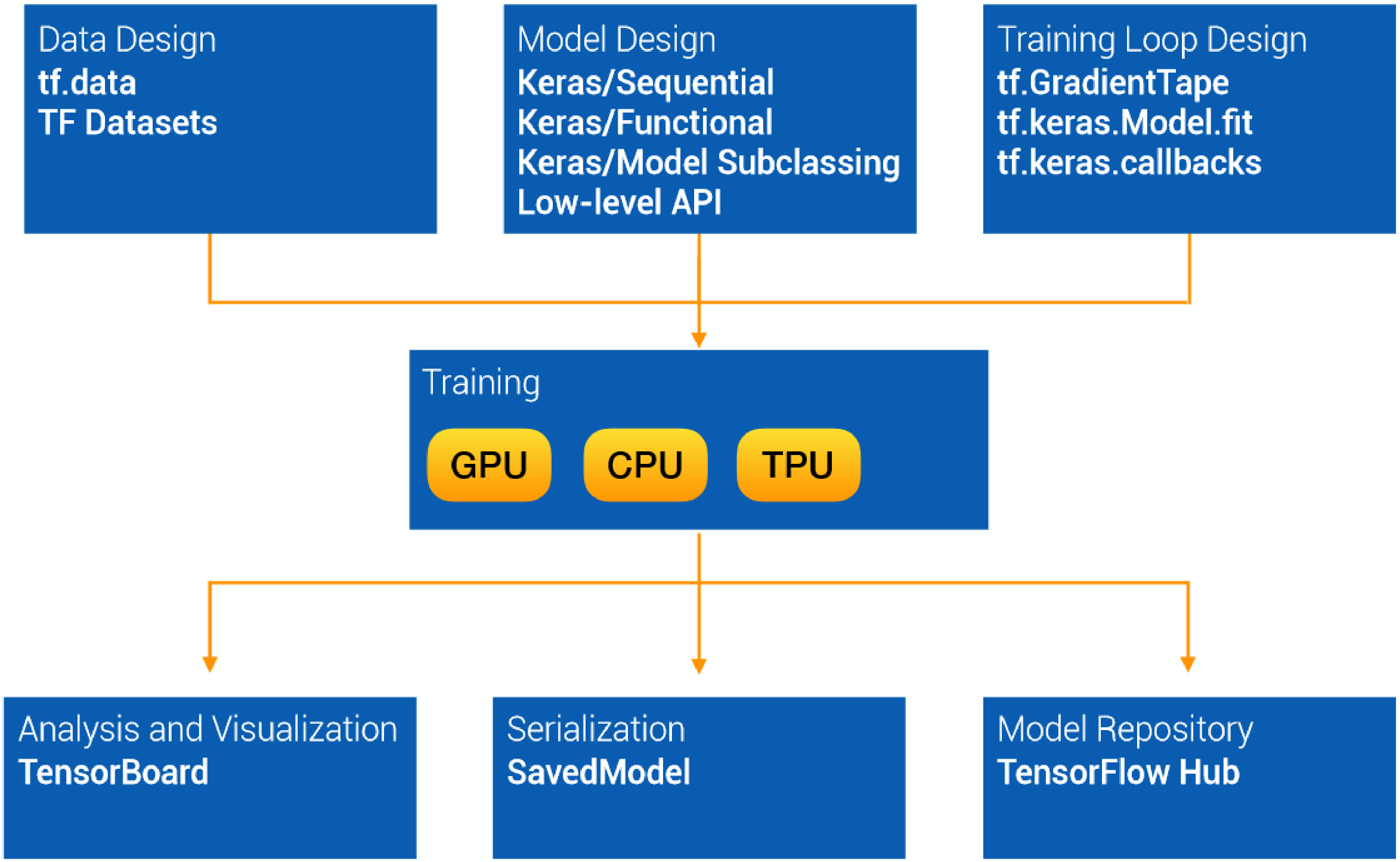
TensorFlow 2.0 Tutorial: Part #5

Saving and Restoring Models

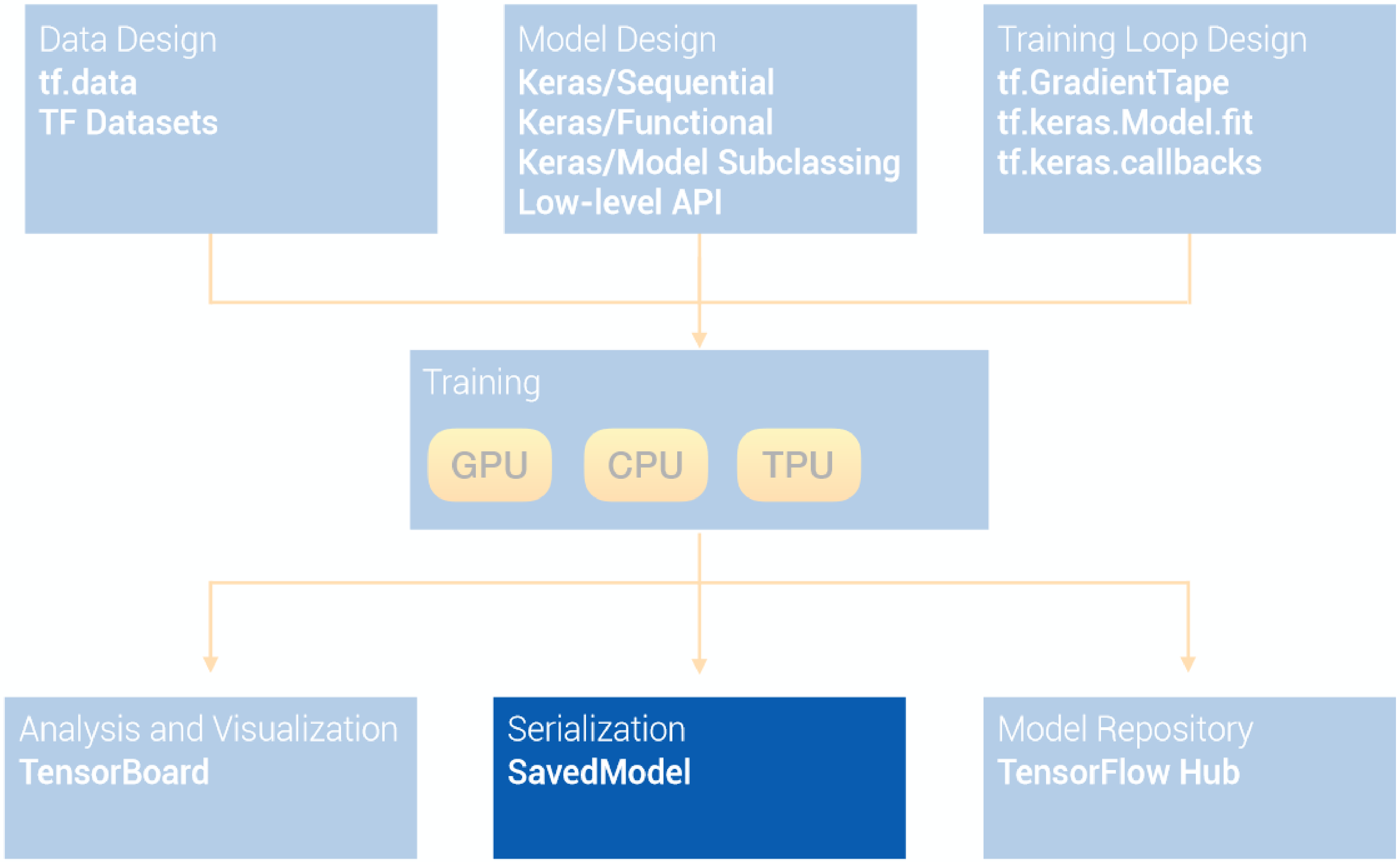
Iran University of Science and Technology (IUST)
Department of Computer Engineering

TensorFlow Overview



[source]

TensorFlow Overview



[source]

Saving Models

Saving Models

Saving Weights

Model

State + Graph

Variables

Code

...

Saving Weights

```
# Define a simple sequential model
def create_model():
  model = tf.keras.models.Sequential([
    keras.layers.Dense(512, activation='relu', input_shape=(784,)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation='softmax')
  ])

  model.compile(optimizer='adam',
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])
  return model


...


model = create_model()


model.compile(loss='sparse_categorical_crossentropy',
              optimizer=keras.optimizers.RMSprop())
model.fit(x_train, y_train,
          batch_size=64,
          epochs=1)
```

Saving Weights

```
model.save_weights('path_to_my_tf_checkpoint.h5')
model.save_weights('path_to_my_tf_checkpoint', save_format='h5')
```

Keras HDF5

```
model.save_weights('path_to_my_tf_checkpoint')
model.save_weights('path_to_my_tf_checkpoint.anything')
model.save_weights('path_to_my_tf_checkpoint', save_format='tf')
```

TensorFlow
Checkpoint

Saving Weights

```
model.save_weights('path_to_my_tf_checkpoint.h5')
model.save_weights('path_to_my_tf_checkpoint', save_format='h5')
```

Keras HDF5

```
model.save_weights('path_to_my_tf_checkpoint')
model.save_weights('path_to_my_tf_checkpoint.anything')
model.save_weights('path_to_my_tf_checkpoint', save_format='tf')
```

TensorFlow Checkpoint

- The layers' weights
- The optimizer's state
- Any variables associated with stateful model metrics (if any)

Loading Weights

```
restored_model = create_model()  #model's architecture
```

```
restored_model.load_weights('path_to_weights')
```

Note that optimizer is not preserved, if you don’t compile the model with the exact same arguments, before `load_weights`

Works with models created using:

- Sequential models
- Functional API
- Sub-classing

Works with models created using:

- Sequential
- Functional API
- Sub-classing → little tricky
  (because of lazy initialization)

Recall

Sequential / Functional API

Tensor shapes are known from the beginning

Sub-classing

Tensor shapes are un-known until initialization

Saving Weights In Sub-classed Models

```
class CustomModel(keras.Model):

  def __init__(self, name=None):
    super(CustomModel, self).__init__(name=name)
    self.dense_1 = layers.Dense(64, activation='relu', name='dense_1')
    self.dense_2 = layers.Dense(64, activation='relu', name='dense_2')
    self.pred_layer = layers.Dense(10, activation='softmax', name='predictions')


  def call(self, inputs):
    x = self.dense_1(inputs)
    x = self.dense_2(x)
    return self.pred_layer(x)


def get_model():
  return CustomModel (name='3_layer_mlp')
```

Saving Weights In Sub-classed Models

```
class CustomModel(keras.Model):

  def __init__(self, name=None):
    super(CustomModel, self).__init__(name=name)
    self.dense_1 = layers.Dense(64, activation='relu', name='dense_1')
    self.dense_2 = layers.Dense(64, activation='relu', name='dense_2')
    self.pred_layer = layers.Dense(10, activation='softmax', name='predictions')


  def call(self, inputs):
    x = self.dense_1(inputs)
    x = self.dense_2(x)
    return self.pred_layer(x)


def get_model():
  return CustomModel (name='3_layer_mlp')
```

Un-known Input shape
until the first call

Saving Weights In Sub-classed Models

```
model = get_model()

...

model.compile(loss='sparse_categorical_crossentropy',
              optimizer=keras.optimizers.RMSprop())

model.fit(x_train, y_train,
          batch_size=64,
          epochs=1)
```

Initializing variables

Loading Weights In Sub-classed Models

```
new_model = get_model()

...

model.compile(loss='sparse_categorical_crossentropy',
              optimizer=keras.optimizers.RMSprop())
```

Initializing optimizer (if you want to resume training)

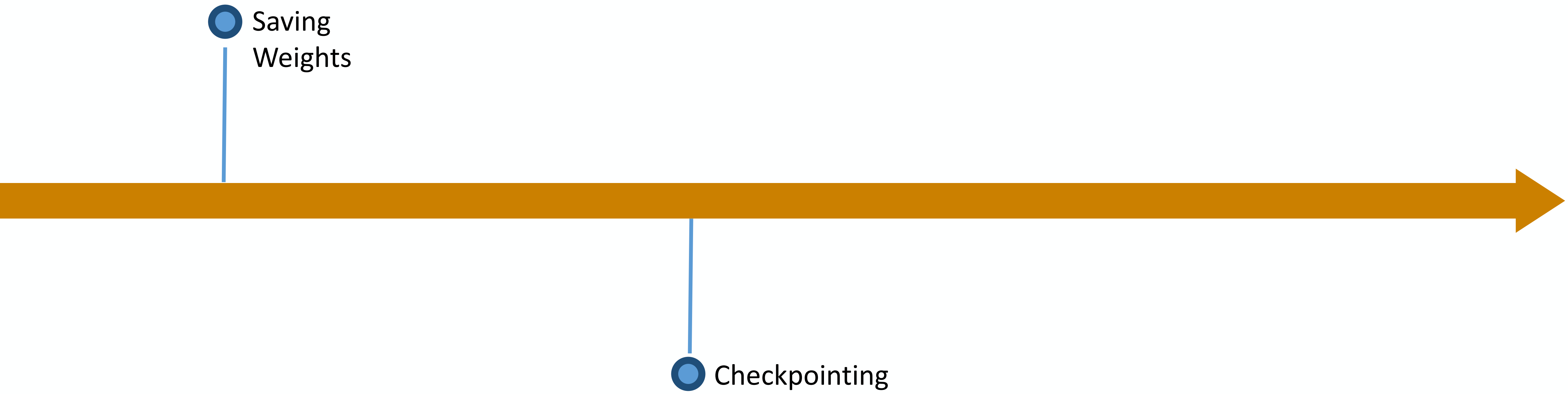
```
model.fit(x_train, y_train,
          batch_size=64,
          epochs=1)
```

Initializing variables

```
new_model.load_weights('path_to_my_weights')
```

Saving Models

Saving Models



Checkpointing

Easy Way

checkpoint callback

Manual Way

checkpoint objects

Manager objects

Callbacks: utilities called at certain points during model training

- LearningRateScheduler: Learning rate scheduler
- ProgbarLogger: Callback that prints metrics to stdout
- ReduceLROnPlateau: Reduce learning rate when a metric has stopped improving

…

- ModelCheckpoint: Save the model

Checkpoint Callback - Storing

```
checkpoint_path = "training_1/cp.ckpt"

# Create a callback that saves the model's weights
cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                 save_weights_only=True,
                                                 verbose=1)
```

Checkpoint Callback - Storing

```
checkpoint_path = "training_1/cp.ckpt"

# Create a callback that saves the model's weights
cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                 save_weights_only=True,
                                                 verbose=1)

# Train the model with the new callback
model.fit(dataset,
          epochs=10,
          validation_data=validation_dataset,
          callbacks=[cp_callback])  # Pass callback to training


# This may generate warnings related to saving the state of the optimizer.
# These warnings (and similar warnings throughout this notebook)
# are in place to discourage outdated usage, and can be ignored.
```

Checkpoint Callback - Loading

Just like before

```
# Create a basic model instance
new_model = create_model()

# Loads the weights
new_model.load_weights(checkpoint_path)
```

Checkpoint Callback - Options

```
# Include the epoch in the file name (uses `str.format`)
checkpoint_path = "training_2/cp-{epoch:04d}.ckpt"
```

```
# Create a callback that saves the model's weights every 5 epochs
cp_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_path,
    verbose=1,
    save_weights_only=True,
    period=5)
```

Checkpointing

Easy Way
checkpoint callback

Manual Way
checkpoint objects
Manager objects

can be used with manual training loops

Manual Checkpointing

```
# Create a model instance
model = create_model()

opt = tf.keras.optimizers.Adam(0.1)
```

Manual Checkpointing

```
# Create a model instance
model = create_model()

opt = tf.keras.optimizers.Adam(0.1)

ckpt = tf.train.Checkpoint (step=tf.Variable(1), optimizer=opt, net=net)
```

Manual Checkpointing

```
# Create a model instance
model = create_model()

opt = tf.keras.optimizers.Adam(0.1)

ckpt = tf.train.Checkpoint (step=tf.Variable(1), optimizer=opt, model=model)
```

```
class Checkpoint(tracking.AutoTrackable):
  def __init__(self, **kwargs):
      ...
```

Manual Checkpointing

```
# Create a model instance
model = create_model()

opt = tf.keras.optimizers.Adam(0.1)

ckpt = tf.train.Checkpoint(step=tf.Variable(1), optimizer=opt, net=net)

manager = tf.train.CheckpointManager(ckpt, './tf_ckpts', max_to_keep=3)
```

tf.train.CheckpointManager can be helpful for managing multiple checkpoints

Manual Checkpointing

```
...

ckpt.restore(manager.latest_checkpoint)
  if manager.latest_checkpoint:
    print("Restored from {}".format(manager.latest_checkpoint))
  else:
    print("Initializing from scratch.")
```

Manual Checkpointing

```
...

ckpt.restore(manager.latest_checkpoint)
  if manager.latest_checkpoint:
    print("Restored from {}".format(manager.latest_checkpoint))
  else:
    print("Initializing from scratch.")

  for example in dataset:


      with tf.GradientTape() as tape:
          output = model(example['x'])
          loss = tf.reduce_mean(tf.abs(output - example['y']))
      variables = model.trainable_variables
      gradients = tape.gradient(loss, variables)
      opt.apply_gradients(zip(gradients, variables))
```

training loop

Manual Checkpointing

```
...

ckpt.restore(manager.latest_checkpoint)
  if manager.latest_checkpoint:
    print("Restored from {}".format(manager.latest_checkpoint))
  else:
    print("Initializing from scratch.")

  for example in dataset:


      with tf.GradientTape() as tape:
          output = model(example['x'])
          loss = tf.reduce_mean(tf.abs(output - example['y']))
      variables = model.trainable_variables
      gradients = tape.gradient(loss, variables)
      opt.apply_gradients(zip(gradients, variables))

    ckpt.step.assign_add(1)
    if int(ckpt.step) % 10 == 0:
      save_path = manager.save()
      print("Saved checkpoint for step {}: {}".format(int(ckpt.step), save_path))
```

checkpointing

Manual Checkpointing - Reloading

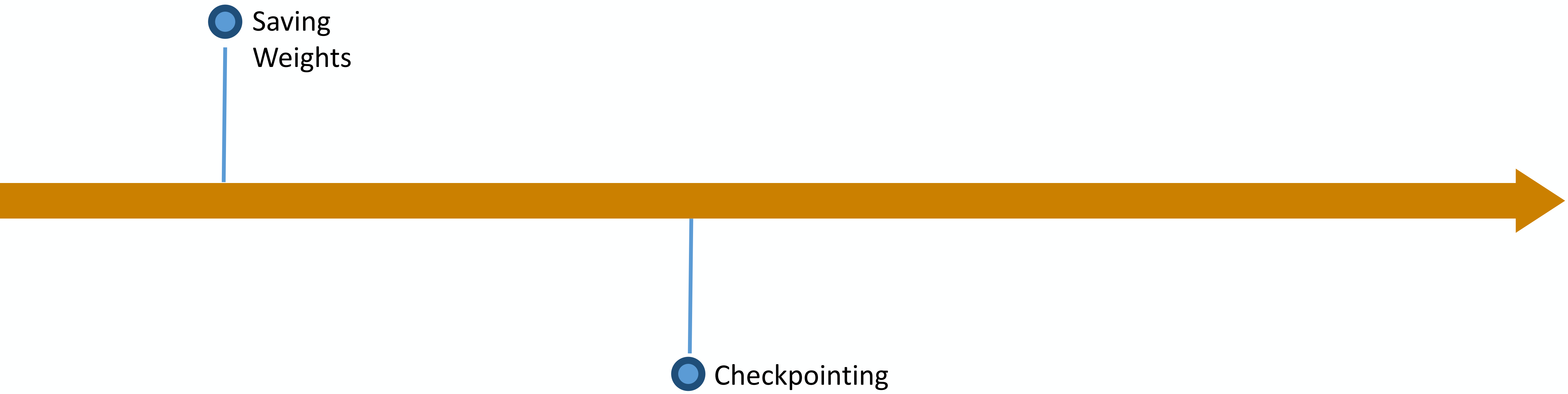
```
opt = tf.keras.optimizers.Adam(0.1)
model = ceate_model()
ckpt = tf.train.Checkpoint(step=tf.Variable(1), optimizer=opt, model=model)
manager = tf.train.CheckpointManager(ckpt, './tf_ckpts', max_to_keep=3)
```

pass a new model and manager and pickup training exactly where you left off

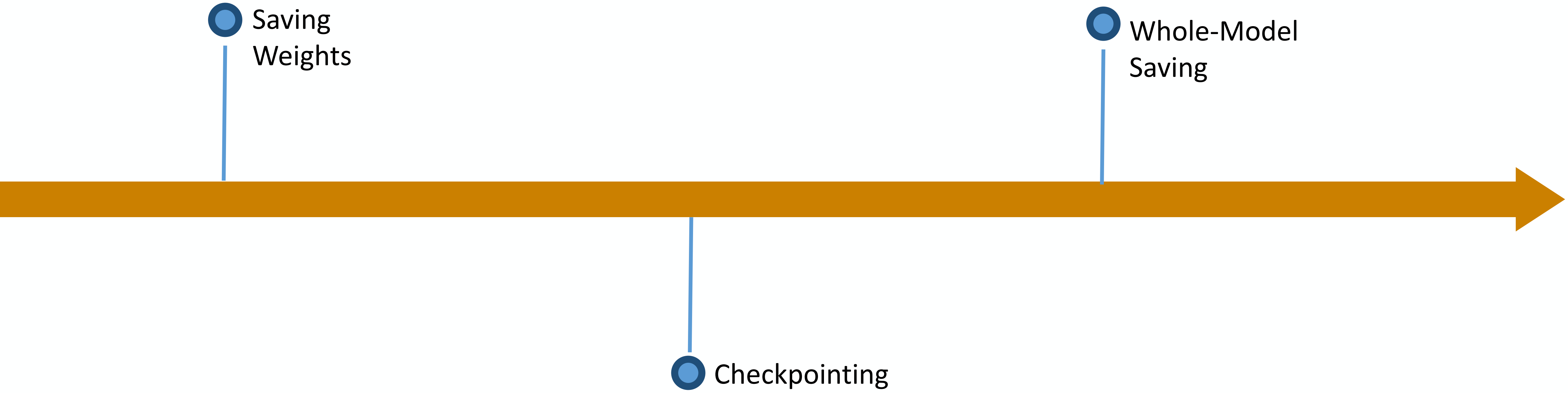
Manual Checkpointing - Reloading

```
opt = tf.keras.optimizers.Adam(0.1)
model = create_model()
ckpt = tf.train.Checkpoint(step=tf.Variable(1), optimizer=opt, model=model)
manager = tf.train.CheckpointManager(ckpt, './tf_ckpts', max_to_keep=3)

ckpt.restore(manager.latest_checkpoint)
  if manager.latest_checkpoint:
    print("Restored from {}".format(manager.latest_checkpoint))
  else:
    print("Initializing from scratch.")

  for example in dataset:
      with tf.GradientTape() as tape:
          output = model(example['x'])
          loss = tf.reduce_mean(tf.abs(output - example['y']))
      variables = model.trainable_variables
      gradients = tape.gradient(loss, variables)
      opt.apply_gradients(zip(gradients, variables))

    ckpt.step.assign_add(1)
    if int(ckpt.step) % 10 == 0:
      save_path = manager.save()
      print("Saved checkpoint for step {}: {}".format(int(ckpt.step), save_path))
```

Saving Models



Saving Models



Whole-model saving

Sequential / Functional API

- Super easy
- Can be fully saved
- Access to all layers

Sub-classing

- Tricky
- Saved partly
- No access to layers

Whole-model saving

Sequential / Functional API

- Super easy
- Can be fully saved
- Access to all layers

→

The model's architecture
The model's weight values
The model's training config (what you passed to compile)
The optimizer and its state

Whole-model saving

```
# Export the whole model to a SavedModel
model.save('path_to_saved_model', save_format='tf')
```

'h5' can also be used like before

Not recommended

Whole-model saving

```
# Export the whole model to a SavedModel
model.save('path_to_saved_model', save_format='tf')
```

```
# Recreate the exact same model
new_model = keras.models.load_model('path_to_saved_model')
```

everything is preserved, including optimizer's config and state

Whole-model saving

```
# Export the whole model to a SavedModel
model.save('path_to_saved_model', save_format='tf')



# Recreate the exact same model
new_model = keras.models.load_model('path_to_saved_model')


# Layers can be accessed using tensofrlow.keras.Model.get_layer
layer = new_model.get_layer ('layer_name')

...
```

How to find layer_name:

- Specify `name` when creating
- Use model.layers
- Use model.summary
- …

Whole-model saving – Sub-classing

- Not recommended

- Possible using @tf.function decorator
- Inner layers can not be accesed

For more details, check out the documentation:
https://www.tensorflow.org/guide/saved_model

Thank you!