Keras Tuner is a library that **helps you pick the optimal set of hyperparameters for your TensorFlow program.**

[**https://www.tensorflow.org/tutorials/keras/keras\_tuner**](https://www.tensorflow.org/tutorials/keras/keras_tuner)

import keras\_tuner as kt

Hyperparameters are of two types:

1. **Model hyperparameters eg number and width of hidden layers**
2. **Algorithm hyperparameters ; eg like learning rate for sgd**

## **Define the model: The model you set up for hypertuning is called a *hypermodel*.**

Two approaches : 1. By using a model builder function

2. By subclassing the **HyperModel** class of the keras Tuner API

For computer vission two pre defind approaches :- HyperModel and HyperResNet

def model\_builder(hp):

-----

-----

  # Tune the number of units in the first Dense layer  
  # Choose an optimal value between 32-512  
  hp\_units = hp.Int('units', min\_value=32, max\_value=512, step=32)

-----

-----

  # Tune the learning rate for the optimizer  
  # Choose an optimal value from 0.01, 0.001, or 0.0001  
  hp\_learning\_rate = hp.Choice('learning\_rate', values=[1e-2, 1e-3, 1e-4])

## **Instantiate the tuner and perform hypertuning**

Keras Tuner has four tuners available - RandomSearch, Hyperband, BayesianOptimization, and Sklearn

Hyperband tunner to optimize and maximum number of epoch to trian.

tuner = kt.Hyperband(model\_builder,  
                     objective='val\_accuracy',  
                     max\_epochs=10,  
                     factor=3,  
                     directory='my\_dir',  
                     project\_name='intro\_to\_kt')

**check internal performance for memory allocations.?**

stop\_early = tf.keras.callbacks.EarlyStopping(monitor='val\_loss', patience=5)

tuner.search(img\_train, label\_train, epochs=50, validation\_split=0.2, callbacks=[stop\_early])  
# Get the optimal hyperparameters  
best\_hps=tuner.get\_best\_hyperparameters(num\_trials=1)[0]

## **Train the model**

# Build the model with the optimal hyperparameters and train it on the data for 50 epochs  
model = tuner.hypermodel.build(best\_hps)  
history = model.fit(img\_train, label\_train, epochs=50, validation\_split=0.2)

best\_epoch = val\_acc\_per\_epoch.index(max(val\_acc\_per\_epoch)) + 1

Retrain the model

hypermodel = tuner.hypermodel.build(best\_hps)  
  
# Retrain the model  
hypermodel.fit(img\_train, label\_train, epochs=best\_epoch, validation\_split=0.2)

eval\_result = hypermodel.evaluate(img\_test, label\_test)

**Good eg to cover:**

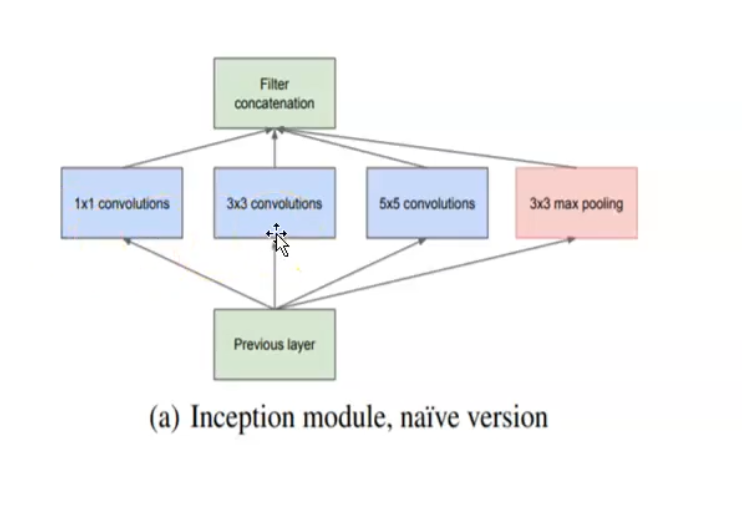
<https://www.tensorflow.org/tutorials/keras/overfit_and_underfit>

**Epoch** : iteration over entire dataset

**Batch Size** : we have to divide the dataset into number of batches.

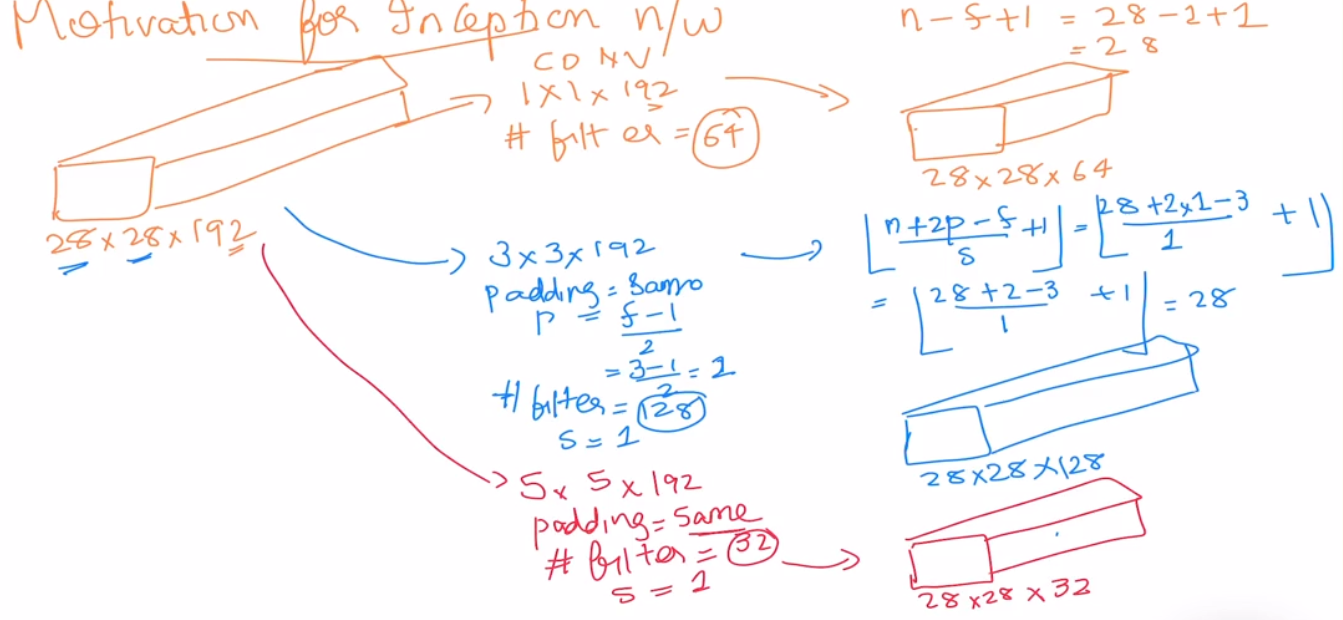
**Iteration**: if 1000 images and batch size 20 then 1000/20 = 50 iteration

**Model can be saved : .h5, .pkl, .pt, .tfjs**

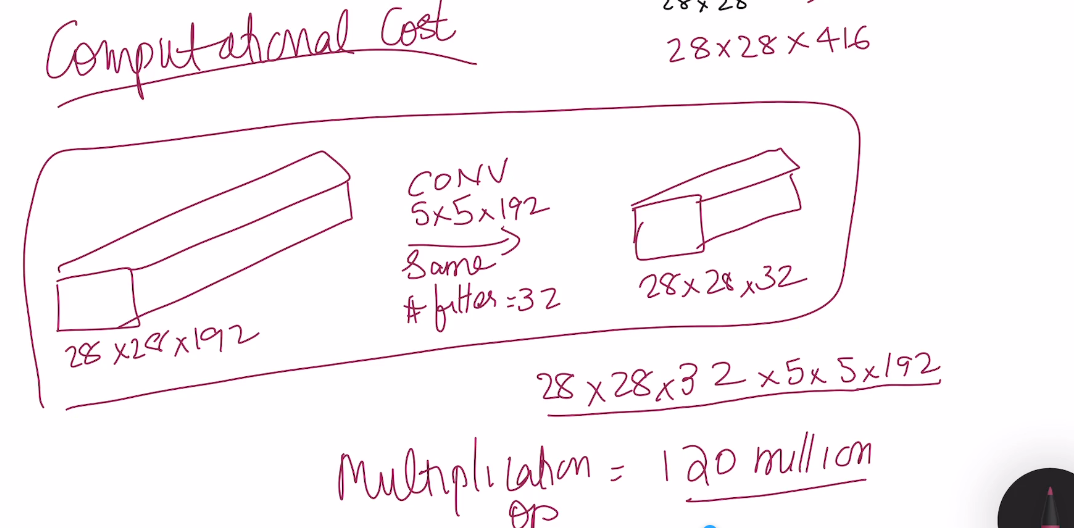
**Inception** : For bigger network it is difficult to set filter becasuse larger filter work better for the image with information distributed globally while for image with smaller portion we requireed smaller filter.

Padding = same

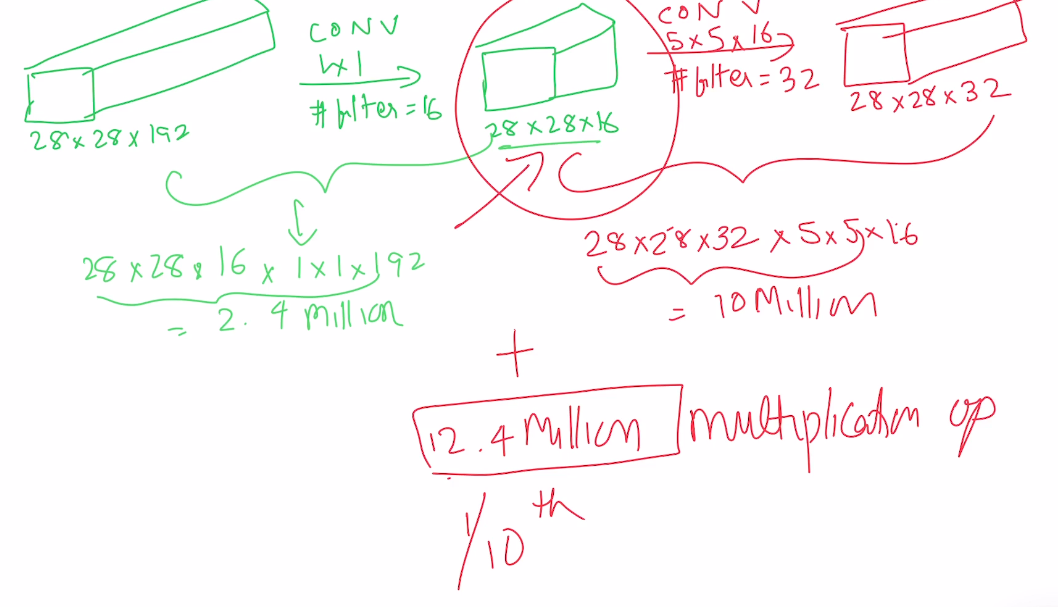
= (f-1)/2

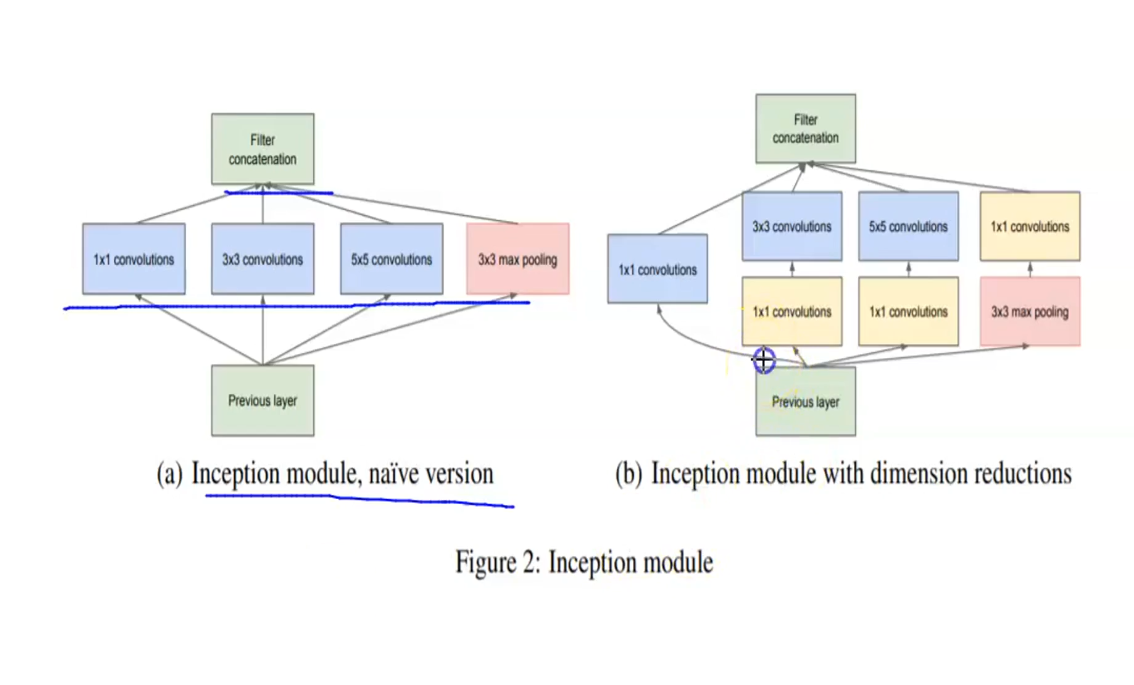


Computational cosat is very high:



Computational cosat is very low as compared to previous version:





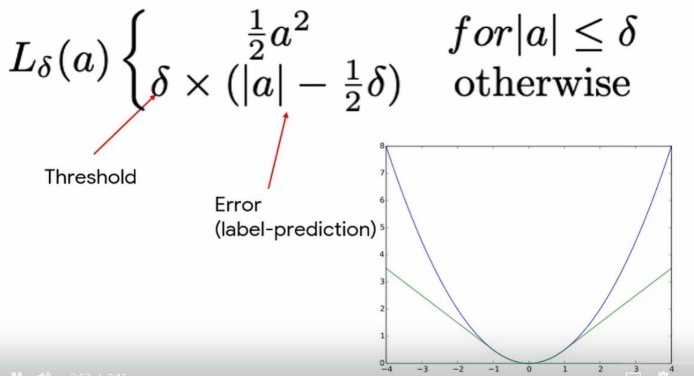
Train is 60000,28, 28

Keras take data in this format X\_train.reshape(60000,28,28,1)

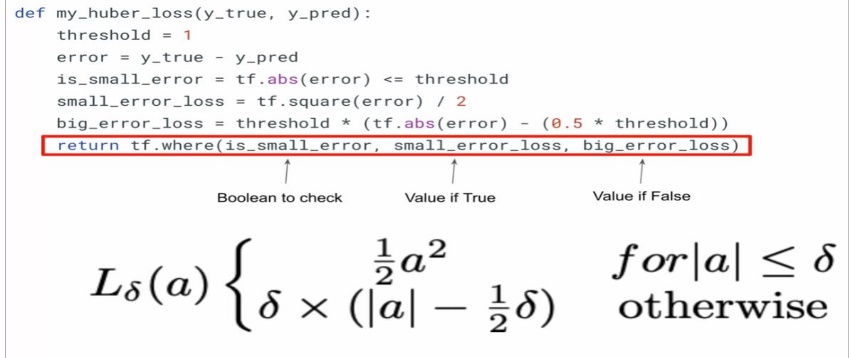
----------------------------

**Loss functions** help measure how well a model is doing

Huber Loss: The Huber loss function can be used **to balance between the Mean Absolute Error, or MAE, and the Mean Squared Error, MSE**. It is therefore a good loss function for when you have varied data or only a few outliers



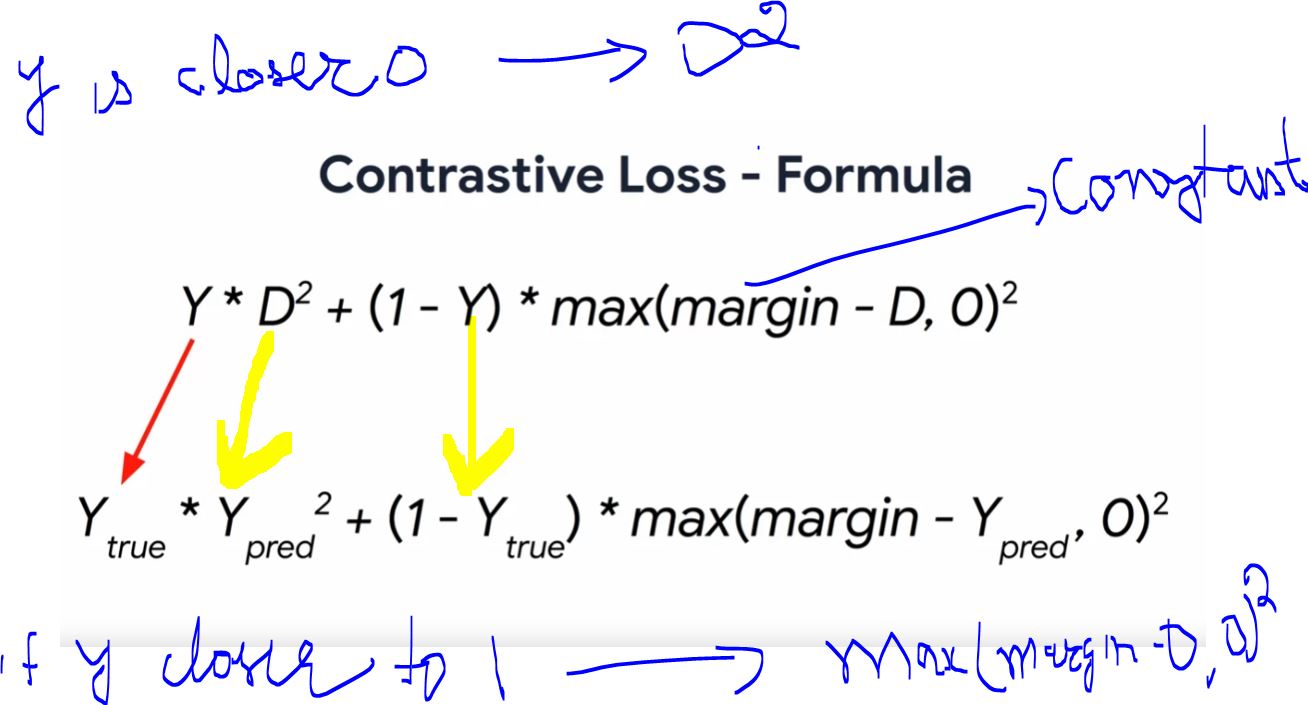
**Huber loss implemention:**



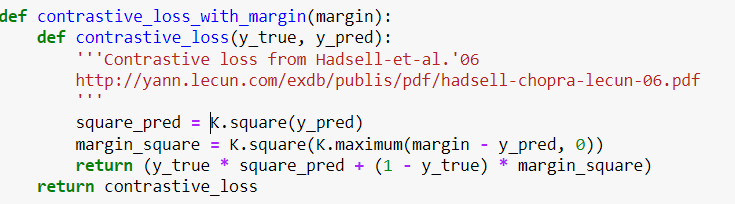
Class level implementation:-



**Contrastive loss:** Used to find similarity between two vactors. Contrastive loss, like triplet and magnet loss, is used **to map vectors that model the similarity of input items**



**Contrastive loss implementaions:**



# measure the similarity of the two vector outputs

output = Lambda(euclidean\_distance, name="output\_layer", output\_shape=eucl\_dist\_output\_shape)([vect\_output\_a, vect\_output\_b])

# specify the inputs and output of the model

model = Model([input\_a, input\_b], output)

**Custom Dense Layer** will contain weights that can be updated during training.

 Requires three functions: \_\_init\_\_(), build() and call()

class SimpleDense(Layer):

# add an activation parameter

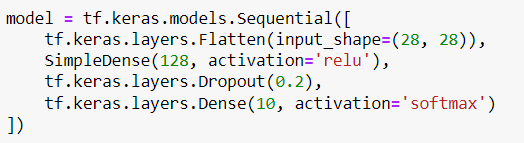
def \_\_init\_\_(self, units=32, activation=None):

def build(self, input\_shape):

def call(self, inputs):

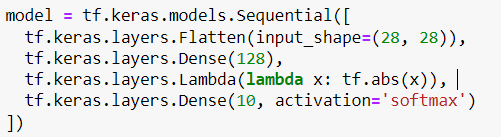
# pass the computation to the activation layer

return self.activation(tf.matmul(inputs, self.w) + self.b)



<https://www.coursera.org/learn/custom-models-layers-loss-functions-with-tensorflow/ungradedLab/laUcE/custom-dense-layer/lab?path=%2Fnotebooks%2FC1_W3_Lab_2_custom-dense-layer.ipynb%23Ungraded-Lab%3A-Building-a-Custom-Dense-Layer>

**Lambda layer** define a custom function that the Lambda layer will call



<https://www.coursera.org/learn/custom-models-layers-loss-functions-with-tensorflow/ungradedLab/AFDgQ/lambda-layer/lab?path=%2Fnotebooks%2FC1_W3_Lab_1_lambda-layer.ipynb>

**Test Implementation of calls from Utils layers:-**

import utils

utils.test\_simple\_quadratic(SimpleQuadratic)

# Coding a Wide and Deep Model

# inherit from the Model base class

class WideAndDeepModel(Model):

def \_\_init\_\_(self, units=30, activation='relu', \*\*kwargs): # Initialize the instance attributes.

def call(self, inputs): ##build the network and return the output layers

# create an instance of the model

model = WideAndDeepModel()

<https://www.coursera.org/learn/custom-models-layers-loss-functions-with-tensorflow/ungradedLab/JB1Zr/build-a-basic-model/lab?path=%2Fnotebooks%2FC1_W4_Lab_1_basic-model.ipynb>

**Residual Networks** make use of skip connections to make deep models easier to train.

## Implement Model subclasses

class IdentityBlock(tf.keras.Model):

def \_\_init\_\_(self, filters, kernel\_size):

super(IdentityBlock, self).\_\_init\_\_(name='')

def call(self, input\_tensor):

<https://www.coursera.org/learn/custom-models-layers-loss-functions-with-tensorflow/ungradedLab/hStfq/build-a-resnet-model/lab?path=%2Fnotebooks%2FC1_W4_Lab_2_resnet-example.ipynb>

Sequential and Functional APIs have their limitations?

**Basic Tensor and Python points:-**

**\_\_dict\_\_ is a Python dictionary that contains the object's instance variables and values as key value pairs.**

If you call vars() and pass in an object, it will call the object's \_\_dict\_\_ attribute

# Format a string using f-string notation

i=1

print(f"var{i}")

# Format a string using .format notation

i=2

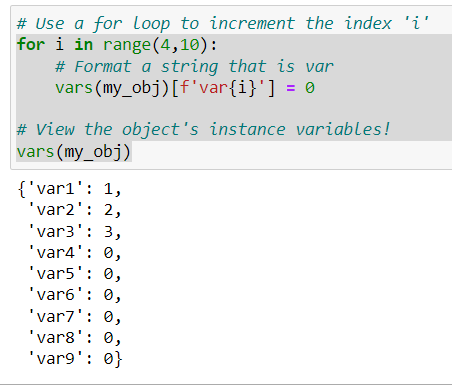
print("var{}".format(i))

**my\_obj = MyClass()**

my\_obj = MyClass()

my\_obj.\_\_dict\_\_ # both will give same output

vars(my\_obj) # both will give same output



Important : vars can be shared across the different methods in the pyhton.

<https://www.coursera.org/learn/custom-distributed-training-with-tensorflow/ungradedLab/mDT5h/basic-tensors/lab>

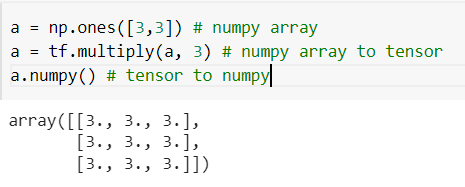
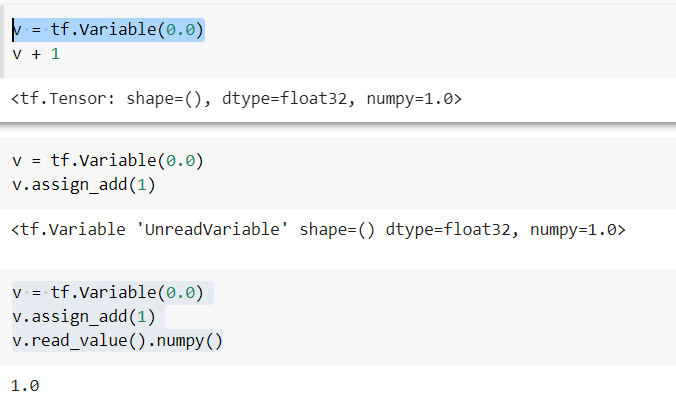
a = tf.constant([[1 , 2],

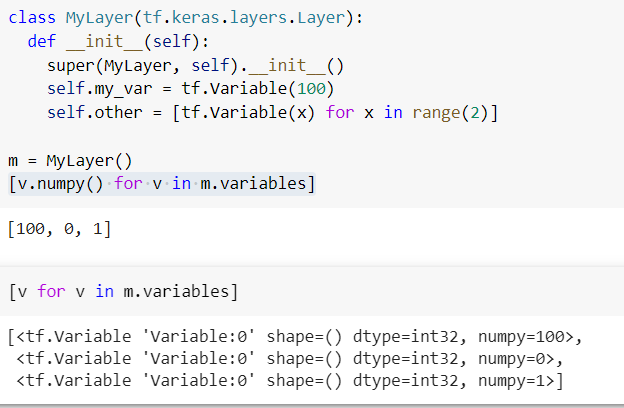
[3, 4]])

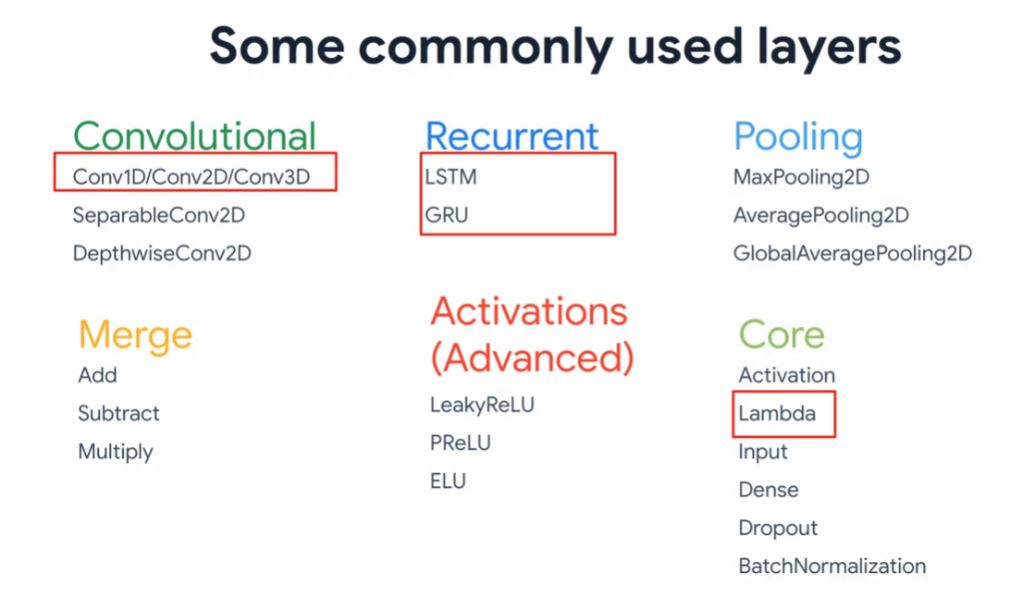
tf.add(a, 1)

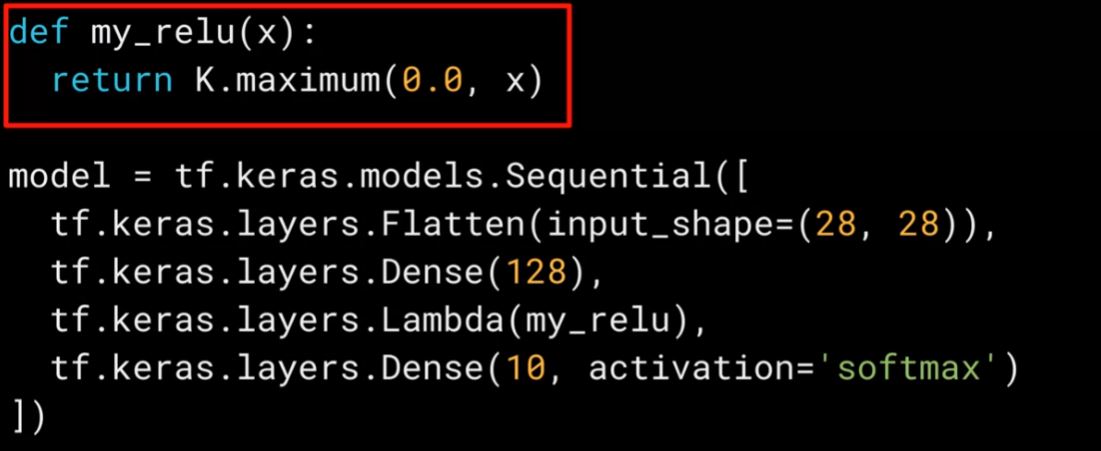
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=array([[2, 3], [4, 5]], dtype=int32)>

Numpy interoperability







Example for implementing multiple loss and metrics:

model.compile(optimizer=rms,

loss = {'wine\_type' : 'binary\_crossentropy',

'wine\_quality' : 'mean\_squared\_error'

},

metrics = {'wine\_type' : 'accuracy',

'wine\_quality': tf.keras.metrics.RootMeanSquaredError() } )

**VGG-16** : 16 layers, 3 \* 3 kernel, 2 \* 2 max pooling size.

ImageDataGenerator class supports a number of pixel scaling methods, as well as a range of data augmentation techniques.

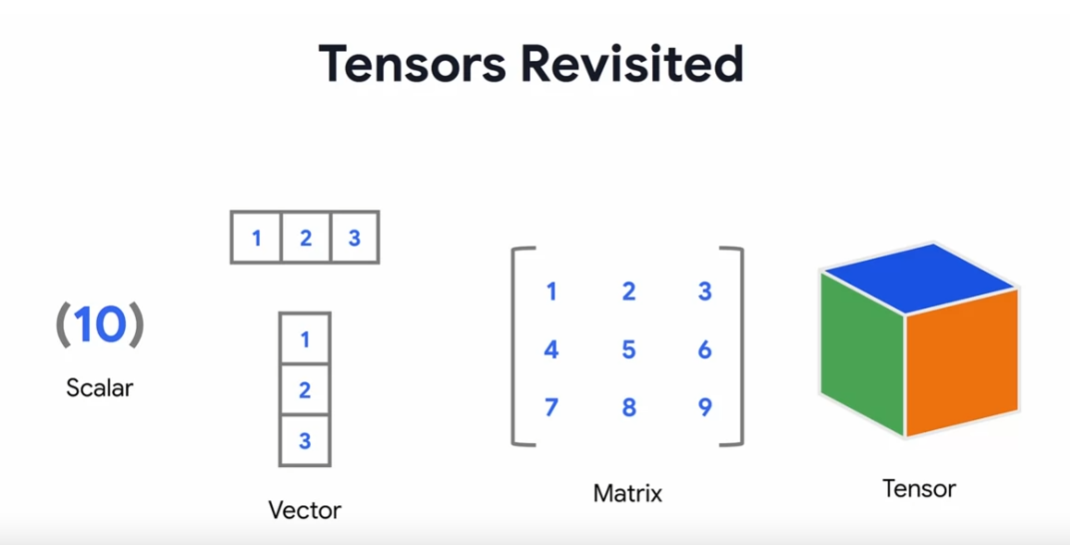
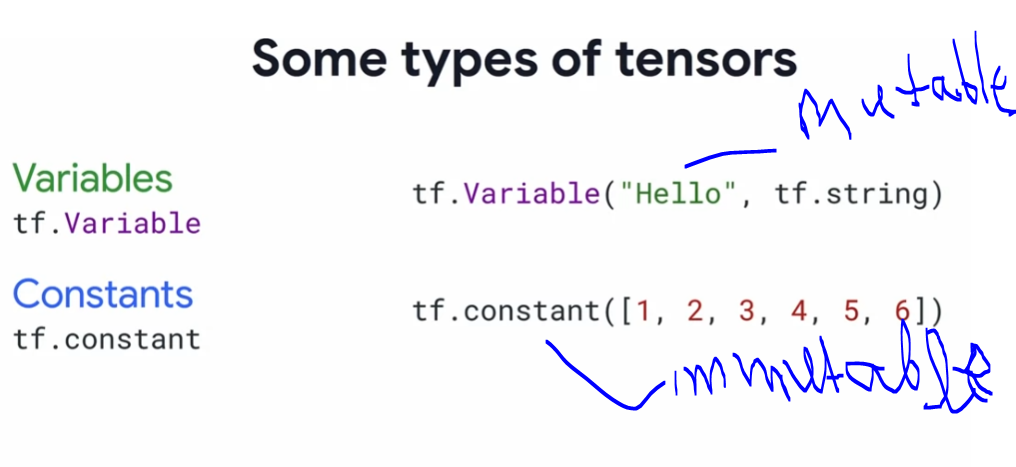
ModelCheckpoint : model will be saved to disk only if the validation accuracy of the model in current epoch is greater than what it was in the last epoch.

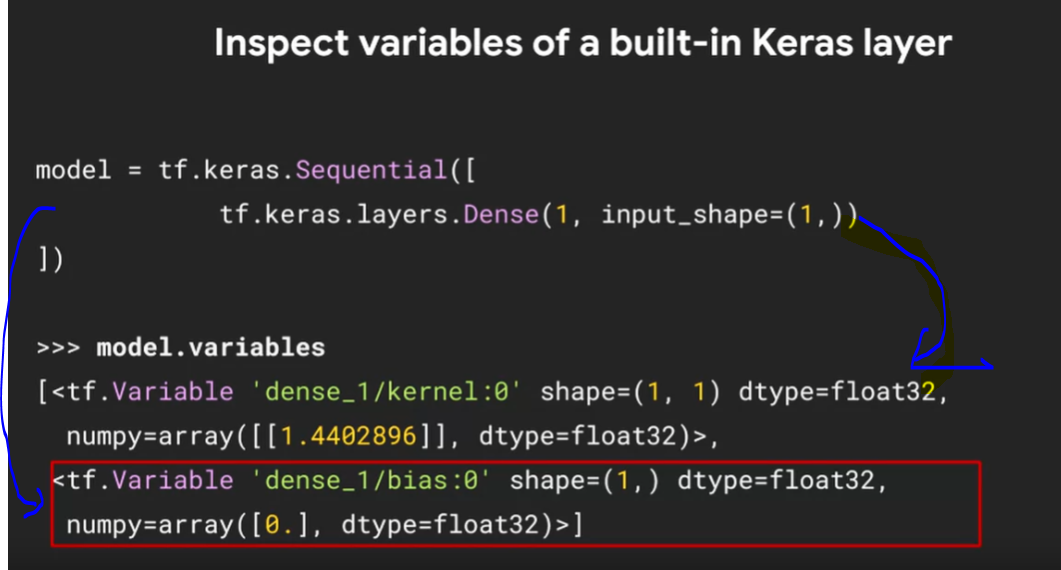
Early Stopping : set patience to 5 which means that the model will stop to train if it doesn’t see any rise in Validation accuracy in 5 consecutive epochs.

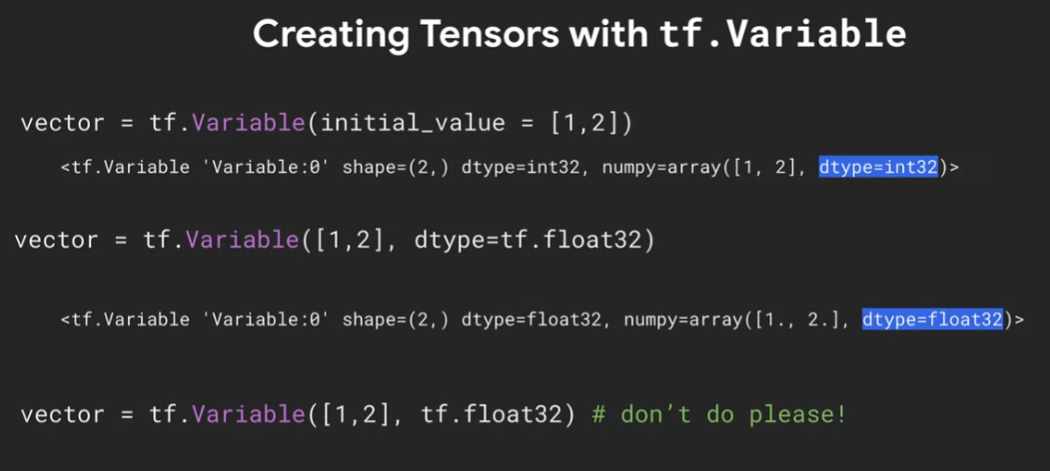
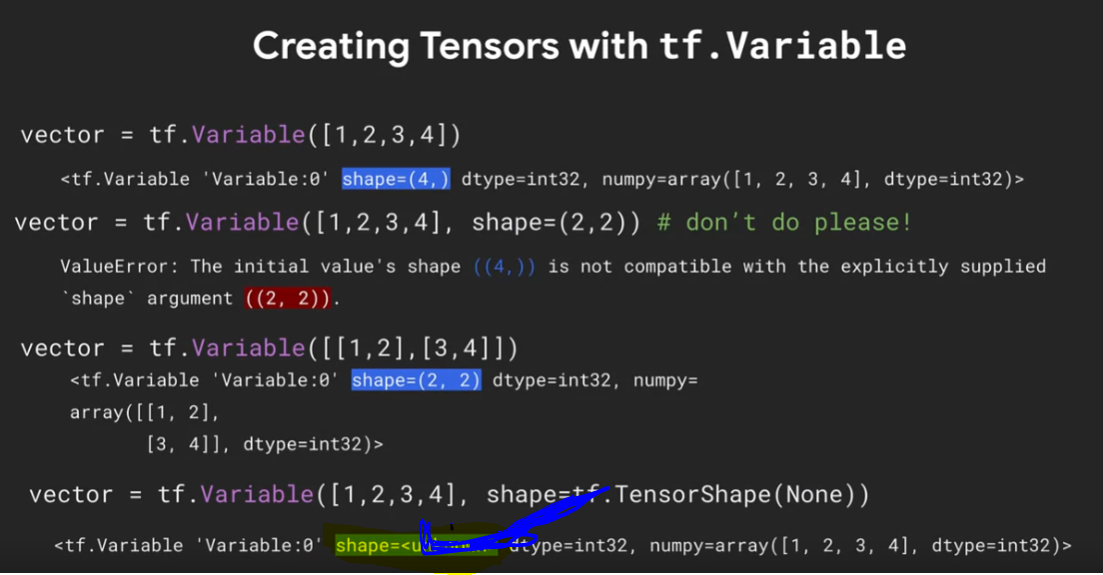
**Eager vs Graph based Execution:**

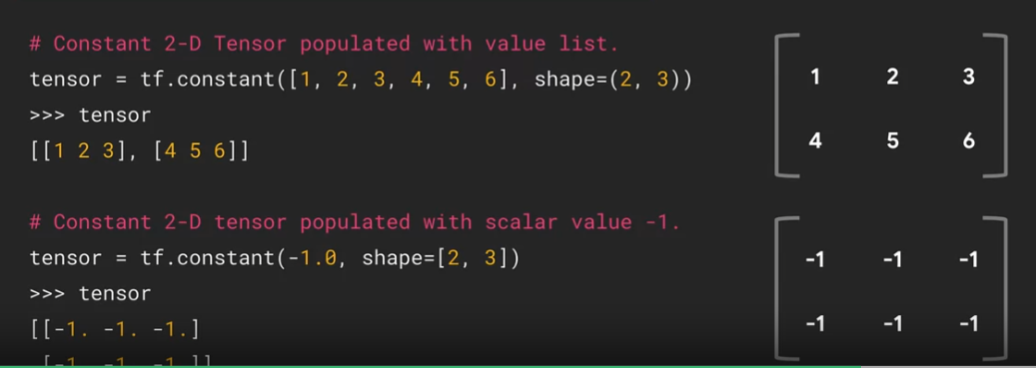
Difference between  graph-based execution and eager execution in TensorFlow?

eager is user friendly



## An overview of callback methods:

tf.keras.callbacks.Callback – call back main class

Common methods for training/testing/predicting

For training, testing, and predicting, following methods are provided to be overridden.

### on\_(train|test|predict)\_begin(self, logs=None)

### on\_(train|test|predict)\_end(self, logs=None)

### on\_(train|test|predict)\_batch\_begin(self, batch, logs=None)

### on\_(train|test|predict)\_batch\_end(self, batch, logs=None)

### Training specific methods

on\_epoch\_begin(self, epoch, logs=None)

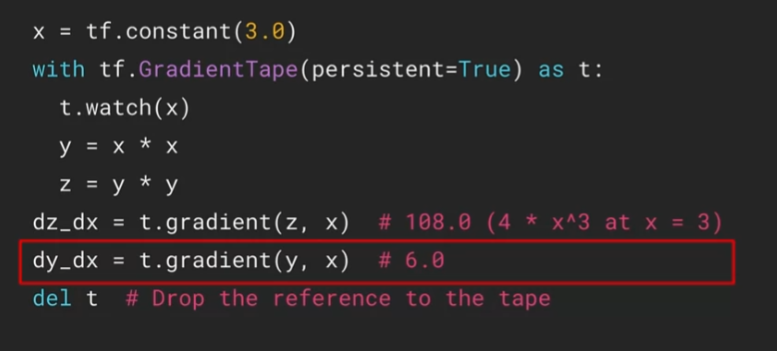
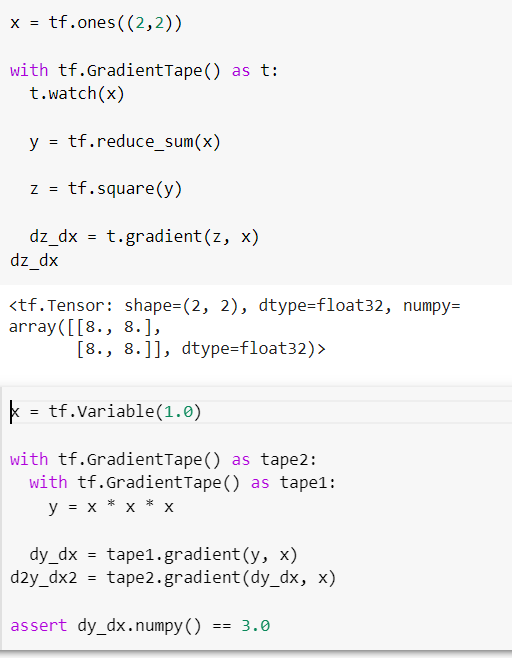
on\_epoch\_end(self, epoch, logs=None)

https://www.coursera.org/learn/custom-models-layers-loss-functions-with-tensorflow/ungradedLab/UaFHl/custom-callbacks/lab?path=%2Fnotebooks%2FC1\_W5\_Lab\_2\_custom-callbacks.ipynb

**Eager mode** : One type ovpaf mode in TensorFlow allows for immediate evaluation of values

**Gradient Tape**: Intensive flow optimizers are implemented using TensorFlow **automatic differentiation API call** **Gradient Tape**

**Basic gradient eg:- Basic gradient presistant = true eg:-**



<https://www.coursera.org/learn/custom-distributed-training-with-tensorflow/ungradedLab/jKn7w/gradient-tape-basics/lab?path=%2Fnotebooks%2FC2_W1_Lab_2_gradient-tape-basics.ipynb>

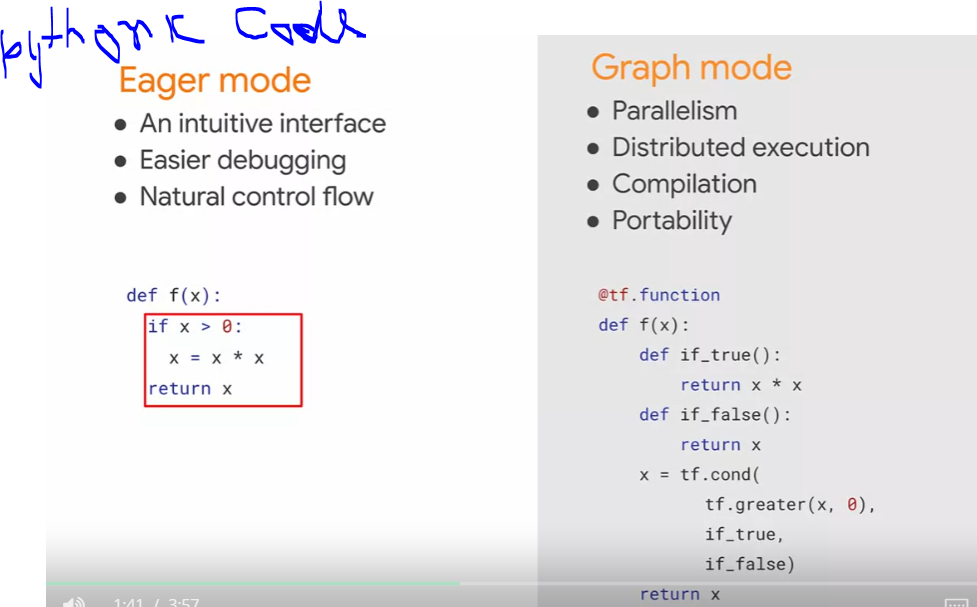
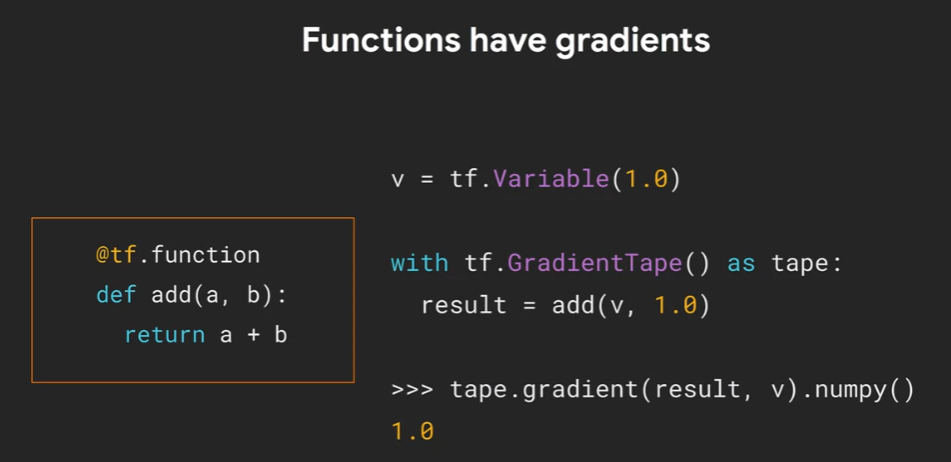
<https://www.coursera.org/learn/custom-distributed-training-with-tensorflow/programming/TwjF2/basic-tensor-operations/lab?path=%2Fnotebooks%2FC2W1_Assignment.ipynb>

<https://www.youtube.com/watch?v=TudQZtgpoHk&t=1409s>

<https://www.geeksforgeeks.org/optimizers-in-tensorflow/#:~:text=Optimizers%20are%20techniques%20or%20algorithms,better%20accuracy%20of%20model%20faster>

**Graph mode:**

* How to Use decorators and tf.autograph to convert code into graph based code

Function have gradients also work in graph mode.

@tf.function ## 1st step

def f(x,y): -----

print(f(1.0, 2.0))

print(tf.autograph.to\_code(f.python\_function))## 2nd step

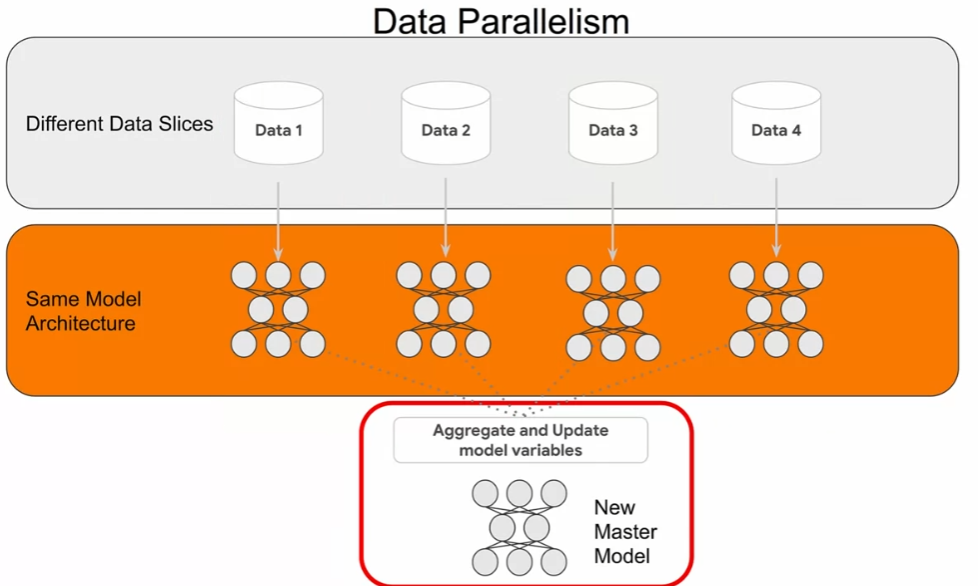
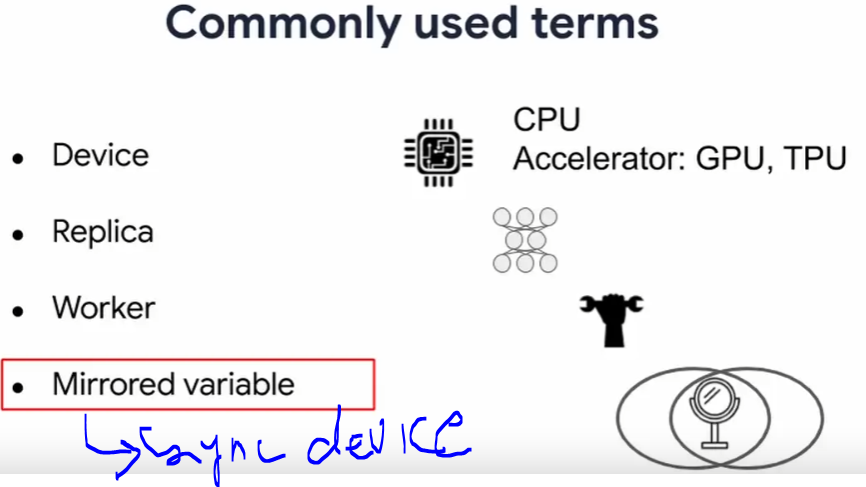
http://wiki.c2.com/?FizzBuzzTest

**Week 4:-**

how distributed training is different from regular model training?

Mirrored Strategy to train a model on multiple GPUs on the same device?

TPU Strategy to train on multiple cores of a TPU?

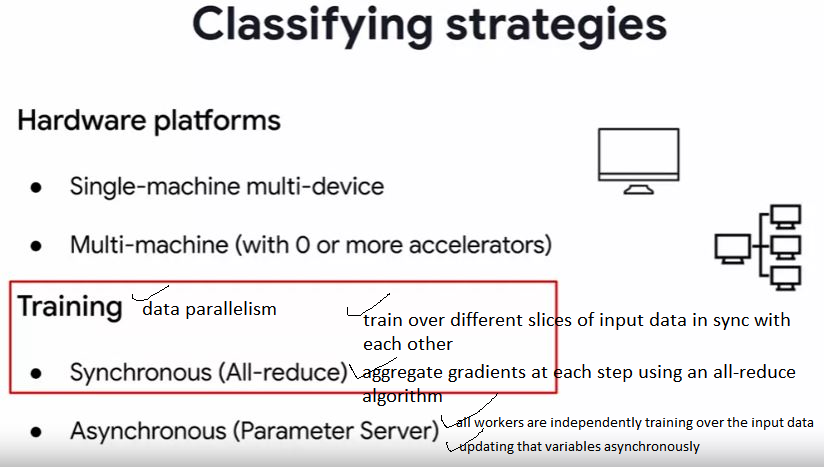
 

**Mirored variables** which we want to keep in sync across the different worker.

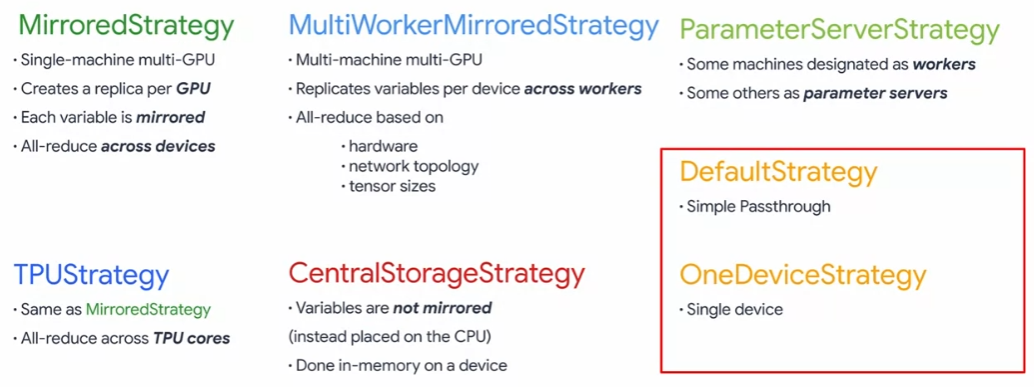
**Replica** : Copies of the models variables are placed on several devices

**Distribution stratergies:-**

* Tf.distribute.strategy
* High levels APIs
* Custom training loops
* Tensorflow 2: eager mode and graph mode
* Supported on multiple configurations
* Convenient to use with little to no code changes



**Async training** : synchronize the distributed model through something called a parameter server architecture



Mirored stratergy:-

* Create a model replica on each GPU.
* Mirror the variables.
* Parameters are merged using an all-reduce across each of the devices.

**MultiWorkerMirroredStrategy:**

* Same as Mirored stratergy
* it replicates and mirrors across each worker instead of each GPU device
* Then it uses an all-reduce algorithm based on the hardware setup

**MirroredStrategy :-**

**tf.distribute.MirroredStrategy**

Synchronous training across multiple replicas on one machine.

**Parent class** : Strategy

**1st part:**

# Define the strategy to use and print the number of devices found

strategy = tf.distribute.MirroredStrategy() # stratergy object contains properties like number of replicas in sync

**2nd part:**

# Use for Mirrored Strategy

BATCH\_SIZE = BATCH\_SIZE\_PER\_REPLICA \* strategy.num\_replicas\_in\_sync

# Set up the train and eval data set

train\_dataset = mnist\_train.map(scale).cache().shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE)

eval\_dataset = mnist\_test.map(scale).batch(BATCH\_SIZE)

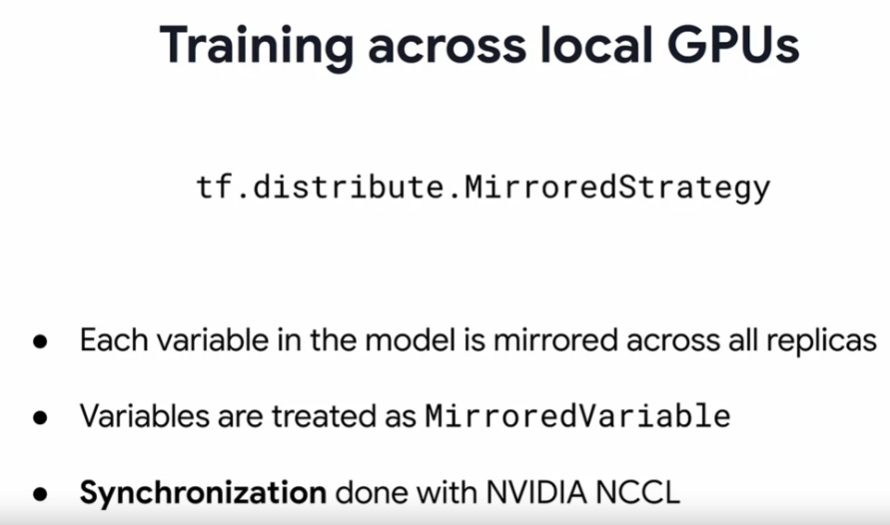
**3rd part:**

# Use for Mirrored Strategy -- comment out `with strategy.scope():` and deindent for no strategy

with strategy.scope():

model = tf.keras.Sequential([

This strategy is typically used for training on one machine with multiple GPUs.



<https://www.tensorflow.org/api_docs/python/tf/distribute/MirroredStrategy>

<https://www.coursera.org/learn/custom-distributed-training-with-tensorflow/ungradedLab/alLiN/mirrored-strategy/lab?path=%2Fnotebooks%2FC2_W4_Lab_1_basic-mirrored-strategy.ipynb>

**tf.function:** Compiles a function into a callable TensorFlow graph. (deprecated arguments) (deprecated arguments)

## **Multiple GPU Mirrored Strategy:**

create a model with Custom Training?

how it splits the data training across GPUs?

how the aggregation of lost data is then managed?

Steps :

Create datsets form the batches

Create distributed datasets from the datasets

Loop custom distributed training set batch by batch

Pass batch to distributed train steps to calculate loss and gradients

Distributed train step call to train steps with stratergy.run() and reduce sum all the loss.

train steps executed parallel with help of gradient tape and calculate the loss and prediction

code steps :-

## Setup Distribution Strategy

# less, you need to set this. In this case one of my GPUs has 4 cores

**os.environ["TF\_MIN\_GPU\_MULTIPROCESSOR\_COUNT"] = "4"**

# If you have \*different\* GPUs in your system, you probably have to set up cross\_device\_ops like this

**strategy = tf.distribute.MirroredStrategy(cross\_device\_ops=tf.distribute.HierarchicalCopyAllReduce())**

# Batch the input data

BUFFER\_SIZE = len(train\_images)

BATCH\_SIZE\_PER\_REPLICA = 64

**GLOBAL\_BATCH\_SIZE = BATCH\_SIZE\_PER\_REPLICA \* strategy.num\_replicas\_in\_sync**

# Create Datasets from the batches

**train\_dataset = tf.data.Dataset.from\_tensor\_slices((train\_images, train\_labels)).shuffle(BUFFER\_SIZE).batch(GLOBAL\_BATCH\_SIZE)**

**test\_dataset = tf.data.Dataset.from\_tensor\_slices((test\_images, test\_labels)).batch(GLOBAL\_BATCH\_SIZE)**

# Create Distributed Datasets from the datasets

**train\_dist\_dataset = strategy.experimental\_distribute\_dataset(train\_dataset)**

**test\_dist\_dataset = strategy.experimental\_distribute\_dataset(test\_dataset)**

<https://www.coursera.org/learn/custom-distributed-training-with-tensorflow/lecture/EDiRd/custom-training-for-multiple-gpu-mirrored-strategy>

**Other Distributed Strategies:**

tf.distribute.OneDeviceStratergy 🡪 input data is distributed

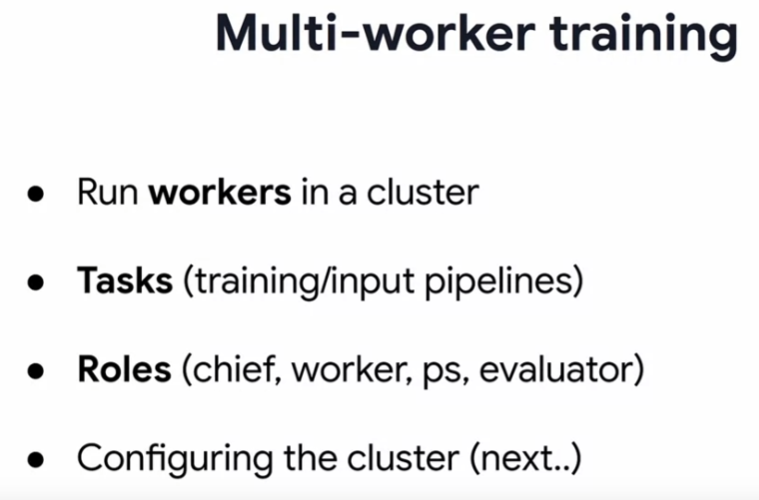
when we deliberately wish to perform training on one specific device of your choice.

**Mirrored strategy** could be used where training was done across multiple GPUs in a single machine

**MultiWorkerMirroredStrategy**: there are multiple GPUs on multiple machines and each machine can have a different number of GPUs on them.

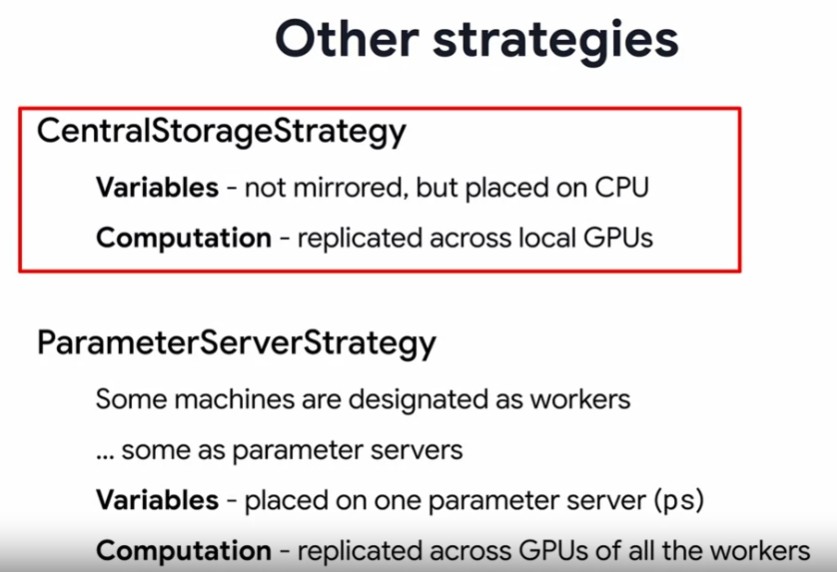
* **Fault tolerance** is vital and this can be achieved using checkpoints
* Synchronization and reduction is also more complex and it is performed using something called **CollectiveOps.**

These above apis are responsiable for making everything in sync.

Worker:-

* A worker is a piece of code that runs a task such as managing training or input pipelines.
* A worker is given a particular role such as a chief or an evaluator
* Usually all responsibility for workers are mentioned in Clustor specification files with JSON.

<https://www.tensorflow.org/tutorials/distribute/multi_worker_with_keras>