ADVANCED ARTIFICIAL INTELLIGENCE: COMPUTER VISION

Day 1

Schedule

- Day 1:
  - review of logistic regression, feedforward networks, data loaders
- Day 2
  - convolutional networks, training and optimization
- Day 3
  - generative networks: GANs, and variational autoencoders
- Day 4
  - a "paper" quiz, a classification exercise, a generative exercise

Notes, code snippets, etc. to be posted at
`https://georgeturk.github.io/ai2`

Program Staff

- George Turkiyyah (email: george.turkiyyah@kaust.edu.sa)
- Fares Fourati, TA
- Selma Kharrat, TA
- Somayah Al Gashgari, admin

Supervised vs unsupervised ML

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn function to map
$x \rightarrow y$

Examples: Classification, regression, object detection, semantic segmentation, etc.

Unsupervised Learning

Data: x
x is data, no labels!

Goal: Learn the *hidden* or *underlying structure* of the data

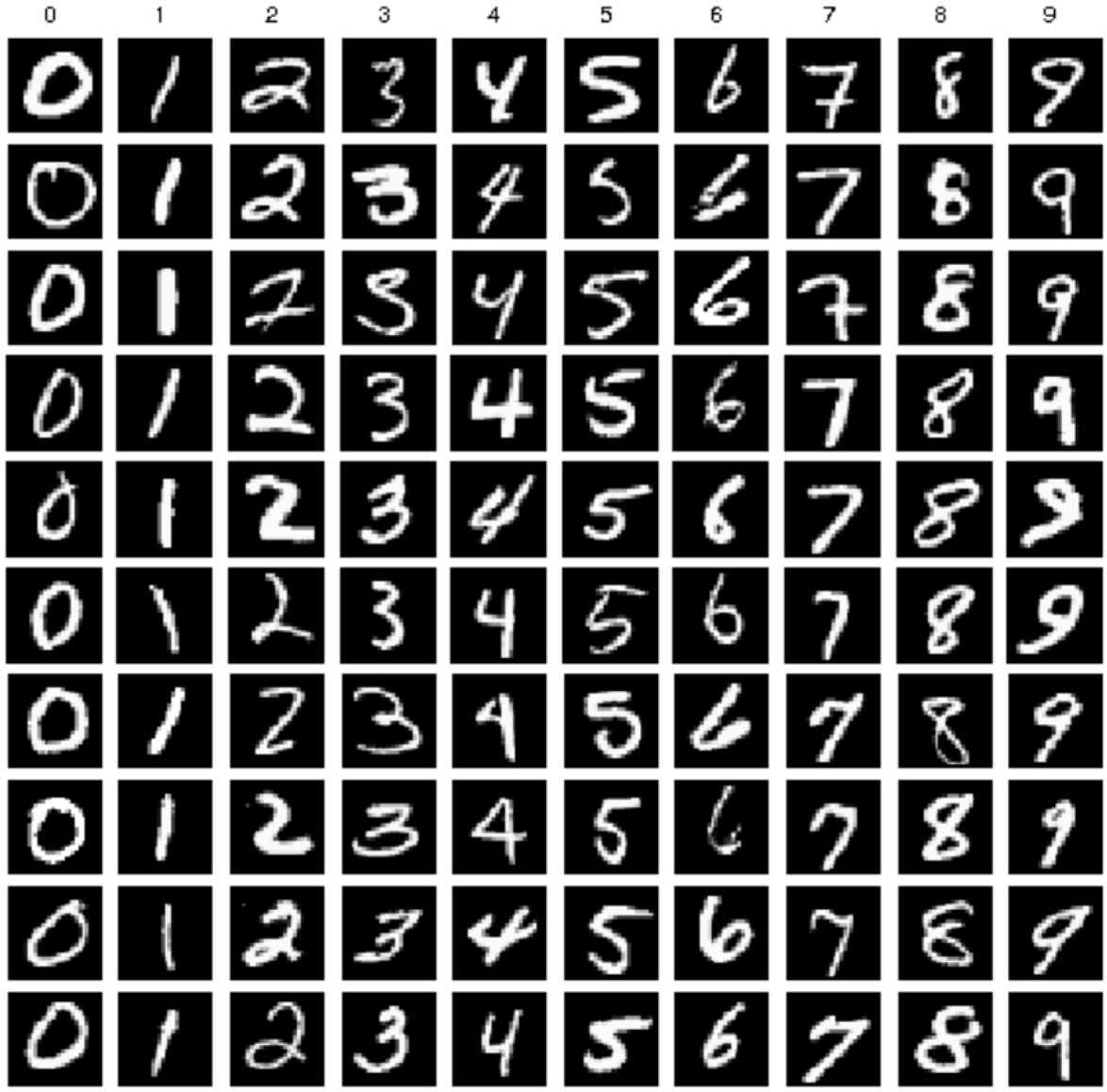
Examples: Clustering, feature or dimensionality reduction, etc.

Example

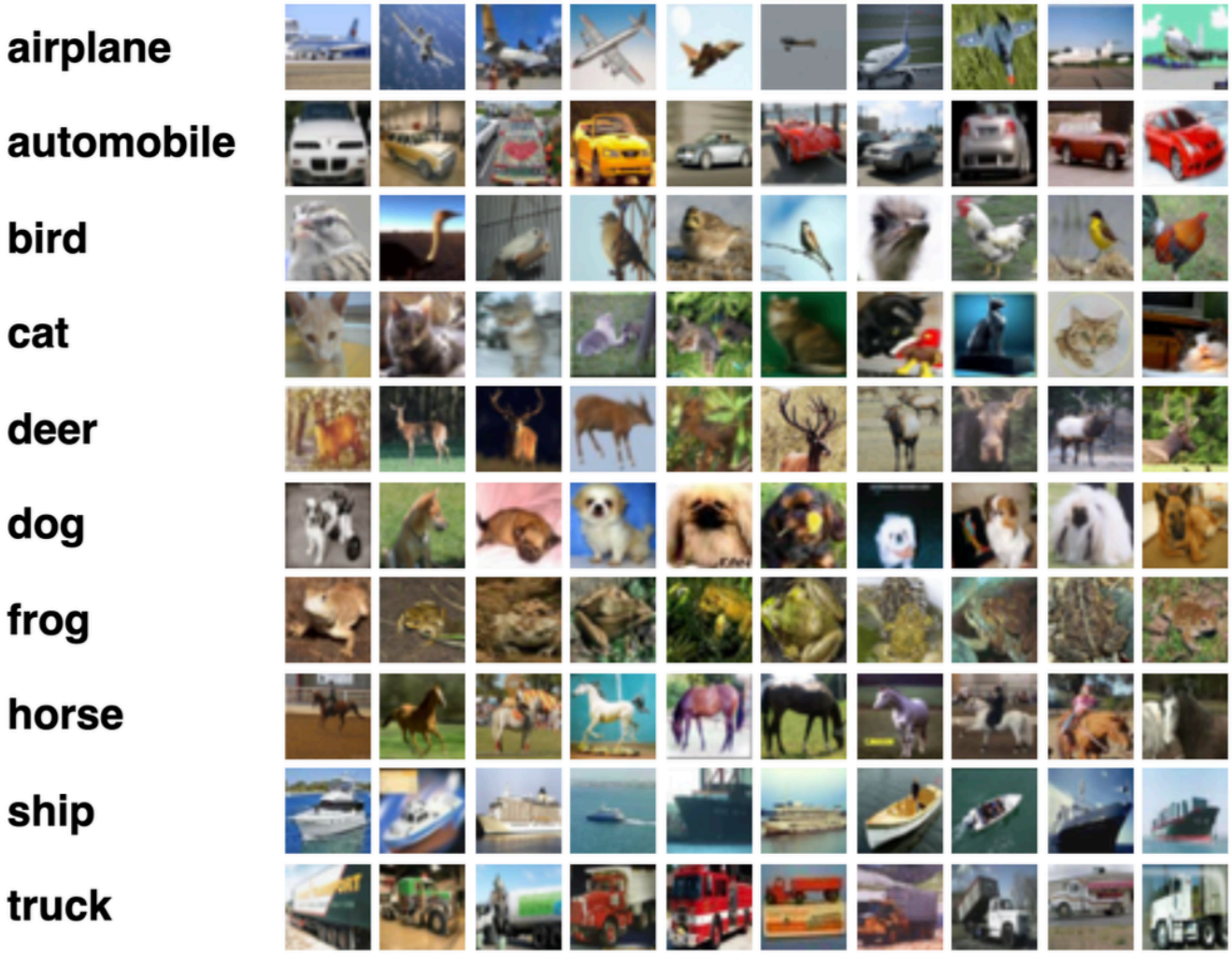


index	sl	sw	pl	pw	label
0	5.1	3.5	1.4	0.2	Setosa
1	4.9	3.0	1.4	0.2	Setosa
	...				
50	7.0	3.2	4.7	1.4	Versicolor
	...				
149	5.9	3.0	5.1	1.8	Virginica

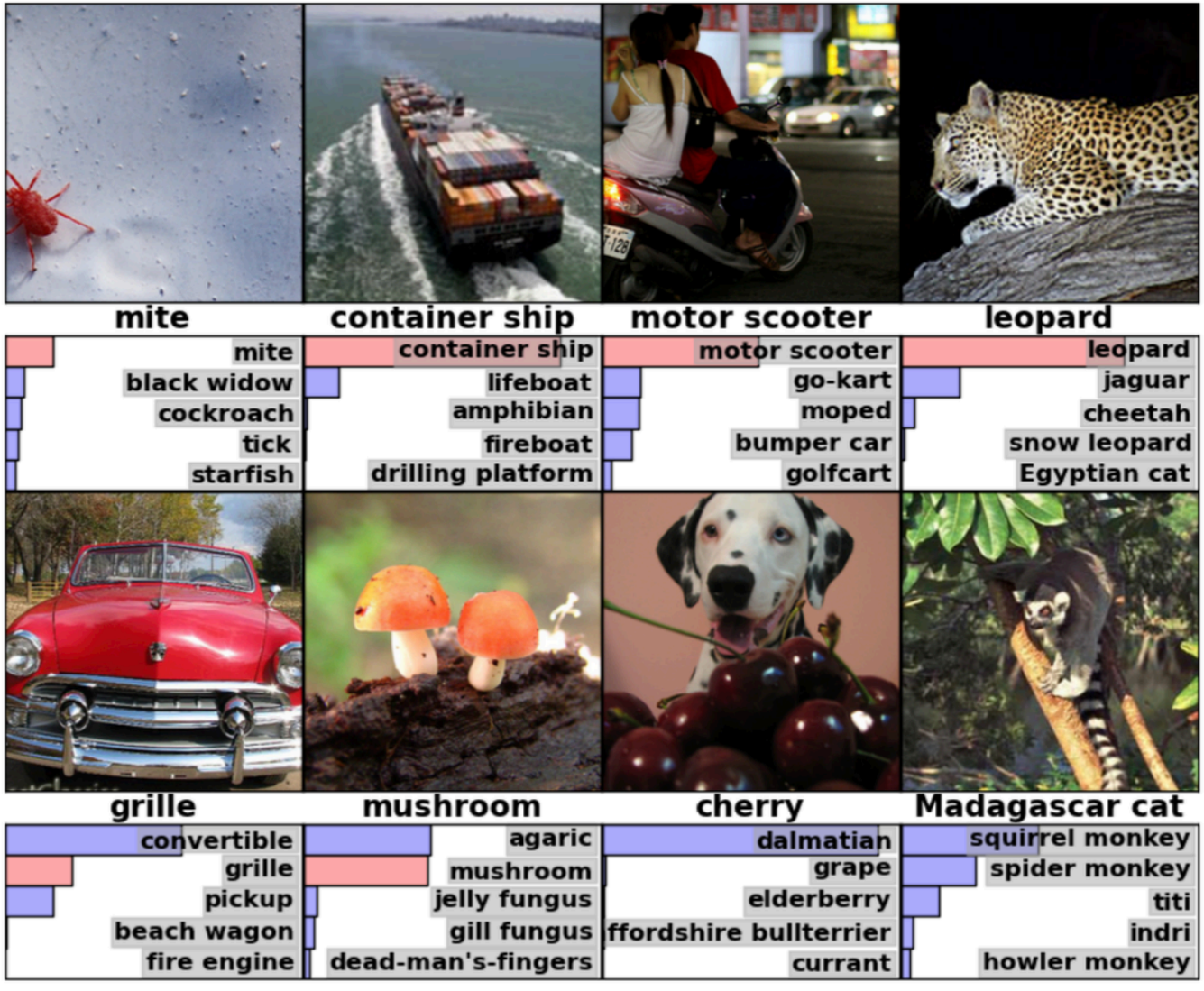
Example: MNIST



Example: CIFAR10



Example: ImageNet



Example: IMDB

A wonderful little production. The filming technique is very unassuming- very old-time-B...	positive
I thought this was a wonderful way to spend time on a too hot summer weekend, sitting in the air con...	positive
Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & his par...	negative

Example



The supervised learning problem

- mapping from inputs $x \in \mathcal{X}$ to outputs $y \in \mathcal{Y}$
- inputs x are also called *features*. Often a fixed-dimensional vector of numbers, such as the height and weight of a person, or the pixels in an image. $\mathcal{X} = \mathbb{R}^D$
- outputs y also called *labels*. In prediction problems, y is a real value, $\mathcal{Y} = \mathbb{R}$. In classification problems, the output space is a set of C unordered and mutually exclusive labels known as classes, $\mathcal{Y} = \{1, 2, \ldots, C\}$
- we are given a set of N input-output pairs, known as a training set. $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$. N is called the sample size
- goal is to learn a mapping $\hat{y} = \hat{f}(x; \theta)$ from limited data

Loss function

- a loss function $\ell(y, \hat{y})$ is used to quantify how well we are doing with our predictions
- we can define the average loss of the predictor on the training set as:

$$\mathcal{L}(\theta) \triangleq \frac{1}{N} \sum_{n=1}^{N} \ell(y, \hat{f}(x_n; \theta))$$

- this is called the *empirical risk*

Training

- empirical risk minimization

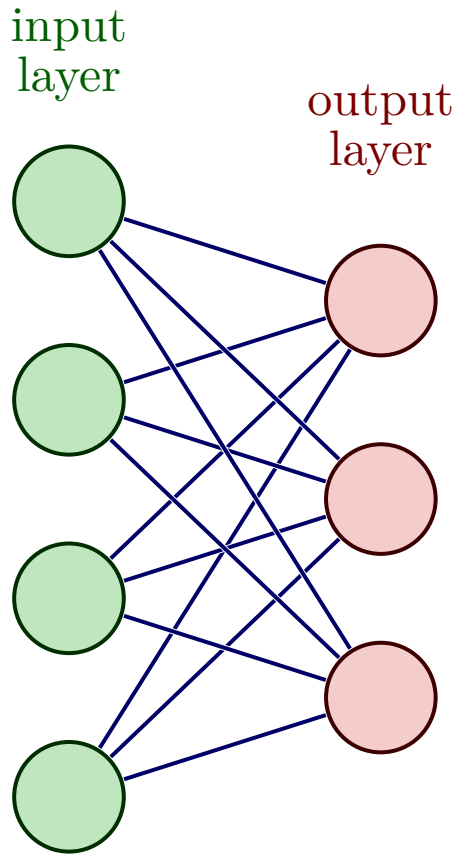
$$\theta^* = \operatorname*{argmin}_{\theta} \mathcal{L}(\theta) = \operatorname*{argmin}_{\theta} \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, \hat{f}(x_n; \theta))$$

- However, our true goal is to minimize the expected loss on future data that *we have not yet seen*.
- That is, we want to generalize, rather than just do well on the training set.

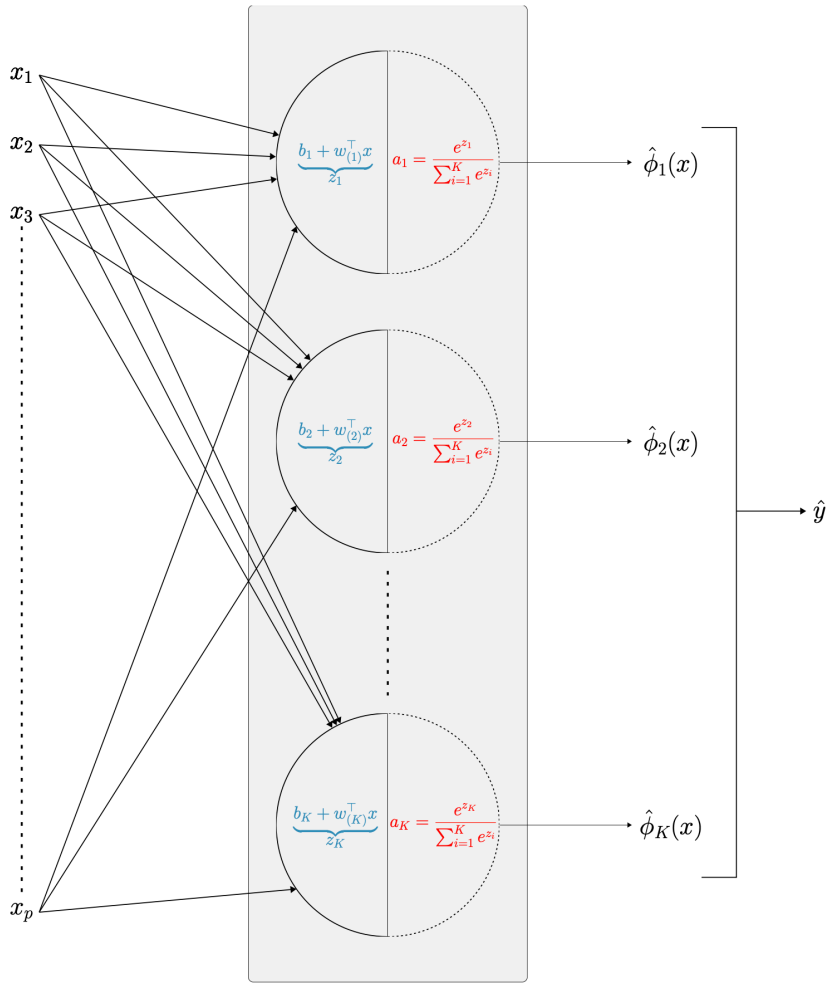
Multiclass logistic regression

- all the ideas of the binary case can be generalized to the case where we have to classify among C classes
- the class label $y \in \{1, \ldots, C\}$
- the classification is a (discrete) probability distribution over the C classes
- the sigmoid function is generalized to the softmax function
- the binary cross-entropy function is generalized to the cross-entropy function

Multiclass logistic regression



Multiclass logistic regression



(non-normalized) predictions, denoted by z here, are called *logits*

Multiclass logistic regression

- the softmax function

$$\mathcal{S}(z) \triangleq \left[\frac{e^{z_1}}{\sum_{j=1}^{C} e^{z_j}}, \cdots, \frac{e^{z_C}}{\sum_{j=1}^{C} e^{z_j}} \right]$$

- where z is a generalization of $w^T x + b$ (from the binary case)

$$z = \Theta x = Wx + b$$

  where W is a $C \times D$ matrix and b is a $C \times 1$ bias vector

- the multiclass logistic loss function is the cross entropy function

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{c=1}^{C} y_{nc} \log \hat{y}_{nc}$$

- its gradient may be written as $\nabla \mathcal{L} = -\frac{1}{N} \sum_{n=1}^{N} (y_{nc} - \hat{y}_{nc}) x_n$ but we will use auto differentiation tools to compute it

Intuition for cross entropy as loss function

- consider classifying an iris sample into one of the three classes
- $\hat{y}$ is a predicted probability distribution
- target label may be interpreted as a probability distribution
  - interpret label $y = 0$ as the distribution $y = [1, 0, 0]$
  - interpret label $y = 1$ as the distribution $y = [0, 1, 0]$
  - interpret label $y = 1$ as the distribution $y = [0, 0, 1]$
- the cross entropy function behaves as we want a loss function to behave:

$$\mathsf{CE}(\hat{y}, y) = -y_0 \log \hat{y}_0 - y_1 \log \hat{y}_1 - y_2 \log \hat{y}_2$$

Example: learning an iris classifier

- let's take a detailed look at a colab notebook

Validation

- many metrics can be used, but a simple one is how many samples are classified correctly
- but the real learning is when we can classify *unseen* examples; otherwise the learner could simply memorize the data!
- therefore we should reserve part of the data for testing
- testing data cannot be used to modify the learned classifier

Why deep learning?

- recall that decision function

$$f(x;\theta) = Wx + b$$

  is linear in the parameters $\theta = (W, b)$

- easy to fit but feature transformation by hand is very limiting
- why not endow the feature extractor with its own parameters

$$f(x;\theta) = W\phi(x, \theta_2) + b$$

  where $\theta = (\theta_1, \theta_2)$ and $\theta_1 = (W, b)$

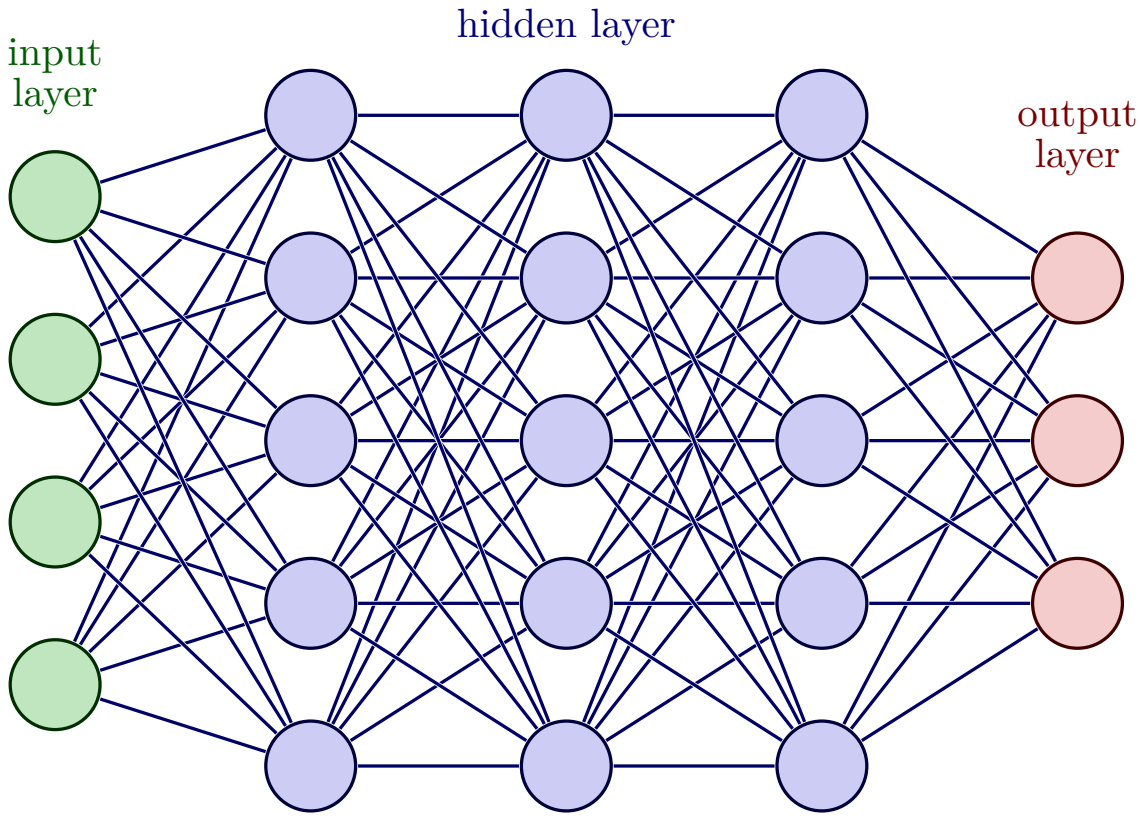
- repeat this idea recursively

$$f(x;\theta) = f_L(f_{L-1}(\cdots(f_1(x))\cdots))$$

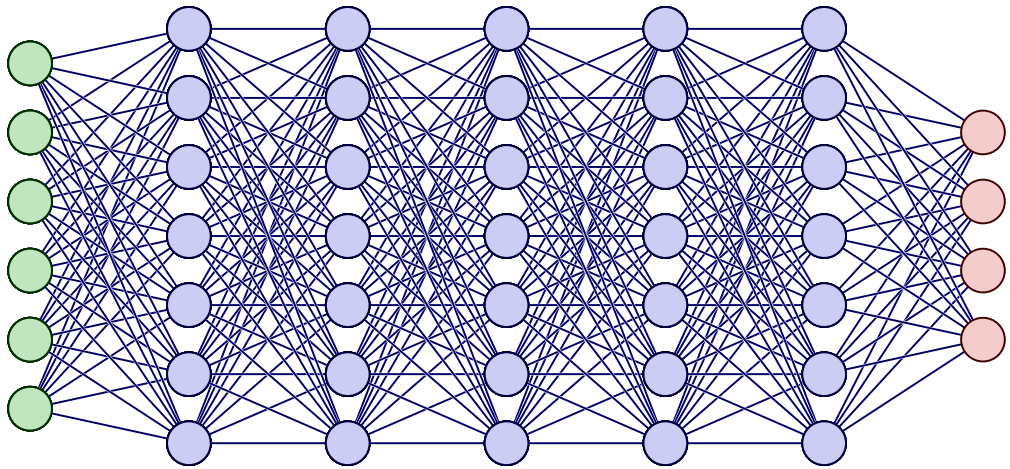
  where $f_l(x) = f(x;\theta_l)$ is the function at layer l

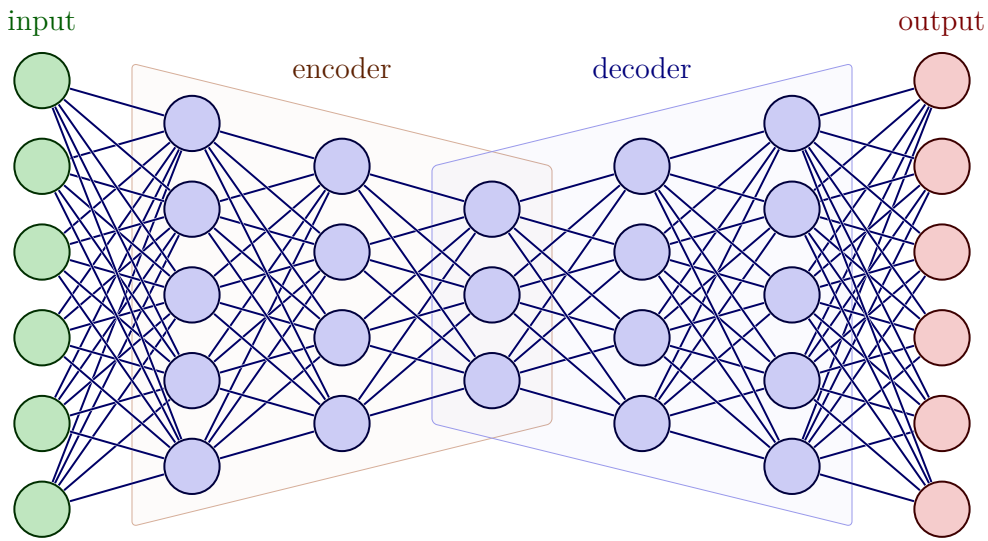
- a key idea behind deep neural networks or DNNs

Network architectures



Network architectures



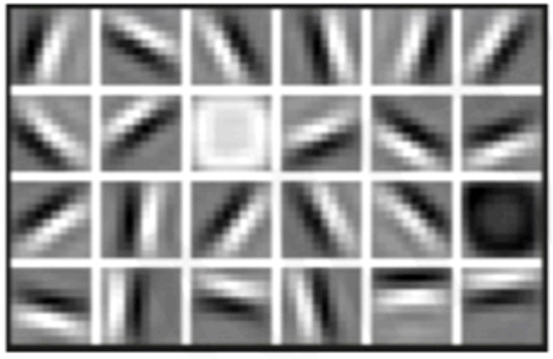


Deep neural networks

- DNNs encompasses a large family of models in which we can compose differentiable functions into any kind of DAG (directed acyclic graph), mapping input to output
- the simplest example is a chain
- this is known as a feedforward neural network, or multilayer perceptrons
- so far we worked with training data that is an $N \times D$ table
- but the input features can have additional structure (images, sequences, or graphs)
- structure be exploited by networks designed to work with these data: CNN, RNN/transformers, GNN

Why deep learning?

Low Level Features



Lines & Edges

Mid Level Features

Eyes & Nose & Ears

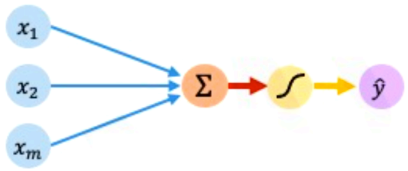
High Level Features

Facial Structure

Core ideas

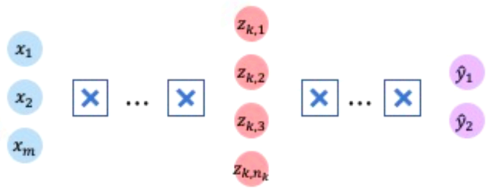
The Perceptron

- Structural building blocks
- Nonlinear activation functions



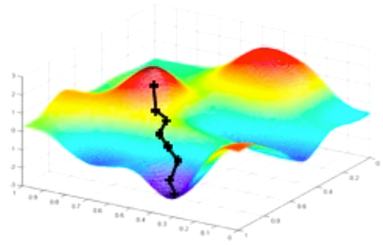
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation

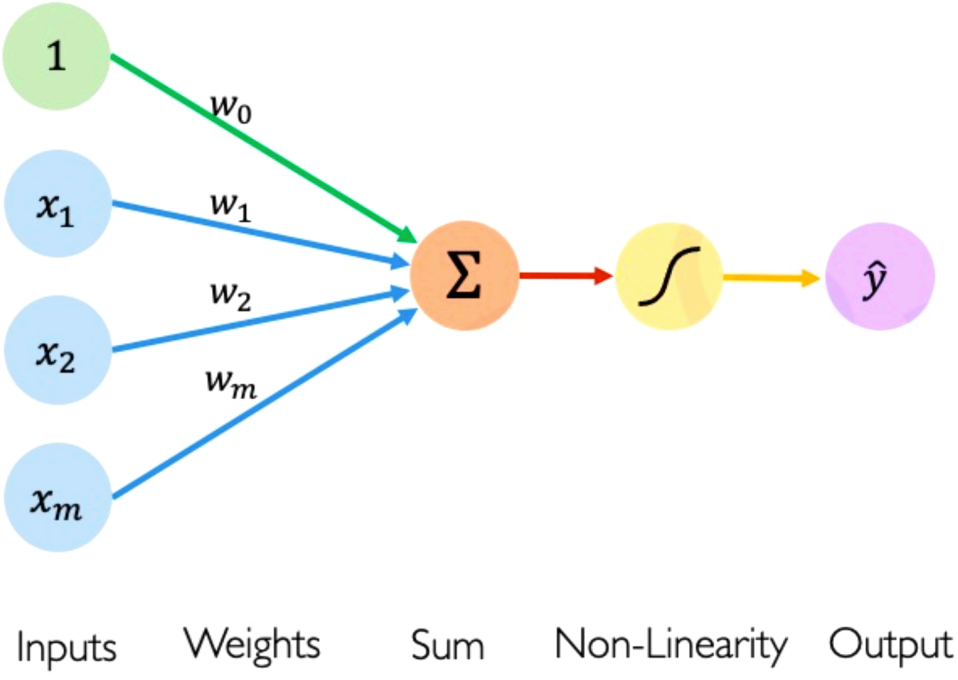


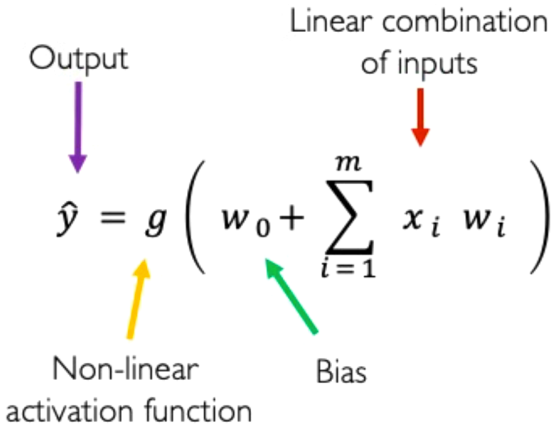
Training in Practice

- Adaptive learning
- Batching
- Regularization



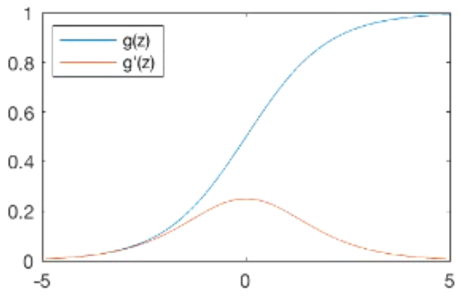
Artificial neuron





Nonlinear activation functions

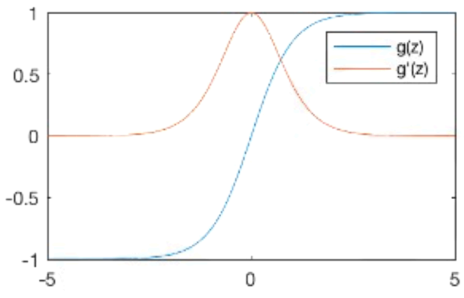
Sigmoid Function



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

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

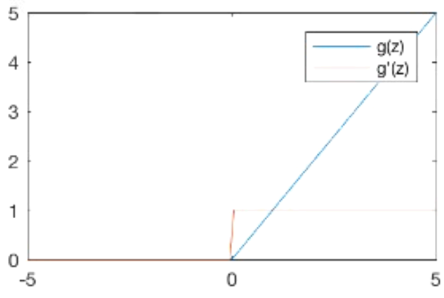
Hyperbolic Tangent



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

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

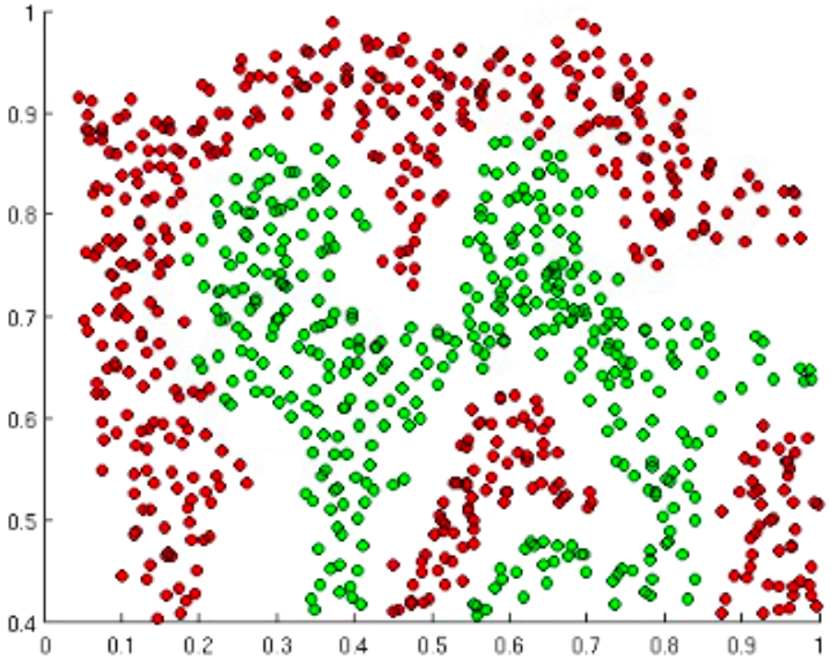
Rectified Linear Unit (ReLU)



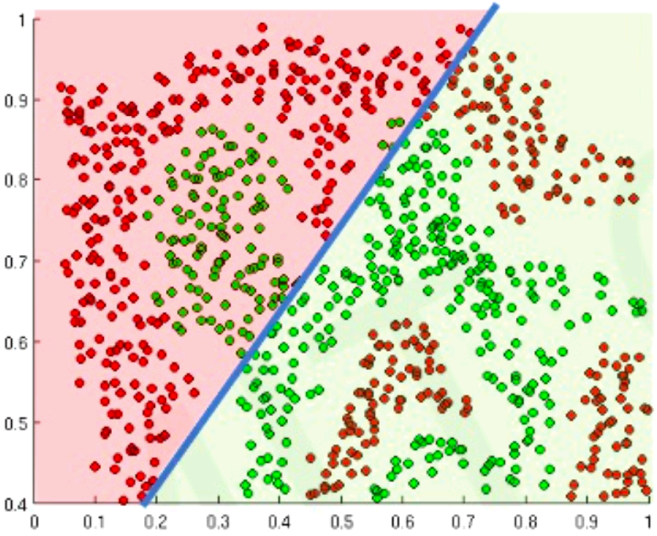
$$g(z) = \max(0, z)$$

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

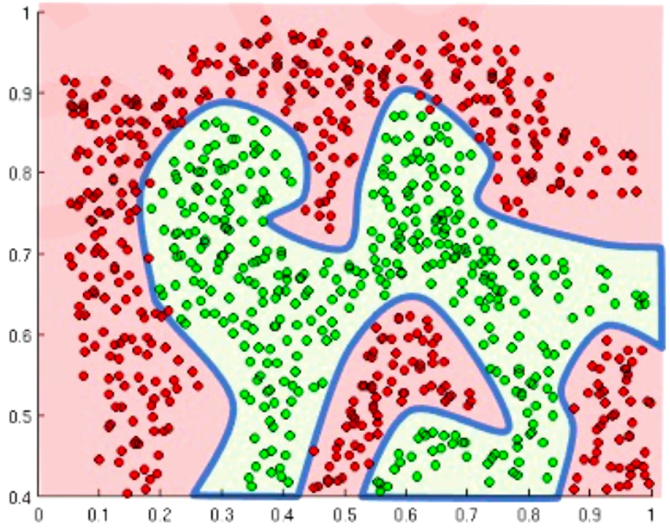
The need for nonlinearities



The need for nonlinearities

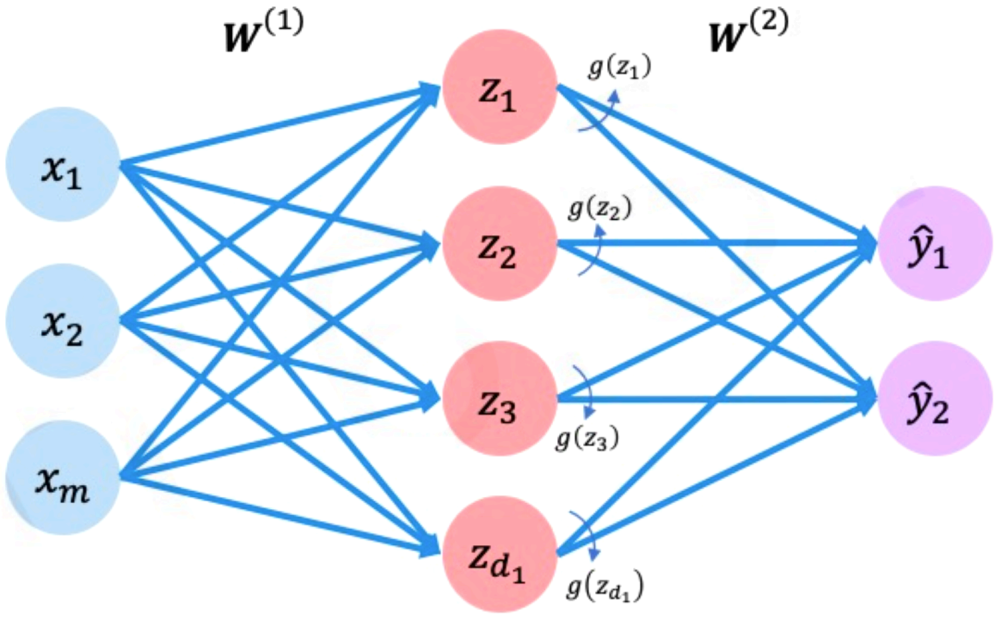


Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

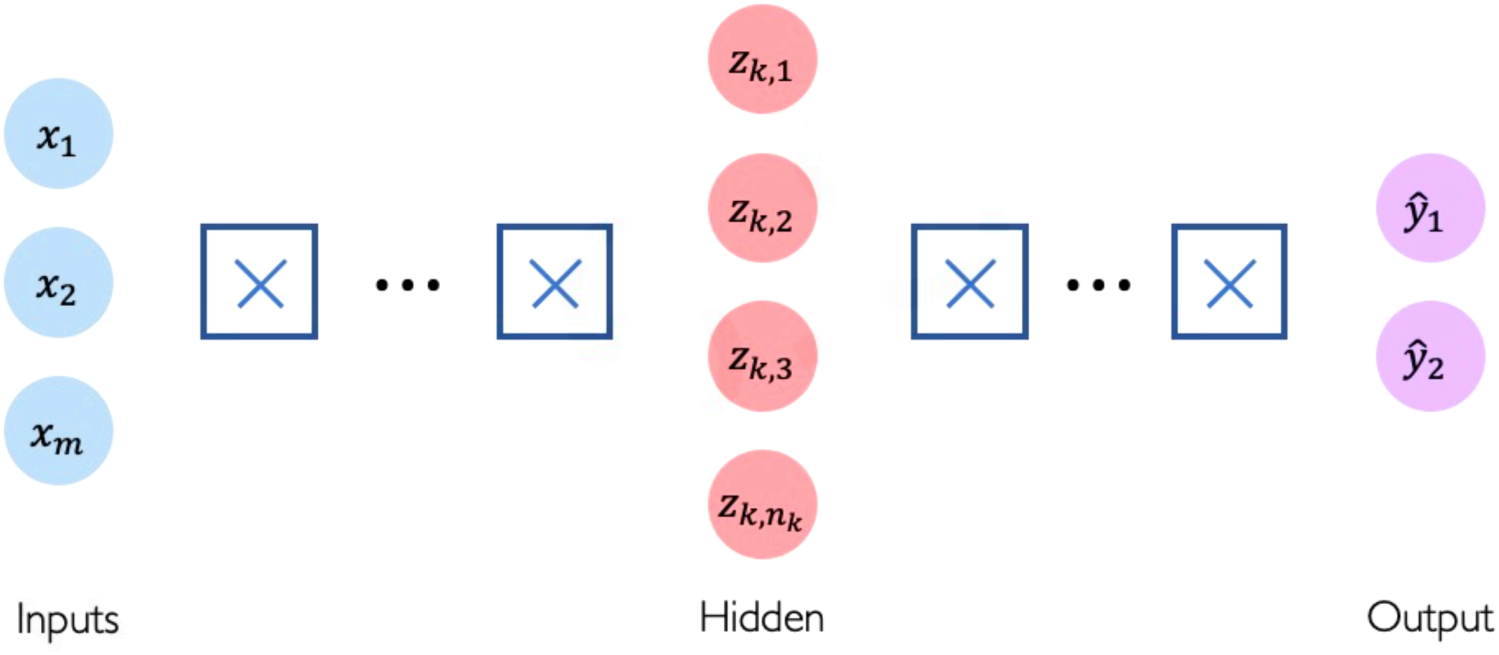
Hidden layers



Inputs Hidden Final Output

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \, w_{j,i}^{(2)} \right)$$

Deep networks



$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

In Pytorch

- multilayer network definition with nonlinear ReLU activation

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(D, 128),
    nn.ReLU(),
    nn.Linear(128, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)
```

- we can ask pytorch to tell us about network details

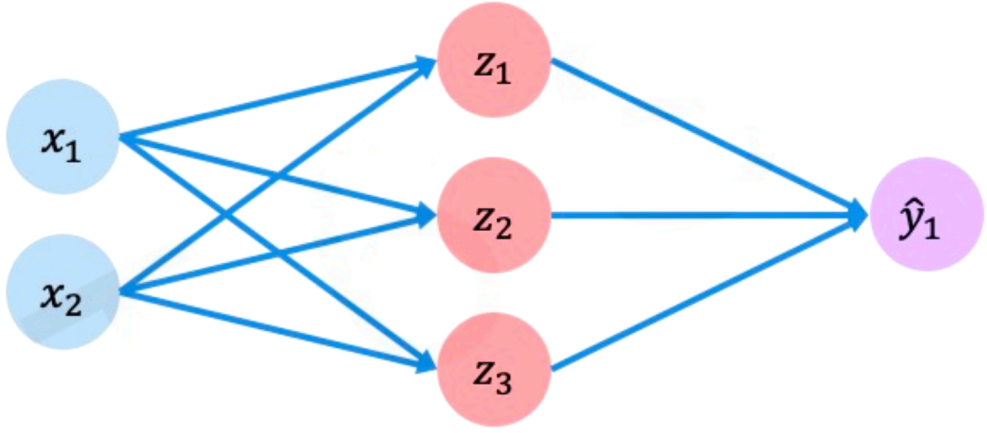
```
print(model)
summary(model, input_size=(batch_size, D))
```

A more structured way to define the model

```
import torch.nn.functional as F

class FNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(28*28, 128)
        self.linear2 = nn.Linear(128, 128)
        self.linear3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        logits = self.linear3(x)
        return logits

model = FNN()
summary(model, (1,28,28))    # displays internal sizes
print(model)                 # prints model statistics
```

Loss functions

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$

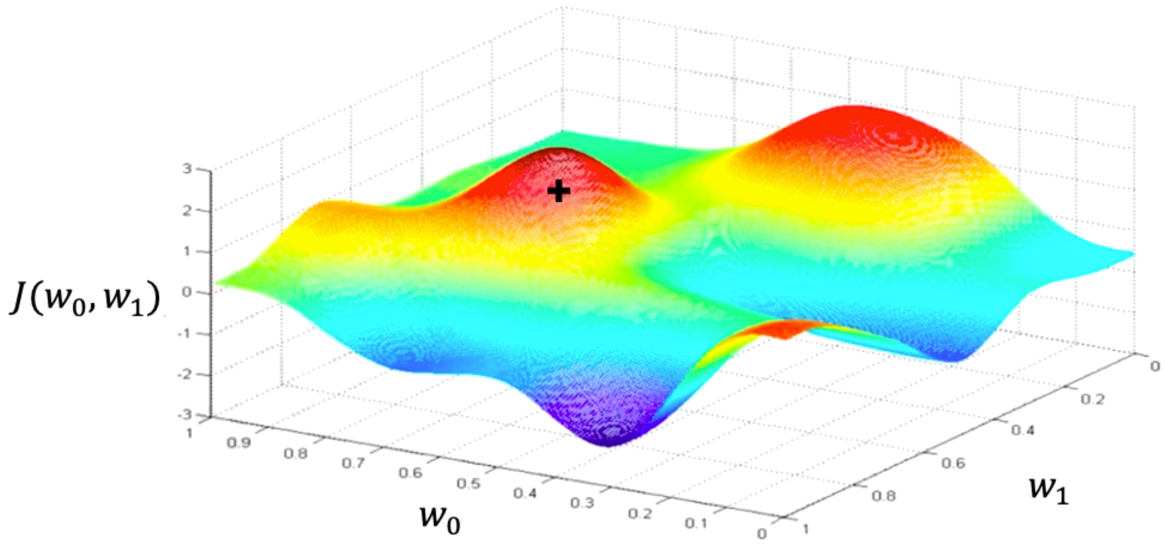


$f(x)$ y

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$\boldsymbol{J}(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

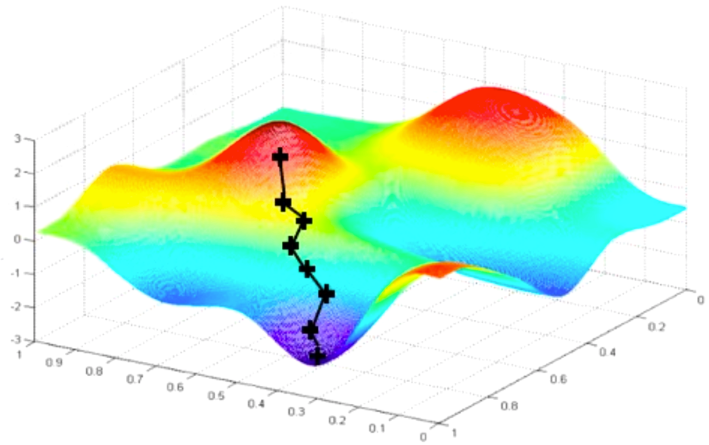
Loss landscape



Gradient descent

Algorithm

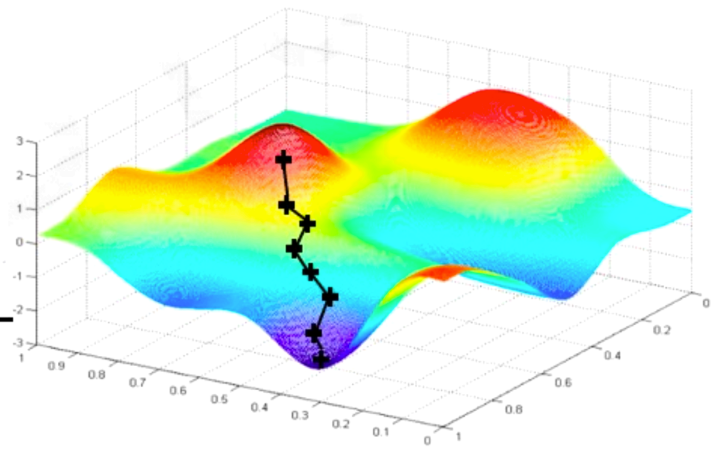
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights



Stochastic gradient descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



MLP for classifying 2d data into 2 classes

- Demo at http://playground.tensorflow.org
- model has the form

$$
\begin{aligned}
p(y|x;\theta) &= \mathrm{Ber}(y|\sigma(a_3)) \\
a_3 &= w_3^T z_2 + b3 \\
z_2 &= \varphi(W_2 z_1 + b2) \\
z_1 &= \varphi(W_1 x + b1)
\end{aligned}
$$

- parameters $\theta = (W_1, b_1, W_2, b_2, w_3, b_3)$

Three exercices to do in the next 90 minutes

- Iris data with logistic regression
  - do the two extensions in notebook
- MNIST data with logistic regression
  - start with the given template
  - MNIST data is already separated into training and testing sets (so you can train on one and test on the other)
- MNIST data with FNN
  - start with the same template, but the model will now have two hidden layers with 128 neurons each and ReLU activation