Digital Image Processing

Tutorial 6

Andreas Ley andreas.ley@tu-berlin.de

Winter Semester 2019/2020

Berlin University of Technology (TUB),
Computer Vision and Remote Sensing Group
Berlin, Germany

Sometime in the 1990s-2000s

Convolutional Neural Networks

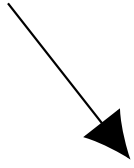
[Various]

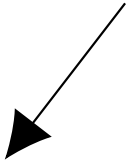
- Convolutions (Filters)
- Partly handcrafted
- Partly learned

Multi Layer Perceptron

[Hinton et.al.]

- Mat-Mul + Nonlinearity
- Backpropagation





"Modern" Convnets [LeCun et.al.]

- Convolutions (Filters)
- Backpropagation
- End-to-end training

LeNet (1998)

PROC. OF THE IEEE, NOVEMBER 1998 1

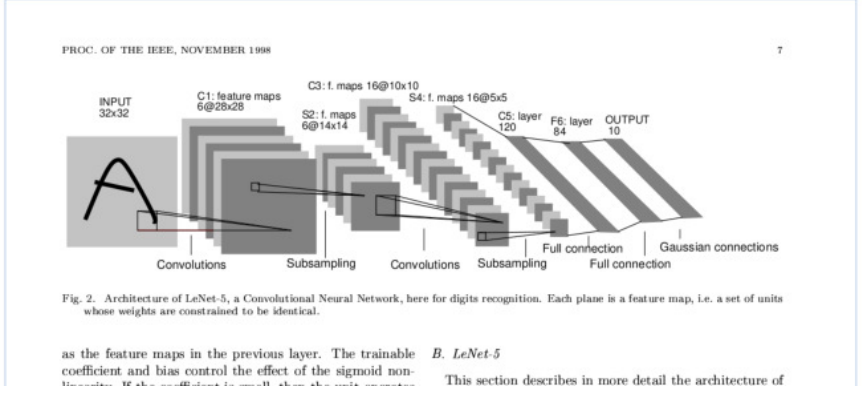
Gradient-Based Learning Applied to Document Recognition

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

Abstract—
Multilayer Neural Networks trained with the backpropagation algorithm constitute the best example of a successful Gradient-Based Learning technique. Given an appropriate network architecture, Gradient-Based Learning algorithms can be used to synthesize a complex decision surface that can classify high-dimensional patterns such as handwritten char-

I. Introduction

Over the last several years, machine learning techniques, particularly when applied to neural networks, have played an increasingly important role in the design of pattern recognition systems. In fact, it could be argued that the



PROC. OF THE IEEE, NOVEMBER 1998 7

Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

as the feature maps in the previous layer. The trainable coefficient and bias control the effect of the sigmoid non-

B. LeNet-5

This section describes in more detail the architecture of

Gradient-Based Learning Applied to Document Recognition [Lecun et al., 1998]

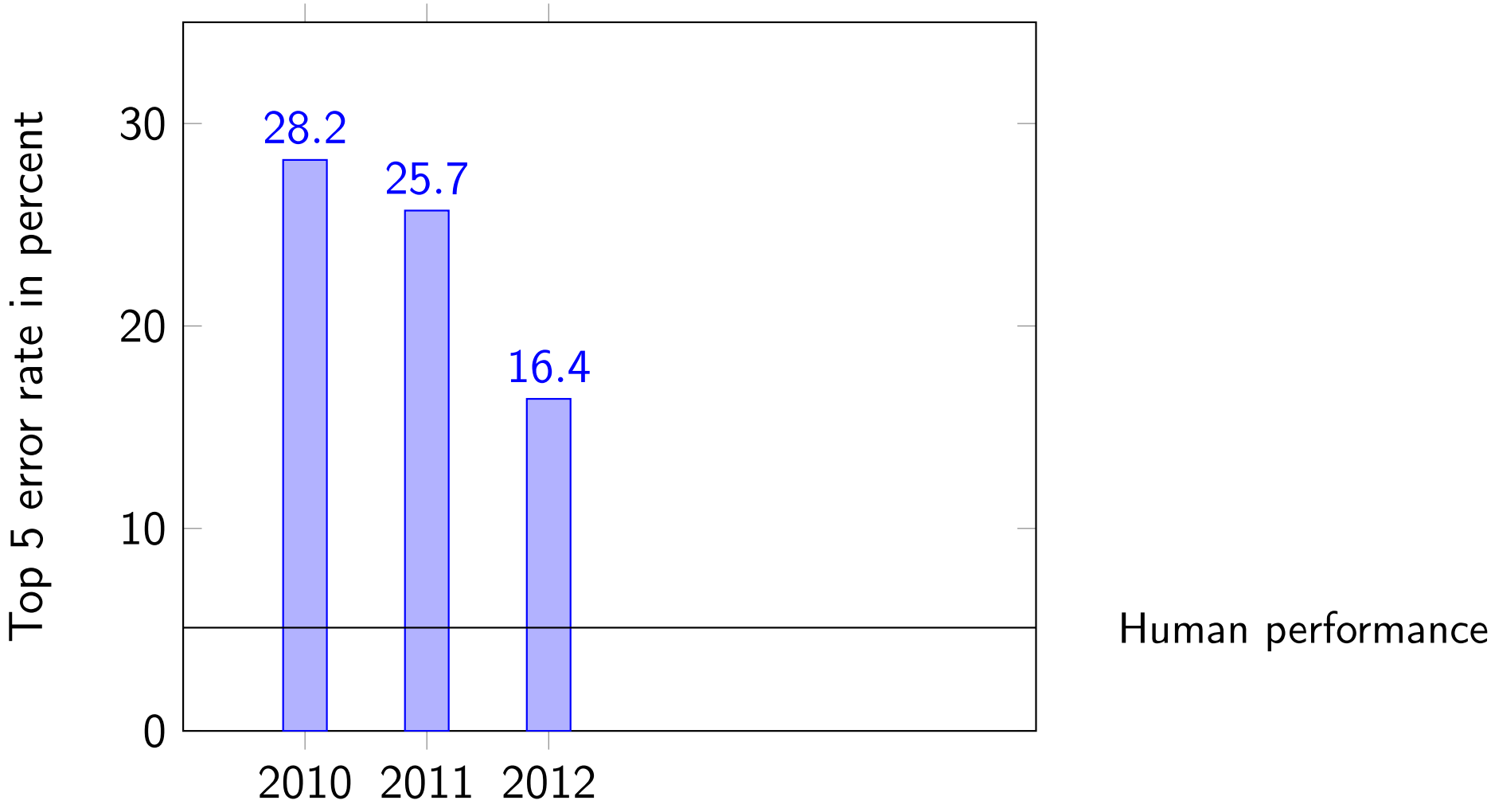
- Handwritten Digit Recognition (MNIST)
- Good performance
- But no guarantees
- Will training converge?
- Will it converge to optimum?
- "Alchemy"

ILSVRC (2010)

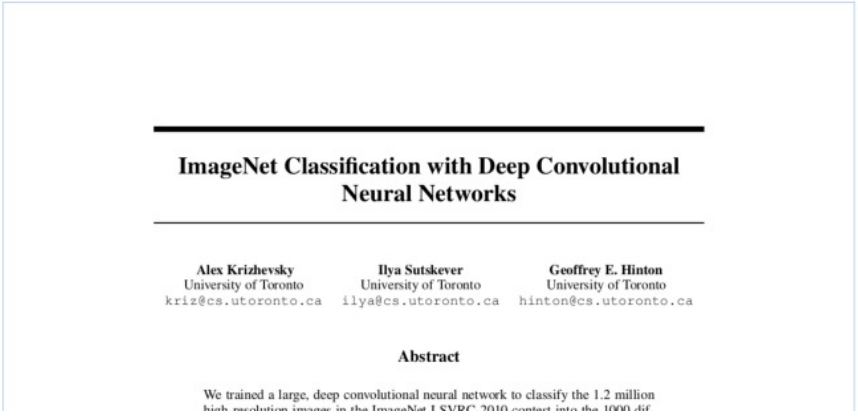
Large Scale Visual Recognition Challenge [Russakovsky et al., 2015]

- Held every year 2010-2017
- 1.2 million training images
- 1000 categories
- 150k test images with hidden labels

ILSVRC Results



AlexNet (2012)



ImageNet Classification with Deep Convolutional Neural Networks

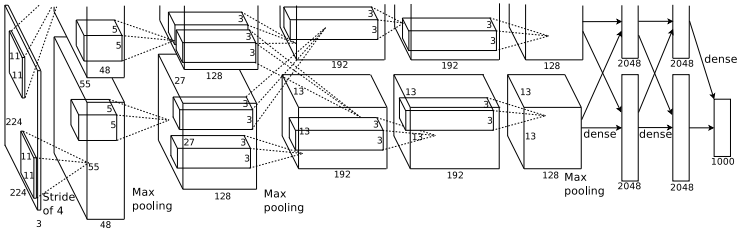
Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

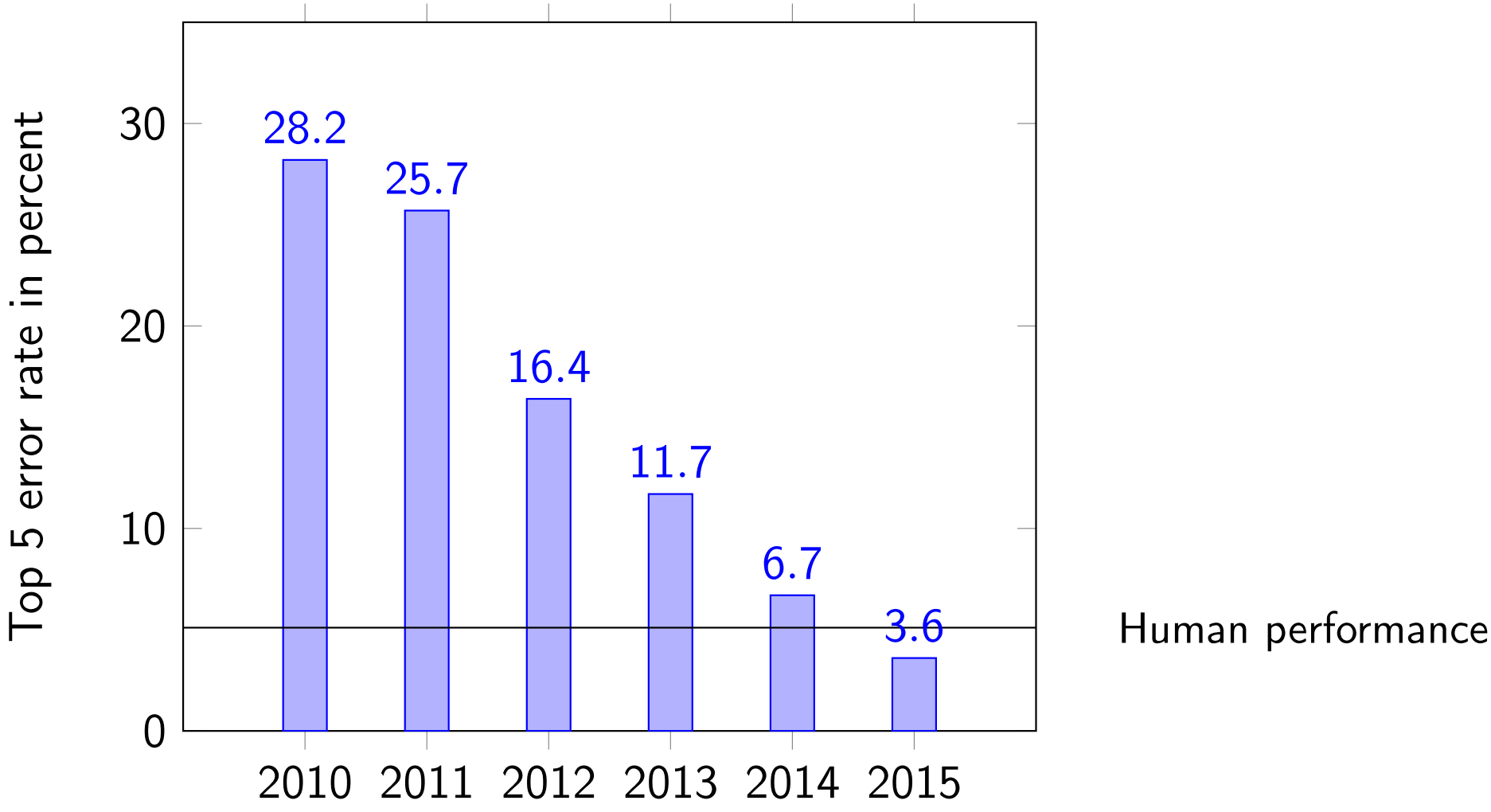
We trained a large, deep convolutional neural network to classify the 1.2 million



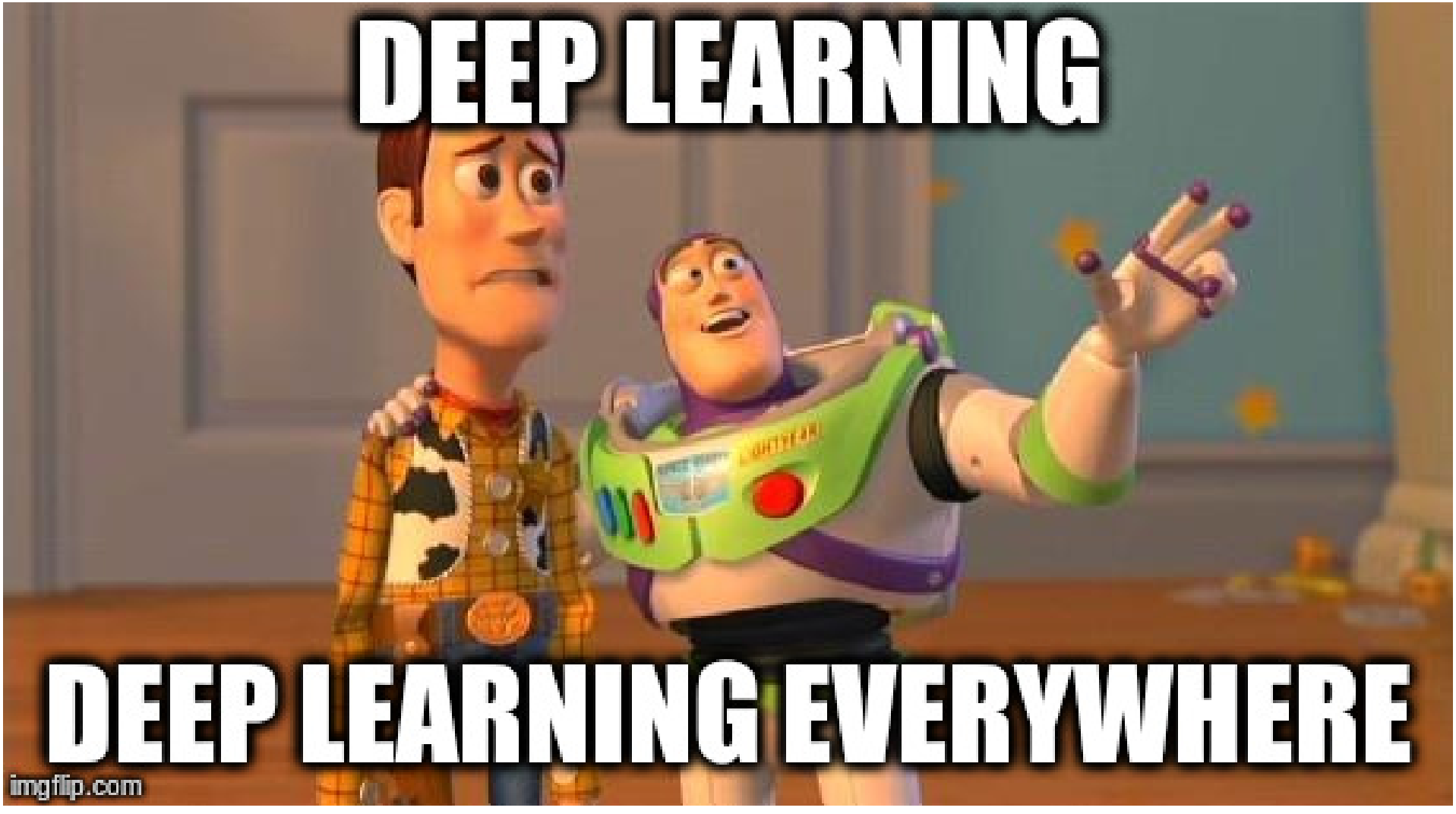
ImageNet Classification with Deep Convolutional Neural Networks [Krizhevsky et al., 2012]

- ILSVRC 2012 winner
- 16.4% vs. 26.2% (second place)
- ReLU + Dropout
- 8 layer convnet
- 2x GTX580
- 5-6 days (approx. 90 epochs)

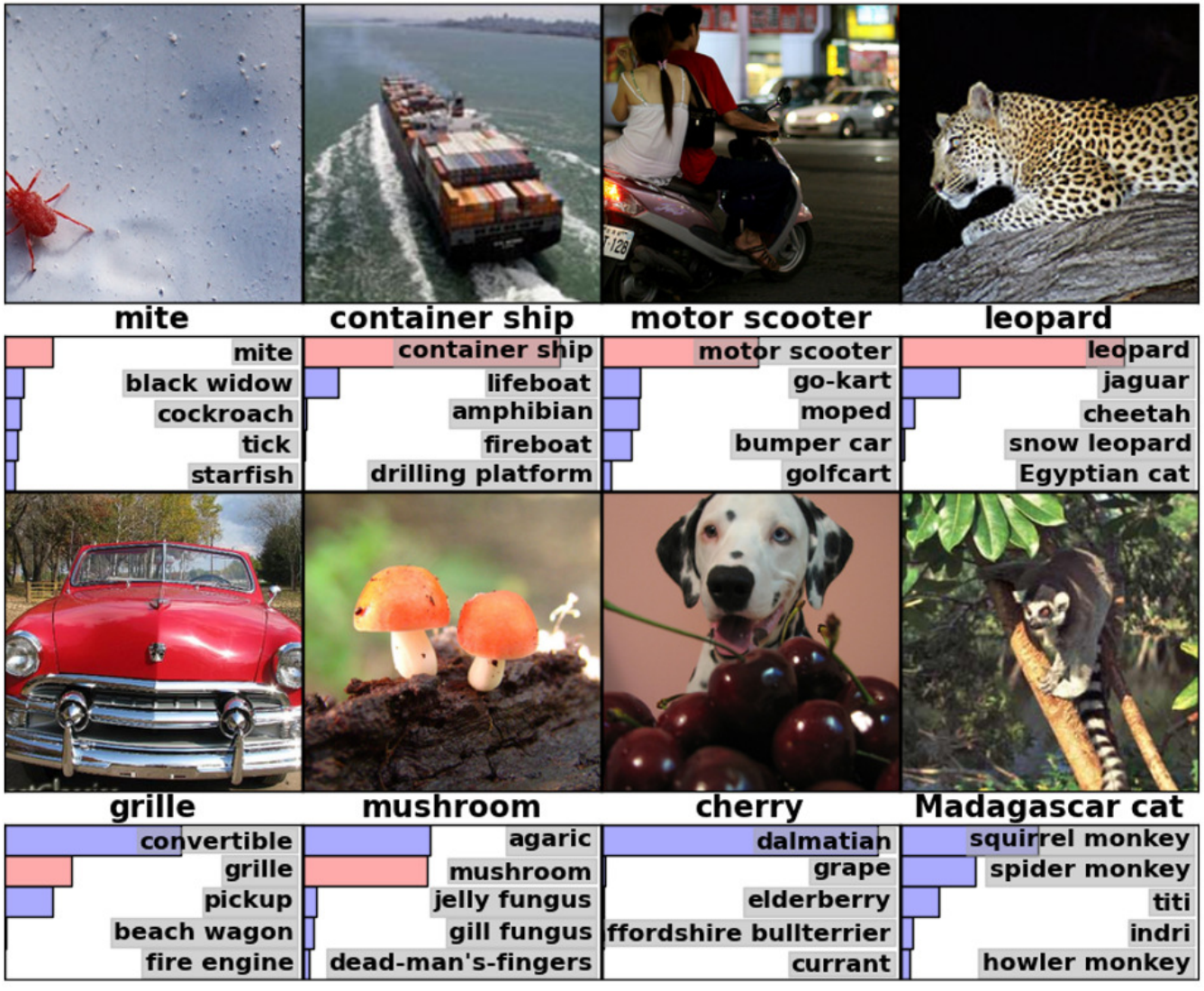
ILSVRC Results



Deep Learning is Everywhere

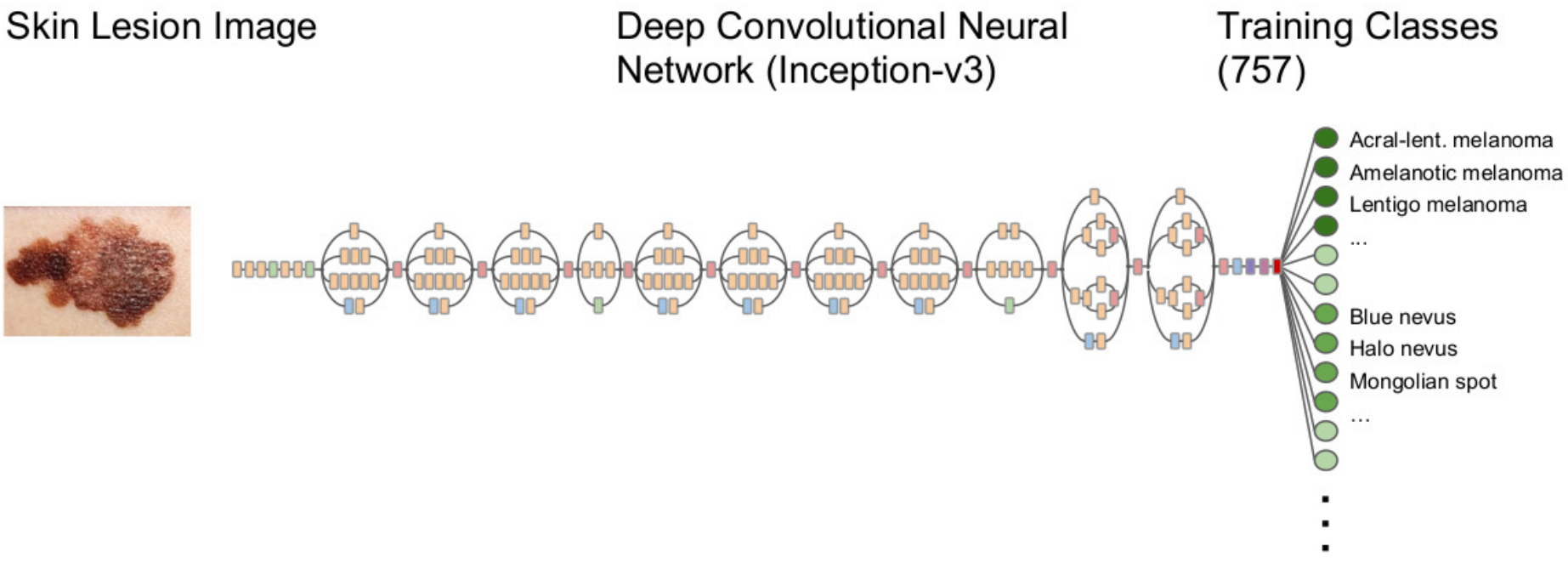


Classification: Image Net



(Krizhevsky et al., "ImageNet Classification with Deep Convolutional Neural

Classification: Skin Cancer



(Esteva et al., "Dermatologist-level classification of skin cancer with deep neural networks")

Classification: Autonomous Driving

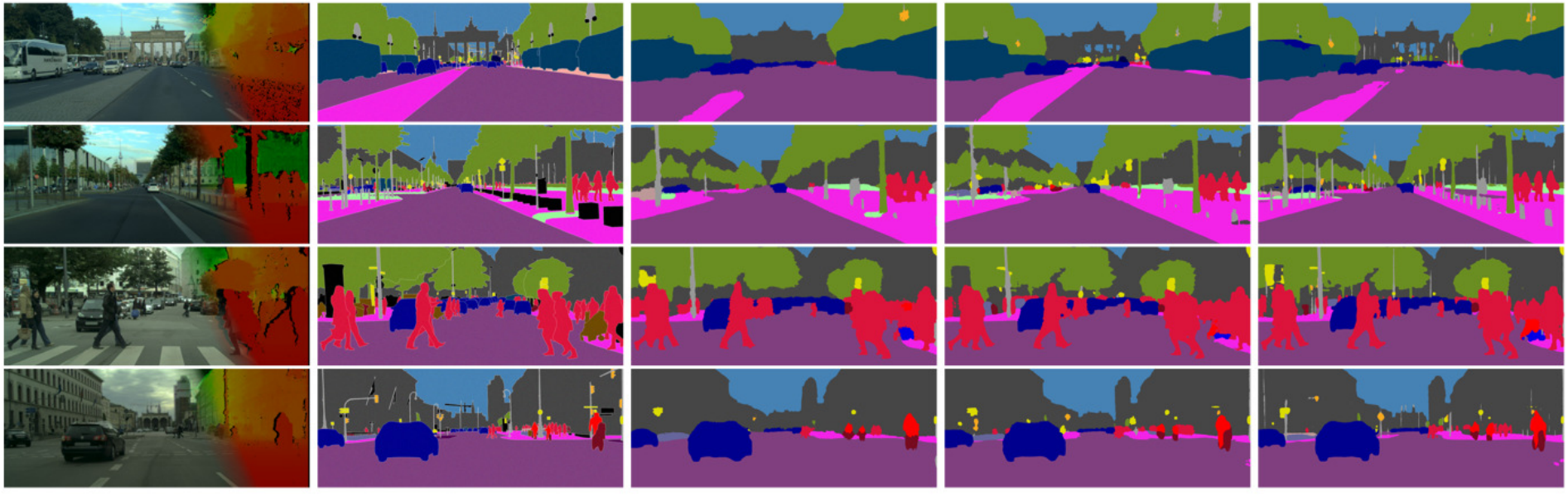


Figure 5. Qualitative examples of selected baselines. From left to right: image with stereo depth maps partially overlayed, annotation, DeepLab [48], Adelaide [37], and Dilated10 [79]. The color coding of the semantic classes matches Fig. 1.

(Cordts et al., "The Cityscapes Dataset for Semantic Urban Scene Understanding")

Advanced: GAN Faces

(Karras et al., "Progressive Growing of GANs for Improved Quality, Stability, and Variation")

Advanced: Image to Text

(Vinyals et al., "Show and Tell: A Neural Image Caption Generator")

Advanced: Text to Image

this small bird has a pink breast and crown, and black primaries and secondaries.

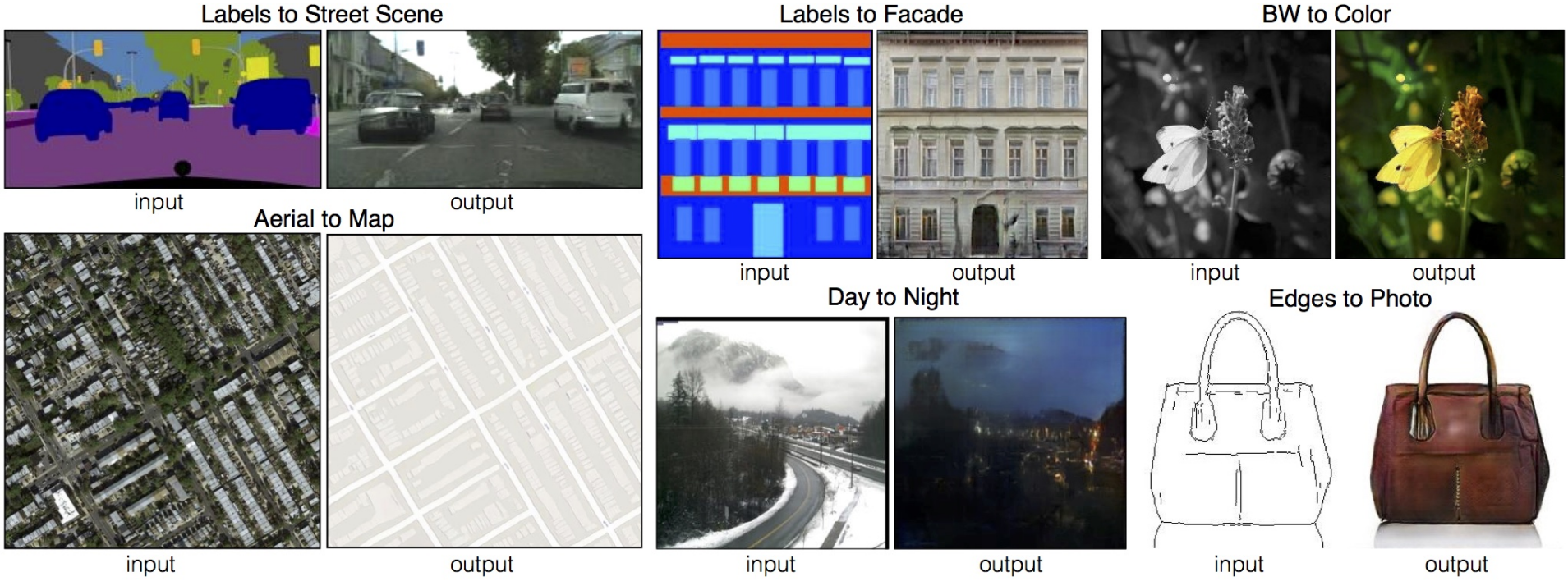
this magnificent fellow is almost all black with a red crest, and white cheek patch.

the flower has petals that are bright pinkish purple with white stigma

this white and yellow flower have thin white petals and a round yellow stamen

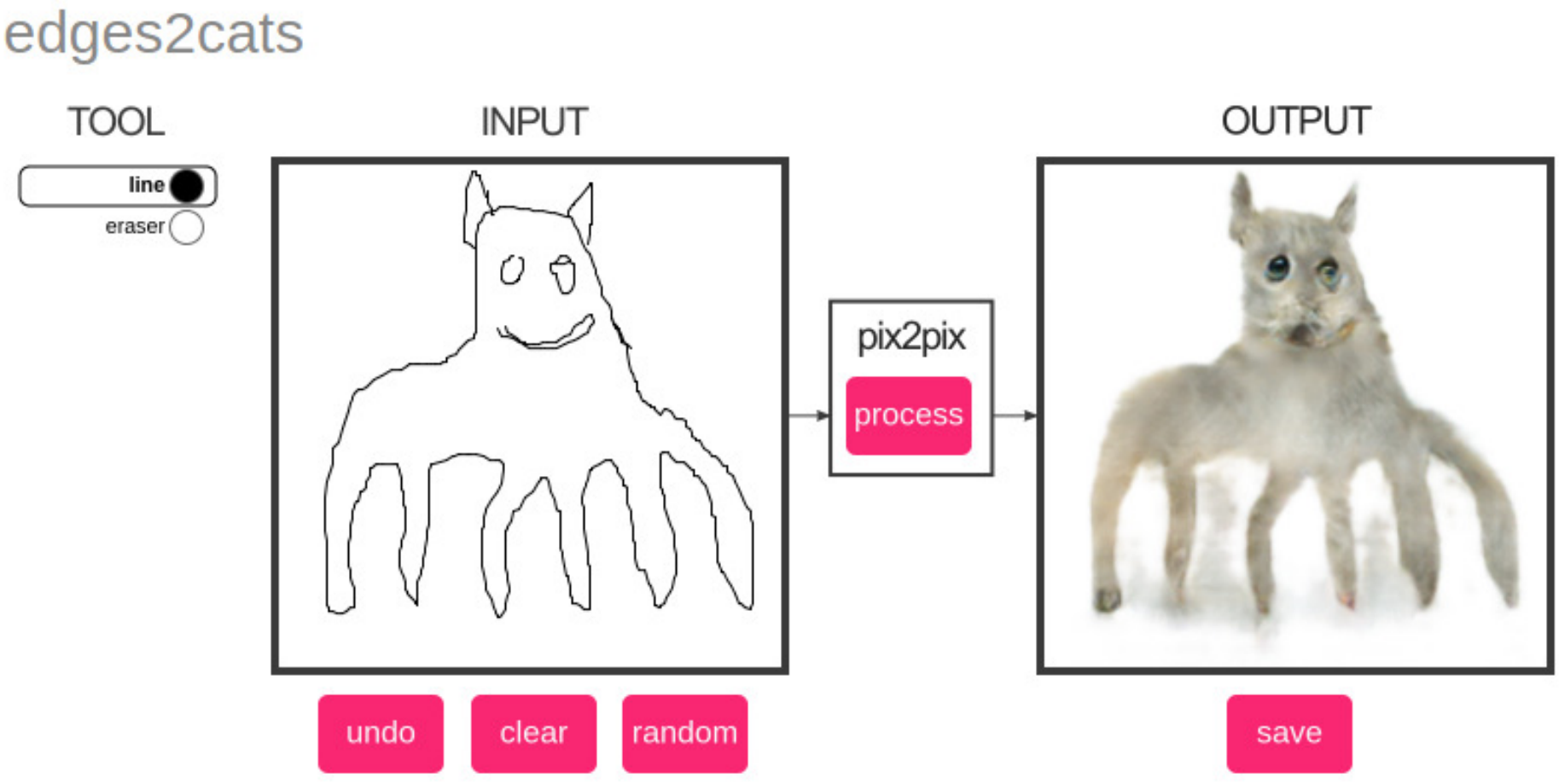
(Reed et al., "Generative Adversarial Text to Image Synthesis")

Advanced: Transcoding



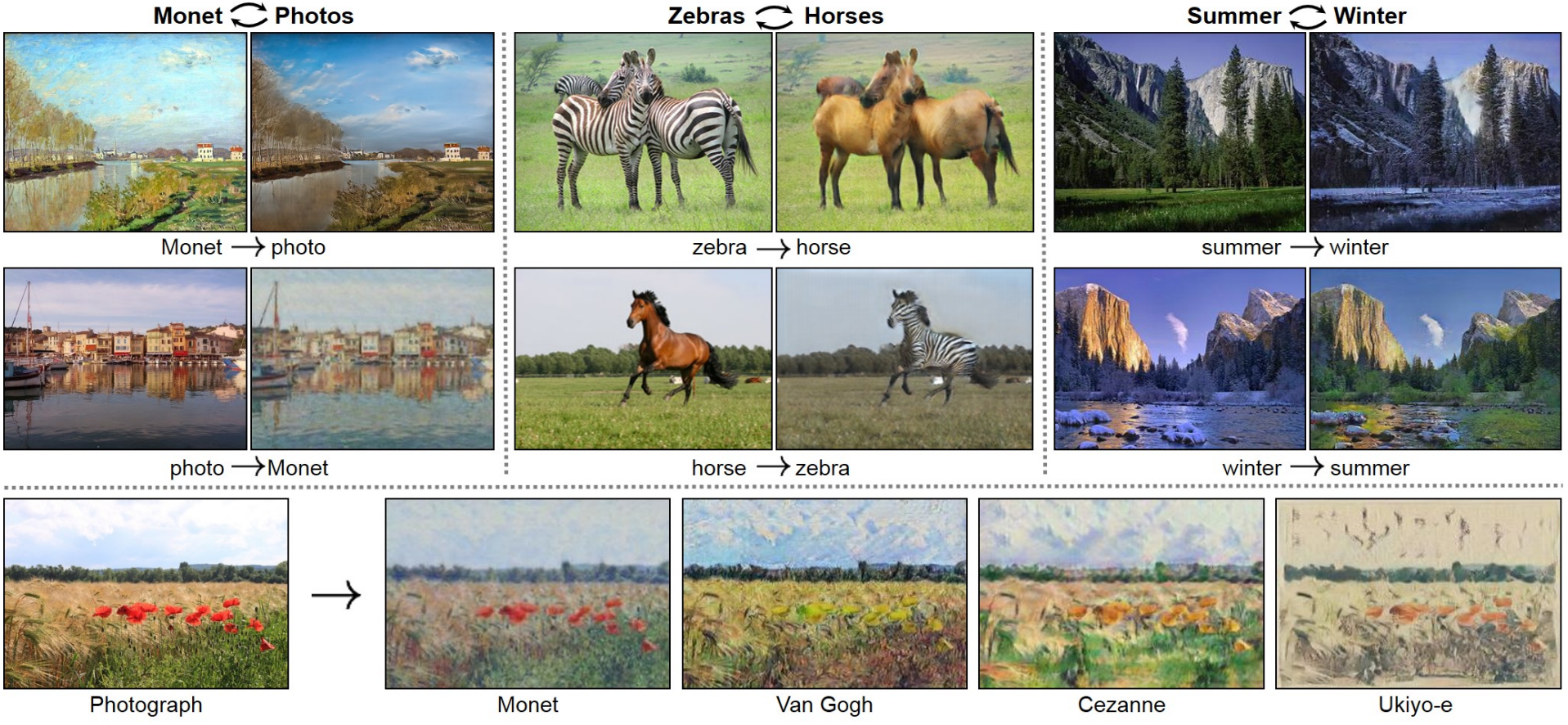
(Isola et al., "Image-to-Image Translation with Conditional Adversarial Nets")

Advanced: Transcoding



Try the web-demo at `https://affinelayer.com/pixsrv/`.
edges2cats is best!

Advanced: Transcoding



(Zhu et. al., "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks")

Advanced: Transcoding

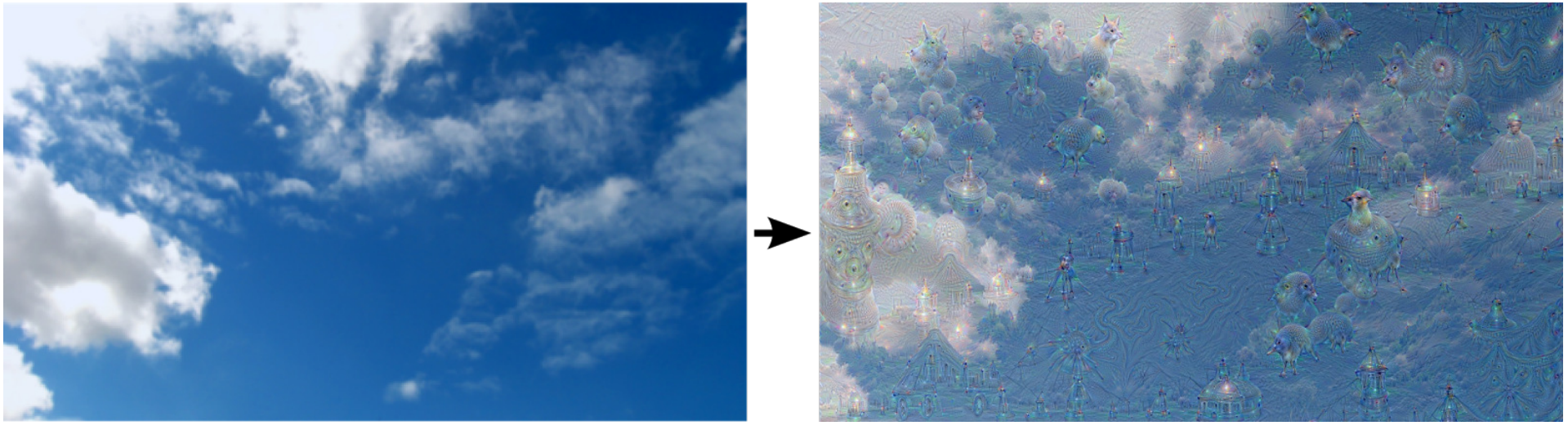
(some limitations remain)

Artsy & Fun: Style Transfer

(Gatys et. al., "Image Style Transfer Using Convolutional Neural Networks")

(Check out `http://deepart.io`)

Artsy & Fun: Deep Dream



(Mordvintsev et. al., "Inceptionism: Going Deeper into Neural Networks")

Artsy & Fun: Deep Dream

(Mordvintsev et. al., "Inceptionism: Going Deeper into Neural Networks")

Artsy & Fun: Deep Dream

Before:

Artsy & Fun: Deep Dream

After:

Deep Fake

Nicolas Cage Everywhere

Regulation?

The New York Times

US Restricts Exports of AI for Analyzing Satellite Images

By The Associated Press

Jan. 5, 2020

WASHINGTON — U.S. technology companies that build artificial intelligence software for analyzing satellite imagery will face new restrictions on exporting their products to China and elsewhere.

The Commerce Department said new export rules take effect Monday that target emerging technology that could give the U.S. a significant military or intelligence advantage. A special license would be required to sell software outside the U.S. that can automatically scan aerial images to identify objects of interest, such as vehicles or houses.

Regulation?

DEPARTMENT OF COMMERCE

Bureau of Industry and Security

15 CFR Part 774

[Docket No. 191217–0116]

RIN 0694–AH89

Addition of Software Specially Designed To Automate the Analysis of Geospatial Imagery to the Export Control Classification Number 0Y521 Series

AGENCY: Bureau of Industry and Security, Commerce.

ACTION: Interim final rule with request for comments.

SUMMARY: In this interim final rule, the Bureau of Industry and Security (BIS) amends the Export Administration Regulations (EAR) to make certain items subject to the EAR and to impose a license requirement for the export and reexport of those items to all destinations, except Canada. Specifically, this rule classifies software specially designed to automate the analysis of geospatial imagery, as specified, under the Export Control Classification Number (ECCN) 0Y521 series, specifically under ECCN 0D521. BIS adds this item to the 0Y521 series of ECCNs upon a determination by the Department of Commerce, with the concurrence of the Departments of Defense and State, and other agencies as appropriate, that the items warrant control for export because the items may provide a significant military or intelligence advantage to the United States or because foreign policy reasons justify control, pursuant to the ECCN 0Y521 series procedures.

DATES: This rule is effective January 6, 2020. Comments must be received by March 6, 2020.

"[...] items may provide significant military or intelligence advantage to the United States [...]"

Why are we looking at this?

- This is Machine Learning
- Fits Automatic Image Analysis (summer term) at best
- Inner workings is low level image operations
- Mostly convolutions

Essentially, this is DIP on steroids...

Exercise 6
Super resolution

3x super resolved →

Exercise 6
Super resolution

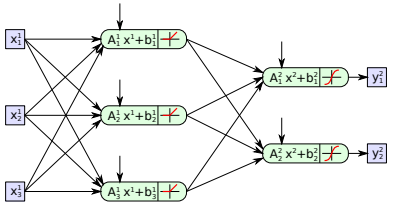
Train a ConvNet that upsamples an image 3x.

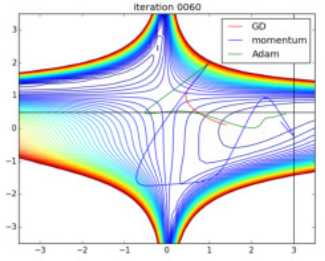
- Small, simple architecture (we don't have GPUs)
- Train on pairs of low and high resolution image patches
- Patches taken from DIV2K Dataset [Agustsson and Timofte, 2017]
- Patches are desired output of network. Downsampled patches are input.
- Downsampling done on the fly (in a crappy way)

The Plan for Today

1. Intro
2. Theory
3. ConvNets
4. Performance
5. Homework

Theory

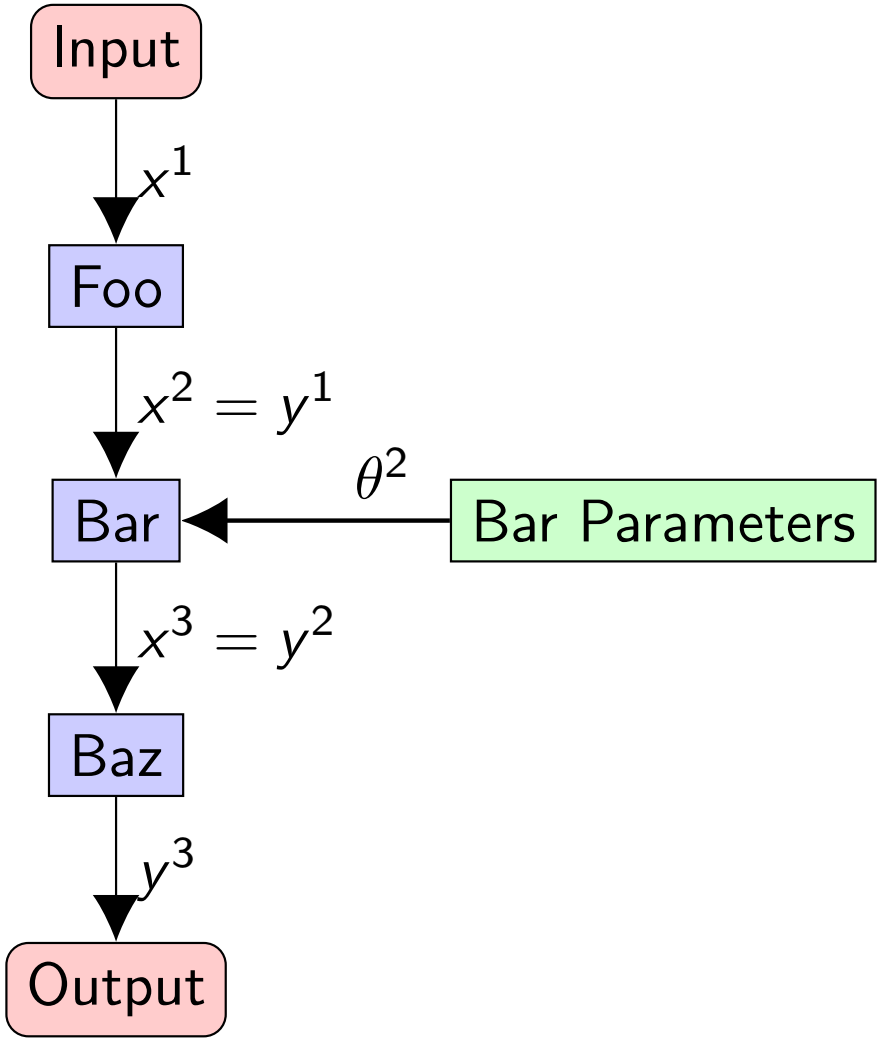




Graph of Processing Nodes

Feed-Forward Network

- Network of *layers*
- Data flows through layers
- Feed-Forward means no cycles
- Layers transform data in certain ways
- Some layers have learned parameters
- Data usually vectors or tensors



Complexity from Simple Mappings

Core idea

- Concatenate multiple simple mappings to get one powerful mapping
- Multiple simple steps more powerful than one complex step
- Keep everything (mostly) differentiable
- Train by doing gradient descend on classification error

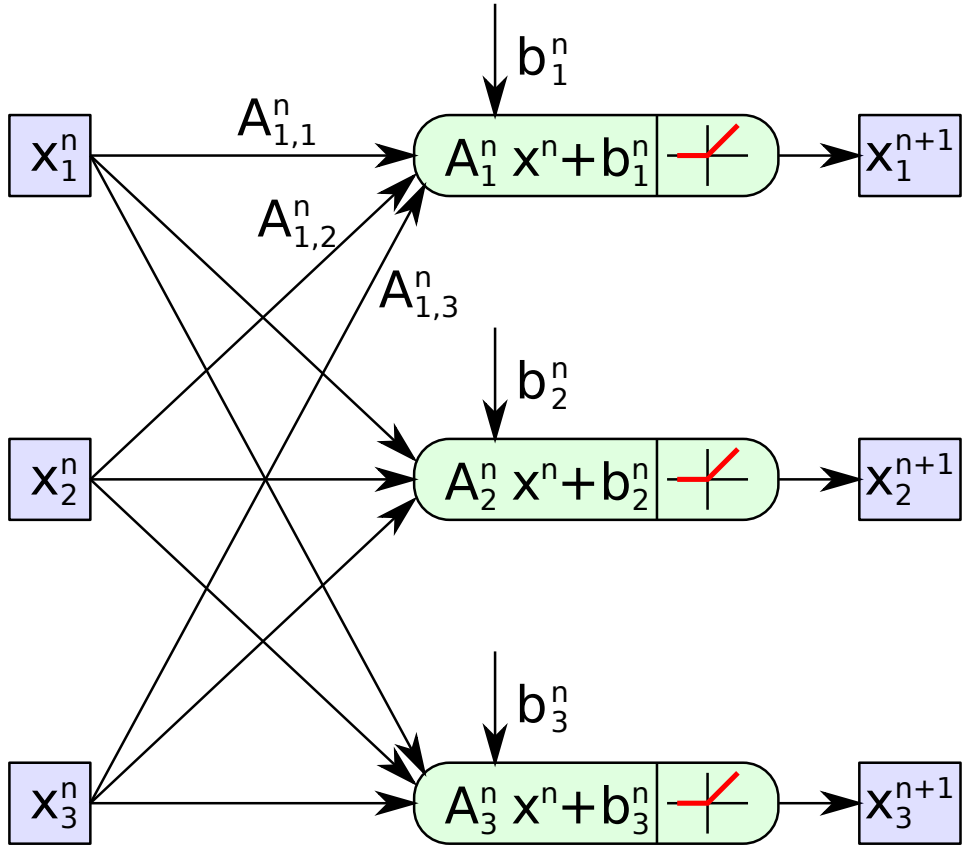
Lots of nice properties:

- Encapsulation if functionality into layers
- Flexibility to exchange layers, introduce new ones
- End-2-end training to fit everything to specific use case

Fully Connected Layer

$$\mathbf{x}^{n+1} = \mathbf{y}^n = f(\mathbf{A}^n \cdot \mathbf{x}^n + \mathbf{b}^n) \quad (1)$$

- $\mathbf{x}^n$: Layer input
- $\mathbf{y}^n = \mathbf{x}^{n+1}$: Layer output
- $\mathbf{A}^n$: Weights
- $\mathbf{b}^n$: Bias
- $\boldsymbol{\theta}^n$: Weights and Biases
- $f(\cdot)$: Activation function



Activation Functions

$$\mathbf{y}^n = f(\mathbf{A}^n \cdot \mathbf{x}^n + \mathbf{b}^n) \tag{2}$$

- Assume $f(x) = x$
- Layer can assume any linear function (plus offset)
- Stacked layers can't improve that
- Activation function must be non-linear

Activation Functions

Typical choices:

ReLU

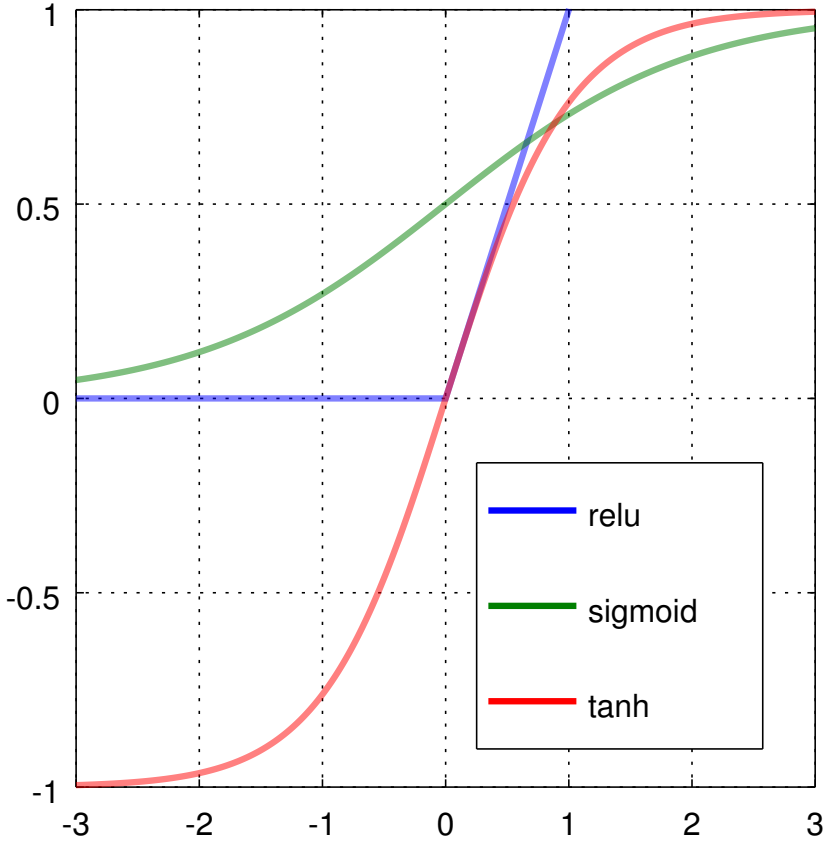
$$f(\mathbf{x}_i)_i = max(\mathbf{x}_i, 0) \quad (3)$$

Sigmoid / Logistic

$$f(\mathbf{x}_i)_i = \frac{1}{1 + e^{-\mathbf{x}_i}} \quad (4)$$

TanH

$$f(\mathbf{x}_i)_i = tanh(\mathbf{x}_i) = \frac{e^{\mathbf{x}_i} - e^{-\mathbf{x}_i}}{e^{\mathbf{x}_i} + e^{-\mathbf{x}_i}} \quad (5)$$



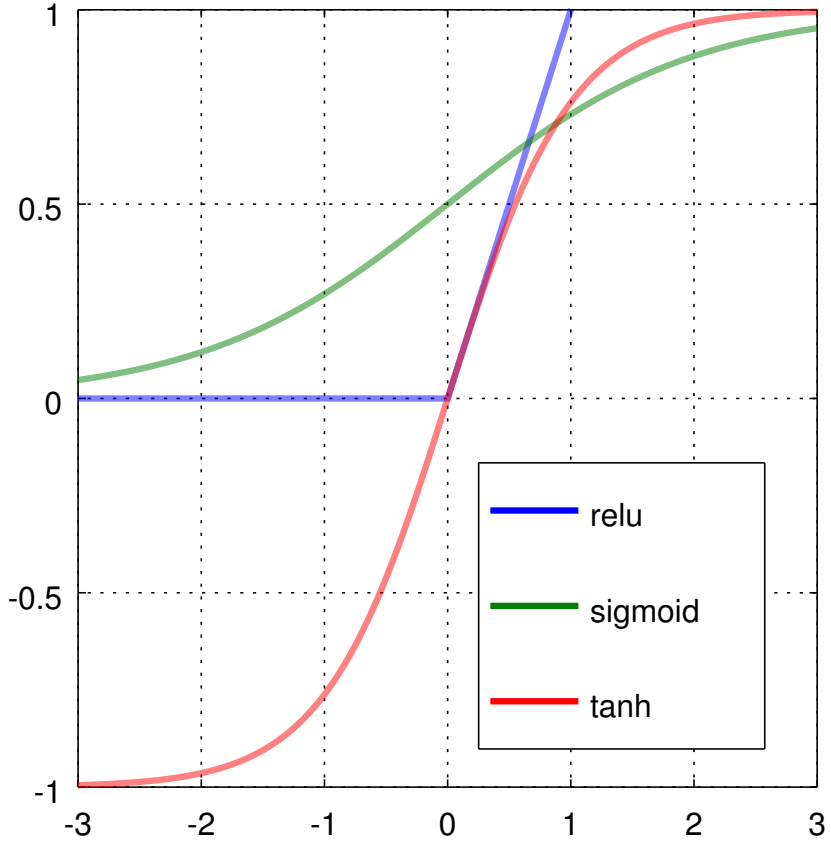
Activation Functions

Typical choices:

ReLU

$$f(\mathbf{x}_i)_i = max(\mathbf{x}_i, 0) \quad (6)$$

- ReLU (and variations of it) today the most common choice
- Better for deep networks
  - Derivative of activation function = 1 (in positive direction)
  - No saturation (in positive direction)
  - Gradients propagate better



Training

How to find correct model parameters $\boldsymbol{\theta}$?

- weight values
- bias values
- sometimes aux parameters

Gradient Descend

- Setup/define energy function objective $E(\boldsymbol{\theta})$
- Derive analytic gradients $\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$
- Perform gradient descend $\Delta\boldsymbol{\theta} = -\lambda \cdot \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$
  - Usually slightly more sophisticated, more later

Training Objective

Assume network with L layers and training data set of N data points:

$$((v_{1,\boldsymbol{x}}, v_{1,\hat{\boldsymbol{y}}}), \ldots, (v_{N,\boldsymbol{x}}, v_{N,\hat{\boldsymbol{y}}})) \tag{7}$$

Empirical Risk Minimization (over N training samples)

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{N} e(\mathbf{y}^L(\underbrace{v_{i,\boldsymbol{x}}}_{\text{Training sample}}, \boldsymbol{\theta}), \overbrace{v_{i,\hat{\boldsymbol{y}}}}^{\text{Known/Desired value/label of } v_{i,\boldsymbol{x}}}) \tag{8}$$

with, e.g.,:

$$e(\mathbf{a}, \mathbf{b}) = ||\mathbf{a} - \mathbf{b}||^2 \tag{9}$$

- Energy function defines training loss
- Gradient descend will try to minimize this
- Usually not convex (as network not convex)

Backpropagation

How to compute $\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$?

- Gradient is sum over gradients for individual training samples: $\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \sum_{i=1}^{N} \frac{\partial}{\partial \boldsymbol{\theta}} e(\mathbf{y}^L(v_{i,\boldsymbol{x}}, \boldsymbol{\theta}), v_{i,\hat{\boldsymbol{y}}})$
- For simplicity, let $N = 1$ and worry about the summation later.
- MLP is concatenation of "simple" functions $\mathbf{y}^L(\ldots \mathbf{y}^2(\mathbf{y}^1(\mathbf{x}^1, \boldsymbol{\theta}^1), \boldsymbol{\theta}^2), \ldots \boldsymbol{\theta}^L)$
- Exploit chain rule

$$\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^k} = \frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{y}^L} \cdot \ldots \cdot \underbrace{\frac{\partial \mathbf{y}^{k+2}}{\partial \mathbf{y}^{k+1}}}_{\text{per layer output derivative}} \cdot \frac{\partial \mathbf{y}^{k+1}}{\partial \mathbf{y}^k} \cdot \overbrace{\frac{\partial \mathbf{y}^k}{\partial \boldsymbol{\theta}^k}}^{\text{per layer parameter derivative}} \tag{10}$$

Backpropagation

$$\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^k} = \frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{y}^L} \cdot \ldots \cdot \underbrace{\frac{\partial \mathbf{y}^{k+2}}{\partial \mathbf{y}^{k+1}}}_{\text{per layer output derivative}} \cdot \frac{\partial \mathbf{y}^{k+1}}{\partial \mathbf{y}^k} \cdot \overbrace{\frac{\partial \mathbf{y}^k}{\partial \boldsymbol{\theta}^k}}^{\text{per layer parameter derivative}} \tag{11}$$

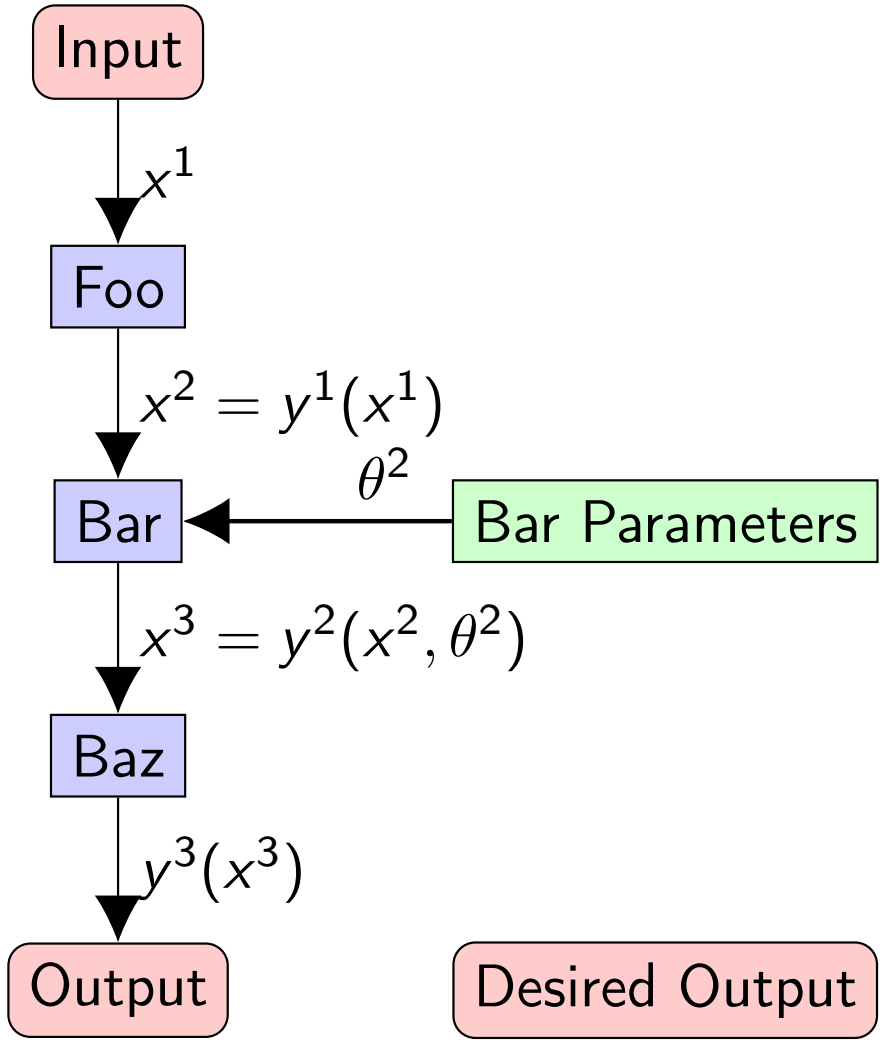
Gradient computation happens in two passes

- Forward pass:
  - Feeds training data through network
  - Computes all $\mathbf{y}^k$ and training loss
- Backward pass:
  - Feeds error gradient backward through network
  - Computes all $\frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{y}^k}$ and $\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^k}$

Training Iteration

Forward Pass

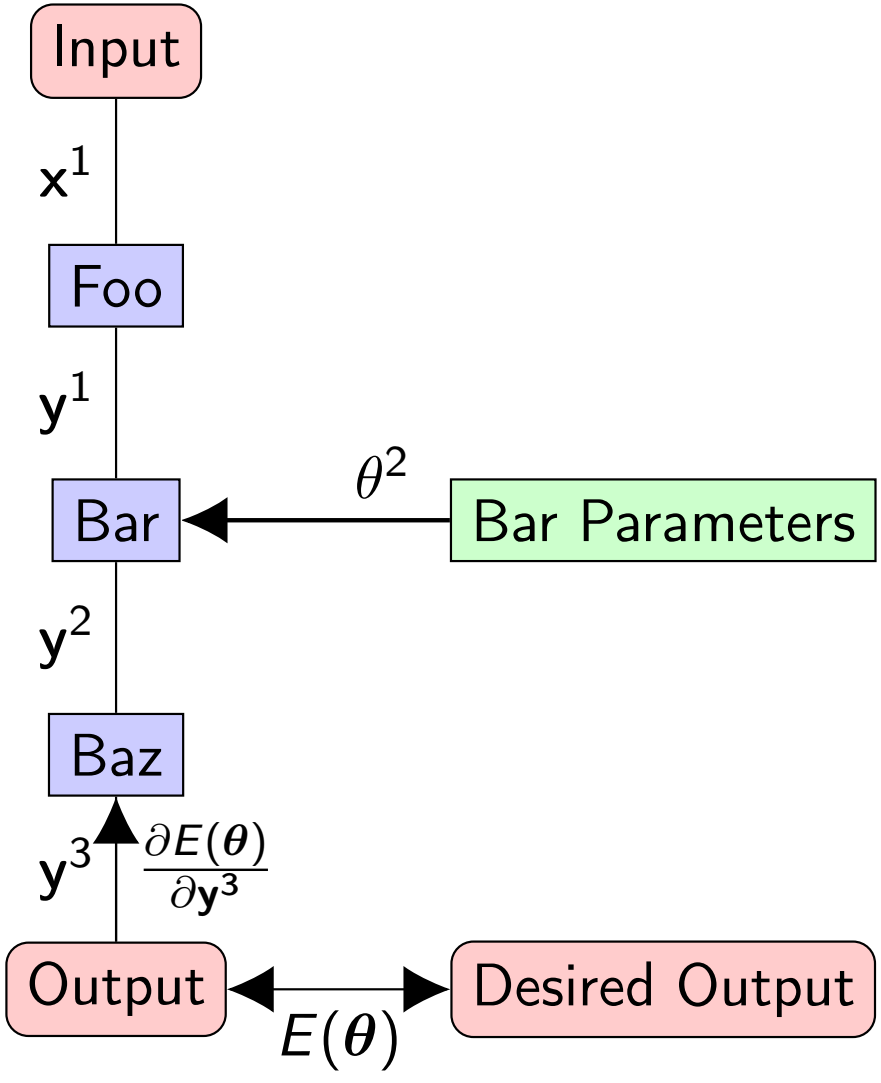
- Start with input data
- Iteratively compute activations **y**
- Store activations (sometimes needed for backwards pass)



Training Iteration

Loss computation

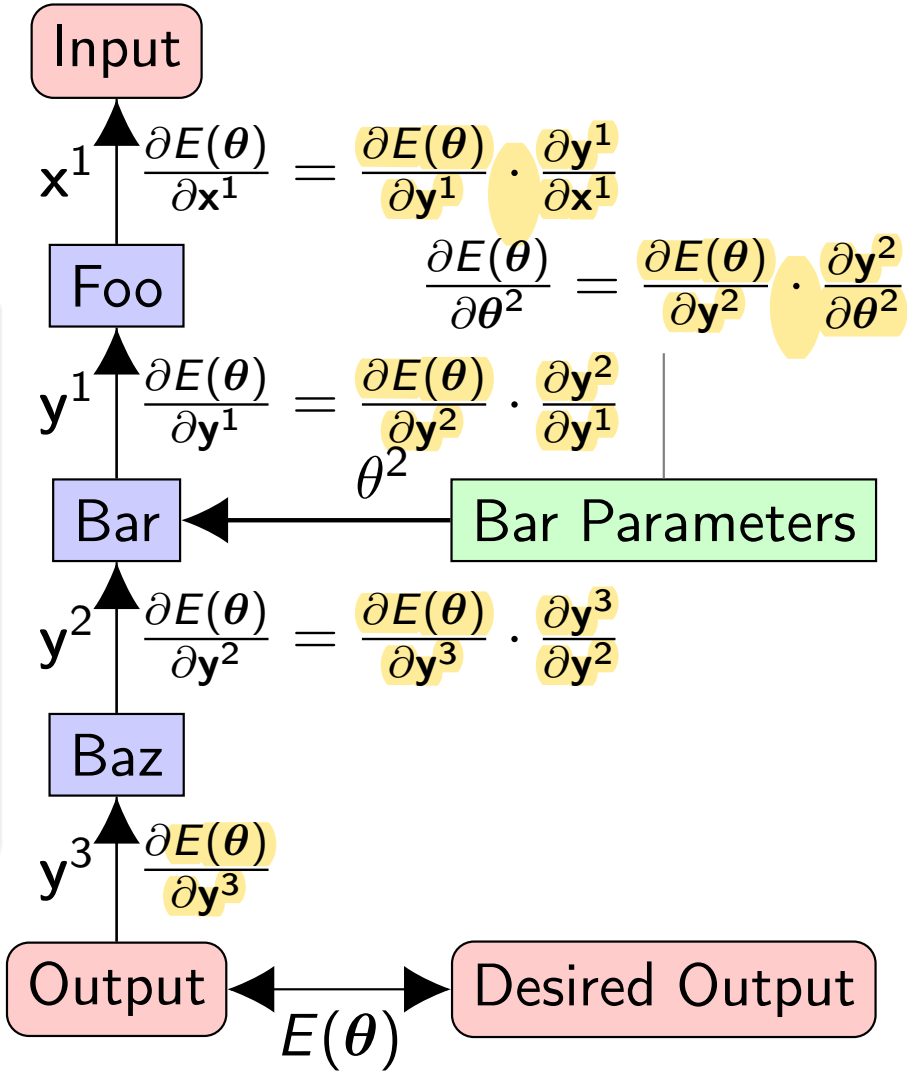
- Compare output $\mathbf{y}_L$ to desired output
- Loss compute loss $E(\boldsymbol{\theta})$ for tracking convergence
- Compute $\frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{y}^L}$



Training Iteration

Backward Pass

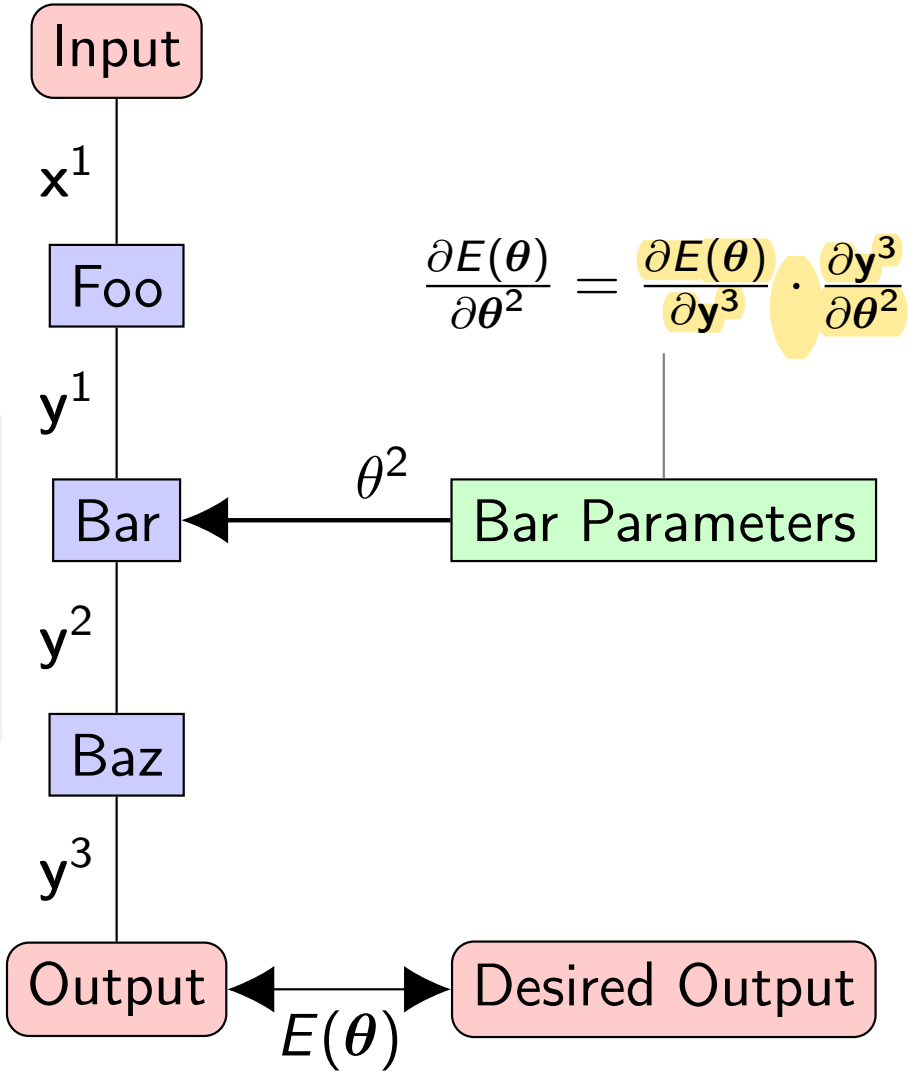
- Propagate gradients backwards through network
- Compute as a running product
- $\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^k} = \frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{y}^k} \cdot \frac{\partial \mathbf{y}^k}{\partial \boldsymbol{\theta}^k}$
- $\frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{y}^{k-1}} = \frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{y}^k} \cdot \frac{\partial \mathbf{y}^k}{\partial \mathbf{y}^{k-1}}$



Training Iteration

Parameter Update

- Update parameters according to update scheme
- $\Delta\boldsymbol{\theta}^k = -\lambda \cdot \frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^k}$



Layers have Common Interface

Unparametrized Layer

Forward:

$$\mathbf{x}^{k+1} = \mathbf{y}^k(\mathbf{x}^k) \tag{12}$$

Backward Data:

$$\frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{x}^k} = \frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{y}^k} \cdot \frac{\partial \mathbf{y}^k}{\partial \mathbf{x}^k} \tag{13}$$

Parametrized Layer

All the above, plus Backward Parameters:

$$\frac{\partial E(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^k} = \frac{\partial E(\boldsymbol{\theta})}{\partial \mathbf{y}^k} \cdot \frac{\partial \mathbf{y}^k}{\partial \boldsymbol{\theta}^k} \tag{14}$$

(note that $\mathbf{x}^{k+1} = \mathbf{y}^k$)

Example: Fully Connected Layer

(Assuming activation function is a separate layer)

$$\mathbf{y}^k(\mathbf{x}^k) = \mathbf{A}^k \cdot \mathbf{x}^k + \mathbf{b}^k \tag{15}$$

Dropping k index:

$$\mathbf{y} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b} \tag{16}$$

$$\mathbf{y}_i = \sum_j \mathbf{A}_{i,j} \cdot \mathbf{x}_j + \mathbf{b}_i \tag{17}$$

$$\frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_j} = \mathbf{A}_{i,j} \tag{18}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{A} \tag{19}$$

$$\frac{\partial E}{\partial \mathbf{x}^k} = \frac{\partial E}{\partial \mathbf{y}^k} \cdot \mathbf{A} \tag{20}$$

Example: Fully Connected Layer

(continued)

$$\mathbf{y}_i = \sum_j \mathbf{A}_{i,j} \cdot \mathbf{x}_j + \mathbf{b}_i \tag{21}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \mathbf{I} \quad \text{(Identity matrix)} \tag{22}$$

$$\frac{\partial E}{\partial \mathbf{b}} = \frac{\partial E}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \frac{\partial E}{\partial \mathbf{y}} \tag{23}$$

$$\frac{\partial \mathbf{y}_i}{\partial \mathbf{A}_{i,j}} = x_j \tag{24}$$

$$\frac{\partial E}{\partial \mathbf{A}_{i,j}} = \left(\frac{\partial E}{\partial \mathbf{y}}\right)_i \cdot \mathbf{x}_j \tag{25}$$

$$\frac{\partial E}{\partial \mathbf{A}} = \overbrace{\left(\frac{\partial E}{\partial \mathbf{y}}\right)^T}^{\text{Outer product}} \cdot \mathbf{x}^T \tag{26}$$

(with some abuse of notation)

Example: Fully Connected Layer

Forward

$$\mathbf{y}(\mathbf{x}^k) = \mathbf{A}^k \cdot \mathbf{x}^k + \mathbf{b}^k \tag{27}$$

Backward Data

$$\frac{\partial E}{\partial \mathbf{x}^k} = \frac{\partial E}{\partial \mathbf{y}^k} \cdot \mathbf{A}^k \tag{28}$$

Backward Parameters

$$\frac{\partial E}{\partial \mathbf{b}^k} = \frac{\partial E}{\partial \mathbf{y}^k} \tag{29}$$

$$\frac{\partial E}{\partial \mathbf{A}^k} = \left(\frac{\partial E}{\partial \mathbf{y}^k}\right)^T \cdot \left(\mathbf{x}^k\right)^T \tag{30}$$

Example: ReLU Layer

Forward

$$\mathbf{y}(\mathbf{x}^k) = \max(\mathbf{x}^k, 0) \tag{31}$$

Backward Data

$$\left(\frac{\partial E}{\partial \mathbf{x}^k}\right)_i = \left(\frac{\partial E}{\partial \mathbf{y}^k}\right)_i \cdot \begin{cases} 1 & \text{if } (\mathbf{x}^k)_i > 0 \\ 0 & \text{else} \end{cases} \tag{32}$$

Stochastic Gradient Descend

- Exact gradient usually not needed or wanted
- Just empirical average over N samples anyways
- Stochastic Gradient Descend: Split into batches of $M < N$ samples and update weights after every batch

$$\Delta\boldsymbol{\theta} = -\lambda \cdot \frac{\partial \hat{E}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{\partial}{\partial \boldsymbol{\theta}} \sum_{k}^{M} e(\mathbf{y}^L(v_{k,\boldsymbol{x}}, \boldsymbol{\theta}), v_{k,\hat{\boldsymbol{y}}}) \tag{33}$$

Usually small batch sizes (eg. around 128) sufficient

- Stepsize limited by curvature, not by precision of gradient
- Time increases with $O(M)$, precision of gradient only with $O(\sqrt{M})$
- Large batch sizes lead to sharp minimizers that don't generalize
- Further reading: [Keskar et al., 2016]

Parameter Update Rule

Vanilla Steepest (Gradient) Descend

$$\Delta\boldsymbol{\theta} = -\lambda \cdot \frac{\partial \hat{E}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \qquad (34)$$

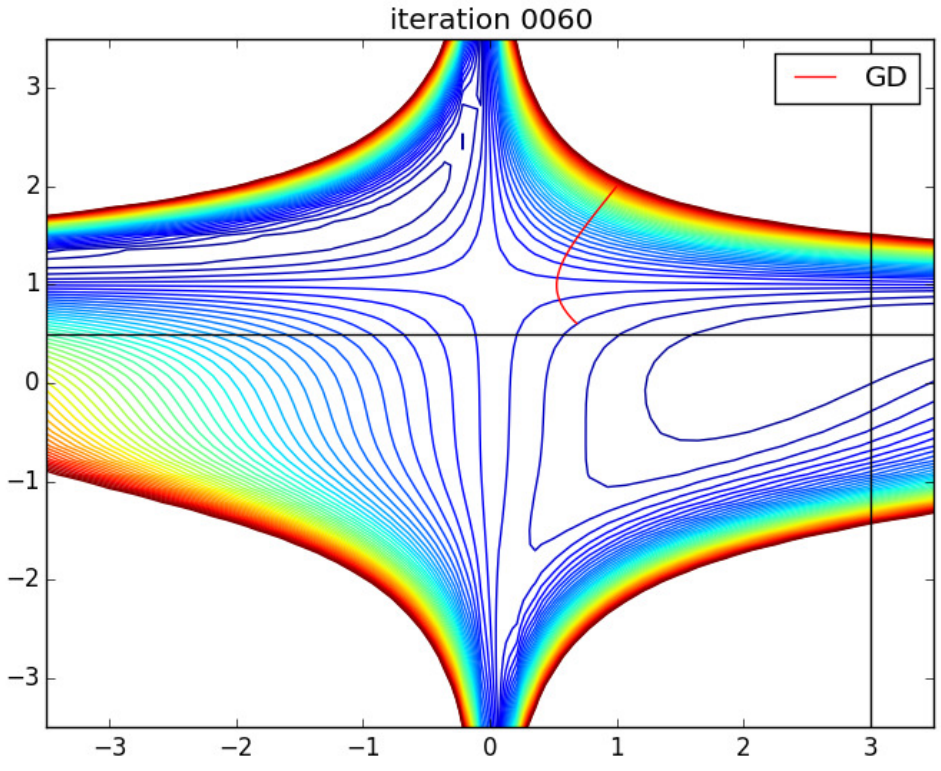
- Most simple update rule
- How to choose λ?

Parameter Update Rule

Vanilla Steepest (Gradient) Descend

$$\Delta\boldsymbol{\theta} = -\lambda \cdot \frac{\partial \hat{E}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \qquad (35)$$

λ too small

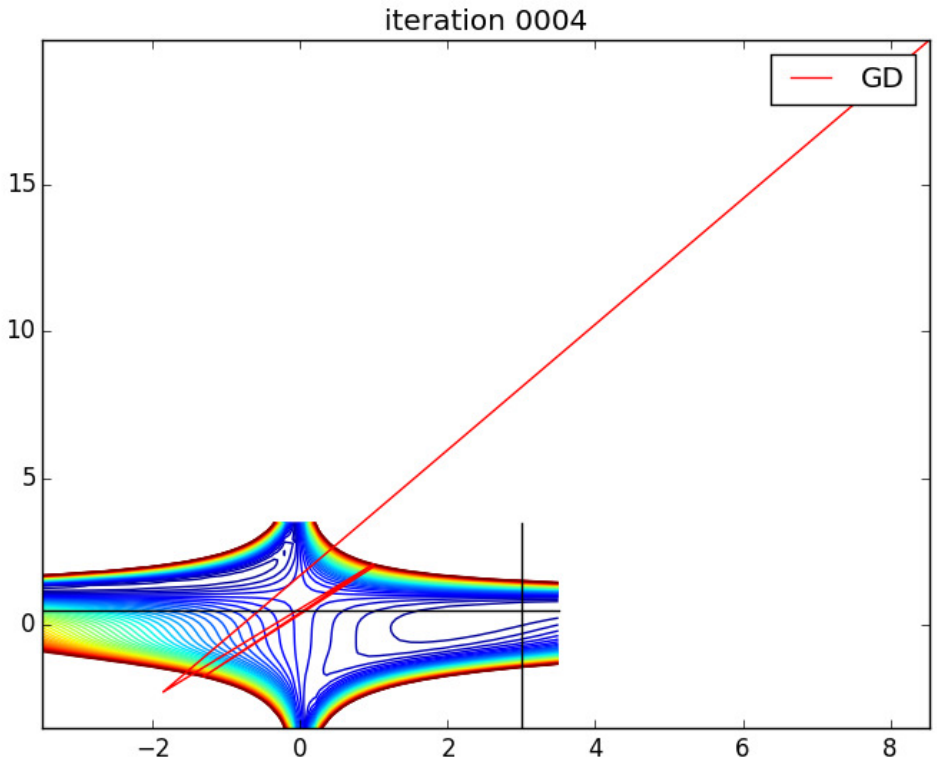


Parameter Update Rule

Vanilla Steepest (Gradient) Descend

$$\Delta\boldsymbol{\theta} = -\lambda \cdot \frac{\partial \hat{E}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \qquad (36)$$

λ too large

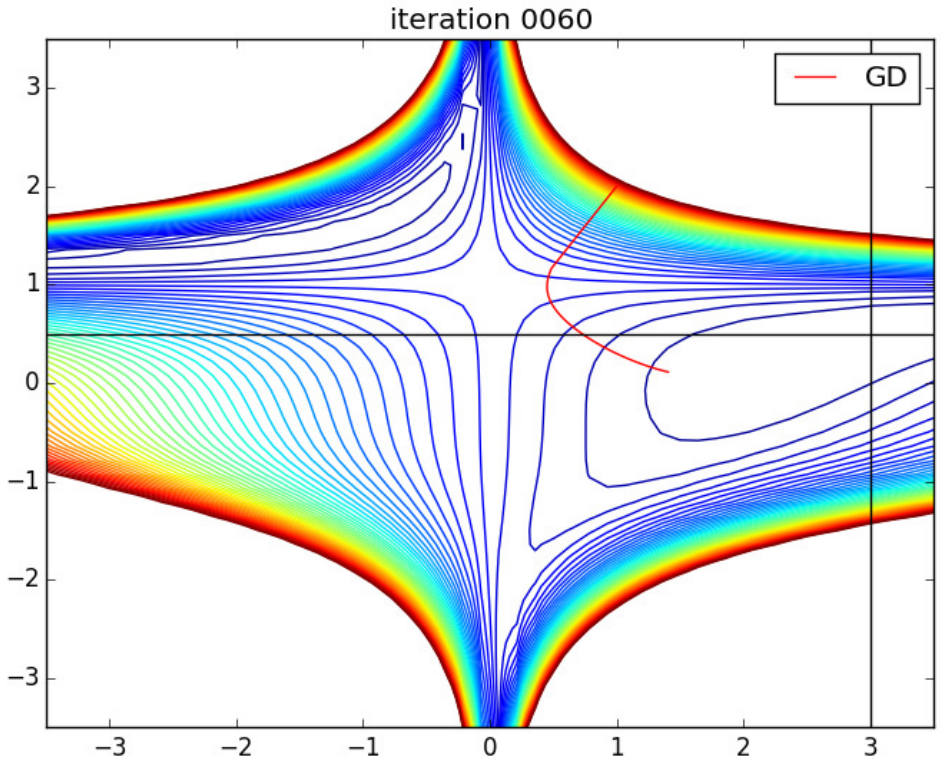


Parameter Update Rule: Steepest Descend

Vanilla Steepest (Gradient) Descend

$$\Delta\boldsymbol{\theta} = -\lambda \cdot \frac{\partial \hat{E}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \qquad (37)$$

λ just right



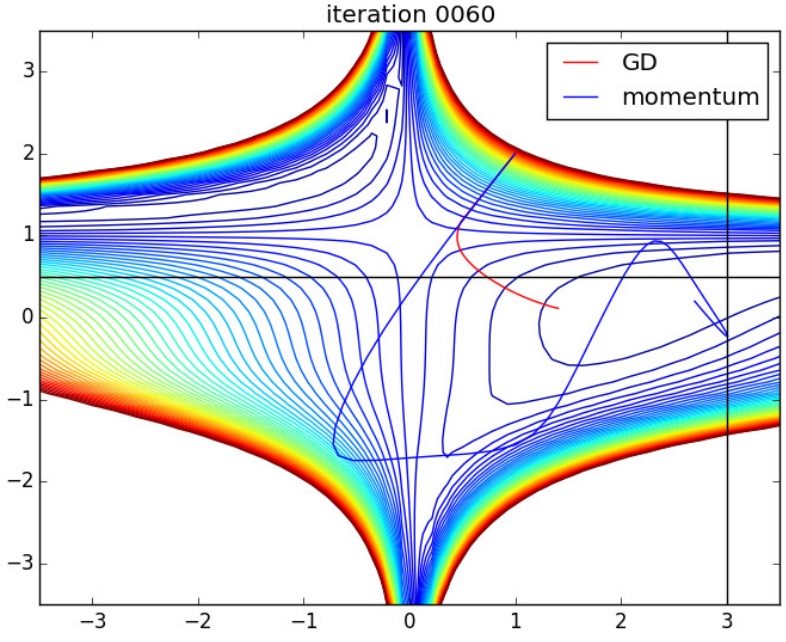
Parameter Update Rule: Momentum

Gradient Descend with Momentum

$$m_t = \beta \cdot m_{t-1} - \lambda \cdot \frac{\partial \hat{E}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{38}$$

$$\Delta\boldsymbol{\theta} = m_t \tag{39}$$

- Accumulate "momentum" over time
- Pick up speed in the valley direction, average out noise



Parameter Update Rule: Adam

Adam [Kingma and Ba, 2015]

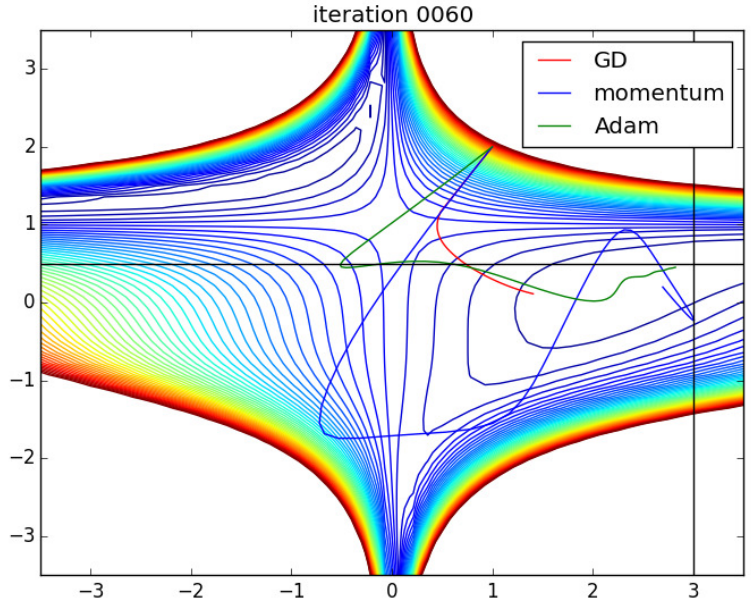
$$g_t = \frac{\partial \hat{E}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{40}$$

$$m_t = \beta_1 \cdot g_{t-1} + (1-\beta_1) \cdot g_t \tag{41}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2 \tag{42}$$

$$\Delta\boldsymbol{\theta} = -\lambda \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{43}$$

ignoring here some details for the first iterations; all operations element wise



- Compute variance of gradients
- Accelerate on smooth slopes

Parameter Update Rule: Adam

Adam [Kingma and Ba, 2015]

$$g_t = \frac{\partial \hat{E}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \quad (44)$$

$$m_t = \beta_1 \cdot g_{t-1} + (1 - \beta_1) \cdot g_t \quad (45)$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \quad (46)$$

$$\Delta\boldsymbol{\theta} = -\lambda \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} \quad (47)$$

ignoring here some details for the first iterations; all operations element wise

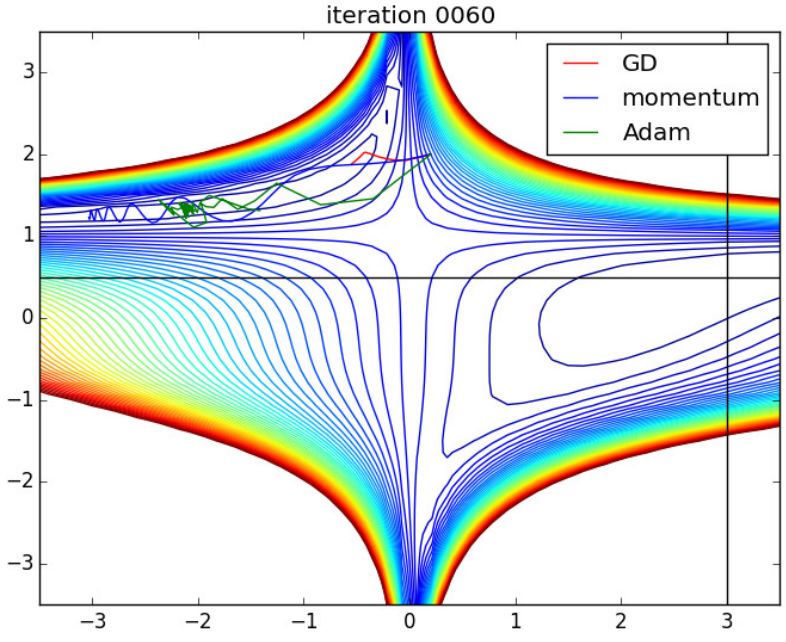
More variants exist:

- Adadelta [Zeiler, 2012]
- Adagrad
- ...

- Compute variance of gradients
- Accelerate on smooth slopes

Local Minima

- All methods only converge to a local minimum
- Solution depends on initialization



Local Minima

Interesting Property: Over-parametrization seems to help!

- Initialize with more channels (bigger vectors)
- Train
- Prune "unnecessary channels"

Lottery Ticket Hypothesis [Frankle and Carbin, 2018]

- Assume a certain amount of channels is needed
- More channels means subset is sufficient
- Number of permutations of sufficient sub-networks grows rapidly
- The more sub-network, the higher the chance to win the "initialization lottery"

Other things that help:

- Network architecture (ResNet)
- Noisy Gradients (small BatchSize)

Parameter Initialization

How to initialize θ?

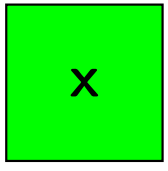
- Random Gaussian
- Xavier (and some variants) [Glorot and Bengio, 2010]
  - Draw weights randomly
  - Choose variance per layer depending on input/output size
  - Balance variance to keep signal/gradient variance constant

ConvNets

MLP to ConvNet
Change of data

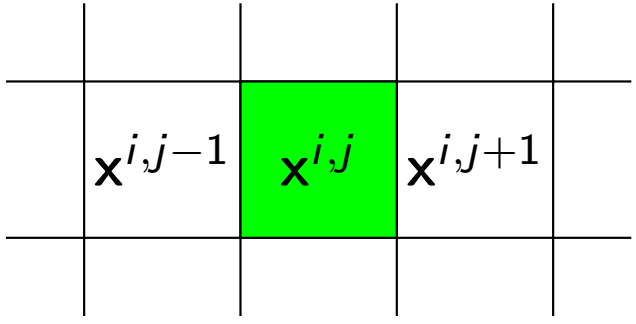
The data changes:

Multi Layer Perceptron



Single vector $\mathbf{x} \in \mathbb{R}^c$

ConvNet



Grid of vectors $\mathbf{x}^{i,j} \in \mathbb{R}^c$

Let's call c the number of channels.

Similarly, for ConvNets the gradients are also grids of vectors, i.e.:
$\mathbf{x}, \frac{\partial E}{\partial \mathbf{x}} \in \mathbb{R}^{h \times w \times c}$

ConvNets

Most of the stuff works like before:

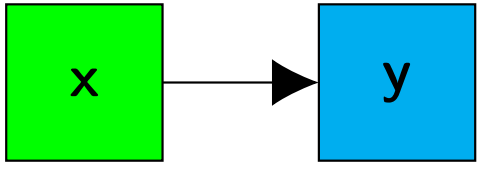
- Training: Forward + backward pass
- Loss function: Basically element wise (for regression)
- Activation/ReLU: Element wise
- Branching: Element wise copy on forward, addition on backward pass
- Parameter update: Independent of data

Only real change: The convolution layer

MLP to ConvNet
From FC to Conv

(dropping all layer indices and focusing on a single layer)

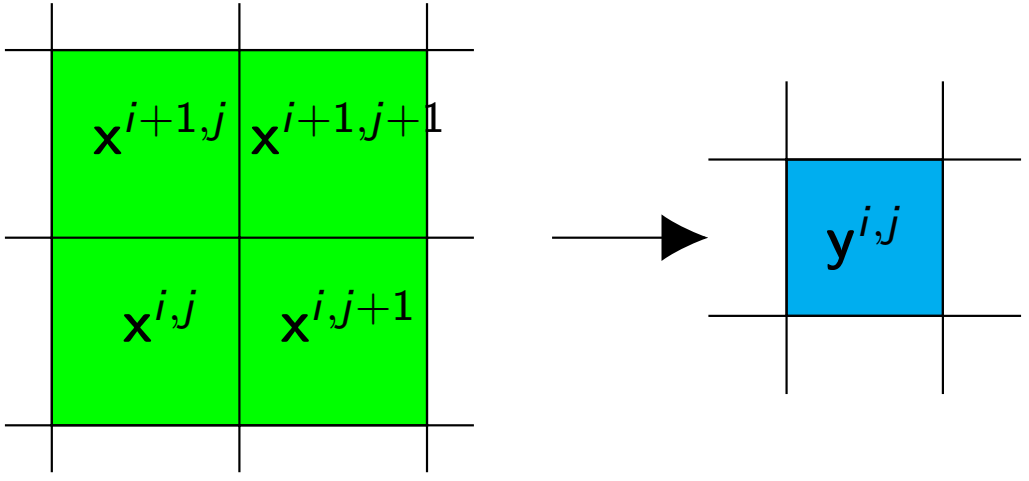
Fully Connected



$$\mathbf{x} \in \mathbb{R}^{c_1}, \mathbf{y} \in \mathbb{R}^{c_2}$$

$$\mathbf{y} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b} \qquad (48)$$

Conv Layer

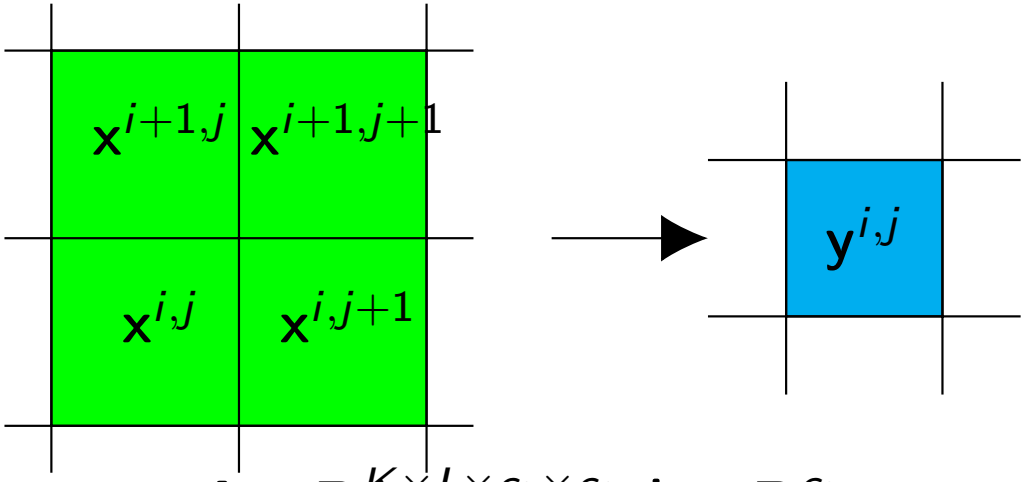


$$\mathbf{x} \in \mathbb{R}^{h_1 \times w_1 \times c_1}, \mathbf{y} \in \mathbb{R}^{h_2 \times w_2 \times c_2}$$

$$\mathbf{y}^{i,j} = \sum_{k,l} \mathbf{A}^{k,l} \cdot \mathbf{x}^{i+k,j+l} + \mathbf{b} \qquad (49)$$

Note: For ease of implementation, conv layers are often actually correlations!

Conv Layer
Forward pass



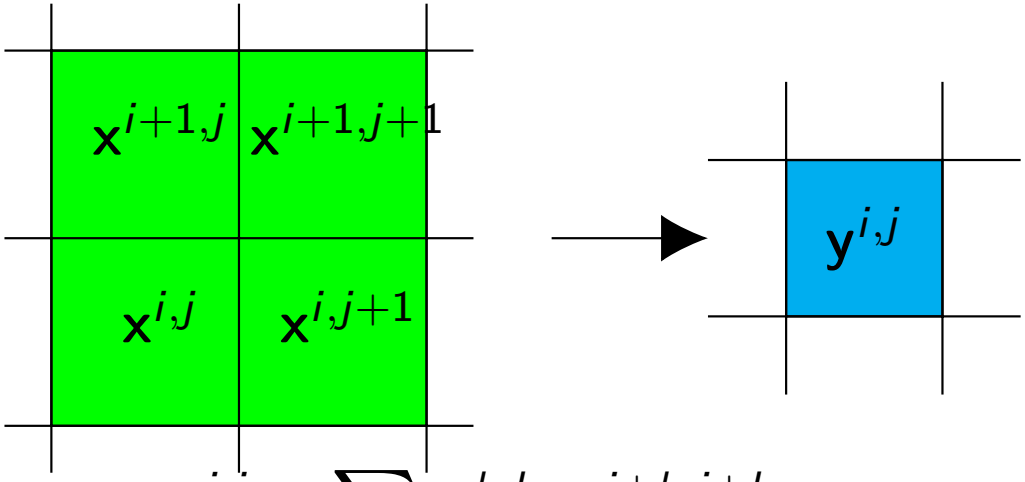
$$\mathbf{A} \in \mathbb{R}^{K \times L \times c_2 \times c_1}, \mathbf{b} \in \mathbb{R}^{c_2}$$

$$\mathbf{x} \in \mathbb{R}^{h_1 \times w_1 \times c_1}, \mathbf{y} \in \mathbb{R}^{h_2 \times w_2 \times c_2}, h_2 = h_1 - K + 1, w_2 = w_1 - L + 1$$

$$\mathbf{y}^{i,j} = \sum_{k,l} \mathbf{A}^{k,l} \cdot \mathbf{x}^{i+k,j+l} + \mathbf{b} \tag{50}$$

- To stop messing with kernel flipping, *technically correlation*
- Kernel elements are matrices: $\mathbf{A}^{k,l} \in \mathbb{R}^{c_2 \times c_1}$
- Each pixel kernel-element multiplication is a matrix-vector multiplication

Conv Layer
Backward pass data



$$\mathbf{y}^{i,j} = \sum_{k,l} \mathbf{A}^{k,l} \cdot \mathbf{x}^{i+k,j+l} + \mathbf{b} \tag{51}$$

$$\overbrace{\frac{\partial E}{\partial \mathbf{x}}}^{\text{wanted}} = \overbrace{\frac{\partial E}{\partial \mathbf{y}}}^{\text{from next layer}} \cdot \overbrace{\frac{\partial \mathbf{y}}{\partial \mathbf{x}}}^{\text{needed}} \tag{52}$$

$$\frac{\partial \mathbf{y}^{i,j}}{\partial \mathbf{x}^{i+k,j+l}} = \begin{cases} \mathbf{A}^{k,l} & \text{if} \quad 0 \leq k < K \wedge 0 \leq l < L \\ \mathbf{0} & \text{else} \end{cases} \tag{53}$$

(Like in MLP case, see Slide 49)

$$\mathbf{y}^{i,j} = \sum_{k,l} \mathbf{A}^{k,l} \cdot \mathbf{x}^{i+k,j+l} + \mathbf{b} \tag{54}$$

$$\frac{\partial \mathbf{y}^{i,j}}{\partial \mathbf{x}^{i+k,j+l}} = \begin{cases} \mathbf{A}^{k,l} & \text{if} \quad 0 \leq k < K \wedge 0 \leq l < L \\ \mathbf{0} & \text{else} \end{cases} \tag{55}$$

Shift of coordinates:

$$\frac{\partial \mathbf{y}^{i-k,j-l}}{\partial \mathbf{x}^{i,j}} = \begin{cases} \mathbf{A}^{k,l} & \text{if} \quad 0 \leq k < K \wedge 0 \leq l < L \\ \mathbf{0} & \text{else} \end{cases} \tag{56}$$

$$\frac{\partial E}{\partial \mathbf{x}^{i,j}} = \sum_{k,l} \frac{\partial E}{\partial \mathbf{y}^{i-k,j-l}} \cdot \frac{\partial \mathbf{y}^{i-k,j-l}}{\partial \mathbf{x}^{i,j}} = \sum_{k,l} \frac{\partial E}{\partial \mathbf{y}^{i-k,j-l}} \cdot \mathbf{A}^{k,l} \tag{57}$$

If the forward pass is a correlation of input with kernel:

$$\overbrace{\mathbf{y}^{i,j}}^{\text{output}} = \sum_{k,l} \mathbf{A}^{k,l} \cdot \overbrace{\mathbf{x}^{i+k,j+l}}^{\text{input}} + \mathbf{b} \tag{58}$$

The backward data pass is a convolution (with shift) of output gradient with kernel:

$$\underbrace{\frac{\partial E}{\partial \mathbf{x}^{i,j}}}_{\substack{\text{input gradient} \\ \text{wanted}}} = \sum_{k,l} \underbrace{\frac{\partial E}{\partial \mathbf{y}^{i-k,j-l}}}_{\substack{\text{output gradient} \\ \text{from next layer}}} \cdot \mathbf{A}^{k,l} \tag{59}$$

$$\mathbf{y}^{i,j} = \sum_{k,l} \mathbf{A}^{k,l} \cdot \mathbf{x}^{i+k,j+l} + \mathbf{b} \tag{60}$$

Bias gradients from bias-output-jacobian and output-gradients:

$$\frac{\partial E}{\partial \mathbf{b}} = \sum_{i,j} \frac{\partial E}{\partial \mathbf{y}^{i,j}} \cdot \overbrace{\frac{\partial \mathbf{y}^{i,j}}{\partial \mathbf{b}}}^{\text{Identity matrix}} \tag{61}$$

(Like in MLP case, see Slide 49)

$$\frac{\partial E}{\partial \mathbf{b}} = \sum_{i,j} \frac{\partial E}{\partial \mathbf{y}^{i,j}} \tag{62}$$

$$\mathbf{y}^{i,j} = \sum_{k,l} \mathbf{A}^{k,l} \cdot \mathbf{x}^{i+k,j+l} + \mathbf{b} \tag{63}$$

Weight gradients from weight-output-jacobian and output-gradients:

$$\frac{\partial \mathbf{y}_{c_2}^{i,j}}{\partial \mathbf{A}_{c_2,c_1}^{k,l}} = \mathbf{x}_{c_1}^{i+k,j+l} \quad 0 \leq k < K \wedge 0 \leq l < L \tag{64}$$

$$\frac{\partial E}{\partial \mathbf{A}_{c_2,c_1}^{k,l}} = \sum_{i,j} \frac{\partial E}{\partial \mathbf{y}_{c_2}^{i,j}} \cdot \frac{\partial \mathbf{y}_{c_2}^{i,j}}{\partial \mathbf{A}_{c_2,c_1}^{k,l}} = \sum_{i,j} \frac{\partial E}{\partial \mathbf{y}_{c_2}^{i,j}} \cdot \mathbf{x}_{c_1}^{i+k,j+l} \tag{65}$$

$$\frac{\partial E}{\partial \mathbf{A}^{k,l}} = \sum_{i,j} \left(\frac{\partial E}{\partial \mathbf{y}^{i,j}} \right)^T \cdot \left(\mathbf{x}^{i+k,j+l} \right)^T \tag{66}$$

(with some abuse of notation like in Slide 48)

ConvLayer

Forward

Correlation of input values and kernel (plus bias):

$$\mathbf{y}^{i,j} = \sum_{k,l} \mathbf{A}^{k,l} \cdot \mathbf{x}^{i+k,j+l} + \mathbf{b} \quad (67)$$

Backward Data

Convolution of output gradients and kernel:

$$\frac{\partial E}{\partial \mathbf{x}^{i,j}} = \sum_{k,l} \frac{\partial E}{\partial \mathbf{y}^{i-k,j-l}} \cdot \mathbf{A}^{k,l} \quad (68)$$

Backward Parameters: Bias

Sum of output gradients:

$$\frac{\partial E}{\partial \mathbf{b}} = \sum_{i,j} \frac{\partial E}{\partial \mathbf{y}^{i,j}} \quad (69)$$

Backward Parameters: Weights

Correlation of output gradients and input values:

$$\frac{\partial E}{\partial \mathbf{A}^{k,l}} = \sum_{i,j} \left(\frac{\partial E}{\partial \mathbf{y}^{i,j}} \right)^T \cdot \left(\mathbf{x}^{i+k,j+l} \right)^T \quad (70)$$

Performance

Optimization

So far, optimization was not important in DIP. Why care now?

Example Convnet

Back of the envelope calculation:

- 64×64 pixel input data (ignore border effects)
- 5 layers, all 64 channels (for simplicity)
- all 3×3 sized kernels

How many multiplications and additions are needed for a single forward pass (only Conv, ignoring ReLU)?

$$\overbrace{5}^{\text{layers}} \cdot \overbrace{64 \cdot 64}^{\text{all pixels}} \cdot \overbrace{3 \cdot 3}^{\text{kernel summation}} \cdot \overbrace{64 \cdot 64}^{\text{MatMul products}} \cdot \overbrace{2}^{\text{Multiply + Add}} = 1'509'949'440 \tag{71}$$

About 1.5 billion computations!

Training this for 500'000 iterations with a mini-batch size of 16: About $1.2080 \cdot 10^{16}$ operations or 12 quadrillion computations. Plus backward pass.

Optimization

So far, optimization was not important in DIP. Why care now?

12 quadrillion computations to do, your CPU can maybe push 50-250 billion computations per second.
Better not waste any!

Heap Allocation

What does the following compile to?

```
struct Vec {
    float x, y;
};
Vec vec;
```

Answer: Usually nothing

Maybe (but usually fused with frame setup):

```
sub     $0x10,%rsp
```

Heap Allocation

What does the following compile to?

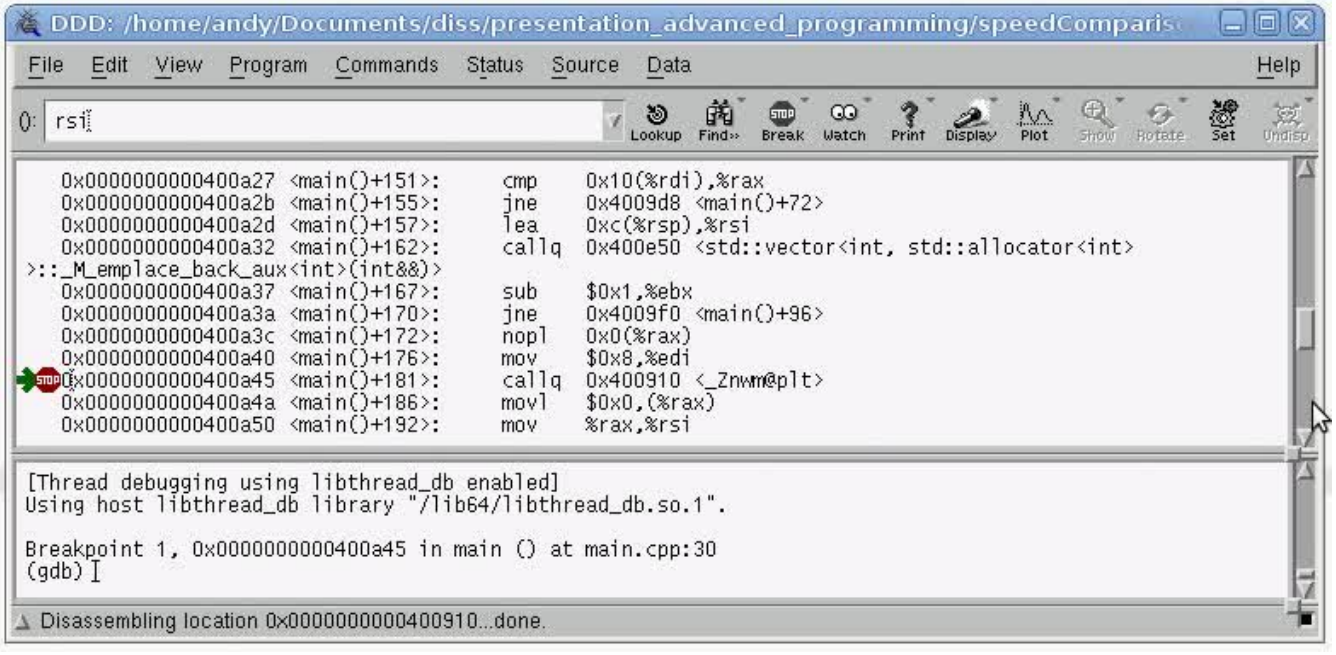
```
struct Vec {
    float x, y;
};
Vec *vec = new Vec();
```

Heap Allocation

What does the following compile to?

```
struct Vec {
    float x, y;
};
Vec *vec = new Vec();
```

Answer:



The Rules

The rules of this homework:

1. No heap allocation (at least not inside inner loops)!

   - No malloc
   - No new
   - No std::vector::resize etc.
   - No cv::Mat::create etc.

   Tip: Don't use any OpenCV functions, just use for loops.

Memory

My CPU can do about 75 billion float ops per second.
How fast is the following code (only the addition)?

```
a = np.zeros((2048, 2048))
b = np.zeros((2048, 2048))
c = a + b
```

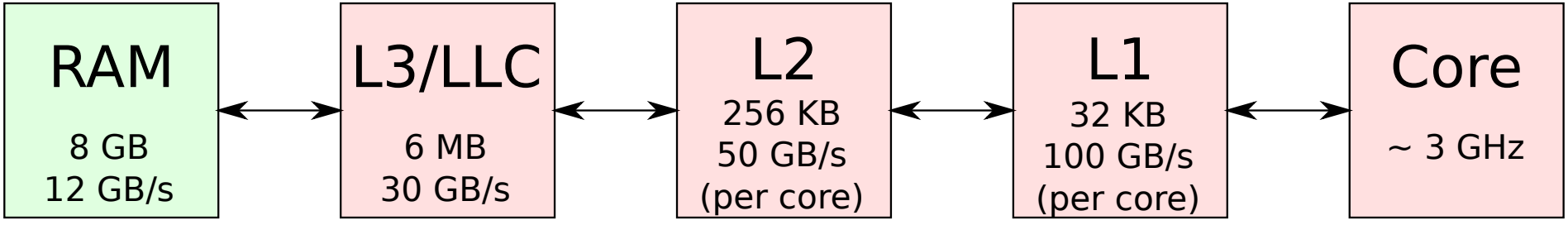
$$\frac{2048 \cdot 2048}{75'000'000'000} = 0.055924ms \tag{72}$$

It actually takes $4.25ms$!

$$\frac{2048 \cdot 2048}{4.25ms} = 1 \text{ billion float ops per second} \tag{73}$$

Memory

DRAM is *really slow*:



12 GB/s is 3 billion floats per second. Every operation reads two floats and write one.

My CPU can do 75 GFLOPS, but my RAM only supports 1 GFLOPS.

DRAM burst is 64 byte wide: Every access is always 64 bytes.

Don't random-access your RAM. Read blocks in sequence, reuse data in cache!

Tensor class

In the homework, the `Tensor` class implements 4D blocks of floats.

```
dip6::Tensor data;

data.allocate(H, W, C, N);

for (unsigned h = 0; h < data.getSize(0); h++)
    for (unsigned w = 0; w < data.getSize(1); w++)
        for (unsigned c = 0; c < data.getSize(2); c++)
            for (unsigned n = 0; n < data.getSize(3); n++)
                data(h, w, c, n) = ....
```

Whenever possible, organize loops in the order of the indices.

When order is unimportant:

```
for (unsigned i = 0; i < data.getTotalSize(); i++)
    data[i] = ....
```

The Rules

The rules of this homework:

1. No heap allocation (at least not inside inner loops)!
2. Access data in sequence

Whenever possible, organize loops in the order of the tensor indices.

Multi-Threading

CPUs have multiple cores, each core can source instructions from multiple threads. Make use of this by splitting the work up into chunks, e.g.:

Single threaded

```
for (unsigned h = 0; h < data.getSize(0); h++) {
    for (unsigned w = 0; w < data.getSize(1); w++)
    // ...
        data(h, w, c, n) = ....
}
```

Multi threaded

```
parallelFor(0, data.getSize(0), 1, [&](unsigned h) {
    for (unsigned w = 0; w < data.getSize(1); w++)
    // ...
        data(h, w, c, n) = ....
});
```

Signature: `parallelFor(` start index, end index, chunk size, functor)

Single Instruction Multiple Data

All float operations are actually element-wise vector operations.

- ARM: Usually 4 elements
- Intel and AMD: Usually 4-8 elements
- NVidia: 32 elements
- ATI: 64 elements

Full access through *intrinsics*.

SIMD Abstraction

dip6::simd::Vector<> stores a vector of floats

All operations performed element wise.

```
dip6::simd::Vector<instances> a, b, c;
a.load(&input1(h, w, c, 0));
b.load(&input2(h, w, c, 0));
c = a + b; // element wise addition
c.store(&output(h, w, c, 0));
```

dip6::simd::Scalar stores a single float

```
dip6::simd::Vector<instances> a;
dip6::simd::Scalar f;
a.load(&input(h, w, c, 0)); // loads multiple floats
f.load(&kernel(h, w, c1, c2)); // loads a single float
a *= f; // applied to all elements in a
a.store(&output(h, w, c, 0)); // stores multiple floats
```

`instances` is the emulated width of the vector. It must be a compile time constant (e.g. a template parameter).

Data Layout

To make your code easier, vectorization over instances:

- Number of instances in mini batch is multiple of 8
- In the optimized case, inner-most loop (over instances) handled by vector operations

```
dip6::simd::Tensor v, k, b;
```

Data layout for values and gradients:

v(row, column, channel, instance)

Data layout for kernel and kernel gradients:

k(row, column, outputChannel, inputChannel)

Data layout for bias and bias gradients:

b(0, 0, outputChannel, 0)

Homework

Structure

Executables

- createDB: reads DIV2K images and creates shuffled dataset of crops
- trainSmall: trains a small convnet to learn identity function
- trainBig: trains a less small convnet to learn super resolution
- application: loads a parameter snapshot and applies super resolution to an image

Files

- one .cpp for each executable
- bunch of files to handle plumbing, training, SIMD, threading
- Dip6.h/cpp: where you need to implement stuff
- shuffled dataset of DIV2K crops (separate download)
  - large training dataset + smaller validation dataset

Your Task I

I recommend to do it in this order:

1. Reference implementation ConvLayer:
   - Implement forward pass
   - Backward data pass is given
   - Backward parameter pass is given
2. Optimized implementation of ReLU:
   - Implement forward pass
   - Implement backward data pass
   - Use SIMD, multi threading not necessary
3. Implement MSE Loss:
   - No SIMD or multi threading needed
4. Run trainSmall (does it learn identity function?)

Your Task II

5. Implement Adam:
   - Implementation of SGD with momentum is given as reference
   - No SIMD or multi threading needed
6. Optimized implementation of ConvLayer
   - Implement forward pass
   - Implement backward data pass
   - Backward parameter pass is given (because it's ugly)
7. Run trainBig (this might run for a night)
8. Apply trained network to an image of your choice

Check unit tests in between!

ConvLayer reference forward pass

Signature:

```
void reference_convolutionForward(
        const Tensor &input, const Tensor &kernel, const Tensor &bias,
        Tensor &output)
{
    // TO DO !!!
}
```

$$\mathbf{y}^{i,j} = \sum_{k,l} \mathbf{A}^{k,l} \cdot \mathbf{x}^{i+k,j+l} + \mathbf{b} \tag{74}$$

- Assume output is allocated to right size
- Loop over all output pixels, output channels, and instances
  - Initialize a summation float to the bias value of the output channel
  - Loop over all kernel elements and input channels
    - Add product of kernel and input value to summation float
  - Store summation float to output

ReLU Layer forward pass

Signature:

```
void ReLU::forward(const Tensor &input)
{
    m_output.allocateLike(input);
    // TO DO !!!
}
```

$$\mathbf{y}(\mathbf{x}^k) = \max(\mathbf{x}^k, 0) \tag{75}$$

Implement with SIMD-vectorization:

- Assume total size is multiple of 8
- Loop over all `input.getTotalSize()` elements in blocks of 8
- Process with `simd::Vector<8>`

Tip:

```
simd::Scalar zero(simd::INIT_ZERO);
simd::Vector<8> v;
// ...
v = simd::max(v, zero);
```

ReLU Layer backward pass

Signature:

```
void ReLU::backward(const Tensor &input, const Tensor &outputGradients)
{
    m_inputGradients.allocateLike(input);
    // TO DO !!!
}
```

$$\left(\frac{\partial E}{\partial \mathbf{x}^k}\right)_i = \left(\frac{\partial E}{\partial \mathbf{y}^k}\right)_i \cdot \begin{cases} 1 & \text{if } \left(\mathbf{x}^k\right)_i > 0 \\ 0 & \text{else} \end{cases} \tag{76}$$

Implement with SIMD-vectorization:

- Assume total size is multiple of 8
- Loop over all `input.getTotalSize()` elements in blocks of 8
- Compare input to zero to create a mask
  - `simd::Vector<8> mask = a > b;`
- Based on mask, select between output gradient or zero
  - `simd::Vector<8> vOrZero = simd::selectOrZero(mask, v);`
- Store to input gradient tensor

MSE Loss

```
float MSELoss::computeLoss(
        const Tensor &computedOutput, const Tensor &desiredOutput,
        Tensor &computedOutputGradients)
{
    computedOutputGradients.allocateLike(computedOutput);
    // TO DO !!!
    return 0.0f;
}
```

- Due to non-padding of filters, `computedOutput` is smaller then `desiredOutput`
  - Only consider centered inner region
- Compute average squared difference between `computedOutput` and `desiredOutput`
  - For averaging, divide by `computedOutput.getTotalSize()`
  - Return this loss as return value
- Compute gradient of loss $\frac{\partial E}{\partial y}$
  - $\frac{\partial E}{\partial y} = \frac{2}{H \cdot W \cdot C \cdot N} (\mathbf{y} - \hat{\mathbf{y}})$
  - Divide by `computedOutput.getTotalSize()`
  - Return gradients in `computedOutputGradients`

Adam Optimizer

Signature:

```
void Adam::performStep(float stepsize)
{
    const float oneMinusBeta1 = 1.0f - m_beta1;
    const float oneMinusBeta2 = 1.0f - m_beta2;
    for (unsigned i = 0; i < m_parameterMomentum.size(); i++) {
        Tensor &Tvariance = m_parameterGradExpectation[i];
        Tensor &Tmomentum = m_parameterMomentum[i];
        Tensor &Tgradients = m_layer->getParameterGradients()[i];
        Tensor &Tvalues = m_layer->getParameters()[i];

        for (unsigned i = 0; i < Tvalues.getTotalSize(); i++) {
            // TO DO !!!
        }
    }
}
```

No SIMD/Multi threadring, just implement the element wise formulas. SGD with momentum is given for reference. Assume $epsilon = 10^{-8}$.

ConvLayer optimized forward pass

Signature:

```
template<unsigned inputChannels,
         unsigned instances>
void convolutionForward_CN(const Tensor &input, const Tensor &kernel, const T
{
    // TO DO !!!
}
```

Same as reference implementation, but:

- Use parallelFor on outermost loop
- Use `simd::Vector<instances>` to vectorize over instances (Inner most instance loop no longer needed).
- Use `simd::Scalar` to load the bias value to initialize accu with.
- Use `inputChannels` and `instances` in favor of `input.getSize(2)` and `input.getSize(3)` (helps compiler).

ConvLayer optimized backwards data pass

Signature:

```
template<unsigned inputChannels,
         unsigned outputChannels,
         unsigned instances>
void convolutionBackwardData_ION(const Tensor &input, const Tensor &kernel, c
{
    // TO DO !!!
}
```

Same as reference implementation, but:

- Use parallelFor on outermost loop
- Use `simd::Vector<instances>` to vectorize over instances (Inner most instance loop no longer needed).
- Use `simd::Vector<instances>::setZero()` to initialize accu to zero.
- Use template parameters in favor of `input.getSize(...)` (helps compiler).

Submission

As always I want:

- Your code (*.cpp, *.h, CMakeLists.txt)
- No build products
- Your input and output image

Deadlines & Next Meeting

Deadline: 28th of January
Deadline is in one week!

Next meeting is in two weeks: 4th of February
Bring your own questions, I'll not have content prepared.

Agustsson, E. and Timofte, R. (2017).

Ntire 2017 challenge on single image super-resolution: Dataset and study.

In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

Frankle, J. and Carbin, M. (2018).

The lottery ticket hypothesis: Training pruned neural networks.

CoRR, abs/1803.03635.

Glorot, X. and Bengio, Y. (2010).

Understanding the difficulty of training deep feedforward neural networks.

In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, pages 249–256.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2016).

On large-batch training for deep learning: Generalization gap and sharp minima.

CoRR, abs/1609.04836.

Kingma, D. P. and Ba, J. (2015).

Adam: A method for stochastic optimization.

In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012).

Imagenet classification with deep convolutional neural networks.

In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998).

Gradient-based learning applied to document recognition.

In *Proceedings of the IEEE*, pages 2278–2324.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015).

ImageNet Large Scale Visual Recognition Challenge.

International Journal of Computer Vision (IJCV), 115(3):211–252.

Zeiler, M. D. (2012).

ADADELTA: an adaptive learning rate method.

CoRR, abs/1212.5701.