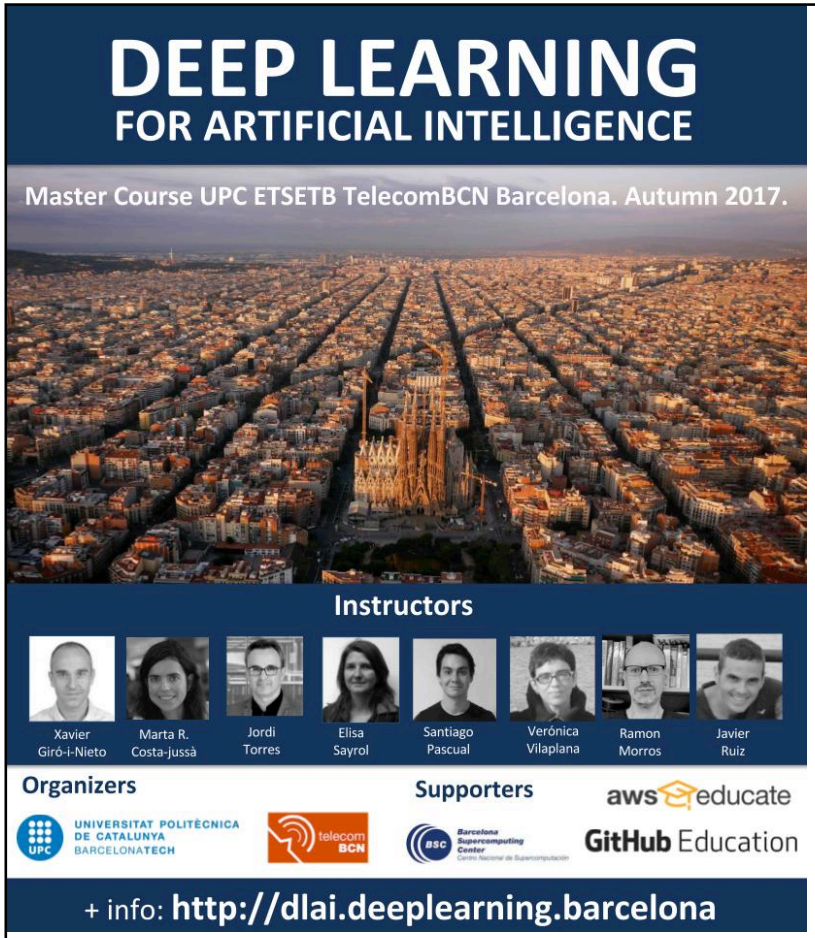


[course site]

#DLUPC

Day 4 Lecture 1

Optimization for neural network training

Verónica Vilaplana
veronica.vilaplana@upc.edu

Associate Professor
Universitat Politecnica de Catalunya
Technical University of Catalonia

Previously in DLAI…

- Multilayer perceptron
- Training: (stochastic / mini-batch) gradient descent
  - Backpropagation

but…

What type of optimization problem?

Do local minima and saddle points cause problems?

Does gradient descent perform well?

How to set the learning rate?

How to initialize weights?

How does batch size affect training?

Index

- **Optimization for a machine learning task; difference between learning and pure optimization**
  - Expected and empirical risk
  - Surrogate loss functions and early stopping
  - Batch and mini-batch algorithms

- **Challenges**
  - Local minima
  - Saddle points and other flat regions
  - Cliffs and exploding gradients

- **Practical algorithms**
  - Stochastic Gradient Descent
  - Momentum
  - Nesterov Momentum
  - Learning rate
  - Adaptive learning rates: adaGrad, RMSProp, Adam

- **Approximate second-order methods**
- **Parameter initialization**
- **Batch Normalization**

Differences between learning and pure optimization

Optimization for NN training

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

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

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

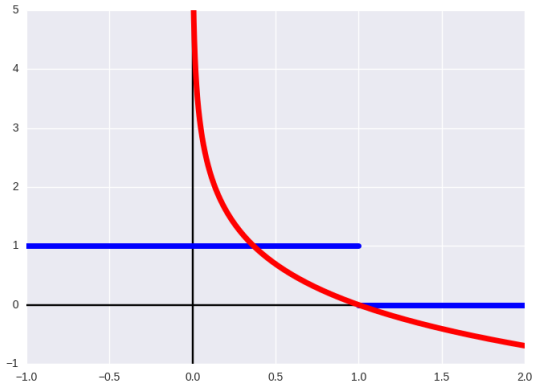
- But we only have a training set of samples: we minimize the empirical risk, average loss on a finite dataset D

$$J(\theta) = \mathrm{E}_{(x,y)\sim \hat{p}_{data}} L(f(x;\theta), y) = \frac{1}{|D|} \sum_{(x^{(i)},y^{(i)})\in D} L(f(x^{(i)},\theta), y^{(i)})$$

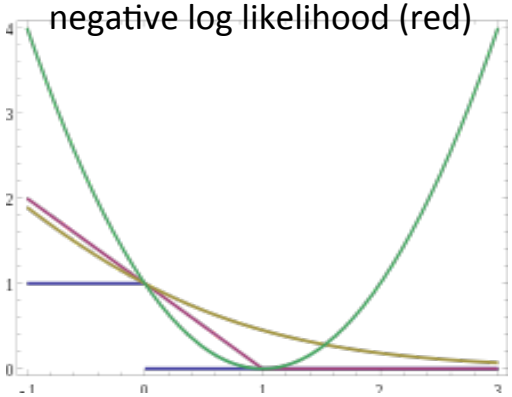
where $\hat{p}_{data}$ is the empirical distribution, $|D|$ is the number of examples in D

Surrogate loss

- Often minimizing the real loss is intractable
  - e.g. 0-1 loss (0 if correctly classified, 1 if it is not)
    (intractable even for linear classifiers (Marcotte 1992)
- Minimize a surrogate loss instead
  - e.g. negative log-likelihood for the 0-1 loss
- Sometimes the surrogate loss may learn more
  - test error 0-1 loss keeps decreasing even after training 0-1 loss is zero
  - further pushing classes apart from each other



0-1 loss (blue) and
negative log likelihood (red)



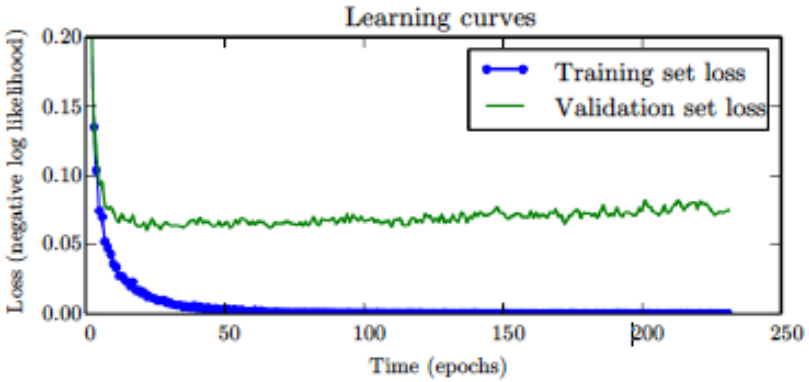
0-1 loss (blue) and surrogate
losses (square, hinge, logistic)

Surrogate loss functions

	Binary classifier	Multiclass classifier
Probabilistic classifier	Outputs probability of class 1 $g(x) \approx P(y=1 \mid x)$ Probability for class 0 is $1-g(x)$ Binary cross-entropy loss: $L(g(x),y) = -(y \log(g(x)) + (1-y) \log(1-g(x))$ Decision function: $f(x) = I_{g(x)>0.5}$	Outputs a vector of probabilities: $g(x) \approx (P(y=0\|x), \ldots, P(y=m-1\|x))$ Negative conditional log likelihood loss $L(g(x),y) = -\log g(x)_y$ Decision function: $f(x) = \text{argmax}(g(x))$
Non-probabilistic classifier	Outputs a «score» $g(x)$ for class 1. score for the other class is $-g(x)$ Hinge loss: $L(g(x),t) = \max(0, 1-tg(x))$ where $t=2y-1$ Decision function: $f(x) = I_{g(x)>0}$	Outputs a vector $g(x)$ of real-valued scores for the m classes. Multiclass margin loss $L(g(x),y) = \max(0,1+\max_{k \neq y} g(x)_k - g(x)_y)$ Decision function: $f(x) = \text{argmax}(g(x))$

Early stopping

- Training algorithms usually **do not halt at a local minimum**
- Early stopping:
  - **based on the 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 algorithms

- In most optimization methods used in ML the objective function decomposes as a sum over a training set
- Gradient descent: $\nabla_\theta J(\theta) = \mathrm{E}_{(x,y)\sim\hat{p}_{data}} \nabla_\theta L(f(x;\theta),y) = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta),y^{(i)})$

  examples from the training set $\{x^{(i)}\}_{i=1\ldots m}$ with corresponding targets $\{y^{(i)}\}_{i=1\ldots m}$
- Using the complete training set can be very expensive (the gain of using more samples is less than linear – standard error of mean drops proportionally to sqrt(m)-, training set may be redundant: **use a subset of the training set**
- How many samples in each update step?
  - Deterministic or batch gradient methods: process all training samples in a large batch
  - Stochastic methods: use a single example at a time
    - online methods: samples are drawn from a stream of continually created samples
  - Mini-batch stochastic methods: use several (not all) samples

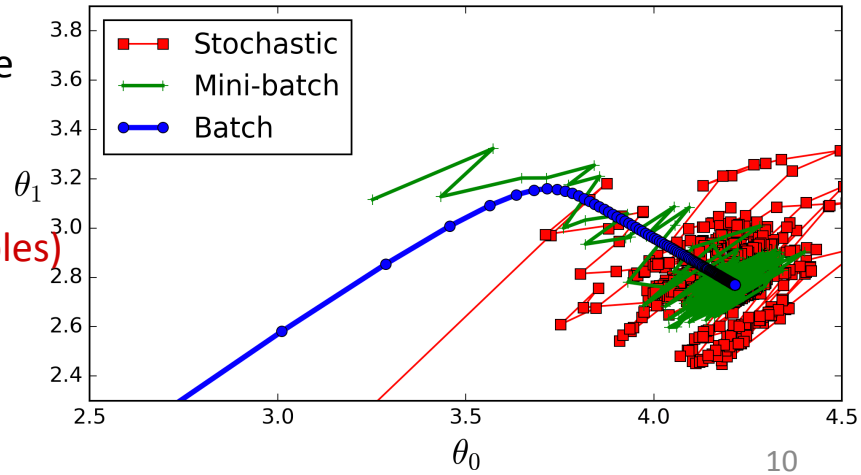
Batch and mini-batch algorithms

Mini-batch size?

- Larger batches: more accurate estimate of the gradient but less than linear return
- Very small batches: Multicore architectures under-utilized
- If samples processed in parallel: memory scales with batch size
- Smaller batches provide noisier gradient estimates
- Small batches may offer a regularizing effect (add noise)
  - but may require small learning rate
  - may increase number of steps for convergence

Minbatches should be selected randomly (shuffle samples)

- unbiased estimate of gradients

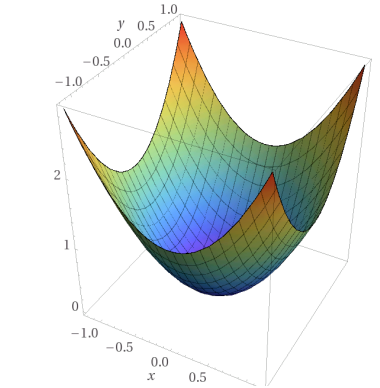


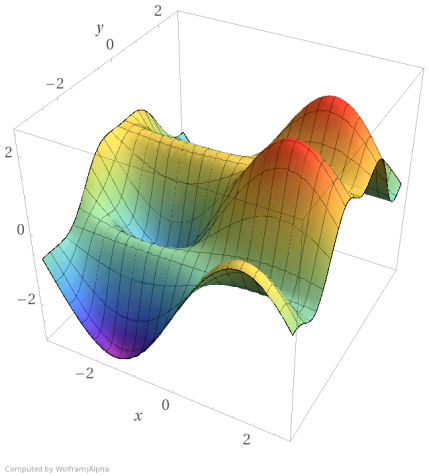
Challenges in NN optimization

Local minima

- 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!



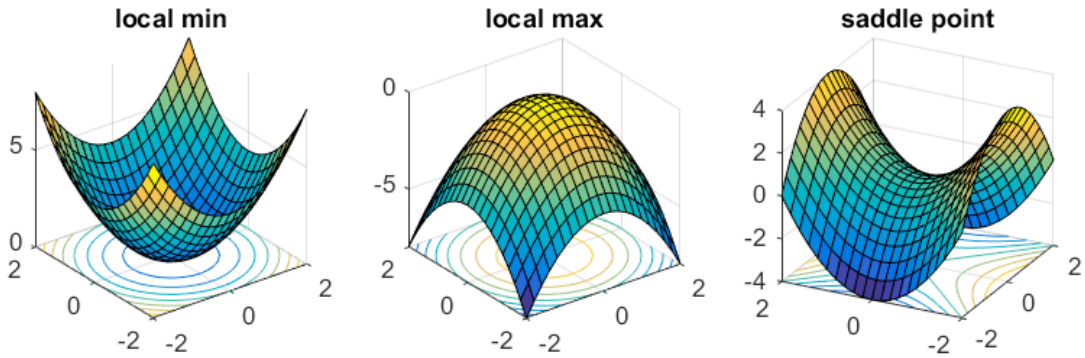


Local minima and saddle points

- Critical points:

$$f : \mathbb{R}^n \to \mathbb{R}$$

$$\nabla_x f(x) = 0$$



- For high dimensional loss functions, local minima are rare compared to 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

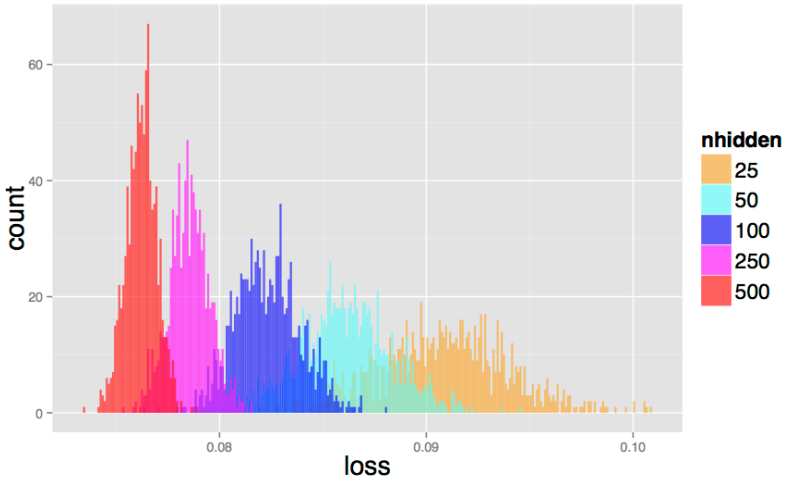
Dauphin et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. NIPS 2014

Local minima and saddle points

- For many random functions local minima are **more likely to have low cost than high cost**.

- It is believed that for many problems including learning deep nets, almost all local minimum have very similar function value to the global optimum

- Finding a local minimum is good enough

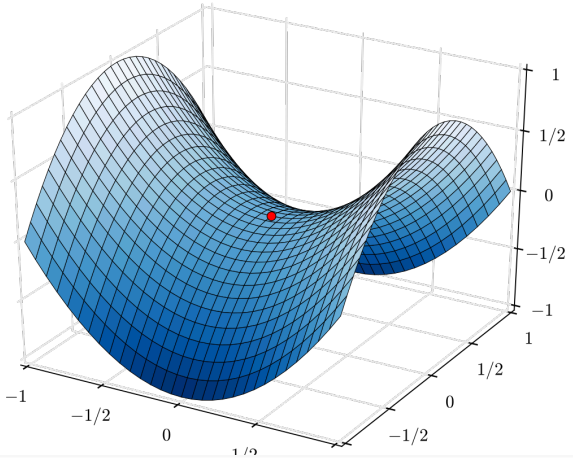


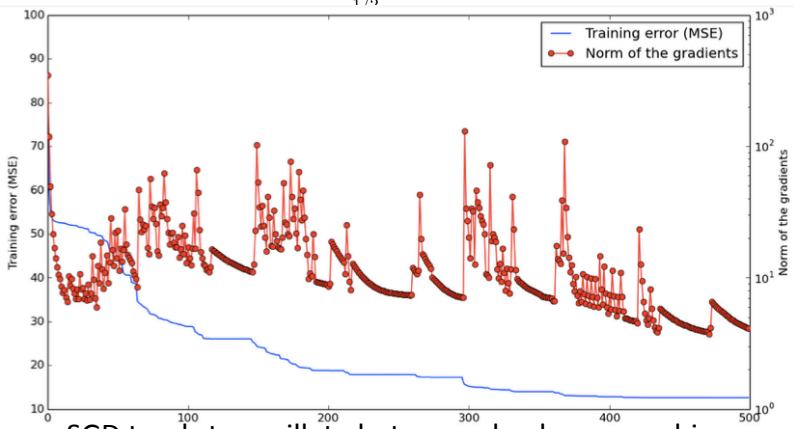
Value of local minima found by running SGD for 200 iterations on a simplified version of MNIST from different initial starting points. As number of parameters increases, local minima tend to cluster more tightly.

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

Saddle points

- How to escape from saddle points?
- First order methods
  - 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
- Second order moments:
  - Netwon's method can jump to saddle points (where gradient is 0)



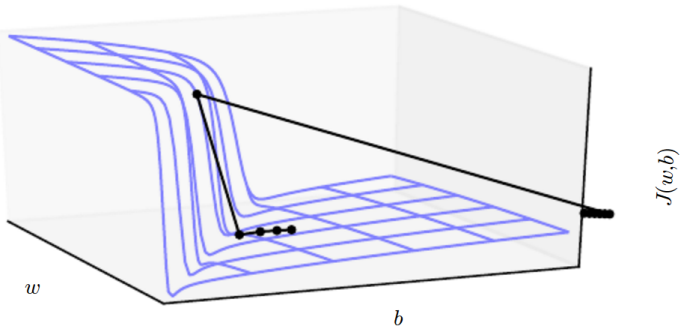


SGD tends to oscillate between slowly approaching a saddle point and quickly escaping from it

S. Credit: K.McGuinness

Other difficulties

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



- Long term dependencies:
  - computational graph becomes very deep: vanishing and exploding gradients

Algorithms

Stochastic Gradient Descent (SGD)

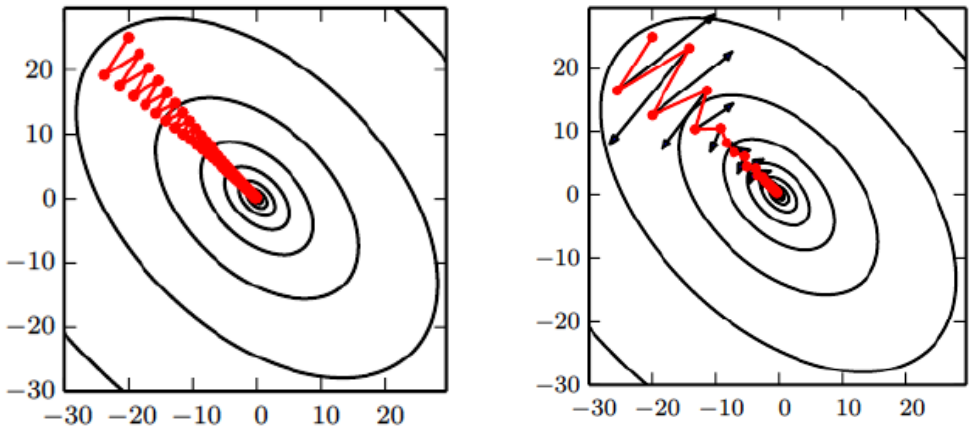
- Most used algorithm for deep learning
- Do not confuse with deterministic gradient descent: stochastic uses mini-batches

Algorithm

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

Momentum

- Designed to accelerate learning, especially for high curvature, small but consistent gradients or noisy gradients
- Momentum aims to solve: poor conditioning of Hessian matrix and variance in the stochastic gradient



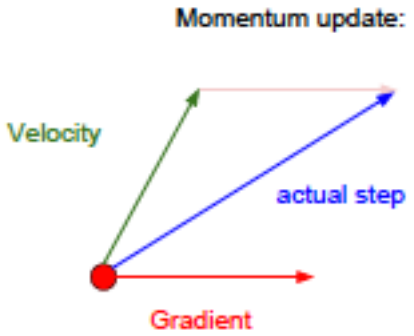
Contour lines= a quadratic loss with poor conditioning of Hessian
Path (red) followed by SGD (left) and momentum (right)

Momentum

- New variable v (velocity). direction and speed at which parameters move: exponentially decaying average of negative gradient

Algorithm

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



- Typical values λ=.5, .9,.99 (in [0,1})
- Size of step depends on how large and aligned a sequence of gradients are.
- Read physical analogy in Deep Learning book (Goodfellow et al)

Nesterov accelerated gradient (NAG)

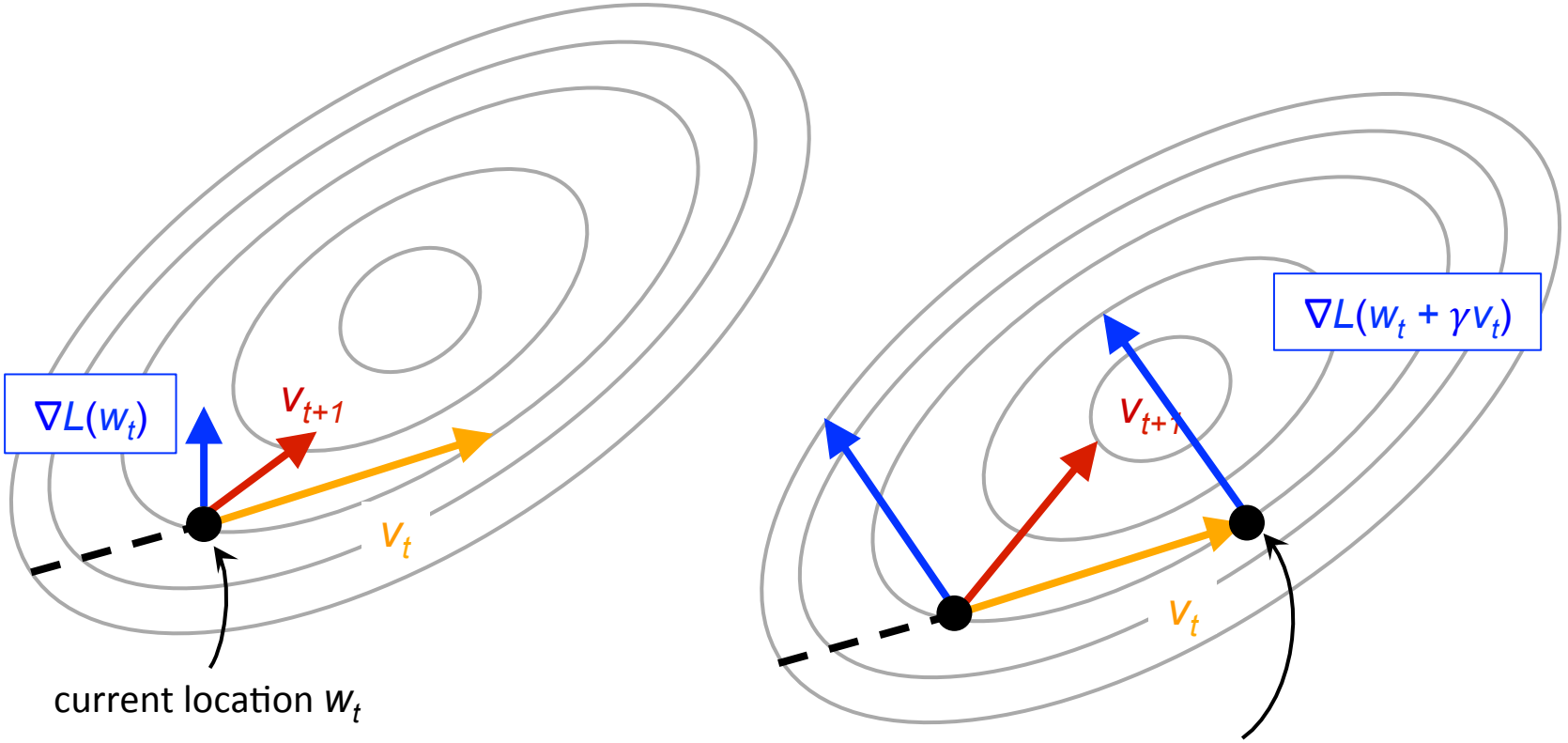
- A variant of momentum, where gradient is evaluated after current velocity is applied:
  - Approximate where the parameters will be on the next time step using current velocity
  - **Update velocity using gradient where we predict parameters will be**

Algorithm

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

- Interpretation: add a correction factor to momentum

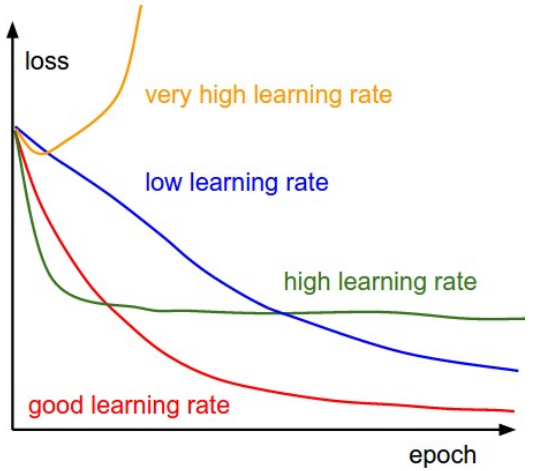
Nesterov accelerated gradient (NAG)



S. Credit: K. McGuinness

SGD: learning rate

- Learning rate is a crucial parameter for SGD
  - To large: overshoots local minimum, loss increases
  - Too small: makes very slow progress, can get stuck
  - Good learning rate: makes steady progress toward local minimum



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

t = iteration number

- Sufficient conditions for convergence: $\sum_{t=1}^{\infty} \alpha_t = \infty \qquad \sum_{t=1}^{\infty} \alpha_t^2 = \infty$
- 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

- Learning rate is one of the hyperparameters that is the most difficult to set; it has a significant impact on the model performance
- Cost 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
  - RMSProp with Nesterov momentum

AdaGrad

- Adapts the learning rate of each parameter based on sizes of previous updates:
  - scales updates to be larger for parameters that are updated less
  - scales updates to be smaller for parameters that are updated more
- The net effect is greater progress in the more gently sloped directions of parameter space
  - Desirable theoretical properties but empirically (for deep models) **can result in a premature and excessive decrease in effective learning rate**

- **Require:** learning rate α, initial parameter θ, **small constant δ** (*e.g. 10^{-7}) for numerical stability*
- **Update:**
  - compute gradient $g \leftarrow +\frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$
  - accumulate squared gradient $r \leftarrow r + g \odot g$

    sum of all previous squared gradients
  - compute update $\Delta\theta \leftarrow -\frac{\alpha}{\delta + \sqrt{r}} \odot g$

    updates inversely proportional to the square root of the sum
  - apply update $\theta \leftarrow \theta + \Delta\theta$

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

Root Mean Square Propagation (RMSProp)

- Modifies AdaGrad to perform better in non-convex surfaces, for aggressively decaying learning rates
- Changes gradient accumulation by an **exponentially decaying average** of sum of squares of gradients

- **Requires:** learning rate α, initial parameter θ, **decay rate ρ**, small constant δ *(e.g. 10^{-7}) for numerical stability*
- **Update:**
  - compute gradient $g \leftarrow +\frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$
  - 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 $\theta \leftarrow \theta + \Delta\theta$

It can be combined with Nesterov momentum

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 (firs and second moments) to account for their initialization at origin

Update:

- compute gradient $g \leftarrow +\frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$
- **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 $\theta \leftarrow \theta + \Delta\theta$

δ=e-8, ρ1=0.9, ρ2=0.999

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

Example: test function

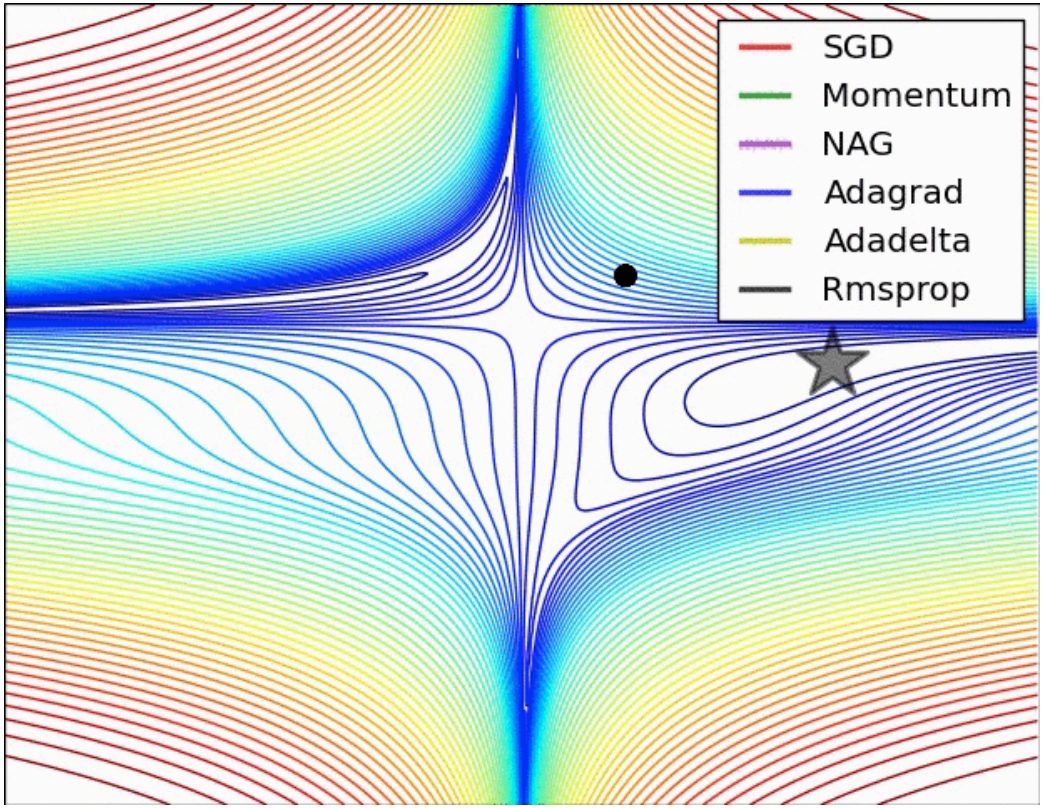
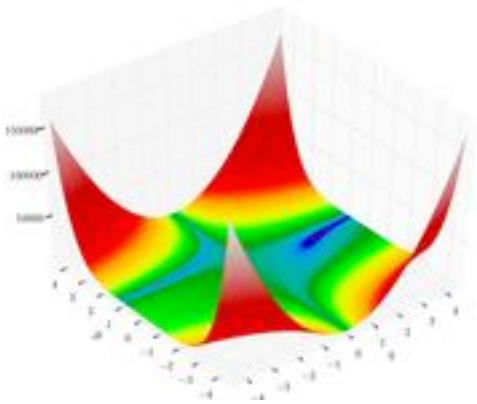


Image credit: Alec Radford.



Beale's function

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

Example: saddle point

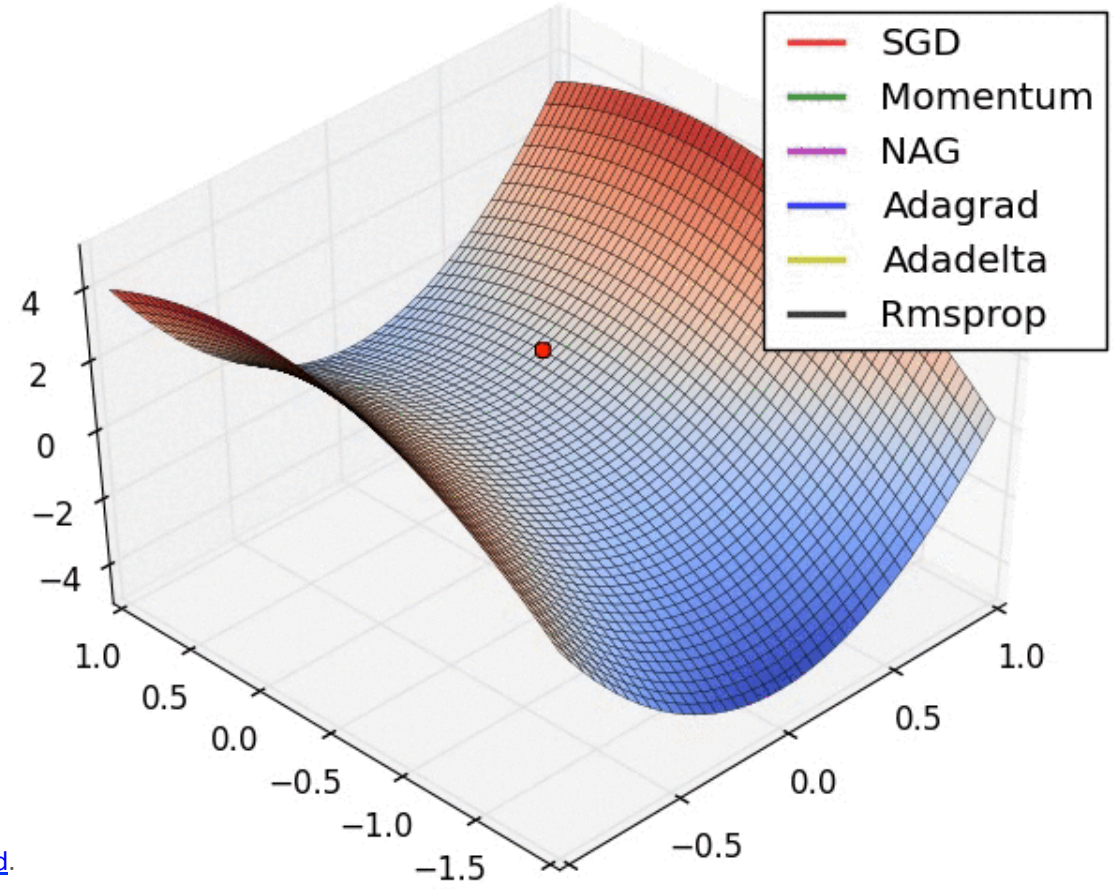
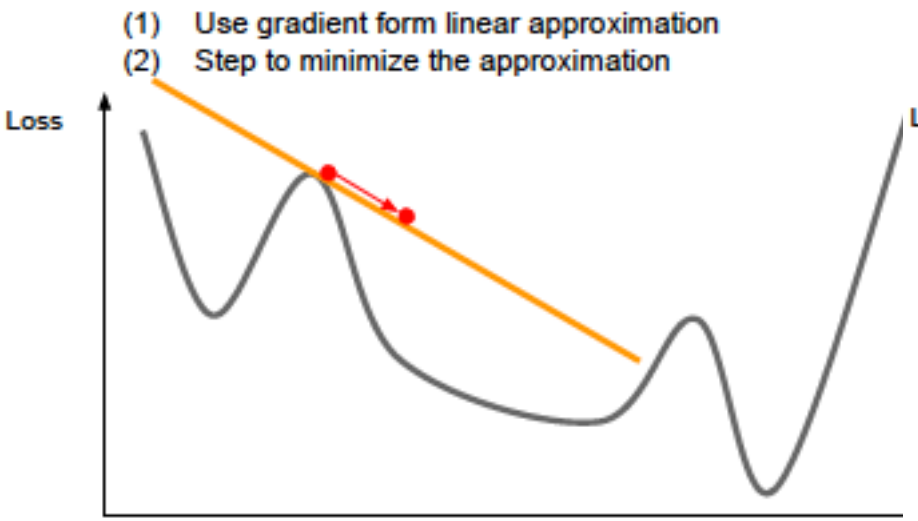


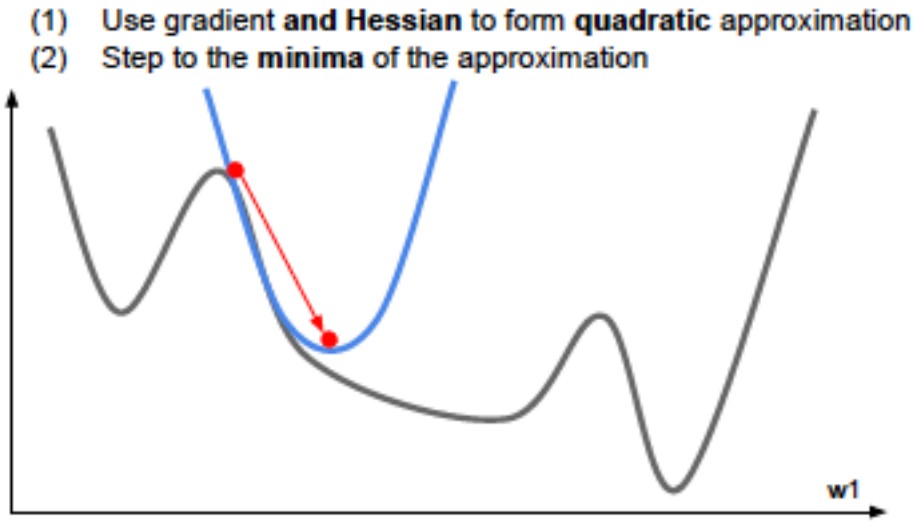
Image credit: Alec Radford.

Second order optimization

First order



Second order



Second order optimization

- Second order Taylor expansion

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H(\theta - \theta_0)$$

- Solving for the critical point we obtain the Newton parameter update:

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0)$$

- **Problem:** Hessian has O(N^2) elements, inverting H O(N^3) (N parameters ≈millions)

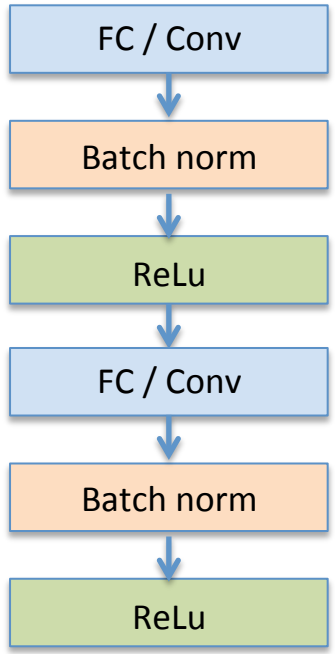
- **Alternatives:**
  - Quasi-Newton methods (**BGFS** Broyden-Fletcher-Goldfarb-Shanno): instead of inverting Hessian, approximate inverse Hessianwith rank 1 updates over time O(N2) each
  - **L-BFGS** (Limited memory BFGS): does not form/store the full inverse Hessian

Parameter 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** $N(0,10^{-2})$
    - if weights start too small, the signal shrinks as it passes through each layer until it is too tiny to be useful
  - **Calibrating variances with 1/sqrt(n) (Xavier Initialization)**
    - 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

Batch normalization

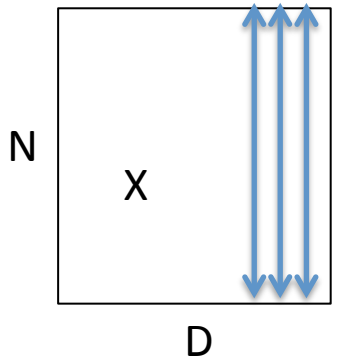
- As learning progresses, the distribution of the layer inputs changes due to parameter updates (internal covariate shift)
- This can result in most inputs being in the non-linear regime of the activation function, slowing down learning
- Bach normalization is a technique to reduce this effect
  - Explicitly force the layer activations to have zero mean and unit variance w.r.t running batch estimates
  - Adds a learnable scale and bias term to allow the network to still use the nonlinearity



Ioffe and Szegedy, 2015. "Batch normalization: accelerating deep network training by reducing internal covariate shift"

Batch normalization

- Can be applied to any input or hidden layer
- For a mini-batch of N activations of the layer
  1. compute empirical mean and variance for each dimension D
  2. normalize



$$\hat{x}^{(k)} = \frac{x^{(k)} - E(x^{(k)})}{\sqrt{\text{var}(x^{(k)})}}$$

- Note: normalization can reduce the expressive power of the network (e.g. normalize intputs of a sigmoid would constrain them to the linear regime
  - So let the network learn the identity: scale and shift $y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$
  - To recover the identity mapping the newtork can lean

$$\gamma^{(k)} = \sqrt{\text{var}(x^{(k)})} \qquad \beta^{(k)} = E(x^{(k)})$$

Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

1. Improves gradient flow through the network
2. Allows higher learning rates
3. Reduces the strong dependency on initialization
4. Reduces the need of regularization

Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

At test time BN layers functions differently:

1. Mean and std are not computed on the batch.

2. Instead, a single fixed empirical mean and std of activations computed during training is used

   (can be estimated with running averages)

Summary

- Optimization for NN is different from pure optimization:
  - GD with mini-batches
  - early stopping
  - non-convex surface, local minima and saddle points
- Learning rate has a significant impact on model performance
- Several extensions to SGD can improve convergence
- Adative learning-rate methods are likely to achieve best results
  - RMSProp, Adam
- Weight initialization: He w= randn(n) sqrt(2/n)
- Batch normalization to reduce the internal covariance shift

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