Assignment - 2

Pallavi. K. G
1JT20AI026.

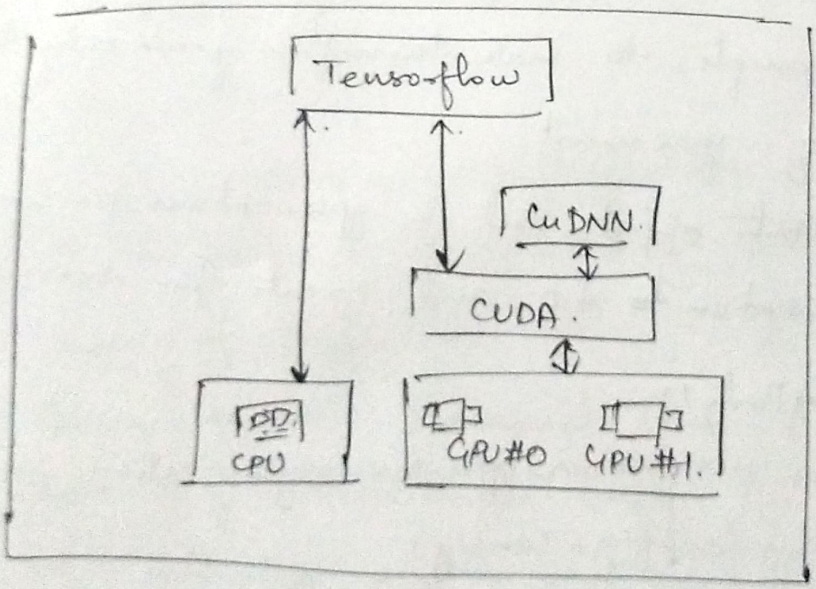
Neural Networks and deeplearning

1). With a neat diagram explain how tensorflow used CUDA and CuDNN to control GPU's and boost DNNs (or) with diagram explain how tensorflow can use multiple GPU cards on one Machine.

Ans: To utilize multiple GPU cards on a single machine with tensorflow, you need to ensure that your GPU cards have NVidia Compute Capability of 3.0 or higher. This includes cards like Nvidia's titan, titan x, k90, and k40. Once you have a compatible card download the necessary libraries.

CUDA (Computer Unified Device Architecture): allows ~~devel~~ developers to CUDA-related GPU's for all sorts of computations.

cuDNN: (CUDA Deep Neural Network): is a GPU accelerated library for primitives for DNNs.



Tensorflow uses CUDA and CuDNN to control the GPU cards and accelerate computations.

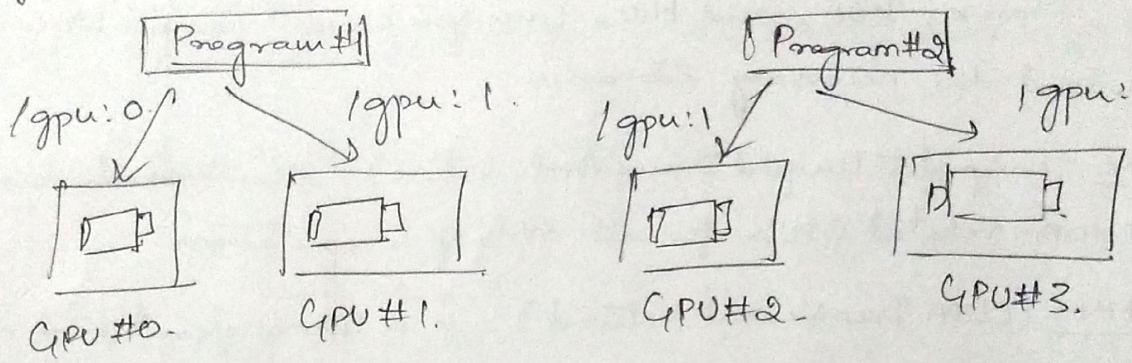
2). How tensorflow manages GPU RAM.

Ans: By default tensorflow automatically grabs all the RAM in all available GPU's the first time you run a graph, so you will not be able to start a second tensorflow program while

first one is still running. If you try, you'll get an error.
One solution is to run each process on different GPU cards. To do this, the simplest option is to set the CUDA_VISIBLE_DEVICES environment variable so that each process only sees the appropriate GPU cards.

Ex: CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py

program #1 will only see GPU cards 0 and 1. and program #2 will only see GPU cards 2 and 3.

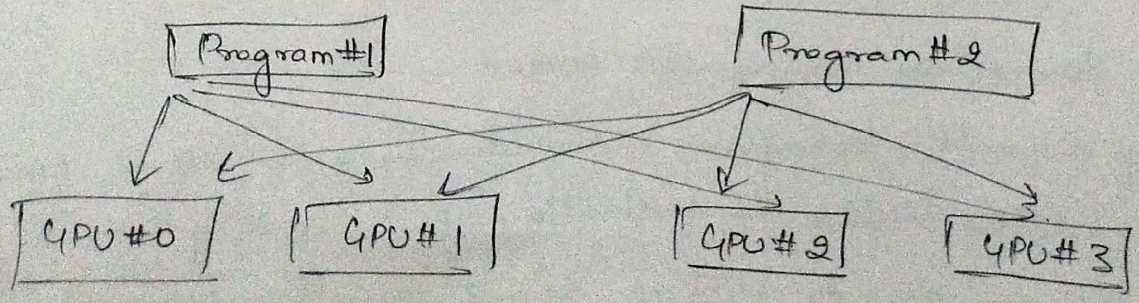


Another option is to tell tensorflow to grab only a fraction of the memory, for example, to make tensorflow grab only 40% of each GPU's memory, you must

create a ConfigProto object, set its gpu_options.per_process_gpu_memory_fraction option to 4.0, and create the session.

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4.
Session = tf.session(config=config).
```

Now 2 programs like this one can run parallel using the same GPU cards.



3). Placing operations on Devices on tensorflow

Ans: * Simple placement: whenever you run a graph, if tensorflow needs to evaluate a node that is not yet placed on a device, it uses the simple placer to place it. The simple placer follows the rules:

→ If a node was already placed on a device in a previous run of the graph, it is left on that device.

→ Else, if the user pinned a node to a device, the placer places it on that device

→ Else, it defaults to GPU #0, or the CPU if there is no GPU.

Ex: with tf.device ("/cpu:0")

a = tf.variable(3.0)

b = tf.constant(4.0)

c = a*b.

* Logging placements

For this you need to set the log-device-placement option to True; this tells the placer to log a message whenever it places a node.

Ex: Config = tf.ConfigProto()

Config.log-device_placement = True.

sess = tf.Session(config = config)

>>> x.initializer.run(session = sess).

>>> sess.run().

* Dynamic placement function:

when you create a device block, you can specify a function instead of a device name. Tensorflow will call this function for each operation it needs to place in the device block, and the function must return the name of the device to pin the operation on

```
def variables-on-cpu(op):
    if op.type = 'Variables':
        return = "/cpu:0"
    else:
        return "/gpu:0"

with tf.device(variables-on-cpu):
    a = tf.Variables(3.0)
    b = tf.constant(4.0)
    c = a*b
```

* Operations And kernals

For a ~~tensor~~ tensorflow operation to run on a device, it needs to have an implementation for that device; this is called a kernel. Many operations have kernals for both CPU's and GPUs, but not all of them.

Ex:
```
with tf.device("/gpu:0"):
    i = tf.variable(3).
sess.run(i.initializer).
```

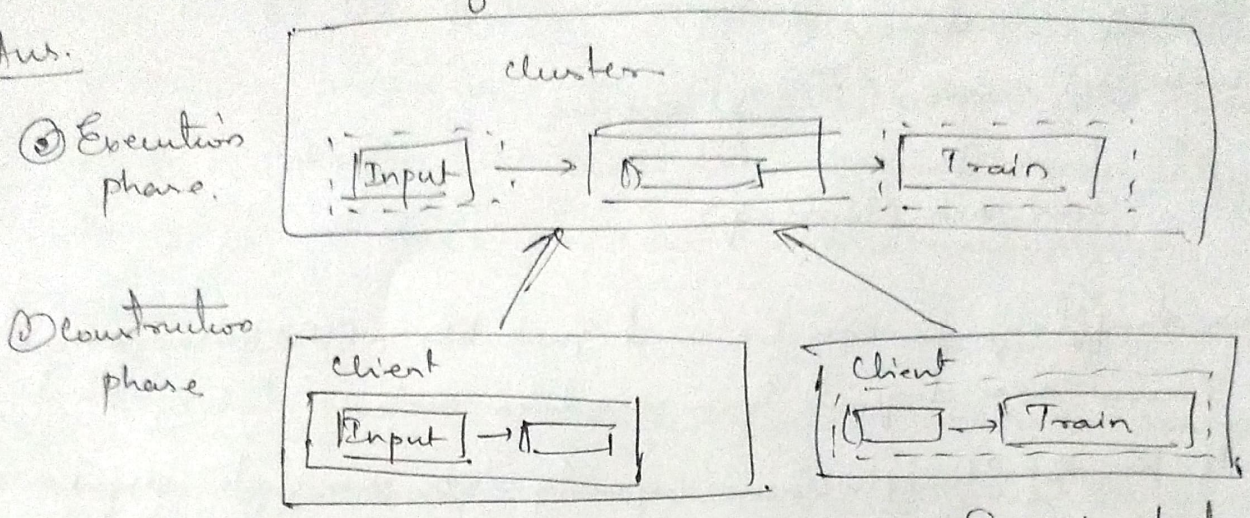
o/p: errors:

* Soft placement: By default, if you try to pin an operation on a device for which the operation has no kernel, you get the exception shown earlier when tensorflow tries to place the operation on device. If you prefer tensorflow to fall back to the CPU instead, you can set the allow_soft_placement configuration option to true:

```
with tf.device("/gpu:0")
    i = tf.variable(3)
Config = tf.configproto()
Config.allow_soft_placement = True
Sess = tf.session(config=(config)
```

5. With a neat diagram and code snippets illustrate asynchronous communication using tensorflow queues.

Ans.



Tensorflow provides various kinds of queues. The simplest kind is the first-in first-out (FIFO) queue.

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.float32], shapes=[[2]],
        name='q', shared_name="shared_q")
```

Enqueuing Data: To push data to a ~~queue~~ queue, you must create an enqueue operation.

```
training_instances = tf.placeholder(tf.float32, shape=(None,2))
enque_many = q.enqueue([training_instances])
with tf.Session("grpc://machine-a.example.com:2222") as sess:
    sess.run(enque_many, feed_dict={training_instances:
            [[1.,2.],[3.,4.],[5.,6.]]})
```

Dequeuing Data: To pull the instances out of the queue, on the other ~~hand~~ end, you need to use a dequeue operation.

Queues of tuples: Each item in a queue can be a tuple of tensors instead of just a single tensor.

Ex:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.int32, tf.float32],
        shapes=[[], [3,2]], name="q", shared_name="shared_q")
```

closing a queue : It is possible to close a queue to signal to the other sessions that no more data will be enqueued.

```
close_q = q.close()
with tf.Session([...]) as sess:
    [...]
    sess.run(close_q)
```

Random shuffle queue can be used just like a FIFO Queue except that items are dequed in a random order.

```
q = tf.RandomShuffleQueue(capacity=50, min_after_deque=10,
    dtypes=[tf.float32], shapes=[()], name="q", shared_name
    = "shared_q")
```

6) Compare and contrast In graph and Between graph replication

Ans:

In graph	Between graph
+ All neural networks are integrated into a single large graph.	+ Each neural network operates within its own graph structure
+ Coordination and computation are handled within a single session.	+ Synchronization & coordination are managed accross multiple graphs.
+ Simplicity in implementation due to centralized management.	+ Requires the setup of multiple clients and queues for communication.
+ All components are contained within one graph structure.	+ Offers greater flexibility in handling complex scenarios.
+ Requires less overhead in managing multiple entities.	+ Allows for customization and optimization of individual neural networks.
+ Offers a clear overview of the entire computation flow.	+ More complex to implement due to the need for managing multiple entities.

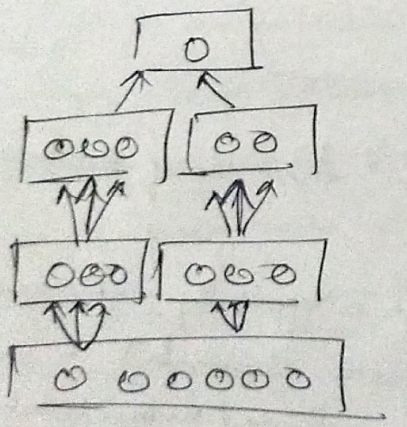
7. What is Model parallelism? Explain different types.

Ans: Model parallelism is a technique used in distributed computing where a single neuron is split into separate parts or chunks and each part is processed on a different computational device or node simultaneously

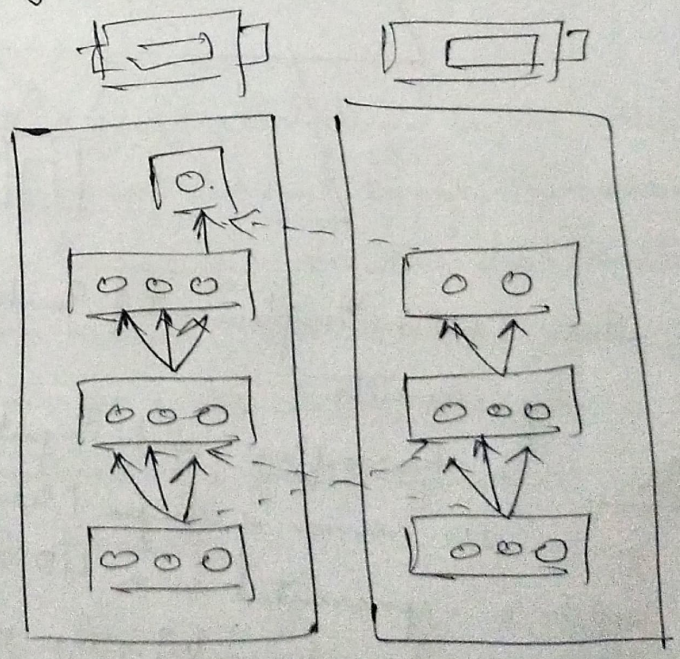
Types:

* Horizontal Model Parallelism: different layers or components of the neural network are assigned to different devices. Each device processes a portion of the data independently, but there is often significant communication overhead b/w devices, especially if the layers are fully connected or tightly coupled.

* Vertical Model Parallelism: the neural n/w is split along the depth or "vertical" dimension of the n/w architecture. Each device is responsible for computing a subset of the operations within a layer



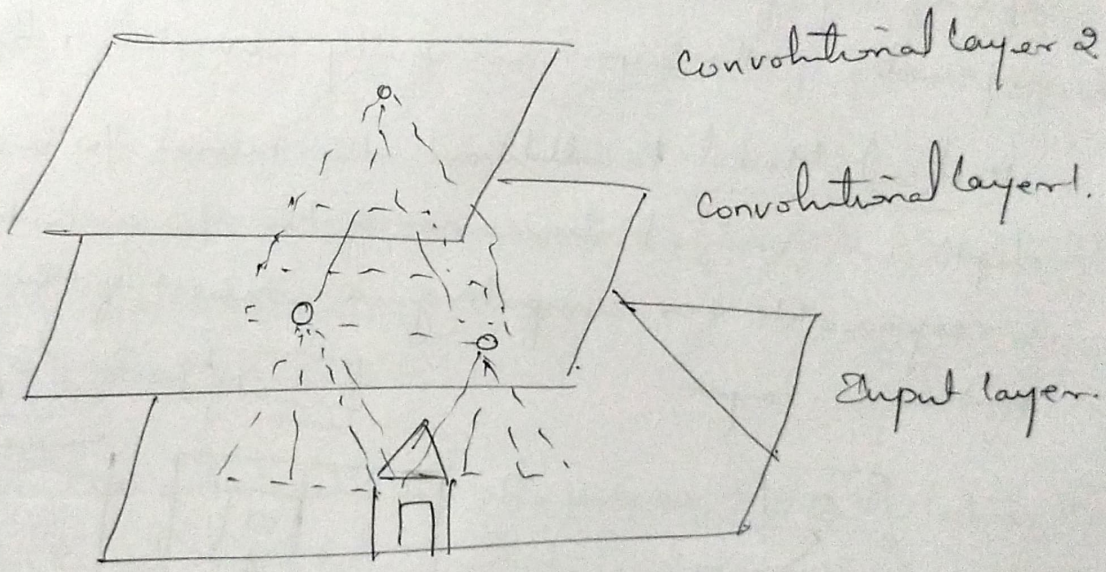
Parallel connected neural n/w.



Vertical split.

8. Illustrate convolutional layer with a neat diagram. How to compute the output of a neuron in a convolution layer?

Ans: The most imp. building block of a CNN is the convolutional layer: neurons in the 1st layer are not connected to every single pixels in their respective fields. In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to concentrate on low level features in the 1st hidden layer, then assemble them into high-level features in the next hidden layer, and so on.



9. How CNN is implemented in tensorflow? Also brief the memory requirements.

Ans: In tensorflow, each input image is typically represented as a 3D tensor of shape [height, width, channels]. A mini-batch is represented as a 4D tensor of shape [mini_batch size, height, width, channels]. The weights of a convolutional layer are represented as a 4D tensor of shape [f_h, f_w, f_n, $f_{n'}$]. The bias term of a convolutional layer are simply represented as a 1D tensor of shape [f_n].

fully connected

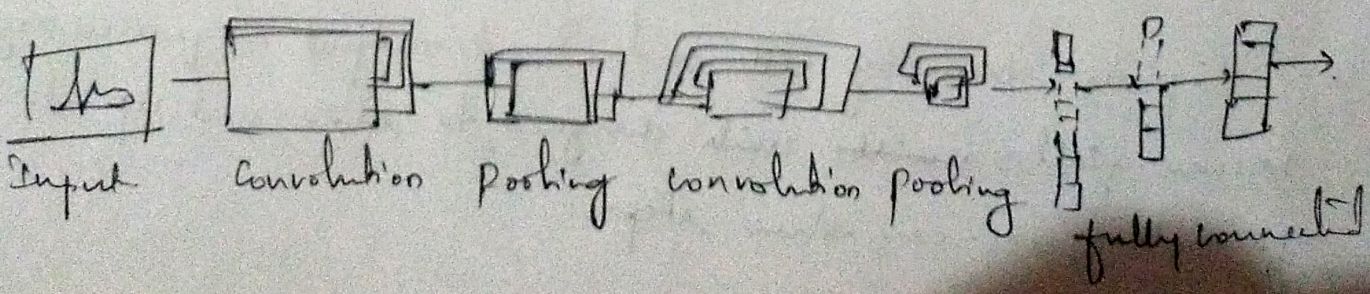
Ex: import numpy as np.

from sklearn.datasets import load_sample_images.

```
dataset = np.array(load_sample_images().images, dtype=np.float32)
batch_size, height, width, channels = dataset.shape.
filters_test = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters_test[:, 3:, 0] = 1.
filters_test[3, :, :1] = 1
x = tf.placeholder(tf.float32, shape=(None, height, weight, channels))
convolution = tf.nn.conv2D(x, filters, strides=[1,2,2,1], padding=
                                                        'same').
with tf.Session() as sess:
    output = sess.run(convolution, feed_dict={x: dataset})
plt.imshow(output[0,:,:,1]).
plt.show()
```

10) Explain CNN architectures. list the various CNN architectures. Explain AlexNet, ResNet, GoogleNet.

Ans: CNN architectures: stack a few convolutional layers, then a pooling layer, then another few convolutional layers, then another pooling layer and soon. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper. ~~then~~ At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers, and the final layer o/p the prediction.



Types: + LeNet-5
+ AlexNet
+ GoogleNet
+ ResNet

+ AlexNet: won 2012 imagenet compilation. It is similar to LeNet-5, only much larger and deeper and it was the first to stack convolutional layers directly on top of each other, instead of stacking a pooling layer on top of each convolutional layer.

+ GoogleNet: won imagenet compilation at 2014. ~~Two~~ Inception modules allows googlenet to ~~allow~~ use parameters much more efficients than previous architecture.

+ ResNet: won imagenet computation at 2015.
Uses skip connection: the signal feeding into a layer is also added to the o/p of a layer located a bit higher up the stack.

11Y. with neat diagrams and equations explain ~~resnet~~ recurrent neurons.

Ans: A recurrent neuron (h_t) at time step (t) takes 2 inputs: the current input (x_t) and the previous state (h_{t-1}). It computes its output (h_t) as a function of both the current inputs and the previous state:

$$h_t = f(x_t, h_{t-1})$$

this function (f) can be vary depending on the specific architecture of the recurrent neuron

19) Explain memory cells and input output sequence organization
+ Memory cell: ~~units~~ units designed to store and update information accross time steps.

Input output sequence organization: organized in a time-dependent manner. At each each time step (t), the n/w receives an i/p (x_t) and produces an o/p (y_t). The n/w's internal state (h_t), which encapsulates the memory of the n/w up to time step (t), is updated based on current i/p (x_t) and the previous state (h_{t-1}).

13. Explain static and dynamic unrolling.

Ans: Static rolling: computation graph of the RNN is constructed for a fixed no. of time steps during graph creation phase.

Dynamic rolling through time: the computational graph is constructed dynamically during the execution phase, allowing for flexibility in handling sequences of varying lengths.

14. Explain BPTT.

Ans: It is technique used to train ~~recurrent~~ RNN by unrolling the n/w over a sequence of inputs & computing gradients through time.

* Unrolling through time
* Gradient computation
* ~~param~~ parameters update.

Q17 Explain RNN.

Ans: commonly used to train & predict time series data due to their ability to capture temporal dependencies

Training with RNNs.

+ Data preparation
+ Model training
+ BPTT

Predicting with RNNs

+ Sequence generation
+ forecasting

18. Explain Encoder and decoder.

Ans: used for tasks like machine translation where the i/p & o/p sequence have different lenghts.

Encoder:

+ i/p encoding
+ Context vector

Decoder:

+ Initial state
+ o/p generation
+ training

Inference

+ Greedy decoding
+ Beam search.