(2, 2, 1, 3)
```

```
x: Float[torch.Tensor, "batch seq heads hidden"] =
torch.ones(2, 2, 1, 3)
```

- einops einsum

```
# old way
# z = x @ y.transpose(-2, -1)
z = einsum(x, y, "batch seq1 hidden, batch seq2 hidden ->
batch seq1 seq2")
```

- einops reduce (e.g., sum, mean, max, min)

```
# old way
# y = x.mean(dim=-1)
y = reduce(x, "... hidden -> ...", "sum")
```

- einops rearrange

```
x = rearrange(x, "... (heads hidden1) -> ... heads
hidden1", heads=2)
x = einsum(x, w, "... hidden1, hidden1 hidden2 -> ...
hidden2")
```

3.2.3 Randomness

```
# Torch
seed = 0
torch.manual_seed(seed)

# NumPy
import numpy as np
np.random.seed(seed)

# Python
import random
random.seed(seed)
```

3.3 Computation

- FLOPs: floating-point operations.
- FLOP/s: floating-point operations per second (FLOPS).

Forward & Backward & Update with Batch Size

- Forward Pass: Computes the model outputs. $\propto batch\_size$
- Backward Pass: Computes gradients. $\propto batch\_size$
- Parameter update = Weight update: Updates model parameters using gradients (Optimizer's job). **Stay same** while batch size scaling.

a linear layer (fully connected layer)

$$y = xW,\ x \in \mathbb{R}^{B\times D}, W \in \mathbb{R}^{D\times K}$$

3.3.1 Forward

We have two FLOPs (one ***multiplication*** (x[i][j] * w[j][k]) and one ***addition***) per (i, j, k) triple in a matrix multiplication.

```
# (B, D) @ (D, K)
for i in range(B):
    for j in range(D):
        for k in range(K):
            y[i][k] += x[i][j] * w[j][k] # two FLOPs
```

So in one matrix multiplication we need $2 \times B \times D \times K$ FLOPs.

- elementswise operation needs $O(mn)$ FLOPs.
    - Addition of two $m \times n$ matrices requires $m \times n$ FLOPs.
- FLOP/s depends on **hardware** and **data type** (e.g. BF16 vs Float32)
- Model FLOPs Utilization(MFU): (actual FLOP/s) / (promised FLOP/s)
    - ignore communication/overhead

- = 0.5 is very good.

No other operation that you'd encounter in deep learning is as expensive as matrix multiplication

3.3.2 Backward & Gradients Computation

Backward Zhihu

In the backward pass, we compute the gradients of the loss L with respect to the weights W, the input x.

1. The first step is to compute the gradient of the loss function L with respect to each output y, i.e., $\frac{\partial L}{\partial y}$.

For example, assume we are using Mean Squared Error (MSE) as the loss function:

$$L = \frac{1}{2} \sum_{i=1}^{B} \|y_i - \hat{y}_i\|^2$$
$$\frac{\partial L}{\partial y_i} = y_i - \hat{y}_i$$

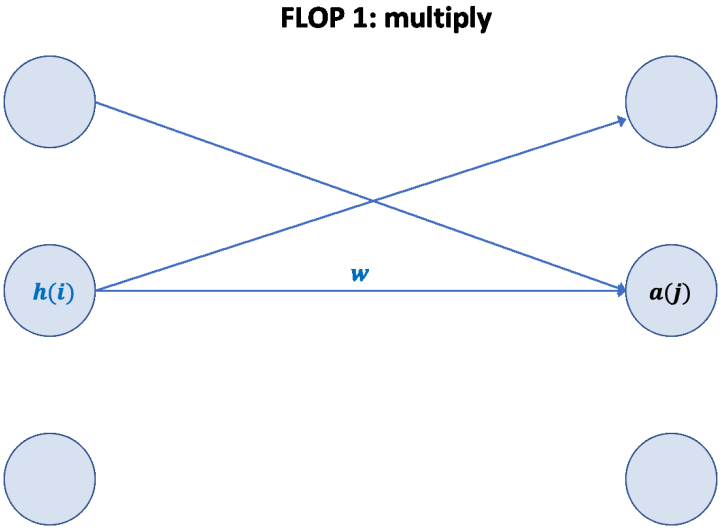
where $\hat{y}_i$ is the true label.

- $B \times K$ FLOPs ~~(ignore)~~

2. Gradient with Respect to the Weights and Inputs.

$$\frac{\partial L}{\partial W} = x^T \cdot \frac{\partial L}{\partial y},\ x^T \in \mathbb{R}^{D \times B}, \frac{\partial L}{\partial y} \in \mathbb{R}^{B \times K}$$
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot W^T,\ W^T \in \mathbb{R}^{K \times D}$$

- $2 \times D \times B \times K + 2 \times B \times K \times D$ FLOPs



3. Gradient with Respect to the Bias.

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}$$

- $B \times K$ FLOPs (~~Reuse. ignore~~)

4. Optimizer (e.g. Adam): Formula Derivation

- $D \times K$ = (#parameters) FLOPs (~~ignore~~)

Summary:

- Forward pass: 2 (#data points) (#parameters) FLOPs
- Backward pass: 4 (#data points) (#parameters) FLOPs
- Total: 6 (#data points) (#parameters) FLOPs

3.4 Parameter Initialization & Data Loading

3.4.1 nn.Parameter()

```
w = nn.Parameter(torch.randn(input_dim, output_dim) /
np.sqrt(input_dim))
```

Large values can cause gradients to blow up and cause training to be unstable. So, we simply rescale by $1/\sqrt{\text{input\_dim}}$.

3.4.2 Data Loader

Don't want to load the entire data into memory at once.
so use `memmap` to lazily load only the accessed parts into memory.

```
data = np.memmap("data.npy", dtype=np.int32)
```

A **data loader** generates a **batch** of sequences for training.

```
start_indices = torch.randint(len(data) - sequence_length,
(batch_size,))

x = torch.tensor([data[start:start + sequence_length] for
start in start_indices])
```

3.4.3 Pin Memory

By default, PyTorch CPU tensors are stored in **paged memory**. Because of the OS's paging mechanism, this memory may be swapped out to disk.
However, that also means the GPU **cannot use direct memory access (DMA)** efficiently to read from it, which slows down (CPU => GPU) data transfers.
So `pin_memory()` moves the tensor `x` from normal paged memory to **page-locked (pinned) memory**, which allows us to copy `x` from CPU into GPU **asynchronously**.

```
if torch.cuda.is_available():
    x = x.pin_memory()

x = x.to(device, non_blocking=True)
```

This allows us to do two things in parallel (not done here):

- Fetch the next batch of data into CPU
- Process `x` on the GPU.

3.5 Optimizer

- momentum = SGD + exponential averaging of grad
- AdaGrad = SGD + averaging by grad^2
- RMSProp = AdaGrad + exponentially averaging of grad^2
- Adam = RMSProp + momentum

```
optimizer = AdaGrad(model.parameters(), lr=0.01)
optimizer.step()
optimizer.zero_grad(set_to_none=True) # free up memory
```

3.6 Memory

```
# B points x D dimension
# Parameters
num_parameters = (D * D * num_layers) + D
assert num_parameters == get_num_parameters(model)

# Activations: element wise
num_activations = B * D * num_layers

# Gradients
num_gradients = num_parameters

# Optimizer states
num_optimizer_states = num_parameters

# Putting it all together, assuming float32
total_memory = 4 * (num_parameters + num_activations +
num_gradients + num_optimizer_states)
```

3.7 Checkpoint

During training, it is useful to periodically save your model and optimizer state to disk.

```
checkpoint = {
    "model": model.state_dict(),
    "optimizer": optimizer.state_dict(),
}
torch.save(checkpoint, "model_checkpoint.pt")

# Load the checkpoint:
loaded_checkpoint = torch.load("model_checkpoint.pt")
```

3.8 Mixed Precision Training

> https://arxiv.org/pdf/1710.03740.pdf

Choice of data type (float32, bfloat16, fp8) have tradeoffs.

- Higher precision: more accurate/stable, more memory, more compute
- Lower precision: less accurate/stable, less memory, less compute

Solution: use float32 by default, but use {bfloat16, fp8} when possible.
A concrete plan:

- Use {bfloat16, fp8} for the forward pass (activations).
- Use float32 for the rest (parameters, gradients).

PyTorch has an automatic mixed precision (AMP) library.
https://pytorch.org/docs/stable/amp.html
https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/

NVIDIA's Transformer Engine supports FP8 for linear layers.

---

4 Architectures

- Pre vs Post Norm
- Layer Norm vs RMS Norm
    - RMS Faster and cheaper in memory: FLOPs is not Runtime
    - drop bias