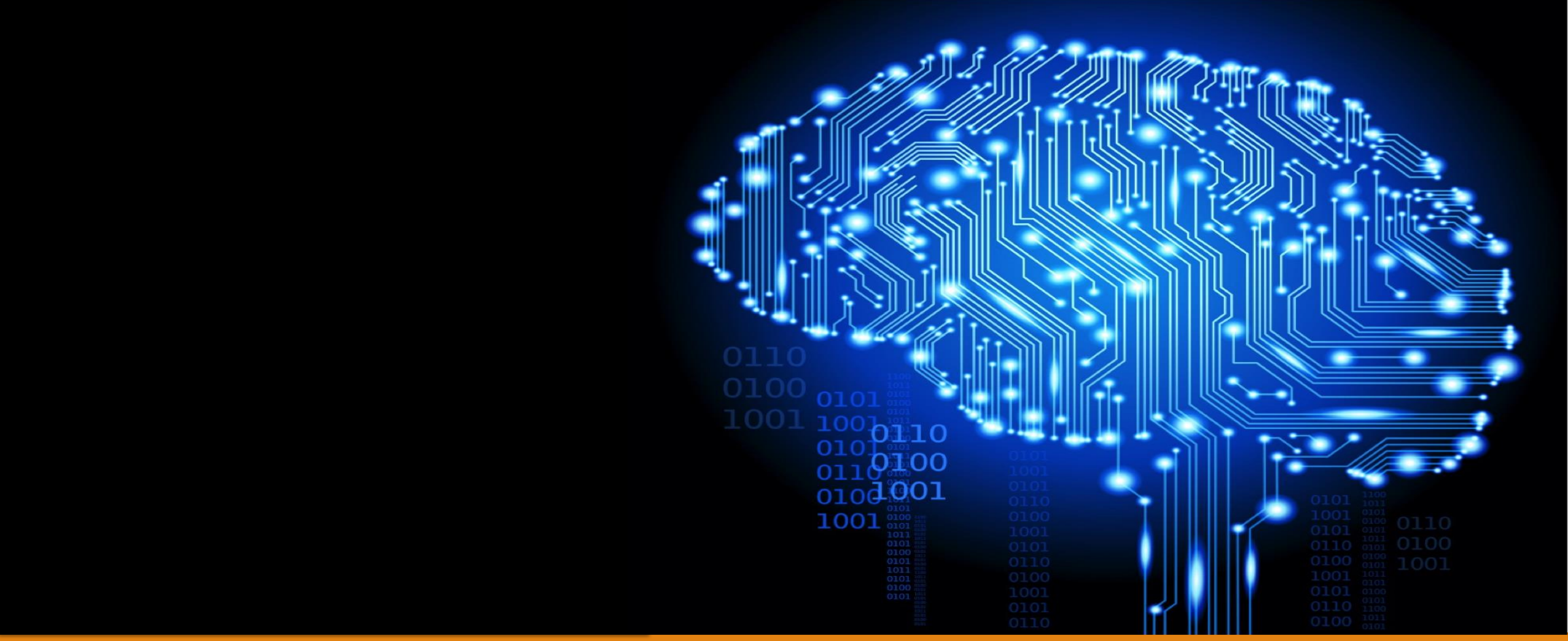


Lecture 3: Deeper into Deep Learning and Optimizations

Deep Learning @ UvA

Previous lecture

- Machine Learning Paradigm for Neural Networks
- The Backpropagation algorithm for learning with a neural network
- Neural Networks as modular architectures
- Various Neural Network modules
- How to implement and check your very own module

Lecture overview

- How to defining 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

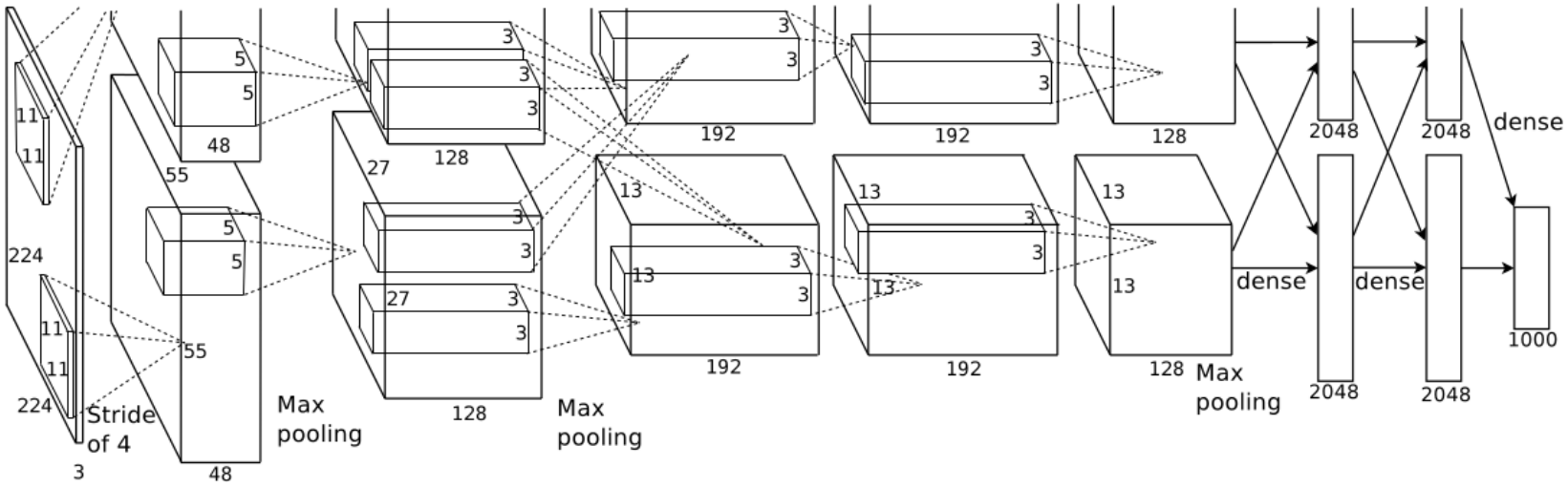


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input 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

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

2. Learning by minimizing empirical error

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

3. Optimizing with Gradient Descend based methods

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

SGD vs GD

1. The Neural Network

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

2. Learning by minimizing empirical error

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

3. Optimizing with Gradient Descend based methods

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

Backpropagation again

- **Step 1.** Compute forward propagations for all layers, starting from the first layer until the last loss layer

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

- **Step 2.** Once done with forward propagation, follow the reverse path. Start from the last layer and for each new layer compute the gradients

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

  - Cache computations when possible to avoid redundant operations

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

Backpropagation again

- **Step 1.** Compute forward propagations for all layers, starting from the first layer until the last loss layer

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

- **Step 2.** Once done with forward propagation, follow the reverse path. Start from the last layer and for each new layer compute the gradients

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

  - Cache computations when possible to avoid redundant operations

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

Vector with dimensions $[d_l \times 1]$

Vector with dimensions $[d_{l+1} \times 1]$

Jacobian matrix with dimensions $[d_{l+1} \times d_l]$

Backpropagation again

- **Step 1.** Compute forward propagations for all layers, starting from the first layer until the last loss layer

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

- **Step 2.** Once done with forward propagation, follow the reverse path. Start from the last layer and for each new layer compute the gradients

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

  - Cache computations when possible to avoid redundant operations

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

Vector with dimensions $[d_l \times 1]$

Vector with dimensions $[d_{l+1} \times 1]$

Jacobian matrix with dimensions $[d_{l+1} \times d_l]$

Matrix with dimensions $[d_l \times d_{l-1}]$

Vector with dimensions $[d_l \times 1]$

Vector with dimensions $[1 \times d_{l-1}]$

Practical example and dimensionality analysis

- Layer $l-1$ has 15 neurons ($d_{l-1}=15$), l has 10 neurons ($d_l=10$) and $l+1$ has 5 neurons ($d_{l+1}=5$)
- My activation functions are $a_l = w_l x_l$ and $a_{l+1} = w_{l+1} x_{l+1}$
- The dimensionalities are *(remember $x_l = a_{l-1}$)*
  - $a_{l-1} \rightarrow [15 \times 1],\ a_l \rightarrow [10 \times 1],\ a_{l+1} \rightarrow [5 \times 1]$
  - $x_l \rightarrow [15 \times 1],\ x_{l+1} \rightarrow [10 \times 1]$
  - $\theta_l \rightarrow [10 \times 15], w_{l+1} \rightarrow [5 \times 10]$
- The gradients are
  - $\frac{\partial \mathcal{L}}{\partial a_l} \rightarrow [10 \times 5] \cdot [5 \times 1] = [10 \times 1]$
  - $\frac{\partial \mathcal{L}}{\partial \theta_l} \rightarrow [10 \times 1] \cdot [1 \times 15] = [10 \times 15]$

Practical example and dimensionality analysis

- Layer $l-1$ has 15 neurons ($d_{l-1}=15$), l has 10 neurons ($d_l=10$) and $l+1$ has 5 neurons ($d_{l+1}=5$)
- My activation functions are $a_l = w_l x_l$ and $a_{l+1} = w_{l+1} x_{l+1}$
- The dimensionalities are *(remember $x_l = a_{l-1}$)*
  - $a_{l-1} \rightarrow [15 \times 1],\ a_l \rightarrow [10 \times 1],\ a_{l+1} \rightarrow [5 \times 1]$
  - $x_l \rightarrow [15 \times 1],\ x_{l+1} \rightarrow [10 \times 1]$
  - $\theta_l \rightarrow [10 \times 15], w_{l+1} \rightarrow [5 \times 10]$
- The gradients are
  - $\frac{\partial \mathcal{L}}{\partial a_l} \rightarrow [10 \times 5] \cdot [5 \times 1] = [10 \times 1]$
  - $\frac{\partial \mathcal{L}}{\partial \theta_l} \rightarrow [10 \times 1] \cdot [1 \times 15] = [10 \times 15]$

Practical example and dimensionality analysis

- Layer $l-1$ has 15 neurons ($d_{l-1}=15$), l has 10 neurons ($d_l=10$) and $l+1$ has 5 neurons ($d_{l+1}=5$)
- My activation functions are $a_l = w_l x_l$ and $a_{l+1} = w_{l+1} x_{l+1}$
- The dimensionalities are *(remember $x_l = a_{l-1}$)*
  - $a_{l-1} \rightarrow [15 \times 1]$, $a_l \rightarrow [10 \times 1]$, $a_{l+1} \rightarrow [5 \times 1]$
  - $x_l \rightarrow [15 \times 1]$, $x_{l+1} \rightarrow [10 \times 1]$
  - $\theta_l \rightarrow [10 \times 15]$, $w_{l+1} \rightarrow [5 \times 10]$
- The gradients are
  - $\frac{\partial \mathcal{L}}{\partial a_l} \rightarrow [10 \times 5] \cdot [5 \times 1] = [10 \times 1]$
  - $\frac{\partial \mathcal{L}}{\partial \theta_l} \rightarrow [10 \times 1] \cdot [1 \times 15] = [10 \times 15]$

Still, backpropagation can be slow

- Often loss surfaces are
  - non-quadratic
  - highly non-convex
  - very high-dimensional
- No real guarantee that
  - the final solution will be good
  - we converge fast to final solution
  - or that there will be convergence
- How can we protect ourselves better?

Stochastic Gradient Descend (SGD)

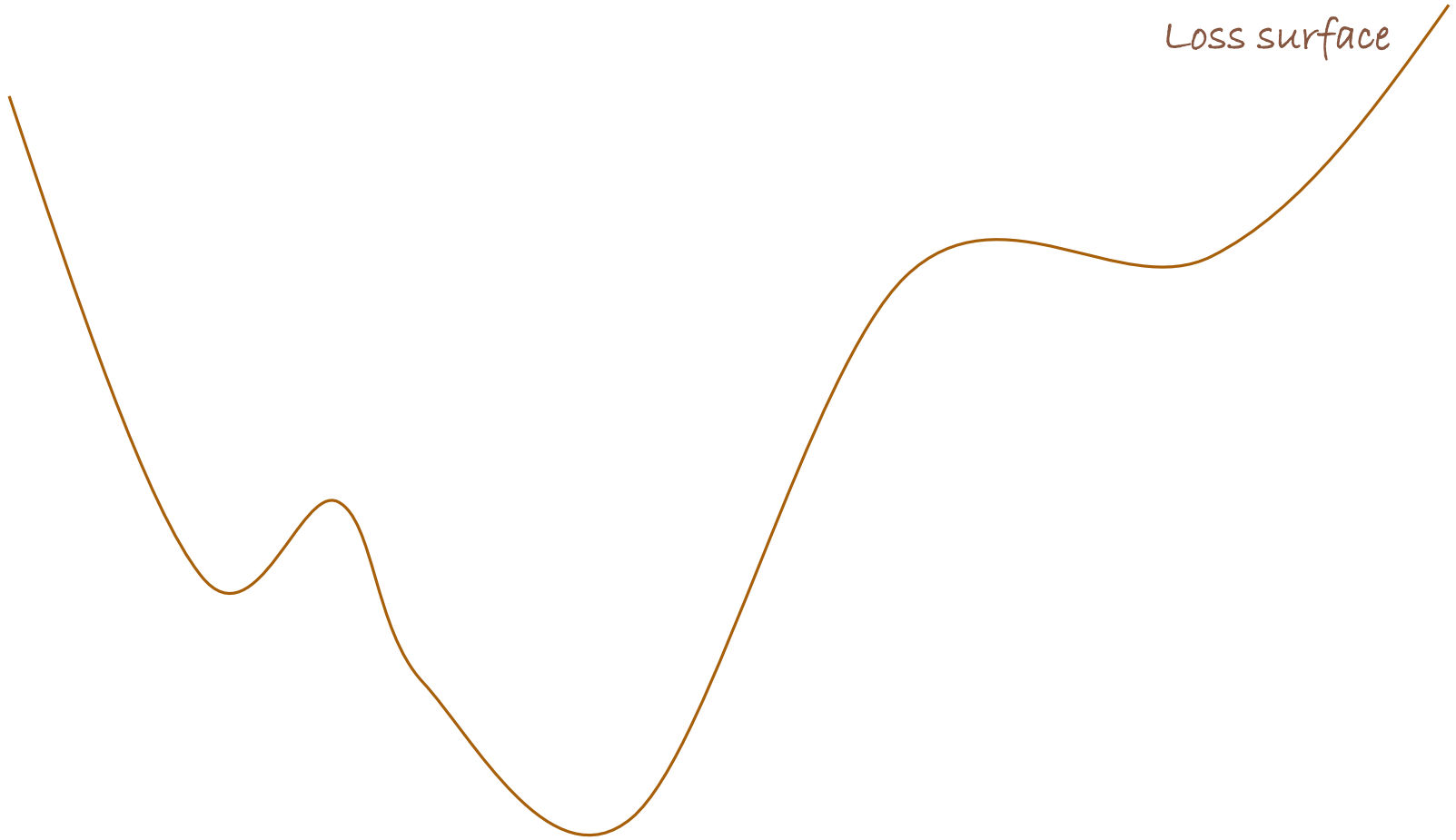
- Stochastically sample "mini-batches" from dataset D
  - The size of B_j can contain even just 1 sample

$$B_j = sample(D)$$

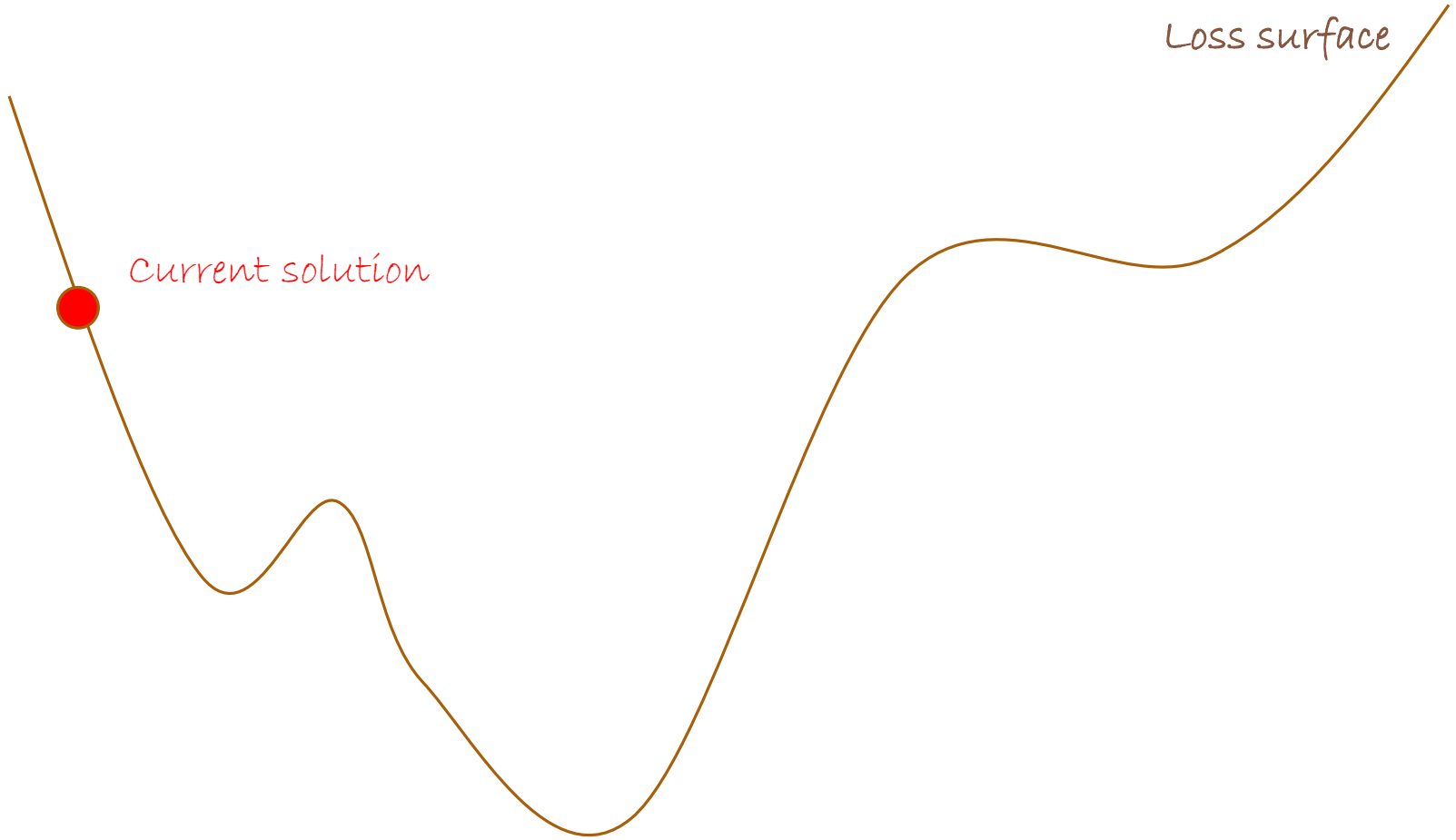
$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta_t}{|B_j|} \sum_{i \in B_j} \nabla_\theta \mathcal{L}_i$$

- Much faster than Gradient Descend
- Results are often better
- Can be used for dynamically changed datasets

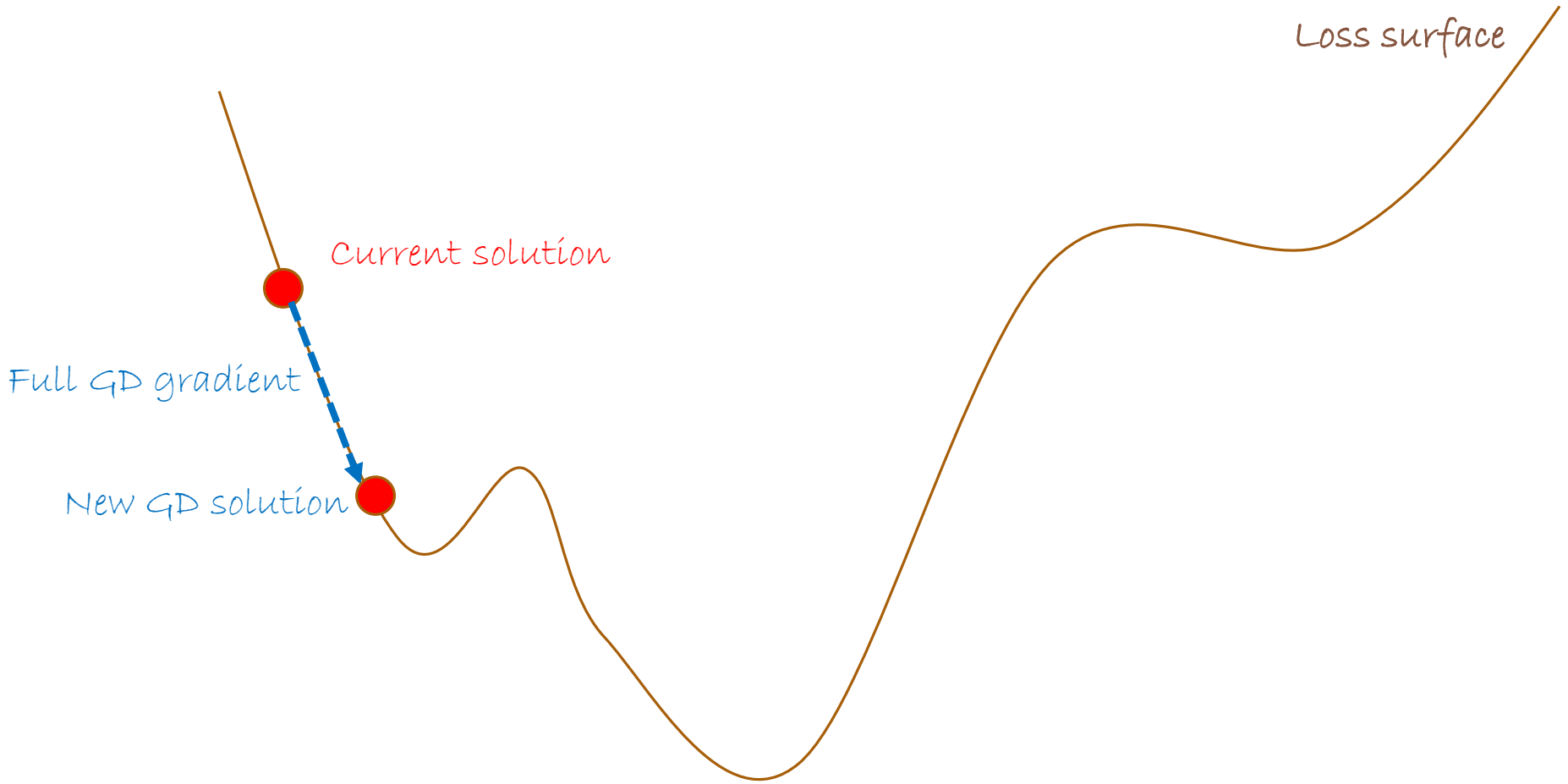
SGD is often better



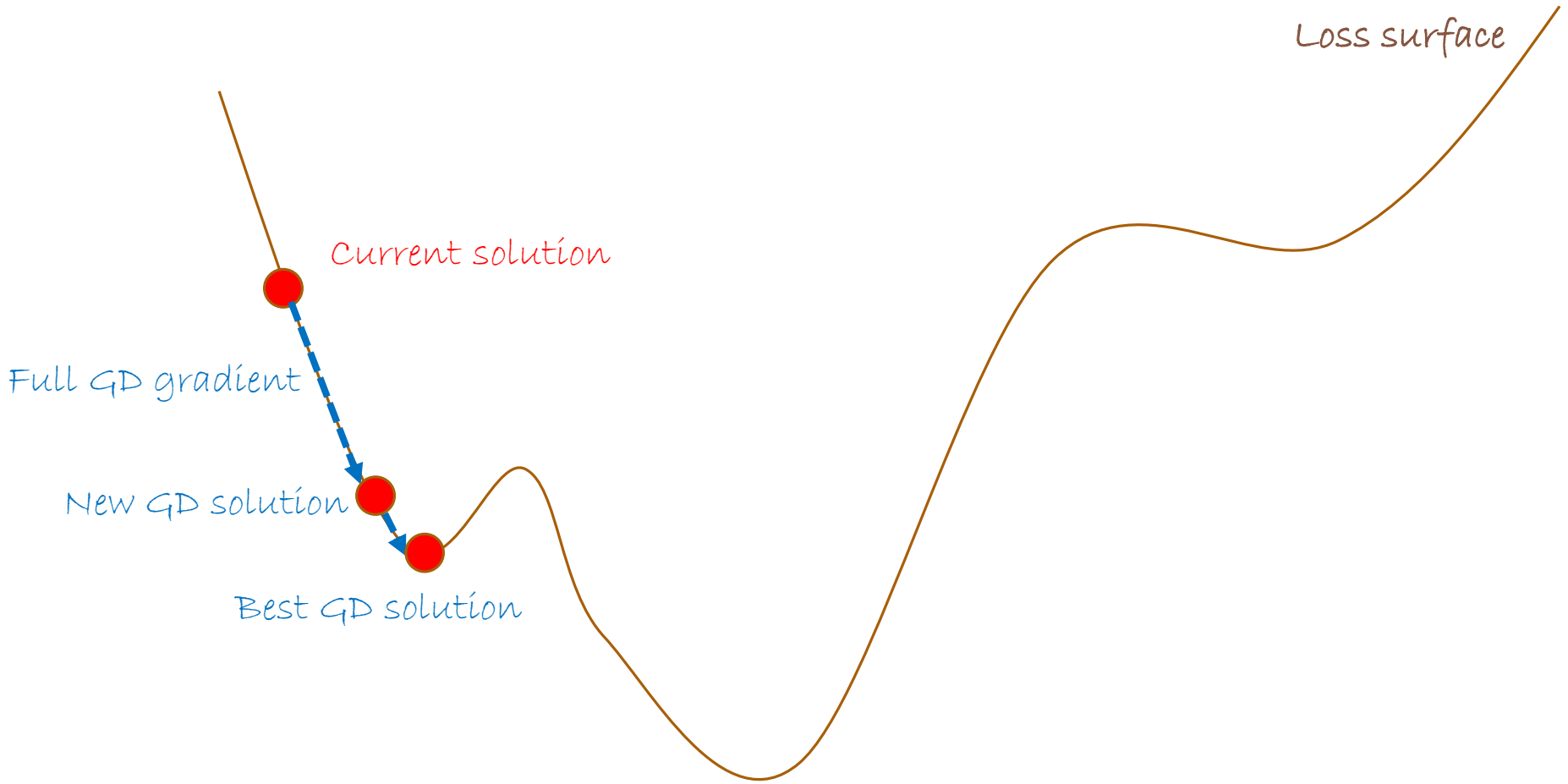
SGD is often better



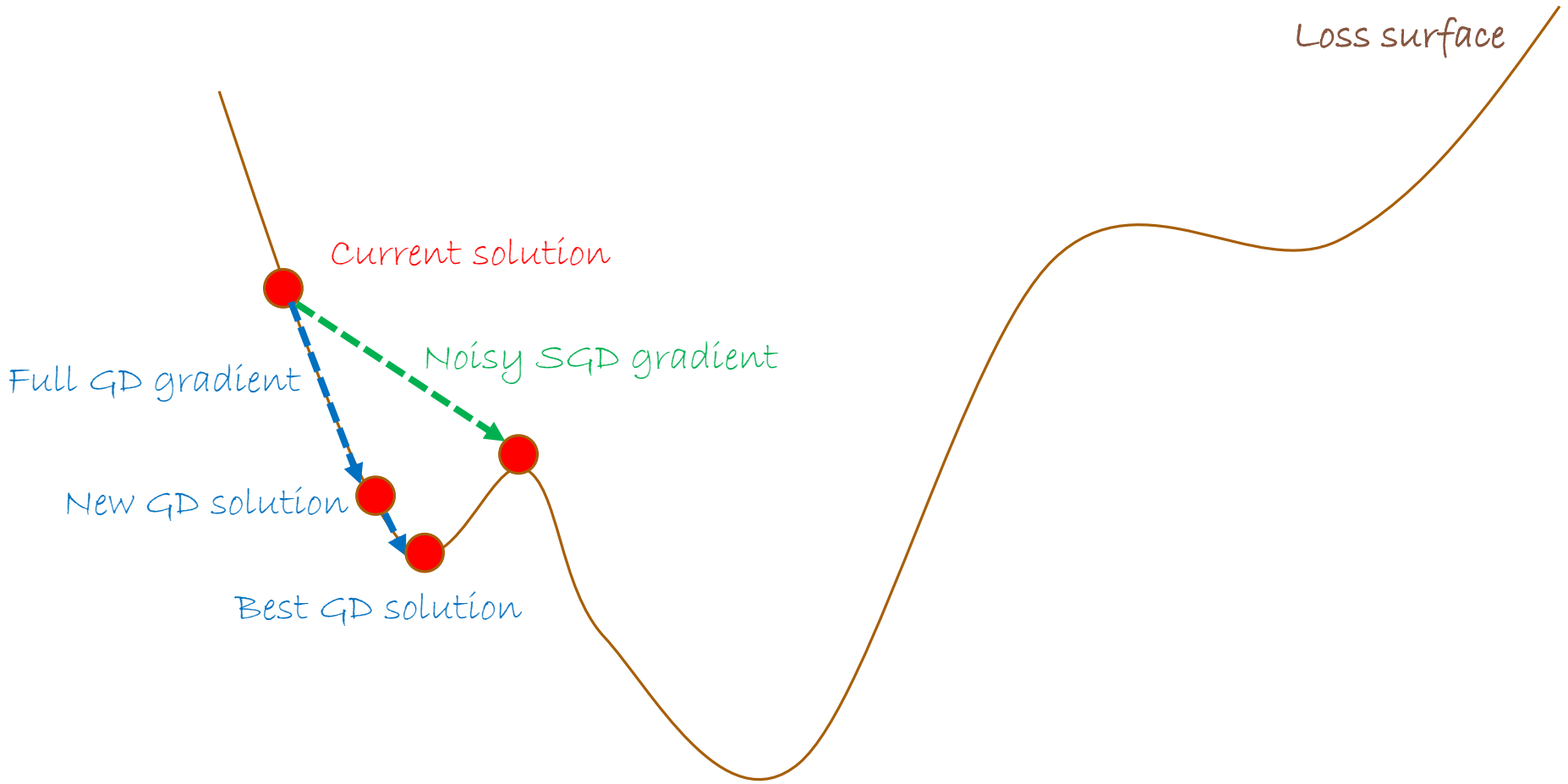
SGD is often better



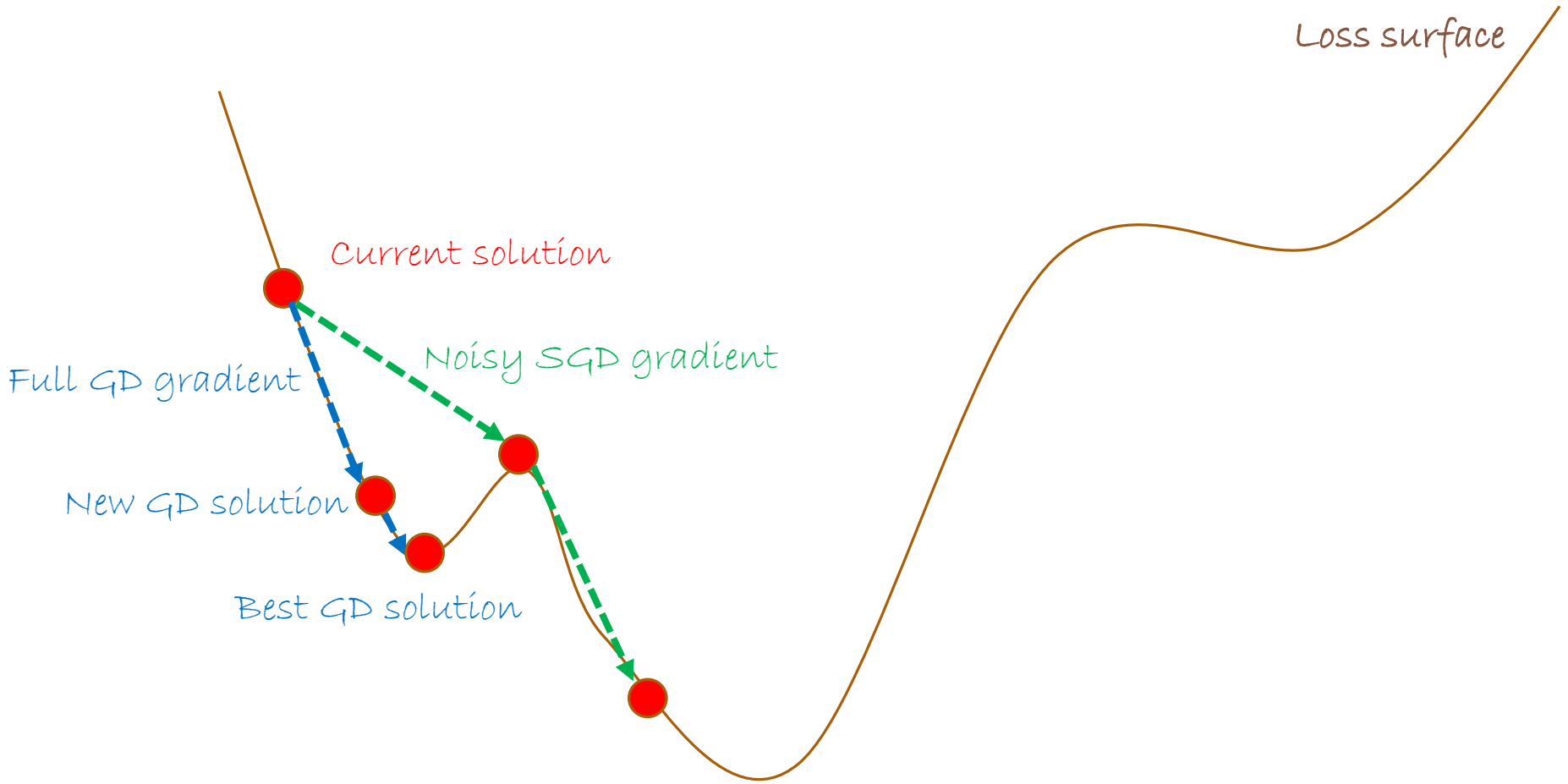
SGD is often better



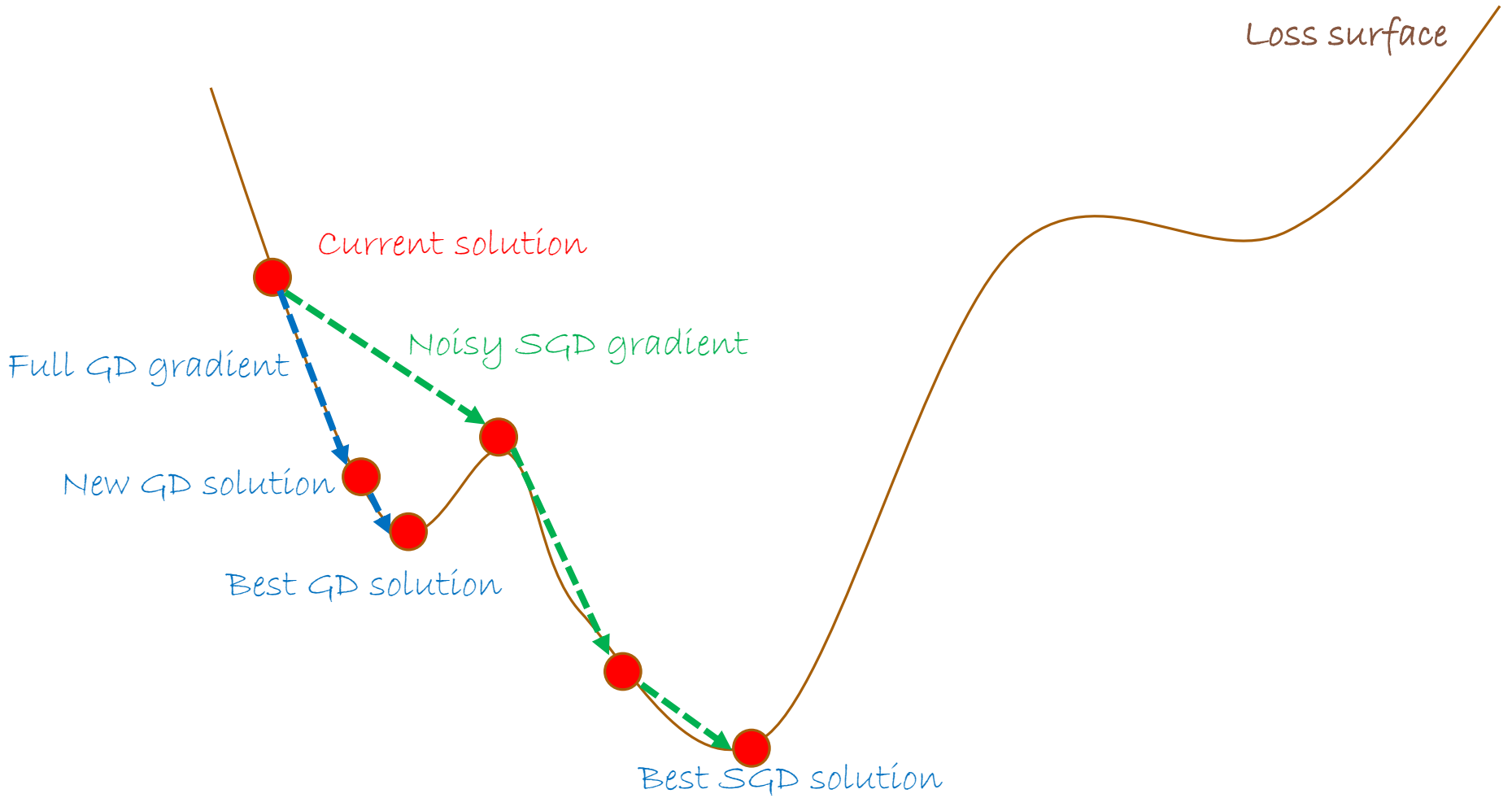
SGD is often better



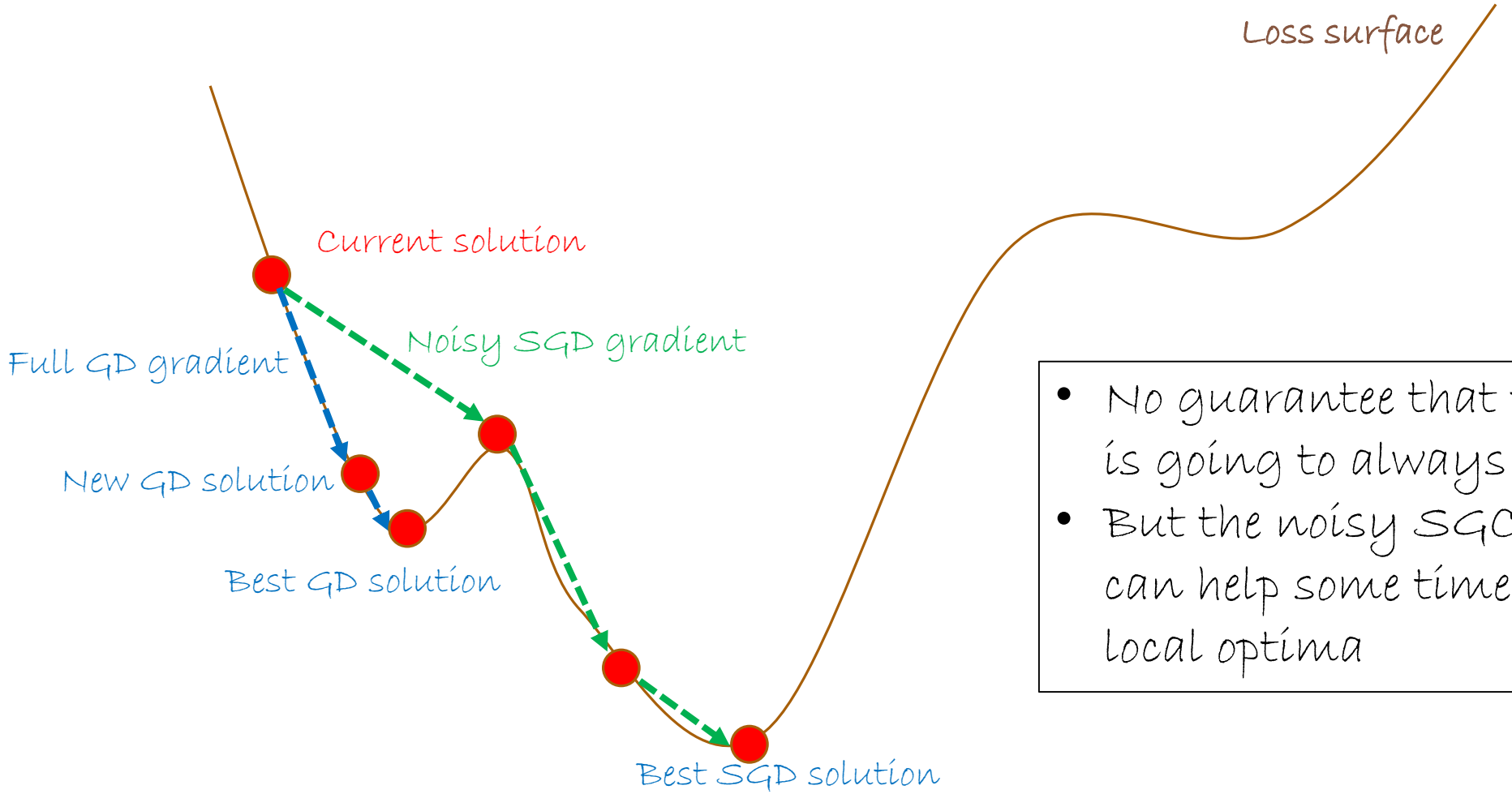
SGD is often better



SGD is often better



SGD is often better

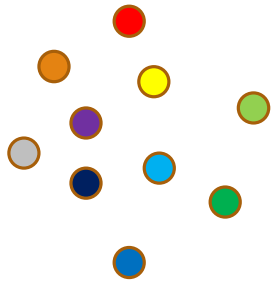


- No guarantee that this is what is going to always happen.
- But the noisy SGC gradients can help some times escaping local optima

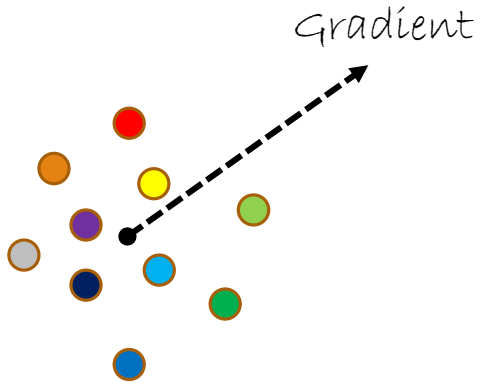
SGD is often better

- The gradient is more “noisy”
- A noisy gradient acts as regularization
- Model does not assume that the training samples are the “absolute representative” of the input distribution
  - Traditional optimization problems: “find optimal route”
- Instead, the model assumes that the sampled training data is roughly representative
- So, model does not overfit to the particular training samples

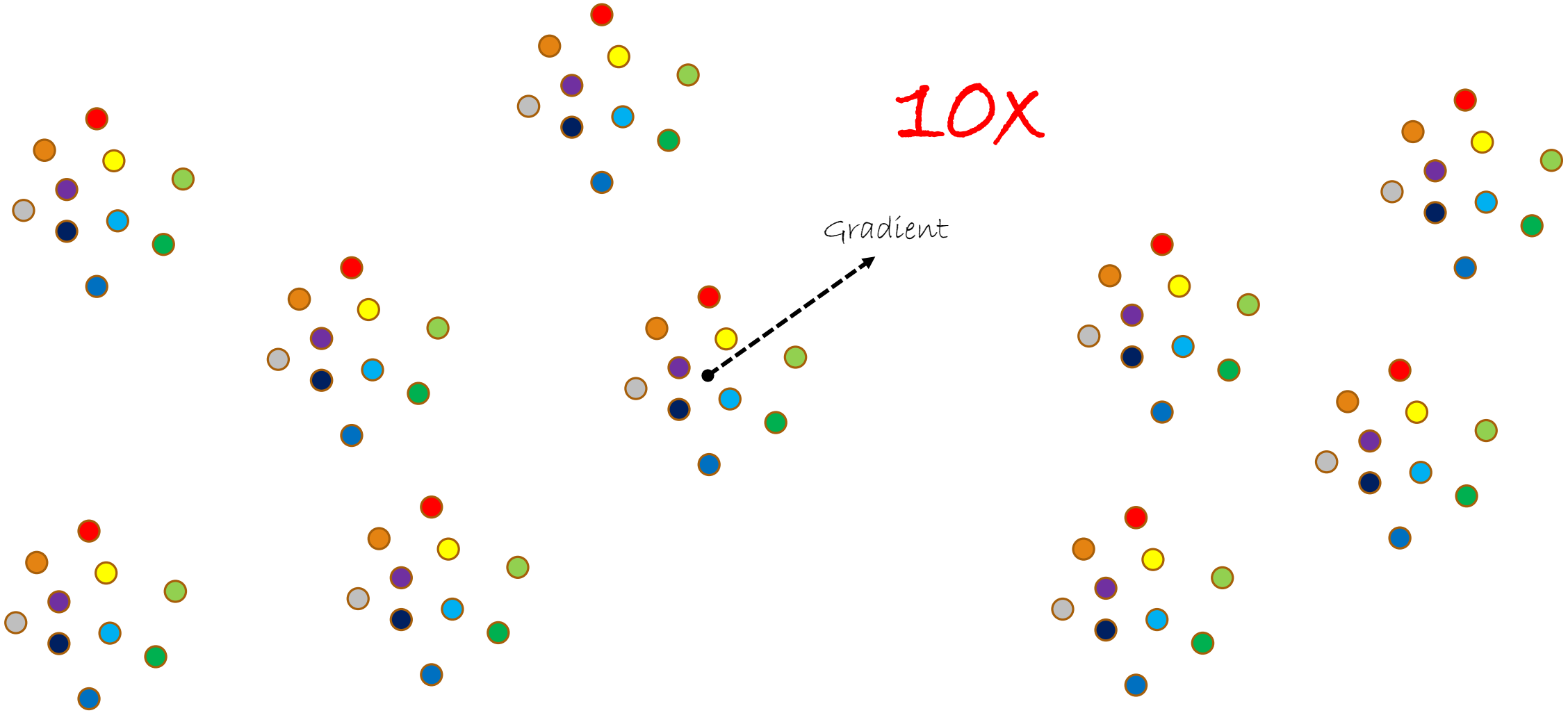
SGD is faster



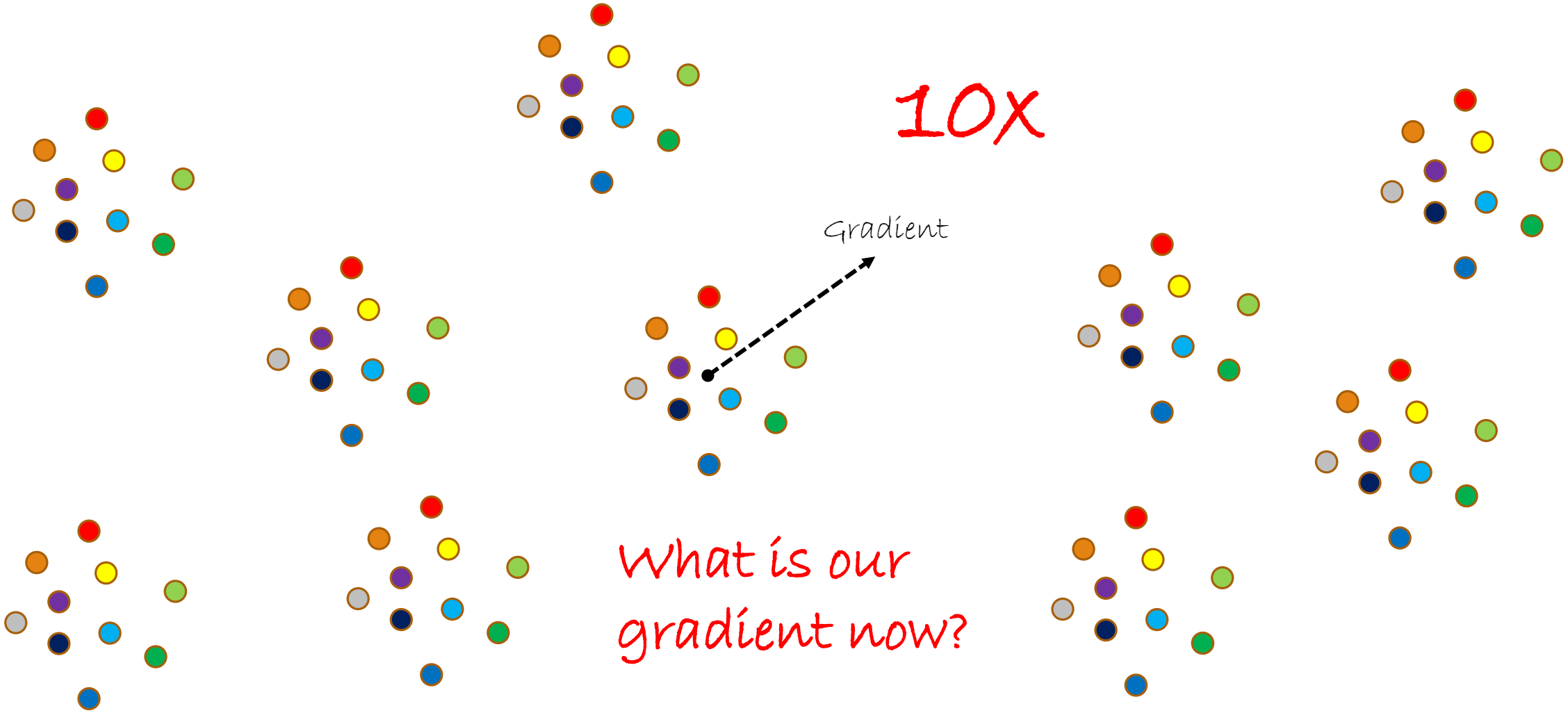
SGD is faster



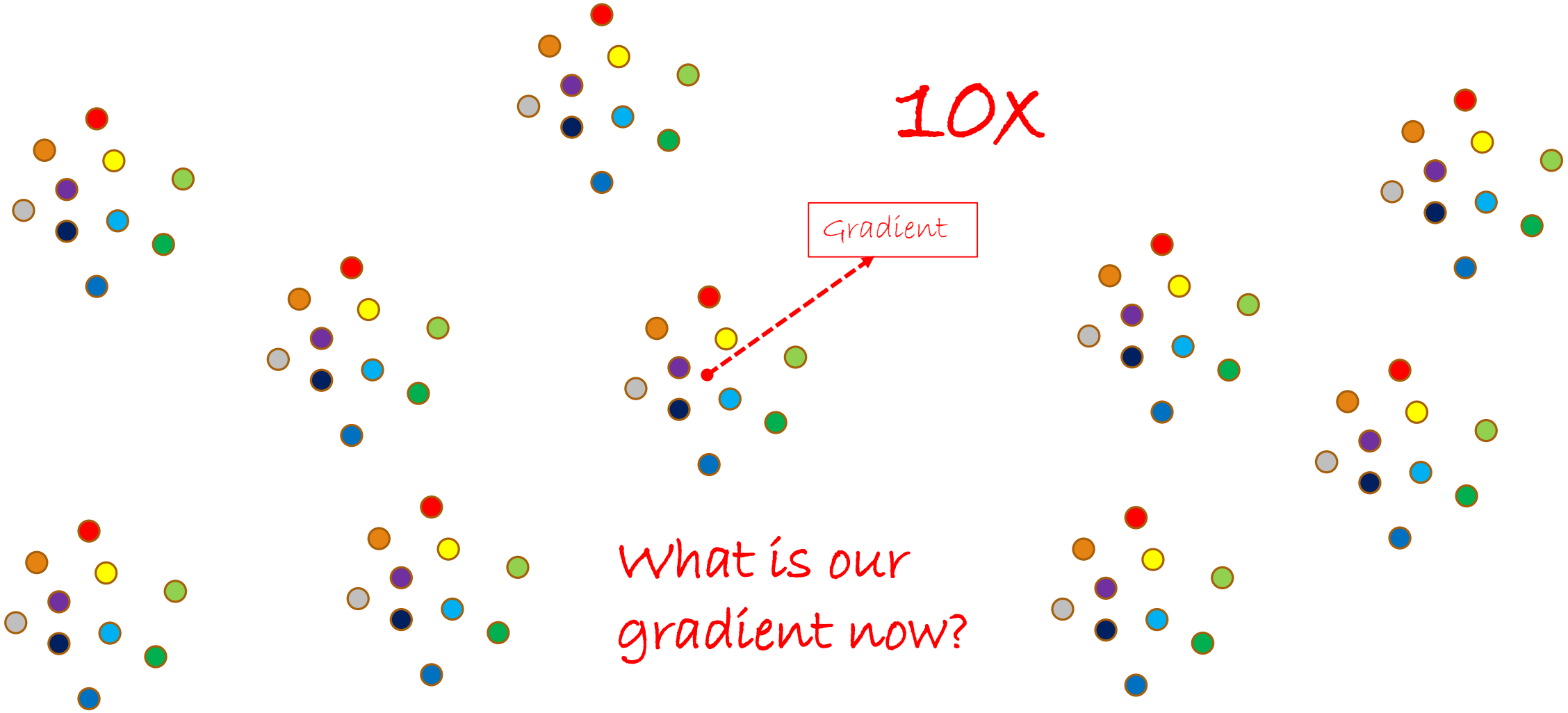
SGD is faster



SGD is faster



SGD is faster



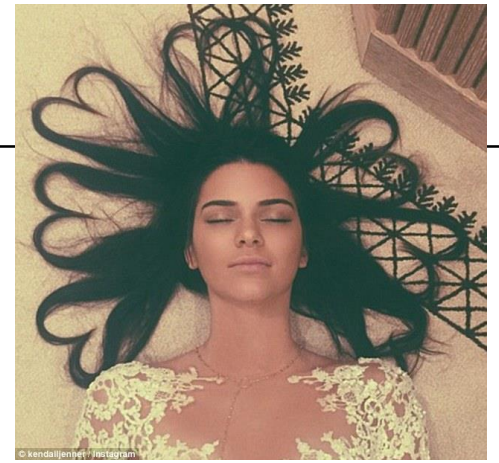
SGD is faster

- 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]

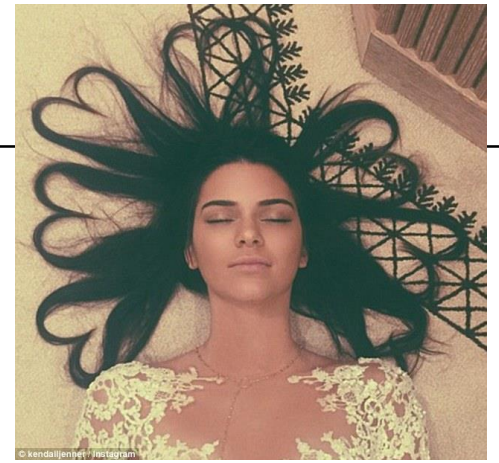
SGD for dynamically changed datasets



- 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]

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]

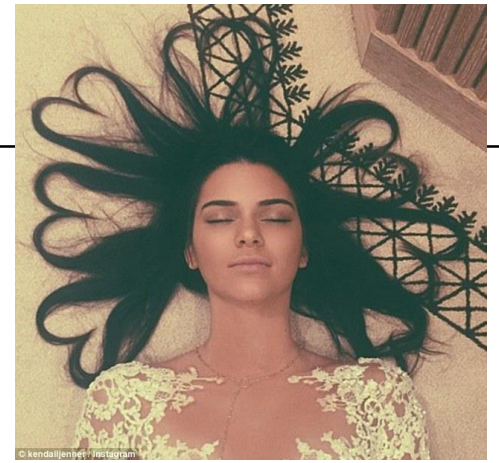


Cool this week

Cool in 2014

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]



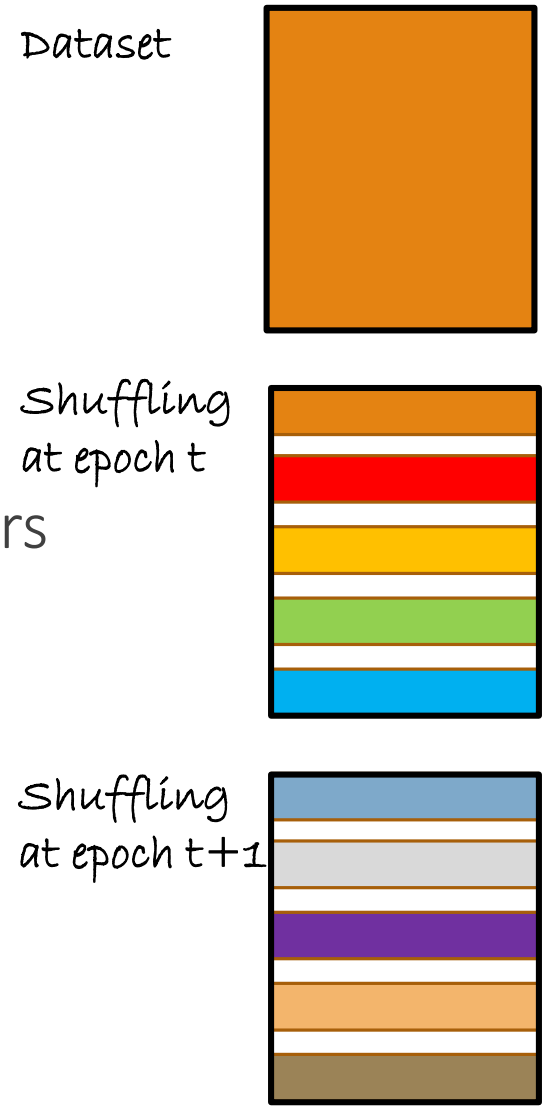
Cool this week

Cool in 2014

Cool in 2010

Shuffling examples

- Applicable only with SGD
- Choose samples with maximum information content
- Shuffle samples so that in a mini-batches the training examples are from different classes
  - As different as possible
- Prefer samples that are more likely to generate larger errors
  - Otherwise gradients will be small and learning will be slow
  - 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
  - At each new epoch create new batches with new, randomly shuffled examples



Advantages of Gradient Descend batch learning

- Conditions of convergence well understood
  - The “good noise” prevents from finding the absolutely best (for our given training dataset) solution
- 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

In practice

- SGD is preferred to Gradient Descend
- Training is orders faster
  - In real datasets Gradient Descend is not even realistic
- Solutions are better and with better generalization
  - Important not only for efficiency, but also for dataset size scale-up
  - Much larger datasets, much better generalization
- How many samples per mini-batch?
  - Hyper-parameter, trial & error
  - Usually between 32-256 samples for image datasets

Data preprocessing & normalization

1. The Neural Network

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

2. Learning by minimizing empirical error

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

3. Optimizing with Gradient Descend based methods

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

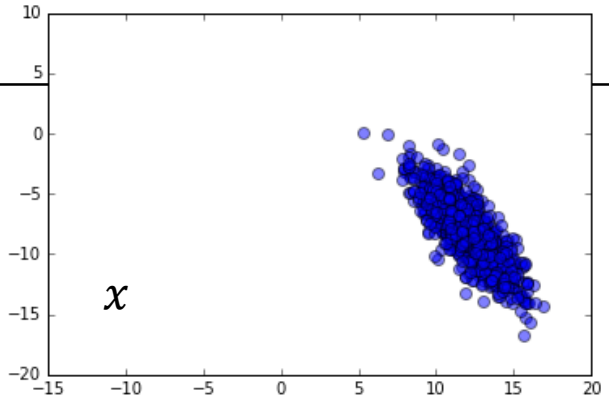
Data pre-processing

- The average of every input variable should be roughly 0
  - Convergence usually faster
  - Otherwise there is bias on the gradient direction, which slows down learning
- Scale input variables so that they have similar diagonal covariances

$$C_i = \sum_j (x_i^{(j)})^2$$

  - Similar covariances help to balance out better the rate at which the weights learn
  - Rescaling to 1 is a good choice, unless some dimensions are less important
- Input variables should be as uncorrelated as possible
  - Input variables are “more independent”, hence one can optimize them better in isolation (not jointly)
  - Caution: extreme correlation (linear dependency) might cause problems

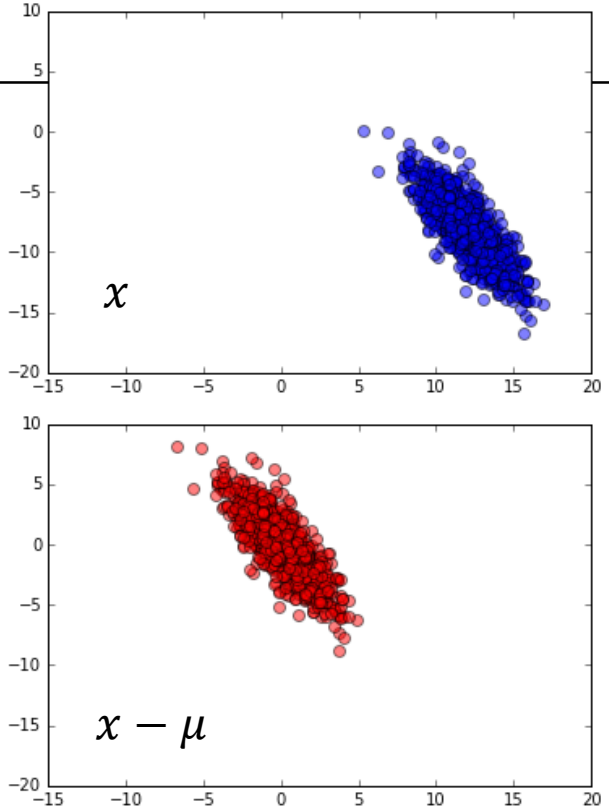
Normalization: $N(\mu, \sigma^2) = N(0, 1)$



- 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

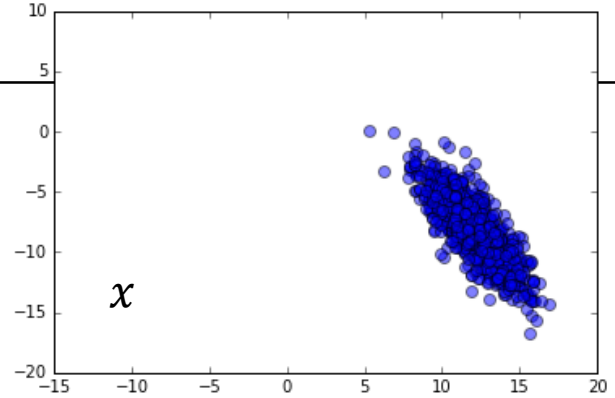
Normalization: $N(\mu, \sigma^2) = N(0, 1)$

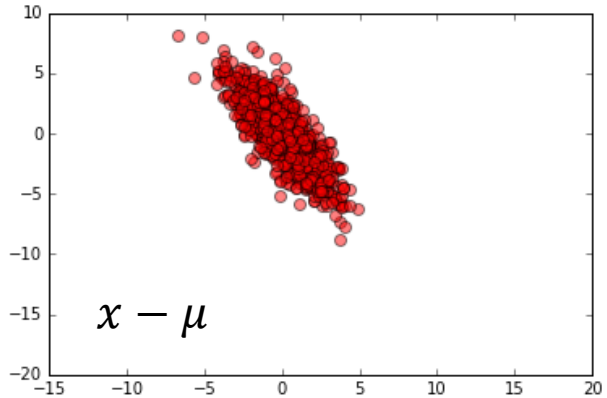
- Input variables follow a Gaussian distribution (roughly)
- In practice:
  - from training set compute mean and standard deviation
  - Then subtract the mean from training samples
  - Then divide the result by the standard deviation

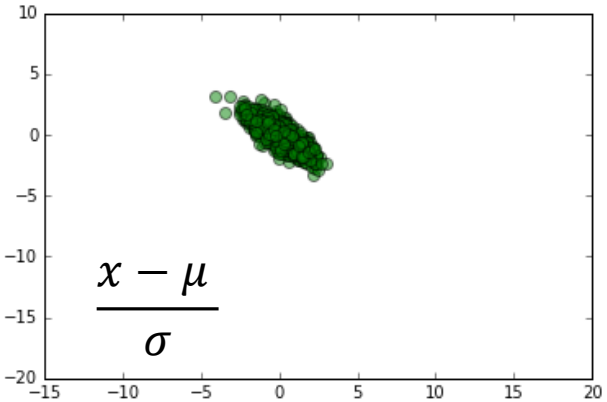


Normalization: $N(\mu, \sigma^2) = N(0, 1)$

- 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







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

- This normalization can be done for all input variables simultaneously
  - If they take more or less similar values, like pixels in natural images
  - Compute one (μ, σ^2) instead of as many as the input variables
- E.g. for images you can compute the general pixel average/variance
  - Or the per color channel pixel average/variance

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

- Or for every variable dimension, e.g. for every pixel R, G, B varible

PCA Whitening

- If $X = [x_1, \dots, x_N]$ and C the covariance matrix is your dataset, εigenvalues and eigenvectors are computed with SVD

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

- Then, the decorrelated (PCA-ed) version of the dataset is obtained by

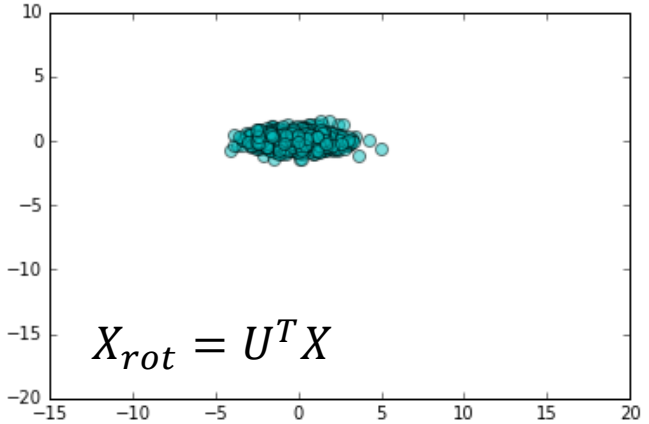
$$X_{rot} = U^T X$$

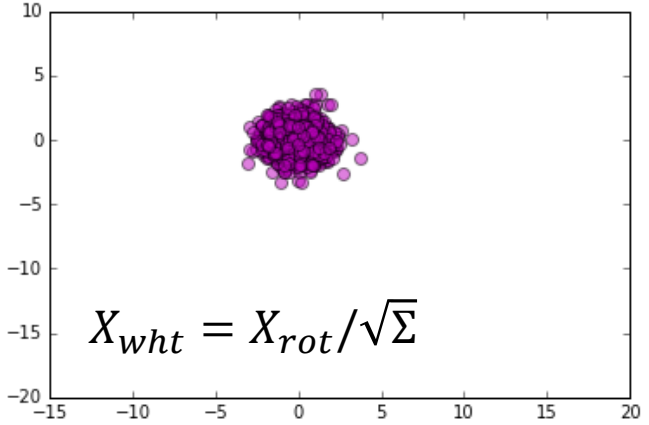
  - Few eigenvectors $U' = [u_1, \dots, u_q]$ return rotated and reduced (in dimensions) version of the data

- Scaling by the square root of eigenvalues gives the whitened data

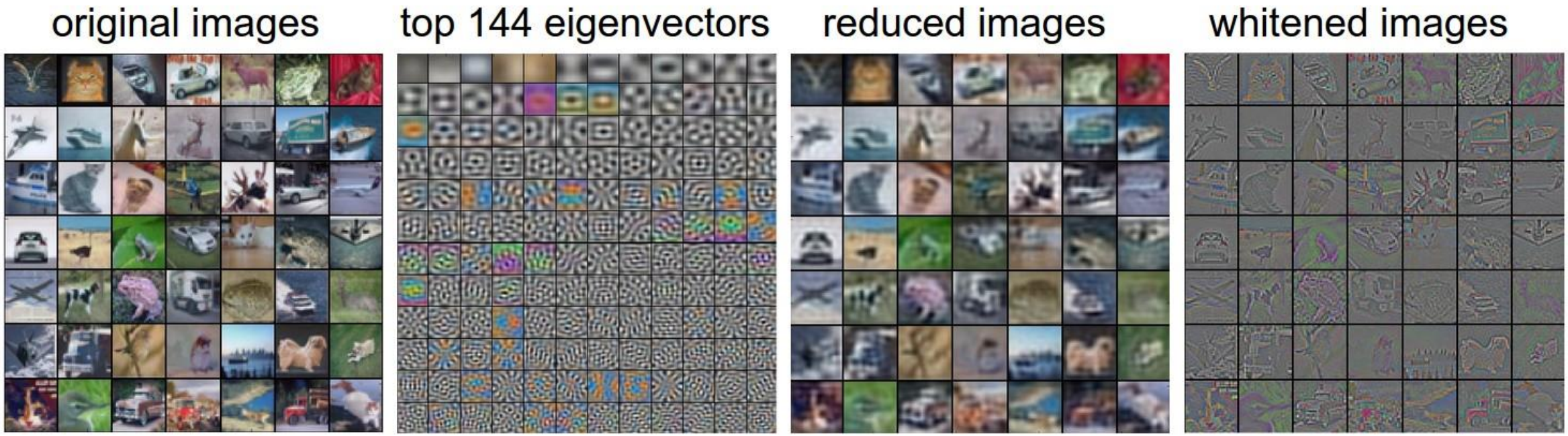
$$X_{wht} = X_{rot} / \sqrt{\Sigma}$$

- With Convolutional Neural Nets this normalization is not used that much
  - The zero mean normalization is more important





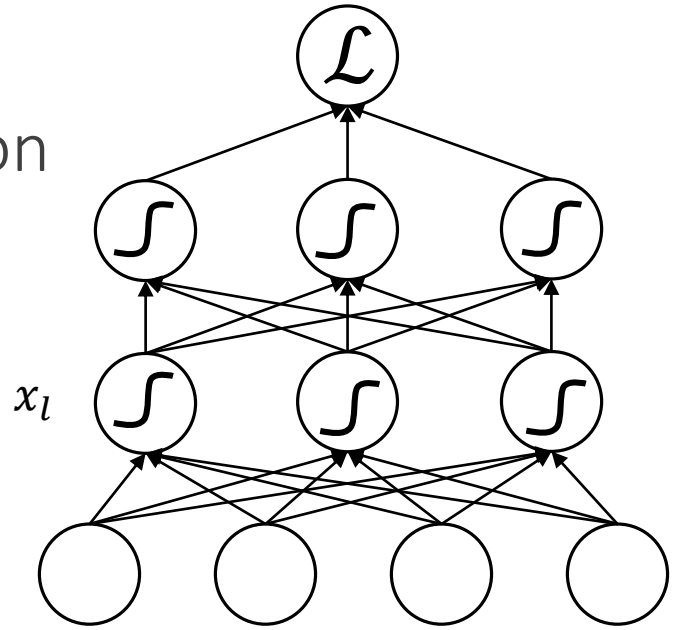
Example

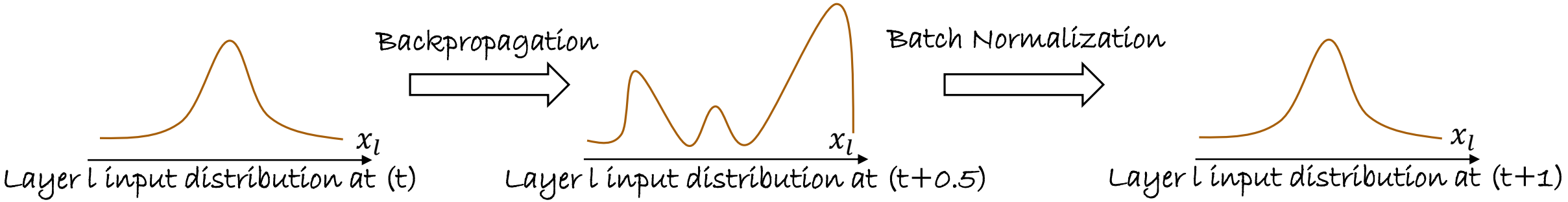


Images taken from A. Karpathy course website: http://cs231n.github.io/neural-networks-2/

Batch normalization

- 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
- Benefits
  - Neurons get activated in a near optimal "regime"
  - Gradients can be stronger, learning rates can be higher
  - Training becomes faster





Data augmentation

Original

Flip

Random crop

Contrast

Tint

Regularization

1. The Neural Network

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

2. Learning by minimizing empirical error

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

3. Optimizing with Gradient Descend based methods

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

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

- ℓ_2-regularization is one of the most important techniques

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

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

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

- λ is usually about $10^{-1}, 10^{-2}$
- Good practice: divide by the number of samples in your (mini-) batch

$$(1 - \lambda\eta_t)\theta^{(t)}$$

if your loss is also averaged by the number of samples

ℓ_2-regularization

- ℓ_2-regularization is one of the most important techniques

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

- The ℓ_2-regularization can pass inside the gradient descend 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 $10^{-1}, 10^{-2}$
- Good practice: divide by the number of samples in your (mini-) batch

$$(1-\lambda\eta_t)\theta^{(t)}$$

if your loss is also averaged by the number of samples

ℓ_1-regularization

- ℓ_1-regularization is one of the most important techniques

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

- The ℓ_1-regularization can pass inside the gradient descend update rule

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

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

- ℓ_1-regularization induces model sparsity
  - Weights are more likely to become 0 with larger λ

ℓ_1-regularization

- ℓ_1-regularization is one of the most important techniques

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

- The ℓ_1-regularization can pass inside the gradient descend update rule

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

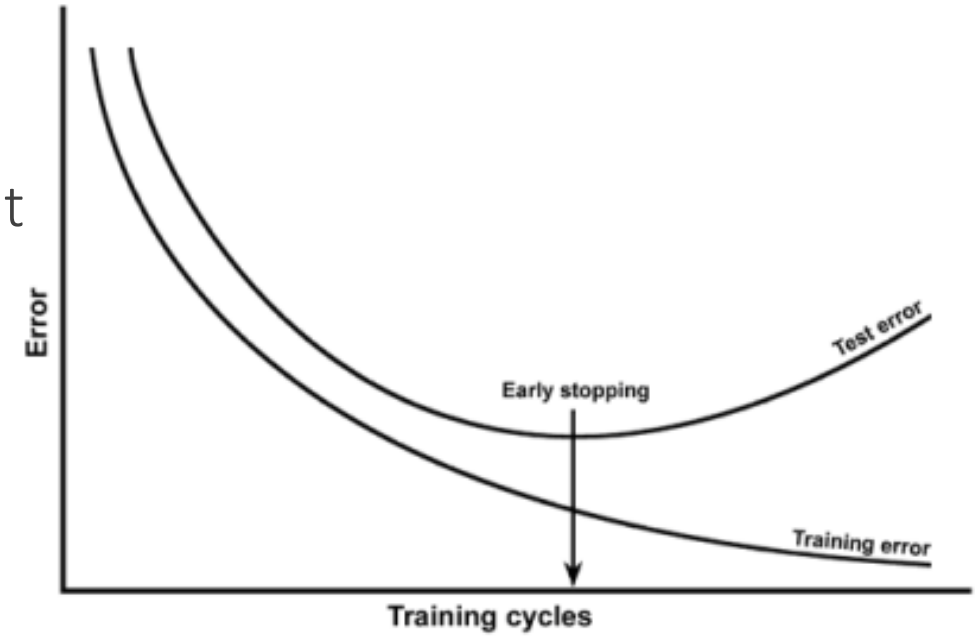
$$\theta^{(t+1)} = \theta^{(t)} - \lambda\eta_t \frac{\theta^{(t)}}{|\theta^{(t)}|} - \eta_t \nabla_\theta \mathcal{L}$$

Sign function

- ℓ_1-regularization induces model sparsity
  - Weights are more likely to become 0 with larger λ

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

- During training setting activations randomly to 0
  - Neurons sampled at random from a Bernoulli distribution with $p = 0.5$
- Effectively, a different architecture at every training epoch
  - Reduced network, as some nodes do not contribute to final score
- Benefits
  - Reduces complex co-adaptations or co-dependencies between neurons
  - No “free-rider” neurons that rely on others
  - Every neuron becomes more robust
  - Overall, decreases significantly overfitting
  - Also, improves significantly training speed
- At test time all neurons are used
  - Neuron activations reweighted by p
- Particularly popular in computer vision, speech recognition

Architectural details

1. The Neural Network

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

2. Learning by minimizing empirical error

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

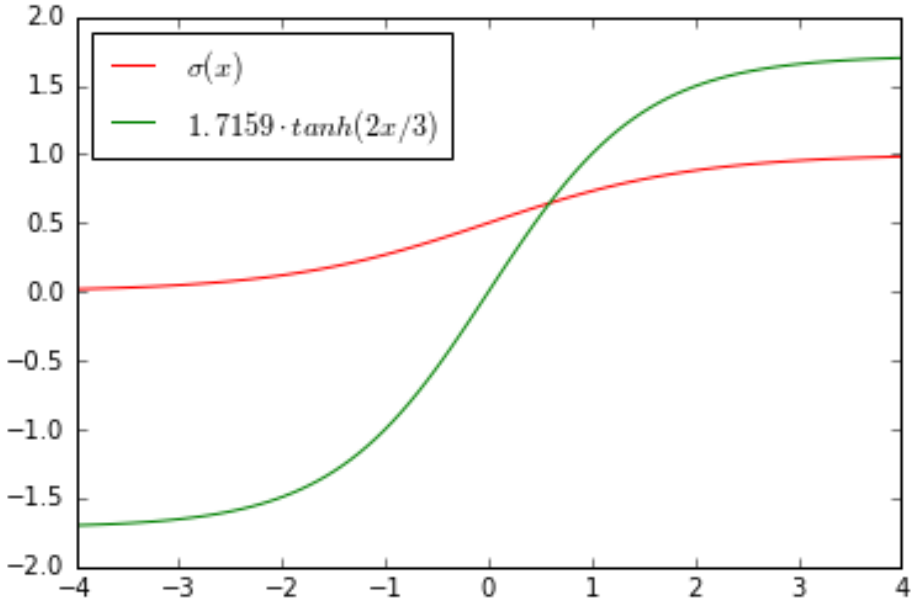
3. Optimizing with Gradient Descend based methods

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

Sigmoid-like activation functions

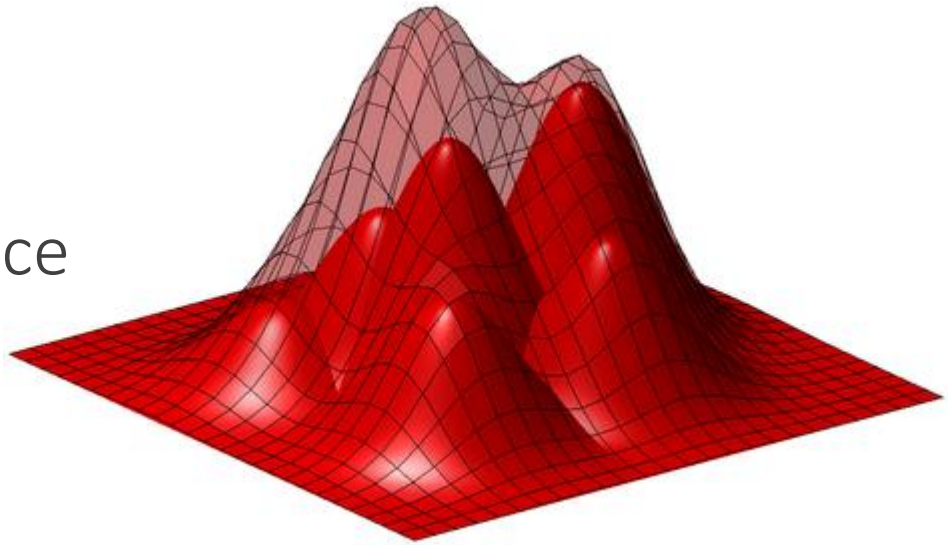
- Straightforward sigmoids are not a very good idea
- Symmetric sigmoids, like tanh, converge faster
- A recommended sigmoid is $a = h(x) = 1.7159 \tanh(\frac{2}{3}x)$
  - A $\tanh$ can be computationally expensive, maybe approximate by ratio of polynomials
- You can add a linear term to avoid flat areas

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



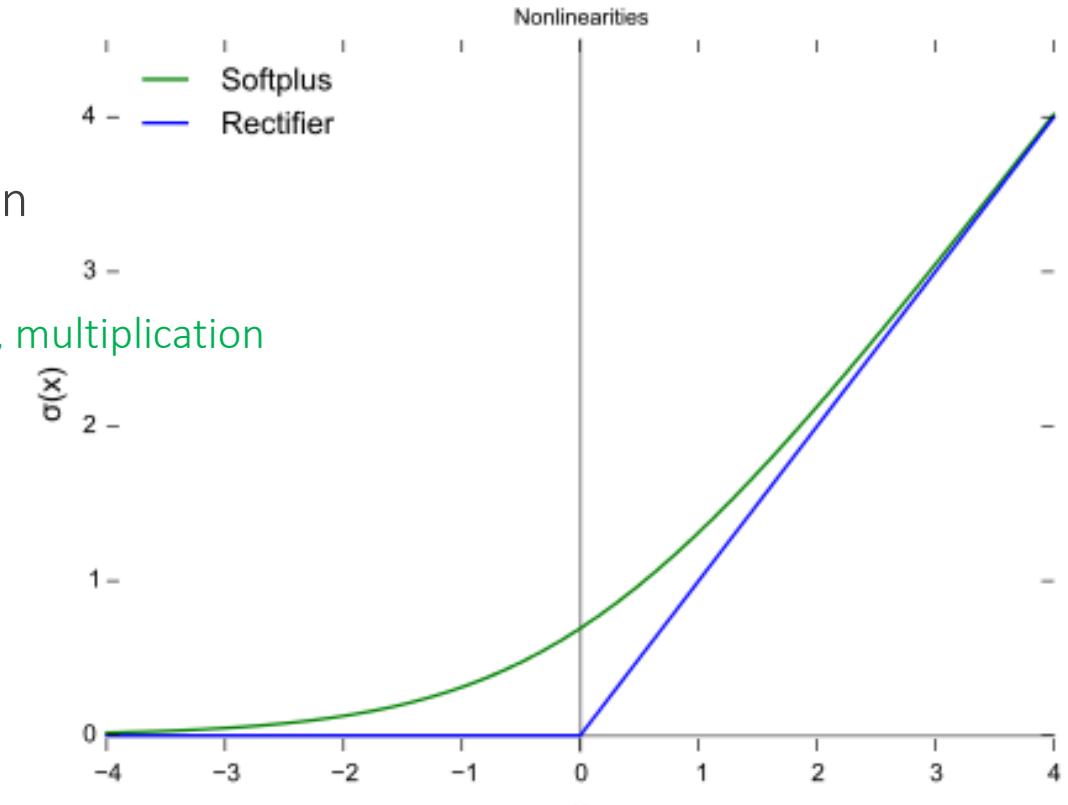
RBFs vs "Sigmoids"

- RBF: $a = h(x) = \sum_j u_j \exp\left(-\beta_j \left(x - w_j\right)^2\right)$
- 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

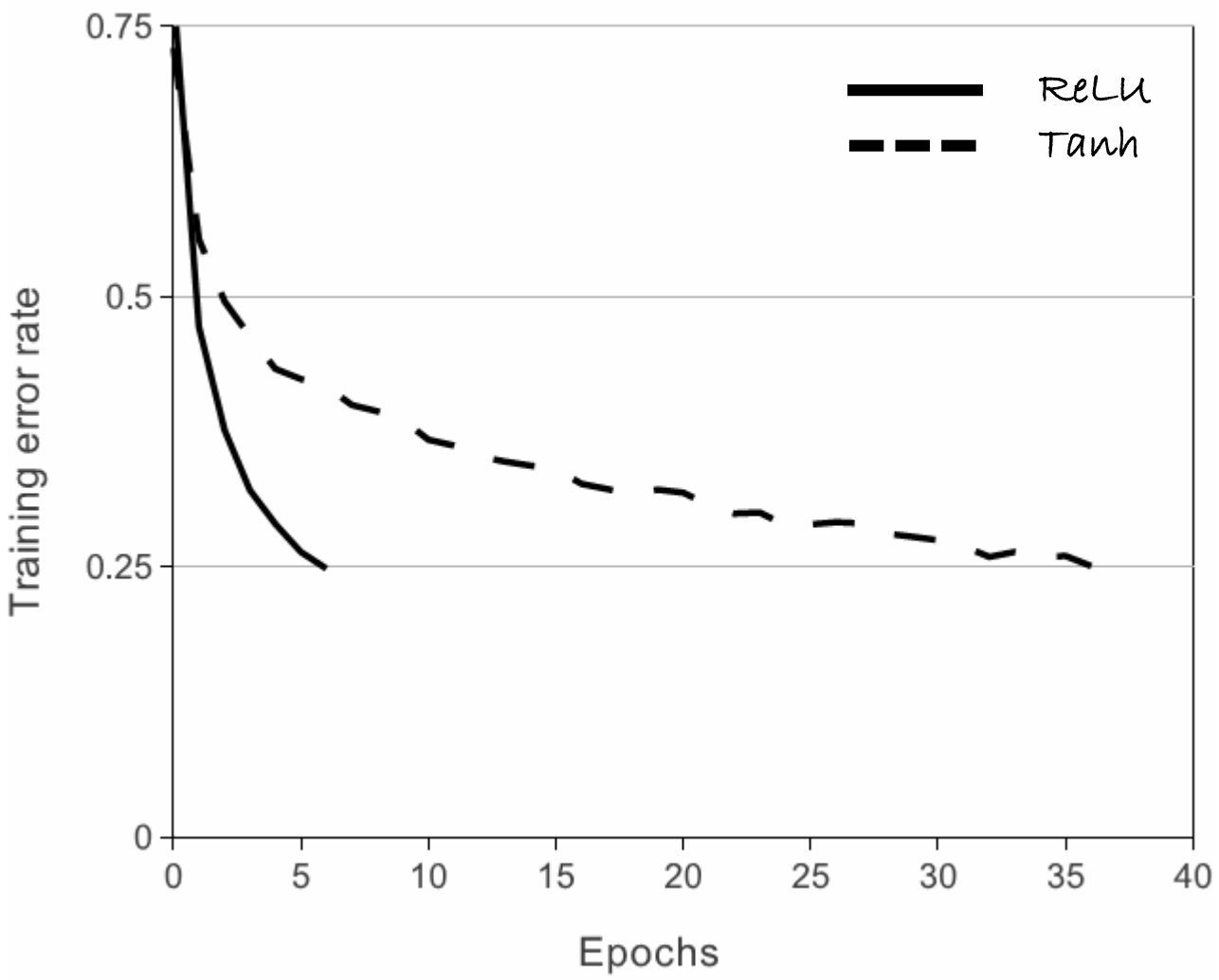


Rectified Linear Unit (ReLU) module (Alexnet)

- 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



Other ReLUs

- Soft approximation (softplus): $a = h(x) = \ln(1 + e^x)$
  - Gradient is the sigmoid $\frac{\partial a}{\partial x} = \sigma(\chi)$
- Noisy ReLU: $a = h(x) = \max(0, \mathrm{x} + \varepsilon), \varepsilon \sim N(0, \sigma(\mathrm{x}))$
- Leaky ReLU: $a = h(x) = \begin{cases} x, if\ x > 0 \\ 0.01x\ otherwise \end{cases}$
- Parametric ReLu: $a = h(x) = \begin{cases} x, if\ x > 0 \\ \beta x\ otherwise \end{cases}$
  - parameter β is trainable

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 hidden units, number of hidden layers

- Getting these hyper-parameters is dataset dependent
- Start small and gradually increase complexity
- With no regularization the plot of number of hidden units vs. generalization performance graph will be U-shaped
- E.g. start with a few hidden layers, 2 or 3
- And a few dozen hidden units per layer and see if performance is reasonable
  - Start increasing the number of layers and see if performance improves
  - Start increasing the number of hidden units and see if performance improves

Use ℓ_2-regularization!

- In general though ℓ_2-regularization is more important!!
- It's alright if you have a deep or wide network
- If there is the ℓ_2-regularization is strong enough, your network will generally not overfit

Learning rate

1. The Neural Network

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

2. Learning by minimizing empirical error

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

3. Optimizing with Gradient Descend based methods

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

Learning rate

- The right learning rate is important for fast convergence
  - Too strong, the gradients overshoot and bounce
  - Too weak, the gradients are too small to influence the parameters → slow training
- Sometimes learning rate per weight is advantageous
  - Some weights are near convergence, others not
- If weights are shared, a good idea is the learning rate to be proportional to the square root of the number of connections sharing the weight
- 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. $\eta_t/2$ or $\eta_t/10$) every T number of epochs
- Inverse decay $\eta_t = \frac{\eta_0}{1+\varepsilon t}$
- Exponential decay $\eta_t = \eta_0 e^{-\varepsilon t}$
- Generally step decay is simple, intuitive, it works well and does not require tuning extra hyper-parameters, other than when to decrease η_t

Learning rate schedules

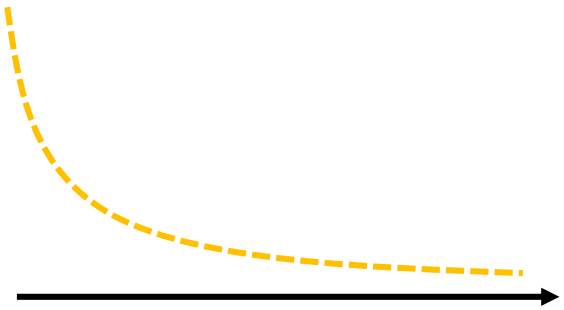
- Constant
  - Learning rate remains the same for all epochs
- Step decay
  - Decrease (e.g. $\eta_t/2$ or $\eta_t/10$) every T number of epochs
- Inverse decay $\eta_t = \frac{\eta_0}{1+\varepsilon t}$
- Exponential decay $\eta_t = \eta_0 e^{-\varepsilon t}$
- Generally step decay is simple, intuitive, it works well and does not require tuning extra hyper-parameters, other than when to decrease η_t

Learning rate schedules

- Constant
  - Learning rate remains the same for all epochs
- Step decay
  - Decrease (e.g. $\eta_t/2$ or $\eta_t/10$) every T number of epochs
- Inverse decay $\eta_t = \frac{\eta_0}{1+\varepsilon t}$
- Exponential decay $\eta_t = \eta_0 e^{-\varepsilon t}$
- Generally step decay is simple, intuitive, it works well and does not require tuning extra hyper-parameters, other than when to decrease η_t

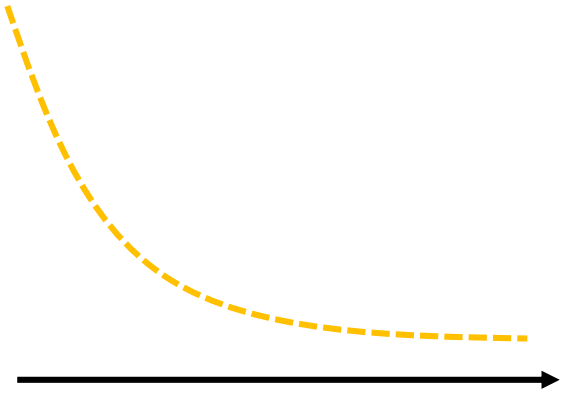
Learning rate schedules

- Constant
  - Learning rate remains the same for all epochs
- Step decay
  - Decrease (e.g. $\eta_t/2$ or $\eta_t/10$) every T number of epochs
- Inverse decay $\eta_t = \frac{\eta_0}{1+\varepsilon t}$
- Exponential decay $\eta_t = \eta_0 e^{-\varepsilon t}$
- Generally step decay is simple, intuitive, it works well and does not require tuning extra hyper-parameters, other than when to decrease η_t



Learning rate schedules

- Constant
  - Learning rate remains the same for all epochs
- Step decay
  - Decrease (e.g. $\eta_t/2$ or $\eta_t/10$) every T number of epochs
- Inverse decay $\eta_t = \frac{\eta_0}{1+\varepsilon t}$
- Exponential decay $\eta_t = \eta_0 e^{-\varepsilon t}$
- Generally step decay is simple, intuitive, it works well and does not require tuning extra hyper-parameters, other than when to decrease η_t



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

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

2. Learning by minimizing empirical error

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

3. Optimizing with Gradient Descend based methods

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

Weight initialization

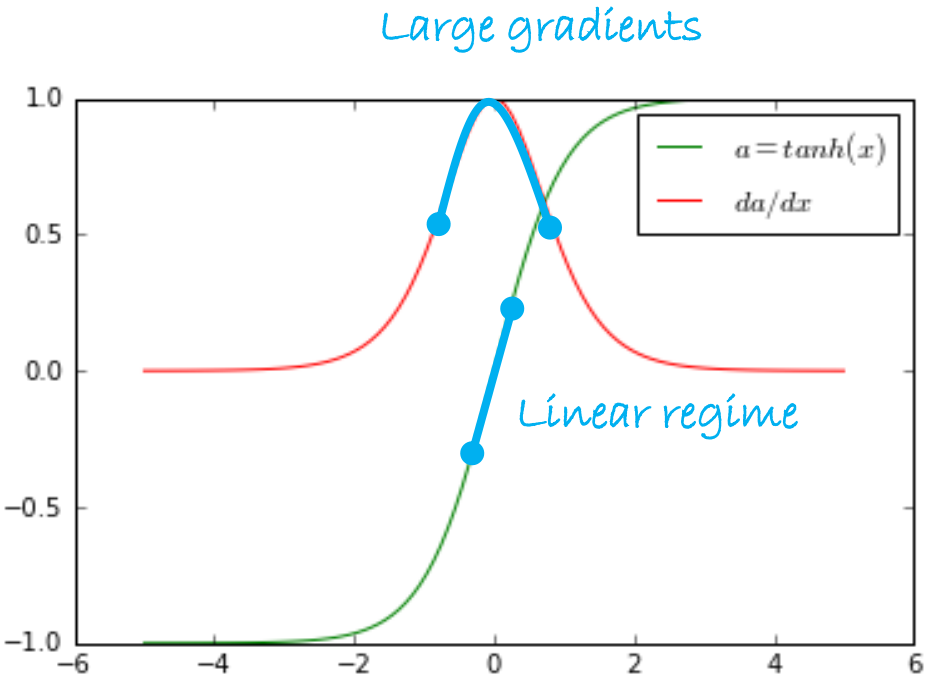
- There are few contradictory requirements
- Weights need to be small enough
  - e.g. around the origin ($\vec{\mathbf{0}}$) for symmetric functions (tanh, sigmoid)
  - the activation functions operate near their linear regime → large gradients → faster training
- Weights need to be large enough
  - The generated gradients are also large enough → faster training

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
- Good practice: initialize weights to be asymmetric, e.g. no same values for different weights (like all $\vec{\mathbf{0}}$)
  - Otherwise all neurons generate the same gradient, no real change
  - Alternatively, initialize to $\vec{\mathbf{0}}$ but break some node to node connections to create asymmetries
- Generally, Initialization must be coordinated with the choice of non-linear activation functions and data normalization

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



Weight initialization for ReLUs

- For ReLU's you also want to initialize the weights so the neurons have similar variances
- Currently the suggested practice is to fill in the weights with random samples draw from

$$\mathrm{w} \sim N\left(0, \sqrt{2/d}\right)$$

where d is the number of neurons in the input [HeICCV2015]

Loss functions

1. The Neural Network

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

2. Learning by minimizing empirical error

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

3. Optimizing with Gradient Descend based methods

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

Multi-class classification

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

$\mathcal{L}(\theta; x, y) = -\sum_{c=1}^{C} y_c \log a_L^c$ 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, yes or no?"
  - "Does this contain a person, yes or no?"
  - "Does this contain a motorbike, yes or no?"
  - "Does this contain a horse, yes or no?"

- 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 sum of squared errors

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

- Or the ℓ_1 distance

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

Even better optimizations

1. The Neural Network

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

2. Learning by minimizing empirical error

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

3. Optimizing with Gradient Descend based methods

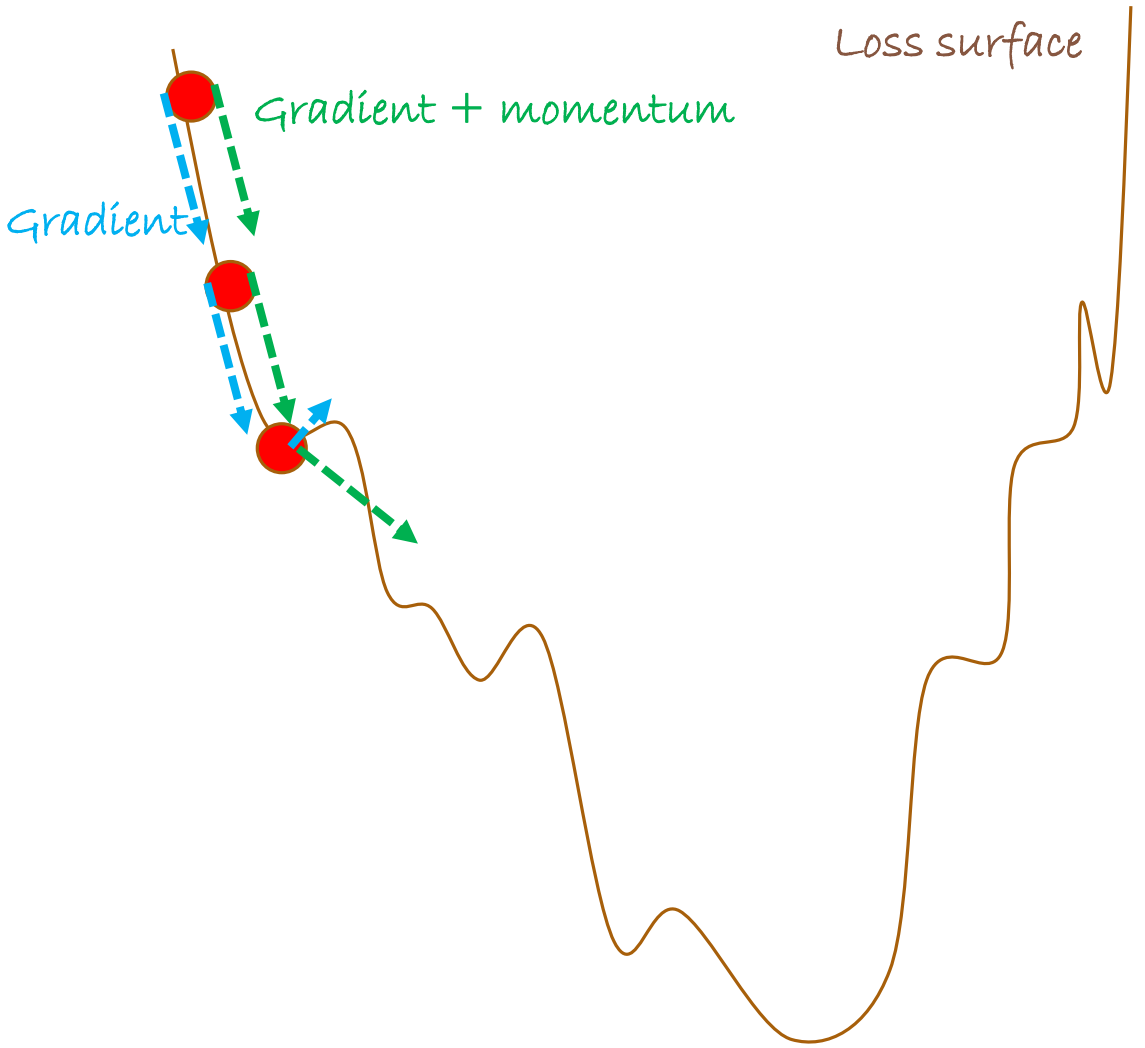
$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}$$

Momentum

- Instead of switching gradients all the time, maintain some "momentum" from the previous parameters

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

- Gradients and learning are more robust, faster convergence
- Nice "physics"-based interpretation
  - Instead of updating the position of the "ball", we update the velocity, which will update the position

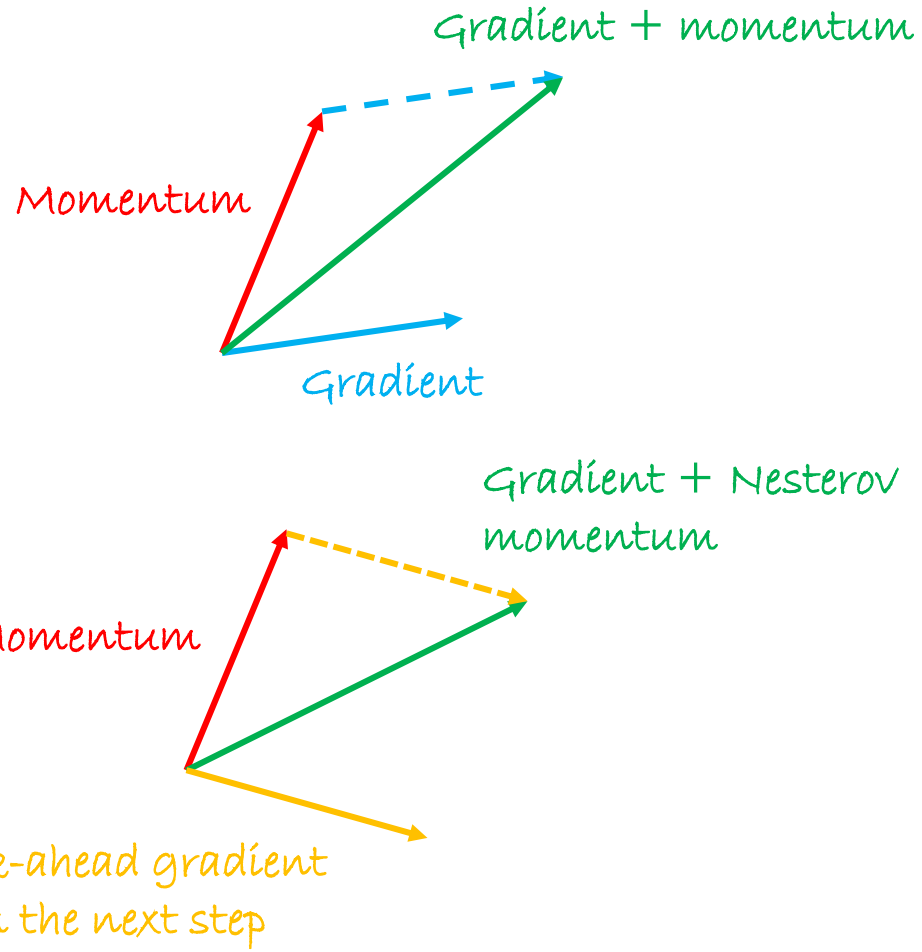


Nesterov Momentum

- Use the future gradient instead of the current gradient

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

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



Second order optimizations

- Normally we update all weights with same “aggressiveness”
  - Yet, some parameters could enjoy more “teaching”
  - While others are already about there
- Second-order methods adapt the learning according to the per parameter behavior

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

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

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

Second order optimization methods in practice

- Computing the inverse of the Hessian with thousands of parametesr is usually very expensive
- Instead approximations are sough for, e.g. the L-BFGS algorithm
  - Keeps memory of gradients to approximate the inverse Hessian
- However, L-BFGS works alright with Batch Gradient Descend
  - What about SGD?
- In practice SGD with a good momentum works alright

Per parameter adaptive optimization

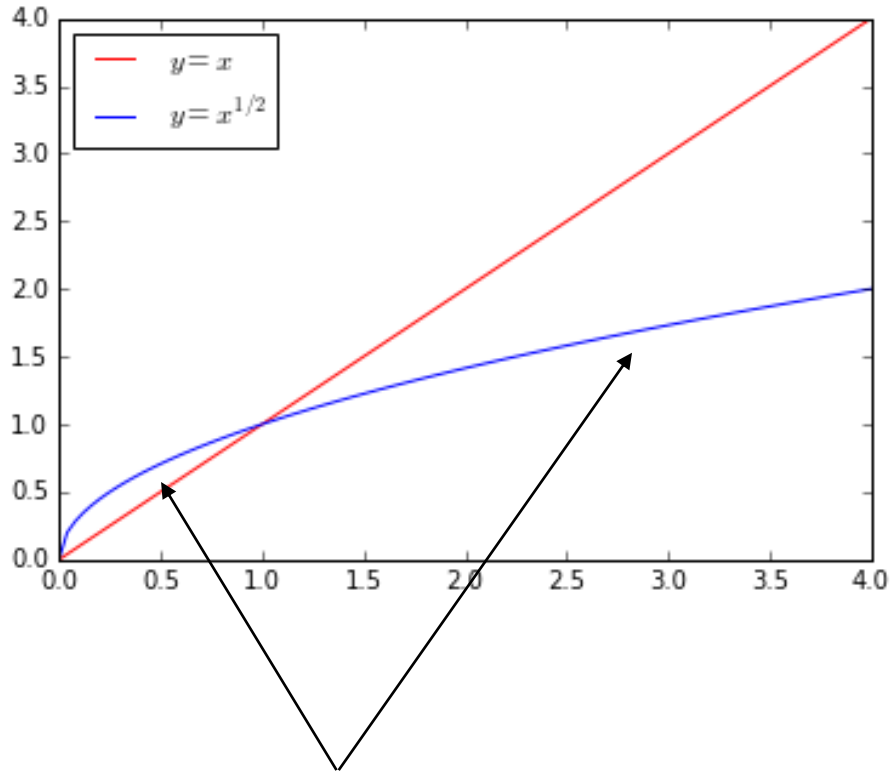
- Adagrad [Duchi2011]
- RMSprop
- Adam [Kingma2014]

Adagrad [Duchi2011]

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

RMSprop

- Schedule
  - $m_j = \alpha \sum_{\tau=1}^{t-1} (\nabla_\theta \mathcal{L}_j)^2 + (1-\alpha) \nabla_\theta^{(t)} \mathcal{L}_j \quad \Longrightarrow \quad \theta^{(t+1)} = \theta^{(t)} - \eta_t \frac{\nabla_\theta \mathcal{L}}{\sqrt{m}+\varepsilon}$
- α is a decay hyper-parameter
- Similar like Adagrad, but uses a moving average of the squared gradients
- When gradients are too large (maybe too "noisy" loss surface)
  - Updates are tamed
- When gradients are too small (maybe stuck in flat loss surface ravine)
  - Updates become more aggressive

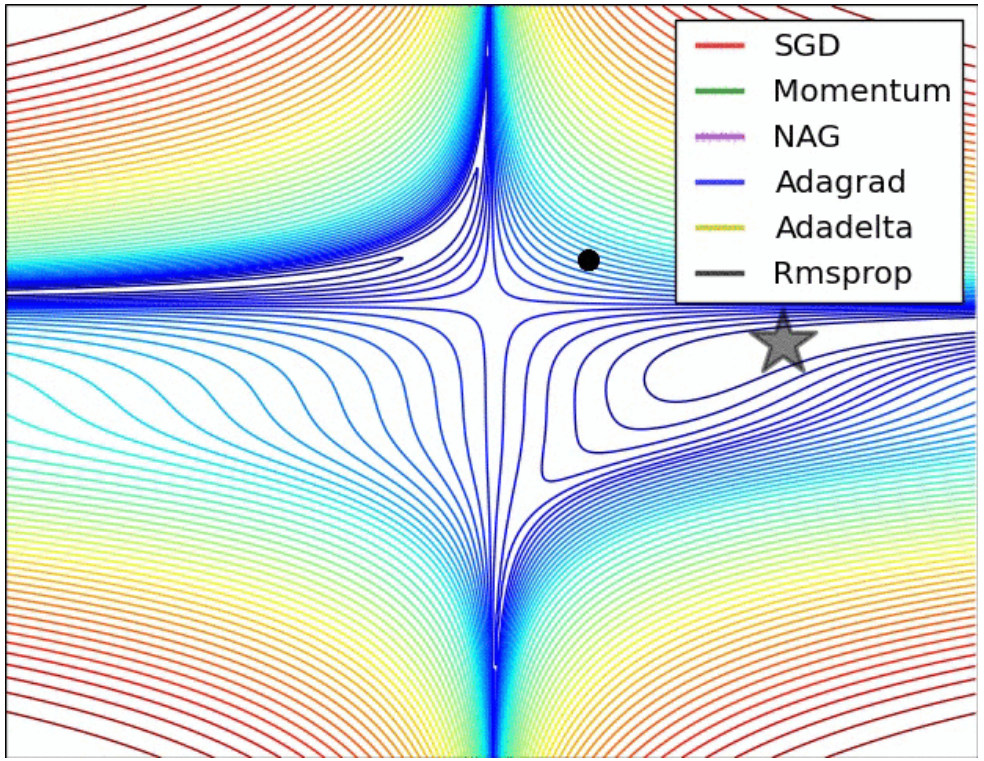


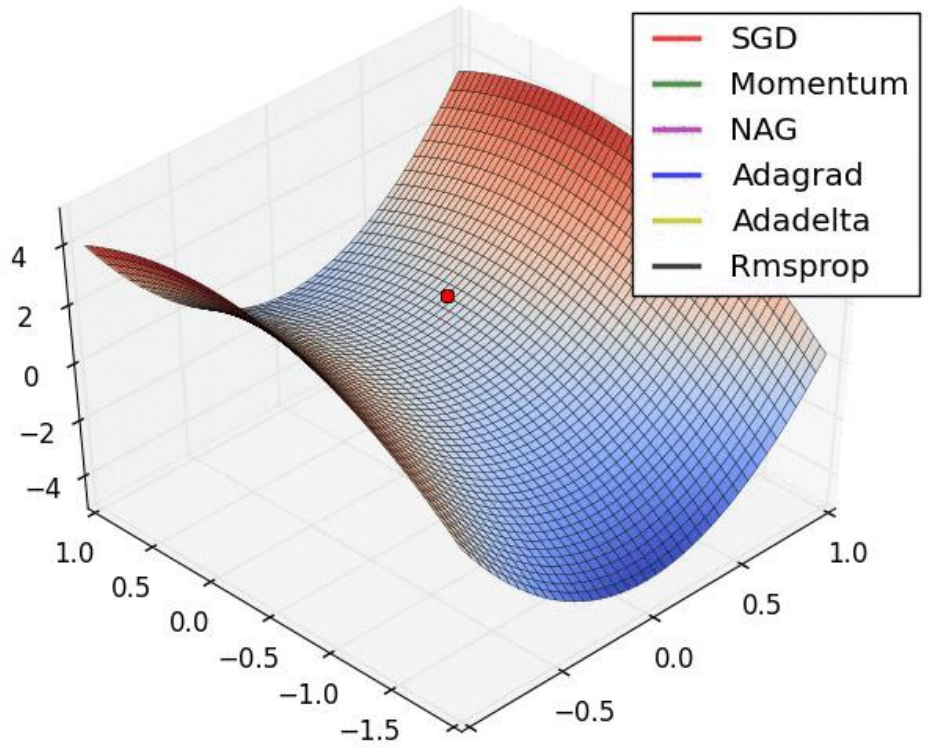
Square rooting boosts small values while suppresses large values

Adam [Kingma2014]

- $m_j = \sum_\tau (\nabla_\theta \mathcal{L}_j)^2$
- $\theta^{(t+0.5)} = \beta_1 \theta^{(t)} + (1-\beta_1)\nabla_\theta \mathcal{L}$
- $v^{(t+0.5)} = \beta_2 v^{(t)} + (1-\beta_2) m$
- $\theta^{(t+1)} = \theta^{(t)} - \eta_t \frac{\theta^{(t+0.5)}}{\sqrt{v^{(t+0.5)}} + \varepsilon}$
- Similar to RMSprop with momentum
- Recommended values: $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$

Visual overview





Picture credit: Alec Radford

Good practice

- Preprocess the data to have 0 mean
  - Either normalize to have standard deviation 1 or the inputs to lie in the range [-1, 1]
- Initialize weights according to you activations functions
  - For ReLU initialize from $N(0, \sqrt{\frac{2}{d}})$, d is the number of input neurons
- Always use ℓ_2-regularization and dropout
- Use batch normalization

Babysitting Deep Nets

1. The Neural Network

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

2. Learning by minimizing empirical error

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

3. Optimizing with Gradient Descend based methods

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

Babysitting Deep Nets

- Check your gradients
- 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 defining 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

Next lecture

- What are the Convolutional Neural Networks?
- Why are they so important for Computer Vision?
- How do they differ from standard Neural Networks?
- How can we train a Convolutional Neural Network?