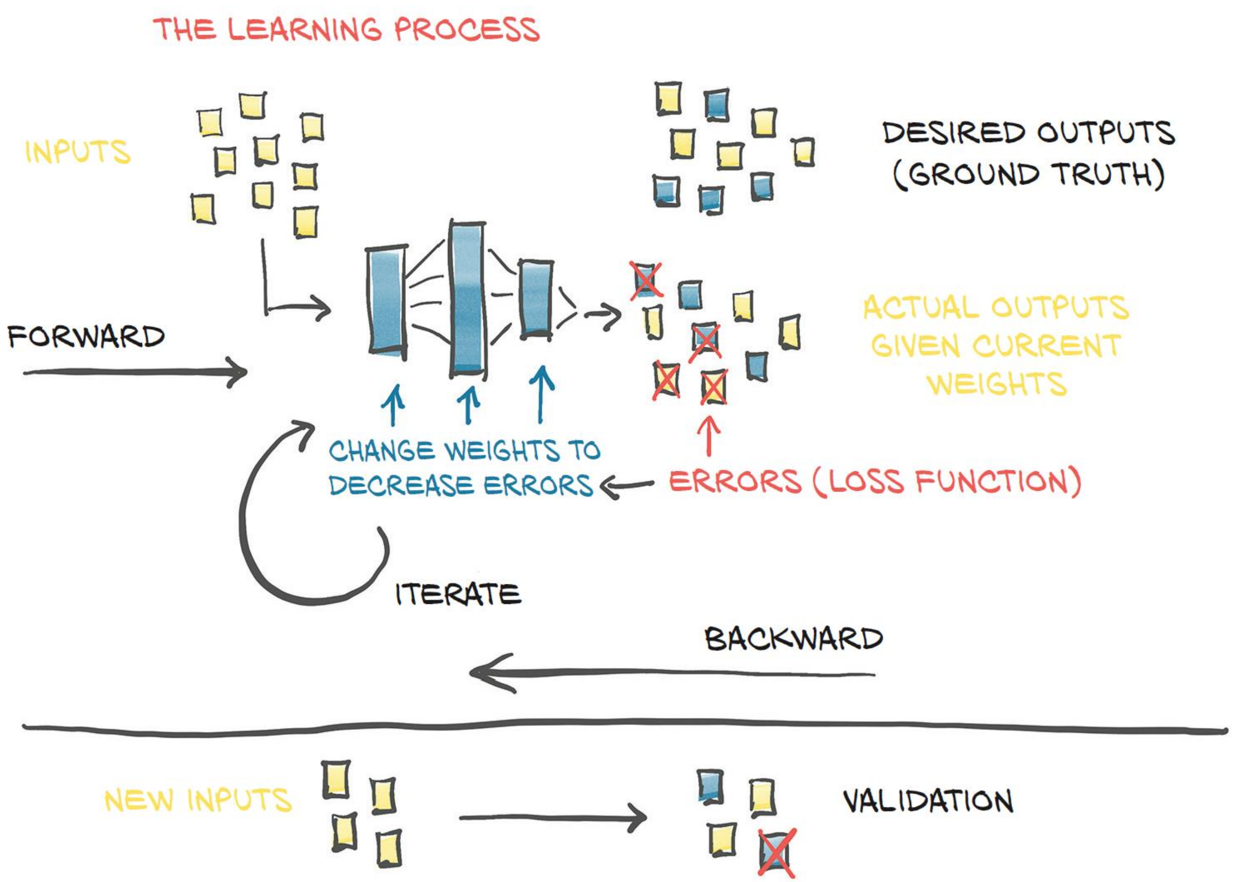
Introduction to ML
Lecture 16: Gradient Descent, BackPropagation in Pytorch

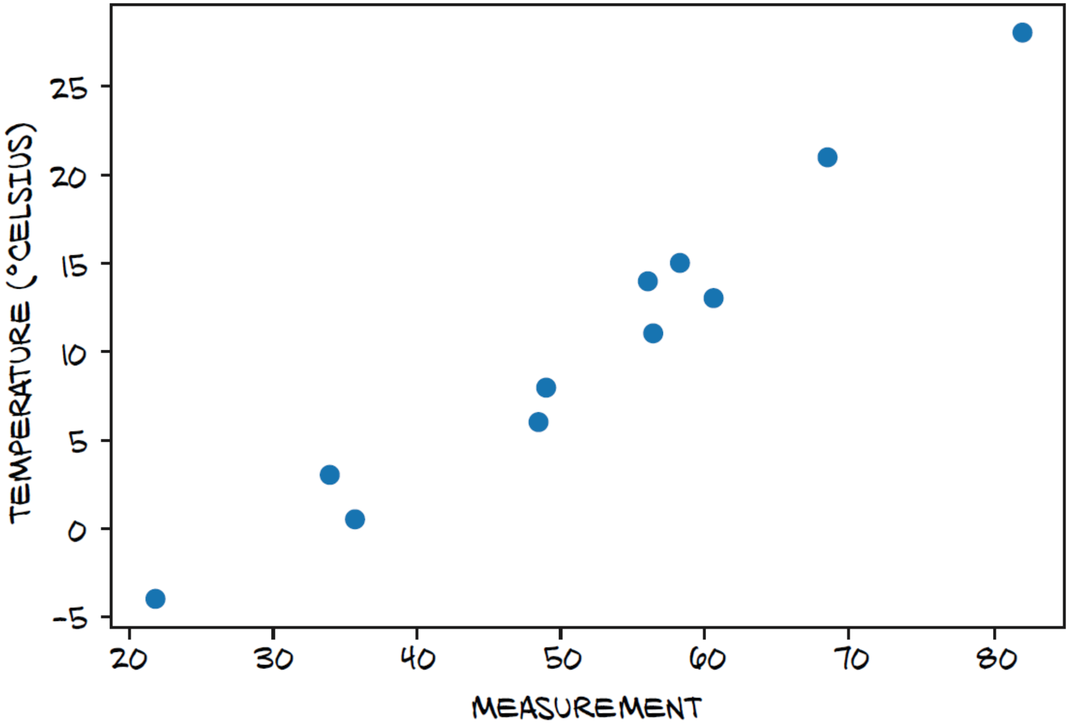
Hamed Tabkhi
Department of Electrical and Computer Engineering,
University of North Carolina Charlotte (UNCC)
htabkhiv@uncc.edu

General Supervised Learning Framework



Example

Goal: Predicting temperature based on some measured values.




```
# In[2]:
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0]
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]
t_c = torch.tensor(t_c)
t_u = torch.tensor(t_u)
```

Linear Model

```
t_c = w * t_u + b
```

Finding a linear relationship between t_u and t_C

Pytorch code:

```
def model(t_u, w, b):
return w * t_u + b
```

Aim: finding a linear relation shop between the input and the desired output.

Loss Calculation

How to calculate the loss:

`|t_p - t_c|` and `(t_p - t_c)^2`.

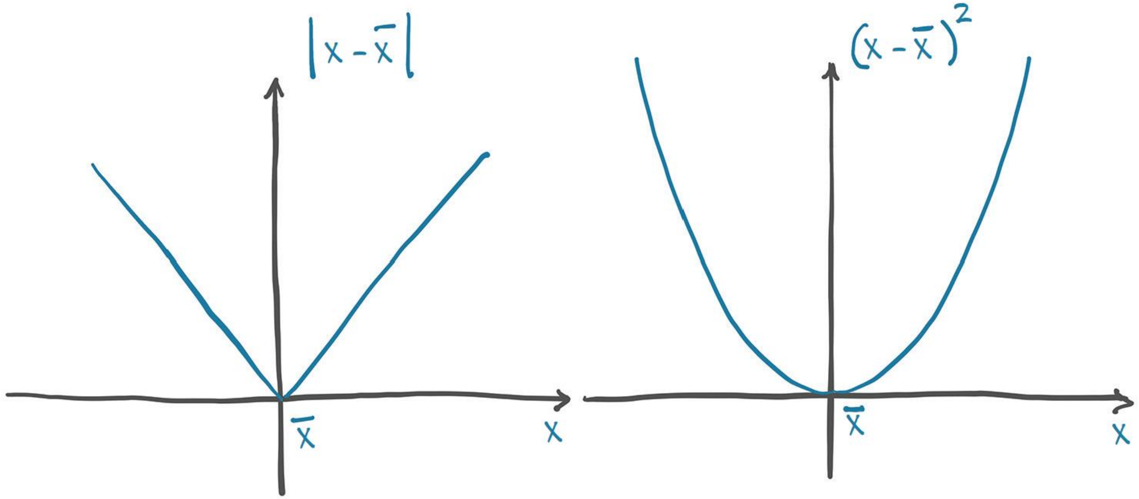
Note: Loss should be a positive number

The square of the differences behaves more nicely around the minimum.

The square difference also penalizes wildly wrong results more than the absolute difference does.

```
def loss_fn(t_p, t_c):
squared_diffs = (t_p - t_c)**2
return squared_diffs.mean()
```

Note: this is the average loss



Loss Calculation

```
# In[5]:
w = torch.ones(())                   initial W is 1
b = torch.zeros(())                 initial b is 0
t_p = model(t_u, w, b)
t_p
# Out[5]:
tensor([35.7000, 55.9000, 58.2000, 81.9000, 56.3000, 48.9000, 33.9000,
21.8000, 48.4000, 60.4000, 68.4000])
```

- And check the value of the loss:

```
# In[6]:
loss = loss_fn(t_p, t_c)
loss
# Out[6]:
tensor(1763.8846)
```

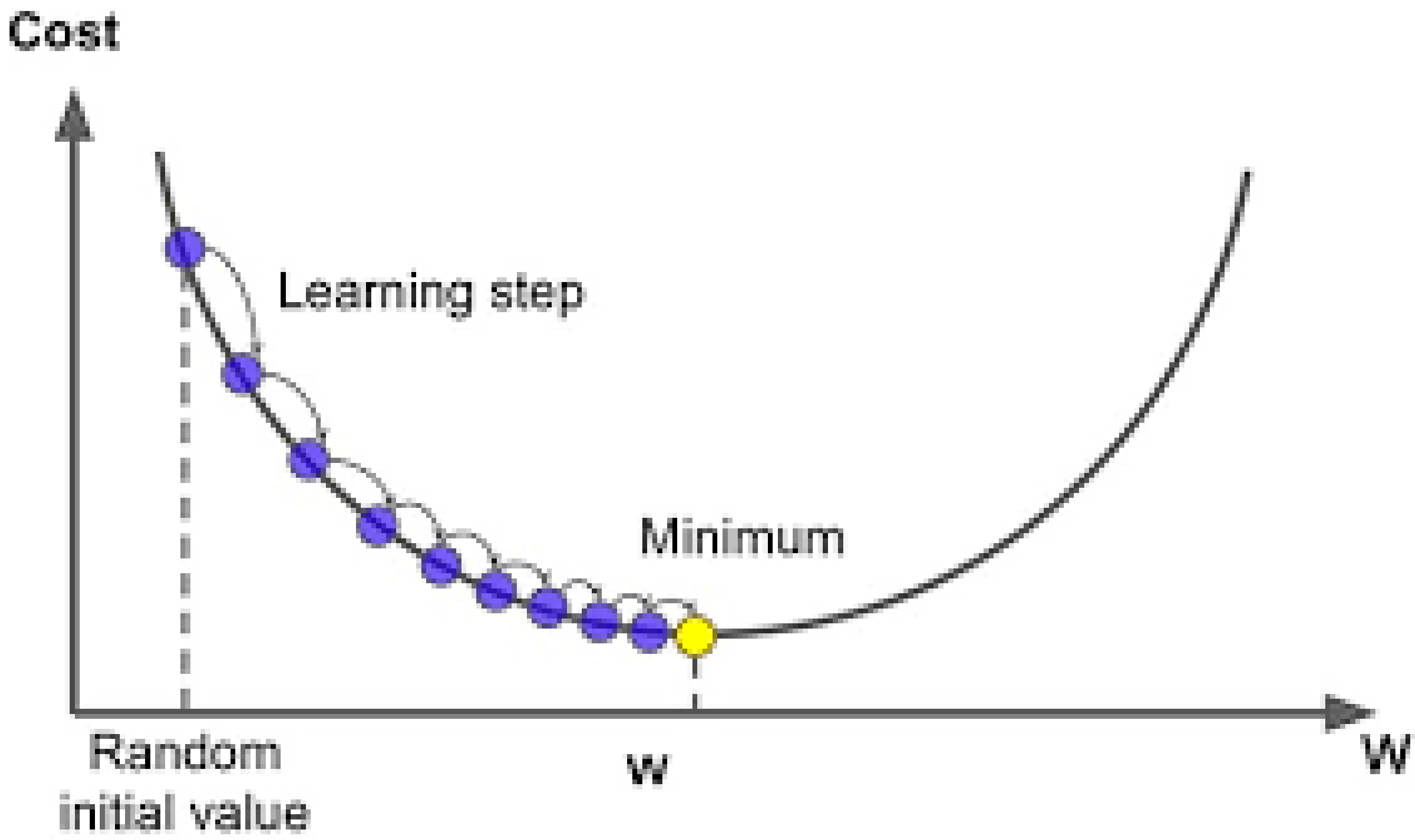
Gradient Decent

- The idea is to compute the rate of change of the loss with respect to each parameter (in this case W), and modify each parameter in the direction of decreasing loss.
- `# In[8]:`

```
delta = 0.1
loss_rate_of_change_w = \
(loss_fn(model(t_u, w + delta, b), t_c) -
loss_fn(model(t_u, w - delta, b), t_c)) / (2.0 * delta)
```

- **This is saying that in the neighborhood of the current values of `w` and `b`, a unit increase in `w` leads to some change in the loss.**
  - If the change is negative, then we need to increase `w` to minimize the loss,
  - if the change is positive, we need to decrease `w`.

Gradient Decent



Scaling Factor (Learning Rate)

- Defines the rate of decreasing/increasing parameters (w and b)!

```
# In[9]:
learning_rate = 1e-2     means 0.01 as the changing rate for our parameters.
w = w - learning_rate * loss_rate_of_change_w
```

- The idea is to compute the rate of change of the loss with respect to each parameter, and modify each parameter in the direction of decreasing loss
- Learning rate defines how fast or slow we will move towards the optimal weights.
- Basically, learning rate defines the impact of loss rate in changing the values of our parameters.

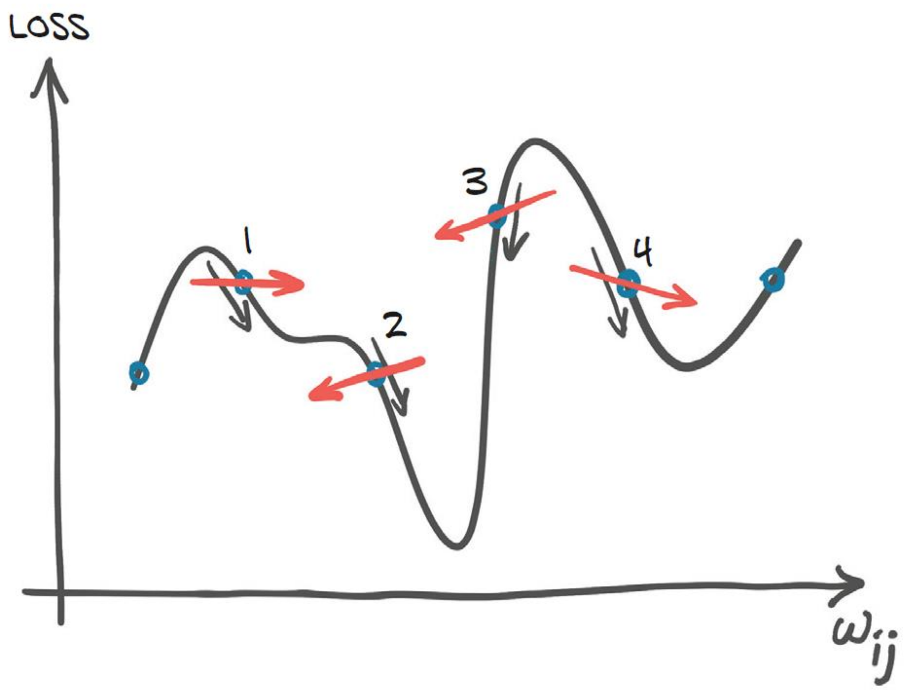
Scaling Factor (Learning Rate)

- We can do the same with `b`:

```
# In[10]:
loss_rate_of_change_b = \
(loss_fn(model(t_u, w, b + delta), t_c) -
loss_fn(model(t_u, w, b - delta), t_c)) / (2.0 * delta)
b = b - learning_rate * loss_rate_of_change_b
```

Analytical Insight

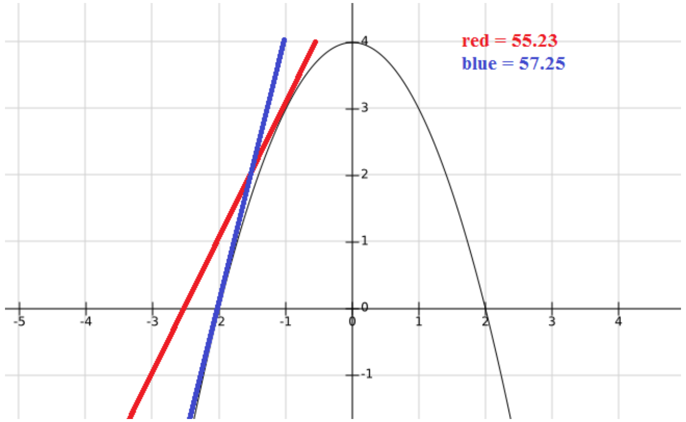
- Value of Delta determines the direction of training per each iteration.
  - Reducing or increasing the parameters
- Value of Learning Rate determines the rate of updating the parameters.
  - Amount we reduce or increase the parameters with respect to already determined direction



What are the right values for Delta and Learning Rate (training hyper-parameters)

Gradient

- We chose `delta` equal to 0.1 in the previous section, but it all depends on the shape of the loss as a function of `w` and `b`.
  - If the loss changes too quickly compared to `delta`, we won't have a very good idea of in which direction the loss is decreasing the most.
- What if we could make the neighborhood infinitesimally small?
- That's what we are seeking to achieve!
- In a model with two or more parameters, we compute the individual derivatives of the loss with respect to each parameter and put them in a vector of derivatives: which we call it the *gradient*.



Calculative the derivatives and Gradient

- We want to calculate the derivative of loss function over derivative of w (dw) (in genera parameters)

d loss_fn / d w

- It is computed through derivative of the loss with respect to its input (which is the output of the model – predicted values), times the derivative of the the output of the model with respect to the derivates of the parameter (w):

d loss_fn / d w = (d loss_fn / d t_p) * (d t_p / d w)

loss $L(m_{w,b}(x))$

$$\nabla_{w,b} L = \left(\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b}\right) = \left(\frac{\partial L}{\partial m}\cdot\frac{\partial m}{\partial w}, \frac{\partial L}{\partial m}\cdot\frac{\partial m}{\partial b}\right)$$

gradient

partial derivatives

model $m_{w,b}(x)$

parameters

Calculative the derivatives and Gradient

```
# In[3]:
def model(t_u, w, b):
    return w * t_u + b
```

```
# In[4]:
def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()
```

Remembering that `d x^2 / d x = 2 x`, we get

```
# In[11]:
def dloss_fn(t_p, t_c):
    dsq_diffs = 2 * (t_p - t_c) / t_p.size(0)
    return dsq_diffs
```

```
# In[12]:
def dmodel_dw(t_u, w, b):
    return t_u
```

```
# In[13]:
def dmodel_db(t_u, w, b):
    return 1.0
```

loss $L(m_{w,b}(x))$

$$\nabla_{w,b} L = \left(\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b}\right) = \left(\frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial w}, \frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial b}\right)$$

gradient

partial derivatives

model $m_{w,b}(x)$

parameters

Putting everything together

```
# In[14]:
def grad_fn(t_u, t_c, t_p, w, b):
    dloss_dtp = dloss_fn(t_p, t_c)
    dloss_dw = dloss_dtp * dmodel_dw(t_u, w, b)
    dloss_db = dloss_dtp * dmodel_db(t_u, w, b)
    return torch.stack([dloss_dw.sum(), dloss_db.sum()])
```

The summation is the reverse of the broadcasting we implicitly do when applying the parameters to an entire vector of inputs in the model.

loss $L(m_{w,b}(x))$

$$\nabla_{w,b} L = \left(\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b}\right) = \left(\frac{\partial L}{\partial m}\cdot\frac{\partial m}{\partial w}, \frac{\partial L}{\partial m}\cdot\frac{\partial m}{\partial b}\right)$$

gradient

partial derivatives

model $m_{w,b}(x)$

parameters

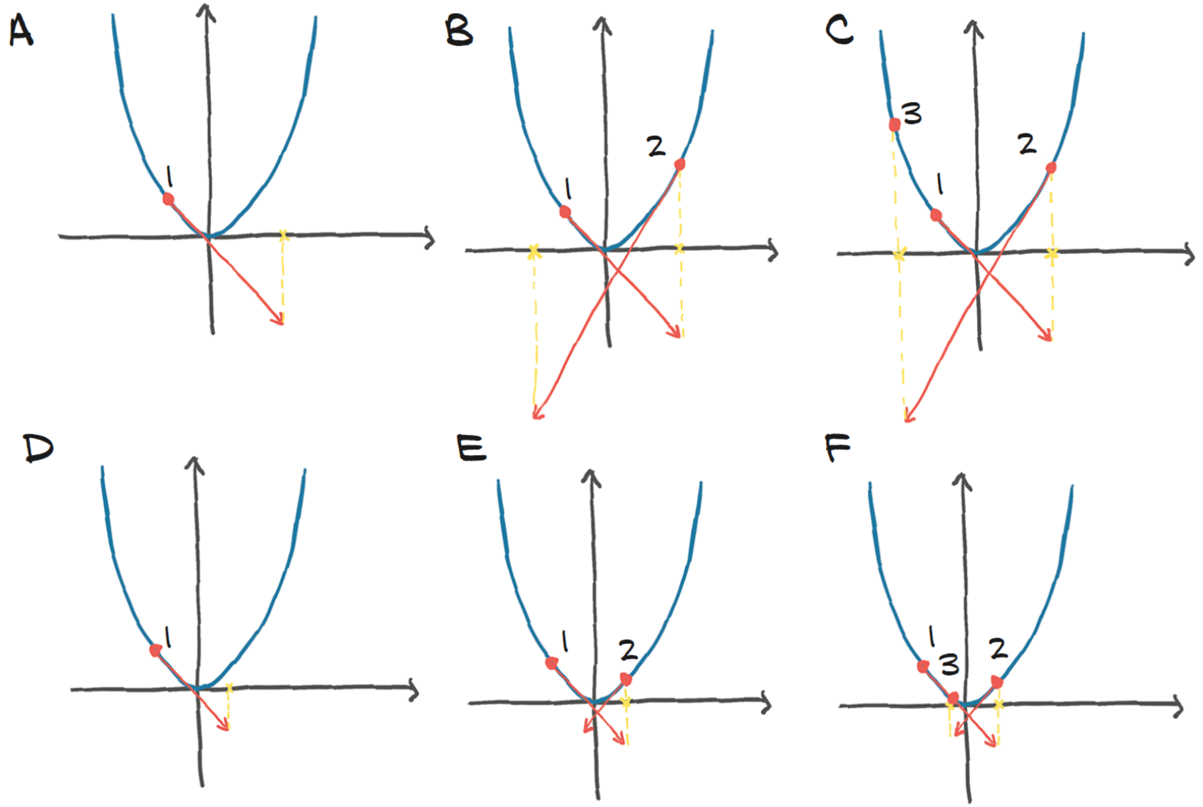
Note: Remember the old constant delta values,
The gradient replace the Delta, which is

Iterating to Fit the Model

```
# In[15]:
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        w, b = params

        t_p = model(t_u, w, b)                      <-- Forward pass
        loss = loss_fn(t_p, t_c)
        grad = grad_fn(t_u, t_c, t_p, w, b)         <-- Backward pass

        params = params - learning_rate * grad

        print('Epoch %d, Loss %f' % (epoch, float(loss)))  <-- This logging line can be very verbose.

    return params
```

We call a training iteration during which we update the parameters for all of our training samples an ***epoch***.

How about Learning Rate?



Too high learning rate

Ideal learning rate

How can we limit the magnitude of `learning_rate * grad`?

Finding the right learning rate

```
In[17]:
training_loop(
n_epochs = 100,
learning_rate = 1e-2,
params = torch.tensor([1.0, 0.0]),
t_u = t_u,
t_c = t_c)

In[17]:
training_loop(
n_epochs = 100,
learning_rate = 1e-4,
params = torch.tensor([1.0, 0.0]),
t_u = t_u,
t_c = t_c)
```

Output when `learning_rate = 1e-2`

```
# Out[17]:
Epoch 1, Loss 1763.884644
    Params: tensor([-44.1730,  -0.8260])
    Grad:   tensor([4517.2969,   82.6000])
Epoch 2, Loss 5802485.500000
    Params: tensor([2568.4014,   45.1637])
    Grad:   tensor([-261257.4219,   -4598.9712])
Epoch 3, Loss 19408035840.000000
    Params: tensor([-148527.7344,   -2616.3933])
    Grad:   tensor([15109614.0000,   266155.7188])
...
Epoch 10, Loss 90901154706620645225508955521810432.000000
    Params: tensor([3.2144e+17, 5.6621e+15])
    Grad:   tensor([-3.2700e+19, -5.7600e+17])
Epoch 11, Loss inf
    Params: tensor([-1.8590e+19, -3.2746e+17])
    Grad:   tensor([1.8912e+21, 3.3313e+19])

tensor([-1.8590e+19, -3.2746e+17])
```

This is a clear sign that `params` is receiving updates that are too large, and their values start oscillating back and forth as each update overshoots and the next overcorrects even more.

Output when `learning_rate = 1e-4`

```
# Out[18]:
Epoch 1, Loss 1763.884644
    Params: tensor([ 0.5483, -0.0083])
    Grad:   tensor([4517.2969,   82.6000])
Epoch 2, Loss 323.090546
    Params: tensor([ 0.3623, -0.0118])
    Grad:   tensor([1859.5493,   35.7843])
Epoch 3, Loss 78.929634
    Params: tensor([ 0.2858, -0.0135])
    Grad:   tensor([765.4667,  16.5122])
...
Epoch 10, Loss 29.105242
    Params: tensor([ 0.2324, -0.0166])
    Grad:   tensor([1.4803, 3.0544])
Epoch 11, Loss 29.104168
    Params: tensor([ 0.2323, -0.0169])
    Grad:   tensor([0.5781, 3.0384])
...
Epoch 99, Loss 29.023582
    Params: tensor([ 0.2327, -0.0435])
    Grad:   tensor([-0.0533,  3.0226])
Epoch 100, Loss 29.022669
    Params: tensor([ 0.2327, -0.0438])
    Grad:   tensor([-0.0532,  3.0226])

tensor([ 0.2327, -0.0438])
```

The behavior is now stable. But there's another problem: the updates to parameters are very small, so the loss decreases very slowly and eventually stalls.

We can see that the first-epoch gradient for the weight is about 50 times larger than the gradient for the bias. This means the weight and bias live in differently scaled spaces.

Normalizing the Inputs

- Changing the inputs so that the gradients aren't quite so different. We can make sure the range of the input doesn't get too far from the range of −1.0 to 1.0, roughly speaking.

```
# In[19]:
t_un = 0.1 * t_u

# In[20]:
training_loop(
n_epochs = 100,
learning_rate = 1e-2,
params = torch.tensor([1.0, 0.0]),
t_u = t_un,
t_c = t_c)
```

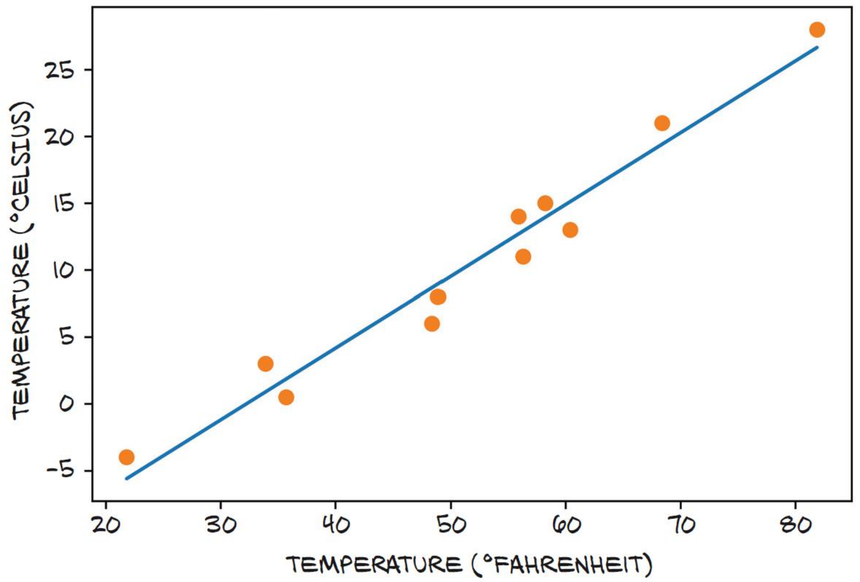
```
# Out[20]:
Epoch 1, Loss 80.364342
    Params: tensor([1.7761, 0.1064])
    Grad:   tensor([-77.6140, -10.6400])
Epoch 2, Loss 37.574917
    Params: tensor([2.0848, 0.1303])
    Grad:   tensor([-30.8623,  -2.3864])
Epoch 3, Loss 30.871077
    Params: tensor([2.2094, 0.1217])
    Grad:   tensor([-12.4631,   0.8587])
...
Epoch 10, Loss 29.030487
    Params: tensor([ 2.3232, -0.0710])
    Grad:   tensor([-0.5355,  2.9295])
Epoch 11, Loss 28.941875
    Params: tensor([ 2.3284, -0.1003])
    Grad:   tensor([-0.5240,  2.9264])
...
Epoch 99, Loss 22.214186
    Params: tensor([ 2.7508, -2.4910])
    Grad:   tensor([-0.4453,  2.5208])
Epoch 100, Loss 22.148710
    Params: tensor([ 2.7553, -2.5162])
    Grad:   tensor([-0.4446,  2.5165])
```

Let's do it for much larger number of epochs

```
# In[21]:
params = training_loop(
n_epochs = 5000,
learning_rate = 1e-2,
params = torch.tensor([1.0, 0.0]),
t_u = t_un,
t_c = t_c,
print_params = False)
params
```

```
# Out[21]:
Epoch 1, Loss 80.364342
Epoch 2, Loss 37.574917
Epoch 3, Loss 30.871077
...
Epoch 10, Loss 29.030487
Epoch 11, Loss 28.941875
...
Epoch 99, Loss 22.214186
Epoch 100, Loss 22.148710
...
Epoch 4000, Loss 2.927680
Epoch 5000, Loss 2.927648

tensor([  5.3671, -17.3012])
```

Now Visualizing the Outcome




```
# In[22]:
%matplotlib inline
from matplotlib import pyplot as plt

t_p = model(t_un, *params)

fig = plt.figure(dpi=600)
plt.xlabel("Temperature (°Fahrenheit)")
plt.ylabel("Temperature (°Celsius)")
plt.plot(t_u.numpy(), t_p.detach().numpy())
plt.plot(t_u.numpy(), t_c.numpy(), 'o')
```

Remember that we're training on the normalized unknown units. We also use argument unpacking.

But we're plotting the raw unknown values.

Backpropagation

- In our little adventure, we just saw a simple example of backpropagation
  - We computed the gradient of a composition of functions—the model and the loss—with respect to their innermost parameters (`w` and `b`) by propagating derivatives backward using the *chain rule*.

- The basic requirement here is that all functions we're dealing with can be differentiated analytically.
  - We can compute the gradient— what we earlier called "the rate of change of the loss"—with respect to the parameters in one sweep.

- Even if we have a complicated model with millions of parameters, as long as our model is differentiable, computing the gradient of the loss with respect to the parameters amounts

Autograd: Computing the gradient automatically

- This is when PyTorch tensors come to the rescue, with a PyTorch component called *Autograd*.

```
# In[3]:
def model(t_u, w, b):
return w * t_u + b

# In[4]:
def loss_fn(t_p, t_c):
squared_diffs = (t_p - t_c)**2
return squared_diffs.mean()

# In[5]:
params = torch.tensor([1.0, 0.0],
requires_grad=True)
```

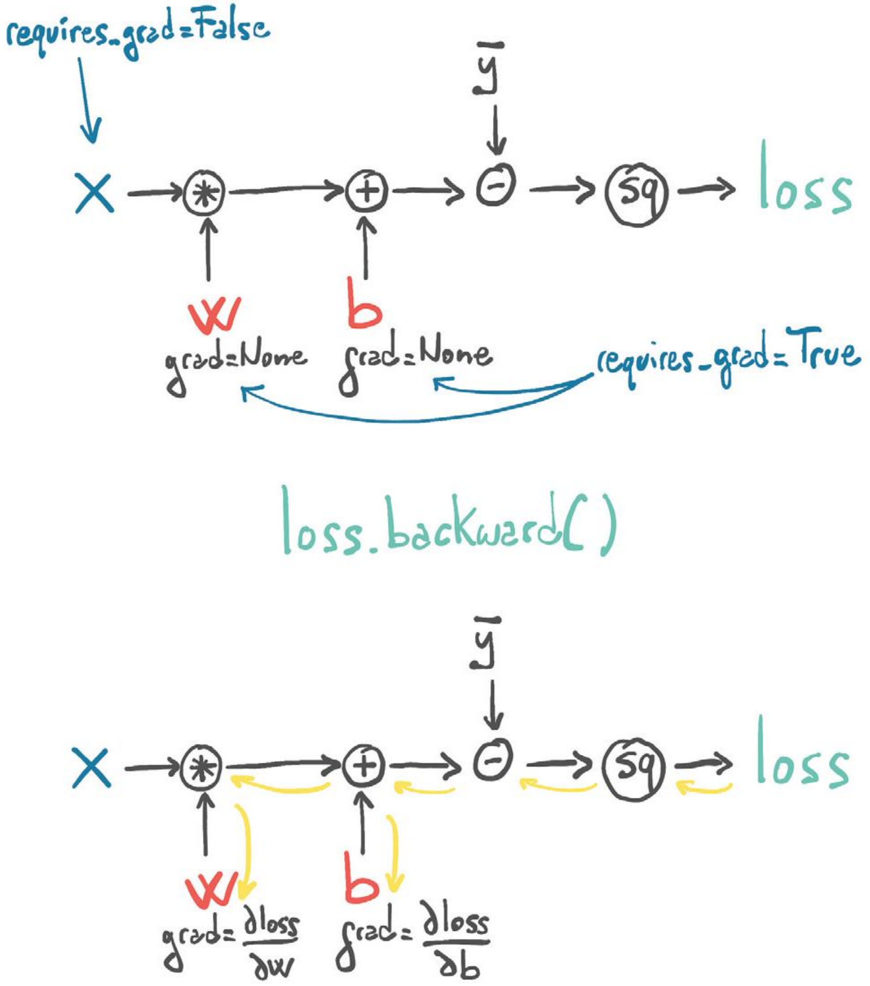
Autograd

- `requires_grad=True` is telling PyTorch to track the entire family tree of tensors resulting from operations on all `params` involved in the model.
- All we have to do is to start with `params` tensor with `requires_grad` set to `True`, then call the model and compute the loss, and then call `backward` on the `loss` tensor:

```
# In[7]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
loss = loss_fn(model(t_u, *params), t_c)
loss.backward()
params.grad
# Out[7]:
tensor([4517.2969, 82.6000])
```

At this point, the `grad` attribute of `params` contains the derivatives of the loss with respect to each element of params.

Autograd and Forward and Backward Graphs



- **Forward:** when we compute our `loss` while the parameters `w` and `b` require gradients, in addition to performing the actual computation, PyTorch creates the autograd graph with the operations (in black circles) as nodes.
- **Backward:** when we call `loss.backward()`, PyTorch traverses this graph in the reverse direction to compute the gradients, as shown by the arrows

Putting Everything together with Autograd

```
# In[9]:
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        with torch.no_grad():
            params -= learning_rate * params.grad

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

This could be done at any point in the loop prior to calling loss.backward().

This is a somewhat cumbersome bit of code, but as we'll see in the next section, it's not an issue in practice.

Calling `backward` will lead derivatives to *accumulate* at leaf nodes. We need to *zero the gradient explicitly* per each iteration of training.

Putting Everything together with Autograd

- We are encapsulating the parameters update in a `no_grad` context using the Python `with` statement. This means within the `with` block, the PyTorch autograd mechanism will not be applied.

```
# In[9]:
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        with torch.no_grad():
            params -= learning_rate * params.grad

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

This could be done at any point in the loop prior to calling loss.backward().

This is a somewhat cumbersome bit of code, but as we'll see in the next section, it's not an issue in practice.

Putting Everything together with Autograd

```
# In[10]:
training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0], requires_grad=True),
    t_u = t_un,
    t_c = t_c)
```

Adding requires_grad=True is key.

Again, we're using the normalized t_un instead of t_u.

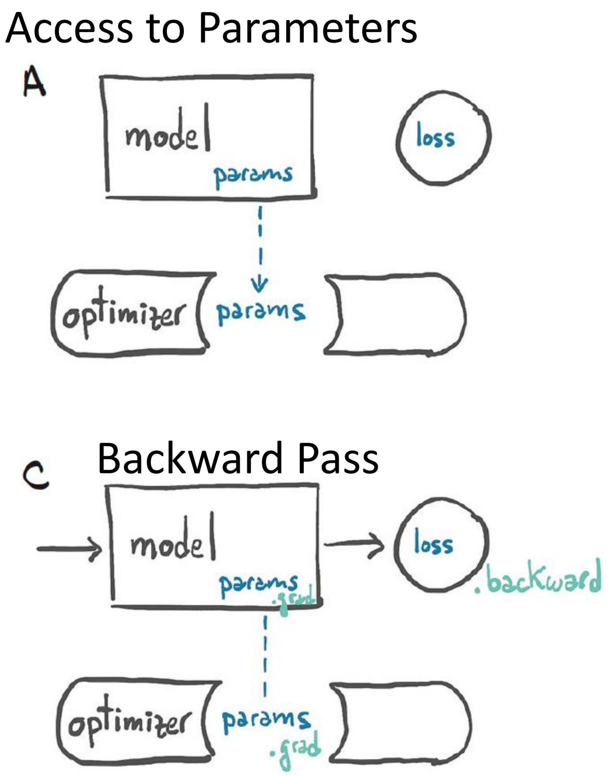
```
# Out[10]:
Epoch 500, Loss 7.860116
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927679
Epoch 4500, Loss 2.927652
Epoch 5000, Loss 2.927647

tensor([  5.3671, -17.3012], requires_grad=True)
```

NOTE: We get the same result as before

Gradient Decent Optimizer

- In the example code, we used ***vanilla* gradient descent** for optimization, which worked fine for our simple case.
- Here, **vanilla** means pure / without any adulteration.
- Its main feature is that we take small steps in the direction of the minima by taking **gradient** of the cost function.
- This is the simplest form of **gradient descent** technique. For Complex Models with many parameters, more complex gradient decent optimizers can be used.

Gradient Decent Optimizer



- Every optimizer constructor takes a list of parameters (aka PyTorch tensors, typically with `requires_grad` set to `True`) as the first input.

- All parameters passed to the optimizer are retained inside the optimizer object so the optimizer can update their values and access their `grad` attribute.

Gradient Decent Optimizer

- The `torch` module has an `optim` submodule where we can find classes implementing different optimization algorithms. Here's an abridged list (code/p1ch5/3_optimizers.ipynb):

```
# In[5]:
import torch.optim as optim
dir(optim)
# Out[5]:
['ASGD',
'Adadelta',
'Adagrad',
'Adam',
'Adamax',
'LBFGS',
'Optimizer',
'RMSprop',
'Rprop',
'SGD',
'SparseAdam',
...
]
```

- Each optimizer exposes two methods: `zero_grad` and `step`.
- **`zero_grad`** zeroes the `grad` attribute of all the parameters passed to the optimizer upon construction.
- **`step`** updates the value of those parameters according to the optimization strategy implemented by the specific optimizer.

Stochastic Gradient Descent (SGD)

- The term *stochastic* comes from the fact that the gradient is typically obtained by averaging over a random subset of all input samples, called a *minibatch.*
- Actually, the optimizer itself is exactly a vanilla gradient descent (as long as the `momentum` argument is set to `0.0`, which is the default).
- The algorithm is literally the same in the two cases.
- vanilla is evaluated on all the samples

```
# In[7]:
t_p = model(t_u, *params)
loss = loss_fn(t_p, t_c)
loss.backward()
optimizer.step()
params
# Out[7]:
tensor([ 9.5483e-01, -8.2600e-04],
requires_grad=True)
```

- The value of `params` is updated upon calling `step`.
  - The optimizer looks into `params.grad` and updates `params`, subtracting `learning_rate` times `grad` from it, exactly as in our former handwrittedn code.

Putting Everything together with Optimizer

```
# In[8]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

t_p = model(t_un, *params)
loss = loss_fn(t_p, t_c)

optimizer.zero_grad()
loss.backward()
optimizer.step()

params

# Out[8]:
tensor([1.7761, 0.1064], requires_grad=True)
```

As before, the exact placement of this call is somewhat arbitrary. It could be earlier in the loop as well.

- All we have to do is provide a list of params to it (that list can be extremely long, as is needed for very deep neural network models), and we can forget about the details.

Putting Everything together with Optimizer

```
# In[9]:
def training_loop(n_epochs, optimizer, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params

# In[10]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    params = params,
    t_u = t_un,
    t_c = t_c)
```

It's important that both params are the same object; otherwise the optimizer won't know what parameters were used by the model.

```
# Out[10]:
Epoch 500, Loss 7.860118
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927680
Epoch 4500, Loss 2.927651
Epoch 5000, Loss 2.927648

tensor([  5.3671, -17.3012], requires_grad=True)
```

We get the same result as before

Adam Optimizer

- Adam Optimizer is a more sophisticated optimizer in which the learning rate is set adaptively.
- In addition, it is a lot less sensitive to the scaling of the parameter.

```
# In[11]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-1
optimizer = optim.Adam([params], lr=learning_rate)     <-- New optimizer class

training_loop(
    n_epochs = 2000,
    optimizer = optimizer,
    params = params,
    t_u = t_u,        <-- We're back to the original
    t_c = t_c)            t_u as our input.

# Out[11]:
Epoch 500, Loss 7.612903
Epoch 1000, Loss 3.086700
Epoch 1500, Loss 2.928578
Epoch 2000, Loss 2.927646

tensor([  0.5367, -17.3021], requires_grad=True)
```

We can go back to using the original (non-normalized) input `t_u`, and even increase the learning rate to `1e-1`, and Adam won't even blink.