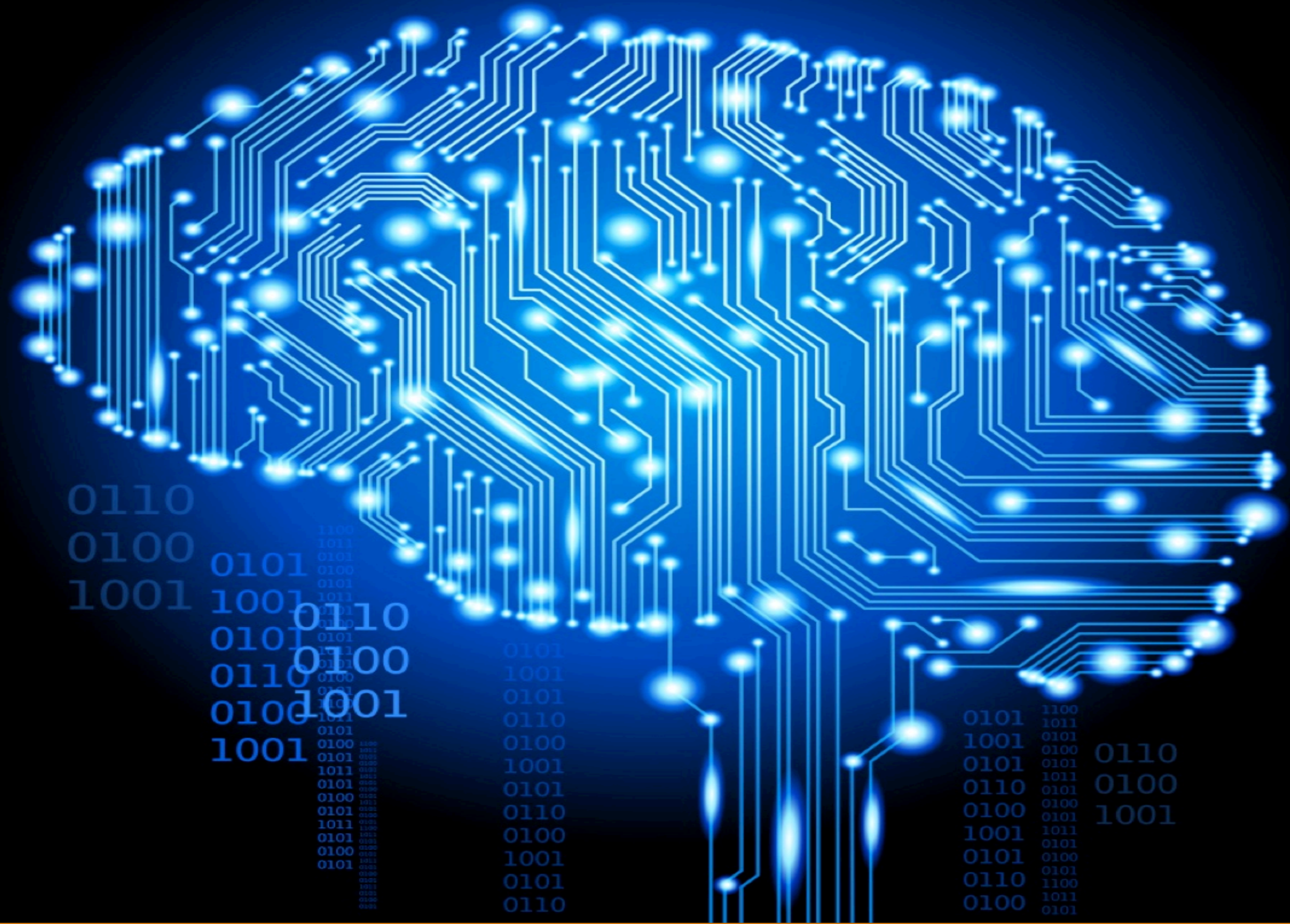


Lecture 3: Deep Learning Optimizations

Deep Learning @ UvA

Lecture overview

- How to define our model and optimize it in practice
- Optimization methods
- Data preprocessing and normalization
- Regularizations
- Learning rate
- Weight initializations
- Good practices

Empirical Risk Minimization

A Neural/Deep Network in a nutshell

1. The Neural Network

$$\boxed{a_L(x; w_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x, \mathrm{w}_1), \mathrm{w}_{L-1}), \mathrm{w}_L)}$$

2. Learning by minimizing empirical error

$$\boxed{\mathrm{w}^* \leftarrow \arg\min_w \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; w_{1,\ldots,L}))}$$

3. Optimizing with Stochastic Gradient Descent based methods

$$\boxed{w_{t+1} = w_t - \eta_t \nabla_w \mathcal{L}}$$

What is a difference between Optimization and Machine Learning?

- The optimal machine learning solution is not necessarily the optimal solution
- They are practically equivalent
- Machine learning relates to optimization, with some differences
- In learning we usually do not optimize the intended task but an easier surrogate one
- Optimization is offline while Machine Learning can be online

What is a difference between Optimization and Machine Learning?

- The optimal machine learning solution is not necessarily the optimal solution
- They are practically equivalent
- Machine learning relates to optimization, with some differences
- In learning we usually do not optimize the intended task but an easier surrogate one
- Optimization is offline while Machine Learning can be online

What is a difference between Optimization and Machine Learning?

- The optimal machine learning solution is not necessarily the optimal solution
- They are practically equivalent
- Machine learning relates to optimization, with some differences
- In learning we usually do not optimize the intended task but an easier surrogate one
- Optimization is offline while Machine Learning can be online

What is a difference between Optimization and Machine Learning?

- The optimal machine learning solution is not necessarily the optimal solution
- They are practically equivalent
- Machine learning relates to optimization, with some differences
- In learning we usually do not optimize the intended task but an easier surrogate one
- Optimization is offline while Machine Learning can be online

What is a difference between Optimization and Machine Learning?

- The optimal machine learning solution is not necessarily the optimal solution
- They are practically equivalent
- Machine learning relates to optimization, with some differences
- In learning we usually do not optimize the intended task but an easier surrogate one
- Optimization is offline while Machine Learning can be online

What is a difference between Optimization and Machine Learning?

- The optimal machine learning solution is not necessarily the optimal solution
- They are practically equivalent
- Machine learning relates to optimization, with some differences
- In learning we usually do not optimize the intended task but an easier surrogate one
- Optimization is offline while Machine Learning can be online

Pure Optimization vs Machine Learning Training?

- Pure optimization has a very direct goal: finding the optimum
  - Step 1: Formulate your problem mathematically as best as possible
  - Step 2: Find the optimum solution as best as possible
  - E.g., optimizing the railroad network in the Netherlands
    - Goal: find optimal combination of train schedules, train availability, etc

- In Machine Learning, instead, the real goal and the trainable goal are quite often different (but related)
  - Even "optimal" parameters are not necessarily **optimal** $\leftarrow$ **Overfitting …**
  - E.g., You want to recognize cars from bikes (<u>0-1 problem</u>) in <u>unknown</u> images, but you optimize the classification log probabilities (<u>continuous</u>) in <u>known</u> images

Empirical Risk Minimization

- We ideally should optimize for

$$\min_{\mathrm{w}} \mathrm{E}_{x,\mathrm{y}\sim p_{\mathrm{data}}}[\mathcal{L}(\mathrm{w};\mathrm{x},\mathrm{y})]$$

  i.e. the expected loss under the true underlying distribution

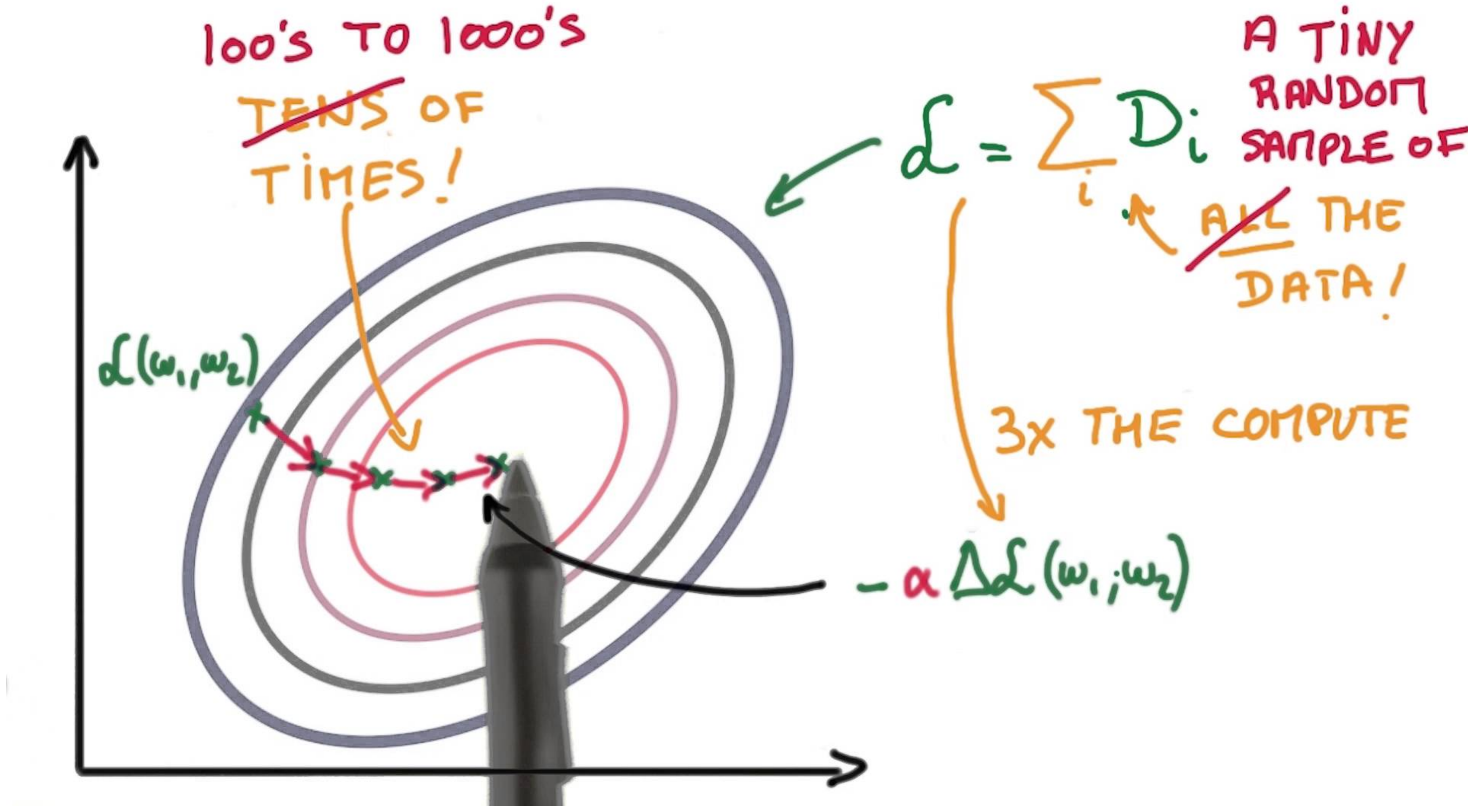
  but we do not have access to this distribution

- Thus, borrowing from optimization, the best way we can get satisfactory solutions is by minimizing the empirical risk

$$\min_{w} \mathrm{E}_{x,\mathrm{y}\sim \hat{p}_{\mathrm{data}}}[\mathcal{L}(\mathrm{w};\mathrm{x},\mathrm{y})] = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(h(x_i;w),y_i)$$

  - That is, minimize the risk on the available training data sampled by the empirical data distribution (mini-batches)
  - While making sure your parameters do not overfit the data

Stochastic Gradient Descent (SGD)


100's TO 1000's
~~TENS~~ OF TIMES!
$\mathcal{L} = \sum_i D_i$
A TINY RANDOM SAMPLE OF ~~ALL~~ THE DATA!
$\mathcal{L}(w_1,w_2)$
3x THE COMPUTE
$-\alpha \Delta \mathcal{L}(w_1;w_2)$

Gradient Descent

- To optimize a given loss function, most machine learning methods rely on Gradient Descent and variants

$$w_{t+1} = w_t - \eta_t g_t$$

  - Gradient $g_t = \nabla_t \mathcal{L}$

- Gradient on full training set → Batch Gradient Descent

$$g_t = \frac{1}{m} \sum_{i=1}^{m} \nabla_w \mathcal{L}\left(w; x_i, y_i\right)$$

  - Computed empirically from all available training samples (x_i, y_i)
  - Sample gradient → Only an approximation to the true gradient g_t^* if we knew the real data distribution

Advantages of Batch Gradient Descent batch learning

- Conditions of convergence well understood
  - Simpler theoretical analysis on weight dynamics and convergence rates
- Acceleration techniques can be applied
  - Second order (Hessian based) optimizations are possible
    - Measuring not only gradients, but also curvatures of the loss surface

Disadvantages of Batch Gradient Descent?

- Data is often too large to compute the full gradient, so slow training
- The loss surface is highly non-convex, so cannot compute the real gradient
- No real guarantee that leads to a good optimum
- No real guarantee that it will converge faster

Disadvantages of Batch Gradient Descent?

- Data is often too large to compute the full gradient, so slow training
- The loss surface is highly non-convex, so cannot compute the real gradient
- No real guarantee that leads to a good optimum
- No real guarantee that it will converge faster

Disadvantages of Batch Gradient Descent?

- Data is often too large to compute the full gradient, so slow training
- The loss surface is highly non-convex, so cannot compute the real gradient
- No real guarantee that leads to a good optimum
- No real guarantee that it will converge faster

Disadvantages of Batch Gradient Descent?

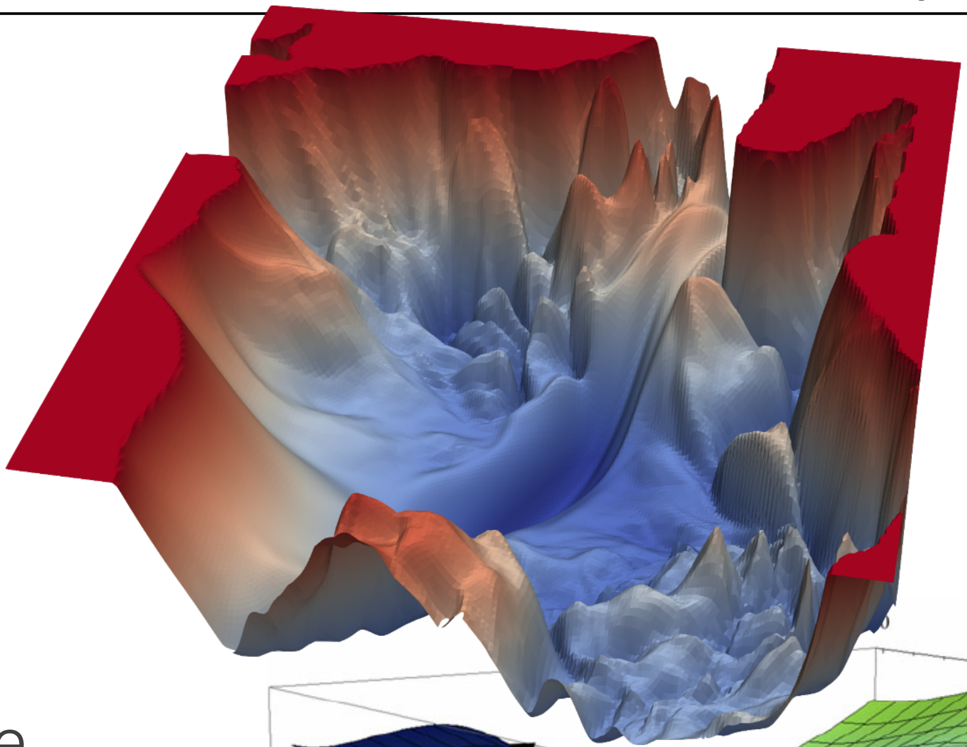
- Data is often too large to compute the full gradient, so slow training
- The loss surface is highly non-convex, so cannot compute the real gradient
- No real guarantee that leads to a good optimum
- No real guarantee that it will converge faster

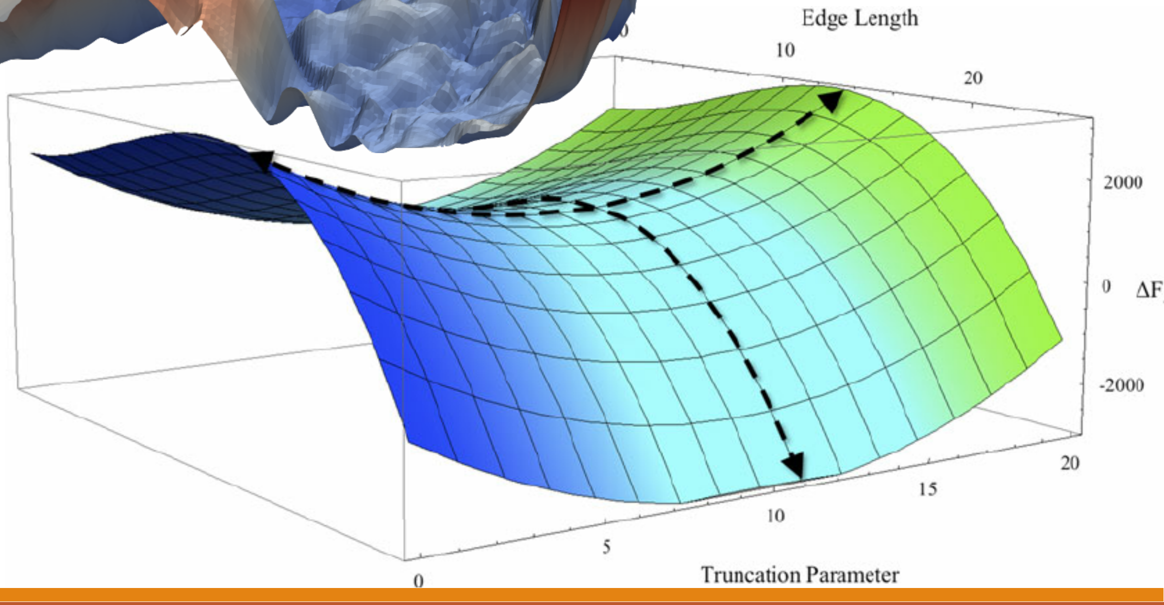
Disadvantages of Batch Gradient Descent?

- Data is often too large to compute the full gradient, so slow training
- The loss surface is highly non-convex, so cannot compute the real gradient
- No real guarantee that leads to a good optimum
- No real guarantee that it will converge faster

Still, optimizing with Gradient Descent is not perfect

- Often loss surfaces are
  - highly non-convex
  - very high-dimensional
- No real guarantee that
  - the final solution will be good
  - we converge fast to final solution
- Datasets are typically too large to compute complete gradients





Gradient Descent

- The gradient approximates the expectation $\mathrm{E}(\nabla_\theta \mathcal{L})$ by taking samples

$$\mathrm{E}(\nabla_\theta \mathcal{L}) \approx {}^{1}/_{m} \sum \nabla_\theta \mathcal{L}_i$$

  - So called Monte Carlo approximation

- Following the central limit theorem, the standard error of this first approximation is given by ${}^{\sigma}/_{\sqrt{m}}$
  - So, the error drops sublinearly with m. To compute <u>2x more</u> accurate gradients, we need <u>4x data </u>points
  - And what's the point anyways, since our loss function is only a surrogate?

Stochastic Gradient Descent (SGD)

- Introduce a second approximation in computing the gradients → SGD
  - Stochastically sample "mini-training" sets ("mini-batches") from dataset D

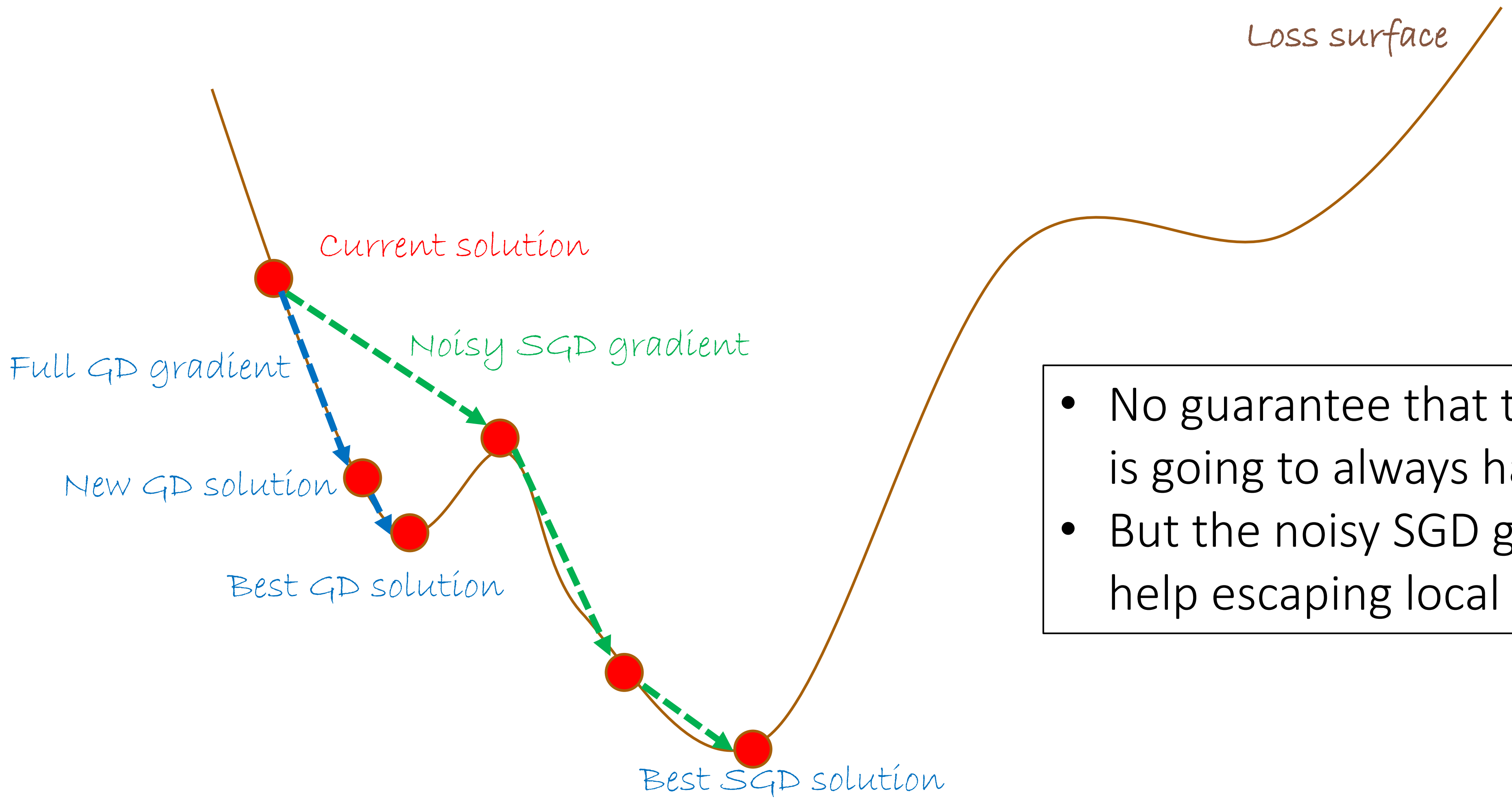
$$B_j = sample(D)$$

$$w_{t+1} = w_t - \frac{\eta_t}{|B_j|} \sum_{i \in B_j} \nabla_w \mathcal{L}_i$$

Some advantages of SGD

- Randomness helps avoid overfitting solutions
  - Variance of gradients increases when batch size decreases
- In practice, accuracy is often better
- Much faster than Gradient Descent
- Suitable for datasets that change over time

SGD is often better



- No guarantee that this is what is going to always happen.
- But the noisy SGD gradients can help escaping local optima

SGD helps avoid overfitting

- Gradient Descent: Complete gradients fit optimally the (arbitrary) data we have, not necessarily the distribution that generates them
  - All training samples are the "absolute representative" of the input distribution
  - Suitable for traditional optimization problems: "find optimal route"
  - But for ML we cannot make this assumption → test data are always different
- SGB: sampled mini-batches produce roughly representative gradients
  - Model does not overfit (as much) to the particular training samples

SGD for dynamically changing datasets

- Often data distribution changes over time, e.g. Instagram
  - Should “cool 2010 pictures” have as much influence as 2018?
- GD is biased towards the more “past” samples
- A properly implemented SGD can track changes better [LeCun2002]

Popular last year

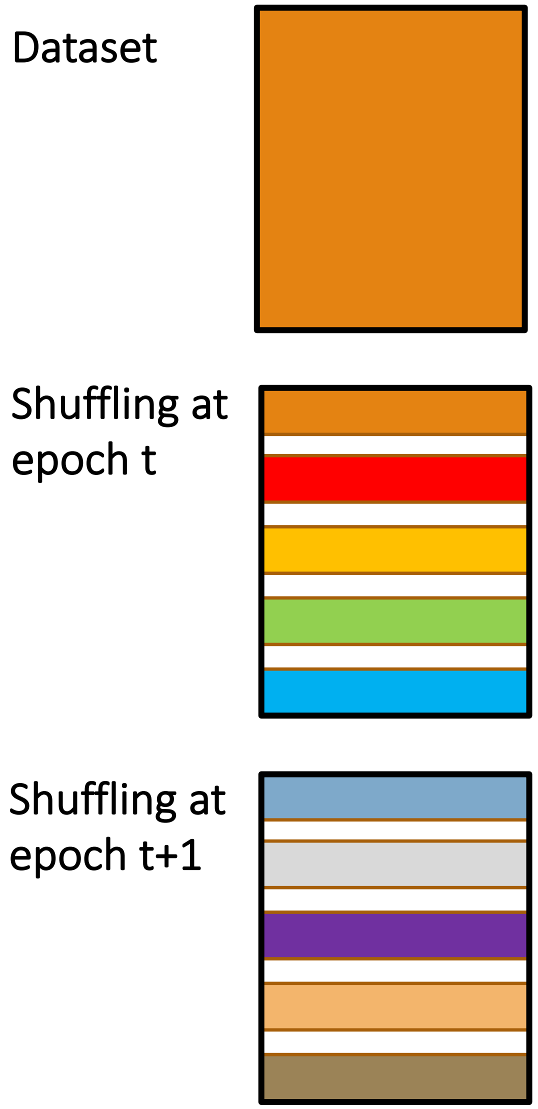
Kiki challenge

Popular in 2014

Popular in 2010

Shuffling examples

- Applicable only with SGD
- Choose samples with maximum information content
  - Mini-batches should contain examples from different classes
  - Prefer samples likely to generate larger errors
    - Otherwise gradients will be small → slower learning
    - Check the errors from previous rounds and prefer “hard examples”
    - Don’t overdo it though :P, beware of outliers
- In practice, split your dataset into mini-batches
  - New epoch →create new randomly shuffled batches



In practice

- SGD is preferred to Gradient Descent
- Training is orders of magnitude faster
  - In real datasets Gradient Descent is not even realistic
- Solutions generalize better
  - Noisier gradients can help escape local minima
  - More efficient → larger datasets → better generalization
- How many samples per mini-batch?
  - Hyper-parameter, trial & error
  - Usually between 32-256 samples
  - A good rule of thumb → as many as your GPU fits

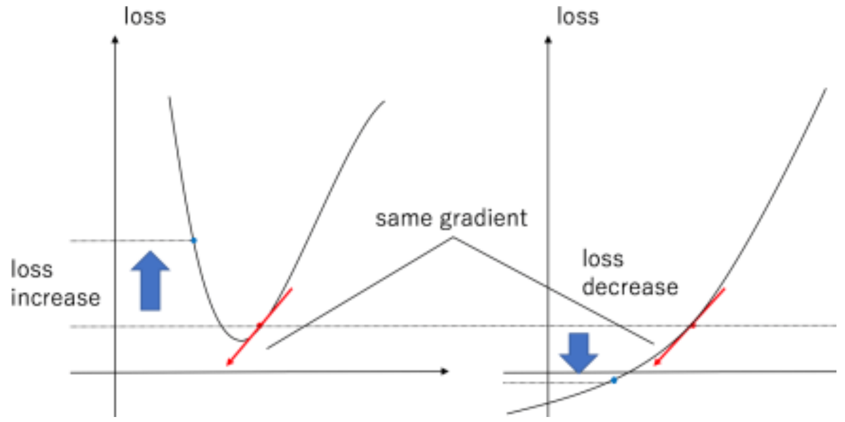
Challenges in optimization

- Ill conditioning
  - Let's check the 2nd order Taylor dynamics for optimizing the cost function

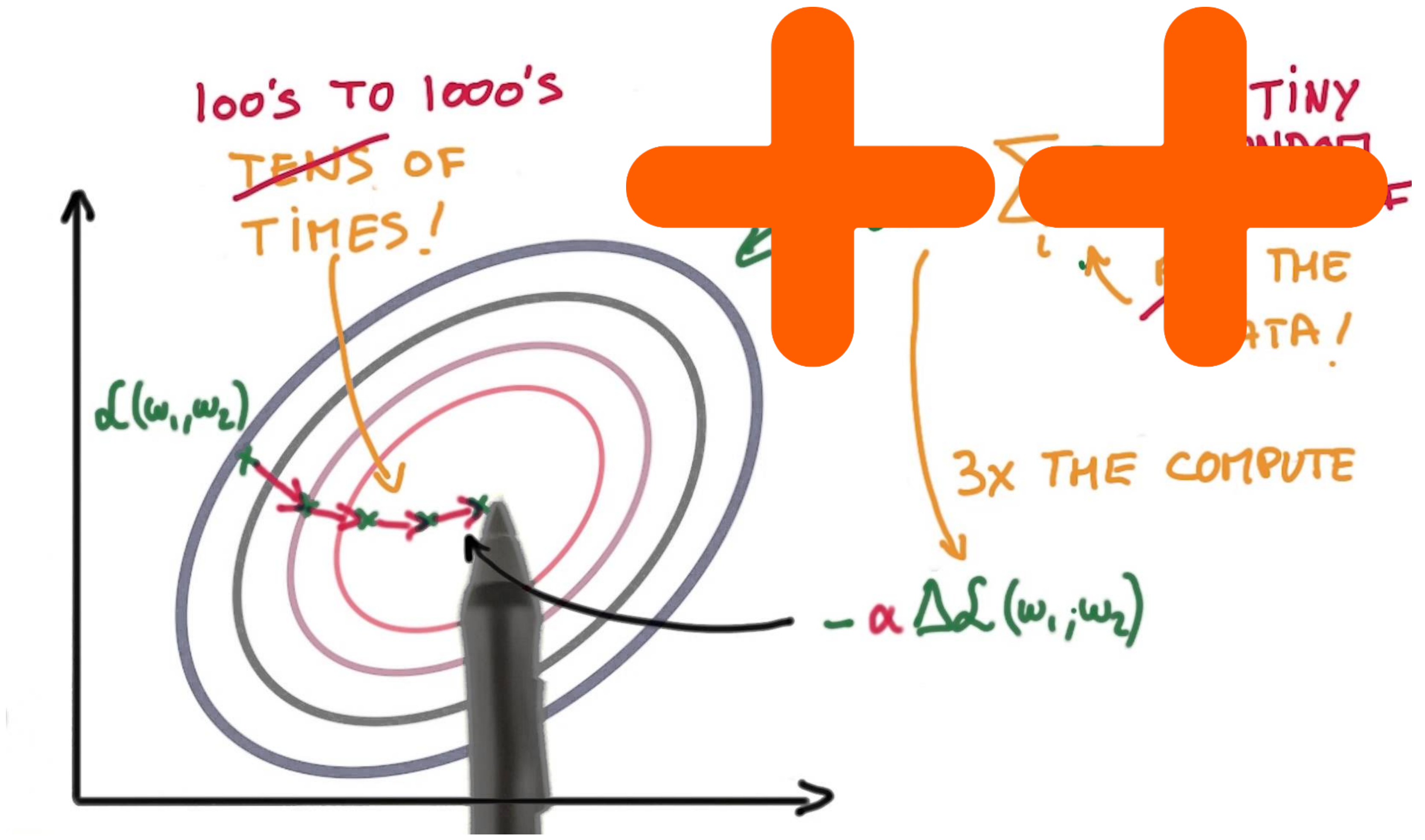
$$\mathcal{L}(\theta) = \mathcal{L}(\theta') + (\theta - \theta')^{\mathrm{T}} g + \frac{1}{2}(\theta - \theta')^{\mathrm{T}} \mathrm{H}(\theta - \theta') \quad (\mathrm{H}\text{:Hessian})$$

$$\mathcal{L}(\theta' - \varepsilon g) \approx \mathcal{L}(\theta') - \varepsilon g^{\mathrm{T}} g + \frac{1}{2} g^T H g$$

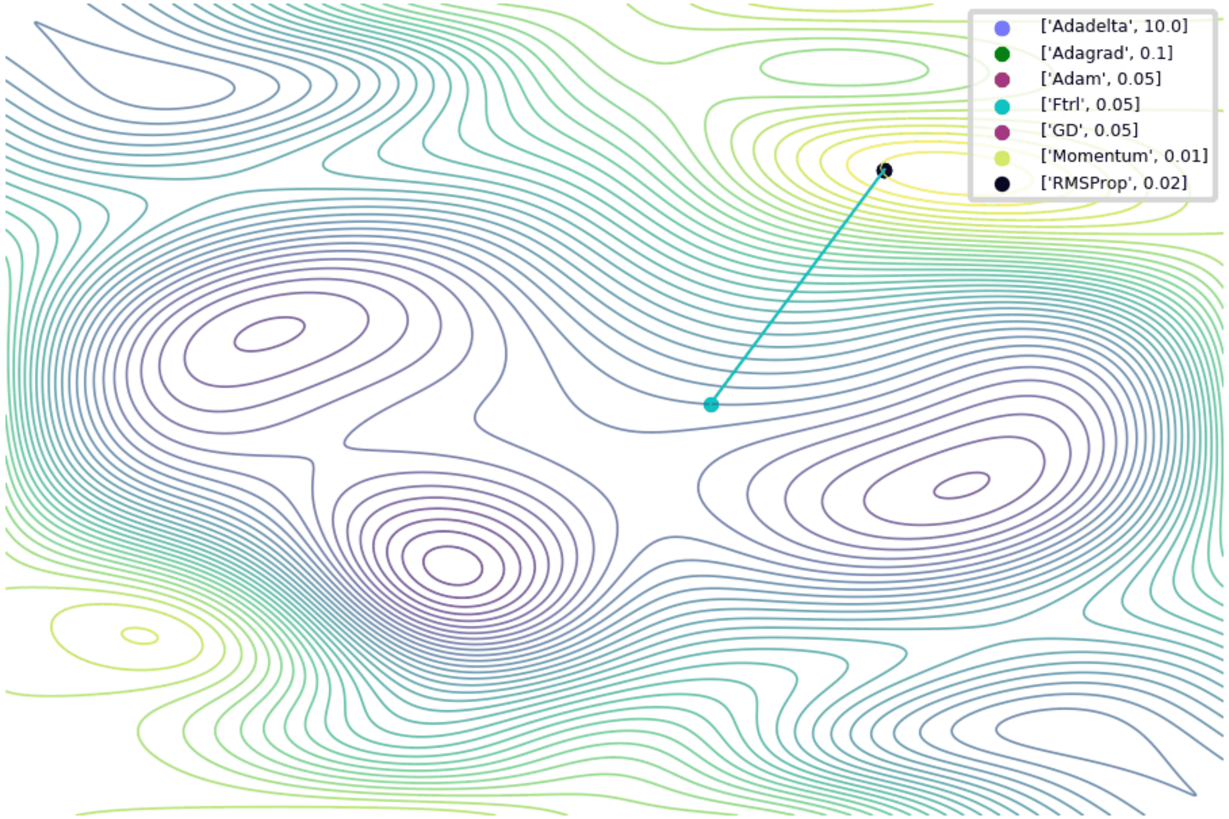
  - Even if the gradient g is strong, if $\frac{1}{2} g^T H g > \varepsilon g^{\mathrm{T}} g$ the cost will increase
- Local minima
  - Non-convex optimization produces lots of equivalent, local minima
- Plateaus and cliffs
- Vanishing and exploding gradients
- Long-term dependencies

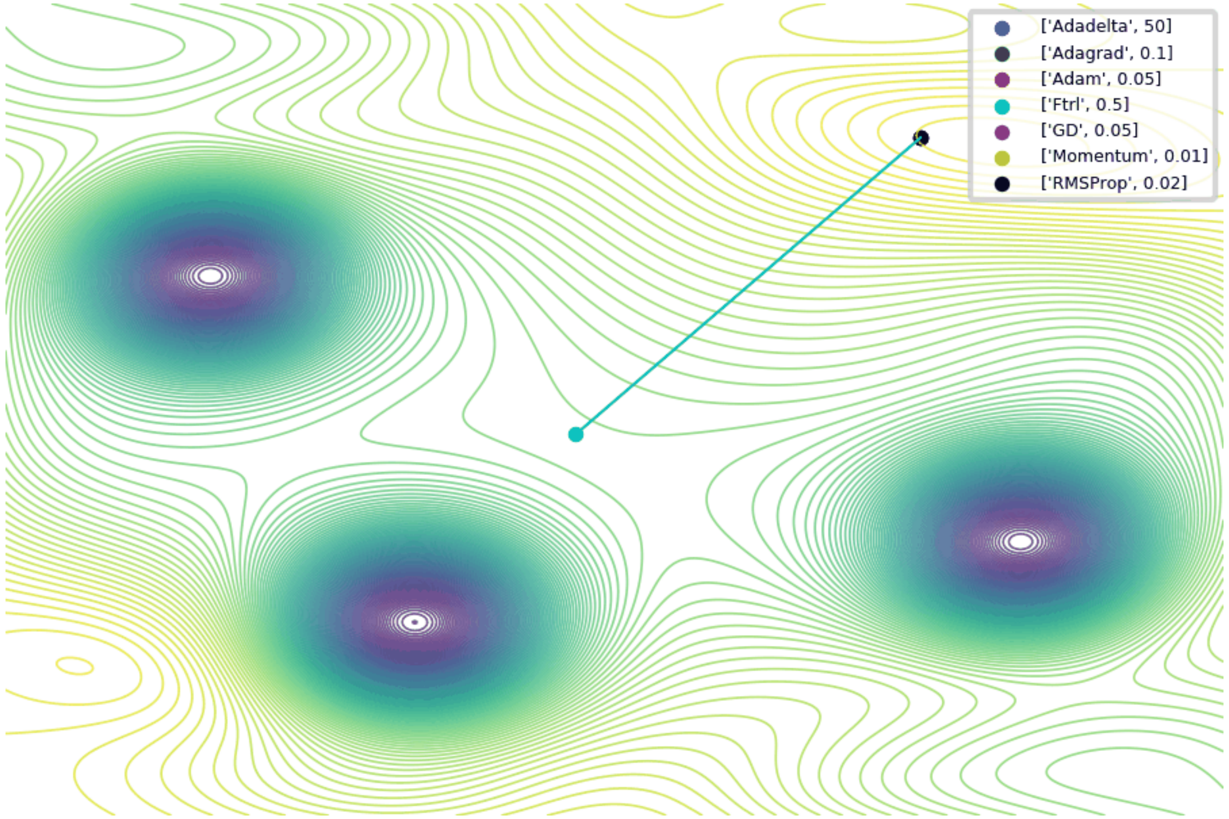


Advanced Optimizations



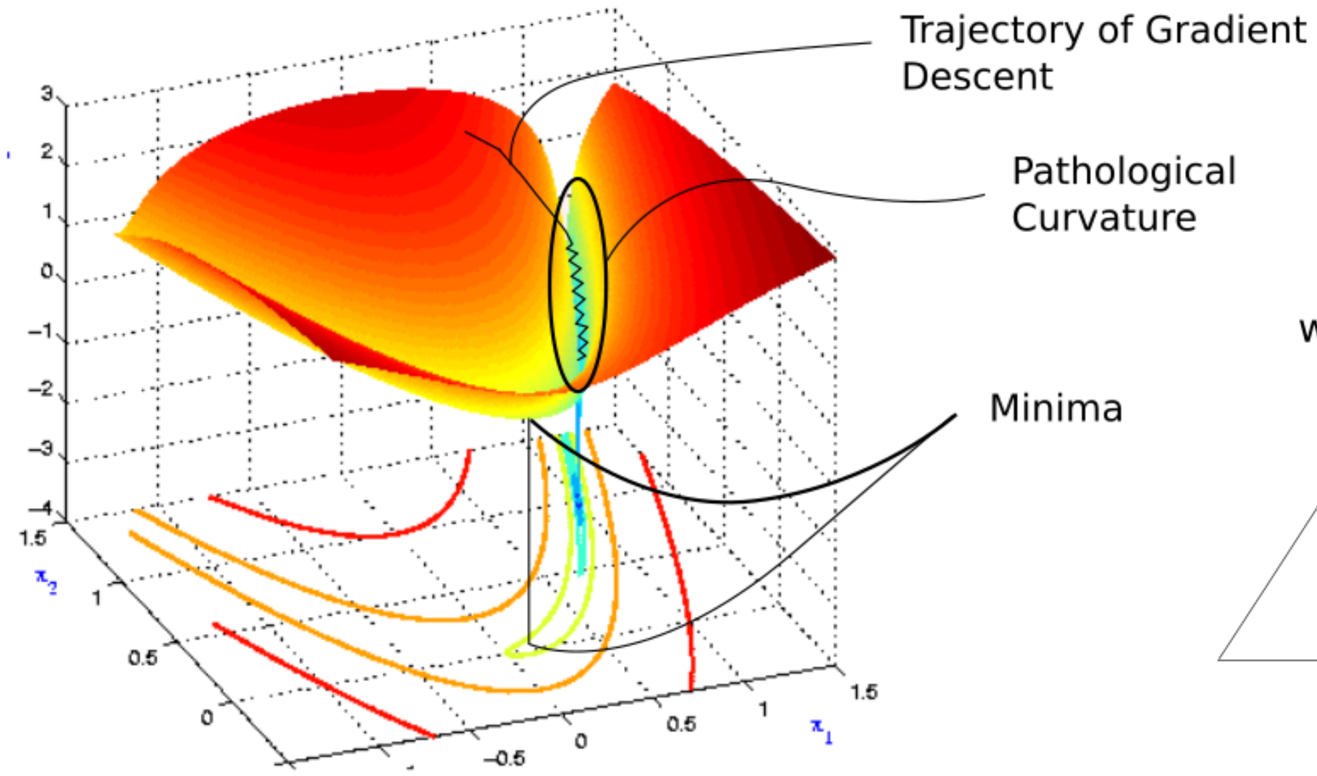
Using different optimizers

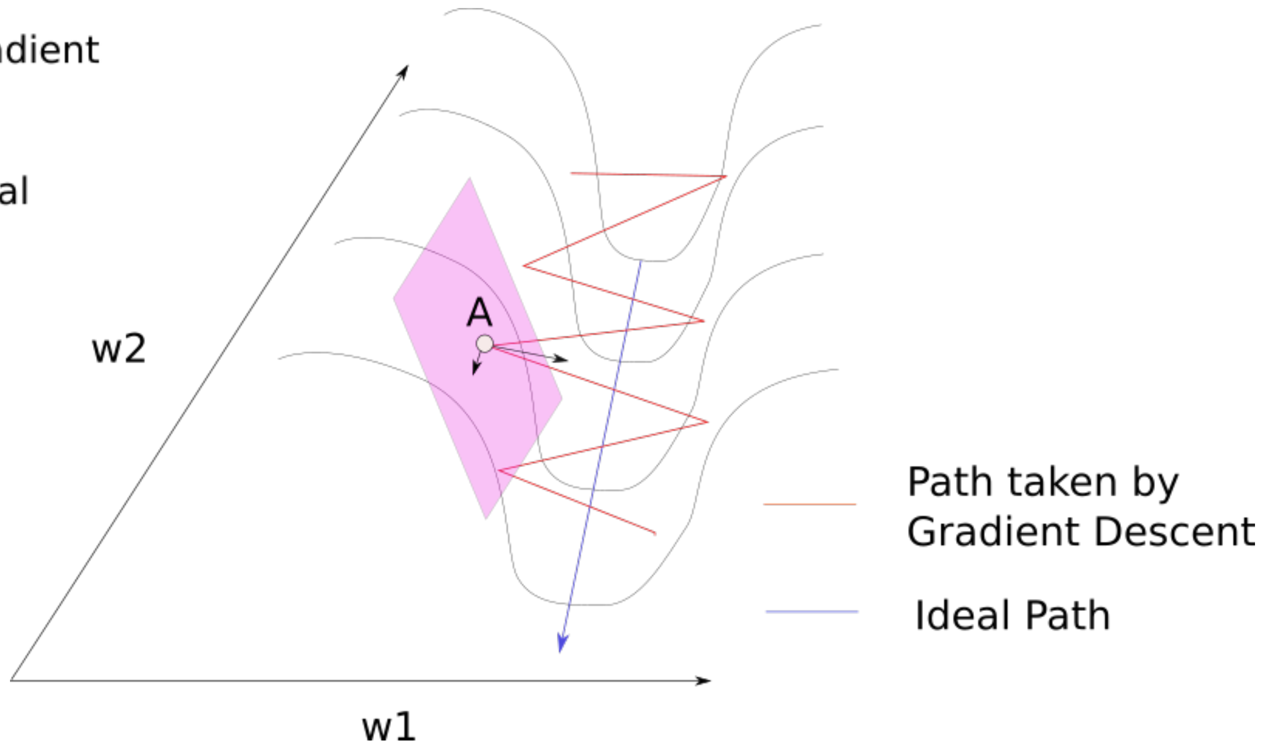




Picture credit: Jaewan Yun

Pathological curvatures





Picture credit: Team Paperspace

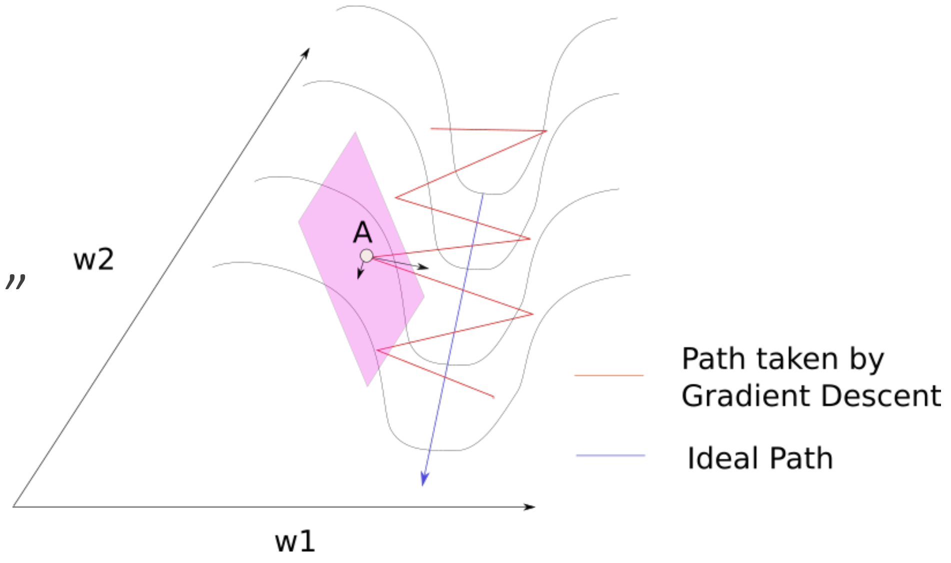
Second order optimization

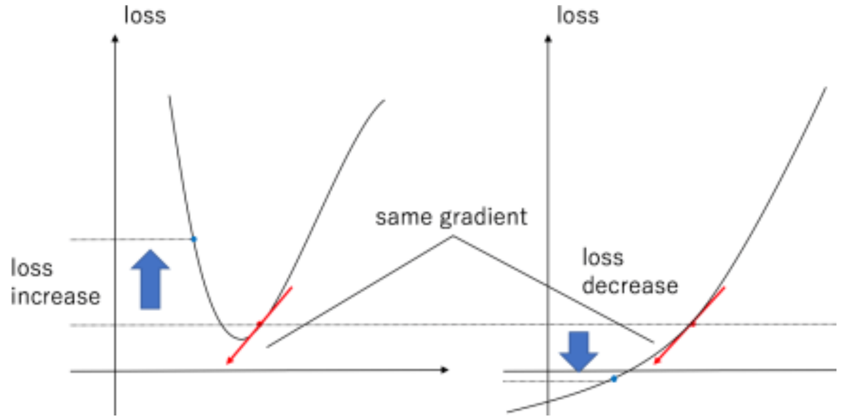
- Normally all weights updated with same "aggressiveness"
  - Often some parameters could enjoy more "teaching"
  - While others are already about there
- Adapt learning per parameter

$$w_{t+1} = w_t - H_{\mathcal{L}}^{-1} \eta_t g_t$$

- $H_{\mathcal{L}}$ is the Hessian matrix of $\mathcal{L}$: second-order derivatives

$$H_{\mathcal{L}}^{ij} = \frac{\partial \mathcal{L}}{\partial w_i \partial w_j}$$





Is it easy to use the Hessian in a Deep Network?

- Yes, you just use the auto-grad
- Yes, you just compute the square of your derivatives
- No, the matrix would be too huge

Is it easy to use the Hessian in a Deep Network?

- Yes, you just use the auto-grad
- Yes, you just compute the square of your derivatives
- No, the matrix would be too huge

Second order optimization methods in practice

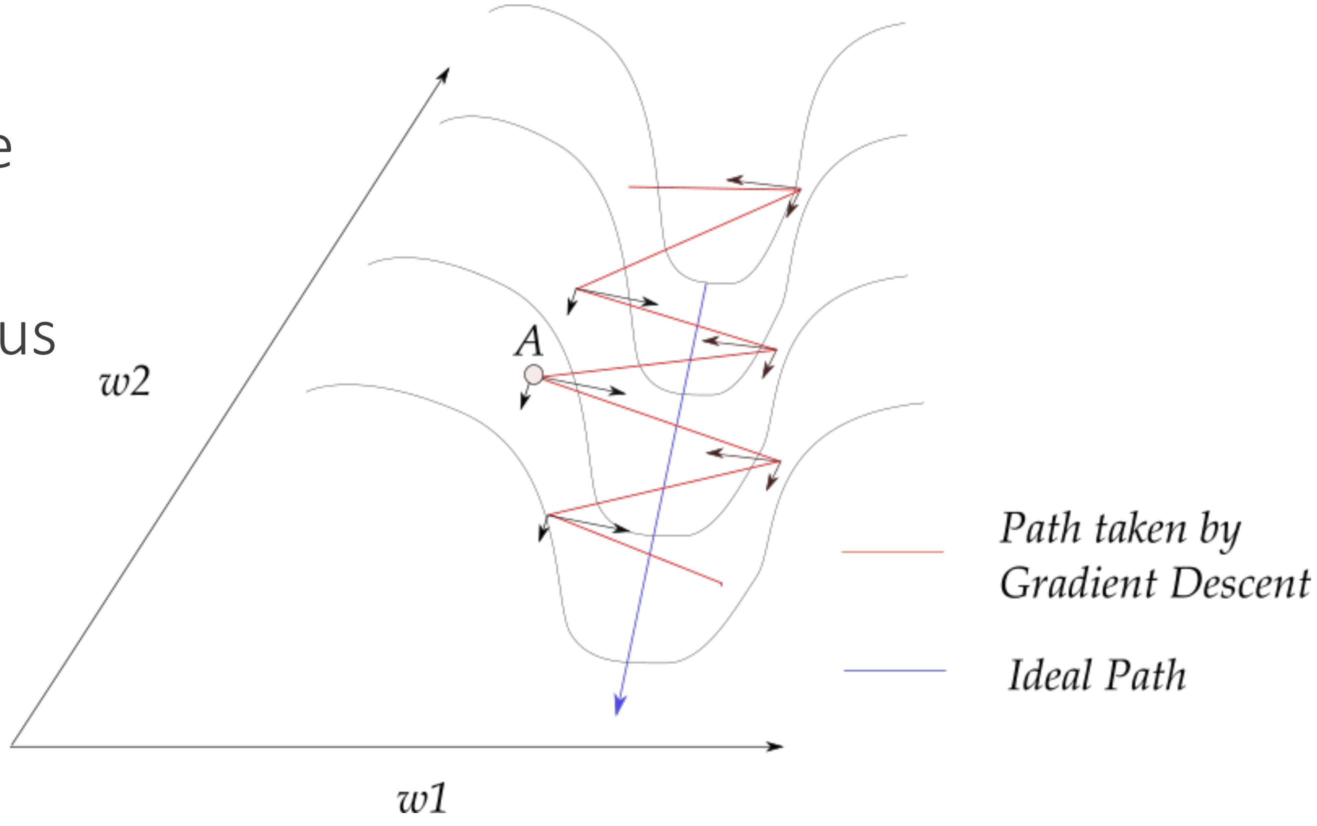
- Inverse of Hessian usually very expensive
  - Too many parameters
- Approximating the Hessian, e.g. with the L-BFGS algorithm
  - Keeps memory of gradients to approximate the inverse Hessian
  - L-BFGS works alright with Gradient Descent. What about SGD?
- In practice, SGD with momentum works just fine quite often

Momentum

- Don’t switch update direction all the time
- Maintain “momentum” from previous updates → dampens oscillations

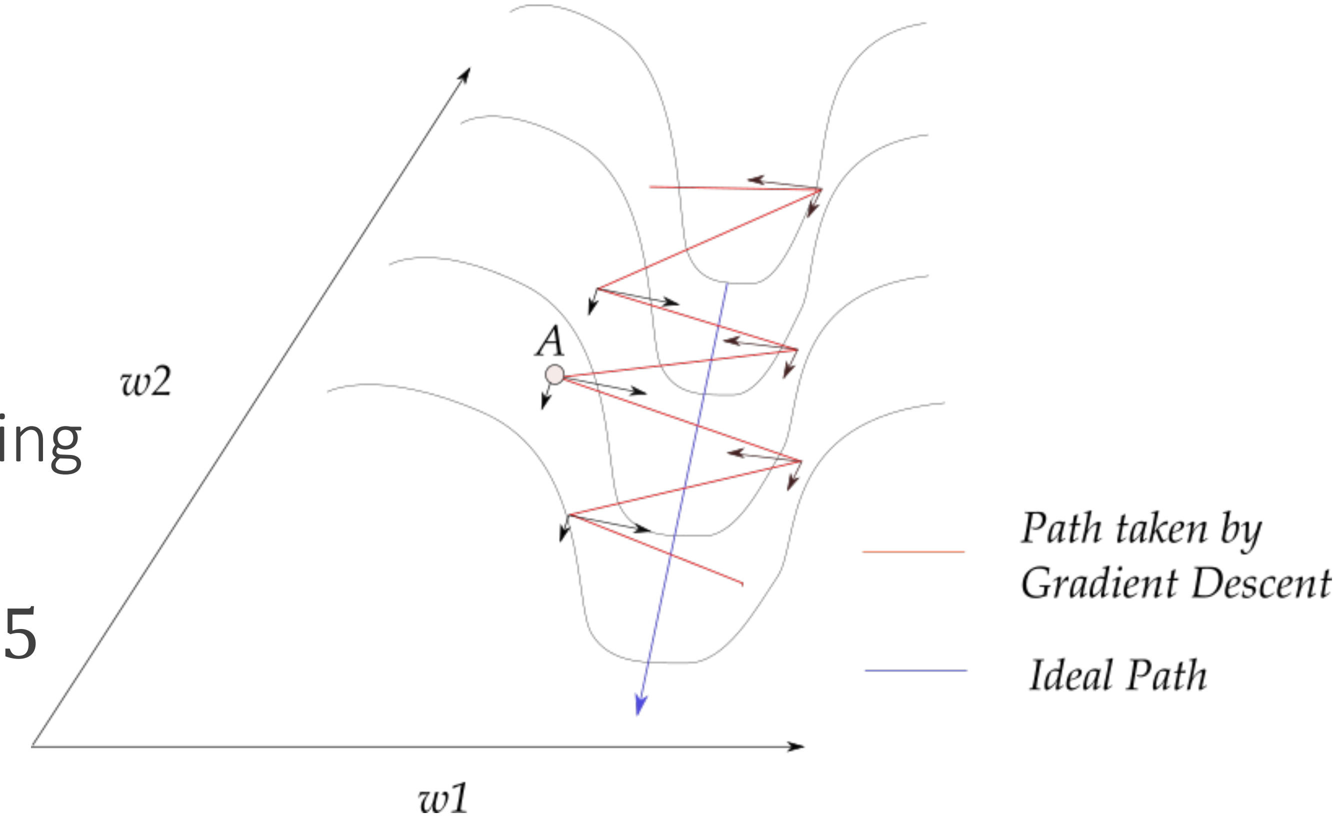
$$u_{t+1} = \gamma u_t - \eta_t g_t$$
$$w_{t+1} = w_t + u_{t+1}$$

- Exponential averaging
  - With $\gamma = 0.9$ and $u_0 = 0$
  - $u_1 \propto -g_1$
  - $u_2 \propto -0.9 g_1 - g_2$
  - $u_3 \propto -0.81 g_1 - 0.9 g_2 - g_3$



Momentum

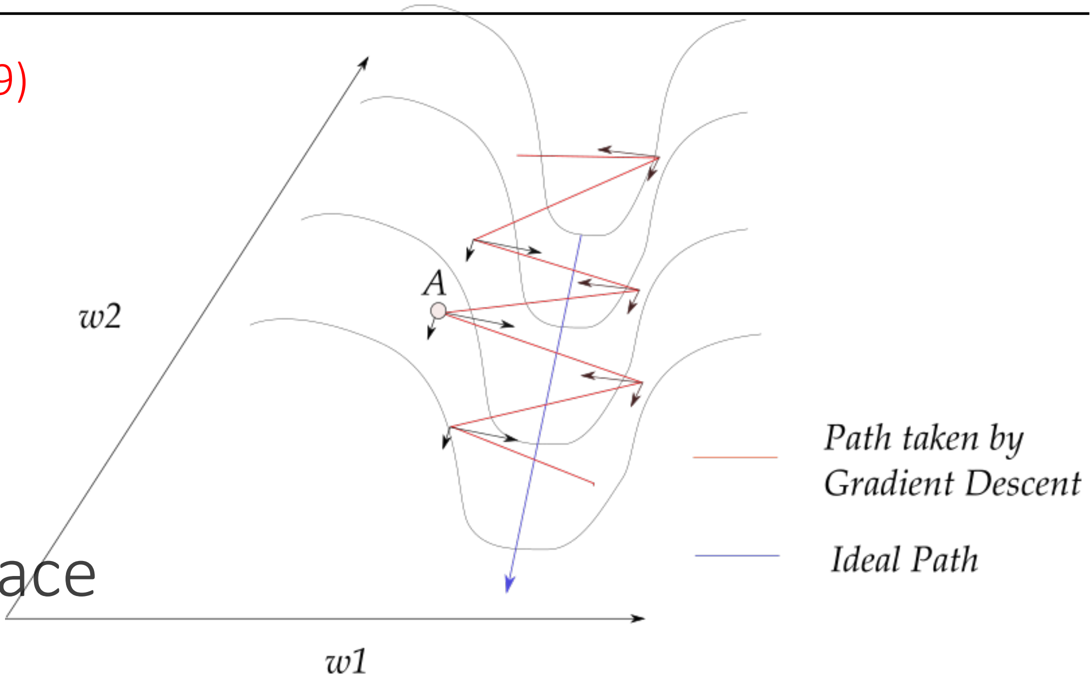
- The exponential averaging
  - cancels out the oscillating gradients
  - gives more weight to recent updates
- More robust gradients and learning → faster convergence
- In practice, initialize $\gamma = \gamma_0 = 0.5$ and anneal to $\gamma_\infty = 0.9$

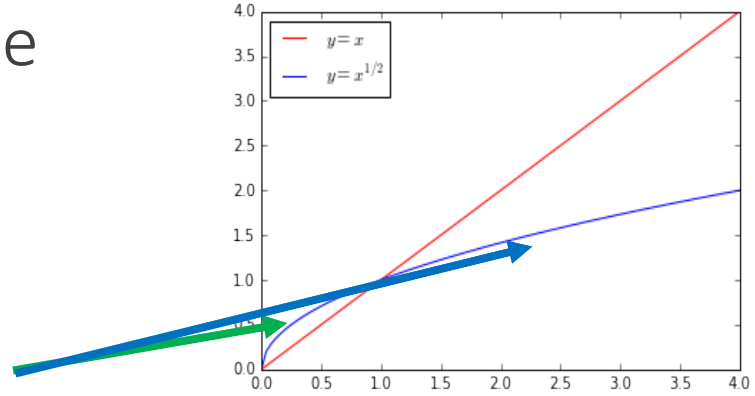


RMSprop

Decay hyper-parameter (Usually 0.9)

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





Square rooting boosts small values while suppresses large values

Adam [Kingma2014]

- One of the most popular learning algorithms

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

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

- Similar to RMSprop, but with momentum & correction bias

Adagrad [Duchi2011]

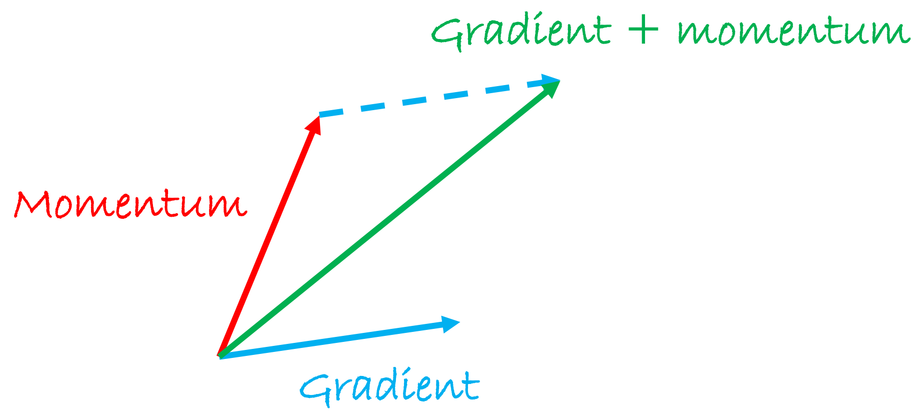
- Schedule
  - $r = \sum_\tau (\nabla_\theta \mathcal{L})^2 \implies w_{t+1} = w_t - \eta \frac{g_t}{\sqrt{r} + \varepsilon}$
  - Gradients become gradually smaller and smaller

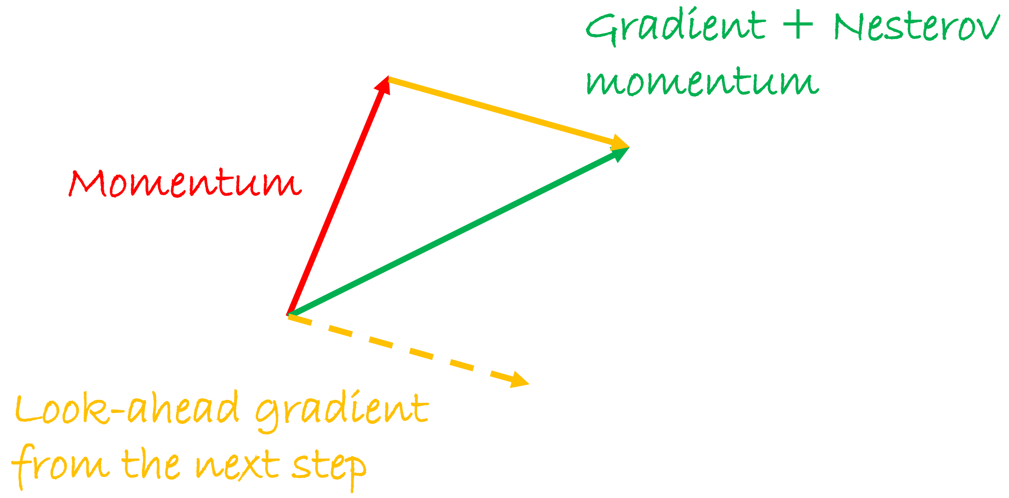
Nesterov Momentum [Sutskever2013]

- Use the future gradient instead of the current gradient

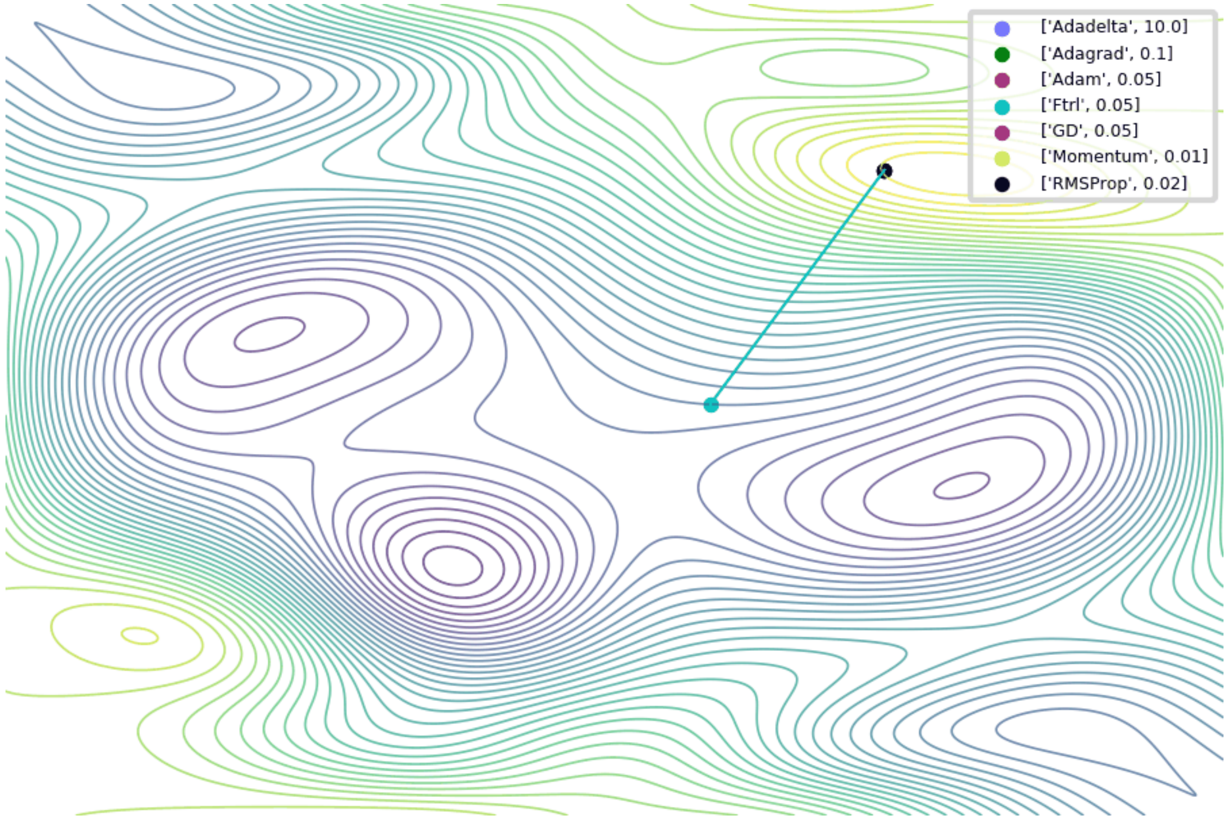
$$w_{t+0.5} = w_t + \gamma u_\tau$$
$$u_{t+1} = \gamma u_\tau - \eta_t \nabla_{w_{t+0.5}} \mathcal{L}$$
$$w_{t+1} = w_t + u_{t+1}$$

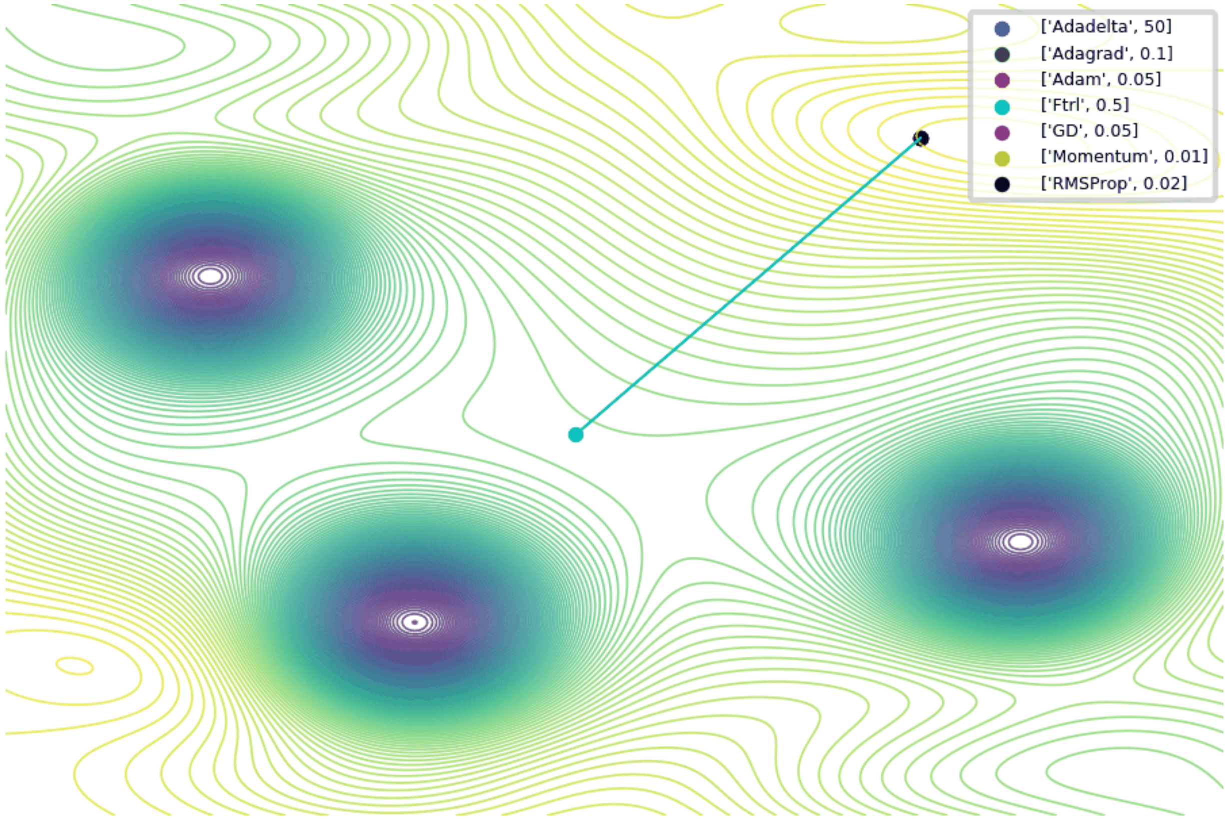
- Better theoretical convergence
- Generally works better with Convolutional Neural Networks




Gradient + Nesterov momentum
Momentum
Look-ahead gradient from the next step

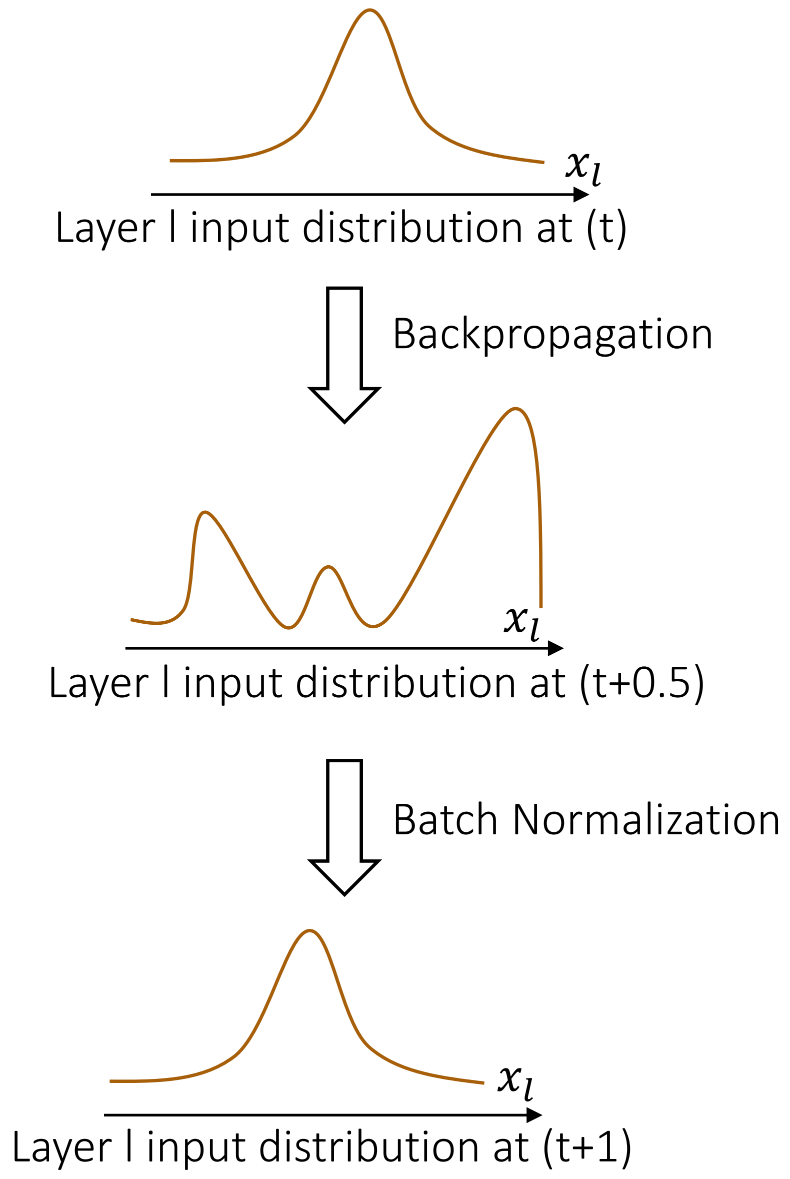
Visual overview





Picture credit: Jaewan Yun

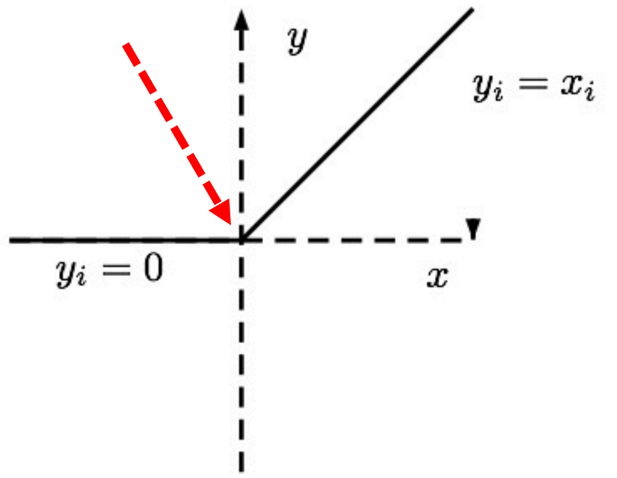
Input Normalization

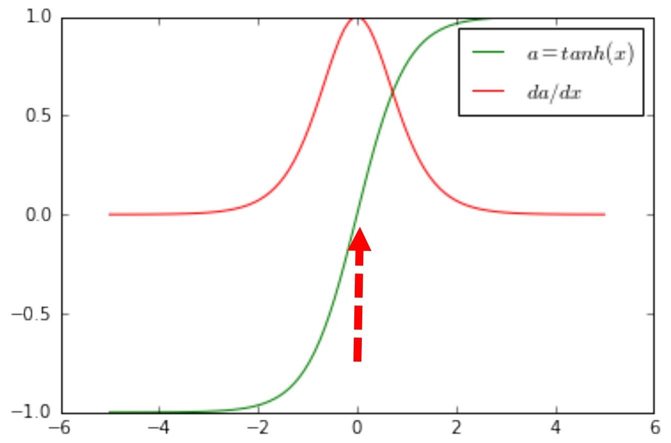

x_l
Layer l input distribution at (t)
Backpropagation
x_l
Layer l input distribution at (t+0.5)
Batch Normalization
x_l
Layer l input distribution at (t+1)

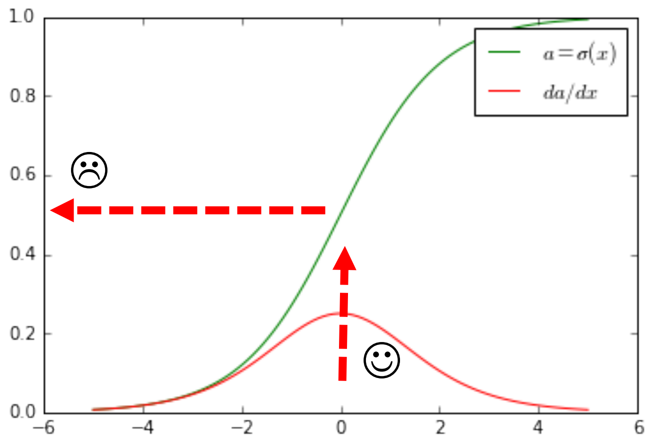
Data pre-processing

- Most common: center data roughly around 0
  - Activation functions usually “centered” around 0
    - Important for propagation to next layer: x=0 → y=0 does not introduce bias within layers (for ReLU and tanh)
    - Important for training: strongest gradients around x=0 (for tanh and sigmoid)

ReLU ☺ $\tanh(x)$ ☺ $\sigma(x)$ ☹

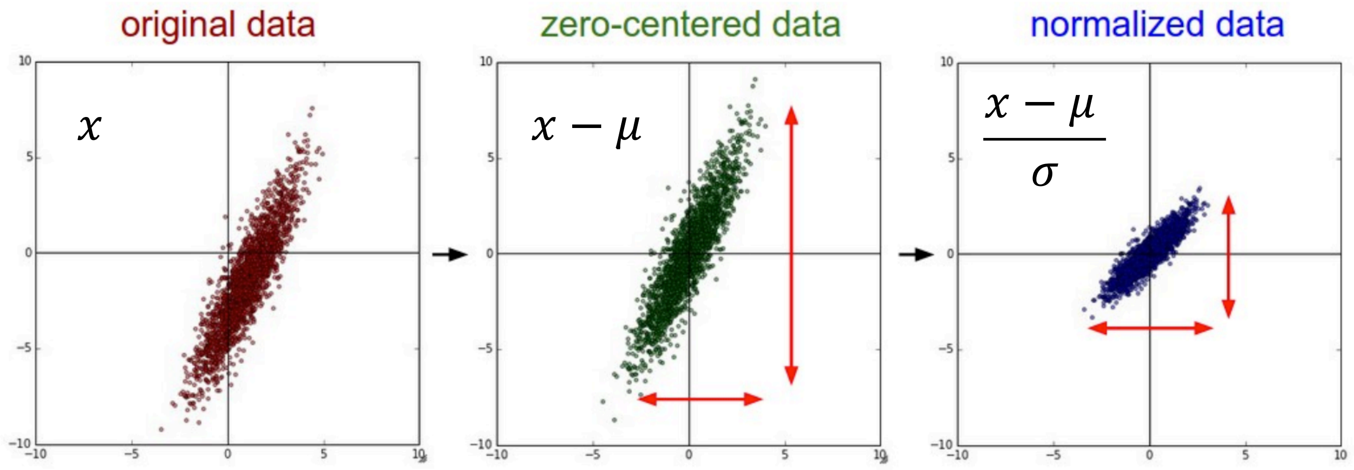






Unit Normalization: $N(\mu, \sigma^2) \rightarrow N(0, 1)$

- Assume: Input variables follow a Gaussian distribution (roughly)
- Normalize by:
  - Computing mean and standard deviation from **training set**
  - Subtracting the mean from training/validation/testing samples and dividing the result by the standard deviation



Picture credit: Stanford Course

Even simpler: Centering the input

- When input dimensions have similar ranges …

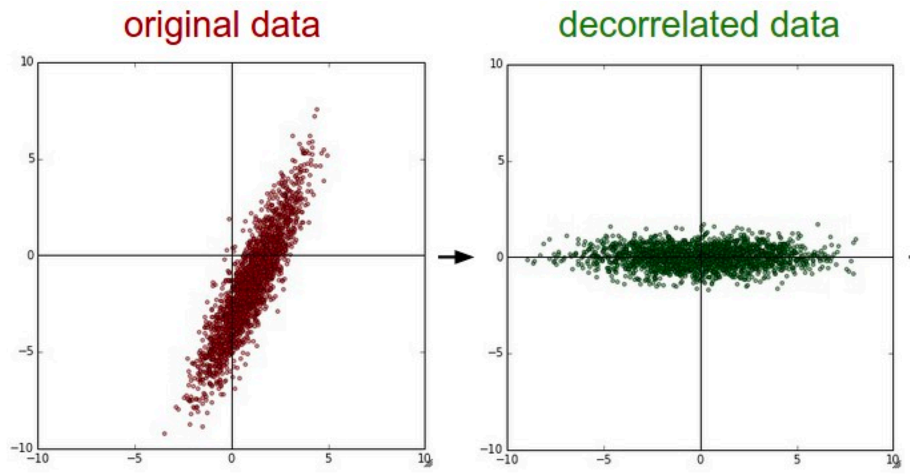
  … and with the right non-linearity …

  … centering might be enough (i.e. subtract the mean)

  - e.g. in images all dimensions are pixels - all pixels have more or less the same ranges

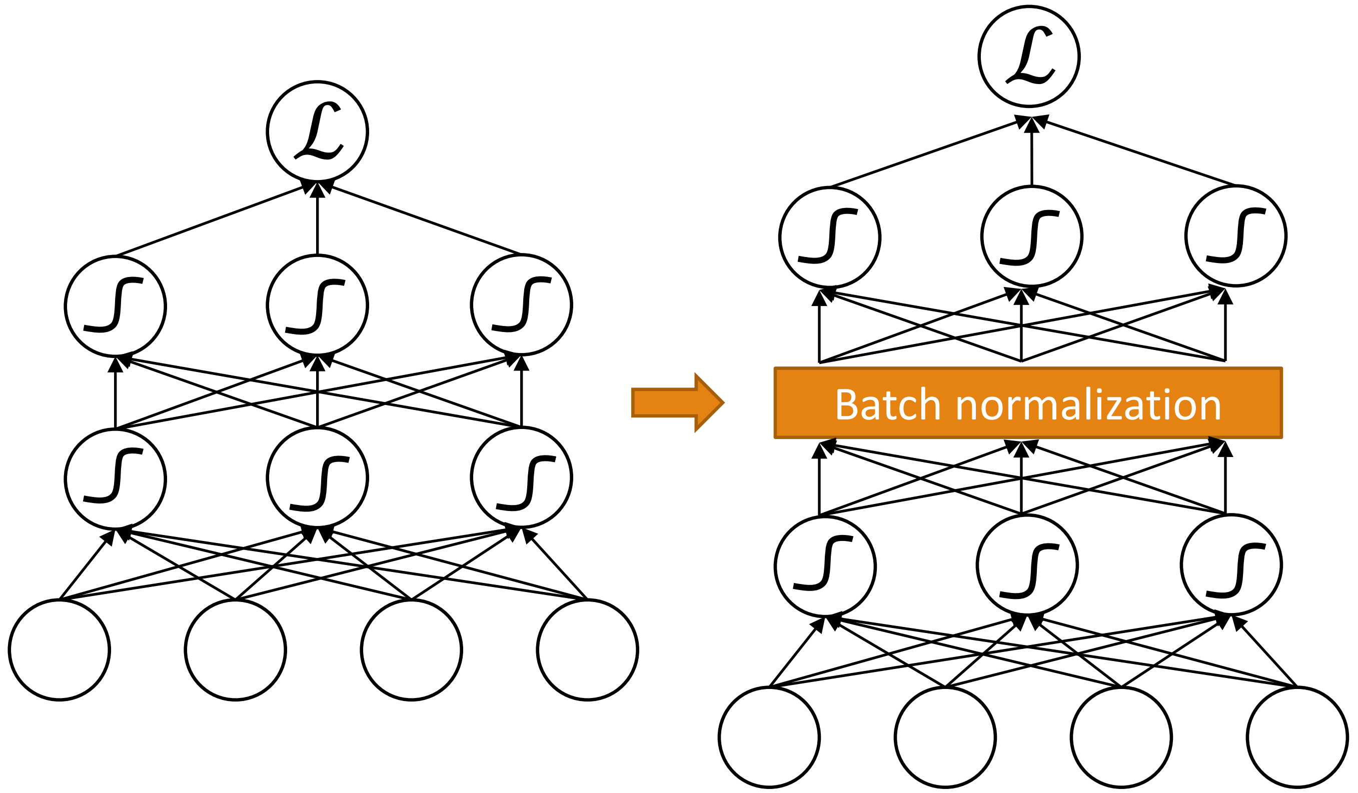
Data pre-processing

- Input variables should be as decorrelated as possible
  - Input variables are “more independent”
  - Model is forced to find non-trivial correlations between inputs
  - Decorrelated inputs → Better optimization
- Obviously decorrelating inputs is not good when inputs are by definition correlated, like in sequences



Batch normalization [Ioffe2015]

- Input distributions change for every layer, especially during training
- Normalize the layer inputs with batch normalization
  - Roughly speaking, normalize x_l to $N(0,1)$, then rescale using trainable parameters



Batch normalization – The algorithm

- $\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$ [compute mini-batch mean]
- $\sigma_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$ [compute mini-batch variance]
- $\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$ [normalize input]
- $\widehat{y}_i \leftarrow \gamma \widehat{x}_i + \beta$ [scale and shift input]

Trainable parameters

What is the mean/stdev Batch Norm $y = \gamma x + \beta$?

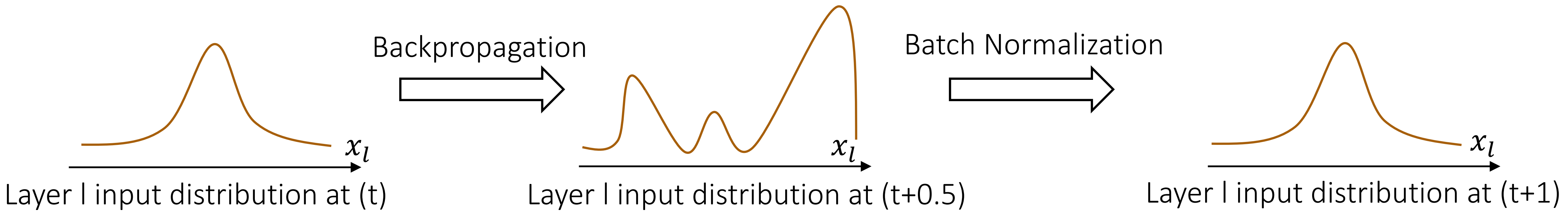
- $\mu = \mu_x + \beta, \sigma = \sigma_x + \gamma$
- $\mu = \beta, \sigma = \gamma$
- $\mu = \beta, \sigma = \beta + \gamma$
- $\mu = \gamma, \sigma = \beta$

What is the mean/stdev Batch Norm $y = \gamma x + \beta$?

- $\mu = \mu_x + \beta, \sigma = \sigma_x + \gamma$
- $\mu = \beta, \sigma = \gamma$
- $\mu = \beta, \sigma = \beta + \gamma$
- $\mu = \gamma, \sigma = \beta$

Batch normalization – Intuition I

- Covariate shift
  - At each step, a layer must not only adapt the weights to fit better the data
  - It must also adapt to the change of its input distribution, as its input is itself the result of another layer that changes over steps
- The distribution fed to the layers of a network should be somewhat:
  - Zero-centered
  - Constant through time and data


Backpropagation
Batch Normalization
x_l
x_l
x_l
Layer l input distribution at (t)
Layer l input distribution at (t+0.5)
Layer l input distribution at (t+1)

Batch normalization – Intuition II

- β, γ are trainable parameters, so when they change there is still internal covariate shift
- 2nd explanation: Batch norm simplifies the learning dynamics
  - Neural network output is determined by higher order interactions between layers; this complicates the gradient update
  - Mean of BatchNorm output is β, std is γ; independent from the activation values themselves → suppresses higher order interactions and makes training easier
- This angle better explains practical observations:
  - Why batch norm works better after the nonlinearity?
  - Why have γ and β if the problem is the covariate shift?

Batch normalization - Benefits

- Can use higher learning rates → faster training
- Neurons of all layers get activated in a near optimal "regime"
- Model regularization
  - Neuron activations not deterministic, depend on the batch
  - Per mini-batch mean and variance are noisy
  - Injected noise reduces overfitting during search

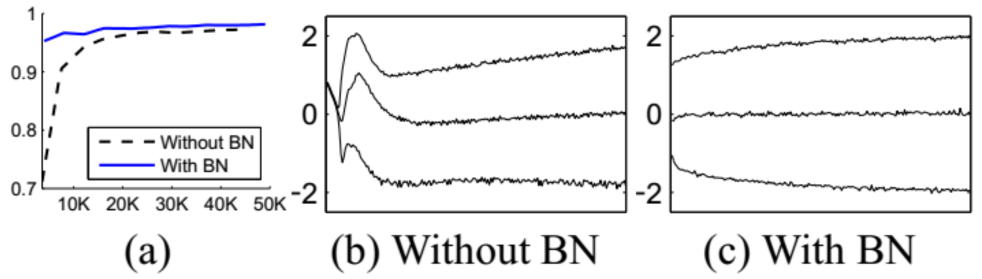


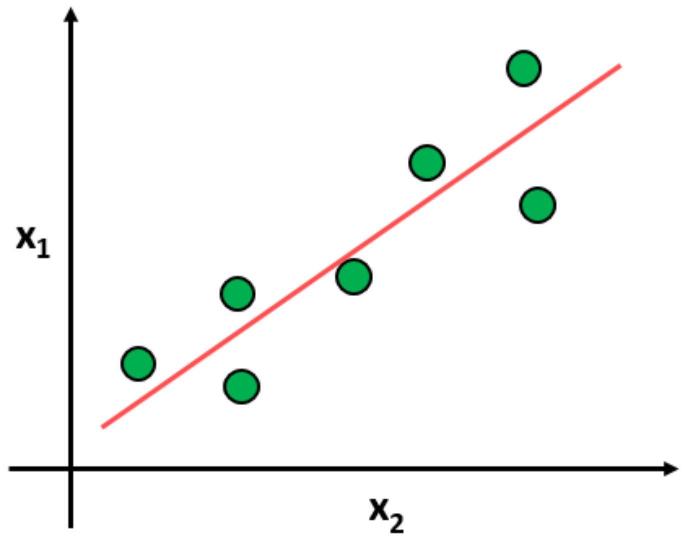
Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

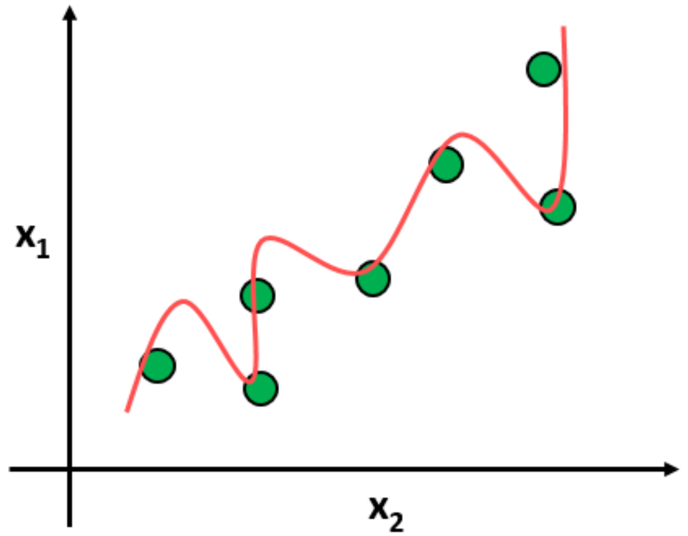
From training to test time

- How do we ship the Batch Norm layer after training?
  - We might not have batches at test time
- Usually: keep a moving average of the mean and variance during training
  - Plug them in at test time
  - To the limit, the moving average of mini-batch statistics approaches the batch statistics

- $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
- $\sigma_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$
- $\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$
- $\hat{y}_i \leftarrow \gamma \hat{x}_i + \beta$

Regularization





Regularization

- Neural networks typically have thousands, if not millions of parameters
  - Usually, the dataset size smaller than the number of parameters
- Overfitting is a grave danger
- Proper weight regularization is crucial to avoid overfitting

$$\mathrm{w}^* \leftarrow \arg\min_w \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; w_{1,\dots,\mathrm{L}})) + \lambda\Omega(\theta)$$

- Possible regularization methods
  - ℓ_2-regularization
  - ℓ_1-regularization
  - Dropout
  - …

ℓ_2-regularization

- Most important (or most popular) regularization

$$\mathrm{w}^* \leftarrow \arg\min_w \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; w_{1,\ldots,\mathrm{L}})) + \frac{\lambda}{2}\sum_l {w_l}^2$$

- The ℓ_2-regularization is added to the gradient descent update rule

$$w_{t+1} = w_t - \eta_t(\nabla_\theta \mathcal{L} + \lambda w_l) \Longrightarrow$$
$$w_{t+1} = (1 - \lambda\eta_t)w^{(t)} - \eta_t \nabla_\theta \mathcal{L}$$

"Weight decay", because weights get smaller

- λ is usually about $10^{-1}, 10^{-2}$

ℓ_1-regularization

- ℓ_1-regularization is one of the most important regularization techniques

$$\mathrm{w}^* \leftarrow \arg\min_w \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; w_{1,\dots,\mathrm{L}})) + \frac{\lambda}{2}\sum_l |w_l|$$

- Also ℓ_1-regularization is added to the gradient descent update rule

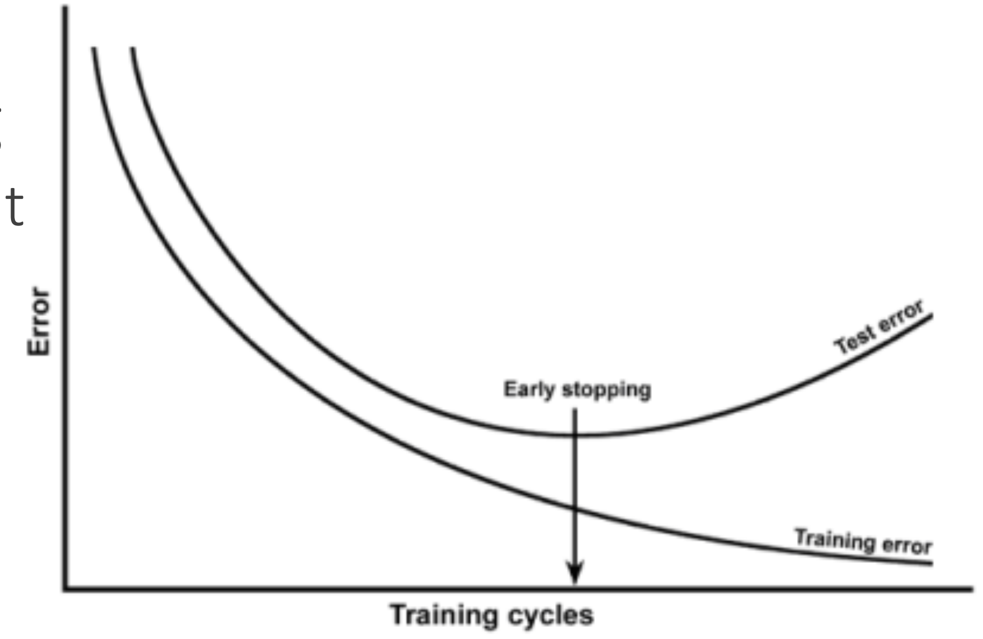
$$w_{t+1} = w_t - \eta_t \left(\nabla_\theta \mathcal{L} + \lambda \frac{w^{(t)}}{|w^{(t)}|} \right)$$

Sign function

- ℓ_1-regularization → sparse weights
  - $\lambda \nearrow$ → more weights become 0

Early stopping

- To tackle overfitting another popular technique is early stopping
- Monitor performance on a separate validation set
- Training the network will decrease training error, as well validation error (although with a slower rate usually)
- Stop when validation error starts increasing
  - This quite likely means the network starts to overfit

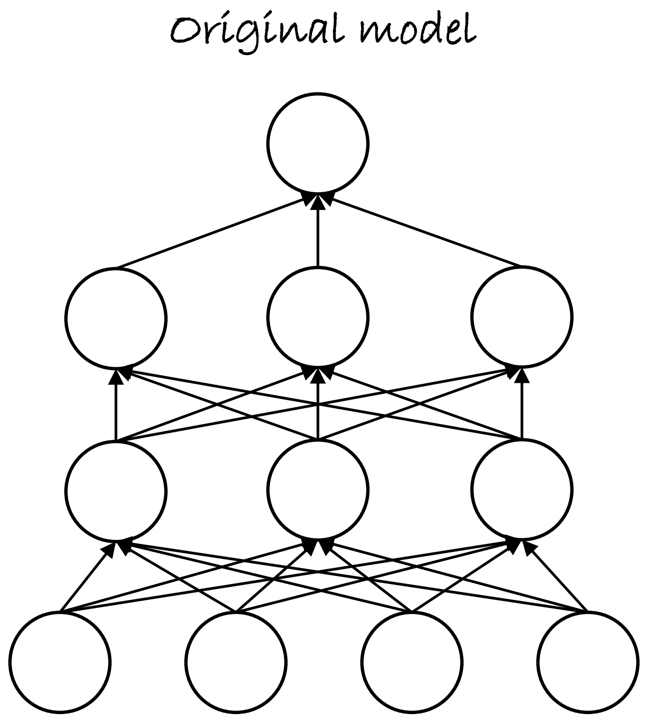


Dropout [Srivastava2014]

- During training randomly set activations to 0
  - Neurons sampled at random from a Bernoulli distribution with $p = 0.5$
- During testing all neurons are used
  - Neuron activations reweighted by p
- Benefits
  - Reduces complex co-adaptations or co-dependencies between neurons
  - Every neuron becomes more robust
  - Decreases overfitting

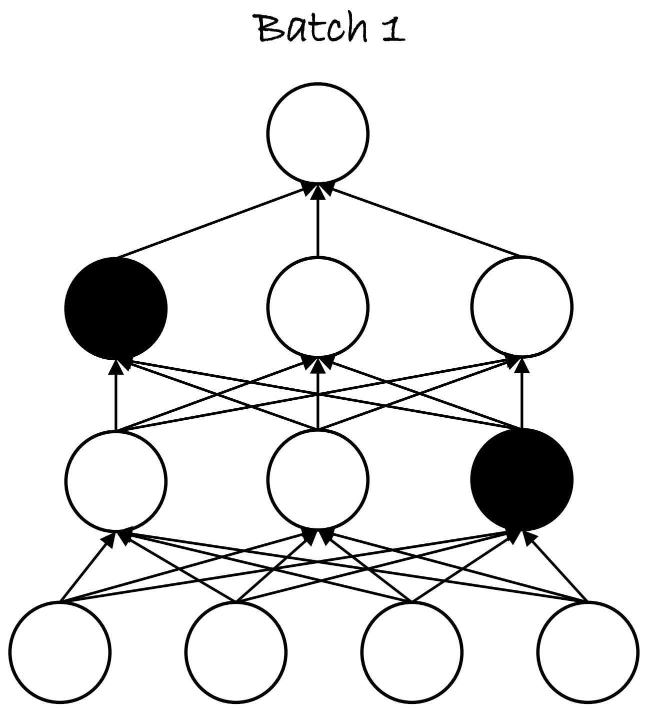
Dropout

- Effectively, a different architecture for every input batch during training
  - Similar to model ensembles



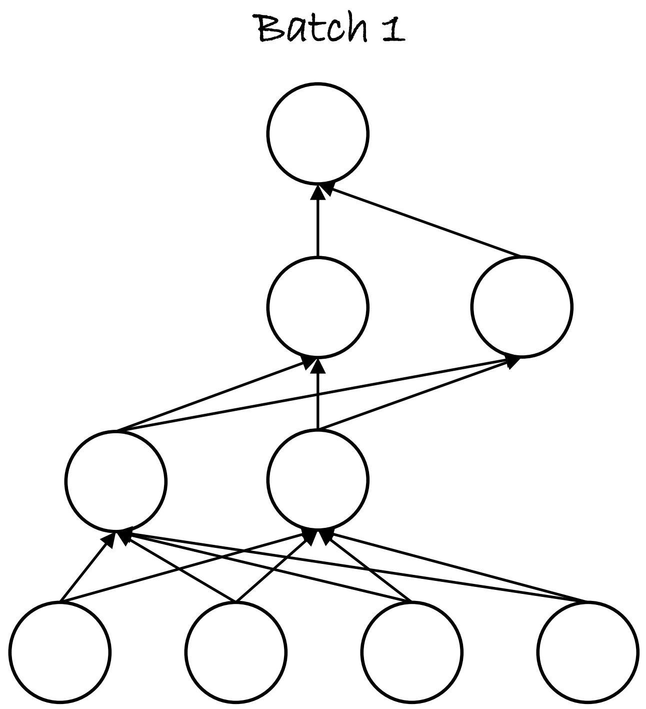
Dropout

- Effectively, a different architecture for every input batch during training
  - Similar to model ensembles



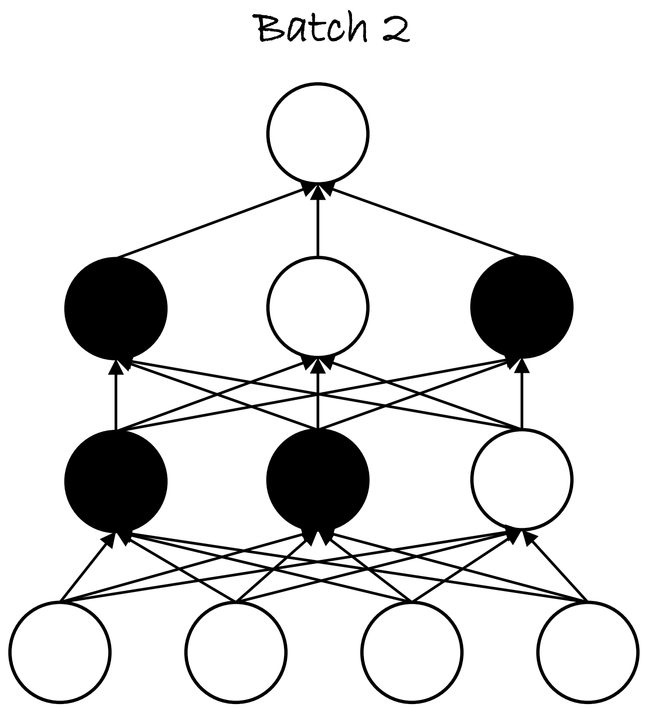
Dropout

- Effectively, a different architecture for every input batch during training
  - Similar to model ensembles



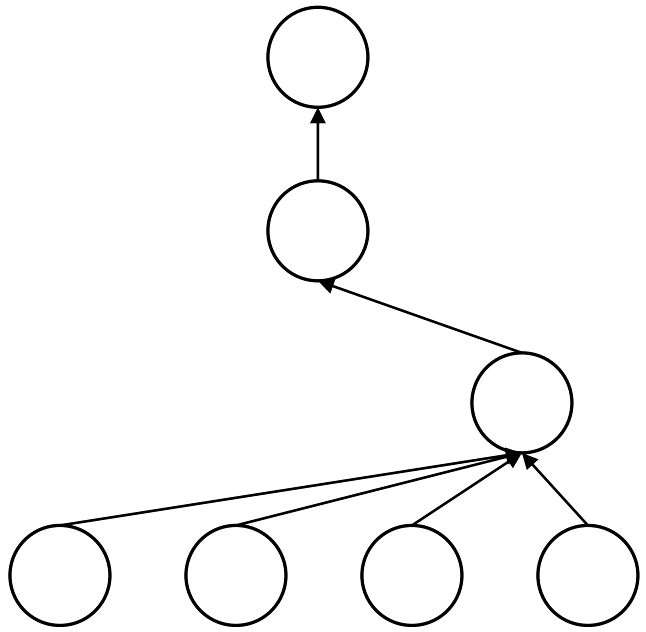
Dropout

- Effectively, a different architecture for every input batch during training
  - Similar to model ensembles

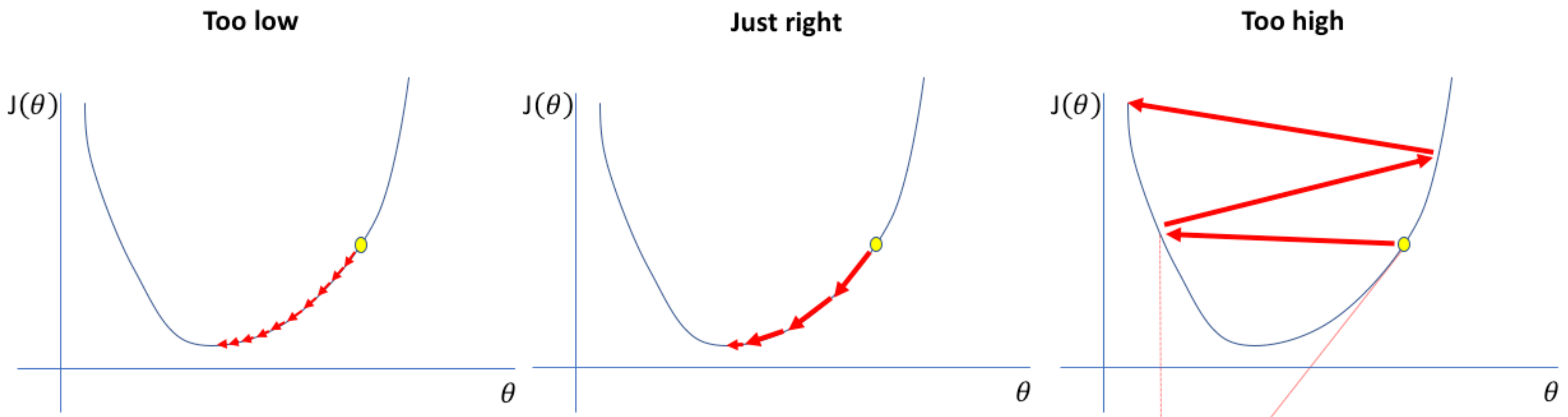


Dropout

- Effectively, a different architecture for every input batch during training
  - Similar to model ensembles



Learning rate



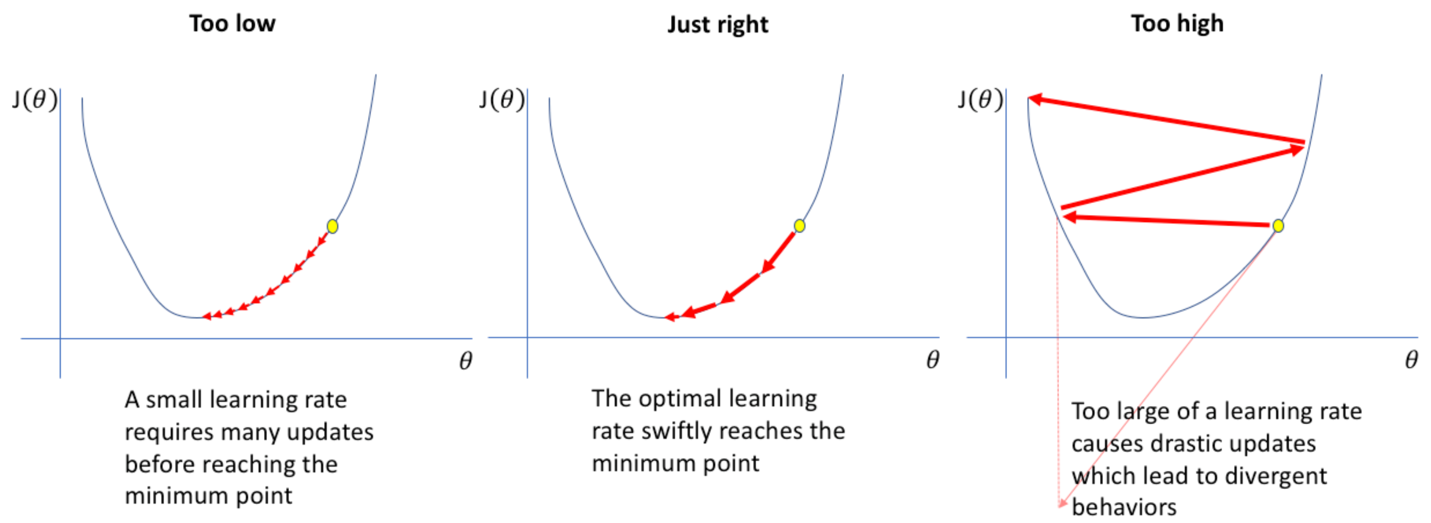
A small learning rate requires many updates before reaching the minimum point

The optimal learning rate swiftly reaches the minimum point

Too large of a learning rate causes drastic updates which lead to divergent behaviors

Learning rate

- The right learning rate η_t very important for fast convergence
  - Too strong → gradients overshoot and bounce
  - Too weak → slow training
- Learning rate per weight is often advantageous
  - Some weights are near convergence, others not


Too low
Just right
Too high
$J(\theta)$
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Convergence

- The step sizes **theoretically** should satisfy the following [Robbins–Monro]

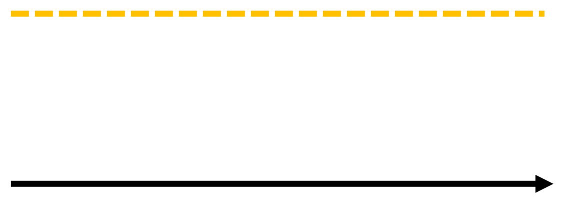
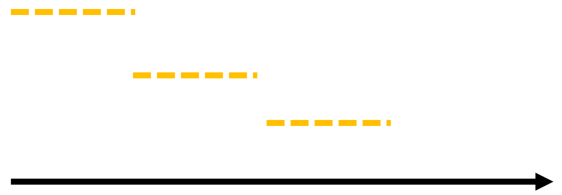
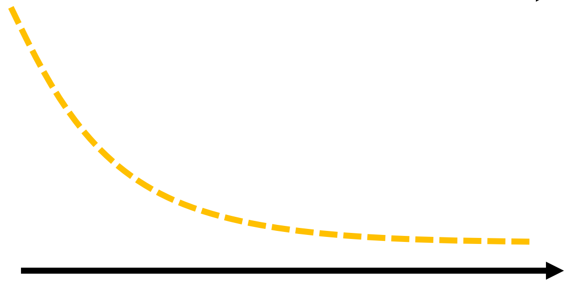
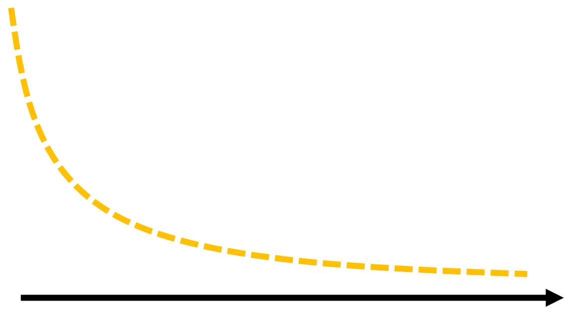
$$\sum_t^\infty \eta_t = \infty \quad \text{and} \quad \sum_t^\infty \eta_t^2 < \infty$$

- Intuitively,
  - The first term ensures that search will explore enough
  - The second term ensures convergence

Learning rate schedules

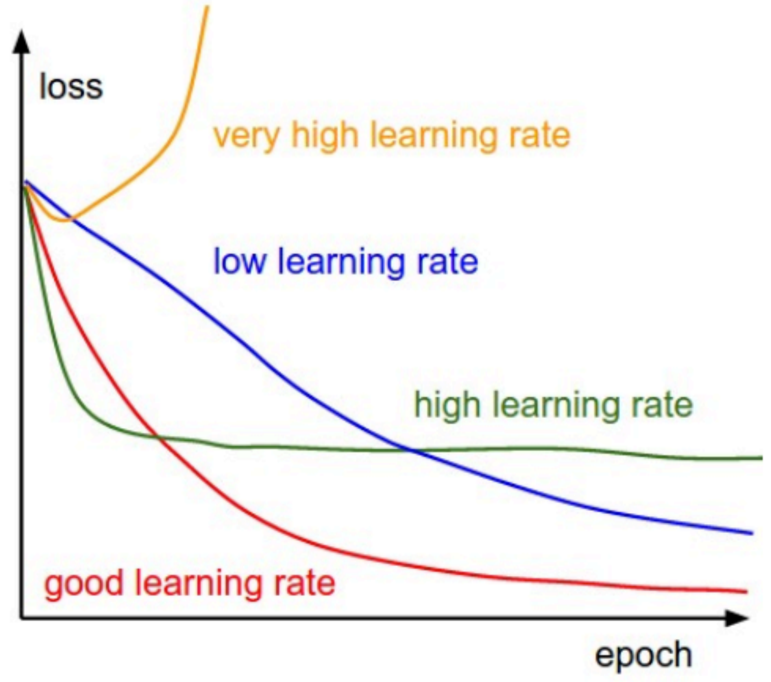
- Constant
  - Learning rate remains the same for all epochs
- Step decay
  - Decrease every T number of epochs
    or when validation loss stopped decreasing
- Inverse decay $\eta_t = \frac{\eta_0}{1+\varepsilon t}$
- Exponential decay $\eta_t = \eta_0 e^{-\varepsilon t}$

- Often step decay preferred
  - simple, intuitive, works well

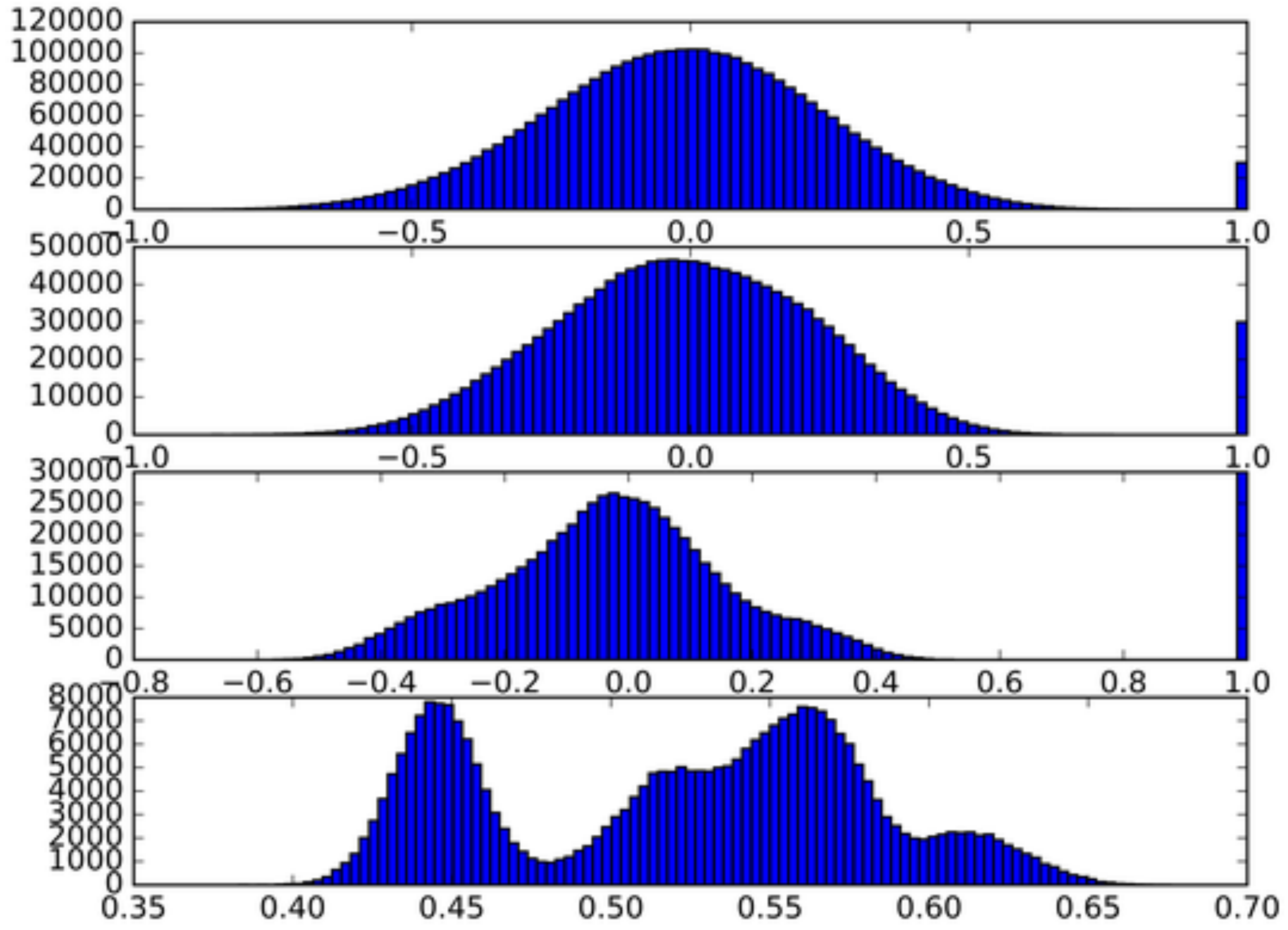
In practice

- Try several log-spaced values $10^{-1}, 10^{-2}, 10^{-3}, \ldots$ on a smaller set
  - Then, you can narrow it down from there around where you get the lowest **validation** error
- You can decrease the learning rate every 10 (or some other value) full training set epochs
  - Although this highly depends on your data



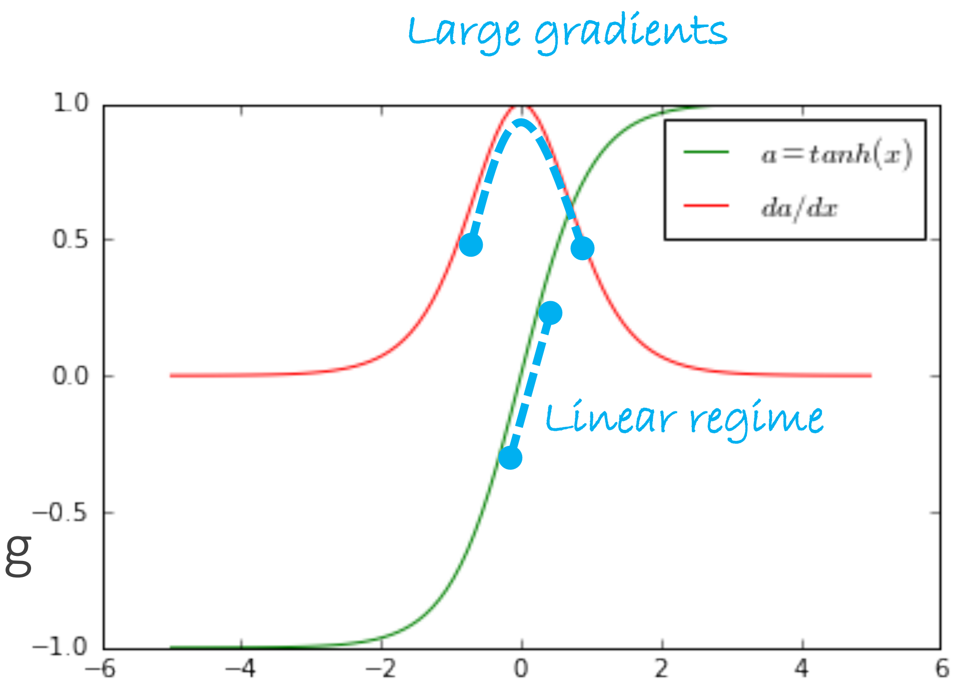
Picture credit:
Stanford Course

Weight initialization



Weight initialization

- There are few contradictory requirements:
- Weights need to be small enough
  - Otherwise output values explode
- Weights need to be large enough
  - Otherwise signal is too weak for any serious learning



- Around origin ($\vec{\mathbf{0}}$) for symmetric functions (tanh, sigmoid)
  - When training starts, better stimulate activation functions near their linear regime
  - larger gradients → faster training

Weight initialization

- Weights must be initialized **to preserve the variance** of the activations during the forward and backward computations

Question: Why similar input/output variance?

- Initialize weights to be different from one another
  - Don’t give same values to all weights (like all $\vec{\mathbf{0}}$)
  - In that case all neurons generate same gradient → no learning
- Generally speaking initialization depends on
  - non-linearities
  - data normalization

Weight initialization

- Weights must be initialized **to preserve the variance** of the activations during the forward and backward computations

Question: Why similar input/output variance?

Answer: Because the output of one module is the input to another

- Initialize weights to be different from one another
  - Don't give same values to all weights (like all $\vec{\mathbf{0}}$)
  - In that case all neurons generate same gradient → no learning
- Generally speaking initialization depends on
  - non-linearities
  - data normalization

One way of initializing weights

- For $a = wx$ the variance is
$$var(a) = E[x]^2 var(w) + \mathrm{E}[w]^2 var(x) + var(x) var(w)$$
- Since $E[x] = E[w] = 0$
$$var(a) = var(x) var(w) \approx d \cdot var(x_i) var(w_i)$$
- For $var(a) = var(x) \Rightarrow var(w_i) = \frac{1}{d}$
- Draw random weights from
$$w \sim N(0, 1/d)$$

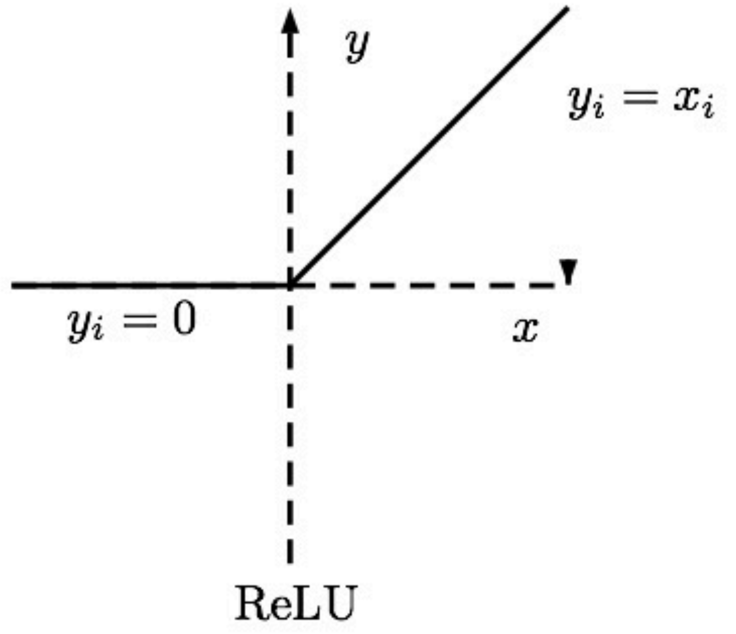
  where d is the number of input variables to the layer

Xavier initialization [Glorot 2010]

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

[He2015] Initialization for ReLUs

- Unlike sigmoidals, ReLUs return 0 half of the time
- Double the weight variance
  - Compensate for the zero flat-area
    → Input and output maintain same variance
- Draw random weights from $\mathrm{w} \sim N(0, 2/d)$ where d is the number of input variables to the layer



Babysitting Deep Nets

- Always check your gradients if not computed automatically
- Check that in the first round you get loss that corresponds to random guess
- Check network with few samples
  - Turn off regularization. You should predictably overfit and get a loss of 0
  - Turn on regularization. The loss should be higher than before
- Have a separate validation set
  - Use validation set for hyper-parameter tuning
  - Compare the curve between training and validation sets - there should be a gap, but not too large
- Preprocess the data (at least to have 0 mean)
- Initialize weights based on activations functions
  - Xavier or He initialization
- Use regularization (ℓ_2-regularization, dropout, ...)
- Use batch normalization

Summary

- SGD and advanced SGD-like optimizers
- Input normalization and Batch normalization
- Regularization
- Learning rate
- Weight initialization

Reading material

- Chapter 8, 11
- And the papers mentioned in the slide

Reading material

Deep Learning Book

- Chapter 8, 11

Papers

- Efficient Backprop
- How Does Batch Normalization Help Optimization? (No, It Is Not About Internal Covariate Shift)

Blog

- https://medium.com/paperspace/intro-to-optimization-in-deep-learning-momentum-rmsprop-and-adam-8335f15fdee2
- http://ruder.io/optimizing-gradient-descent/
- https://github.com/Jaewan-Yun/optimizer-visualization
- https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/