Lecture 02

Gradient Descent, Back Propagation

CSCI E-89 Deep Learning

Fall 2024

Zoran B. Djordjević

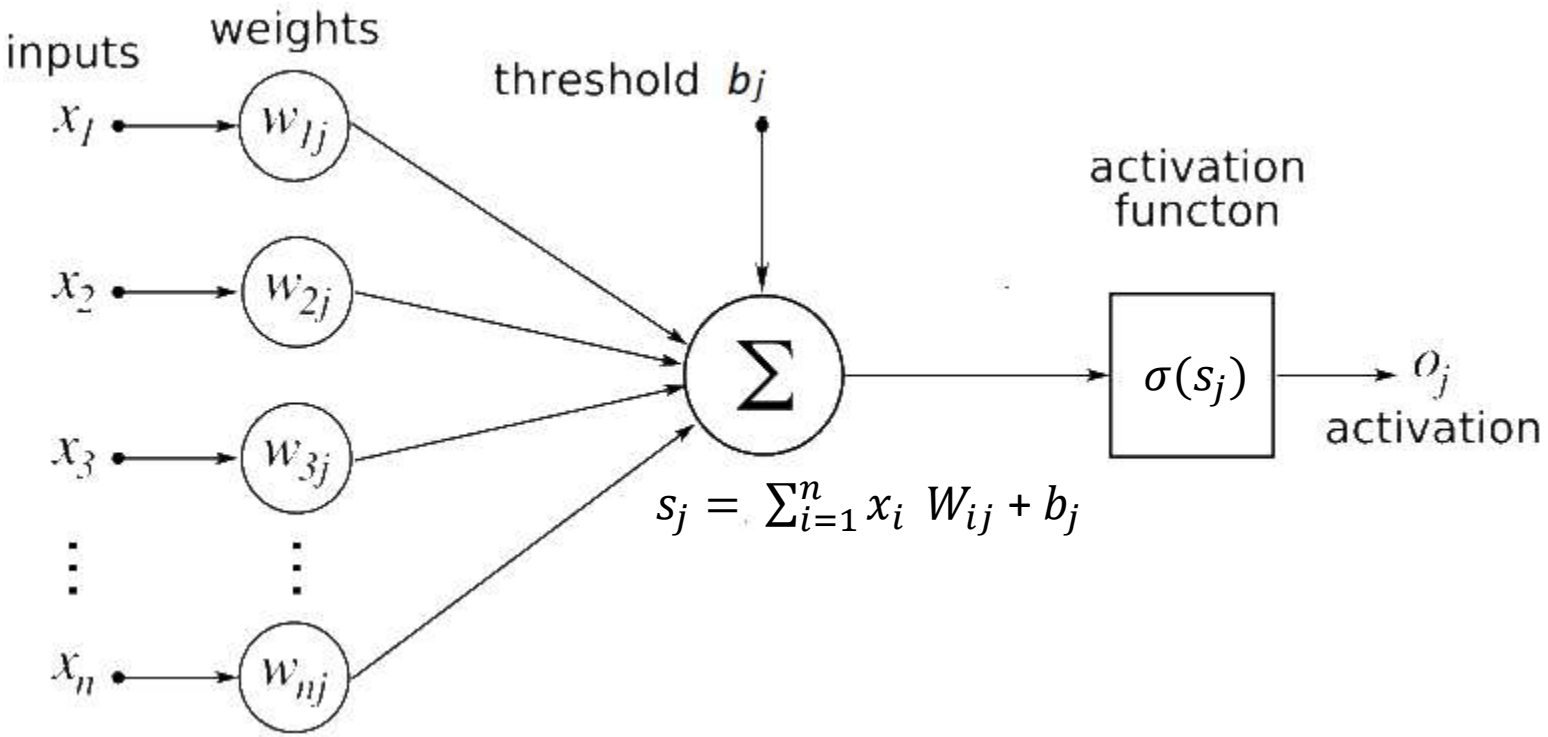
References

- This lecture relies on materials in
- Chapters 2 and 3 of *Deep Learning in Pyth*on, 2nd Edition by Francois Chollet, and
- Chapter 10, *Hands-on-machine learning* …, 3rd Edition by Aurelien Geron, or
- Chapter 12, *Hands-on-machine learning* …, 2nd Edition by Aurelien Geron

Review of the Optimization Problem and Gradient Descent

Neurons, from Last Lecture

- We said that we will build neural networks using neurons of a single simple type.
- The neuron with index j in a layer could have many inputs $\{x_i\}$. Every input x_i is multiplied by a weight W_{ij}. The sum of such products is incremented by a bias (threshold) b_j. The result of that summation, s_j, is fed into an activation function $\sigma(s_j)$. The activation function is non-linear, e.g., $sigmoid\ \sigma()$, $\tanh()$, $ReLU()$, or other.



Optimization Problem, from last lecture

- Initially, we do not know proper (optimal) values of parameters ($W_{ij}^{(k)} and\ b_j^{(k)}$) of any neuron in any layer. (k) is the index of a neuron layer. We have a training set of N experiments $\{X^i\}\ and$ a set of proper (correct) labels $\{Y^i\}$ for all experiments.
- Forward calculation for any one experiment X^i produced an output vector $Y^i_{predicted}$. The value of that vector typically does not match the known label Y^i.
- The sum over all N samples of the difference between known values Y^i and the forward calculated values $Y^i_{predicted}$ is a measure of the quality of the network.
- Usually, we express that quality as the ***Cost*** or ***Loss Function*** of the form:

$$L_2(\{W\},\{b\}) = \sum_{i=1}^{N}(Y^i - Y^i_{predicted})^2$$

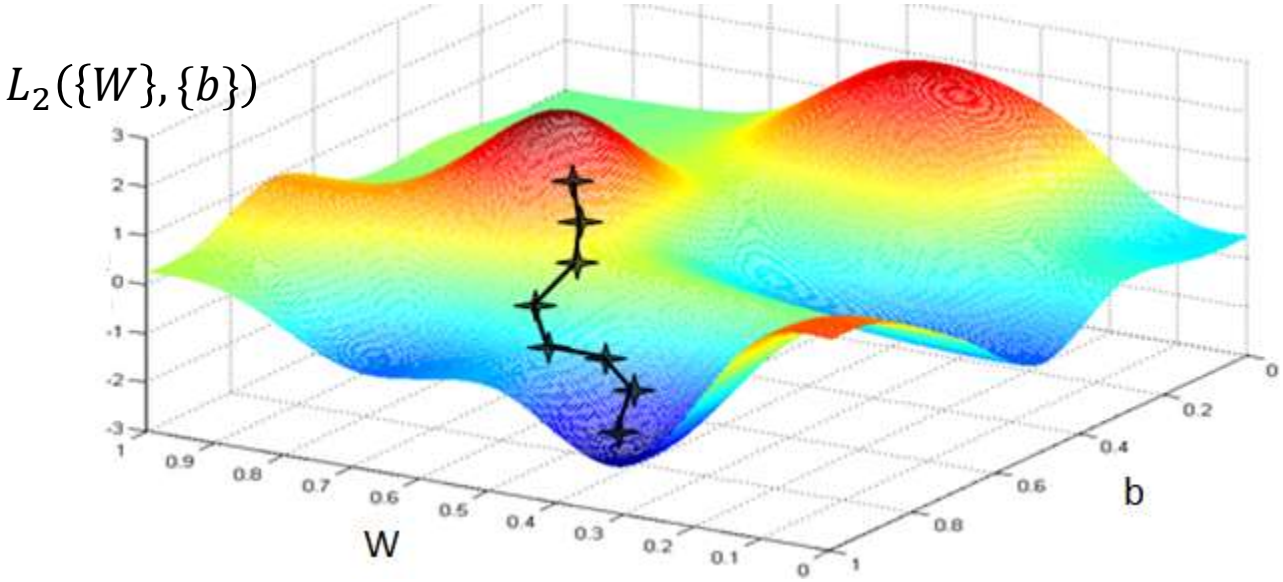
- which is the sum of squares of the errors the model makes at every experiment. Notice that the loss function depends on values of all weights and biases of all neurons in all hidden layers.
- Finding the best collection of weights (and biases) which minimizes the loss function is an optimization problem.
- The input vectors could have very large dimensionality, 10s, 100s, 1000s, even more. We could have very large number of weights just in the first layer. Since we could have many hidden layers, the number of weights to find could be truly large.
- We are dealing with an optimization in a space of a (very) large dimensionality.

Gradient Descent, from last lecture

- Input variables for the optimization, i.e., minimization of the Loss function L_2 are the model weights $\{W\}$ and biases $\{b\}$. The dataset features (inputs) $\{X^i\}$ are fixed and are not variables in the optimization process.
- We search for minimum of L_2 by starting from a random point in space {W,b} and by moving away from that point in the direction opposite to the gradient

$$\nabla L_2 = \left(\frac{\partial L_2}{\partial W}, \frac{\partial L_2}{\partial b}\right), \qquad W_{step\ i+1} = W_{step\ i} - \lambda \nabla L_2(W_{step\ i})$$

- The gradient indicates the direction of maximum growth for the loss function. To find the minimum of L_2 we move in the direction opposite to the gradient.
- That is where the name "gradient descent" comes from.



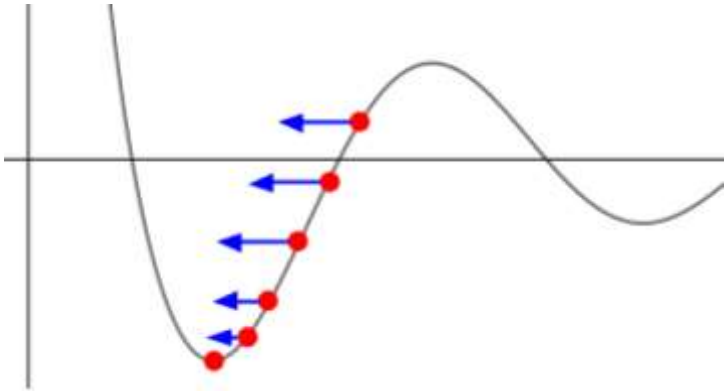
Gradient Descent and Backpropagation

- The objective of the gradient descent is finding the point where the loss (cost) function achieves its minimum value.
- "Learning" is a process of improving the model parameters to minimize the loss function through several iterations, i.e., training steps.
- Model learns from the input data through by searching for the minimum of the loss function using the process of gradient decent.
- Gradient is a vector, generally represented with the ∇ symbol (Greek letter nabla). It is analogous to a derivative but used with functions with multi-dimensional inputs. Gradient is a single vector containing all the partial derivatives of a function of many variables.
- The elements of the gradient are partial derivatives, one derivative per every independent argument of the function. We minimize loss function with respect to weights and biases, W-s and b-s, not the coordinates of samples $\{X_i{}^k\}$.
- The weights and biases are the independent coordinates with respect to which we calculate partial derivatives:

$$\nabla \equiv \left(\frac{\partial}{\partial W_{11}^1}, \frac{\partial}{\partial W_{12}^1}, \ldots, \frac{\partial}{\partial b_1}, \frac{\partial}{\partial W_{11}^2}, \ldots, \frac{\partial}{\partial W_{nn}^N}, \frac{\partial}{\partial b_N}\right)$$

- Each partial derivatives measures the rate of change of the function with respect to one particular input variable.
- Each partial derivative with respect to a variable (weight or bias) tells us how much the function will increase if we increase that particular variable and keep other variables fixed.

Gradient Descent and Learning Rate λ



- We define gradient descent algorithm as:

$$W_{step\ i+1} = W_{step\ i} - \lambda \nabla L_2(W_{step\ i})$$

- Scalar λ is called ***the learning rate***. λ determines how much new value moves away from the old in the direction opposite to the gradient.
- The learning rate is not a value that model determines. λ is a *hyperparameter*, or a manually configurable parameter of the model.
- We need to determine the right value for λ . If λ is too small, it will take many learning cycles to find the minimum of the loss function. If λ is too large, the algorithm may simply "skip over" the minimum and never find it, jumping cyclically. That's known as overshooting.
- The learning rate does not have to be fixed. We could adjust its value during the leaning process. Several optimization algorithms do that automatically.

How to Implement the Gradient Descent

- To reiterate, the loss function quantifies the quality of any particular set of weights $\{W, b\}$. The goal of optimization is to find W-s and b-s that minimize the loss function.

The first very bad idea : Random search

- Since it is so simple to check how good a given set of parameters W is, the first (very bad) idea that may come to mind is to simply try out many different random weights and keep track of what works the best.

- In a space with one, two or a small number of dimensions, this strategy might work. In Deep Learning, the dimension of $\{W, b\}$ space is usually huge. There is no way to fill in that space with random points in an efficient manner.

Strategy #2: Random Local Search

- Another strategy could be to try to extend one foot in a random direction and then take a step only if it leads downhill. Concretely, we will start out with a random value of {W, b}, generate a random perturbation to that value and if the loss at the perturbed point is lower, we will perform an update. This strategy is not much more efficient that the one above.

- You have noticed that evaluating the numerical gradient has complexity proportional the number of parameters. If we have 30,730 parameters, we need to perform on the order of 2 X 30,731 evaluations of the loss function to evaluate the gradient and to perform only a single parameter update.
- Modern Neural Networks can easily have tens of millions of parameters. Clearly, simply finding gradients numerically is not scalable and we need a better approach.

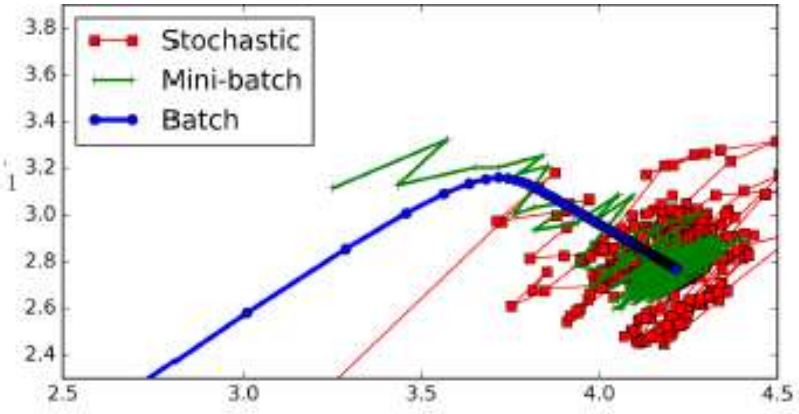
Stochastic Gradient Descent

- Our loss function is a sum over the entire batch of N samples. When N is small, and the dimension of $\{W, b\}$ space is not huge, we can perform that sum. Such calculation is referred to as the *Batch Gradient Descent*.
- Typically, both N and the dimension of the weight space are very large. Performing such sum at every point along the gradient descent curve would be prohibitive.
- We could perform gradient descent by taking a single sample X^i and calculating the gradient with of the single term $(Y^i - Y^i_{predicted})^2$ with respect to parameters $\{W, b\}$. This is obviously an approximation but is a much faster procedure. To account for the influence of different samples, after every step we could choose a different sample in a random (stochastic) way. Such process is named *Stochastic Gradient Descent.* As we are changing samples, gradients typically vary wildly.

- A somewhat better approach is to take a moderately small number of samples: 64, 128 or some other power of 2 and approximate the loss function with

$$L_2(\{W\},\{b\}) \approx \sum_{i}^{mini\_batch} (Y^i - Y^i_{predicted})^2$$

- Summation of errors over a mini-batch smooths the gradient and also speeds up the calculation since our hardware efficiently parallelizes such calculations.



The axes of the above plot are two weights W.

Stochastic Gradient Descent

- Stochastic Gradient Descent (SGD) implemented with one sample at a time is rarely used. Due to vectorized code optimizations in our GPU and multi-CPU machines, it is computationally much more efficient to evaluate the gradient for 128 examples in one go, rather than the gradient for 128 individual samples 128 times over.
- SGD at one point referred to using a single example at a time to evaluate the gradient. Today, when we use the term SGD, we typically refer to “mini-batch gradient descent”.
- Mentions of MGD for "Minibatch Gradient Descent", or BGD for "Batch gradient descent" are rare.
- The size of the mini-batch is a hyperparameter, but it is not very common to cross-validate it. It is usually based on memory constraints (if any), or rather arbitrarily set to some value, for example 32, 64 or 128.
- In practice, we use powers of 2 for the mini-batch size because many vectorized operations work faster when their inputs are sized in powers of 2.

Back Propagation

Back Propagation

- We stated that neural networks can learn their weights and biases using the gradient descent algorithm. There is, however, a gap in our explanation: we did not explain how we compute the gradient of the loss (cost) function.
- We calculate the gradients using a fast algorithm known as ***backpropagation***.\

- The backpropagation algorithm was originally introduced in the 1970s, but its importance was not fully appreciated until now famous paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams entitled: *Learning representations by back-propagating errors*, Nature Vol. 323 9 Oct 1986.
- That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural networks to deal with problems previously too costly to solve.

- Backpropagation updates the weights and biases in the network after each forward pass of the information. After making a prediction (forward pass), backpropagation process measures the error and goes through each layer in reverse to measure the error contribution from each connection (reverse pass) using chain rule and finally tweaks the connection weights to reduce the error.
- The backpropagation computes the partial derivatives $\partial L/\partial w$ and $\partial L/\partial b$ of the cost function L with respect to any weight w or bias b in the network.

Automated Derivatives, Backpropagation

- When using TensorFlow, we do not explicitly calculate any derivatives. We do not have to.
- Tensorflow includes the method `tf.gradients()` to symbolically compute the gradients of all graph steps and return those as tensors.
- We rarely need to make calls to method `tf.gradients()`, because TensorFlow also includes optimized implementations of the gradient descent algorithm.
- When writing TF code, we present high level formulas on how things should work without a need to go in-depth with implementation details and the math.
- We still need understanding of backpropagation. It is a technique used for efficiently computing the gradients on a computational graph.

Derivatives, Simple Rules

- Partial derivative, with symbol $\frac{df}{dx}$ replaced with $\frac{\partial f}{\partial x}$ is used when f is a function of several variables, i.e., $f = f(x, y, z, \dots)$ and we are interested in the rate of change of f when we change one variable and keep other variables fixed.
- For example, if $f(x, y) = xy$, then $\frac{\partial f}{\partial x} = y \qquad \frac{\partial f}{\partial y} = x$
- Similarly, if $f(x, y) = x + y$, then $\frac{\partial f}{\partial x} = 1 \qquad \frac{\partial f}{\partial y} = 1$

$$f(x, y) = \max(x, y) \rightarrow \frac{\partial f}{\partial x} = 1(x >= y) \quad \frac{\partial f}{\partial y} = 1(y >= x)$$

$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

Sigmoid Expression

- Sigmoid, which we frequently use , has a useful property. The derivative of the sigmoid function could be simply expressed over the sigmoid function itself:

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

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right)$$

$$\frac{d\sigma(x)}{dx} = (1-\sigma(x))\,\sigma(x)$$

- Once you calculate the value

$\sigma(x)$, you simply use this formula to determine its derivative:

$$d\sigma(x)/dx$$

Chain Rule

- Chain rule is a convenient mnemonics for writing derivatives of nested functions.
- For example, if y is a result of nesting functions $f()$, $g()$ and $h(x)$, such as

$$y = f\Big(g\big(h(x)\big)\Big),$$

- The derivative dy/dx is determined as:

$$\frac{dy}{dx} = \frac{df}{dg}\,\frac{dg}{dh}\,\frac{dh}{dx}$$

- You could prove the rule for yourself by looking at a simpler expression, first:

$$y = f(g(\mathrm{x})), \quad \frac{dy}{dx} = \lim_{\Delta x \to 0}\frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0}\frac{f\big(g(x+\Delta x)\big) - f(g(x))}{\Delta x}$$

$$\frac{\Delta y}{\Delta x} = \frac{f\big(g(x+\Delta x)\big) - f(g(x))}{\Delta x} = \frac{f\Big(g(x) + \Delta x\frac{\Delta g}{\Delta x}\Big) - f(g(x))}{\Delta x} = \frac{f\big(g(x)\big) + \frac{\Delta f}{\Delta g}\frac{\Delta g}{\Delta x}\Delta x - f(g(x))}{\Delta x} = \frac{\Delta f}{\Delta g}\,\frac{\Delta g}{\Delta x}$$

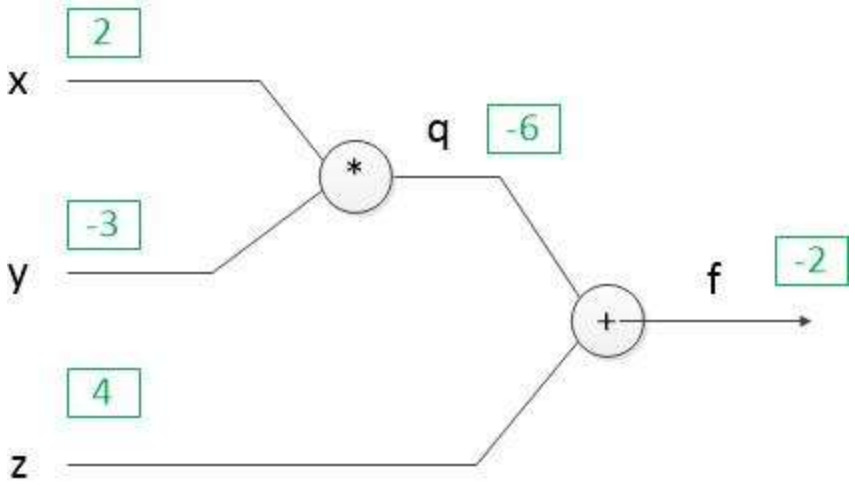
$$\frac{dy}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

Backpropagation, Computational Graph

- Objective of backpropagation is to find the value of the gradient of the loss function L with respect to weights W and biases b.
- L is usually a very, very massive and complex function. Let us examine a much simpler function.

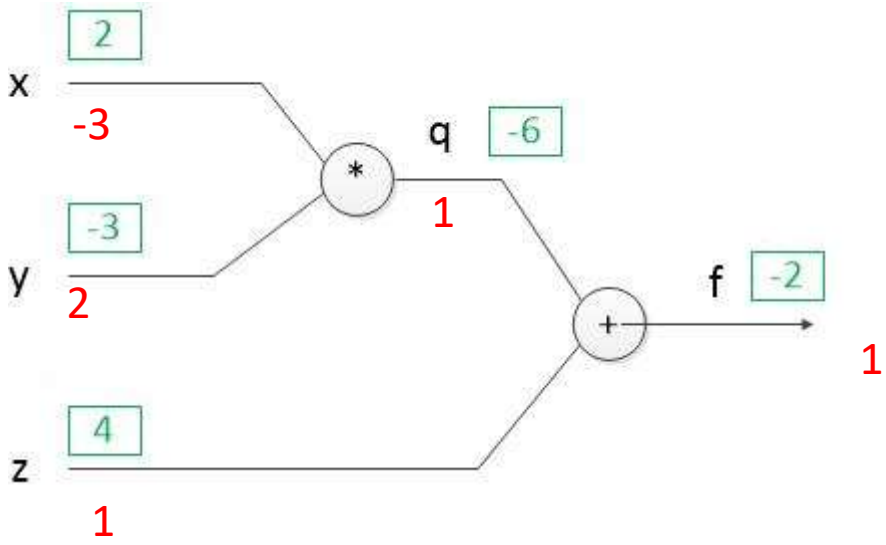
$$f = x * y + z$$

- Computational graph for this expression looks like the one below. Let us choose values of the variables, $x = 2, y = -3$ and $z = 4$ and calculate every expression along the way. Those are forward values. Notice that we treat operations as nodes and label all intermediary variables, like $q = x * y$



Rate of Change of function $f(x, y, z)$

- We want to find the rate of change of $f(x, y, z) = x * y + z = q + z$, with respect to all variables: x, y and z, i. e. , record the derivatives of any function with respect to its inputs.
- Notice: $\frac{\partial f}{\partial z} = 1$. Let us place that 1 below the input value of z.
- Also, $\frac{\partial f}{\partial q} = 1$. Let us place that 1 below the forward value of q.
- $\frac{\partial q}{\partial x} = y$ = -3 and $\frac{\partial q}{\partial y} = \mathrm{x} = 2$



Please notice that all red values could be calculated during the forward pass. Those are cheep calculations. As we are moving forward, we calculate all intermediary values, and also all the partial derivatives of those intermediaries with respect to various inputs.

We store those partial derivatives as partial content of the variable used in the denominator of the derivative. For example, value of $\partial q/\partial x$ = -3, is stored with variable x

- To get gradient of f we apply the chain rule:

$$\frac{\partial f}{\partial z} = 1 \quad \frac{\partial f}{\partial f} = 1$$

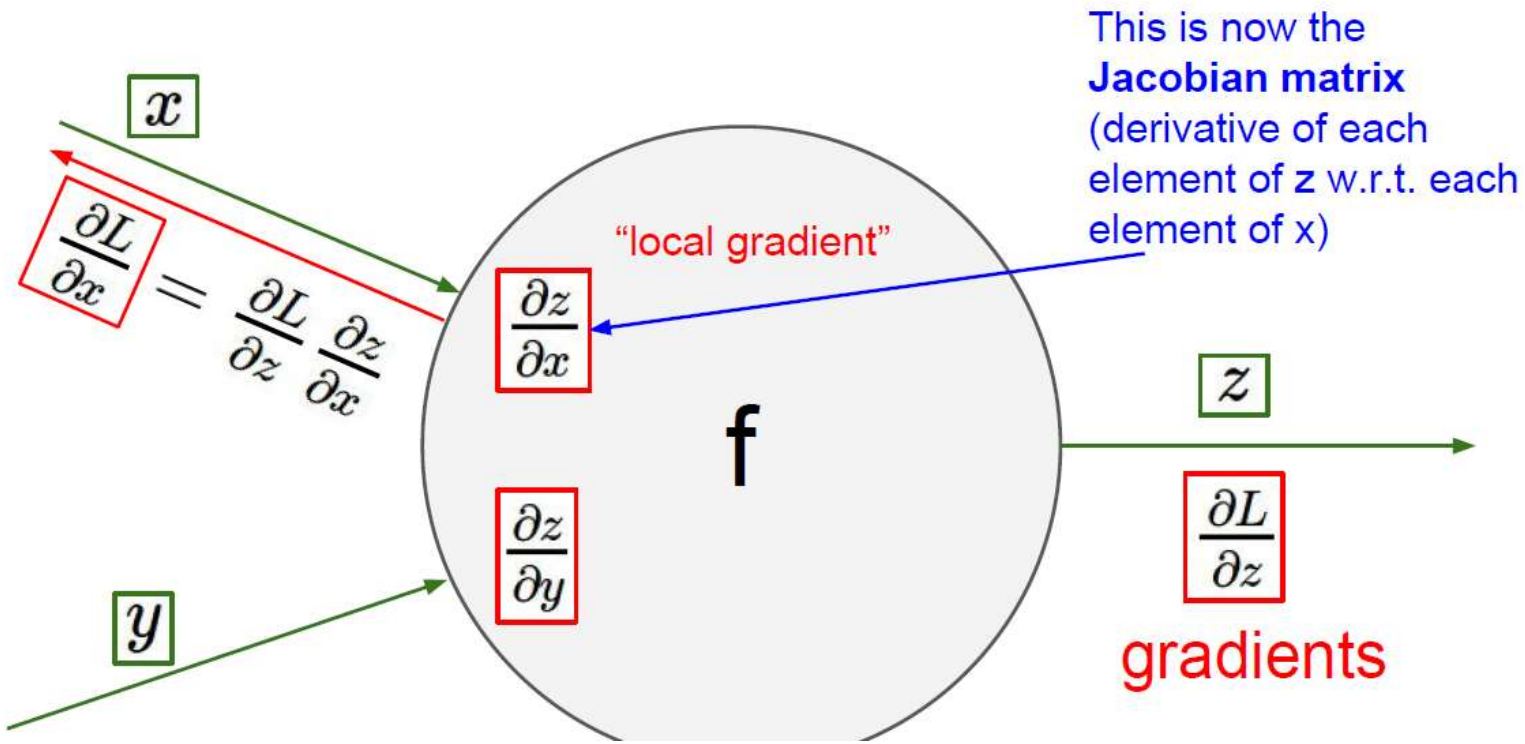
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = 1 * (-3) = -3$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = 1 * (2) = 2$$

:

Gradients for vectorized code

- If our flow variables: z, y and z in the diagram below are vectors rather than scalars, partial derivatives are replaced with Jacobian matrices:



Jacobian

- In vector calculus, all first-order partial derivatives of a vector-valued function of several variables can be arranged as a matrix. Such a matrix is called the Jacobian (/dʒəˈkoʊbiən/) matrix .
- When the function has the same number of arguments as the number of vector components of its output, the matrix is a square-matrix and its determinant is called the Jacobian determinant.
- Both the matrix and the determinant are often referred to simply as the Jacobian.
- Suppose f : $\mathbb{R}^n \to \mathbb{R}^m$ is a function such that each of its first-order partial derivatives exist on $\mathbb{R}^n$. This function takes a point x ∈ $\mathbb{R}^n$ as input and produces a vector f(x) ∈ $\mathbb{R}^m$ as an output. Then the Jacobian matrix of f is defined to be an $m \times n$ matrix, denoted by **J**, whose (i, j) entry is $\mathbf{J_{ij}} = \frac{\partial f_i}{\partial x_j}$, or explicitly:

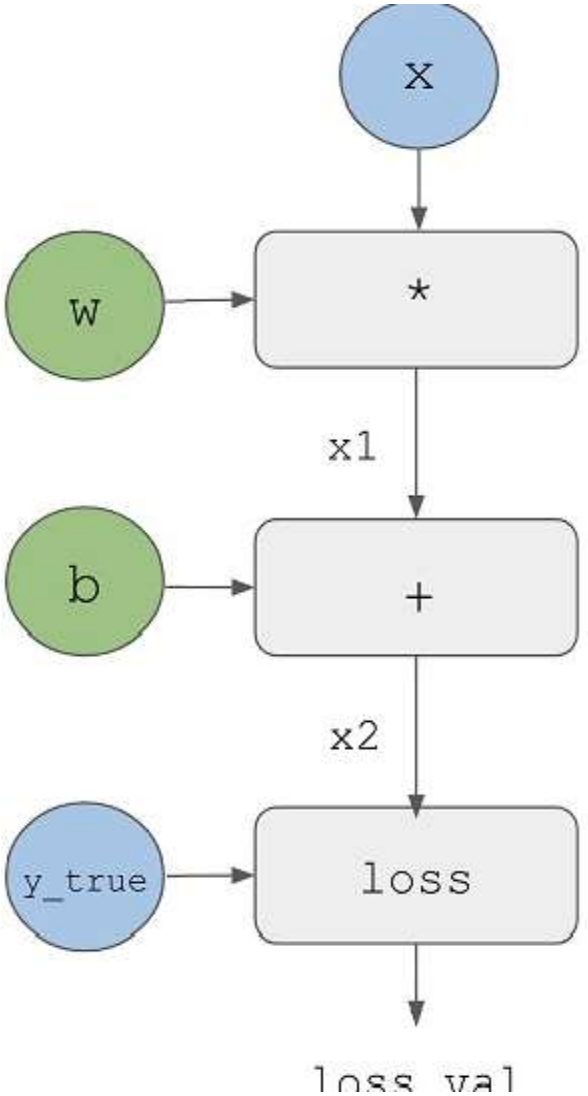
$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Backpropagation on a Simple Perceptron

- We will consider a simplified version of a Perceptron, where all variables are scalars.
- We have two scalar variables `w`, and `b` and one scalar input `x`. Usually, we apply some non-linear operation, e.g., sigmoid, to produce an output y. We are not doing that here:

  `y = x2 = w*x + b`
- Finally, we calculate loss function. The loss is simple, an absolute value of the difference:
  `loss_val=abs(y_true - y)`

- We want to update `w` and `b` to minimize `loss_val`. We are interested in computing :
- `grad(loss_val, b)` and `grad(loss_val, w)`.



Forward Pass

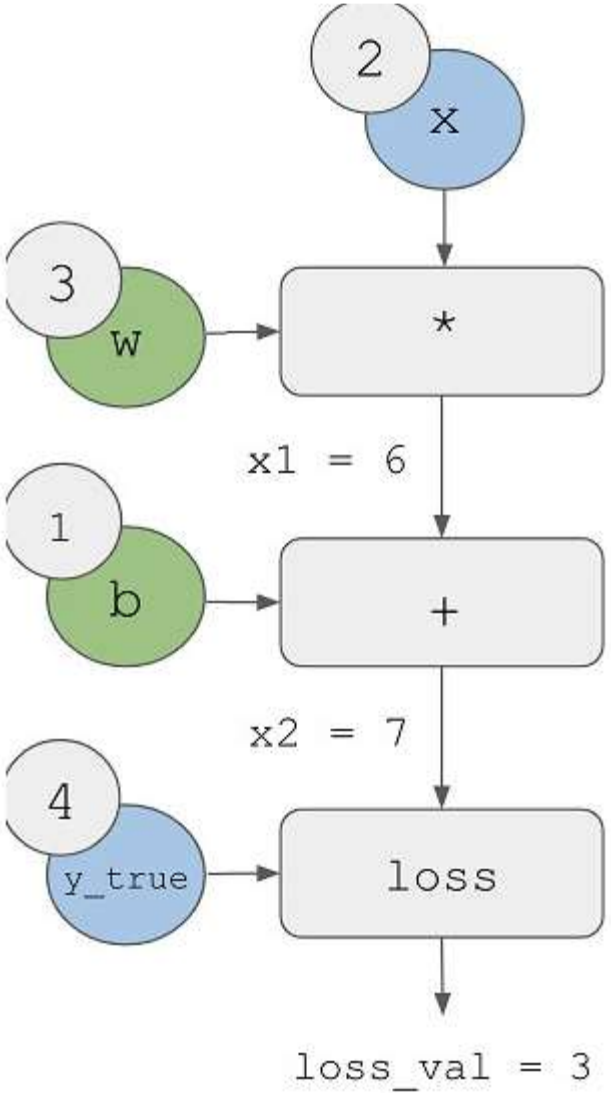
- Let us set concrete values for the "input nodes" of the graph: `x`, the target `y_true`, `w` and `b`.
- We perform calculations with those values at all nodes in the graph, from top to bottom, until we reach `loss_val`. This is the *forward pass*.
- We use:

```
x = 2,
w = 3,
b = 1,
y_true = 4
```

- To get:

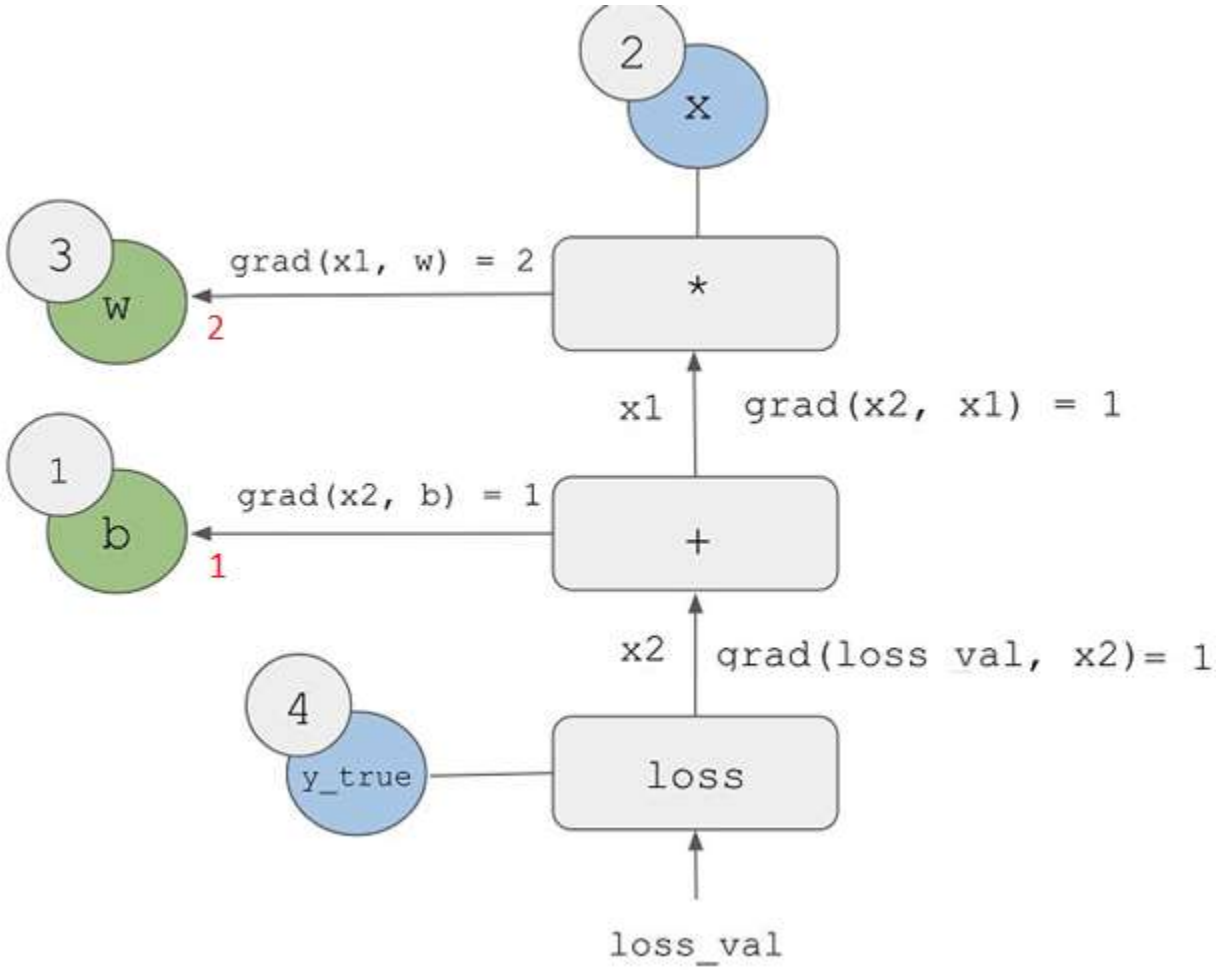
```
x1 = x*w = 6
x2 = x1 + b = 7
loss = abs(y_true - x2) = 3
```

-



Reverse Graph -> Backward Pass

- Now let's "reverse" the graph: for each edge in the graph going from `a` to `b`, we will create an opposite edge from `b` to `a`, and ask "how much does `b` vary when `a` vary"? That is to say, what is `grad(b, a)`?
- We will annotate each inverted edge with this value. This backward graph represents the *backward pass*.



Calculate Gradients

We have:

- `grad(loss_val, x2) = 1`, because as x_2 varies by an amount epsilon, `loss_val = abs(4 - x2)` varies by the same amount.
- `grad(x2, x1) = 1`, because as x1 varies by an amount epsilon, `x2 = x1 + b = x1 + 1` varies by the same amount.
- `grad(x2, b) = 1`, because as b varies by an amount epsilon, `x2 = x1 + b = 6 + b` varies by the same amount.
- `grad(x1, w) = 2`, because as w varies by an amount `epsilon`, `x1 = x * w = 2 * w` varies by `2 * epsilon`.

- What the chain rule says about this backward graph is that you can obtain the derivative of a node with respect to another node by multiplying the derivatives for each edge along the path linking the two nodes.
- For instance:

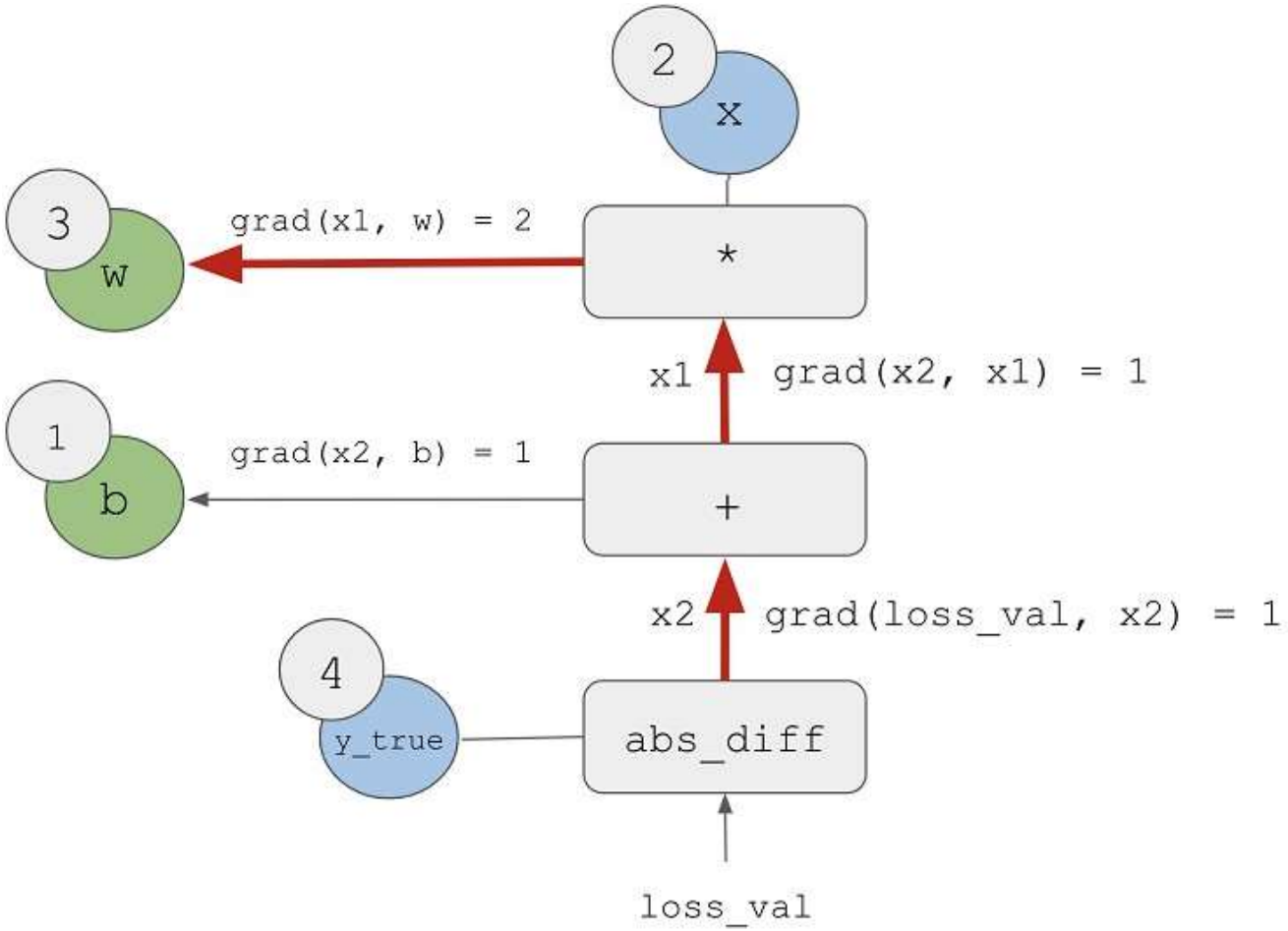
```
grad(loss_val, w) = grad(loss_val, x2) * grad(x2, x1) * grad(x1, w).
```

Path from `loss_val` to `w` in backward graph

- By applying the chain rule to our graph, we obtain what we were looking for:

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

Note: If there are multiple paths linking the two nodes of interest `a, b` in the backward graph, we would obtain `grad(b,a)` contributions of all the paths



Back Propagation

- Backpropagation is simply the application of the chain rule to a computation graph. There's nothing more to it.
- All the partial derivatives with respect to the input variables at every step of the forward pass are calculated at the same time as the forward calculations at approximately the same computational cost as the forward calculations.
- When we need the partial derivative of the final loss value with respect to any variable (any W-weight and any b-bias) we start with the final value and work backward from the top layers to the bottom layers, computing the contribution that each parameter has on the gradient of the loss value. That's where the name "backpropagation" comes from: we "back propagate" the contributions of different nodes in a computation graph.
- Today, we build neural networks in modern frameworks, such as TensorFlow or PyTorch, that are capable of *automatic differentiation.*
- Automatic differentiation is implemented with the kind of computation graph presented on previous slides
- Automatic differentiation makes it possible to retrieve the gradients of arbitrary compositions of differentiable tensor operations without doing any extra work besides writing down the forward pass.

Pattern of BackPropagation, `tf.GradientTape`

- You should remember the pattern. The derivative on each layer is the product of the derivatives of the layers after it and the output of the layer before. That's the magic of the chain rule and what the algorithm takes advantage of.
- We go forward from the inputs calculating the outputs of each hidden layer up to the output layer. Then we start calculating derivatives going backwards through the hidden layers. We do approximately the same number of calculations in the "backward" pass as we did in the forward pass since we are reusing elements already calculated. Computational complexity and cost of backpropagation is of the same order of magnitude as the forward calculation.
- TensorFlow provides the `tf.GradientTape` API for automatic differentiation - computing the gradient of any differentiable function with respect to its input variables.
- Tensorflow "records" all operations executed inside the context of a `tf.GradientTape` onto a "`tape`".
- Tensorflow then uses that `tape` and the gradients associated with each recorded operation to compute the gradients of a "recorded" computation using reverse mode differentiation.

- Process of automated (automatic) calculation of various gradients is frequently referred to as *Autodifferentiation of Autodiff.*

Autodiff and GradientTape

Autodiff

- To understand how to use TensorFlow `autodiff` to compute gradients automatically, let's consider a simple toy function:

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

- Here, `w1**2` is the square operator: `w**2 == w`^{2}. Using calculus, you can analytically find that the partial derivative of this function

$$f(W_1, W_2) = 3 * {W_1}^2 + 2 * W_1 * W_2$$

- with respect to `w1`, is equal to:

$$\partial f/(\partial W_1) = 6 * W_1 + 2 * W_2.$$

- The partial derivative with respect to `w2` is equal to:

$$\frac{\partial f}{\partial W_2} = 2 * W_1$$

- For this example, at the point `(w1, w2) = (5, 3)`, these partial derivatives are equal to `36` and `10`, respectively, so the gradient vector is `(36, 10)`.

Calculating partial derivatives by increments

- If this were a neural network, the function would be much more complex, typically with tens of thousands of parameters, and finding the partial derivatives analytically by hand would be an almost impossible task.
- For example, one solution could be to compute an approximation of each partial derivative by measuring how much the function's output changes when you change the corresponding parameters by a small variation `eps`. `eps` is a common symbol for a small value or increment.

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.000000003174137
```

- Apart from small numerical error, this is about right! Importantly, you need to call `f()` at least 1.5 time per parameter (`f(w1 +eps, w2)`, `f(w1,w2+eps)`, we compute `f(w1, w2)` just once).
- This works rather well. It is trivial to implement, and it is a good approximation. Does it?
- For 10,000 or million parameters, in large neural networks, this approach is intractable

Autodiff

- Calculating gradients using small increments of all variables might be too complex, too lengthy and error prone. Instead, we should use `autodiff` techniques .
- TensorFlow, PyTorch and other APIs make this relatively simple. The main construct in TensorFlow is the class `tf.GradientTape`.
- `tf.GradientTape` is a context manager. Operations are recorded if they are executed within this context manager and if at least one of their inputs is being "`watched`".
- Trainable variables (created by `tf.Variable` or `tf.compat.v1.get_variable`, where `trainable=True` is default in both cases) are automatically watched.
- Tensors can be manually watched by invoking the `watch` method on this context manager.
- For example, consider the function `y = x * x`. The gradient at `x = 3.0` can be computed as:

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

dy_dx = g.gradient(y, x) #
print(dy_dx.numpy())
6.0
```

Example, continued

- Partial derivatives of a function of two variable could be determined by the following :

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
        z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

- We first define two variables `w1` and `w2`, then we create a `tf.GradientTape` context that will automatically record every operation that involves those variables, and finally we ask this tape to compute the gradients of the result `z` with respect to both variables `[w1, w2]`.

```
print(gradients[0].numpy(), gradients[1].numpy())
36.0 10.0
```

- Not only is the result accurate (the precision is only limited by the floating-point errors), but the `gradient()` method only goes through the recorded computations once (in reverse order), no matter how many variables there are, so it is very efficient.

Note on Memory, Efficiency and tape longevity

- We should only put the strict minimum of code inside `the tf.GradientTape()` block, to save memory.
- Alternatively, we can pause recording by creating with `tape.stop_recording()` block inside the `tf.GradientTape()` block.

- The tape is automatically erased immediately after you call its `gradient()` method, so you will get an exception if you try to call `gradient()` twice:

```
with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1)
try:
    dz_dw2 = tape.gradient(z, w2)
except RuntimeError as ex:
    print(ex)

GradientTape.gradient can only be called once on non-persistent
tapes.
```

Persistent `tape`

- By default, the resources held by a `GradientTape` are released as soon as `GradientTape.gradient()` method is called.
- To compute multiple gradients over the same computation, create a persistent gradient `tape`. This allows multiple calls to the `gradient()` method as resources are released only when the `tape` object is garbage collected.
- If you need to call `gradient()` more than once, you must make the `tape` persistent, and delete it when you are done with it to free resources:

```
with tf.GradientTape(persistent=True) as tape:
        z = f(w1, w2)

dz_dw1, dz_dw2 = tape.gradient(z, [w1,w2]) # => tensor 36.0,10.0
dz_dw2 = tape.gradient(z, w2) # => tensor 10.0, works fine now!
del tape
```

Gradients with respect to constants

- By default, the tape will only track operations involving `tf.Variables`, so if you try to compute the gradient of `z` with regards to anything else than a variable, the result will be None:

```
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
        z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [None, None]
```

- However, you can force the tape to watch any tensors you like, and to record every operation that involves them. You can then compute gradients with respect to these tensors, as if they were variables:

```
with tf.GradientTape() as tape:
        tape.watch(c1)
        tape.watch(c2)
        z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) #
# returns [tensor 36., tensor 10.]
```

- This can be useful in some cases. For example, if you want to implement a regularization loss that penalizes activations that vary considerably when the inputs vary little: the loss will be based on the gradient of the activations with respect to the inputs. Since the inputs are not variables, you would need to tell the tape to watch them.

Gradients of a List of Tensors

- If you compute the gradient of a list of tensors (e.g., `[z1, z2, z3]`) with regards to some variables (e.g., `[w1, w2]`), TensorFlow computes the sum of the gradients of those tensors (i.e.,

```
gradient(z1, [w1, w2] ) +  gradient(z2, [w1, w2] ) + gradient(z3, [w1, w2])
```

- Due to the way reverse-mode autodiff works, it is not possible to compute the individual gradients (`z1`, `z2` and `z3`) without calling `gradient()` multiple times (once for `z1`, once for `z2` and once for z3) which requires making the tape persistent (and deleting it afterwards).

Second Order Derivatives

- `GradientTapes` can be nested to compute higher-order derivatives. For example

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
  g.watch(x)
  with tf.GradientTape() as gg:
    gg.watch(x)
    y = x * x
  dy_dx = gg.gradient(y, x)      # Will compute to 6.0
d2y_dx2 = g.gradient(dy_dx, x)   # Will compute to 2.0
```

Jacobian

- In vector calculus, all first-order partial derivatives of a vector-valued function of several variables can be arranged as a matrix. Such a matrix is called the Jacobian (/dʒəˈkoʊbiən/) matrix .
- When the function has the same number of arguments as the number of vector components of its output, the matrix is a square-matrix and its determinant is called the Jacobian determinant.
- Both the matrix and the determinant are often referred to simply as the Jacobian.
- Suppose f : $\mathbb{R}^n \to \mathbb{R}^m$ is a function such that each of its first-order partial derivatives exist on $\mathbb{R}^n$. This function takes a point x ∈ $\mathbb{R}^n$ as input and produces a vector f(x) ∈ $\mathbb{R}^m$ as an output. Then the Jacobian matrix of f is defined to be an $m \times n$ matrix, denoted by **J**, whose (i, j) entry is $\mathbf{J_{ij}} = \frac{\partial f_i}{\partial x_j}$, or explicitly:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Hessian Matrix

- Suppose $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is a function taking as input a vector $x \in \mathbb{R}^n$ and outputting a scalar $f(x) \in \mathbb{R}$; if all the second partial derivatives of f exist and are continuous over the domain of the function, then the Hessian matrix $\mathbf{H}$ of f is a square-matrix, of dimensions $n * n$, usually defined and arranged as follows:

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\, \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\, \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\, \partial x_1} & \dfrac{\partial^2 f}{\partial x_n\, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Calculation of Hessian Matrix

- To compute second order partial derivatives (the Hessians, i.e., the partial derivatives of the partial derivatives), we need to record the operations that are performed when computing the first-order partial derivatives (the Jacobians).
- Jacobians require a second tape. Here is how it works:

```
with tf.GradientTape(persistent=True) as hessian_tape:
    with tf.GradientTape() as jacobian_tape:
        z = f(w1, w2)
    jacobians = jacobian_tape.gradient(z, [w1, w2])

hessians = [hessian_tape.gradient(jacobian, [w1, w2])
for jacobian in jacobians]
del hessian_tape
```

- The inner tape is used to compute the Jacobians, as we did earlier.
- The outer tape is used to compute the partial derivatives of each Jacobian. Since we need to call `gradient()` once for each Jacobian (or else we would get the sum of the partial derivatives over all the Jacobians, as explained earlier), we need the outer tape to be persistent, so we delete it at the end.
- The Jacobians are obviously the same as earlier (36.0 and 10.0).
- Now we also have the Hessians:

```
 print(hessians)  # dz_dw1_dw1, dz_dw1_dw2, dz_dw2_dw1, dz_dw2_dw2
[[<tf.Tensor: id=830578, shape=(), dtype=float32, numpy=6.0>,
<tf.Tensor: id=830595, shape=(), dtype=float32, numpy=2.0>],
[<tf.Tensor: id=830600, shape=(), dtype=float32, numpy=2.0>, None]]
```

Preventing gradients from backpropagation

- In some rare cases you may want to stop gradients from backpropagating through some part of your neural network.
- To do this, you must use the `tf.stop_gradient()` function. This function just returns its inputs during the forward pass (like `tf.identity()`), but it does not let gradients through during backpropagation (it acts like a constant).
- For example:

```
def f(w1, w2):
        return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

with tf.GradientTape() as tape:
        z = f(w1, w2) # same result as without stop_gradient()

gradients = tape.gradient(z, [w1, w2])
# => returns [tensor 30., None]
```

Linear Classifier Example

A linear classifier using *tf.GradientTape*

- As an illustration of techniques, we discussed, we will implement a binary classifier. We will use TensorFlow tensors and `tf.GradientTape` machinery.
- First, we will produce some linearly-separable synthetic data to work with: two classes of points in a 2D plane.

```
import numpy as np
import tensorflow as tf
num_samples_per_class  = 1000
negative_samples = np.random.multivariate_normal(
    mean=[0, 3], cov=[[1, 0.5],[0.5, 1]], size=num_samples_per_class)
positive_samples = np.random.multivariate_normal(
    mean=[3, 0], cov=[[1, 0.5],[0.5, 1]], size=num_samples_per_class)
```

- Both groups of 1000 random 2D points have a specified "mean“ and "covariance matrix". The first (negative) group is positioned around point `[x=0,y=3]`. The second (positive) is positioned around point `[x=3,y=0]`.
- The "covariance matrix" describes the shape of the point cloud, and the "mean" describes its position in the plane. `Cov=[[1, 0.5],[0.5, 1]]` corresponds to "an oval-like point cloud oriented from bottom left to top right".
- The other class of points has a different mean and the same covariance matrix (point cloud with a different position and the same shape).

Two Classes

- Stacking the two classes into an array with shape (2000, 2)

```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
```

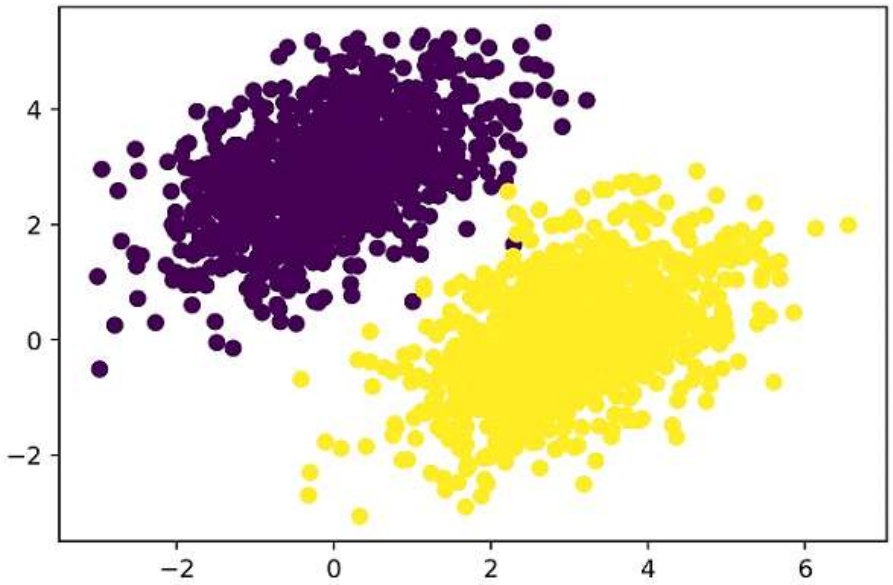
- The corresponding targets (labels) is an array of zeros and ones of shape (2000, 1), where `targets[i, 0]` is `0` if `inputs[i]` belongs to class `0`. Similarly, the target is `1` if `inputs[i]` belongs to class `1`:

```
targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype='float32'),
np.ones((num_samples_per_class, 1), dtype='float32')))
```

- We plot our data with Matplotlib:

```
import matplotlib.pyplot as plt
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])
plt.show()
```

- We are seeking the decision boundary between those two classes.
- The decision boundary (linear classifier)will separate those two groups of points.



Linear, Binary, Classifier

- A linear classifier is an "affine" transformation (`prediction` = `W` • `input` + `b`) trained to minimize the square of the difference between predictions and the targets.
- Let's create our variables `W` and `b`, initialized with random values and with zeros respectively.

```
input_dim = 2
output_dim = 1
W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim,output_dim)))
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
```

- The forward pass function can be defined as:

```
def model(inputs):
      return tf.matmul(inputs, W) + b
```

- Inputs are vectors `[x,y]`. The linear classifier operates on inputs by multiplying them with 2D vector `W`. `W` is just two scalar coefficients, `w1` and `w2`: `W = [[w1], [w2]]`.
- Bias `b` is a single scalar coefficient.
- For given input point `[x, y]`, the prediction value of the model is:

```
prediction = [[w1], [w2]] • [x, y] + b = w1 * x+ w2 * y + b
```

The Loss Function

- As the loss function, we will use the mean square error loss:

```
def square_loss(targets,  predictions):
    per_sample_losses = tf.square(targets - predictions)
    return tf.reduce_mean(per_sample_losses)
```

- Remember, `per_sample_losses` is a tensor with the same shape as `targets` and `predictions`, containing per-sample loss scores for all 2000 points. Function `tf.square()` calculates element wise square of `targets - predictions`

- We need to average these per-sample loss scores into a single scalar loss value. This is what `tf.reduce_mean()` does.

- In the following we will do *batch training* rather than *mini-batch training*.

Training Step

- The training step receives some training data and updates the weights `W` and `b` so as to minimize the `loss` on the data.
- Every training step moves parameters `W` and `b` by the learing rate λ in the direction opposite to the gradient of the `loss` function with respect to `W` and `b`. We will assume somewhat arbitrary value of λ: `learning_rate = 0.1`.
- Inside each training step, we make a forward calculation of the `loss` and collect the gradients needed to perform the adjustment of the values of `W` and `b`. We collect those gradients using `tf.GradientTape()`

```
def training_step(inputs,  targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs) # forward pass
        loss = square_loss(predictions,targets)
        # find gradients of loss with respect to W and b
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss,[W, b])
    # Update W and b by descending against the gradient
    W.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b * learning_rate)
    return loss
```

The Batch Training Loop

- For simplicity, we do *batch training* instead of *mini-batch training.* We run each training step (gradient computation and weight update) on the entire data, rather than iterate over the data in small batches.

```
for step in range(35):
    loss = training_step(inputs,  targets)
    print('Loss at step %d: %.4f' % (step, loss))
```

```
Loss at step 0: 1.2237
Loss at step 1: 0.2472
Loss at step 2: 0.1246
Loss at step 3: 0.1027
Loss at step 4: 0.0944
Loss at step 5: 0.0882
Loss at step 6: 0.0828
Loss at step 7: 0.0778
Loss at step 8: 0.0733
Loss at step 9: 0.0691
Loss at step 10: 0.0653
. . . .
Loss at step 31: 0.0308
Loss at step 32: 0.0302
Loss at step 33: 0.0298
Loss at step 34: 0.0293
```

After 30 steps, the training loss seems to have stabilized around 0.029

Predictions and the Decision Boundary

- Because our targets are zeros and ones, a given input point will be classified as "0" if its prediction value is below 0.5, and as "1" if it is above 0.5.
- Function `model()` will assume the values of `W` and `b` determined by just finished training loop.
- Recall that the prediction value for a given point `[x, y]` is simply `prediction == [[w1], [w2]] • [x, y] + b == w1 * x + w2 * y + b`.
- Thus, class "`0`" is defined as: `w1 * x + w2* y + b < 0.5` and class "`1`" is defined as: `w1 * x + w2 * y + b > 0.5`.
- What we looking at is the equation of a line in the 2D plane:
- `w1 * x + w2 * y + b = 0.5`. Above the line, class 1, below the line, class 0.
- Line equations is usually displayed in the format `y = a * x + b`;
- In that format, our line becomes:

```
y = - w1 / w2 * x +(0.5 - b) / w2.
```

- We will generate 100 regularly spaced numbers between -1 and 4, which we will use to plot the line (the decision boundary).
- In plotting this line, we use ('-r'), what means "plot it as a red line"). We plot model's predictions on the same plot

Display the Predictions and the Decision Boundary

```
predictions = model(inputs)
plt.scatter(inputs[:,  0], inputs[:, 1], c=predictions[:, 0] > 0.5)
x = np.linspace(-1, 4, 100)
y = - W[0] /  W[1] * x + (0.5 - b) / W[1]
plt.plot(x, y, '-r')
plt.scatter(inputs[:,  0], inputs[:, 1], c=predictions[:, 0] > 0.5)

plt.show()
```