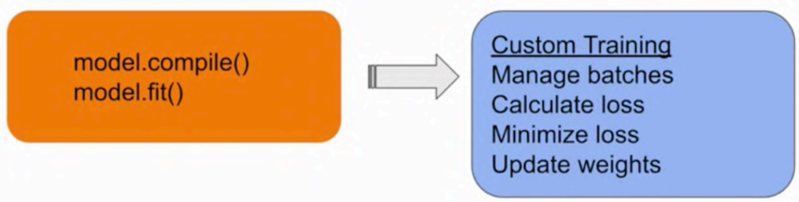
Built in Solution to Training

- Train using model.compile() and model.fit().
- Specify optimizer, loss etc in model.compile()
- model.fit() loops through batches of training data to:
  - Update trainable weights to minimize loss.
  - Achieves the above using chosen optimizer.

Custom Training Loops



Steps to training network

1. **Define** the network
2. **Prepare** the training data
3. **Define** loss and optimizer
4. **Train** the model on training inputs by minimizing loss using custom optimizer.
5. **Validate** the model.

1. Define the Model

```python
class Model():
  def __init__(self):
    self.w = tf.Variable(5.0)
    self.b = tf.Variable(0.0)

  def __call__(self, x):
    return self.w * x + self.b
```

Since we are building everything from scratch, we are defining a class called model without inheriting from a TensorFlow class.

Inside it's init constructor, model contains two variables, self.w and self.b, which are the trainable weights of the model and they're initialized with some arbitrary values, say 5 and 0.

Model has a call function where we define a linear equation that does the following calculation as the returned output.

As training progresses, w and b will be updated by our gradient descent optimization as to minimize the loss.

2. Prepare Training Data

```python
TRUE_w = 3.0
TRUE_b = 2.0
NUM_EXAMPLES = 1000

random_xs  = tf.random.normal(shape=[NUM_EXAMPLES])

ys = (TRUE_w * random_xs) + TRUE_b
```

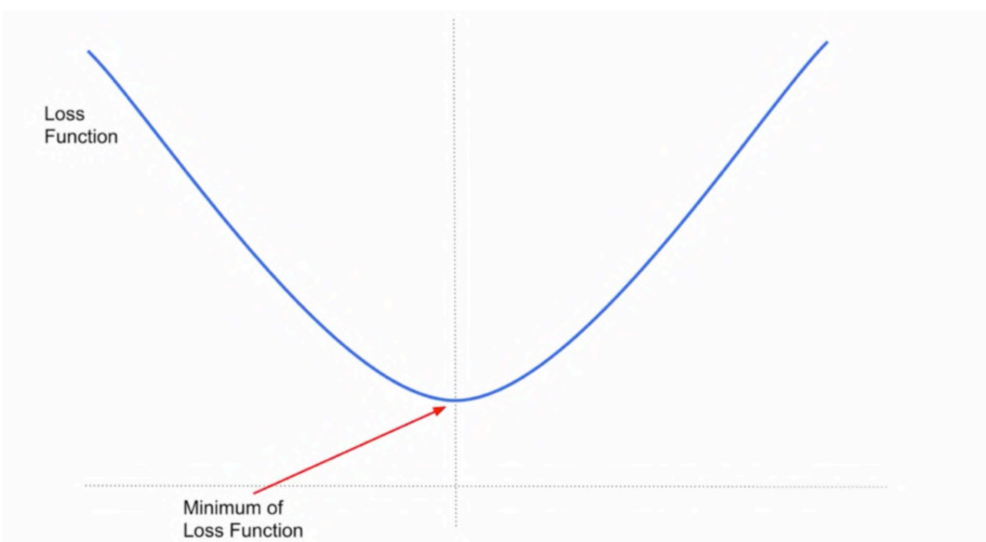
Mean Squared Error Loss

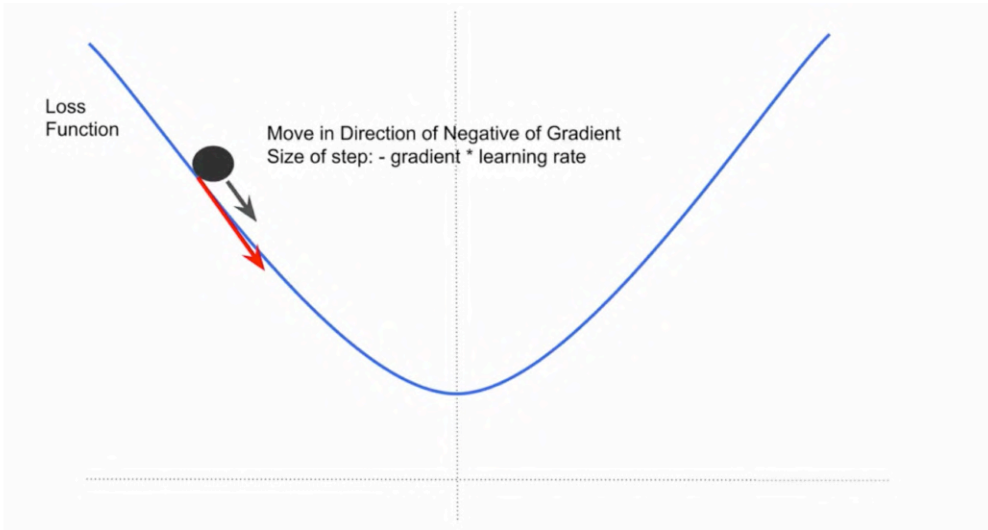
$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - y_i^{'}\right)^2$$

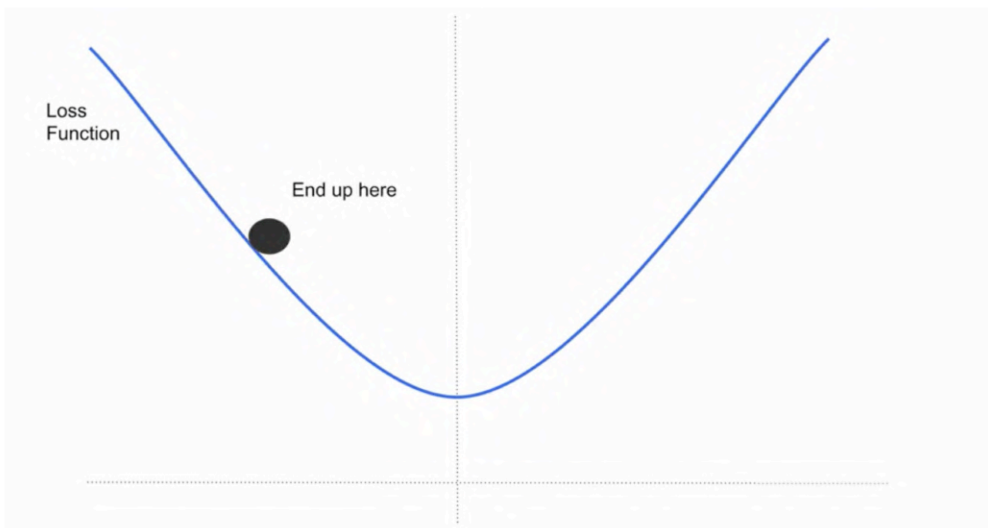
$$MSE = mean\left((Y_{true} - Y_{pred})^2\right)$$

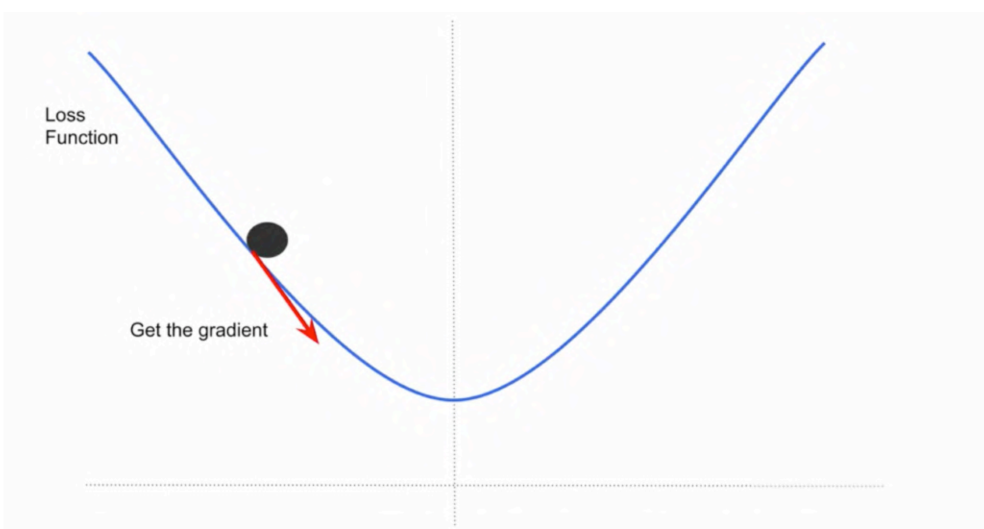
3. Mean Squared Error Loss

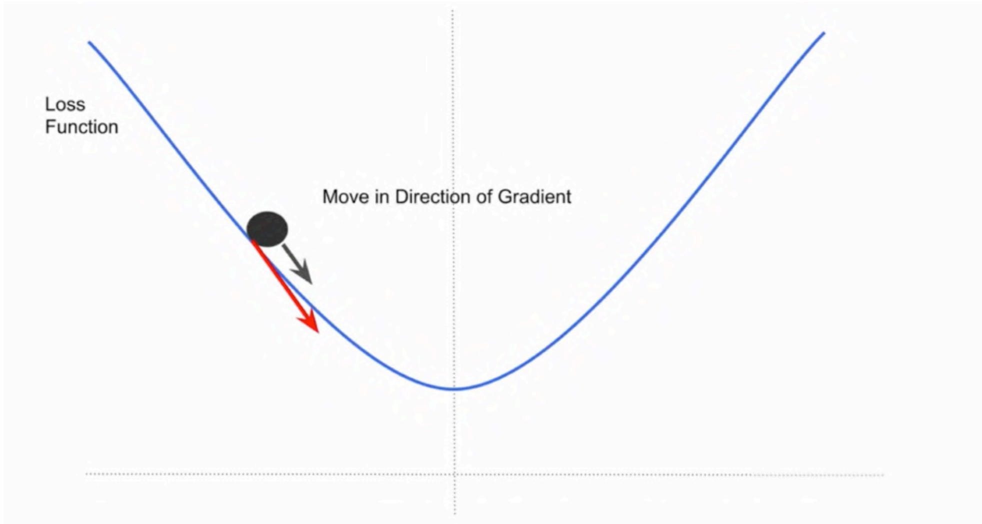
```
def loss(y_true, y_pred):
  return tf.reduce_mean(tf.square(y_true - y_pred))
```

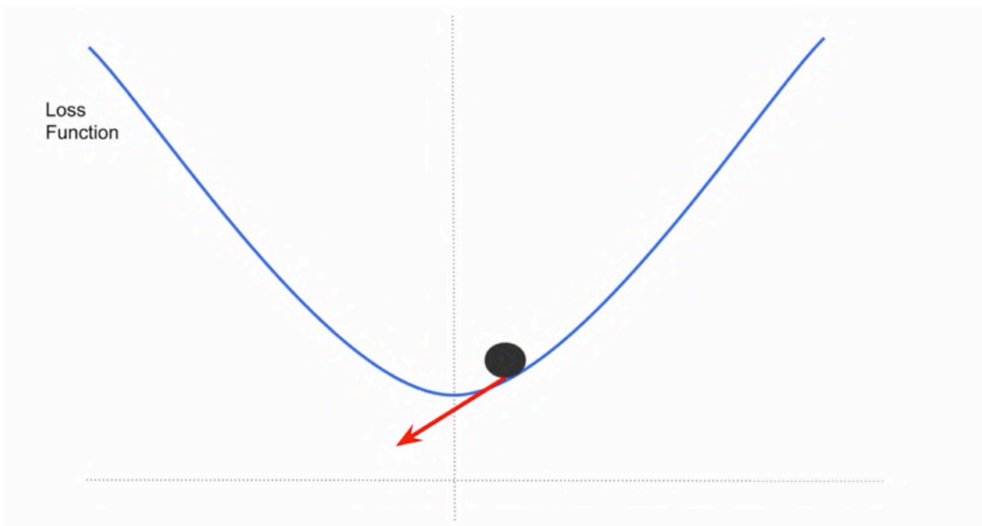


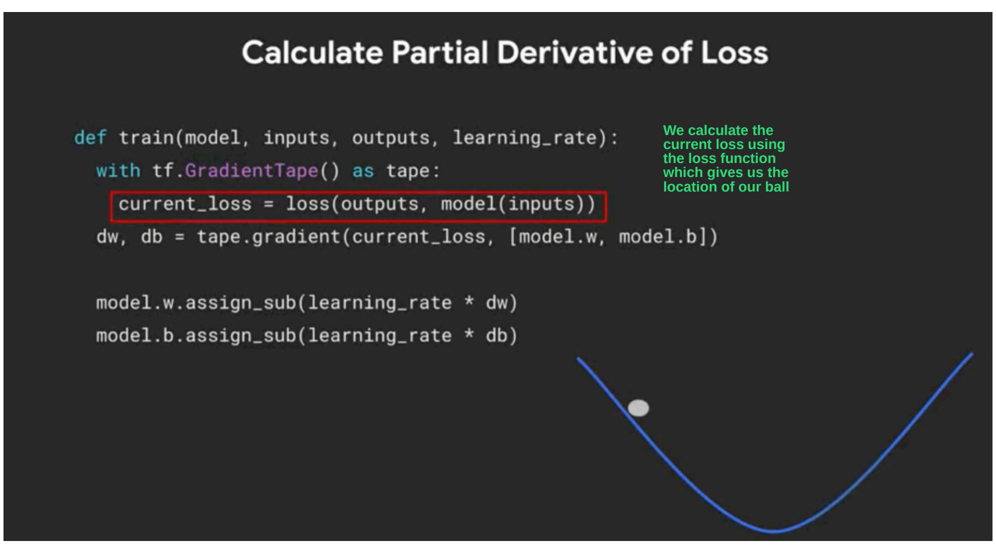

Loss
Function
Move in Direction of Negative of Gradient
Size of step: - gradient * learning rate


Loss
Function
End up here


Loss
Function
Get the gradient


Loss
Function
Move in Direction of Gradient

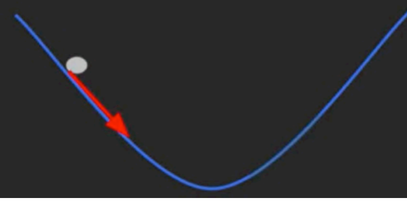

Loss
Function


Calculate Partial Derivative of Loss
def train(model, inputs, outputs, learning_rate):
with tf.GradientTape() as tape:
current_loss = loss(outputs, model(inputs))
dw, db = tape.gradient(current_loss, [model.w, model.b])
model.w.assign_sub(learning_rate * dw)
model.b.assign_sub(learning_rate * db)
We calculate the current loss using the loss function which gives us the location of our ball

Calculate Partial Derivative of Loss

```
def train(model, inputs, outputs, learning_rate):
  with tf.GradientTape() as tape:
    current_loss = loss(outputs, model(inputs))
  dw, db = tape.gradient(current_loss, [model.w, model.b])

  model.w.assign_sub(learning_rate * dw)
  model.b.assign_sub(learning_rate * db)
```

We then use the gradient tape to calculate the gradient. The negative of the gradient gives us our desired direction and the magnitude to move the ball based on the learning rate



Calculate Partial Derivative of Loss

```
def train(model, inputs, outputs, learning_rate):
  with tf.GradientTape() as tape:
    current_loss = loss(outputs, model(inputs))
  dw, db = tape.gradient(current_loss, [model.w, model.b])

  model.w.assign_sub(learning_rate * dw)
  model.b.assign_sub(learning_rate * db)
```

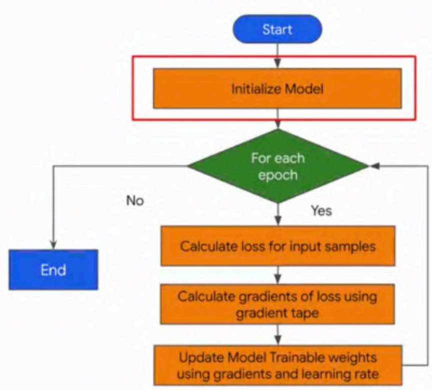
So we update w in direction of the gradient with respect w multiplied by the learning rate. The w.assign_sub() function does heavy lifting for you by updating the model variables with their new values and using the correct direction of the gradient.

The same is done for b.

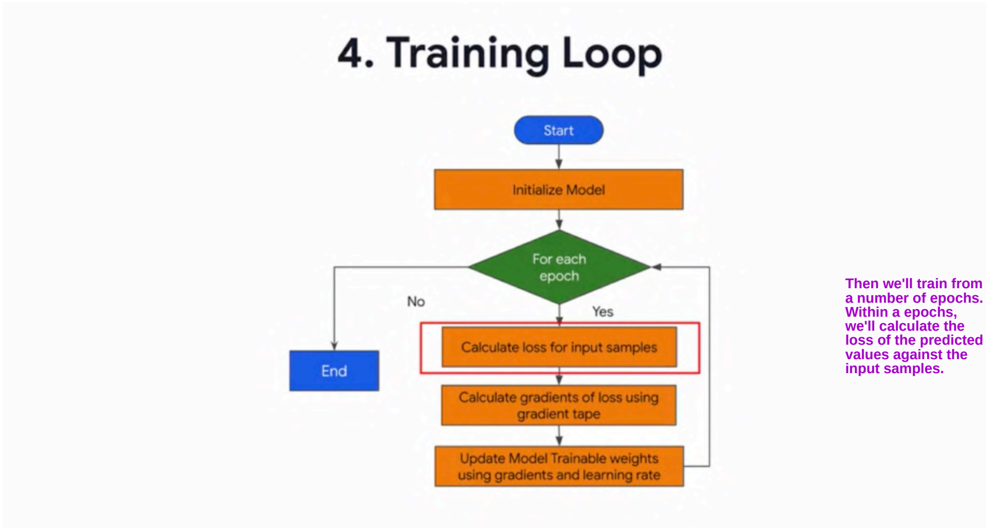
$$w = w - \alpha \times \frac{dL}{dw}$$

$$b = b - \alpha \times \frac{dL}{db}$$

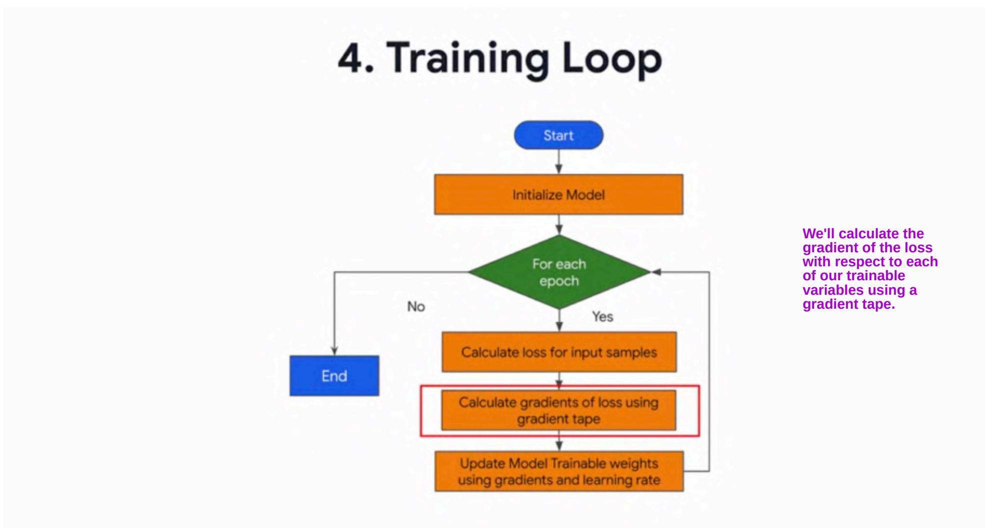
4. Training Loop



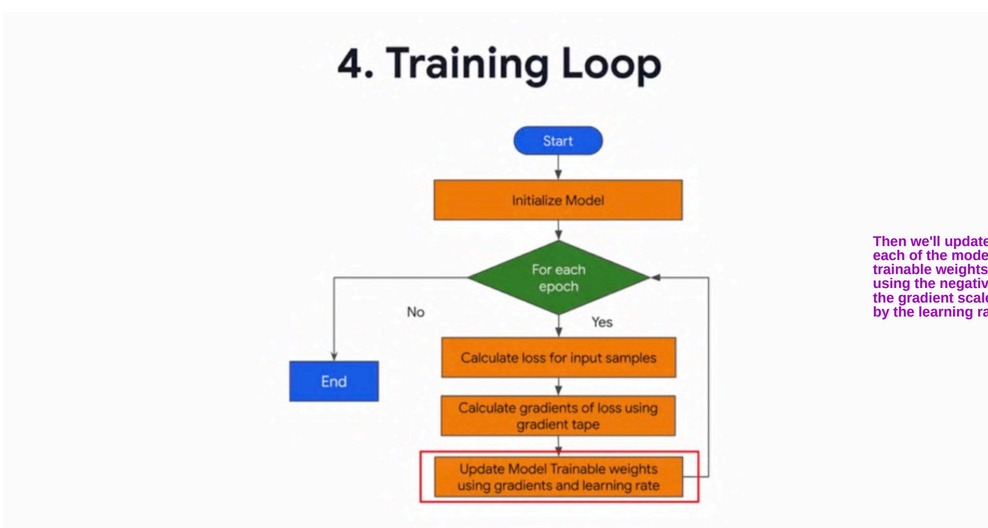
We'll initialize our model, including our trainable variables which we refer to as the weights.



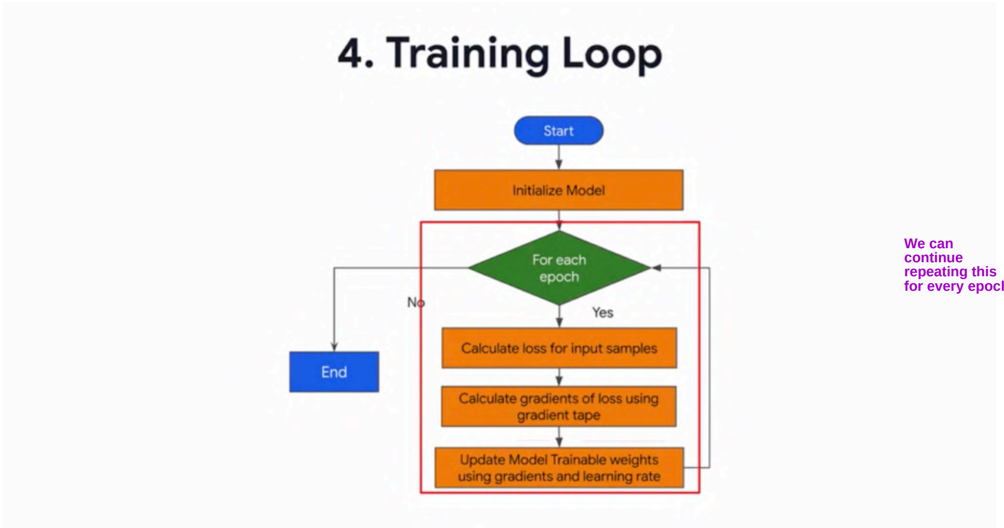
Then we'll train from a number of epochs. Within a epochs, we'll calculate the loss of the predicted values against the input samples.



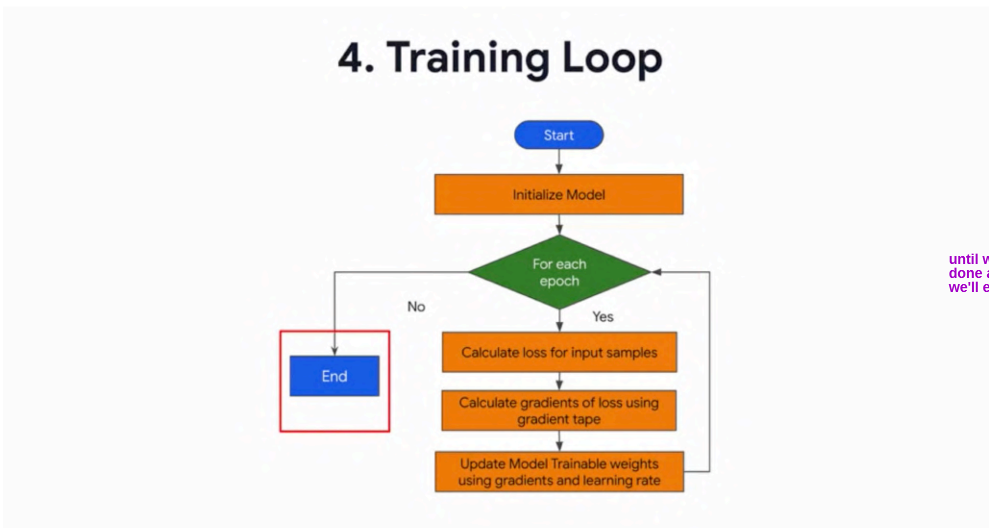
We'll calculate the gradient of the loss with respect to each of our trainable variables using a gradient tape.



Then we'll update each of the models trainable weights using the negative of the gradient scaled by the learning rate.



We can continue repeating this for every epoch



until we're done and then we'll end.

Calculate Partial Derivative of Loss

Within a training function, we do what we showed earlier, calculate the loss, get the gradients, and then update the trainable variables.

```
def train(model, inputs, outputs, learning_rate):
  with tf.GradientTape() as tape:
    current_loss = loss(outputs, model(inputs))
  da, db = tape.gradient(current_loss, [model.a, model.b])

  model.a.assign_sub(learning_rate * da)
  model.b.assign_sub(learning_rate * db)
```

Define Training Loop

```
epochs = range(20)
for epoch in epochs:
    train(model, inputs, outputs, learning_rate=0.1)
```

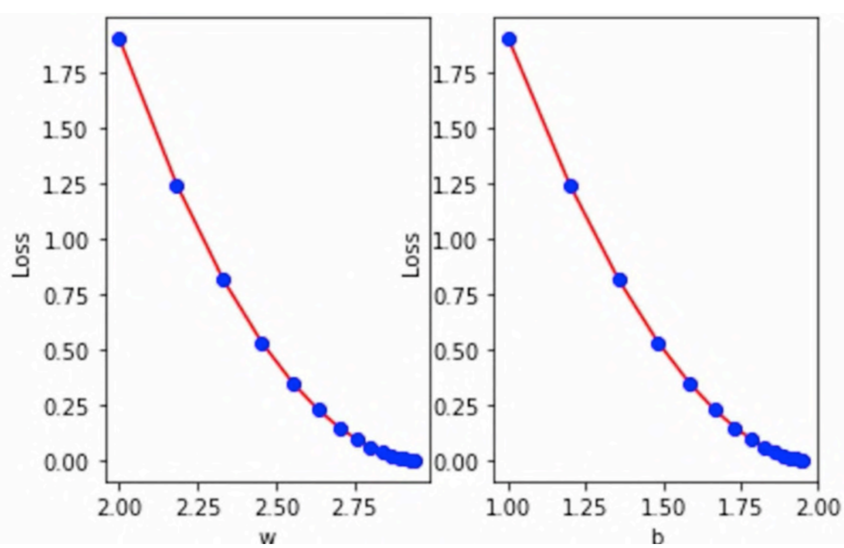
To implement the loop we showed in the flowchart, we use a fore loop that goes through our desired number of epochs on calls that's training function at each iteration.

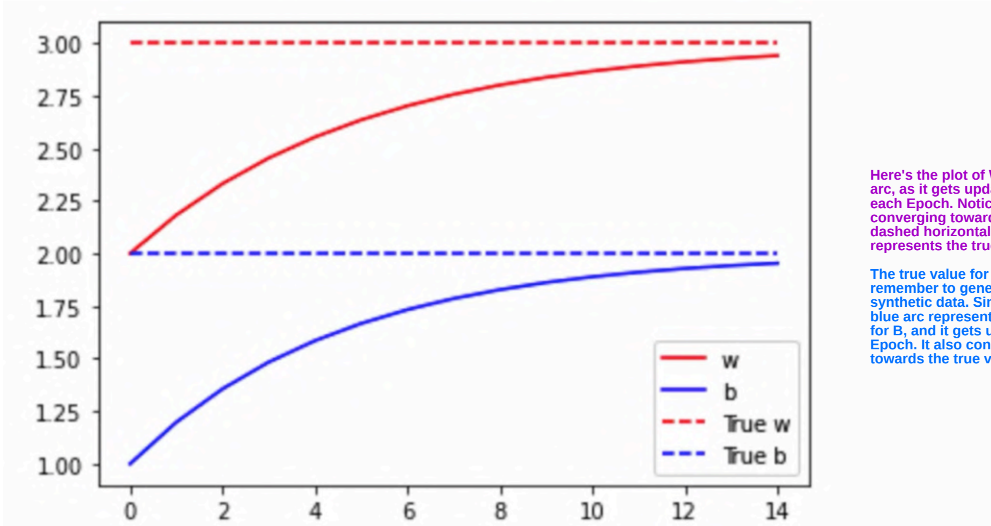
6. Validate the Model

1. Draw plots of loss for w and b over time
2. Draw plots of trainable weights over time.
3. Calculate loss

I'll also plot the values of the trainable variables as they're updated during training. Then finally, I can calculate the final loss after training ends.

First, here's the values of W and B with respect to loss. We can see that they start moving down towards the minimum. This looks a little bit like the ball moving down the curve that we showed earlier on.





Here's the plot of W, the red arc, as it gets updated during each Epoch. Notice that it's converging towards the red dashed horizontal line, which represents the true value for W.

The true value for W was used remember to generate our synthetic data. Similarly, the blue arc represents the value for B, and it gets updated every Epoch. It also converges towards the true value for B.

```
Epoch  0: w=2.00 b=1.00, loss=1.90155
Epoch  1: w=2.18 b=1.20, loss=1.24631
Epoch  2: w=2.33 b=1.36, loss=0.81714
Epoch  3: w=2.45 b=1.48, loss=0.53595
Epoch  4: w=2.55 b=1.59, loss=0.35164
Epoch  5: w=2.64 b=1.67, loss=0.23080
Epoch  6: w=2.70 b=1.73, loss=0.15153
Epoch  7: w=2.76 b=1.79, loss=0.09953
Epoch  8: w=2.80 b=1.83, loss=0.06539
Epoch  9: w=2.84 b=1.86, loss=0.04297
Epoch 10: w=2.87 b=1.89, loss=0.02825
Epoch 11: w=2.89 b=1.91, loss=0.01858
Epoch 12: w=2.91 b=1.93, loss=0.01222
Epoch 13: w=2.93 b=1.94, loss=0.00804
Epoch 14: w=2.94 b=1.95, loss=0.00529
```

Finally, here's the behavior of the network over 15 epochs. We can see the loss value that is calculated for a given W and B. First is quite large one point nine indicating that our proverbial ball is far away from the minimum.

But with each Epoch, we can see it's that closer and closer to the minimum and it ends at point zero zero five two nine on the 15th Epoch.

What we'll cover

1. Define custom training loop that takes input pipeline from Tensorflow Datasets.
2. Use pre-built loss function and optimizer within training loop
3. Use and track performance with test set
4. Handling training metrics.

Steps to training this network

1. **Define** the network
2. **Prepare** the training data pipeline
3. **Specify** Loss and Optimizer
4. **Train** the model to minimize loss using optimizer.
5. **Test** the model.

1. Define Network

```
def base_model():
  inputs = tf.keras.Input(shape=(784,), name='clothing')
  x = tf.keras.layers.Dense(64, activation='relu', name='dense_1')(inputs)
  x = tf.keras.layers.Dense(64, activation='relu', name='dense_2')(x)
  outputs = tf.keras.layers.Dense(10, activation='softmax', name='predictions')(x)
  model = tf.keras.Model(inputs=inputs, outputs=outputs)
  return model
```

2. Prepare Training Data Pipeline

1. Load Fashion MNIST using TensorFlow Datasets
2. We ***normalize*** the inputs pixels to restrict them between 0 and 1.
3. Split dataset into training and test sets.

```python
train_data = tfds.load("fashion_mnist", split = "train")
test_data = tfds.load("fashion_mnist", split = "test")
def format_image(data):
    image = data["image"]
    image = tf.reshape(image, [-1])
    image = tf.cast(image, 'float32')
    image = image / 255.0
    return image, data["label"]
train_data = train_data.map(format_image)
test_data = test_data.map(format_image)
batch_size = 64
train = train_data.shuffle(buffer_size=1024).batch(batch_size)
test = test_data.batch(batch_size=batch_size)
```

```python
train_data = tfds.load("fashion_mnist", split = "train")
test_data = tfds.load("fashion_mnist", split = "test")
def format_image(data):
    image = data["image"]
    image = tf.reshape(image, [-1])
    image = tf.cast(image, 'float32')
    image = image / 255.0
    return image, data["label"]
train_data = train_data.map(format_image)
test_data = test_data.map(format_image)
batch_size = 64
train = train_data.shuffle(buffer_size=1024).batch(batch_size)
test = test_data.batch(batch_size=batch_size)
```

Implement a function to format the data so that can be fed into the model.

First thing we'll want to do is format the image and flatten it into a one-dimensional array.

Then we'll use tf.cast to convert the pixels from integers into floating point values so that we can divide by 255, which makes all of the image values in a range from zero to one.

Then we'll return both the formatted image and its associated label as a tuple.

```python
train_data = tfds.load("fashion_mnist", split = "train")
test_data = tfds.load("fashion_mnist", split = "test")
def format_image(data):
    image = data["image"]
    image = tf.reshape(image, [-1])
    image = tf.cast(image, 'float32')
    image = image / 255.0
    return image, data["label"]
train_data = train_data.map(format_image)
test_data = test_data.map(format_image)
batch_size = 64
train = train_data.shuffle(buffer_size=1024).batch(batch_size)
test = test_data.batch(batch_size=batch_size)
```

We'll use the.map function on the training data to apply this format image function on each image in the training data. It's just a standard TFDS mapping function.

Similarly, we'll call.map to format each example in the test data.

```python
train_data = tfds.load("fashion_mnist", split = "train")
test_data = tfds.load("fashion_mnist", split = "test")
def format_image(data):
    image = data["image"]
    image = tf.reshape(image, [-1])
    image = tf.cast(image, 'float32')
    image = image / 255.0
    return image, data["label"]
train_data = train_data.map(format_image)
test_data = test_data.map(format_image)
batch_size = 64
train = train_data.shuffle(buffer_size=1024).batch(batch_size)
test = test_data.batch(batch_size=batch_size)
```

Normally when shuffling data, all the values are loaded into memory and everything is shuffled at once.

Note, however, that you might be working with training data that's too large to fit into your computer's memory all at the same time.

In order to work with large datasets, you can start with a buffer of the first 1024 examples from the training dataset and hold them in memory, and then randomly sample from that buffer. In this case, we're calling.batch with a batch size of 64, so we'll sample 64 random values one at a time from the 1024 examples that are in the buffer.

Each time an example is added to the batch, another example is taken from the dataset and placed into the buffer. The shuffle remains a buffer of 1024 examples. It keeps maintaining them. Each time we iterate on the train dataset, we'll get a batch of 64 examples randomly sampled from that buffer of 1024.

3. Define Loss and Optimizer

```python
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()

optimizer = tf.keras.optimizers.Adam()
```

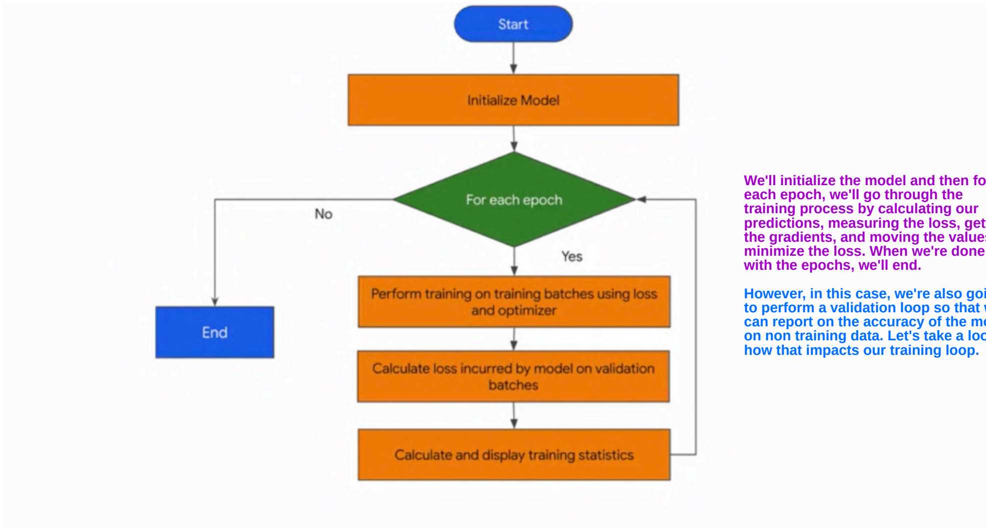
Next, we can pick our loss function and our optimizer. For the loss function, we want a function that handles categorical predictions as we're classifying 10 different categories of clothing.

We can pick sparse categorical cross entropy. We use the sparse version of categorical cross entropy when the category labels are integers and not already one-hot encoded.

The sparse version is a little bit more memory and computation efficient than the regular categorical cross entropy two. Optimizers will pick the standard Adam one.

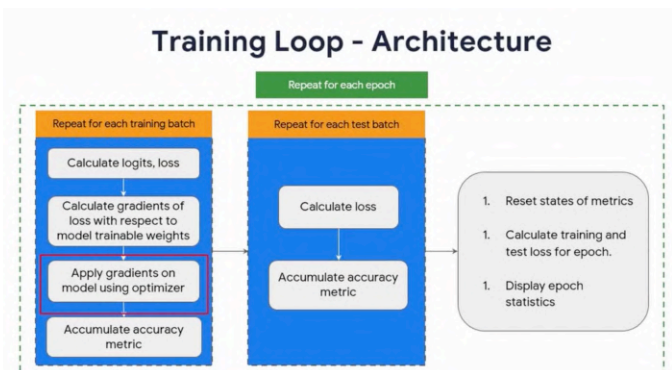
4. Define Custom Training Loop

1. For each epoch, loop through the training batches and calculate gradients
2. These gradients are used according to the optimization algorithm chosen, to update the trainable weights of the model.
3. Loop through validation batches and calculate validation loss.



We'll initialize the model and then for each epoch, we'll go through the training process by calculating our predictions, measuring the loss, getting the gradients, and moving the values to minimize the loss. When we're done with the epochs, we'll end.

However, in this case, we're also going to perform a validation loop so that we can report on the accuracy of the model on non training data. Let's take a look at how that impacts our training loop.



The from loop that performs the training over n number of epochs can further be broken down as shown here. Let's dive in a little further.

For each batch in the training data, we first calculate the logits and that's the output of the model on the current input batch, as well as the loss value for these as calculated by the loss function. We'll then calculate the gradients of loss with respect to each of the model trainable variables. We'll then update the model trainable variables using these gradients and the optimizer, and then finally, we'll calculate the accuracy metric.

This simply gives us the quotient of how many of the predictions were correct, divided by how many predictions were attempted by the model.

On each batch in the test set, we calculate the loss using the loss function, as well as the accuracy metric by counting how many predictions for the test set were correct and dividing by the number of test examples of the model predicted on.

We can then calculate the training loss for the entire epoch by taking the mean of the losses for each batch in the training set. We perform the same on the test losses to get the overall test loss for the epoch.

Finally, we display the training statistics for each epoch. After that, we can reset the states of the metrics so that they're fresh for the next epoch. Let's now look at that in code.

Define Custom Training Loop

```
model = base_model()
epochs = 20
for epoch in range(epochs):
  #Run through training batch
  losses_train = train_data_for_one_epoch()

  #Calculate validation losses and metrics.
  losses_val = perform_validation()

  losses_train_mean = np.mean(losses_train)
  losses_val_mean = np.mean(losses_val)
```

Here's the code for 20 epoch training loop. We'll start by instantiating the model. This was defined earlier in the base model function.

Define Custom Training Loop

```python
model = base_model()
epochs = 20
for epoch in range(epochs):
  #Run through training batch
  losses_train = train_data_for_one_epoch()

  #Calculate validation losses and metrics.
  losses_val = perform_validation()

  losses_train_mean = np.mean(losses_train)
  losses_val_mean = np.mean(losses_val)
```

We'll then set up our loop to run for 20 epochs

Define Custom Training Loop

```python
model = base_model()
epochs = 20
for epoch in range(epochs):
  #Run through training batch
  losses_train = train_data_for_one_epoch()

  #Calculate validation losses and metrics.
  losses_val = perform_validation()

  losses_train_mean = np.mean(losses_train)
  losses_val_mean = np.mean(losses_val)
```

The training will take place in the train data for one epoch function that we'll see shortly. This will return the set of losses for the training data for the current epoch.

Define Custom Training Loop

```python
model = base_model()
epochs = 20
for epoch in range(epochs):
  #Run through training batch
  losses_train = train_data_for_one_epoch()

  #Calculate validation losses and metrics.
  losses_val = perform_validation()

  losses_train_mean = np.mean(losses_train)
  losses_val_mean = np.mean(losses_val)
```

To validate on the test set, we'll create a perform validation function, which you'll also see shortly. This will return the losses value for the current epoch.

Define Custom Training Loop

```
model = base_model()
epochs = 20
for epoch in range(epochs):
  #Run through training batch
  losses_train = train_data_for_one_epoch()

  #Calculate validation losses and metrics.
  losses_val = perform_validation()

  losses_train_mean = np.mean(losses_train)
  losses_val_mean = np.mean(losses_val)
```

We can then calculate the mean of the losses on both training and validation sets using these values, so we can report an epoch by epoch loss progress.

Define Custom Training Loop

Earlier we called a function called train data for one epoch that perform the training step. And here's the code for it. Let's look at it step-by-step.

We're using batch data when training this network so we can create a list of losses where we'll append the loss for each batch.

```
def train_data_for_one_epoch():
  losses = []
  for step, (x_batch_train, y_batch_train) in enumerate(train_datset):
      logits, loss_value = apply_gradient(optimizer, model, x_batch_train, y_batch_train)
      losses.append(loss_value)
  return losses
```

Define Custom Training Loop

```
def train_data_for_one_epoch():
  losses = []
  for step, (x_batch_train, y_batch_train) in enumerate(train_datset):
      logits, loss_value = apply_gradient(optimizer, model, x_batch_train, y_batch_train)
      losses.append(loss_value)
  return losses
```

As our data is in batches, we have to train in a number of steps. At each iteration of the for loop, the train data set iterator yields a batch of 64 examples, which is the tuple of x batch train and y batch train.

We're using enumerate to handle the batches for us, which also returns an integer stored in step to track the batch number at each loop. The step integer is used also when printing out the status of this step.

Define Custom Training Loop

For each step, we'll call apply gradient, giving it the optimizer, the model, and the current training batch.

This will give us back the logits and the loss value. This is a custom function that you'll write shortly.

```
def train_data_for_one_epoch():
  losses = []
  for step, (x_batch_train, y_batch_train) in enumerate(train_datset):
      logits, loss_value = apply_gradient(optimizer, model, x_batch_train, y_batch_train)
      losses.append(loss_value)
  return losses
```

Define Custom Training Loop

We'll then append the loss value for this batch to the losses array

```
def train_data_for_one_epoch():
  losses = []
  for step, (x_batch_train, y_batch_train) in enumerate(train_datset):
      logits, loss_value = apply_gradient(optimizer, model, x_batch_train, y_batch_train)
      losses.append(loss_value)
  return losses
```

Define Custom Training Loop

```
def train_data_for_one_epoch():
  losses = []
  for step, (x_batch_train, y_batch_train) in enumerate(train_datset):
      logits, loss_value = apply_gradient(optimizer, model, x_batch_train, y_batch_train)
      losses.append(loss_value)
  return losses
```

When the loop is done, we'll return the losses. Remember earlier that the training loop that calls this function then averages these out to get the overall loss.

Do note here that if your batch size doesn't divide into the training set evenly, that the final batch will have a different size from the other batches and that can skew the overall average, maybe only a little bit, but it will still be skewed.

For example, M-Nest has 60,000 items in the training set. If you had for example, used a batch size of 25, 000, then you'd have three batches, 25, 000, 25,000, and 10,000.

The overall average loss would be biased in favor of the 10,000 batch using this methodology.

Calculate and Apply Gradients

```
def apply_gradient(optimizer, model, x, y):
  with tf.GradientTape() as tape:
    logits = model(x)
    loss_value = loss_object(y_true=y, y_pred=logits)

  gradients = tape.gradient(loss_value, model.trainable_weights)
  optimizer.apply_gradients(zip(gradients, model.trainable_weights))

  return logits, loss_value
```

Start by defining gradient tape. This will allow us to calculate a gradient for all of the trainable weights in the model.

Calculate and Apply Gradients

```
def apply_gradient(optimizer, model, x, y):
  with tf.GradientTape() as tape:
    logits = model(x)
    loss_value = loss_object(y_true=y, y_pred=logits)

  gradients = tape.gradient(loss_value, model.trainable_weights)
  optimizer.apply_gradients(zip(gradients, model.trainable_weights))

  return logits, loss_value
```

We will then pass the x values, in other words training data, into the model and get the logits back.

We can compare these to the true values.

Calculate and Apply Gradients

```
def apply_gradient(optimizer, model, x, y):
  with tf.GradientTape() as tape:
    logits = model(x)
    loss_value = loss_object(y_true=y, y_pred=logits)

  gradients = tape.gradient(loss_value, model.trainable_weights)
  optimizer.apply_gradients(zip(gradients, model.trainable_weights))

  return logits, loss_value
```

Then we can get loss values by calling the loss function with the true values and the logits we just got back.

Calculate and Apply Gradients

We can calculate the gradients for each of the trainable weights in the model by differentiating them against the loss.

```
def apply_gradient(optimizer, model, x, y):
  with tf.GradientTape() as tape:
    logits = model(x)
    loss_value = loss_object(y_true=y, y_pred=logits)

  gradients = tape.gradient(loss_value, model.trainable_weights)
  optimizer.apply_gradients(zip(gradients, model.trainable_weights))

  return logits, loss_value
```

Calculate and Apply Gradients

We can then use the optimizer to update. Model's trainable weights using the calculated gradients

Note if the batch has 64 examples, the gradients variable contains 64 sets of gradients. One for each set of trainable variables.

To line up the array of 64 gradients with the 64 trainable variables that the gradients will update, we can use Python's zip function.

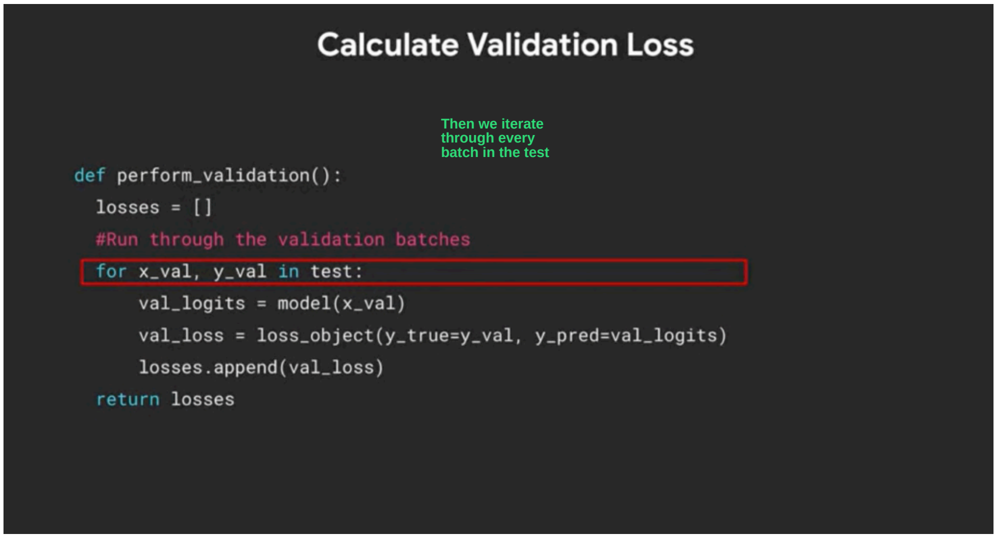
```
def apply_gradient(optimizer, model, x, y):
  with tf.GradientTape() as tape:
    logits = model(x)
    loss_value = loss_object(y_true=y, y_pred=logits)

  gradients = tape.gradient(loss_value, model.trainable_weights)
  optimizer.apply_gradients(zip(gradients, model.trainable_weights))

  return logits, loss_value
```

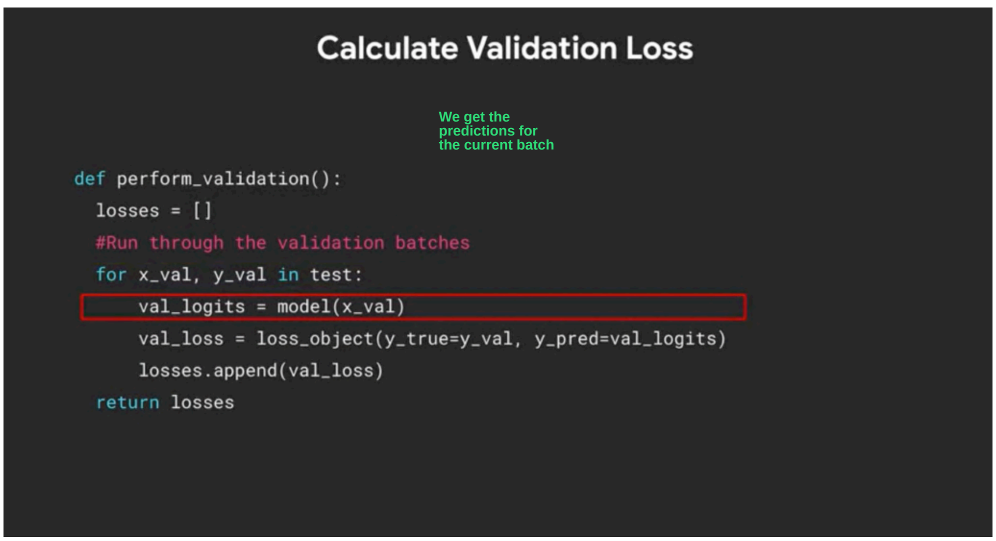
Calculate Validation Loss

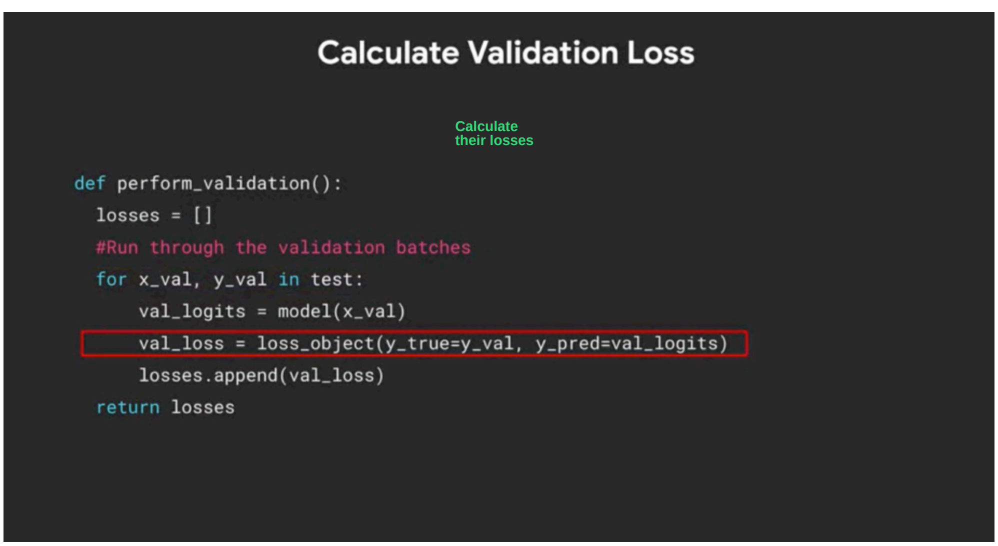
The validation calculation is then simple.

We create an array to hold all the losses.

```
def perform_validation():
  losses = []
  #Run through the validation batches
  for x_val, y_val in test:
      val_logits = model(x_val)
      val_loss = loss_object(y_true=y_val, y_pred=val_logits)
      losses.append(val_loss)
  return losses
```


Calculate Validation Loss
Then we iterate through every batch in the test
def perform_validation():
losses = []
#Run through the validation batches
for x_val, y_val in test:
val_logits = model(x_val)
val_loss = loss_object(y_true=y_val, y_pred=val_logits)
losses.append(val_loss)
return losses


Calculate Validation Loss
We get the predictions for the current batch
def perform_validation():
losses = []
#Run through the validation batches
for x_val, y_val in test:
val_logits = model(x_val)
val_loss = loss_object(y_true=y_val, y_pred=val_logits)
losses.append(val_loss)
return losses


Calculate Validation Loss
Calculate their losses
def perform_validation():
losses = []
#Run through the validation batches
for x_val, y_val in test:
val_logits = model(x_val)
val_loss = loss_object(y_true=y_val, y_pred=val_logits)
losses.append(val_loss)
return losses

Calculate Validation Loss

Append the results of that to the array of losses.

```
def perform_validation():
  losses = []
  #Run through the validation batches
  for x_val, y_val in test:
      val_logits = model(x_val)
      val_loss = loss_object(y_true=y_val, y_pred=val_logits)
      losses.append(val_loss)
  return losses
```

Calculate Validation Loss

Return it to the caller when finished.

If you recall, this was called form within the training loop and the validation losses were averaged out to report on the overall validation loss

```
def perform_validation():
  losses = []
  #Run through the validation batches
  for x_val, y_val in test:
      val_logits = model(x_val)
      val_loss = loss_object(y_true=y_val, y_pred=val_logits)
      losses.append(val_loss)
  return losses
```

Metrics in Keras

- Metrics can be modelled as function or class.
- Defined in *tf.keras.metrics*
  - mean_squared_error(...) class MeanSquaredError
  - mean_absolute_error(...) class MeanAbsoluteError

https://www.tensorflow.org/api_docs/python/tf/keras/metrics

Low Level Handling of Metrics

1. Call ***metric.update_state()*** to accumulate metric statistics after each batch.
2. Call ***metric.result*** to get current value of metric for display.
3. Call ***metric.reset_state()*** to reset metric value typically at end of epoch.

Low Level Handling of Metrics

1. Call ***metric.update_state()*** to accumulate metric statistics after each batch.
2. Call ***metric.result*** to get current value of metric for display.
3. Call ***metric.reset_state()*** to reset metric value typically at end of epoch.

Low Level Handling of Metrics

1. Call ***metric.update_state()*** to accumulate metric statistics after each batch.
2. Call ***metric.result*** to get current value of metric for display.
3. Call ***metric.reset_state()*** to reset metric value typically at end of epoch.

Low Level Handling of Metrics in Practice

```
train_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()
val_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()
```

Low Level Handling of Metrics - Training

If you recall the train data for one epoch function you created earlier that handled the training, we can then simply update the state of the training accuracy metric by calling its update state, sending it the ground truth labels, in this case y_batch_train, and the current predictions, logits.

The metric object will then do the rest by calculating loss, accuracy, ect.

```
def train_data_for_one_epoch():
  losses = []
  for step, (x_batch_train, y_batch_train) in enumerate(train_datset):
      ...
      #Accumulate metrics
      train_acc_metric.update_state(y_batch_train, logits)

  return losses
```

Low Level Handling of Metrics - Training

In the calling loop, once the epoch is complete, we can read the results from the training accuracy metrics to see how we did across all batches before finally resetting the metric so it's ready for the next epoch.

```
for epoch in range(epochs):
  #Run through training batch
  losses_train = train_data_for_one_epoch()
  …
  train_acc = train_acc_metric.result()
  train_acc_metric.reset_states()
  ...
```

Low Level Handling of Metrics - Validation

Similarly, in the perform_validation() loop, we can add an update state to the validation accuracy metric giving it the ground truth, y_val, and predictions, logits, and it can then handle all of the calculations for us.

```
def perform_validation():
  losses = []
  for x_val, y_val in test_dataset:
      logits = model(x_val)
         ...
      #Accumulate metrics
      val_acc_metric.update_state(y_val, logits)

  return losses
```

Low Level Handling of Metrics - Validation

Then in the main training loop, after we have done validation, we can load a variable val_acc with the result of the metric, and then reset the metric so it's ready to go into the next loop.

```
for epoch in range(epochs):
  #Run through training batch
  losses_val = perform_validation()
  …
  val_acc = val_acc_metric.result()
  val_acc_metric.reset_states()
  ...
```

6. Validate the Model

1. Show training progress and calculate loss and accuracy for each epoch.
2. Draw plots for loss function.
3. Visualize performance on test data.