Intro to Machine Learning and Deep Learning

Christos Kozanitis, FORTH
kozanitis@ics.forth.gr

What is ML?

- Practical definition
  - Label assignment to data
- Broad field
  - Computer Science
  - Probability + Statistics
  - Optimization
  - Linear Algebra

Some examples

- Face recognition
- Link prediction
- Text classification (e.g. spam detection)
- Games (e.g. Backgamon)
- Chat

Terminology

- Observations: Items or entities used for learning or evaluation
  - E.g. emails
- Features: Attributes (usually numeric) used to represent observations
  - E.g. Length, date, presence of keywords
- Labels: Values/categories assigned to an observation
  - E.g. spam/not spam
- Training and Test Data: Observations used to train and evaluate a learning algorithm (e.g. a set of emails + labels).
  - Training data is given to the algorithm from training
  - Test data is withheld at train time.

Different learning approaches

- Supervised: Learning from labeled observations
  - Labels teach algorithm to learn mapping from training dataset
- Unsupervised: Learning from unlabeled observations
  - Learning algorithm must find underlying structure from features alone
  - Can be a goal from itself (discover hidden patterns, explore data)
  - Part of preprocessing (e.g. feature extraction) of a supervised algorithm

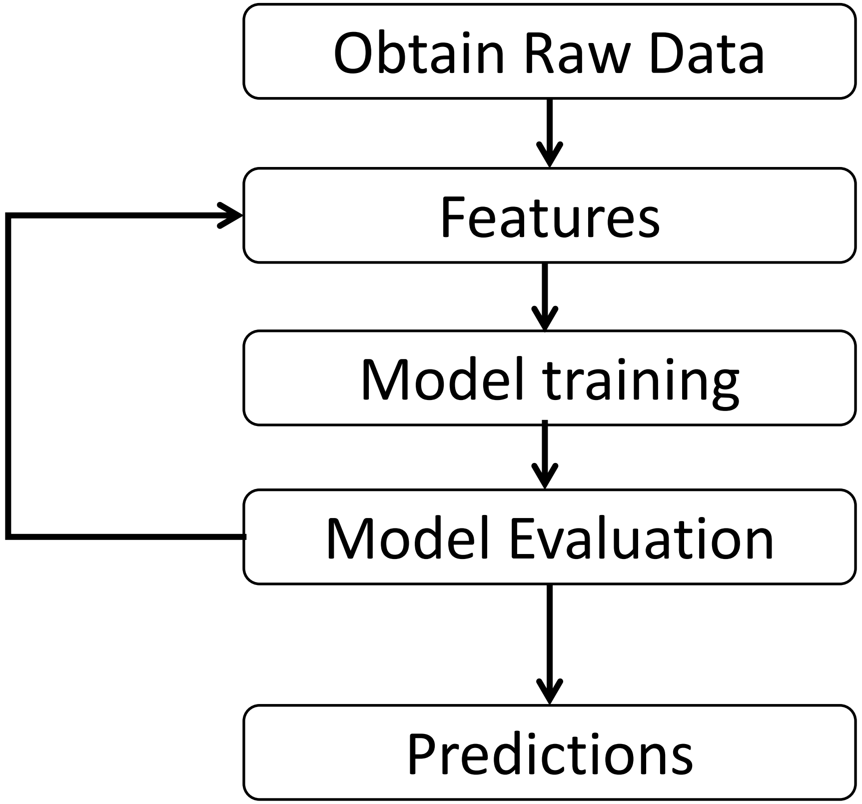
Examples of supervised learning

- Regression: Predict a real value for each item (e.g. stock prices)
  - Labels are continuous
- Classification: Assign a category to each item (e.g. spam/not spam)
  - Categories are discrete

Examples of unsupervised learning

- Clustering: Partition observations into homogeneous regions
  - E.g. identify similar images
- Dimensionality reduction: Transform an initial set of features into a more concise representation
  - E.g. visualization

A typical supervised ML pipeline

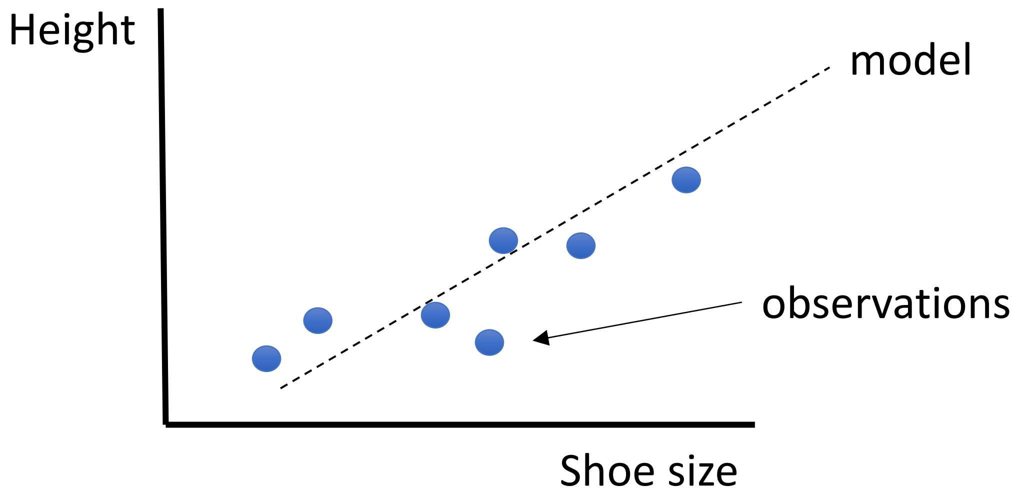


Data without label

Crucial step

We want numeric features

A toy Machine Learning problem: Predict people heights from their shoe sizes



- X: features (shoe size)
- Y: Labels (height)

- Model hypothesis: $y \sim \hat{y} = w_0 + w_1 x$

- Learning Goal: Find proper w0, w1

How to learn w0, w1

- Need a loss function to minimize and a learning algorithm

$$\min_{w_0, w_1} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 = \min_{w_0, w_1} \sum_{i=1}^{N} (w_0 + w_1 x_i - y_i)^2$$

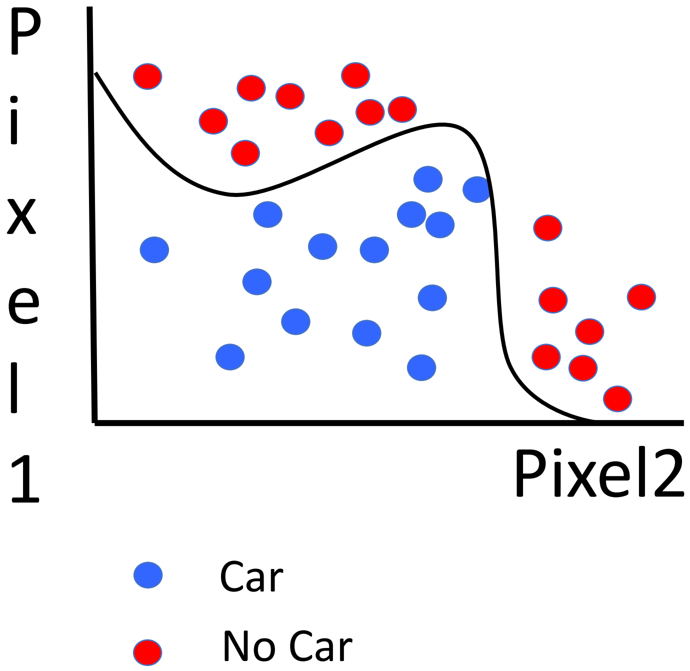
- Learning algorithms:
  - Algebraic solutions (accurate, but slow)
  - Gradient descent (linear with data dimensions)

Moving to more complex problems: How about non linear relationships?

- **Problem**: Recognize that an image contains a car
- Input: A set of images (arrays of pixels)
- Output: 0/1

How would we classify images of cars?

Pixel 2

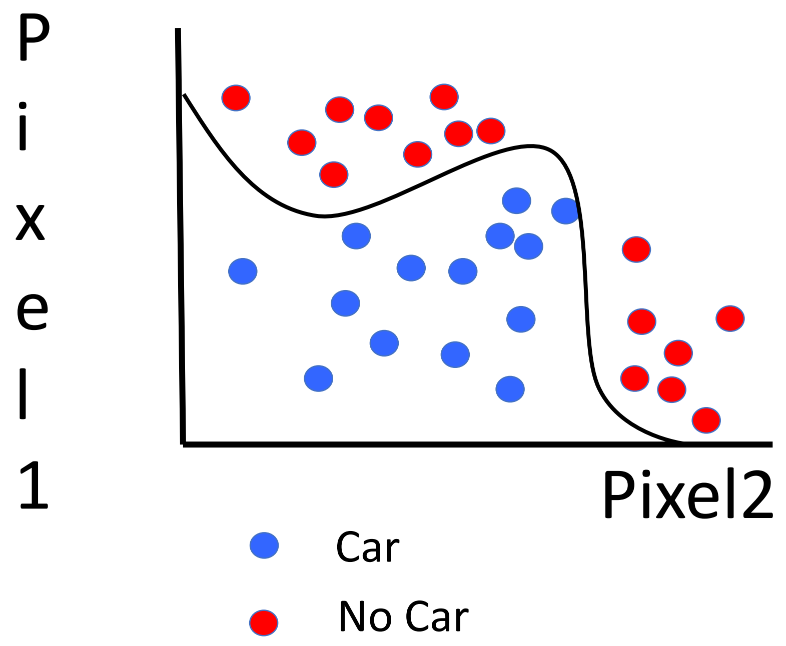


Choose two pixels and plot them for each image of the dataset

Possible way of introducing non linearity to linear models: quadratic transformations

- Get all pairwise multiplications
- Remove redundant pairs
- Linear models now connect labels with the quadratic transformations of features => non linear relation with features themselves.

Revisiting car classification



Strategy: Choose two pixels and plot them for each image of the dataset

- What if we use ALL pixels?
- 50x50 => d=2500 pixels (grayscale)
  - d=7500 pixels RGB
- Quadratic features: ~3M

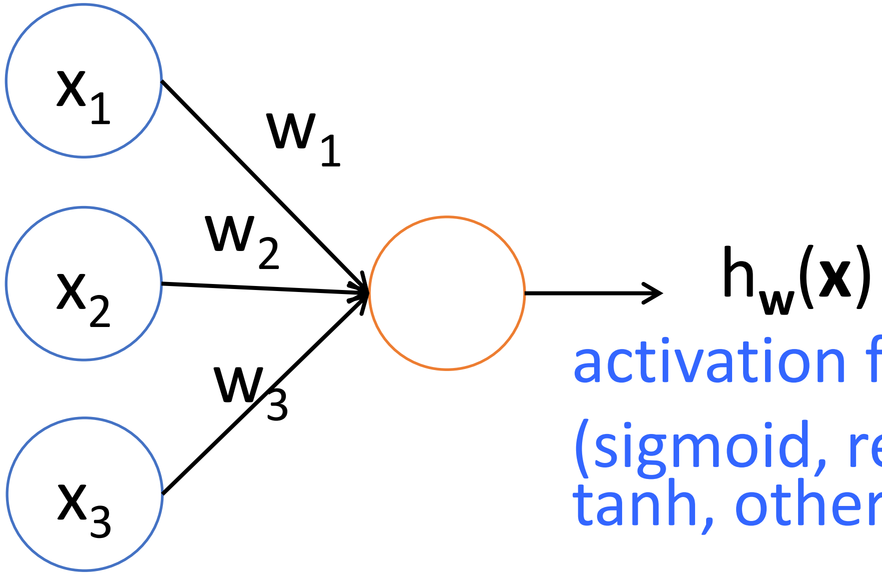
Summary

- Linear models + complex non linear hypothesis:
  - Need quadratic or cubic features
  - Feature numbers explode
  - Models => trainable parameters explode + overfitting danger
  - Training => takes a lot of time
- Alternative solution: Neural Networks
  - Pass non linearity from features to the model

Neuron unit: A function of the dot product of the input with weight vectors

- If x_0 present: Always 1
  - Bias unit

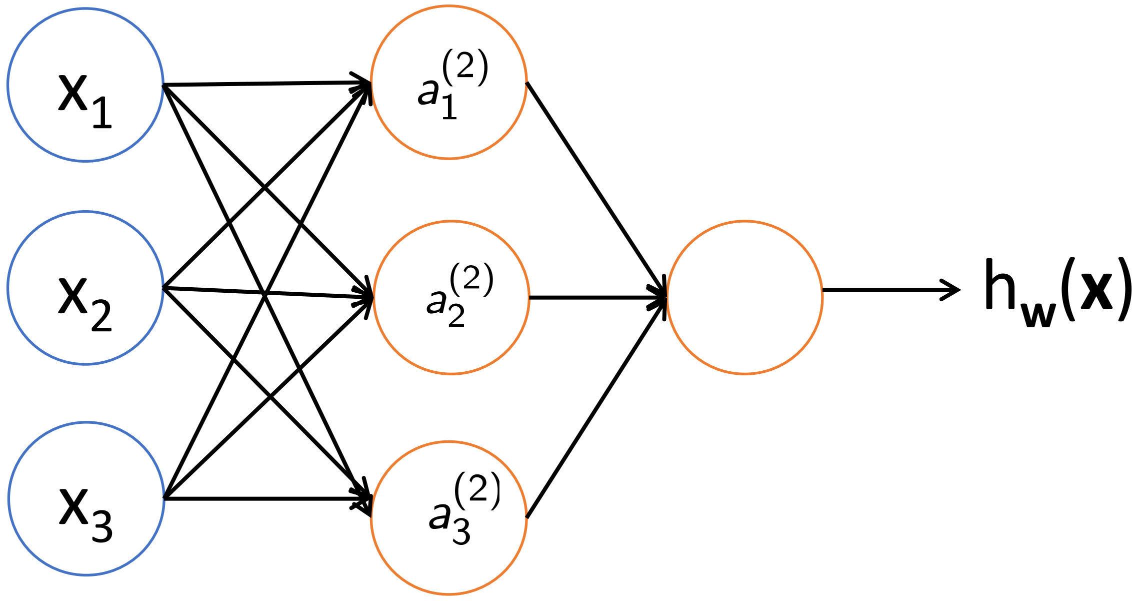
$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}$$



activation function

(sigmoid, relu, softmax, tanh, others)

Neural Network: A grouping of neuron units



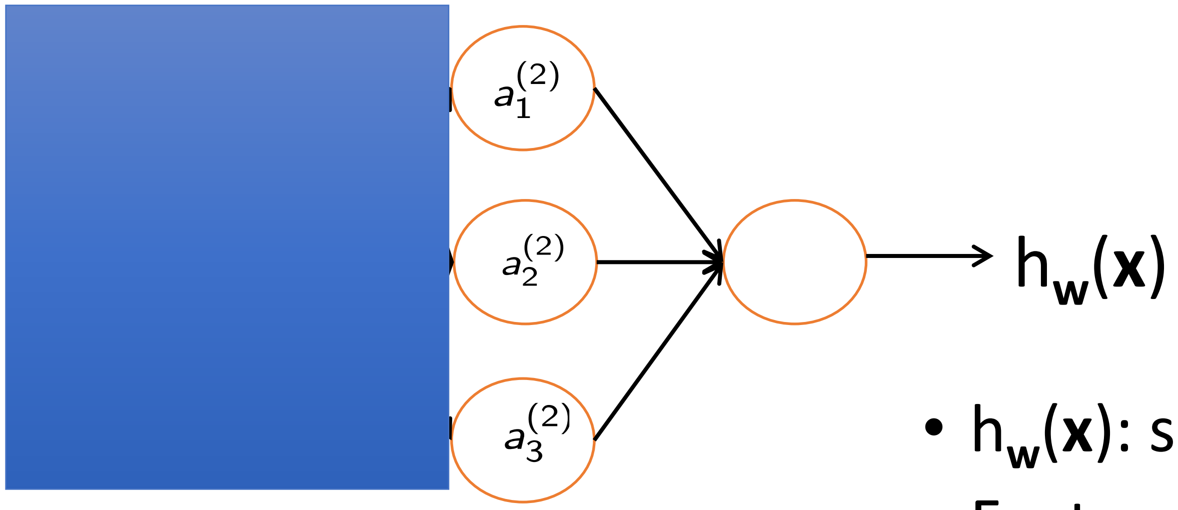
Layer 1
(Input x)

Layer 2
(hidden)

Layer 3
(Output y)

What is the big deal about NNs?

- NNs learn their own features

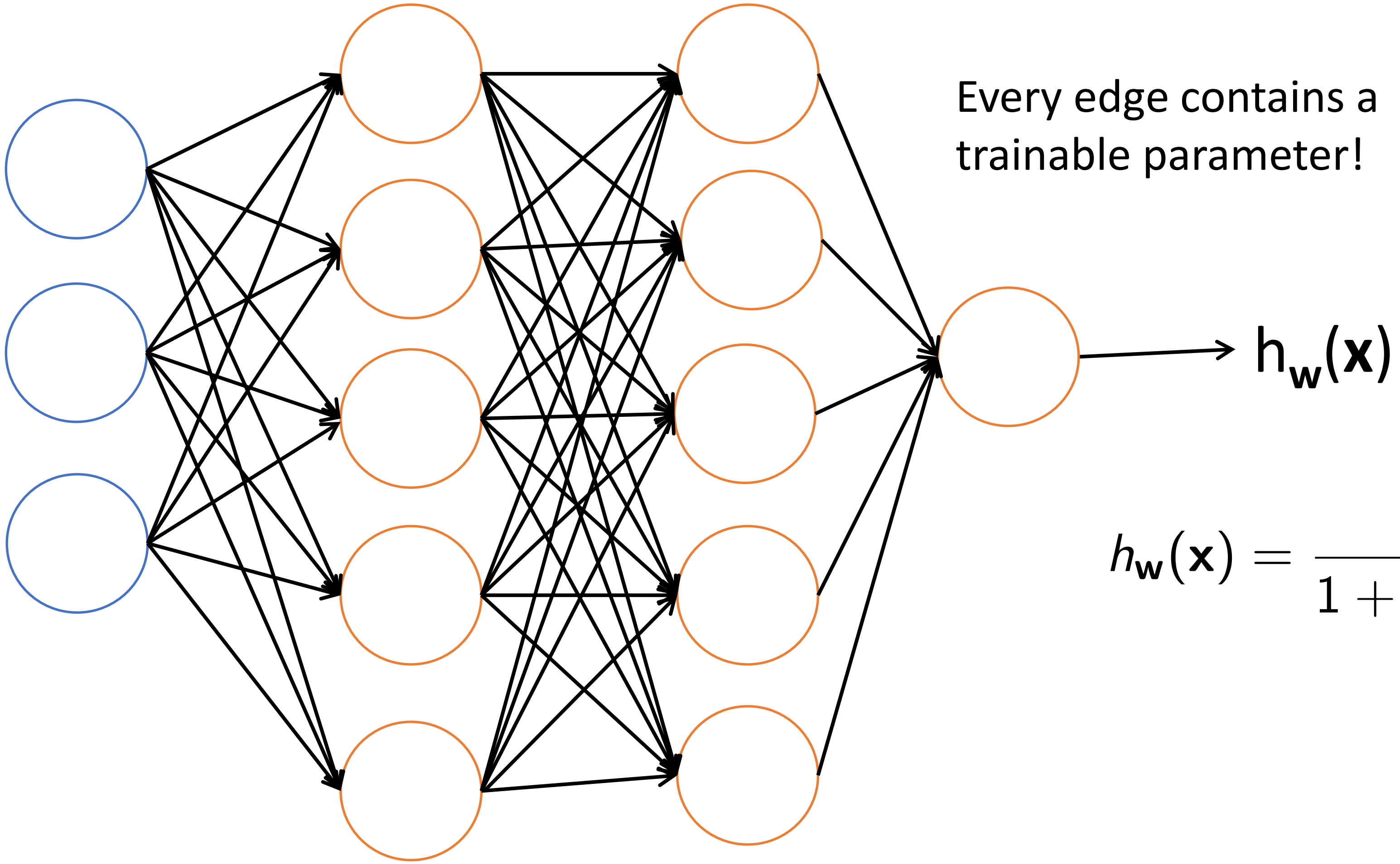


- $h_{\mathbf{w}}(\mathbf{x})$: simple logistic regression
- Features: **a** not **x**
- The network learns **a** in the previous layer

Formalizing our problem

- Training set: Labelled images (car, not a car)
- $(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})$
- $\mathbf{x}$: A vector of pixels
- y: 0/1

Model setup: Predict the probability that an image is a car



Every edge contains a trainable parameter!

$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}}$$

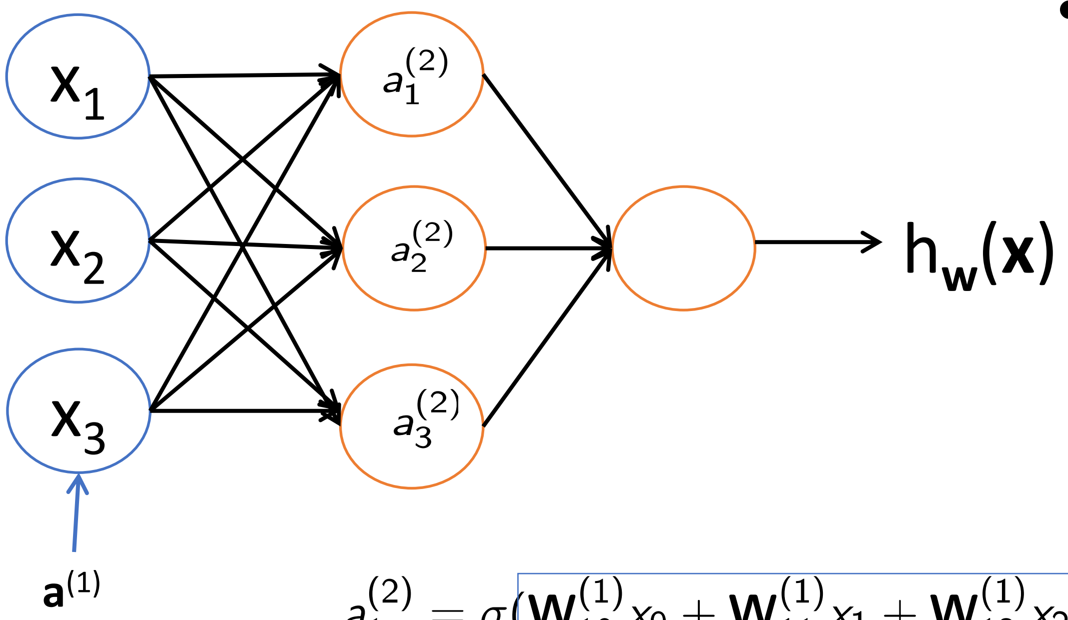
In order to train

- Cost function that learnable parameters will minimize:
  - Logistic loss
- Algorithm
  - Gradient descent family

How training proceeds

- Forward propagation to compute output as a function of input
- Loss evaluation
- Backward propagation to calculate gradients
- Weight update

Forward Propagation: Vectorized implementation



- How do we compute the output?

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} z_1^2 \\ z_2^2 \\ z_3^2 \end{bmatrix}$$

$\mathbf{z}^{(2)} = \mathbf{W}^{(1)}\mathbf{x}$

$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)})$

Add $\mathbf{a}_0^{(2)} = 1$

$\mathbf{z}^{(3)} = \mathbf{W}^{(2)}\ \mathbf{a}^{(2)}$

$\mathbf{a}^{(3)} = \sigma(\mathbf{z}^{(3)})$

Fwd propagation

$$a_1^{(2)} = \sigma(\mathbf{W}_{10}^{(1)} x_0 + \mathbf{W}_{11}^{(1)} x_1 + \mathbf{W}_{12}^{(1)} x_2 + \mathbf{W}_{13}^{(1)} x_3) \leftarrow \mathbf{z_1}^{(2)}$$

$$a_2^{(2)} = \sigma(\mathbf{W}_{20}^{(1)} x_0 + \mathbf{W}_{21}^{(1)} x_1 + \mathbf{W}_{22}^{(1)} x_2 + \mathbf{W}_{23}^{(1)} x_3) \leftarrow \mathbf{z_2}^{(2)}$$

$$a_3^{(2)} = \sigma(\mathbf{W}_{30}^{(1)} x_0 + \mathbf{W}_{31}^{(1)} x_1 + \mathbf{W}_{32}^{(1)} x_2 + \mathbf{W}_{33}^{(1)} x_3) \leftarrow \mathbf{z_3}^{(2)}$$

$$h_{\mathbf{W}}(x) = a_1^{(3)} = \sigma(\mathbf{W}_{10}^{(2)} a_0^{(2)} + \mathbf{W}_{11}^{(2)} a_1^{(2)} + \mathbf{W}_{12}^{(2)} a_2^{(2)} + \mathbf{W}_{13}^{(2)} a_3^{(2)})$$

Backward propagation

- Starting from the final layer:
  - compute the cost by comparing output with true label
- Moving to every layer from right to left:
  - Compute the partial derivative of the cost function J(W) for every weight at every layer

$$\frac{\partial}{\partial W_{i,j}^{(\ell)}} J(\mathbf{W})$$

- Involves multiple vector-vector products

Where to run? How to code?

The entire process is too slow

- A lot of matrix multiplications
- Scale out to make computations fast? Not really working.
    - If every Android user wants to translate 3 min audio every day => Google needs to double its datacenter.
- We cannot avoid Scale Up computation (GPUs, TPU etc)
- How do we program this? Do we need to learn CUDA?