Artificial Neural Networks (ANN)

Nikola Popović
nipopovic@vision.ee.ethz.ch

CVL Computer Vision Lab

ETH zürich

1st wave of NN – The birth of the idea

- [1943] First artificial neuron model – W. McCulloch , W. Pitts
- [1949] Hebb’s learning law – D. Hebb
- [1958] Perceptron – F. Rosenblatt
- [1962] Delta Learning Rule – B. Widrow, T. Hoff

2nd wave – Excitement again

- [1986] Bacpropagation popularized - D. Rumelhart, G. E. Hinton, R. Williams
- [1997] LSTM’s - H. Sepp, J. Schmidhuber
- [1998] CNN’s– Y. LeCun, L. Bottu, Y. Bengio, P. Haffner

3rd wave – ANN seem to work

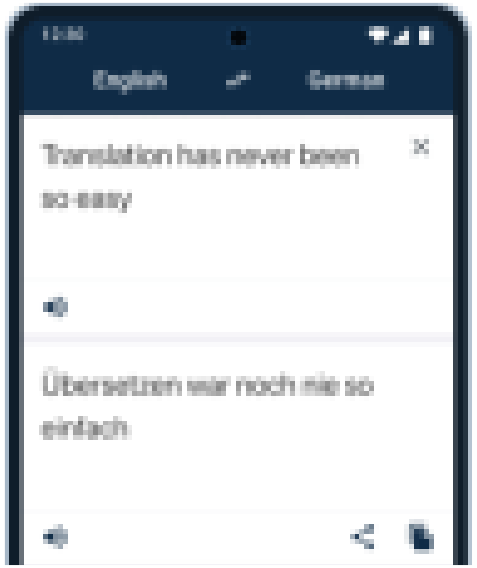
- Much better hardware and software
- Much bigger data sets

- Some influential works
  - [2006] Deep Belief Networks – G. E. Hinton, S. Osindero, Y.-W. Teh
  - [2009] Speech processing – G. E. Hinton, L. Deng
  - [2012] Image classification – A. Krizhevsky, I. Sutskever, G. E. Hinton

- Many astonishing results followed

Deep Learning applications

Machine translation



https://www.deepl.com/translator

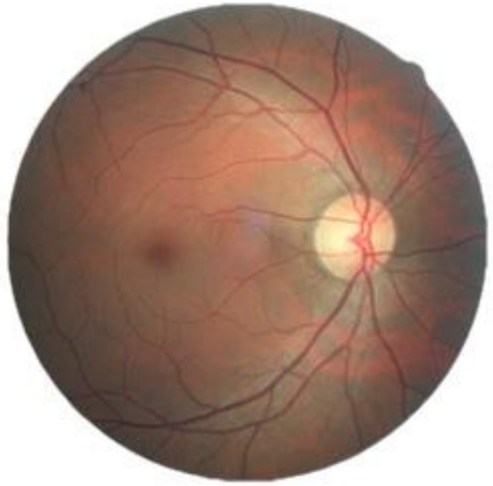
Autonomous cars

https://www.ptolemus.com/topics/autonomous-vehicles/

Speech recognition

https://play.google.com/store/apps/details?id=com.prometheusinteractive.voice_launcher

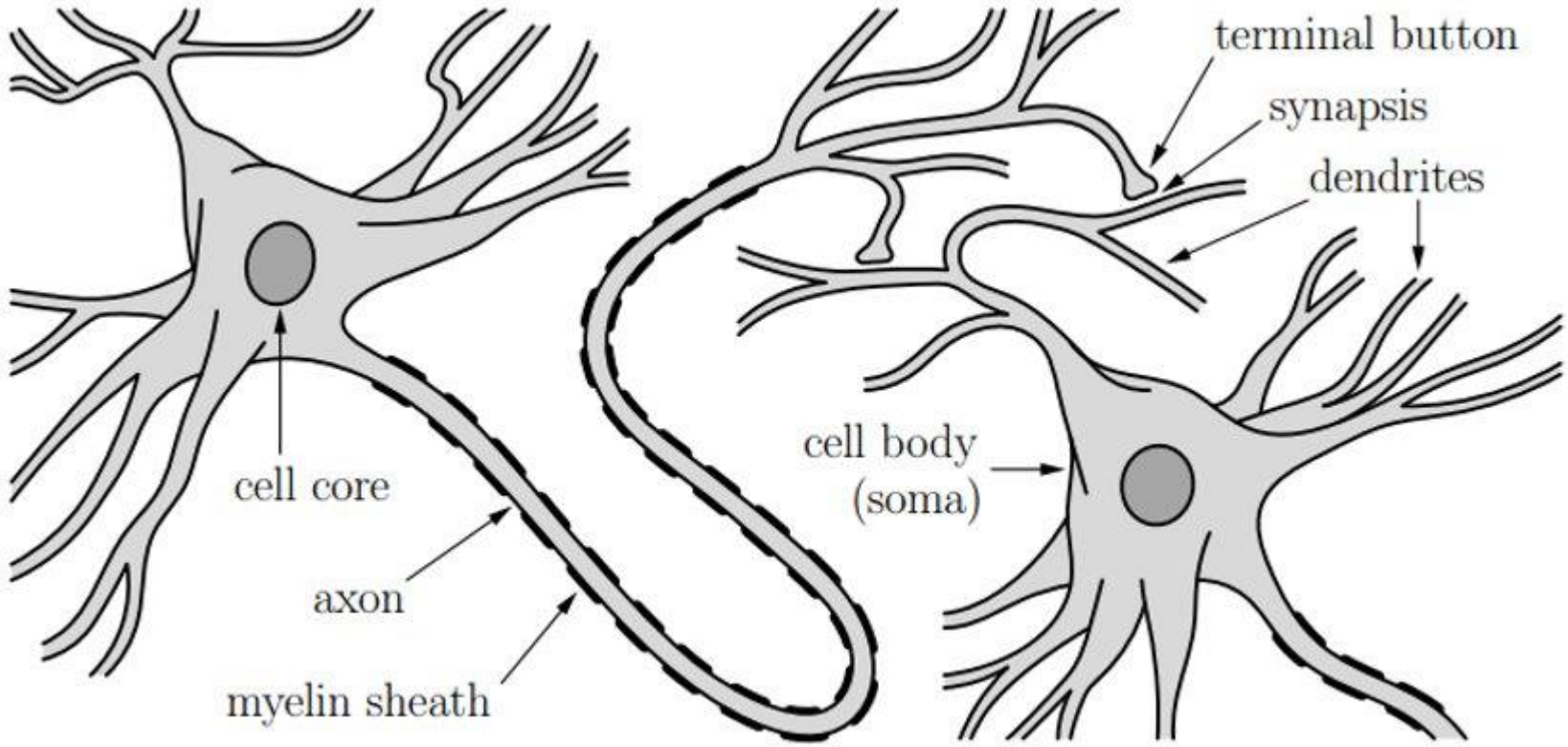
Diabetic retinopathy detection



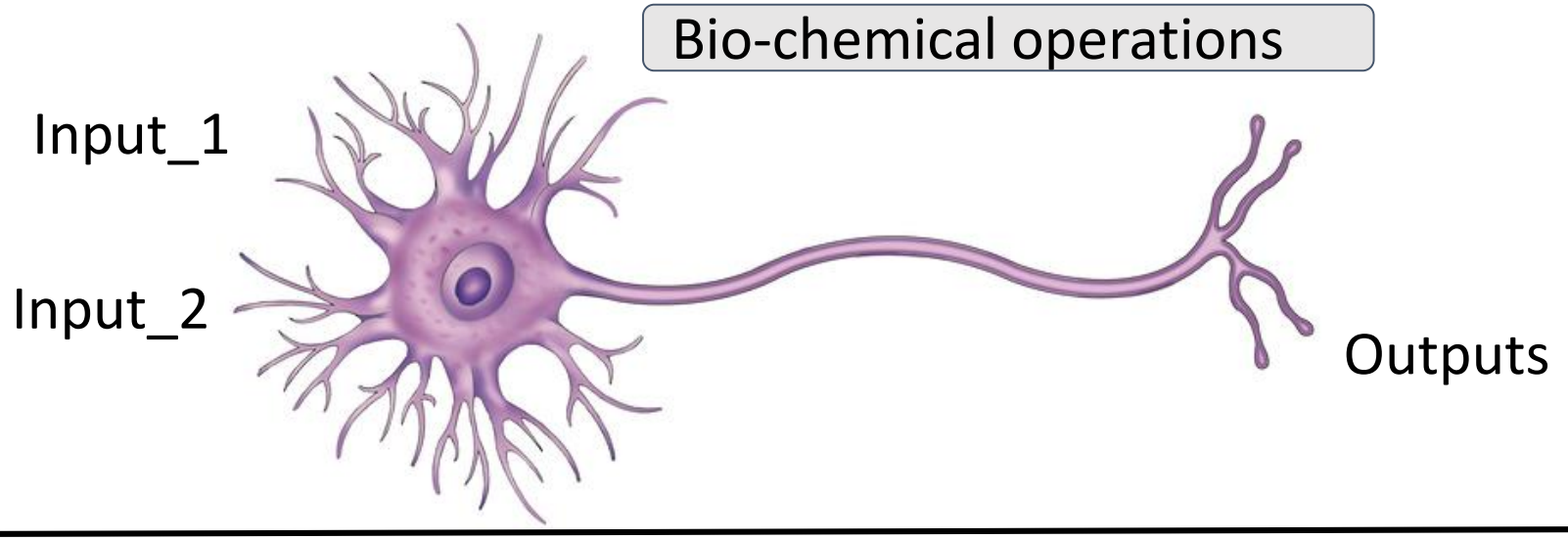
https://www.kaggle.com/c/diabetic-retinopathy-detection

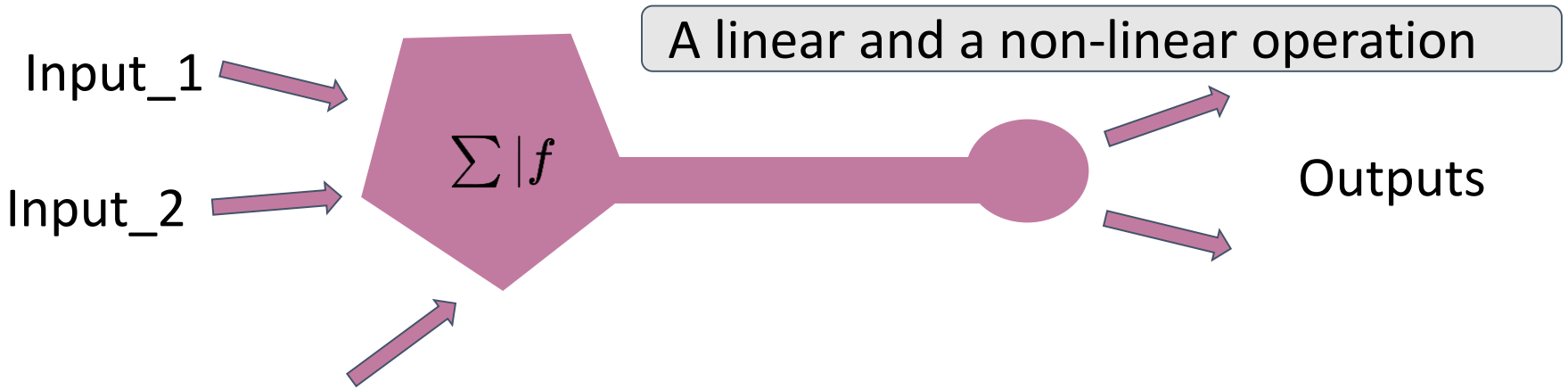
Biological and Artificial Neurons

- ANN were inspired by human nervous system



Biological and Artificial Neurons





One Artificial Neuron

- Input $\boldsymbol{x} = [x_1, \ldots, x_m]^T$
- Weights $\boldsymbol{w} = [w_1, \ldots, w_m]$
- Bias b
- Pre-activation $z = \sum_i x_i \, w_i + b = \boldsymbol{w}\boldsymbol{x} + b$
- Activation fn. $f(\cdot)$
- Neuron output $y = f(z) = f(\sum_i x_i \, w_i + b)$

ANN
Single layer

- Input – vector $\boldsymbol{x} = [x_1, \dots, x_m]^T$
- Different weights for each of k neurons

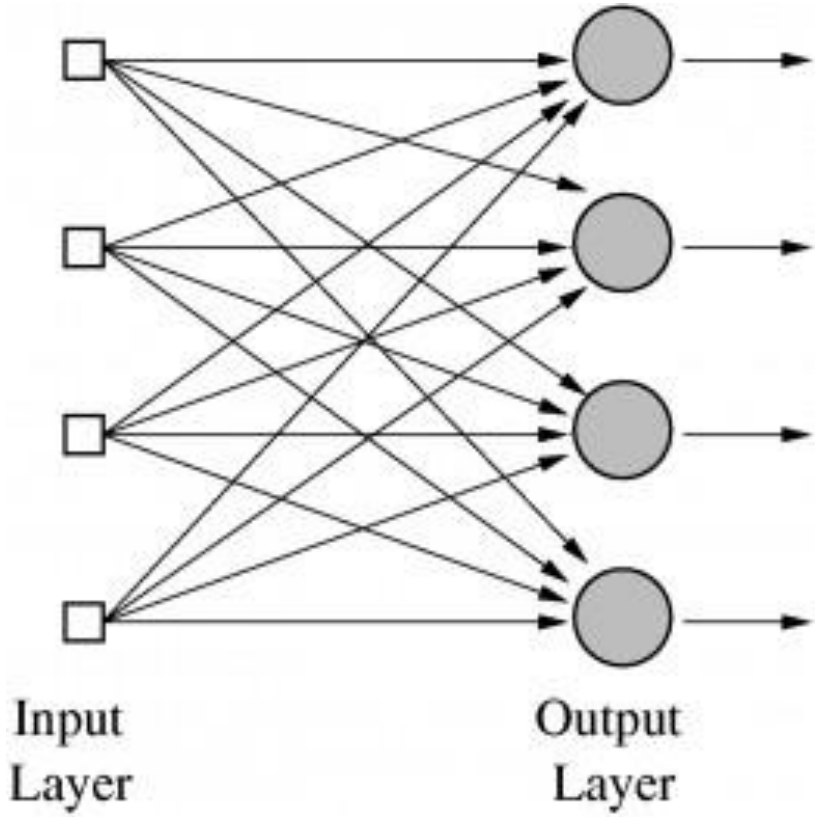
$$\boldsymbol{W} = \begin{bmatrix} w_{11} & \cdots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{k1} & \cdots & w_{km} \end{bmatrix} = \begin{bmatrix} \boldsymbol{w_1} \\ \dots \\ \boldsymbol{w_k} \end{bmatrix}$$

- Pre-activation vector

$$\boldsymbol{z} = [z_1, \dots, z_k]^T = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

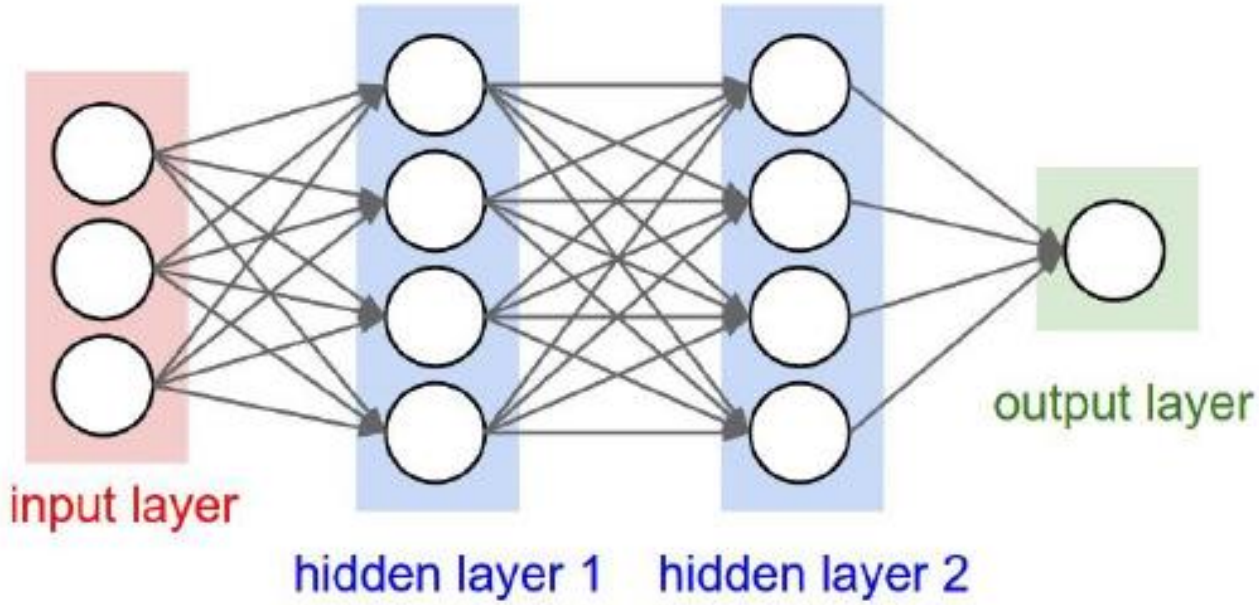
- Output vector

$$\boldsymbol{y} = [y_1, \dots, y_k]^T = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$



Multi Layer ANN

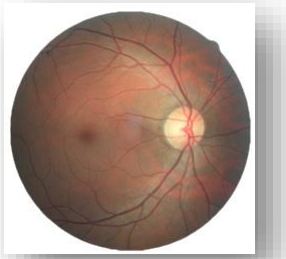
- Input layer
  - The input vector
- Hidden layer(s)
  - Intermediate computation
  - Extract features
- Output layer
  - Final computation and decision making
- Nowadays called multi layer perceptron (MLP)



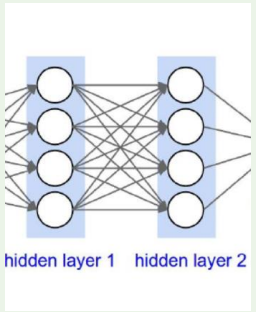
Stanford, CS231n course, Lecture 4 presentation

High-level view

- Binary classification example



Input image x
(pixel rbg values)

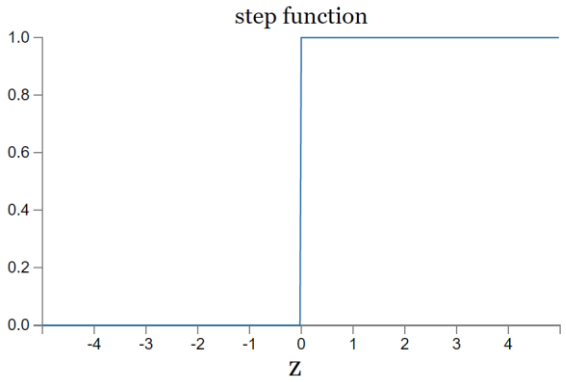


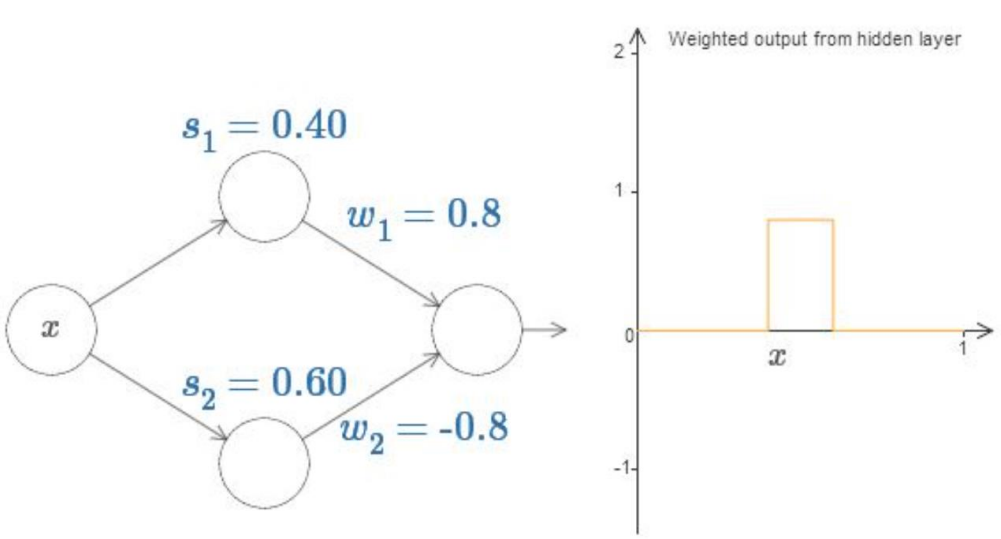
ANN
Model
(parameters: w, b)

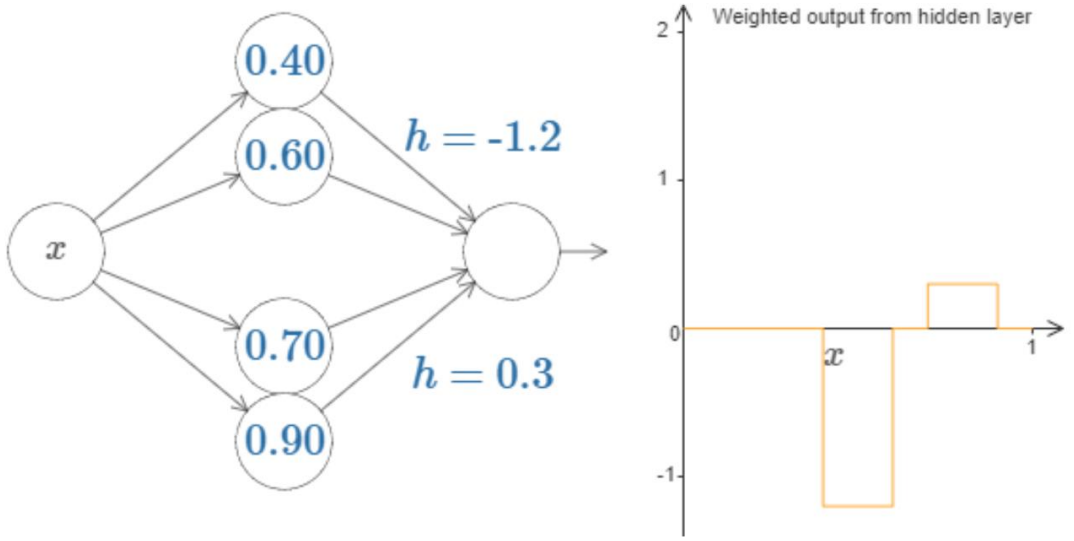
Output y

y = probability of an eye disease
y = 0 => healthy
y = 1 => disease

Intuition behind the power of ANN



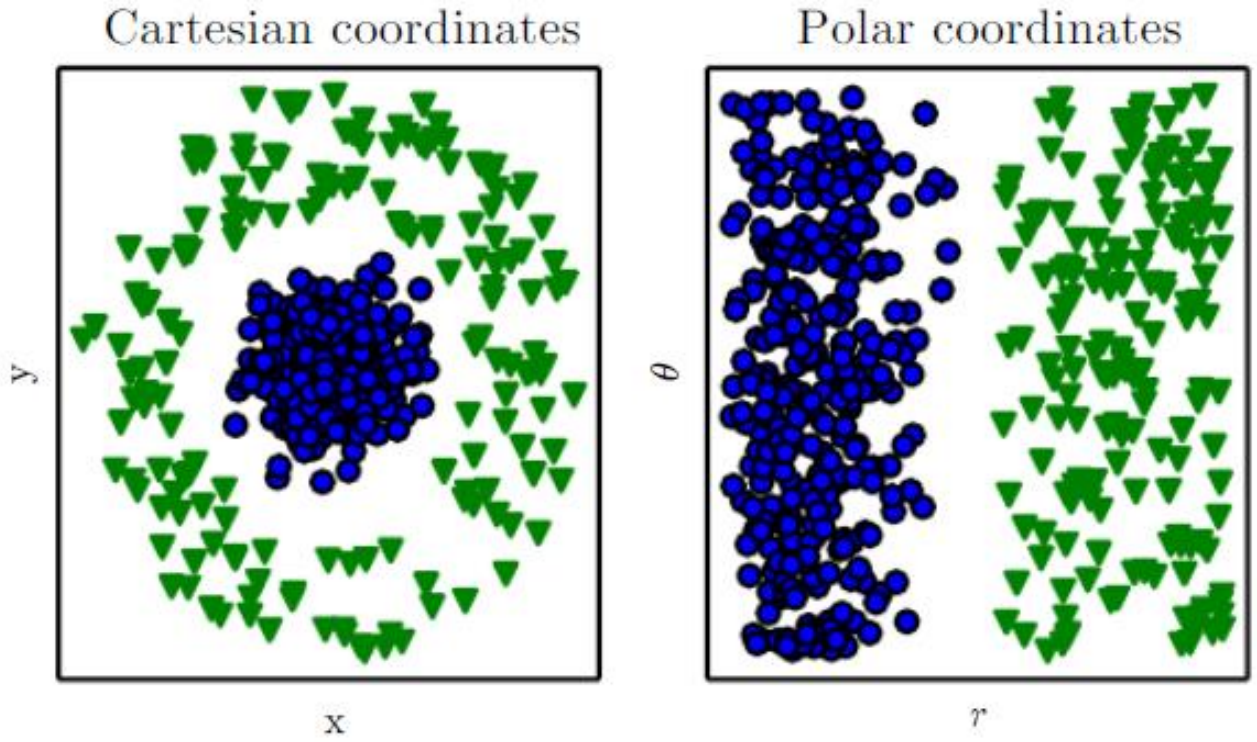




Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015

Motivation on need for good features

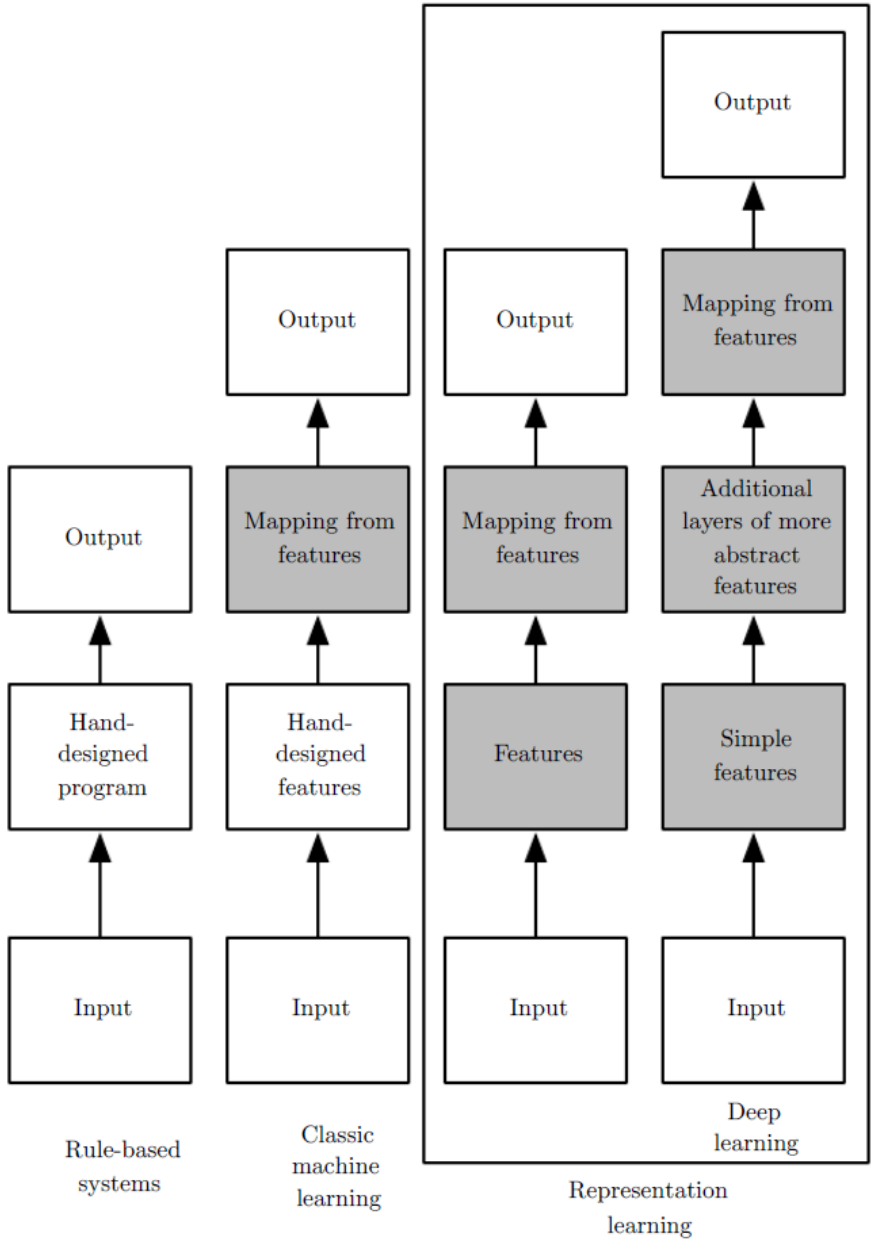
- The left plot cannot be seperated with a linear classifier
- With good features (right plot) we can seperate these two classes with a line



Goodfellow et al., Deep Learning, MIT Press 2016

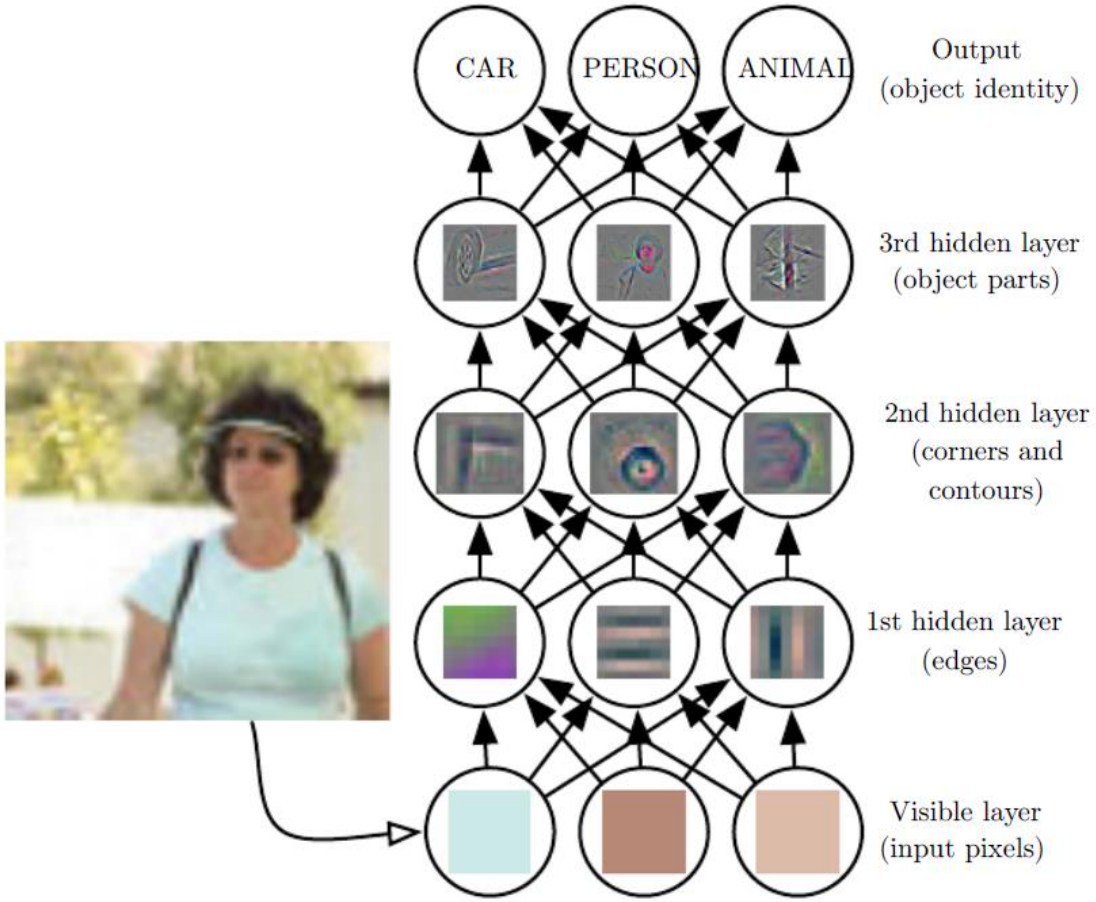
What is Deep Learning?

- Before – multiple stages
  - Designing features
  - Prediction algorithm (takes features as input)
- Now - Deep Learning
  - Raw data as input
  - Prediction as output



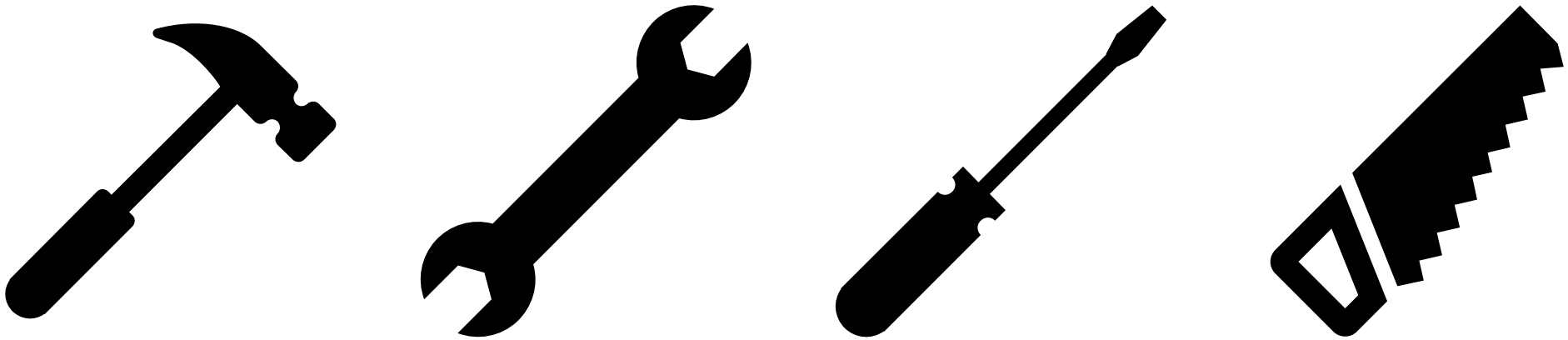
Goodfellow et al., Deep Learning, MIT Press 2016

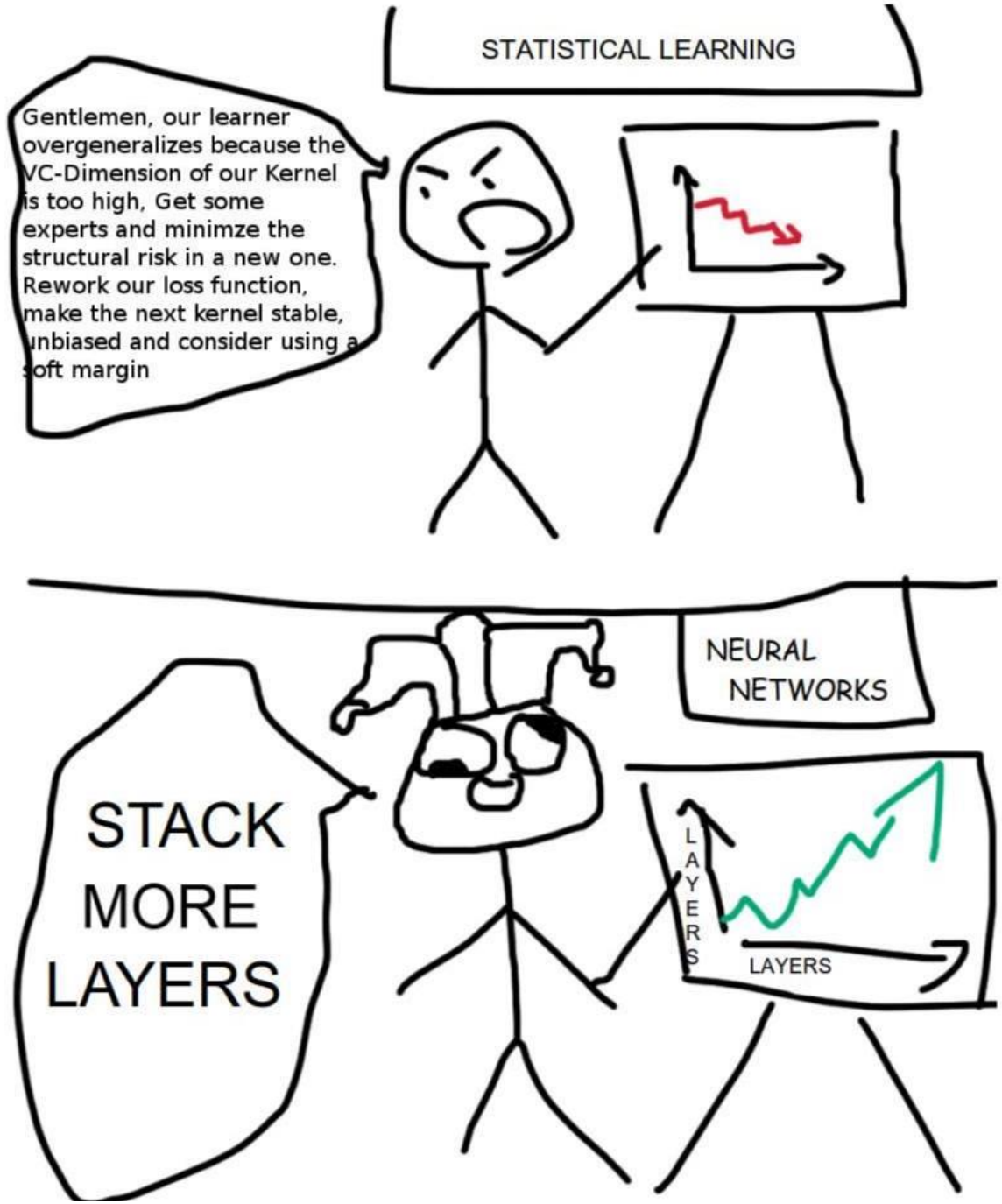
Intuition behind Deep ANN



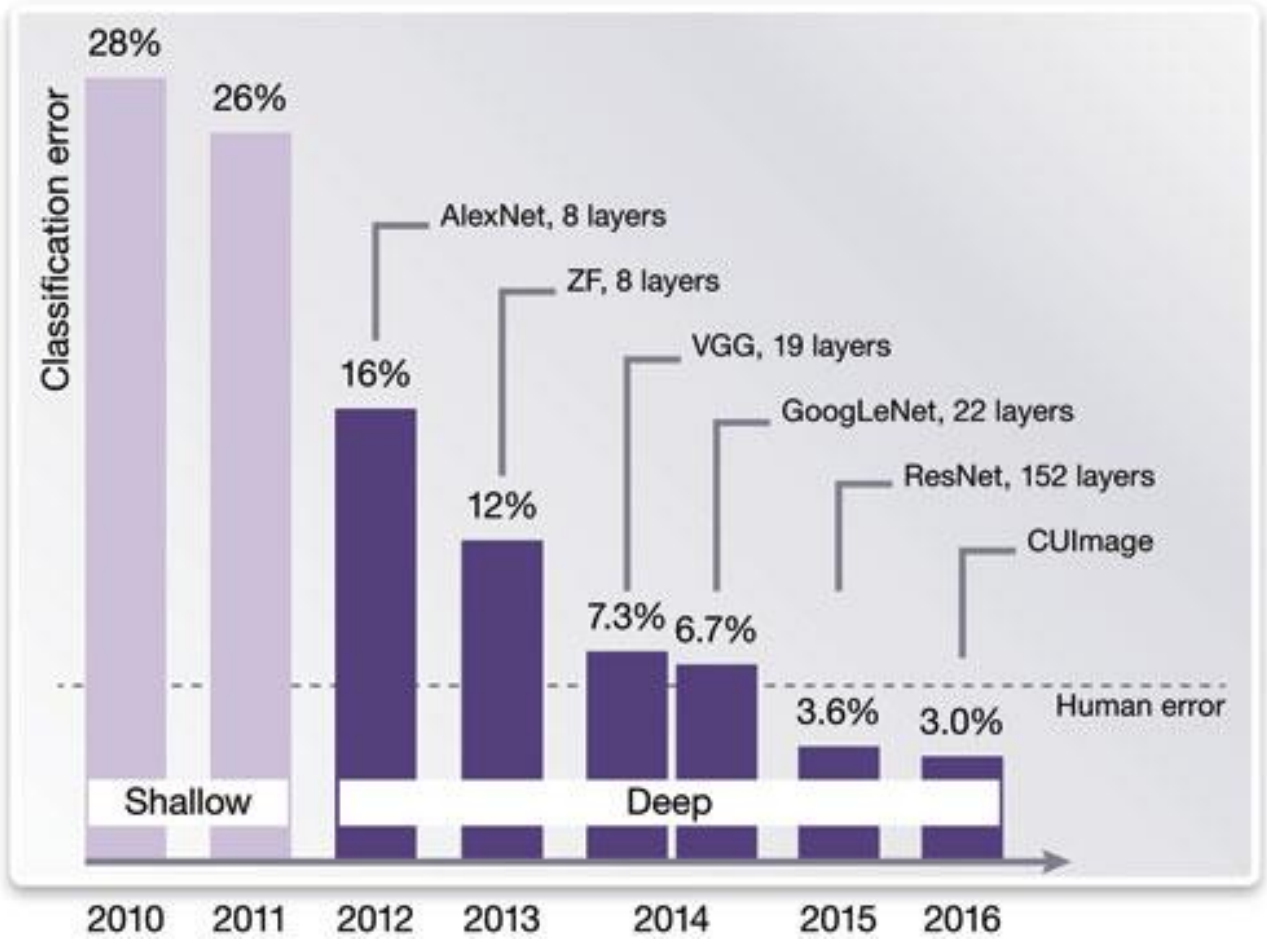
Goodfellow et al., Deep Learning, MIT Press 2016

ANN are tools




STATISTICAL LEARNING
Gentlemen, our learner overgeneralizes because the VC-Dimension of our Kernel is too high, Get some experts and minimze the structural risk in a new one. Rework our loss function, make the next kernel stable, unbiased and consider using a soft margin
NEURAL NETWORKS
STACK MORE LAYERS
LAYERS
LAYERS

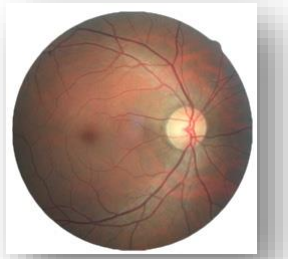
ImageNet Challenge



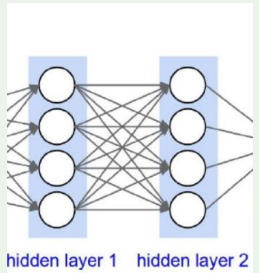
Learning

How do we get the model parameters (w,b)?

By designing loss functions and optimizing them.



Input image x
(pixel rbg values)



ANN
Model
(parameters: w, b)

y = probability of an eye disease
y = 0 => healthy
y = 1 => disease

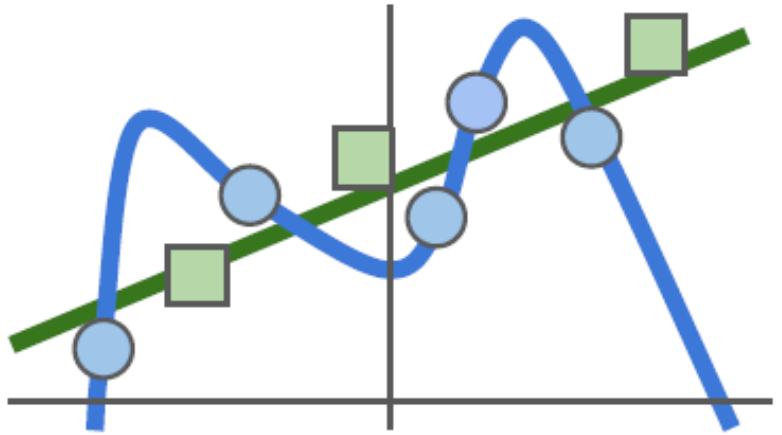
Output y

Supervised learning objective

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)$$

Loss function

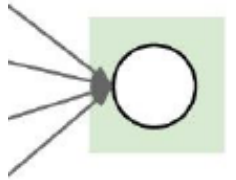
Model predictions should match training data



Stanford, CS231n course, Lecture 3 presentation

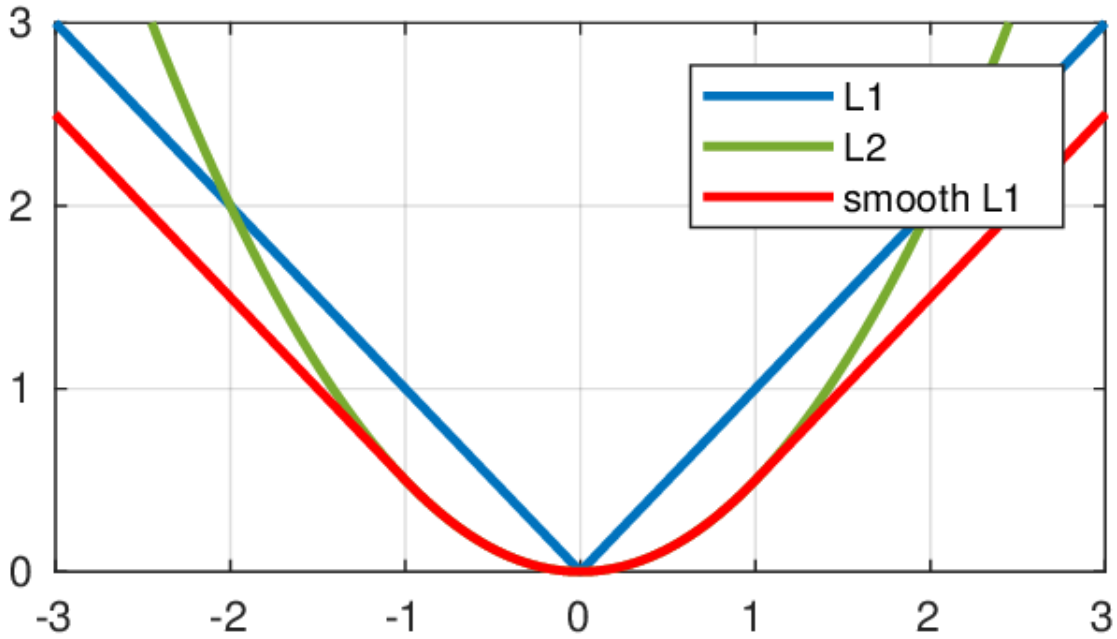
Regression

- In case of one output: $\hat{y}^{(i)} = f(x^{(i)}, \mathrm{W})$



- L_2 loss(MSE): $L_i = \left(\hat{y}^{(i)} - y^{(i)}\right)^2$
- L_1 loss: $L_i = \left|\hat{y}^{(i)} - y^{(i)}\right|$
- Smooth L_1 loss (less sensitive to outliers):

$$L_i = \begin{cases} L_2 \text{ loss} & \left|\hat{y}^{(i)} - y^{(i)}\right| \leq a \\ L_1 \text{ loss} & \left|\hat{y}^{(i)} - y^{(i)}\right| > a \end{cases}$$

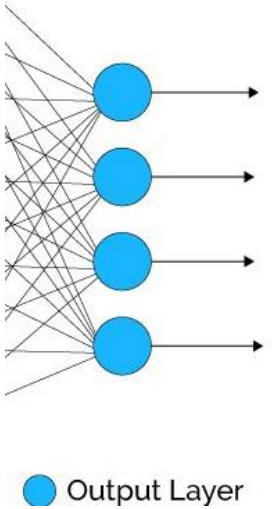


https://www.researchgate.net/figure/Plots-of-the-L1-L2-and-smooth-L1-loss-functions_fig4_321180616

Regression

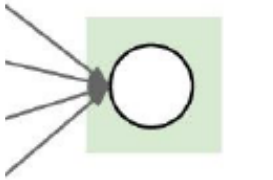
- For multiple outputs, summ the losses for each output

$$L_i = \sum_j L_{ij}$$



Sigmoid output layer (Classification)

- Binary classification problem with one output



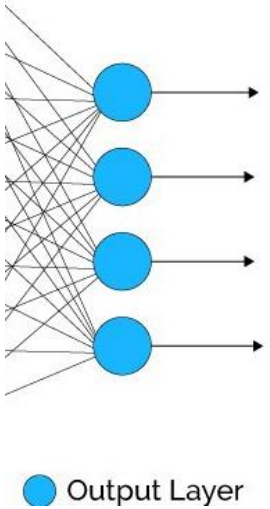
- Network output: $\hat{p} = P\left(Y = 1 \middle| X = x^{(i)}\right) = \frac{1}{1+e^{-net^{(i)}}}$
  - $net^{(i)} = f(x^{(i)}, \mathrm{W})$ (score for class 1)

- Cross-entropy loss

$$L_i = -\log P\left(Y = y^{(i)} \middle| X = x^{(i)}\right)$$

$$= -y^{(i)} \log(\hat{p}) - \left(1 - y^{(i)}\right) \log(1 - \hat{p})$$

Softmax output layer (Classification)



- Multi-class classes classification problem (m classes)

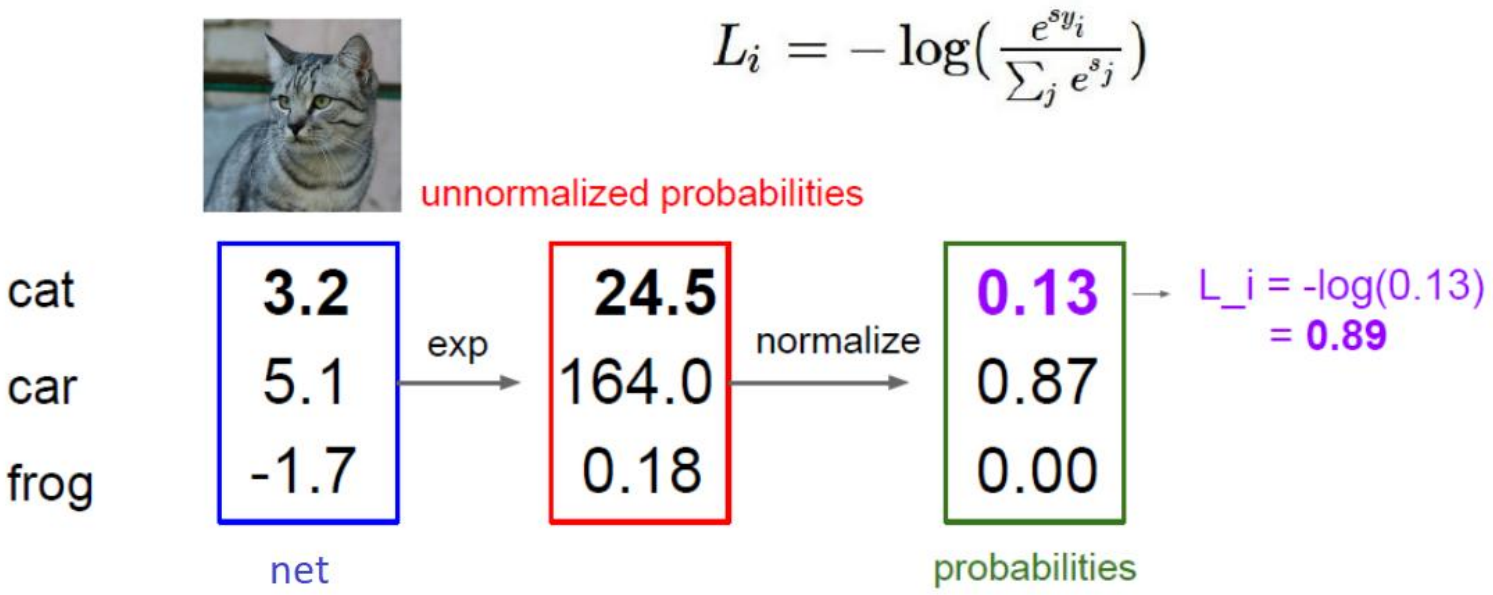
- Network outputs: $\hat{p}_k = P\left(Y = k \middle| X = x^{(i)}\right) = \frac{e^{net_k{}^{(i)}}}{\sum_{j=1}^{m} e^{net_j{}^{(i)}}}$
  - $net_k{}^{(i)} = f(x^{(i)}, \mathrm{W})$ – score for class k

- Negative log likelihood

$$L_i = -\log P\left(Y = y^{(i)} \middle| X = x^{(i)}\right)$$

$$= -\log\left(\frac{e^{net_{y^{(i)}}{}^{(i)}}}{\sum_{j=1}^{m} e^{net_j{}^{(i)}}}\right)$$

Softmax output layer (Classification)

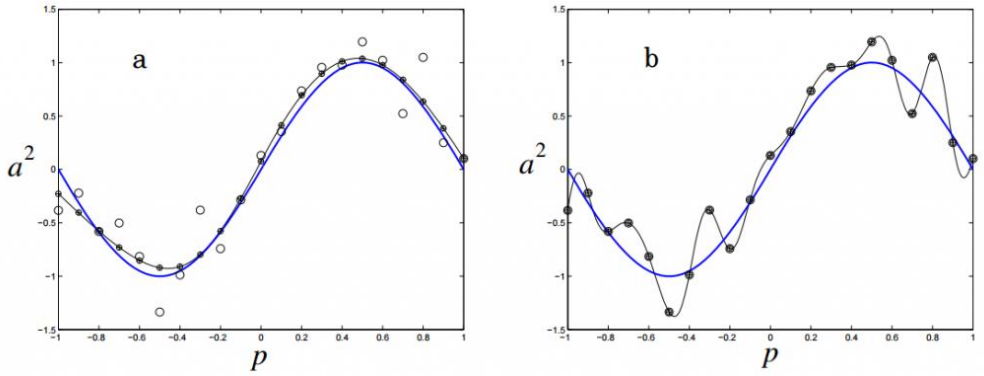


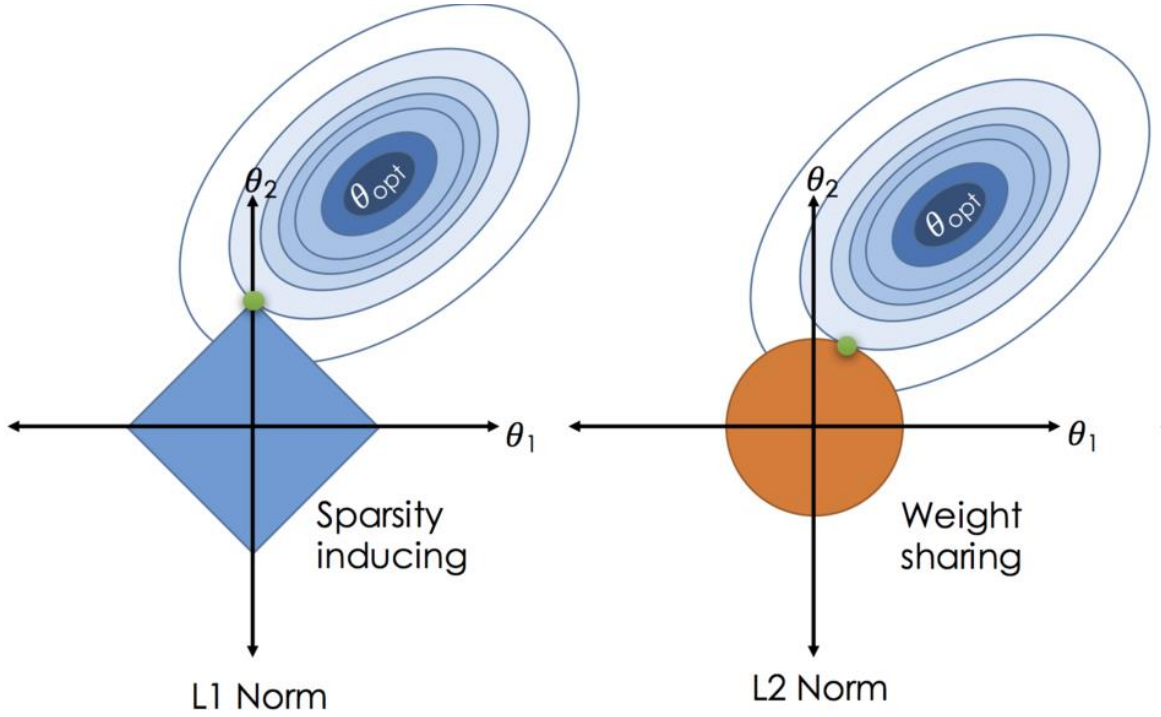
- Worst prediction
  $L_i = -\log(0) \rightarrow +\infty$
- Best prediction
  $L_i = -\log(1) = 0$

Slide taken from - Stanford, CS231n course, Lecture 3 presentation
(http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture3.pdf)

Regularization

- $\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \boldsymbol{\lambda R(W)}$
  - λ – regularization strength (hyperparameter)
- Commonly used
  - $L_2 - R(W) = \sum W^2$ (Weight decay)
  - $L_1 - R(W) = \sum |W|$





Gradient Descent Optimization

Stanford, CS231n course, Lecture 3 presentation

Gradient descent (GD)

- Technique for solving numerical optimization problems

- Algorithm:
  1) Initialize parameters ($\boldsymbol{W}$ – weights, $\boldsymbol{b}$ – biases)
  2) Repeat
     - Compute error gradient for each parameter respectively $\frac{\partial L}{\partial w_{ij}}$

       $\mathrm{i} = 1..m,\ j = 1..k$
     - Update parameters using gradients $w_{ij}(n+1) = w_{ij}(n) - \alpha \frac{\partial L}{\partial w_{ij}}$

- Parameter α is a **learning rate** and it should be chosen carefully!

Gradient Descent

- Gradient Descent

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(f(x_i, W), y_i) + \lambda \nabla_W R(W)$$

- (Mini-)Batch GD

$$\nabla_W L(W) = \frac{1}{N_{mb}}\sum_{i=1}^{N_{mb}} \nabla_W L_i(f(x_i, W), y_i) + \lambda \nabla_W R(W)$$

- Stochastic (incremental) GD (SGD)

$$\nabla_W L(W) = \nabla_W L_i(f(x_i, W), y_i) + \lambda \nabla_W R(W)$$

Gradient Descent Implementations

- Gradient Descent
  - Too slow for large datasets
  - We need to keep intermediate results for each datapoint forward pass
    - Memory problems
  - May be quickly stuck in local minimum

- Mini-batch GD
  - This is used in Deep Learning practice
  - Gradient approximation
  - May skip some local minima
  - N_{mb} is usually 32/64/128/256 (depends on data/model/working memory size)

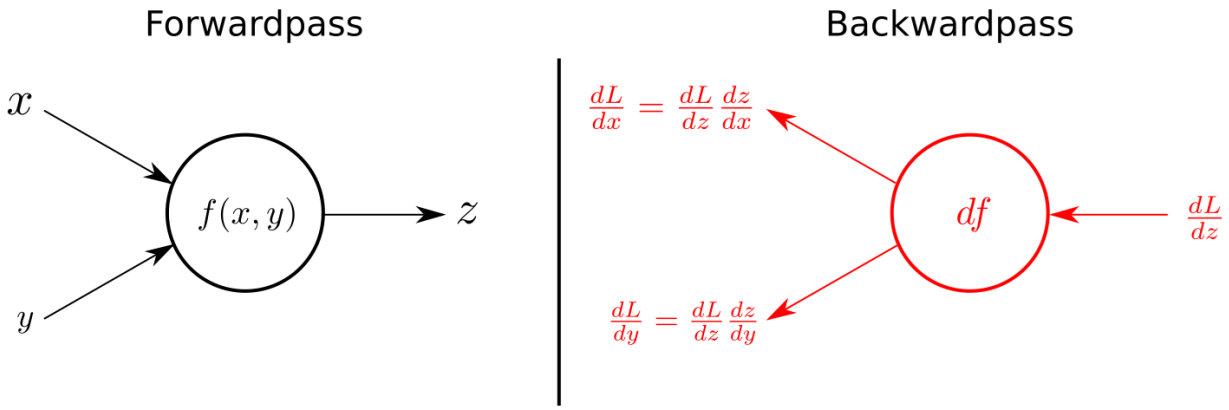
Chain rule

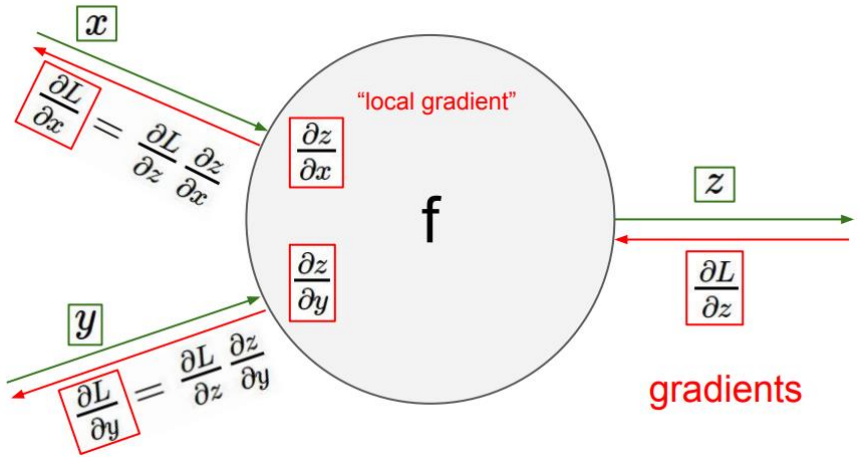
- Backpropagation is based on the chain rule:

$$h = f_1(z)$$
$$g = f_2(h)$$
$$y = f_3(g)$$
$$y = f_3\left(f_2\left(f_1(z)\right)\right)$$

$$\frac{\partial y}{\partial z} = \frac{\partial y}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial z}$$

Backpropagation





Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

$$E = \tfrac{1}{2}\sum_c \sum_j (y_{j,c} - d_{j,c})^2 \qquad (3)$$

The backward pass starts by computing $\partial E/\partial y$ for each of the output units. Differentiating equation (3) for a particular case, c, and suppressing the index c gives

$$\partial E/\partial y_j = y_j - d_j \qquad (4)$$

We can then apply the chain rule to compute $\partial E/\partial x_j$

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot \mathrm{d}y_j/\mathrm{d}x_j$$

Differentiating equation (2) to get the value of $\mathrm{d}y_j/\mathrm{d}x_j$ and substituting gives

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot y_j(1-y_j) \qquad (5)$$

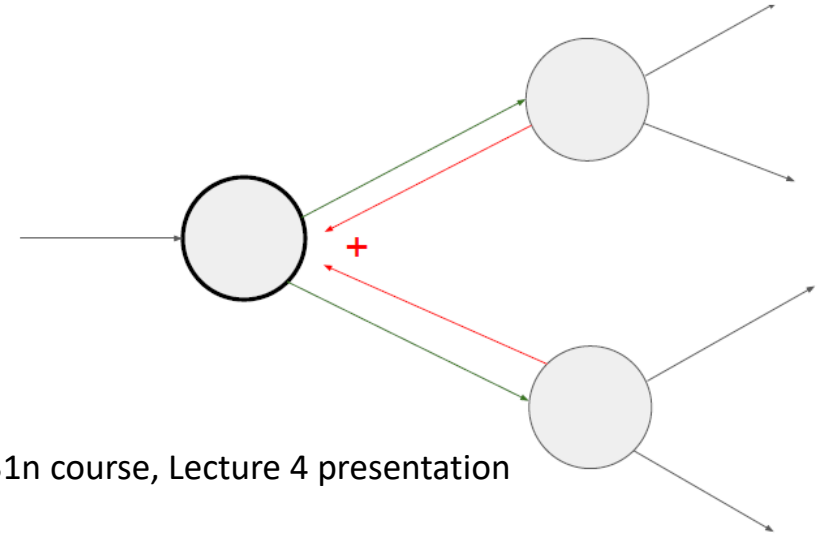
This means that we know how a change in the total input x to an output unit will affect the error. But this total input is just a linear function of the states of the lower level units and it is also a linear function of the weights on the connections, so it is easy to compute how the error will be affected by changing these states and weights. For a weight w_{ji}, from i to j the derivative is

$$\partial E/\partial w_{ji} = \partial E/\partial x_j \cdot \partial x_j/\partial w_{ji}$$
$$= \partial E/\partial x_j \cdot y_i \qquad (6)$$

Source: becominghuman.ai
Rumelhar et al. Nature 1986

Computational graphs

- Neuron with multiple outputs – gradients add



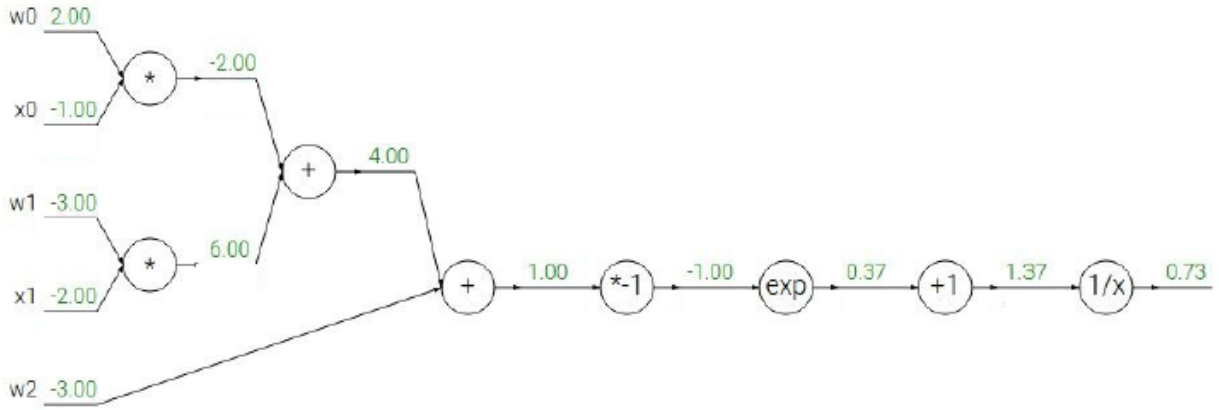
Stanford, CS231n course, Lecture 4 presentation

- Forward and backward passes in these graphs can be vectorized (Jacobian matrix instead of gradient)

Computational graphs

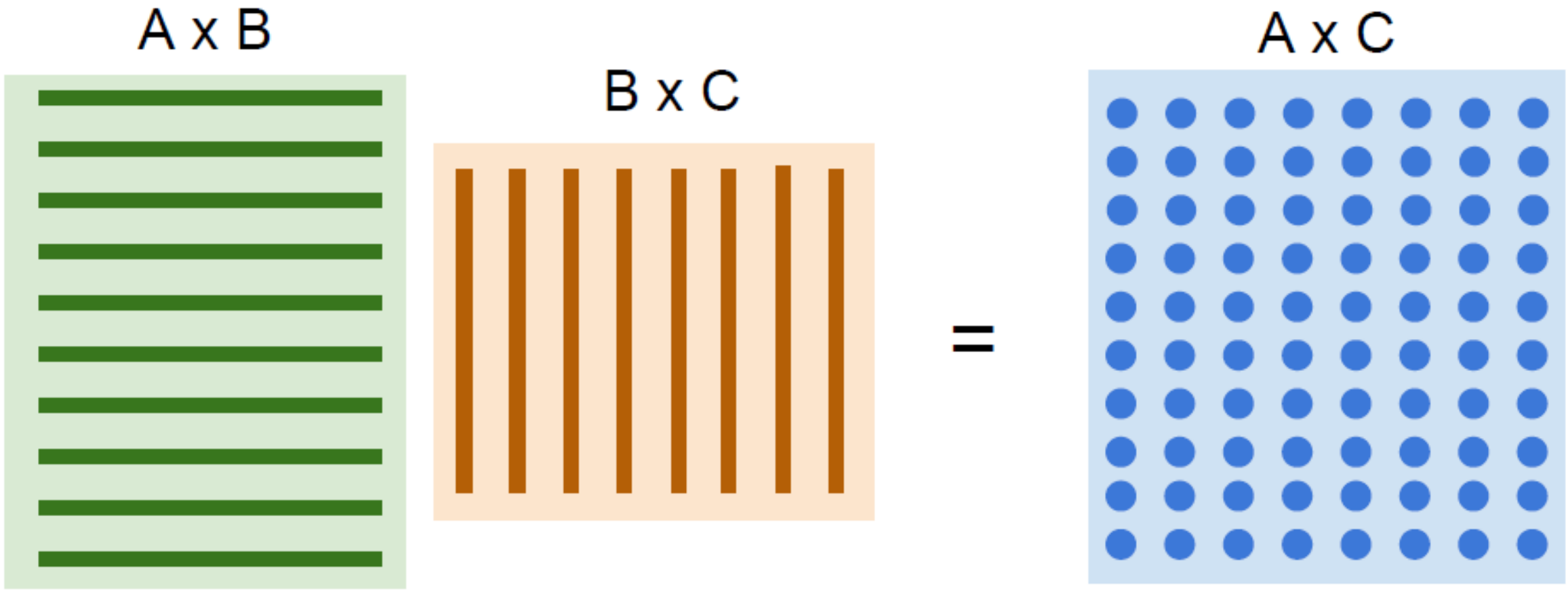
- ANN will be very large: impractical to calculate all gradient formulas by hand
- Implementation of DL algorithms comes down to implementing computational graphs and calculating their partial derivatives
  - This is called backpropagation

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



Stanford, CS231n course, Lecture 4 presentation

Matrix multiplications can be parallelized ⇒ GPUs



Stanford, CS231n course, Lecture 8 presentation

Common practices

- Usually we start with certain amount of labeled samples
  - $(x_i, y_i); i = 1, \dots, n$
- The dataset is divided into three partitions
  - Training set – used in training
  - Validation set – used to tune hyperparameters
  - Test set – used to measure quality of the network
- Neural network are “data” hungry
- Randomization
  - Present samples in random order

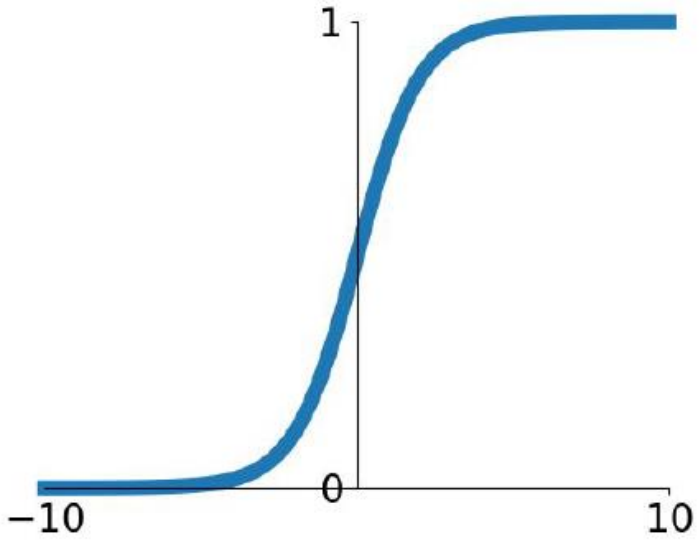
Some terminology

- **Forward pass:** Calculate the network outputs y_i based on the inputs x_i
- **Backward pass:** Calculate gradients $\frac{\partial L}{\partial w_{ij}}$, by using the chain rule and intermediate calculations from the forward pass
- **Backpropagation:** Update network weights w_{ij} by using the calculated gradients $\frac{\partial L}{\partial w_{ij}}$
- **Mini-batch:** examples whose gradients are averaged before backpropagation
- **Epoch:** The ENTIRE dataset is passed ONCE forwards and backwards
  - A complete training goes through multiple epoch

You can train an ANN now


DEEP LEARNING
WHY YOU NO WORK?!

Activation Functions - Sigmoid

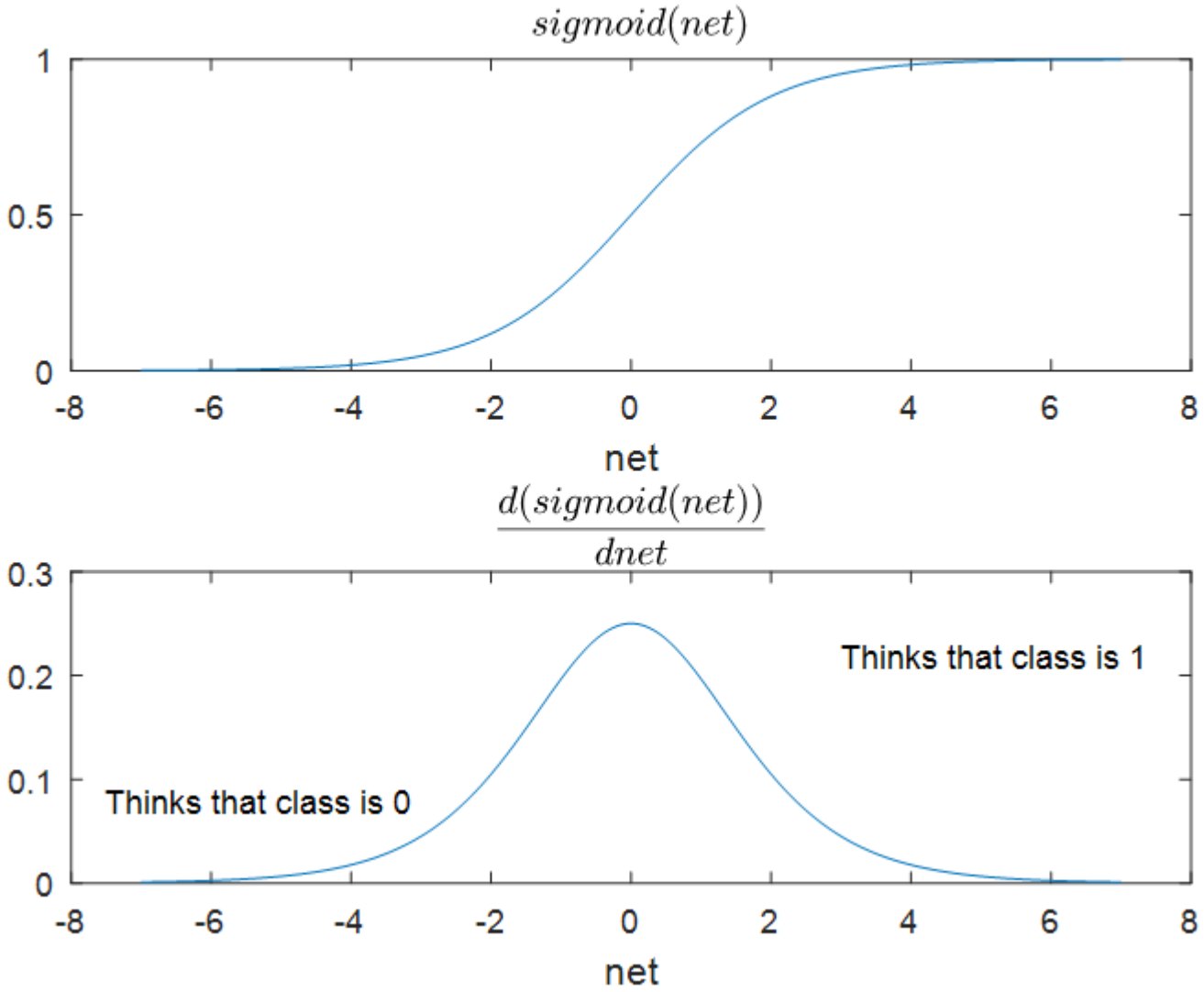


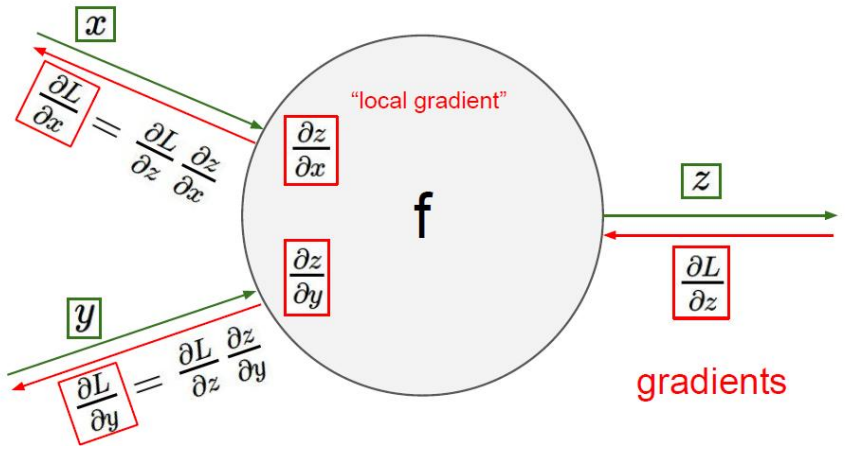
Stanford, CS231n course, Lecture 6 presentation

- Saturation “kills” the gradients ✗

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

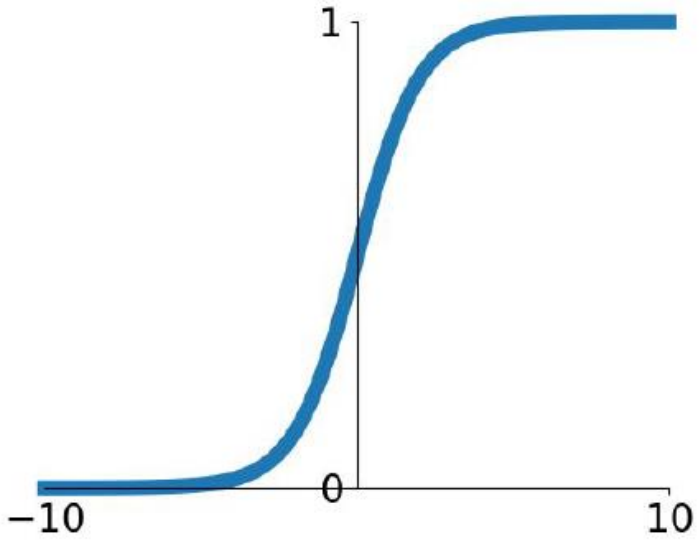
Activation Functions - Sigmoid





Stanford, CS231n course, Lecture 4 presentation

Activation Functions - Sigmoid



Stanford, CS231n course, Lecture 6 presentation

- Saturation “kills” the gradients ✗
- Sigmoid always outputs a positive number ✗

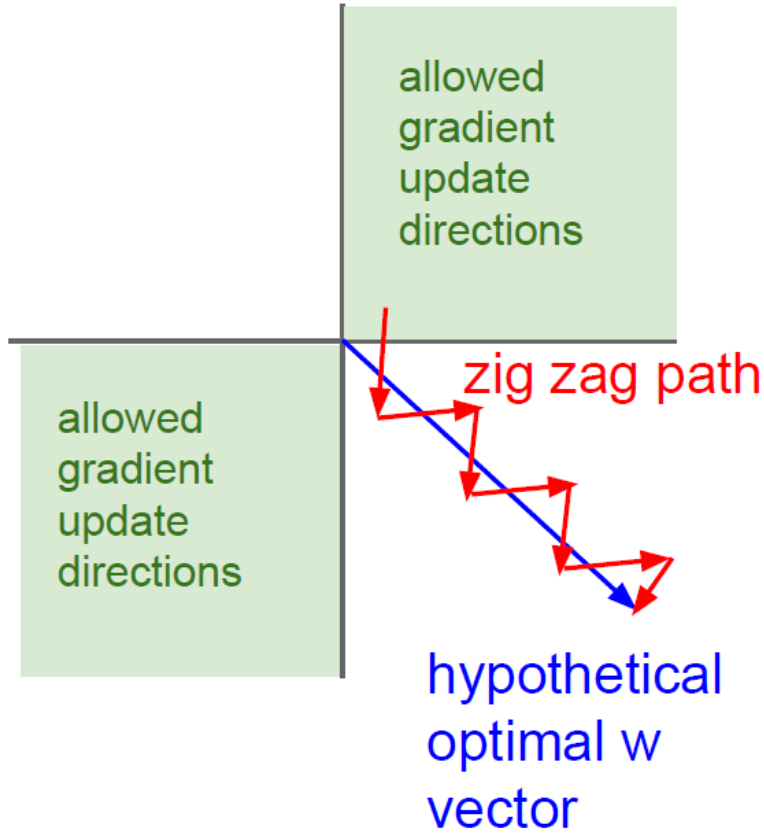
$$\sigma(net) = \frac{1}{1 + e^{-net}}$$

Activation Functions - Sigmoid

- $\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_i}$
  - $z = \sum_i x_i\, w_i + b$ (pre-activation)

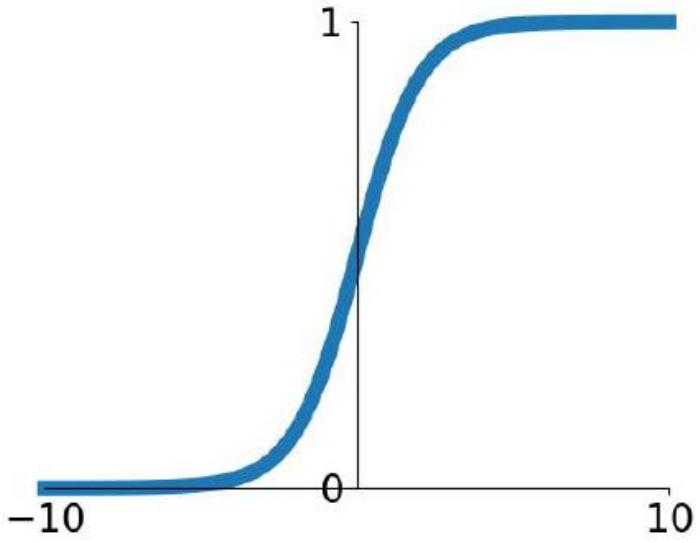
1. When inputs x_i are always positive
   - $\Rightarrow \frac{\partial z}{\partial w_i} = x_i$ is always positive
2. Each w_i uses the same gradient $\frac{\partial L}{\partial z}$

$\Rightarrow$ Gradient update rule is positive/negative for all w_i



Stanford, CS231n course, Lecture 6 presentation

Activation Functions - Sigmoid

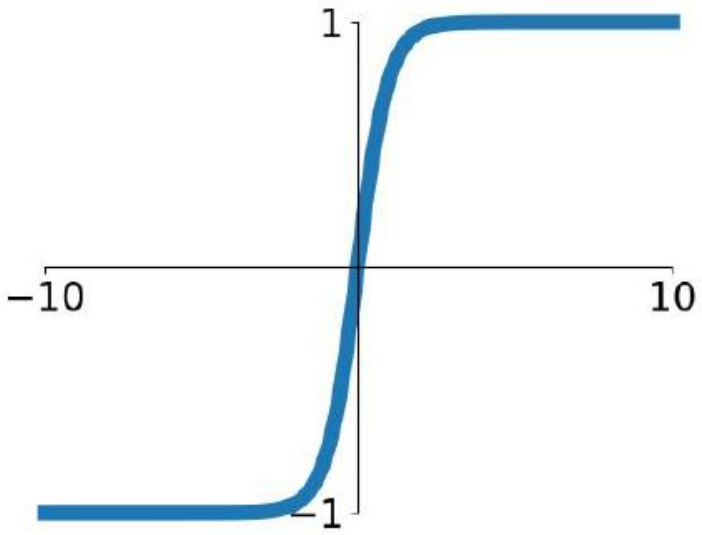


Stanford, CS231n course, Lecture 6 presentation

- Saturation “kills” the gradients ✗
- Sigmoid always outputs a positive number ✗
- exp( ) is a bit computationally expensive ✗
  - Not a big problem

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

Activation Functions – Hyperbolic Tangent



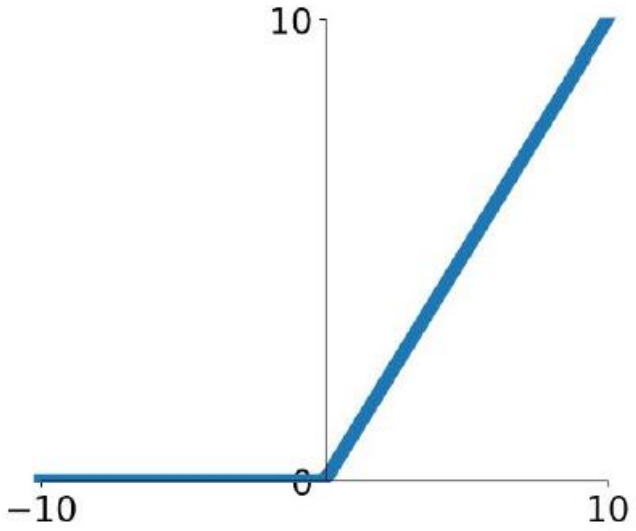
Stanford, CS231n course, Lecture 6 presentation

$$\sigma(net) = \tanh(net)$$

- Zero centered ✓
- Saturation “kills” the gradients ✗
- A bit computationally expensive ✗

  (not a big problem)

Activation Functions – ReLU



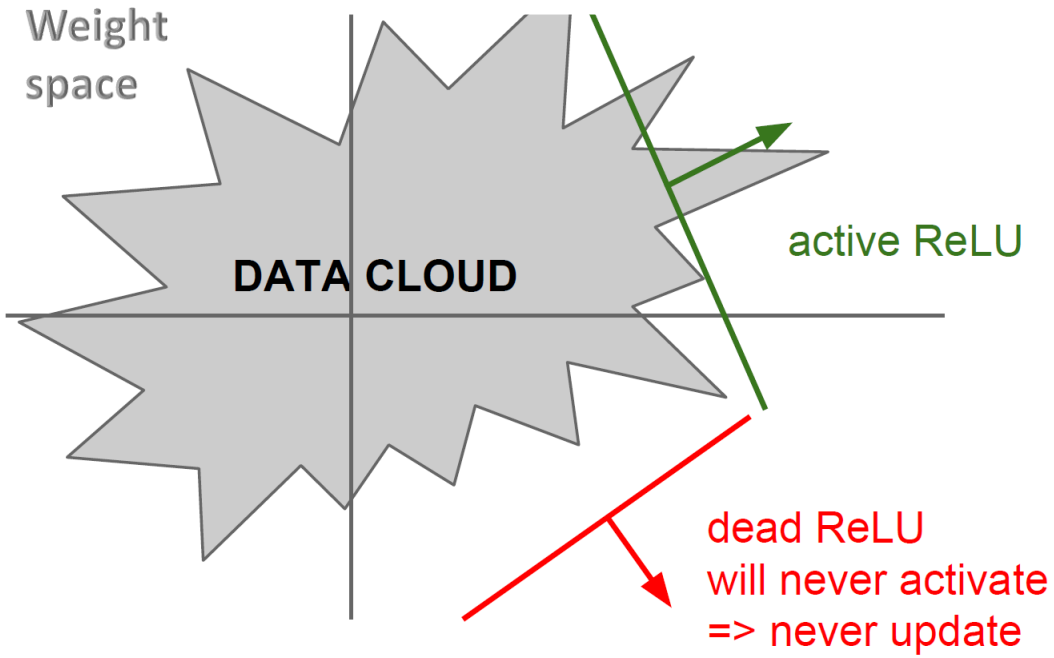
Stanford, CS231n course, Lecture 6 presentation

$$\sigma(net) = \max(0, net)$$

- Doesn't saturate in + region ✓
- Computationally efficient ✓
- Converges much faster than sigmoid/tanh in practice ✓

- Output not zero-centered ✗
- Gradient is 0 in – region ✗ (saturation)

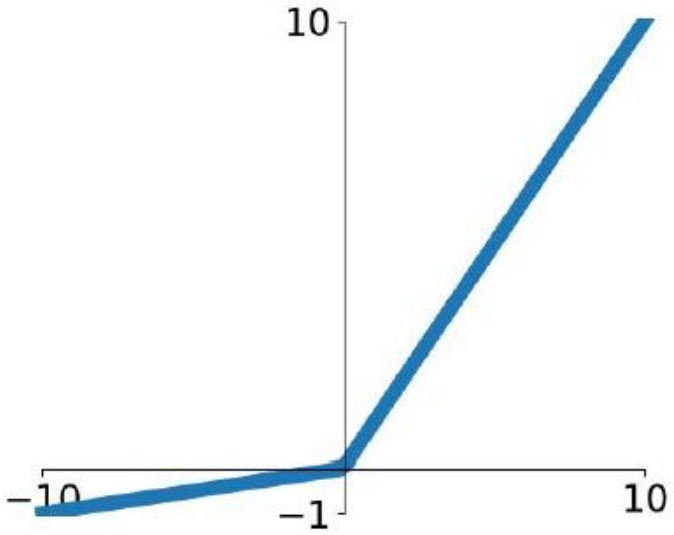
Activation Functions – ReLU



Stanford, CS231n course, Lecture 6 presentation

- If datapoints don't activate the ReLU, it won't update weights in a BP step
  - ReLUs with bad initialized weights will stay dead
- Learning rate too high
  - ReLU could fall into the dead ReLU region
- Initializing ReLU biases with small positive numbers often helps

Activation Functions – Leaky ReLU



Stanford, CS231n course, Lecture 6 presentation

$$\sigma(net) = \max(0.01net, net)$$

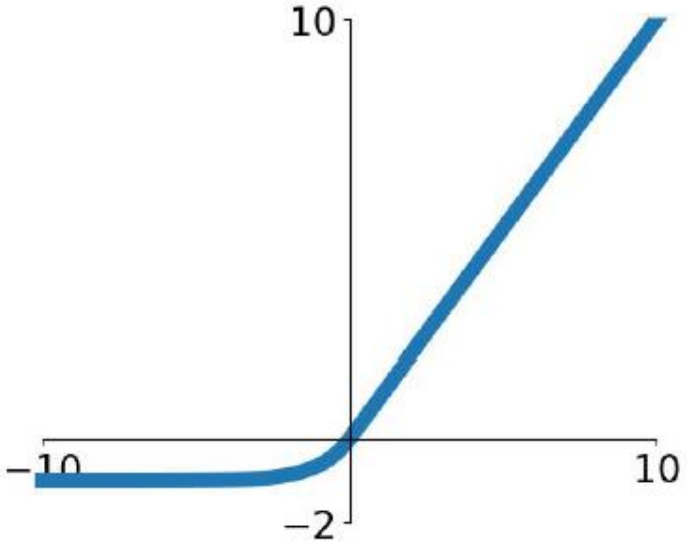
- Doesn’t saturate in + region ✓
- Computationally efficient ✓
- Converges much faster than sigmoid/tanh in practice ✓
- Doesn’t saturate in - region ✓

  (will not die)

- Slope in the – region could be parametrized

  (learn it with backprop)

Activation Functions – ELU



Stanford, CS231n course, Lecture 6 presentation

- Similar to leaky ReLU ✓
- Saturation in the – region adds some robustness to noise compared to leaky ReLU ✓

- exp( ) is a bit computationally expensive ✗

$$\sigma(net) = \begin{cases} net & net > 0 \\ \alpha(e^{net} - 1) & net \leq 0 \end{cases}$$

Activation Functions – Maxout

$$\sigma(net) = (\mathrm{w}_1^{\mathrm{T}}\mathrm{x} + \mathrm{b}_1, \mathrm{w}_2^{\mathrm{T}}\mathrm{x} + \mathrm{b}_2)$$

- Generelizes ReLU and Leaky ReLU ✓
- Doesn’t saturate ✓
- Doesn’t die ✓

- Twice as many parameters as ReLU ✗

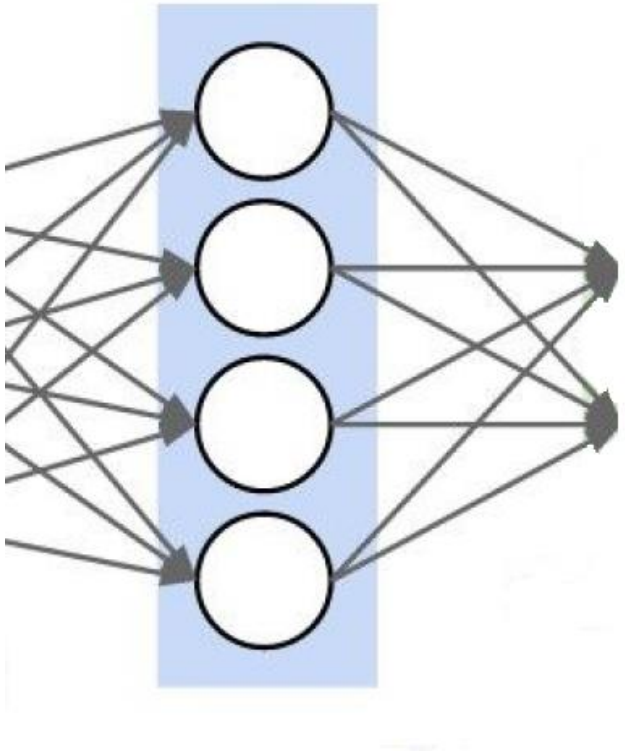
Activation Functions – Practical advice

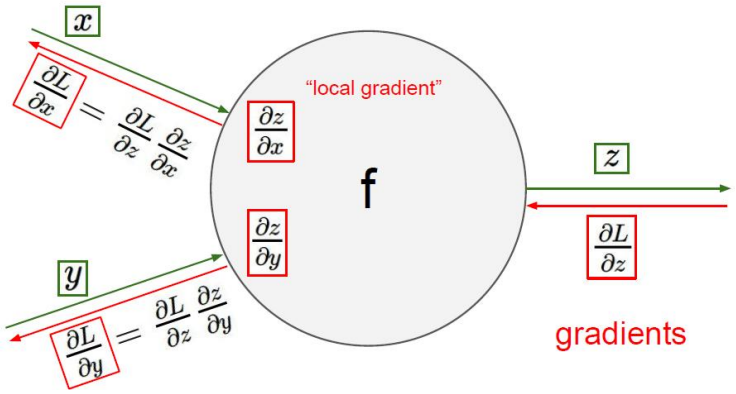
- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid

Slide taken from - Stanford, CS231n course, Lecture 6 presentation
http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture06.pdf

Weight Initialization – W=0

- What will happen if we initialize all weights to 0?

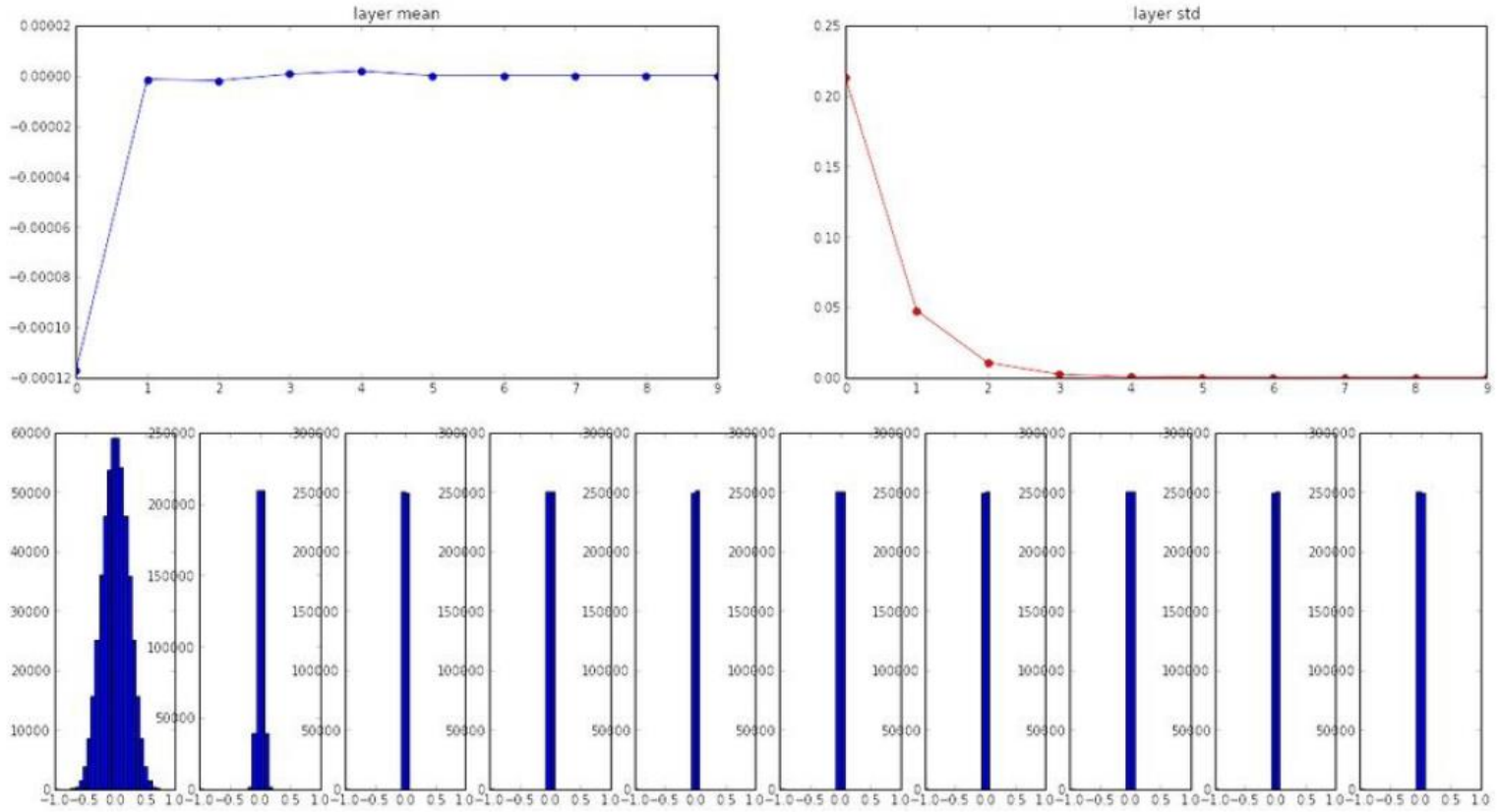




Stanford, CS231n course, Lecture 4 presentation

Weight Initialization – Small Random Values

Tanh activations

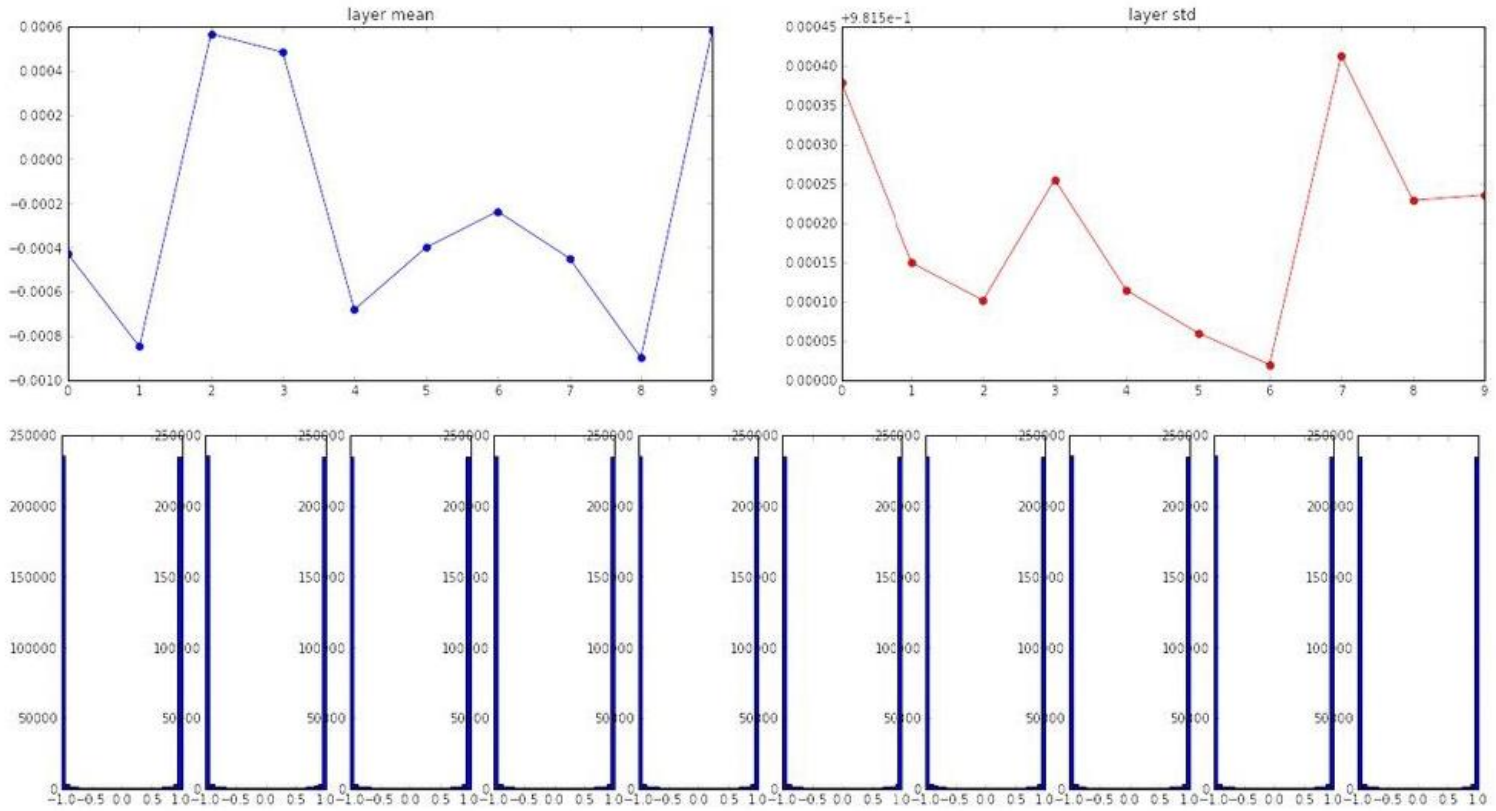


- Deeper into the network activations become zero
- During backpropagation each layer is multiplied by w
  - Slow learning

Stanford, CS231n course, Lecture 6 presentation

Weight Initialization – Big Random Values

Tanh activations

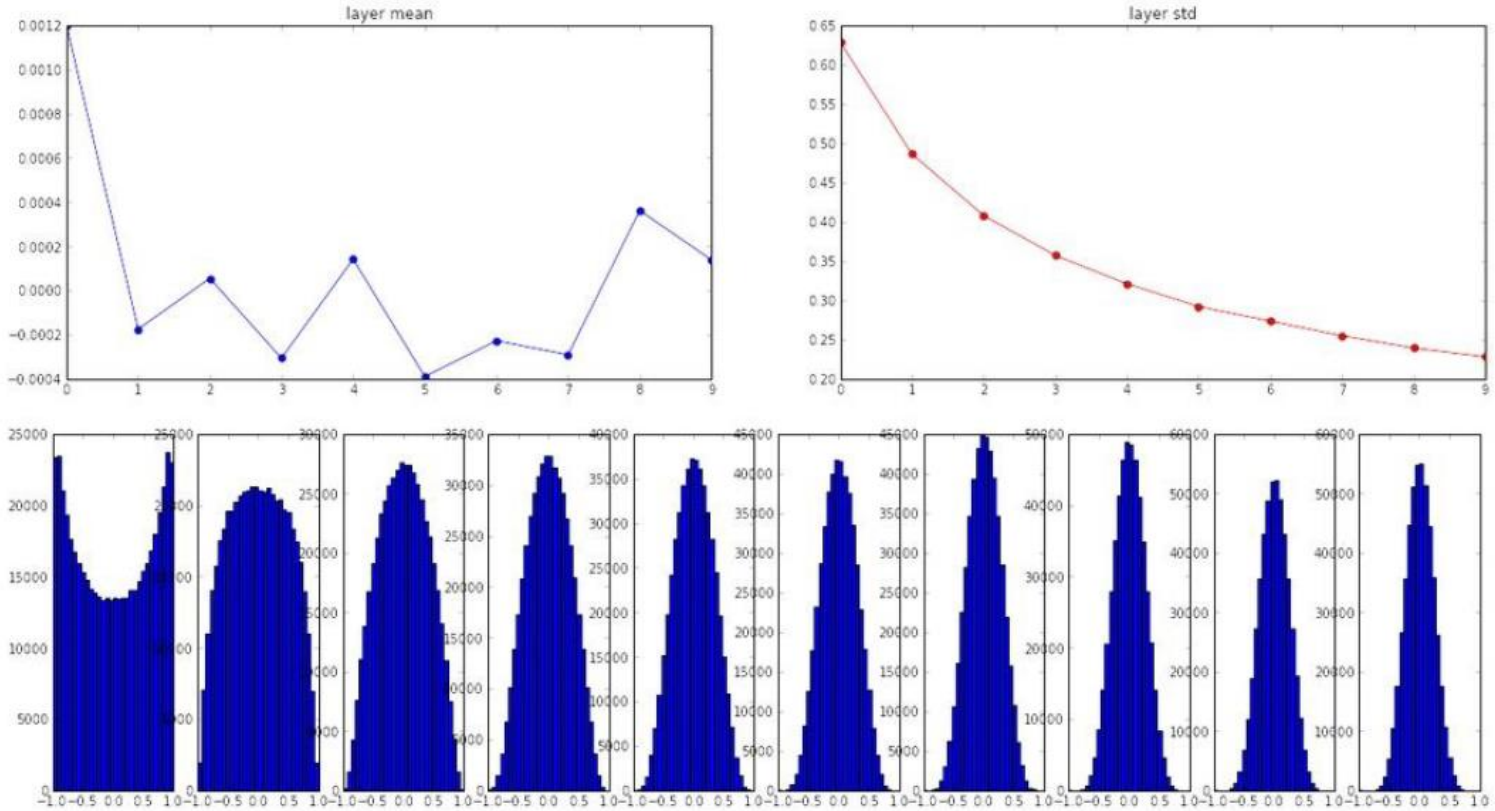


- Most of the activations go into saturation.
- Gradients will be small because of the saturations
  - Slow learning

Stanford, CS231n course, Lecture 6 presentation

Weight Initialization – Xavier

Tanh activations



Stanford, CS231n course, Lecture 6 presentation

- Glorot et al., 2010
  - Wants the same variance at input & output of a layer
- Layer initialization:
  - Typically Gaussian or Uniform

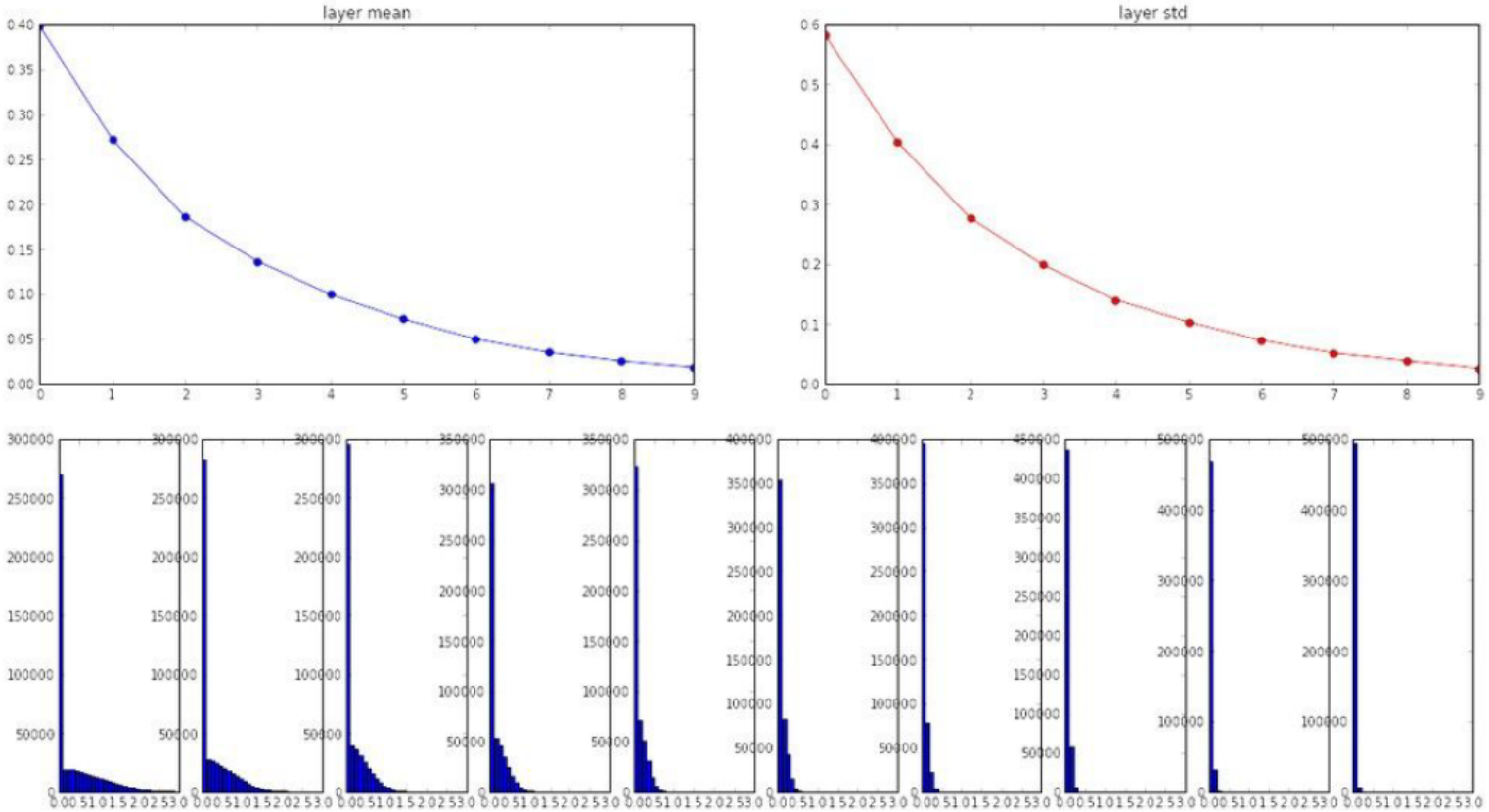
$$\mu(W)=0$$

$$\sigma(W) = \frac{2}{n_{in} + n_{out}}$$

  - n_{in} – number of units in the previous layer
  - n_{out} – number of units in the next layer

Weight Initialization – Xavier

ReLU activations

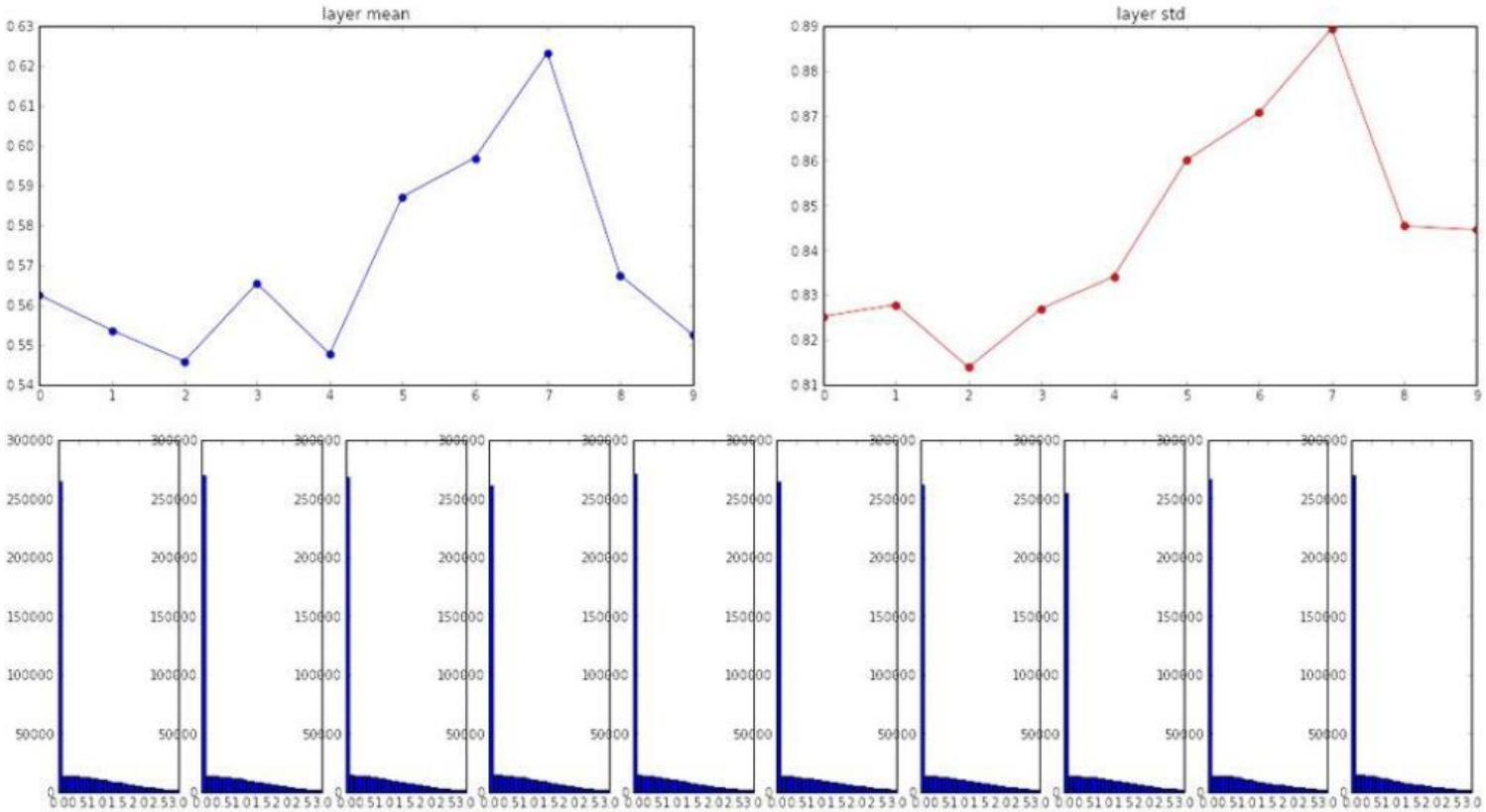


Stanford, CS231n course, Lecture 6 presentation

- Doesn’t work well on ReLU

Weight Initialization – Recent Recommendation

ReLU activations



Stanford, CS231n course, Lecture 6 presentation

- He et al., 2015
- Layer initialization:
  - Typically Gaussian or Uniform

$$\mu(W)=0$$

$$\sigma(W) = \frac{2}{n_{in}}$$

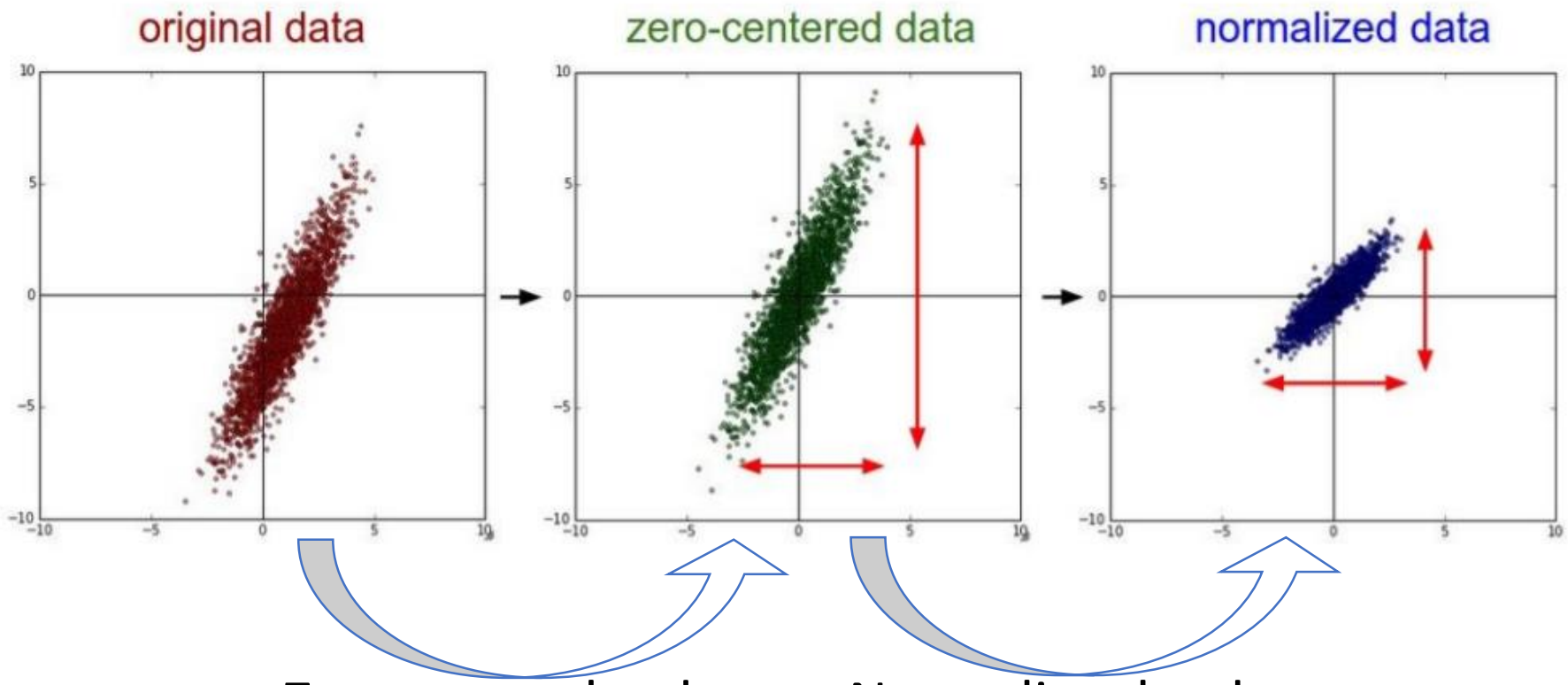
  - n_{in} – number of units in the previous layer

Weight Initialization

- Having a good initialization scheme is important in practice
- Some networks couldn’t be trained at all without a good initialization
- Still an open area of research

Data preprocessing

Stanford, CS231n course, Lecture 6 presentation



All features are in the same range ⇒ they all contribute equally

In practice

Training phase:

- Subtract the mean for each image
- Normalize with standard deviation

Testing phase:

- Subtract empirical mean (obtained from training data)

Only helps for the first layer!

Batch normalization

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

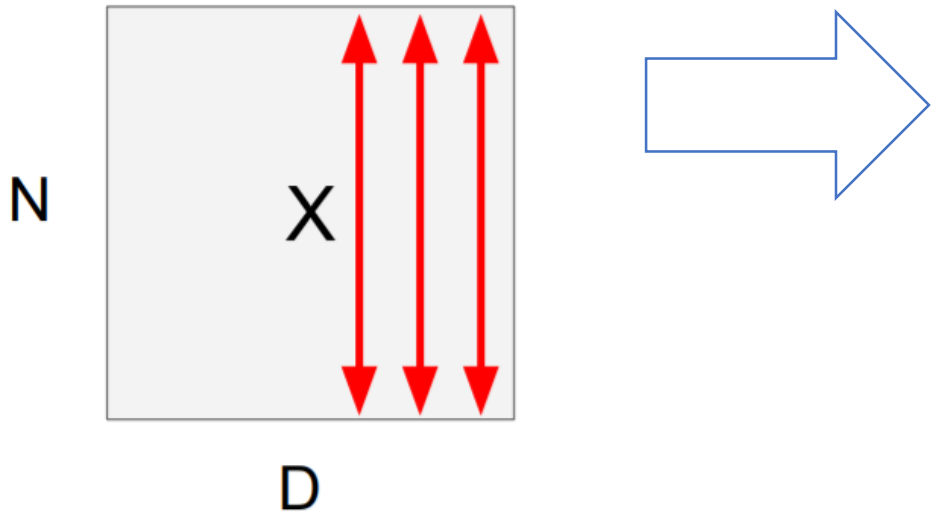
$$y^{(k)} = \gamma^{(k)}\, x^{(k)} + \beta^{(k)}$$

- β & γ are hyperparameters that can change during training
- Network can learn to undo the normalization of not suitable

$$\gamma^{(k)} = \mathrm{var}\{x^{(k)}\}$$

$$\beta^{(k)} = \mathrm{E}\{x^{(k)}\}$$

Batch normalization



Compute empirical mean and variance independently for each dimension

$x^{(k)}$ – current batch

N – training examples in the batch

D – dimension of each batch

$\mathbf{x}: \mathbf{N} \times \mathbf{D}$

Normalize

$\boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times \mathbf{D}$

$\gamma, \beta: 1 \times \mathbf{D}$

$\mathbf{y} = \gamma(\mathbf{x}-\boldsymbol{\mu})/\boldsymbol{\sigma}+\beta$

Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

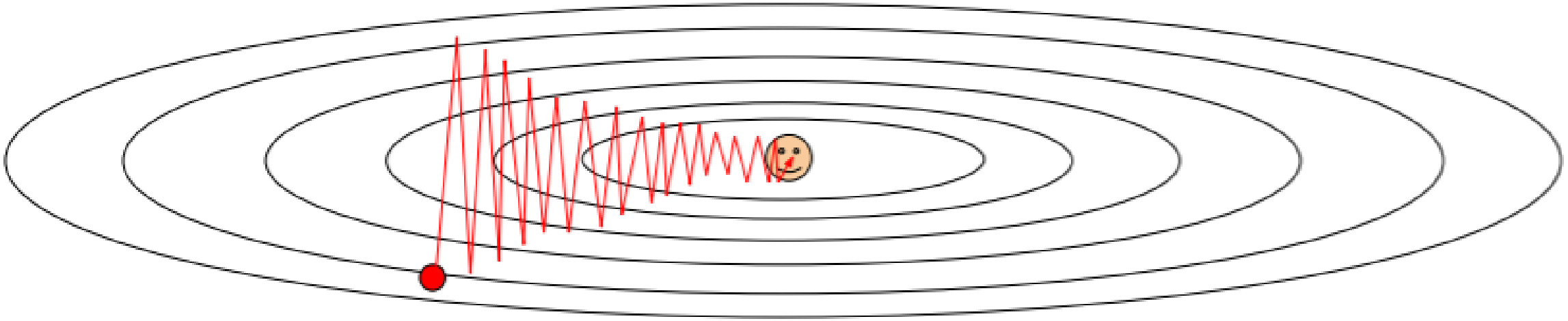
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

Stanford, CS231n course, Lecture 6 presentation

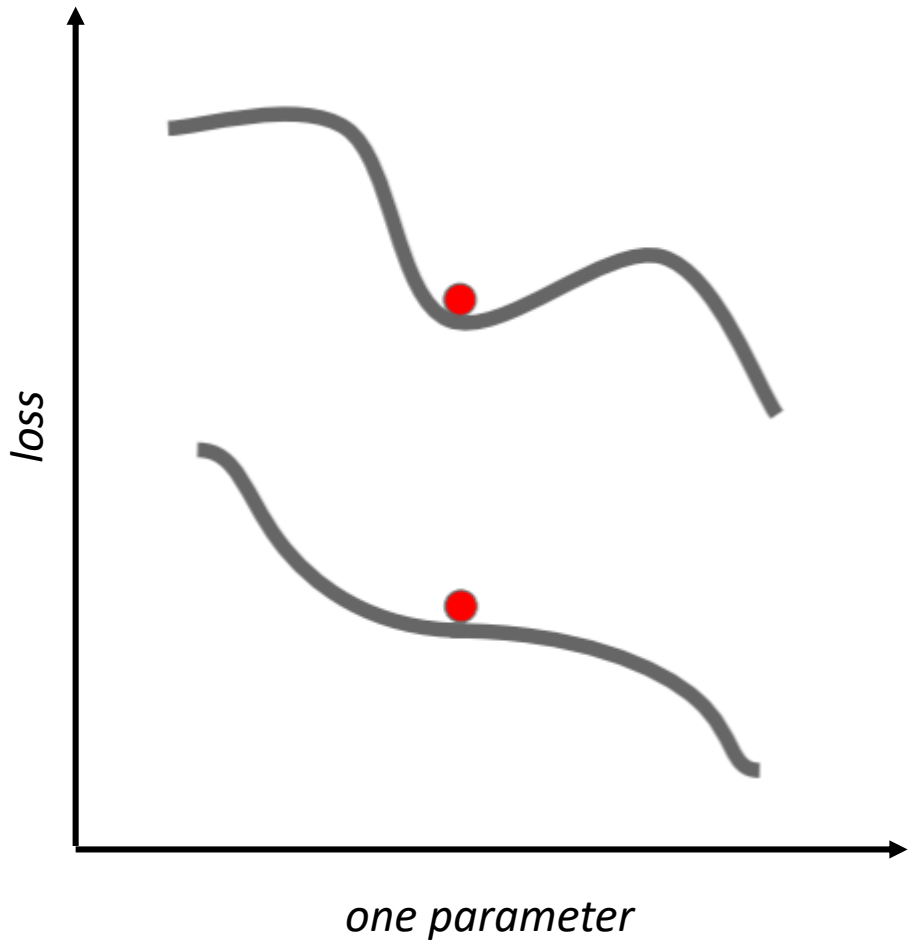
- More robust to:
  - Higher learning rates
  - Different kinds of weight init.
- Improves gradient flow
  - Easier to train
- Some kind of regularization
- It is not the same during training and testing

1° problem with SGD



- Different sensitivity to different dimensions
  - Slow progress along shallow dimension
  - Zigzag behavior along steep dimension

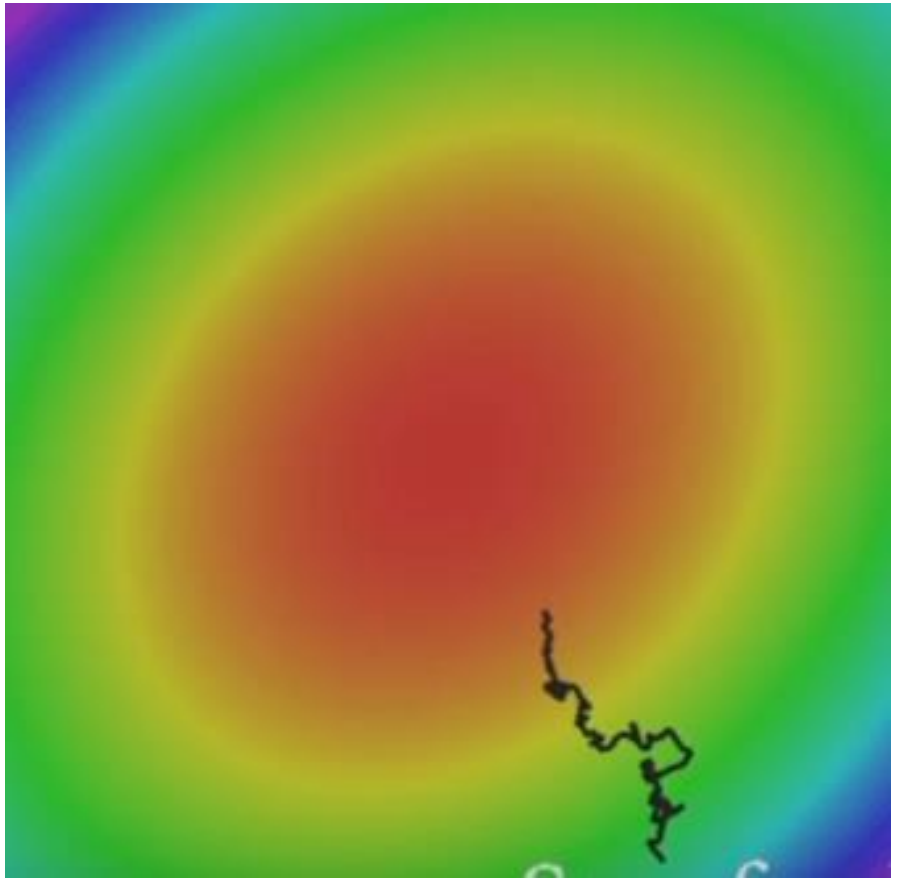
2° problem with SGD



Stanford, CS231n course, Lecture 7 presentation

- Can get stuck in local minima
- Local minima is not a big problem for high dimensions
- Saddle point is more common problem
  - Gradient is very small

3° Problem with SGD



Stanford, CS231n course, Lecture 7 presentation

- We usually use minibatches
  - We are only getting the noisy estimation of the gradient in the current minibatch

Momentum

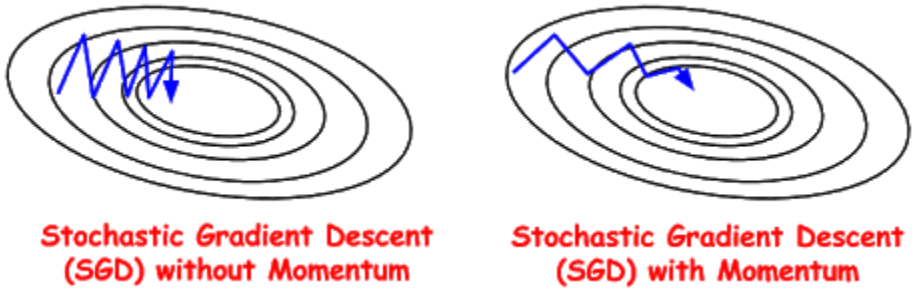
SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

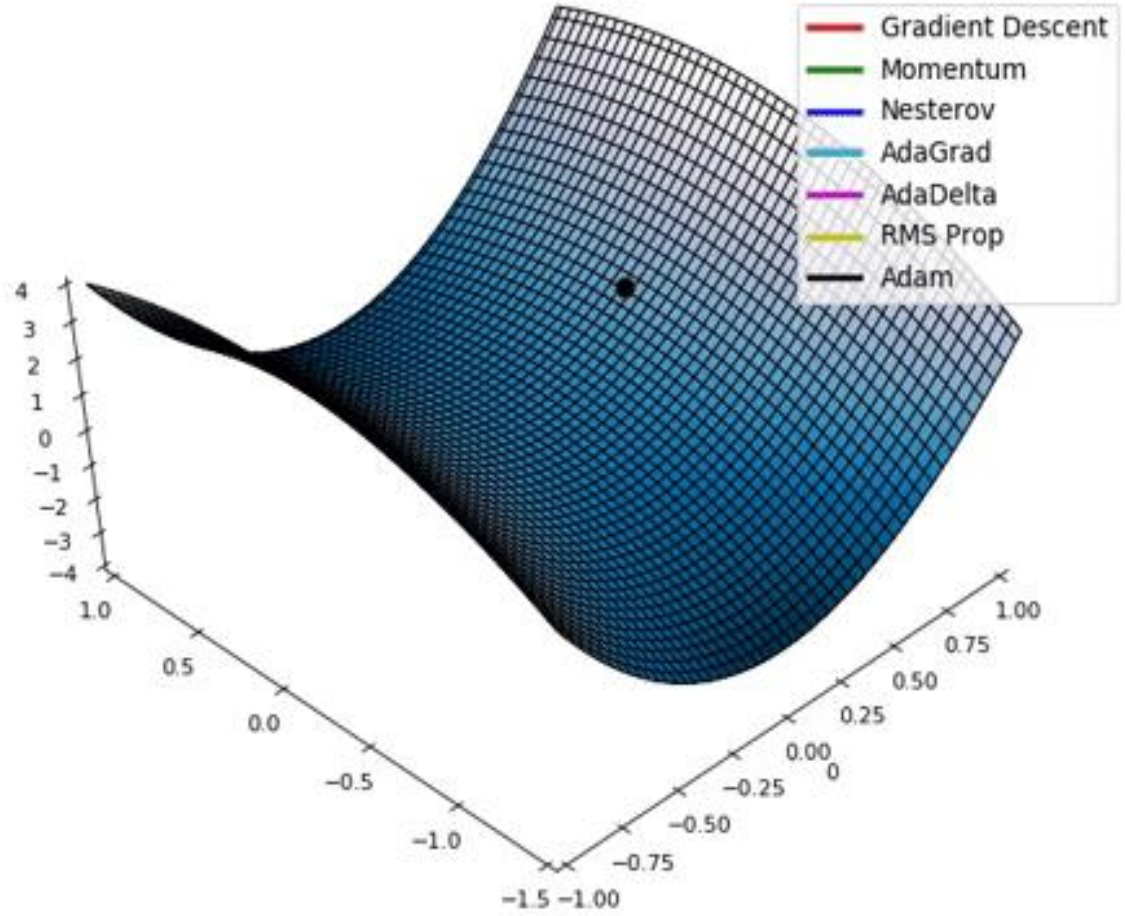
SGD + Momentum

$$v_{t+1} = \rho v_t + \alpha \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

- SGD – always stepping in the direction of the gradient
- SGD + Momentum – stepping in the direction of velocity

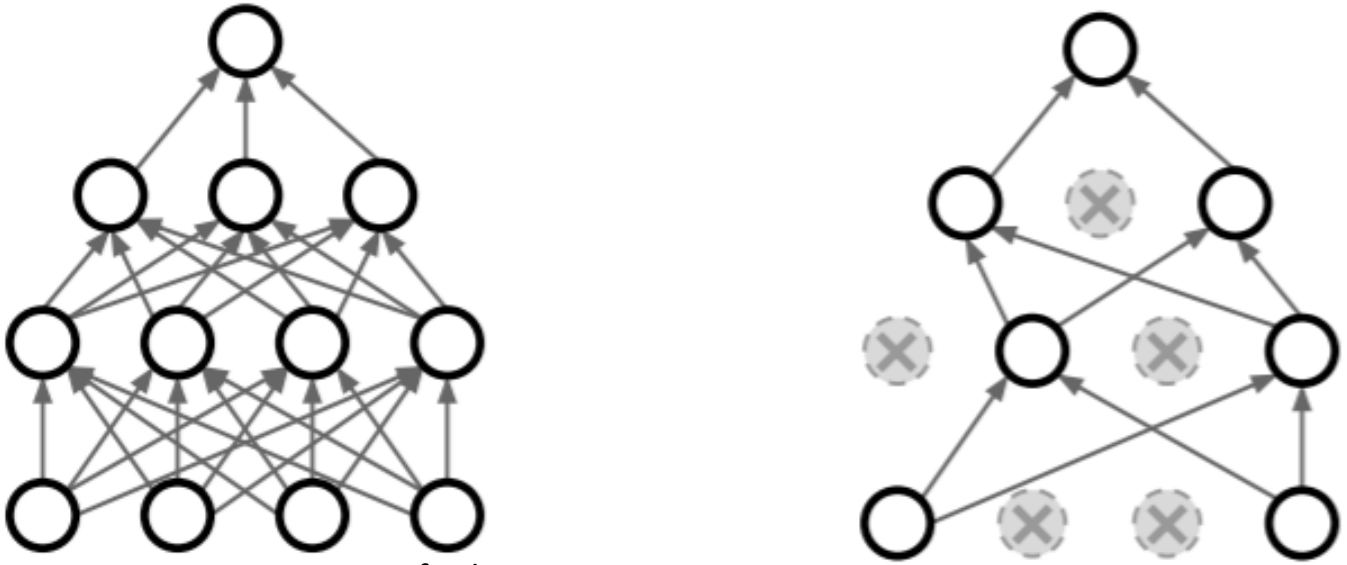


More sophisticated options



Regularization - dropout

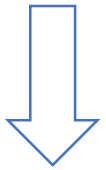
- In each pass through the NN we randomly set some of the neurons to zero, with probability p one layer at the time
- Dropping probability p is a hyperparameter
- Each dropout gives us a different subset of the NN
  - Dropout is like learning whole ensemble of networks at the same time



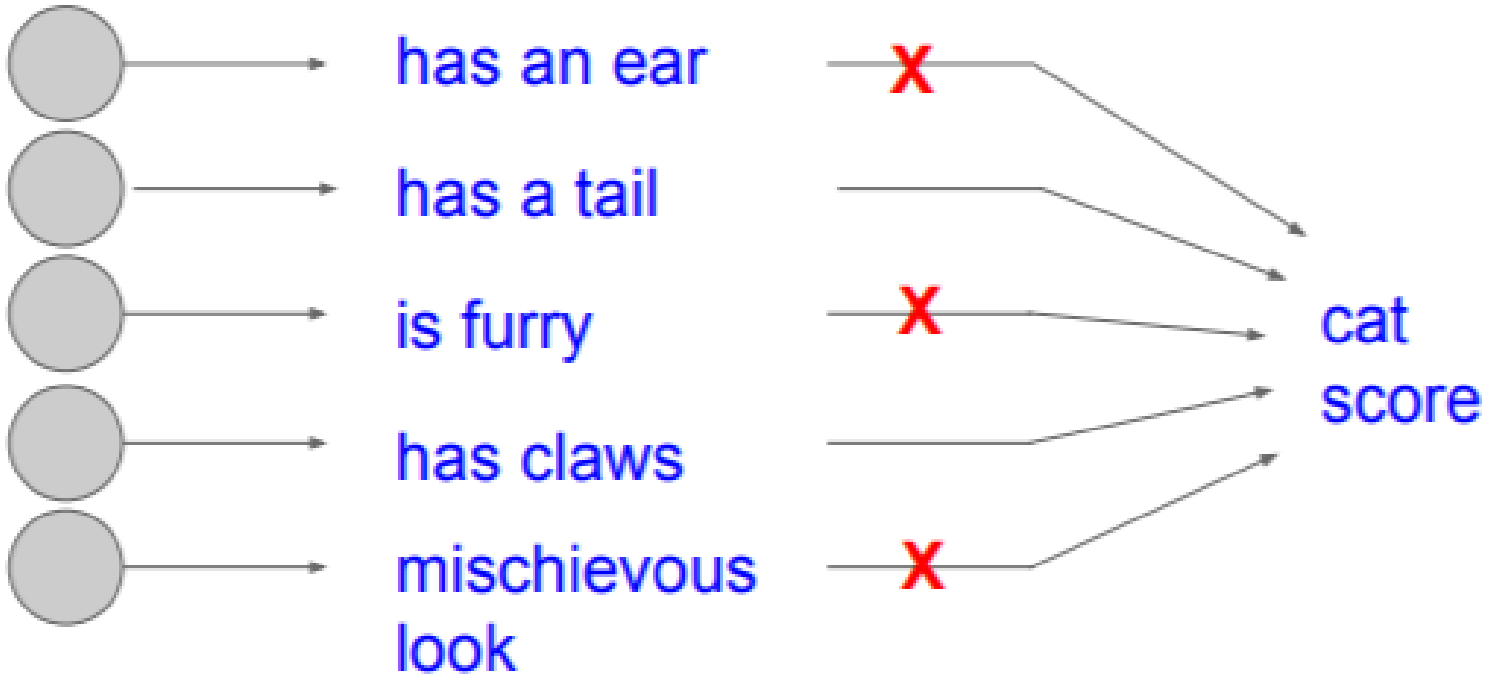
Stanford, CS231n course, Lecture 7 presentation

Dropout – why is it good?

- Helps to prevent co-adaptation of features
- The algorithm doesn't depend to much on one feature



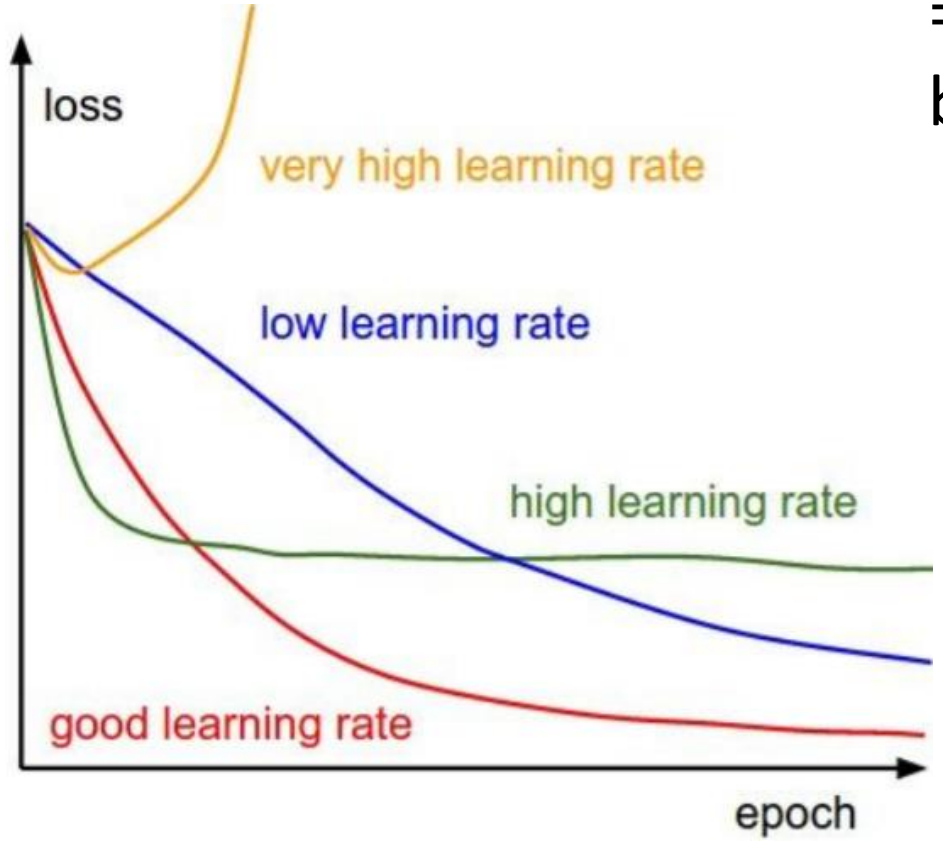
- Can help in overfitting!



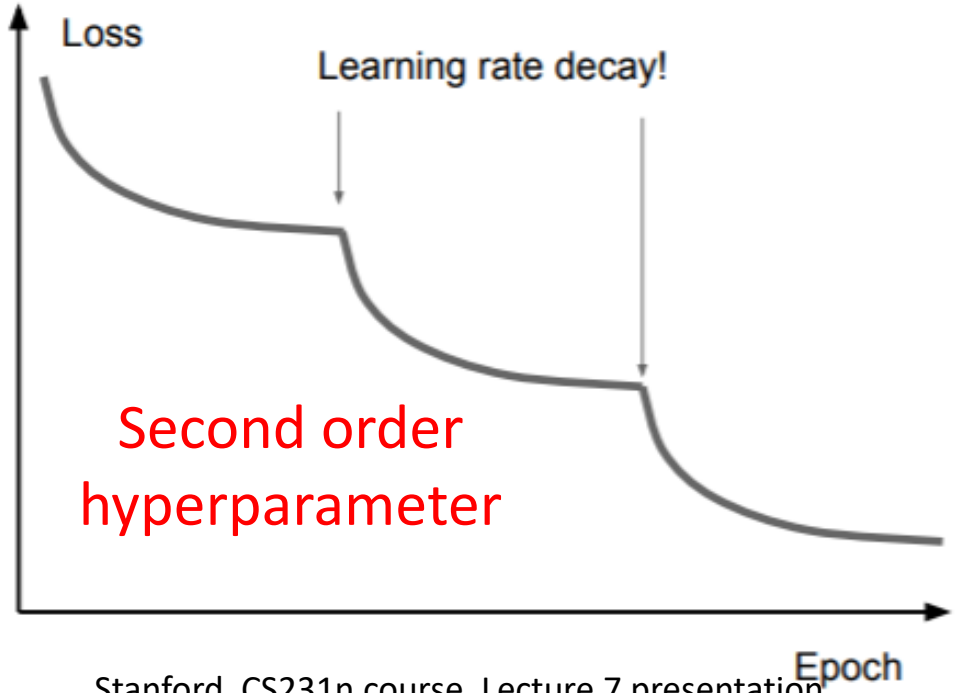
Stanford, CS231n course, Lecture 7 presentation

Learning rate adjusting

- SGD, SGD + Momentum, AdaGrad, RMSProp and Adam have learning rate as hyperparameter

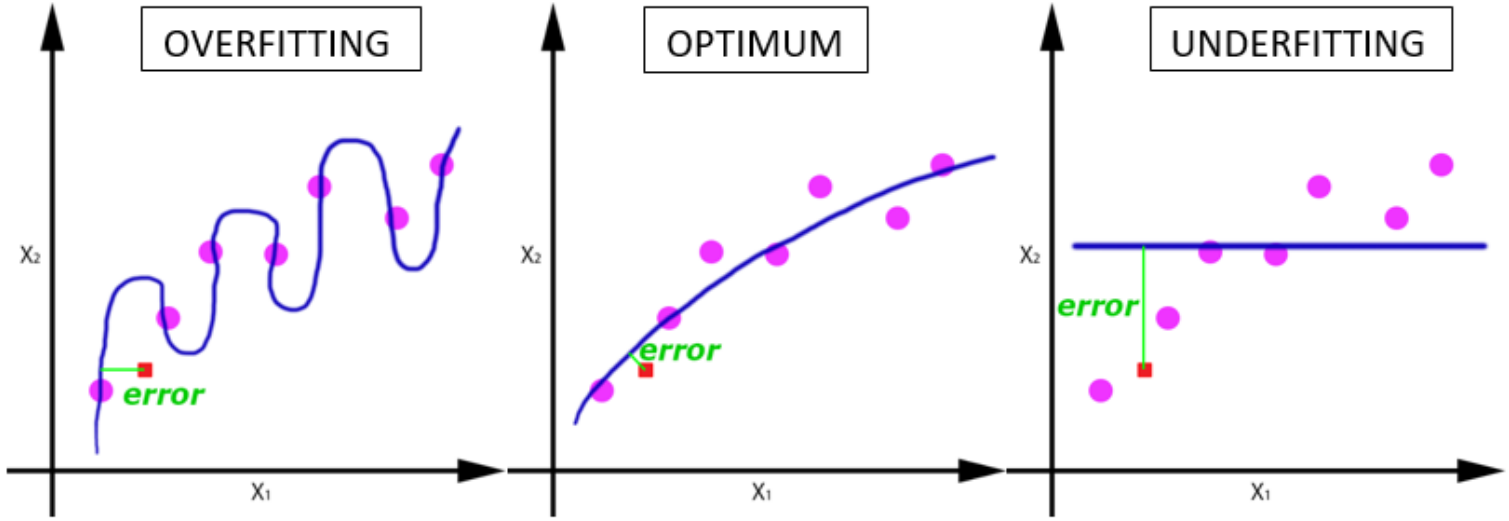


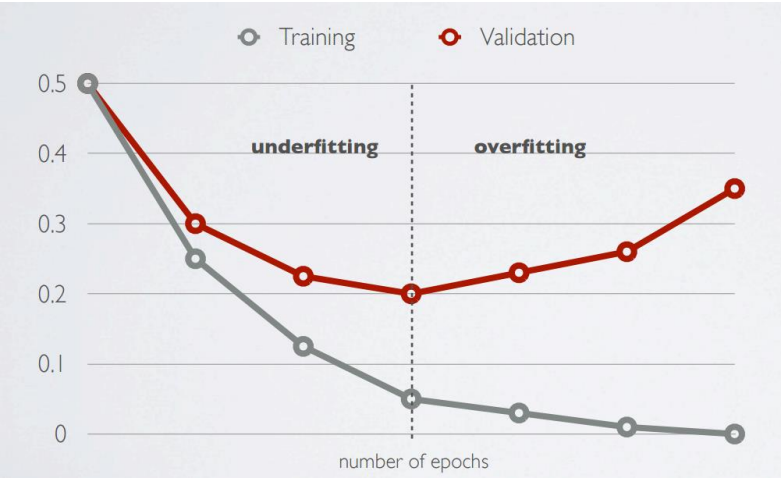
⇒ Learning rate decay over time: start with bigger value, and decrease it over time



Stanford, CS231n course, Lecture 7 presentation

What is Overfitting?





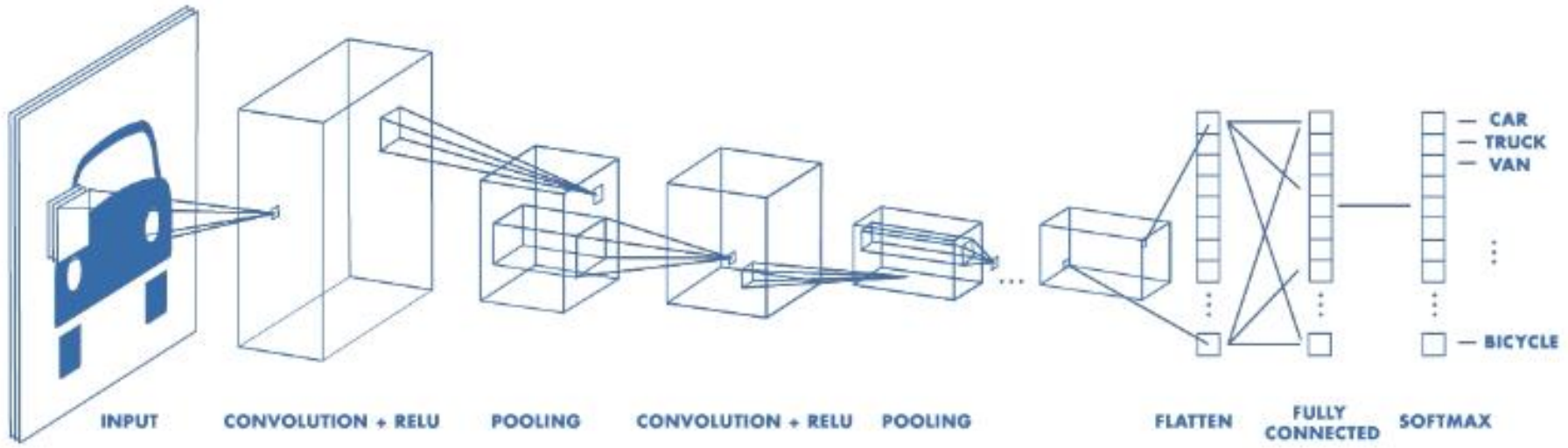
Fighting overfitting

- Smaller network
- Weight decay
- Dropout
- Early stopping
- More data
  - Data augmentation

Source: https://towardsdatascience.com/

Popular Deep Learning models

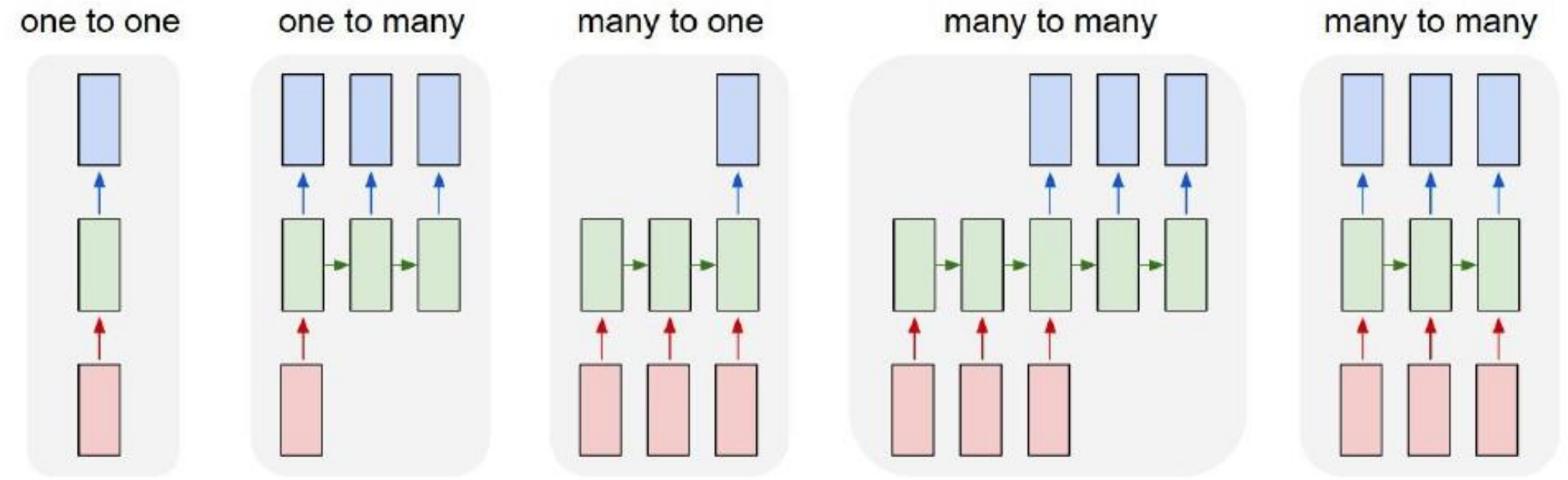
- **Convolutional Neural Networks (CNN)** – process Image-like data



https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765771.html

Popular Deep Learning models

- **Recurrent Neural Networks (RNN)** –process sequences



Feedforward NN

e.g. Image Captioning

e.g. Sentiment Classification

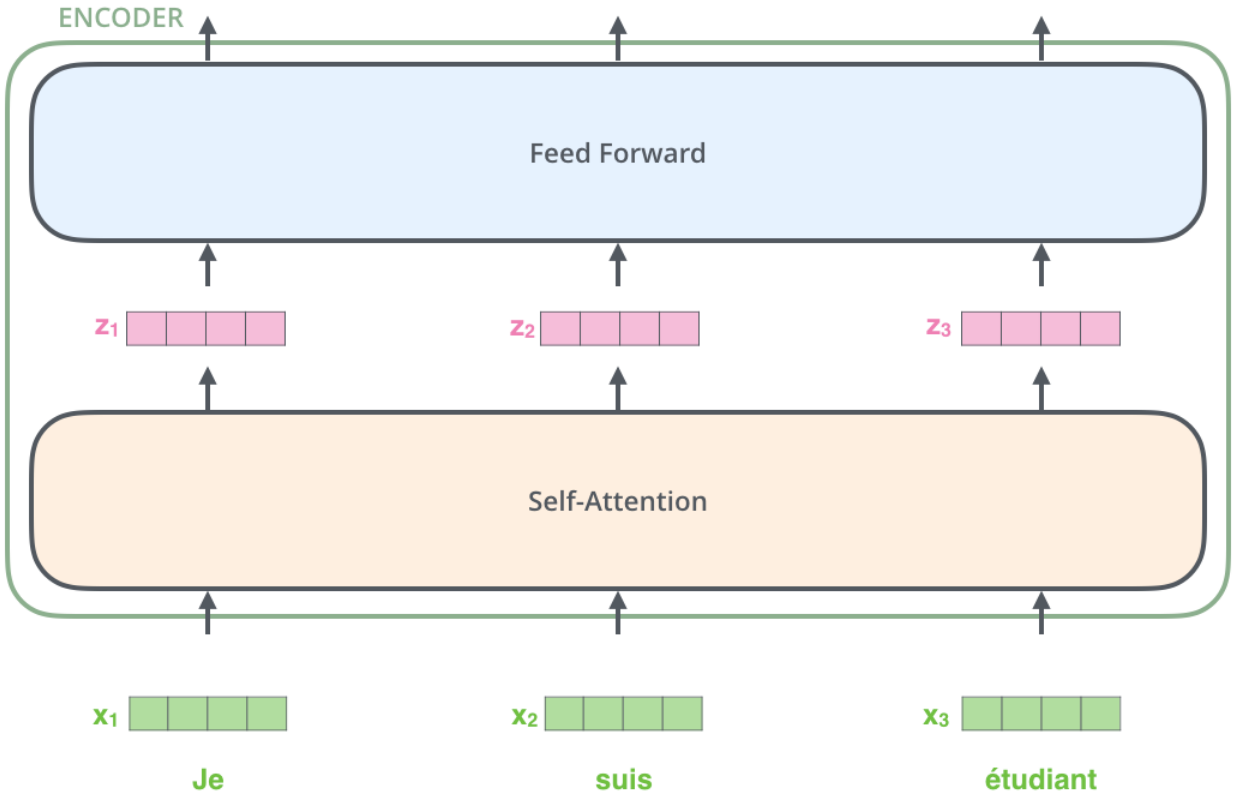
e.g. Machine Translation

e.g. Video Stream Classification

Slide taken from - Stanford, CS231n course, Lecture 10 presentation
http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf

Popular Deep Learning models

- **Transformers** –process sets and sequences



https://jalammar.github.io/illustrated-transformer/

Some terminology

- Optimizer
- Loss
- Weights
- Backprop
- Learning rate
- Batch size
- Weight decay
- Dropout
- Hyperparameters
- Batch Norm
- Layer
- Activations
- Augmentation
- Softmax
- …..

Thank you!