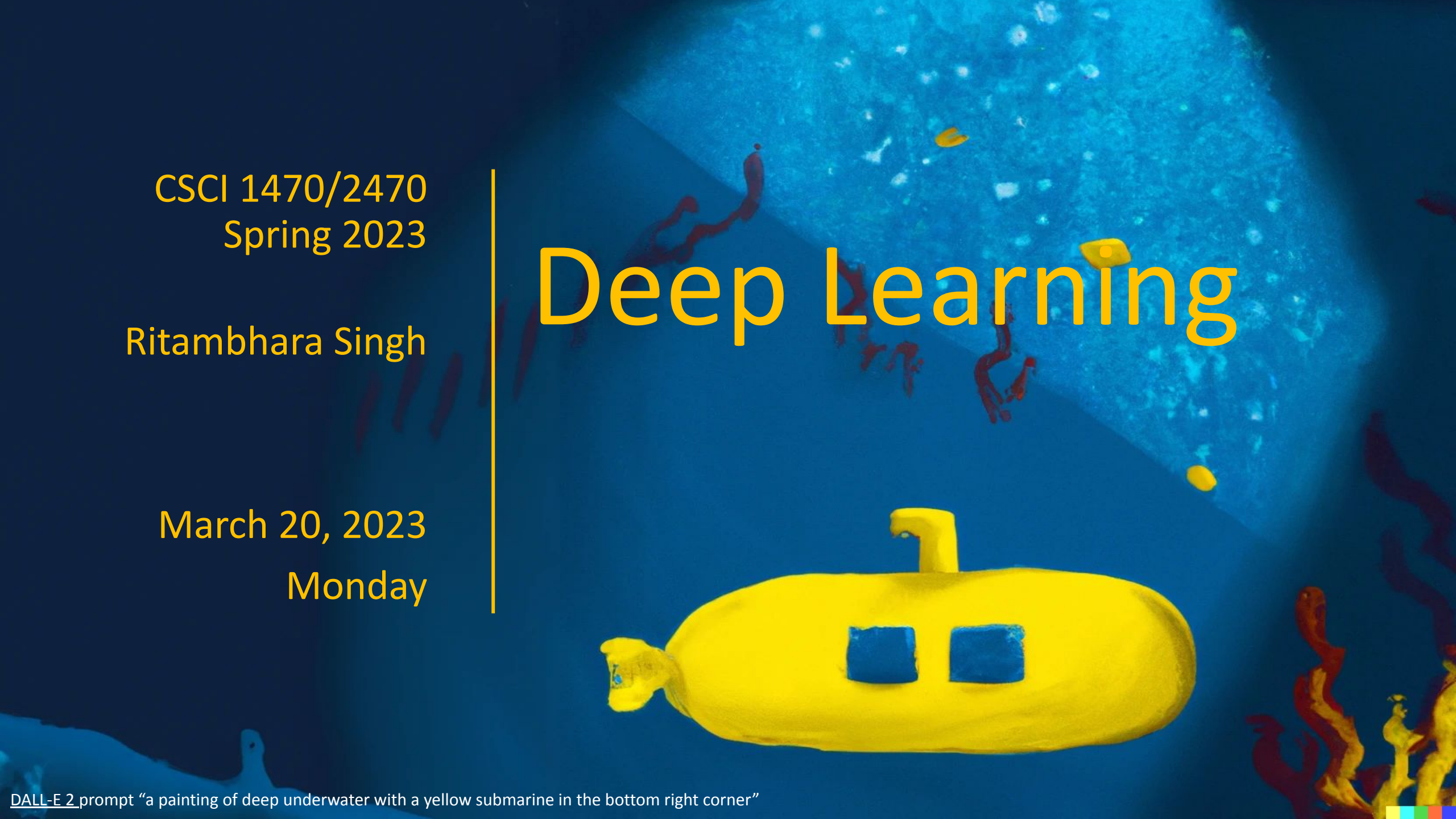
CSCI 1470/2470
Spring 2023

Ritambhara Singh

March 20, 2023

Monday

Deep Learning



DALL-E 2 prompt “a painting of deep underwater with a yellow submarine in the bottom right corner”

What to do when DL systems get ***big***

- Big = big data
- Big = big model

Today's goal – learn about scaling deep learning models and sustainable deep learning

(1) Managing memory constraints

(2) Distributing work across processors, GPUs, machines

(3) Development of sustainable DL systems

- Near-term solutions
- Mid-term solutions
- Long-term solutions

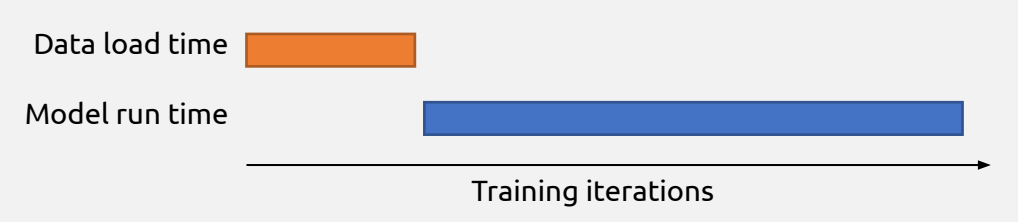
Scaling: Some Key Questions

- **What do I do when my dataset won't fit in memory?**
- Can I use multiple processors to train faster?
- Can I use multiple GPUs to train faster (or train bigger models?)
- Can I use multiple machines to train faster (or train bigger models?)

Review: Consequences of batched data loading

- Great! We can train/test on all our data without blowing out memory
- But, there's a price to pay:
  - More time loading data, in general
  - Disk is idle while model is running
- ***What can we do about this?***

Loading data all at once:

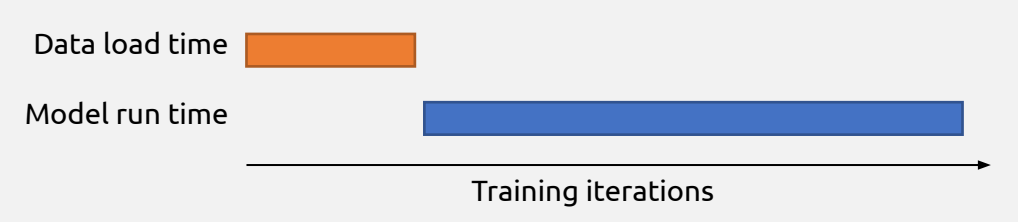


Batched data loading:

Consequences of batched data loading

- Great! We can train/test on all our data without blowing out memory
- But, there's a price to pay:
  - More time loading data, in general
  - Disk is idle while model is running
- ***What can we do about this?***

Loading data all at once:



Batched data loading:

Reducing data loading time

- Use the fastest disk you can get your hands on

Magnetic Disk Drive (HDD)

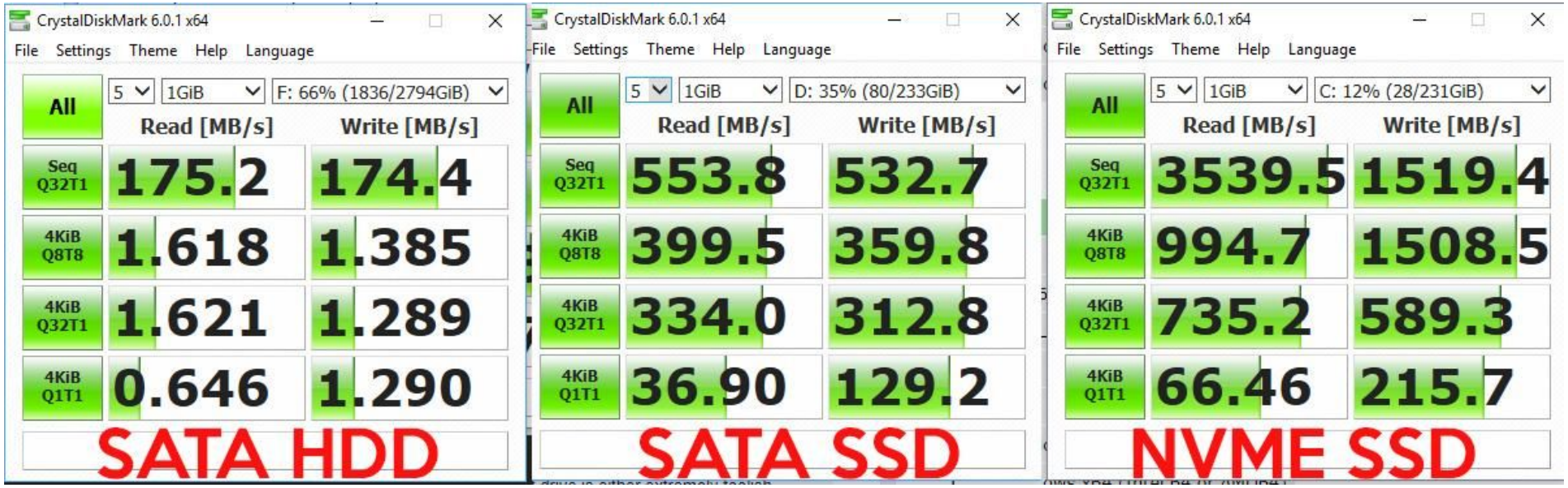
SATA Solid State Disk (SDD)

NVMe Solid State Disk (NVMe)

https://unihost.com/help/nvme-vs-ssd-vs-hdd-overview-and-comparison/

Reducing data loading time

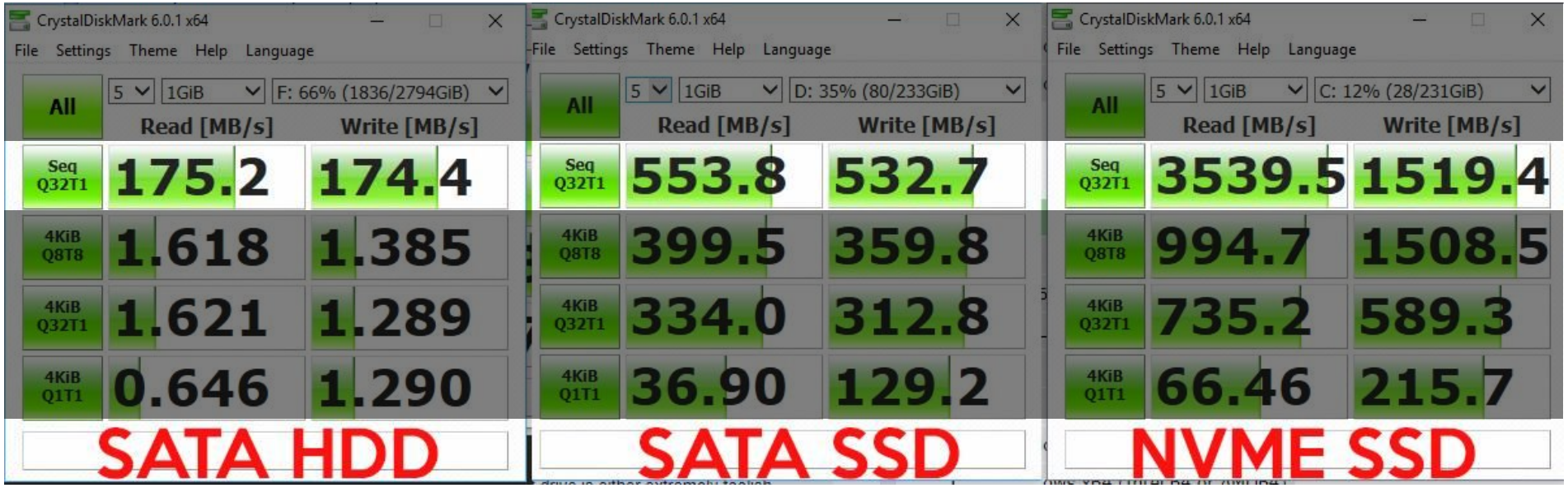
- Use the fastest disk you can get your hands on



https://unihost.com/help/nvme-vs-ssd-vs-hdd-overview-and-comparison/

Reducing data loading time

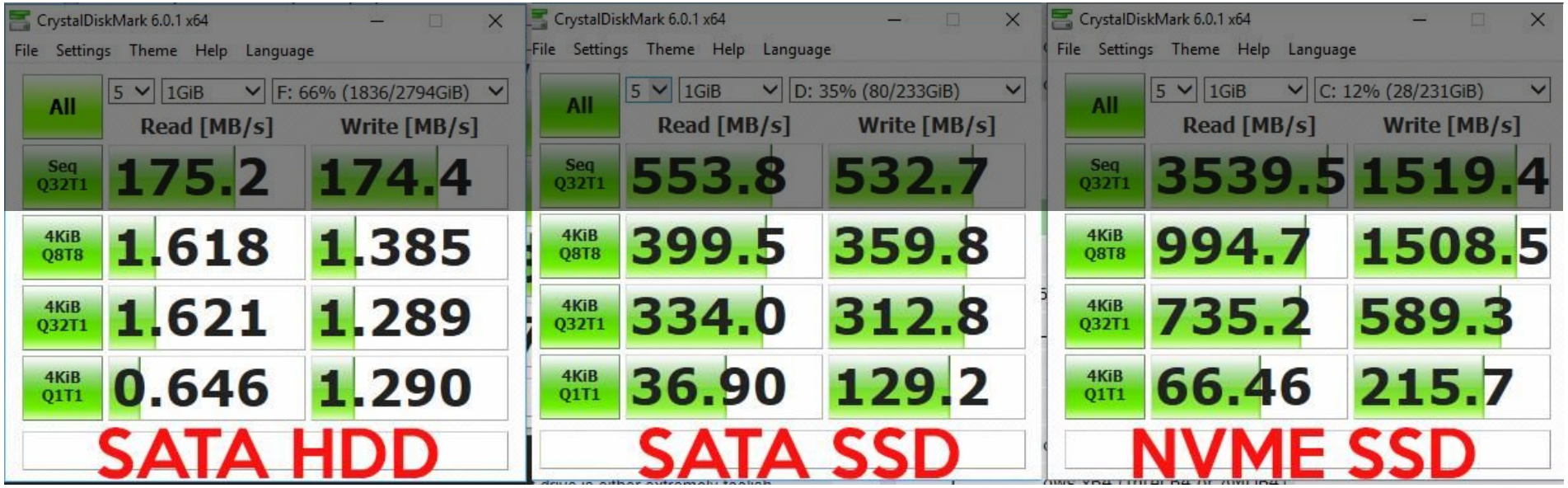
- Sequential accesses (memory addresses are contiguous)



https://unihost.com/help/nvme-vs-ssd-vs-hdd-overview-and-comparison/

Reducing data loading time

- Random accesses

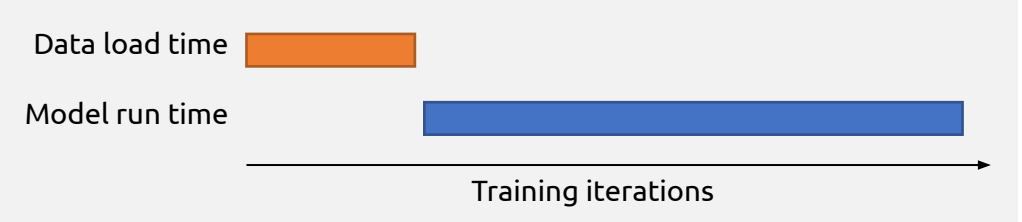


https://unihost.com/help/nvme-vs-ssd-vs-hdd-overview-and-comparison/

Consequences of batched data loading

- Great! We can train/test on all our data without blowing out memory
- But, there's a price to pay:
  - More time loading data, in general
  - Disk is idle while model is running
- ***What can we do about this?***

Loading data all at once:

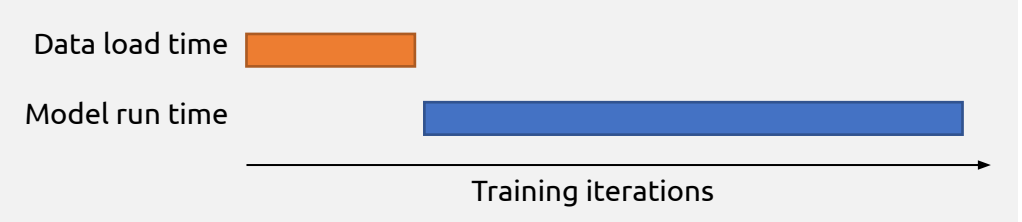


Batched data loading:

Consequences of batched data loading

- Great! We can train/test on all our data without blowing out memory
- But, there's a price to pay:
  - More time loading data, in general
  - Disk is idle while model is running
- ***What can we do about this?***

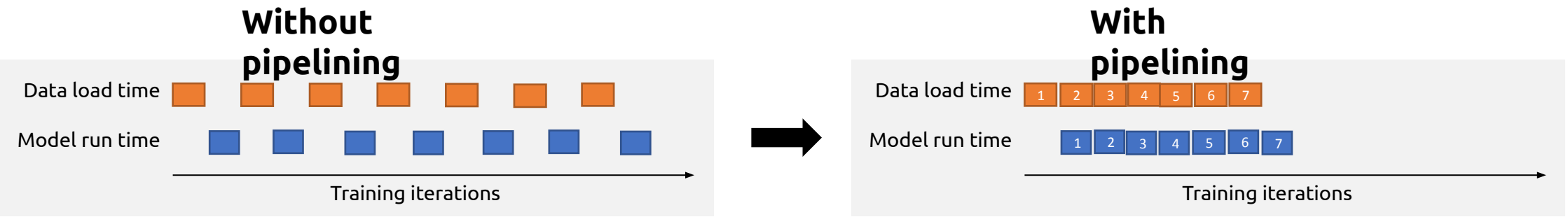
Loading data all at once:



Batched data loading:

CPU/GPU Pipelining

- If we train our model on the GPU...
- ...then the CPU is free to handle data loading while the model is running
- GPU doesn't have to "wait" on the CPU to load data



In Tensorflow: tf.data.Dataset does this

```
# Create a Dataset that contains all .jpg files
# in a directory
dir_path = dir_name + '/*.jpg’
dataset = tf.data.Dataset.list_files(dir_path)

# Apply a function that will read the contents of #
each file into a tensor
dataset =
dataset.map(map_func=load_and_process_image)

# Load up data in batches
dataset = dataset.batch(batch_size)

# Iterate over dataset
for i, batch in enumerate(dataset):
   # processing code goes here
```

```
def load_and_process_image(file_path):
   # Load image
   image = tf.io.decode_jpeg(
      tf.io.read_file(file_path),
      channels=3)

   # Convert image to normalized float [0, 1]
   image = tf.image.convert_image_dtype(
      image,
      tf.float32)

   # Rescale data to range (-1, 1)
   image = (image - 0.5) * 2
   return image
```

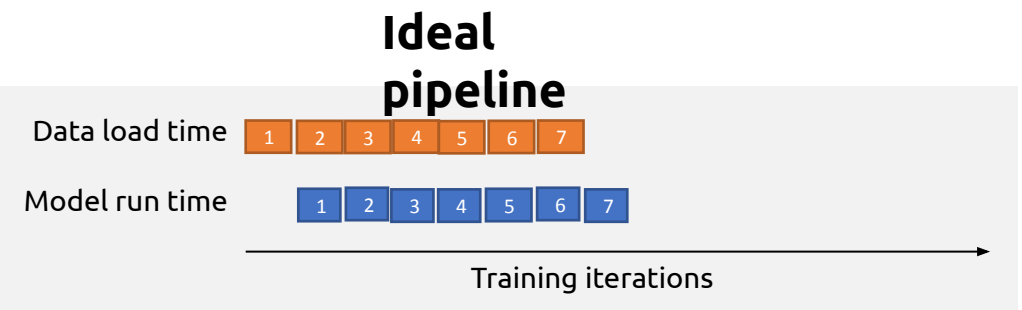
In Tensorflow: tf.data.Dataset does this

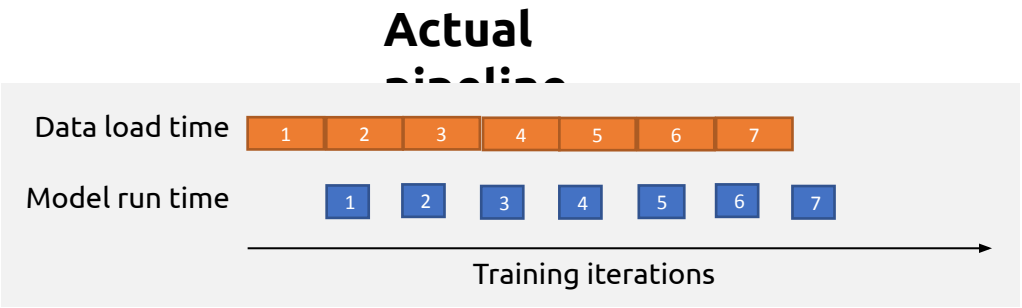
```
# Create a Dataset that contains all .jpg files
# in a directory
dir_path = dir_name + '/*.jpg’
dataset = tf.data.Dataset.list_files(dir_path)

# Apply a function that will read the contents of #
each file into a tensor
dataset =
dataset.map(map_func=load_and_process_image)

# Load up data in batches
dataset = dataset.batch(batch_size)

# Prefetch the next batch while GPU is training
dataset = dataset.prefetch(1)

# Iterate over dataset
for i, batch in enumerate(dataset):
   # processing code goes here
```

```
def load_and_process_image(file_path):
   # Load image
   image = tf.io.decode_jpeg(
      tf.io.read_file(file_path),
      channels=3)

   # Convert image to normalized float [0, 1]
   image = tf.image.convert_image_dtype(
      image,
      tf.float32)

   # Rescale data to range (-1, 1)
   image = (image - 0.5) * 2
   return image
```

The pipeline isn't always so perfect

- If you have large batch sizes, high-resolution images, etc., it can take longer to load the batch from disk than it takes the model to run





- GPU sits idle, wasting compute potential
  - This is an example of an ***I/O bound*** program (i.e. the bottleneck is disk I/O)
  - A ***compute-bound*** program = bottleneck is processor speed
  - A ***memory-bound*** program = bottleneck is memory read/write throughput
- **What can we do?**

Scaling: Some Key Questions

- **What do I do when my dataset won't fit in memory?**
- Can I use multiple processors to train faster?
- Can I use multiple GPUs to train faster (or train bigger models?)
- Can I use multiple machines to train faster (or train bigger models?)

Scaling: Some Key Questions

- What do I do when my dataset won't fit in memory?
- **Can I use multiple processors to train faster?**
- Can I use multiple GPUs to train faster (or train bigger models?)
- Can I use multiple machines to train faster (or train bigger models?)

In Tensorflow: tf.data.Dataset does this

```
# Create a Dataset that contains all .jpg files
# in a directory
dir_path = dir_name + '/*.jpg’
dataset = tf.data.Dataset.list_files(dir_path)

# Apply a function that will read the contents of #
each file into a tensor
dataset =
dataset.map(map_func=load_and_process_image)

# Load up data in batches
dataset = dataset.batch(batch_size)

# Prefetch the next batch while GPU is training
dataset = dataset.prefetch(1)

# Iterate over dataset
for i, batch in enumerate(dataset):
   # processing code goes here
```

```
def load_and_process_image(file_path):
   # Load image
   image = tf.io.decode_jpeg(
      tf.io.read_file(file_path),
      channels=3)

   # Convert image to normalized float [0, 1]
   image = tf.image.convert_image_dtype(
      image,
      tf.float32)

   # Rescale data to range (-1, 1)
   image = (image - 0.5) * 2
   return image
```

Parallel Data Loading

- If you're doing preprocessing on each loaded datum to prepare it for training, then the program may actually be compute-bound.
  - Disk is capable of higher read throughput, but the CPU is has to spend time doing stuff to one datum before it can request the next one.

Any ideas?

- Solution: use multiple CPUs!

In Tensorflow: tf.data.Dataset does this

```
# Create a Dataset that contains all .jpg files
# in a directory
dir_path = dir_name + '/*.jpg’
dataset = tf.data.Dataset.list_files(dir_path)

# Apply a function that will read the contents of #
each file into a tensor
dataset =
dataset.map(map_func=load_and_process_image)

# Load up data in batches
dataset = dataset.batch(batch_size)

# Prefetch the next batch while GPU is training
dataset = dataset.prefetch(1)

# Iterate over dataset
for i, batch in enumerate(dataset):
   # processing code goes here
```

```
def load_and_process_image(file_path):
   # Load image
   image = tf.io.decode_jpeg(
      tf.io.read_file(file_path),
      channels=3)

   # Convert image to normalized float [0, 1]
   image = tf.image.convert_image_dtype(
      image,
      tf.float32)

   # Rescale data to range (-1, 1)
   image = (image - 0.5) * 2
   return image
```

In Tensorflow: tf.data.Dataset does this

```
# Create a Dataset that contains all .jpg files
# in a directory
dir_path = dir_name + '/*.jpg'
dataset = tf.data.Dataset.list_files(dir_path)

# Apply a function that will read the contents of #
each file into a tensor
dataset =
dataset.map(map_func=load_and_process_image,
   num_parallel_calls=8)

# Load up data in batches
dataset = dataset.batch(batch_size)

# Prefetch the next batch while GPU is training
dataset = dataset.prefetch(1)

# Iterate over dataset
for i, batch in enumerate(dataset):
   # processing code goes here
```

```
def load_and_process_image(file_path):
   # Load image
   image = tf.io.decode_jpeg(
      tf.io.read_file(file_path),
      channels=3)

   # Convert image to normalized float [0, 1]
   image = tf.image.convert_image_dtype(
      image,
      tf.float32)

   # Rescale data to range (-1, 1)
   image = (image - 0.5) * 2
   return image
```

A note about Python parallelism

- Python does not support multithreading
  - One Python process = one thread
- tf.data.Dataset gets around this using multiprocessing
  - Each ‘thread’ is actually a separate Python process (with its own interpreter)
- This means there are limits to what your per-datum preprocessing functions can do.
  - In particular, no shared access to the state of Python objects.

Example: keeping track of images loaded

```
def load_and_process_image(file_path):
    # Load image
    image = tf.io.decode_jpeg(
        tf.io.read_file(file_path),
        channels=3)

    # Convert image to normalized float [0, 1]
    image = tf.image.convert_image_dtype(
        image,
        tf.float32)

    # Rescale data to range (-1, 1)
    image = (image - 0.5) * 2
    return image
```

Example: keeping track of images loaded

Will this work?

This will not work!

```
images_loaded = []
def load_and_process_image(file_path):
    images_loaded.append(file_path)
    # Load image
    image = tf.io.decode_jpeg(
        tf.io.read_file(file_path),
        channels=3)

    # Convert image to normalized float [0, 1]
    image = tf.image.convert_image_dtype(
        image,
        tf.float32)

    # Rescale data to range (-1, 1)
    image = (image - 0.5) * 2
    return image
```

Scaling: Some Key Questions

- What do I do when my dataset won't fit in memory?
- **Can I use multiple processors to train faster?**
- Can I use multiple GPUs to train faster (or train bigger models?)
- Can I use multiple machines to train faster (or train bigger models?)

Scaling: Some Key Questions

- What do I do when my dataset won't fit in memory?
- Can I use multiple processors to train faster?
- **Can I use multiple GPUs to train faster (or train bigger models?)**
- Can I use multiple machines to train faster (or train bigger models?)

Multi-GPU Training

- Thus far, we've just talked about using multiple CPUs for data loading
- But we can also talk about how to split up the model's computation onto multiple ***GPUs***, if we have more than one available.

Multi-GPU Training

Two major ways to split up the model computation

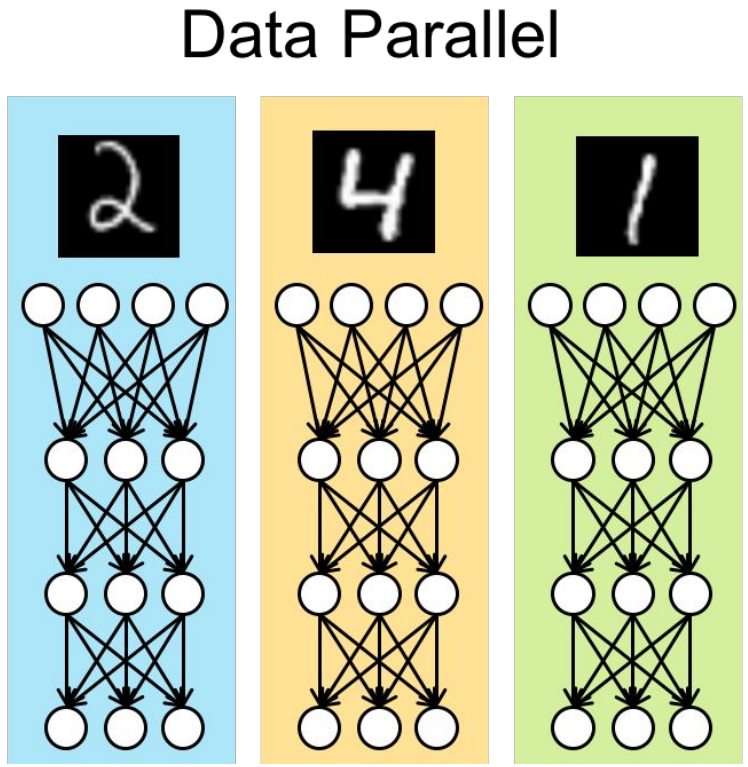
https://medium.com/@esaliya/model-parallelism-in-deep-learning-is-not-what-you-think-94d2f81e82ed

Multi-GPU Training

Two major ways to split up the model computation

Data parallelism:

compute gradients on larger batches by splitting the batches into smaller sub-batches, one per GPU.

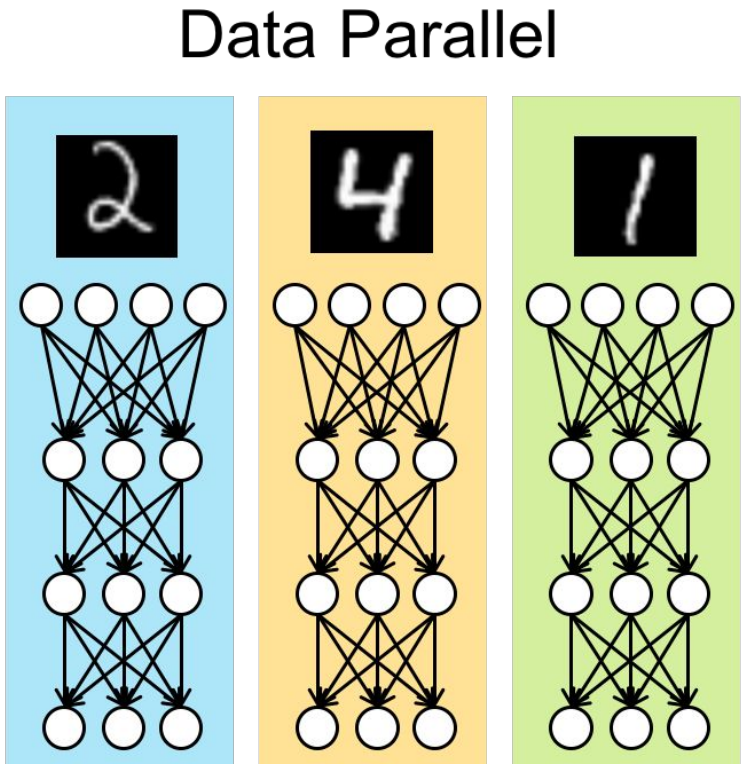


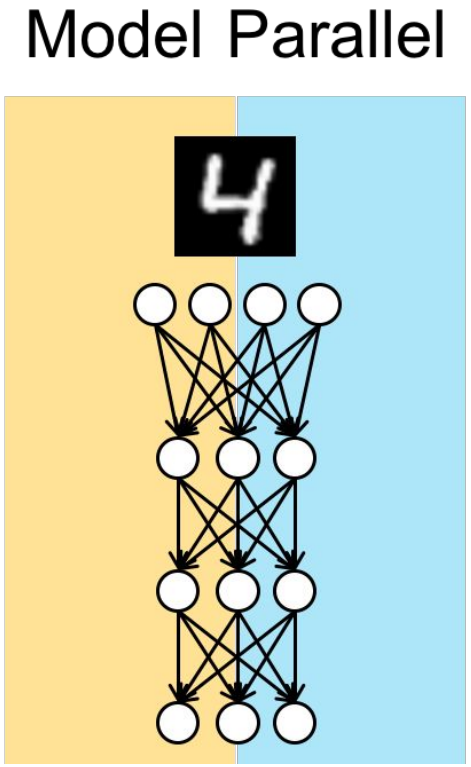
Multi-GPU Training

Two major ways to split up the model computation

Data parallelism:

compute gradients on larger batches by splitting the batches into smaller sub-batches, one per GPU.



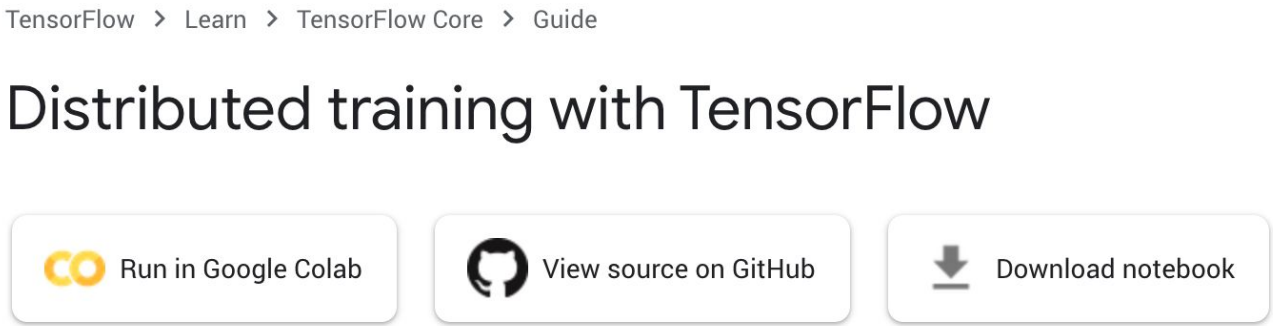


Model parallelism:

compute different parts of the model on different GPUs (Weights of FC layer)

Multi-GPU training in Tensorflow

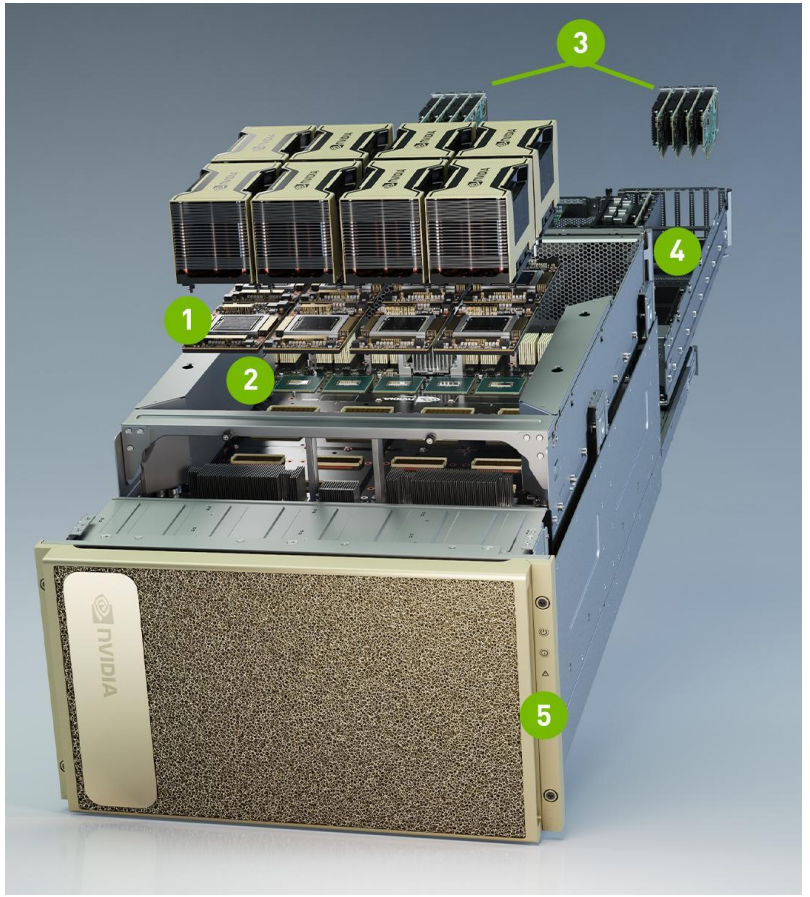
- Tensorflow provides an API for data parallelism across multiple GPUs



- Workload partitioning: TF has no utilities for automatically splitting up your model across multiple GPUs. You'd have to engineer that yourself.

Hardware support for multi-GPU training

- NVIDIA cards support NVLink, which allows for fast direct memory transfer between GPUs
- Their line of DGX workstations/supercomputers takes advantage of this feature
  - But they cost upwards of $40k...



Scaling: Some Key Questions

- What do I do when my dataset won't fit in memory?
- Can I use multiple processors to train faster?
- **Can I use multiple GPUs to train faster (or train bigger models?)**
- Can I use multiple machines to train faster (or train bigger models?)

Scaling: Some Key Questions

- What do I do when my dataset won't fit in memory?
- Can I use multiple processors to train faster?
- Can I use multiple GPUs to train faster (or train bigger models?)
- **Can I use multiple machines to train faster (or train bigger models?)**

Distributed Training

- What if multiple GPUs isn't enough? Suppose your model (or your data) is so big, the only way you can train it in a reasonable amount of time is to use GPUs on a whole cluster of machines?
  - Side note: unlikely to be a problem for you unless you're in a big company with tons of data and compute resource (e.g. Google/Facebook)

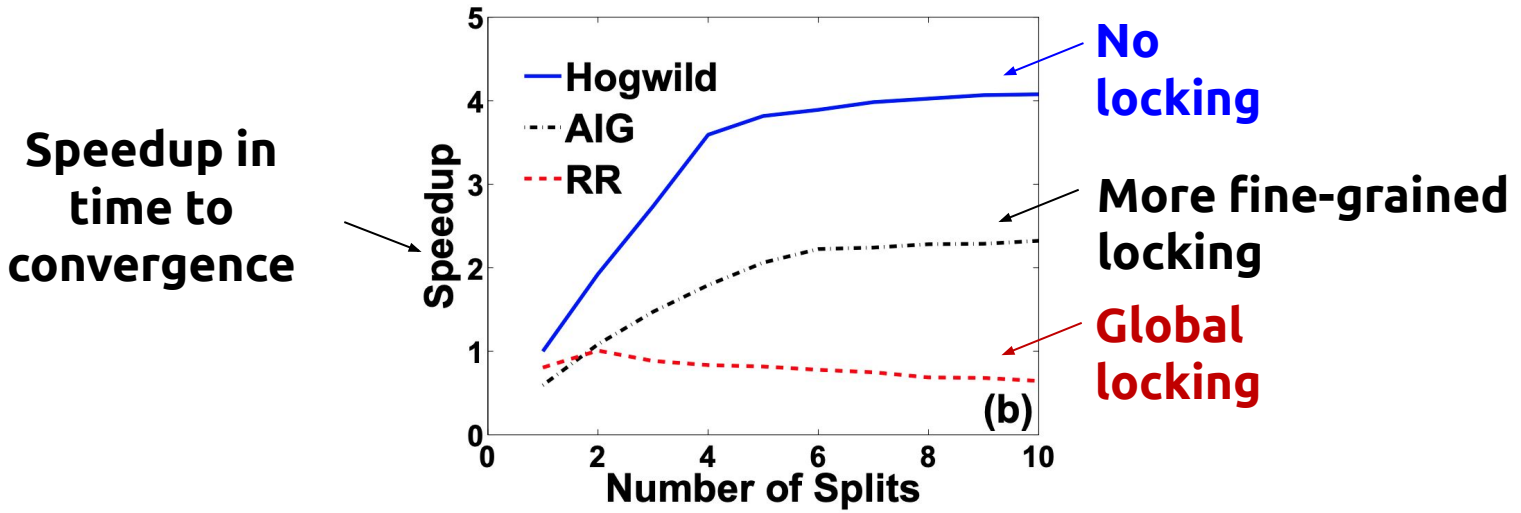
[Minerva supercomputer, Max Planck Institute for Gravitational Physics]

Distributed Training

- Recall our definition of data parallelism:
  - **Data parallelism:** compute gradients on larger batches by splitting the batches into smaller sub-batches, one per GPU.
- If we're splitting up the batch across a cluster of, say, 100 machines, then we need to wait for all 100 machines to compute their gradients before we can update the model's parameters.
- This ***synchronization bottleneck*** slows things down, preventing us from getting the 100x speedup we might expect.
- What if we just didn't synchronize, and let each machine use its gradient to update the model parameters whenever it's ready?
- **Do you think this will work?**

Lock-free parallel gradient updates?

- Surprisingly, lock-free gradient updating actually ***does*** work.
- Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent was the first paper to show this.
- Achieves optimal convergence rates in theory if the gradient updates are sparse. Even if they're not, it often performs well in practice.



Any questions?

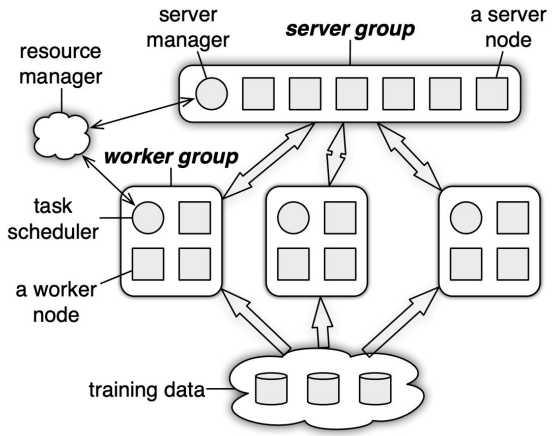
Architectures for distributed training

- Lots of decisions to make:
  - How many nodes should be workers?
  - How many nodes should store/update model parameters?
- ParameterServer is one prominent architecture for managing this design space

tf.distribute.experimental.ParameterServerStrategy

TensorFlow 1 version | View source on GitHub

An multi-worker tf.distribute strategy with parameter servers.



Today's goal – learn about scaling deep learning models and sustainable deep learning

(1) Managing memory constraints

(2) Distributing work across processors, GPUs, machines

(3) Development of sustainable DL systems

- **Near-term solutions**
- **Mid-term solutions**
- **Long-term solutions**

How do we train and run our neural nets?

```
$ python3 <whatever_script>.py
```

What's *actually* happening when we run the script?

Three options (in this class):

1. The network trains locally on your CPU
2. The network trains on Brown CS department GPUs
3. The network trains remotely on GPUs owned by Google (GCP)

(in the world, could also likely be)

4. The network trains remotely on GPUs owned by Amazon (AWS)
5. The network trains remotely on GPUs owned by Microsoft

For example:

The server farm for processing data at CERN

A closer look at power consumption <—> GPUs

- How much power does a GPU use while training a network?
  - Depends on the GPU, but it is ***a lot***
- Strubell et al. (2019) estimated the power consumption involved in training state of the art neural networks (GPT2, Transformer, ELMo, BERT)

Carbon emissions the average American produces in a year:

36,156 lbs

Train an NLP pipeline: how many times of the average US yearly emissions?

What about a Transformer?

Go to **www.menti.com** and use the code **4138 7474**

Train an NLP pipeline (incl. tuning/experimentation):

~78,500 lbs (2.17x US yearly)

Train a transformer pipeline (like GPT-2):

3x greater than average US yearly emissions?
5x? 10x? More??

Train a transformer pipeline (incl. tuning/experimentation):

~626,000 lbs (17.32x US yearly)

Consumption	CO_2e (lbs)
Air travel, 1 passenger, NY↔SF	1984
Human life, avg, 1 year	11,023
American life, avg, 1 year	36,156
Car, avg incl. fuel, 1 lifetime	126,000
Training one model (GPU)	
NLP pipeline (parsing, SRL)	39
w/ tuning & experimentation	78,468
Transformer (big)	192
w/ neural architecture search	626,155

Table 1: Estimated CO_2 emissions from training common NLP models, compared to familiar consumption.[1]

From Strubell et al., 2019

That’s a lot of power!

…where is it coming from?

How is “the cloud” powered?

Consumer	Renew.	Gas	Coal	Nuc.
China	22%	3%	65%	4%
Germany	40%	7%	38%	13%
United States	17%	35%	27%	19%
Amazon-AWS	17%	24%	30%	26%
Google	56%	14%	15%	10%
Microsoft	32%	23%	31%	10%

Table 2: Percent energy sourced from: Renewable (e.g. hydro, solar, wind), natural gas, coal and nuclear for the top 3 cloud compute providers (Cook et al., 2017), compared to the United States,[4] China[5] and Germany (Burger, 2019).

From Strubell et al., 2019

How can DL be more efficient & sustainable?

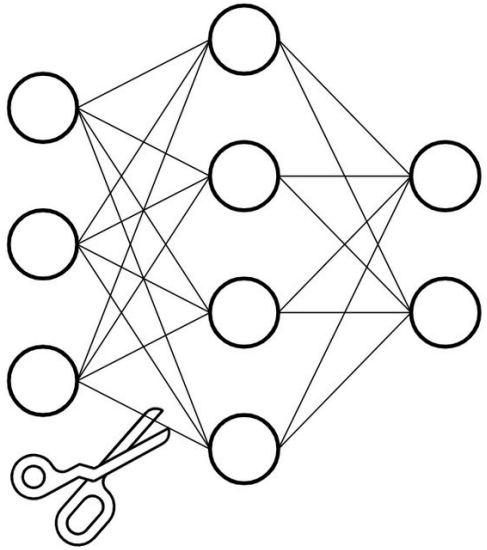
Future Directions (near-term)

- Research/production should prioritize computationally efficient networks
  - There's already movement in this direction, e.g. MobileNet for deploying CNNs on low-power mobile devices

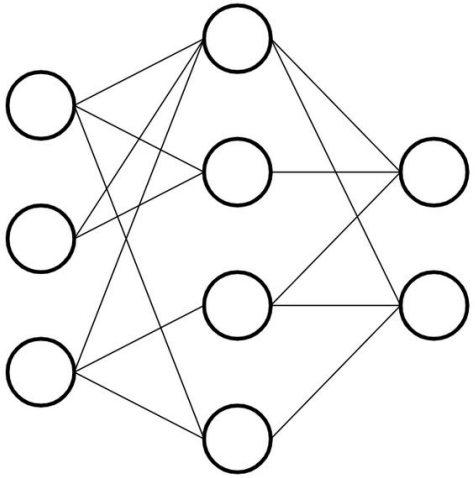
- Researchers should report training time, training hardware used, ***and hyperparameter sensitivity***
  - Gives others a sense of costs and benefits of training a network
  - Reporting time & hardware are already standard practice; hyperparameter sensitivity less so...

Future Directions (mid-term)

- Network pruning
  - Random weight initialization on large number of connections — only some weights are going to be meaningful
  - After training a network, most connections have weightings of approx 0 and only ~10-20% of connections have a meaningful weighting
  - Thus, we can drop (or ***prune***) 80-90% of network connections and maintain high network performance
  - This process radically increases network speed (and decreases power consumption) in production



Before pruning



After pruning

https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html

Future Directions (mid-term)

- Network pruning (continued)
  - What if we take the pruned network (i.e. the one with 80-90% of connections dropped), reset its weights to their original initialization values, and try to train the network again?

Do you think this will work?

Future Directions (mid-term)

- Network pruning (continued)
  - What if we take the pruned network (i.e. the one with 80-90% of connections dropped), reset its weights to their original initialization values, and try to train the network again?
  - Conventional wisdom says: “that network, because it’s smaller, won’t learn as well—you ***need*** the extra connections to allow the network to find a good local optimum”
  - What actually happens: the network trains as well, or sometimes even ***better***, than the full network!

Future Directions (mid-term)

- Network pruning (continued)
  - What's going on??
  - The current working hypothesis: every big network contains within it some smaller sub-network(s) that, when *combined with the right weight initialization*, performs as well or better than the big network.
  - Finding one of these sub-networks has been compared to "finding the winning lottery ticket"...
  - ...so this is known as the ***Lottery Ticket Hypothesis***
  - It's an open area of research

Future Directions (mid-term)

- Network pruning (continued)
  - However, there is a bias-complexity tradeoff…
  - While this prevents the model from overfitting and ensures the model is more generalizable for future unseen data,
  - the model becomes smoother and may become more susceptible to underfitting
  - May amplify already-existing biases in deep learning networks

Future Directions (long-term)

- More efficient physical substrates for neural networks
  - The massive parallelism of GPUs has proven to be useful for training deep networks, but GPUs are also power hungry.
  - Are there other physical computing devices that might also be a good fit for the kinds of computations that deep nets perform?

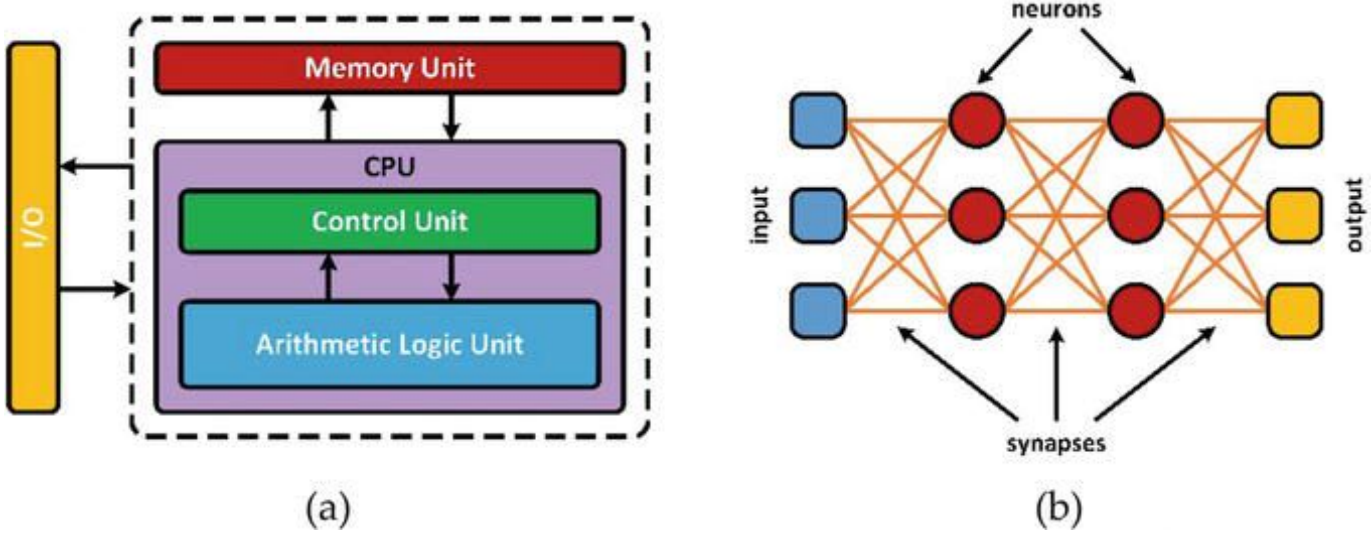
Future Directions (long-term)

- Neuromorphic computation
  - “neuro-”: brain- or neuron-like
  - “-morphic”: having the shape or structure of

- What if we could make a piece of hardware (i.e. a chip) that interfaces with a computer and creates a *physical* neural network?
  - i.e. instead of simulating ‘neurons’ with digital computers, simulate them with analog circuits
  - Potentially 1000s of times more energy- and space-efficient than GPUs
  - A specific instance of an ***ASIC*** (“application-specific integrated circuit”)
    - Other domains where this has been successful: cameras have specialized processing chips called ISPs (“image signal processors”)

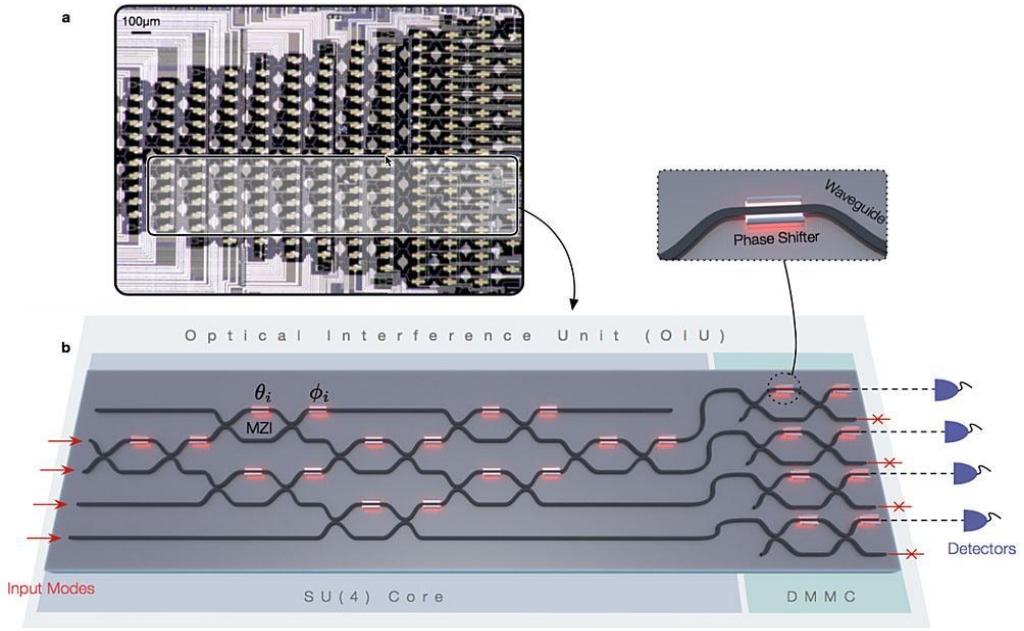
Future Directions (long-term)

- Physical connection ≅ connection between two layers in a network
- When electricity flows through, the connection is reinforced
  - Allows training / updating weights directly in hardware
- Proposed designs use ***memristors*** (memory + resistors) to implement connections



Future Directions (long-term)

- Optical neural networks
  - What if we replaced the electrons flowing through our neuromorphic chips with photons?
  - Tiny amount of energy — measured in attojoules (millionth of a trillionth of a joule, 10^{-18} joules)



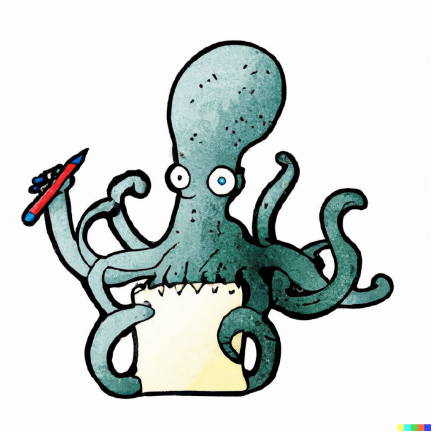
Any questions?

Future Directions (long-term)

- Neuromorphic computing is also an open area of research
- Optical neural networks are also an open area of research

Recap

Scaling deep learning systems



Sustainable deep learning

- Scaling across processors
- Scaling across GPUs
- Scaling across machines

- Near-term solutions
- Mid-term solutions (Network pruning)
- Long-term solutions (Hardware upgrade)