Deep Learning

Alexander Panin, Andrey Ustyuzhanin, Alena Zarodnyuk

2021, LaSalle University

LAMBDA • HSE

Recap of day3

- KNN classification
- Random Forest
- Boosting, Bagging
- ROC curve, AUC
- X-validation
- Regularization
- Decision Tree
- Overfitting
- Loss function
- Purity of a leaf
- Accuracy/Recall/Precision
- Confusion Matrix
- RMSE

Linear Regression

Model: $x \longrightarrow wx + b \longrightarrow y^{pred}$

Objective function: $$L = \frac{1}{N} \sum_i (y_i - y_i^{pred})^2$$

Optimization (exact): $$w = (X^T \cdot X)^{-1} X^T y$$

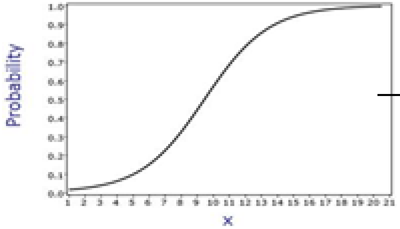
Linear Regression

Model: $x \longrightarrow wx + b \longrightarrow y^{pred}$

Objective function: $$L = \frac{1}{N} \sum_i (y_i - y_i^{pred})^2$$

Optimization (iterative): $$w_{i+1} := w_i - \alpha \cdot \frac{\partial L}{\partial w_i}$$

Logistic Regression

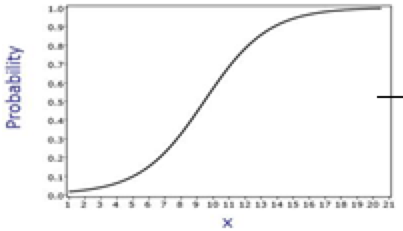
X → wx +b →  → p(y|x)

$$p(y|x) = \sigma(Wx + b) = \frac{1}{1 + e^{-(Wx+b)}}$$

Objective function ?

Logistic Regression

Model:

X ⟶ wx +b ⟶ 

⟶ P(y)

Objective function:

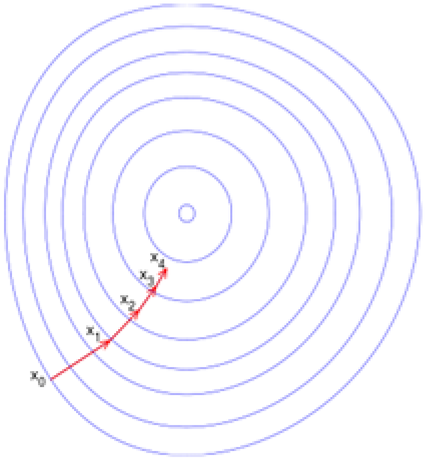
$$L = -\frac{1}{N}\sum_i y_i \cdot \log p(y|x_i) + (1 - y_i) \cdot \log(1 - p(y|x_i)))$$

Optimization(iterative): **same as linear regression**

Recap: Gradient Descent

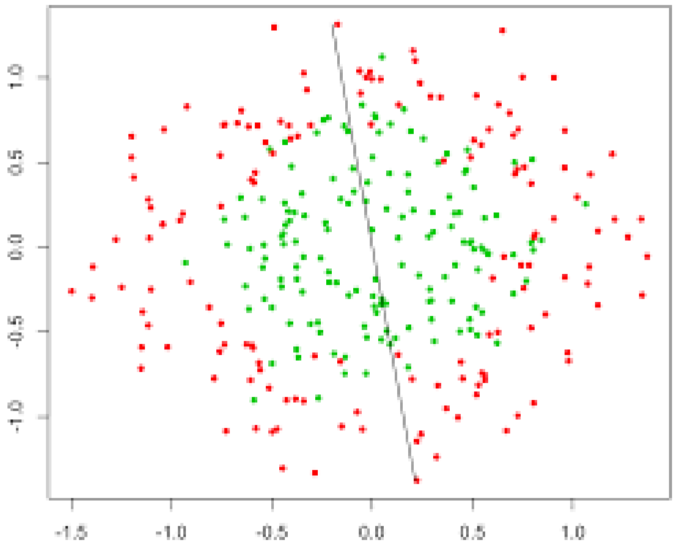
Update:

$$w_{i+1} := w_i - \alpha \cdot \frac{\partial L}{\partial w_i}$$

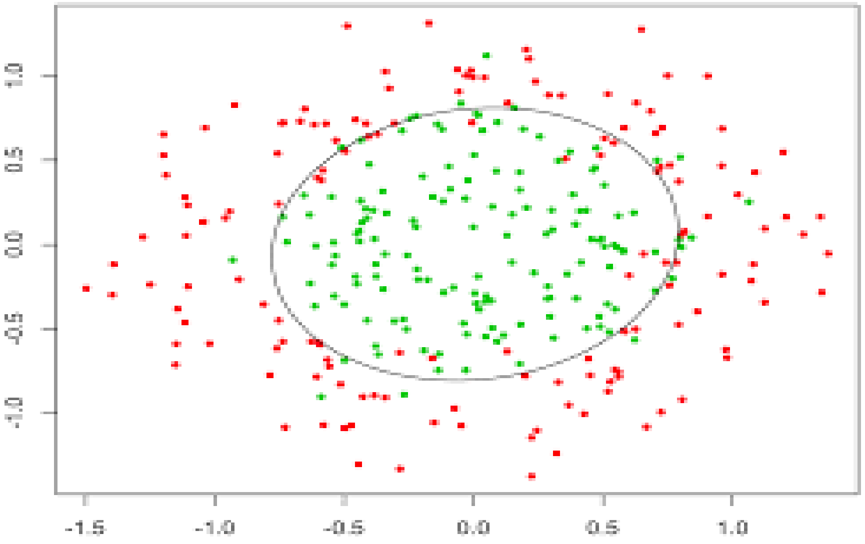


- a – learning rate a<<1
- L – loss function

Nonlinear dependencies



What we have



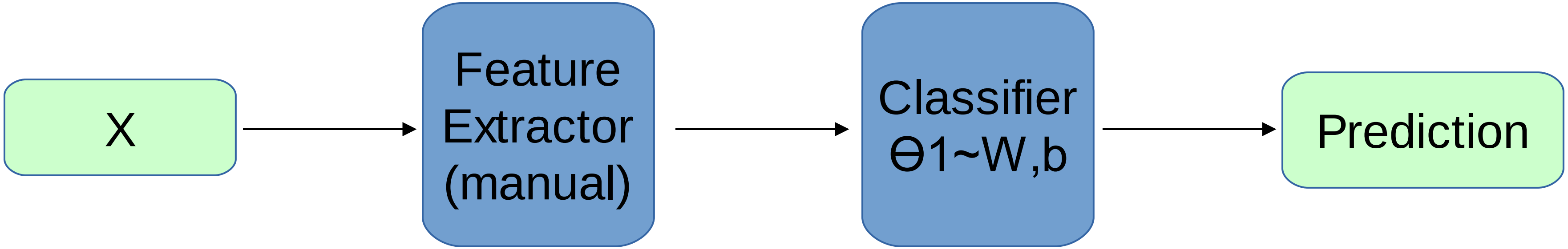
What we want

How to learn that?

Feature extraction

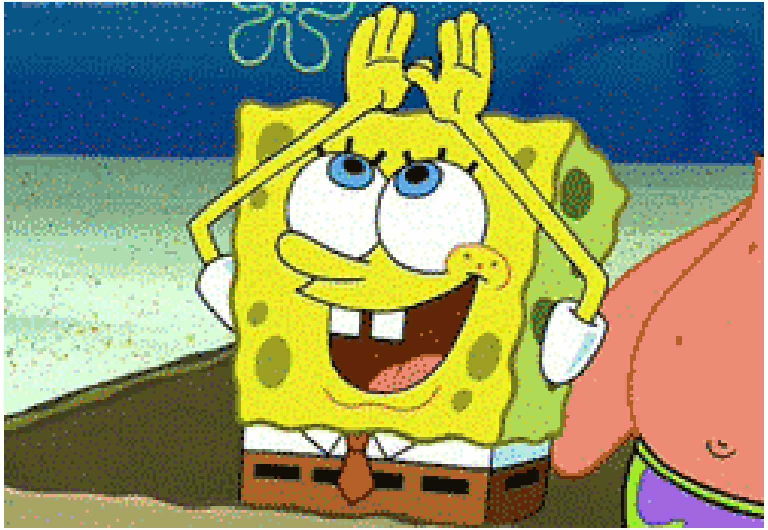
Loss: same as linear/logistic regression

Model:



Training: $$\underset{\theta_1}{argmin} \quad L$$

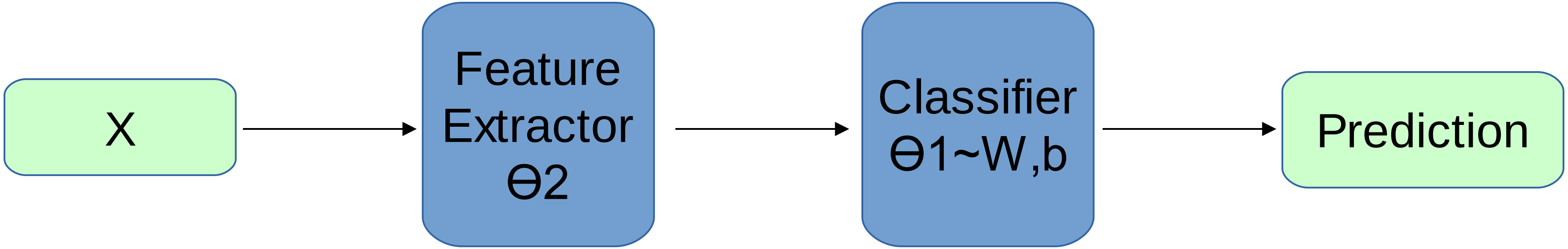
What if...



Features would tune to your problem automagically!

What do we want, exactly?

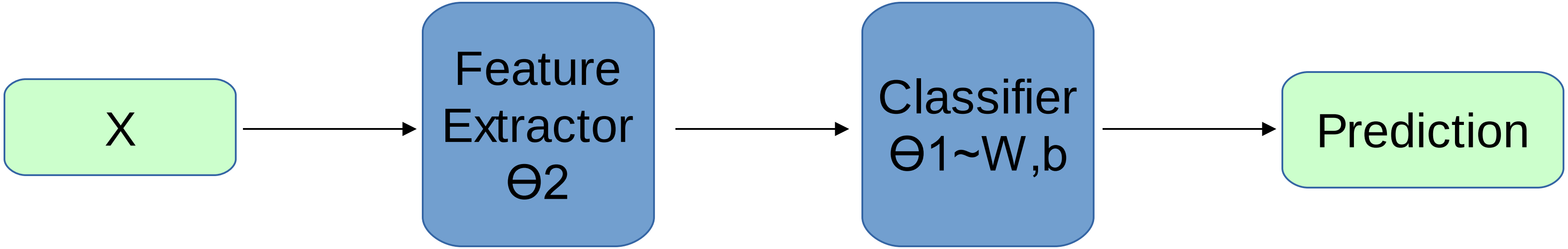
Model:



Objective: **?** $\underset{\theta_1}{argmin} \quad L$

What do we want, exactly?

Model:

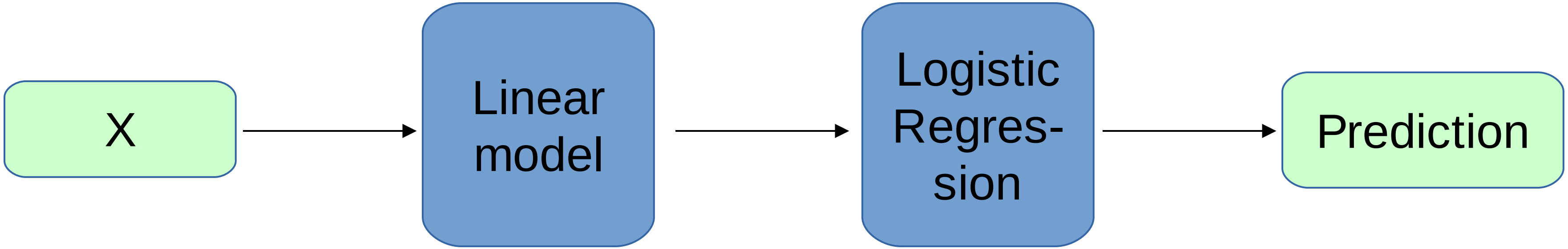


Joint training: $\underset{\theta_1,\theta_2}{argmin} \quad L$

Okay, how do we extract features?

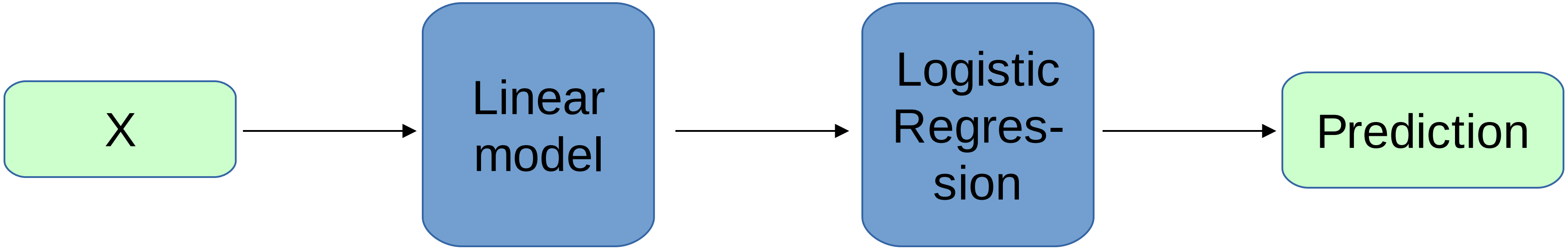
Try linear

Model:



Try linear

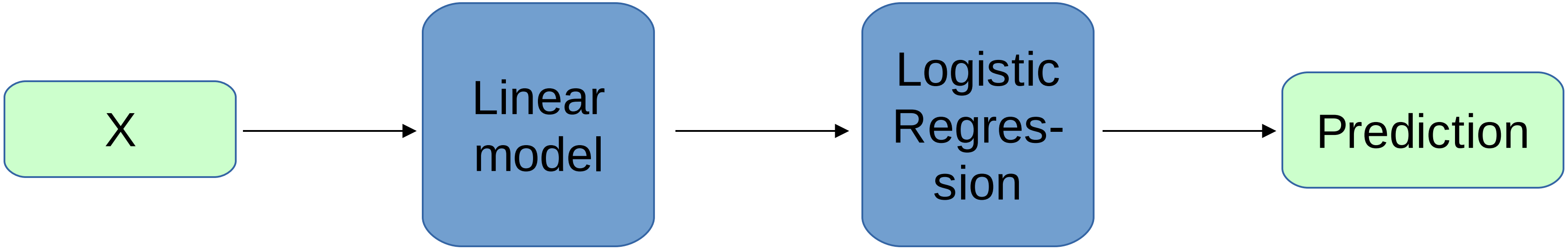
Model:



$$h_j = \sum_i w_{ij}^h x_i + b_j^h \qquad \sigma(\sum_j w_j h_j + b_j)$$

Try linear

Model:



$$h_j = \sum_i w^h_{ij} x_i + b^h_j \qquad \sigma\Big(\sum_j w_j h_j + b_j\Big)$$

Output: $$p(y|x) = \sigma\Big(\sum_j w_j\Big(\sum_i w^h_i x_i + b^h_i\Big) + b_j\Big)$$

Is it any better than logistic regression?

Try linear

$$p(y|x) = \sigma(\sum_j w_j(\sum_i w_i^h x_i + b_i^h) + b_j)$$

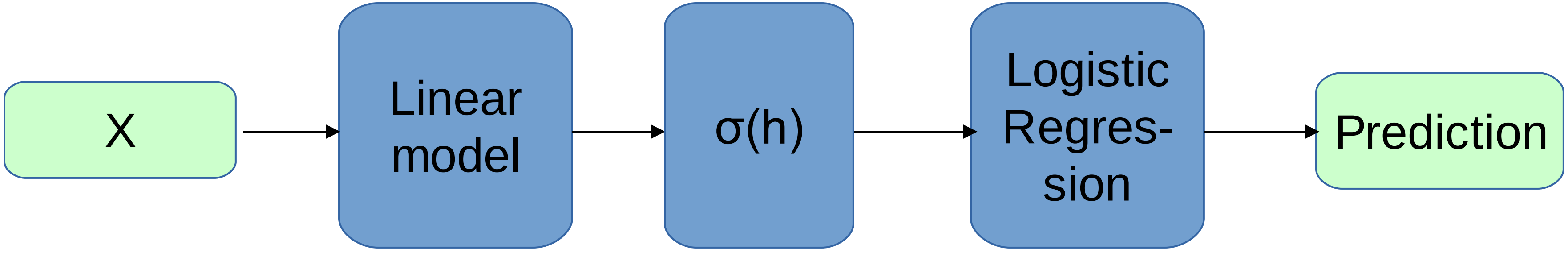
$$p(y|x) = \sigma(\sum_i [\sum_j w_j w_i^h] x_i + \sum_i b_i^h + b_j)$$

$$\hat{w}_i = [\sum_j w_j w_i^h] x_i; \quad \hat{b} = \sum_i b_i^h + b_j$$

$$p(y|x) = \sigma(\sum_i \hat{w}_i x_i + \hat{b}_i)$$

Add nonlinearity

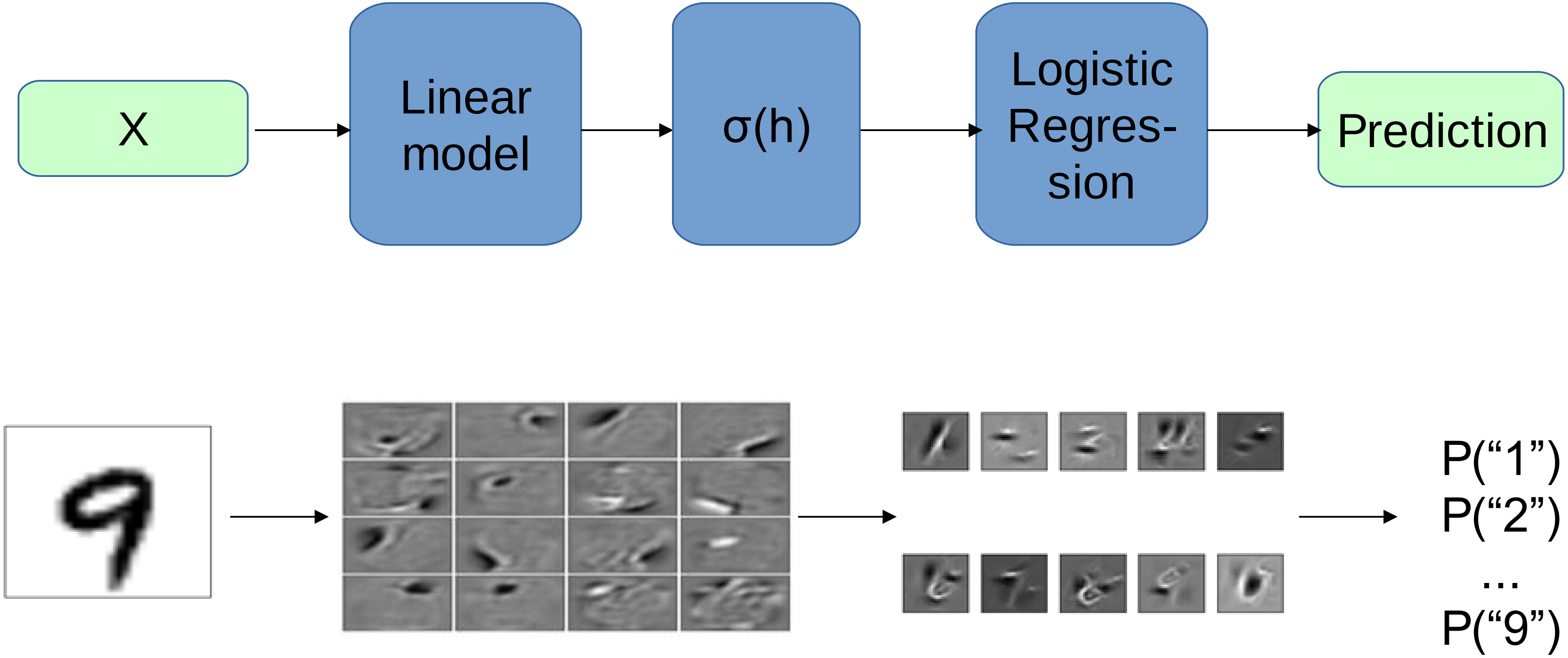
Model:



$$p(y|x) = \sigma(\sum_j w_j \sigma(\sum_i w_i^h x_i + b_i^h) + b_j)$$

Add nonlinearity

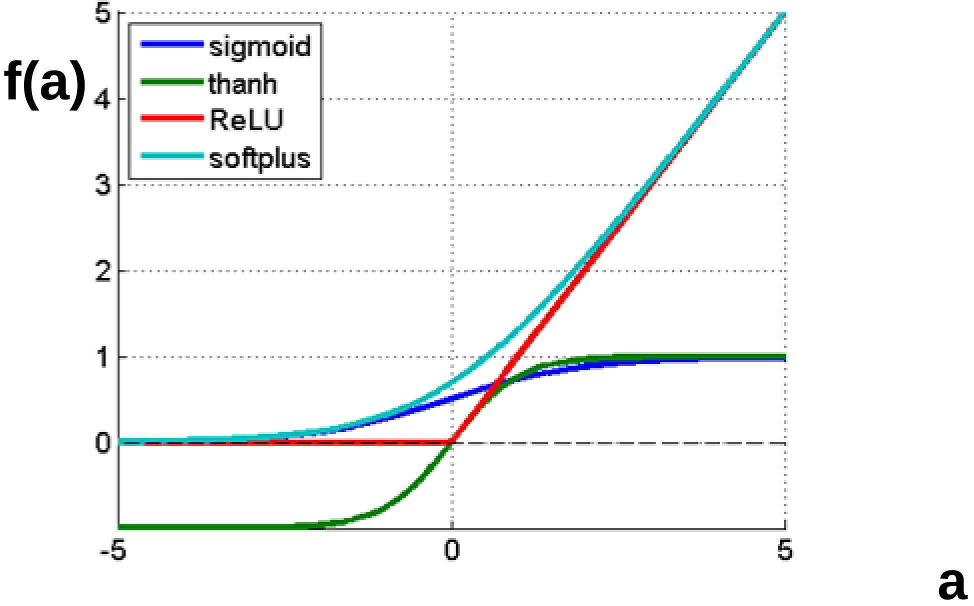
Model:



Add nonlinearity

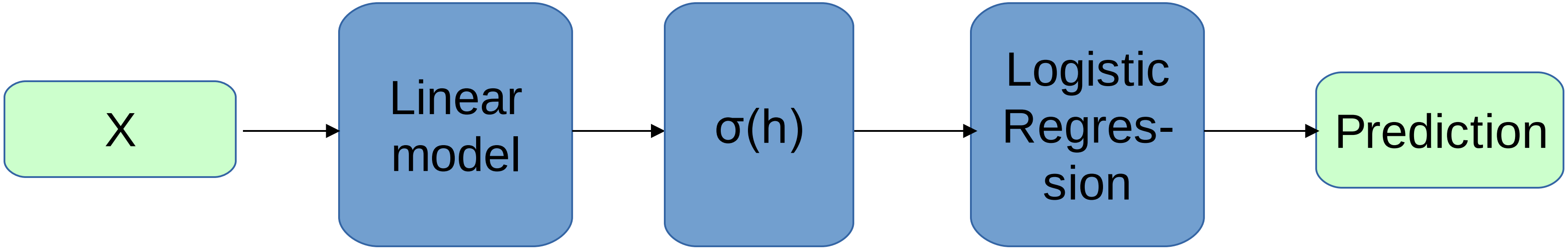
- f(a) = 1/(1+e^a)
- f(a) = tanh(a)

- f(a) = max(0,a)
- f(a) = log(1+e^a)



Training the monster

Model:

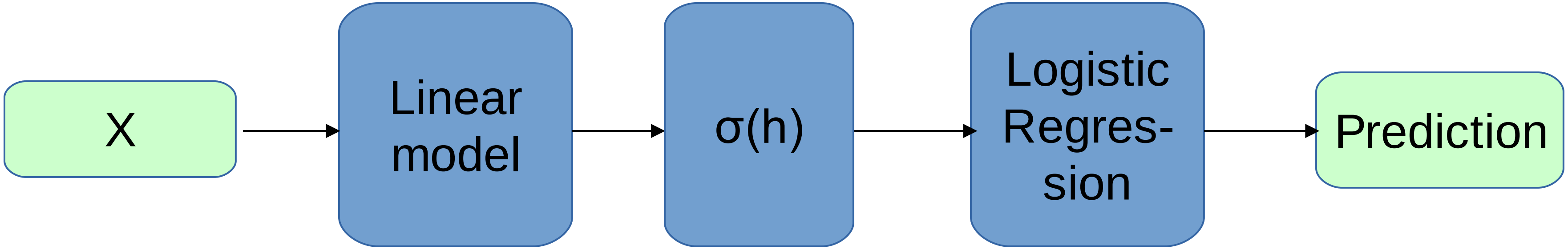


Output: $$p(y|x) = \sigma(\sum_j w_j \sigma(\sum_i w_i^h x_i + b_i^h) + b_j)$$

Training: **?!**

Training the monster

Model:



Output: $p(y|x) = \sigma(\sum_j w_j \sigma(\sum_i w_i^h x_i + b_i^h) + b_j)$

Training: gradient descent! $w_{i+1} := w_i - \alpha \cdot \frac{\partial L}{\partial w_i}$

Backpropagation

- **TL;DR:** backprop = chain rule*

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x}$$

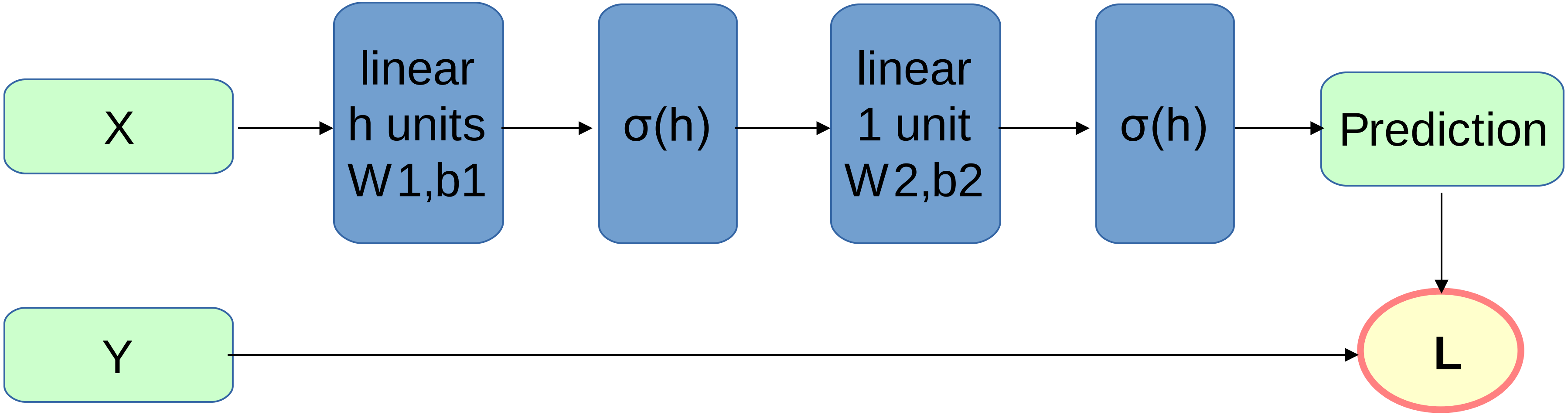
Backpropagation

- **TL;DR:** backprop = chain rule*

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x}$$

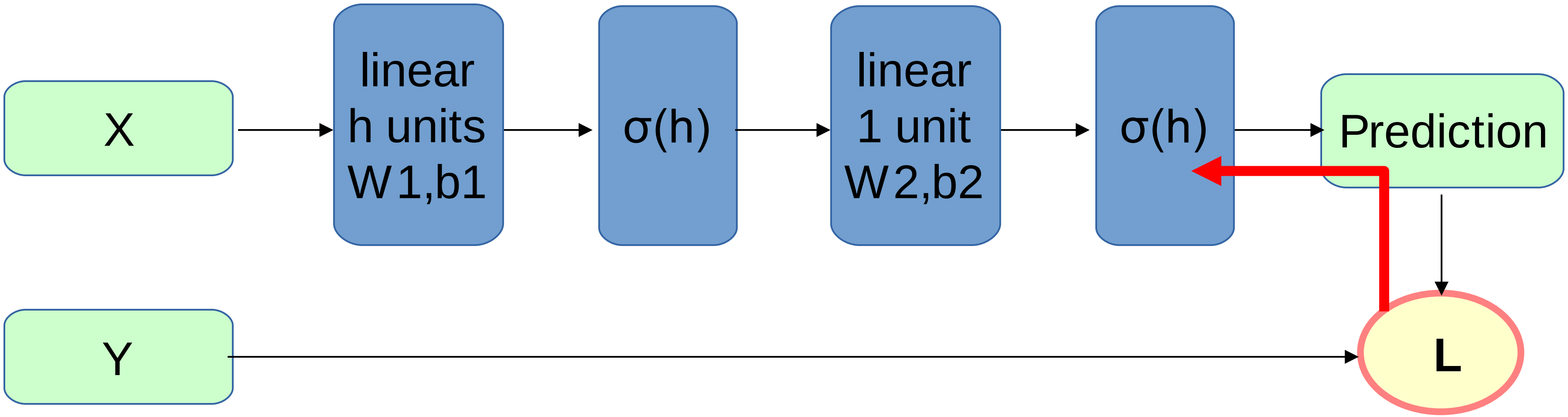
* g and x can be vectors/vectors/tensors

Backpropagation



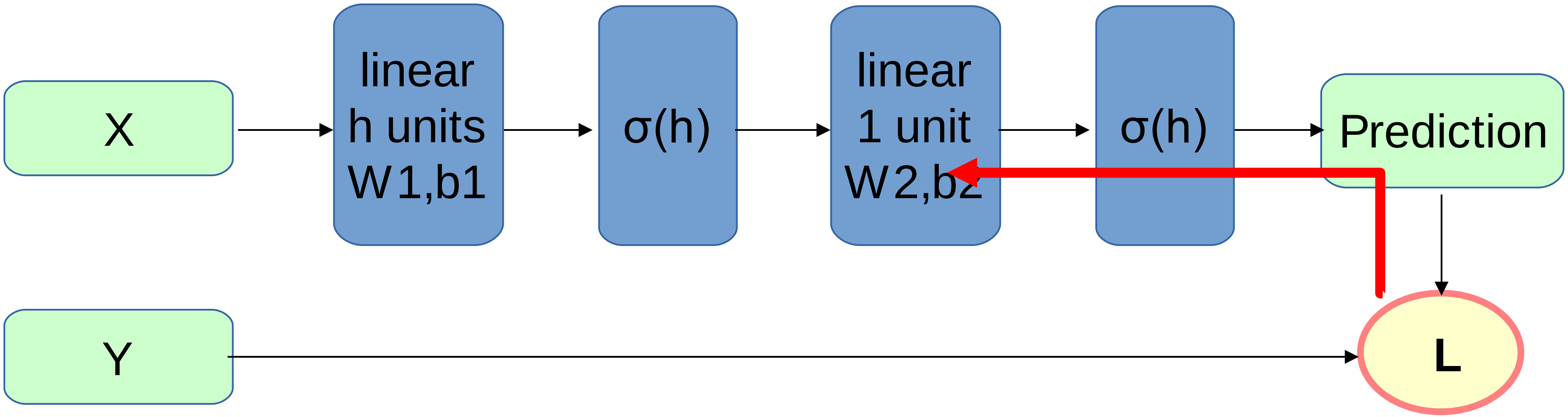
$$\frac{\partial L(linear_2(\sigma(linear_1(x))))}{\partial W_1} = \ldots$$

Backpropagation



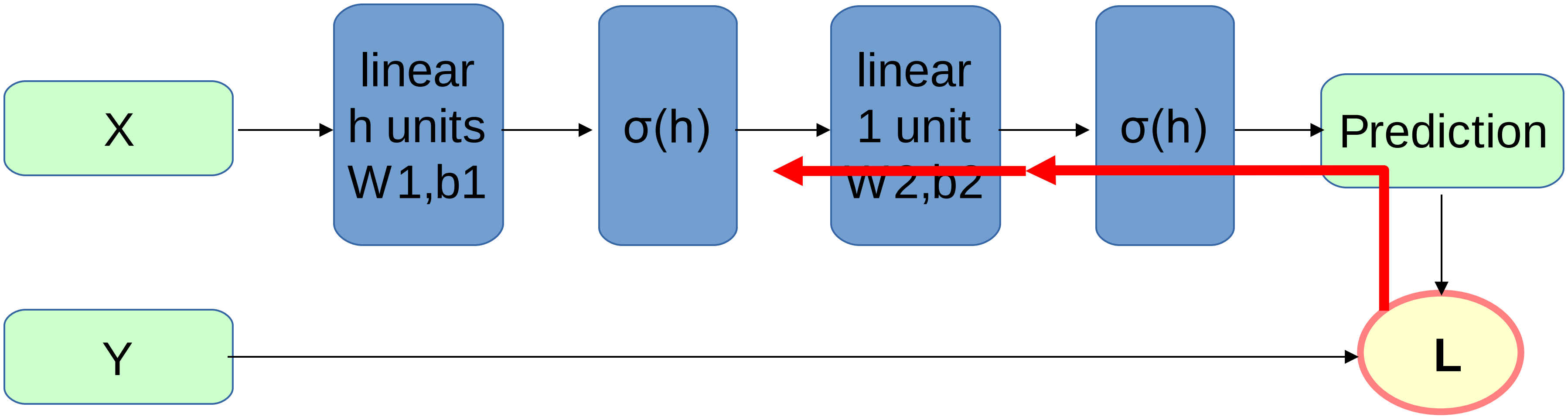
$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \sigma_2} \cdot$$

Backpropagation



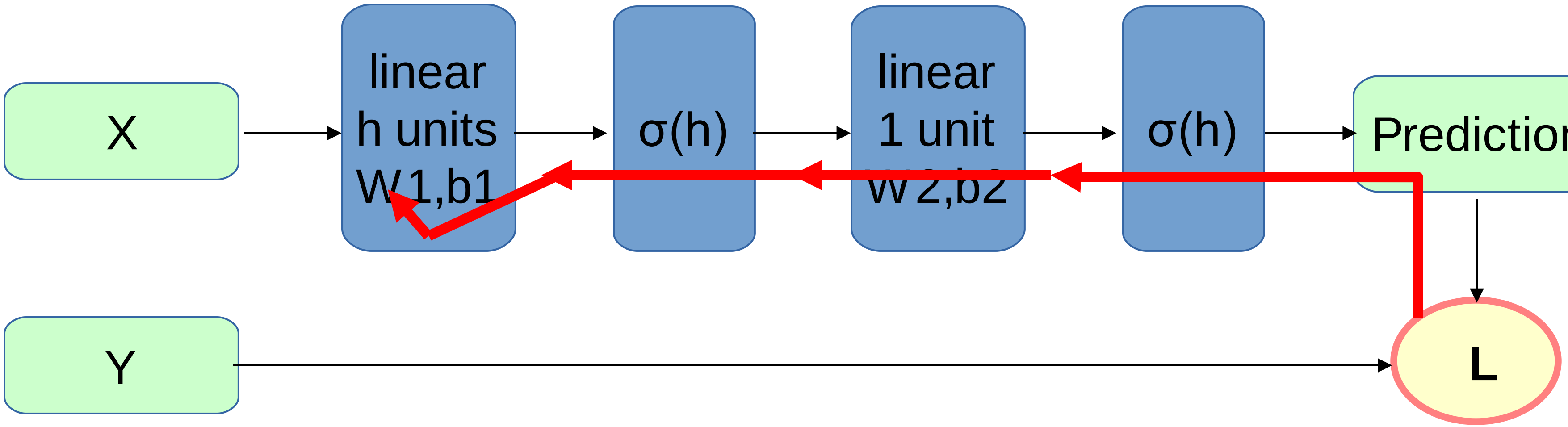
$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \sigma_2} \cdot \frac{\partial \sigma_2}{\partial linear_2} \cdot$$

Backpropagation



$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \sigma_2} \cdot \frac{\partial \sigma_2}{\partial linear_2} \cdot \frac{\partial linear2}{\partial \sigma_1} \cdot \cdot$$

Backpropagation



$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \sigma_2} \cdot \frac{\partial \sigma_2}{\partial linear_2} \cdot \frac{\partial linear2}{\partial \sigma_1} \cdot \frac{\partial \sigma_1}{\partial linear_1} \cdot \frac{\partial linear_1}{\partial W_1}$$

Matrix derivatives

Let's compute:

$$\frac{\partial L(X \times W + b)}{\partial X} = \frac{\partial L(X \times W + b)}{\partial [X \times W + b]} \times \boxed{\textbf{What?}}$$

Variable shapes:

X	W	b
[batch size, features]	[features, outputs]	[outputs]

$$\frac{\partial L(X \times W + b)}{\partial X}$$

[batch size, features]

$$\frac{\partial L(X \times W + b)}{\partial [X \times W + b]}$$

[batch size, outputs]

Matrix derivatives

Hint: 1. figure out scalar case,
2. match shapes for matrices

Let's compute:

$$\frac{\partial L(X \times W + b)}{\partial X} = \frac{\partial L(X \times W + b)}{\partial [X \times W + b]} \times \boxed{\textbf{What?}}$$

Variable shapes:

X [batch size, features]

W [features, outputs]

b [outputs]

$\frac{\partial L(X \times W + b)}{\partial X}$ [batch size, features]

$\frac{\partial L(X \times W + b)}{\partial [X \times W + b]}$ [batch size, outputs]

Matrix derivatives

Let's compute:

$$\frac{\partial L(X \times W + b)}{\partial X} = \frac{\partial L(X \times W + b)}{\partial [X \times W + b]} \times W^T$$

Variable shapes:

X [batch size, features]

W [features, outputs]

b [outputs]

$\frac{\partial L(X \times W + b)}{\partial X}$ [batch size, features]

$\frac{\partial L(X \times W + b)}{\partial [X \times W + b]}$ [batch size, outputs]

Matrix derivatives

Let's compute:

$$\frac{\partial L(X \times W + b)}{\partial W} =$$ **What?**

Variable shapes:

X [batch size, features]

W [features, outputs]

b [outputs]

$$\frac{\partial L(X \times W + b)}{\partial X}$$

[batch size, features]

$$\frac{\partial L(X \times W + b)}{\partial [X \times W + b]}$$

[batch size, outputs]

Matrix derivatives

Let's compute:

$$\frac{\partial L(X \times W + b)}{\partial W} = X^T \times \frac{\partial L(X \times W + b)}{\partial [X \times W + b]}$$

Variable shapes:

X [batch size, features]

W [features, outputs]

b [outputs]

$\frac{\partial L(X \times W + b)}{\partial X}$ [batch size, features]

$\frac{\partial L(X \times W + b)}{\partial [X \times W + b]}$ [batch size, outputs]

Cheat sheet for seminar

Gradient of $\sum_i \log p(y_i|x_i, w) = \sum_i \text{gradient} \log p(y_i|x_i, w)$

linear over X : $\dfrac{\partial L}{\partial[X \times W + b]} \times W^T$

linear over W : $\dfrac{1}{||X||} \cdot X^T \times \dfrac{\partial L}{\partial[X \times W + b]}$

sigmoid : $\dfrac{\partial L}{\partial \sigma(x)} \cdot [\sigma(x) \cdot (1 - \sigma(x))]$

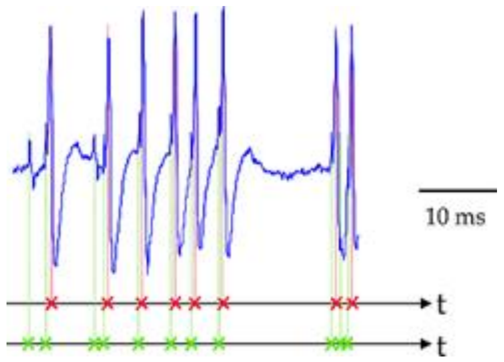
Works for any kind of x
(scalar, vector, matrix, tensor)

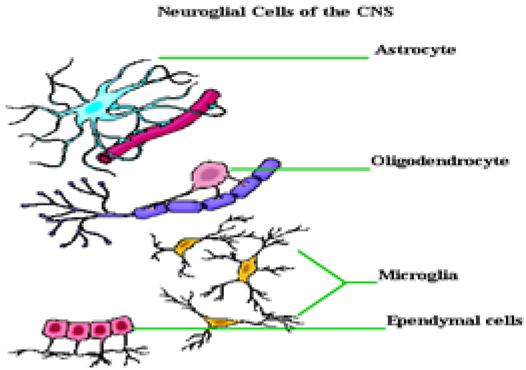
Not actual neurons :)

- Neurons output in “spikes”,
  not real numbers

- No one knows for sure
  how they “train”

- There are other cell types
  e.g. neuroglial cells

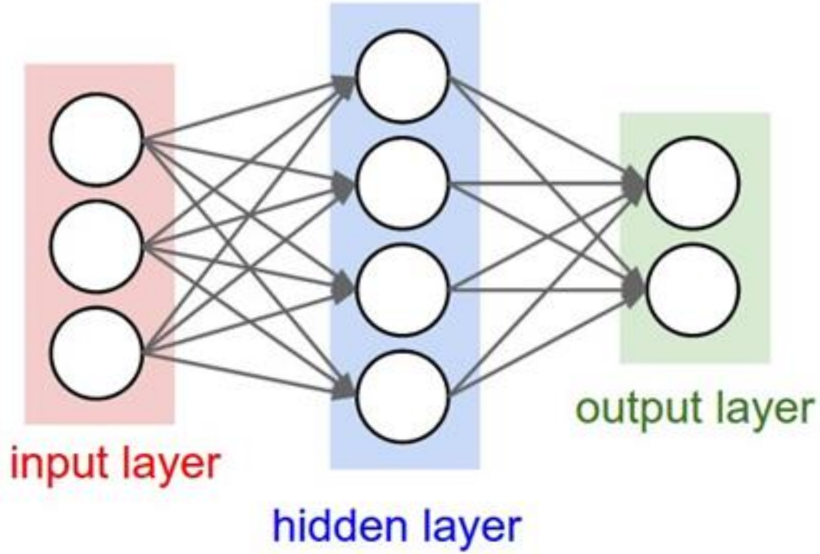




Connectionist phrasebook

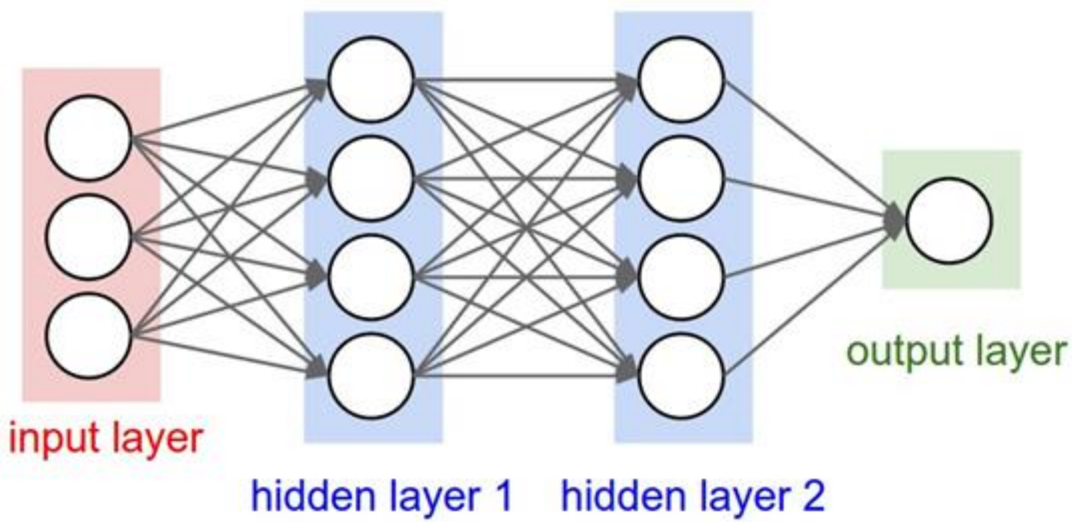
- Layer – a building block for NNs :
  - “Dense layer”: $f(x) = Wx+b$
  - “Nonlinearity layer”: $f(x) = \sigma(x)$
  - Input layer, output layer
  - A few more we gonna cover later
- Activation – layer output
  - i.e. some intermediate signal in the NN
- Backpropagation – a fancy word for “chain rule”

Connectionist phrasebook



"Train it via backprop!"

Connectionist phrasebook



How do we train it?

Potential caveats?

Problems with deep learning

- Hardcore overfitting

- No “golden standard” for architecture

- Computationally heavy

Back to neural networks

Model:

X → linear h units W1,b1 → σ(h) → linear 1 unit W2,b2 → σ(h) → Prediction

Training:

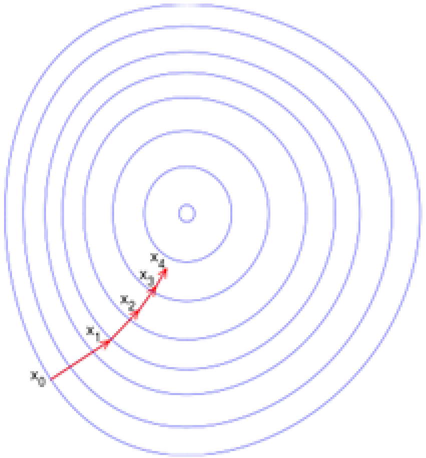
see the demo

Gradient Descent

Update:

$$w_{i+1} := w_i - \alpha \cdot \frac{\partial L}{\partial w_i}$$

- a – learning rate a<<1
- L – loss function

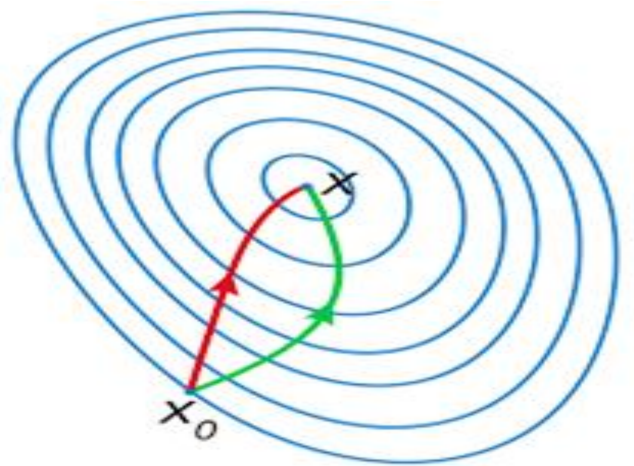


Can we do better?

Newton-Raphson

Parameter update

$$w_{i+1} = w_i - \alpha \cdot H^{-1} \frac{\partial L}{\partial w_i}$$



Green: Gradient Descent
Red: Newton-Raphson

Hessian:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Any drawbacks?

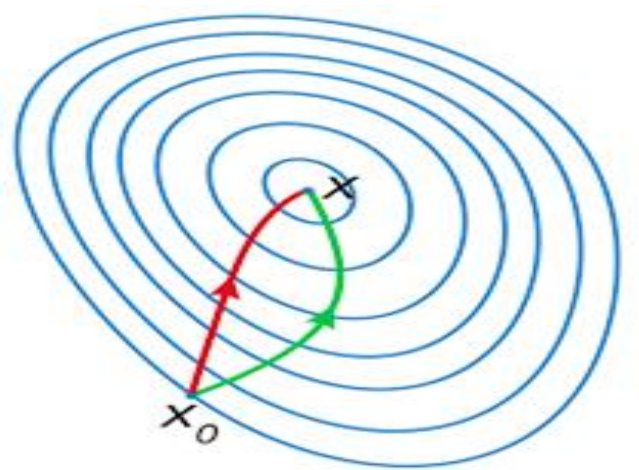
Newton-Raphson

Parameter update

$$w_{i+1} = w_i - \alpha \cdot H^{-1} \frac{\partial L}{\partial w_i}$$

Hessian:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$



Green: Gradient Descent
Red: Newton-Raphson

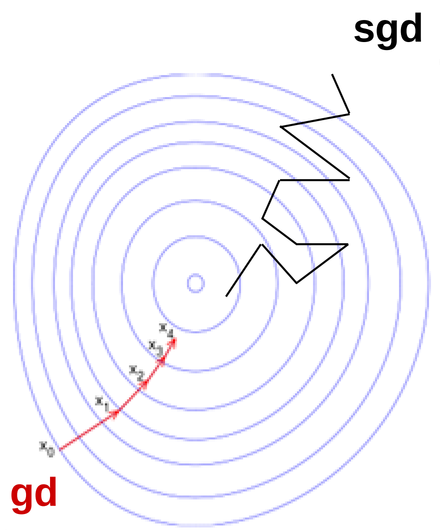
Inverting H might be infeasible

Stochastic gradient descent

Approximate objective with samples

$$L = \frac{1}{N}\sum_i f(x_i, y_i, w) = \underset{i\sim U(1,N)}{E} f(x_i, y_i, w)$$

$$L \approx f(x_i, y_i, w)\big|_{i\sim U(1,N)}$$

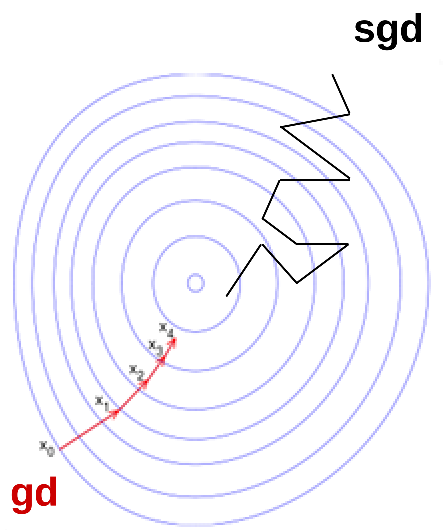


Stochastic gradient descent

Approximate objective with samples

$$L = \frac{1}{N} \sum_i f(x_i, y_i, w) = \underset{i \sim U(1,N)}{E} f(x_i, y_i, w)$$

$$L \approx f(x_i, y_i, w)|_{i \sim U(1,N)}$$



Update:

$$w_{i+1} = w_i - \alpha \cdot \frac{\partial \hat{L}}{\partial w}$$ where $\hat{L} = f(x_i, y_i, w)|_{i \sim U(1,N)}$

SGD with momentum

Idea: average gradient over consecutive steps

aka "add inertia"

$$w_0 := 0; \nu_0 := 0$$

$$\nu_{i+1} := \alpha \cdot \frac{\partial L}{\partial w} + \mu \cdot \nu_i$$

$$w_{i+1} := w_i - \nu_{i+1}$$

SGD with momentum

Idea: average gradient over consecutive steps

aka "add inertia"

$$w_0 := 0; \nu_0 := 0$$

$$\nu_{i+1} := \alpha \cdot \frac{\partial L}{\partial w} + \mu \cdot \nu_i$$

// reuse gradient from previous steps

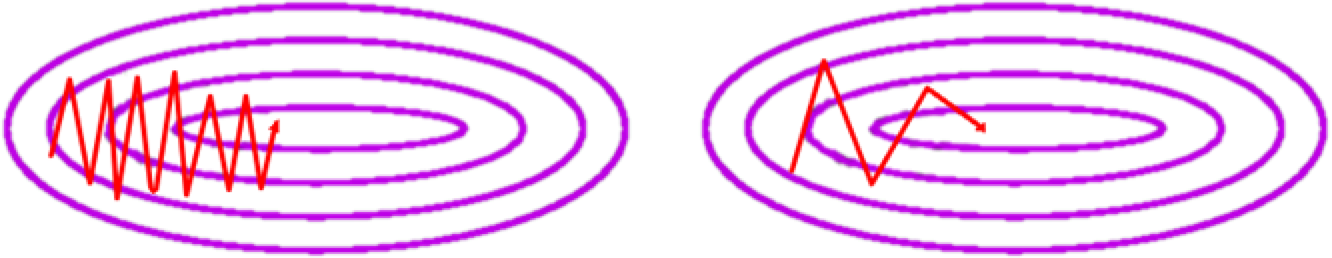
$$w_{i+1} := w_i - \nu_{i+1}$$

SGD with momentum

Idea: average gradient over consecutive steps

(see this demo)

$$\nu_{i+1} := \alpha \cdot \frac{\partial L}{\partial w} + \mu \cdot \nu_i \qquad w_{i+1} := w_i - \nu_{i+1}$$



RMSProp

Idea: adapt learning rate to the slope of objective function

large gradient = go slower, small gradient = go faster

$$w_0 := 0; ms_0 := 0$$

$$ms_{i+1} := \gamma \cdot ms_i + (1-\gamma) \cdot ||\frac{\partial L}{\partial w}||$$

$$w_{i+1} := w_i - \frac{\alpha}{\sqrt{ms_{i+1} + \epsilon}} \frac{\partial L}{\partial w}$$

RMSProp

Idea: adapt learning rate to the slope of objective function

large gradient = go slower, small gradient = go faster

$$w_0 := 0; ms_0 := 0$$

moving average norm^2

$$ms_{i+1} := \gamma \cdot ms_i + (1-\gamma) \cdot ||\frac{\partial L}{\partial w}||$$

$$w_{i+1} := w_i - \frac{\alpha}{\sqrt{ms_{i+1} + \epsilon}} \frac{\partial L}{\partial w}$$

Adam

Idea: combine momentum and rmsprop in one method

the "default" optimizer, see the paper for details

$$w_0 := 0; ms_0 := 0$$

moving average norm^2

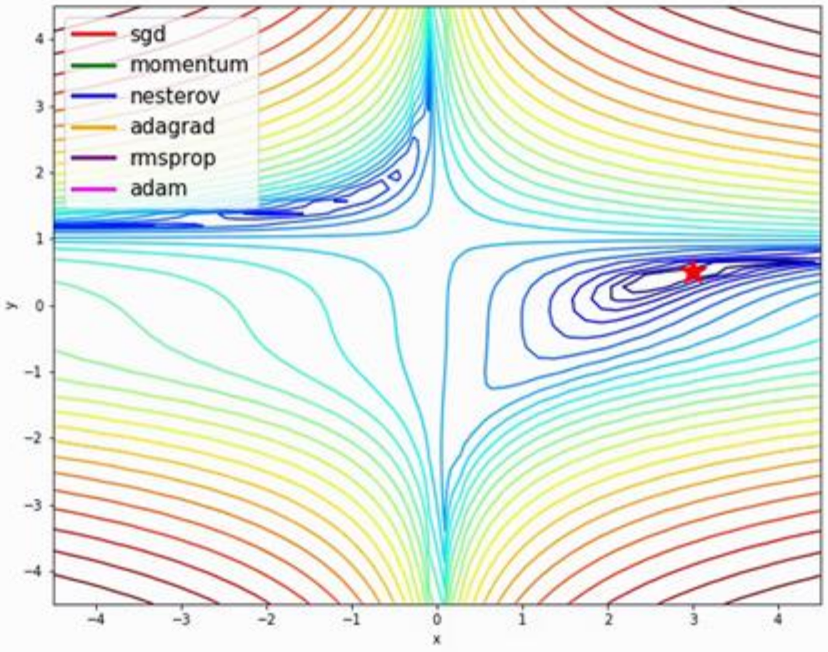
$$ms_{i+1} := \gamma \cdot ms_i + (1-\gamma) \cdot ||\frac{\partial L}{\partial w}||$$

$$w_{i+1} := w_i - \frac{\alpha}{\sqrt{ms_{i+1} + \epsilon}} \frac{\partial L}{\partial w}$$

TL;DR stochastic optimization

Tips & tricks

- Adam works fine out of the box
- One can usually beat adam with tuned sgd+momentum
  Tuning may take long
- Everyone has his favorite optimizer :)



Stuff we won’t cover

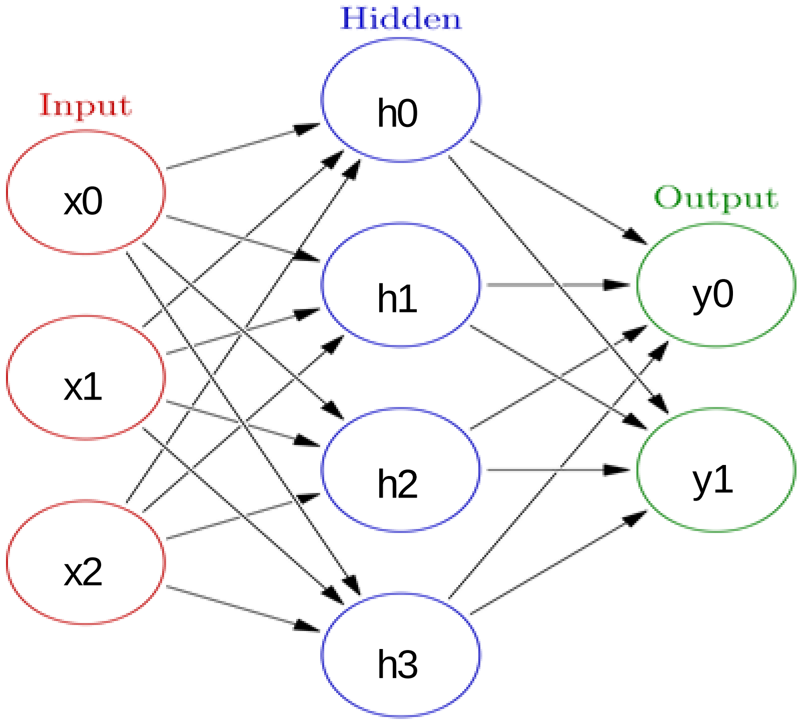
First order

- Adam flavors: NAdam, Adamax, QHAdam
- Adagrad, Radagrad, Adadelta - alternative adaptive lr

Approx second order

- BFGS, L-BFGS
- K-FAC

Initialization, symmetry problem

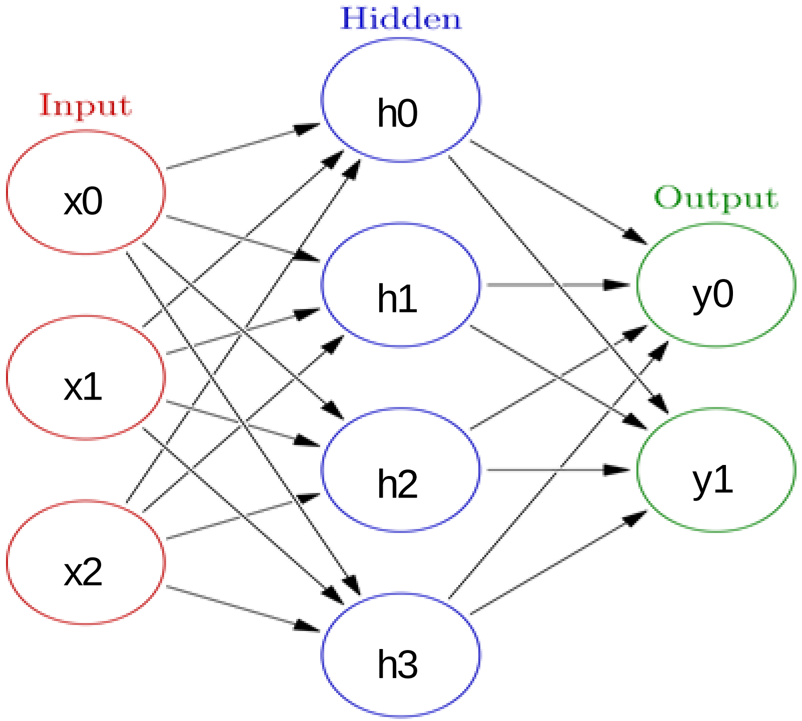


Initialize with zeros

$$W := 0$$

What will the first step look like?

Initialization, symmetry problem



- Break the symmetry!
- Initialize with random numbers!

$$W := N(0,0.01)?$$
$$W := U(-0.1,0.1)?$$

read more

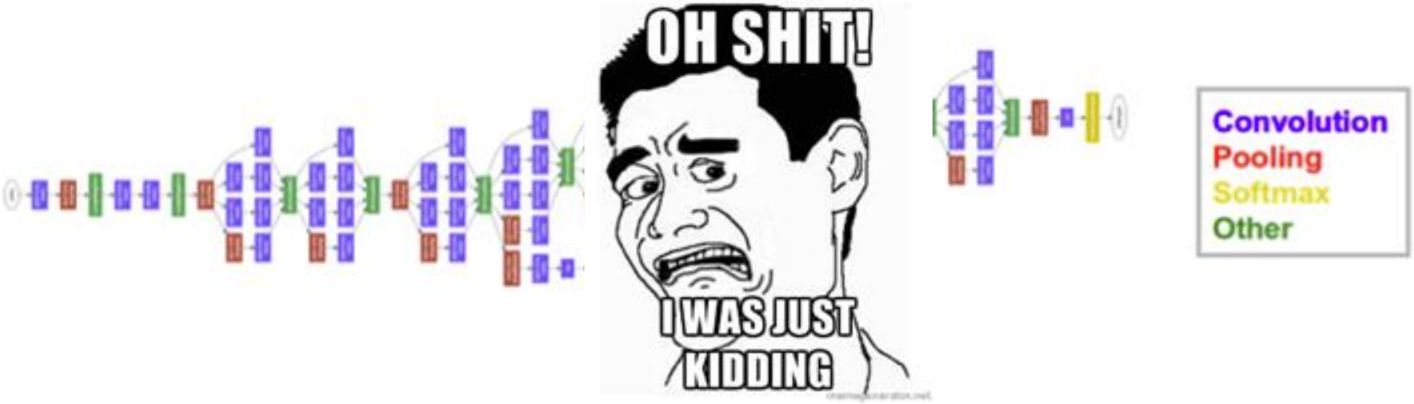
Nuff said

Let's go implement that!

And now let's differentiate

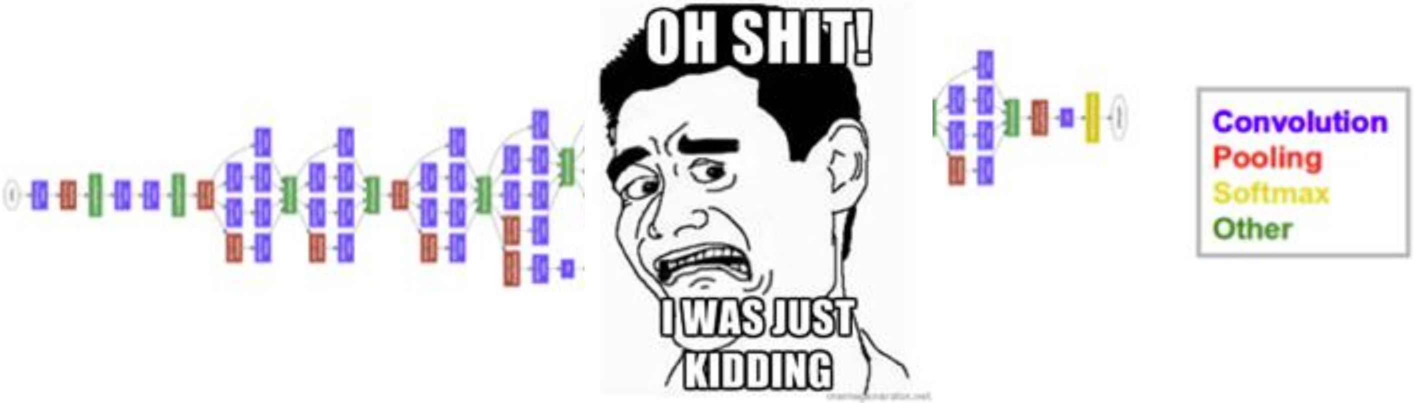
- near state-of-the-art in image classification
- 5+ types of layers
- parallel branches with independent losses
- few hundred megabytes of weights

And now let's differentiate



- near state-of-the-art in image classification
- 5+ types of layers
- parallel branches with independent losses
- few hundred megabytes of weights

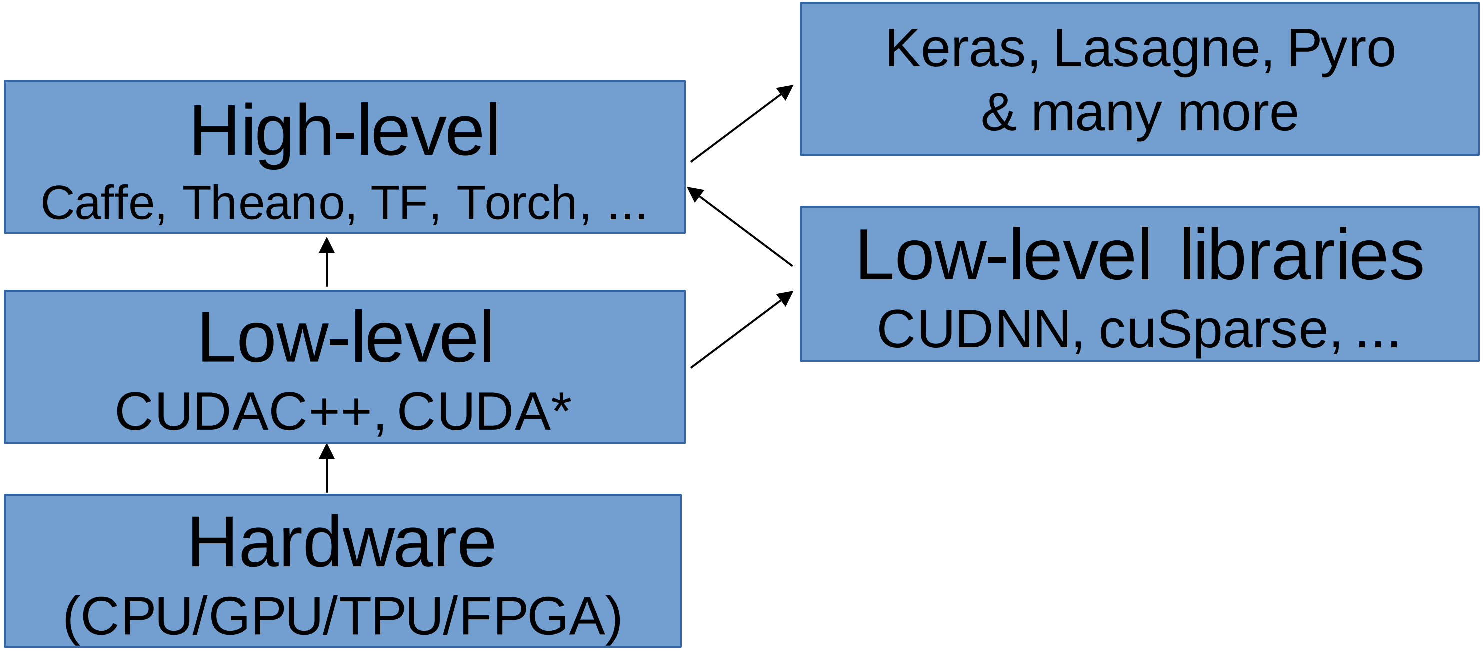
Deep learning frameworks



- near state-of-the-art in image classification
- 5+ types of layers
- parallel branches with independent losses
- few hundred megabytes of weights

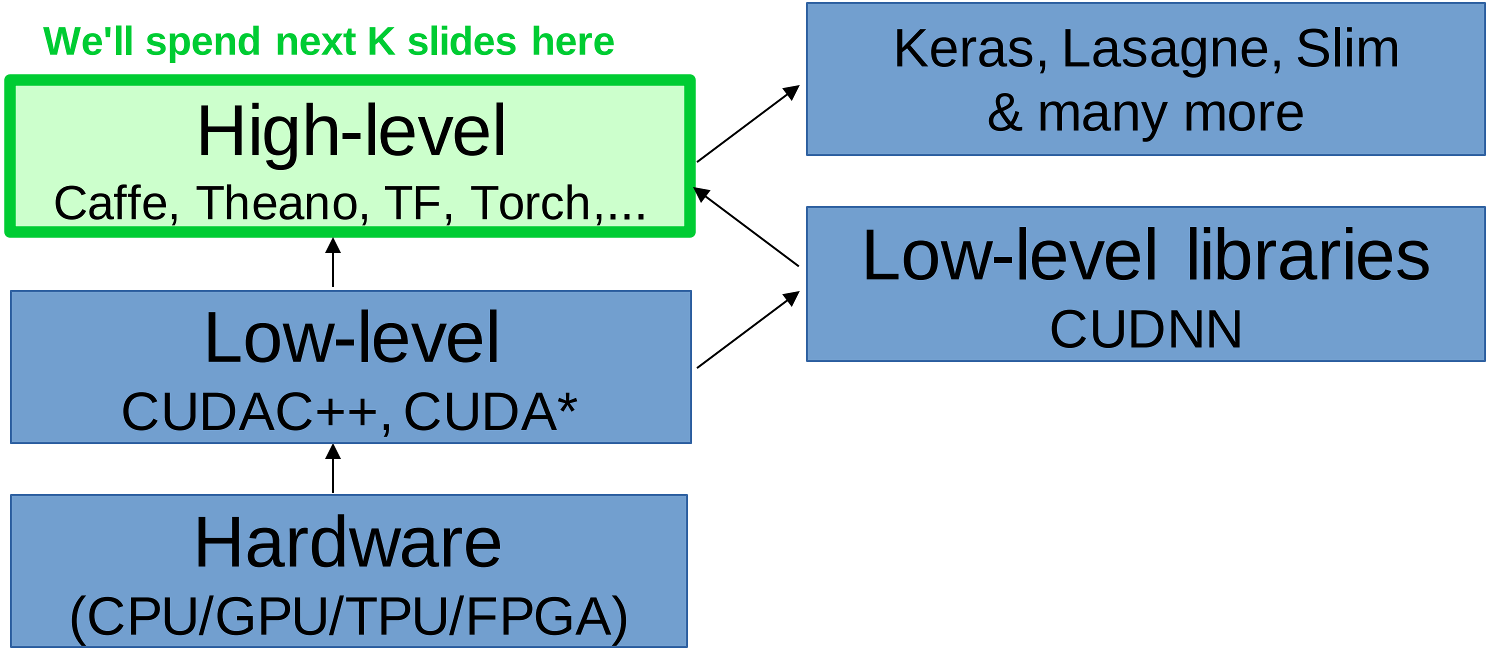
Deep learning frameworks

- Core idea: helps you define and train neural nets



Deep learning frameworks

- Core idea: helps you define and train neural nets

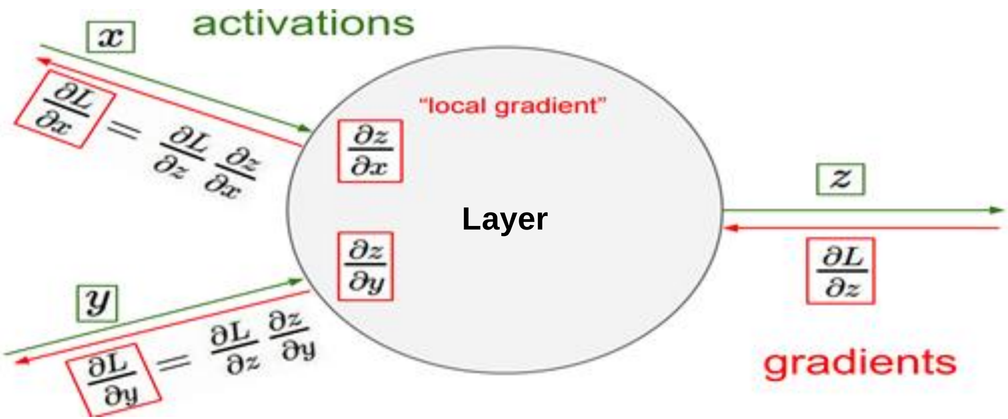


Deep learning frameworks

- Layer-based frameworks:
  - Same idea as in our hand-made neural net

Deep learning frameworks

- Layer-based frameworks:
  - Same idea as in our hand-made neural net
    - this one - http://bit.ly/2w9kAHm



Deep learning frameworks

Caffe

```
name: "LeNet"
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {lr_mult: 1}
  param {lr_mult: 2}
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }}}
....
```

[130 lines]()

You define model in config file by stacking layers (left)

Then run train script:

```
caffe train -solver examples/mnist/lenet_solver.prototxt
```

Deep learning frameworks

Caffe

```
name: "LeNet"
layer {
 name: "conv1"
 type: "Convolution"
 bottom: "data"
 top: "conv1"
 param {lr_mult: 1}
 param {lr_mult: 2}
 convolution_param {
  num_output: 20
  kernel_size: 5
  stride: 1
  weight_filler {
   type: "xavier"
  }}}
....
```

130 lines

- \+ Easy to deploy (C++)
- \+ A lot of pre-trained models (model zoo)
- \- Model as protobuf
- \- Hard to build new layers
- \- Hard to debug

Still used for computer vision

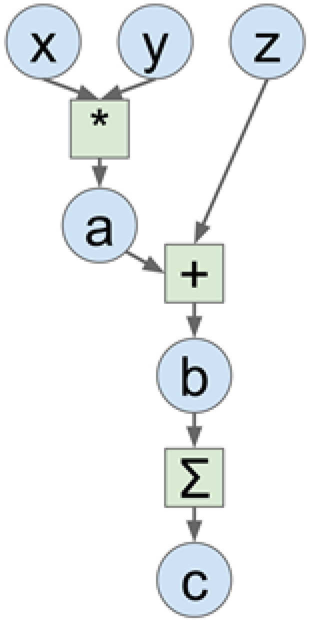
Computation graphs

What does your CPU do
when you write this?

```
a = x * y
b = a + z
c = np.sum(b)
```

This many further slides taken from Ars Ashukha @deepbayes2017

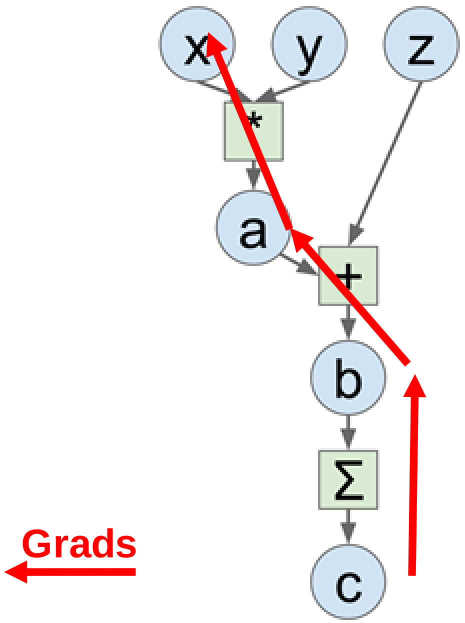
Computation graphs




```
a = x * y
b = a + z
c = np.sum(b)
```

Idea: let's define this graph explicitly!

Computation graphs



- \+ Automatic gradients!
- \+ Easy to build new layers
- \+ We can optimize the Graph
- \- Graph is static during training
- \- Need time to compile/optimize
- \- Hard to debug

This many further slides taken from Ars Ashukha @deepbayes2017

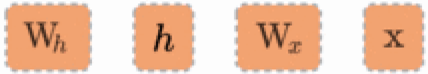
Dynamic graphs

- Chainer, DyNet, Pytorch

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```



Dynamic graphs

- Chainer, DyNet, Pytorch

PYTORCH

Chainer

\+ Can change graph on the fly
\+ Can get value of any tensor at any time (easy debugging)
\- Hard to optimize graphs (especially large graphs)
\- Still early development

Researchers love these!

Dynamic graphs

Researchers love them!