Introduction to Deep learning: a 2-weeks lecture Part 1

Presented by: Dra. Jeaneth Machicao
machicao@usp.br

October/2020

Course overview: STAT 453: Deep Learning, Spring 2020 by Prof. Sebastian Raschka

Part1: Introduction

- Introduction to deep learning
- The brief history of deep learning
- Single-layer neural networks: The perceptron
- Motivation: cases of use
- Hands-on

Part2: Mathematical and computational foundations

- Linear algebra and calculus for deep learning
- Parameter optimization with gradient descent
- Automatic differentiation & PyThorch

Part3: Introduction to neural networks

- Multinomial logistic regression
- Multilayer pecerptrons
- Regularization
- Input normalization and weight initiliazation
- Learning rated and advanced optimization algorithms

Part4: DL for computer vision and language modeling

- Introduction to convolutional neural networks 1-2
  - CNNs Architectures Illustrated
- Introduction to recurrent neural networks 1-2

Part5: Deep generative models

- Autoencoders,
- Autoregressive models
- Variational autoencoders
- Normalizing Flow models
- Generative adversarial networks
- Evaluating generative models

http://stat.wisc.edu/~sraschka/teaching/stat453-ss2020/
https://github.com/rasbt/stat453-deep-learning-ss20

- Course Playlist on youtube:

Prof. Dalcimar Casanova
https://www.youtube.com/watch?v=0VD_2t6EdS4&list=PL9At2PVRU0ZqVArhU9QMyI3jSe113_m2-
Prof. Sebastian Raschka
https://www.youtube.com/watch?v=e_I0q3mmfw4&list=PLTKMiZHVd_2JkR6QtQEnml7swCnFBtq4P

Overview of our 2-weeks lecture!

1st week

1: Introduction

- Introduction to deep learning
- The brief history of deep learning
- Single-layer neural networks: The perceptron
- Motivation: cases of use
- Hands-on (report)

2: Mathematical and computational foundations

- Linear algebra and calculus for deep learning
- Parameter optimization with gradient descent
- Automatic differentiation & PyThorch

3: Introduction to neural networks

- Multinomial logistic regression
- Multilayer pecerptrons
- Regularization
- Input normalization and weight initiliazation
- Learning rated and advanced optimization algorithms

2nd week

4: DL for computer vision and language modeling

- Introduction to convolutional neural networks 1-2
  - CNNs Architectures Illustrated
- Introduction to recurrent neural networks 1-2

3rd week

- Deliver report of the hands-on

http://stat.wisc.edu/~sraschka/teaching/stat453-ss2020/
https://github.com/rasbt/stat453-deep-learning-ss20

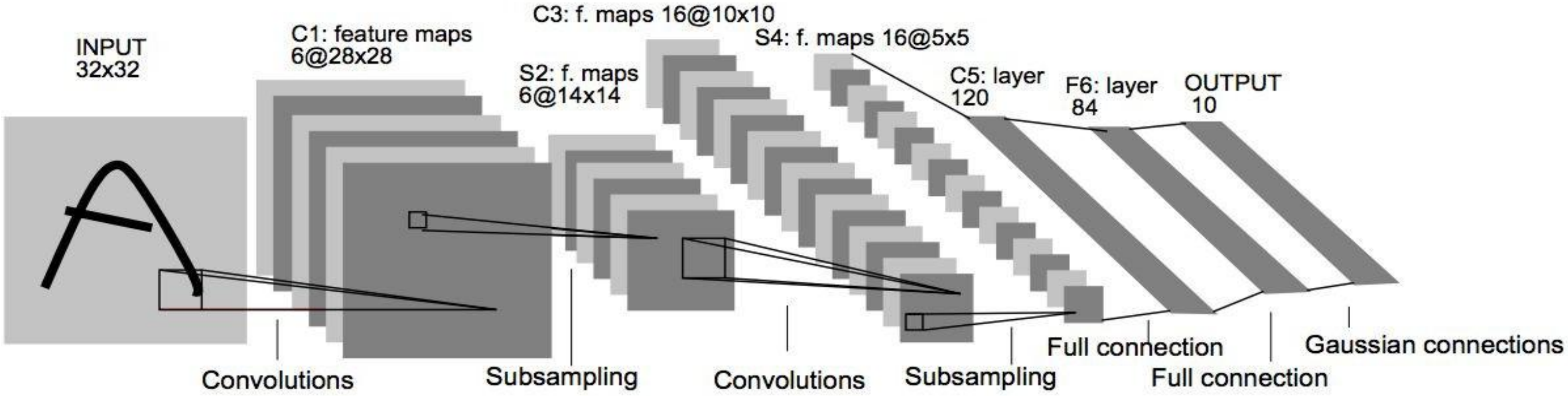
- Course Playlist on youtube:

Prof. Dalcimar Casanova
Prof. Sebastian Raschka

When you move on to Deep Learning

Deep learning（DL）- A little of history（I）

- Lenet - Classic CNN. They were born in the early 90s.
- Predecessor: Neocognitron



(Y. LeCun - 1998)

Source: Rayner Montes

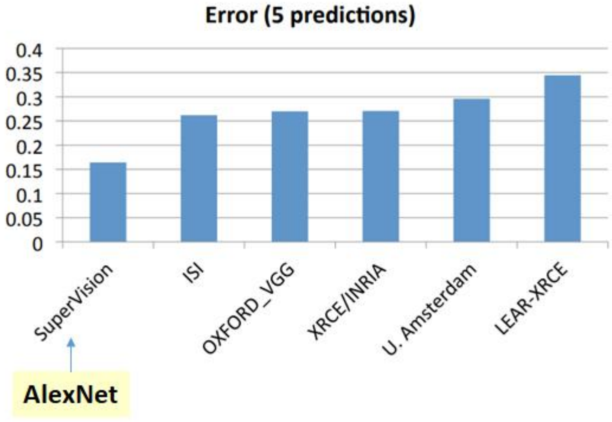
Deep learning (DL) - A little of history (II)

Big paper: "ImageNet Classification with Deep Convolutional Neural Networks" (Alex Krizhevsky - 2012)

- **Task:** Object classification, 1000 classes. Millions of training images (ImageNet competition)

- ***Alexnet*** was much better than all state of the art methods.

Ranking of the best results from each team



Source: Rayner Montes

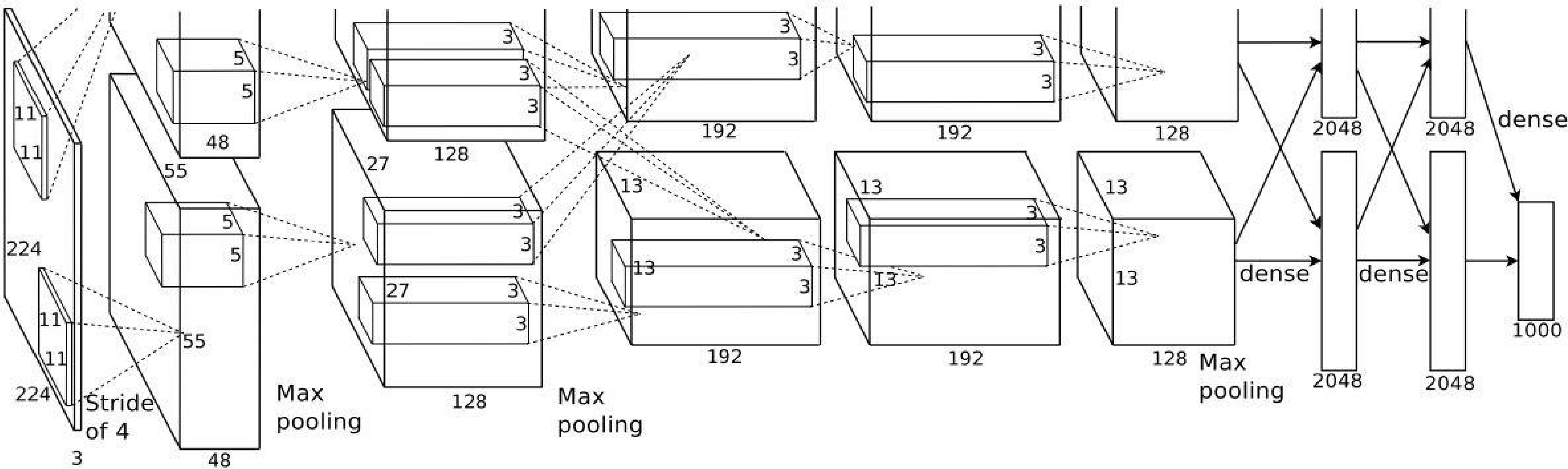
Why did it take so long?
From early 90s to 2012

Answer: NO GPUS

GeForce GTX 1080

Why GPU？(I)

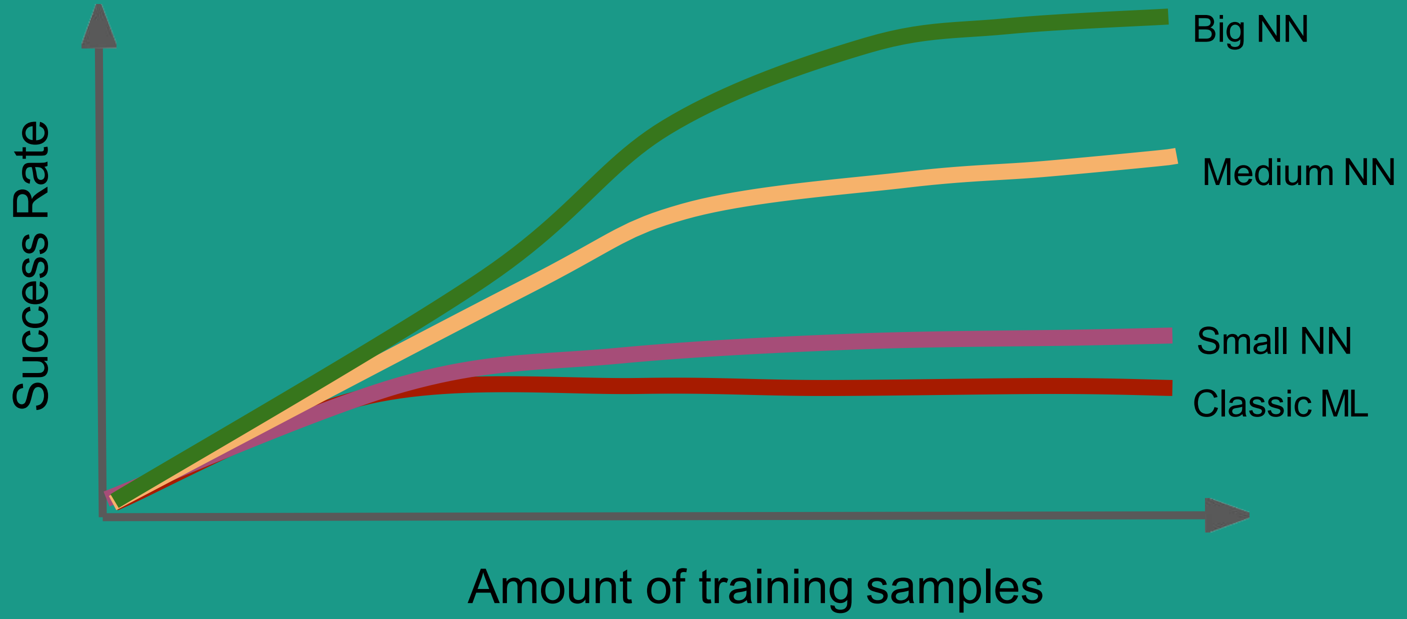
More layers → more training samples → more execution time



Source: Rayner Montes

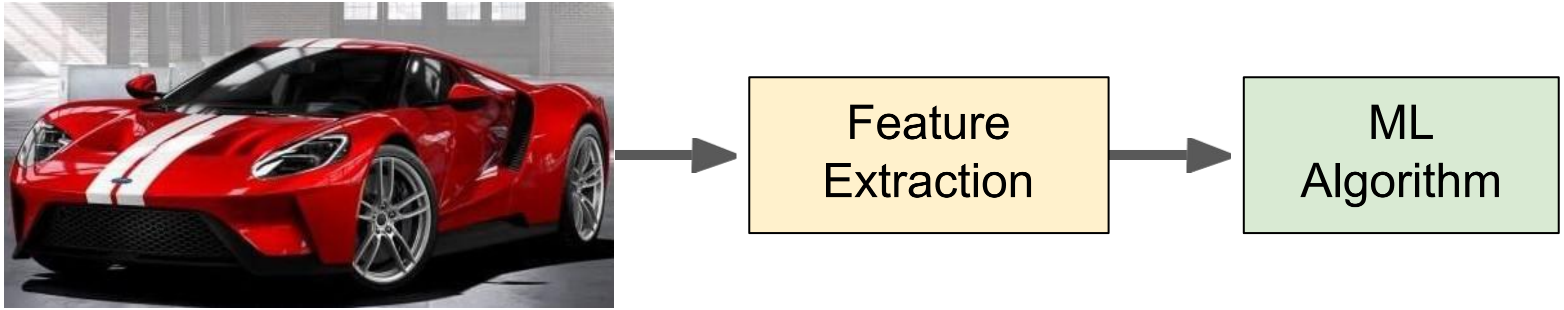
Why GPU？(II)

- Training Samples



Source: Rayner Montes

Object Classification – Classic Machine Learning

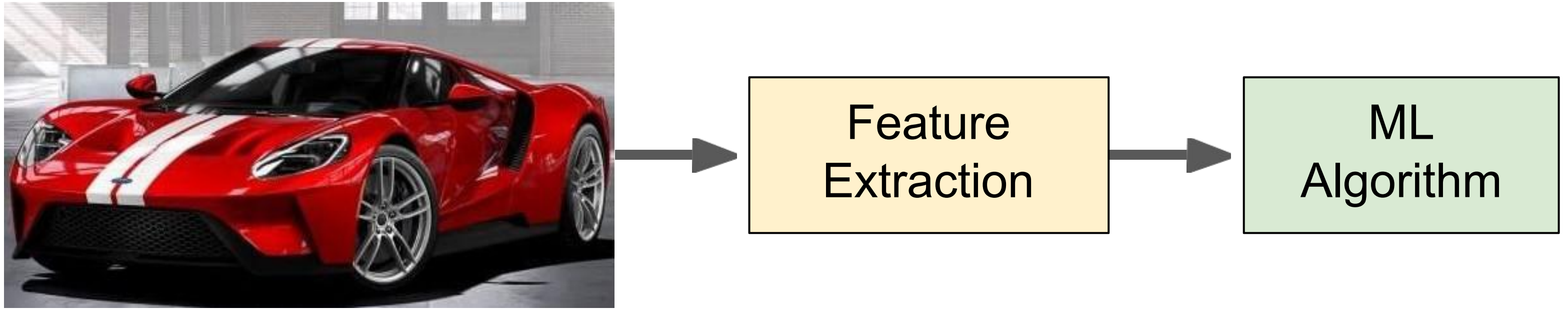


Feature Extraction

ML Algorithm

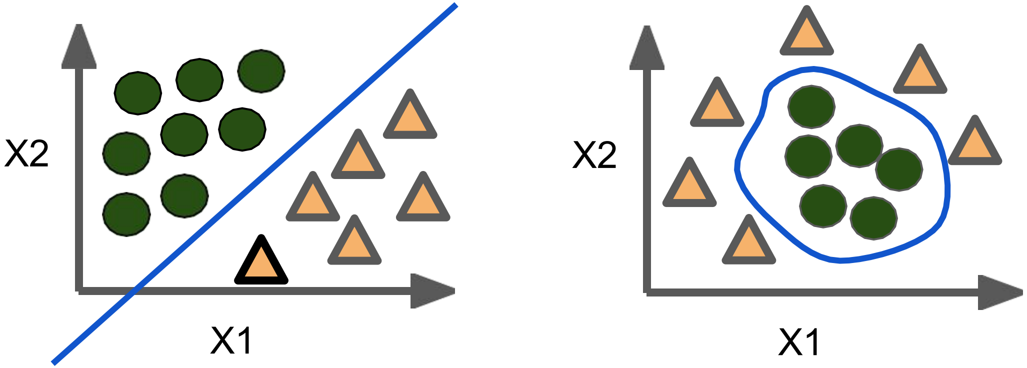
Source: Rayner Montes

Object Classification – Classic ML



Feature Extraction

ML Algorithm

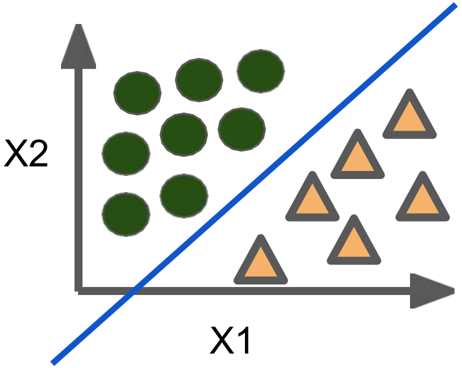


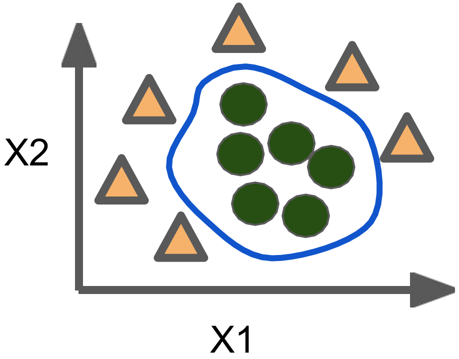
Source: Rayner Montes

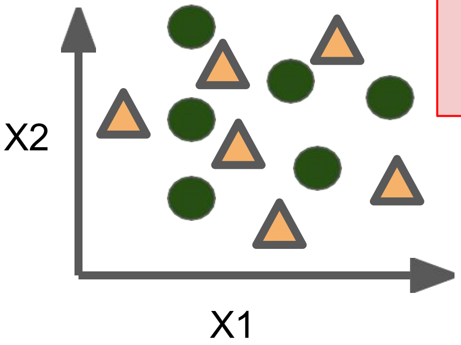
Object Classification – Classic ML

Feature Extraction

ML Algorithm



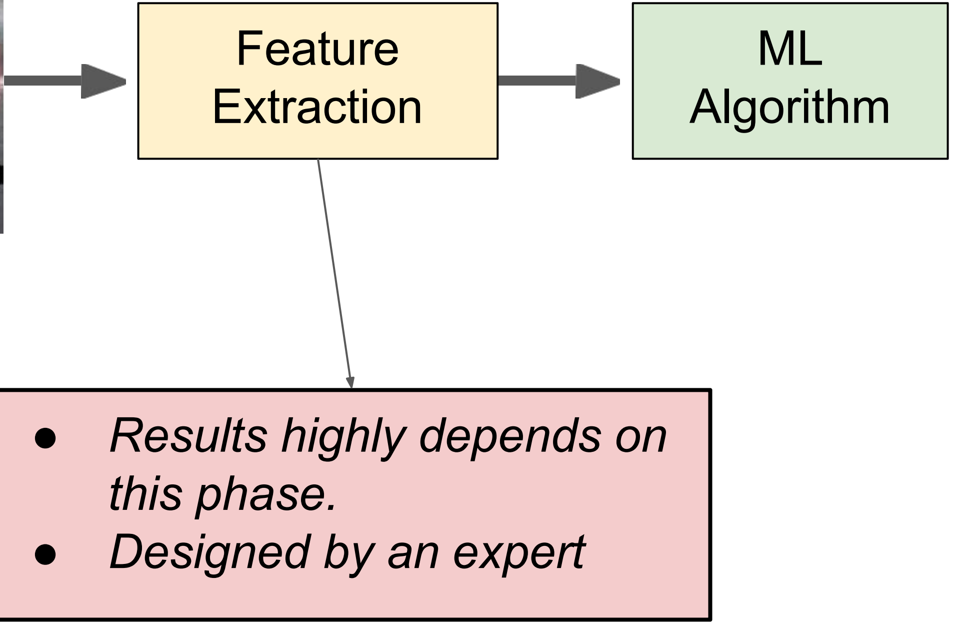




??

Source: Rayner Montes

Object Classification – Classic ML



Feature Extraction

ML Algorithm

- *Results highly depends on this phase.*
- *Designed by an expert*

Source: Rayner Montes

https://paperswithcode.com/sota

https://github.com/terryum/awesome-deep-learning-papers

State-of-the-art

Computer Vision

- Image Segmentation
- Image Classification
- Object Detection
- Image Generation

Natural Language Processing

- Machine Translation
- Question Answering
- Sentiment Analysis
- Text Classification

Medical

- Medical Image Segmentation
- Drug Discovery
- Lesion Segmentation
- Brain Tumor Segmentation

Speech

- Speech Recognition
- Speech Synthesis
- Speech Enhancement
- Speaker Verification

Time Series

- Imputation
- Time Series Classification
- Time Series Forecasting
- Gesture Recognition

Audio

- Music Generation
- Audio Classification
- Audio Generation
- Sound Event Detection

Music

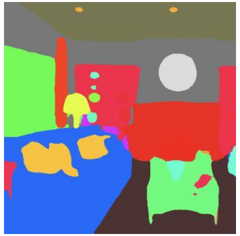
- Music Generation
- Music Information Retrieval
- Music Source Separation
- Music Modeling

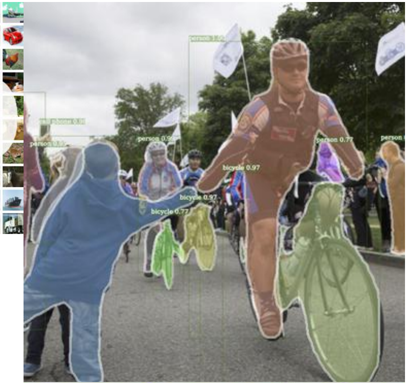
Computer Code

- Dimensionality Reduction
- Feature Selection
- Code Generation
- Program Synthesis

Playing Games

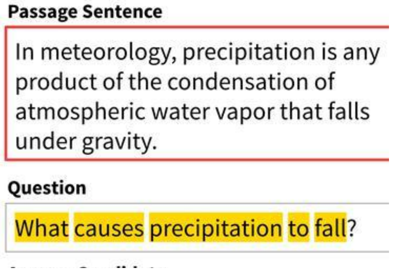
- Atari Games
- Continuous Control
- Starcraft
- Real-Time Strategy Games





Computer Vision

- Image Segmentation; Image Classification
- Object Detection; Image Generation
- Super-Resolution; Autonomous Vehicles
- Video...

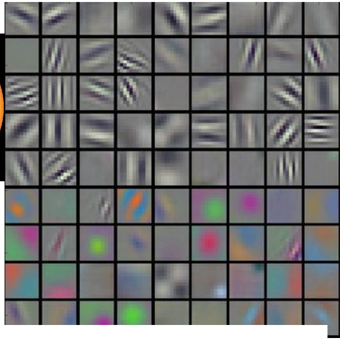


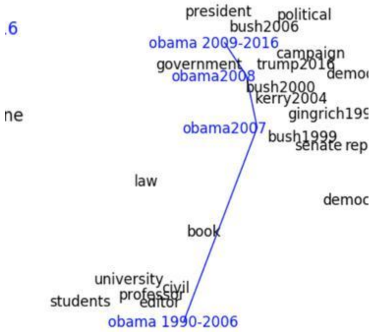
Passage Sentence

In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.

Question

What causes precipitation to fall?





Natural Language Processing

- Machine Translation
- Question Answering
- Sentiment Analysis
- Text Classification
- Representation Learning
- Word Embeddings

Some Applications Of Machine Learning/Deep Learning

AI in marketing & sales: Propensity to buy

The problem

- A lack of knowledge about a customer's propensity to buy
- "*Propensity to buy*" is the likelihood of a customer to purchase a particular product.

What can be achieved?

- Classify potential customers by their likelihood to purchase a particular product.
- This can be integrated into to marketing and sales strategies.

Opportunity for DL

- Model using a combination of semantic analysis:
  - Text written by the customer,
  - Demographic information,
  - Purchase history
  - Information about how they navigate the website to make a prediction for that customer's propensity to buy.

Data requirements:

- A model like this would need historical data of demographics and pre-purchase behavior of customers linked to if a purchase was made.

Source: https://peltarion.com/use-cases/propensity-to-buy

Using AI to detect fraud

The problem

- Globally, fraud costs ~£3.24tn.
- Classical done by rules-based algorithms
  - typically complicated and not always very hard to circumvent.

What can be achieved?

- The improved accuracy promises substantial cost reduction for many industries and sectors.

Opportunity for DL

- Detect complicated underlying patterns from seemingly unrelated information.
- Ability to continuously learn and evolve to remain up to date with a dynamic environment.

Data requirements:

- Historical data of demographics and pre-purchase behavior of customers
  - fraudulent or normal.

Source: https://peltarion.com/use-cases/fraud-detection

Automated defect detection

The problem

- Product quality testing is slow and inefficient (bottlenecks).
- Traditional automated systems are both expensive and difficult to implement.Opportunity for deep learning

What can be achieved?

- AI to reduce production cost, speed and accuracy.

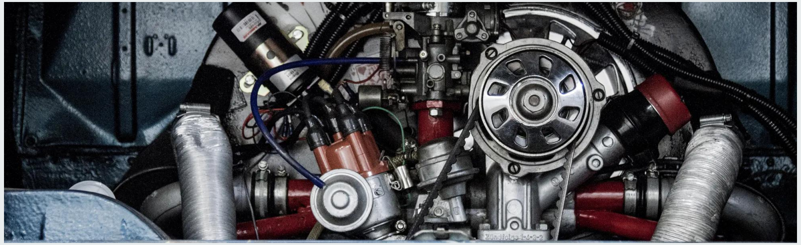
Opportunity for DL

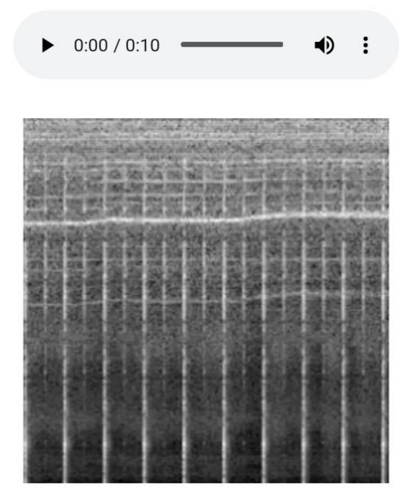
- DL for fully automated production line and enable more accurate analysis of the quality of each individual part.

Data requirements

- Trained on images of manufactured parts,
  - defective or non-defective.
- Cameras mounted on the production line feed images to the model.

Source: https://peltarion.com/use-cases/defect-detection





Mel-spectrogram of an industrial solenoid valve

Source: https://peltarion.com/use-cases/machinery-operating-conditions

Audio analysis for industrial maintenance

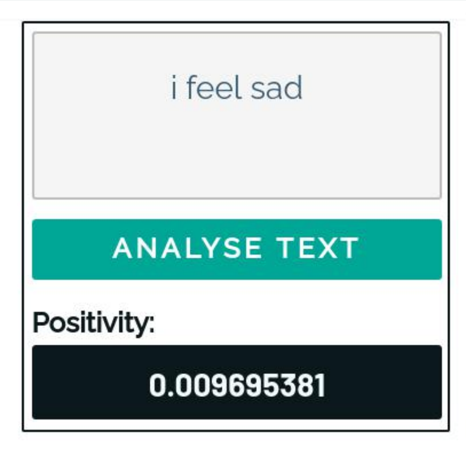
A key part of smart manufacturing and a modern factory approach involves real-time monitoring of machinery operating conditions.

What can be achieved?

- DL to detect malfunctioning machinery in real-time will lead to increased productivity and decreased costs.

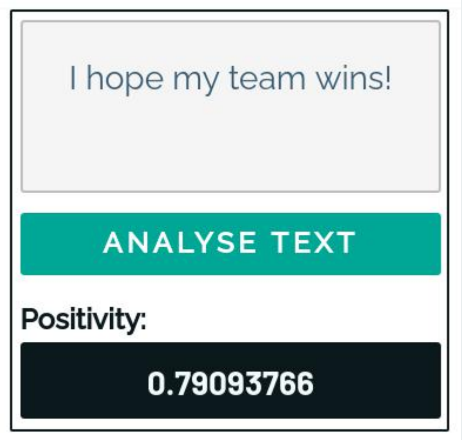
Data requirements

- Audio recordings
  - functioning or malfunctioning machinery.
- Microphones mounted in key parts of each machine.



i feel sad

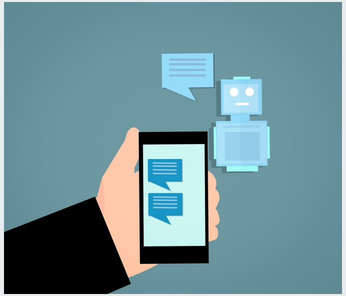
Positivity: 0.009695381



I hope my team wins!

Positivity: 0.79093766

Automated customer service phone calls and chatbots are becoming increasingly easy to interact with.



Source: https://peltarion.com/use-cases/customer-service-sentiment-analysis

Improving customer service through sentiment

The problem

- Frustration associated with bad experiences can have a significant impact on customer retention.

Opportunity for DL

- Natural language processing (NLP) are ideal for gaining insight into the user experience in customer service interactions.

Data requirements

- Text or audio from historical examples
  - successful and unsuccessful automated customer service interactions

Main researchers in Deep Learning

Samy Bengio	https://research.google.com/pubs/bengio.html
Yoshua Bengio	http://www.iro.umontreal.ca/~bengioy/yoshua_en/research.html
Thomas Dean	https://research.google.com/pubs/author189.html
Jeffrey Dean	https://research.google.com/pubs/jeff.html
Nando de Freitas	https://www.cs.ox.ac.uk/people/nando.defreitas/
Geoff Hilton	http://www.cs.toronto.edu/~hinton/
Yann LeCun	http://yann.lecun.com/
Andrew Ng	http://www.andrewng.org/
Quoc Le, Honglak Lee, Tommy Poggio, ...	

Resources

- Aurélien Géron. Hands-On Machine Learning with Scikit-Learn and TensorFlow. 2017 (☆☆☆☆☆)
- François Chollet. Deep Learning with Python. 2017 (☆☆☆☆)
  - Practitioner’s approach. Keras implementation per topic
- Ian Goodfellow and Yoshua Bengio and Aaron Courville. Deep Learning (Adaptive Computation and Machine Learning series). *2015* (☆☆)
  - Theoretical book. There is no code covered in the book.
- Michael Nielsen. Neural Networks and Deep Learning
  - Theory-based learning approach. Some code snippets.
- Gulli and Kapoor. TensorFlow Deep Learning Cookbook.
  - Lots of code and explanations of what the code is doing
- Adrian Rosebrock. Deep Learning for Computer Vision with Python.
- Sandro Skansi. Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence. 2018
- Andriy Burkov. The Hundred-Page Machine Learning Book.
  - All started because of challenge accepted
- Andrew Ng. Machine Learning Yearning: Technical strategy for AI engineers, in the era of Deep Learning.
- Yaser S. Abu-Mostafa, Malik Magdon-Ismail, Hsuan-Tien Lin. Learning from Data: A short course.
  - Supplement with lectures and videos.

Libraries for Deep Learning

Pandas

NumPy

NLTK

PyTorch

K Keras

TensorFlow

Caffe

scikit learn

theano

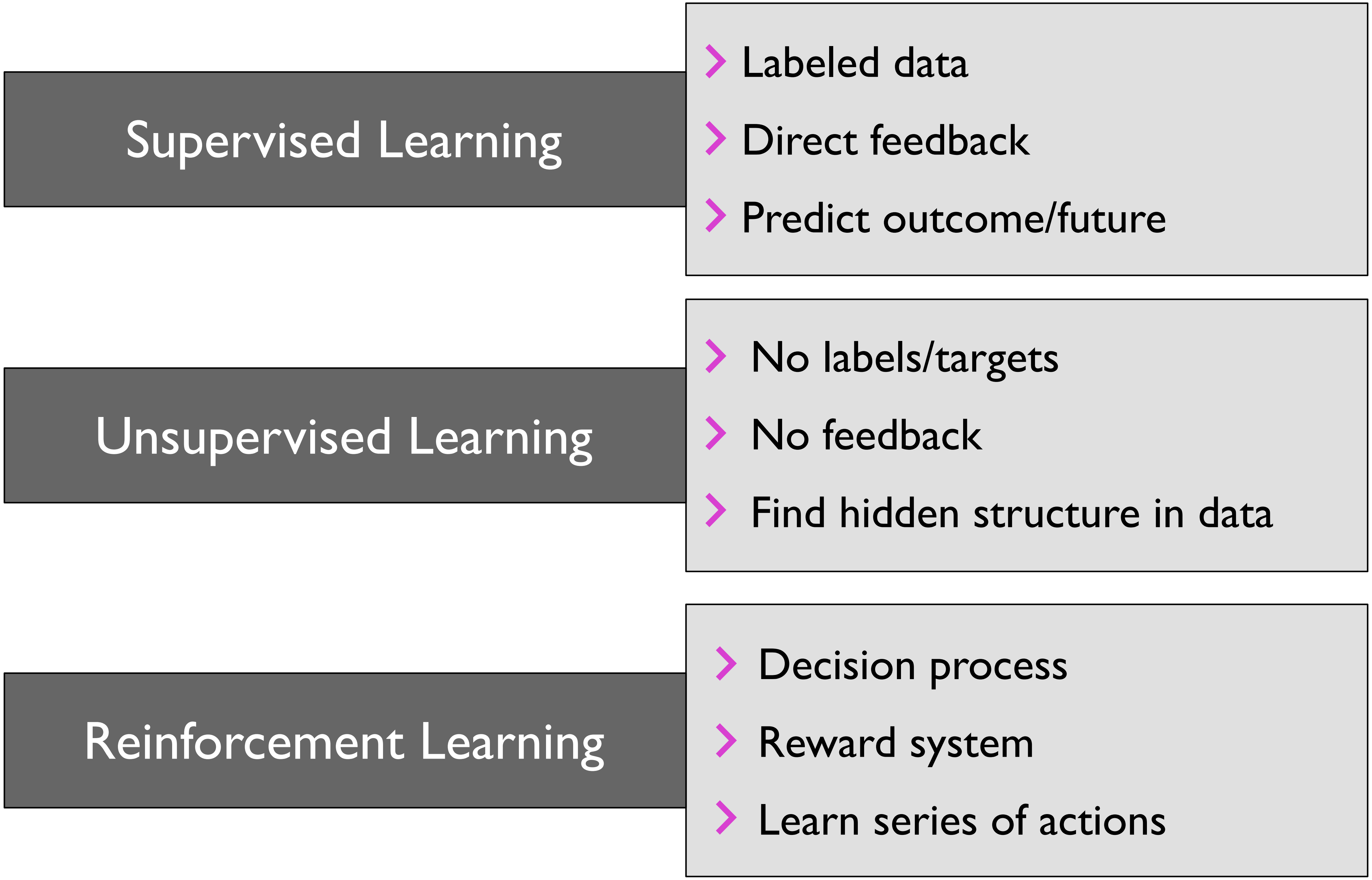
Lecture 01

What Are Machine Learning And Deep Learning? An Overview.

STAT 453: Introduction to Deep Learning and Generative Models
Spring 2020

Sebastian Raschka
http://stat.wisc.edu/~sraschka/teaching/stat453-ss2020/

The 3 Broad Categories Of ML (And DL)



Supervised Learning

- Labeled data
- Direct feedback
- Predict outcome/future

Unsupervised Learning

- No labels/targets
- No feedback
- Find hidden structure in data

Reinforcement Learning

- Decision process
- Reward system
- Learn series of actions

Source: Raschka and Mirjalily (2019). *Python Machine Learning, 3rd Edition*

Machine Learning Terminology and Notation

(Again, this also applies to DL)

1/5 -- What Is Machine Learning?
2/5 -- The 3 Broad Categories of ML
3/5 -- Machine Learning Terminology and Notation
4/5 -- Machine Learning Modeling Pipeline
5/5 --The Practical Aspects: Our Tools!

Machine Learning Jargon 1/2

- ***supervised learning:***
  learn function to map input *x* (features) to output *y* (targets)

- ***structured data:***
  databases, spreadsheets/csv files

- ***unstructured data:***
  features like image pixels, audio signals, text sentences
  (previous to DL, extensive feature engineering required)

Supervised Learning (More Formal Notation)

"training examples"

Training set: $\mathscr{D} = \{\langle \mathbf{x}^{[i]}, y^{[i]} \rangle, i = 1, \dots \ , n\},$

Unknown function: $f(\mathbf{x}) = y$

Hypothesis: $h(\mathbf{x}) = \hat{y}$

sometimes *t* or *o*

Classification

Regression

$h : \mathbb{R}^m \rightarrow \mathscr{Y}, \quad \mathscr{Y} = \{1,...,k\}$

$h : \mathbb{R}^m \rightarrow \mathbb{R}$

Data Representation

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

Feature vector

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix}$$

Design Matrix

$$\mathbf{X} = \begin{bmatrix} x_1^{[1]} & x_2^{[1]} & \cdots & x_m^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \cdots & x_m^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{[n]} & x_2^{[n]} & \cdots & x_m^{[n]} \end{bmatrix}$$

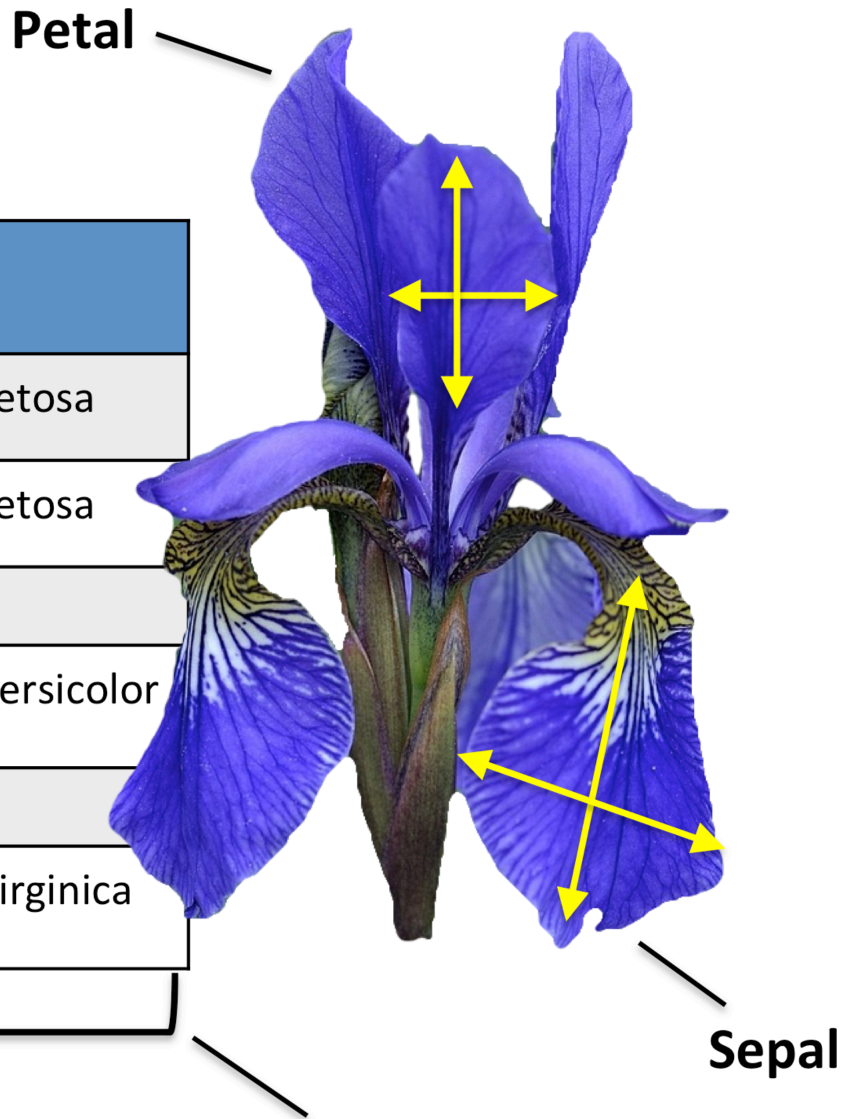
Design Matrix

Data Representation (structured data)

$m =$ ___

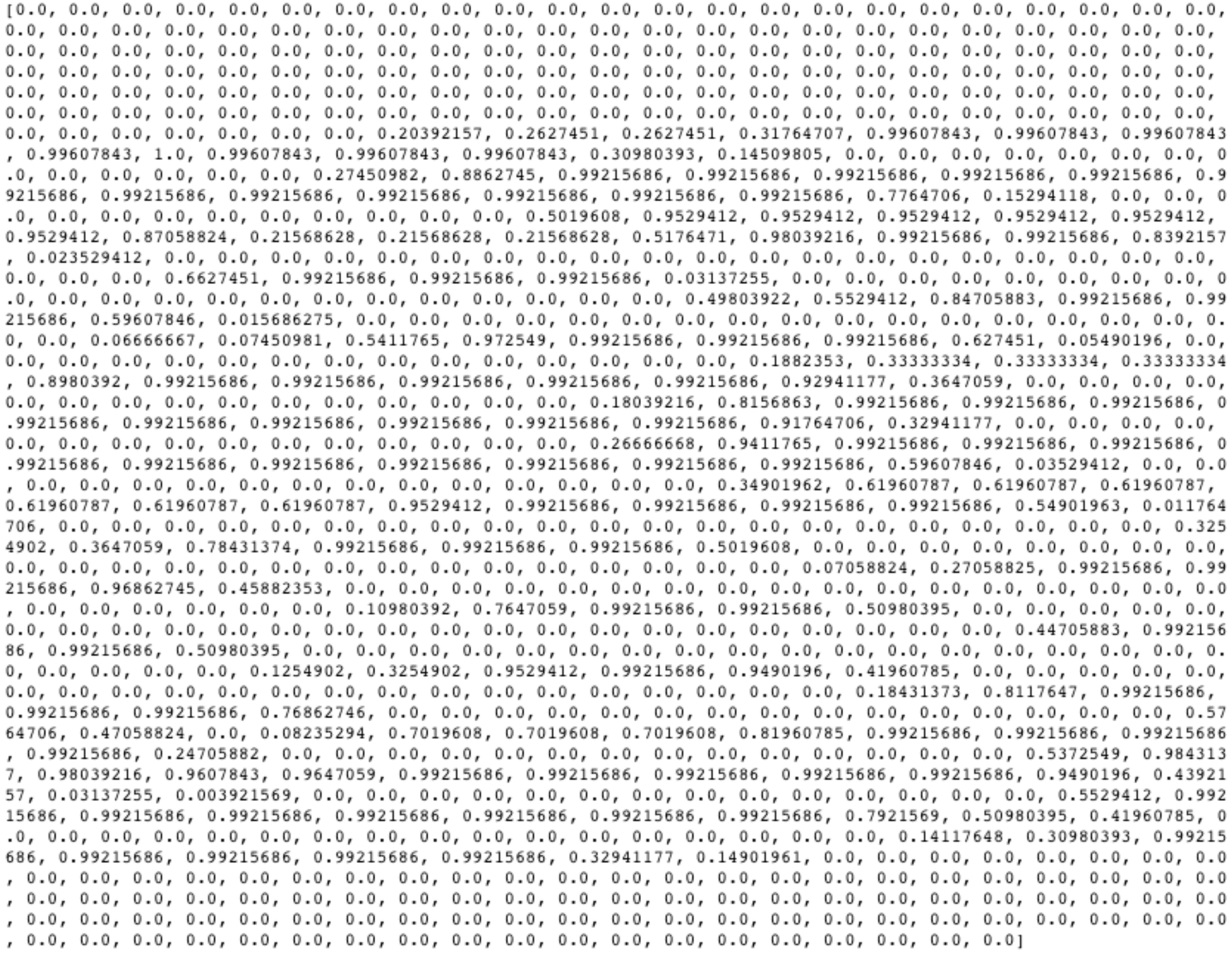
$n =$ ___

	Sepal length	Sepal width	Petal length	Petal width	
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
…					
50	6.4	3.5	4.5	1.2	Versicolor
…					
150	5.9	3.0	5.0	1.8	Virginica

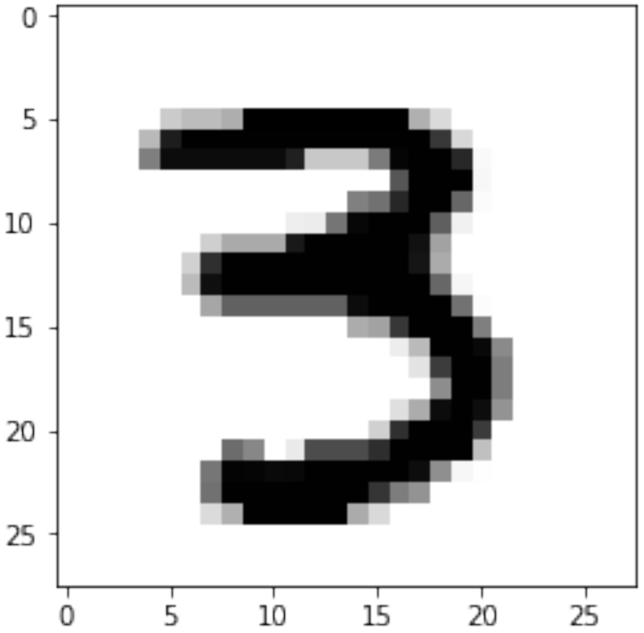


Data Representation (unstructured data; images)

"traditional methods"



```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.20392157, 0.2627451, 0.2627451, 0.31764707, 0.99607843, 0.99607843, 0.99607843
, 0.99607843, 1.0, 0.99607843, 0.99607843, 0.99607843, 0.30980393, 0.14509805, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.27450982, 0.8862745, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.9
9215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.7764706, 0.15294118, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5019608, 0.9529412, 0.9529412, 0.9529412, 0.9529412, 0.9529412,
0.9529412, 0.87058824, 0.21568628, 0.21568628, 0.21568628, 0.5176471, 0.98039216, 0.99215686, 0.99215686, 0.8392157
, 0.023529412, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.6627451, 0.99215686, 0.99215686, 0.99215686, 0.03137255, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.49803922, 0.5529412, 0.84705883, 0.99215686, 0.99
215686, 0.59607846, 0.015686275, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.06666667, 0.07450981, 0.5411765, 0.972549, 0.99215686, 0.99215686, 0.99215686, 0.627451, 0.05490196, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.1882353, 0.33333334, 0.33333334, 0.33333334
, 0.8980392, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.92941177, 0.3647059, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.18039216, 0.8156863, 0.99215686, 0.99215686, 0.99215686, 0
.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.91764706, 0.32941177, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.26666668, 0.9411765, 0.99215686, 0.99215686, 0.99215686, 0
.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.59607846, 0.03529412, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.34901962, 0.61960787, 0.61960787, 0.61960787,
0.61960787, 0.61960787, 0.61960787, 0.9529412, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.54901963, 0.011764
706, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.325
4902, 0.3647059, 0.78431374, 0.99215686, 0.99215686, 0.99215686, 0.5019608, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.07058824, 0.27058825, 0.99215686, 0.99
215686, 0.96862745, 0.45882353, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.10980392, 0.7647059, 0.99215686, 0.99215686, 0.50980395, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.44705883, 0.992156
86, 0.99215686, 0.50980395, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.1254902, 0.3254902, 0.9529412, 0.99215686, 0.9490196, 0.41960785, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.18431373, 0.8117647, 0.99215686,
0.99215686, 0.99215686, 0.76862746, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.57
64706, 0.47058824, 0.0, 0.08235294, 0.7019608, 0.7019608, 0.7019608, 0.81960785, 0.99215686, 0.99215686, 0.99215686
, 0.99215686, 0.24705882, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5372549, 0.984313
7, 0.98039216, 0.9607843, 0.9647059, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.9490196, 0.43921
57, 0.03137255, 0.003921569, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5529412, 0.992
15686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.7921569, 0.50980395, 0.41960785, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.14117648, 0.30980393, 0.99215
686, 0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.32941177, 0.14901961, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

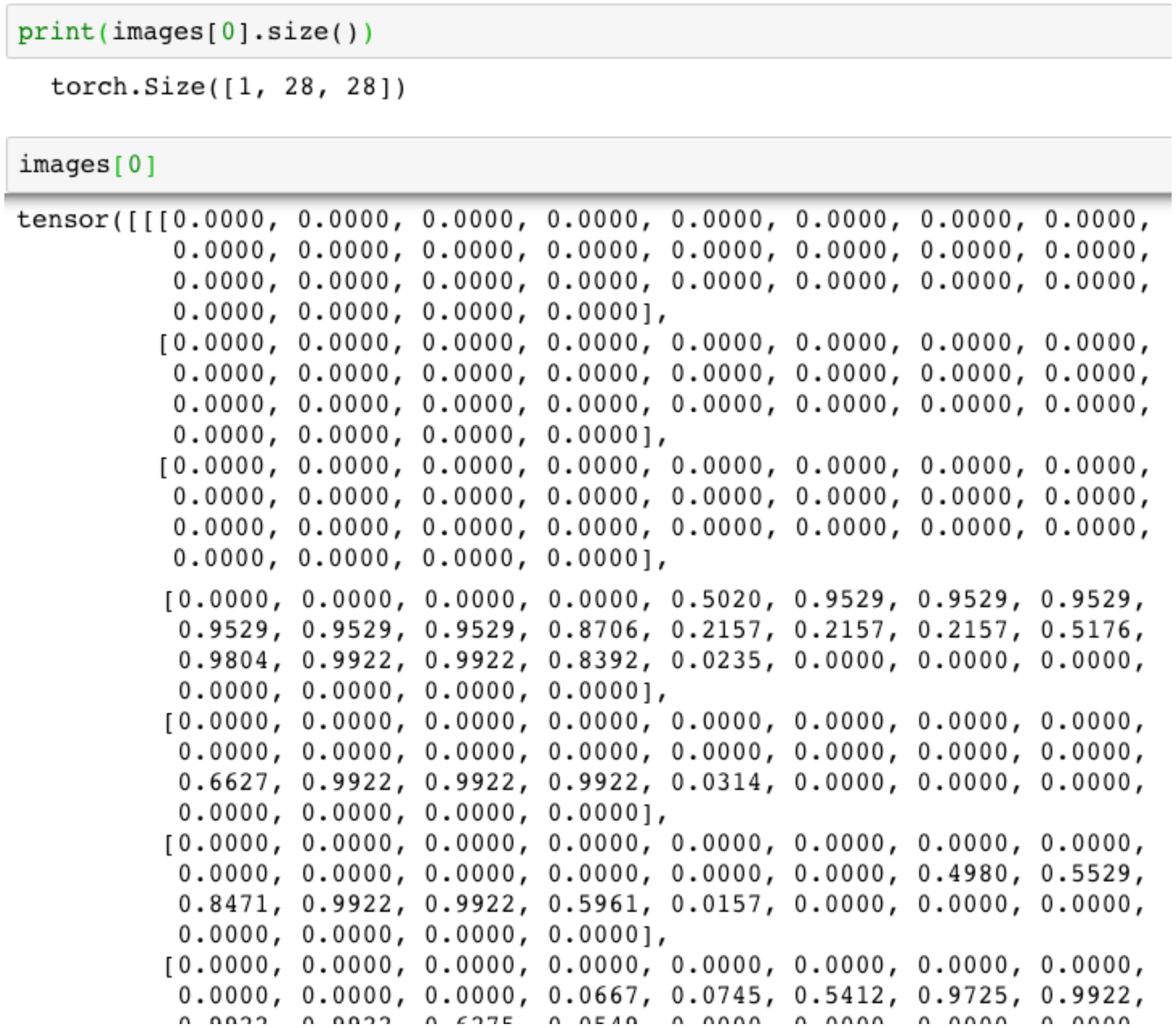


Data Representation (unstructured data; images)

Convolutional Neural Networks

```
Image batch dimensions: torch.Size([128, 1, 28, 28])
Image label dimensions: torch.Size([128])
```

"NCHW" representation (more on that later)

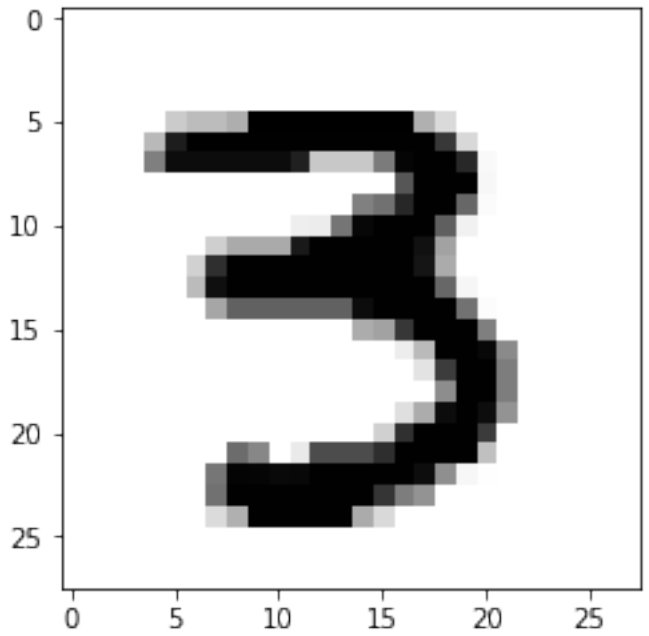


```
print(images[0].size())
```

```
  torch.Size([1, 28, 28])
```

```
images[0]
```

```
tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],

         [0.0000, 0.0000, 0.0000, 0.0000, 0.5020, 0.9529, 0.9529, 0.9529,
          0.9529, 0.9529, 0.9529, 0.8706, 0.2157, 0.2157, 0.2157, 0.5176,
          0.9804, 0.9922, 0.9922, 0.8392, 0.0235, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.6627, 0.9922, 0.9922, 0.9922, 0.0314, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.4980, 0.5529,
          0.8471, 0.9922, 0.9922, 0.5961, 0.0157, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0667, 0.0745, 0.5412, 0.9725, 0.9922,
          0.9922, 0.9922, 0.6275, 0.0549, 0.0000, 0.0000, 0.0000, 0.0000,
```



Machine Learning Jargon 2/2

- **Training example**, synonymous to
  observation, training record, training instance, training sample (in some contexts, sample refers to a collection of training examples)
- **Feature**, synonymous to
  predictor, variable, independent variable, input, attribute, covariate
- **Target**, synonymous to
  outcome, ground truth, output, response variable, dependent variable, (class) label (in classification)
- **Output / Prediction**, use this to distinguish from targets; here, means output from the model

- use loss L for a single training example
- use cost C for the average loss over the training set
- use $\phi(\cdot)$, unless noted otherwise, for the activation function

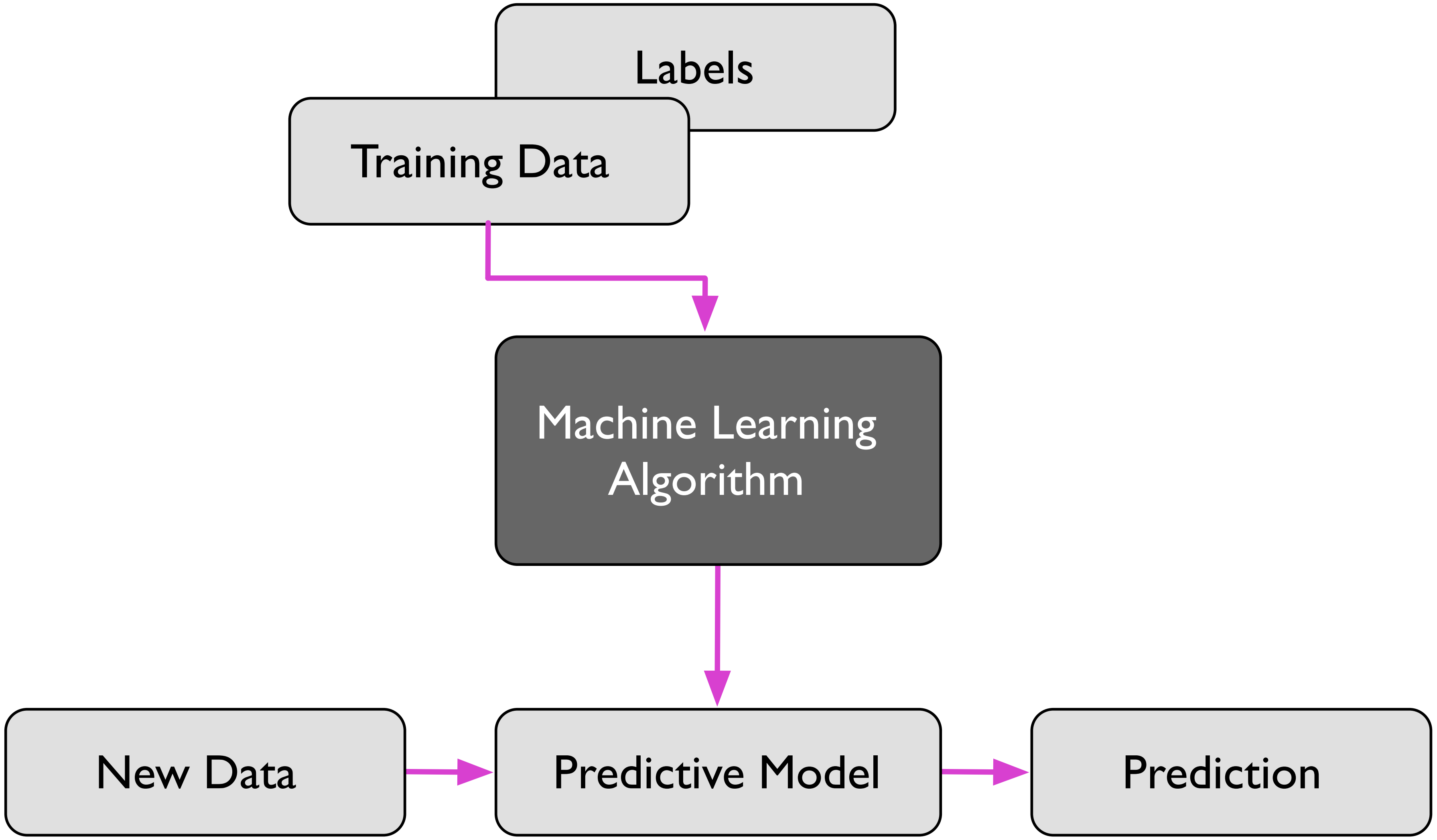
(will make more sense later)

Machine Learning Modeling Pipeline

(Like before, this also applies to DL)

1/5 -- What Is Machine Learning?
2/5 -- The 3 Broad Categories of ML
3/5 -- Machine Learning Terminology and Notation
4/5 -- Machine Learning Modeling Pipeline
5/5 --The Practical Aspects: Our Tools!

Supervised Learning Workflow


Labels

Training Data

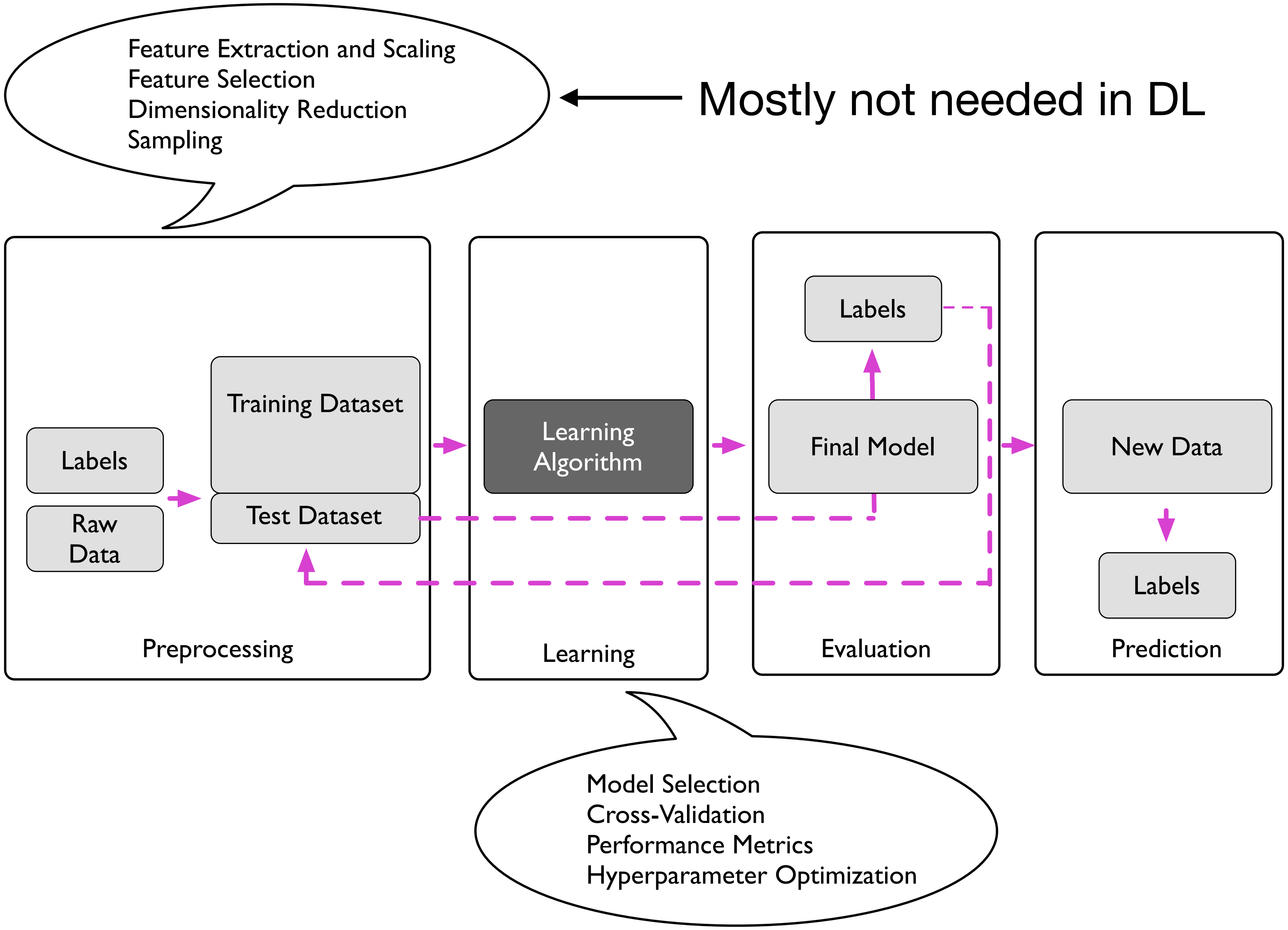
Machine Learning Algorithm

New Data

Predictive Model

Prediction

Supervised Learning Workflow (more detailed)



Feature Extraction and Scaling
Feature Selection
Dimensionality Reduction
Sampling

Mostly not needed in DL

Labels
Raw Data
Training Dataset
Test Dataset
Preprocessing

Learning Algorithm
Learning

Labels
Final Model
Evaluation

New Data
Labels
Prediction

Model Selection
Cross-Validation
Performance Metrics
Hyperparameter Optimization

Source: Raschka and Mirjalily (2019). *Python Machine Learning, 3rd Edition*

Lecture 05

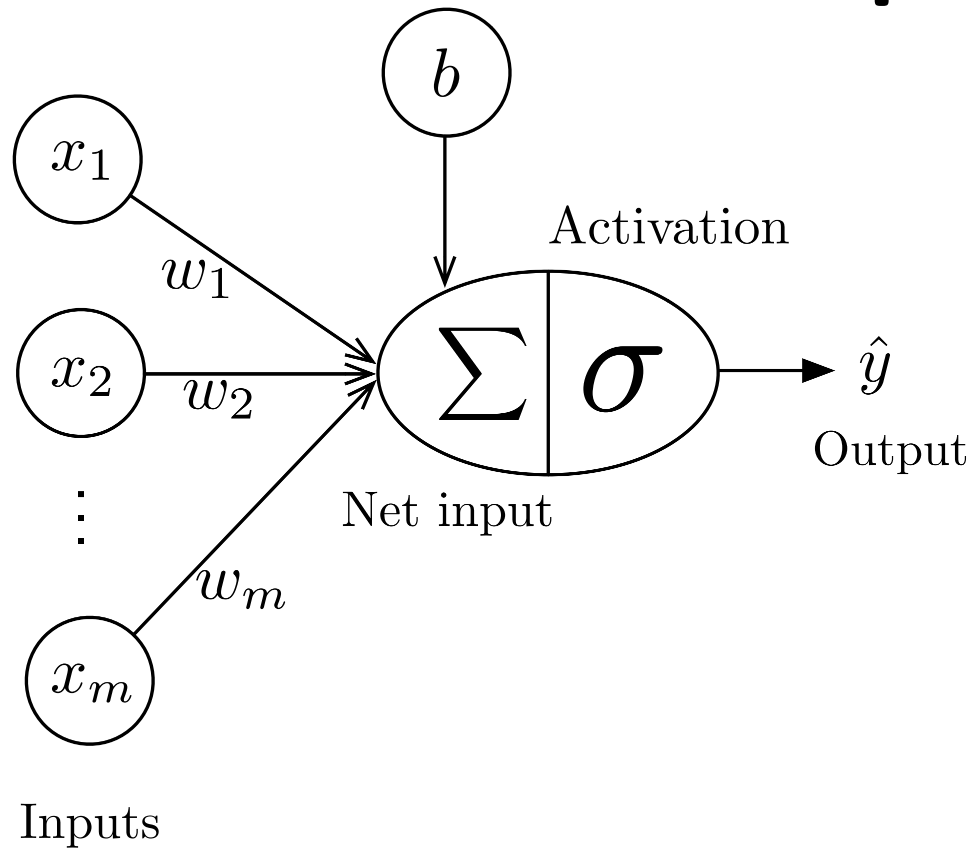
Fitting Neurons with Gradient Descent

STAT 453: Deep Learning, Spring 2020

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching/stat453-ss2020/

Perceptron Recap



$$\sigma\left(\sum_{i=1}^{m} x_i w_i + b\right) = \sigma\left(\mathbf{x}^T\mathbf{w} + b\right) = \hat{y}$$

$$\sigma(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$$

$$b = -\theta$$

Let $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0,1\})^n$

```
1.  Initialize  $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$, $\mathbf{b} := 0$
2.  For every training epoch:
    A.  For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
        (a)  $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T}\mathbf{w} + b)$  ← Compute output (prediction)
        (b)  $\mathrm{err} := (y^{[i]} - \hat{y}^{[i]})$  ← Calculate error
        (c)  $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$ , $b := b + err$  ← Update parameters
```

General Learning Principle

Let $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

"On-line" mode

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$, $\mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
        (a) Compute output (prediction)
        (b) Calculate error
        (c) Update $\mathbf{w}, b$
```

This applies to all common neuron models and (deep) neural network architectures!

There are some variants of it, namely the "batch mode" and the "minibatch mode" which we will briefly go over in the next slides and then discuss more later

General Learning Principle

Let $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

Minibatch mode

(mix between on-line and batch)

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
   A. Initialize $\Delta\mathbf{w} := 0, \ \Delta b := 0$
   B. For every $\{\langle \mathbf{x}^{[i]}, y^{[i]} \rangle, ..., \langle \mathbf{x}^{[i+k]}, y^{[i+k]} \rangle\} \subset \mathcal{D}$ :
      (a) Compute output (prediction)
      (b) Calculate error
      (c) Update $\Delta\mathbf{w}, \Delta b$
   C. Update $\mathbf{w}, b$ :
         $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, b := +\Delta b$
```

Most commonly used in DL, because

1. Choosing a subset (vs 1 example at a time) takes advantage of vectorization (faster iteration through epoch than on-line)

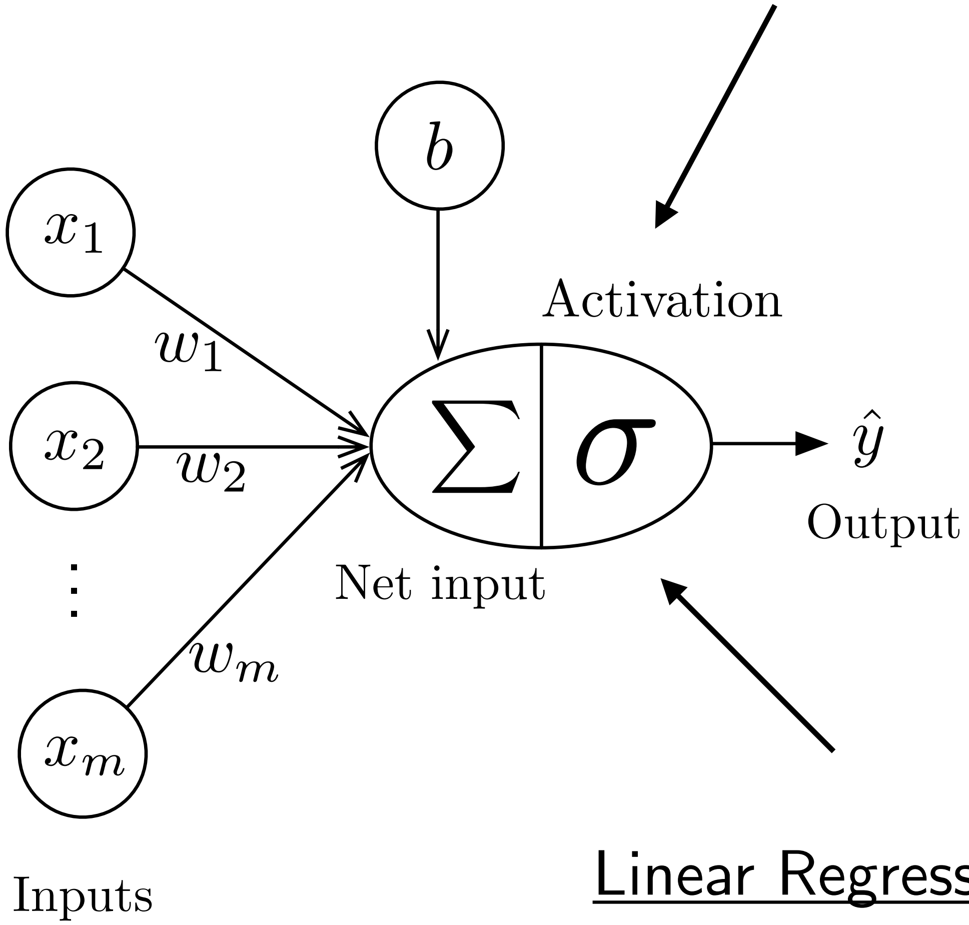
2. having fewer updates than "on-line" makes updates less noisy

3. makes more updates/ epoch than "batch" and is thus faster

Linear Regression

Perceptron: Activation function is the threshold function

The output is a binary label $\hat{y} \in \{0, 1\}$



You can think of linear regression as a linear neuron!

Linear Regression: Activation function is the identity function

$$\sigma(x) = x$$

The output is a real number $\hat{y} \in \mathbb{R}$

(Least-Squares) Linear Regression iteratively

- A very naive way to fit a linear regression model (and any neural net) is to start with initializing the parameters to 0's or small random values
- Then, for *k* rounds
  - Choose another random set of weights
  - If the model performs better, keep those weights
  - If the model performs worse, discard the weights

There's a better way!

- We will analyze what effect a change of a parameter has on the predictive performance (loss) of the model
  then, we change the weight a little bit in the direction that improves the performance (minimizes the loss) the most
- We do this in several (small) steps until the loss does not further decrease

(Least-Squares) Linear Regression

The update rule turns out to be this:

"On-line" mode

Perceptron learning rule

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
        (a) $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T}\mathbf{w} + b)$
        (b) $\mathrm{err} := (y^{[i]} - \hat{y}^{[i]})$
        (c) $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$
            $b := b + err$
```

Stochastic gradient descent

```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m, \mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
        (a) $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T}\mathbf{w} + b)$
        (b) $\nabla_{\mathbf{w}}\mathcal{L} = (y^{[i]} - \hat{y}^{[i]})\mathbf{x}^{[i]}$
            $\nabla_{b}\mathcal{L} = (y^{[i]} - \hat{y}^{[i]})$
        (c) $\mathbf{w} := \mathbf{w} + \eta \times (-\nabla_{\mathbf{w}}\mathcal{L})$
            $b := b + \underset{\nwarrow \text{ learning rate}}{\eta} \times \underbrace{(-\nabla_{b}\mathcal{L})}_{\text{negative gradient}}$
```

(Least-Squares) Linear Regression

The update rule turns out to be this:

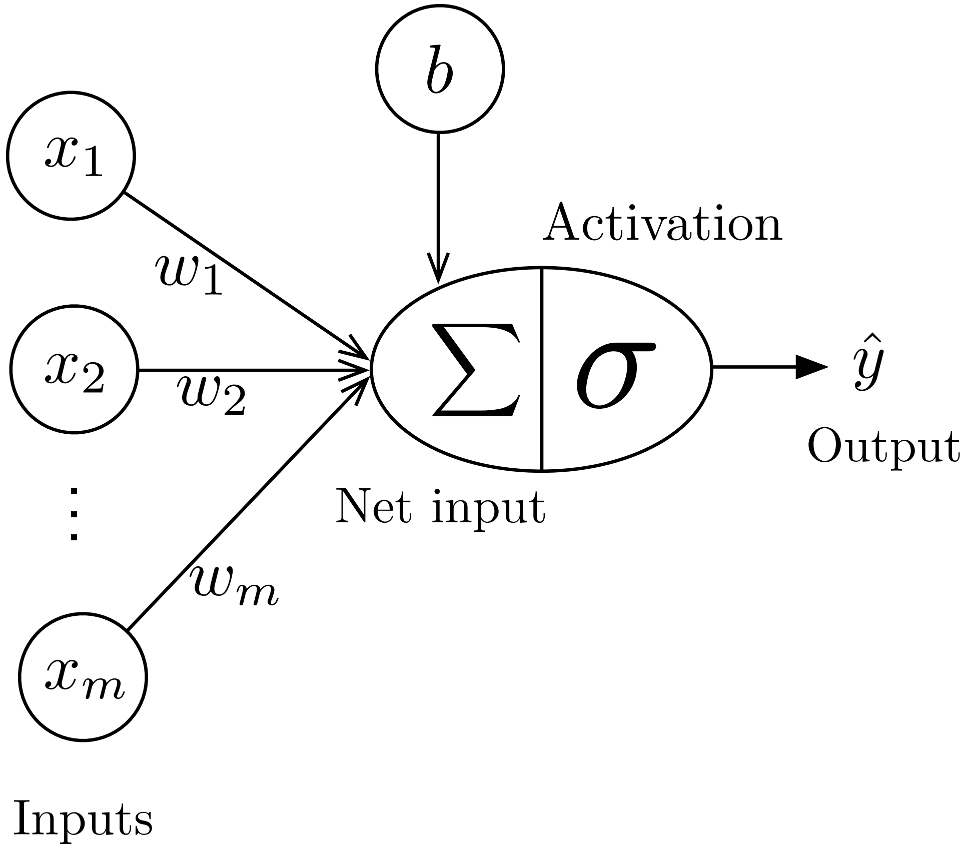
"On-line" mode

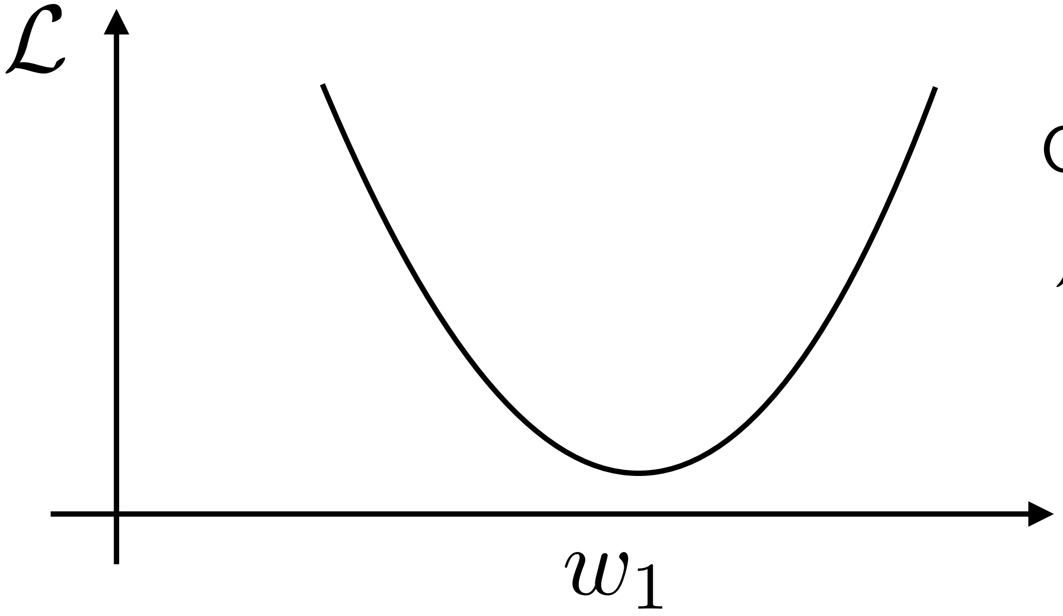
```
1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$, $\mathbf{b} := 0$
2. For every training epoch:
    A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
        (a) $\hat{y}^{[i]} := \sigma\big(\mathbf{x}^{[i]T}\mathbf{w} + b\big)$
    B. For weight $j$ in $\{1, ..., m\}$:
        (b) $\frac{\partial \mathcal{L}}{\partial w_j} = \big(y^{[i]} - \hat{y}^{[i]}\big)x_j^{[i]}$
        (c) $w_j := w_j + \boxed{\eta \times (-\frac{\partial \mathcal{L}}{\partial w_j})}$
    C. $\frac{\partial \mathcal{L}}{\partial b} = \big(y^{[i]} - \hat{y}^{[i]}\big)$
       $b := b + \boxed{\eta \times (-\frac{\partial \mathcal{L}}{\partial b})}$
```

Coincidentally, this appears almost to be the same as the perceptron rule, except that the prediction is a real number and we have a learning rate

This learning rule (from the previous slide) is called (stochastic) gradient descent. So, how did we get there?

Back to Linear Regression



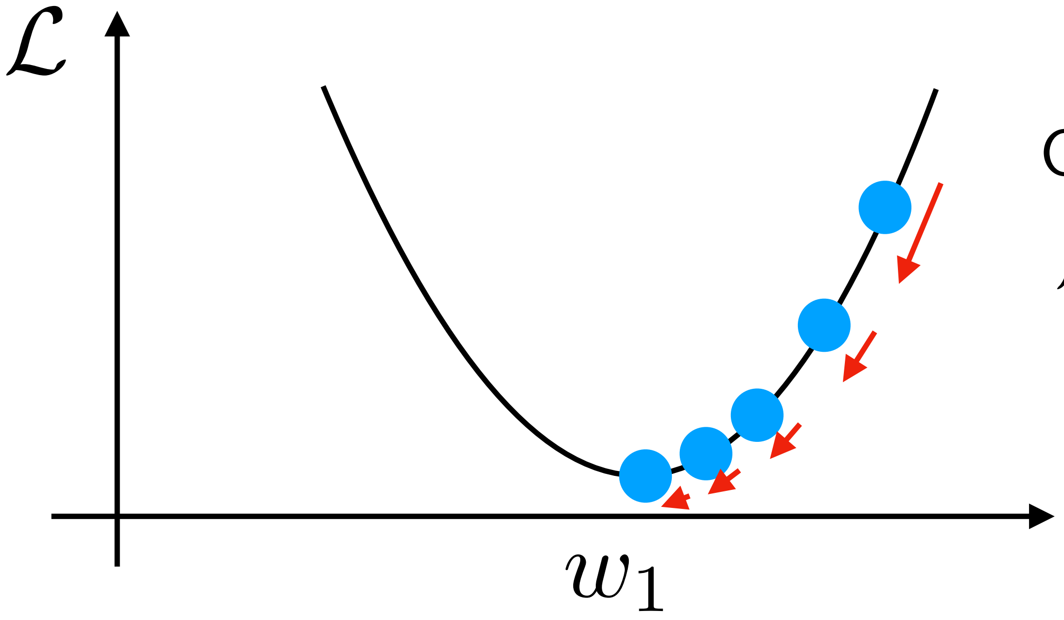


Convex loss function

$$\mathcal{L}(\mathbf{w}, b) = \sum_i (\hat{y}^{[i]} - y^{[i]})^2$$

Gradient Descent

Learning rate and steepness of the gradient determine how much we update

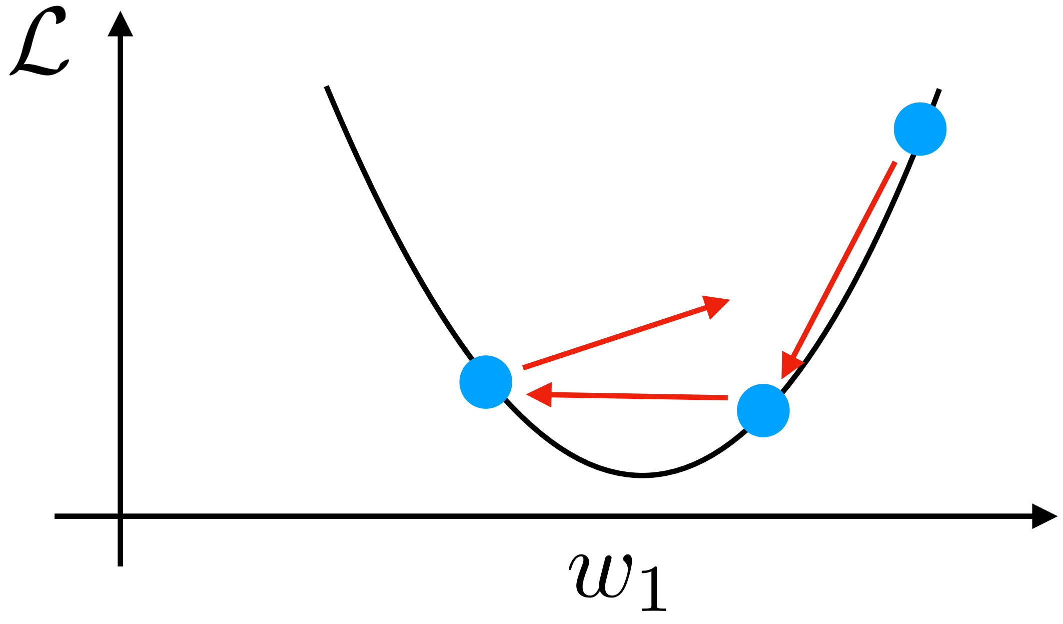


Convex loss function

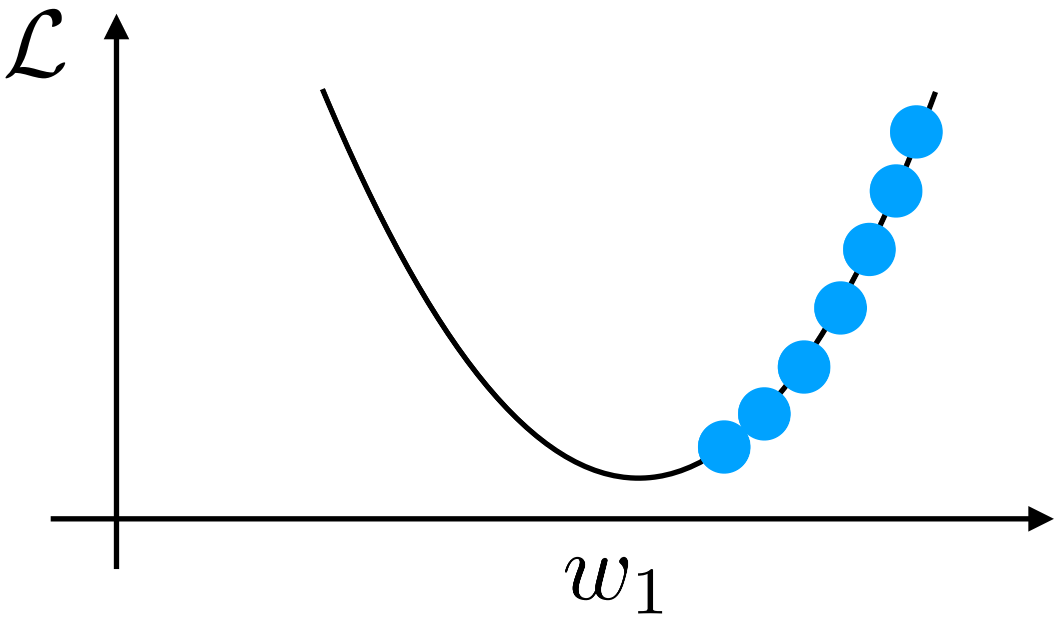
$$\mathcal{L}(\mathbf{w}, b) = \sum_i (\hat{y}^{[i]} - y^{[i]})^2$$

Gradient Descent

If the learning rate is too large,
we can overshoot



If the learning rate is too small,
convergence is very slow



Linear Regression Loss Derivative

$$\mathcal{L}(\mathbf{w}, b) = \sum_i (\hat{y}^{[i]} - y^{[i]})^2$$

Sum Squared Error (SSE) loss

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} \sum_i (\hat{y}^{[i]} - y^{[i]})^2 \\
&= \frac{\partial}{\partial w_j} \sum_i (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]})^2 \\
&= \sum_i 2(\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{\partial}{\partial w_j} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \\
&= \sum_i 2(\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^T \mathbf{x}^{[i]})} \frac{\partial}{\partial w_j} \mathbf{w}^T \mathbf{x}^{[i]} \\
&= \sum_i 2(\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^T \mathbf{x}^{[i]})} x_j^{[i]} \\
&= \sum_i 2(\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) x_j^{[i]}
\end{aligned}$$

(Note that the activation function is the identity function in linear regression)

Linear Regression Loss Derivative (alt.)

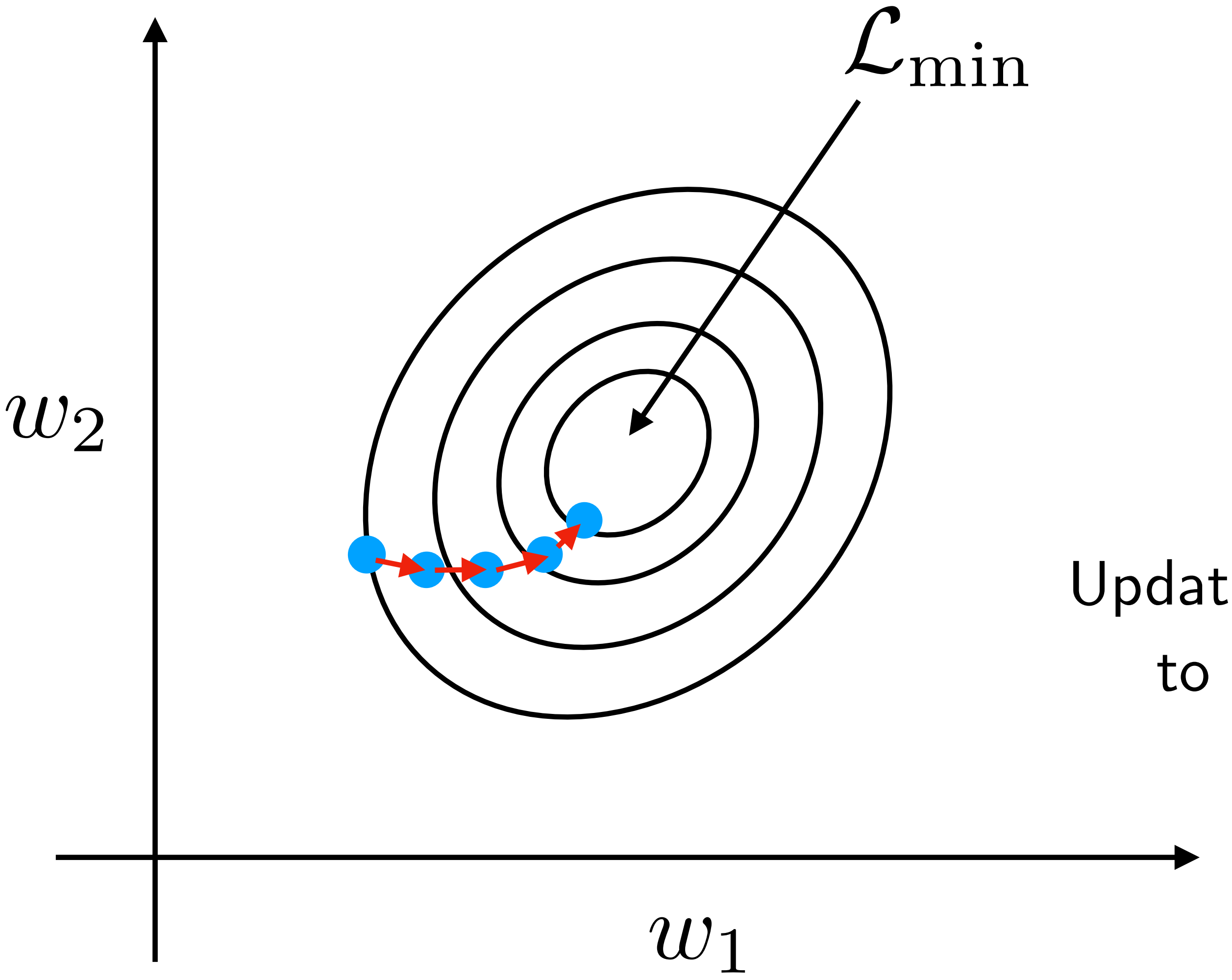
$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2n} \sum_i (\hat{y}^{[i]} - y^{[i]})^2$$

Mean Squared Error (MSE) loss often scaled by factor 1/2 for convenience

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2n} \sum_i (\hat{y}^{[i]} - y^{[i]})^2 \\
&= \frac{\partial}{\partial w_j} \sum_i \frac{1}{2n} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]})^2 \\
&= \sum_i \frac{1}{n} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{\partial}{\partial w_j} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \\
&= \frac{1}{n} \sum_i (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^T \mathbf{x}^{[i]})} \frac{\partial}{\partial w_j} \mathbf{w}^T \mathbf{x}^{[i]} \\
&= \frac{1}{n} \sum_i (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^T \mathbf{x}^{[i]})} x_j^{[i]} \\
&= \frac{1}{n} \sum_i (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) x_j^{[i]}
\end{aligned}$$

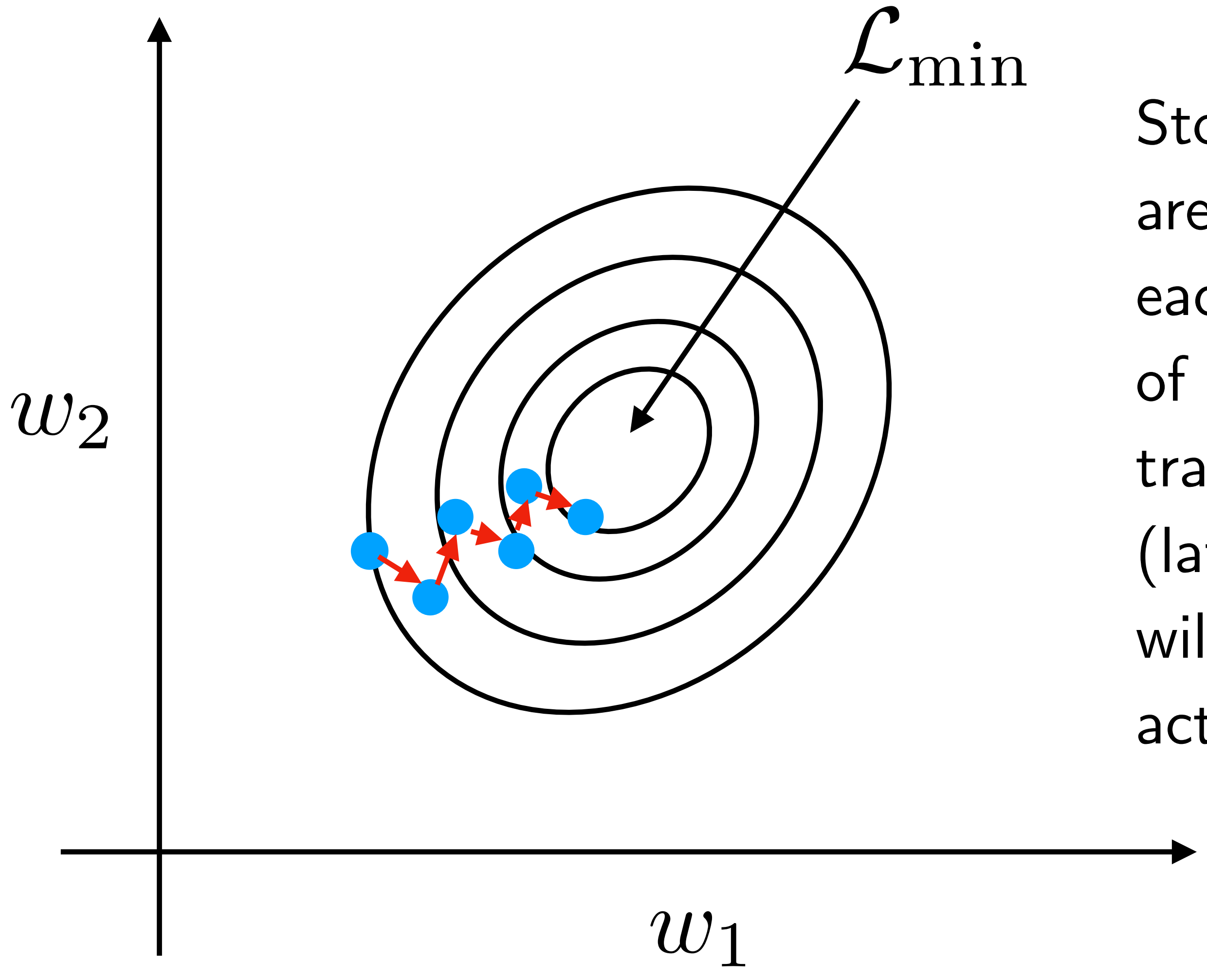
(Note that the activation function is the identity function in linear regression)

Batch Gradient Descent as Surface Plot



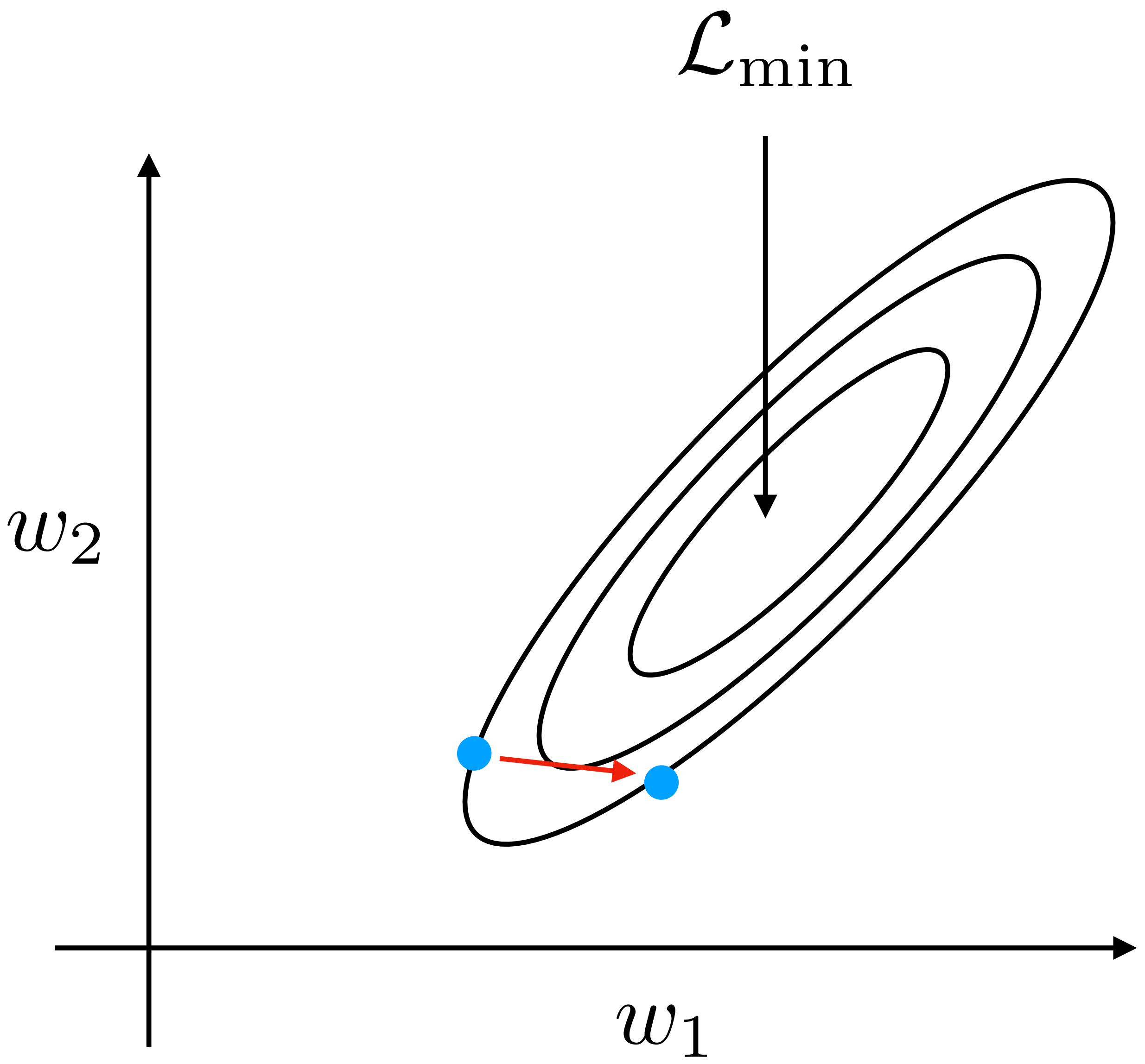
Updates perpendicular to contour lines

Stochastic Gradient Descent as Surface Plot



Stochastic updates
are a bit noisier, because
each batch is an approximation
of the overall loss on the
training set
(later, in deep neural nets, we
will see why noisier updates are
actually helpful)

Batch Gradient Descent as Surface Plot

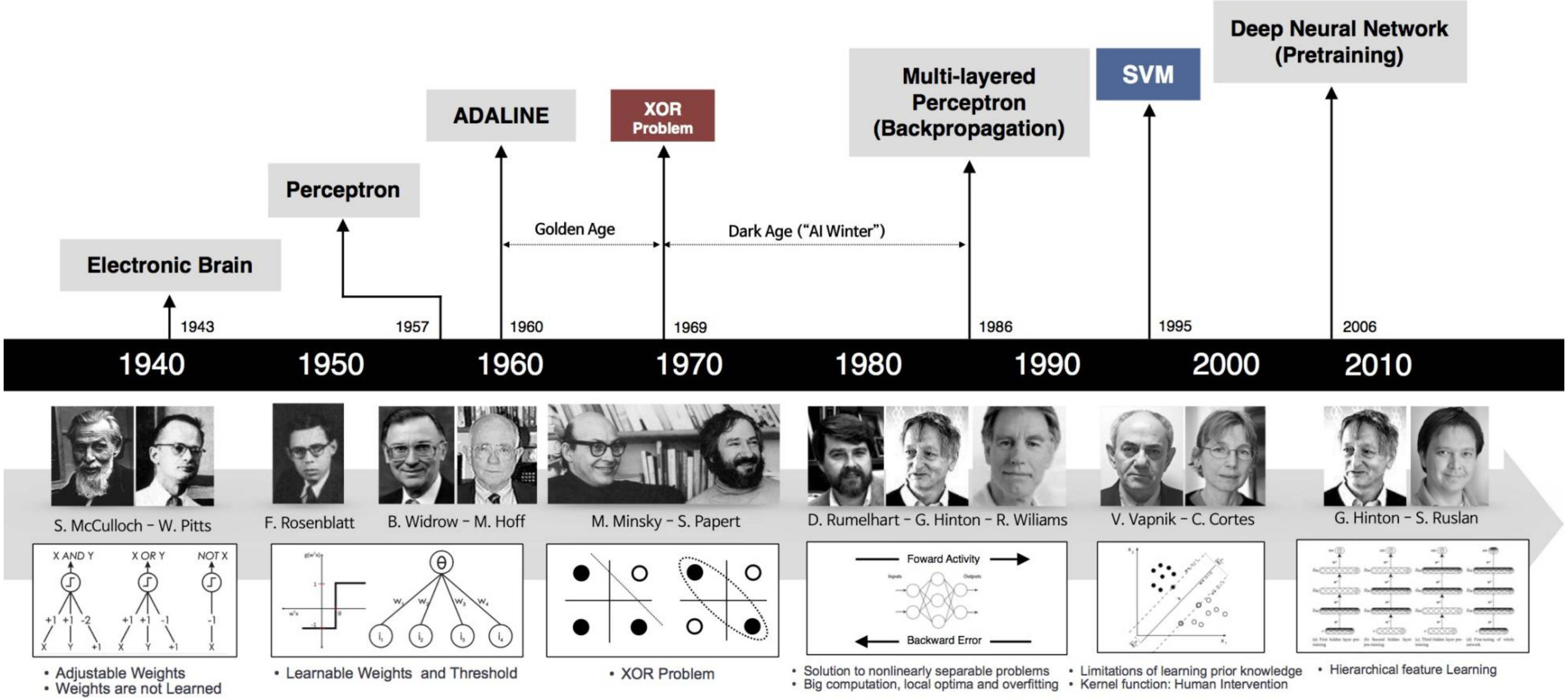


If inputs are on very different scales some weights will update more than others ... and it will also harm convergence (always normalize inputs!)

Multilayer Perceptrons

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Marcos Históricos:



Electronic Brain — 1943

Perceptron — 1957

ADALINE — 1960

XOR Problem — 1969

Multi-layered Perceptron (Backpropagation) — 1986

SVM — 1995

Deep Neural Network (Pretraining) — 2006

Golden Age

Dark Age (“AI Winter”)

1940 1950 1960 1970 1980 1990 2000 2010

S. McCulloch – W. Pitts

- Adjustable Weights
- Weights are not Learned

F. Rosenblatt

B. Widrow – M. Hoff

- Learnable Weights and Threshold

M. Minsky – S. Papert

- XOR Problem

D. Rumelhart – G. Hinton – R. Wiliams

- Solution to nonlinearly separable problems
- Big computation, local optima and overfitting

V. Vapnik – C. Cortes

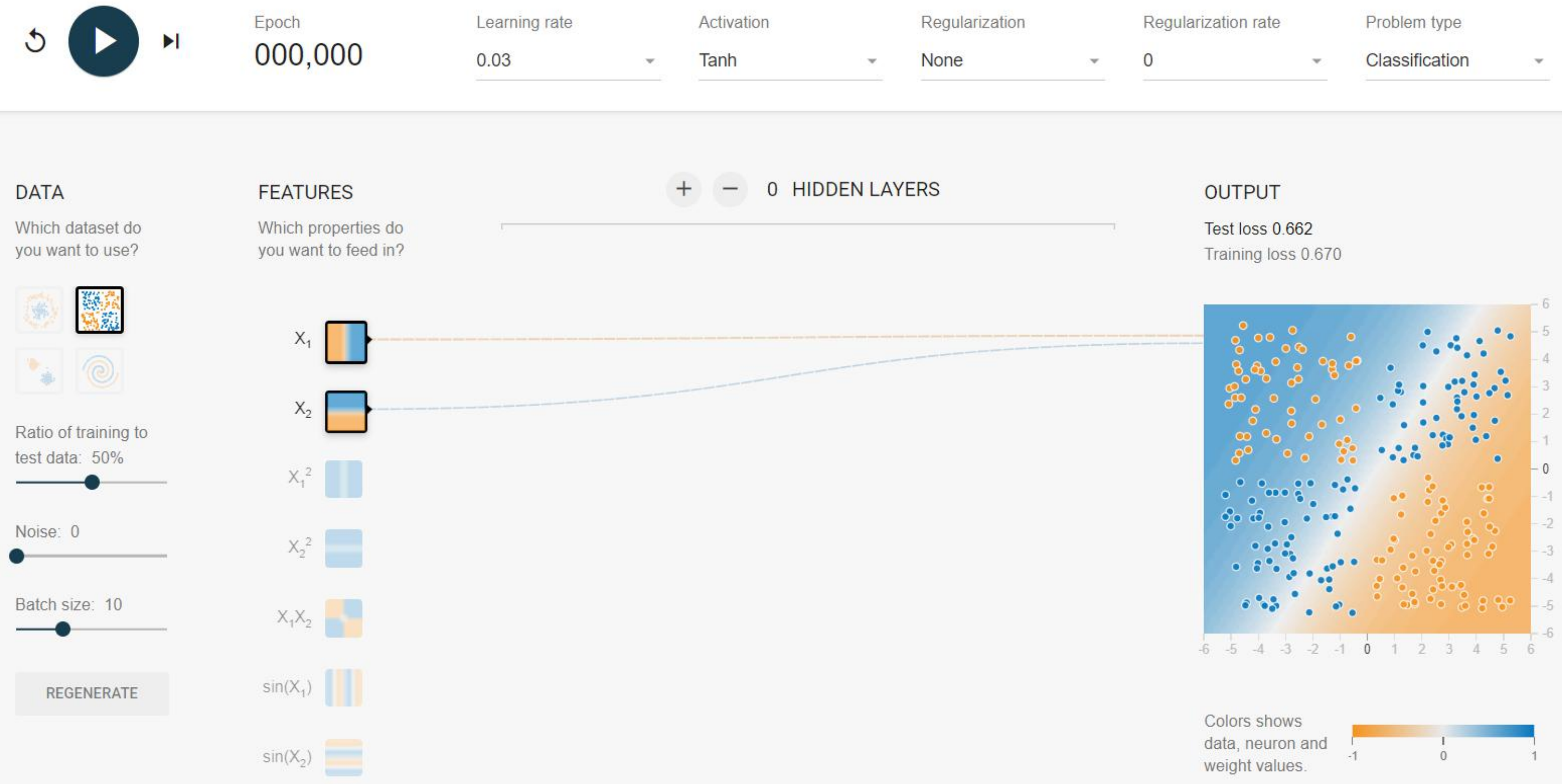
- Limitations of learning prior knowledge
- Kernel function: Human Intervention

G. Hinton – S. Ruslan

- Hierarchical feature Learning

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

With 1 layer and 1 neuron



Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

+ − 0 HIDDEN LAYERS

OUTPUT

Test loss 0.662

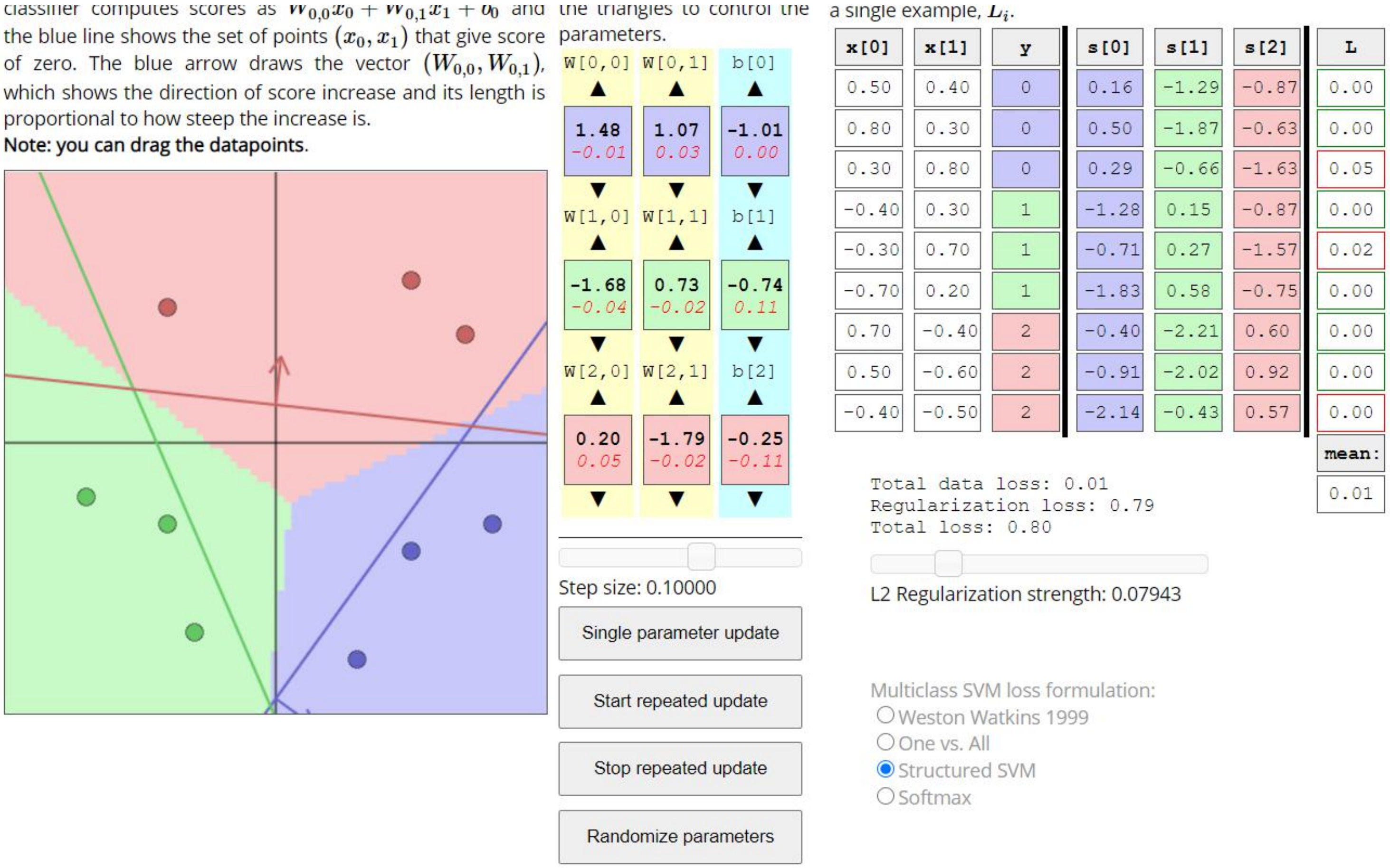
Training loss 0.670

Colors shows data, neuron and weight values.

http://playground.tensorflow.org/

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

With 1 layer and N neuron



classifier computes scores as $W_{0,0}x_0 + W_{0,1}x_1 + b_0$ and the blue line shows the set of points (x_0, x_1) that give score of zero. The blue arrow draws the vector $(W_{0,0}, W_{0,1})$, which shows the direction of score increase and its length is proportional to how steep the increase is.

Note: you can drag the datapoints.

the triangles to control the parameters.

W[0,0]	W[0,1]	b[0]
1.48 -0.01	1.07 0.03	-1.01 0.00
W[1,0]	W[1,1]	b[1]
-1.68 -0.04	0.73 -0.02	-0.74 0.11
W[2,0]	W[2,1]	b[2]
0.20 0.05	-1.79 -0.02	-0.25 -0.11

Step size: 0.10000

a single example, L_i.

x[0]	x[1]	y	s[0]	s[1]	s[2]	L
0.50	0.40	0	0.16	-1.29	-0.87	0.00
0.80	0.30	0	0.50	-1.87	-0.63	0.00
0.30	0.80	0	0.29	-0.66	-1.63	0.05
-0.40	0.30	1	-1.28	0.15	-0.87	0.00
-0.30	0.70	1	-0.71	0.27	-1.57	0.02
-0.70	0.20	1	-1.83	0.58	-0.75	0.00
0.70	-0.40	2	-0.40	-2.21	0.60	0.00
0.50	-0.60	2	-0.91	-2.02	0.92	0.00
-0.40	-0.50	2	-2.14	-0.43	0.57	0.00
						mean:
						0.01

```
Total data loss: 0.01
Regularization loss: 0.79
Total loss: 0.80
```

L2 Regularization strength: 0.07943

Multiclass SVM loss formulation:

○ Weston Watkins 1999
○ One vs. All
● Structured SVM
○ Softmax

http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Multi-Layer Perceptron

- O grande desafio foi achar um algoritmo de aprendizado para atualizar dos pesos das camadas intermediarias

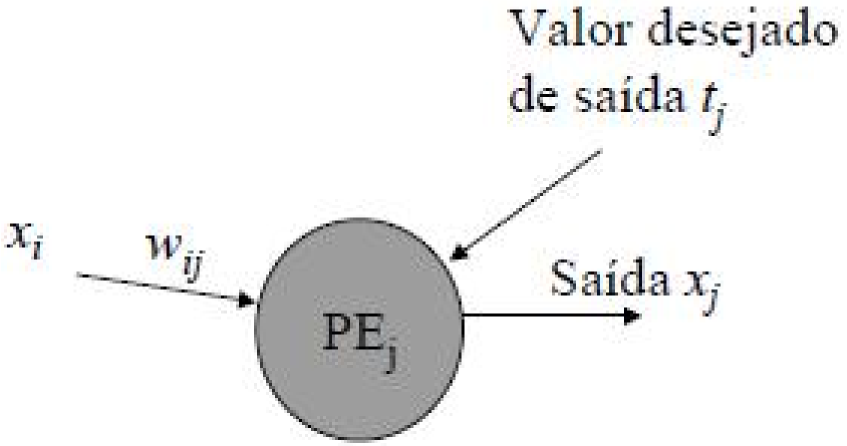
- Idéia Central

  Os erros dos elementos processadores da camada de saída (conhecidos pelo treinamento supervisionado) são **retro-propagados**s para as camadas intermediarias

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

- Processador j pertence à Camada de Saída:

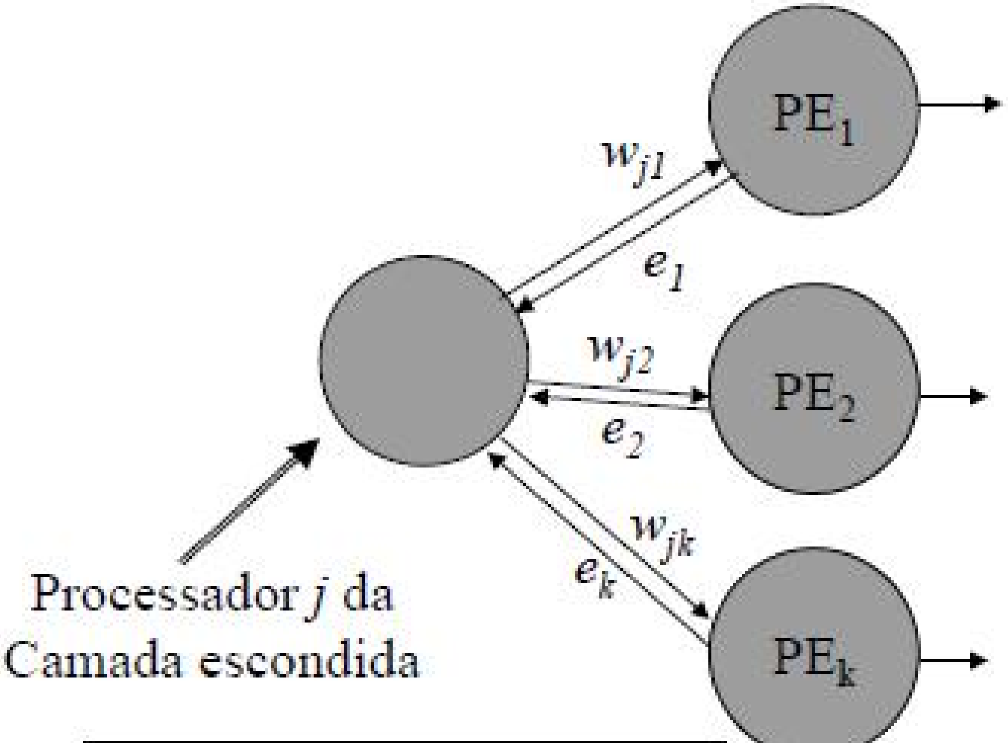


$$\boxed{e_j = (t_j - x_j)F'(y_j)}$$

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

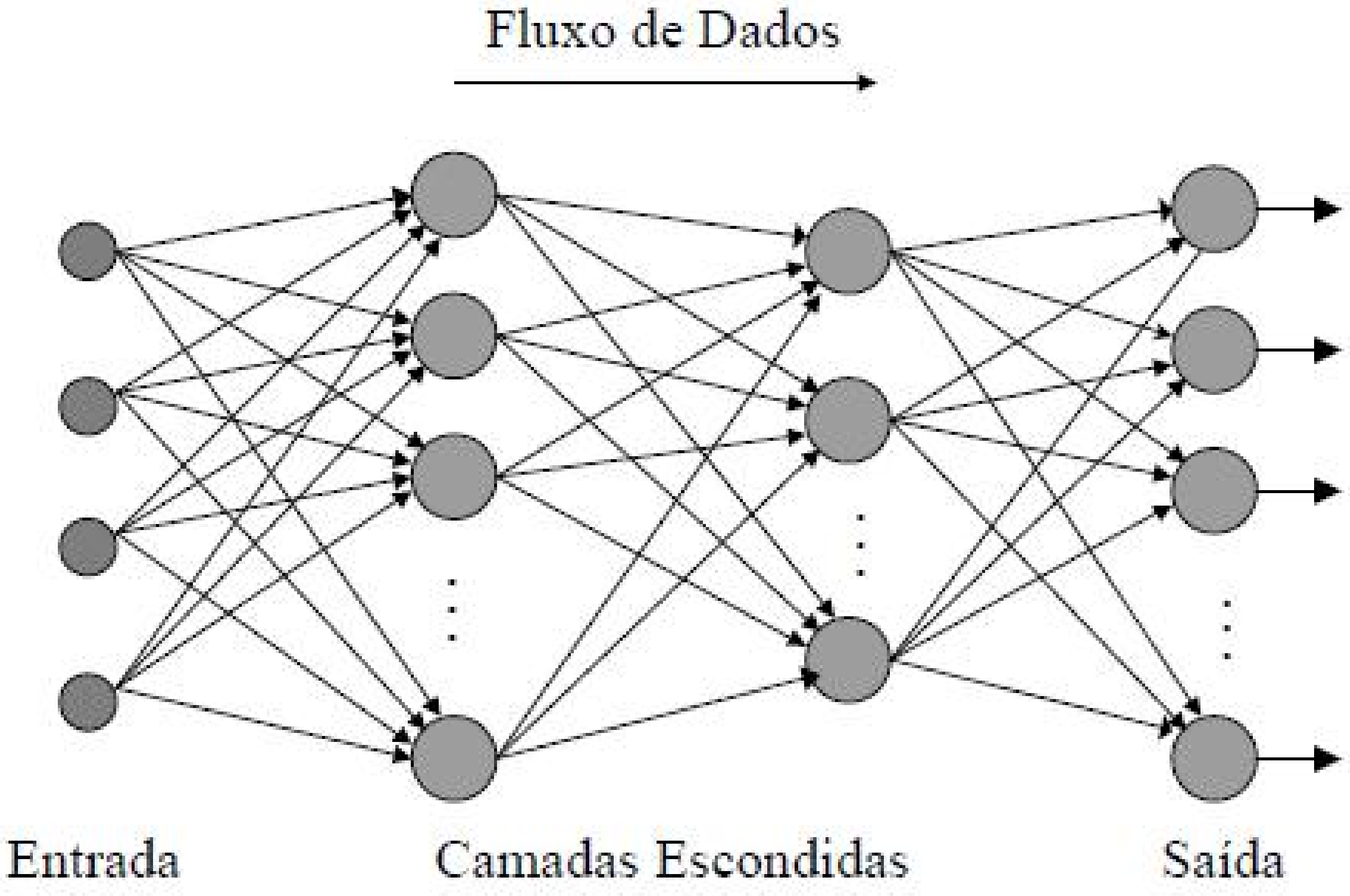
- Processador *j* pertence à Camada Escondida:



$$\boxed{e_j = F'(v_j)\sum_k (e_k w_{jk})}$$

Processo de aprendizado

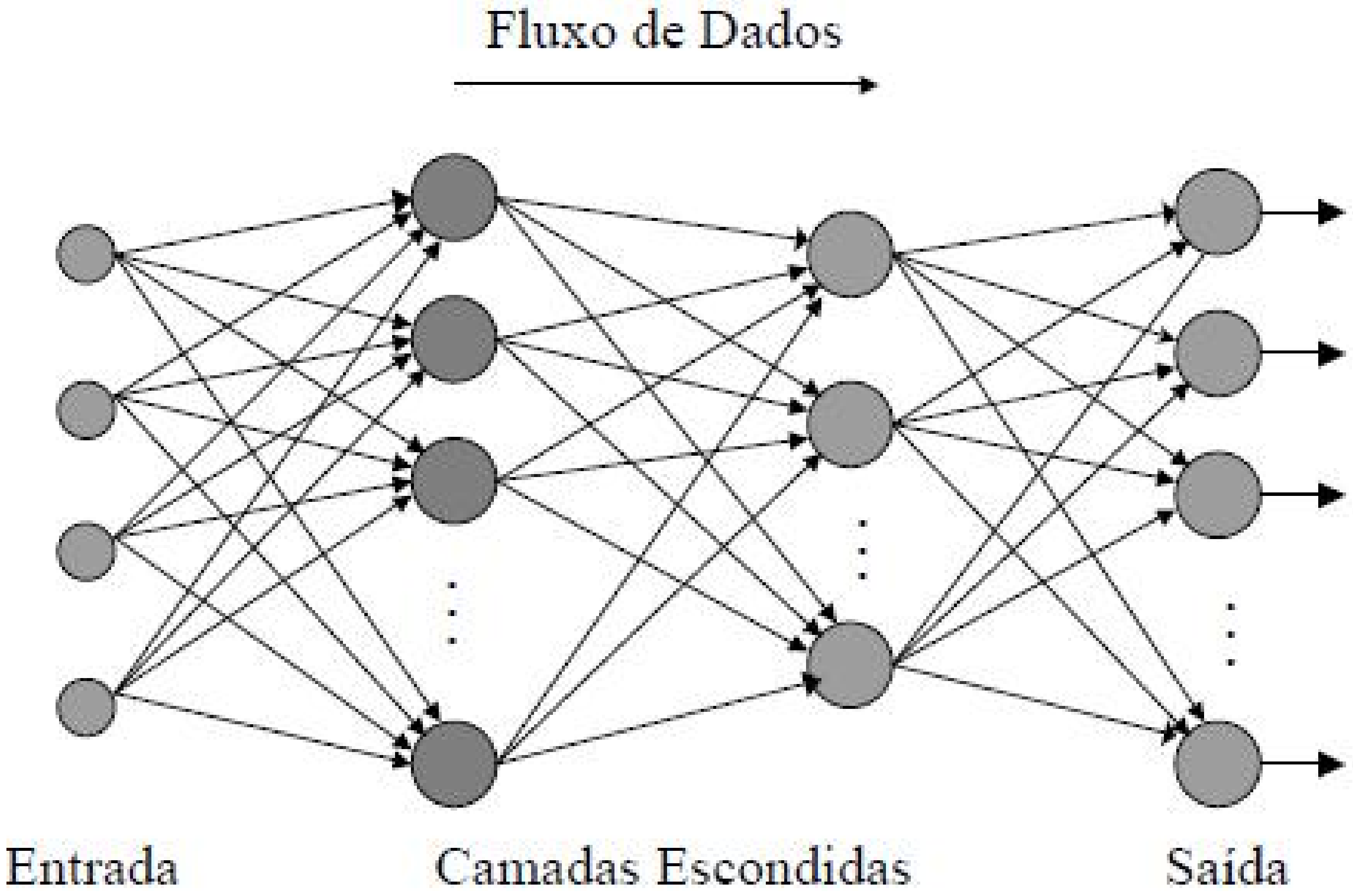
- Fase 1: Feed-Forward



Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

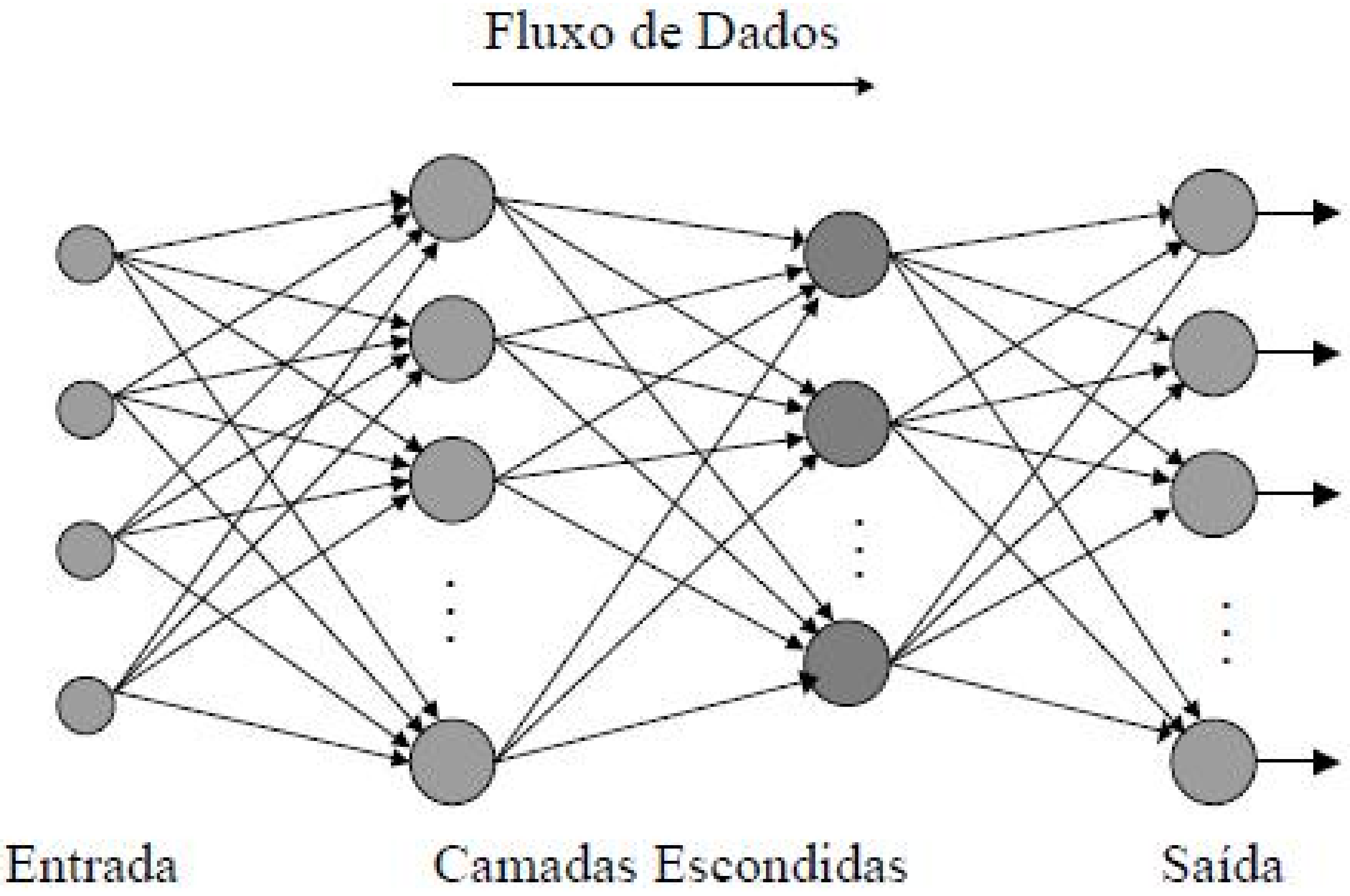
- Fase 1: Feed-Forward



Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

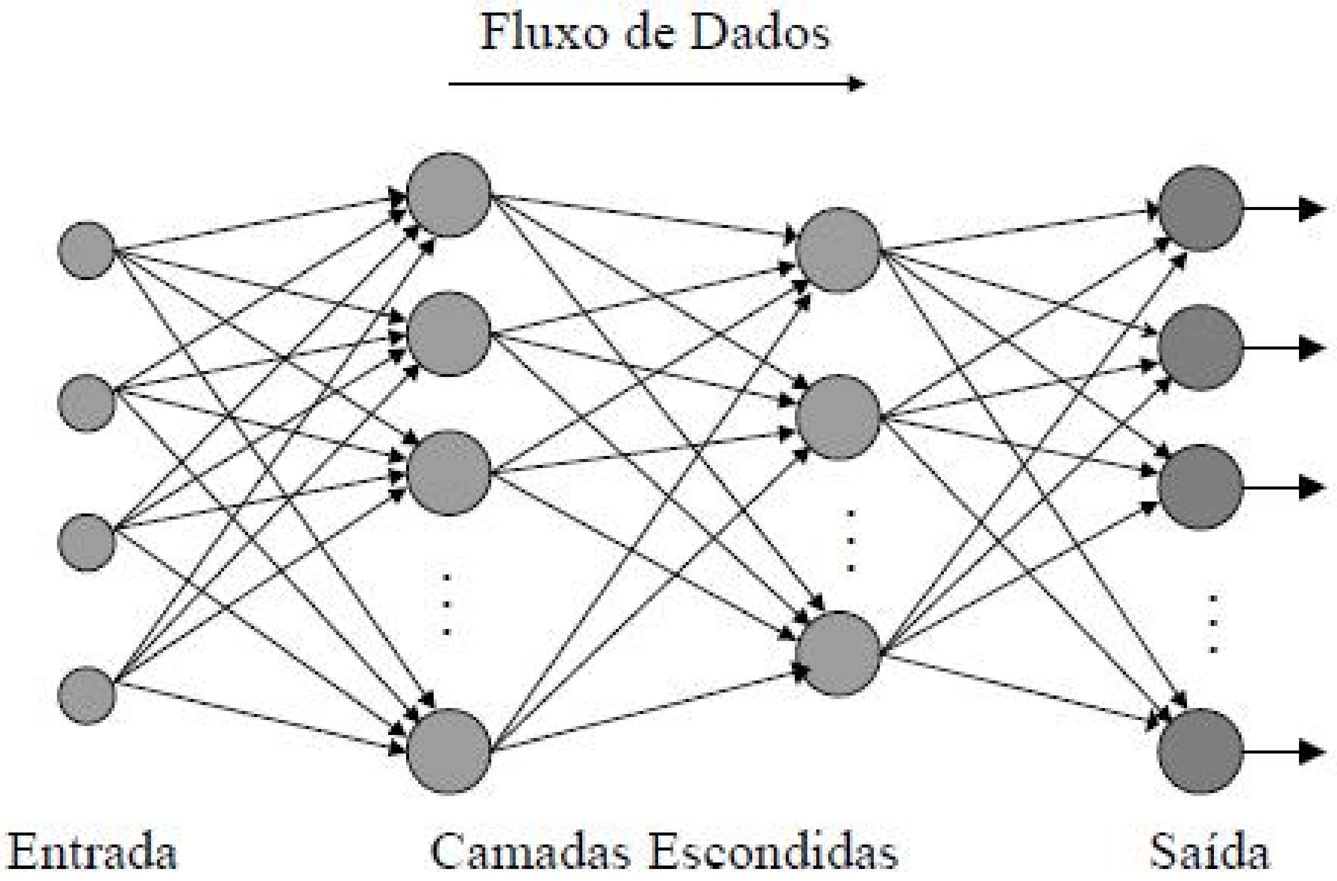
- Fase 1: Feed-Forward



Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

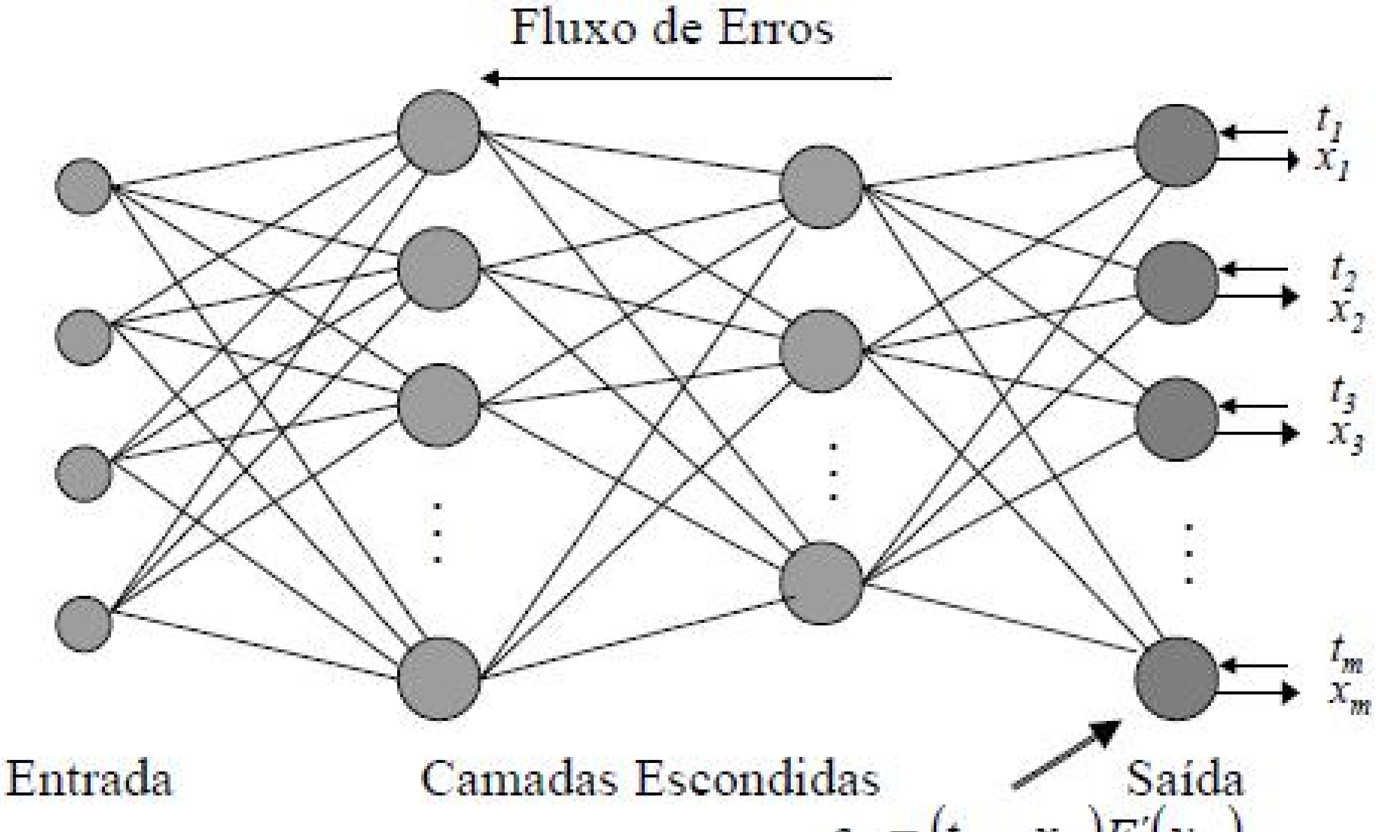
- Fase 1: Feed-Forward



Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

- Fase 1: Feed-Backward
  - Cálculo do erro da camada de saída



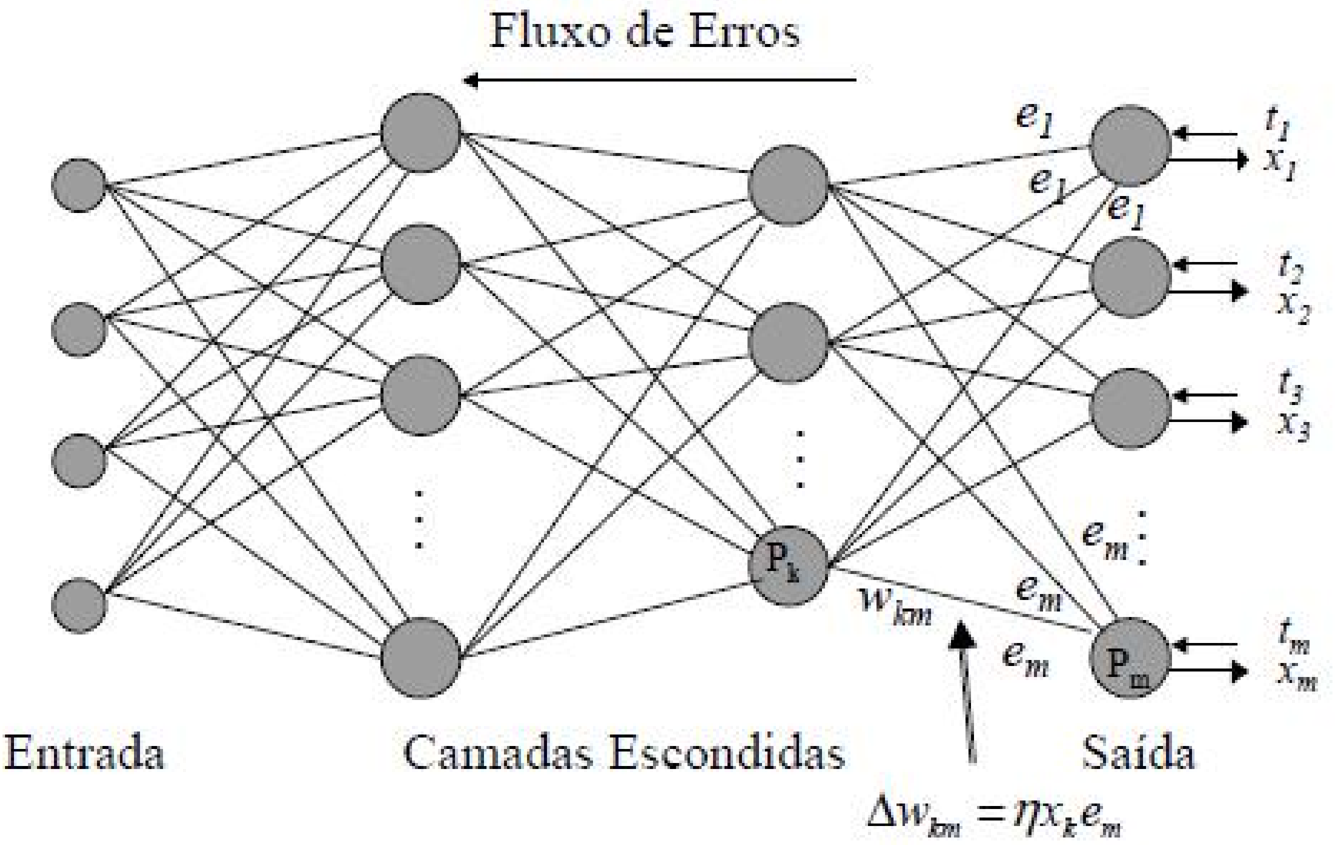
$$e_m = (t_m - x_m)F'(y_m)$$

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

- Fase 1: Feed-Backward

  Atualização dos pesos da camada de saída

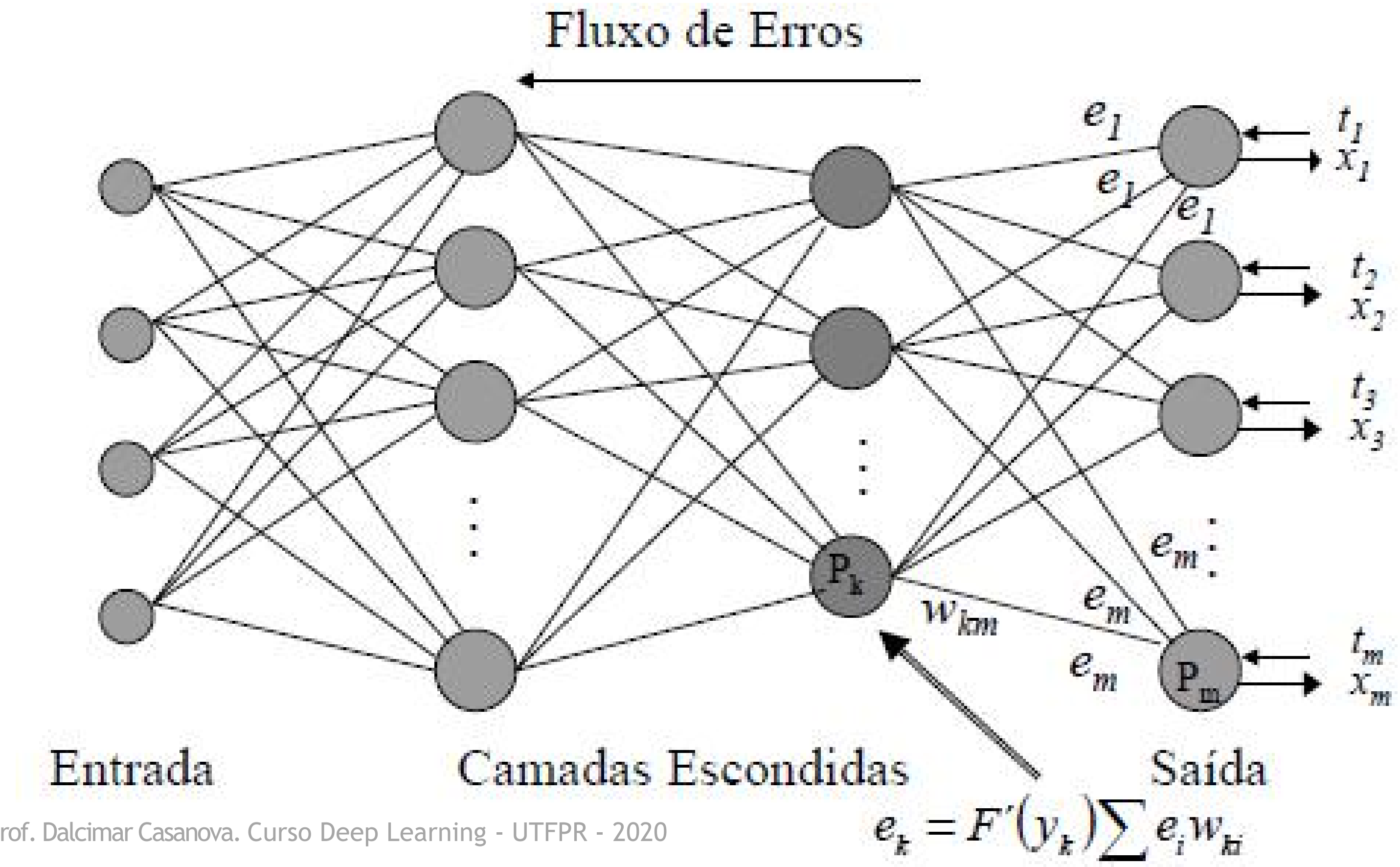


Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

- Fase 1: Feed-Backward

  Cálculo do erro da 2º camada escondida



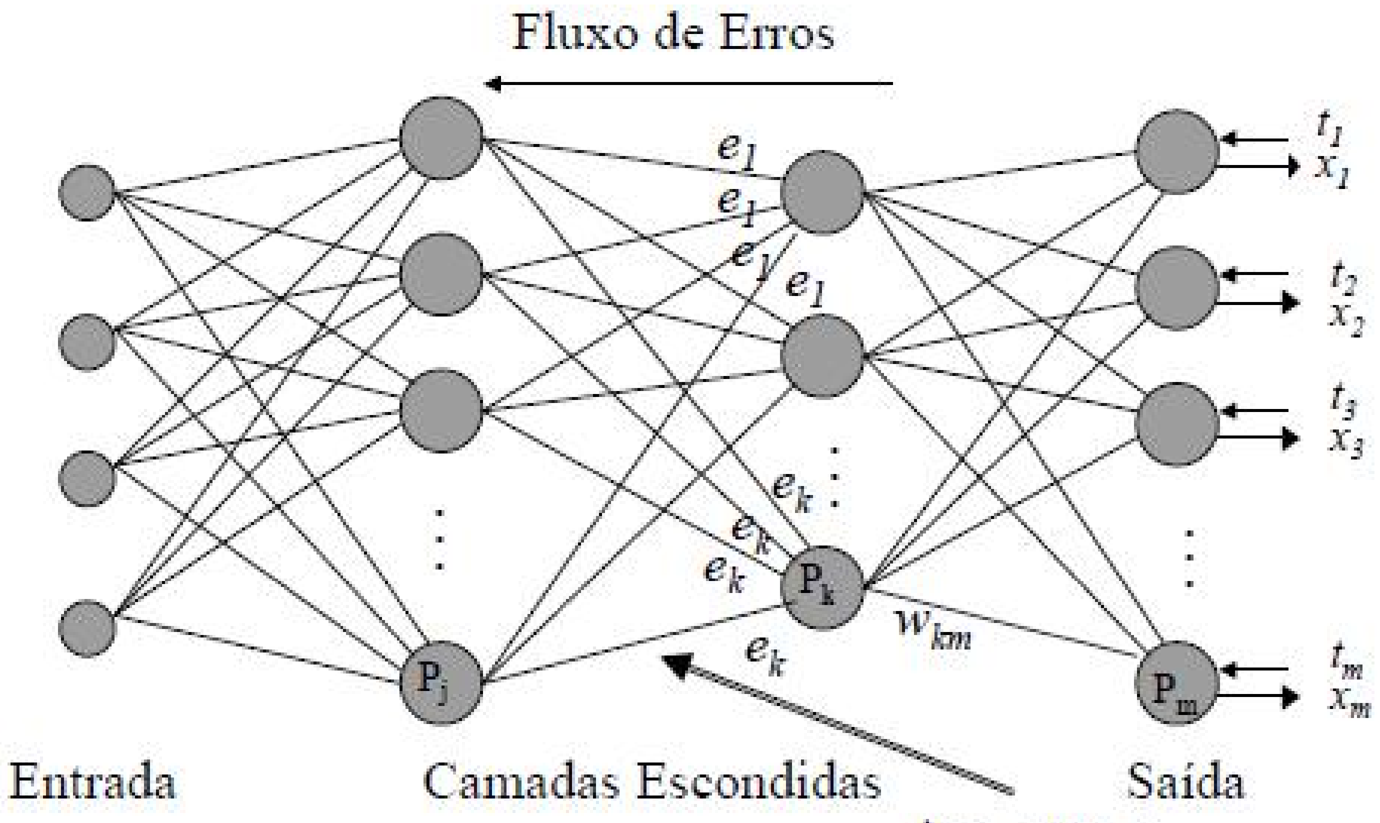
$$e_k = F'(y_k)\sum e_i w_{ki}$$

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

- Fase 1: Feed-Backward

  Atualização dos pesos da 2º camada escondida



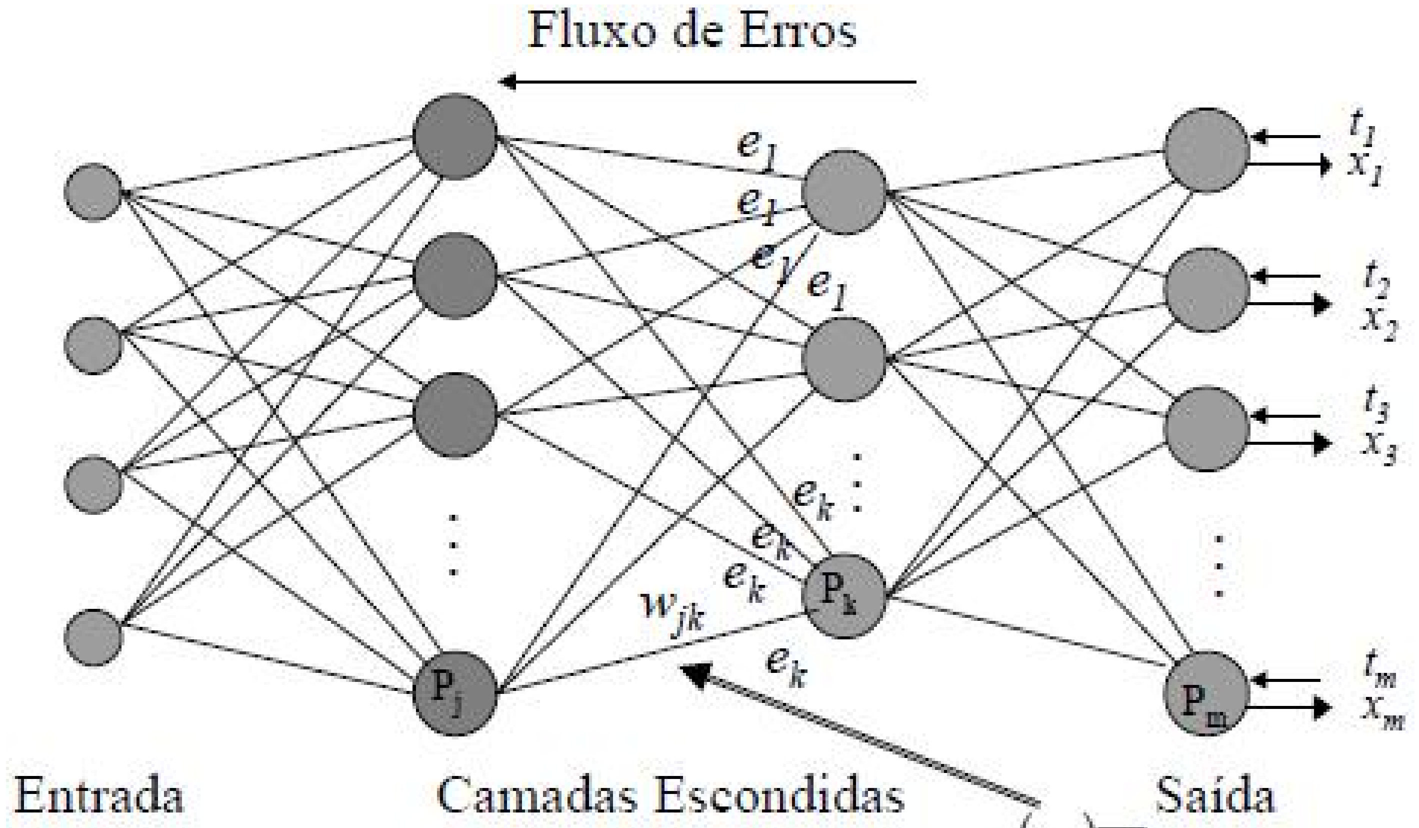
$$\Delta w_{jk} = \eta x_j e_k$$

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

- Fase 1: Feed-Backward

  Cálculo do erro da 1º camada escondida

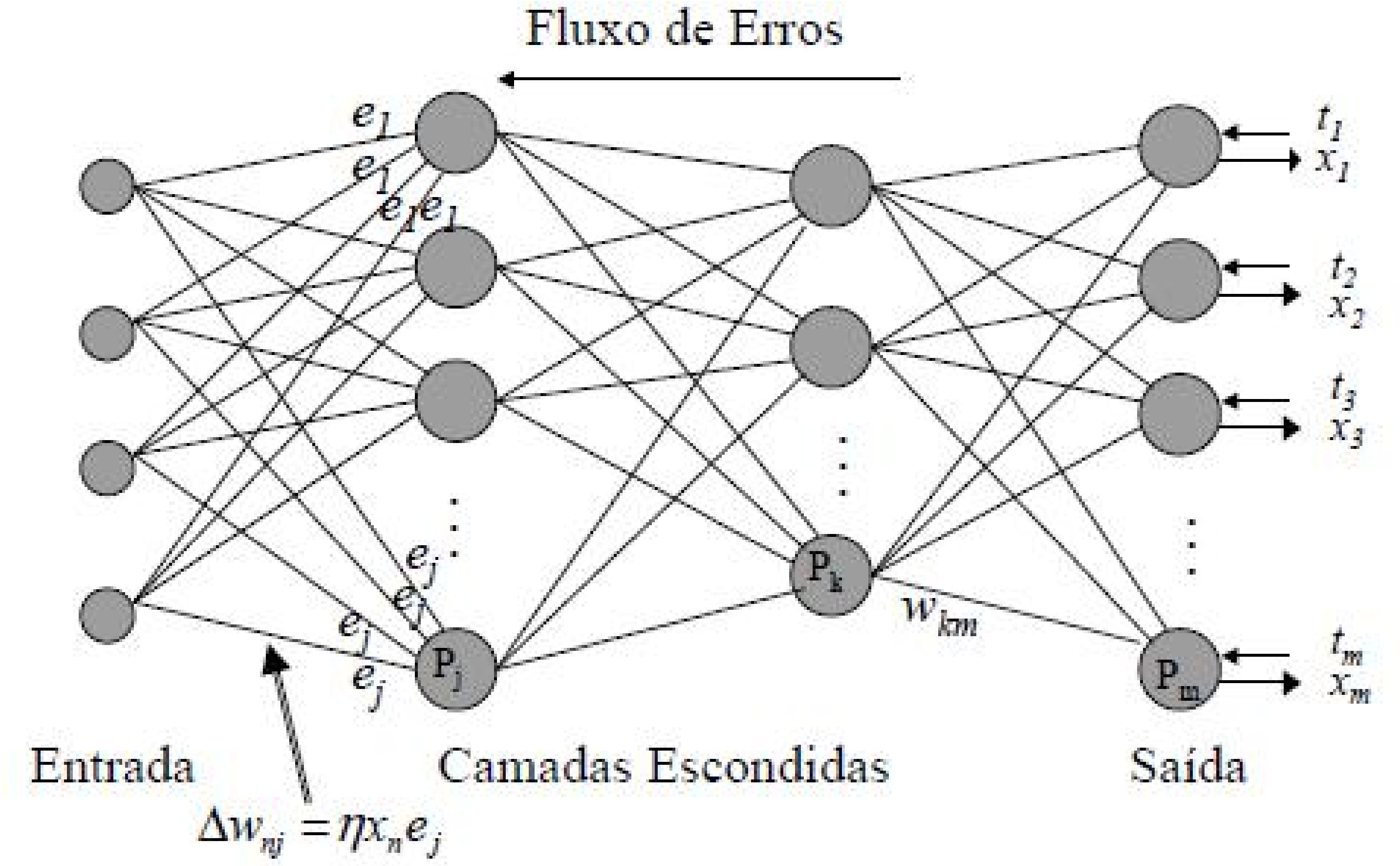


$$e_j = F'(y_j)\sum e_i w_{ji}$$

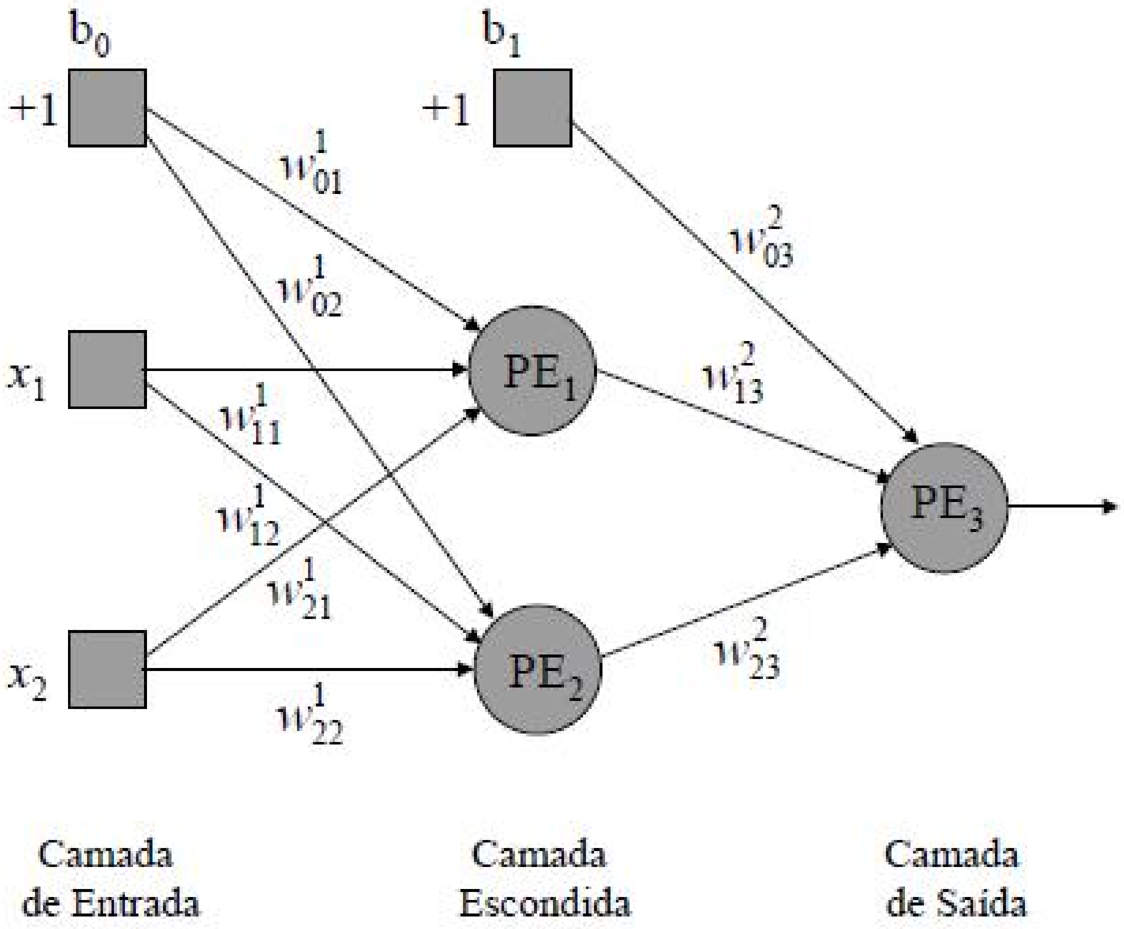
Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Processo de aprendizado

- Fase 1: Feed-Backward
  - Atualização dos pesos da 1º camada escondida



Exemplo MLP



Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Exemplo MLP

- Entrada:

  $x_1=1$, $x_2=0$

- Saída Desejada:

  $t_3 = 1$

- Pesos iniciais:

  $w_{ij}(0) = 0$

- Taxa de Aprendizagem:

  $\eta = 0.5$

- Função de Ativação:

$$F(y_i)=\frac{1}{1+\exp(-y_i)}$$

- Derivada da Função de Ativação:

$$F'(y_i)=\frac{\exp(-y_i)}{[1+\exp(-y_i)]^2}$$

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Exemplo MLP

- Algoritmo de Aprendizado:

$$w_{ij} = w_{ij} + \eta x_i e_j$$

Camada de Saída

$$e_j = \left(t_j - x_j\right) F'\left(v_j\right)$$

Camada Escondida

$$e_j = F'\left(v_j\right) \sum_k e_k w_{jk}$$

Source: Prof. Dalcimar Casanova. Curso Deep Learning - UTFPR - 2020

Exemplo MLP

- Feed-Forward:

  $y_1 = 1*0+1*0+0*0 = 0$

  - $x_1 = F(y_1) = 0.5$

  $y_2 = 1*0+1*0+0*0 = 0$

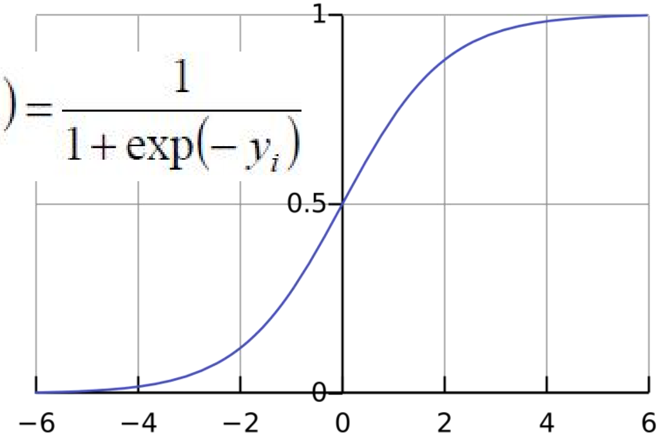
  - $x_2 = F(y_2) = 0.5$

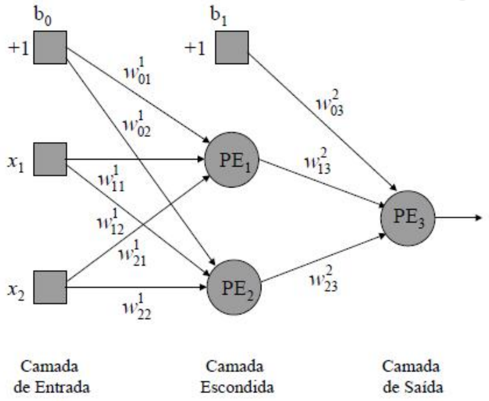
  $y_3 = 1*0+0.5*0+0.5*0 = 0$

  - $x_3 = F(y_3) = 0.5$

$$y_j = \sum x_i w_{ij} + \theta_j$$

$$F(y_i) = \frac{1}{1+\exp(-y_i)}$$





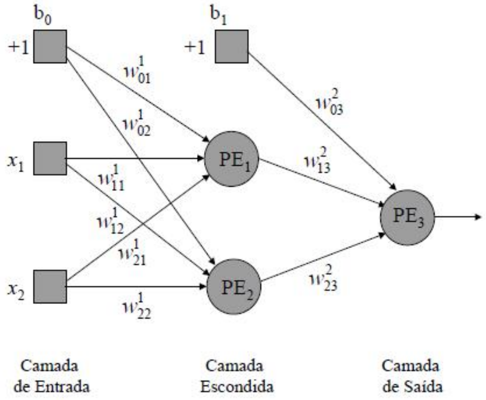
Exemplo MLP

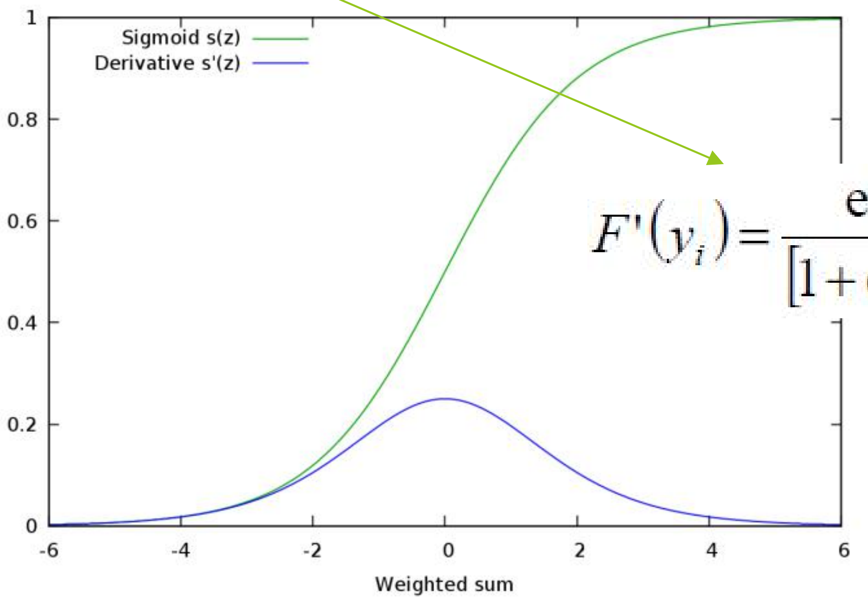
- Feed-Backward:

  t_3-x_3 = 1-0.5 = 0.5

  e_3 = 0.5*0.25 = 0.125

$$e_j = (t_j - x_j)F'(y_j)$$





$$F'(y_i) = \frac{\exp(-y_i)}{[1+\exp(-y_i)]^2}$$

Exemplo MLP

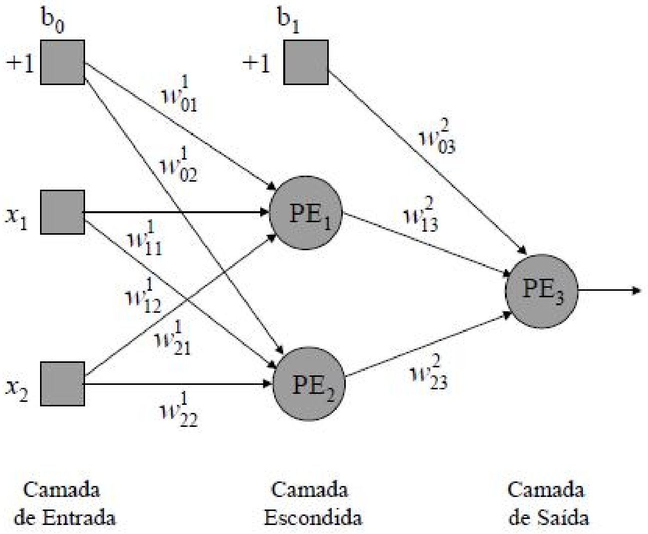
- Feed-Backward:

$$w_{ij} = w_{ij} + \eta x_i e_j$$

$w^2_{03} = 0+0.5*1*0.125 = 0.0625$

$w^2_{13} = 0+0.5*0.5*0.125 = 0.0313$

$w^2_{23} = 0+0.5*0.5*0.125 = 0.0313$



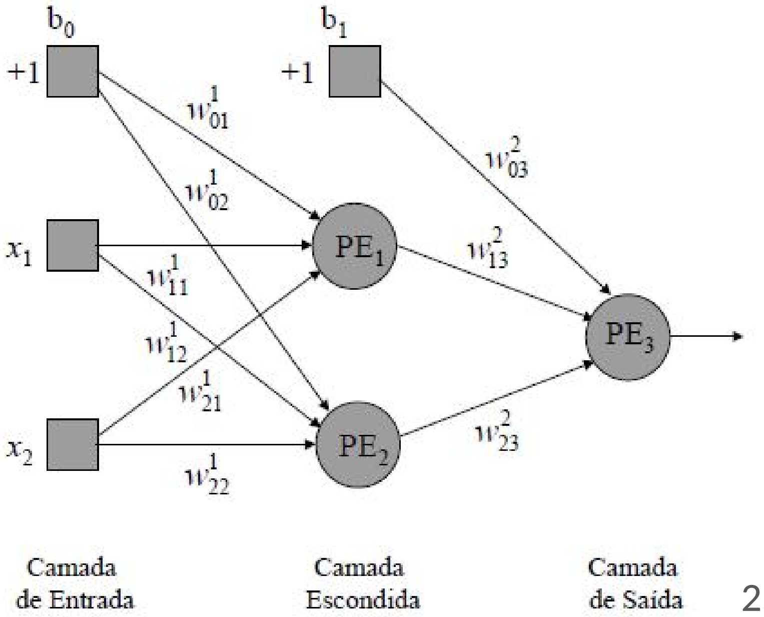
Exemplo MLP

- Feed-Backward:

$$e_1 = 0.25*(0.125*0.0313) = 0.00097813$$

$$e_2 = 0.25*(0.125*0.0313) = 0.00097813$$

$$e_j = F'(v_j)\sum_k e_k w_{jk}$$



Exemplo MLP

- Feed-Backward:

$$w_{ij} = w_{ij} + \eta x_i e_j$$

$w^1_{01} = 0+0.5*1*0.00097813 = 0.00048907$

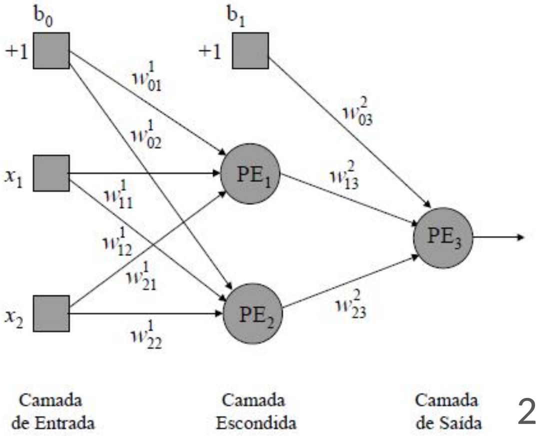
$w^1_{02} = 0+0.5*1*0.00097813 = 0.00048907$

$w^1_{11} = 0+0.5*1*0.00097813 = 0.00048907$

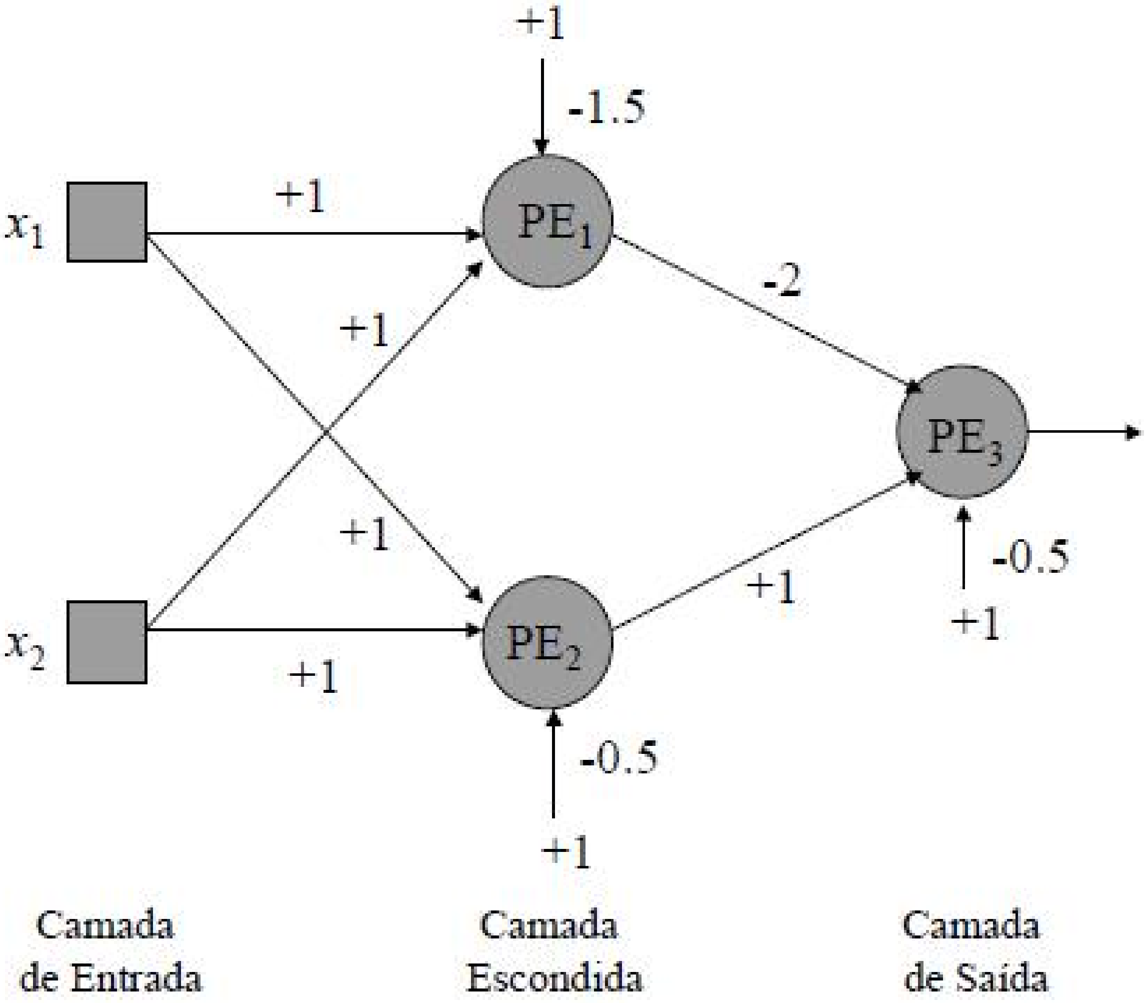
$w^1_{12} = 0+0.5*1*0.00097813 = 0.00048907$

$w^1_{21} = 0+0.5*0*0.00097813 = 0$

$w^1_{22} = 0+0.5*0*0.00097813 = 0$

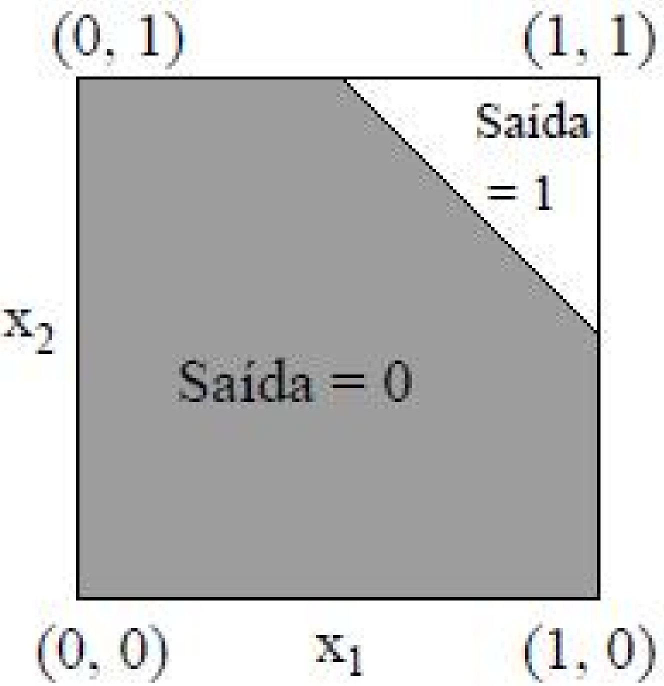


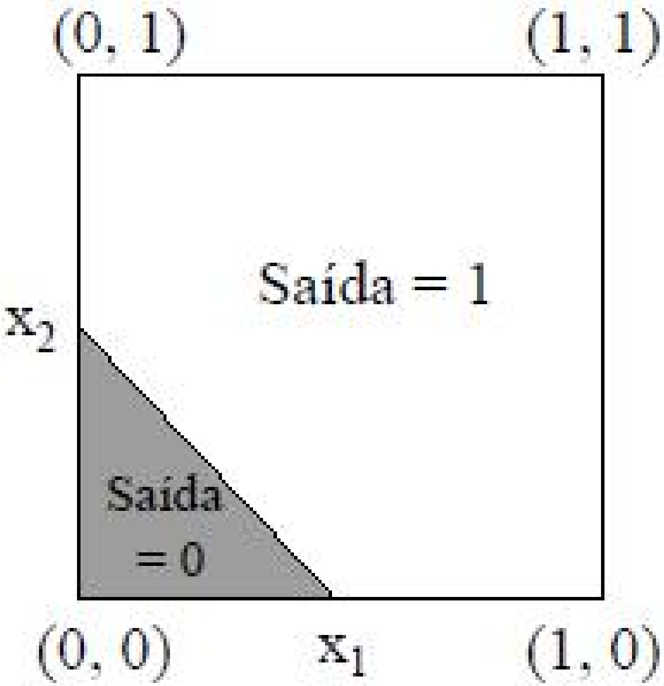
Problema XOR



Problema XOR

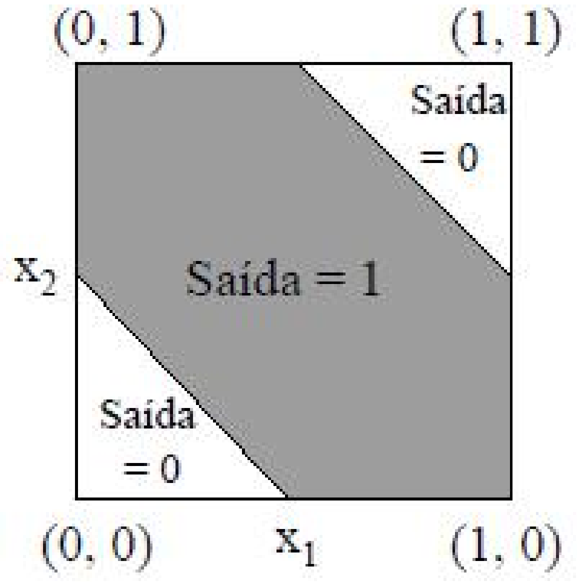
- Borda de decisão construída pelo 1º neurônio escondido
- Borda de decisão construída pelo 2º neurônio escondido



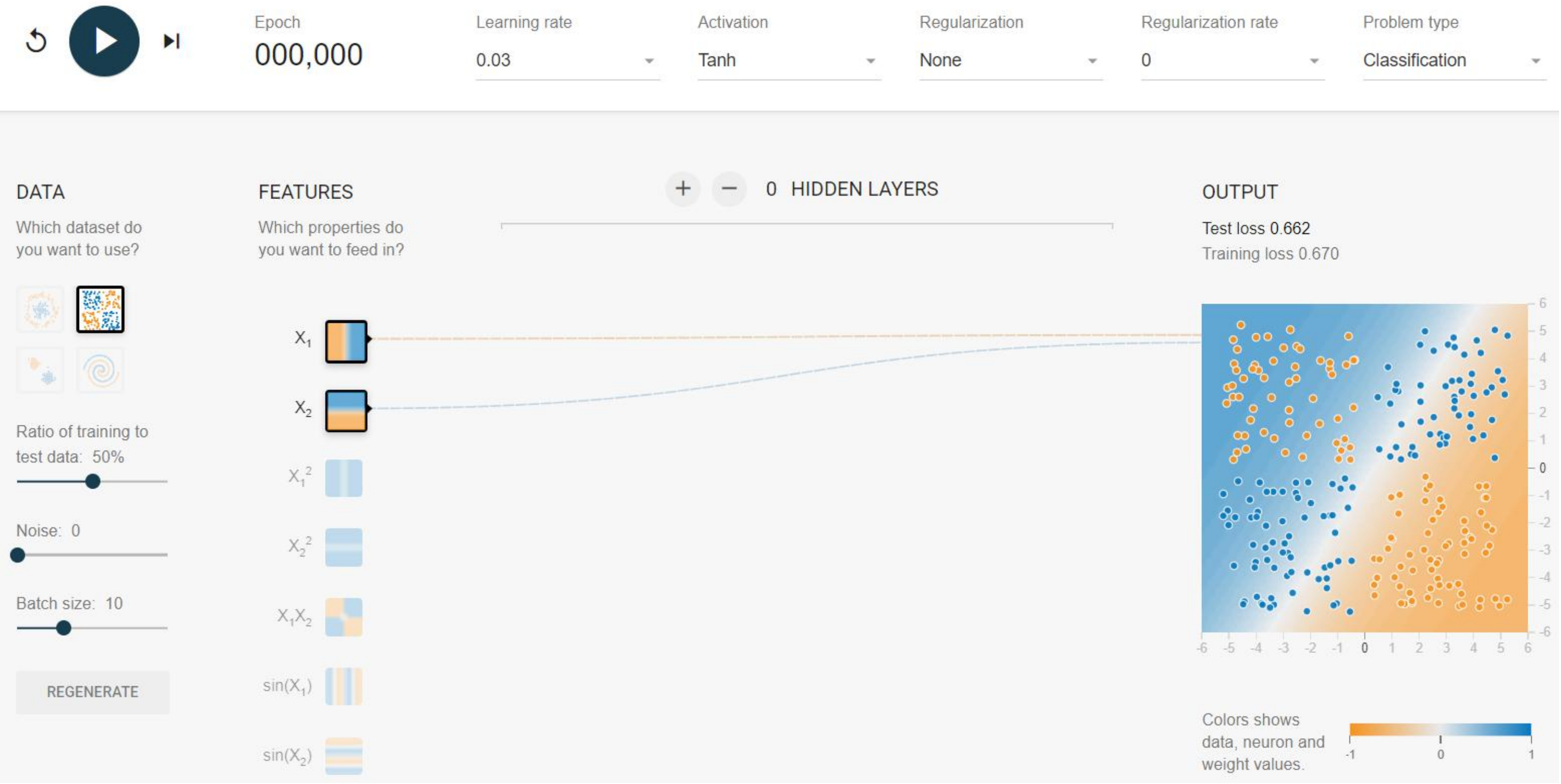


Problema XOR

- Borda de decisão construída pela rede completa



With N layer and 1 neuron



Epoch
000,000

Learning rate
0.03

Activation
Tanh

Regularization
None

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT

Test loss 0.662

Training loss 0.670

Colors shows data, neuron and weight values.

http://playground.tensorflow.org/

Lecture 09

Multilayer Perceptrons

STAT 453: Deep Learning, Spring 2020

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching/stat453-ss2020/

Topics

Multilayer Perceptron Architecture

Nonlinear Activation Functions

Multilayer Perceptron Code Examples

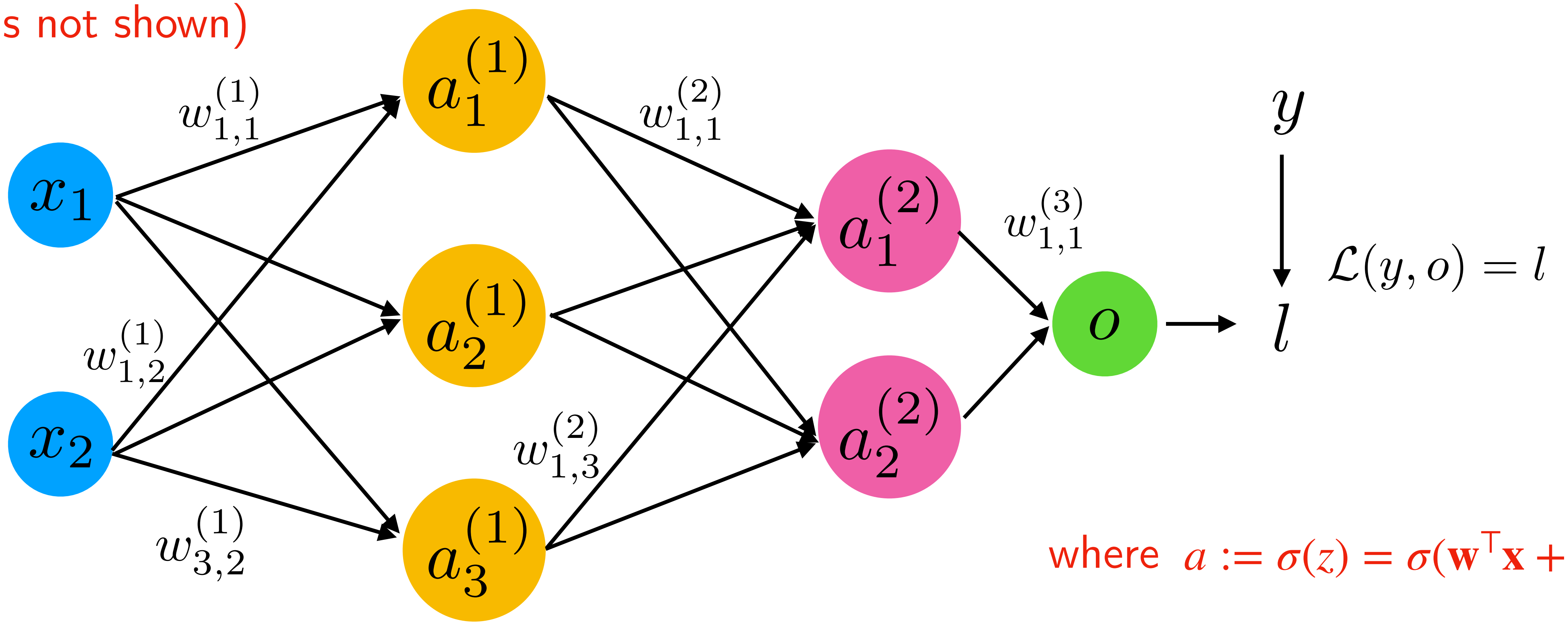
Overfitting and Underfitting

Cats & Dogs and Custom Data Loaders

Graph with Fully-Connected Layers = Multilayer Perceptron

Nothing new, really

(bias not shown)

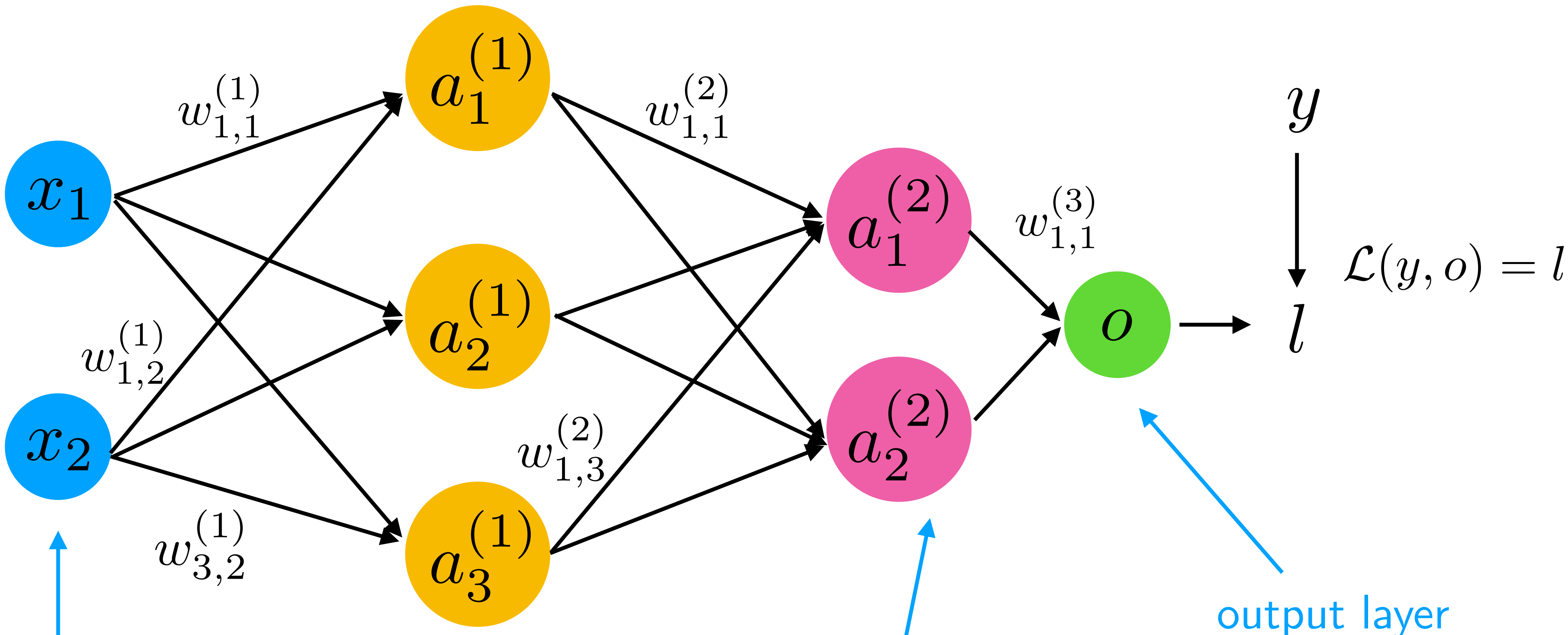


where $a := \sigma(z) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$

$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

(Assume network for binary classification)

Graph with Fully-Connected Layers = Multilayer Perceptron



input layer (layer 1)

1st hidden layer (layer 2)

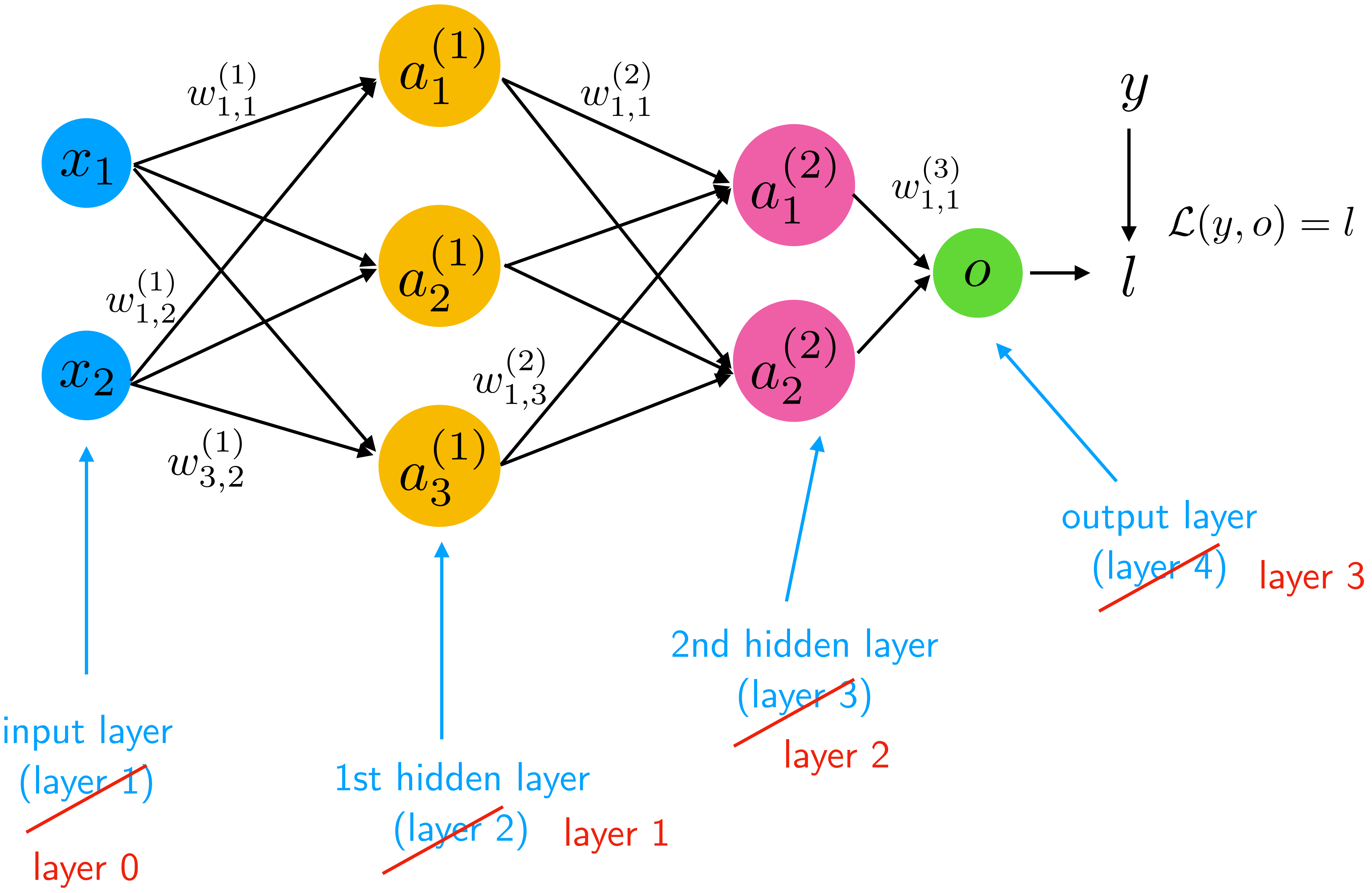
2nd hidden layer (layer 3)

output layer (layer 4)

here, we could also write o as $a_1^{(3)}$

Graph with Fully-Connected Layers
= Multilayer Perceptron

A more common counting/naming scheme, because then a perceptron/Adaline/ logistic regression model can be called a "1-layer neural network"



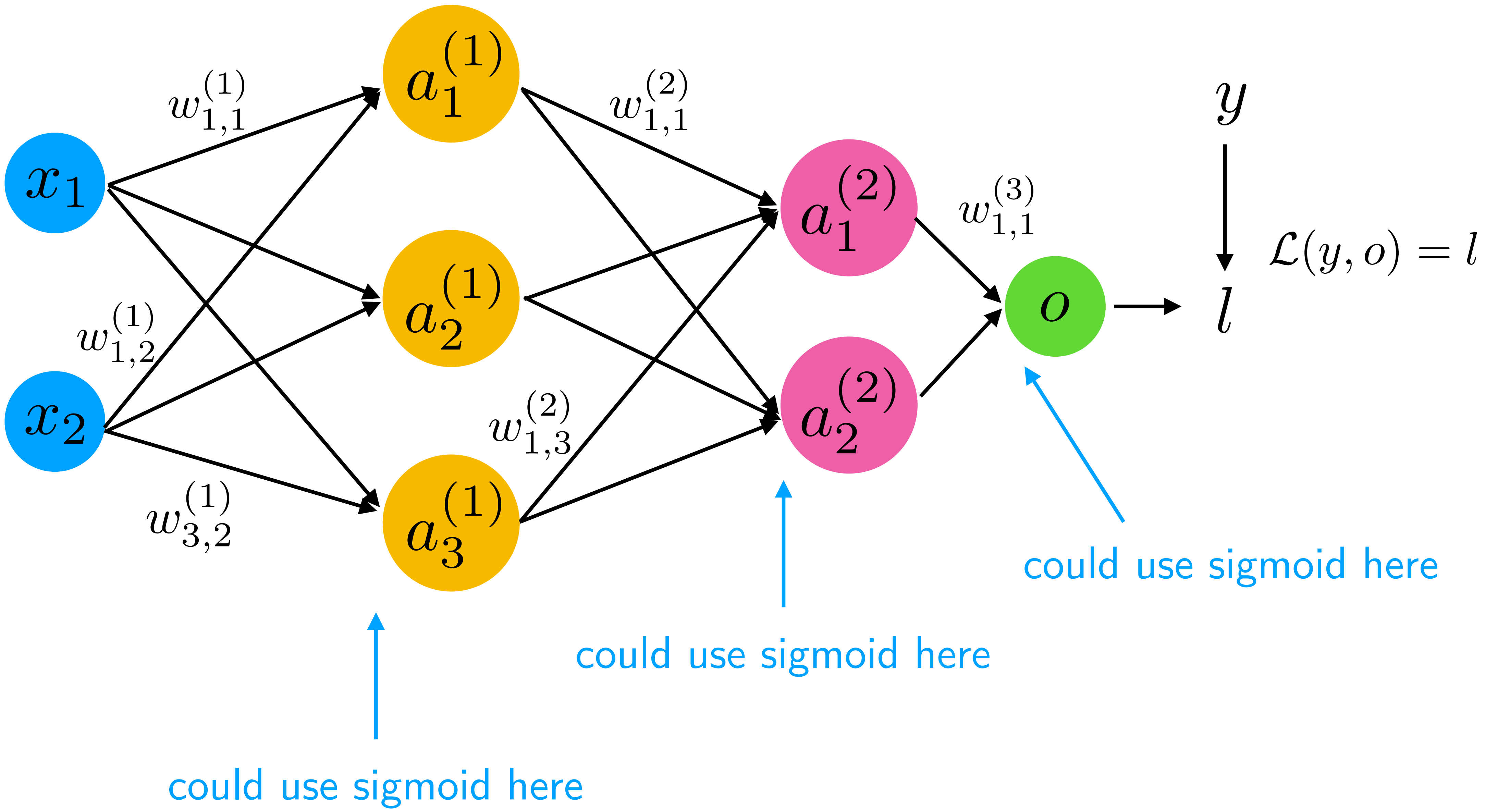
input layer
(~~layer 1~~)
layer 0

1st hidden layer
(~~layer 2~~) layer 1

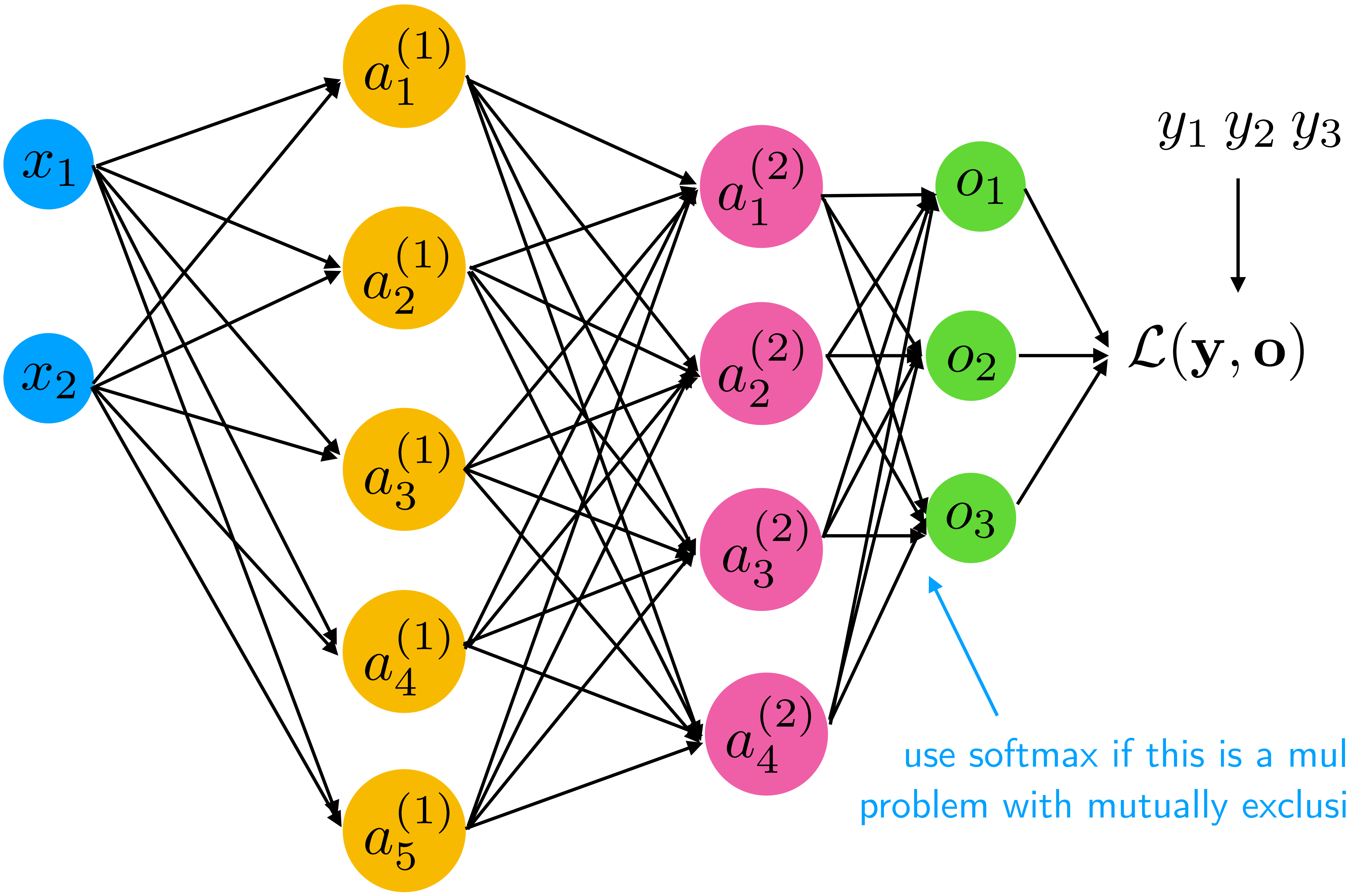
2nd hidden layer
(~~layer 3~~)
layer 2

output layer
(~~layer 4~~) layer 3

Graph with Fully-Connected Layers = Multilayer Perceptron



Graph with Fully-Connected Layers = Multilayer Perceptron

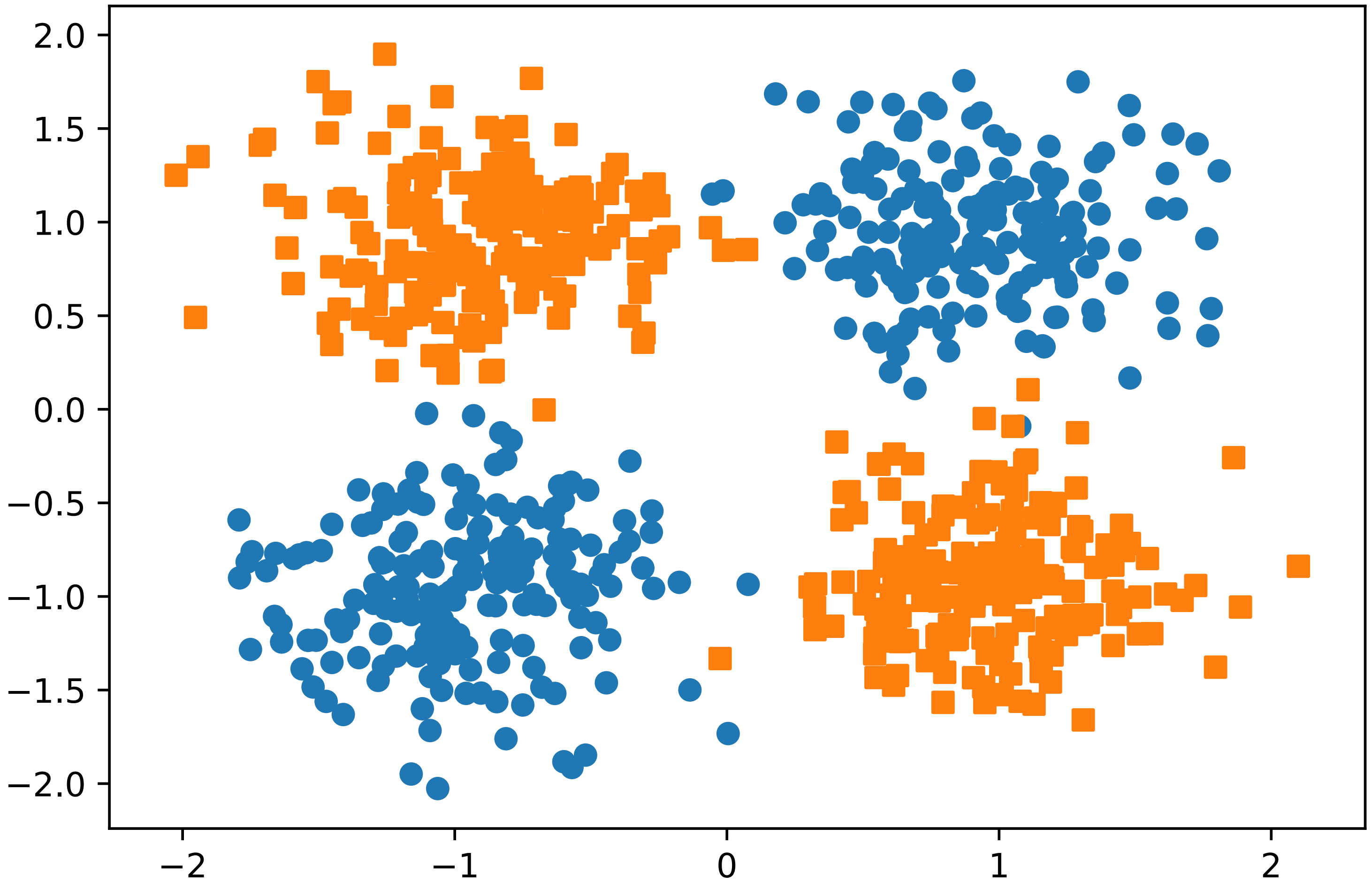


use softmax if this is a multi-class problem with mutually exclusive classes

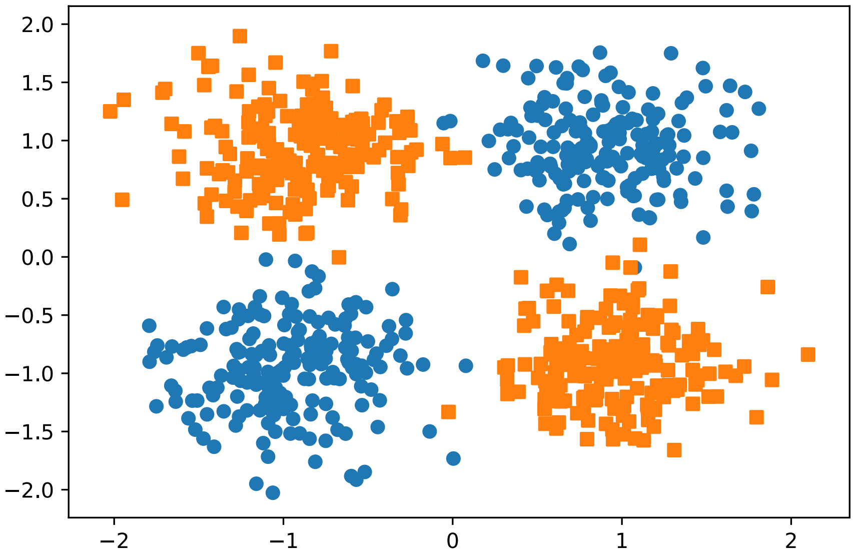
Activation Functions

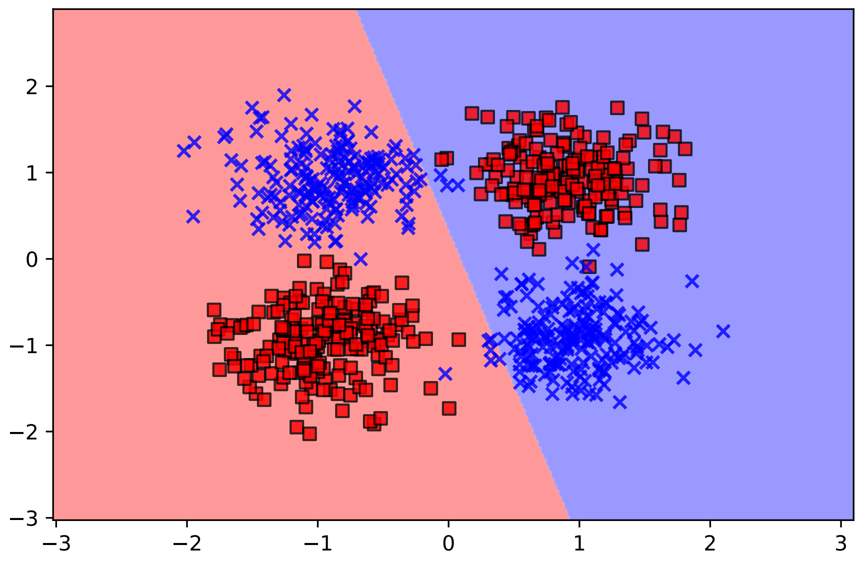
Question: What happens if we don't use non-linear activation functions?

Solving the XOR Problem with Non-Linear Activations

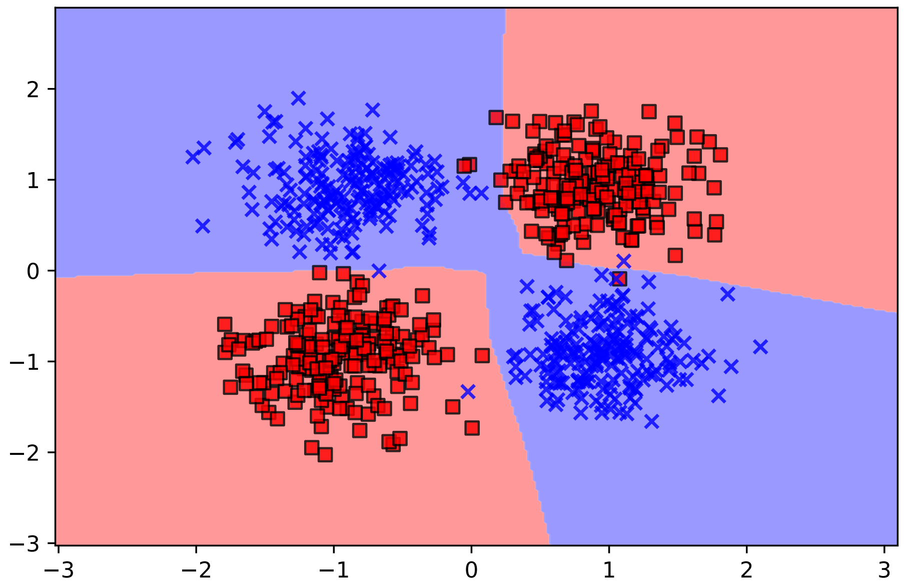


Solving the XOR Problem with Non-Linear Activations





1-hidden layer MLP
with linear activation function

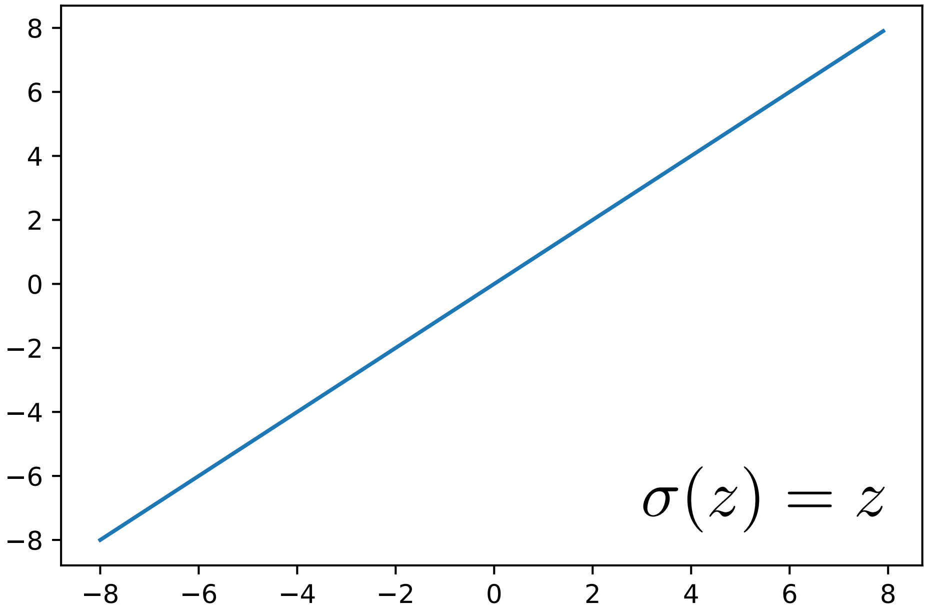


1-hidden layer MLP
with non-linear activation function (ReLU)

https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L08-mlp/code/xor-problem.ipynb

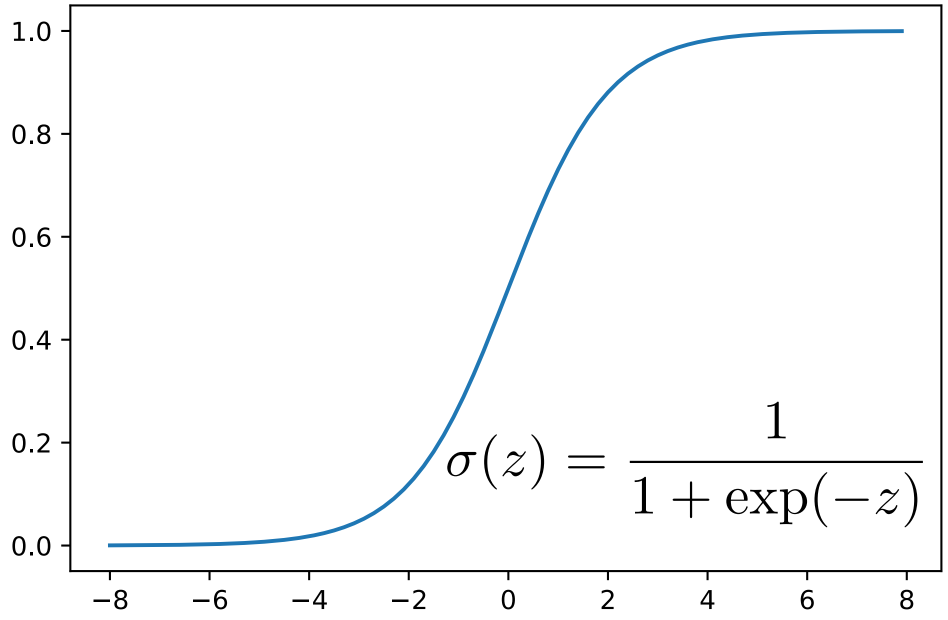
A Selection of Common Activation Functions (1)

Identity



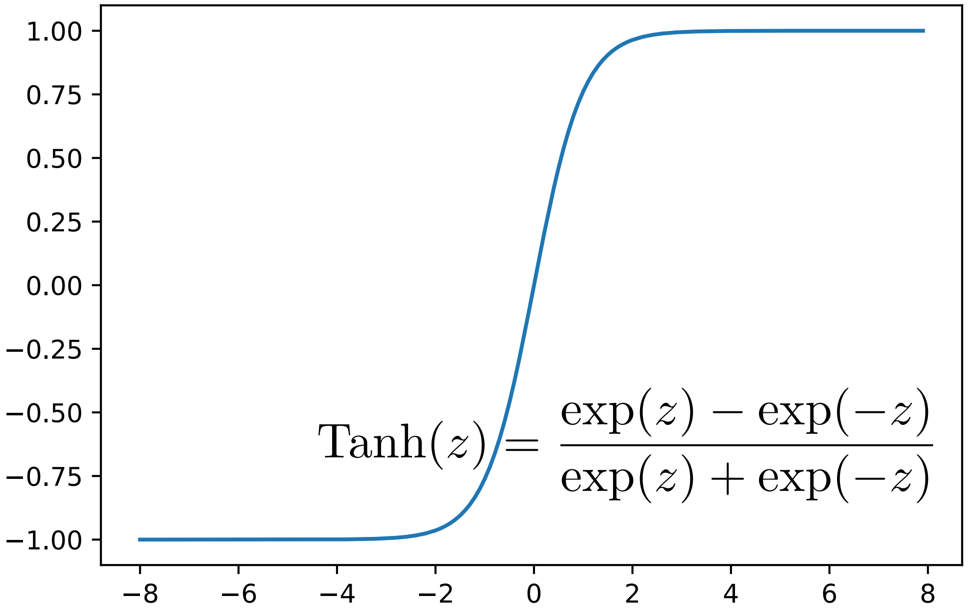
$$\sigma(z) = z$$

(Logistic) Sigmoid



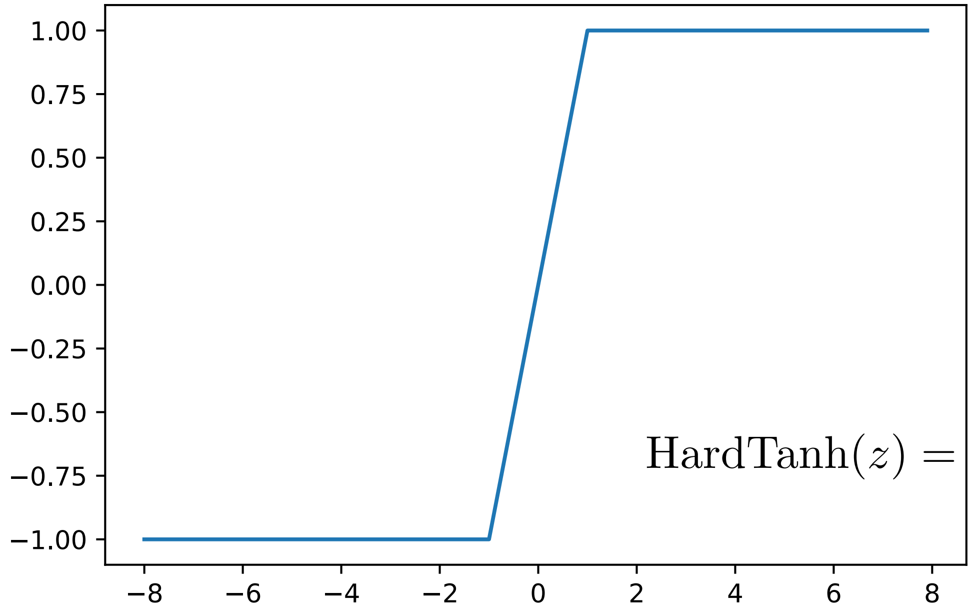
$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

Tanh ("tanH")



$$\text{Tanh}(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

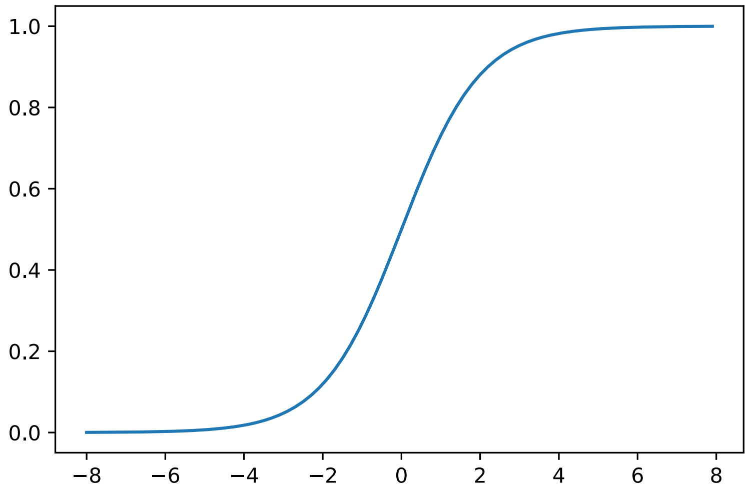
Hard Tanh



$$\text{HardTanh}(z) = \begin{cases} 1 & \text{if } z > 1 \\ -1 & \text{if } z < -1 \\ z & \text{otherwise} \end{cases}$$

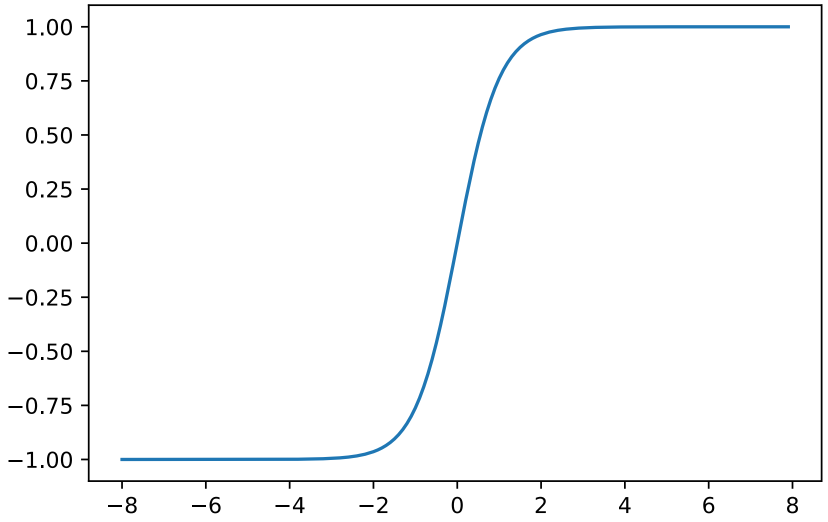
A Selection of Common Activation Functions (1)

(Logistic) Sigmoid



- Advantages of Tanh
- Mean centering
- Positive and negative values
- Larger gradients

Tanh ("tanH")



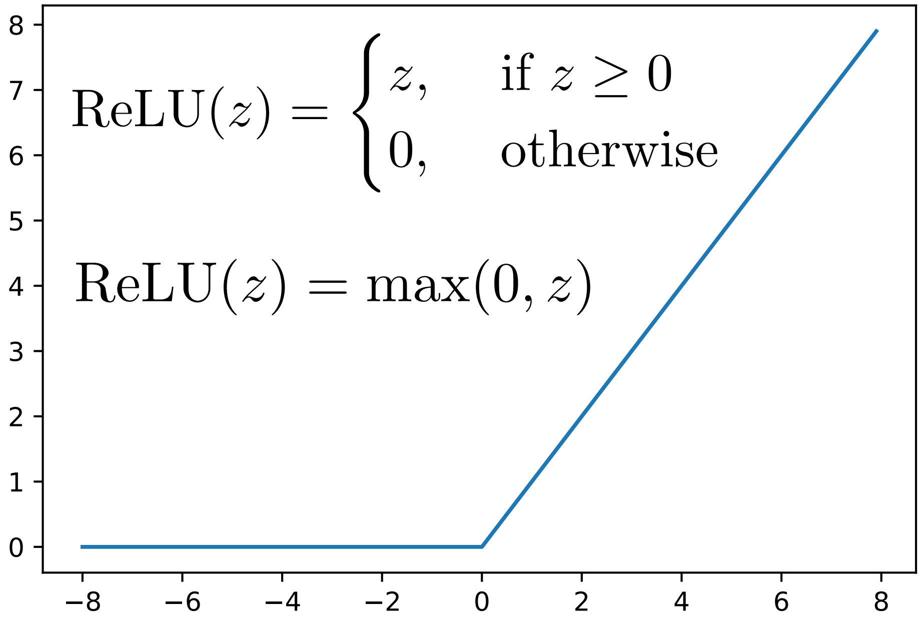
Additional tip: Also good to normalize inputs to mean zero and use random weight initialization with avg. weight centered at zero

Also simple derivative:

$$\frac{d}{dz}Tanh(z) = 1 - Tanh(z)^2$$

A Selection of Common Activation Functions (2)

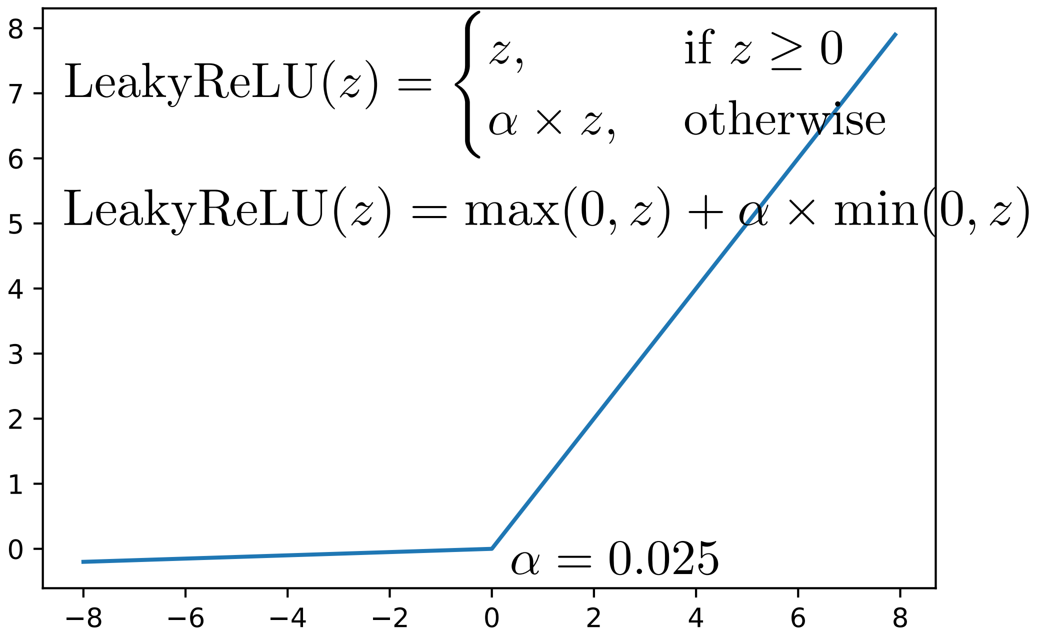
ReLU (Rectified Linear Unit)



$$\mathrm{ReLU}(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\mathrm{ReLU}(z) = \max(0, z)$$

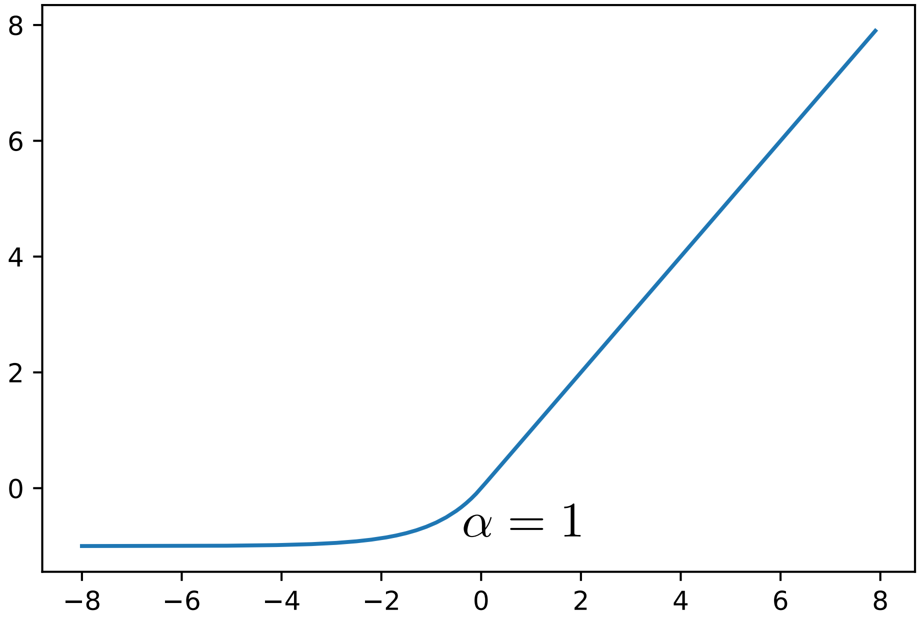
Leaky ReLU



$$\mathrm{LeakyReLU}(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha \times z, & \text{otherwise} \end{cases}$$

$$\mathrm{LeakyReLU}(z) = \max(0, z) + \alpha \times \min(0, z)$$

ELU (Exponential Linear Unit)



$$\mathrm{ELU}(z) = \max(0, z) + \min(0, \alpha \times (\exp(z) - 1))$$

PReLU (Parameterized Rectified Linear Unit)

here, alpha is a trainable parameter

$$\mathrm{PReLU}(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha z, & \text{otherwise} \end{cases}$$

$$\mathrm{PReLU}(z) = \max(0, z) + \alpha \times \min(0, z)$$

Model Evaluation

Multilayer Perceptron Architecture

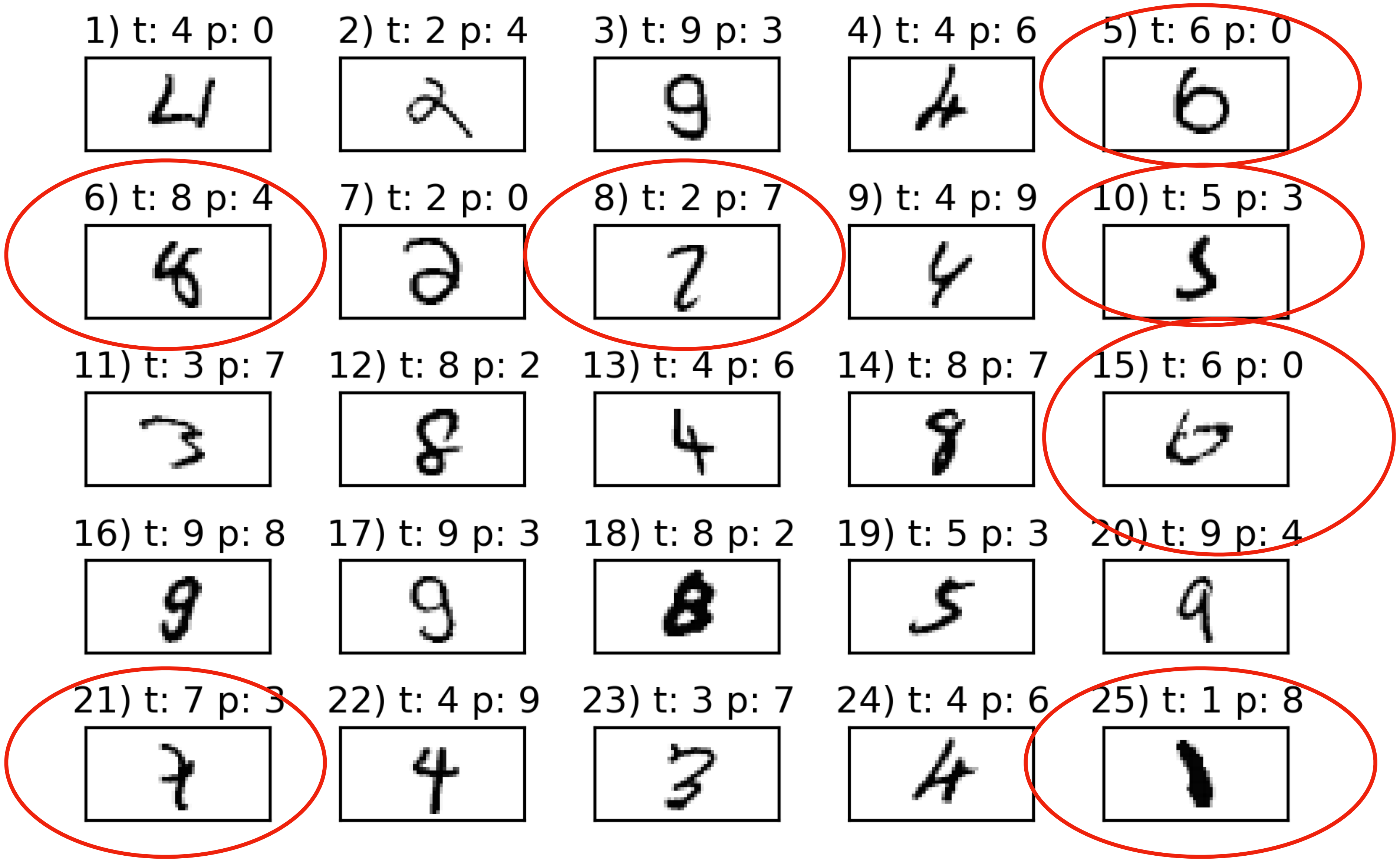
Nonlinear Activation Functions

Multilayer Perceptron Code Examples

Overfitting and Underfitting

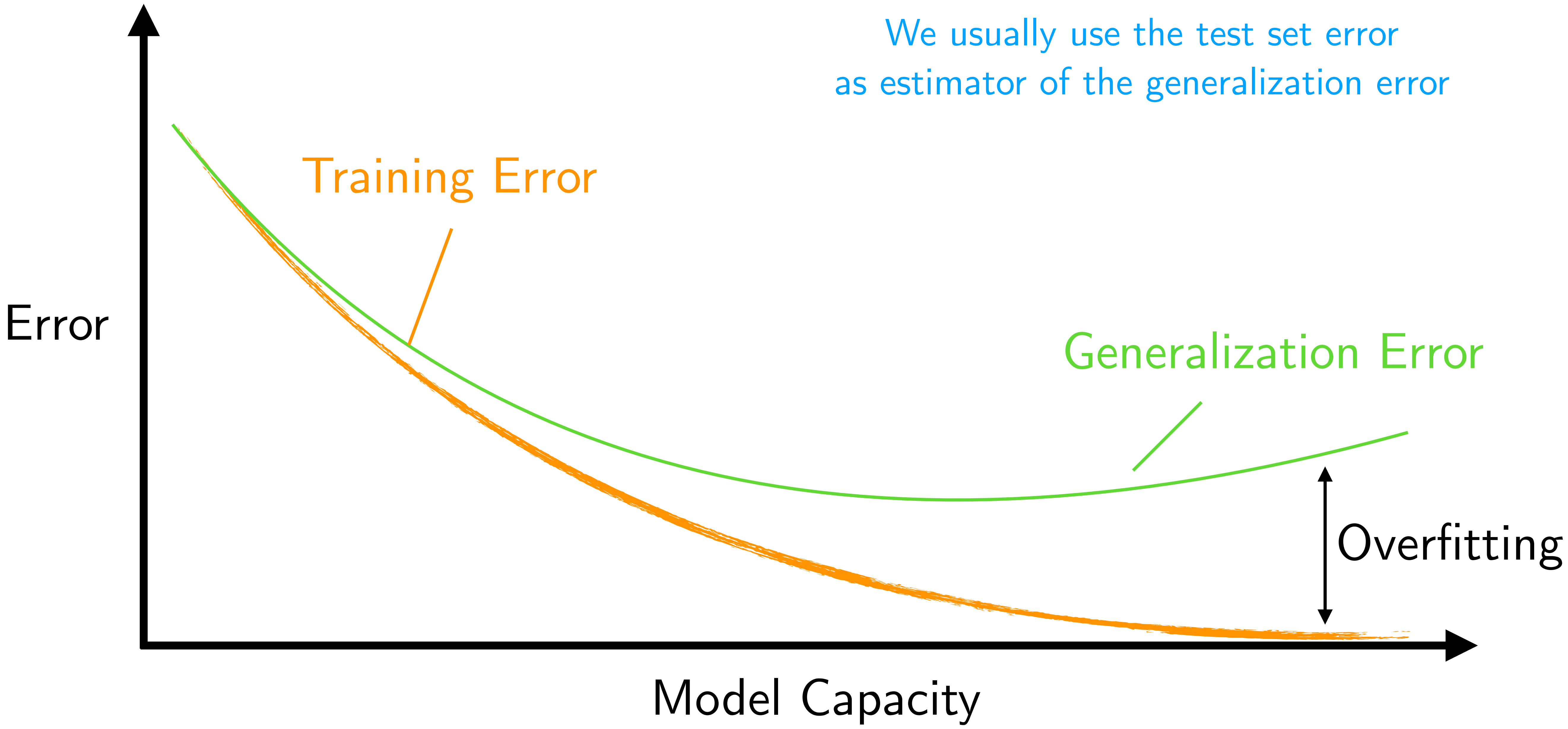
Cats & Dogs and Custom Data Loaders

Recommended Practice: Looking at Some Failure Cases



Failure cases of a ~93% accuracy (not very good, but beside the point)
2-layer (1-hidden layer) MLP on MNIST
(where *t=target* class and *p=predicted class*)

Overfitting and Underfitting

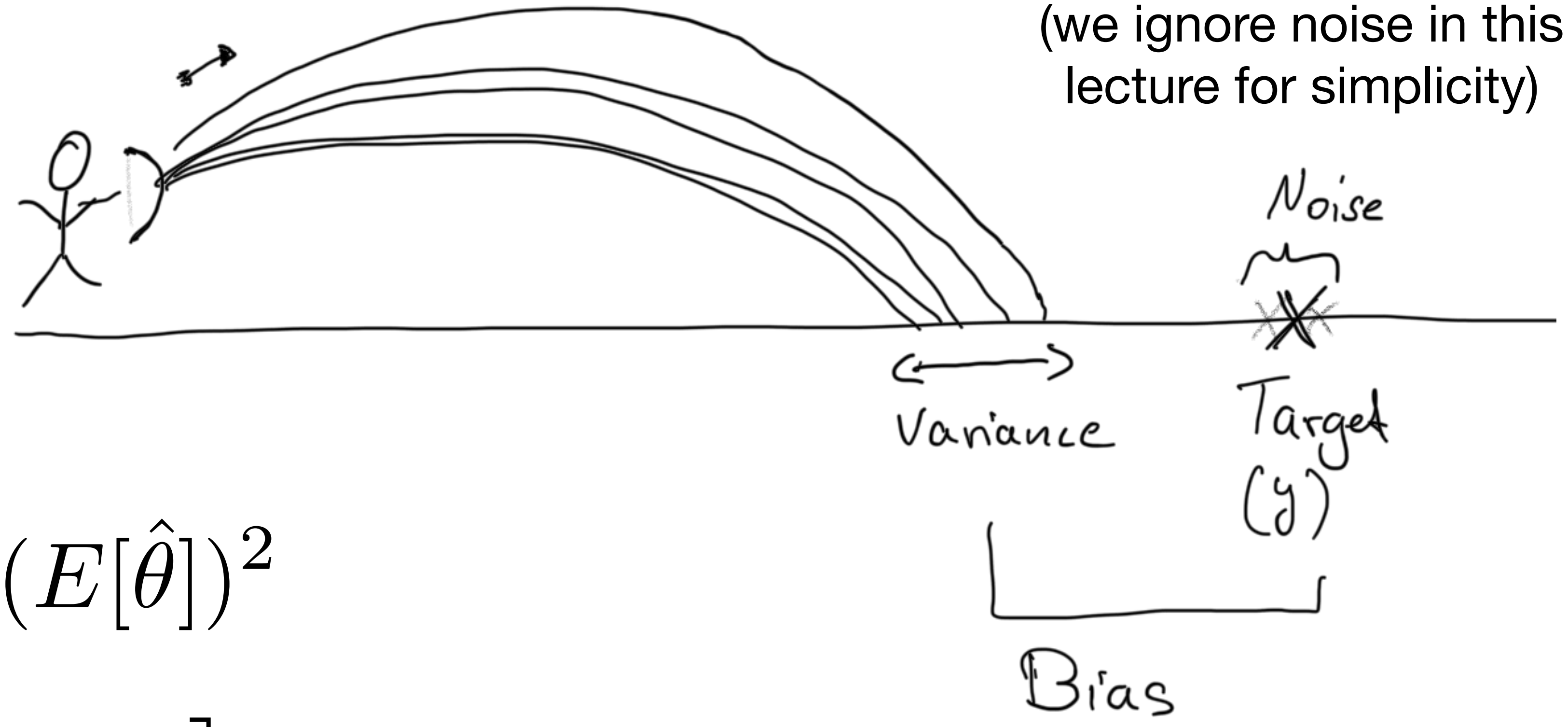


Bias-Variance Decomposition

General Definition:

Intuition:

$$\mathrm{Bias}_\theta[\hat{\theta}] = E[\hat{\theta}] - \theta$$

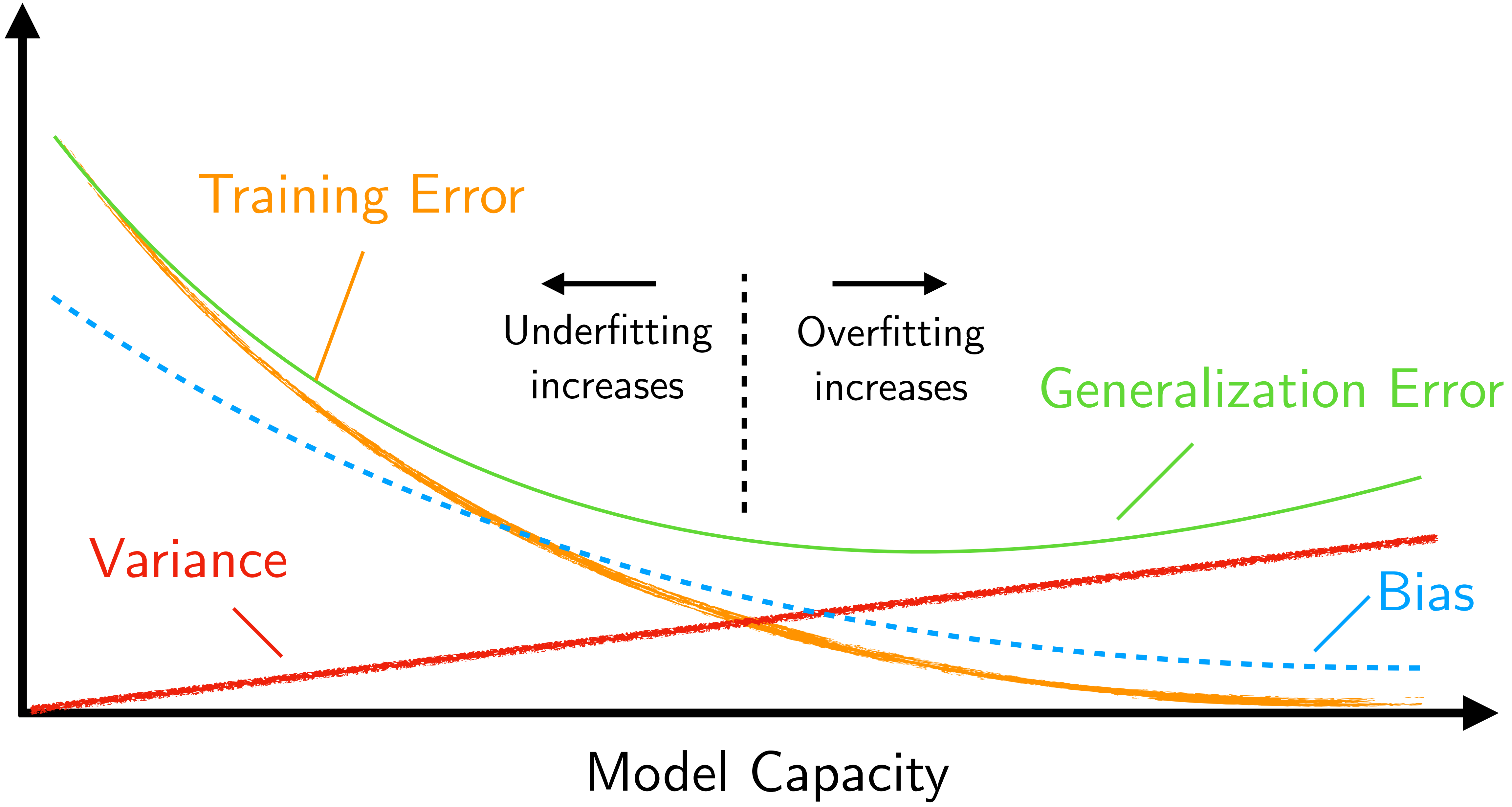


(we ignore noise in this lecture for simplicity)

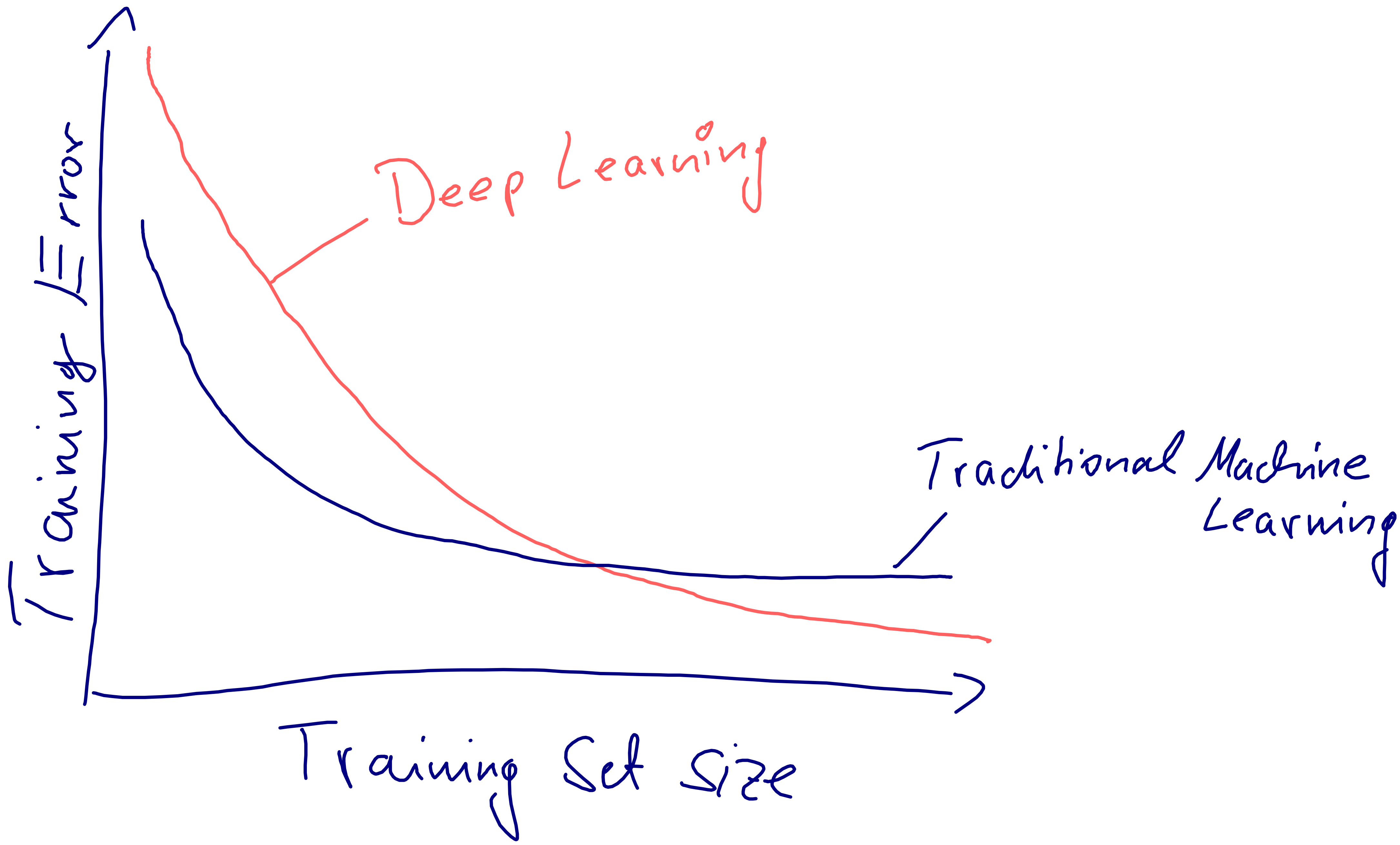
$$\mathrm{Var}_\theta[\hat{\theta}] = E\left[\hat{\theta}^2\right] - (E[\hat{\theta}])^2$$

$$\mathrm{Var}_\theta[\hat{\theta}] = E\left[(E[\hat{\theta}] - \hat{\theta})^2\right]$$

Bias & Variance vs Overfitting & Underfitting



Deep Learning Works Best with Large Datasets



Bias & Variance vs Overfitting & Underfitting

When reading DL resources, you'll notice many researchers use *bias* and *variance* to describe *underfitting* and *overfitting* (they are related but not the same!)

Multilayer Perceptron Architecture

Nonlinear Activation Functions

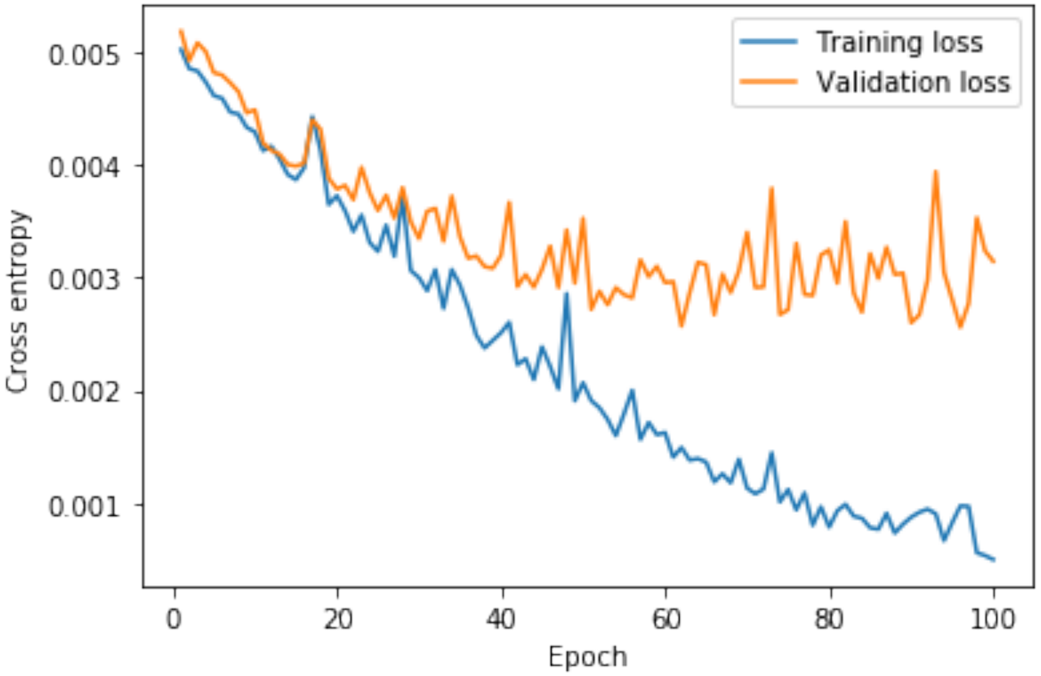
Multilayer Perceptron Code Examples

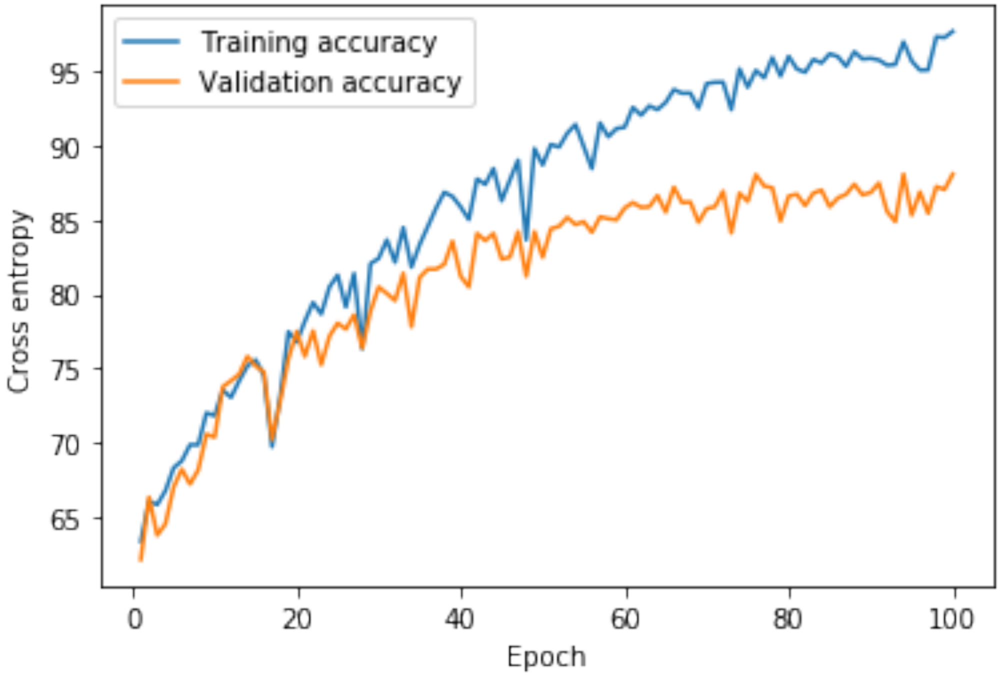
Overfitting and Underfitting

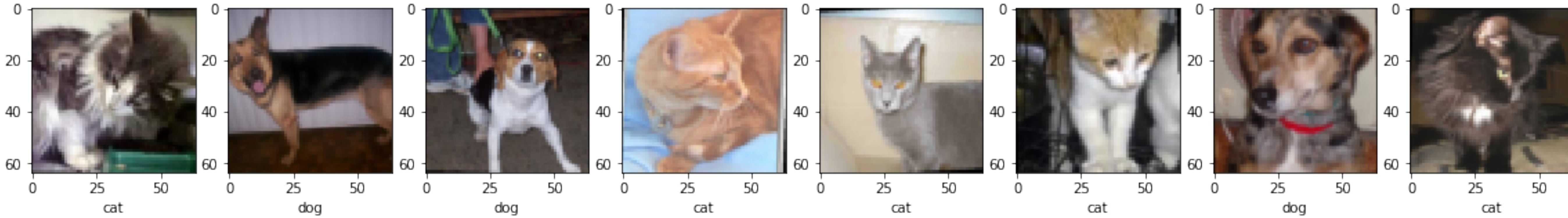
Cats & Dogs and Custom Data Loaders

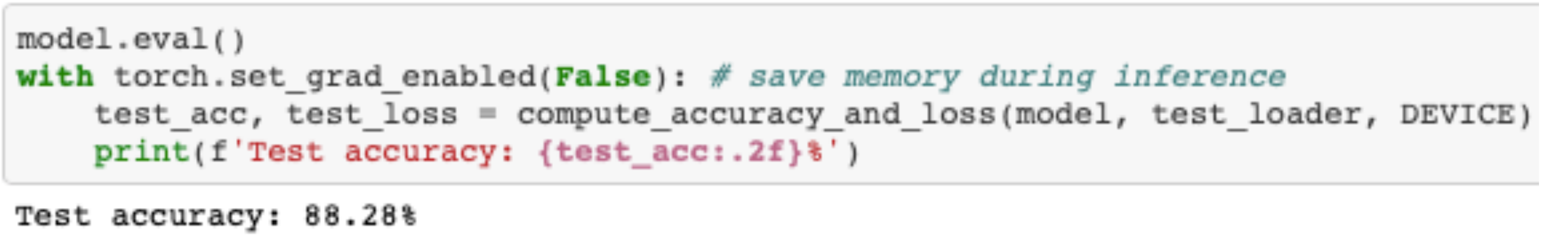
VGG16 Convolutional Neural Network for Kaggle's Cats and Dogs Images

A "real world" example









```
model.eval()
with torch.set_grad_enabled(False): # save memory during inference
    test_acc, test_loss = compute_accuracy_and_loss(model, test_loader, DEVICE)
    print(f'Test accuracy: {test_acc:.2f}%')
```

```
Test accuracy: 88.28%
```

Training/Validation/Test splits

Ratio depends on the dataset size, but a 80/5/15 split is usually a good idea

- Training set is used for training, it is not necessary to plot the training accuracy during training but it can be useful
- Validation set accuracy provides a rough estimate of the generalization performance (it can be optimistically biased if you design the network to do well on the validation set ("information leakage")
- Test set should only be used once to get an unbiased estimate of the generalization performance

Training/Validation/Test splits

```
Epoch: 001/100 | Batch 000/156 | Cost: 1136.9125
Epoch: 001/100 | Batch 120/156 | Cost: 0.6327
Epoch: 001/100 Train Acc.: 63.35% | Validation Acc.: 62.12%
Time elapsed: 3.09 min
Epoch: 002/100 | Batch 000/156 | Cost: 0.6675
Epoch: 002/100 | Batch 120/156 | Cost: 0.6640
Epoch: 002/100 Train Acc.: 66.05% | Validation Acc.: 66.32%
Time elapsed: 6.15 min
Epoch: 003/100 | Batch 000/156 | Cost: 0.6137
Epoch: 003/100 | Batch 120/156 | Cost: 0.6311
Epoch: 003/100 Train Acc.: 65.82% | Validation Acc.: 63.76%
Time elapsed: 9.21 min
Epoch: 004/100 | Batch 000/156 | Cost: 0.5993
Epoch: 004/100 | Batch 120/156 | Cost: 0.5832
Epoch: 004/100 Train Acc.: 66.75% | Validation Acc.: 64.52%
Time elapsed: 12.27 min
Epoch: 005/100 | Batch 000/156 | Cost: 0.5918
Epoch: 005/100 | Batch 120/156 | Cost: 0.5747
Epoch: 005/100 Train Acc.: 68.29% | Validation Acc.: 67.00%
Time elapsed: 15.33 min
...
```

Parameters vs Hyperparameters

Parameters

- weights (weight parameters)
- biases (bias units)

Hyperparameters

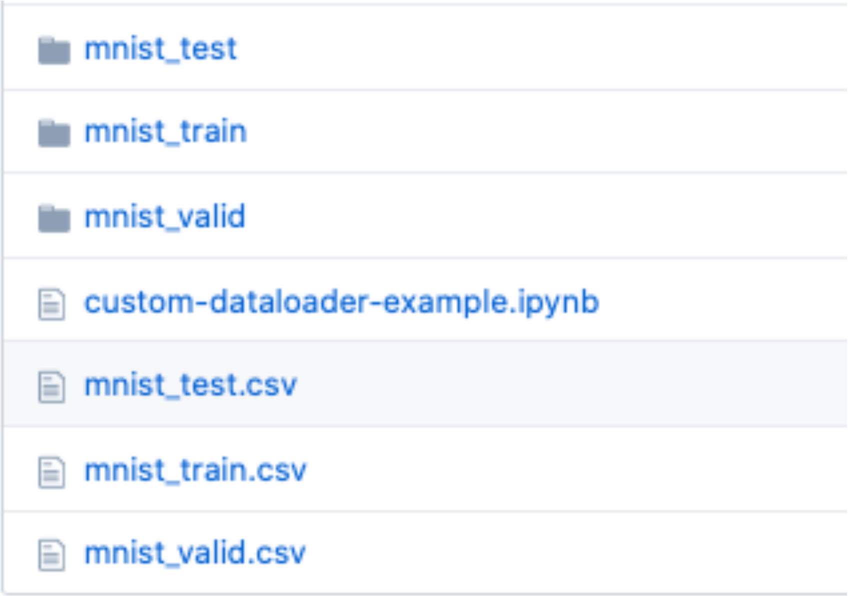
- minibatch size
- data normalization schemes
- number of epochs
- number of hidden layers
- number of hidden units
- learning rates
- (random seed, why?)
- loss function
- various weights (weighting terms)
- activation function types
- regularization schemes (more later)
- weight initialization schemes (more later)
- optimization algorithm type (more later)
- ...

(Mostly no scientific explanation, mostly engineering; need to try many things -> "graduate student descent")

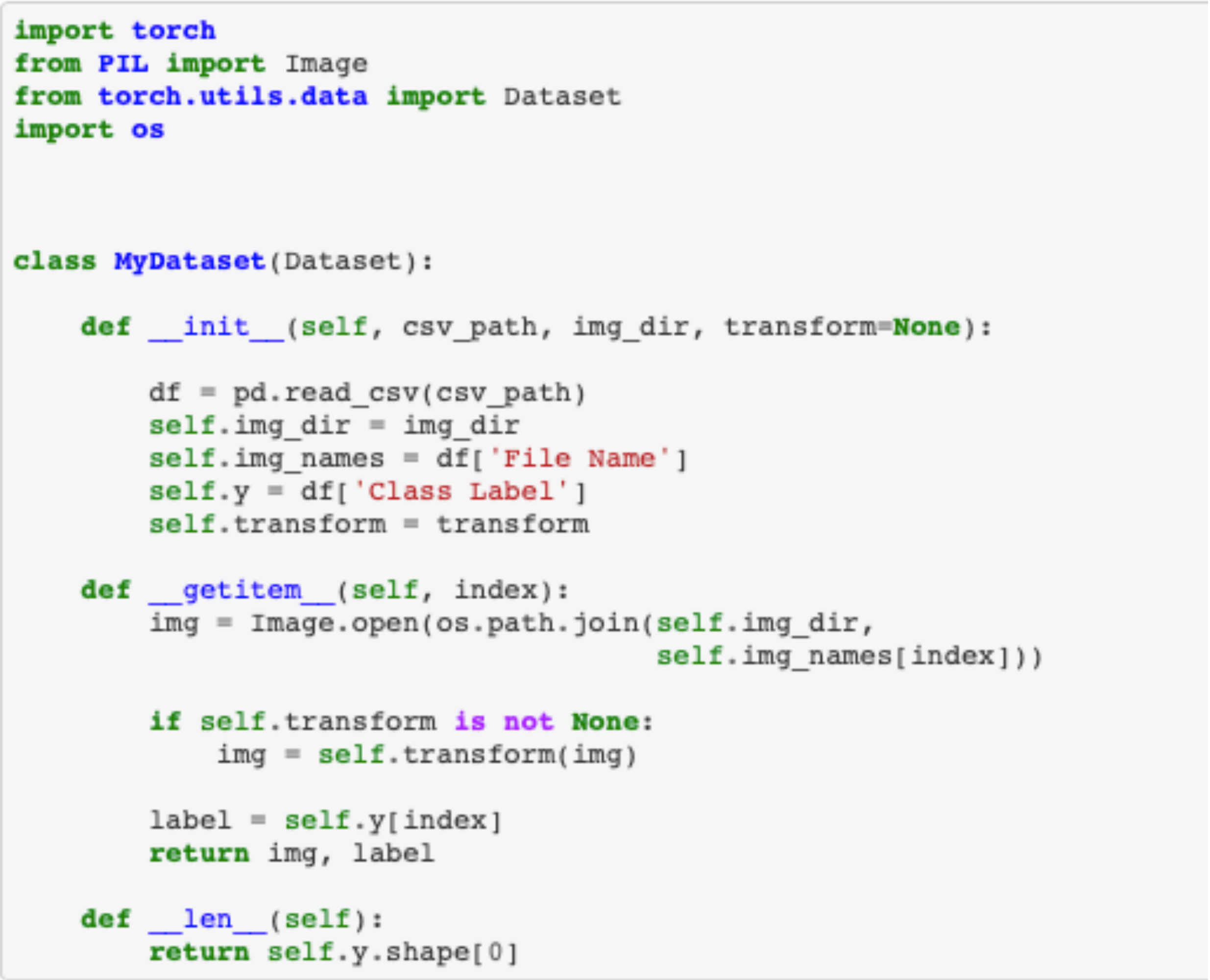
Custom DataLoader Classes ...

- Example showing how you can create your own data loader to efficiently iterate through your own collection of images
  (pretend the MNIST images there are some custom image collection)

  https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L08-mlp/code/custom-dataloader/custom-dataloader-example.ipynb



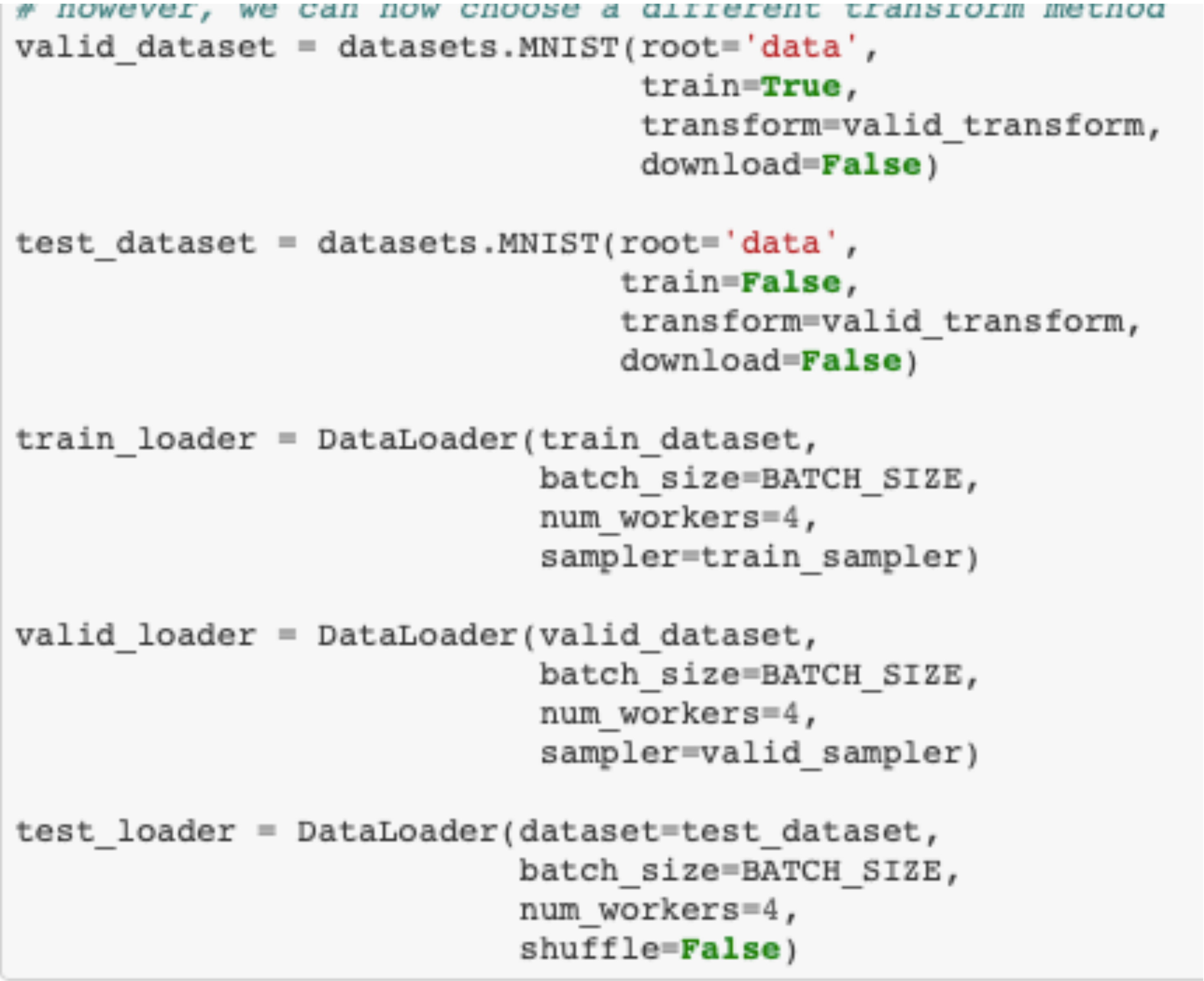
- mnist_test
- mnist_train
- mnist_valid
- custom-dataloader-example.ipynb
- mnist_test.csv
- mnist_train.csv
- mnist_valid.csv



```
import torch
from PIL import Image
from torch.utils.data import Dataset
import os


class MyDataset(Dataset):

    def __init__(self, csv_path, img_dir, transform=None):

        df = pd.read_csv(csv_path)
        self.img_dir = img_dir
        self.img_names = df['File Name']
        self.y = df['Class Label']
        self.transform = transform

    def __getitem__(self, index):
        img = Image.open(os.path.join(self.img_dir,
                                      self.img_names[index]))

        if self.transform is not None:
            img = self.transform(img)

        label = self.y[index]
        return img, label

    def __len__(self):
        return self.y.shape[0]
```

DataLoader with Train/Validation/Test splits

https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L08-mlp/code/mnist-validation-split.ipynb



```
# however, we can now choose a different transform method
valid_dataset = datasets.MNIST(root='data',
                               train=True,
                               transform=valid_transform,
                               download=False)

test_dataset = datasets.MNIST(root='data',
                              train=False,
                              transform=valid_transform,
                              download=False)

train_loader = DataLoader(train_dataset,
                          batch_size=BATCH_SIZE,
                          num_workers=4,
                          sampler=train_sampler)

valid_loader = DataLoader(valid_dataset,
                          batch_size=BATCH_SIZE,
                          num_workers=4,
                          sampler=valid_sampler)

test_loader = DataLoader(dataset=test_dataset,
                         batch_size=BATCH_SIZE,
                         num_workers=4,
                         shuffle=False)
```

Lecture 09

Regularization

STAT 453: Deep Learning, Spring 2020

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching/stat453-ss2020/

Goal: Reduce Overfitting

usually achieved by reducing model capacity and/or reduction of the variance of the predictions (as explained last lecture)

Regularization

In the context of deep learning, regularization can be understood as the process of adding information / changing the objective function to prevent overfitting

Regularization / Regularizing Effects

Goal: reduce overfitting

usually achieved by reducing model capacity and/or reduction of the variance of the predictions (as explained last lecture)

Common Regularization Techniques for DNNs:

- Early stopping
- L_1/L_2 regularization (norm penalties)
- Dropout

Lecture Overview

1. **Avoiding overfitting with more data and data augmentation**

2. Reducing network capacity & early stopping

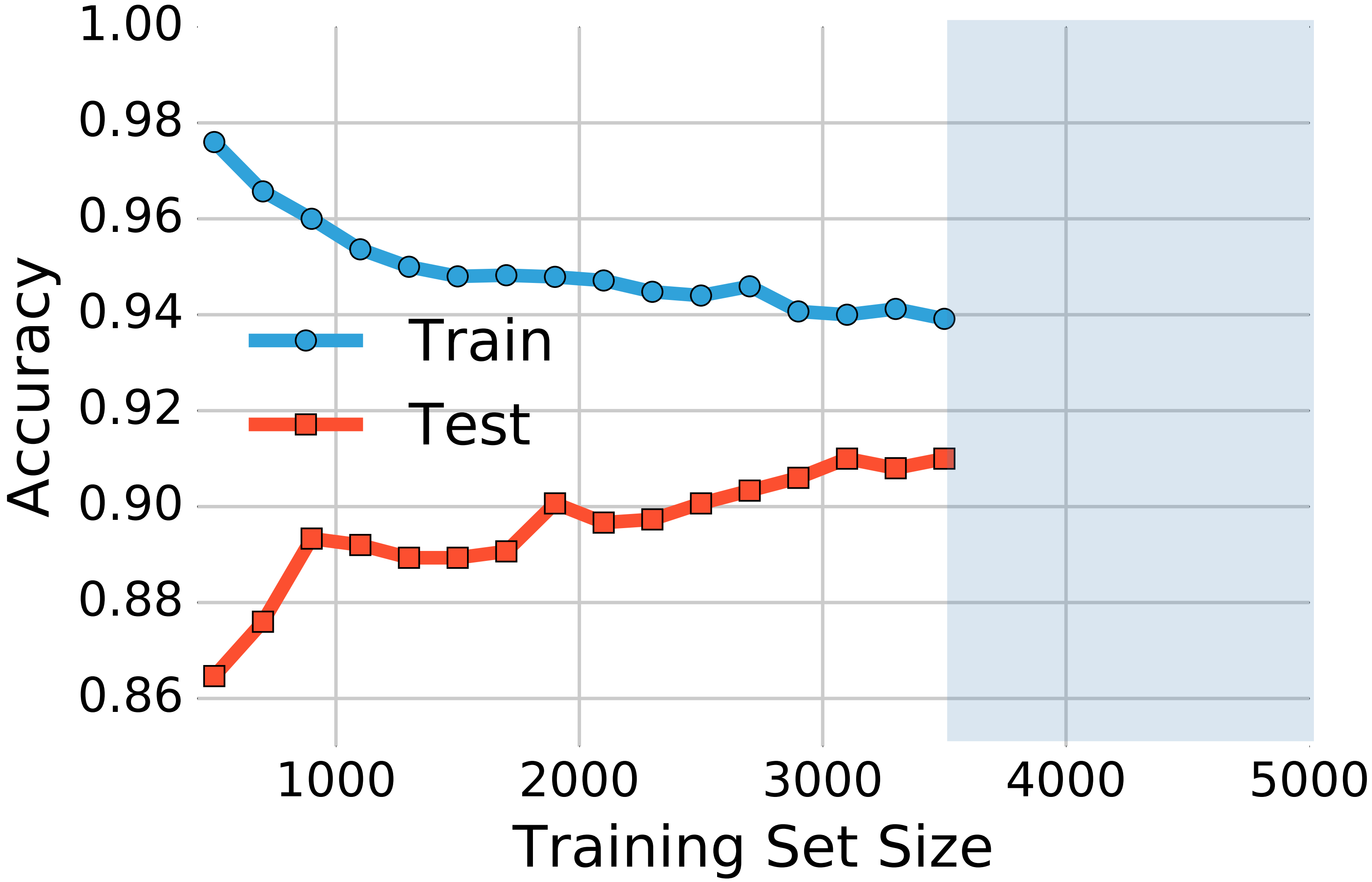
3. Adding norm penalties to the loss: L1 & L2 regularization

4. Dropout

General Strategies to Avoid Overfitting

1. Collecting more data is best & always recommended
2. Data augmentation is also helpful (e.g., for images: random rotation, crop, translation ...)
3. Additionally, reducing the model capacity by reducing the number of parameters or adding regularization (better) helps

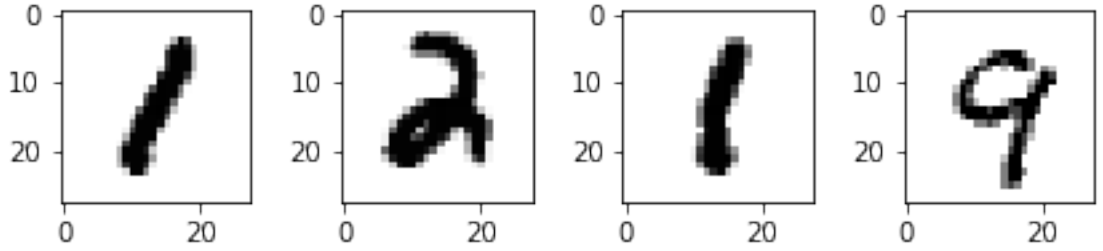
Best Way to Reduce Overfitting is Collecting More Data



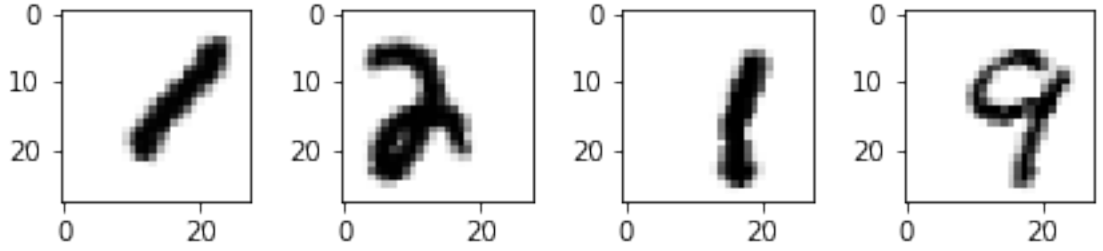
Softmax on MNIST subset (test set size is kept constant)

Data Augmentation in PyTorch via TorchVision

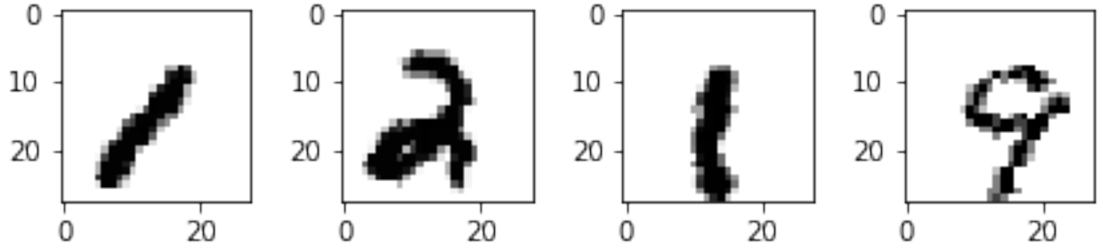
Original



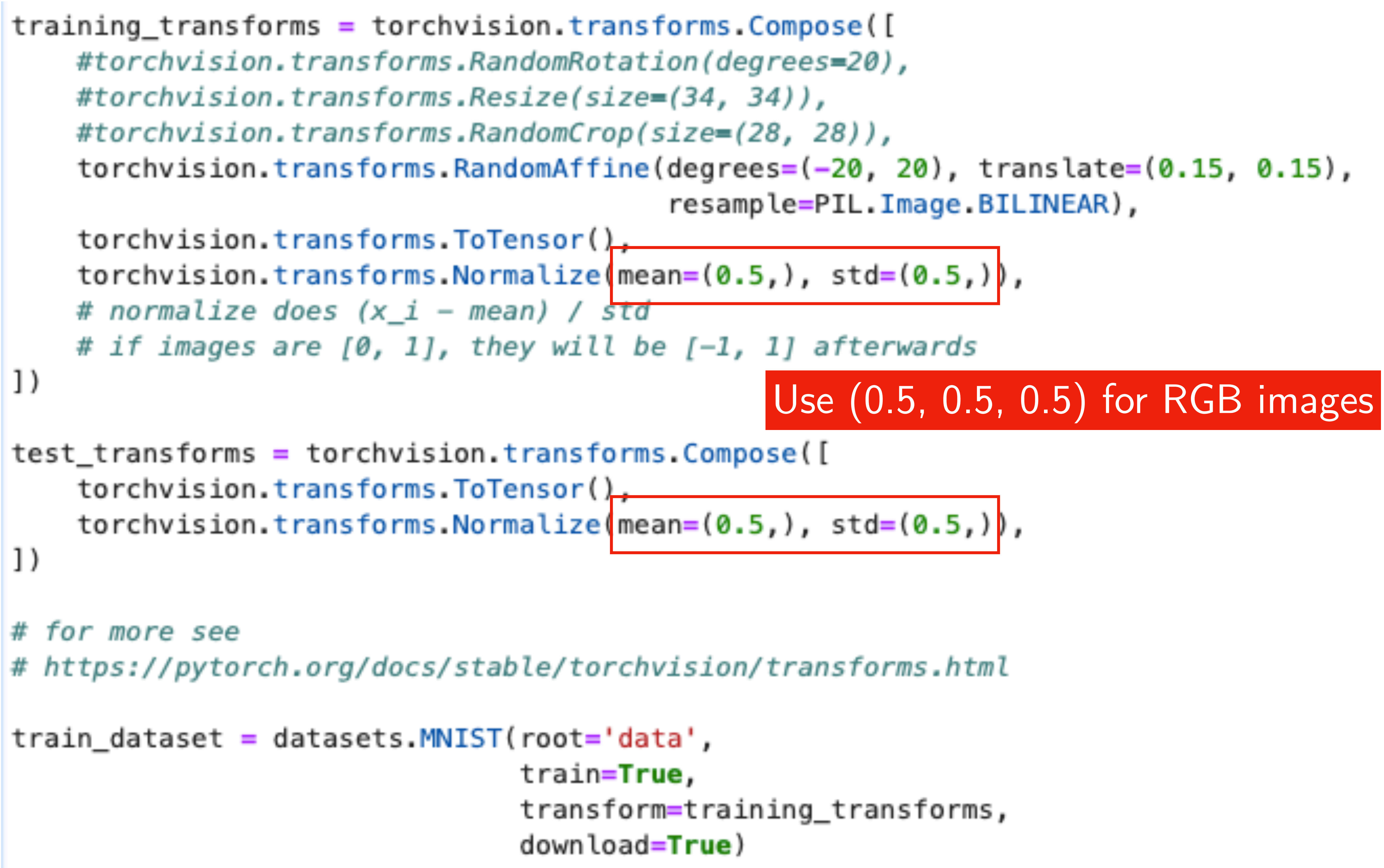
Randomly Augmented



Randomly Augmented
without `resample=PIL.Image.BILINEAR`



https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L09-regularization/code/data-augmentation.ipynb



```
training_transforms = torchvision.transforms.Compose([
    #torchvision.transforms.RandomRotation(degrees=20),
    #torchvision.transforms.Resize(size=(34, 34)),
    #torchvision.transforms.RandomCrop(size=(28, 28)),
    torchvision.transforms.RandomAffine(degrees=(-20, 20), translate=(0.15, 0.15),
                                        resample=PIL.Image.BILINEAR),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.5,), std=(0.5,)),
    # normalize does (x_i - mean) / std
    # if images are [0, 1], they will be [-1, 1] afterwards
])

test_transforms = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.5,), std=(0.5,)),
])

# for more see
# https://pytorch.org/docs/stable/torchvision/transforms.html

train_dataset = datasets.MNIST(root='data',
                               train=True,
                               transform=training_transforms,
                               download=True)
```

Use (0.5, 0.5, 0.5) for RGB images

Other Ways for Dealing with Overfitting if Collecting More Data is not Feasible => Reducing Network's Capacity by Other Means

1. Avoiding overfitting with more data and data augmentation
2. **Reducing network capacity & early stopping**
3. Adding norm penalties to the loss: L1 & L2 regularization
4. Dropout

Other Ways for Dealing with Overfitting if Collecting More Data is not Feasible => Reducing Network's Capacity by Other Means

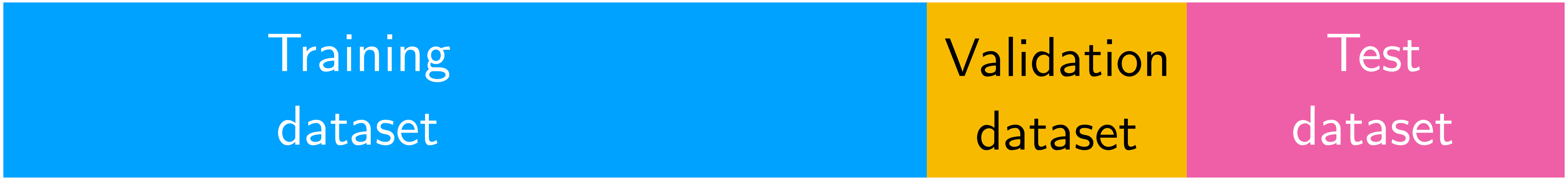
- choose a smaller architecture: fewer hidden layers & units, add dropout, (use ReLU, which can result in "dead activations", add L1 norm penalty)
- enforce smaller weights: Early stopping, L2 norm penalty
- add noise: Dropout

Early Stopping

Step 1: Split your dataset into 3 parts (always recommended)

- use test set only once at the end (for unbiased estimate of generalization performance)
- use validation accuracy for tuning (always recommended)

Dataset



Training dataset

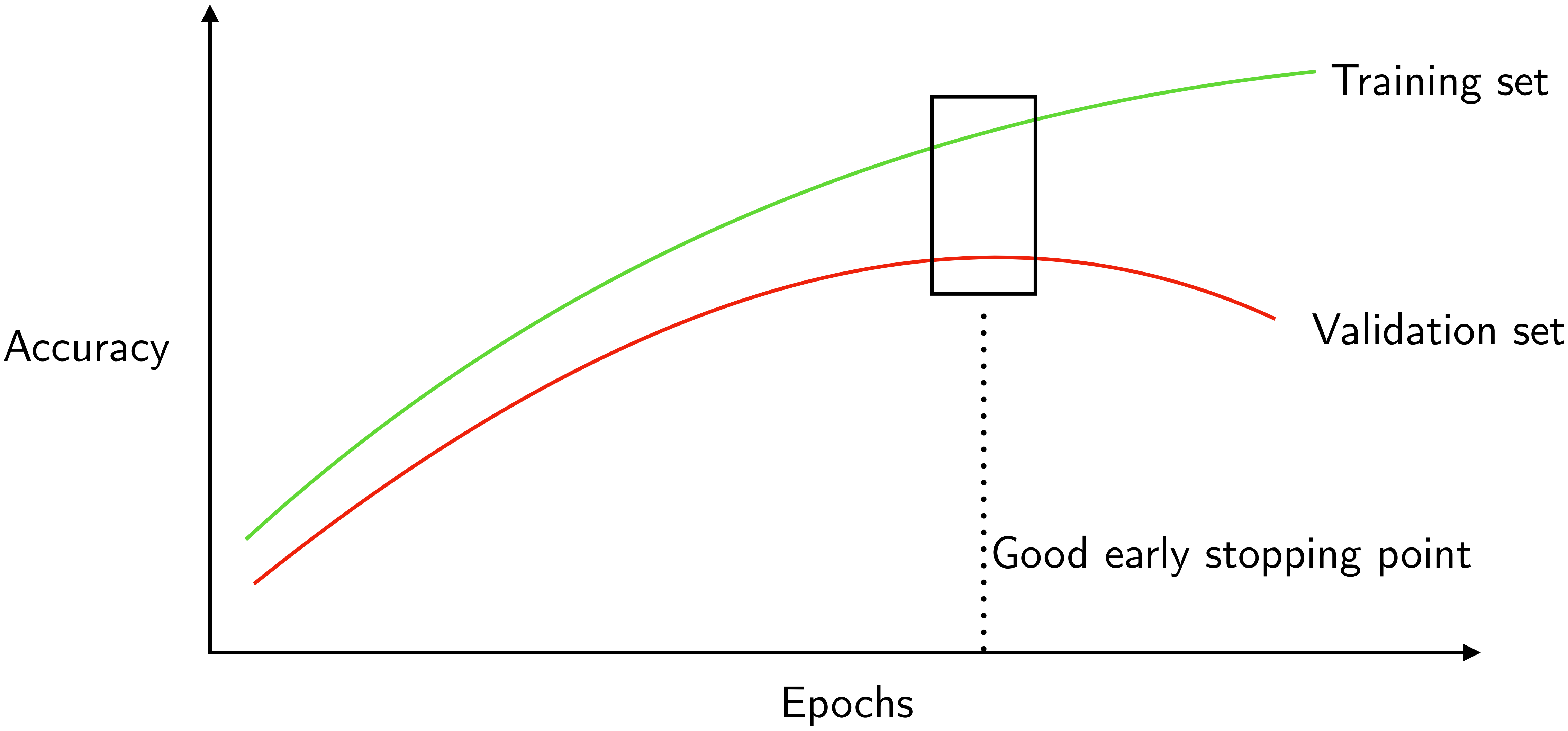
Validation dataset

Test dataset

Early Stopping

Step 2: Early stopping (not very common anymore)

- reduce overfitting by observing the training/validation accuracy gap during training and then stop at the "right" point



1. Avoiding overfitting with more data and data augmentation
2. Reducing network capacity & early stopping
3. **Adding norm penalties to the loss: L1 & L2 regularization**
4. Dropout

L_1/L_2 Regularization

As I am sure you already know it from various statistics classes, we will keep it short:

- L_1-regularization => LASSO regression
- L_2-regularization => Ridge regression (Thikonov regularization)

Basically, a "weight shrinkage" or a "penalty against complexity"

L_1/L_2 Regularization for Linear Models (e.g., Logistic Regression)

$$\text{Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]})$$

$$\text{L2-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{j} w_j^2$$

where: $\sum_j w_j^2 = ||\mathbf{w}||_2^2$

and λ is a hyperparameter

L_2 Regularization for Multilayer Neural Networks

$$\text{L2-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{l=1}^{L}||\mathbf{w}^{(l)}||_F^2$$

sum over layers

where $||\mathbf{w}^{(l)}||_F^2$ is the Frobenius norm (squared):

$$||\mathbf{w}^{(l)}||_F^2 = \sum_{i}\sum_{j}(w_{i,j}^{(l)})^2$$

L_2 Regularization for Neural Nets

Regular gradient descent update:

$$w_{i,j} := w_{i,j} - \eta \frac{\partial \mathcal{L}}{\partial w_{i,j}}$$

Gradient descent update with L2 regularization:

$$w_{i,j} := w_{i,j} - \eta \left(\frac{\partial \mathcal{L}}{\partial w_{i,j}} \boxed{+ \frac{2\lambda}{n} w_{i,j}} \right)$$

L_2 Regularization for Neural Nets in PyTorch

- For all layers, same as before ("automatic approach" via **`weight_decay`**)

- Or, manually:

```
for epoch in range(NUM_EPOCHS):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(DEVICE)
        targets = targets.to(DEVICE)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)

        cost = F.cross_entropy(logits, targets)

        # regularize loss
        L2 = 0.
        for p in model.parameters():
            L2 = L2 + (p**2).sum()
        cost = cost + 2./targets.size(0) * LAMBDA * L2

        optimizer.zero_grad()
        cost.backward()
```

L_2 Regularization for Neural Nets in PyTorch

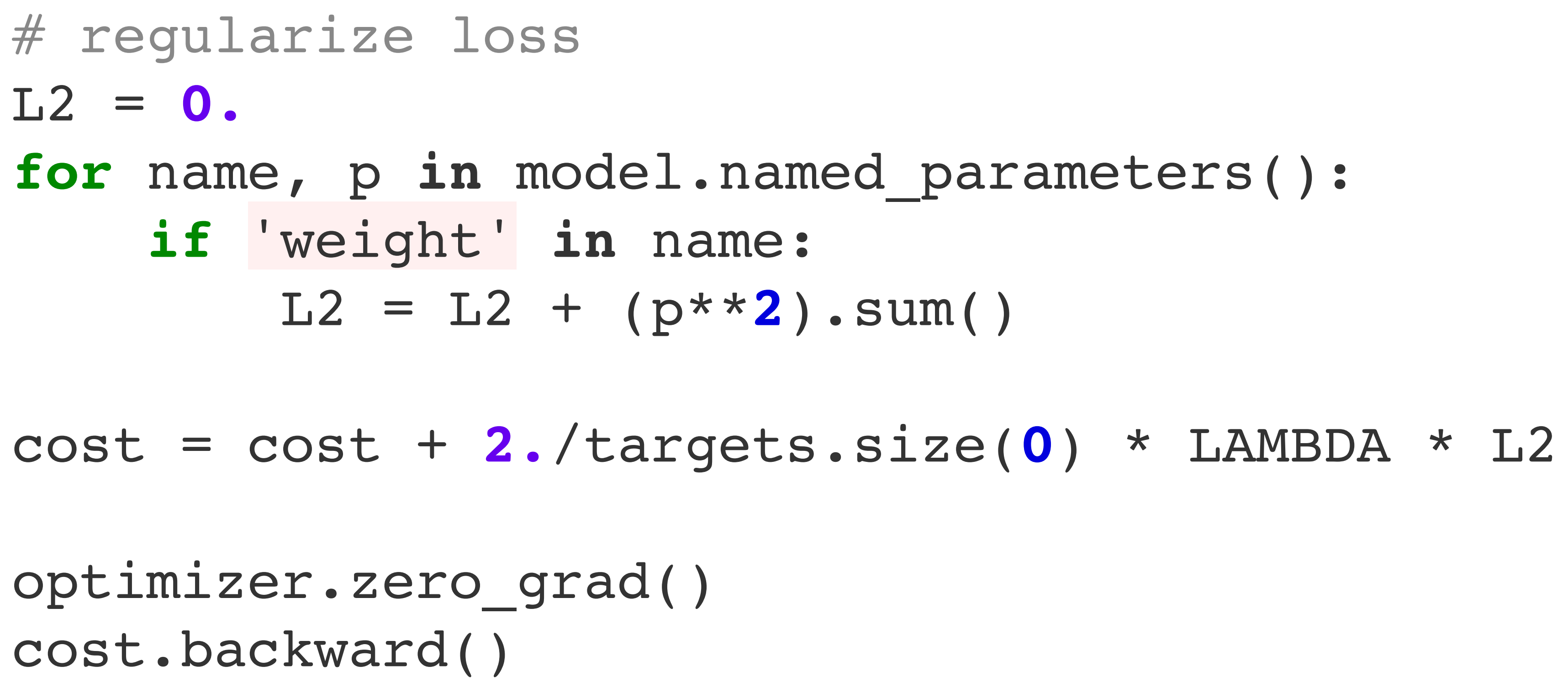
- For all layers, same as before ("automatic approach" via **`weight_decay`**)
- Or, manually:

```
for epoch in range(NUM_EPOCHS):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(DEVICE)
        targets = targets.to(DEVICE)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)

        cost = F.cross_entropy(logits, targets)

        # regularize loss
        L2 = 0.
        for p in model.parameters():
            L2 = L2 + (p**2).sum()
        cost = cost + 2./targets.size(0) * LAMBDA * L2

        optimizer.zero_grad()
        cost.backward()
```

Why did I use "/target.size(0)" here?

L_2 Regularization for Neural Nets in PyTorch

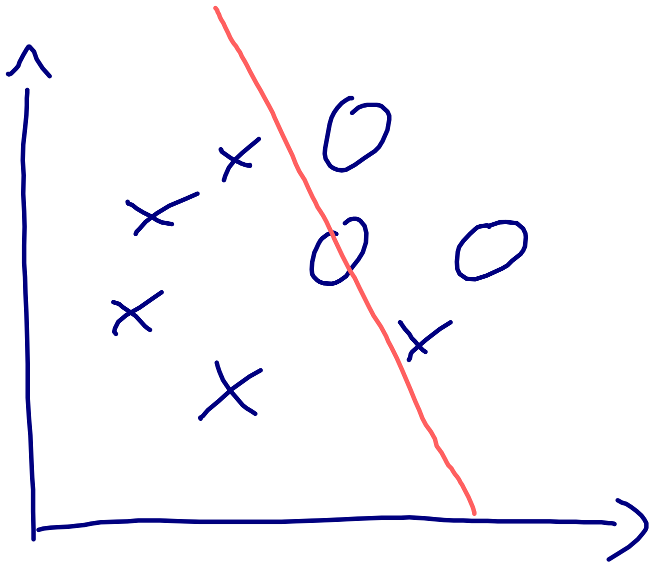
- Or, if you only want to regularize the weights, not the biases:



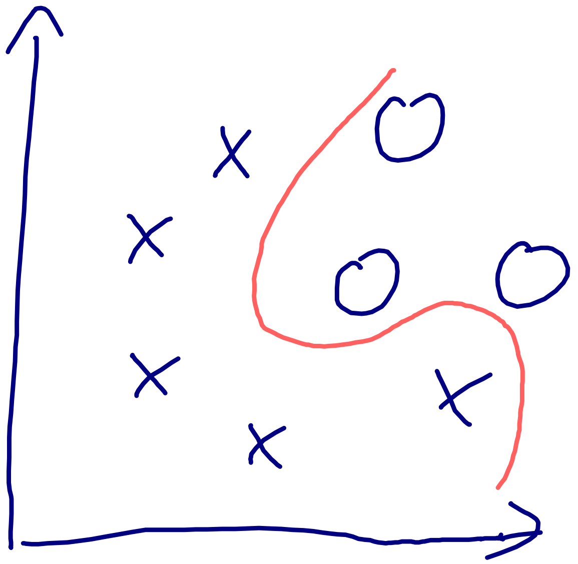
```
# regularize loss
L2 = 0.
for name, p in model.named_parameters():
    if 'weight' in name:
        L2 = L2 + (p**2).sum()

cost = cost + 2./targets.size(0) * LAMBDA * L2

optimizer.zero_grad()
cost.backward()
```

Effect of Norm Penalties on the Decision Boundary

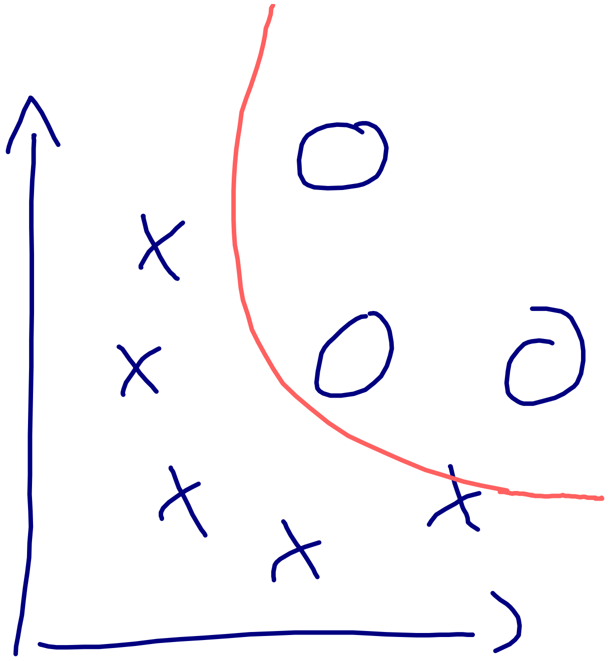
Assume a nonlinear model



Large regularization penalty
=> high bias



Low regularization
=> high variance



Good compromise

Dropout*

*Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929-1958.
http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

1. Avoiding overfitting with more data and data augmentation
2. Reducing network capacity & early stopping
3. Adding norm penalties to the loss: L1 & L2 regularization
4. **Dropout**

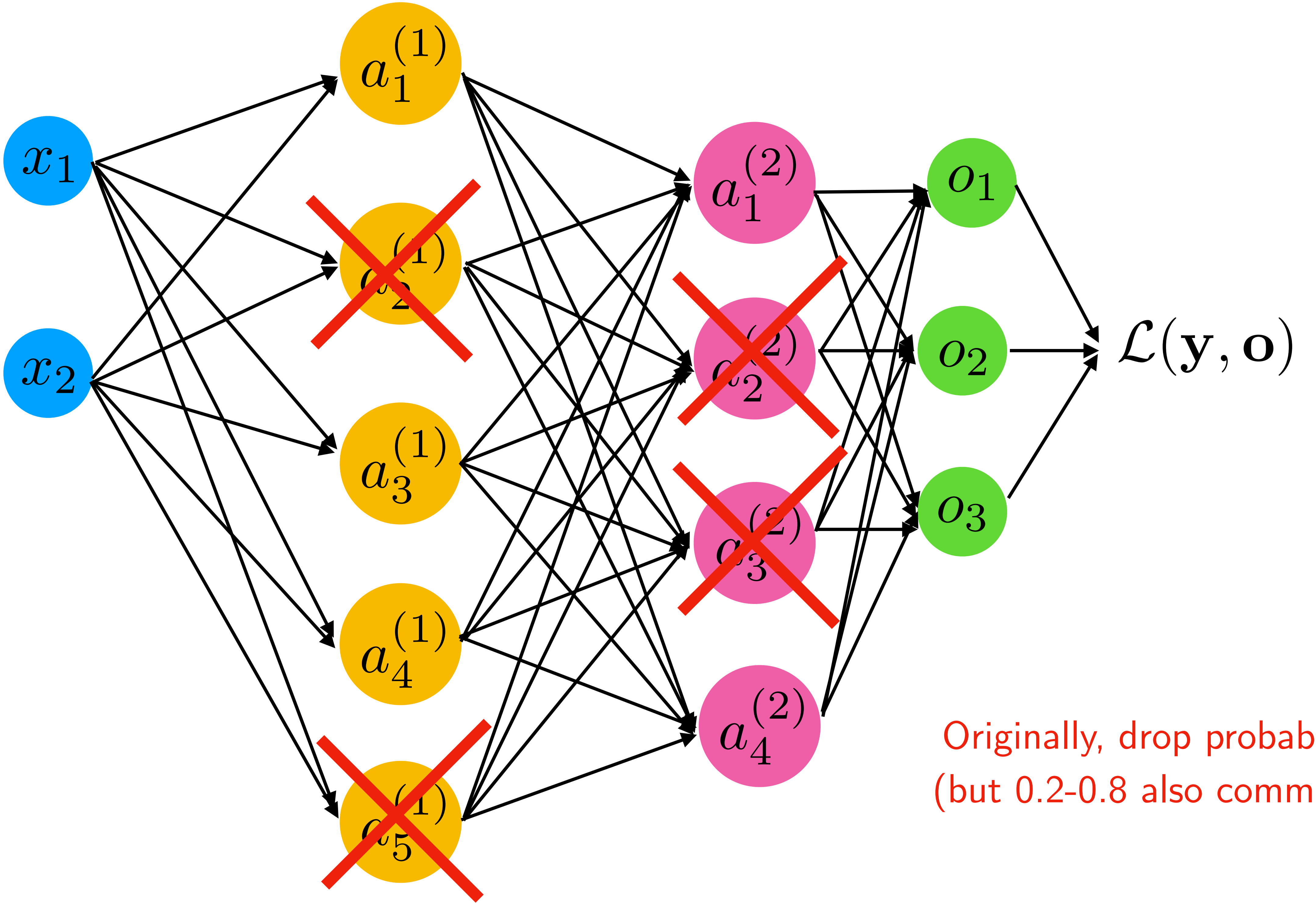
Dropout

Original research articles:

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929-1958.

Dropout in a Nutshell: Dropping Nodes



Originally, drop probability 0.5
(but 0.2-0.8 also common now)

Dropout in a Nutshell: Dropping Nodes

How do we drop the nodes practically/efficiently?

Bernoulli Sampling (during training):

- p := drop probability
- $\mathbf{v}$:= random sample from uniform distribution in range [0, 1]
- $\forall i \in \mathbf{v} : v_i := 0 \text{ if } v_i < p \text{ else } v_i$
- $\mathbf{a} := \mathbf{a} \odot \mathbf{v}$ *(p × 100% of the activations a will be zeroed)*

Dropout in a Nutshell: Dropping Nodes

How do we drop the nodes practically/efficiently?

Bernoulli Sampling (during training):

- $p :=$ drop probability
- $\mathbf{v} :=$ random sample from uniform distribution in range [0, 1]
- $\forall i \in \mathbf{v} : v_i := 0 \text{ if } v_i < p \text{ else } v_i$
- $\mathbf{a} := \mathbf{a} \odot \mathbf{v}$ *(p × 100% of the activations a will be zeroed)*

Then, after training when making predictions (DL jargon: "inference") scale activations via $\mathbf{a} := \mathbf{a} \odot (1 - p)$

Q for you: Why is this required?

Dropout: Co-Adaptation Interpretation

Why does Dropout work well?

- Network will learn not to rely on particular connections too heavily
- Thus, will consider more connections (because it cannot rely on individual ones)
- The weight values will be more spread-out (may lead to smaller weights like with L2 norm)
- Side note: You can certainly use different dropout probabilities in different layers (assigning them proportional to the number of units in a layer is not a bad idea, for example)

Inverted Dropout

- Most frameworks implement inverted dropout
- Here, the activation values are scaled by the factor *(1-p)* during training instead of scaling the activations during "inference"
- I believe Google started this trend (because it's computationally cheaper in the long run if you use your model a lot after training)
- PyTorch's Dropout implementation is also inverted Dropout

Dropout in PyTorch (Functional API)

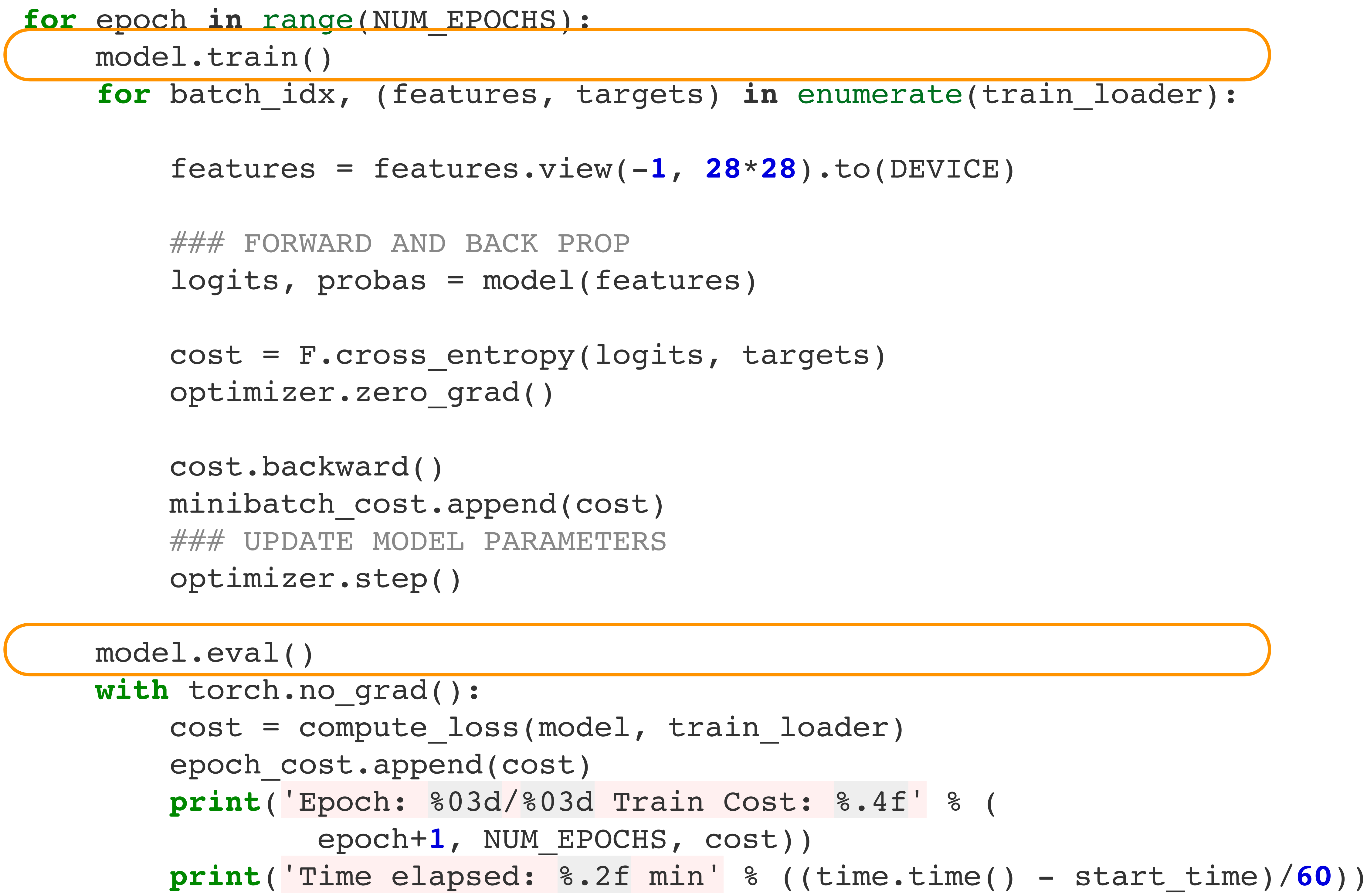
```python
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super(MultilayerPerceptron, self).__init__()

        self.drop_proba = drop_proba
        self.linear_1 = torch.nn.Linear(num_features,
                                        num_hidden_1)

        self.linear_2 = torch.nn.Linear(num_hidden_1,
                                        num_hidden_2)

        self.linear_out = torch.nn.Linear(num_hidden_2,
                                          num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = F.dropout(out, p=self.drop_proba, training=self.training)
        out = self.linear_2(out)
        out = F.relu(out)
        out = F.dropout(out, p=self.drop_proba, training=self.training)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
```

Dropout in PyTorch ([more] Object-Oriented API)

```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super(MultilayerPerceptron, self).__init__()

        self.my_network = torch.nn.Sequential(
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Dropout(drop_proba),
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Dropout(drop_proba),
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

Dropout in PyTorch

Here, is is very important that you use `model.train()` and `model.eval()`!

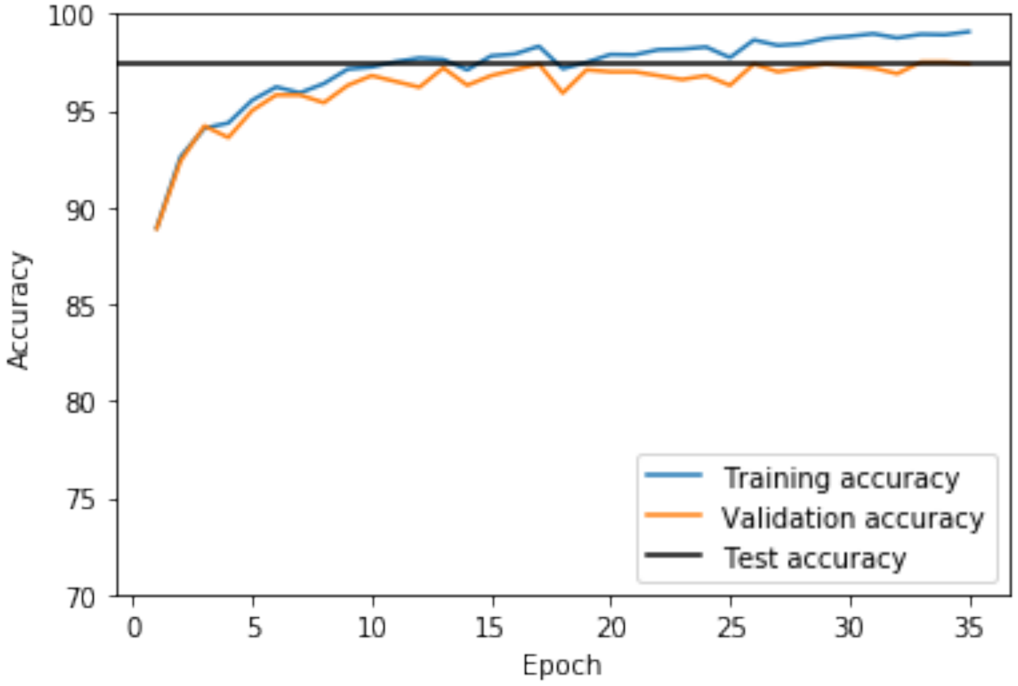


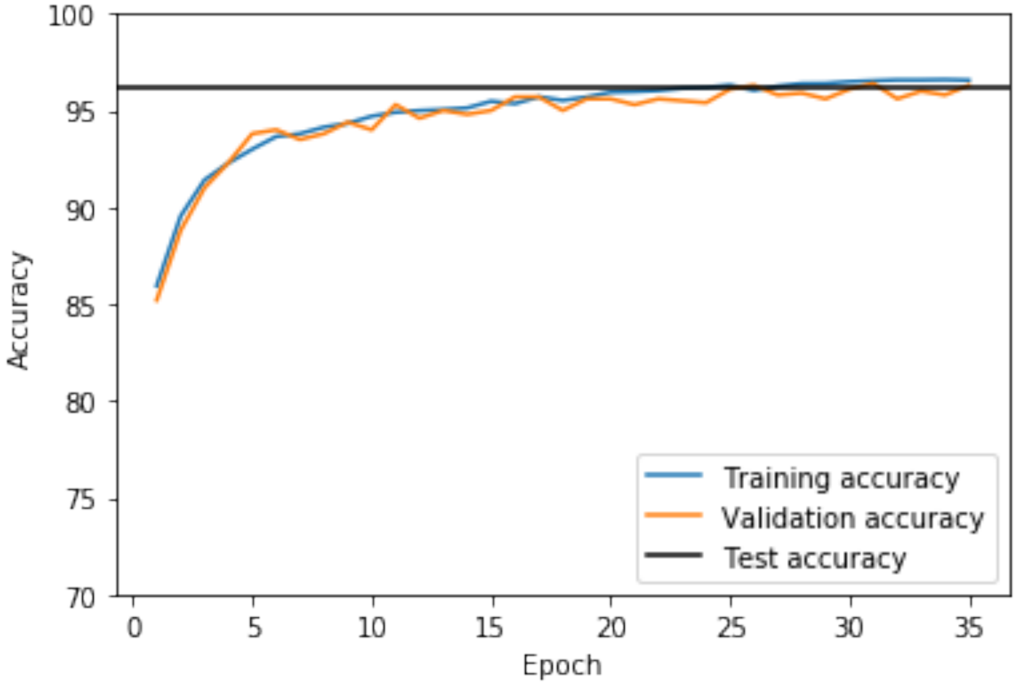
```
for epoch in range(NUM_EPOCHS):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(DEVICE)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)

        cost = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        cost.backward()
        minibatch_cost.append(cost)
        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        cost = compute_loss(model, train_loader)
        epoch_cost.append(cost)
        print('Epoch: %03d/%03d Train Cost: %.4f' % (
                epoch+1, NUM_EPOCHS, cost))
        print('Time elapsed: %.2f min' % ((time.time() - start_time)/60))
```

Dropout in PyTorch (Functional API)

Example implementation of the 3 previous slides:

https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L09-regularization/code/dropout.ipynb





Dropout: More Practical Tips

- Don't use Dropout if your model does not overfit
- However, in that case above, it is then recommended to increase the capacity to make it overfit, and then use dropout to be able to use a larger capacity model (but make it not overfit)

Lecture 10

Feature Normalization and Weight Initialization

STAT 453: Deep Learning, Spring 2020

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching/stat453-ss2020/

Slides:

https://github.com/rasbt/stat453-deep-learning-ss20/10_norm-and-init/

"Tricks" for Improving *Deep* Neural Network Training

Today:

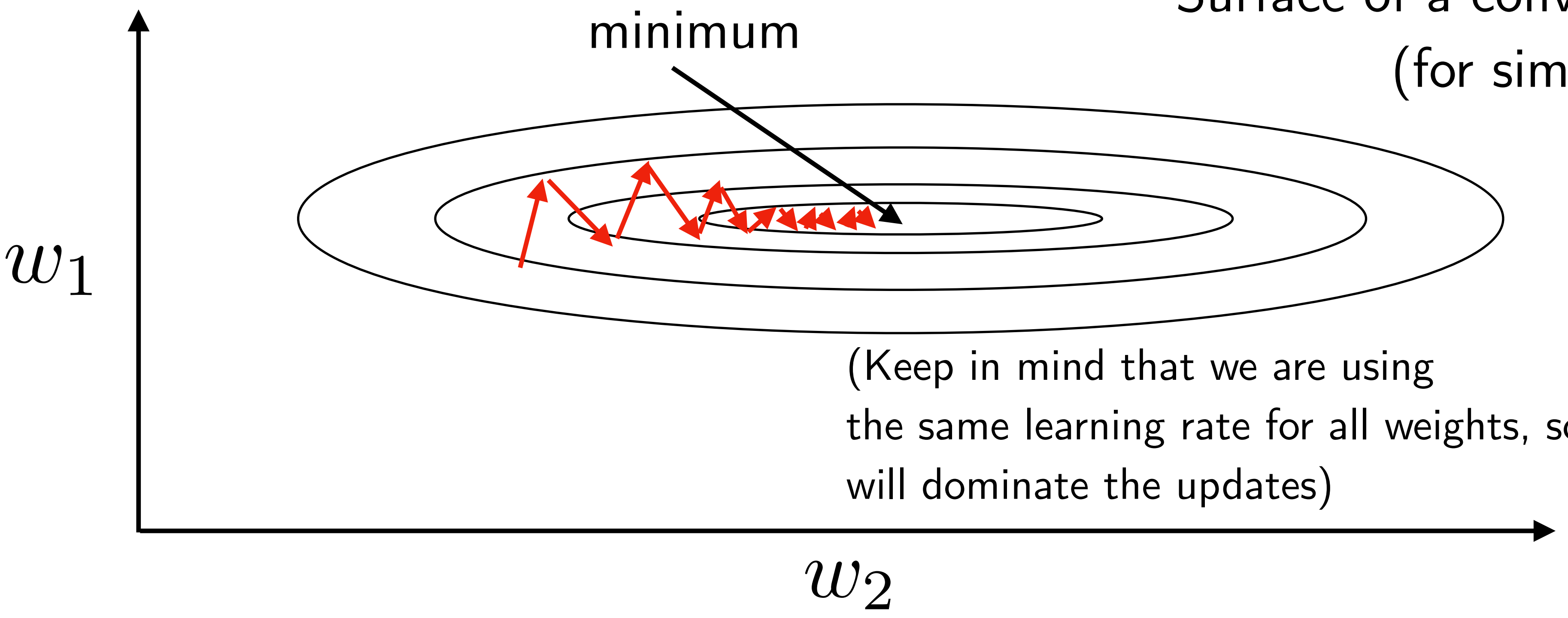
1. Feature/Input Normalization
   (BatchNorm, InstanceNorm, GroupNorm, LayerNorm)
2. Weight Initialization (Xavier Glorot, Kaiming He)

Next Lecture:

3. Optimization Algorithms (RMSProp, Adagrad, ADAM)

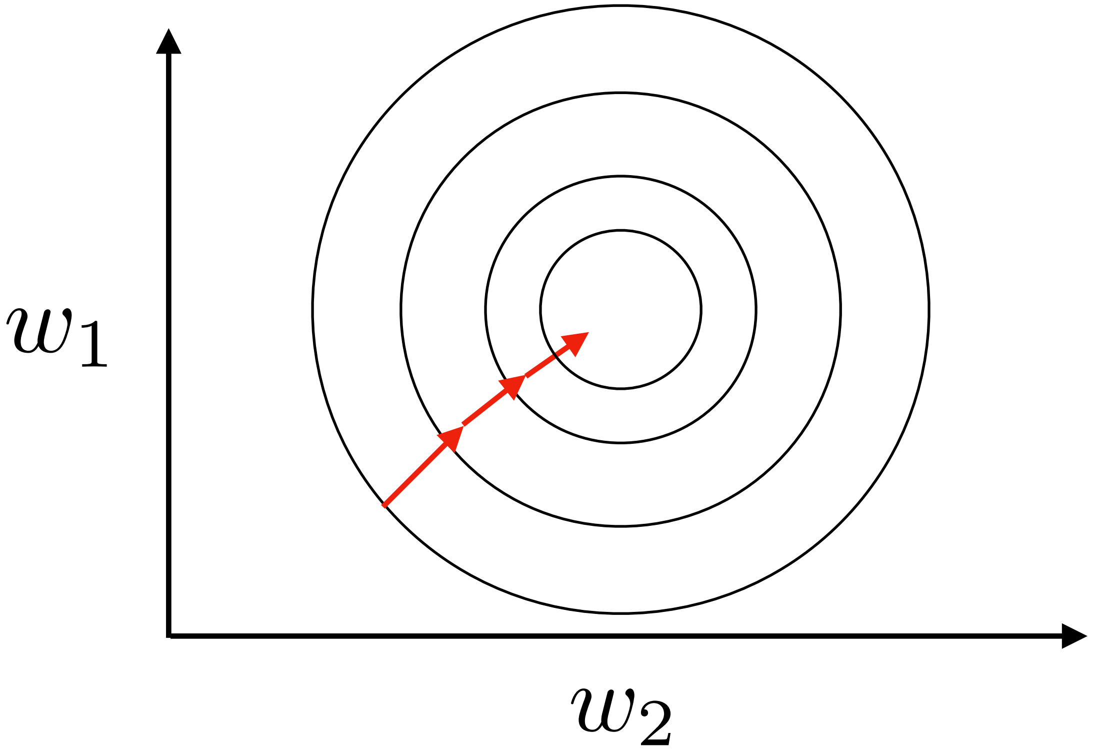
Part 1: Input Normalization

Recap: Why We Normalize Inputs for Gradient Descent



Surface of a convex cost function
(for simplicity)

(Keep in mind that we are using
the same learning rate for all weights, so large parameters
will dominate the updates)



"Standardization" of input features

$$x_j'^{[i]} = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$$

(scaled feature will have
zero mean, unit variance)

However, normalizing

the inputs to the network

only affects the first hidden layer ...

What about the other hidden layers?

Batch Normalization ("BatchNorm")

Ioffe, S., & Szegedy, C. (2015, June). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448-456).

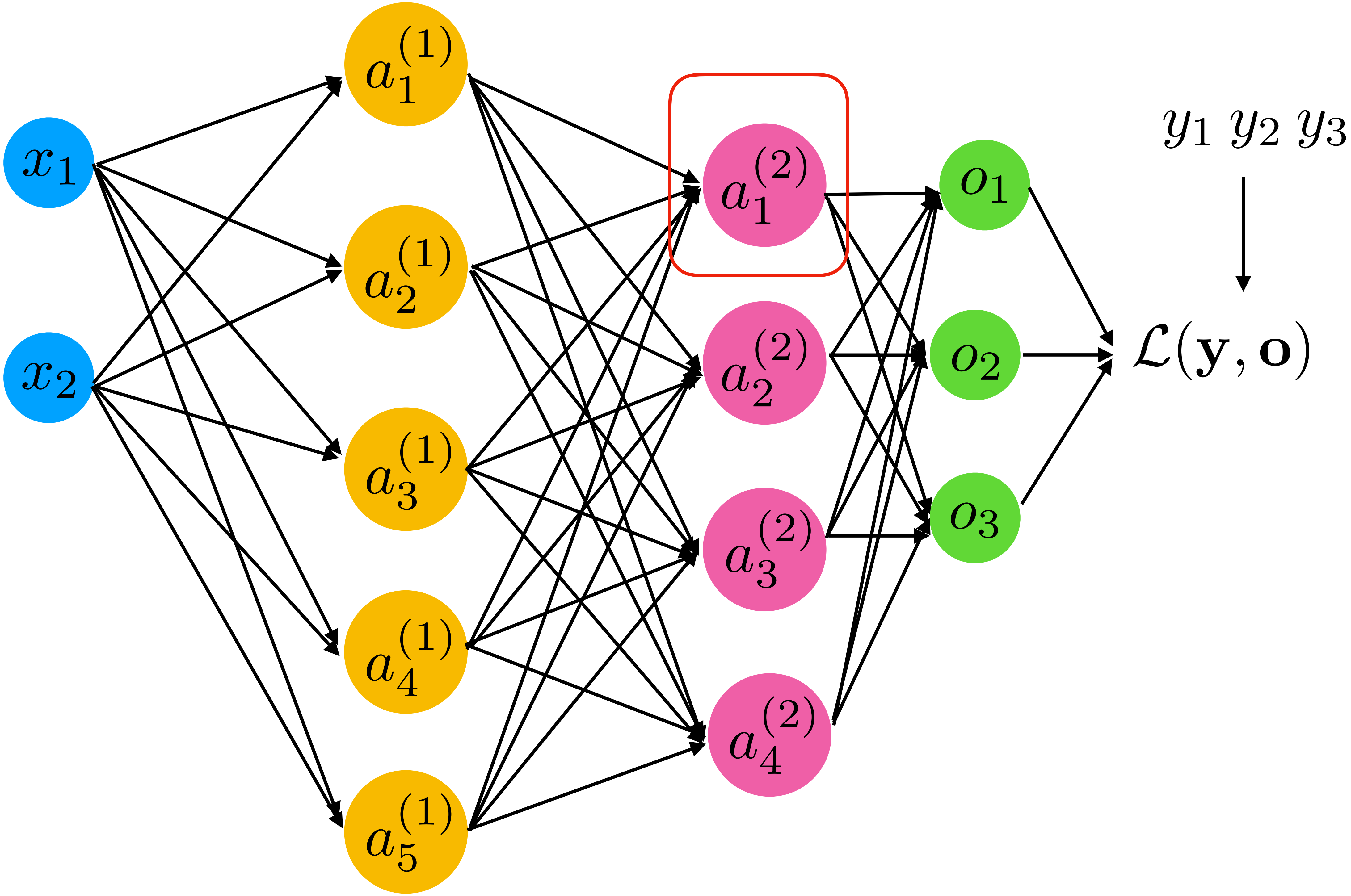
http://proceedings.mlr.press/v37/ioffe15.html

Batch Normalization ("BatchNorm")

- Normalizes hidden layer inputs
- Helps with exploding/vanishing gradient problems
- Can increase training stability and convergence rate
- Can be understood as additional (normalization) layers (with additional parameters)

Suppose, we have net input $z_1^{(2)}$

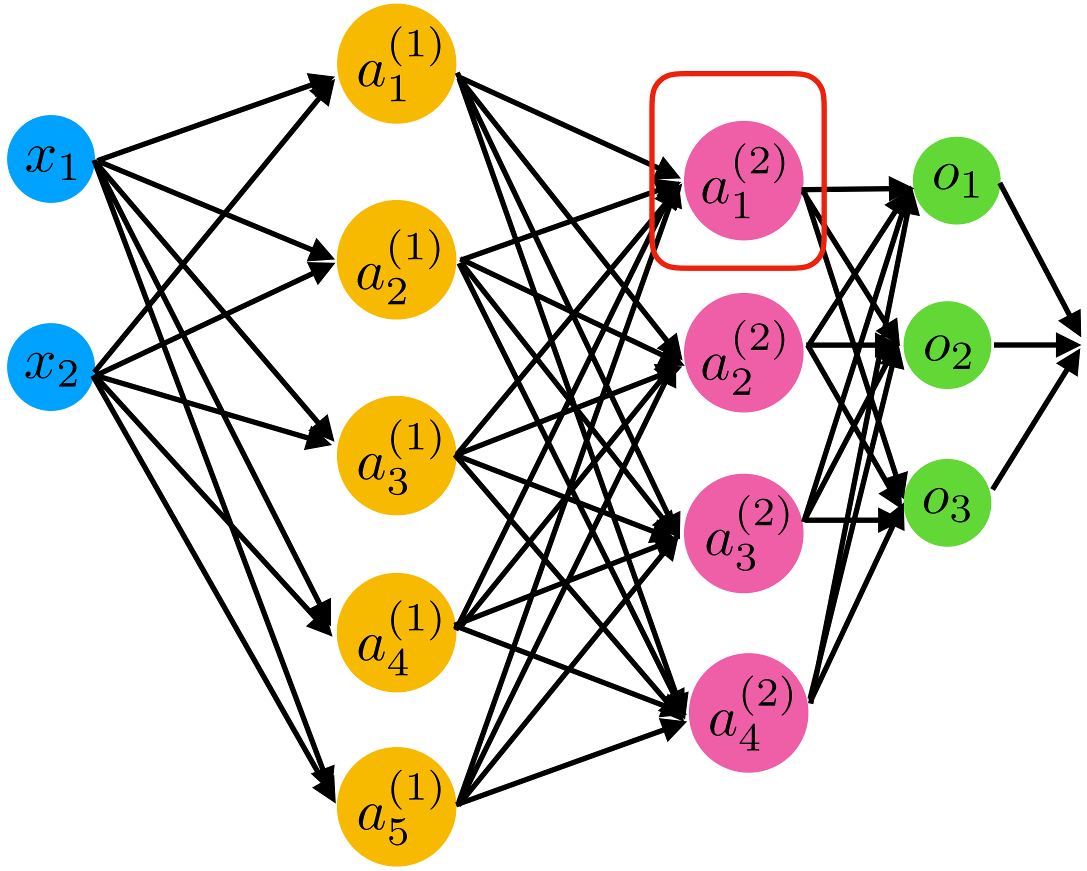
associated with an activation in the 2nd hidden layer



Now, consider all examples in a minibatch such that the net input of a given training example

at layer 2 is written as $z_1^{(2)[i]}$

where $i \in \{1, ..., n\}$



In the next slides, let's omit the layer index, as it may be distracting...

BatchNorm Step 1: Normalize Net Inputs

$$\mu_j = \frac{1}{n} \sum_i z_j^{[i]}$$

$$\sigma_j^2 = \frac{1}{n} \sum_i (z_j^{[i]} - \mu_j)^2$$

$${z'}_j^{[i]} = \frac{z_j^{[i]} - \mu_j}{\sigma_j}$$

BatchNorm Step 1: Normalize Net Inputs

$$\mu_j = \frac{1}{n} \sum_i z_j^{[i]}$$

$$\sigma_j^2 = \frac{1}{n} \sum_i (z_j^{[i]} - \mu_j)^2$$

$$z'^{[i]}_j = \frac{z_j^{[i]} - \mu_j}{\sigma_j}$$

In practice:

$$z'^{[i]}_j = \frac{z_j^{[i]} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

For numerical stability, where epsilon is a small number like 1E-5

BatchNorm Step 2: Pre-Activation Scaling

$$z'^{[i]}_j = \frac{z^{[i]}_j - \mu_j}{\sqrt{\sigma^2_j + \epsilon}}$$

$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

These are learnable parameters

BatchNorm Step 2: Pre-Activation Scaling

$${z'}_j^{[i]} = \frac{z_j^{[i]} - \mu_j}{\sigma_j}$$

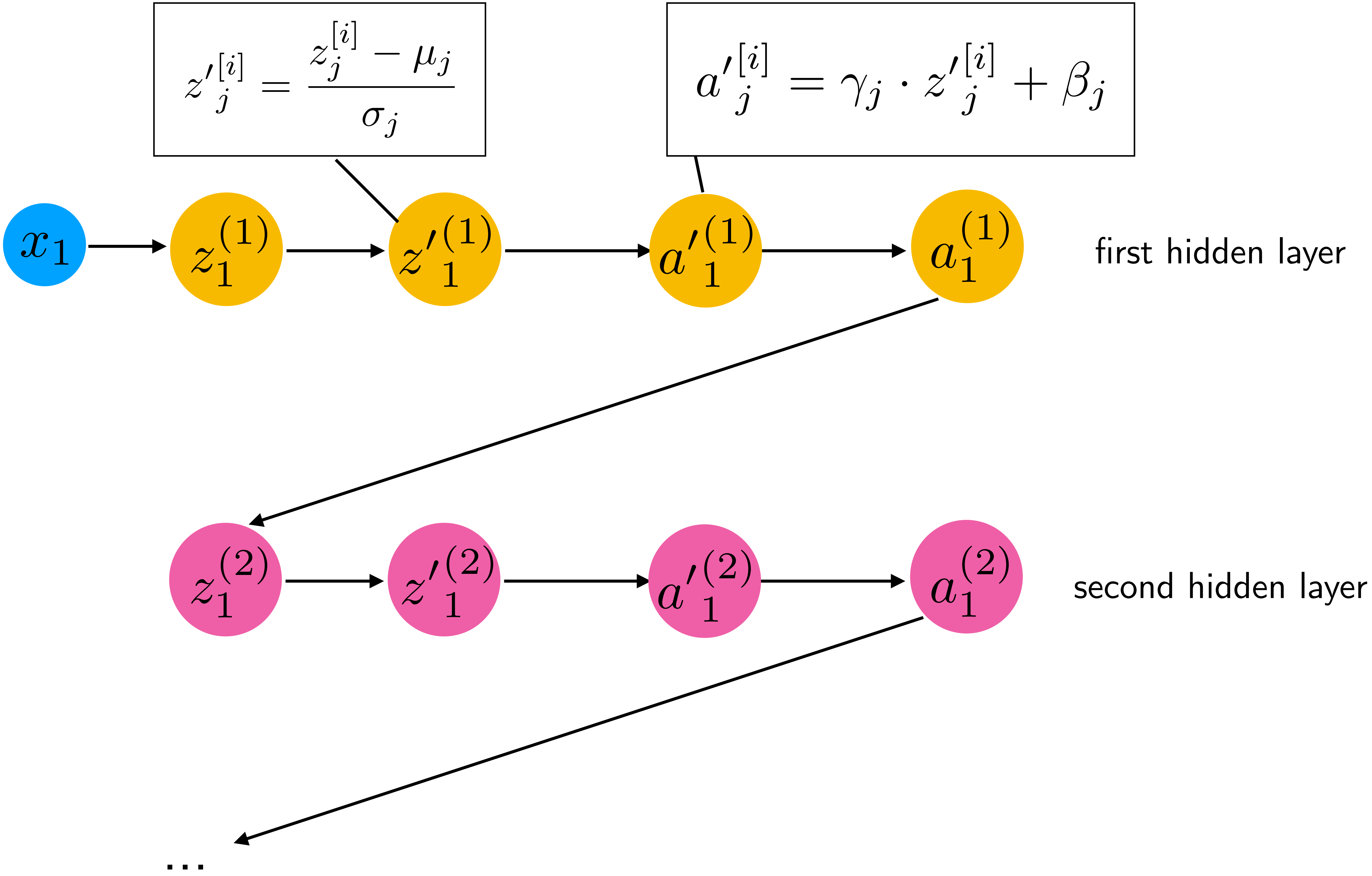
$${a'}_j^{[i]} = \gamma_j \cdot {z'}_j^{[i]} + \beta_j$$

Controls the mean

Controls the spread or scale

Technically, a BatchNorm layer could learn to perform "standardization" with zero mean and unit variance

BatchNorm Step 1 & 2 Summarized



$$z'^{[i]}_j = \frac{z^{[i]}_j - \mu_j}{\sigma_j}$$

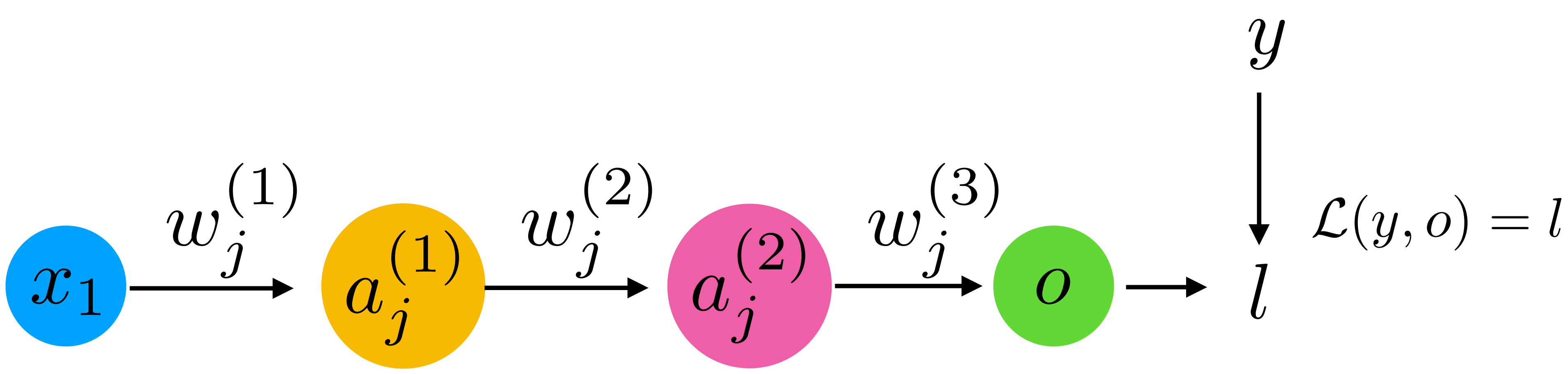
$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

first hidden layer

second hidden layer

Backpropagation for BatchNorm Parameters

Let's consider a simpler case ...

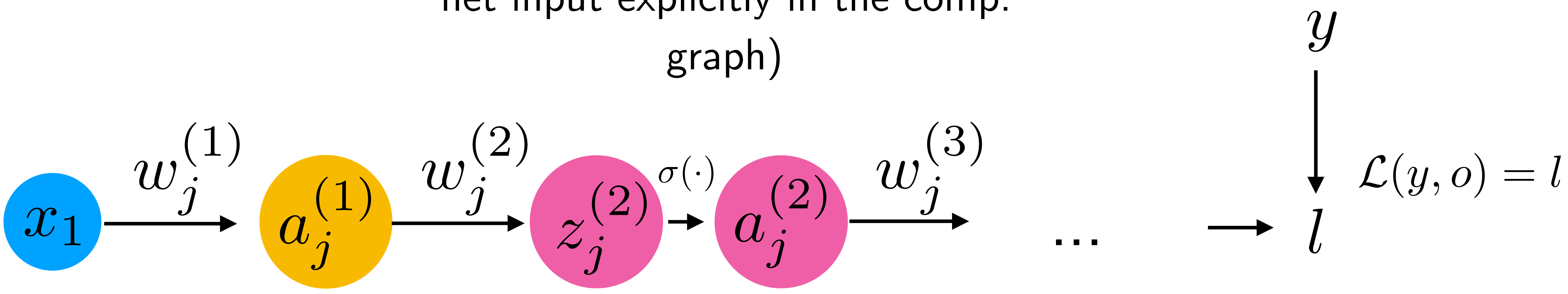


$$\frac{\partial l}{\partial w_j^{(3)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial w_j^{(3)}}$$

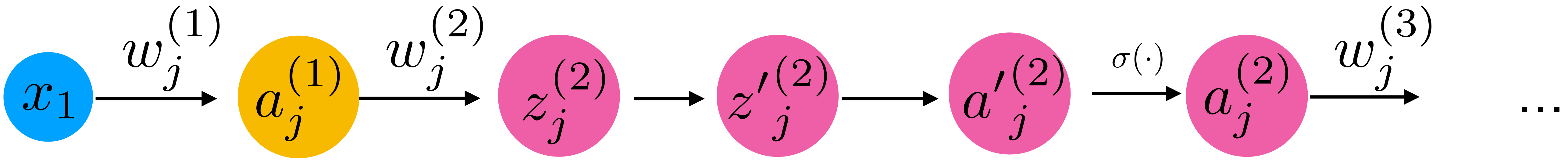
$$\frac{\partial l}{\partial w_j^{(2)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_j^{(2)}} \cdot \frac{\partial a_j^{(2)}}{\partial w_j^{(2)}}$$

$$\frac{\partial l}{\partial w_j^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_j^{(2)}} \cdot \frac{\partial a_j^{(2)}}{\partial a_j^{(1)}} \cdot \frac{\partial a_j^{(1)}}{\partial w_j^{(1)}}$$

(previously, we didn't write the net input explicitly in the comp. graph)



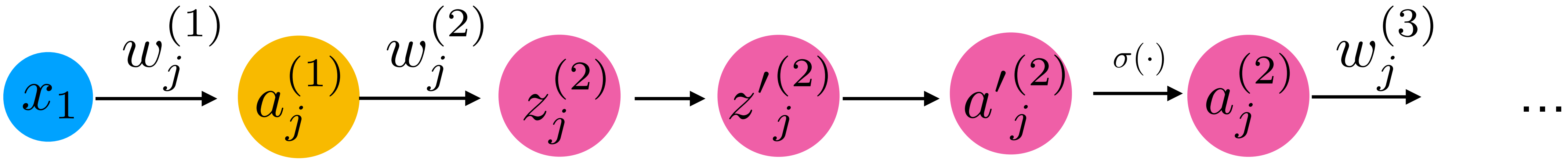
Adding a BatchNorm layer ...



$$z'^{(2)}_j = \frac{z_j^{(2)} - \mu_j}{\sigma_j} \qquad a'^{(2)}_j = \gamma_j \cdot z'^{(2)}_j + \beta_j$$

Backprop for BatchNorm Parameters

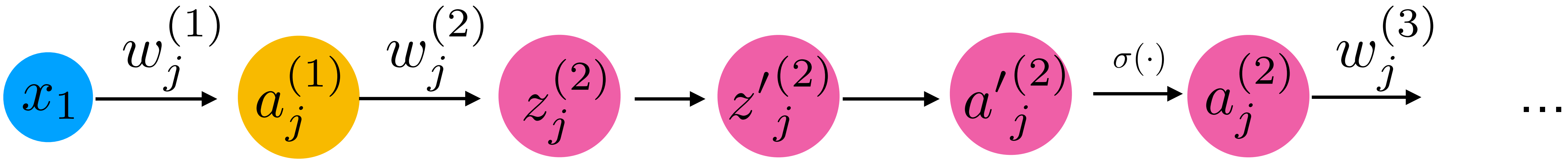
$$z'^{(2)}_j = \frac{z^{(2)}_j - \mu_j}{\sigma_j} \qquad\qquad a'^{(2)}_j = \gamma_j \cdot z'^{(2)}_j + \beta_j$$



$$\frac{\partial l}{\partial \beta_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j} \cdot \frac{\partial a'^{(2)[i]}_j}{\partial \beta_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j}$$

$$\frac{\partial l}{\partial \gamma_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j} \cdot \frac{\partial a'^{(2)[i]}_j}{\partial \gamma_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j} \cdot z'^{(2)[i]}_j$$

Backprop Beyond the BatchNorm Layer



Since the minibatch mean and variance act as parameters, we can/have to apply the multivariable chain rule

$$\frac{\partial l}{\partial z_j^{(2)[i]}} = \frac{\partial l}{\partial {z'}_j^{(2)[i]}} \cdot \frac{\partial {z'}_j^{(2)[i]}}{\partial z_j^{(2)[i]}} + \frac{\partial l}{\partial \mu_j} \cdot \frac{\partial \mu_j}{\partial z_j^{(2)[i]}} + \frac{\partial l}{\partial \sigma_j^2} \cdot \frac{\partial \sigma_j^2}{\partial z_j^{(2)[i]}}$$

$$= \frac{\partial l}{\partial {z'}_j^{(2)[i]}} \cdot \frac{1}{\sigma_j} + \frac{\partial l}{\partial \mu_j} \cdot \frac{1}{n} + \frac{\partial l}{\partial \sigma_j^2} \cdot \frac{2(z_j^{(2)} - \mu_j)}{n}$$

BatchNorm in PyTorch

```python
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_hidden_1)
        self.linear_1_bn = torch.nn.BatchNorm1d(num_hidden_1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1, num_hidden_2)
        self.linear_2_bn = torch.nn.BatchNorm1d(num_hidden_2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        # note that batchnorm is in the classic
        # sense placed before the activation
        out = self.linear_1_bn(out)
        out = F.relu(out)

        out = self.linear_2(out)
        out = self.linear_2_bn(out)
        out = F.relu(out)

        logits = self.linear_out(out)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

BatchNorm in PyTorch

```python
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_hidden_1)
        self.linear_1_bn = torch.nn.BatchNorm1d(num_hidden_1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1, num_hidden_2)
        self.linear_2_bn = torch.nn.BatchNorm1d(num_hidden_2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        # note that batchnorm is in the classic
        # sense placed before the activation
        out = self.linear_1_bn(out)
        out = F.relu(out)

        out = self.linear_2(out)
        out = self.linear_2_bn(out)
        out = F.relu(out)

        logits = self.linear_out(out)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

don't forget `model.train()`
and `model.eval()`
in training and test loops

BatchNorm During Prediction ("Inference")

- Use exponentially weighted average (moving average) of mean and variance

```
running_mean = momentum * running_mean
               + (1 - momentum) * sample_mean
```

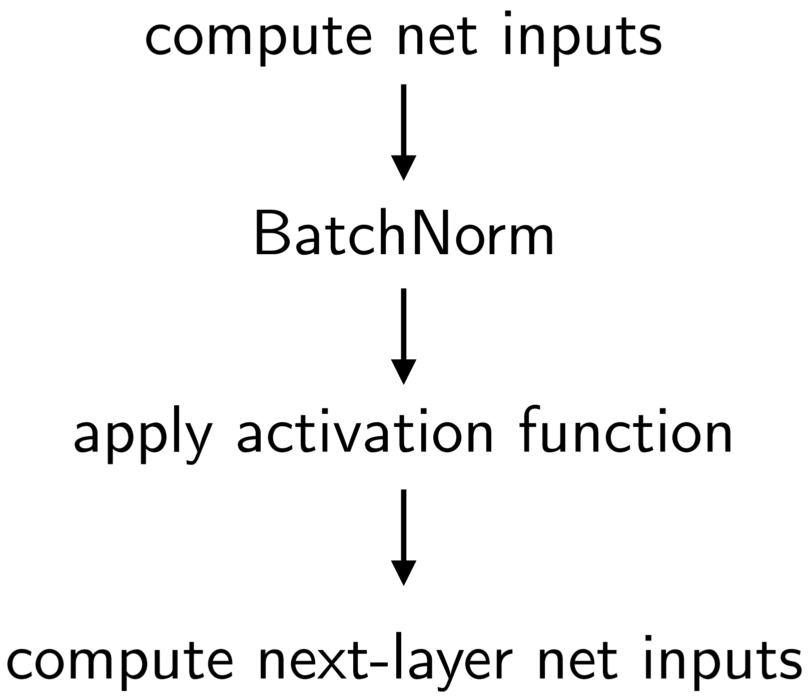
(where momentum is typically ~0.1; and same for variance)

- Alternatively, can also use global training set mean and variance

BatchNorm Variants

Pre-Activation

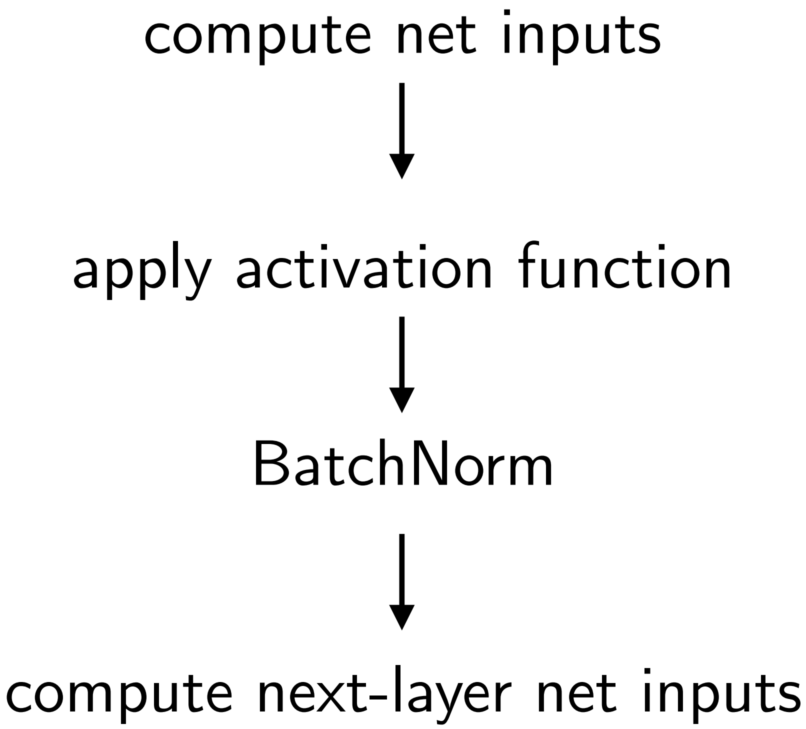
"Original" version
as discussed in
previous slides



compute net inputs
↓
BatchNorm
↓
apply activation function
↓
compute next-layer net inputs

Post-Activation

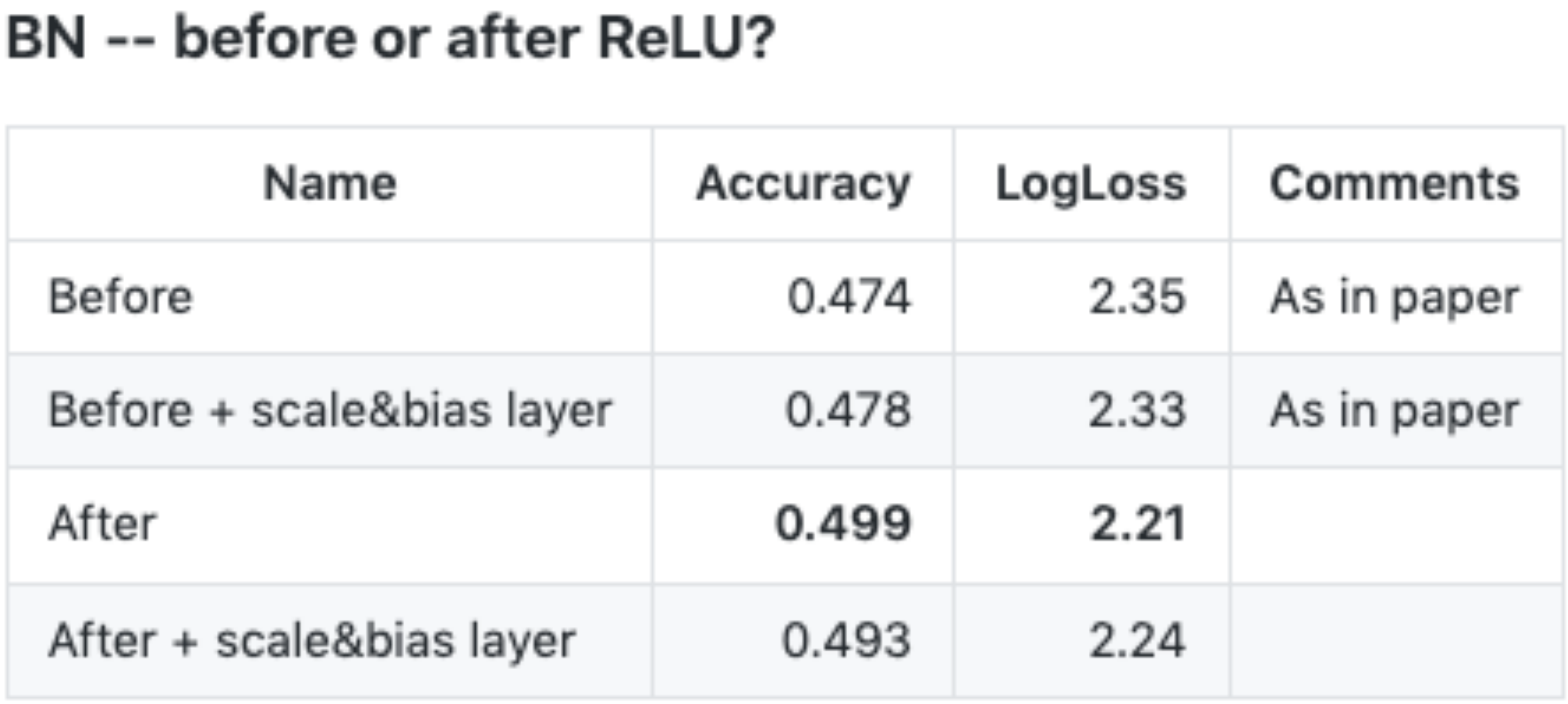
May make more sense,
but less common



compute net inputs
↓
apply activation function
↓
BatchNorm
↓
compute next-layer net inputs

Some Benchmarks

https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md#bn----before-or-after-relu

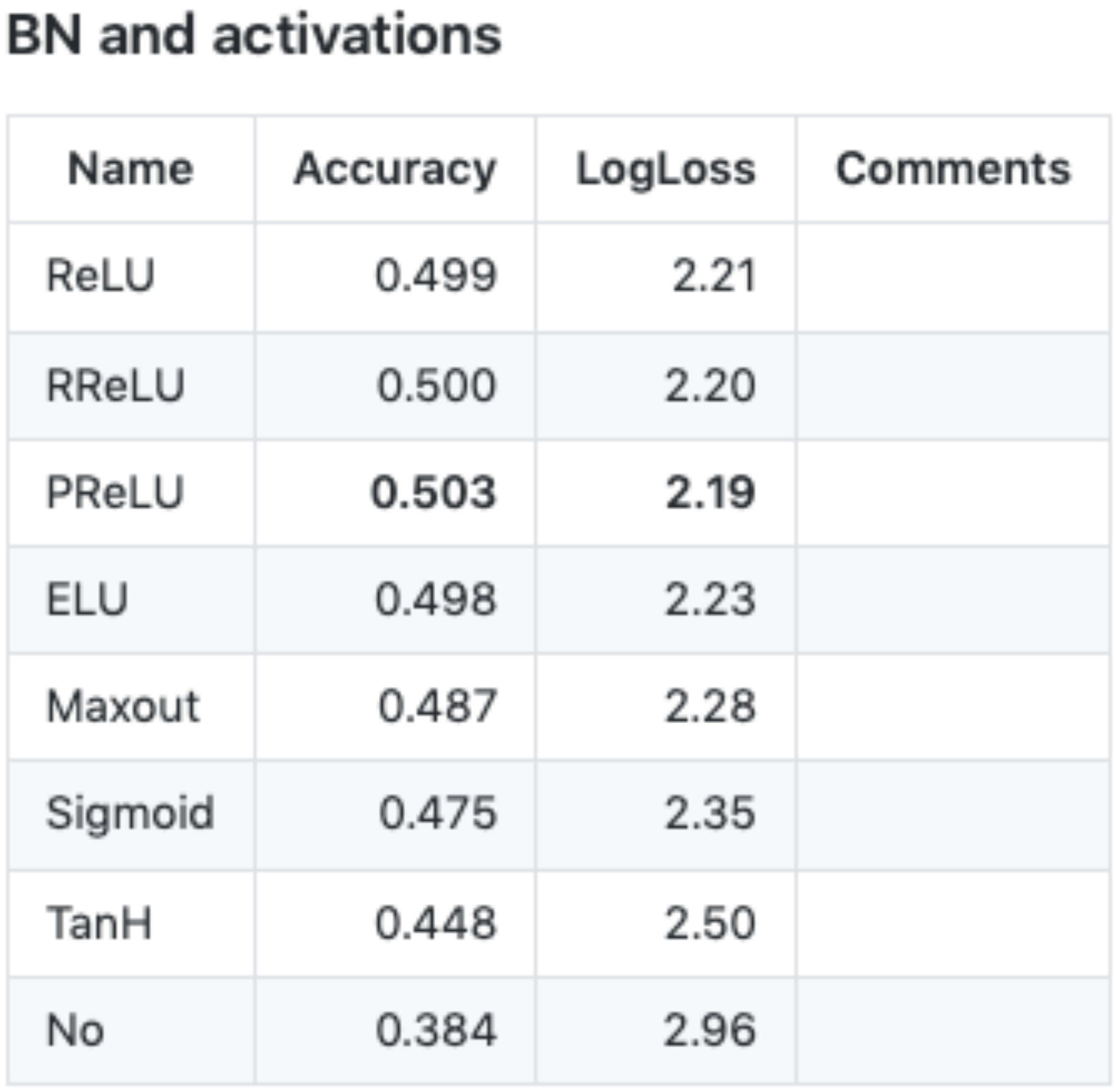


BN -- before or after ReLU?

Name	Accuracy	LogLoss	Comments
Before	0.474	2.35	As in paper
Before + scale&bias layer	0.478	2.33	As in paper
After	**0.499**	**2.21**	
After + scale&bias layer	0.493	2.24	

Some Benchmarks

https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md#bn----before-or-after-relu

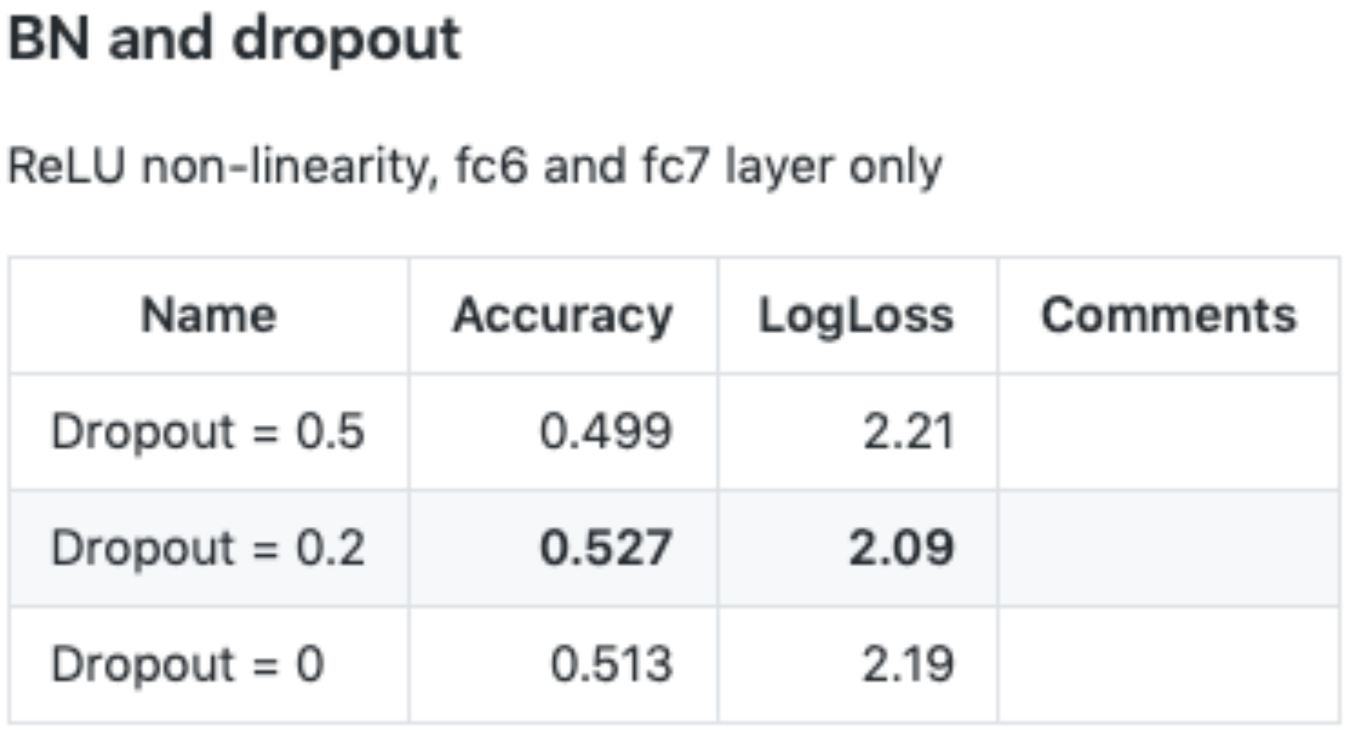


BN and activations

Name	Accuracy	LogLoss	Comments
ReLU	0.499	2.21	
RReLU	0.500	2.20	
PReLU	**0.503**	**2.19**	
ELU	0.498	2.23	
Maxout	0.487	2.28	
Sigmoid	0.475	2.35	
TanH	0.448	2.50	
No	0.384	2.96	

Some Benchmarks

https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md#bn----before-or-after-relu



BN and dropout

ReLU non-linearity, fc6 and fc7 layer only

Name	Accuracy	LogLoss	Comments
Dropout = 0.5	0.499	2.21	
Dropout = 0.2	**0.527**	**2.09**	
Dropout = 0	0.513	2.19	

Other Normalization Methods for Hidden Activations

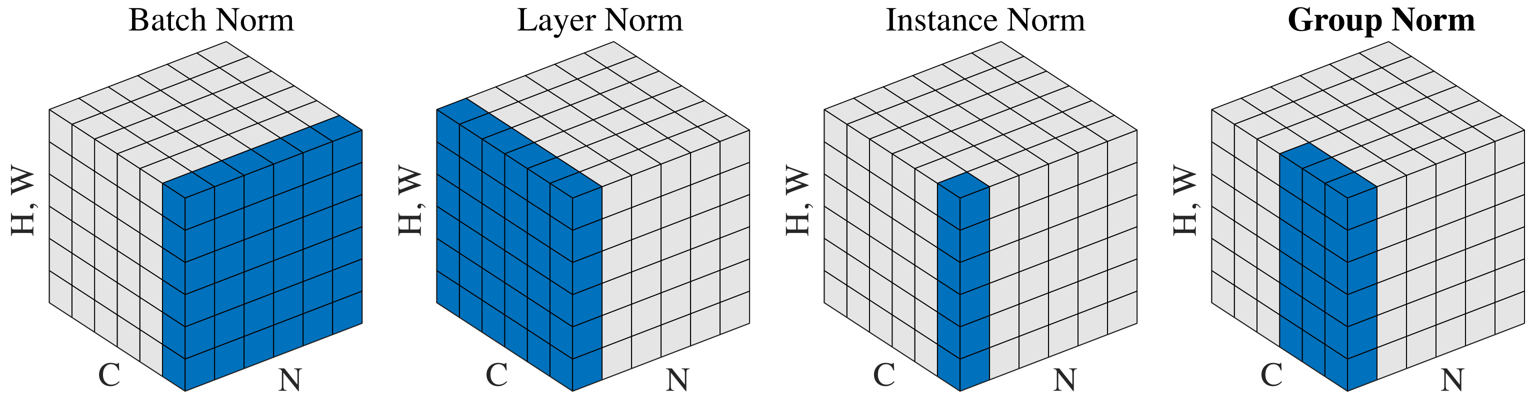


Figure 2. Normalization methods. Each subplot shows a feature map tensor. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels. Group Norm is illustrated using a group number of 2.

Wu, Y., & He, K. (2018). Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 3-19).

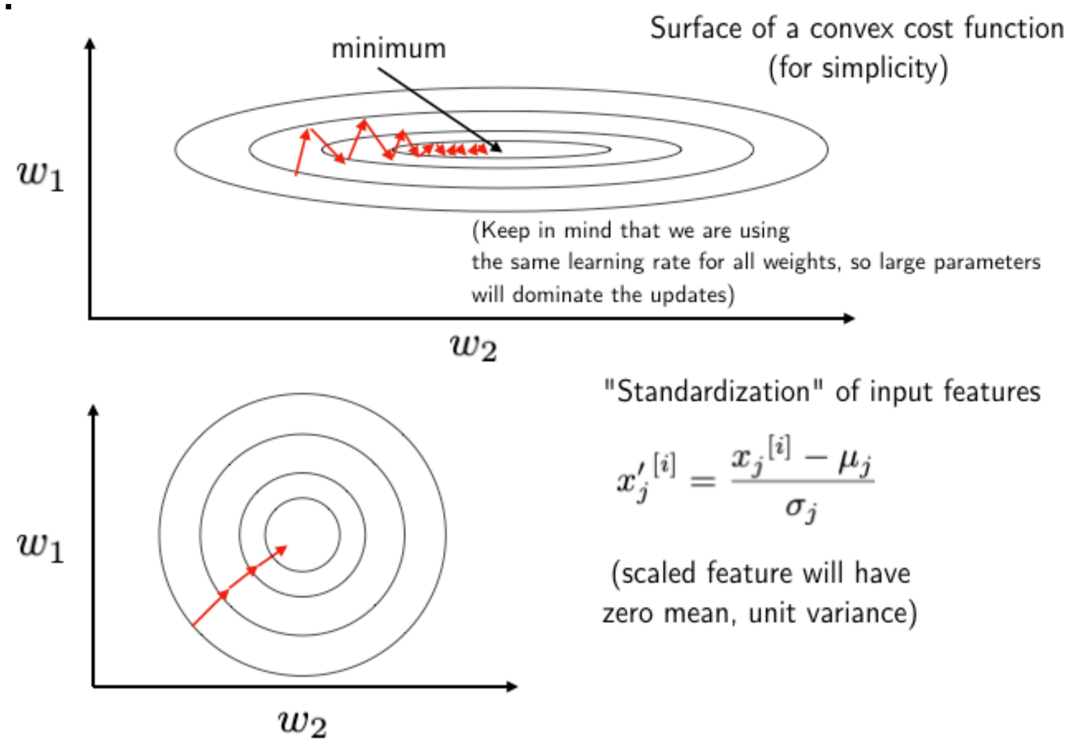
(will revisit after introducing Convolutional Neural Networks)

Part 2: Weight Initialization

Weight Initialization

- We previously discussed that we want to initialize weight to small, random numbers to break symmetry
- Also, we want the weights to be relatively, why?

Tip (from an earlier slide):



Surface of a convex cost function (for simplicity)

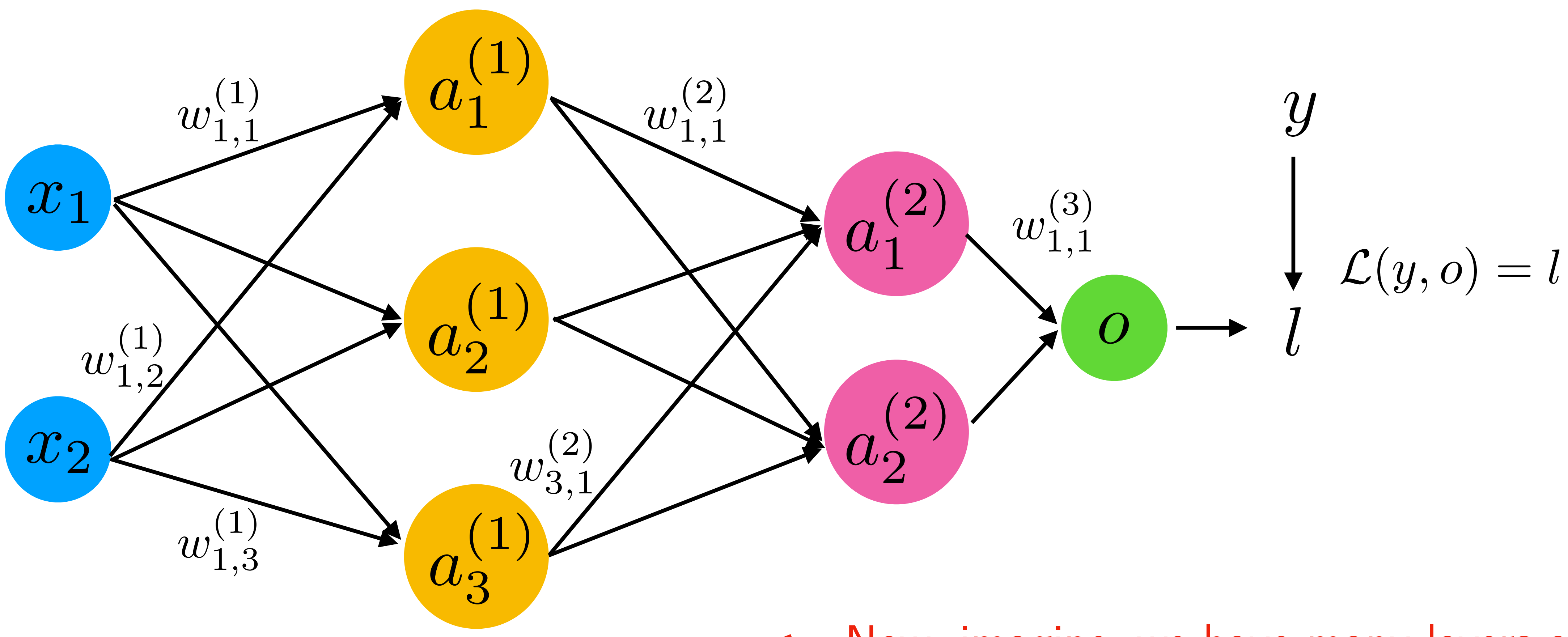
(Keep in mind that we are using the same learning rate for all weights, so large parameters will dominate the updates)

"Standardization" of input features

$$x'^{[i]}_j = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$$

(scaled feature will have zero mean, unit variance)

Sidenote: Vanishing/Exploding Gradient Problems



Now, imagine, we have many layers and sigmoid activations ...

$$\frac{\partial l}{\partial w^{(1)}_{1,1}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a^{(2)}_1} \cdot \boxed{\frac{\partial a^{(2)}_1}{\partial a^{(1)}_1}} \cdot \frac{\partial a^{(1)}_1}{\partial w^{(1)}_{1,1}}$$

$$+ \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a^{(2)}_2} \cdot \frac{\partial a^{(2)}_2}{\partial a^{(1)}_1} \cdot \frac{\partial a^{(1)}_1}{\partial w^{(1)}_{1,1}}$$

Sidenote: Vanishing/Exploding Gradient problems

$$\begin{aligned}\frac{\partial l}{\partial w_{1,1}^{(1)}} &= \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \boxed{\frac{\partial a_1^{(2)}}{\partial a_1^{(1)}}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} \\ &\quad + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}\end{aligned}$$

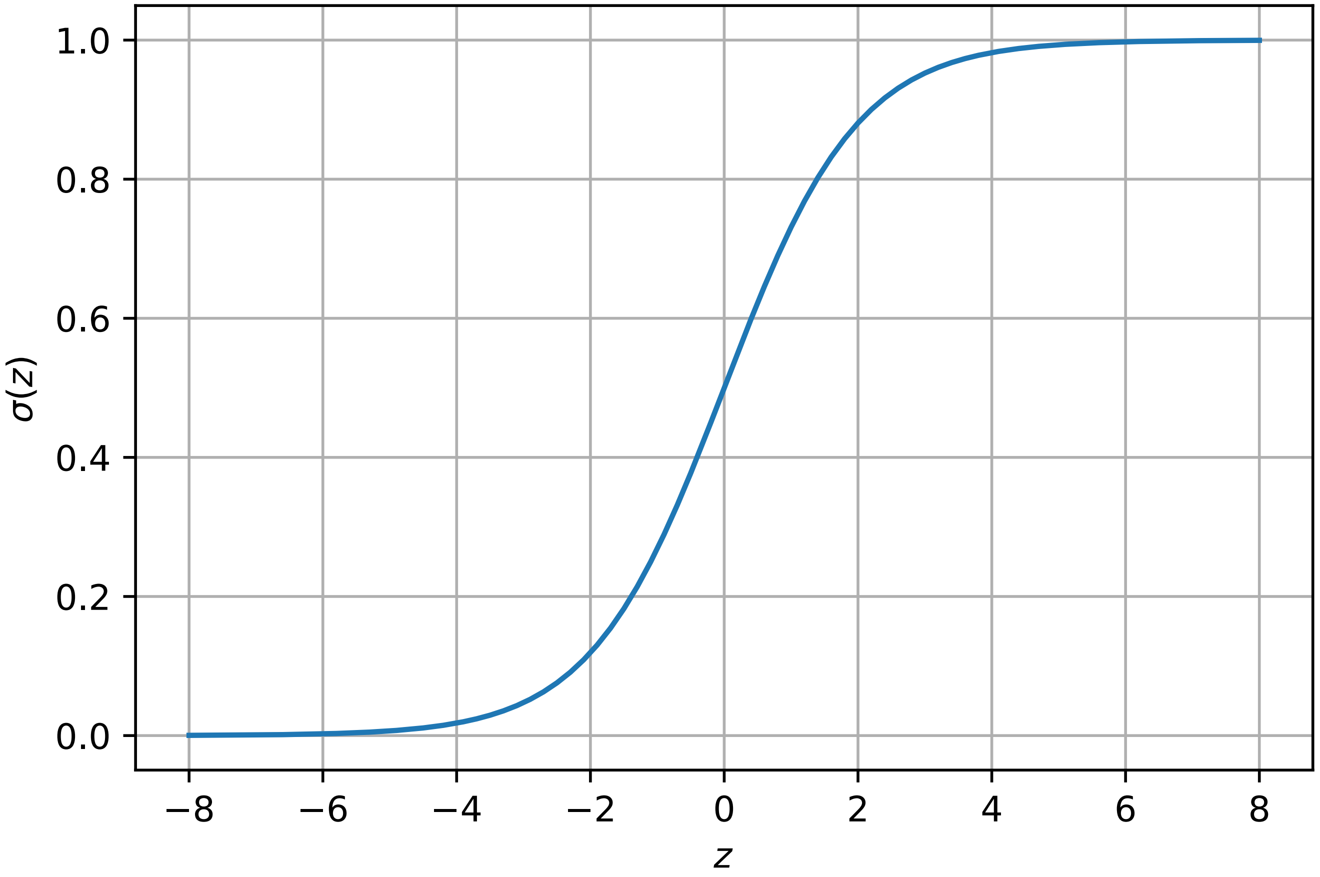
Now, imagine, we have many layers and logistic sigmoid activations ...

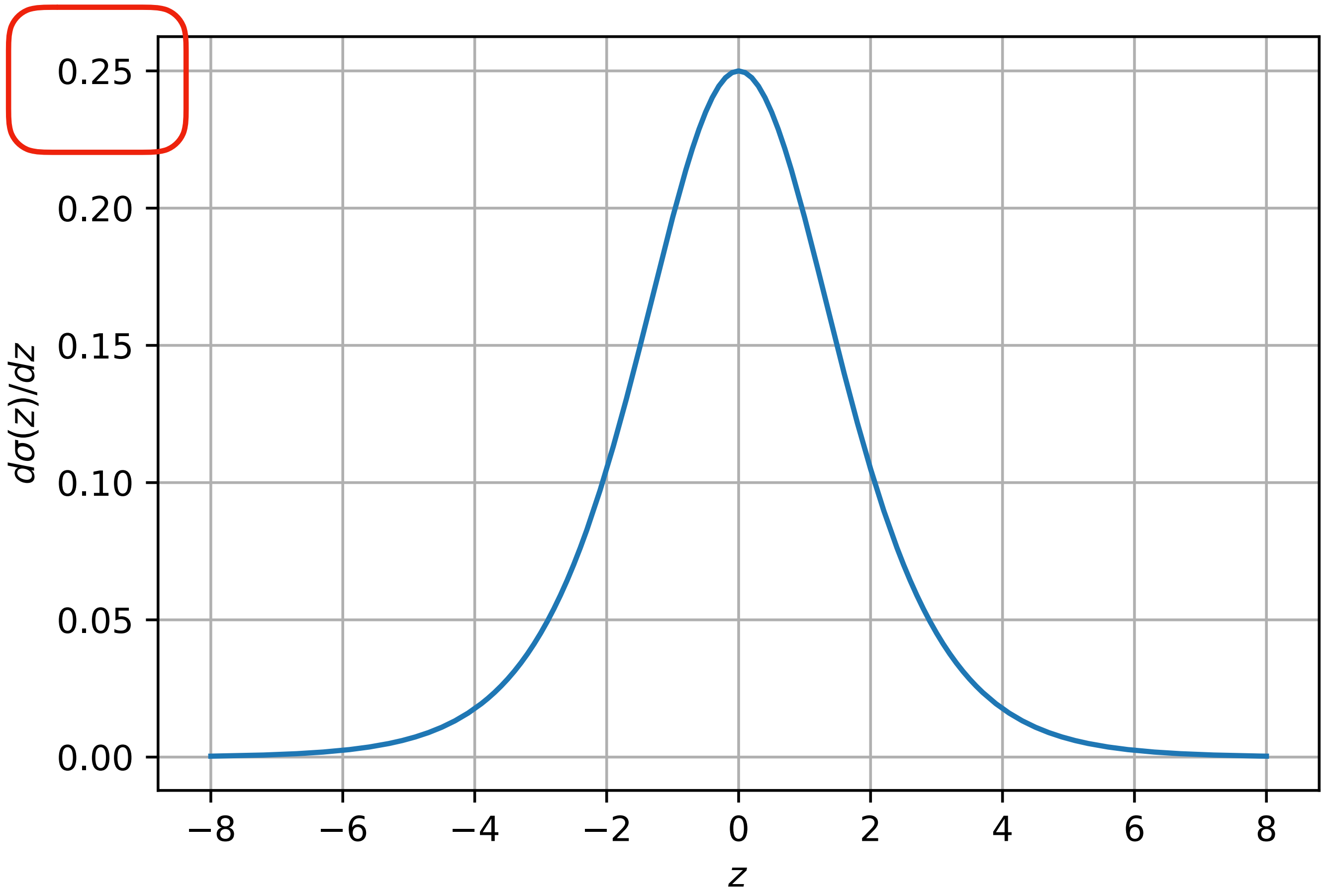
$$\sigma'(z^{[i]}) = \sigma(z^{[i]}) \cdot (1 - \sigma(z^{[i]}))$$

Sidenote: Vanishing/Exploding Gradient Problems

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

$$\frac{d}{dz}\sigma(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma(z)(1-\sigma(z))$$

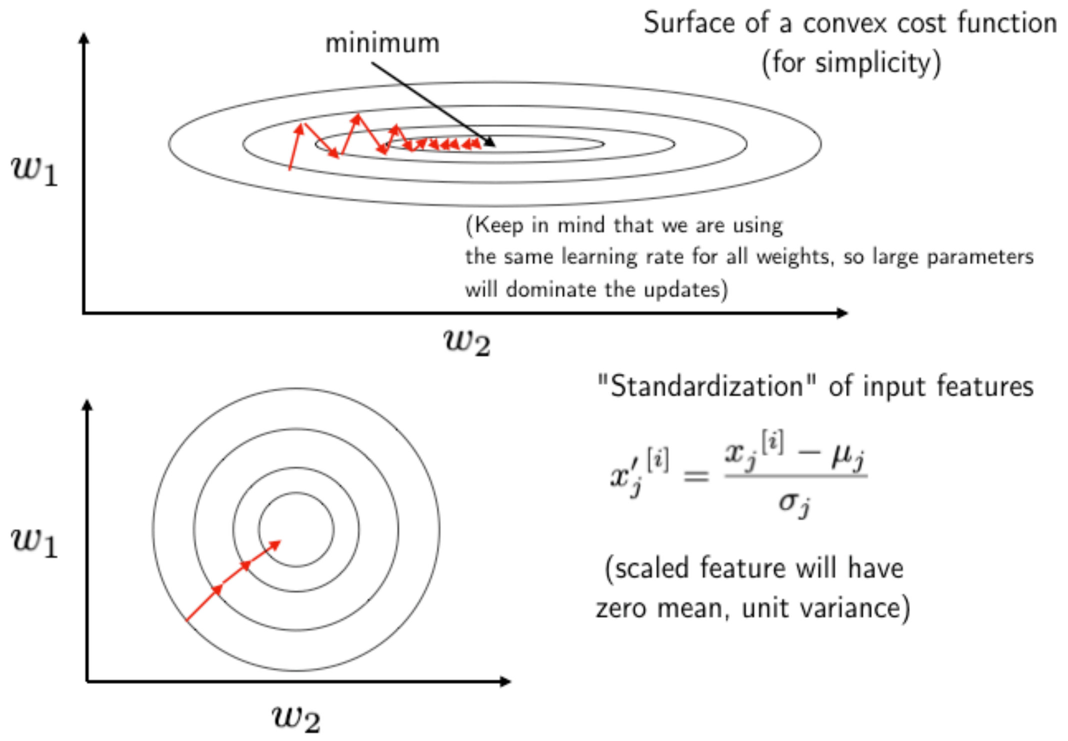




Weight Initialization

- Traditionally, we can initialize weights by sampling from a random uniform distribution in range [0, 1], or better, [-0.5, 0.5]
- Or, we could sample from a Gaussian distribution with mean 0 and small variance (e.g., 0.1 or 0.01)
- When would you choose which?

Tip (from an earlier slide):



minimum

Surface of a convex cost function (for simplicity)

(Keep in mind that we are using the same learning rate for all weights, so large parameters will dominate the updates)

"Standardization" of input features

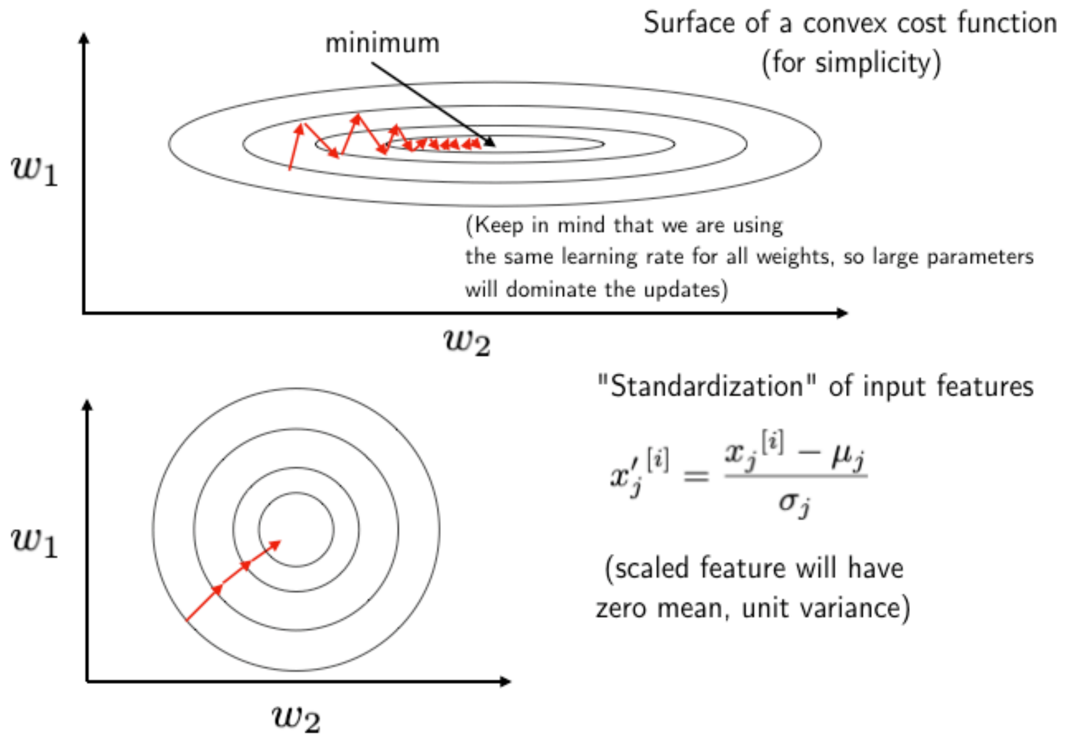
$$x_j'^{[i]} = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$$

(scaled feature will have zero mean, unit variance)

Weight Initialization

- Traditionally, we can initialize weights by sampling from a random uniform distribution in range [0, 1], or better, [-0.5, 0.5]
- Or, we could sample from a Gaussian distribution with mean 0 and small variance (e.g., 0.1 or 0.01)
- When would you choose which?

Tip (from an earlier slide):



minimum

Surface of a convex cost function (for simplicity)

(Keep in mind that we are using the same learning rate for all weights, so large parameters will dominate the updates)

"Standardization" of input features

$$x'^{[i]}_j = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$$

(scaled feature will have zero mean, unit variance)

Sidenote: You can initialize the bias units to all zeros

Custom Weight Initialization in PyTorch

```python
class MLP(torch.nn.Module):

    def __init__(self, num_features, num_hidden, num_classes):
        super(MLP, self).__init__()

        self.num_classes = num_classes

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_hidden)
        self.linear_1.weight.detach().normal_(0.0, 0.1)
        self.linear_1.bias.detach().zero_()

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden, num_classes)
        self.linear_out.weight.detach().normal_(0.0, 0.1)
        self.linear_out.bias.detach().zero_()

    def forward(self, x):
        out = self.linear_1(x)
        out = torch.sigmoid(out)
        logits = self.linear_out(out)
        probas = torch.sigmoid(logits)
        return logits, probas
```

Weight Initialization -- Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

- TanH is a bit more robust regarding vanishing gradients (compared to logistic sigmoid)
- It still has the problem of saturation (near zero gradients if inputs are very large, positive or negative values)
- Xavier initialization is a small improvement for initializing weights for tanH

Weight Initialization -- Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

Method:

Step 1: Initialize weights from Gaussian or uniform distribution with (previous slide)

Step 2: Scale the weights proportional to the number of inputs to the layer

(For the first hidden layer, that is the number of features in the dataset;
for the second hidden layer, that is the number of units in the 1st hidden layer
etc.)

Weight Initialization -- Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

Method:

Scale the weights proportional to the number of inputs to the layer

In particular, scale as follows:

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{1}{m^{(l-1)}}}$$

where m is the number of input units to the next layer

e.g.,

$$W_{i,j}{}^{(l)} \sim N(\mu = 0, \sigma^2 = 0.01)$$

(or uniform distr. in a fixed interval, as in the original paper)

Xavier Initialization in PyTorch

Semi-Automatic:

```
...
self.linear = torch.nn.Linear(...)
torch.nn.init.xavier_uniform_(conv1.weight)
...
```

More conveniently for all weights in e.g., fully-connected layers:

```
...
def weights_init(m):
    if isinstance(m, nn.Linear):
        torch.nn.init.xavier_uniform_(m.weight)
        torch.nn.init.xavier_uniform_(m.bias)

model.apply(weights_init)
...
```

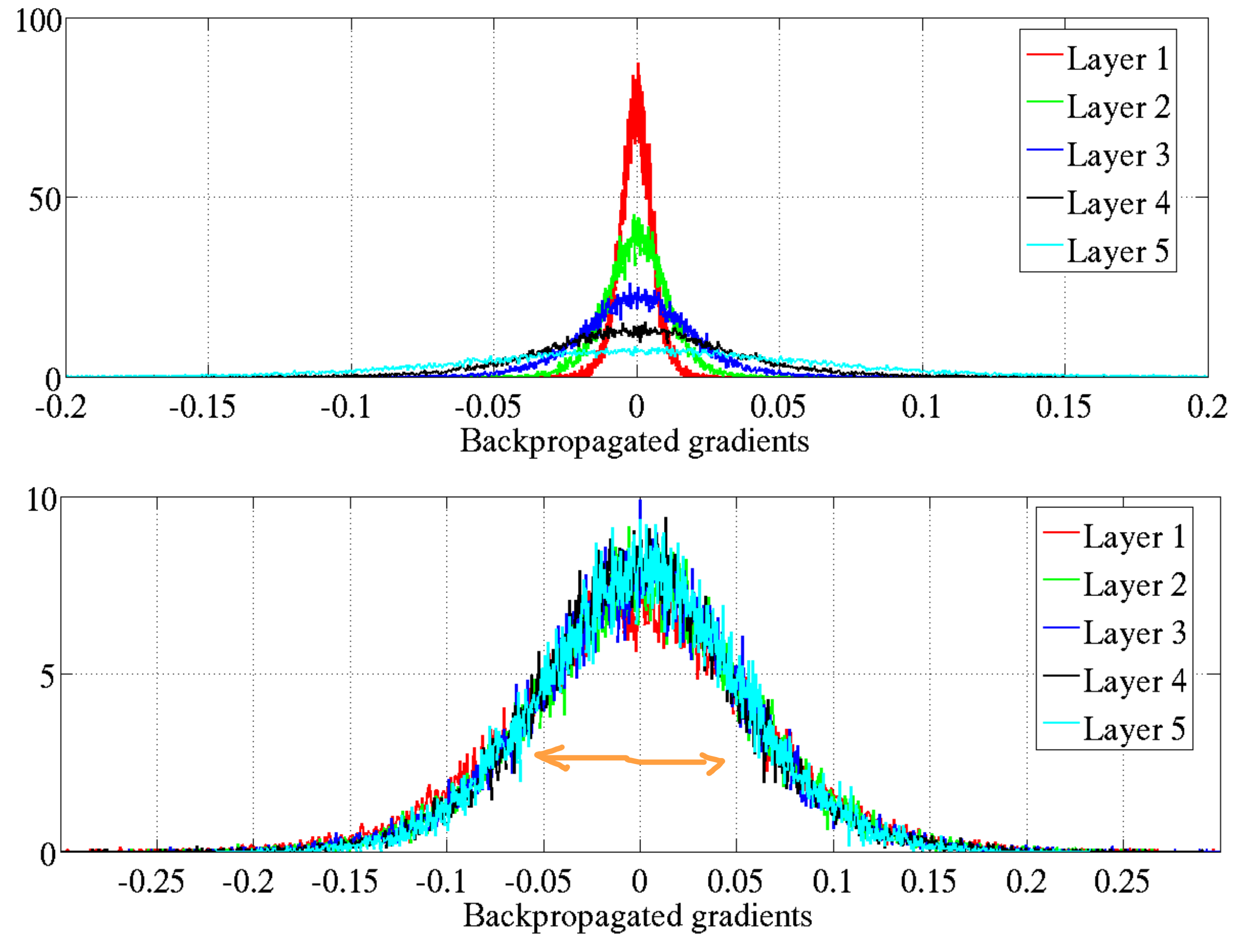
Weight Initialization -- Xavier Initialization

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{1}{m^{[l-1]}}}$$

Again, some DL jargon: This is sometimes called "fan in"
(= number of inputs to a layer)

Weight Initialization -- Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.



Without Xavier weight initialization

Exploding gradient problem

With Xavier weight initialization

Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Weight Initialization -- He Initialization

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In *Proceedings of the IEEE international conference on computer vision*, pp. 1026-1034. 2015.

- Assuming activations with mean 0, which is reasonable, Xavier Initialization assumes a derivative of 1 for the activation function (which is reasonable for tanH)
- For ReLU, this is different, as the activations are not centered at zero anymore
- He initialization takes this into account (to see that worked out in math, see the paper)
- The result is that we add a scaling factor of $2^{0.5}$

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{2}{m^{[l-1]}}}$$

Weight Initialization -- He Initialization

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In *Proceedings of the IEEE international conference on computer vision*, pp. 1026-1034. 2015.

- Assuming activations with mean 0, which is reasonable, Xavier Initialization assumes a derivative of 1 for the activation function (which is reasonable for tanH)
- For ReLU, this is different, as the activations are not centered at zero anymore
- He initialization takes this into account (to see that worked out in math, see the paper)
- The result is that we add a scaling factor of $2^{0.5}$

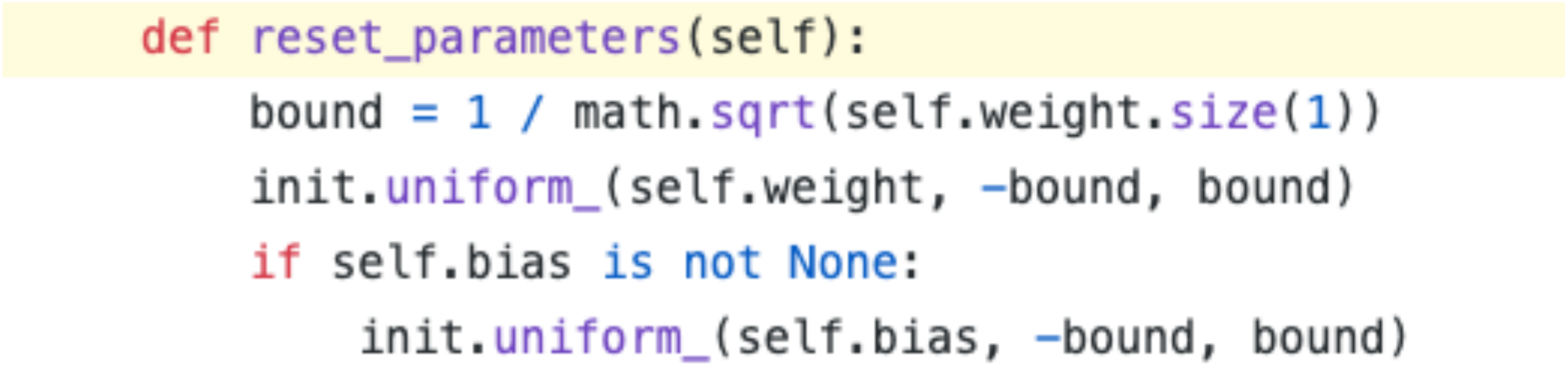
$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{2}{m^{[l-1]}}}$$

For Leaky ReLU with negative slope alpha:

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{2}{(1+\alpha^2) \cdot m^{[l-1]}}}$$

PyTorch Default Weights

PyTorch uses the following scheme by default, which is somewhat similar to Xavier initialization, and works ok in practice most of the time

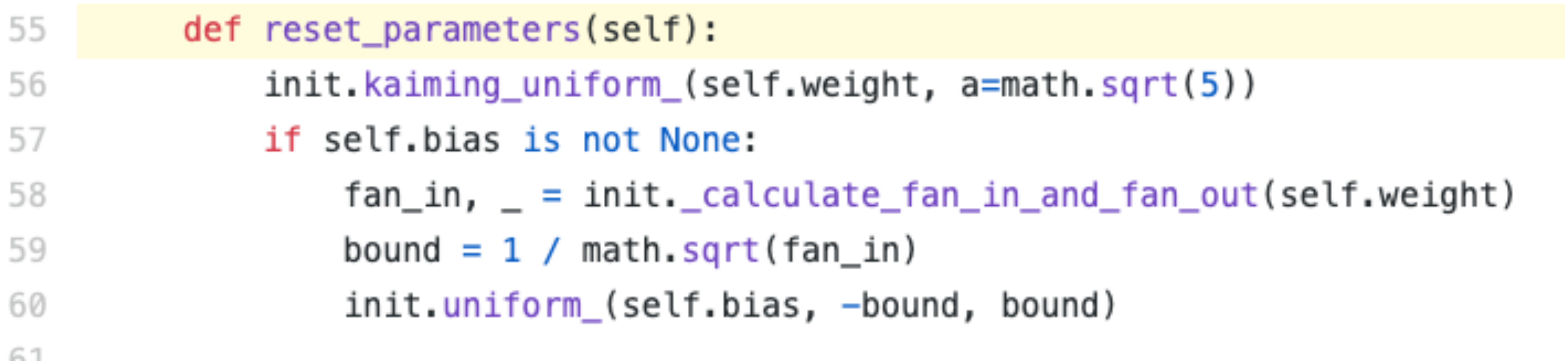


```
    def reset_parameters(self):
        bound = 1 / math.sqrt(self.weight.size(1))
        init.uniform_(self.weight, -bound, bound)
        if self.bias is not None:
            init.uniform_(self.bias, -bound, bound)
```

https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/linear.py#L148

PyTorch Default Weights

However, note that different layers have different defaults



```
    def reset_parameters(self):
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            init.uniform_(self.bias, -bound, bound)
```

https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/conv.py#L55

Note that if BatchNorm is used,
initial feature weight choice is less important anyway

Hands-on Fashion-MNIST

Exercise in pairs:

- Ouput: A report of ~3 pages describing a solution to the **Self sorting wardrobe** problem using Fashion-MNIST dataset to be **delivered in 3 weeks.**
  - Describe the problem, analysis and results obtained following the Data Science process cycle (see file hands-on Excercise.pptx slides 4-5)
- Read/run the suggested tutorial using Google Colab
  - https://www.tensorflow.org/tutorials/keras/classification?hl=pt-br
  - Try other neural network architectures
- Read/run the tutorial Self sorting wardrobe using Peltarion platform
  - https://peltarion.com/knowledge-center/documentation/tutorials/self-sorting-wardrobe
  - Optional