# CH2. Mathematical Building Blocks of Deep-learning (3)

**Il-Youp Kwak**
Chung-Ang University

# Mathematical Building Blocs of Deep-learning

A first look on neural network

Data representations for neural network

The gear of neural networks: tensor operations

**The engine of neural networks: gradient-based optimization**

**Looking back at our first example**

# The engine of neural networks: gradient-based optimization

## A dense layer:

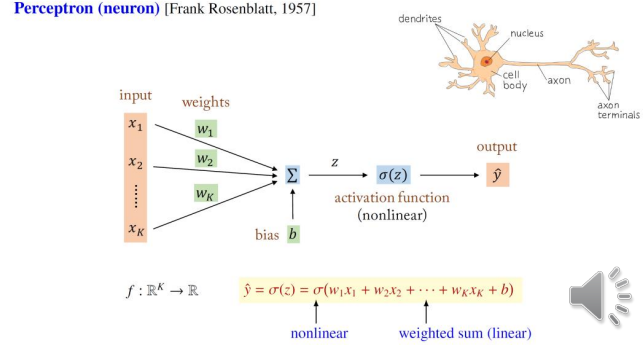
relu(W₁x + b₁)

```
output = relu(dot(W, input) + b)
```

W **and** b **are** **weights** **or** **trainable parameters**

**Gradual adjustment of them is called** **training**

**Perceptron (neuron)** [Frank Rosenblatt, 1957]



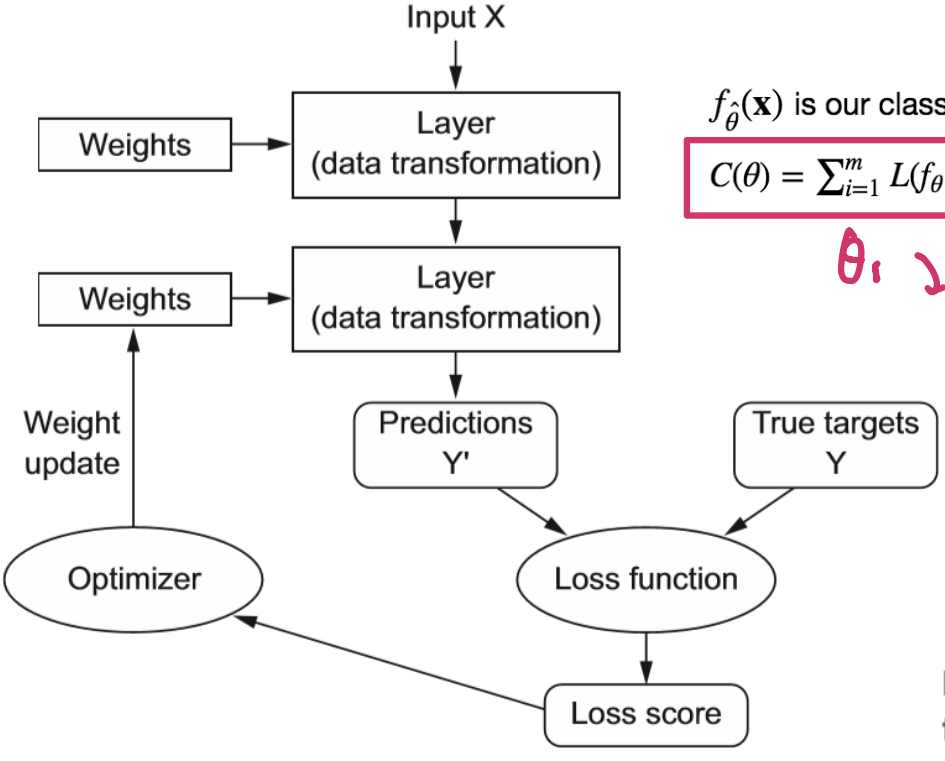
# Training loop

What we learned

- **Draw a batch of training samples *x* and corresponding targets *y*.**
- **Run the network on *x* (a step called the forward pass) to obtain predictions *y_pred*.**
- **Compute the loss of the network on the batch, a measure of the mismatch between *y_pred* and *y*.**

- **Update all weights of the network in a way that slightly reduces the loss on this batch.**

What we will learn this chapter




$f_{\hat{\theta}}(\mathbf{x})$ is our classification model with estimated parameters $\hat{\theta}$

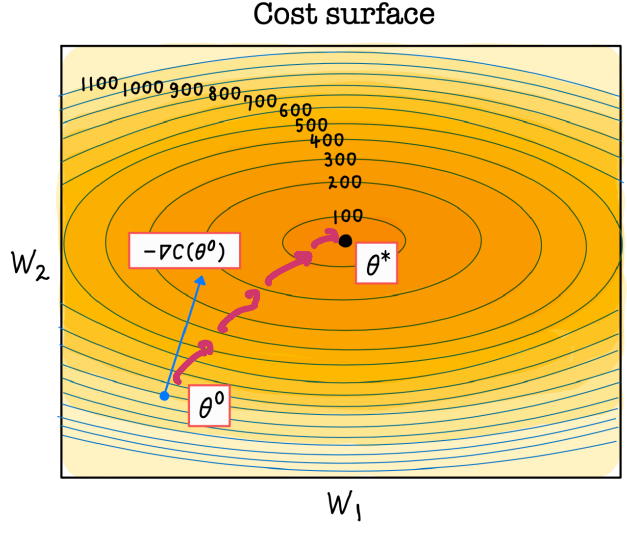
$$C(\theta) = \sum_{i=1}^{m} L(f_\theta(\mathbf{x}_i), y_i)$$

$\theta_1 \rightarrow \theta_2$

**Figure 1.9 The loss score is used as a feedback signal to adjust the weights.**

# Gradient Descent 경사하강법

- **Gradient descent is a first-order (requiring first-derivative/gradient) iterative optimization algorithm for finding the minimum of a function.**




network with two parameters $\theta = \{w_1, w_2\}$

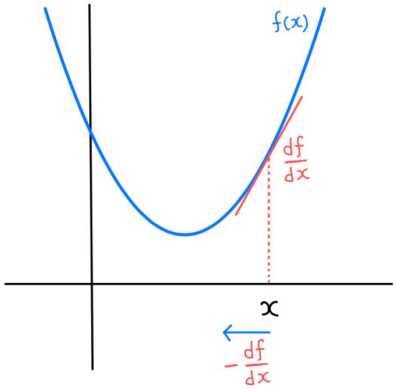
Ramdomly pick a starting point $\theta^0$.

Compute the negative gradient at each $\theta^t$.

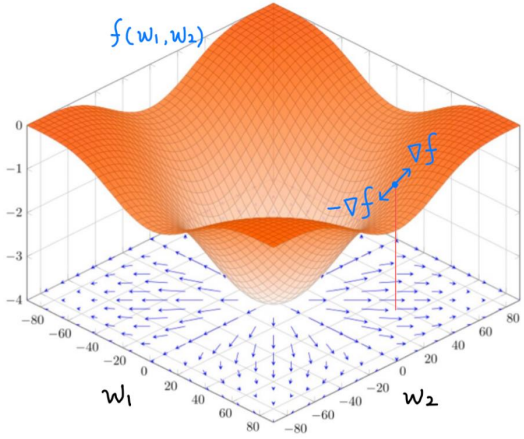
$$\Longrightarrow \quad -\nabla_\theta C(\theta^t)$$

$$\nabla_\theta C(\theta^t) = \begin{bmatrix} \frac{\partial C(\theta^t)}{\partial w_1} \\ \frac{\partial C(\theta^t)}{\partial w_2} \end{bmatrix}$$

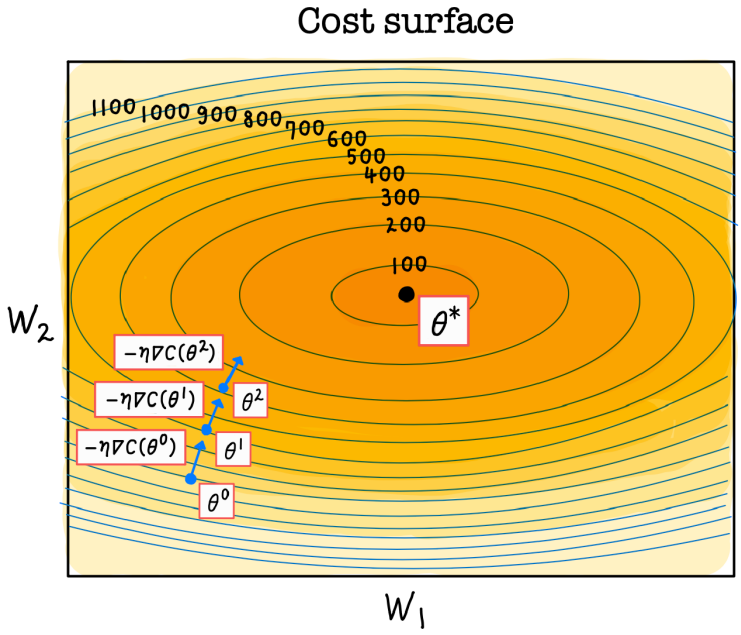
## Derivative




## Gradient




∗ Gradient vector $\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$ indicates the direction and rate of fastest increase.




$$\theta^{t+1} = \theta^t - \eta \nabla_\theta C(\theta^t)$$

Ramdomly pick a starting point $\theta^0$.

Compute the negative gradient at each $\theta^t$ followed by multiplying $\eta$.

$$\Longrightarrow \quad \boxed{-\eta \nabla_\theta C(\theta^t)}$$

learning rate

# Ex) Binary Classification Model

- We are Classifying Cat or Dog $f : \mathbf{x} \xrightarrow{f_\theta} \mathbb{R}_{[0,1]}$ (0 : cat, 1 : dog)
- Define a Cost function $C(\theta; X, \mathbf{y}) = \sum_{i=1}^{m} L(\theta; x_i, y_i)$
- Minimize Cost w.r.t $\theta$ given data $argmin_\theta C(\theta; X, \mathbf{y})$
- Iterate gradient updates $\hat{\theta}^* = \theta^t - \eta \nabla_\theta C(\theta^t)$ ($\theta^{t+1} = \theta^t - \eta \nabla_\theta C(\theta^t)$)
- $f_{\hat{\theta}}(\mathbf{x})$ is your classification model

# Ex) Binary Classification Model

- Minimize Loss w.r.t $\theta$ given data $argmin_\theta C(\theta; X, y)$

  This part may suffer (memory problem)

- We can consider using mini-batch Gradient Descent, Stochastic Gradient Descent

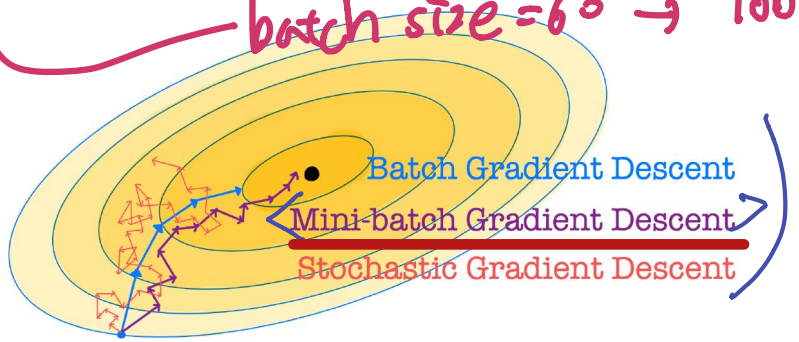
# Gradient descent variants

(1) Batch Gradient Descent: batch size = size of training dataset

(2) Stochastic Gradient Descent (SGD): batch size = 1

(3) Minibatch Gradient Descent

* Batch size: number of samples processed before the model parameters are updated.
* Number of epochs: number of complete passes through the entire training dataset.



# Variants of SGD

**SGD with momentum as well as Adagrad, RMSProp, and several others.**

**Known as *optimization methods* or *optimizers*.**

"SGD" ( Stochastic gradient 하강법 )

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

# Chaining derivatives: the Backpropagation algorithm

**chain rule:** *f(g(x)) = f'(g(x)) * g'(x).*

**D. Rumelhart, G. Hinton, and R. Williams (1986) proposed *backpropagation*, basically, it is gradient decent with chain rule**

$$f_\theta(x) = f_3(f_2(f_1(x))),$$

$$f_\theta'(x) = f_3'(f_2(f_1(x)))(f_2'(f_1(x)))f_1'(x)$$

$\theta^{t+1} = \theta^t - \eta \nabla_\theta L(\theta^t)$

$L(\theta) = \sum_{i=1}^{m} L(\theta)$

$= \sum_{i=1}^{m} H(f_\theta(x_i), y_i)$

# Backpropagation algorithm

y = w*x + b

loss_val = |y_true – y|

grad(loss_val, w)? grad(loss_val, b)?

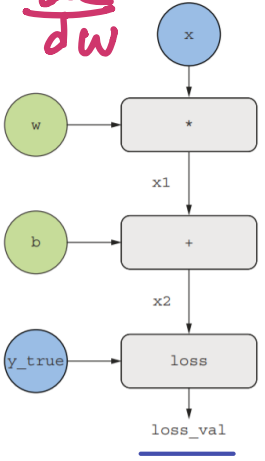



Figure 2.22 A basic example of a computation graph

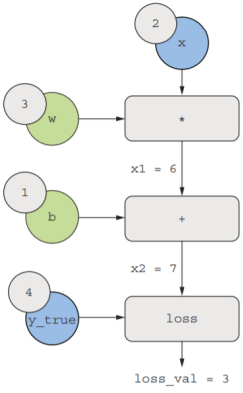



Figure 2.23 Running a forward pass

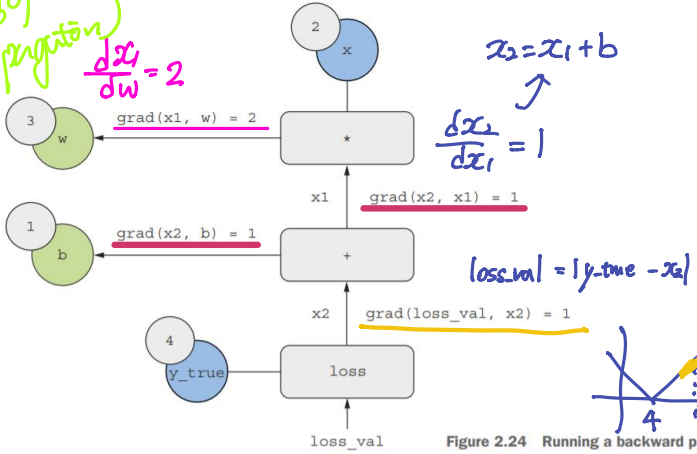



Figure 2.24 Running a backward pass

# Backpropagation algorithm

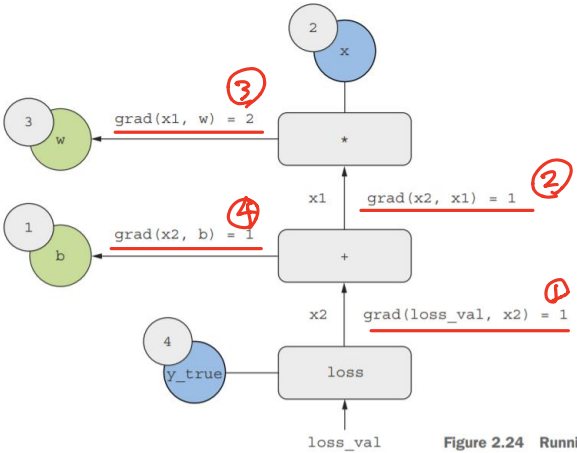



Figure 2.24 Running a backward pass

grad(loss_val, w) = 1 * 1 * 2 = 2

$\frac{dl}{dw} = \frac{dl}{dx_2}\frac{dx_2}{dx_1}\frac{dx_1}{dw} = 1 \times 1 \times 2 = 2$

grad(loss_val, b) = 1 * 1 = 1

$\frac{dl}{db} = \frac{dl}{dx_2}\frac{dx_2}{db} = 1 \times 1 = 1$

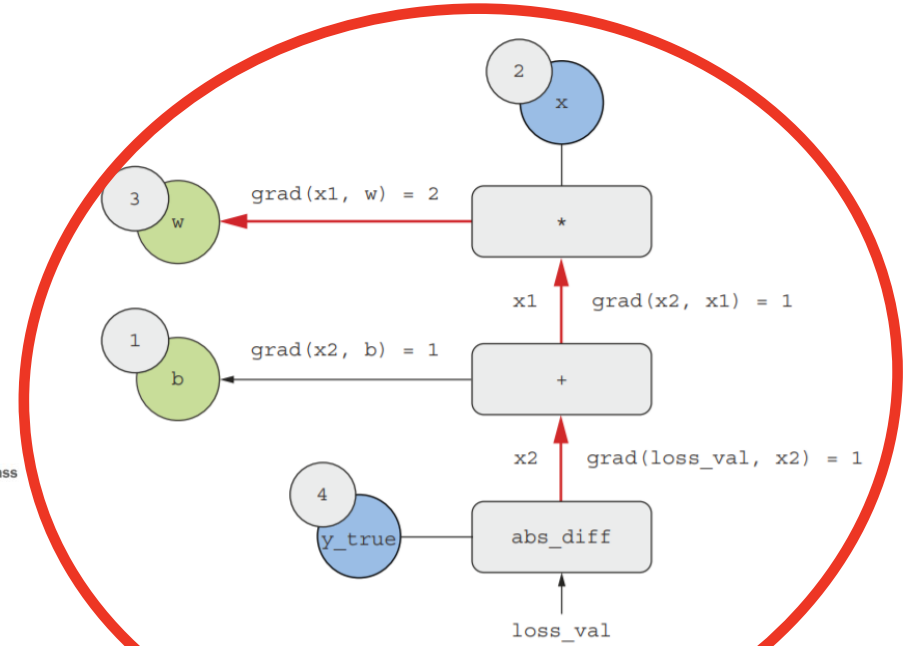



Figure 2.25 Path from loss_val to w in the backward graph

# Gradient tape in Tensorflow

Instantiate a scalar Variable with an initial value of 0.

Open a GradientTape scope.

Inside the scope, apply some tensor operations to our variable.

```
import tensorflow as tf
x = tf.Variable(0.)
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

Use the tape to retrieve the gradient of the output y with respect to our variable x.

The GradientTape works with tensor operations:

```
x = tf.Variable(tf.random.uniform((2, 2)))
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

Instantiate a Variable with shape (2, 2) and an initial value of all zeros.

grad_of_y_wrt_x is a tensor of shape (2, 2) (like x) describing the curvature of y = 2 * a + 3 around x = [[0, 0], [0, 0]].

# Gradient tape in Tensorflow

It also works with lists of variables:

```
W = tf.Variable(tf.random.uniform((2, 2)))
b = tf.Variable(tf.zeros((2,)))
x = tf.random.uniform((2, 2))
with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])
```

matmul is how you say "dot product" in TensorFlow.

grad_of_y_wrt_W_and_b is a list of two tensors with the same shapes as W and b, respectively.

$y = Wx + b$ $\frac{dy}{dW}$ or $\frac{dy}{db}$ ??

## Automatic differentiation with computation graphs

+ Code + Text

### The gradient tape in TensorFlow

[16]
```
import tensorflow as tf
x = tf.Variable(0.)
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

[18]
```
x = tf.Variable(tf.random.uniform((2, 2)))
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

[20]
```
W = tf.Variable(tf.random.uniform((2, 2)))
b = tf.Variable(tf.zeros((2,)))
x = tf.random.uniform((2, 2))
with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])
```

## Looking back at our first example

0s completed at 5:50 PM

# Looking back at our first example

## This was our input data:

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

## This was our network:

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

## This was our network-compilation step:

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

# This was the training loop:

```
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

chapter02_mathematical-building-blocks.i

File Edit View Insert Runtime Tools Help Cannot save changes

Share

Table of contents



+ Code + Text Copy to Drive

RAM Disk Editing

## Looking back at our first example

```
[ ] (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
    train_images = train_images.reshape((60000, 28 * 28))
    train_images = train_images.astype("float32") / 255
    test_images = test_images.reshape((10000, 28 * 28))
    test_images = test_images.astype("float32") / 255
```

```
[ ] model = keras.Sequential([
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
```

```
[ ] model.compile(optimizer="rmsprop",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
```

```
[ ] model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

469 x 128 = 60000

469 times updated per epoch

"SGD" 중에서 mini batch.

## Reimplementing our first example from scratch in TensorFlow

## A simple Dense class

0s completed at 5:54 PM

# Reimplementing our first example from scratch in TensorFlow

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

## Dense class

```
output = activation(dot(W, input) + b)
```

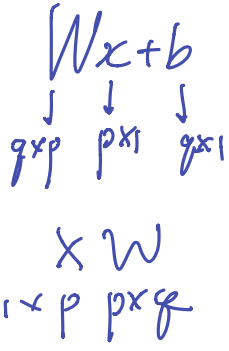
```
import tensorflow as tf

class NaiveDense:
    def __init__(self, input_size, output_size, activation):
        self.activation = activation

        w_shape = (input_size, output_size)
        w_initial_value = tf.random.uniform(w_shape, minval=0, maxval=1e-1)
        self.W = tf.Variable(w_initial_value)

        b_shape = (output_size,
        b_initial_value = tf.zeros(b_shape)
        self.b = tf.Variable(b_initial_value)

    def __call__(self, inputs)::
        return self.activation(tf.matmul(inputs, self.W) + self.b)

    @property
    def weights(self):
        return [self.W, self.b]
```

Create a matrix, W, of shape (input_size, output_size), initialized with random values.

Create a vector, b, of shape (output_size,), initialized with zeros.

Apply the forward pass.

Convenience method for retrieving the layer's weights



## Sequential class

```
class NaiveSequential:
    def __init__(self, layers):
        self.layers = layers

    def __call__(self, inputs):
        x = inputs
        for layer in self.layers:
            x = layer(x)
        return x

    @property
    def weights(self):
        weights = []
        for layer in self.layers:
            weights += layer.weights
        return weights
```

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

```
model = NaiveSequential([
    NaiveDense(input_size=28 * 28, output_size=512, activation=tf.nn.relu),
    NaiveDense(input_size=512, output_size=10, activation=tf.nn.softmax)
])
assert len(model.weights) == 4
```

## Batch Generator

```
import math

class BatchGenerator:
    def __init__(self, images, labels, batch_size=128):
        assert len(images) == len(labels)
        self.index = 0
        self.images = images
        self.labels = labels
        self.batch_size = batch_size
        self.num_batches = math.ceil(len(images) / batch_size)

    def next(self):
        images = self.images[self.index : self.index + self.batch_size]
        labels = self.labels[self.index : self.index + self.batch_size]
        self.index += self.batch_size
        return images, labels
```

X Y

No shuffling in this batch generator

---

Run the "forward pass" (compute the model's predictions under a GradientTape scope).

```
def one_training_step(model, images_batch, labels_batch):
    with tf.GradientTape() as tape:
        predictions = model(images_batch)
        per_sample_losses = tf.keras.losses.sparse_categorical_crossentropy(
            labels_batch, predictions)
        average_loss = tf.reduce_mean(per_sample_losses)
    gradients = tape.gradient(average_loss, model.weights)
    update_weights(gradients, model.weights)
    return average_loss
```

C(θ)

Update the weights using the gradients (we will define this function shortly).

Compute the gradient of the loss with regard to the weights. The output gradients is a list where each entry corresponds to a weight from the model.weights list.

## Naive mini-batch gradient update

```
learning_rate = 1e-3

def update_weights(gradients, weights):
    for g, w in zip(gradients, weights):
        w.assign_sub(g * learning_rate)
```

assign_sub is the equivalent of -= for TensorFlow variables.

## Using optimizer

```
from tensorflow.keras import optimizers

optimizer = optimizers.SGD(learning_rate=1e-3)

def update_weights(gradients, weights):
    optimizer.apply_gradients(zip(gradients, weights))
```

## Full training loop

```
def fit(model, images, labels, epochs, batch_size=128):
    for epoch_counter in range(epochs):
        print(f"Epoch {epoch_counter}")
    batch_generator = BatchGenerator(images, labels)
    for batch_counter in range(batch_generator.num_batches):
        images_batch, labels_batch = batch_generator.next()
        loss = one_training_step(model, images_batch, labels_batch)
        if batch_counter % 100 == 0:
            print(f"loss at batch {batch_counter}: {loss:.2f}")
```

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255

fit(model, train_images, train_labels, epochs=10, batch_size=128)
```

```
60000/128
```

```
468.75
```

## Reimplementing our first example from scratch in TensorFlow

### A simple Dense class

```
[ ]  import tensorflow as tf

     class NaiveDense:
         def __init__(self, input_size, output_size, activation):
             self.activation = activation

             w_shape = (input_size, output_size)
             w_initial_value = tf.random.uniform(w_shape, minval=0, maxval=1e-1)
             self.W = tf.Variable(w_initial_value)

             b_shape = (output_size,)
             b_initial_value = tf.zeros(b_shape)
             self.b = tf.Variable(b_initial_value)

         def __call__(self, inputs):
             return self.activation(tf.matmul(inputs, self.W) + self.b)

         @property
         def weights(self):
             return [self.W, self.b]
```

9s completed at 6:01 PM

# Thank you!