Neural Networks.
Introduction to Deep Learning.

Radu Ionescu, Prof. PhD.

raducu.ionescu@gmail.com

Faculty of Mathematics and Computer Science

University of Bucharest

From previous lecture:

- We have some dataset of (x,y)
- We have a **score function:** $s = f(x; W) \stackrel{\text{e.g.}}{=} Wx$
- We have a **loss function**:

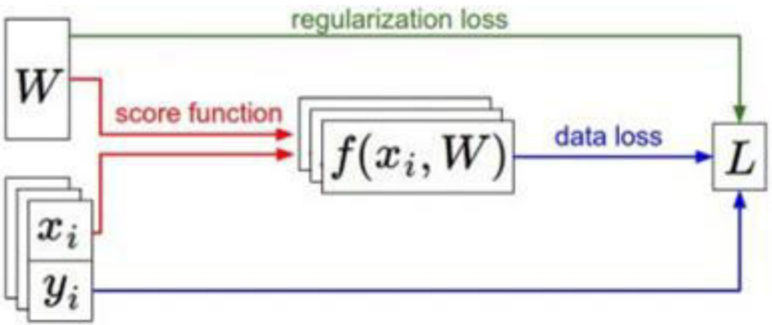
Softmax

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

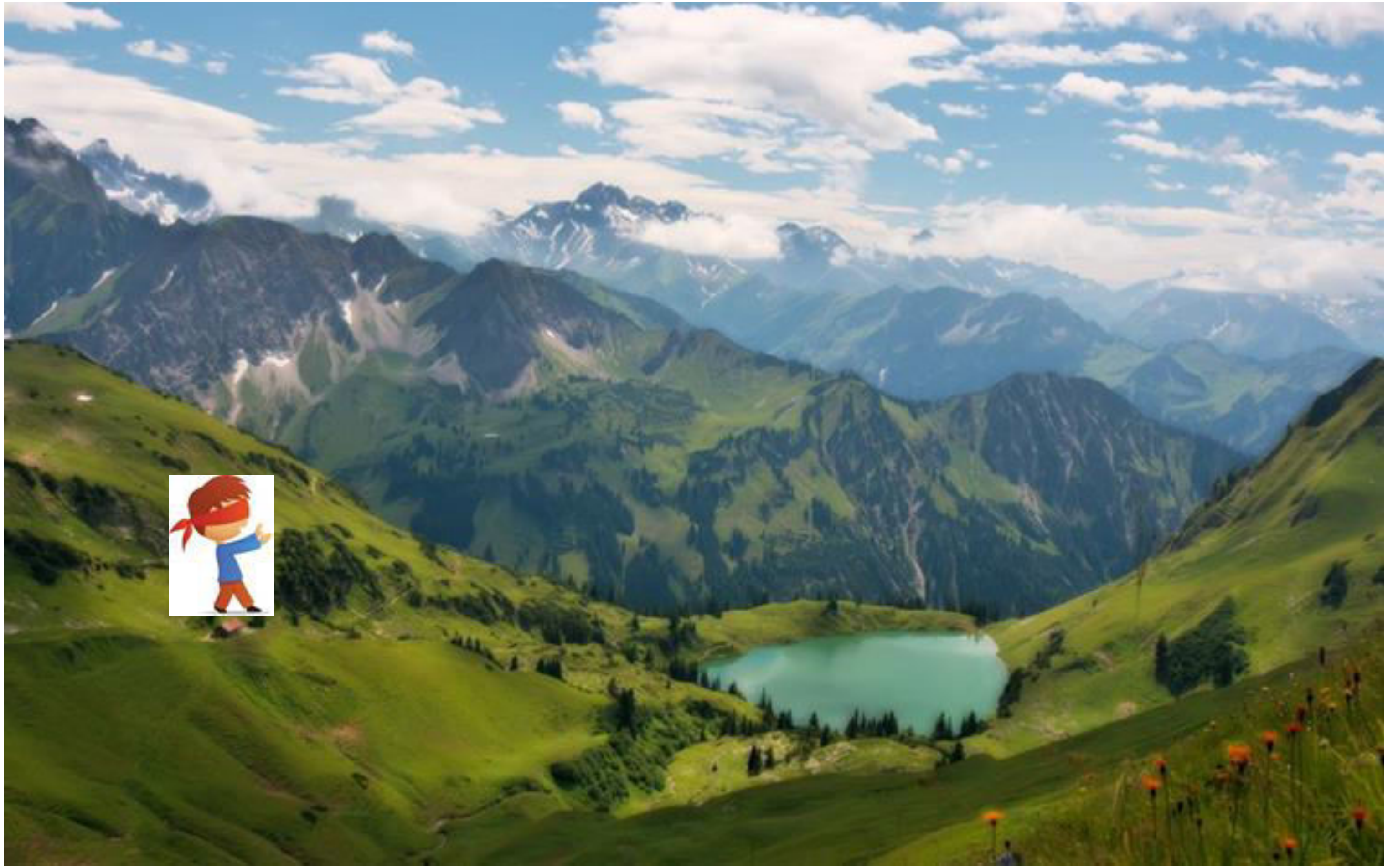
SVM

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W)$$ Full loss



Gradient Descent Algorithm



Gradient Evaluation

1) Numerical approach

We choose a small positive h and apply the formula:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- We obtain an approximate value
- Very slow to compute

2) Analytic approach

We use calculus to determine the gradient’s formula as a function of X and W

In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

=>

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient checking.**

Gradient Descent (Python)

```
def GD(W0, X, goal, learningRate):
  perfGoalNotMet = true
  W = W0

  while perfGoalNotMet:
    gradient = eval_gradient(X, W)
    W_old = W
    W = W – learningRate * gradient
    perfGoalNotMet = sum(abs(W - W_old)) > goal
```

From feature extract to end-to-end learning

vector describing various
image statistics

[32x32x3]

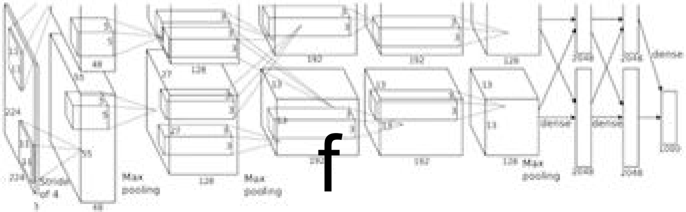
Feature
Extraction

f

training

N numbers, indicating
class scores

[32x32x3]

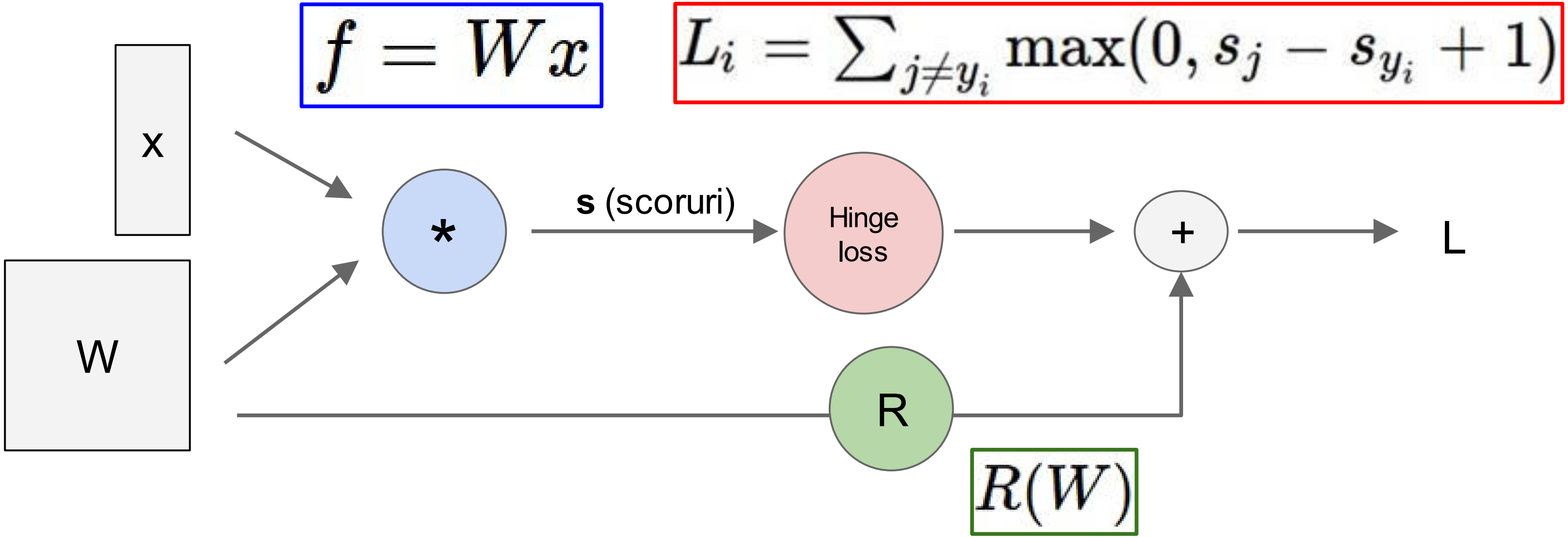


f

training

N numbers, indicating
class scores

Computational Graph



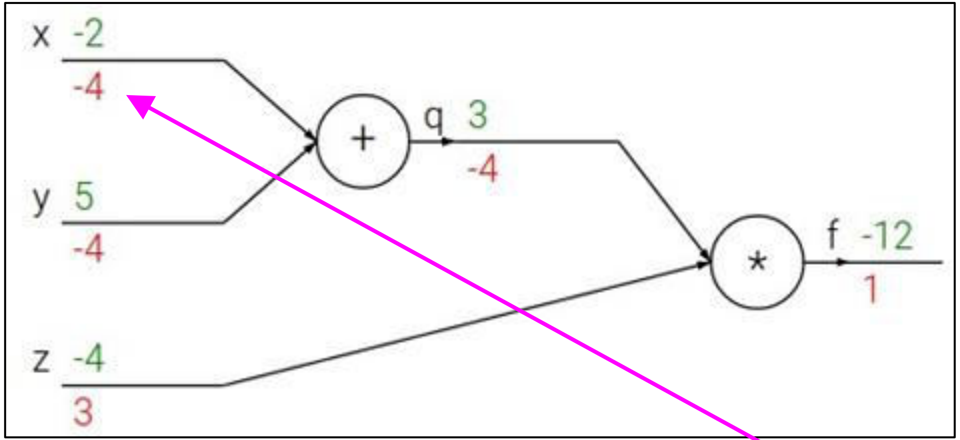
$$f(x,y,z) = (x+y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

vrem: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

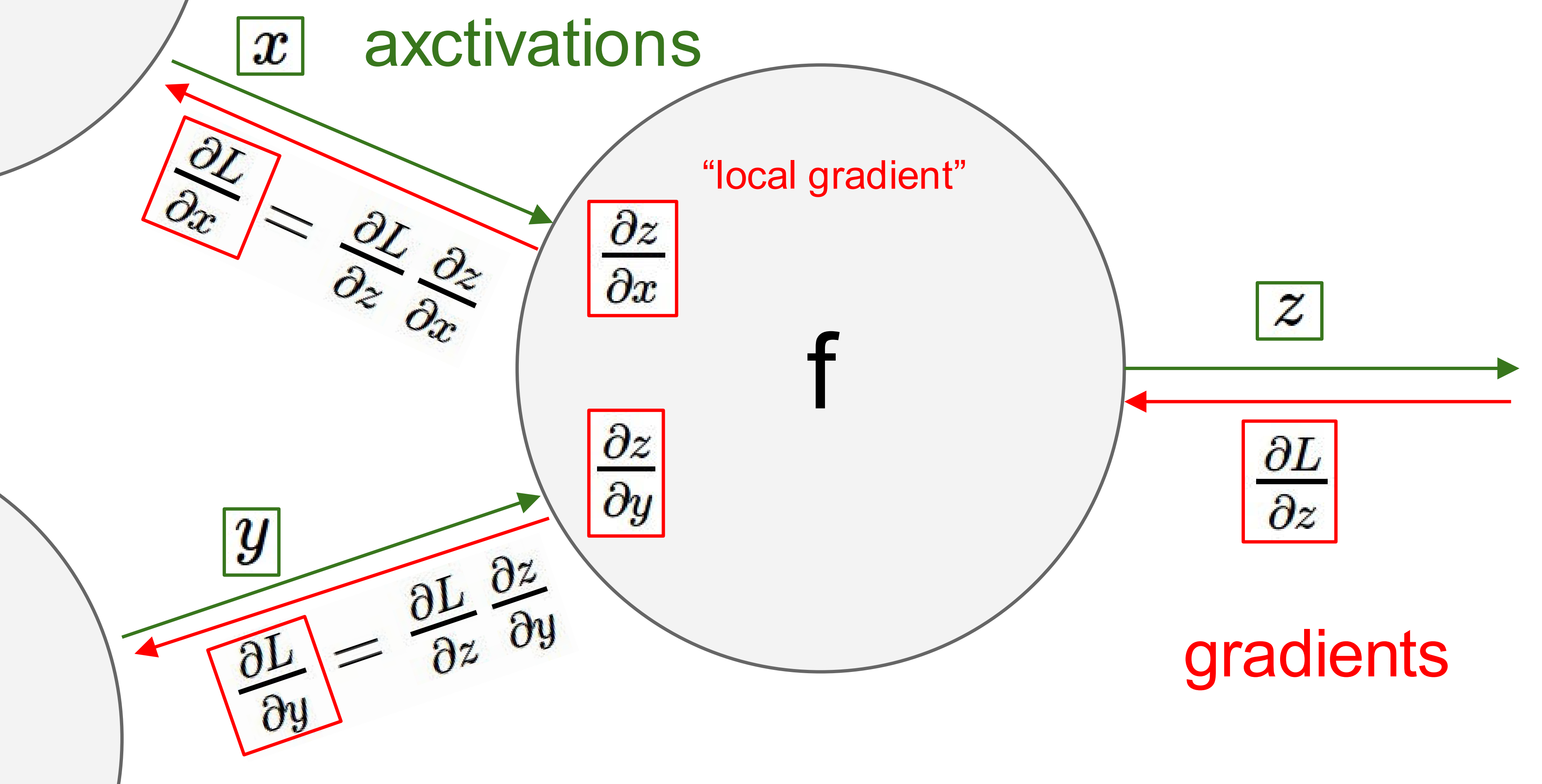


$\frac{\partial f}{\partial x}$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Gradient propagation using chain rule



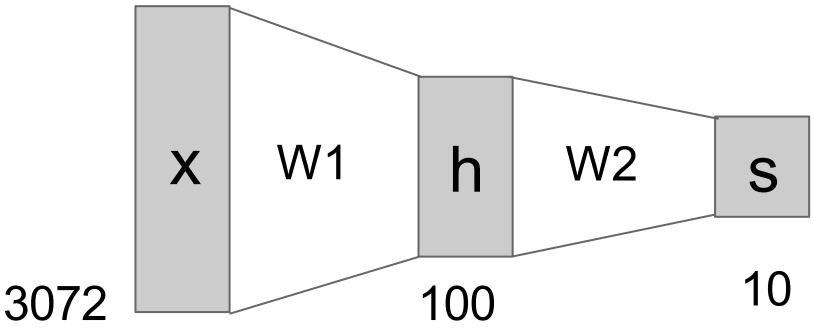
From previous lecture…

- Neural nets will be very large: no hope of writing down gradient formula by hand for all parameters
- **Backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs / parameters / intermediates
- Implementations maintain a graph structure, where the nodes implement the **forward**() / **backward**() API
- **Forward**: compute result of an operation and save any intermediates needed for gradient computation in memory
- **Backward**: apply the chain rule to compute the gradient of the loss function with respect to the inputs

Neural Network: without the brain stuff

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$



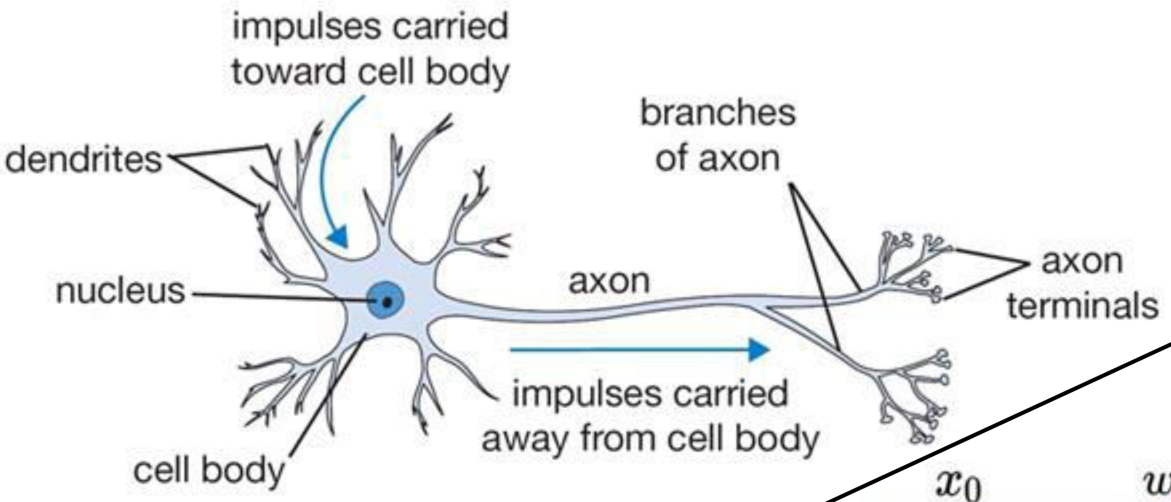
Neural Network: without the brain stuff

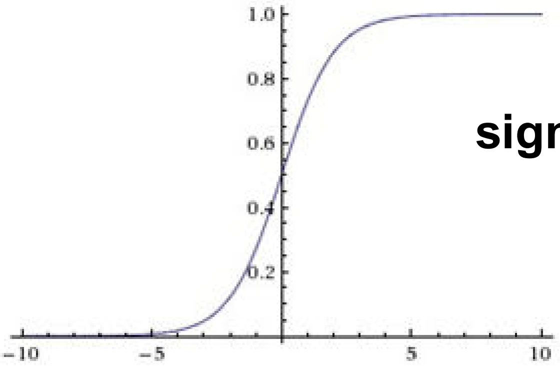
(**Before**) Linear score function: $$f = Wx$$

(**Now**) 2-layer Neural Network $$f = W_2 \max(0, W_1 x)$$

or 3-layer Neural Network

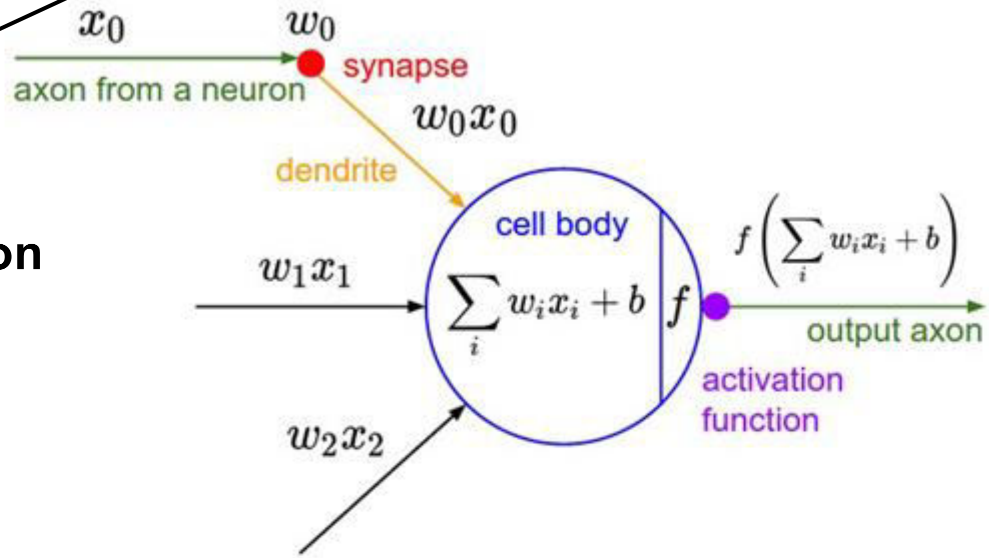
$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$





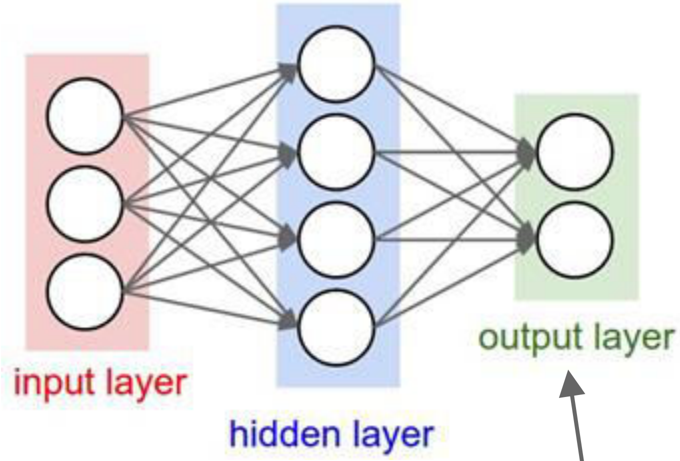
sigmoid activation

$$\frac{1}{1+e^{-x}}$$

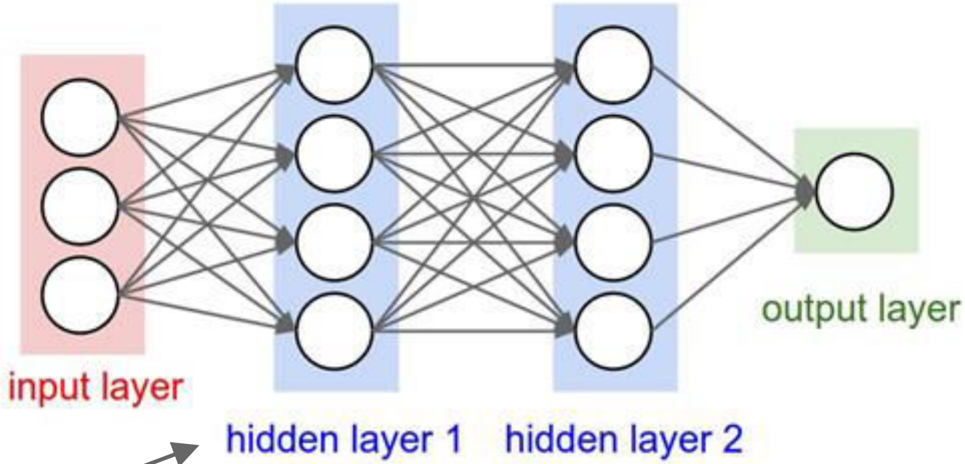


Neural Networks: Architectures

2-layer Neural Net
(1-hidden-layer Neural Net)

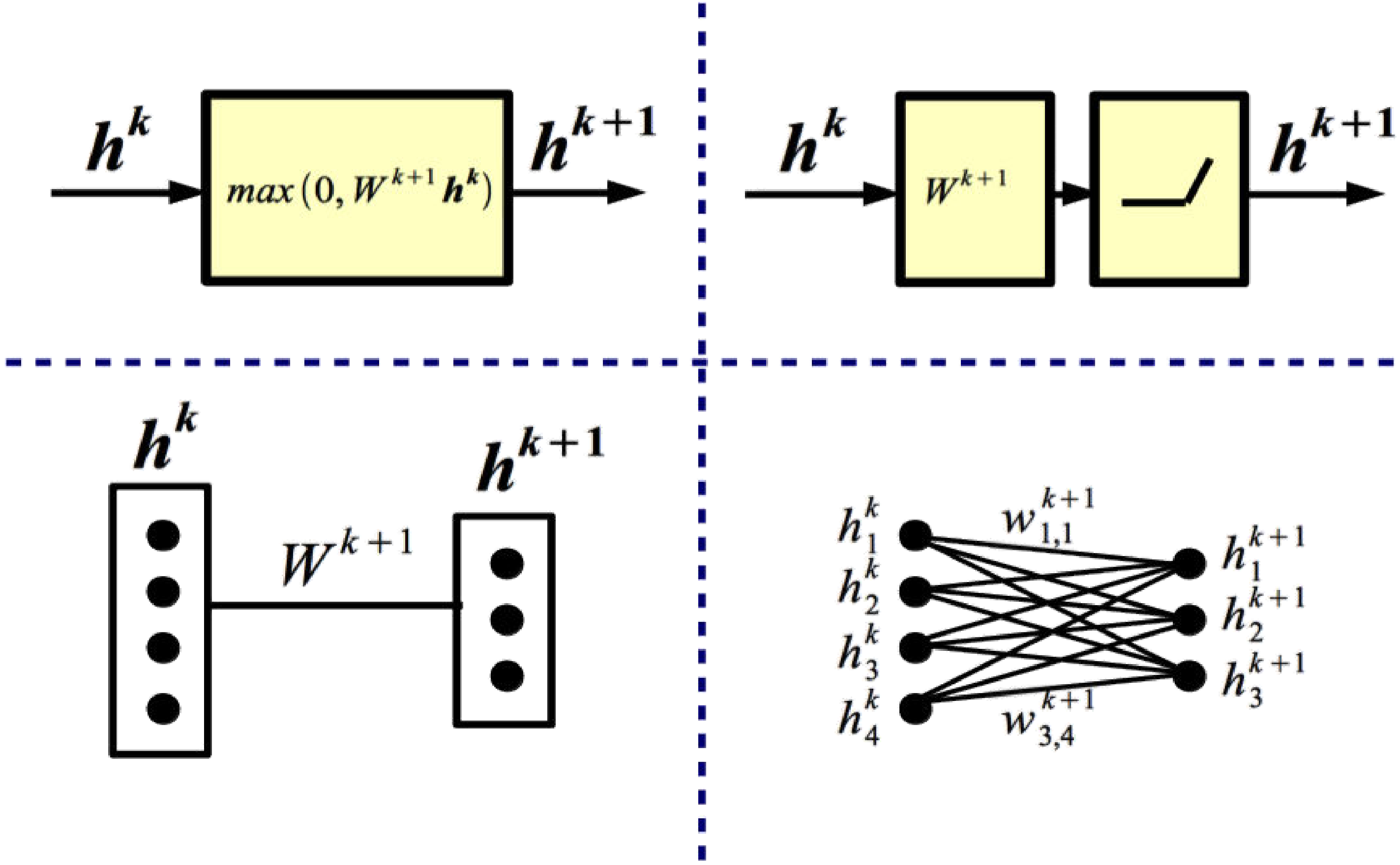


3-layer Neural Net
(2-hidden-layer Neural Net)



“Fully-connected” layers

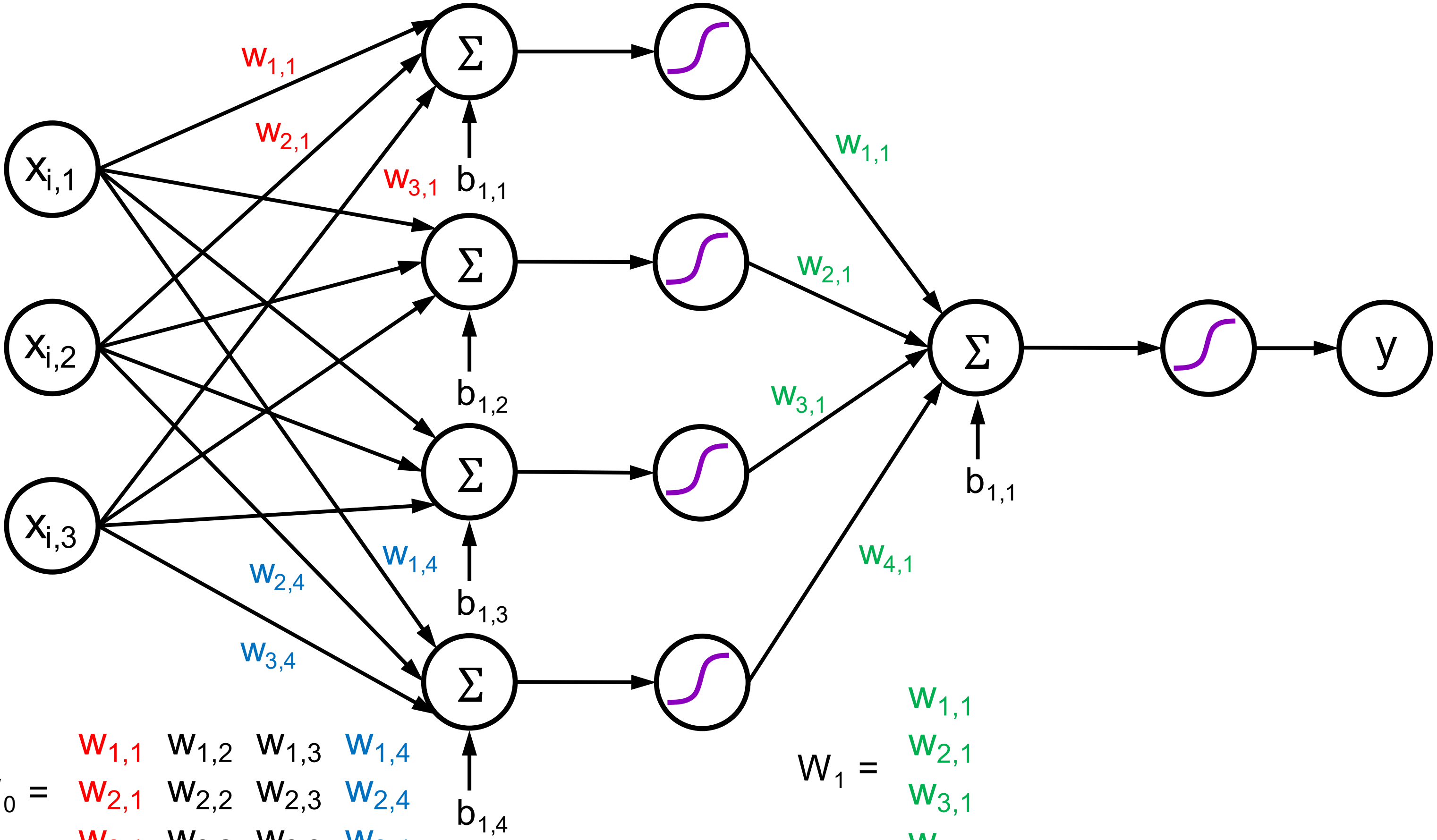
Equivalent Representations



Training a 2-layer Neural Network needs ~11 lines (Python)

```
X = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])
Y = np.array([[0,1,1,0]]).T

W0 = 2 * np.random.random((3,4)) - 1
W1 = 2 * np.random.random((4,1)) - 1

for i in range(5000):

    # forward pass
    l1 = 1 / (1 + np.exp(-np.matmul(X, W0)))
    l2 = 1 / (1 + np.exp(-np.matmul(l1, W1)))

    # backward pass
    delta_l2 = (Y - l2) * (l2 * (1 - l2))
    delta_l1 = np.matmul(delta_l2, W1.T) * (l1 * (1 - l1))

    # gradient descent
    W1 = W1 + np.matmul(l1.T, delta_l2)
    W0 = W0 + np.matmul(X.T, delta_l1)
```

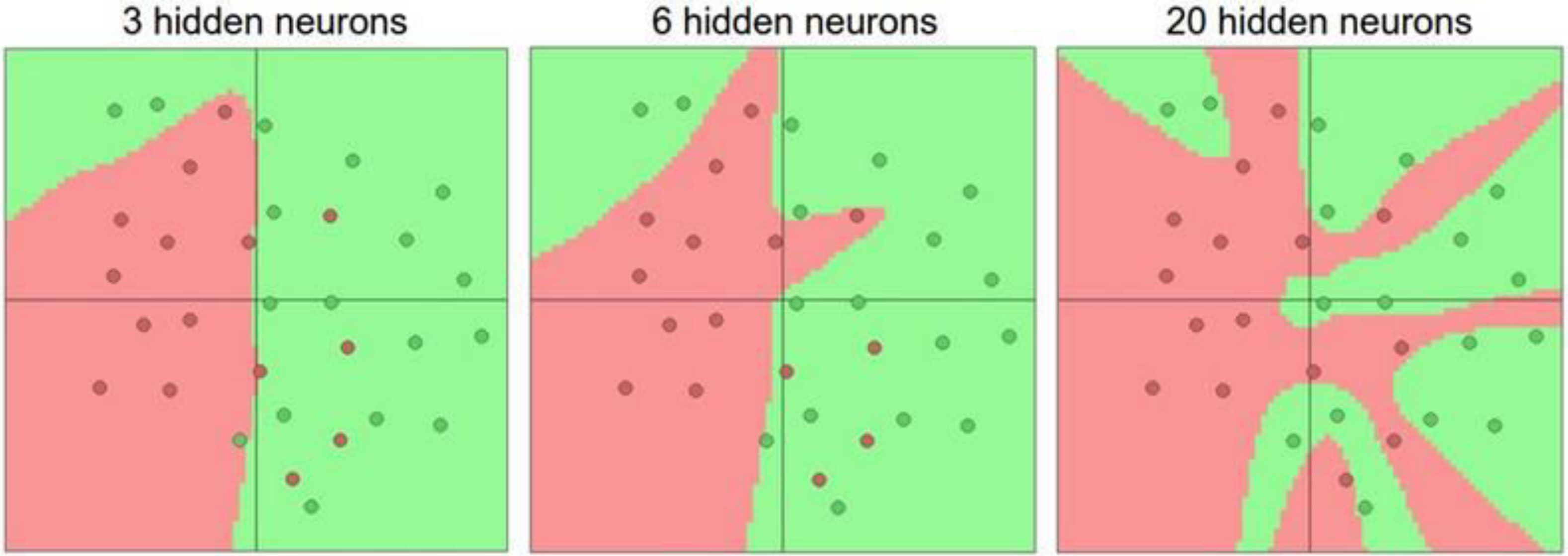
The 2-layer neural network implemented in the previous slide



$$W_0 = \begin{matrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \end{matrix}$$

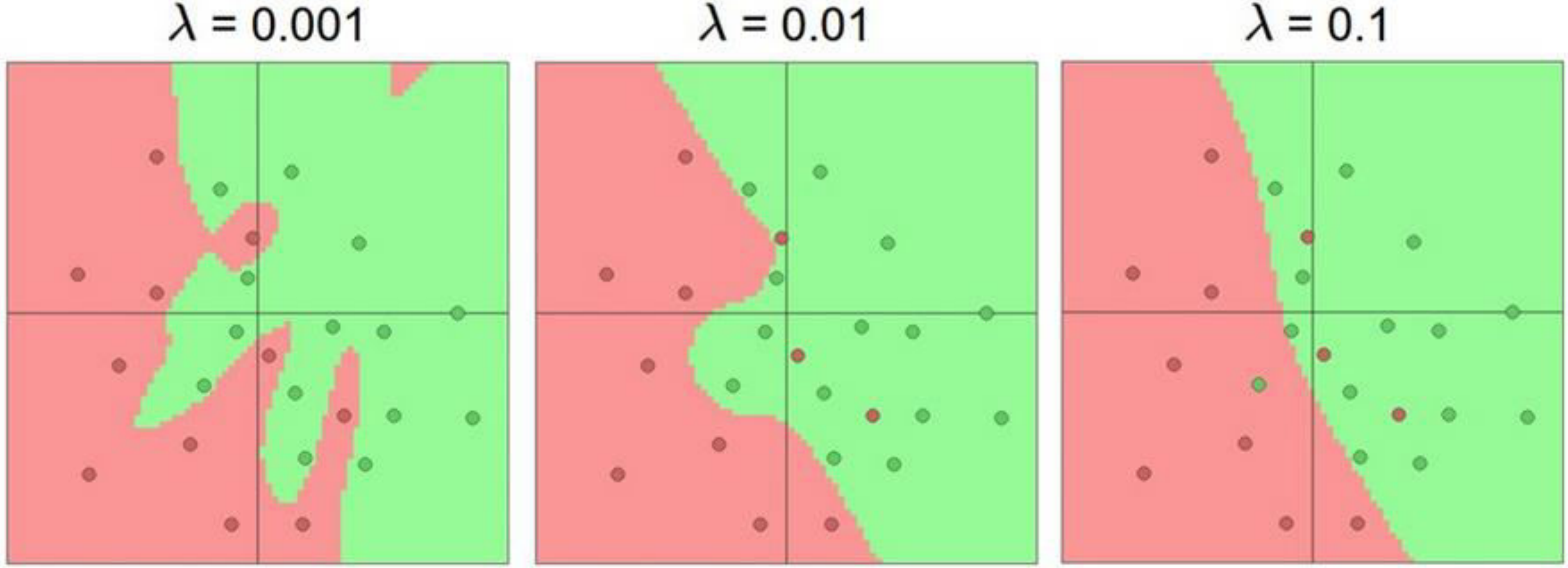
$$W_1 = \begin{matrix} w_{1,1} \\ w_{2,1} \\ w_{3,1} \\ w_{4,1} \end{matrix}$$

Setting the number of layers and their sizes



more neurons = more capacity

Do not use size of neural network as a regularizer. Use stronger regularization instead:



Practical advice: In general, it is better to use stronger regularization than reducing the model's capacity

Choosing the right architecture

- We arrange neurons into fully-connected layers
- The abstraction of a **layer** has the nice property that it allows us to use efficient vectorized code (e.g. matrix multiplies)
- neural networks: bigger = better (but might have to regularize more strongly)

Training Neural Networks

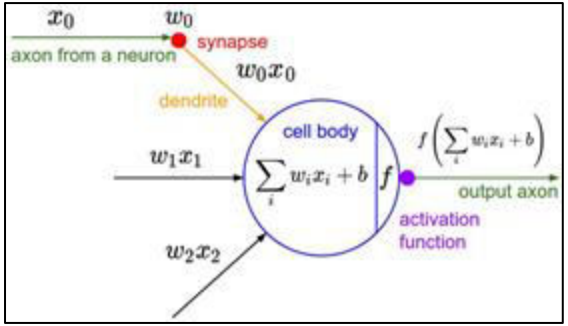
A bit of history

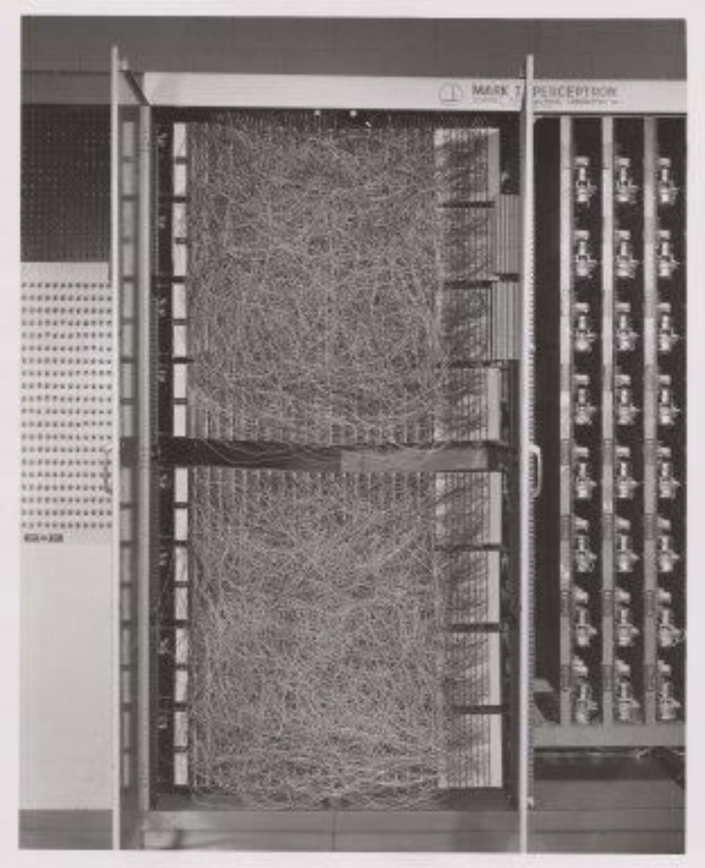
The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.

The machine was connected to a camera that used 20×20 cadmium sulfide photocells to produce a 400-pixel image.

Recognized letters of the alphabet.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$





Update rule:

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i},$$

Frank Rosenblatt, ~1957: Perceptron

A bit of history

The **ADALINE** machine is based on memistors capable of executing logical operations and storing information.

Loss (sum squared error):

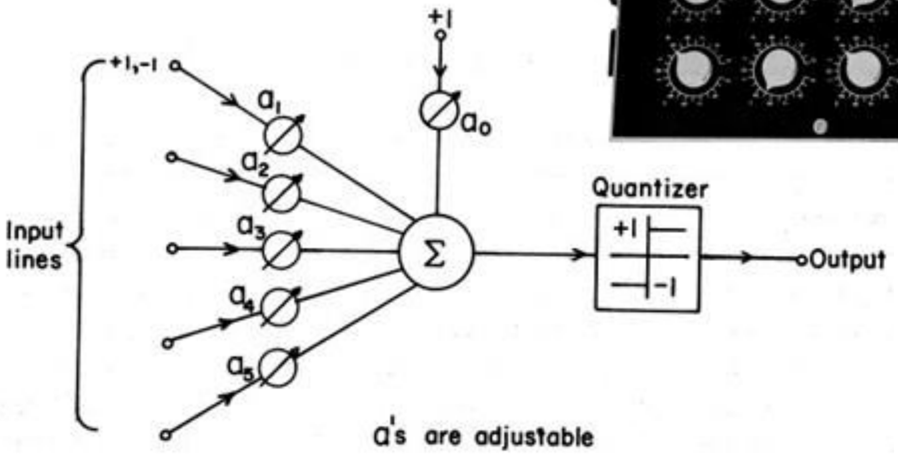
$$\frac{1}{2}\sum_i (d^i - y^i)^2, \text{ unde } y^i = (x^i)^T w + b$$

Update rule:

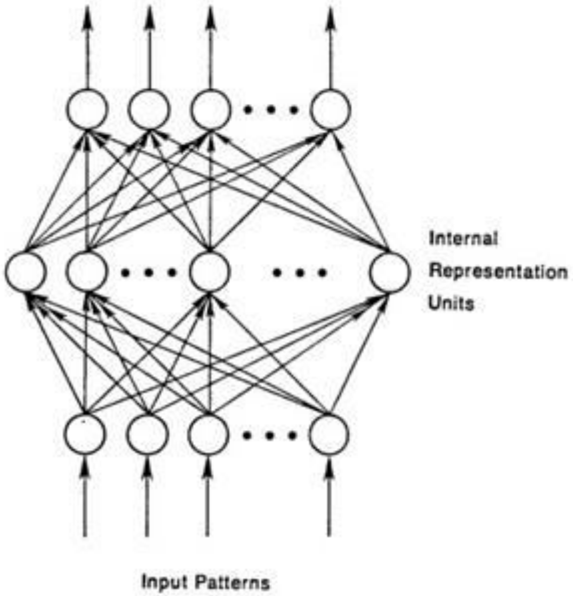
$$w^{k+1} = w^k + \mu \sum_{i=1}^{m} (d^i - y^i) x^i$$

$$b^{k+1} = b^k + \mu \sum_{i=1}^{m} (d^i - y^i)$$

Widrow and Hoff, ~1960: Adaline



A bit of history



To be more specific, then, let

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 \tag{2}$$

be our measure of the error on input/output pattern p and let $E = \sum E_p$ be our overall measure of the error. We wish to show that the delta rule implements a gradient descent in E when the units are linear. We will proceed by simply showing that

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_{pi},$$

which is proportional to $\Delta_p w_{ji}$ as prescribed by the delta rule. When there are no hidden units it is straightforward to compute the relevant derivative. For this purpose we use the chain rule to write the derivative as the product of two parts: the derivative of the error with respect to the output of the unit times the derivative of the output with respect to the weight.

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ji}}. \tag{3}$$

The first part tells how the error changes with the output of the jth unit and the second part tells how much changing w_{ji} changes that output. Now, the derivatives are easy to compute. First, from Equation 2

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) = -\delta_{pj}. \tag{4}$$

Not surprisingly, the contribution of unit u_j to the error is simply proportional to δ_{pj}. Moreover, since we have linear units,

$$o_{pj} = \sum_i w_{ji} i_{pi}, \tag{5}$$

from which we conclude that

$$\frac{\partial o_{pj}}{\partial w_{ji}} = i_{pi}.$$

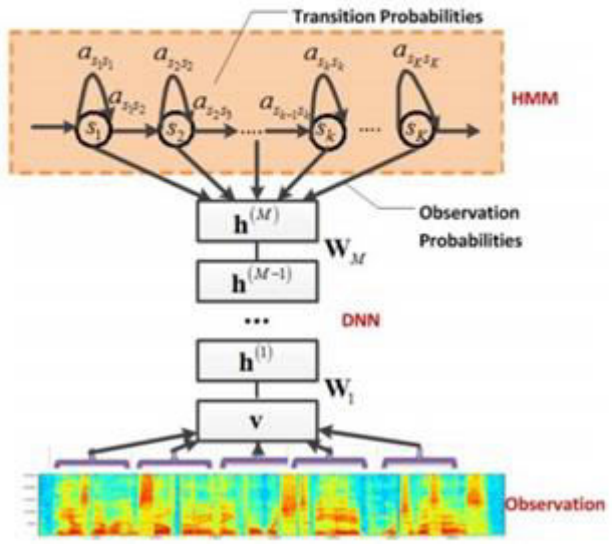
Thus, substituting back into Equation 3, we see that

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_i \tag{6}$$

recognizable maths

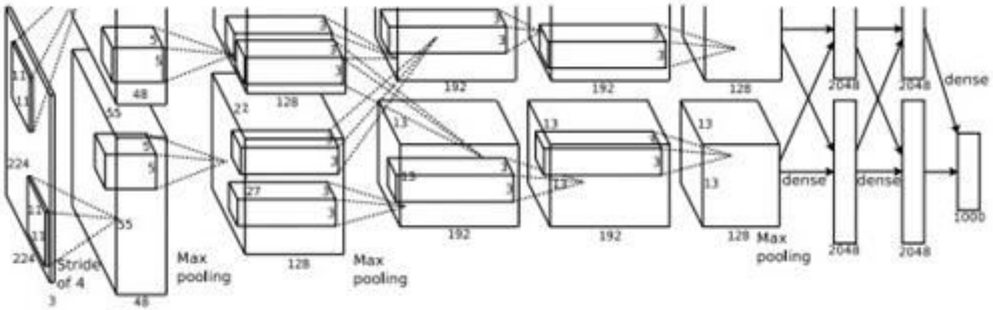
Hinton et al. 1986: First time back-propagation became popular

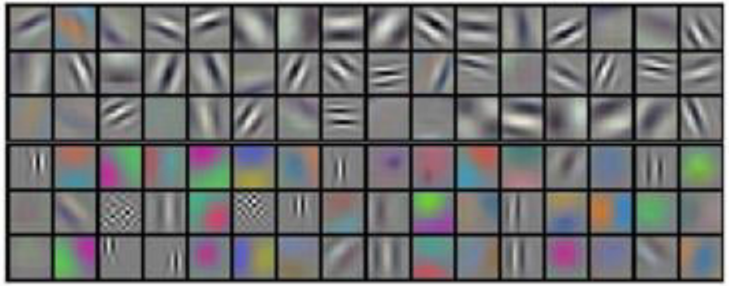
First strong results with deep learning



Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition
George Dahl, Dong Yu, Li Deng, Alex Acero, 2010

Imagenet classification with deep convolutional neural networks
Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton, 2012



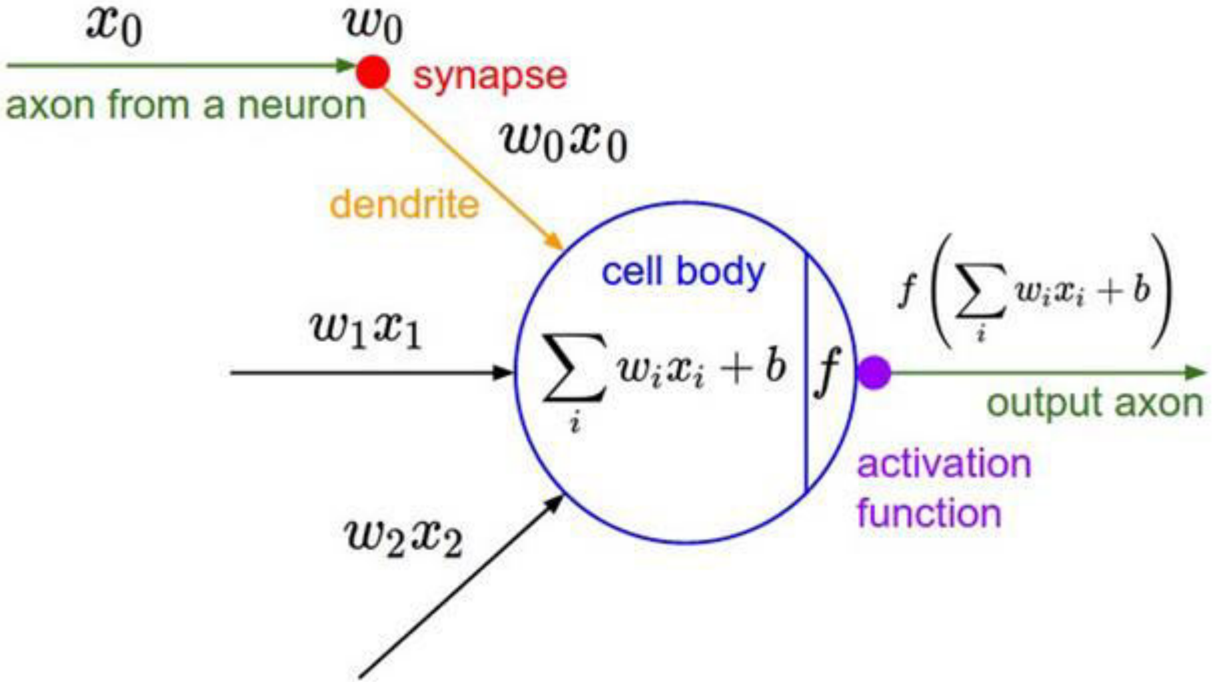


Overview

1. **One time setup**
   activation functions, preprocessing, weight initialization, regularization, gradient checking
2. **Training dynamics**
   babysitting the learning process, parameter updates, hyperparameter optimization
3. **Evaluation**
   model ensembles

Activation Functions

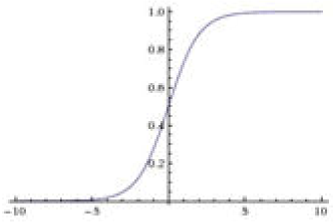
Activation Functions



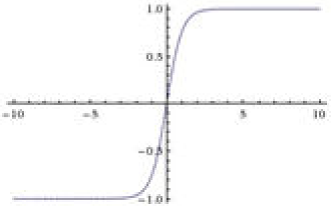
Activation Functions

sigmoid

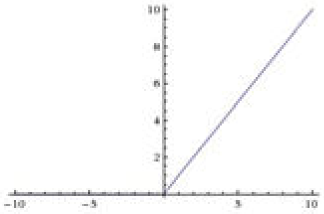
$$\sigma(x) = 1/(1+e^{-x})$$



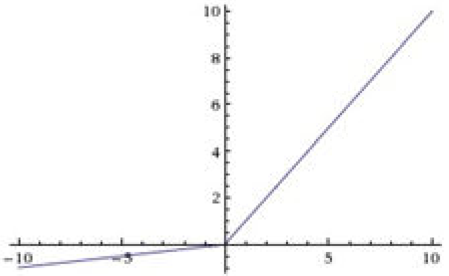
tanh tanh(x)



ReLU max(0,x)

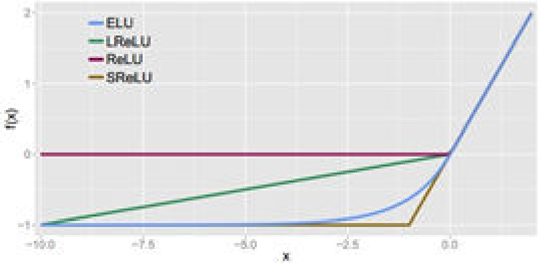


Leaky ReLU
max(0.1x, x)

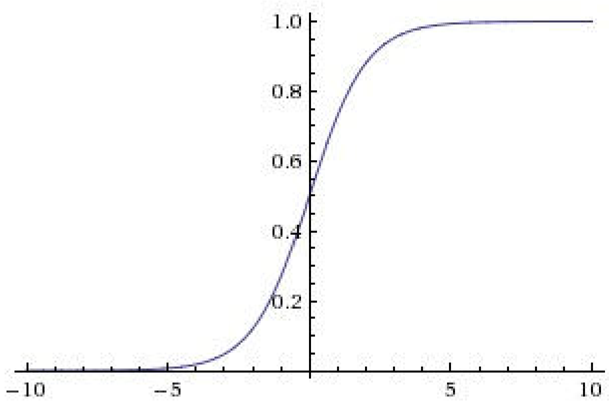


Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$



Activation Functions



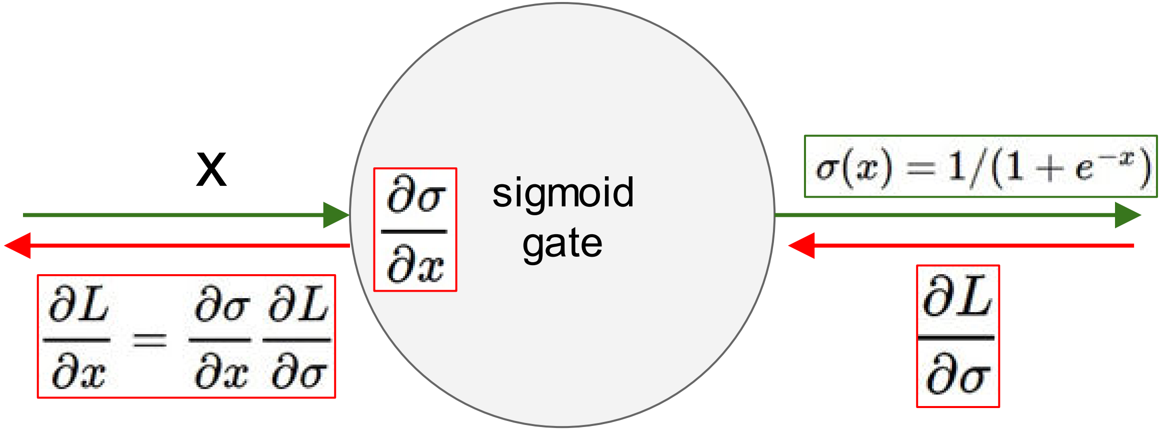
sigmoid

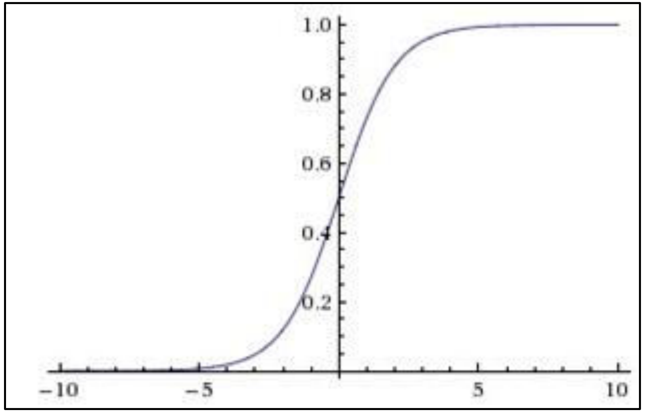
$$\sigma(x) = 1/(1+e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

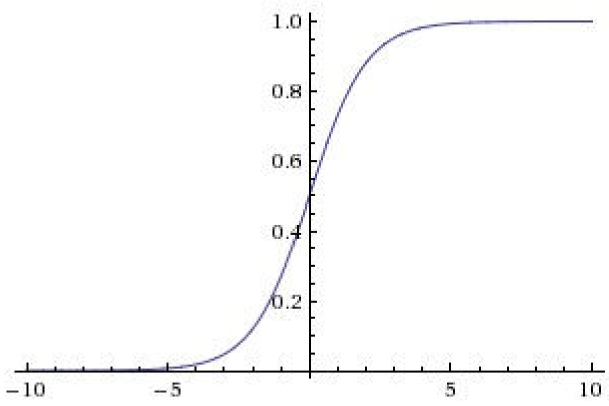
1. Saturated neurons "kill" the gradients





What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Activation Functions



sigmoid

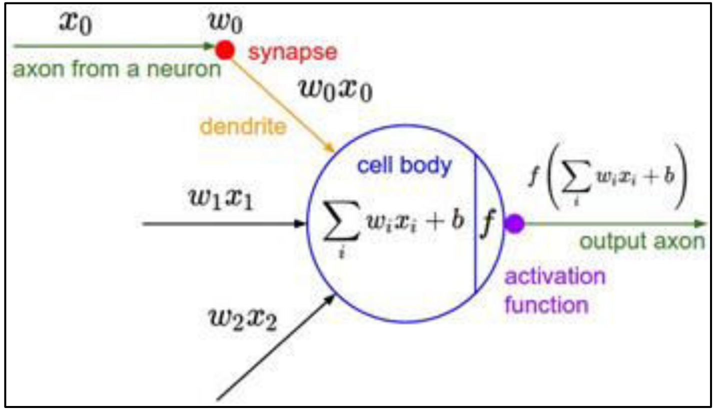
$$\sigma(x) = 1/(1+e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered

Consider what happens when the input to a neuron (x) is always positive:

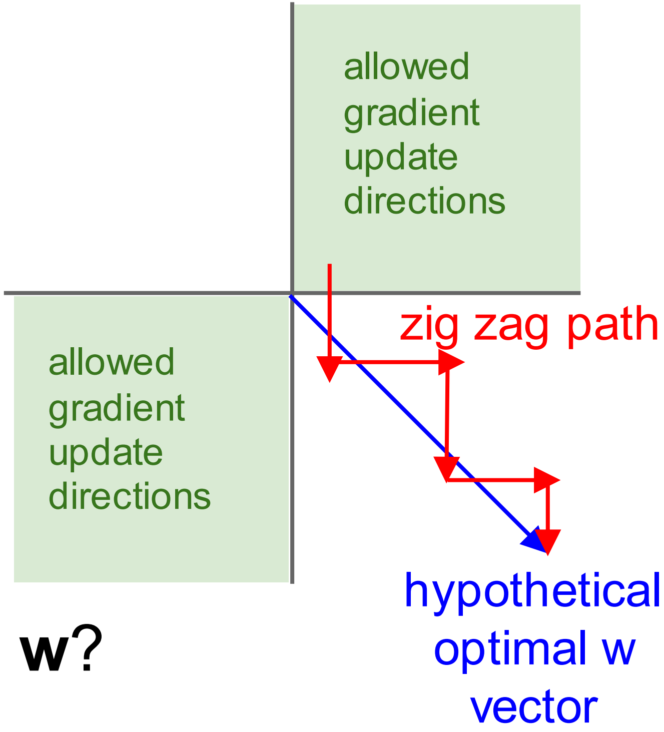


$$f\left(\sum_i w_i x_i + b\right)$$

What can we say about the gradients on **w**?

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$

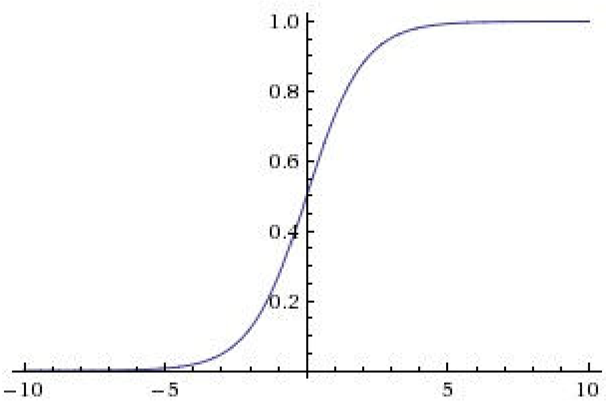


What can we say about the gradients on **w**?

Always all positive or all negative :(

Note: this is also why you want zero-mean data!

Activation Functions



sigmoid

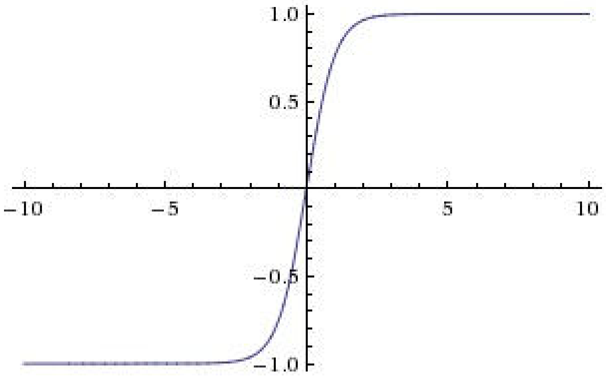
$$\sigma(x) = 1/(1+e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit computationally expensive

Activation Functions



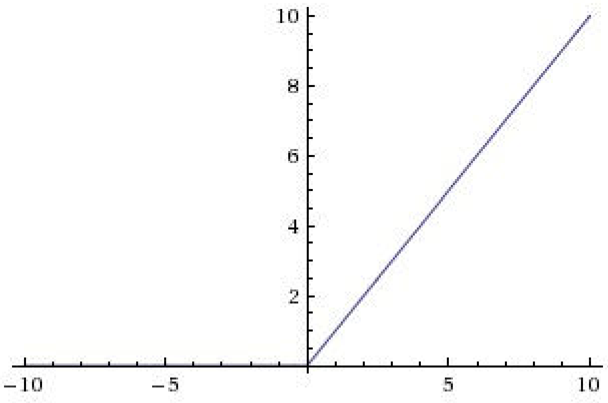
tanh(x)

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

Activation Functions



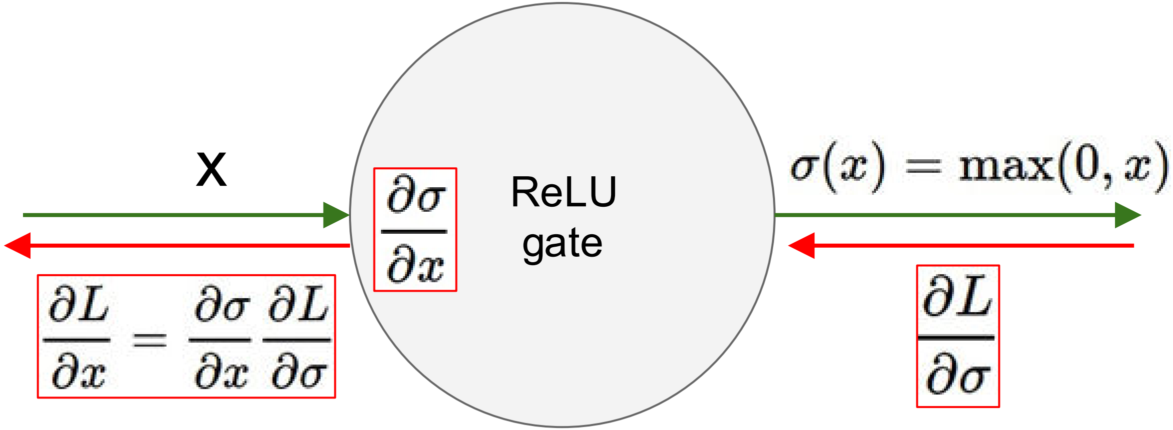
ReLU
(Rectified Linear Unit)

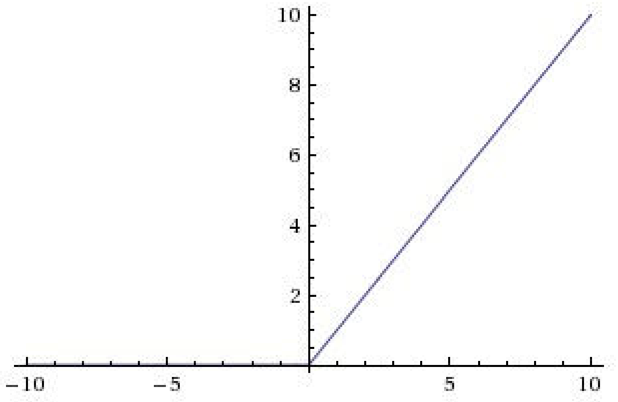
f(x) = max(0,x)

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

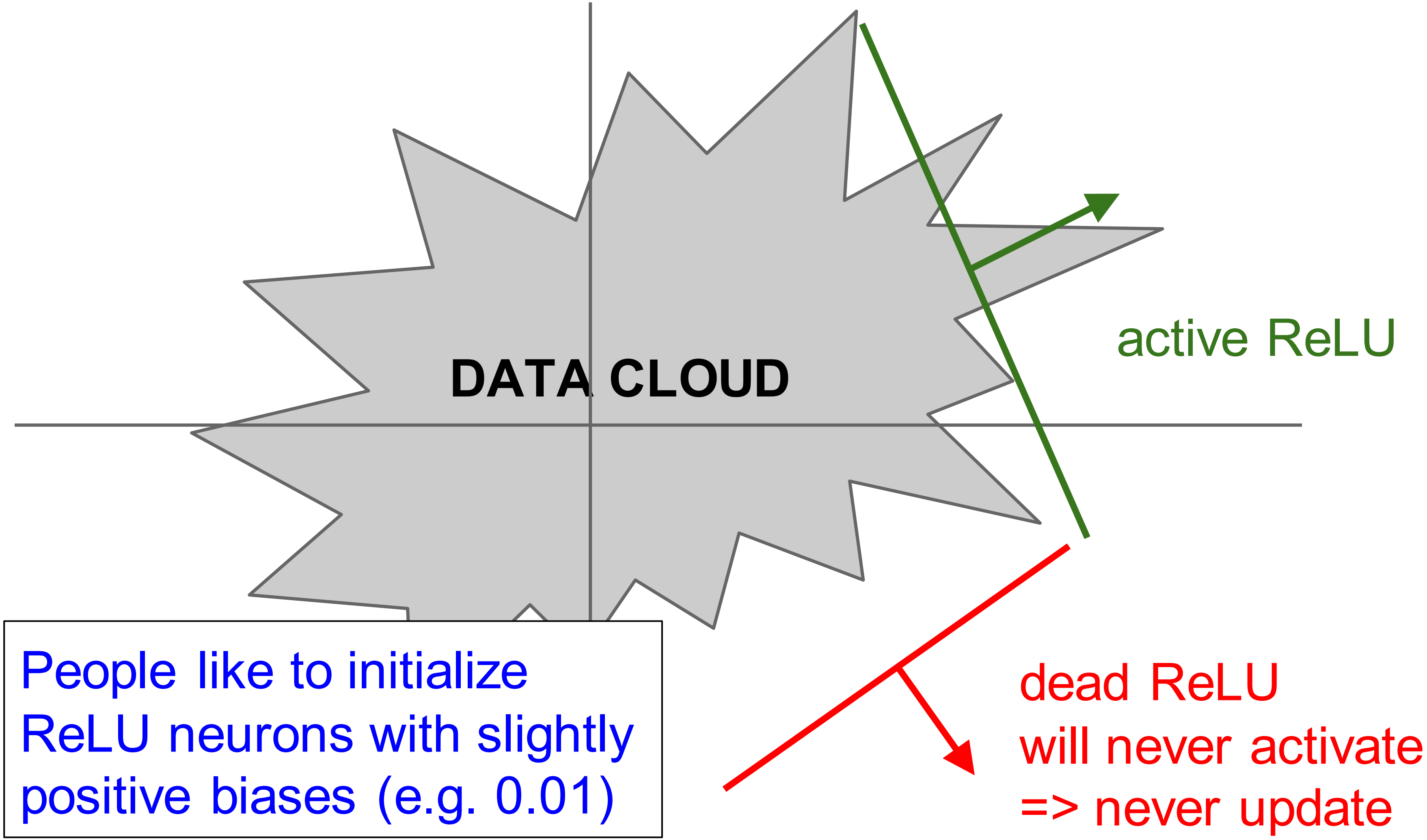
- Not zero-centered output
- An annoyance: what is the gradient when $x < 0$?

[Krizhevsky et al., 2012]



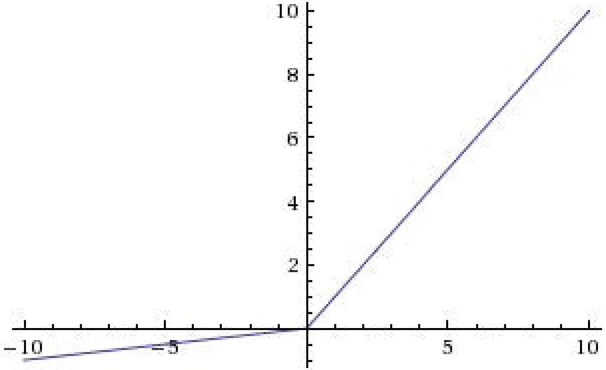


What happens when x = -10?
What happens when x = 0?
What happens when x = 10?



People like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Activation Functions



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die"**

Parametric Rectifier (PReLU)

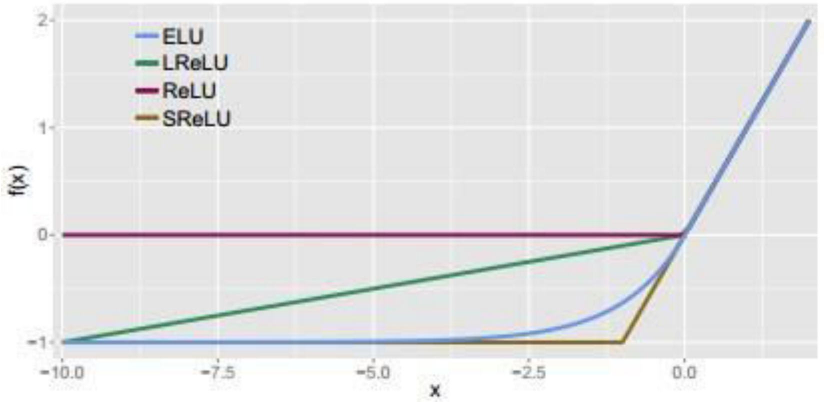
$$f(x) = \max(\alpha x, x)$$

backprop into α
(parameter)

[Mass et al., 2013]
[He et al., 2015]

Activation Functions

Exponential Linear Units (ELU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs

- Computation requires exp()

[Clevert et al., 2015]

Maxout "Neuron"

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

Problem: doubles the number of parameters/neurons :(

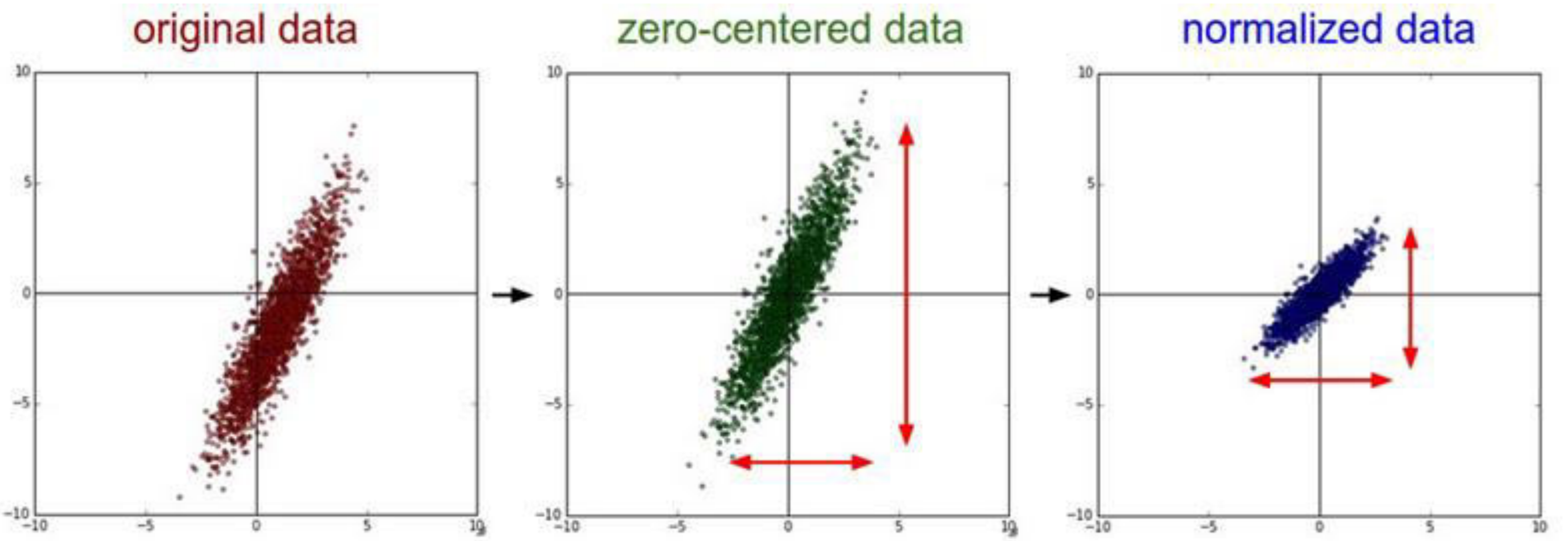
[Goodfellow et al., 2013]

TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Avoid using sigmoid

Data Preprocessing

Step 1: Preprocess the data



X = X – np.mean(X, axis=0, keepdims=True)

X = X / np.std(X, axis=0, keepdims=True))

(Assume X [NxD] is data matrix, each example in a row)

TLDR: In practice for images: center only

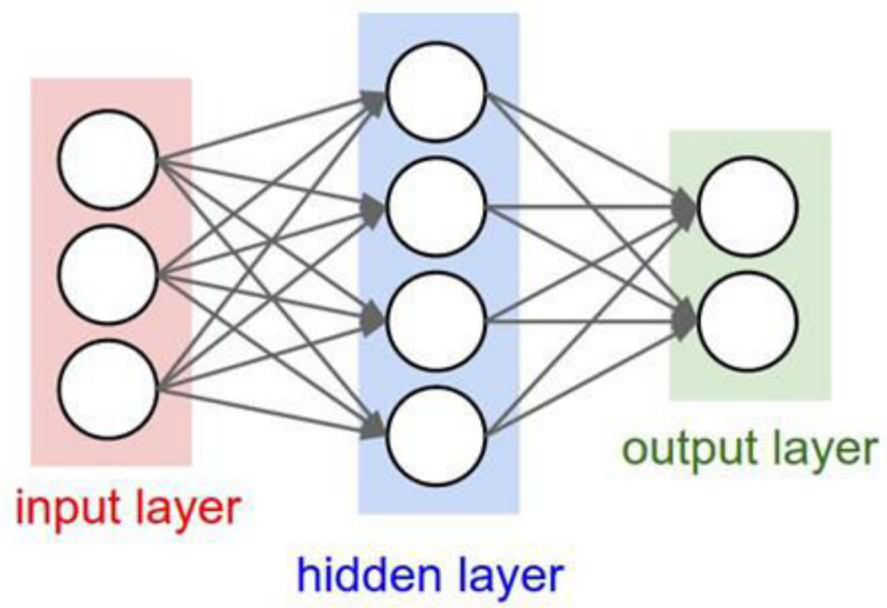
Consider CIFAR-10 example with [32,32,3] images:

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

It is not mandatory to normalize data for deep networks

Weight Initialization

Q: what happens when W=0 init is used?



First idea: **Small random numbers**

W = np.random.normal(0, 0.01, (N,D))
(gaussian with zero mean and 0.01 standard deviation)

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

Second idea: **Xavier Initialization**

Problems with choosing the initial weights:
- If they are too small, the signal strength propagating in the network drops with each level until it becomes too small to be useful
- If they are too big, the signal strength propagating in the network grows with each level until it becomes too big to be useful

- Xavier initialization ensures that the weight have the right magnitude, keeping the signal strength in a reasonable interval.
- The initial weights come from a normal distribution of mean 0 and standard deviation given by the number of perceptrons in previous/next layer:

$$\mathrm{Var}(W) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

[Glorot and Bengio, 2010]

Proper initialization is an active area of research…

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks
by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks
by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks
by Krähenbühl et al., 2015

All you need is a good init
by Mishkin and Matas, 2015

…

Batch Normalization

"you want unit gaussian activations?
just make them so."

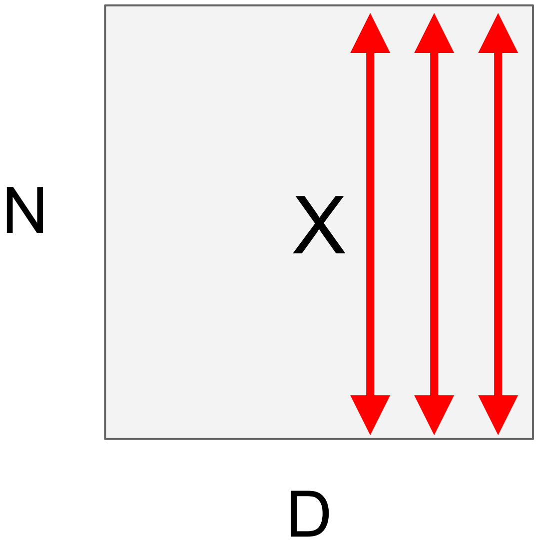
Consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

[Ioffe and Szegedy, 2015]

Batch Normalization

“you want unit gaussian activations?
just make them so.”



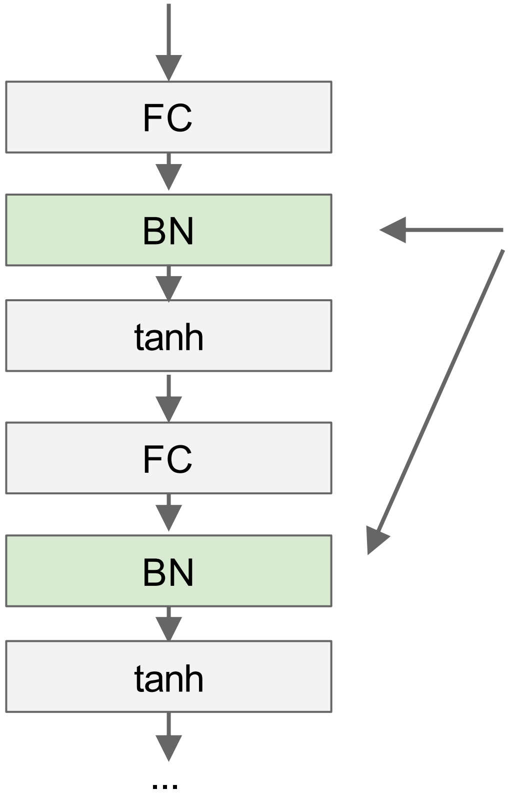
1. Compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

[Ioffe and Szegedy, 2015]

Batch Normalization



Usually inserted after fully-connected or convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

[Ioffe and Szegedy, 2015]

Batch Normalization

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

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

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

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

to recover the identity mapping.

[Ioffe and Szegedy, 2015]

Batch Normalization

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

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

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

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

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

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

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

[Ioffe and Szegedy, 2015]

Batch Normalization

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

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

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

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

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

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

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

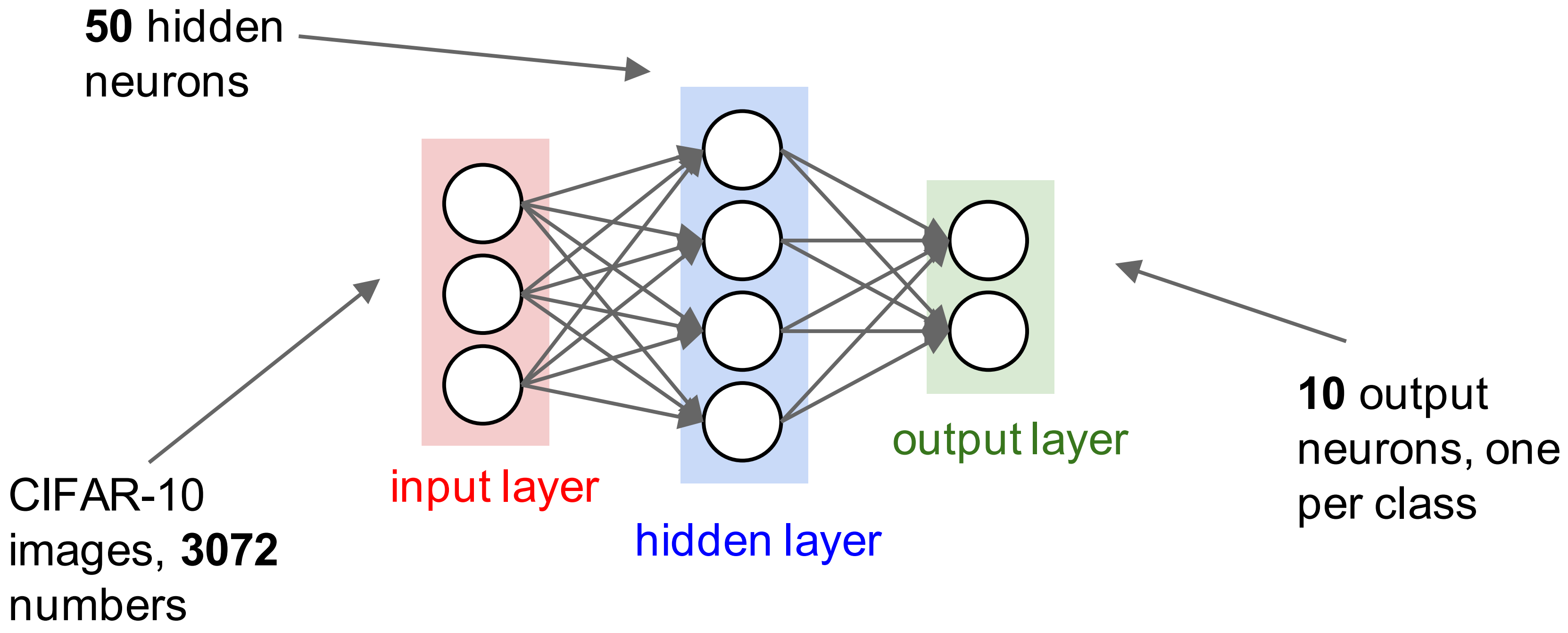
(e.g. can be estimated during training with running averages)

[Ioffe and Szegedy, 2015]

Babysitting the Learning Process

Step 2: Choose the architecture

Say we start with one hidden layer of 50 neurons, then we gradually add more layers

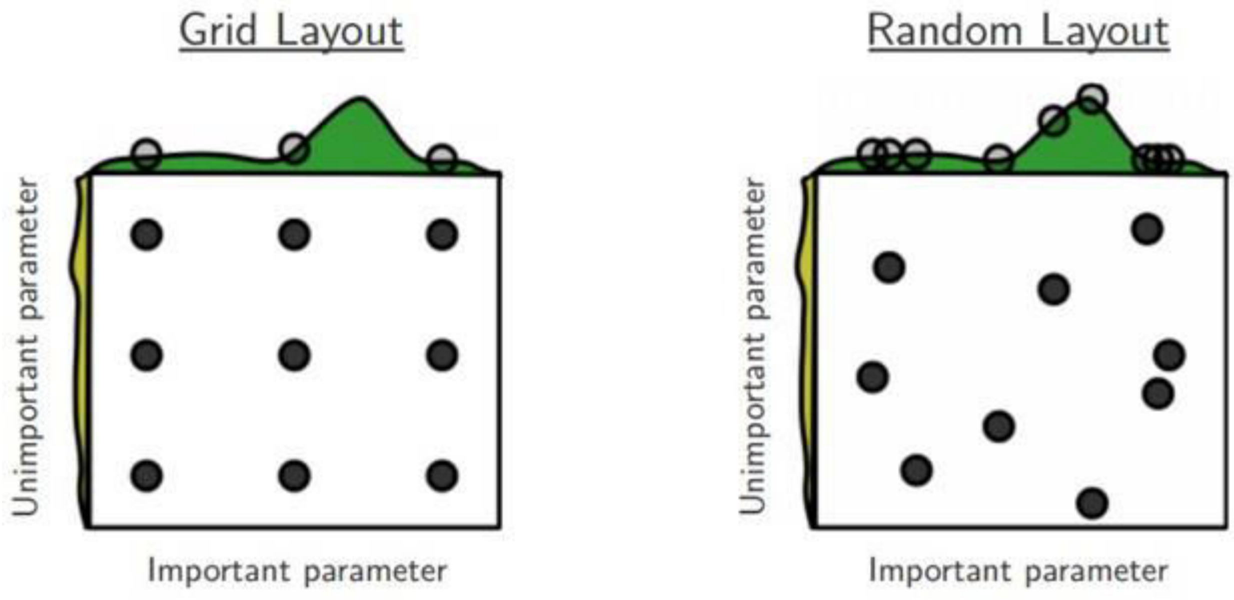


Practical Advice

1. Turn off regularization and check that the loss has a reasonable value (~2.5 for 10 classes is ok)
2. Turn on regularization and make sure that the value of the loss function grows, e.g. 3.2
3. Make sure that we can do overfitting on a small subset of the training data (e.g. 20 samples)
4. Start with small regularization and find learning rate that makes the loss go down

Hyperparameter Optimization

Random Search vs. Grid Search



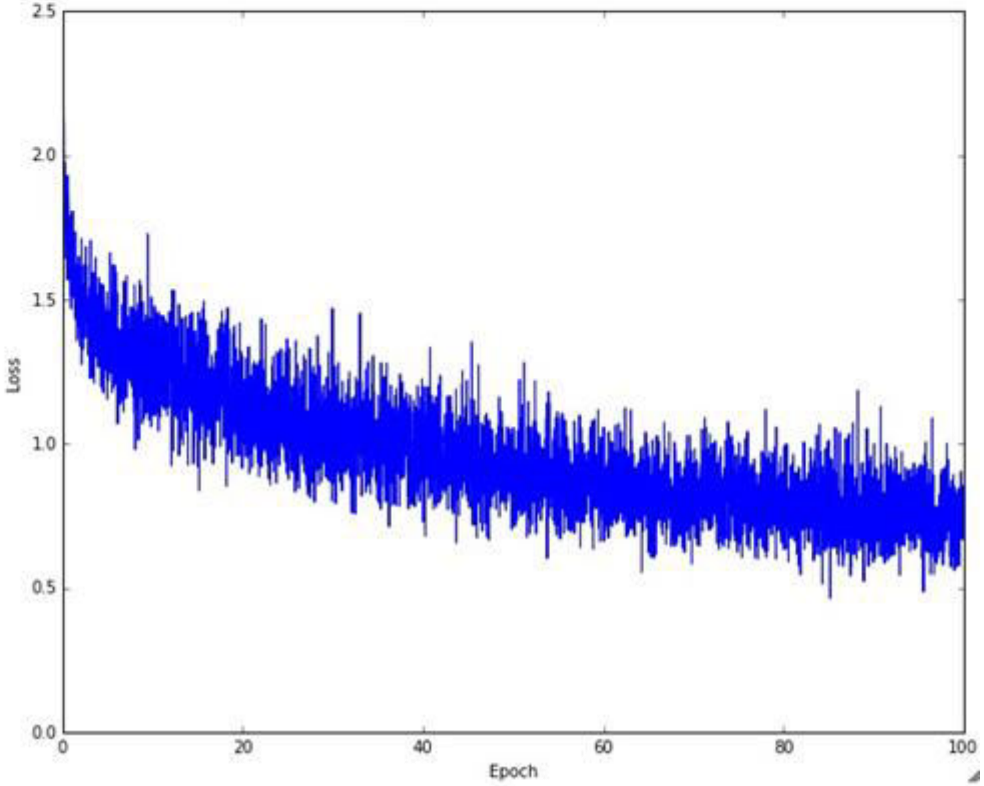
Random Search for Hyper-Parameter Optimization
[Bergstra and Bengio, 2012]

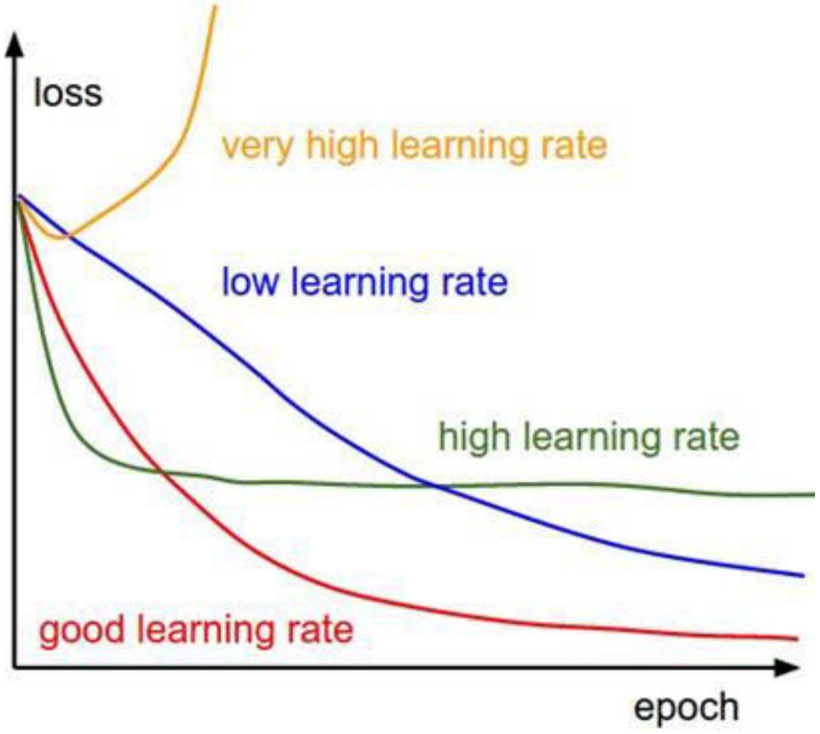
Hyperparameters to play with

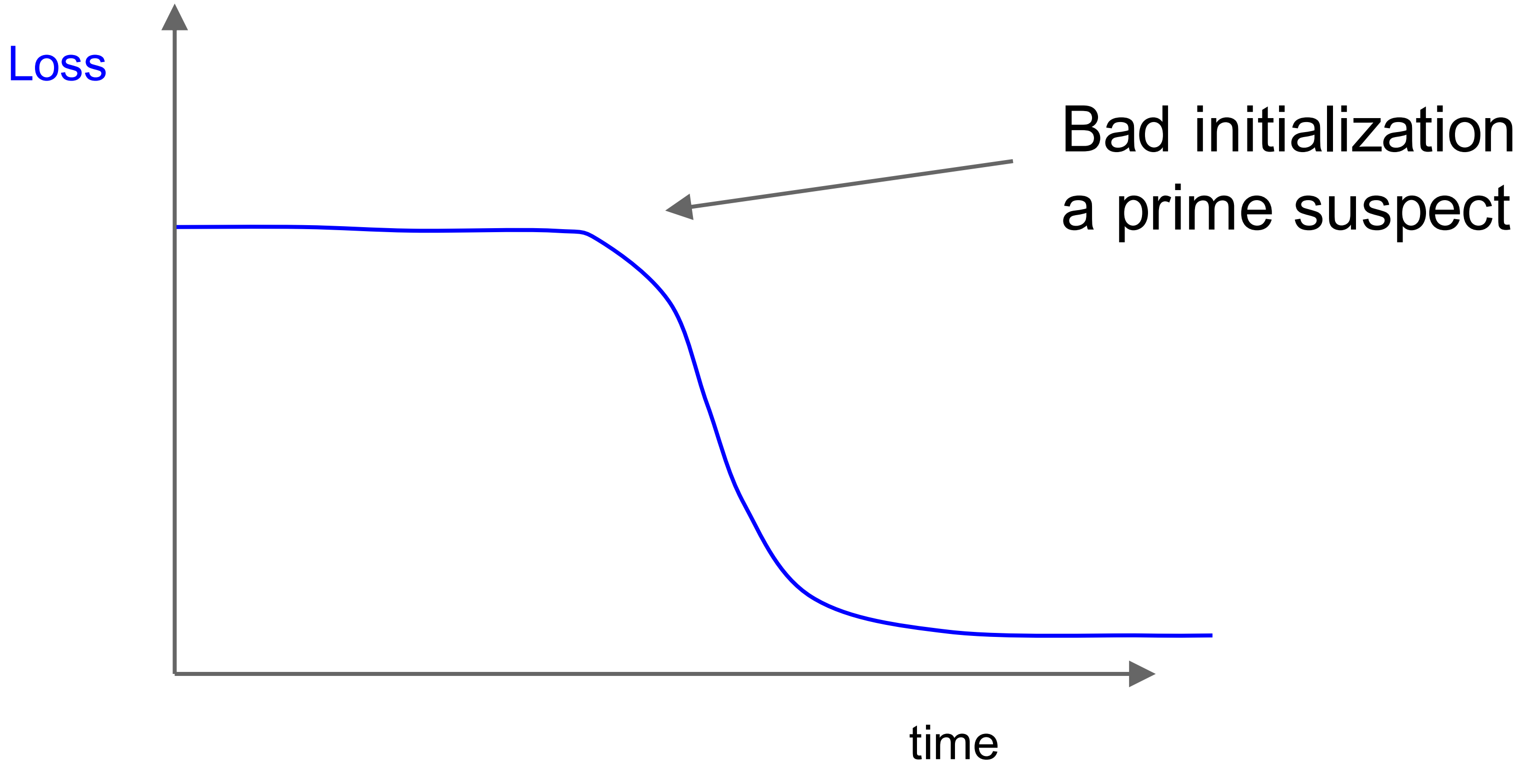
- Network architecture
- Learning rate, its decay schedule, update type
- Algorithm: SGD, SGD with momentum, Adam
- Regularization (L2 / Dropout strength)

Neural networks practitioner
music = loss function

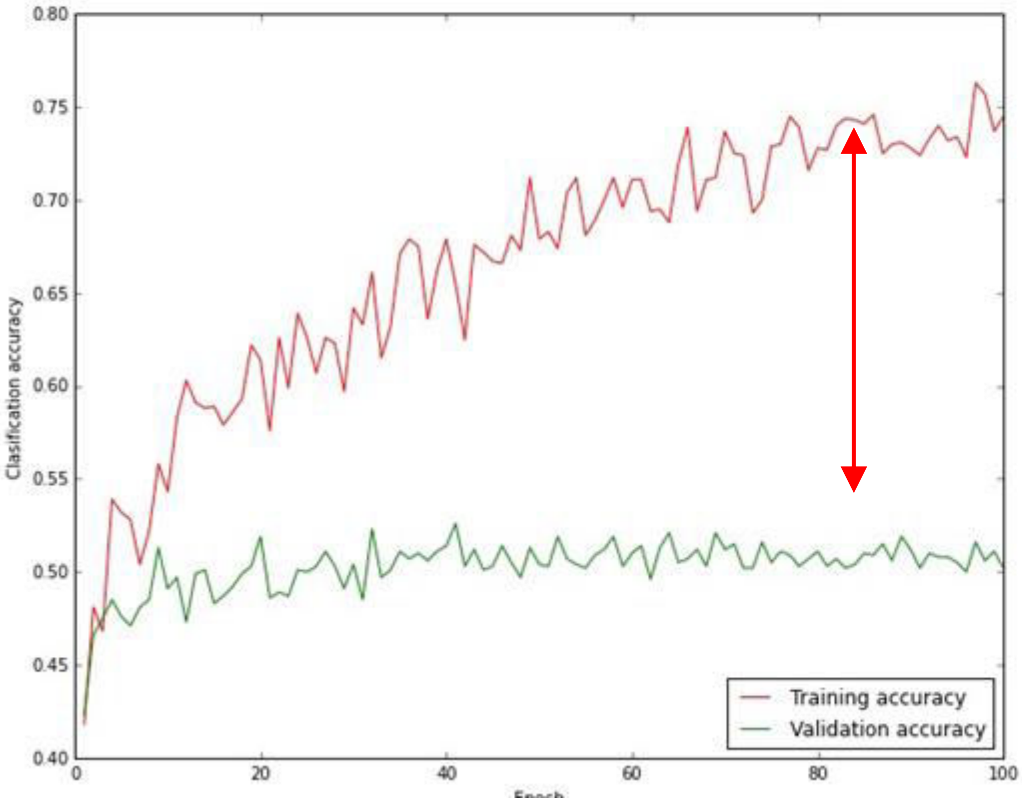
Monitor and visualize the loss curve






Loss
Bad initialization
a prime suspect
time

Monitor and visualize the accuracy



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

TLDRs

Summary

We looked in detail at:

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization
  (random sampling, in log space when appropriate)

Training Algorithm

There are multiple training algorithms

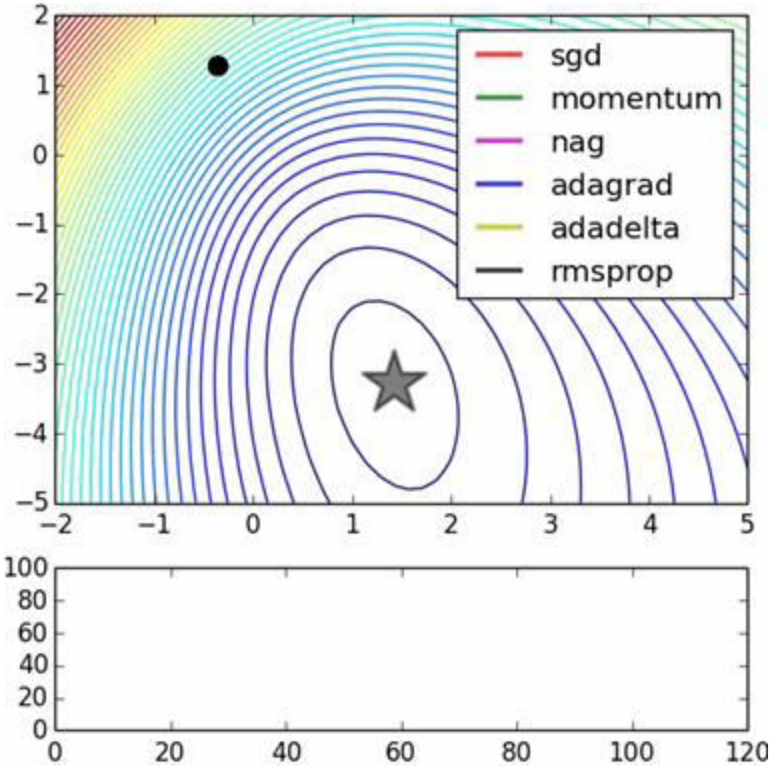
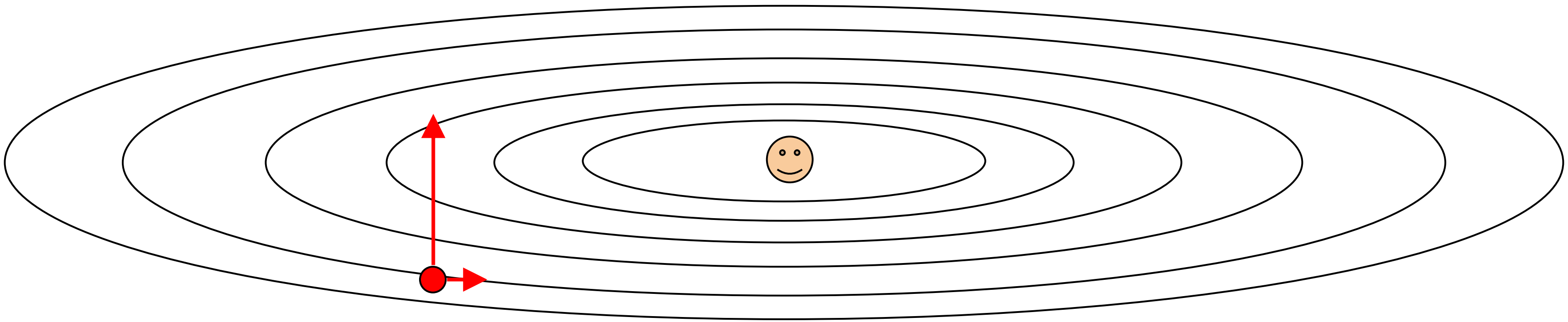


Image credits: Alec Radford

Gradient Descent (Python)

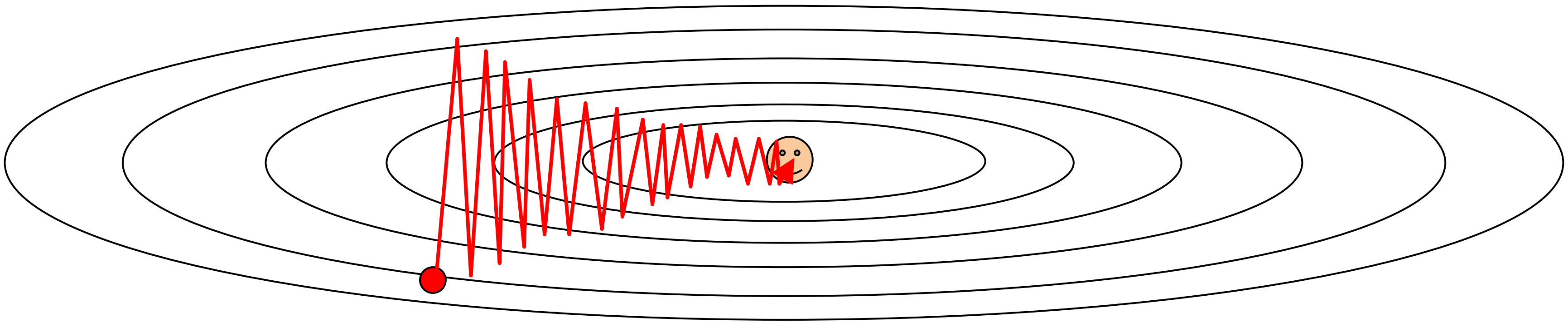
```
def GD(W0, X, goal, learningRate):
    perfGoalNotMet = true
    W = W0

    while perfGoalNotMet:
        gradient = eval_gradient(X, W)
        W_old = W
        W = W – learningRate * gradient
        perfGoalNotMet = sum(abs(W - W_old)) > goal
```

Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD?

Suppose loss function is steep vertically but shallow horizontally:



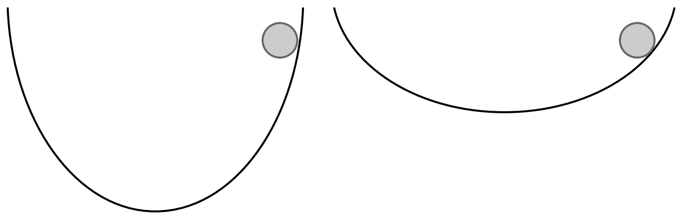
Q: What is the trajectory along which we converge towards the minimum with SGD? Very slow progress along flat direction, jitter along steep one

SGD with momentum (Python)

```
def GD(W0, X, goal, learningRate, mu):
   perfGoalNotMet = true
   W = W0
   V = 0
   while perfGoalNotMet:
      gradient = eval_gradient(X, W)
      W_old = W
      V = mu * V – learningRate * gradient
      W = W + V

      perfGoalNotMet = sum(abs(W - W_old)) > goal
```

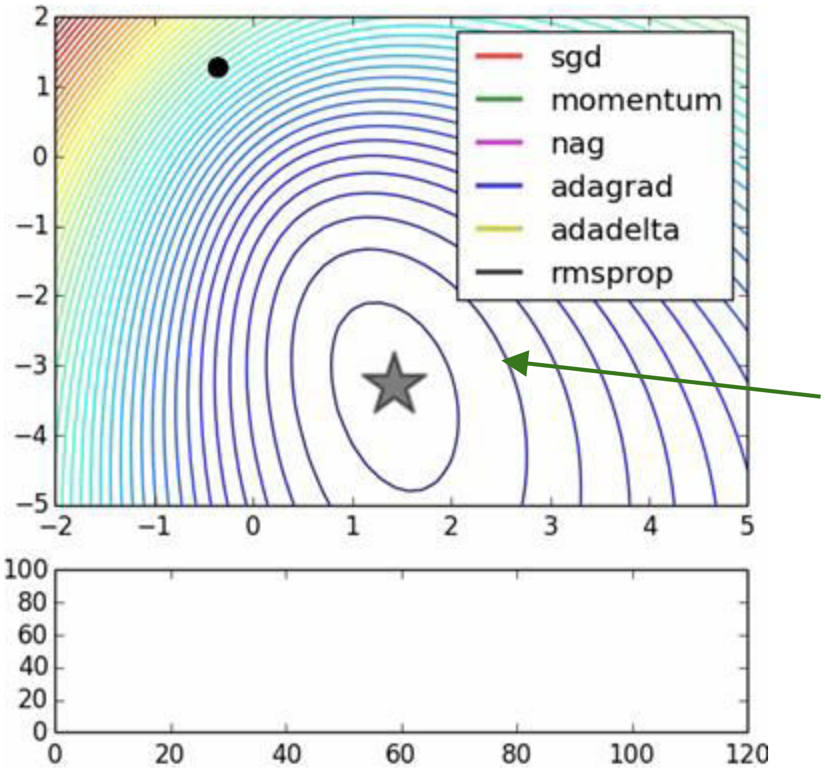
SGD with momentum

- Physical interpretation as ball rolling down the loss function + friction (mu coefficient)
- mu = usually ~0.5, 0.9, or 0.99 (sometimes annealed over time, e.g. from 0.5 -> 0.99)



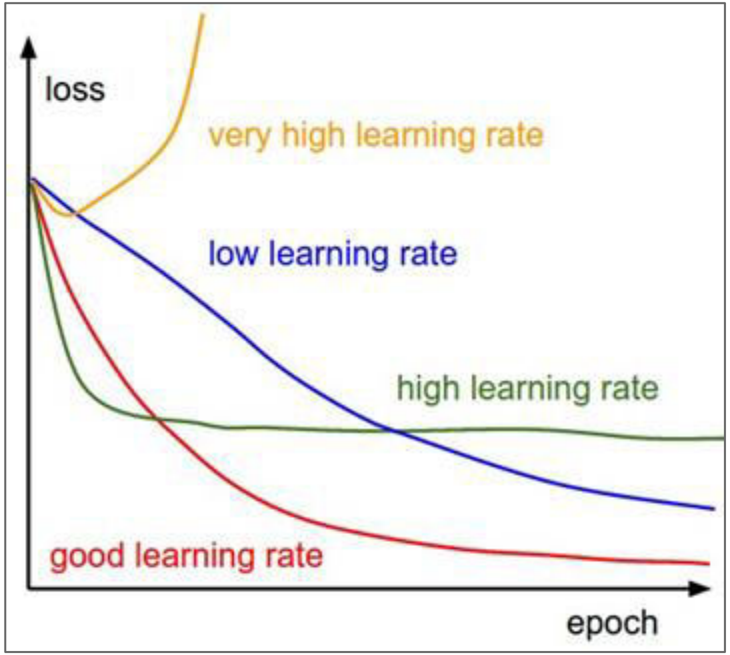
- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

SGD vs. SGD with momentum



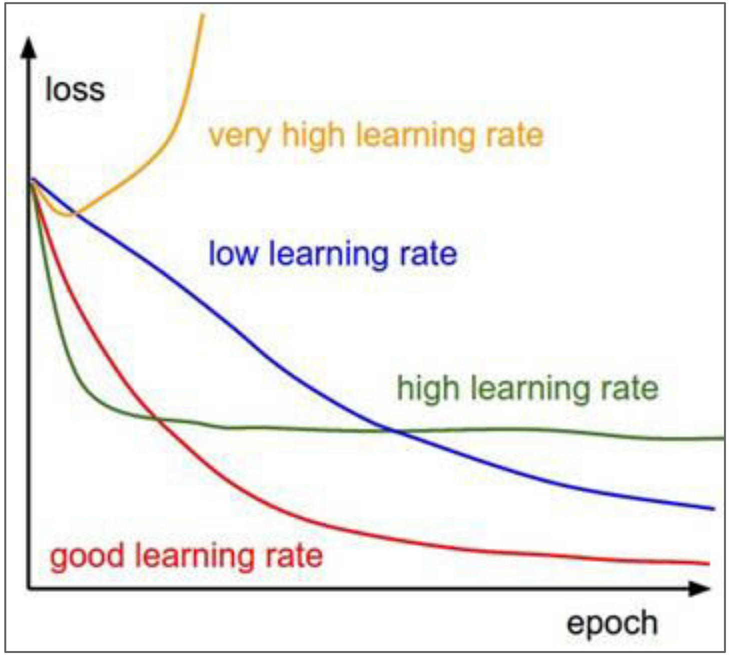
Notice momentum overshooting the target, but overall getting to the minimum much faster.

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs

exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Evaluation:

Model Ensembles

Model Ensembles

1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance

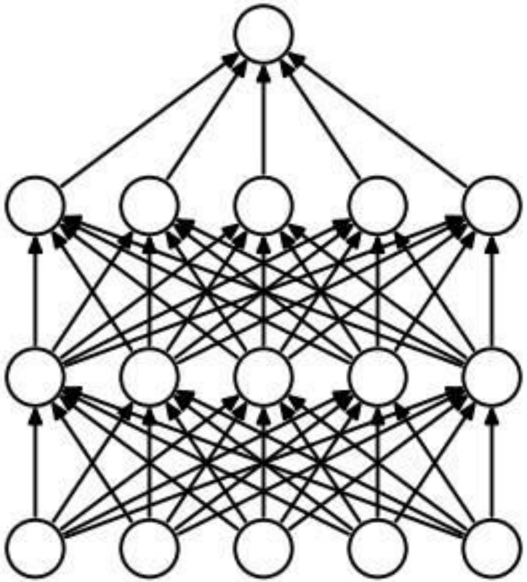
Tips & Tricks

- Can also get a small boost from averaging multiple model checkpoints of a single model
- Keep track of (and use at test time) a running average parameter vector

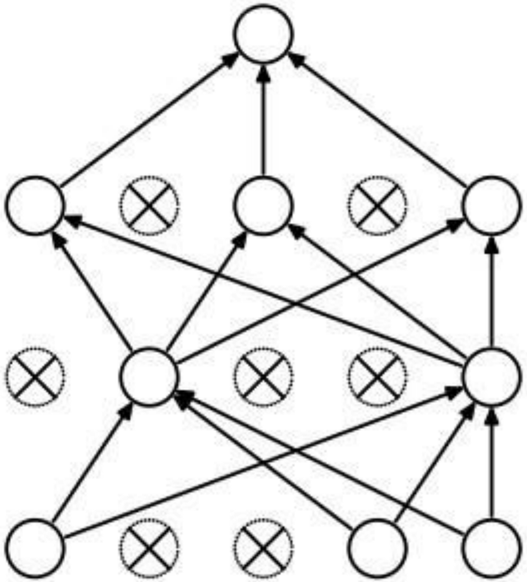
Regularization with **Dropout**

Regularization: **Dropout**

"randomly set some neurons to zero in the forward pass"



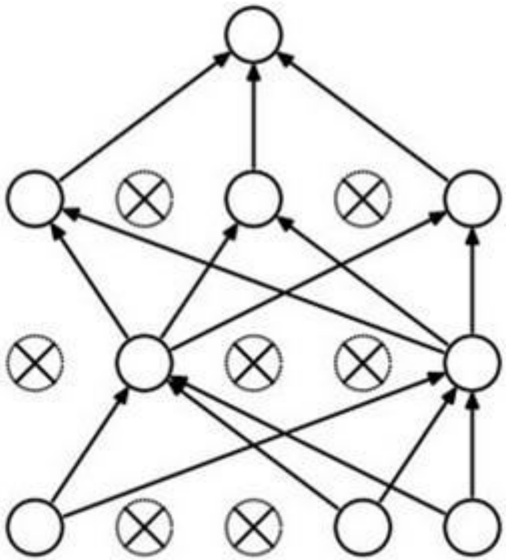
(a) Standard Neural Net



(b) After applying dropout.

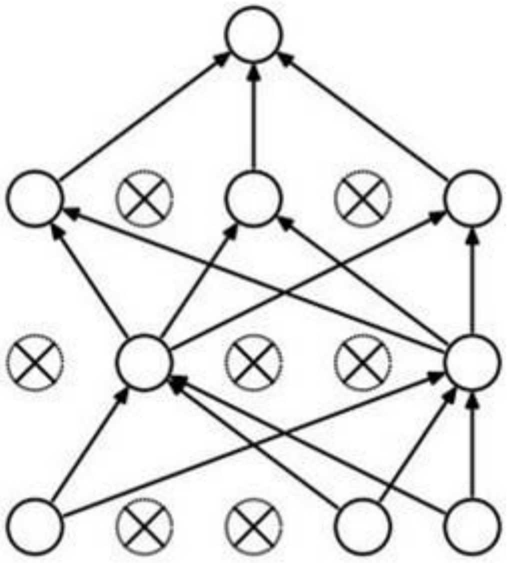
[Srivastava et al., 2014]

Waaaait a second…
How could this possibly be a good idea?



Forces the network to have a redundant representation.

Waaaait a second…
How could this possibly be a good idea?

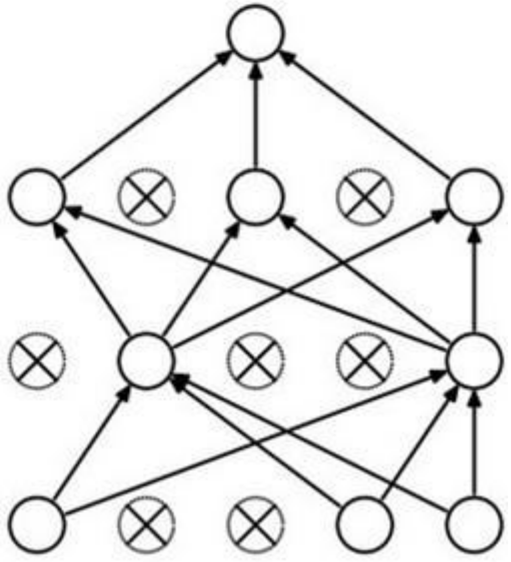


Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained on only ~one datapoint.

At test time….

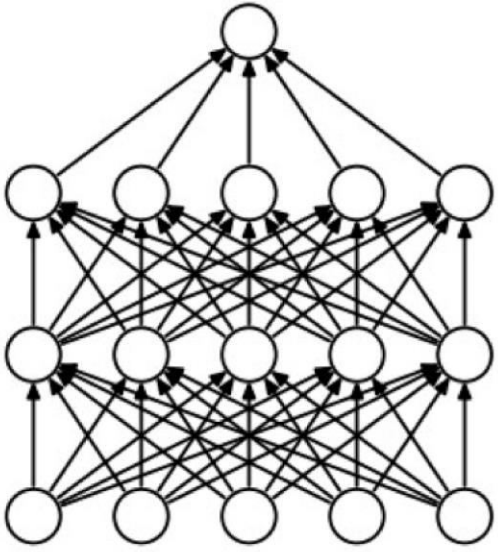


Ideally:
We want to integrate out all the noise

Monte Carlo approximation:
Do many forward passes with different dropout masks, average all predictions

At test time….

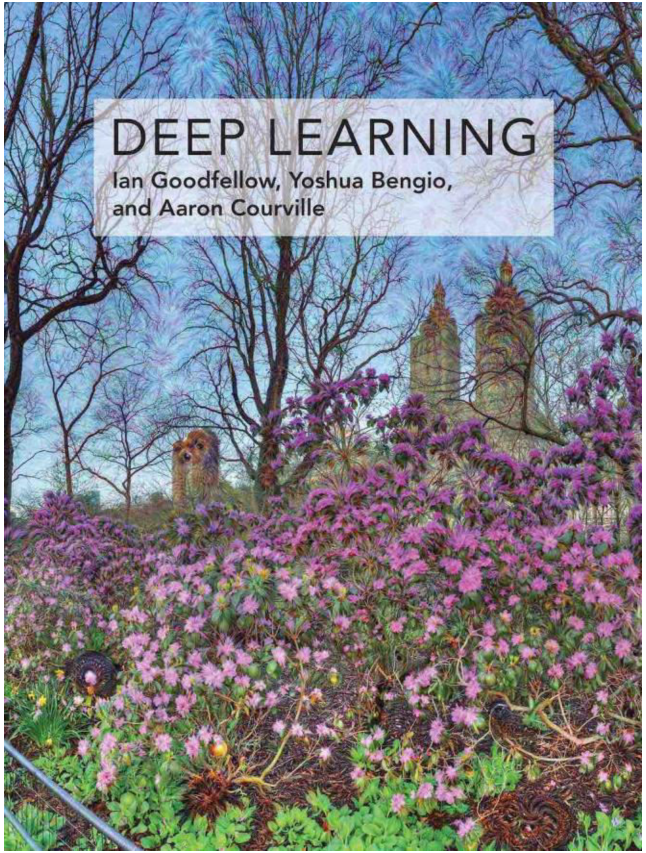
Can in fact do this with a single forward pass! (approximately)



Leave all input neurons turned on (no dropout).

(this can be shown to be an approximation to evaluating the whole ensemble)

Bibliography


DEEP LEARNING
Ian Goodfellow, Yoshua Bengio,
and Aaron Courville

O'REILLY

Hands-On Machine Learning with Scikit-Learn & TensorFlow

CONCEPTS, TOOLS, AND TECHNIQUES TO BUILD INTELLIGENT SYSTEMS

powered by jupyter

Aurélien Géron