



Deep Learning 1

2024-2025 – 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 Neural Networks I
7	Attention
9	Self-supervised and vision-language learning
11	The oddities of deep learning
13	Deep learning for videos

Lecture	Title
2	Manually forward, automatically backward
4	Deep learning optimization II
6	Convolutional Neural Networks II
8	Graph Neural Networks
10	Auto-encoding and generation
12	Non-Euclidean deep learning
14	Q&A

This lecture

Stochastic gradient descent

Advanced optimizers

Network initialization

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

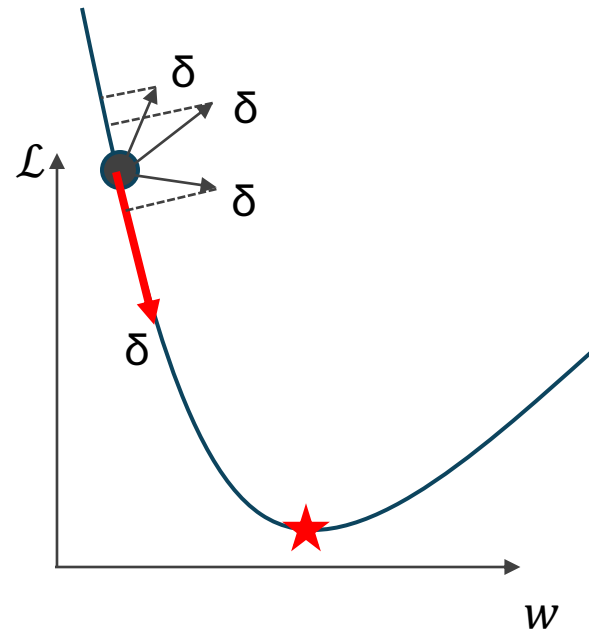
$$\min_{\mathbf{w}} \mathbb{E}_{\mathbf{x}, y \sim p_{data}} [\mathcal{L}(f(\mathbf{x}, \mathbf{w}), y)] + \lambda \Omega(\mathbf{w})$$

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

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

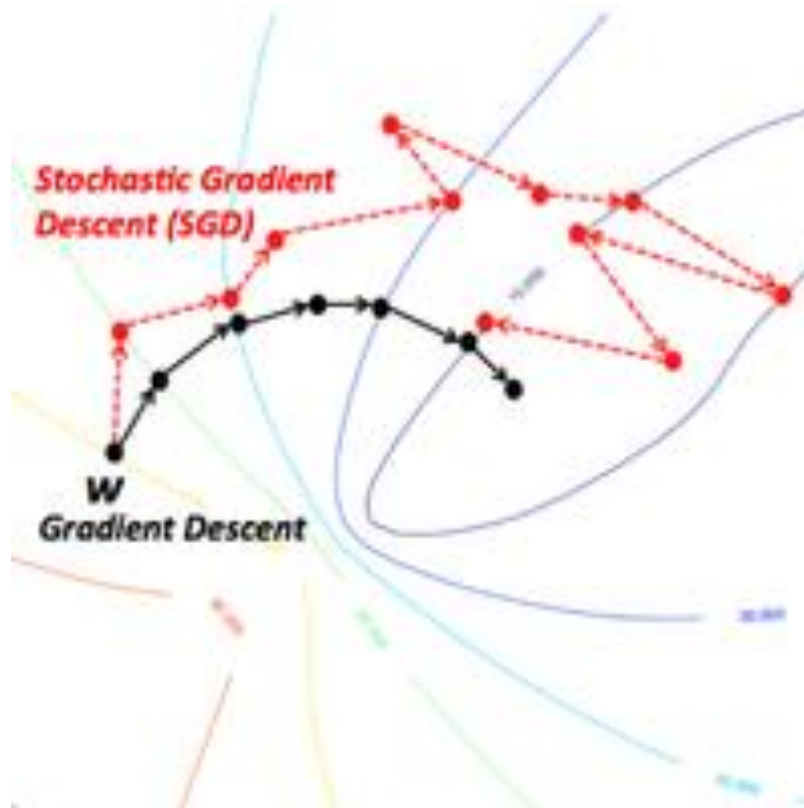
with \hat{p}_{data} the empirical data distribution, defined by a set of training examples.

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

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

Gradient descent based on optimization.

Stochastic gradient descent



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

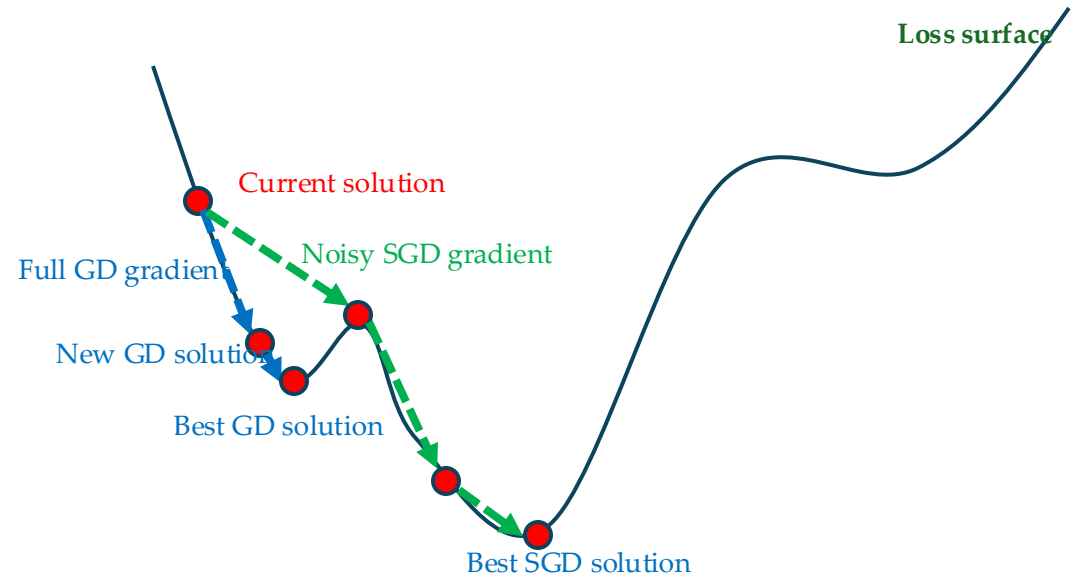
Properties of SGD

Randomness actually reduces overfitting.

Reshuffling is important!

One epoch = go through all mini-batches.

Be careful to balance class/data per batch.

[illegible]

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.

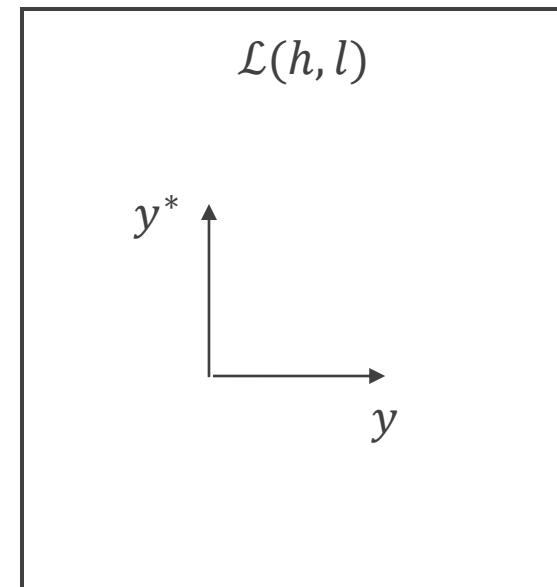
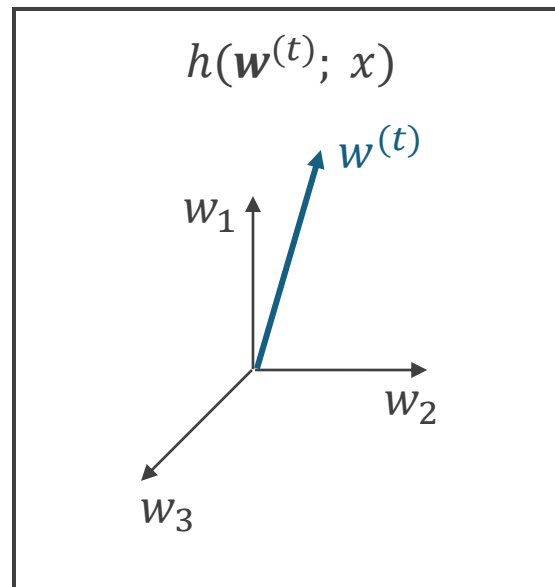
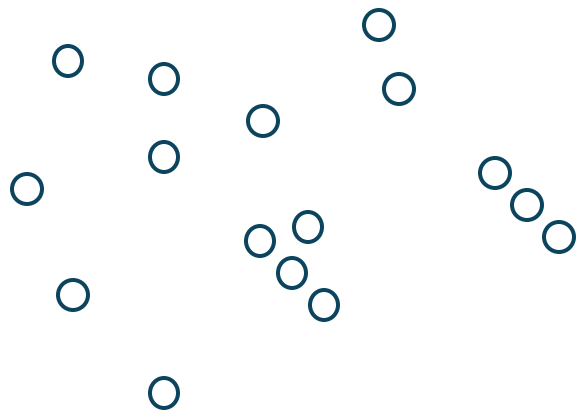
The training samples may have lots of noises or outliers or biases.

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

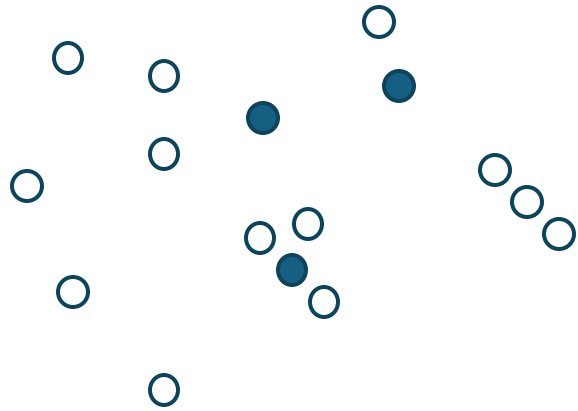
Real gradient might get stuck into a local minima.

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

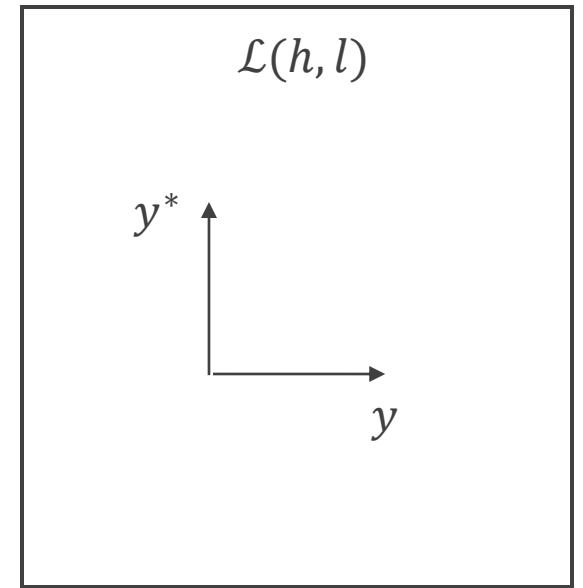
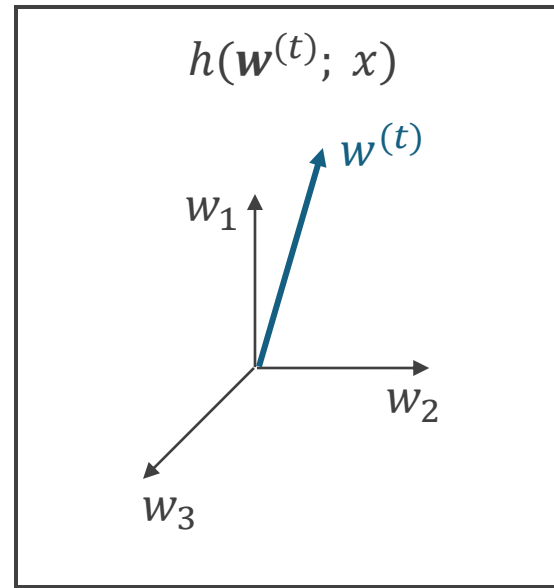
Stochastic gradient descent



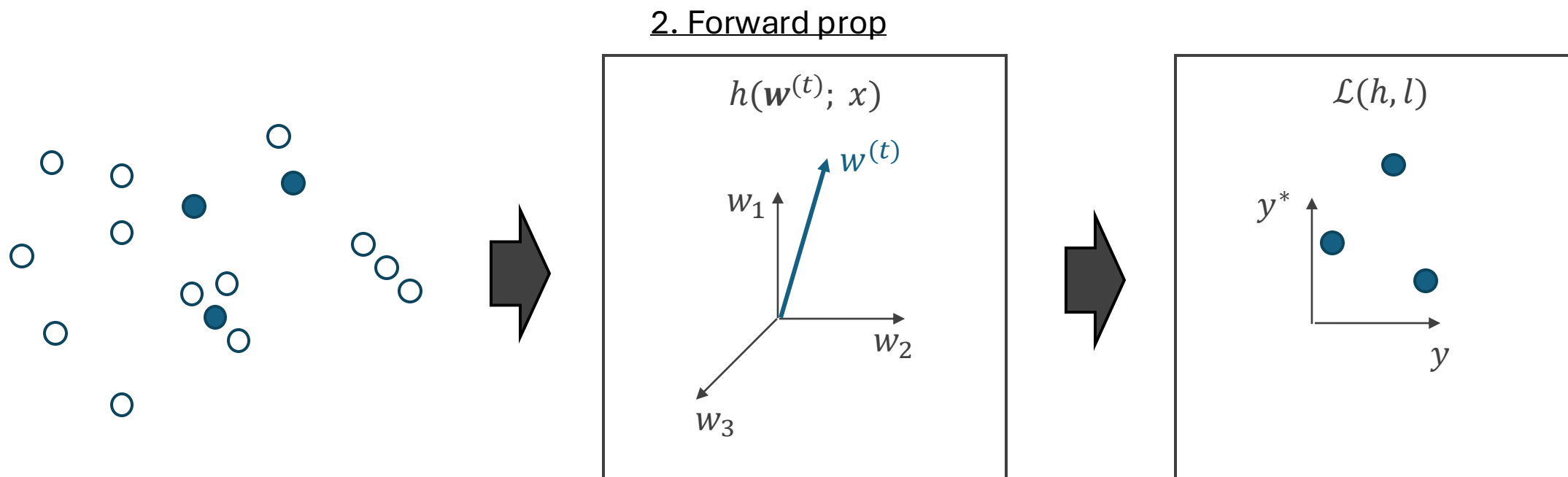
Stochastic gradient descent



1. Sample mini-batch

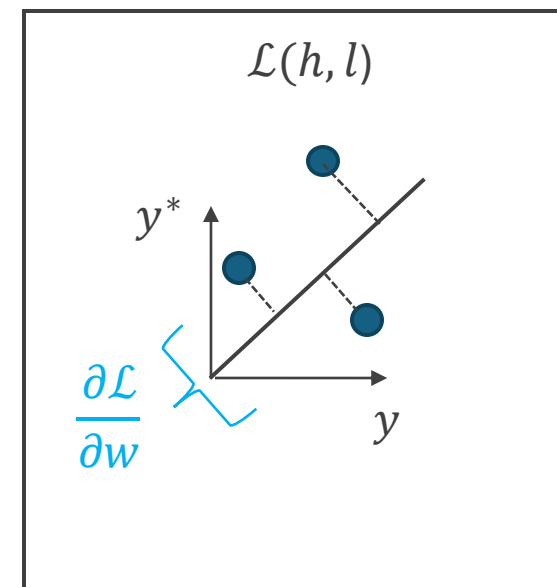
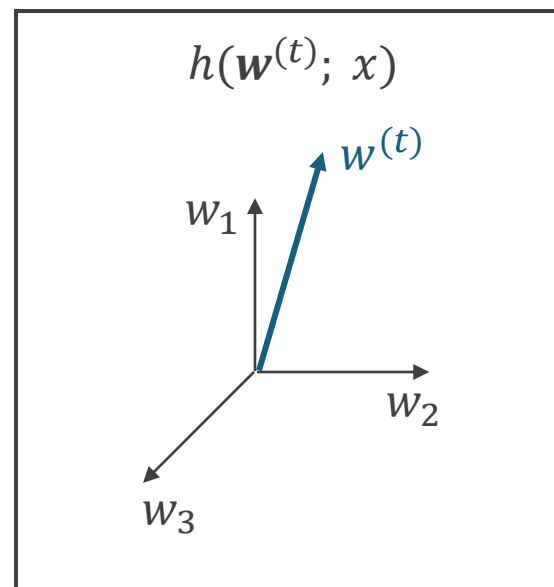
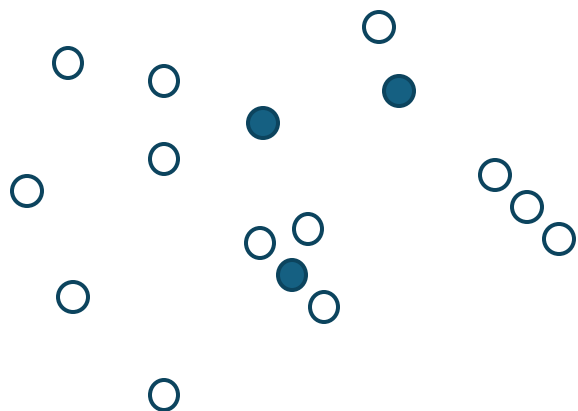


Stochastic gradient descent



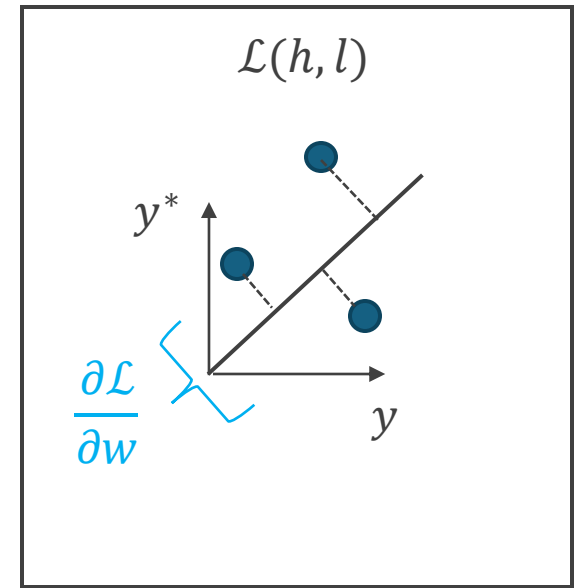
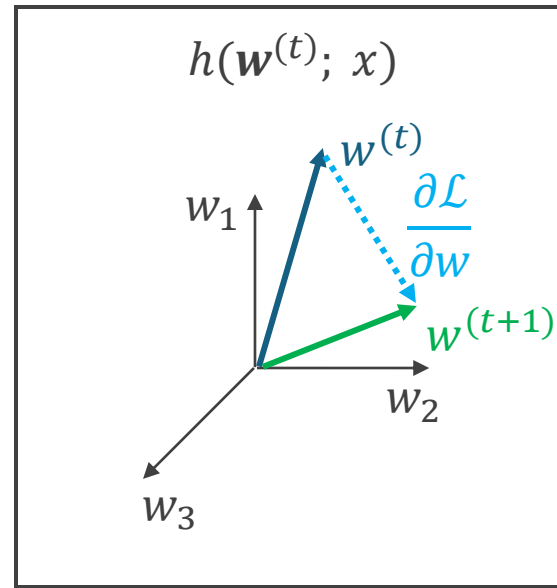
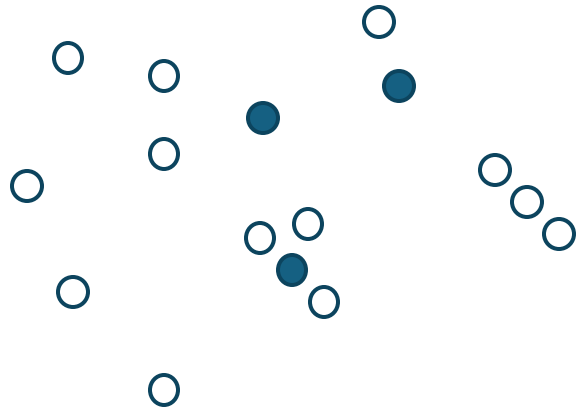
Stochastic gradient descent

3. Compute errors and gradients



Stochastic gradient descent

4. Update model parameters and repeat



In a nutshell

First, define your neural network

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

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

Finding an “optimal” neural network means minimizing a loss function

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

Rely on stochastic gradient descent methods to obtain desired parameters

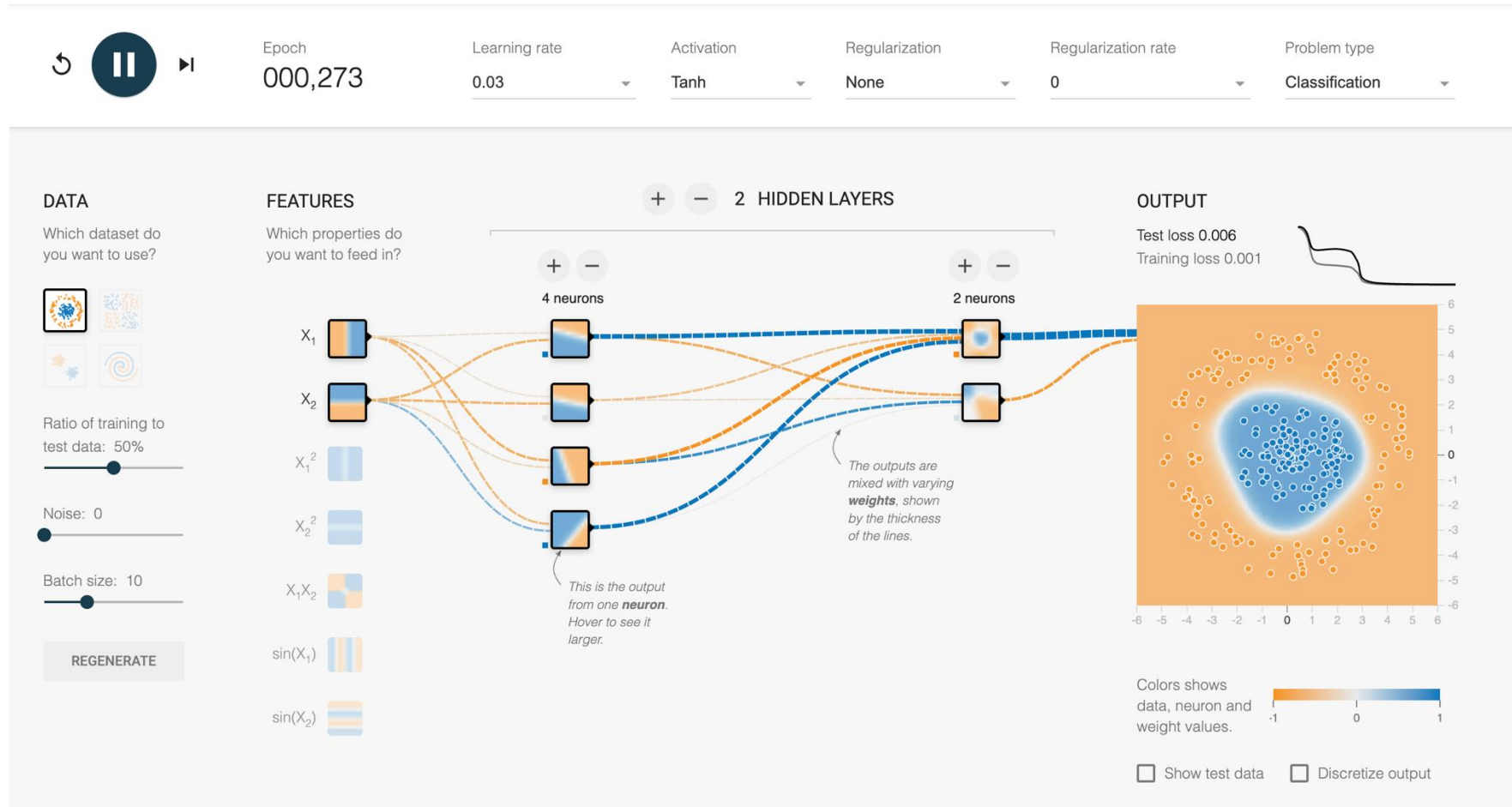
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{dL}{d\mathbf{w}}$$

where η is the step size or learning rate.

Gradient vs Stochastic Gradient Descent

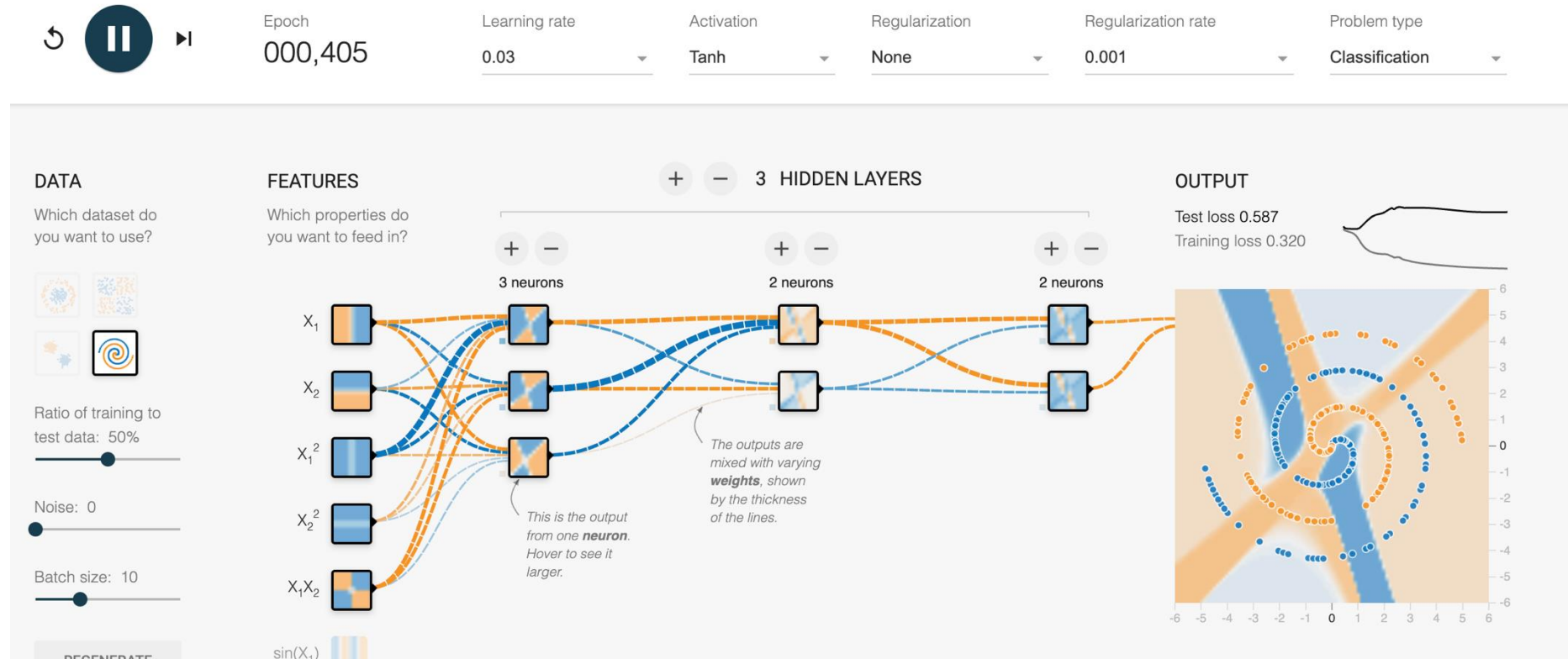
Gradient Descent	Stochastic Gradient Descent
Computes gradient using the whole Training dataset	Computes gradient using a single training sample
Slow and computationally expensive algorithm	Faster and less computationally expensive than GD
Not suggested for huge training samples	Can be used for large training samples
Deterministic in nature	Stochastic in nature
Gives optimal solution given sufficient time to converge*	Gives good solution but not optimal
No random shuffling of points are required	Shuffling needed. More hyperparameters like batchsize.
Can't escape shallow local minima easily	SGD can escape shallow local minima more easily
Convergence is slow	Reaches the convergence much faster

Practical examples



<https://playground.tensorflow.org/>

Practical examples



<https://playground.tensorflow.org/>

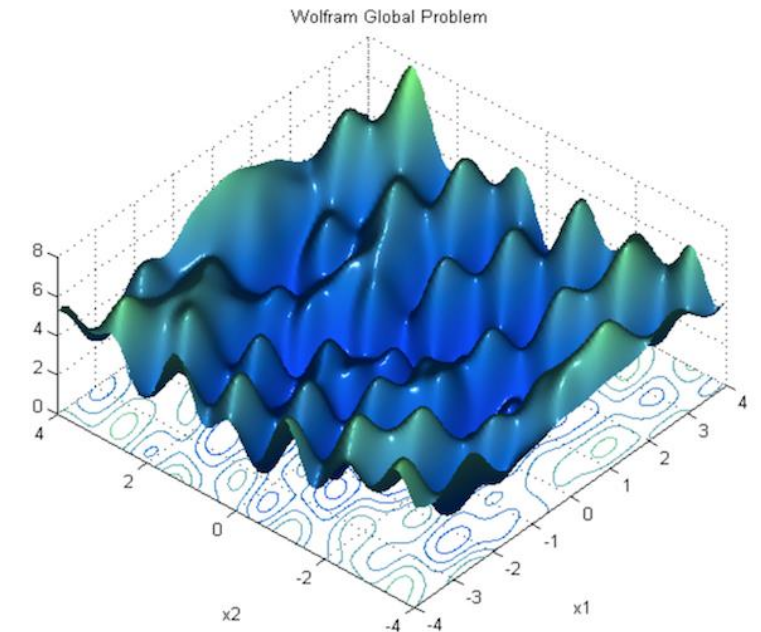
Challenges of optimizing deep networks

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?
- ...



Main challenges in optimization

1. Ill conditioning → a strong gradient might not even be good enough
2. Local optimization is susceptible 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.

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

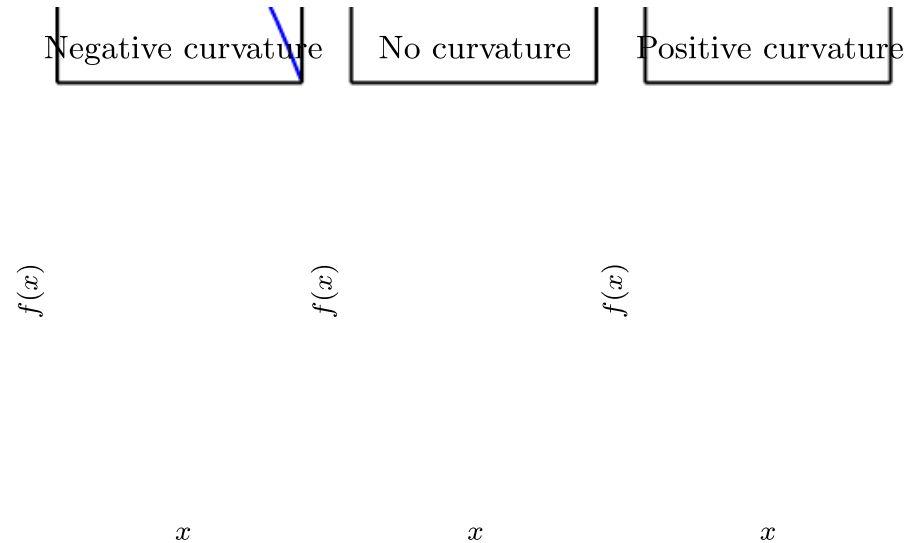
1. Ill conditioning

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.

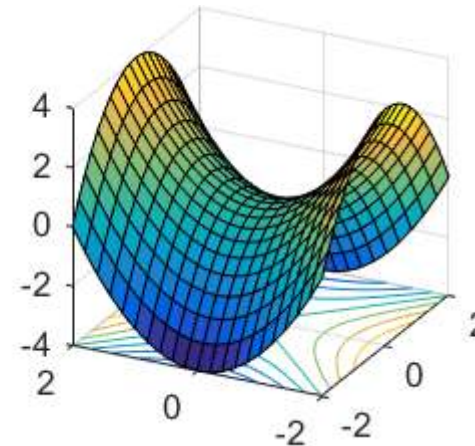
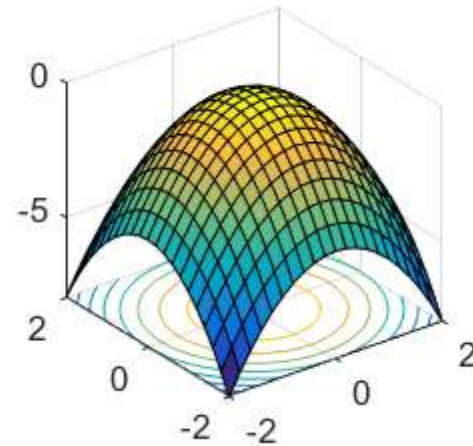
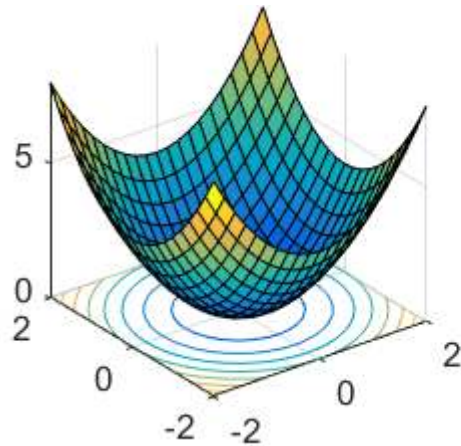
Positive curvature: the function decreases slower than expected and eventually begins to increase.



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?



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 how 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

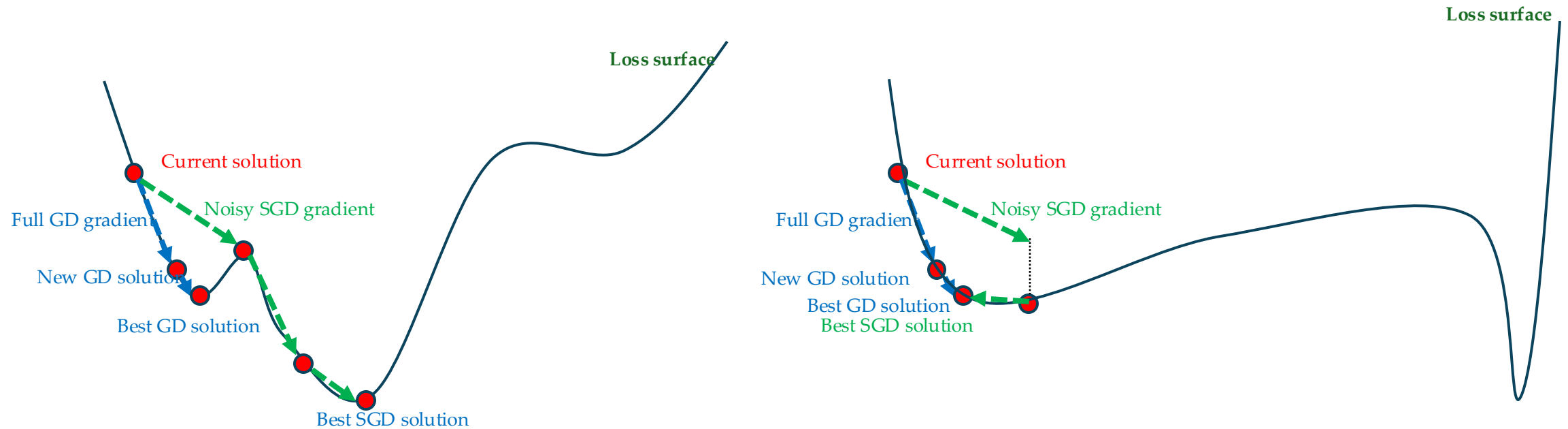
Model identifiability

- 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

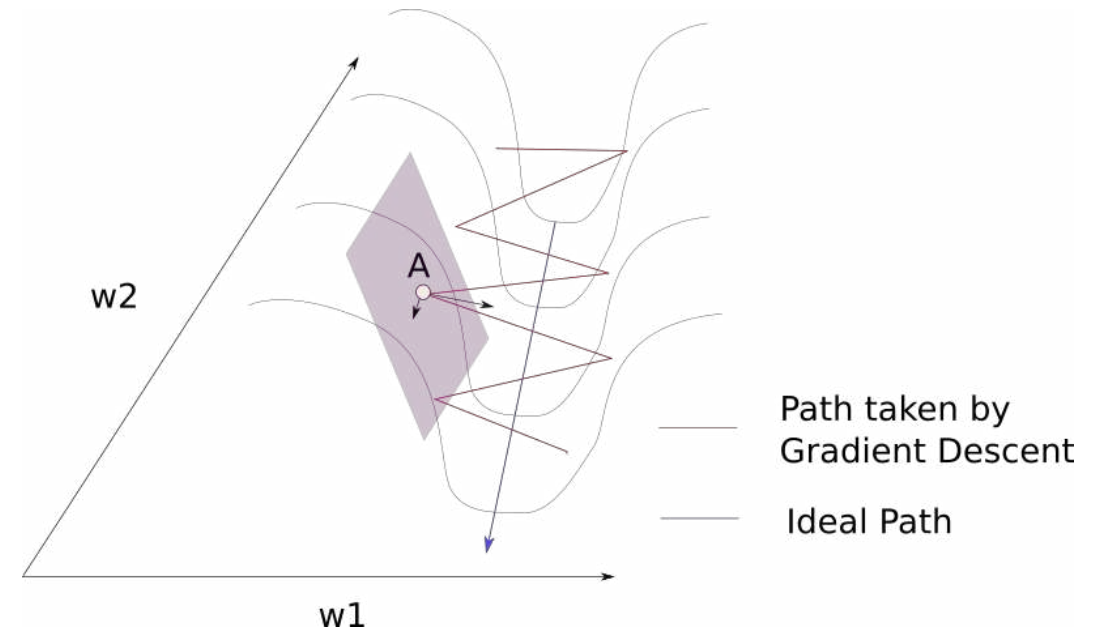
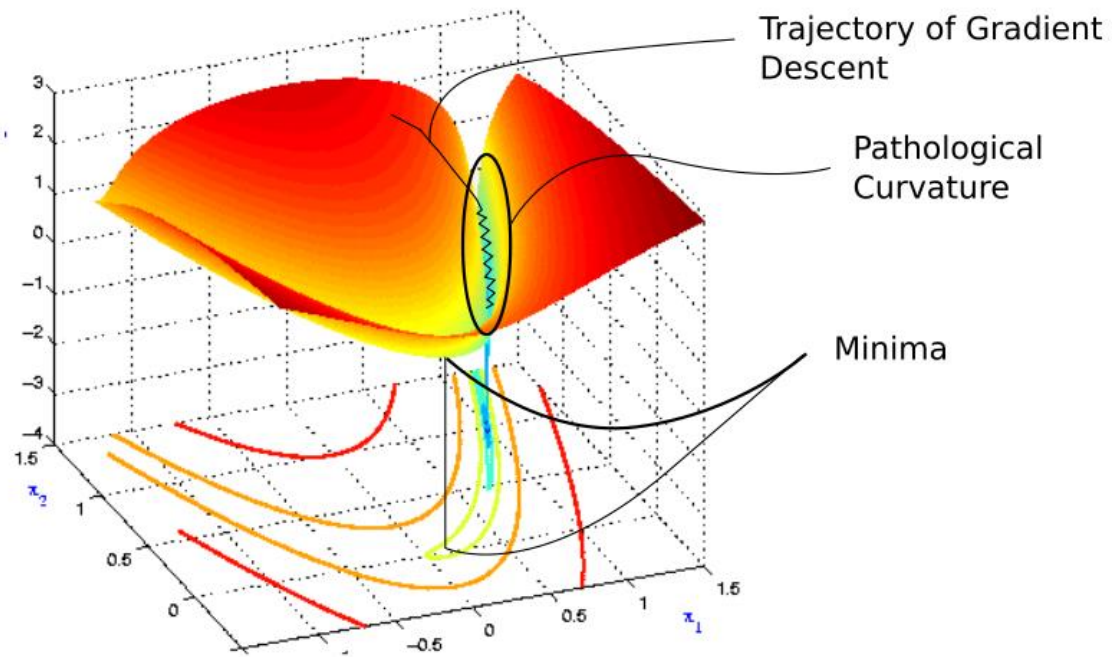
- However, all local minima from non-identifiability are equivalent in cost function value.
- *Those* local minima are not a problematic form of non-convexity.
- The other local minima (next slides) are.

Local minima



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

3. Ravines



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

3. Plateaus and flat areas

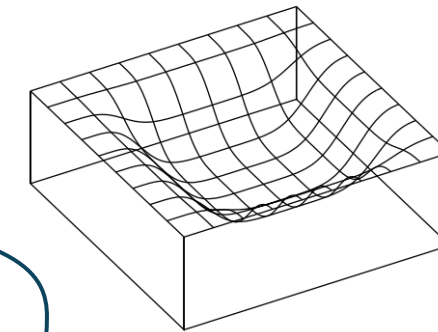
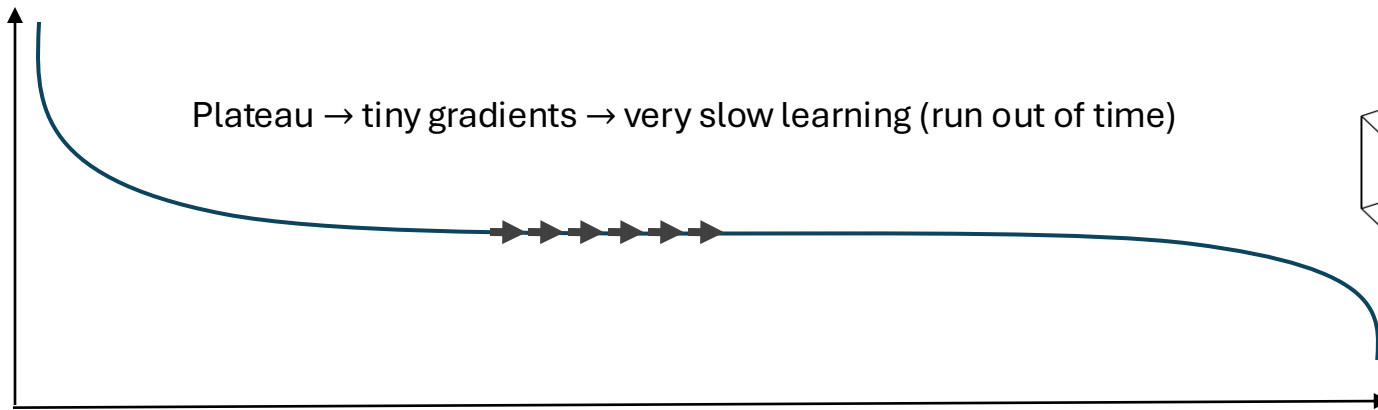


Figure 1: *Example of a “flat” minimum.*

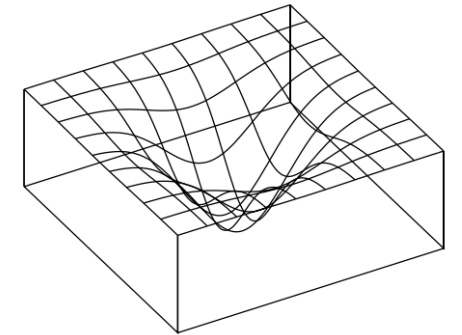


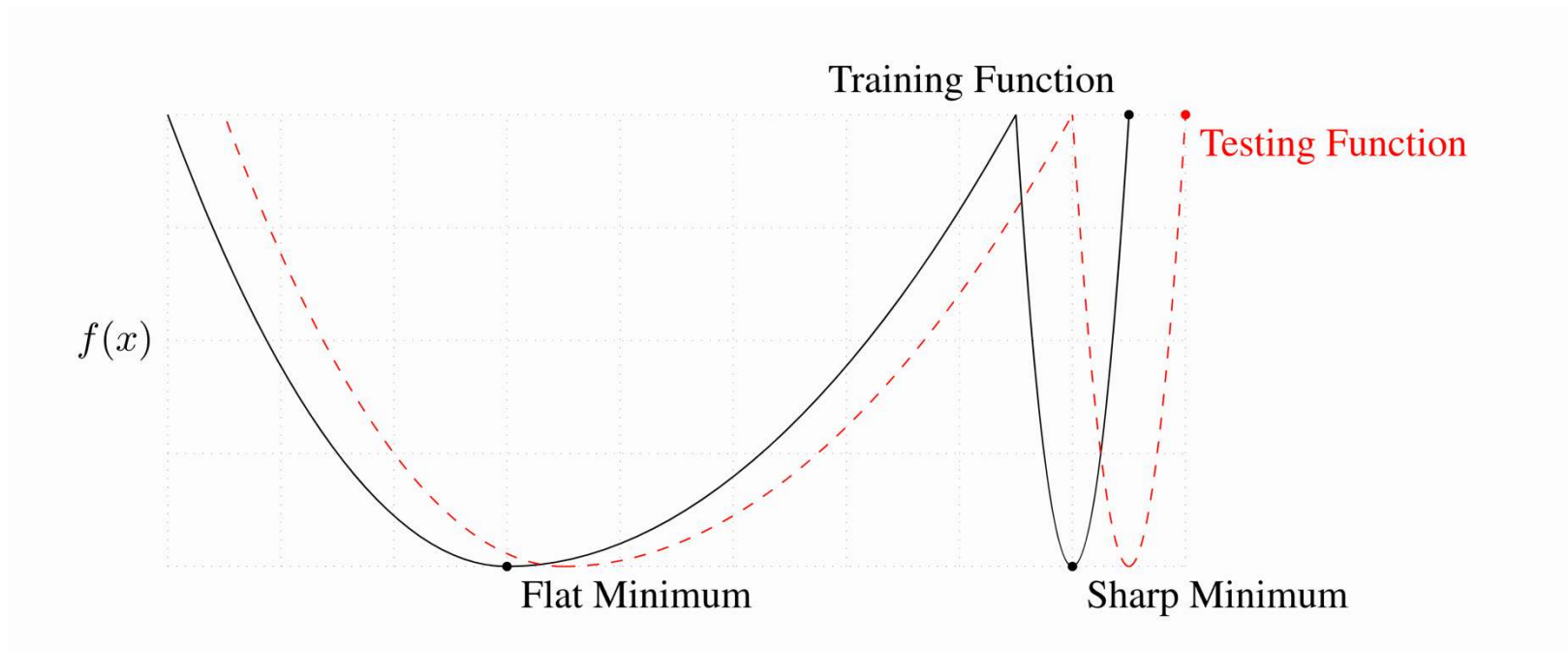
Figure 2: *Example of a “sharp” minimum.*

[Link](#)

Near zero gradients in flat areas, hence no learning.

Why are flat minima still preferred?

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



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

4. Cliffs and exploding gradients

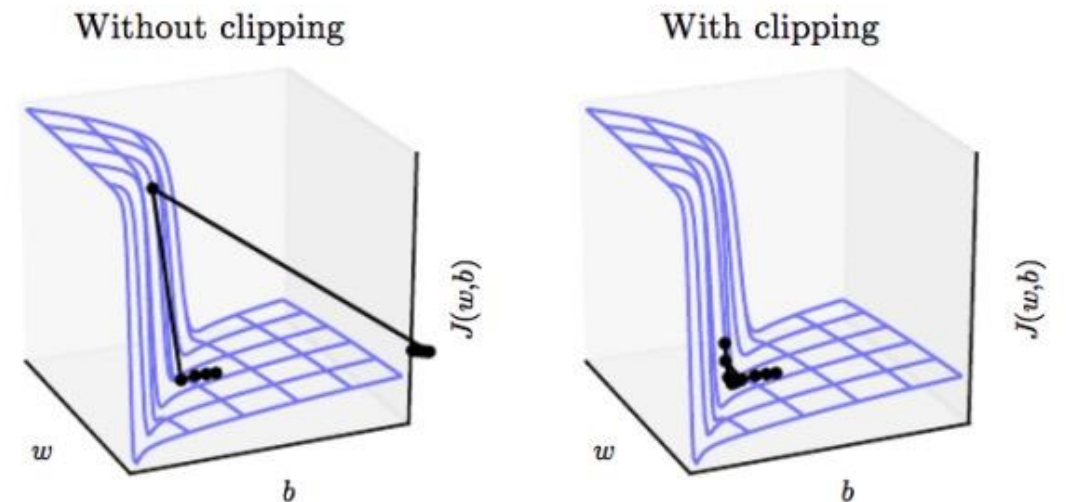
Neural networks with many layers often have steep regions resembling cliffs.

These result from the multiplication of several large weights together.

A simple trick: gradient clipping:

if $|g| > \eta$:

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

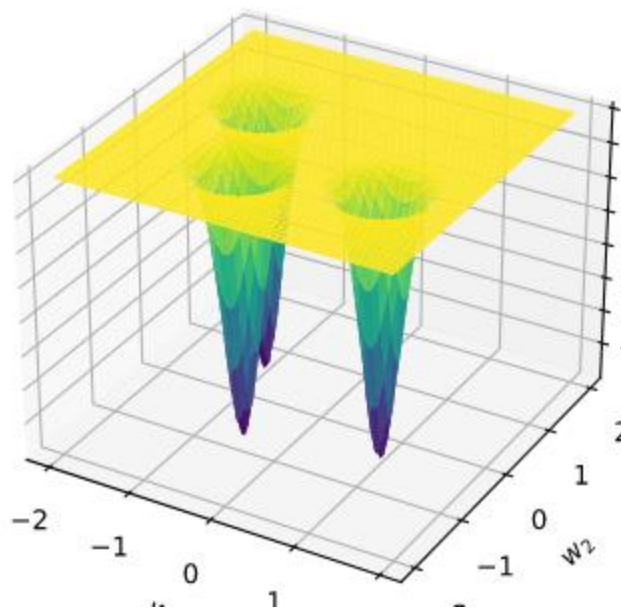


Flat areas + steep minima = annoying

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: $p(y|x) = \text{softmax}(\text{logits}/0.00001)$



5. Long-term dependencies

Especially for networks with many layers or recurrent neural networks.

The vanishing and exploding gradient problem

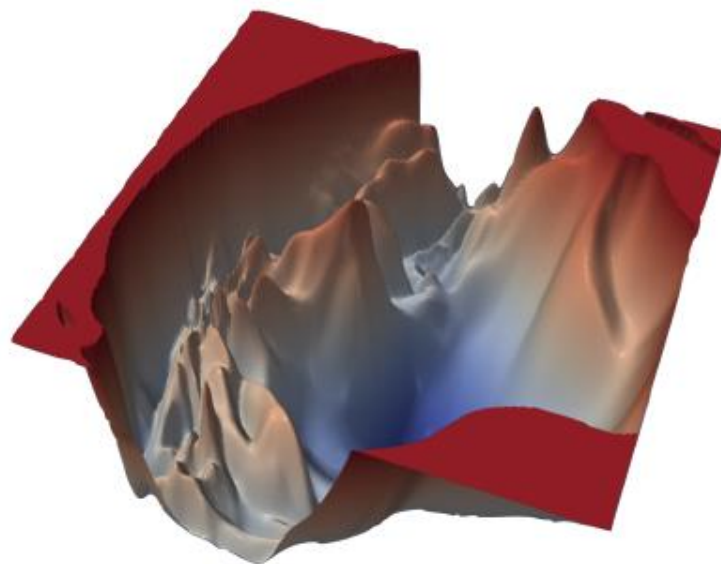
- Certain functions lead to a scaling of the gradient (potentially often).
- Vanishing gradients -> no direction to move
- Exploding gradients -> learning unstable.

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

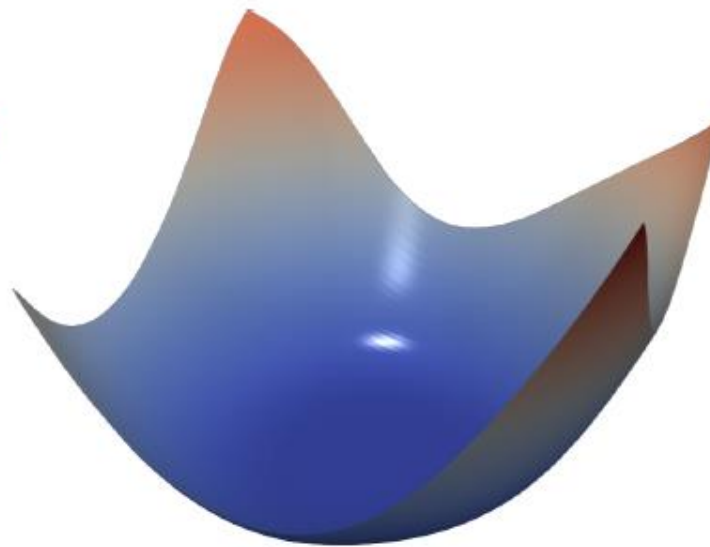
Is gradient descent even a good idea?

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

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



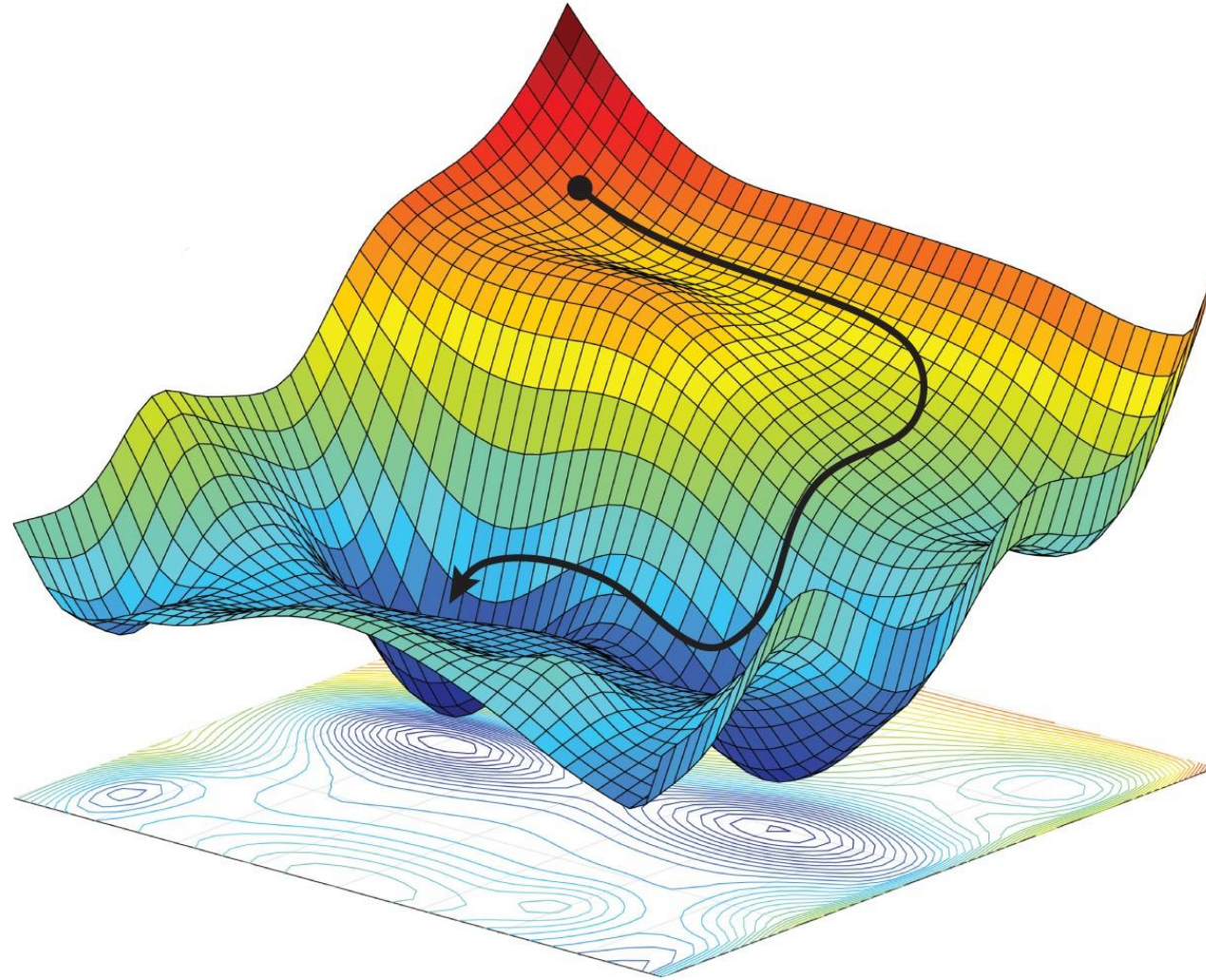
(a) ResNet-110, no skip connections




(b) DenseNet, 121 layers

Break

Advanced optimizers



Gradient descent itself can be enhanced

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{d\mathcal{L}(\mathbf{w})}{d\mathbf{w}}$$
The diagram consists of two blue arrows. One arrow originates from the text 'Can we improve the learning rate setting?' and points to the orange learning rate symbol η in the equation. The other arrow originates from the text 'Can we get a better or more useful gradient?' and points to the blue gradient term $\frac{d\mathcal{L}(\mathbf{w})}{d\mathbf{w}}$ in the equation.

Can we improve the
learning rate setting?

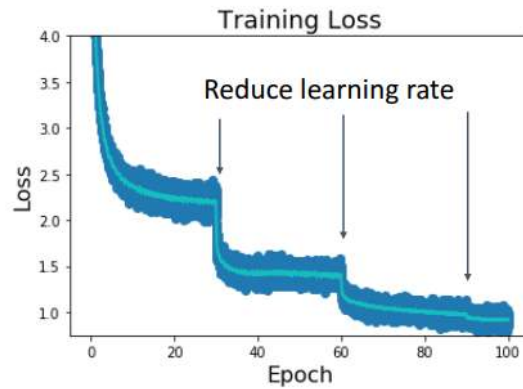
Can we get a better or
more useful gradient?

Setting the learning rate

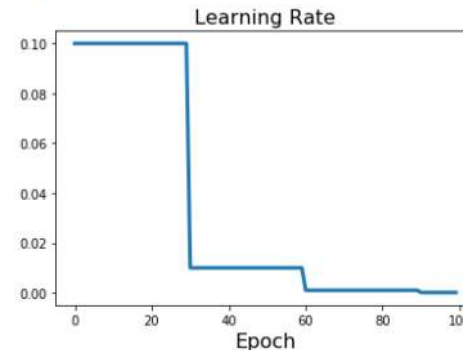
Truly an empirical endeavour, unique to each problem and dataset.

Big trick: learning rate schedulers.

Learning Rate Decay: Step



Step: Reduce learning rate at a few fixed points.
E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.



Improving gradient descent

Stochastic Gradient Descent with momentum.

Nesterov momentum.

Stochastic Gradient Descent with adaptive learning rates.

E.g., AdaGrad, RMSProp, Adam

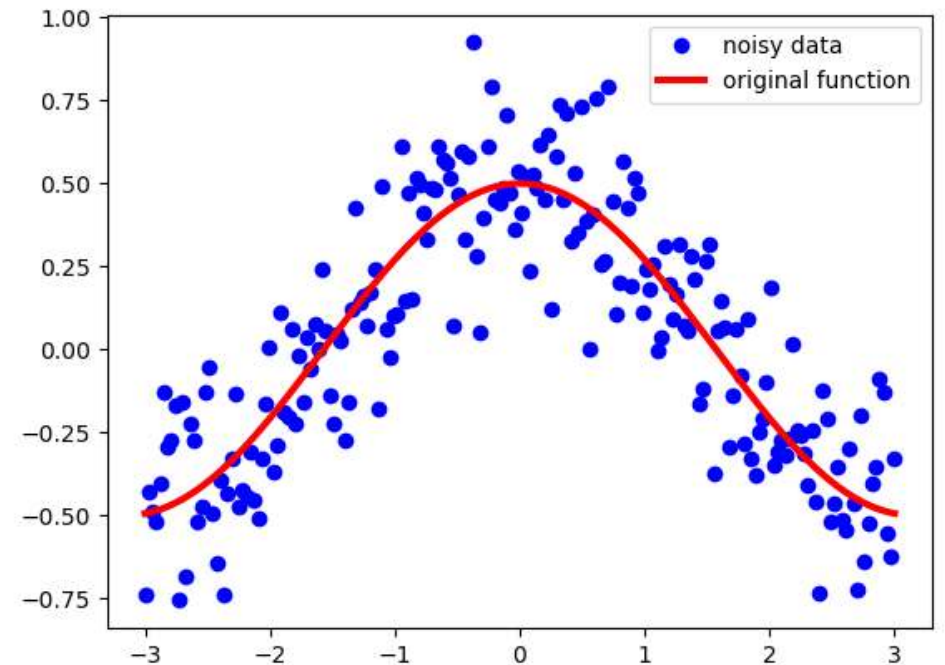
Second-order approximation, such as Newton's methods.

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:



Momentum

Exponentially weighted averages:

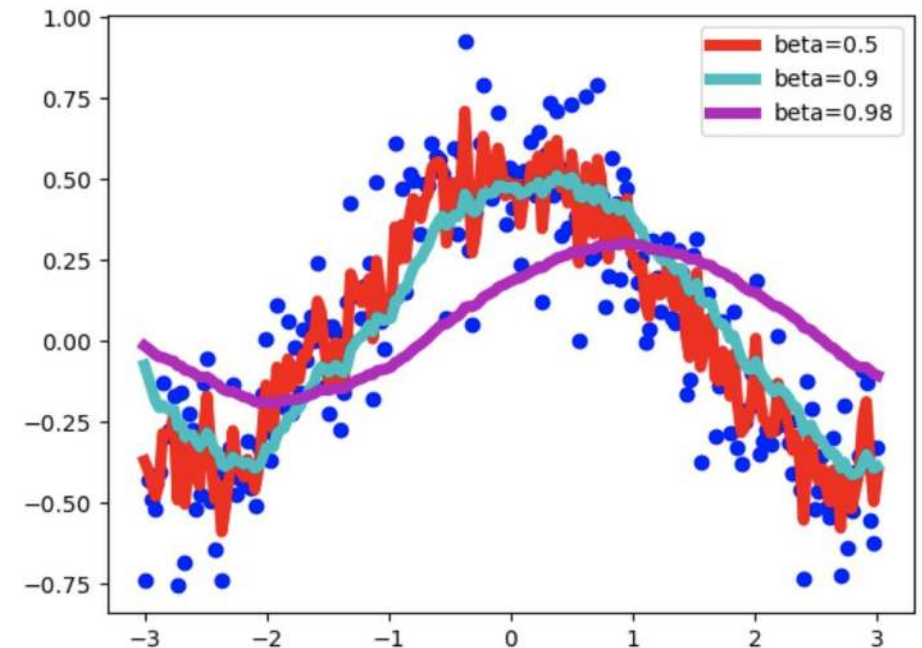
$$V_t = \beta V_{t-1} + (1 - \beta) S_t, \quad \beta \in [0, 1], \quad V_0 = 0$$

Small β leads to more fluctuations.

- $\beta=0.9$ 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

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 $v_0 = 0$

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

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

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

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

Correction from
previous slides:
It should be t+1 not t-1

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

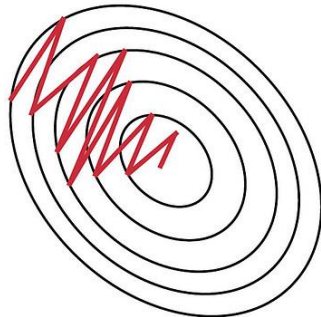
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Physical interpretation of momentum

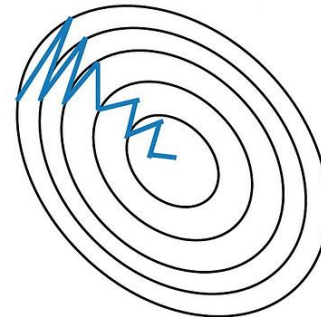
See gradient descent as rolling a ball down a hill.

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

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



without momentum



with momentum

Nesterov momentum

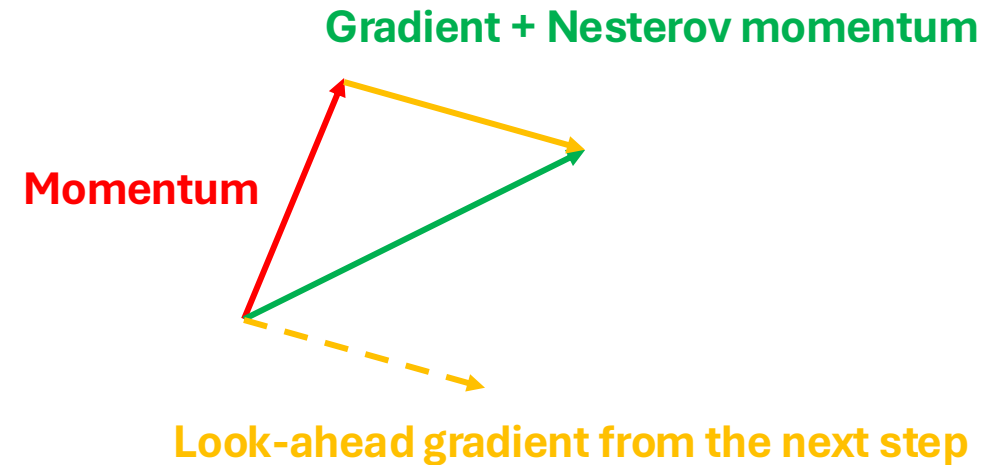
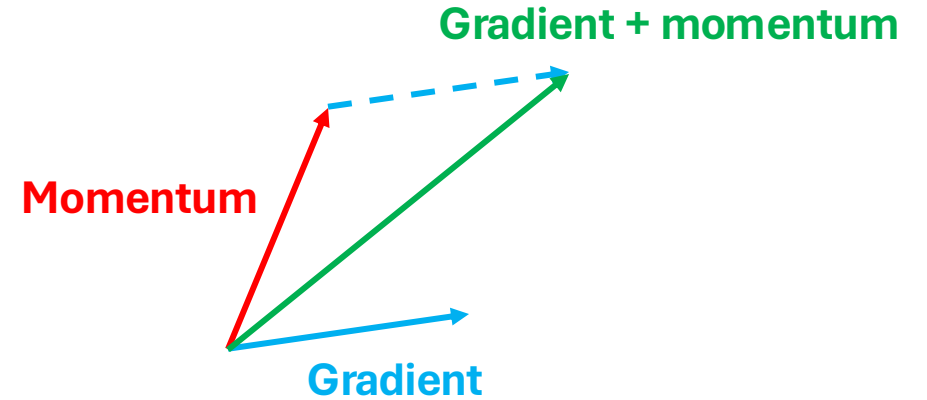
Use future gradient instead of current gradient:

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

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

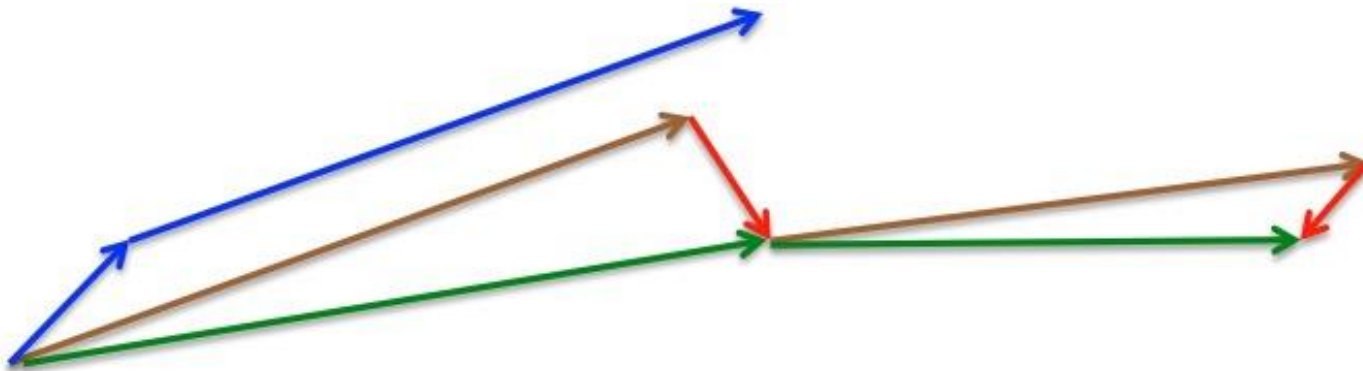
Prevents us from going too fast.

Also increases responsiveness.



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.



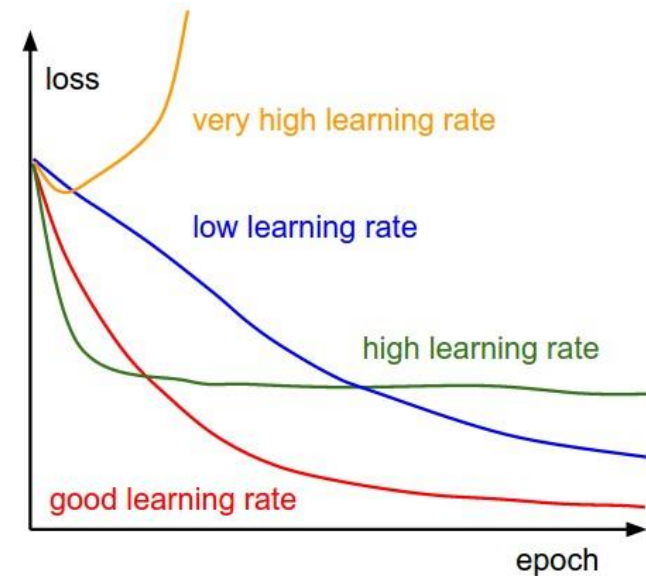
brown vector = jump;
red vector = correction;
green vector = accumulated gradient;
blue vectors = standard momentum

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

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

Clarification from lecture: the sum is over the timesteps, hence \mathbf{r} remains a vector.

AdaGrad and AdaDelta

Advantages:

- It “eliminates” the need to manually tune the learning rate.
- Faster and more reliable when the scaling of the weights is unequal.

Adadelta: Adagrad++

- Seeks to reduce its aggressive, monotonically decreasing learning rate.
- Restricts the window of accumulated past gradients to some fixed size, instead of accumulating all past squared gradients.
- No need to set a default learning rate, as it has been eliminated from the update rule.

RMSprop

Decay hyper-parameter (usually 0.9)

Schedule

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

Large gradients, e.g., too “noisy” loss surface

- Updates are tamed

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

- Updates become more aggressive

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

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

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

Bias corrections

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

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

Adaptive learning rate as **RMSPprop**, but with **momentum** & **bias correction**

Adam is hyperparameter-free?

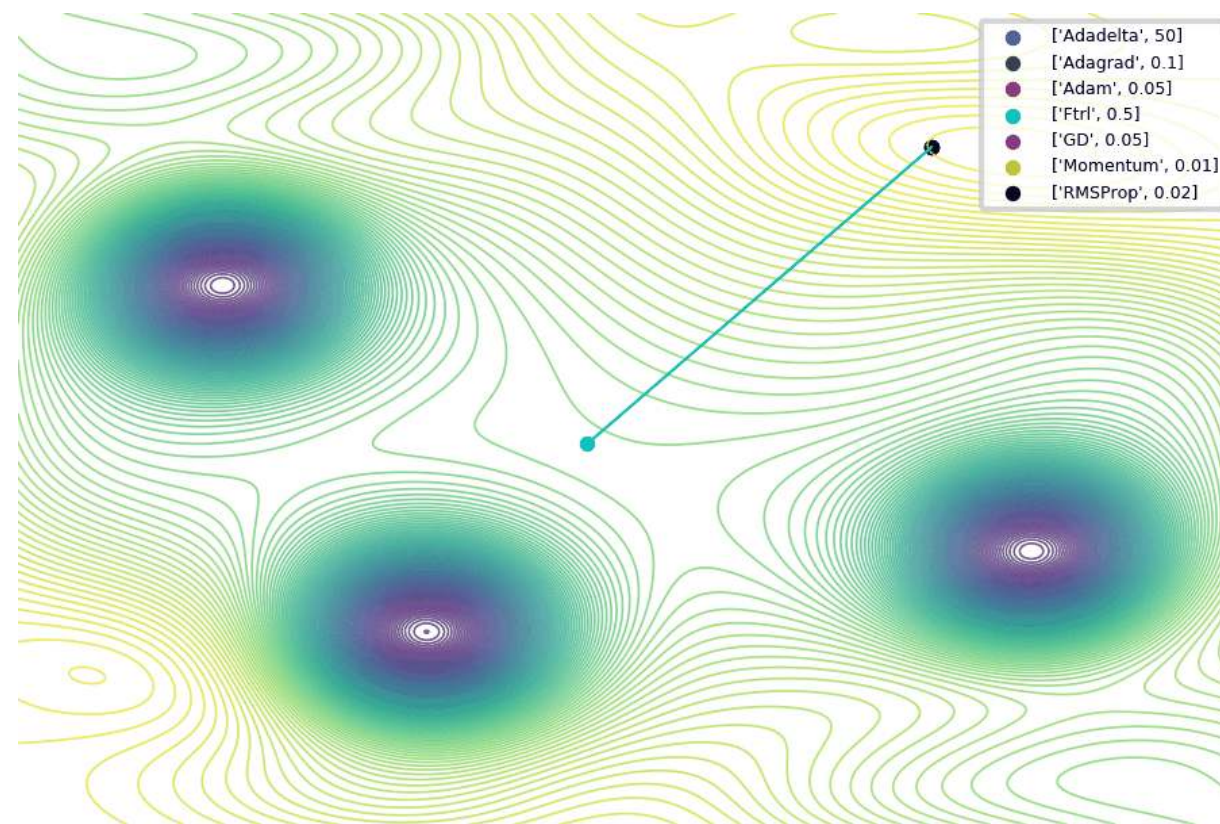
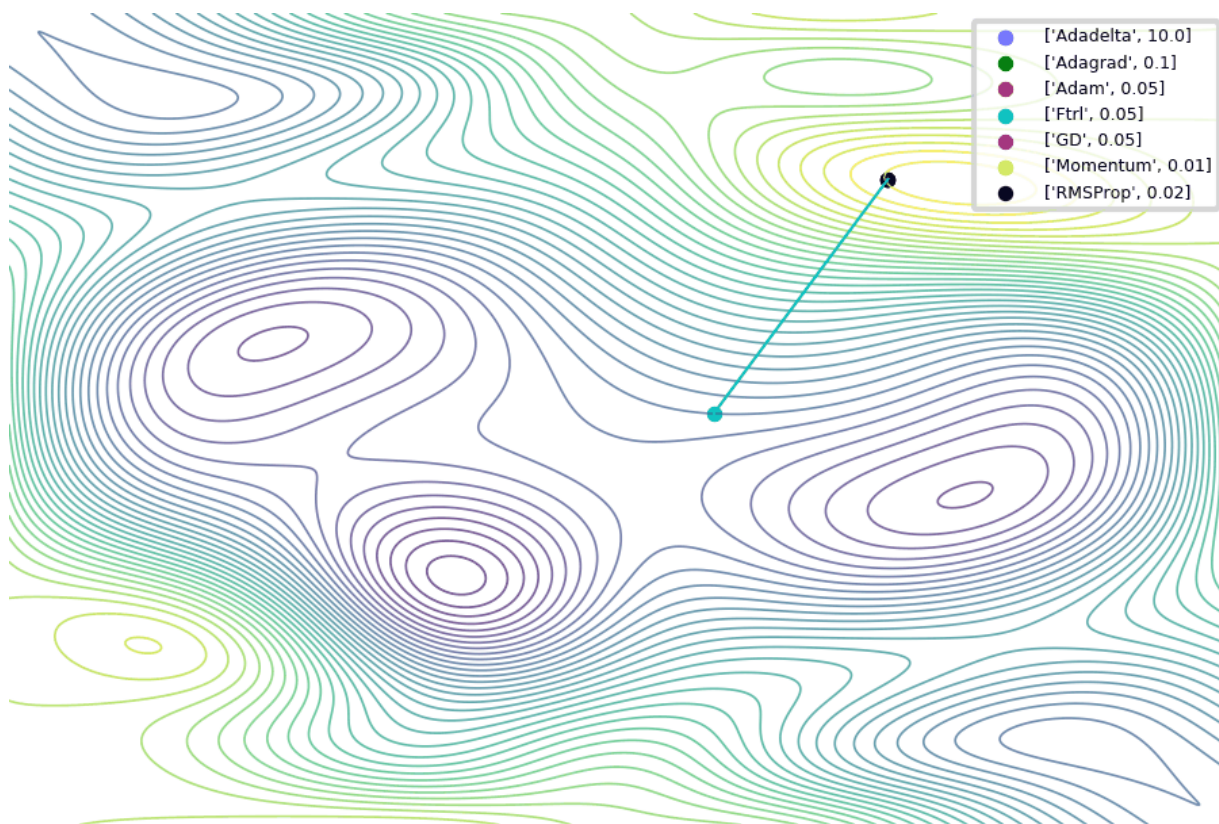
$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned}$$

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

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

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

Visual overview



Interactive visualization

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

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.

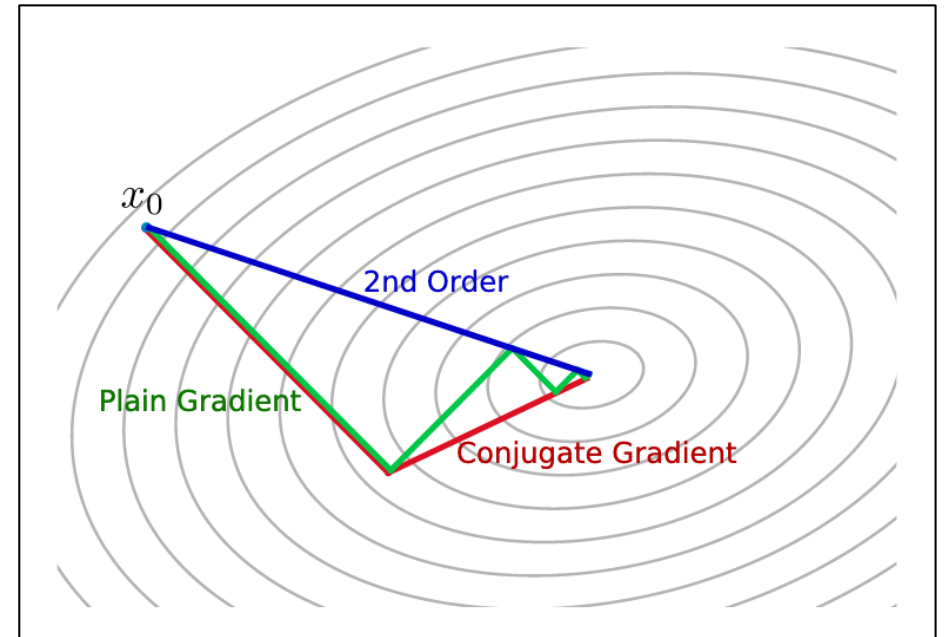
Oddity: even in “learning rate adjusting” optimizers like Adam, we add learning rate decays.

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 reduction 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:

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_{\theta} J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^\top \mathbf{H}(\theta - \theta_0)$$

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

$$\theta^* = \theta_0 - \mathbf{H}^{-1} \nabla_{\theta} J(\theta_0)$$

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*.

Newton's method

Newton's method is appropriate only when the Hessian is positive definite.

- near a saddle point, the Hessian are not all positive

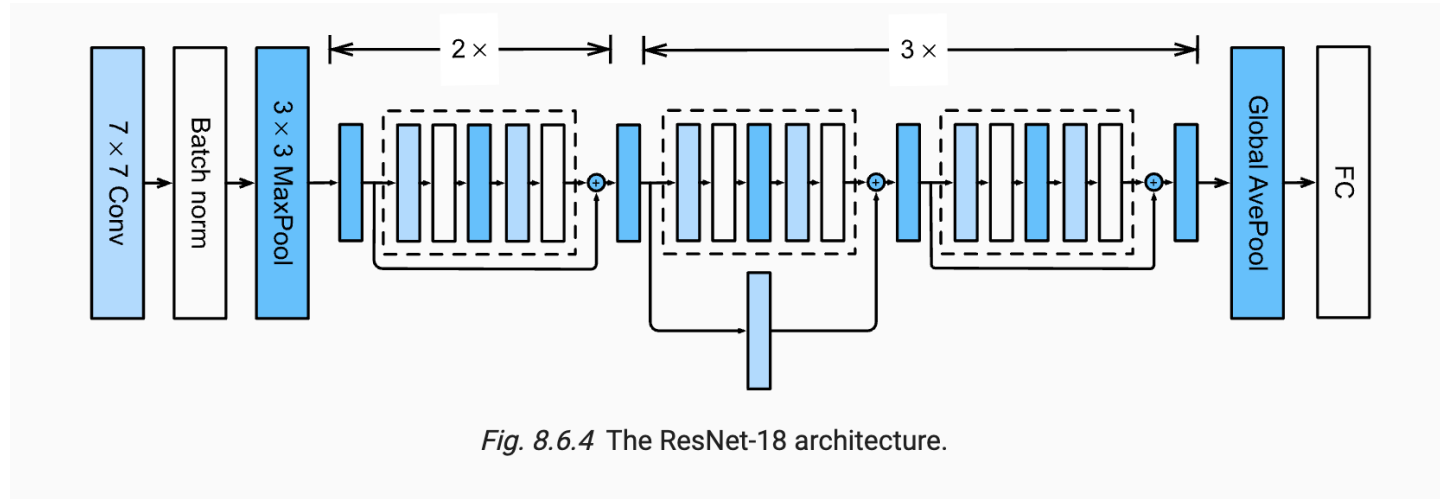
The regularized update becomes

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [H(f(\boldsymbol{\theta}_0)) + \alpha \mathbf{I}]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0)$$

With all these advantages, why is this not the go-to standard?

- Requires the inversion of the Hessian matrix at every training iteration.

Network initialization



Given a network, what is a better way to set the starting values?

1. All zeros.
2. Zeros for weights, random values for biases.
3. Random values for weights and biases.

Weight initialization

To prevent layer activation outputs from exploding or vanishing gradients.

Initialize weights correctly and our objective will be achieved in the least time.

Zero Initialization

- Leads to symmetric hidden layers.
- Makes your network no better than a linear model.
- Setting biases to 0 will not create any problems.

Random Initialization

- Breaks symmetry.
- Prevents neurons from learning the same features.

Random how?

Weights initialized **to preserve the variance** of the activations

- During the forward and backward computations.
- We want similar input and output variance because of modularity.

Weights must be initialized to be different from one another

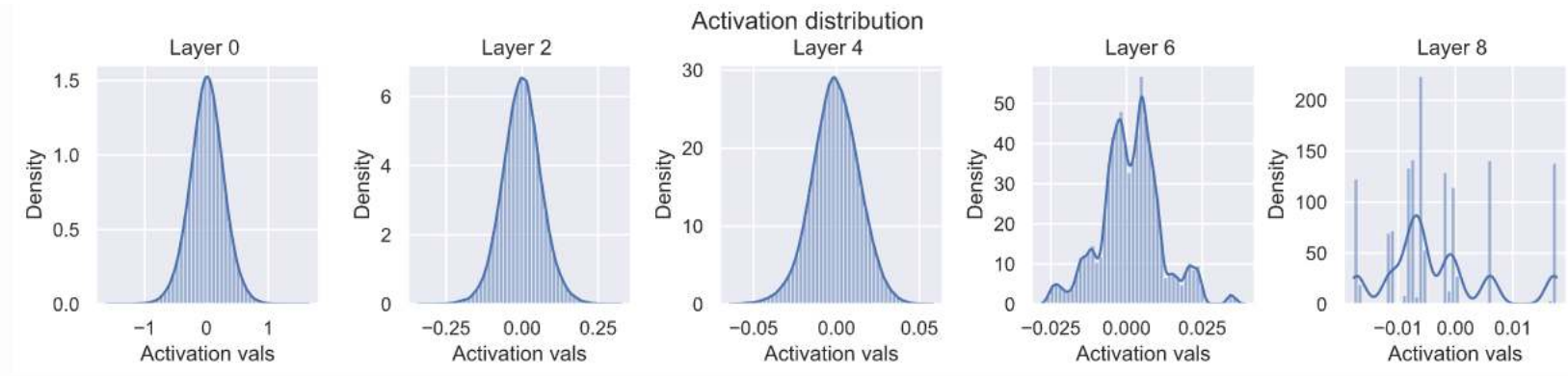
- Don't give same values to all weights (like all 0).
- All neurons (in one layer) generate same gradient → no learning.

Initialization depends on:

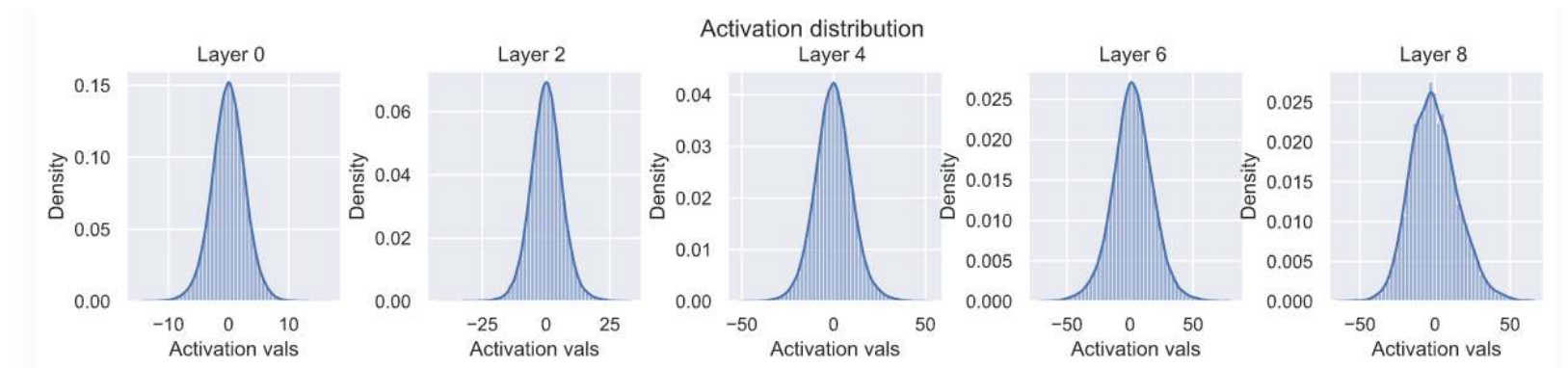
- non-linearities.
- data normalization.

Bad initialization will come back to haunt you

Initializing weights in every layer with same constant variance → can diminish variance in activations



Initializing weights in every layer with increasing variance → can explode the variance in activations



Preserving variance

For x and y independent

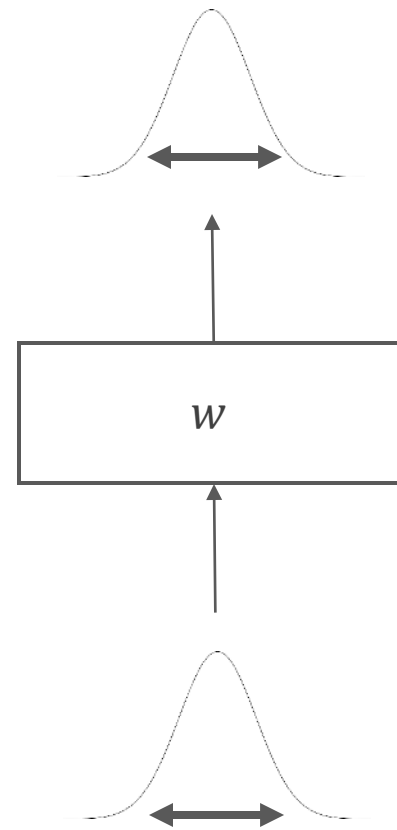
$$\text{var}(xy) = \mathbb{E}[x]^2 \text{var}(y) + \mathbb{E}[y]^2 \text{var}(x) + \text{var}(x) \text{var}(y)$$

For $a = wx \Rightarrow \text{var}(a) = \text{var}(\sum_i w_i x_i) = \sum_i \text{var}(w_i x_i) \approx d \cdot \text{var}(w_i x_i)$

$$\begin{aligned} \text{var}(w_i x_i) &= \mathbb{E}[x_i]^2 \text{var}(w_i) + \mathbb{E}[w_i]^2 \text{var}(x_i) + \text{var}(x_i) \text{var}(w_i) \\ &= \text{var}(x_i) \text{var}(w_i) \end{aligned}$$

Because we assume that x_i, w_i are unit Gaussians $\rightarrow \mathbb{E}[x_i] = \mathbb{E}[w_i] = 0$

So, the variance in our activation $\text{var}(a) \approx d \cdot \text{var}(x_i) \text{var}(w_i)$



Preserving variance

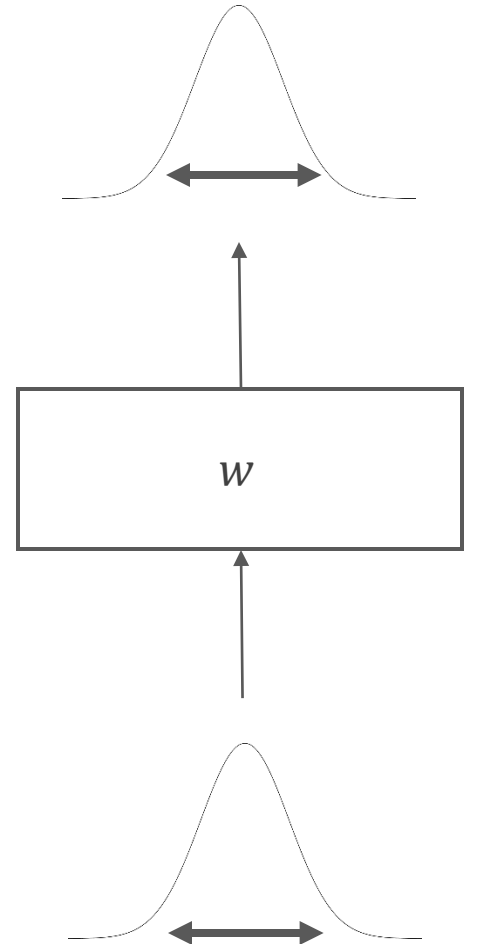
Since we want the same input and output variance

$$\text{var}(a) = d \cdot \text{var}(x_i) \text{var}(w_i) \Rightarrow \text{var}(w_i) = \frac{1}{d}$$

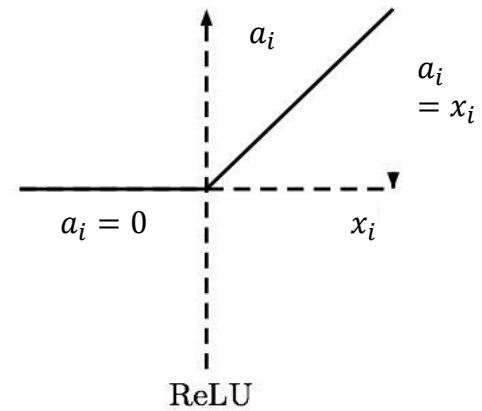
Draw random weights from

$$w \sim N(0, 1/d)$$

where d is the number of input variables to the layer



Kaiming Initialization



ReLU's return 0 half of the time: $\mathbb{E}[w_i] = 0$ but $\mathbb{E}[x_i] \neq 0$

$$\text{var}(w_i x_i) = \text{var}(w_i)(\mathbb{E}[x_i]^2 + \text{var}(x_i))$$

$$= \text{var}(w_i) \mathbb{E}[x_i^2] \quad (\text{var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2)$$

$$\mathbb{E}[x_i^2] = \int_{-\infty}^{\infty} x_i^2 p(x_i) dx_i = \int_{-\infty}^{\infty} \max(0, a_i)^2 p(a_i) da_i = \int_0^{\infty} a_i^2 p(a_i) da_i$$

$$= 0.5 \int_{-\infty}^{\infty} a_i^2 p(a_i) da_i = 0.5 \cdot \mathbb{E}[a_i^2] = 0.5 \cdot \text{var}(a_i)$$

Draw random weights from $w \sim N(0, 2/d)$ – **Kaiming Initialization**

Xavier initialization

For tanh: initialize weights from $U \left[-\sqrt{\frac{6}{d_{l-1}+d_l}}, \sqrt{\frac{6}{d_{l-1}+d_l}} \right]$

d_{l-1} is the number of input variables to the tanh layer and d_l is the number of the output variables

For a sigmoid $U \left[-4 \cdot \sqrt{\frac{6}{d_{l-1}+d_l}}, 4 \cdot \sqrt{\frac{6}{d_{l-1}+d_l}} \right]$

Random networks are already great deep learners

What's Hidden in a Randomly Weighted Neural Network? Ramanujan et al. 2019

*Hidden in a **randomly weighted Wide ResNet-50** we find a subnetwork (with random weights) that is smaller than, but matches the performance of a ResNet-34 trained on ImageNet [4]. Not only do these “untrained subnetworks” exist, but we provide an algorithm to effectively find them.*

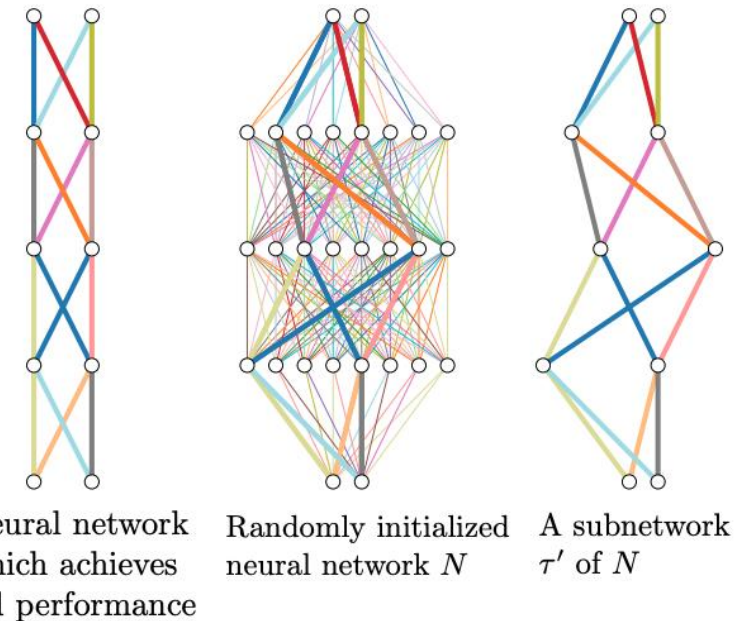


Figure 1. If a neural network with random weights (center) is sufficiently overparameterized, it will contain a subnetwork (right) that perform as well as a trained neural network (left) with the same number of parameters.

Next lecture

Lecture	Title
1	Intro and history of deep learning
3	Deep learning optimization I
5	Convolutional Neural Networks I
7	Attention
9	Self-supervised and vision-language learning
11	The oddities of deep learning
13	Deep learning for videos

Lecture	Title
2	Manually forward, automatically backward
4	Deep learning optimization II
6	Convolutional Neural Networks II
8	Graph Neural Networks
10	Auto-encoding and generation
12	Non-Euclidean deep learning
14	Q&A

Next lecture from 9:00-10:15 continuous, I am in a PhD committee at 11 in the city centre!

Thank you!