Deep Learning

Mohammad Reza Mohammadi

2021

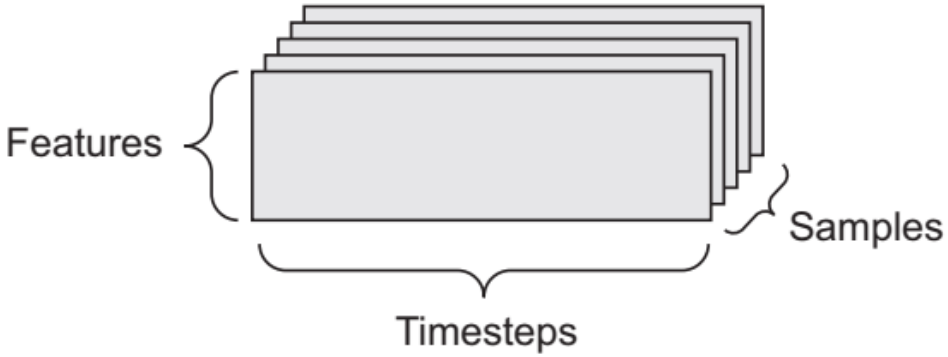
Real-world examples of data tensors

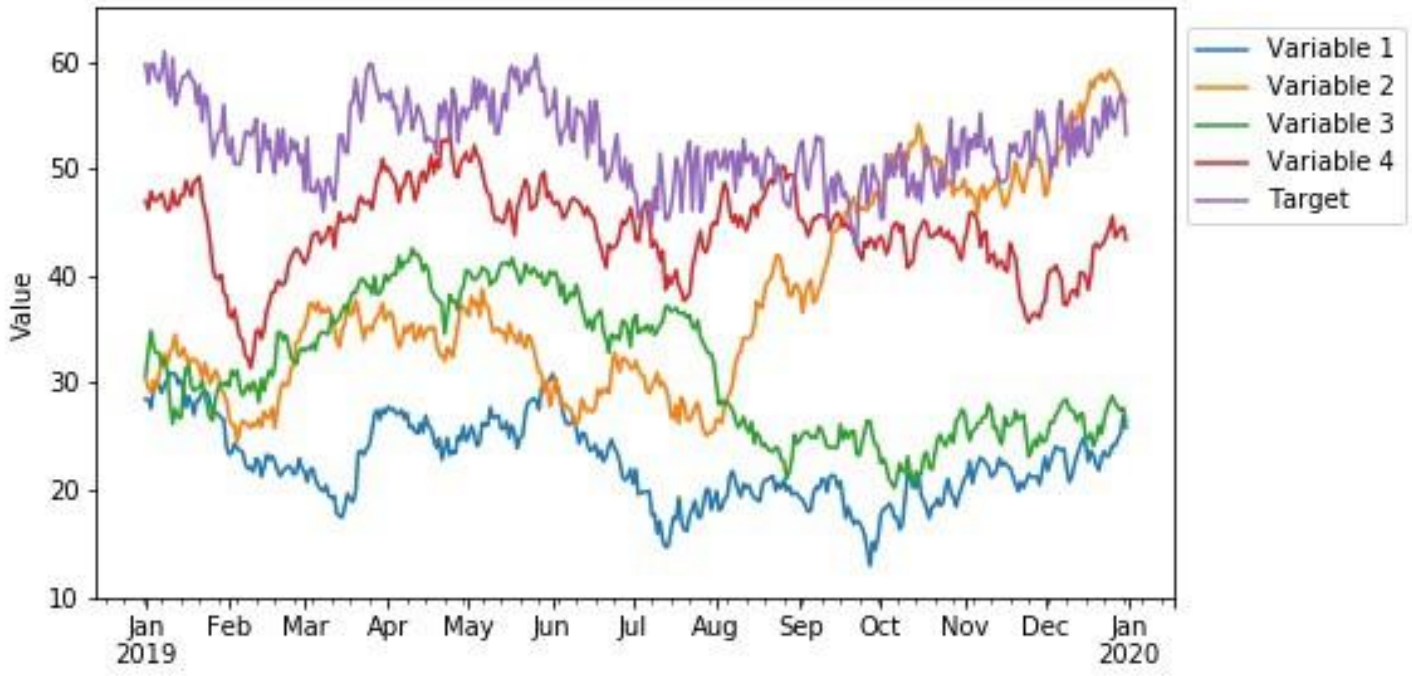
- Vector data - 2D tensors of shape (samples, features)

date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sqft_basement	yr_built	yr_renovated
20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	...	7	1180	0	1955	0
20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	...	7	2170	400	1951	1991
20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	...	6	770	0	1933	0
20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	0	...	7	1050	910	1965	0
20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	0	...	8	1680	0	1987	0

Real-world examples of data tensors

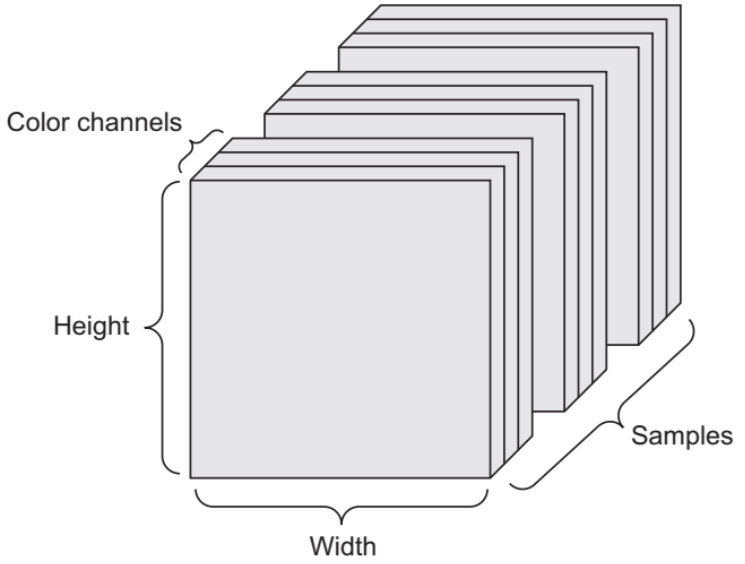
- Vector data - 2D tensors of shape (samples, features)
- Timeseries data or sequence data - 3D tensors of shape (samples, timesteps, features)





Real-world examples of data tensors

- Vector data - 2D tensors of shape (samples, features)
- Timeseries data or sequence data - 3D tensors of shape (samples, timesteps, features)
- Images - 4D tensors of shape (samples, height, width, channels) or (samples, channels, height, width)



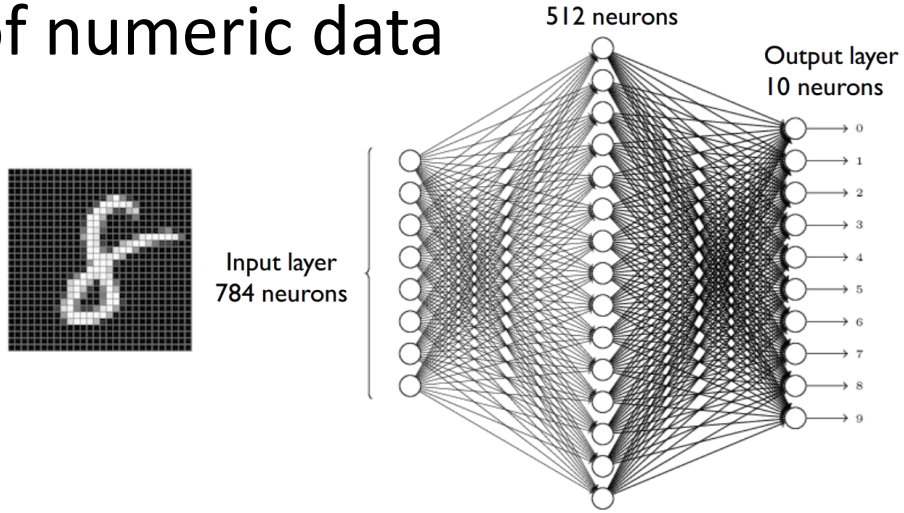
Real-world examples of data tensors

- Vector data - 2D tensors of shape (samples, features)
- Timeseries data or sequence data - 3D tensors of shape (samples, timesteps, features)
- Images - 4D tensors of shape (samples, height, width, channels) or (samples, channels, height, width)
- Video - 5D tensors of shape (samples, frames, height, width, channels) or(samples, frames, channels, height, width)

Tensor operations

- All transformations learned by deep neural networks can be reduced to a handful of tensor operations applied to tensors of numeric data
- From our example:

```
keras.layers.Dense(512, activation='relu')
```

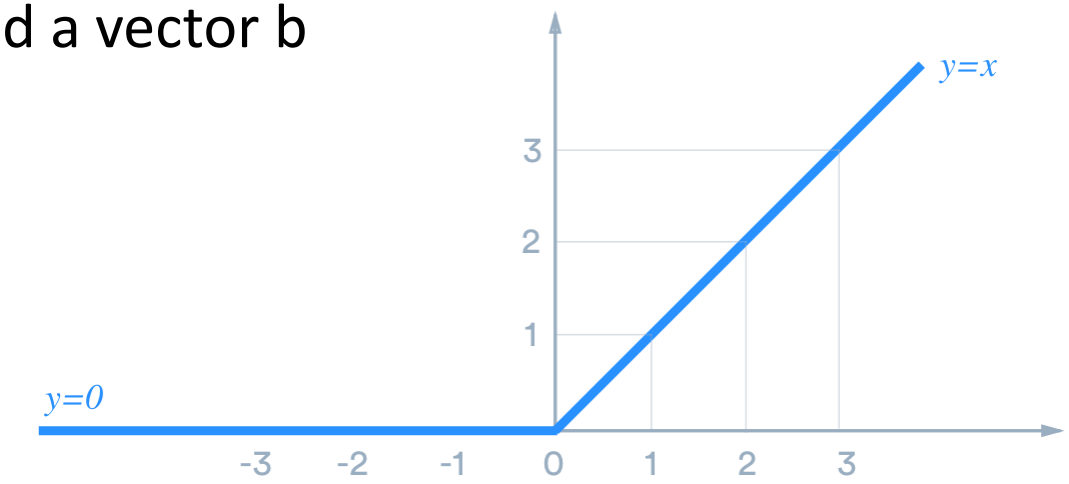


- This layer can be interpreted as a function, which takes as input a 2D tensor and returns another 2D tensor:

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

Tensor operations

- We have three operations:
  - Dot product (dot) between input and W tensors
  - An addition between the resulting 2D tensor and a vector b
  - And finally a relu operation:
    - relu(x) = max(x,0)




```
keras.layers.Dense(512, activation='relu')
```

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

Element-wise operations

- Addition and relu are element-wise operations
  - Highly parallelizable

```
import numpy as np

z = x + y                  <-- Element-wise addition
z = np.maximum(z, 0.)      <-- Element-wise relu
```

```
def naive_relu(x):
    assert len(x.shape) == 2          <-- x is a 2D Numpy tensor.

    x = x.copy()                      <-- Avoid overwriting the input tensor.
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

```
def naive_add(x, y):
    assert len(x.shape) == 2          <-- x and y are 2D Numpy tensors.
    assert x.shape == y.shape

    x = x.copy()                      <-- Avoid overwriting the input tensor.
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

Broadcasting

- In the Dense layer of our example, we added a 2D tensor with a vector. How?
- When possible, and if there's no ambiguity, the smaller tensor will be broadcasted to match the shape of the larger tensor
- Broadcasting consists of two steps:
  - Axes (called broadcast axes) are added to the smaller tensor to match the ndim of the larger tensor
  - The smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor

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

Broadcasting

```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]

    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

x is a 2D Numpy tensor.

y is a Numpy vector.

Avoid overwriting the input tensor.

```
import numpy as np

x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))

z = np.maximum(x, y)
```

x is a random tensor with shape (64, 3, 32, 10).

y is a random tensor with shape (32, 10).

The output z has shape (64, 3, 32, 10) like x.

Tensor dot

- Also called a tensor product
  - (not to be confused with an elementwise product)

```
import numpy as np

z = np.dot(x, y)
```

- In mathematical notation, you'd note the operation with a dot (.)

```
z = x . y
```

- The output is a scalar

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]

    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

x and y are Numpy vectors.

Tensor dot

```
def naive_matrix_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]

    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

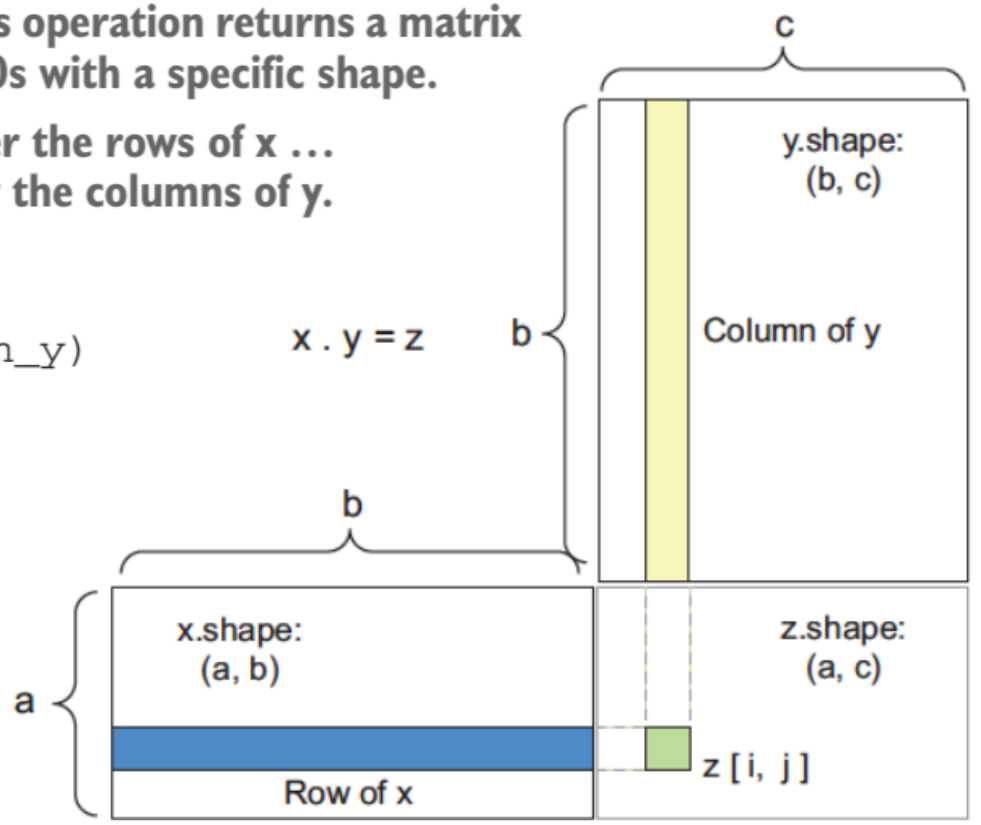
x and y are Numpy matrices.

The first dimension of x must be the same as the 0th dimension of y!

This operation returns a matrix of 0s with a specific shape.

Iterates over the rows of x ...

... and over the columns of y.



Tensor reshaping

- In our example, we reshaped the input matrix using the Flatten layer

```
model.add(keras.layers.Input(shape=x_train[0].shape))
```

- Reshaping a tensor means rearranging its rows and columns to match a target shape
- Naturally, the reshaped tensor has the same total number of coefficients as the initial tensor

```
>>> x = np.array([[0., 1.],
                  [2., 3.],
                  [4., 5.]])
>>> print(x.shape)
(3, 2)
```

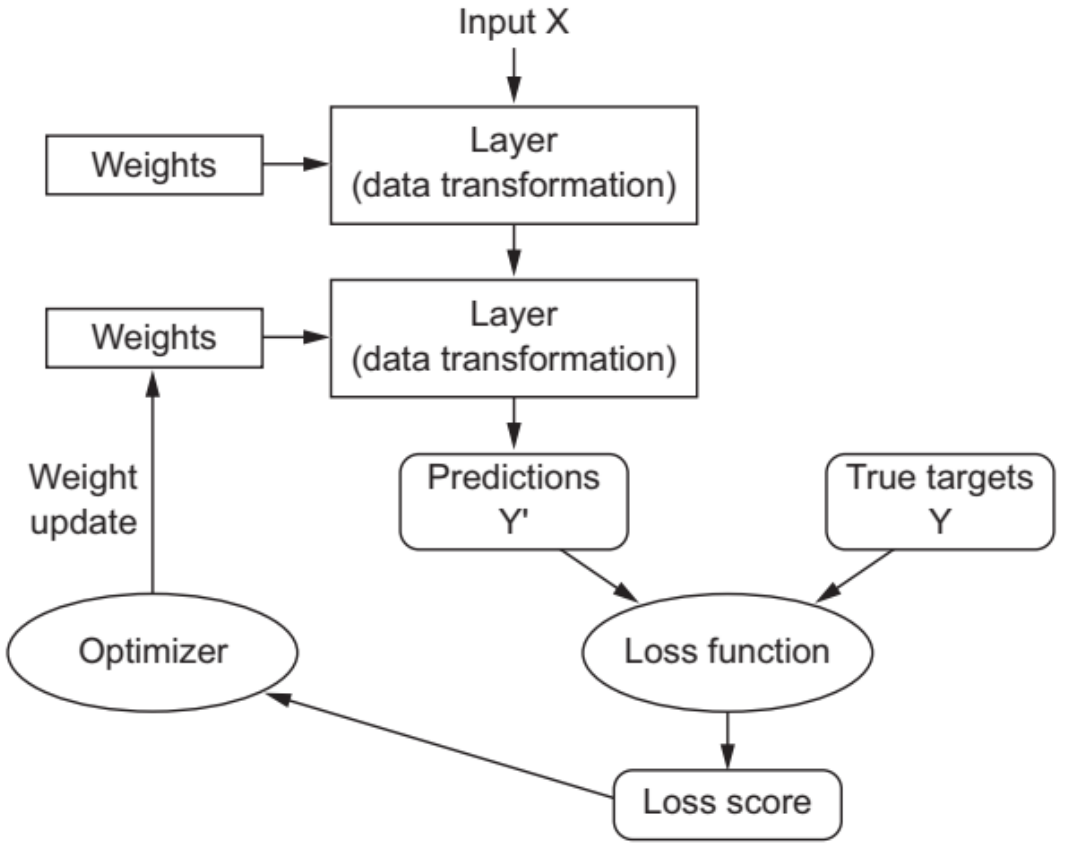
```
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
```

```
>>> x = x.reshape((2, 3))
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
```

Gradient-based optimization

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

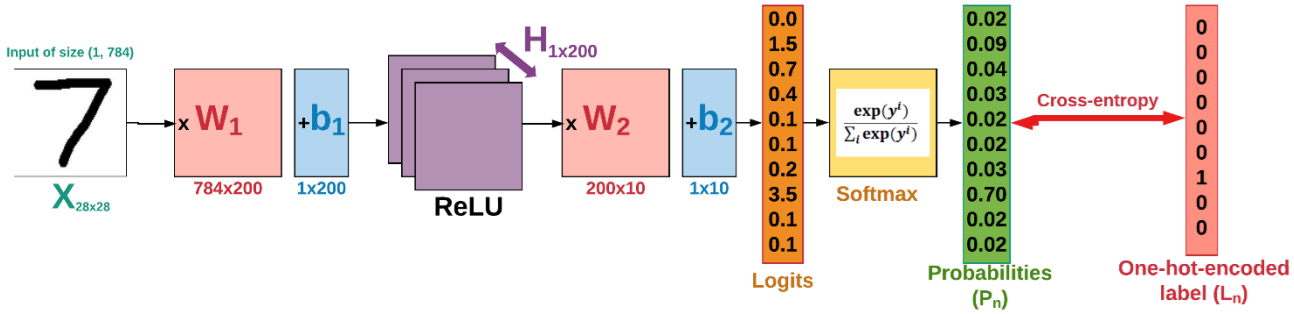
- W and b are weights or trainable parameters of the layer
- These weights contain the information learned by the network from exposure to training data
- Initially, these are filled with random values (random initialization)
- But, we gradually adjust these weights based on the feedback signal



Training loop

- This gradual adjustment, also called training, is basically the learning that machine learning is all about

1. Draw a batch of training samples x and corresponding targets y
2. Run the network on x (a step called the forward pass) to obtain predictions y_pred
3. Compute the loss of the network on the batch, a measure of the mismatch between y_pred and y
4. Update all weights of the network in a way that slightly reduces the loss on this batch



Strategy #1: Random search

```python
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in range(1000):
  W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
  loss = L(X_train, Y_train, W) # get the loss over the entire training set
  if loss < bestloss: # keep track of the best solution
    bestloss = loss
    bestW = W
  print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (trunctated: continues for 1000 lines)
```

- A very bad idea solution!
- 15.5% test accuracy!
- SOTA is ~99.3%

```python
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

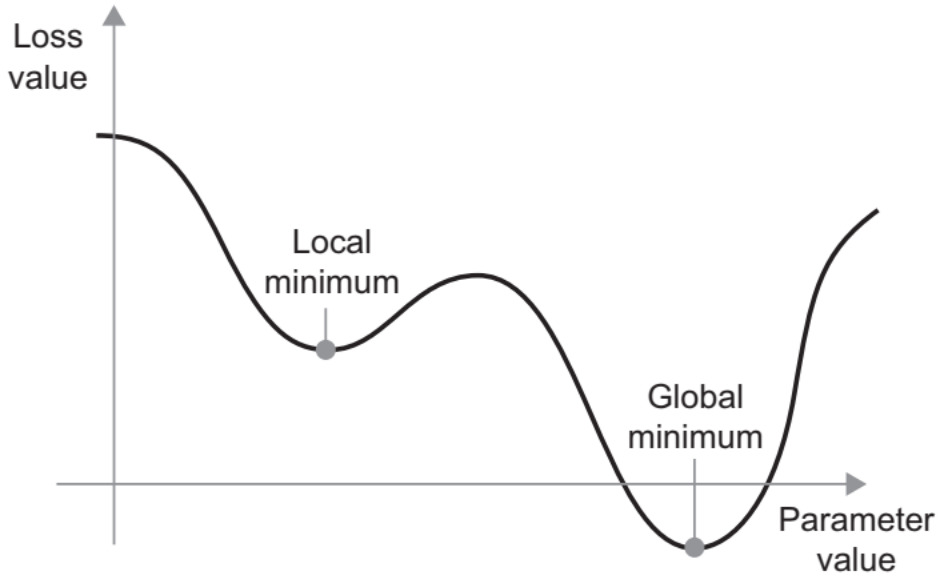
Strategy #2: Random local search

- Try to extend one foot in a random direction and then take a step only if it leads downhill
- This approach achieves test set classification accuracy of 21.4%

```
W = np.random.randn(10, 3073) * 0.001 # generate random starting W
bestloss = float("inf")
for i in range(1000):
  step_size = 0.0001
  Wtry = W + np.random.randn(10, 3073) * step_size
  loss = L(Xtr_cols, Ytr, Wtry)
  if loss < bestloss:
    W = Wtry
    bestloss = loss
  print 'iter %d loss is %f' % (i, bestloss)
```

Strategy #3: Follow the slope

- In 1-dimension, the derivative of a function: $\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$
- In multiple dimensions, the gradient is the vector of (partial derivatives) along each dimension
- The slope in any direction is the dot product of the direction with the gradient
- The direction of steepest descent is the negative gradient



Strategy #3: Follow the slope

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

Strategy #3: Follow the slope

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (first dim)**:**

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25322

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

Strategy #3: Follow the slope

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (first dim)**:**

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25322

gradient dW:

[**-2.5**,
?,
?,

(1.25322 - 1.25347)/0.0001
= -2.5

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

?,
?,…]

Strategy #3: Follow the slope

current W:	**W + h** (second dim)**:**	**gradient dW:**
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…]	[0.34, -1.11 + **0.0001**, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…]	[-2.5, ?, ?, ?, ?, ?, ?, ?, ?,…]
loss 1.25347	**loss 1.25353**	

Strategy #3: Follow the slope

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (second dim)**:**

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25353

gradient dW:

[-2.5,
0.6,
?,
?,

(1.25353 - 1.25347)/0.0001
= 0.6

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

?,…]

Strategy #3: Follow the slope

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[-2.5,
0.6,
?,
?,
?,
?,
?,
?,
?,…]

Strategy #3: Follow the slope

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

W + h (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?,…]

(1.25347 - 1.25347)/0.0001
= 0

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Strategy #3: Follow the slope

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

W + h (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?,…]

Numeric Gradient
- Slow! Need to loop over all dimensions
- Approximate

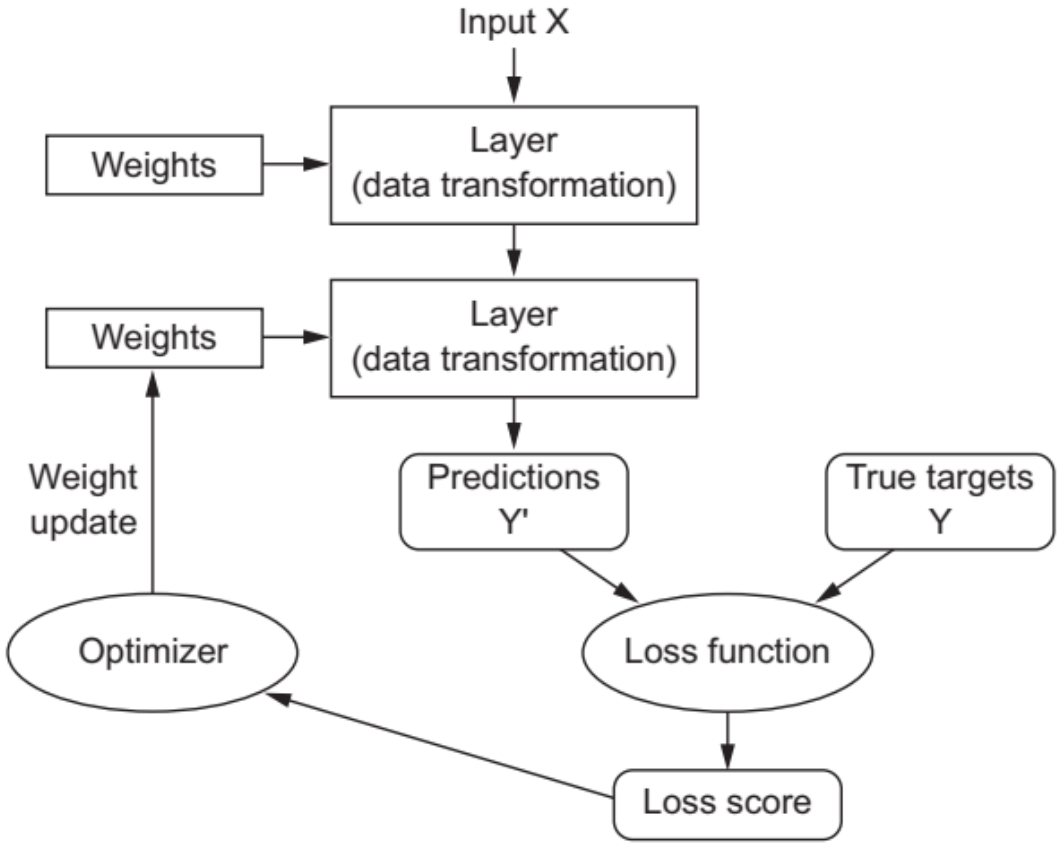
Strategy #3: Follow the slope

- Use calculus to compute an analytic gradient!

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i(s_i, y_i)$$

$$s_i = f(x_i, W) = W\, x_i$$

- We want $\nabla_W L$
  - Is a function of data and W



Strategy #3: Follow the slope

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

dW = ...
(some function
data and W)

gradient dW:

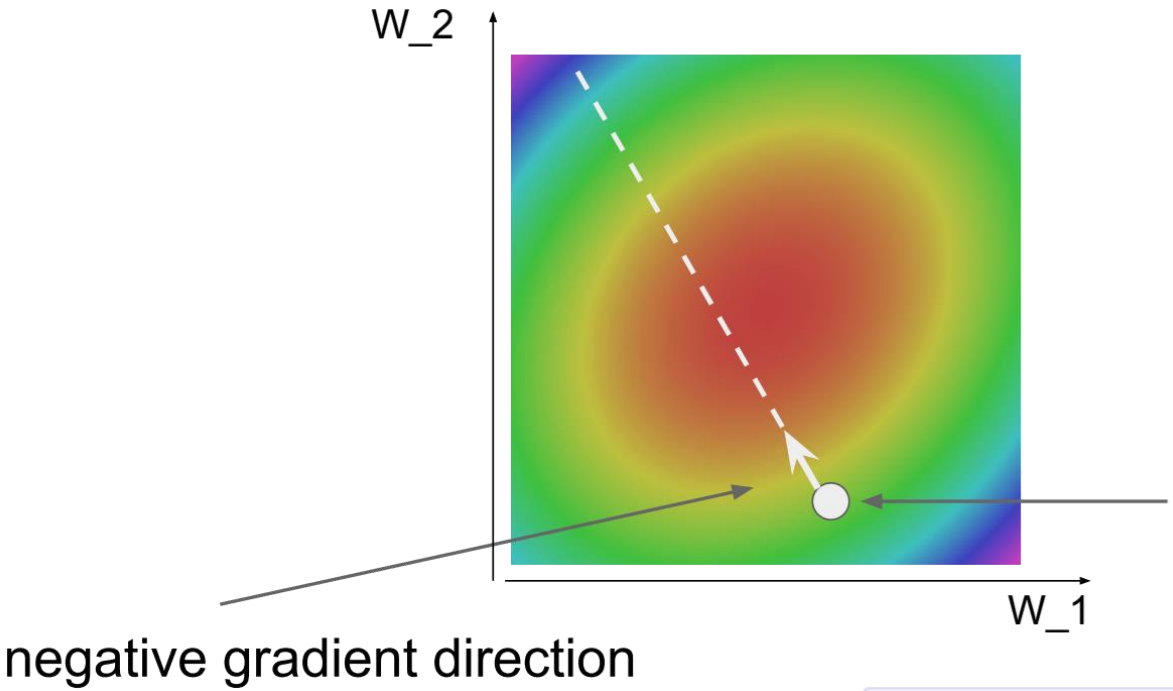
[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,…]

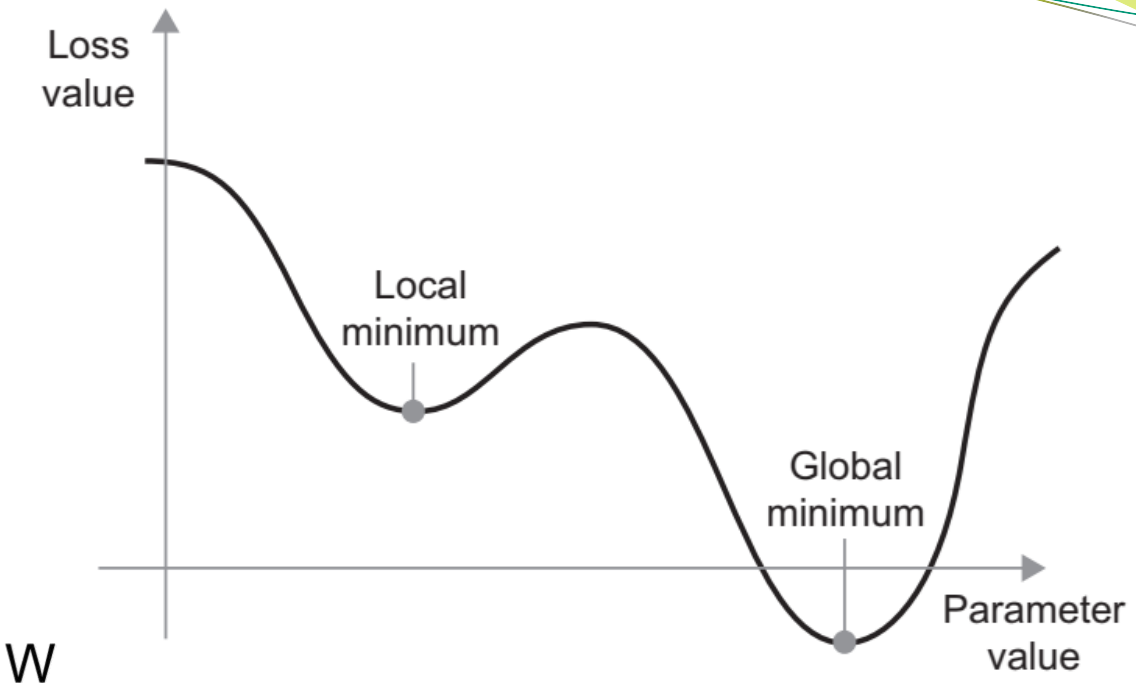
Strategy #3: Follow the slope

- In summary:
  - Numerical gradient: approximate, slow, easy to write
  - Analytic gradient: exact, fast, error-prone

- In practice:
  - Always use analytic gradient
  - But check implementation with numerical gradient
  - This is called a gradient check

Gradient Descent






```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic Gradient Descent (SGD)

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i(s_i, y_i)$$

$$\nabla_W L = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(s_i, y_i)$$

$$s_i = f(x_i, W) = W\, x_i$$

$$W = W - \eta \nabla_W L$$

- Full sum expensive when N is large!
- Approximate sum using a minibatch of examples
  - 32 / 64 / 128 common

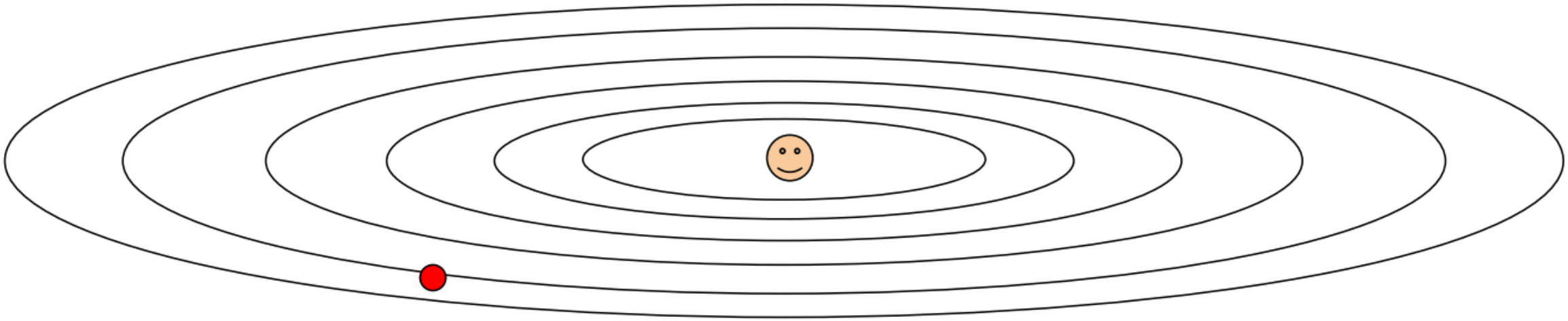
```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

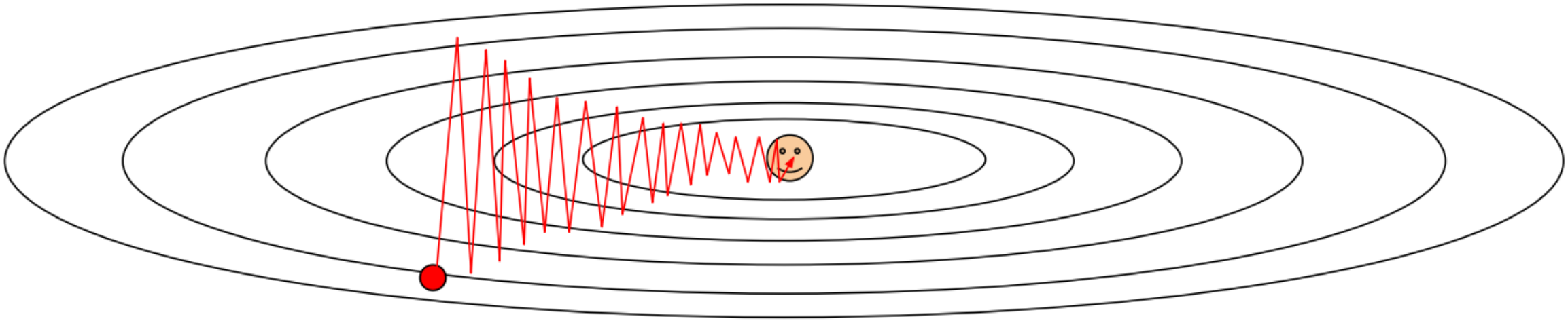
Problems with SGD

- What if loss changes quickly in one direction and slowly in another?
- What does gradient descent do?



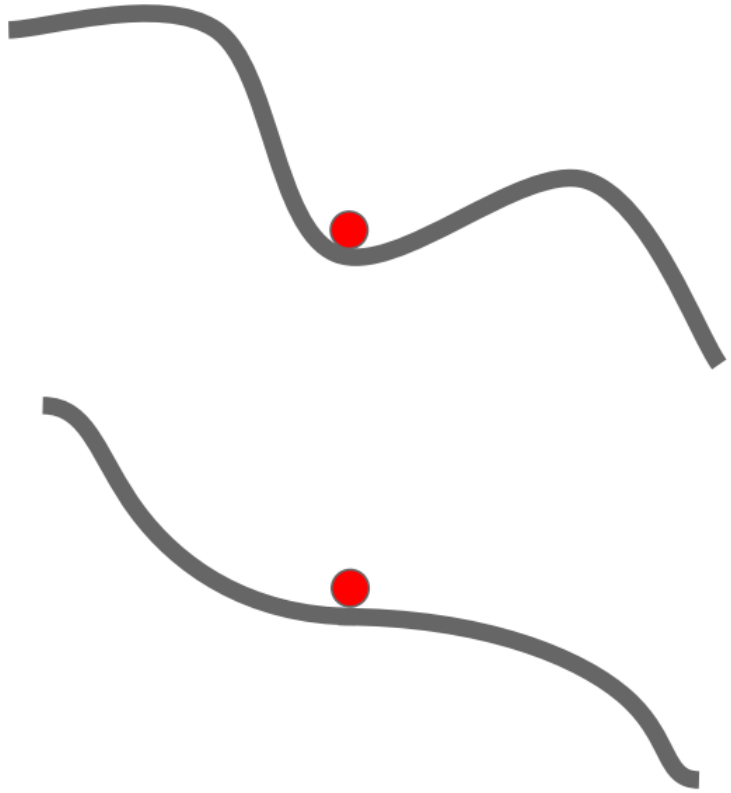
Problems with SGD

- What if loss changes quickly in one direction and slowly in another?
- What does gradient descent do?
- Very slow progress along shallow dimension, jitter along steep direction



Problems with SGD

- What if the loss function has a local minima or saddle point?
- Zero gradient, gradient descent gets stuck

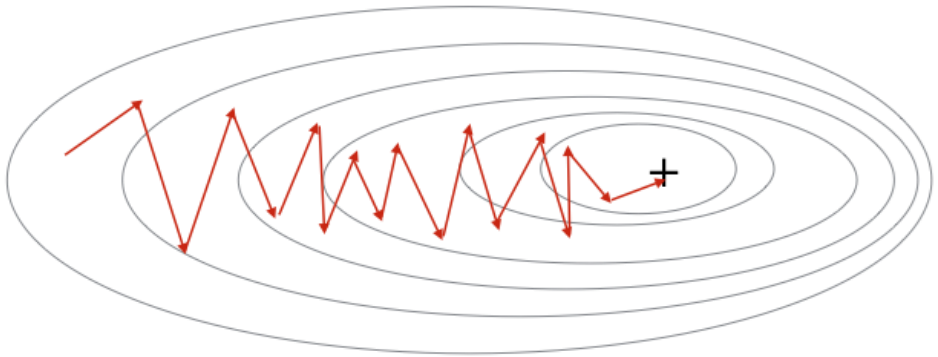


Problems with SGD

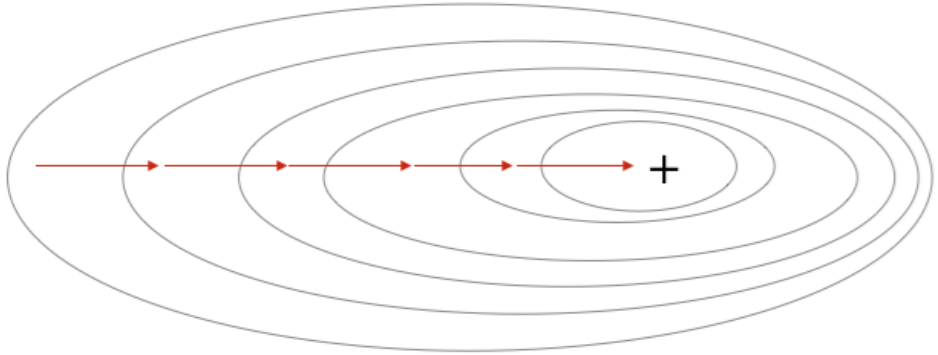
- Our gradients come from minibatches so they can be noisy!
- Gradients are noisy but still make good progress on average

$$\nabla_W L = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(s_i, y_i)$$

Stochastic Gradient Descent



Gradient Descent



SGD + Momentum

- The loss can be interpreted as the height of a hilly terrain
- Initializing the parameters with random numbers is equivalent to setting a particle with zero initial velocity at some location
- The optimization process can then be seen as equivalent to the process of simulating the parameter vector (i.e. a particle) as rolling on the landscape

https://cs231n.github.io/neural-networks-3/

SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD + Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

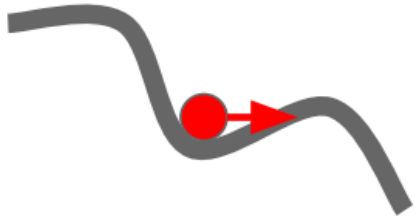
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically $\rho = 0.9$ or 0.99

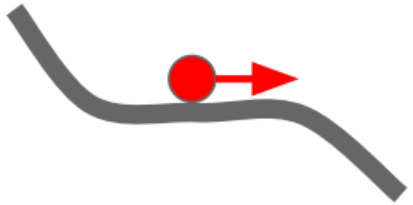
$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

SGD + Momentum

Local Minima



Saddle points



Poor Conditioning