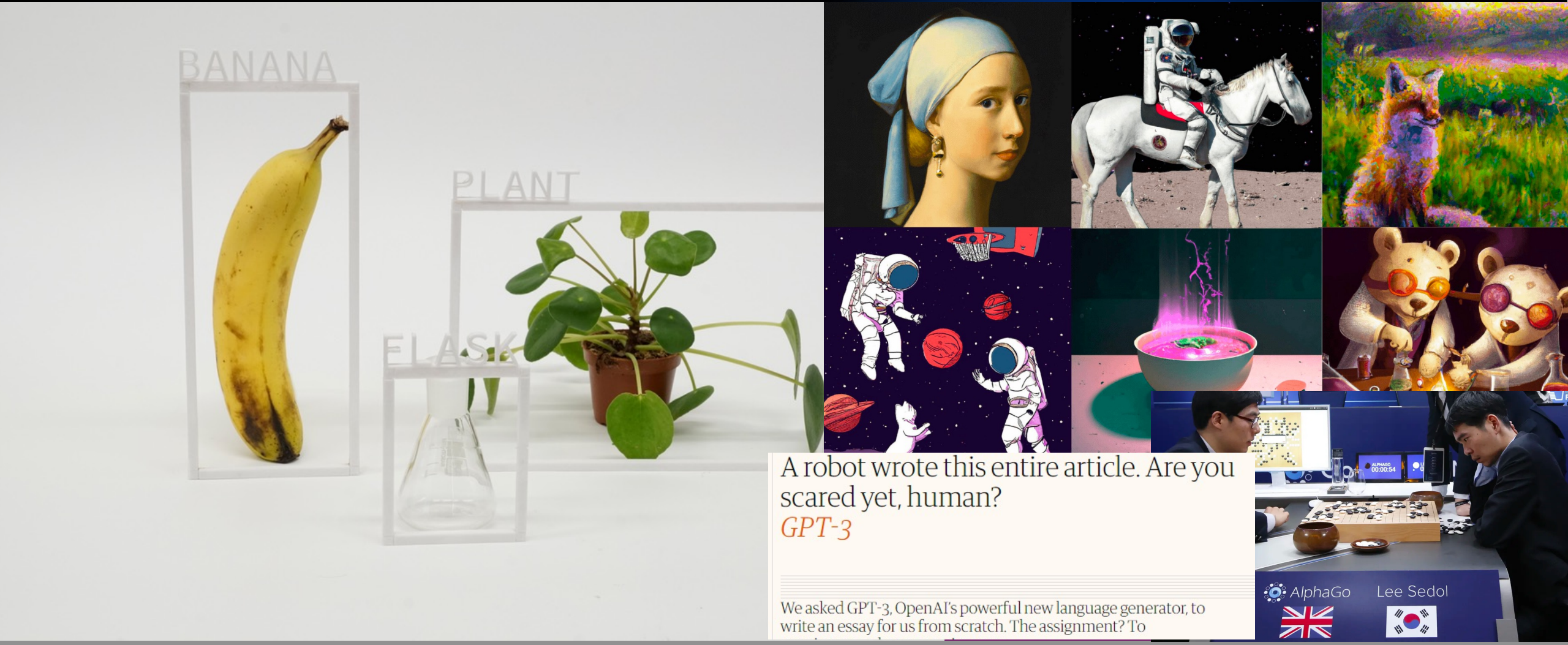
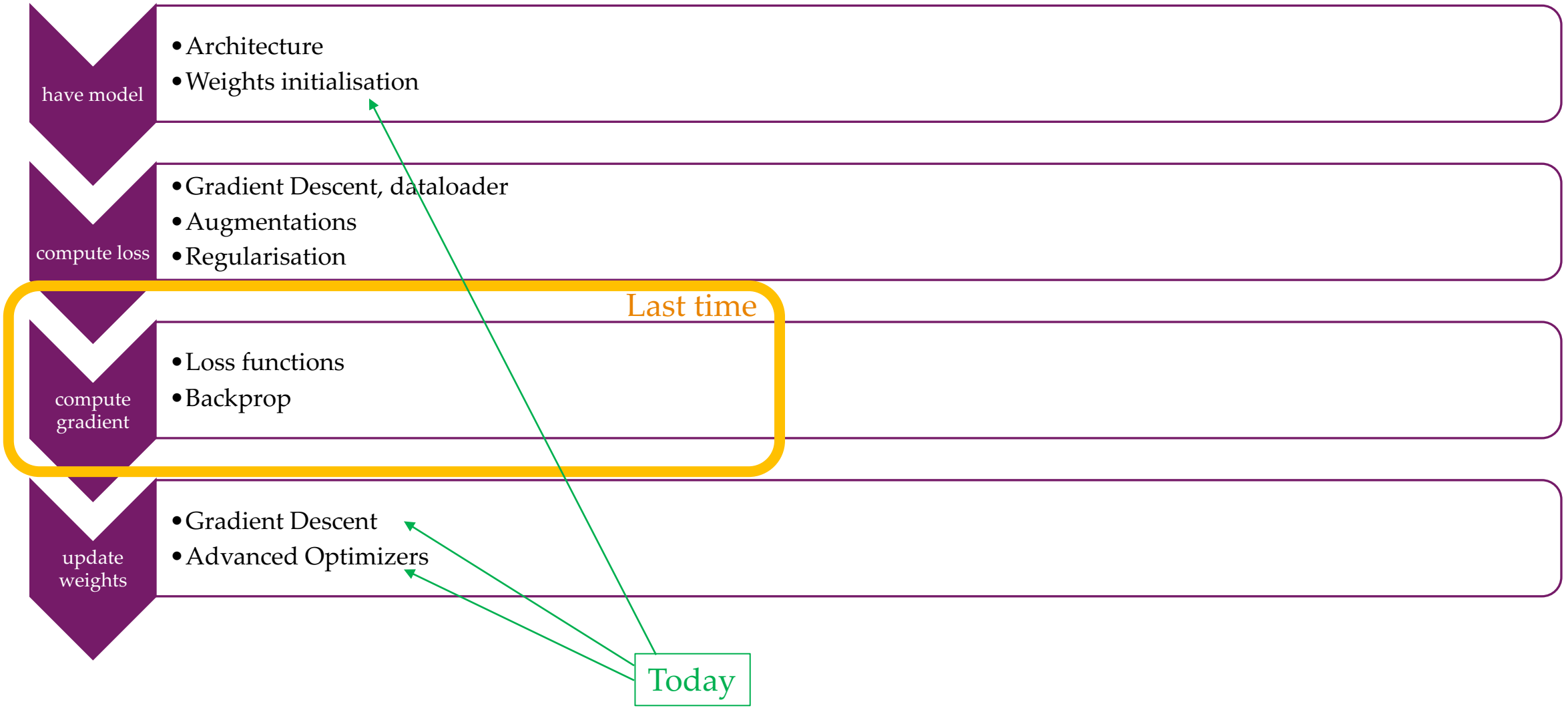


Lecture 3: Deep Learning Optimizations I

Deep Learning 1 @ UvA

Yuki M. Asano

Optimizing neural networks

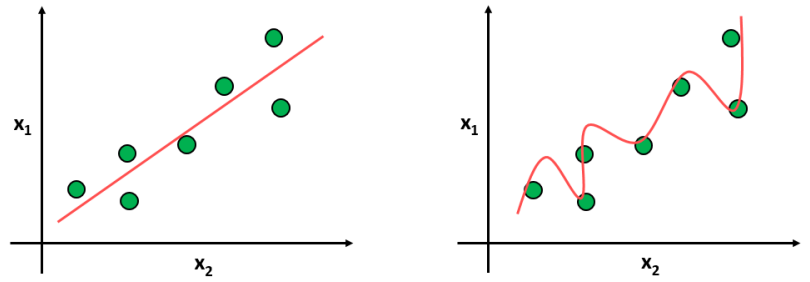


Lecture overview

- Gradient descent
- Advanced optimizers
- Initialization

Optimization *vs.* 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
  - Eg, 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.
  - Otherwise we're "cramming for the exam"



Risk minimization

- 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) make sure the model predictions are not too wrong
- While (2) not being "too geared/optimized" towards the observed data
- The expectation is taken over the true underlying distribution p_{data}
  - Usually, the true distribution p_{data} is not available

Minimizing the *empirical* risk, simply, the "loss"

- In practice having p_{data} is not possible.
- We only have a training set of data

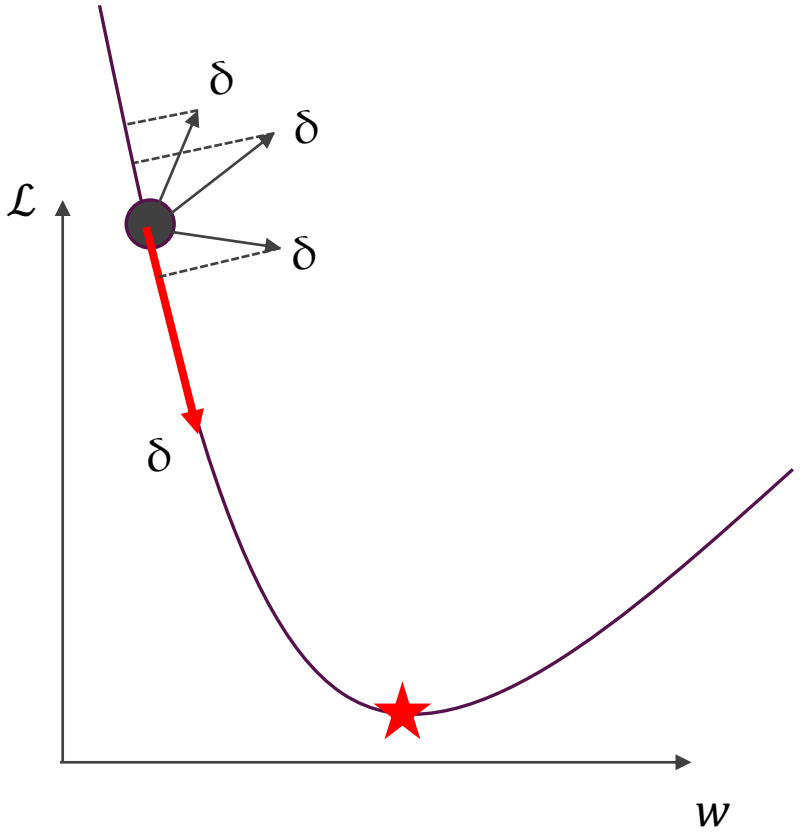
$$\min_{\mathbf{w}} \mathbb{E}_{\boldsymbol{x},\mathrm{y}\sim\hat{p}_{data}}[\mathcal{L}(f(\boldsymbol{x},\boldsymbol{w}),y)] + \lambda\Omega(\boldsymbol{w})$$

  - where $\hat{p}_{data}$ is the empirical data distribution
  - $\hat{p}_{data}$ is defined by a set of training examples.
- To minimize any function, we take a step δ
  - Our best bet: the (negative) gradient

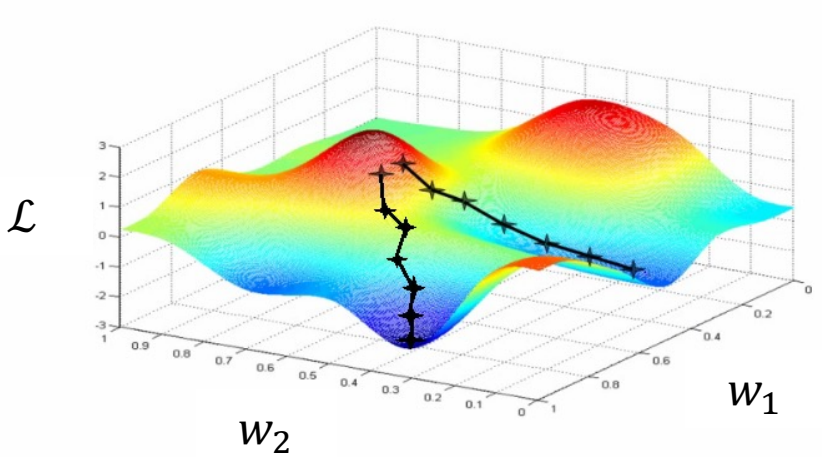
$$-\sum \frac{d}{dw}\mathcal{L}(f(\boldsymbol{x},\boldsymbol{w}),y)$$

How to compute this, we've covered in the last lecture

  - Gradient descent based on optimization



Gradient descent

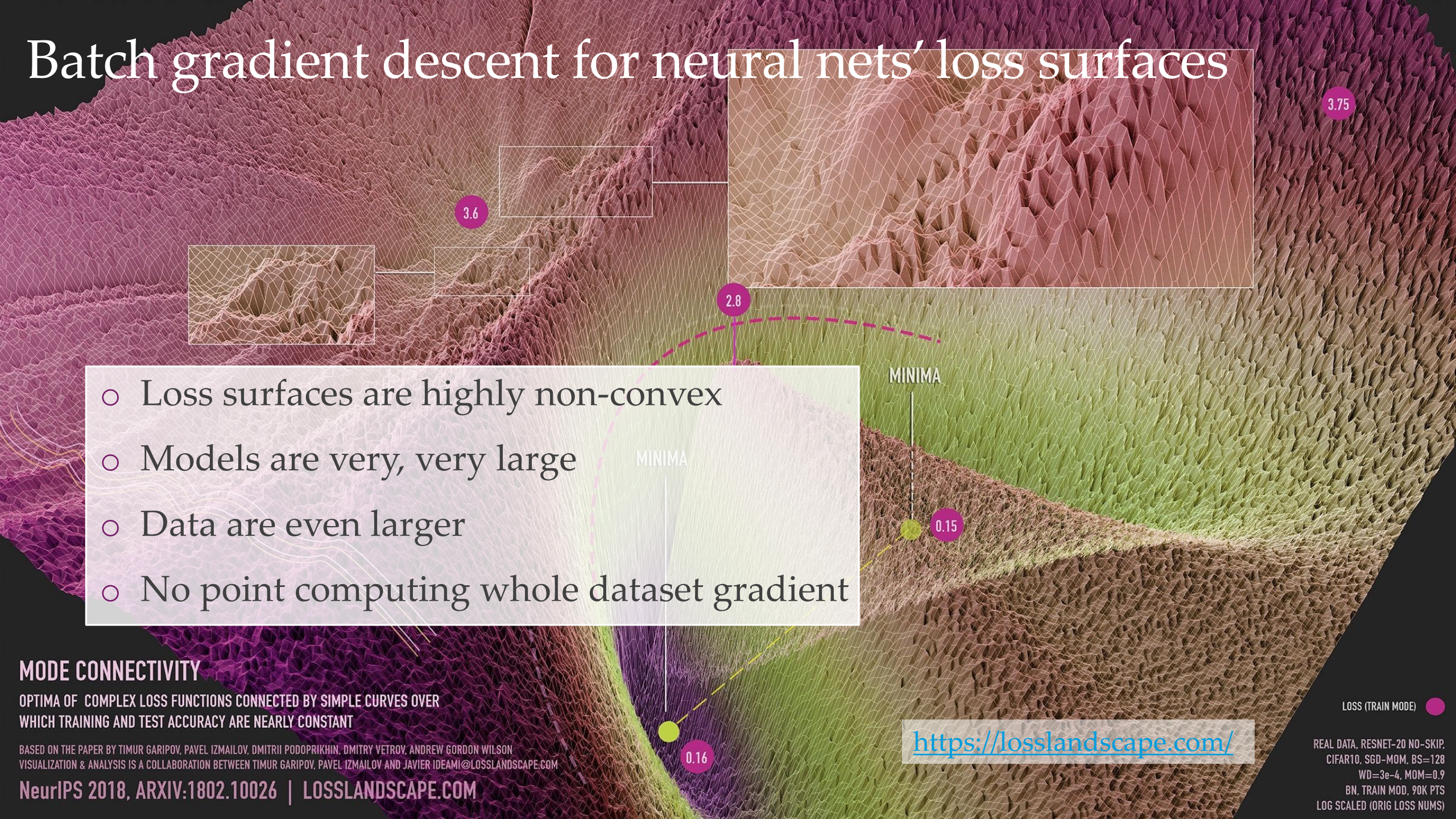


- Gradient descent is the iterative method

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

  - $\boldsymbol{w}^{(t)}$ are the parameters of our neural network at t
  - $\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}}$ is the gradient of the loss w.r.t. the parameters
- If $\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}}$ is computed from whole dataset → gradient descent
- If $\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}}$ is computed from a mini-batch of samples → "SGD"
- If $\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}}$ is computed from a *single* sample → Stochastic/online gradient descent
- *On non-convex landscapes no guarantee for global optimum*

Batch gradient descent for neural nets' loss surfaces

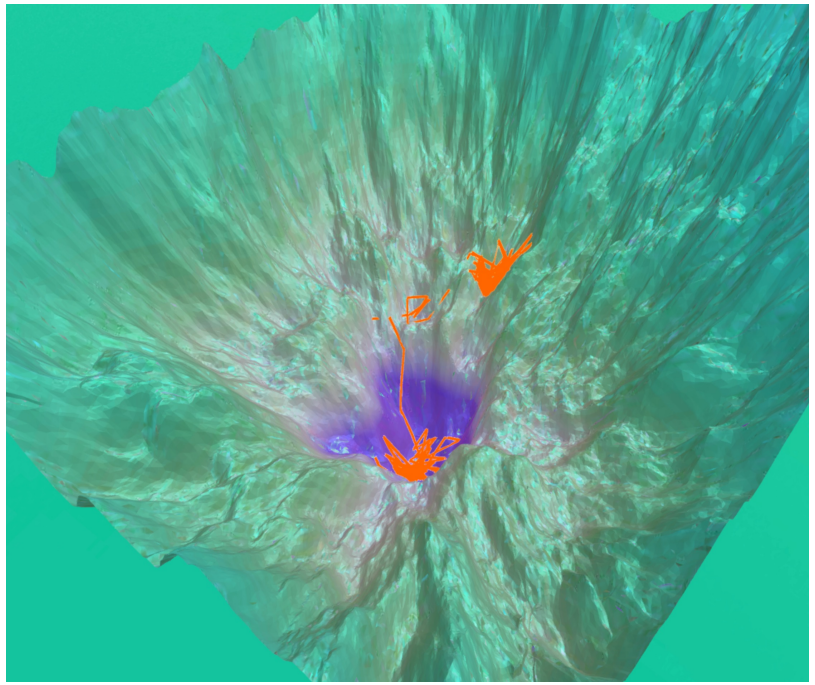


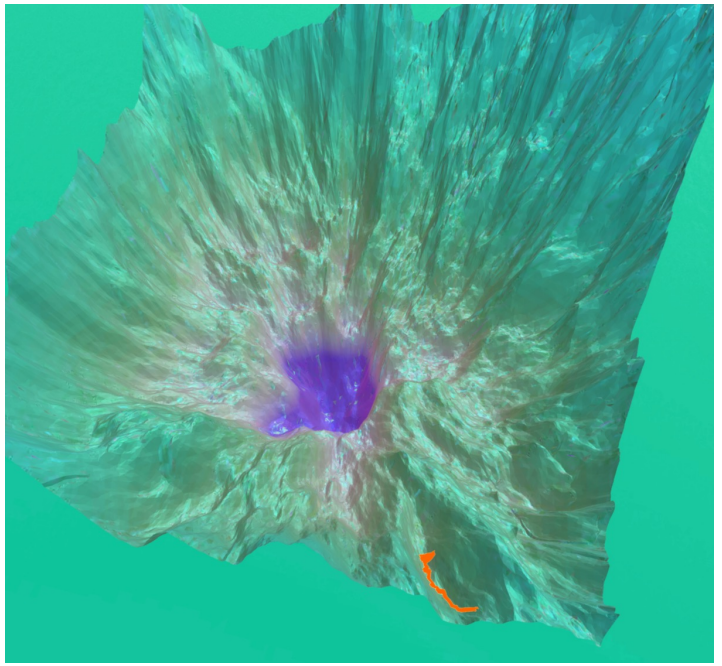
- Loss surfaces are highly non-convex
- Models are very, very large
- Data are even larger
- No point computing whole dataset gradient

https://losslandscape.com/

https://losslandscape.com/explorer

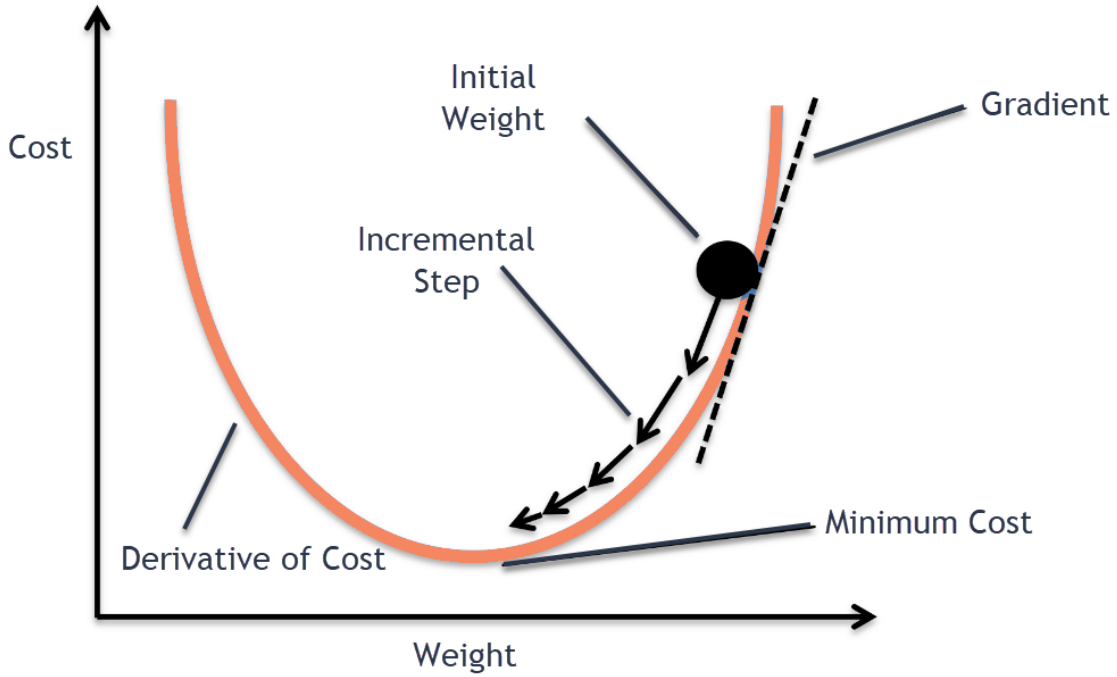
- Explore: play around with momentum, lr etc.





Gradient descent vs. Stochastic Gradient Descent

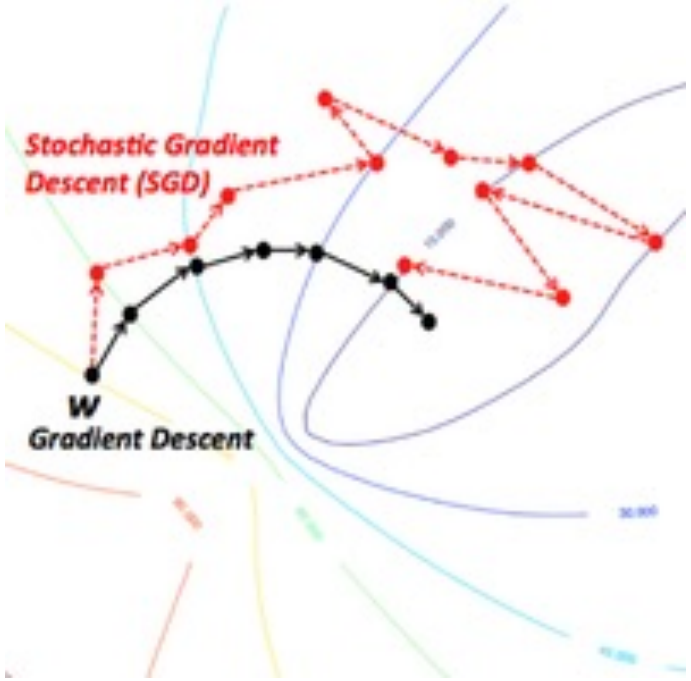
Gradient descent



Link

Calculate the gradients for the whole dataset to perform just one update

Stochastic gradient descent



Link

Perform a parameter update for *each* mini-batch

SGD properties

- SGD approximates the true gradient
  - Estimating whole training set with a single sample (a “mini-batch” or just “batch”)
- Recall standard error: $\sigma/\sqrt{n_b}$
  - where σ is the variance in $p(\boldsymbol{x}, y)$, and n_b is the batch size
  - the reduction of standard error is the square root of the increase of sample size
  - To compute 2x more accurate gradients, we need 4x data points
- Batchsize n_b is *highly dependent* on task. Videos/large images ~8, Self-supervised Learning: 4K or more

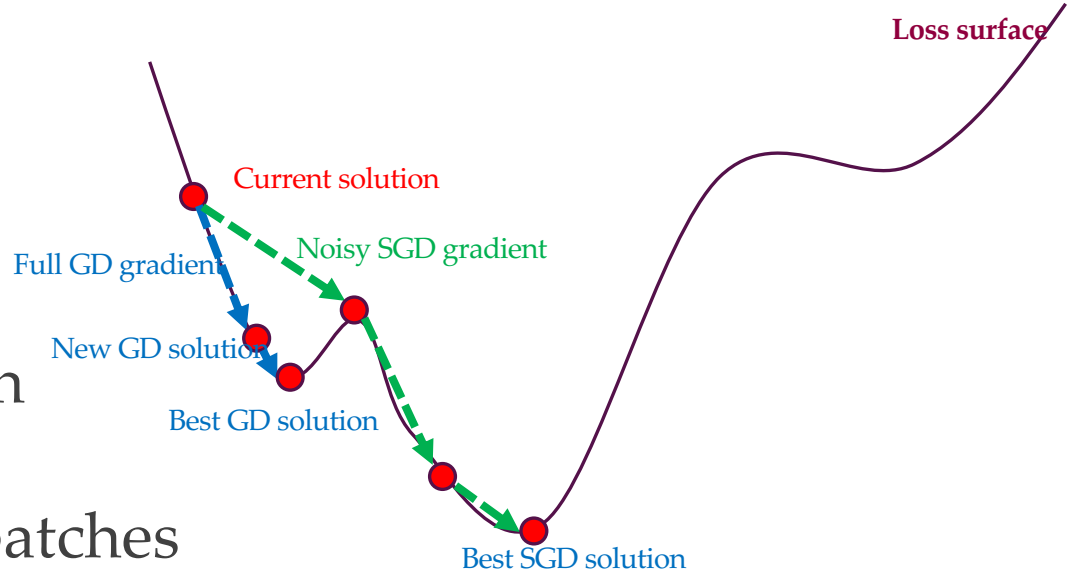
Quiz

In the previous slide I said batch size of ~8 for videos,
while for other data we might use 4K. Why?

1) The high correlation between frames requires lower batchsizes
2) The GPUs cannot handle much more
3) Videos are processed at higher resolutions than images

SGD properties

- Randomness reduces overfitting
  - No guarantees though
  - Shuffling is important!
  - New mini-batch combinations per epoch
    (one epoch = going through train dataset completely)
- Must make sure of class/data balance in batches
- Try this out yourself:




```
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)
```

Batch size

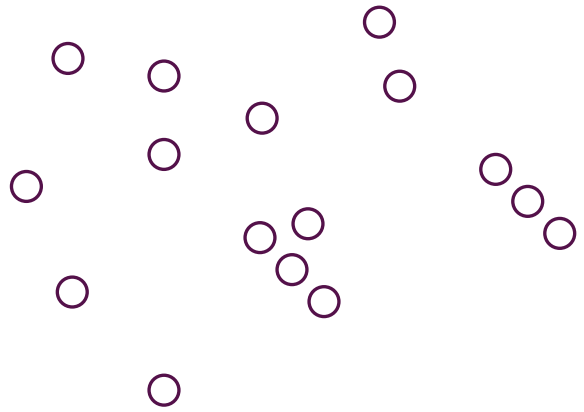
- Large batch size
  - more accurate estimate of the gradient, but less than linear returns
- Extremely small batch size (BS) underutilizes hardware, why?
  - Re-use of computations (matrix multiplies)
- Power of 2 batch size achieves better runtime for GPUs
  - 32 ~ 8192
- Small batches can offer regularizing effect
  - add noise to the learning process
  - small batches requires very small learning rate, longer total runtime
- General rule: Batchsize ~ learning rate (ie double BS -> double the LR
  (*Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour.* Goyal et al. 2017: Batchsize of 8K)
- Generally: use largest batch size that fits on GPU, but stick to powers of 2.

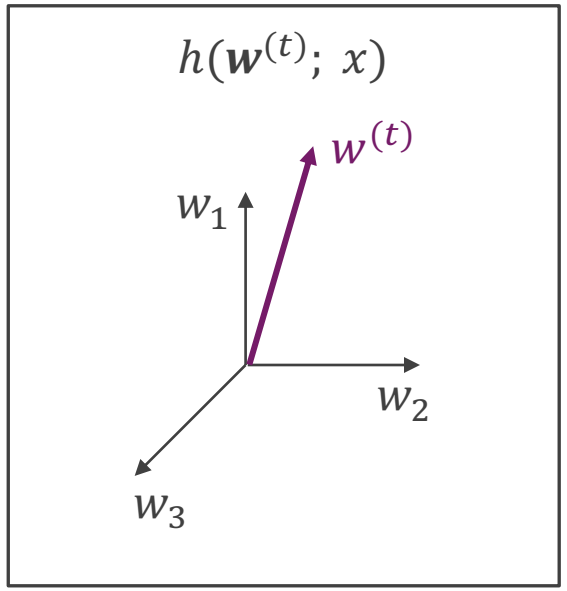
Why does mini-batch SGD work?

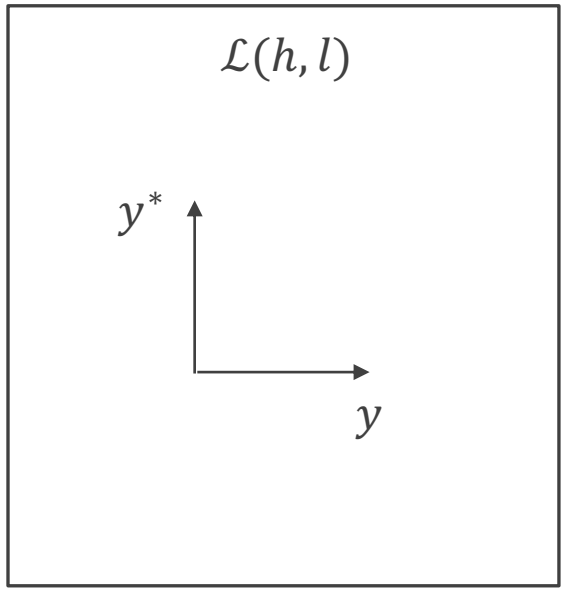
- The "true" gradient of the cost function is the expectation of the gradient g over the true data generating distribution
- Reduced sample size does not imply reduced gradient quality
- The training samples may have lots of noises or outliers or biases.
  - A randomly sampled minibatch may reflect the true data generating distribution better (or worse)
- Might be easier to trap a *regular gradient* into a local minima
  - than to trap the *temporarily random gradient* computed with minibatch.

Stochastic gradient-based optimization

- An iterative method for optimizing an objective function
- It is local, so prone to local minima
- But efficient and effective, so it gets the job done

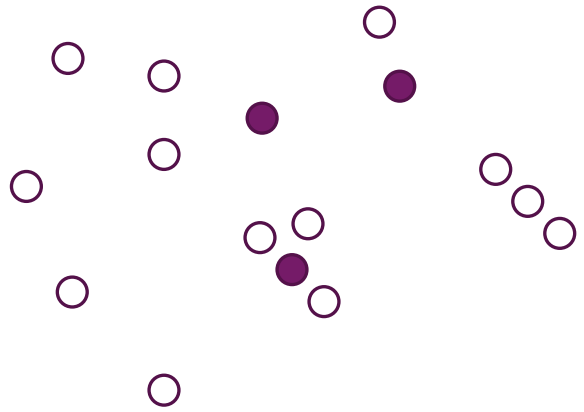




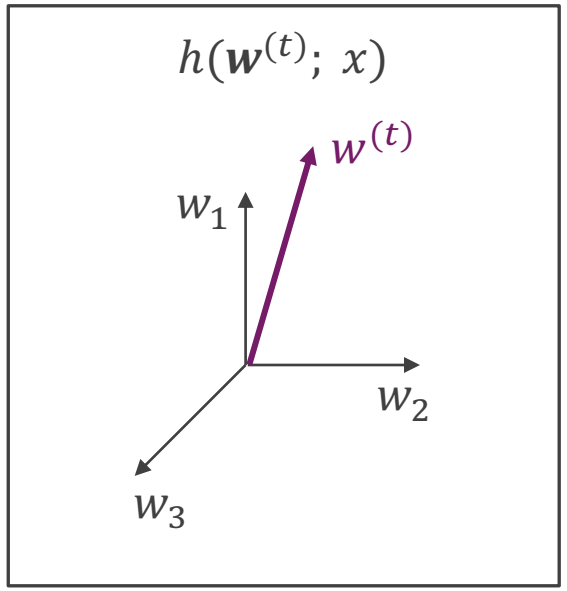


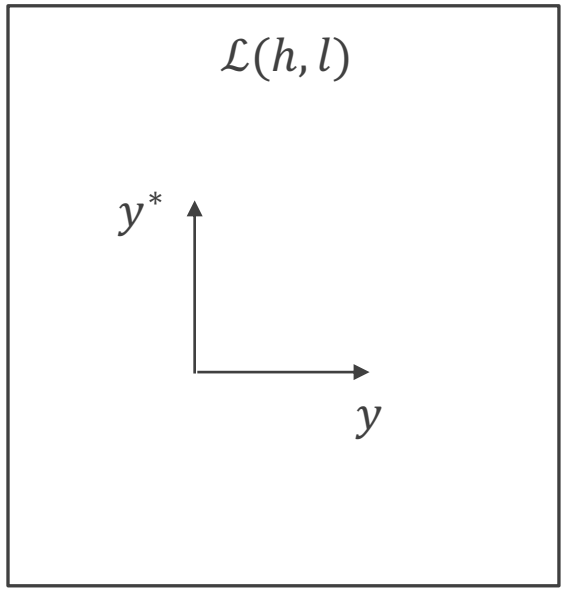
Stochastic gradient-based optimization

- An iterative method for optimizing an objective function
- It is local, so prone to local minima
- But efficient and effective, so it gets the job done



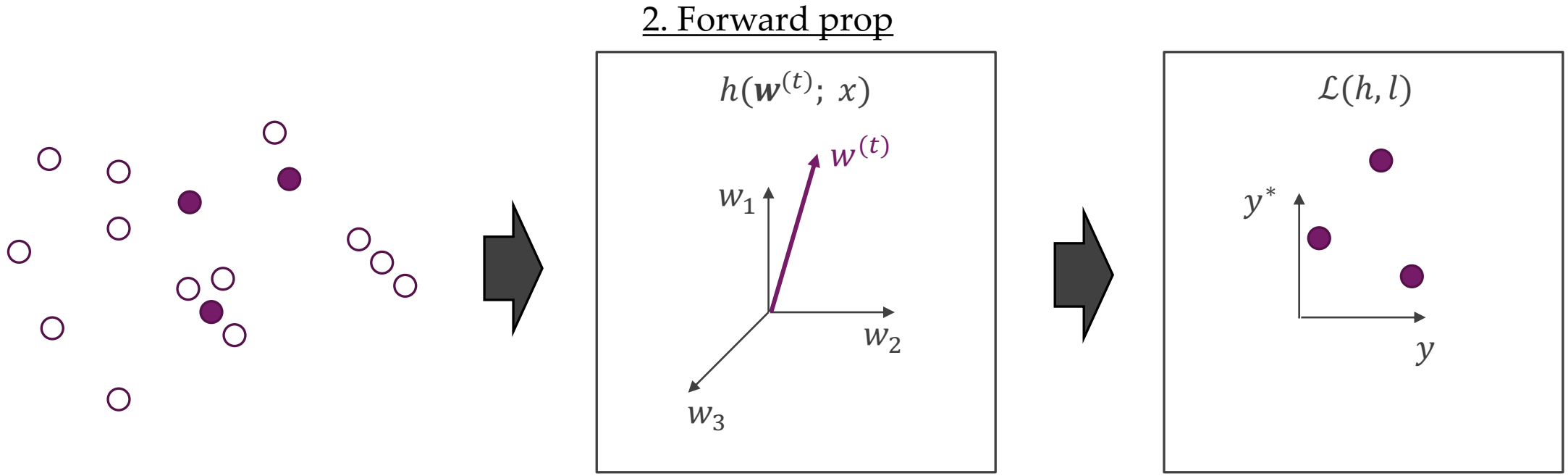
1. Sample mini-batch





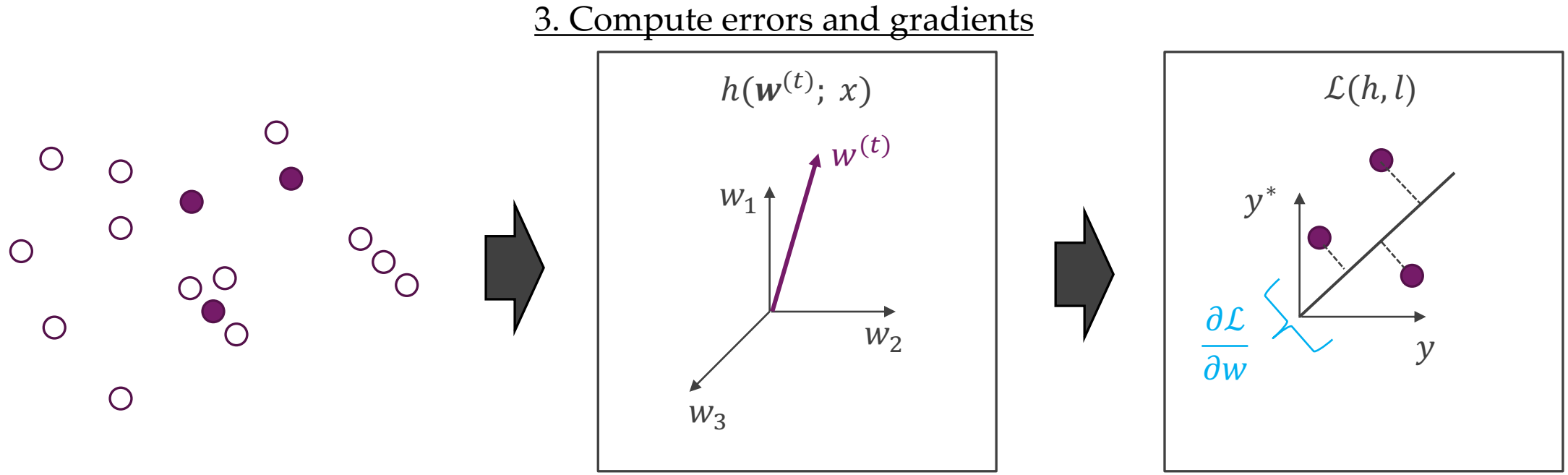
Stochastic gradient-based optimization

- An iterative method for optimizing an objective function
- It is local, so prone to local minima
- But efficient and effective, so it gets the job done



Stochastic gradient-based optimization

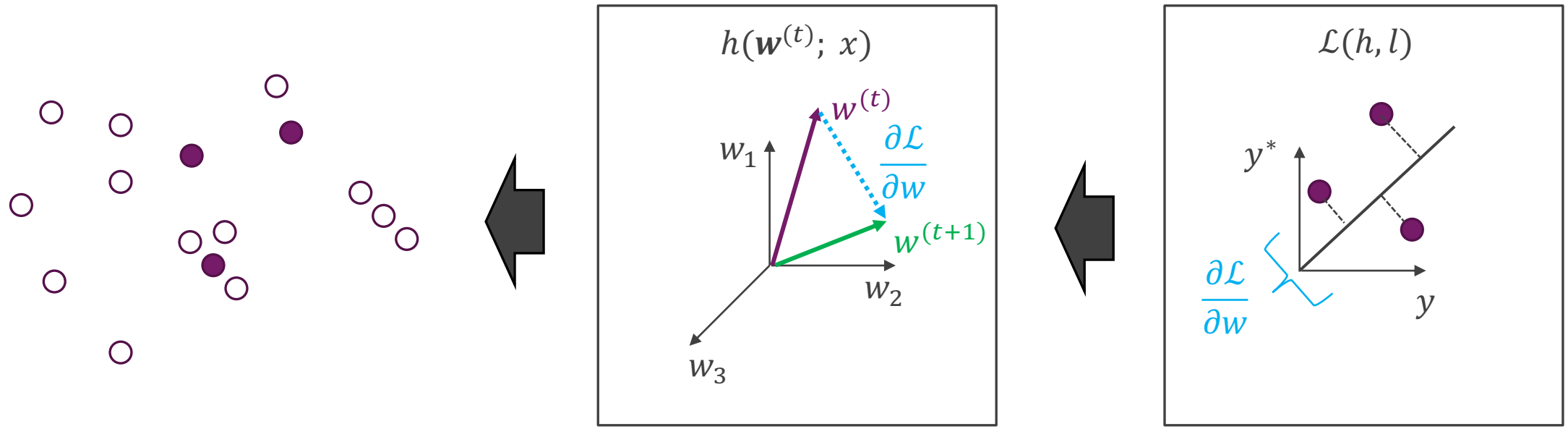
- An iterative method for optimizing an objective function
- It is local, so prone to local minima
- But efficient and effective, so it gets the job done



Stochastic gradient-based optimization

- An iterative method for optimizing an objective function
- It is local, so prone to local minima
- But efficient and effective, so it gets the job done

4. Update model parameters and repeat



Gradient descent *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

Gradient descent *vs.* stochastic gradient descent

- In practice: everyone, always, anytime uses batch gradient descent with stochastic batches and simply refers to it as “SGD” (if at all).
- Adam (we will see) is also based on stochastic gradient descent

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

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

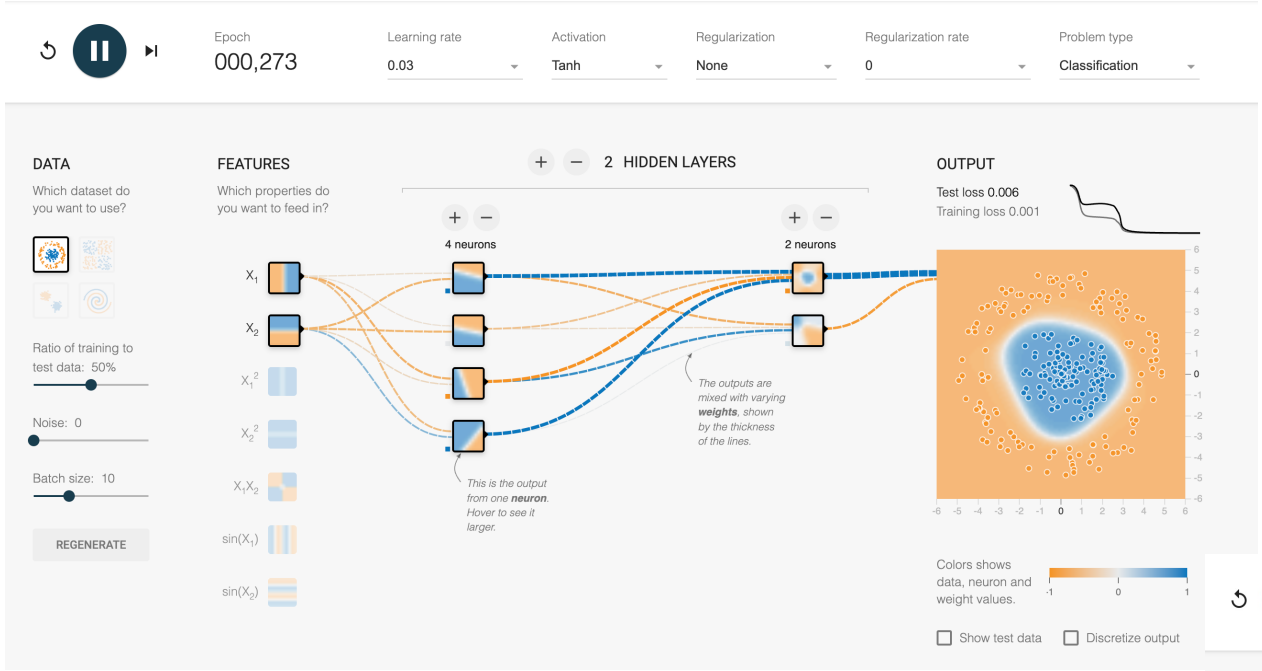
  - where η is the step size or learning rate.

Let's see this in practice

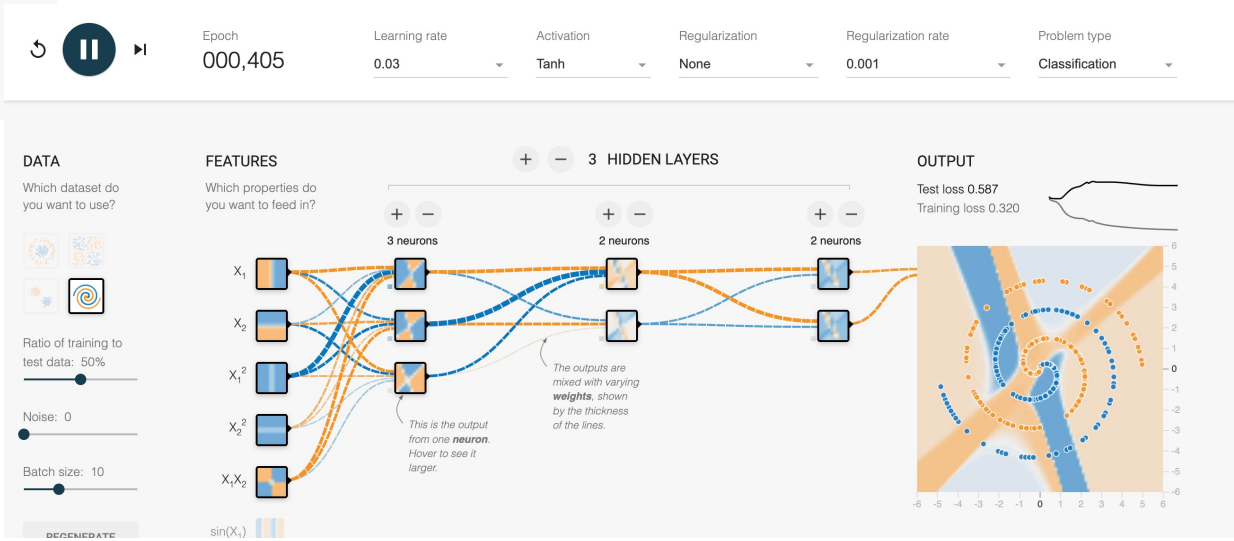
- https://playground.tensorflow.org
- Vary learning rate
- Vary MLP width, depth
- Vary Batchsize

Interactive session

https://playground.tensorflow.org/

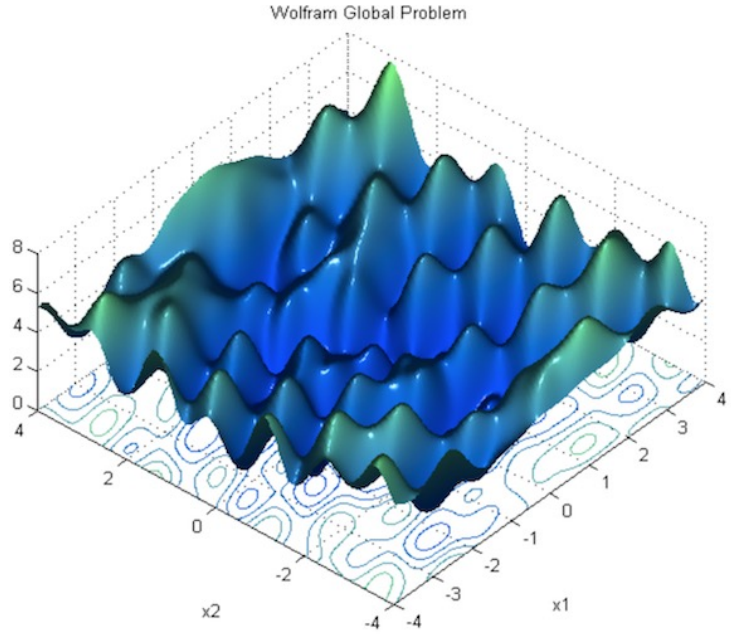


Vary learning rate
Vary MLP width, depth
Vary Batchsize



Challenges in optimization

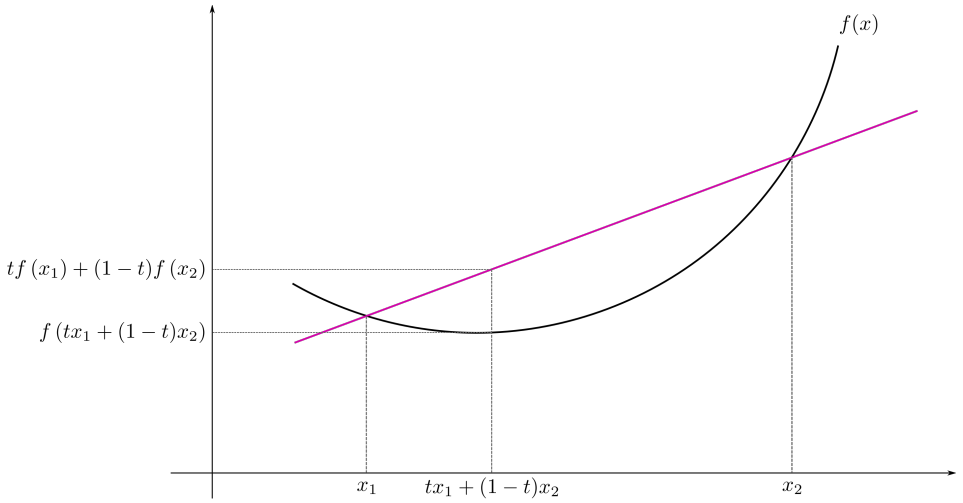
- Neural network training is **non-convex** optimization
  - involves a function which has multiple optima
  - Extremely difficult to locate the global optimum.
- The associated problems
  - How do we avoid getting stuck in local optima?
  - What is a reasonable learning rate to use?
  - What if the loss surface morphology changes?
  - …



Why are NN losses not convex?

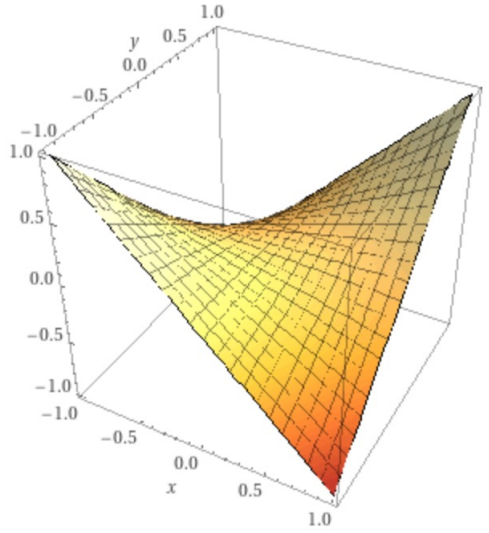
- Convex means:

$$f\left(tx_1 + (1-t)x_2\right) \leq tf\left(x_1\right) + (1-t)f\left(x_2\right)$$



Graph: cc Eli Osherovich

Consider y = x_1 * x_2
(doesn't even have a non-linearity)



Cannot have a string that connects any two points but stays above curve. --> not convex

Challenges in optimization

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

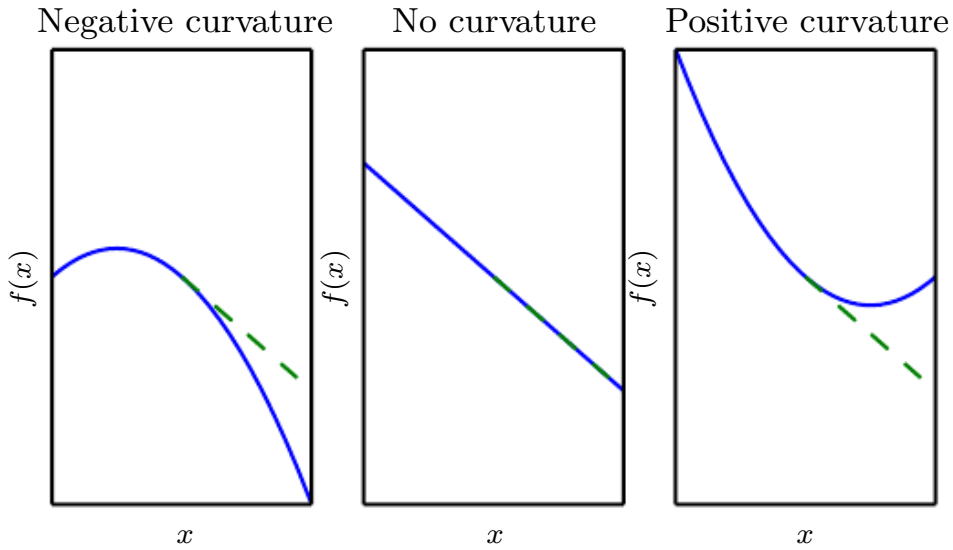
1. Ill-conditioning

- Hessian matrix *H*
  - A 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 a symmetric matrix
  - The Hessian matrix of a function *f* is the Jacobian matrix of the gradient of the function *f*

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

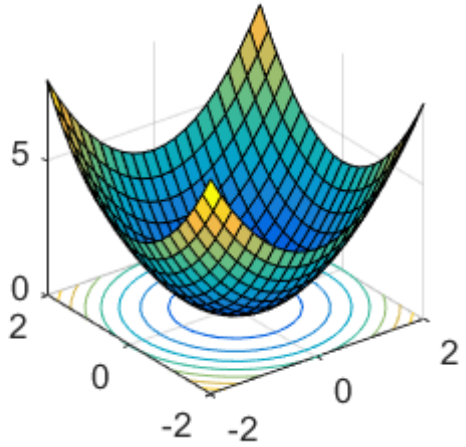
1. Ill-conditioning

- Curvature is determined by the second derivative
  - *Negative curvature:* the 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

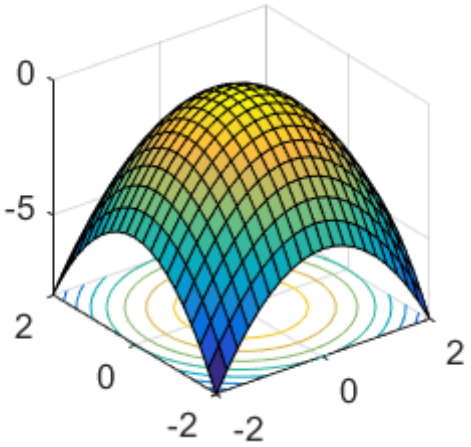


1. Ill-conditioning

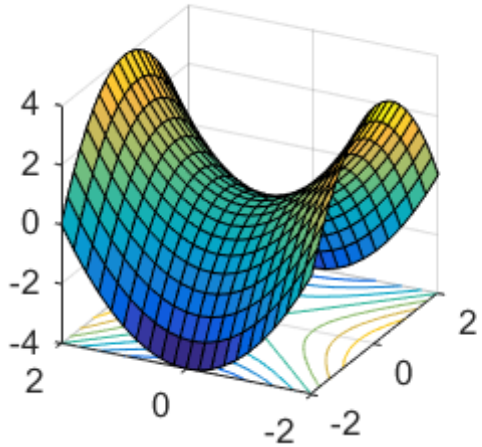
- Critical points – Hessian matrix
  - *A local minimum:* positive definite (all its eigenvalues are positive)
  - *A local maximum:* negative definite (all its eigenvalues are negative)
  - *A saddle point:* at least one eigenvalue is positive and at least one eigenvalue is negative. Why is this bad?



local minimum



local maximum



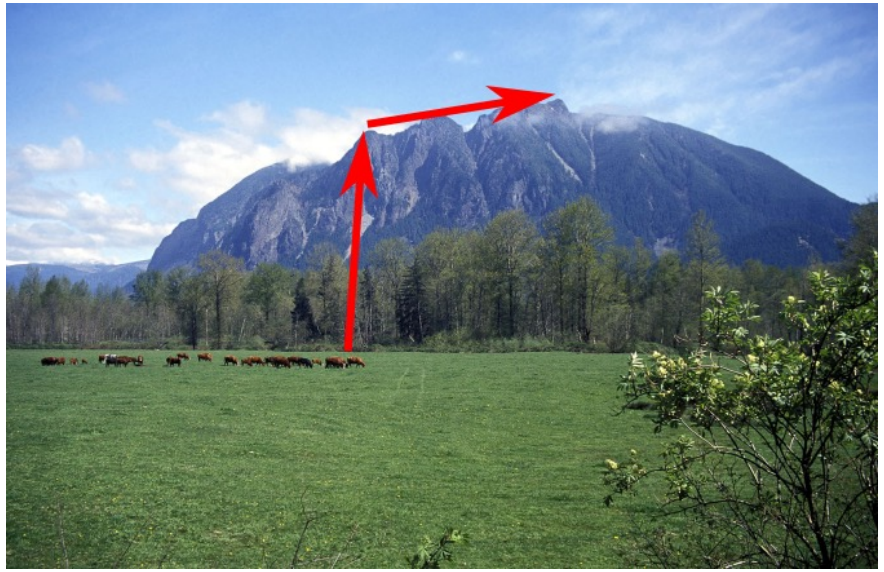
saddle point

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).
- The condition number measure show much the second derivatives differ from each other.
- With a poor (large) condition number, gradient descent performs poorly.
- It also makes it difficult to choose a good step size.



Gradient mostly points into 1 direction.
For n-dimension mountain --> n steps

1. Ill-conditioning

- We can analyze possible behaviors of the neural network loss function
  - Resort to the 2nd order Taylor expansion around the current weight $\mathbf{w}'$

$$\mathcal{L}(\boldsymbol{w}) = \mathcal{L}(\boldsymbol{w}') + \boldsymbol{g}(\boldsymbol{w} - \boldsymbol{w}') + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}')^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \mathbf{w}') \text{ where } \boldsymbol{g} = \frac{d\mathcal{L}}{d\boldsymbol{w}}$$

- If we analyze the loss around the current weight $\mathbf{w}'$ plus a small step

$$\boldsymbol{w} \leftarrow \boldsymbol{w}' - \varepsilon\boldsymbol{g} \Rightarrow \mathcal{L}(\boldsymbol{w}' - \varepsilon\boldsymbol{g}) \approx \mathcal{L}(\boldsymbol{w}') - \varepsilon\boldsymbol{g}^{\mathrm{T}}\boldsymbol{g} + \frac{1}{2}\varepsilon^2\boldsymbol{g}^{\mathrm{T}}\boldsymbol{g}$$

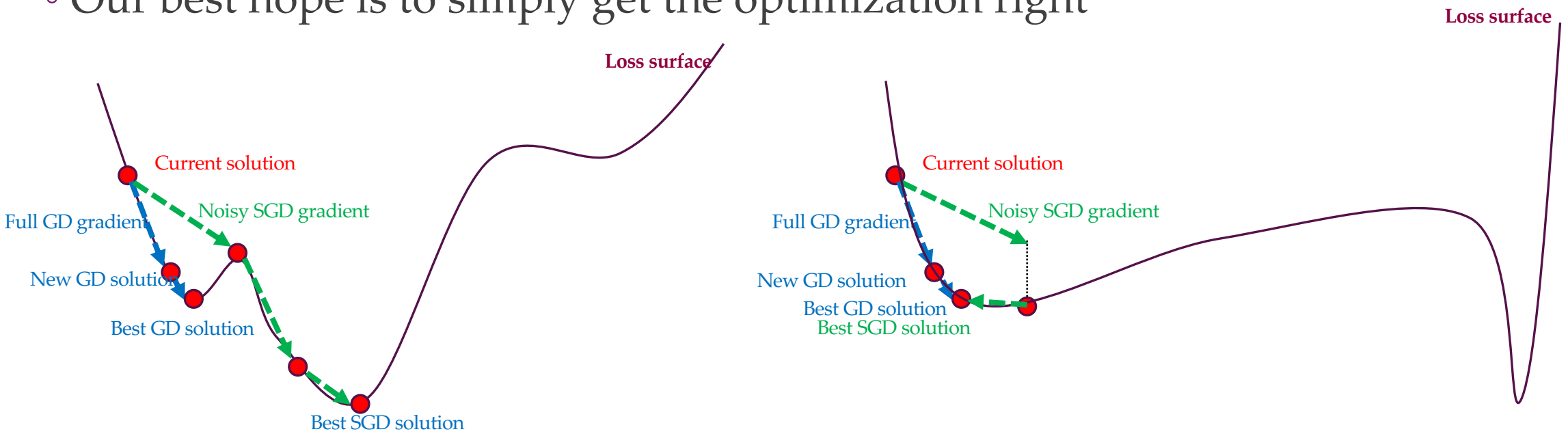
- There are cases where $\boldsymbol{g}$ is "strong" but $\frac{1}{2}\varepsilon^2\boldsymbol{g}^T\boldsymbol{H}\boldsymbol{g} > \varepsilon\boldsymbol{g}^{\mathrm{T}}\boldsymbol{g}$
  - In these cases, the loss would still go higher after we take a gradient step
- This ill-conditioning is reflected in error landscapes *which contain many saddle points and flat areas.*

2. Local minima

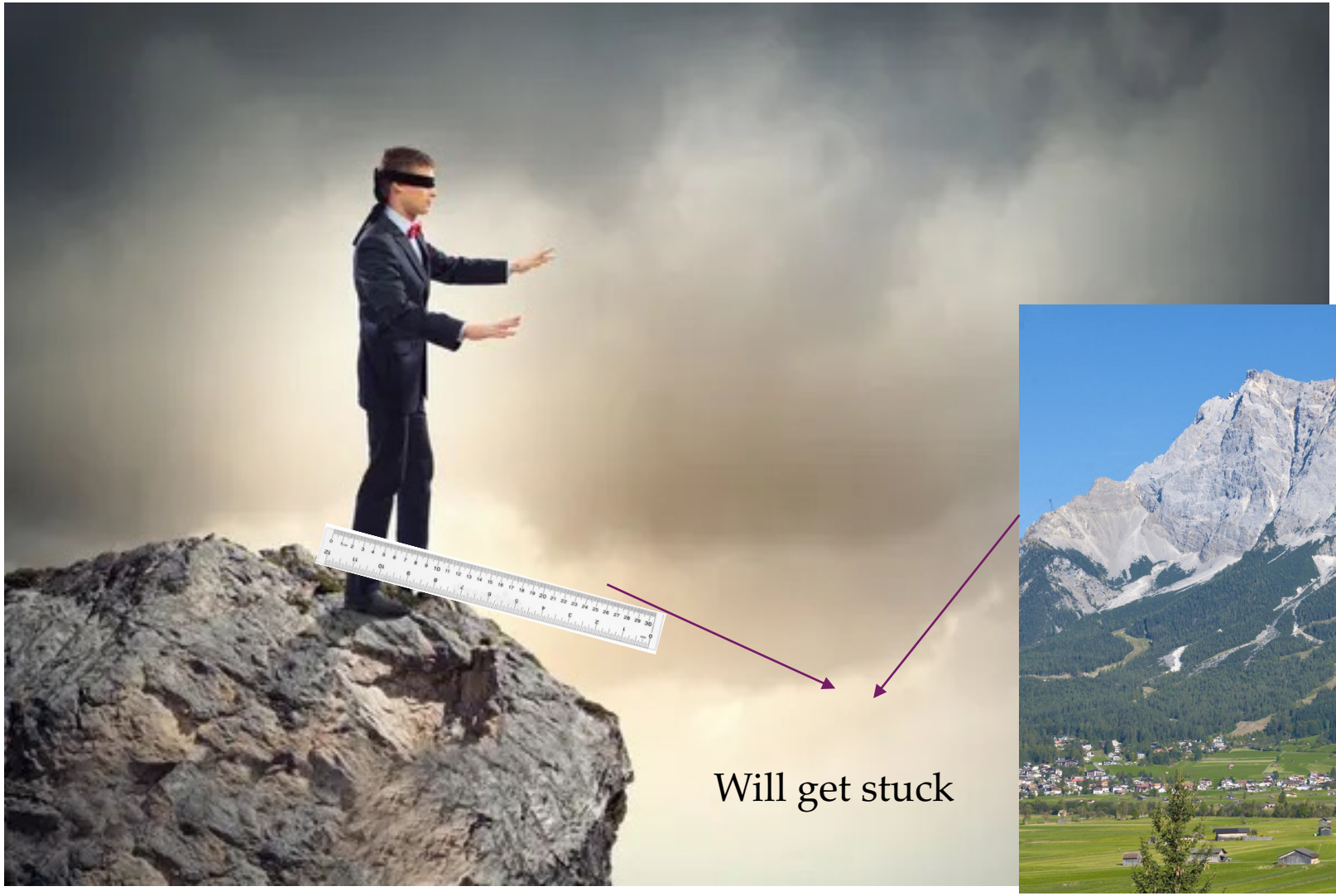
- Model identifiability
  - A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters.
  - Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other.
- Local minima can be extremely numerous
  - However, all these local minima arising from non-identifiability are equivalent to each other in cost function value.
  - *Those* local minima are not a problematic form of non-convexity.
  - The other local minima (next slides) are.

2. Local minima

- Stochasticity alone is not always enough to escape local minima
- These nice visualization below: just in our own imagination
- In practice, we (and the NNs) are blind of what the landscape really looks like
  - Our best hope is to simply get the optimization right

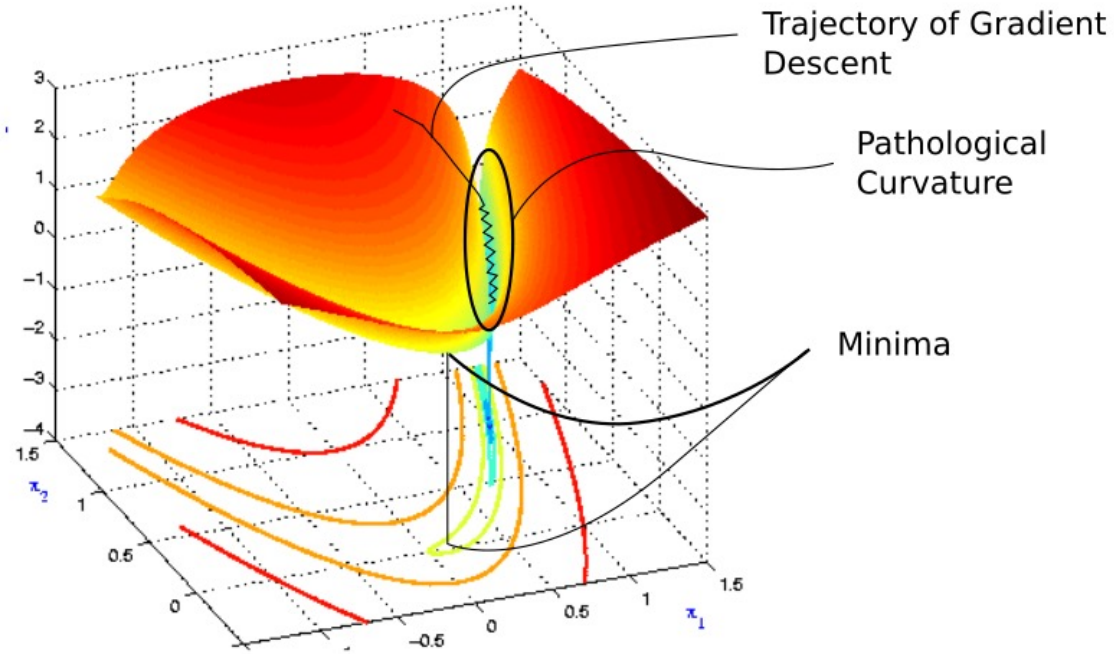


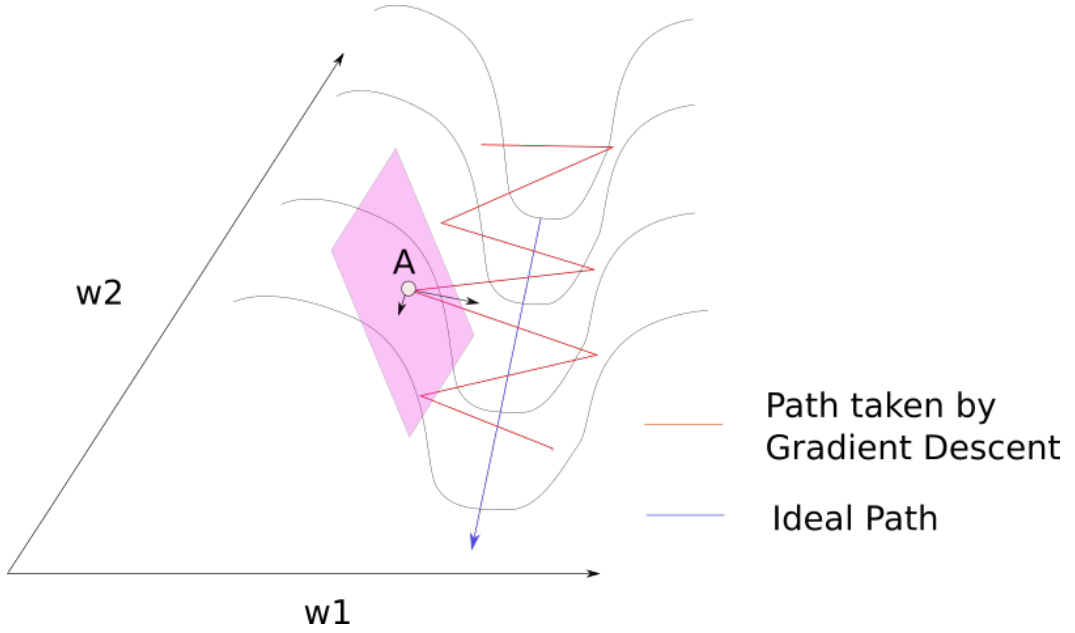
2. Local minima: tricky thing about "blindness"



3. Ravines

- Locations where the gradient is large in one direction and small in another

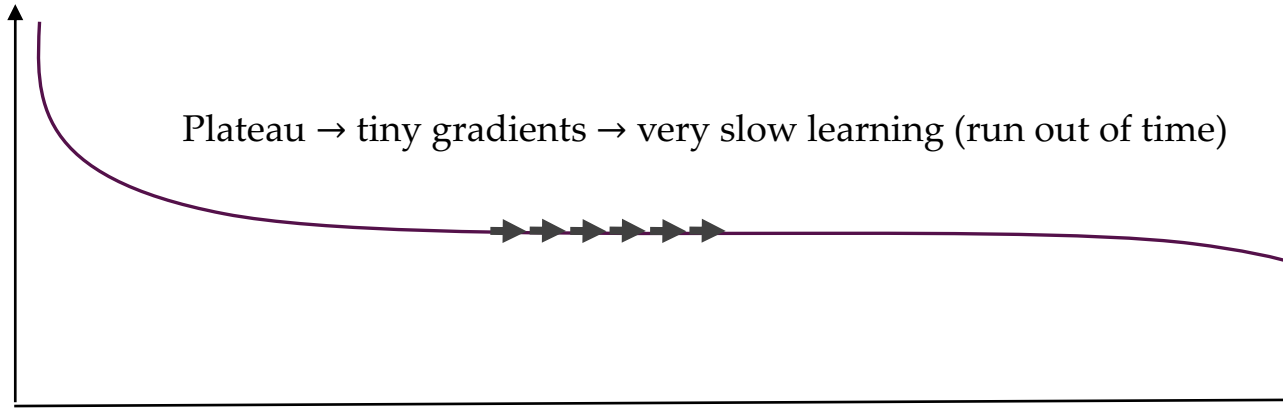




Picture credit: [Team Paperspace]

3. Plateaus/Flat areas

- In flat areas, there is almost zero-gradients → no updates → no learning
- That said, flat areas *that are minima* generalize well



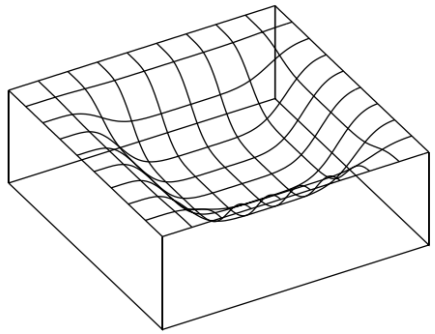


Figure 1: *Example of a "flat" minimum.*

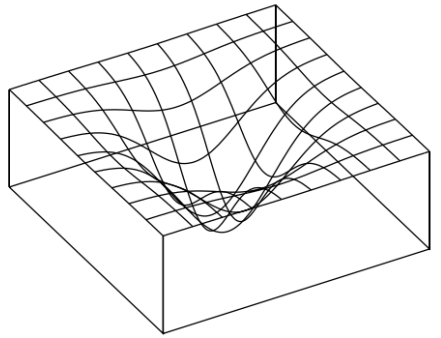


Figure 2: *Example of a "sharp" minimum.*

Link

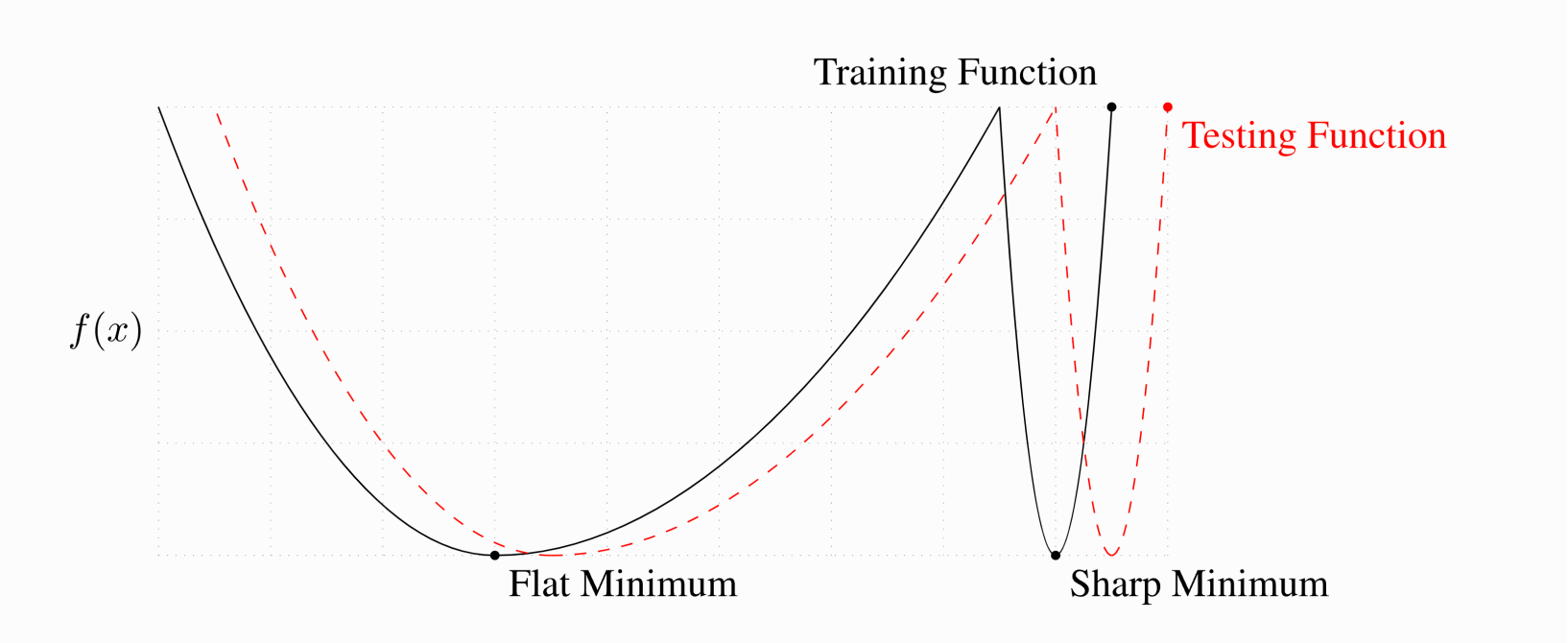
Quiz

Why might flat minima generalize better? Because..
1) They generally have lower loss values
2) Neural network weights are always only of finite precision
3) Training distribution might be slightly different to testing distribution

Why are “flat” minima preferred?

Because even if you miss the test distributions minimum slightly, you’re still in a good place.

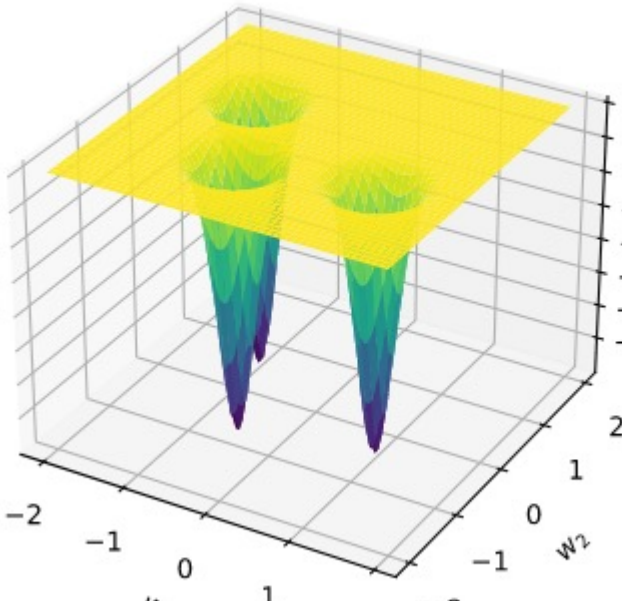
“small-batch methods converge to flat minimizers characterized by having numerous small eigenvalues of the Hessian”



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

4. Flat areas, 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?
- Toy example:
  temperate-scaled logits & cross-entropy: $p(y|x) = \text{softmax}(\text{logits}/0.00001)$

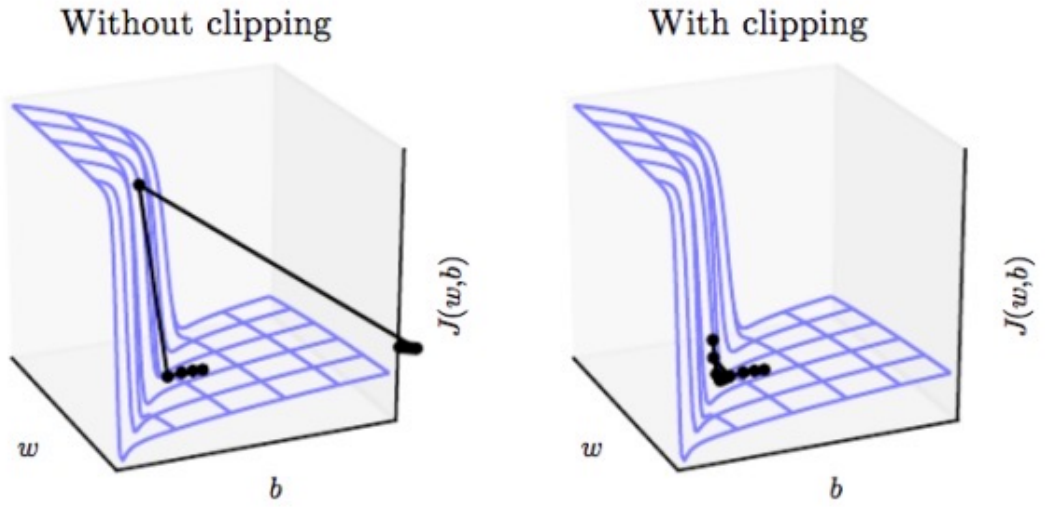


4. Cliffs and Exploding Gradients

- Cliffs
  - Neural networks with many layers often have extremely steep regions resembling cliffs.
  - These result from the multiplication of several large weights together.
- Gradient clipping
  - a simple trick

  if $|\mathbf{g}| > \eta$:

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



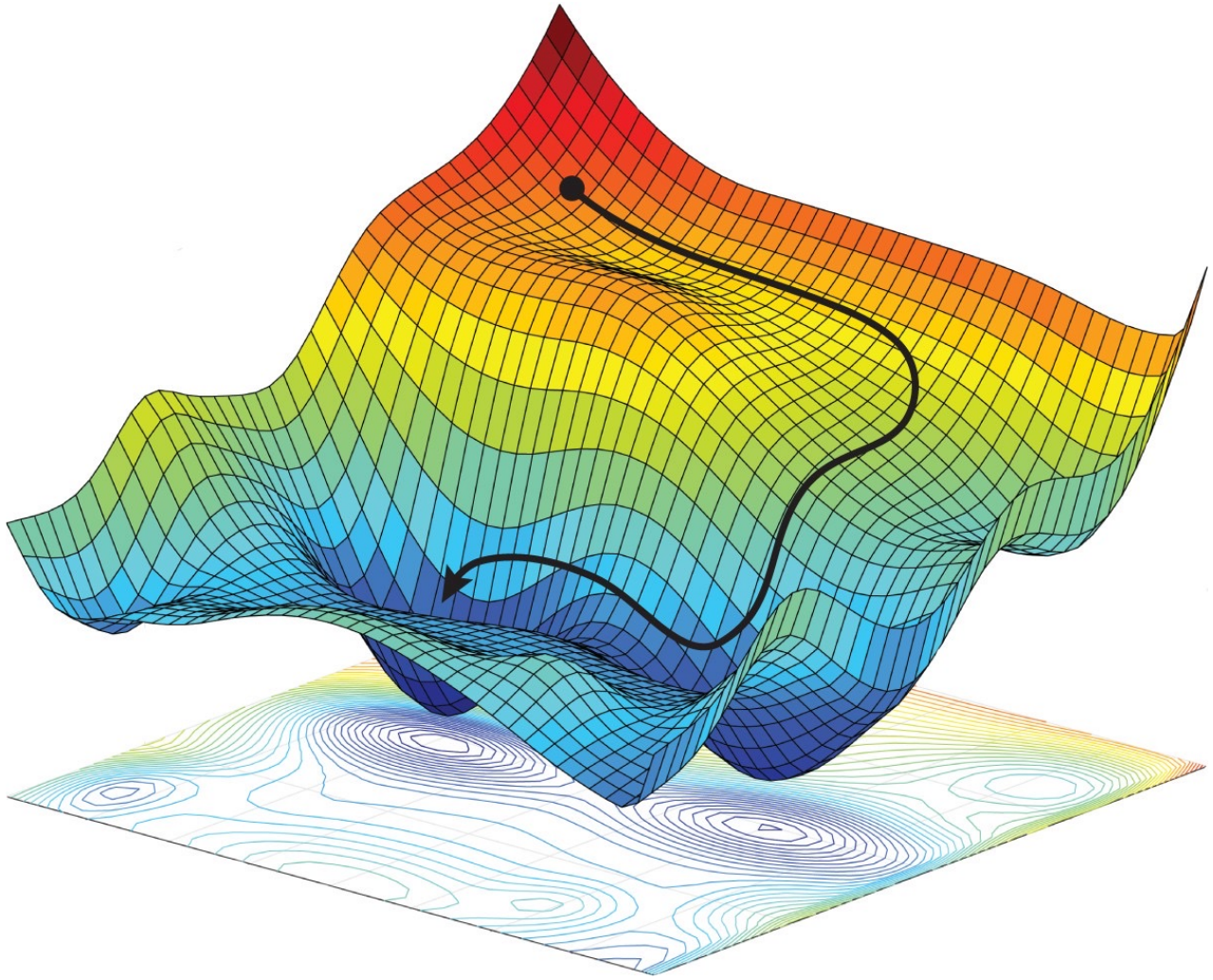
5. Long term dependencies

- Networks with many layers
  - Recurrent neural networks
  - Repeatedly applying the same operation at each time step

$$\boldsymbol{W}^t = \left(\boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}\right)^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}$$

- The vanishing and exploding gradient problem
  - gradients through are also scaled according to diag(λ)^t.
  - Vanishing gradients -> no direction to move
  - Exploding gradients -> learning unstable.
- Also training-trajectory dependency: hard to recover from a bad start!

Advanced Optimizers



Revisit of gradient descent

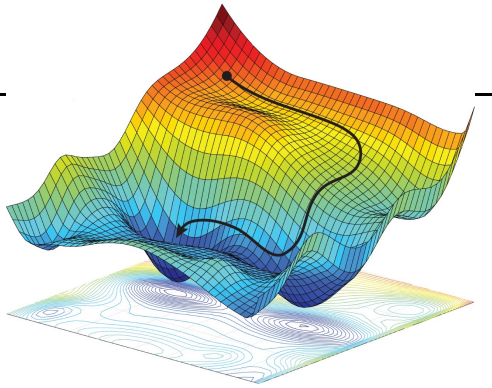
- The update rule

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

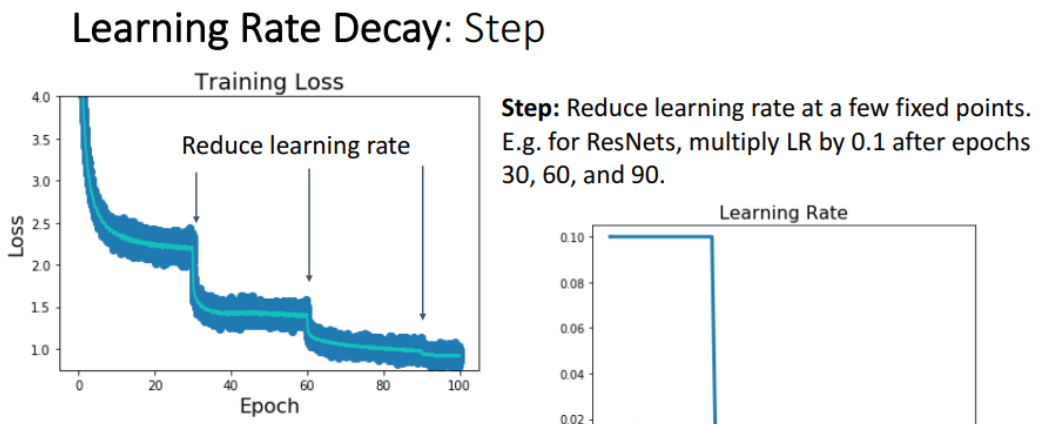
a better learning rate

a better gradient

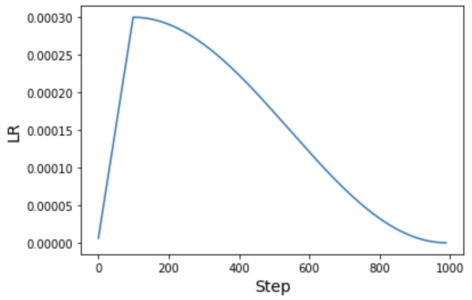
Setting the learning rate



- Go from high-> to low.
- How? "Learning rate schedules"
  - Step decay



  - Half Cosine w/ warmup



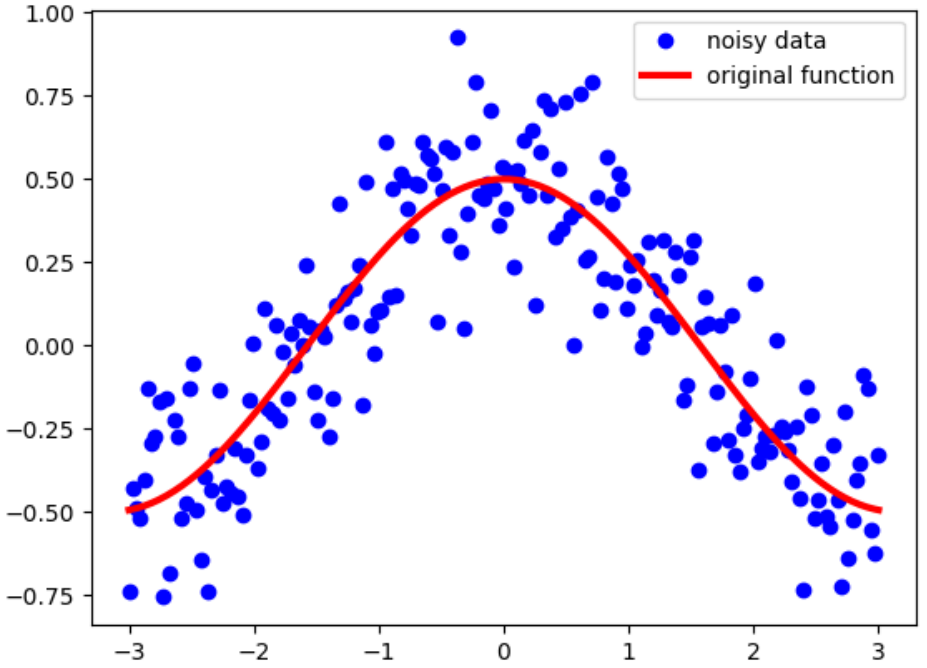
Advanced optimizers

- SGD with momentum
- Nesterov momentum
- SGD with adaptive learning rates
  - AdaGrad
  - RMSProp
  - Adam
- Second-order approximation
  - Newton's methods

Momentum.

Momentum: designed to accelerate learning, especially when loss is high curvature

- Recall: Exponentially weighted (moving) averages
  - Used for sequences of numbers.
  - Suppose, we have some sequence S which is noisy.

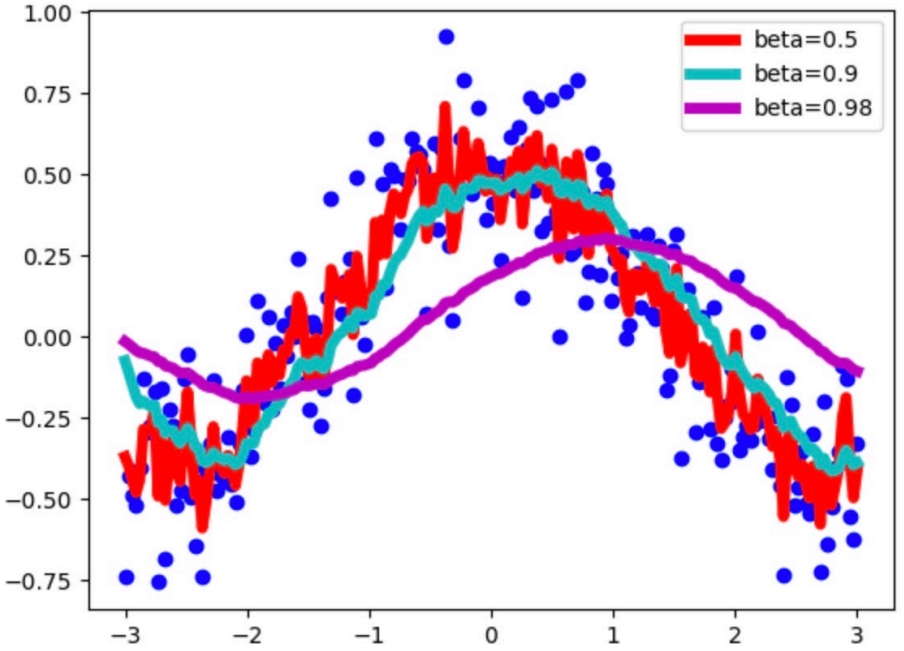


Momentum

- Exponentially weighted averages
  - define a new sequence V with the following equation:
  - $V_t = \beta V_{t-1}+(1-\beta)S_t,\ \beta \in [0\ 1], V_0=0$

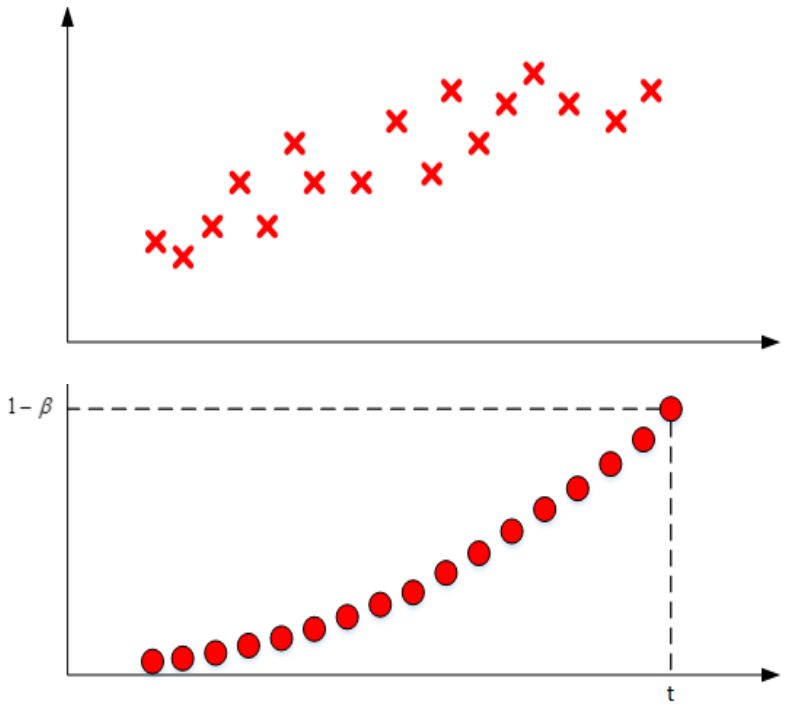
- Smoothness
  - smaller beta -> a more fluctuating curve
  - lager beta -> a smoother curve
  - β=0.9 provides a good balance

- Bias correction
  - Eg: $V_1 = \beta V_0+(1-\beta)S_1$: biased towards V_0
  - $V_t = \frac{V_t}{1-\beta^t}$



Momentum

- Exponentially weighted averages
  - Three consecutive elements of the new sequence V.
  - $V_t = \beta\ \beta V_{t-2} + \beta(1\text{-}\ \beta)S_{t-1}+(1\text{-}\ \beta)S_t$
- At some point the weight is going to be so small ($\frac{1}{e}$)
  - We 'forget' that value because its contribution becomes too small to notice.
  - E.g.,



$$\beta = 0.9,\ 0.9^{10} = 0.35 = \frac{1}{e};\ \beta = 0.98,\ \ 0.98^{50} = 0.35 = \frac{1}{e}$$

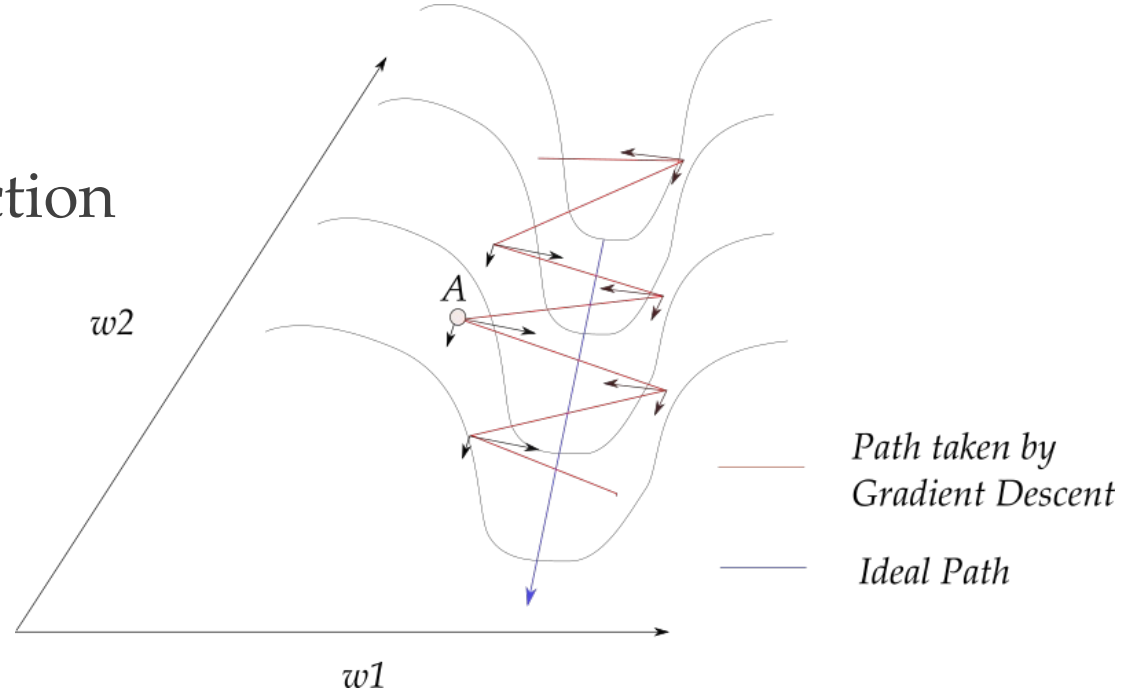
SGD with momentum

- Don't switch update direction all the time
- Maintain "*momentum*" from previous updates → dampens oscillations

$$v_{t+1} = \gamma v_t + \eta_t g_t, \qquad \eta_t = \text{LR}$$

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

- Exponential averaging keeps steady direction
- Example: $\gamma = 0.9$ and $v_0 = 0$
  - $v_1 \propto -g_1$
  - $v_2 \propto -0.9g_1 - g_2$
  - $v_3 \propto -0.81g_1 - 0.9g_2 - g_3$



SGD with 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

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

$$x_{t+1} = x_t - \alpha v_{t+1}$$

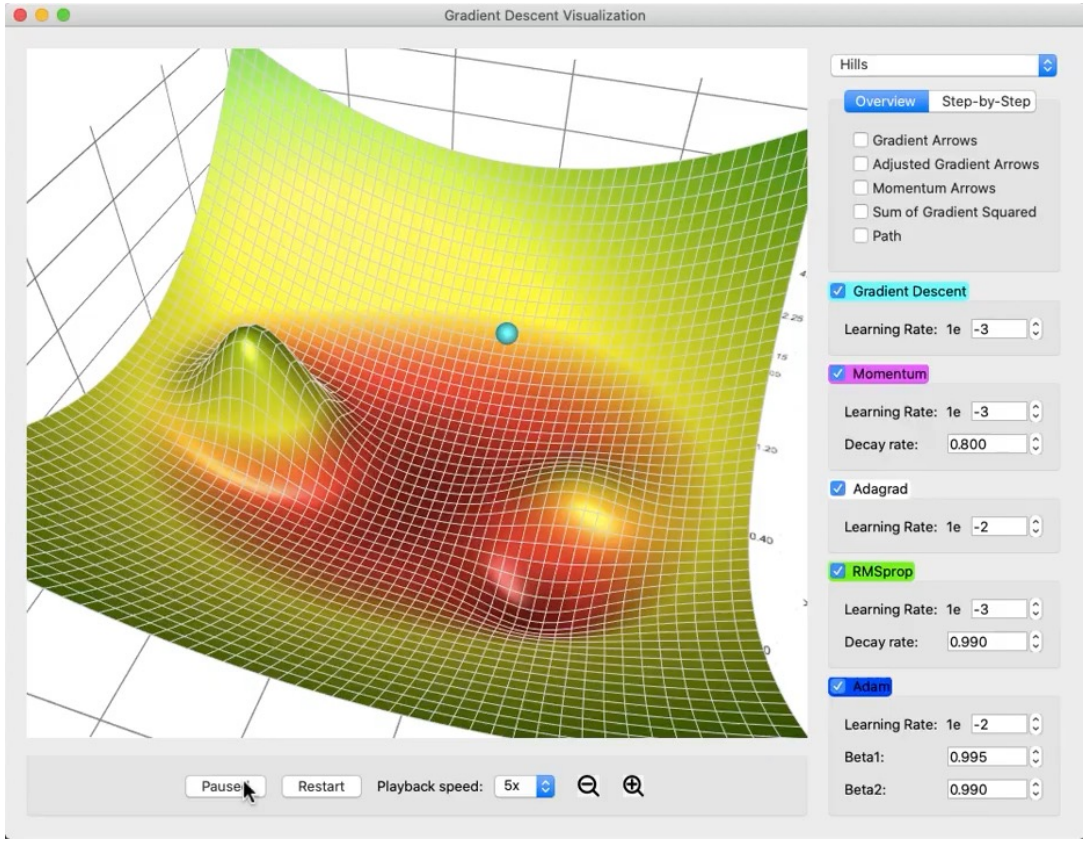
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

- 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

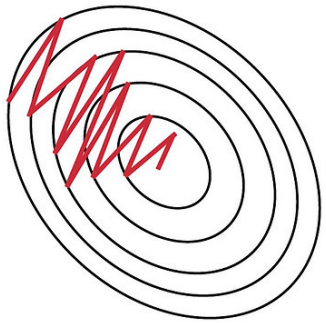
SGD with momentum

- The exponential averaging
  - cancels out the oscillating gradients
  - gives more weight to recent updates
- More robust gradients and learning
  → faster convergence

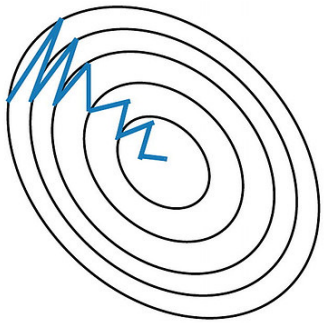


SGD with momentum

- A ball goes down a hill.
  - With momentum, the ball accumulates momentum as it rolls downhill, becoming faster on the way, such that it can overcome minor obstacles
- Parameter update
  - The momentum term increases for dimensions whose gradients point in the same directions.
  - and reduces updates for dimensions whose gradients change directions.



without momentum



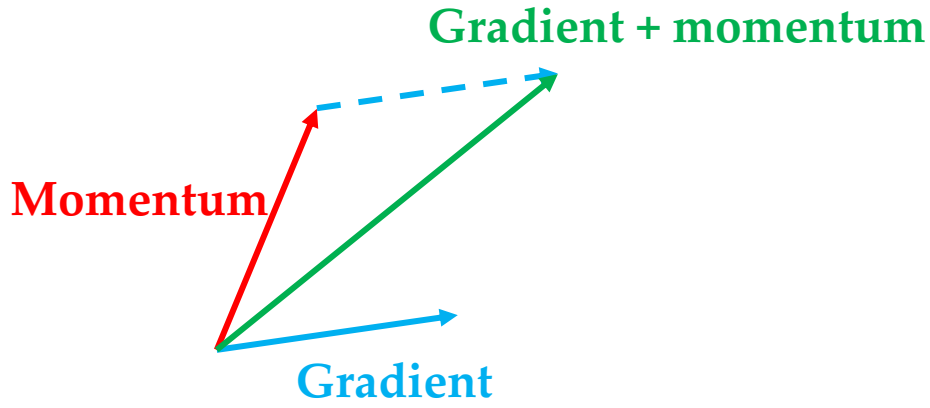
with momentum

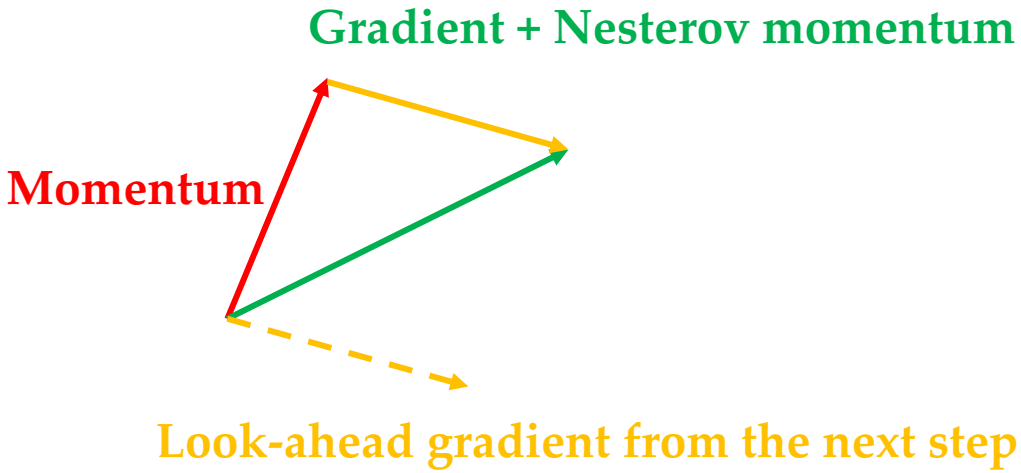
Nesterov momentum

- Use the future gradient instead of the current gradient

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

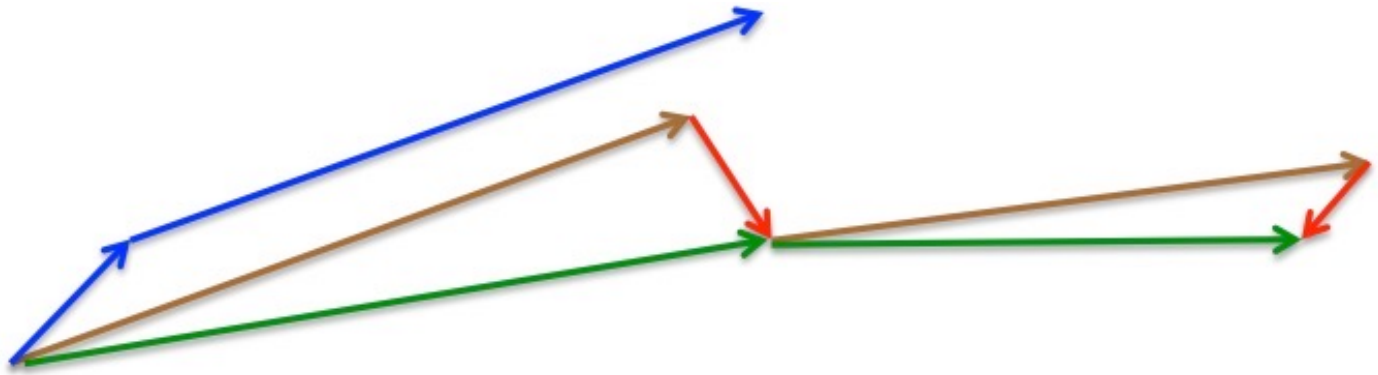
- A way to give our momentum term this kind of prescience.
- Prevents us from going too fast and results in increased responsiveness





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.

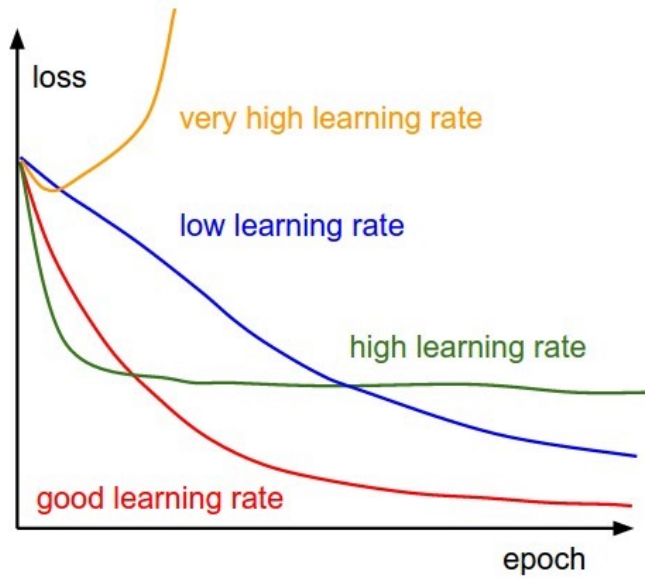


Picture from Hinton lecture

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

SGD with adaptive step sizes

- Learning rates ~ step size
  - one of the hyperparameters that is the most difficult to set
  - it has a significant impact on model performance.
  - The cost is often highly sensitive to some directions in parameter space and insensitive to others.
  - a separate, adaptive learning rate for each parameter?
- Various optimizers help:
- AdaGrad
- RMSProp
- Adam



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 $r = \sum_t (\nabla_w \mathcal{L})^2$

Adagrad

- Advantages
  - It “eliminates” the need to manually tune the learning rate
  - faster and more reliable, when the scaling of the weights is unequal
- Adadelta - an extension of 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.

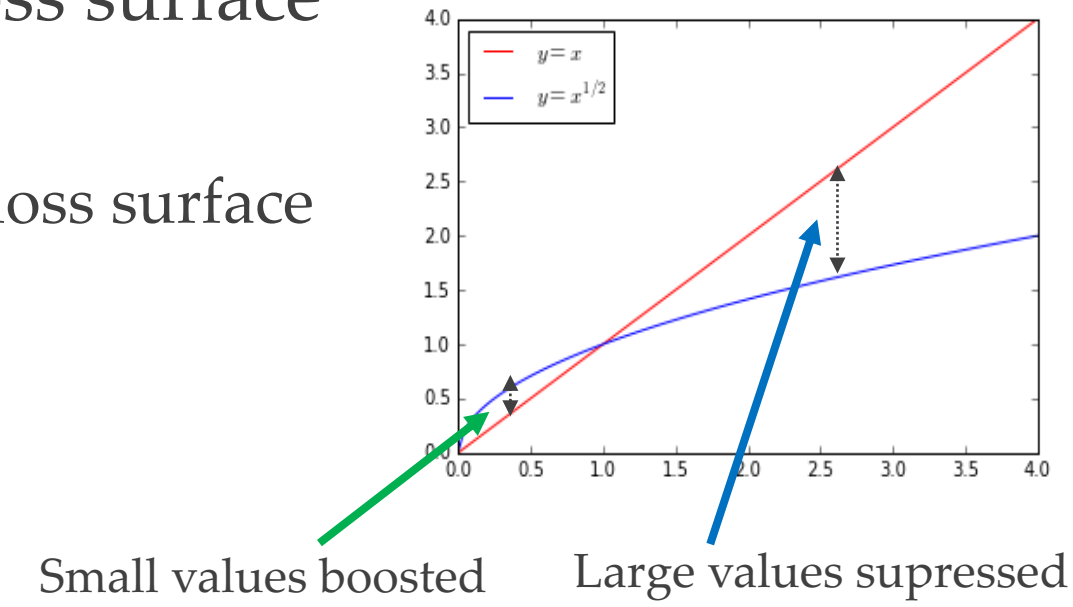
RMSprop

- Root Mean Square Propagation (RMSprop)
  - Modify AdaGrad for non-convex optimization
  - Use an exponentially weighted average to accumulate gradients.
- Further development
  - with standard momentum
  - with Nesterov momentum
  - …

RMSprop

Decay hyper-parameter (usually 0.9)

- Schedule
  - $r_t = \alpha r_{t-1} + (1-\alpha) g_t^2$
  - $v_t = \frac{\eta}{\sqrt{r_t}+\varepsilon} \odot g_t$
  - $w_{t+1} = w_t - v_t$
- **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
- Sort of performs simulated annealing



Adam

- Adaptive Moment Estimation (Adam)
  - combines RMSProp and momentum.
  - computes an adaptive learning rate for each parameter (high memory)
  - also keeps an exponentially decaying average of past gradients like momentum.
  - introduces bias corrections to the estimates of moments.
  - behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.
  - Generally best place to start, as not that hyperparameter dependent.
- One of the most popular learning algorithms
  - Especially for transformer architectures

Adam

- Schedule

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Bias corrections

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

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

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

Notice something?

Adam

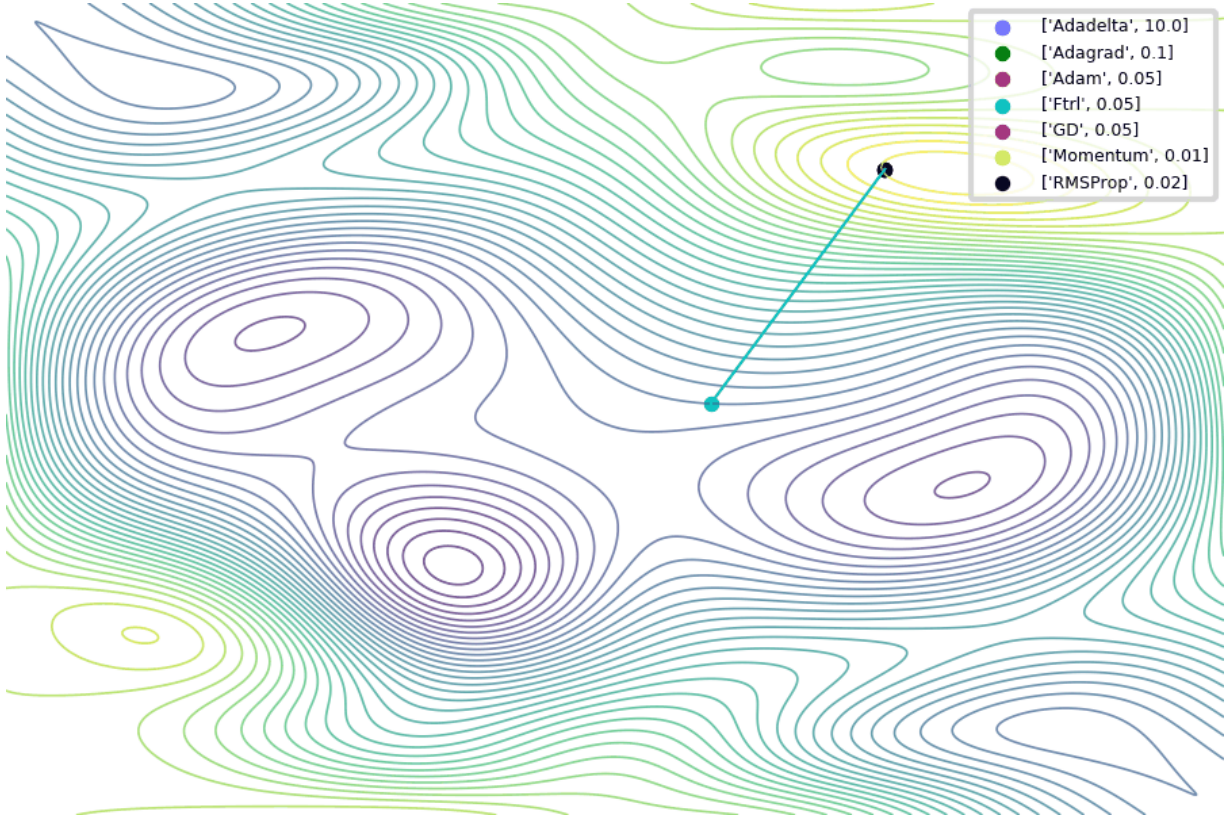
- Schedule

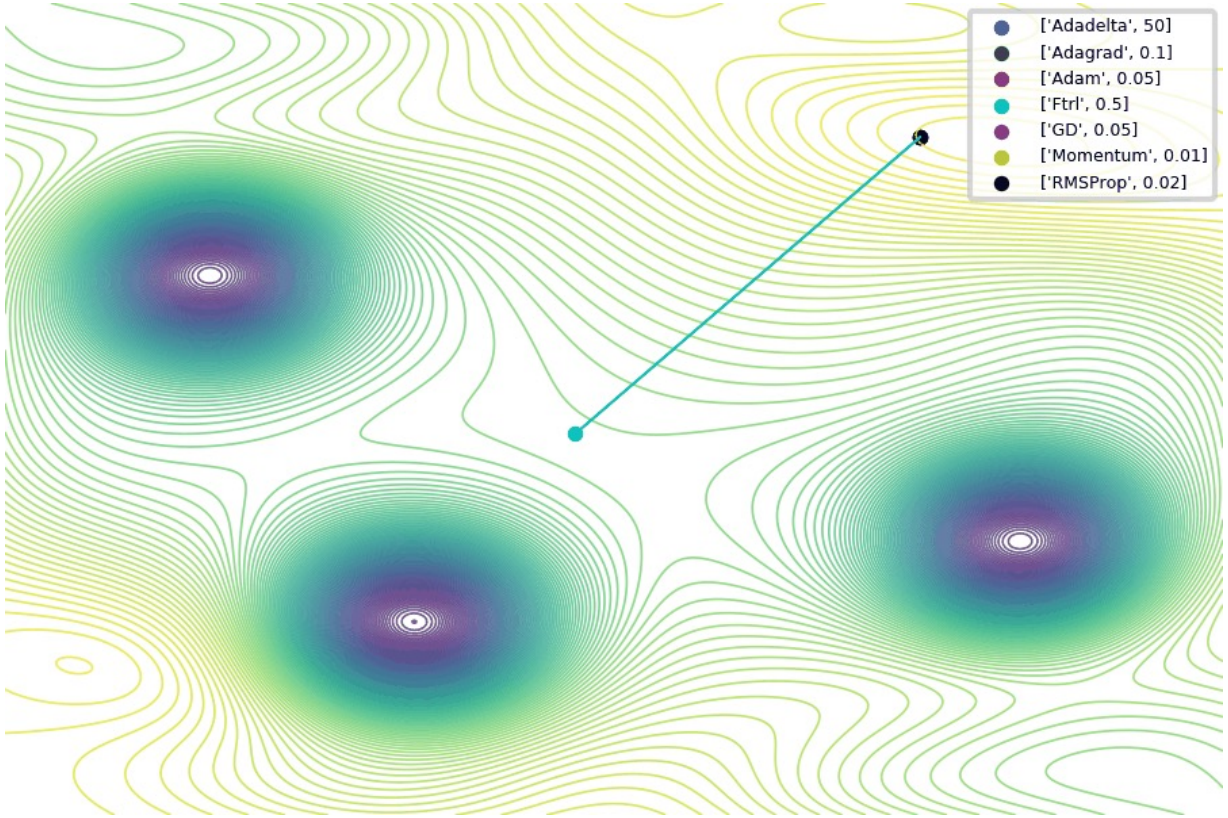
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$ Bias corrections
$$u_t = \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t$$
$$w_{t+1} = w_t - u_t$$

  - Recommended values: $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$
- Adaptive learning rate as RMSprop, but with momentum & bias correction

Still requires a learning rate!
(though much less finicky)

Visual overview



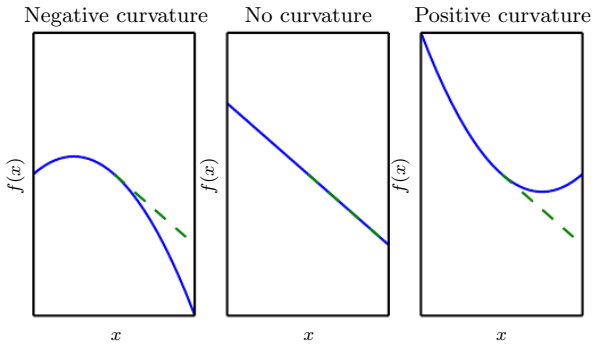


Picture credit: Jaewan Yun

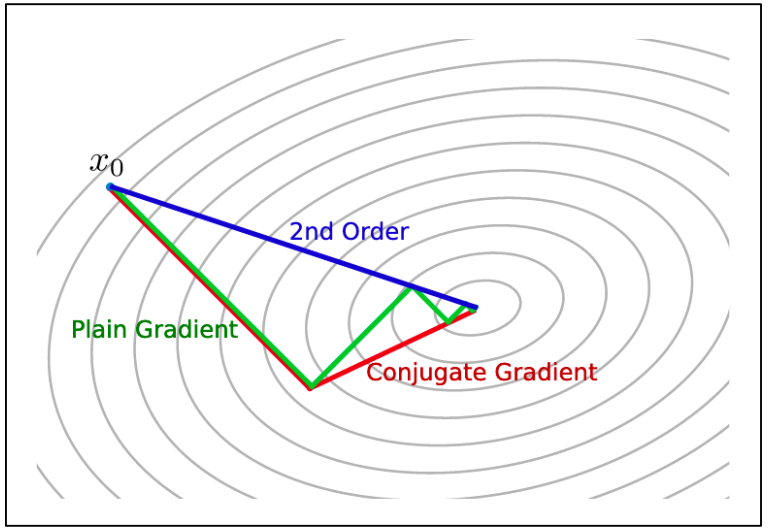
Which optimizer to use?

- Tuned SGD+momentum works often best.
- For more complex models Adam is often the preferred choice.
  - However, Adam tends to "over-optimize".
- Adam with weight-decay (AdamW) is the standard for optimizing transformer architectures (lecture 7)
- Fun fact: even in "learning rate adjusting" optimizers like Adam, we do learning rate decays.

Approximate Second-Order Methods



- Why second-order methods?
  - Better direction
  - Better step-size
  - A full step jumps directly to the minimum of the local squared approx.
  - Additional step size reduction and dampening are straight-forward
- Newton's Method
- Conjugate gradient
- Quasi-Newton
  - BFGS, L-BFGS



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:

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

- For a locally quadratic function, Newton's method jumps directly to the minimum.
- If convex but not quadratic (there are higher-order terms), *this 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

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

- Computationally expensive
  - require the inversion of the Hessian matrix
  - at every training iteration

Quasi-Newton methods

- Broyden–Fletcher–Goldfarb–Shannon (BFGS)
  - Newton's method without the computational burden
  - approximate the inversion M of the Hessian matrix.
  - iteratively refined by low-rank updates (no need to fully compute Hessian)

- Limited Memory BFGS
  - assume that $M^{(t-1)}$ is the identity matrix,
  - rather than storing the approximation from one step to the next.

- Rarely used

Interactive session

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

Reading materials

- Chapter 4 & 8, Deep Learning

How research gets done part III

Step 3 of deep learning research

[Recap: step 1&2: understand fundamentals, read papers.]

Next, start reproducing results and playing around.

- After reading an interesting paper, try to implement it from scratch.
- Compare against the open-source version of the authors/other researchers.
- Try to reproduce numbers and get familiar with tricks and hacks.

Quiz

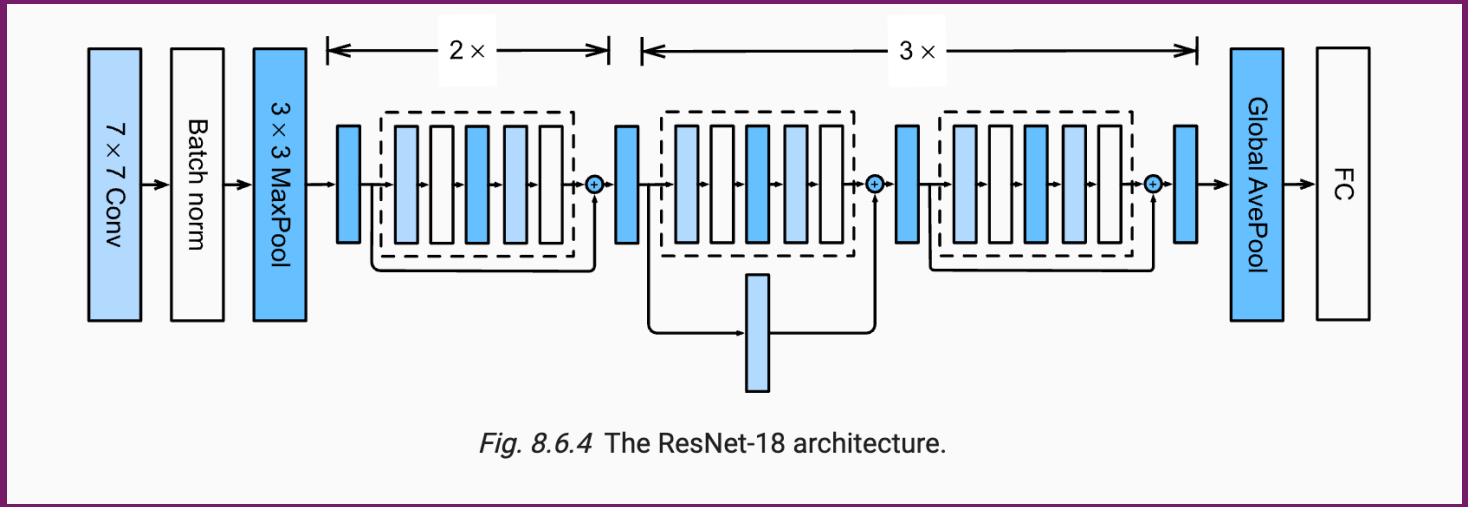
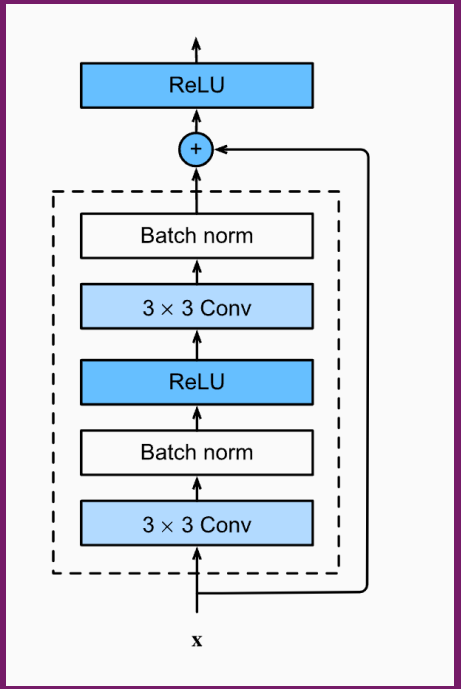


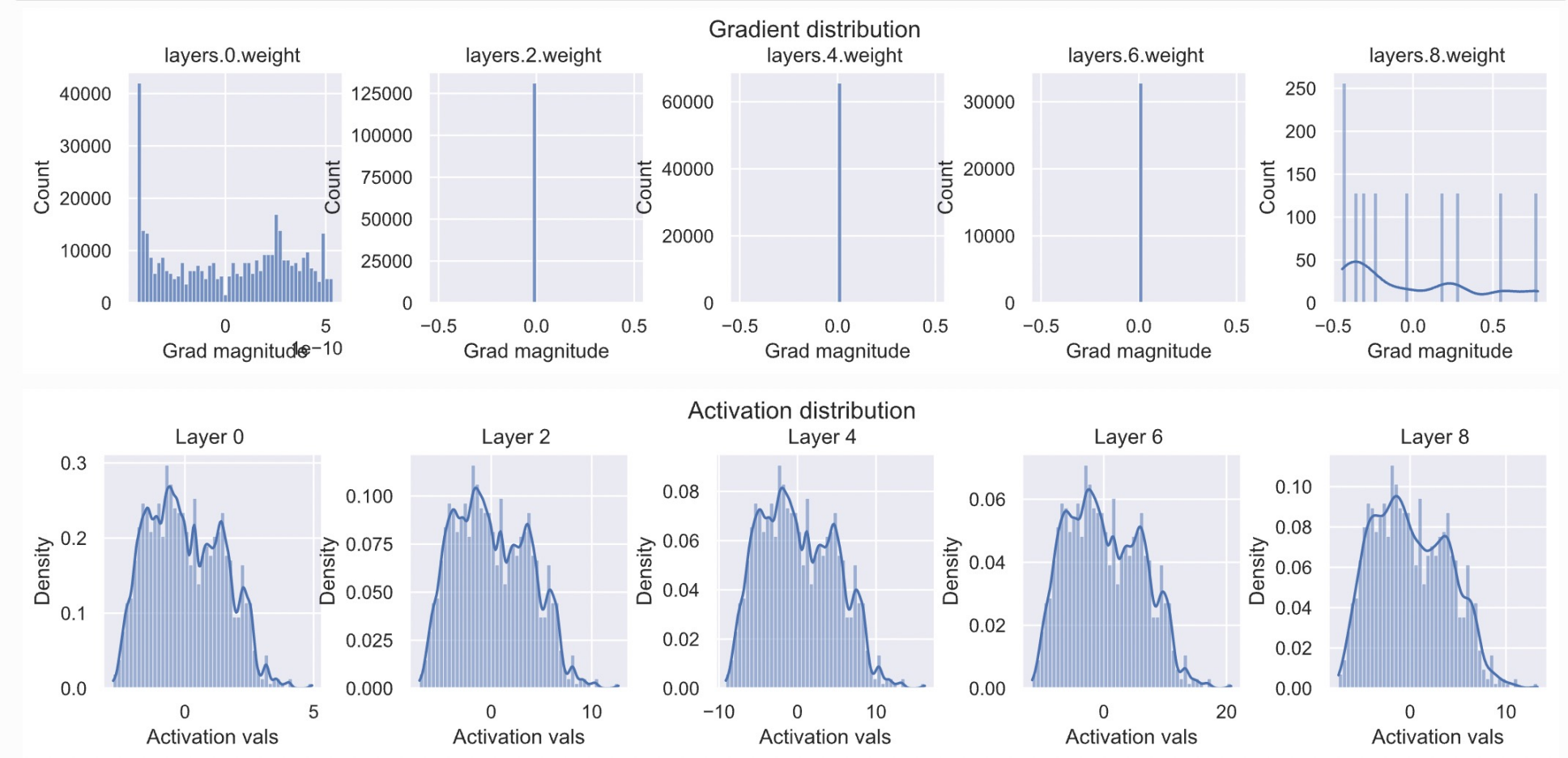
Fig. 8.6.4 The ResNet-18 architecture.



You start training with SGD.
What will not work?

1) If all weights are initialized to zeros
2) Only zeros in weights, but random biases
3) All weights set to a fixed set of random numbers
4) Setting all the weights to “42”

Re: constant init: see Tutorial next week!



Initialization

Weight initialization

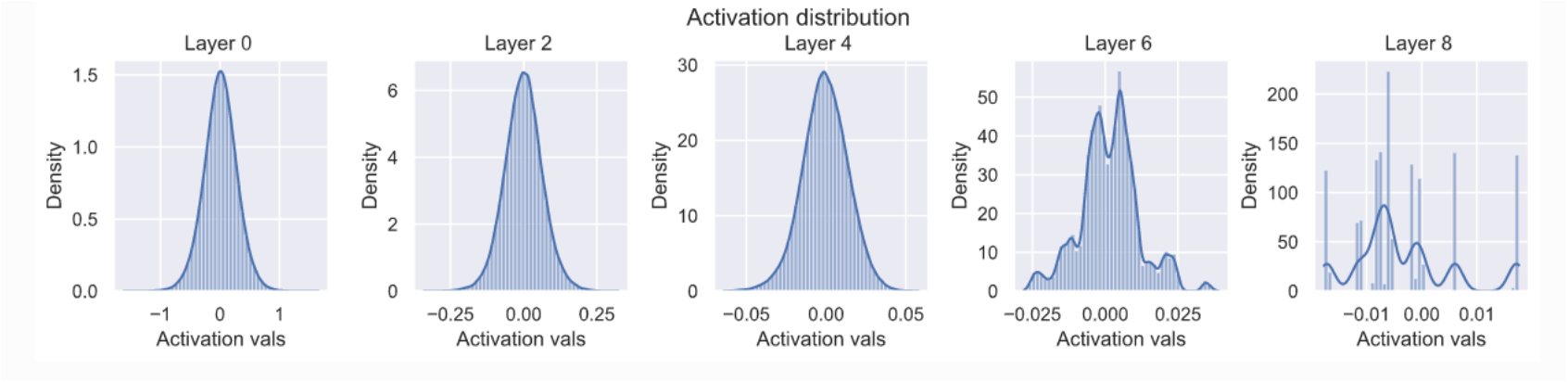
- To prevent layer activation outputs from exploding or vanishing gradients during the forward propagation.
- Initialized the weights correctly, then our objective will be achieved in the least time
- Zero Initialization
  - lead to symmetric hidden layers.
  - make your network no better than a linear model.
  - Setting biases to 0 will not create any problems
- **Random Initialization**
  - breaks the symmetry
  - prevents neurons from learning the same features

Random: yes. But how?

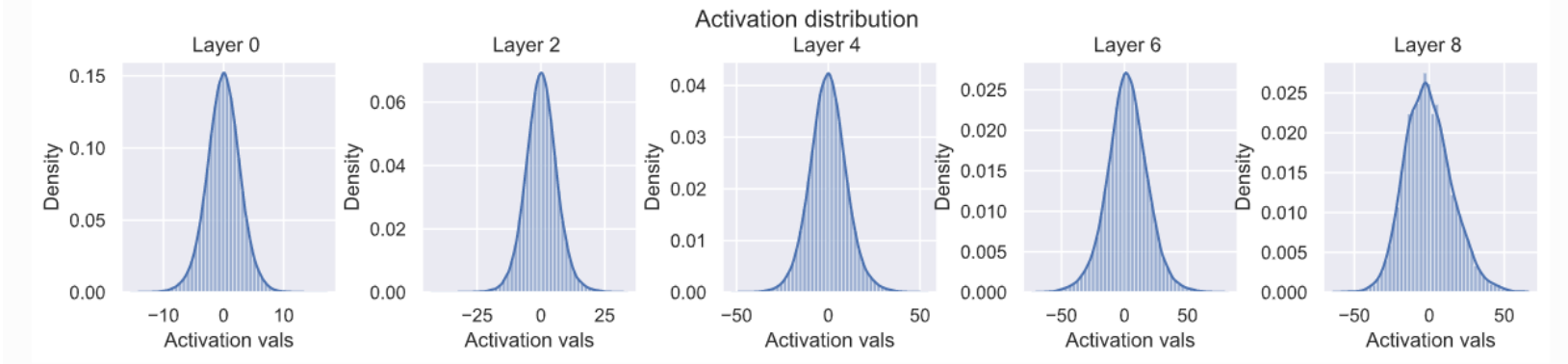
- Weights initialized **to preserve the variance** of the activations
  - During the forward and backward computations
  - We want similar input and output variance because of modularity
- Weights must be initialized to be different from one another
  - Don't give same values to all weights (like all 0)
  - All neurons (in one layer) generate same gradient → no learning
- Initialization depends on
  - non-linearities
  - data normalization

Bad initialization can cause problems

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



Initializing weights by preserving variance

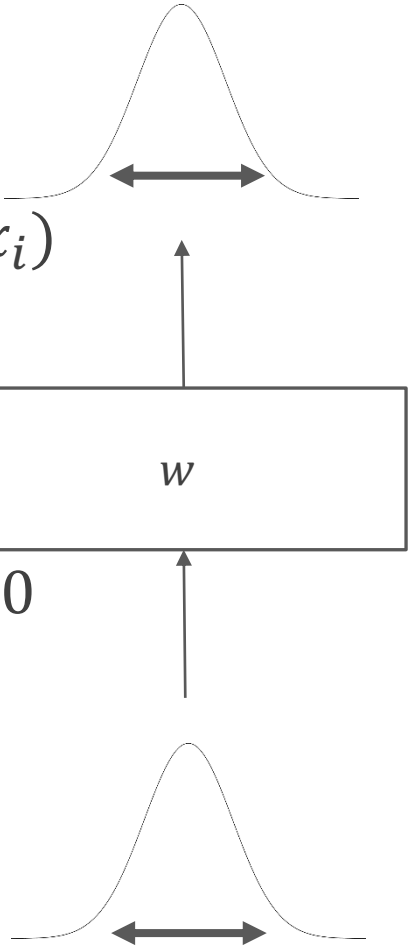
- For x and y independent
  - $var(xy) = \mathbb{E}[x]^2 var(y) + \mathbb{E}[y]^2 var(x) + var(x)var(y)$
- For $a = wx \Rightarrow var(a) = var(\sum_i w_i x_i) = \sum_i var(w_i x_i) \approx d \cdot var(w_i x_i)$

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

$$= var(x_i)var(w_i)$$

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 $var(a) \approx d \cdot var(x_i)var(w_i)$



Understanding the difficulty of training deep feedforward neural networks, Glorot, Bengio, 2010

Initializing weights by preserving variance

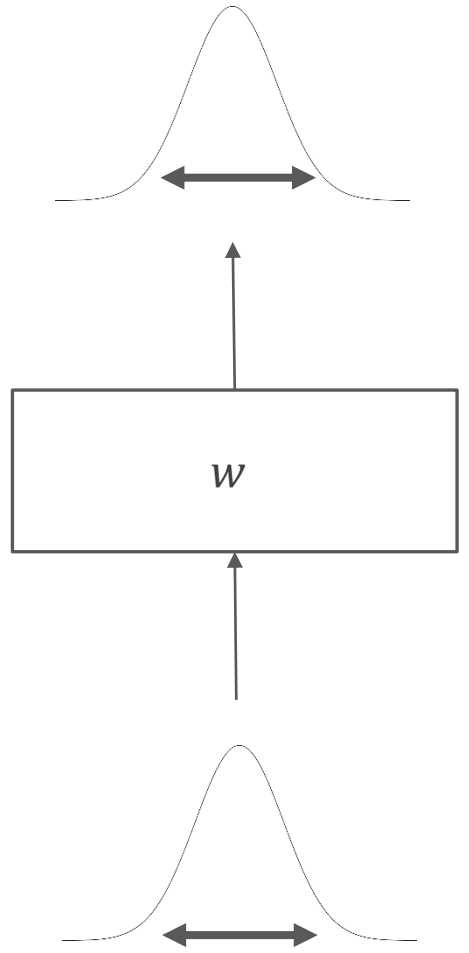
- Since we want the same input and output variance

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

- Draw random weights from

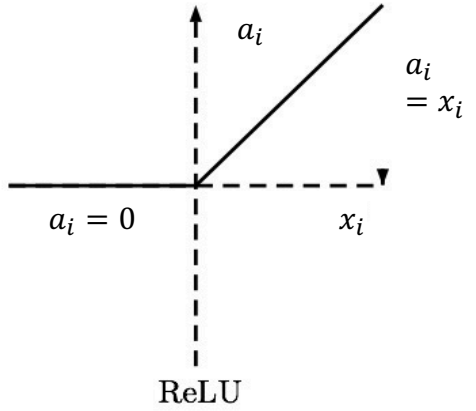
$$w \sim N(0, 1/d)$$

where d is the number of input variables to the layer



Understanding the difficulty of training deep feedforward neural networks, Glorot, Bengio, 2010

Initialization for ReLUs



- Unlike sigmoidals, ReLUs return 0 half of the time
  - $\mathbb{E}[w_i] = 0$ but $\mathbb{E}[x_i] \neq 0$
- Redoing the computations

$$var(w_i x_i) = var(w_i)(\mathbb{E}[x_i]^2 + var(x_i)) = var(w_i)\mathbb{E}\left[x_i^2\right] \; (var(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2)$$

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

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

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

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, He, Zhang, Ren, Sun, 2015 Link

Xavier initialization

- For tanh: initialize weights from $\mathrm{U}\left[-\sqrt{\frac{6}{d_{l-1}+d_l}}, \sqrt{\frac{6}{d_{l-1}+d_l}}\right]$
  - 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 $\mathrm{U}\left[-4 \cdot \sqrt{\frac{6}{d_{l-1}+d_l}}, 4 \cdot \sqrt{\frac{6}{d_{l-1}+d_l}}\right]$

Understanding the difficulty of training deep feedforward neural networks, Glorot, Bengio, 2010

Interesting results with randomly initialized networks

- Hidden in a **randomly weighted Wide ResNet-50** [32] we find a subnetwork (with random weights) **that is smaller than, but matches the performance of a ResNet-34 [9] trained on ImageNet** [4]. Not only do these "untrained subnetworks" exist, but we provide an algorithm to effectively find them.
- Random matrices are essentially full-rank (c.f. Johnson-Lindenstrauss Lemma)
- (Currently one of my MSc students working on generalising this)

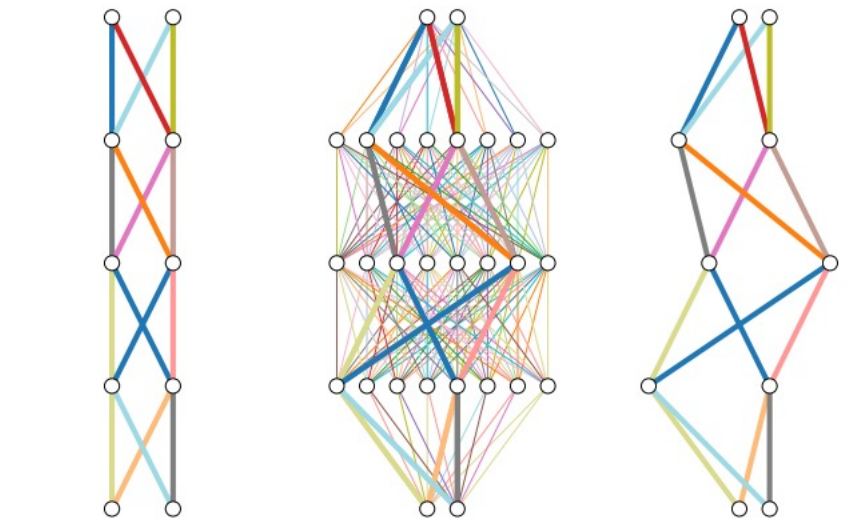


Figure 1. If a neural network with random weights (center) is sufficiently overparameterized, it will contain a subnetwork (right) that perform as well as a trained neural network (left) with the same number of parameters.

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

Reading materials

- Chapter 8, Deep Learning
- Chapter 6, (8) UDL book
- Optional
  - Recommendation: Gradient descent, how neural networks learn | Chapter 2, Deep learning by 3Blue1Brown
  - Paper: Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning.