Réseaux de Neurones Profonds, Apprentissage de Représentations

Thierry Artières

ECM, Equipe QARMA @LIS, AMU, CNRS

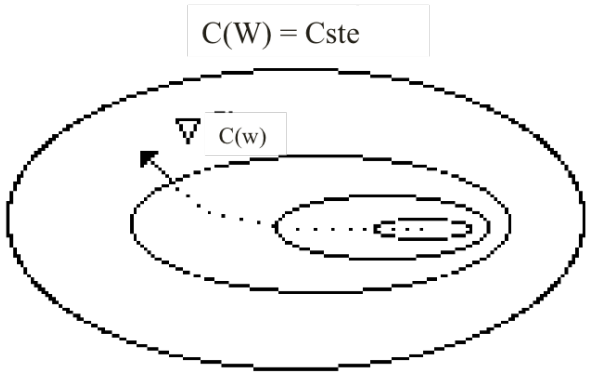
October 8, 2018

Outline

Gradient Descent Optimization

Gradient Descent Optimization

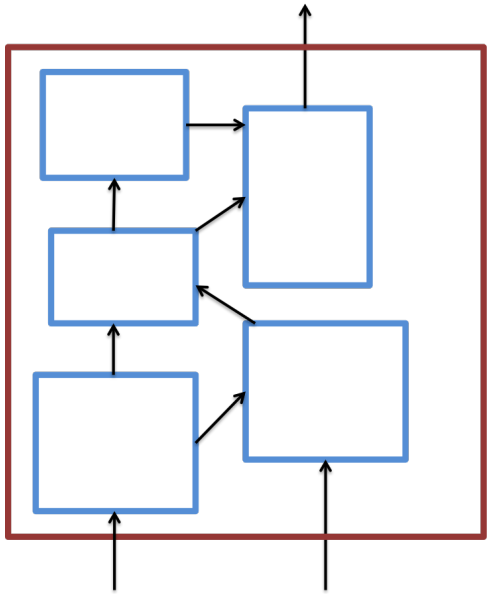
- Initialize Weights (Randomly)
- Iterate (till convergence)
  - Restimate $\mathbf{w}_{t+1} = \mathbf{w}_t - \epsilon \frac{\partial C(\mathbf{w})}{\partial \mathbf{w}}|_{\mathbf{w}_t}$
  - Note that $\frac{\partial C(\mathbf{w})}{\partial \mathbf{w}}$ is per default noted as $\nabla C(\mathbf{w})$ hereafter



$\Rightarrow$ Few illustrations in these slides are taken from [LeCun et al, 1993], [Fei Fei Li lecture 6], and from *S. Ruder's blog*

GD for general architectures

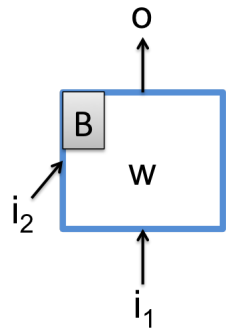
Graph of modules (better without cycles...)



Still optimized with Gradient Descent !!

$$W = W - \epsilon \frac{\partial C(W)}{\partial W}$$

- provided functions implemented by blocks are differentiable
- and derivatives $\frac{\partial Out(B)}{\partial In(B)}$ and $\frac{\partial Out(B)}{\partial W(B)}$ are available for every block



Gradient Descent: Tuning the Learning rate

Two classes Classification problem

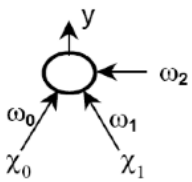
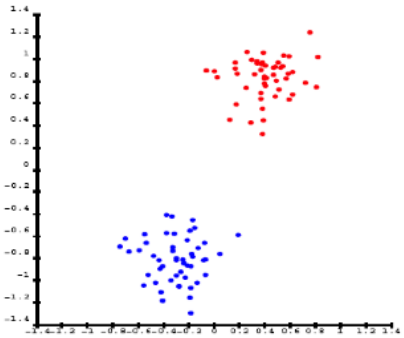
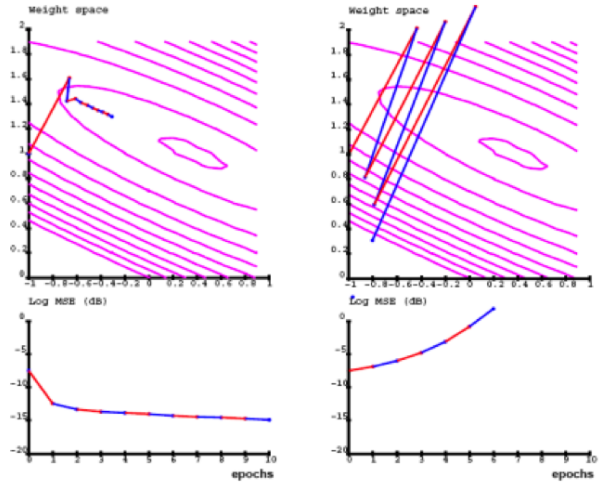


Fig. 9. Simple linear network.



Weight trajectory for two different gradient step settings.



(a) (b)

Fig. 11. Weight trajectory and error curve during learning for (a) $\eta = 1.5$ and (b) $\eta = 2.5$.

Images from [LeCun et al.]

Gradient Descent: Tuning the Learning rate

Effect of learning rate setting

- Assuming the gradient direction is good, there is an optima value for the learning rate
- Using a smaller value slows the convergence and may prevent from converging
- Using a bigger value makes convergence chaotic and may cause divergence

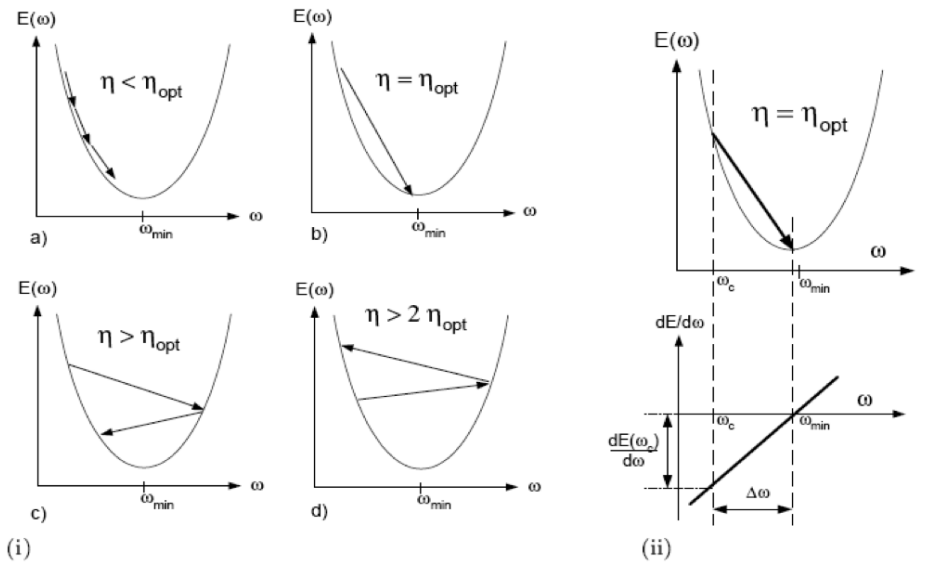


Fig. 6. Gradient descent for different learning rates.

Images from [LeCun et al.]

Optimal learning rate and convergence speed

Second order point of view

- Taylor expansion, noting $\nabla^2 C(w)$ the Hessian (a $N \times N$ matrix with N a model with parameters)

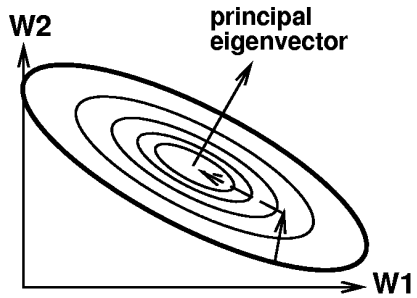
$$C(w') = C(w) + (w' - w)^T \nabla C(w) + \frac{1}{2}(w' - w)^T \nabla^2 C(w)(w' - w)$$

$$\nabla C(w)|_{w'} = \nabla C(w)|_w + \nabla^2 C(w)(w' - w)$$

- Optimum rule (setting $\nabla C(w)|_{w'}$ to 0):

$$w' = w - (\nabla^2 C(w))^{-1} \nabla C(w)$$

- Optimal move not in the direction of the gradient
- Said differntly: Not a identical step in every direction !
- In Order 1 Gradient descent the optimal the optimal value of ϵ depends on eigen values of the Hessian $\nabla^2 C(w)$
- The optimal value depends on the highest eigen value ($\hat{\epsilon} = \frac{1}{\lambda_{max}}$) of the Hessian



From [Lecun et al, 93]

Gradient Descent: Stochastic, Batch and mini batchs

Objective : Minimize $C(\mathbf{w}) = \sum_{i=1..N} L_w(i)$ with $L_w(i) = L_w(x^i, y^i, w)$

Batch vs Stochastic vs Minibatchs

- Batch gradient descent
  - Use $\nabla C(\mathbf{w})$
  - Every iteration all samples are used to compute the gradient direction and amplitude
- Stochastic gradient
  - Use $\nabla L_w(i)$
  - Every iteration one sample (randomly chosen) is used to compute the gradient direction and amplitude
  - Introduce randomization in the process.
  - Minimize $C(w)$ by minimizing parts of it sucessively
  - Allows faster convergence, avoiding local minima etc
- Minibatch
  - Use $\nabla \sum_{\text{few } j} L_w(j)$
  - Every iteration a batch of samples (randomly chosen) is used to compute the gradient direction and amplitude
  - Introduce randomization in the process.
  - Optimize the GPU computation ability

Outline

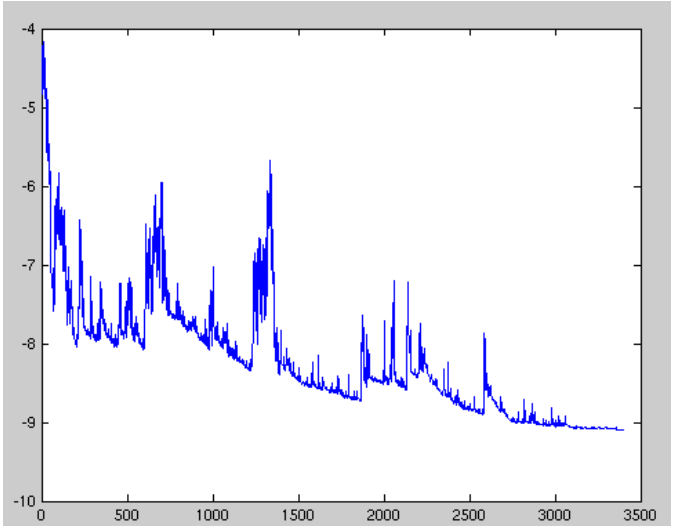
Using Momentum

SGD with Momentum

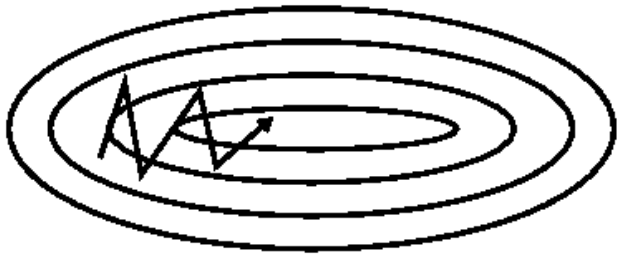
- Standard Stochastic Gradient descent : $w = w - \epsilon \frac{\partial C(w)}{\partial w}$
- SGD with Momentum:

$$v = \gamma v + \epsilon \frac{\partial C(w)}{\partial w}$$

$$w = w - v$$



SGD standard



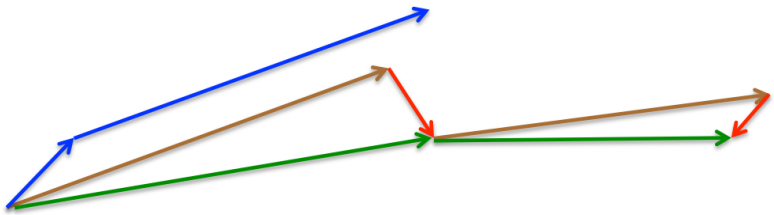
SGD avec momentum

Nesterov Accelerated Gradient

Principle

- Idea: Better anticipate when to slow down by looking forward

$$v_{t+1} = \gamma v_t + \epsilon \nabla C(w)|_{w_t - \gamma v_t}$$
$$w_{t+1} = w_t - v_{t+1}$$



- Blue vectors: standard momentum
- Brown vectors: jump
- Red vectors: correction
- Green vectors: accumulated gradient

Adagrad

Reminder: Optimally one needs to adapt the learning rate to every weight

- Define $g_{t,i} = \frac{\partial C(w)}{\partial w_i}$ the derivative wrt a single weight value w_i
- $w_{t+1,i} = w_{t,i} - \frac{\epsilon}{\sqrt{G_{t,ii}+\gamma}} g_{t,i}$
  - where $G_{t,ii}$ is a diagonal matrix with i^{th} element equal to $\sum_t g_{t,i}^2$
  - γ is a very small value to avoid numerical exceptions
  - Standard value $\epsilon = 0.01$
- Variants that aim at minimizing the aggressive feature of Adagrad: Adadelta , Adam, and RmsProp

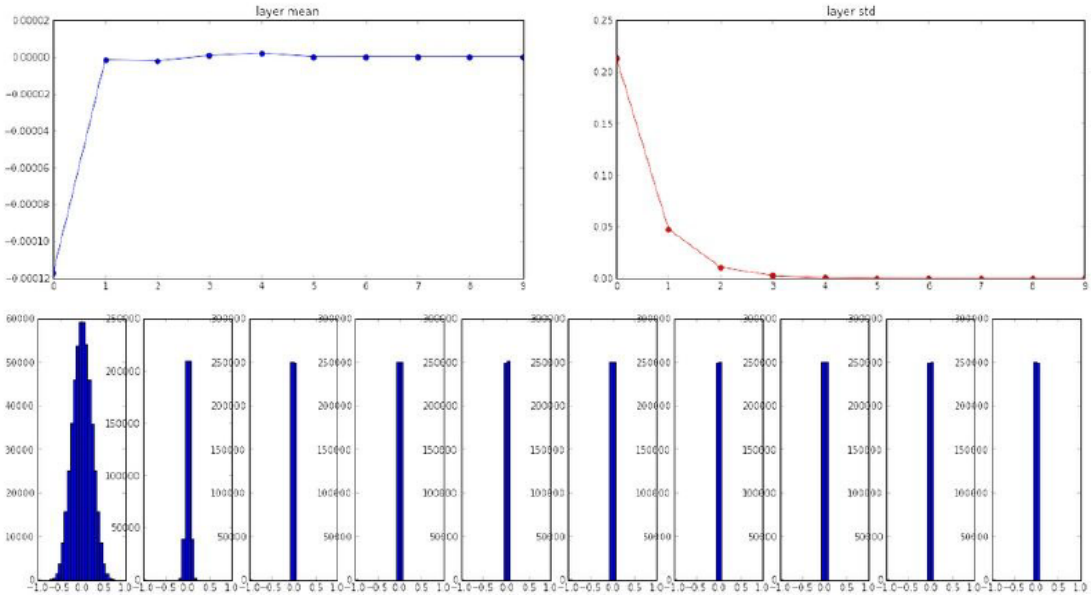
Outline

Activity propagation in deep NNs

Few slides from *Fei Fei Li*

Standard initialization schema for MLPs

- 10 layers networks (500 neurones each, with tanh)
- Initialization : gaussian random with small (std=0.01) values (what if all null initialization?)
- All activations at 0
- What about the gradient ?

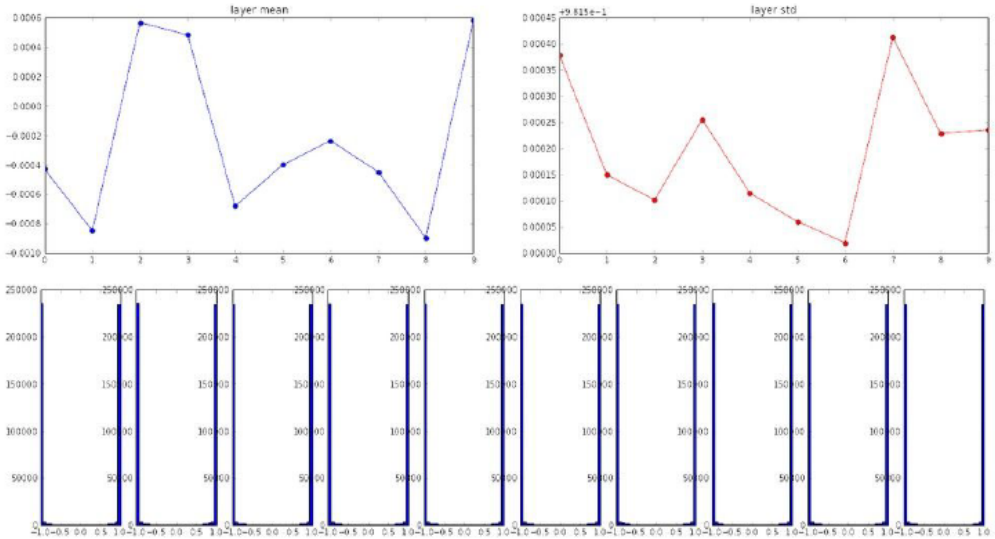


From Fei Fei Li's slides

Tuning weights initialization

Increasing weights initial values comes with neuron saturation problem

- 10 layers networks (500 neurones each, with tanh)
- Initialization : gaussian random with normal (std=1.0) values
- All neurons saturate
- No gradient backpropagated

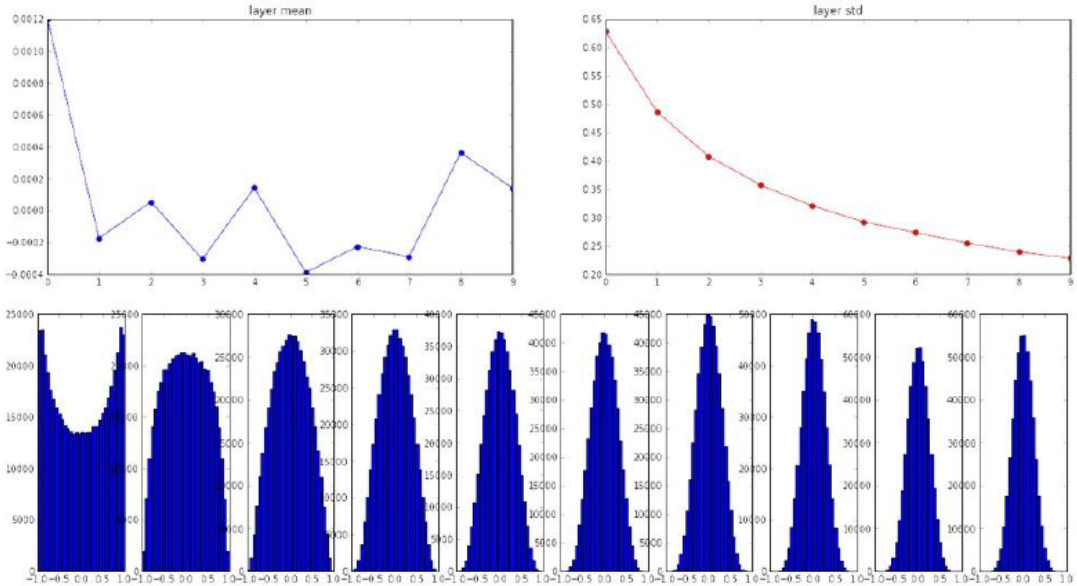


From Fei Fei Li's slides

Smarter intialization

Good (but not enough)

- 10 layers networks (500 neurones each, with tanh)
- Xavier initialization : random gaussian with std dev $= \frac{1}{N_{previouslayer}}$
- Much better behavior but fails with RELU activation (assuming normalized inout data)

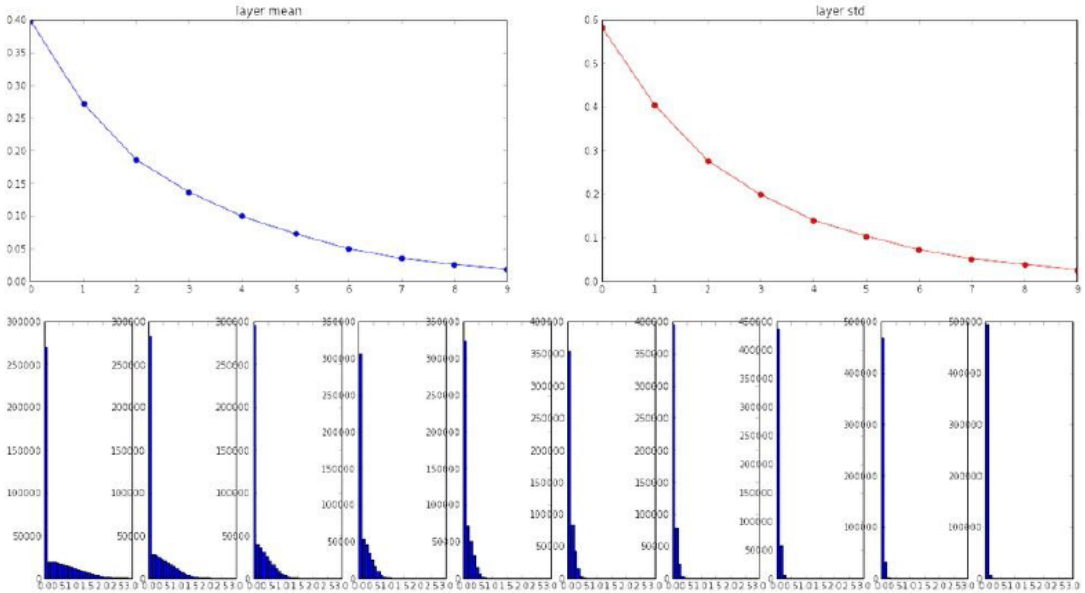


From Fei Fei Li's slides

Smarter intialization

Good (but not enough)

- 10 layers networks (500 neurones each, with tanh)
- Xavier initialization : random gaussian with std dev $= \frac{1}{N_{previouslayer}}$
- Much better behavior but fails with RELU activation (assuming normalized inout data)

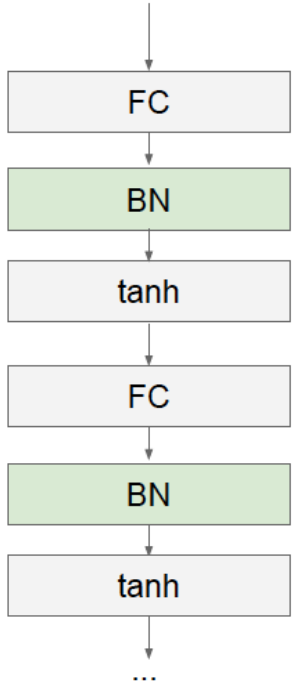


From Fei Fei Li's slides

Batch Normalization

Main idea

- Usually inputs to neural networks are normalized to either the range of [0, 1] or [-1, 1] or to mean=0 and variance=1
- BN essentially performs Whitening to the intermediate layers of the networks.
- Usually placed before nonlinearities



From Fei Fei Li's slides

Batch Normalization

BN layer

- Normalizes the output of a layer by scaling neuron's outputs within a minibatch (of size M)
- For one neuron of the input layer, its output is modified according to:

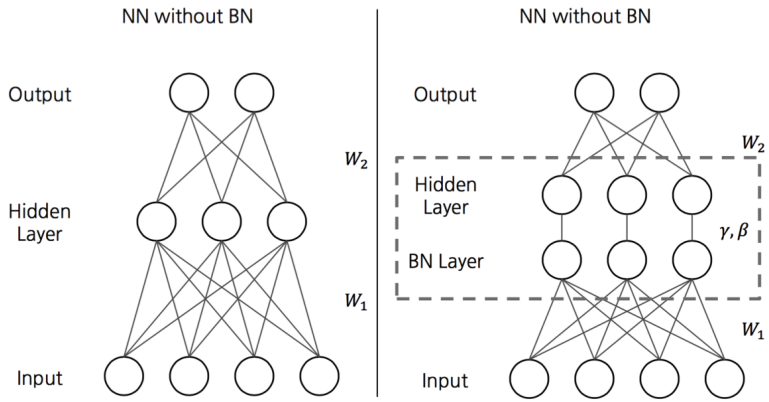
$$\mu_B = \frac{1}{M}\sum_{i=1}^{M} x_i \tag{1}$$

$$\sigma_B^2 = \frac{1}{M}\sum_{i=1}^{M}(x_i - \mu_B)^2 \tag{2}$$

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \tau}} \tag{3}$$

$$y_i = \gamma x_i + \beta \tag{4}$$

- Use a different computation at inference time (empirical mean and variance computed on the full training set)



From Fei Fei Li's slides

Outline

Guiding the learning through reguarization

Regularization

- Constraints on weights (L1 or L2)
- Constraints on activities (of neurons in a hidden layer)
  - L1 or L2
    - Push useless weights to 0
  - Mean activity constraint (Sparse autoencoders, [Ng et al.])
  - Sparsity constraint (in a layer and/or in a batch)
  - Winner take all like strategies
- Disturb learning for avoiding learning by heart the training set
  - Noisy inputs (e.g. Denoising Autoencoder, link to L2 regularization)
  - Noisy labels

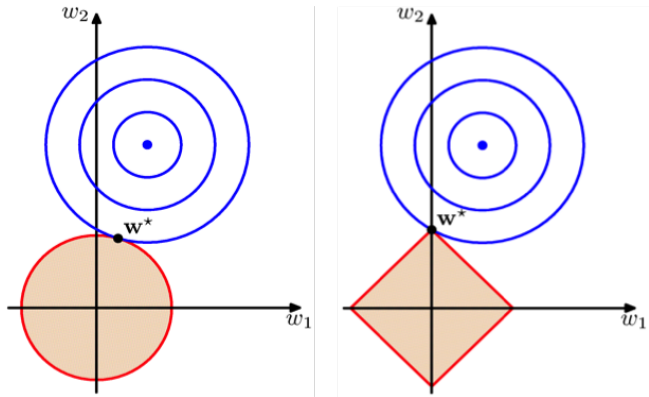
Constraints on weights

L2 norm on weights (known as Weight Decay)

- Penalizing the weights through adding a weighted L2 norm $\lambda \|w\|^2$ to the loss
- It is equivalent to defining a family of models such that $\|w\|^2 leq C_\lambda$ with C_λ increasing when λ decreases
- L2 norm penalization $\leftrightarrow$ diminishing the space of functions implemented with the network architecture

L2 and L1 norms

- L2 norm move useless weights to 0 (without reaching 0)
- L1 norm set useless weights to 0



Guiding the learning

Improved representations with denoising autoencoders and Deep Belief Networks

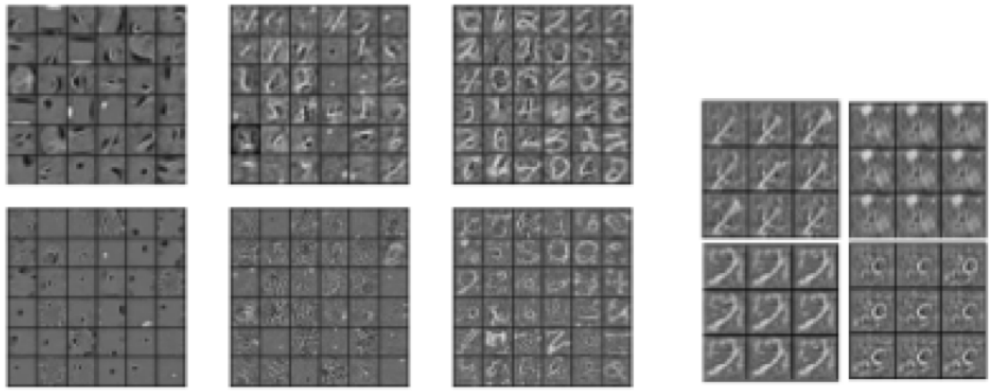
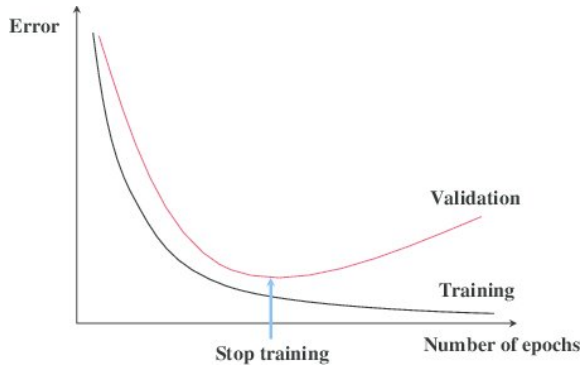


Figure 1: Activation maximization applied on MNIST. On the left side: visualization of 36 units from the first (1st column), second (2nd column) and third (3rd column) hidden layers of a DBN (top) and SDAE (bottom), using the technique of maximizing the activation of the hidden unit. On the right side: 4 examples of the solutions to the optimization problem for units in the 3rd layer of the SDAE, from 9 random initializations.

- Examples of learned filters with Denoising Autoencoders (top)

Early stopping and callbacks

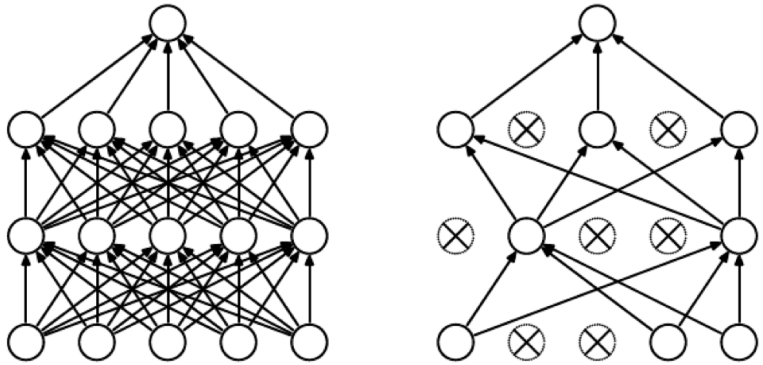
Principle



- Early stopping monitors performance (loss) on validation set
- Stopes before it reaches a plateau and starts increasing
- Related to the idea that the implemented model's capacity increases with the number of iteration
  - Think of small weights initialization and sigmoid activation
  - $\Rightarrow$ at the beginning the model is a linear one !

Dropout

Principle

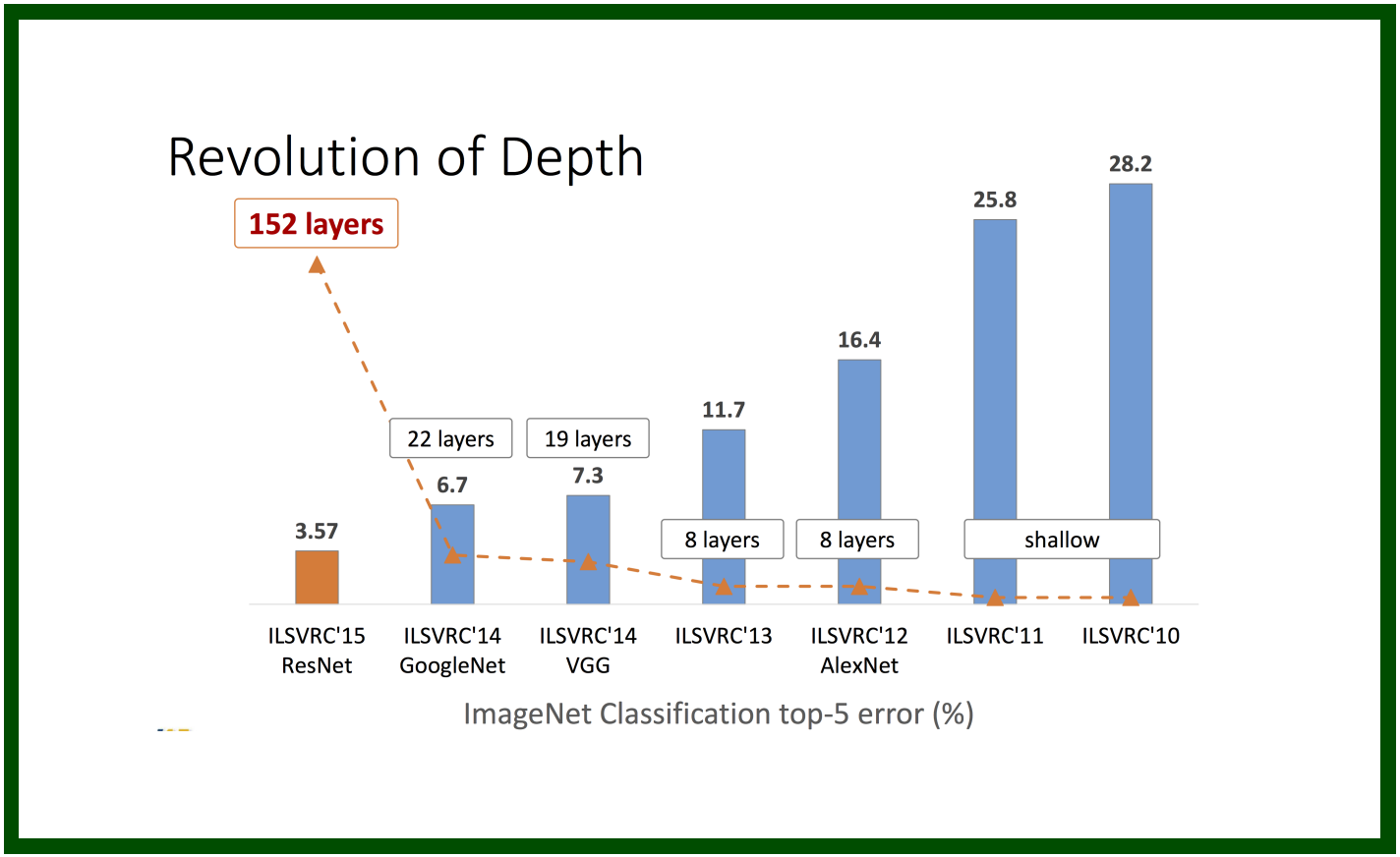


(a) Standard Neural Net (b) After applying dropout.

- First method that allowed learning rellay deep networks without pretraining and smart initialization
- Related to ensemble of models
- Weights are normalized at inference time

Outline

The Times They Are A Changing

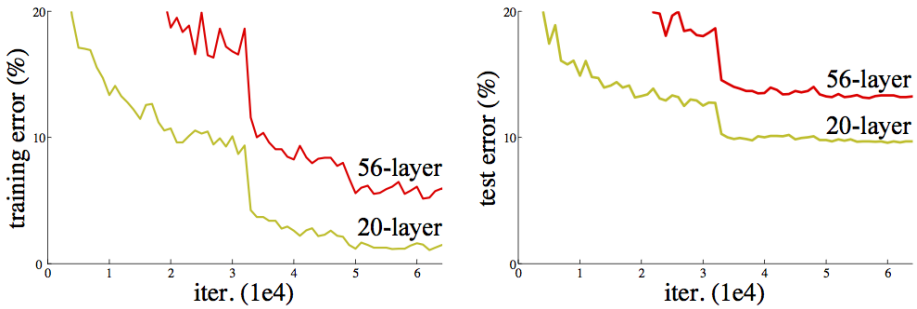


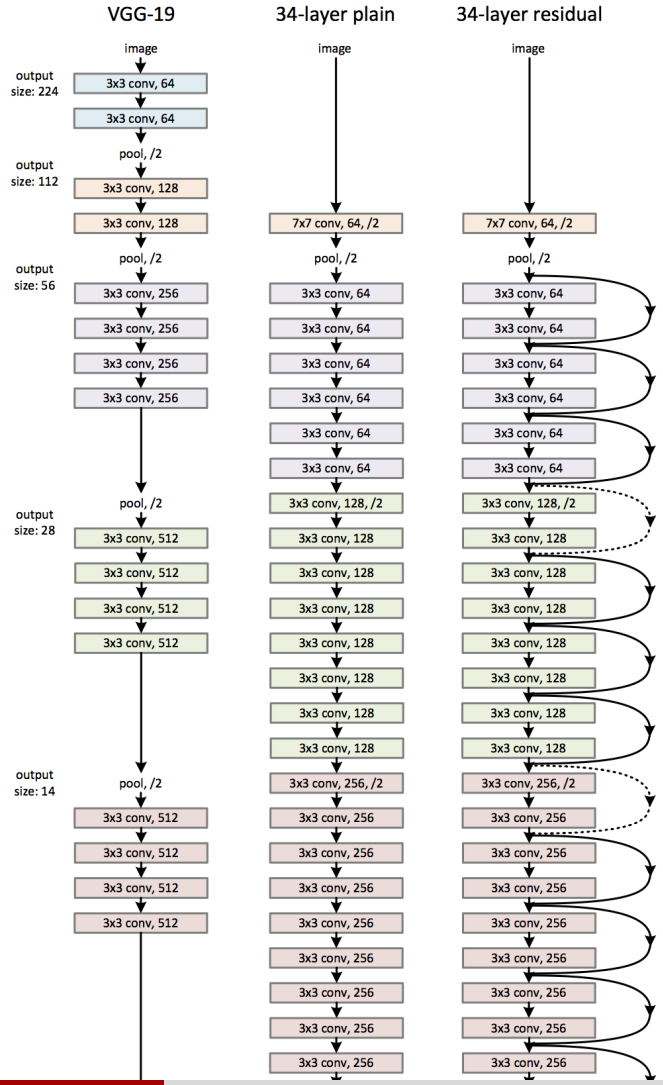
(slide from [Kaiming He])

From shalow to deep

Simply stacking layers does not work (CIFAR results) ! (figures form [He and al., 2015])

...



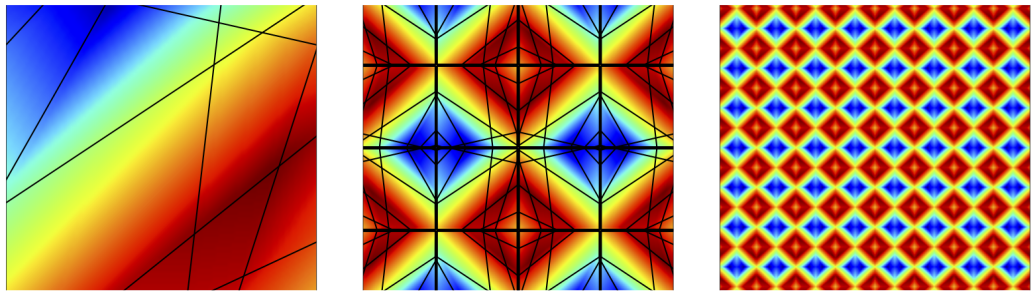


Deep vs Shalow ?

Characterizing the complexity of functions a DNN may implement [Pascanu and al., 2014]

- DNNs with RELU activation function $\Rightarrow$ piecewise linear function
- Complexity of DNN function as the Number of linear regions on the input data
- Case of n_0 inputs and $n = 2n_0$ hidden cells per HL (k HL) :
  - Maximum number of regions : $2^{(k-1)n_0} \sum_{j=0}^{n_0} \binom{2n_0}{j}$
- Example: $n_0 = 2$
  - Shallow model: $4n_0$ units $\rightarrow$ 37 regions
  - Deep model with 2 hidden layers with $2n_0$ units each $\rightarrow$ 44 regions
  - Shallow model: $6n_0$ units $\rightarrow$ 79 regions
  - Deep model with 3 hidden layers with $2n_0$ units each $\rightarrow$ 176 regions
- Exponentially more regions per parameter in terms of number of HL
- At least order (k-2) polynomially more regions per parameter in terms of width of HL n

Deep vs Shalow ?



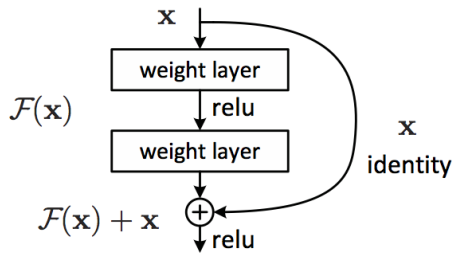
From [Pascanu and al., 2014]

- Left: Regions computed by a layer with 8 RELU hidden neurons on the input space of two dimensions (i.e. the output of previous layer)
- Middle: Heat map of a function computed by a rectifier network with 2 inputs, 2 hidden layers of width 4, and one linear output unit. Black lines delimit regions of linearity of the function
- Right: Heat map of a function computed by a 4 layer model with a total of 24 hidden units. It takes at least 137 hidden units on a shallow model to represent the same function.

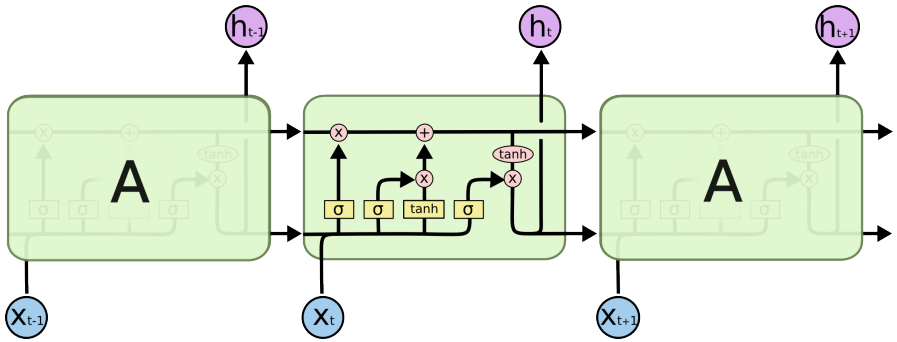
The depth alone is not enough

Making gradient flow for learning deep models

- Main mechanism : Include the identity mapping as a possible path from the input to the output of a layers
- ResNet building block [He and al., 2015]]



- LSTM (deep in time) [Hochreichter and al., 1998]



About generalization, overtraining, local minimas etc

Traditional Machine Learning

- Overfiting is the enemy
- One may control generalization with appropriate regularization

Recent results in DL

- The Overfit idea should be revised for DL [Zhand and al., 2017]
  - Deep NN may learn noise !
  - Regularization may slightly improve performance but is not THE answer for improving generalization
- Objective function do not exhibit lots of saddle points and most local minima are good and close to globale minimas [Choromanska et al., 2015]
  - Not clear what in the DNN may allow to predict its generalization ability