TensorFlow 2.0 Tutorial: Part #4

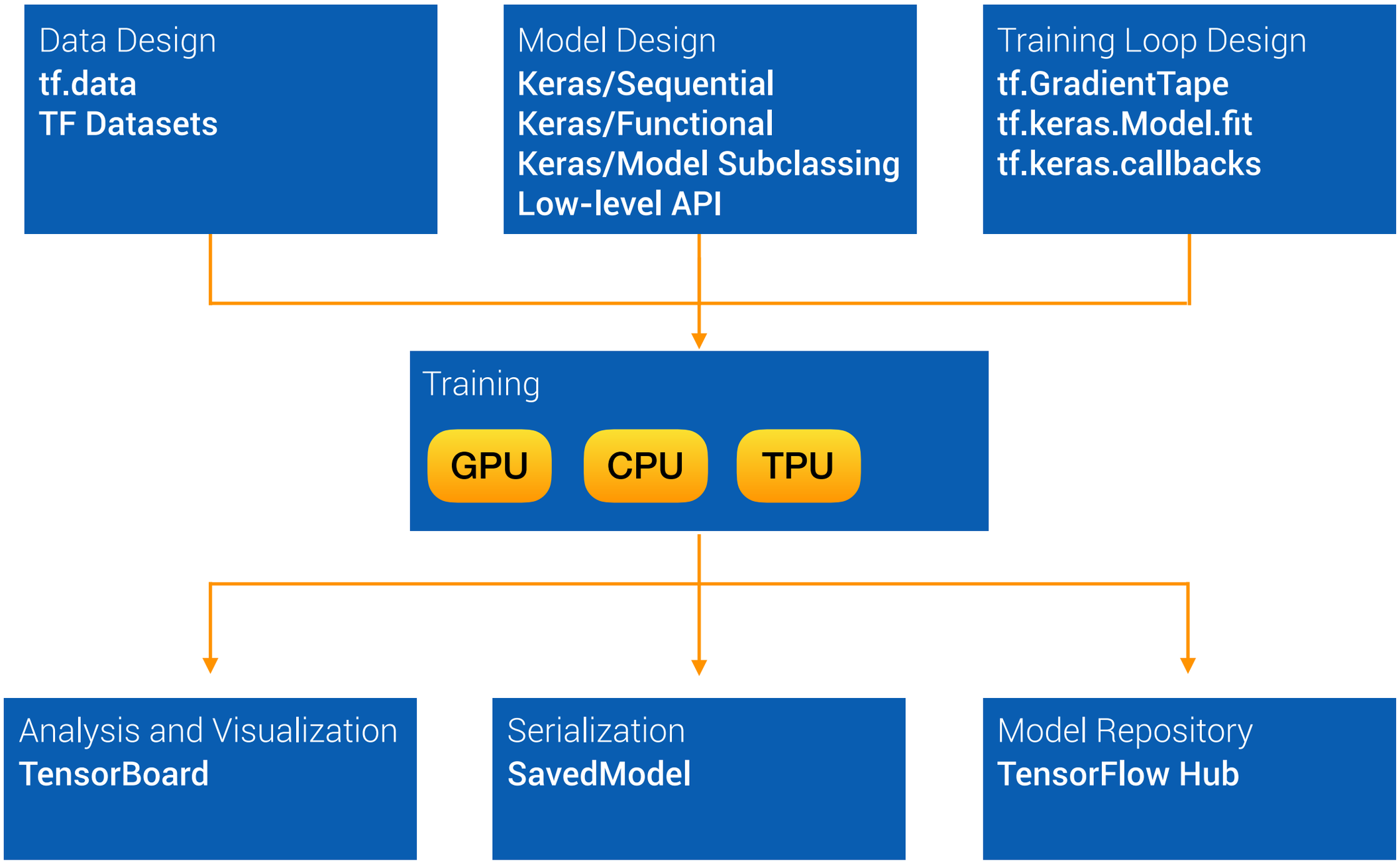
High-level APIs for Model training
(model.fit(..) & GradientTape)

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

URL:

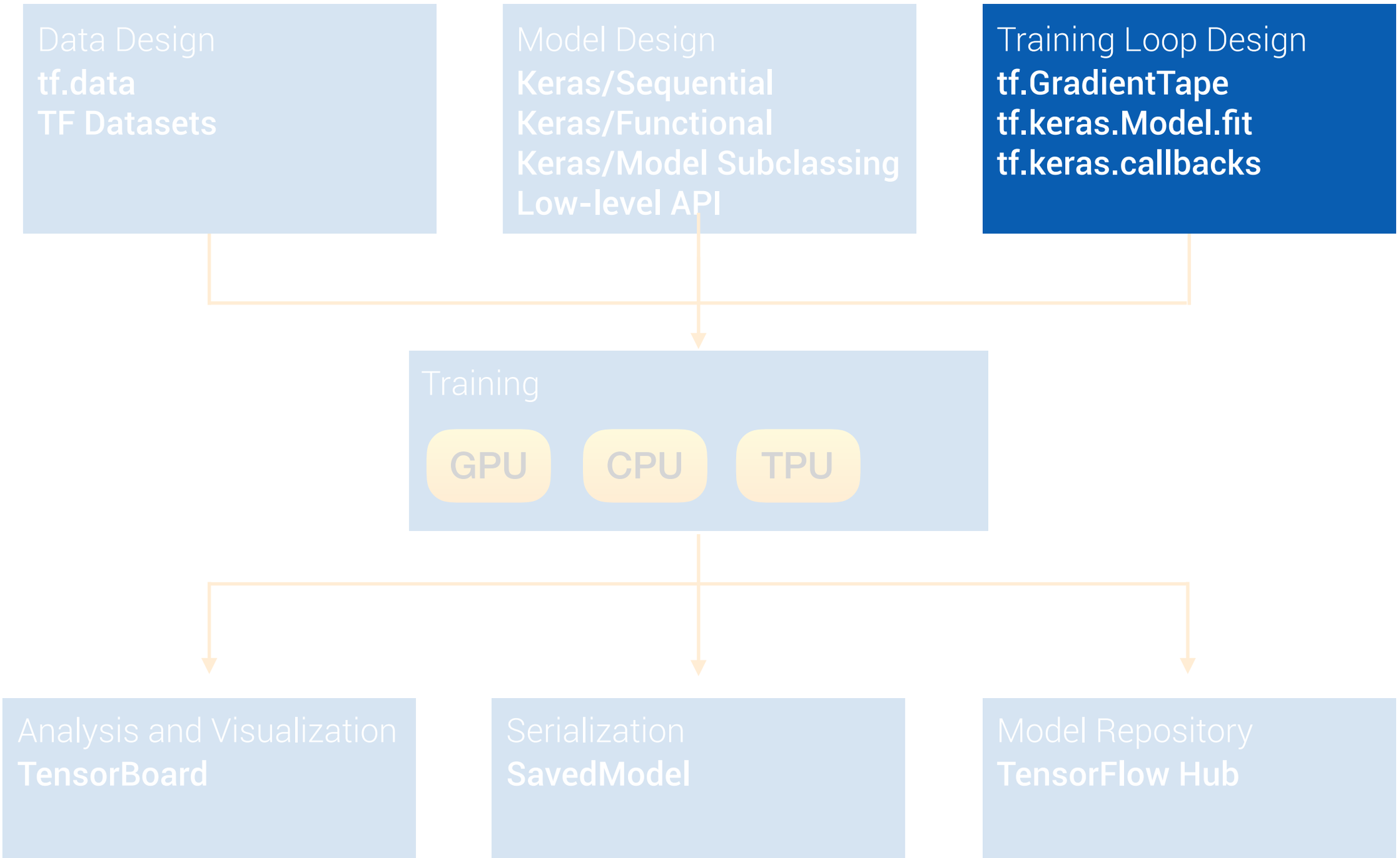
github.com/iust-deep-learning/tensorflow-2-tutorial/tree/master/part_04_model_training_apis

TensorFlow Overview



[source]

TensorFlow Overview



Model Training

Model Training

- Keras **built-in** loops

  model.fit()
  + callbacks

  `Fast prototyping`

Keras built-in training loops

```
model = MyModel()
```

Keras built-in training loops

```
model = MyModel()
```

Keras Sequential,
Functional,
and Model Sub-classing APIs

Keras built-in training loops

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()])
```

Keras built-in training loops

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()])
```

Keras built-in training loops

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()])
```

- .compile() is about configuring the training process, such as specifying the optimizer, loss, and metrics.

Keras built-in training loops

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()])

history = model.fit(data,
                    epochs=10, batch_size=128,
                    validation_data=val_data,
                    callbacks=[EarlyStopping(),
                               TensorBoard(),
                               ModelCheckpoint()])
```

Keras built-in training loops

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()])

history = model.fit(data,
              epochs=10, batch_size=128,
              validation_data=val_data,
              callbacks=[EarlyStopping(),
                         TensorBoard(),
                         ModelCheckpoint()])

results = model.evaluate(test_data, batch_size=128)
```

Keras built-in training loops (run in dynamic graph mode)

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()]
              run_eagerly=True)

history = model.fit(data,
          epochs=10, batch_size=128,
          validation_data=val_data,
          callbacks=[EarlyStopping(),
                     TensorBoard(),
                     ModelCheckpoint()])

results = model.evaluate(test_data, batch_size=128)

predictions = model.predict(x_test[:3])
```

Model Training

- Keras **built-in** loops

  model.fit()
  + callbacks

  `Fast prototyping`

Model Training

- Keras **built-in** loops

  model.fit()
  + callbacks

  `Fast prototyping`

- **GradientTape**

  custom training loops

  `Complete control`

Gradient tapes

Tensorflow "records" all operations executed inside the context of a *tf.GradientTape* onto a "tape".

Tensorflow then uses that tape and the gradients associated with each recorded operation to compute the gradients of a "recorded" computation using reverse mode differentiation.

Gradient Tape

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
    g.watch(x)
    y = x * x

dy_dx = g.gradient(y, x) # Will compute to 6.0
```

Gradient Tape

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
    g.watch(x)
    y = x * x

dy_dx = g.gradient(y, x) # Will compute to 6.0
```

How to calculate second-oder derivatives?

Gradient Tape

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
    g.watch(x)
    with tf.GradientTape() as gg:
        gg.watch(x)
        y = x * x
    dy_dx = gg.gradient(y, x)      # Will compute to 6.0

d2y_dx2 = g.gradient(dy_dx, x)  # Will compute to 2.0
```

Gradient Tape; A real-world usage

Gradient Tape; A real-world usage

```
model = MyModel()
```

Keras Sequential,
Functional,
and Model Sub-classing APIs

Gradient Tape; A real-world usage

```
model = MyModel()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
```

Gradient Tape; A real-world usage

```
model = MyModel()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
```

```
logits = model(features)
loss = loss_fn(labels, logits)
```

Gradient Tape; A real-world usage

```
model = MyModel()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
```

```
logits = model(features)
loss = loss_fn(labels, logits)
```

Gradient Tape; A real-world usage

```
model = MyModel()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
```

```
with tf.GradientTape() as tape:
    logits = model(features)
    loss = loss_fn(labels, logits)
```

Gradient Tape; A real-world usage

```
model = MyModel()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
```

```
with tf.GradientTape() as tape:
    logits = model(features)
    loss = loss_fn(labels, logits)

grads = tape.gradient(loss, model.trainable_variables)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Gradient Tape; A real-world usage

```
model = MyModel()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
```

```
with tf.GradientTape() as tape:
    logits = model(features)
    loss = loss_fn(labels, logits)

grads = tape.gradient(loss, model.trainable_variables)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Gradient Tape; A real-world usage

```
model = MyModel()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)


def train_step(features, labels):
    with tf.GradientTape() as tape:
        logits = model(features)
        loss = loss_fn(labels, logits)

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Gradient Tape; A real-world usage

```
model = MyModel()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)

@tf.function # optional
def train_step(features, labels):
    with tf.GradientTape() as tape:
        logits = model(features)
        loss = loss_fn(labels, logits)

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Gradient Tape; A real-world usage

```
model = MyModel()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)

@tf.function # optional
def train_step(features, labels):
    with tf.GradientTape() as tape:
        logits = model(features)
        loss = loss_fn(labels, logits)

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

for features, labels in data:
    loss = train_step(features, labels)
```

Gradient Tape; A real-world usage: Measuring the Accuracy

Gradient Tape; A real-world usage: Measuring the Accuracy

```
model = ...; optimizer = ...; loss_fn = ...
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
model = ...; optimizer = ...; loss_fn = ...

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
    name='train_accuracy')
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
model = ...; optimizer = ...; loss_fn = ...

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
    name='train_accuracy')

@tf.function
def train_step(features, labels):
    with tf.GradientTape() as tape:
        logits = model(features)
        loss = loss_fn(labels, logits)

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
model = ...; optimizer = ...; loss_fn = ...

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
    name='train_accuracy')

@tf.function
def train_step(features, labels):
    with tf.GradientTape() as tape:
        logits = model(features)
        loss = loss_fn(labels, logits)

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

    train_loss(loss)
    train_accuracy(labels, logits)
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
    for images, labels in train_ds:
      train_step(images, labels)
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
EPOCHS = 5
for epoch in range(EPOCHS):



    for images, labels in train_ds:
        train_step(images, labels)
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
EPOCHS = 5
for epoch in range(EPOCHS):



    for images, labels in train_ds:
        train_step(images, labels)

    template = 'Epoch {}, Loss: {}, Accuracy: {}'
    print(template.format(epoch+1,
                          train_loss.result(),
                          train_accuracy.result()*100)
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
EPOCHS = 5
for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_states()
    train_accuracy.reset_states()

    for images, labels in train_ds:
        train_step(images, labels)

    template = 'Epoch {}, Loss: {}, Accuracy: {}'
    print(template.format(epoch+1,
                          train_loss.result(),
                          train_accuracy.result()*100)
```

How to use tf.keras.metrics?

```
m = SomeMetric(...)
for input in ...:
    m.update_state(input)

print('Final result: ', m.result())
```

How to use tf.keras.metrics?

```
m = SomeMetric(...)
for input in ...:
    m.update_state(input)

print('Final result: ', m.result())
```

or

```
m = SomeMetric(...)
for input in ...:
    print('Current result': m(input))

print('Final result: ', m.result().numpy())
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
EPOCHS = 5
for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_states()
    train_accuracy.reset_states()

    for step, (images, labels) in enumerate(train_ds):
        train_step(images, labels)



template = 'Epoch {}, Loss: {}, Accuracy: {}'
print(template.format(epoch+1,
                      train_loss.result(),
                      train_accuracy.result()*100)
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
EPOCHS = 5
for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_states()
    train_accuracy.reset_states()

    for step, (images, labels) in enumerate(train_ds):
        train_step(images, labels)

        if step % 50 == 0:
            print(f'Step {step}, Loss{train_loss.result()}')

    template = 'Epoch {}, Loss: {}, Accuracy: {}'
    print(template.format(epoch+1,
                          train_loss.result(),
                          train_accuracy.result()*100)
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
...

    for step, (images, labels) in enumerate(train_ds):
        train_step(images, labels)

        if step % 50 == 0:
            print(f'Step {step}, Loss{train_loss.result()}')

...
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
...

    for step, (images, labels) in enumerate(train_ds):
        train_step(images, labels)

        if step % 50 == 0:
            print(f'Step {step}, Loss{train_loss.result()}')

...
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
...

    for step, (images, labels) in enumerate(train_ds):
        train_step(images, labels)

        if step % 50 == 0:
            print(f'Step {step}, Loss{train_loss.result()}')

            if step % 500 == 0:
                for valid_images, valid_labels in valid_ds:
                    test_step(valid_images, valid_labels)

                print(f'Step {step}',
                      f'Valid Loss: {valid_loss.result()}',
                      f'Valid Accuracy: {valid_accuracy.result()*100}')

...
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
...

    for step, (images, labels) in enumerate(train_ds):
        train_step(images, labels)

        if step % 50 == 0:
            print(f'Step {step}, Loss{train_loss.result()}')

            if step % 500 == 0:
                for valid_images, valid_labels in valid_ds:
                    test_step(valid_images, valid_labels)

                print(f'Step {step}',
                      f'Valid Loss: {valid_loss.result()}',
                      f'Valid Accuracy: {valid_accuracy.result()*100}')

...
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
@tf.function
def test_step(features, labels):
  logits = model(images)
  loss = loss_object(labels, logits)

valid_loss(loss)
valid_accuracy(labels, predictions)
```

Gradient Tape; A real-world usage: Measuring the Accuracy

```
valid_loss = tf.keras.metrics.Mean(name='valid_loss')
valid_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
    name='valid_accuracy')


@tf.function
def test_step(features, labels):
  logits = model(images)
  loss = loss_object(labels, logits)

valid_loss(loss)
valid_accuracy(labels, predictions)
```

Summary

- Keras built-in training loops
- TF Gradient Tape
- Custom Training Loops
- Using tf.keras.metrics

Thank you!