CS163: Deep Learning for Computer Vision

Lecture 4: Optimization Algorithms

Courtesy of Prof. Justin Johnson at Umich and the team of Stanford CS231n for the slides

The Nobel Prize in Physics 2024

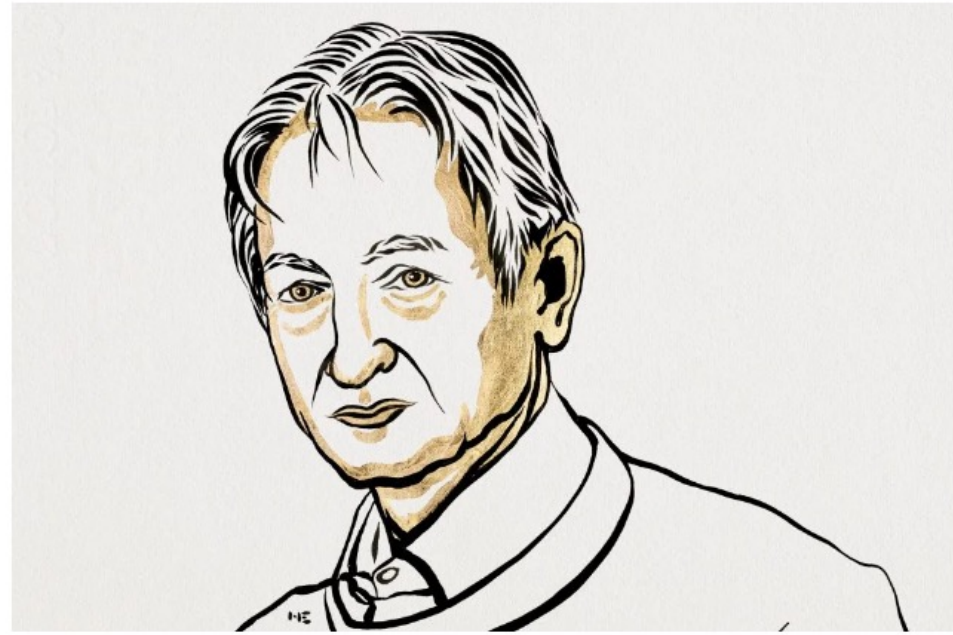
John Hopfield

"for foundational discoveries and inventions that enable machine learning with artificial neural networks"

Ill. Niklas Elmehed © Nobel Prize Outreach

Geoffrey Hinton

"for foundational discoveries and inventions that enable machine learning with artificial neural networks"



Ill. Niklas Elmehed © Nobel Prize Outreach

Scientific Background to the Nobel Prize in Physics 2024 The Nobel Committee for Physics

- Link: https://www.nobelprize.org/uploads/2024/09/advanced-physicsprize2024.pdf
- Links: https://www.nobelprize.org/uploads/2024/10/popular-physicsprize2024.pdf

Introduction

With its roots in the 1940s, machine learning based on artificial neural networks (ANNs) has developed over the past three decades into a versatile and powerful tool, with both everyday and advanced scientific applications. With ANNs the boundaries of physics are extended to host phenomena of life as well as computation.

Inspired by biological neurons in the brain, ANNs are large collections of "neurons", or nodes, connected by "synapses", or weighted couplings, which are trained to perform certain tasks rather than asked to execute a predetermined set of instructions. Their basic structure has close similarities with spin models in statistical physics applied to magnetism or alloy theory. This year's Nobel Prize in Physics recognizes research exploiting this connection to make breakthrough methodological advances in the field of ANN.

Scientific Background to the Nobel Prize in Physics 2024 The Nobel Committee for Physics

- Link: https://www.nobelprize.org/uploads/2024/09/advanced-physicsprize2024.pdf
- Links: https://www.nobelprize.org/uploads/2024/10/popular-physicsprize2024.pdf

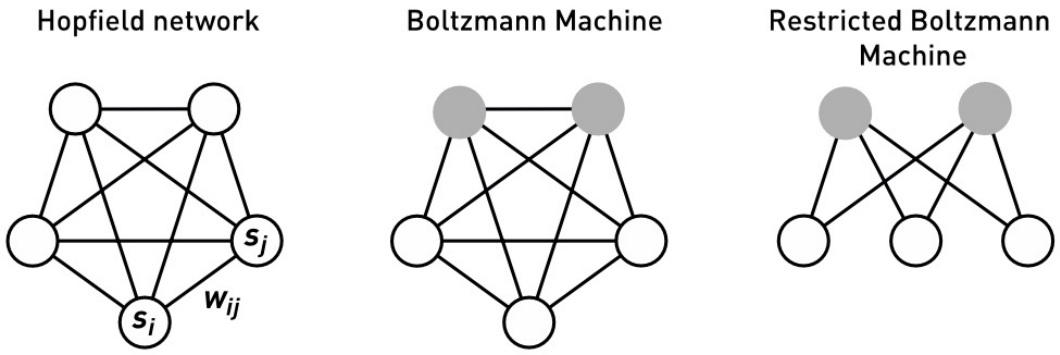


Figure 1. *Recurrent networks of N binary nodes s_i (0 or 1), with connection weights w_{ij}. (Left) The Hopfield model. (Centre) Boltzmann machine. The nodes are divided into two groups, visible (open circles) and hidden (grey) nodes. The network is trained to approximate the probability distribution of a given set of visible patterns. Once trained, the network can be used to generate new instances from the learned distribution. (Right) Restricted Boltzmann Machine (RBM). Same as the Boltzmann machine, but without any couplings within the visible layer or between hidden nodes. This variant can be used for layer-by-layer pre-training of deep networks.*

the network was assigned an energy E given by

$$E = -\sum_{i<j} w_{ij} s_i s_j,$$

By creating and exploring the above physics-based dynamical models – not only the milestone associative memory model but also those that followed – Hopfield made a foundational contribution to our understanding of the computational abilities of neural networks.

Scientific Background to the Nobel Prize in Physics 2024 The Nobel Committee for Physics

- Link: https://www.nobelprize.org/uploads/2024/09/advanced-physicsprize2024.pdf
- Links: https://www.nobelprize.org/uploads/2024/10/popular-physicsprize2024.pdf

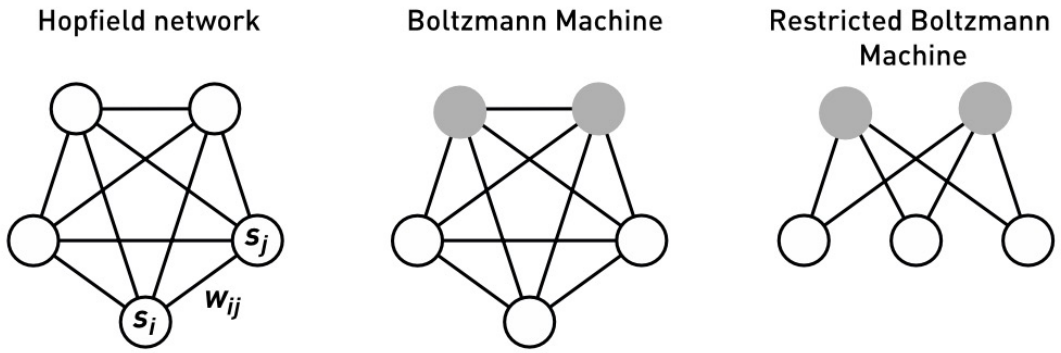


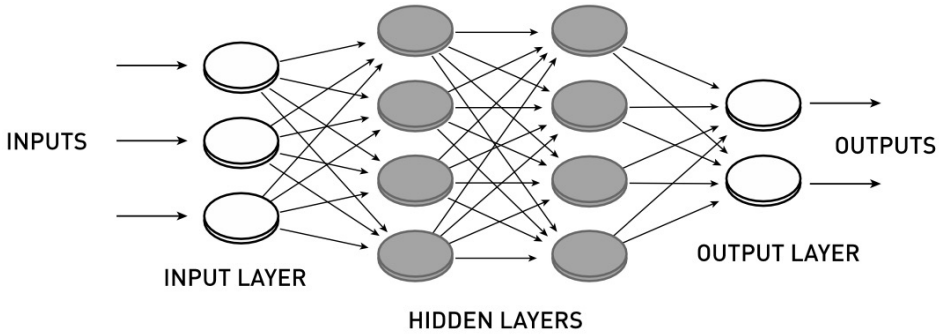
Figure 1. *Recurrent networks of N binary nodes s_i (0 or 1), with connection weights w_{ij}. (Left) The Hopfield model. (Centre) Boltzmann machine. The nodes are divided into two groups, visible (open circles) and hidden (grey) nodes. The network is trained to approximate the probability distribution of a given set of visible patterns. Once trained, the network can be used to generate new instances from the learned distribution. (Right) Restricted Boltzmann Machine (RBM). Same as the Boltzmann machine, but without any couplings within the visible layer or between hidden nodes. This variant can be used for layer-by-layer pre-training of deep networks.*

In 1983–1985 Geoffrey Hinton, together with Terrence Sejnowski and other coworkers, developed a stochastic extension of Hopfield's model from 1982, called the Boltzmann machine [23,24]. Here, each state $\mathbf{s}=(s_1,\ldots,s_N)$ of the network is assigned a probability given by the Boltzmann distribution

$$P(\boldsymbol{s}) \propto e^{-E/T} \qquad E = -\sum_{i<j} w_{ij} s_i s_j - \sum_i \theta_i s_i$$

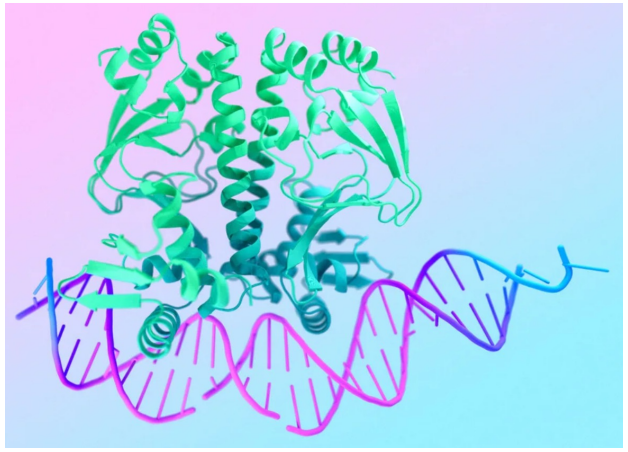
where T is a fictive temperature and θ_i is a bias, or local field.

Back-propagation to train the networks with hidden layers



Today: Nobel Prize in Chemistry 2024

- AlphaFold from Google DeepMind
- https://deepmind.google/technologies/alphafold/



Ill. Niklas Elmehed © Nobel Prize Outreach

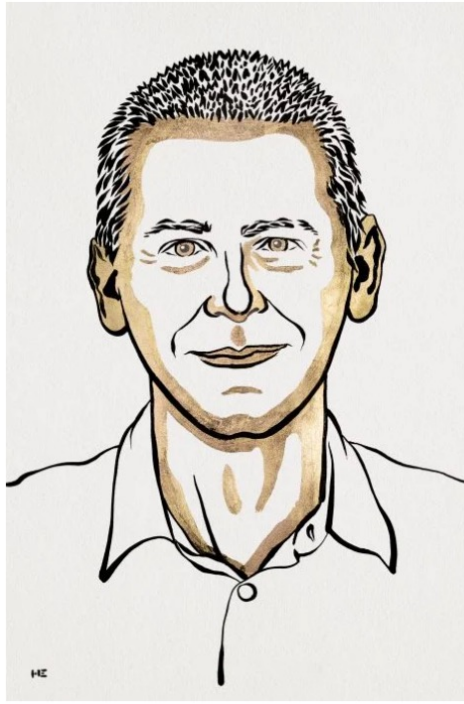
David Baker

Prize share: 1/2

Ill. Niklas Elmehed © Nobel Prize Outreach

Demis Hassabis

Prize share: 1/4



Ill. Niklas Elmehed © Nobel Prize Outreach

John M. Jumper

Prize share: 1/4

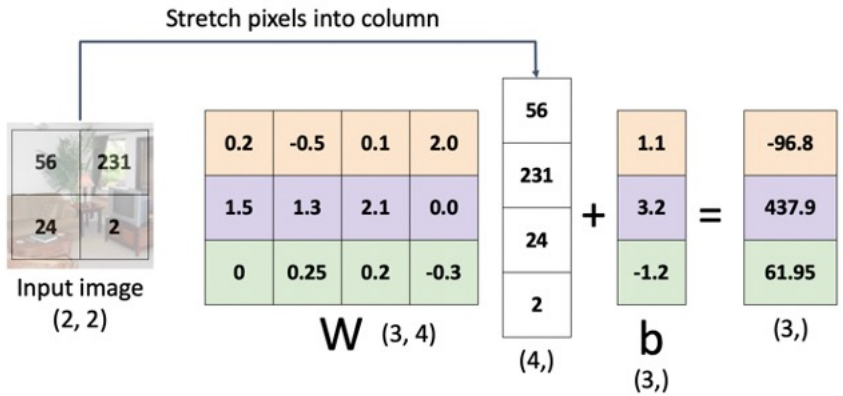
The Nobel Prize in Chemistry 2024 was divided, one half awarded to David Baker "for computational protein design", the other half jointly to Demis Hassabis and John M. Jumper "for protein structure prediction"

Recap: Linear NN/Classifier: Three Viewpoints

Equation Viewpoint

f(x,W) = Wx

f(x,W) = Wx + b

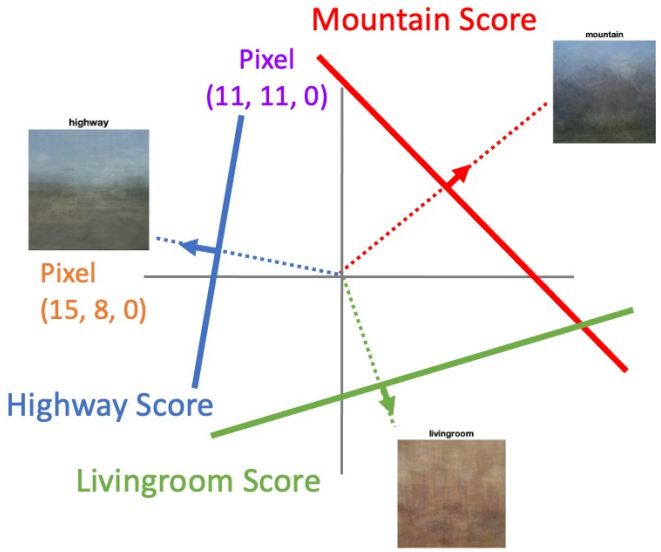


Visualization Viewpoint

One template per class

Geometric Viewpoint

Hyperplanes cutting up space



Recap: Loss Functions quantify preferences

- We have some dataset of (x, y)
- We have a **score function:**
- We have a **loss function**:

Linear classifier

$$s = f(x; W, b) = Wx + b$$

Linear regression loss:

$$L_i(W) = \frac{1}{2}(s_i - y_i)^2$$

Analytical solution:

$$\boldsymbol{W^* = (X^T X)^{-1} X^T y}$$

Cross-entropy loss:

$$L_i = -\log\left(\frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}\right)$$

SVM loss:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

But after we compute loss,
how do we find the best W?

A quick review on Cross-Entropy Loss

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k \mid X = x_i) = \frac{\exp(s_k)}{\sum_j \exp(s_j)}$$

Softmax function

$$L_i = -\log P(Y = y_i \mid X = x_i)$$

	Unnormalized log-probabilities / logits	exp	Probabilities must be >= 0	normalize	Probabilities must sum to 1	Compare	Correct probs
Bridge	**3.2**	→	**24.5**	→	**0.13**	→	**1.00**
Mountain	5.1		164.0		0.87		0.00
Coast	-1.7		0.18		0.00		0.00

Kullback–Leibler divergence

$$D_{KL}(\boldsymbol{P} \,||\, \boldsymbol{Q}) = \sum_y \boldsymbol{P}(\boldsymbol{y}) \log \frac{\boldsymbol{P}(\boldsymbol{y})}{\boldsymbol{Q}(\boldsymbol{y})}$$

A quick review on Cross-Entropy Loss

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k \mid X = x_i) = \frac{\exp(s_k)}{\sum_j \exp(s_j)}$$

Softmax function

Bridge **3.2**

Mountain 5.1

Coast -1.7

Maximize probability of correct class

$$L_i = -\log P(Y = y_i \mid X = x_i)$$

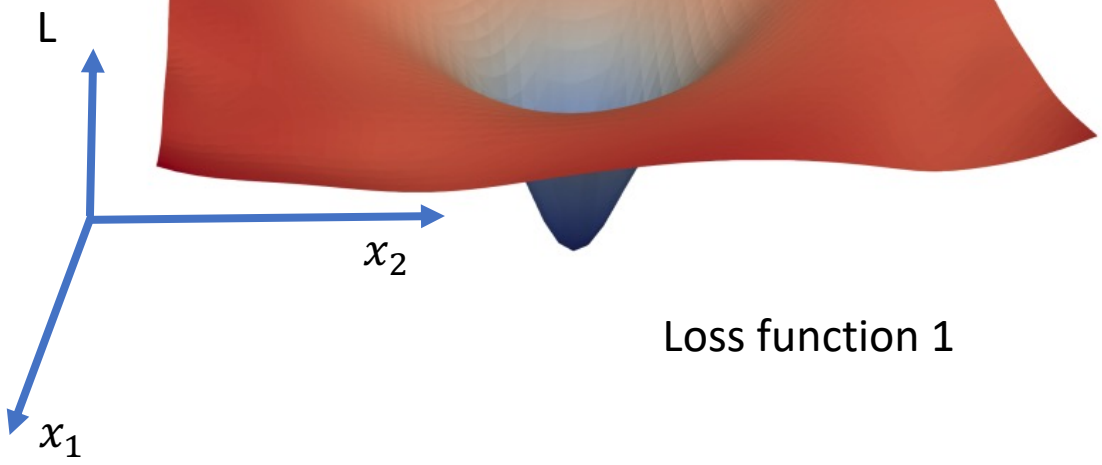
Putting it all together:

$$L_i = -\log\left(\frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}\right)$$

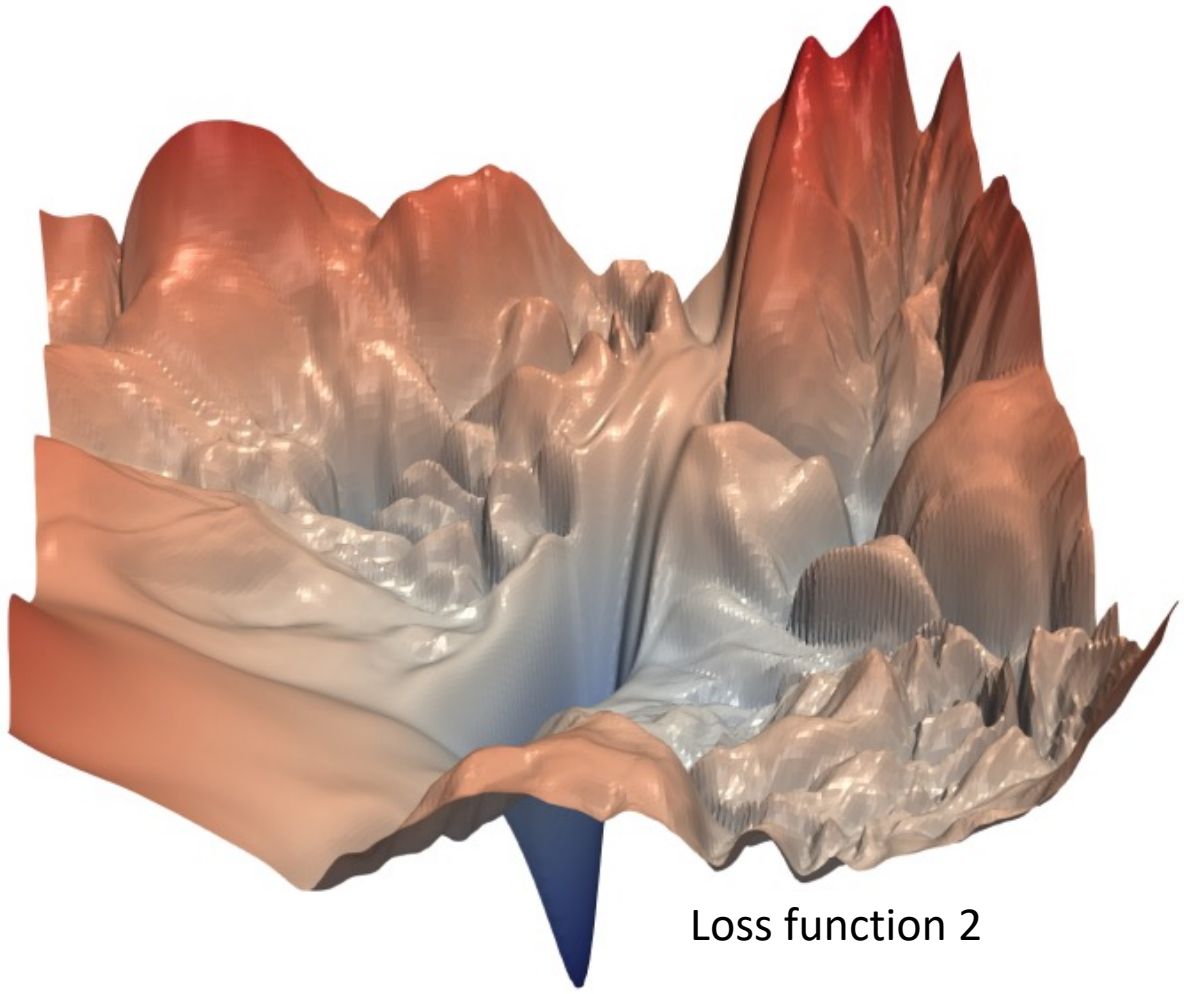
Optimization

$$w^* = \arg\min_w L(w)$$

Loss Landscape $L = f(x_1, x_2)$

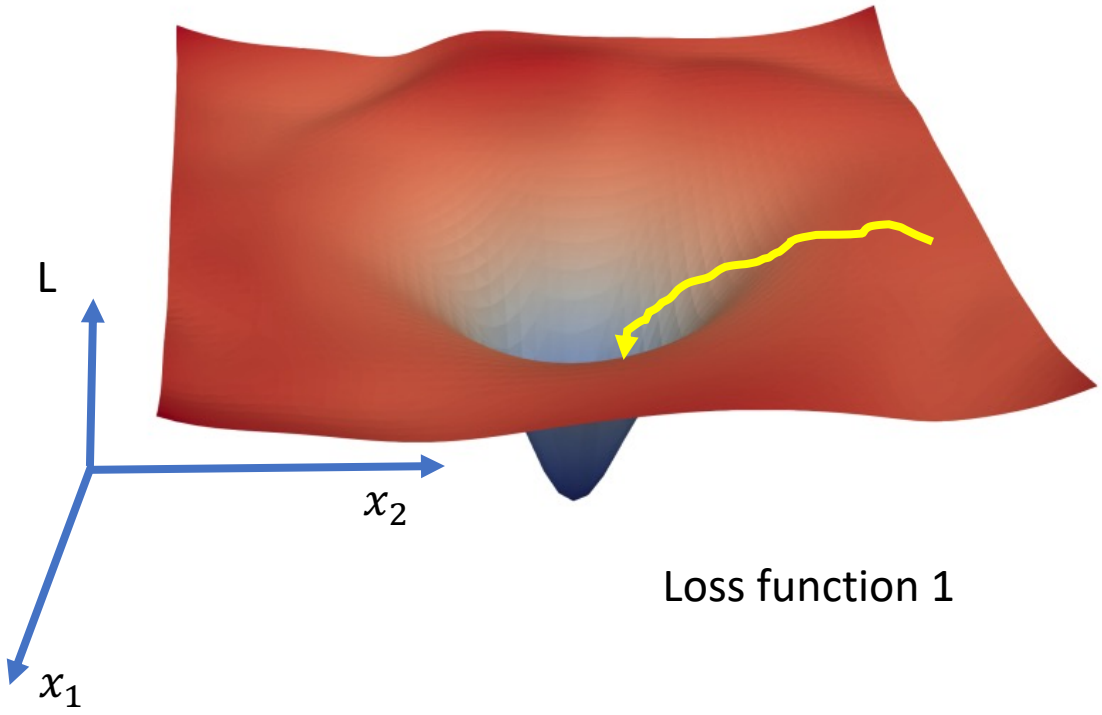


Loss function 1

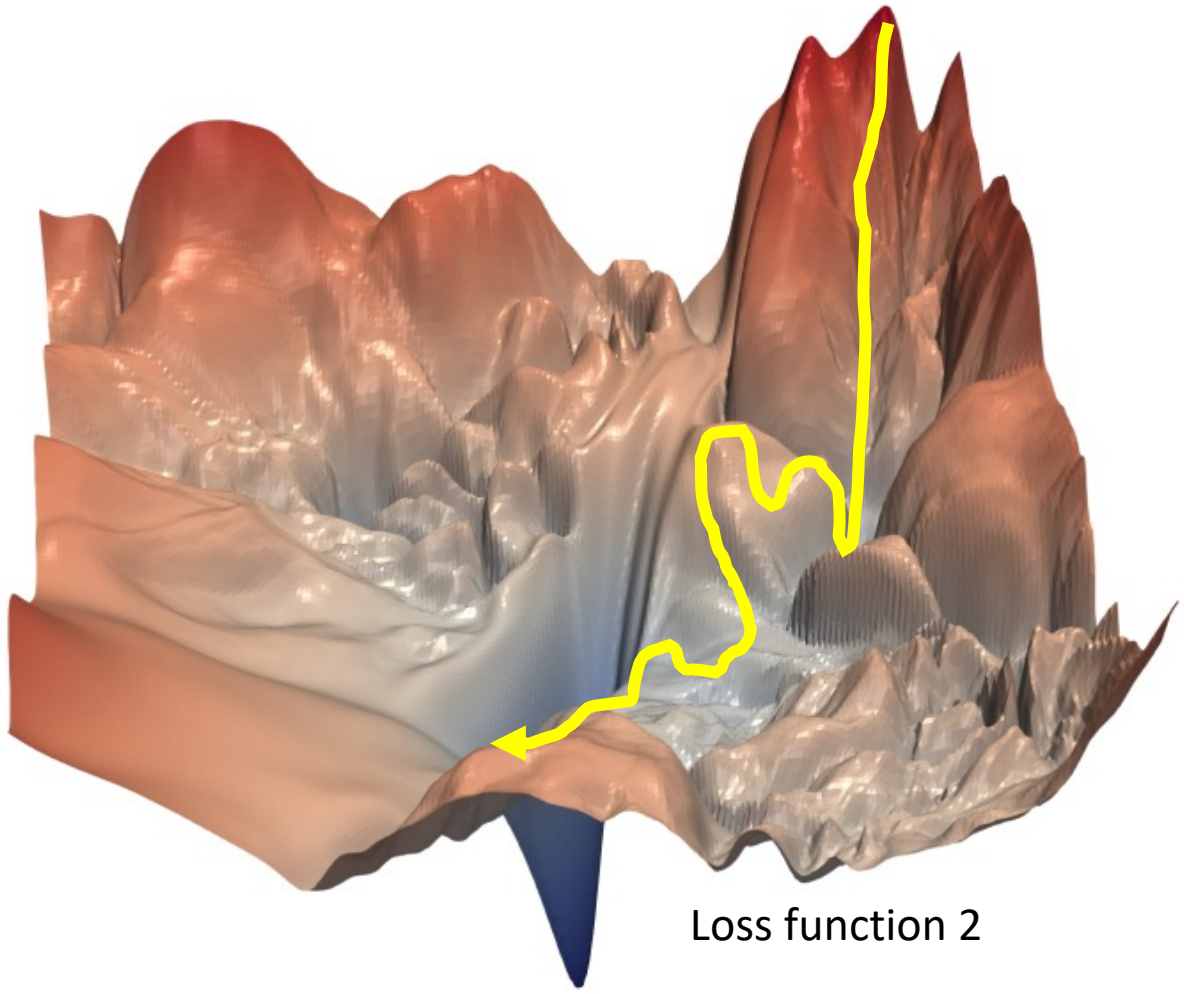


Loss function 2

Loss Landscape $L = f(x_1, x_2)$

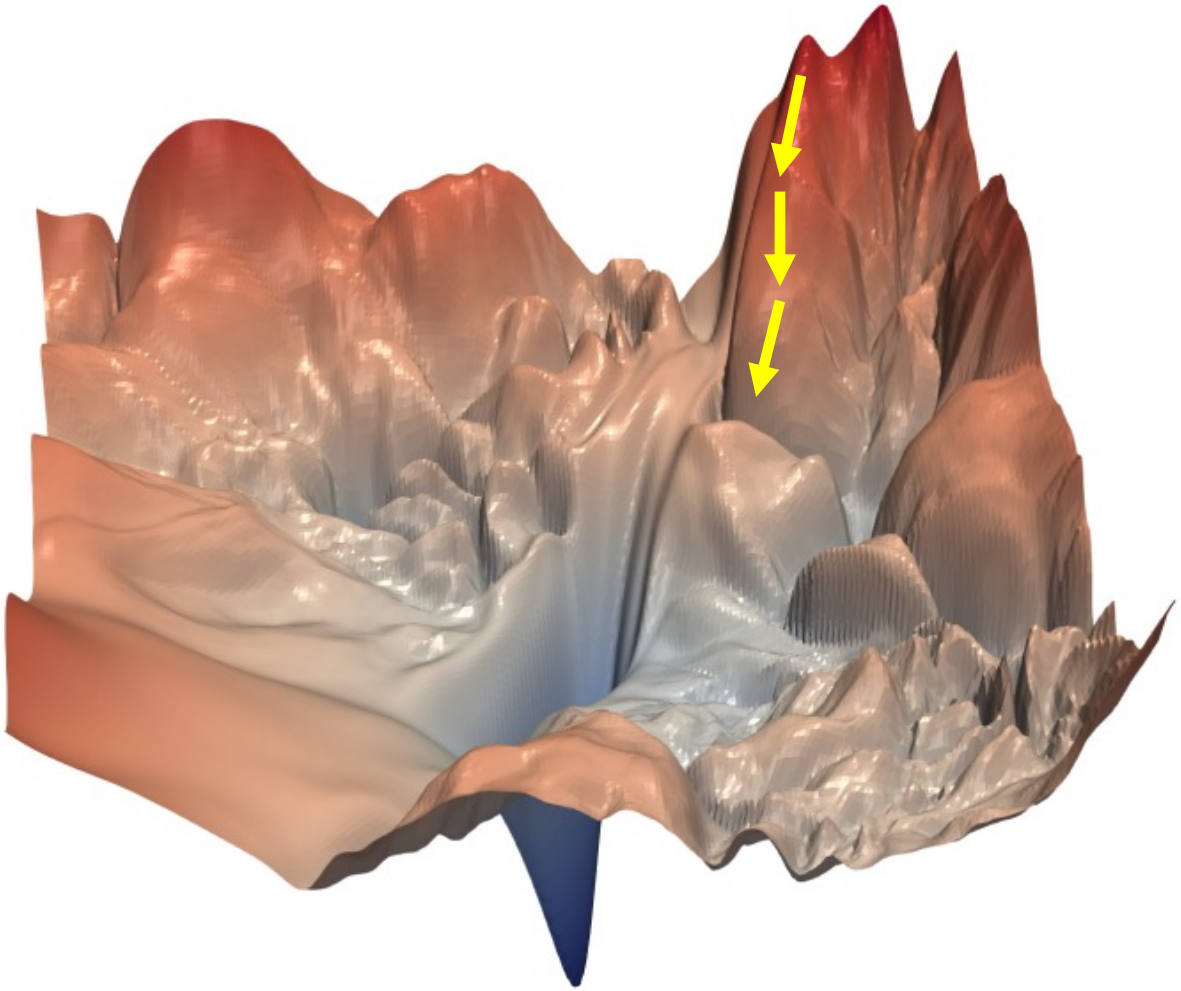


Loss function 1



Loss function 2

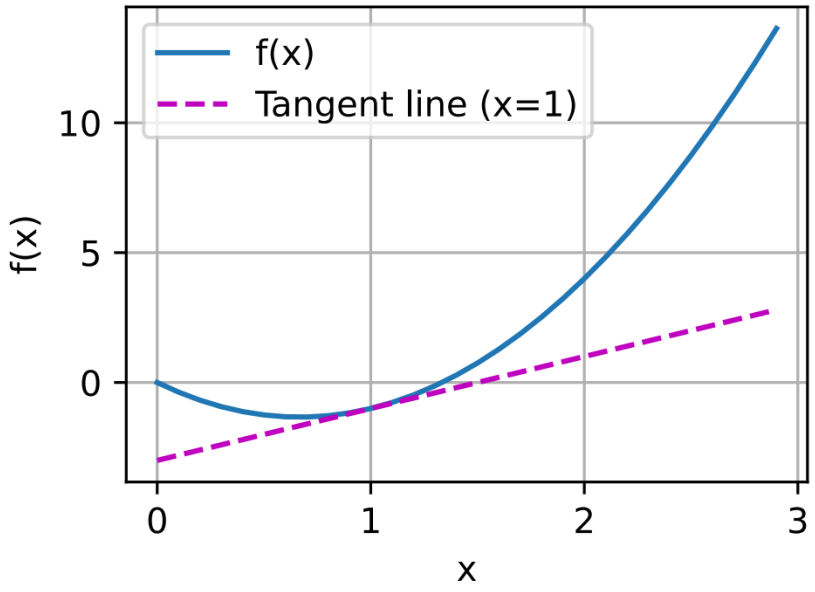
Follow the slope



Follow the slope

In 1-dimension, the **derivative** of a function gives the slope:

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



Follow the slope

In 1-dimension, the **derivative** of a function gives the slope:

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

In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n} \right]^\top,$$

Follow the slope

In 1-dimension, the **derivative** of a function gives the slope:

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

The slope in any direction is the **dot product** of the direction with the gradient
The direction of steepest descent is the **negative gradient**

current W:

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

loss 1.25347

gradient dL/dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

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 dL/dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

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 dL/dW:

[-2.5,
?,
?,

(1.25322 - 1.25347)/0.0001
= -2.5

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

?,
?,…]

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 dL/dW:

[-2.5,
?,
?,
?,
?,
?,
?,
?,
?,…]

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 dL/dW:

[-2.5,
0.6,
?,
?,

(1.25353 - 1.25347)/0.0001
= 0.6

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

?,…]

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 dL/dW:

[-2.5,
0.6,
?,
?,
?,
?,
?,
?,
?,…]

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 dL/dW:

[-2.5,
0.6,
0.0,
?,
?,

(1.25347 - 1.25347)/0.0001
= 0.0

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

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 dL/dW:

[-2.5,
0.6,
0.0,
?,
?,

Numeric Gradient:
- Slow: O(#dimensions)
- Approximate

Loss is a function of W

$$L(W) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}(s_i - y_i)^2$$

$s_i = Wx_i$

Want $\nabla_w L$

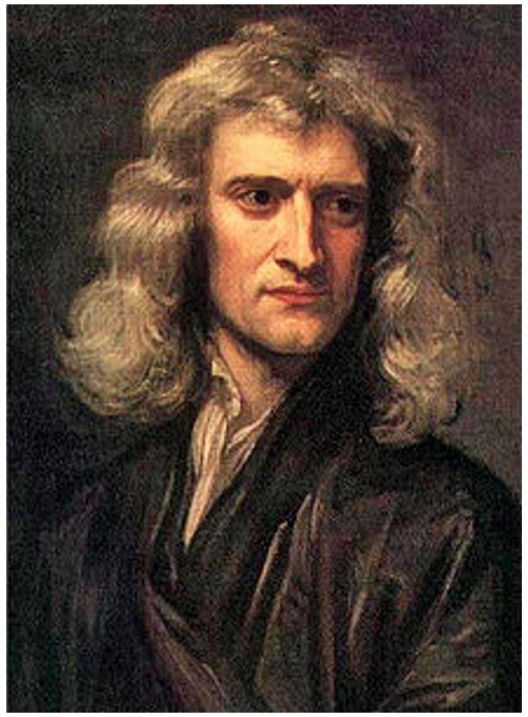
Loss is a function of W: **Analytic Gradient**

$$L(W) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}(s_i - y_i)^2$$

$$s_i = Wx_i$$

Want $\nabla_w L$

Newton



This image is in the public domain

Leibniz

This image is in the public domain

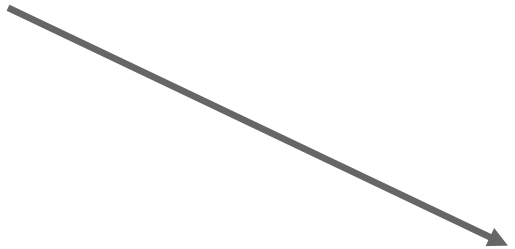
Use calculus to compute an **analytic gradient**

current W:

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

loss 1.25347

dL/dW = ...
(some function
of data and W)



gradient dL/dW:

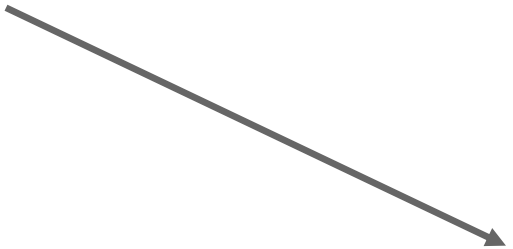
[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,…]

current W:

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

loss 1.25347

dL/dW = ...
(some function
of data and W)



(In practice we will compute dL/dW using backpropagation; wait until Lecture 6)

gradient dL/dW:

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

Computing Gradients

- **Numeric 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.**

Computing Gradients

- **Numeric 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.**

`torch.autograd.gradcheck(`*`func`*`,` *`inputs`*`,` *`eps=1e-06`*`,` *`atol=1e-05`*`,` *`rtol=0.001`*`,`
`raise_exception=True``,` *`check_sparse_nnz=False`*`,` *`nondet_tol=0.0`*`)` [SOURCE]

Check gradients computed via small finite differences against analytical gradients w.r.t. tensors in `inputs` that are of floating point type and with `requires_grad=True`.

The check between numerical and analytical gradients uses `allclose()`.

Gradient Descent

Iteratively step in the direction of
the negative gradient
(direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization method
- Number of steps
- Learning rate

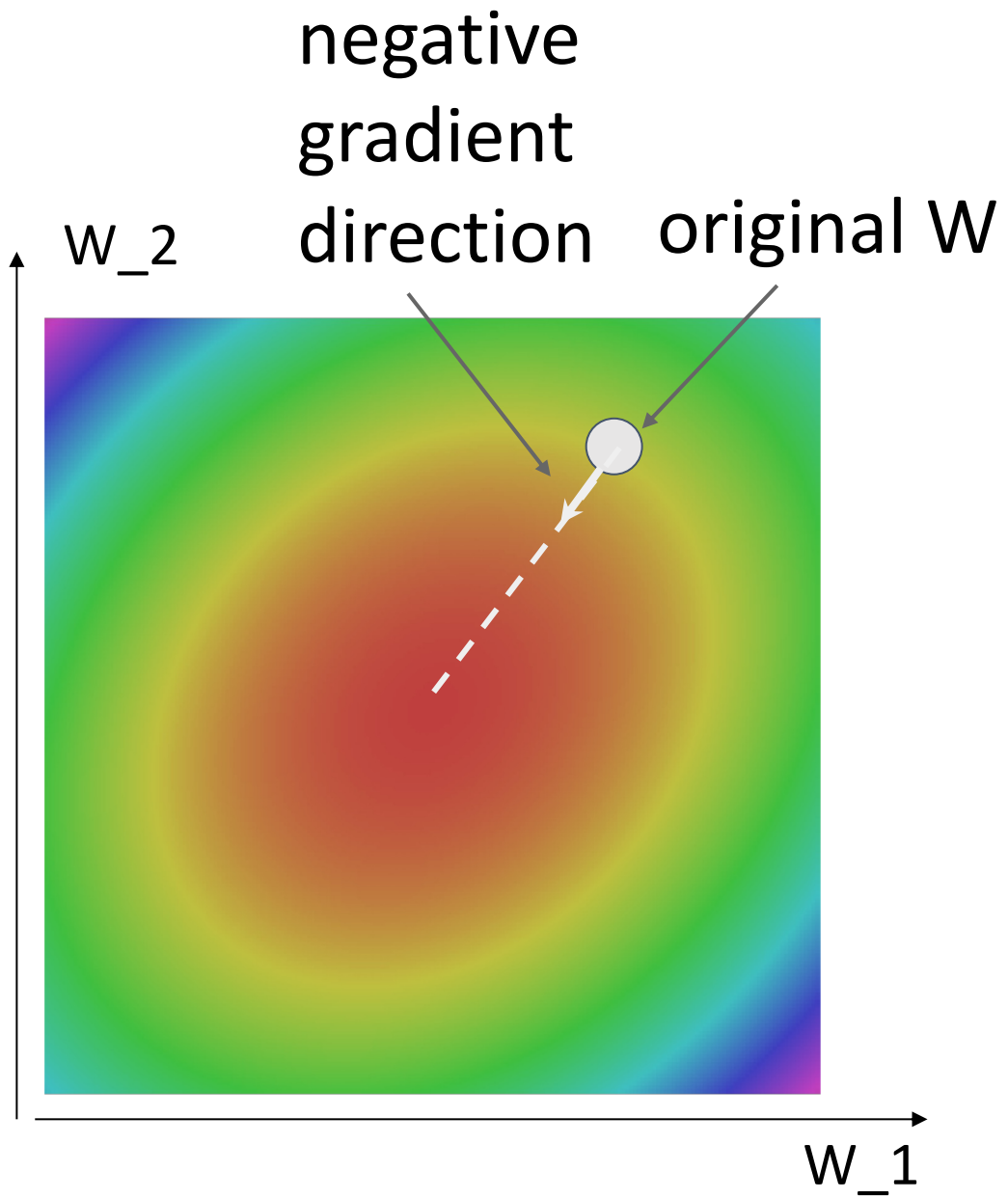
Gradient Descent

Iteratively step in the direction of the negative gradient
(direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization method
- Number of steps
- Learning rate



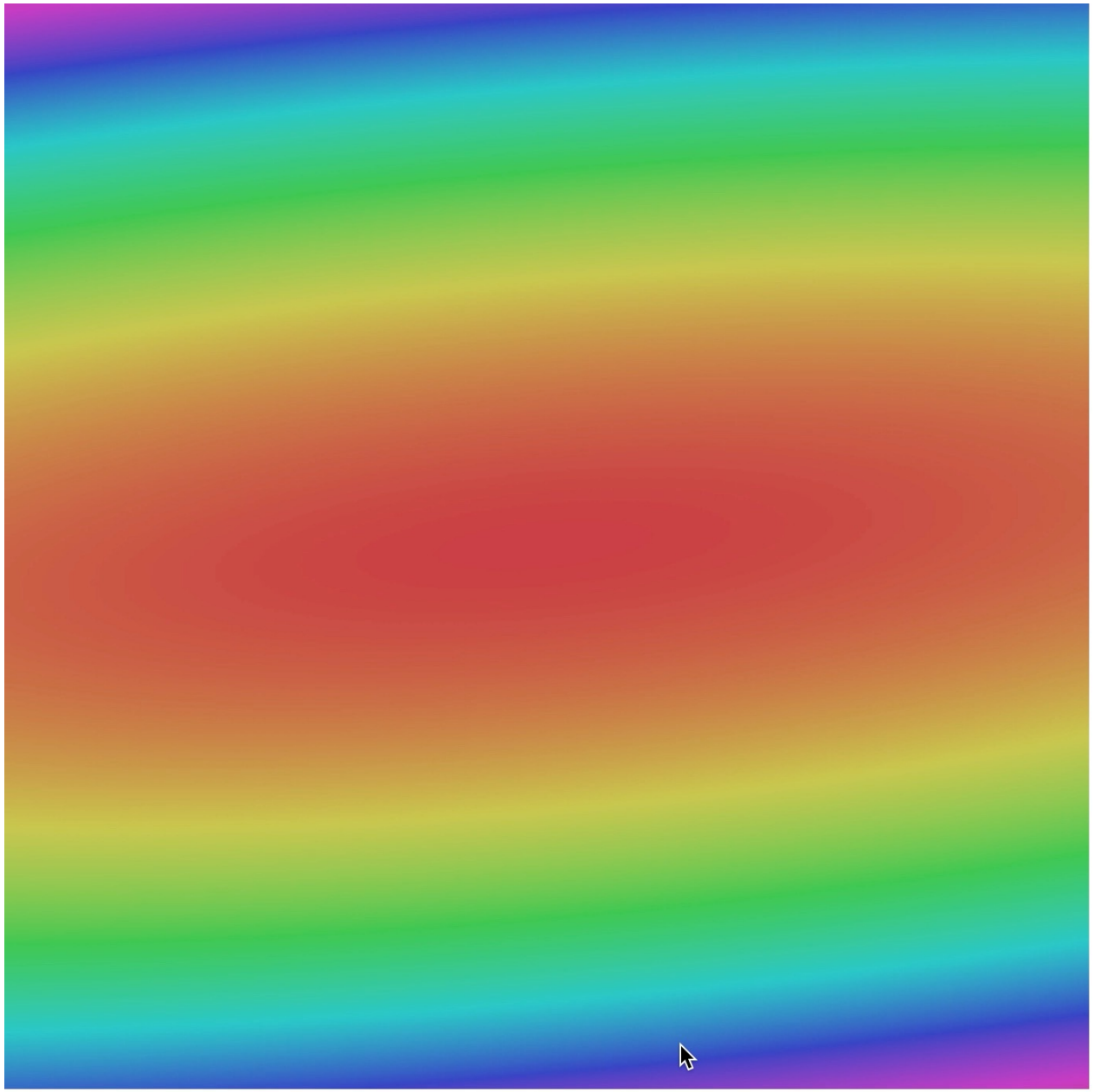
Gradient Descent

Iteratively step in the direction of the negative gradient
(direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization method
- Number of steps
- Learning rate



One-Dimensional Gradient Descend: Why the gradient descend algorithm may reduce the value of the objective function

Using a Taylor expansion, we have: $f(x+\epsilon) = f(x) + \epsilon f'(x) + \mathrm{O}(\epsilon^2)$

Let $\epsilon = -\eta f'(x)$ we can have: $f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + \mathrm{O}(\eta^2 f'^2(x))$

Thus $f\big(x - \eta f'(x)\big) \lesssim f(x)$

So we can iterate over $x \leftarrow x - \eta f'(x)$, the value of function $f(x)$ might decline

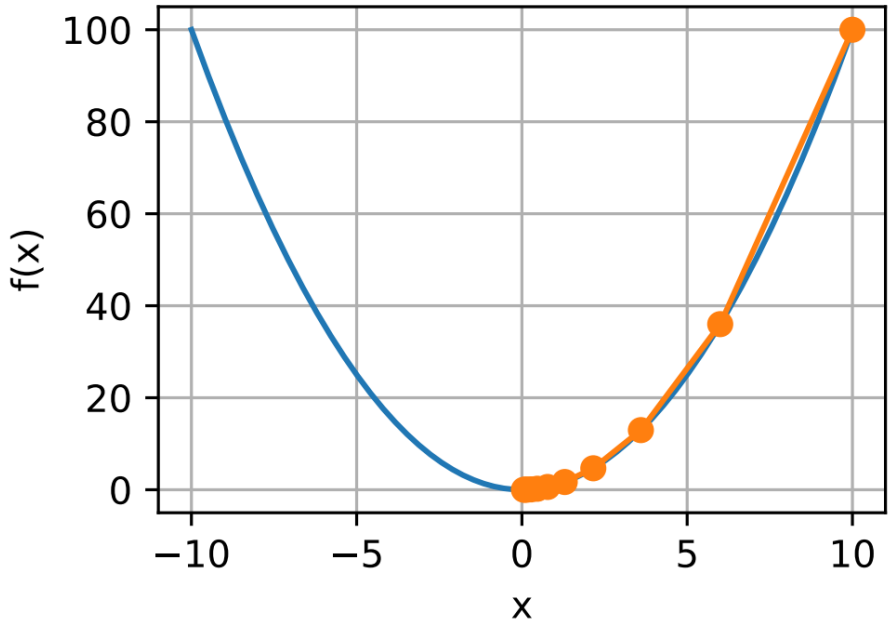
One-Dimensional Gradient Descend

Gradient Descend

```
def gd(eta, f_grad):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x)
        results.append(float(x))
    print(f'epoch 10, x: {x:f}')
    return results

results = gd(0.2, f_grad)
```

```
def f(x):  # Objective function
    return x ** 2

def f_grad(x):  # Gradient (derivative) of the objective functior
    return 2 * x
```

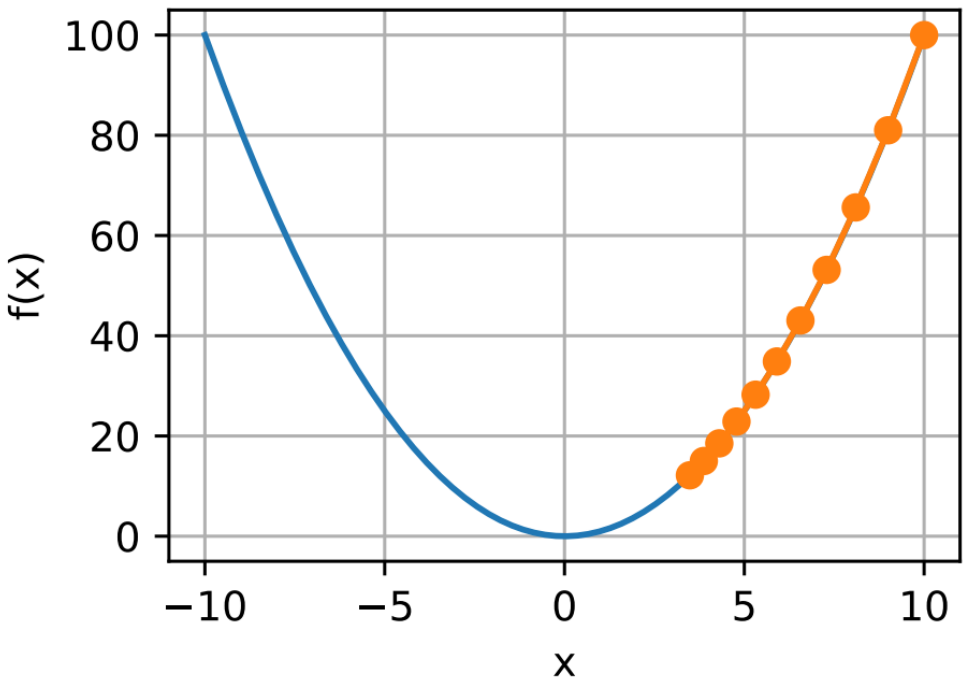
```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = torch.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [[f(x) for x in f_line], [
        f(x) for x in results]], 'x', 'f(x)', fmts=['-', '-o'])

show_trace(results, f)
```



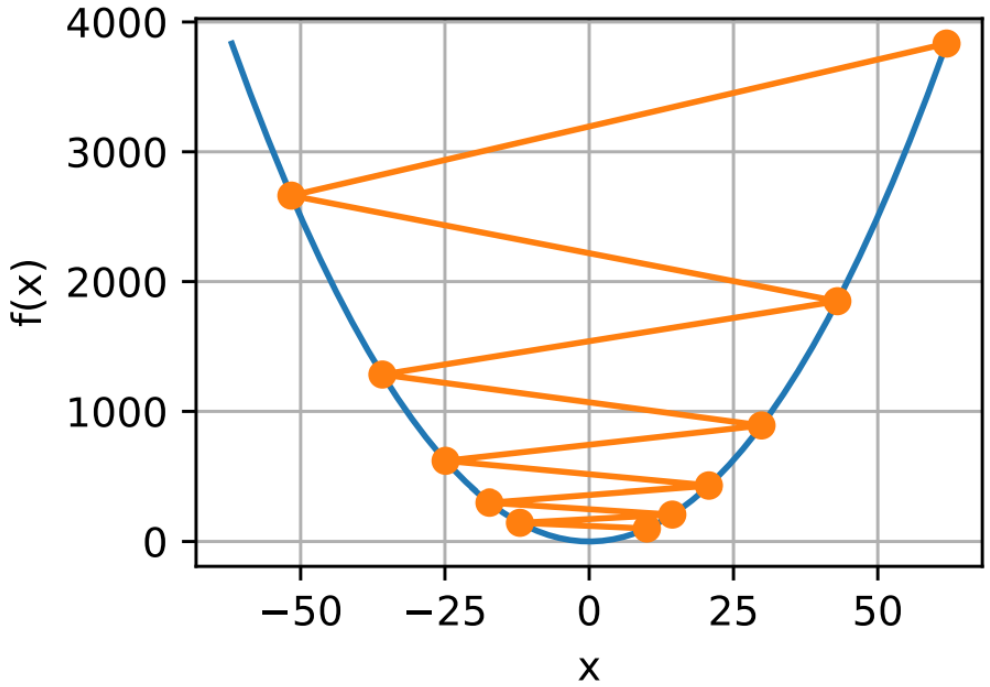
One-Dimensional Gradient Descend

Influence of the learning rate

```
show_trace(gd(0.05, f_grad), f)
```



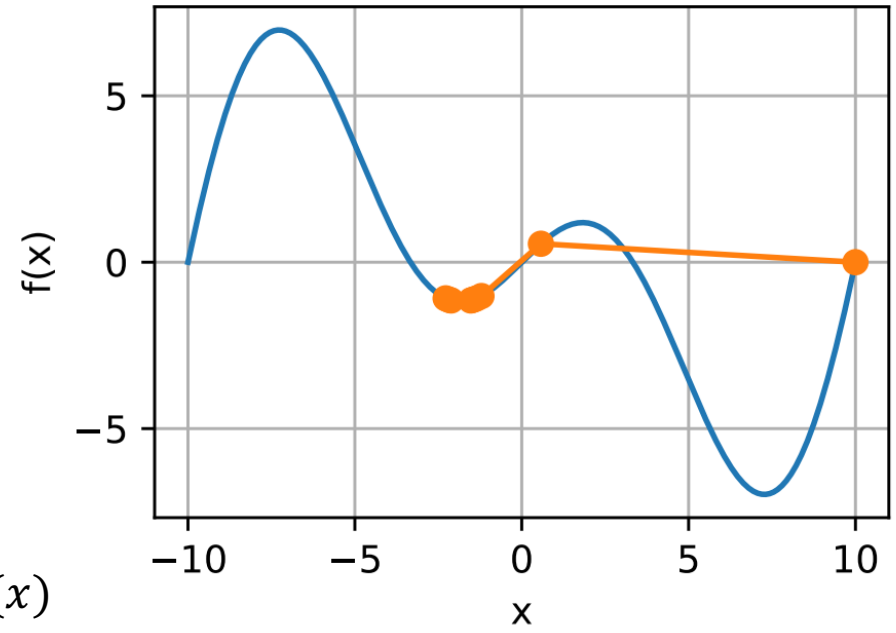

```
show_trace(gd(1.1, f_grad), f)
```



One-Dimensional Gradient Descend

Influence of local minimum

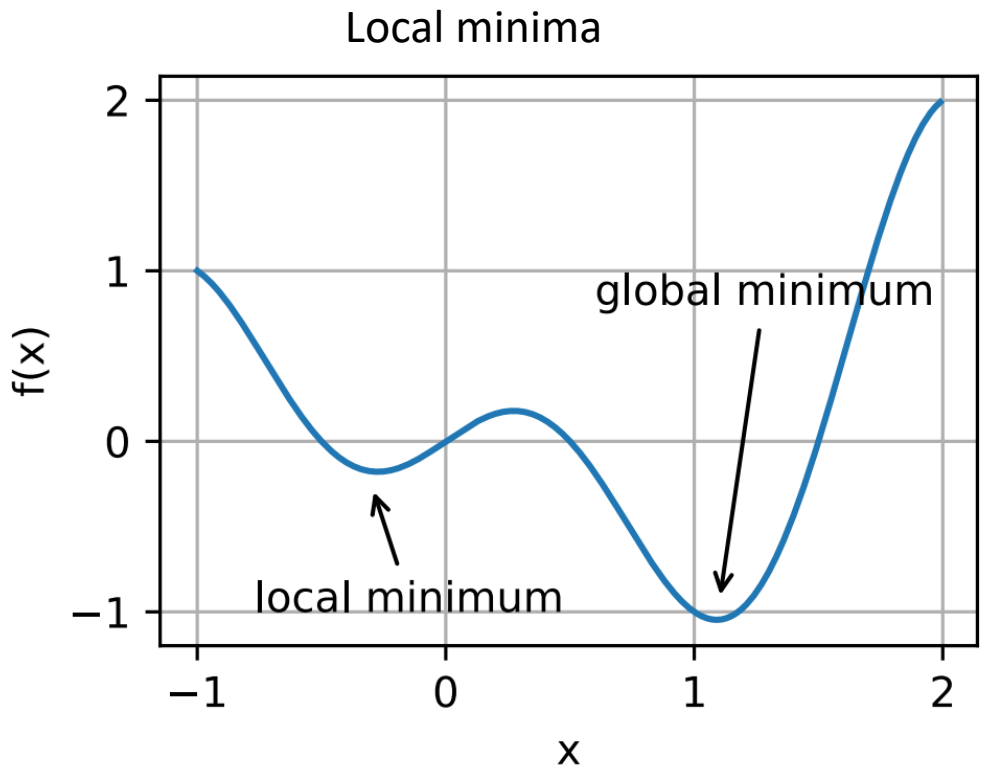
```
c = torch.tensor(0.15 * np.pi)

def f(x):  # Objective function
    return x * torch.cos(c * x)

def f_grad(x):  # Gradient of the objective function
    return torch.cos(c * x) - c * x * torch.sin(c * x)

show_trace(gd(2, f_grad), f)
```

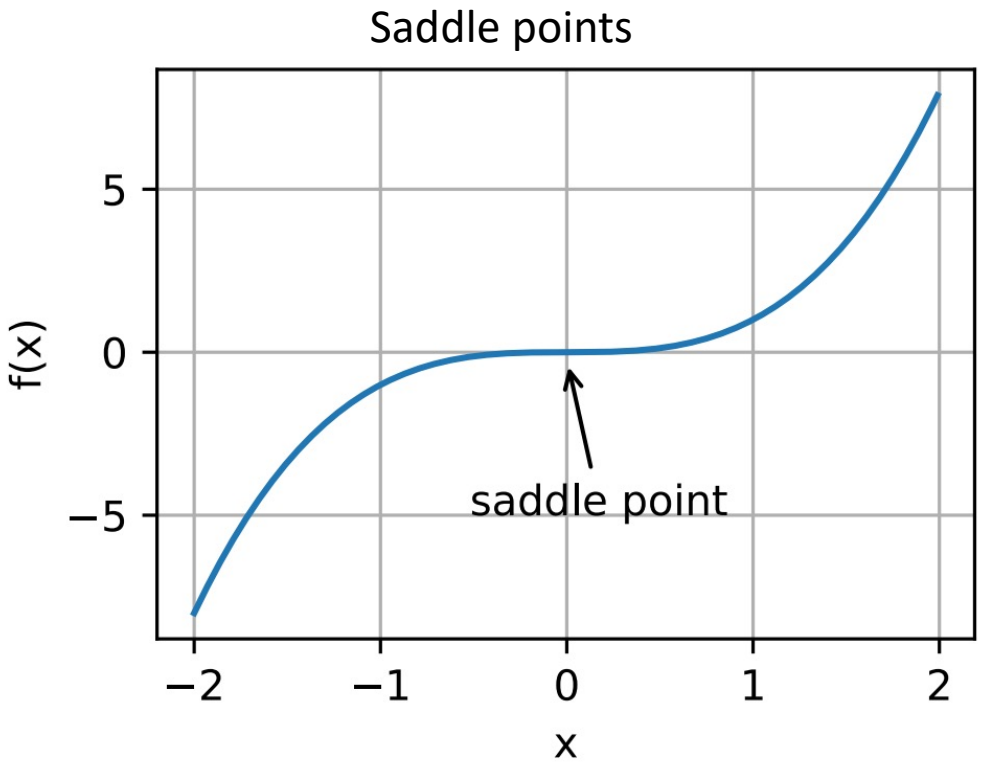


When iterate over $x \leftarrow x - \eta f'(x)$, the value of function $f(x)$ stops when $f'(x)$ becomes zero

Challenges with Gradient Descend

Gradient becomes zeros or vanishes





Zero gradient, gradient descent gets stuck

Figures from d2l.ai

Batch Gradient Descent

A function of both samples and model parameter

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

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

Full sum is expensive when N is large!

Stochastic Gradient Descent (SGD)

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

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

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples 32 / 64 / 128 common

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization
- Number of steps
- Learning rate
- Batch size
- Data sampling

Stochastic Gradient Descent (SGD)

$$L(W) = \mathbb{E}_{(x,y)\sim p_{data}}[L(x, y, W)]$$
$$\approx \frac{1}{N}\sum_{i=1}^{N} L(x_i, y_i, W)$$

Think of loss as an expectation over the full **data distribution** p_{data}

Approximate expectation via sampling

Stochastic Gradient Descent (SGD)

$$L(W) = \mathbb{E}_{(x,y)\sim p_{data}}[L(x, y, W)]$$
$$\approx \frac{1}{N}\sum_{i=1}^{N} L(x_i, y_i, W)$$

Think of loss as an expectation over the full **data distribution** p_{data}

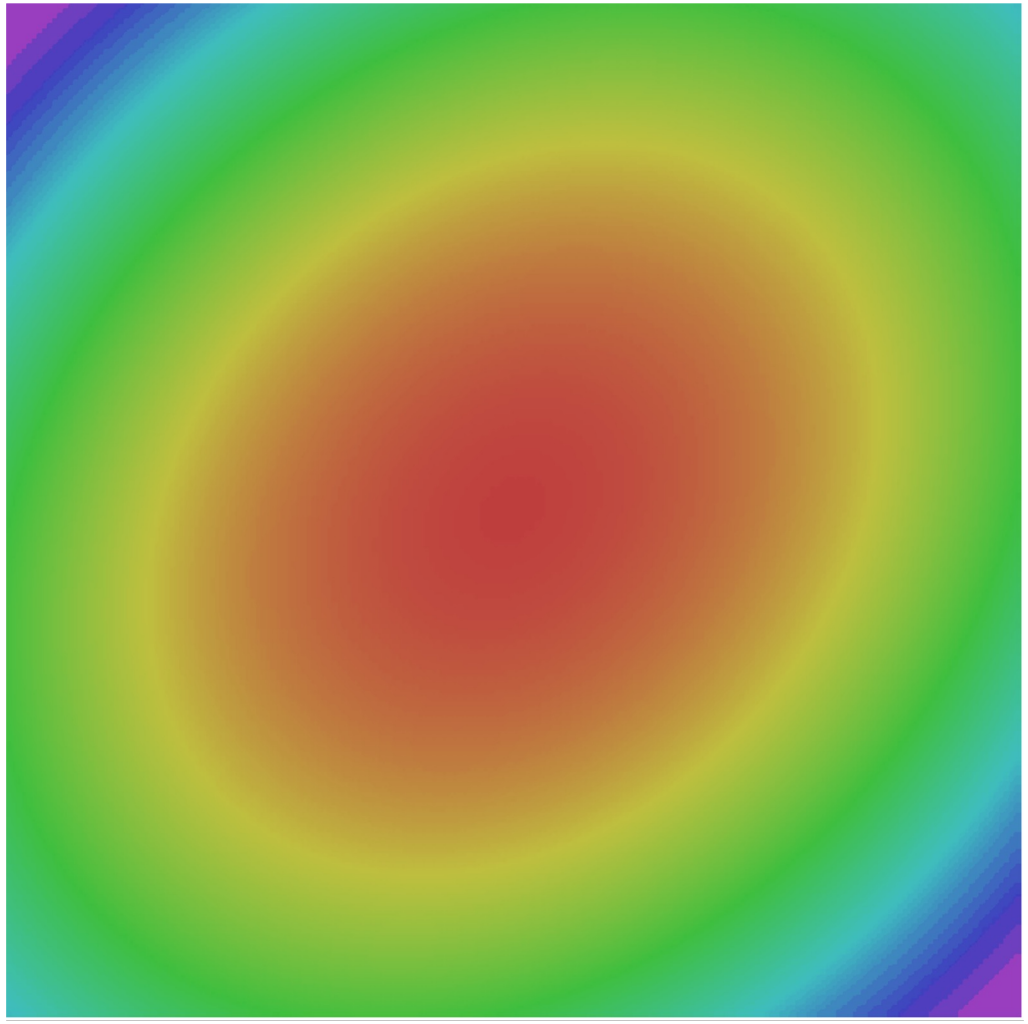
Approximate expectation via sampling

$$\nabla_W L(W) = \nabla_W \mathbb{E}_{(x,y)\sim p_{data}}[L(x, y, W)]$$
$$\approx \sum_{i=1}^{N} \nabla_w L_W(x_i, y_i, W)$$

Gradient can be noisy when minibatch SGD

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

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



SGD

SGD

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

```
for t in range(num_steps):
  dw = compute_gradient(w)
  w -= learning_rate * dw
```

SGD + Momentum

SGD

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

```
for t in range(num_steps):
  dw = compute_gradient(w)
  w -= learning_rate * dw
```

SGD+Momentum

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

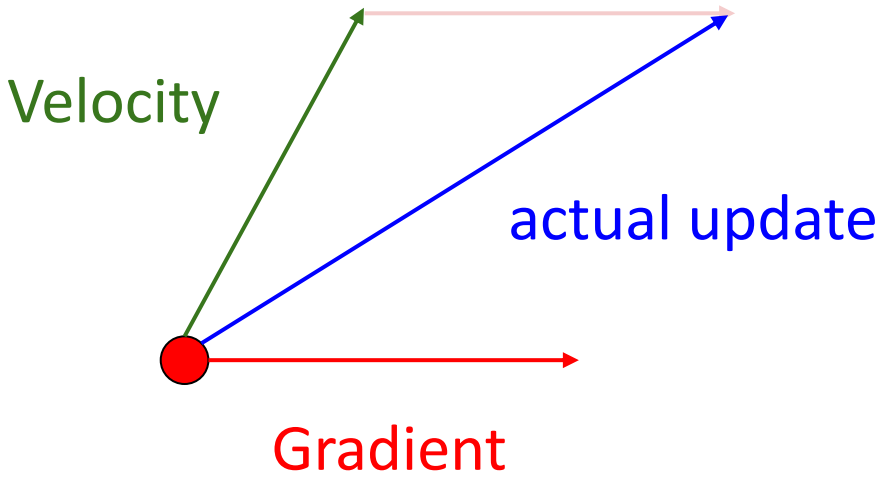
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

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

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

Momentum update:



Combine gradient at current point with velocity to update weights

SGD+Momentum

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

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

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

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

SGD+Momentum

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

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v - learning_rate * dw
  w += v
```

SGD+Momentum

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

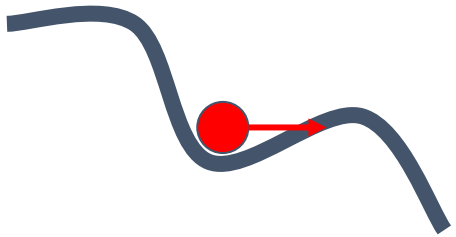
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of x

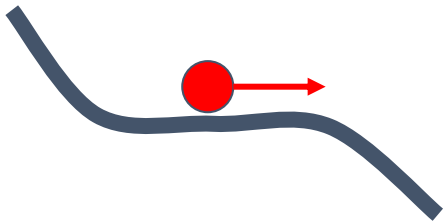
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

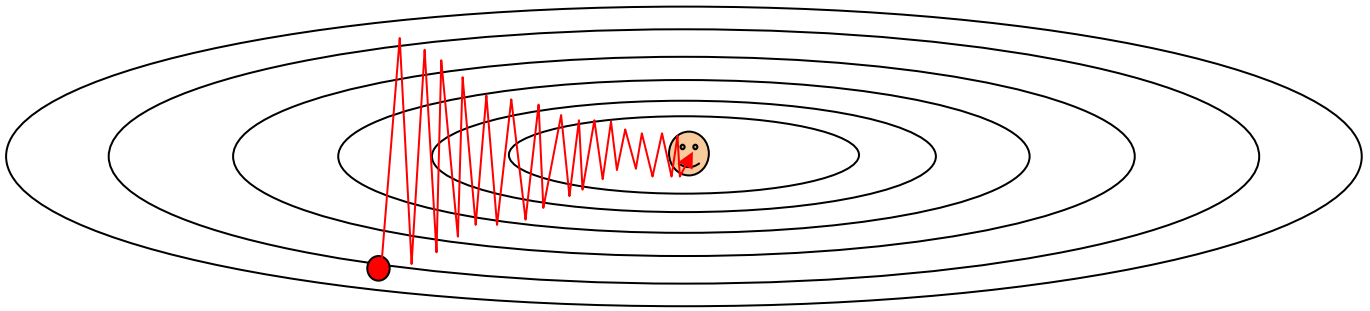
Local Minima



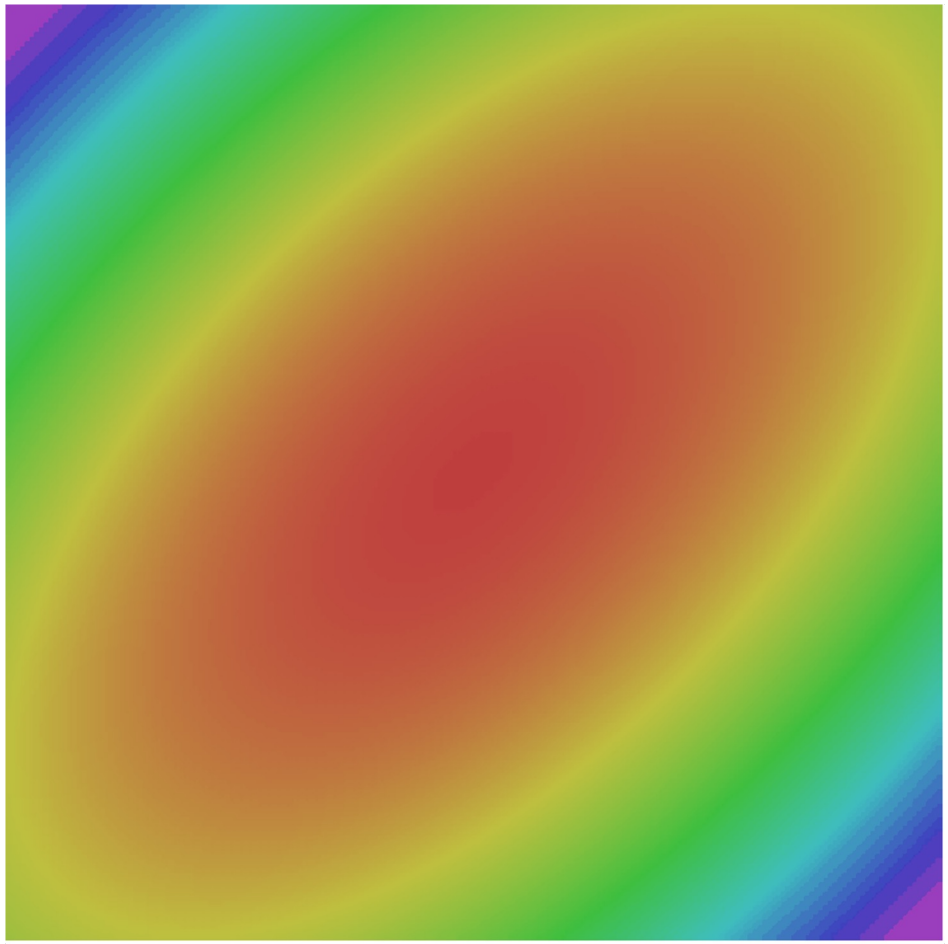
Saddle points



Poor Conditioning



Gradient Noise



SGD SGD+Momentum

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

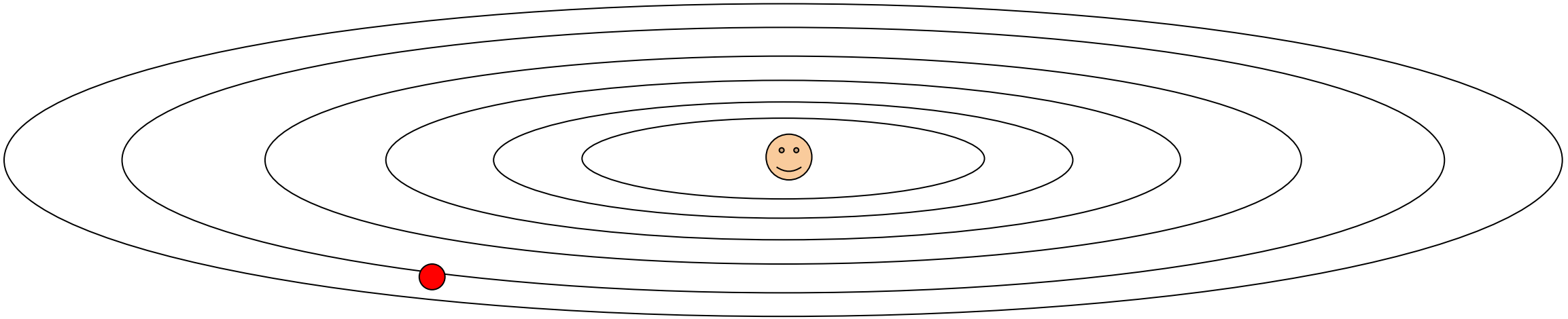
"Per-parameter learning rates"
or "adaptive learning rates"

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

AdaGrad

$$\mathbf{g}_t = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})),$$
$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2,$$
$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.$$

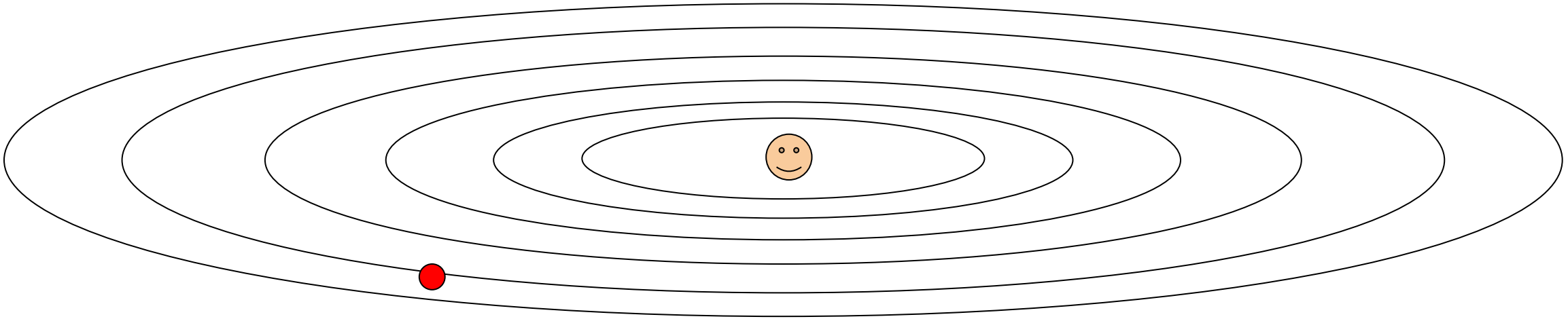
```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```



Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

AdaGrad

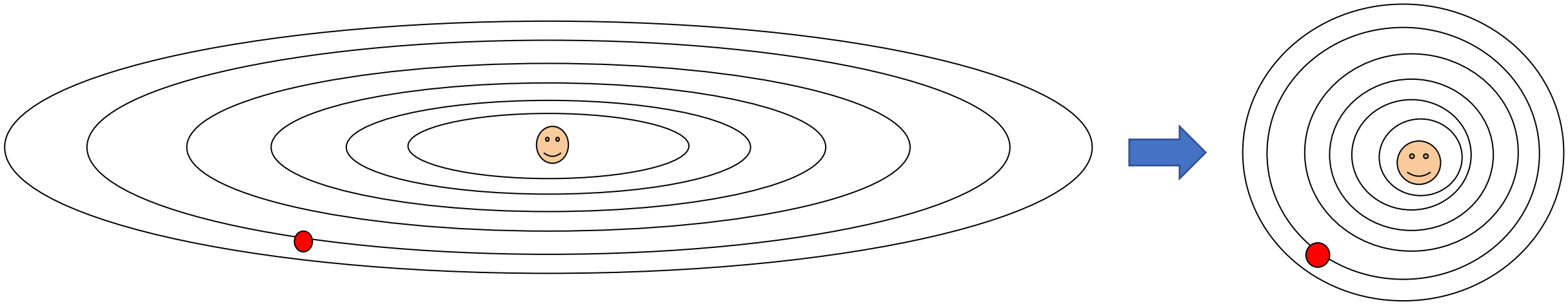
```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```



Q: What happens with AdaGrad?

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```



Q: What happens with AdaGrad?

Progress along “steep” directions is damped;
progress along “flat” directions is accelerated

RMSProp: "Weighted Adagrad"

$$\mathbf{g}_t = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})),$$
$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2,$$
$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.$$

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

$$\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2,$$
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t.$$

Tieleman and Hinton, 2012

RMSProp

SGD

SGD+Momentum

RMSProp

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Common choice: beta1 = 0.9, beta2 = 0.999

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

SGD+Momentum

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

AdaGrad / RMSProp

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

AdaGrad / RMSProp

Bias correction

Q: What happens at t=0 and moment1=0, moment2 = 0?
(Assume beta2 = 0.999)

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

AdaGrad / RMSProp

Bias correction

Q: What happens at t=0 and moment1=0, moment2 = 0?
(Assume beta2 = 0.999)
A: Bias towards smaller values at the beginning

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  moment1_unbias = moment1 / (1 - beta1 ** t)
  moment2_unbias = moment2 / (1 - beta2 ** t)
  w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

Momentum

AdaGrad / RMSProp

Bias correction

Bias correction for the fact that first and second moment estimates start at zero

$$\mathbf{v}_t \leftarrow \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\mathbf{g}_t,$$
$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2.$$

$$\sum_{i=0}^{t} \beta^i = \frac{1-\beta^t}{1-\beta}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t}.$$

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

More explanation on the bias correction: https://www.youtube.com/watch?v=lWzo8CajF5s

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  moment1_unbias = moment1 / (1 - beta1 ** t)
  moment2_unbias = moment2 / (1 - beta2 ** t)
  w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3, 5e-4, 1e-4 is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam: A method for stochastic optimization

DP Kingma, J Ba

arXiv preprint arXiv:1412.6980, 2014 • arxiv.org

We introduce Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-

SHOW MORE

☆ Save Cite Cited by 165687 Related articles All 27 versions »

Published as a conference paper at ICLR 2015

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma* University of Amsterdam, OpenAI dpkingma@openai.com

Jimmy Lei Ba* University of Toronto jimmy@psi.utoronto.ca

ABSTRACT

We introduce *Adam*, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning. Some connections to related algorithms, on which *Adam* was inspired, are discussed. We also analyze the theoretical convergence properties of the algorithm and provide a regret bound on the convergence rate that is comparable to the best known results under the online convex optimization framework. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods. Finally, we discuss *AdaMax*, a variant of *Adam* based on the infinity norm.

https://arxiv.org/pdf/1412.6980.pdf

4 CONVERGENCE ANALYSIS

We analyze the convergence of Adam using the online learning framework proposed in (Zinkevich, 2003). Given an arbitrary, unknown sequence of convex cost functions $f_1(\theta), f_2(\theta),..., f_T(\theta)$. At each time t, our goal is to predict the parameter θ_t and evaluate it on a previously unknown cost function f_t. Since the nature of the sequence is unknown in advance, we evaluate our algorithm using the regret, that is the sum of all the previous difference between the online prediction $f_t(\theta_t)$ and the best fixed point parameter $f_t(\theta^*)$ from a feasible set $\mathcal{X}$ for all the previous steps. Concretely, the regret is defined as:

$$R(T) = \sum_{t=1}^{T} [f_t(\theta_t) - f_t(\theta^*)] \tag{5}$$

where $\theta^* = \arg\min_{\theta \in \mathcal{X}} \sum_{t=1}^{T} f_t(\theta)$. We show Adam has $O(\sqrt{T})$ regret bound and a proof is given in the appendix. Our result is comparable to the best known bound for this general convex online learning problem. We also use some definitions simplify our notation, where $g_t \triangleq \nabla f_t(\theta_t)$ and $g_{t,i}$ as the i^{th} element. We define $g_{1:t,i} \in \mathbb{R}^t$ as a vector that contains the i^{th} dimension of the gradients over all iterations till t, $g_{1:t,i} = [g_{1,i}, g_{2,i}, \cdots, g_{t,i}]$. Also, we define $\gamma \triangleq \frac{\beta_1^2}{\sqrt{\beta_2}}$. Our following theorem holds when the learning rate α_t is decaying at a rate of $t^{-\frac{1}{2}}$ and first moment running average coefficient $\beta_{1,t}$ decay exponentially with λ, that is typically close to 1, e.g. $1 - 10^{-8}$.

Theorem 4.1. *Assume that the function f_t has bounded gradients, $\|\nabla f_t(\theta)\|_2 \leq G$, $\|\nabla f_t(\theta)\|_\infty \leq G_\infty$ for all $\theta \in R^d$ and distance between any θ_t generated by Adam is bounded, $\|\theta_n - \theta_m\|_2 \leq D$, $\|\theta_m - \theta_n\|_\infty \leq D_\infty$ for any $m, n \in \{1, ..., T\}$, and β_1, $\beta_2 \in [0, 1)$ satisfy $\frac{\beta_1^2}{\sqrt{\beta_2}} < 1$. Let $\alpha_t = \frac{\alpha}{\sqrt{t}}$ and $\beta_{1,t} = \beta_1 \lambda^{t-1}$, $\lambda \in (0, 1)$. Adam achieves the following guarantee, for all $T \geq 1$.*

$$R(T) \leq \frac{D^2}{2\alpha(1-\beta_1)} \sum_{i=1}^{d} \sqrt{T\hat{v}_{T,i}} + \frac{\alpha(1+\beta_1)G_\infty}{(1-\beta_1)\sqrt{1-\beta_2}(1-\gamma)^2} \sum_{i=1}^{d} \|g_{1:T,i}\|_2 + \sum_{i=1}^{d} \frac{D_\infty^2 G_\infty \sqrt{1-\beta_2}}{2\alpha(1-\beta_1)(1-\lambda)^2}$$

Our Theorem 4.1 implies when the data features are sparse and bounded gradients, the summation term can be much smaller than its upper bound $\sum_{i=1}^{d} \|g_{1:T,i}\|_2 << dG_\infty\sqrt{T}$ and $\sum_{i=1}^{d} \sqrt{T\hat{v}_{T,i}} << dG_\infty\sqrt{T}$, in particular if the class of function and data features are in the form of section 1.2 in (Duchi et al., 2011). Their results for the expected value $\mathbb{E}[\sum_{i=1}^{d} \|g_{1:T,i}\|_2]$ also apply to Adam. In particular, the adaptive method, such as Adam and Adagrad, can achieve $O(\log d\sqrt{T})$, an improvement over $O(\sqrt{dT})$ for the non-adaptive method. Decaying $\beta_{1,t}$ towards zero is important in our theoretical analysis and also matches previous empirical findings, e.g. (Sutskever et al., 2013) suggests reducing the momentum coefficient in the end of training can improve convergence.

Finally, we can show the average regret of Adam converges,

Corollary 4.2. *Assume that the function f_t has bounded gradients, $\|\nabla f_t(\theta)\|_2 \leq G$, $\|\nabla f_t(\theta)\|_\infty \leq G_\infty$ for all $\theta \in R^d$ and distance between any θ_t generated by Adam is bounded, $\|\theta_n - \theta_m\|_2 \leq D$, $\|\theta_m - \theta_n\|_\infty \leq D_\infty$ for any $m, n \in \{1, ..., T\}$. Adam achieves the following guarantee, for all $T \geq 1$.*

$$\frac{R(T)}{} \quad 1$$

Adam

SGD

SGD+Momentum

RMSProp

Adam

Optimization Algorithm Comparison

Algorithm	Tracks first moments (Momentum)	Tracks second moments (Adaptive learning rates)	Weighted second moments	Bias correction for moment estimates
SGD	✗	✗	✗	✗
SGD+Momentum	✓	✗	✗	✗
AdaGrad	✗	✓	✗	✗
RMSProp	✗	✓	✓	✗
Adam	✓	✓	✓	✓

Overfitting

A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data

Overfitting

A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data

Example: Linear classifier with 1D inputs, 2 classes, softmax loss

$$s_i = w_i x + b_i$$

$$p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$

$$L = -\log(p_y)$$

Overfitting

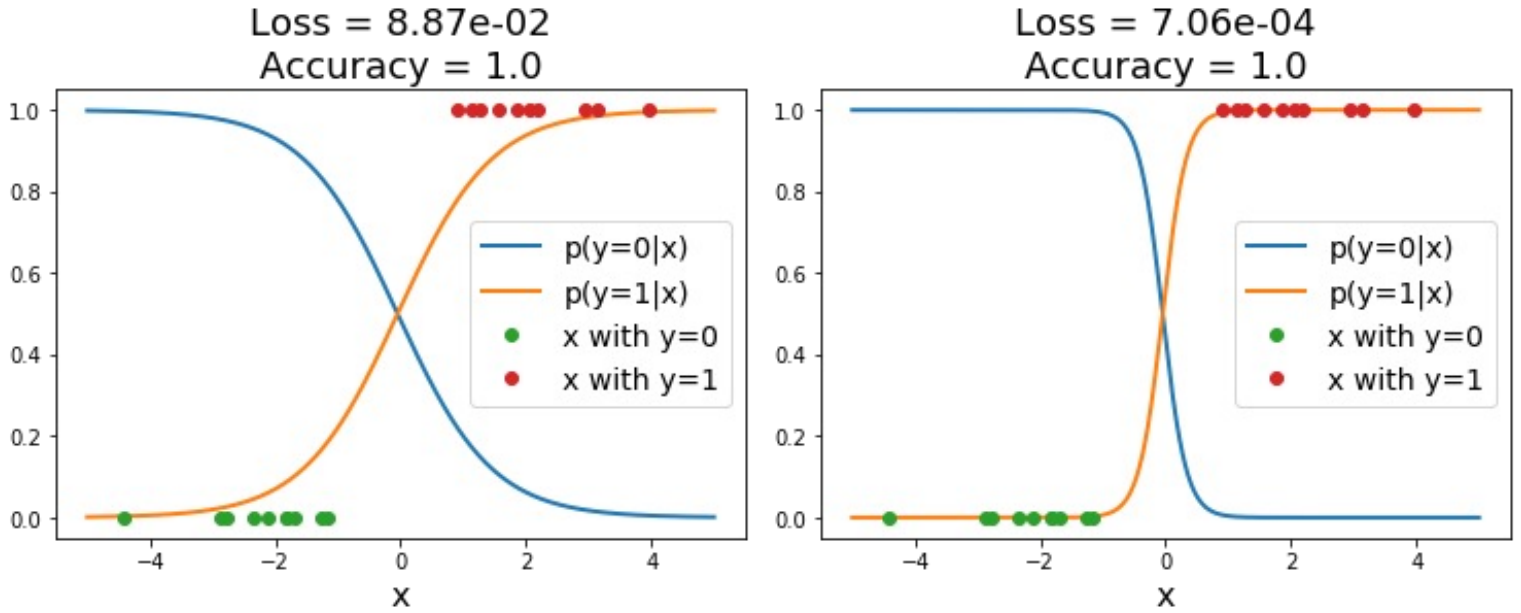
A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data

Example: Linear classifier with 1D inputs, 2 classes, softmax loss

$$s_i = w_i x + b_i$$

$$p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$

$$L = -\log(p_y)$$



Both models have perfect accuracy on train data!

Overfitting

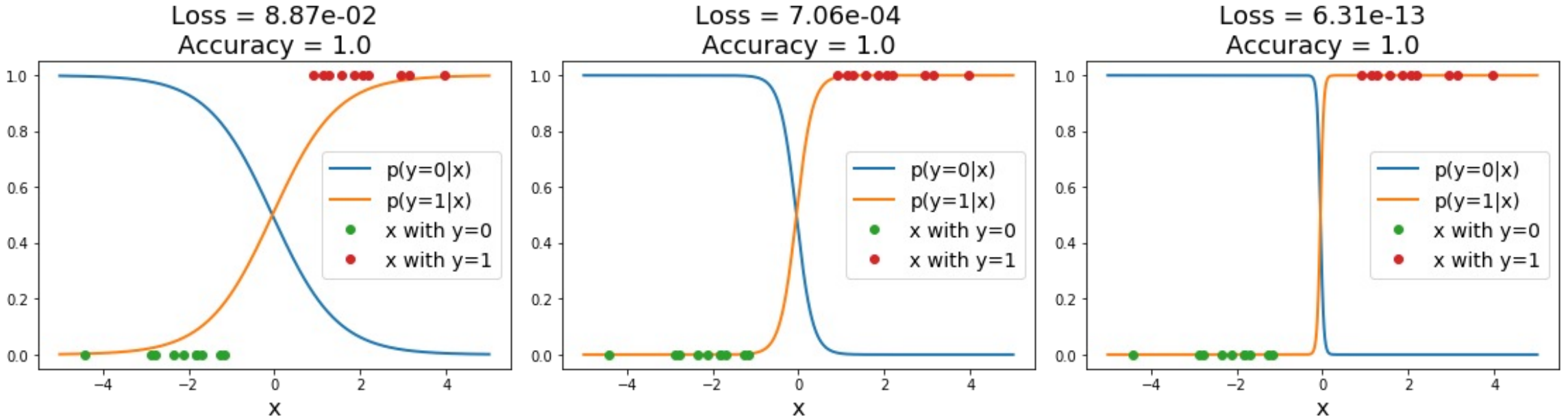
A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data

Example: Linear classifier with 1D inputs, 2 classes, softmax loss

$$s_i = w_i x + b_i$$

$$p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$

$$L = -\log(p_y)$$



Both models have perfect accuracy on train data!

Low loss, but unnatural "cliff" between training points

Regularization: Beyond Training Error

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}$$

Data loss: Model predictions should match training data

Regularization: Beyond Training Error

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

λ is a hyperparameter giving regularization strength

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data, reduce the complexity of the learned model

Regularization: Beyond Training Error

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

λ is a hyperparameter giving regularization strength

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_{k,l} W_{k.l}^2$

L1 regularization: $R(W) = \sum_{k,l} |W_{k,l}|$

Regularization: Beyond Training Error

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

λ is a hyperparameter giving regularization strength

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_{k,l} W_{k.l}^2$

L1 regularization: $R(W) = \sum_{k,l} |W_{k,l}|$

More complex:

Dropout

Batch normalization

Cutout, Mixup, Stochastic depth, etc…

Regularization: Prefer Simpler Models

Example: Linear classifier with 1D inputs, 2 classes, softmax loss

$$s_i = w_i x + b_i \qquad p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$

$$L = -\log(p_y) + \boldsymbol{\lambda \sum_i w_i^2}$$

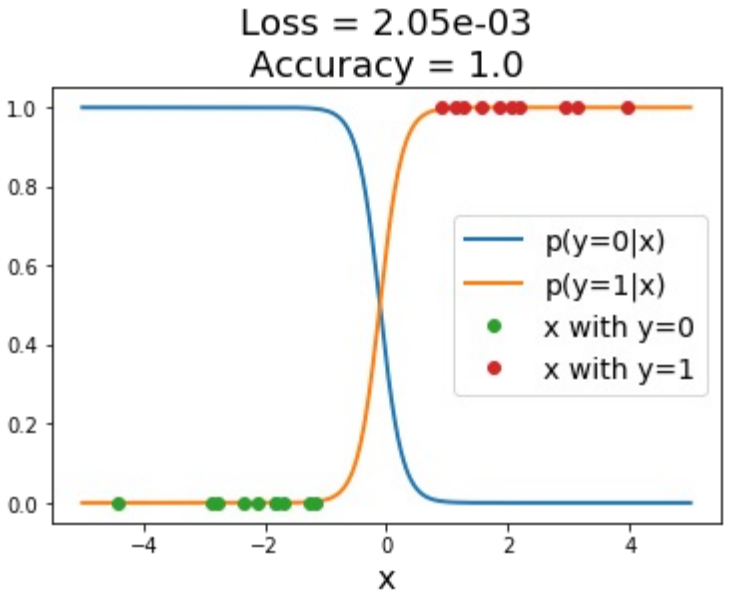
Regularization: Prefer Simpler Models

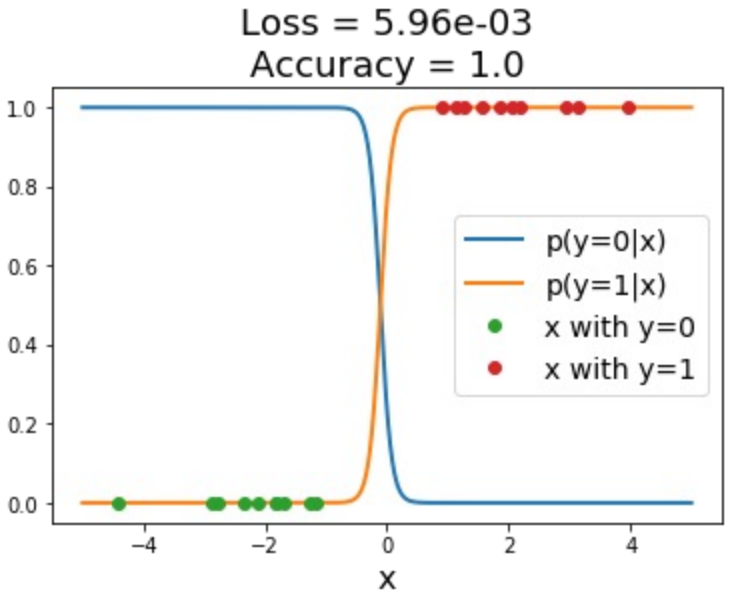
Example: Linear classifier with 1D inputs, 2 classes, softmax loss

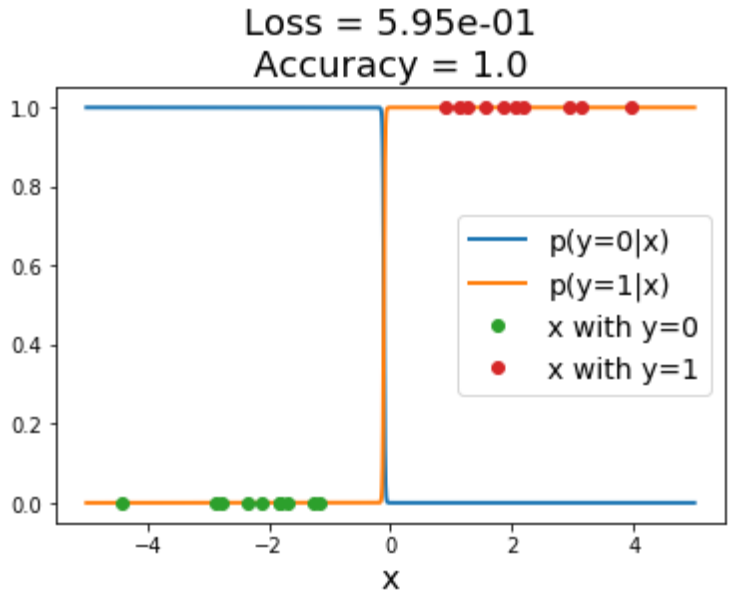
$$s_i = w_i x + b_i \qquad p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$

$$L = -\log(p_y) + \lambda \sum_i w_i^2$$

Regularization term causes loss to **increase** for model with sharp cliff







Regularization: Expressing Preferences

L2 Regularization

$$R(W) = \sum_{k,l} W_{k.l}^2$$

$$x = [1, 1, 1, 1]$$
$$w_1 = [1, 0, 0, 0]$$
$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

Same predictions, so data loss will always be the same

Regularization: Expressing Preferences

L2 Regularization

$$R(W) = \sum_{k,l} W_{k.l}^2$$

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

L2 regularization prefers weights to be "spread out"

$$w_1^T x = w_2^T x = 1$$

Same predictions, so data loss will always be the same

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization

$$L(w) = L_{data}(w) + \boldsymbol{\lambda |w|^2}$$
$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + \boldsymbol{2\lambda w_t}$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

Loshchilov and Hutter, "Decoupled Weight Decay Regularization", ICLR 2019

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization

$$L(w) = L_{data}(w) + \boldsymbol{\lambda |w|^2}$$
$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + \boldsymbol{2\lambda w_t}$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

Weight Decay

$$L(w) = L_{data}(w)$$
$$g_t = \nabla L_{data}(w_t)$$
$$s_t = optimizer(g_t) + \boldsymbol{2\lambda w_t}$$
$$w_{t+1} = w_t - \alpha s_t$$

Loshchilov and Hutter, "Decoupled Weight Decay Regularization", ICLR 2019

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization

$$L(w) = L_{data}(w) + \boldsymbol{\lambda |w|^2}$$
$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + \boldsymbol{2\lambda w_t}$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

Weight Decay

$$L(w) = L_{data}(w)$$
$$g_t = \nabla L_{data}(w_t)$$
$$s_t = optimizer(g_t) + \boldsymbol{2\lambda w_t}$$
$$w_{t+1} = w_t - \alpha s_t$$

Loshchilov and Hutter, "Decoupled Weight Decay Regularization", ICLR 2019

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization

$$L(w) = L_{data}(w) + \boldsymbol{\lambda |w|^2}$$
$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + \boldsymbol{2\lambda w_t}$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

But they are not the same for adaptive methods (AdaGrad, RMSProp, Adam, etc)

Weight Decay

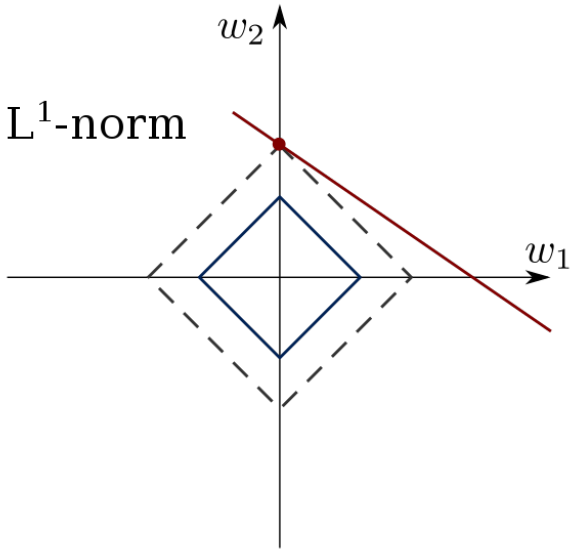
$$L(w) = L_{data}(w)$$
$$g_t = \nabla L_{data}(w_t)$$
$$s_t = optimizer(g_t) + \boldsymbol{2\lambda w_t}$$
$$w_{t+1} = w_t - \alpha s_t$$

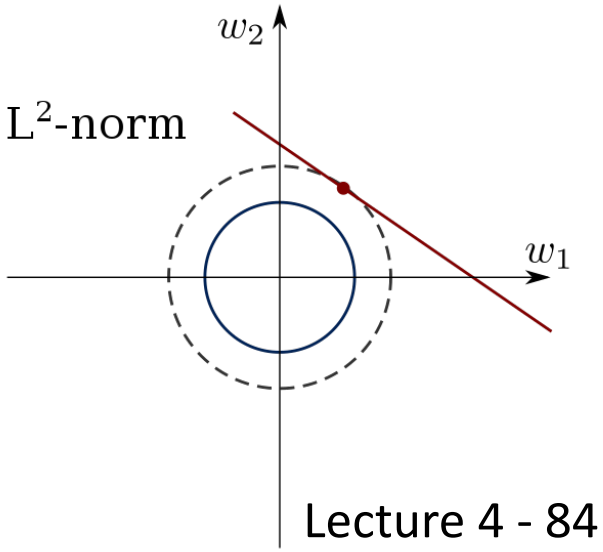
Loshchilov and Hutter, "Decoupled Weight Decay Regularization", ICLR 2019

L1 Regularization versus L2 Regularization

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \sum_{k,l} W_{k.l}^2$$

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \sum_{k,l} |W_{k.l}|$$

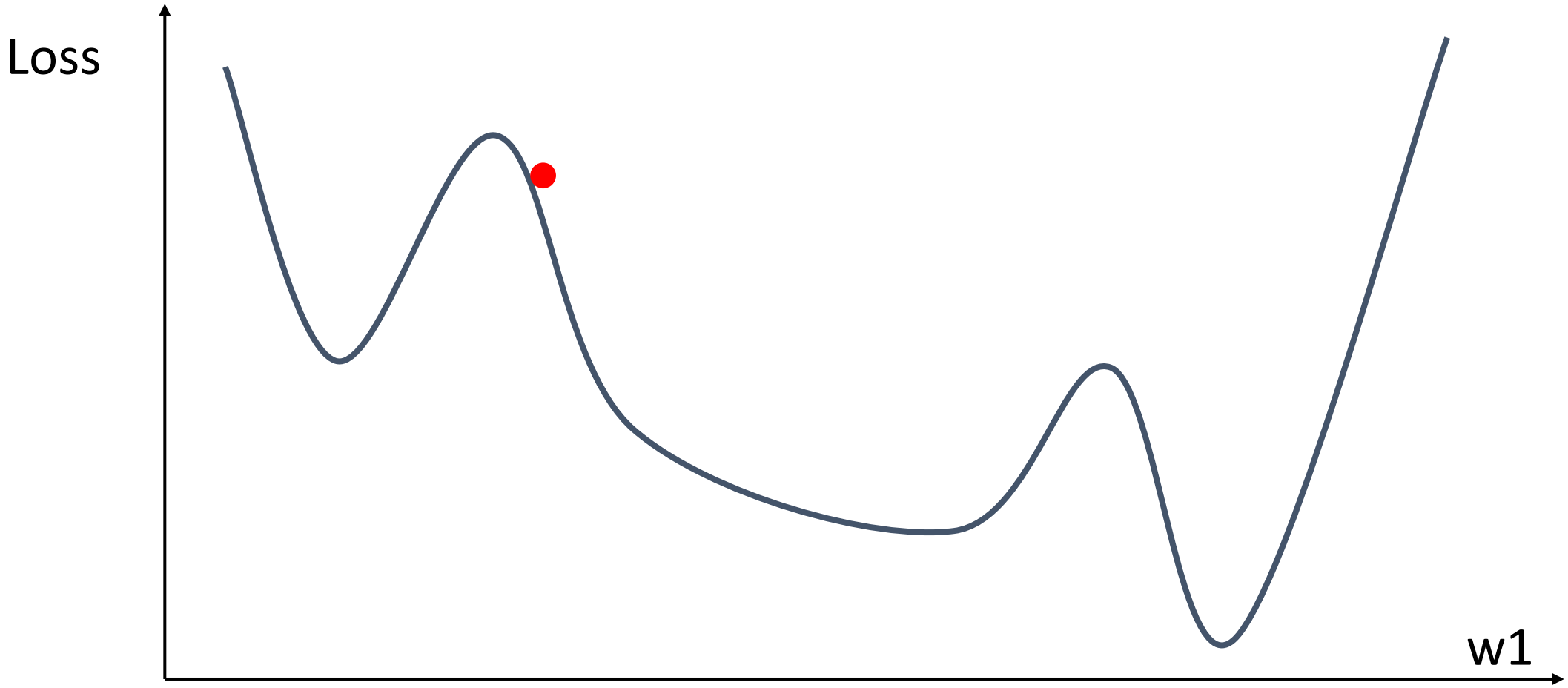




L1 regularization leads to more sparse weights

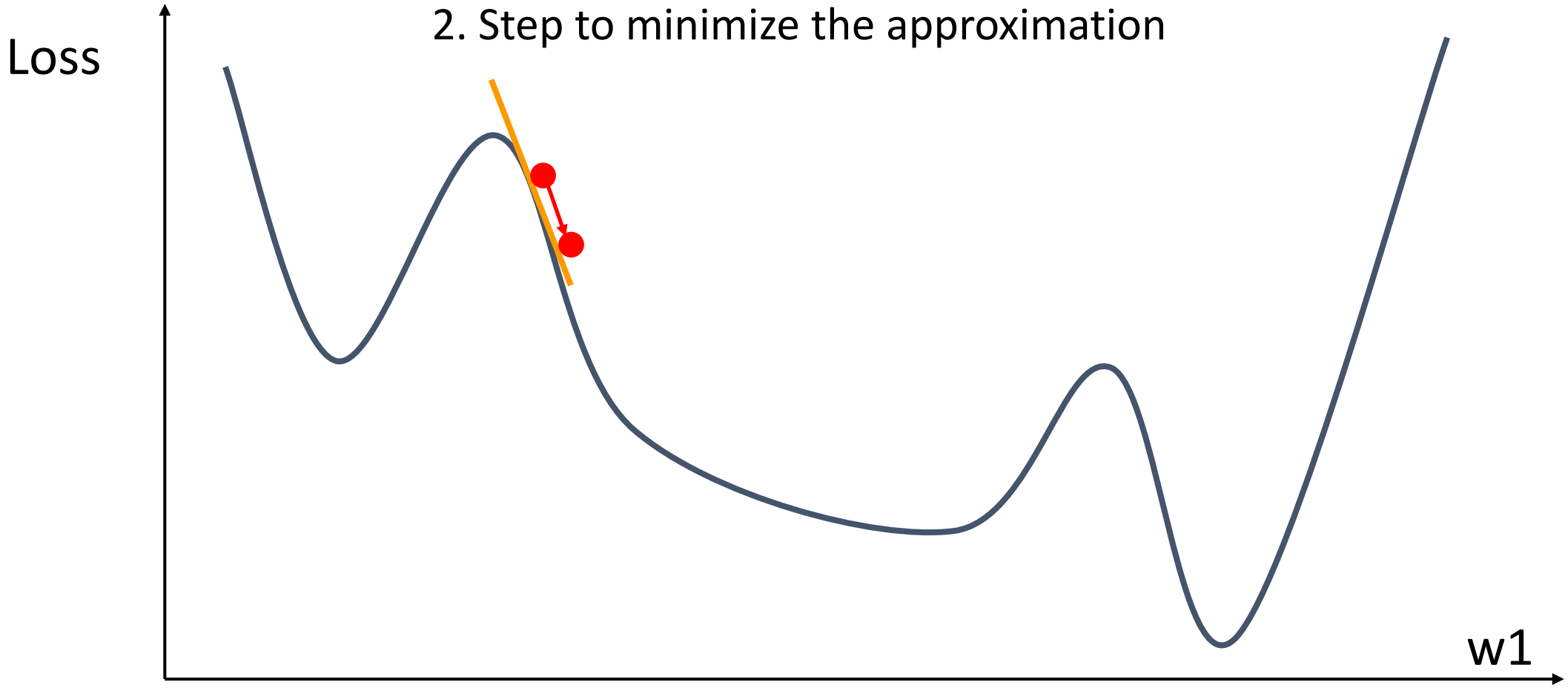
https://en.wikipedia.org/wiki/Lasso_(statistics)

So far: First-Order Optimization



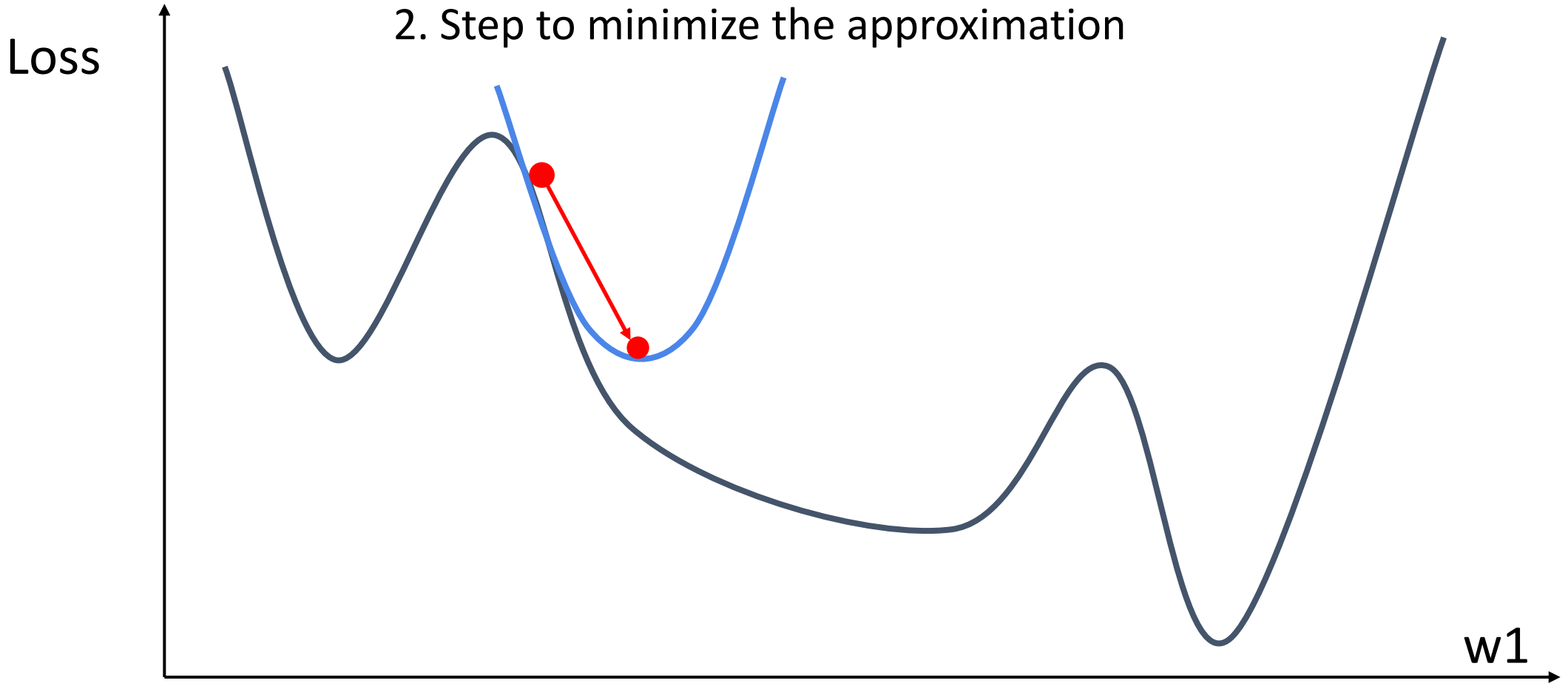
So far: **First-Order** Optimization

1. Use gradient to make linear approximation
2. Step to minimize the approximation



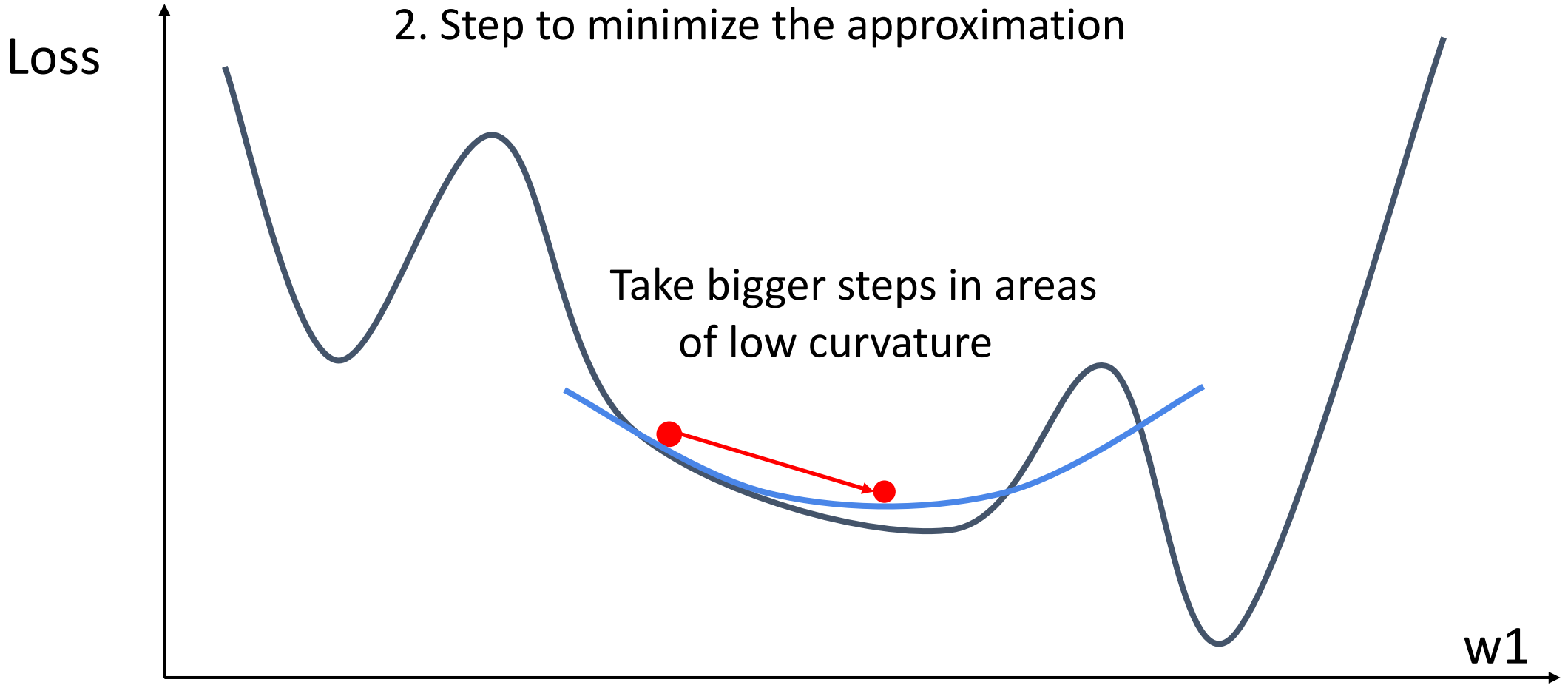
Second-Order Optimization

1. Use gradient and Hessian to make quadratic approximation
2. Step to minimize the approximation



Second-Order Optimization

1. Use gradient and Hessian to make quadratic approximation
2. Step to minimize the approximation



Second-Order Optimization

Second-Order Taylor Expansion:

$$L(w) \approx L(w_0) + (w - w_0)^\intercal \nabla_w L(w_0) + \frac{1}{2}(w - w_0)^\intercal \mathbf{H}_w L(w_0)(w - w_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

Second-Order Optimization

Second-Order Taylor Expansion:

$$L(w) \approx L(w_0) + (w - w_0)^\intercal \nabla_w L(w_0) + \frac{1}{2}(w - w_0)^\intercal \mathbf{H}_w L(w_0)(w - w_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

Q: Why is this impractical?

Second-Order Optimization

Second-Order Taylor Expansion:

$$L(w) \approx L(w_0) + (w - w_0)^\intercal \nabla_w L(w_0) + \frac{1}{2}(w - w_0)^\intercal \mathbf{H}_w L(w_0)(w - w_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

Q: Why is this impractical?

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

Second-Order Optimization

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

- Quasi-Newton methods (**BGFS** most popular):
  instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).

- **L-BFGS** (Limited memory BFGS):
  Does not form/store the full inverse Hessian.

Second-Order Optimization: L-BFGS

- **Usually works very well in full batch, deterministic mode** i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Le et al, “On optimization methods for deep learning, ICML 2011”
Ba et al, “Distributed second-order optimization using Kronecker-factored approximations”, ICLR 2017

In practice:

- **Adam** is a good default choice in many cases
  SGD+Momentum can outperform Adam but may require more tuning

- If you can afford to do full batch updates, then try out **L-BFGS** (and don't forget to disable all sources of noise)

Summary: Use gradient descend to find a good W

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

Loss function consists of **data loss** to fit the training data and **regularization** to prevent overfitting

Then use optimizer on $L(W)$

Summary

1. Use **Linear Models** for image classification problems
2. Use **Loss Functions** to express preferences over different choices of weights
3. Use **Stochastic Gradient Descent** to minimize our loss functions and train the model
4. Use **Regularization** to prevent overfitting to training data

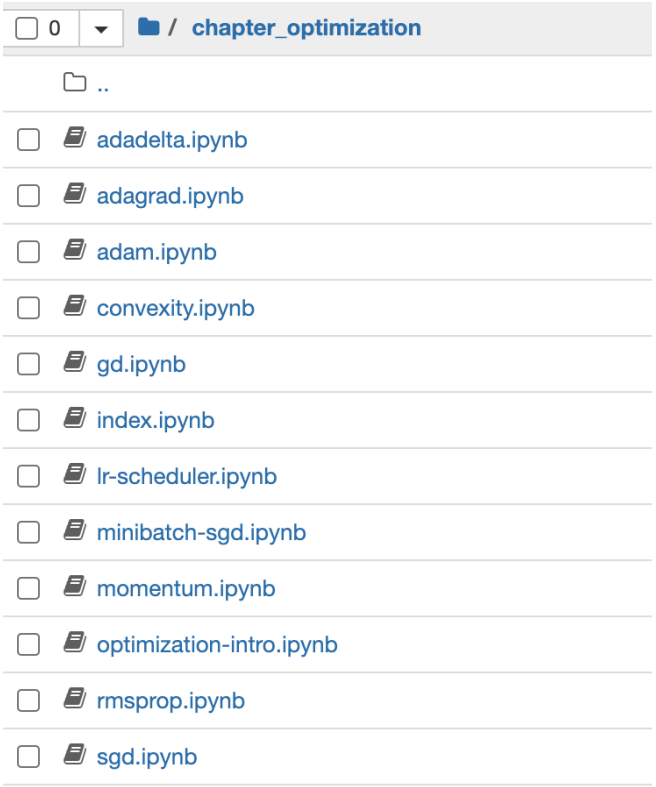
$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$ Softmax

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

Reading and Practice

D2L.ai Chapter 11. Optimization Algorithms.
Notebooks: chapter_optimization



Next time:
Neural Networks