



FORNAX

Introduction to Deep Learning, part 1

Krzysztof Kolasiński, Rafał Cycoń - 30 March - 2017

WWW.FORNAX.CO

Presentation outline



1. A short motivation for DL
2. Mathematical model of neuron
3. DL - extreme crash course

This talk:

- more technical, less practical
- we want to stress the importance of basic building blocks
- dense material, may be hard to follow
- non overlapping concepts
- assignment: try to understand covered concepts
- the feedback form



Why do we need DL?

Why do we need DL?

1. DL is fun!



Deep dream



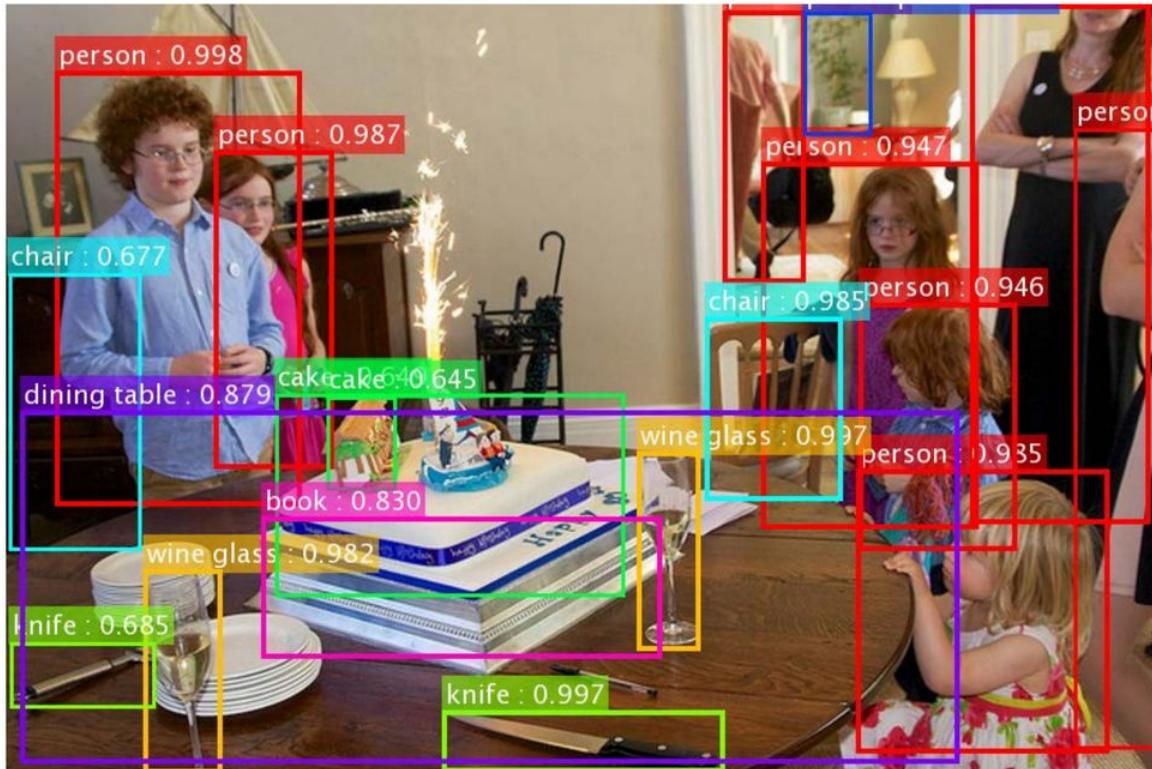
Neural style transfer

Why do we need DL?

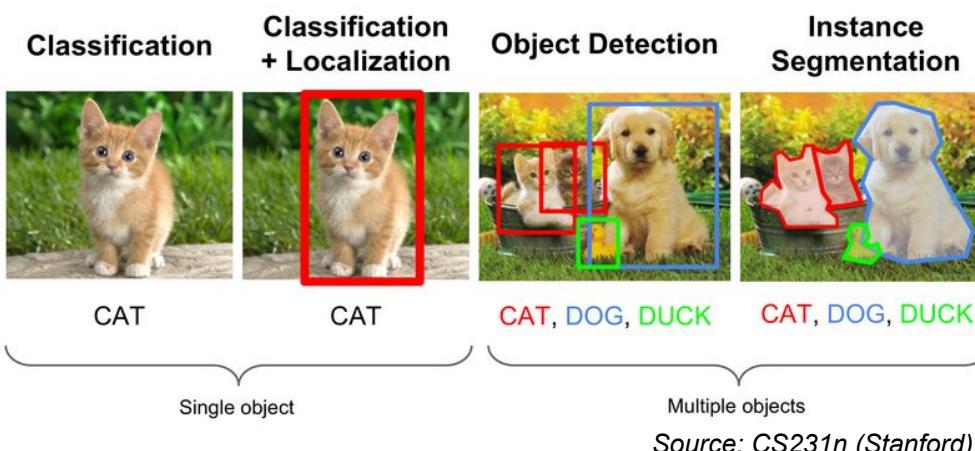
2. DL is important



image segmentation

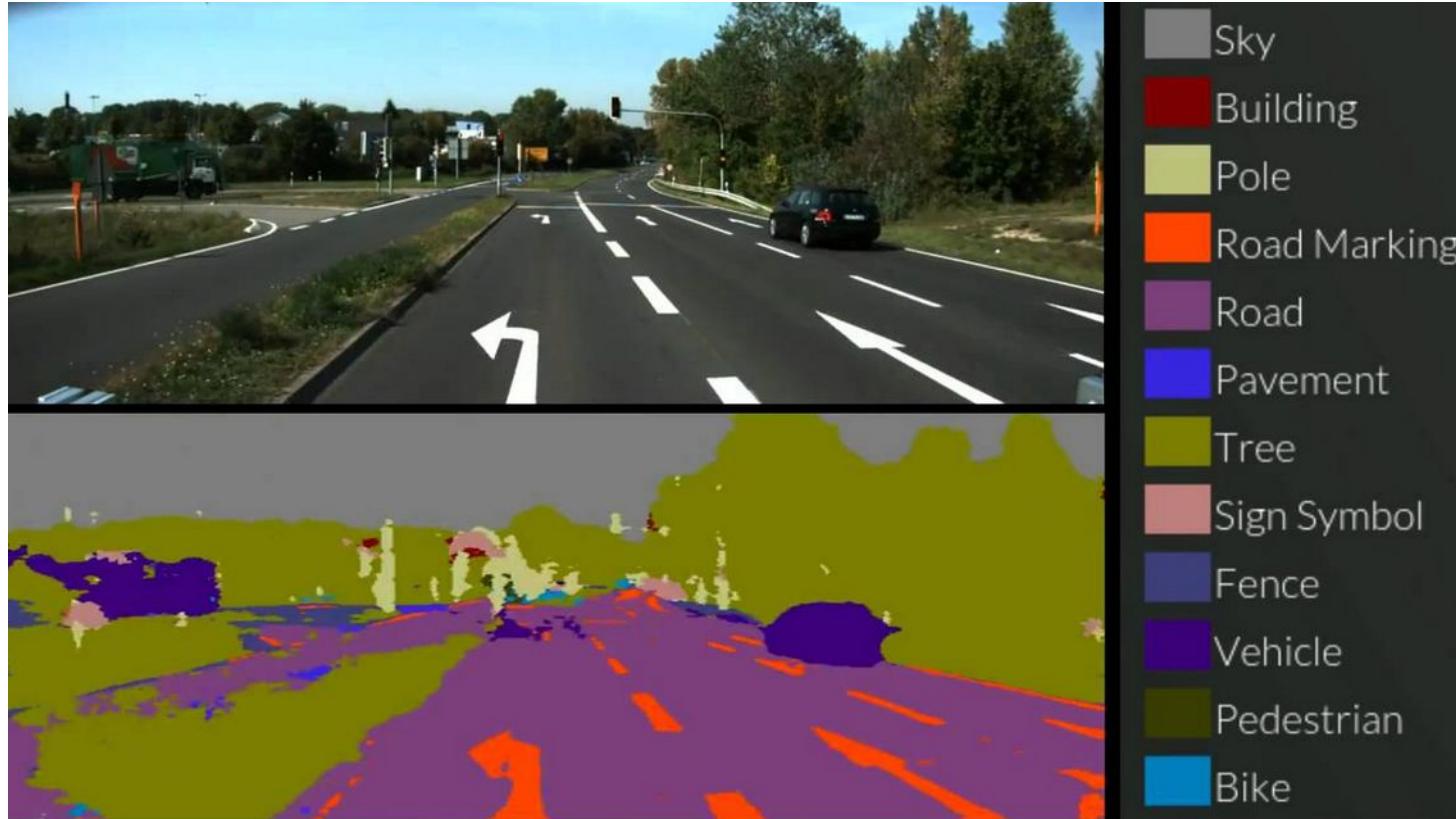


object detection



Why do we need DL?

2. DL is important



Application to self driving cars

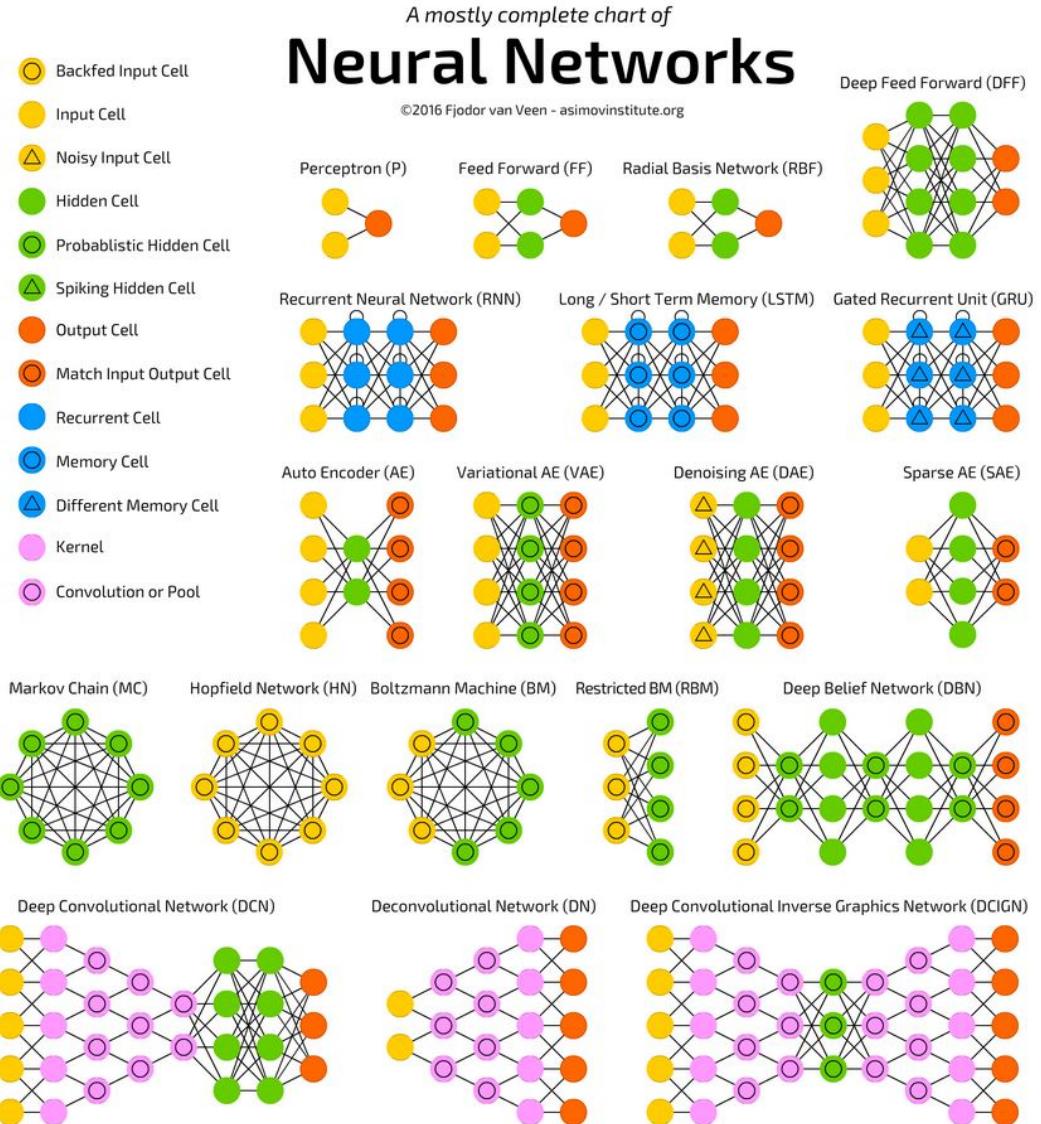
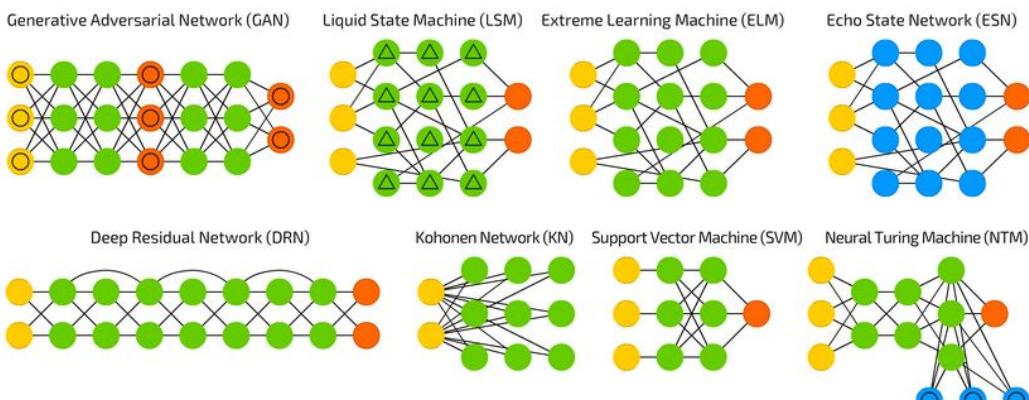


And many more...

The NN zoo

Understanding basic building blocks helps to create new and understand existing architectures

That NN family is growing up every day



<http://www.asimovinstitute.org/neural-network-zoo/>

Prologue: The basics

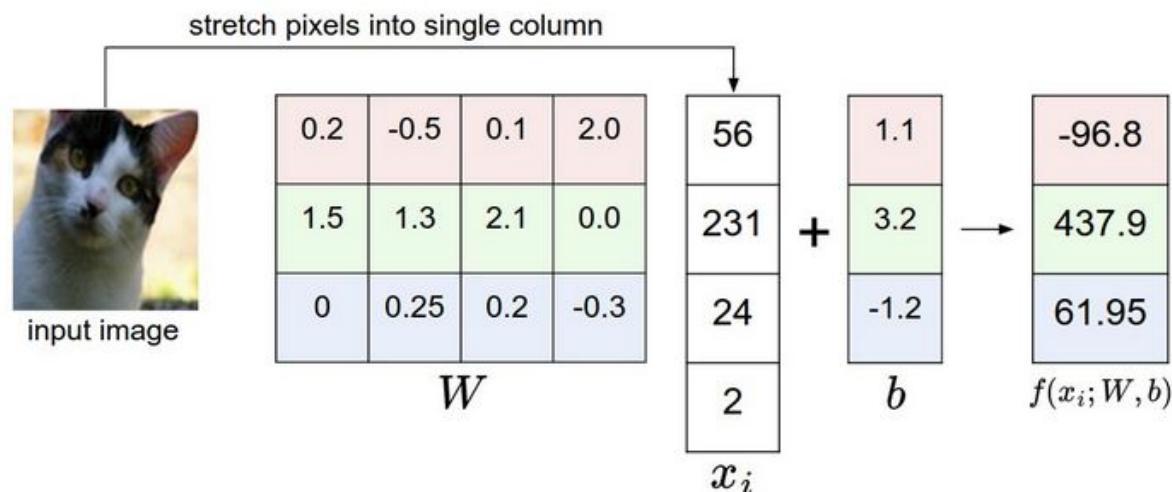
The simple Neuron unit

Feed forward Neural Network



We define linear unit as

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

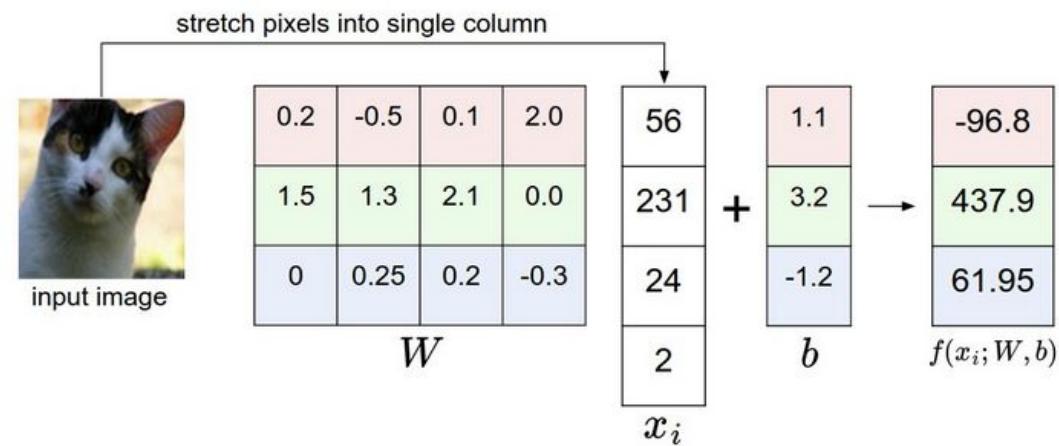


source: Stanford CS231 course

The simple Neuron unit

Linear classifier

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$



source: Stanford CS231 course

Case study: **MNIST** dataset and linear classifier

Steps:

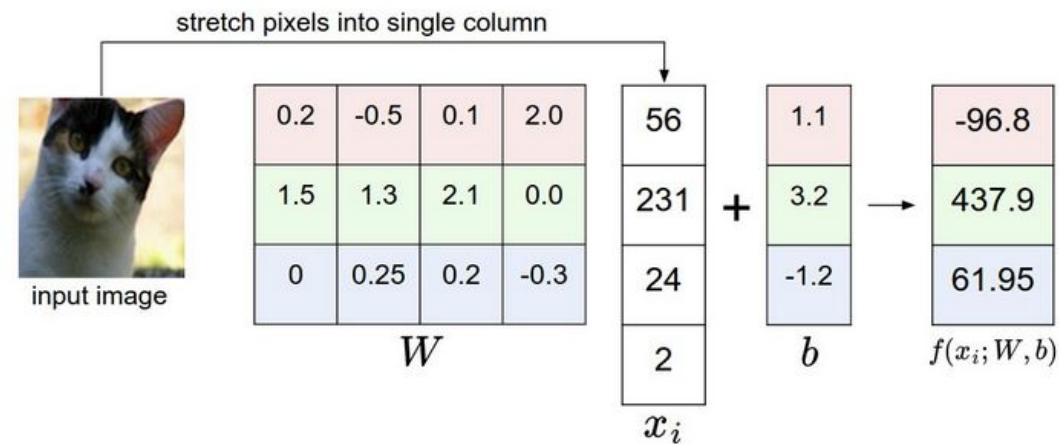
1. Dataset consists of N images (28×28 pixels/gray scale) of handwritten digits between 0-9



The simple Neuron unit

Linear classifier

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$



source: Stanford CS231 course

Case study: **MNIST** dataset and linear classifier

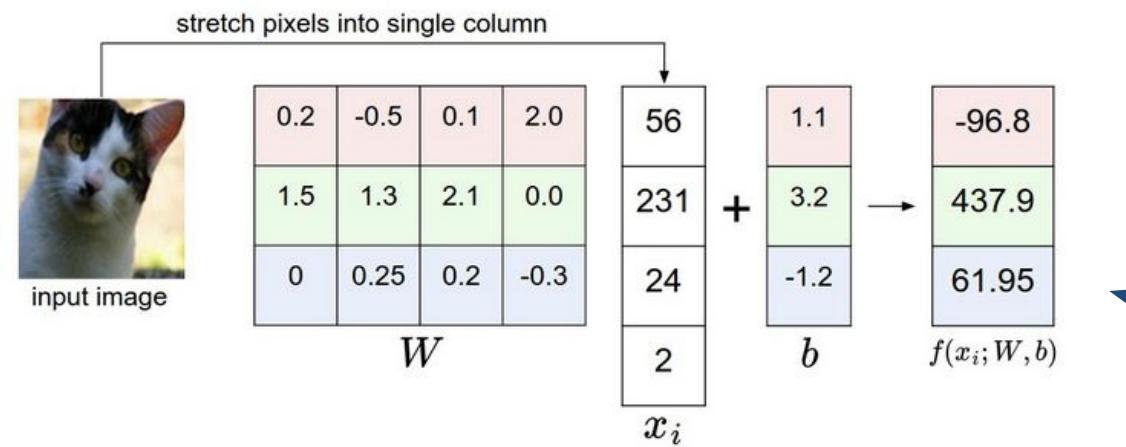
Steps:

1. Dataset consists of N images (28×28 pixels/gray scale) of handwritten digits between 0-9
2. The images are labeled (0-9)

The simple Neuron unit

Linear classifier

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$



source: Stanford CS231 course

Case study: **MNIST** dataset and linear classifier

Steps:

1. Dataset consists of N images (28x28 pixels/gray scale) of handwritten digits between 0-9
2. The images are labeled (0-9)
3. Lets solve the simplest possible **supervised** task:

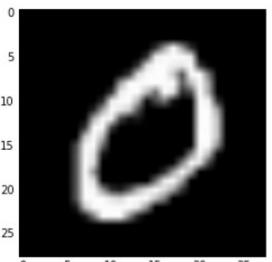
$$\hat{\mathbf{y}}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

model prediction

- We can compare this prediction with given label

$$\mathbf{y}_i = [0, \dots, 1_i, \dots, 0]$$

For example if the input image is **0** the label becomes

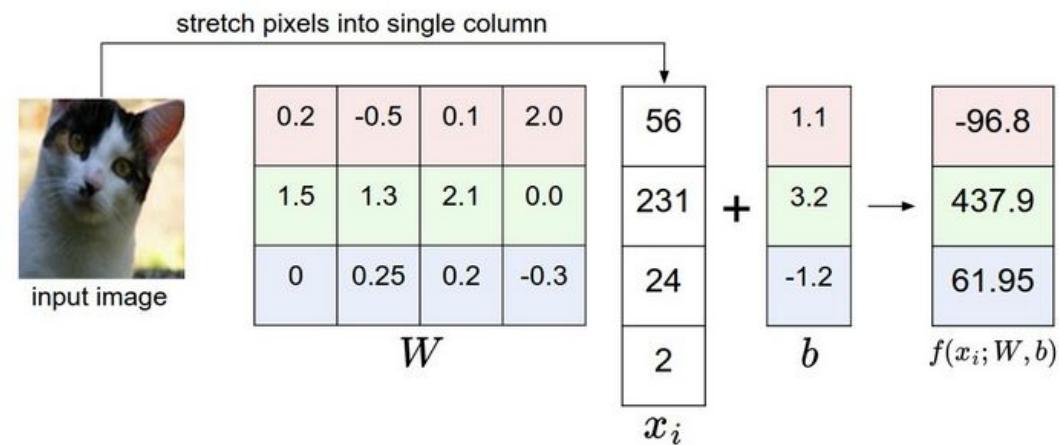


$$\mathbf{y}_i = [1, 0, \dots]$$

The simple Neuron unit

Linear classifier

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$



source: Stanford CS231 course

Case study: **MNIST** dataset and linear classifier

Steps:

1. Dataset consists of N images (28×28 pixels/gray scale) of handwritten digits between 0-9
2. The images are labeled (0-9)
3. Lets solve the simplest possible **supervised** task:

$$\hat{\mathbf{y}}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

- For each training example i we can define the loss function

$$\mathcal{L}_i \equiv l(\hat{\mathbf{y}}_i, \mathbf{y}_i)$$

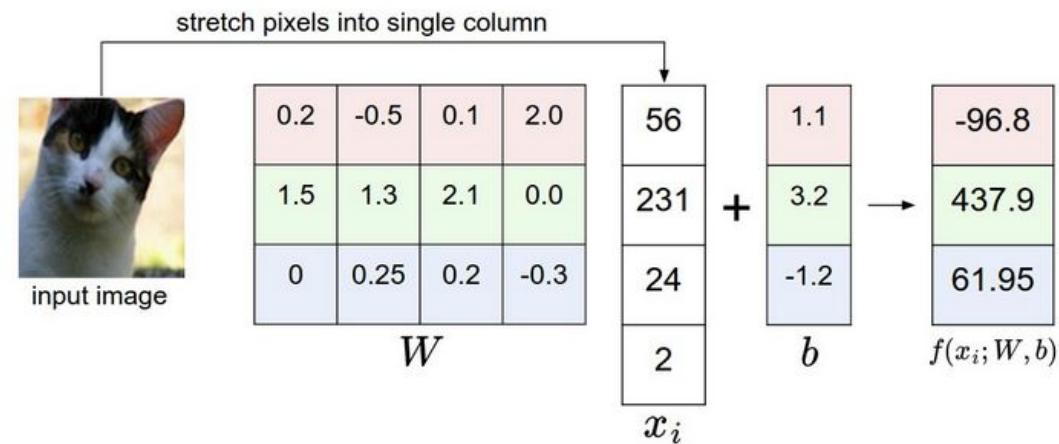


In general lower loss == better prediction

The simple Neuron unit

Linear classifier

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$



source: Stanford CS231 course

Case study: **MNIST** dataset and linear classifier

Steps:

1. Dataset consists of N images (28×28 pixels/gray scale) of handwritten digits between 0-9
2. The images are labeled (0-9)
3. Lets solve the simplest possible supervised task:

$$\hat{\mathbf{y}}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

- For each training example i we can define the loss function

$$\mathcal{L}_i \equiv l(\hat{\mathbf{y}}_i, \mathbf{y}_i)$$

- For simplicity we choose MSE loss

$$l(\hat{\mathbf{y}}_i, \mathbf{y}_i) = MSE(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \frac{1}{N_c} \sum_{j=1}^{N_c} (\hat{y}_{ij} - y_{ij})^2$$

The simple Neuron unit

Linear classifier

$$\hat{\mathbf{y}}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

prediction loss per one image

$$\mathcal{L}_i \equiv l(\hat{\mathbf{y}}_i, \mathbf{y}_i) \quad l(\hat{\mathbf{y}}_i, \mathbf{y}_i) = MSE(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \frac{1}{N_c} \sum_{j=1}^{N_c} (\hat{y}_{ij} - y_{ij})^2$$

The **MNIST** dataset (keras) consists of 60000 training images. Hence the total loss is given:

$$\mathcal{L}_{\text{total}} \equiv \frac{1}{N_s} \sum_{i=1}^{N_s} MSE(\hat{\mathbf{y}}_i, \mathbf{y}_i)$$

Target: find **W** and **b** such that **loss** will be minimal

Solution: use gradient descent method to update model parameters:

$$\theta^{k+1} := \theta^k - \lambda \frac{\partial \mathcal{L}_{\text{total}}}{\partial \theta^k}$$

one has to compute gradient of loss w.r.t. model parameters

The simple Neuron unit

Linear classifier

$$\hat{\mathbf{y}}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

prediction loss per one image

$$\mathcal{L}_i \equiv l(\hat{\mathbf{y}}_i, \mathbf{y}_i) \quad l(\hat{\mathbf{y}}_i, \mathbf{y}_i) = MSE(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \frac{1}{N_c} \sum_{j=1}^{N_c} (\hat{y}_{ij} - y_{ij})^2$$

The **MNIST** dataset (keras) consists of 60000 training images. Hence the total loss is given:

It is more practical to use mini-batch approximation of gradients instead taking whole sum:

$$\mathcal{L}_{\text{total}} \equiv \frac{1}{N_s} \sum_{i=1}^{N_s} MSE(\hat{\mathbf{y}}_i, \mathbf{y}_i) \quad \rightarrow \quad \mathcal{L}_{\text{mini-batch}} \equiv \frac{1}{N_{\text{MB}}} \sum_{i \in \text{MB}}^{N_{\text{MB}}} MSE(\hat{\mathbf{y}}_i, \mathbf{y}_i)$$

mini-batch size

This lead to mini-batch **stochastic gradient descent** algorithm:

$$\theta^{k+1} := \theta^k - \lambda \left. \frac{\partial \mathcal{L}}{\partial \theta^k} \right|_{\text{mini-batch}}$$

The simple Neuron unit



Linear classifier - steps:

1. Define the model: $\hat{\mathbf{y}}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$
2. Define the objective: $\mathcal{L}_{\text{mini-batch}} \equiv \frac{1}{N_{\text{MB}}} \sum_{i \in \text{MB}} MSE(\hat{\mathbf{y}}_i, \mathbf{y}_i)$
3. Define parameter optimizer: $\theta^{k+1} := \theta^k - \lambda \frac{\partial \mathcal{L}}{\partial \theta^k} \Big|_{\text{mini-batch}}$

Example in [Keras](#)

The screenshot shows the Keras Documentation homepage. The header features a red bar with the Keras logo and a search bar. Below the header, the page title is "Keras: Deep Learning library for Theano and TensorFlow". A sub-header says "You have just found Keras.". The main content area describes Keras as a high-level neural networks API written in Python, capable of running on top of either TensorFlow or Theano. It emphasizes fast experimentation and prototyping. Below this, there's a section titled "Use Keras if you need a deep learning library that:" followed by a bulleted list of features.

Keras Documentation

Docs » Home

Edit on GitHub

Keras: Deep Learning library for Theano and TensorFlow

You have just found Keras.

Keras is a high-level neural networks API, written in Python and capable of running on top of either [TensorFlow](#) or [Theano](#). It was developed with a focus on enabling fast experimentation. *Being able to go from idea to result with the least possible delay is key to doing good research.*

Use Keras if you need a deep learning library that:

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Runs seamlessly on CPU and GPU.

WWW.FORNAX.CO

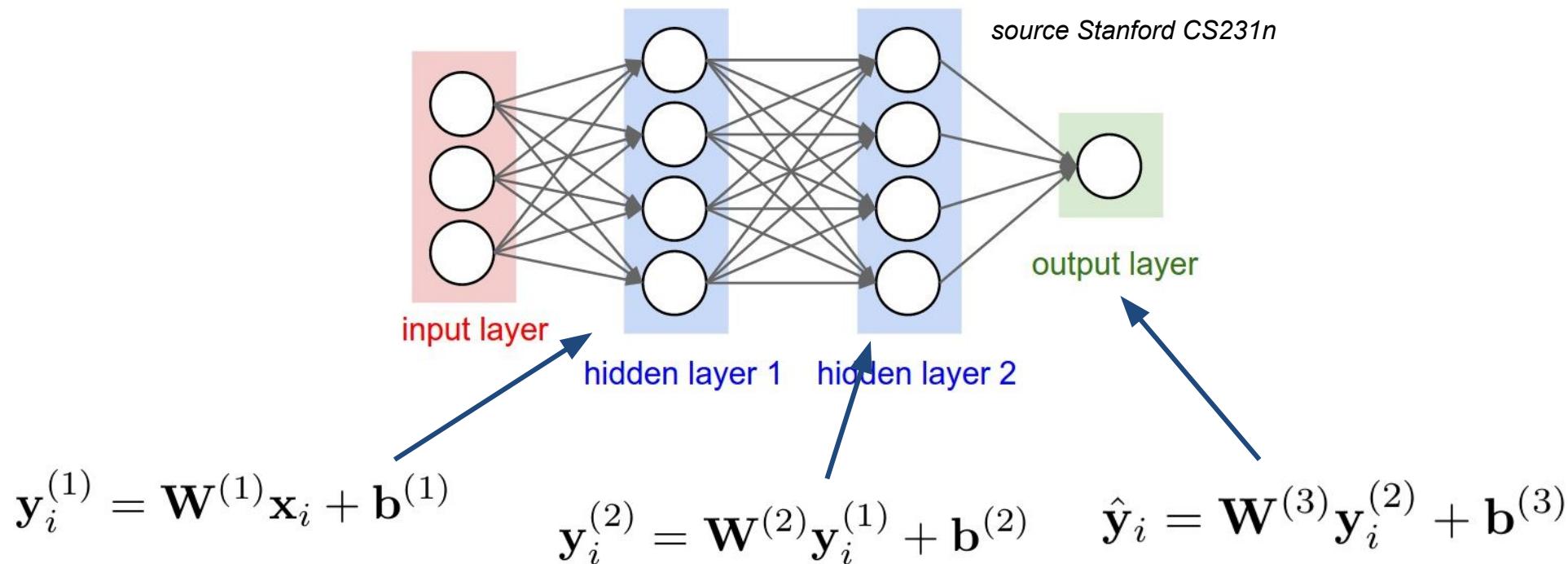


The simple Neuron unit

Linear classifier gives 85% accuracy

$$\hat{\mathbf{y}}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

Can we stack multiple **linear classifiers** to build something more powerful?



The simple Neuron unit

Linear classifier gives 85% accuracy

$$\hat{\mathbf{y}}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

Can we stack multiple **linear classifiers** to build something more powerful?

$$\mathbf{y}_i^{(1)} = \mathbf{W}^{(1)}\mathbf{x}_i + \mathbf{b}^{(1)}$$

source Stanford CS231n

$$\mathbf{y}_i^{(2)} = \mathbf{W}^{(2)}\mathbf{y}_i^{(1)} + \mathbf{b}^{(2)}$$

$$\hat{\mathbf{y}}_i = \mathbf{W}^{(3)}\mathbf{y}_i^{(2)} + \mathbf{b}^{(3)} \longrightarrow \hat{\mathbf{y}}_i = \mathbf{W}^{(3)} \left(\mathbf{W}^{(2)} \left(\mathbf{W}^{(1)}\mathbf{x}_i + \mathbf{b}^{(1)} \right) + \mathbf{b}^{(2)} \right) + \mathbf{b}^{(3)}$$

Lets

$$\hat{\mathbf{y}}_i = \boxed{\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x}_i} + \boxed{\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{b}^{(1)} + \mathbf{W}^{(3)}\mathbf{b}^{(2)} + \mathbf{b}^{(3)}}$$

$$\hat{\mathbf{y}}_i = \mathbf{W}'\mathbf{x}_i + \mathbf{b}'$$

The model with multiple layers cannot be better than single layer case!

The simple Neuron unit



The definition of neuron

To obtain neuron (fully connected layer) we replace

$$\hat{\mathbf{y}}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

with

$$\hat{\mathbf{y}}_i = f(\mathbf{W}\mathbf{x}_i + \mathbf{b})$$

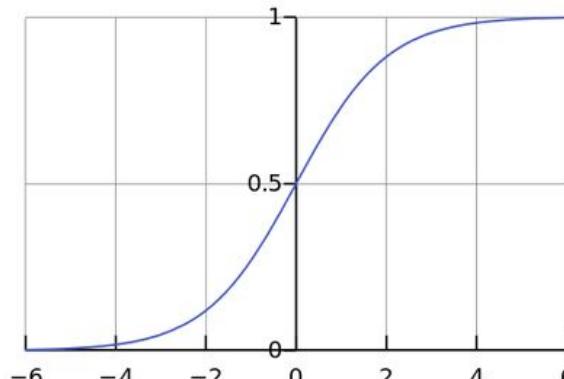
where $f(\dots)$ is so-called activation function

f is a non linear function hence we cannot factorize multiple layers as before

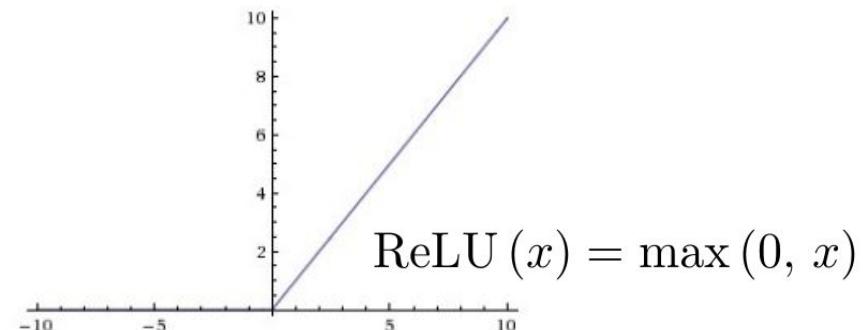
Some examples:

- sigmoid

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



- ReLU (Rectified Linear Unit)



The simple Neuron classifier revisited

We are going to solve the same problem as before lets add nonlinearity

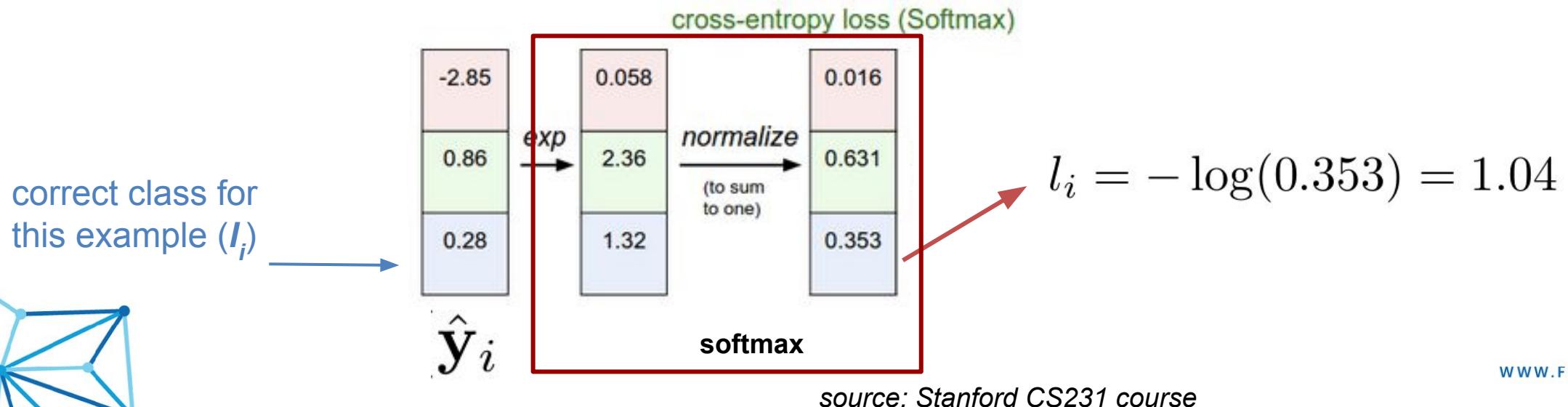
$$\hat{\mathbf{y}}_i = f(\mathbf{W}\mathbf{x}_i + \mathbf{b})$$

The standard approach for last layer in classification problem is a softmax activation function for 10 classes:

$$f(\hat{\mathbf{y}}_i) = \frac{e^{\hat{y}_{il_i}}}{\sum_{k=1}^{10} e^{\hat{y}_{ik}}}$$

Note: In keras softmax activation returns vector of probabilities

Then the standard approach is to use cross-entropy loss function: $l_i = -\log(p(\hat{\mathbf{y}}_i))$

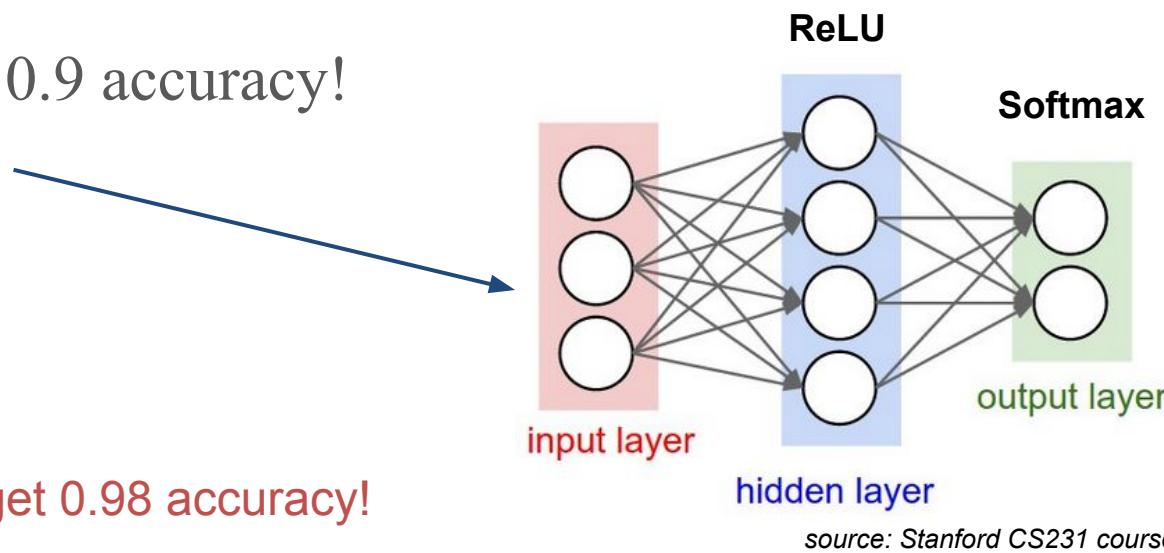


The simple Neuron classifier revisited

Examples in Keras:

- single layer with softmax: 0.9 accuracy!
- double layer with softmax

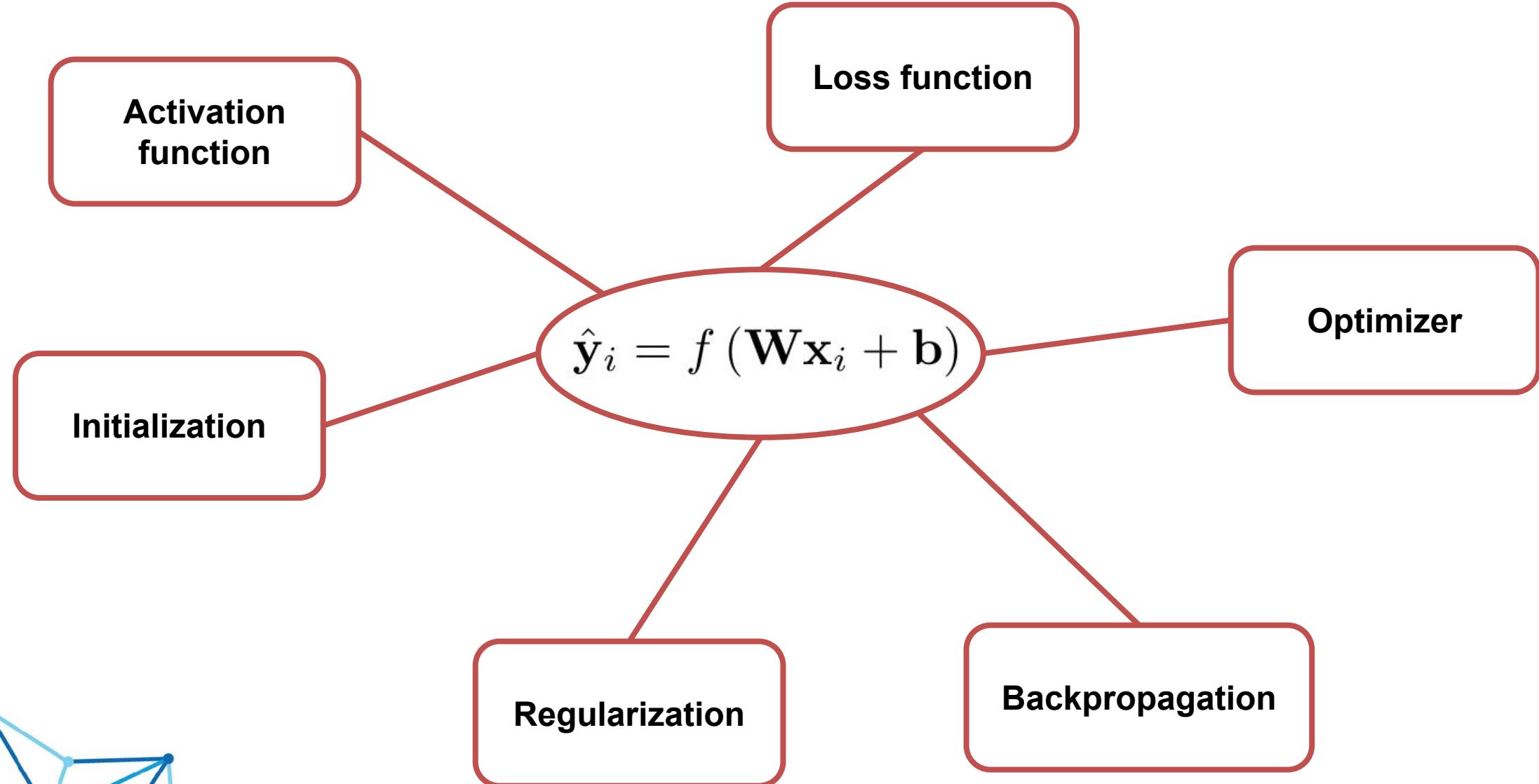
with double layer NN we get 0.98 accuracy!



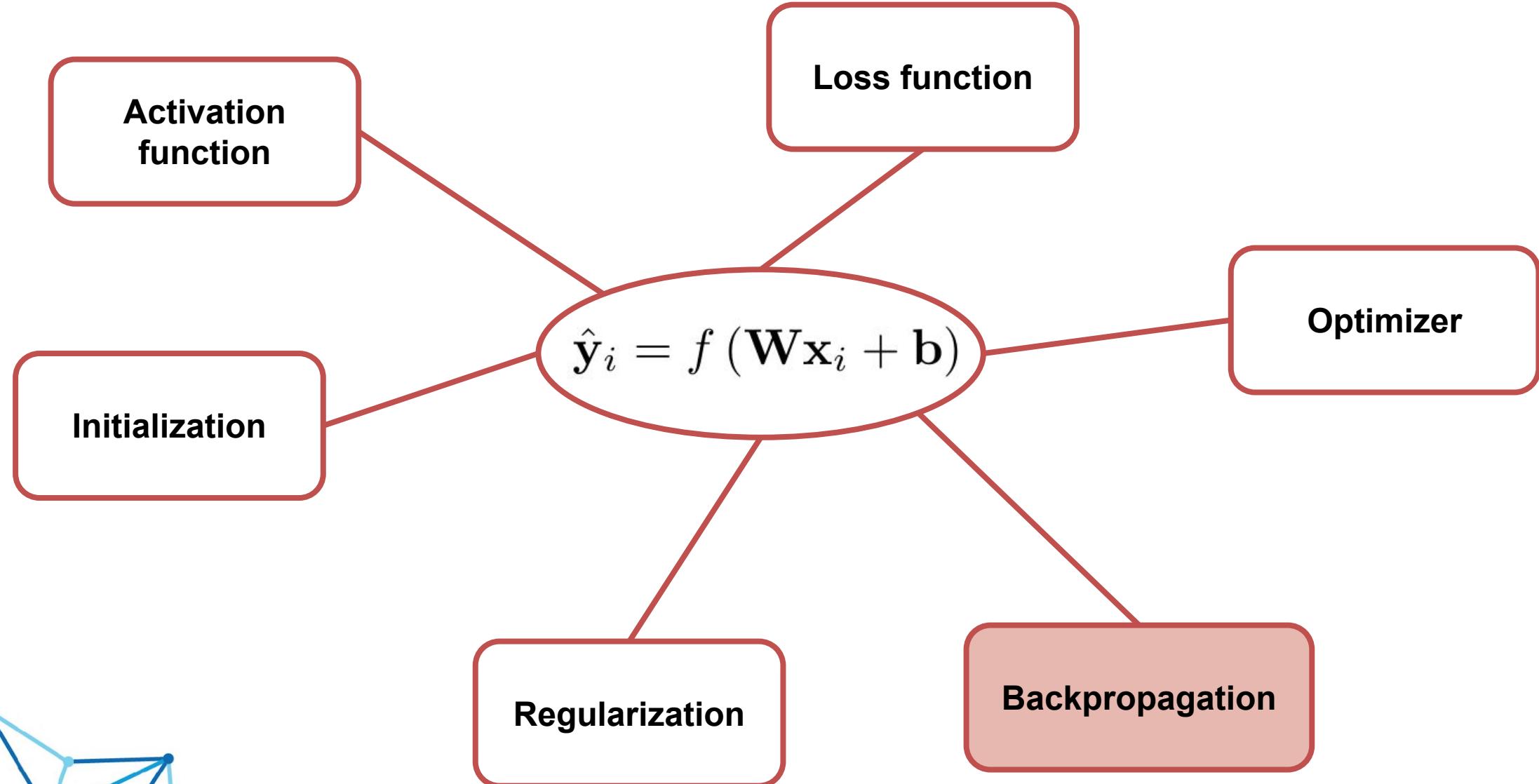
source: Stanford CS231 course

NN extreme crash course

Neural Networks



Neural Networks



Backpropagation



Backpropagation is a pillar of deep learning

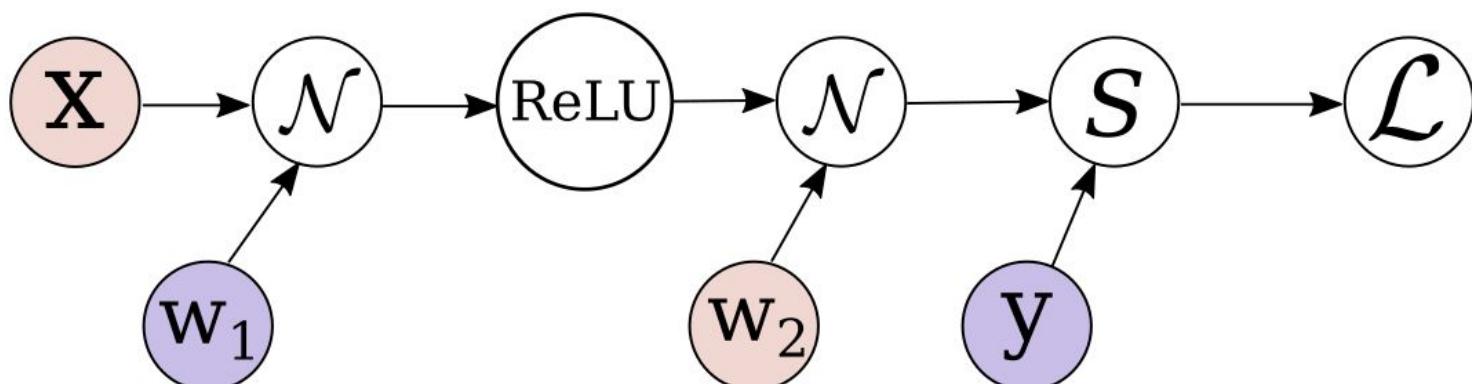
$$\theta^{k+1} := \theta^k - \lambda \frac{\partial \mathcal{L}}{\partial \theta^k} \Big|_{\text{mini-batch}}$$

$$\mathcal{L}_{\text{mini-batch}} \equiv \frac{1}{N_{\text{MB}}} \sum_{i \in \text{MB}}^{N_{\text{MB}}} MSE(\hat{\mathbf{y}}_i, \mathbf{y}_i)$$

Layer (type)	Output Shape	Param #
x (InputLayer)	(None, 784)	0
w1 (Dense)	(None, 100)	78500
w2 (Dense)	(None, 10)	1010

we need to compute gradient of loss with respect to model parameters

Lets come back to our two layer neural network



Backpropagation

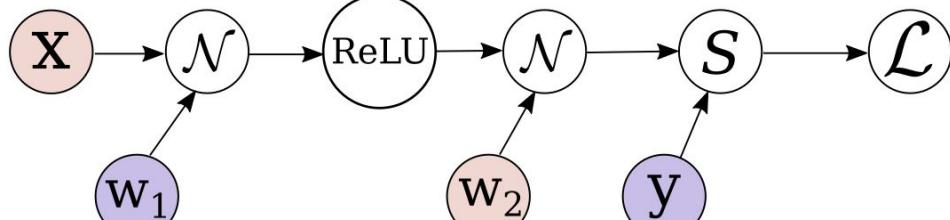
Backpropagation of mini-batch

$$\theta^{k+1} := \theta^k - \lambda \frac{\partial \mathcal{L}}{\partial \theta^k} \Big|_{\text{mini-batch}}$$

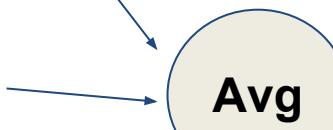
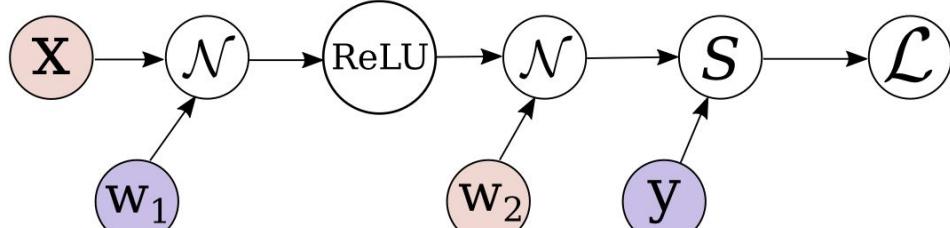
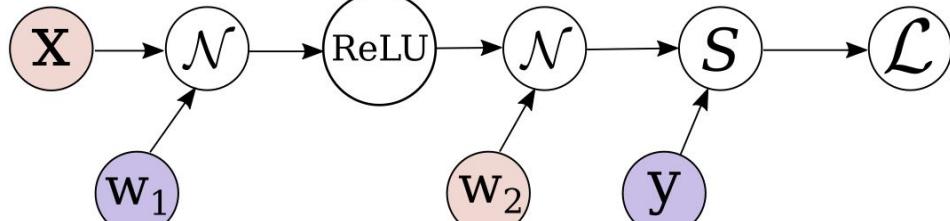
$$\mathcal{L}_{\text{mini-batch}} = \frac{1}{N_{MB}} \sum_{i \in MB} \mathcal{L}_i \quad \text{hence the gradient}$$

$$\frac{\partial \mathcal{L}}{\partial \theta} \Big|_{\text{mini-batch}} = \frac{1}{N_{MB}} \sum_{i \in MB} \frac{\partial \mathcal{L}_i}{\partial \theta}$$

1st example in MB



2nd example in MB

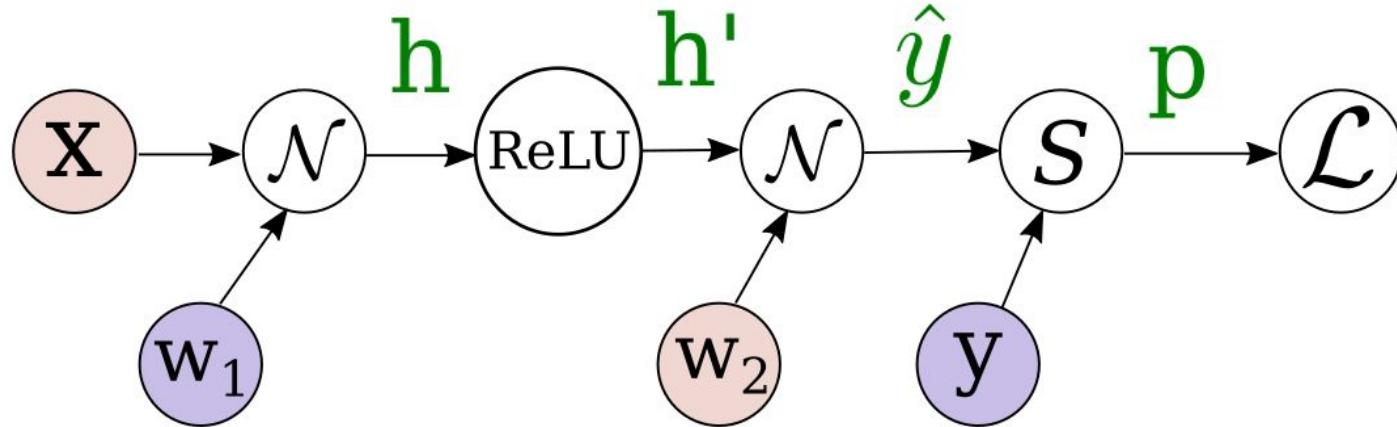


Avg - is a linear operation hence we can compute gradients for each example in mini-batch separately and then compute mean gradient on mini-batch

Backpropagation

$$\theta^{k+1} := \theta^k - \lambda \frac{\partial \mathcal{L}}{\partial \theta^k} \Big|_{\text{mini-batch}}$$

Forward pass



$$\mathcal{N}(x; w_1) = w_1 x$$

$$\text{ReLU}(x) = \max(0, x)$$

$$\mathcal{L}(x) = -\log(x)$$

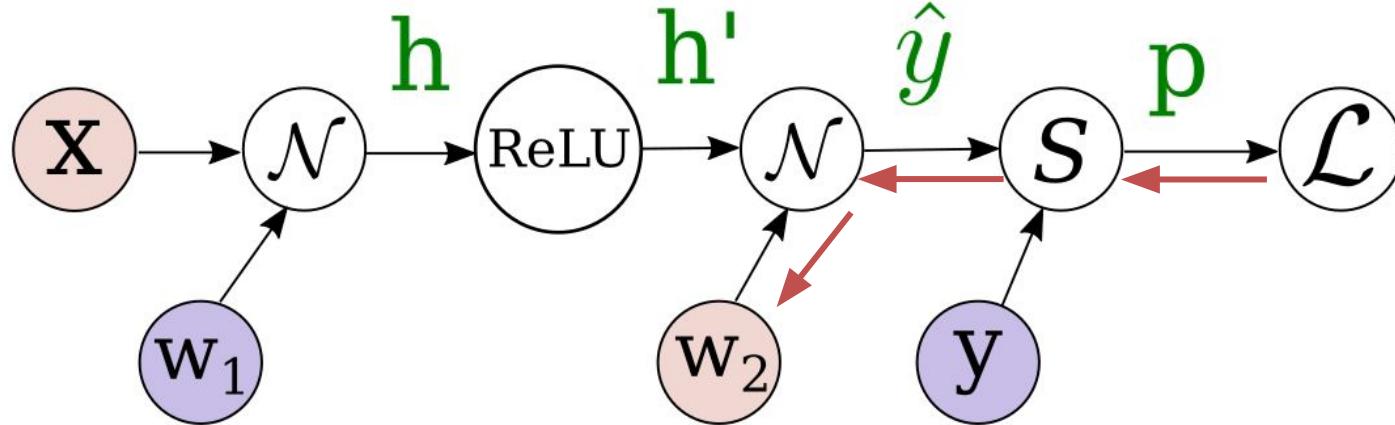
$$p(\hat{y}_i) = \frac{e^{\hat{y}_{il_i}}}{\sum_{k=1}^{10} e^{\hat{y}_{ik}}}$$

In forward pass we have to define output value for each type node for given input vector

Backpropagation

$$\theta^{k+1} := \theta^k - \lambda \frac{\partial \mathcal{L}}{\partial \theta^k} \Big|_{\text{mini-batch}}$$

In backward pass we use chain rule to find gradients



$$\mathcal{N}(x; w_1) = w_1 x$$

$$\text{ReLU}(x) = \max(0, x)$$

$$\mathcal{L}(x) = -\log(x)$$

$$p(\hat{y}_i) = \frac{e^{\hat{y}_{il_i}}}{\sum_{k=1}^{10} e^{\hat{y}_{ik}}}$$

Chain rule for w_2

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial p} \frac{\partial p}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2} = -\frac{1}{p} s \cdot h'^T$$

Each of the components can be computed easily e.g.:

$$\frac{\partial \mathcal{L}(p)}{\partial p} = -\frac{1}{p}$$

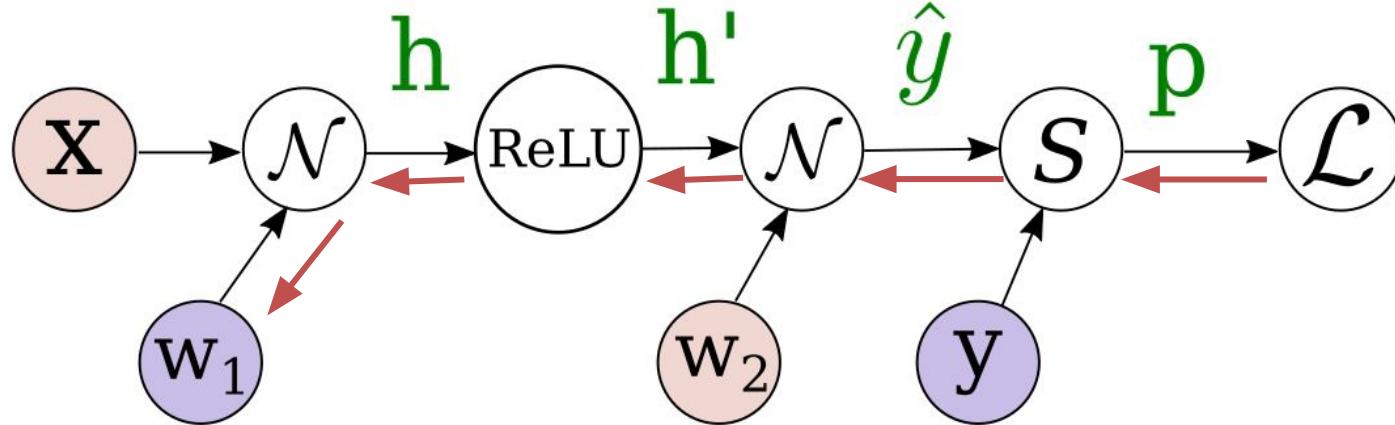
$$\frac{\partial p}{\partial \hat{y}} = s$$

$$\frac{\partial \hat{y}}{\partial w_2} = h'^T$$

Backpropagation

$$\theta^{k+1} := \theta^k - \lambda \frac{\partial \mathcal{L}}{\partial \theta^k} \Big|_{\text{mini-batch}}$$

In backward pass we use chain rule to find gradients



$$\mathcal{N}(x; w_1) = w_1 x$$

$$\text{ReLU}(x) = \max(0, x)$$

$$\mathcal{L}(x) = -\log(x)$$

$$p(\hat{y}_i) = \frac{e^{\hat{y}_{il_i}}}{\sum_{k=1}^{10} e^{\hat{y}_{ik}}}$$

Chain rule for w_2

$$\frac{\partial \mathcal{L}}{\partial w_2} = \boxed{\frac{\partial \mathcal{L}}{\partial p} \frac{\partial p}{\partial \hat{y}}} \frac{\partial \hat{y}}{\partial w_2}$$

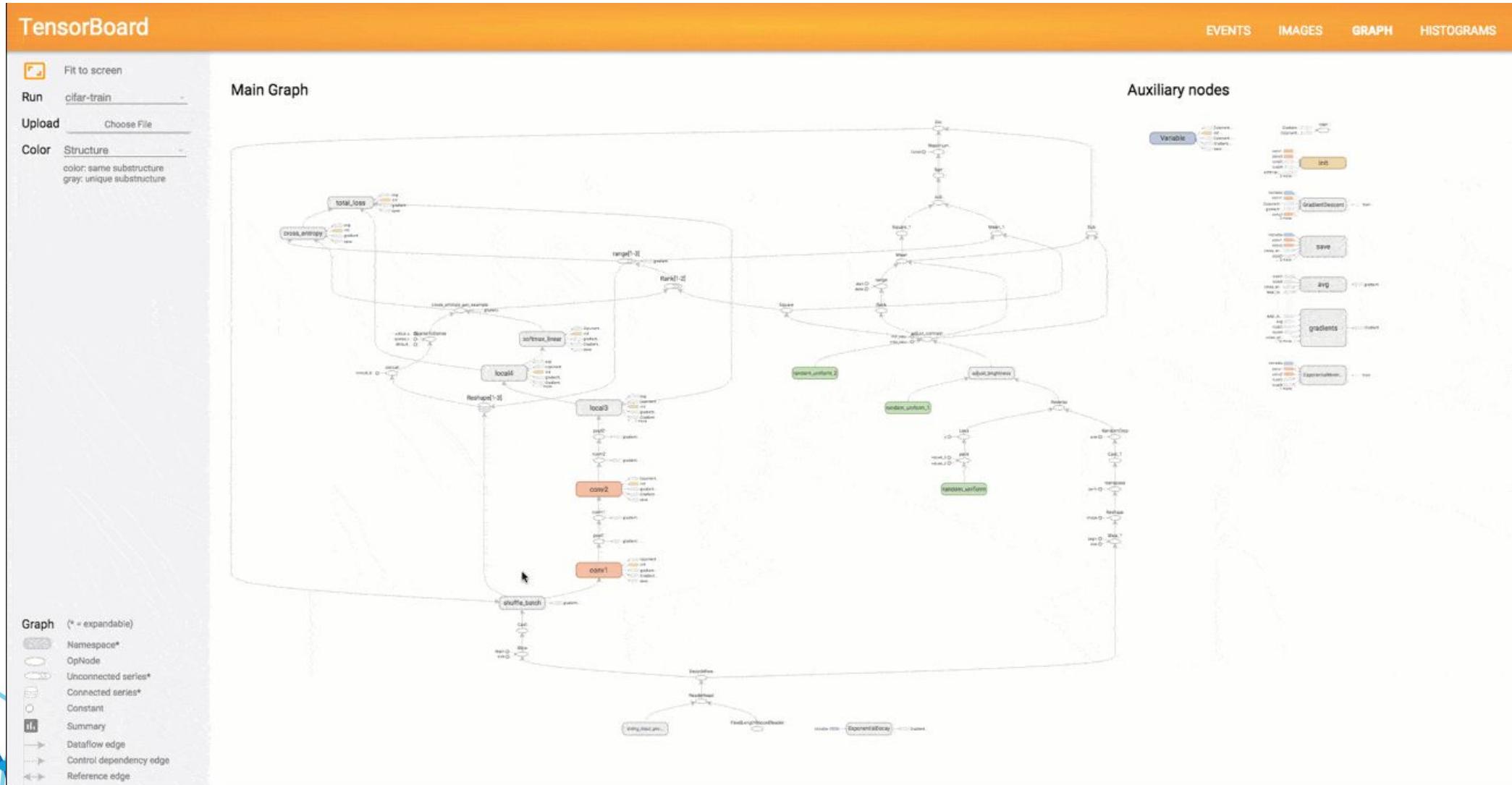
Chain rule for w_1

$$\frac{\partial \mathcal{L}}{\partial w_1} = \boxed{\frac{\partial \mathcal{L}}{\partial p} \frac{\partial p}{\partial \hat{y}}} \frac{\partial \hat{y}}{\partial h'} \frac{\partial h'}{\partial h} \frac{\partial h}{\partial w_1}$$

Backpropagation

$$\theta^{k+1} := \theta^k - \lambda \frac{\partial \mathcal{L}}{\partial \theta^k} \Big|_{\text{mini-batch}}$$

Everything is a graph...



Backpropagation

$$\theta^{k+1} := \theta^k - \lambda \frac{\partial \mathcal{L}}{\partial \theta^k} \Big|_{\text{mini-batch}}$$

What we can learn from that:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial p} \frac{\partial p}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h'} \frac{\partial h'}{\partial h} \frac{\partial h}{\partial w_1}$$

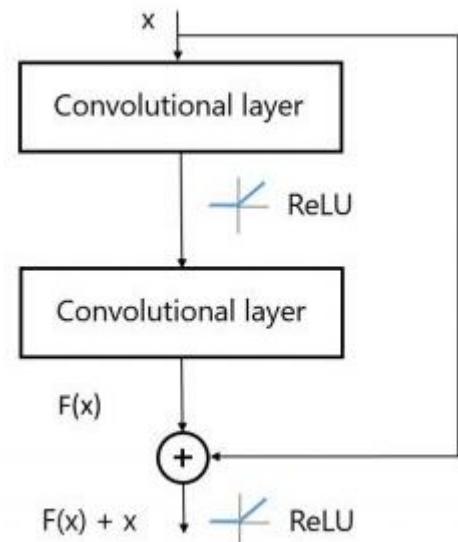
- every differentiable (almost everywhere) operation can be represented as a node in computational graph
- if one of the factors is close to zero, gradient vanishes, hence no update
- oppositely, many multiplication of large numbers may lead to exploding gradients problem

Backpropagation

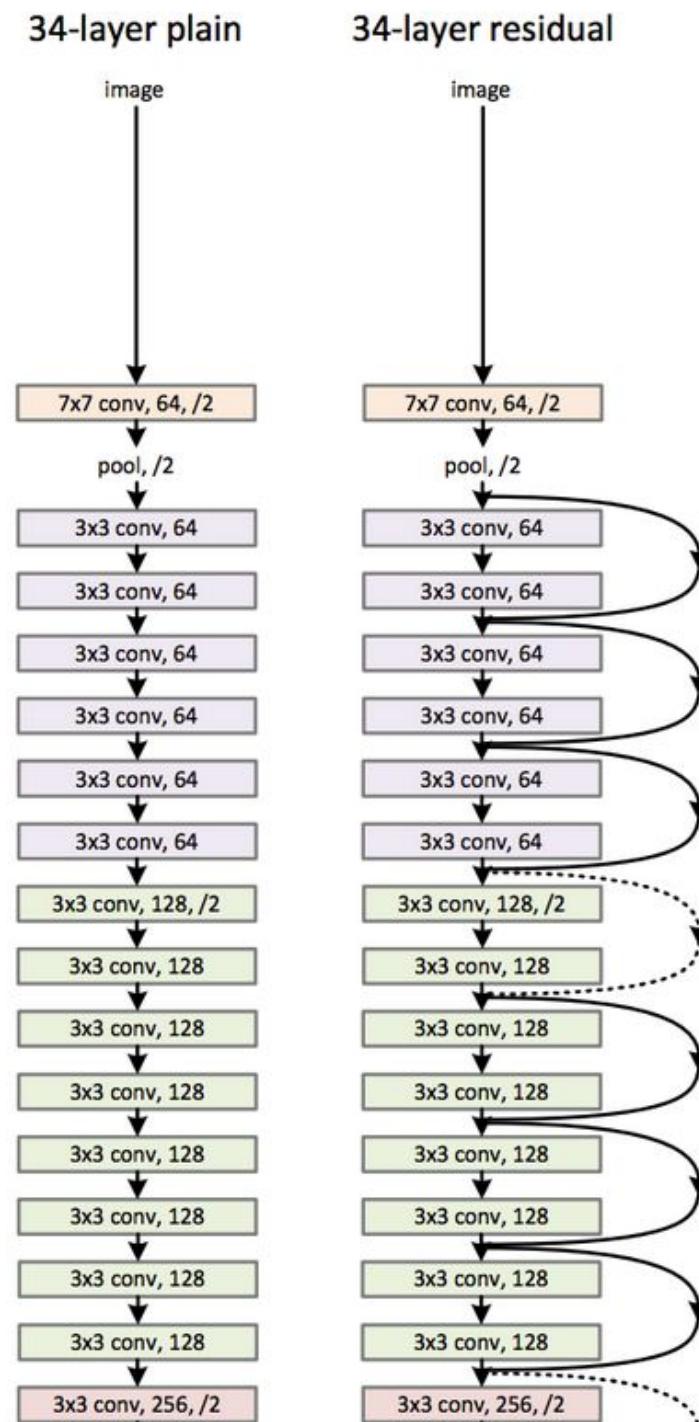
What we can learn from that:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial p} \frac{\partial p}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h'} \frac{\partial h'}{\partial h} \frac{\partial h}{\partial w_1}$$

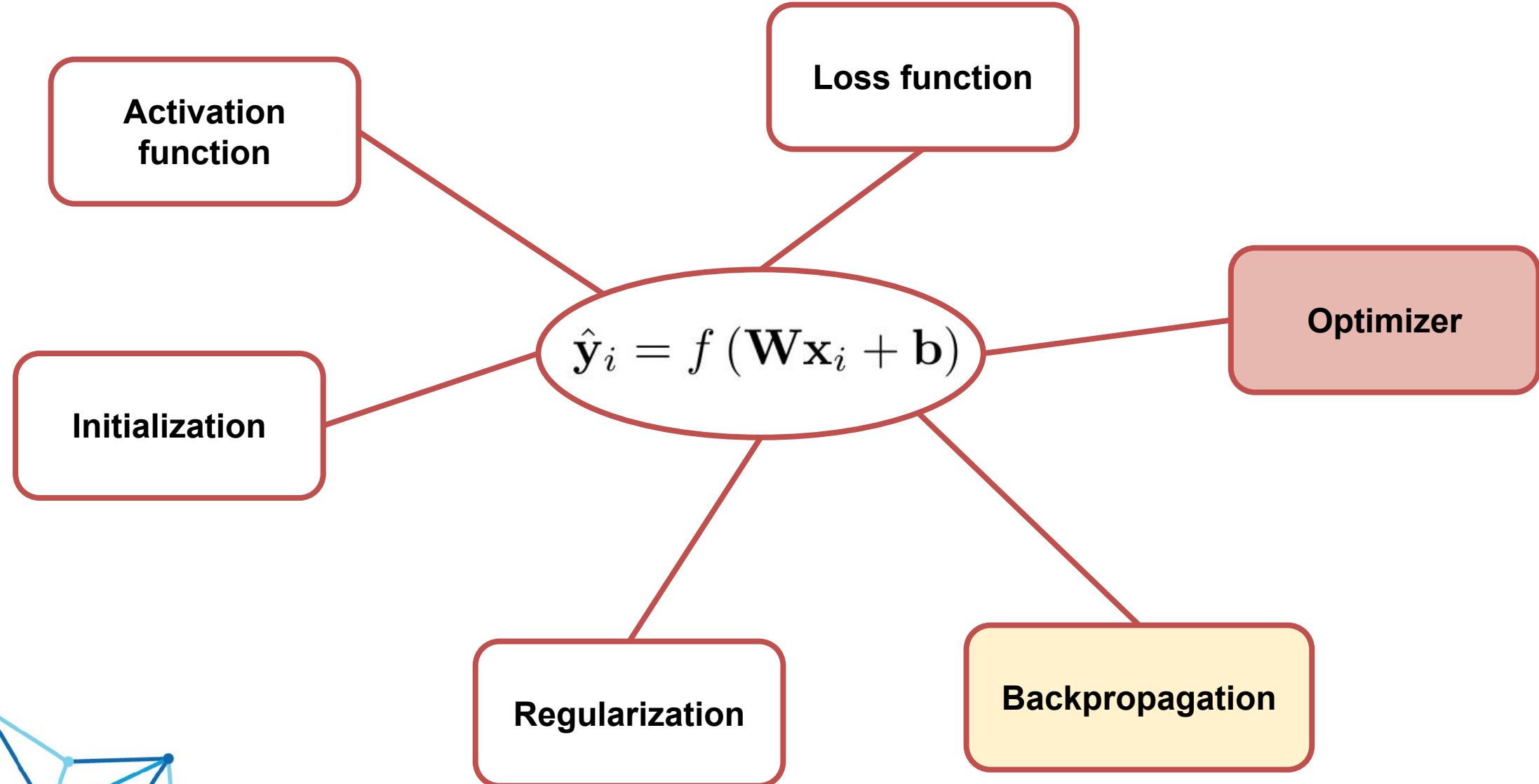
- We problem of vanishing gradients can be solve with skip connections: ResNet architecture
- ImageNet 2015 winning solution



in such case the
gradient can propagate
by two branches



Neural Networks



Optimizer

We know how the gradients are computed - backpropagation i.e. chain rule

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial p} \frac{\partial p}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h'} \frac{\partial h'}{\partial h} \frac{\partial h}{\partial w_1}$$

gradients are used to update model parameters:

$$\theta^{k+1} := \theta^k - \lambda \left. \frac{\partial \mathcal{L}}{\partial \theta^k} \right|_{\text{mini-batch}}$$

- **Batch gradient descent** - the simplest and the slowest one, compute gradients for all data

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta)$$

Optimizer

We know how the gradients are computed - backpropagation i.e. chain rule

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial p} \frac{\partial p}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h'} \frac{\partial h'}{\partial h} \frac{\partial h}{\partial w_1}$$

gradients are used to
update model
parameters:

$$\theta^{k+1} := \theta^k - \lambda \left. \frac{\partial \mathcal{L}}{\partial \theta^k} \right|_{\text{mini-batch}}$$

- **Batch gradient descent** - the simplest and the slowest one, compute gradients for all data
- **Stochastic gradient descent (SGD)** - (mini-batch version) - compute gradients for set of examples, typically ~10 to ~100 examples in batch

Optimizer

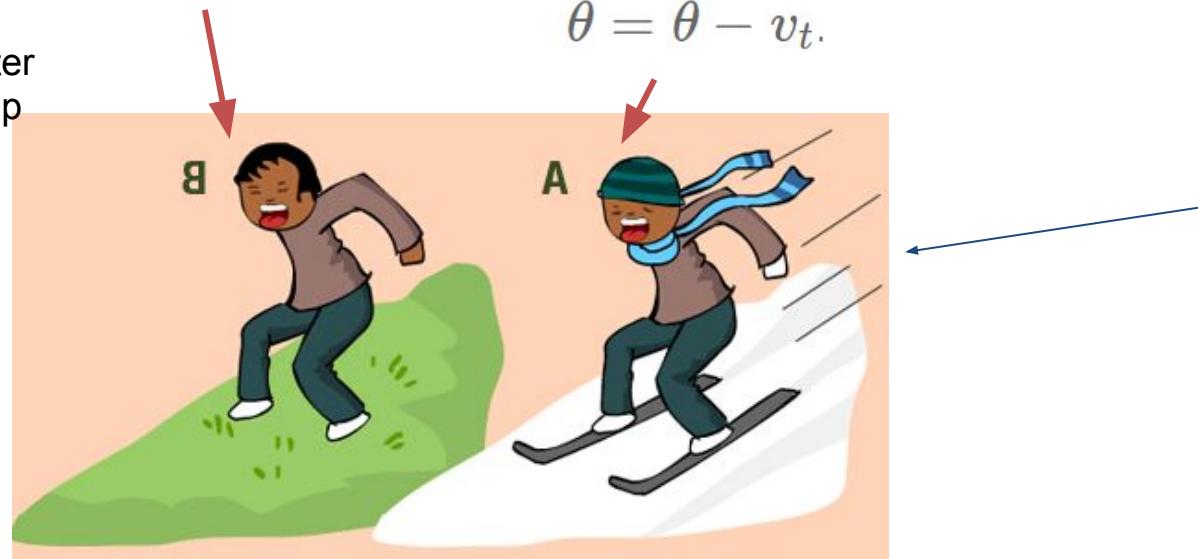


- **Batch gradient descent** - the simplest and the slowest one, compute gradients for all data
- **Stochastic gradient descent (SGD)** - (mini-batch version) - compute gradients for set of examples, typically ~10 to ~100 examples in batch
- **Momentum** - improves SGD by adding velocity term which accumulates gradient vector during training

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta)$$

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta).$$

$$\theta = \theta - v_t.$$



SGD
stops after
each step

gamma - momentum term, usually 0.9, when 0 we recover SGD

It simulates Newton dynamics problem with gradient being the force

Note: Momentum equal 1 means no friction, the movement will never stop

Optimizer



- **Batch gradient descent** - the simplest and the slowest one, compute gradients for all data
- **Stochastic gradient descent (SGD)** - (mini-batch version) - compute gradients for set of examples, typically ~10 to ~100 examples in batch
- **Momentum** - improves SGD by adding velocity term which accumulates gradient vector during training
- **Nesterov accelerated gradient (NAG)** - extension to the momentum method, compute gradient by approximating position in the next step

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta).$$



$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1}).$$

$$\theta = \theta - v_t.$$

$$\theta = \theta - v_t.$$



Optimizer



- **Batch gradient descent** - the simplest and the slowest one, compute gradients for all data
- **Stochastic gradient descent (SGD)** - (mini-batch version) - compute gradients for set of examples, typically ~10 to ~100 examples in batch
- **Momentum** - improves SGD by adding velocity term which accumulates gradient vector during training
- **Nesterov accelerated gradient (NAG)** - extension to the momentum method, compute gradient by approximating position in the next step
- **Adagrad** (Duchi, J 2011)- performs adaptive tuning of learning rate based on model parameters

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla_{\theta} J(\theta_i)$$

Gt is a sum of square gradients accumulated over all time steps

```
dLdw_cache += dLdw**2  
W += - learning_rate * dLdw / (np.sqrt(dLdw_cache) + eps)
```



Optimizer

- **Batch gradient descent** - the simplest and the slowest one, compute gradients for all data
- **Stochastic gradient descent (SGD)** - (mini-batch version) - compute gradients for set of examples, typically ~10 to ~100 examples in batch
- **Momentum** - improves SGD by adding velocity term which accumulates gradient vector during training
- **Nesterov accelerated gradient (NAG)** - extension to the momentum method, compute gradient by approximating position in the next step
- **Adagrad** (Duchi, J 2011)- performs adaptive tuning of learning rate based on model parameters
- **Adadelta** (Zeiler, M. D. (2012)), **RMSprop** (G. Hinton) - extensions to Adagrad, reduces the problem with monotonically decreasing learning rate by

```
dLdw_cache = dLdw_cache*decay_rate + dLdw**2*(1 - decay_rate)
W += - learning_rate * dLdw / (np.sqrt(dLdw_cache) + eps)
```

RMSprop implementation: decay_rate is usually 0.9

Optimizer

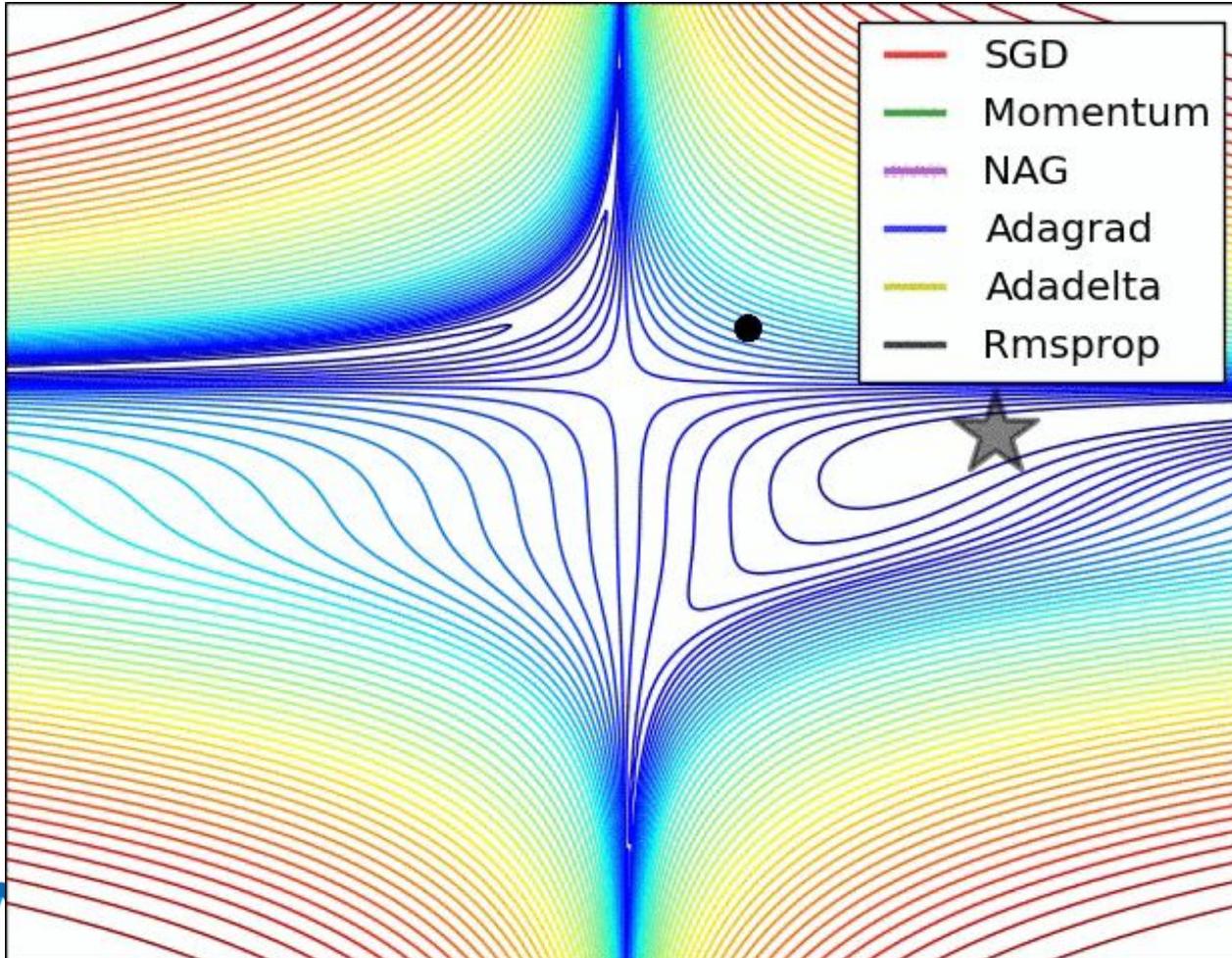


- **Batch gradient descent** - the simplest and the slowest one, compute gradients for all data
- **Stochastic gradient descent (SGD)** - (mini-batch version) - compute gradients for set of examples, typically ~10 to ~100 examples in batch
- **Momentum** - improves SGD by adding velocity term which accumulates gradient vector during training
- **Nesterov accelerated gradient (NAG)** - extension to the momentum method, compute gradient by approximating position in the next step
- **Adagrad** (Duchi, J 2011)- performs adaptive tuning of learning rate based on model parameters
- **Adadelta** (Zeiler, M. D. (2012)), **RMSprop** (G. Hinton) - extensions to Adagrad, reduces the problem with monotonically decreasing learning rate by
- **Adam** (Adaptive Moment Estimation) (Kingma, D. P., & Ba, J. L. (2015)) - a marriage of **momentum** and **RMSprop** methods



Optimizer

Comparison of different optimizers

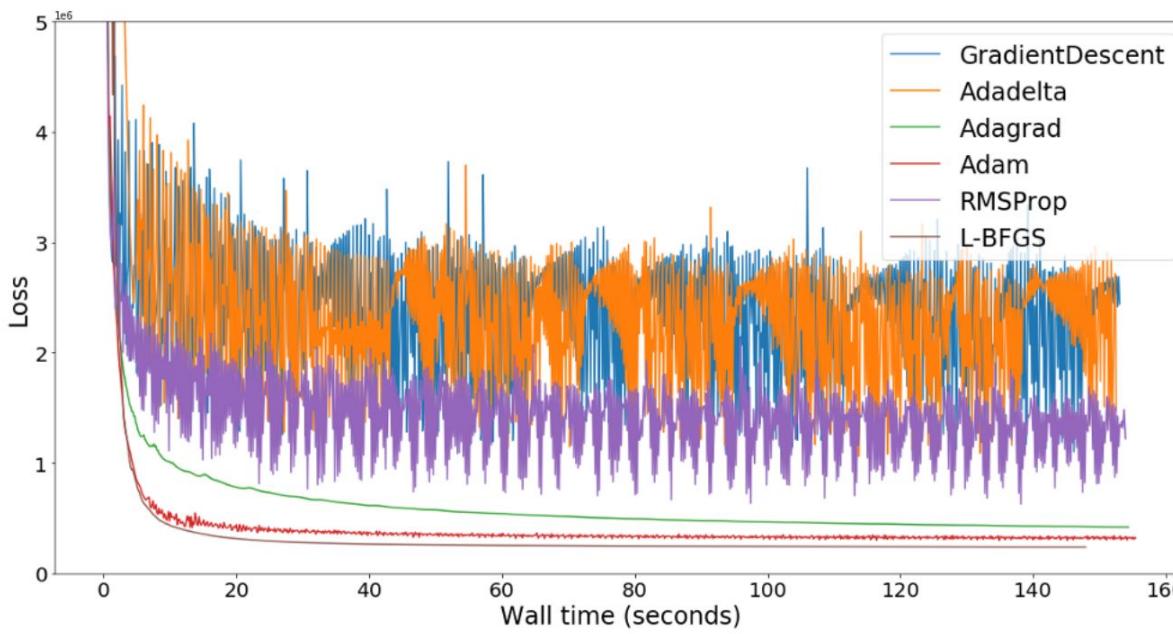


Now it is recommended to use Adam, Adadelta, RMSprop

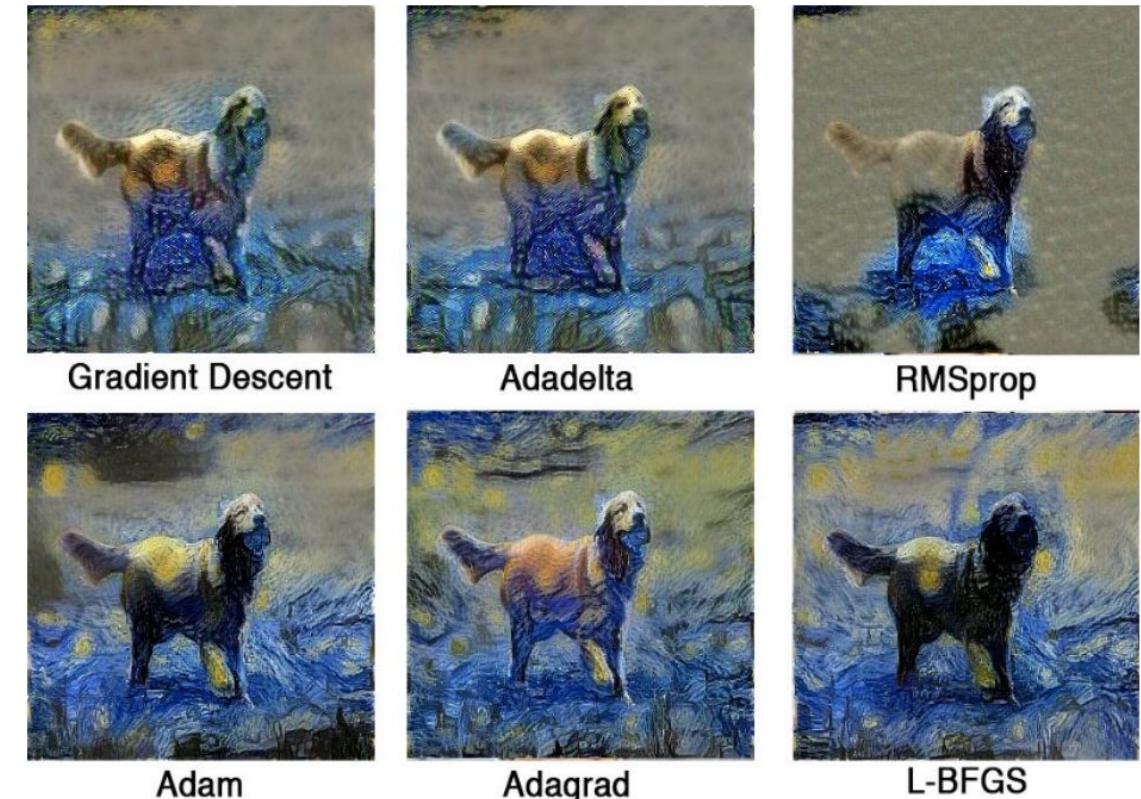
Optimizer



The choice of the optimizer may significantly affect model predictions



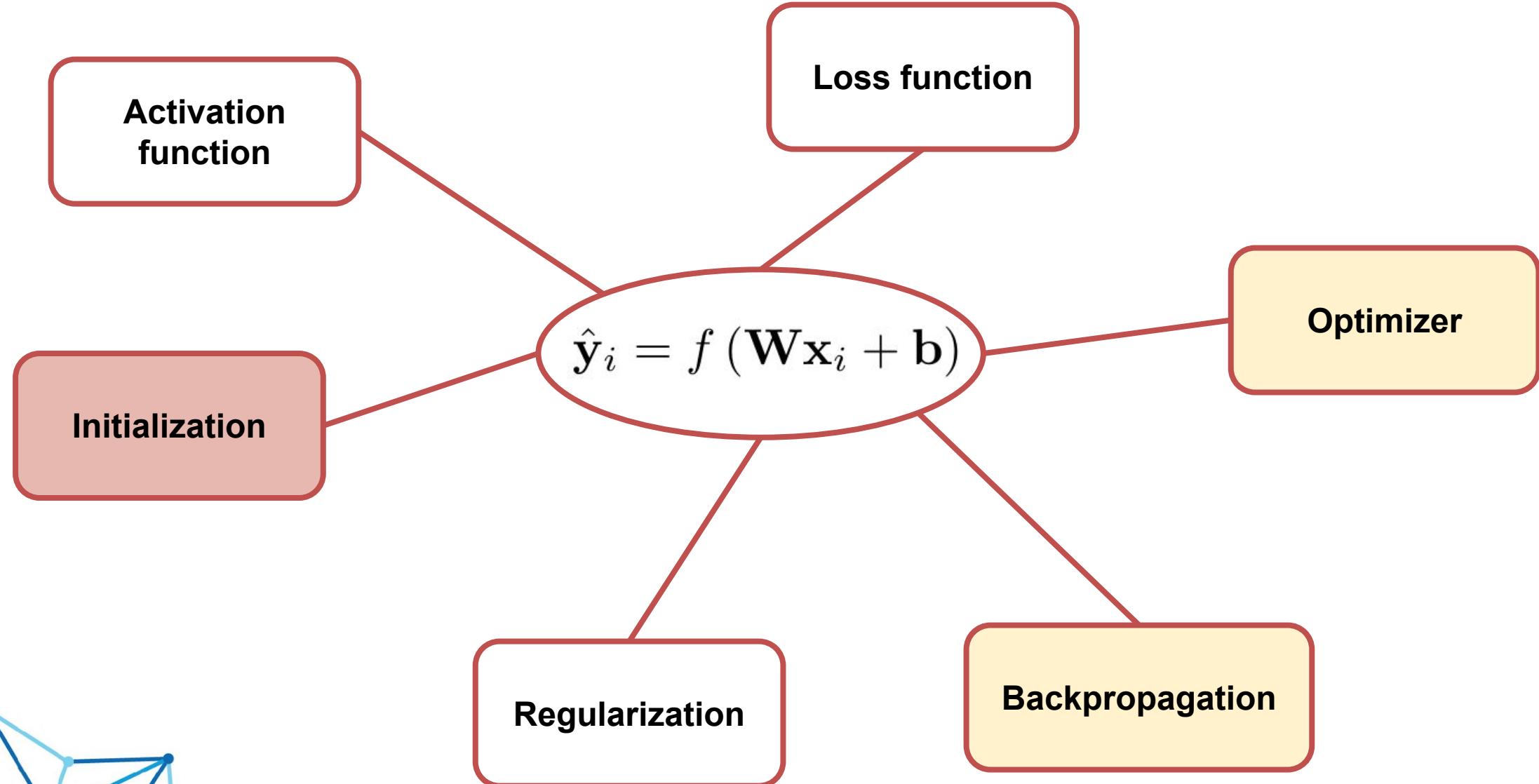
Style transfer: which optimizer choose to get the best results?



$$x \leftarrow x - [Hf(x)]^{-1} \nabla f(x)$$

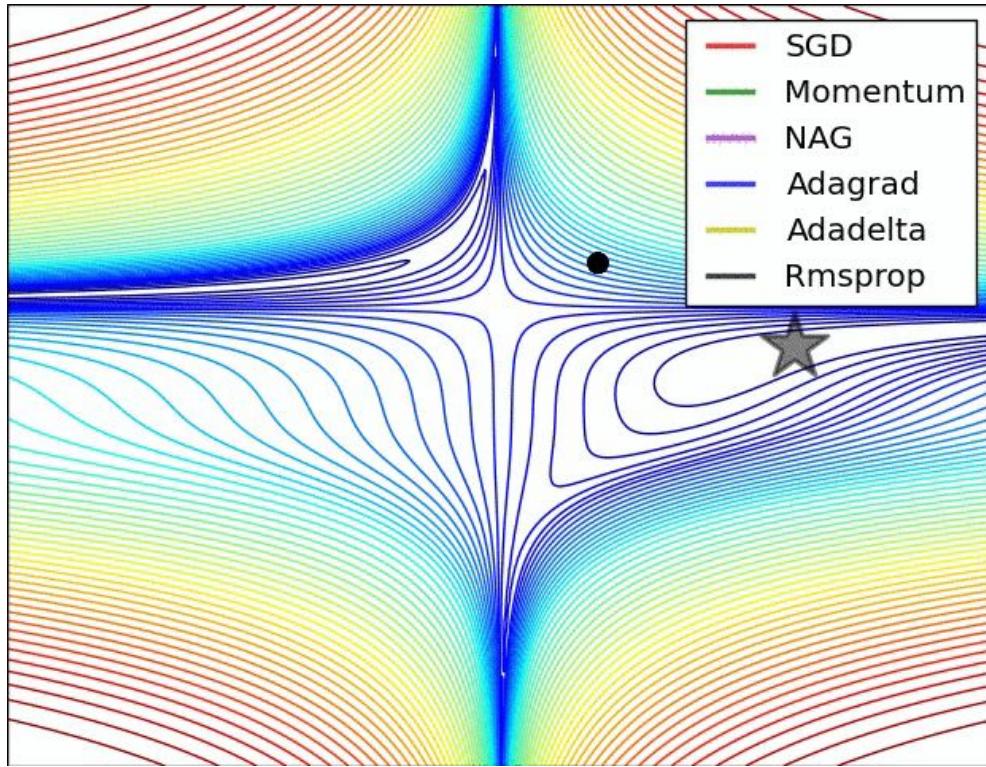
source: [Slav Ivanov blog post](#)

Neural Networks



Initialization

We've seen the performance of each methods on the convergence:



Images credit: [Alec Radford](#).

- One of the biggest problems in DL was improper initialization of system parameters
- Mostly solved thanks to normalization techniques (**discussed later in the talk**)
- We initialize weights with some small random numbers
- This helps to break symmetries in our model

Probably the most popular is **Glorot, Bengio (2010)** initialization:

$$W_{i,j} \sim U\left(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}\right)$$

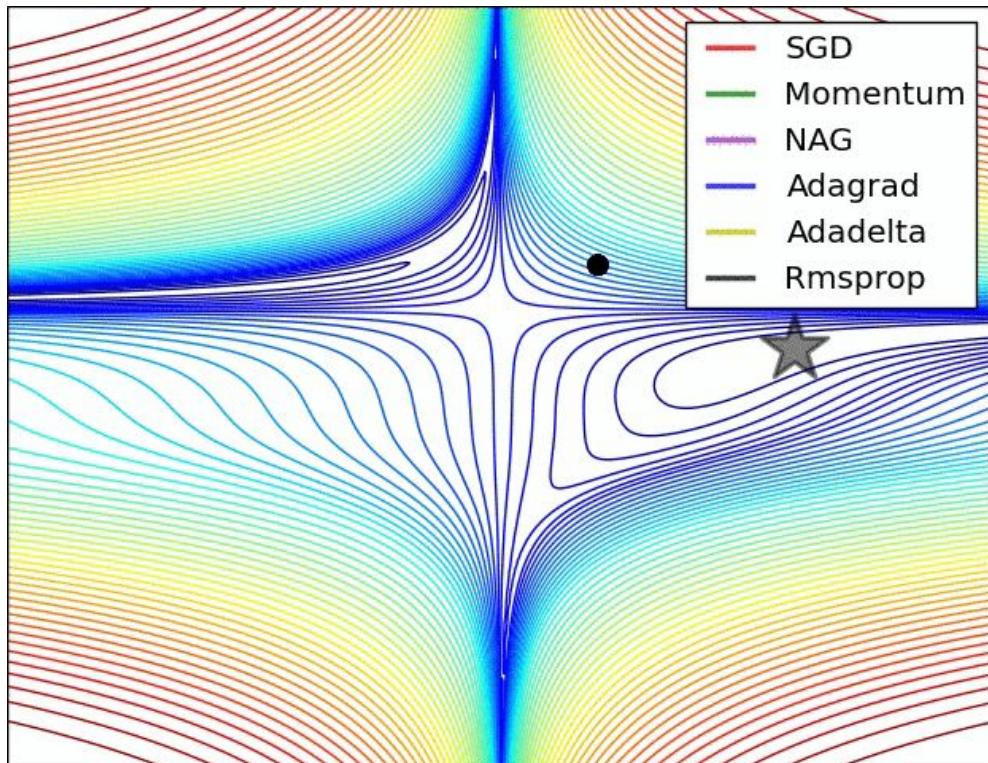


m/n - number of input.output units

We usually initialize bias vector with zero

Initialization

We've seen the performance of each methods on the convergence:



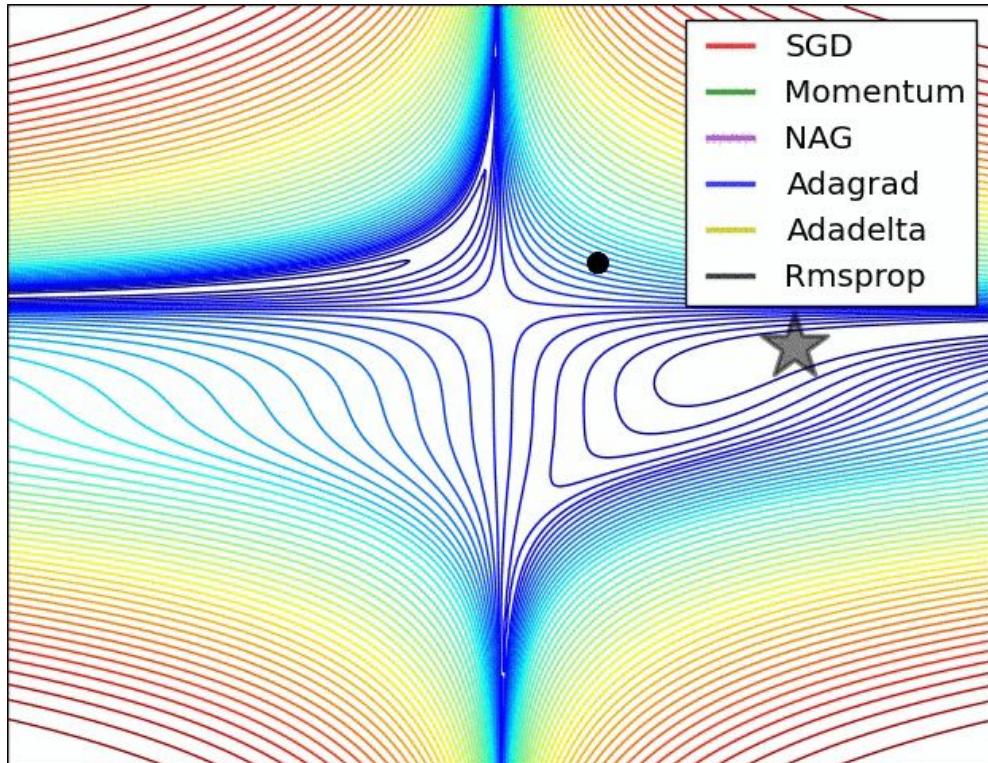
Images credit: [Alec Radford](#).

- One of the biggest problems in DL was improper initialization of system parameters
- Mostly solved thanks to normalization techniques (**discussed later in the talk**)
- We initialize weights with some small random numbers
- This helps to break symmetries in our model

Less popular but worth to try: **orthogonal initialization (Saxe et al. (2013))**: orthogonal matrices preserve the norm, hence we can mostly avoid the vanishing and exploding gradients.

Initialization

We've seen the performance of each methods on the convergence:



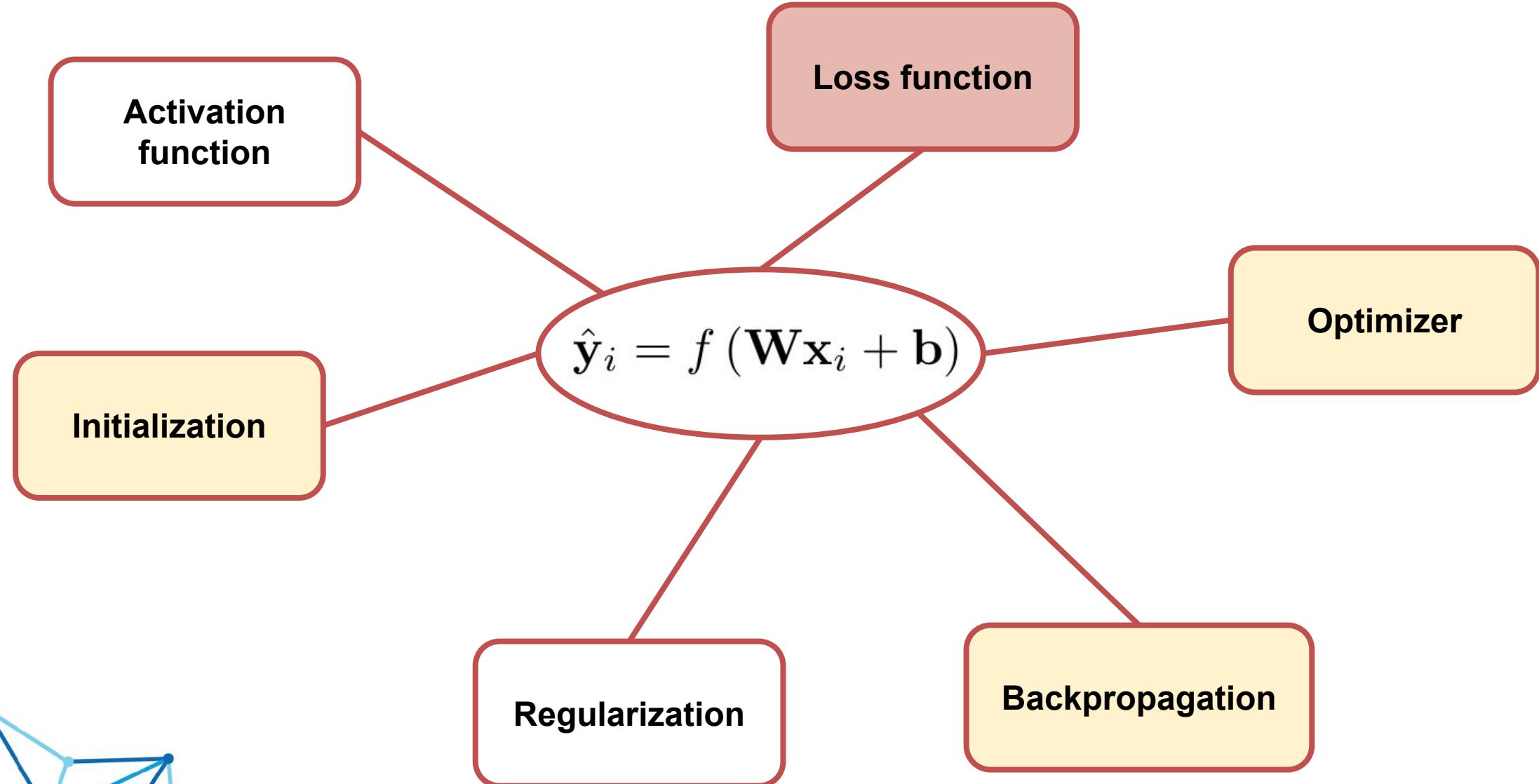
Images credit: [Alec Radford](#).

- One of the biggest problems in DL was improper initialization of system parameters
- Mostly solved thanks to normalization techniques (**discussed later in the talk**)

- We initialize weights with some small random numbers
- This helps to break symmetries in our model

Sparse initialization (Martens (2010)) - each unit is initialized to have k non-zero weights

Neural Networks



Loss function



By choosing loss function we shape the gradient flow in the network

$$\frac{\partial \mathcal{L}}{\partial w_1} = \boxed{\frac{\partial \mathcal{L}}{\partial p}} \frac{\partial p}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h'} \frac{\partial h'}{\partial h} \frac{\partial h}{\partial w_1}$$

The type of used loss function will depend on the type of problem we want to solve

Classification problem:

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

hinge loss

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

softmax loss

also squared hinge loss



Loss function

Regression problem: prediction of real valued quantities (e.g. average price)

$$L_i = \|f - y_i\|_2^2$$

squared L2 norm

$$L_i = \|f - y_i\|_1 = \sum_j |f_j - (y_i)_j|$$

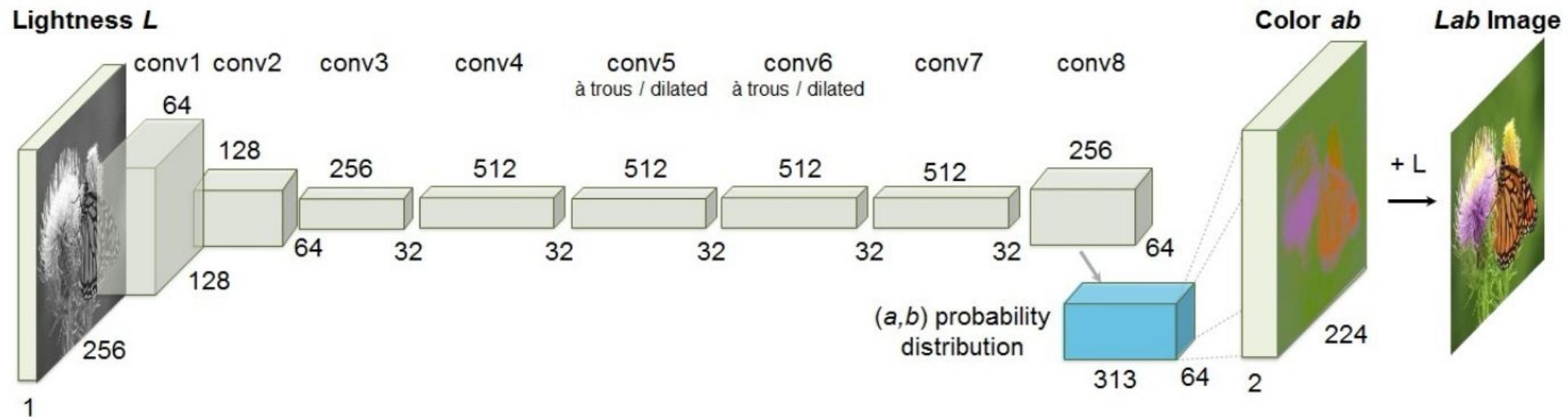
L1 norm

and many others: cosine proximity, KL divergence, ...

Loss function

Choosing loss function may have very serious consequences on model accuracy

Task: Given the input image in grayscale predict its pixel color:



Zhang, Isola, Efros. Colorful Image Colorization. ECCV, 2016.

Given an input lightness channel $\mathbf{X} \in \mathbb{R}^{H \times W \times 1}$, our objective is to learn a mapping $\hat{\mathbf{Y}} = \mathcal{F}(\mathbf{X})$ to the two associated color channels $\mathbf{Y} \in \mathbb{R}^{H \times W \times 2}$, where H, W are image dimensions.

Loss function

Choosing loss function may have very serious consequences on model accuracy

Given an input lightness channel $\mathbf{X} \in \mathbb{R}^{H \times W \times 1}$, our objective is to learn a mapping $\hat{\mathbf{Y}} = \mathcal{F}(\mathbf{X})$ to the two associated color channels $\mathbf{Y} \in \mathbb{R}^{H \times W \times 2}$, where H, W are image dimensions.

$$L_2(\hat{\mathbf{Y}}, \mathbf{Y}) = \frac{1}{2} \sum_{h,w} \|\mathbf{Y}_{h,w} - \hat{\mathbf{Y}}_{h,w}\|_2^2$$

A natural choice of loss function could be L2 norm

Instead of they use multinomial classification problem on ab color

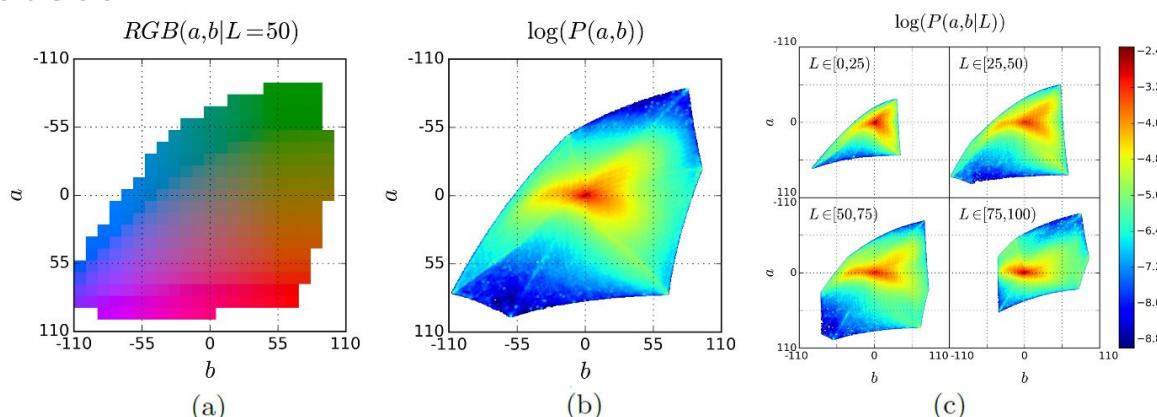


Fig. 3. (a) Quantized ab color space with a grid size of 10. A total of 313 ab pairs are in gamut. (b) Empirical probability distribution of ab values, shown in log scale. (c) Empirical probability distribution of ab values, conditioned on L, shown in log scale.

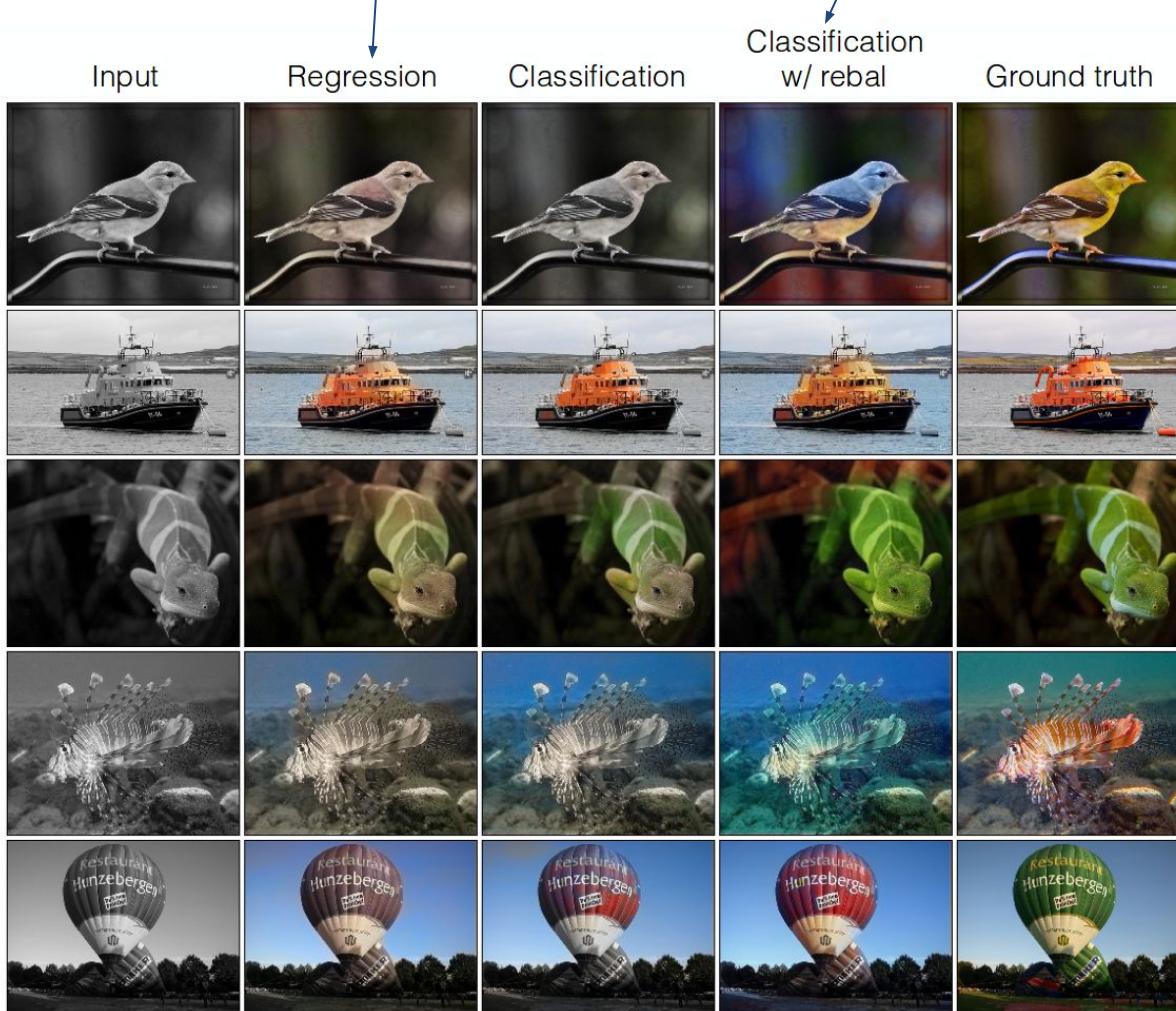
$$L_{cl}(\hat{\mathbf{Z}}, \mathbf{Z}) = - \sum_{h,w} v(\mathbf{Z}_{h,w}) \sum_q \mathbf{Z}_{h,w,q} \log(\hat{\mathbf{Z}}_{h,w,q})$$

Weighted cross-entropy loss

Loss function

Choosing loss function may have very serious consequences on model accuracy

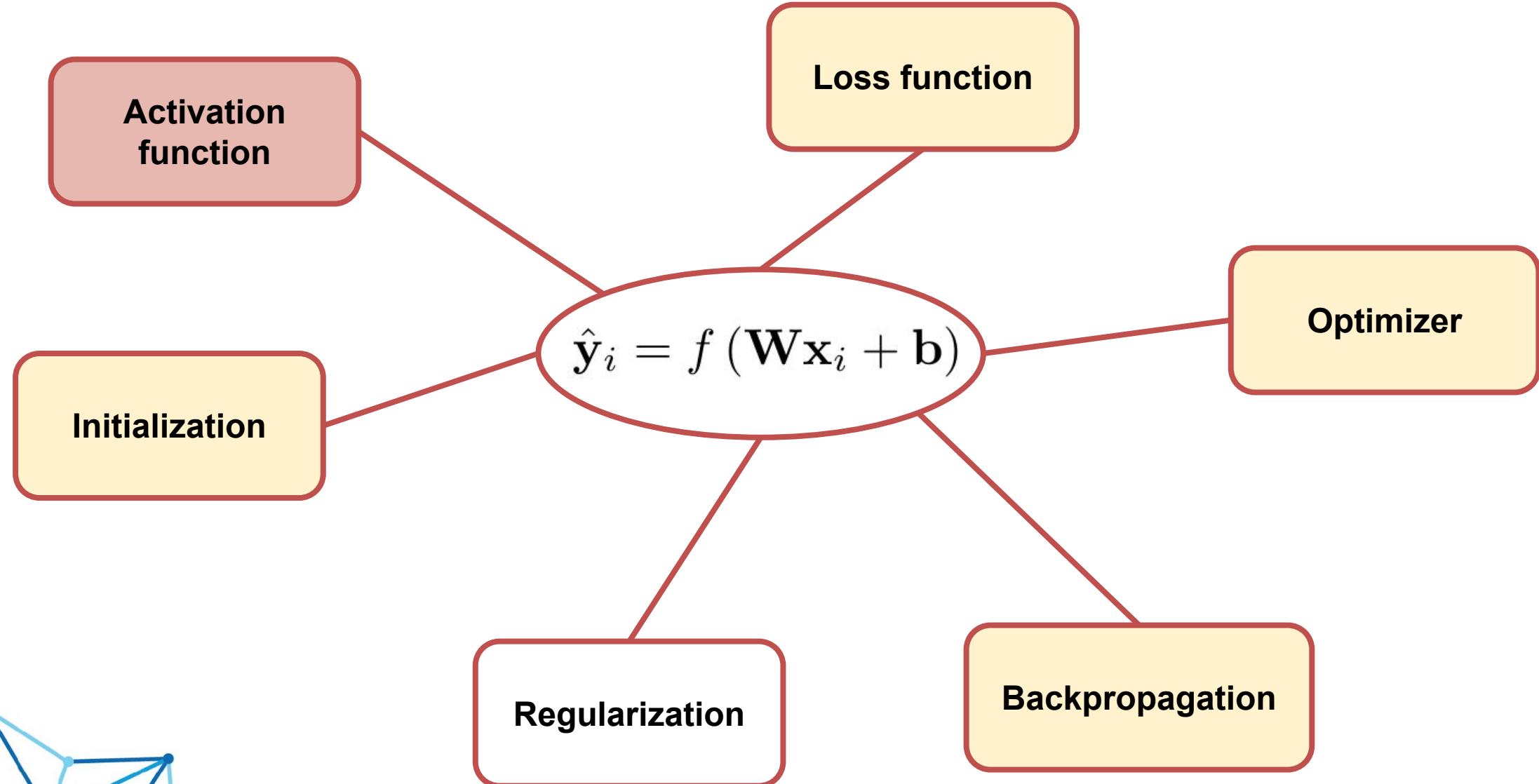
$$L_2(\hat{\mathbf{Y}}, \mathbf{Y}) = \frac{1}{2} \sum_{h,w} \|\mathbf{Y}_{h,w} - \hat{\mathbf{Y}}_{h,w}\|_2^2 \quad L_{cl}(\hat{\mathbf{Z}}, \mathbf{Z}) = - \sum_{h,w} v(\mathbf{Z}_{h,w}) \sum_q \mathbf{Z}_{h,w,q} \log(\hat{\mathbf{Z}}_{h,w,q})$$



Colorization of old and out of the dataset images:



Neural Networks



Activation functions - zoo



Activation function is a necessary building block which introduces nonlinearity to increase capacity of neural network

$$\hat{\mathbf{y}}_i = f(\mathbf{W}\mathbf{x}_i + \mathbf{b})$$

We have already seen some of them:

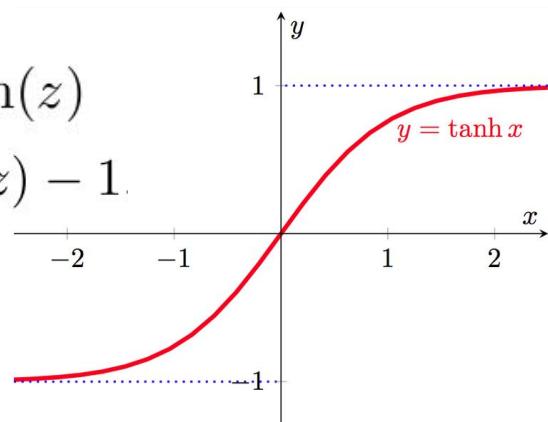
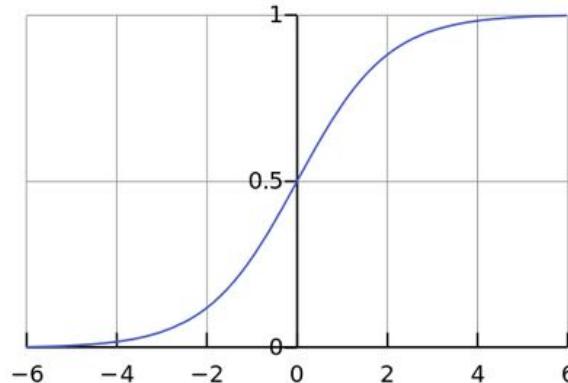
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

hard to train because of saturated gradient

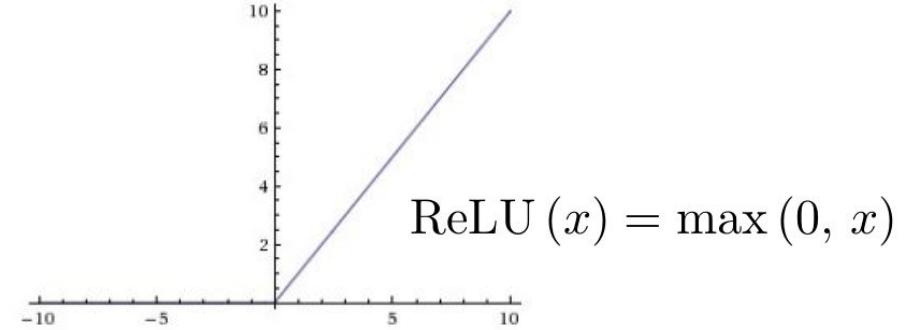
Hyperbolic tangent:

$$g(z) = \tanh(z)$$

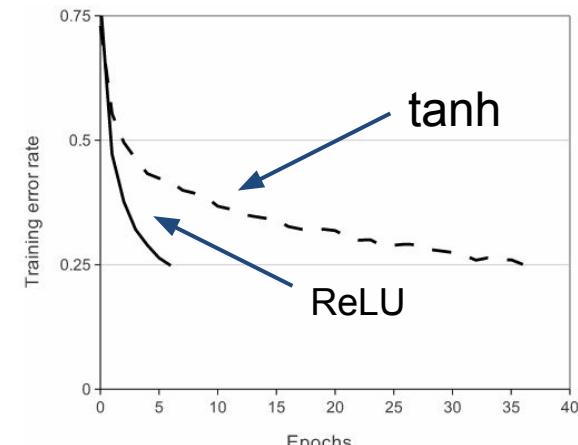
$$\tanh(z) = 2\sigma(2z) - 1$$



most common and easy to optimize



2012 - It was shown that network with ReLU learnt several times faster with ReLU units

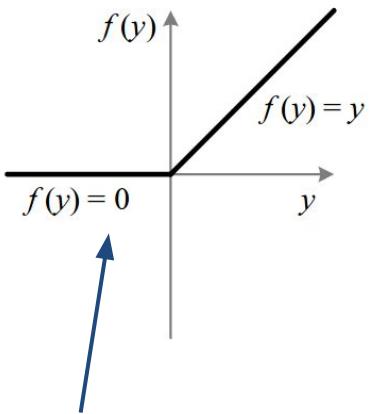


Krizhevsky, A. 2012
ImageNet
winning solution

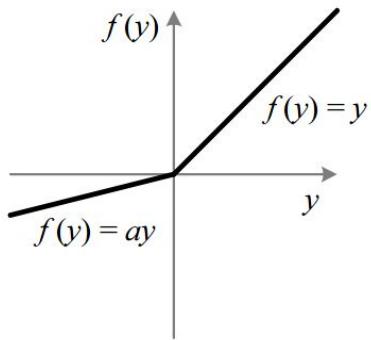
Activation functions - zoo

Activation function is a necessary building block which introduces nonlinearity to increase capacity of neural network

$$\hat{\mathbf{y}}_i = f(\mathbf{W}\mathbf{x}_i + \mathbf{b})$$



dying gradients problem

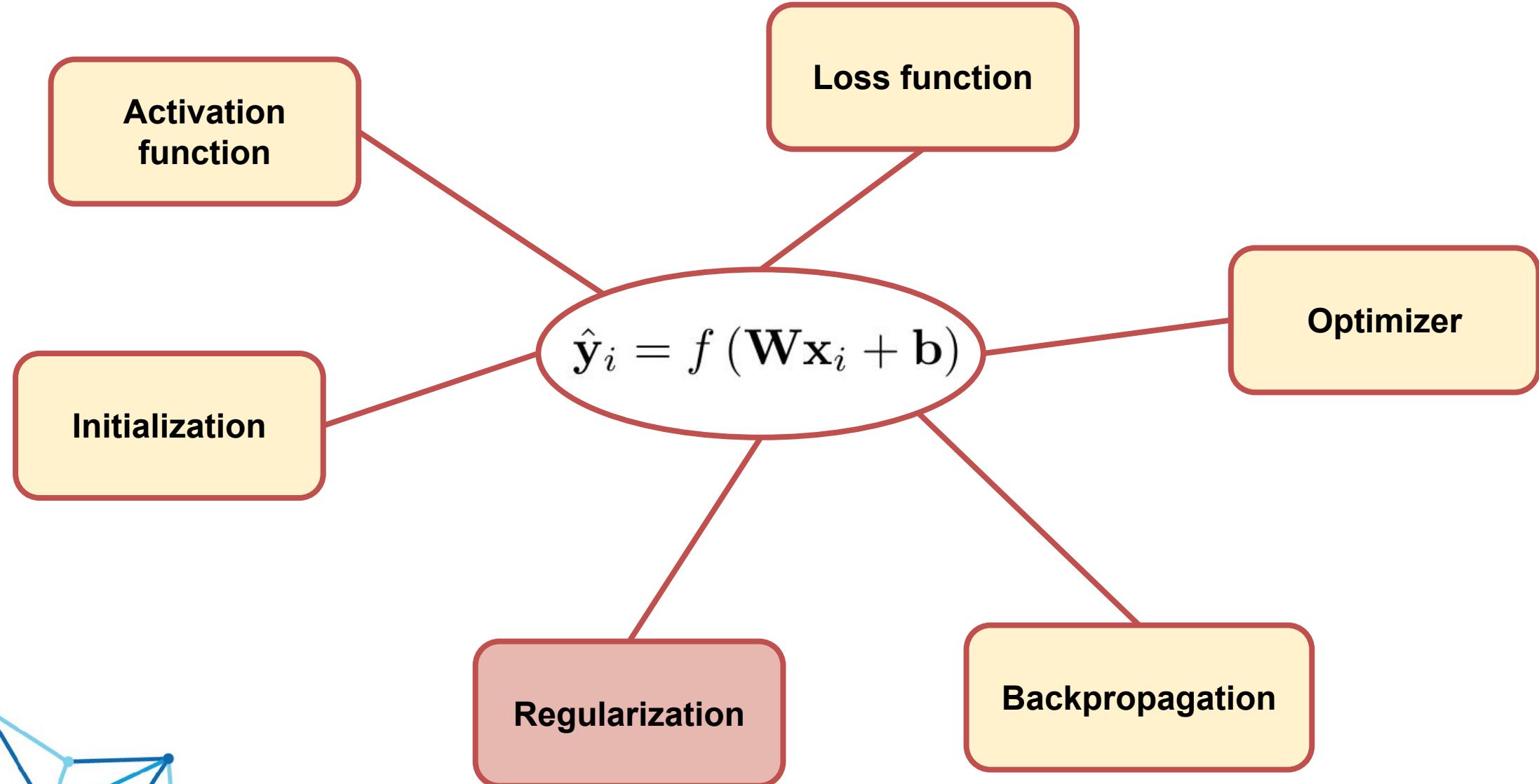


- ReLU
- Leaky ReLU - allows for small negative values
- Parametric ReLU - alpha is learned
- maxout - as a generalization of ReLUs family

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

It is worth to try all activations from ReLU family, but probably one will end up with the simplest one.

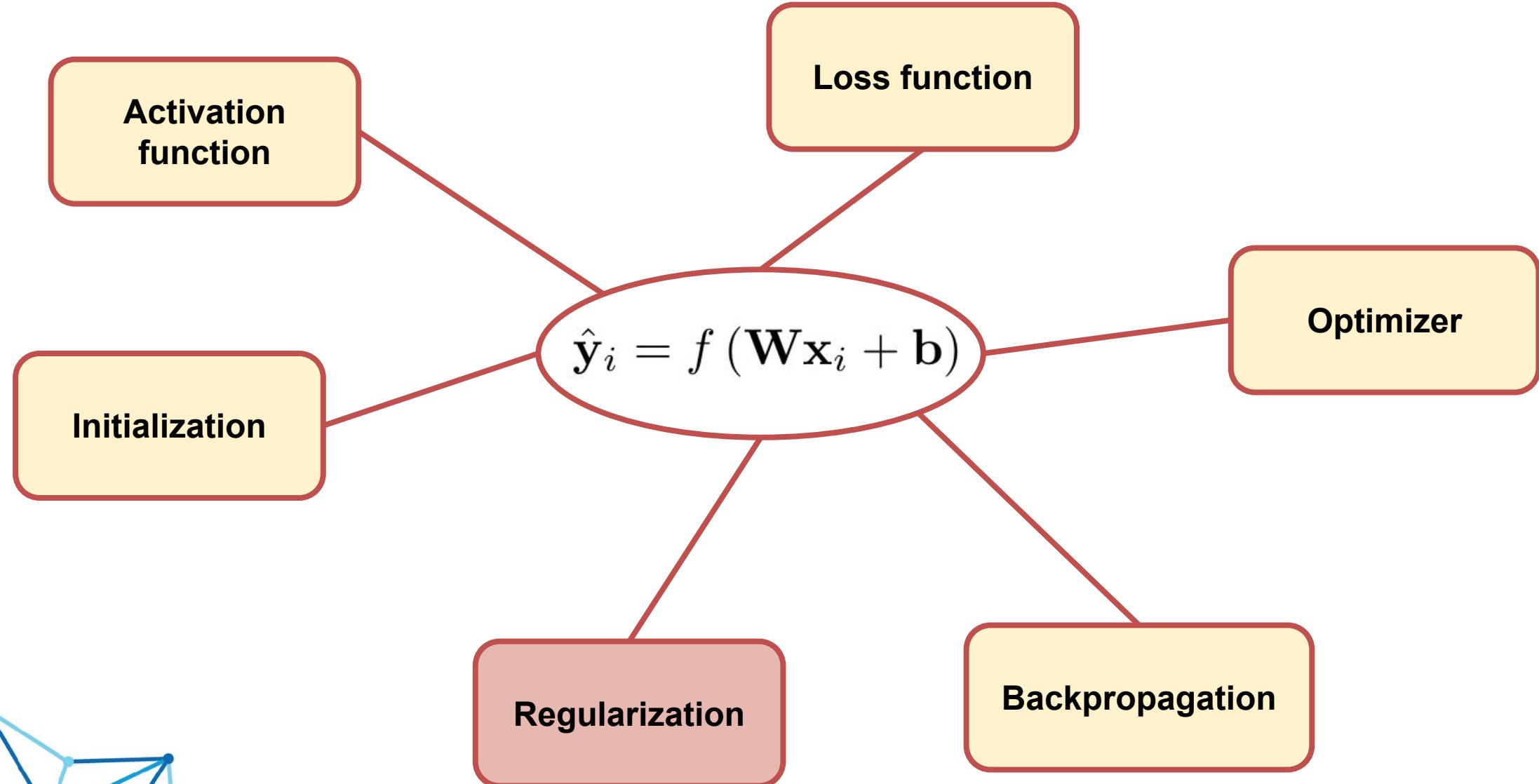
Neural Networks





Short break

Neural Networks



Regularization

Regularization is a way to preventing our model from overfitting

Overfitting is when our model learn to remember data instead to understanding them

Keras overfitting example

Regularization - zoo



Parameter norm penalties

$$\hat{\mathbf{y}}_i = f(\mathbf{W}\mathbf{x}_i + \mathbf{b})$$

we want to put penalties on our model parameters in order to prevent overfitting

Add second term which penalizes special values of model parameters

$$\mathcal{L}_{\text{total}}(\theta; \mathbf{y}, \hat{\mathbf{y}}) = \mathcal{L}_{\text{objective}}(\theta; \mathbf{y}, \hat{\mathbf{y}}) + \mathcal{L}_{\theta}(\theta)$$

- **L2 regularization** $\mathcal{L}(\mathbf{W}; \lambda) = \frac{1}{2}\lambda \|\mathbf{W}\|_2^2$ with gradient $\frac{\partial \mathcal{L}(\mathbf{W}; \lambda)}{\partial \mathbf{W}} = \lambda \mathbf{W}$



usually small number, model hyperparameter

- L2 works as soft constraint on the values of the \mathbf{W} matrix, large values are forbidden since they lead to large loss



Regularization - zoo



Parameter norm penalties

- **L2 regularization** $\mathcal{L}(\mathbf{W}; \lambda) = \frac{1}{2}\lambda \|\mathbf{W}\|_2^2$ with gradient

$$\frac{\partial \mathcal{L}(\mathbf{W}; \lambda)}{\partial \mathbf{W}} = \lambda \mathbf{W}$$

Regularization has smoothing effect on model predictions:
consider following task: separate green dots from red

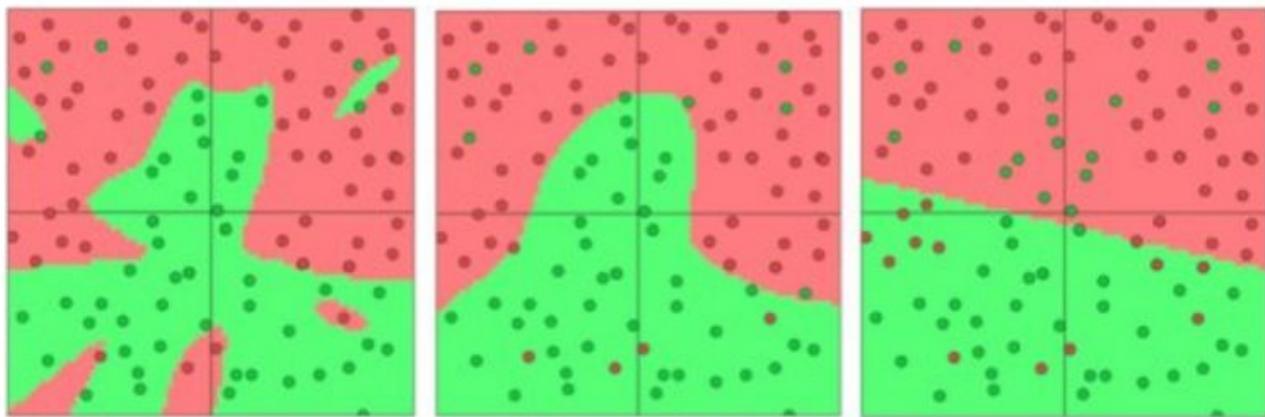


Fig 5. Separating green dots vs red dots, L2 regularization strengths of 0.01, 0.1, and 1

source: kdnuggets.com



Regularization - zoo

Parameter norm penalties

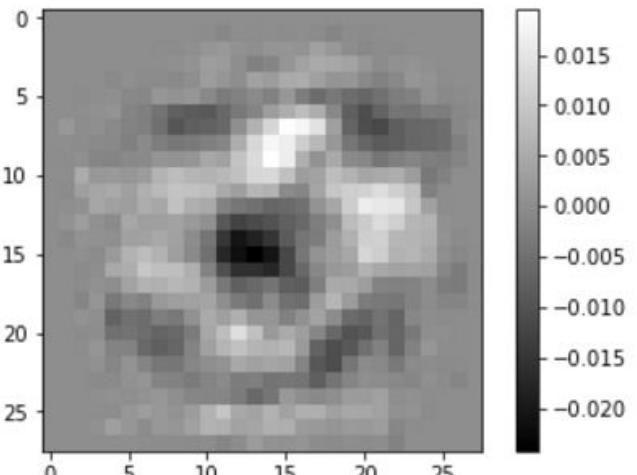
- **L2 regularization** $\mathcal{L}(\mathbf{W}; \lambda) = \frac{1}{2}\lambda \|\mathbf{W}\|_2^2$ with gradient

$$\frac{\partial \mathcal{L}(\mathbf{W}; \lambda)}{\partial \mathbf{W}} = \lambda \mathbf{W}$$

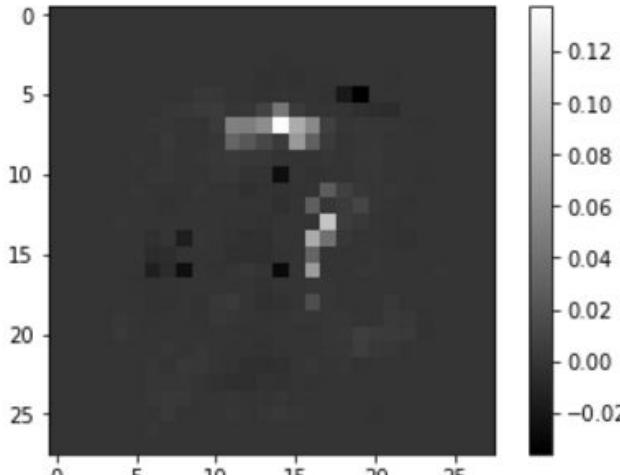
- **L1 regularization** $\mathcal{L}(\mathbf{W}; \lambda) = \lambda |\mathbf{W}|$ with gradient

$$\frac{\partial \mathcal{L}(\mathbf{W}; \lambda)}{\partial \mathbf{W}} = \lambda \text{sign}(\mathbf{W})$$

the L1 regularization leads to a sparse matrices



first “W image” with L2



first “W image” with L1

In practice L2 can be expected to overperform L1 regularization

Regularization - zoo



Constraints regularizers

- max norm constraint - $\|\vec{w}\|_2 < c$
- non negativity constraint - *each element of the W matrix must be positive*
- unit norm - *enforces the matrix to have unit norm along the last axis*

in keras we can do it easily:

```
from keras.constraints import maxnorm  
model.add(Dense(64, kernel_constraint=max_norm(2.)))
```



Regularization - zoo



Data augmentation = more data better generalization => generate more data based on existing ones

- for images one may apply translations, rotations, stretching, shearing, distortion

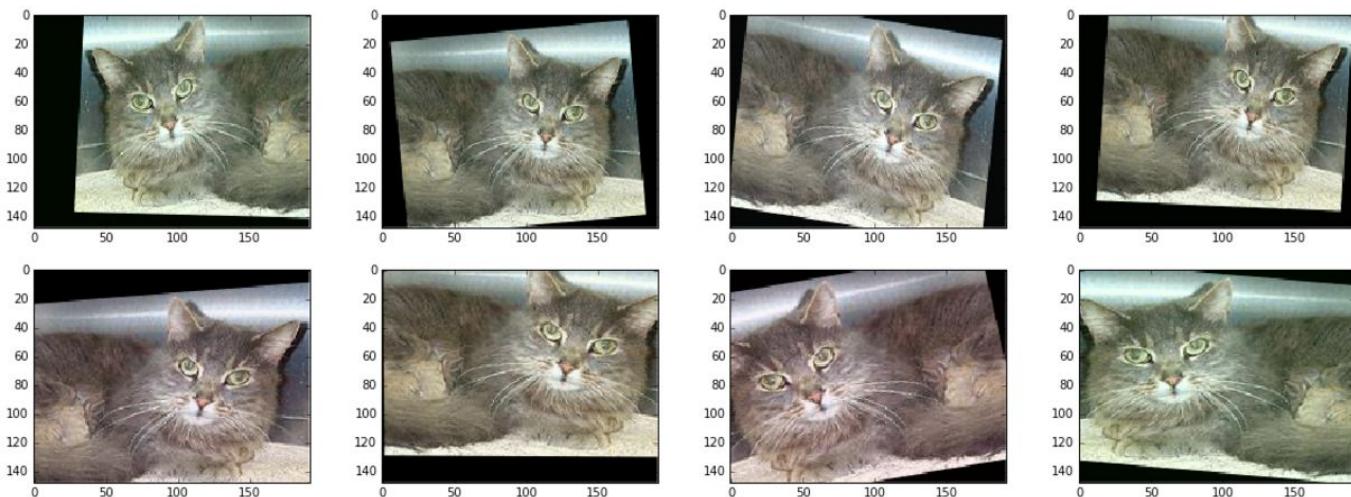
Keras has cool Image generator class which allows for online augmentation during training

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
Y_train = np_utils.to_categorical(y_train, num_classes)
Y_test = np_utils.to_categorical(y_test, num_classes)

datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)

# compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied)
datagen.fit(X_train)

# fits the model on batches with real-time data augmentation:
model.fit_generator(datagen.flow(X_train, Y_train, batch_size=32),
                    steps_per_epoch=len(X_train), epochs=epochs)
```



whenever you can, use it !



Regularization - zoo



Dropout - 2014

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava
Geoffrey Hinton
Alex Krizhevsky
Ilya Sutskever
Ruslan Salakhutdinov

NITISH@CS.TORONTO.EDU
HINTON@CS.TORONTO.EDU
KRIZ@CS.TORONTO.EDU
ILYA@CS.TORONTO.EDU
RSALAKHU@CS.TORONTO.EDU

The idea behind the dropout

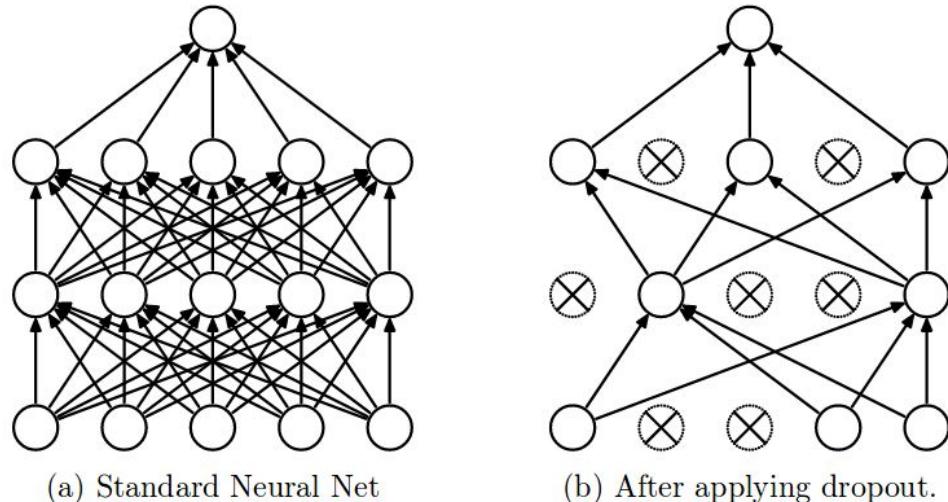


Figure 1: Dropout Neural Net Model. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

source: Srivastava, 2014

Example in keras

$$\begin{aligned} z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}), \end{aligned}$$

Standard Neuron with dropout:

$$\begin{aligned} r_j^{(l)} &\sim \text{Bernoulli}(p), \\ \tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\ z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{y}}^l + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}). \end{aligned}$$

binary mask
created with
probability p

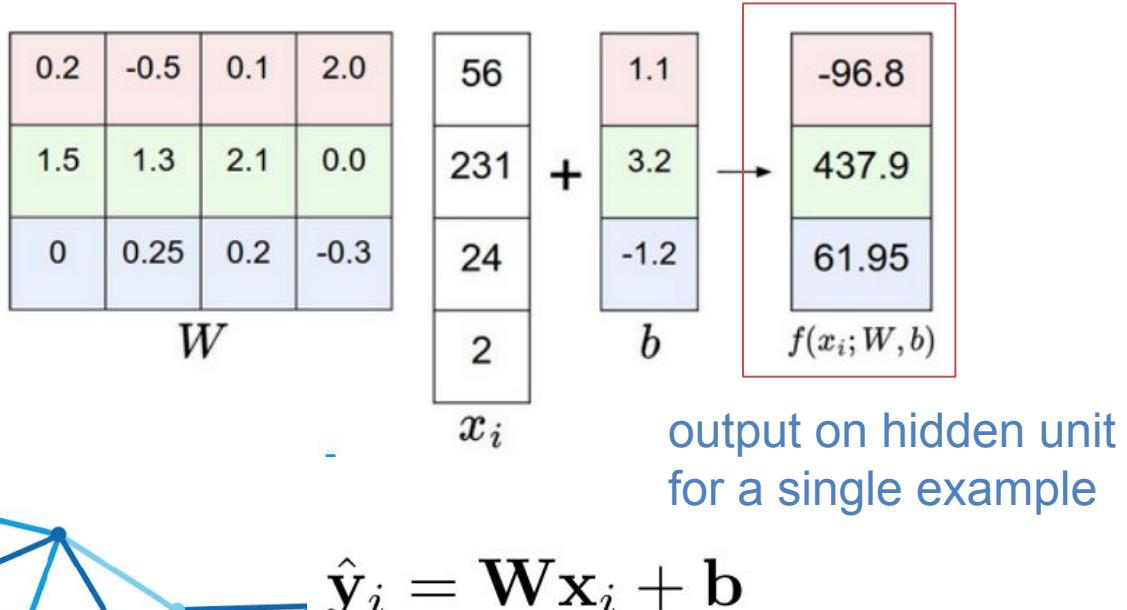
Regularization - zoo



Normalization layers family - prevent overfitting and speed up training

- **Batch normalization (2015)** - introduces normalization operation in a differentiable manner

Normalize the inputs of hidden layer using
mini-batch statistic

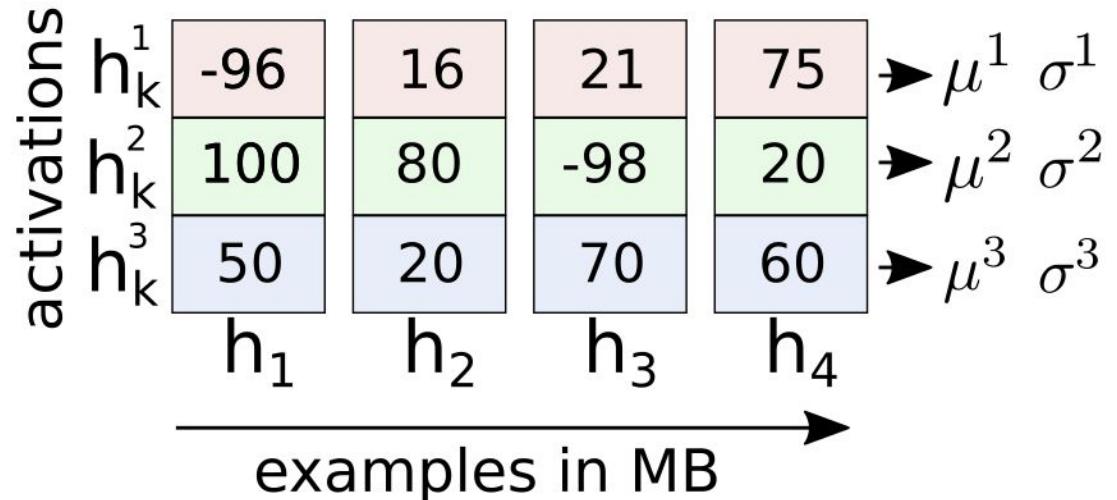


Batch Normalization: Accelerating Deep Network Training by
Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., sioffe@google.com

Christian Szegedy
Google Inc., szegedy@google.com

In mini-batch we have many examples



Regularization - zoo

Normalization layers family - prevent overfitting and speed up training

- **Batch normalization (2015)** - introduces normalization operation in a differentiable manner

Normalize the inputs of hidden layer using mini-batch statistic

The NB algorithm (per particular activation)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

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

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

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

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

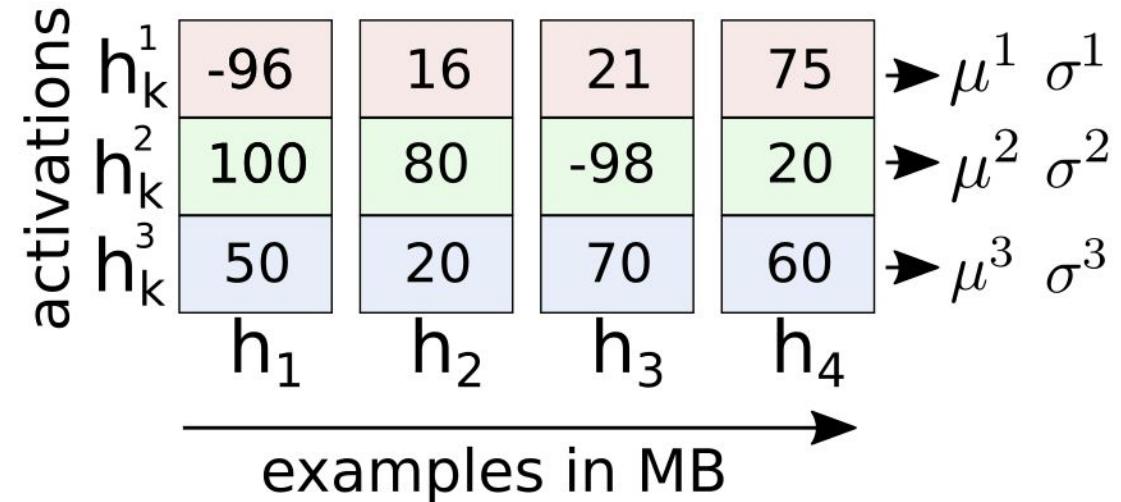
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., sioffe@google.com

Christian Szegedy
Google Inc., szegedy@google.com

In mini-batch we have many examples



Regularization - zoo



Normalization layers family - prevent overfitting and speed up training

- **Batch normalization (2015)** - introduces normalization operation in a differentiable manner

With batch normalization one may:

- increase learning rate
- in some cases remove Dropout
- reduce L2 regularization

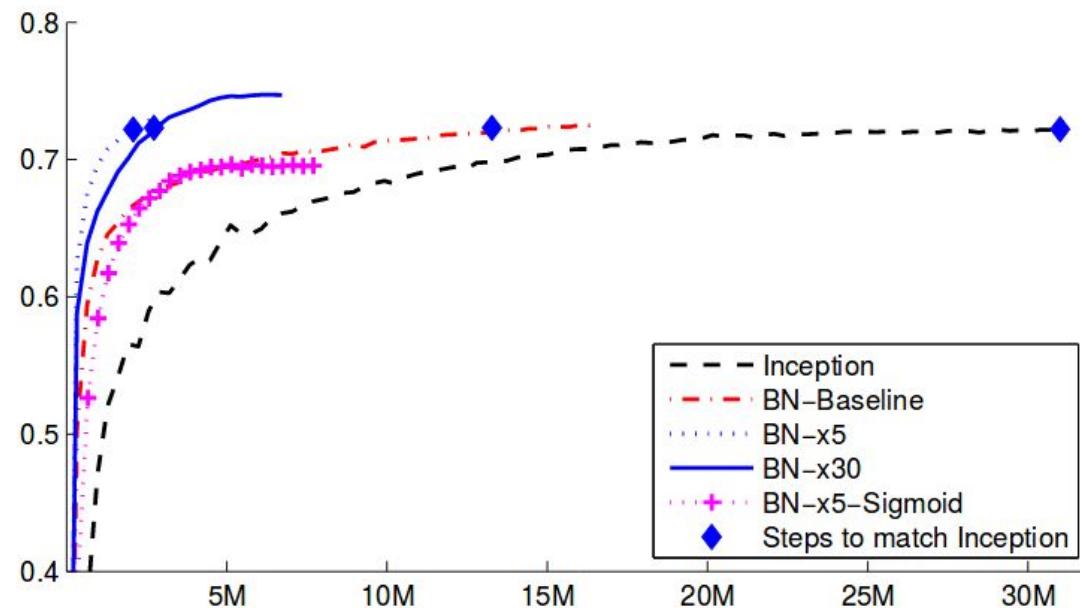


Figure 2: Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.



Regularization - zoo

Normalization layers family - prevent overfitting and speed up training

- **Batch normalization (2015)** - introduces normalization operation in a differentiable manner
- **Layer normalization (2016)** - normalize hidden activation per example in mini-batch ([paper](#))

Batch normalization has problems with RNNs, here the normalization depends only on current step hence it can be applied to normalize RNNs networks

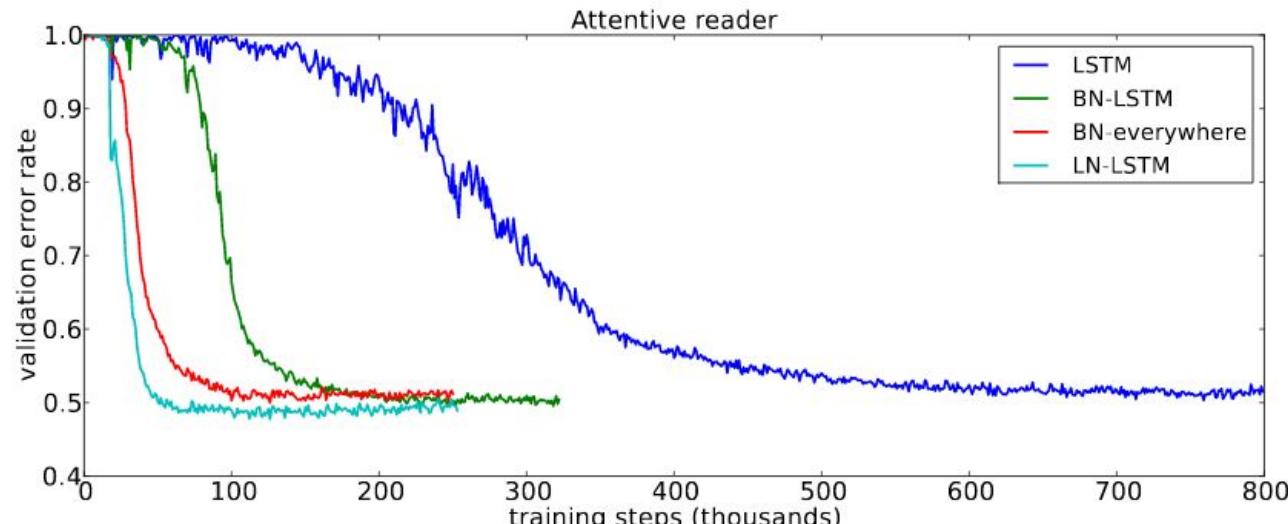


Figure 2: Validation curves for the attentive reader model. BN results are taken from [Cooijmans et al., 2016].

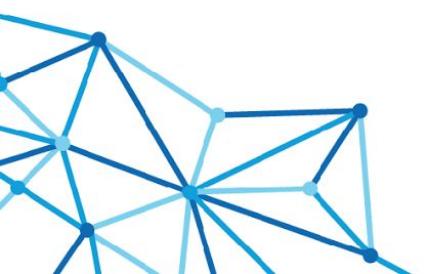
Regularization - zoo



Normalization layers family - prevent overfitting and speed up training

- **Batch normalization (2015)** - introduces normalization operation in a differentiable manner
- **Layer normalization (2016)** - normalize hidden activation per example in mini-batch ([paper](#))
- **Weight normalization (2016)** - normalize weight matrix instead of activations: replace \mathbf{w} with \mathbf{v} and g ([paper](#)), can be applied to RNNs

$$\mathbf{w} = \frac{g}{\|\mathbf{v}\|} \mathbf{v}$$



Regularization - zoo



Normalization layers family - prevent overfitting and speed up training

- **Batch normalization (2015)** - introduces normalization operation in a differentiable manner
- **Layer normalization (2016)** - normalize hidden activation per example in mini-batch ([paper](#))
- **Weight normalization (2016)** - normalize weight matrix instead of activations: replace w with v and g ([paper](#)), can be applied to RNNs
- **Batch renormalization (Feb 2017!)** - quite promising, extension to the original BN paper. Its done by introducing additional learnable parameters which fix the difference between train and test inference, learning is quite tricky and demands hand tuning.



Regularization - zoo

Normalization layers family - prevent overfitting and speed up training

- **Batch normalization (2015)** - introduces normalization operation in a differentiable manner
- **Layer normalization (2016)** - normalize hidden activation per example in mini-batch ([paper](#))
- **Weight normalization (2016)** - normalize weight matrix instead of activations: replace w with v and g ([paper](#)), can be applied to RNNs
- **Batch renormalization (Feb 2017!)** - quite promising, extension to the original BN paper. Its done by introducing additional learnable parameters which fix the difference between train and test inference, learning is quite tricky and demands hand tuning ([paper](#))
- **Orthogonalization regularization (Feb 2017!)** - make your matrix to be an orthogonal one! Cool - it preserves the norm and makes gradients more stable ([Neural Photo Editing with Introspective Adversarial Networks](#))

$$\mathcal{L}_{ortho} = \Sigma(|WW^T - I|)$$

Regularization - zoo

Normalization layers family - prevent overfitting and speed up training

- **Batch normalization (2015)** - introduces normalization operation in a differentiable manner
- **Layer normalization (2016)** - normalize hidden activation per example in mini-batch ([paper](#))
- **Weight normalization (2016)** - normalize weight matrix instead of activations: replace w with v and g ([paper](#)), can be applied to RNNs
- **Batch renormalization (Feb 2017!)** - quite promising, extension to the original BN paper. Its done by introducing additional learnable parameters which fix the difference between train and test inference, learning is quite tricky and demands hand tuning ([paper](#))
- **Orthogonalization regularization (Feb 2017!)** - make your matrix to be an orthogonal one! Cool - it preserves the norm and makes gradients more stable ([Neural Photo Editing with Introspective Adversarial Networks](#))
- **Cosine normalization (Feb 2017!)** - replace dot product with cosine similarity ([paper](#))

Regularization - zoo

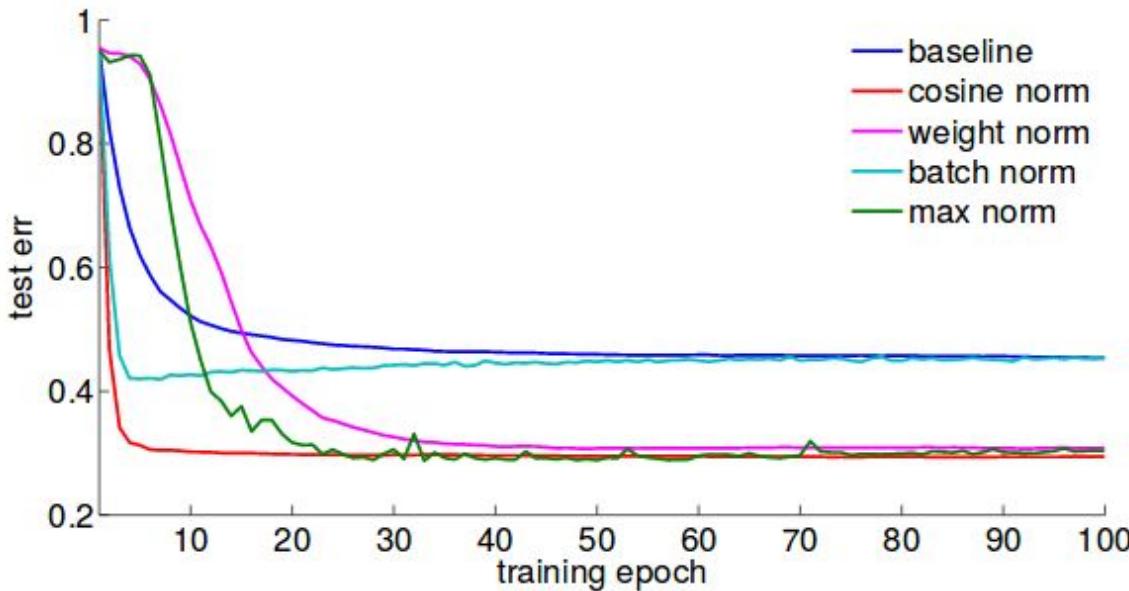
Normalization layers family - prevent overfitting and speed up training



- **Cosine normalization (Feb 2017!)** - replace dot product with cosine similarity

instead this: $\vec{w} \cdot \vec{x}$

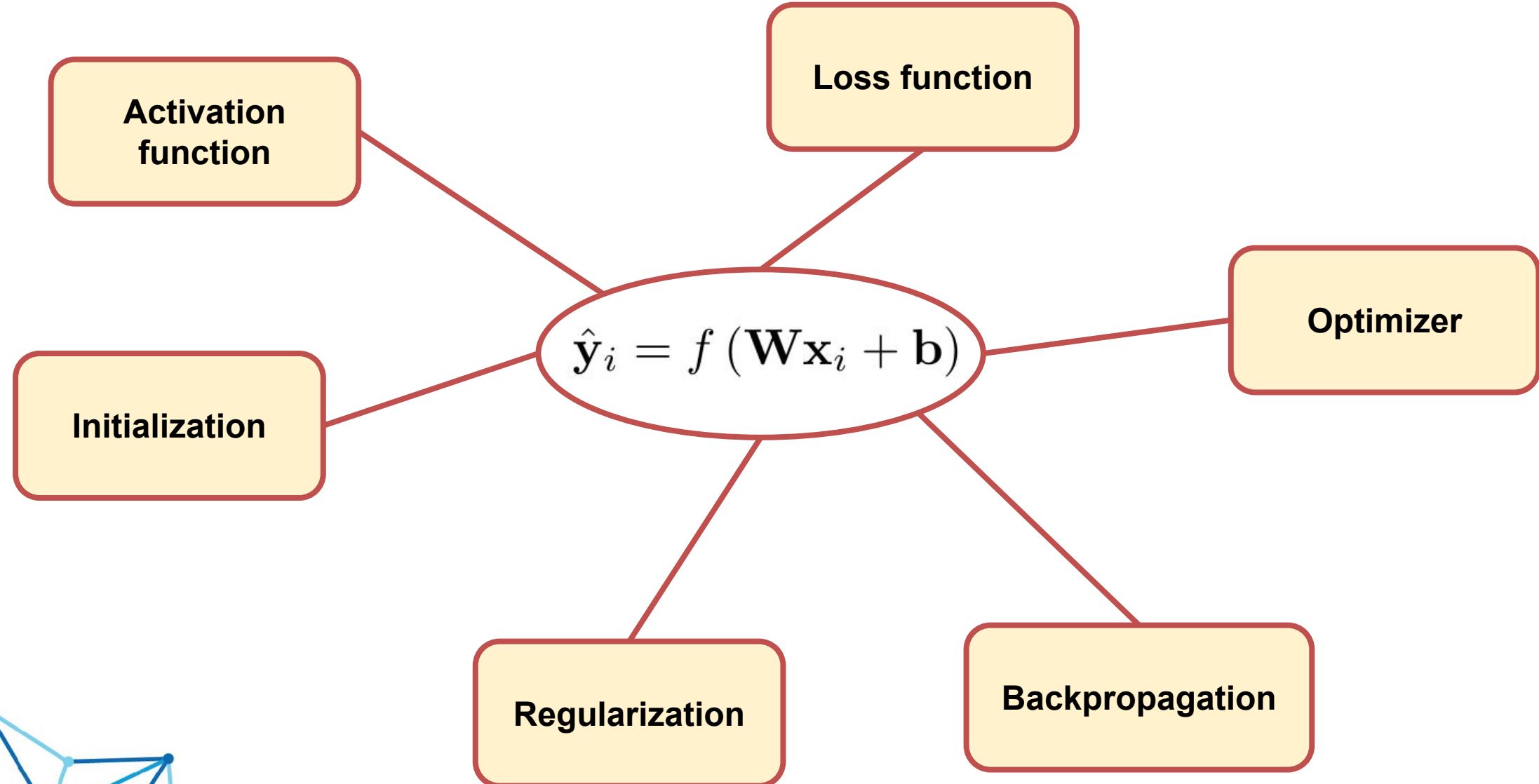
use this:
$$\frac{\vec{w} \cdot \vec{x}}{|\vec{w}| |\vec{x}|}$$



- easy to implement
- faster learning
- better accuracy
- can be applied to RNNs

Figure 6. The 20NEWS test err of networks trained with differ-

Neural Networks





Thank you for your attention

Krzysztof Kolasiński krzysztof.kolasinski@fornax.ai

Appendix

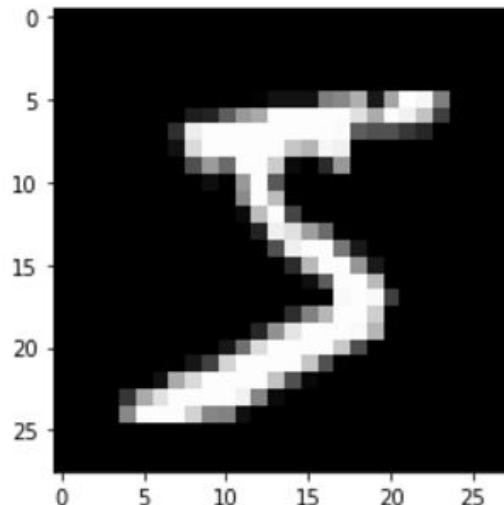


FORNAX

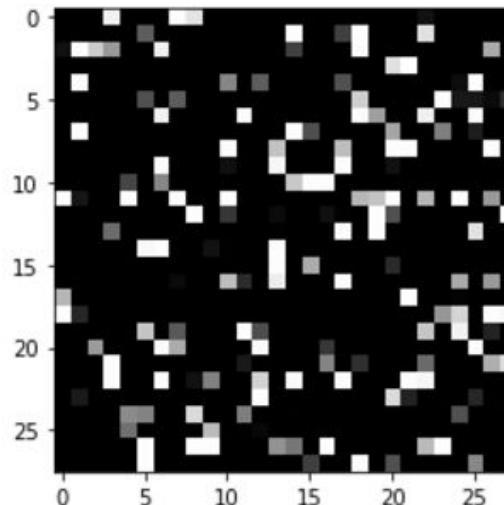
WWW.FORNAX.CO

Epilogue: The neural nets “paradox”

Given the input image:



We permute every pixel:



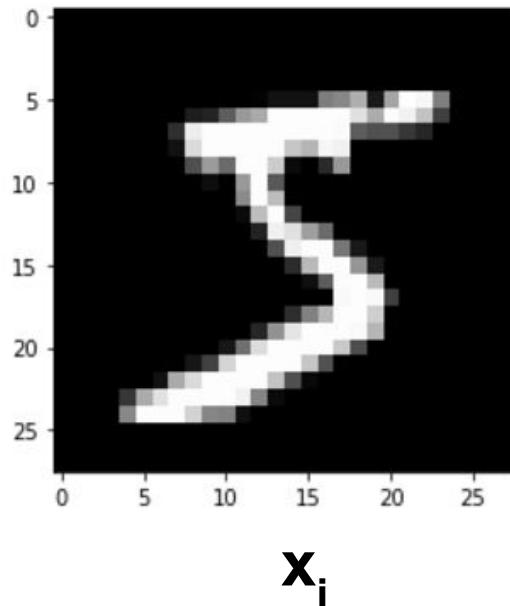
Then apply the same permutation for every image in dataset and train Neural Network

What NN network will learn? Will it perform better or worse ?

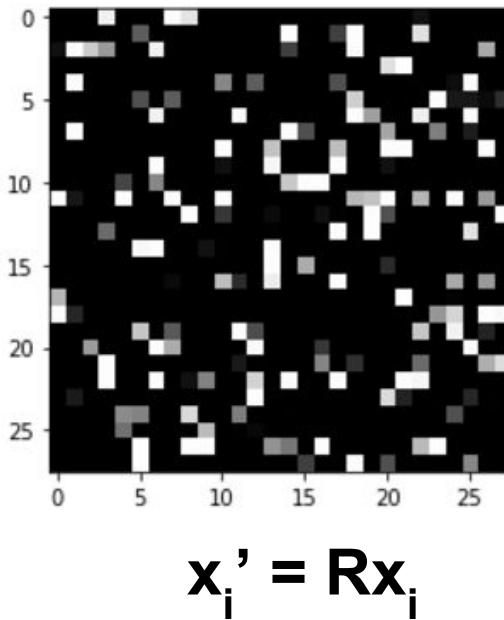
It will learn exactly the same (in terms of accuracy)!

Epilogue: The neural nets “paradox”

Given the input image:



We permute every pixel:



Explanation:

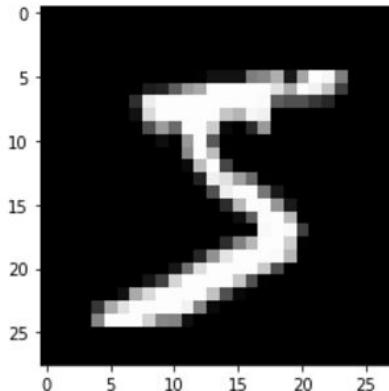
- we notice that permutation is a rotation operation
$$\mathbf{x}'_i = \mathbf{R}\mathbf{x}_i \quad \mathbf{R}^T\mathbf{R} = 1$$
- we work with single layer NN (before permutation)
$$\hat{\mathbf{y}}_i = f(\mathbf{W}\mathbf{x}_i + \mathbf{b})$$
- and after permutation

$$\hat{\mathbf{y}}'_i = f(\mathbf{W}'\mathbf{x}'_i + \mathbf{b}')$$

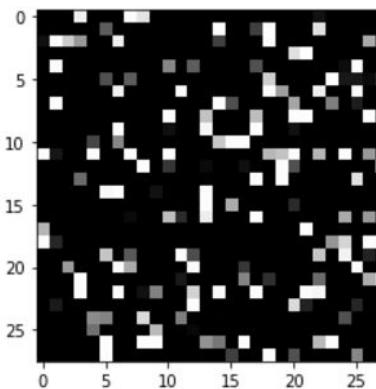
We assume (with primes) that our model will learn different parameters

Epilogue: The neural nets “paradox”

Given the input image: \mathbf{x}_i



$$\mathbf{x}'_i = \mathbf{R}\mathbf{x}_i$$



$$\mathbf{x}'_i = \mathbf{R}\mathbf{x}_i$$

- we don't permute labels hence our predictions must be the same
- we ask then if this is possible

$$\hat{\mathbf{y}}'_i = \hat{\mathbf{y}}_i$$

- we get condition to satisfy

$$f(\mathbf{W}'\mathbf{x}'_i + \mathbf{b}') = f(\mathbf{W}\mathbf{x}_i + \mathbf{b})$$

$$\mathbf{W}'\mathbf{x}'_i + \mathbf{b}' = \mathbf{W}\mathbf{x}_i + \mathbf{b} \quad + \text{const}$$

$$\mathbf{W}'\mathbf{R}\mathbf{x}_i + \mathbf{b}' = \mathbf{W}\mathbf{x}_i + \mathbf{b} \quad + \text{const}$$

- hence both sides are equal when

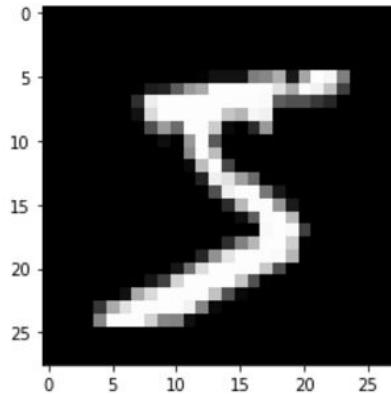
$$\mathbf{W} = \mathbf{W}'\mathbf{R} \Rightarrow \mathbf{W}' = \mathbf{W}\mathbf{R}^T$$

$$\mathbf{b} = \mathbf{b}' - \text{const}$$

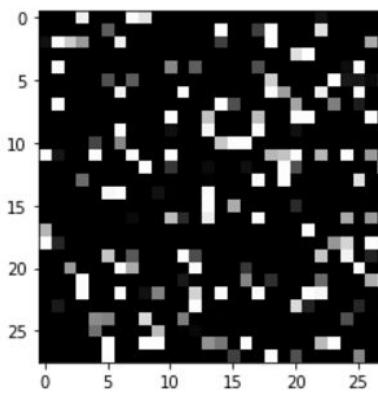
softmax

Epilogue: The neural nets “paradox”

Given the input image: \mathbf{x}_i



$$\mathbf{x}'_i = \mathbf{R}\mathbf{x}_i$$

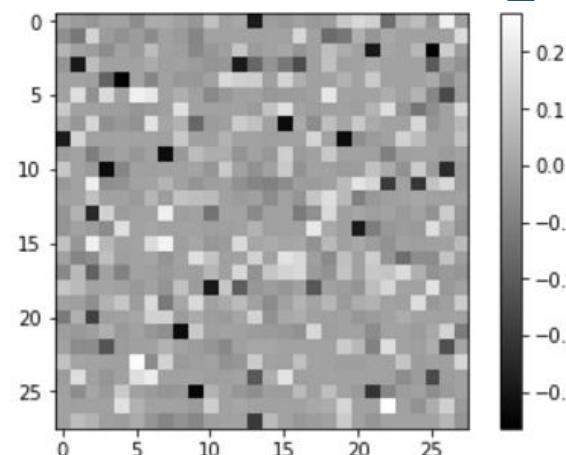


Let's check it in Keras: $\mathbf{W} = \mathbf{W}'\mathbf{R} \Rightarrow \mathbf{W}' = \mathbf{W}\mathbf{R}^T$

$$\mathbf{W} = \mathbf{W}'\mathbf{R} \Rightarrow \mathbf{W}' = \mathbf{W}\mathbf{R}^T$$

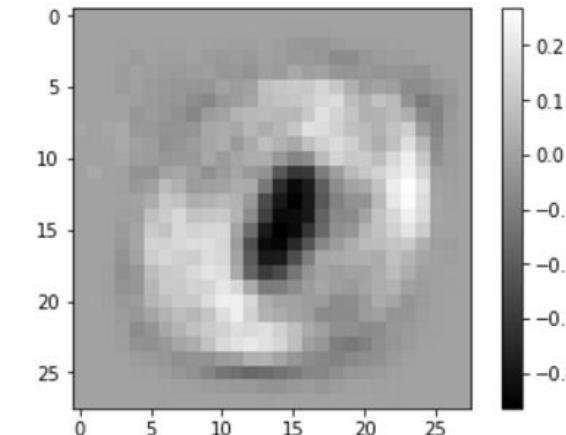
Learned weights do not correspond to anything intuitive (for humans)

$$\mathbf{W}'$$



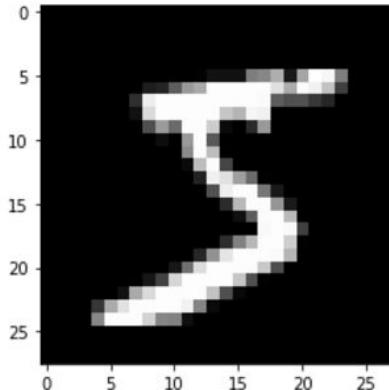
After rotation!

$$\mathbf{W} = \mathbf{W}'\mathbf{R}$$

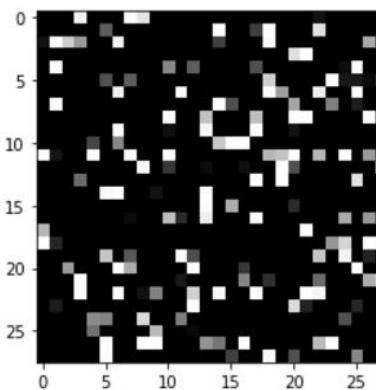


Epilogue: The neural nets “paradox”

Given the input image: \mathbf{x}_i



$$\mathbf{x}'_i = \mathbf{R}\mathbf{x}_i$$



Ok, but what we can learn from that?

- NNs do not capture spatial correlations in data
- NNs are not translational and rotational invariant, small shift in the image is enough to get wrong predictions
- The same with scaling

Ok, how we can solve those problems?

- For images use CNNs! Next talk :)

References

- <http://uvadlc.github.io/lectures/lecture1.pdf>
- <https://chaosmail.github.io/deeplearning/2016/10/22/intro-to-deep-learning-for-computer-vision/>
- <http://karpathy.github.io/2016/05/31/rl/>
- <https://culurciello.github.io/tech/2016/06/04/nets.html>
- <http://www.asimovinstitute.org/neural-network-zoo/>
- <https://blog.acolyer.org/2017/03/24/a-miscellany-of-fun-deep-learning-papers/> importance of loss function
- <http://cs231n.github.io/linear-classify/>
- <http://sebastianruder.com/optimizing-gradient-descent/>

Why do we need DL?

1. DL is fun!

Input



Reference



Output



[Deep Photo Style Transfer](#) paper from 22 March 2017 !

Project page: <https://github.com/luanjunf/Deep-Photo-StyleTransfer>

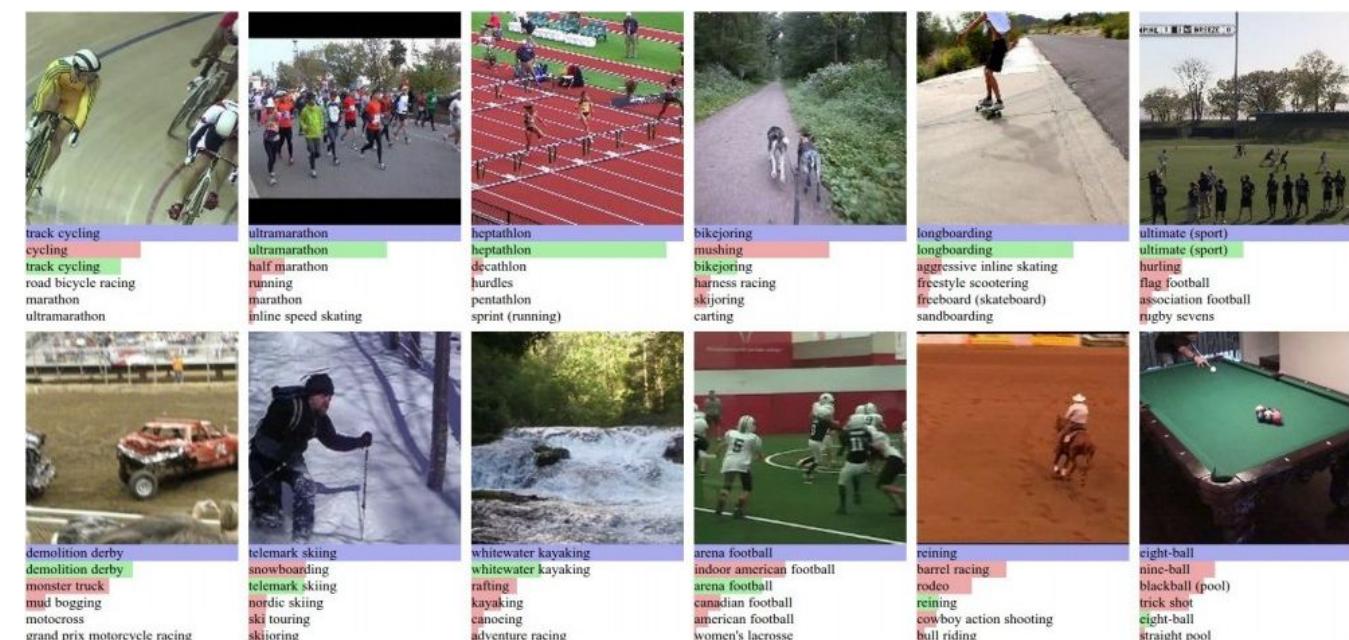
Learning process



Learning is a mathematical process of tuning our DL architecture to achieve some desired target

We can distinguish few types of learning strategies:

- **supervised learning** (today's workshop!) - neural networks, CNNs
- unsupervised learning - Autoencoders, RBMs
- semi-supervised learning - mix of above
- reinforcement learning - agents must act to maximize the rewards



Regularization - zoo

Dropout - 2014

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava
Geoffrey Hinton
Alex Krizhevsky
Ilya Sutskever
Ruslan Salakhutdinov

NITISH@CS.TORONTO.EDU
HINTON@CS.TORONTO.EDU
KRIZ@CS.TORONTO.EDU
ILYA@CS.TORONTO.EDU
RSALAKHU@CS.TORONTO.EDU

Method	Test Classification error %
L2	1.62
L2 + L1 applied towards the end of training	1.60
L2 + KL-sparsity	1.55
Max-norm	1.35
Dropout + L2	1.25
Dropout + Max-norm	1.05

Table 9: Comparison of different regularization methods on MNIST.

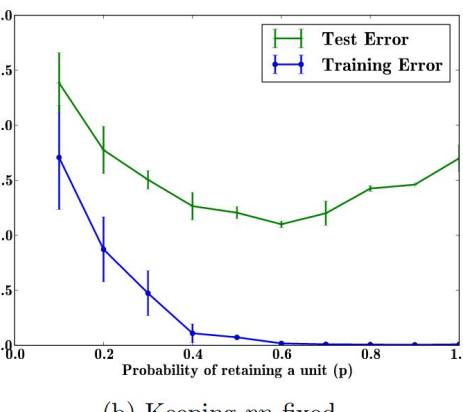
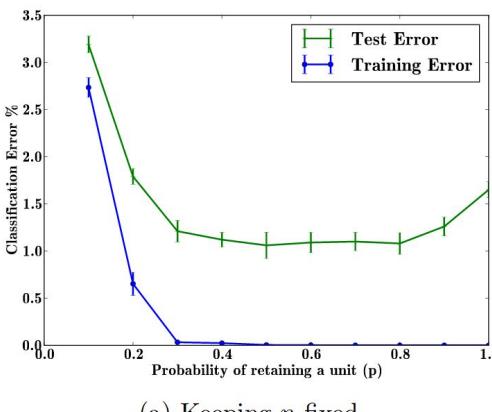


Figure 9: Effect of changing dropout rates on MNIST

Standard Neuron:

$$\begin{aligned} z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}), \end{aligned}$$

Standard Neuron with dropout:

$$\begin{aligned} r_j^{(l)} &\sim \text{Bernoulli}(p), \\ \tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\ z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{y}}^l + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}). \end{aligned}$$

binary mask
created with
probability p

Why do we need DL?

2. DL is important

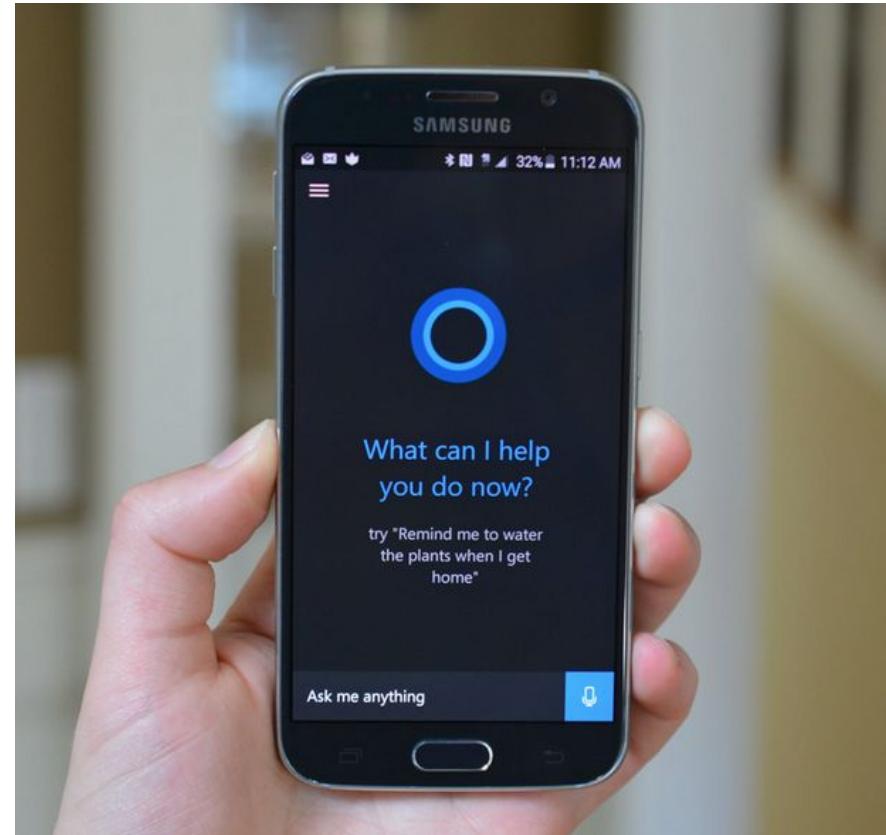
LETTER

doi:10.1038/nature14236

Human-level control through deep reinforcement learning

Volodymyr Mnih^{1*}, Koray Kavukcuoglu^{1*}, David Silver^{1*}, Andrei A. Rusu¹, Joel Veness¹, Marc G. Bellemare¹, Alex Graves¹, Martin Riedmiller¹, Andreas K. Fidjeland¹, Georg Ostrovski¹, Stig Petersen¹, Charles Beattie¹, Amir Sadik¹, Ioannis Antonoglou¹, Helen King¹, Dharshan Kumaran¹, Daan Wierstra¹, Shane Legg¹ & Demis Hassabis¹

"DQN achieved human-level performance in almost half of the 50 games to which it was applied"



Natural Language Processing e.g. speech recognition task

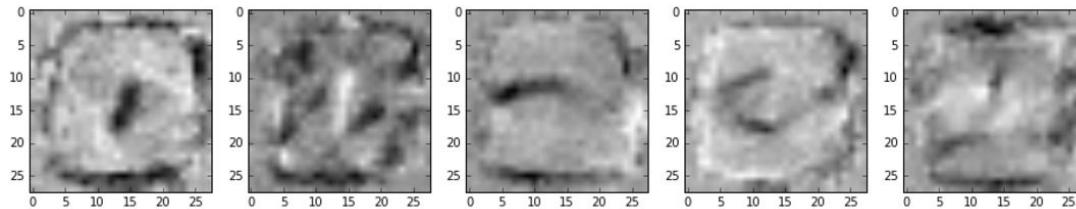
Regularization - zoo

Parameter norm penalties

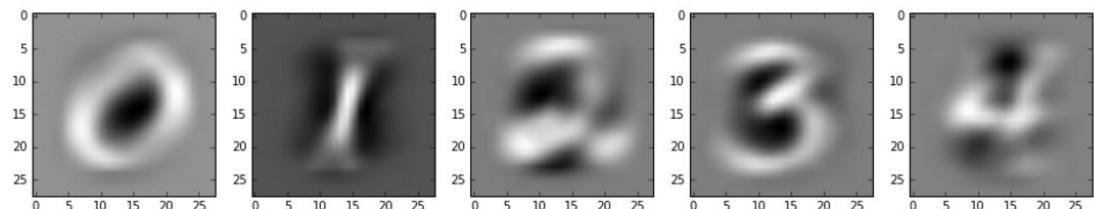
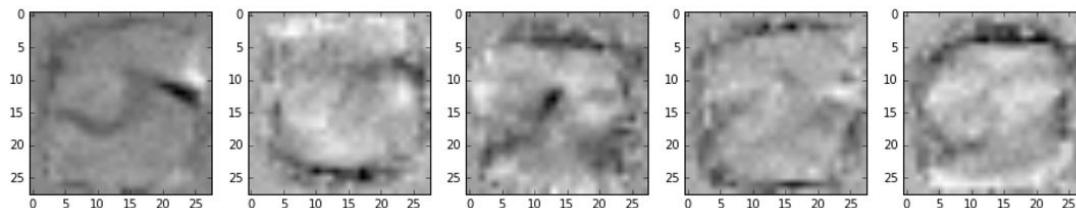
- **L2 regularization** $\mathcal{L}(\mathbf{W}; \lambda) = \frac{1}{2}\lambda \|\mathbf{W}\|_2^2$ with gradient

$$\frac{\partial \mathcal{L}(\mathbf{W}; \lambda)}{\partial \mathbf{W}} = \lambda \mathbf{W}$$

Regularization has smoothing effect on model parameters:



before regularization



after adding L2 regularization

[source: Sam's Research Blog](#)

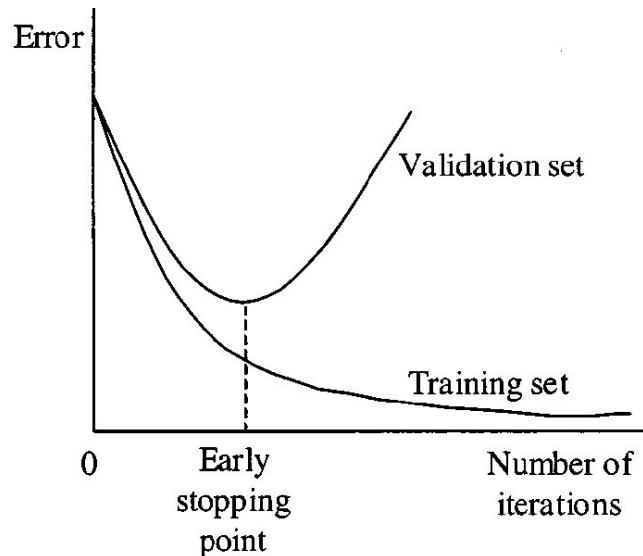
Regularization - zoo



Noise regularizers

- add noise with infinitesimal variance to input of the model
- add noise to hidden activations

Early stopping: *stop training before overfitting of model with large capacity*



Algorithm:

- Monitor your training and validation loss
- Stop training when the validation loss starts to increase

In keras we can use callbacks

EarlyStopping

[source]

```
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=0, verbose=0, mode='auto')
```

Stop training when a monitored quantity has stopped improving.

Regularization - zoo



Noise regularizers

- add noise with infinitesimal variance to input of the model
- add noise to hidden activations

Early stopping: *stop training before overfitting of model with large capacity*

Ensembling and bagging methods, etc ...

