CMC MSU Department of Algorithmic Languages
Samsung Moscow Research Center

Neural Networks for Natural Language Processing

Нейронные сети в задачах автоматической обработки текстов

Lecture 4. Optimization Basics

Arefyev Nikolay
CMC MSU Department of Algorithmic Languages &
Samsung Moscow Research Center

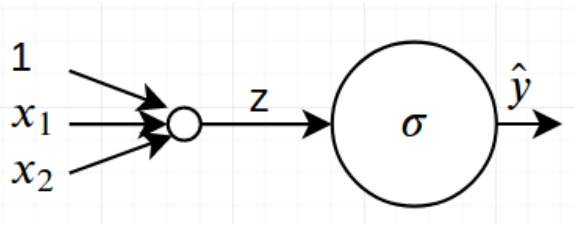
A machine learning algorithm usually corresponds to a combination of the following 3 elements:

(either explicitly specified or implicit)

- ✓ the choice of a specific function family: F
  (often a parameterized family)
- ✓ a way to evaluate the quality of a function $f \in F$
  (typically using a cost (or loss) function L
  mesuring how wrongly f prédicts)
- ✓ a way to search for the «best» function $f \in F$
  (typically an optimization of function parameters to minimize the overall loss over the training set).

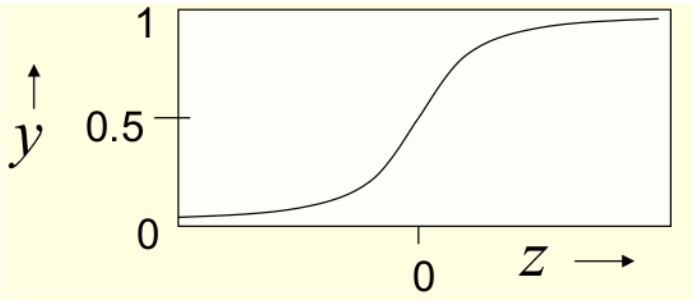
Reminder: LR hypothesis

- Logistic regression hypothesis:



$$(preactivation)\ z = b + w^T x = b + \sum_{j=1}^{m} w_j * x_j$$

$$\hat{y} = h_w(x) = \sigma(z) = \frac{1}{1+e^{-z}}$$



- Implementation tricks:
  - Treat bias as w_0:

$$z = [1; x]^T w \ (concatenate\ x_0 = 1, then\ w_0 = b)$$

  - Vectorize: compute for all examples in parallel:

$$(preactivation)\ Z = Xw \leftarrow dimensions\ ?$$

$$\hat{Y} = \sigma(Z) \leftarrow componentwise$$

Figure from Hinton. Neural Networks for Machine Learning course, 2012 @ coursera.org

FFNN hypothesis

- Feed-forward neural network = composition of logistic regressions

$$a^{(0)} = x^T \quad \leftarrow inputs = 0\,th\ layer\ activations\ (1 \times s^{(0)})$$

$$z^{(l)} = [1; a^{(l-1)}] W^{(l)} \quad \leftarrow lth\ layer\ preactivations\ (1 \times s^{(l)})$$

$$a^{(l)} = g^{(l)}(z^{(l)}) \quad \leftarrow lth\ layer\ activations\ (outputs)\ (1 \times s^{(l)})$$

$$\hat{y} = h_W(x) = a^{(L)} \quad \leftarrow outputs = Lth\ layer\ activations\ (1 \times s^{(L)})$$

- Vectorized version?

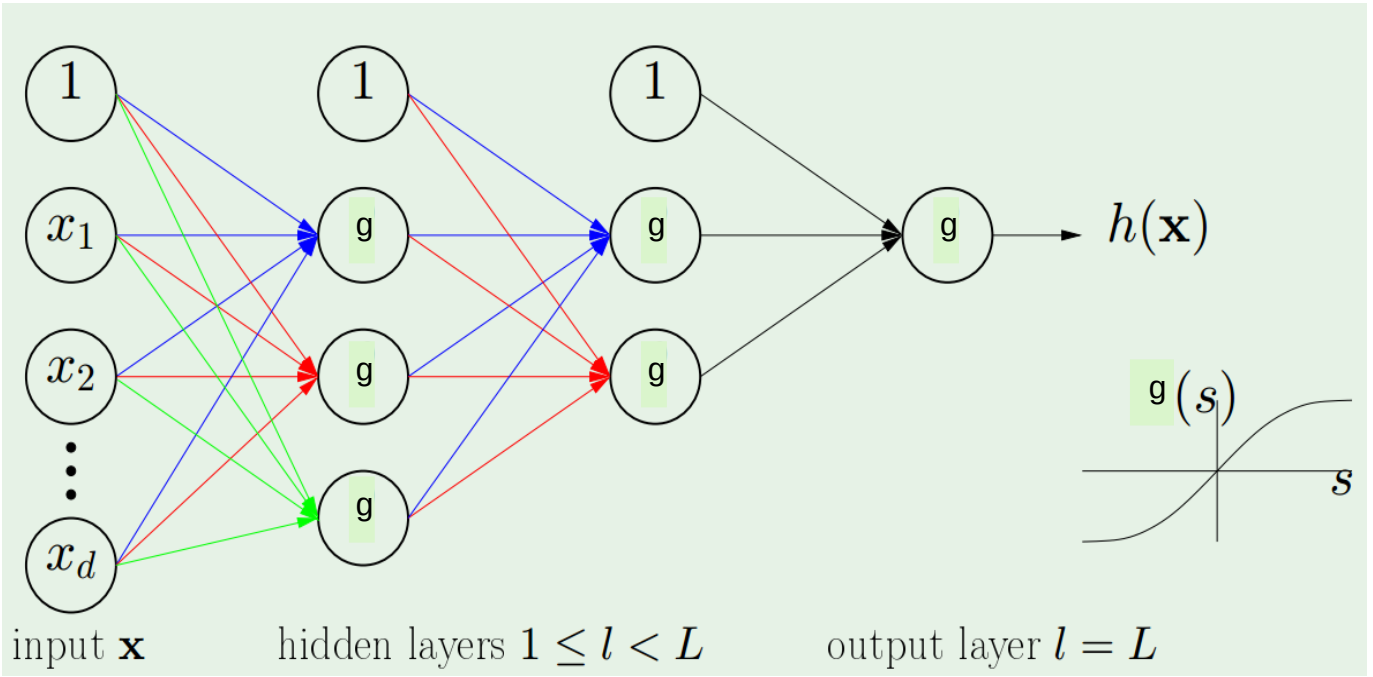


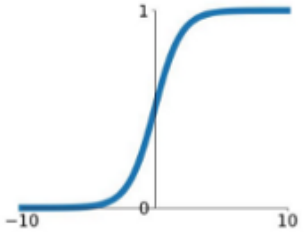
Image from Yaser Abu-Mostafa. Machine learning video library (https://work.caltech.edu/library/)

Popular activation functions

- Don’t use Sigmoid FFNN (not centered in 0)!
  - Except the last layer (before binary CE loss)
- Tanh is good default

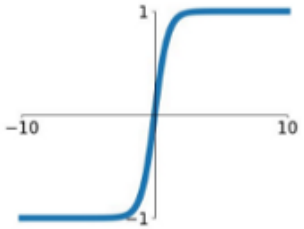
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



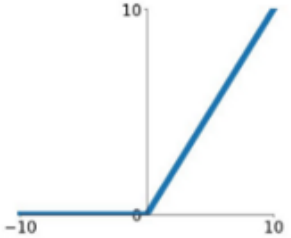
tanh

$\tanh(x)$



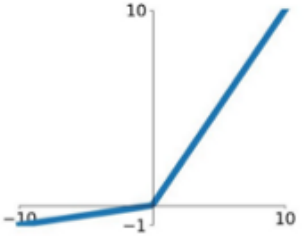
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

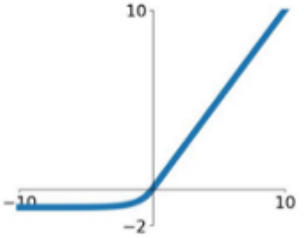


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

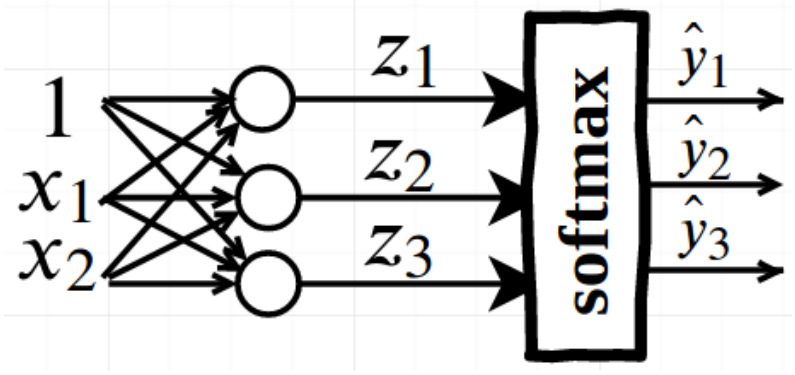
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Shruti Jadon. Introduction to Different Activation Functions for Deep Learning @ medium.com

Multiclass/multilabel classification

- For multiclass (exactly one class per example)
  - One vs. rest (K independent binary classifiers)
  - **Softmax classifier** = multinomial logistic regression =
    = Linear layer + **Softmax function**



$$\vec{z} = [1; x]^T W$$

$$[softmax(\vec{z})]_k = \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}}$$

  - NN + softmax classifier
- For multilabel (several labels per example)
  - Bunch of logistic regressions (one per label)
    - Often used as last layer of NN

Reminder: CE Loss

- Binary cross-entropy loss ← for classification

$$E(w) = -\frac{1}{N}\sum_{i=1}^{N} y_{\{i\}} \log\left(h_w\left(x_{\{i\}}\right)\right) + \left(1 - y_{\{i\}}\right) \log\left(\left(1 - h_w\left(x_{\{i\}}\right)\right)\right)$$

  - for LR – convex w.r.t. weights (unlike MSE)
  - Justified by Maximum Likelihood
  - DERIVE from CE definition, PLOT w.r.t. h(x)
- Cross-entropy loss for multiclass
  - DERIVE from CE definition given y is one-hot

$$E(w) = -\frac{1}{N}\sum_{i=1}^{N} \log\left(\left[h_w\left(x_{\{i\}}\right)\right]_t\right), t = argmax\ y_{\{i\}}\left(true\ class\right)$$

- Vectorized version?

LR/FFNN Optimization

- Minimizing E(w) – tuning knobs
  - no closed-form solution (unlike linear regression)
  - trial and error method dosn't work: too many knobs, their relative positions matter
  - Calculus to the rescue!

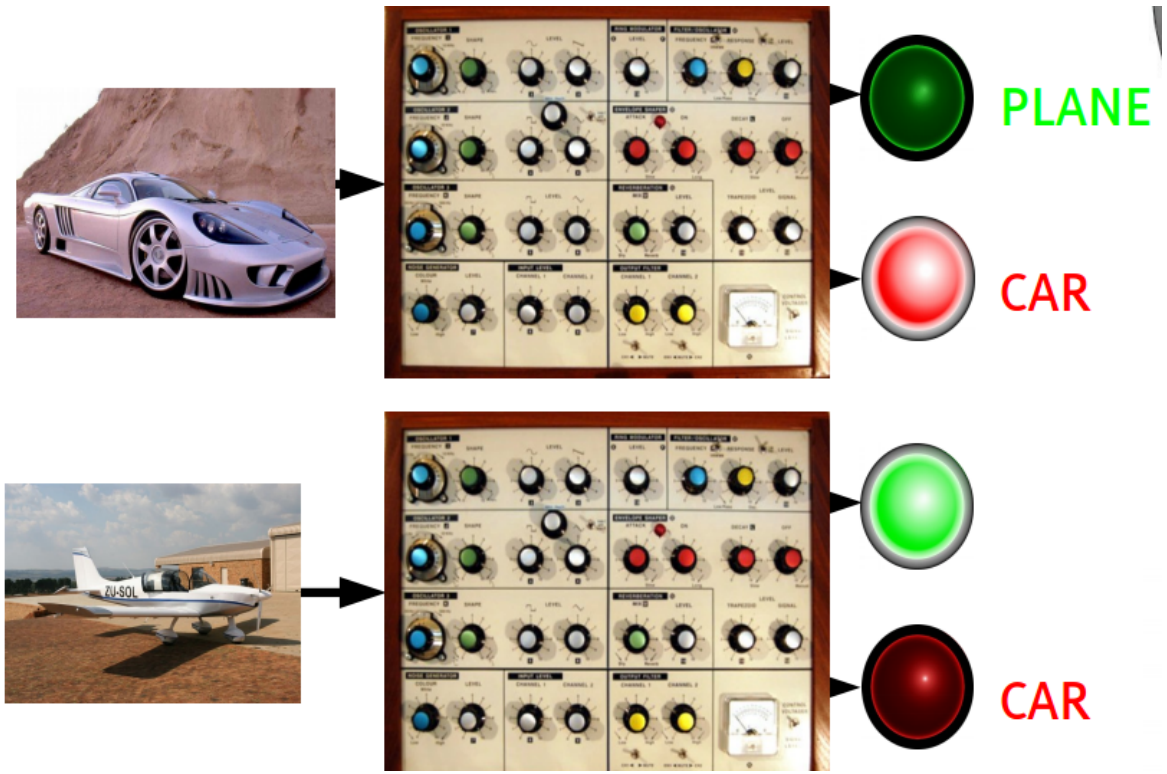


Image from Yann Le Cun, Deep Learning and the Future of AI, 2016

Linear approximation

Approximate exp(x) near x=0:

$$f(x) = e^x \approx p(x) = c_0 + c_1 x$$
$$f(0) = 1,\ p(0) = c_0 \Rightarrow c_0 = 1$$
$$f'(0) = 1,\ p'(0) = c_1 \Rightarrow c_1 = 1$$

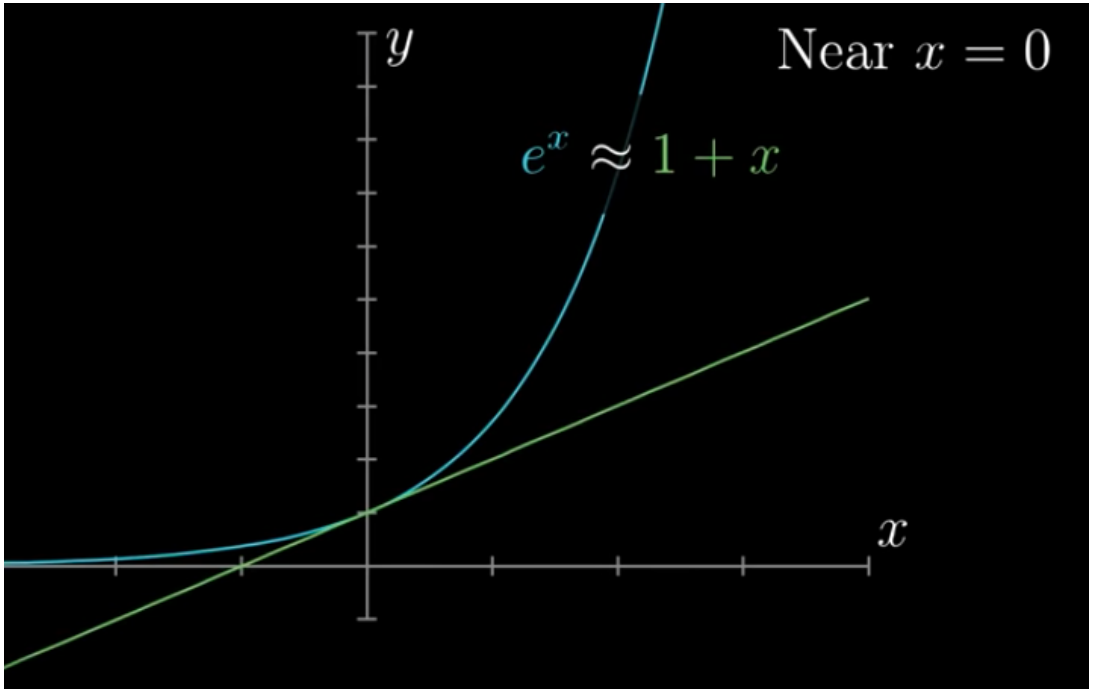


Image from Taylor series | Essence of calculus, chapter 11, www.youtube.com/watch?v=3d6DsjIBzJ4

Quadratic approximation

Approximate exp(x) near x=0

$$f(x)=e^x\approx 1+x+c_2x^2$$

$$f''(0)=1, p''(0)=2c_2\Rightarrow c_2=\frac{1}{2}$$

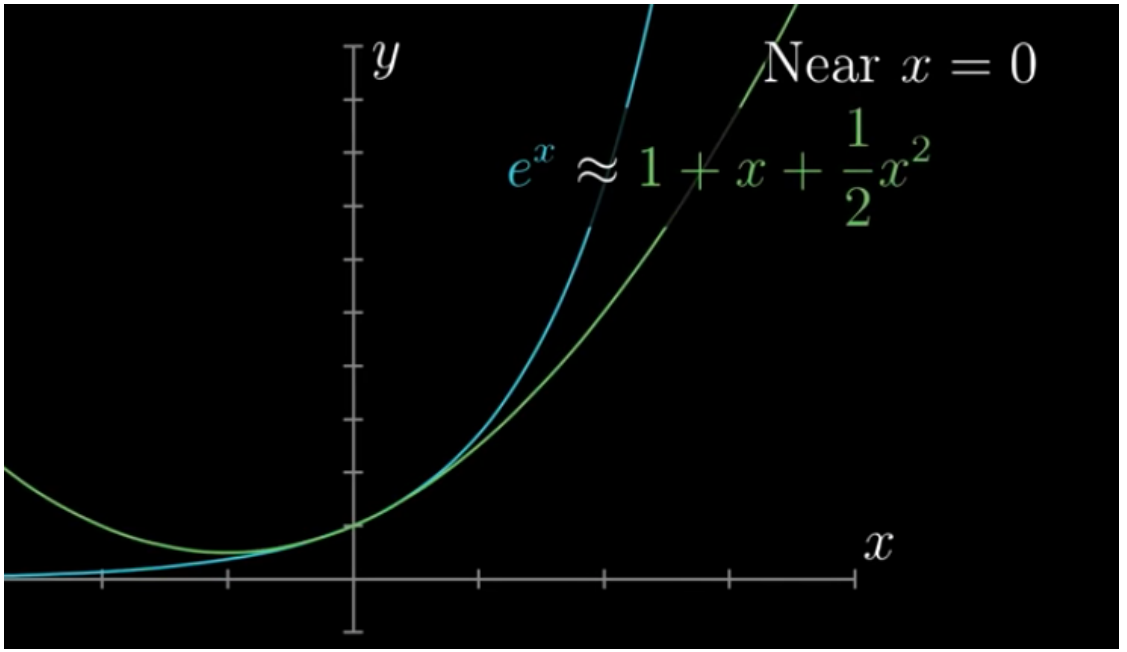
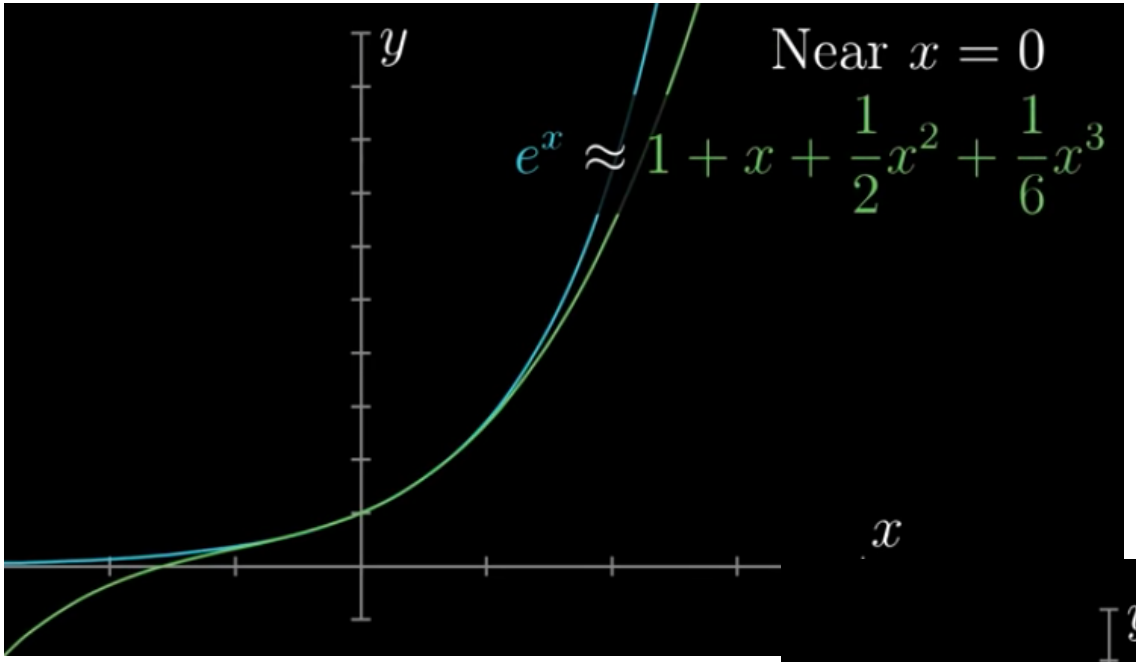


Image from Taylor series | Essence of calculus, chapter 11, www.youtube.com/watch?v=3d6DsjIBzJ4

Higher order approximation



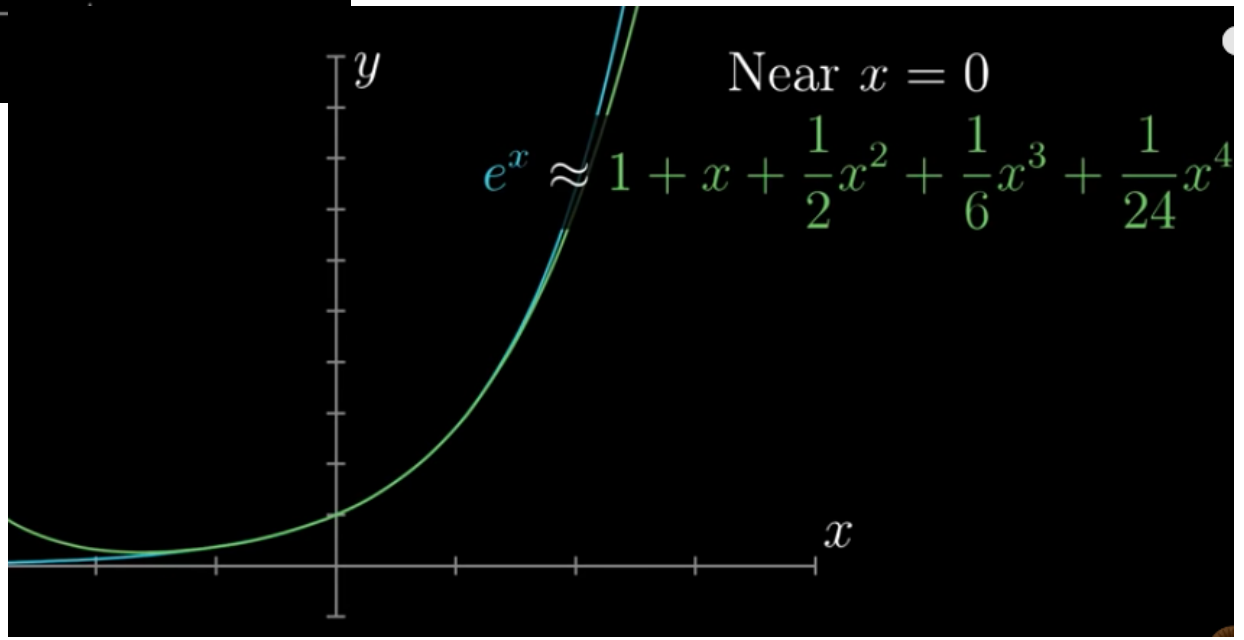


Image from Taylor series | Essence of calculus, chapter 11, www.youtube.com/watch?v=3d6DsjIBzJ4

Taylor series approximation

Common case: expansion near x=a:

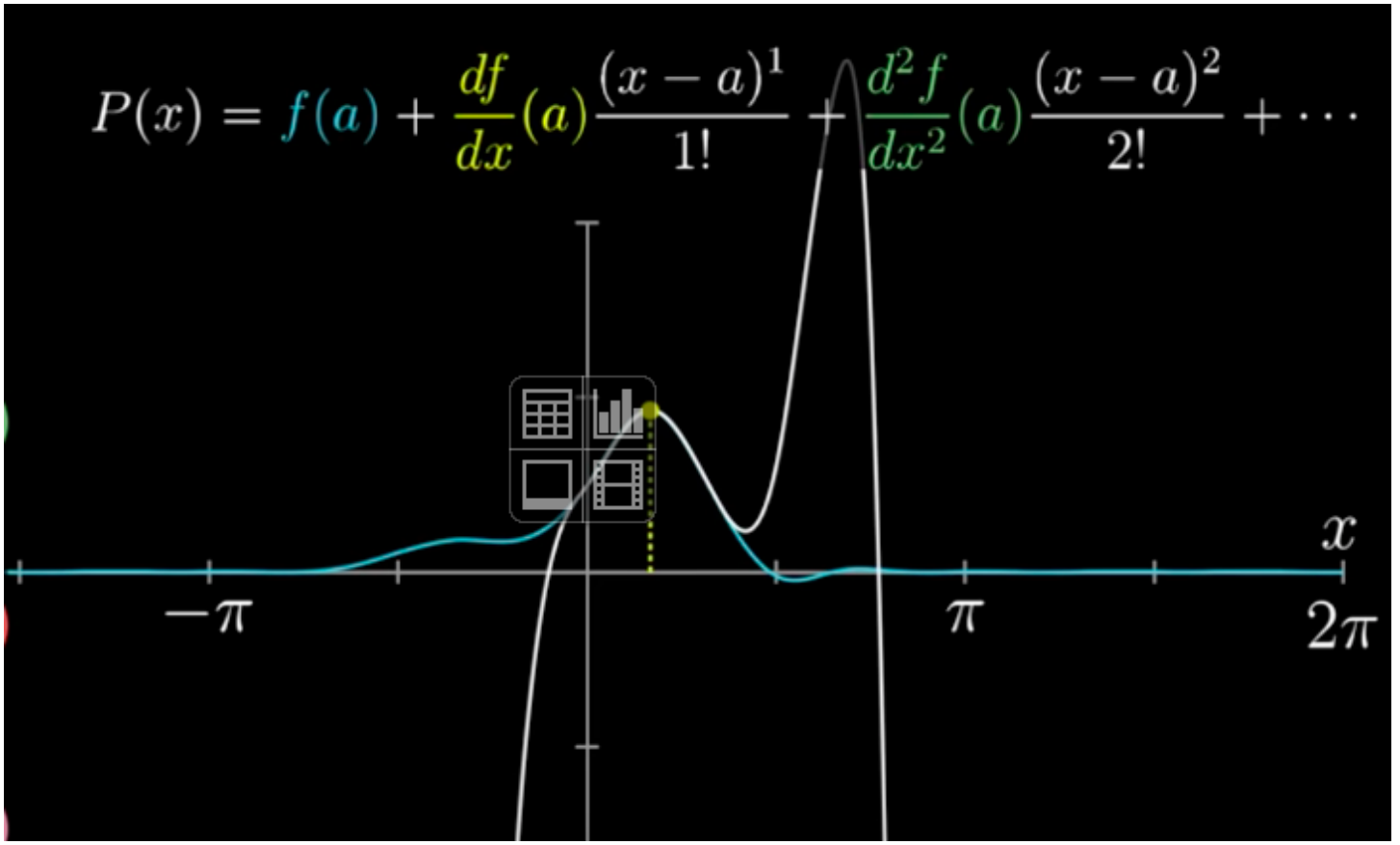


Image from Taylor series | Essence of calculus, chapter 11, www.youtube.com/watch?v=3d6DsjIBzJ4

Multivariate function approximation

Linear approximation near x=a

- For 2 variables:

$$f(x_1, x_2) \approx f(a_1, a_2) + \frac{\partial f(a_1, a_2)}{\partial x_1}(x_1 - a_1) + \frac{\partial f(a_1, a_2)}{\partial x_2}(x_2 - a_2)$$

- For many variables:

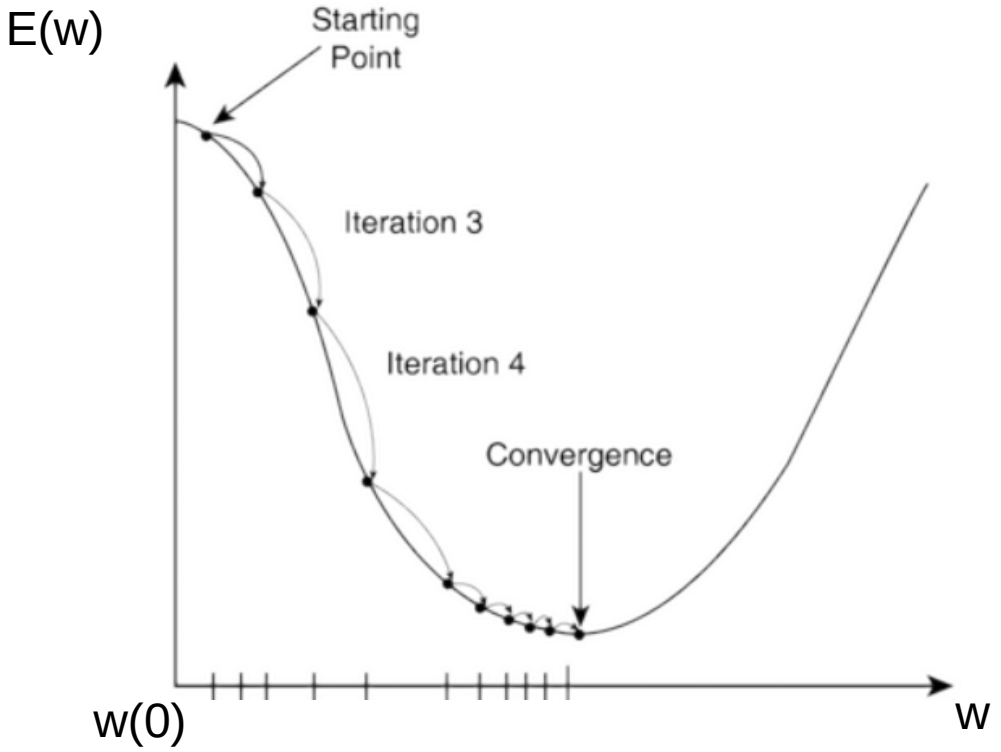
$$f(\vec{x}) \approx f(\vec{a}) + \nabla f(\vec{a})^T (\vec{x} - \vec{a})$$

gradient is a vector of partial derivatives

1D gradient descent: idea

- Start with random weights w(0)
- Until ?convergence?:

  take a (small) step in the direction of descent
  - Left if E’(w)>0
  - Right E’(w)<0



Gradient descent: fixed step size

- Start with random weights w(0)
- Until ?convergence?:

$$w(t+1) := w(t) - \eta \, sign(E'(w(t)))$$

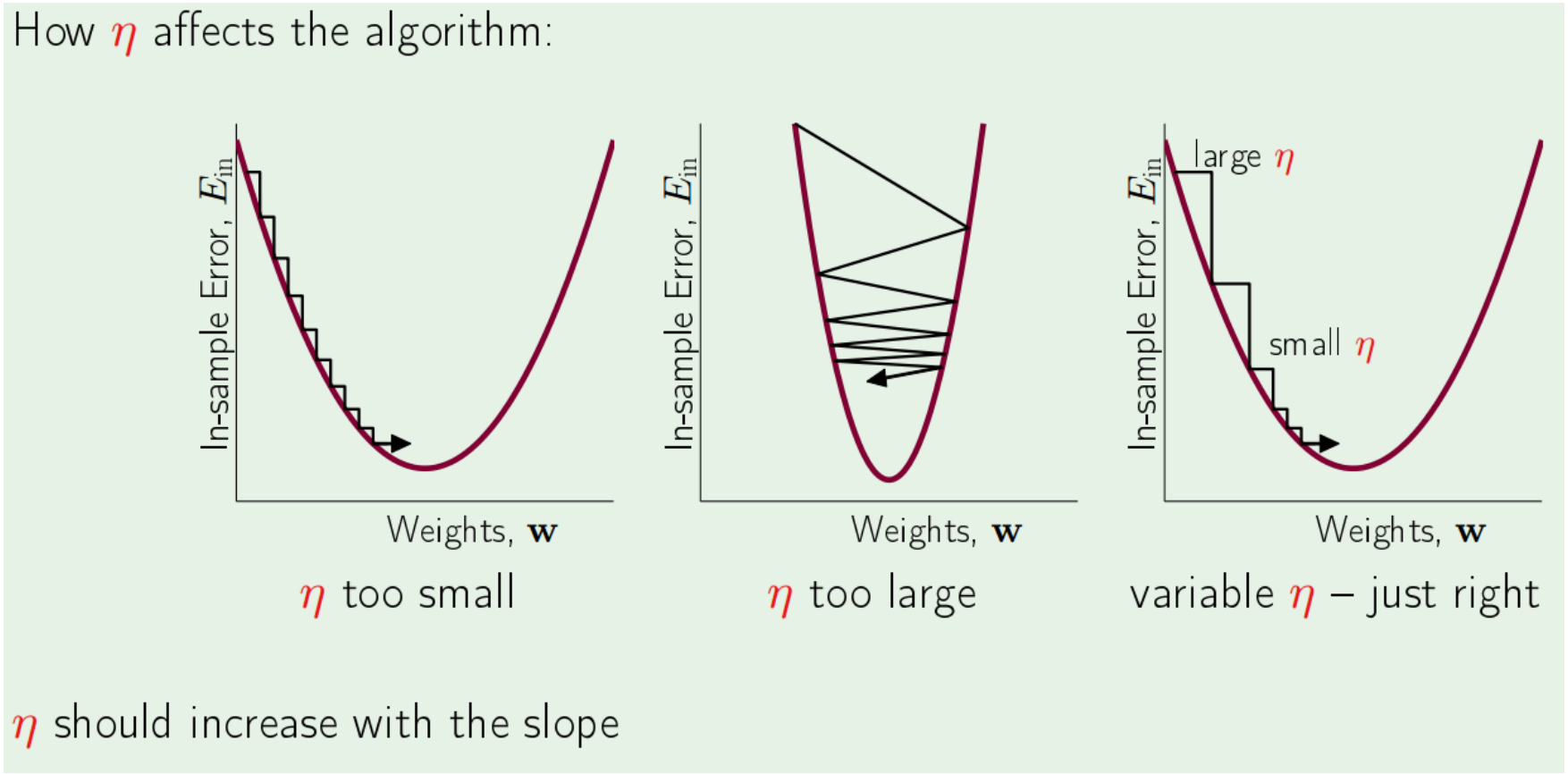


Image from Yaser Abu-Mostafa. Machine learning video library (https://work.caltech.edu/library/)

Gradient descent: fixed learning rate

- Start with random weights w(0)
- Until ?convergence?:

$$w(t+1) := w(t) - \lambda E'(w(t)), \eta = \lambda \left| E'(w(t)) \right|$$

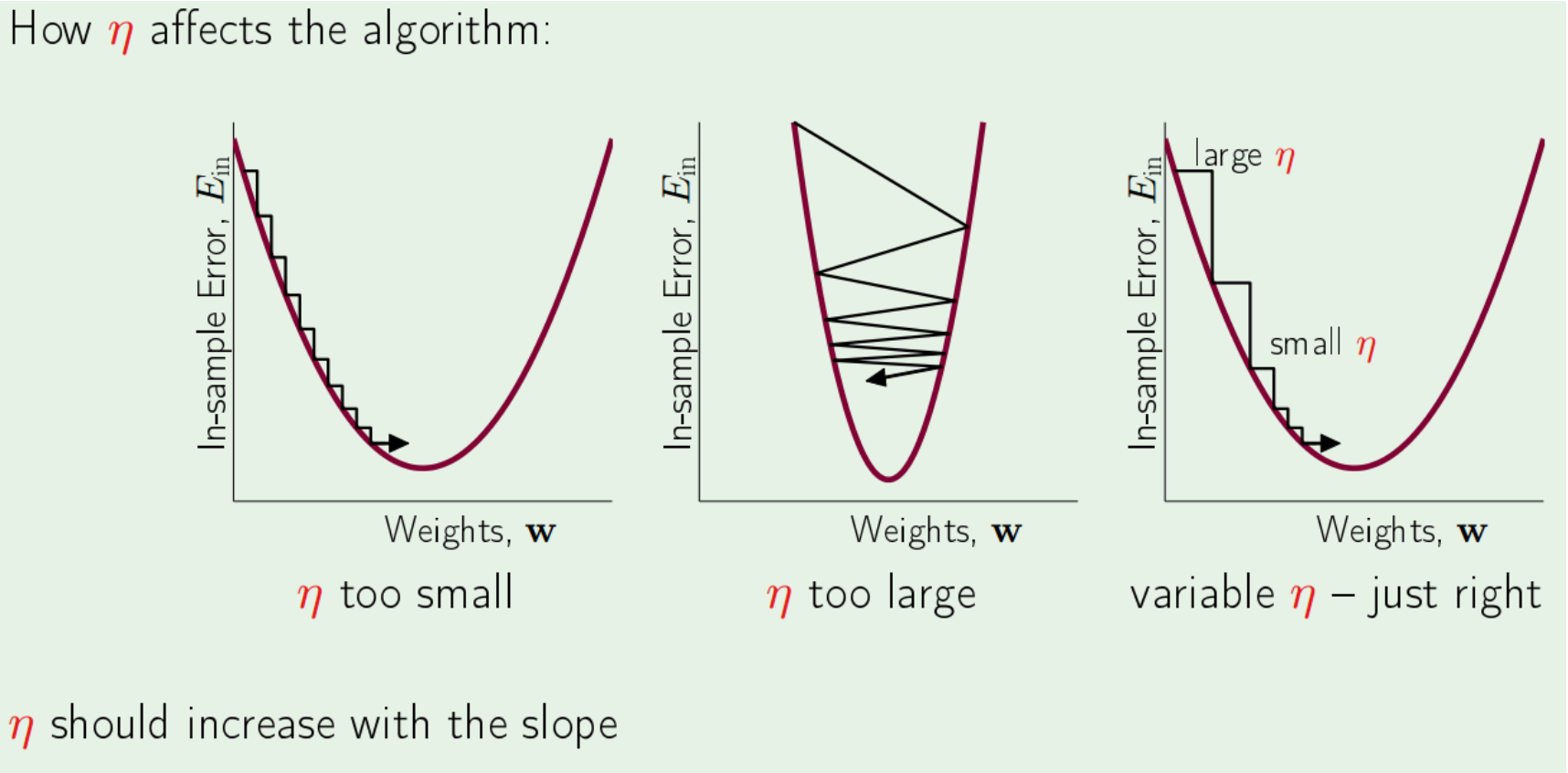
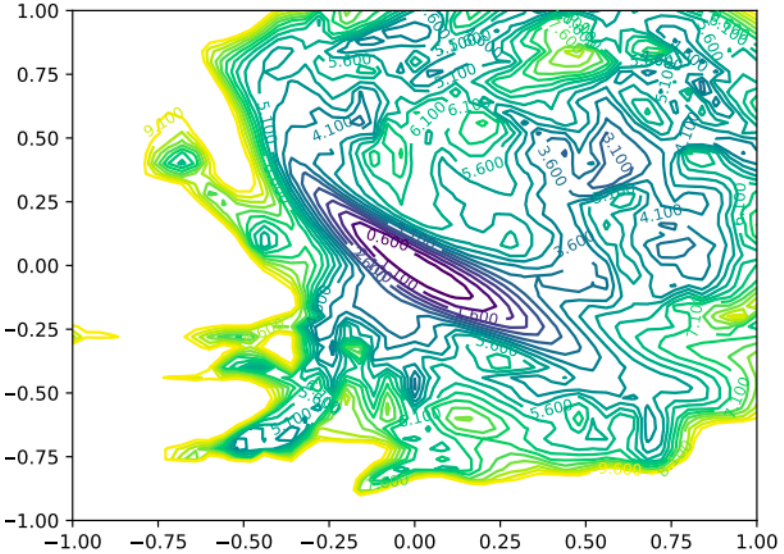
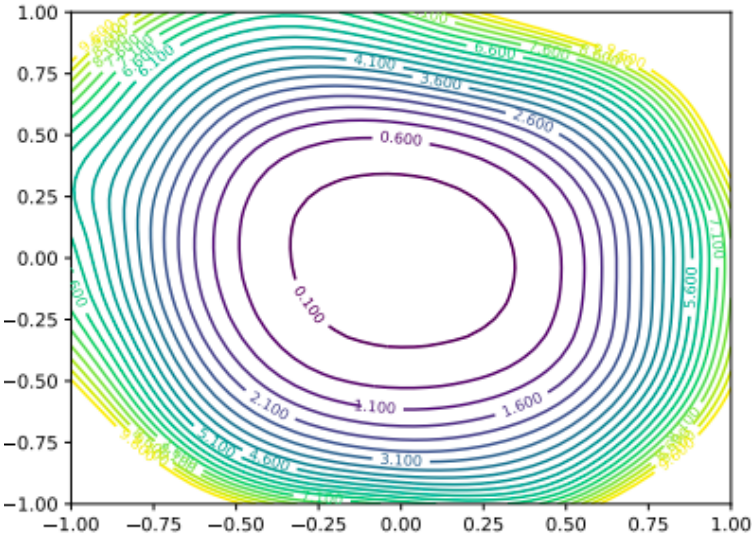


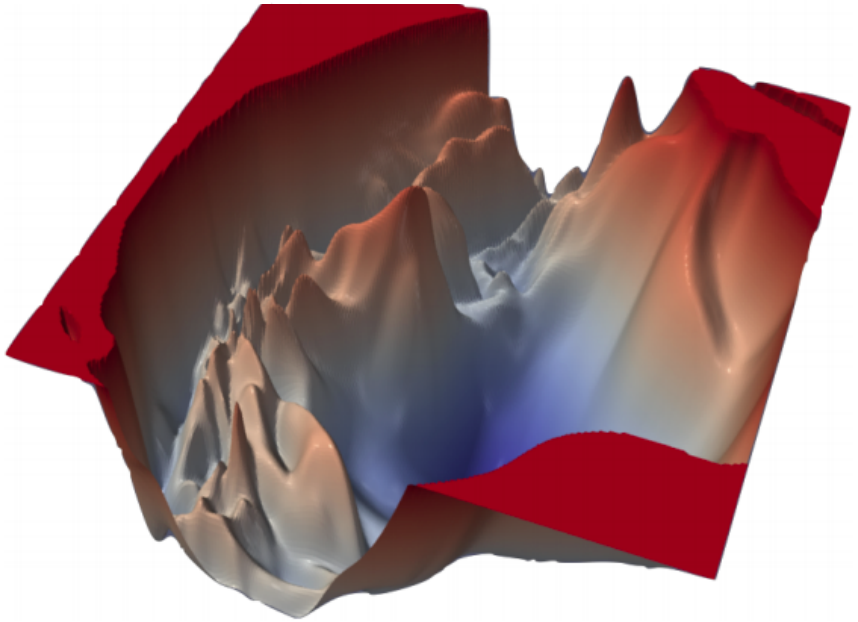
Image from Yaser Abu-Mostafa. Machine learning video library (https://work.caltech.edu/library/)

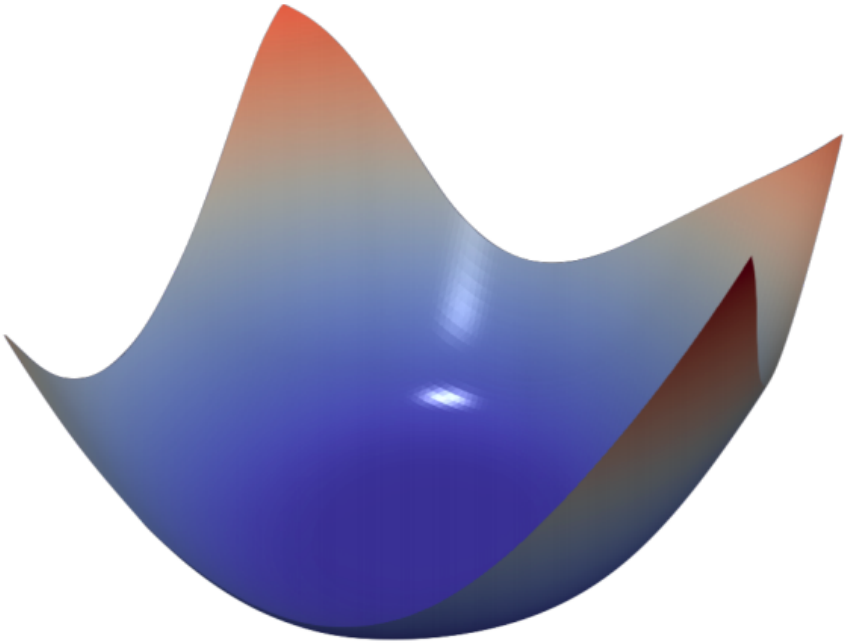
Loss surface visualization

- Parameters space dimensionality reduced → information lost (real surface is more complicated!)









Images from Li et al. VISUALIZING THE LOSS LANDSCAPE OF NEURAL NETS, 2018

Gradient descent: direction?

Using only **local** information which is the most promising direction to move w?

$$E(w(0)+\eta\vec{v}) \approx E(w(0)) + \eta \nabla E(w(0))^T \vec{v} \rightarrow \underset{\|\vec{v}\|=1}{argmin}$$

$$\nabla E(w(0))^T \vec{v} = \|\nabla E(w(0))\| \|\vec{v}\| \cos(\nabla E(w(0)), \vec{v})$$

$$\vec{v}^* = -\frac{\nabla E(w(0))}{\|\nabla E(w(0))\|}$$

Gradient shows steepest ascent direction

=> go in the opposite direction

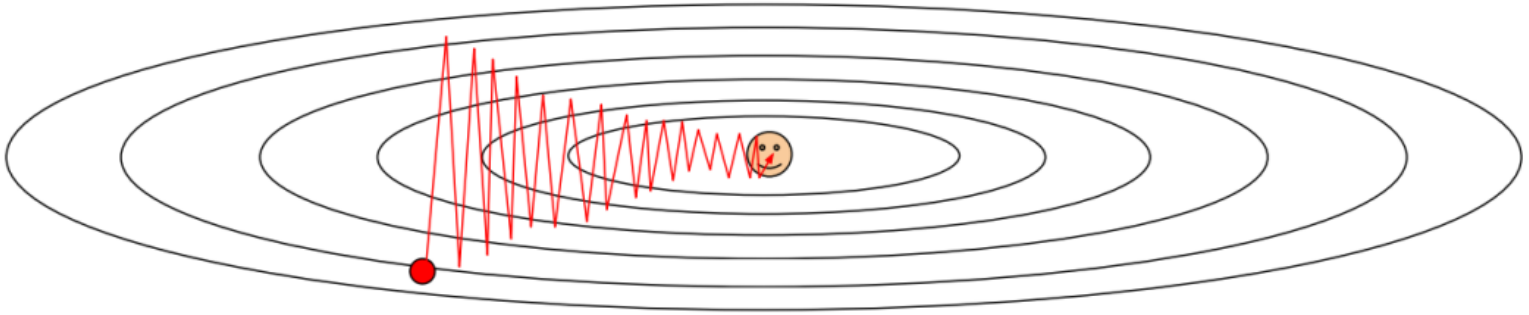
$$w(t+1) := w(t) - \lambda \nabla_w E(w(t))$$

Gradient descent weights update rule

Gradient descent in practice

Gradient doesn’t point towards minimum!

- Gradient is orthogonal to contour lines.
- When contour lines of the error surface are elongated ellipses, convergence is very slow.
- Consider bag-of-words representation



Help gradient descent: ← next lectures

- Normalize inputs
- Use advanced optimization (Momentum, Adam)

Gradient descent in practice

Gradient descent needs many (thousands, millions) steps to converge, but even 1 step can be slow (sum over whole dataset).

- Estimate loss & its gradient on part of the train set

  known as SGD (stochastic/mini-batch gradient descent)

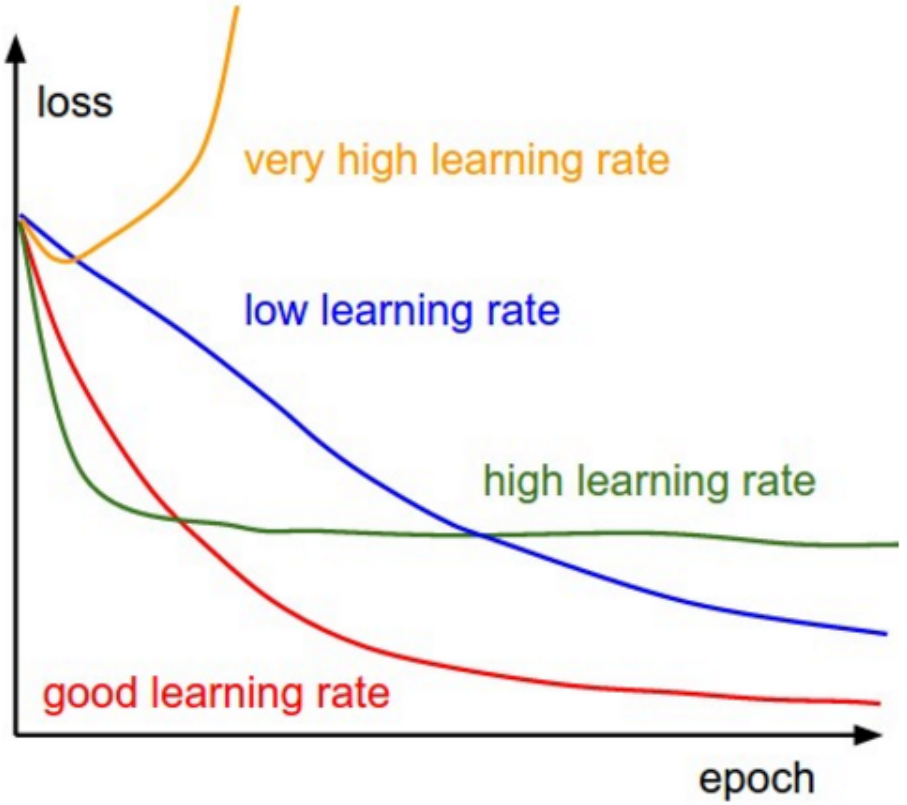
  - People use small batches (1-10 ex) and large batches (1k-10k ex), 100 ex is a good default
- Lower learning rate while training

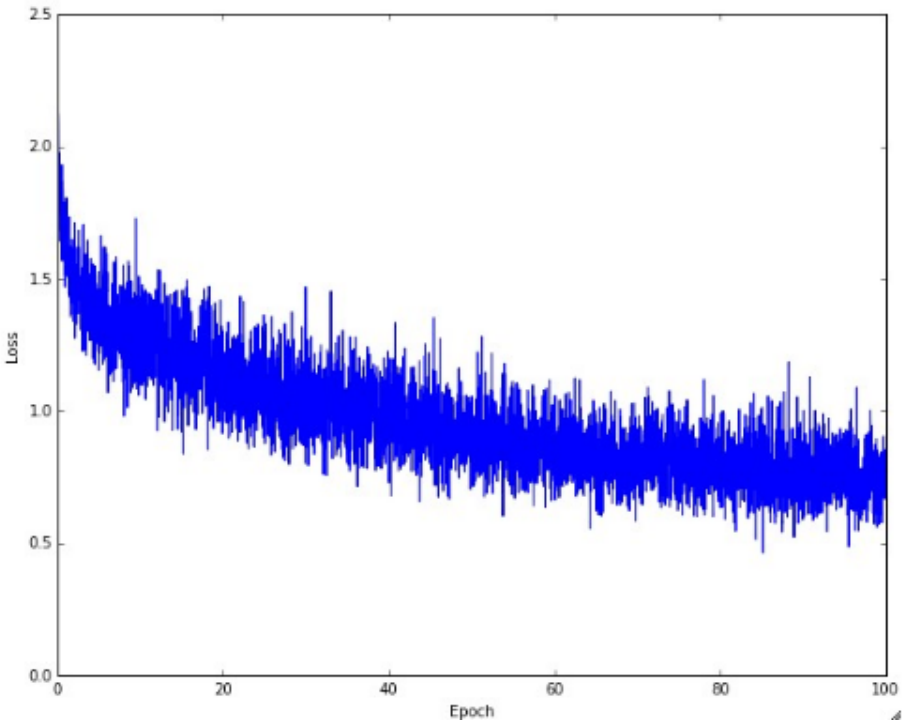
  known as learning rate decay / annealing
- Shuffle dataset
  - What happens if we showed first all cats, then all dogs?

Learning curves

Always plot learning curves (how loss/accuracy changes while training)!

- Select learning rate resulting in fastest loss descrease
- Lower learning rate when the loss plateaus



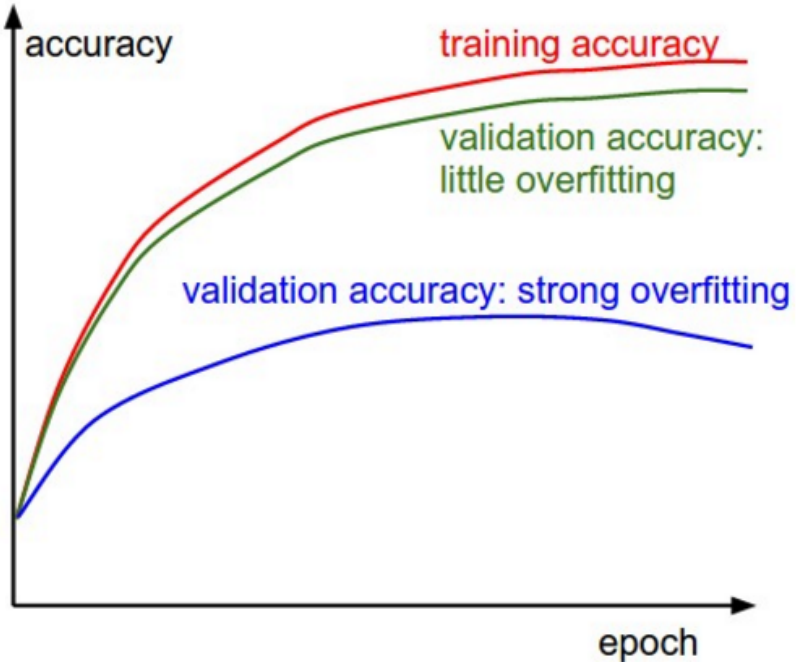


Images from CS231n Convolutional neural Networks for Visual Recognition course
http://cs231n.github.io/neural-networks-3/

Learning curves

Plot learning curves for both train and validation sets!

- Remember our final goal: work on new examples!
- Distinguish bad model vs. bad optimization
  - Loss/accuracy is bad on validation set, but good on train set
    => bad model (overfitting)
  - Loss/accuracy on train set is bad
    => maybe optimization problems?



Images from CS231n Convolutional neural Networks for Visual Recognition course
http://cs231n.github.io/neural-networks-3/

Gradient for LR (1 example)

- Hypothesis:

$$(preactivation)\ z = [1; x]^T w$$

$$\hat{y} = h_w(x) = \sigma(z) = \frac{1}{1+e^{-z}}$$

- Loss:

$$bce(w) = -\left(y \log\left(h_w(x)\right) + (1-y) \log\left(\left(1-h_w(x)\right)\right)\right)$$

- Derivatives: ← DERIVE

$$\frac{\partial bce(w)}{\partial w_j} = (\hat{y} - y)\, x_j$$

$$\Rightarrow \nabla_w bce(w) = (\hat{y} - y)\, x$$

Gradient for LR

- Hypothesis:

$$(preactivation)\ z = [1; x]^T w$$

$$\hat{y} = h_w(x) = \sigma(z) = \frac{1}{1+e^{-z}}$$

- Loss:

$$BCE(w) = -\frac{1}{N}\sum_{i=1}^{N} y_{\{i\}} \log\left(h_w(x_{\{i\}})\right) + \left(1 - y_{\{i\}}\right) \log\left(\left(1 - h_w(x_{\{i\}})\right)\right)$$

- Derivatives: ← DERIVE

$$\frac{\partial\, BCE(w)}{\partial\, w_j} = \frac{1}{N}\sum_{i=1}^{N} \left(\hat{y}_{\{i\}} - y_{\{i\}}\right) x_{\{i\},j}$$

$$\Rightarrow \nabla_w BCE(w) = \frac{1}{N}\sum_{i=1}^{N} \left(\hat{y}_{\{i\}} - y_{\{i\}}\right) x_{\{i\}} \quad \leftarrow \text{VECTORIZE}$$

Training LR with SGD

- Initialize w randomly (0s are o.k. for LR, but not for NN)
- Until ?convergence?
  - From the train get next mini-batch
  - On mini-batch Compute loss E(w) and it's gradient
  - Update weights $$w(t+1) := w(t) - \lambda \nabla_w E(w(t))$$
  - ?Change learning rate if necessary?

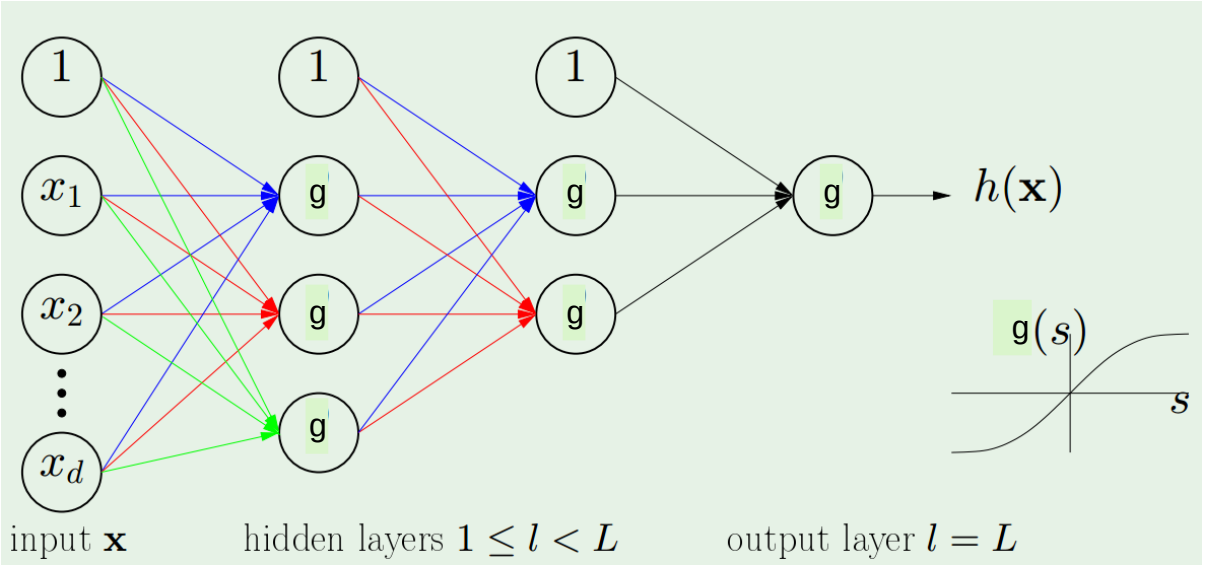
Training FFNN with SGD

- **Initialize w randomly**
  - 0s don’t work: need symmetry breaking
  - Use small random weights, better from specific distributions (Xavier, He, Glorot initialization)
- Until ?convergence?
  - From the train get next mini-batch
  - On mini-batch Compute loss E(w) and it’s **gradient**
  - Update weights $$w(t+1) := w(t) - \lambda \nabla_w E(w(t))$$
  - ?Change learning rate if necessary?

Gradient for FFNN

- Hypothesis:

$$a^{(0)} = x^T$$
$$z^{(l)} = [1; a^{(l-1)}] W^{(l)}$$
$$a^{(l)} = g^{(l)}(z^{(l)})$$
$$\hat{y} = h_W(x) = a^{(L)}$$



- Loss:

$$BCE(w) = -\frac{1}{N}\sum_{i=1}^{N} y_{\{i\}} \log\left(h_w(x_{\{i\}})\right) + \left(1 - y_{\{i\}}\right) \log\left(\left(1 - h_w(x_{\{i\}})\right)\right)$$

- Derivatives:
  - Can use symmetric difference approximation ← very slow, use for gradient checks!
  - Can write down formula for E(w) containing all w explicitly and differentiate ← too complicated and errorprone
  - Backprop – an efficient way to calculate derivatives recurrently, based on chain rule

Image from Yaser Abu-Mostafa. Machine learning video library (https://work.caltech.edu/library/)

Chain rule

- For 1 argument:

$$E(w)=f(g(w))$$
$$E'(w)=f'(g(w))g'(w)$$

- For 2 arguments: ← sum over internal functions!

$$E(w)=f(g(w),h(w))$$
$$\frac{\partial E(w)}{\partial w}=\frac{\partial f(g(w),h(w))}{\partial g(w)}\frac{\partial g(w)}{\partial w}+\frac{\partial f(g(w),h(w))}{\partial h(w)}\frac{\partial h(w)}{\partial w}$$

Backprop = (Error) Backpropagation

- Define error as $\delta_j^{(l)} = \frac{\partial e(w)}{\partial z_j^{(l)}}$
- Calculate for the last layer
  - for sigmoid + BCE
    y is 0/1 $\delta_1^{(L)} = \hat{y} - y$
  - similar for softmax + CE
    y is one-hot vector $\delta^{(L)} = \hat{y} - y$
- Recurrently calculate error in (l-1)-th layer from error in l-th layer

$$\delta^{(l-1)} = \ldots \delta^{(l)} \ldots$$

- Calculate gradient w.r.t. weights from errors

$$\nabla_{W^{(l)}} e = \ldots \delta^{(l)} \ldots$$

Backprop = (Error) Backpropagation

- Define error as $\delta_j^{(l)} = \frac{\partial e(w)}{\partial z_j^{(l)}}$
- Calculate for the last layer
  - for sigmoid + BCE
    y is 0/1
    $\delta_1^{(L)} = \hat{y} - y$
  - similar for softmax + CE
    y is one-hot vector
    $\delta^{(L)} = \hat{y} - y$

Predicted-real output looks like error!

- Recurrently calculate error in (l-1)-th layer from error in l-th layer

$$\delta^{(l-1)} = \delta^{(l)} W_{[1:,:]}^{(l)T} \circ g'\left(z^{(l-1)}\right)$$

Backpropagate error

- Calculate gradient w.r.t. weights from errors

$$\nabla_{W^{(l)}} e = a^{(l-1)T} \delta^{(l)}$$

For efficiency:

  - In forward prop save intermediate quantities required by backprop!
  - Use matrices D, Z, A of shape (batch_size, layer_size) and adapt formulas, always keep batch_size as first axis (common practice)

Initializing FFNN weights

$$w_{ij}^{(l)}(t) := w_{ij}^{(l)}(t-1) - \lambda \delta_j^{(l)} a_i^{(l-1)}$$

If all weights initialized to 0s (or other constant):

- all activations in the previous layer will compute the same function (will be equal)
- all errors backpropagated from the next layer will be equal ← WHY?

all updates will be the same and all weights will stay the same. Similar to using hidden layers of size 1!

- Break symmetry: initialize weights with small random numbers

  from special distributions (Xavier/Glorot/He) to keep activations & gradients from vanishing or exploding.