A GENTLE INTRODUCTION TO DEEP LEARNING WITH TENSORFLOW

Michelle Fullwood
@michelleful

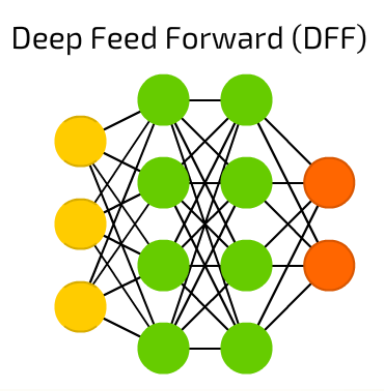
Slides: michelleful.github.io/PyCon2017

PREREQUISITES

- Knowledge of concepts of supervised ML
- Familiarity with linear and logistic regression

TARGET

(Deep) Feed-forward neural networks

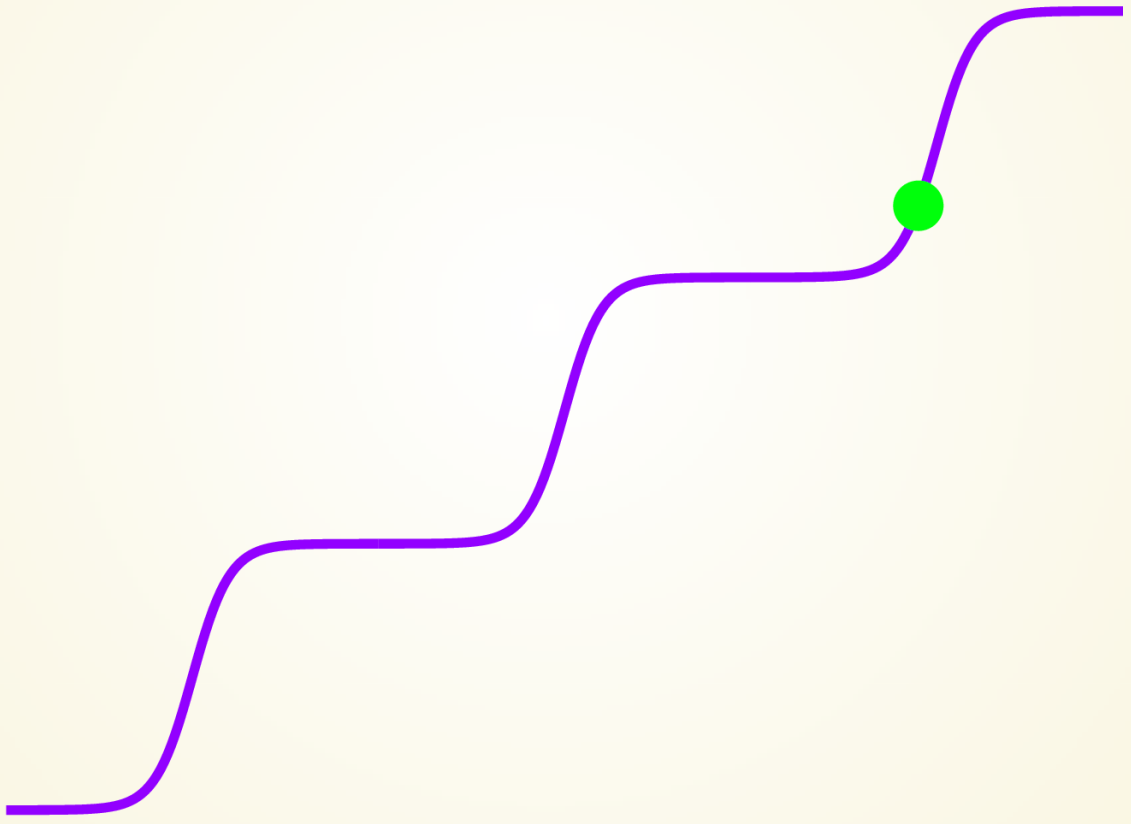


- How they're constructed
- Why they work
- How to train and optimize them

Image source: Fjodor van Veen (2016) Neural Network Zoo

DEEP LEARNING LEARNING CURVE

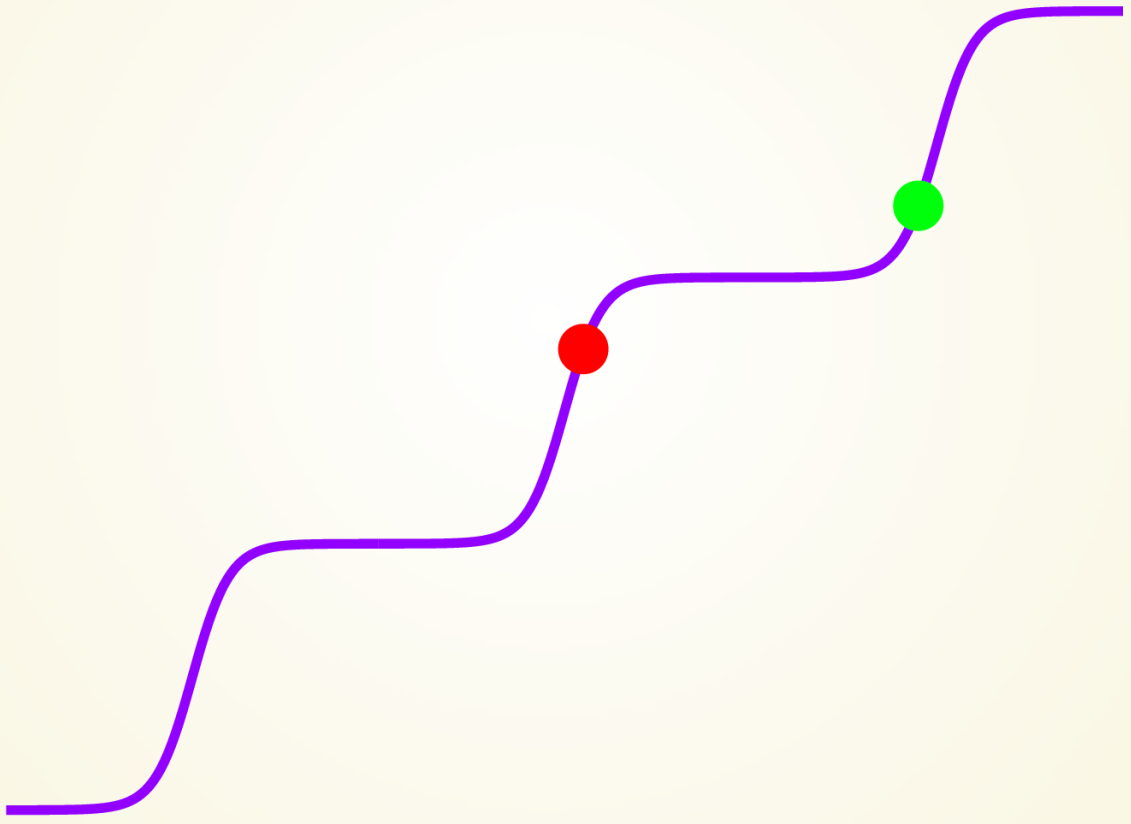
DEEP LEARNING LEARNING CURVE

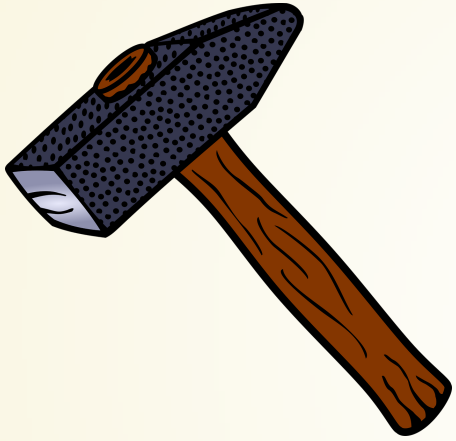


DEEP LEARNING LEARNING CURVE

DEEP LEARNING LEARNING CURVE

DEEP LEARNING LEARNING CURVE





Traditional machine learning

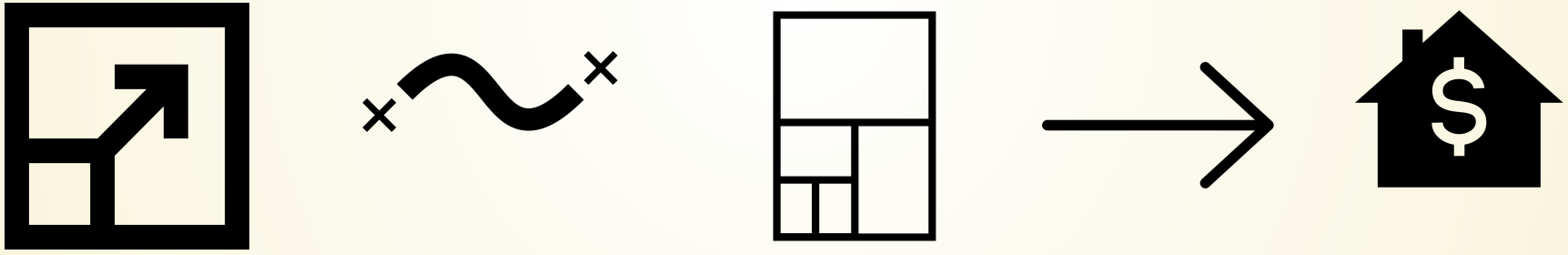
Deep learning

TENSORFLOW

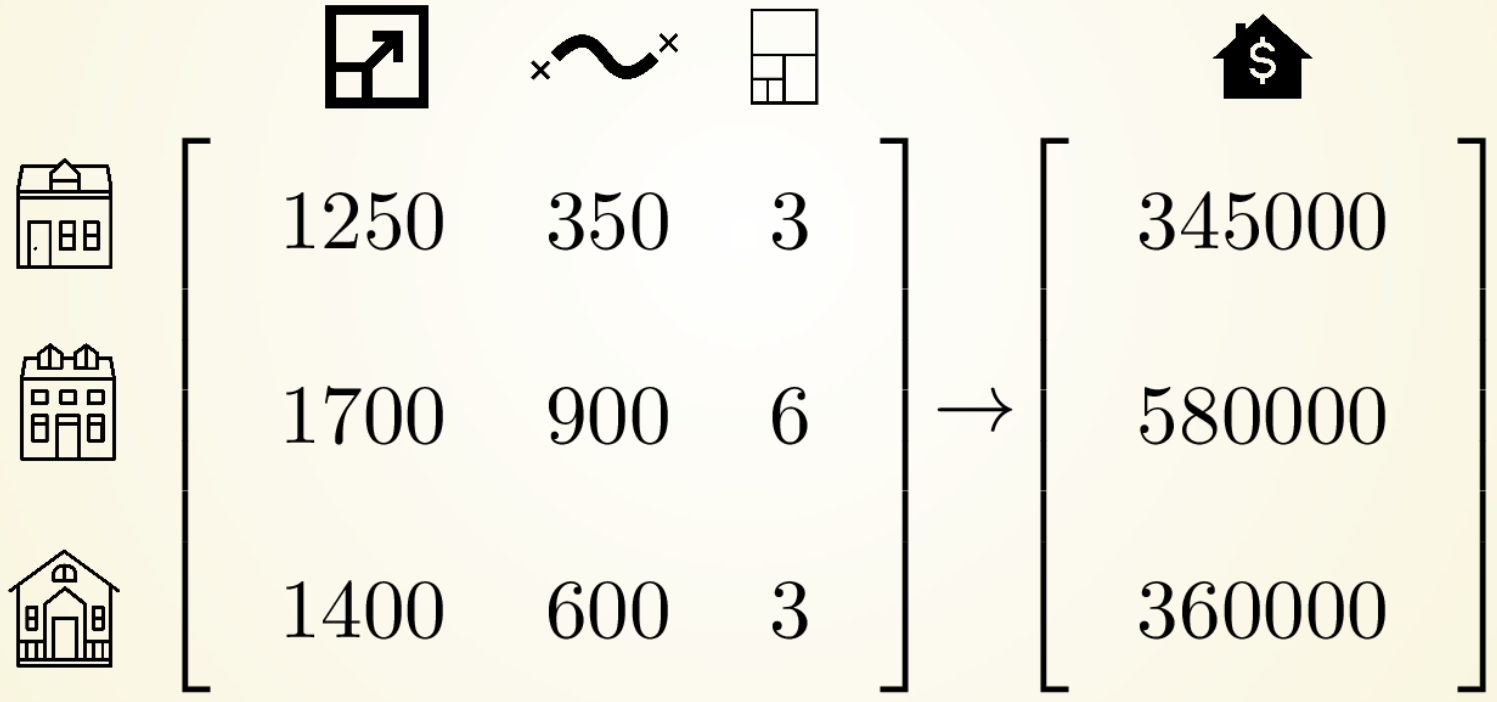
- Popular deep learning toolkit
- From Google Brain, Apache-licensed
- Python API, makes calls to C++ back-end
- Works on CPUs and GPUs

LINEAR REGRESSION FROM SCRATCH

LINEAR REGRESSION



INPUTS



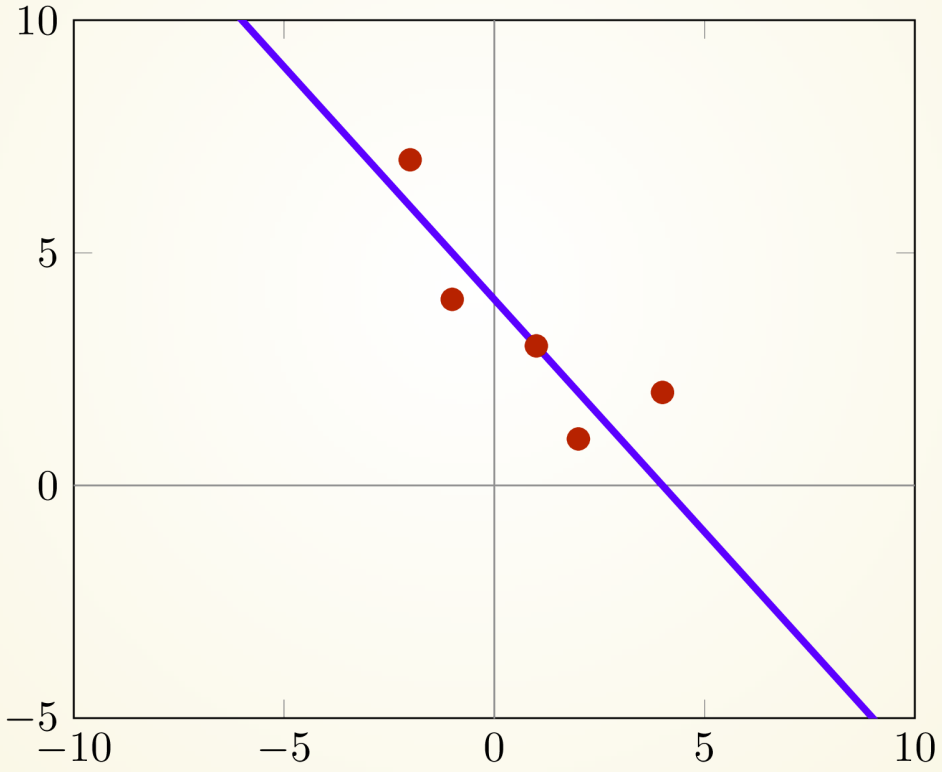
INPUTS

```
X_train = np.array([
    [1250, 350, 3],
    [1700, 900, 6],
    [1400, 600, 3]
])

Y_train = np.array([345000, 580000, 360000])
```

MODEL

Multiply each **feature** by a **weight** and add them up.
Add an **intercept** to get our final **estimate**.

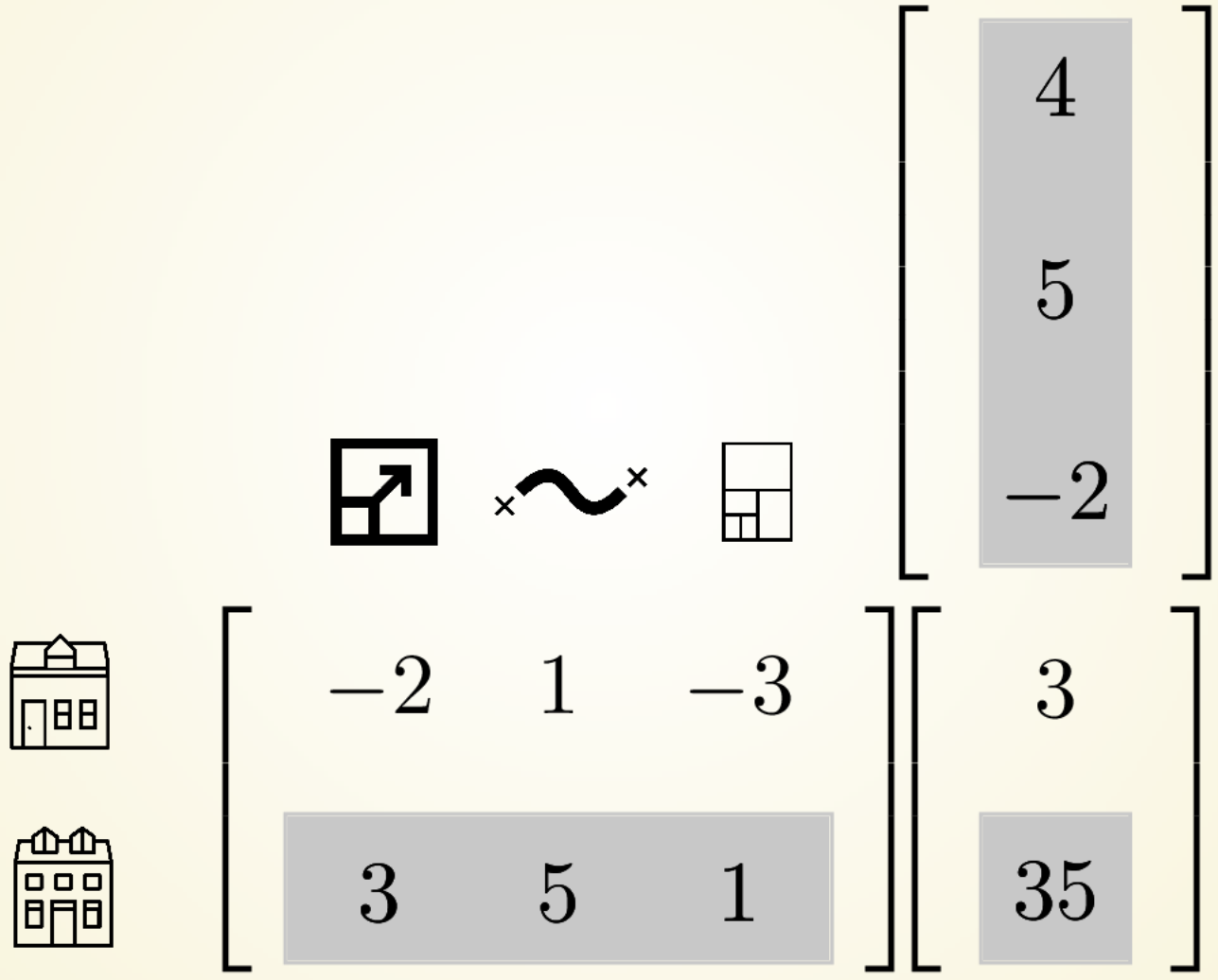
MODEL



MODEL - PARAMETERS

```
weights = np.array([300, -10, -1])
intercept = -26497
```

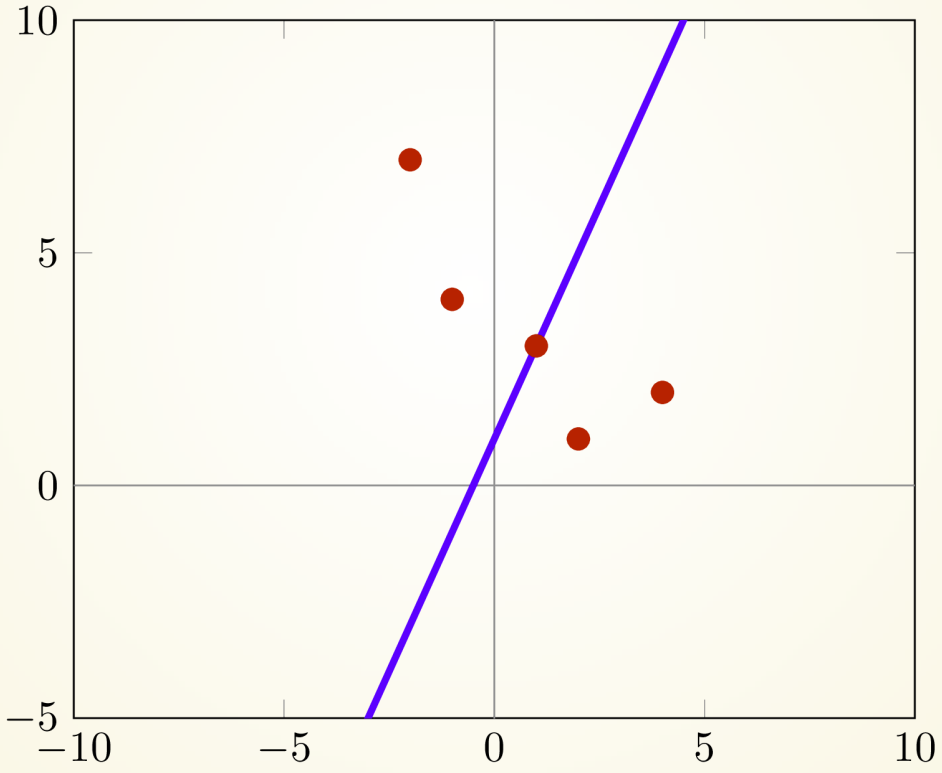
MODEL - OPERATIONS



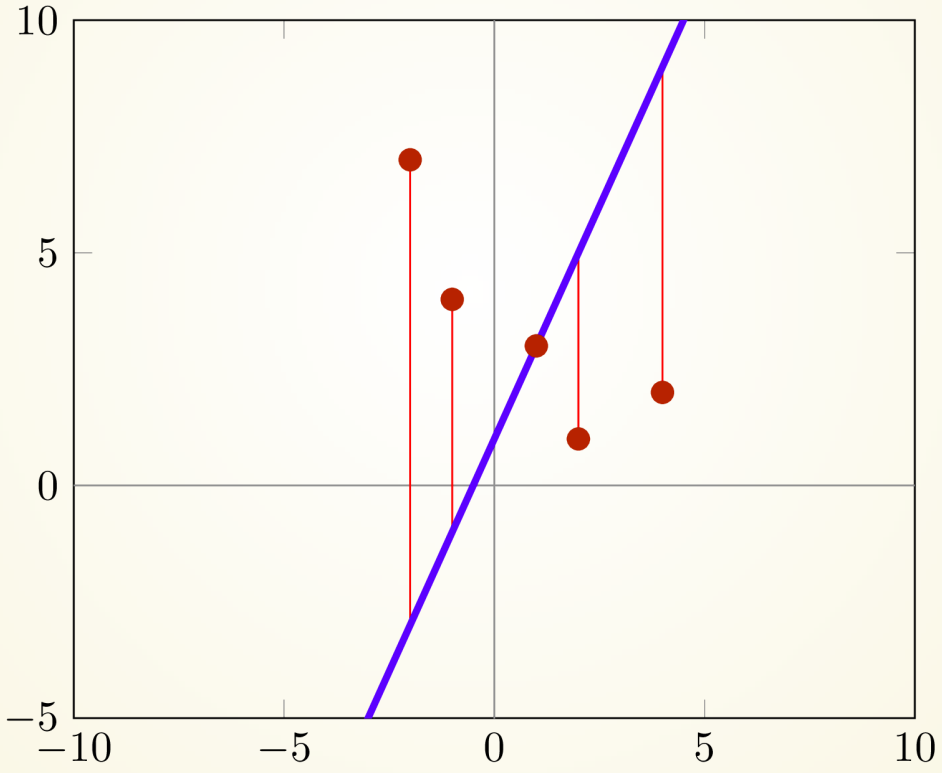
MODEL - OPERATIONS

```
def model(X, weights, intercept):
    return X.dot(weights) + intercept

Y_hat = model(X_train, weights, intercept)
```

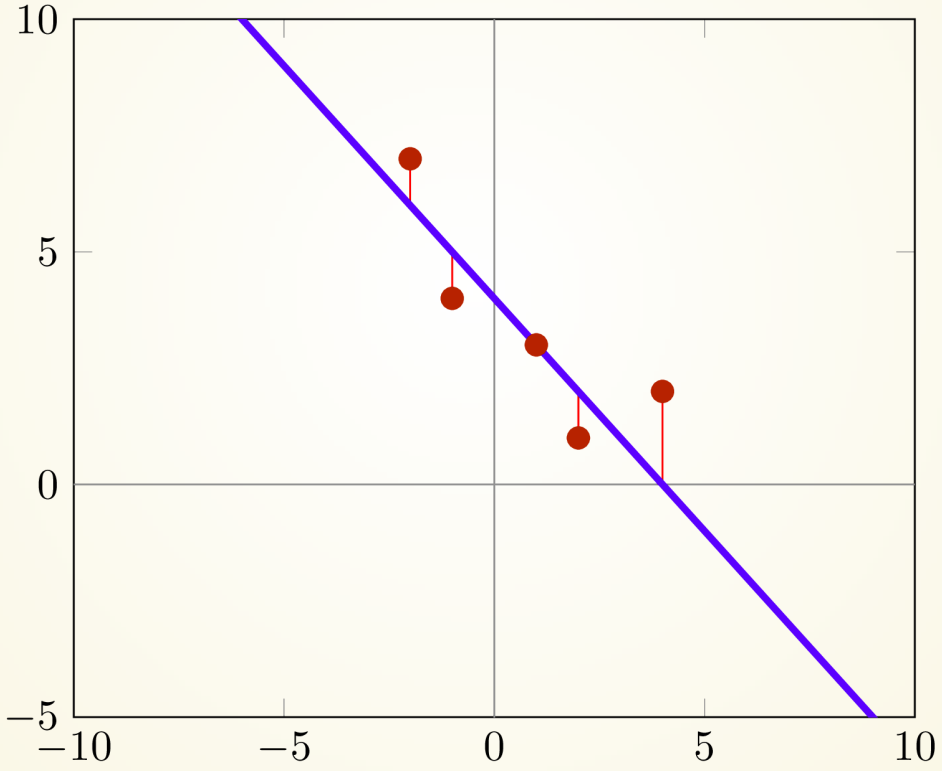
MODEL - COST FUNCTION



MODEL - COST FUNCTION



MODEL - COST FUNCTION



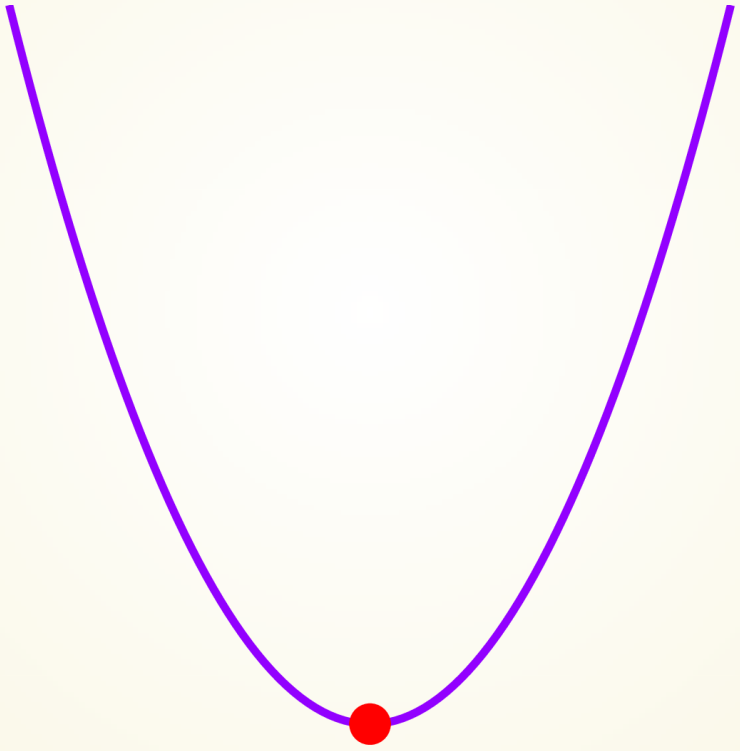
COST FUNCTION

```
def cost(Y_hat, Y):
    return np.sum((Y_hat - Y)**2)
```

OPTIMIZATION

Hold X and Y constant.
Adjust **parameters** to minimize **cost**.

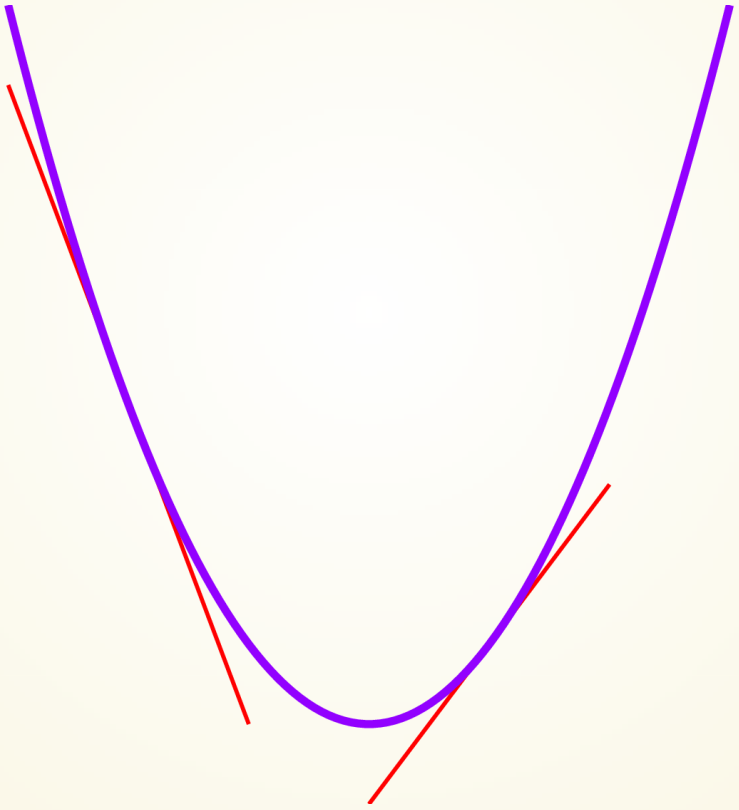
OPTIMIZATION



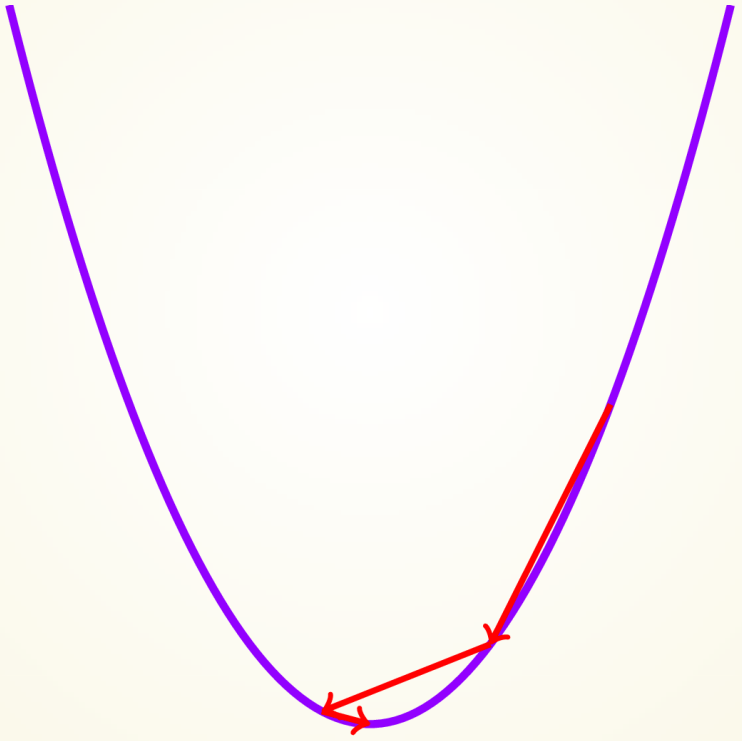
TRIAL AND ERROR

Image source: Wikimedia Commons

OPTIMIZATION



OPTIMIZATION



OPTIMIZATION - GRADIENT CALCULATION

$$\hat{y} = w_0 x_0 + w_1 x_1 + w_2 x_2 + b$$
$$\epsilon = (y - \hat{y})^2$$

Goal: $\frac{\partial \epsilon}{\partial w_i}$, $\frac{\partial \epsilon}{\partial b}$

OPTIMIZATION - GRADIENT CALCULATION

Chain rule: $\frac{\partial \epsilon}{\partial w_i} = \frac{d\epsilon}{d\hat{y}} \frac{\partial \hat{y}}{\partial w_i}$

OPTIMIZATION - GRADIENT CALCULATION

$$\hat{y} = w_0 x_0 + w_1 x_1 + w_2 x_2 + b$$

$$\frac{\partial \hat{y}}{\partial w_0} = x_0$$

OPTIMIZATION - GRADIENT CALCULATION

$$\epsilon = (y - \hat{y})^2$$

$$\frac{d\epsilon}{d\hat{y}} = -2(y - \hat{y})$$

OPTIMIZATION - GRADIENT CALCULATION

$$\frac{\partial \hat{y}}{\partial w_0} = x_0$$

$$\frac{d\epsilon}{d\hat{y}} = -2(y - \hat{y})$$

$$\frac{\partial \epsilon}{\partial w_0} = -2(y - \hat{y})x_0$$

OPTIMIZATION - GRADIENT CALCULATION

$$\hat{y} = w_0 x_0 + w_1 x_1 + w_2 x_2 + b \cdot 1$$

$$\frac{\partial \epsilon}{\partial b} = -2(y - \hat{y}) \cdot 1$$

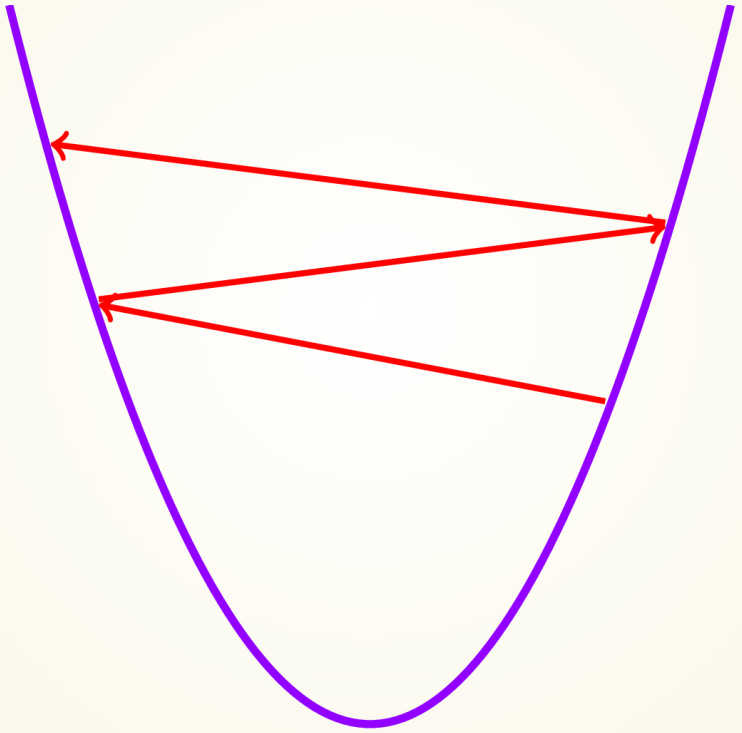
OPTIMIZATION - GRADIENT CALCULATION

```
delta_y = y - y_hat
gradient_weights = -2 * delta_y * weights
gradient_intercept = -2 * delta_y * 1
```

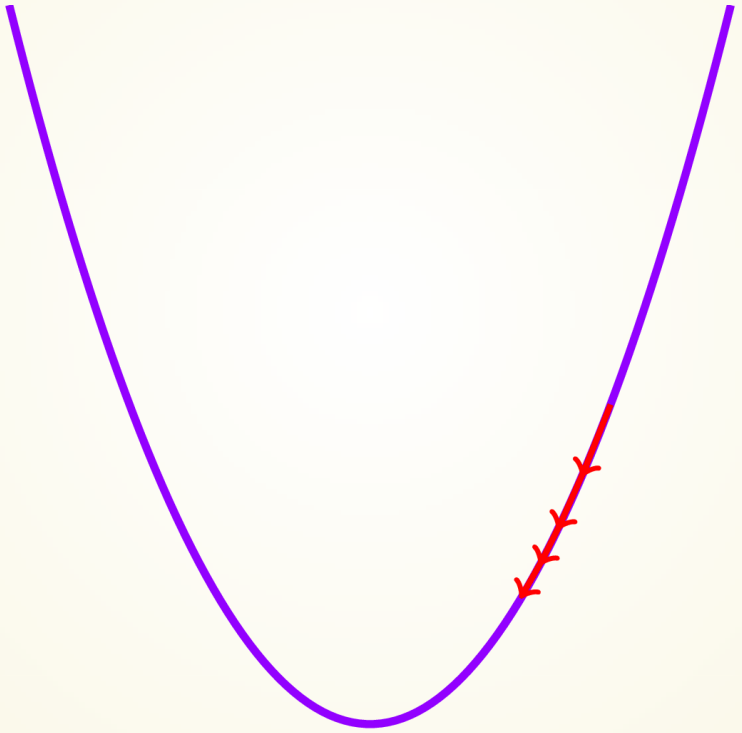
OPTIMIZATION - PARAMETER UPDATE

```
weights = weights - gradient_weights
intercept = intercept - gradient_intercept
```

OPTIMIZATION - OVERSHOOT



OPTIMIZATION - UNDERSHOOT



OPTIMIZATION - PARAMETER UPDATE

```
learning_rate = 0.05

weights = weights - \
              learning_rate * gradient_weights
intercept = intercept - \
              learning_rate * gradient_intercept
```

TRAINING

```
def training_round(x, y, weights, intercept,
                   alpha=learning_rate):
    # calculate our estimate
    y_hat = model(x, weights, intercept)

    # calculate error
    delta_y = y - y_hat

    # calculate gradients
    gradient_weights = -2 * delta_y * weights
    gradient_intercept = -2 * delta_y

    # update parameters
    weights = weights - alpha * gradient_weights
    intercept = intercept - alpha * gradient_intercept

    return weights, intercept
```

TRAINING

```
NUM_EPOCHS = 100

def train(X, Y):
    # initialize parameters
    weights = np.random.randn(3)
    intercept = 0

    # training rounds
    for i in range(NUM_EPOCHS):
        for (x, y) in zip(X, Y):
            weights, intercept = training_round(x, y,
                                                weights, intercept)
```

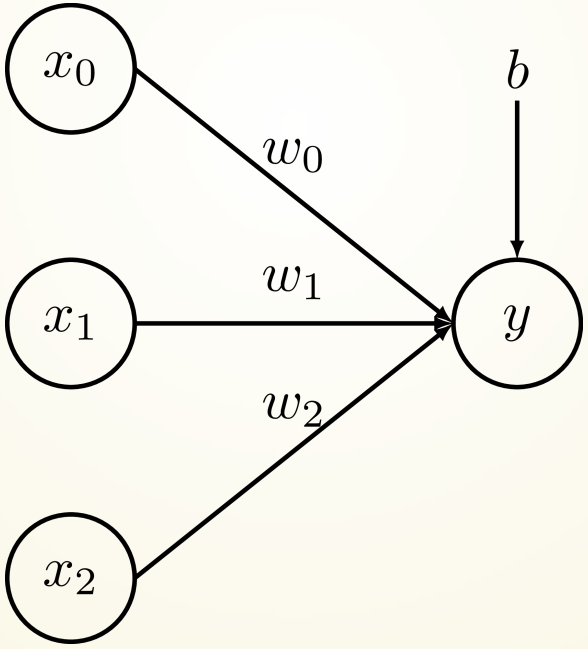
TESTING

```
def test(X_test, Y_test, weights, intercept):
    Y_predicted = model(X_test, weights, intercept)
    error = cost(Y_predicted, Y_test)
    return np.sqrt(np.mean(error))

>>> test(X_test, Y_test, final_weights, final_intercept)
6052.79
```

Uh, wasn't this supposed to be a talk about neural networks? Why are we talking about linear regression?

SURPRISE!
YOU'VE ALREADY MADE A NEURAL NETWORK!

LINEAR REGRESSION = SIMPLEST NEURAL NETWORK



ONCE MORE, WITH TENSORFLOW

- Inputs
- Model - Parameters
- Model - Operations
- Cost function
- Optimization
- Train
- Test

INPUTS ⟶ PLACEHOLDERS

```
import tensorflow as tf

X = tf.placeholder(tf.float32, [None, 3])
Y = tf.placeholder(tf.float32, [None, 1])
```

PARAMETERS ⟶ VARIABLES

```
# create tf.Variable(s)
W = tf.get_variable("weights", [3, 1],
       initializer=tf.random_normal_initializer())
b = tf.get_variable("intercept", [1],
       initializer=tf.constant_initializer(0))
```

OPERATIONS

```
Y_hat = tf.matmul(X, W) + b
```

COST FUNCTION

```
cost = tf.reduce_mean(tf.square(Y_hat - Y))
```

OPTIMIZATION

```
learning_rate = 0.05
optimizer = tf.train.GradientDescentOptimizer
                (learning_rate).minimize(cost)
```

TRAINING

```
with tf.Session() as sess:
    # initialize variables
    sess.run(tf.global_variables_initializer())

    # train
    for _ in range(NUM_EPOCHS):
        for (X_batch, Y_batch) in get_minibatches(
                X_train, Y_train, BATCH_SIZE):
            sess.run(optimizer,
                     feed_dict={
                         X: X_batch,
                         Y: Y_batch
                     })
```

TRAINING

```
with tf.Session() as sess:
    # initialize variables
    sess.run(tf.global_variables_initializer())

    # train
    for _ in range(NUM_EPOCHS):
        for (X_batch, Y_batch) in get_minibatches(
                X_train, Y_train, BATCH_SIZE):
            sess.run(optimizer,
                     feed_dict={
                         X: X_batch,
                         Y: Y_batch
                     })
```

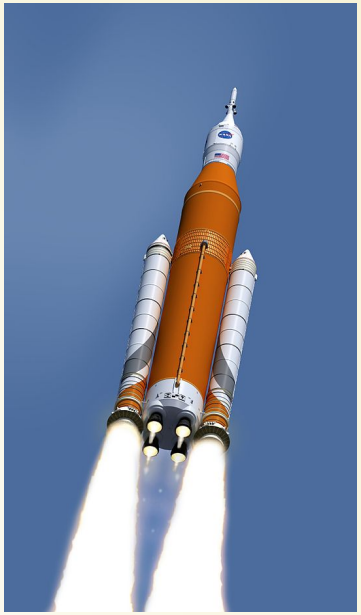
TRAINING

```
with tf.Session() as sess:
    # initialize variables
    sess.run(tf.global_variables_initializer())

    # train
    for _ in range(NUM_EPOCHS):
        for (X_batch, Y_batch) in get_minibatches(
                X_train, Y_train, BATCH_SIZE):
            sess.run(optimizer,
                     feed_dict={
                         X: X_batch,
                         Y: Y_batch
                     })
```

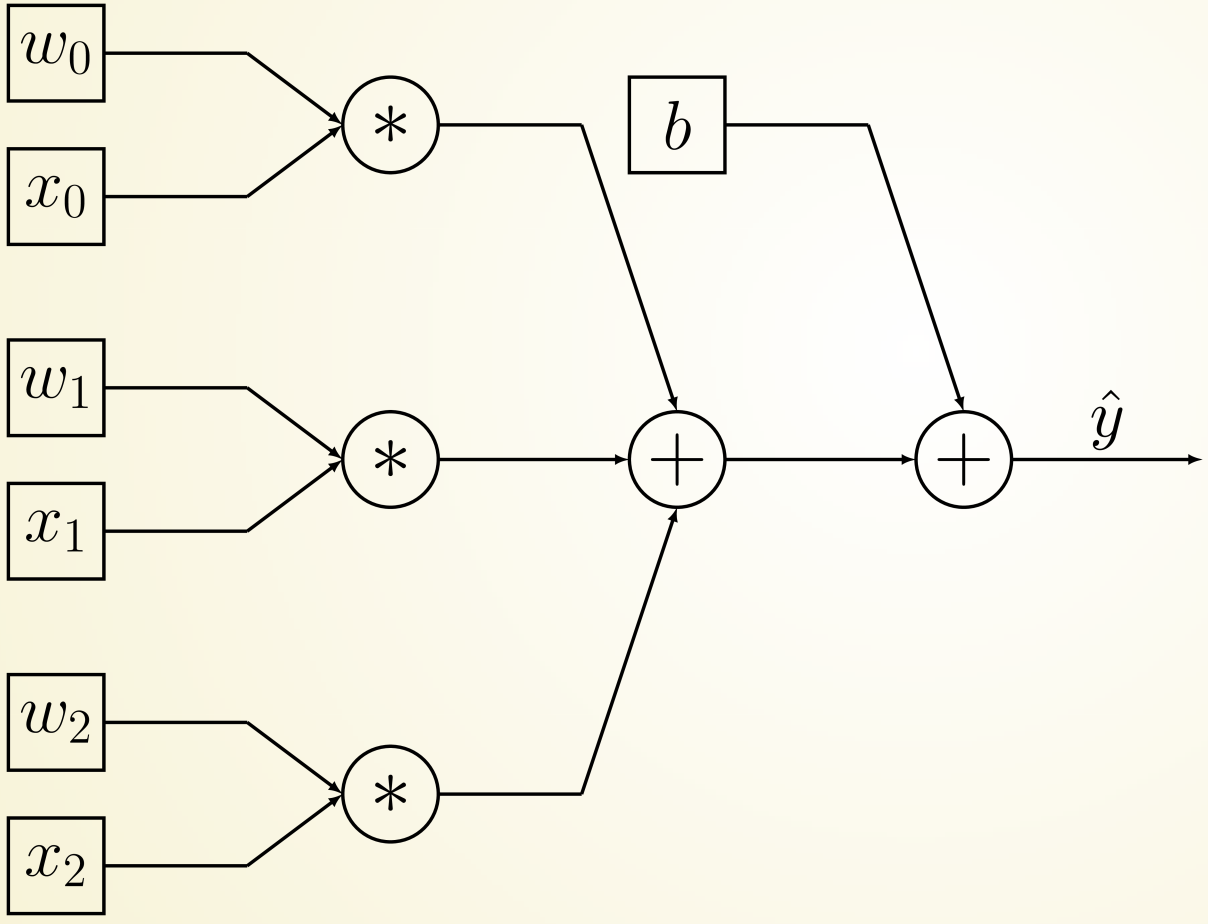
TRAINING

```
with tf.Session() as sess:
    # initialize variables
    sess.run(tf.global_variables_initializer())

    # train
    for _ in range(NUM_EPOCHS):
        for (X_batch, Y_batch) in get_minibatches(
                X_train, Y_train, BATCH_SIZE):
            sess.run(optimizer,
                     feed_dict={
                         X: X_batch,
                         Y: Y_batch
                     })
```

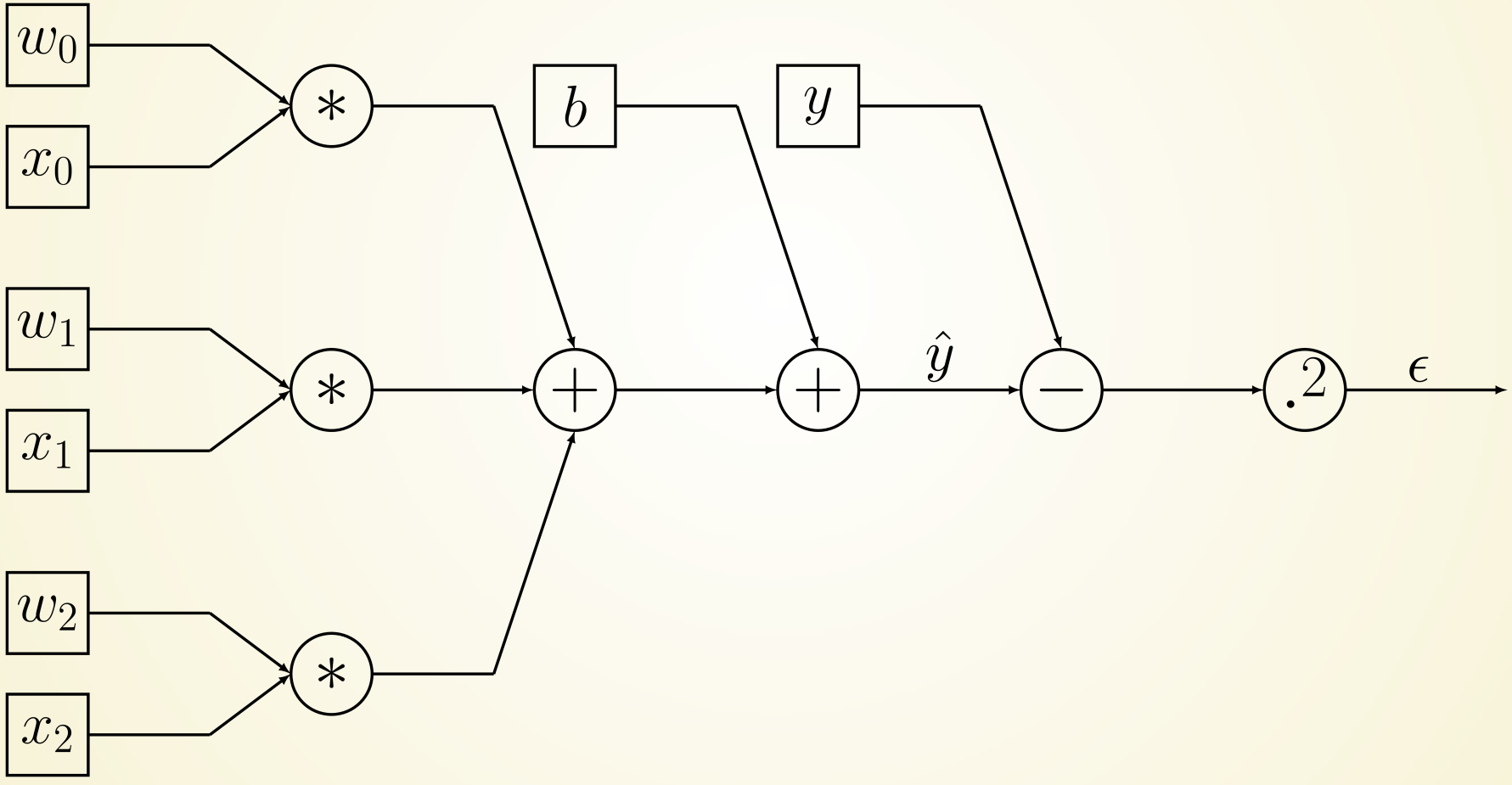
```
# Placeholders
X = tf.placeholder(tf.float32, [None, 3])
Y = tf.placeholder(tf.float32, [None, 1])

# Parameters/Variables
W = tf.get_variable("weights", [3, 1],
       initializer=tf.random_normal_initializer())
b = tf.get_variable("intercept", [1],
       initializer=tf.constant_initializer(0))

# Operations
Y_hat = tf.matmul(X, W) + b

# Cost function
cost = tf.reduce_mean(tf.square(Y_hat - Y))

# Optimization
optimizer = tf.train.GradientDescentOptimizer
               (learning_rate).minimize(cost)

# ------------------------------------------------

# Train
with tf.Session() as sess:
    # initialize variables
    sess.run(tf.global_variables_initializer())

    # run training rounds
    for _ in range(NUM_EPOCHS):
        for X_batch, Y_batch in get_minibatches(
                    X_train, Y_train, BATCH_SIZE):
            sess.run(optimizer,
               feed_dict={X: X_batch, Y: Y_batch})
```



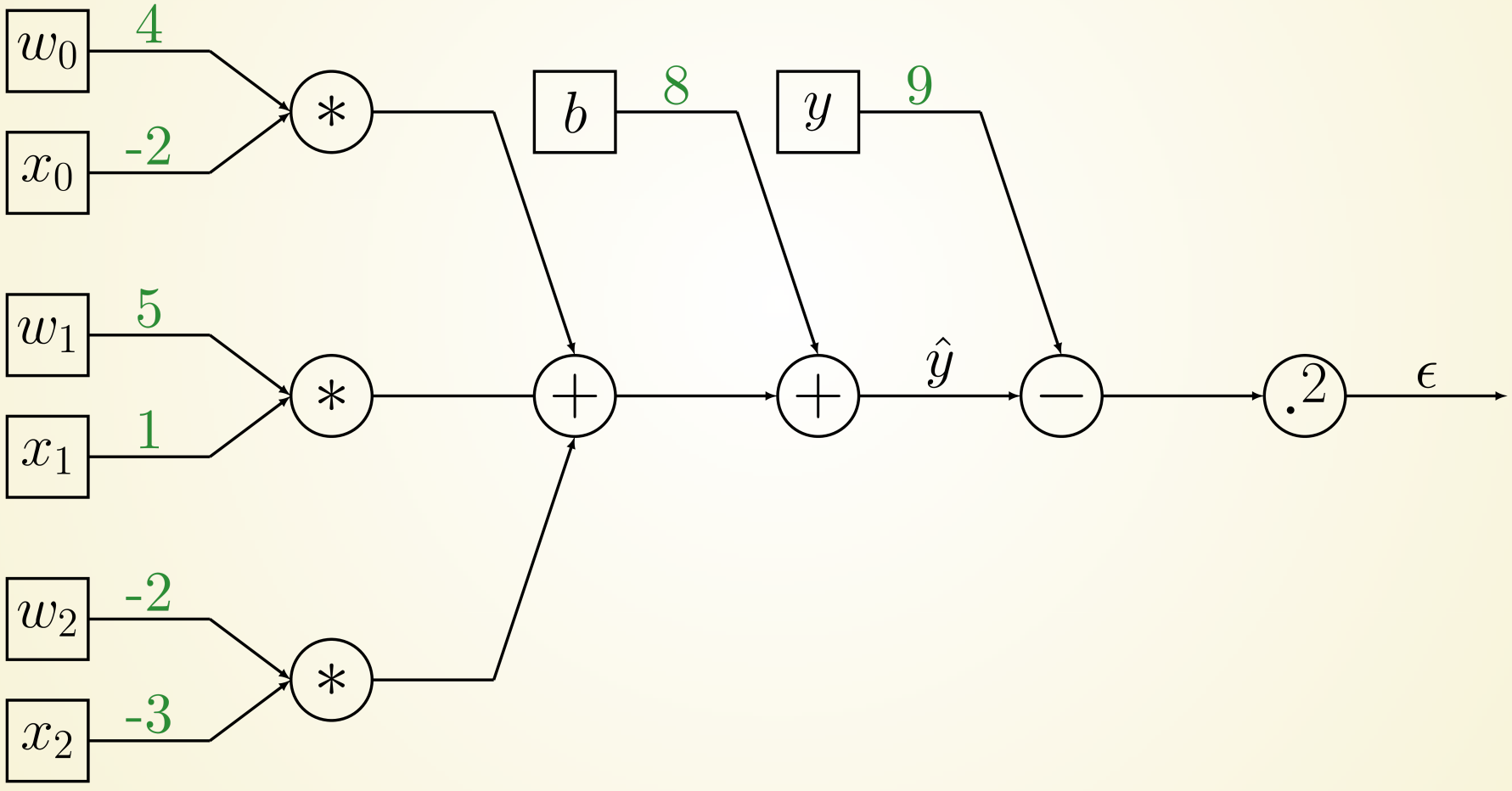
COMPUTATION GRAPH



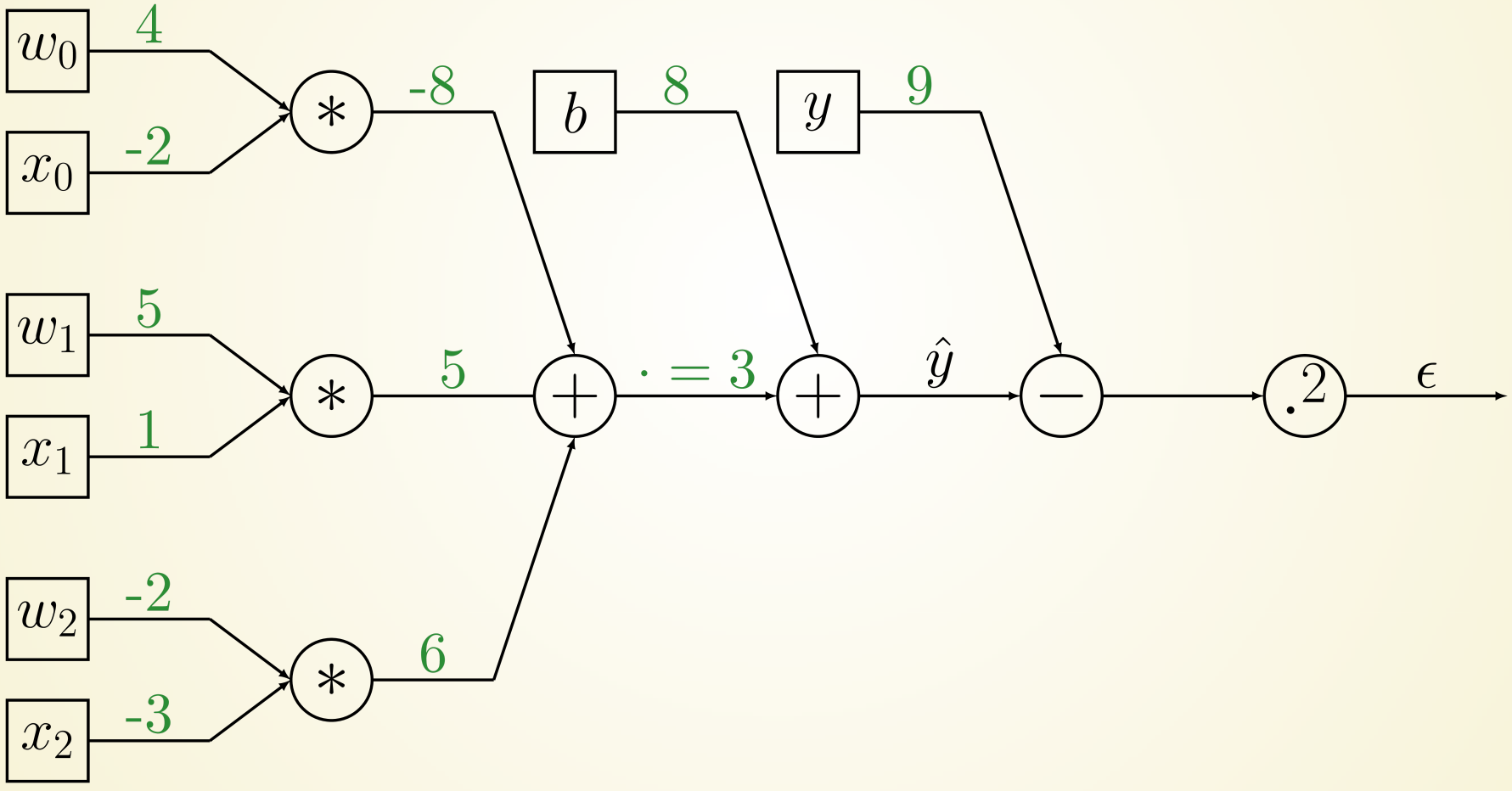
COMPUTATION GRAPH



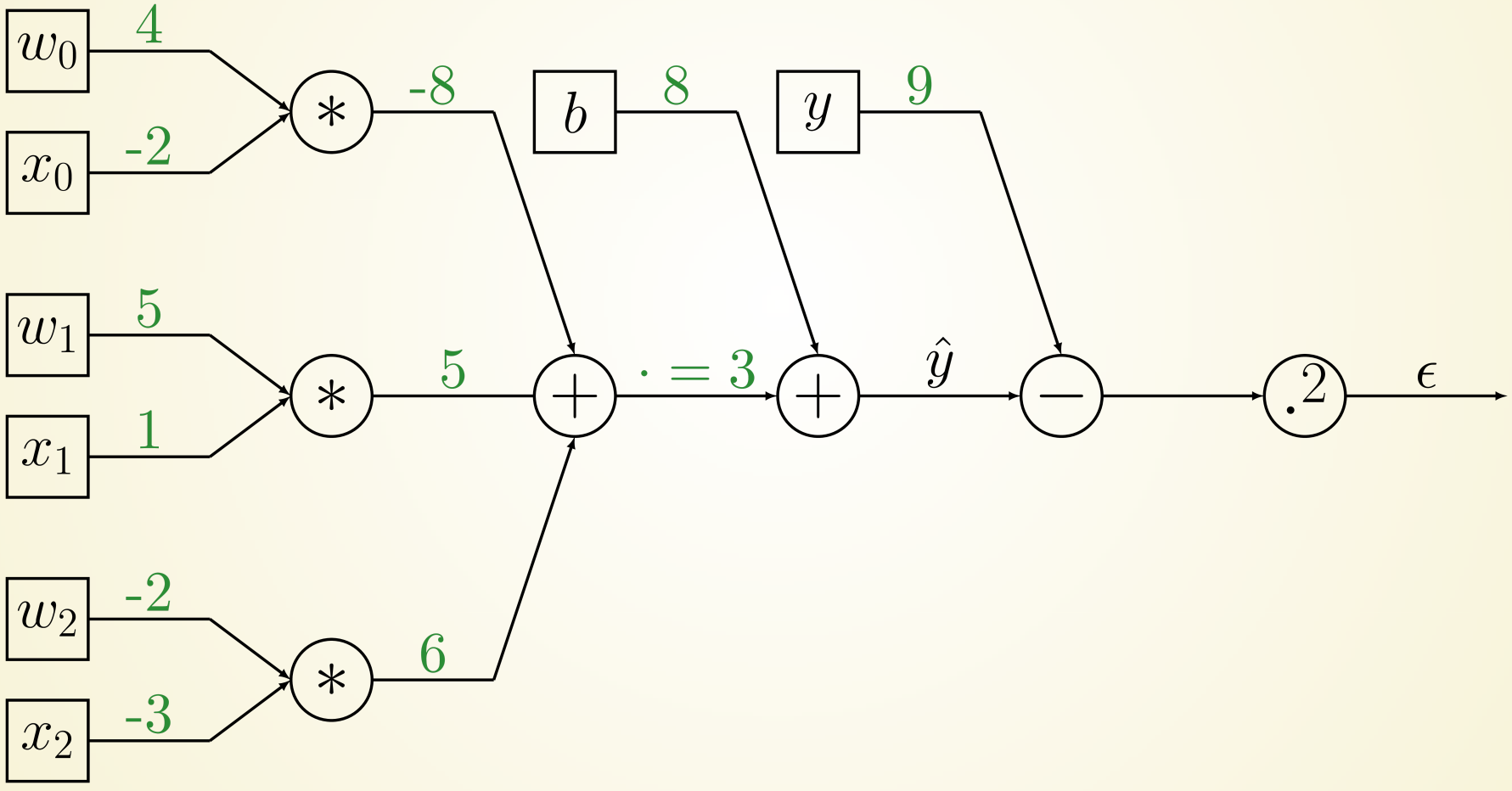
FORWARD PROPAGATION



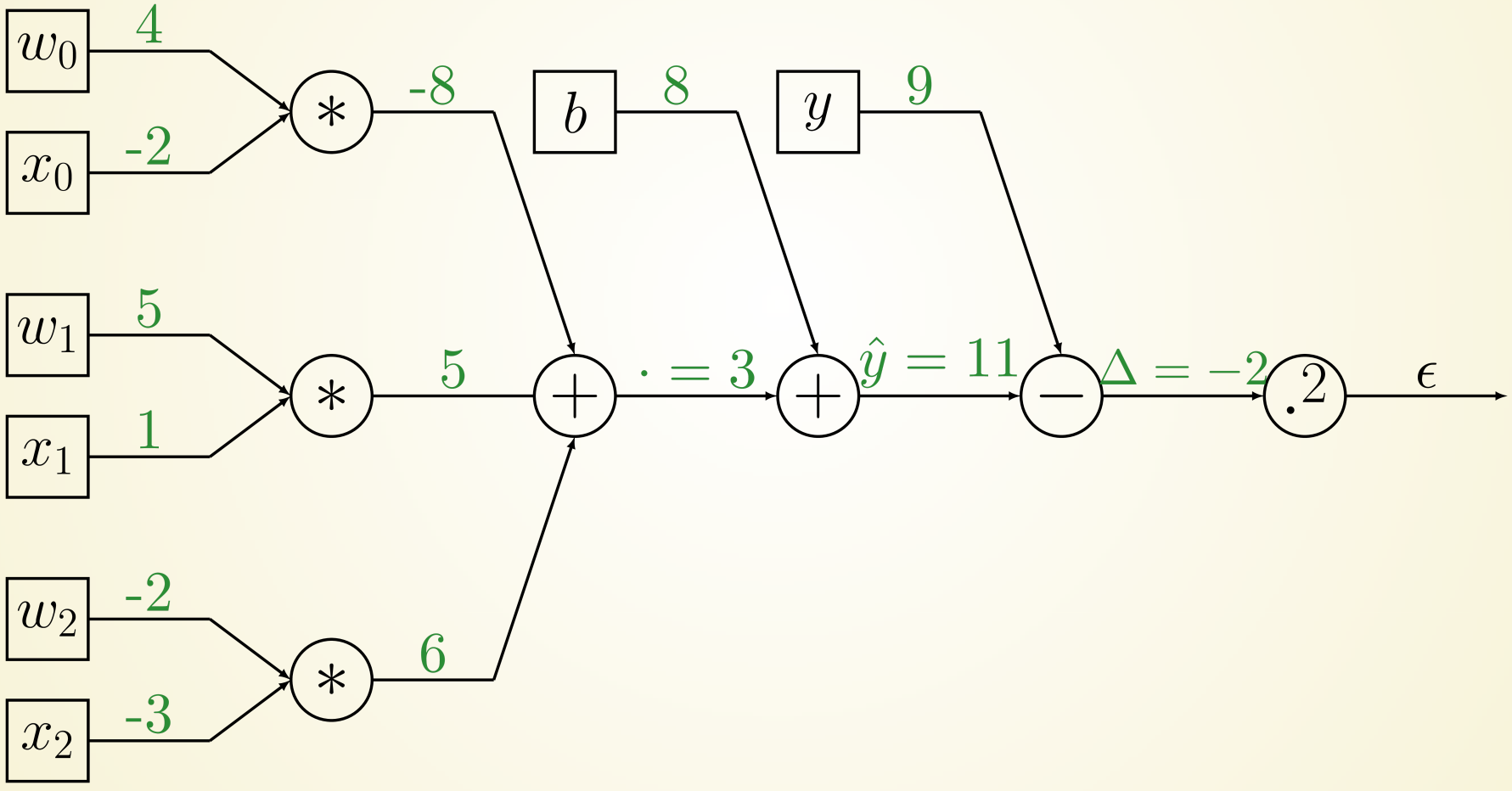
FORWARD PROPAGATION



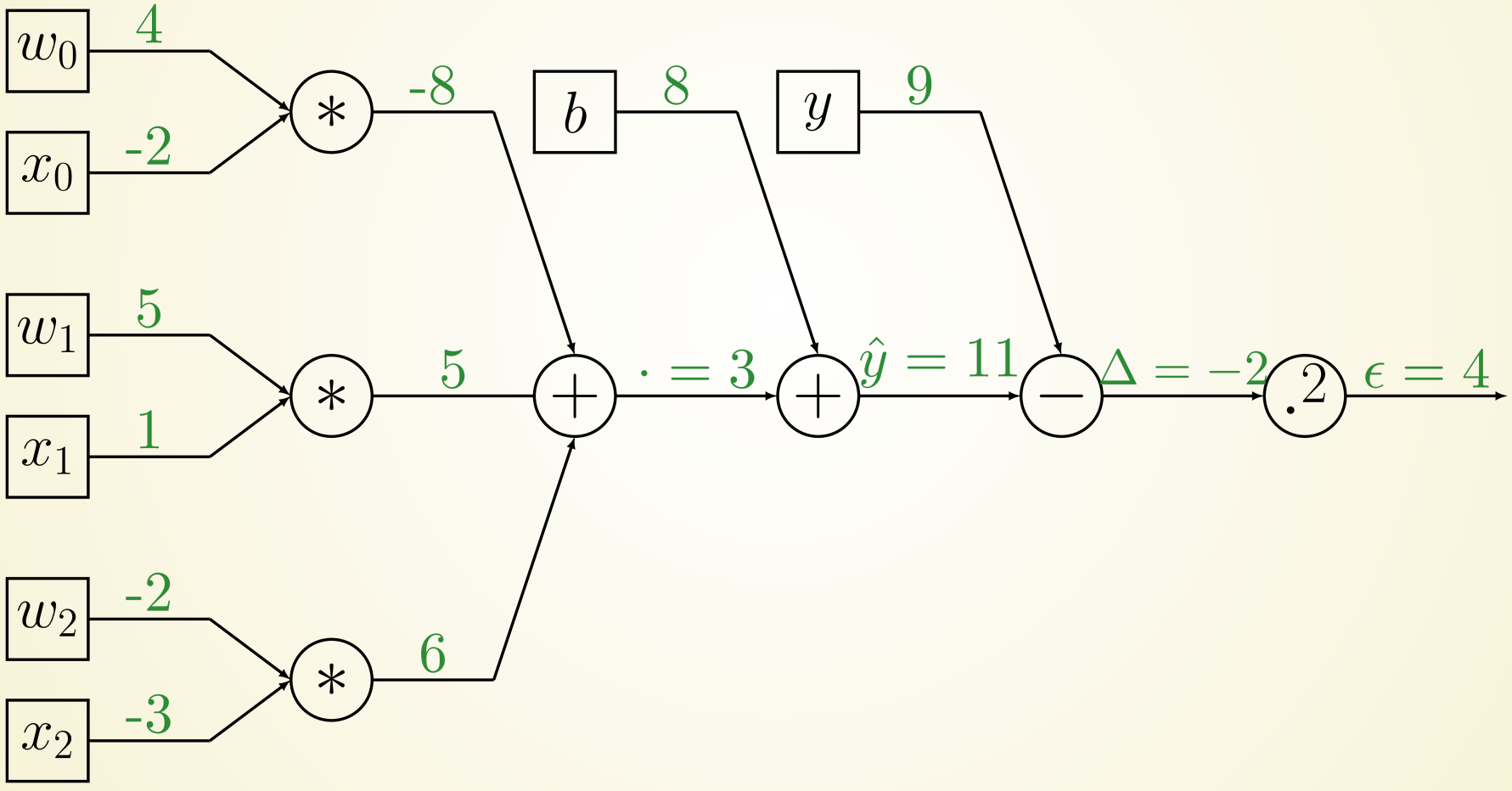
FORWARD PROPAGATION



FORWARD PROPAGATION



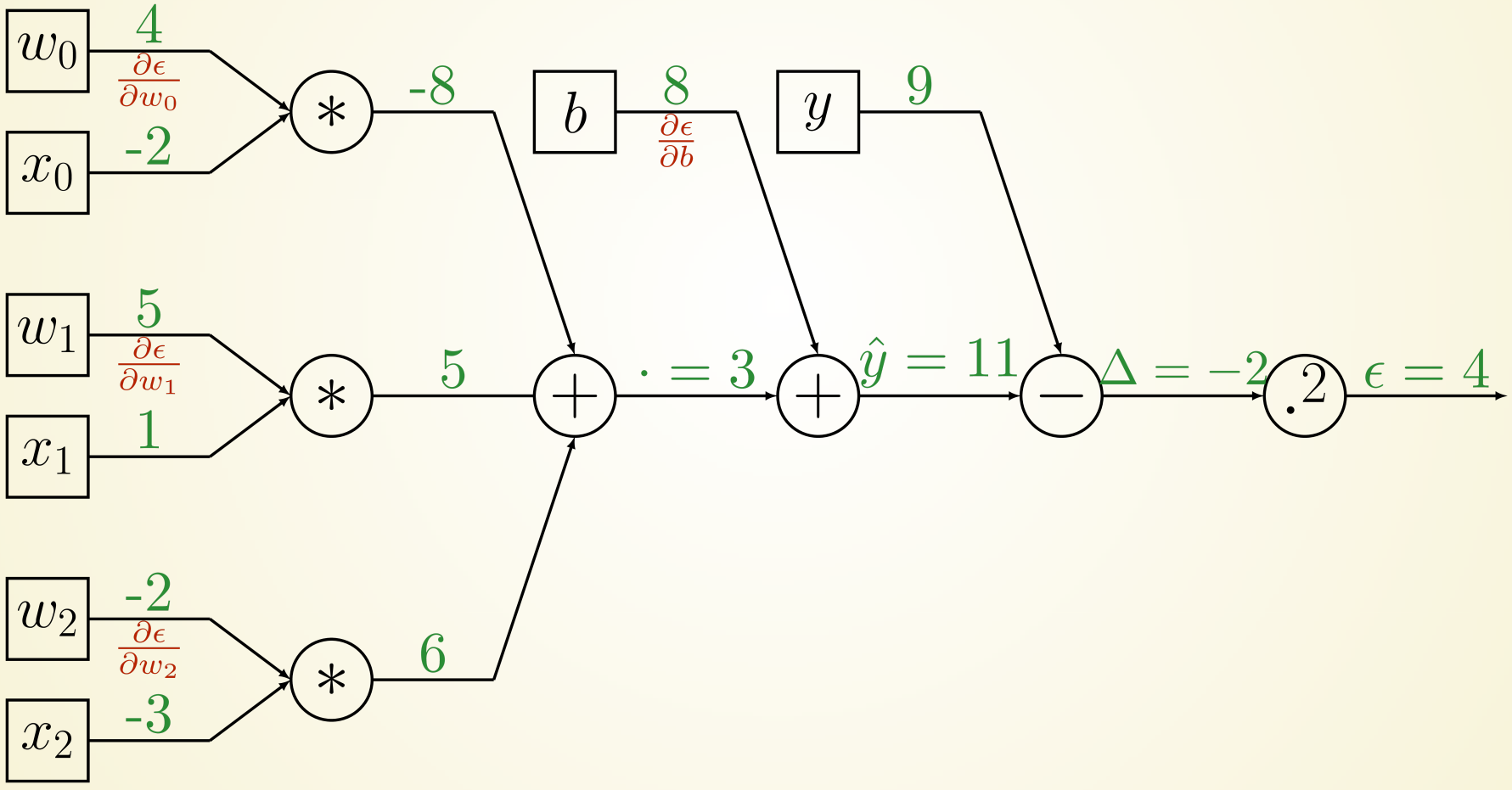
FORWARD PROPAGATION



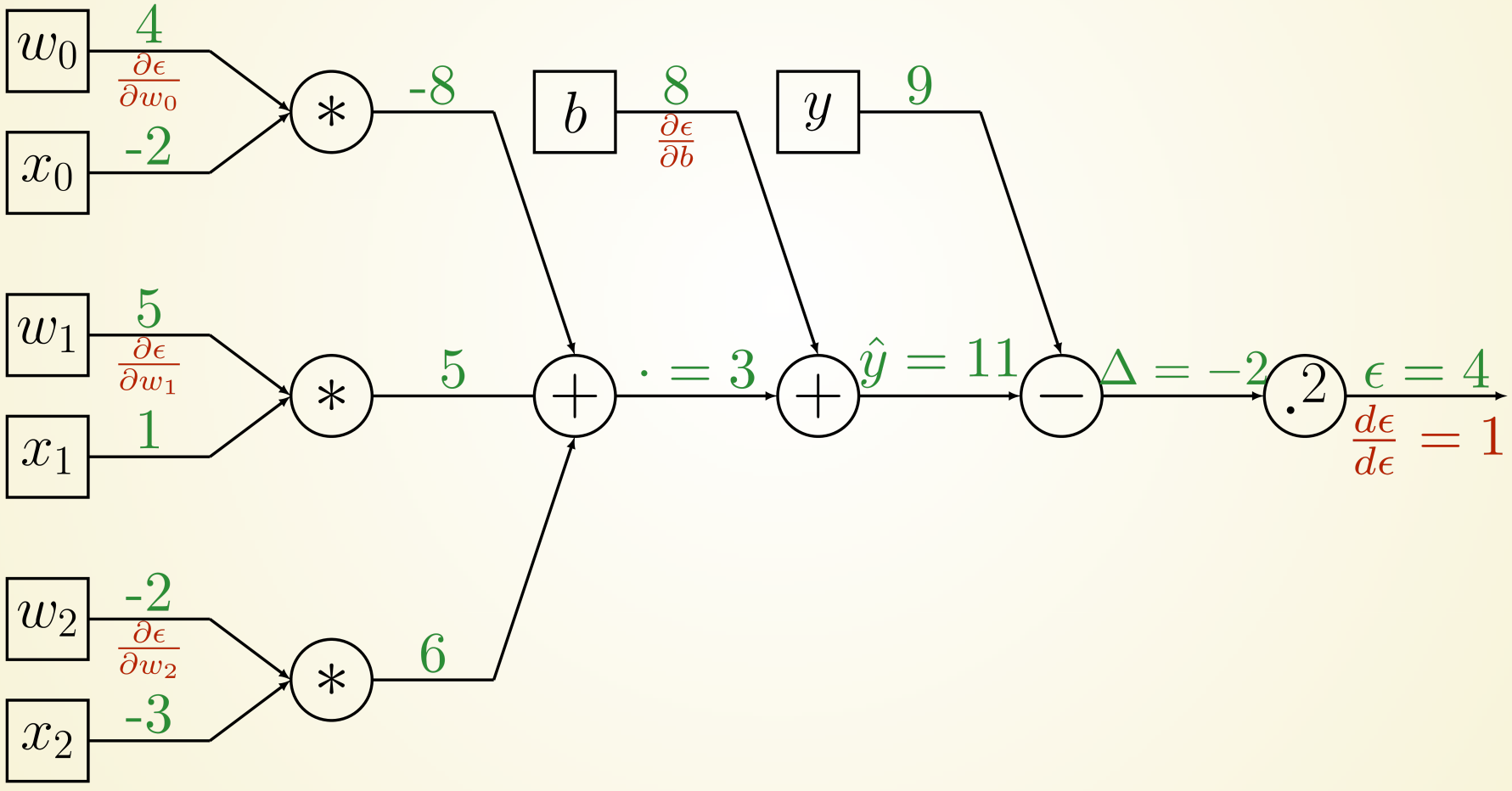
FORWARD PROPAGATION

```
def training_round(x, y, weights, intercept,
                   alpha=learning_rate):
    # calculate our estimate
    y_hat = model(x, weights, intercept)

    # calculate error
    delta_y = y - y_hat

    # calculate gradients
    gradient_weights = -2 * delta_y * weights
    gradient_intercept = -2 * delta_y

    # update parameters
    weights = weights - alpha * gradient_weights
    intercept = intercept - alpha * gradient_intercept

    return weights, intercept
```

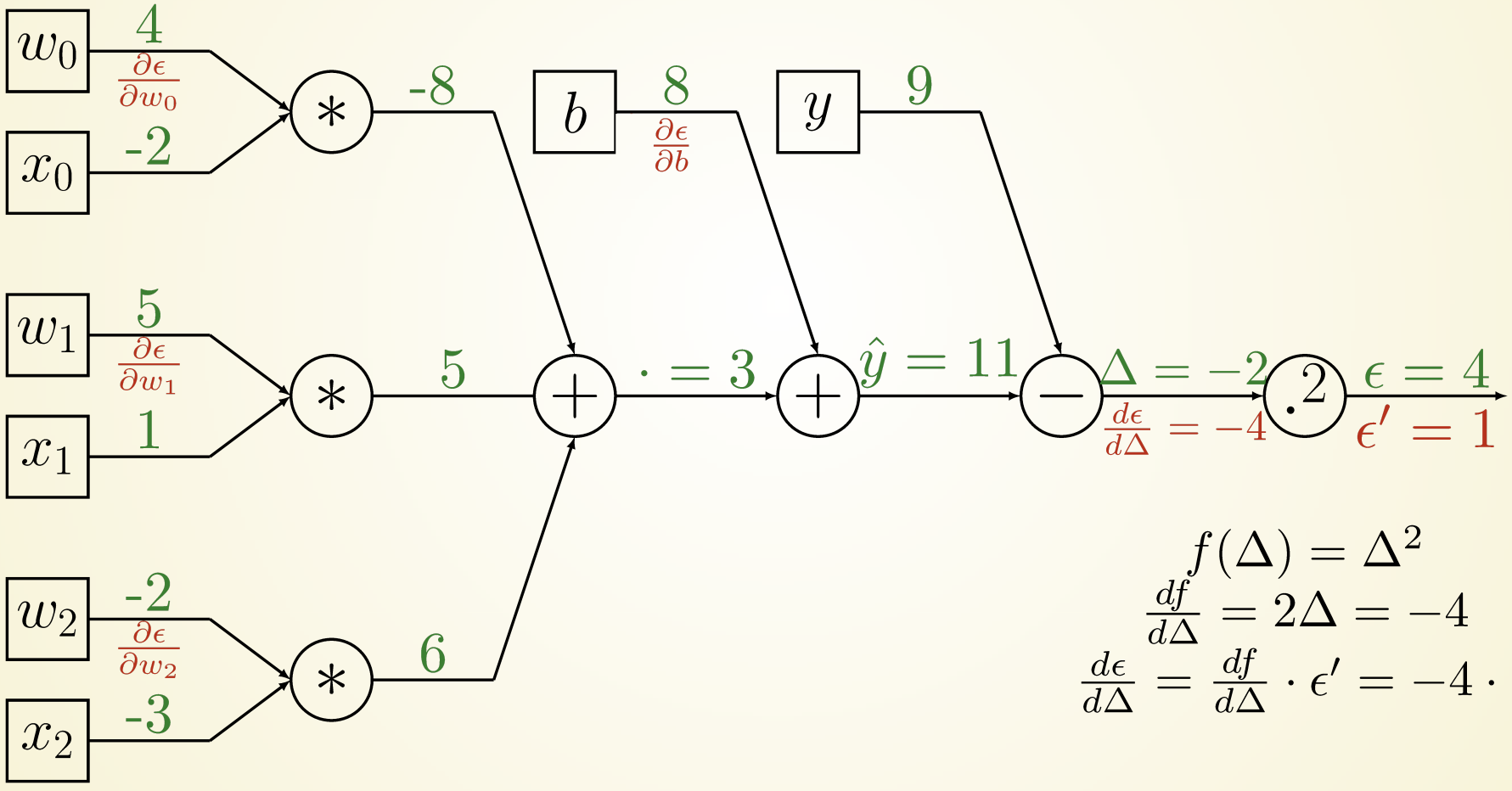
BACKPROPAGATION



BACKPROPAGATION

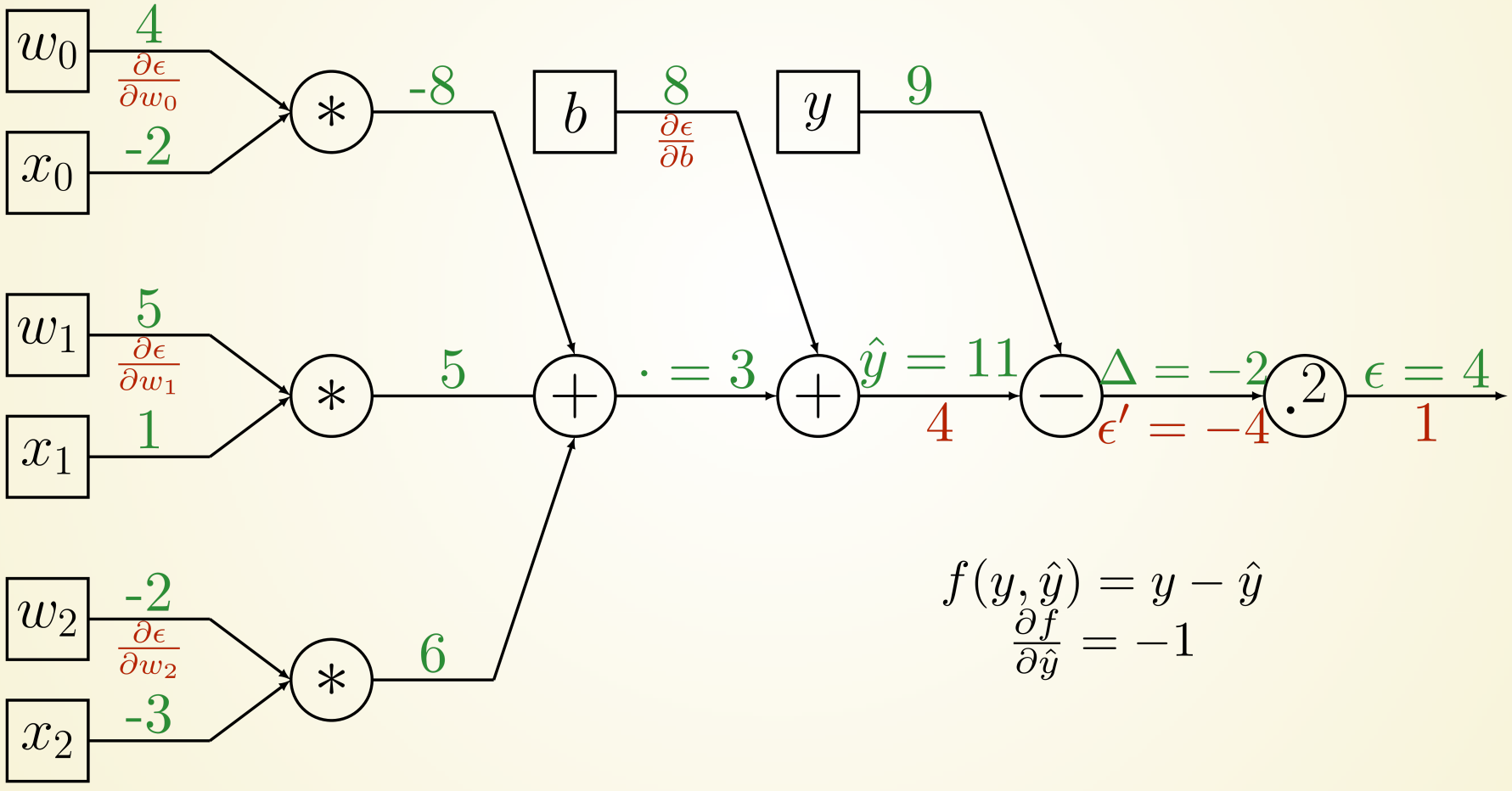


BACKPROPAGATION



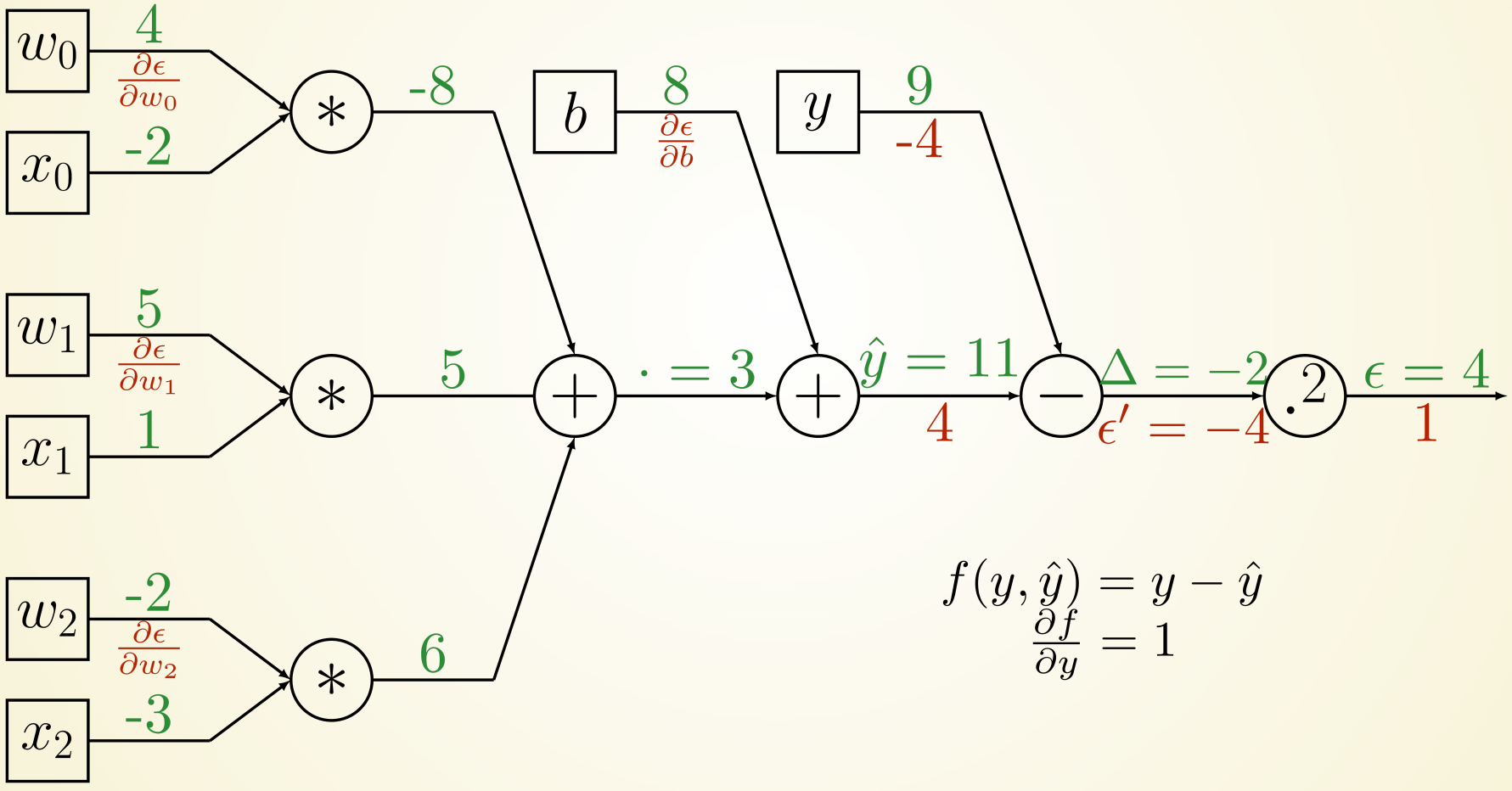
$$f(\Delta) = \Delta^2$$
$$\frac{df}{d\Delta} = 2\Delta = -4$$
$$\frac{d\epsilon}{d\Delta} = \frac{df}{d\Delta} \cdot \epsilon' = -4 \cdot 1$$

BACKPROPAGATION



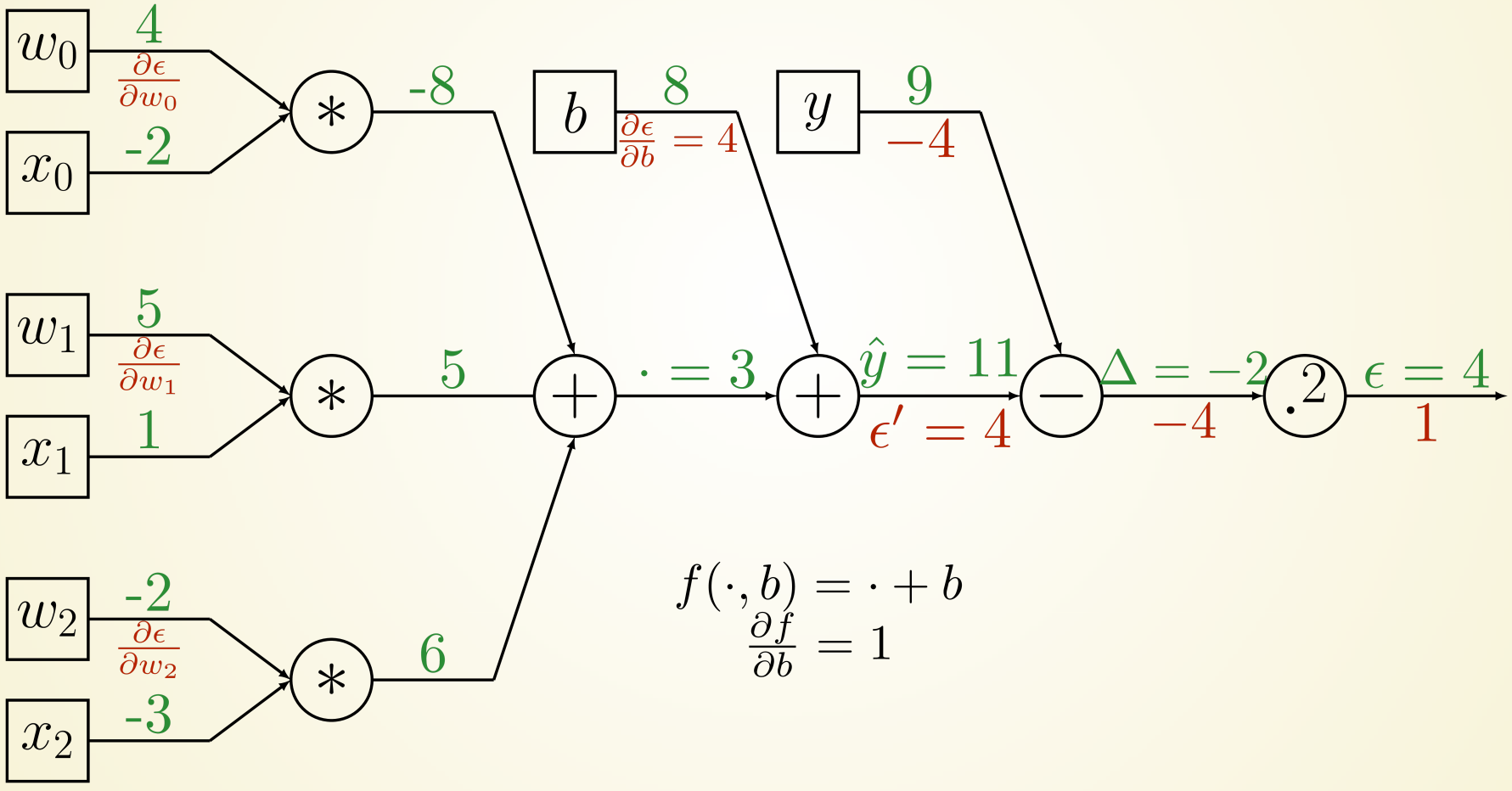
$$f(y, \hat{y}) = y - \hat{y}$$
$$\frac{\partial f}{\partial \hat{y}} = -1$$

BACKPROPAGATION

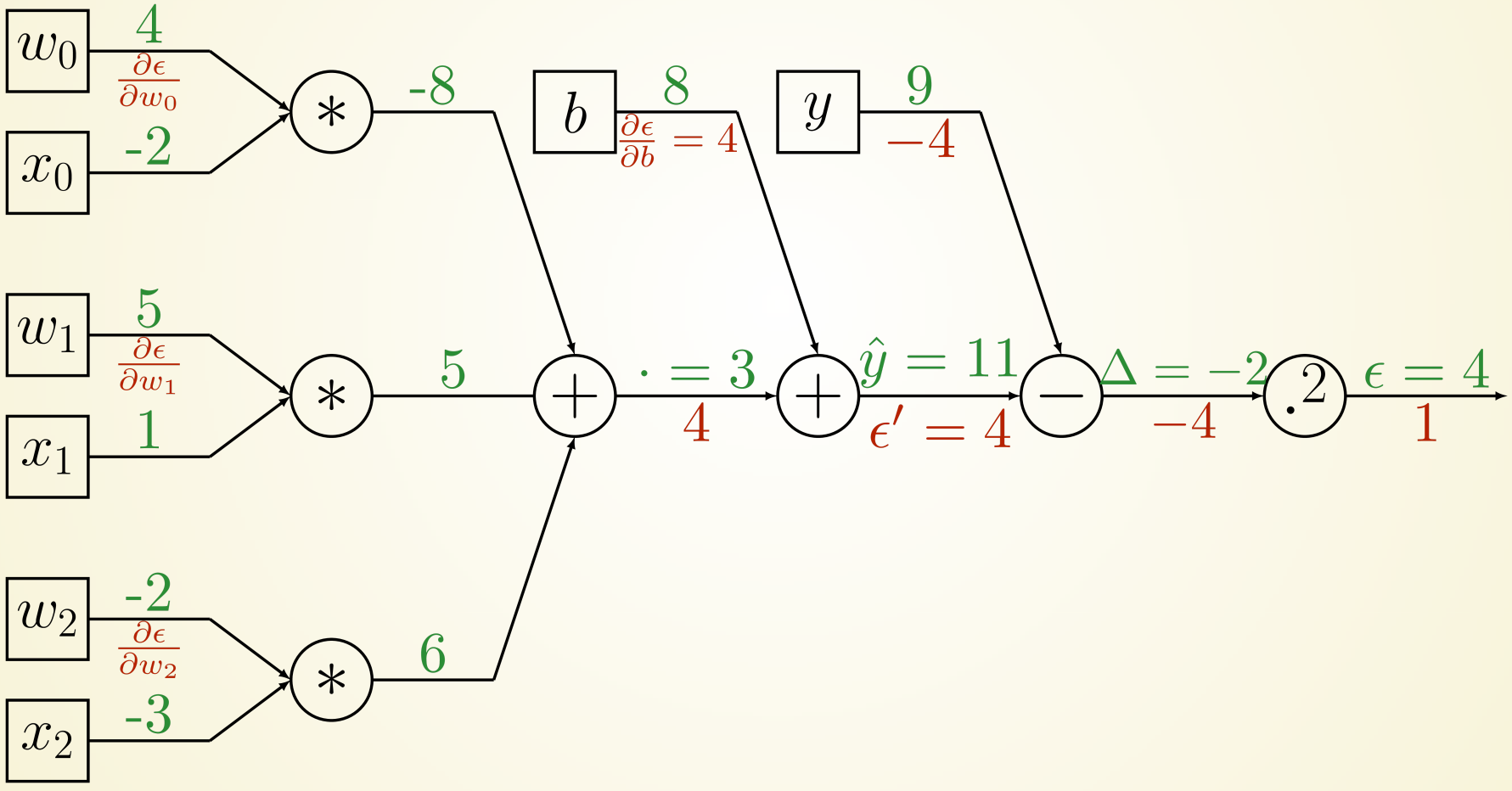


$$f(y, \hat{y}) = y - \hat{y}$$
$$\frac{\partial f}{\partial y} = 1$$

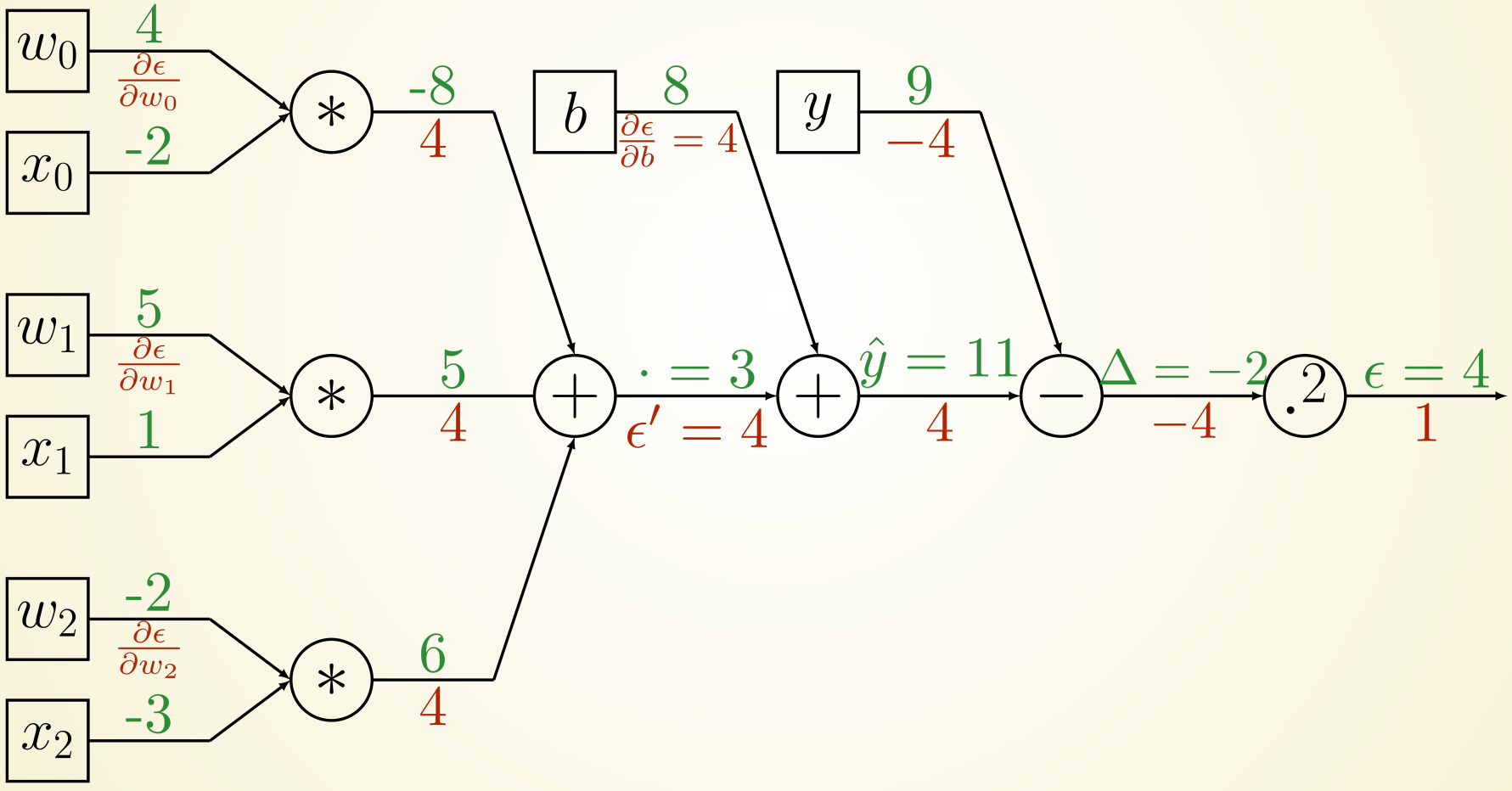
BACKPROPAGATION



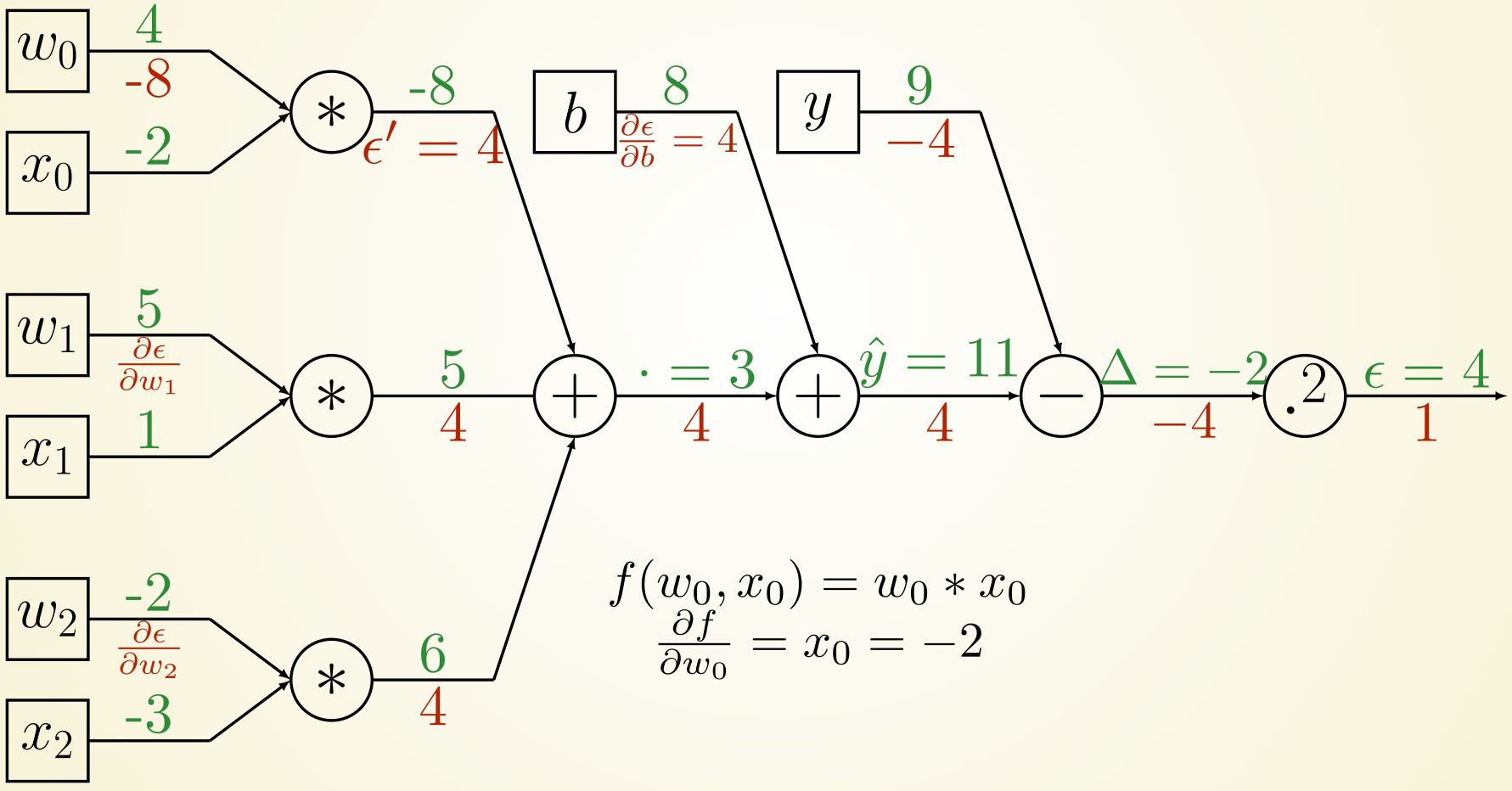
BACKPROPAGATION



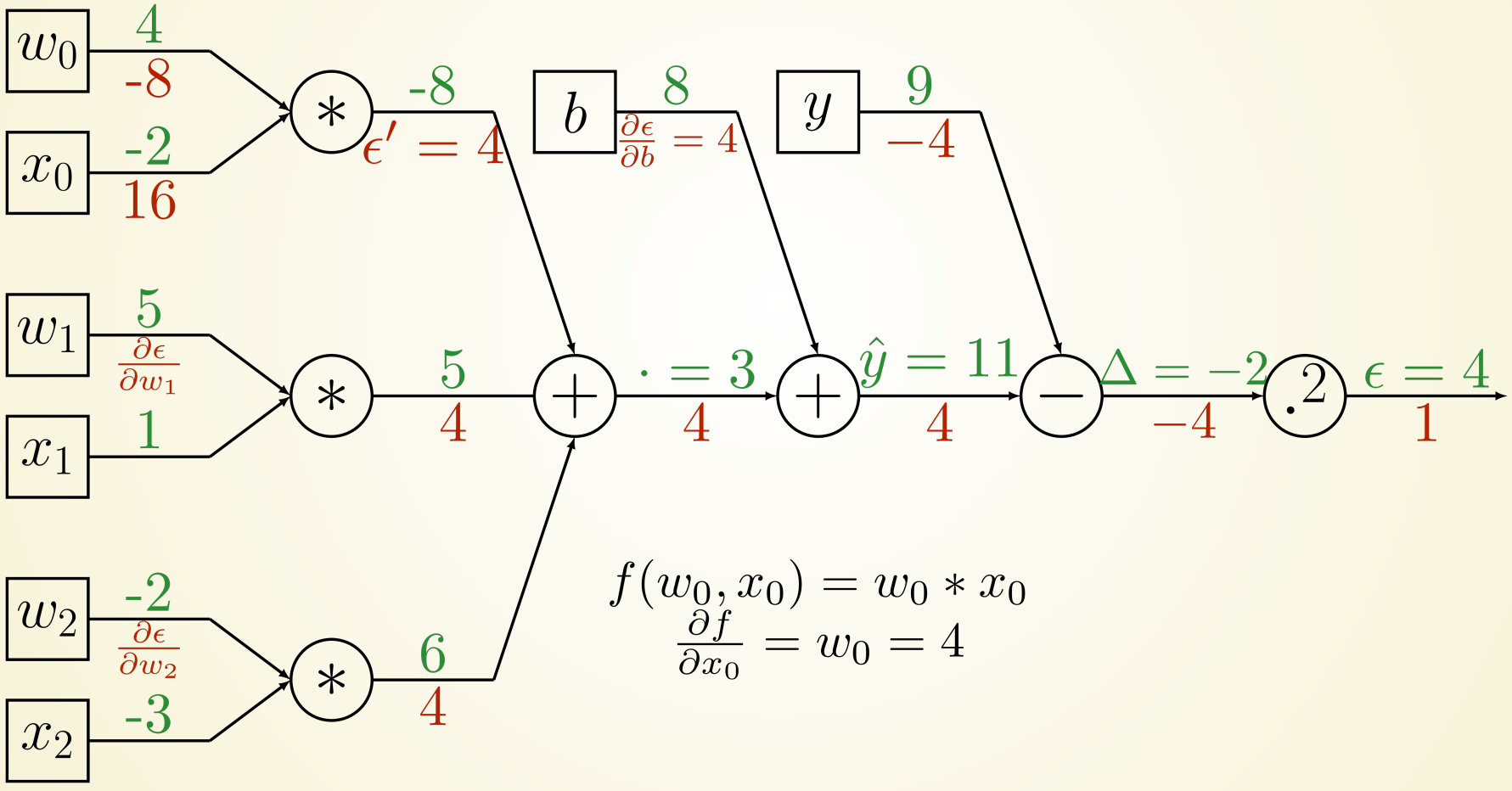
BACKPROPAGATION



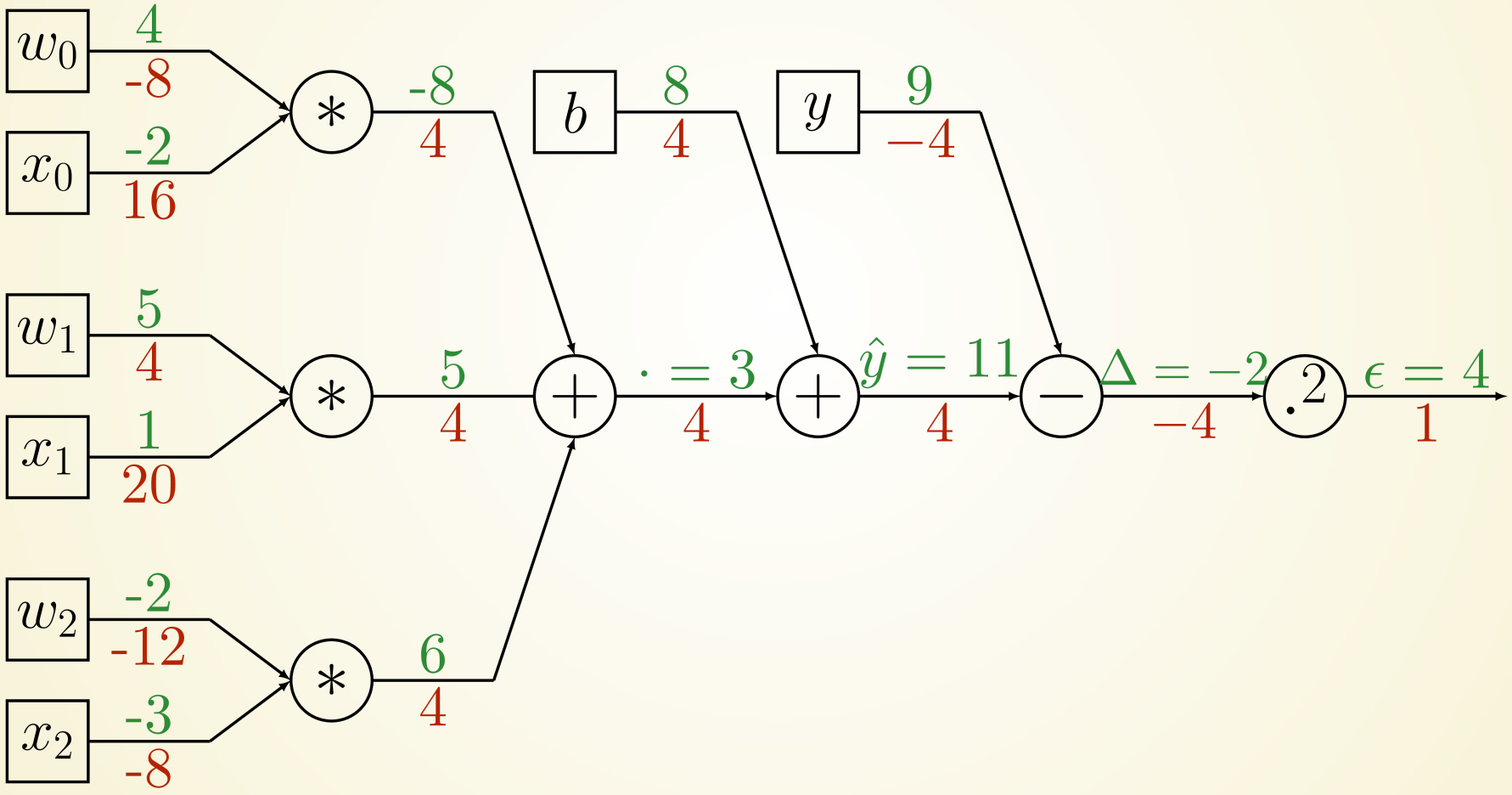
BACKPROPAGATION



BACKPROPAGATION



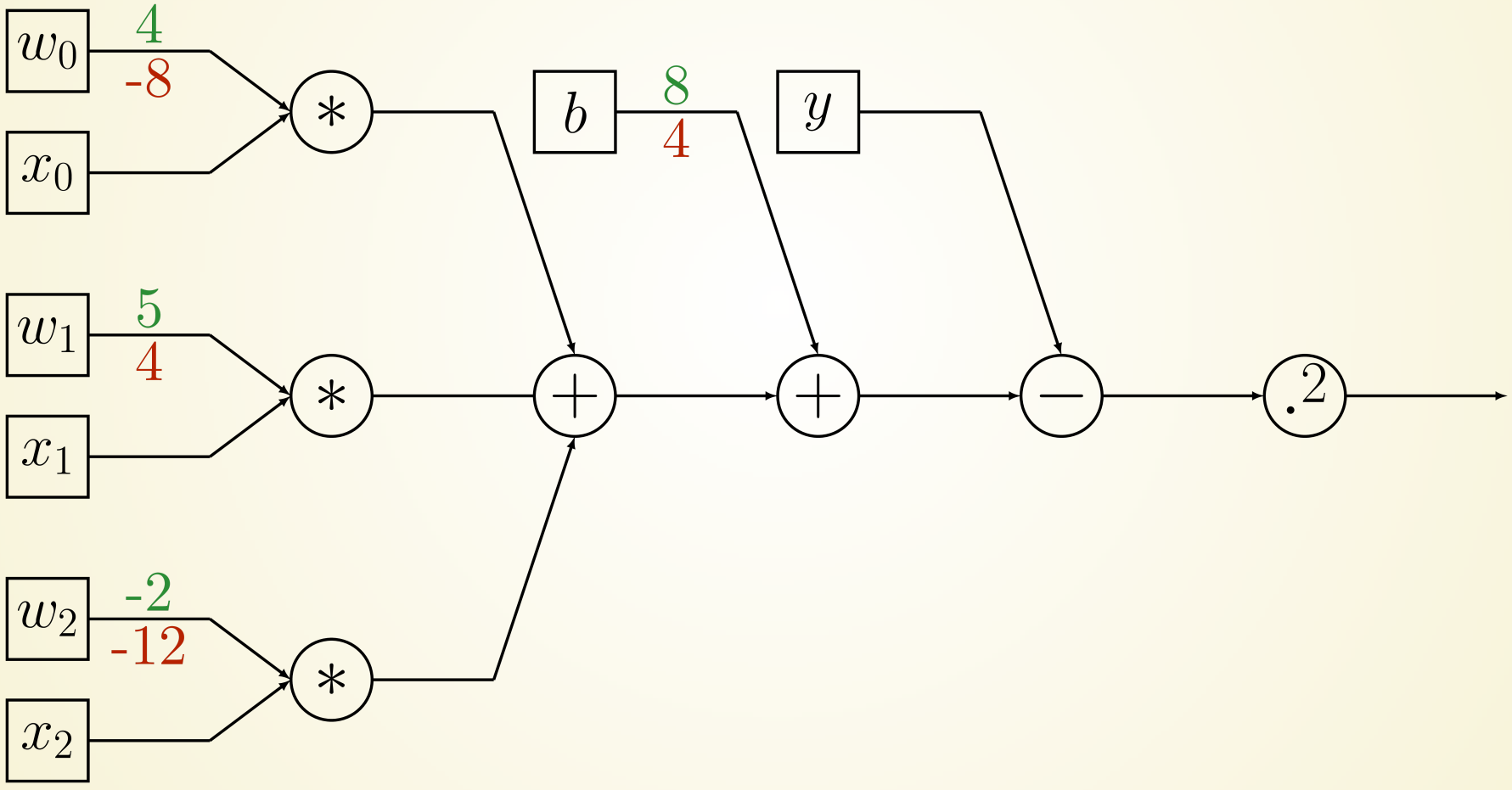
BACKPROPAGATION



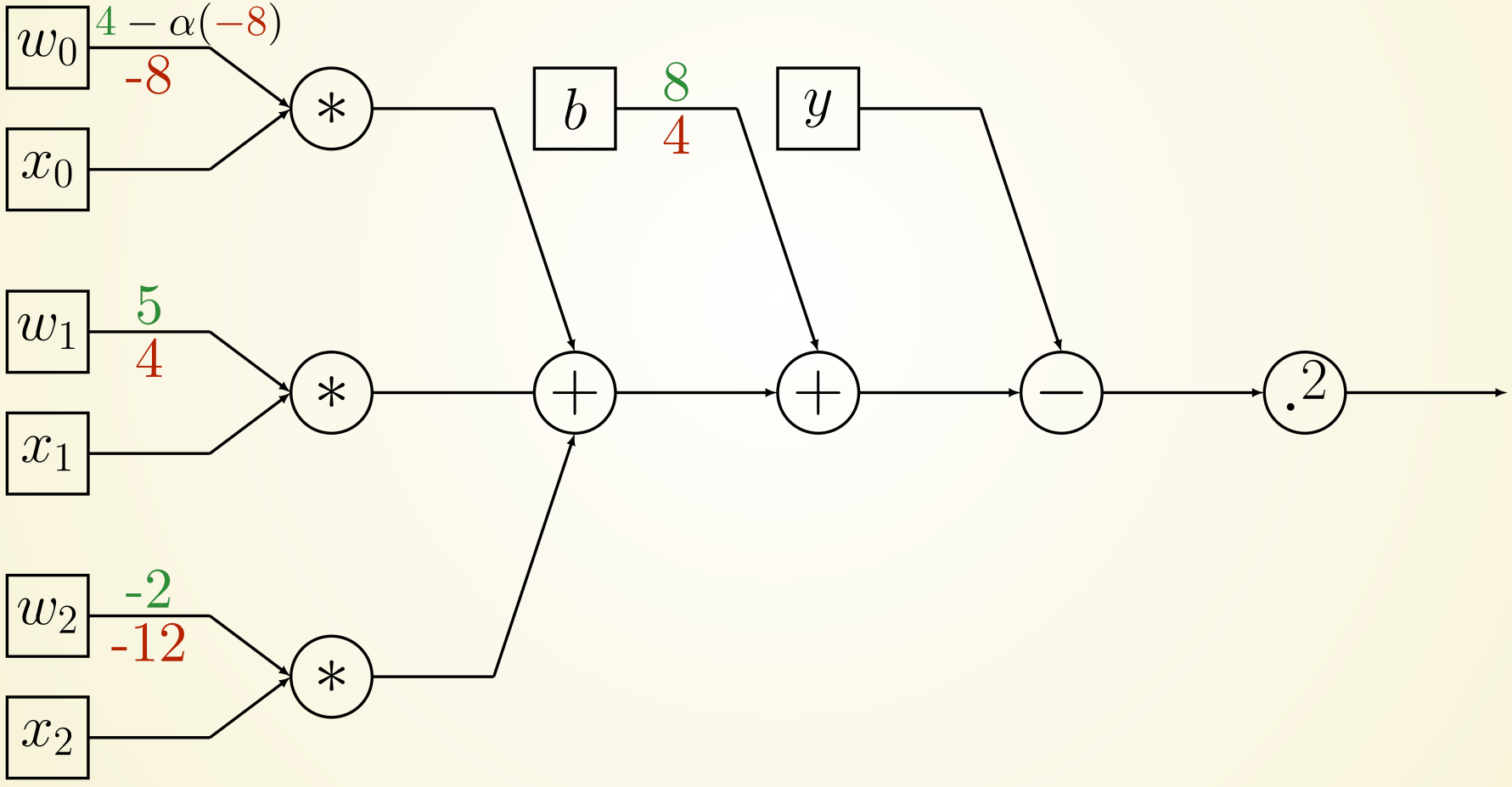
BACKPROPAGATION

```
def training_round(x, y, weights, intercept,
                   alpha=learning_rate):
    # calculate our estimate
    y_hat = model(x, weights, intercept)

    # calculate error
    delta_y = y - y_hat

    # calculate gradients
    gradient_weights = -2 * delta_y * weights
    gradient_intercept = -2 * delta_y

    # update parameters
    weights = weights - alpha * gradient_weights
    intercept = intercept - alpha * gradient_intercept

    return weights, intercept
```

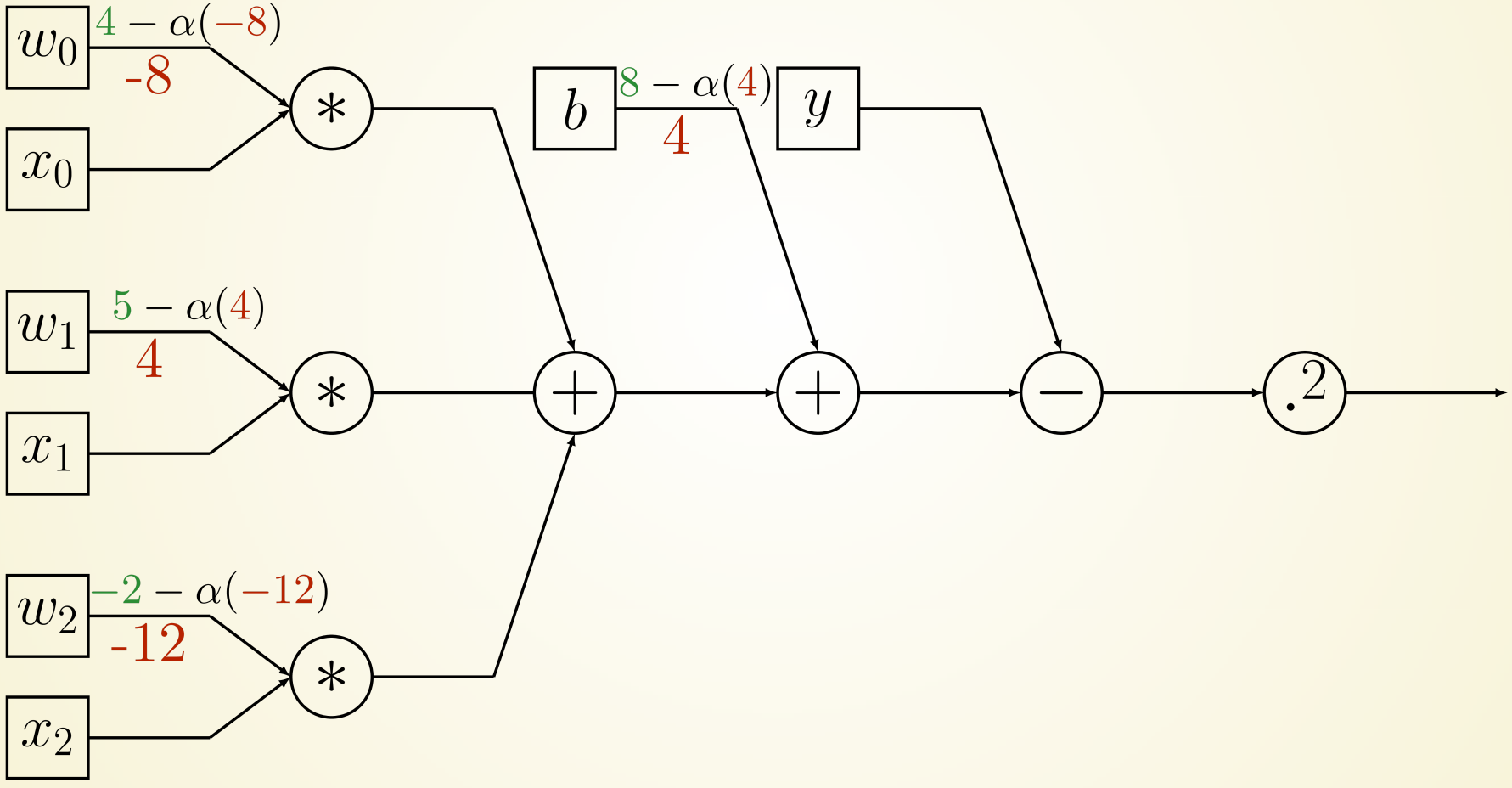
VARIABLE UPDATE



VARIABLE UPDATE



VARIABLE UPDATE



VARIABLE UPDATE

```
def training_round(x, y, weights, intercept,
                   alpha=learning_rate):
    # calculate our estimate
    y_hat = model(x, weights, intercept)

    # calculate error
    delta_y = y - y_hat

    # calculate gradients
    gradient_weights = -2 * delta_y * weights
    gradient_intercept = -2 * delta_y

    # update parameters
    weights = weights - alpha * gradient_weights
    intercept = intercept - alpha * gradient_intercept

    return weights, intercept
```

NUMPY ⟶ TENSORFLOW

```
sess.run(optimizer,
         feed_dict={
             X: X_batch,
             Y: Y_batch
         })
```

TESTING

```
with tf.Session() as sess:
    # train
    # ... (code from above)

    # test
    Y_predicted = sess.run(model,
                           feed_dict = {X: X_test})
    squared_error = tf.reduce_mean(
                        tf.square(Y_test, Y_predicted))

>>> np.sqrt(squared_error)
5967.39
```

LOGISTIC REGRESSION

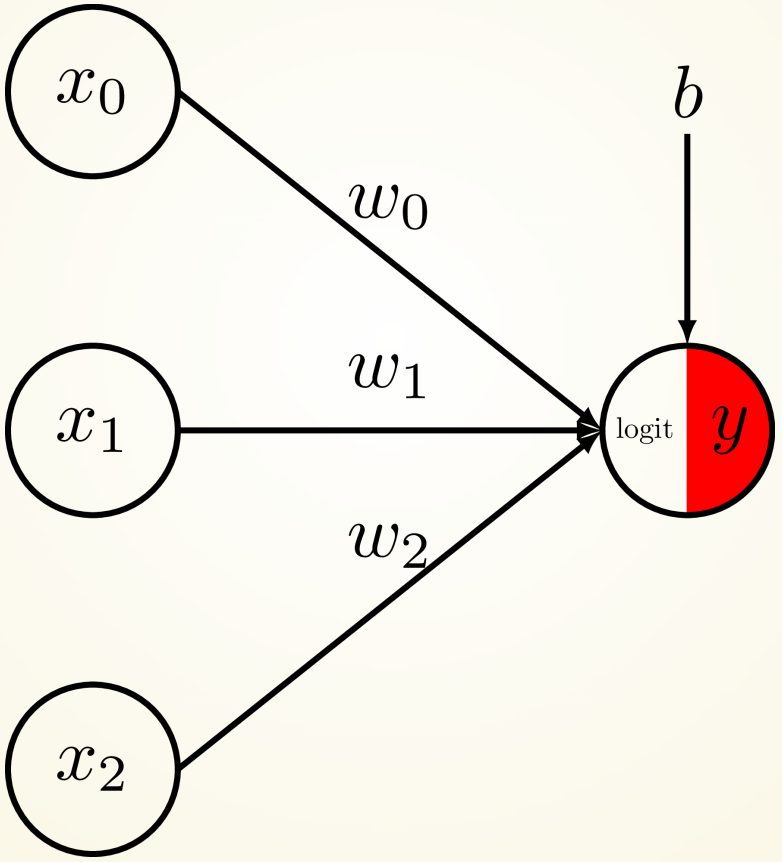
PROBLEM

BINARY CLASSIFICATION

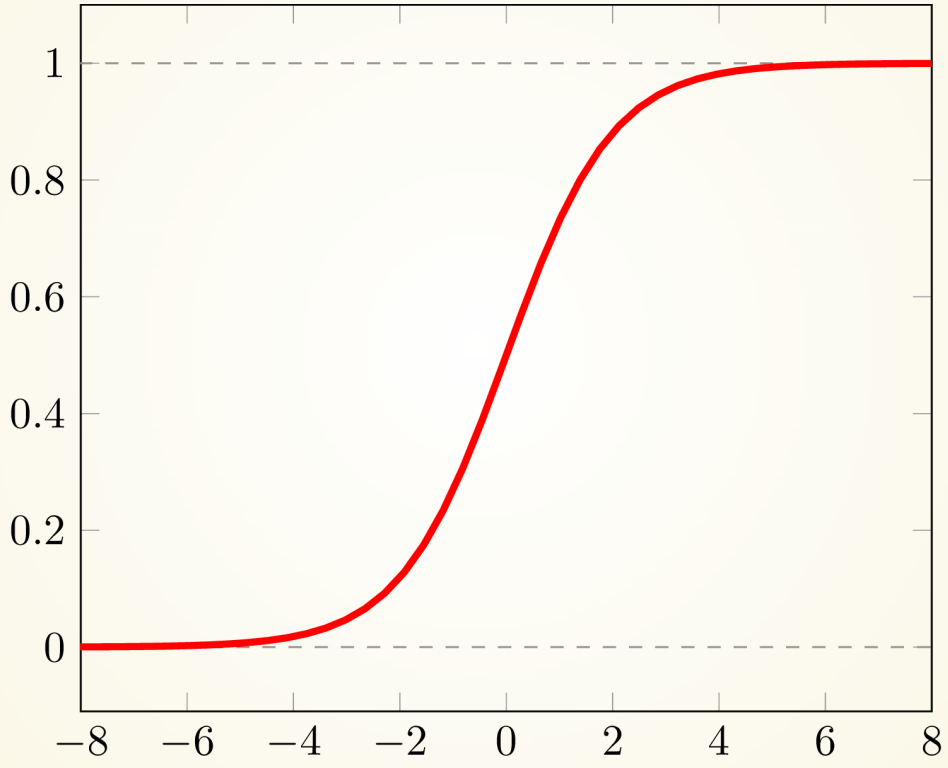
BINARY LOGISTIC REGRESSION - MODEL

Take a **weighted sum** of the features
and add a **bias term** to get the **logit**.
Convert the logit to a **probability**
via the **logistic-sigmoid function**.

BINARY LOGISTIC REGRESSION - MODEL



LOGISTIC-SIGMOID FUNCTION



$$f(x) = \frac{e^x}{1+e^x}$$

CLASSIFICATION WITH LOGISTIC REGRESSION

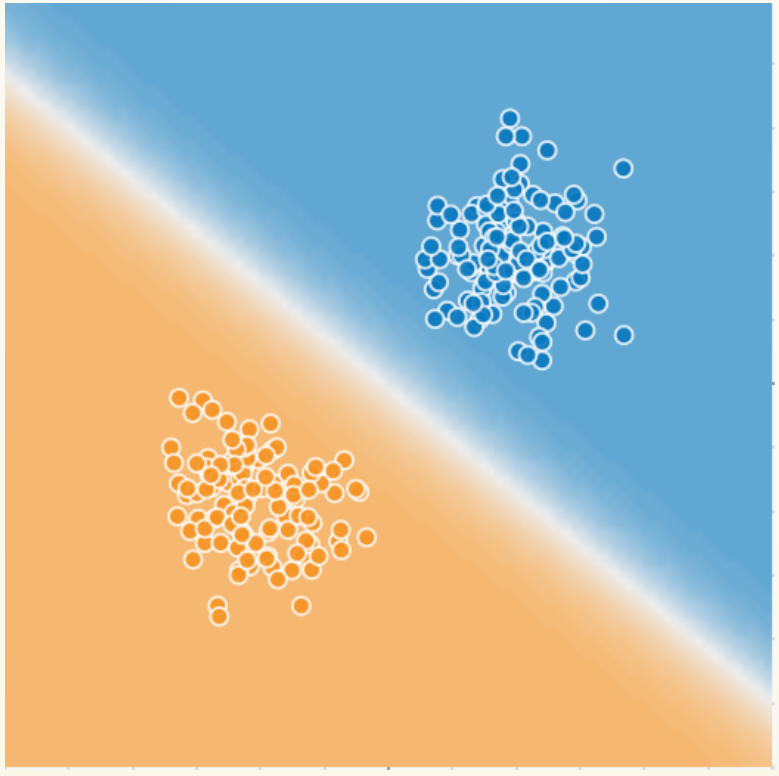
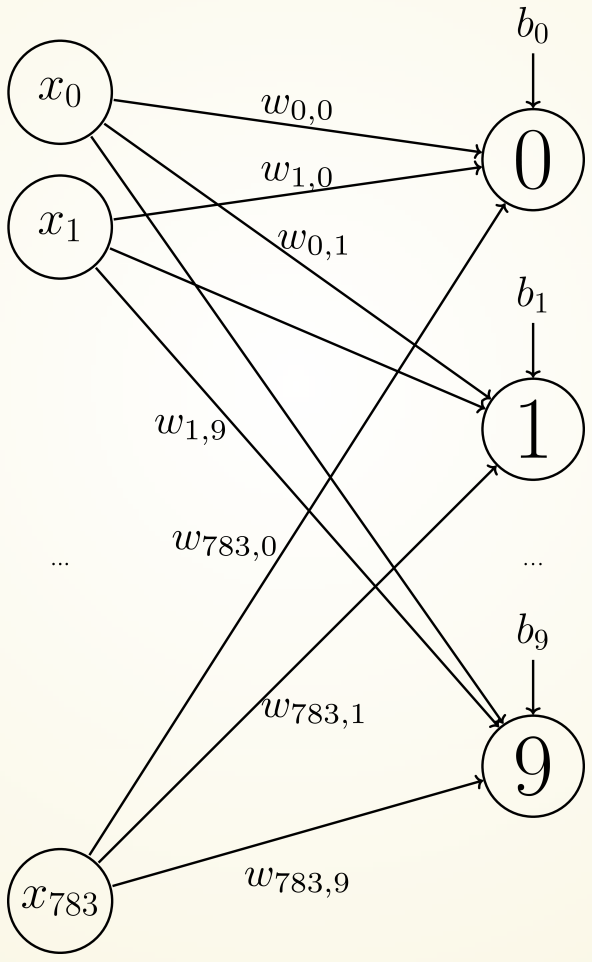


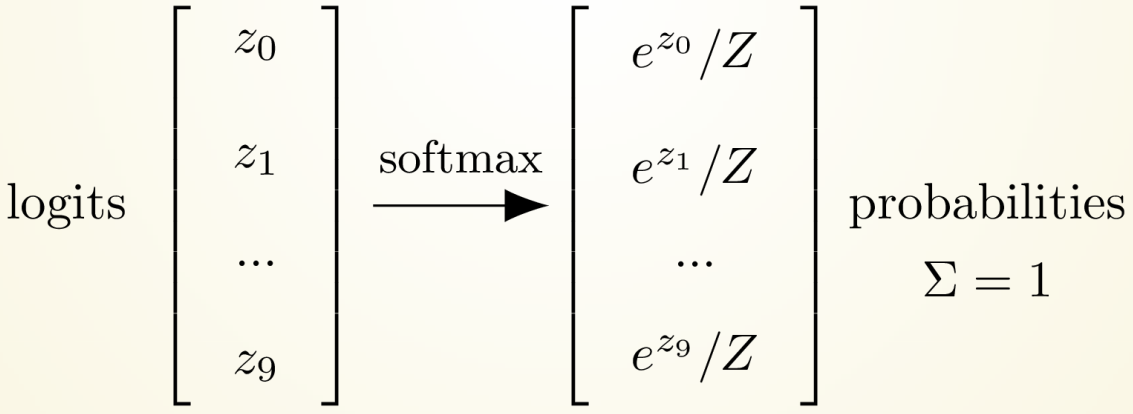
Image generated with playground.tensorflow.org

MODEL

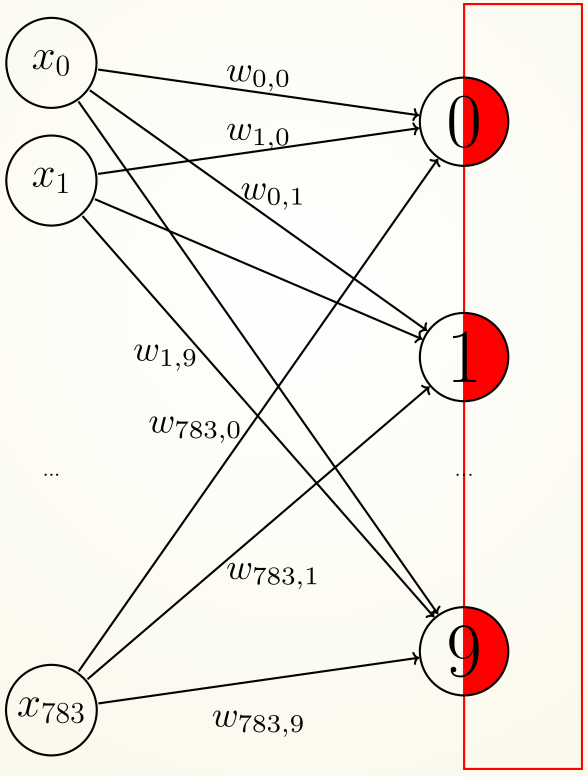


SOFTMAX

```
Z = np.sum(np.exp(logits))
```



MODEL



PLACEHOLDERS

```
# X = vector length 784 (= 28 x 28 pixels)

# Y = one-hot vectors
# digit 0 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

X = tf.placeholder(tf.float32, [None, 28*28])
Y = tf.placeholder(tf.float32, [None, 10])
```

VARIABLES

```
# Parameters/Variables
W = tf.get_variable("weights", [784, 10],
        initializer=tf.random_normal_initializer())
b = tf.get_variable("bias", [10],
        initializer=tf.constant_initializer(0))
```

OPERATIONS

```
Y_logits = tf.matmul(X, W) + b
```

COST FUNCTION

```
cost = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
        logits=Y_logits, labels=Y))
```

COST FUNCTION

Cross Entropy

$$H(\hat{y}) = -\sum_{i} y_i \log(\hat{y}_i)$$

OPTIMIZATION

```
learning_rate = 0.05
optimizer = tf.train.GradientDescentOptimizer
                (learning_rate).minimize(cost)
```

TRAINING

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for i in range(NUM_EPOCHS):
        for (X_batch, Y_batch) in get_minibatches(
                X_train, Y_train, BATCH_SIZE):
            sess.run(optimizer,
                     feed_dict={X: X_batch,
                                Y: Y_batch})
```

TESTING

```
predict = tf.argmax(Y_logits, 1)

with tf.Session() as sess:
    # training code from above

    predictions = sess.run(predict,
                           feed_dict={X: X_test})
    accuracy  = tf.reduce_mean(np.mean(
        np.argmax(Y_test, axis=1) == predictions)

>>> accuracy
0.925
```

DEFICIENCIES OF LINEAR MODELS

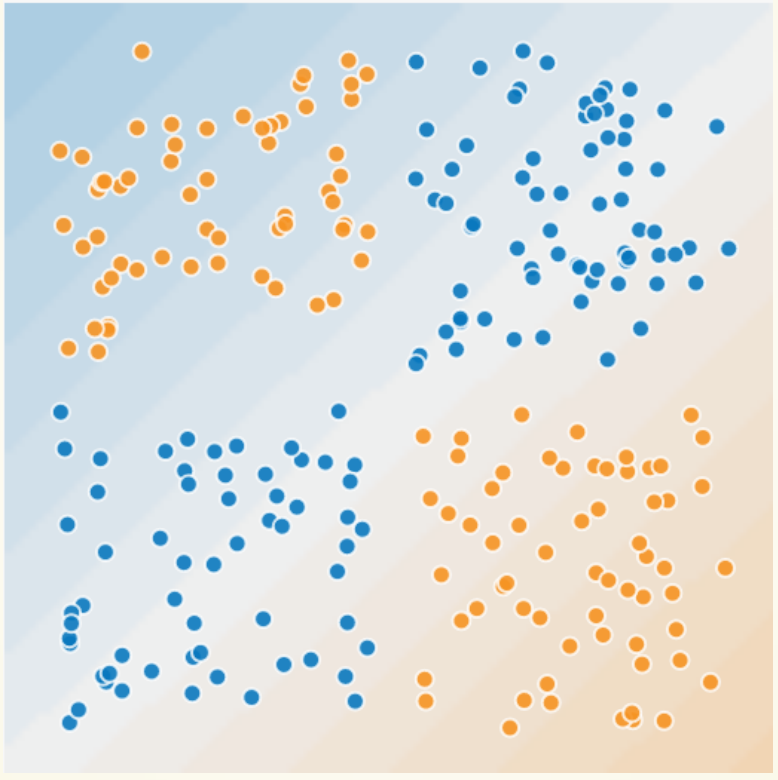


Image generated with playground.tensorflow.org

DEFICIENCIES OF LINEAR MODELS

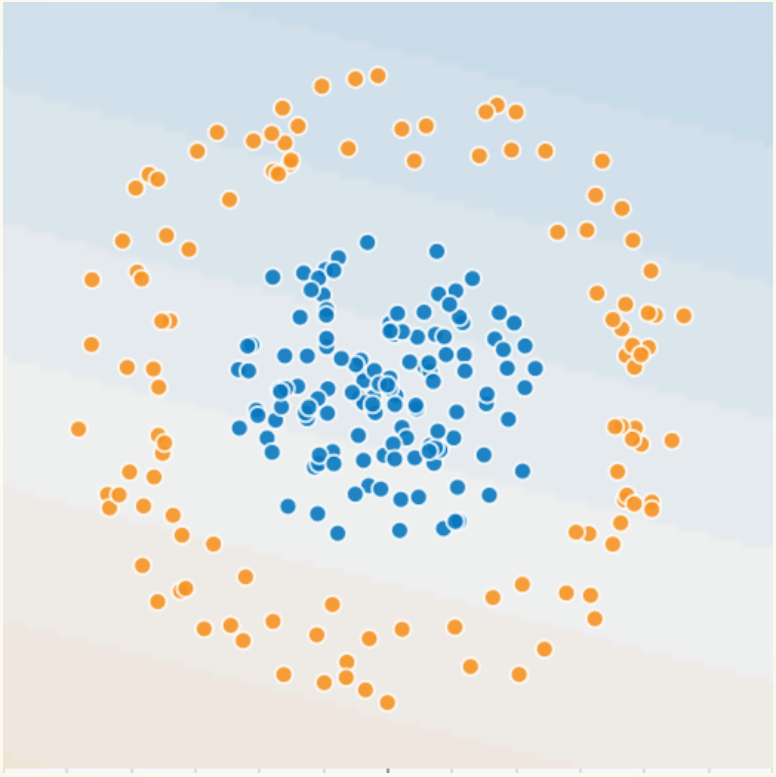
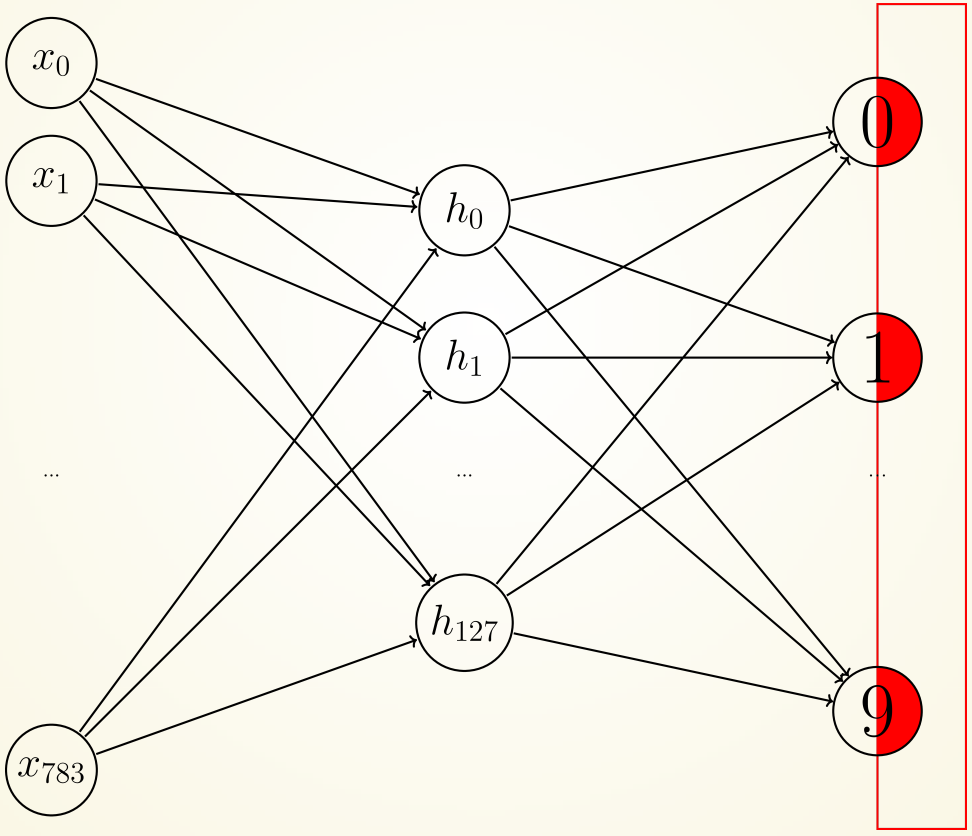


Image generated with playground.tensorflow.org

LET'S GO DEEPER!

ADDING ANOTHER LAYER



ADDING ANOTHER LAYER - VARIABLES

```
HIDDEN_NODES = 128
W1 = tf.get_variable("weights1", [784, HIDDEN_NODES],
       initializer=tf.random_normal_initializer())
b1 = tf.get_variable("bias1", [HIDDEN_NODES],
       initializer=tf.constant_initializer(0))
W2 = tf.get_variable("weights2", [HIDDEN_NODES, 10],
      initializer=tf.random_normal_initializer())
b2 = tf.get_variable("bias2", [10],
      initializer=tf.constant_initializer(0))
```

ADDING ANOTHER LAYER - OPERATIONS

```
hidden   = tf.matmul(X, W1) + b1
y_logits = tf.matmul(hidden, W2) + b2
```

RESULTS

# hidden layers	Train accuracy	Test accuracy
0	93.0	92.5
1	89.2	88.8

IS DEEP LEARNING JUST HYPE?

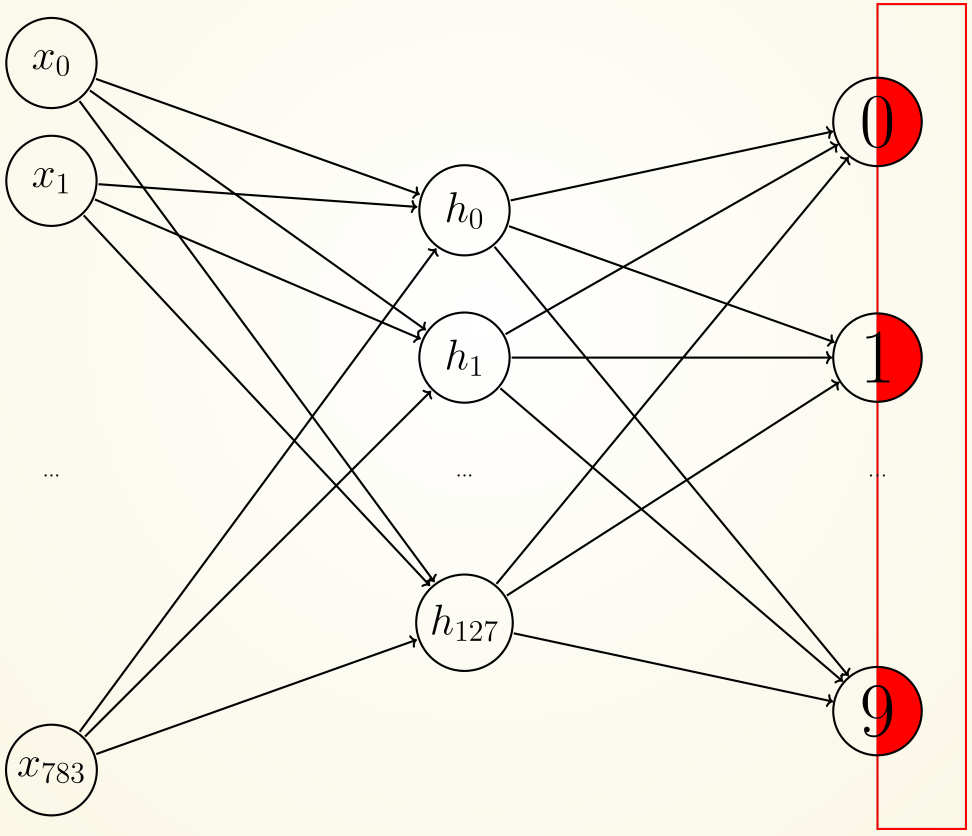
(Well, it's a little bit over-hyped...)

PROBLEM

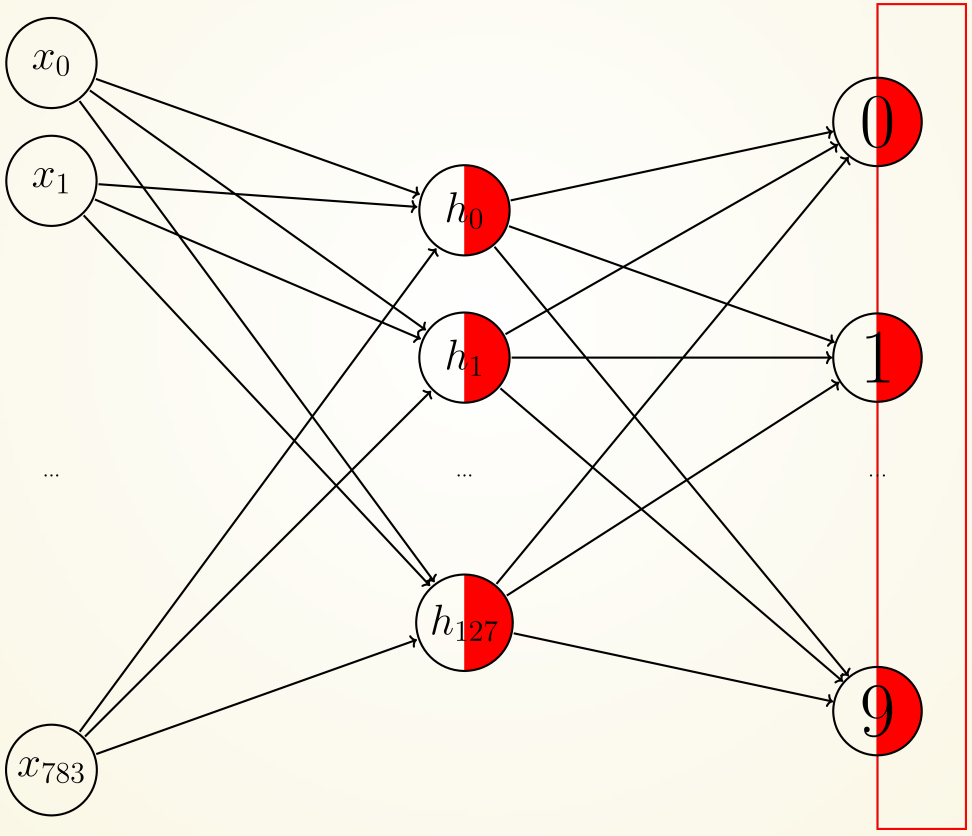
A linear transformation of a linear transformation is **still** a linear transformation!

We need to add **non-linearity** to the system.

ADDING NON-LINEARITY

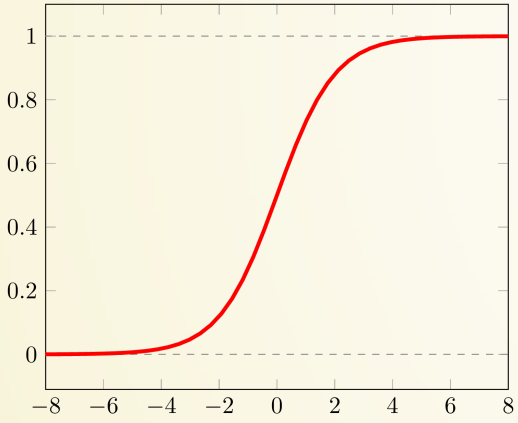


ADDING NON-LINEARITY

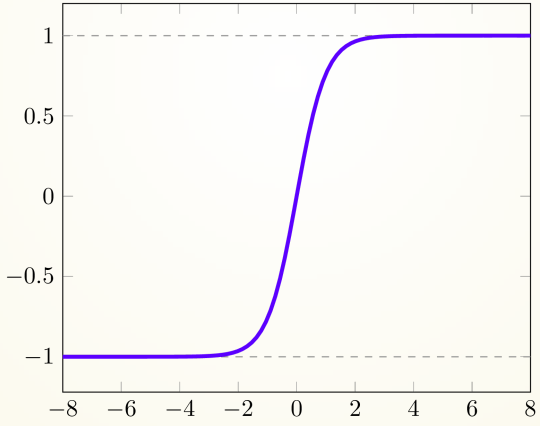


NON-LINEAR ACTIVATION FUNCTIONS

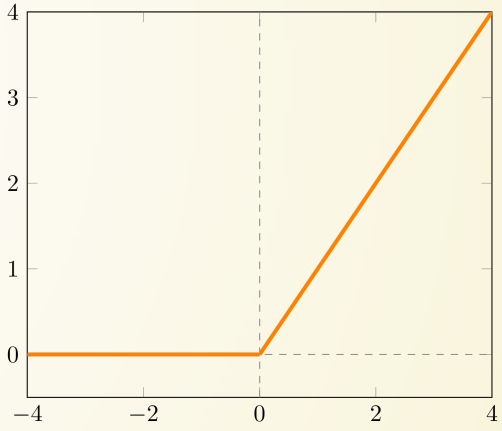
Logistic sigmoid



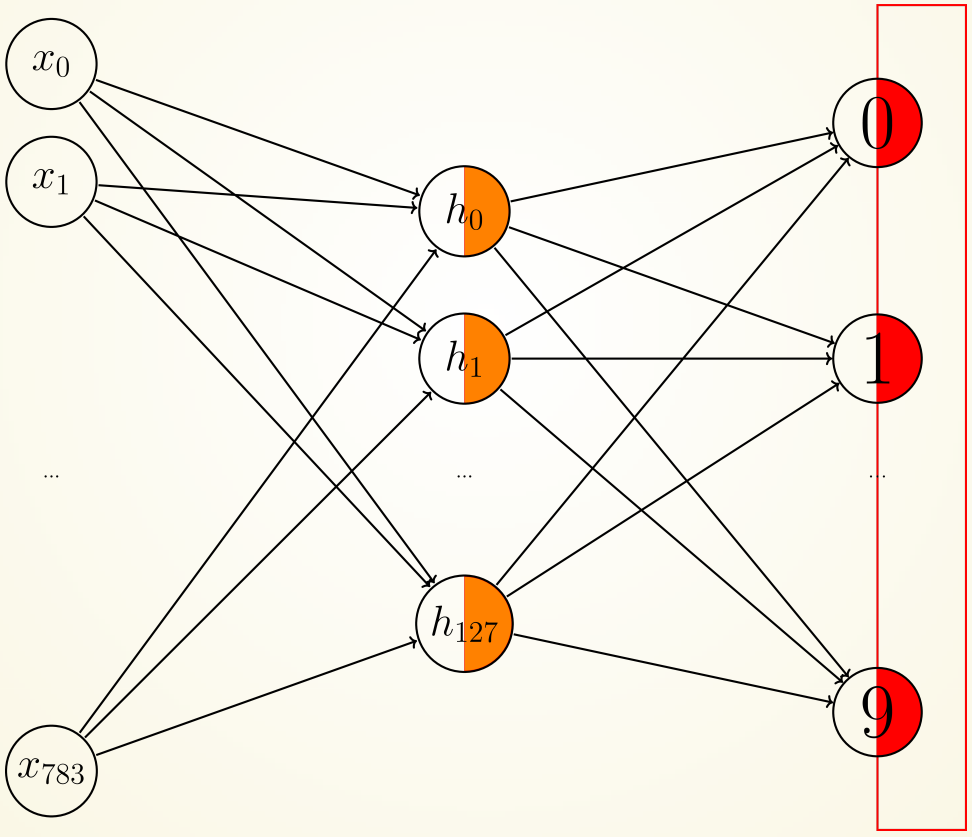
Hyperbolic tangent (tanh)



Rectified linear (ReLU)



ADDING NON-LINEARITY



OPERATIONS

```
hidden   = tf.nn.relu(tf.matmul(X, W1) + b1)
y_logits = tf.matmul(hidden, W2) + b2
```

RESULTS

# hidden layers	Train accuracy	Test accuracy
0	93.0	92.5
1	97.9	95.2

WHAT THE HIDDEN LAYER BOUGHT US

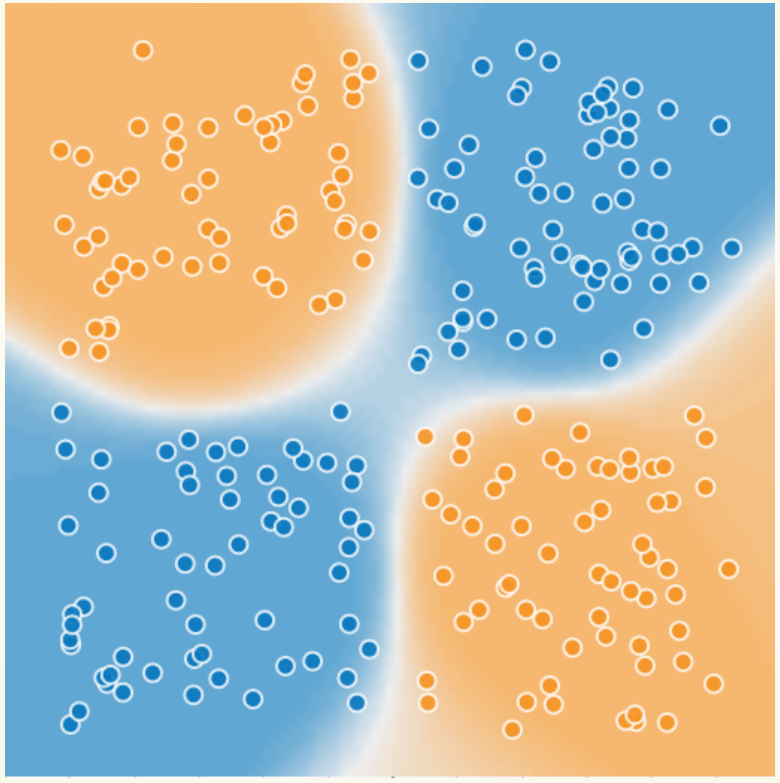


Image generated with playground.tensorflow.org

WHAT THE HIDDEN LAYER BOUGHT US

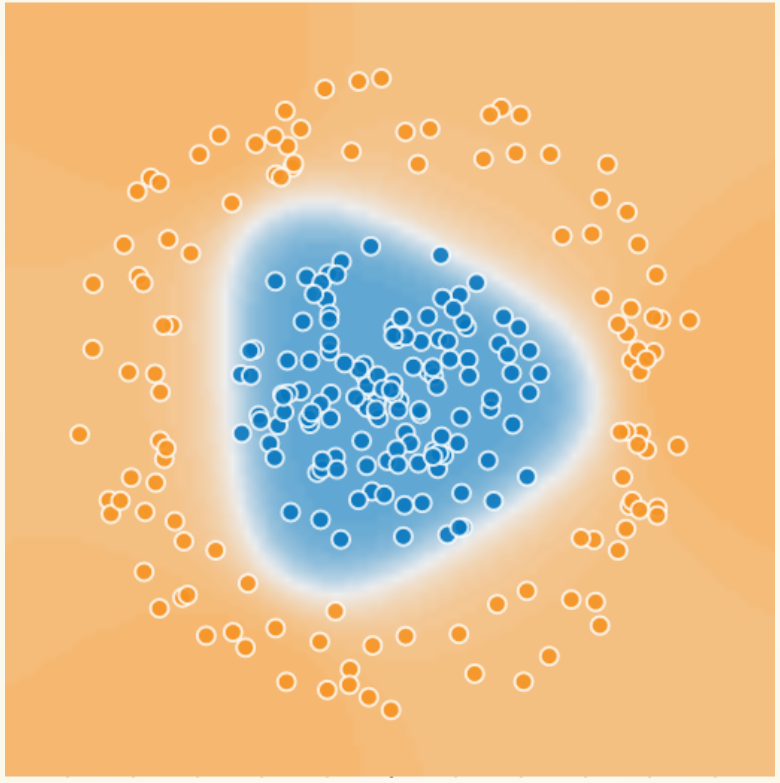
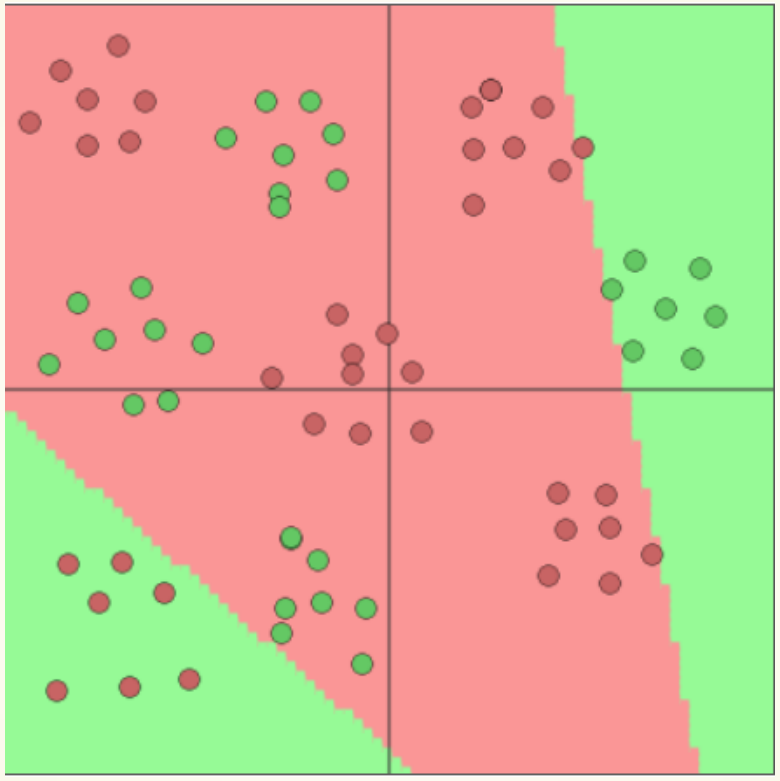


Image generated with playground.tensorflow.org

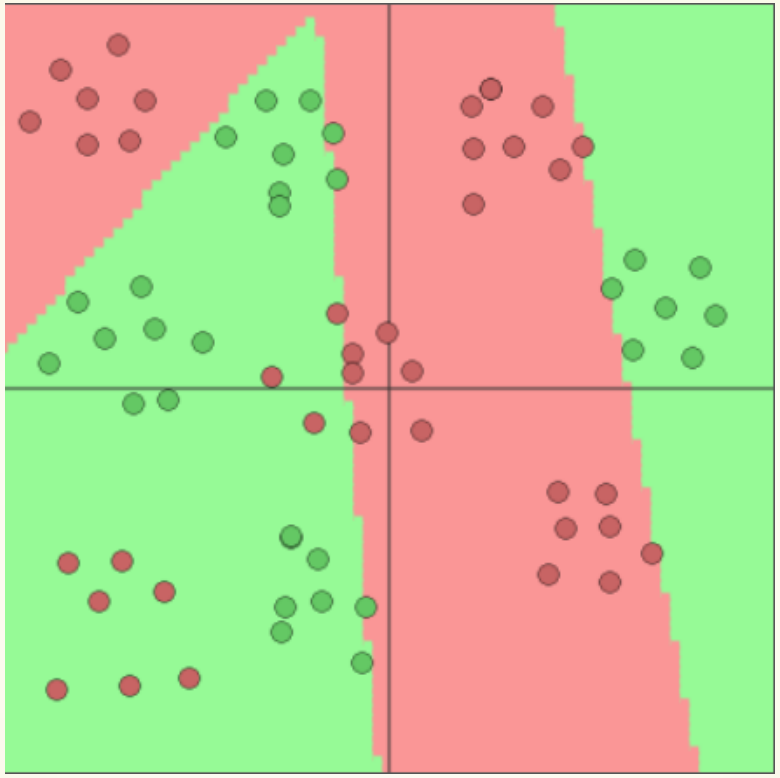
ADDING HIDDEN NEURONS



2 hidden neurons

Image generated with ConvNetJS by Andrej Karpathy

ADDING HIDDEN NEURONS



3 hidden neurons

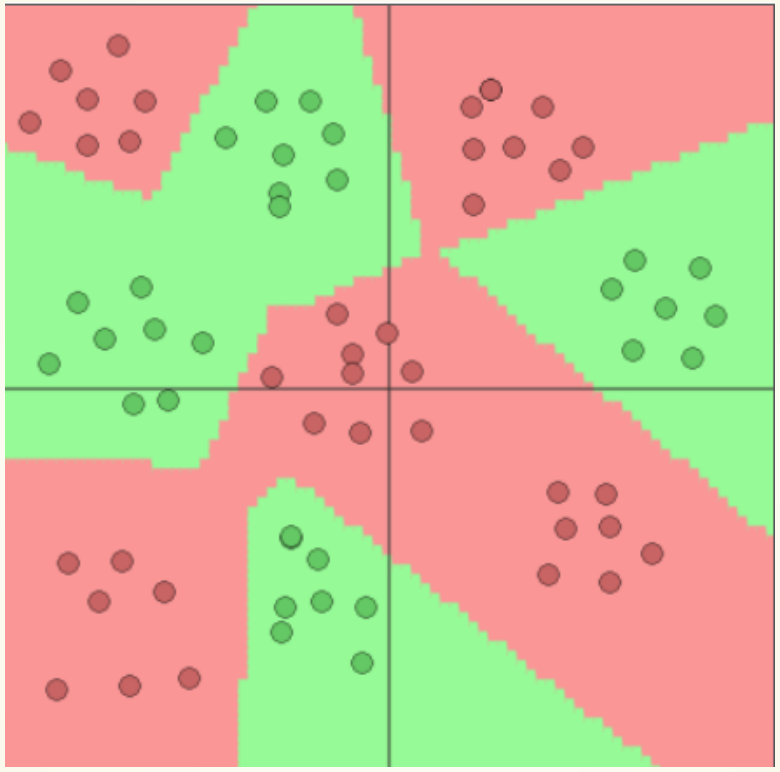
Image generated with ConvNetJS by Andrej Karpathy

ADDING HIDDEN NEURONS

4 hidden neurons

Image generated with ConvNetJS by Andrej Karpathy

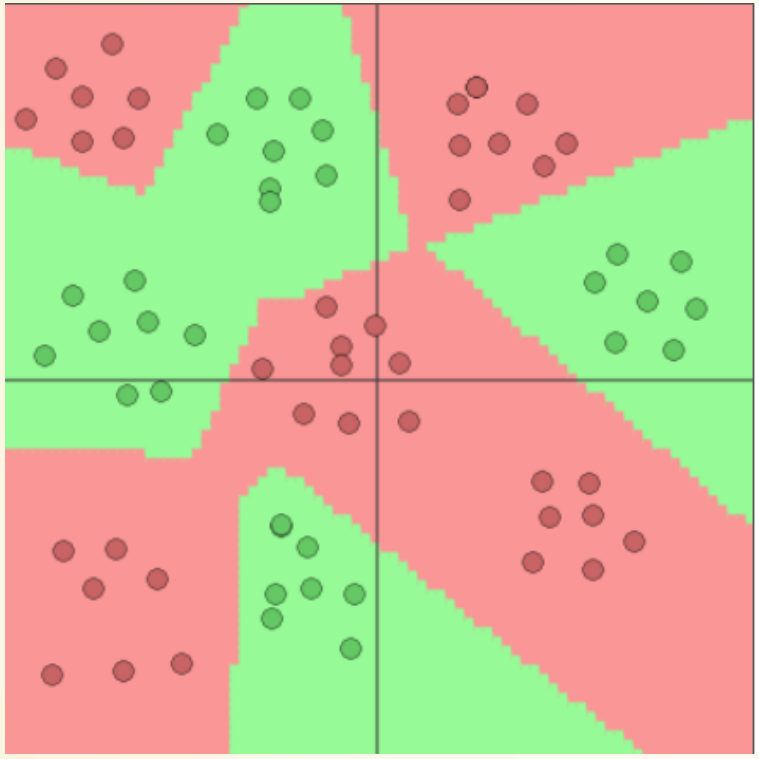
ADDING HIDDEN NEURONS



5 hidden neurons

Image generated with ConvNetJS by Andrej Karpathy

ADDING HIDDEN NEURONS



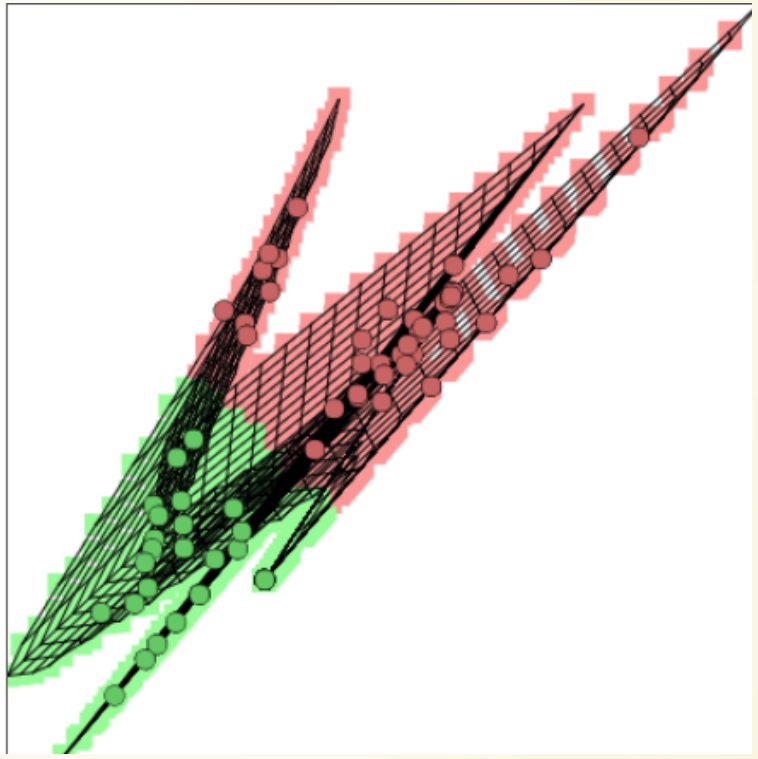
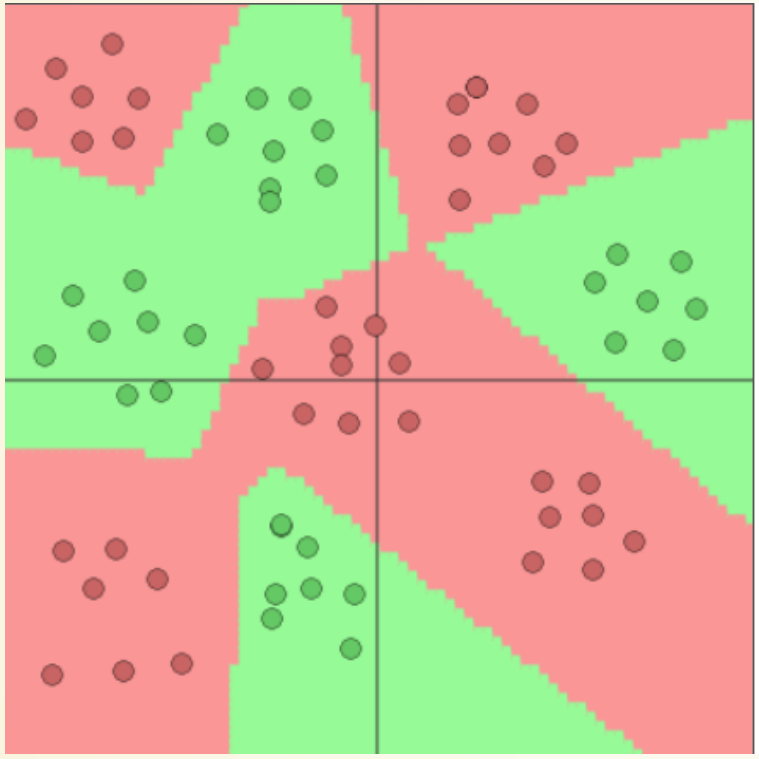


Image generated with ConvNetJS by Andrej Karpathy

ADDING HIDDEN NEURONS



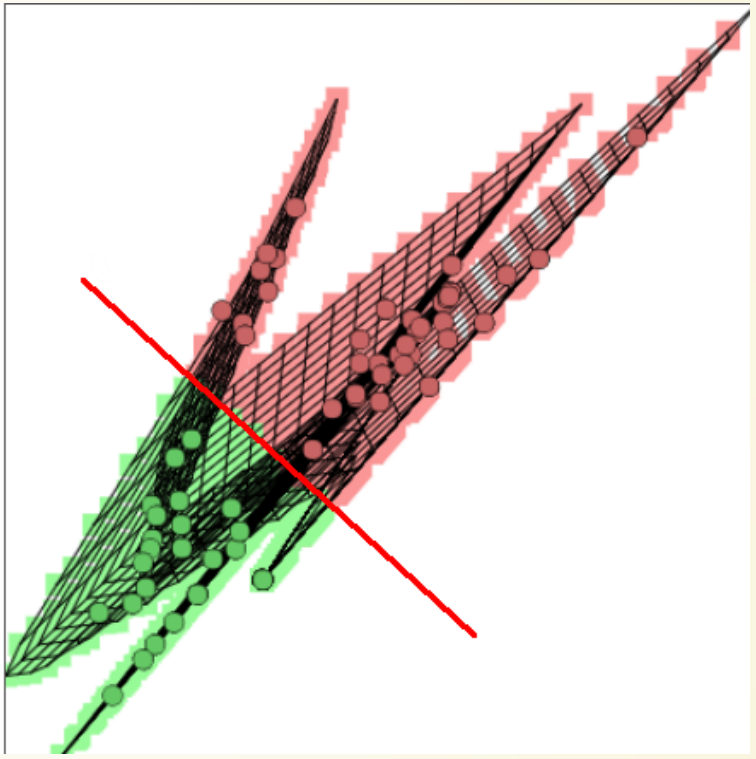


Image generated with ConvNetJS by Andrej Karpathy

UNIVERSAL APPROXIMATION THEOREM

A **feedforward network** with a **single hidden layer** containing a finite number of neurons **can approximate** (basically) **any interesting function**

ARE WE DEEP LEARNING YET?

No!

OPERATIONS

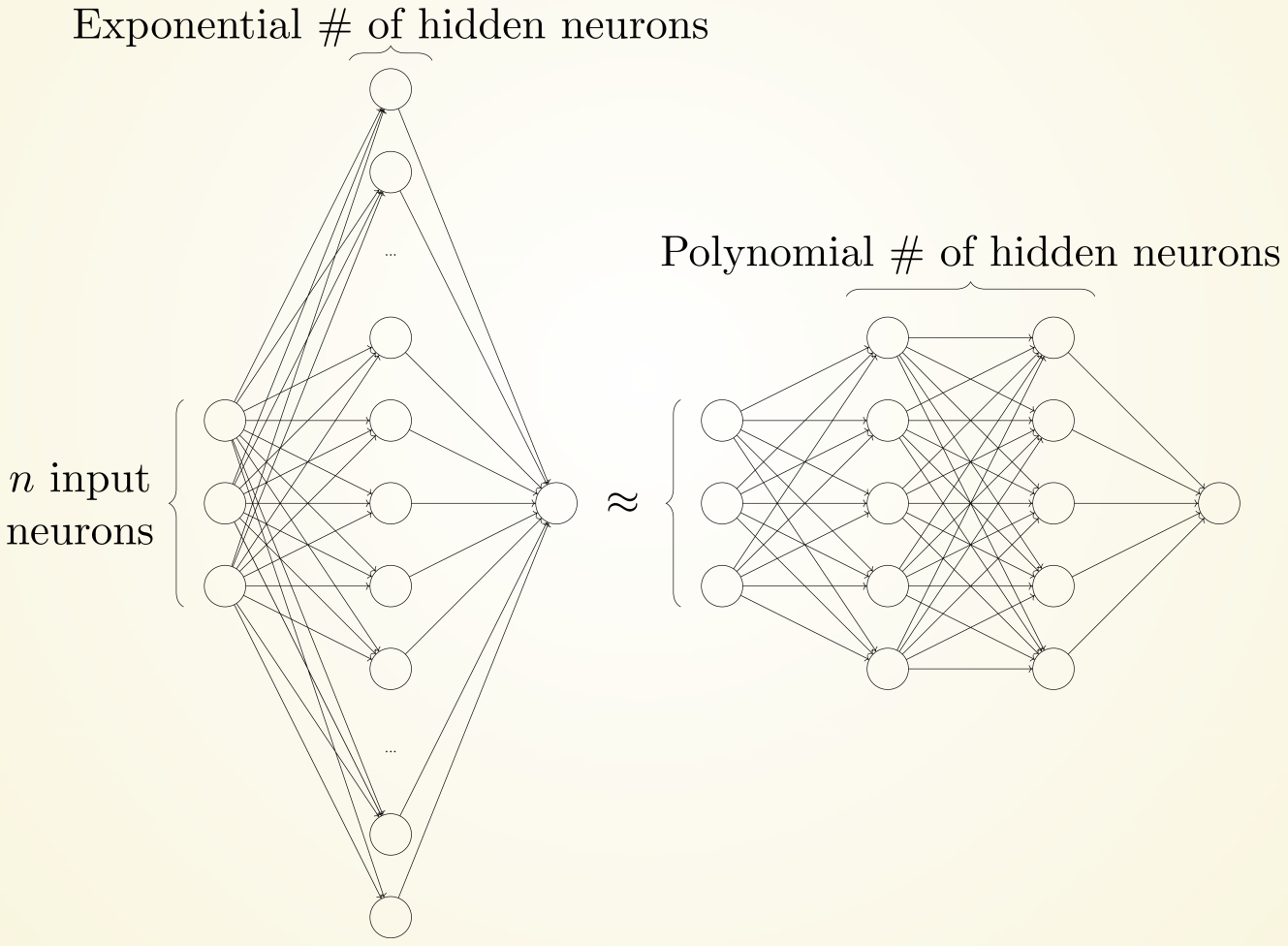
```
hidden_1 = tf.nn.relu(tf.matmul(X, W1) + b1)
hidden_2 = tf.nn.relu(tf.matmul(hidden_1, W2) + b2)
y_logits = tf.matmul(hidden_2, W3) + b3
```

WHY GO DEEP?

3 reasons:

- Deeper networks are **more powerful**

MORE POWERFUL

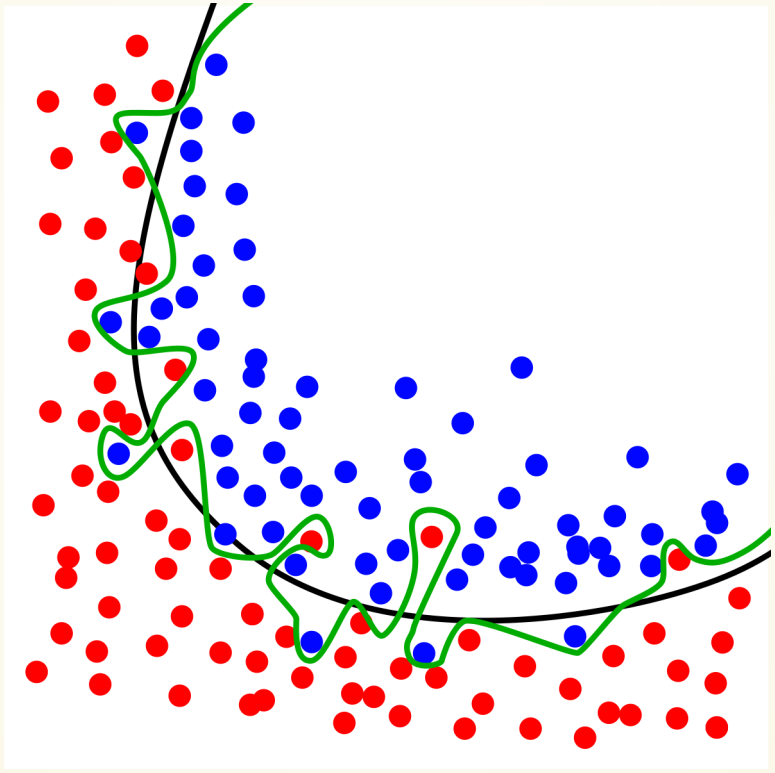


WHY GO DEEP?

3 reasons:

- Deeper networks are **more powerful**
- Narrower networks are **less prone to overfitting**

OVERFITTING



LESS PRONE TO OVERFITTING