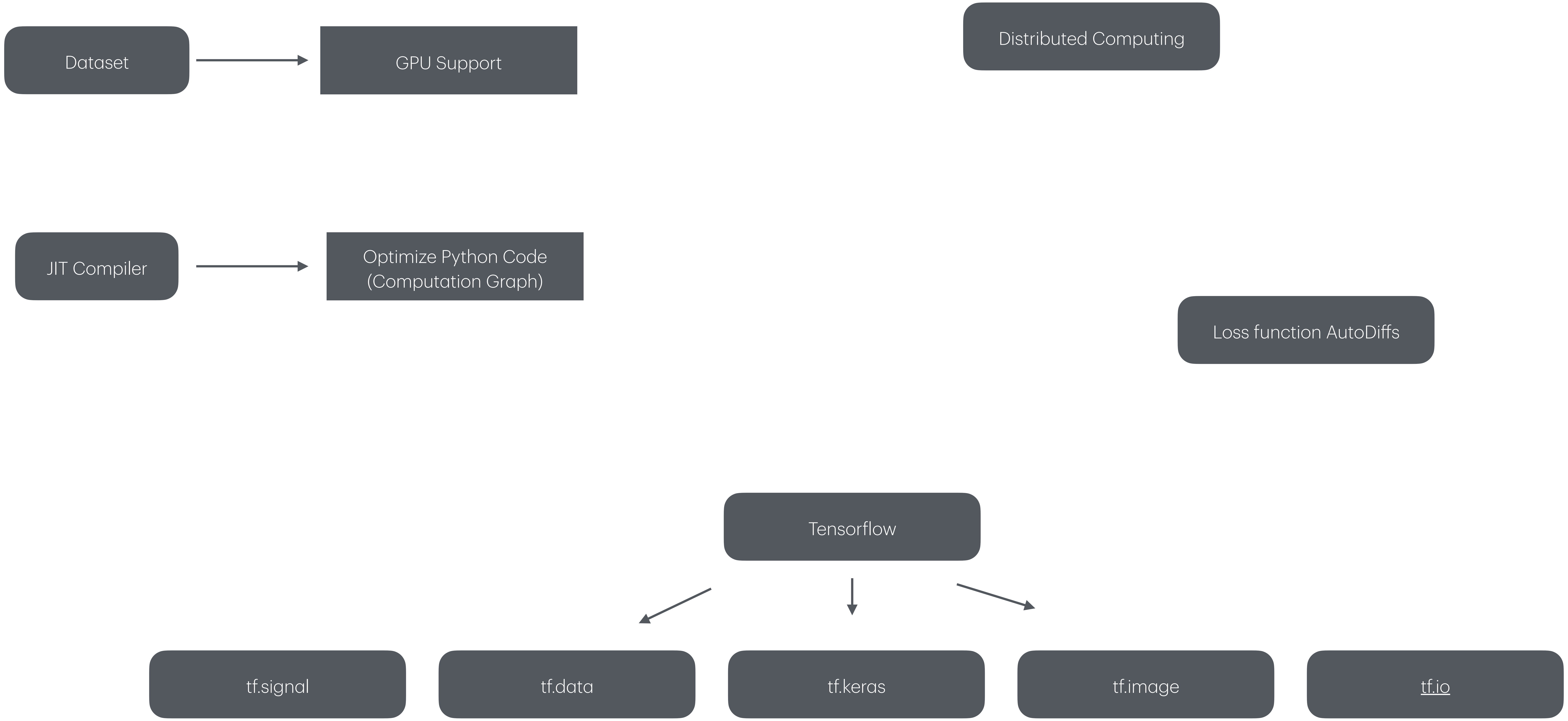
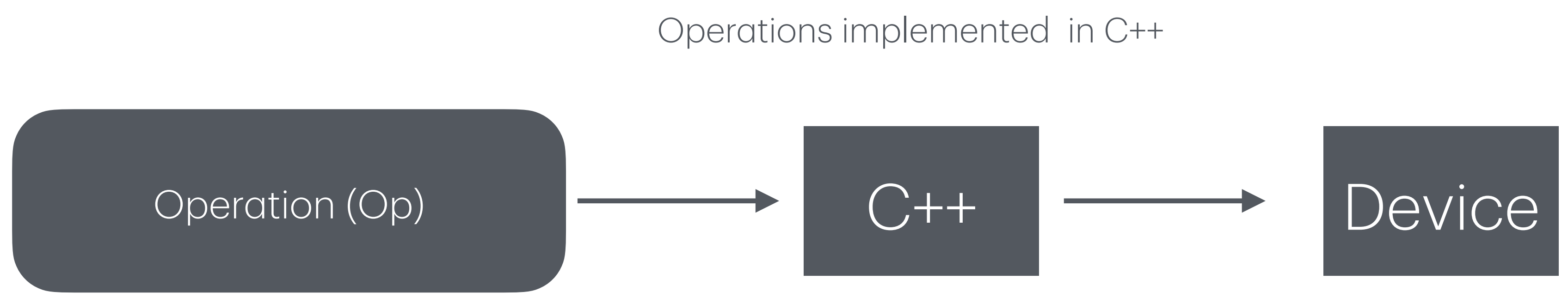
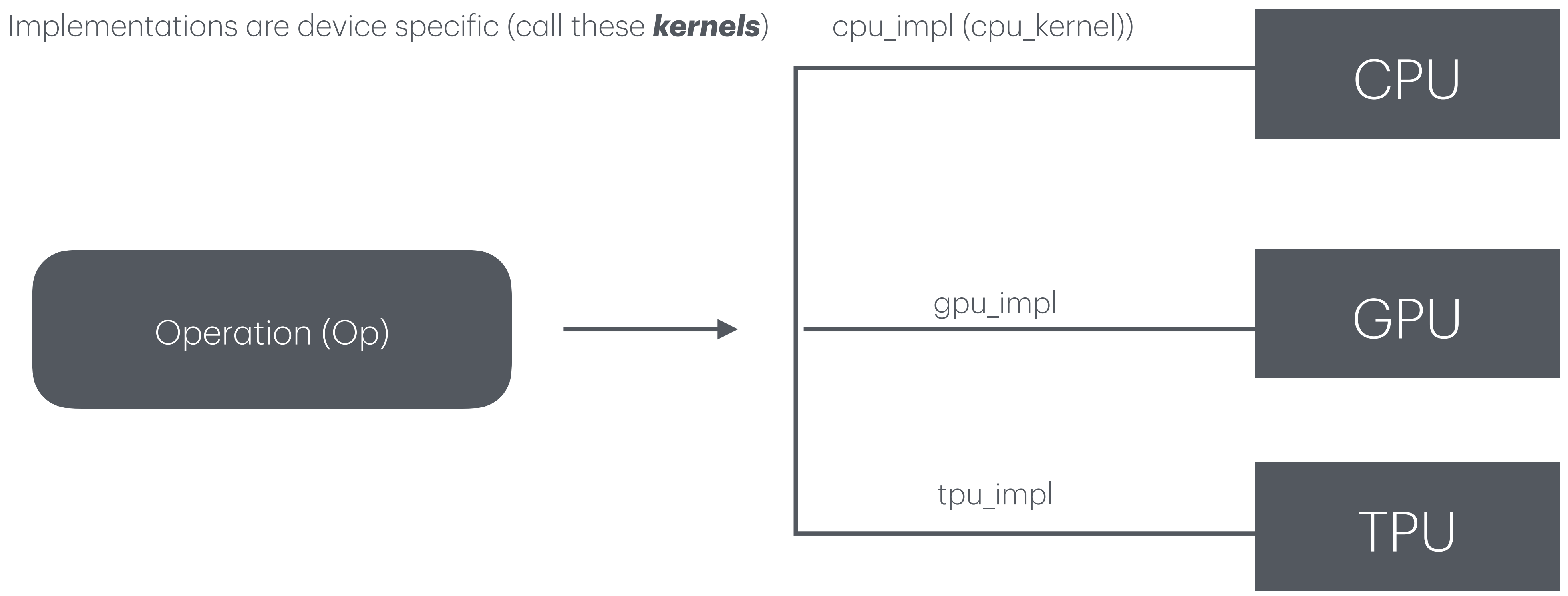
Customize with Tensorflow

Tensorflow

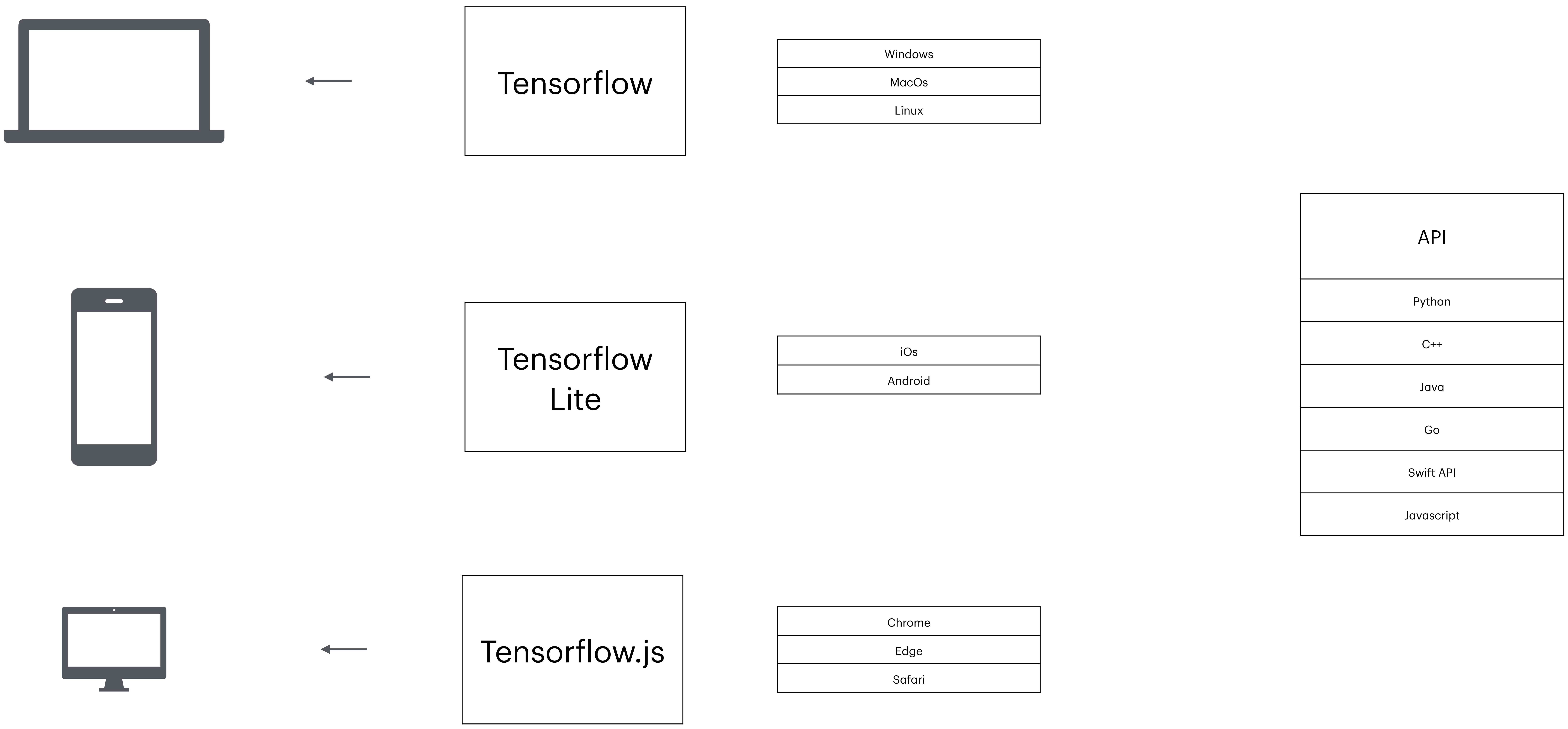

Dataset
GPU Support
Distributed Computing
JIT Compiler
Optimize Python Code
(Computation Graph)
Loss function AutoDiffs
Tensorflow
tf.signal
tf.data
tf.keras
tf.image
tf.io

Tensorflow


Operations implemented in C++
Operation (Op)
C++
Device


Implementations are device specific (call these ***kernels***)
cpu_impl (cpu_kernel))
CPU
Operation (Op)
gpu_impl
GPU
tpu_impl
TPU

Tensorflow


Tensorflow
Windows
MacOs
Linux
Tensorflow Lite
iOs
Android
Tensorflow.js
Chrome
Edge
Safari
API
Python
C++
Java
Go
Swift API
Javascript

Tensorflow

Tensorflow Extended

- Production Projects
  - Data Validation
  - Preprocessing
  - Model Analysis
  - Serving (TF Serving)

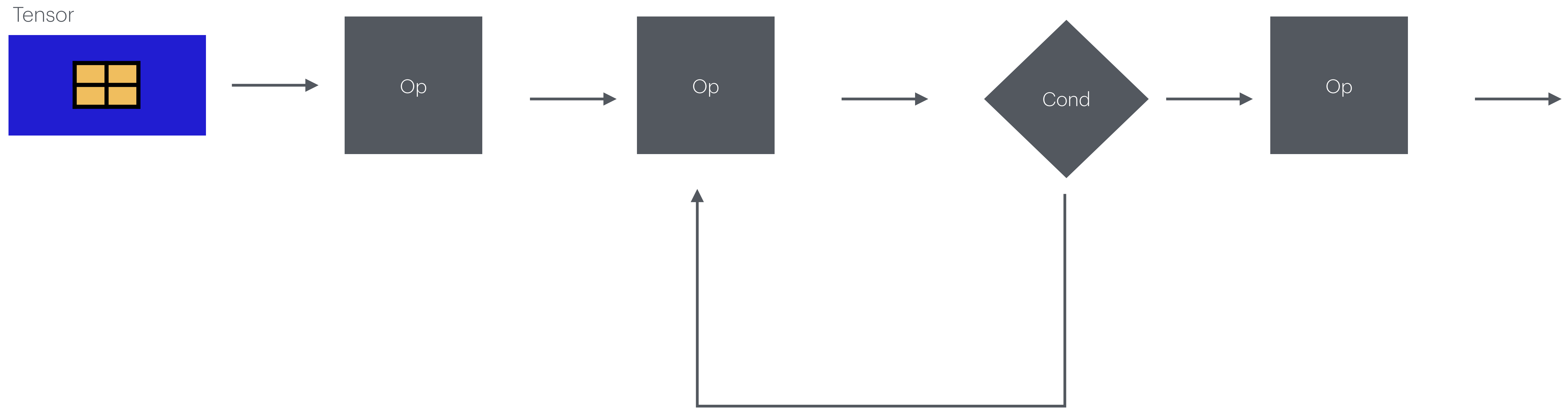
Core

TensorBoard

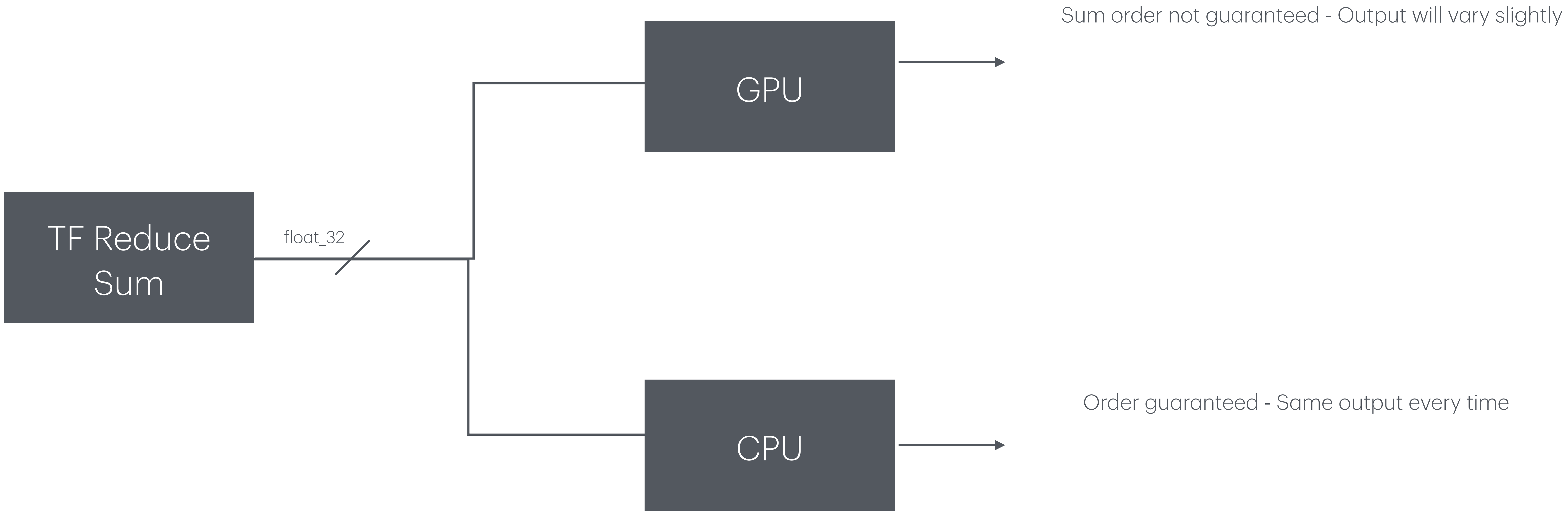
Tensorflow Hub

- Download/Reuse pertained NN

Tensorflow



Note on Kernels

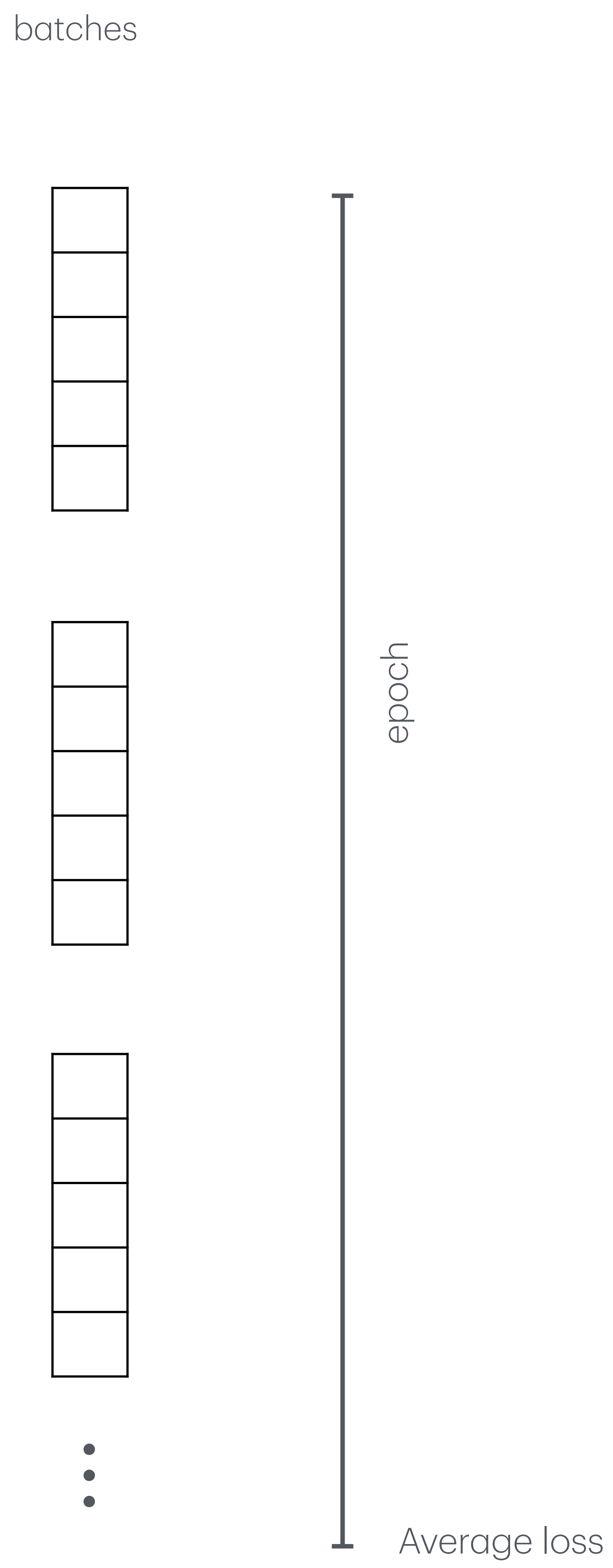


Write Custom Code

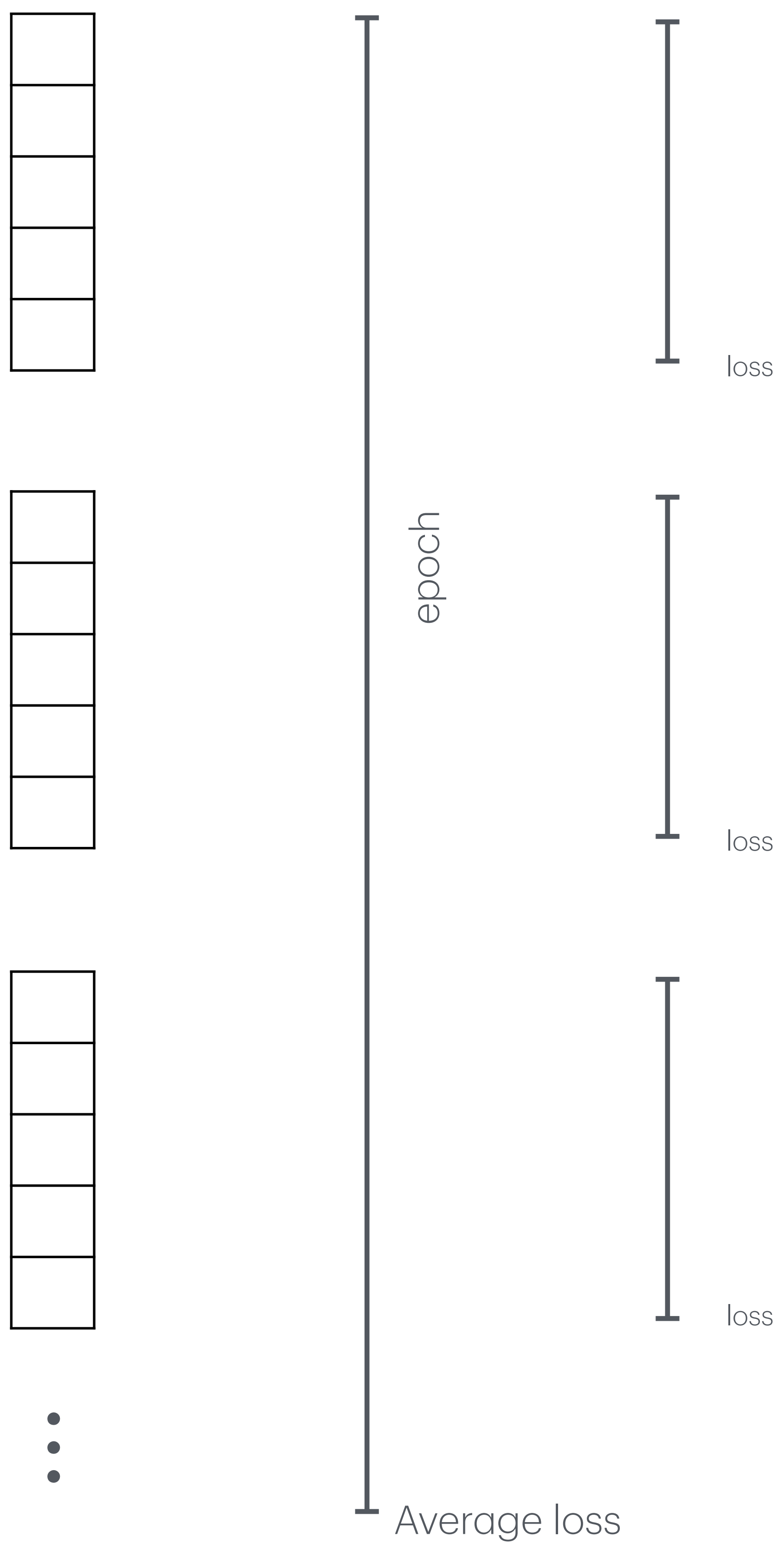
Do not use TensorFlow
API when creating
portable code

Custom Metrics

Metrics usually captured at the end of each epoch



There are scenarios when **streaming metrics** are required (metrics after each batch)

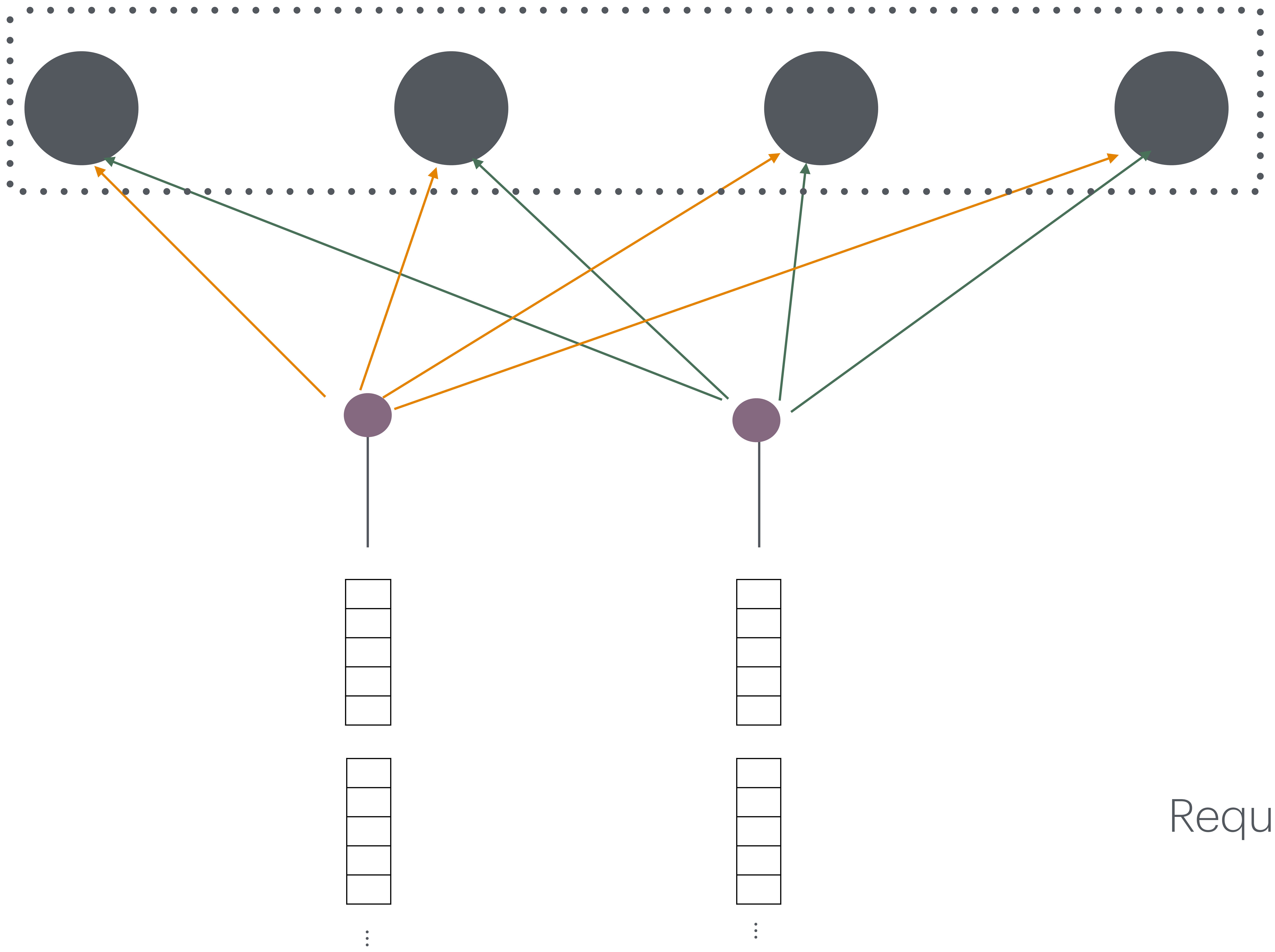


Custom Metrics

```python
class CustomMetric(tf.keras.Metric):
  def __init__(self, custom_hyperparam=100, **kwargs):
    super().__init__(**kwargs)
    self.custom_hyperparam = custom_hyperparam
    # add instance parameters dependent on metric
  def update_state(self, y_true, y_pred, sample_weight=None, **kwargs):
    pass
    # handle processing of batch predictions and labels (update instance parameters)
  def result(self):
    pass
    # computes the final result returned after at the end of model.fit(..)
  def get_config(self):
    base_config = super().get_config()
    return {**base_config, "custom_hyperparam": self.custom_hyperparam} # ensures custom hyperparameters are saved with model
```

```python
model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(), optimizer=tf.keras.optimizers.Nadam(lr) , metrics=[CustomMetric()])
```

Custom Layer



Requires Layer Subclassing

Custom Layer

$X_{batch} 5x2$

$Kernel 2x4$

$Bias 1x4$

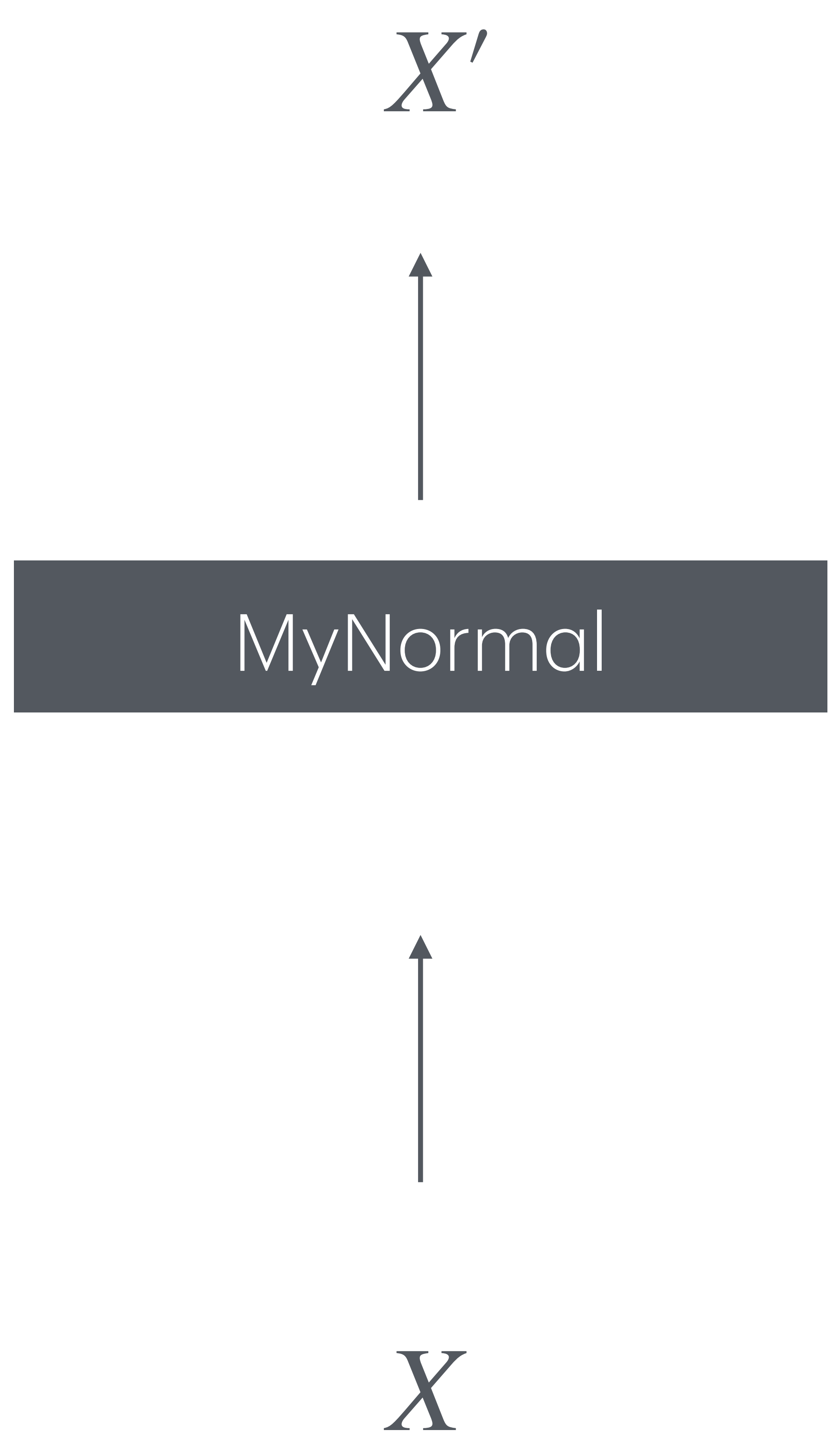
Custom Layer

```python
class MyDenseLayer(tf.keras.Layer):
  def __init__(self, units_hyp, activation_hyp=None, **kwargs):
    super().__init__(**kwargs)
    self.units = units_hyp
    self.activation = tf.keras.activations.get(activation_hyp)
  def build(self, batch_input_shape):
    # batch_input_shape = [5, 2]
    self.kernel = self.add_weight(
      name='kernel',
      shape=[ batch_input_shape[-1] , self.units ] # [ 2 , 4 ]
    )
    self.bias = self.add_weight(
      name='bias',
      shape=[self.units], # [ 4 ]
      initializer=tf.keras.initializers.Zeros()
    )
    super().build(batch_input_shape)
  def call(self, X):
    # output of layer
    return self.activation(X @ self.kernel + self.bias)
  def compute_output_shape(self, batch_input_shape):
    # output shape (batch included)
    return tf.TensorShape( tf.shape(batch_input_shape[:-1]).numpy().tolist()  + [ self.units ] )
  def get_config(self):

    base_config = super().get_config()
    return {**base_config, "units": self.units, "activation": self.activation} # ensures custom hyperparameters are saved with model
```

```python
MyDenseLayer(units_hyp=4, activation_hyp=tf.keras.activations.relu)
```

Custom Layer - 2



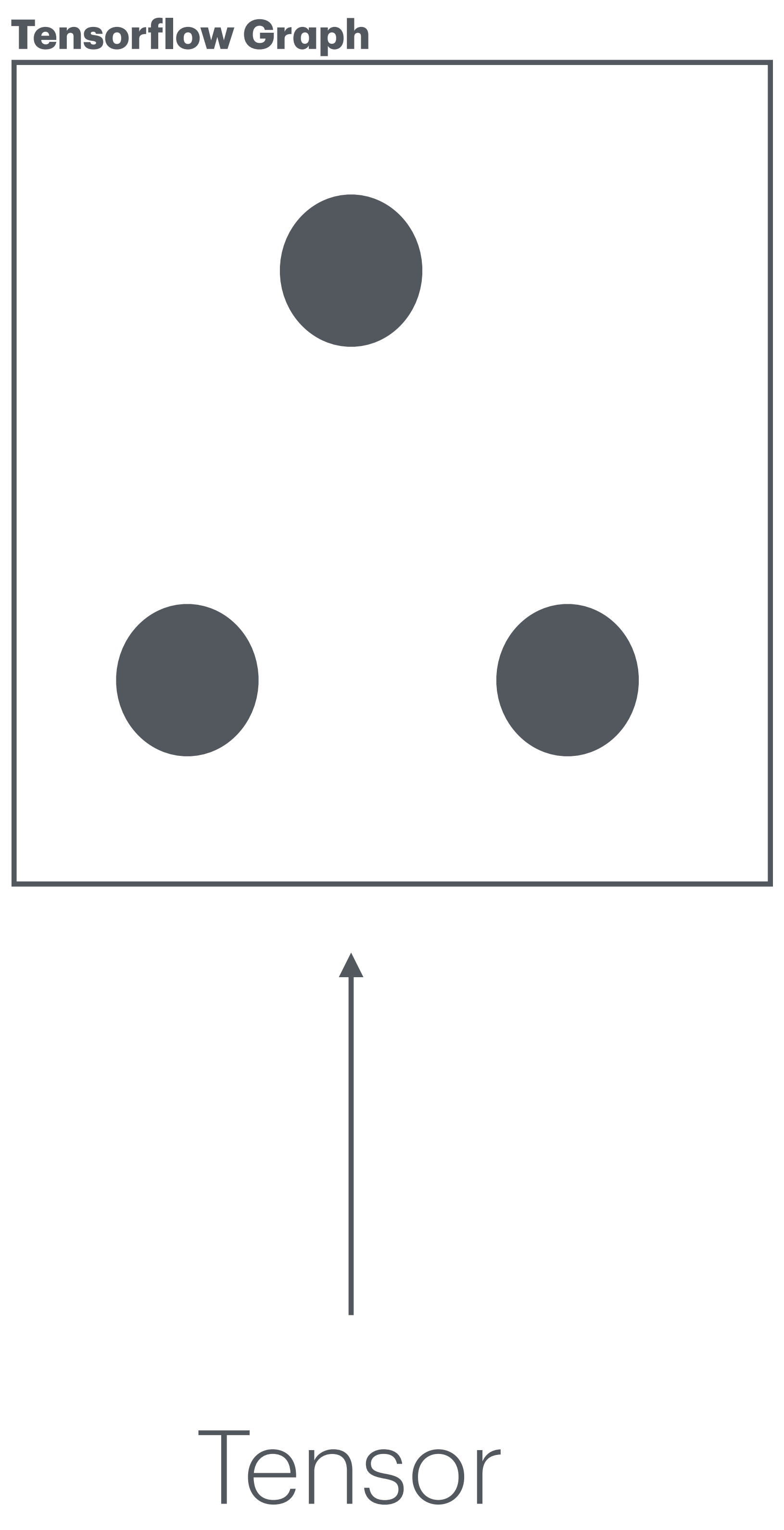

```python
class MyNormalNoise(tf.keras.Layer):
  def __init__(self, mean=0.0, stddev=1.0, seed=32):
    self.mean = mean
    self.stddev = stddev
    self.seed = seed
  def call(self, X, training=None):
    if training:
      return X + tf.random.normal(tf.shape(X), mean=self.mean, stddev=self.stddev, seed=self.seed)
    else:
      return X
  def compute_output_shape(self, batch_input_shape):
    return batch_input_shape
```

Custom Loop

Control over metrics, losses, averaging, optimizers, conditional flow, kernel constraints, regularizers, etc.

This replaces the use of model.compile() and model.fit()

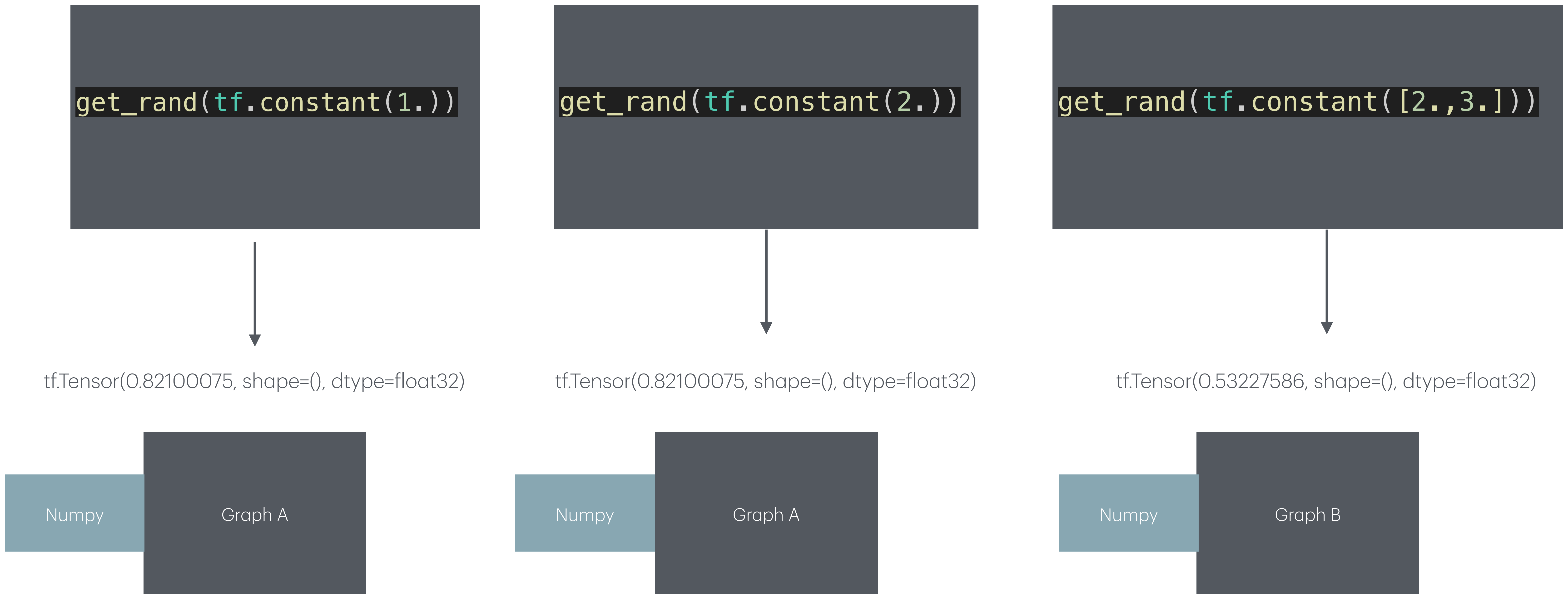
TF Functions



Use TF constructs when creating TF functions.

TF Functions

```python
@tf.function
def get_rand(value):
  return np.random.rand()
```



Tensor flow input controls graph generate.
Graphs are shared for common datatypes

Notice numpy construct (random number) ***is not*** captured in the graph. It is **generated on time** during function **tracing**

TF Functions

```
@tf.function
def get_rand(value):
  return tf.random.uniform([])
```

```
get_rand(tf.constant(1.))
```

tf.Tensor(0.22313416, shape=(), dtype=float32)

Graph A

```
get_rand(tf.constant(2.))
```

tf.Tensor(0.46204436, shape=(), dtype=float32)

Graph A

```
get_rand(tf.constant([2.,3.]))
```

tf.Tensor(0.7035961, shape=(), dtype=float32)

GraphB

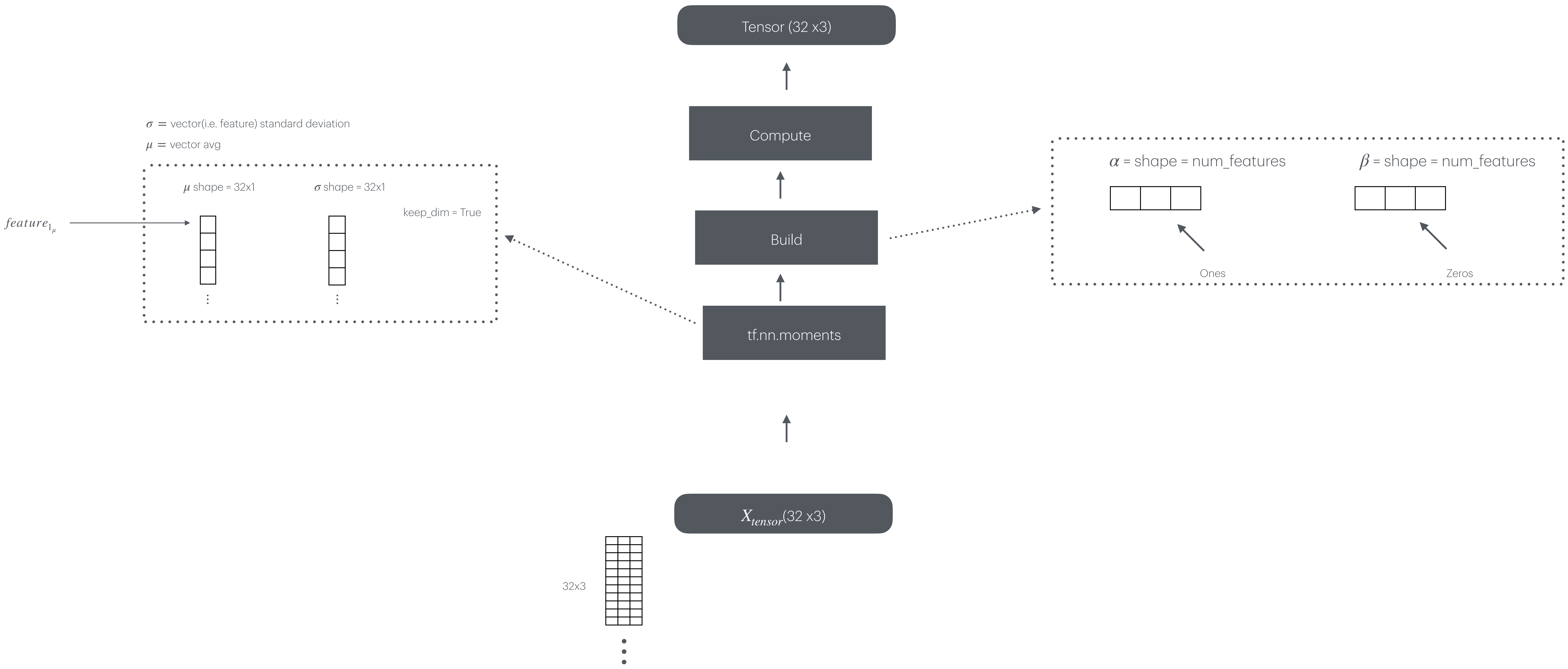
Function is fully part of graph. New random number generated on every call because function is fully part of graph

TF Functions

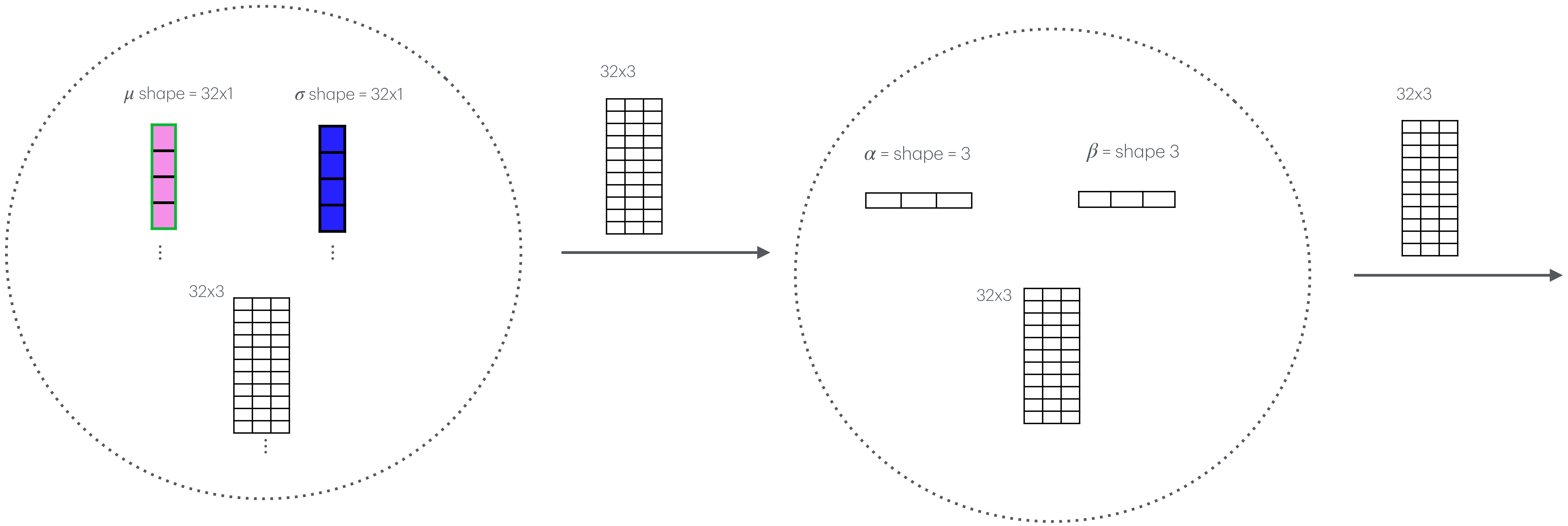
```
@tf.function
def get_rand(value):
  return tf.random.uniform([])
```

Passing python variables to TF Functions counteracts the optimized polymorphism of Tensorflow graphs. ***New Graphs will be generated for every function call which eats up RAM*** (deleting the function is the only way to remove graphs)

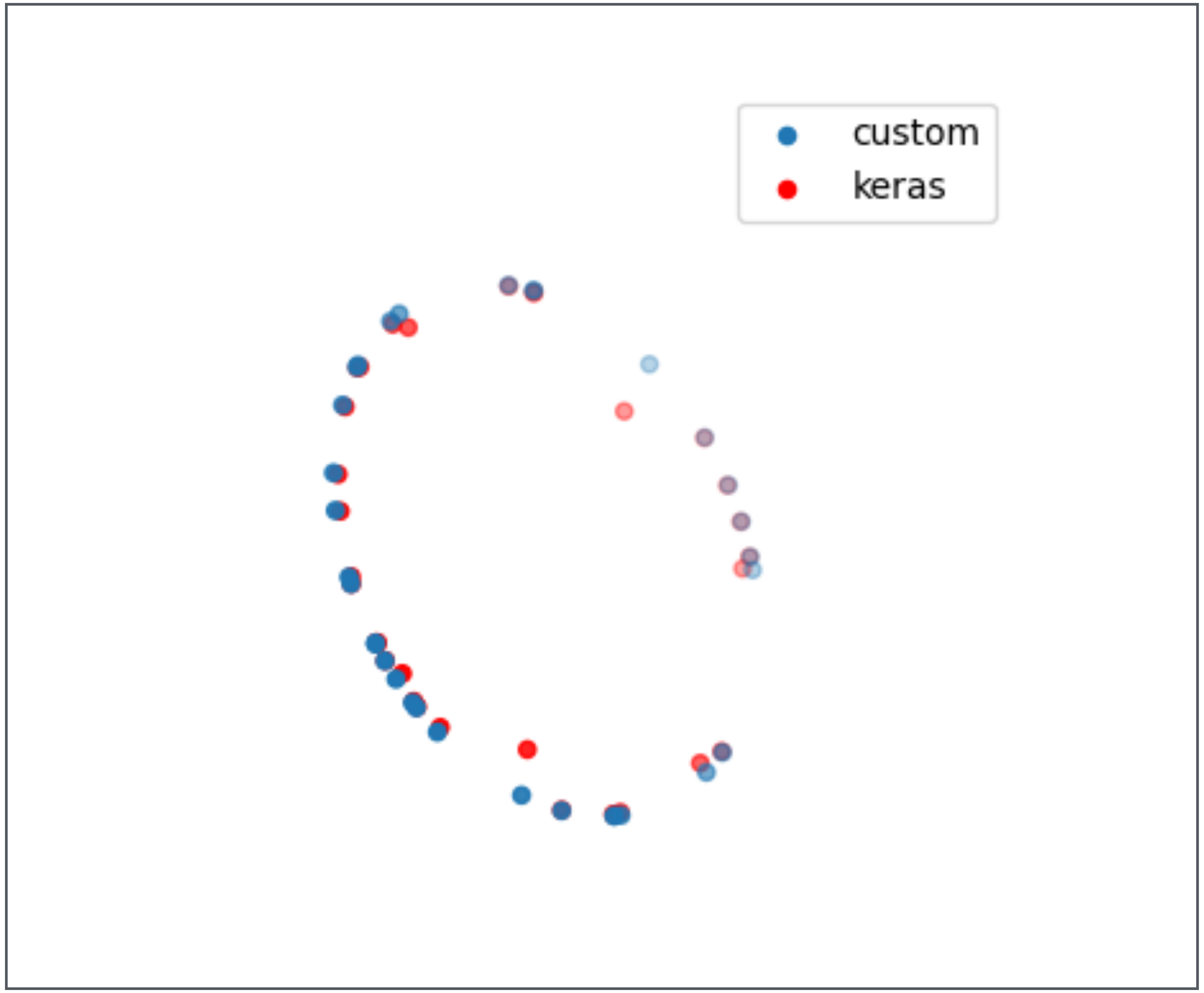
Custom Layer Normalization



Custom Layer Normalization



Custom Layer Normalization



Three Dimensional Plot