Machine Learning Modeling Pipelines in Production

In the third course of Machine Learning Engineering for Production Specialization, you will build models for different serving environments; implement tools and techniques to effectively manage your modeling resources and best serve offline and online inference requests; and use analytics tools and performance metrics to address model fairness, explainability issues, and mitigate bottlenecks.

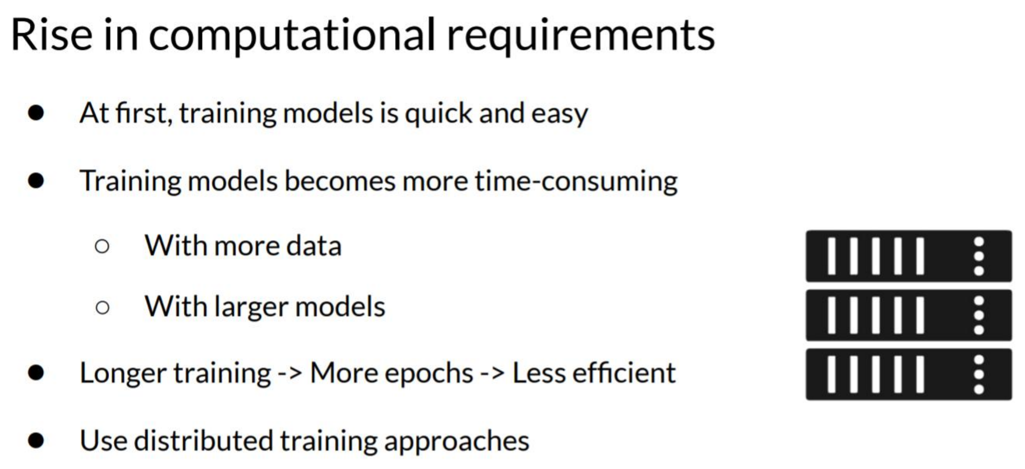
Understanding machine learning and deep learning concepts is essential, but if you're looking to build an effective AI career, you need production engineering capabilities as well. Machine learning engineering for production combines the foundational concepts of machine learning with the functional expertise of modern software development and engineering roles to help you develop production-ready skills.

Week 3: High-Performance Modeling

Contents

Compiled by **Kenneth Leung** Version: **202201**

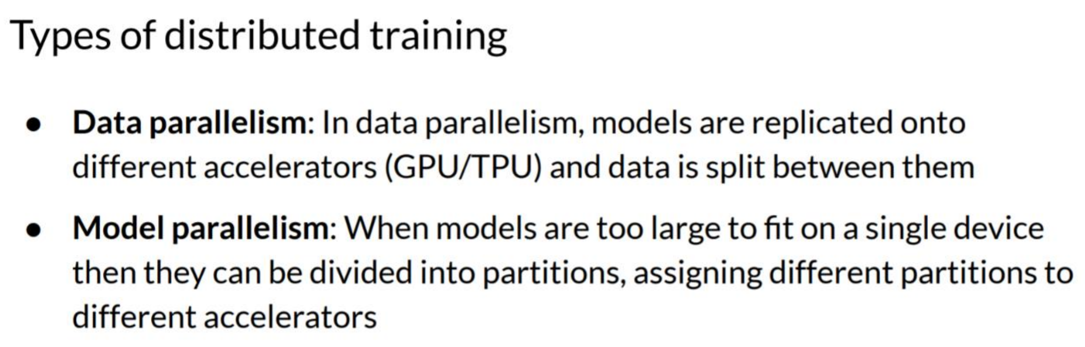
Distributed Training



Rise in computational requirements

- At first, training models is quick and easy
- Training models becomes more time-consuming
  - With more data
  - With larger models
- Longer training -> More epochs -> Less efficient
- Use distributed training approaches

- We're going to start with the discussion of distributed training, including a couple of different kinds of parallelism. Then we'll turn to high performance modeling, including high performance ingestion.
- Then we're going to turn to one of my favorite topics, knowledge distillation.
- Let's get started. We'll start with the discussion of distributed training.
- When you start prototyping, training your model might be a fast and simple task.
- However, fully training a model can become a very time-consuming task. Datasets in many domains are getting larger and larger.
- As the size of the datasets increase, models take longer and longer to train.
- The same logic applies for larger architectures, and it's not just training time. The number of epochs for a model also increases as a result.
- Solving this kind of problem usually requires **distributed training**.
- Distributed training allows us to train huge models, and at the same time, speed up training.
- In this lesson, you'll see how models can be accelerated using data and model parallelism.
- Also, you'll look at high performance modeling techniques like distribution strategies and high performance ingestion pipelines such as TF data.

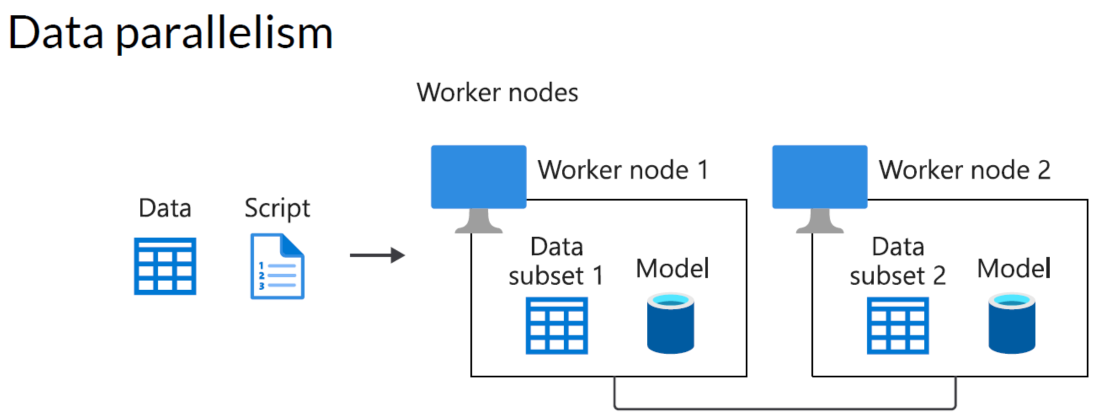


Types of distributed training

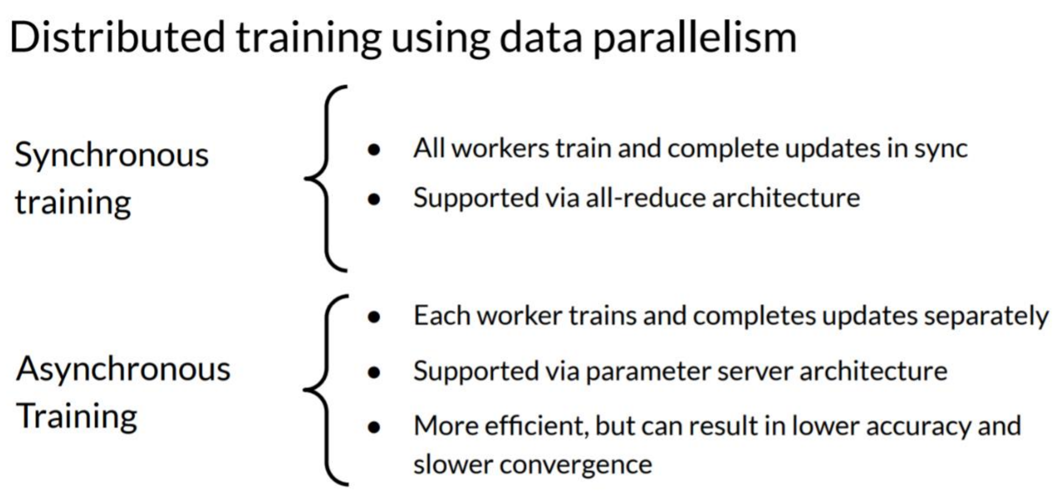
- **Data parallelism**: In data parallelism, models are replicated onto different accelerators (GPU/TPU) and data is split between them
- **Model parallelism**: When models are too large to fit on a single device then they can be divided into partitions, assigning different partitions to different accelerators

- Now, let's look into basic ways to perform distributed training. These include **data parallelism** and **model parallelism.**
- Data parallelism is probably the easier of the two to implement.

- In this approach, you divide the data into partitions. You copy the complete model to all of the workers, where each one operates on a different partition of the data and the model updates are synchronized across workers.
- This type of parallelism is **model agnostic** and can be applied to any neural network architecture.
- Usually, the scale of data parallelism corresponds to the batch size.
- In model parallelism, however, you segment the model into different parts, training concurrently on different workers.
- Each model will train on the same piece of data. Here workers only need to **synchronize the shared parameters**, usually once for each forward or back propagation step.
- You generally use model parallelism when you have a larger model which won't fit into memory on your accelerators.
- Implementation of model parallelism is relatively advanced compared to data parallelism.
- Model parallelism is a more complex topic. So for now, let's look at how data parallelism works with an illustration and then revisit model parallelism.



- In data parallelism, the data is split into partitions.
- The number of partitions is usually the total number of available workers in the compute cluster.
- To train, you copy the model onto each of these worker nodes, with each worker training on its own subset of the data.
- This requires each worker to have enough memory to load the entire model, which for larger models can be a problem.
- Each worker independently computes the errors between its predictions for its training samples and the labeled data.
- Then each worker performs back-propagation to update its model based on the errors, and communicates all of its changes to the other workers so that they can update their models.
- This means that the workers need to **synchronize their gradients at the end of each batch** to ensure that they are training a consistent model.

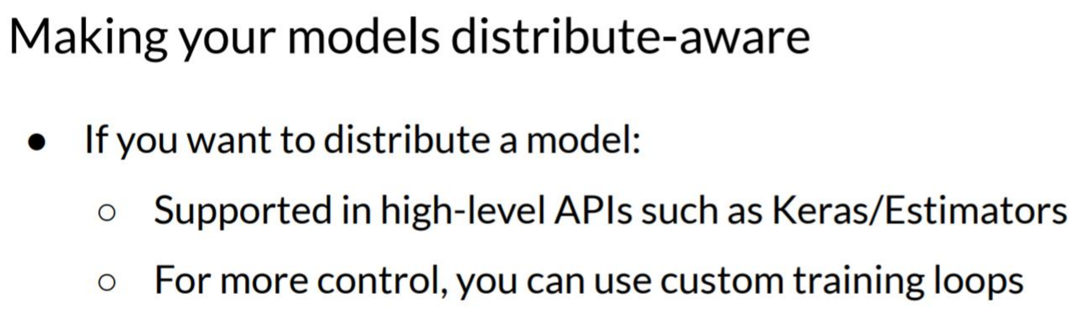


Distributed training using data parallelism

Synchronous training
- All workers train and complete updates in sync
- Supported via all-reduce architecture

Asynchronous Training
- Each worker trains and completes updates separately
- Supported via parameter server architecture
- More efficient, but can result in lower accuracy and slower convergence

- There are two basic styles of distributed training using data parallelism.
- In synchronous training, each worker trains on its current mini batch of data, applies its own updates, communicates out its updates to the other workers, and waits to receive and apply all of the updates from the other workers before proceeding to the next mini batch.
- And all-reduce algorithm is an example of this.
- In asynchronous training, all workers are independently training over their mini batch of data and updating variables asynchronously.
- Asynchronous training tends to be more efficient, but can be more difficult to implement.
- A parameter server algorithm is an example of this.
- One major disadvantage of asynchronous training is reduced accuracy and slower convergence, which means that more steps are required to converge.
- Slow convergence may not be a problem, since the speed up in asynchronous training may be enough to compensate.
- However, the accuracy loss may be an issue depending on how much accuracy is lost and the requirements of the application.



Making your models distribute-aware

- If you want to distribute a model:
  - Supported in high-level APIs such as Keras/Estimators
  - For more control, you can use custom training loops

- To use distributed training, it's important that models become distribute-aware.
- Fortunately, high-level APIs like Keras or Estimators support distributed training.
- You can even create custom training loops to provide more precise control.
- You'll see that to make these ordinary models capable of performing training or inference in a distributed manner, you need to make them distribute-aware with some small changes of code.
- By doing this, you get the power of training your model on a large number of accelerators, like GPUs or TPUs.

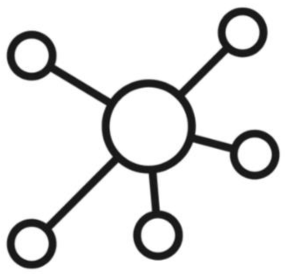
tf.distribute.Strategy

- Library in TensorFlow for running a computation in multiple devices
- Supports distribution strategies for high-level APIs like Keras and custom training loops
- Convenient to use with little or no code changes

- In particular, to perform distributed training in TensorFlow, you can make use of TensorFlow's distribute.Strategy class library.
- This class supports several distribution strategies for high-level APIs and also supports training using a custom training loop.
- The tf.distribute.Strategy class supports the execution of TensorFlow code not only in eager mode, but also in graph mode, as well as using tf function.
- In addition to training models, it's also possible to use this API to perform model evaluation and prediction in a distributed manner on different platforms.
- It requires a minimal amount of extra code to adapt your models for distributed training.
- You can easily switch between different strategies to experiment and find the ones that best fit your needs.

Distribution Strategies supported by tf.distribute.Strategy

- **One Device Strategy**
- **Mirrored Strategy**
- **Parameter Server Strategy**
- Multi-Worker Mirrored Strategy
- Central Storage Strategy
- TPU Strategy

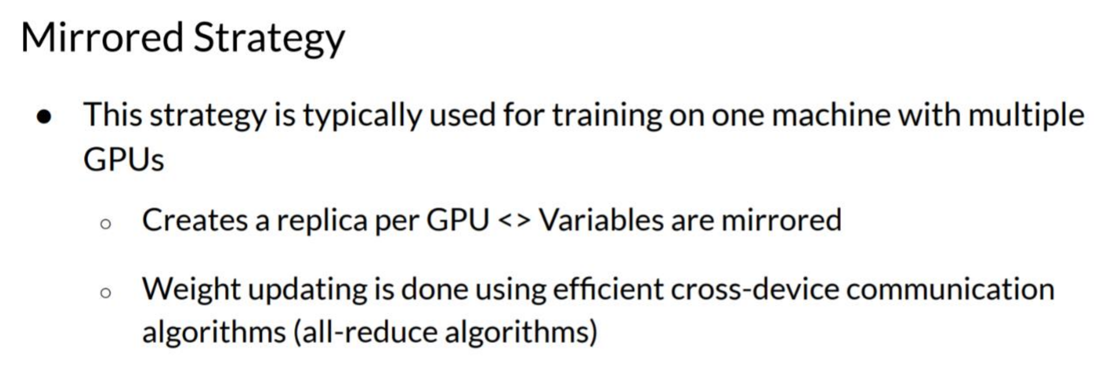


- There are many **different strategies for performing distributed training** with TensorFlow.
- The following ones are the most used: one device strategy, mirrored strategy, parameter service strategy, multi-worker mirrored strategy, central storage strategy, and TPU strategy.
- Let's focus on the first three options.

One Device Strategy

- Single device - no distribution
- Typical usage of this strategy is testing your code before switching to other strategies that actually distribute your code

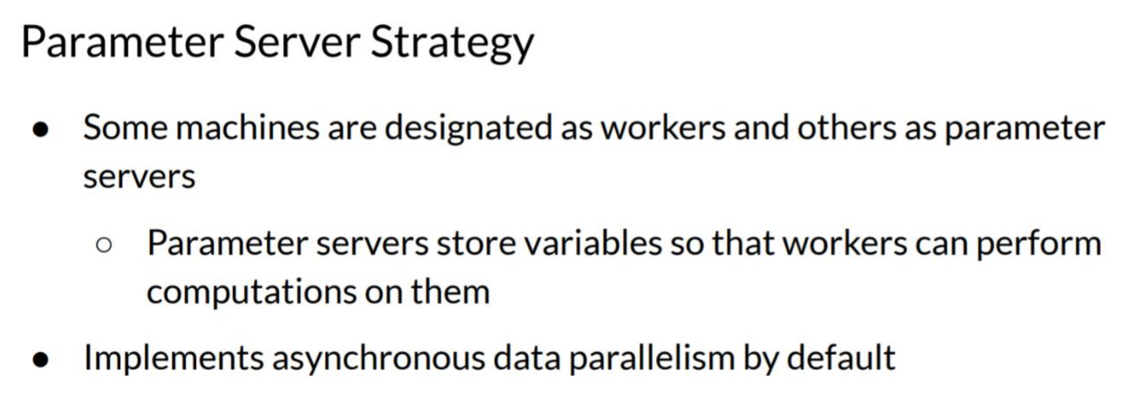
- A one-device strategy will place any variables created in its scope on the specified device.
- The term device is very commonly used to refer to a CPU or an accelerator like a GPU or TPU on any physical machine which runs machine learning models during different stages of its life cycle.
- Input distributed through this strategy will be pre-fetched to the specified device.
- Moreover, any functions called via strategy.run will also be placed on the specified device as well.
- Typical usage of this strategy could be testing your code with tf.distribute.Strategy API before switching to other strategies which actually distribute to multiple devices and machines.
- So this is typically something you'd use in development.



Mirrored Strategy

- This strategy is typically used for training on one machine with multiple GPUs
  - Creates a replica per GPU <> Variables are mirrored
  - Weight updating is done using efficient cross-device communication algorithms (all-reduce algorithms)

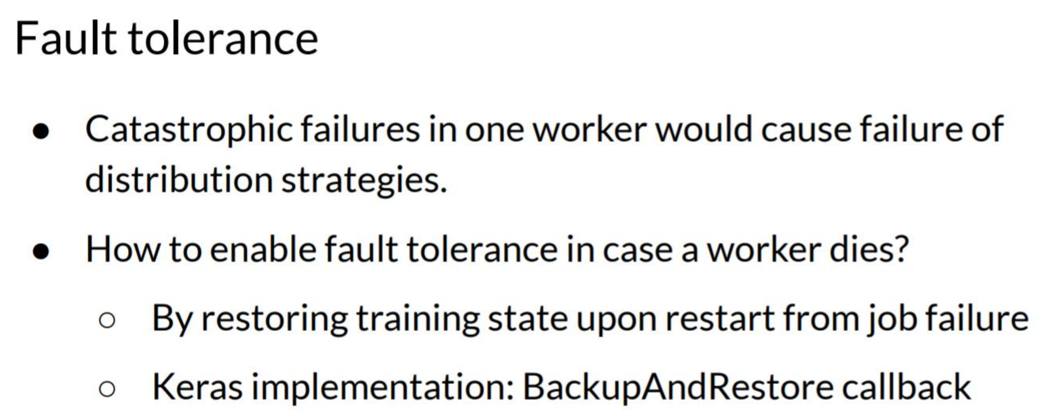
- Mirrored strategy supports **synchronous** distributed training on multiple GPUs on one machine.
- It creates one replica per GPU device. Each variable in the model is mirrored across all the other replicas
- Together these variables form a single conceptual variable called a mirrored variable i.e. when you copy the same variables in the model to multiple devices, they are called mirrored variables.
- These variables are kept in sync with each other by applying identical updates.
- Efficient all-reduce algorithms are used to communicate the variable updates across the devices.
- All-reduce aggregates tensors across all the devices by adding them up and makes them available on each device.
- It's a fused algorithm that is very efficient and can reduce the overhead of synchronization significantly.



Parameter Server Strategy

- Some machines are designated as workers and others as parameter servers
  - Parameter servers store variables so that workers can perform computations on them
- Implements asynchronous data parallelism by default

- The parameter service strategy is a common **asynchronous** data parallel method to scale up model training on multiple machines.
- A parameter server training cluster consists of workers and parameter servers.
- Variables are created on parameter servers, and they are read and updated by workers in each step.
- By default, workers read and update these variables independently without synchronizing with each other.
- This is why sometimes parameter server style training is also referred to as asynchronous training.

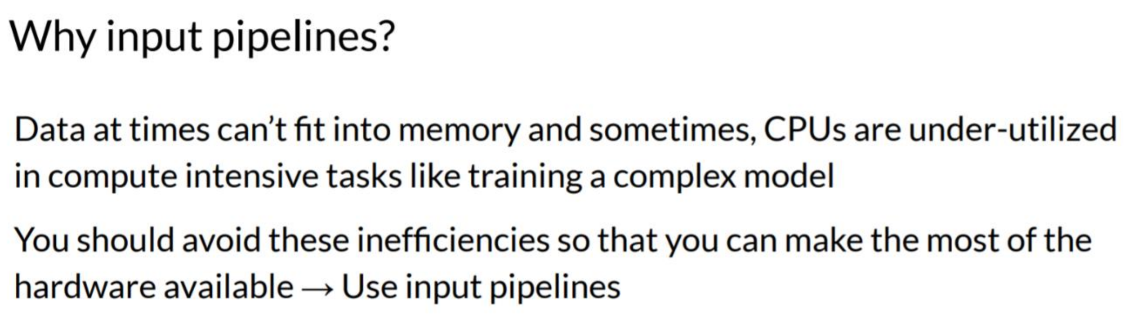


Fault tolerance

- Catastrophic failures in one worker would cause failure of distribution strategies.
- How to enable fault tolerance in case a worker dies?
    - By restoring training state upon restart from job failure
    - Keras implementation: BackupAndRestore callback

- Typically in synchronous training, the entire cluster of workers would fail if one or more of the workers were to fail.
- It's important to consider some form of fault tolerance in cases where workers die or become unstable.
- This allows you to recover from a failure incurred by preempting workers.
- This can be done by **preserving the training state** in the distributed file system.
- Since all the workers are kept in sync in terms of training epochs and steps, other workers would need to wait for the failed or preempted worker to restart in order to continue.
- In the multi-worker mirrored strategy, for example, if a worker gets interrupted, the whole cluster pauses until the interrupted worker is restarted.
- Other workers will also restart, and the interrupted worker rejoins the cluster.
- Then there's needs to be a way so that every worker picks up its former state, thereby allowing the cluster to get back in sync to allow for training to proceed smoothly.
- For example, Keras provides this functionality in the BackupAndRestore callback.

High-Performance Ingestion



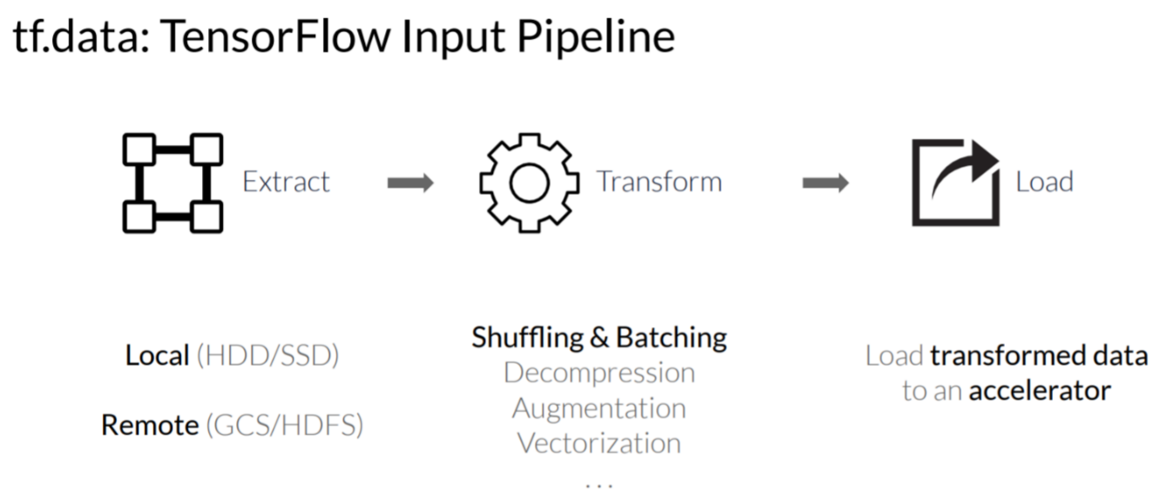
Why input pipelines?

Data at times can't fit into memory and sometimes, CPUs are under-utilized in compute intensive tasks like training a complex model

You should avoid these inefficiencies so that you can make the most of the hardware available → Use input pipelines

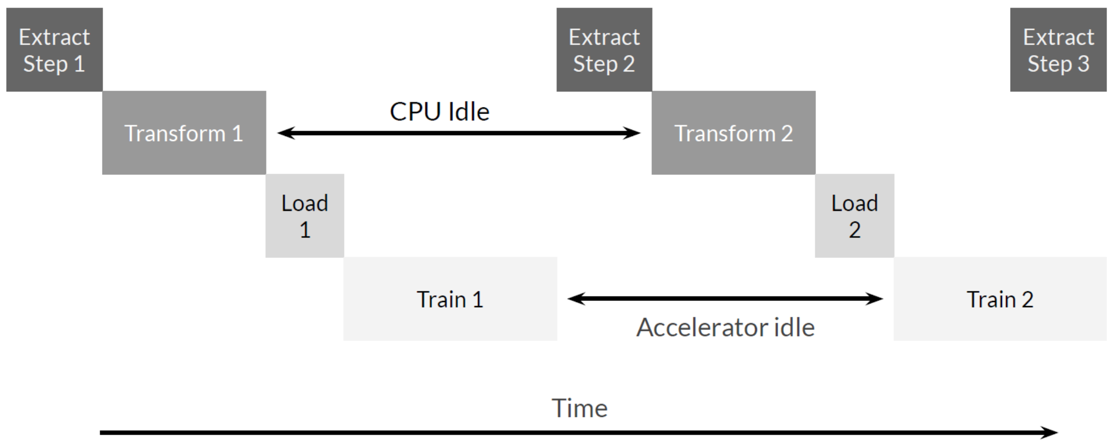
- Accelerators are a key part of high-performance modeling, training, and inference, but accelerators are also expensive, so it's important to use them efficiently.
- That means keeping them busy, which requires you to **supply them with enough data fast enough**.
- That's why high-performance ingestion is important in high-performance modeling. Let's discuss that now.
- Let's discuss why input pipelines are often needed for training models and the issues around applying transformations on data.
- Generally, transformations deal with pre-processing tasks that tend to add overhead to your training input pipeline.
- For example, when augmenting huge image classification datasets, many of these transformations are often applied element wise.

- As a result, if it takes too long to apply transformations, this can lead to the CPU being underutilized while it waits for data
- This is not just limited to the transformations. At times, you could have data that simply cannot fit into memory, and it often becomes a challenge to build a scalable input data pipeline that will feed your data fast enough to keep your training processors busy.
- In the end, the goal should be to efficiently utilize the hardware available and reduce the time required to load the data from the disk, and also reduce the time required for pre-processing it.
- Input pipelines are important part of many training pipelines, but there are often similar requirements for inference pipelines as well.
- In the larger context of a training pipeline such as a TFX training pipeline, a high-performance input pipeline would be part of the trainer component and possibly other components like Transform that may need to do quite a bit of work on the data.



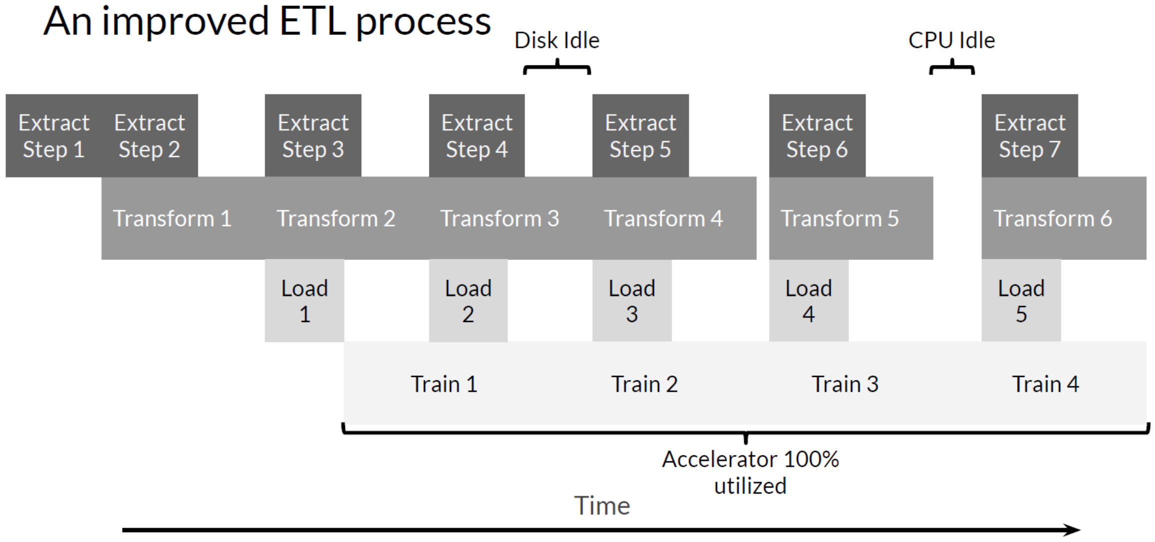
- There are many ways to design an efficient input pipeline.
- One framework that can help is TensorFlow data or tf.data. Let's consider tf.data as an example of how to design an efficient input pipeline.
- You can view input pipelines as an ETL process, providing a framework to facilitate applying performance optimization.
- The 1st step involves extracting data from data stores that may be either local or remote, like hard drives, SSDs, cloud storage, and HDFS.
- In the 2nd step, data often needs to be pre-processed. This includes shuffling, batching, and repeating data.
- The way you order these transformations may have an impact on your pipelines performance.
- This is something that you need to be aware of when using any data transformation like map or batch or shuffle or repeat, and so forth.
- You then load the pre-processed data into the model, which may be training on a GPU or TPU and start training.
- A key requirement for high-performance input Pipelines is the parallel processing of data across the various systems to try to make maximum efficient use of the available compute, IO, and network resources.
- Especially for more expensive components such as accelerators, you want to keep them busy as much as possible.

Inefficient ETL process



- Let's look at a typical pattern that is easy to fall into and one that you really want to avoid.
- In this scenario, key hardware components, including CPUs and accelerators sit idle, waiting for the previous steps to complete.
- If you think about it, ETL (extract, transform, and load) is a good mental model for data performance.
- Now, to give you some intuition on how pipelining can be carried out, each of the different phases of ETL use different hardware components in your system.
- Your extract phase is exercising your disk, or your network for your loading from a remote system.
- **Transform typically happens on the CPU** and can be very CPU hungry.
- The load phase is exercising the DMA, or direct memory access subsystem and the connections to your accelerator, probably a GPU or TPU.

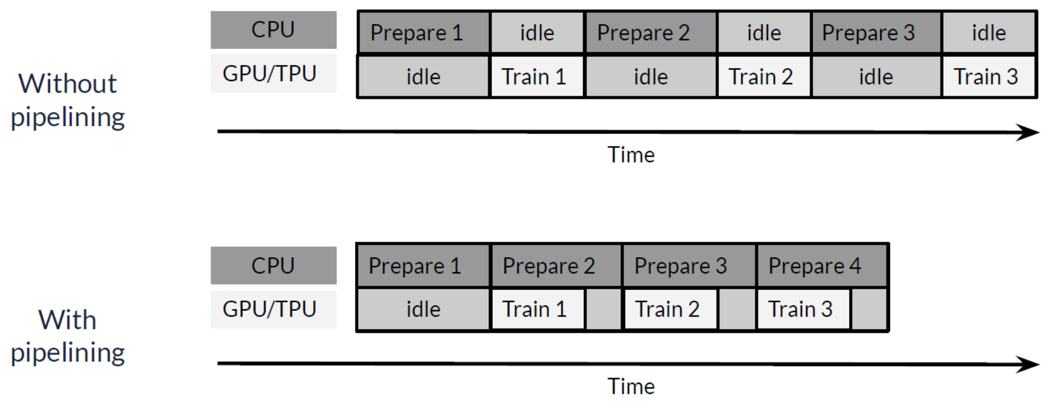
An improved ETL process



- Now, this is a much more efficient pattern than the previous one, although it's still not quite optimal.
- In practice, this pattern may be difficult to optimize further in many cases.
- As this diagram shows by parallelizing operations, you can overlap the different parts of ETL using a technique known as **software pipelining**.
- With software pipelining, you're extracting data for step 5, while you're transforming for step 4, and while you're loading data for step 3, and finally, your training for step 2 all at the same time.

- This results in a very efficient use of your compute resources.
- As a result, your training is much faster and your resource utilization is much higher.
- Notice that now there are only a few instances where your hard drive and CPU are actually sitting idle.

Pipelining



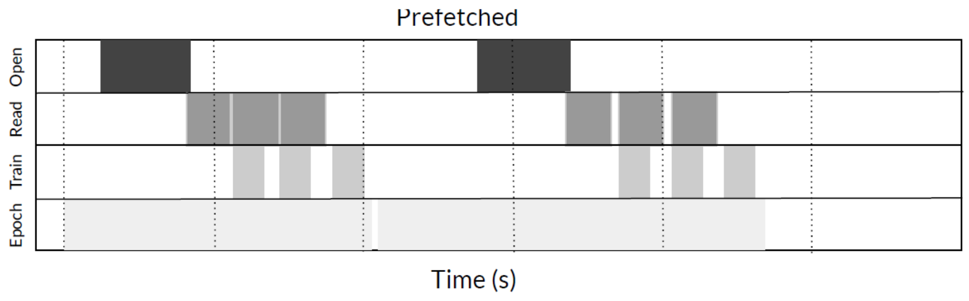
- Through pipelining your training procedure, you can overcome CPU bottlenecks by overlapping the CPU pre-processing and model execution of accelerators.
- While the accelerator is busy with training the model in the background, the CPU starts preparing data for the next training step.
- You can see there's a significant improvement in model training time when using pipelining.
- While you can still expect some idle time, it's greatly reduced by using pipelining.

How to optimize pipeline performance?

- Prefetching
- Parallelize data extraction and transformation
- Caching
- Reduce memory

- How do you optimize your data pipeline in practice? There are a few basic approaches that could potentially be used to accelerate your pipeline.
- Prefetching, for example, where you begin **loading data for the next step before the current step completes**.
- Other techniques involve parallelizing data extraction and transformation.
- Caching the dataset to get started with training immediately once a new epoch begins, is also very effective when you have enough cache.
- Finally, you need to be aware of how you order these optimizations in your pipeline, to maximize the pipelines efficiency.

Optimize with prefetching



```
benchmark(
    ArtificialDataset()
    .prefetch(tf.data.experimental.AUTOTUNE)
)
```

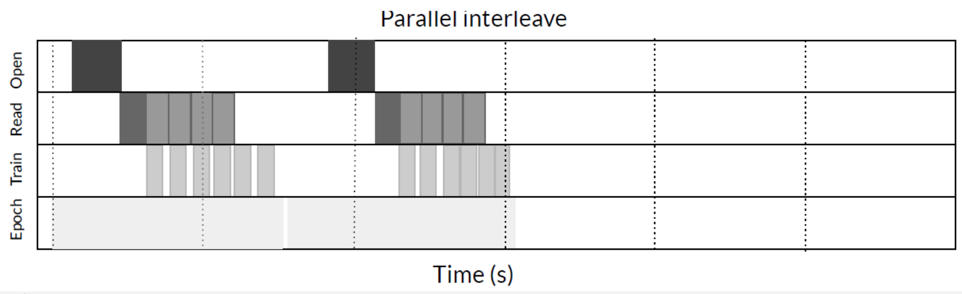
- With prefetching, you overlap the work of a producer with a work of a consumer, while the model is executing step S, the input pipeline is reading the data for step S plus 1.
- This reduces the total time it takes for a step to either train the model or extract data from disk, whichever is highest.
- The tf.data API provides the tf.dataset **prefetch** transformation.
- You can use this to decouple the time when data is produced, from the time when the data is consumed.
- This transformation uses a background thread and an internal buffer to prefetch elements from the input dataset ahead of time before they are requested.
- Ideally, the number of elements to prefetch should be equal to, or possibly greater than, the number of batches consumed by a single training step.
- You can either manually tune this value, or set it to tf data experimental **autotune**, as in this example, which will configure the tf.data runtime, to optimize the value dynamically at runtime.

Parallelize data extraction

- Time-to-first-byte: Prefer local storage as it takes significantly longer to read data from remote storage
- Read throughput: Maximize the aggregate bandwidth of remote storage by reading more files

- In a real-world setting, the input data may be stored remotely, for example, on GCS or HDFS.
- A dataset pipeline that works well when reading data locally, might become bottlenecked on IO when reading data remotely, because of the following differences between local and remote storage.
- Time=to-first-byte. Reading the first byte of a file from remote storage, can take orders of magnitude longer, than from local storage.
- Read throughput. Again, while remote storage typically offers large aggregate bandwidth, reading a single file might only be able to utilize a small fraction of this bandwidth.

Parallel interleave



```
benchmark(
    tf.data.Dataset.range(2)
    .interleave(
        ArtificialDataset,
        num_parallel_calls=tf.data.experimental.AUTOTUNE
    )
)
```

- To reduce data extraction overhead, the tf.data dataset **interleave** transformation is used to parallelize the data loading step, including interleaving contents of other datasets.
- The number of datasets to overlap as specified by the cycle length argument, while the level of parallelism is set with the num_parallel_calls argument.
- Similar to the prefetch transformation, the interleaved transformation supports tf data experimental auto-tune, which will delegate the decision about what level of parallelism to use for the tf data runtime.

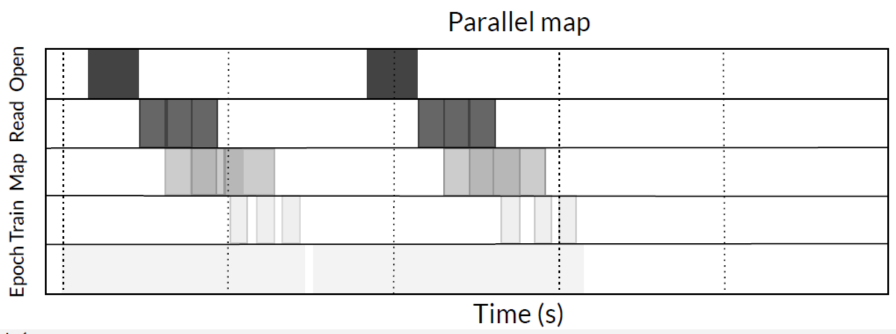
Parallelize data transformation

- Post data loading, the inputs may need preprocessing
- Element-wise preprocessing can be parallelized across CPU cores
- The optimal value for the level of parallelism depends on:
  - Size and shape of training data
  - Cost of the mapping transformation
  - Load the CPU is experiencing currently
- With tf.data you can use AUTOTUNE to set parallelism automatically

- When preparing data, input elements may need to be preprocessed.
- For example, the tf.data API, offers the tf.data dataset map transformation, which implies a user-defined function to each element of the input dataset.
- **Element-wise preprocessing can be parallelized** across **multiple** CPU cores.
- Similar to the prefetch and interleaved transformation, the map transformation provides the num_parallel_calls argument to specify the level of parallelism.
- Choosing the best value for the num_parallel_calls argument depends on your hardware, the characteristics of your training data such as size and shape, the cost of your map function, and what other processing is happening on the CPU at the same time.
- A simple heuristic is to use the number of available CPU cores.

- However, as with prefetch interleave transformations, the map transformation in tf.data supports the tf data experimental autotune parameter, which will delegate the decision about which level of parallelism to use for the tf data runtime.

Parallel mapping



```
benchmark(
    ArtificialDataset()
    .map(
        mapped_function,
        num_parallel_calls=tf.data.AUTOTUNE
    )
)
```

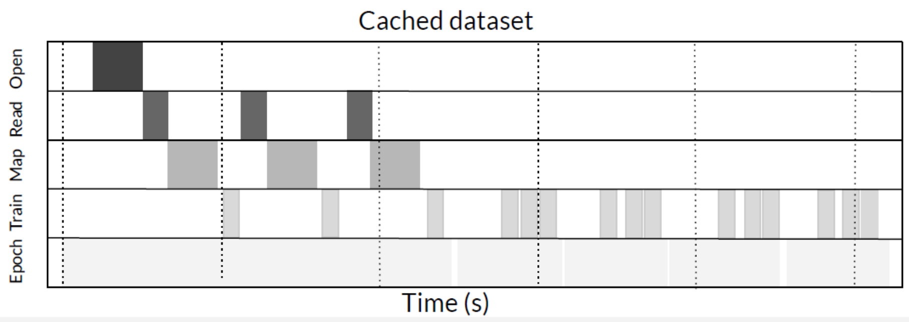
- Let's see what happens when you apply pre-processing in parallel on multiple samples.
- Here, the preprocessing steps overlap, reducing the overall time for a single iteration, this can potentially be a big win.
- Here, you implement the same preprocessing function, but apply it in parallel, on multiple samples.

Improve training time with caching

- In-memory: `tf.data.Dataset.cache()`
- Disk: `tf.data.Dataset.cache(filename=...)`

- Tf.data dataset includes the ability to cache a dataset, either in memory or on local storage.
- In many instances, caching is advantageous and leads to increased performance.
- This will **save some operations, like file opening and data reading**, from being executed during each epoch.
- When you cache a dataset, the transformations before caching, like file opening and data reading, are executed only during the first epoch,
- In the next epochs, we'll re-use the cache data.

Caching



```
benchmark(
    ArtificialDataset().map(mapped_function).cache(),5
)
```

- Let's consider two scenarios for caching.
- If the user-defined function passed into the map transformation is expensive, then it makes sense to apply the cache transformation **after** the map transformation, as long as the resulting dataset can fit into memory or local storage.
- On the contrary, if the user-defined function increases the space required to store the dataset beyond the cache capacity, either apply it after cache transformation, or consider pre-processing your data before your training job, to reduce resource requirement.

Training Large Models – The Rise of Giant Neural Nets and Parallelism

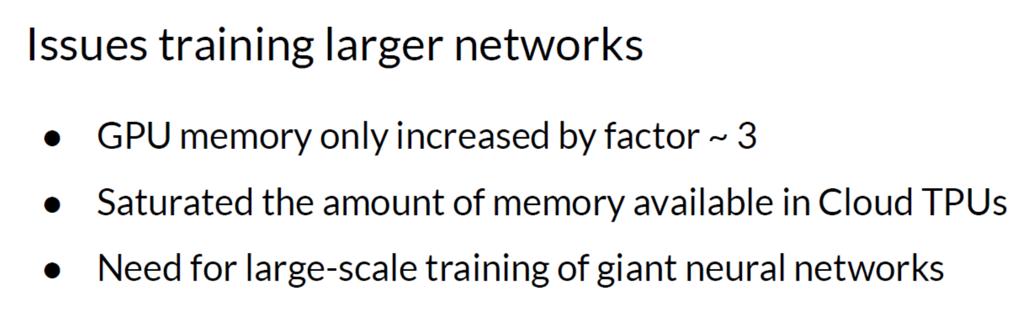
Rise of giant neural networks

- In 2014, the ImageNet winner was GoogleNet with *4 mil. parameters* and scoring a 74.8% top-1 accuracy
- In 2017, Squeeze-and-excitation networks achieved 82.7% top-1 accuracy with *145.8 mil. Parameters*

36 fold increase in the number of parameters in just 3 years!

- Now let's talk about high-performance modeling and some of the issues with increasingly large models, along with some advanced techniques for training large models.
- In recent years, the size of machine learning datasets and models has been continuously increasing, allowing for improved results on a wide range of tasks, including speech recognition, visual recognition, and language processing.
- Recent advances by BigGAN, BERT, and GPT-3 have shown that ever-larger models lead to better task performance.
- At the same time, hardware accelerators like GPUs and TPUs have also been increasing in power, but at a significantly slower pace.

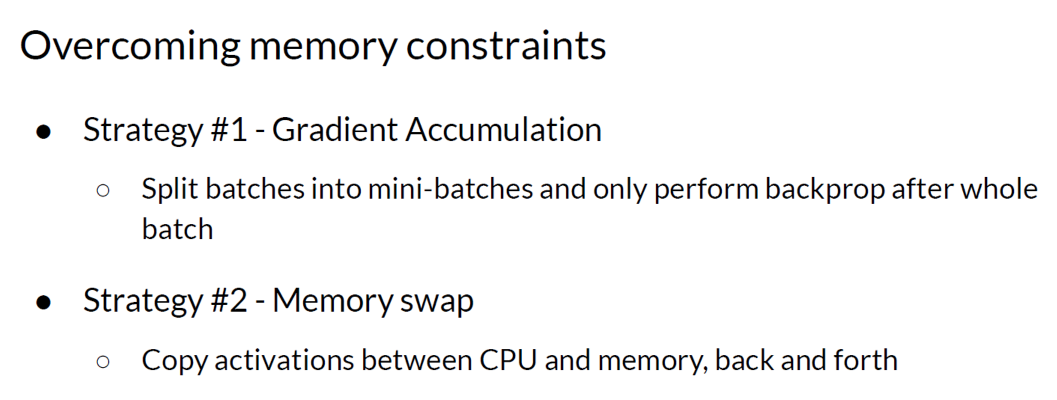
- **The gap between model growth and hardware improvement has increased the importance of parallelism.**
- Parallelism in this context means training a single machine learning model on multiple hardware devices.
- Some model architectures, especially small models, are conducive to parallelism and can be divided quite easily between hardware devices.
- In enormous models, synchronization costs lead to degraded performance, preventing them from being used.
- There's also a strong correlation between model size and classification accuracy. For example, the winner of the 2014 ImageNet Visual Recognition Challenge was GoogleNet, which achieved 74.8 top-1 accuracy with four million parameters.
- While just three years later, the winner of the 2017 ImageNet Challenge went to Squeeze-and-Excitation Networks, which gained an 82.7 top-1 accuracy with 145.8 million parameters.
- This was a 36 fold increase in the number of parameters in just three years.



Issues training larger networks

- GPU memory only increased by factor ~ 3
- Saturated the amount of memory available in Cloud TPUs
- Need for large-scale training of giant neural networks

- **Massive numbers of weights and activation parameters require massive memory storage.**
- The natural question is where to store them as the models continue to require more and more parameters?
- In the last few years, memory capacity made very modest increases compared to model parameters growth.
- More concretely, GPU memory has only increased by a factor of approximately three, and the current state-of-the-art image models have already reached the available memory found on Cloud TPU V2s.
- There is a strong and pressing need for an efficient, scalable infrastructure that enables large-scale deep learning and overcomes the **memory limitations on current accelerators**.
- But in some ways, this is not a new problem. Let's look at how some older methods approach this problem of overcoming memory constraints and see if we could apply them today. Then let's look at some newer methods and some of the advances that they've been able to make.



Overcoming memory constraints

- Strategy #1 - Gradient Accumulation
  - Split batches into mini-batches and only perform backprop after whole batch
- Strategy #2 - Memory swap
  - Copy activations between CPU and memory, back and forth

- There is a need to strategize and implement solutions to overcome these important memory constraints.
- One strategy that can overcome problems with insufficient GPU memory is **gradient accumulation**.
- Gradient accumulation is a mechanism to split full batches into several mini-batches. During backpropagation, the model isn't updated with each mini-batch, and instead, the gradients are accumulated.

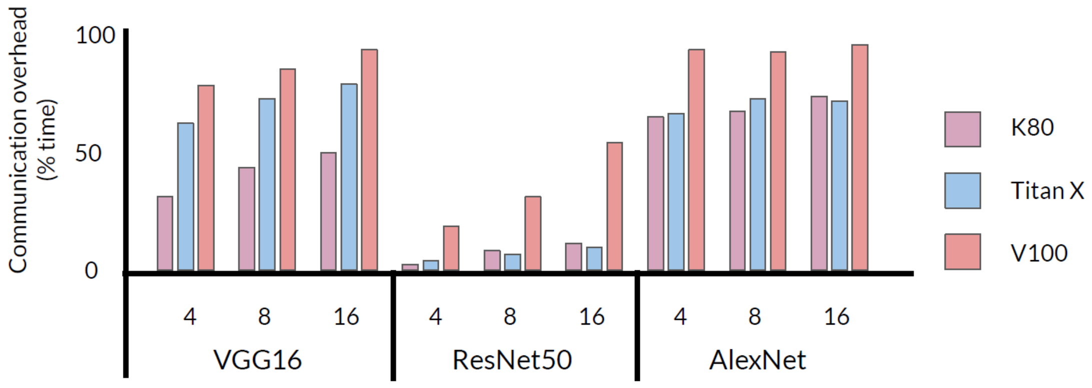
- When a full batch completes, the accumulated gradients of all the previous mini-batches are used for backprop to update the model.
- This process is as effective as using a full batch for training the network since model parameters are updated the same number of times.
- The second approach is **swapping**. Here, since there isn't enough storage on the accelerator, you copy activations back to the CPU or memory and then back to the accelerator.
- The problem here is that it's slow, and the communication between CPU or memory and the accelerator becomes the bottleneck.

Parallelism revisited

- **Data parallelism**: In data parallelism, models are replicated onto different accelerators (GPU/TPU) and data is split between them
- **Model parallelism**: When models are too large to fit on a single device then they can be divided into partitions, assigning different partitions to different accelerators

- Returning to our discussion of parallelism, the basic idea is to split the computation between multiple workers.
- You've already seen two ways to parallelize, data parallelism and model parallelism.
- Data parallelism splits the input across workers.
- Model parallelism splits the model across workers.
- In data parallelism, different workers or GPUs work on the same model but deal with different data.
- The model is replicated across a number of workers and each worker performs the forward and backward pass. When it finishes the process, it synchronizes the updated model weights with the other devices and calculates the updated weights of the entire mini-batch.
- In model parallelism, however, workers only need to synchronize the shared parameters, usually once for each forward or backprop step.
- Also, larger models aren't a major concern since each worker operates on a subsection of the model using the same training data.
- When using model parallelism in training, the model is divided across k workers with each worker holding a part of the model.
- A naïve approach to model parallelism is to **divide an n-layered neural network** into k workers by simply hosting n over k layers on each worker.
- More sophisticated methods make sure that each worker is similarly busy by analyzing the computational complexity of each layer.
- Standard model parallelism enables training of larger neural networks but suffers from a large hit in performance since workers are constantly waiting for each other and only one can perform updates at a given time.

Challenges in data parallelism



- Moving back to data parallelism, let's look at some benchmarks.
- This chart shows communication overhead as a percentage of total training time for three different hardware configurations.
- Many models like AlexNet and VGG16 have a high communication overhead even on the relatively slow K80 accelerator.
- Two factors contribute to an increase in the communication overhead across all models: an increase in the number of data-parallel workers and an increase in GPU compute capacity.
- With data parallelism, the input dataset is partitioned across multiple GPUs.
- Each GPU maintains a full copy of the model and trains on its own partition of data while periodically synchronizing weights with other GPUs, using either collective communication primitives or parameter servers.
- The frequency of parameter synchronization affects both statistical and hardware efficiency.
- Synchronizing at the end of every mini-batch reduces the staleness of weights used to compute gradients, ensuring good statistical efficiency.
- Unfortunately, this requires each GPU to wait for gradients from other GPUs, which significantly lowers hardware efficiency.
- Communication stalls are inevitable in data-parallel training due to the structure of neural networks, and the result is that communication can often dominate total execution time.
- Rapid increases in accelerator speeds further shift the training bottleneck towards communication.

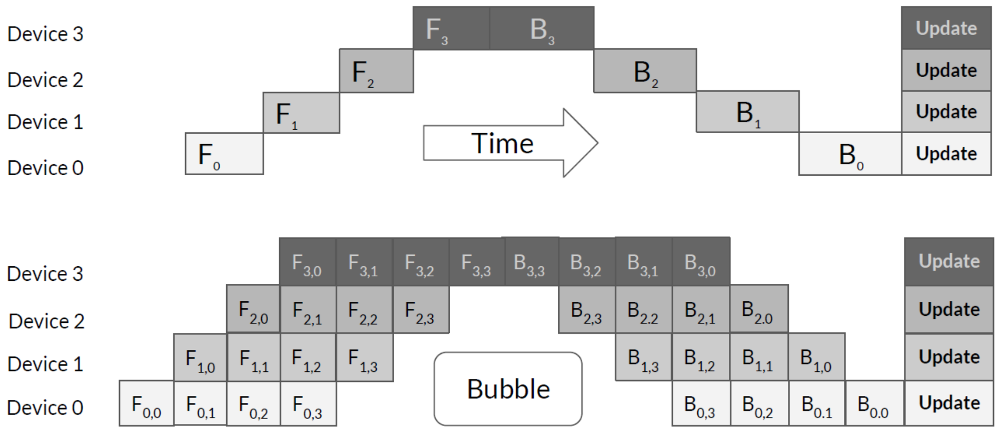
Challenges keeping accelerators busy

- Accelerators have limited memory
- Model parallelism: large networks can be trained
  - But, accelerator compute capacity is underutilized
- Data parallelism: train same model with different input data
  - But, the maximum model size an accelerator can support is limited

- Then there's another problem. Accelerators.
- They have limited memory and limited communication bandwidth on the host machine.

- That means that model parallelism is needed for training bigger models on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.
- But due to the **sequential nature of neural networks**, this naïve strategy may result in only one accelerator being active during computation, significantly underutilizing accelerator compute capacity.
- On the other hand, a standard data parallelism approach allows concurrent training of the same model with different input data on multiple accelerators, but cannot increase the maximum model size an accelerator can support.

Pipeline parallelism


Device 3
Device 2
Device 1
Device 0
F_3
B_3
F_2
B_2
F_1
B_1
F_0
B_0
Time
Update
Update
Update
Update
Device 3
Device 2
Device 1
Device 0
$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$ $B_{2,3}$ $B_{2,2}$ $B_{2,1}$ $B_{2,0}$
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ $B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$
Bubble
Update
Update
Update
Update

In this diagram, F refers to forward pass while B refers to backward pass distributed among the accelerators

- The issues with data and model parallelism have led to the development of **pipeline parallelism**.
- In this diagram. A naive model parallelism strategy (diagram at the top) leads to severe underutilization due to the sequential nature of the model, only one accelerator is active at a time.
- To enable efficient training across multiple accelerators, you need to find a way so that you can partition a model across different accelerators and automatically split a mini-batch of training examples into smaller **micro**-batches.
- By pipelining any execution across micro-batches, accelerators can operate in parallel.
- In addition, gradients are consistently accumulated across micro-batches so that the number of partitions does not affect model quality.
- Google's **GPipe** is an open source library for efficiently training large-scale models using pipeline parallelism.
- In the diagram at the bottom, GPipe divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on separate micro-batches at the same time.
- GPipe essentially presents a new way to approach model parallelism, which allows training of large models on multiple hardware devices with an almost one-to-one improvement in performance.
- It also helps models include significantly more parameters, allowing for better results in training.
- There's also PipeDream from Microsoft, which also supports pipeline parallelism.
- GPipe and PipeDream are similar in many ways.

Pipeline parallelism

- Integrates both data and model parallelism:
    - Divide mini-batch data into micro-batches
    - Different workers work on different micro-batches in parallel
    - Allow ML models to have significantly more parameters

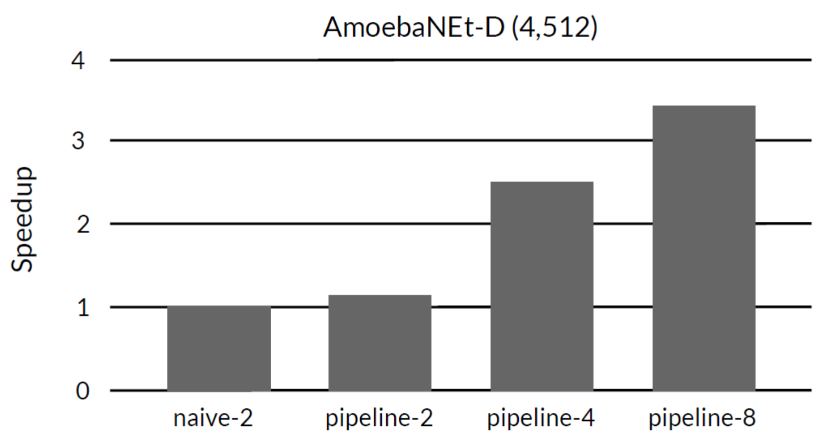
- Pipeline parallelism frameworks such as GPipe and PipeDream **integrate both data and model parallelism** to achieve high efficiency and preserve model accuracy.
- They do that by **dividing mini-batches into smaller micro-batches**, by allowing different workers to work on different micro-batches in parallel.
- As a result, they can train models with significantly more parameters.

GPipe - Key features

- Open-source TensorFlow library (using Lingvo)
- Inserts communication primitives at the partition boundaries
- Automatic parallelism to reduce memory consumption
- Gradient accumulation across micro-batches, so that model quality is preserved
- Partitioning is heuristic-based

- GPipe is an open source distributed machine learning library that uses synchronous mini-batch gradient descent for training.
- It partitions a model across different accelerators and automatically splits a mini-batch of training examples into micro-batches.
- By pipelining the execution across micro-batches, accelerators can train in parallel.
- GPipe receives as input an architecture of a neural network, a mini-batch size and the number of hardware devices that will be available for the calculation.
- It then automatically divides the network layers into stages and the mini-batches into micro-batches, spreading them across the devices.
- To divide the model into key stages, GPipe estimates the cost of each layer given its activation function and the content of the training data.

GPipe Results



- GPipe attempts to maximize memory allocation for model parameters.
- The research team at Google ran experiments on Cloud TPU v2s, each of which has eight accelerator cores and 64 gigabytes of memory; eight gigabytes per accelerator.
- Without GPipe, a single accelerator can only train up to 82 million model parameters due to memory constraints.
- Thanks to re-computation in back propagation and batch splitting, GPipe reduce intermediate activation memory from 6.2 gigabytes to 3.4 gigabytes, enabling a single accelerator to train up to 318 million parameters.
- They also saw that with pipeline parallelism, the maximum model size was proportional to the number of partitions as expected.
- With GPipe, testing with an AmoebaNet model was able to incorporate 1.8 billion parameters on the eight accelerators of the Cloud TPU, 25 times more than is possible without GPipe.
- To test efficiency, they measured the effects of GPipe on the throughput of an AmoebaNet D model.
- Since training required at least two accelerators to fit the model size, they measured the speed up with respect to the naïve case with two partitions, but no pipeline parallelism.
- They observed an almost linear speedup in training.
- Compared to the naive approach with two partitions, distributing the model across four times the accelerators achieved a speedup of 3.5 times.

Teacher and Student Networks

Sophisticated models and their problems

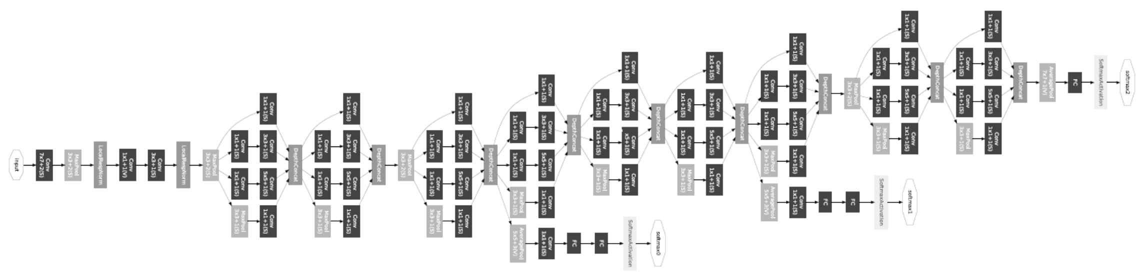
- Larger sophisticated models become complex
- Complex models learn complex tasks
- Can we express this learning more efficiently?

Is it possible to ‘distill’ or concentrate this complexity into smaller networks?

- So far, we've discussed ways to optimize the implementation of models to make them more efficient.

- But you can also try to capture or distill the knowledge that has been learned by a model, in a more of compact model by using a different style of training.
- This is known as **knowledge distillation**.
- By now you've seen a few different ways to optimize your models. But what happens if you want to deploy a model that is relatively complex and significantly large in size?
- Would merely reducing the number of parameters help with deployment?
- Or would reducing the model complexity help you deploy these larger, more powerful models? Let's try to answer those questions.
- For starters, let's look at how larger models become complex, and try to find out whether it's possible to have the same level of sophistication with smaller models.
- Models tend to become larger and more complex as they try to capture more information or knowledge in order to learn complex tasks.
- But if we can **express or represent this learning more efficiently**, we might be able to create smaller models that are equivalent to these larger, more complex models.
- As an example, let's take a look at a complex image classification model which presents challenges in deployment.

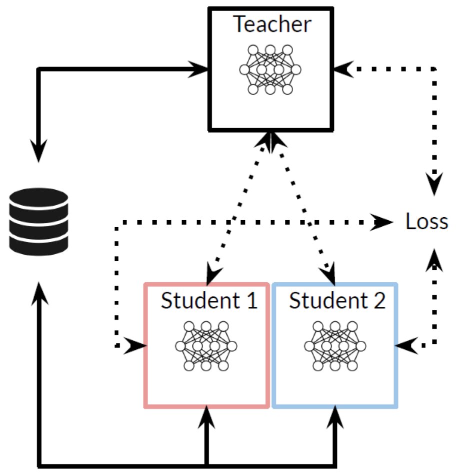
GoogLeNet



- For example, let's take a look at GoogLeNet. It's such a deep and complex network, that it doesn't even fit on this slide.
- The fact that it is so deep, gives it the ability to express complex relationships between features.
- But because it's so large, it's difficult or impossible to deploy it in many production environments, including mobile phones and edge devices.
- But can you have the best of both worlds, and **capture the knowledge contained** in a complex model like GoogLeNet in a much smaller, more efficient model?

Knowledge distillation

- Duplicate the performance of a complex model in a simpler model
- Idea: Create a simple *'student'* model that learns from a complex *'teacher'* model



- That's the goal of **knowledge distillation.**
- Rather than optimizing the network implementation, as we saw with quantization and pruning, knowledge distillation seeks to create a more efficient model which captures the same knowledge as a more complex model.
- If needed, further optimization can then be applied to the result.
- Knowledge distillation is a way to train a small model, to **mimic a larger model**, or even an ensemble of models.
- It starts, by first training a complex model or model ensemble to achieve a high level of accuracy.
- It then uses that model as a teacher for a simpler student model, which will then be the actual model that gets deployed to production.
- This teacher network can be either fixed or jointly optimized, and can even be used to train multiple student models of different sizes simultaneously.

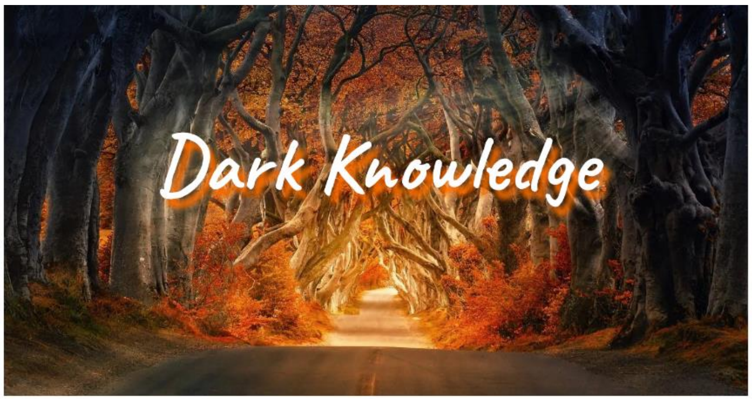
Knowledge Distillation Techniques

Teacher and student

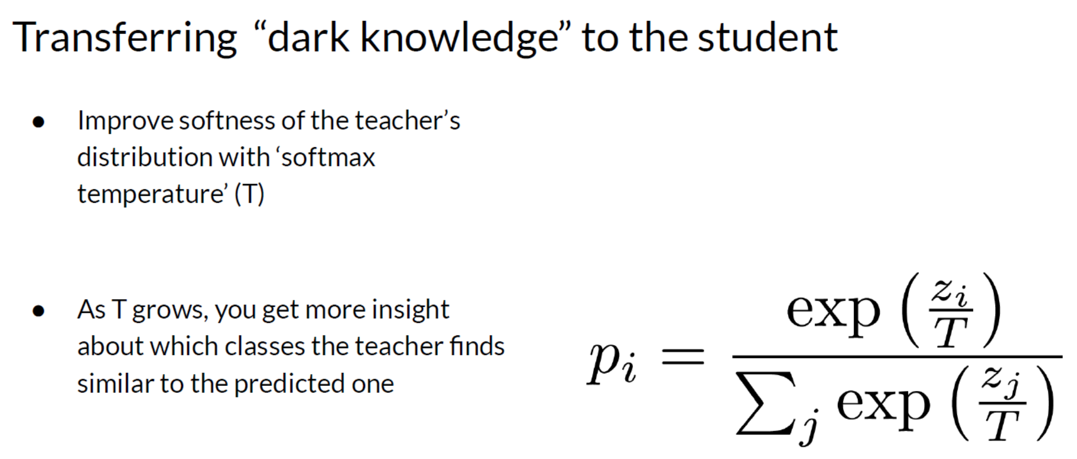
- Training objectives of the models vary
- Teacher (normal training)
  - maximizes the actual metric
- Student (knowledge transfer)
  - matches p-distribution of the teacher's predictions to form 'soft targets'
  - 'Soft targets' tell us about the knowledge learned by the teacher

- Let's take a deeper look at how knowledge distillation works.
- Knowledge distillation introduces the idea of a teacher model and a student model.
- In knowledge distillation, the training **objective functions** for the student and the teacher are **different**.

- The teacher will be trained first using a standard objective function that seeks to maximize the accuracy or a similar metric of the model. This is normal model training.
- The student then **seeks transferable knowledge**.
- It uses that objective function that seeks to **match the probability distribution of the predictions of the teacher.**
- Notice that the student is not just learning the teacher's predictions, but the **probabilities** of the predictions.
- The probabilities of the predictions of the teacher form **soft targets**, which provide more information about the knowledge learned by the teacher than the resulting predictions themselves.



- Don’t worry, knowledge distillation is not one of the dark arts, but it does involve dark knowledge.



Transferring “dark knowledge” to the student

- Improve softness of the teacher’s distribution with ‘softmax temperature’ (T)
- As T grows, you get more insight about which classes the teacher finds similar to the predicted one

$$p_i = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_j \exp\left(\frac{z_j}{T}\right)}$$

- The way knowledge distillation works is that you transfer knowledge from the teacher to the student by **minimizing a loss function**, in which the target is the distribution of class probabilities predicted by the teacher model.
- What happens here is that the **teacher model’s logits form the input to the final softmax layer**, which is often used since they provide more information about the probabilities of all target classes for each example.
- However, in many cases, this probability distribution has the correct class at a very high probability with all the other class probabilities very close to zero.
- So realistically, it sometimes doesn't provide much information beyond the ground truth labels already provided in the dataset.
- To tackle this issue, Hinton, Vinyals, and Dean, introduced the concept of a **softmax temperature**.
- By **raising the temperature** in the objective functions of the student and teacher, you can improve the softness of the teacher's distribution.

- In the formula here, the probability p of class i is calculated from the logits z as shown.
- T simply refers to the temperature parameter. When T is 1, you get the standard softmax function.
- But as T starts **growing**, the **probability distribution generated by the softmax function becomes softer, providing more information as to which classes the teacher found more similar to the predicted class**.
- The authors call this 'the dark knowledge' embedded in the teacher model. It is this dark knowledge that you are transferring to the student model in the distillation process.

Techniques

- Approach #1: **Weigh objectives** (student and teacher) and combine during backprop
- Approach #2: **Compare distributions of the predictions** (student and teacher) using KL divergence

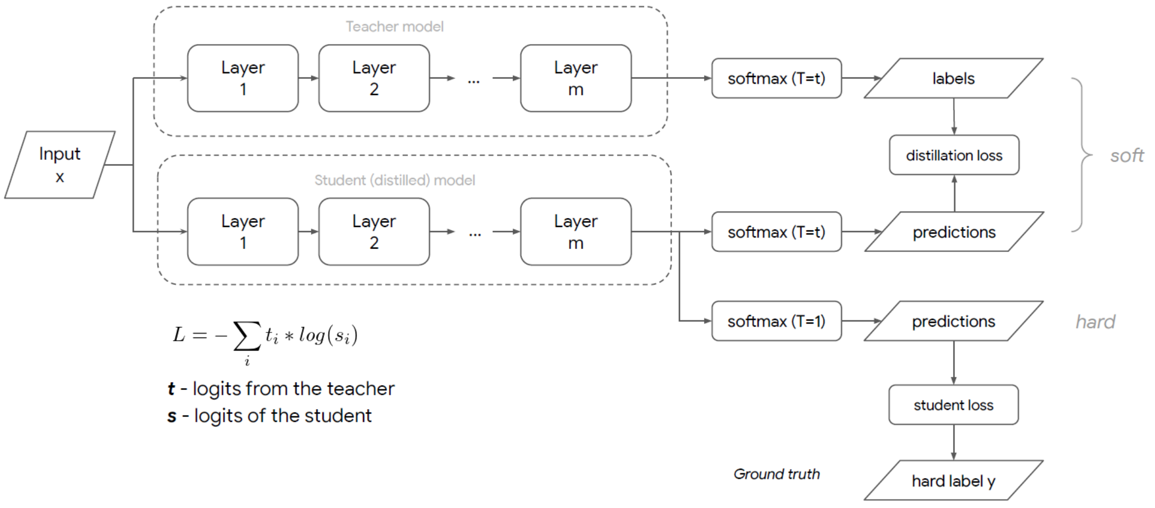
- Various techniques are used to train the student to match the teacher's soft targets.
- **Soft targets encode more information** about the knowledge learned by the teacher than its output class prediction per example as they include information about all the classes per training example through the probability distribution.
- In one approach, the student is trained on both the teacher's logits and the target labels using a normal objective function, and the two objective functions are weighted and combined in backpropagation.
- In another similar approach, the **distributions** of the student's predictions and the teacher's predictions are **compared** using metrics such as KL divergence.
- Let's look at the second technique now since it's more widely used.

KL divergence

$$L = (1 - \alpha) L_H + \alpha L_{KL}$$

- Generally, knowledge distillation is done by blending two loss functions, choosing a value for Alpha between zero and one.
- Here, L is the cross-entropy loss from the hard labels, and L_KL is the Kullback-Leibler divergence loss from the teacher's logits.
- In the case of heavy augmentation, you simply cannot trust the original hard labels due to the aggressive perturbations applied to data.
- In case of heavy data augmentation after training the teacher network, the alpha hyperparameter should be high in the student network loss function. This **high alpha** parameter would reduce the influence of the hard labels that went through aggressive perturbations due to data augmentation
- The Kullback-Leibler divergence here is a metric of the **difference between two probability distributions**.
- You want those two probability distributions to be as close as possible.
- The objective here is to make the distribution over the classes predicted by the student as close as possible to the teacher.

How knowledge transfer takes place



- When computing the loss function versus the teacher's soft targets, we use the same value of T (softmax temperature) to compute the softmax on the student's logits. This loss is the **distillation loss**.
- The authors also found another interesting behavior. It turns out that distilled models are able to produce the correct labels in addition to the teacher's soft targets.
- That means that you can calculate the standard loss between the student's predicted class probabilities and the ground truth labels. These are known as **hard** labels or targets. This loss is the **student loss**.
- When you're calculating the probabilities for the student, you set the softmax temperature (T) to one.

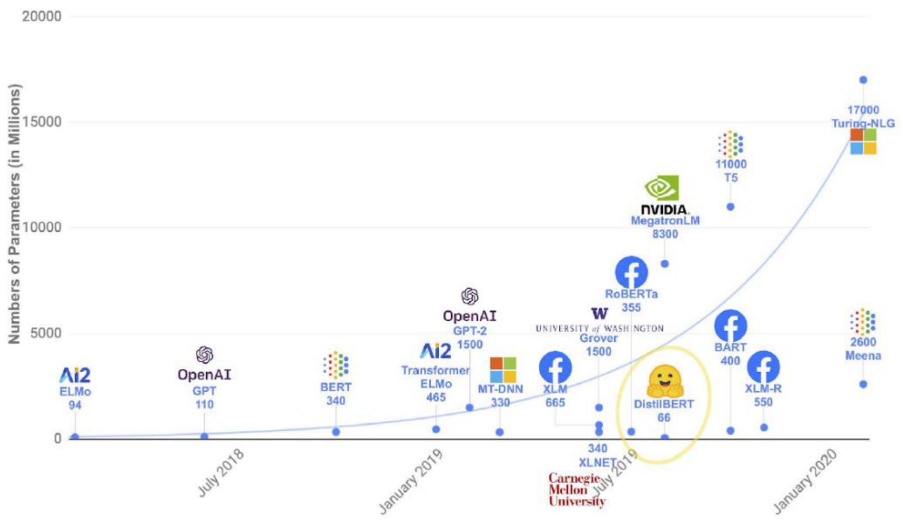
First quantitative results of distillation

Model	Accuracy	Word Error Rate (WER)
Baseline	58.9%	10.9%
10x Ensemble	61.1%	10.7%
Distilled Single Model	60.8%	10.7%

- The first quantitative results of applying knowledge distillation were promising.
- Hinton and his colleagues trained 10 separate models for an automatic speech recognition task using the same architecture and training procedure as the baseline.
- Automatic speech recognition tasks at the time relied on deep neural networks to map a short temporal context of features derived from the waveform to a probability distribution over the discrete states of a hidden Markov model.
- For the models, they randomly initialized their weights with different initial parameter values. This was essentially done so that there was enough diversity in the trained models.
- When averaging the ensemble's predictions, they would outperform the single models with ease.
- They also considered varying the sets of data that each model see, but they found that it wouldn't significantly impact the results so they decided to use this more straightforward strategy of comparing an ensemble of models against a single model.
- For the distillation process, they tried different values for the softmax temperature like **1, 2, 5, and 10**. They also used a relative weight of 0.5 on the cross-entropy for the hard targets.

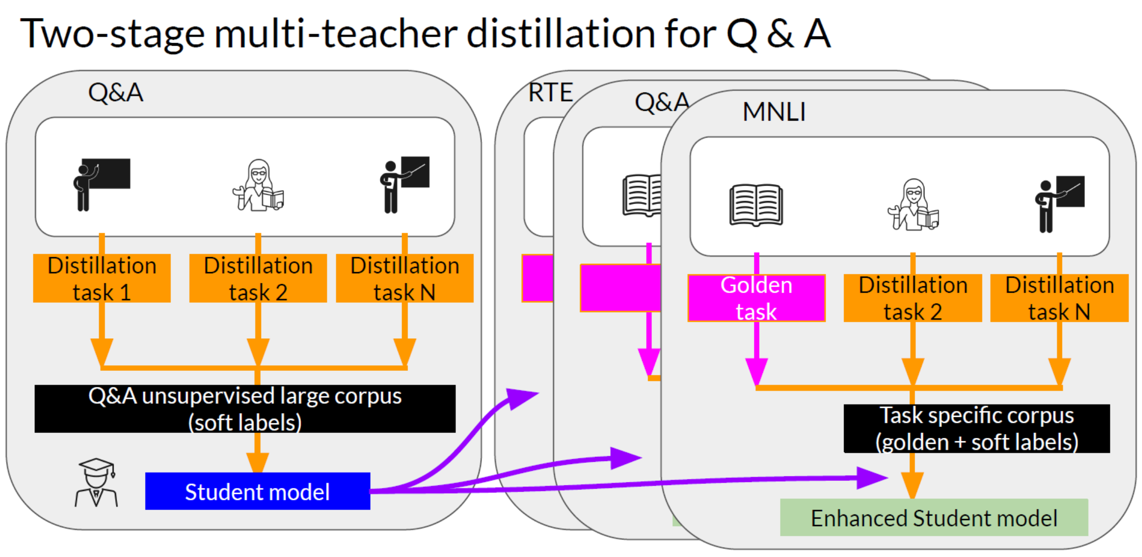
- This table shows that distillation can indeed extract more useful information from the training set than merely using the hard labels to train a single model.
- More than 80 percent of the improvement in accuracy achieved by an ensemble of 10 models is transferred to the distilled model.
- The ensemble gives a smaller gain on the ultimate objective of word error rate on a 23K-word test set due to the mismatch in the objective function.
- Still, again, the increase in the word error rate achieved by the ensemble is transferred to the distilled model.
- With this, they were able to show that the strategy of distilling models is indeed beneficial and can be used to achieve the desired effect of **distilling an ensemble of models into a single model that works significantly better than a model of the same size that has learned directly from the same training data**.

DistilBERT



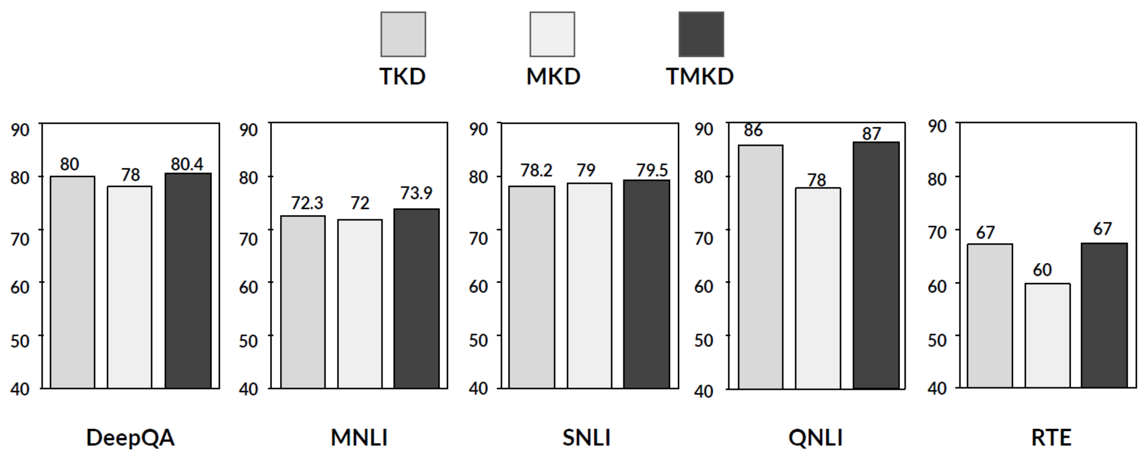
- In the real world though, people are more interested in deploying a low-resource model with close to state of the art results, but a lot smaller and a lot faster.
- That's why **Hugging Face created DistilBERT,** a distilled version of BERT which uses 40 percent fewer parameters, runs 60 percent faster while preserving 97 percent of BERT's performance as measured on the GLUE language understanding benchmark.
- Basically, it's a smaller version of BERT where the token type embeddings and the polar layer typically used for the next sentence classification task are removed.
- To create DistilBERT, the researchers at Hugging Face **applied knowledge distillation to BERT**, and hence the name DistilBERT.
- They kept the rest of the architecture identical while **reducing the number of layers**.

Case Study – How to Distill Knowledge for a Q&A Task



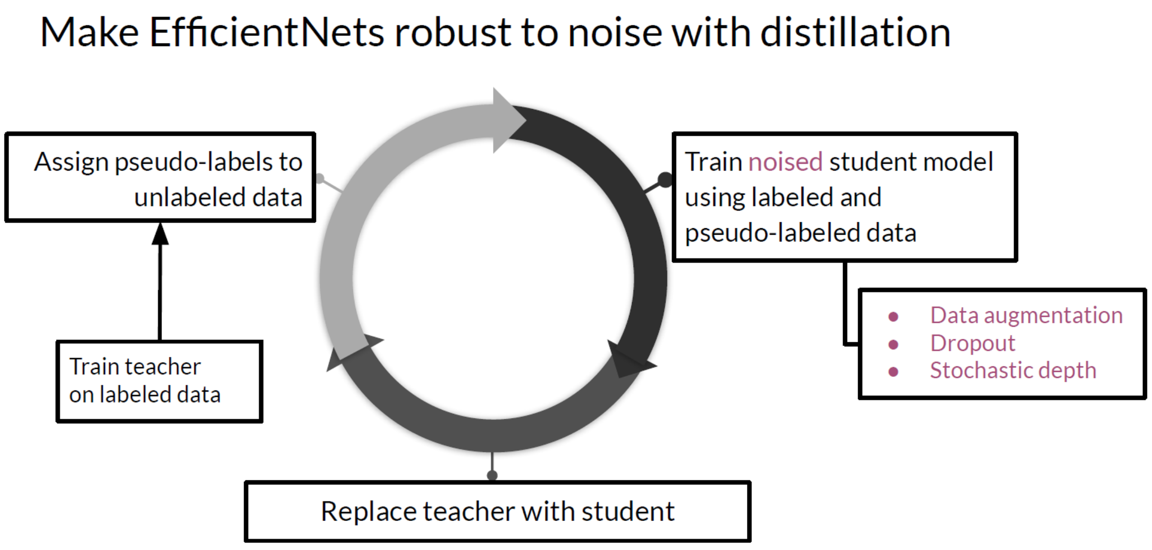
- Let's take a look at a specific implementation of knowledge distillation for a Q&A task, by looking at how knowledge can be distilled for question answering.
- Applying these complex models to real business scenarios become challenging due to the vast amount of model parameters.
- Older model compression methods generally suffer from **information loss during the model compression** procedure, leading to inferior models compared to the original one
- To tackle this challenge, researchers at Microsoft proposed a **two stage multi teacher knowledge distillation** method, or TMKD for short, for a web question answering system.
- In this approach, they **first developed a general Q&A distillation task for student model pre training**, and further **fine tune this pre trained student model with a multi teacher knowledge distillation model**.
- The basic knowledge distillation approach presented so far, is known as a **one on one model** because one teacher transfers knowledge to one student.
- Although this approach can effectively reduce the number of parameters and the time for model inference, due to the information loss during knowledge distillation, the performance of the student model is sometimes not on par with that of the teacher.
- This was the driving force for the authors to create a different approach called **M on M**, or many on many ensemble model, **combining both ensemble and knowledge distillation**.
- This involves first training multiple teacher models.
- The models could be BERT or GPT or others similarly powerful models, each having different hyper parameters.
- Then a student model for each teacher model is then trained.
- Finally, the **student models trained from different teachers are ensemble to create the final result**.
- Here, you prepare each teacher for a particular learning objective and then train them.
- Different models have different generalization of capabilities and they also overfit the training data in different ways, achieving performance close to the teacher model.

Impact of two-stage knowledge distillation



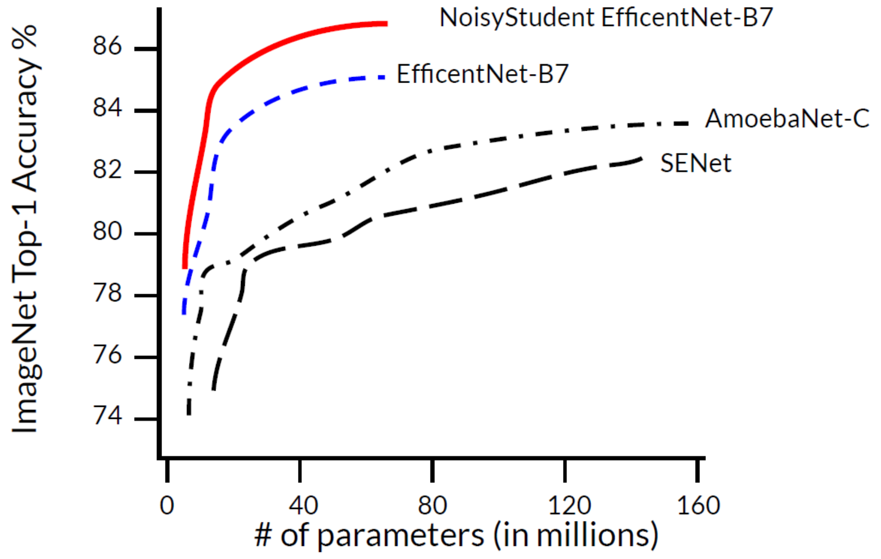
TKD = Temporal Knowledge Distillation, MKD = Multi-task Knowledge Distillation

- TMKD outperforms various state of the art baselines and it has been applied to real commercial scenarios
- Since ensembling is employed here, these compressed models benefit from large scale data and learned feature representations well.
- Results from experiments show that it can considerably outperformed baseline methods (TKD and MKD), and even achieved comparable results to the original teacher models along with a substantial speed up of model inference.
- The others performed experiments on several datasets using benchmarks that are public, and even large scale, to verify the method's effectiveness
- To support these claims, let's look at its advantages one by one.
- A unique aspect of TMKD is that it uses a multi teacher distillation task for student model pre training to boost model performance.
- To analyze the **impact of pre training**, the authors evaluated two models. The first one TKD is a three layer BERT-based model which is first trained using basic knowledge distillation pre training on the comQA dataset, and then fine-tuned on a task-specific corpus by using only one teacher for each task.
- The second model is a traditional knowledge distillation model which is again the same model but without the distillation pre training stage.
- TKD showed significant gains by leveraging large scale unsupervised Q& A pairs for distillation pre training.
- Another benefit of TMKD is its unified framework to learn from multiple teachers jointly.
- For this, the authors were able to compare the impact of multi teacher versus single teacher knowledge distillation using two models. MKD a three layer BERT-based model trained by multi teacher distillation without a pre training stage, and KD, a three layer BERT-based model trained by single teacher distillation without a pre training stage, whose aim is to learn from the average score of the teacher models.
- MKD outperforms KD on a majority of tasks demonstrating that a **multi teacher distillation approach can help the student model learn more generalized knowledge**, fusing knowledge from different teachers.
- Finally, they compared TKD, MKD and TMKD with each other.
- As you can see here, TMKD significantly outperforms TKD at MKD in all datasets, which verifies the complementary impact of the two stages: **distillation pre training and multi teacher fine tuning**.


Make EfficientNets robust to noise with distillation
Assign pseudo-labels to unlabeled data
Train noised student model using labeled and pseudo-labeled data
Data augmentation
Dropout
Stochastic depth
Train teacher on labeled data
Replace teacher with student

- In another example, researchers from Google brain and Carnegie Mellon University trained models with a **semi supervised learning** method called **noisy student**.
- In this approach, the knowledge distillation process is iterative. It uses a variation of the classic teacher student paradigm, but here, the **student is purposely kept larger** in terms of the number of parameters than the teacher.
- This is done so that the model can attain robustness to noisy labels as opposed to traditional knowledge distillation patterns.
- This works by first training an EfficientNet as the teacher model using labeled images, and then using the teacher to **generate pseudo labels** on a larger set of unlabeled images.
- Next they train another larger efficient net as a student model on the combination of labeled and pseudo labeled images.
- This algorithm was reiterated a number of times by treating the student model as a teacher model to relabel the unlabeled data and train a new student model.
- An important element of his approach was to ensure that **noise was added to the student model using dropout, stochastic depth, data augmentation, and so on**, via rand augment during its training.
- This noising pushed it to learn harder from pseudo labels.
- Adding noise to a student ensures that the task is much harder for the student and hence the name noisy student, and it doesn't merely learn the teacher's knowledge.
- In a side note, the teacher model is not noised during the generation of pseudo labels to ensure its accuracy isn't altered in any way.
- The loop closes by replacing the teacher with the optimized student network.

Results of noisy student training



- Comparing the results of noisy student training the authors used EfficientNet as their baseline models.
- This graph shows different sizes of efficient models along with some well-known state of the art models for comparison.
- Note the results of the noisy student in red.
- One key factor is that the data sets were **balanced across different classes, which improved training, especially for smaller models**.
- These results show that knowledge distillation isn't just limited to smaller models, like distilBERT, but can also be used to increase the robustness of an already great model using noisy student training.

References

- Distributed training
- Data parallelism
- Pipeline parallelism
- GPipe
- Knowledge distillation
- Q&A case study