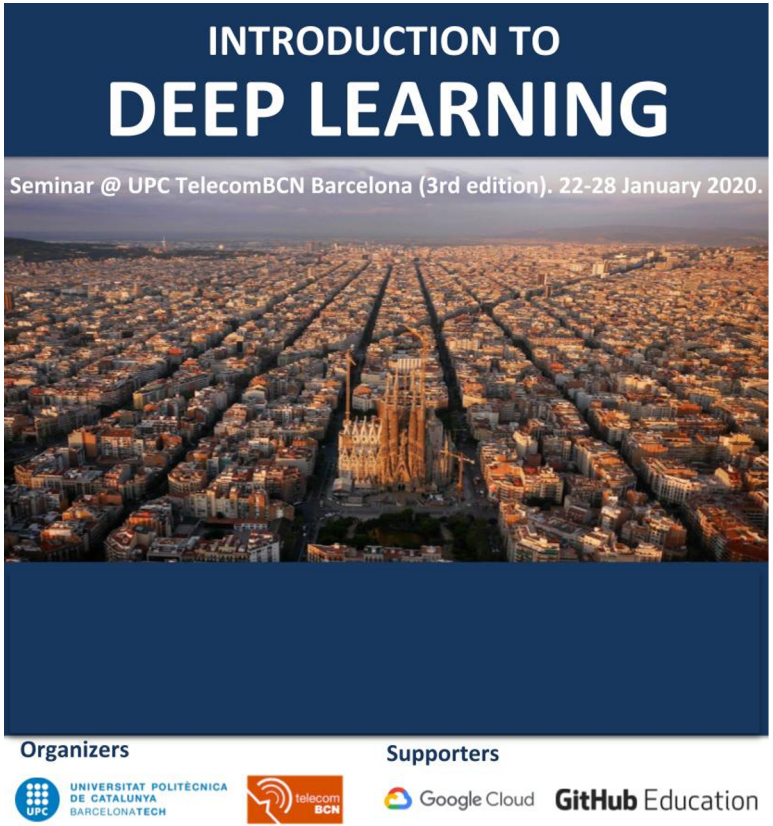


Day 3 Lecture 1

Optimization for neural network training

Verónica Vilaplana
veronica.vilaplana@upc.edu

Associate Professor
Universitat Politecnica de Catalunya

Previously in IDL…

- Multilayer perceptron
- Loss function
- Training: gradient descent
  - Backpropagation

but…

What type of optimization problem?

Do local minima cause problems?

Does gradient descent perform well?

How does dataset size affect training?

How to set the learning rate?

Index

- **Challenges in deep NN optimization**
  - Expected and empirical risk
  - Surrogate loss functions
  - Local minima, saddle points
  - Early stopping

- **Practical algorithms**
  - Gradient descent
  - Batch and mini-batch algorithms
  - Momentum
  - Nesterov Momentum
  - Learning rate
  - Adaptive learning rates: adaGrad, RMSProp, Adam

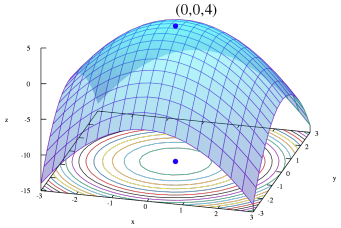
- **Parameter initialization. Normalization**

Challenges in deep NN optimization

Differences between learning and pure optimization

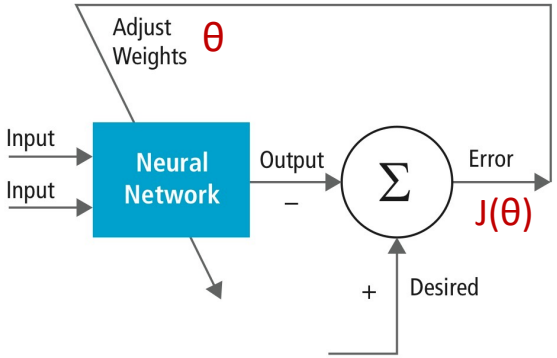
What is optimization?

- The process to find maxima or minima of a function based on constraints



- Involved in many contexts of deep learning, the hardest one is neural network training
- We will focus on one particular case of optimization:

Finding the parameters θ of a network that reduce a cost function J(θ)



Optimization for NN training

- **Goal:** Find the parameters that minimize the **expected risk** (generalization error)

$$J(\theta) = \mathrm{E}_{(x,y)\sim p_{data}} L(f_\theta(x), y)$$

  - x input, $f_\theta(x)$ predicted output, y target output, E expectation
  - p_{data} **true** (**unknown**) data distribution, **L loss function (how wrong predictions are)**

- But we only have a training set of samples: we minimize the **empirical risk**, average loss on a finite training dataset D, hoping that this will also minimize the expected risk

$$J(\theta) = \mathrm{E}_{(x,y)\sim \hat{p}_{data}} L(f_\theta(x), y) = \frac{1}{|D|} \sum_{(x_i,y_i)\in D} L(f_\theta(x_i), y_i)$$

where $\hat{p}_{data}$ is the **empirical** distribution, $|D|$ is the number of examples in D

Surrogate loss

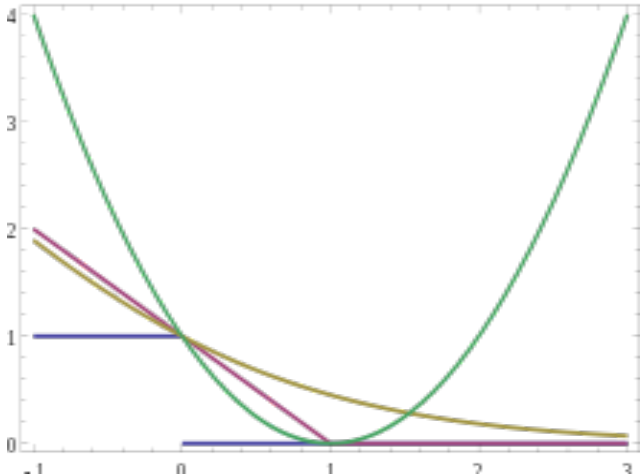
- Often minimizing the real loss is intractable (can't be used with gradient descent)
  - e.g. performance 0-1 loss (0 if correctly classified, 1 if it is not) $L(f(x), y) = I_{(f(x) \neq y)}$

- Minimize a surrogate loss instead
  - e.g. for the 0-1 loss

hinge $L(f(x), y) = \max(0, 1 - yf(x))$

square $L(f(x), y) = (1 - yf(x))^2$

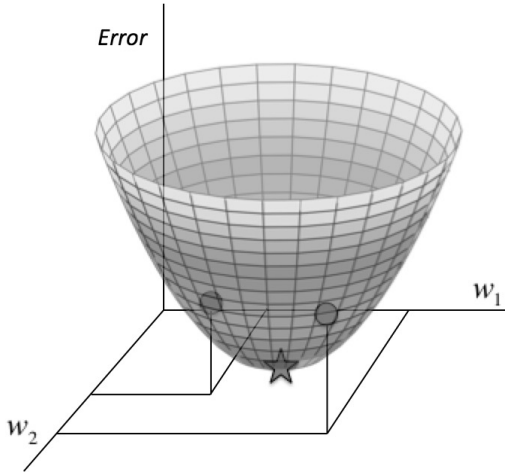
Cross entropy $L(f(x), y) = -(y \log(f(x)) + (1-y) \log(1-f(x)))$



0-1 loss (blue) and surrogate losses
(green: square, purple: hinge, yellow: logistic)

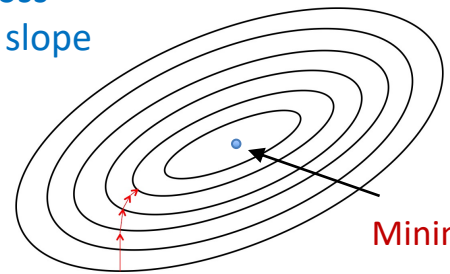
Loss surface

- Let's consider a single linear neuron with two inputs and squared error
- Loss surface is a quadratic bowl:
  - Vertical cross sections are parabolas
  - Horizontal cross sections are ellipses



Two inputs: weights w1,w2

Contours = same value of Loss
Closer contours -> steeper slope

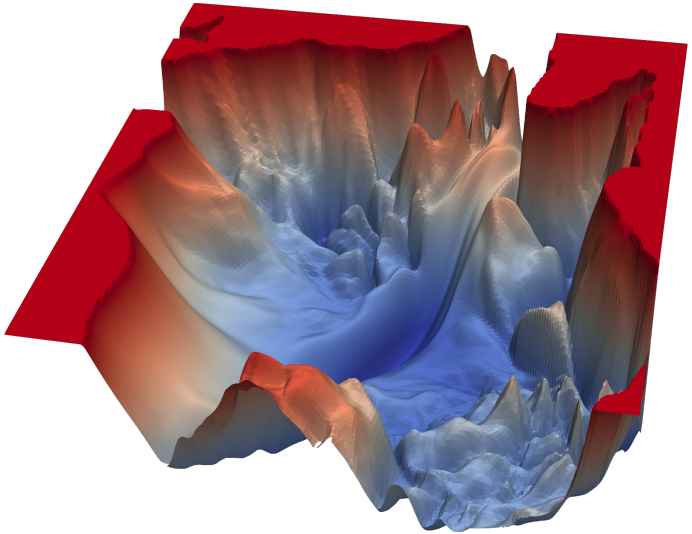


Minimum error

Gradient descent steps

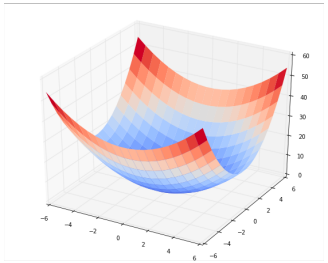
Loss surface

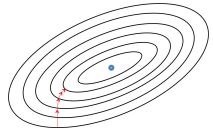
- For multi-layer, non-linear nets, the error surface is much more complicated!



https://www.cs.umd.edu/~tomg/projects/landscapes/

- But locally, a quadratic bowl is a good approximation

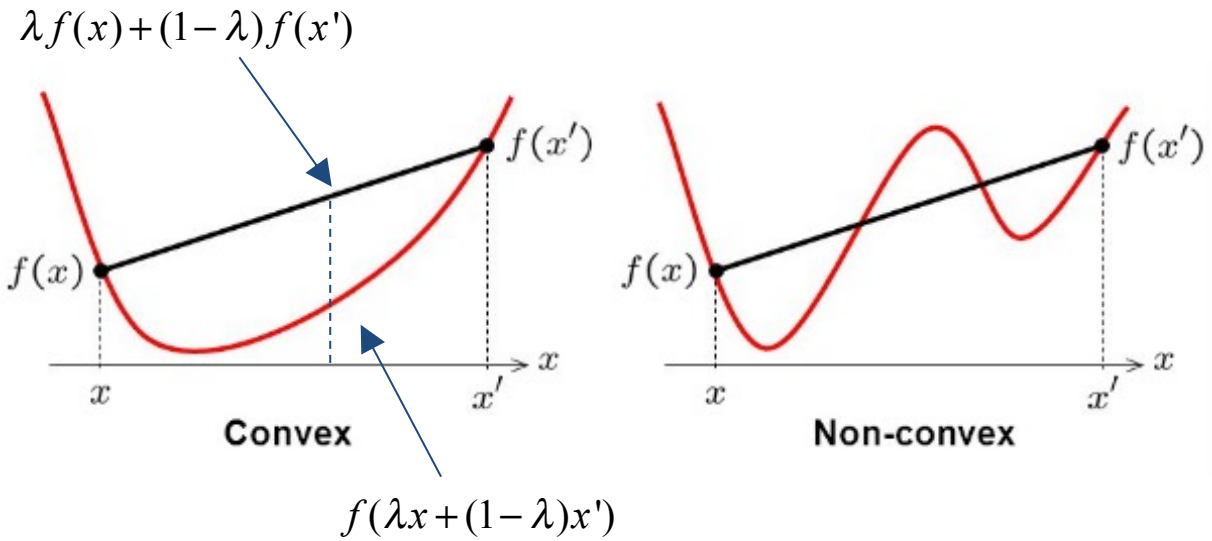




Convex / Non-convex optimization

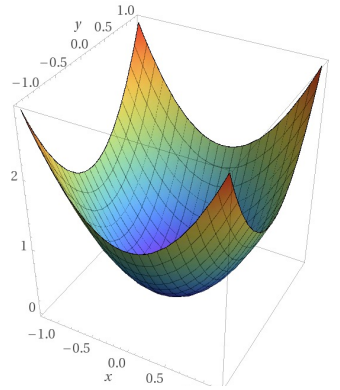
A function $f : X \to \mathbb{R}$ defined on an n-dim interval is convex if for any $x, x' \in X \quad \lambda \in [0,1]$

$$f(\lambda x + (1-\lambda)x') \leq \lambda f(x) + (1-\lambda) f(x')$$

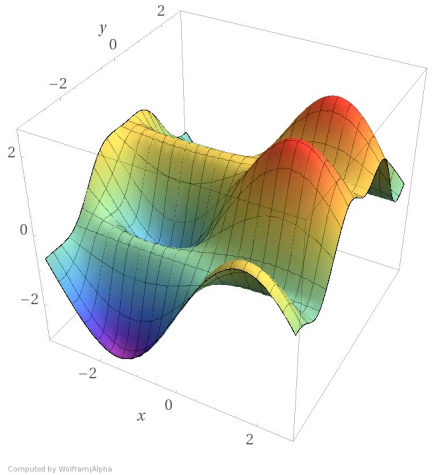


Convex / Non-convex optimization

- Convex optimization
  - any local minimum is a global minimum
  - there are several opt. algorithms (polynomial-time)



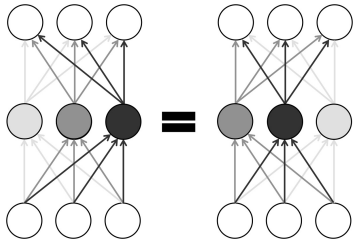
- Non-convex optimization
  - **objective function in deep networks is non-convex**
  - deep models may have several local minima
  - but this is not necessarily a major problem!



Local minima

- How common are local minima in loss surfaces of deep nets?
- Are they problematic for training?

- NNets usually have a large amount of local minima: many **non-identifiable** configurations behave in an indistinguishable fashion
  - But they are not problematic (all configurations achieve same error)



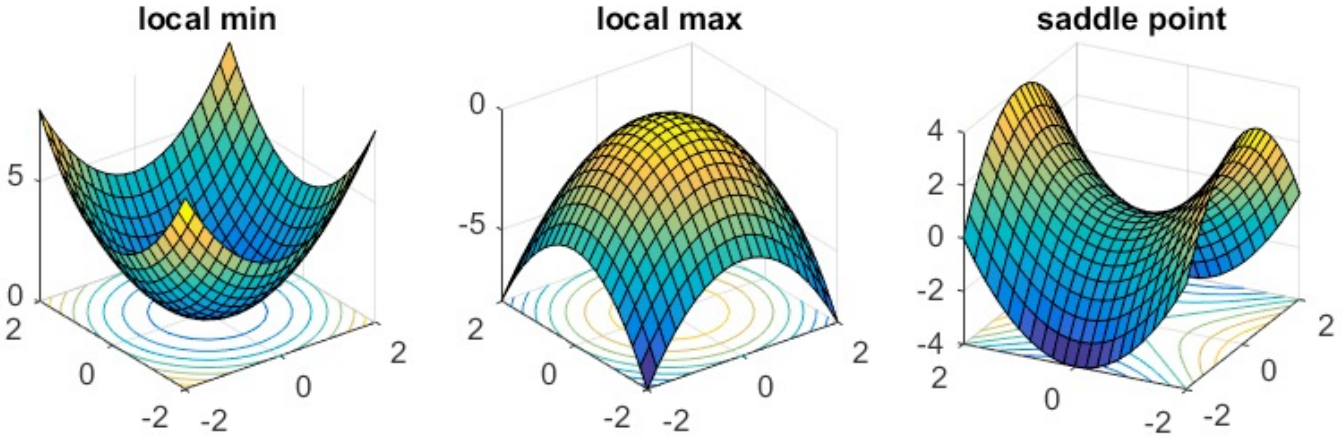
Rearrangement of neurons in a layer results in equivalent configurations due to symmetry

- A local minimum is problematic if it has a high cost compared to the global minimum
  - It is believed that for many problems including learning deep nets, almost all local minimum have very similar function value to the global optimum, so
  - Finding a local minimum is good enough

Choromanska et al. The loss surfaces of multilayer networks, 2015

Local minima and saddle points

- For high dimensional loss functions, local minima are rare compared to saddle points
- Critical points: $\nabla_x f(x) = 0$ $\quad f : \mathbb{R}^n \to \mathbb{R}$



- How to distinguish between critical points?
  - Second order derivatives: Hessian matrix

Dauphin et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. NIPS 2014

Local minima and saddle points

- Hessian matrix:

  real, symmetric

  eigenvector/eigenvalue decomposition

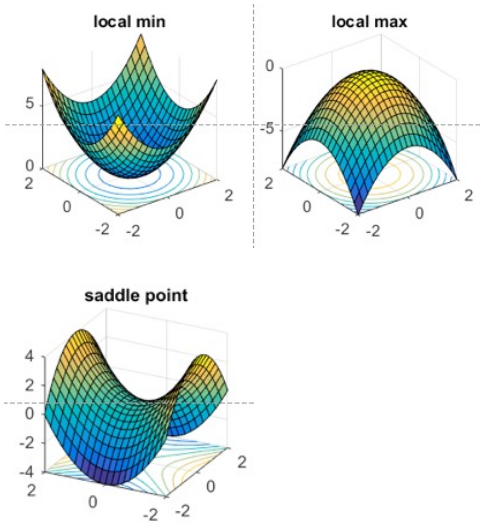
$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

- Intuition: eigenvalues of the Hessian matrix
  - local minimum/maximum: all positive / all negative eigenvalues

    exponentially unlikely as n grows

  - saddle points: both positive and negative eigenvalues

    much more common as n grows



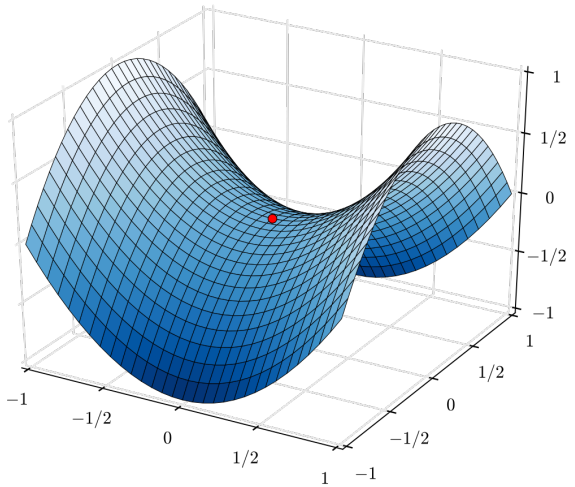
Dauphin et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. NIPS 2014

Saddle points

Does GD get stuck at saddle points?

How to escape from saddle points?

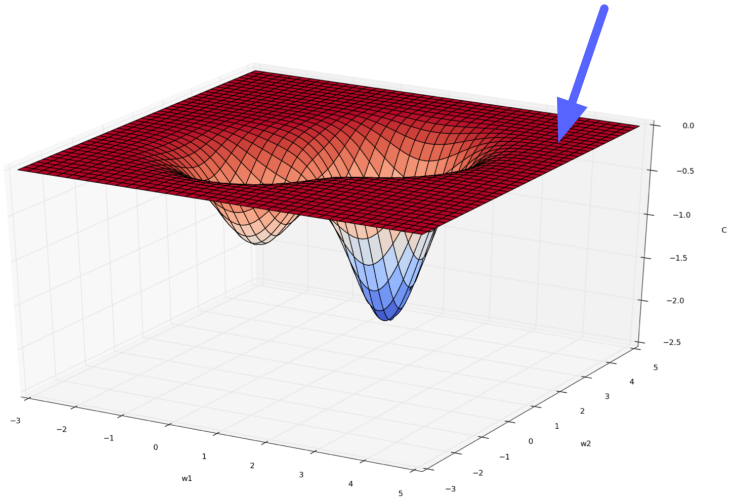
- First order methods (GD and variants)
  - initially attracted to saddle points, but unless exact hit, it will be repelled when close
  - hitting critical point exactly is unlikely (estimated gradient is noisy)
  - saddle points are very *unstable*: noise (stochastic gradient descent) helps convergence, trajectory escapes quickly



Plateaus

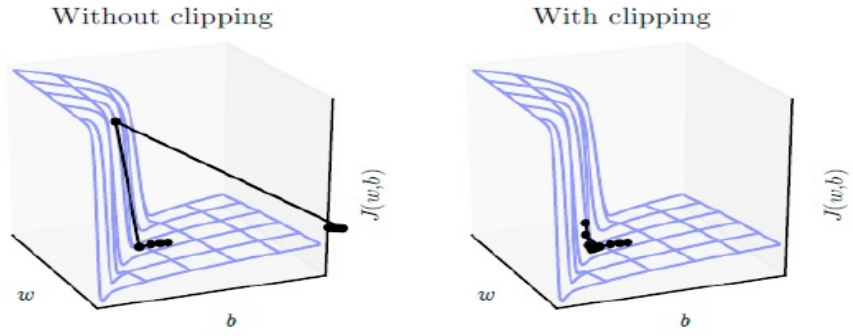
- A plateau is a region where the loss function is flat, or nearly flat: the gradient is zero or very small
- Can cause GD to get stuck in a very bad solution

- Can sometimes be avoided using
  - Careful initialization
  - Non saturating activation functions
  - Dynamic gradient scaling
  - Network design
  - Loss function design



Other difficulties

- Cliffs and exploding gradients
  - Nets with many layers / recurrent nets can contain **very steep regions** (cliffs): gradient descent can move parameters too far, jumping off of the cliff. (solutions: gradient clipping)

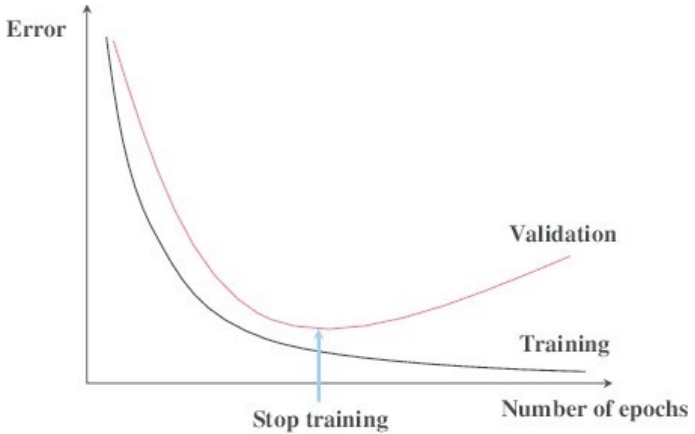


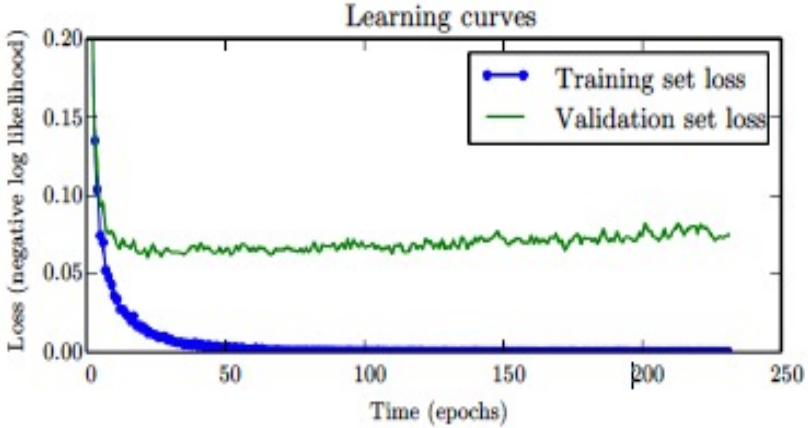
cost function of highly non linear deep nets or recurrent net (Pascanu2013)

- Long term dependencies
  - computational graph becomes very deep (deep nets / recurrent nets)**: vanishing and exploding gradients**

Early stopping

- Training algorithms usually **do not halt at a local minimum**
- Convergence criterion based on early stopping:
  - **based on surrogate loss or true underlying loss** (ex 0-1 loss) **measured on a validation set**
  - # training steps = hyperparameter controlling the effective capacity of the model
  - simple, effective, must keep a copy of the best parameters
  - acts as a regularizer (Bishop 1995,…)





Training error decreases steadily; Validation error begins to increase

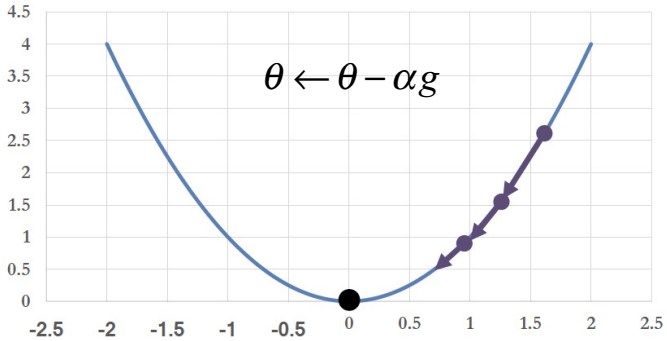
Return parameters at point with lowest validation error

Batch and mini-batch gradient descent

Notation

- Training set $\{(x_i, y_i)\}_{i=1....N}$
- Samples from a subset or "minibatch of size m $\{x_i\}_{i=1....m}$ $\{y_i\}_{i=1....m}$
- Parameters θ
- Prediction for x_i (forward pass) $f_\theta(x_i)$
- Loss for sample x_i $L(f_\theta(x_i), y_i)$
- Gradient estimate on minibatch $\frac{1}{m}\nabla_\theta \sum_i L(f_\theta(x_i), y_i)$

Recall: Gradient Descent



Algorithm

- **Require:** initial parameters θ, **learning rate α**,
- **while** stopping criterion not met **do**
  - compute gradient estimate $g \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f_\theta(x_i), y_i)$
  - apply update $\theta \leftarrow \theta - \alpha g$
- end while

Note: we omit temporal index t:

$$\theta_{t+1} \leftarrow \theta_t - \alpha g$$

Batch and mini-batch algorithms

- Gradient descent at each iteration computes gradients over the entire dataset for one update

$$\nabla_\theta J(\theta) = \frac{1}{N} \nabla_\theta \sum_i L(f_\theta(x_i), y_i)$$

N dataset size

- ↑ Gradients are stable
- ↓ Using the complete training set can be very **expensive** for large datasets
- ↓ Training set may be **redundant**
- Solution: Use a subset of the training set -> minibatch gradient descent

Batch and mini-batch algorithms

- Use a subset of the training set

  Loop:

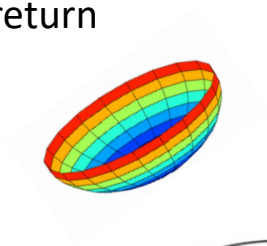
  1. sample a **subset** of data
  2. forward prop through the network
  3. backprop to calculate gradients
  4. update parameters using gradients

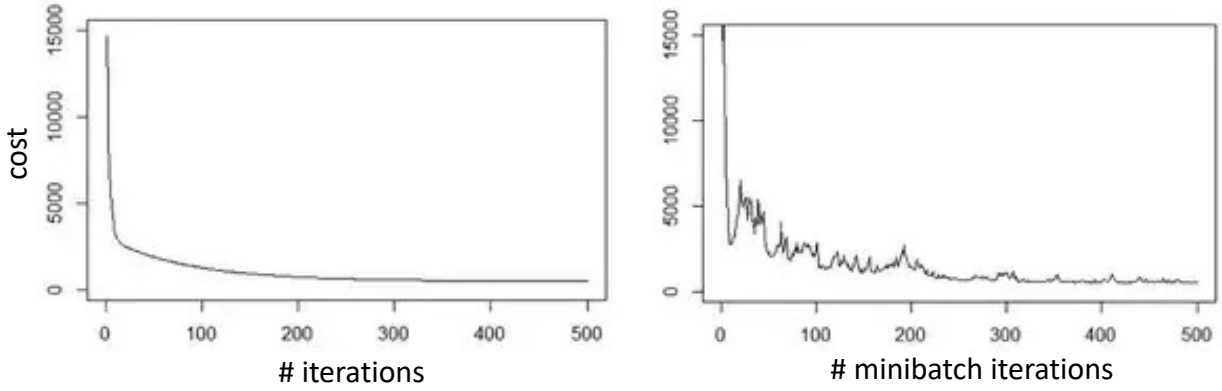
How many samples in each update step?

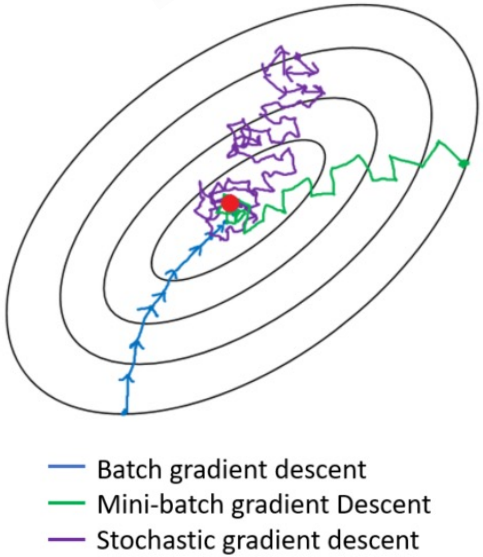
- Deterministic or batch methods: process **all training samples**
- Mini-batch methods: **use several (not all)** samples
- Stochastic methods: use **a single example** at a time
  - online methods: samples are drawn from a stream of continually created samples

Batch and mini-batch algorithms

- Larger batches: more accurate estimate of the gradient but less than linear return
- Smaller batches: provide noisier gradient estimates
  - may have a regularizing effect (add noise)
  - but may require small learning rate
  - may increase number of steps for convergence







batch vs mini-batch gradient descent

Batch and mini-batch algorithms

Depending on the amount of data, tradeoff

- The **accuracy** of the parameter update
- The **time** it takes to perform an update

Method	Accuracy	Time	Memory usage	Online learning
Batch	+	slow	high	no
Mini-batch	+	medium	medium	yes
Stochastic	-	high	low	yes

If small training set (and fits in memory), use batch gradient descent

If large training set, use mini batches (randomly selected)

Algorithms

Mini-batch Gradient Descent

- Most used algorithm for deep learning

Algorithm

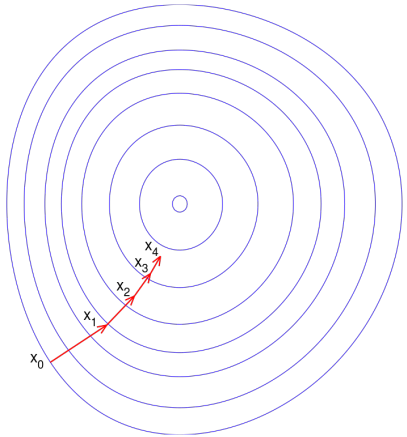
- **Require:** initial parameters θ, **learning rate α**,
- **while** stopping criterion not met **do**
  - sample a minibatch of *m* examples from the training set $\{x_i\}_{i=1....m}$ with $\{y_i\}_{i=1....m}$ corresponding targets
  - compute gradient estimate $g \leftarrow +\frac{1}{m}\nabla_\theta \sum_i L(f_\theta(x^{(i)}), y^{(i)})$
  - apply update $\theta \leftarrow \theta - \alpha g$
- end while

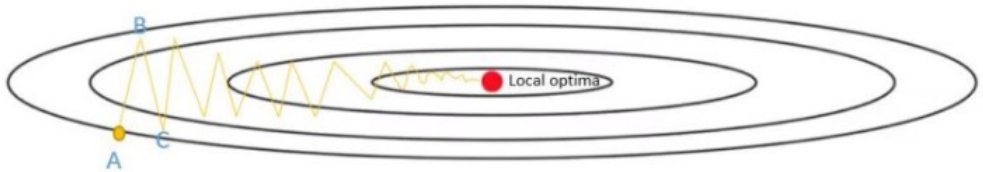
Problems with GD

- **GD can be very slow!!**
- GD can get stuck in local minima or saddle points (not SGD)

The gradient isn't always a good indicator of the good trajectory

Going downhill reduces error, but the direction of steepest descent (gradient) does not point at the minimum unless the ellipse is a circle

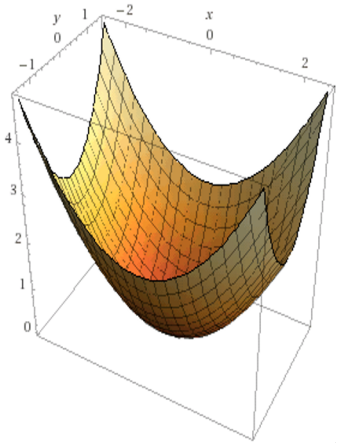


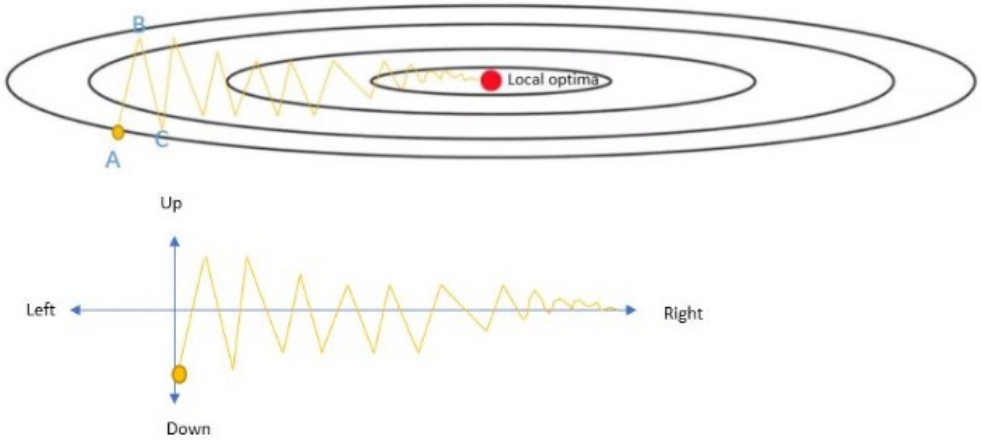


Error surface around a local minimum

Problems with GD

- If the loss changes quickly in one direction and slowly in another, GD makes slow progress along shallow dimension, jitters along steep direction
  - Gradient is big in the direction in which we only want to travel a small distance
  - Gradient is small in the direction in which we want to travel a big distance
  - Loss has an ill-conditioned Hessian matrix: gradients fluctuate widely





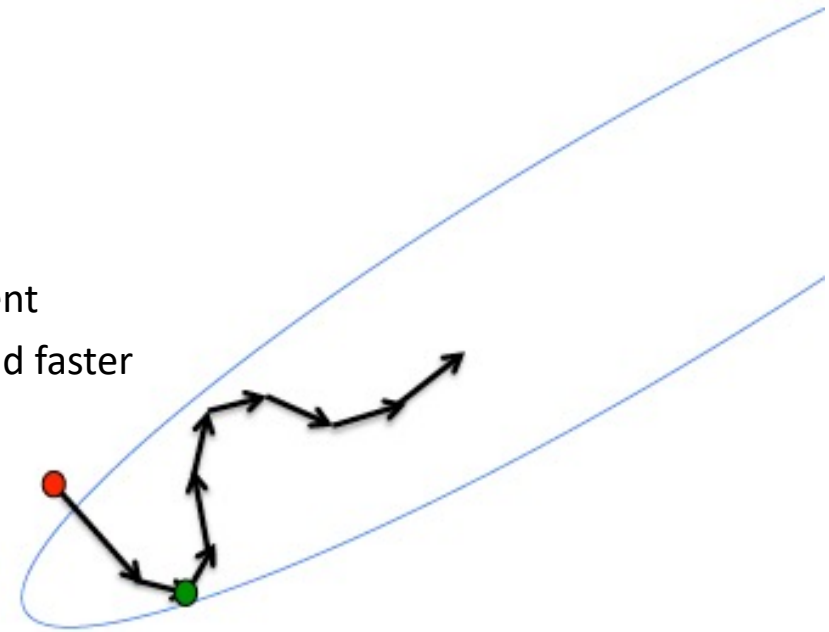
This loss function has a high **condition number** : ratio of largest to smallest singular value of Hessian matrix is large

Momentum

- Momentum is designed to accelerate learning, especially for high curvature, small but consistent gradients or noisy gradients

- **Physics model:** a ball on the error surface
  - It starts following the gradient (acceleration)
  - One it has velocity it does not follow the gradient

  but keeps going in the previous direction, faster and faster



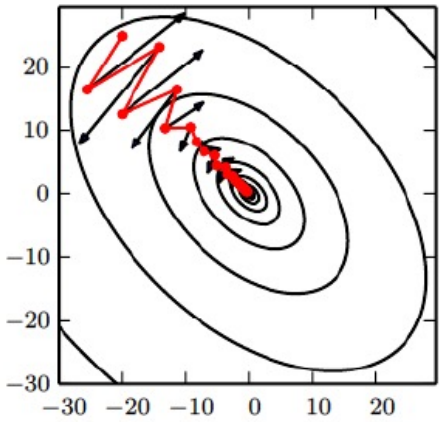
- Momentum uses a new variable **velocity v**
  - direction and speed at which parameters move

Momentum

- New variable **velocity v** : exponentially decaying average of negative gradient

Algorithm

- **Require:** initial parameter θ, learning rate α, **momentum parameter λ , initial velocity v**
- **Update rule:** (g is gradient estimate)
  - compute velocity update $v \leftarrow \lambda v - \alpha g$
  - apply update $\theta \leftarrow \theta + v$



Black: GD steps
Red: momentum steps

- Typical values v_0=0, λ = 0.5, **0.9**, 0.99 (in [0,1})

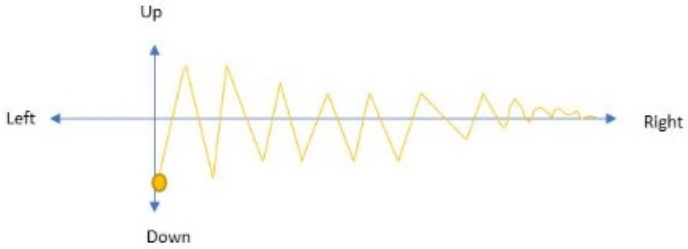
Momentum

- New variable **velocity v** : exponentially decaying average of negative gradient

$$
\begin{aligned}
v_0 &= 0 \\
v_1 &= \lambda v_0 - \alpha g_1 = -\alpha g_1 \\
v_2 &= \lambda v_1 - \alpha g_2 = -\lambda\alpha g_1 - \alpha g_2 = \alpha(-\lambda g_1 - g_2) \\
v_3 &= \alpha(-\lambda^2 g_1 - \lambda g_2 - g_3) \\
&\dots \\
v_n &= \alpha(-\lambda^{n-1} g_1 - \lambda^{n-2} g_2 \dots - \lambda g_{n-1} - g_n)
\end{aligned}
$$

$\lambda=0.9$ $\simeq$ average last 10 values
$\lambda=0.98$ $\simeq$ average last 50 values

- The momentum term v increases (reduces) for dimensions whose gradients point in the same (different) directions.



Nesterov accelerated gradient (NAG)

- Interpretation: 'smarter' ball, that has a notion of where it is going

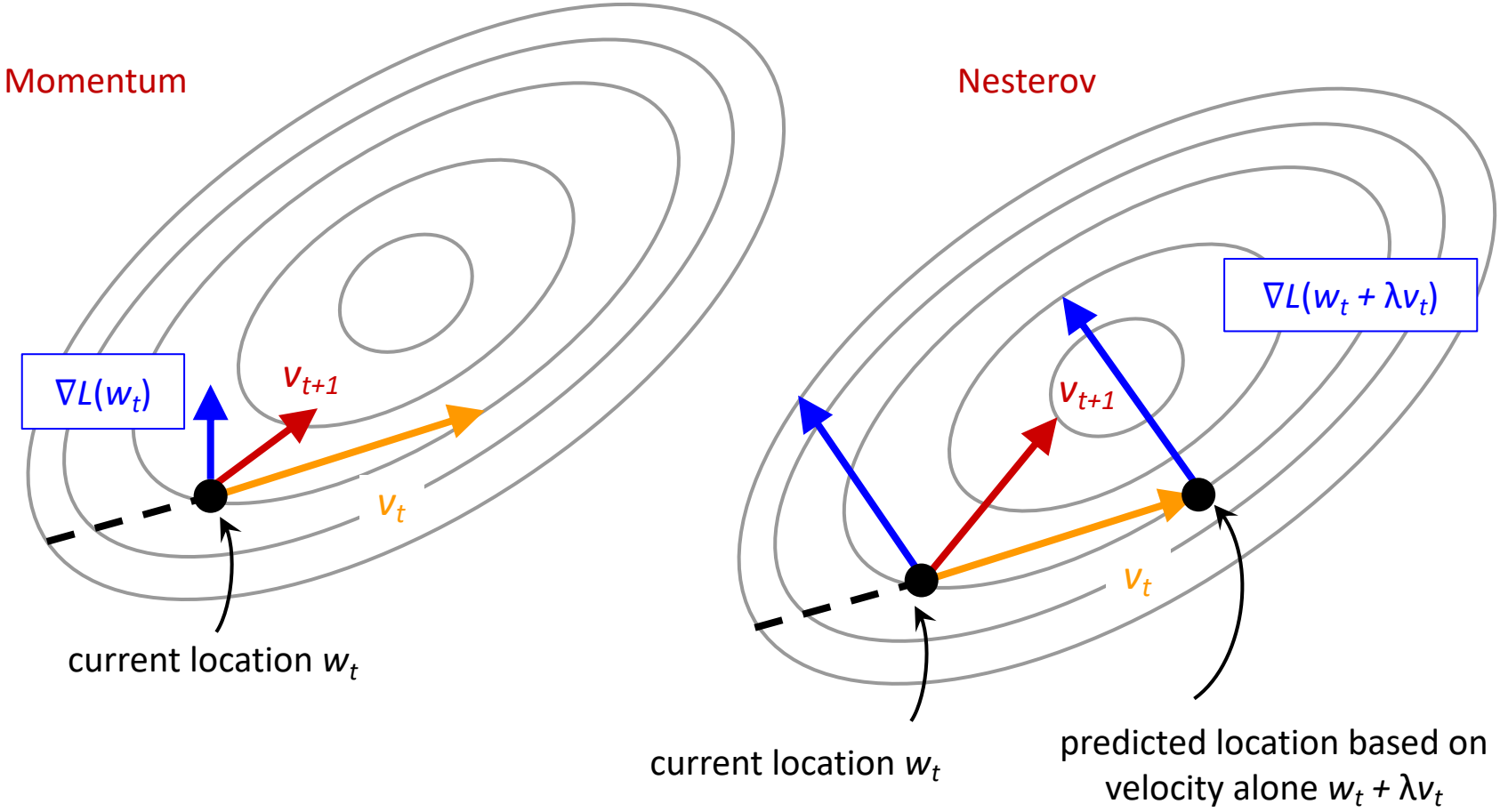
- A variant of momentum, where gradient is evaluated after current velocity is applied:
  - **Approximate** where the parameters will be on the next time step using current velocity
  - **Update velocity using gradient where we predict parameters will be**

Algorithm

- **Require:** initial parameter θ, learning rate α, momentum parameter λ , initial velocity v
- **Update:**
  - apply interim update $\tilde{\theta} \leftarrow \theta + \lambda v$
  - compute gradient (at interim point) $g \leftarrow +\frac{1}{m}\nabla_{\tilde{\theta}} \sum_i L_{\tilde{\theta}}(f(x^{(i)}), y^{(i)})$
  - compute velocity update $v \leftarrow \lambda v - \alpha g$
  - apply update $\theta \leftarrow \theta + v$

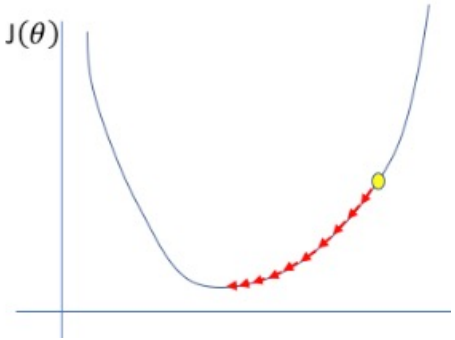
interim

Nesterov accelerated gradient (NAG)

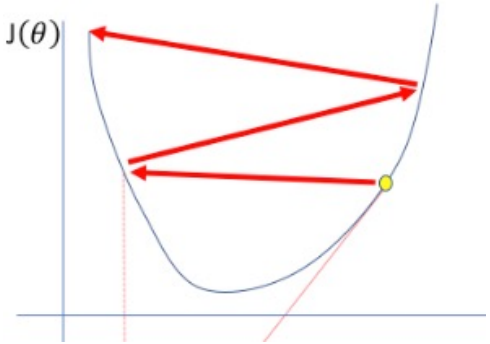


GD: learning rate

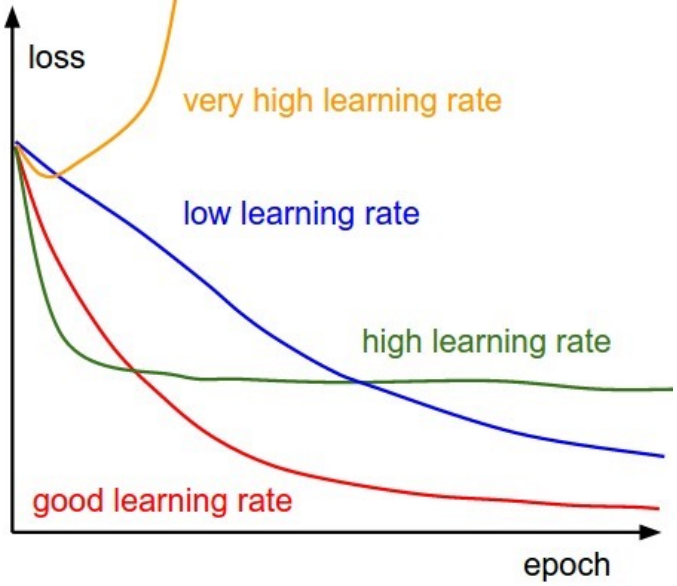
- In GD the learning rate is **the same** for all parameters
- Learning rate is a crucial hyperparameter for GD
  - Too small: makes very slow progress, can get stuck
  - Too large: overshoots local minimum, loss increases
  - Good learning rate: makes steady progress toward local minimum



too small



too large



GD: learning rate decay

- In practice it is necessary to gradually decrease learning rate to speed up the training
  - **step decay** (e.g. reduce by half every few epochs)
  - **exponential decay** $\alpha = \alpha_0 e^{-kt}$
  - **1/t decay** $\alpha = \frac{\alpha_0}{1+kt}$
  - **manual decay**

k *decay rate*

t *iteration number*

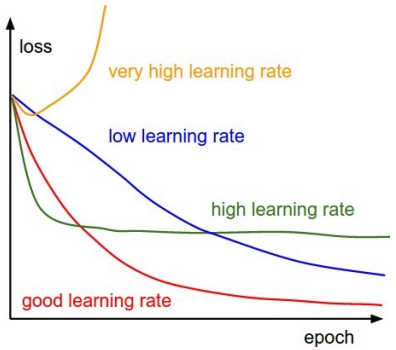
α_0 *initial learning rate*

Sufficient conditions for convergence:

$$\sum_{t=1}^{\infty} \alpha_t = \infty$$

$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

Usually: adapt learning rate by monitoring learning curves that plot the objective function as a function of time(**more of an art than a science!**)



Adaptive learning rates

- Loss if often sensitive to some directions and insensitive to others
  - Momentum/Nesterov mitigate this issue but introduce another hyperparameter

- Solution: **Use a separate learning rate for each parameter** and automatically adapt it through the course of learning

- Algorithms (mini-batch based)
  - AdaGrad
  - RMSProp
  - Adam

AdaGrad

- Adapts the learning rate of each parameter based on sizes of previous updates:

Notation:

- uses an accumulation of the **historical gradients (squared)**: new variable r

$$r \leftarrow r + g \odot g$$ elementwise multiplication

- **divides global learning rate** by sqrt of accumulated gradients

$$\frac{\alpha}{\delta + \sqrt{r}}$$ δ small constant for numerical stability

Effect:	parameters with largest gradients → rapid decrease in learning rate parameters with smaller gradients → small decrease in learning rate

- The net effect is greater progress in the more gently sloped directions of parameter space

AdaGrad

Algorithm

- **Require:** initial parameter θ, learning rate α, small constant δ (*e.g. 10^{-7}) for numerical stability*
- **Update:**
  - accumulate squared gradient
  - compute update
  - apply update

$$r \leftarrow r + g \odot g$$

$$\Delta\theta \leftarrow -\frac{\alpha}{\delta + \sqrt{r}} \odot g$$

$$\theta \leftarrow \theta + \Delta\theta$$

sum of all previous squared gradients

updates inversely proportional to the square root of the sum (elementwise multiplication)

Duchi et al. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. JMRL 2011

Root Mean Square Propagation (RMSProp)

- AdaGrad **can result in a premature and excessive decrease in effective learning rate**
- RMSProp modifies AdaGrad to perform better in non-convex surfaces
- Changes gradient accumulation by an exponentially decaying average of sum of squares of gradients

- **Requires:** initial parameter θ, learning rate α, **decay rate ρ**, small constant δ (*e.g. 10^{-7}*)
- **Update:**
  - accumulate squared gradient $r \leftarrow \rho r + (1-\rho) g \odot g$
  - compute update $\Delta\theta \leftarrow -\frac{\alpha}{\delta + \sqrt{r}} \odot g$
  - apply update $\theta \leftarrow \theta + \Delta\theta$

Geoff Hinton, Unpublished

ADAptive Moments (Adam)

- Combination of RMSProp and momentum, but:
  - Keep decaying average of **both first-order moment of gradient (momentum) and second-order moment** (RMSProp)
  - Includes bias corrections (first and second moments) to account for their initialization at origin (s=0, r=0)

Update:

- updated biased first moment estimate $s \leftarrow \rho_1 s + (1-\rho_1) g$
- update biased second moment $r \leftarrow \rho_2 r + (1-\rho_2) g \odot g$
- correct biases $\hat{s} \leftarrow \frac{s}{1-\rho_1}$ $\hat{r} \leftarrow \frac{r}{1-\rho_2}$
- compute update $\Delta\theta \leftarrow -\alpha \frac{\hat{s}}{\delta + \sqrt{\hat{r}}}$ (operations applied elementwise)
- apply update $\theta \leftarrow \theta + \Delta\theta$

Usual values
$\delta = 10^{-8}$, $\rho 1 = 0.9$, $\rho 2 = 0.999$

Kingma et al. Adam: a Method for Stochastic Optimization. ICLR 2015

Summary

Vanilla GD

$$\theta \leftarrow \theta - \alpha g$$

Momentum

$$v \leftarrow \lambda v - \alpha g$$
$$\theta \leftarrow \theta + v$$

Nesterov

$$\tilde{\theta} \leftarrow \theta + \lambda v$$
$$v \leftarrow \lambda v - \alpha g$$
$$\theta \leftarrow \theta + v$$

Adagrad

$$r \leftarrow r + g \odot g$$
$$\Delta\theta \leftarrow -\frac{\alpha}{\delta + \sqrt{r}} \odot g$$
$$\theta \leftarrow \theta + \Delta\theta$$

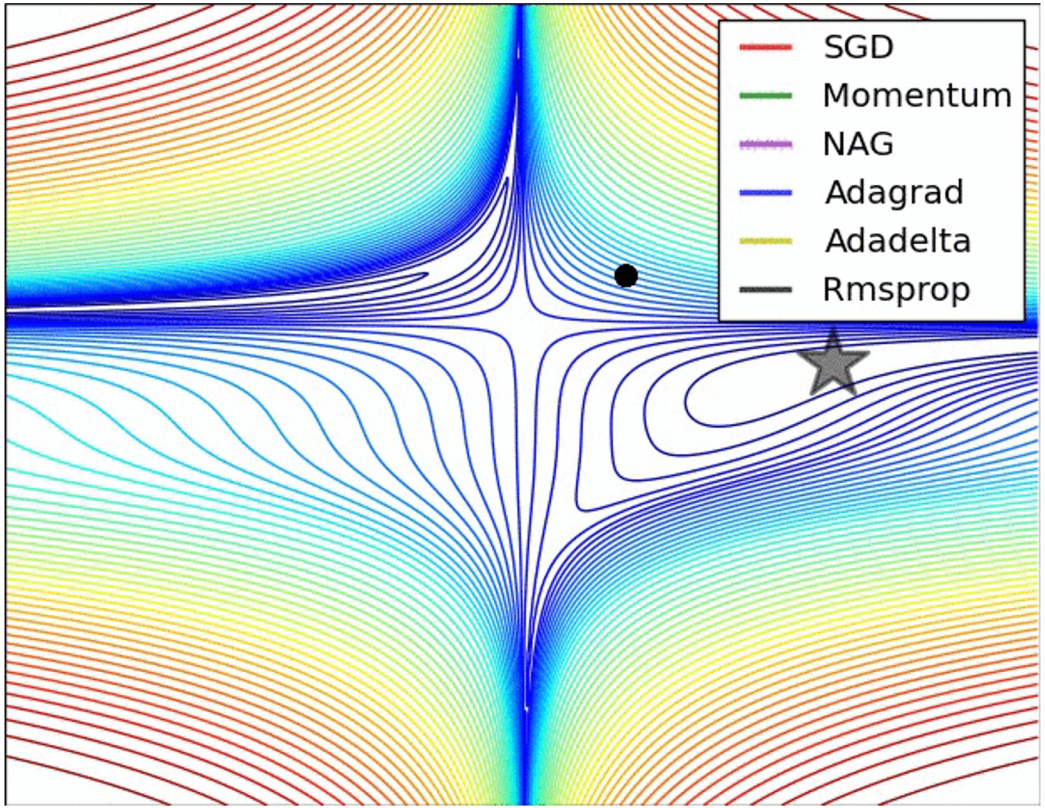
RMSprop

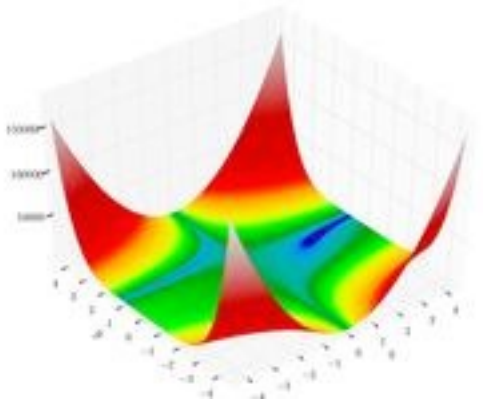
$$r \leftarrow \rho r + (1-\rho) g \odot g$$
$$\Delta\theta \leftarrow -\frac{\alpha}{\delta + \sqrt{r}} \odot g$$
$$\theta \leftarrow \theta + \Delta\theta$$

Adam

$$s \leftarrow \rho_1 s + (1-\rho_1) g$$
$$r \leftarrow \rho_2 r + (1-\rho_2) g \odot g$$
$$\hat{s} \leftarrow \frac{s}{1-\rho_1} \qquad \hat{r} \leftarrow \frac{r}{1-\rho_2}$$
$$\Delta\theta \leftarrow -\alpha \frac{\hat{s}}{\delta + \sqrt{\hat{r}}}$$
$$\theta \leftarrow \theta + \Delta\theta$$

Example: test function





Beale's function

Image credit: Alec Radford.

$$f(\mathbf{x}) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2$$

Example: saddle point

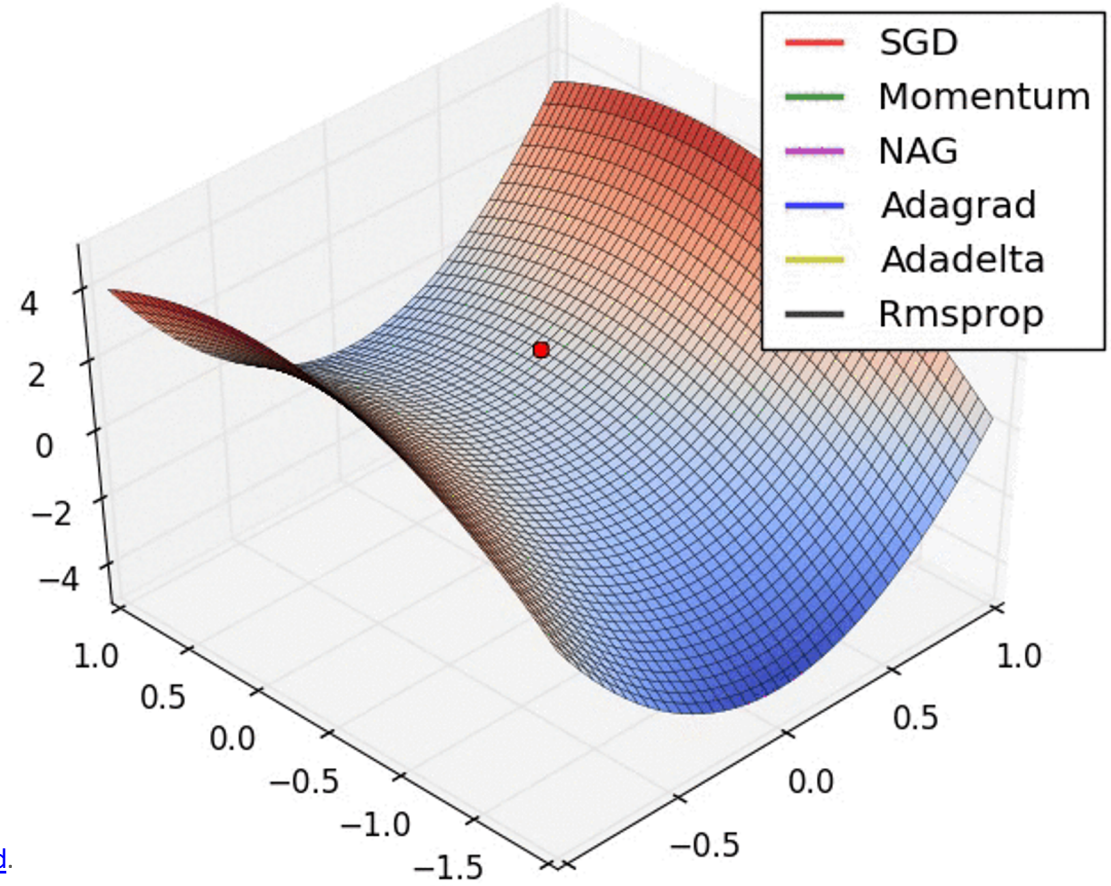


Image credit: Alec Radford.

Parameter initialization

Parameter initialization

Most algorithms are strongly affected by the choice of initialization

- **Weights**
  - Can't initialize weights to 0 (gradients would be 0)
  - Can't initialize all weights to the same value (all hidden units in a layer will always behave the same; need to break symmetry)
  - Small random number (e.g. uniform or Gaussian distribution) ; keep same variance at output
    - "Xavier" initialization (for tanh activations) sqrt(1/n)
      - each neuron: w = randn(n) / sqrt(n) , n inputs
    - "He" initialization (for ReLu activations) sqrt(2/n)
      - each neuron w = randn(n) * sqrt(2.0 /n) , n inputs
- **Biases**
  - initialize all to 0 (except for output unit for skewed distributions, 0.01 to avoid saturating RELU)
- **Alternative:** Initialize using machine learning; parameters learned by unsupervised model trained on the same inputs / trained on unrelated task

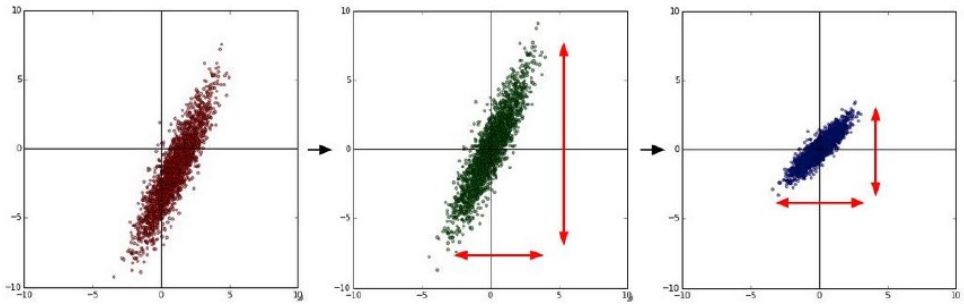
Normalization

Inputs, activations

Normalizing inputs

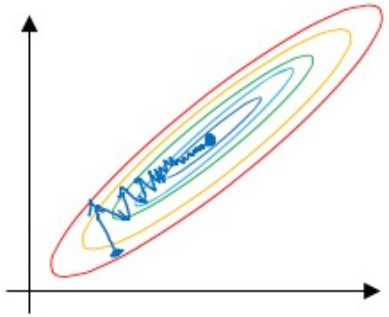
Goal: to speed up learning

- Normalize (scale) values between 0 and 1
- Normalize (scale) values between -1 and 1
- Normalize values to mean = 0, std=1
  - Mean ,std computed from **training** data

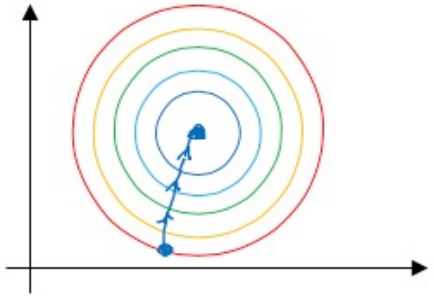


original data mean=0 mean =0, std=1

A simple 2D case



Loss for unnormalized data



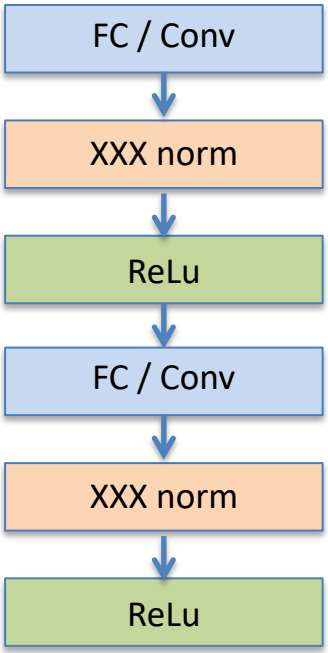
Loss for normalized data

Normalizing activations

Goal: to speed up learning

Different strategies XXX have been proposed

- **Batch** normalization
- **Layer** normalization
- **Instance** normalization
- **Group** normalization
- …



Normalizing activations

General notation: x feature computed by a layer, i index

1. Compute empirical mean and variance

Mean: $\mu_i = \frac{1}{m}\sum_{k \in S_i} x_k$ Standard deviation: $\sigma_i = \sqrt{\frac{1}{m}\sum_{k \in S_i}(x_k - \mu_i)^2 + \epsilon}$

S_i is the set of "points" in which the mean and std are computed, m is the size of S_i

2. Normalize $\hat{x_i} = \frac{1}{\sigma_i}(x_i - \mu_i)$

3. Scale and shift $y_i = \gamma \hat{x_i} + \beta$ (two learnable parameters)

To recover the identity mapping

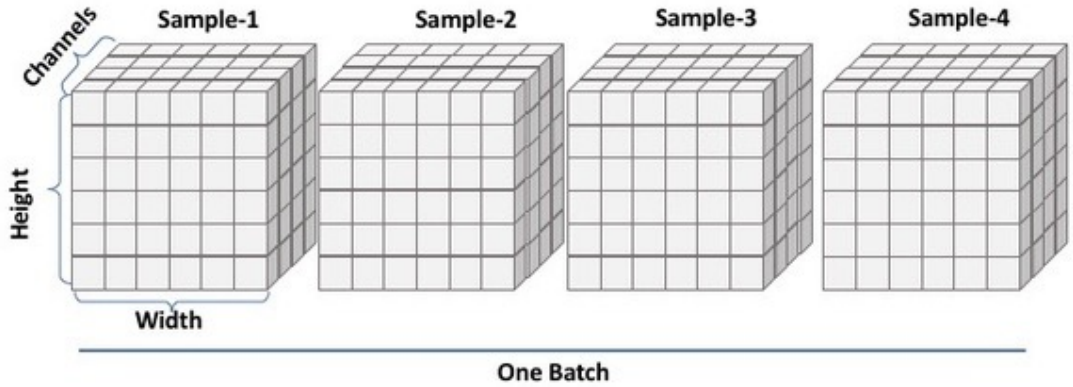
Normalization

- Let's assume h is a feature map in a CNN, that means h has four dimensions:

$$<batch;channel;width;height>,$$

- we can represent h as a 4-dimensional tensor: h_{ncij}

- Example with a batch of size 4 samples each having 4 feature maps of size 6×6:

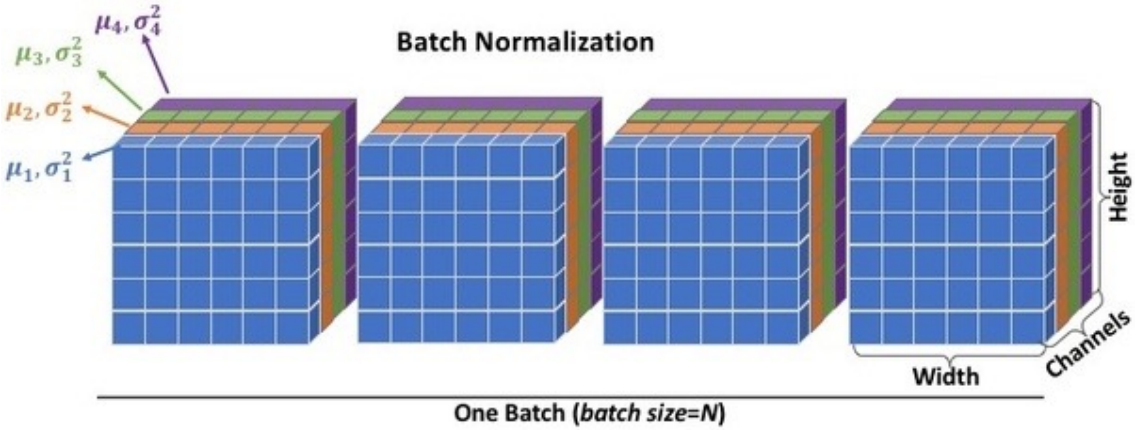


Source Vahid Mirjalili

Batch normalization

- **Batch Normalization** will compute two scalars, μ and σ^2, for each channel.
- Example: each colored group of cells results in one scalar μ and one scalar σ^2.
  In other words, one (μ,σ^2) from all the blue cells, and so on. Therefore, $\mu \in \mathbb{R}^C$ and $\sigma^2 \in \mathbb{R}^C$.

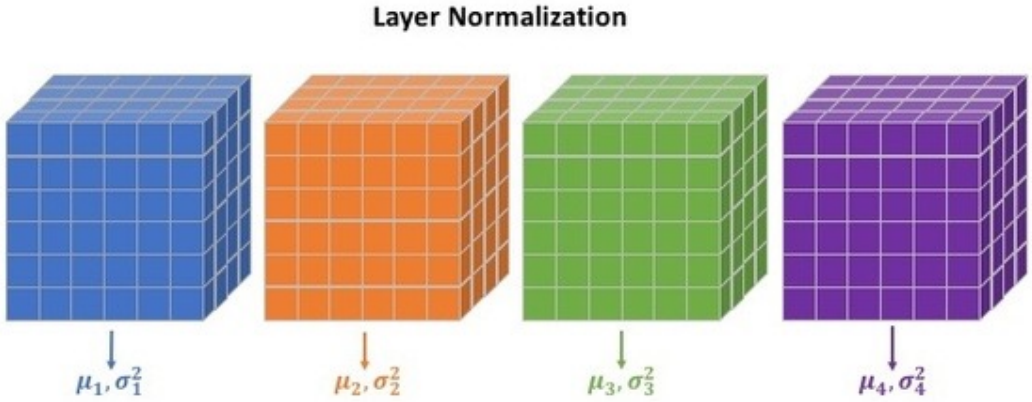
  Note: C denotes the number channels, and N is the number of samples in a batch (batch size).



Source Vahid Mirjalili

Layer normalization

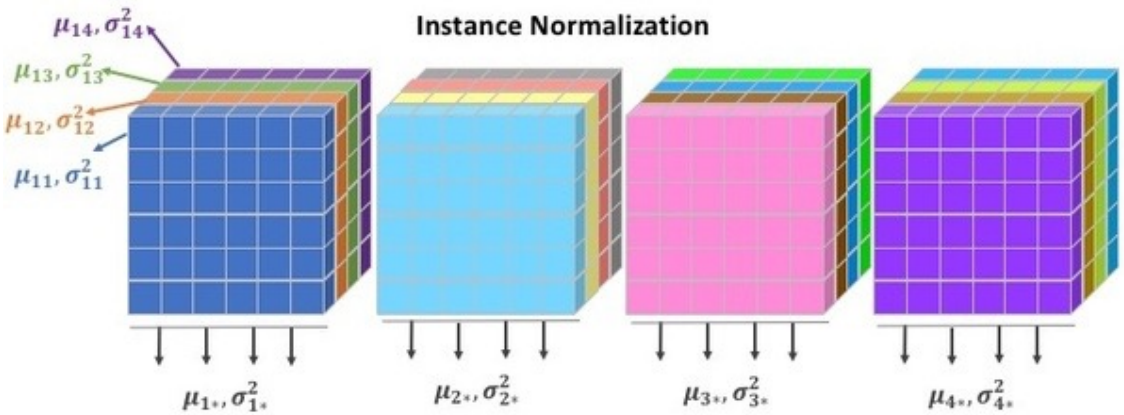
- **Layer Normalization** will compute two scalars μ and σ^2 for each sample. Therefore, $\mu \in \mathbb{R}^N$ and $\sigma^2 \in \mathbb{R}^N$.



Source Vahid Mirjalili

Instance normalization

- **Instance Normalization** will compute two scalars μ and σ^2 for each sample and each channel separately. Therefore, $\mu \in \mathbb{R}^{N \times C}$ and $\sigma^2 \in \mathbb{R}^{N \times C}$.



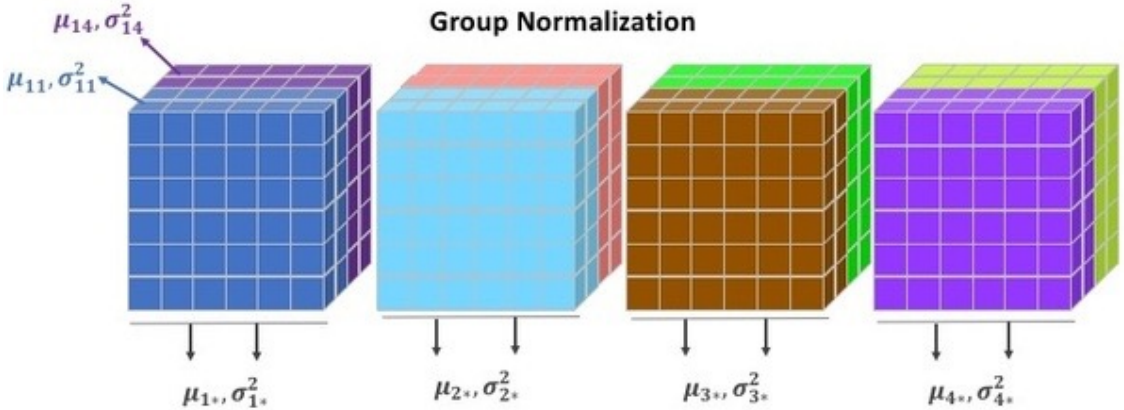
Source Vahid Mirjalili

Group normalization

- **Group Normalization** with a group size g, will group the channels into multiple groups, and then computes two scalars μ and σ^2 for each sample and each group.

- Example: there are 4 channels, and group size is g=2.

  Therefore, there are 2 groups of channels ($groups$=2), and as a result: $\mu \in \mathbb{R}^{N \times C/g}$ and $\sigma^2 \in \mathbb{R}^{N \times C/g}$



Source Vahid Mirjalili

Summary

- Optimization for NN is different from pure optimization:
  - early stopping
  - non-convex surface, saddle points
  - GD with mini-batches
- Learning rate has a significant impact on model performance
- Several extensions to GD can improve convergence
- Adaptive learning-rate methods are likely to achieve best results
  - RMSProp, Adam
- Parameter initialization
- Normalization:
  - Inputs
  - Activations: batch, layer, instance, group normalization

Bibliograpy

- **Goodfellow, I., Bengio, Y., and A., C. (2016), Deep Learning, MIT Press.**
- Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2015), The loss surfaces of multilayer networks. In AISTATS.
- Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Advances in Neural Information Processing. Systems, pages 2933–2941.
- Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research, 12(Jul):2121–2159.
- Goodfellow, I. J., Vinyals, O., and Saxe, A. M. (2015). Qualitatively characterizing neural network optimization problems. In International Conference on Learning Representations.
- Hinton, G. (2012). Neural networks for machine learning. Coursera, video lectures
- Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. Neural networks, 1(4):295–307.
- Kingma, D. and Ba, J. (2014)- Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
- Saxe, A. M., McClelland, J. L., and Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In International Conference on Learning Representations