Deep Learning and Its Applications in Signal Processing

Lesson 3: Distributed Deep Learning

Liang Dong, ECE

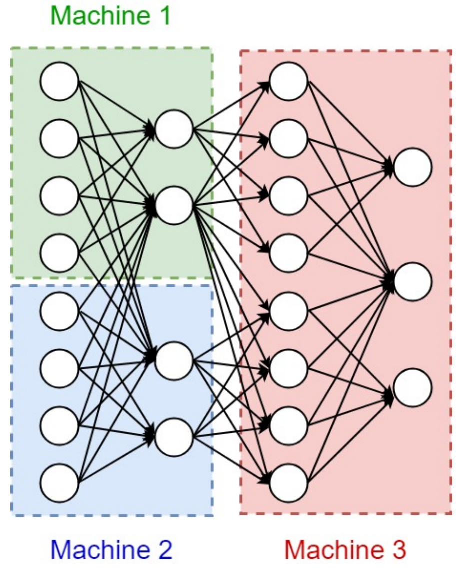
Challenges of Deep Neural Networks

- For today's competitive AI, it is the trend that both the volume of data and the complexity of deep neural networks increase.
- Even with significant advances in GPU hardware, network architecture and training methods, deep neural network training is computationally demanding.
- Solution: Distributed training of deep neural networks on parallel machines.
- Different aspects of training and inference of deep neural networks can be modified to increase concurrency.

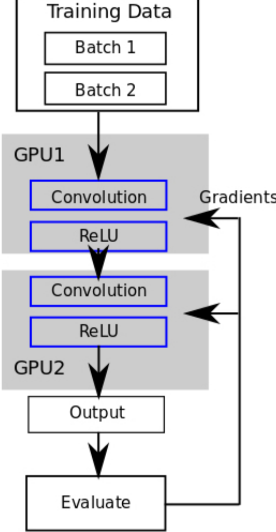
Jeffrey Dean, et. al., "Large Scale Distributed Deep Networks," *NIPS Proceedings*, 2012.

Distributed Training of Deep Neural Networks

- Model Parallelism



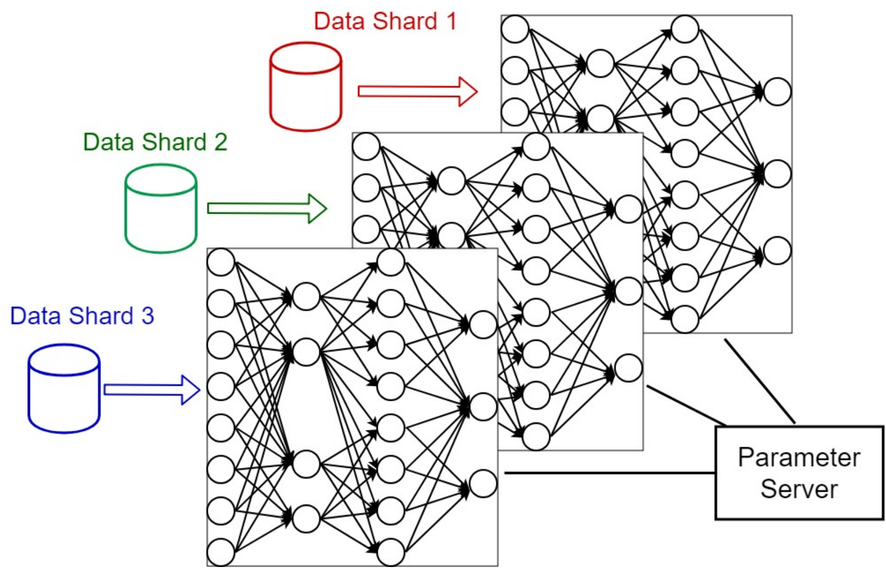
Model Parallelism



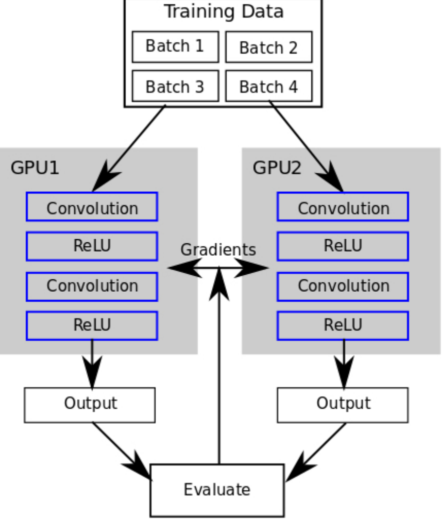
- Different worker machines in a distributed system are responsible for the computation in different parts of a single neural network.
- For example, each layer in the neural network may be assigned to a different machine.

Distributed Training of Deep Neural Networks

- Data Parallelism



Data Parallelism



- Each worker machine has a complete copy of the model.
- Each worker machine gets a different data section. That is, it is trained on a subset of the training data.
- The training results from the worker machines are combined in some way.

Combining Model Parallelism and Data Parallelism

- Model parallelism and data parallelism can be combined.

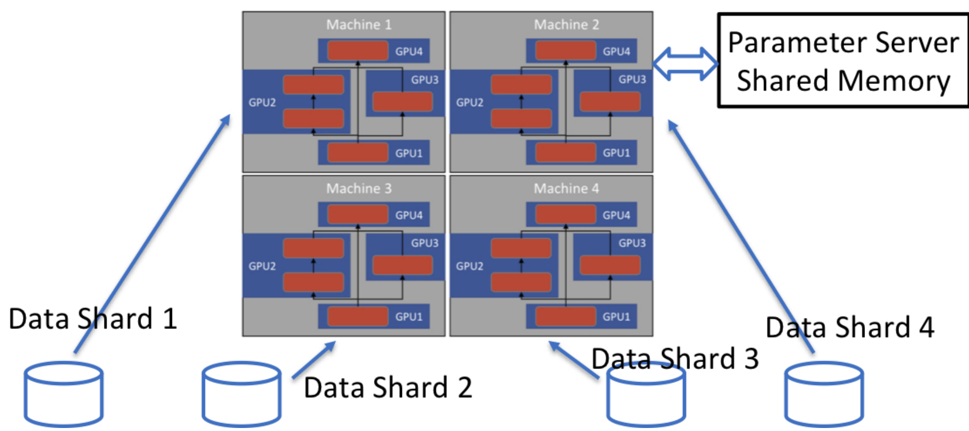
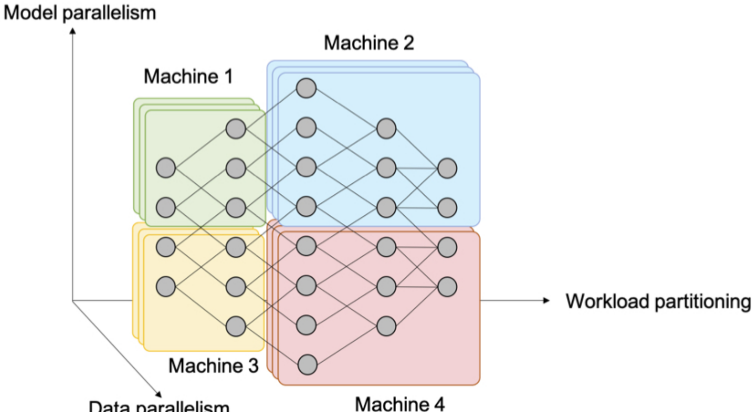


Figure: Multi-GPU systems clustering: We can use model parallelism (model partitioning across GPUs) for each machine, as well as data parallelism between machines.

Comparison of Model Parallelism and Data Parallelism



- Model Parallelism – scalable to large models
- Data Parallelism – easy implementation, good fault tolerance and cluster utilization

Distributed Training with Data Parallelism

- Keep a copy of the entire model on each worker machine, process a different subset of the training data on each worker machine.
- It needs some way to combine the results and a method of synchronizing the model parameters between the worker machines.
- Different approaches:

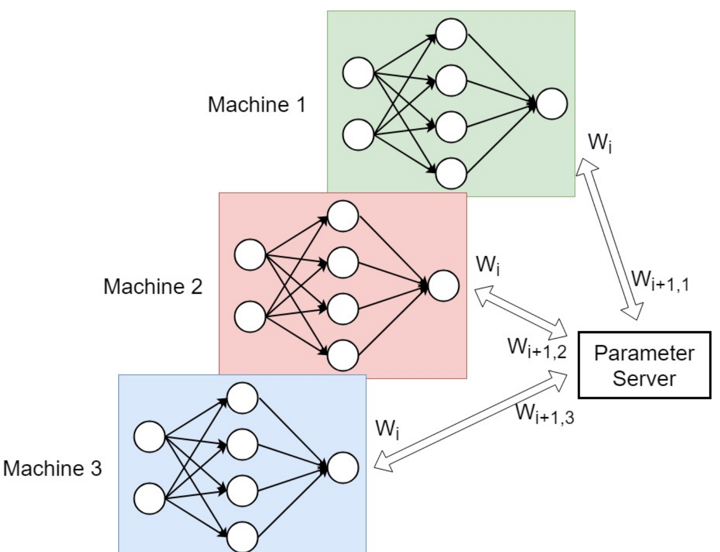
  – Parameter averaging vs. update-based (gradient-based) approach
  – Synchronous vs. asynchronous methods
  – Centralized vs. distributed synchronization

Parameter Averaging

Training Procedure of Parameter Averaging:

1. Randomly initialize network parameters based on the model configuration
2. Distribute a copy of the current parameters to each worker
3. Train each worker on a subset of the data
4. Set the global parameters to be the average of the parameters from each worker
5. While there are more data to process, go to step 2

Parameter Averaging



$$W_{i+1} = \frac{1}{N}\sum_{n=1}^{N} W_{i+1,n}$$

$N = 3$ in this example.

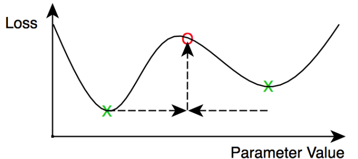
Parameter Averaging

- Parameter averaging is mathematically equivalent to training on a single machine, given that

  – Parameter averaging after each mini-batch
  – No internal update of the optimizer
  – An identical number of examples processed by each worker machine

Problems of Parameter Averaging

- The overhead of network communication and synchronization is high.
- If we average infrequently, the local parameters in the workers may diverge too much. This results in a poor model after averaging.

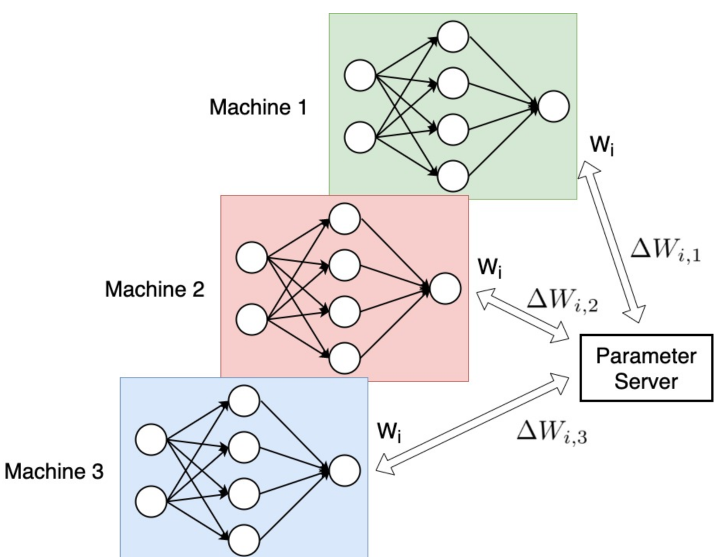


That is, the average of multiple different local minima is not guaranteed to be a local minimum.

Asynchronous Stochastic Gradient Descent (Async SGD)

- Instead of transferring parameters from the workers to the parameter server, we transfer the updates, i.e., the gradients.
- If the parameters are updated synchronously, the update-based approach of data parallelism is equivalent to the parameter averaging approach.

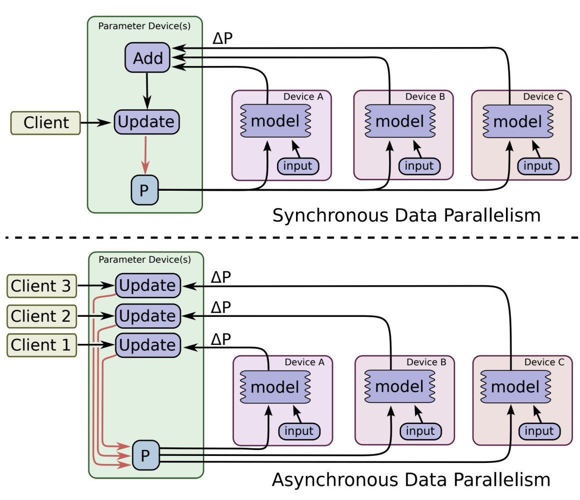
Asynchronous Stochastic Gradient Descent (Async SGD)



$$W_{i+1} = W_i - \lambda \sum_{n=1}^{N} \Delta W_{i,n}$$

$N = 3$ in this example.

Asynchronous Stochastic Gradient Descent (Async SGD)



- Update-based data parallelism becomes more useful when we relax the synchronous update requirement.
- We allow the updates $\Delta W_{i,n}$ to be applied to the parameter vector as soon as they are computed (instead of waiting for other $N-1$ workers).

Advantage of Async SGD

- Higher throughput in the distributed system:
  Worker machines can spend more time performing useful computation instead of waiting for the parameter averaging process to complete.

- Worker machines can potentially incorporate information (parameter updates) from other workers sooner than when using synchronous updating.

Problem of Async SGD – Stale Gradient Problem

- The calculation of gradients (updates) takes time. By the time a worker has finished these calculations and applies the results to the global parameter vector, the parameters may have been updated several times.

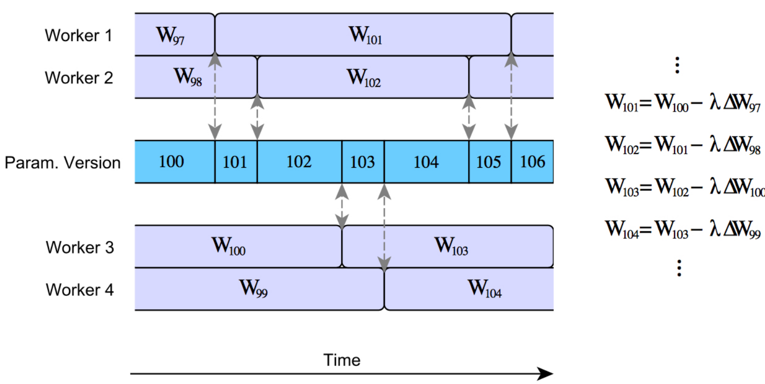


Figure: With asynchronous updates to the parameter vector, we introduce the stale gradient problem.

Stale Gradient Problem

- High gradient staleness can significantly reduce the network convergence speed and even prevent some configurations from converge.
- Many variants of Async SGD maintain the basic approach, but apply various strategies to minimize the effects of stale gradients.

Approaches to Dealing with Stale Gradients

- Scaling λ separately for each update $\Delta W_{i,n}$ based on the staleness of the gradients, such that stale gradients have a smaller impact on the parameter vector.

$$W_{i+1} = W_i - \sum_{n=1}^{N} \lambda_n \Delta W_{i,n}$$

- Soft Synchronization:
  Instead of updating the global parameter vector immediately, the parameter server waits to collect some number S of updates $\Delta W_{i,n}$ from any of the N learners. $(1 \leq S \leq N)$

$$W_{i+1} = W_i - \sum_{n=1}^{S} \lambda_n \Delta W_{i,n}$$

Approaches to Dealing with Stale Gradients

- Using synchronization to bound staleness:

  We delay the faster workers when necessary to ensure that the maximum staleness is below a certain threshold.

Decentralized Async SGD

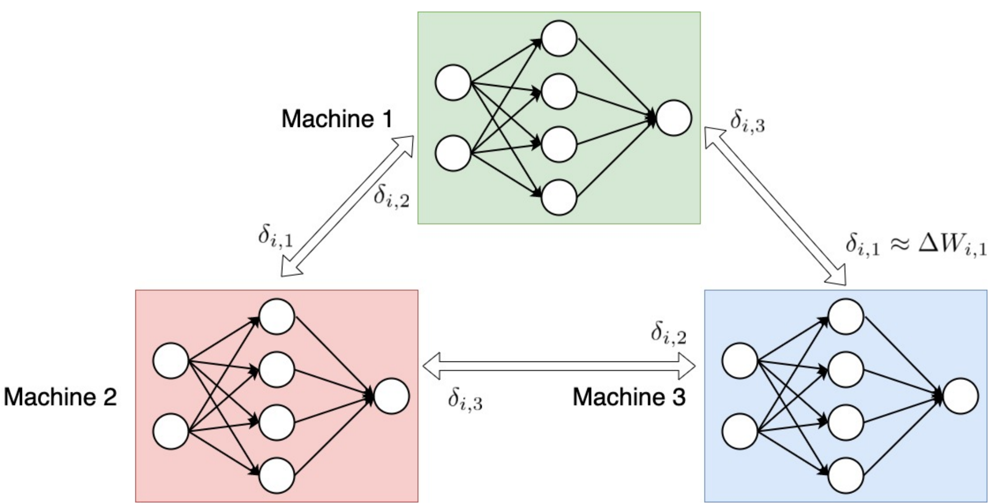


Figure: There is no centralized parameter server in the system. Instead, peer-to-peer communication is used to transfer model updates between workers.

Decentralized Async SGD – Compressed/Quantized Update Vector $\boldsymbol{\delta}_{i,n}$

- Updates can be heavily compressed, so that network traffic can be reduced by orders of magnitude.
- Compressed and quantized update vectors $\boldsymbol{\delta}_{i,n}$:

  – Sparse: Only some gradients are passed in each vector $\boldsymbol{\delta}_{i,n}$ (the others are assumed to be 0). Sparse entries are encoded using an integer index (to identify the entries in the sparse array).

  – Quantized to a single bit: Each element of the sparse update vector takes value $+\tau$ or $-\tau$. The value of τ is the same for all elements of the vector, hence only a single bit is required to differentiate between the two options.

  – Integer indexes can be compressed using entropy coding.

Decentralized Async SGD – Residual Vector $\mathbf{r}_n$

- Residual vector $\mathbf{r}_n$:

  – The difference between the original update vector $\Delta W_{i,n}$ and the compressed/quantized update vector $\boldsymbol{\delta}_{i,n}$ is stored in a residual vector $\mathbf{r}_n$ on worker n, instead of simply being discarded.

  – We quantize and transmit the compressed version of $\mathbf{r}_n$ at each step as well as updating $\mathbf{r}_n$ appropriately.

  – The net effect is that all information from the original update vector $\Delta W_{i,n}$ is only delayed but not lost.

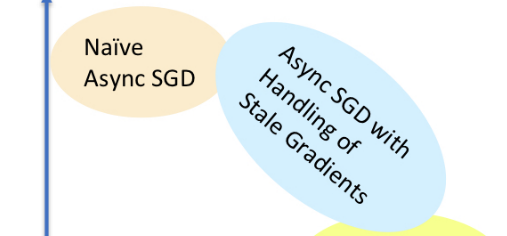
Problems of Decentralized Async SGD

- Convergence may be affected in the early stages of training. It may help to solve this problem by using fewer compute nodes for a part of an epoch.
- Compression and quantization are not free. These processes result in extra computation time for each minibatch, as well as a small amount of memory overhead per worker machine.

Distributed Neural Network Training

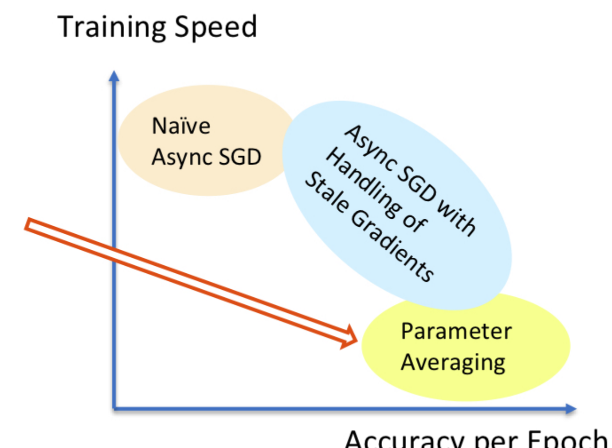
Choose approaches according to the criteria:

- Fastest training speed (highest number of training examples per second, or lowest time per epoch)
- Maximum attainable accuracy as epochs $\to \infty$, for a given amount of time, or for a given number of epochs



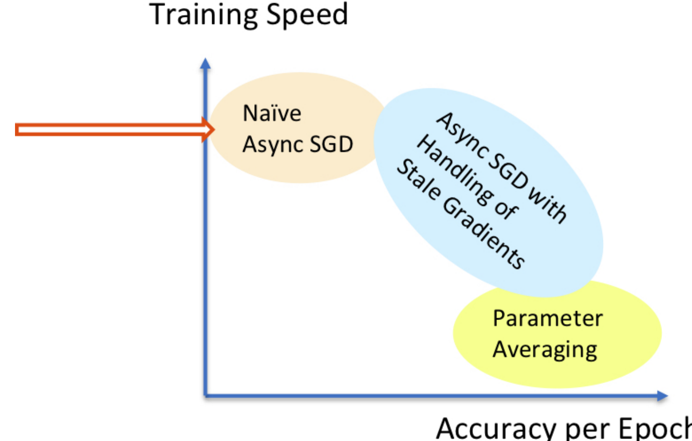
Distributed Neural Network Training

- Parameter averaging has the "last executor" effect: Synchronous systems have to wait on the slowest executor before completing each iteration.
- Consequently, synchronous systems are less viable as the total number of workers increases.



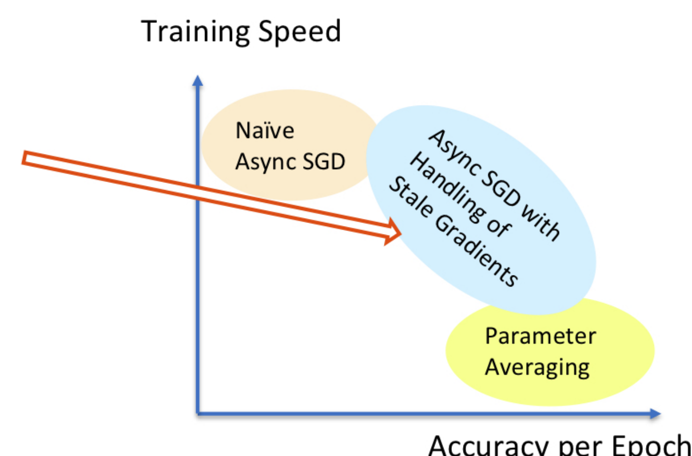
Distributed Neural Network Training

- Asynchronous SGD is a good option for training as long as gradient staleness is appropriately handled.



Distributed Neural Network Training

- Softsync approach can be viewed as spanning a continuum between naïve asynchronous SGD and synchronous implementations, depending on the hyperparameters used.



Centralized versus Decentralized Async SGD

- An asynchronous SGD implementation using a centralized parameter server may introduce a communication bottleneck.
- Utilizing N parameter servers, each handling an equal fraction of the total parameters is a solution to the communication bottleneck problem.
- Decentralized asynchronous SGD is a promising idea with implementations of compression, quantization, etc. of parameter updates.

Distributed Deep Learning Considerations

- Distributed learning systems have overhead compared to training on a single machine due to synchronization and network transfers of data and parameters.
- Setup (i.e., preparing and loading training data) and hyperparameter tuning can be more complex in distributed systems.
- Distributed training tends to be more efficient when the ratio of transfers to computation is low.
- Small and shallow networks are not good candidates for distributed training as they don't have much computation per iteration.
- Networks with parameter sharing (such as CNNs and RNNs) are good candidates for distributed training.

Distributed Deep Learning Considerations

Network Size ↑	Small data	Large data
Large network	Overfitting? Distributed Learning	Distributed Learning
Small network	Single Machine Learning	Underfitting? Distributed Learning

Data Size →

- Distributed deep learning can be considered when either network size is large or the amount of data is large.
- However, a mismatch between the two (large network and small data; small network and lots of data) may lead to underfitting or overfitting – Poor generalization of the final trained model.

Distributed Multi-GPU and TPU Training with

- Model parallelism using multi-GPU systems may be viable for large networks.
- Data parallelism: Keras has a built-in utility, keras.utils.multi_gpu_model, which can produce a data-parallel version of any model.

```
import tensorflow as tf
import numpy as np

from keras.models import Sequential, Model
from keras.layers import Dense, Conv2D, MaxPooling2D, Flatten, Input
from keras.utils import multi_gpu_model
```

Distributed Multi-GPU and TPU Training with


```
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_train, x_test = x_train/255.0, x_test/255.0
x_train = np.float32(x_train)

inputs = Input(shape=(28, 28, 1))
x = Conv2D(32, (5, 5))(inputs)
x = MaxPooling2D(pool_size = (2, 2))(x)
x = Conv2D(64, (5, 5))(x)
x = MaxPooling2D(pool_size = (2, 2))(x)
x = Flatten()(x)
x = Dense(1024, activation = tf.nn.relu)(x)
outputs = Dense(10, activation = tf.nn.softmax)(x)
```

```
with tf.device('/cpu:0'):
  model = Model(inputs, outputs)

# Instantiate the base model.
# Model's weights are hosted on CPU memory.

def f1_score(y_true, y_pred):
  y_true_pos = y_true*y_pred
  sum_true_pos = tf.reduce_sum(y_true_pos)
  sum_true = tf.reduce_sum(y_true)
  sum_pred = tf.reduce_sum(y_pred)
  precision = sum_true_pos/sum_true
  recall = sum_true_pos/sum_pred
  f1 = 2*precision*recall/(precision + recall)
  return(tf.reduce_mean(f1))
```

Distributed Multi-GPU and TPU Training with Keras

```
try:
  pmodel = multi_gpu_model(model, gpus=2)
  # Replicates the model on 2 GPUs.
  print("Training using multiple GPU...")
except ValueError:
  pmodel = model
  print("Training using single GPU or CPU...")

pmodel.compile(optimizer = 'adam',
          loss = 'sparse_categorical_crossentropy',
          metrics = ['accuracy', f1_score])

pmodel.fit(x_train, y_train, epochs = 5, batch_size=100)

[loss_value, accuracy, f1_score] = pmodel.evaluate(x_test, y_test)
```

```
print("Loss: ", loss_value)
print("Accuracy: ", accuracy)
print("F1_Score: ", f1_score)
pmodel.summary()
```

Distributed Multi-GPU and TPU Training with Keras

```
try:
  pmodel = multi_gpu_model(model, cpu_relocation=True)
  # Training models with weights merge on CPU using cpu_relocation
  print("Training using multiple GPU...")
except ValueError:
  pmodel = model
  print("Training using single GPU or CPU...")

pmodel.compile(optimizer = 'adam',
          loss = 'sparse_categorical_crossentropy',
          metrics = ['accuracy', f1_score])

pmodel.fit(x_train, y_train, epochs = 5, batch_size=100)

[loss_value, accuracy, f1_score] = pmodel.evaluate(x_test, y_test)
```

```
print("Loss: ", loss_value)
print("Accuracy: ", accuracy)
print("F1_Score: ", f1_score)
pmodel.summary()
```

Distributed Multi-GPU and TPU Training with

- Device Parallelism: It works best for models that have a parallel architecture, e.g., a model with multiple branches.
- This can be achieved by using TensorFlow device scopes.

Distributed Multi-GPU and TPU Training with Keras

```
# Model where a shared LSTM is used to encode two different sequences in parallel
input_a = keras.Input(shape=(140, 256))
input_b = keras.Input(shape=(140, 256))

shared_lstm = keras.layers.LSTM(64)

with tf.device_scope('/gpu:0'):                          # Process the first sequence on one GPU
    encoded_a = shared_lstm(tweet_a)

with tf.device_scope('/gpu:1'):
    encoded_b = shared_lstm(tweet_b)

with tf.device_scope('/cpu:0'):
    merged_vector = keras.layers.concatenate([encoded_a, encoded_b], axis=-1)
```

Distributed Multi-GPU and TPU Training with


```
# Model where a shared LSTM is used to encode two different sequences in parallel
input_a = keras.Input(shape=(140, 256))
input_b = keras.Input(shape=(140, 256))

shared_lstm = keras.layers.LSTM(64)

with tf.device_scope('/gpu:0'):
        encoded_a = shared_lstm(tweet_a)

with tf.device_scope('/gpu:1'):
        encoded_b = shared_lstm(tweet_b)

with tf.device_scope('/cpu:0'):
        merged_vector = keras.layers.concatenate([encoded_a, encoded_b], axis=-1)
```

 # Process the next sequence on another GPU

Distributed Multi-GPU and TPU Training with Keras

```
# Model where a shared LSTM is used to encode two different sequences in parallel
input_a = keras.Input(shape=(140, 256))
input_b = keras.Input(shape=(140, 256))

shared_lstm = keras.layers.LSTM(64)

with tf.device_scope('/gpu:0'):
        encoded_a = shared_lstm(tweet_a)

with tf.device_scope('/gpu:1'):
        encoded_b = shared_lstm(tweet_b)

with tf.device_scope('/cpu:0'):
        merged_vector = keras.layers.concatenate([encoded_a, encoded_b], axis=-1)
```

 # Concatenate results on CPU