DEEP LEARNING FOR SOLVING AND ESTIMATING DYNAMIC MODELS

University of Leipzig, Switzerland
May 22nd – 24th, 2024

https://github.com/sischei/Deep_Learning_For_Dynamic_Econ

Simon Scheidegger
simon.scheidegger@unil.ch

A DENSE PROGRAM

Day 1, Wednesday, May 22nd, 2024

Time	Main Topics
09:30 - 11:00	Introduction to Machine Learning and Deep Learning for Dynamic Stochastic Economic Models (2 x 45 min)
11:00 - 11:30	Coffee Break
11:30 - 13:00	A hands-on session on Deep Learning and Tensorflow (2 x 45 min)
13:00 - 14:30	Lunch Break
14:30 - 16:00	Deep Equilibrium Nets (2 x 45 min)

Day 2, Thursday, May 23nd, 2024

Time	Main Topics
09:30 - 11:00	Hands-on: Deep Equilibrium Nets (I) – solving a simple dynamic model (2 x 45 min)
11:00 - 11:30	Coffee Break
11:30 - 13:00	Hands-on: Deep Equilibrium Nets (II) – solving a stochastic dynamic model with nonlinearities (2 x 45 min)
13:00 - 14:30	Lunch Break
14:30 - 16:00	Uncertainty Quantification and Estimation for nonlinear models: Gaussian Process Regression and Surrogate models

A DENSE PROGRAM

Day 3, Friday, May 24th, 2024

Time	Main Topics
09:30 - 11:00	Introduction to (stochastic) Integrated Assessment Models (2 x 45 min)
11:00 - 11:30	Coffee Break
11:30 - 13:00	Solving the (non-stationary) DICE model with Deep Equilibrium Nets (2 x 45 min)
13:00 - 14:30	Lunch Break
14:30 - 16:00	Deep Uncertainty Quantification for stochastic Integrated Assessment Models (2 x 45 min)

MOST RECENT INFO AND MATERIALS

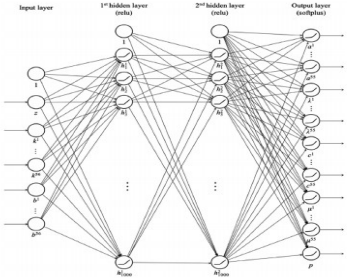
- GITHUB: https://github.com/sischei/Deep_Learning_For_Dynamic_Econ
- Enroll on Nuvolos to get access to the cloud/codes (press the link below)

  → https://app.nuvolos.cloud/enroll/class/OW-jhN1vUjU

INTRO TO DEEP LEARNING

May 22nd, 2023

Simon Scheidegger
simon.scheidegger@unil.ch



FEW WORDS ABOUT MYSELF

- Prof. of Advanced Data Analytics
  at the Department of Economics, University of Lausanne.
- Ph.D. 2010 in theoretical physics (Core-collapse supernova simulations).
- Research interest in computational economics and finance, HPC, and ML applied to macroeconomics, climate economics, finance.
- Website: https://sites.google.com/site/simonscheidegger
- Github: https://github.com/sischei
- Twitter: https://twitter.com/comp_simon

ROAD-MAP OF THIS LECTURE

- Machine Learning Basics
- Intro to Deep Learning
  - The multi-layer perceptron
    - Feed-forward networks
    - Network training – SGD
    - Back-propagation
    - Some notes on over-fitting
    - Deep double descent
- Hands-on:
  - Gradient descent
  - A first glance at Keras/Tensorflow

THE RISE OF NEURAL NETWORKS

TOM SIMONITE BUSINESS 01.25.15 01:05 PM

DEEPMIND BEATS PROS AT STARCRAFT IN ANOTHER TRIUMPH FOR BOTS

A pro gamer and an AI bot duke it out in the strategy game StarCraft, which has becor bed for research on artificial intelligence. STARCRAFT

IN LONDON LAST month, a team from Alphabet's UK-ba artificial intelligence research unit DeepMind quietly l new marker in the contest between humans and compu On Thursday it revealed the achievement in a three-ho YouTube stream, in which aliens and robots fought to t death.

TOM SIMONITE BUSINESS 10.18.17 01:00 PM

THIS MORE POWERFUL VERSION OF ALPHAGO LEARNS ON ITS OWN

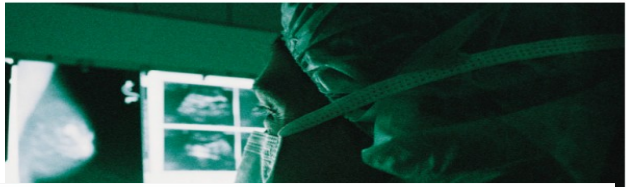
NOAH SHELDON FOR WIRED

AT ONE POINT during his historic defeat to AlphaGo last year, world champion Go play abruptly left the room. The bot had played confounded established theories of the bo moment that came to epitomize the myste AlphaGo.

NEWS BIOLOGY 21 DECEMBER 2017

AI beats docs in cancer spotting

A new study provides a fresh example of machine learning as an important diagnostic tool. Paul Biegler reports.



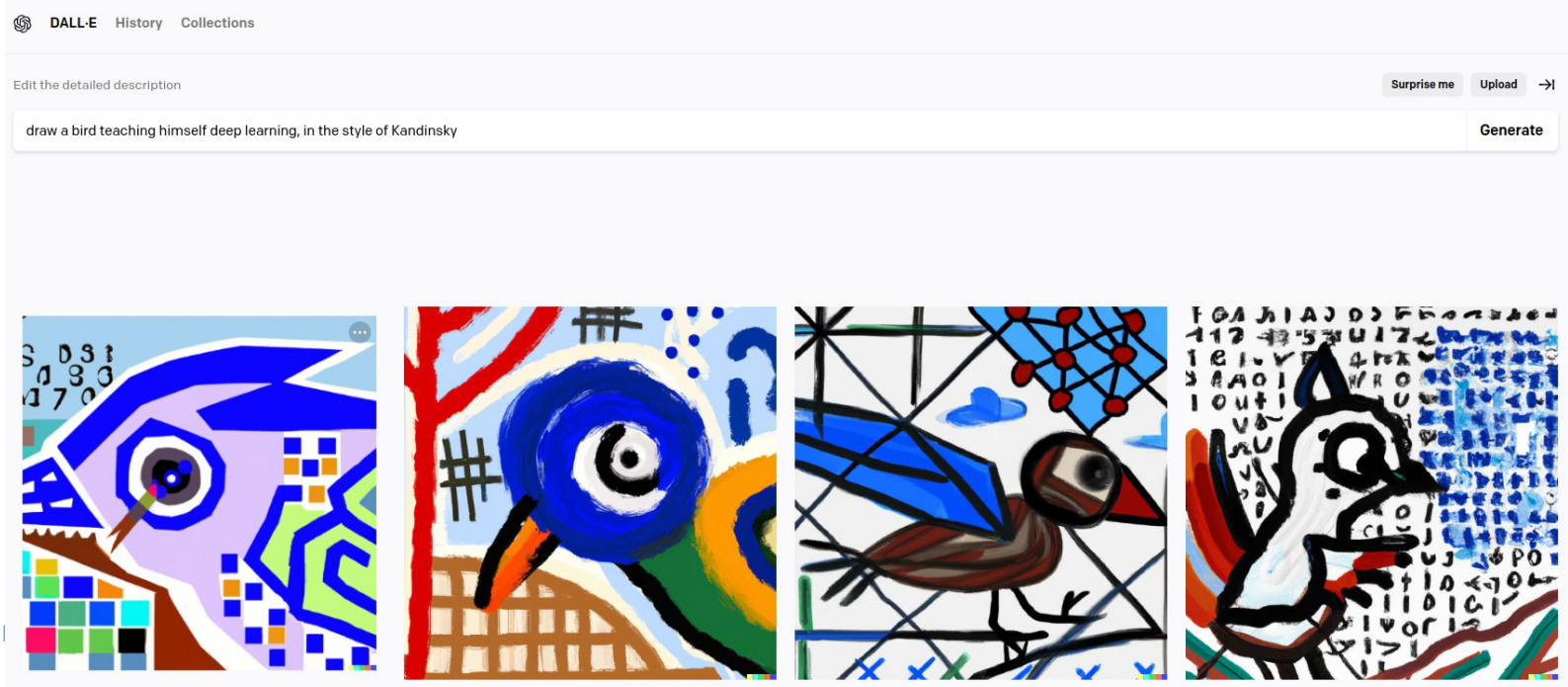
Deep Learning Software Speeds Up Drug Discovery

Wed, 01/16/2019 - 8:00am 1 Comment by Kenny Walter , Science Reporter - @RandDMagazine

The long, arduous process of narrowing down millions of chemical compounds to just a select few that can be further developed into mature drugs, may soon be shortened, thanks to new artificial intelligence (AI) software.

GENERATIVE AI

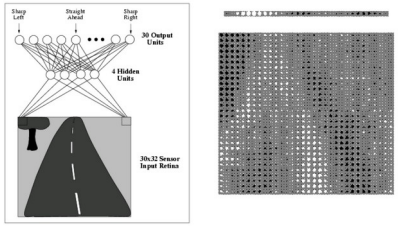
- https://chat.openai.com/
- https://openai.com/dall-e-2



SELF-DRIVING CARS

- Carnegie Mellon University – 1990ies
  - ALVINN: Autonomous Land Vehicle In a Neural Network

ALVINN
[Pomerleau 1993]



- Today (e.g., Waymo)
  - https://www.youtube.com/watch?v=LSX3qdy0dFg

APP: COLORING OLD MOVIES

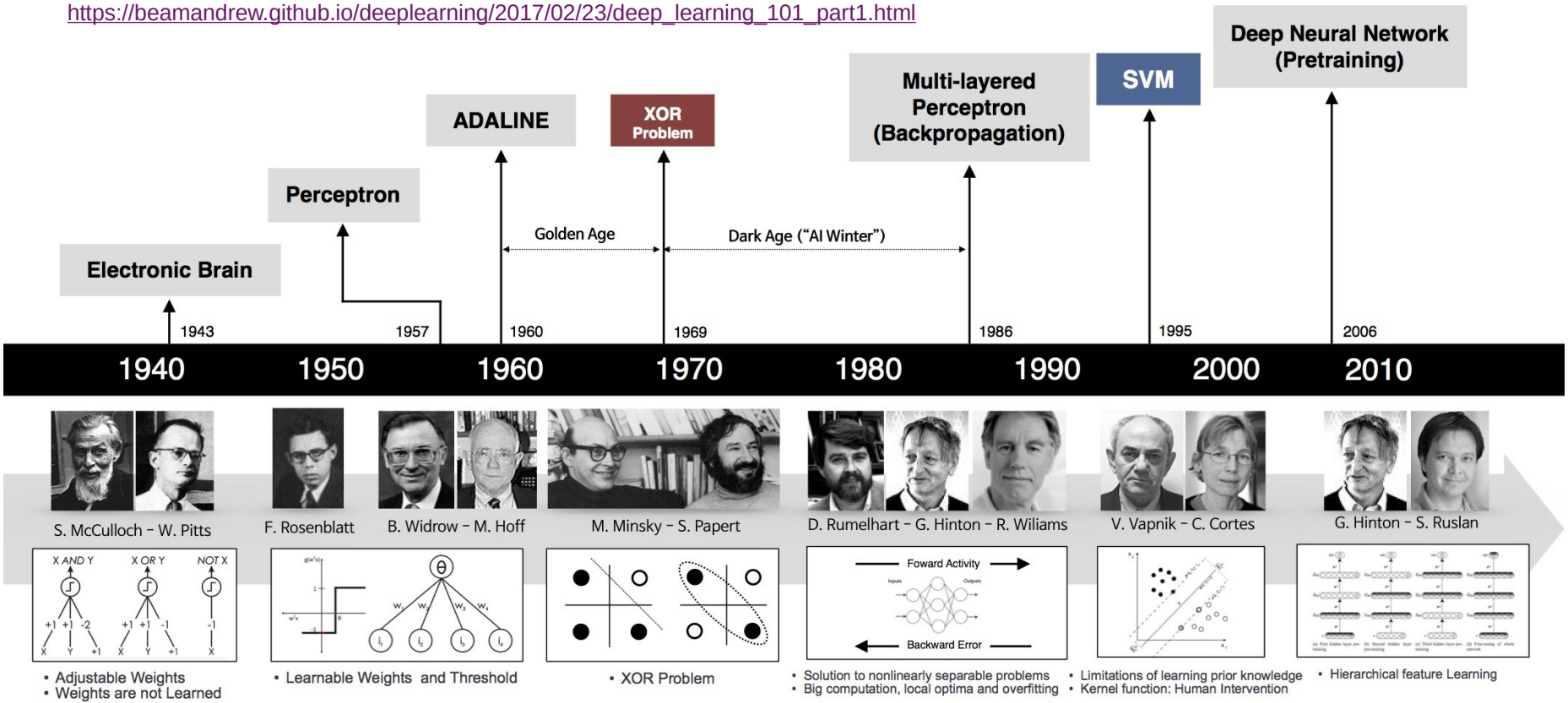
- https://deepsense.ai/ai-movie-restoration-scarlett-ohara-hd/

TWO-LEGGED ROBOTS

- https://www.youtube.com/watch?v=tF4DML7FIWk
- https://www.youtube.com/watch?v=-e1_QhJ1EhQ

A TIMELINE OF DEEP LEARNING

https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html



*J. Schmidhuber clearly missing on this slide

WHY NOW?

- Neural Networks date back decades, so why the resurgence?

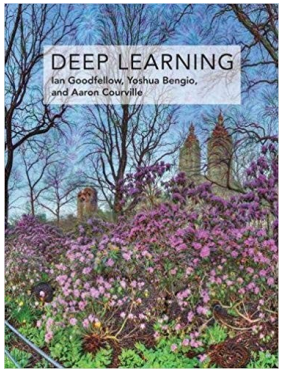
  (Stochastic Gradient Descent: 1952, Perceptron: 1958, Back-propagation: 1986, Deep Convolutional NN: 1995)

  - **Big Data**
    - Large Datasets
    - Easier Collection and Storage
  - **Hardware**
    - GPUs, TPUs,…
  - **Software**
    - Improved Techniques
    - Toolboxes

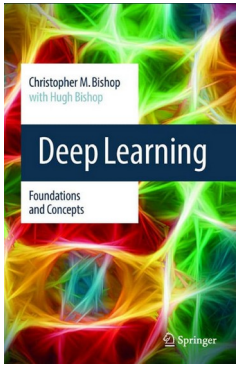
TensorFlow PyTorch

SOME USEFUL MATERIALS

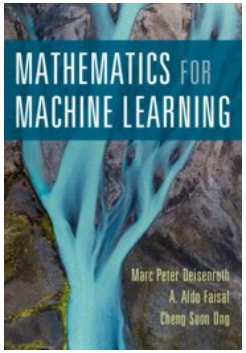
Deep Learning
Ian Goodfellow and Yoshua Bengio and Aaron Courville
MIT Press 2016
http://www.deeplearningbook.org/



Deep Learning – Foundations and Concepts
C. M Bishop, H. Bishop, Springer 2024
(pdf freely available)



Mathematics for Machine Learning
Deisenroth, A. Aldo Faisal, and Cheng Soon Ong.
Cambridge University Press 2020

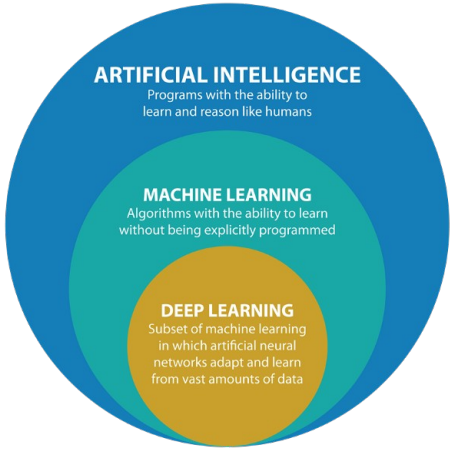


→ ***There is a great community out there (use your browser and Google around...)***

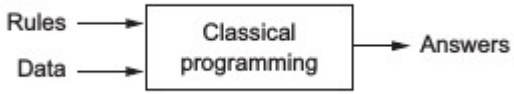
RECAP ON MACHINE LEARNING

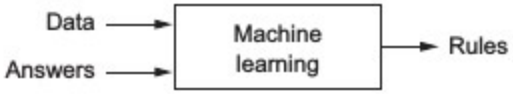
SOME TERMINOLOGY

- **Artificial intelligence (AI)**
  - Can computers be made to “think”—a question whose ramifications we’re still exploring today.
  - A concise definition of the field would be as follows: the effort to automate intellectual tasks normally performed by humans.

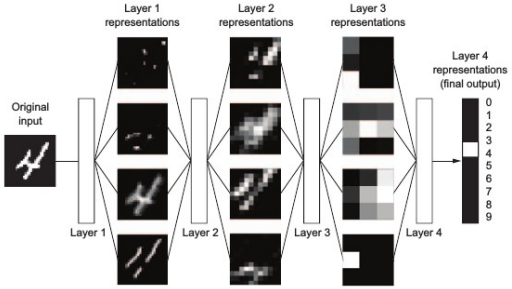


- **Machine learning (e.g., supervised ML)**





- **Deep Learning as a particular example of an ML technique**



Figs. from F. Chollet (2018) – ML with Python

TYPES OF MACHINE LEARNING

- **Supervised Learning**
  - assume that training data is available from which they can learn to predict a target feature based on other features (e.g., monthly rent based on area).

    - **Classification**
    - **Regression**

- **Unsupervised Learning**
  - take a given data-set and aim at gaining insights by identifying patterns, e.g., by grouping similar data points.

- **Reinforcement Learning**

SUPERVISED REGRESSION

- Regression aims at predicting a numerical target feature based on one or multiple other (numerical) features.

- Example: Price of a used car.
  - x : car attributes
  - y : price
  - $y = h(x \mid \theta)$
  - h ( ): model
  - θ : parameters

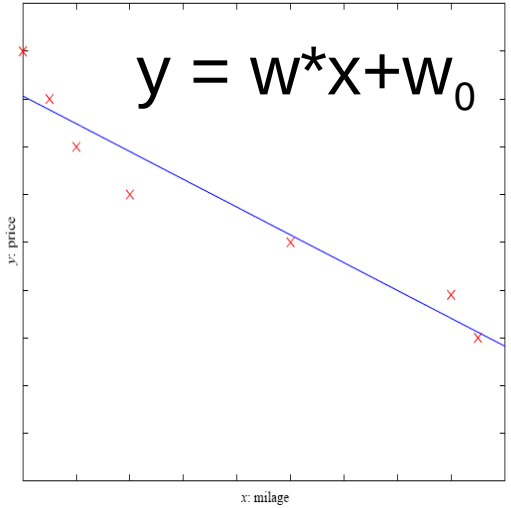


Fig. from Alpaydin (2014)

SUPERVISED CLASSIFICATION

Example 1: Spam Classification

- Decide which emails are Spam and which are not.
- Goal: Use emails seen so far to produce a good prediction
- rule for **future** data.

Example 2: Credit Scoring

- Differentiating between low-risk and high-risk customers from their income and savings.

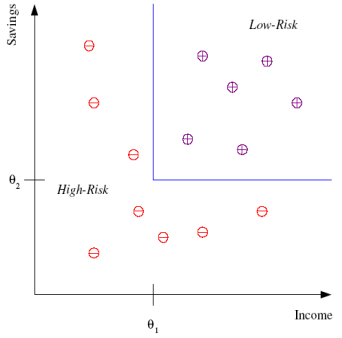
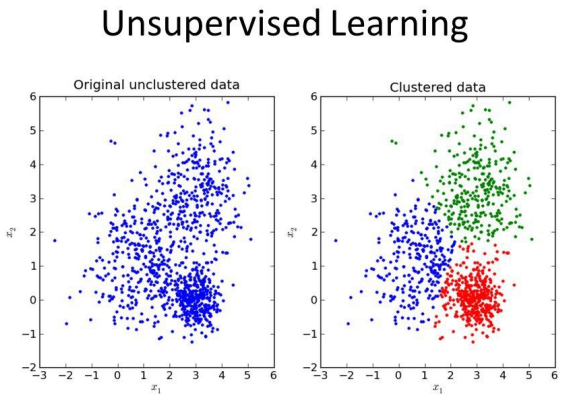


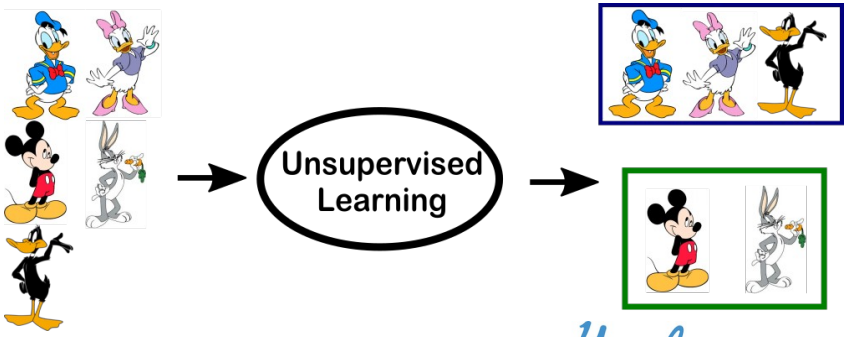
Fig. from Alpaydin (2014)

- Discriminant: IF income > θ_2 AND savings > θ_2
  THEN low-risk ELSE high-risk

UNSUPERVISED ML

- No output
- Clustering: Grouping similar instances
- Example applications:
  - Customer segmentation
  - Image compression
  - Bio-informatics: Learning motifs
  - ...





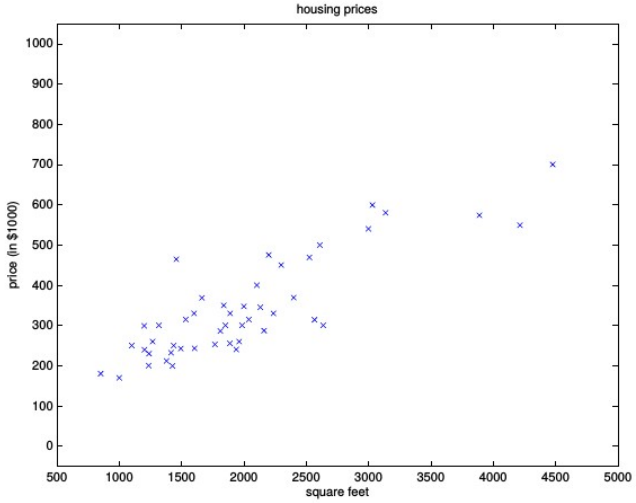
REINFORCEMENT LEARNING

- Learning a policy: A sequence of outputs

- No supervised output but delayed reward
  - Game playing
  - Robot in a maze
  - …
- See, e.g., https://www.youtube.com/watch?v=V1eYniJ0Rnk&vl=en

BUILDING AN ML ALGORITHM (I)

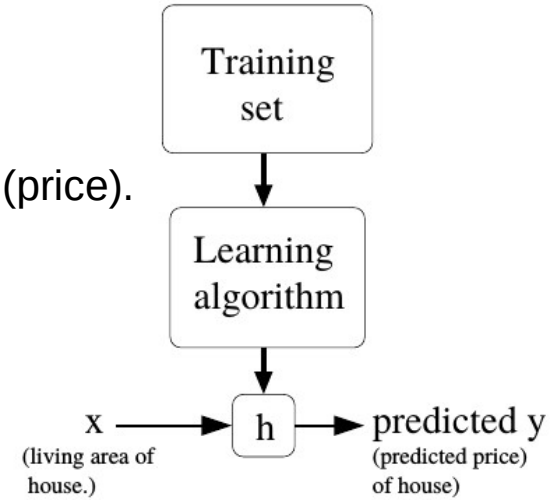
Living area (feet^2)	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮



- Given data like this, how can we learn to predict the prices of other houses as a function of the size of their living areas?

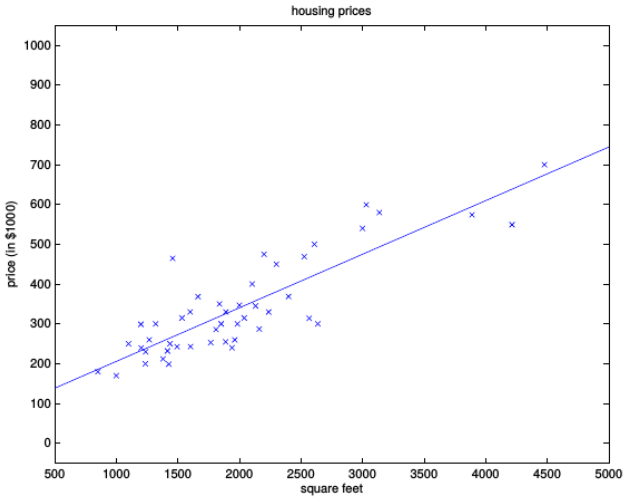
BUILDING AN ML ALGORITHM (II)

- **x(i): "input" variables** (living area in this example), also called **input features**
- **y(i): "output" / target variable** that we are trying to predict (price).
- **Training example**: a pair (x(i) , y(i)).
- **Training set**: a list of m training examples {(x(i), y (i)); i = 1, . . . , m}
  - To perform supervised learning, we must decide how we're going to represent **functions/hypotheses *h*** in a computer.



BUILDING AN ML ALGORITHM (III)

- Model / Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x_1$
  - θ_i's: parameters



- Cost Function:

$$\mathrm{J}(\theta) = \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2$$

  - Minimize J(θ) in order to obtain the coefficients θ.

BUILDING AN ML ALGORITHM (IV)

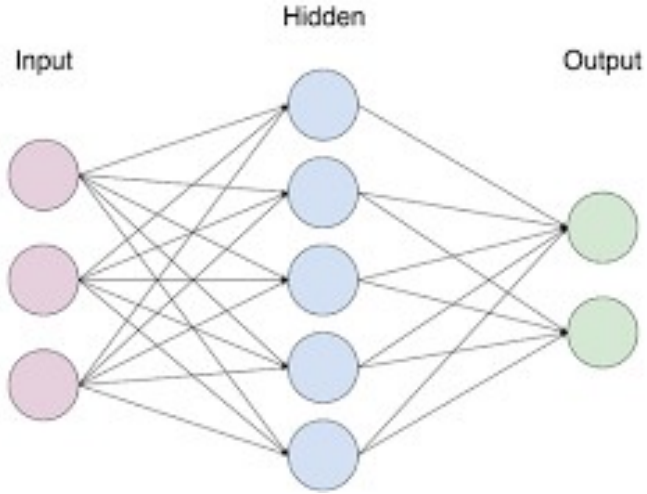
- In General: Machine learning in 3 steps:
  - Choose a **model** $h(x|\theta)$.
  - Define a **cost function** $J(\theta|x)$.
  - **Optimization procedure** to find θ^* that minimizes $J(\theta)$.

- Computationally, we need:
  - data, linear algebra, statistics tools, and optimization routines.

DON’T RE-INVENT THE WHEEL

- Plenty of Frameworks out there
  - Keras
  - Tensorflow
  - Pytorch
  - Caffee
  - Scikit-learn
  - ...

ARTIFICIAL NEURAL NETWORKS

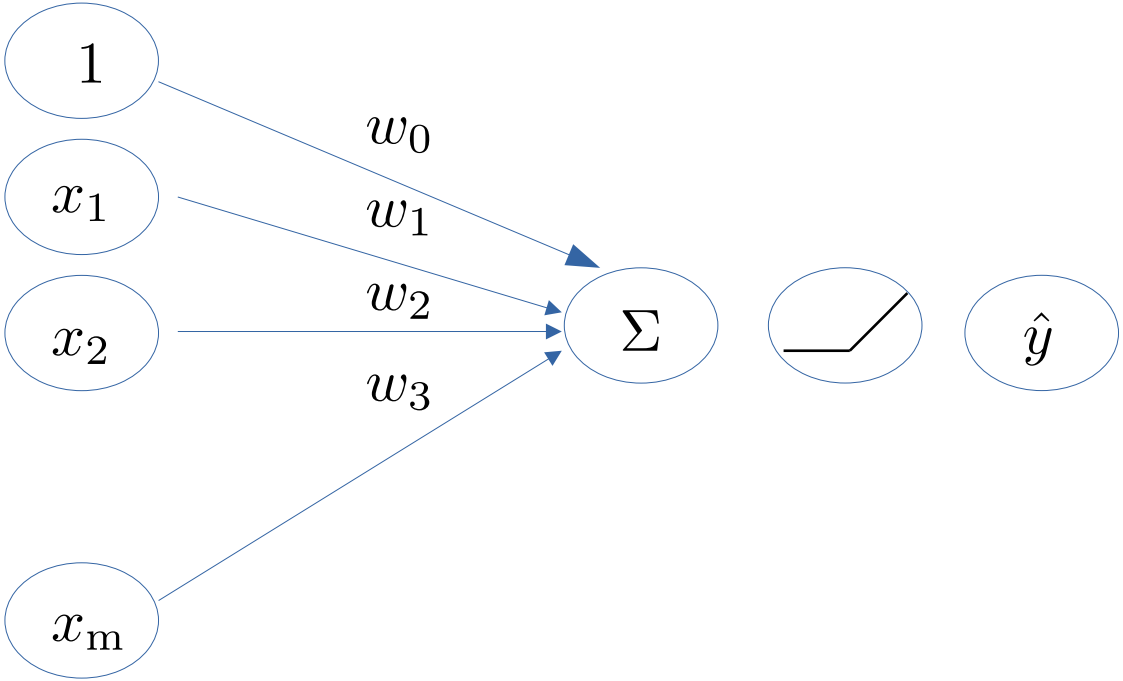


ARTIFICIAL NEURAL NETWORKS

- Artificial Neural networks arise from attempts to model human/animal brains
  - Many models, many claims of biological plausibility.

- We will focus on multi-layer perceptron
  - Mathematical properties rather than plausibility.

A SINGLE NEURON: THE PERCEPTRON



Output Linear combination of inputs

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

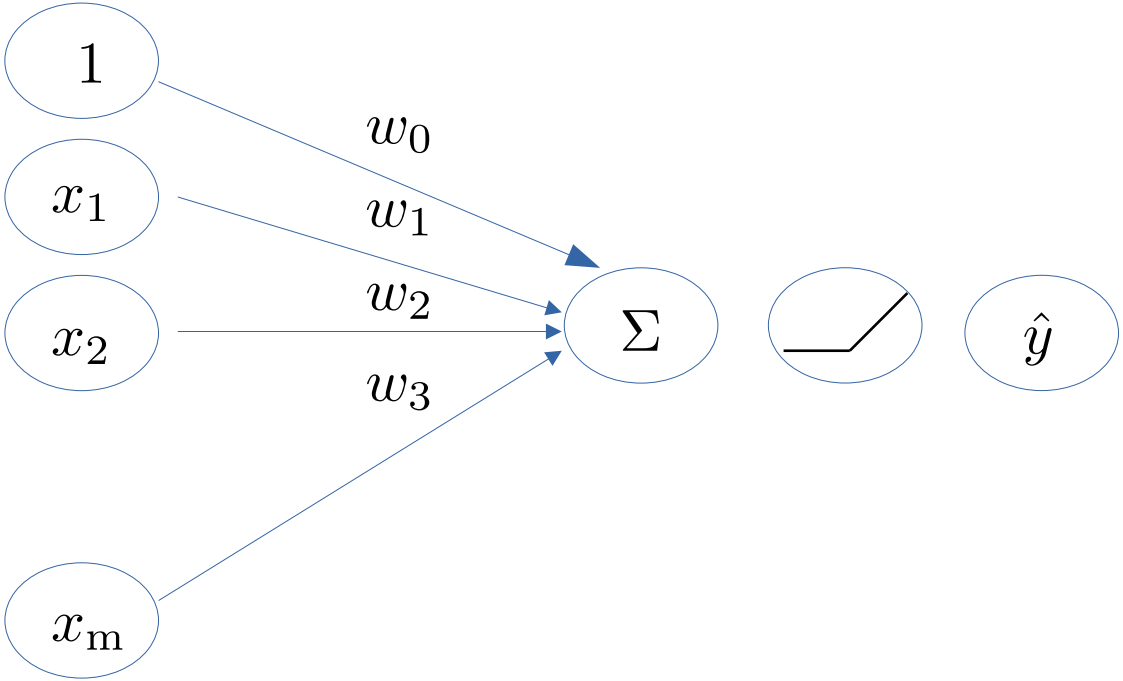
Bias

Non-linear activation function

Inputs Weights Sum Non-Linearity Output

→ **Bias term allows you to shift your activation function to the left or the right**

THE PERCEPTRON: FORWARD PROPAGATION



$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

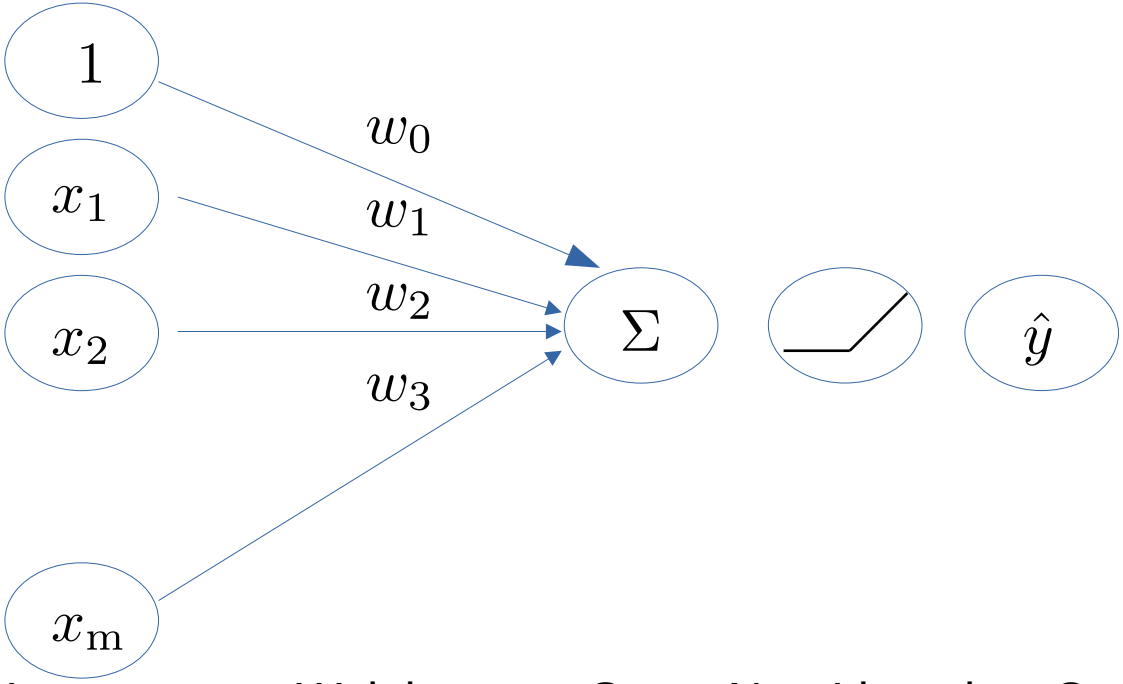
$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

$$\begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } \boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

Inputs Weights Sum Non-Linearity Output

→ **Bias term allows you to shift your activation function to the left or the right**

THE PERCEPTRON: FORWARD PROPAGATION

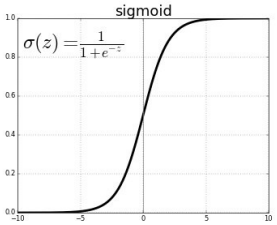


Inputs Weights Sum Non-Linearity Output

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

Activation Functions
e.g. sigmoid function

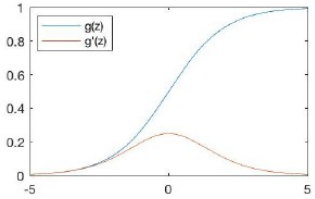
$$g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$$



→ **Bias term allows you to shift your activation function to the left or the right**

FEW ACTIVATION FUNCTIONS

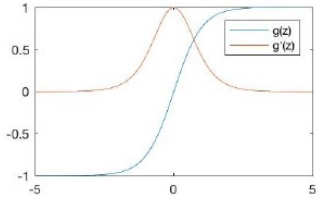
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

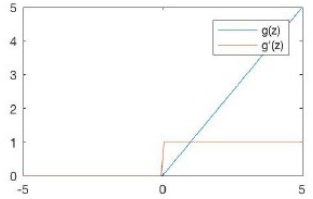
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

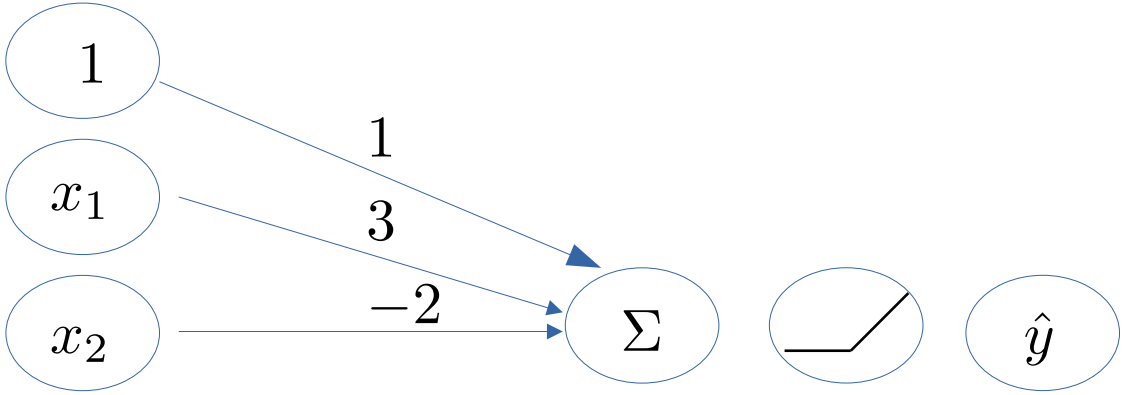


$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

- Needs to be differentiable for gradient-based learning (later)
  - Very useful in practice.
  - Sigmoid function, e.g., useful for classification (Probability).

PERCEPTRON – AN EXAMPLE



We have: $w_0 = 1$ and $W = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

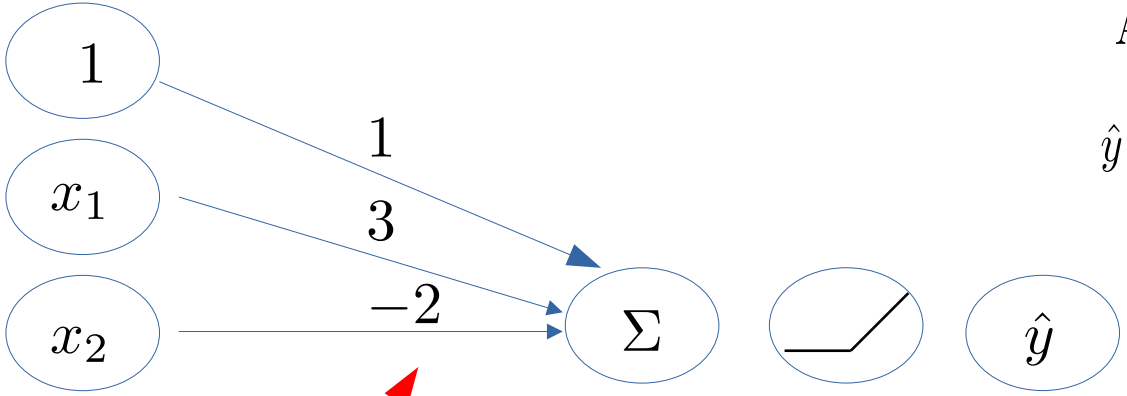
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

$$\hat{y} = g\left(1 + 3x_1 - 2x_2\right)$$

This is just a line in $2D$

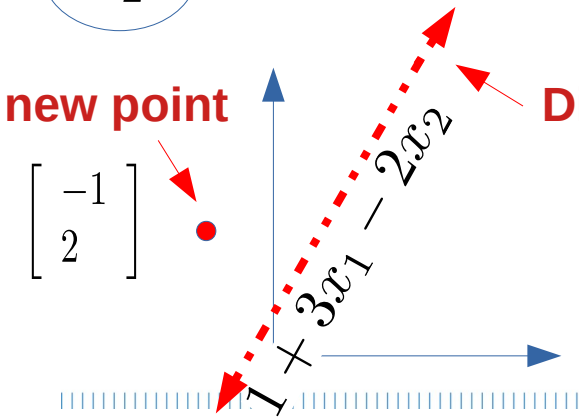
Imagine we have a trained network with weights given.
→ how do we compute the output?

PERCEPTRON – AN EXAMPLE



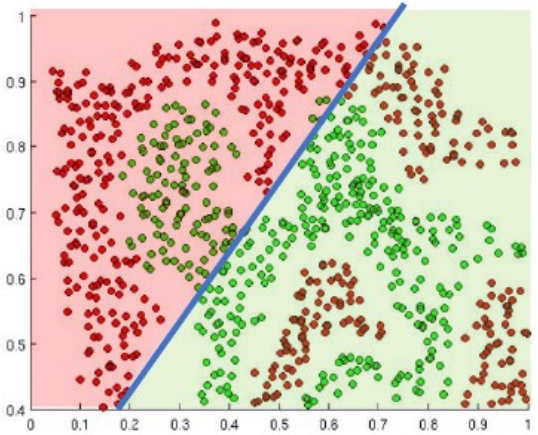
Assume we have input: $X = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\hat{y} = g(1 + (3 * -1) - (2 * 2))$$
$$= g(-6) \approx 0.002$$



IMPORTANCE OF ACTIVATION FCT.

- The purpose of activation functions is to introduce non-linearities into the network.

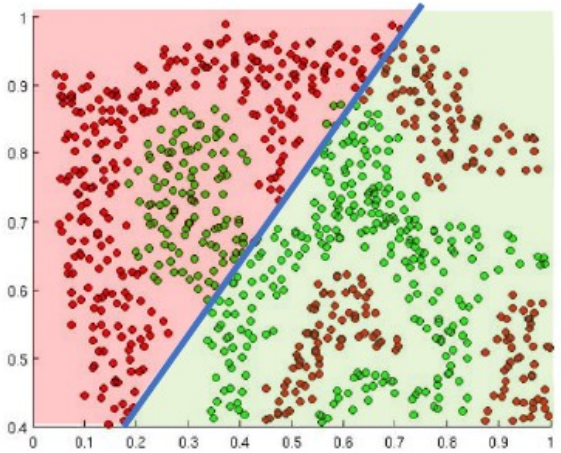


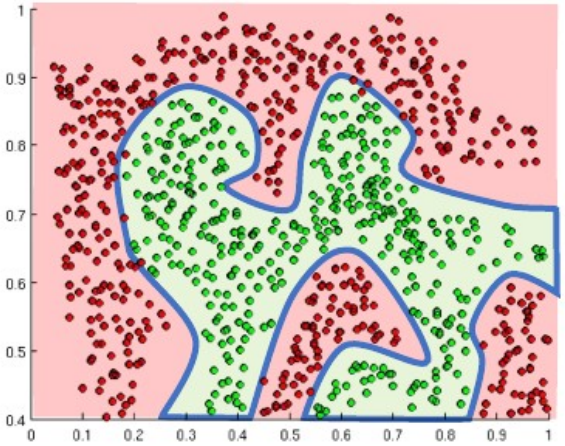
- What if we wanted to build a Neural Network to distinguish green versus red points?

Figs. from A. Ng (http://openclassroom.stanford.edu/)

IMPORTANCE OF ACTIVATION FCT.

- The purpose of activation functions is to introduce non-linearities into the network.

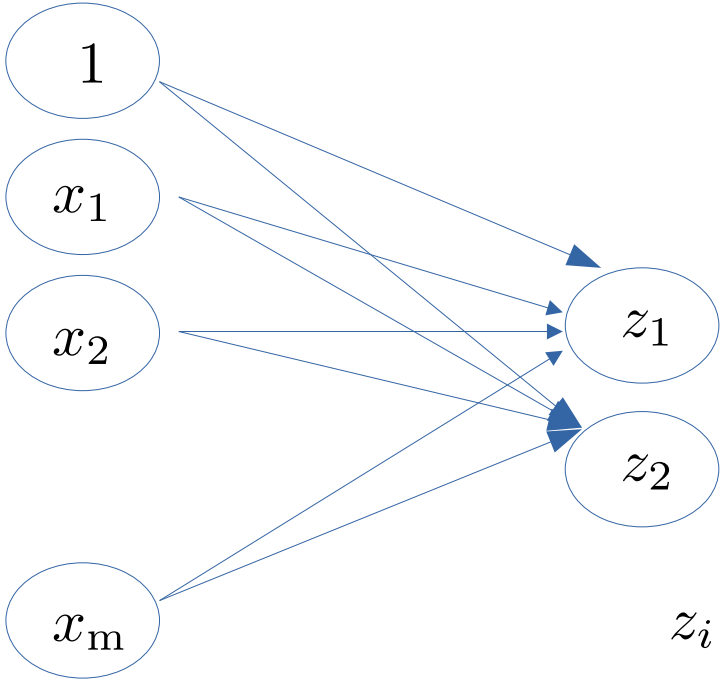




- Linear activation functions produce linear decisions no matter the network size.
- Non-linearities allow us to approximate arbitrarily complex functions.

Figs. from A. Ng (http://openclassroom.stanford.edu/)

BUILDING A NN WITH PERCEPTRONS: A MULTI-OUTPUT PERCEPTRON

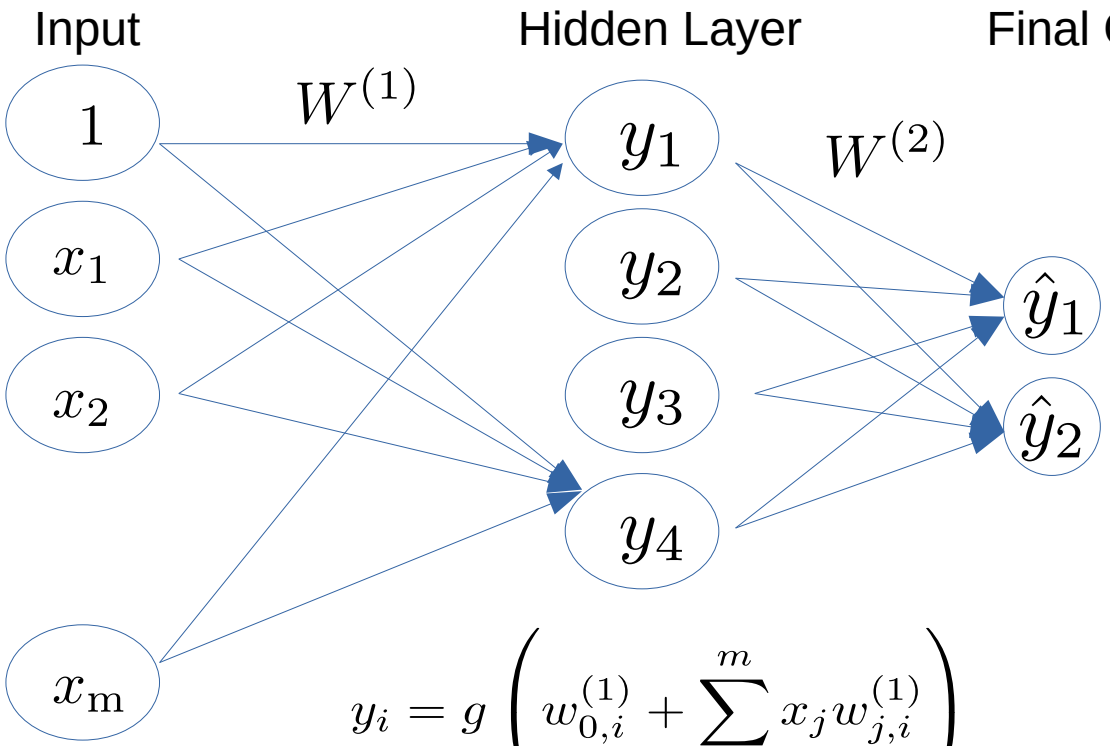


$$y_1 = g(z_1)$$

$$y_2 = g(z_2)$$

$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$$

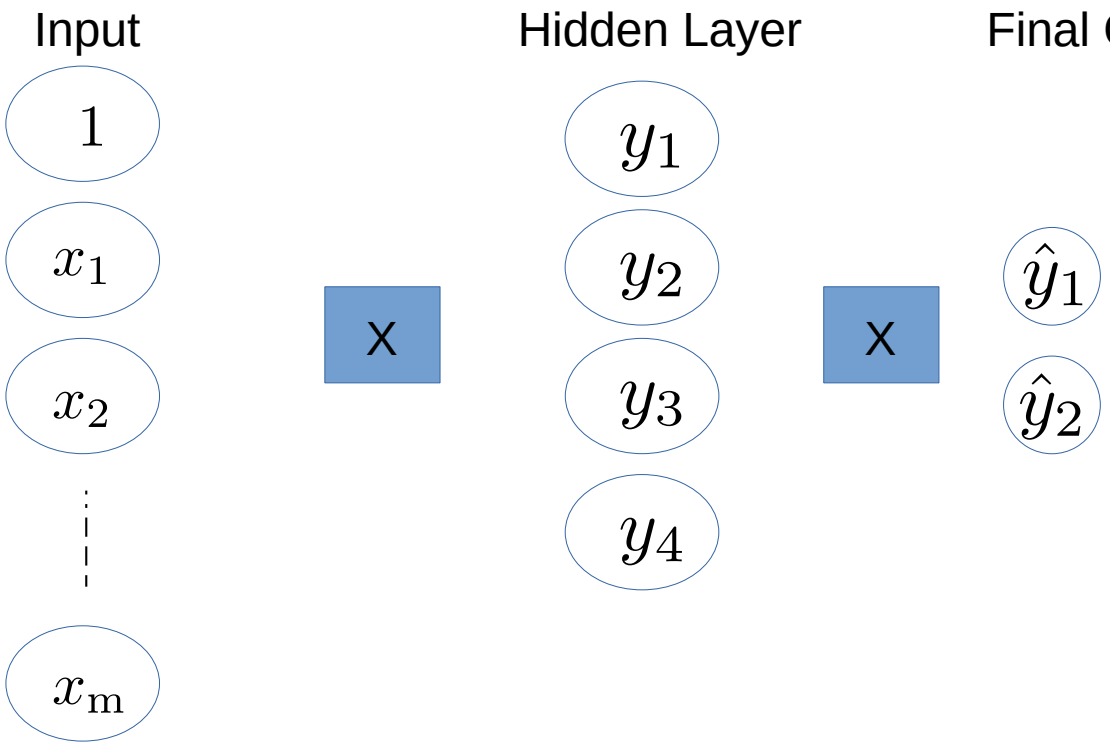
SINGLE HIDDEN LAYER NN



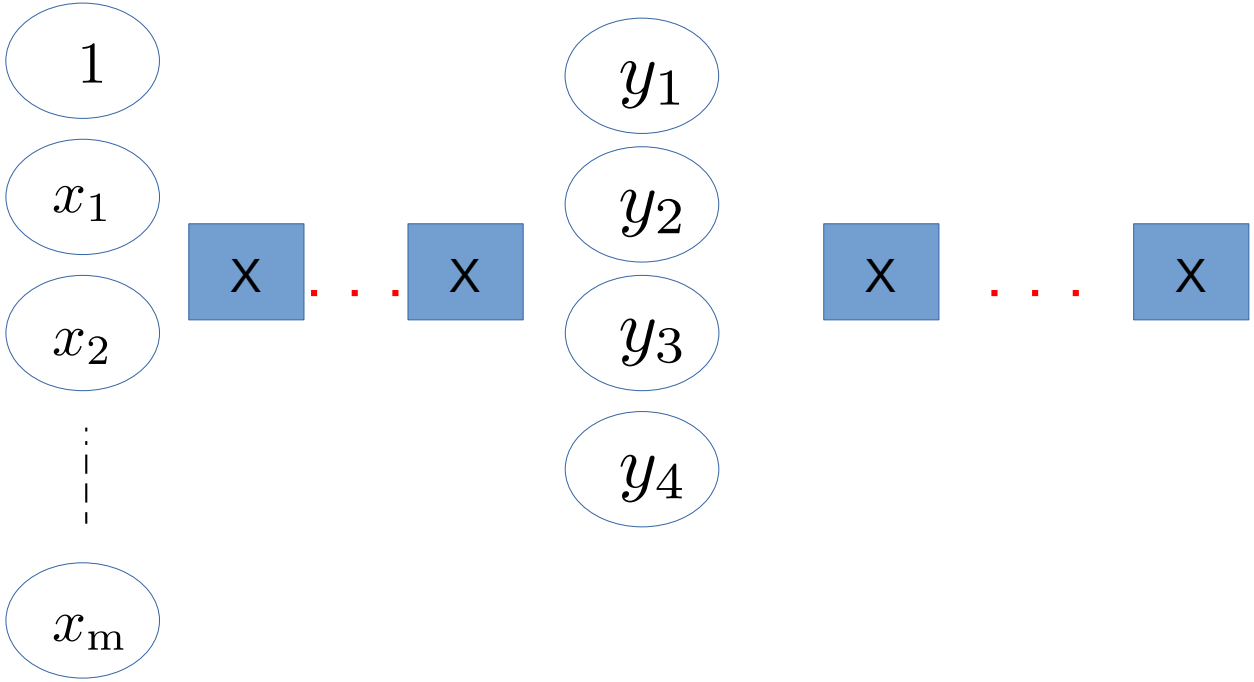
$$y_i = g\left(w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)}\right)$$

$$\hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d} y_j w_{j,i}^{(2)}\right)$$

SINGLE HIDDEN LAYER NN



FULLY CONNECTED DEEP NN



EXPRESSIVENESS OF ANN

- **Boolean functions:**
  - Every Boolean function can be represented by a network with a single hidden layer.
  - Might require exponential (in number of inputs) hidden units.

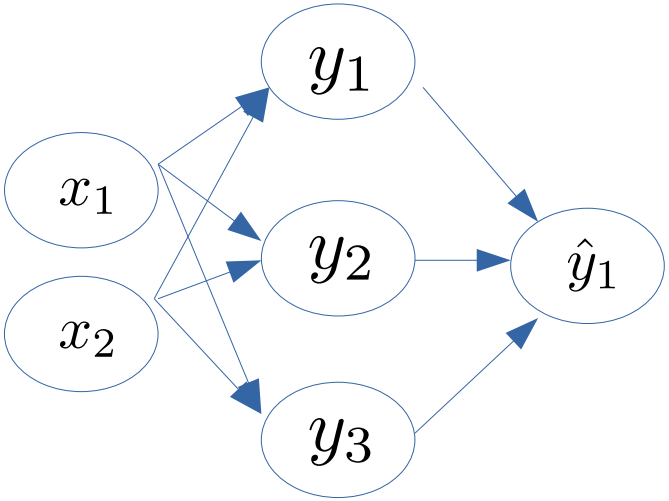
- **Continuous functions:**
  - Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989].

- Deep NN are in practice superior to other ML methods in presence of large data sets.

EMPIRICAL LOSS

The empirical loss measures the total loss over our entire data set.

$$\begin{bmatrix} 80.000, 200.000 \\ 120.000, 400.000 \\ 10.000, 12.000 \\ ..., ... \end{bmatrix}$$



Prediction: 0.2
Actual: 1

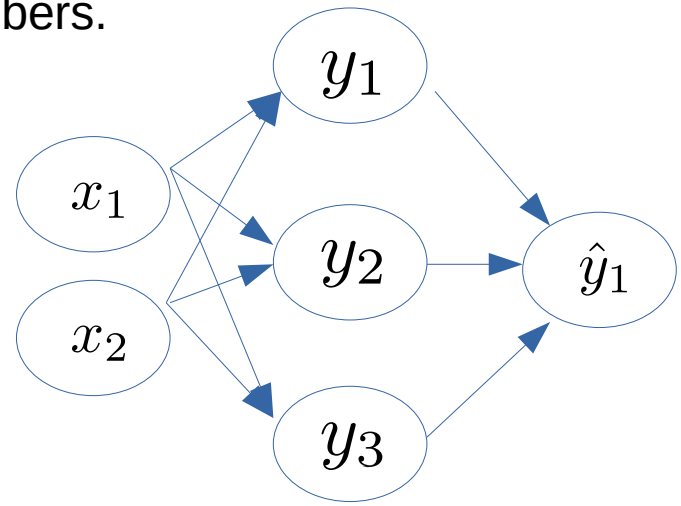
$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{J} \left(f \left(x^{(i)}; \boldsymbol{W} \right), \tilde{y}^{(i)} \right)$$

Predicted Actual

MEAN SQUARED ERROR (MSE)

Mean squared error can be used with regression models that output continuous real numbers.

$$\begin{bmatrix} 80.000, 200.000 \\ 120.000, 400.000 \\ 10.000, 12.000 \\ ..., ... \end{bmatrix}$$



f(x)

$$\begin{bmatrix} 450.000 \\ 250.000 \\ 190.000 \\ ..., ... \end{bmatrix}$$

y_{true}

$$\begin{bmatrix} 470.000 \\ 220.000 \\ 250.000 \\ ..., ... \end{bmatrix}$$

Loan requested

Loan required

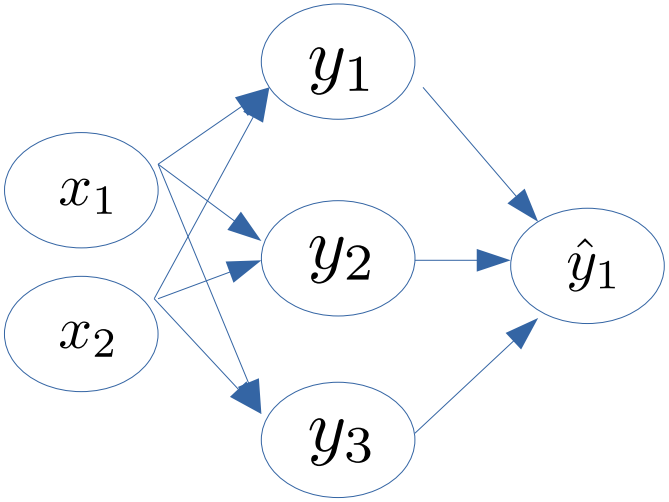
$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \left(y_{true}^{(i)} - f\left(x^{(i)}; W \right) \right)^2$$

Actual Predicted

BINARY CROSS ENTROPY LOSS

Our example was a classification problem with output (0 or 1)

$$\begin{bmatrix} 80.000, 200.000 \\ 120.000, 400.000 \\ 10.000, 12.000 \\ \ldots, \ldots \end{bmatrix}$$



f(x)

$$\begin{bmatrix} 0.2 \\ 0.9 \\ 0.6 \\ \ldots, \ldots \end{bmatrix}$$

y_{true}

$$\begin{bmatrix} 1 \\ 0 \\ 1 \\ \ldots, \ldots \end{bmatrix}$$

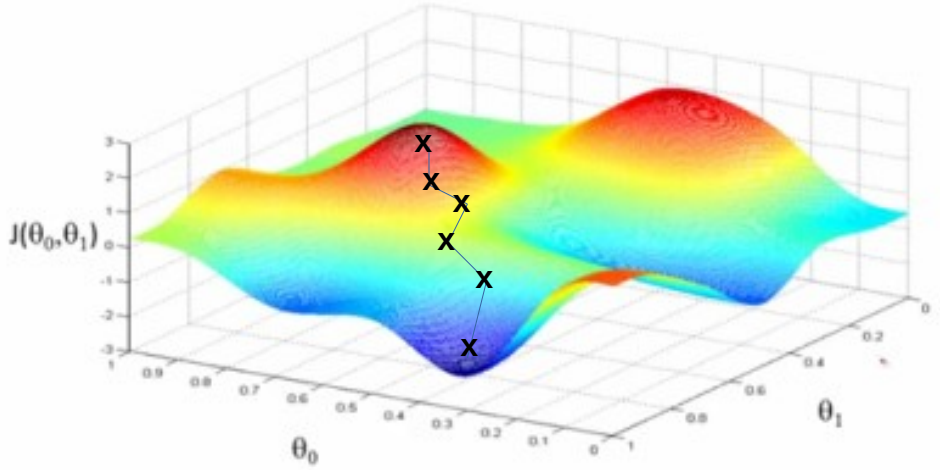
$$J(W) = \frac{1}{n}\sum_{i=1}^{n} y_{true}^{(i)} \log\left(f\left(x^{(i)}; W\right)\right) + \left(1 - y_{true}^{(i)}\right) \log\left(1 - f\left(x^{(i)}; W\right)\right)$$

Actual Predicted Actual Predicted

GRADIENT DESCENT IN WEIGHT SPACE

→ We want to find the network weights that achieve the lowest loss!

- $W^* = \underset{W}{\text{argmin}} J(W)$
- Randomly pick an initial(w_0, w_1)
- Compute gradient
- Take small steps in the opposite direction of gradient.
- Repeat until convergence



GRADIENT DESCENT ALGORITHM

Algorithm

I. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

2. Loop until convergence:

3. Compute gradient, $\frac{\partial J(W)}{\partial W}$ ← Can be computationally expensive

4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial W}$

5. Return weights

GRADIENT DESCENT ALGORITHM

Algorithm

I. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

2. Loop until convergence:

3. Compute gradient, $\frac{\partial J(W)}{\partial W}$

4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial W}$

5. Return weights

All that matters to train a NN. Computationally expensive!

Learning rate

STOCHASTIC GRADIENT DESCENT

Algorithm

I. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

2. Loop until convergence:

3. Pick single data point i

4. Compute gradient, $\frac{\partial J_i(W)}{\partial W}$

5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

6. Return weights

Can be noisy

STOCHASTIC GRADIENT DESCENT

Algorithm

I. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

2. Loop until convergence:

3. Pick batch of B data points

4. Compute gradient, $\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$

5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

6. Return weights

STOCHASTIC GRADIENT DESCENT

Consider the term $\sum_{n=1}^{N}(\nabla L_n(\boldsymbol{\theta}_i))$ in (7.15). We can reduce the amount of computation by taking a sum over a smaller set of L_n. In contrast to batch gradient descent, which uses all L_n for $n = 1, \ldots, N$, we randomly choose a subset of L_n for mini-batch gradient descent. In the extreme case, we randomly select only a single L_n to estimate the gradient. The key insight about why taking a subset of data is sensible is to realize that for gradient descent to converge, we only require that the gradient is an unbiased estimate of the true gradient. In fact the term $\sum_{n=1}^{N}(\nabla L_n(\boldsymbol{\theta}_i))$ in (7.15) is an empirical estimate of the expected value (Section 6.4.1) of the gradient. Therefore, any other unbiased empirical estimate of the expected value, for example using any subsample of the data, would suffice for convergence of gradient descent.

*cf. Deisenroth et al. (2021) Mathematics for Machine learning, Sec. 7.1.3.

MINI-BATCHES WHILE TRAINING

- More accurate estimation of gradient
  - Smoother convergence
  - Allows for larger learning rates

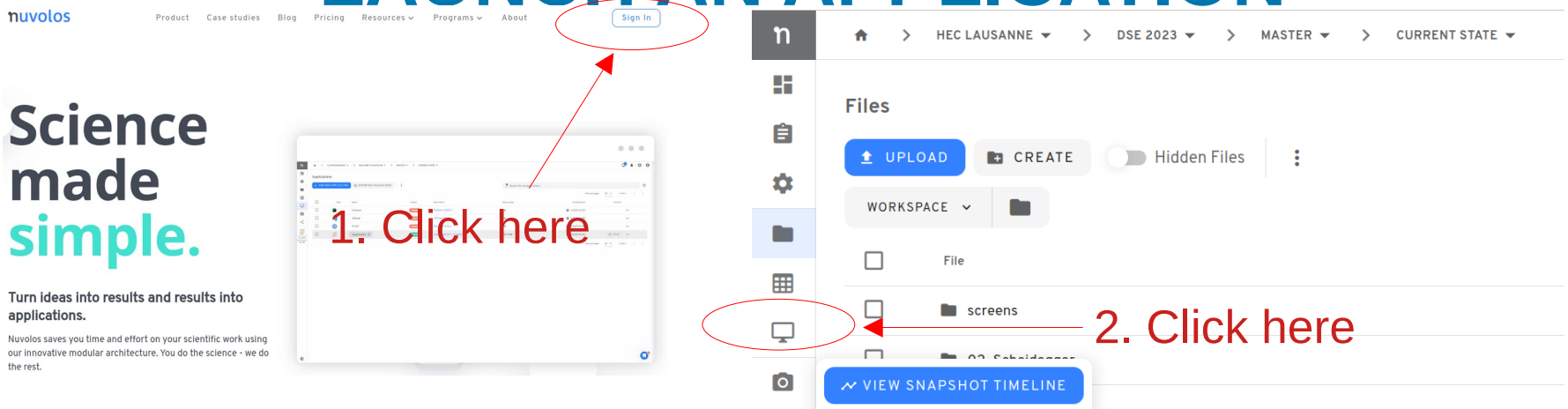
- Mini-batches lead to fast training!
  - Can parallelize computation + achieve significant speed increases on GPU's

- Note: a complete pass over all the patterns in the training set is called an **epoch**.

INTERMEZZO – ACTION REQUIRED

- Let's look at this notebook on Nuvolos.cloud: nuvolos
- **lectures/day1/code/01_GradientDescent_and_StochasticGradientDescent.ipynb**
- If you are not familiar with Jupyter Notebooks, look at **python_refresher/jupyter_intro.ipynb**

LOG INTO NUVOLOS.CLOUD & LAUNCH AN APPLICATION

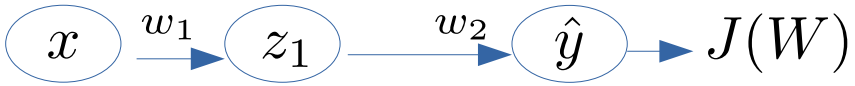


Type	Name	Size	Credit/hour	Status	Description	Space usage
jupyter	tb2	1 CU	-		JupyterLab 3.1 + Python 3.9 with Dash support	4.5 GB
jupyter	Deep Learning For Econ	1 CU	-		JupyterLab 3 + Python 3.8 Essentials	4.5 GB
	VSCode 1.88.1	1 CU	-		VSCode 1.88.1 + Py3.11	157.8 MB

3. Click here

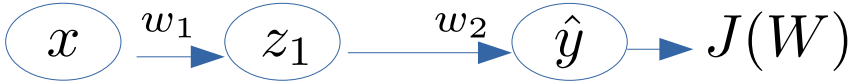
COMPUTING GRADIENTS: ERROR BACKPROPAGATION

- How does a small change in one weight (e.g., w_2) affect the final loss J(W)?



COMPUTING GRADIENTS: ERROR BACKPROPAGATION

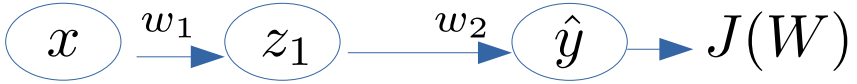
- How does a small change in one weight (e.g., w_2) affect the final loss J(W)?
- Chain rule



$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

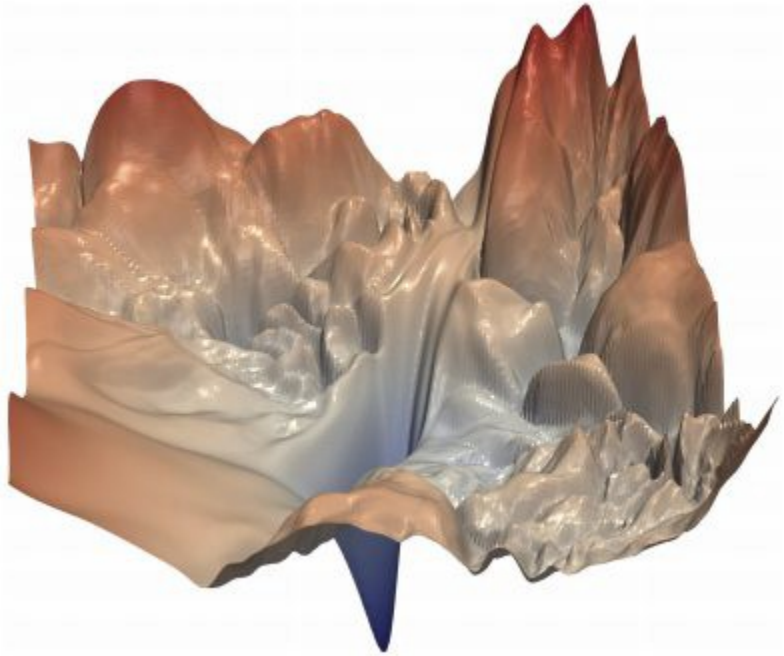
COMPUTING GRADIENTS: ERROR BACKPROPAGATION

- How does a small change in one weight (e.g., w_2) affect the final loss J(W)?
- Chain rule
- **Repeat this for every weight in the network using gradients from later layers**



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1} \longrightarrow \frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

TRAINING NEURAL NETWORKS



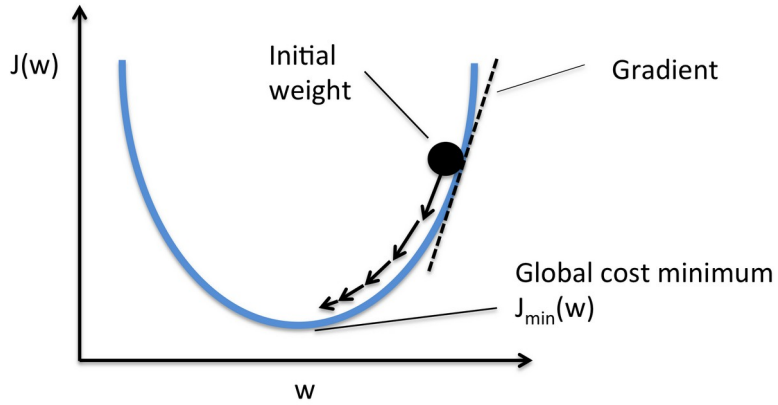
See https://papers.nips.cc/paper/7875-visualizing-the-loss-landscape-of-neural-nets.pdf

LOSS FUNCTION: CAN BE DIFFICULT TO OPTIMIZE

- Remember:
  - Optimization through gradient descent: $$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$
  - How can we set the learning rate?

SETTING THE LEARNING RATE

- Small learning rate converges slowly and gets stuck in false local minima
  - Design an adaptive learning rate that “adapts” to the landscape.



FEW VARIANTS OF SGD

Method	Formula
Learning Rate	$w^{(t+1)} = w^{(t)} - \eta \cdot \nabla \ell(w^{(t)}, z) \quad = w^{(t)} - \eta \cdot \nabla w^{(t)}$
Adaptive Learning Rate	$w^{(t+1)} = w^{(t)} - \eta_t \cdot \nabla w^{(t)}$
Momentum [Qian 1999]	$w^{(t+1)} = w^{(t)} + \mu \cdot (w^{(t)} - w^{(t-1)}) - \eta \cdot \nabla w^{(t)}$
Nesterov Momentum [Nesterov 1983]	$w^{(t+1)} = w^{(t)} + v_t; \quad v_{t+1} = \mu \cdot v_t - \eta \cdot \nabla \ell(w^{(t)} - \mu \cdot v_t, z)$
AdaGrad [Duchi et al. 2011]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A_{i,t}} + \varepsilon}; \quad A_{i,t} = \sum_{\tau=0}^{t} \left(\nabla w_i^{(t)}\right)^2$
RMSProp [Hinton 2012]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A'_{i,t}} + \varepsilon}; \quad A'_{i,t} = \beta \cdot A'_{t-1} + (1-\beta)\left(\nabla w_i^{(t)}\right)^2$
Adam [Kingma and Ba 2015]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot M_{i,t}^{(1)}}{\sqrt{M_{i,t}^{(2)}} + \varepsilon}; \quad M_{i,t}^{(m)} = \frac{\beta_m \cdot M_{i,t-1}^{(m)} + (1-\beta_m)\left(\nabla w_i^{(t)}\right)^m}{1-\beta_m^t}$

For more details, see https://arxiv.org/pdf/1609.04747.pdf, and also https://ruder.io/optimizing-gradient-descent/

WEIGHT INITIALIZATION

- Before the training process starts: all weights vectors must be initialized with some numbers.
- There are many initializers of which random initialization is one of the most widely known ones (e.g., with a normal distribution).
  - Specifically, one can configure the mean and the standard deviation, and once again seed the distribution to a specific (pseudo-)random number generator.
  - which distribution to use, then?
  - random initialization itself can become problematic under some conditions: you may then face the vanishing gradients and exploding gradients problems.
- What to do against these problems?
  - e.g. Xavier & He initialization (available in Keras)
  - They are different in the way how they manipulate the drawn weights to arrive at approximately 1. By consequence, they are best used with different activation functions.
  - Specifically, He initialization is developed for ReLU based activating networks and by consequence is best used on those. For others, Xavier (or Glorot) initialization generally works best.

See, e.g., Glorot & Bengio (2010) - http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf; He et al. (2015) - https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf

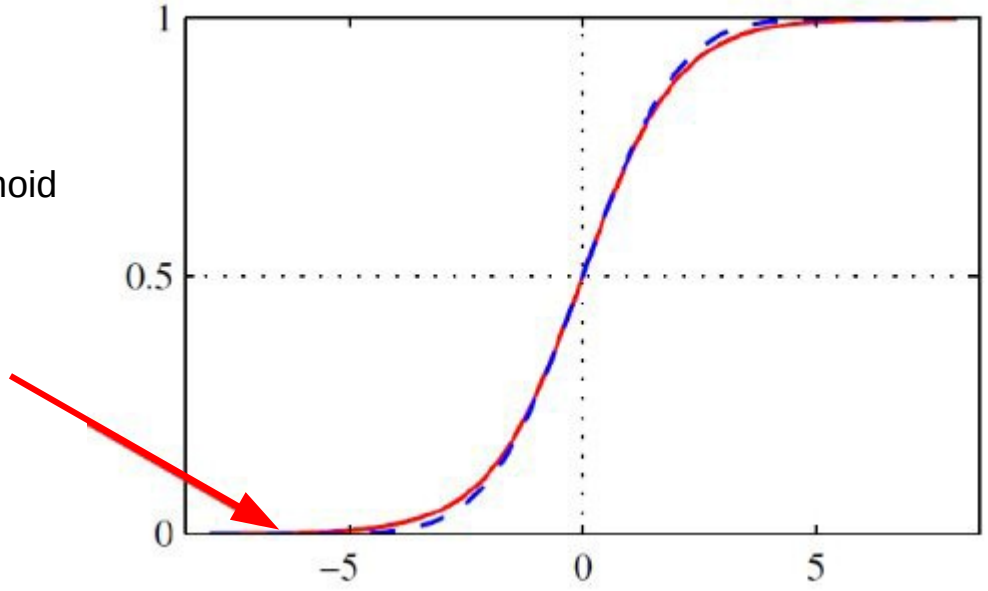
VANISHING GRADIENTS

- Deep learning community often deals with two types of problems during training: vanishing gradients (and exploding) gradients.
  - Vanishing gradients
    - the backpropagation algorithm, which chains the gradients together when computing the error backwards, will find really small gradients towards the left side of the network (i.e., farthest from where error computation started).
    - This problem primarily occurs e.g. with the Sigmoid and Tanh activation functions, whose derivatives produce outputs of $0 < x' < 1$, except for Tanh which produces $x' = 1$ at $x = 0$.
    - Consequently, when using Tanh and Sigmoid, you risk having a suboptimal model that might possibly not converge due to vanishing gradients.
    - ReLU does not have this problem – its derivative is 0 when $x < 0$ and is 1 otherwise.
    - It is computationally faster. Computing this function – often by simply maximizing between (0, x) – takes substantially fewer resources than computing e.g. the sigmoid and tanh functions. By consequence, ReLU is the de facto standard activation function in the deep learning community today.

VANISHING GRADIENTS

Problem with Sigmoid
→ Saturation

Gradient too small



VANISHING GRADIENTS

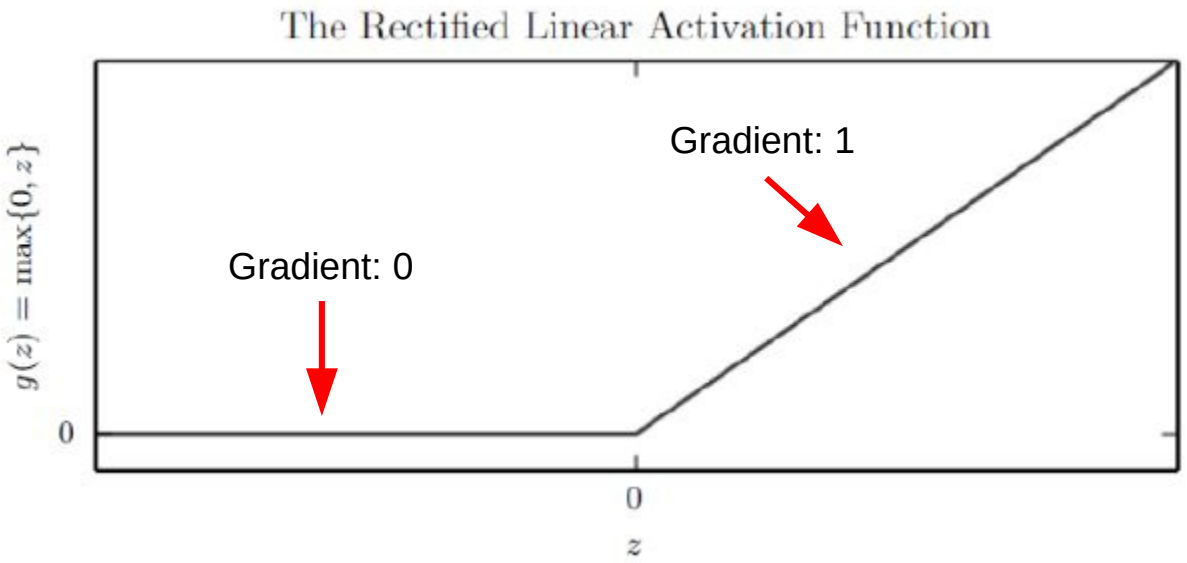
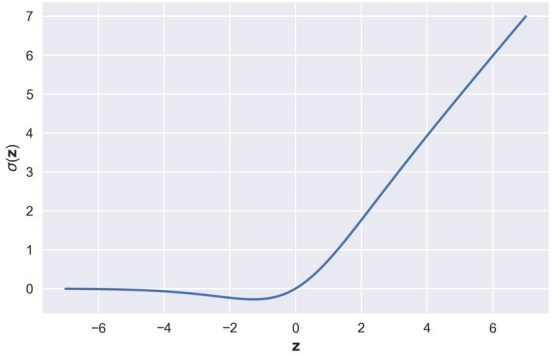


Fig. From Goodfellow et al. (2016)

SWISH ACTIVATION FUNCTION

- Nevertheless, it does not mean that it cannot be improved.
  - Swish activation function.
  - Instead, it does look like the de-facto standard activation function, with one difference: the domain around 0 differs from ReLU.
- Swish is a smooth function. That means that it does not abruptly change direction like ReLU does near x = 0.
  - Swish is non-monotonic. It thus does not remain stable or move in one direction, such as ReLU.
  - It is in fact this property which separates Swish from most other activation functions, which do share this monotonicity.
- In applications - Swish could be better than ReLU.

$$f(x) = x * \text{sigmoid}(x)$$
$$= x * \left(1 + e^{-x}\right)^{-1}$$



See, P Ramachandran, B Zoph, Q V. Le (2017) - https://arxiv.org/pdf/1710.05941v1.pdf

A GEOMETRIC INTERPRETATION

- In 3D, the following mental image may prove useful. Imagine two sheets of colored paper: one red and one blue.
- Put one on top of the other.
- Crumple them together into a small ball. That crumpled paper ball is your input data, and each sheet of paper is a class of data in a classification problem.
- What a neural network (or any other machine-learning model) is meant to do is figure out a **transformation of the paper** ball that would uncrumple it, so as to make the two classes cleanly separable again.
- With deep learning, this would be implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time.

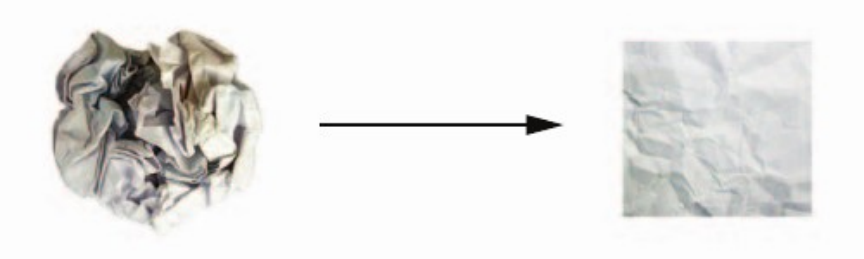
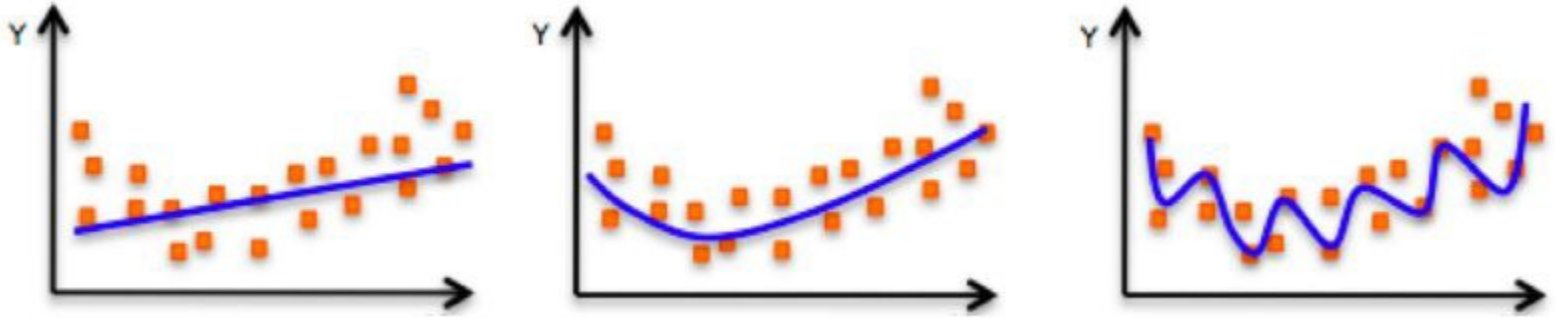


Figure 2.9 Uncrumpling a complicated manifold of data

Fig. from Chollet 2017

NOTES ON OVERFITTING



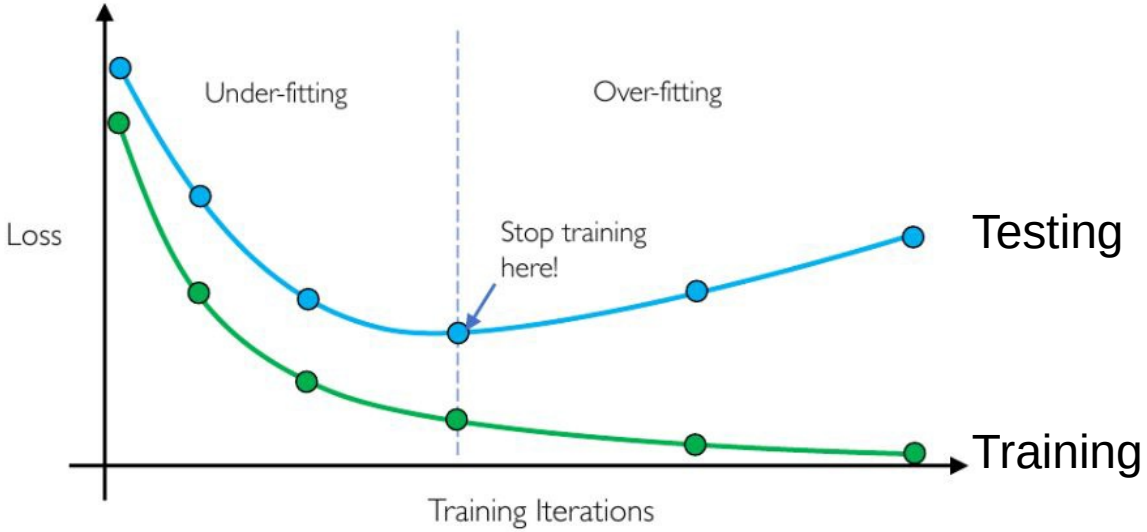
Underfitting
Model does not have capacity to fully learn the data

Ideal Fit

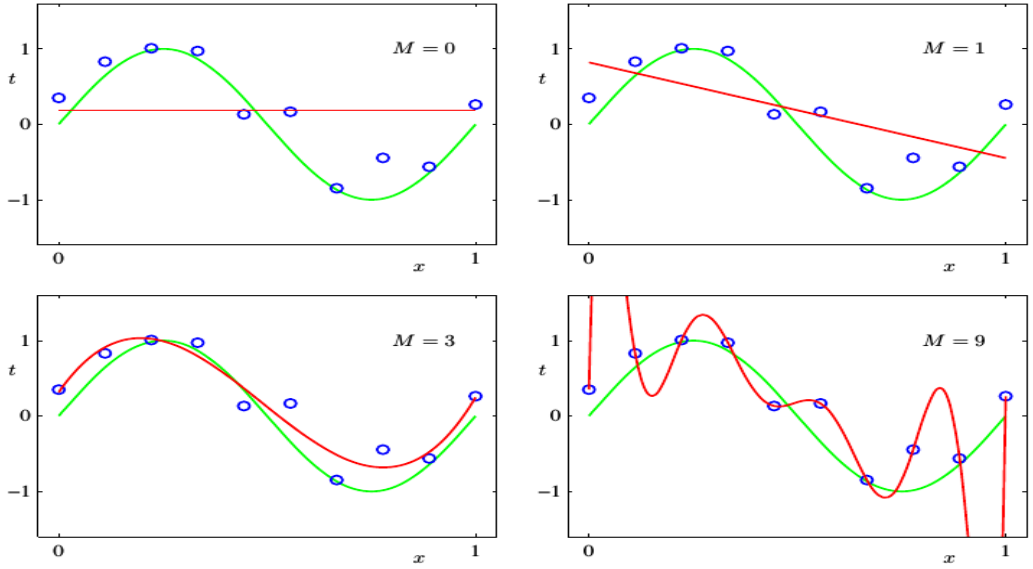
Overfitting
Too complex, extra parameters, does not generalize well

EARLY STOPPING

- Stop training before we have a chance to overfit



NOTES ON OVERFITTING (II)



Polynomial Curve Fitting of polynomials having various orders M, shown as red curves, fitted to the data set shown above.

NOTES ON OVERFITTING (III)

- So far, we have assessed the prediction **quality of our model based on the same data that we used for training**.
- This is a **very bad idea**, since we can not accurately **measure how well our model works for previously unseen data points** (e.g., for cars not in our dataset)
- Our model may over-fit to the training data and loose its ability to make predictions.

→ Next, we'll see some best practices for evaluating machine learning models

SPLIT THE DATA

- To avoid over-fitting, it is good practice to assess the quality of a model based on test data that **must not be used** for training the model.

- The key idea is to split the available data (randomly) into **training**, **validation**, and **test data**.

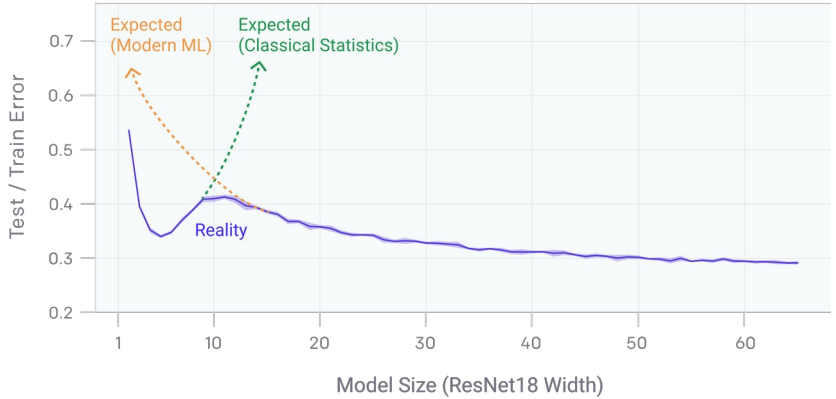
SPLITTING THE DATA

- One common approach to reliably assess the quality of a machine learning model and avoid over-fitting is to randomly split the available data into
  - **training data (~70% of the data)**
    is used for determining optimal coefficients.
  - **validation data (~20% of the data) is used for model selection** (e.g., fixing degree of polynomial, selecting a subset of features, etc.)
  - **test data (~10% of the data)** is used to measure the quality that is reported.

DEEP DOUBLE DESCENT

https://arxiv.org/abs/1912.02292

- The double descent phenomenon occurred so far in practical applications
  - performance first improves, then gets worse, and then improves again with increasing model size, data size, or training time.
  - This effect is often avoided through careful regularization.
  - While this behavior appears to be fairly universal, we don’t yet fully understand why it happens, and view further study of this phenomenon as an important research direction.

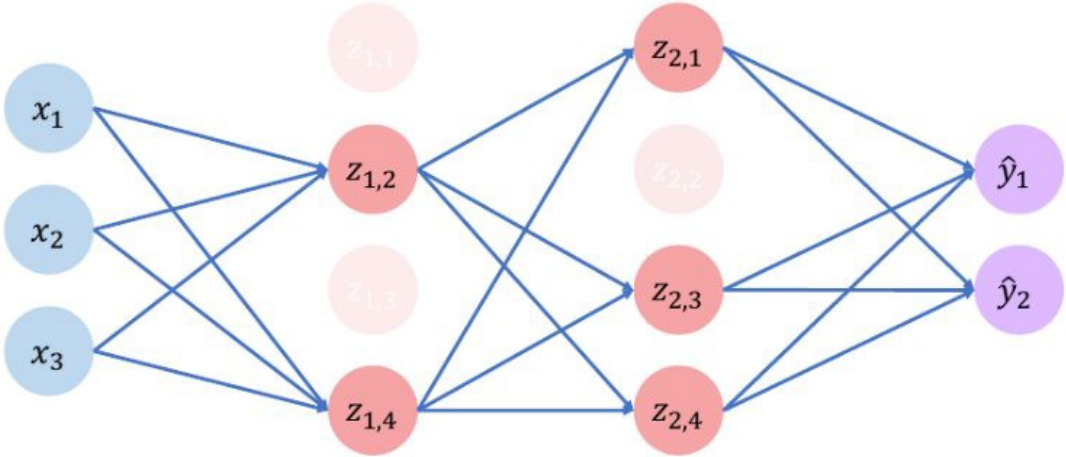


NOTES ON REGULARIZATION

- Regularization is a technique that constrains our optimization problem to discourage complex models.
- We use it to improve generalization of our model on unseen data.

REGULARIZATION IN NN: DROPOUT

- During training, randomly set some activations to 0
  - Typically 'drop' 50 % of activations in layer
  - Forces network to not rely on any node



Srivastava et al., 2014 – http://jmlr.org/papers/v15/srivastava14a.html

*REGULARIZATION IN NN: DROPOUT

- It is an efficient way of performing model averaging with neural networks.
- Can be interpreted as some sort of **bagging**.
- Now, we assume that the model's role is to output a probability distribution. In the case of bagging, each model i produces a probability distribution $p^{(i)}(y \mid x)$.
- The prediction of the ensemble is given by the arithmetic mean of all these distributions:

$$\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y \mid \boldsymbol{x})$$

- In the case of dropout, each sub-model defined by mask vector μ defines a probability distribution p(y | x, μ).
- The arithmetic mean over all masks is given by $\sum_{\mu} p(\boldsymbol{\mu}) p(y \mid \boldsymbol{x}, \boldsymbol{\mu})$ where p(μ) is the probability distribution that was used to sample μ at training time.

RECIPE FOR USING MLP

- **Select inputs and outputs for your problem**
  - Before anything else, you need to think about the problem you are trying to solve, and make sure that you have data for the problem, both input vectors and target outputs.
  - At this stage you need to choose what features are suitable for the problem and decide on the output encoding that you will use — standard neurons, or linear nodes.
  - These things are often decided for you by the input features and targets that you have available to solve the problem.
  - Later on in the learning it can also be useful to re-evaluate the choice by training networks with some input feature missing to see if it improves the results at all.

RECIPE FOR USING MLP (II)

- **Normalize inputs**
  - Re-scale the data by subtracting the mean value from each element of the input vector, and divide by the variance (or alternatively, either the maximum or minus the minimum, whichever is greater).

- **Split the data into training, testing, and validation sets**
  - You cannot test the learning ability of the network on the same data that you trained it on, since it will generally fit that data very well (often too well, over-fitting and modeling the noise in the data as well as the generating function).

  - Recall: we generally split the data into three sets, one for training, one for testing, and then a third set for validation, which is testing how well the network is learning during training.

RECIPE FOR USING MLP (III)

- **Select a network architecture**
  - You already know how many input nodes there will be, and how many output neurons.
  - You need to consider whether you will need a hidden layer at all, and if so how many neurons it should have in it.
  - You might want to consider more than one hidden layer.
  - The more complex the network, the more data it will need to be trained on, and the longer it will take.
  - It might also be more subject to over-fitting.
  - The usual method of selecting a network architecture is to try several with different numbers of hidden nodes and see which works best.

RECIPE FOR USING MLP (IV)

- **Train a network**
  - The training of the NN consists of applying the MLP algorithm to the training data.
  - This is usually run in conjunction with early stopping, where after a few iterations of the algorithm through all of the training data, the generalization ability of the network is tested by using the validation set.
  - The NN is very likely to have far too many degrees of freedom for the problem, and so after some amount of learning it will stop modeling the generating function of the data, and start to fit the noise and inaccuracies inherent in the training data. At this stage the error on the validation set will start to increase, and learning should be stopped.

- **Test the network**
  - Once you have a trained network that you are happy with, it is time to use the test data for the first (and only) time. This will enable you to see how well the network performs on some data that it has not seen before, and will tell you whether this network is likely to be usable for other data, for which you do not have targets.

KERAS & TENSORFLOW BASICS

- tensorflow.org 

- Keras API:
  https://www.tensorflow.org/guide/keras/sequential_model

- Fun data sets to play with: https://www.kaggle.com/datasets

- Some “clean” data to play with: https://archive.ics.uci.edu/ml/index.php

- Help for debugging – Tensorboard: https://www.tensorflow.org/tensorboard

GOOD NEWS

https://keras.io/keras_core/announcement/

Introducing Keras Core: Keras for TensorFlow, JAX, and PyTorch.

Get started | API docs | Guides | GitHub

We're excited to share with you a new library called **Keras Core**, a preview version of the future of Keras. In Fall 2023, this library will become Keras 3.0. Keras Core is a full rewrite of the Keras codebase that rebases it on top of a **modular backend architecture**. It makes it possible to run Keras workflows on top of arbitrary frameworks — starting with TensorFlow, JAX, and PyTorch.

Keras Core is also a drop-in replacement for `tf.keras`, with near-full backwards compatibility with `tf.keras` code when using the TensorFlow backend. In the vast majority of cases you can just start importing it via `import keras_core as keras` in place of `from tensorflow import keras` and your existing code will run with no issue — and generally with slightly improved performance, thanks to XLA compilation.

A GENTLE FIRST EXAMPLE

- Lets look at the notebook: ***02_Gentle_DNN.ipynb***.
- This Notebook contains all the basic functionality from a theoretical point of view.
- 2 simple examples, one regression, and one classification.

ACTION REQUIRED

Look at the test functions below*. Pick three of those test functions (from Genz 1987).

- Approximate a 2-dimensional function stated below with Neural Nets based 10, 50, 100, 500 points randomly sampled from $[0 , 1]^2$. Compute the average and maximum error.
- The errors should be computed by generating 1,000 uniformly distributed random test points from within the computational domain.
- Plot the maximum and average error as a function of the number of sample points.
- Repeat the same for 5-dimensional and 10-dimensional functions. Is there anything particular you observe?

$$\text{oscillatory:}\quad f_1(x) = \cos\left(2\pi w_1 + \sum_{i=1}^{d} c_i x_i\right),$$

$$\text{product peak:}\quad f_2(x) = \prod_{i=1}^{d} \left(c_i^{-2} + (x_i - w_i)^2\right)^{-1},$$

$$\text{corner peak:}\quad f_3(x) = \left(1 + \sum_{i=1}^{d} c_i x_i\right)^{-(d+1)},$$

$$\text{Gaussian:}\quad f_4(x) = \exp\left(-\sum_{i=1}^{d} c_i^2 \cdot (x_i - w_i)^2\right),$$

$$\text{continuous:}\quad f_5(x) = \exp\left(-\sum_{i=1}^{d} c_i \cdot |x_i - w_i|\right),$$

$$\text{discontinuous:}\quad f_6(x) = \begin{cases} 0, & \text{if } x_1 > w_1 \text{ or } x_2 > w_2, \\ \exp\left(\sum_{i=1}^{d} c_i x_i\right), & \text{otherwise.} \end{cases}$$

Varying test functions can be obtained by altering the parameters $c = (c_1, \ldots, c_n)$ and $w = (w_1, \ldots, w_n)$. We chose these parameters randomly from $[0, 1]$. Similarly to Barthelmann et al. [2000], we normalized the c_i such that $\sum_{i=1}^{d} c_i = b_j$, with b_j depending on d, f_j according to

j	1	2	3	4	5	6
b_j	1.5	d	1.85	7.03	20.4	4.3

Furthermore, we normalized the w_i such that $\sum_{i=1}^{d} w_i = 1$.

*Choose the parameters w and c in meaningful ways.

ACTION REQUIRED (II)

- Play with the architecture.
  - Number of hidden layers.
  - activation functions.
  - choice of the stochastic gradient descent algorithm.
  - Monitor the performance with respect to the architecture.

TENSORBOARD

- **lectures/day1/code/03_Tensorboard.ipynb**
- → Introduction to Tensorboard (https://www.tensorflow.org/tensorboard)
- Play with it in your spare time

QUESTIONS?