Deep Learning 1

2025-2026 – Pascal Mettes

Lecture 3

Deep learning optimization I

Previous lecture

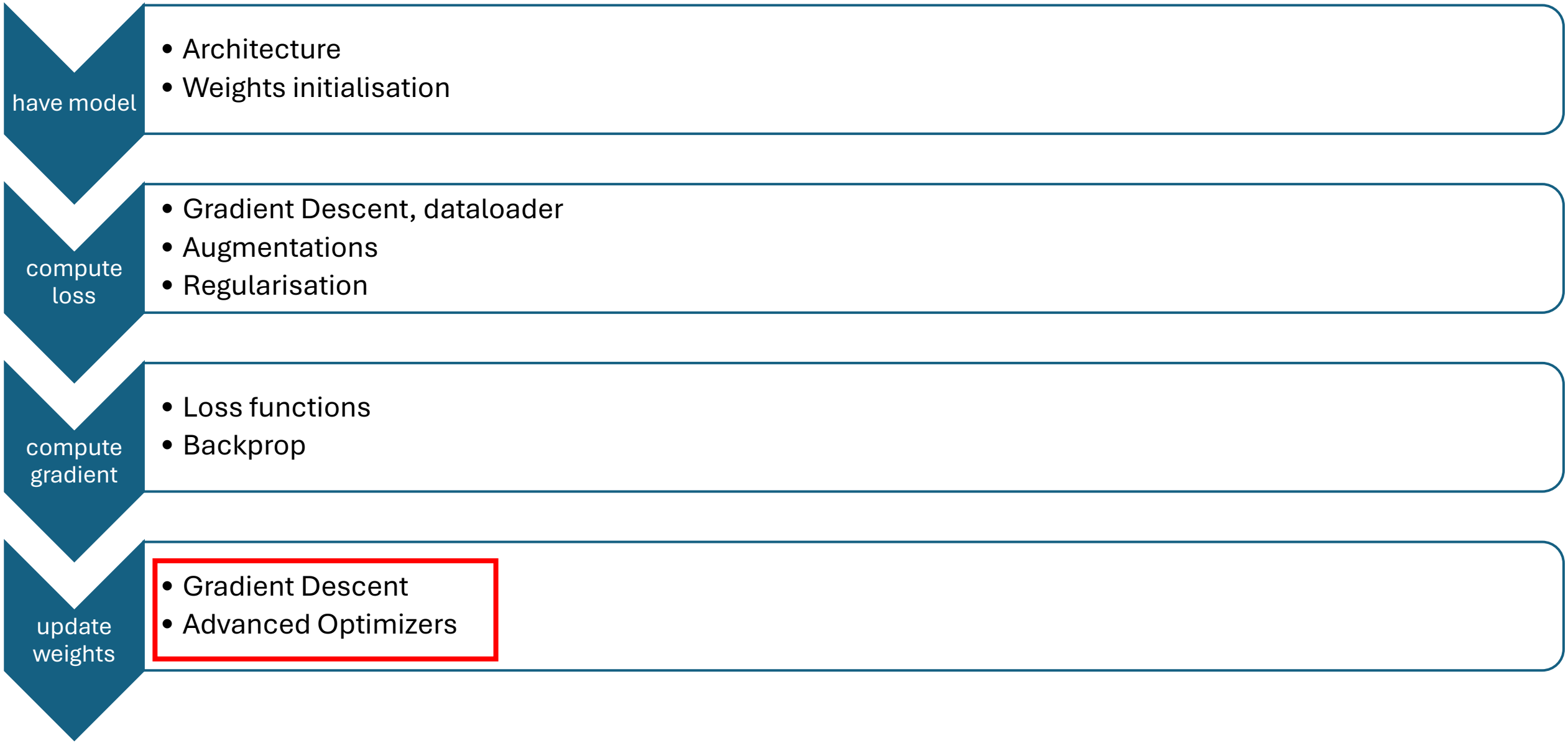
Lecture	Title
1	Intro and history of deep learning
3	Deep learning optimization I
5	Convolutional deep learning
7	Graph deep learning
9	Multi-modal deep learning
11	What doesn't work in deep learning
13	Q&A

Lecture	Title
2	AutoDiff
4	Deep learning optimization II
6	Attention-based deep learning
8	From supervised to unsupervised deep learning
10	Generative deep learning
12	Non-Euclidean deep learning
14	Deep learning for videos

This lecture

Stochastic gradient descent

Advanced optimizers


have model
• Architecture
• Weights initialisation
compute loss
• Gradient Descent, dataloader
• Augmentations
• Regularisation
compute gradient
• Loss functions
• Backprop
update weights
• Gradient Descent
• Advanced Optimizers

Optimization versus learning

Optimization

- given a parametric definition of model and a set of data, we want to discover the optimal parameter that minimize a certain objective function, given some data.
- E.g., find the optimal flight schedule given resources and population.

Learning

- We have observed and unobserved data.
- Reduce errors on the observed data (training data) to *generalize* to unseen data (test data).
- The goal is to reduce the generalization error.

Minimizing risk

We want to optimize on observed data.

Minimizing a cost function, with extra regularizations

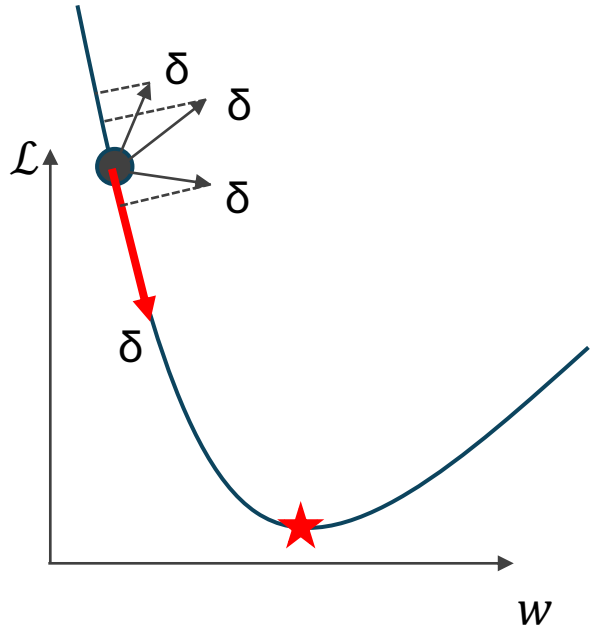
$$\min_{\mathbf{w}} \mathbb{E}_{\boldsymbol{x},\mathrm{y}\sim p_{data}}[\mathcal{L}(\, f(\boldsymbol{x},\boldsymbol{w}),y)] + \lambda\Omega(\boldsymbol{w})$$

where $\hat{y} = f(\boldsymbol{x},\boldsymbol{w}) = h_L \circ h_{L-1} \circ \cdots \circ h_1\,(\boldsymbol{x})$ is the prediction and each h_l comes with parameters $\boldsymbol{w}_l$.

In simple words: (1) predictions are not too wrong, while (2): not being "too geared" towards the observed data.

Problem: the true distribution p_{data} is not available.

Empirical risk minimization



In practice having p_{data} is not possible.

We only have a training set of data

$$\min_{\mathbf{w}} \mathbb{E}_{\boldsymbol{x},\mathrm{y}\sim\hat{p}_{data}}[\mathcal{L}(f(\boldsymbol{x},\boldsymbol{w}),y)] + \lambda\Omega(\boldsymbol{w})$$

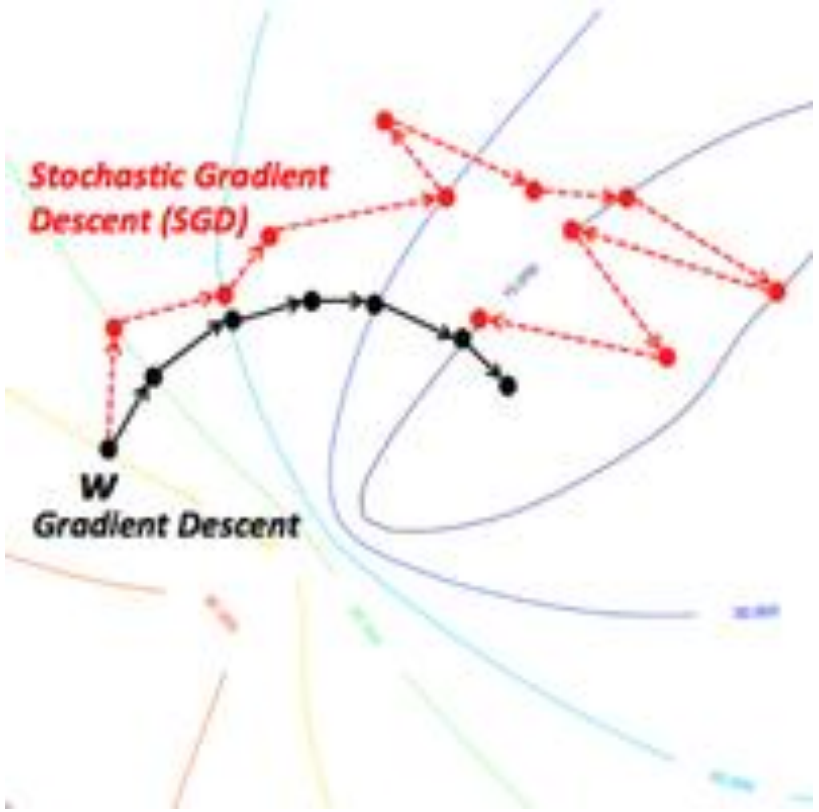
with $\hat{p}_{data}$ the empirical data distribution, defined by a set of training examples.

To minimize any function, we take a step δ. Our best bet: the (negative) gradient

$$-\sum \frac{d}{dw}\mathcal{L}(f(\boldsymbol{x},\boldsymbol{w}),y)$$

Gradient descent based on optimization.

Stochastic gradient descent



The problem is that the training set is too big to be used on GPU during the training phase

Instead of using the entire dataset to calculate gradients, perform parameter update for each mini-batch.

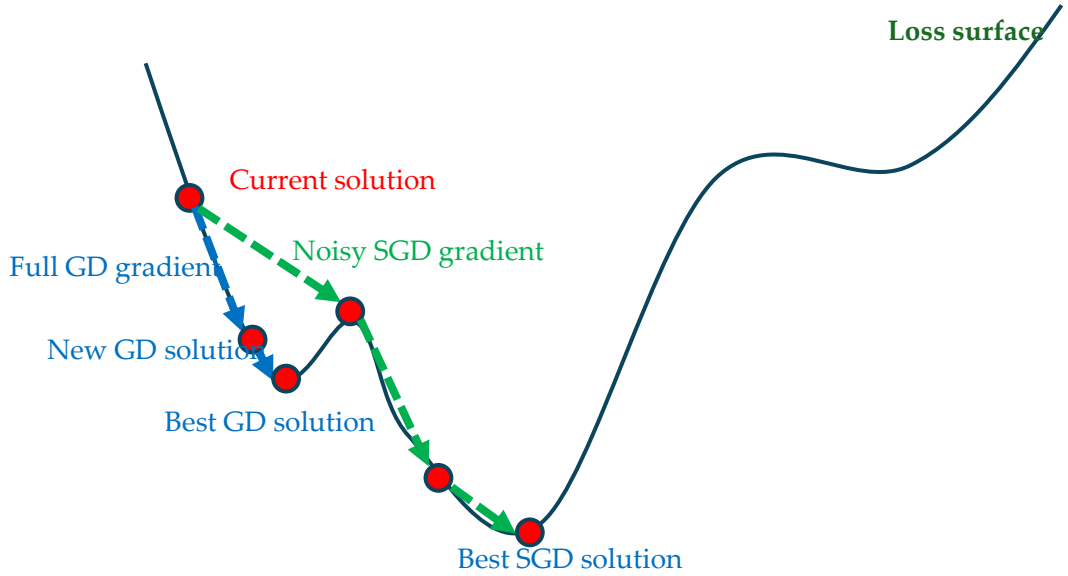
Properties of SGD

Randomness actually reduces overfitting.

Reshuffling is important!

One epoch = go through all mini-batches.

Be careful to balance class/data per batch.




```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)
```

On batch size

Large batch size: more accurate estimation of the gradient.

Very small batch size: underutilizes hardware, but can act as regularizer.

General rule: batch size and learning rate are coupled (double BS = double LR)

(*Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour*. Goyal et al. 2017: Batchsize of 8K)

Guideline: use the largest batch size that fits on the GPU and is a power of 2.

Why does mini-batch SGD work?

Gradient descent is already an approximation; the true data distribution is unknown.

Reduced sample size does not imply reduced gradient quality.

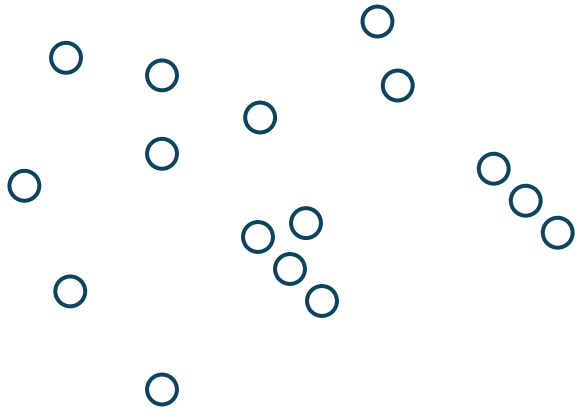
The training samples may have lots of noises or outliers or biases.

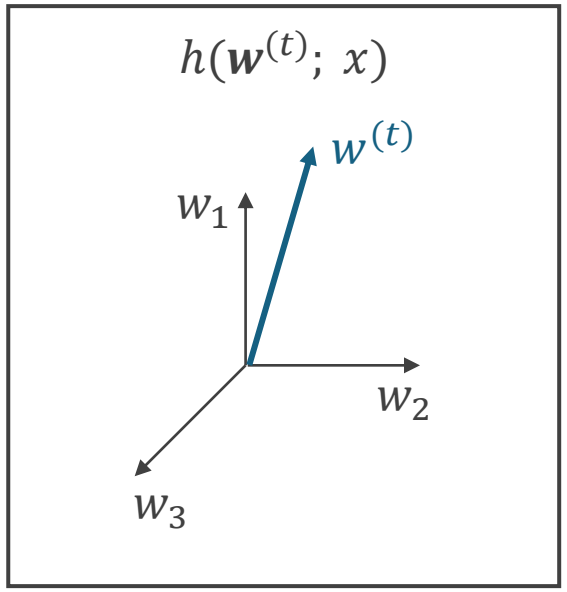
- A randomly sampled minibatch may reflect the true data generating distribution better (or worse).

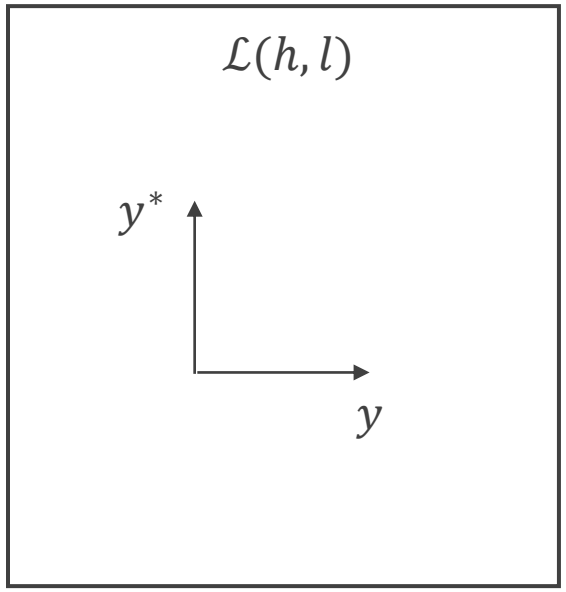
Real gradient might get stuck into a local minima.

- While *more random gradient* computed with minibatch might not.

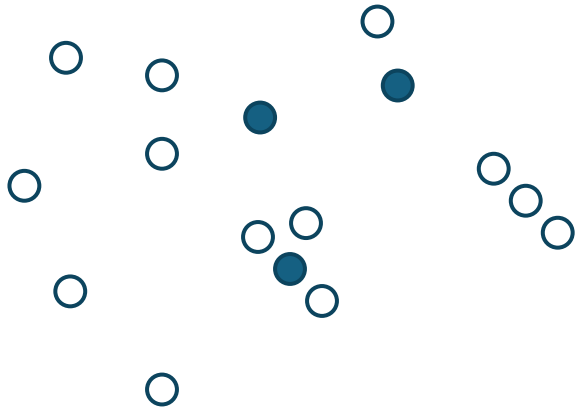
Stochastic gradient descent



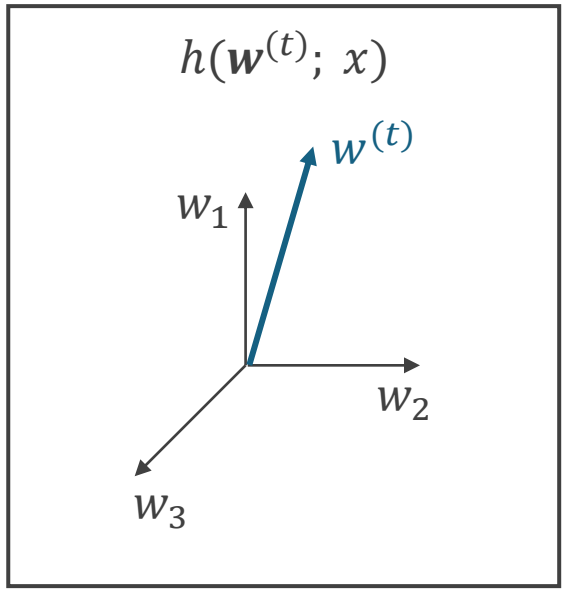


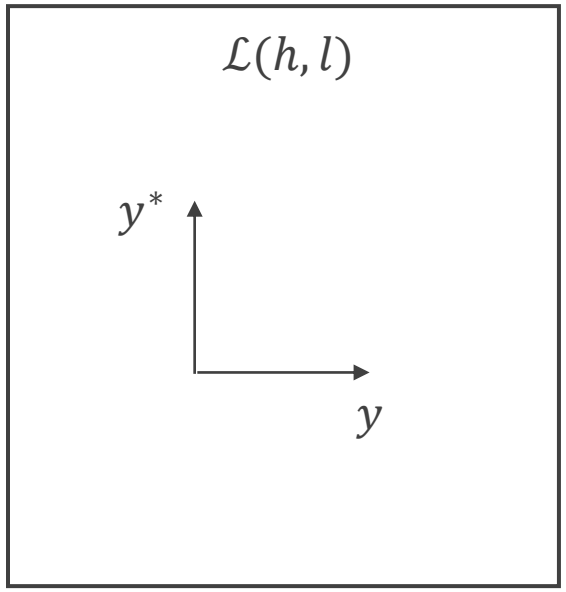


Stochastic gradient descent

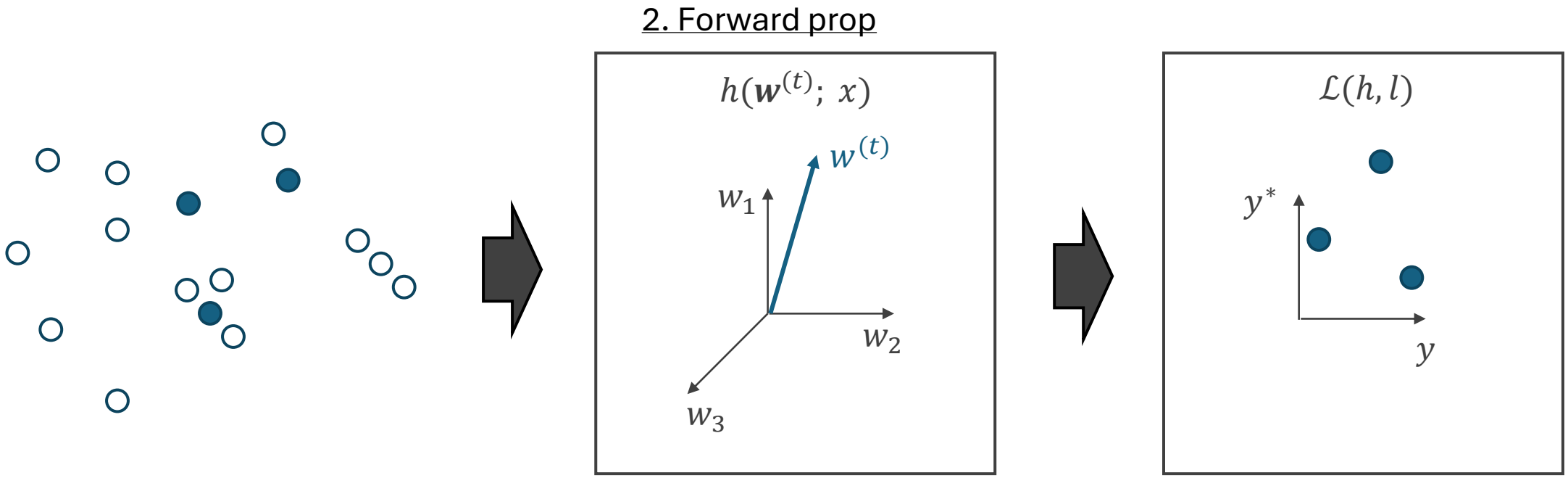


1. Sample mini-batch

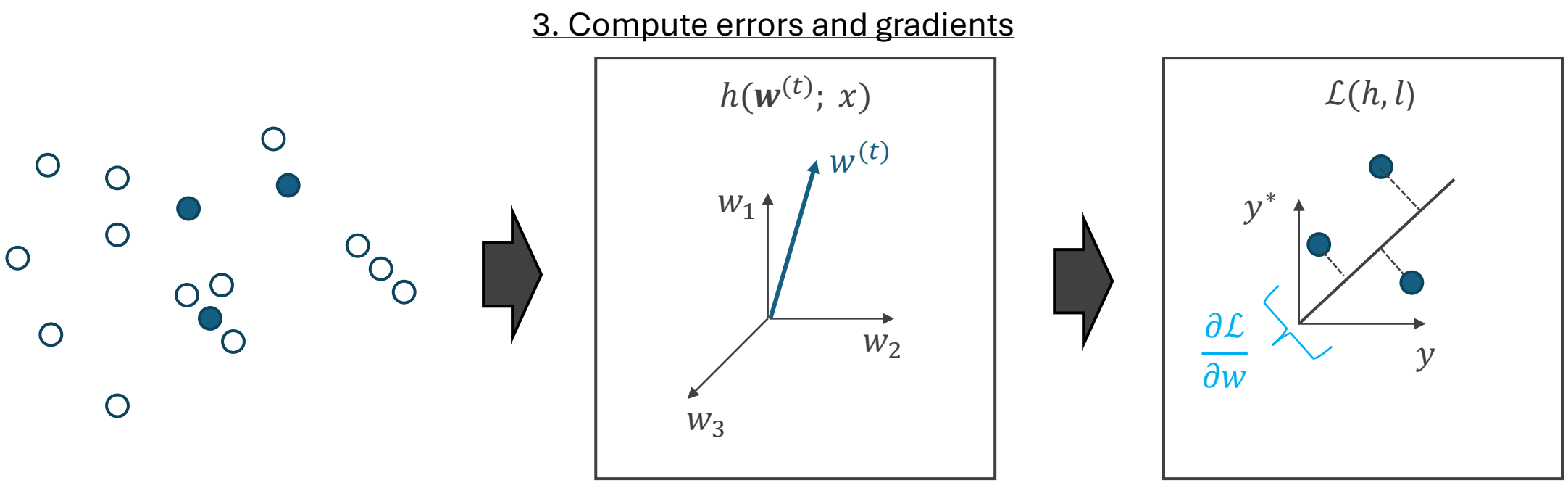




Stochastic gradient descent

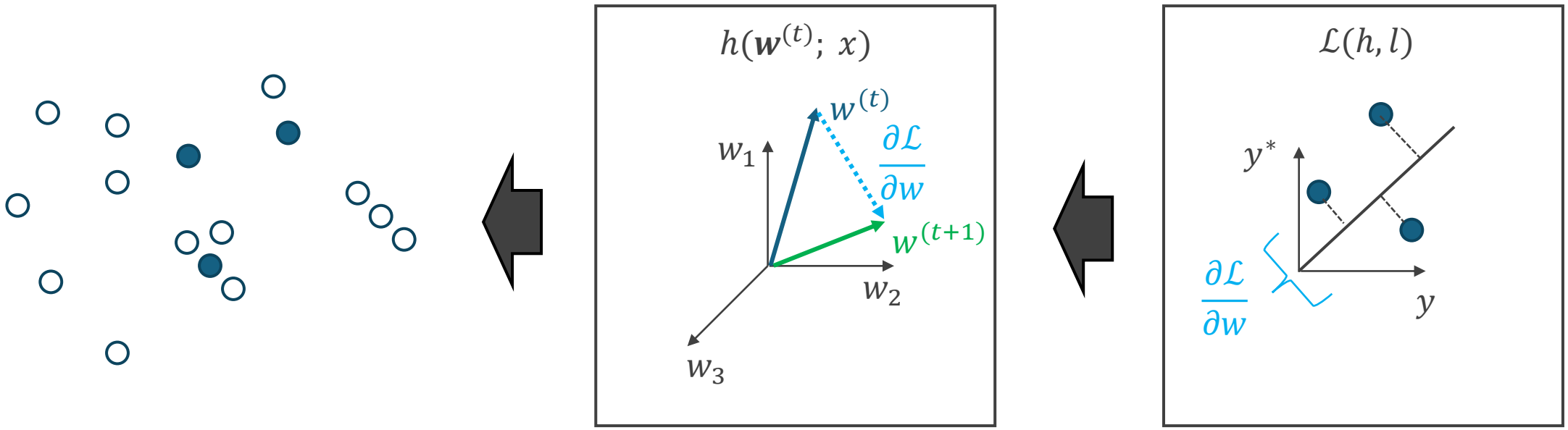


Stochastic gradient descent



Stochastic gradient descent

4. Update model parameters and repeat



In a nutshell

First, define your neural network

$$y = h_L \circ h_{L-1} \circ \cdots \circ h_1 (\boldsymbol{x})$$

where each module h_l comes with parameters $\boldsymbol{w}_l$

Finding an "optimal" neural network means minimizing a loss function

$$\boldsymbol{w}^* = \operatorname{argmin}_{\boldsymbol{w}} L(\boldsymbol{w}) = \sum_{(\boldsymbol{x},y)\in(X,Y)} \mathcal{L}(\, f(\boldsymbol{x}, \boldsymbol{w}), y) + \lambda\Omega(\boldsymbol{w})$$

Rely on stochastic gradient descent methods to obtain desired parameters

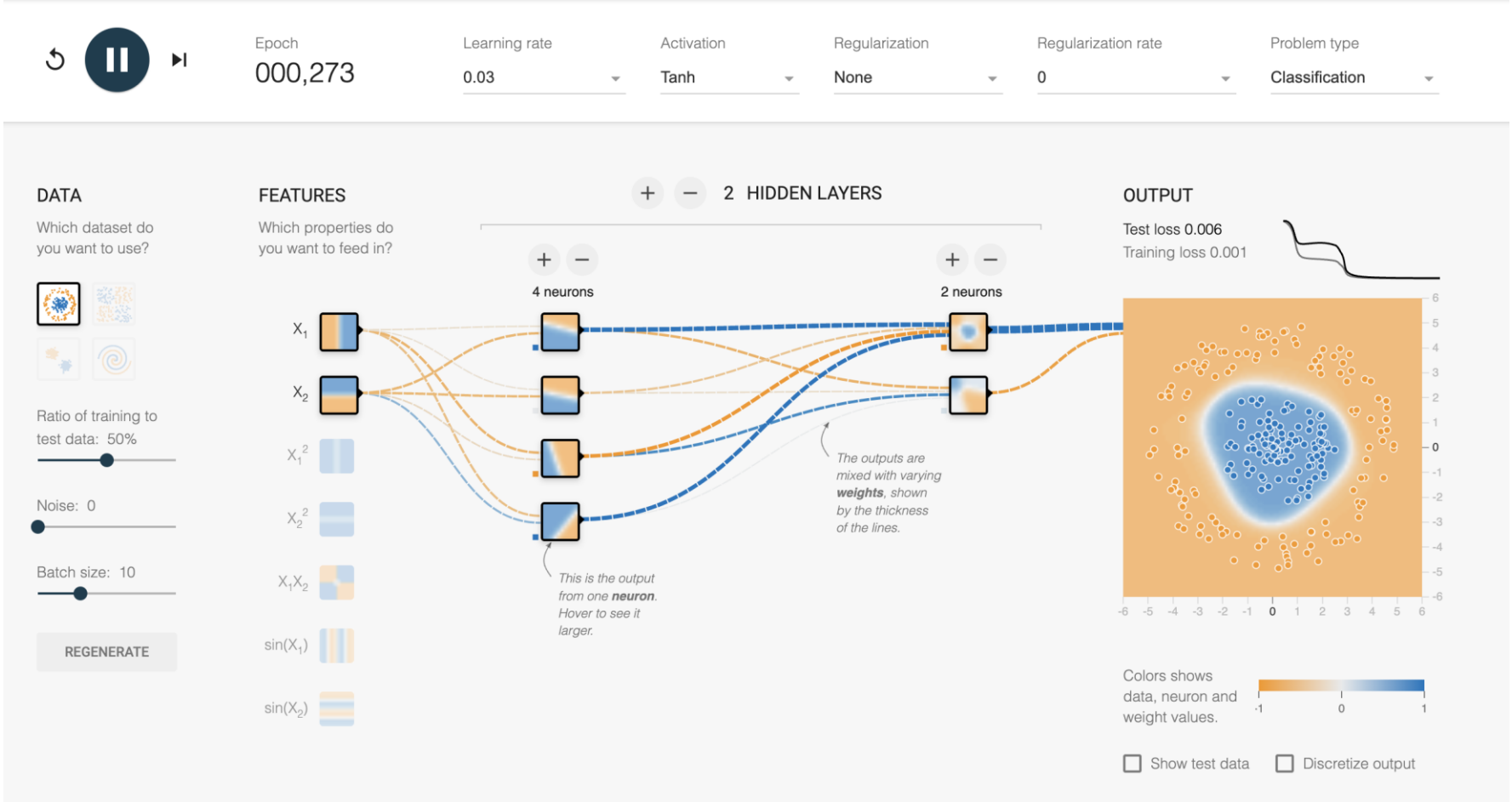
$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \frac{dL}{d\boldsymbol{w}}$$

where η is the step size or learning rate.

Gradient vs Stochastic Gradient Descent

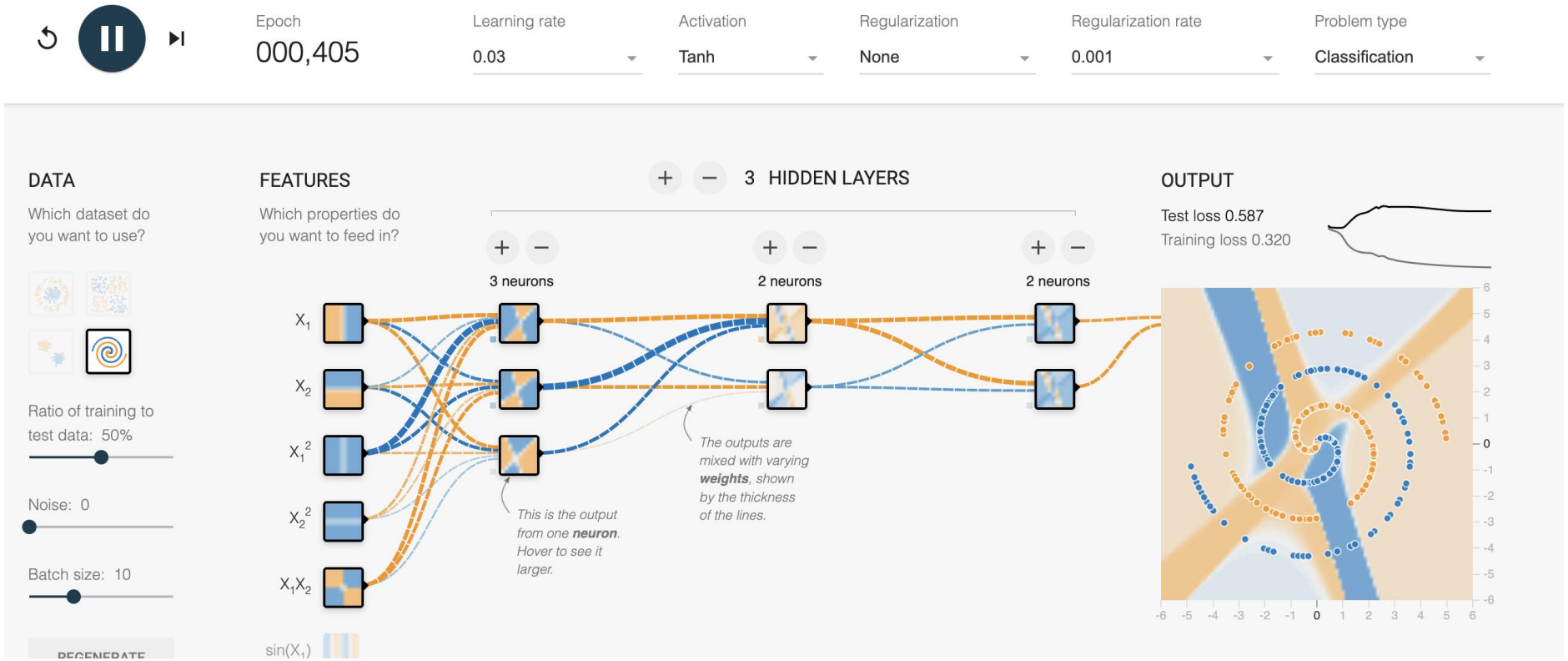
Gradient Descent	Stochastic Gradient Descent
Computes gradient using the whole training dataset	Computes gradient using a single training sample
Not suggested for huge training samples	Can be used for large training samples
Deterministic in nature	Stochastic in nature
No random shuffling of points required	Shuffling needed. More hyperparameters, e.g. batch size
Can’t escape shallow local minima easily	Can escape shallow local minima more easily
Convergence is slow	Reaches the convergence faster

Practical examples



https://playground.tensorflow.org/

Practical examples



https://playground.tensorflow.org/

Challenges of optimizing deep networks

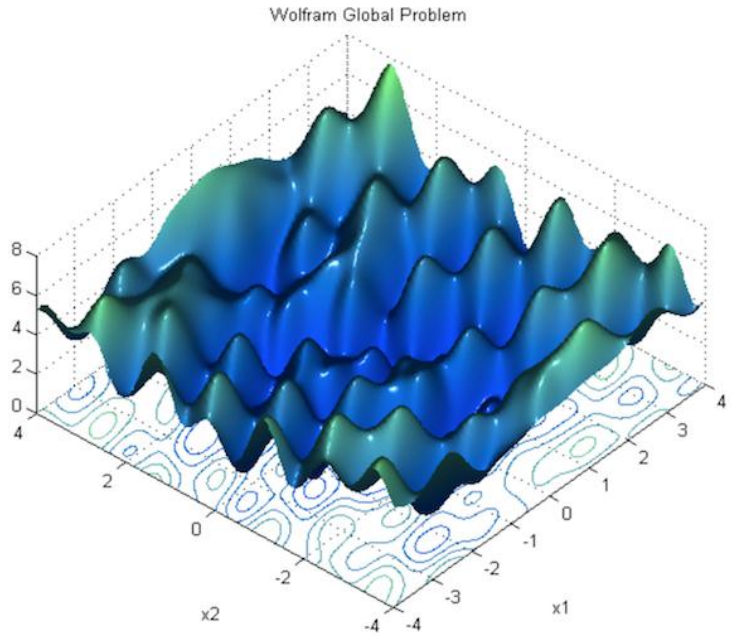
Neural network training is **non-convex** optimization.

- Involves a function which has multiple optima.
- Extremely difficult to locate the global optimum.

This raises many problems:

- How do we avoid getting stuck in local optima?
- What is a reasonable learning rate to use?
- What if the loss surface morphology changes?
- …

One of the main theory in deep learning is that every local minima is a global one, the function appears like a sinusoide (sin or cos). The optimization seems to work too well to find only local minima, so maybe they are all global.



Main challenges in optimization

1. Ill conditioning → a strong gradient might not even be good enough
2. Local optimization is susceptive to local minima
3. Ravines, plateaus, cliffs, and pathological curvatures
4. Vanishing and exploding gradients
5. Long-term dependencies

1. Ill conditioning

Hessian matrix *H*

Square matrix of second-order partial derivatives of *a scalar-valued function*.

The *Hessian* describes the local *curvature* of a function of many variables.

The Hessian matrix is symmetric.

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

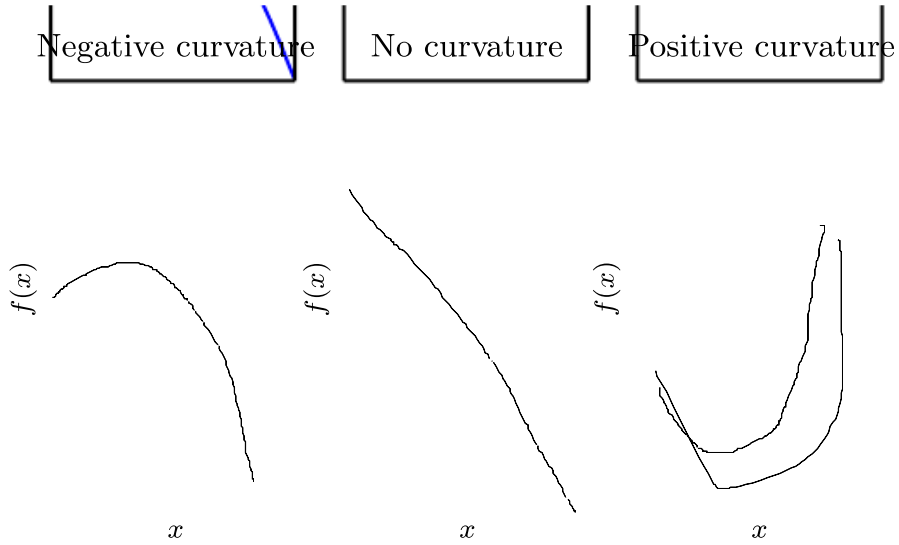
1. Ill conditioning

Curvature is determined by the second derivative

Negative curvature: cost function decreases faster than the gradient predicts.

No curvature: the gradient predicts the decrease correctly.

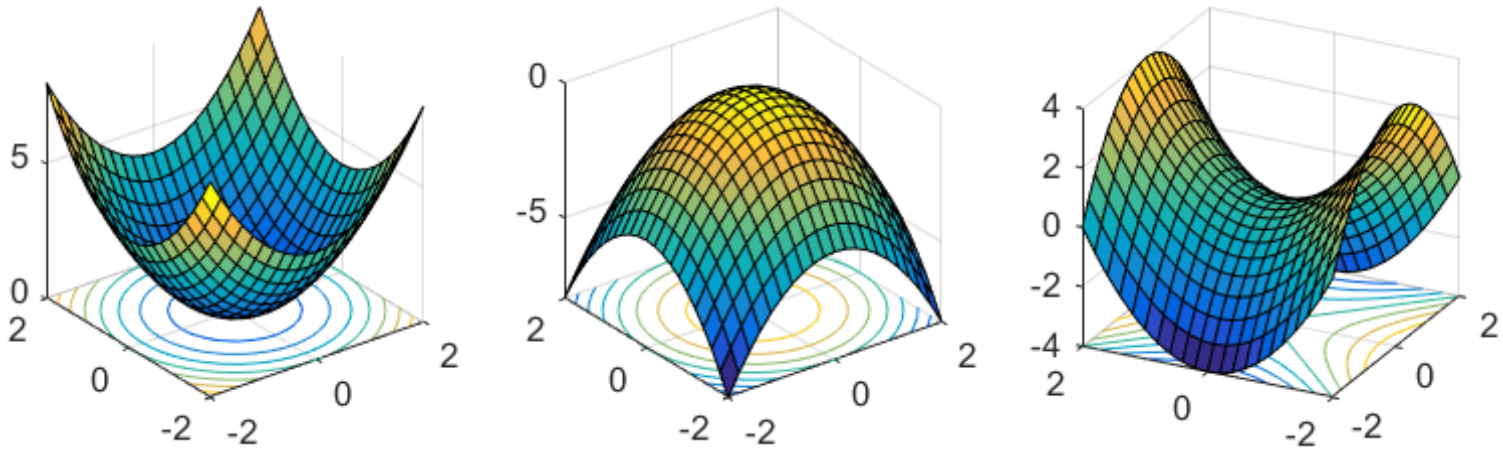
Positive curvature: the function decreases slower than expected and eventually begins to increase.



1. Ill conditioning

Critical points – Hessian matrix

- *A local minimum:* positive definite (all its eigenvalues are positive)
- *A local maximum:* negative definite (all its eigenvalues are negative)
- *A saddle point:* at least one eigenvalue is positive and at least one eigenvalue is negative. Why is this bad?



Prevalence of saddle points: https://ganguli-gang.stanford.edu/pdf/14.SaddlePoint.NIPS.pdf

1. Ill conditioning

Consider the Hessian matrix *H* has an eigenvalue decomposition.

Its condition number is $\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|$

This is the ratio of the magnitude of the largest (i) and smallest eigenvalue (j).

Measures show much the second derivatives differ from each other.

With a poor (large) condition number, gradient descent performs poorly.

- In one direction derivative increases rapidly, in another it increases slowly.
- It also makes it difficult to choose a good step size.

2. Local minima

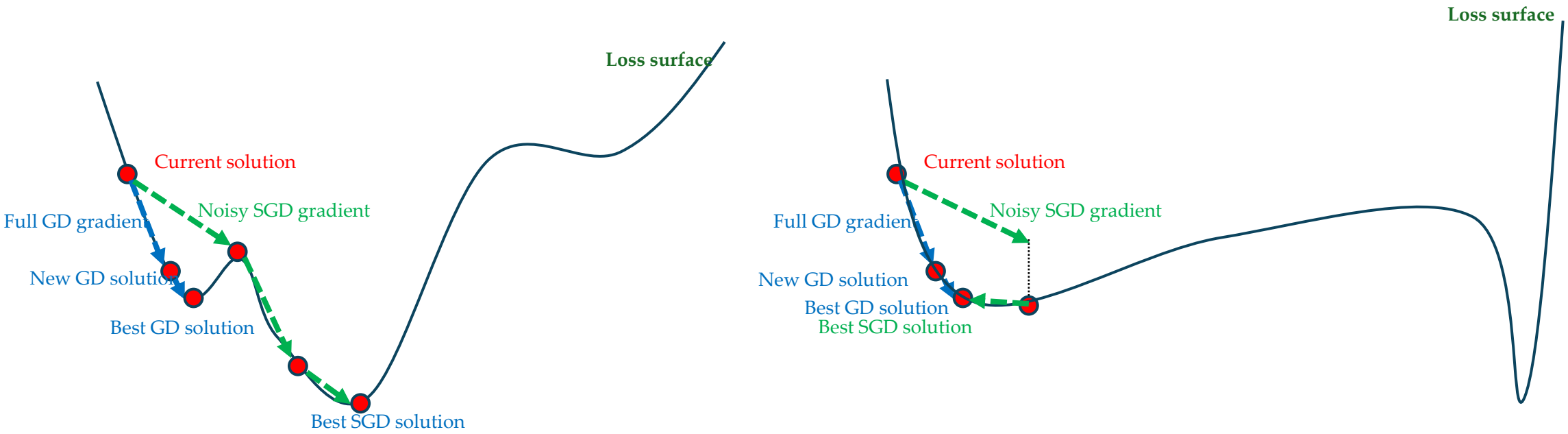
Model identifiability

- A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters.
- Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other.

Local minima can be extremely numerous

- However, all local minima from non-identifiability are equivalent in cost function value.
- *Those* local minima are not a problematic form of non-convexity.
- The other local minima (next slides) are.

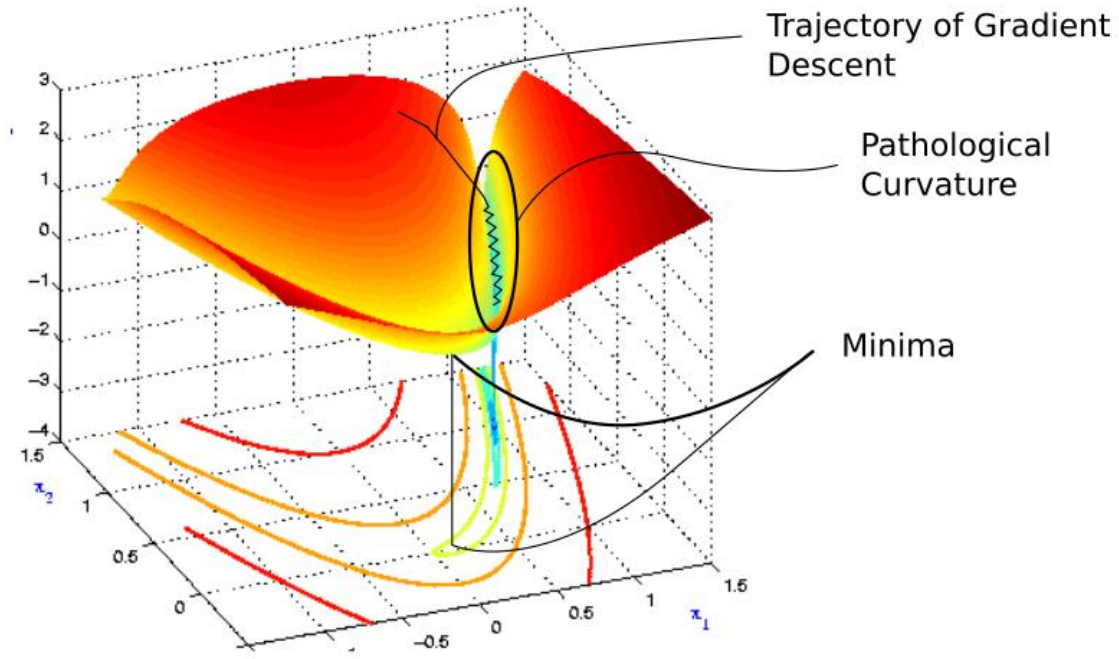
Local minima

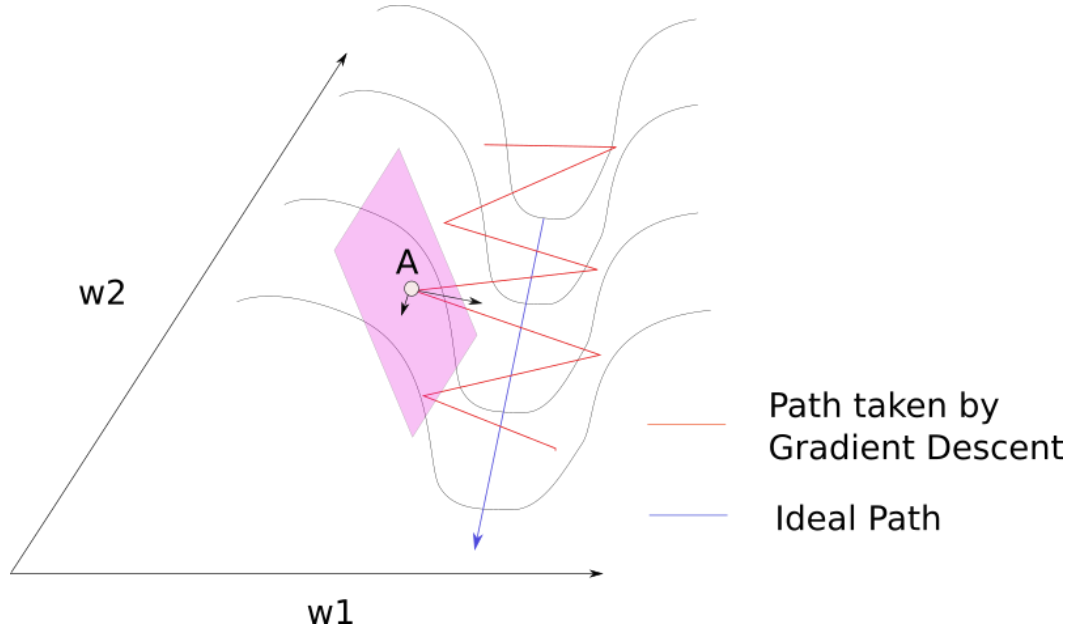


With gradient descent, we are blind to what the landscape looks like.

Picture credit: Team Paperspace

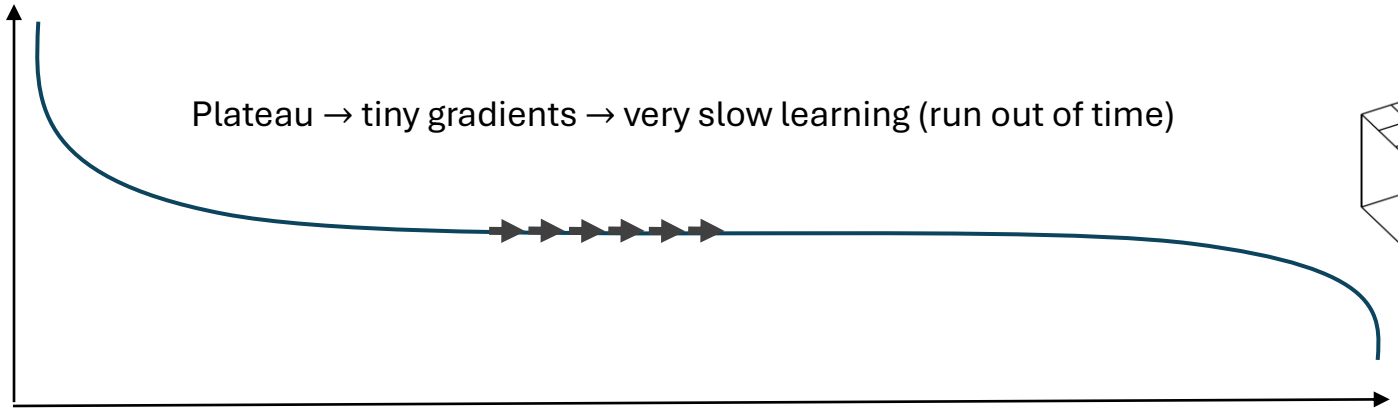
3. Ravines





Areas where the gradient is large in one direction and small in others.

3. Plateaus and flat areas



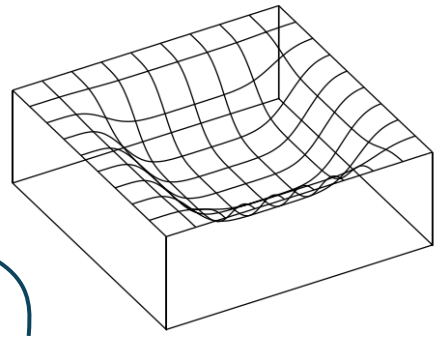


Figure 1: *Example of a "flat" minimum.*

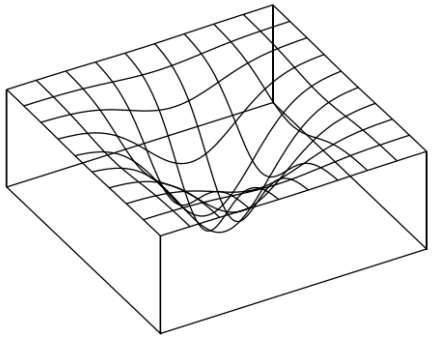


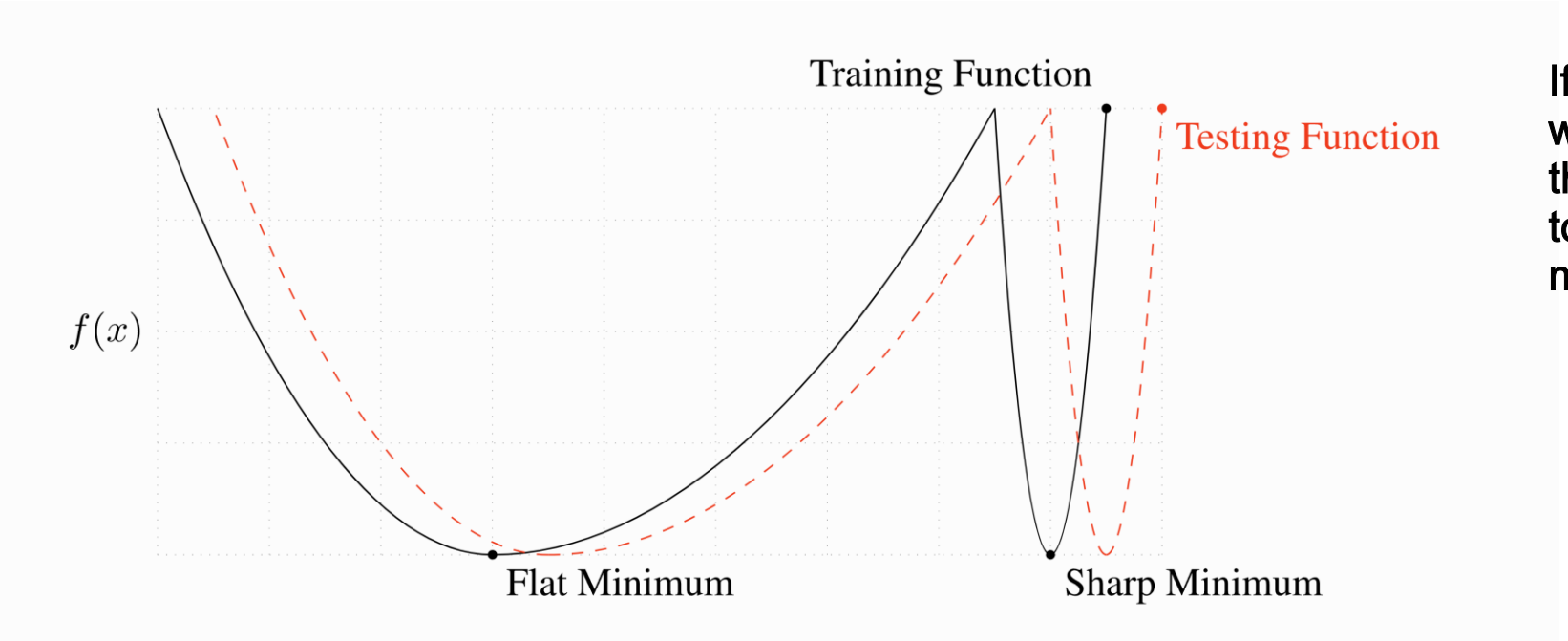
Figure 2: *Example of a "sharp" minimum.*

Link

Near zero gradients in flat areas, hence no learning.

Too step minima also a problem...

On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. Keskar et al. ICLR 2017



If the minimum is sharp we could be farther on the test minima respect to the cases with flatter minimum

Even if you miss the minimum of the test distribution slightly, still good results.

4. Cliffs and exploding gradients

Neural networks with many layers often have steep regions resembling cliffs.

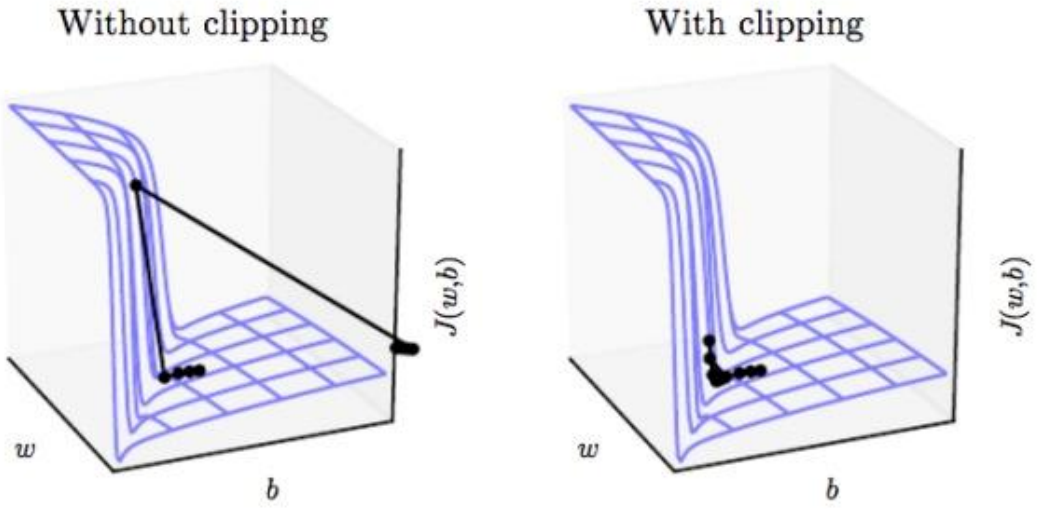
These result from the multiplication of several large weights together.

A simple trick: gradient clipping:

if |g| > η:

$$\mathbf{g} \leftarrow \frac{\eta \mathbf{g}}{\|\mathbf{g}\|}$$

Gradient clipping is a technique used to prevent the exploding gradient problem by limiting the magnitude of gradients



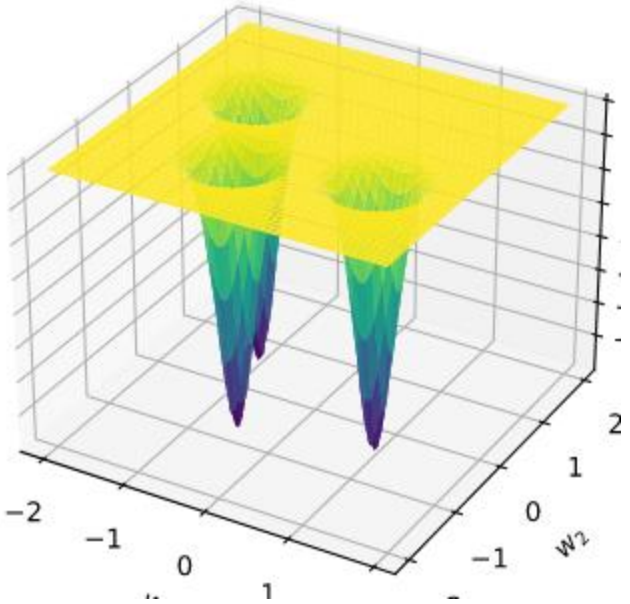
4. Flat areas and steep minima

When combining flat areas with very steep minima → very challenging

How do we even get to the area where the steep minima starts?

E.g.: temperate-scaled logits & cross-entropy: p(y|x) = softmax(logits/0.00001)

This is a problem because you don't know which direction follow since it is all flat, so it is difficult for gradient descent to spot local minima



5. Long-term dependencies

Especially for networks with many layers or recurrent neural networks.

The vanishing and exploding gradient problem

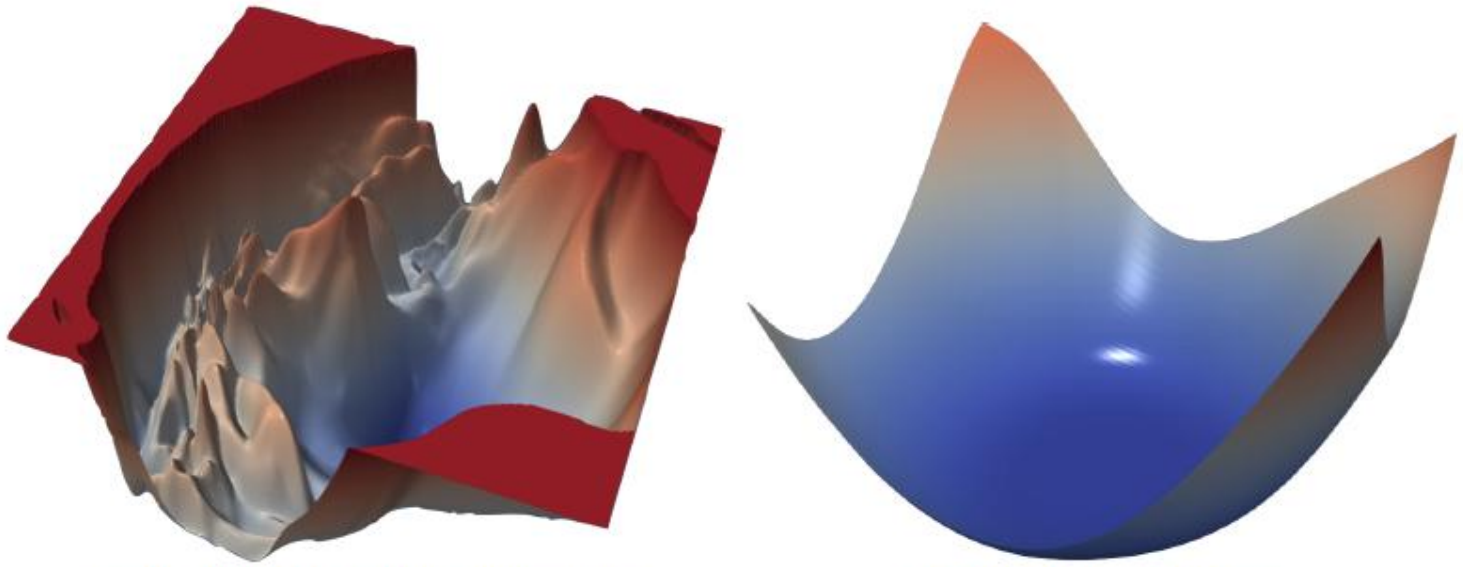
- Certain functions lead to a scaling of the gradient (potentially often).
- Vanishing gradients -> no direction to move
- Exploding gradients -> learning unstable.

For training-trajectory dependency: hard to recover from a bad start!

Is gradient descent even a good idea?

Global minima and local minima are nearby – Choromanska et al. (2015).

Architecture design and tricks have huge impact on loss landscapes (positively).



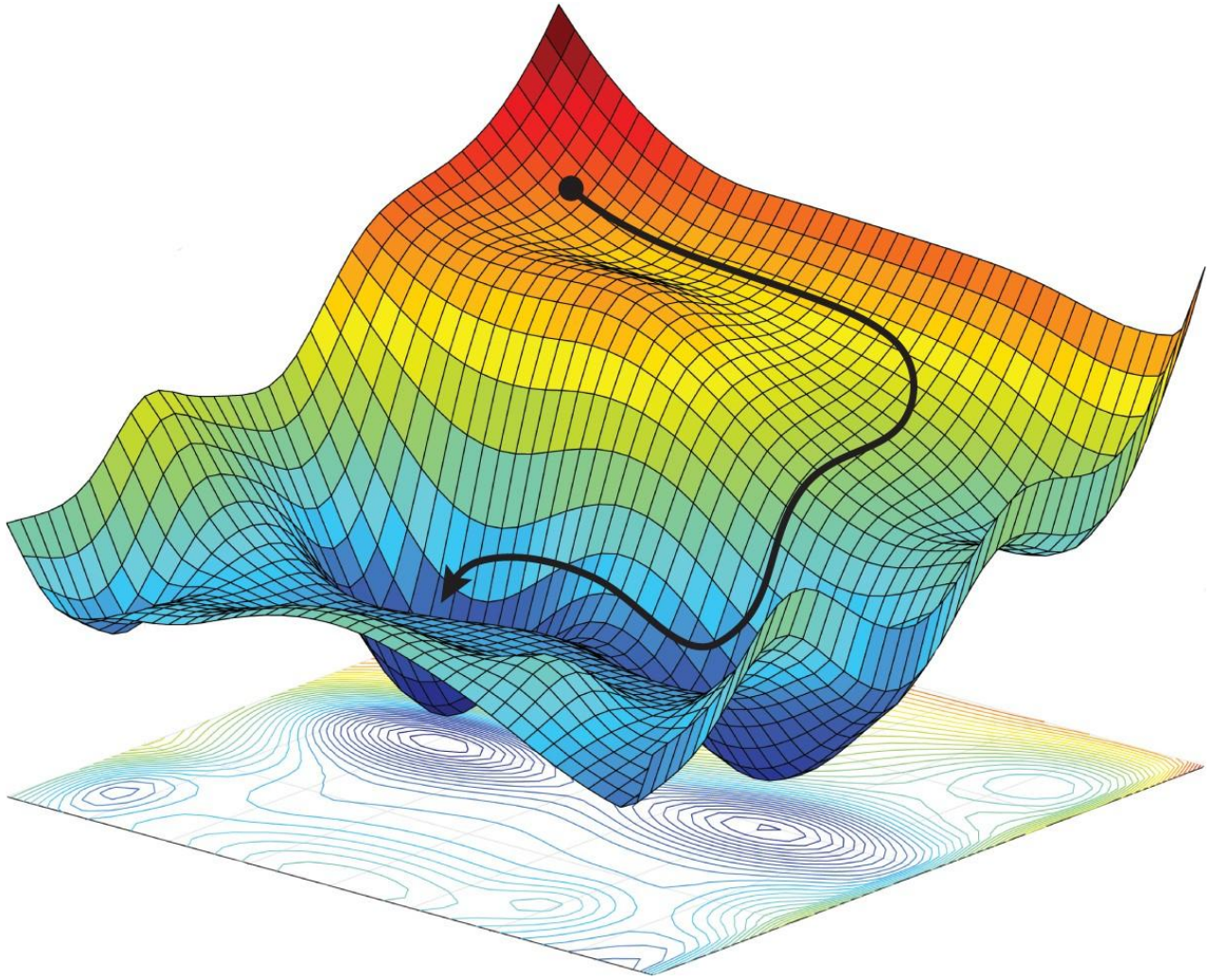
(a) ResNet-110, no skip connections

(b) DenseNet, 121 layers

Li et al. NeurIPS 2018

Break

Advanced optimizers



Gradient descent itself can be enhanced

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \frac{d\mathcal{L}(\boldsymbol{w})}{d\boldsymbol{w}}$$

Can we improve the learning rate setting?

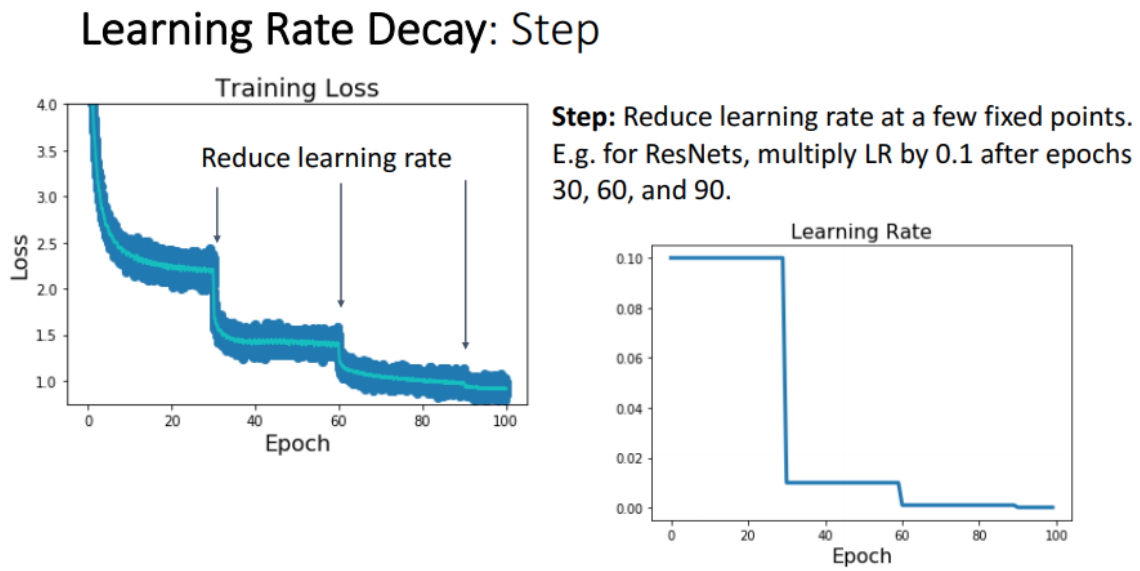
Can we get a better or more useful gradient?

Setting the learning rate

Truly an empirical endeavour, unique to each problem and dataset.

Big trick: learning rate schedulers.

Do not use a fixed learning rate but rather use a scheduler that in function of the epochs reduces the learning rate



Improving gradient descent

Stochastic Gradient Descent with momentum.

Nesterov momentum.

Stochastic Gradient Descent with adaptive learning rates.

E.g., AdaGrad, RMSProp, Adam

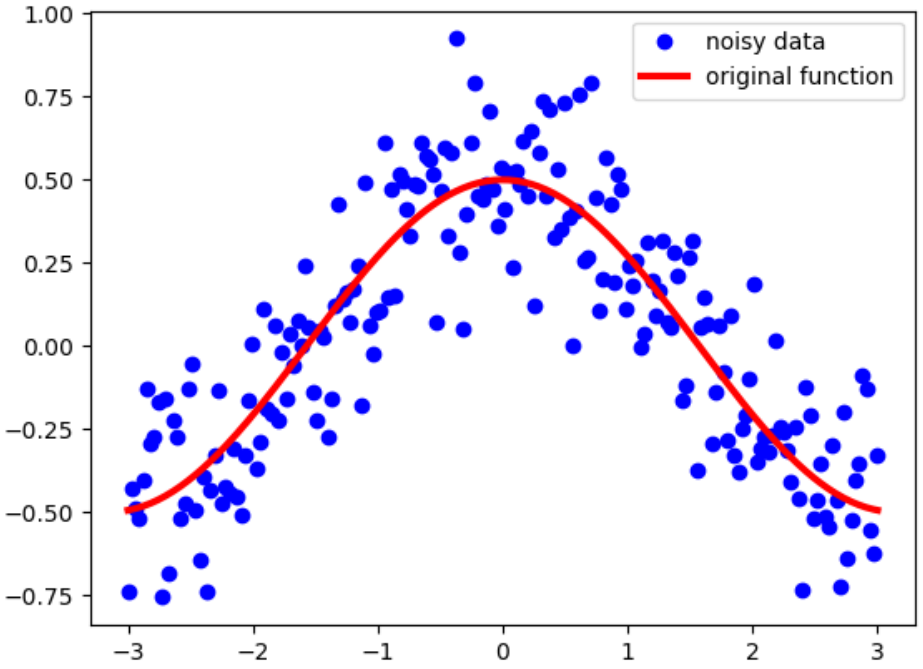
Second-order approximation, such as Newton's methods.

Momentum

Designed to accelerate learning, especially when loss is of high curvature.

We can understand momentum via exponentially weighted moving averages.

Suppose we have a sequence *S* which is noisy:



Instead of using only my mini batch I keep track also of the previous gradients, thus having a moving average. This technique helps the optimizer move faster through the parameter space and navigate challenging landscapes more effectively.

Momentum

Exponentially weighted averages:

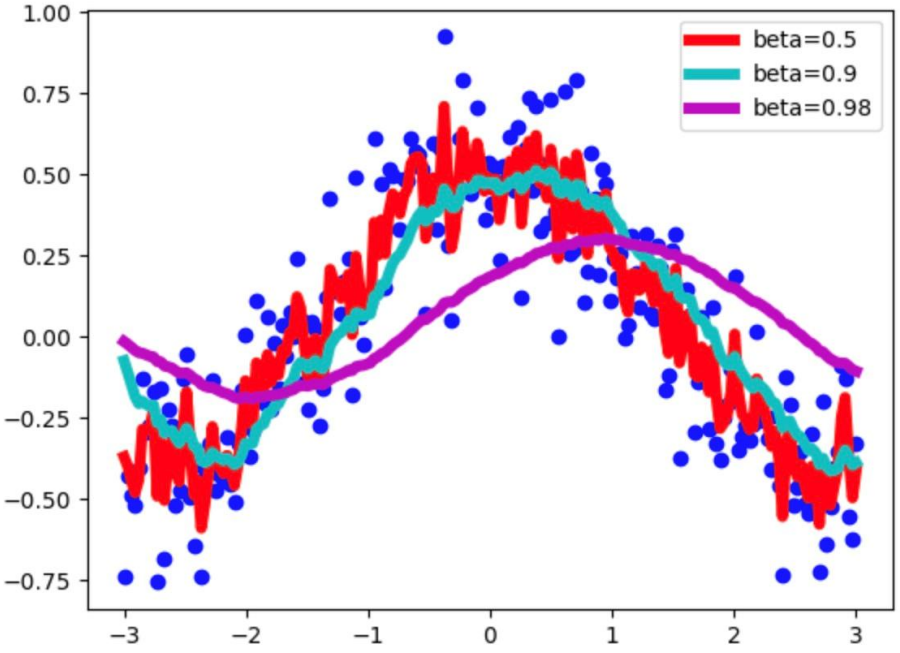
$$V_t = \beta V_{t-1} + (1-\beta) S_t, \ \beta \in [0\ 1], V_0 = 0$$

Small β leads to more fluctuations.

- β=0.9 provides a good balance

Bias correction.

- E.g., $V_1 = \beta V_0 + (1-\beta) S_1$: biased towards V_0
- $V_t = \frac{V_t}{1-\beta^t}$



Stochastic gradient descent with momentum

dampens oscillations means making less strong oscillations

Don’t switch update direction all the time.

Maintain “*momentum*” from previous updates → dampens oscillations.

$$v_{t+1} = \gamma v_t + \eta_t g_t, \quad \eta_t = \text{learning rate}$$

$$w_{t+1} = w_t - v_{t+1}$$

Exponential averaging keeps steady direction.

Example: $\gamma = 0.9$ and $v_0 = 0$

- $v_1 \propto -g_1$
- $v_2 \propto -0.9g_1 - g_2$
- $v_3 \propto -0.81g_1 - 0.9g_2 - g_3$

Standard gradient descent faces several limitations that momentum addresses. In regions with steep gradients in some directions and flat gradients in others, vanilla gradient descent oscillates heavily, slowing convergence. Additionally, it can get stuck in local minima or saddle points where gradients are small. On flat plateaus where gradients diminish, progress becomes extremely slow even when far from the optimum

The algorithm maintains a velocity term that represents the accumulated momentum of previous gradient updates. At each iteration, the velocity is updated as an exponentially weighted combination of the previous velocity and the current gradient

The momentum term produces several advantages. It accelerates convergence by allowing the optimizer to build up speed in consistent directions, much like a ball rolling down a hill gains momentum to carry it over small bumps and plateaus. It dampens oscillations by smoothing out the gradient updates across iterations, reducing the zig-zag pattern common in standard gradient descent. The technique also helps escape local minima and saddle points because accumulated momentum can carry the optimization through regions of small gradients.

Adding momentum is easy

SGD

$$w_{t+1} = w_t - \alpha \nabla f(w_t)$$

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

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(w_t)$$

$$w_{t+1} = w_t - \alpha v_{t+1}$$

```
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

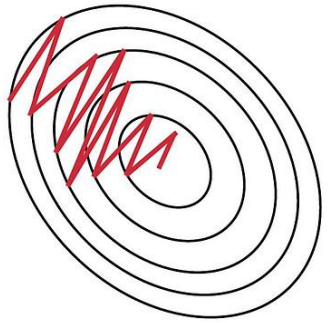
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Physical interpretation of momentum

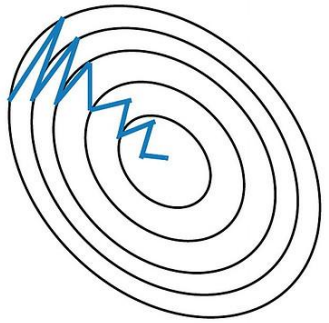
See gradient descent as rolling a ball down a hill.

The ball accumulates momentum, gaining speed down a straight path.

Momentum term increases for dimensions whose graidnets point in the same direction and reduces for dimesnions whose gradients change directions.

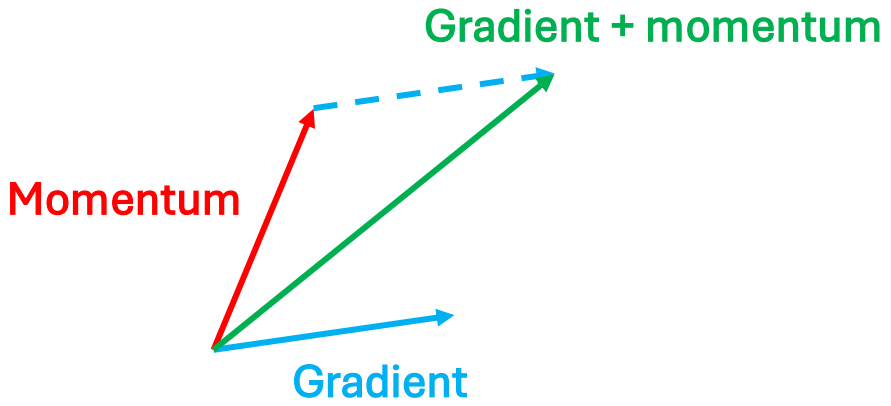


without momentum



with momentum

Nesterov momentum



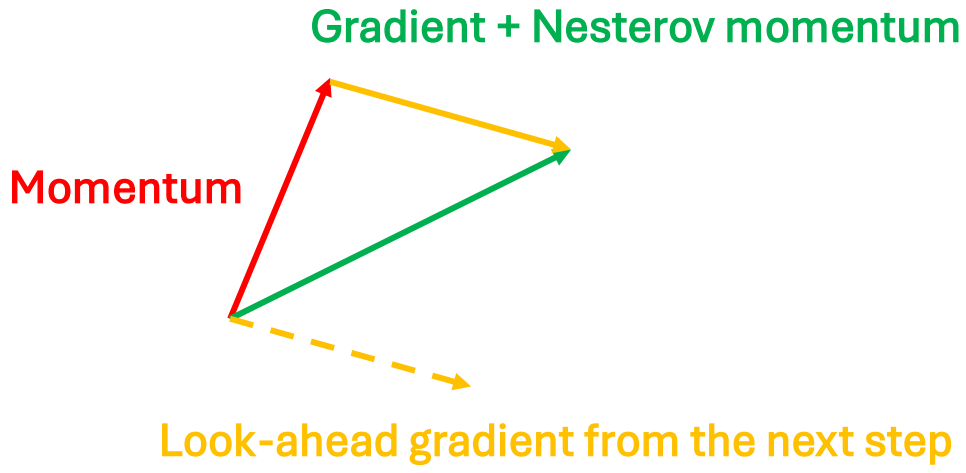
Use future gradient instead of current gradient:

$$v_{t+1} = \gamma v_t + \eta_t \nabla_w \mathcal{L}(w_t - \gamma v_t)$$

$$w_{t+1} = w_t - v_{t+1}$$

Prevents us from going too fast.

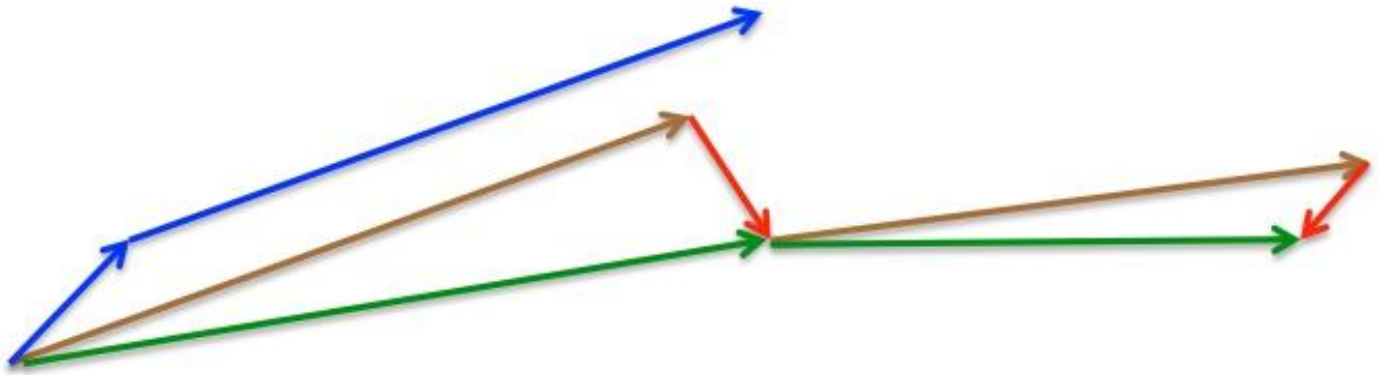
Also increases reponsiveness.



Nesterov momentum

First make a big jump in the direction of the previous accumulated gradient.

Then measure the gradient where you end up and make a correction.



brown vector = jump;
red vector = correction;
green vector = accumulated gradient;
blue vectors = standard momentum

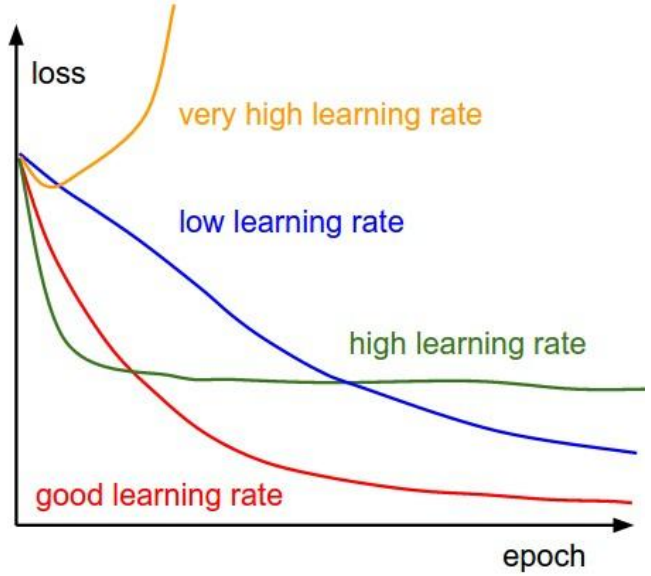
Picture from Hinton lecture

Adaptive step sizes

A fixed learning rate is difficult to set.

Also has significant impact on performance and sensitive.

Is it possible to have a separate adaptive learning rate for each parameter?



AdaGrad

Adaptive Gradient Algorithm – Adagrad:

- The learning rate is adapted **component-wise** to the parameters by incorporating knowledge of past observations.
- Rapid decrease in learning rates for parameters with large partial derivatives.
- Smaller decrease in learning rates for parameters with small partial derivatives.

Schedule

- $w_{t+1} = w_t - \frac{\eta}{\sqrt{r}+\varepsilon} \odot g_t,$
- where $r = \sum_t (\nabla_w \mathcal{L})^2$

learning rate

gradient

epsilon (avoid division by zero)

accumulation of squared gradients

I want to decrease my learning rate if I have large partial derivatives, and increase it otherwise

Instead of accumulating all squared gradients, RMSprop maintains an exponentially decaying moving average of the squared gradients.

RMSprop

Decay hyper-parameter (usually 0.9)

Schedule

- $r_t = \alpha r_{t-1} + (1-\alpha) g_t^2$
- $v_t = \frac{\eta}{\sqrt{r_t}+\varepsilon} \odot g_t$
- $w_{t+1} = w_t - v_t$

The denominator normalizes the gradient update by dividing by the root mean square of past squared gradients, with ϵ (typically 1e-8) preventing division by zero. This means parameters with large gradients receive smaller effective learning rates, while parameters with small gradients maintain relatively larger learning rates

Large gradients, e.g., too "noisy" loss surface

- Updates are tamed

tamed means to make less powerful

Small gradients, e.g., stuck in plateau of loss surface

- Updates become more aggressive

SGD with momentum and Adam are the most used

Adam

One of the most popular algorithms.

Combines RMSprop and momentum.

- Computes adaptive learning rate for each parameter.
- Keeps an exponentially decaying average of past gradients (momentum).
- Introduces bias corrections to the estimates of moments.

Can be seen as a heavy ball with friction, hence a preference for flat minima.

Adam

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$ momentum of my gradients

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$ RMSprop

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Bias corrections

this boost the values at the start temporarily, 1-beta at the umber of steps I took

$$u_t = \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t$$

$$w_{t+1} = w_t - u_t$$

At start t = 1, After all epocvhs t is very high so the value of b^t converges to 0, and the denominator converges to 1

Recommended values: $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$

Adaptive learning rate as RMSprop, but with momentum & bias correction

Adam is hyperparameter-free?

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

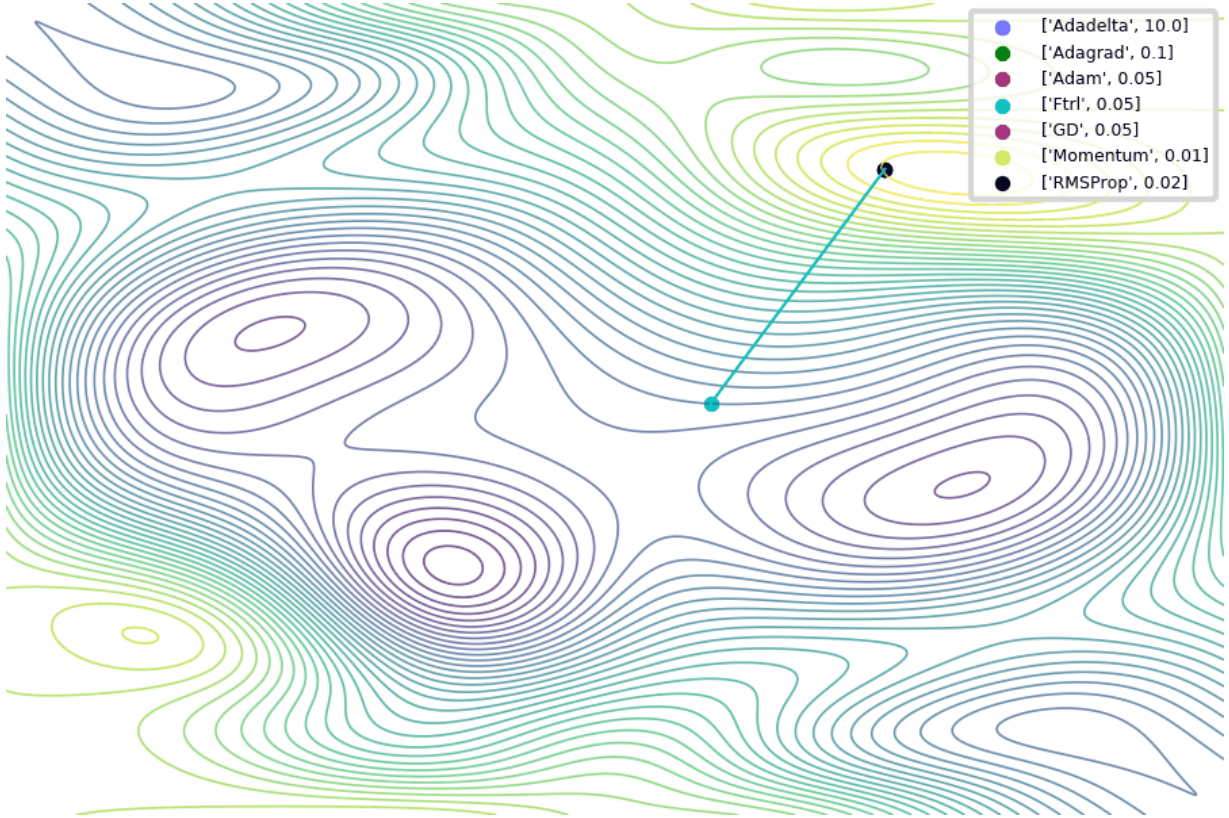
$$u_t = \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t$$
$$w_{t+1} = w_t - u_t$$

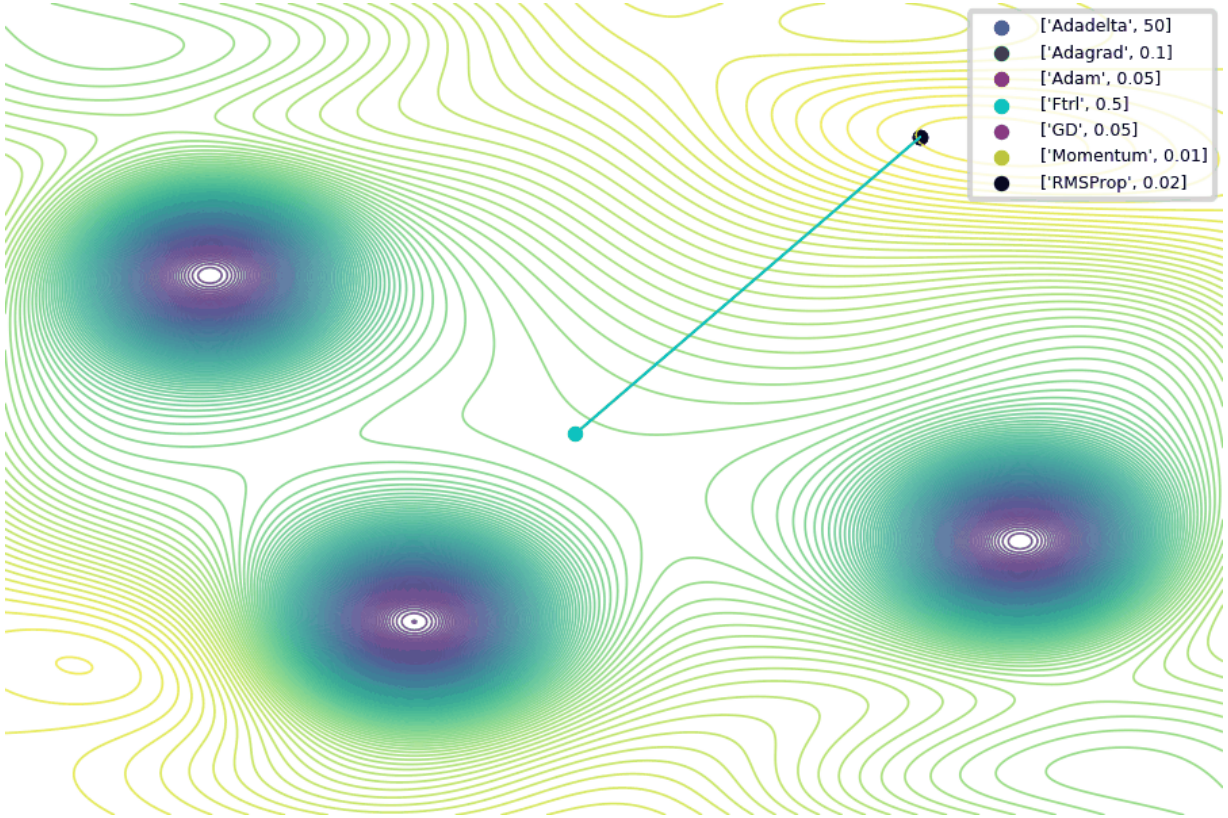
it is NOT hyperpar free, instead it has more hyperpar respect to SGD (3 more). It allows to control over the learning rate across different network parametrs. Sometimes I will update more, other I will update less

It has 4 hyperpar, eta, epsilon, beta1, beta2

More robust to different settings, but many values to set.

Visual overview





Picture credit: Jaewan Yun

Which optimizer to use?

My go-to: SGD with momentum and learning rate decay.

For more complex models, Adam is often the preferred choice.

Adam with weight-decay (AdamW) is the standard for optimizing transformer architectures.

Oddity: even in “learning rate adjusting” optimizers like Adam, we add learning rate decays.

Interactive visualization

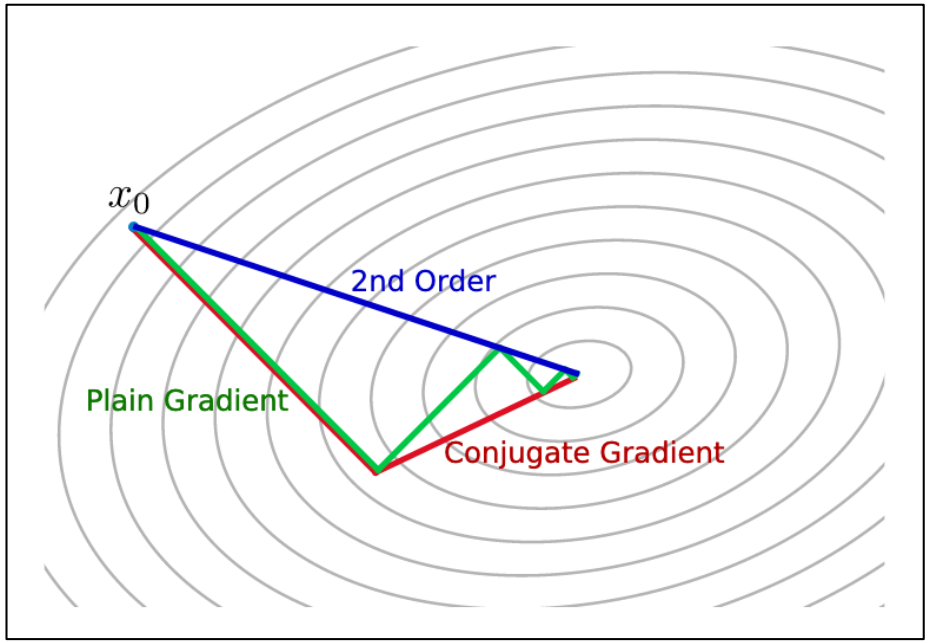
https://emiliendupont.github.io/2018/01/24/optimization-visualization/

Approximate second-order methods

SGD, Adam, etc are first-order: curvature information is ignored.

Benefits of second-order optimization:

- Better direction.
- Better step-size.
- Full step jumps directly to the minimum
- of the local squared approx.
- Additional step size reducation and
- Dampening becomes easy.



Newton's method

A second-order Taylor series expansion to approximate $J(\theta)$ near some point θ_0, ignoring derivatives of higher order:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

If we then solve for *the critical point* of this function, we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Computing the Hessian is expensive, and also inverting it is worse

For a locally quadratic function, Newton's method jumps directly to the minimum.

If convex but not quadratic (there are higher-order terms), *update can be iterated*.

Why we use first order optimization

Disadvantages:

- Super slow: need to compute inverse of Hessian matrix each time.
- More restrictive: 2nd order derivative needs to be possible to compute.
- Limited impact: no major improvement found in practice.

Learning and reflection

Understanding Deep Learning: Chapter 6

Understanding Deep Learning: Chapter 7

Next lecture

Lecture	Title
1	Intro and history of deep learning
3	Deep learning optimization I
5	Convolutional deep learning
7	Graph deep learning
9	Multi-modal deep learning
11	What doesn't work in deep learning
13	Q&A

Lecture	Title
2	AutoDiff
4	Deep learning optimization II
6	Attention-based deep learning
8	From supervised to unsupervised deep learning
10	Generative deep learning
12	Non-Euclidean deep learning
14	Deep learning for videos

Thank you!