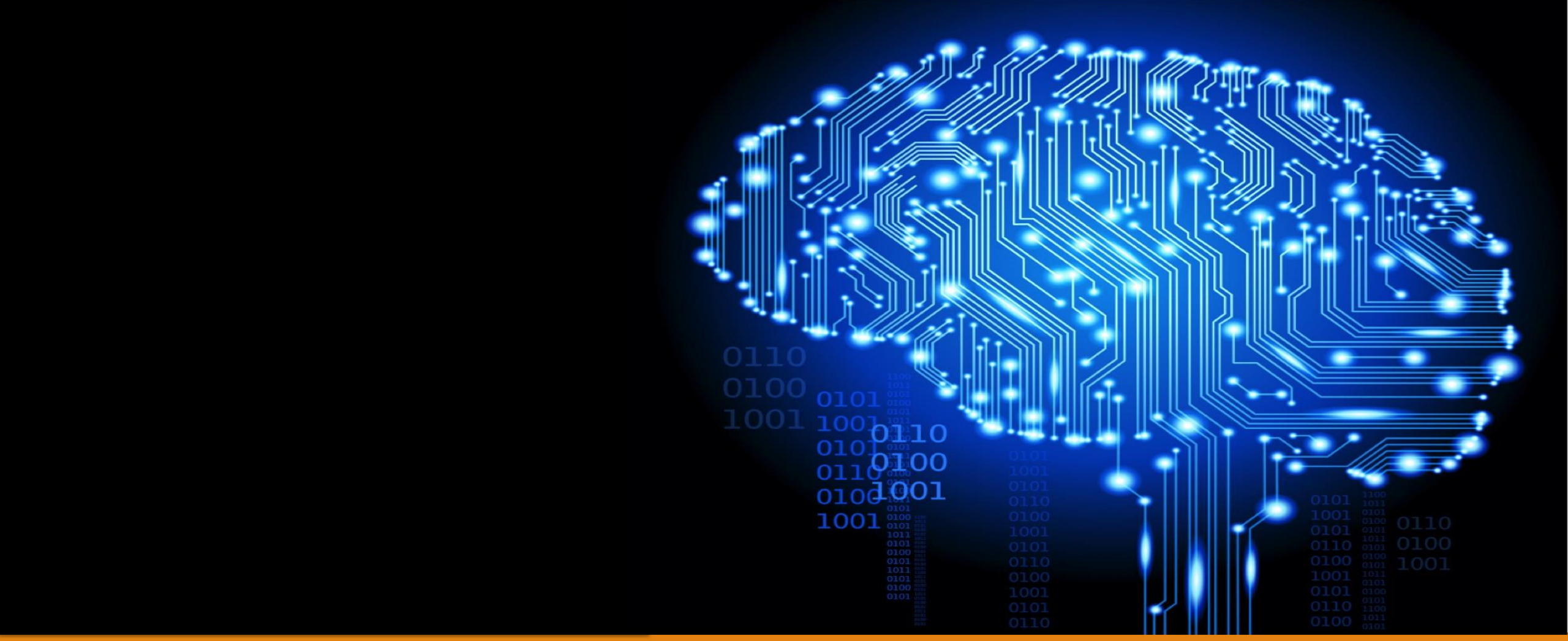


Lecture 3: Deep Learning Optimizations

Deep Learning @ UvA

Exam

- 2 hours
- Multiple choice + explanation questions
- Open questions

- Available during exam: 1 A4 page with anything you want on it
- No laptops, no internet

PhD positions available

- PhD position on privacy preserving machine learning
  - The privacy preserving machine learning group, headed by Dr. Mijung Park at the Max Planck Institute for Intelligent Systems and the University of Tübingen in Germany, is looking for a PhD student (m/f) for a project at the intersection of privacy and machine learning

- PhD position on in the intersection of computer vision, deep learning and biology
  - We are looking for outstanding EU students for funded PhD positions in the intersection of computer vision, deep learning and biology. The candidates are expected to have a degree in a relevant discipline before the fall 2019 including computer science, electrical engineering, physics but not limited to. Very good knowledge of machine learning and/or computer vision and programming experience (python, PyTorch, Tensorflow) are required. The project involves exploration of machine learning methodologies such as domain adaption, transfer and multi-task learning to the biology applications and will be supervised by a biology expert, Dr Sara Buonomo (http://buonomo.bio.ed.ac.uk/) and a machine learning expert, Dr Hakan Bilen (http://homepages.inf.ed.ac.uk/hbilen/) from the University of Edinburgh. The project does not require any background in biology but willing to learn basic biology. The starting date is September 2019 and more info can be found at http://web.inf.ed.ac.uk/cdt/biomedical-ai. If you are interested, please contact Hakan Bilen at hbilen@ed.ac.uk.

Lecture overview

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

Stochastic Gradient Descent

A Neural/Deep Network in a nutshell

1. The Neural Network

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

2. Learning by minimizing empirical error

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

3. Optimizing with Stochastic Gradient Descent based methods

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

What is a difference between Optimization and Machine Learning?

- The optimal learning solution is not the optimal machine learning solution necessarily
- They are practically equivalent
- Machine learning relates to optimization with some extras
- 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 learning solution is not the optimal machine learning solution necessarily
- 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 training, instead, the real goal and the trainable goal are quite often different (but related)
  - Even "optimal" parameters are not necessarily **optimal ← Overfitting …**
  - E.g., You want to recognize cars from bikes (0-1 problem) in unknown images, but you optimize the classification log probabilities (continuous) in known images

Empirical Risk Minimization

- Differently from pure optimization which operates on the training data points, we ideally should optimize for

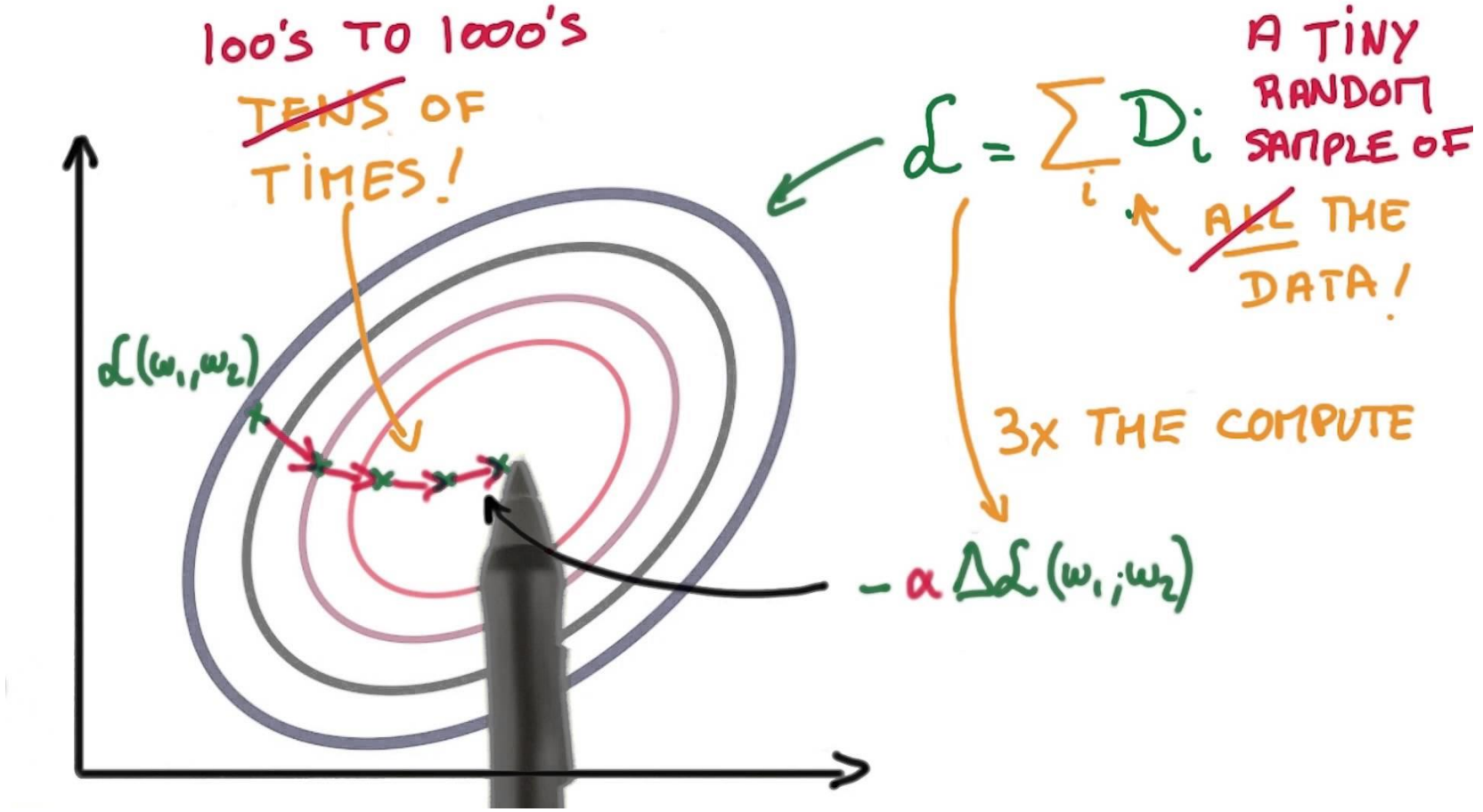
$$\min_{\mathrm{w}} \mathrm{E}_{x,\mathrm{y}\sim\hat{p}_{\mathrm{data}}}[\mathcal{L}(\mathrm{w};\mathrm{x},\mathrm{y})]$$

- Still, borrowing from optimization is the best way we can get satisfactory solutions to our problems by minimizing the empirical risk

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

- 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)



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

Disadvantages of Batch Gradient Descent?

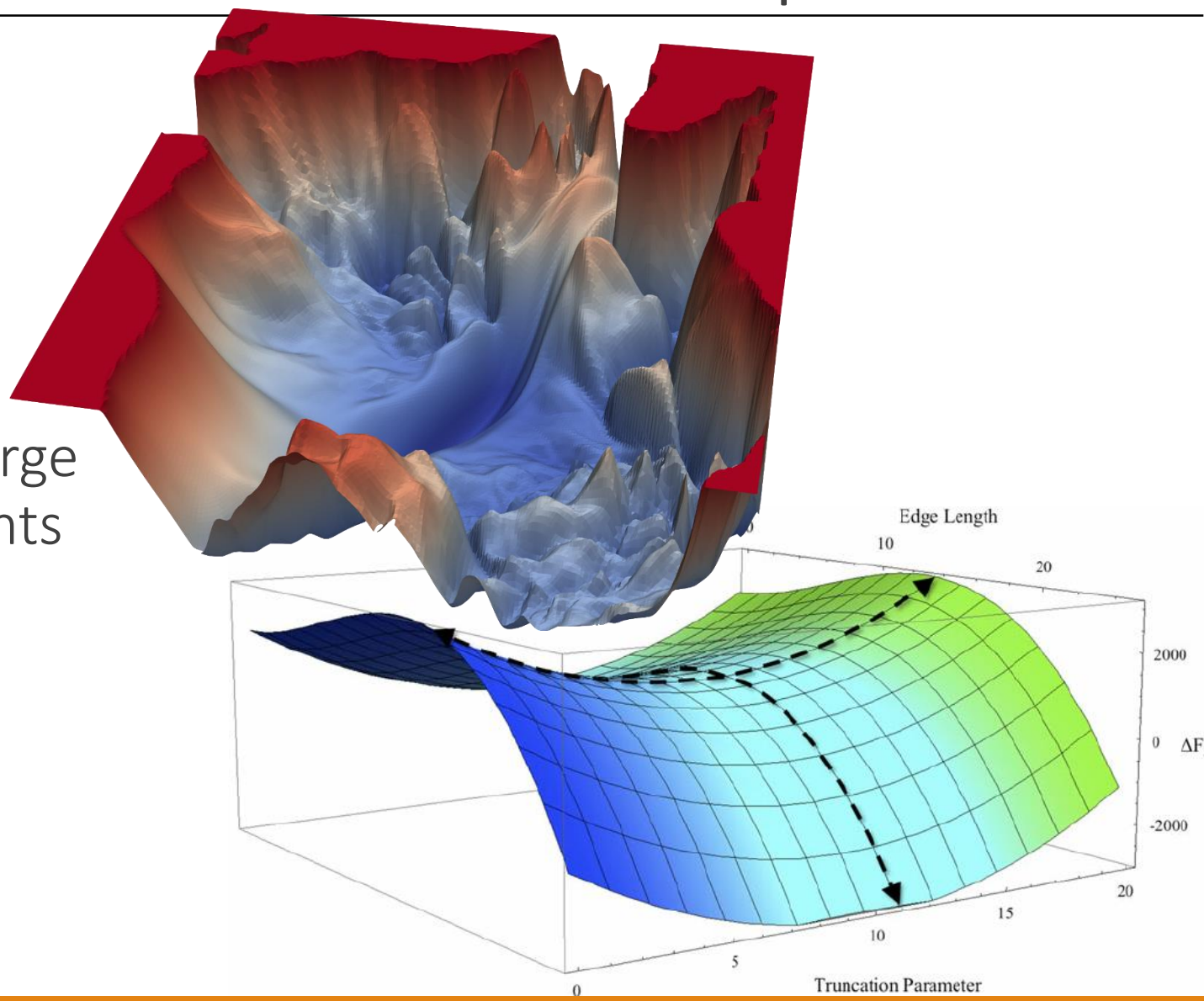
- Data is often too large to compute the full gradient, so slow trainng
- The loss surface is highly non-convex, so cannot compute the real gradient
- No real guarantee that leads to a good optimum
- No real guarantee that it will converge faster

Disadvantages of Batch Gradient Descent?

- Data is often too large to compute the full gradient, so slow trainng
- 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
  - non-quadratic
  - highly non-convex
  - very high-dimensional
- Datasets are typically really large to compute complete gradients
- No real guarantee that
  - the final solution will be good
  - we converge fast to final solution
  - or that there will be convergence



Stochastic Gradient Descent (SGD)

- The gradient equals an expectation $\mathrm{E}(\nabla_\theta \mathcal{L})$. In practice, we compute the mean from samples $\mathrm{E}(\nabla_\theta \mathcal{L}) = {}^{1}/_{m} \sum \nabla_\theta \mathcal{L}_i$.
- The standard error of this first approximation is given by ${}^{\sigma}/_{\sqrt{m}}$
  - So, the error drops sublinearly with m. To compute 2x more accurate gradients, we need 4x data points
  - And what's the point anyways, since our loss function is only a surrogate?
- Introduce a second approximation in computing the gradients
  - Stochastically sample "mini-training" sets ("mini-batches") from dataset D

$$B_j = sample(D)$$

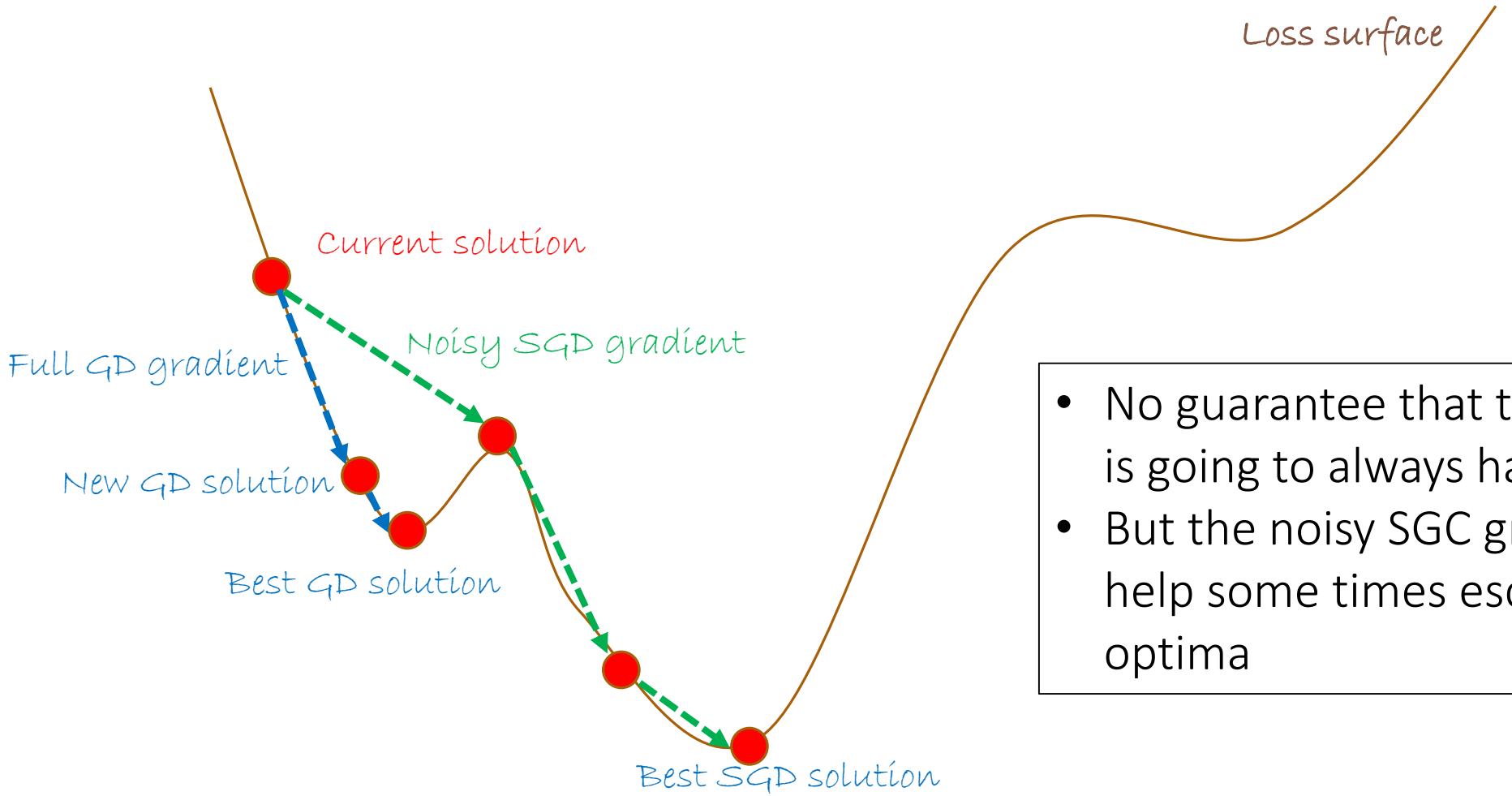
$$w_{t+1} = w_t - \frac{\eta_t}{|B_j|} \sum_{i \in B_j} \nabla_w \mathcal{L}_i$$

- When computed from continuous streams of data (training data only seen once) SGD minimizes generalization error
  - Intuitively, sampling continuously → we sample from the true data distribution: p_{data} not $\hat{p}_{\text{data}}$

Some advantages of SGD

- Randomness helps avoid overfitting solutions
- Random sampling allows to be much faster than Gradient Descent
- In practice accuracy is often better
- Mini-batch sampling is suitable for datasets that change over time
- Variance of gradients increases when batch size decreases

SGD is often better

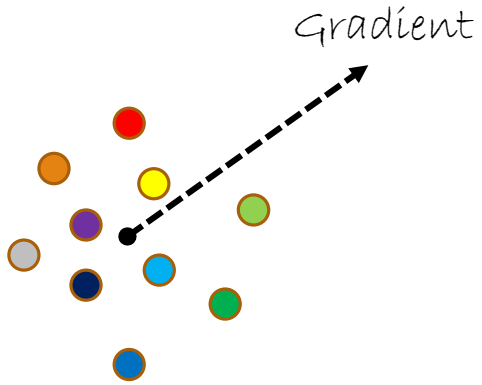


- No guarantee that this is what is going to always happen.
- But the noisy SGC gradients can help some times escaping local optima

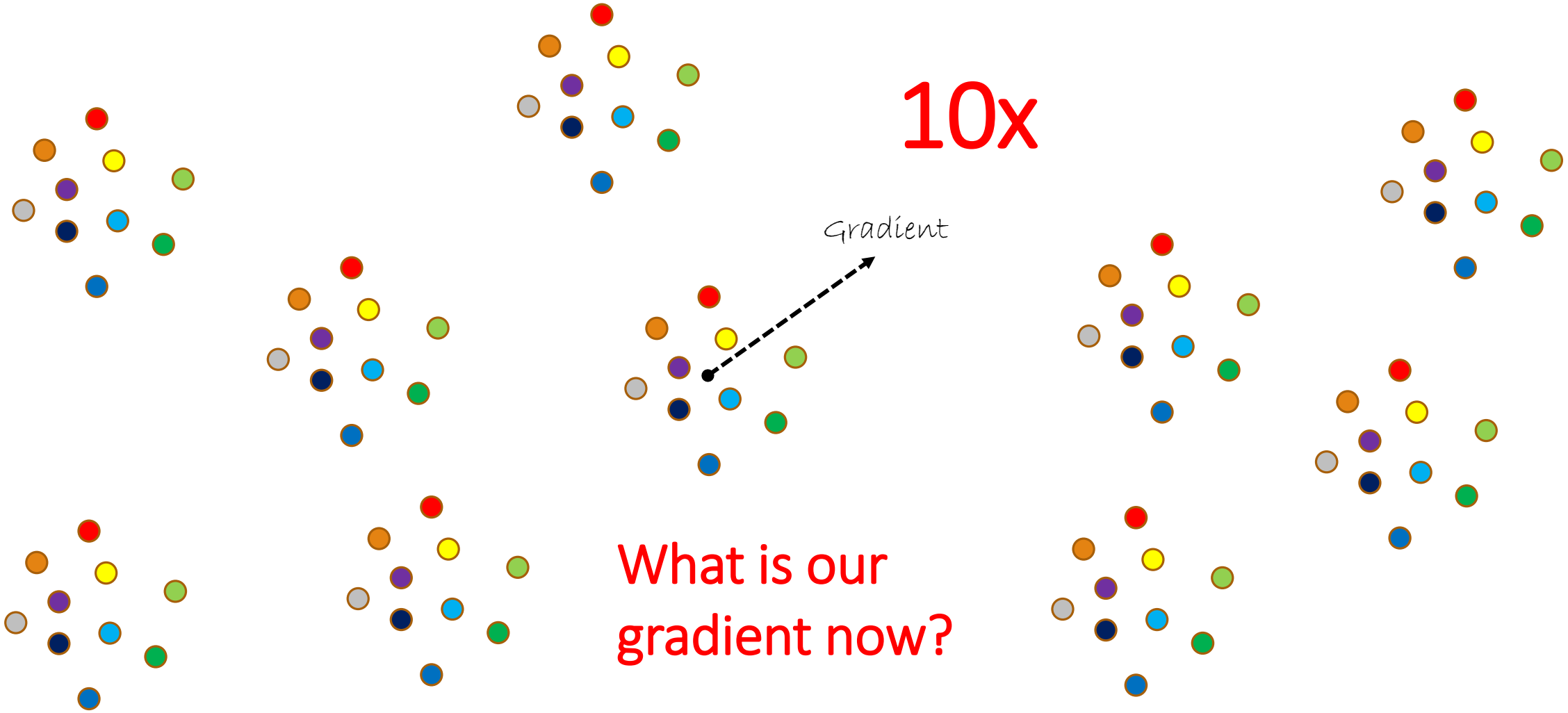
SGD is often better – In more details

- (A bit) Noisy gradients act as regularization
- Gradient Descent → Complete gradients
- 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
  - Test data will be no different than training data
  - Suitable for traditional optimization problems: "find optimal route"
  - But for ML we cannot make this assumption → test data are always different
- Stochastic gradients → sampled training data sample roughly representative gradients
  - Model does not overfit to the particular training samples

SGD is faster



SGD is faster



SGD is faster

- Of course in real situations data do not replicate
- But, with big data there are clusters of similar data
- Hence, the gradient is approximately alright
- Approximately alright is great in many cases actually

SGD for dynamically changed 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 many more “past” samples
- A properly implemented SGD track changes better [LeCun2002]

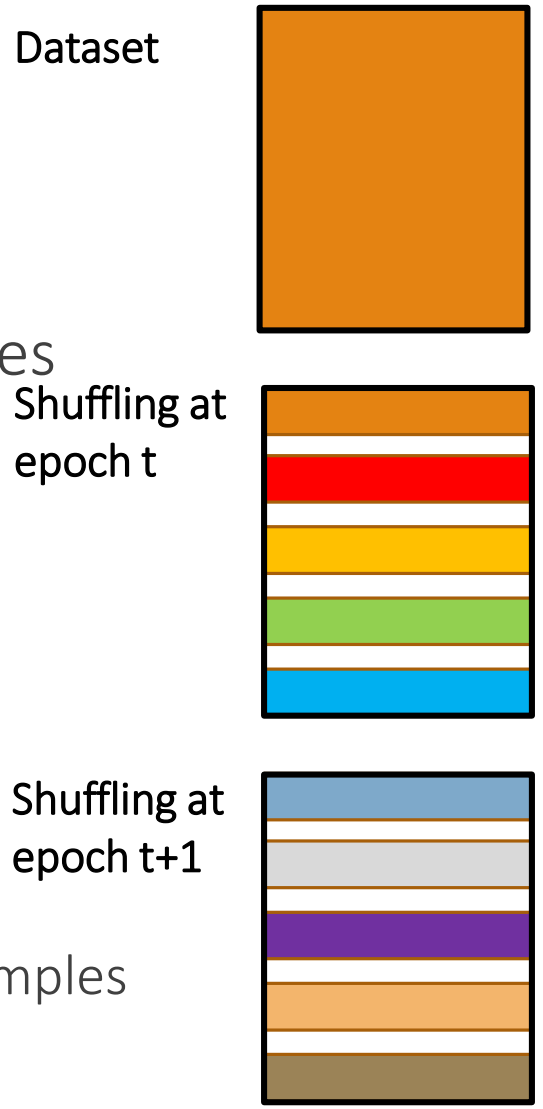
Popular last year
Kiki challenge

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
  - Each mini-batch is as class-divergent and rich as possible
  - New epoch → to be safe new batches & new, randomly shuffled examples



In practice

- SGD is preferred to Gradient Descent
- Training is orders faster
  - In real datasets Gradient Descent is not even realistic
- Solutions generalize better
  - More efficient → larger datasets
  - 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

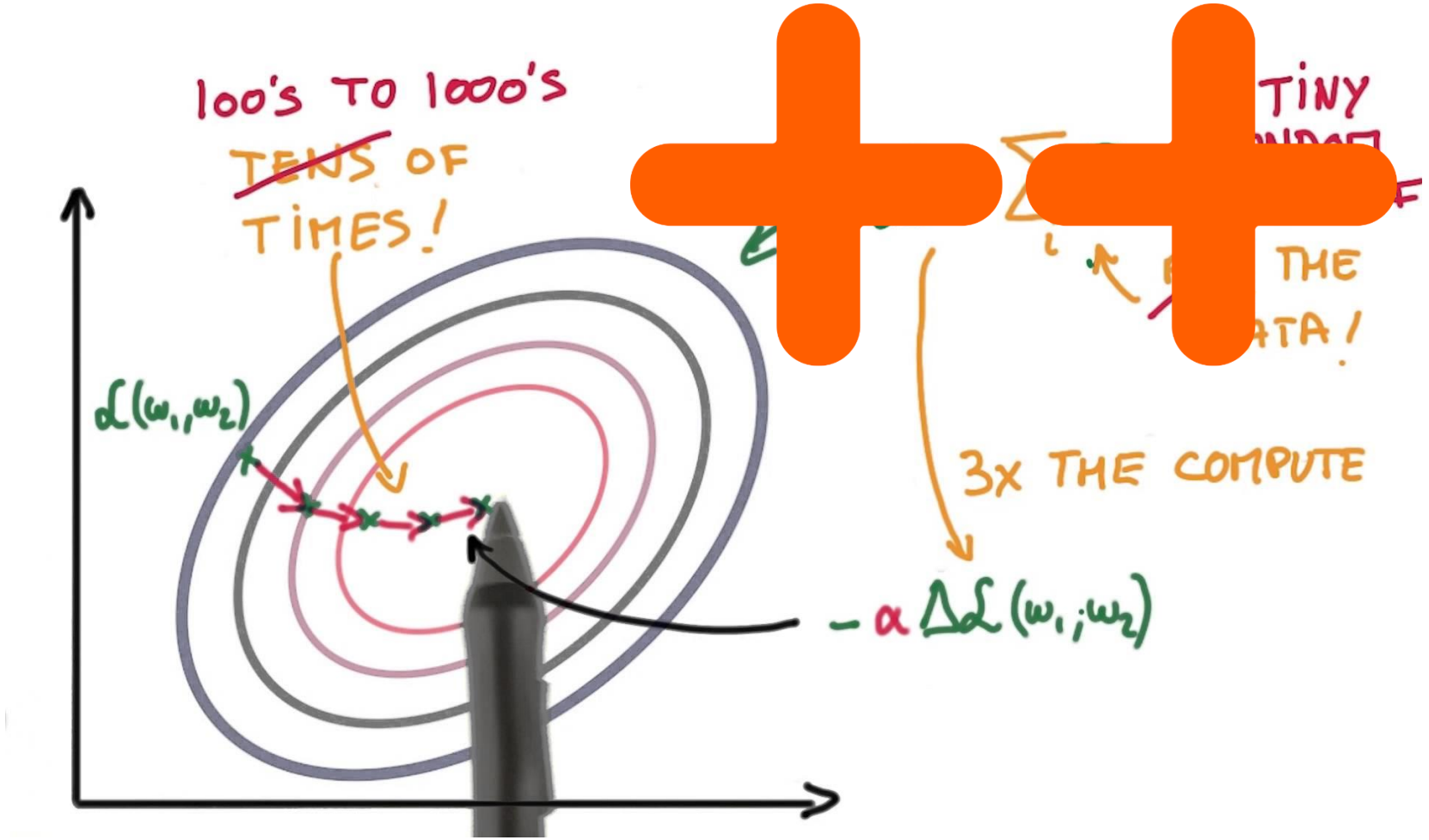
- Ill conditioning
  - Let's check the 2nd order Taylor dynamics of optimizing the cost function

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

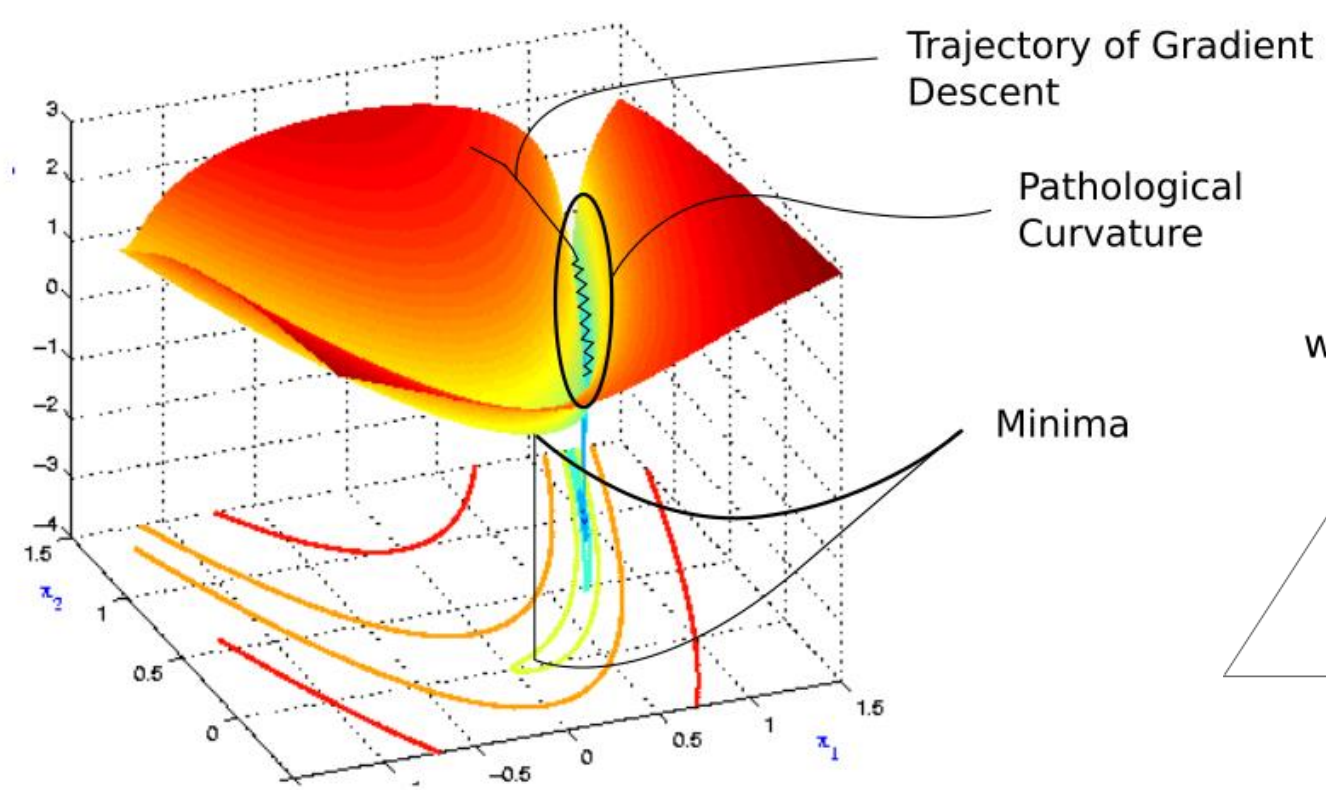
$$\mathcal{L}(\theta' - \varepsilon g) \approx \mathcal{L}(\theta) - \varepsilon g^{\mathrm{T}} g + \frac{1}{2} g^T H g$$

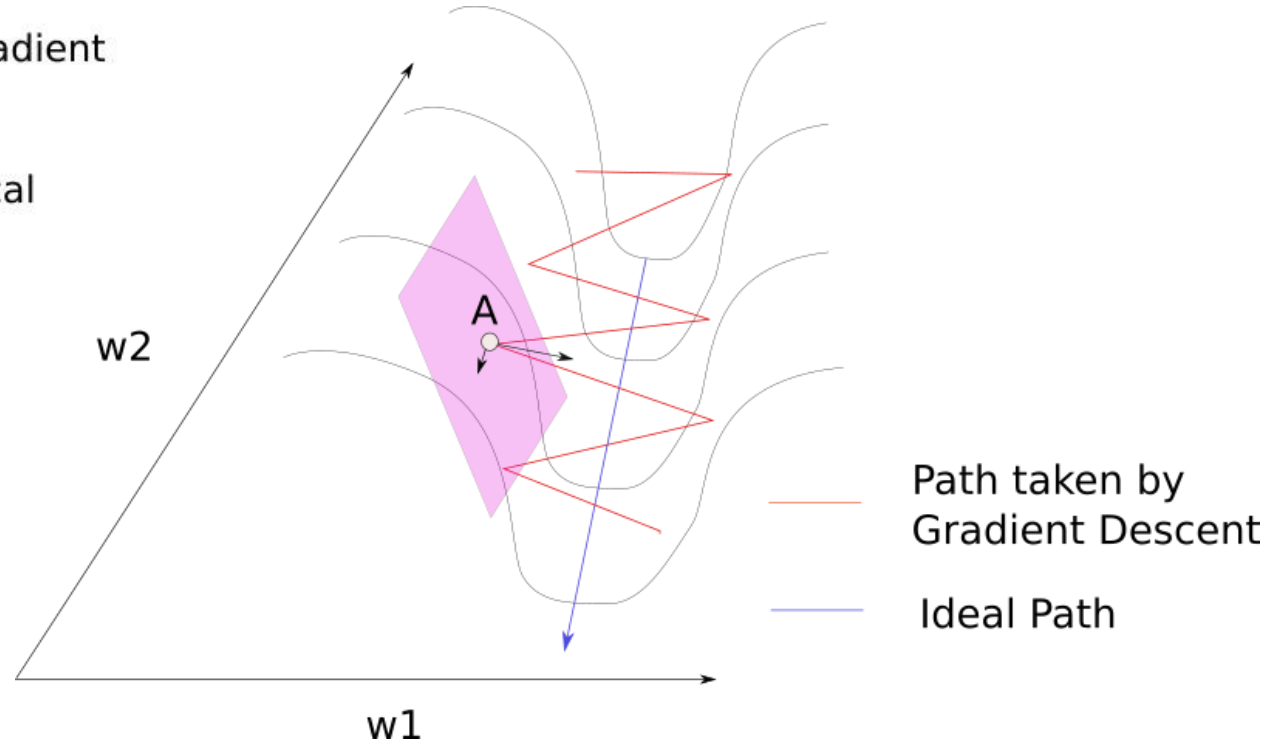
  - Even if the gradient g is strong, if $\frac{1}{2} g^T H g > \varepsilon g^{\mathrm{T}} g$ the cost will increase
- Local minima
  - Non-convex optimization produces lots of equivalent, local minima
- Plateaus
- Cliffs and exploding gradients
- Long-term dependencies

Advanced Optimizations



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

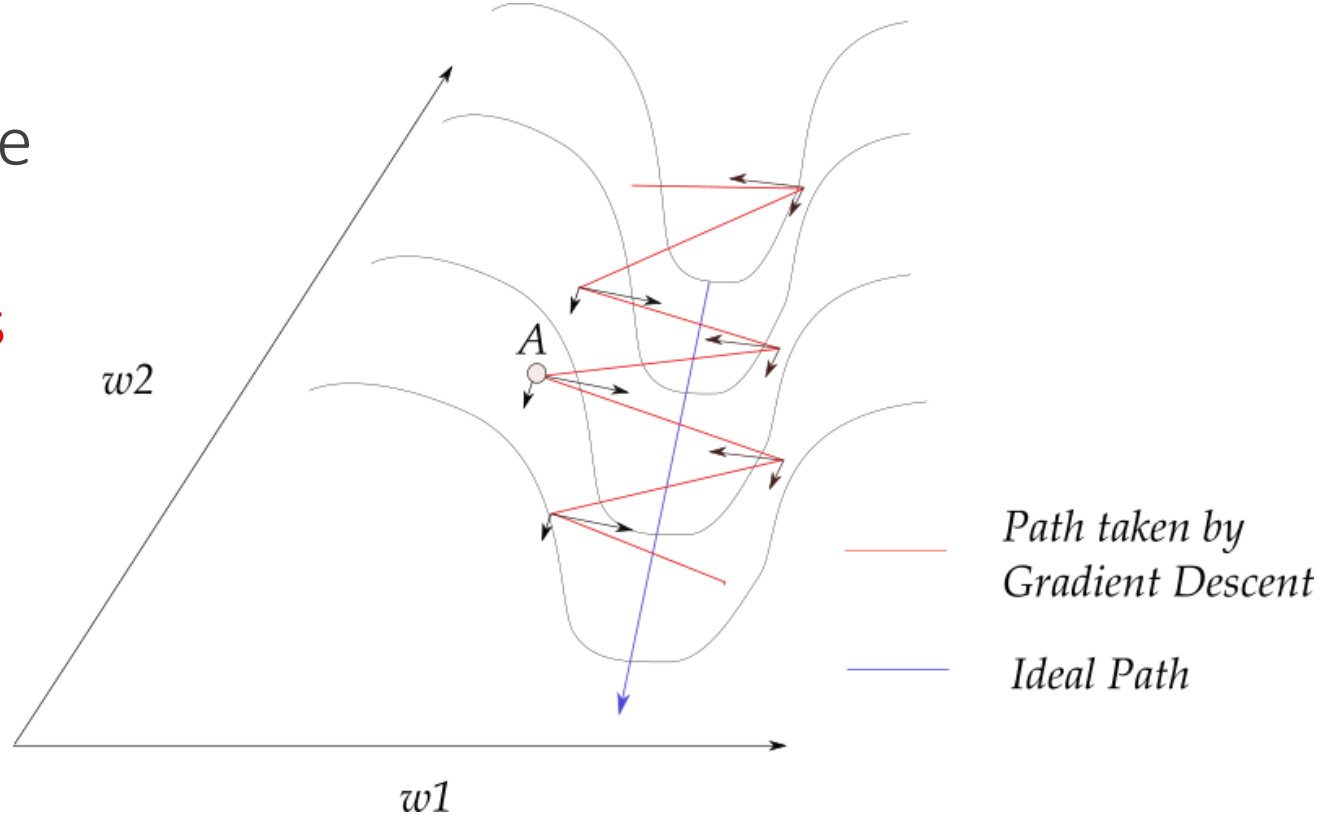
- Inverse of Hessian usually very expensive
  - Too many parameters
- Approximating the Hessian, e.g. with the L-BFGS algorithm
  - Keeps memory of gradients to approximate the inverse Hessian
- L-BFGS works alright with Gradient Descent. What about SGD?
- In practice SGD with some good momentum works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

Momentum

- Don't switch gradients all the time
- Maintain "momentum" from previous parameters → dampens oscillations

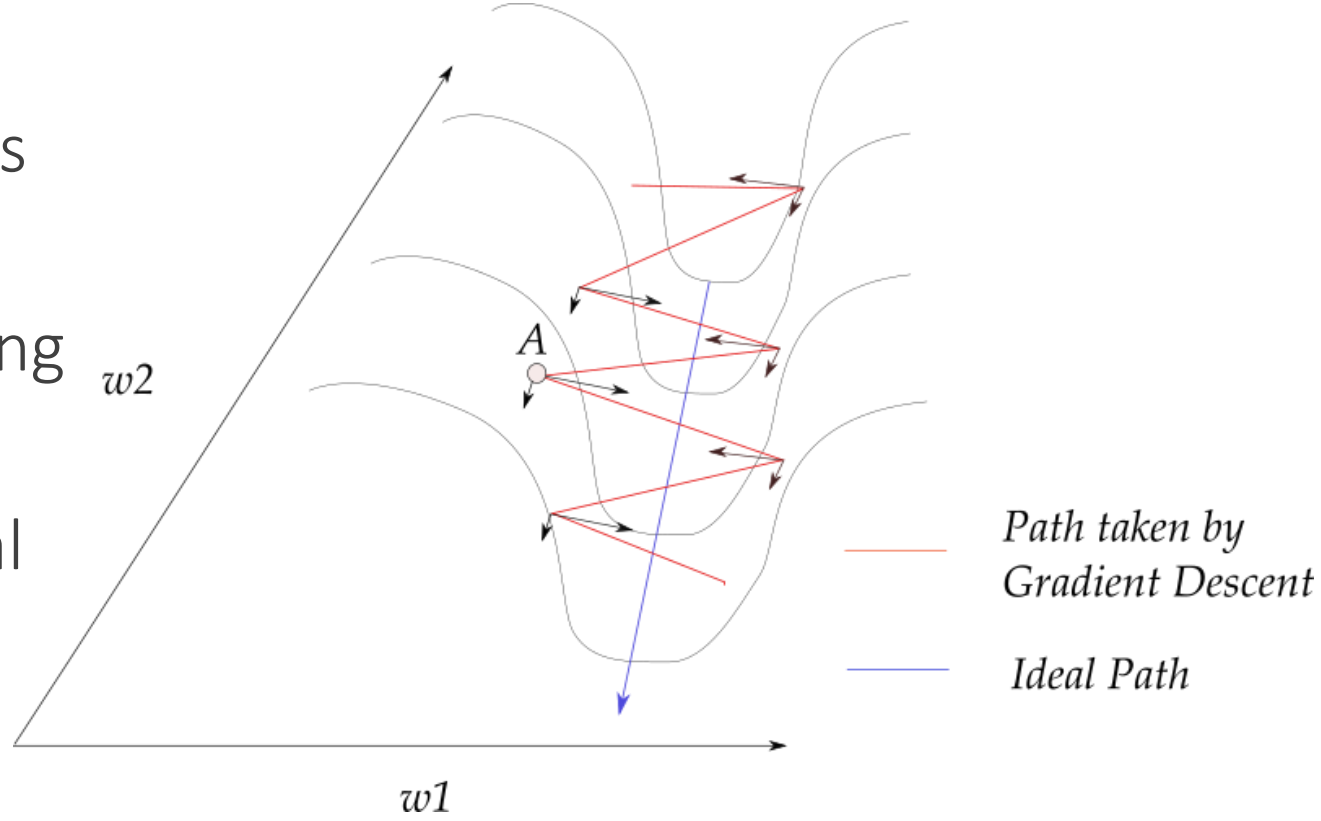
$$u_{t+1} = \gamma u_t - \eta_t g_t$$
$$w_{t+1} = w_t + u_{t+1}$$

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



Momentum

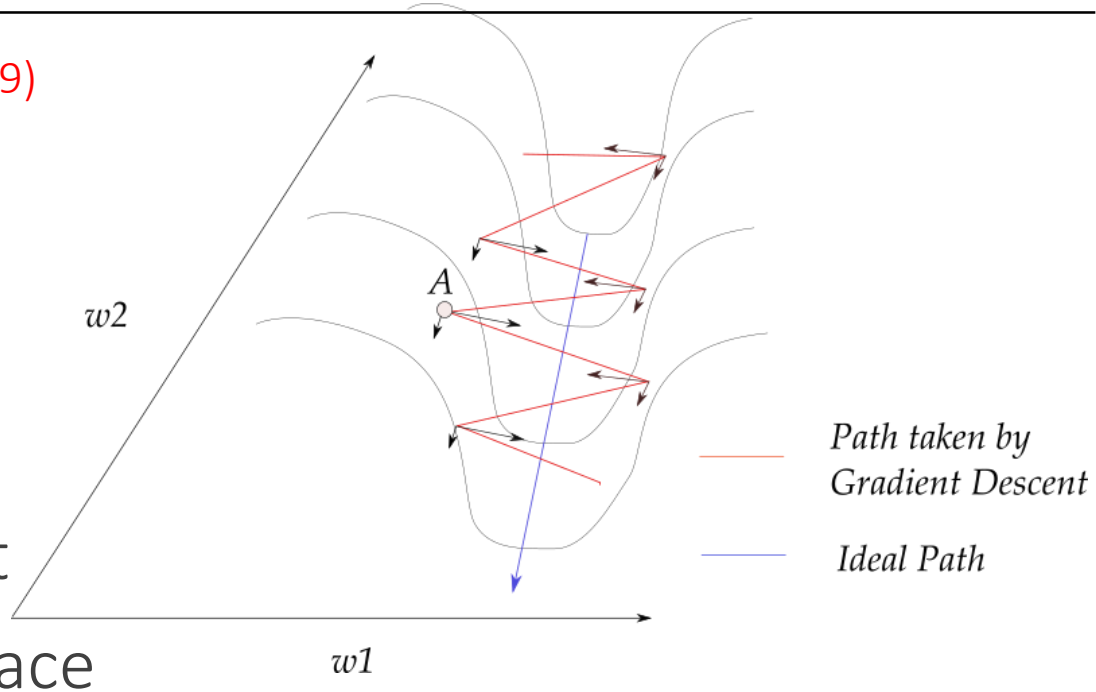
- The exponential averaging cancels out the oscillating gradients
- More robust gradients and learning → faster convergence
- Initialize $\gamma = \gamma_0 = 0.5$ and anneal to $\gamma_\infty = 0.9$

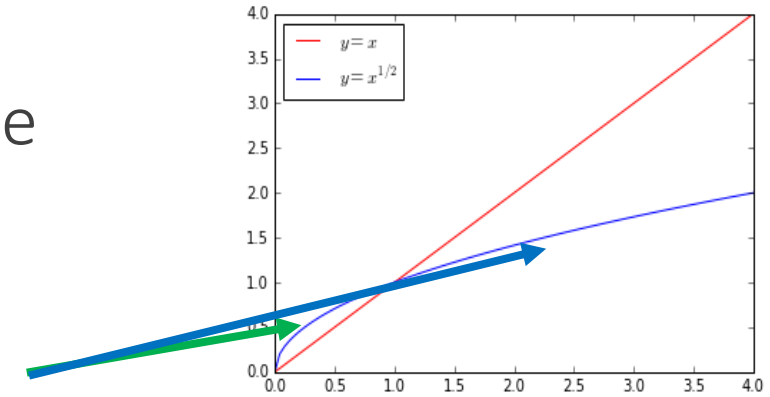


RMSprop

Decay hyper-parameter (Usually 0.9)

- Schedule
  - $r_t = \alpha r_{t-1} + (1-\alpha) \odot g_t^2 \implies$
  - $u_t = -\frac{\eta}{\sqrt{r_t}+\varepsilon} \odot g_t$
  - $w_{t+1} = w_t + \eta_t u_t$
- Squaring and adding → no cancelling out
- **Large gradients**, e.g. too "noisy" loss surface
  - Updates are tamed
- **Small gradients**, e.g. stuck in flat loss surface ravine
  - 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

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

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

Adagrad [Duchi2011]

- Schedule
  - $r_j = \sum_\tau (\nabla_\theta \mathcal{L}_j)^2 \implies w_{t+1} = w_t - \eta_t \frac{g_t}{\sqrt{r}+\varepsilon}$
  - ε is a small number to avoid division with 0
  - Gradients become gradually smaller and smaller

Nesterov Momentum [Sutskever2013]

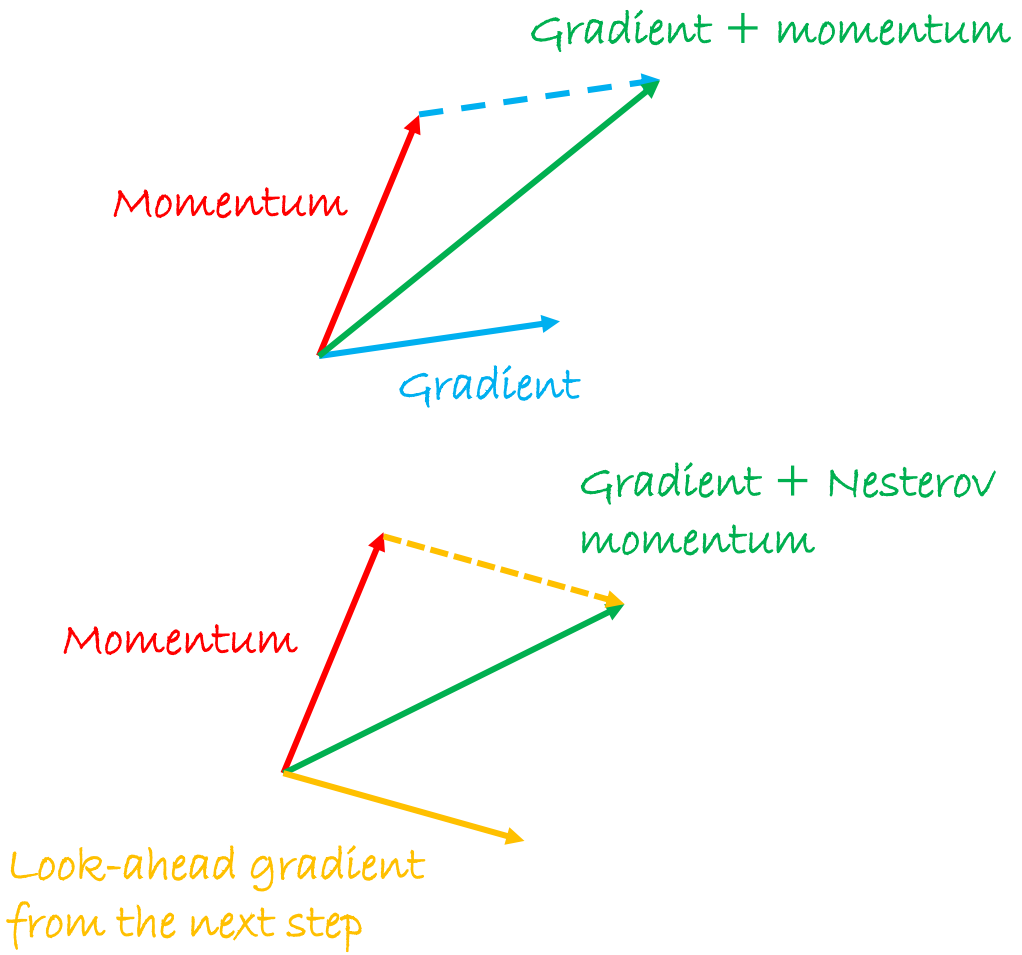
- Use the future gradient instead of the current gradient

$$w_{t+0.5} = w_t + \gamma u_\tau$$

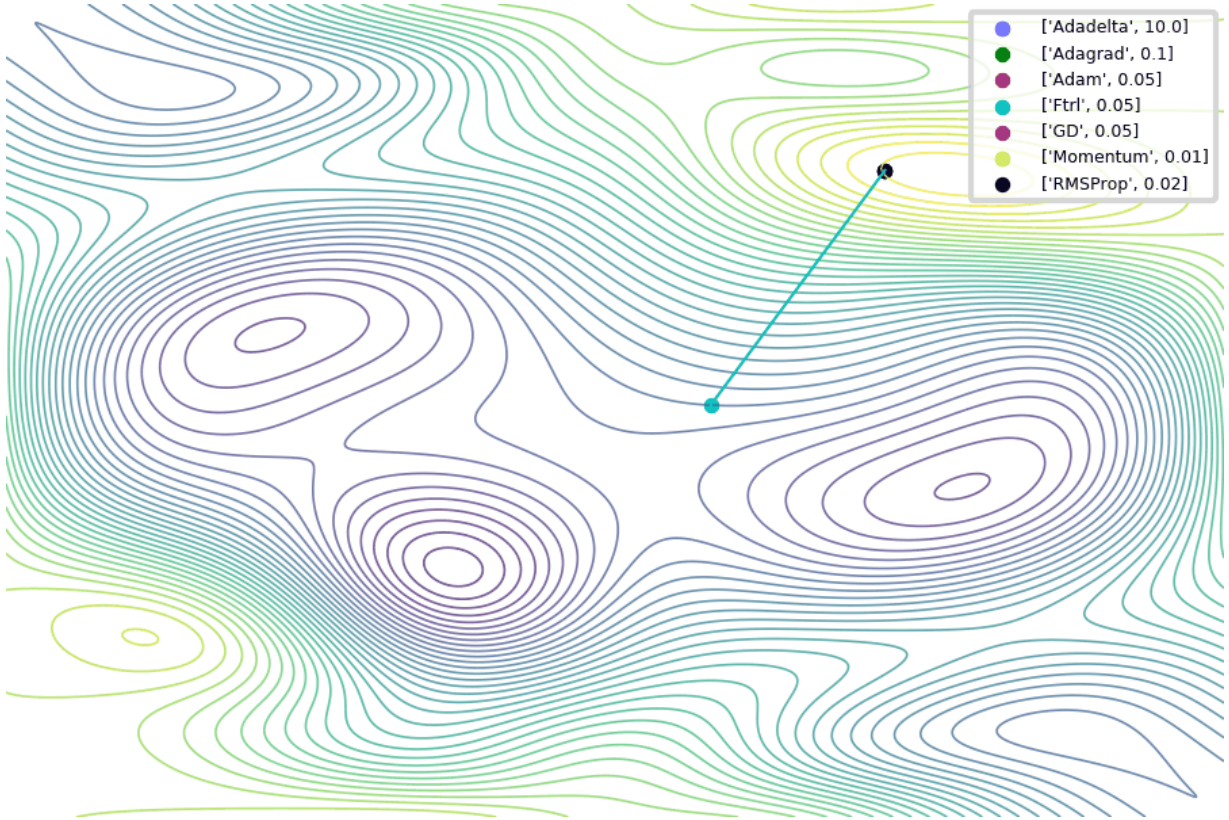
$$w_t = \gamma u_\tau - \eta_t \nabla_{w_{t+0.5}} \mathcal{L}$$

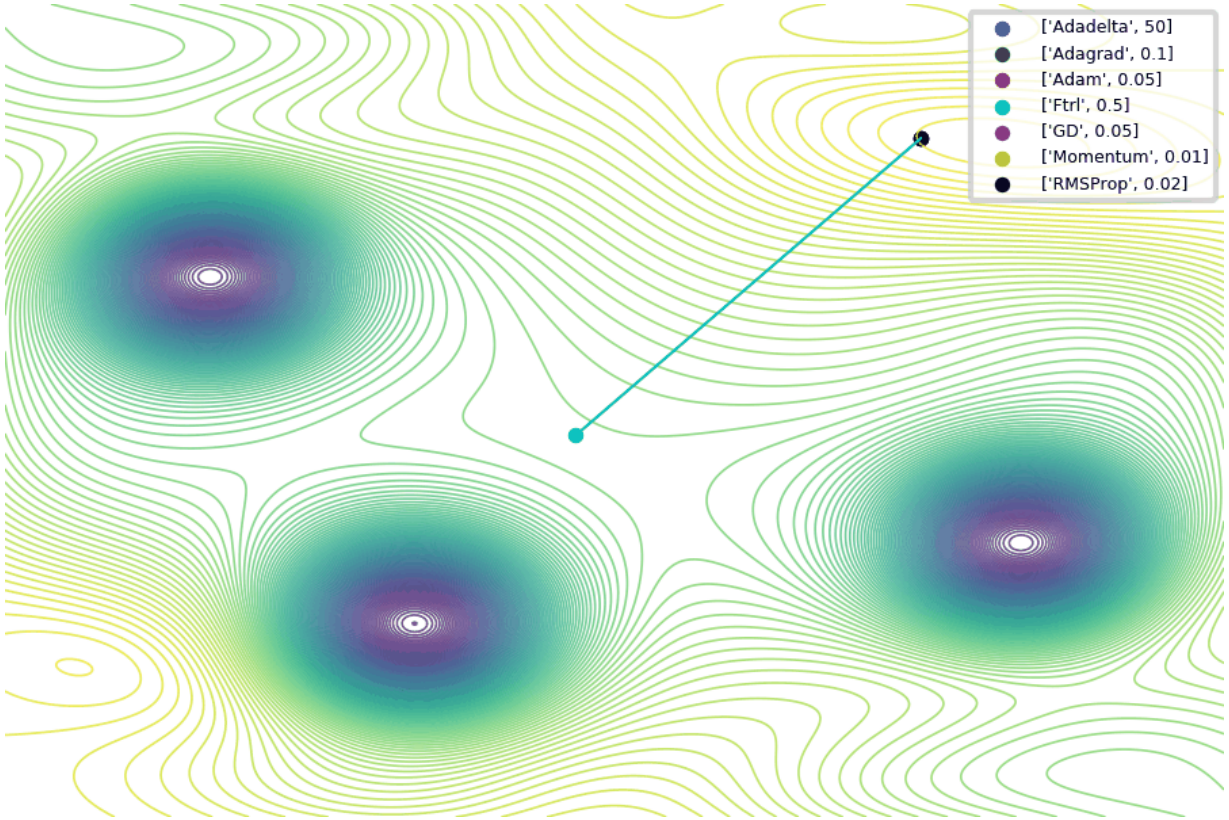
$$w_{t+1} = w_t + u_{t+1}$$

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



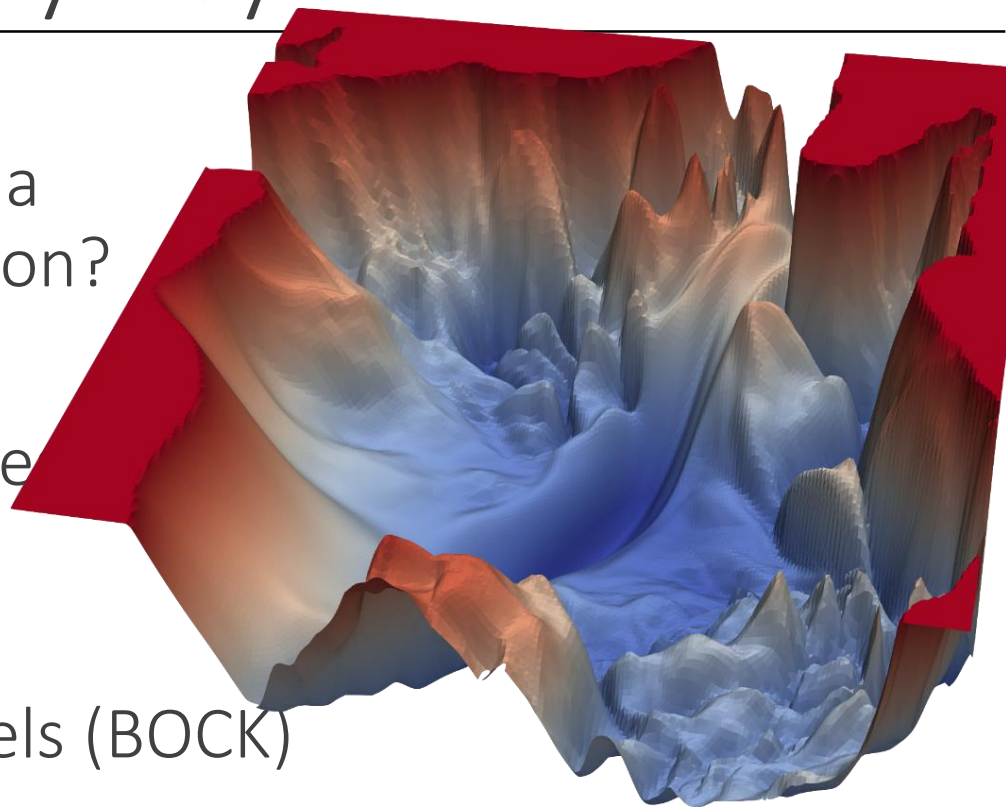
Visual overview





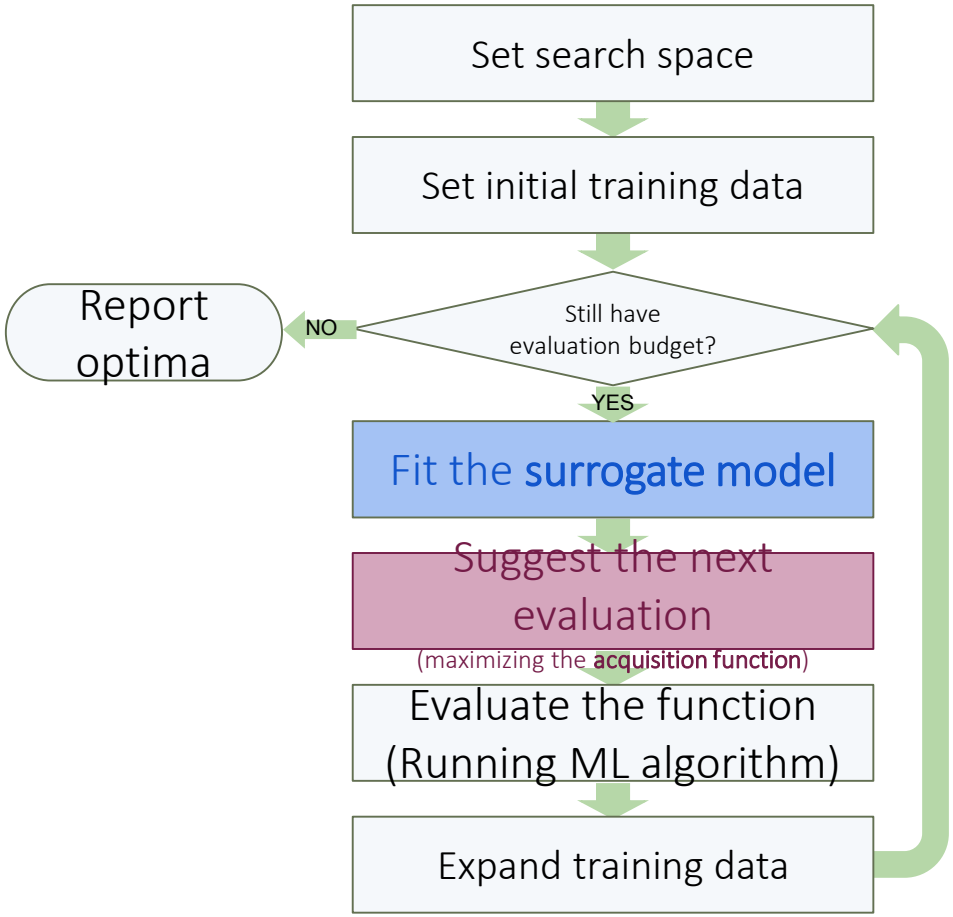
Picture credit: Jaewan Yun

Is Gradient-based Learning the only way?



- Maybe with a loss with so many local minima we better go with a gradient-free optimization?
- Bayesian Optimization tries to find a good solution with educated trial and error guesse
  - Usually it doesn't work on very high dimensional spaces, e.g. more than 20 or 50
- Bayesian Optimization with Cylindrical Kernels (BOCK)
  - C. Oh, E. Gavves, M. Welling, ICML 2018
  - Scales gracefully to 500-1000 dimensions
  - Long way from the millions/billions of parameters in Deep Nets, but there are encouraging signs

Bayesian Optimization

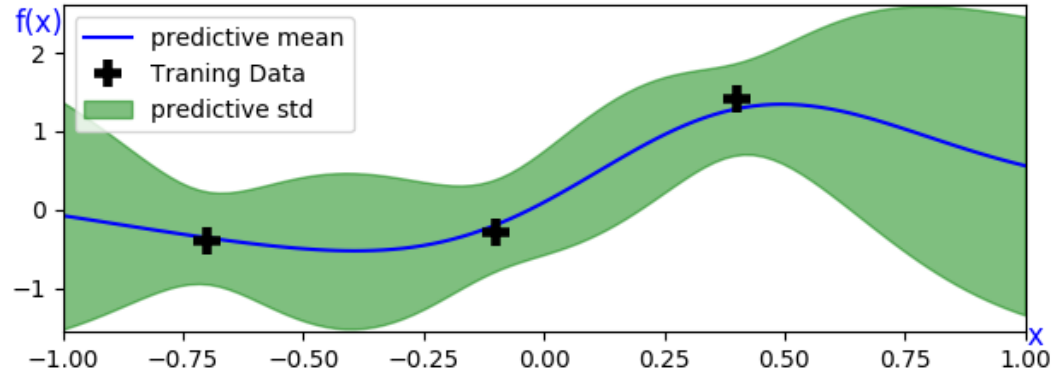


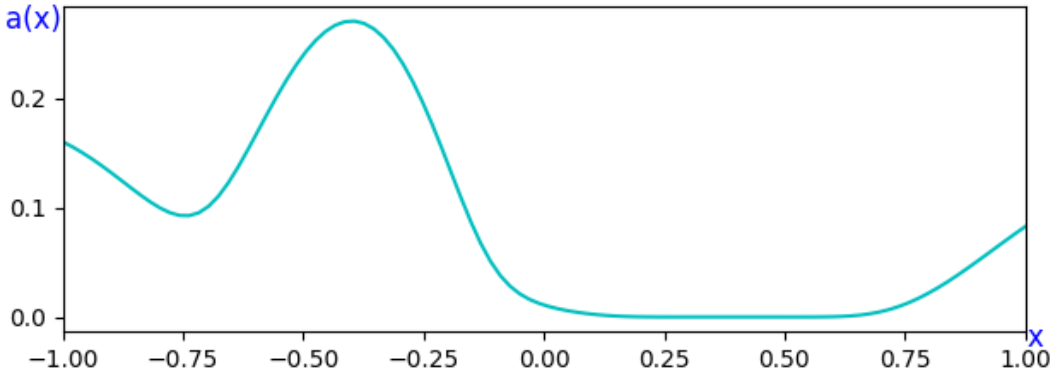
training data
{([x_1, x_2, ...], f(x_1, x_2, ...)}

predictive mean
predictive variance

new input data
(new x_1, x_2, ...)

new output data
(f(x_1, x_2, ...))

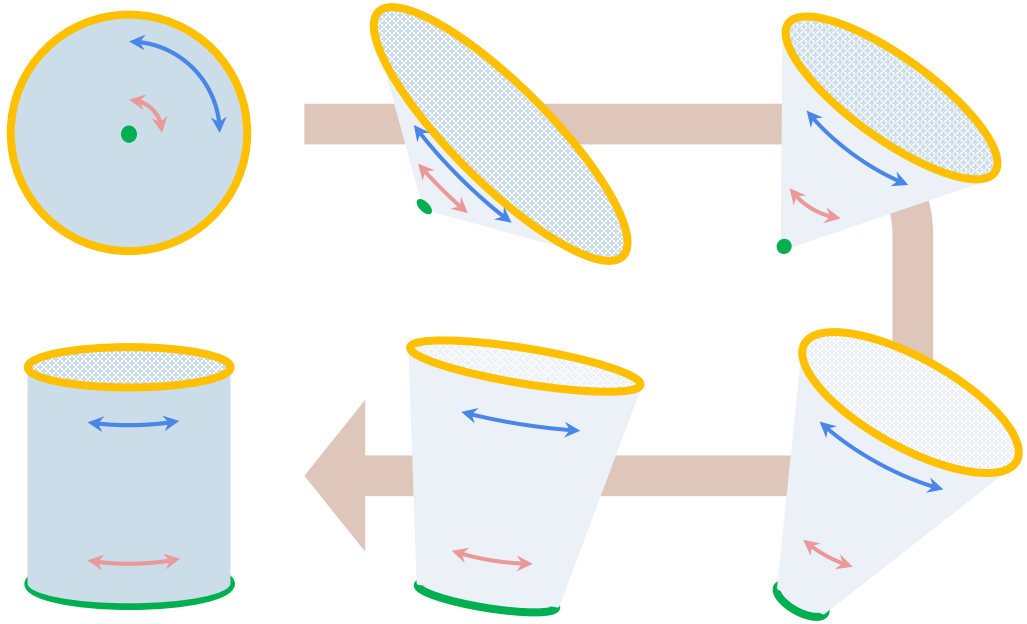


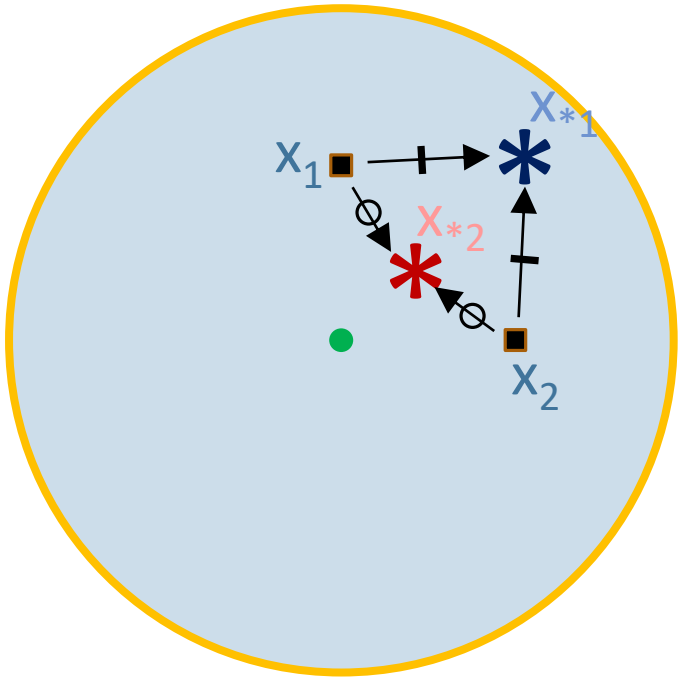


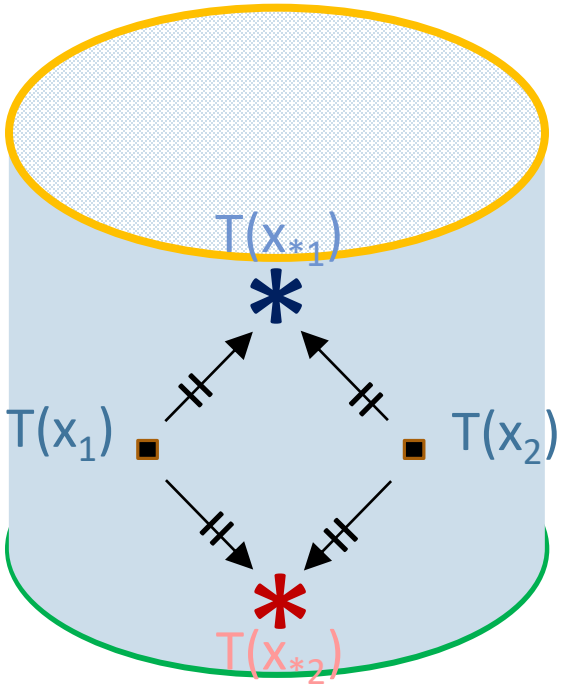
training data U { (new input, new output) }

BOCK for training a neural network layer

- Good, regularized solutions are often near the center of the search space
- But in high-dimensional spaces the density is towards the boundaries
- Solution: Transform the geometry of the search space!

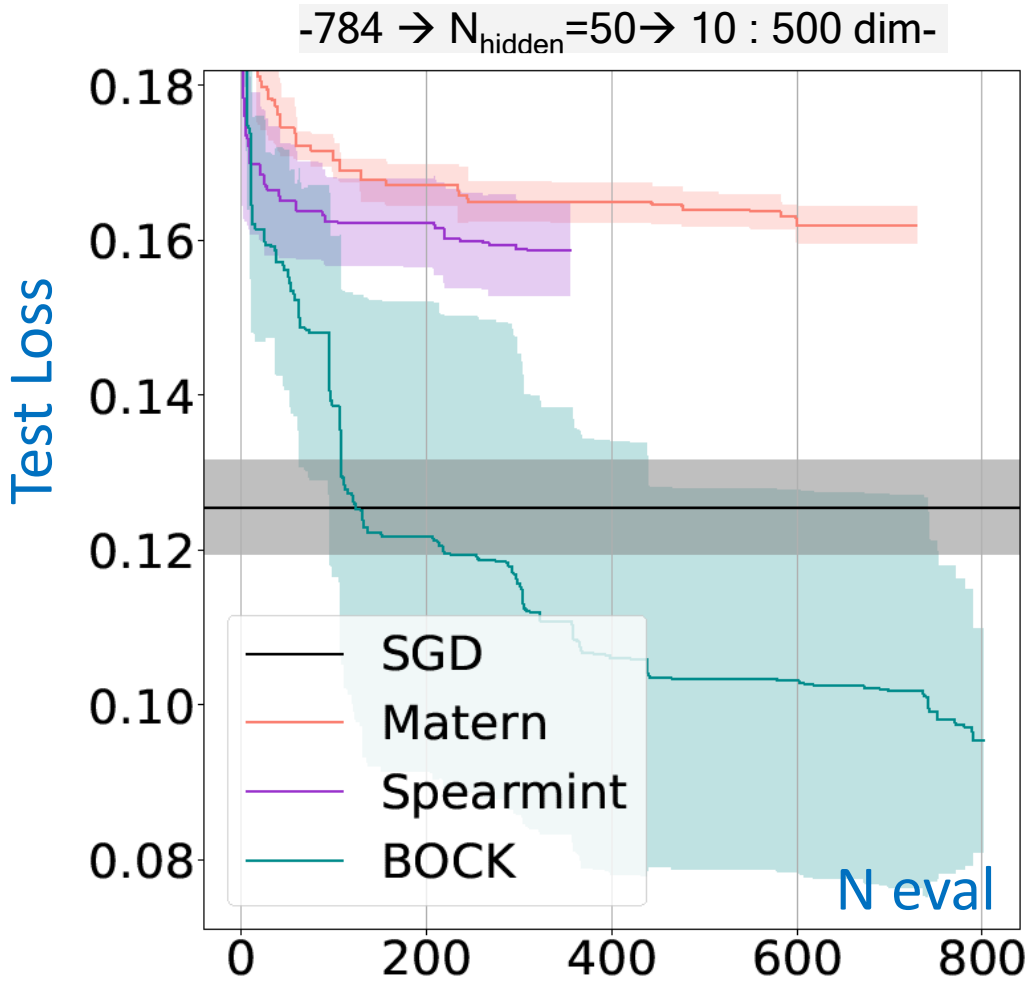




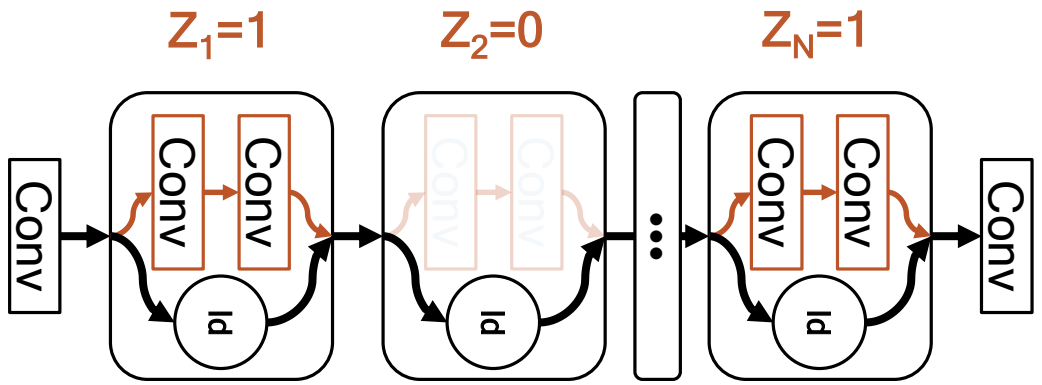


BOCK for hyper/optimizing neural networks

Training even a medium size neural network layer



Hyper-optimizing ResNet Depth



	Test Acc.	Valid Acc.	Exp. Depth
ResNet-110	72.98±0.43	73.03±0.36	110.00
SDResNet-110 Linear	74.90±0.15	75.06±0.04	82.50
SDResNet-110 BOCK	75.06±0.19	75.21±0.05	74.51±1.22

Optimizing hyperparameters in Deep Nets

- There are some automated tools
  - Trust them with a grain of salt
  - Deep Nets are too large and complex
- Usually based on intuition
  - Then manual grid search
- Optimizing Deep Network Architecture
  - DARTS, Liu et al., arXiv 2018
  - Practical Bayesian Optimization of Machine Learning Algorithms, Snoek et al., NIPS 2012
  - https://www.ml4aad.org/automl/literature-on-neural-architecture-search/

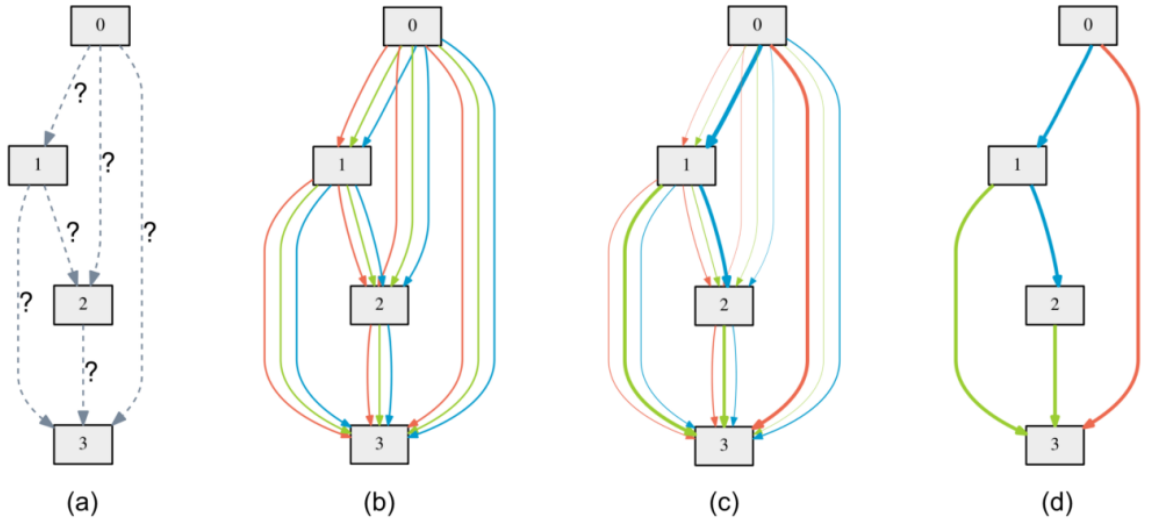
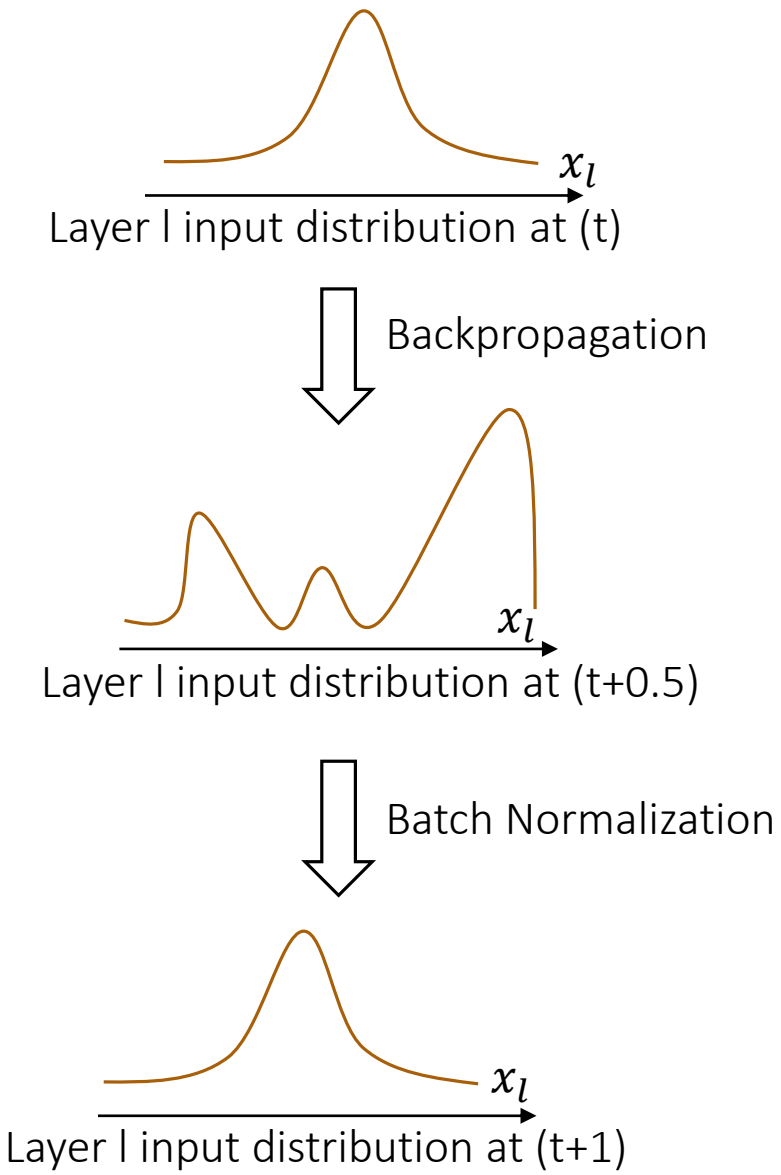


Figure 1: An overview of DARTS: (a) Operations on the edges are initially unknown. (b) Continuous relaxation of the search space by placing a mixture of candidate operations on each edge. (c) Joint optimization of the mixing probabilities and the network weights by solving a bilevel optimization problem. (d) Inducing the final architecture from the learned mixing probabilities.

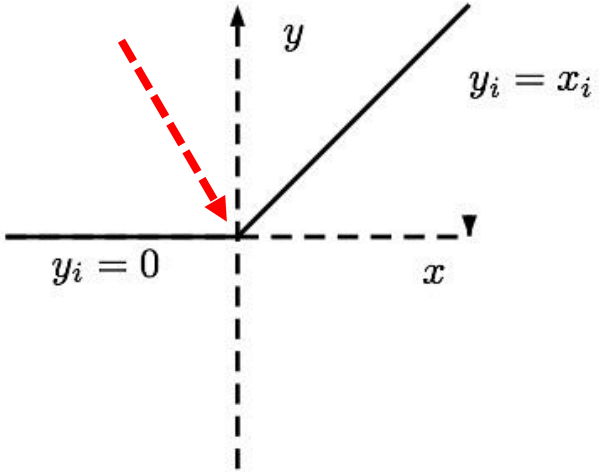
Input normalization



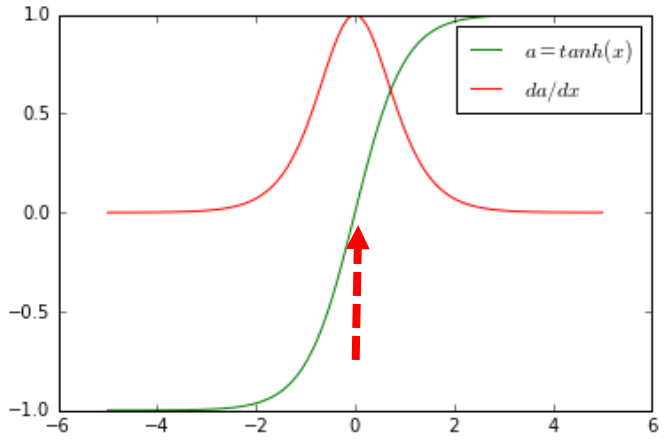
Data pre-processing

- Center data to be roughly 0
  - Activation functions usually “centered” around 0
  - Convergence usually faster
  - Otherwise **maybe** bias on gradient direction → might slow down learning

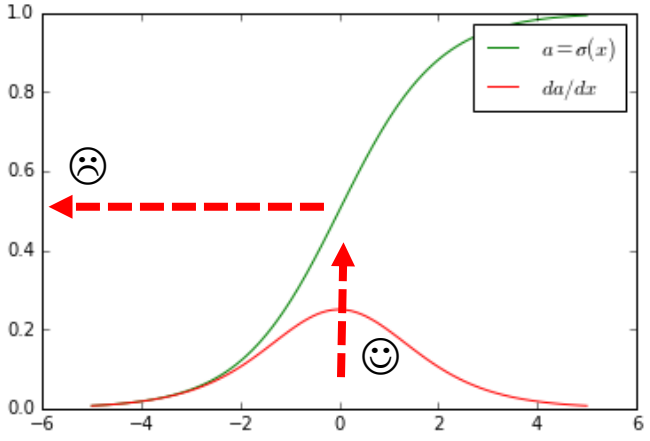
ReLU ☺



$\tanh(x)$ ☺



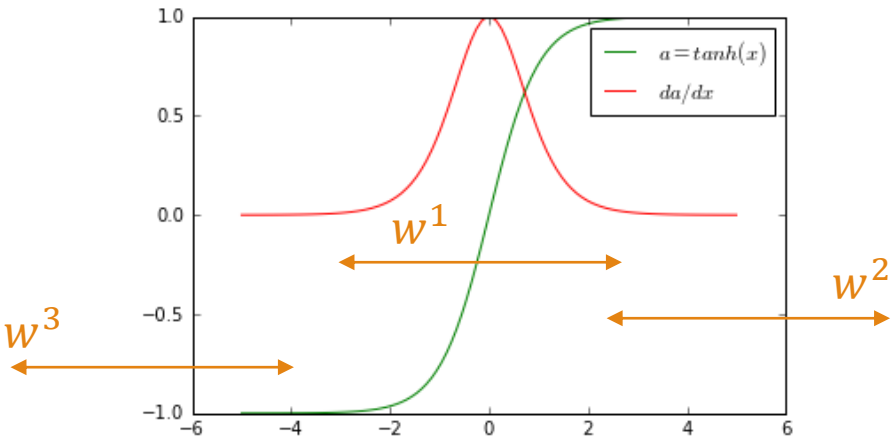
$\sigma(x)$ ☹



Data pre-processing

- Scale input variables to have similar diagonal covariances $c_i = \sum_j (x_i^{(j)})^2$
  - Similar covariances → more balanced rate of learning for different weights
  - Rescaling to 1 is a good choice, unless some dimensions are less important

$x = [x_1, x_2, x_3]^T, w = [w_1, w_2, w_3]^T, a = \tanh(w^{\mathrm{T}} x)$



x_1, x_2, x_3 → much different covariances

↓

Generated gradients $\left.\frac{\mathrm{d}\mathcal{L}}{d\theta}\right|_{x_1,x_2,x_3}$: much different

↓

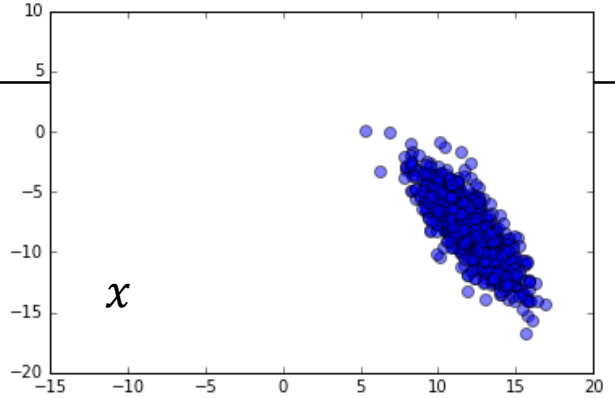
Gradient update harder: $w_{t+1} = w_t - \eta_t \begin{bmatrix} d\mathcal{L}/dw_1 \\ d\mathcal{L}/dw_2 \\ d\mathcal{L}/dw_3 \end{bmatrix}$

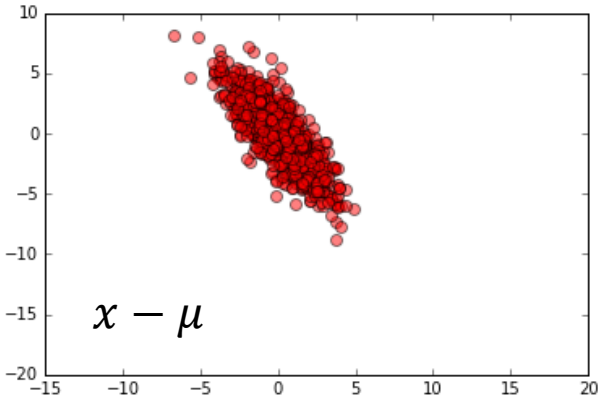
Data pre-processing

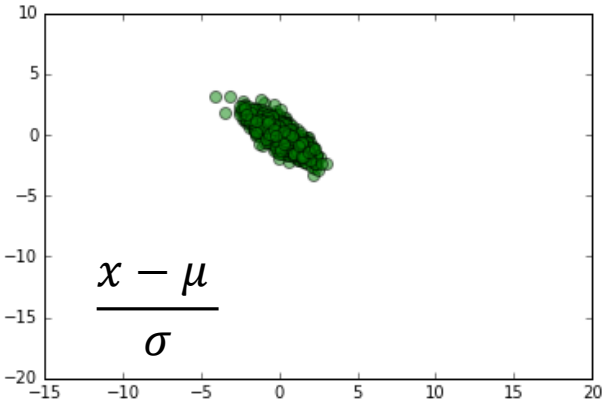
- Input variables should be as decorrelated as possible
  - Input variables are “more independent”
  - Model is forced to find non-trivial correlations between inputs
  - Decorrelated inputs → Better optimization
- Extreme case
  - extreme correlation (linear dependency) might cause problems [CAUTION]
- Obviously decorrelating inputs is not good when inputs are by definition correlated, like when in sequences

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

- Input variables follow a Gaussian distribution (roughly)
- In practice:
  - from training set compute mean and standard deviation
  - Then subtract the mean from training samples
  - Then divide the result by the standard deviation





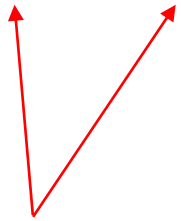


Even simpler: Centering the input

- When input dimensions have similar ranges …
- … and with the right non-linearity …
- … centering might be enough
  - e.g. in images all dimensions are pixels
  - All pixels have more or less the same ranges
- Just make sure images have mean 0 ($\mu = 0$)

Batch normalization – The algorithm

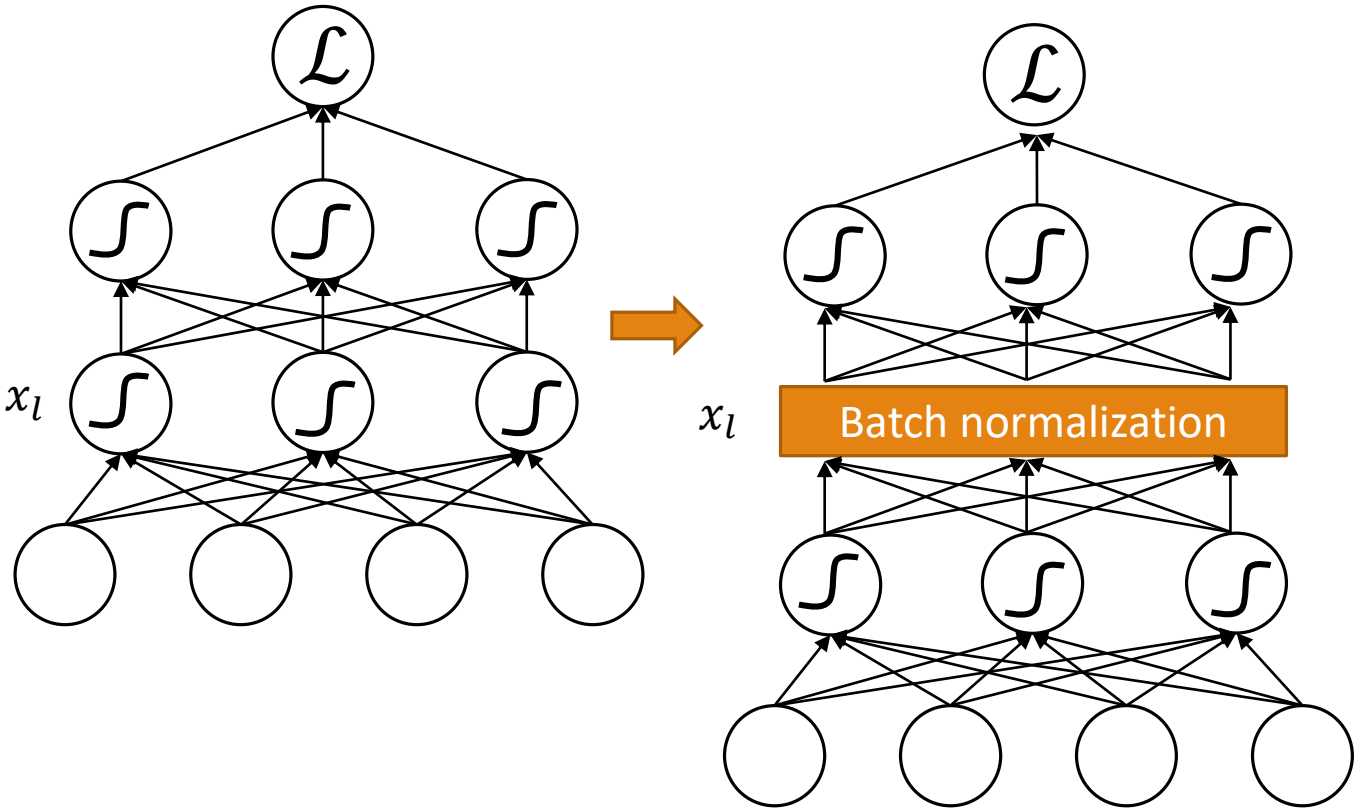
- $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ [compute mini-batch mean]
- $\sigma_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ [compute mini-batch variance]
- $\widehat{x_i} \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$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

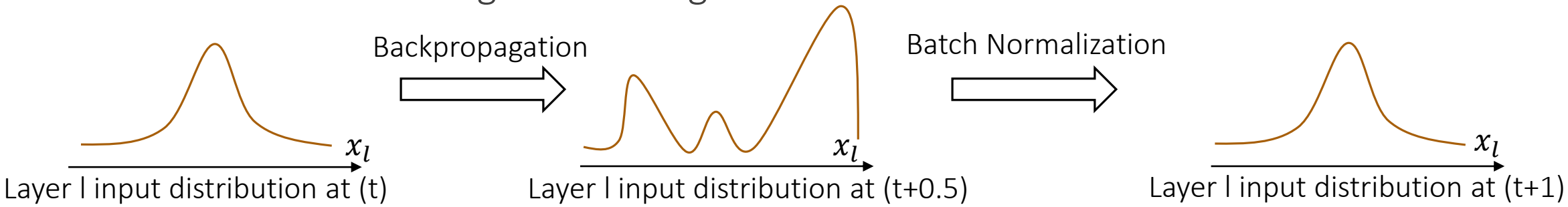


Trainable parameters

Batch normalization [Ioffe2015]

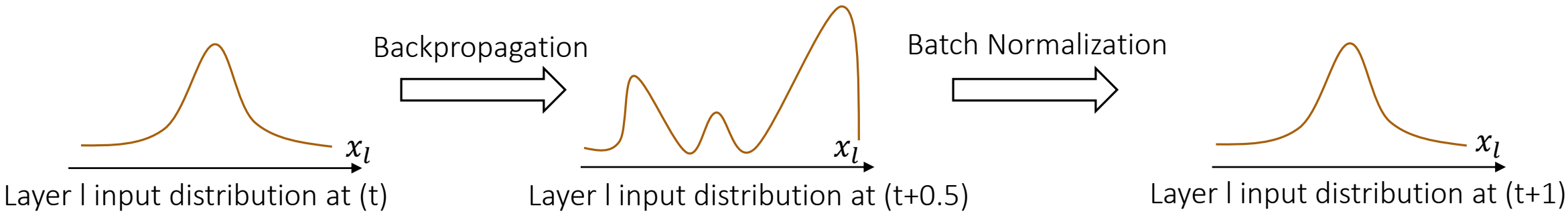
- Weights change → the distribution of the layer inputs changes per round
- Normalize the layer inputs with batch normalization
  - Roughly speaking, normalize x_l to $N(0,1)$, then rescale
  - Rescaling is so that the model decides itself the scaling and shifting



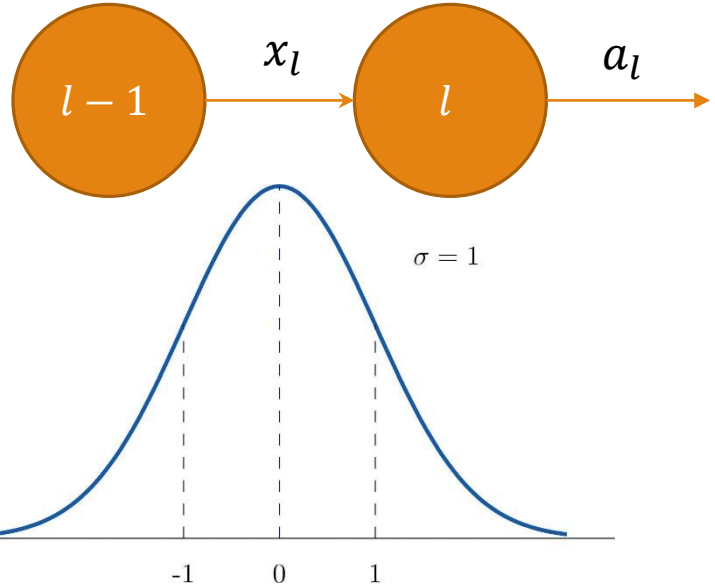


Batch normalization – Intuition I

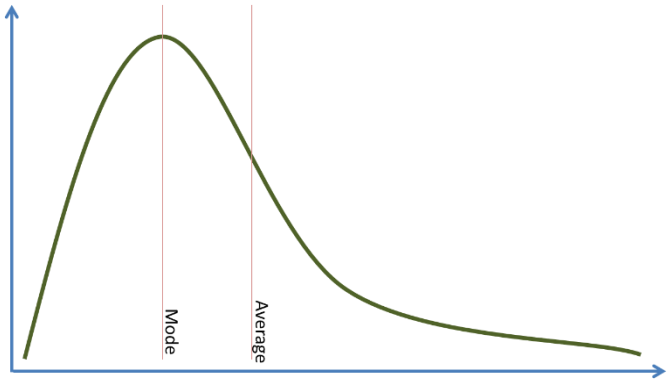
- Covariate shift
  - At each step a layer must not only adapt the weights to fit better the data
  - It must also adapt to the change of its input distribution, as its input is itself the result of another layer that changes over steps
- The distribution fed to the layers of a network should be somewhat: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



An intuitive example

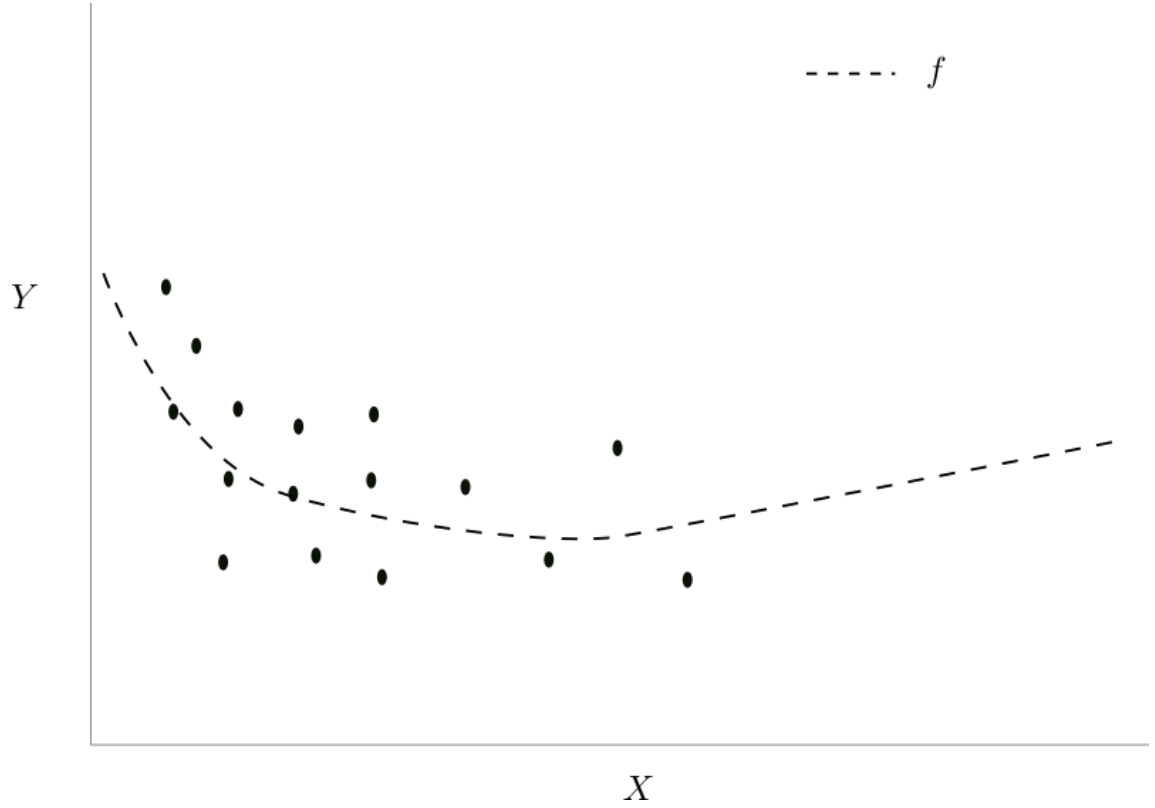


Distiibution of x_l #1



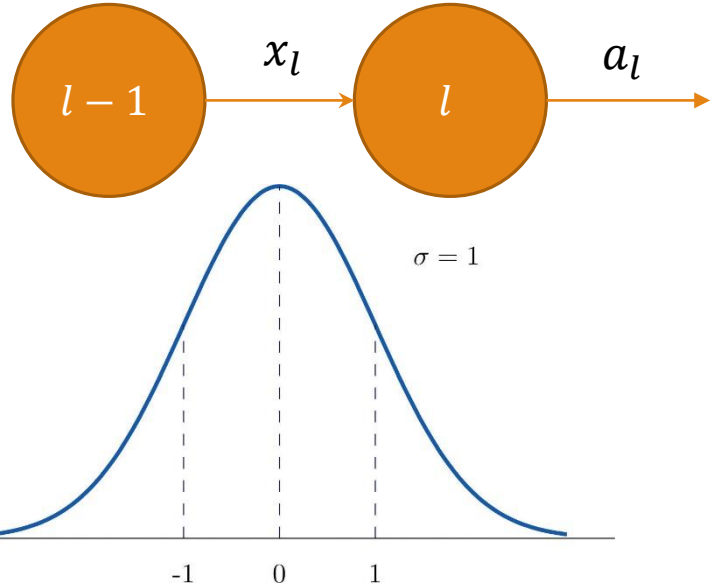
Distiibution of x_l #2

At iteration i

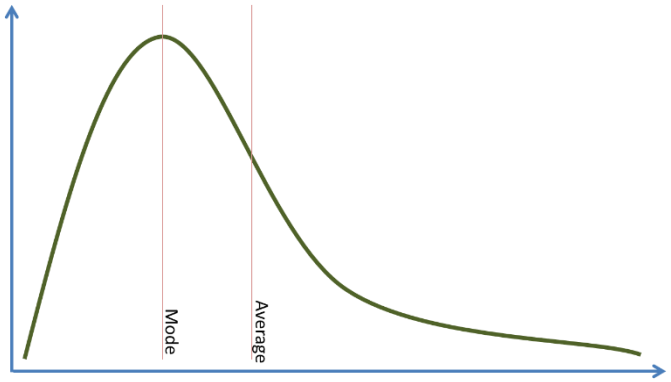


Picture credit: [Team Paperspace]

An intuitive example

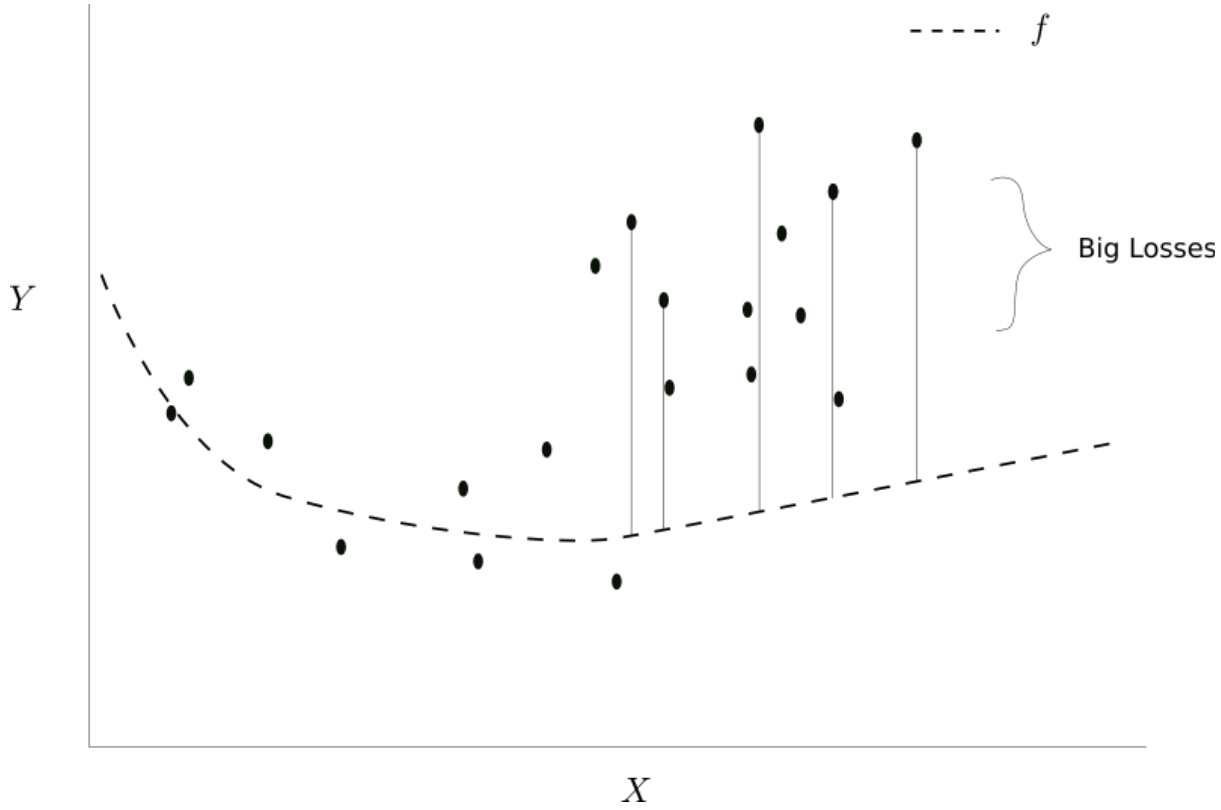


Distiibution of x_l #1



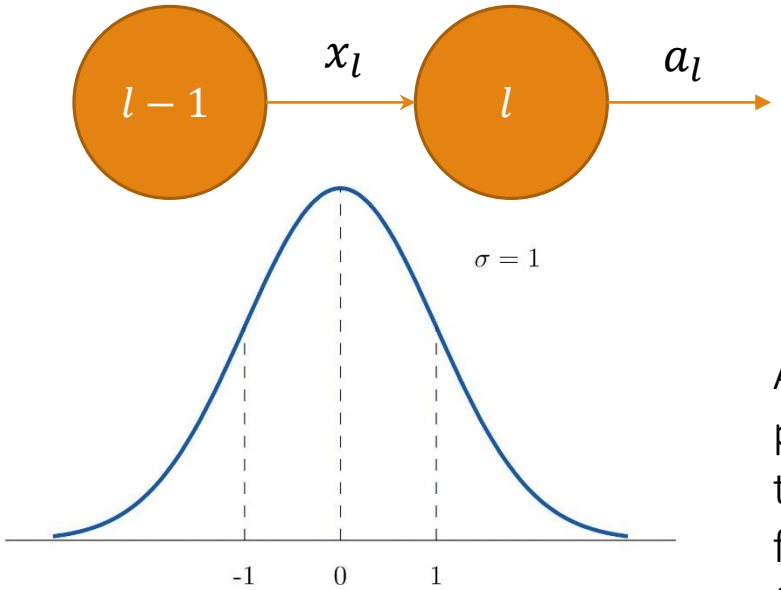
Distiibution of x_l #2

At iteration $i+1$

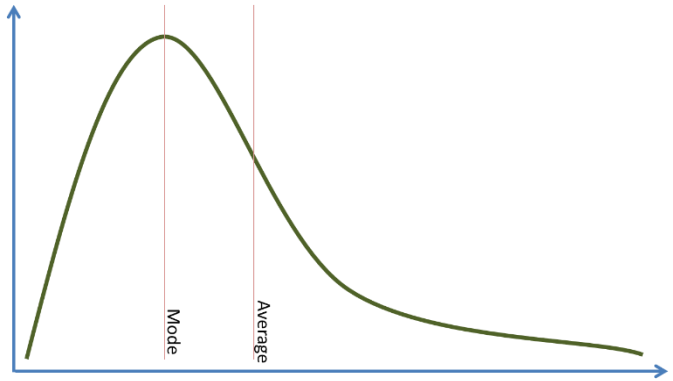


Picture credit: Team Paperspace

An intuitive example

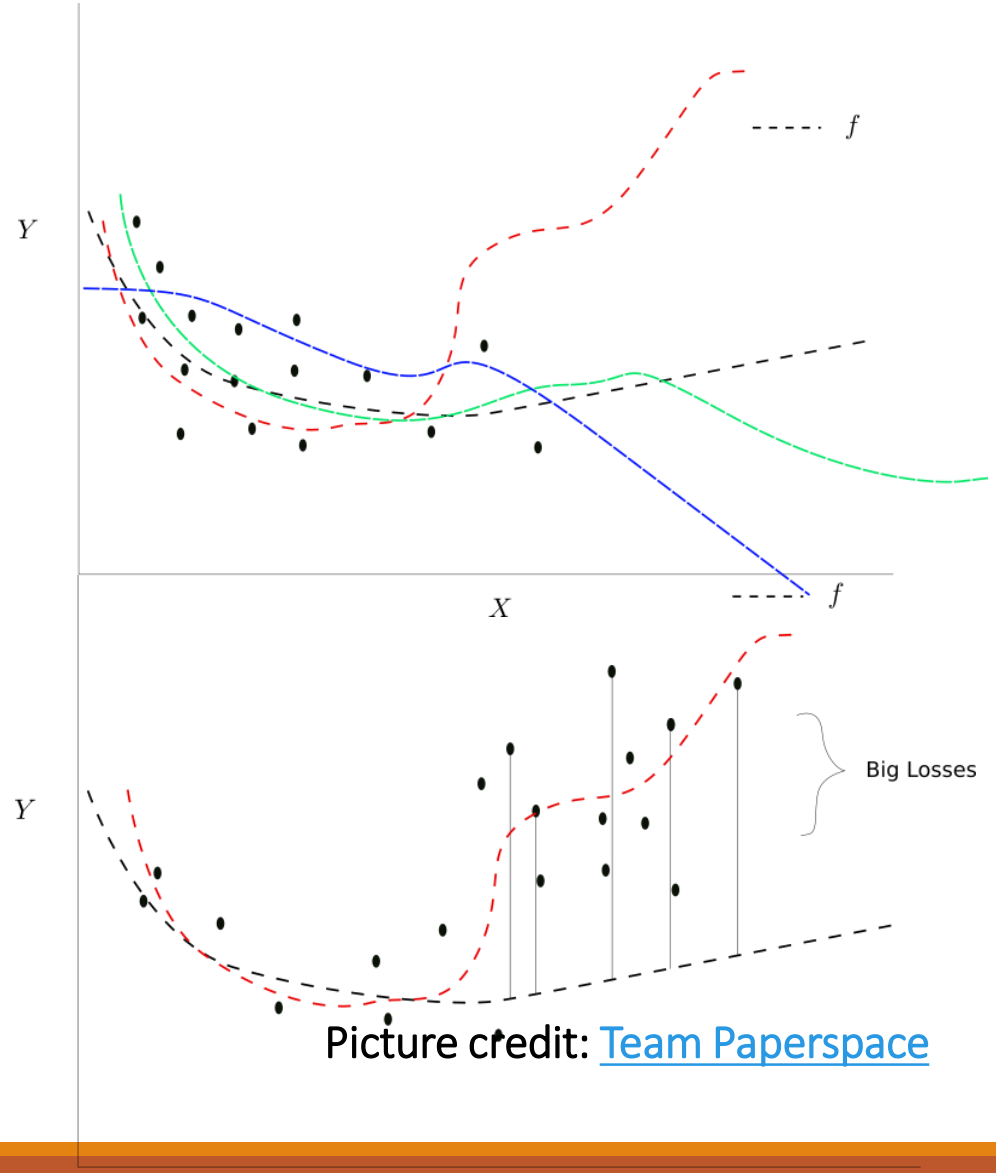


Distiibution of x_l #1



Distiibution of x_l #2

Although we had originally picked the black curve for f, there are many other functions that have similar fitting (losses) on the i-th iteration's dense region, but better behavior on the i-th iteration's the sparse region



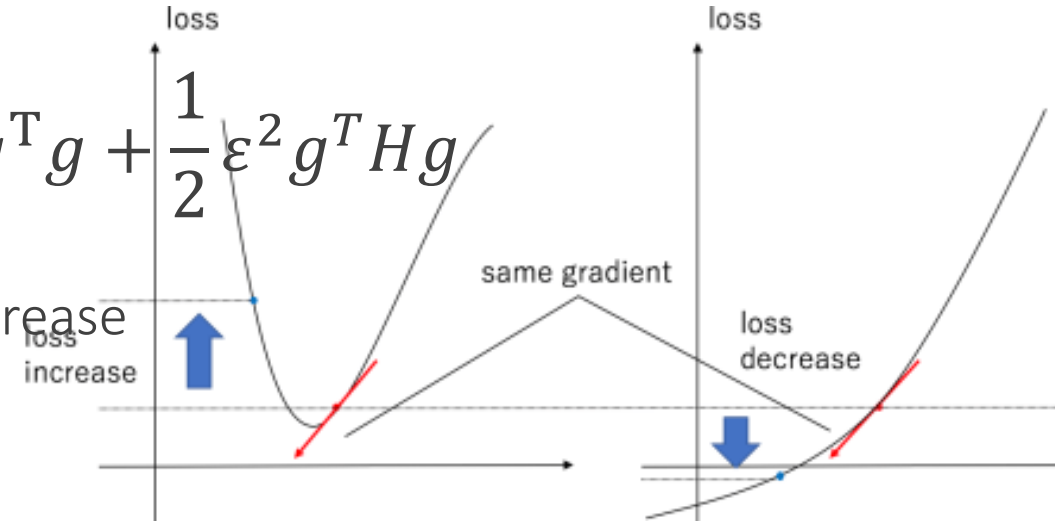
Picture credit: Team Paperspace

Batch normalization – Intuition II

- Batch norm helps the optimizer to control the mean and variance of the layers outputs
- This means, the batch norm sort of cancels out 2nd order effects between different layers
- Loss 2nd order Taylor: $\mathcal{L}(w) = \mathcal{L}(w_0) + (w - w_0)^T g + \frac{1}{2}(w - w_0)^T H (w - w_0)$
- Let's take a miniscule step

$$\mathcal{L}(w - \varepsilon g) = \mathcal{L}(w_0) - \varepsilon g^{\mathrm{T}} g + \frac{1}{2}\varepsilon^2 g^T H g$$

  - With small H, $\varepsilon^2 g^T H g \to 0$ and the loss decreases
  - With large H (high curvature), the loss could even increase
- Bach norm simplifies the learning dynamics



Picture credit: ML Explained

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

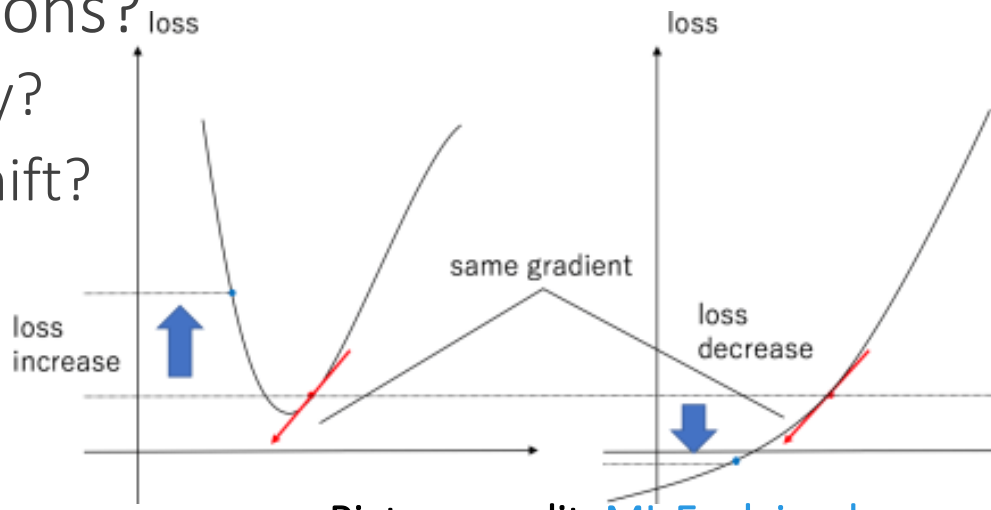
- $\mu = \mu_x + \beta, \sigma = \sigma_x + \gamma$
- $\mu = \beta, \sigma = \gamma$
- $\mu = \beta, \sigma = \beta + \gamma$
- $\mu = \gamma, \sigma = \beta$

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 II

- Bach norm simplifies the learning dynamics
  - Mean of BatchNorm output is β, stdev is γ
  - Mean and Stdev statistics only depend on β, γ, not complex interactions between layers
  - The network must only change β, γ to counter complex interactions
  - And it must change the weights only to fit the data better
- 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?



Picture credit: ML Explained

Batch normalization - Benefits

- Gradients can be stronger → higher learning rates → faster training
  - Otherwise maybe exploding or vanishing gradients or getting stuck to local minima
- Neurons get activated in a near optimal "regime"
- Better model regularization
  - Neuron activations not deterministic, depend on the batch
  - Model cannot be overconfident
- Acts as a regularizer
  - The per mini-batch mean and variance are a noisy version of the true mean and variance
  - Injected noise reduces overfitting during search

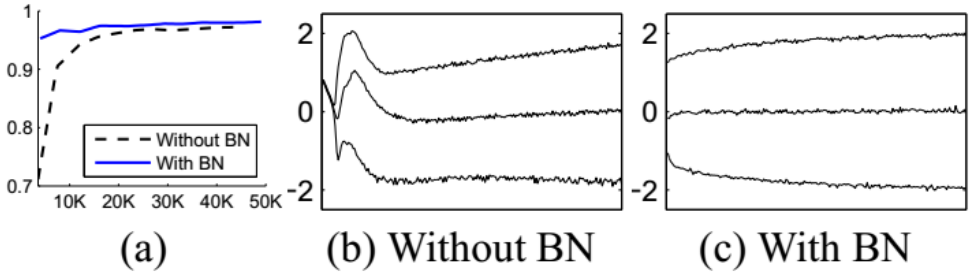


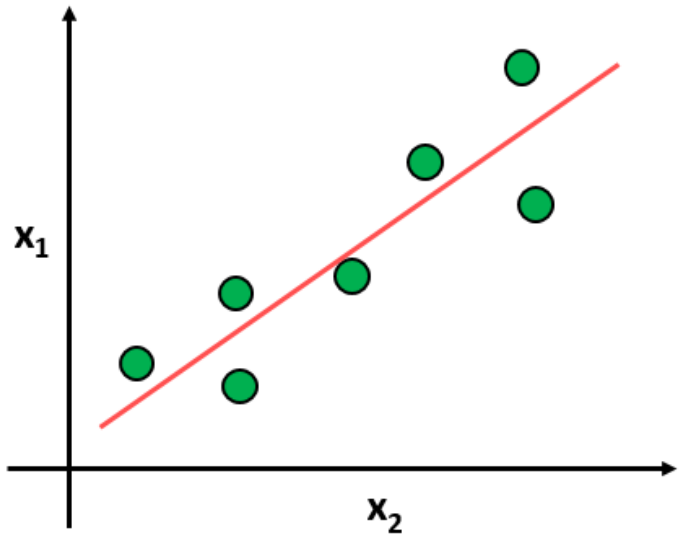
Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of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

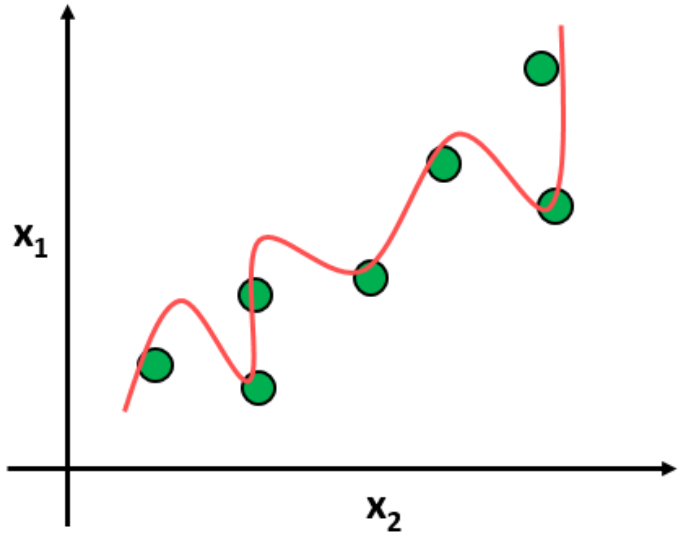
From training to test time

- How do we ship the Batch Norm layer after training
  - We might not have batches at test time
- Often 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

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

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

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

- 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 can pass inside the gradient descent update rule

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

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

"Weight decay", because weights get smaller

ℓ_1-regularization

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

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

- Also ℓ_1-regularization passes inside the gradient descent update rule

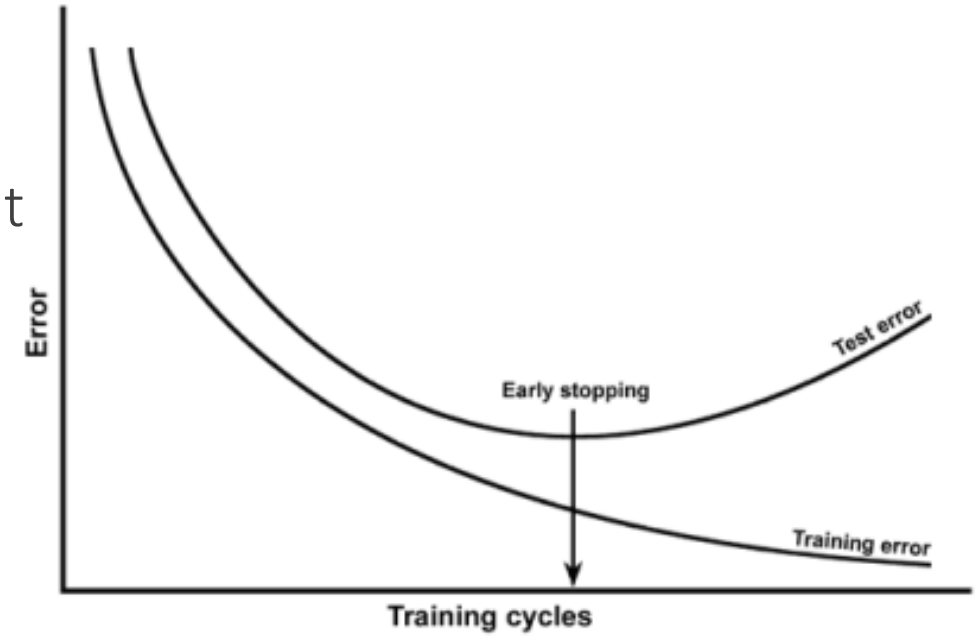
$$w_{t+1} = w_t - \lambda\eta_t \frac{w^{(t)}}{|w^{(t)}|} - \eta_t \nabla_w \mathcal{L}$$

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

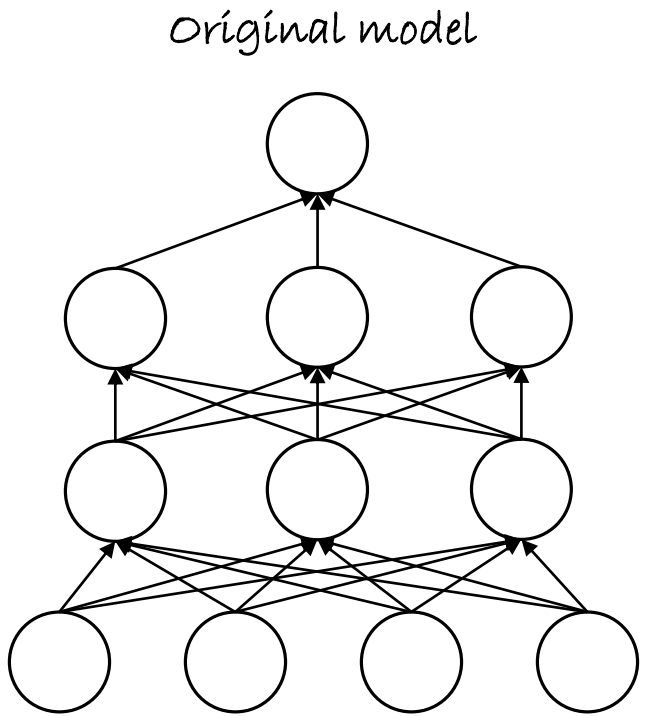


Dropout [Srivastava2014]

- During training setting activations randomly to 0
  - Neurons sampled at random from a Bernoulli distribution with $p = 0.5$
- At test time all neurons are used
  - Neuron activations reweighted by p
- Benefits
  - Reduces complex co-adaptations or co-dependencies between neurons
  - No “free-rider” neurons that rely on others
  - Every neuron becomes more robust
  - Decreases significantly overfitting
  - Improves significantly training speed

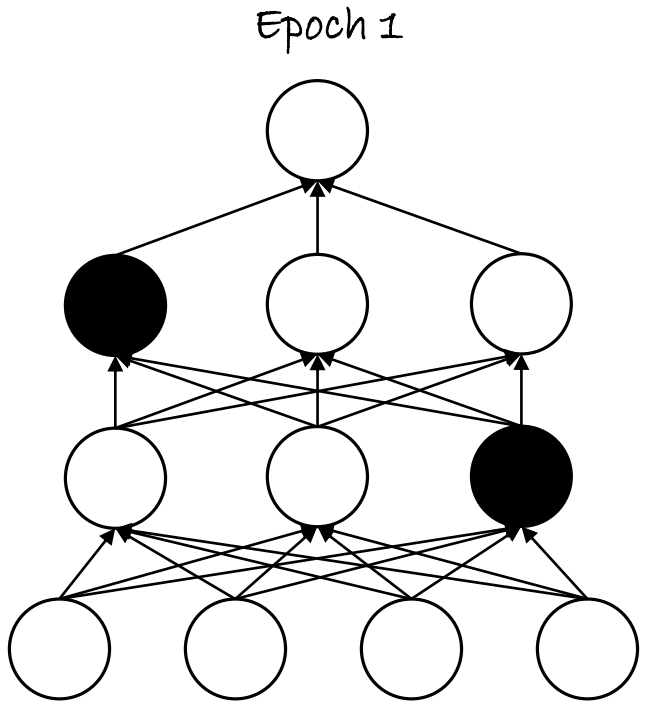
Dropout

- Effectively, a different architecture at every training epoch
  - Similar to model ensembles



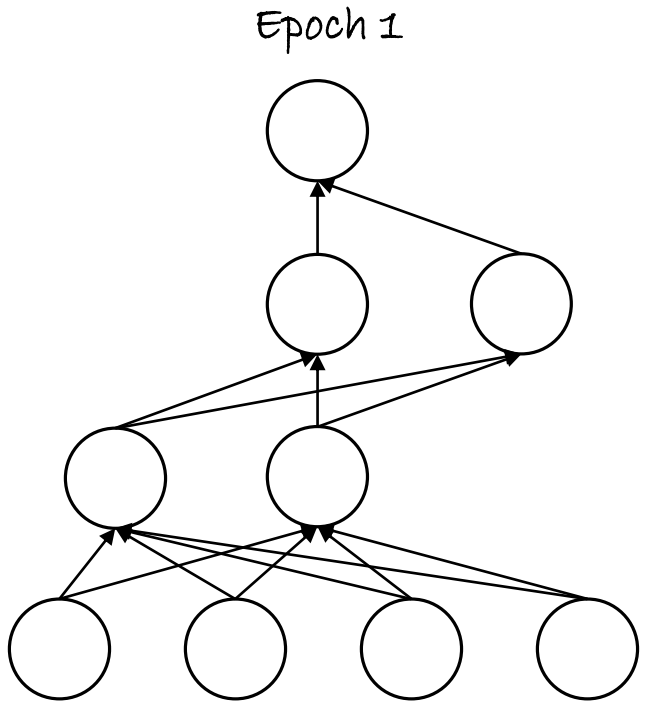
Dropout

- Effectively, a different architecture at every training epoch
  - Similar to model ensembles



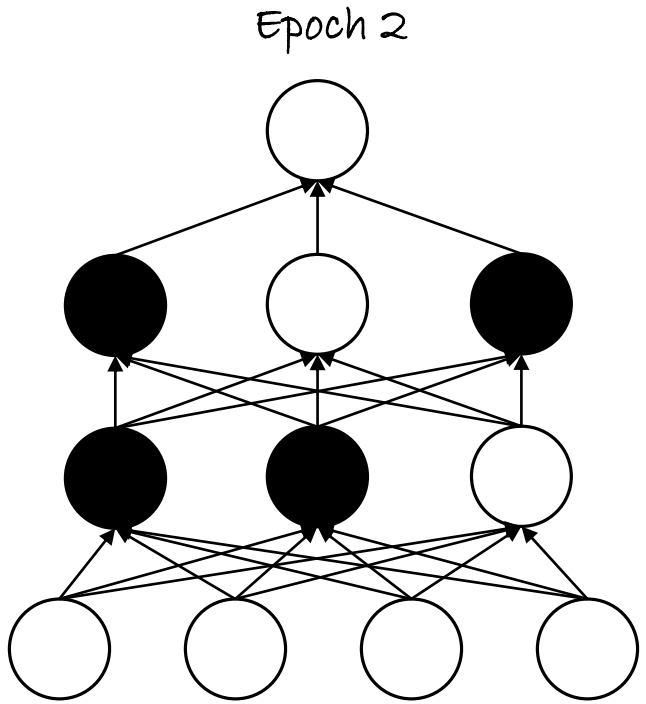
Dropout

- Effectively, a different architecture at every training epoch
  - Similar to model ensembles



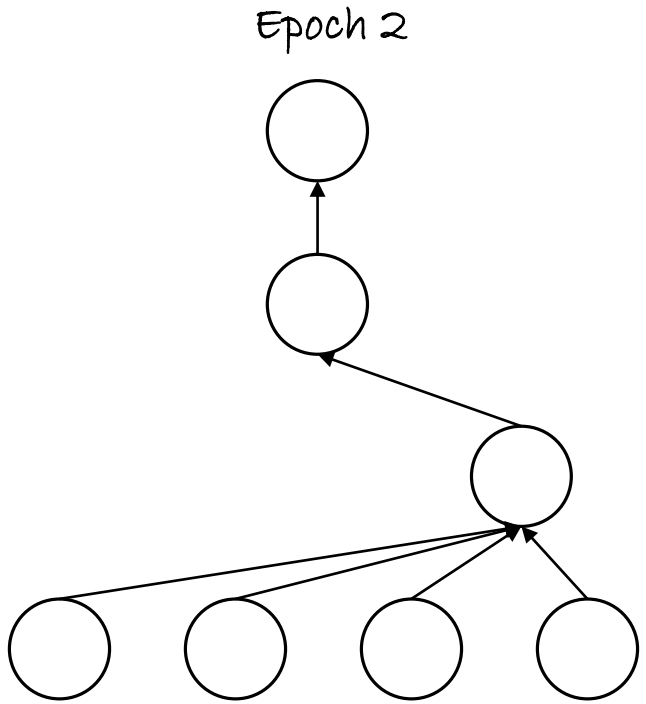
Dropout

- Effectively, a different architecture at every training epoch
  - Similar to model ensembles

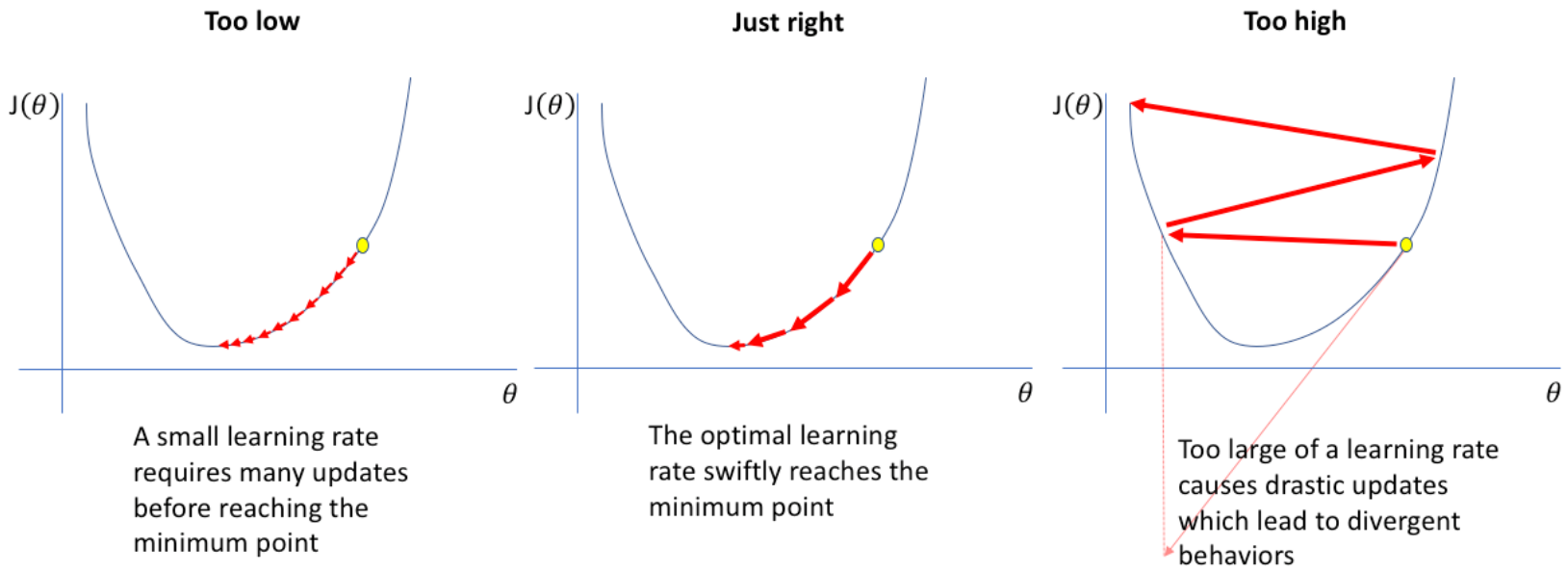


Dropout

- Effectively, a different architecture at every training epoch
  - Similar to model ensembles



Learning rate



Learning rate

- The right learning rate η_t very important for fast convergence
  - Too strong → gradients overshoot and bounce
  - Too weak, → too small gradients → slow training
- Rule of thumb
  - Learning rate of (shared) weights prop. to square root of share weight connections

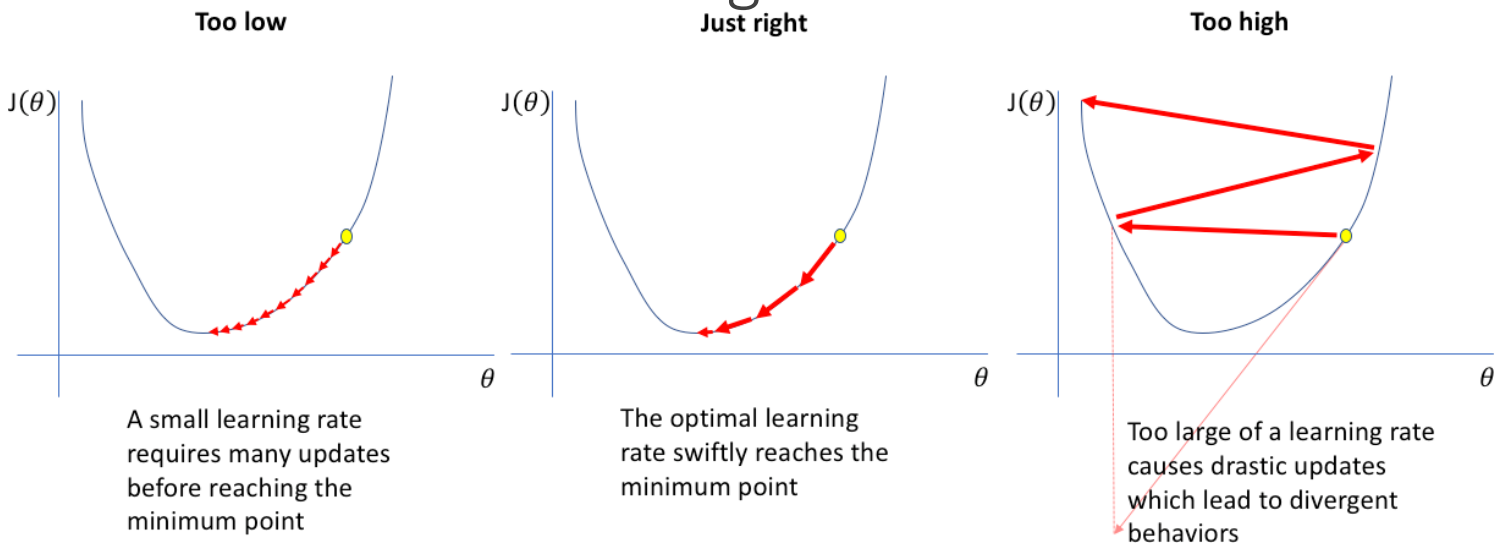
- The right learning rate η_t very important for fast convergence
  - Too strong → gradients overshoot and bounce
  - Too weak, → too small gradients → slow training
- Rule of thumb
  - Learning rate of (shared) weights prop. to square root of share weight connections

Convergence

- The step sizes theoretically should satisfy the following

$$\sum_t^\infty \eta_t = \infty \quad \text{and} \quad \sum_t^\infty \eta_t^2 = 0$$

- Intuitively, the first term ensures that search will reach the high probability regions at some point
- The second term ensures convergence to a mode instead of bouncing

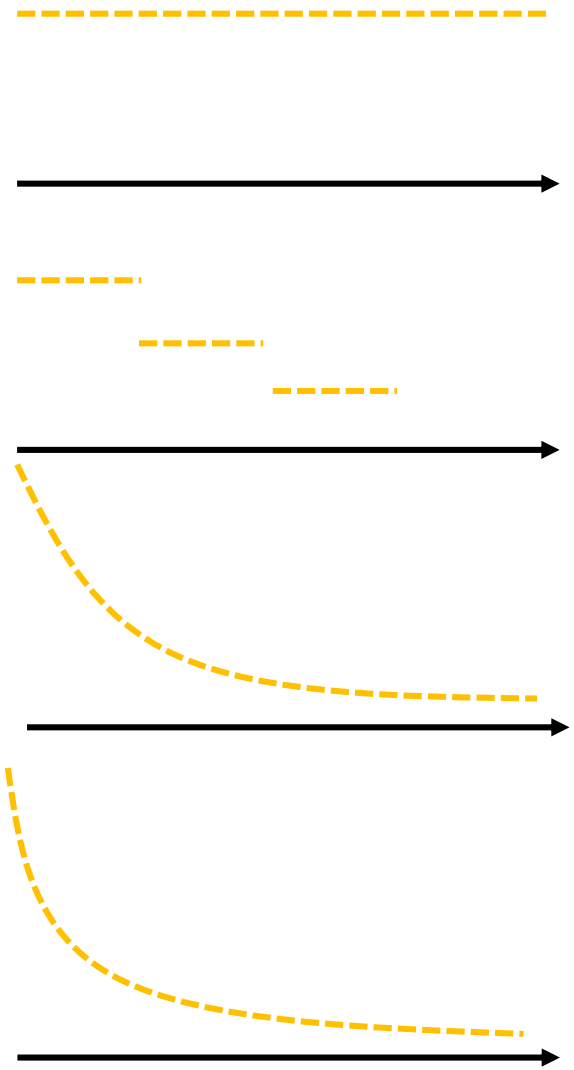


Learning rate

- Learning rate per weight is often advantageous
  - Some weights are near convergence, others not
- Adaptive learning rates are also possible, based on the errors observed
  - [Sompolinsky1995]

Learning rate schedules

- Constant
  - Learning rate remains the same for all epochs
- Step decay
  - Decrease (e.g. η_t/T or η_t/T) every T number of epochs
- 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 and only a single extra hyper-parameter T (T =2, 10)



Stochastic Gradient Langevin Dynamics

- Bayesian Learning via Stochastic Gradient Langevin Dynamics, M. Welling and Y. W. Teh, ICML 2011
- Adding the right amount of noise to a standard stochastic gradient optimization algorithm → converge to samples from the true posterior distribution as the step size is annealed
- Transition between optimization and Bayesian posterior sampling provides an inbuilt protection against overfitting

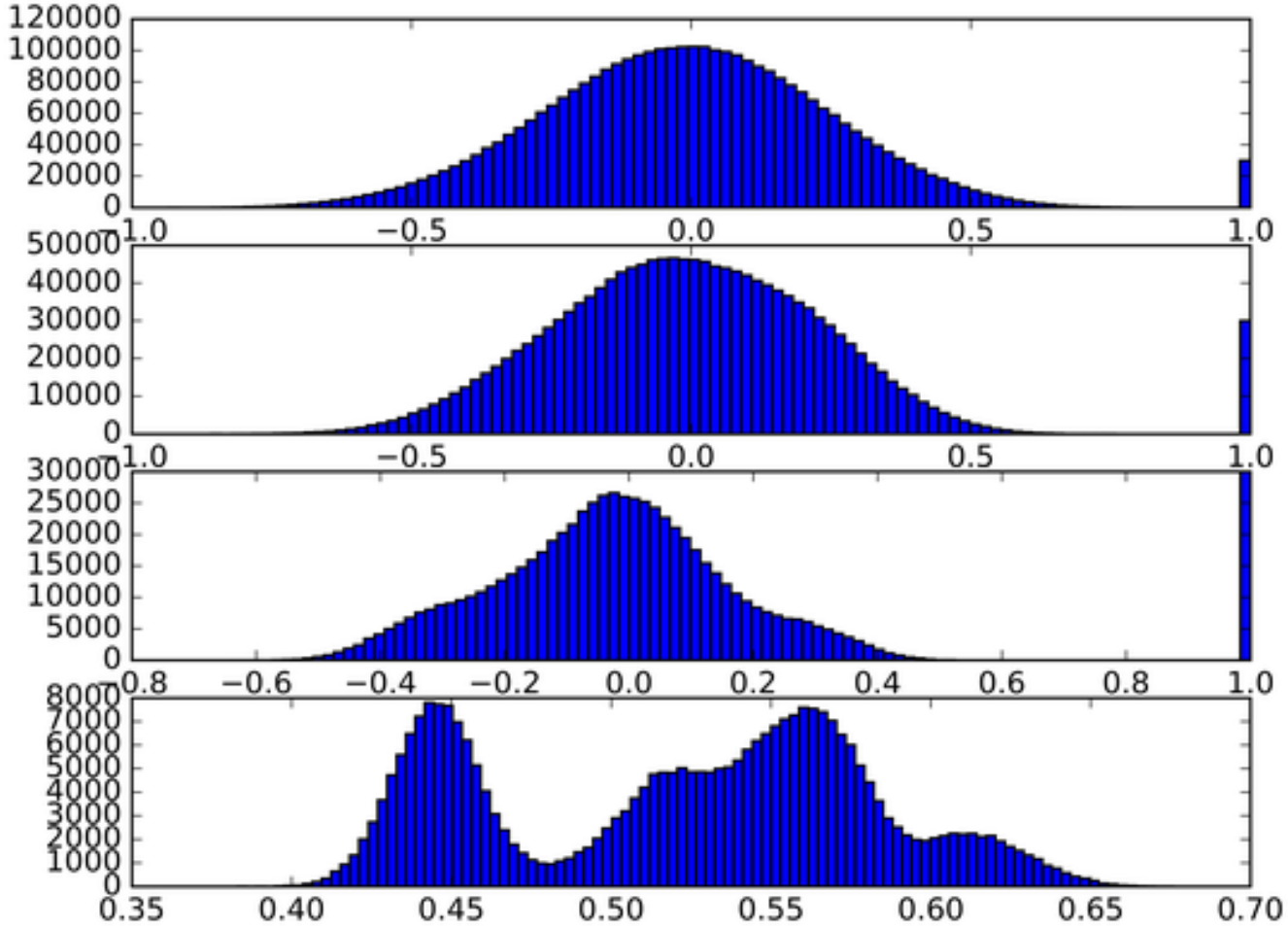
SGD Some noise annealed over time

$$\Delta w_t = \frac{\eta_t}{2}\left(\nabla \log p(\theta_t) + \frac{N}{n}\sum_{i}^{N} \nabla \log p(x_{ti}|w_t)\right) + \varepsilon_t, \qquad \varepsilon_t \sim N(0, \eta_t)$$

In practice

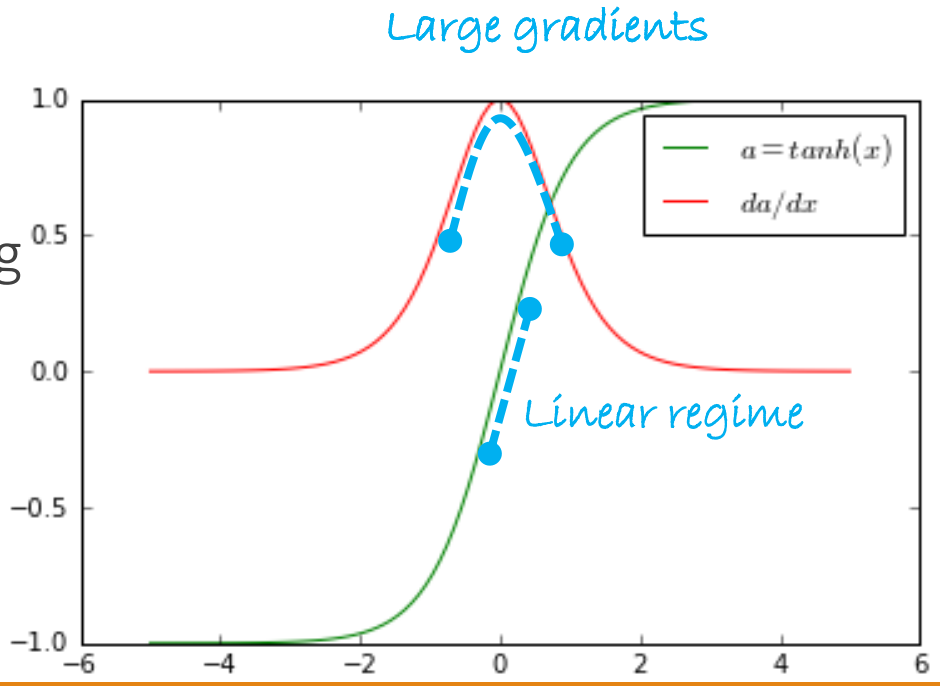
- Try several log-spaced values $10^{-1}, 10^{-2}, 10^{-3}, \ldots$ on a smaller set
  - Then, you can narrow it down from there around where you get the lowest error
- You can decrease the learning rate every 10 (or some other value) full training set epochs
  - Although this highly depends on your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

Weight initialization



Weight initialization

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



Weight initialization

- Weights must be initialized **to preserve the variance** of the activations during the forward and backward computations
  - Especially for deep learning
  - All neurons operate in their full capacity
- Input variance == Output variance

Question: Why similar input/output variance?

- Good practice: initialize weights to be asymmetric
  - Don't give save 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
  - Especially for deep learning
  - All neurons operate in their full capacity
- Input variance == Output variance

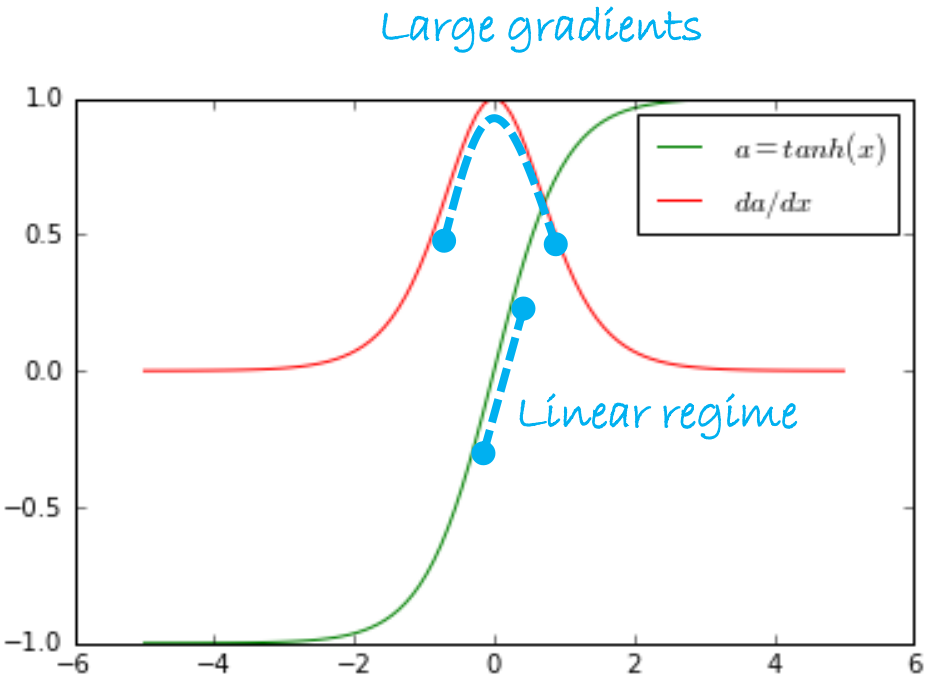
Question: Why similar input/output variance?

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

- Good practice: initialize weights to be asymmetric
  - Don't give save 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 sigmoid-like neurons

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



Xavier initialization [Glorot2010]

- 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

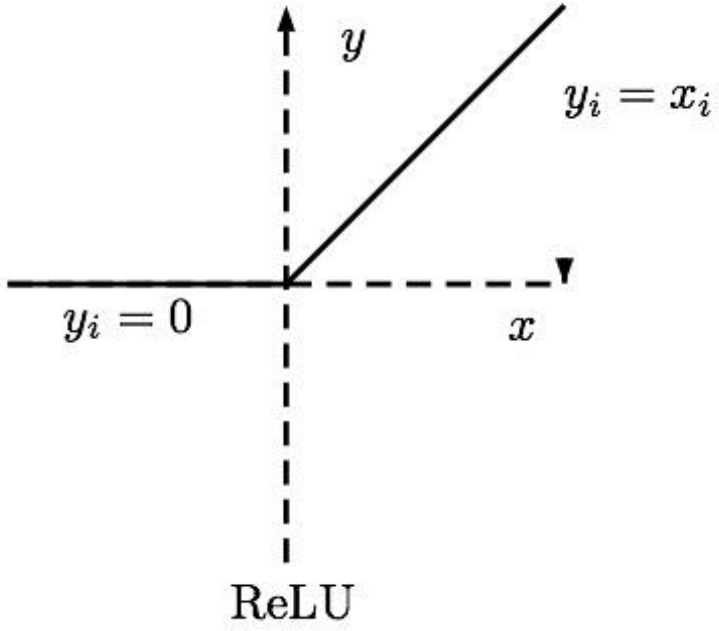
$$w \sim N\left(0, \sqrt{1/d}\right)$$

where d is the number of neurons in the input

[He2015] initialization for ReLUs

- Unlike sigmoids, ReLUs ground to 0 the linear activations half the time
- Double weight variance
  - Compensate for the zero flat-area →
  - Input and output maintain same variance
  - Very similar to Xavier initialization
- Draw random weights from $\mathrm{w} \sim N\left(0, \sqrt{2/d}\right)$

  where d is the number of neurons in the input



Babysitting Deep Nets

- Always check your gradients if not computed automatically
- Check that in the first round you get a random loss
- Check network with few samples
  - Turn off regularization. You should predictably overfit and have a 0 loss
  - Turn or regularization. The loss should increase
- Have a separate validation set
  - Compare the curve between training and validation sets
  - There should be a gap, but not too large
- Preprocess the data to at least have 0 mean
- Initialize weights based on activations functions
  - For ReLU Xavier or HeICCV2015 initialization
- Always use ℓ_2-regularization and dropout
- Use batch normalization

Summary

- SGD and advanced SGD-like optimizers
- Input normalization
- Optimization methods
- Regularizations
- Architectures and architectural hyper-parameters
- 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/