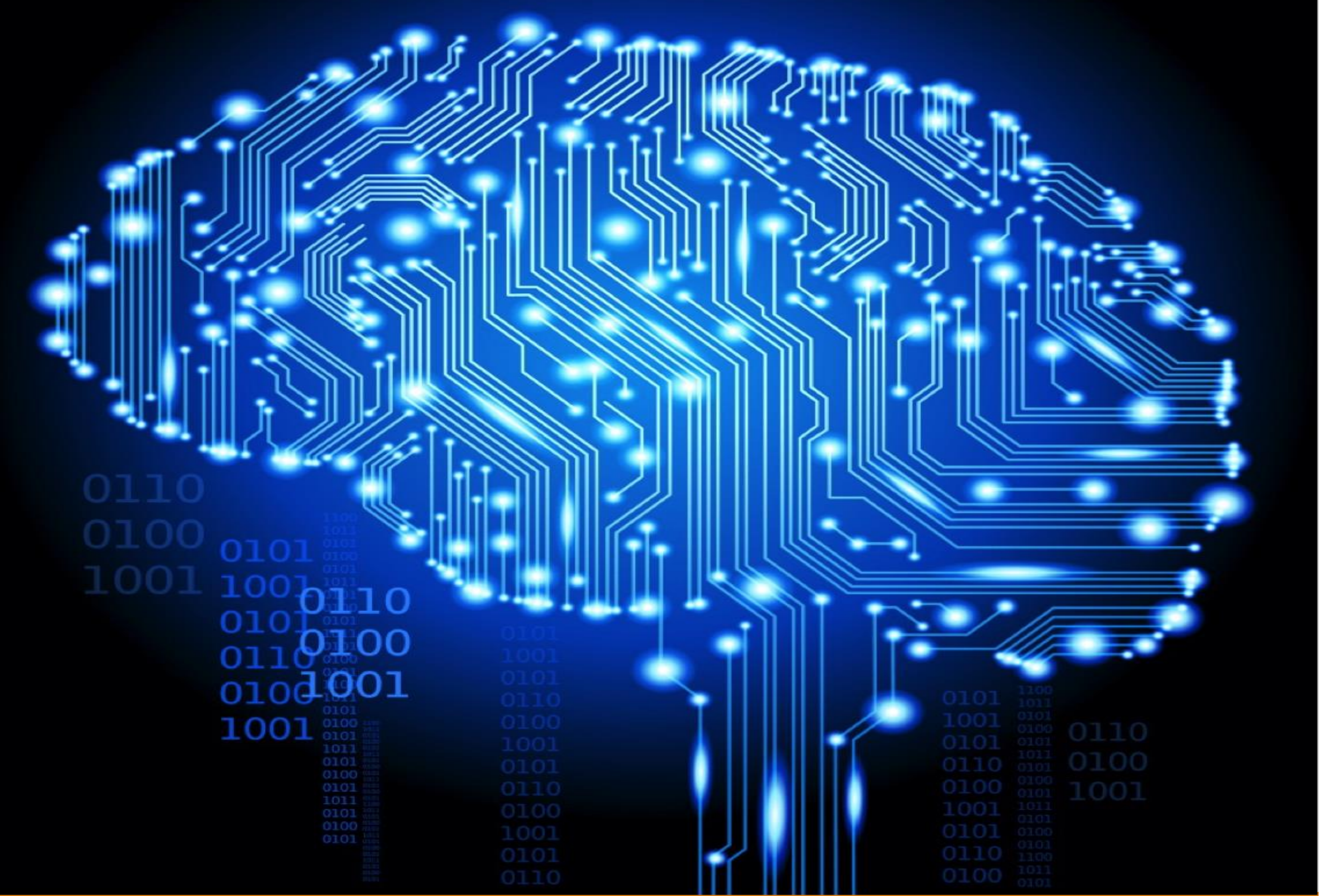


Lecture 3: Advanced Optimizations

Deep Learning @ UvA

Previous lecture

- Machine learning paradigm for neural networks
- Backpropagation algorithm, backbone for training neural networks
- Neural network == modular architecture
- Visited different modules, saw how to implement and check them

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

Deeper into Neural Networks & Deep Neural Nets

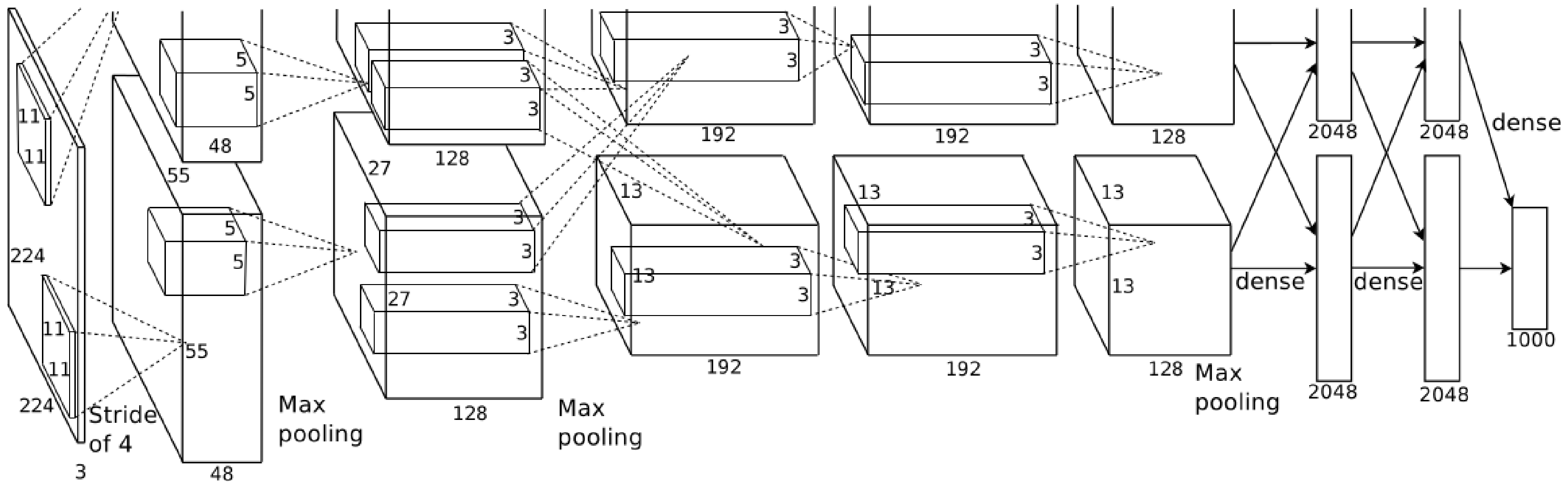


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

A Neural/Deep Network in a nutshell

1. The Neural Network

$$\boxed{a_L(x; \theta_{1,\ldots,\mathrm{L}}) = h_L\,(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}), \theta_L)}$$

2. Learning by minimizing empirical error

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

3. Optimizing with Gradient Descent based methods

$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}}$$

Pure Optimization vs Machine Leanring Training

- In pure optimization the end goal is finding the minimum (or maximum)
  - E.g., optimizingrailroad network in the Netherlands, the end goal is finding the optimal combination of train schedules, train availability, etc
- The goal is very direct
  - Formulate your problem mathematically as best as possible
  - Find the best possible solution of your mathematical solution
- In training Machine Learning algorithms finding the cost function is usually only a surrogate to your goal
  - 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
  - Even the “optimal” parameters are not necessarily the best choice for your end goal

Empirical Risk Minimization

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

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

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

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

  - That is, the risk on the available training samples

Optimizing with Stochastic Gradient Descent

1. The Neural Network

$$\boxed{a_L(x; \theta_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}), \theta_L)}$$

2. Learning by minimizing empirical error

$$\boxed{\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; \theta_{1,\ldots,L}))}$$

3. Optimizing with Gradient Descent based methods

$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}}$$

Gradient Descent

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

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t g^{(t)}$$

  - $g^{(t)}: gradient$

- If the gradient is computed on the whole training set we have batch gradient descent

$$g^{(t)} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \mathcal{L}(\theta; x_i, y_i)$$

  - This is only an approximation to the true gradient, as it is computed empirically from all the available training samples (x_i, y_i)

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

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

$$B_j = sample(D)$$

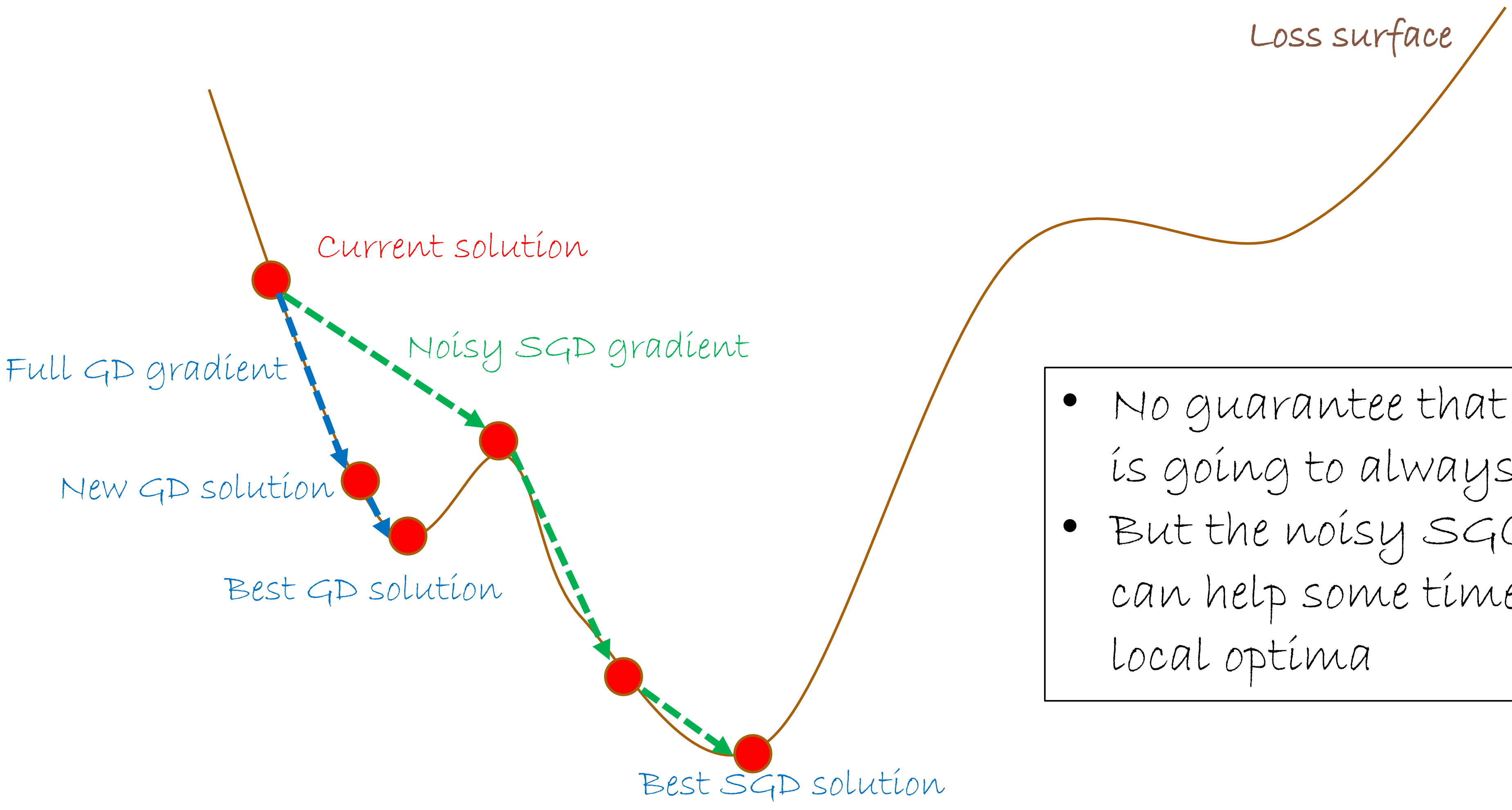
$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta_t}{|B_j|} \sum_{i \in B_j} \nabla_\theta \mathcal{L}_i$$

- 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

- Much faster than Gradient Descent
- Results are often better
- Also suitable for datasets that change over time
- Variance of gradients increases when batch size decreases

SGD is often better

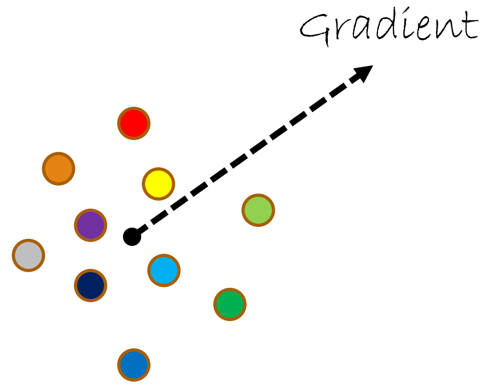


- No guarantee that this is what is going to always happen.
- But the noisy SGC gradients can help some times escaping local optima

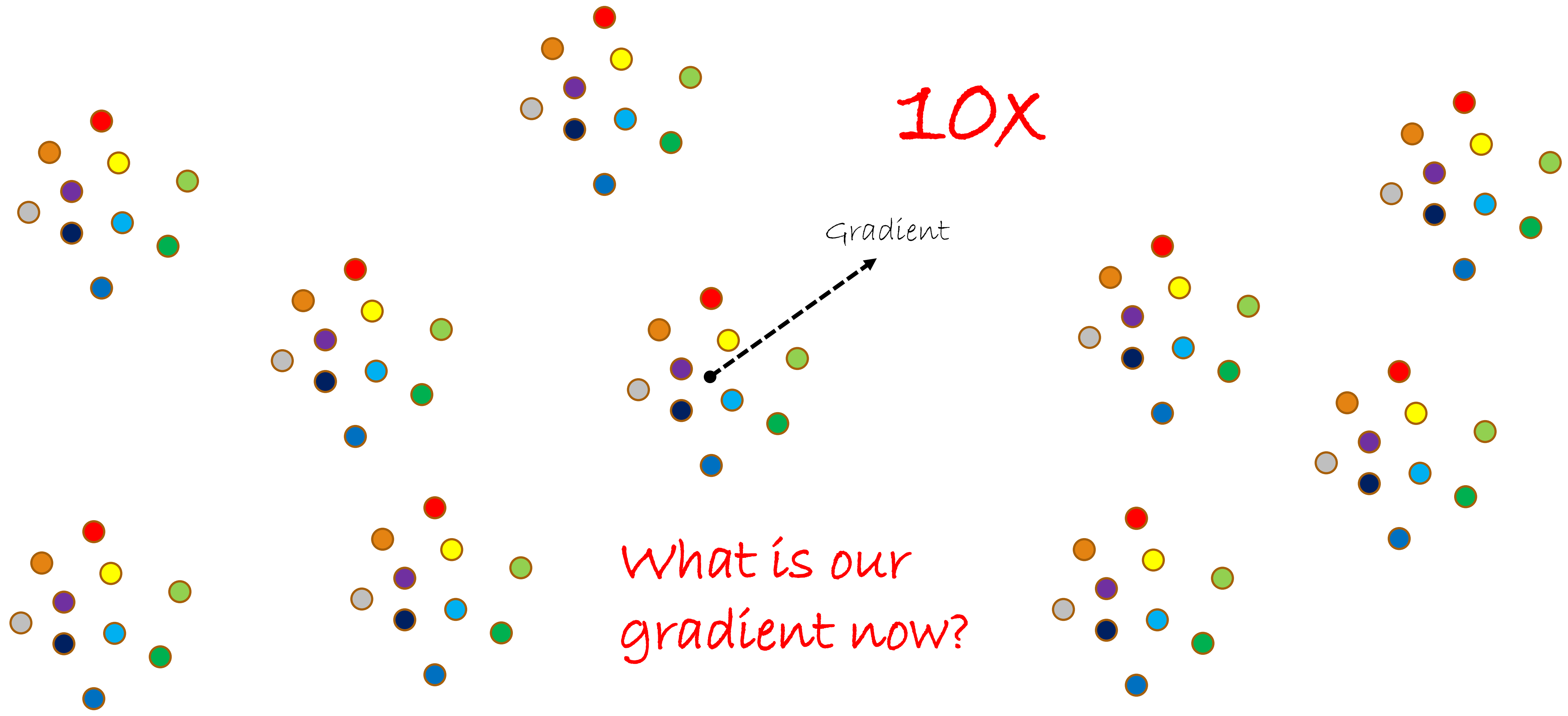
SGD is often better

- (A bit) Noisy gradients act as regularization
- Gradient Descent → Complete gradients
- Complete gradients fit optimally the (arbitrary) data we have, not 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

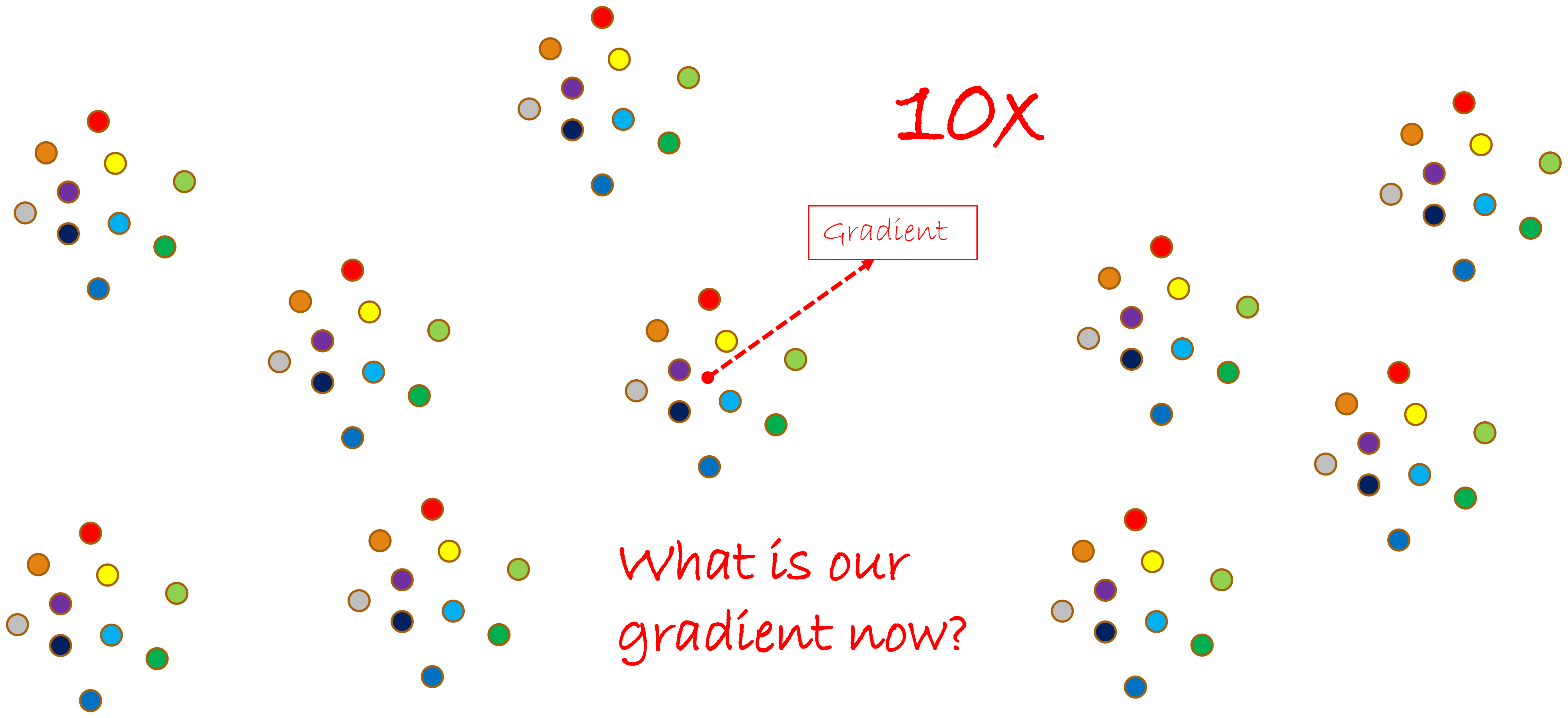


SGD is faster



What is our gradient now?

SGD is faster



SGD is faster

- Of course in real situations data do not replicate
- However, after a sizeable amount of data there are clusters of data that are similar
- Hence, the gradient is approximately alright
- Approximate alright is great, is even better in many cases actually

SGD for dynamically changed datasets

- Often datasets are not "rigid"
- Imagine Instagram
  - Let's assume 1 million of **new** images uploaded per week and we want to build a "cool picture" classifier
  - Should "cool pictures" from the previous year have the same as much influence?
  - No, the learning machine should track these changes
- With GD these changes go undetected, as results are averaged by the many more "past" samples
  - Past "over-dominates"
- A properly implemented SGD can track changes much better and give better models
  - [LeCun2002]

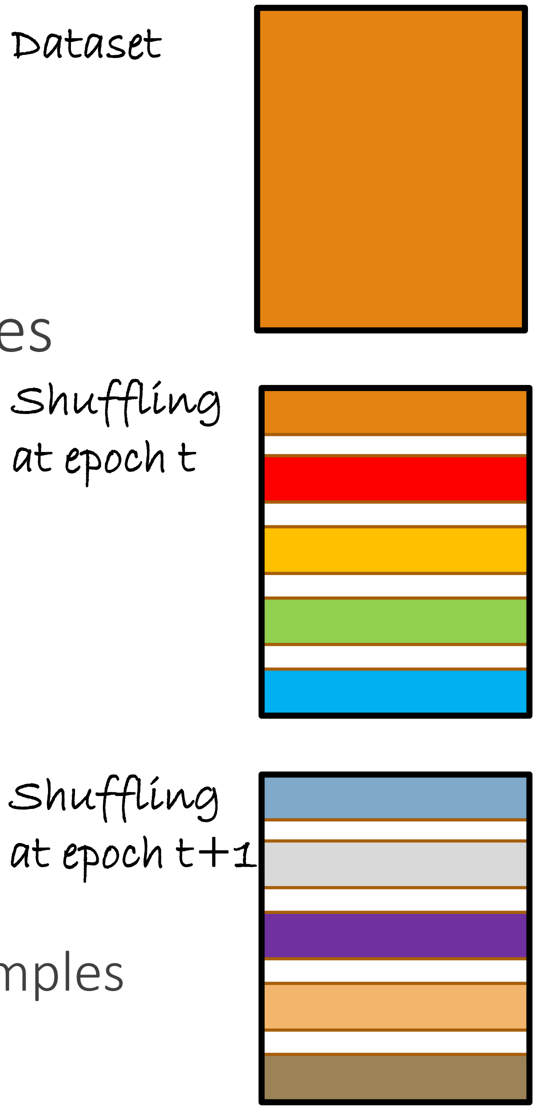
Popular today

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
  - As different as possible
- 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



Backpropagation again

- **Step 1.** Compute forward propagations for all layers recursively

$$\boxed{a_l = h_l(x_l) \text{ and } x_{l+1} = a_l}$$

- **Step 2.** Once done with forward propagation, follow the reverse path.
  - Start from the last layer and for each new layer compute the gradients
  - Cache computations when possible to avoid redundant operations

$$\boxed{\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}} \qquad \boxed{\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T}$$

- **Step 3.** Use the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$ with Stochastic Gradient Descent to train

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

Challenges in optimization

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

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

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

  - 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
- Cliffs and exploding gradients
- Long-term dependencies

Data preprocessing & normalization

1. The Neural Network

$$\boxed{a_L(x; \theta_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}), \theta_L)}$$

2. Learning by minimizing empirical error

$$\boxed{\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; \theta_{1,\ldots,L}))}$$

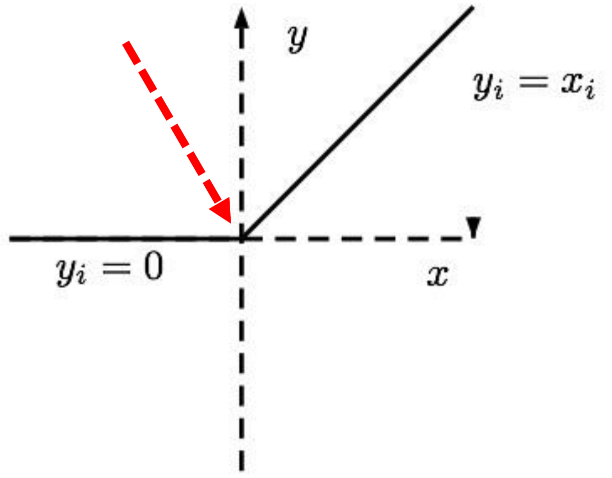
3. Optimizing with Gradient Descent based methods

$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}}$$

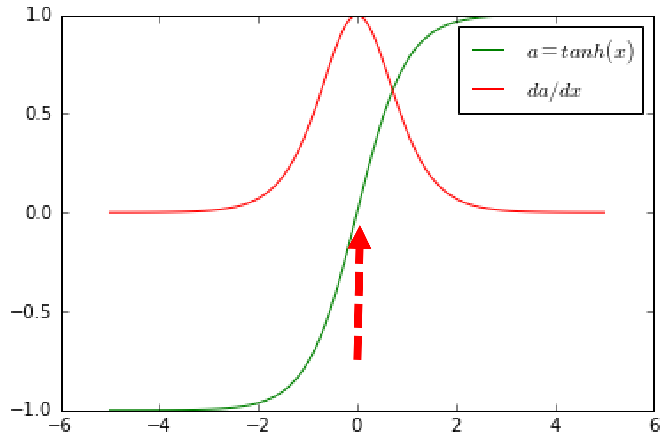
Data pre-processing

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

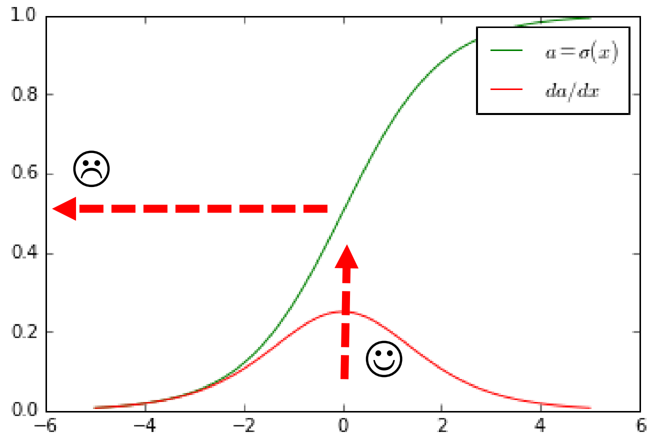
ReLU ☺



$\tanh(x)$ ☺



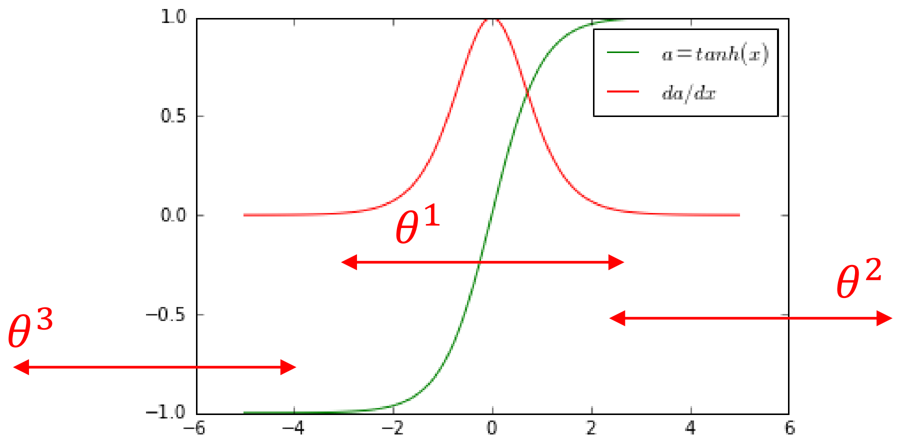
$\sigma(x)$ ☹



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

$x = [x^1, x^2, x^3]^T, \theta = [\theta^1, \theta^2, \theta^3]^T, a = \tanh(\theta^T x)$



x^1, x^2, x^3 → much different covariances

↓

Generated gradients $\left.\frac{d\mathcal{L}}{d\theta}\right|_{x^1, x^2, x^3}$: much different

↓

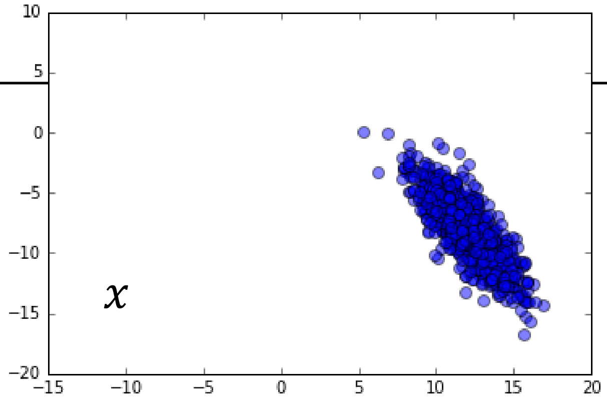
Gradient update harder: $\theta^{(t+1)} = \theta^{(t)} - \eta_t \begin{bmatrix} d\mathcal{L}/d\theta^1 \\ d\mathcal{L}/d\theta^2 \\ d\mathcal{L}/d\theta^3 \end{bmatrix}$

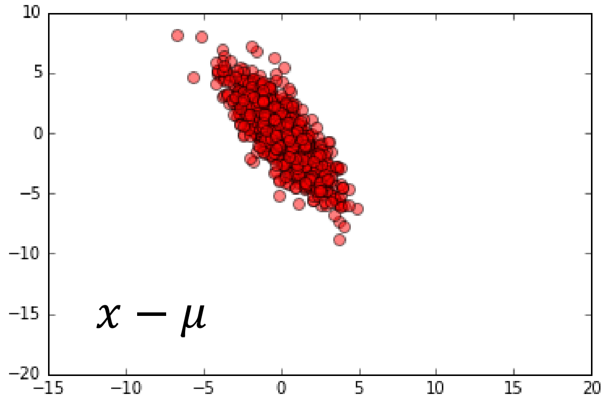
Data pre-processing

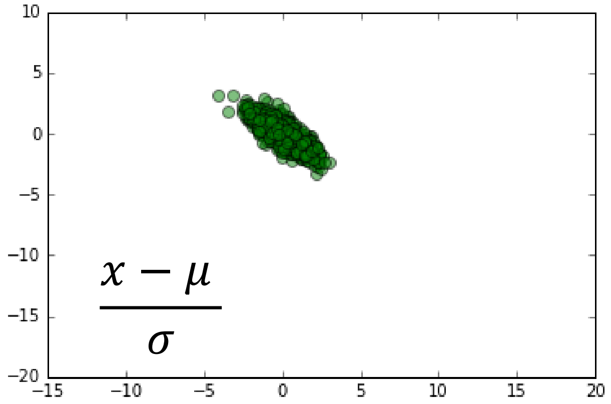
- Input variables should be as decorrelated as possible
  - Input variables are “more independent”
  - Network is forced to find non-trivial correlations between inputs
  - Decorrelated inputs $\rightarrow$ Better optimization
  - Obviously not the case when inputs are by definition correlated (sequences)
- Extreme case
  - extreme correlation (linear dependency) might cause problems [CAUTION]

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







$N(\mu, \sigma^2) = N(0,1)$ – Making things faster

- Instead of “per-dimension” → all input dimensions simultaneously
- If dimensions have similar values (e.g. pixels in natural images)
  - Compute one (μ, σ^2) instead of as many as the input variables
  - Or the per color channel pixel average/variance

$$(\mu_{red}, \sigma_{red}^2), (\mu_{green}, \sigma_{green}^2), (\mu_{blue}, \sigma_{blue}^2)$$

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 ($\mu = 0$)

PCA Whitening

- If C the covariance matrix of your dataset, compute eigenvalues and eigenvectors with SVD

$$U, \Sigma, V^T = svd(C)$$

- Decorrelate (PCA-ed) dataset by

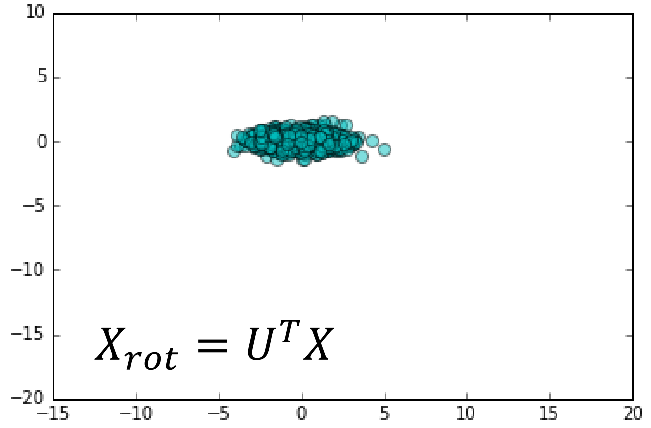
$$X_{rot} = U^T X$$

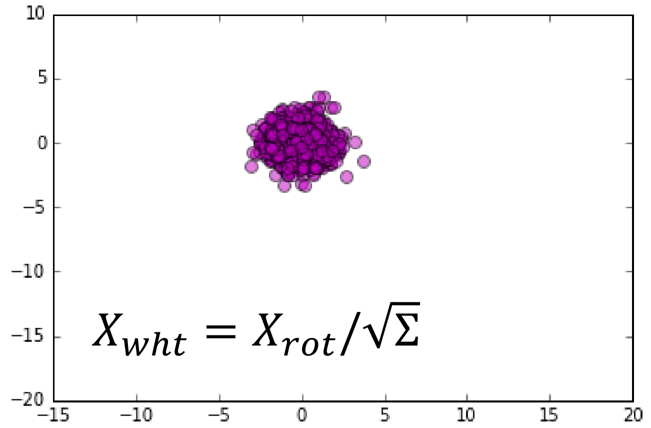
  - Subset of eigenvectors $U' = [u_1, \dots, u_q]$ to reduce data dimensions

- Scaling by square root of eigenvalues to whiten data

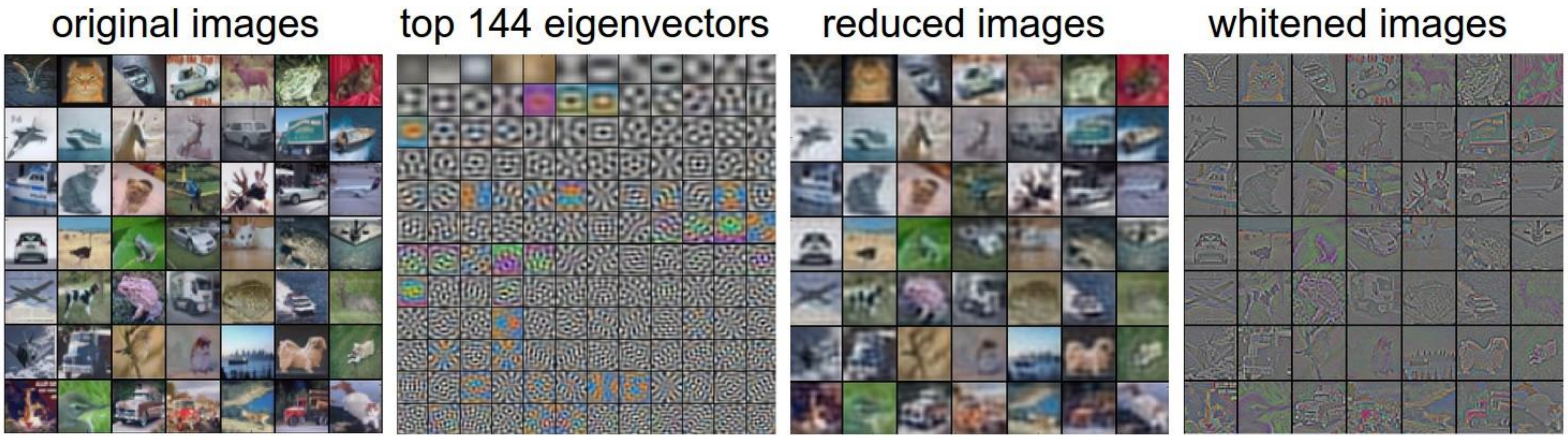
$$X_{wht} = X_{rot} / \sqrt{\Sigma}$$

- Not used much with Convolutional Neural Nets
  - The zero mean normalization is more important





Example

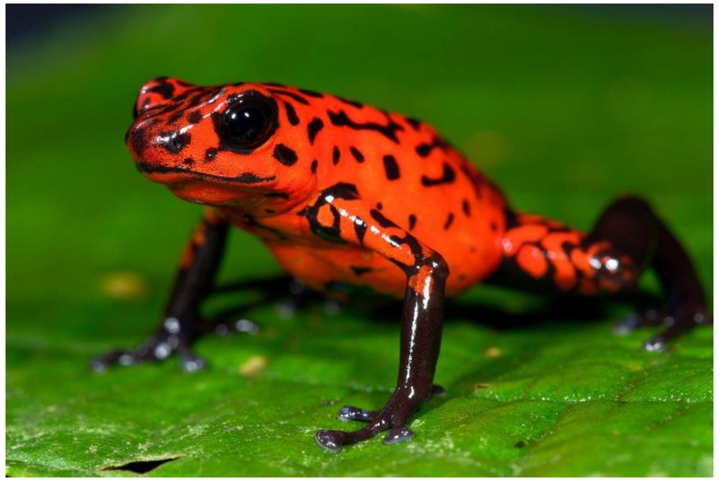


Images taken from A. Karpathy course website: http://cs231n.github.io/neural-networks-2/

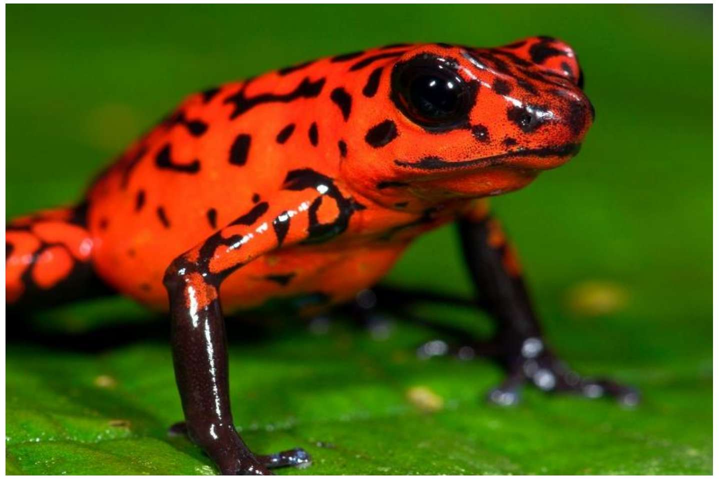
Data augmentation [Krizhevsky2012]

Original

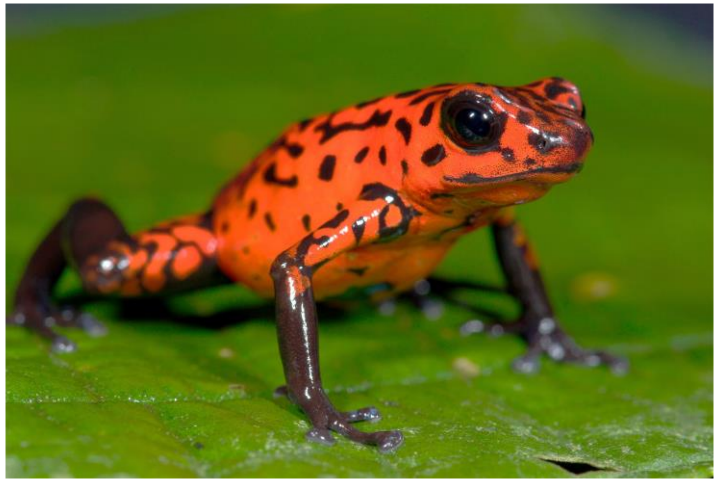
Flip



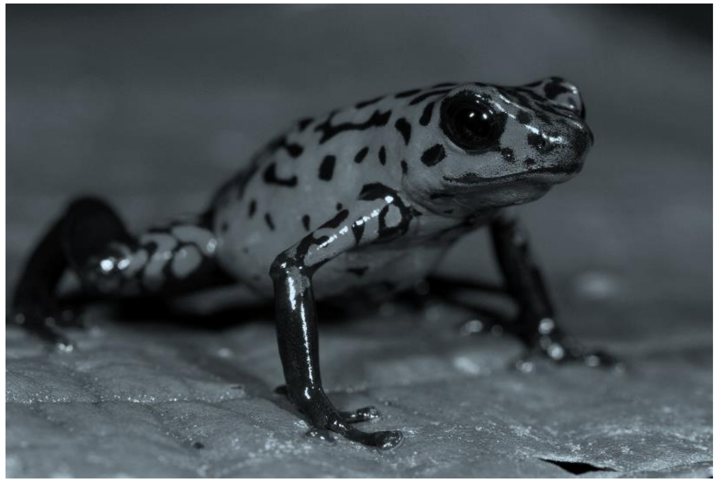
Random crop



Contrast

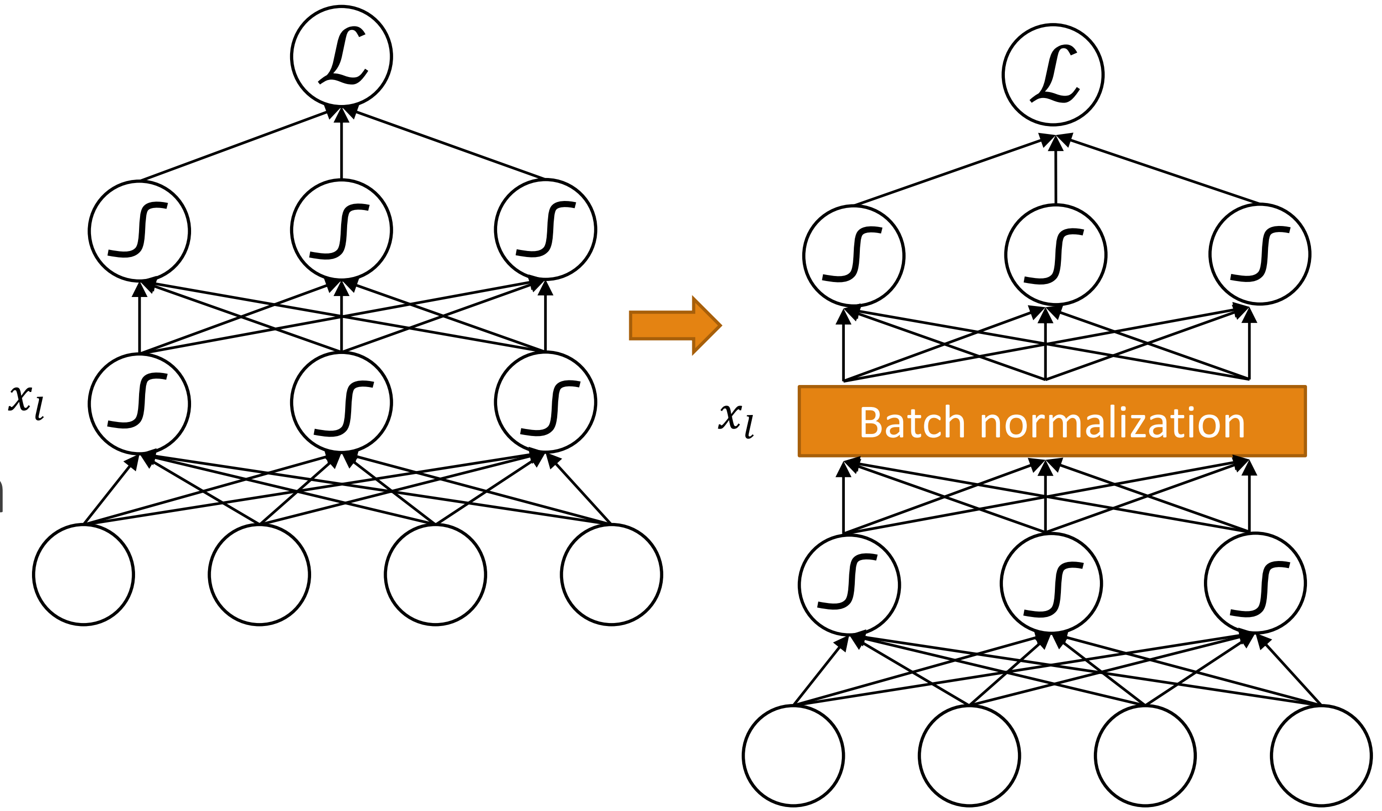


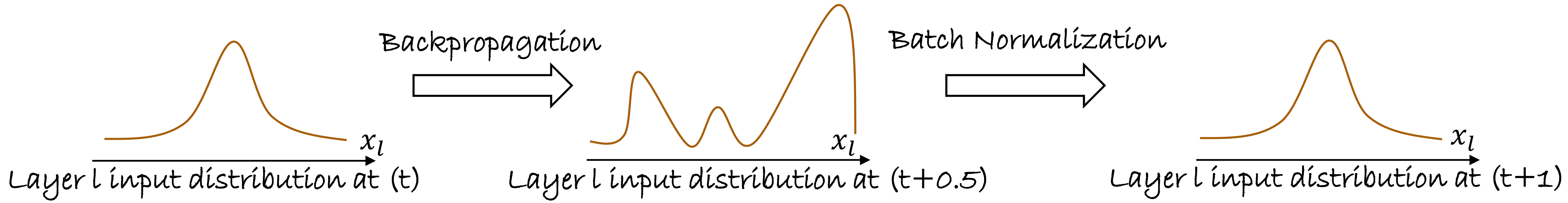
Tint



Batch normalization [Ioffe2015]

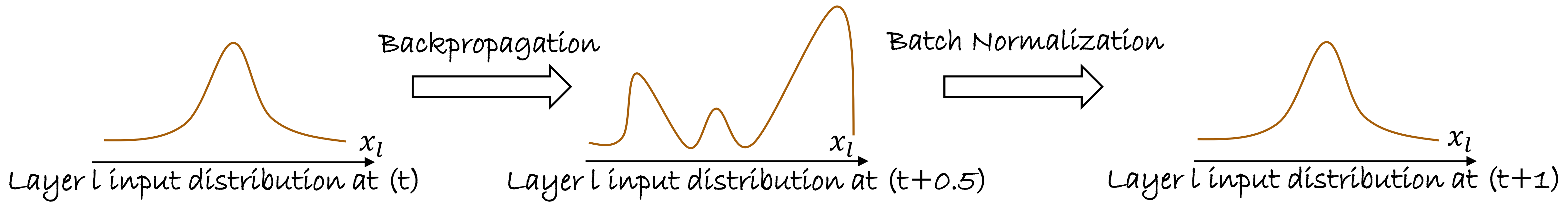
- Weights change → the distribution of the layer inputs changes per round
  - Covariance shift
- Normalize the layer inputs with batch normalization
  - Roughly speaking, normalize x_l to $N(0,1)$ and rescale





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 - Intuitively


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 – 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]

$$\underbrace{\gamma \qquad \beta}_{\text{Trainable parameters}}$$

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

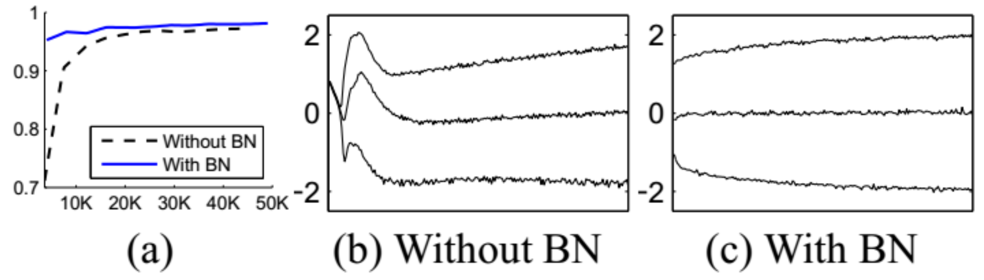


Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of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

Regularization

1. The Neural Network

$$\boxed{a_L(x;\theta_{1,\ldots,\mathrm{L}}) = h_L\,(h_{L-1}(\ldots h_1(x,\theta_1),\theta_{L-1}),\theta_L)}$$

2. Learning by minimizing empirical error

$$\boxed{\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x;\theta_{1,\ldots,\mathrm{L}}))}$$

3. Optimizing with Gradient Descent based methods

$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}}$$

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

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x;\theta_{1,\ldots,L})) + \lambda\Omega(\theta)$$

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

ℓ_2-regularization

- Most important (or most popular) regularization

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x;\theta_{1,\dots,L})) + \frac{\lambda}{2}\sum_l \|\theta_l\|^2$$

- The ℓ_2-regularization can pass inside the gradient descent update rule

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t(\nabla_\theta \mathcal{L} + \lambda\theta_l) \Longrightarrow$$
$$\theta^{(t+1)} = (1 - \lambda\eta_t)\theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}$$

"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

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; \theta_{1,\ldots,L})) + \frac{\lambda}{2}\sum_l \|\theta_l\|$$

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

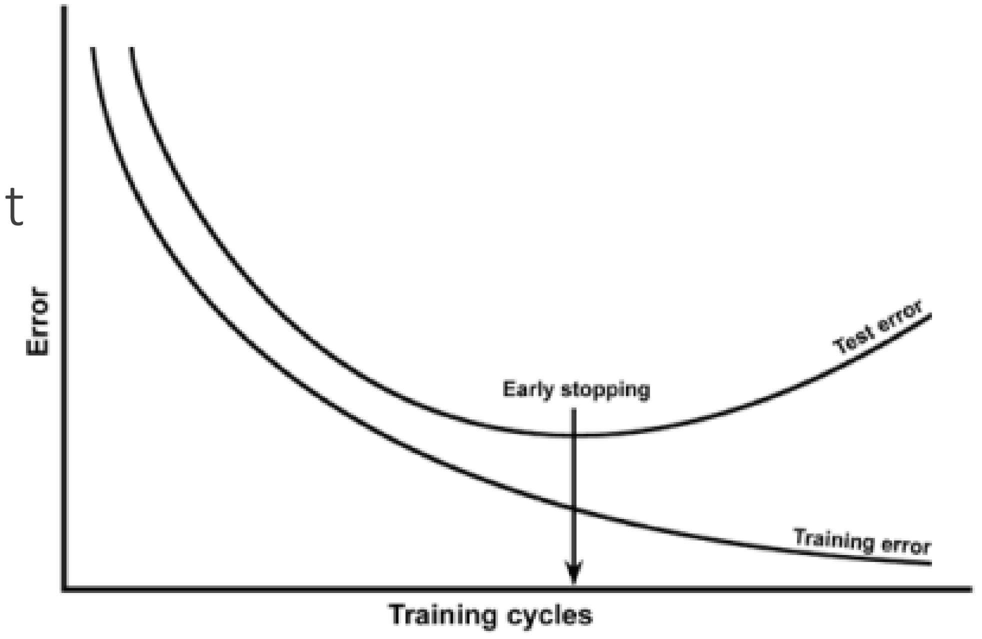
$$\theta^{(t+1)} = \theta^{(t)} - \lambda\eta_t \frac{\theta^{(t)}}{|\theta^{(t)}|} - \eta_t \nabla_\theta \mathcal{L}$$

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

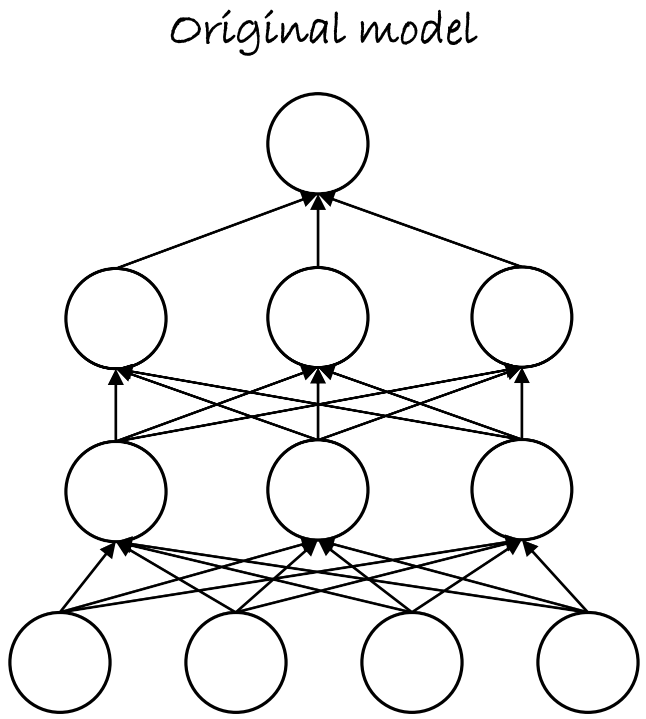


Dropout [Srivastava2014]

- 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

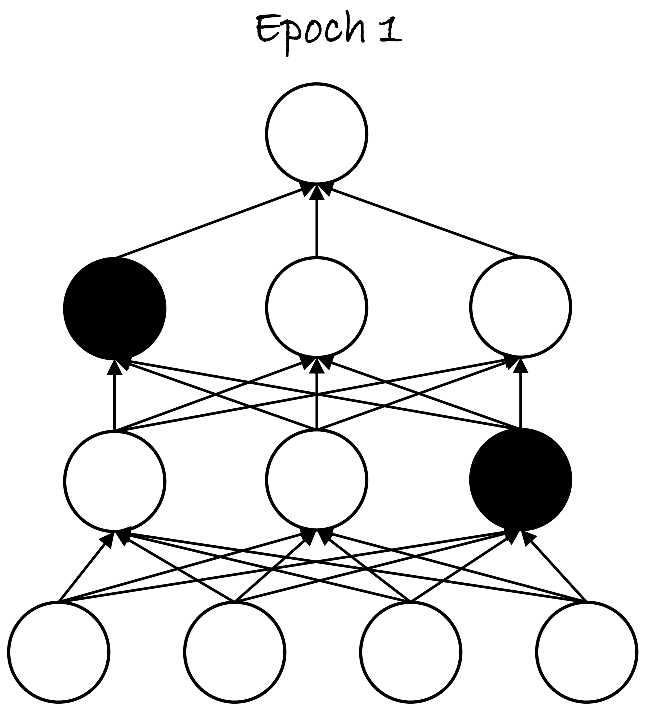
Dropout

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



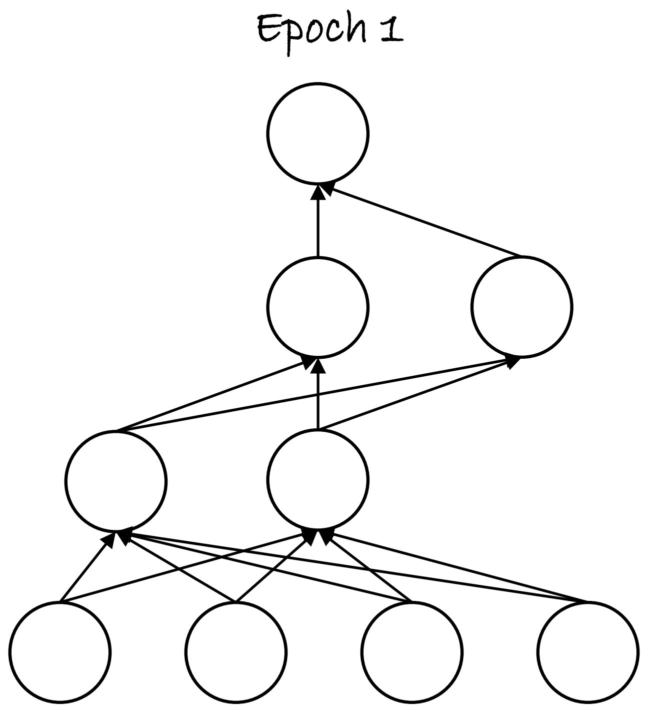
Dropout

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



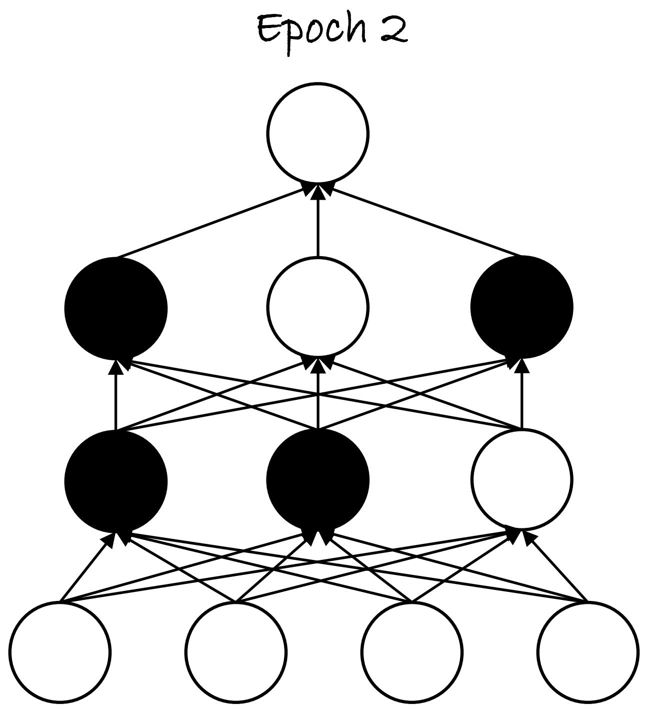
Dropout

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



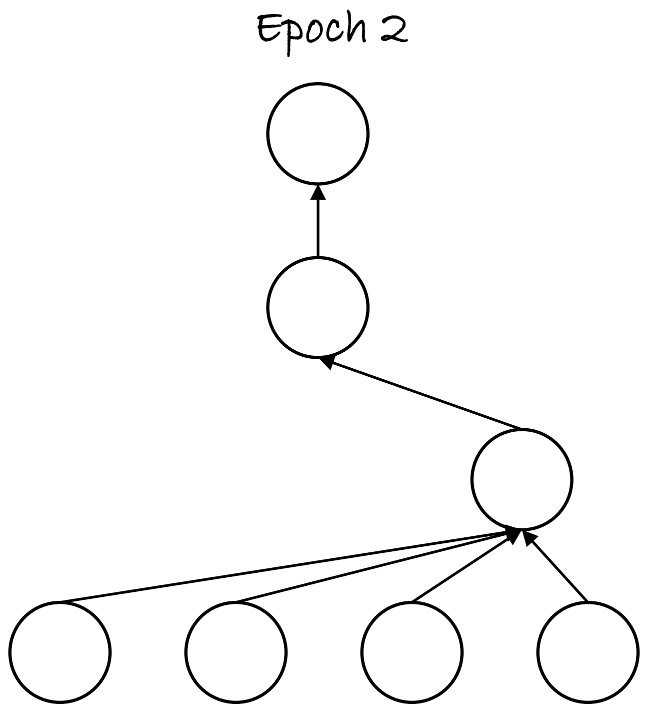
Dropout

- 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



Architectural details

1. The Neural Network

$$\boxed{a_L(x;\theta_{1,\ldots,\mathrm{L}}) = h_L\left(h_{L-1}(\ldots h_1(x,\theta_1),\theta_{L-1}),\theta_L\right)}$$

2. Learning by minimizing empirical error

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

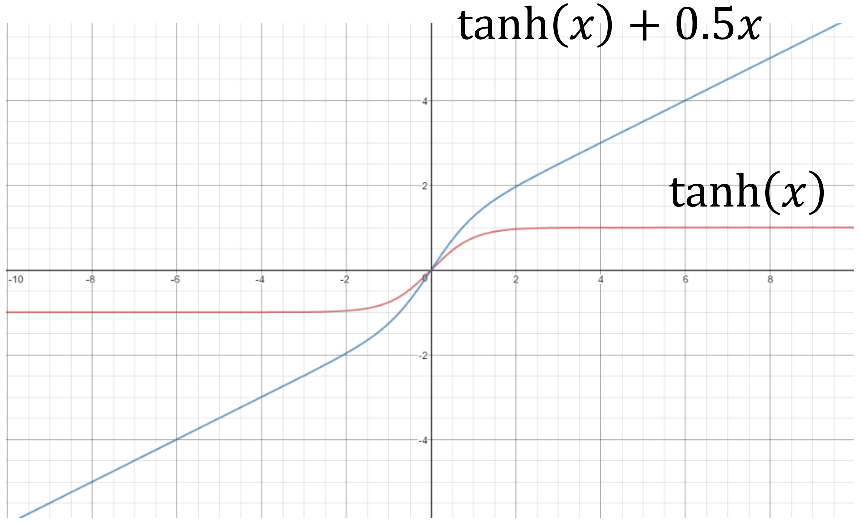
3. Optimizing with Gradient Descent based methods

$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}}$$

Sigmoid-like activation functions

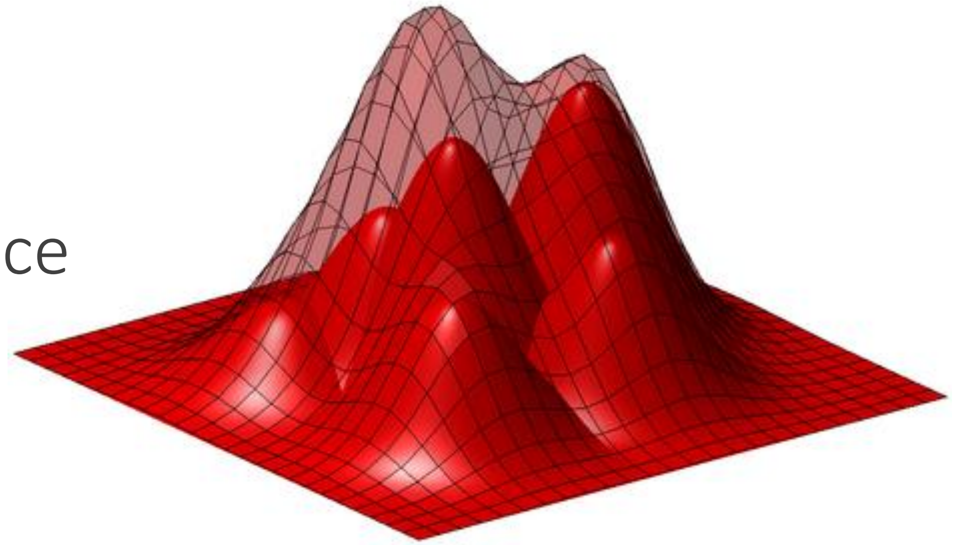
- Straightforward sigmoids not a very good idea
- Symmetric sigmoids converge faster
  - E.g. tanh, returns a(x=0)=0
  - Recommended sigmoid: $a = h(x) = 1.7159 \tanh(\frac{2}{3}x)$
- You can add a linear term to avoid flat areas

$$a = h(x) = \tanh(x) + \beta x$$



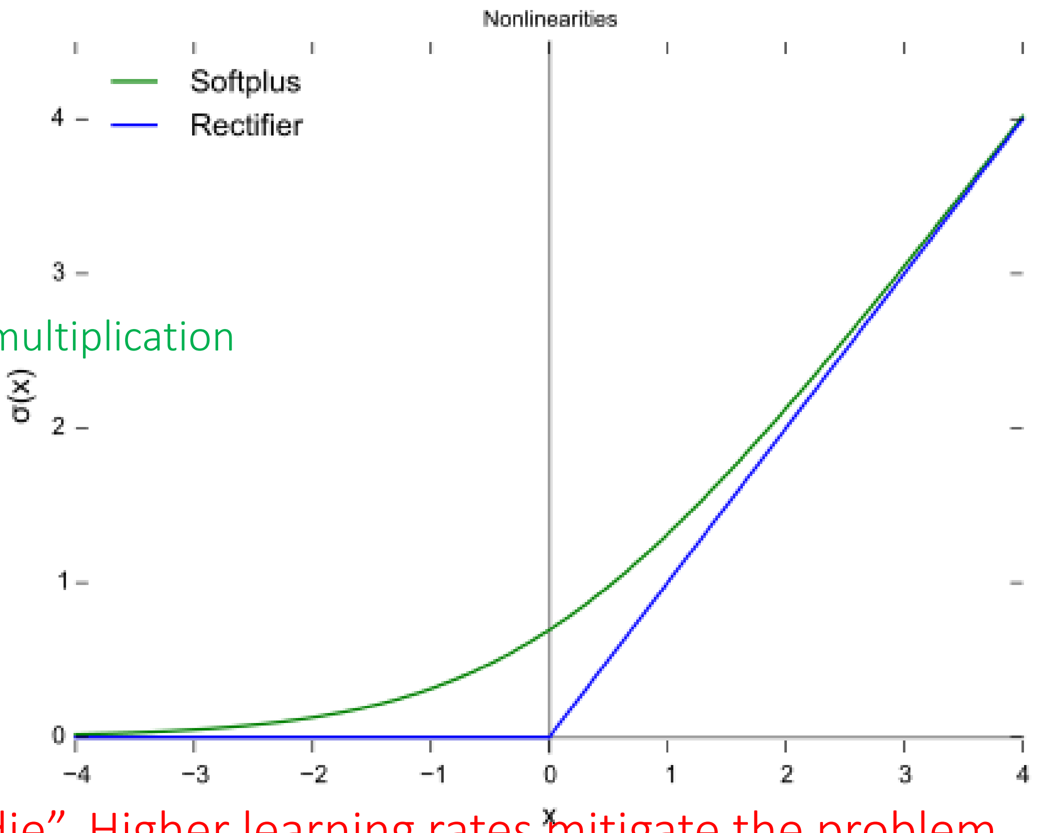
RBFs vs "Sigmoids"

- RBF: $a = h(x) = \sum_j u_j \exp\left(-\beta_j (x - w_j)^2\right)$
- Sigmoid: $a = h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$
- Sigmoids can cover the full feature space
- RBF's are much **more local** in the feature space
  - Can be faster to train but with a more limited range
  - Can give better set of basis functions
  - Preferred in lower dimensional spaces

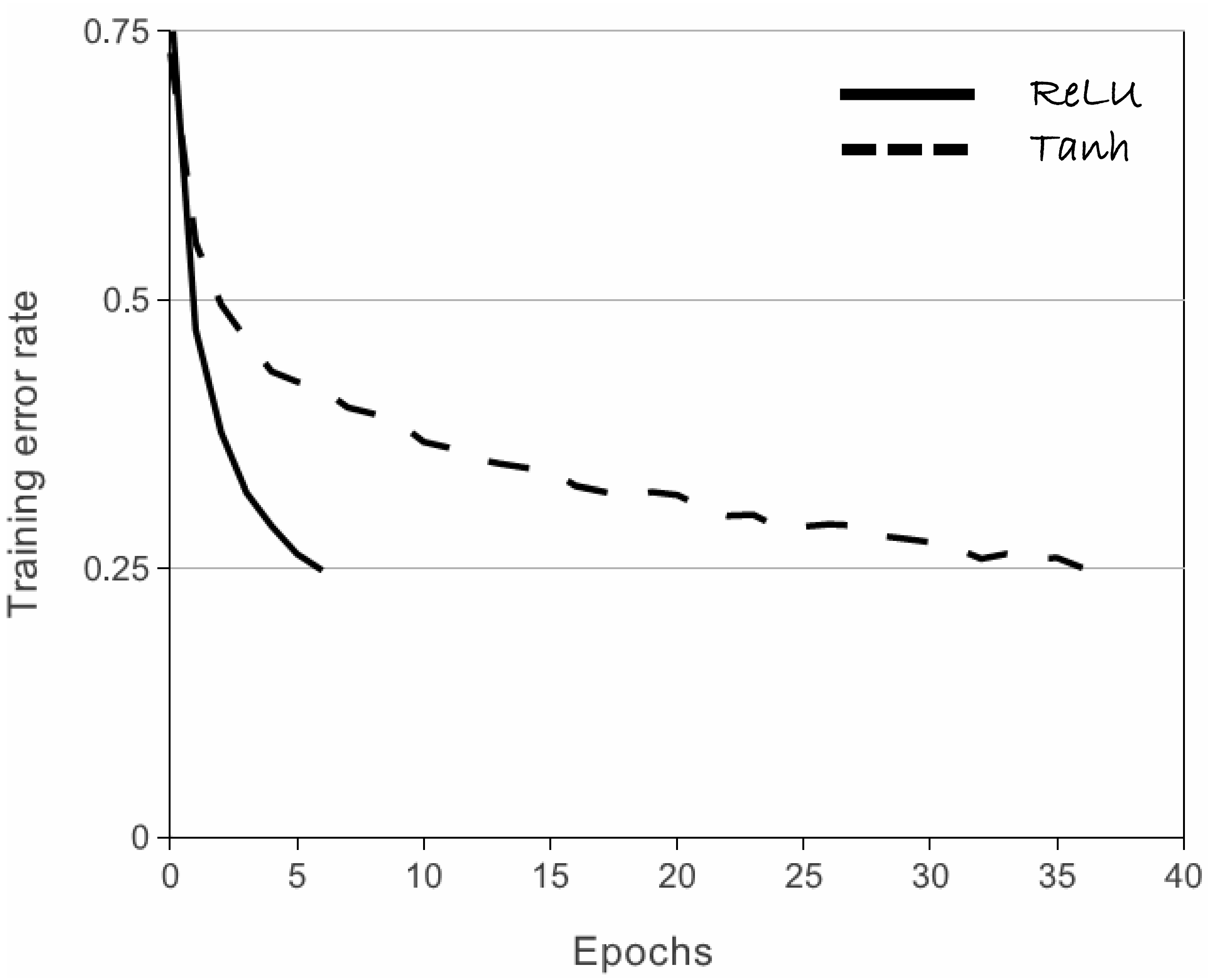


Rectified Linear Unit (ReLU) module [Krizhevsky2012]

- Activation function $a = h(x) = \max(0, x)$
- Gradient wrt the input $\frac{\partial a}{\partial x} = \begin{cases} 0, if\ x \leq 0 \\ 1, if x > 0 \end{cases}$
- Very popular in computer vision and speech recognition
- Much faster computations, gradients
  - No vanishing or exploding problems, only comparison, addition, multiplication
- People claim biological plausibility
- Sparse activations
- No saturation
- Non-symmetric
- Non-differentiable at 0
- A large gradient during training can cause a neuron to “die”. Higher learning rates mitigate the problem



ReLU convergence rate

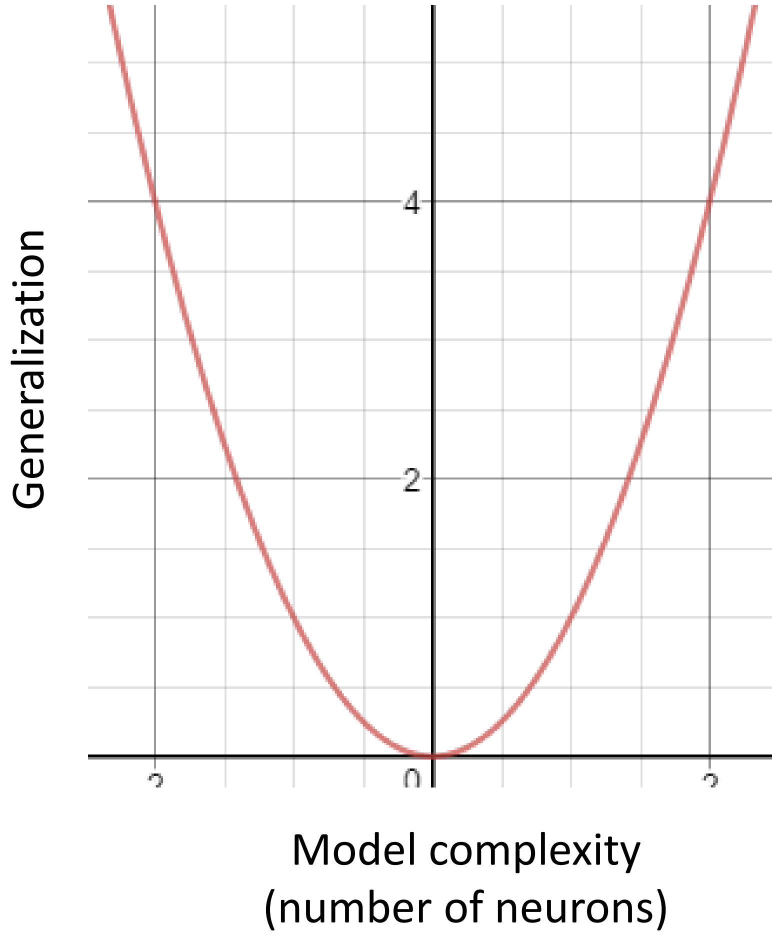


Architectural hyper-parameters

- Number of hidden layers
- Number of neuron in each hidden layer
- Type of activation functions
- Type and amount of regularization

Number of neurons, number of hidden layers

- Dataset dependent hyperparameters
- Tip: **Start small** → increase complexity gradually
  - e.g. start with a 2-3 hidden layers
  - Add more layers → does performance improve?
  - Add more neurons → does performance improve?
- Regularization is very important, use ℓ_2
  - Even if with very deep or wide network
  - With strong ℓ_2-regularization we avoid overfitting



Learning rate

1. The Neural Network

$$\boxed{a_L(x;\theta_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x,\theta_1),\theta_{L-1}),\theta_L)}$$

2. Learning by minimizing empirical error

$$\boxed{\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x;\theta_{1,\ldots,L}))}$$

3. Optimizing with Gradient Descent based methods

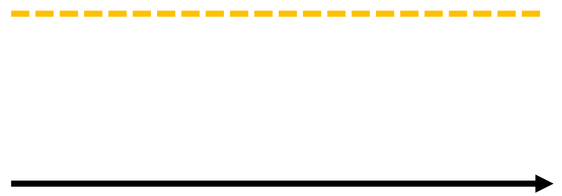
$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}}$$

Learning rate

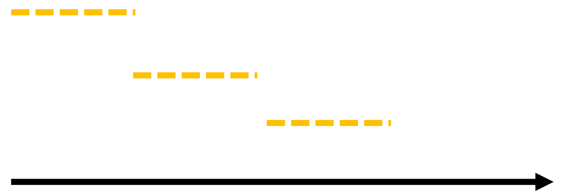
- The right learning rate η_t very important for fast convergence
  - Too strong → gradients overshoot and bounce
  - Too weak, → too small gradients → slow training
- Learning rate per weight is often advantageous
  - Some weights are near convergence, others not
- Rule of thumb
  - Learning rate of (shared) weights prop. to square root of share weight connections
- Adaptive learning rates are also possible, based on the errors observed
  - [Sompolinsky1995]

Learning rate schedules

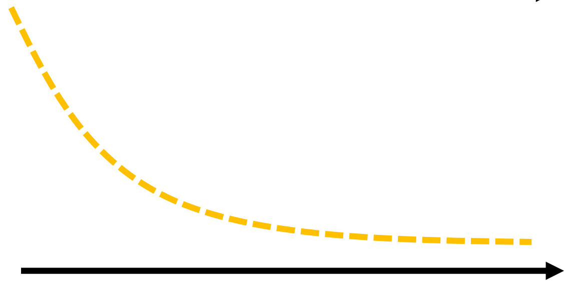
- Constant
  - Learning rate remains the same for all epochs



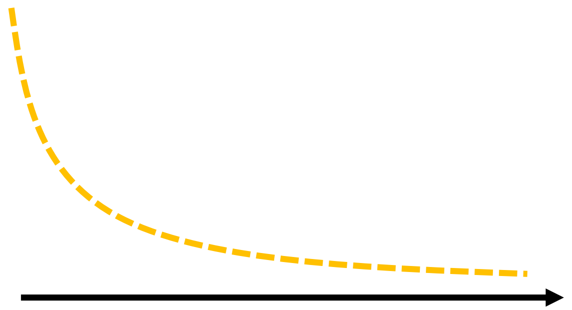
- Step decay
  - Decrease (e.g. η_t/T or η_t/T) every T number of epochs



- 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)

Learning rate 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

1. The Neural Network

$$\boxed{a_L(x;\theta_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x,\theta_1),\theta_{L-1}),\theta_L)}$$

2. Learning by minimizing empirical error

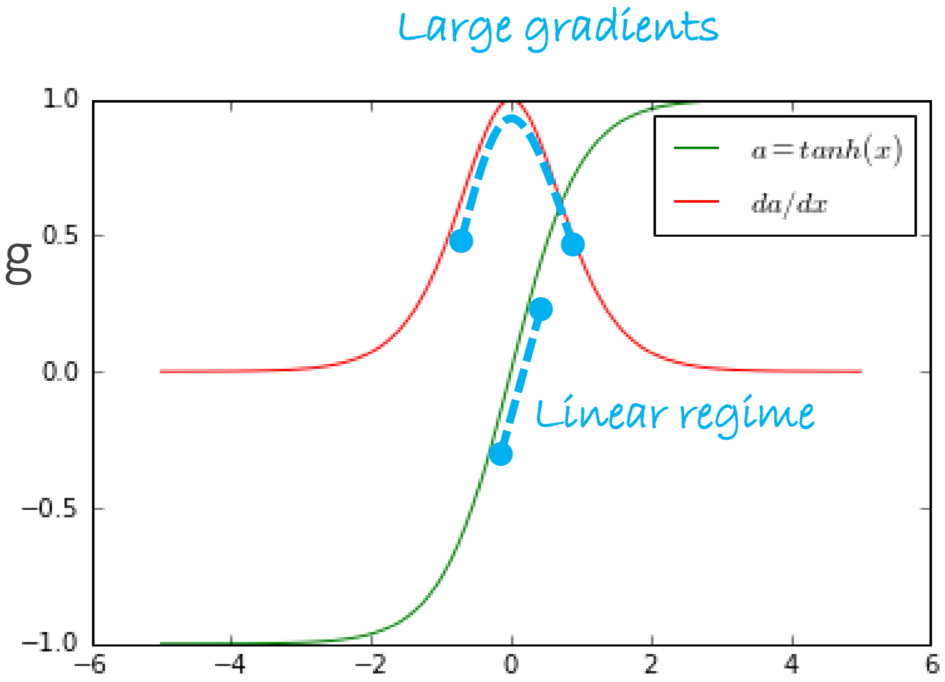
$$\boxed{\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x;\theta_{1,\ldots,L}))}$$

3. Optimizing with Gradient Descent based methods

$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}}$$

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

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

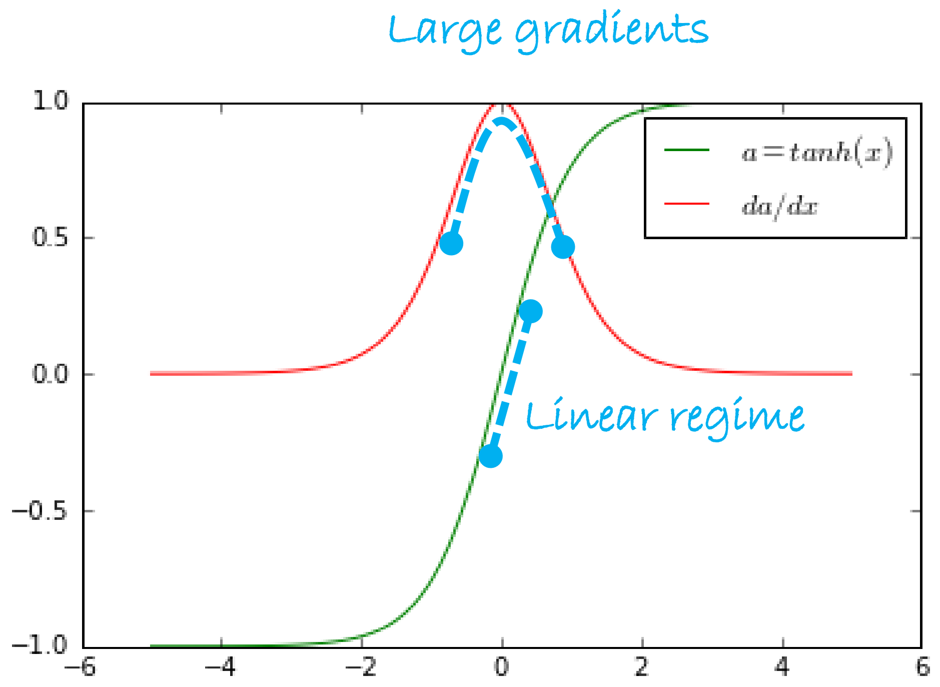
Question: Why similar input/output variance?

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]

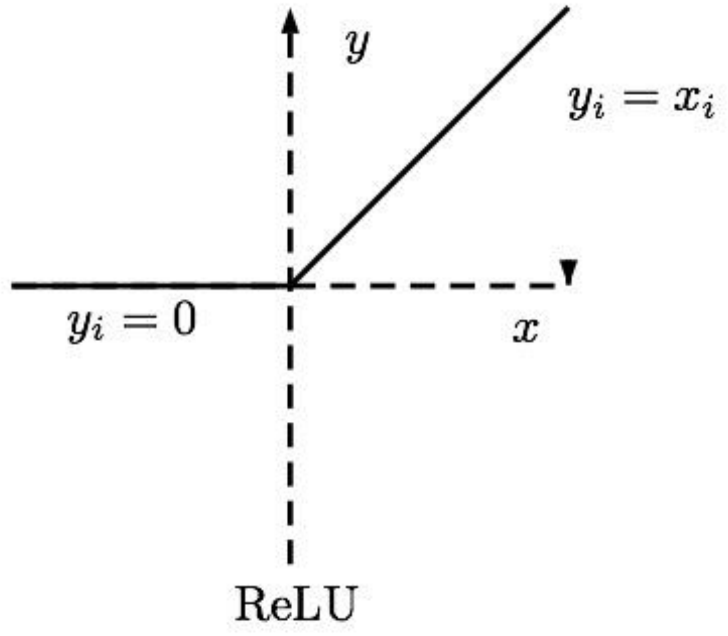
- For $a = \theta x$ the variance is
$$var(a) = E[x]^2 var(\theta) + \mathrm{E}[\theta]^2 var(x) + var(x) var(\theta)$$
- Since $E[x] = E[\theta] = 0$
$$var(a) = var(x) var(\theta) \approx d \cdot var\left(x^i\right) var\left(\theta^i\right)$$
- For $var(a) = var(x) \Rightarrow var\left(\theta^i\right)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

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



Loss functions

1. The Neural Network

$$\boxed{a_L(x; \theta_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}), \theta_L)}$$

2. Learning by minimizing empirical error

$$\boxed{\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; \theta_{1,\ldots,L}))}$$

3. Optimizing with Gradient Descent based methods

$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}}$$

Multi-class classification

- Our samples contains only one class
  - There is only one correct answer per sample
- Negative log-likelihood (cross entropy) + Softmax

$$\mathcal{L}(\theta; x, y) = -\sum_{c=1}^{C} y_c \log a_L^c \quad \text{for all classes } c = 1, \dots, C$$

- Hierarchical softmax when C is very large
- Hinge loss (aka SVM loss)

$$\mathcal{L}(\theta; x, y) = \sum_{\substack{c=1 \\ c \neq y}}^{C} \max(0, a_L^c - a_L^y + 1)$$

- Squared hinge loss

Is it a cat? Is it a horse? ...

Multi-class, multi-label classification

- Each sample can have many correct answers
- Hinge loss and the likes
  - Also sigmoids would also work
- Each output neuron is independent
  - "Does this contain a car, <u>yes</u> or no?"
  - "Does this contain a person, <u>yes</u> or no?"
  - "Does this contain a motorbike, <u>yes</u> or no?"
  - "Does this contain a horse, yes or <u>no</u>?"
- Instead of "Is this a car, motorbike or person?"
  - $p(car|x) = 0.55, p(m/bike|x) = 0.25, p(person|x) = 0.15, p(horse|x) = 0.05$
  - $p(car|x) + p(m/bike|x) + p(person|x) + p(horse|x) = 1.0$

Regression

- The good old Euclidean Loss

$$\mathcal{L}(\theta; x, y) = \frac{1}{2} |y - a_L|_2^2$$

- Or RBF on top of Euclidean loss

$$\mathcal{L}(\theta; x, y) = \sum\nolimits_j u_j \exp(-\beta_j (y - a_L)^2)$$

- Or ℓ_1 distance

$$\mathcal{L}(\theta; x, y) = \sum_j |y_j - a_L^j|$$

Even better optimizations

1. The Neural Network

$$\boxed{a_L(x; \theta_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}), \theta_L)}$$

2. Learning by minimizing empirical error

$$\boxed{\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; \theta_{1,\ldots,L}))}$$

3. Optimizing with Gradient Descent based methods

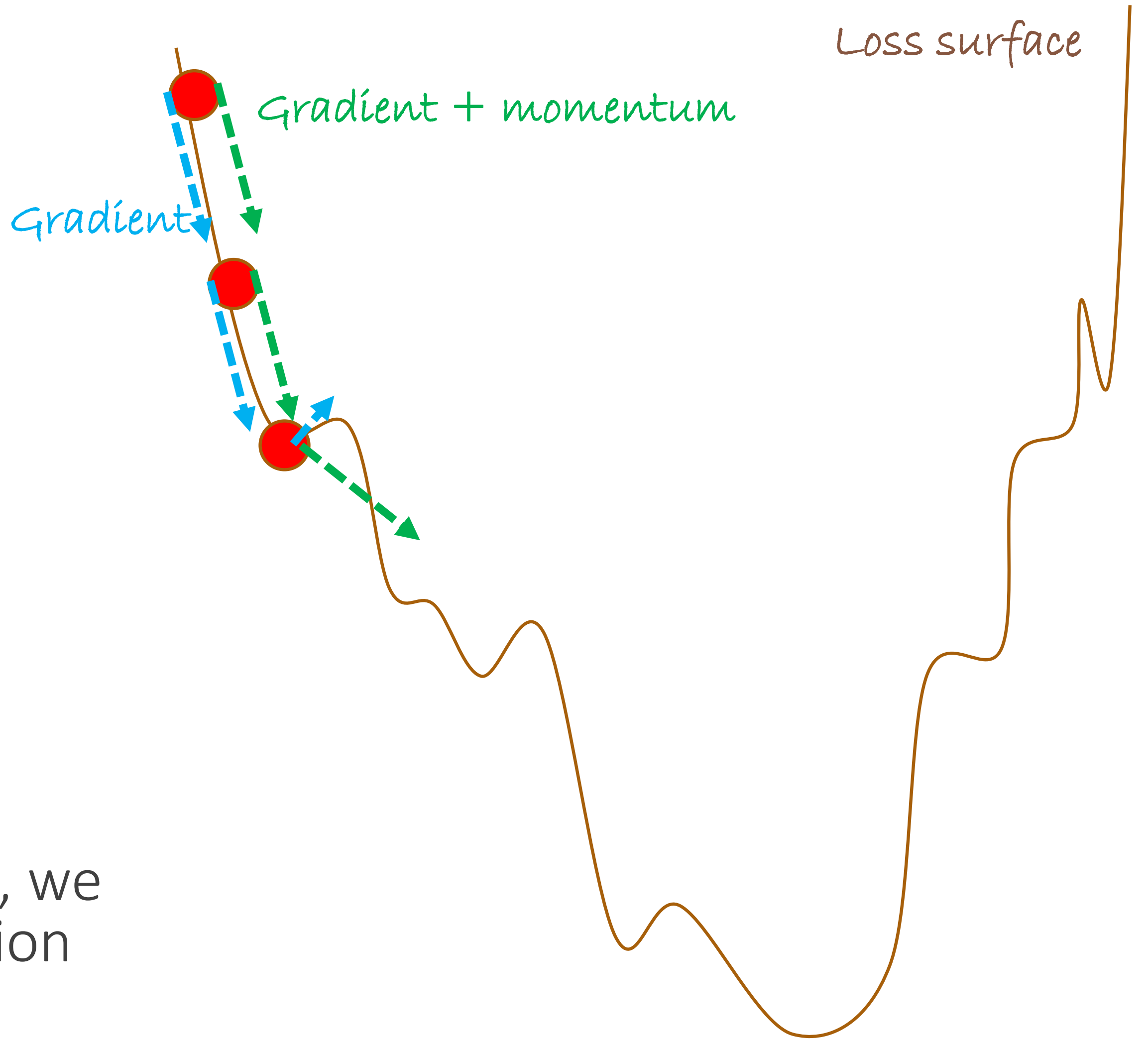
$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}}$$

Momentum

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

$$u^{(t+1)} = \gamma u^{(t)} - \eta_t \nabla_\theta \mathcal{L}$$
$$\theta^{(t+1)} = \theta^{(t)} + u^{(t+1)}$$

- More robust gradients and learning → faster convergence
- Nice "physics"-based interpretation
  - Instead of updating the position of the "ball", we update the velocity, which updates the position

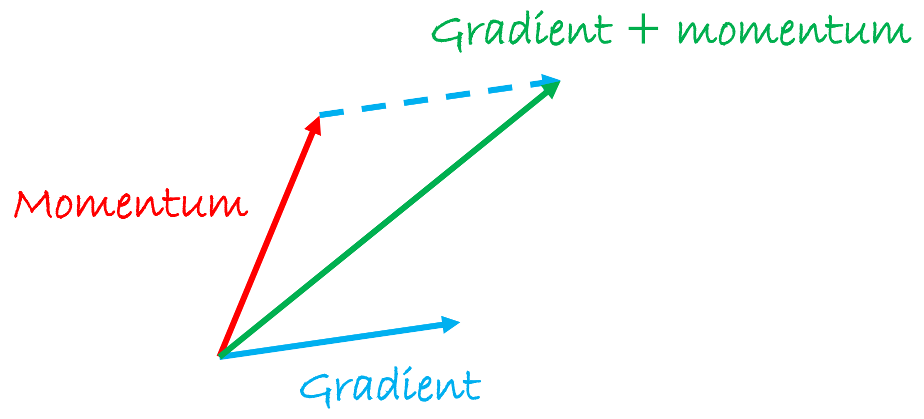


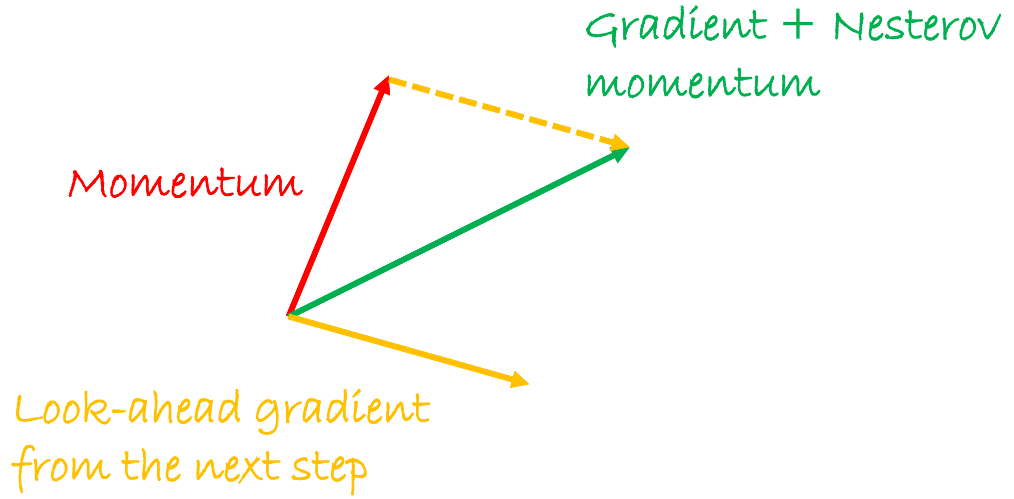
Nesterov Momentum [Sutskever2013]

- Use the future gradient instead of the current gradient

$$\theta^{(t+0.5)} = \theta^{(t)} + \gamma u^{(t)}$$
$$u^{(t+1)} = \gamma u^{(t)} - \eta_t \nabla_{\theta^{(t+0.5)}} \mathcal{L}$$
$$\theta^{(t+1)} = \theta^{(t)} + u^{(t+1)}$$

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




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

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

$$\theta^{(t+1)} = \theta^{(t)} - H_{\mathcal{L}}^{-1} \eta_t \nabla_\theta \mathcal{L}$$

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

$$H_{\mathcal{L}}^{ij} = \frac{\partial \mathcal{L}}{\partial \theta_i \partial \theta_j}$$

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

Other per-parameter adaptive optimizations

- Adagrad [Duchi2011]
- RMSprop
- Adam [Kingma2014]

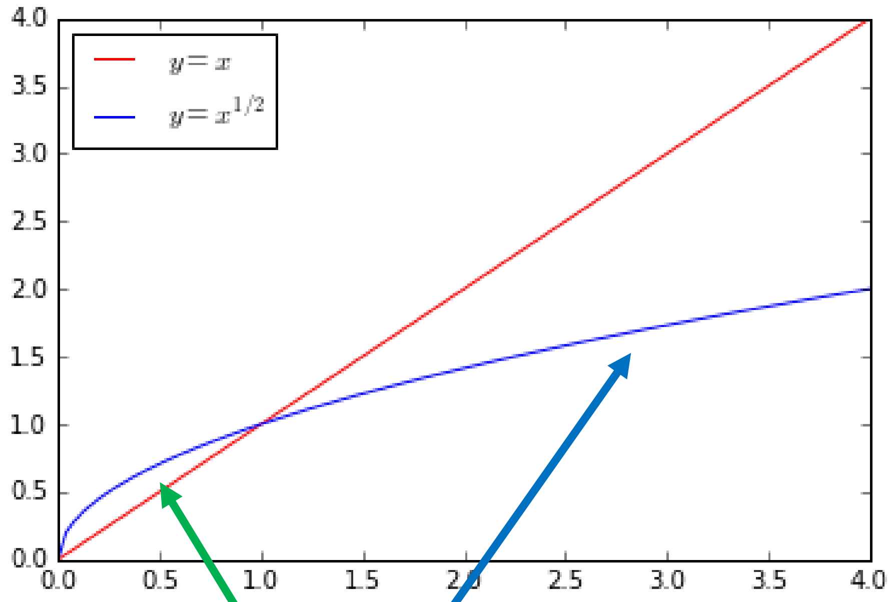
Adagrad [Duchi2011]

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

RMSprop

Decay hyper-parameter

- Schedule
  - $r = \alpha \sum_{\tau=1}^{t-1} (\nabla_\theta^{(t)} \mathcal{L}_j)^2 + (1 - \alpha) \nabla_\theta^{(t)} \mathcal{L}_j \quad \Longrightarrow$
  - $\theta^{(t+1)} = \theta^{(t)} - \eta_t \frac{\nabla_\theta \mathcal{L}}{\sqrt{r} + \varepsilon}$
- Moving average of the squared gradients
  - Compared to Adagrad
- **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



Square rooting boosts small values while suppresses large values

Adam [Kingma2014]

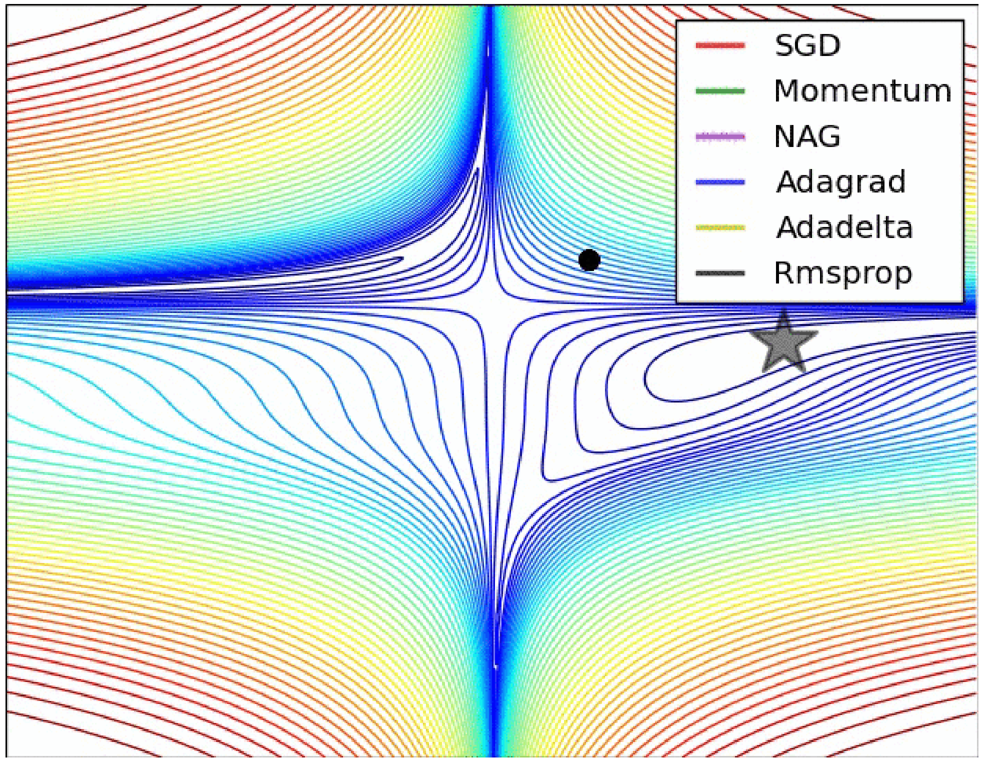
- One of the most popular learning algorithms

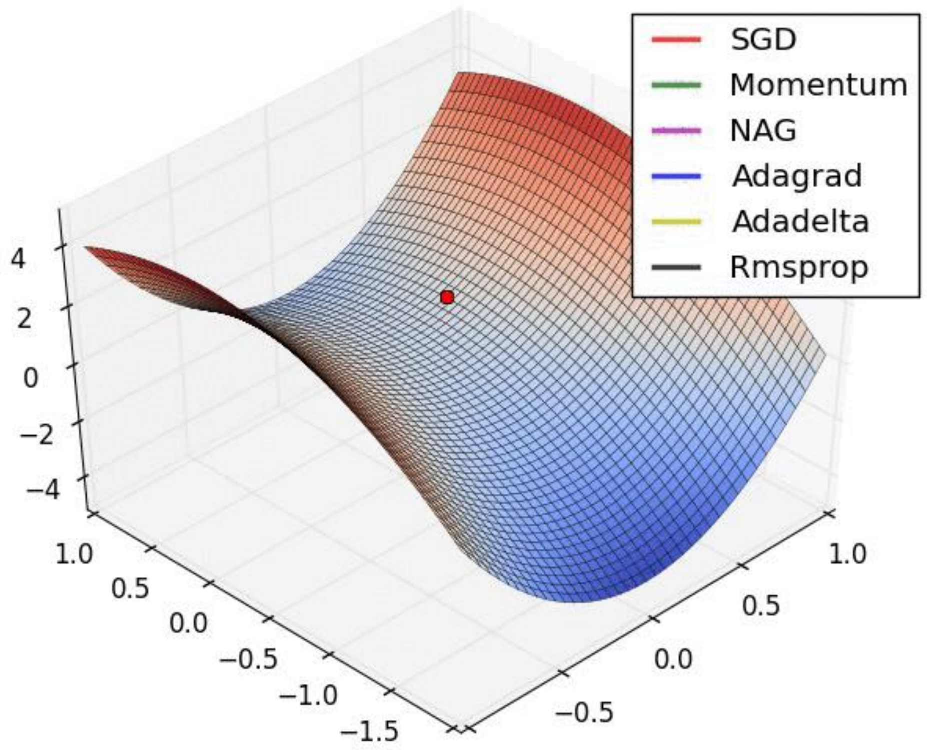
$$g_t = \nabla_\theta \mathcal{L}$$
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \varepsilon}$$

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

- Similar to RMSprop, but with momentum & correction bias

Visual overview





Picture credit: Alec Radford

Another visualization

- https://habrahabr.ru/post/318970/

Learning –not computing– the gradients

- Learning to learn by gradient descent by gradient descent
  - [Andrychowicz2016]
- $\theta^{(t+1)} = \theta^{(t)} + g_t(\nabla_\theta \mathcal{L}, \varphi)$
- g_t is an “optimizer” with its own parameters φ
  - Implemented as a recurrent network

Good practice

- 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

Babysitting Deep Nets

1. The Neural Network

$$\boxed{a_L(x;\theta_{1,\ldots,L}) = h_L\left(h_{L-1}(\ldots h_1(x,\theta_1),\theta_{L-1}),\theta_L\right)}$$

2. Learning by minimizing empirical error

$$\boxed{\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x;\theta_{1,\ldots,L}))}$$

3. Optimizing with Gradient Descent based methods

$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}}$$

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

Summary

- 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

Reading material & references

- Chapter 8, 11

Next lecture

- What are the Convolutional Neural Networks?
- Why are they important in Computer Vision?
- Differences from standard Neural Networks
- How to train a Convolutional Neural Network?