Copyright Notice

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

model.compile()
model.fit()

→

<u>Custom Training</u>
Manage batches
Calculate loss
Minimize loss
Update weights

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.

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.

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.

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.

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.

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

```
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
```

2. Prepare Training Data

```
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
```

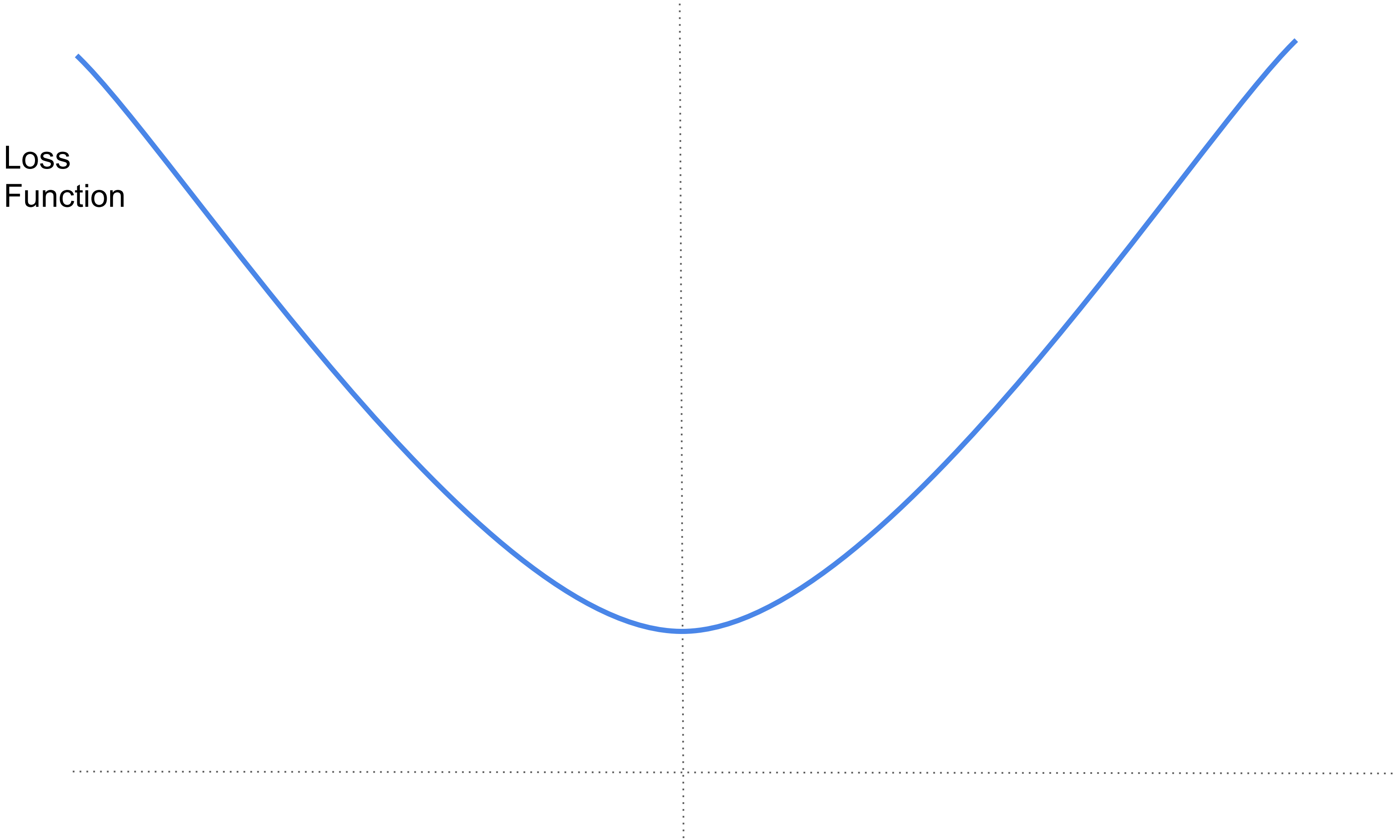
Mean Squared Error Loss

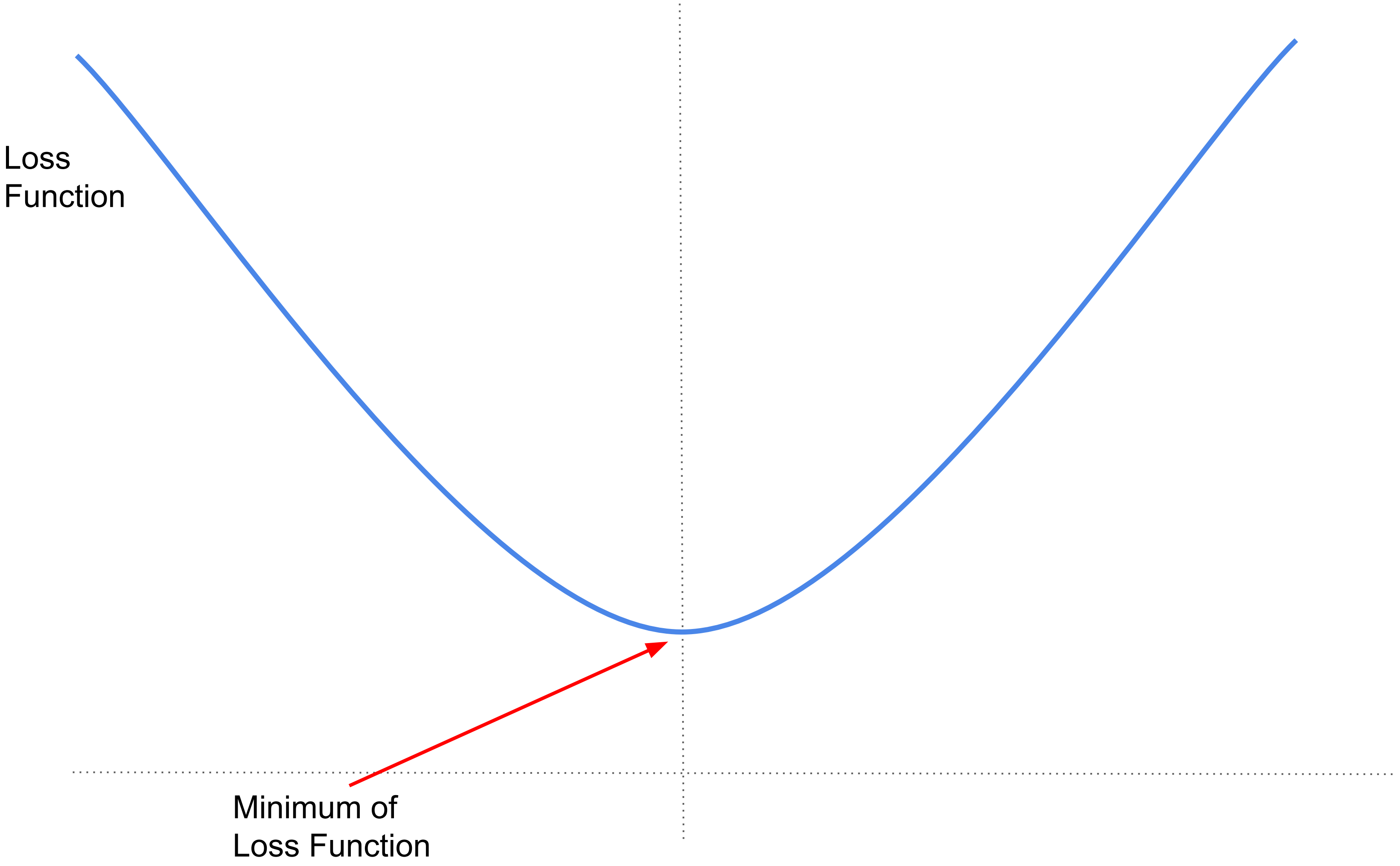
$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - y_i^{'}\right)^2$$

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

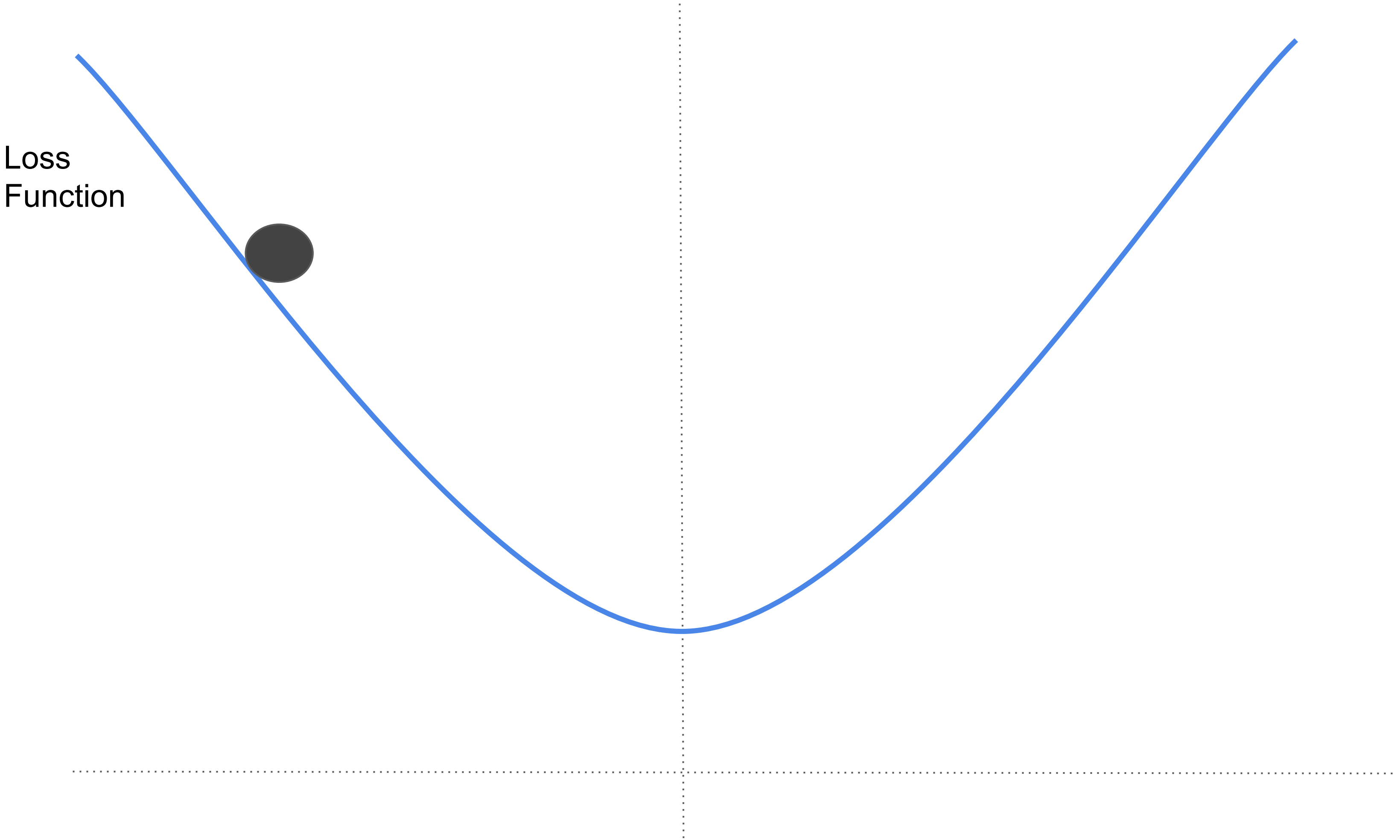
3. Mean Squared Error Loss

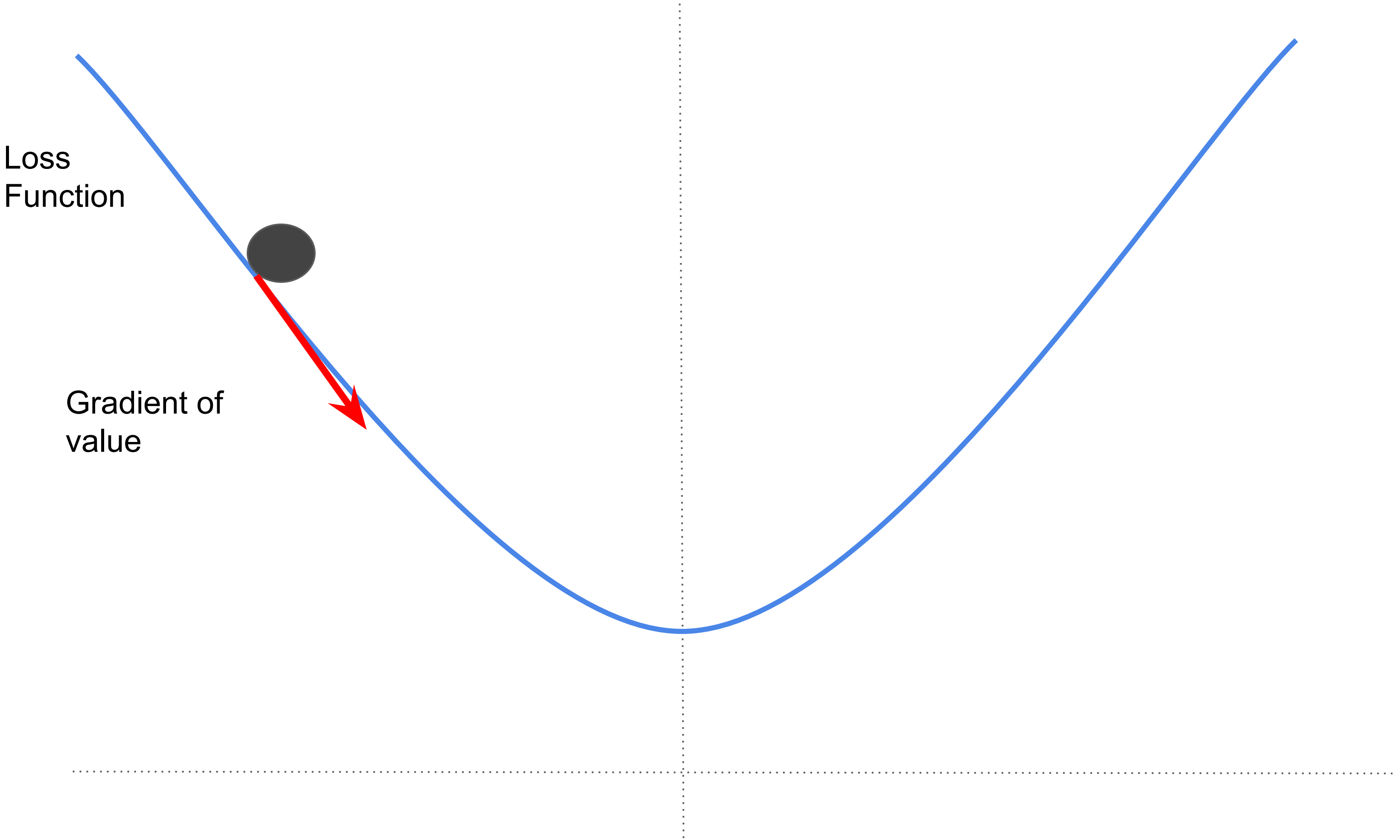
```
def loss(y_true, y_pred):
  return tf.reduce_mean(tf.square(y_true - y_pred))
```

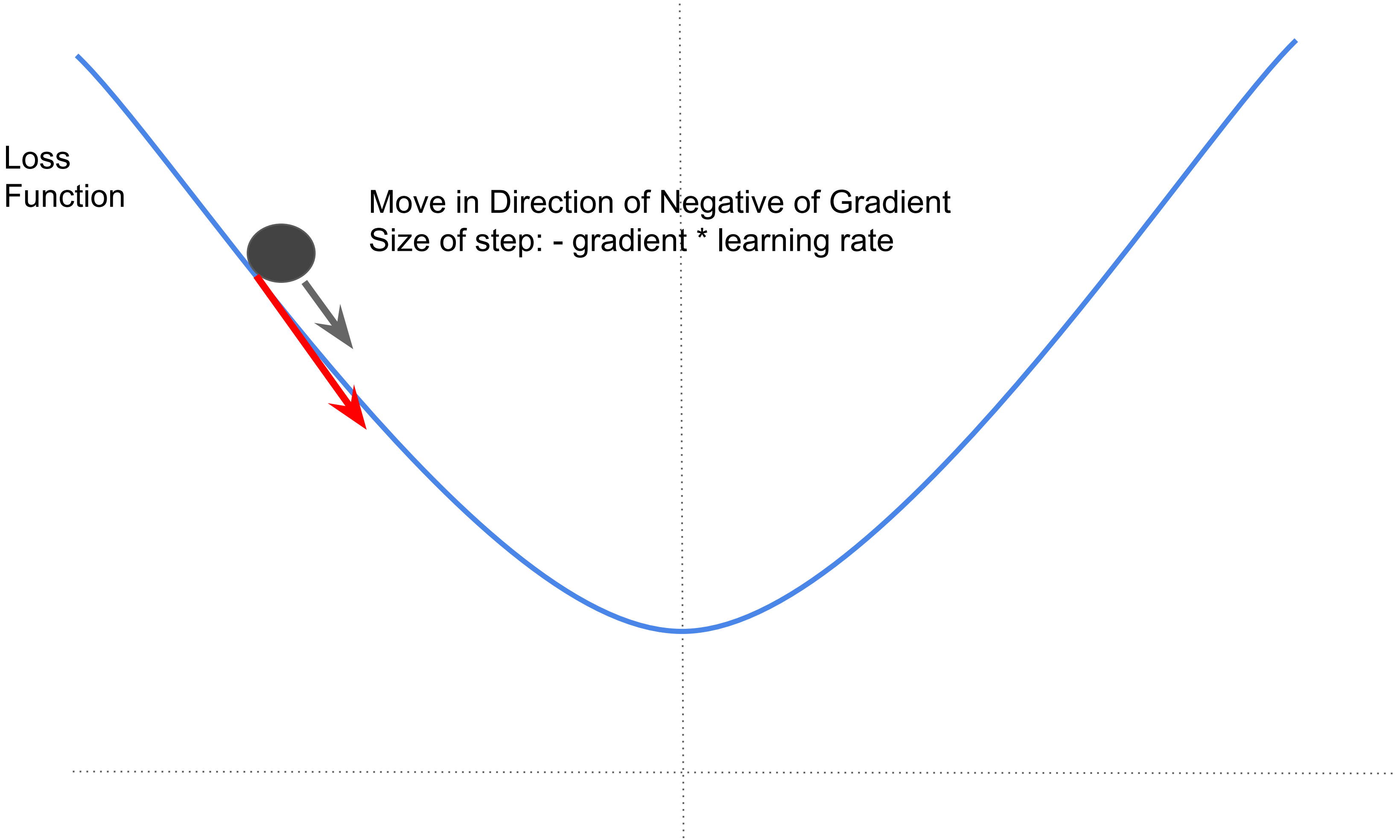

Loss
Function

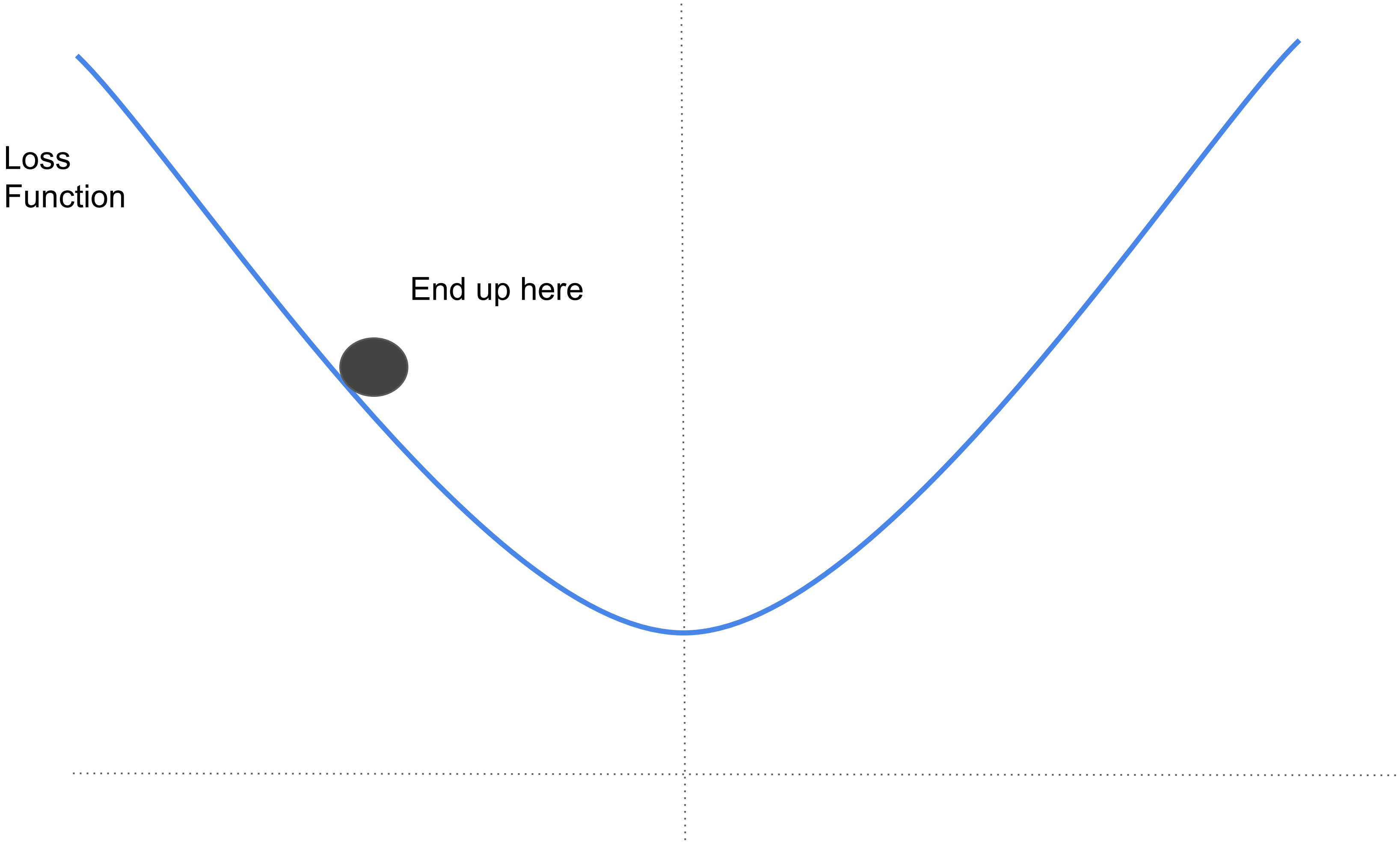


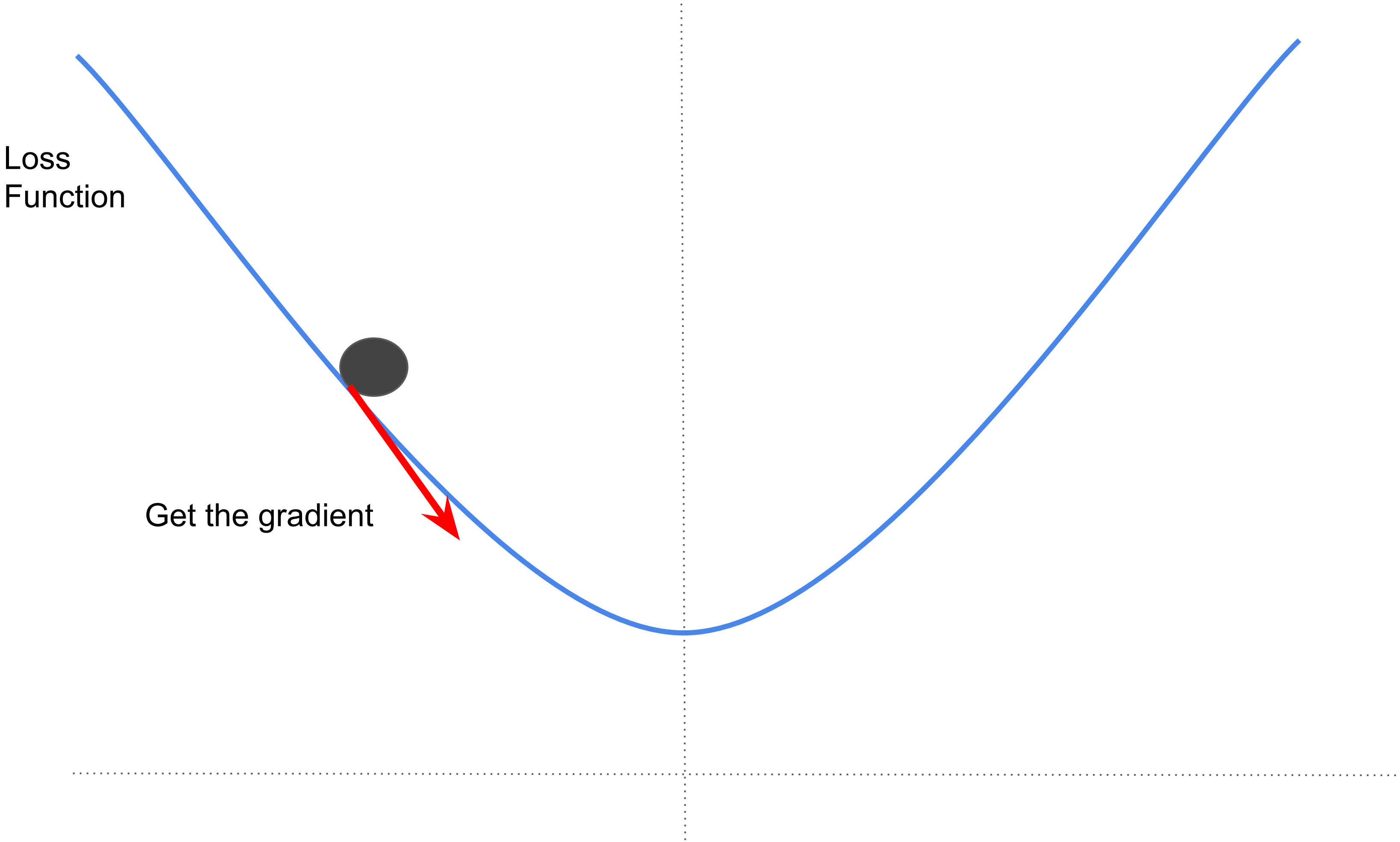
Loss
Function
Minimum of
Loss Function

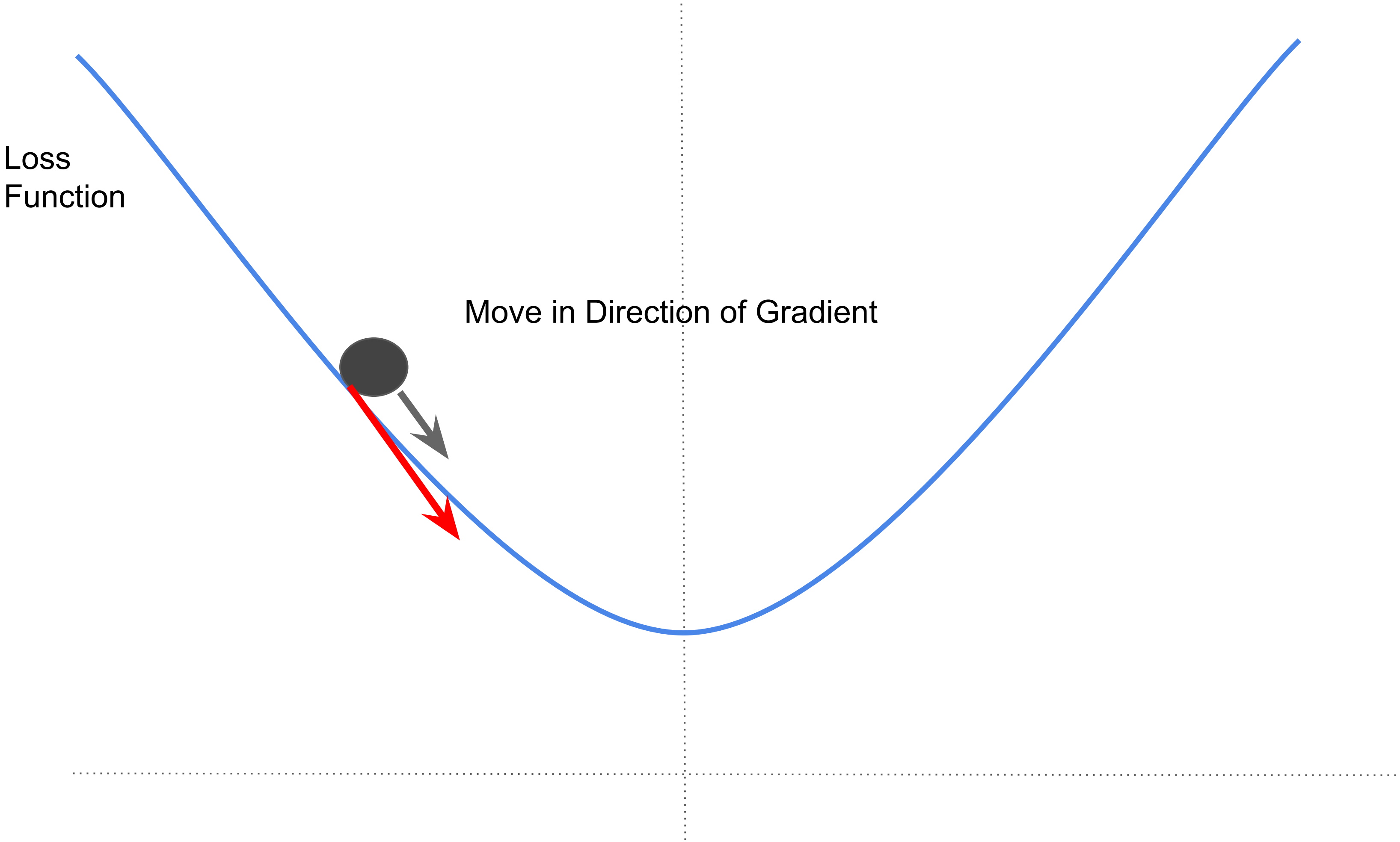

Loss
Function

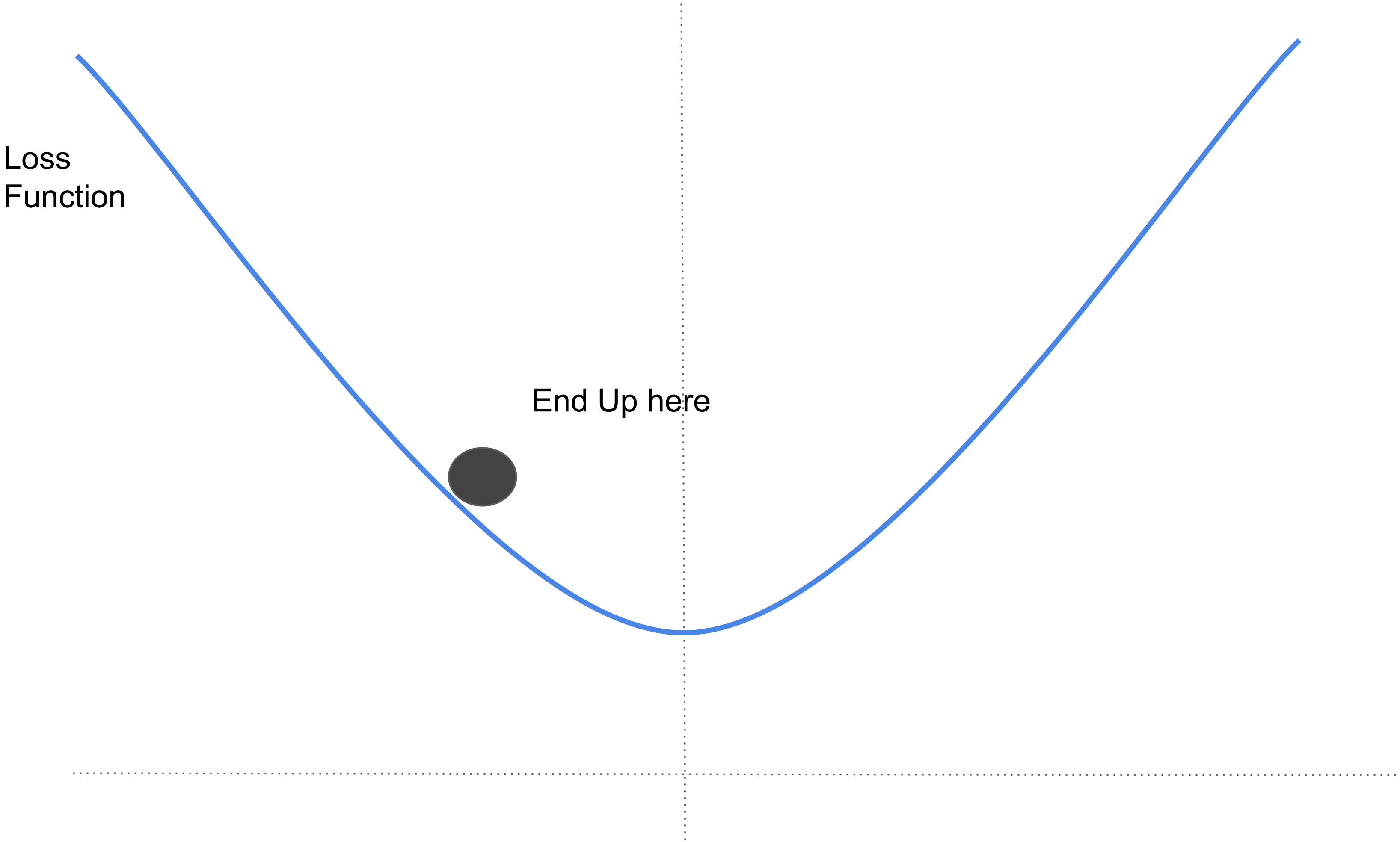

Loss
Function
Gradient of
value

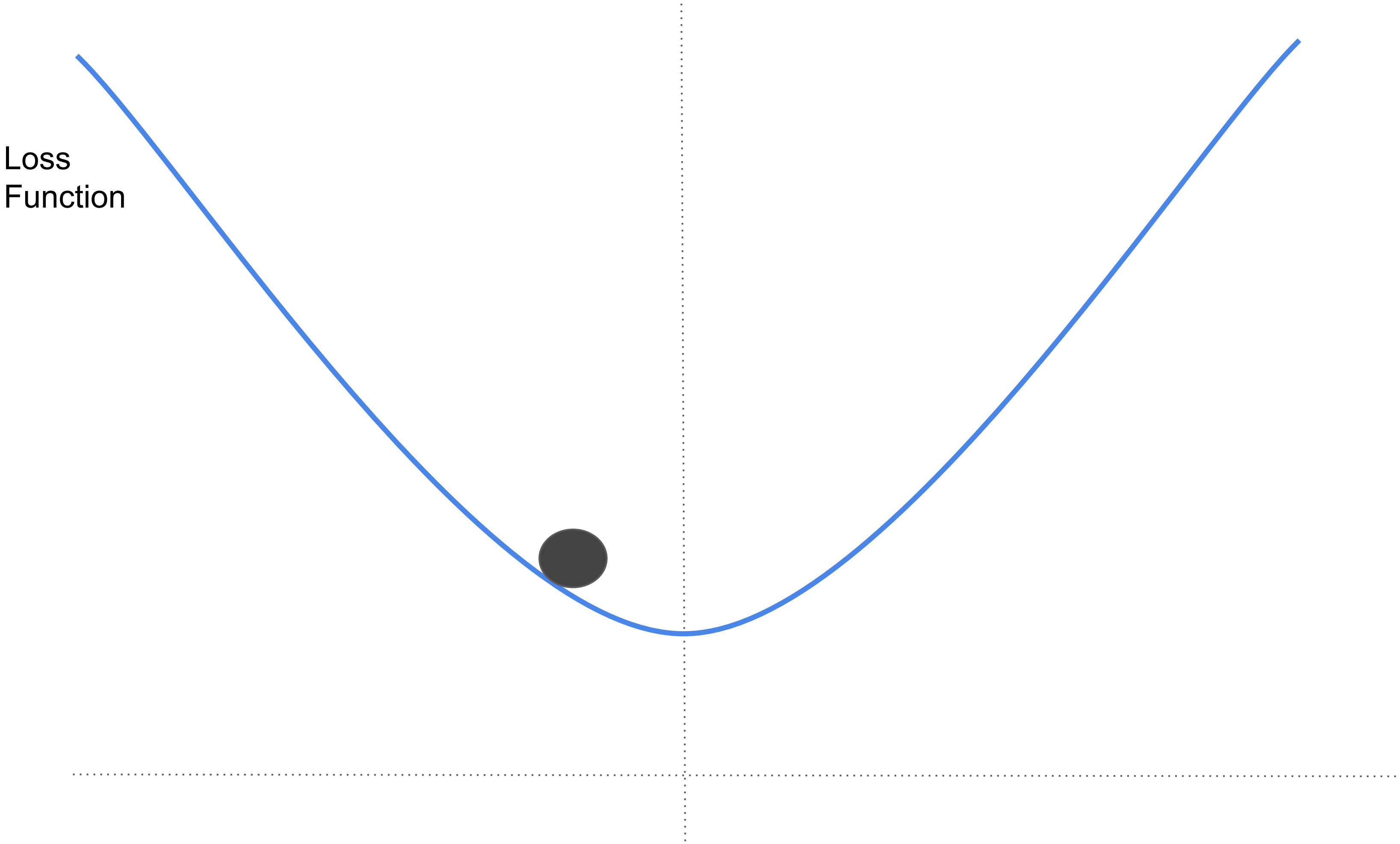

Loss
Function
Move in Direction of Negative of Gradient
Size of step: - gradient * learning rate

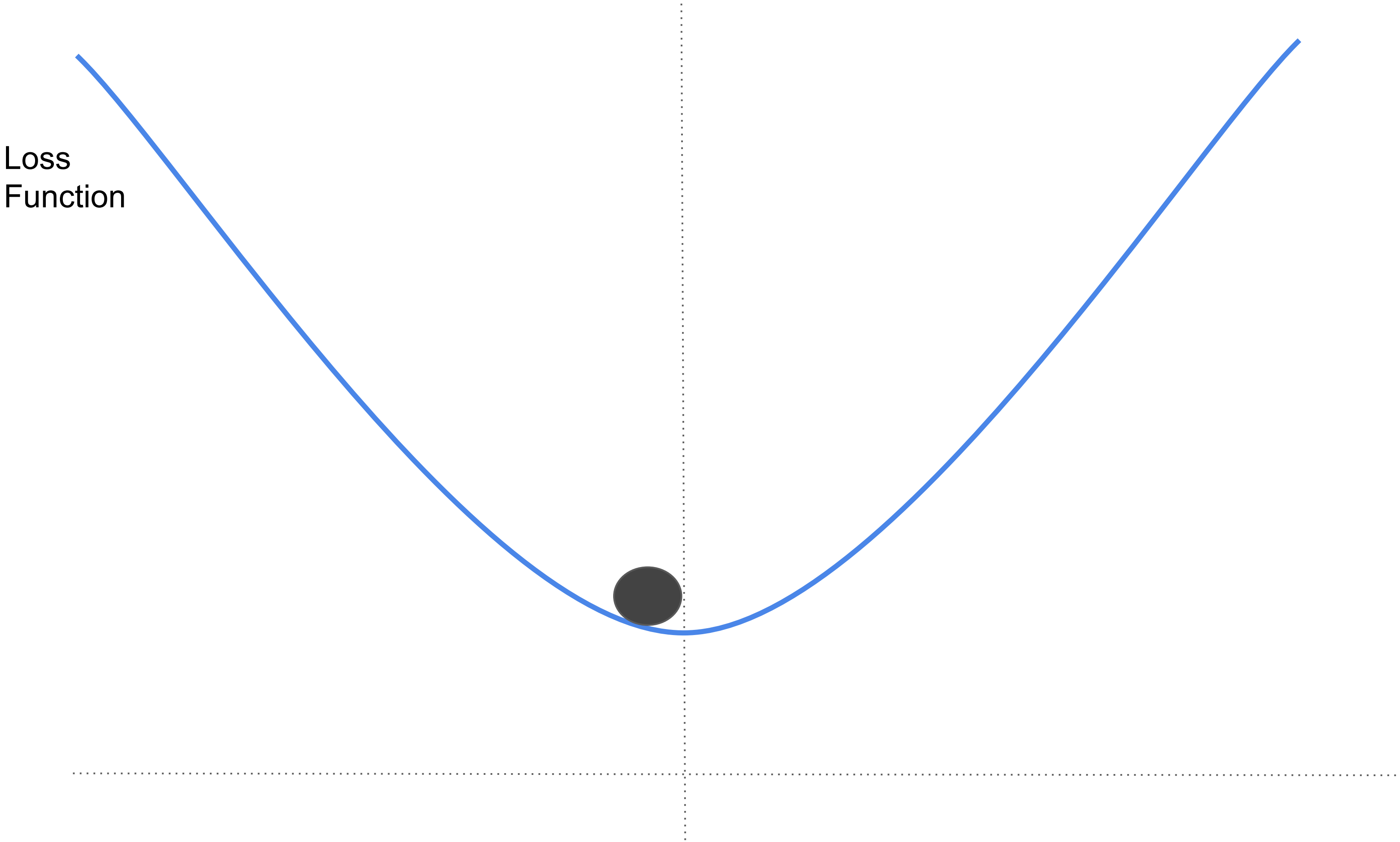

Loss
Function
End up here

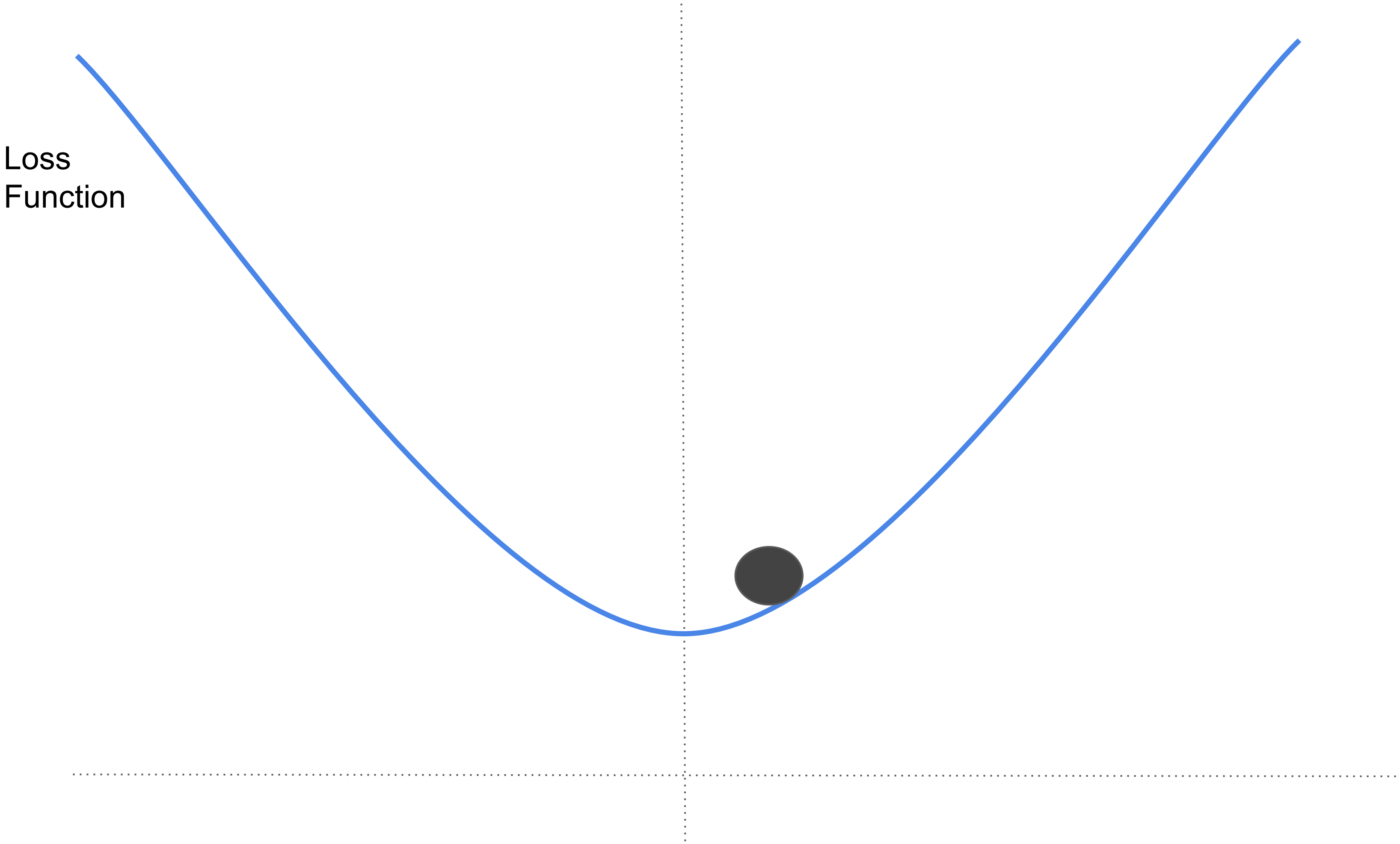

Loss
Function
Get the gradient

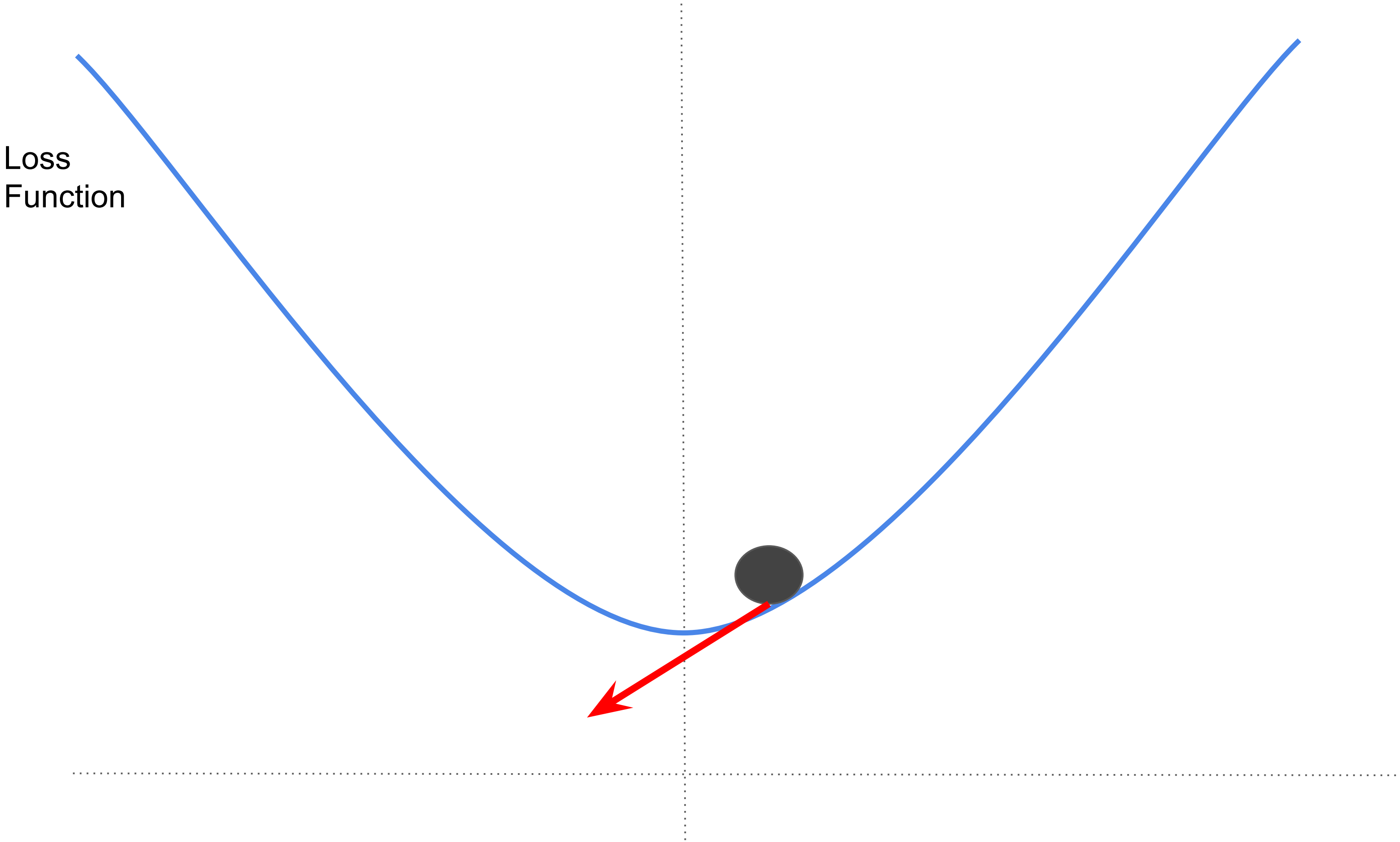

Loss
Function
Move in Direction of Gradient

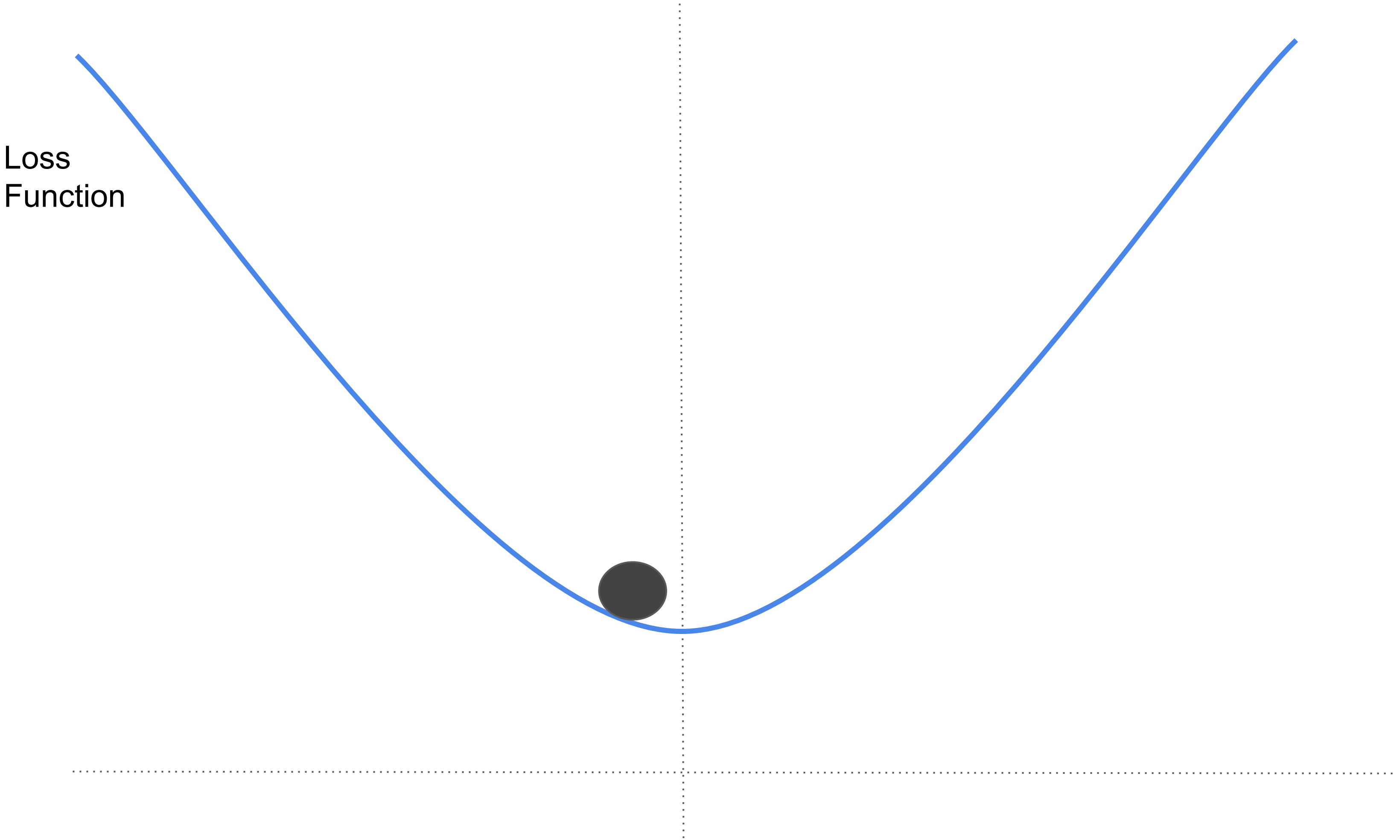

Loss
Function
End Up here

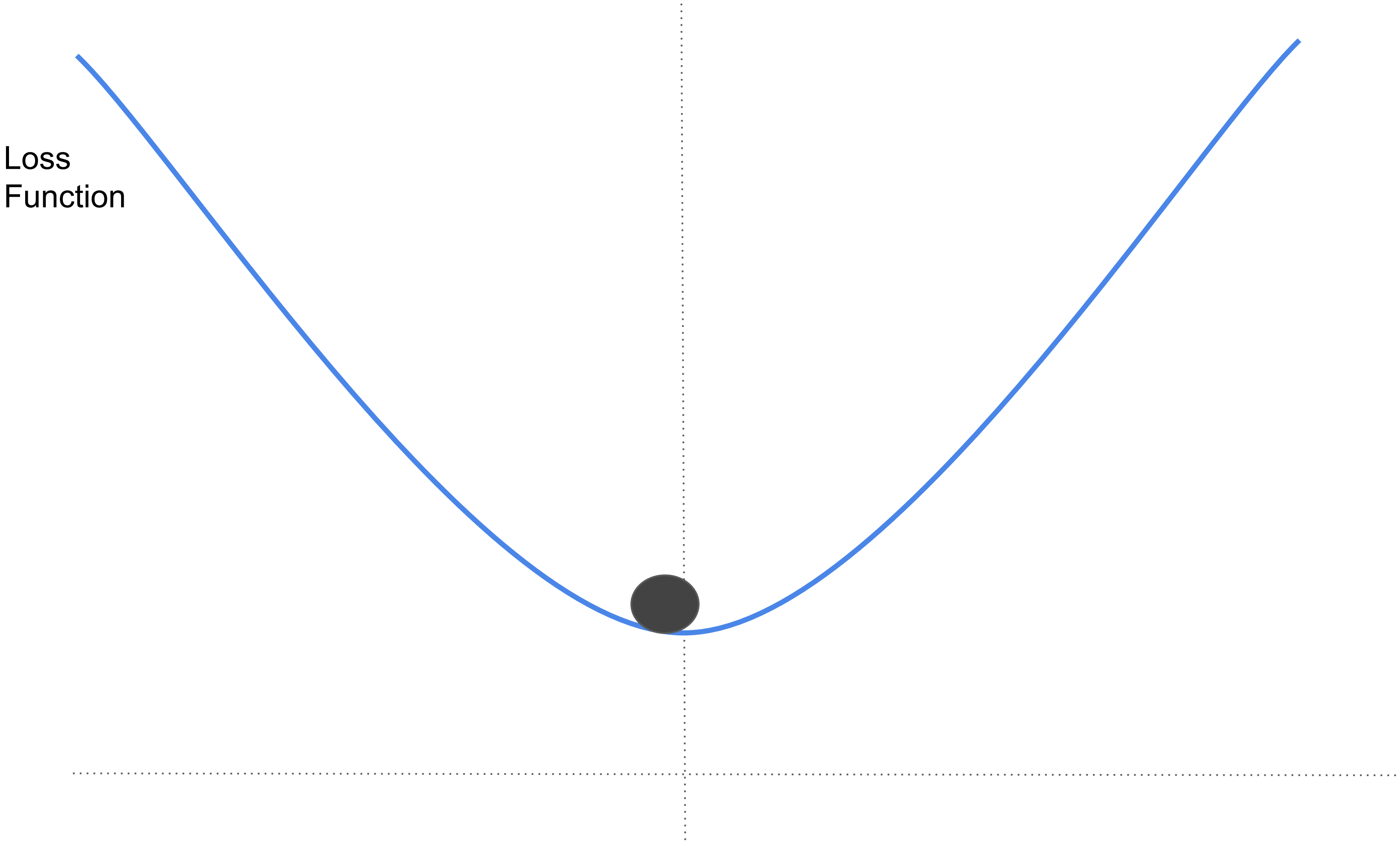

Loss
Function

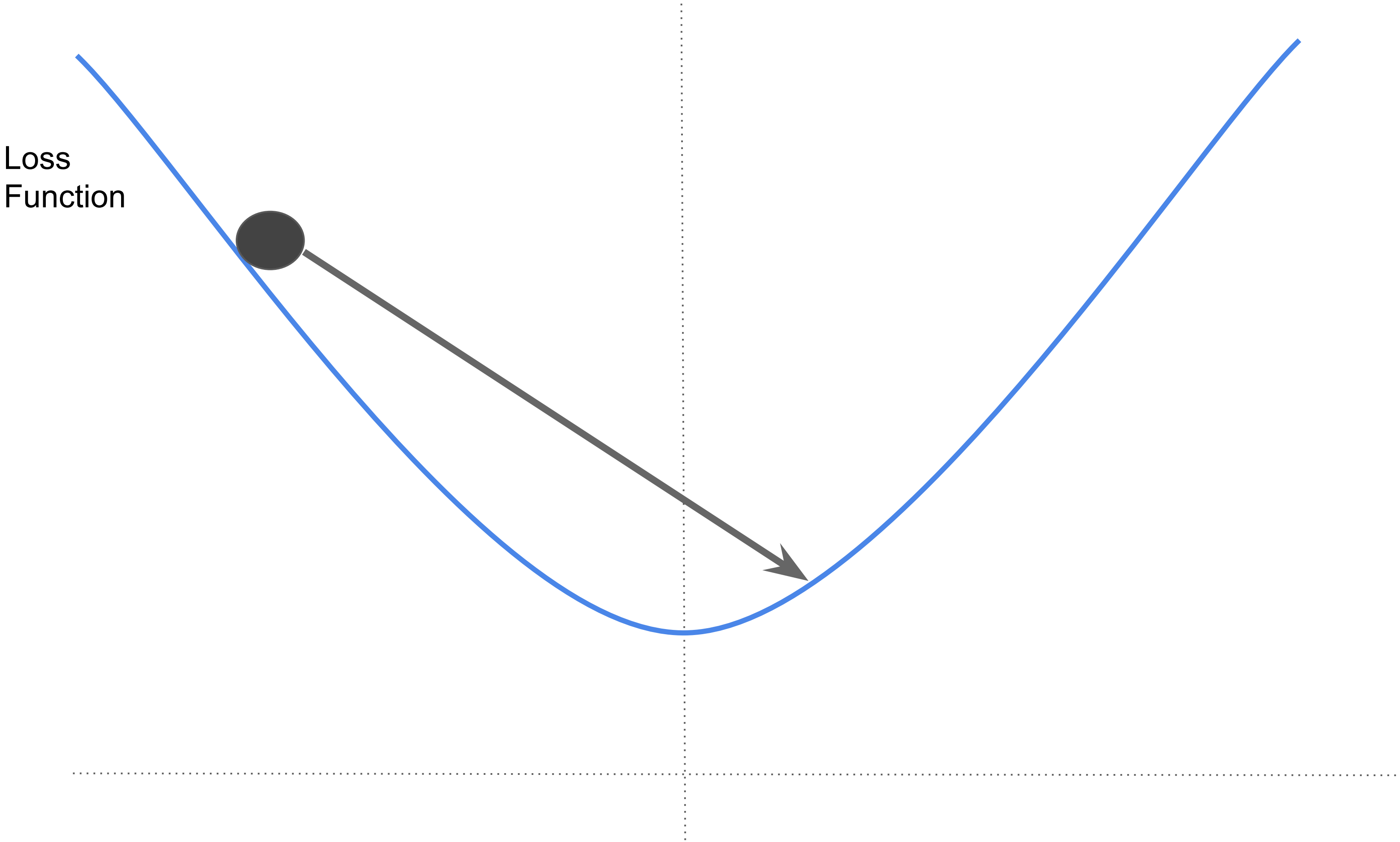

Loss
Function

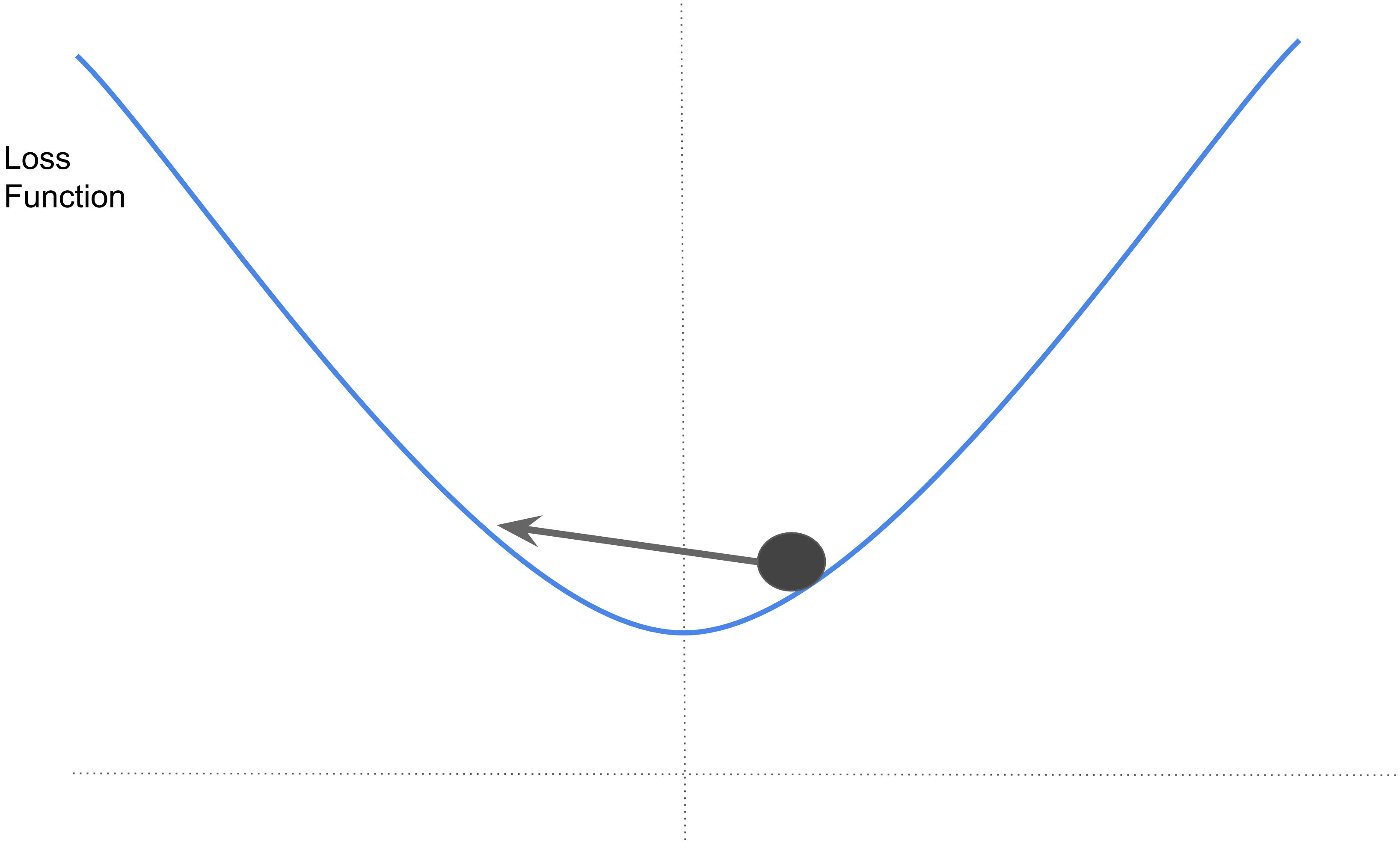

Loss
Function

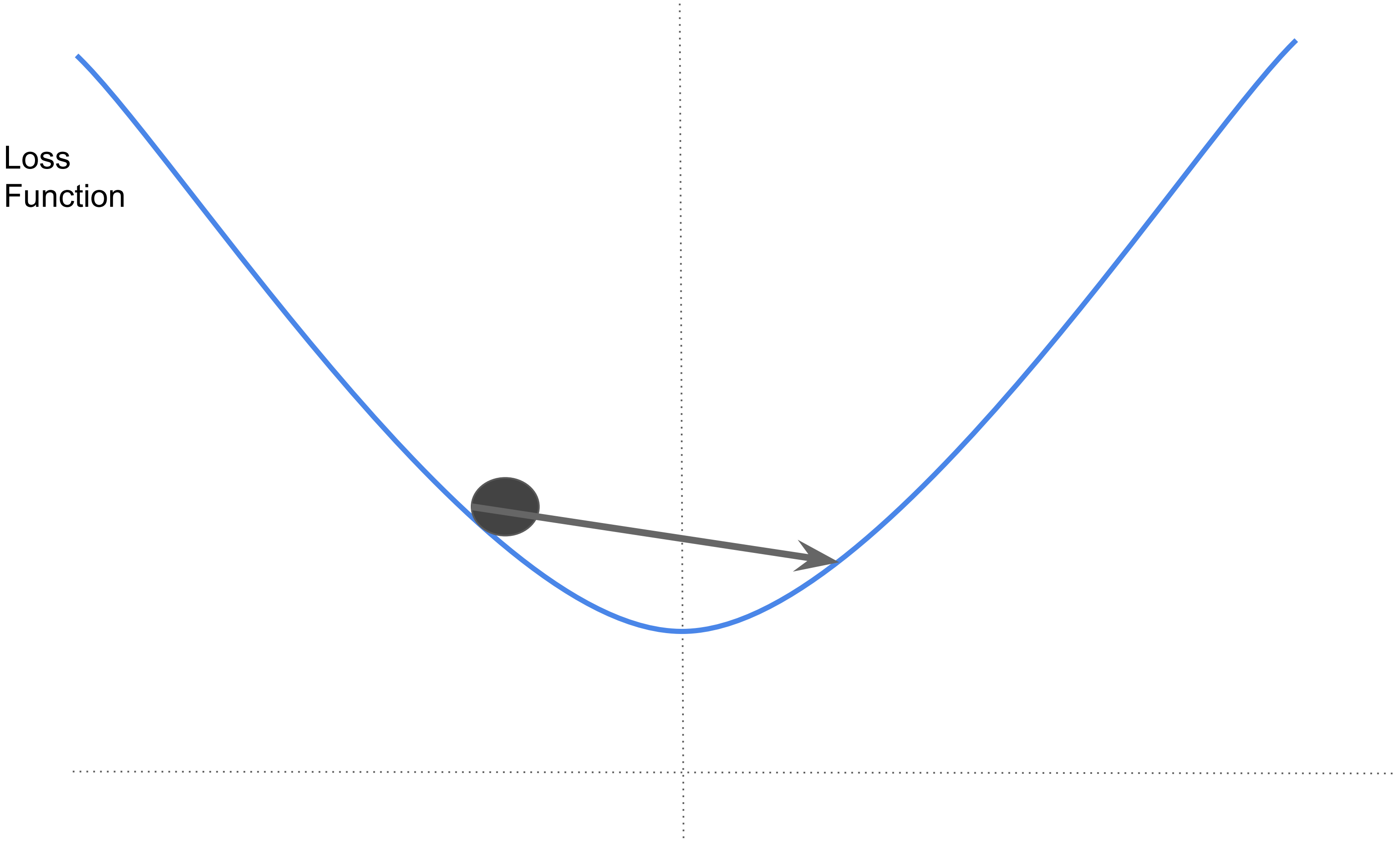

Loss
Function

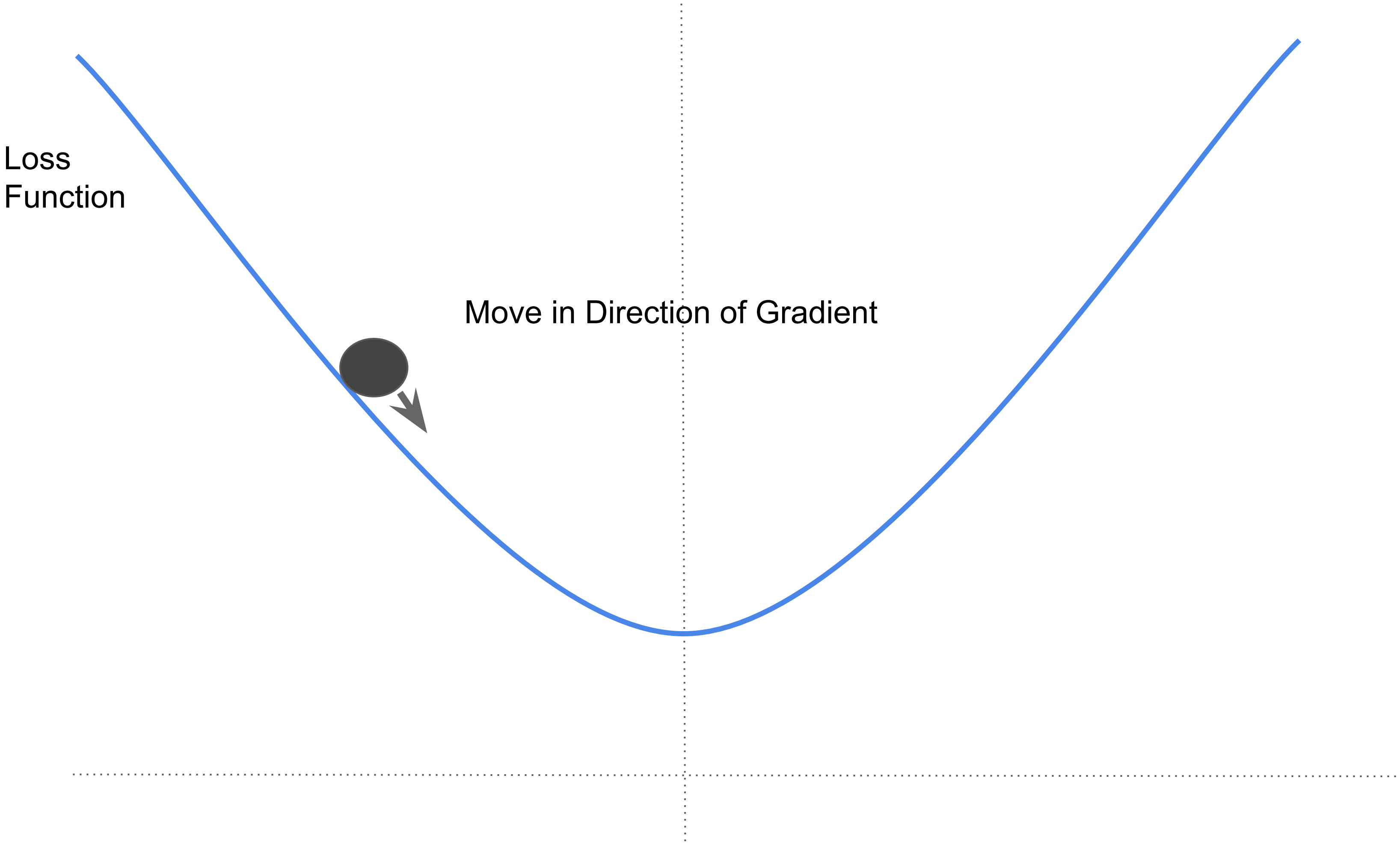

Loss
Function

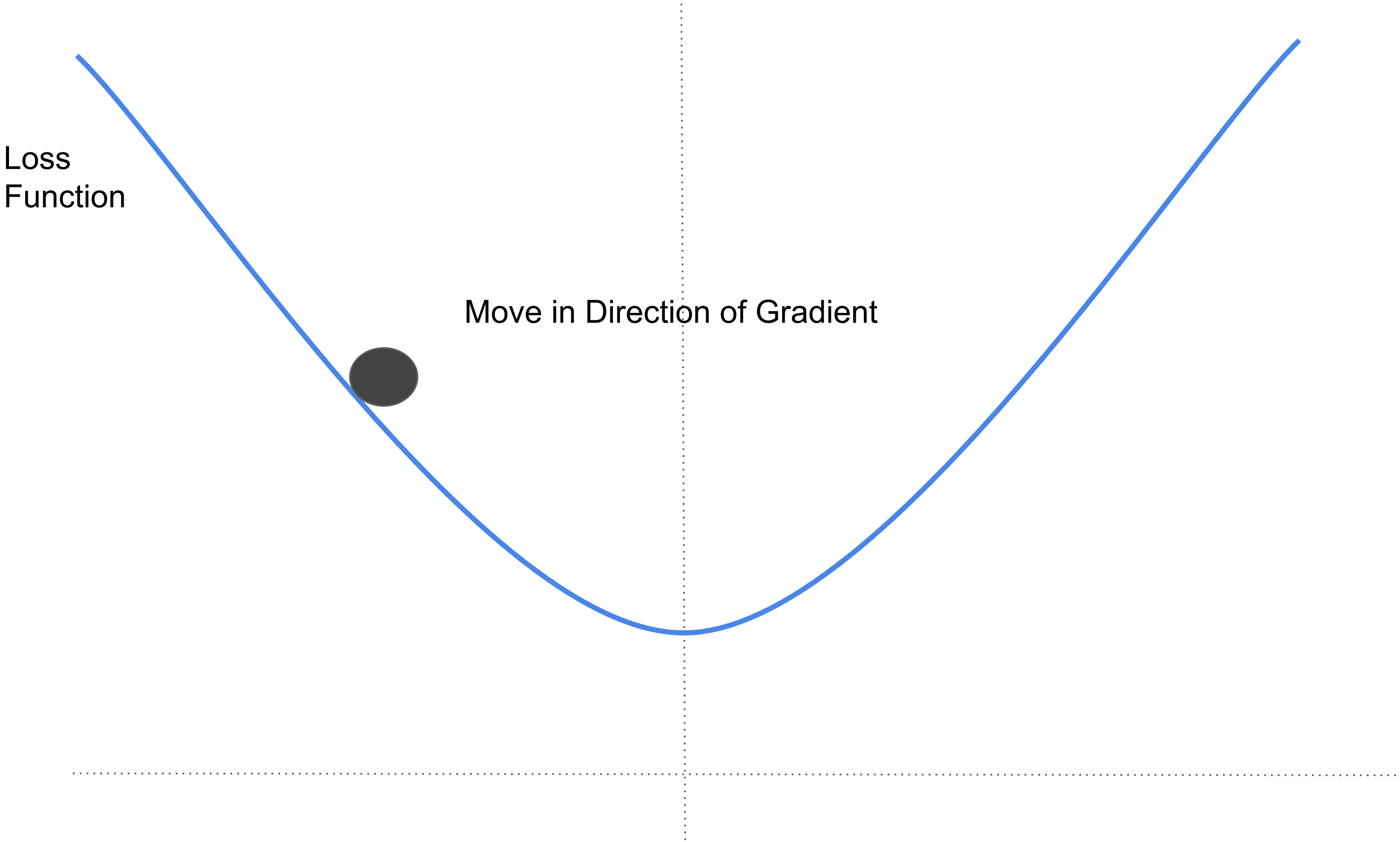

Loss
Function

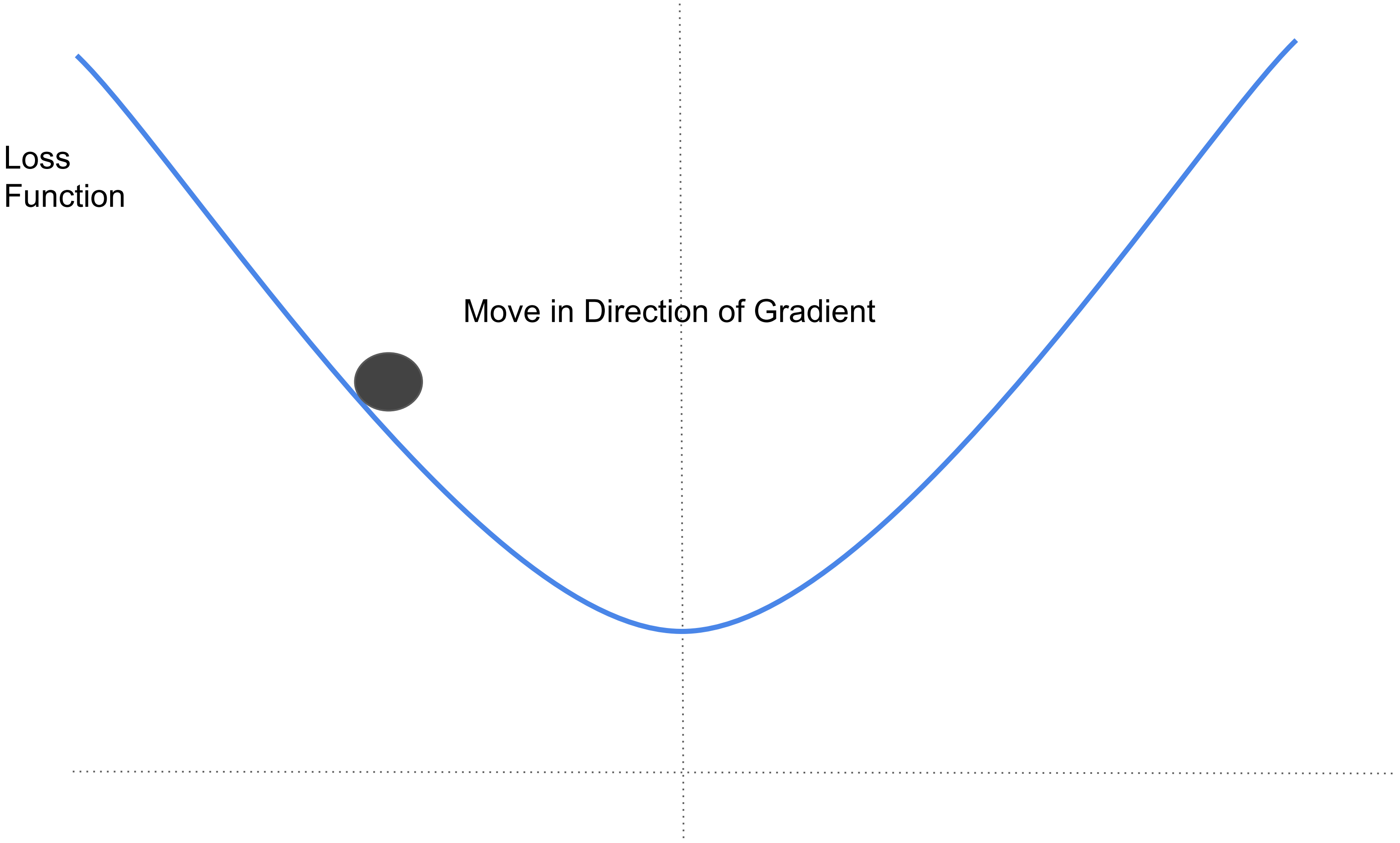

Loss
Function

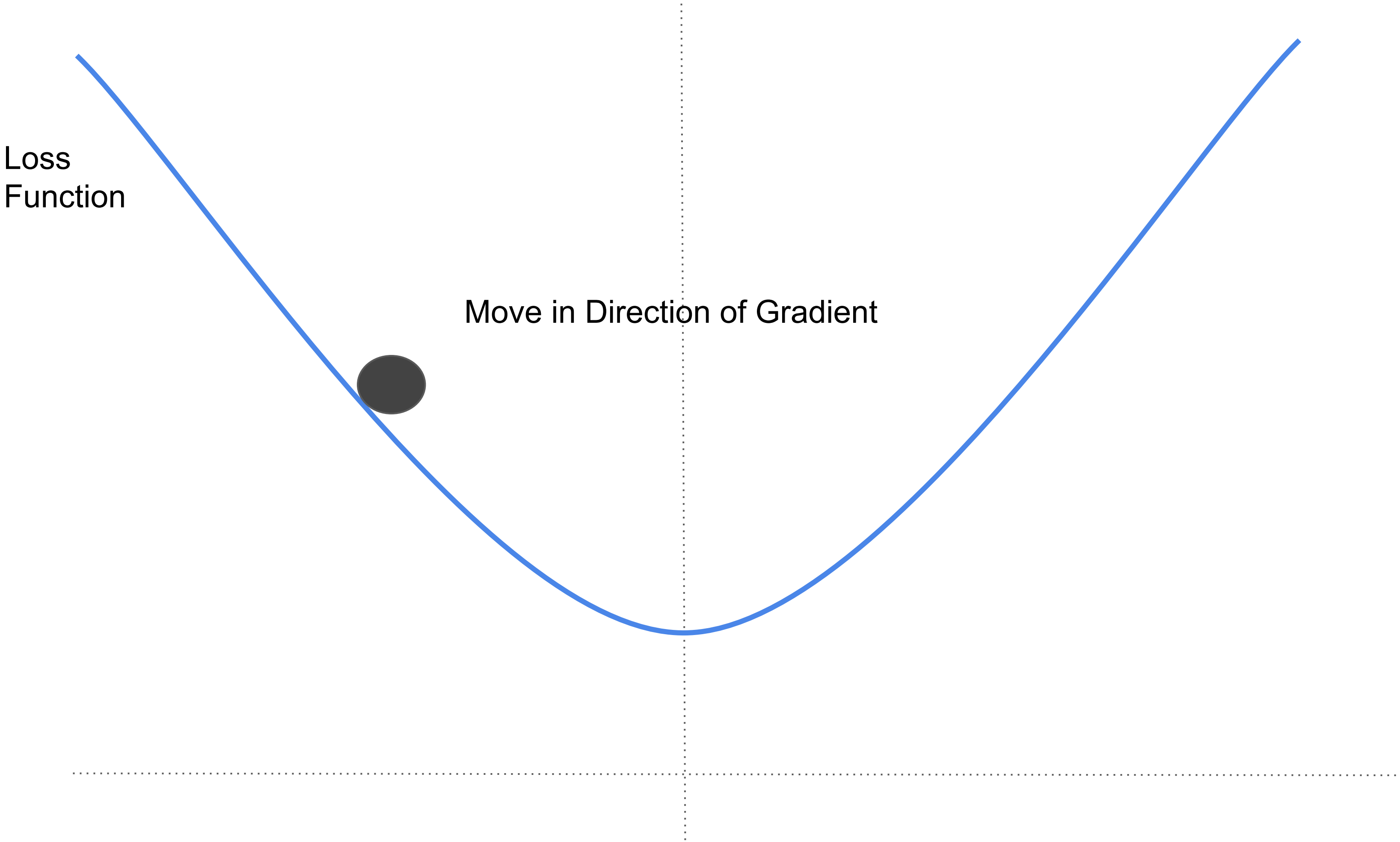

Loss
Function

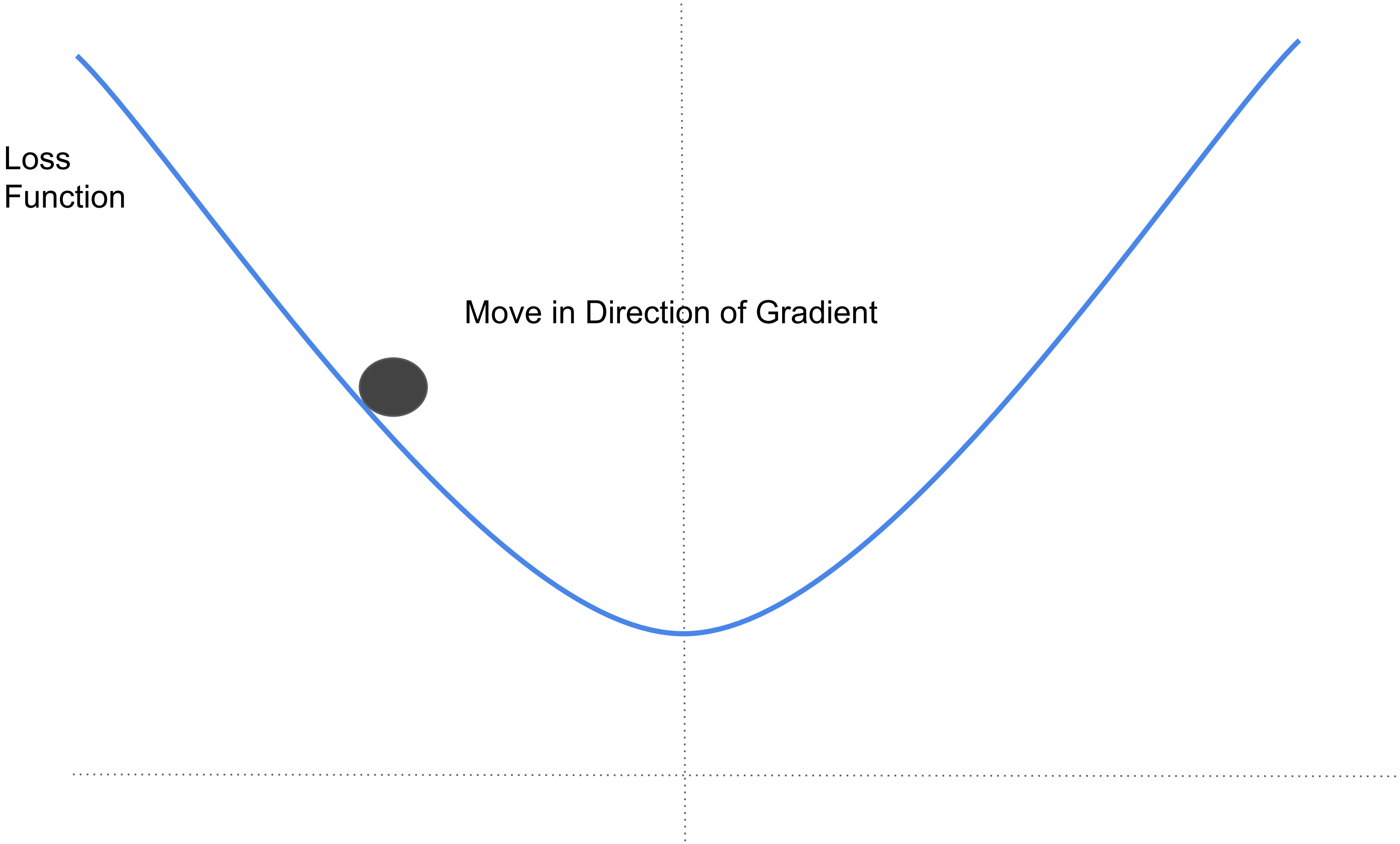

Loss
Function


Loss
Function
Move in Direction of Gradient


Loss
Function
Move in Direction of Gradient


Loss
Function
Move in Direction of Gradient


Loss
Function
Move in Direction of Gradient

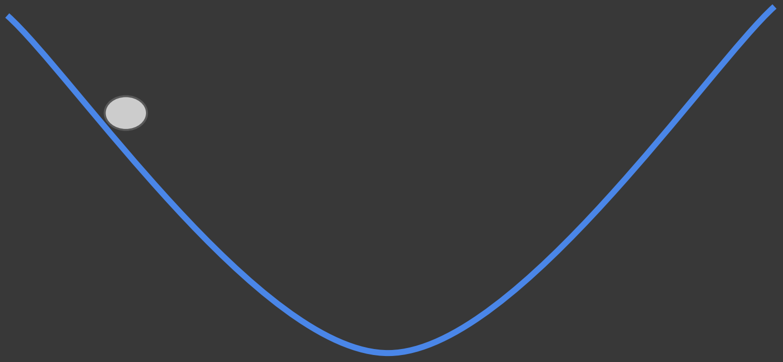

Loss
Function
Move in Direction of Gradient

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)
```

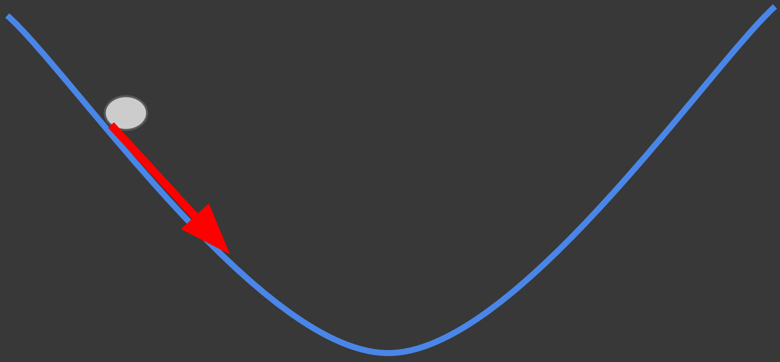
Calculate Partial Derivative of Loss

```
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)
```



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)
```

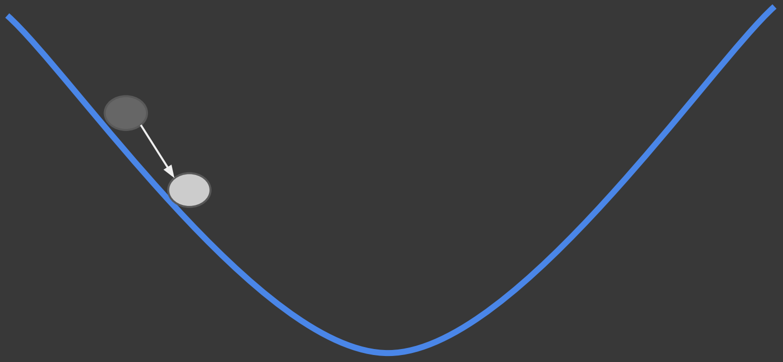


Calculate Partial Derivative of Loss

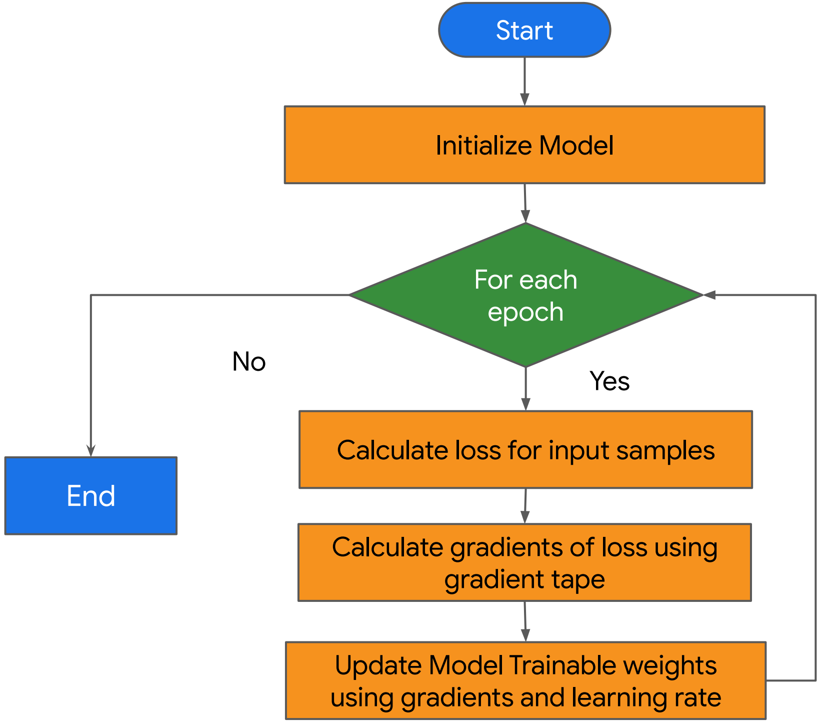
```
def train(model, inputs, outputs, learning_rate):
  with tf.GradientTape() as tape:
    current_loss = loss(outputs, model(inputs))
  dw, db = tape.gradient(current_loss, [model.w, model.b])

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

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

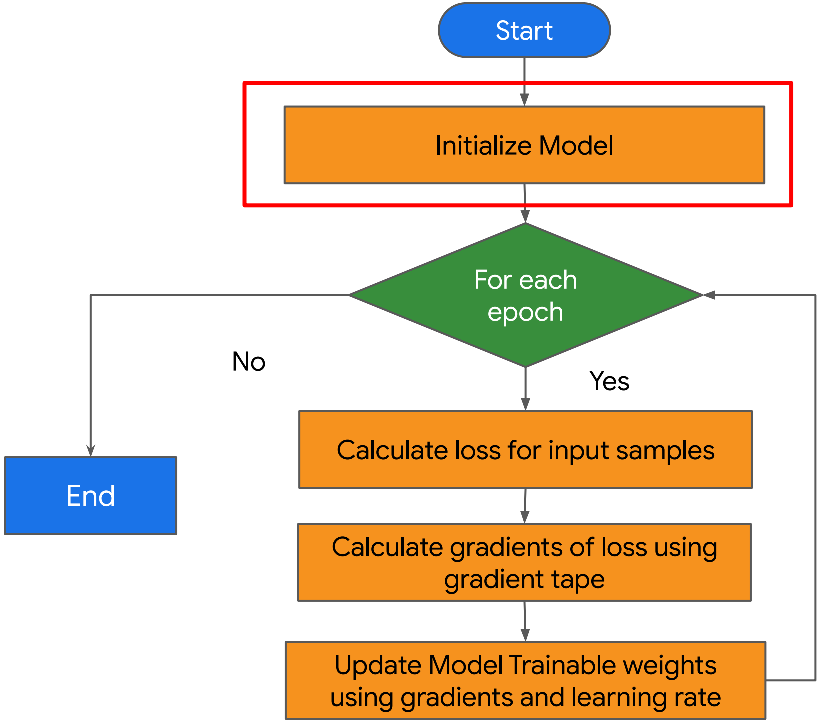
$$b = b - \alpha \times \frac{dL}{db}$$



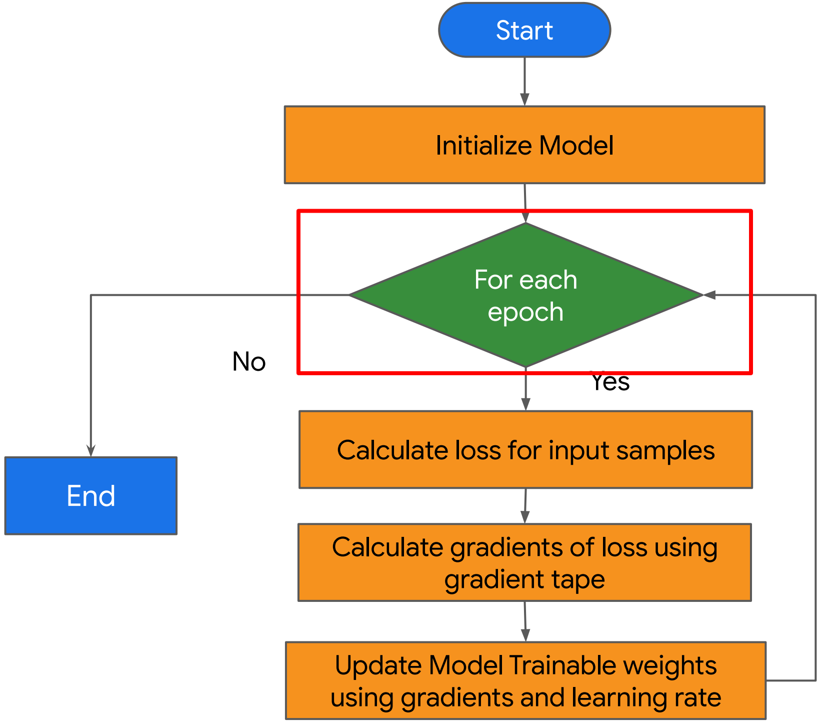
4. Training Loop



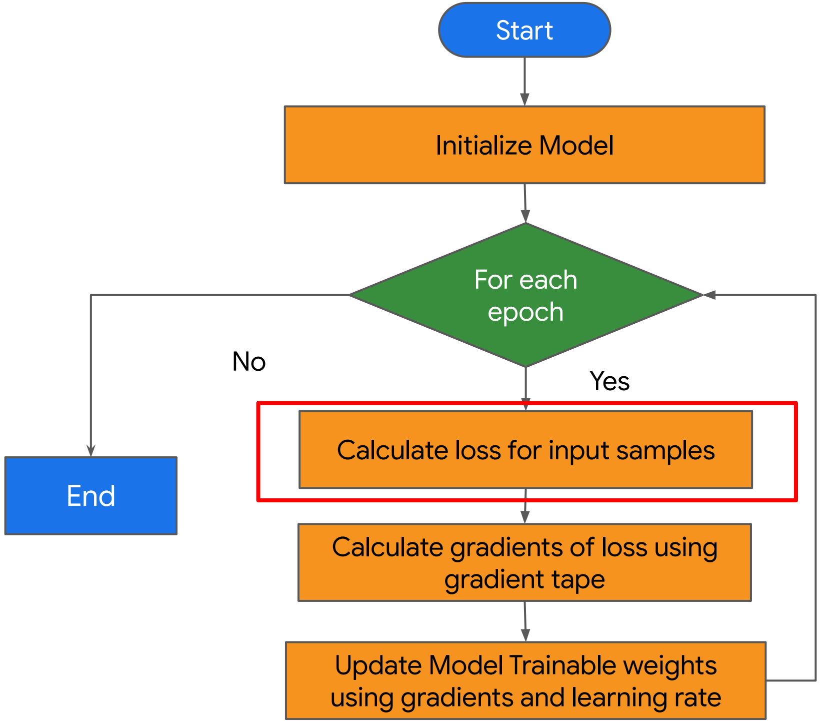
4. Training Loop



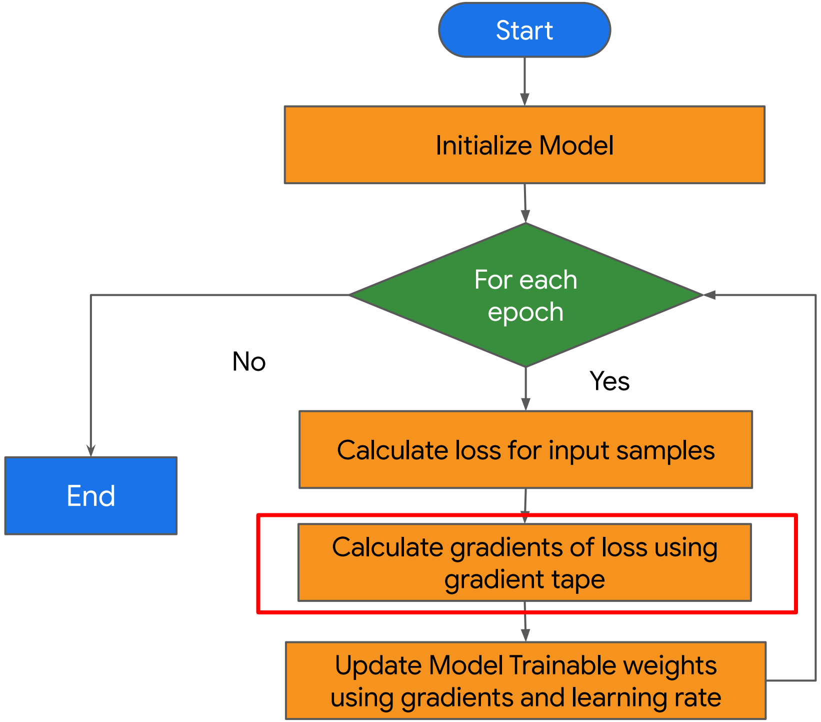
4. Training Loop



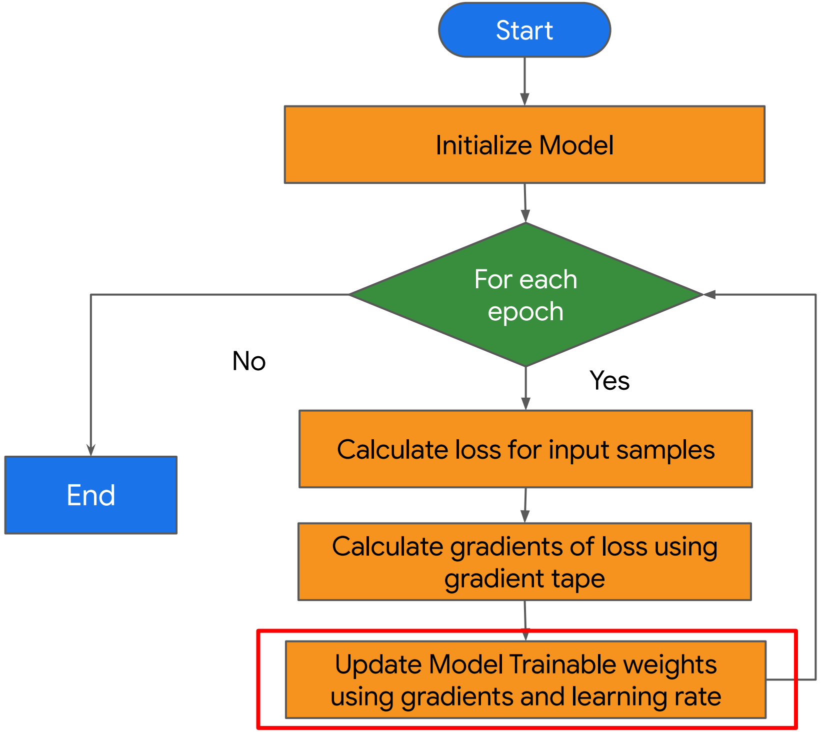
4. Training Loop



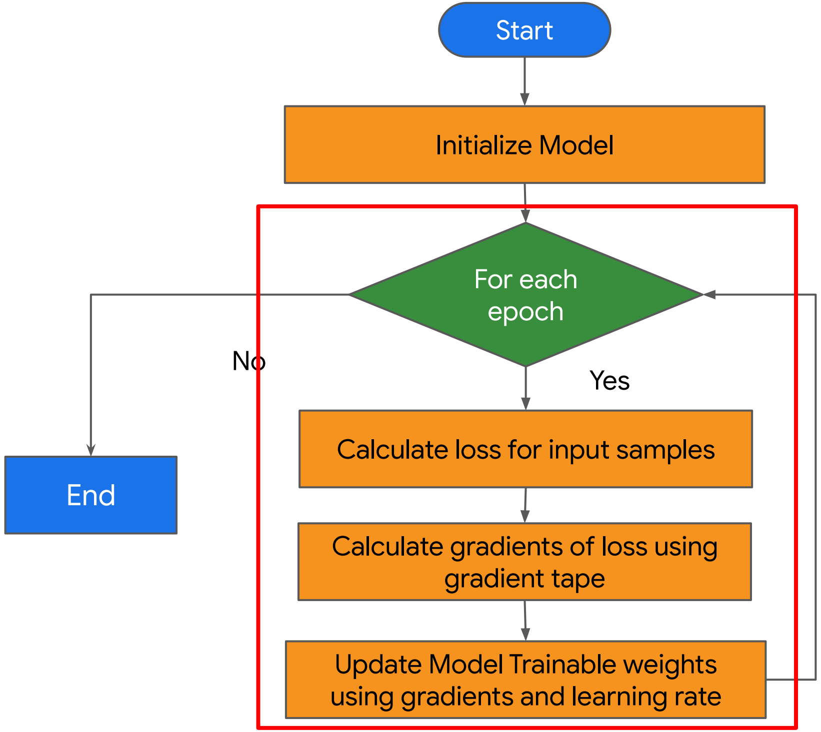
4. Training Loop



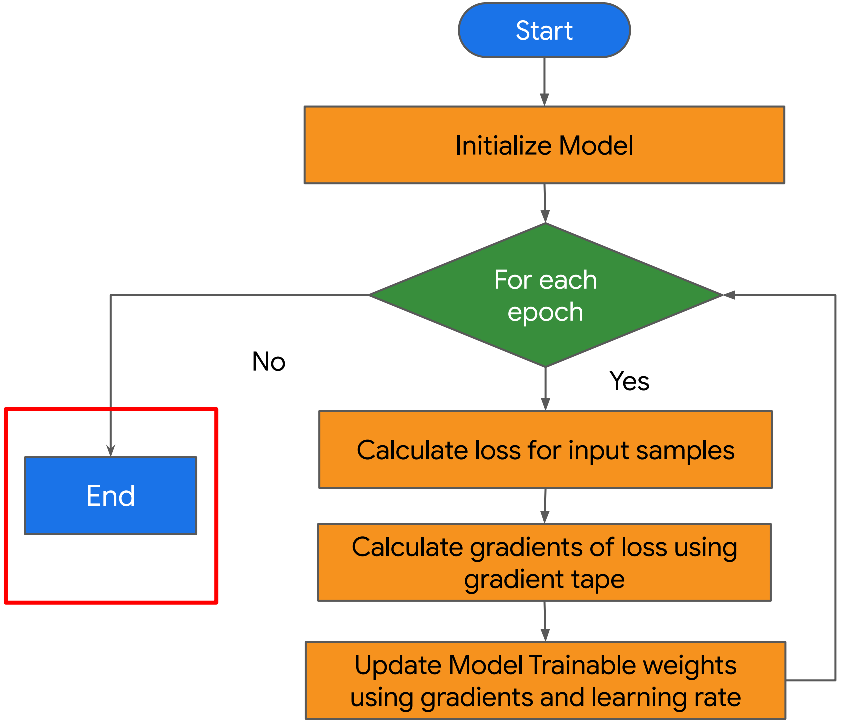
4. Training Loop



4. Training Loop



4. Training Loop



Calculate Partial Derivative of Loss

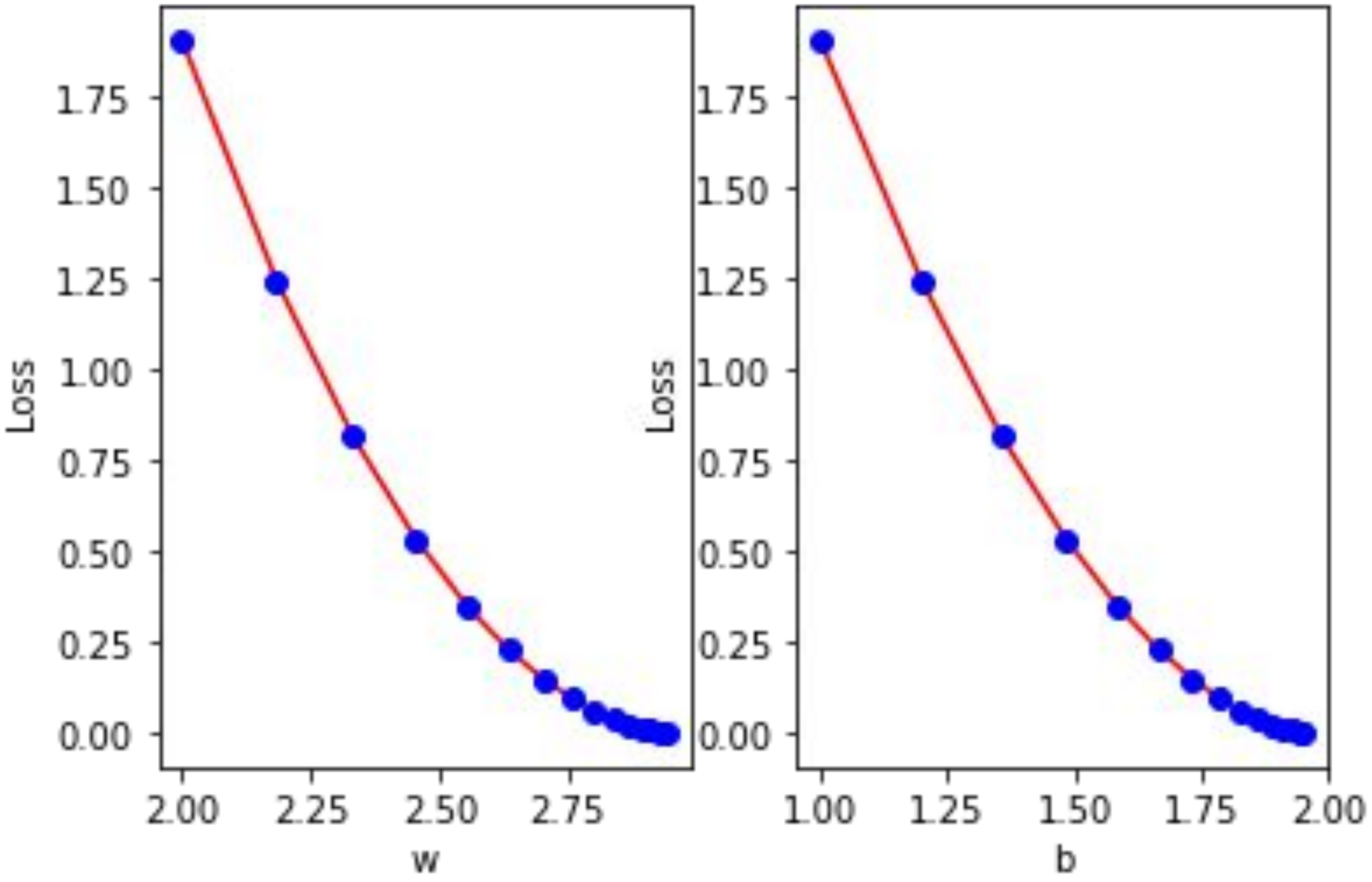
```
def train(model, inputs, outputs, learning_rate):
  with tf.GradientTape() as tape:
    current_loss = loss(outputs, model(inputs))
  da, db = tape.gradient(current_loss, [model.a, model.b])

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

Define Training Loop

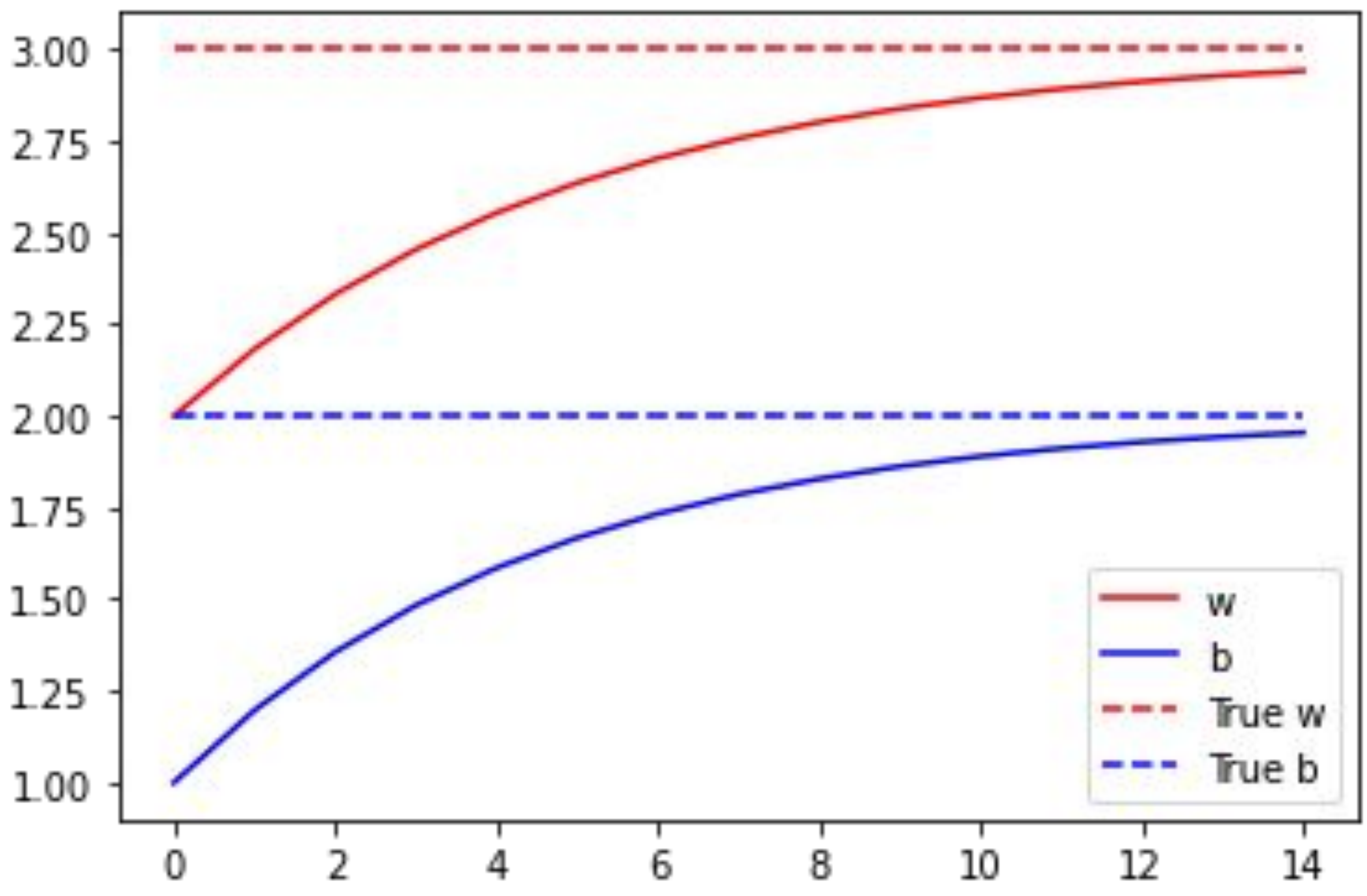
```
epochs = range(20)
for epoch in epochs:
    train(model, inputs, outputs, learning_rate=0.1)
```

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



Loss
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
2.00
2.25
2.50
2.75
w
Loss
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1.00
1.25
1.50
1.75
2.00
b



3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0
2
4
6
8
10
12
14
w
b
True w
True 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
```

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.

```
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)
```

```
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)
```

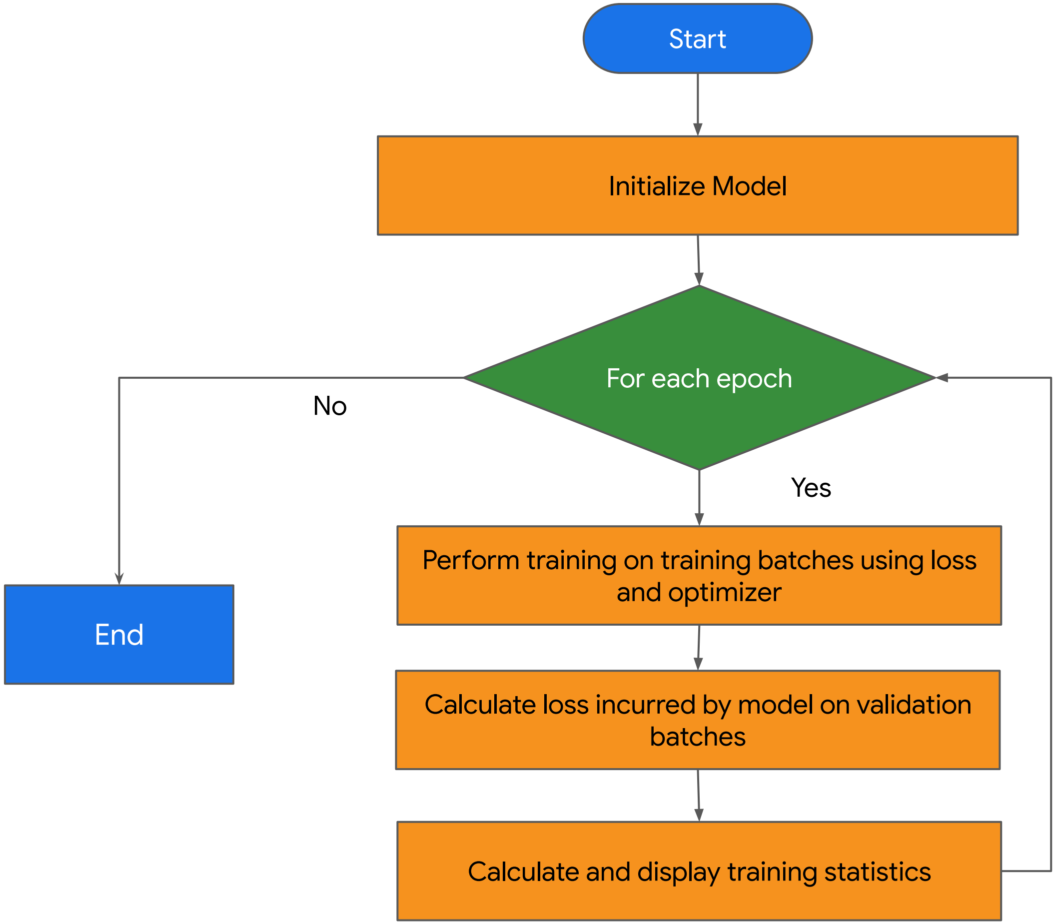
```python
train_data = tfds.load("fashion_mnist", split = "train")
test_data = tfds.load("fashion_mnist", split = "test")
def format_image(data):
    image = data["image"]
    image = tf.reshape(image, [-1])
    image = tf.cast(image, 'float32')
    image = image / 255.0
    return image, data["label"]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

3. Define Loss and Optimizer

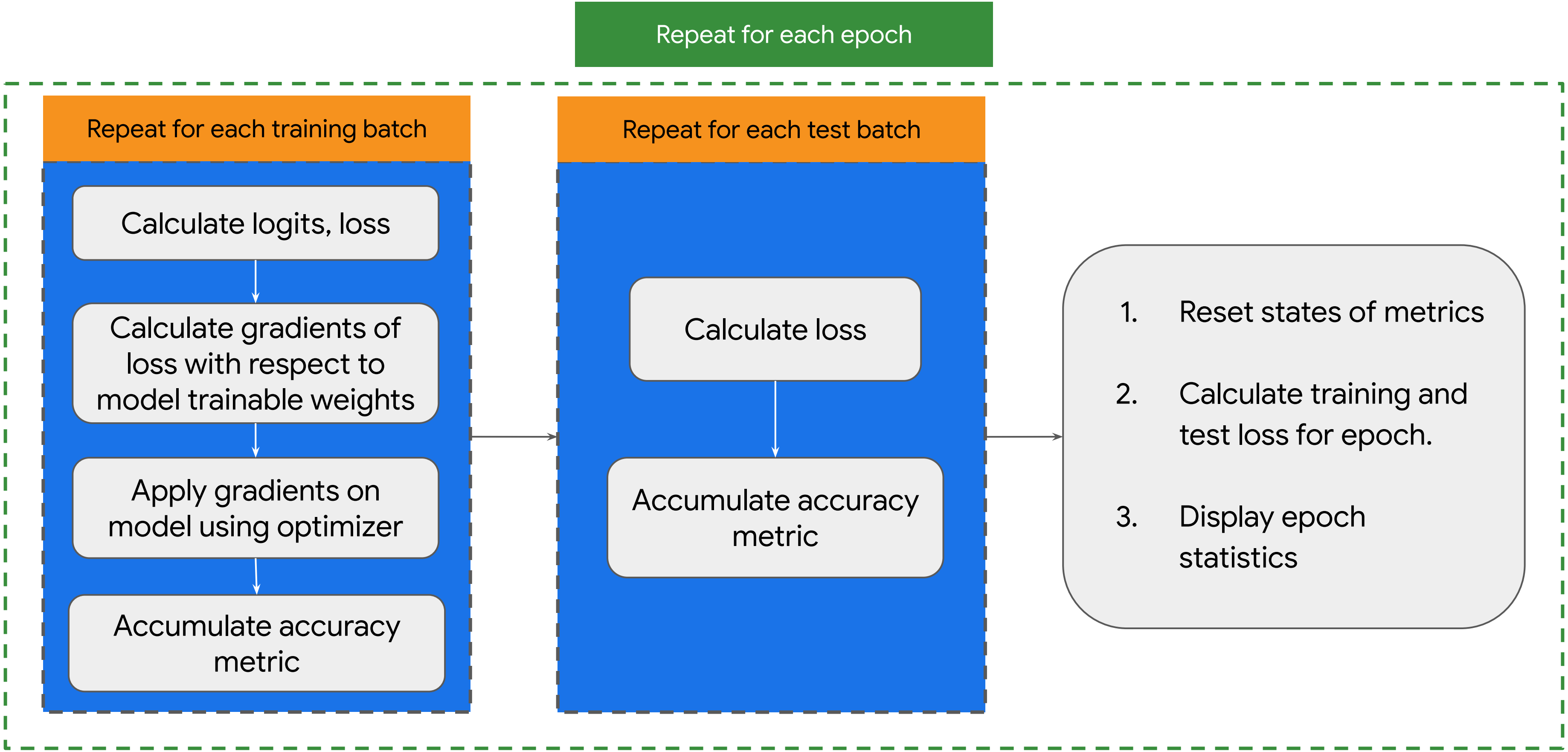
```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()

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

4. Define Custom Training Loop

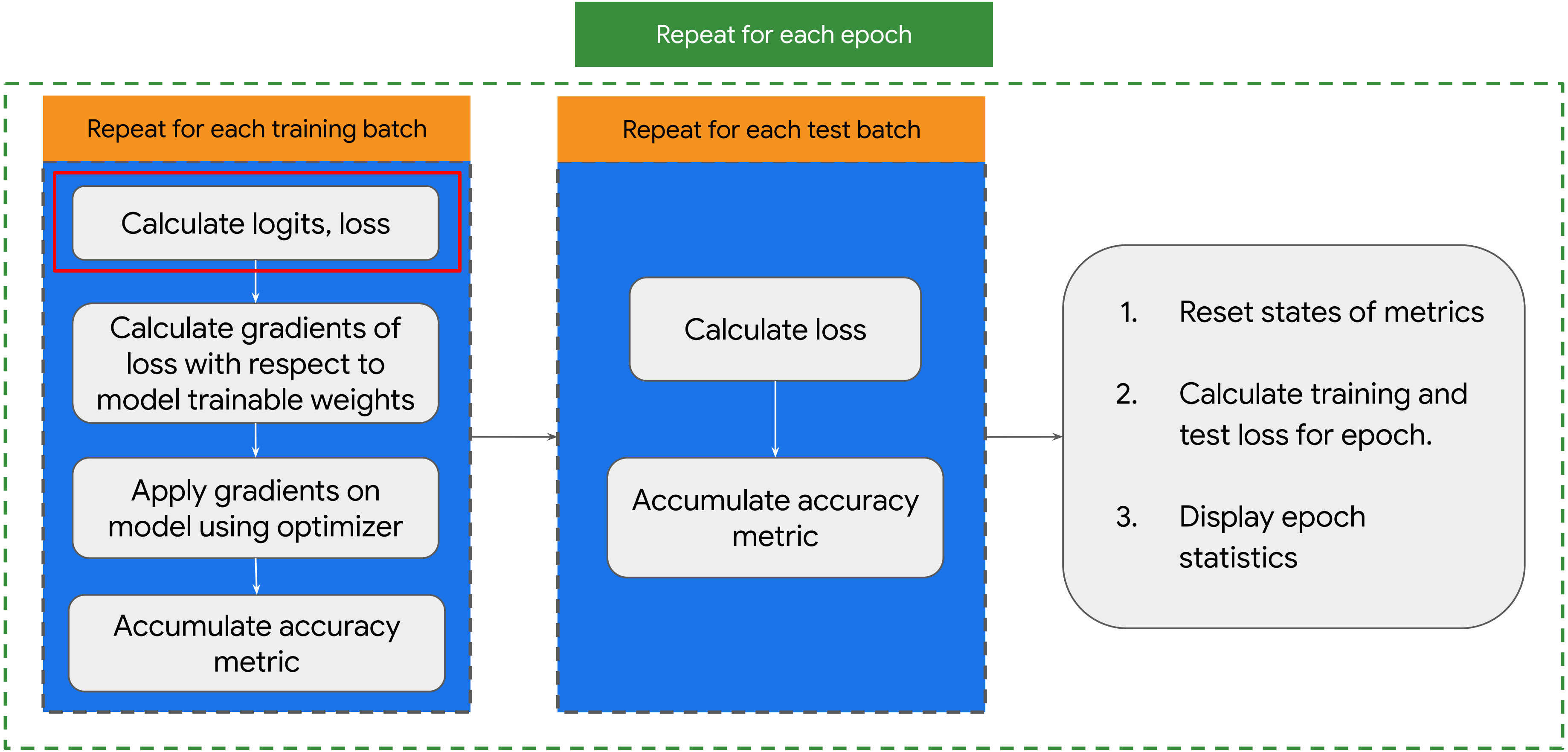
1. For each epoch, loop through the training batches and calculate gradients
2. These gradients are used according to the optimization algorithm chosen, to update the trainable weights of the model.
3. Loop through validation batches and calculate validation loss.


Start
Initialize Model
For each epoch
No
Yes
End
Perform training on training batches using loss and optimizer
Calculate loss incurred by model on validation batches
Calculate and display training statistics

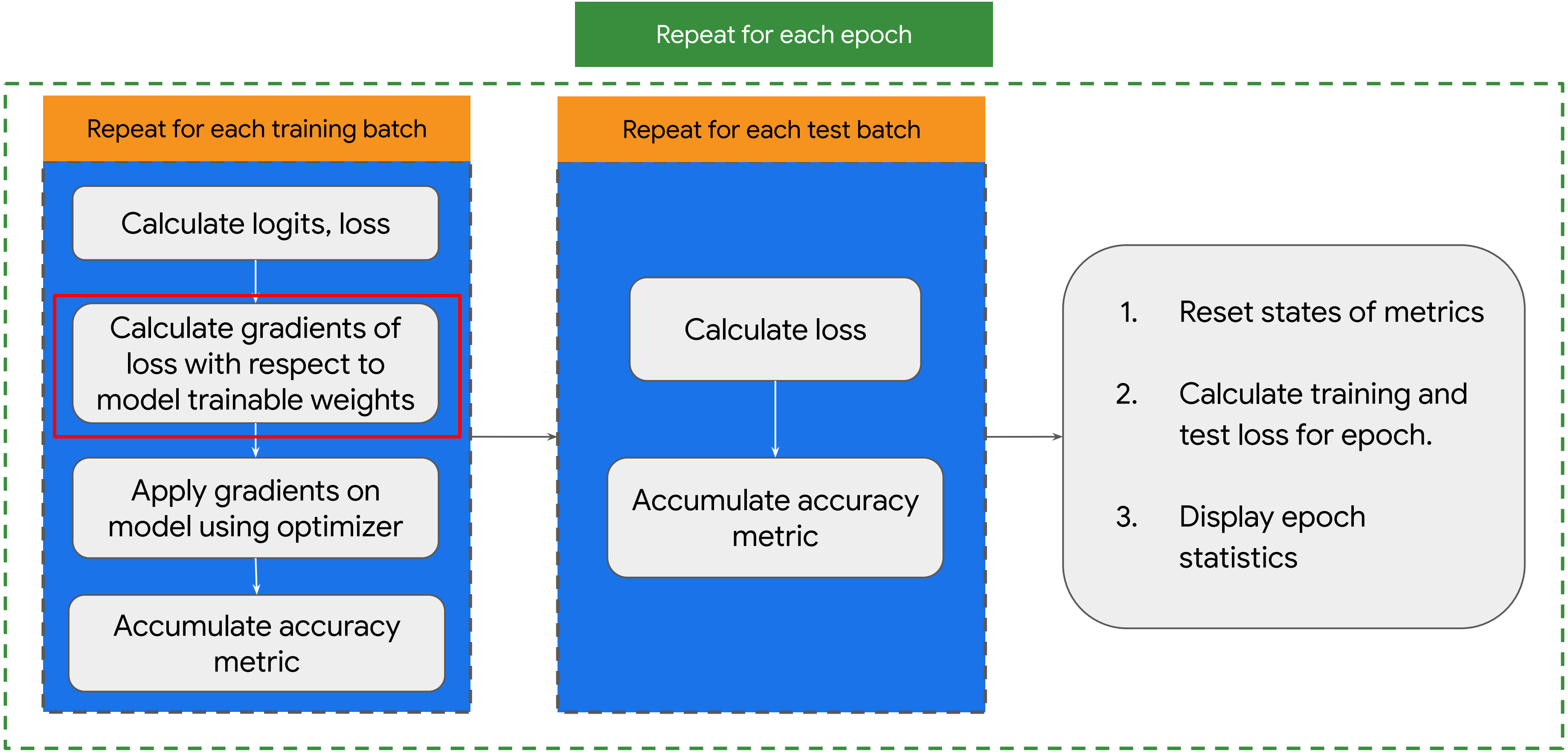
Training Loop - Architecture



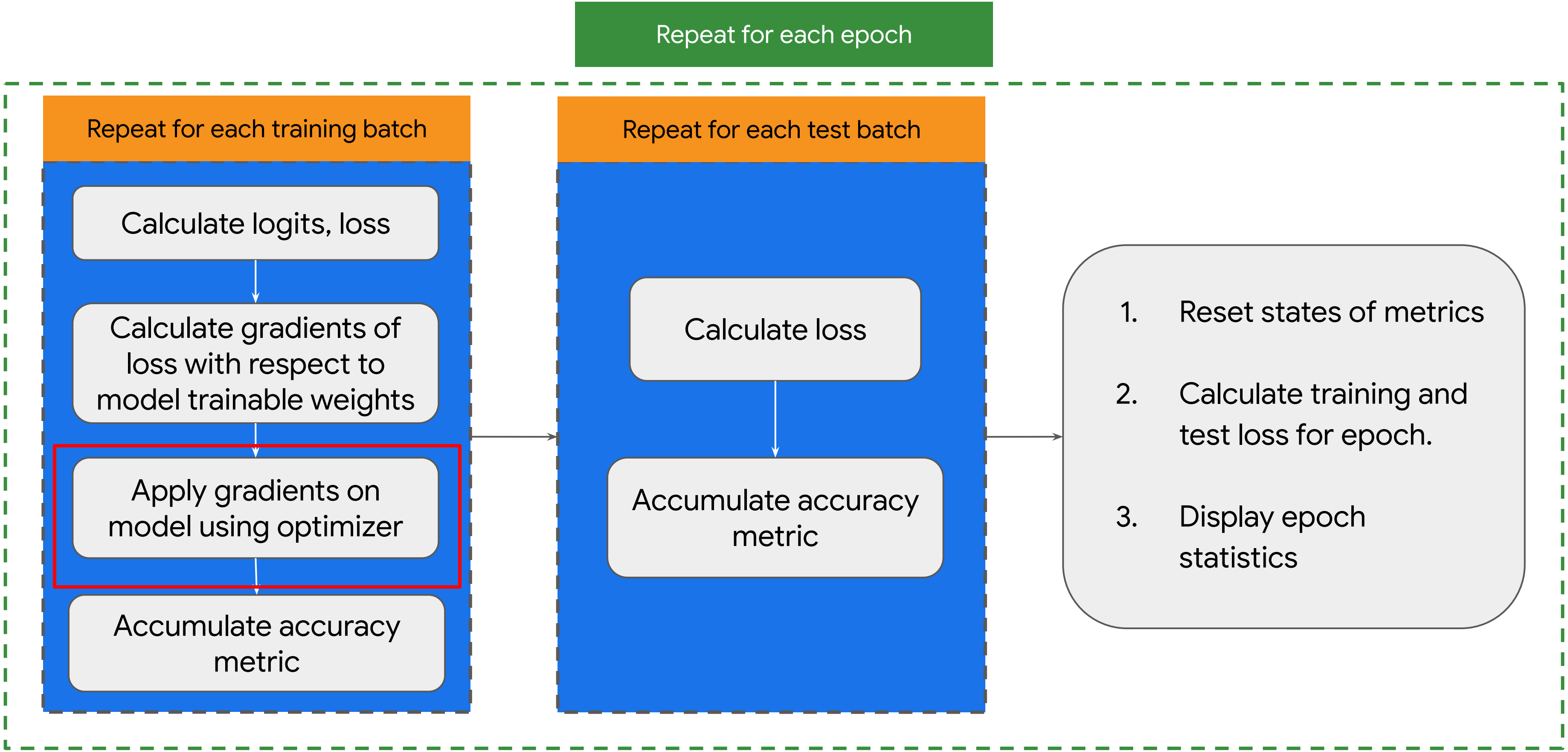
Training Loop - Architecture



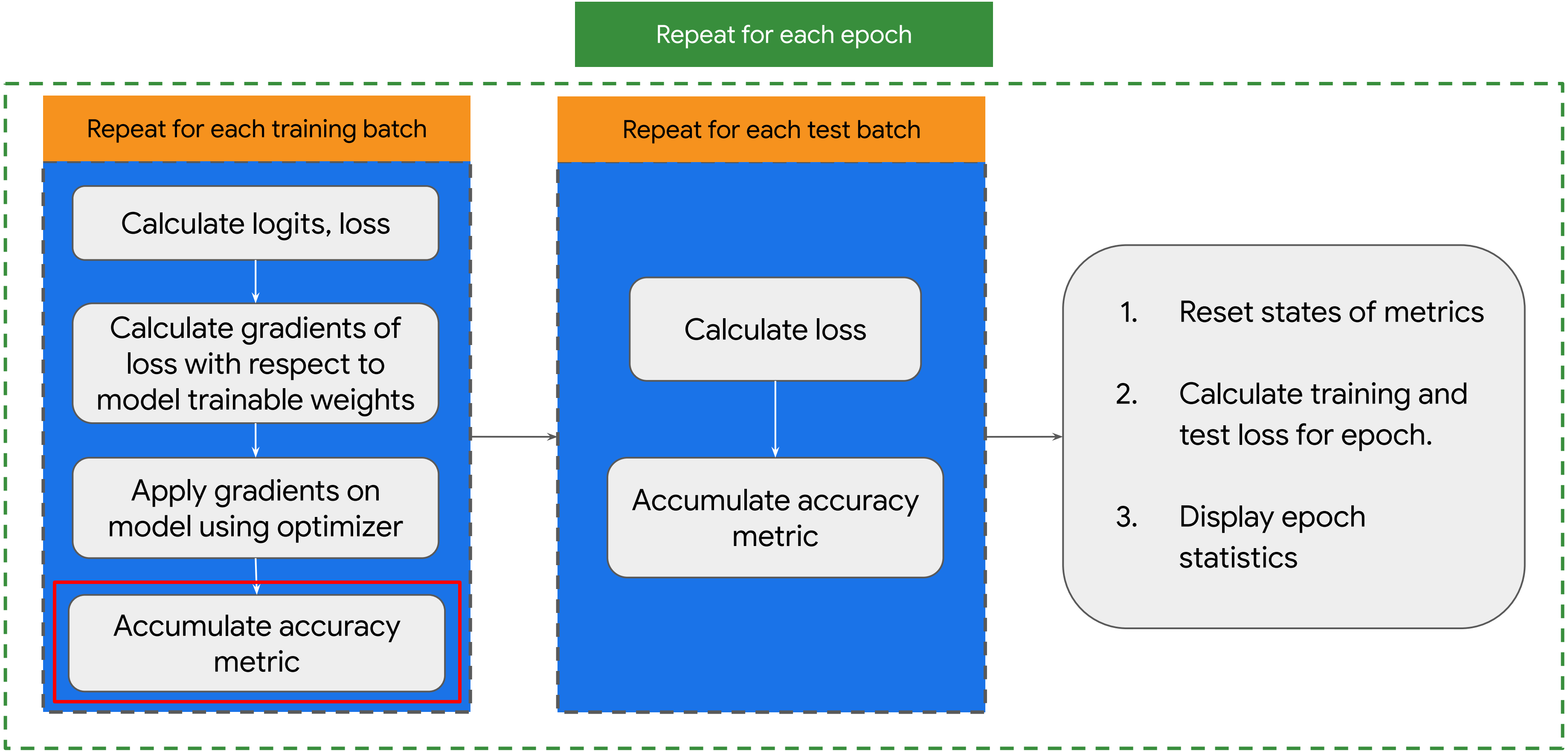
Training Loop - Architecture



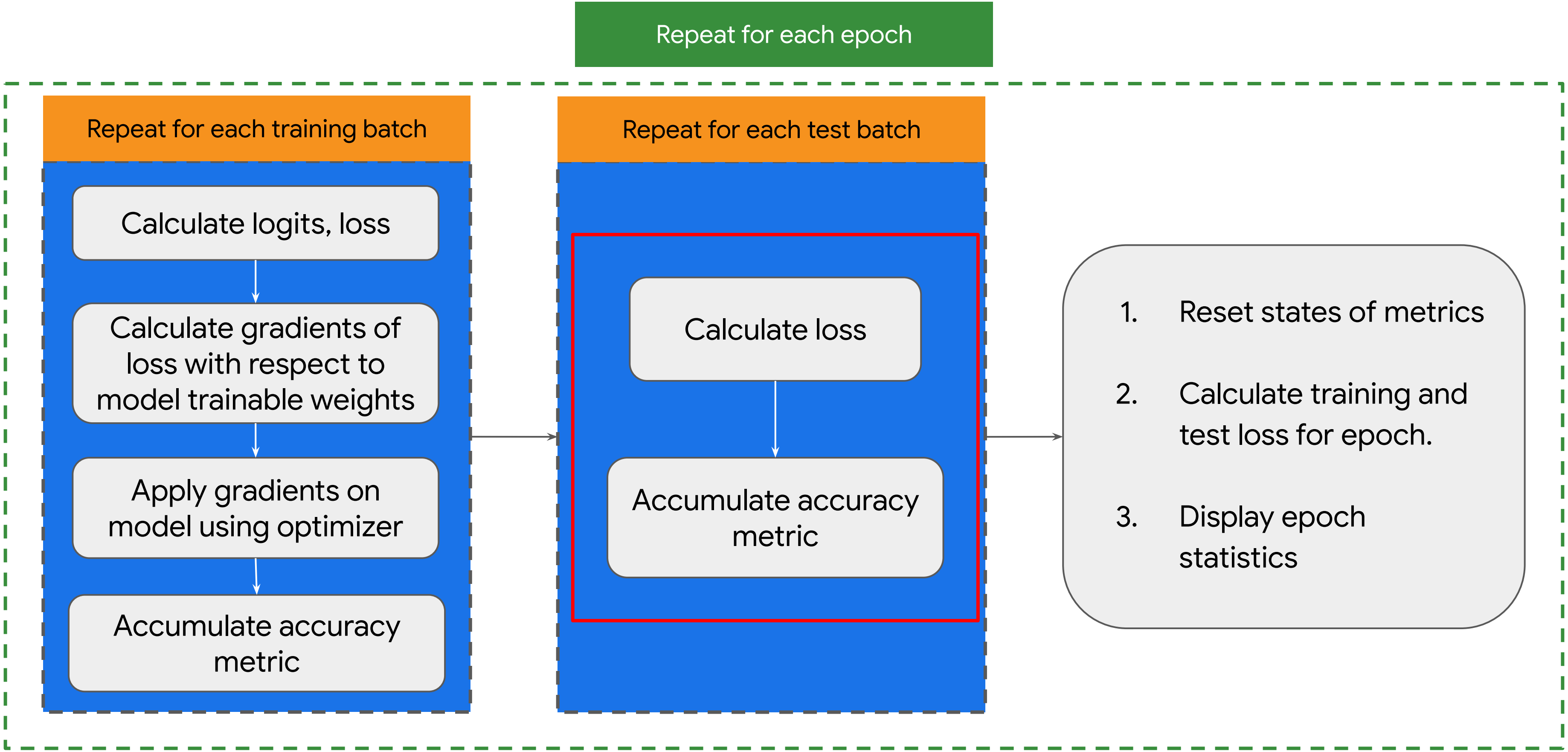
Training Loop - Architecture



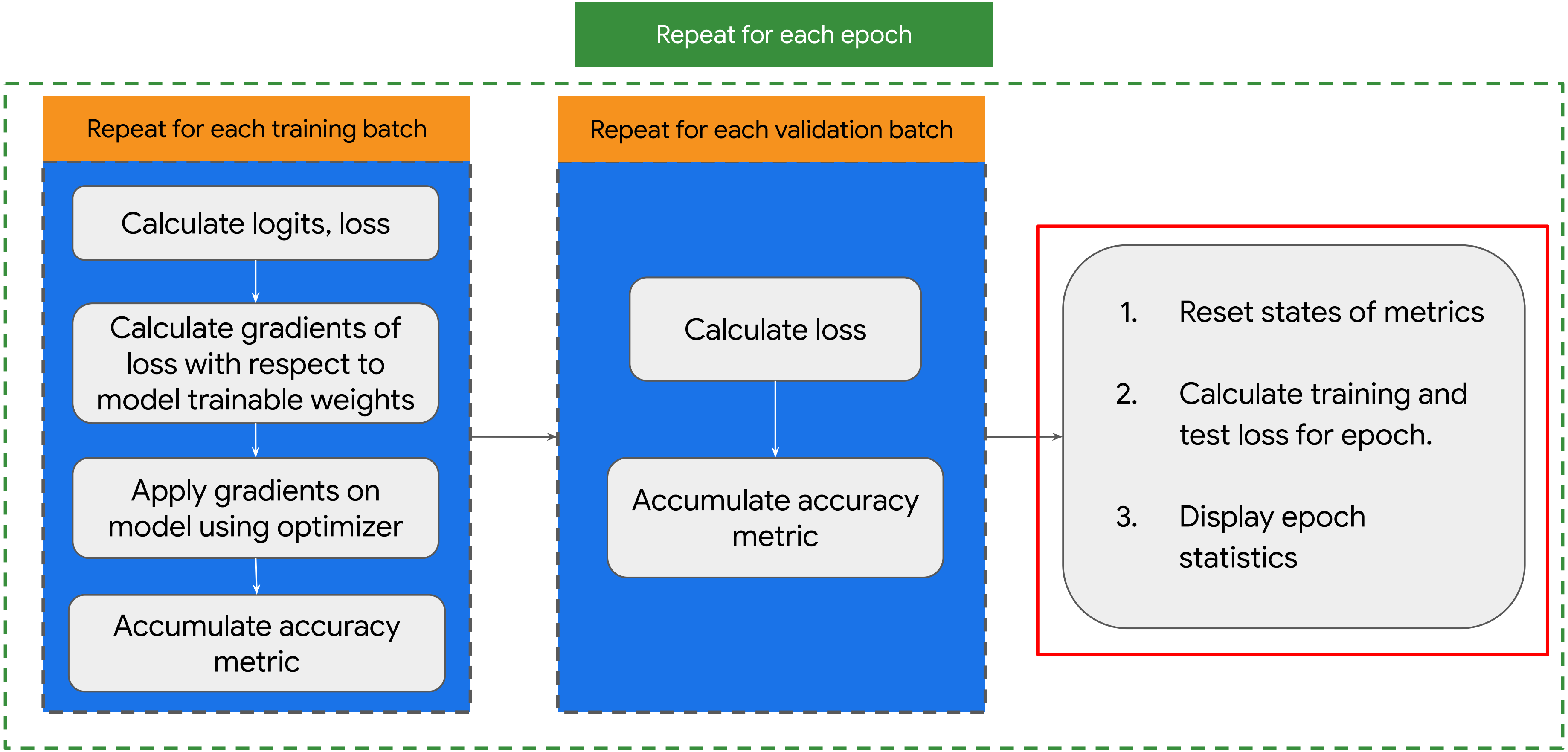
Training Loop - Architecture



Training Loop - Architecture



Training Loop - Architecture



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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

Calculate 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
```

Calculate 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
```

Calculate 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
```

Calculate 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
```

Calculate 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
```

Calculate 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
```

Calculate 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

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

```
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

```
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

```
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

```
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.