Ανάπτυξη Λογισμικού για Δυσεπίλυτα Αλγοριθμικά Προβλήματα

Ενότητα 2: Nearest Neighbors and Clustering

Γιάννης Εμίρης[1,2] & Γιάννης Χαμόδρακας[1]

[1] Τμήμα Πληροφορικής και Τηλεπικοινωνιών, ΕΚΠΑ

[2] Ερευνητικό Κέντρο «Αθηνά»

Νοε.–Δεκ. 2025

Contents

Introduction to AI

Machine Learning

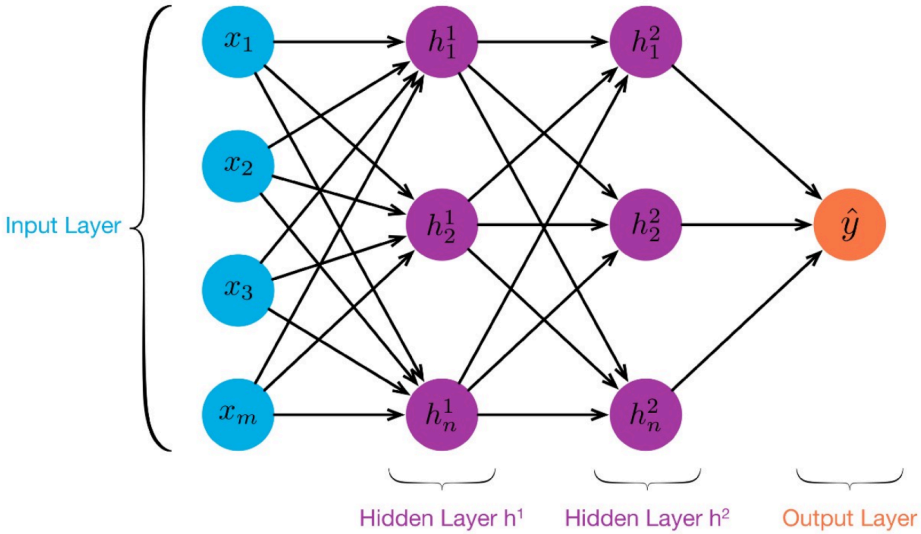
Machine learning (ML) uses statistical techniques to give computer systems the ability to ``learn'' (i.e. progressively improve performance on a specific task) from data, without being explicitly programmed.

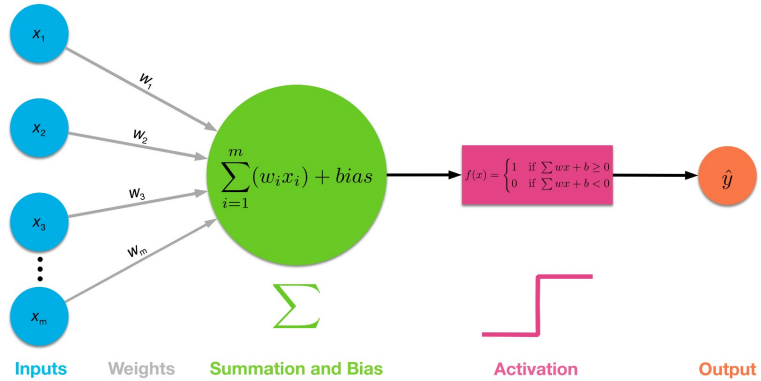
Neural Networks

Neural Networks (and Deep Learning) is a class of ML algorithms that use a cascade of layers of nonlinear processing units, mimicking the human neurons. Each successive layer uses the output from the previous layer as input.

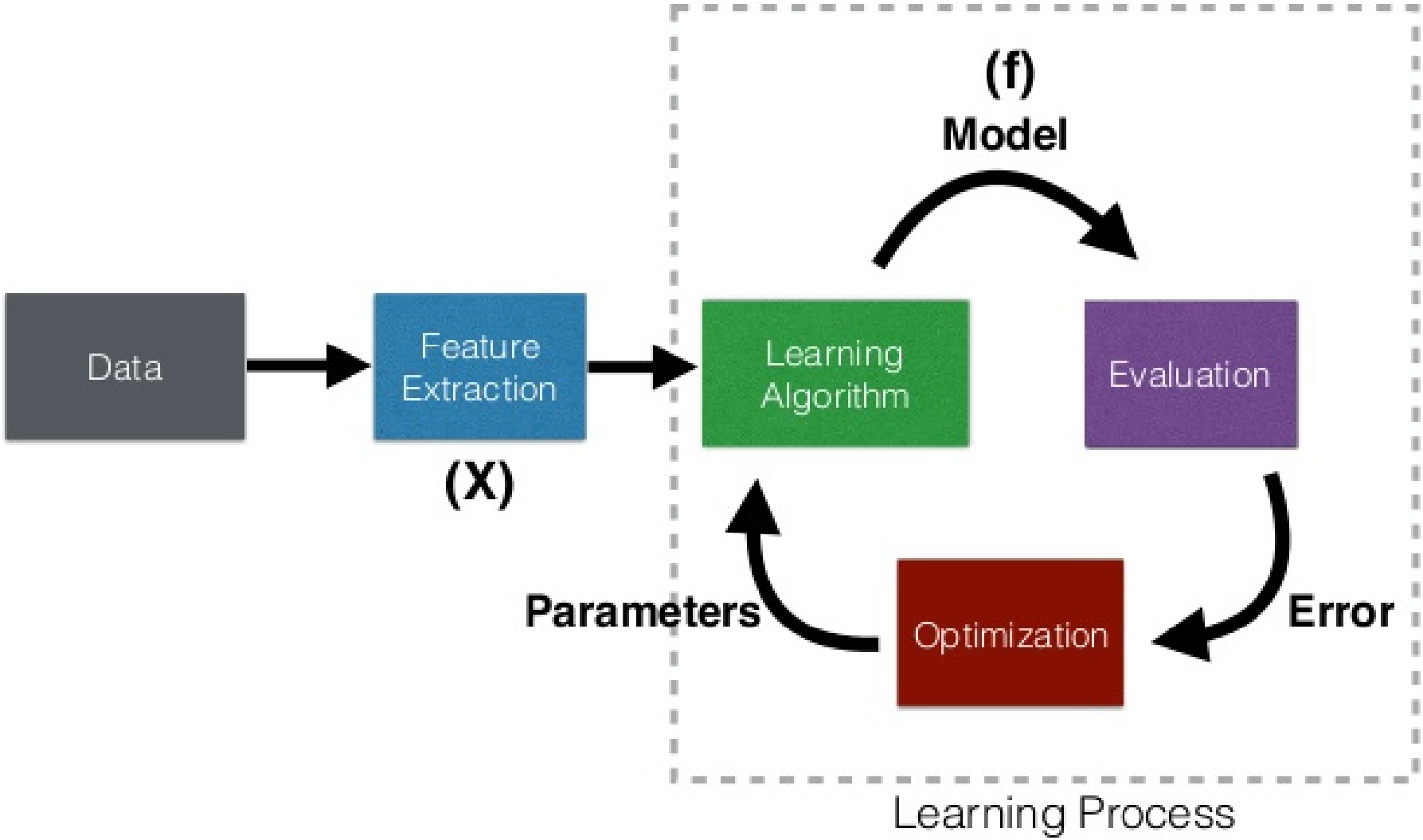
Neural Networks

- NN is a cascade of **hidden layers**
- Each layer is a **data transformation** function
  - weights/parameters determine the transformation
  - transformations are differentiable for improving output





Learning process

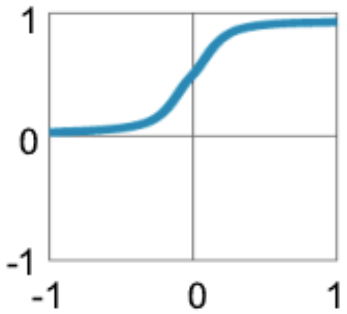


Several epochs (each improving param's) split into batches.

Activation functions

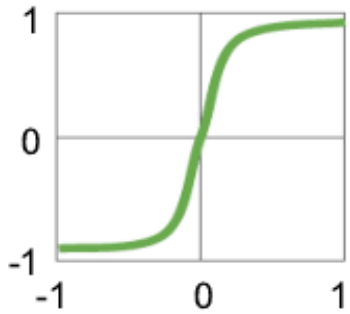
Traditional Non-Linear Activation Functions

Sigmoid



$y=1/(1+e^{-x})$

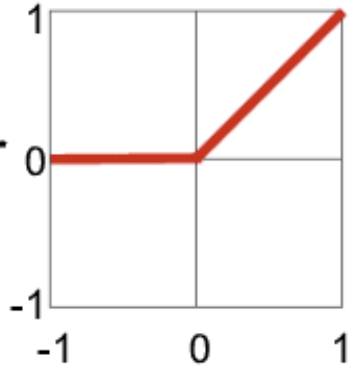
Hyperbolic Tangent



$y=(e^{x}-e^{-x})/(e^{x}+e^{-x})$

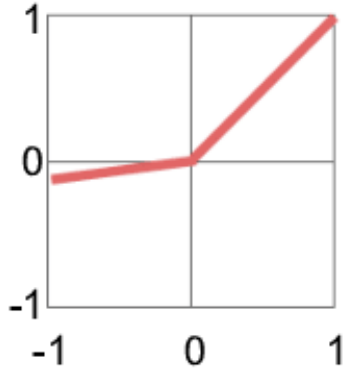
Modern Non-Linear Activation Functions

Rectified Linear Unit (ReLU)



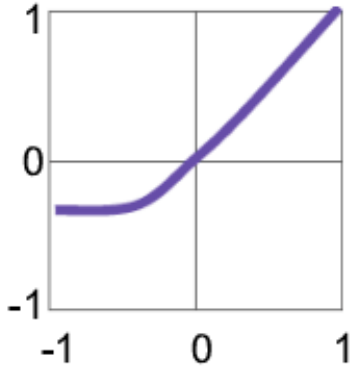
$y=\max(0,x)$

Leaky ReLU



$y=\max(\alpha x,x)$

Exponential LU



$y=\begin{cases} x, & x\geq 0 \\ \alpha(e^{x}-1), & x<0 \end{cases}$

α = small const. (e.g. 0.1)

Loss (error) functions

$y =$ true output label/value, $\widehat{y} =$ predicted class/value,
$n =$ # training instances per batch, $M =$ #classes.

- Regression problems

Mean Squared Error (MSE): $$\frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y_i})^2$$

Mean Absolute Error (MAE): $$\frac{1}{n}\sum_{i=1}^{n}|y_i - \widehat{y_i}|$$

- Classification problems

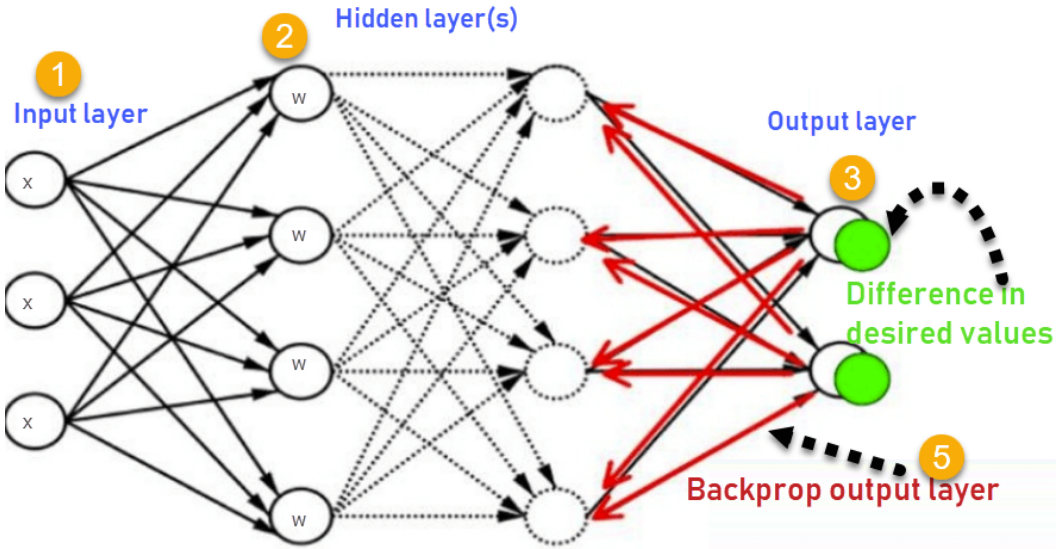
binary cross-entropy: $$-\frac{1}{n}\sum_{i=1}^{n}(y_i \log \widehat{y_i} + (1-y_i)\log(1-\widehat{y_i}))$$

(M classes, one label) Cross-Entropy: $$-\frac{1}{n}\sum_{i=1}^{n}\sum_{j}^{M} y_{ij} \log \widehat{y_{ij}}$$

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

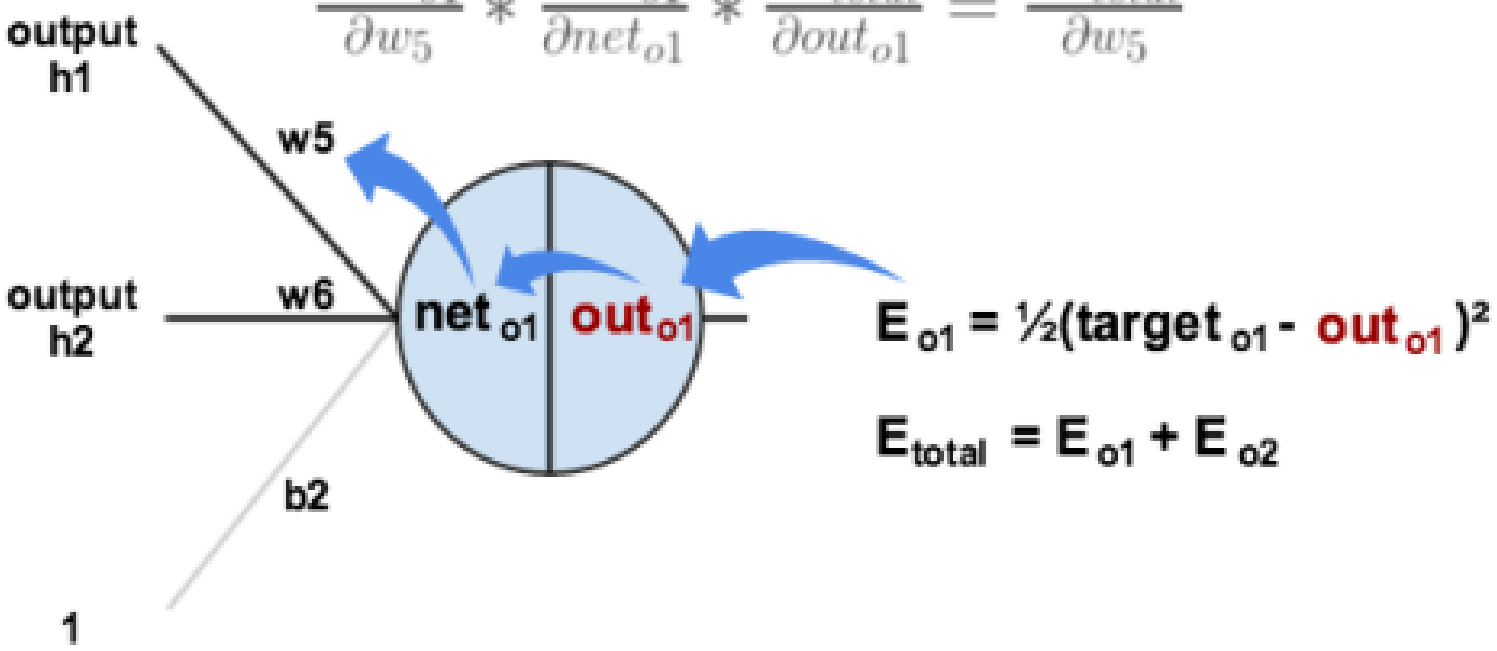
Backward propagation

- Forward propagation: compute activations (black arrows)
- Backprop: compute derivatives optimizing parameters/weights (red)
- Gradient descent exploits chain rule $\frac{dL}{dw} = \frac{dL}{du}\frac{du}{dw}$ wrt **weights**
- Learning rate (e.g. 10^{-5}): step to change weights



Backpropagation

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



Correct earlier weights e.g. w_1 by similarly computing $\partial E_{total}/\partial w_1$, assuming all other weights and inputs fixed.

All such derivatives computed offline and instantiated.

Optimizers

- **SGD (Stochastic gradient descent)**
  - Incremental gradient descent, an iterative method for optimizing a differentiable objective function
- **Adam (Adaptive Moment Estimation)**
  - First-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments
  - Works well in practice, comparable to other stochastic optimization methods
- Adamax
  - Variant of Adam based on the infinity norm
- RMSProp (Root Mean Square Propagation)
  - Learning rate is adapted for each of the parameters
- Adadelta
- ...

Overfitting

NN as universal approximators

Universal Approximation Theorem

Let ξ be a non-constant, bounded, monotonically-increasing continuous ``activation" function, target $f : [0,1]^d \to \mathbb{R}$ continuous, and $\epsilon > 0$. Then, $\exists k$, parameters $a, b \in \mathbb{R}^k, W \in \mathbb{R}^{k \times d}$ s.t.

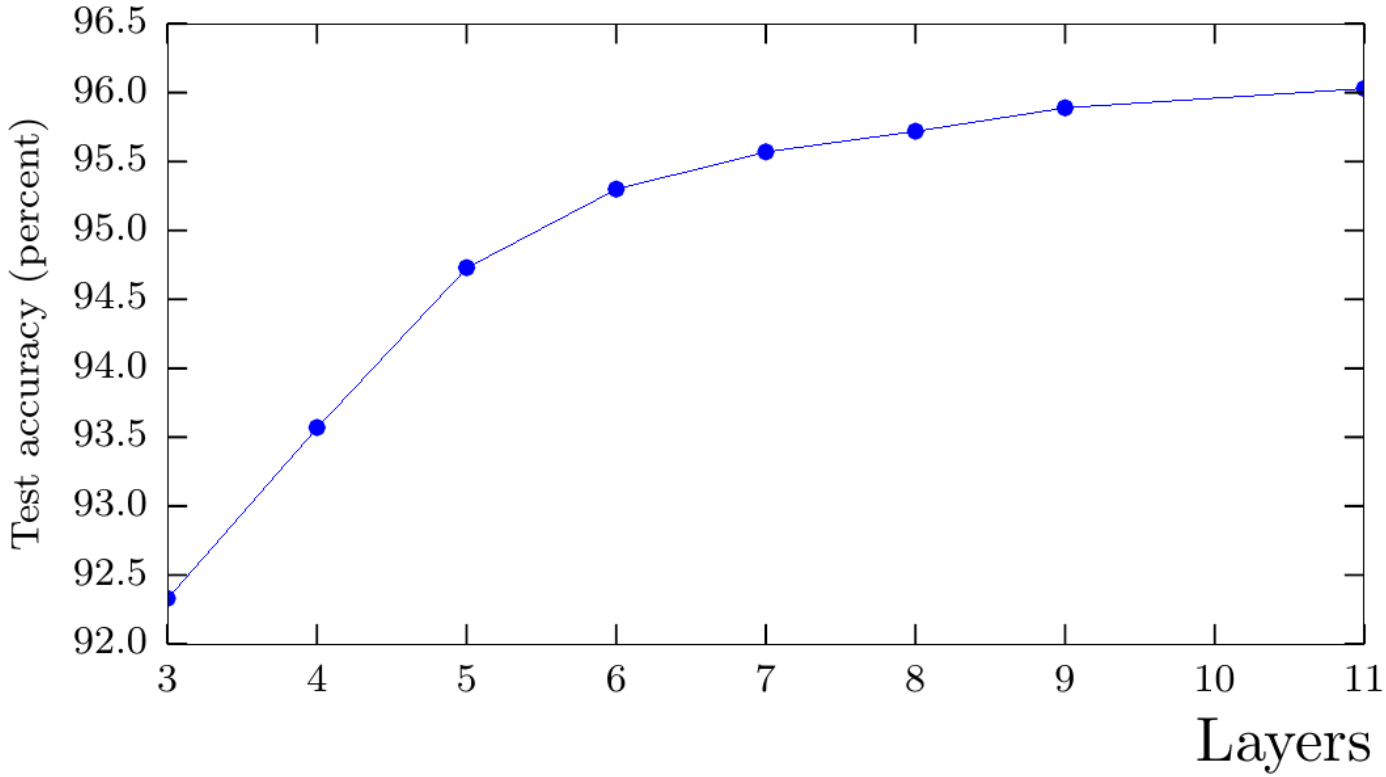
$$\left| \sum_{i=1}^{k} a_i \xi(w_i^T x + b_i) - f(x) \right| < \epsilon.$$

- Any f approximated arbitrarily well by NN with one hidden layer
- How many neurons? How to find the parameters?
- Does it generalize well? Does it overfit?
- Shallow nets work poorly for high-dimensional data like images

Cybenko 1989; Hornik 1991

Better Generalization with Greater Depth

- Empirical results show that **deeper** networks **generalize** better
- Test accuracy consistently increases when increasing depth



Transcribe multidigit numbers from photographs of addresses [Goodfellow'14]

Development Frameworks

In this project we recommend:

- Using **PyTorch** for neural network implementation,
- training on **Google Colab**, which provides free GPU resources.

Contents

Learning Space Partitions for NNS

Given:

- Dataset $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^d$
- Query $q \in \mathbb{R}^d$
- Goal: retrieve k nearest neighbors of q efficiently

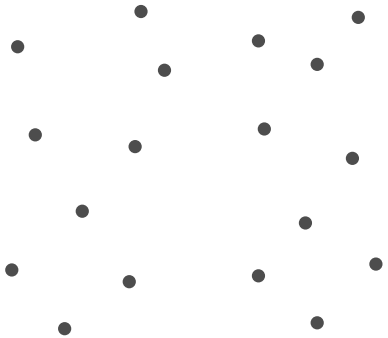
Task: Learn a space partition

$$R : \mathbb{R}^d \to \{1, \ldots, m\}$$

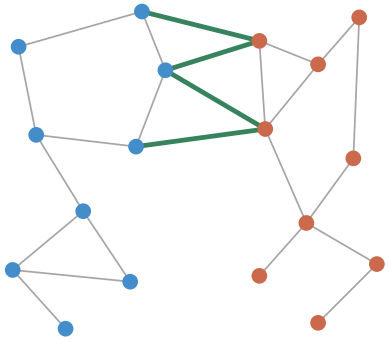
which is [Dong,Indyk et al.'19]:

- **Balanced:** each part/cluster has $\approx n/m$ points
- **Locality:** most true neighbors of q fall in the same part/bin
- **Efficient:** Querying works with a lightweight classifier

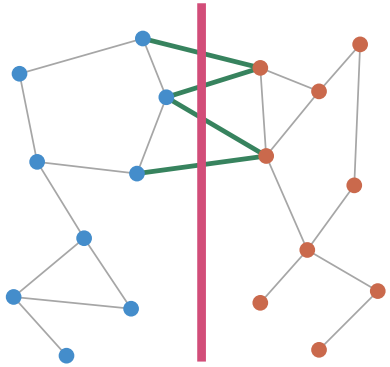
Balanced Graph Partitioning + Classification



(a) Dataset



(b) k-NN graph with balanced partition



(c) Learned partition

Neural LSH Construction

Step 1: Build the k-NN graph $G = (P, E)$ of the dataset.

Step 2: Compute a **balanced partition** of G into m parts: Compute labels $\pi(p) \in \{1, \ldots, m\}$, $\forall p \in P$.

Step 3: Train neural classifier $M : \mathbb{R}^d \to \{1, \ldots, m\}$ on $(p, \pi(p))_{p \in P}$.

Query: For $q \in \mathbb{R}^d$,

$$\text{Candidates}(q) = \{p \in P : M(p) = M(q)\}.$$

Extend to using best b parts ranked by M.

Constructing the k-NN Graph

Given: dataset $P = \{p_i\}_{i=1}^{n} \subset \mathbb{R}^d$ represented by $[n] = \{1, \ldots, n\}$.

Definition: k-nearest neighbor graph Captures Local Neighborhood structure:

$$G = (P, E), \quad E = \{(i, j) : j \in \mathsf{NN}_k(i)\},$$

where $\mathsf{NN}_k(i)$ are the k closest points to p_i under Euclidean distance.

Each vertex corresponds to datapoint p_i, and edges connect locally similar points, encoding the dataset's neighborhood structure.

$\mathsf{NN}_k(\cdot)$ not a symmetric relation.

Preparing the input Graph

Balanced partitioning algorithm "KaHIP" takes weighted undirected input graph.

Symmetrization:

$$\{i, j\} \in E \;\Leftrightarrow\; j \in \mathrm{NN}_k(i) \text{ or } i \in \mathrm{NN}_k(j).$$

Edge Weights:

$$w_{ij} = \begin{cases} 2, & \text{if } i \leftrightarrow j \text{ (mutual neighbors)}, \\ 1, & \text{if one-sided neighbor}, \\ 0, & \text{otherwise (no edge)}. \end{cases}$$

Similarity graph $G = (P, E, w)$ given to KaHIP.

KaHIP Algorithm: Balanced Graph Partitioning

Goal: Split the k-NN graph $G = (P, E, w)$ into m balanced (almost equal size) parts, while minimizing the weights of cut edges.

Objective:

$$\min_{\pi: P \to [m]} \sum_{\{p_i, p_j\} \in E} w_{ij} \left[\pi(p_i) \neq \pi(p_j)\right] \quad \text{s.t.} \quad |P_r| \leq (1+\varepsilon)\frac{|P|}{m}, \ \forall r.$$

Method: Karlsruhe HIgh quality Partitioning uses multilevel strategy: coarsen $\rightarrow$ partition $\rightarrow$ refine. Minimizing weighted cut keeps strongly connected neighbors in same partition; balance constraint enforces approximately equal cluster sizes.

Output: Cluster labels $\pi(p) \in \{1, \ldots, m\}, p \in P$.

KaHIP in Python

Input: arbitrary data matrix $X \in \mathbb{R}^{n \times d}$.

```
# 1. Build a k-NN graph from data (e.g. sklearn NearestNeighbors)
# 2. Convert to (CSR) arrays: xadj, adjncy, adjcwgt, vwgt
# 3. Call KaHIP
import kahip
edgecut, blocks = kahip.kaffpa(vwgt, xadj, adjcwgt, adjncy,
                               nblocks, imbalance,
                               suppress_output, seed, mode)
# 4. Use 'blocks' as partition labels
```

Outputs: edgecut = total cut weight, blocks = list of partition labels.

Format: CSR (Compressed Sparse Row) convention: undirected, integer weights.

KaHIP Parameters and Graph Format

Name	Type	Meaning
`xadj`	list[int]	Aggregate #neighbors (see next example).
`adjncy`	list[int]	Flattened neighbor indices (0-based).
`adjcwgt`	list[int]	Edge weights (importance / penalty to cut).
`vwgt`	list[int]	Node weights (usually all 1).
`nblocks`	int	Number of target parts m.
`imbalance`	float	Allowed imbalance ε (e.g. 0.03).
`mode`	int	Preconfigure: 0 FAST, 1 ECO, 2 STRONG.
`seed`	int	Random seed for deterministic runs.

- Larger imbalance allows cheaper cuts but unequal partitions.
- Higher mode gives better accuracy at higher runtime.

CSR Example: How k-NN graph is stored

Example: $n = 4$ cycle, $k = 2$ neighbors. Adjacency (undirected):

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Flattened CSR representation:

```
# node 0 -> [1, 2]; node 1 -> [0, 3]; node 2 -> [0, 3]; node 3 -> [1, 2]
xadj    = [0, 2, 4, 6, 8]   # aggregate #neighbors
adjncy  = [1, 2, 0, 3, 0, 3, 1, 2]
adjcwgt = [1, 1, 1, 1, 1, 1, 1, 1]
vgt = [1, 1, 1, 1]  # weight per node (here 1)
```

`xadj`: $(i+1)$'s #neighbors = `xadj[i+1] - xadj[i]`.

`adjncy` has the neighbor indices in sequence.

Edge weights in `adjcwgt` follow same ordering.

Classifier Model for Partition Learning

Given data points $p_i \in \mathbb{R}^d$, KaHIP partition labels $\pi(p_i) \in \{1, \ldots, m\}$, **learn neural map** using multilayer perceptron (MLP) f_θ with ReLU activations:

$$M : \mathbb{R}^d \to \mathbb{R}^m, \quad M(p) = \mathsf{softmax}(f_\theta(p)).$$

Output: Neural classifier predicting bin assignment:

$$M(p) = [p_1, \ldots, p_m], \quad p_r = \mathsf{Pr}[\pi(p) = r].$$

Softmax probability distribution:

$$\mathsf{softmax}(z)_r = \frac{e^{z_r}}{\sum_{j=1}^{m} e^{z_j}}, \quad r = 1, \ldots, m.$$

Training Objective minimizes Cross-Entropy loss:

$$\min_\theta \ \mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathsf{CE}\big(M(p_i),\, \pi(p_i)\big).$$

MLP definition

Model: 3-layer fully connected network

```
import torch, torch.nn as nn

# Define a simple feedforward neural network (MLP)
class MLPClassifier(nn.Module):
    def __init__(self, d_in, n_out):
        super().__init__()
        # Sequential block: Linear -> ReLU -> Linear -> ReLU -> Linear
        self.net = nn.Sequential(
            nn.Linear(d_in, 512),   # Input layer
            nn.ReLU(),              # Activation
            nn.Linear(512, 512),    # Hidden layer
            nn.ReLU(),              # Activation
            nn.Linear(512, n_out)   # Output logits
        )
    def forward(self, x):
        # Forward pass through the network
        return self.net(x)
```

Training setup (PyTorch)

Initialize model, optimizer, and data loader

```
device = torch.device("cuda" if
         torch.cuda.is_available() else "cpu")

model = MLPClassifier(d_in=128, n_out=m).to(device)
opt = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

# Hyperparameters
epochs = 10          # number of passes over the dataset
batch_size = 32      # samples per mini-batch
loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

Training loop (PyTorch)

Forward–backward–update routine

```
for epoch in range(epochs):
    for x, y in loader:
        x, y = x.to(device), y.to(device)  # Move to GPU/CPU
        opt.zero_grad()                    # 1. Reset gradients
        logits = model(x)                  # 2. Forward pass
        loss = loss_fn(logits, y)          # 3. Compute loss
        loss.backward()                    # 4. Backward pass
        opt.step()                         # 5. Update weights
```

Multi-Probe Query: Select top-T bins

Goal: improve accuracy by probing multiple likely bins instead of only one.

For $q \in \mathbb{R}^d$, let $p_r \geq 0$ be the predicted probability that q belongs to bin r:

$$p = M_\theta(q) = \mathsf{softmax}(f_\theta(q)) = (p_1, \ldots, p_m).$$

Select indices of T largest probabilities:

$$B(q) = \mathsf{TopT}\left(p_1, \ldots, p_m\right) \subseteq \{1, \ldots, m\}.$$

Candidate set:

$$\mathsf{Candidates}(q) = \bigcup_{r \in B(q)} \{\, p_i : \pi(p_i) = r \,\}.$$

Trade-off: larger $T \Rightarrow$ higher accuracy, smaller $T \Rightarrow$ faster query.

Experiments: Neural LSH vs Baselines

Setup and Metrics

Datasets:

- SIFT (1M, $d{=}128$), GloVe (1.2M, $d{=}100$), MNIST (60k, $d{=}784$)

Baselines: k-means, ITerative Quantization (ITQ), Cross-polytope LSH, PCA tree, Randomized Partition (RP) tree.

Acc@10: for each query q, compute its true $k{=}10$ nearest neighbors under exact distances. Let $\mathrm{Cand}(q)$ be the retrieved candidate set; report

$$\mathrm{Acc@10} = \frac{1}{|Q|} \sum_{q} \frac{|\mathrm{Cand}(q) \cap \mathrm{TrueNN}_{10}(q)|}{10}.$$

Candidate count: For $C(q) = |\mathrm{Cand}(q)|$, report $\mathrm{avg}_q\, C(q)$.

Experiments: Neural LSH vs Baselines

Results and Takeaways

Results:

- Neural LSH $\Rightarrow$ fewer candidates at same accuracy.
- Up to $2.3\times$ fewer candidates vs. k-means.
- Lower variance across queries (smaller 0.95-quantile gap).

Takeaway: Learned partitions outperform quantization, tree-based methods, and CP-LSH, giving stable latency and strong recall.

Contents

Introduction to AI

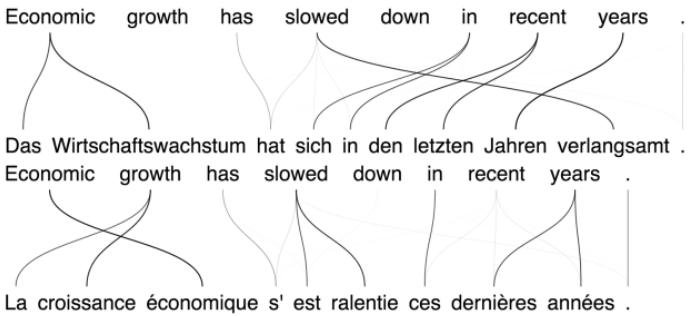
Machine Learning

Machine learning (ML) uses statistical techniques to give computer systems the ability to ``learn'' (i.e. progressively improve performance on a specific task) from data, without being explicitly programmed.

Neural Networks

Neural Networks (and Deep Learning) is a class of ML algorithms that use a cascade of layers of nonlinear processing units for feature extraction and transformation, mimicking the human neurons. Each successive layer uses the output from the previous layer as input.

Examples of learning tasks

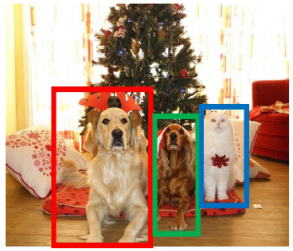


Semantic Segmentation

GRASS, CAT, TREE, SKY

No objects, just pixels

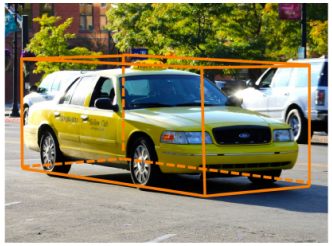
2D Object Detection



DOG, DOG, CAT

Object categories + 2D bounding boxes

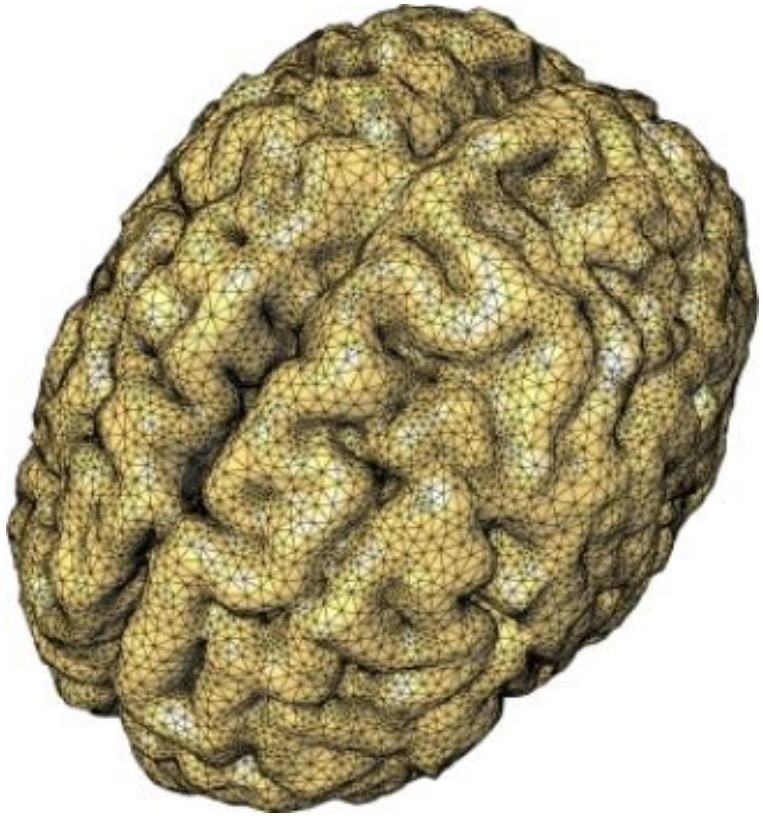
3D Object Detection



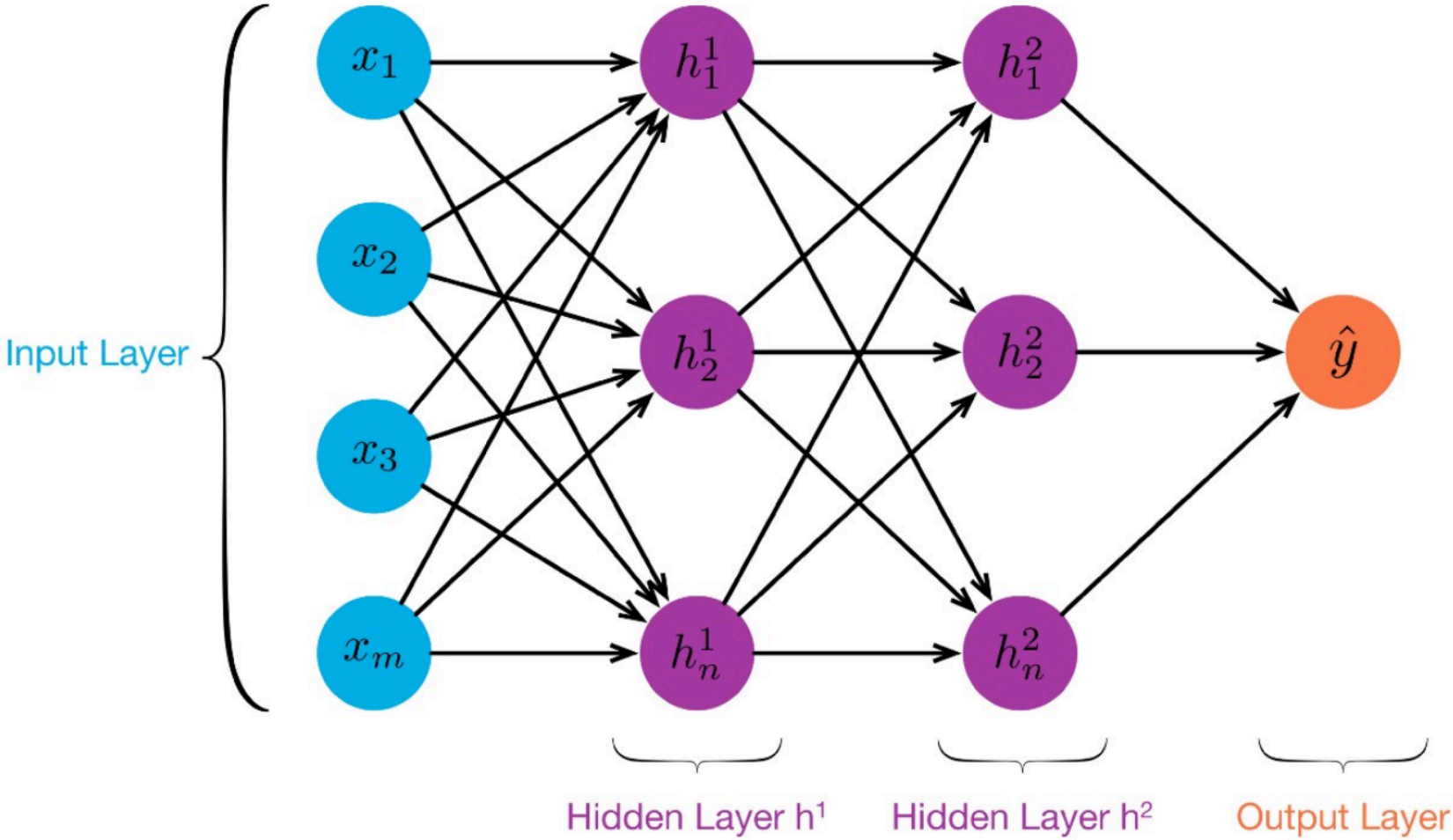
Car

Object categories + 3D bounding boxes

Live Neural Network

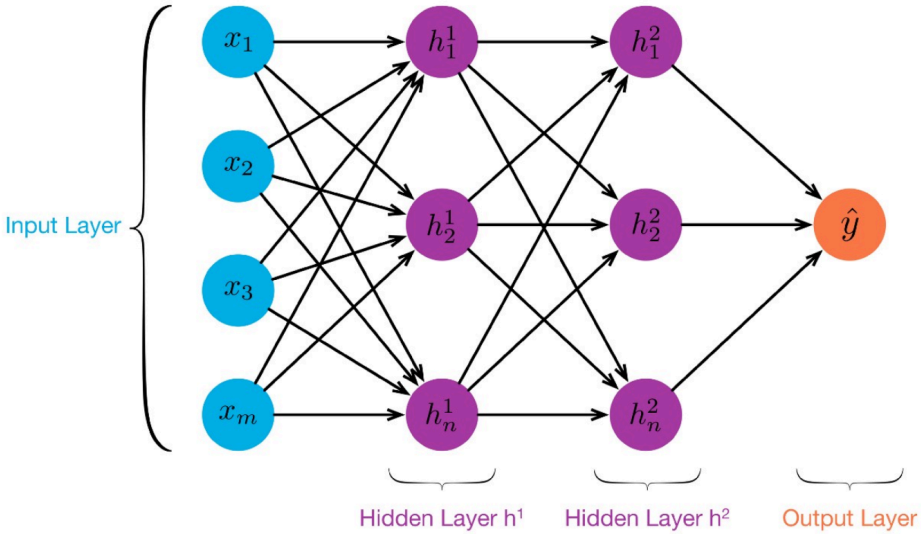


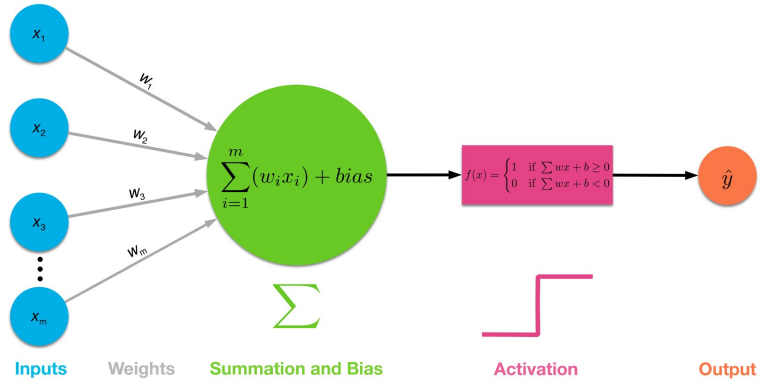
Artificial Neural Network



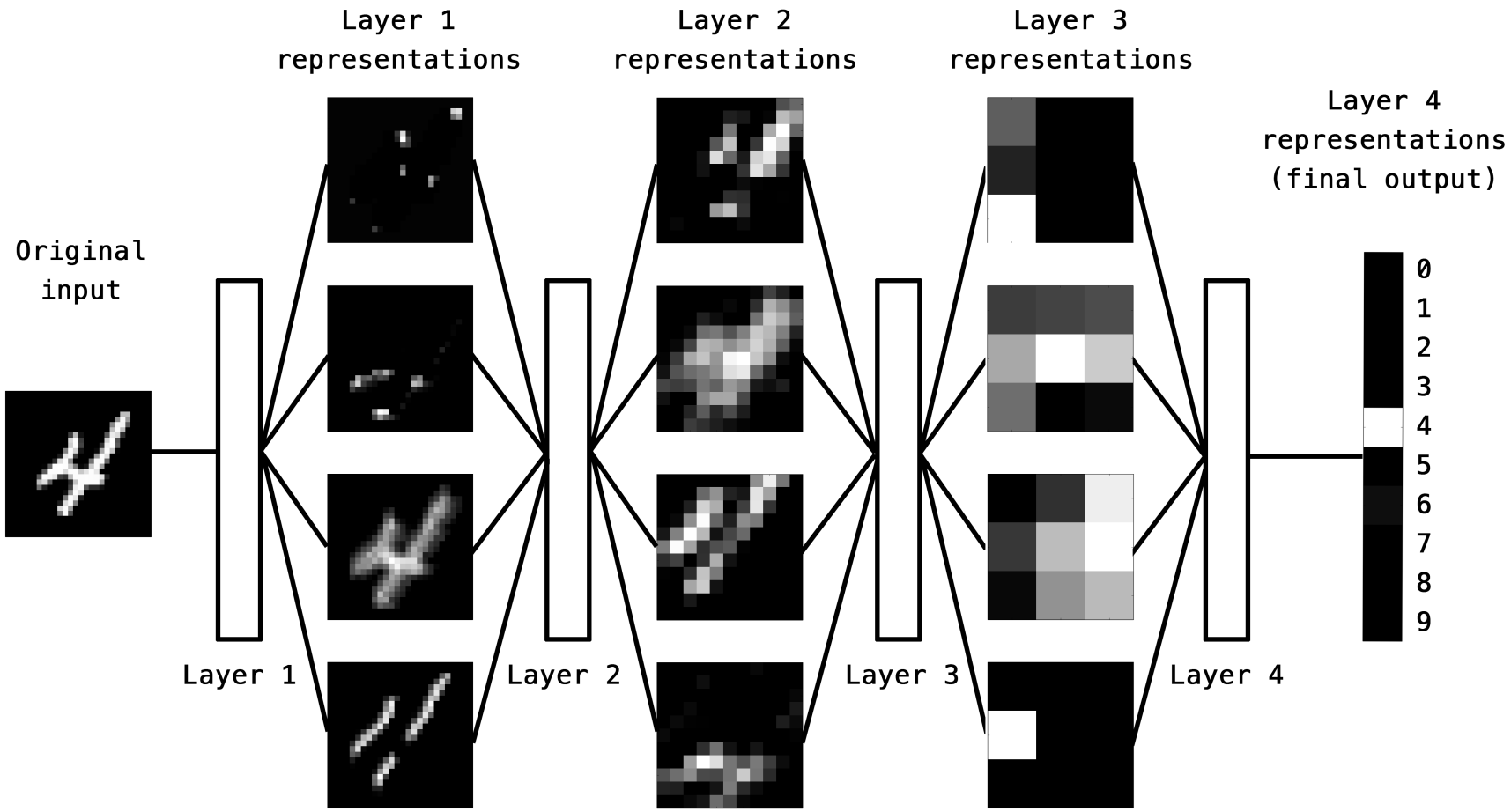
Neural Networks

- NN is a cascade of **hidden layers**
- Each layer is a **data transformation** function
  - weights/parameters determine the transformation
  - transformations are differentiable for improving output



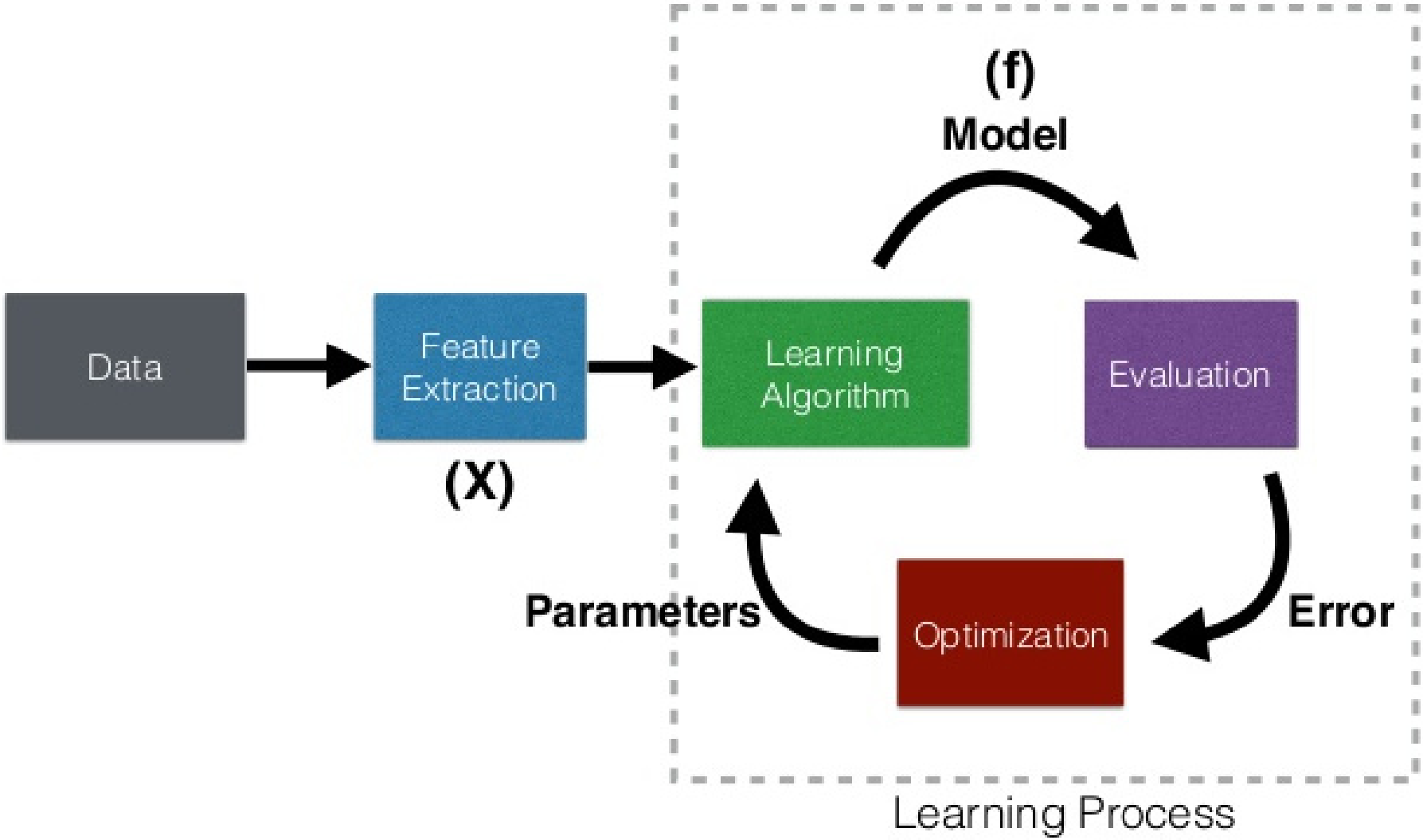


NN Layers of representations



- Input data transformed so that **irrelevant** information is **filtered out**
- Useful information is magnified and refined

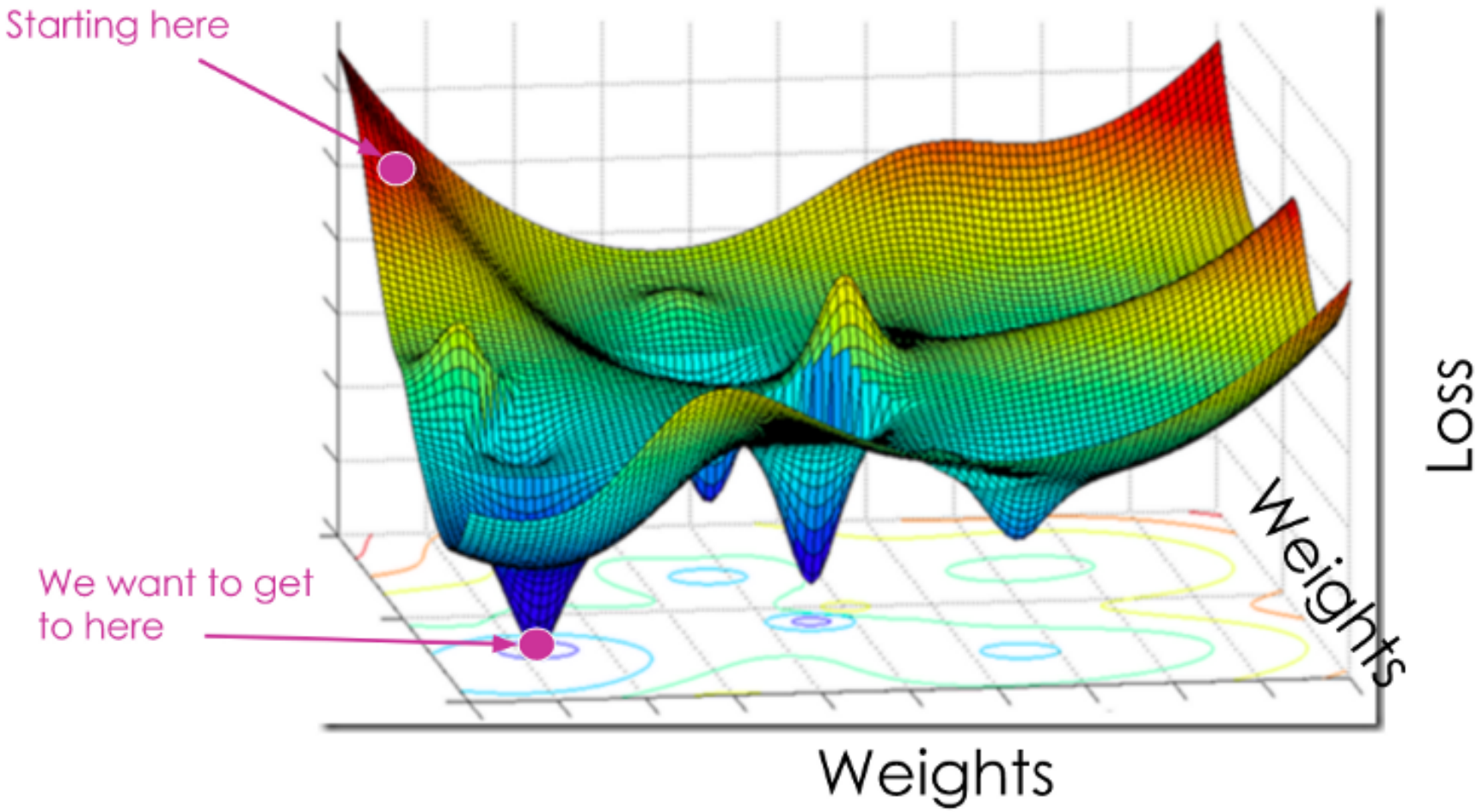
Learning process



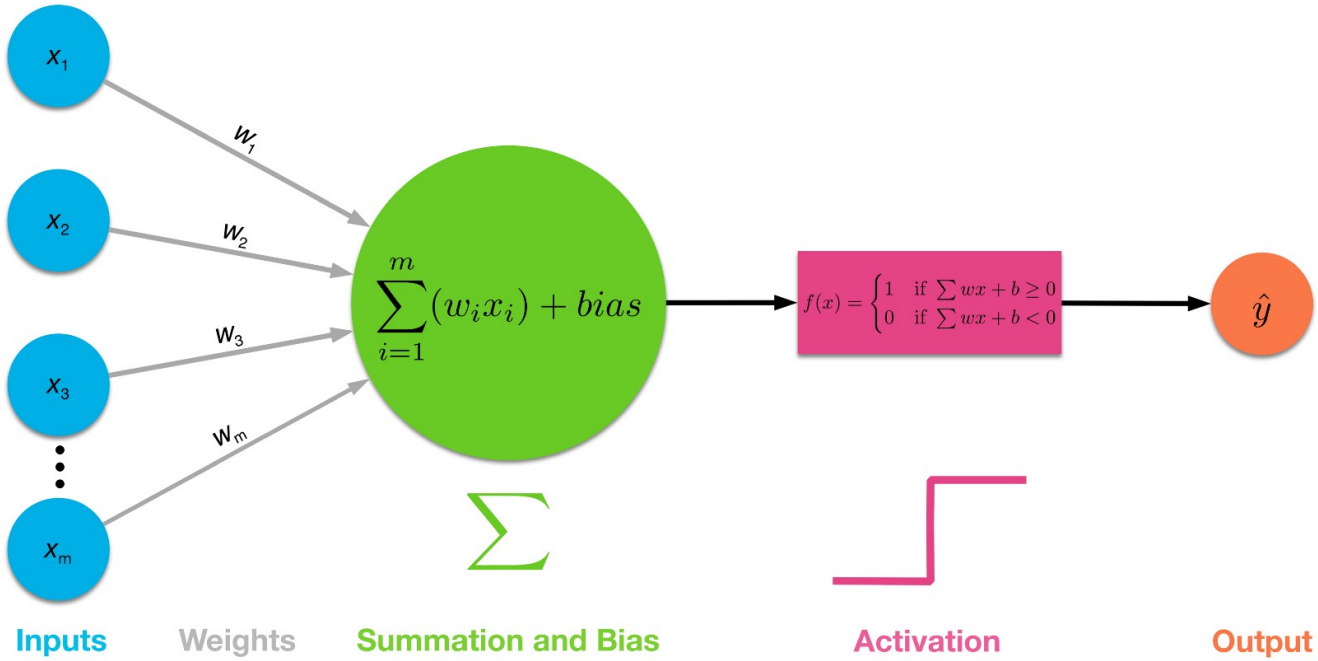
Several epochs (re-ordering instances), each split in batches.

Goal of training

Goal of training $\rightarrow$ find **weights** that minimise loss function



Activation functions

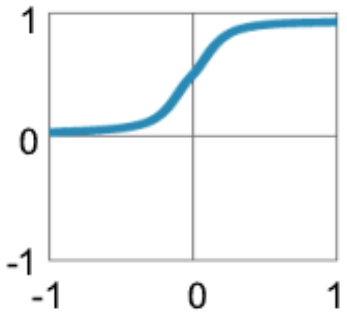


- Decide whether this neuron is **active** or not
- **Non-linear activation** transformation on (weighted) input
- Overall generate **non-linear mappings** from inputs to outputs

Activation functions (I)

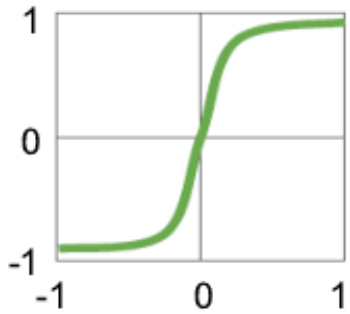
Traditional Non-Linear Activation Functions

Sigmoid



$y=1/(1+e^{-x})$

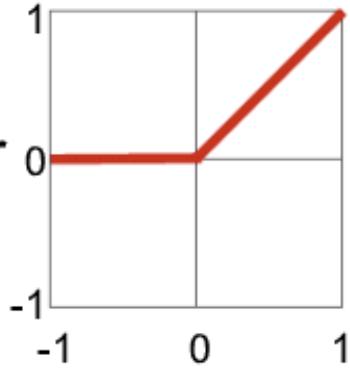
Hyperbolic Tangent



$y=(e^{x}-e^{-x})/(e^{x}+e^{-x})$

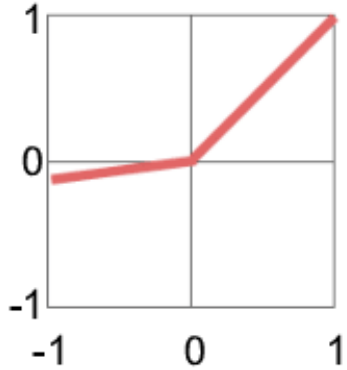
Modern Non-Linear Activation Functions

Rectified Linear Unit (ReLU)



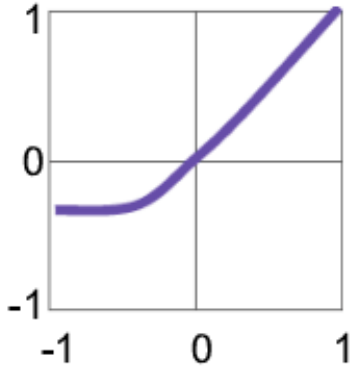
$y=\max(0,x)$

Leaky ReLU



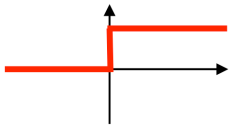
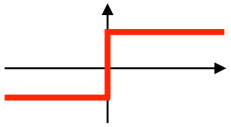
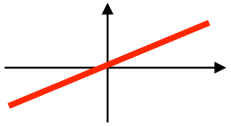
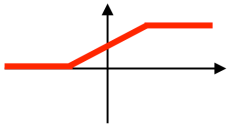
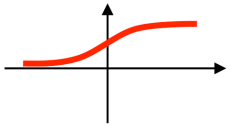
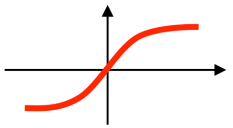
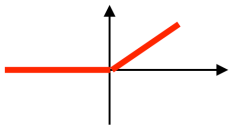
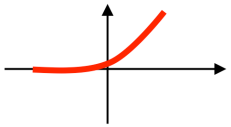
$y=\max(\alpha x,x)$

Exponential LU



$y=\begin{cases} x, & x\geq 0 \\ \alpha(e^{x}-1), & x<0 \end{cases}$

α = small const. (e.g. 0.1)

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

Loss (error) functions

$y =$ true output label/value, $\widehat{y} =$ predicted class/value,
$n =$ # training instances per batch, $M =$ #classes.

- Regression problems

Mean Squared Error (MSE): $$\frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y_i})^2$$

Mean Absolute Error (MAE): $$\frac{1}{n}\sum_{i=1}^{n}|y_i - \widehat{y_i}|$$

- Classification problems

binary cross-entropy: $$-\frac{1}{n}\sum_{i=1}^{n}(y_i \log \widehat{y_i} + (1-y_i)\log(1-\widehat{y_i}))$$

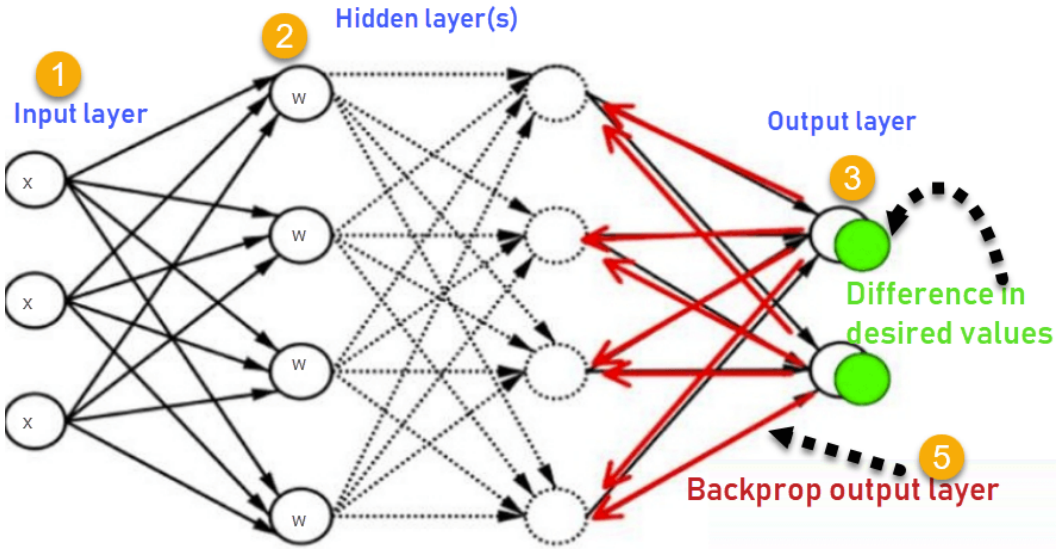
(M classes, one label) cross-entropy: $$-\frac{1}{n}\sum_{i=1}^{n}\sum_{j}^{M} y_{ij} \log \widehat{y_{ij}}$$

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy
Continuous	Arbitrary	See part III: GAN, VAE, FVBN	Various

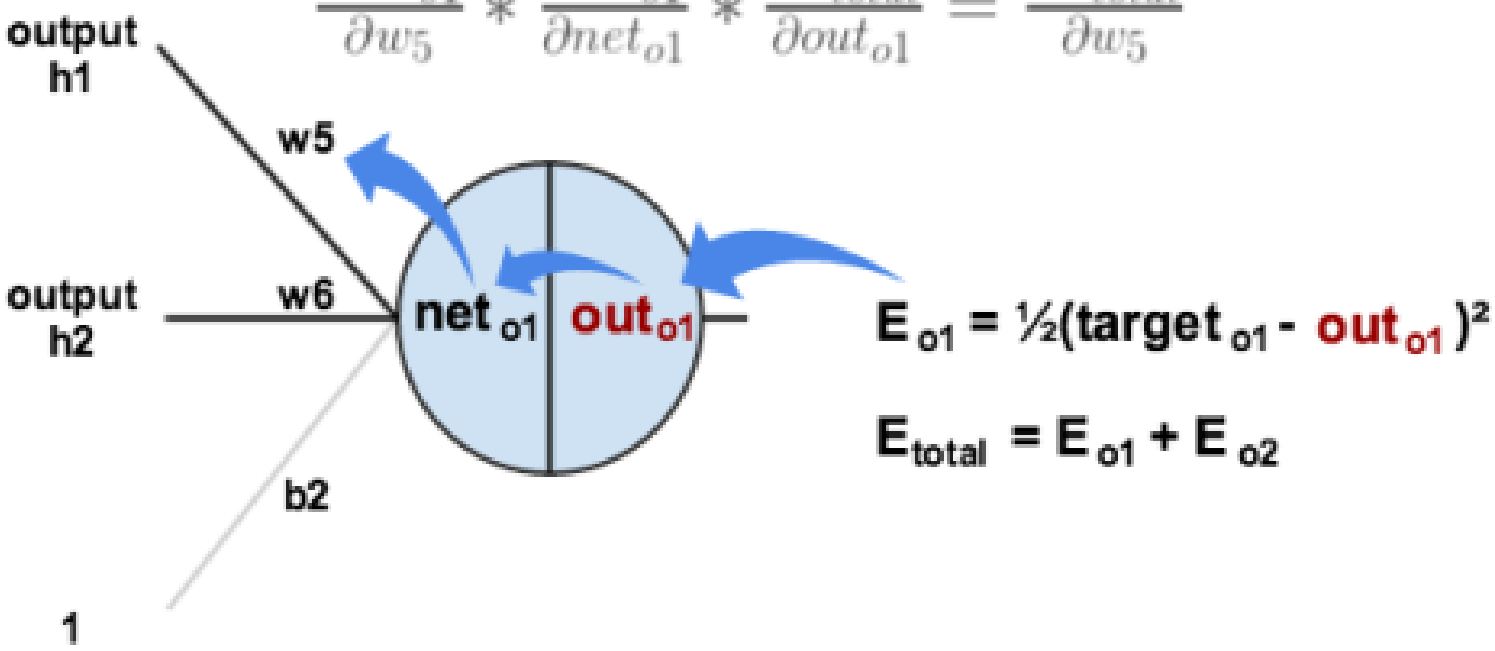
Backward propagation

- Forward propagation: compute activations (black arrows)
- Backprop: compute derivatives optimizing parameters/weights (red)
- Gradient descent exploits chain rule $\frac{dL}{dw} = \frac{dL}{du}\frac{du}{dw}$ wrt **weights**
- Learning rate (e.g. 10^{-5}): step to change weights



Backpropagation

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



Correct earlier weights e.g. w_1 by similarly computing $\partial E_{total}/\partial w_1$, assuming all other weights and inputs fixed.

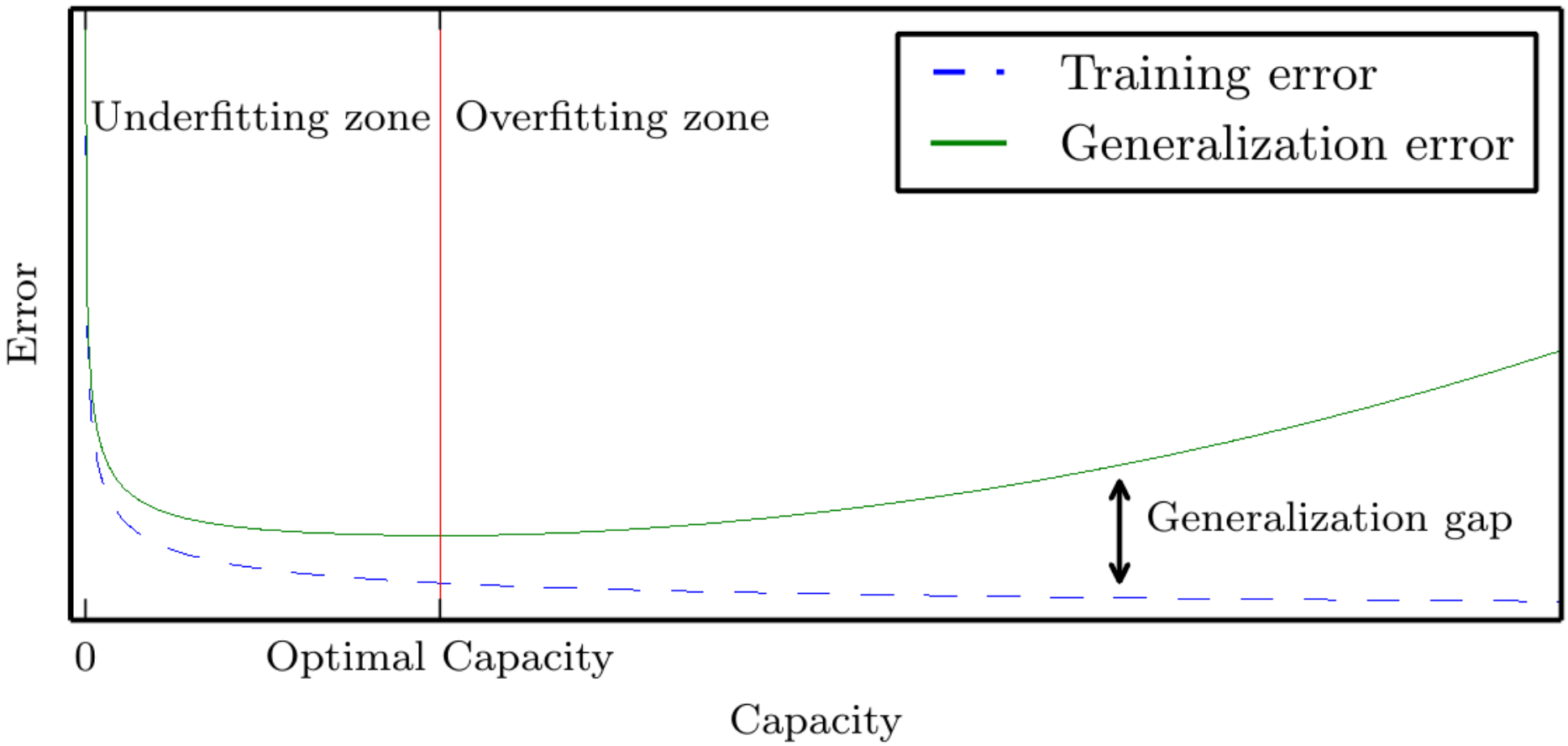
All such derivatives computed offline and instantiated.

Optimizers

- **SGD (Stochastic gradient descent)**
  - Incremental gradient descent, an iterative method for optimizing a differentiable objective function
- **Adam (Adaptive Moment Estimation)**
  - First-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments
  - Works well in practice, comparable to other stochastic optimization methods
- Adamax
  - Variant of Adam based on the infinity norm
- RMSProp (Root Mean Square Propagation)
  - Learning rate is adapted for each of the parameters
- Adadelta
- ...

Overfitting

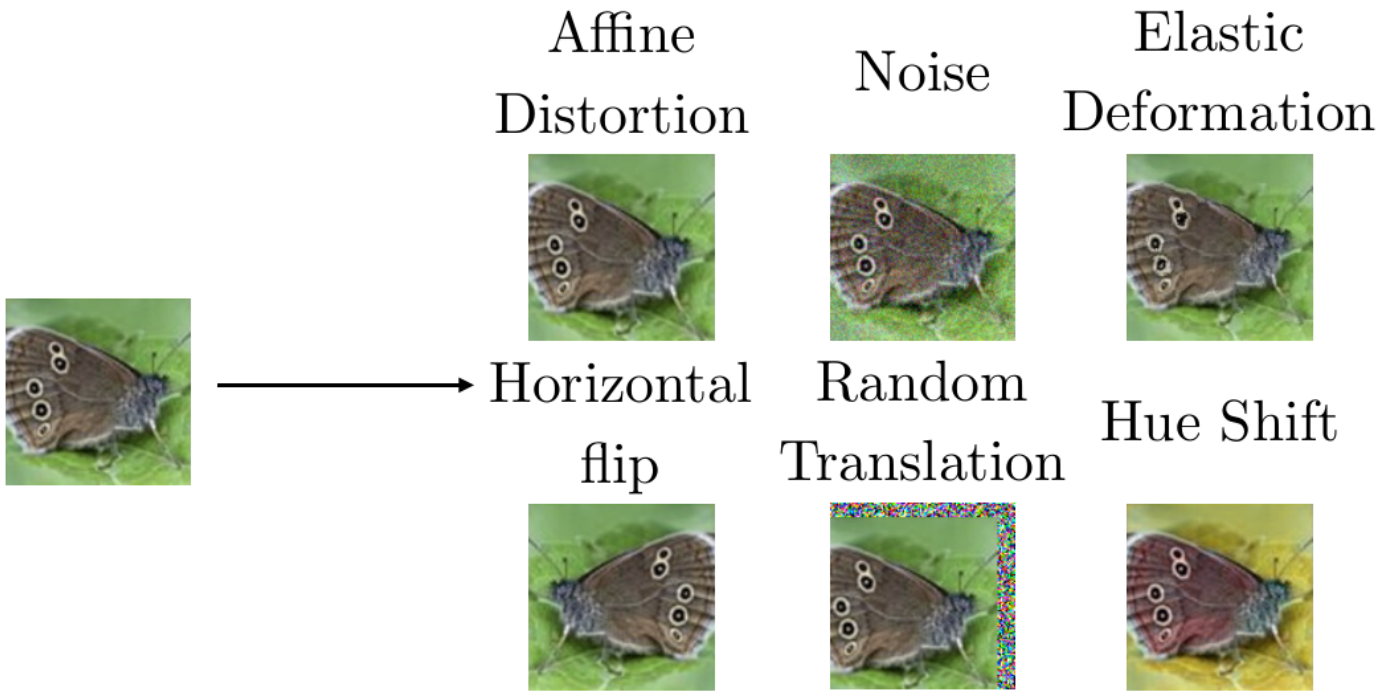
Relationship between train-data and error



Regularization

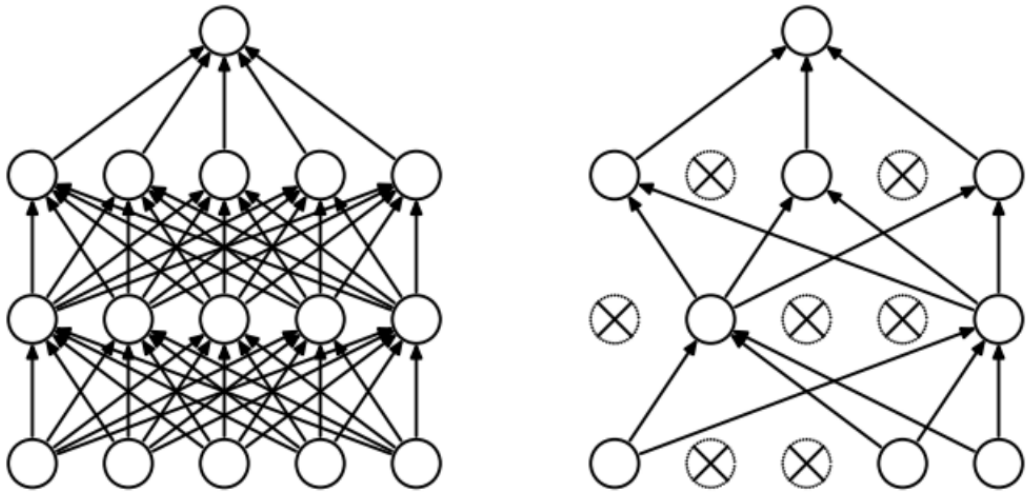
Modifications to reduce generalization error, not training error:

- Data augmentation (figure)
- Injecting Noise
- Node/Connection Dropout (next slide)



Dropout

- Regularization technique to reducing overfitting...
- ...and prevent complex co-adaptations at training.
- Mute units/nodes for specific batch: ignore backprop.



(a) Standard Neural Net

(b) After applying dropout.

NN as universal approximators

Universal Approximation Theorem

Let ξ be a non-constant, bounded, monotonically-increasing continuous ``activation" function, target $f : [0,1]^d \to \mathbb{R}$ continuous, and $\epsilon > 0$. Then, $\exists k$, parameters $a, b \in \mathbb{R}^k, W \in \mathbb{R}^{k \times d}$ s.t.

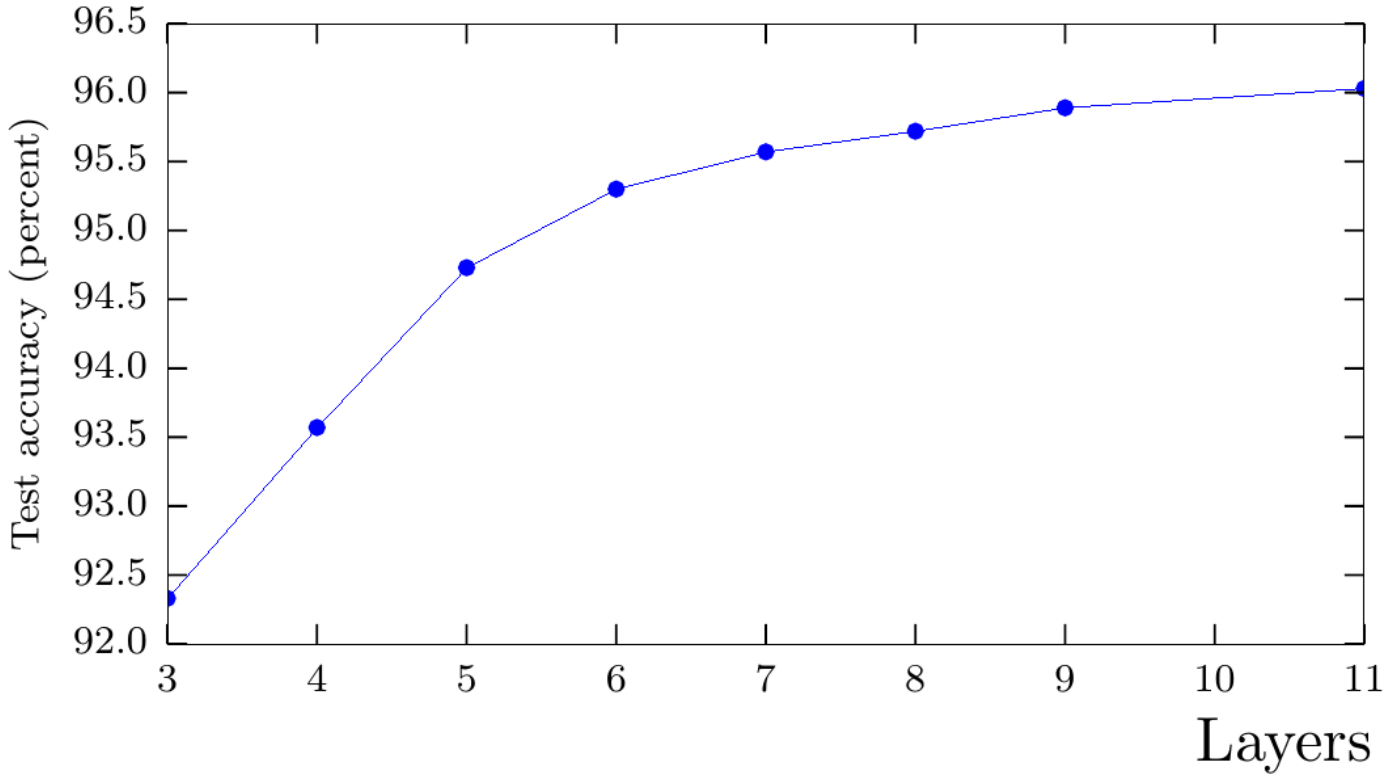
$$\left| \sum_{i=1}^{k} a_i \xi(w_i^T x + b_i) - f(x) \right| < \epsilon.$$

- Any f approximated arbitrarily well by NN with one hidden layer
- How many neurons? How to find the parameters?
- Does it generalize well? Does it overfit?
- Shallow nets work poorly for high-dimensional data like images

Cybenko 1989; Hornik 1991

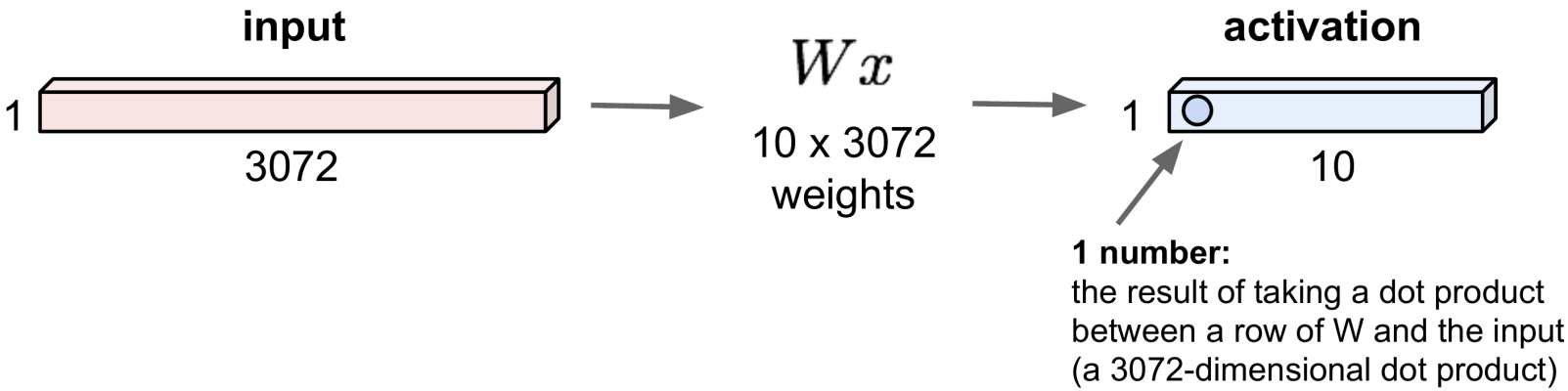
Better Generalization with Greater Depth

- Empirical results show that **deeper** networks **generalize** better
- Test accuracy consistently increases when increasing depth



Transcribe multidigit numbers from photographs of addresses [Goodfellow'14]

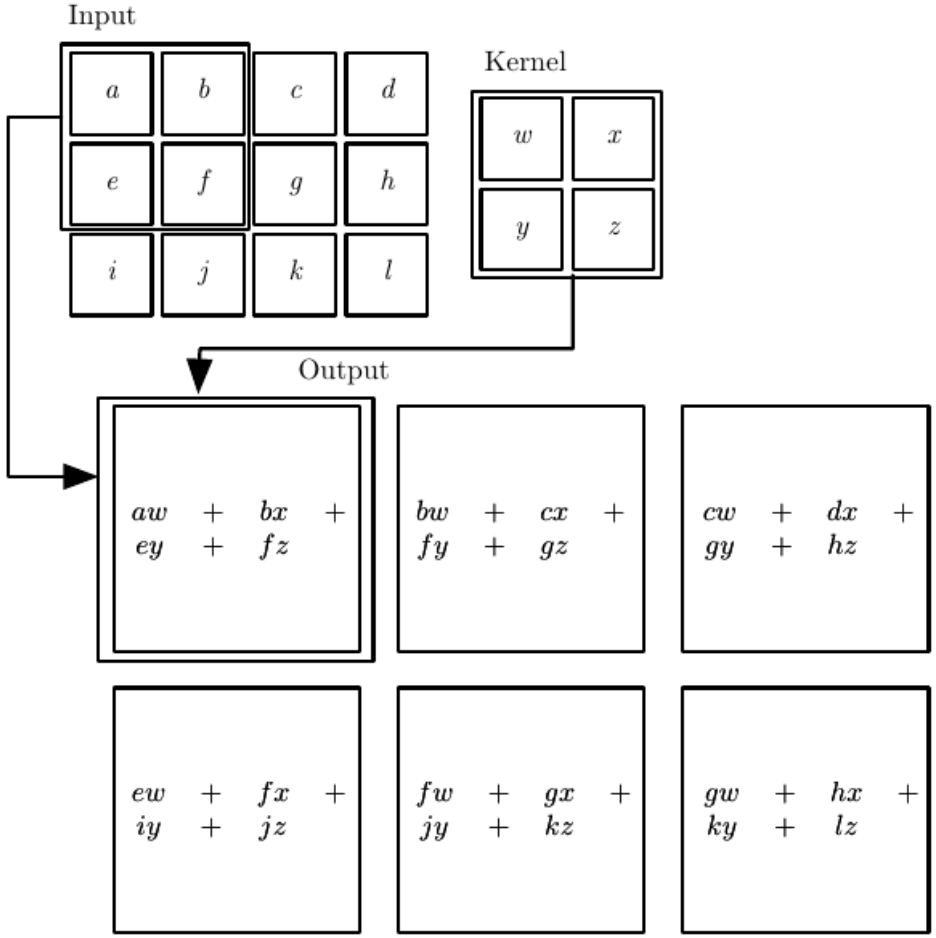
Fully Connected Layer



Matrix multiplication: W is a 10×3072 real matrix.

Convolution definition

- Pointwise matrix multiply
- The kernel (convolution mask) is a small matrix (or flipped)



Convolution

Motivation: Sparse / Parameter sharing / Equivariant representations

Original:

Sharpen:

0	-1	0
-1	5	-1
0	-1	0

Blur: $\frac{1}{9}$

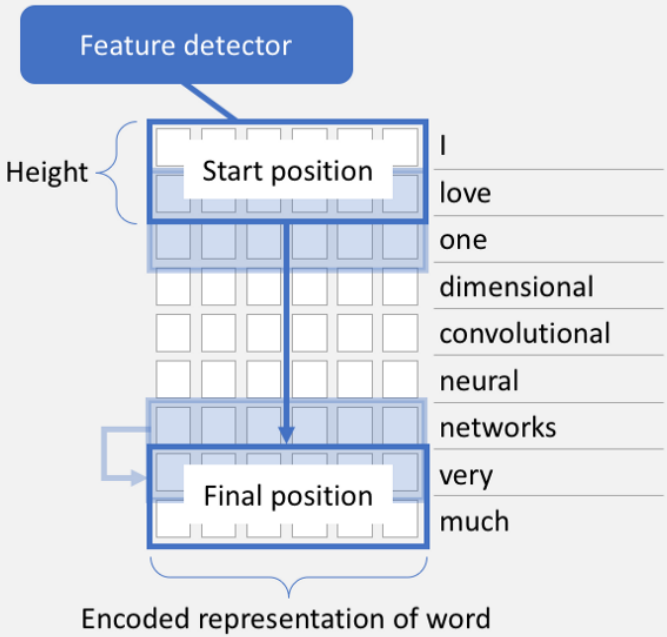
1	1	1
1	1	1
1	1	1

Edge detect:

-1	-1	-1
-1	8	-1
-1	-1	-1

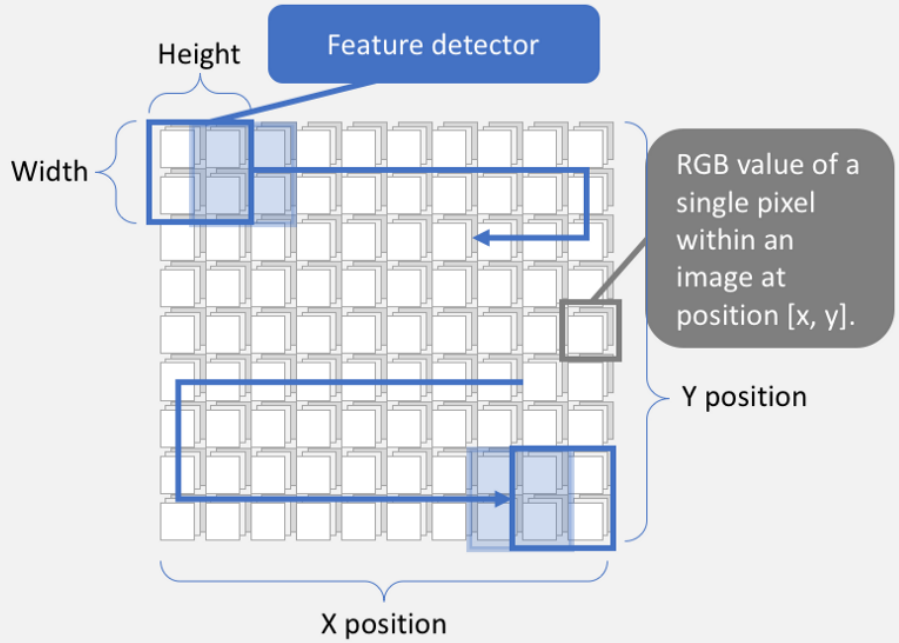
1D vs 2D convolution

1D Convolutional - Example



In this example for natural language processing, a sentence is made up of 9 words. Each word is a vector that represents a word as a low dimensional representation. The feature detector will always cover the whole word. The height determines how many words are considered when training the feature detector. In our example, the height is two. In this example the feature detector will iterate through the data 8 times.

2D Convolutional - Example



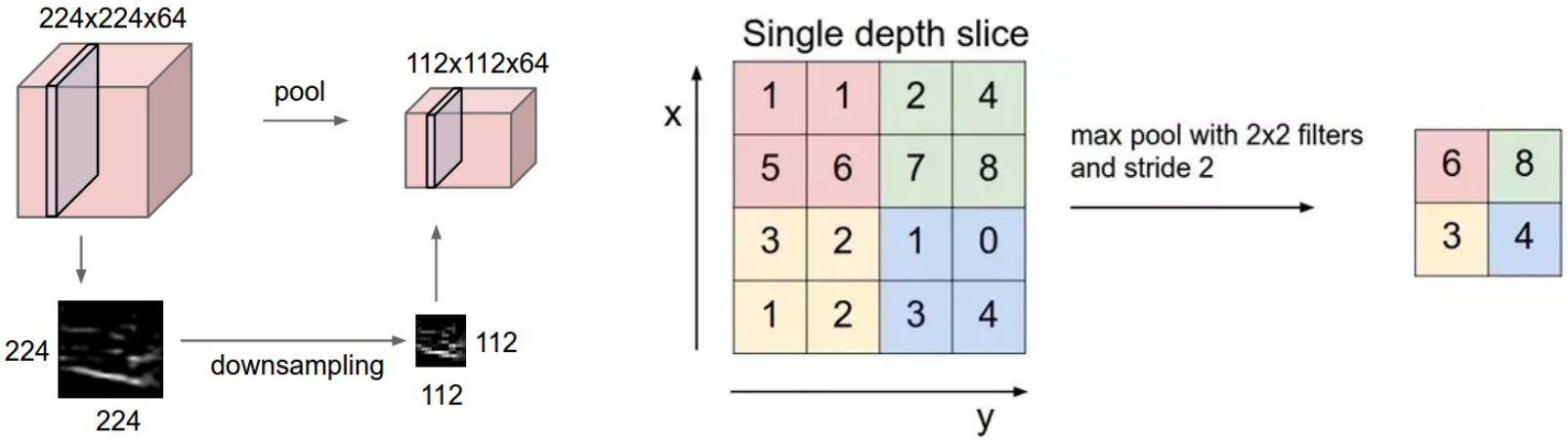
In this example for computer vision, each pixel within the image is represented by its x- and y position as well as three values (RGB). The feature detector has a dimension of 2 x 2 in our example. The feature detector will now slide both horizontally and vertically across the image.

---

https://blog.goodaudience.com/introduction-to-1d-convolutional-neural-networks-in-keras-for-time-sequences-3a7ff

Pooling (Downsampling)

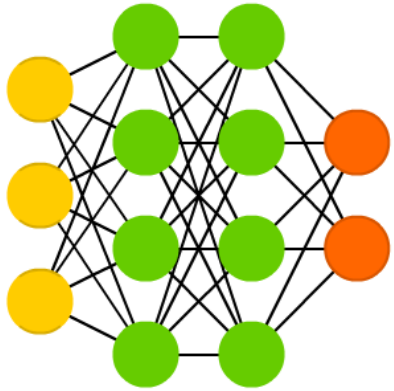
- Makes representations smaller and more manageable
- Helps make representation roughly invariant to small input translations
- **Max**, Avg, L^2 norm, weighted avg distance from the central pixel



Recurrent Neural Networks (RNN)

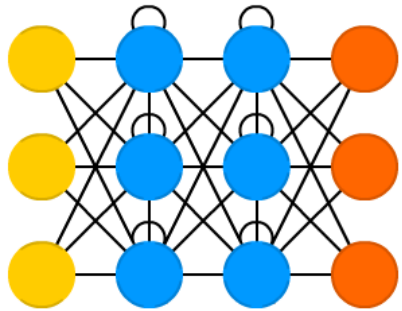
NNs mainly distinguished in:

- Feedforward neural networks (FNN):
  Features processed independently
- Recurrent neural networks (RNN):
  Features processed sequentially e.g. time series

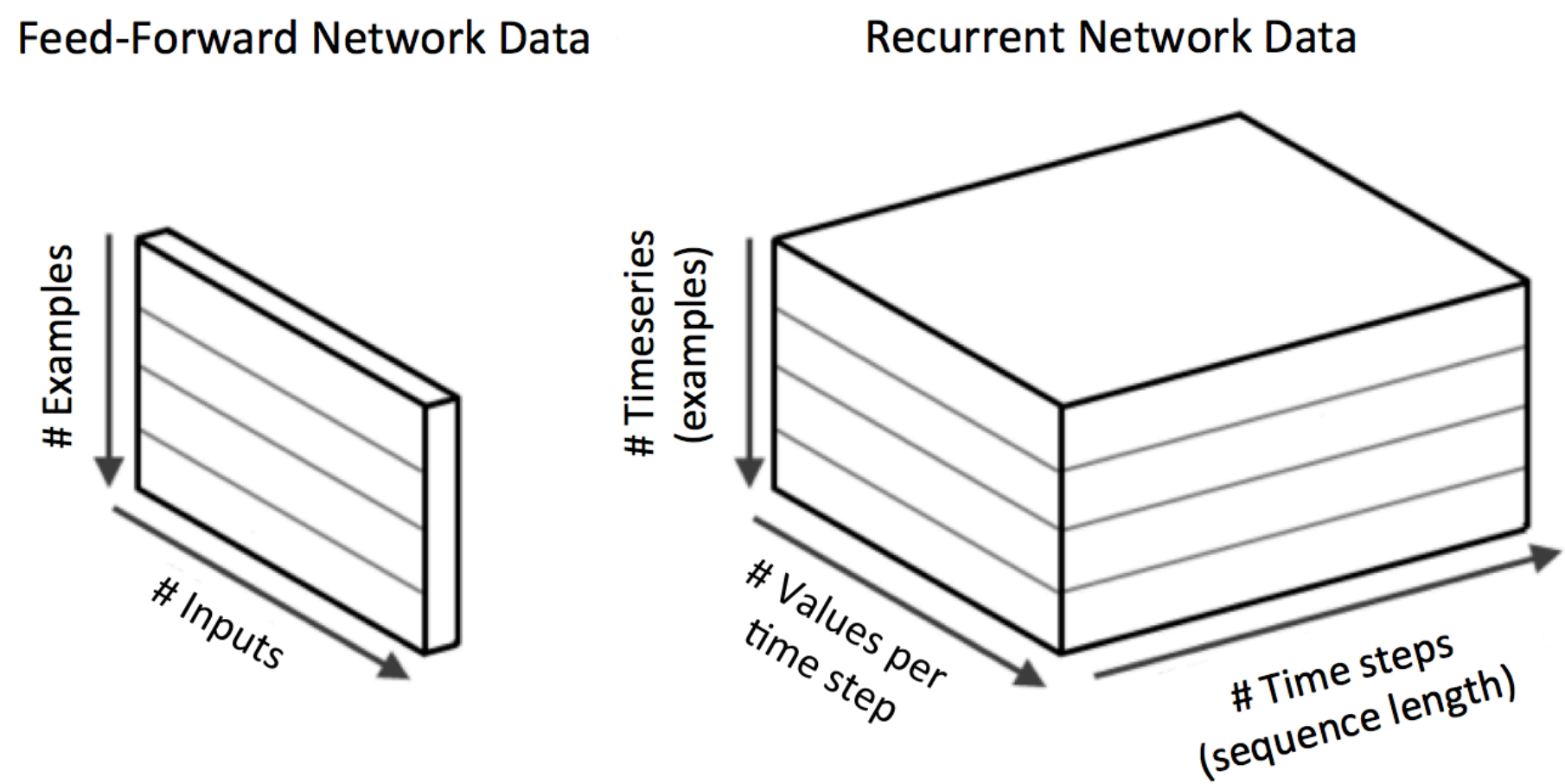


RNN allow **cyclic** connections

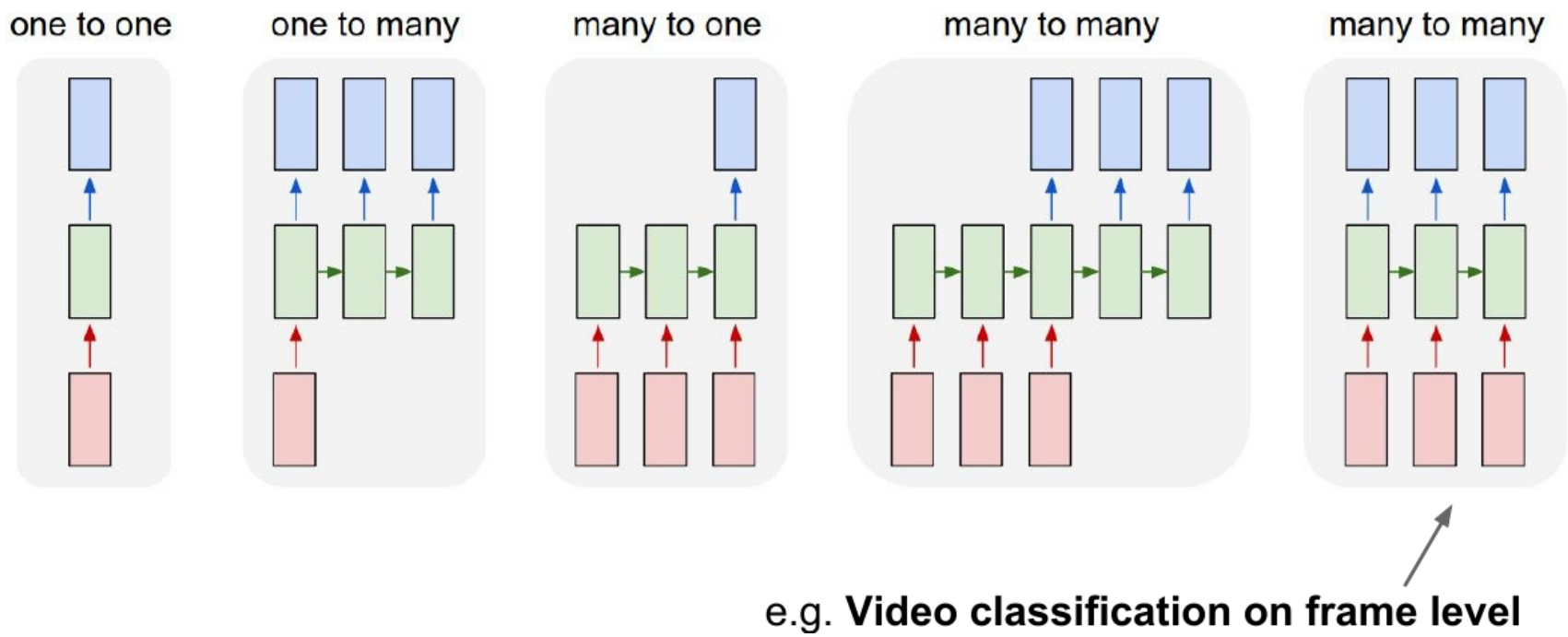
$\longrightarrow$ fit best to process **sequential** data



RNN input


Feed-Forward Network Data
Examples
Inputs
Recurrent Network Data
Timeseries (examples)
Values per time step
Time steps (sequence length)

RNN: Type of sequences



1-many: image captioning (image $\rightarrow$ word sequence)

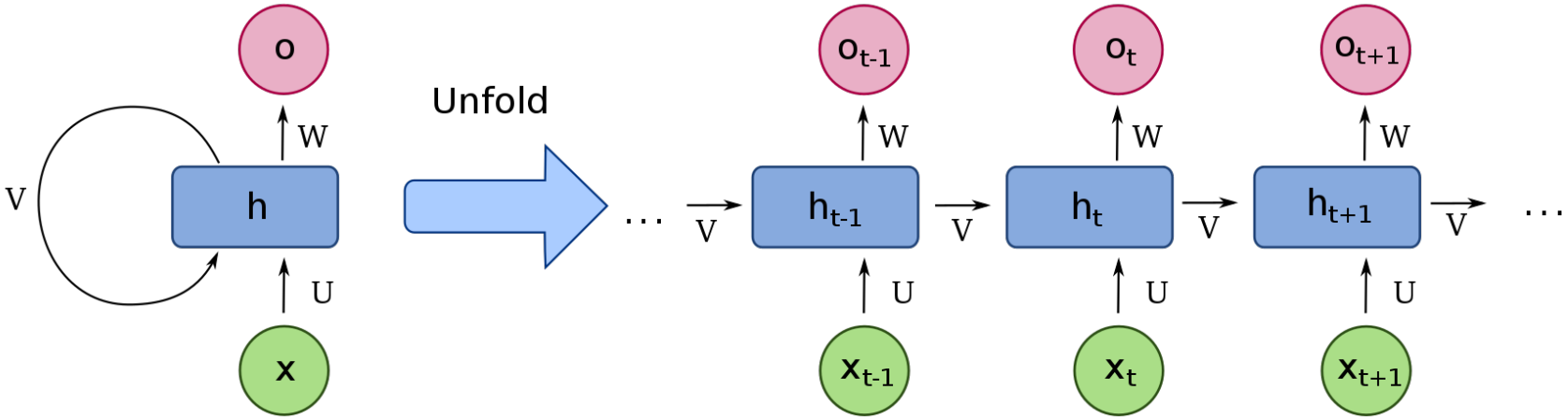
many-1: sentiment classification (word sequence $\rightarrow$ sentiment)

many-many: machine translation (word sequence $\rightarrow$ word seq.)

many-many: video classification on frame level

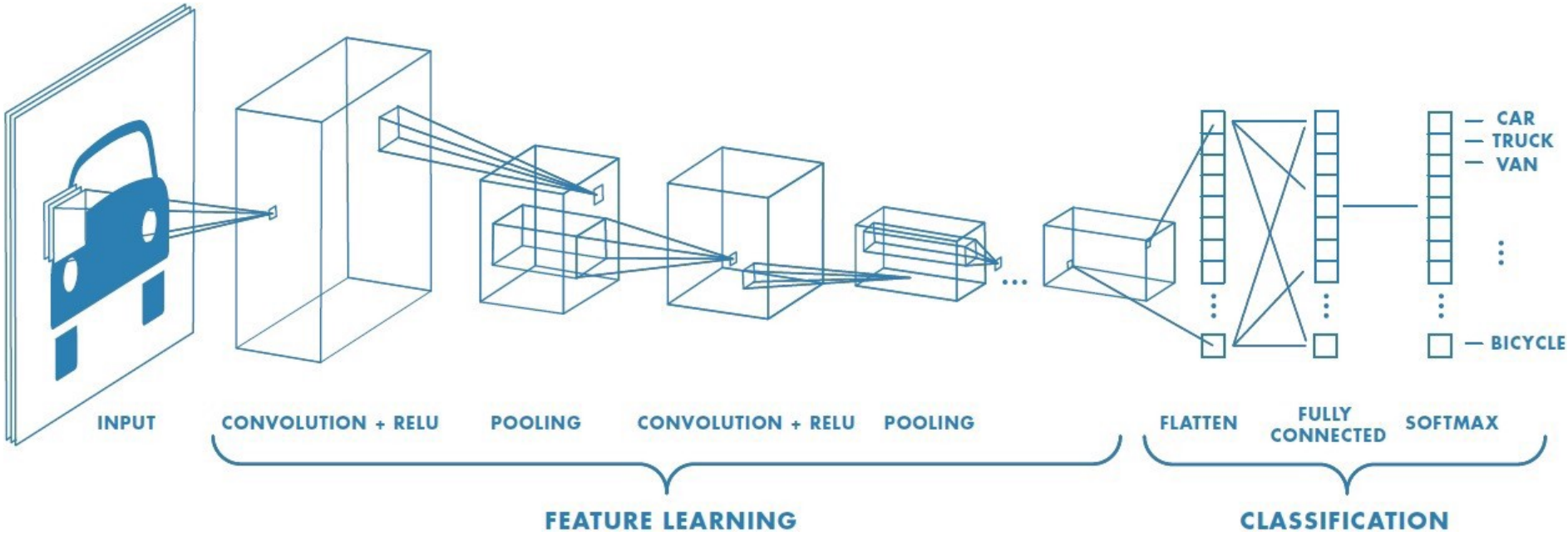
Recurrent neural networks

- allow cyclic connections
- ...essentially offer internal state (memory)
- keep track of arbitrarily long-term dependencies (boon/issue)
- internal state processes input x_t's by applying recurrence formula at every time step t: new state $h(t) \leftarrow f_w(h_{t-1}, x_t)$.

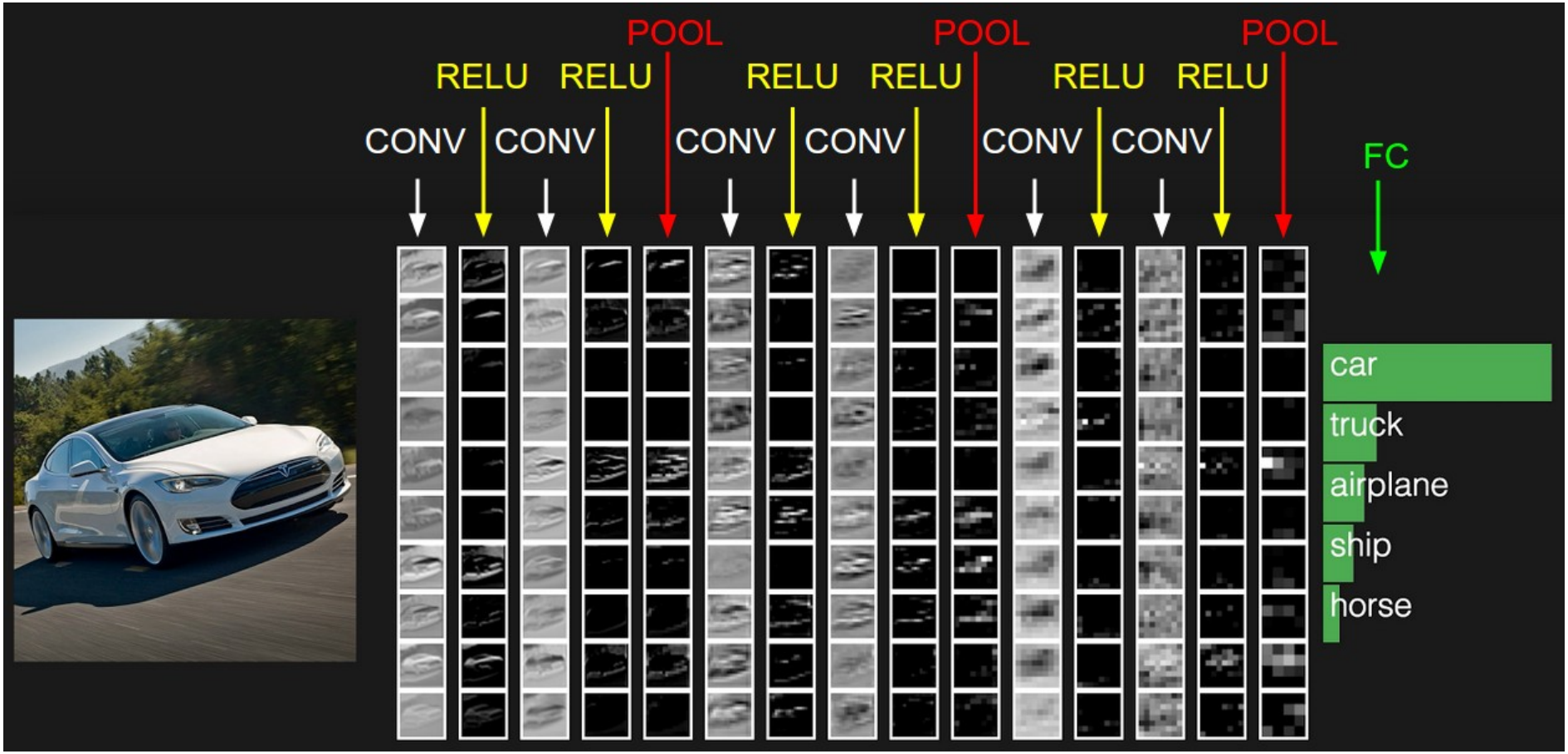


Numerics: Back-propagated gradients may vanish ($\to 0$) or explode ($\to \infty$).

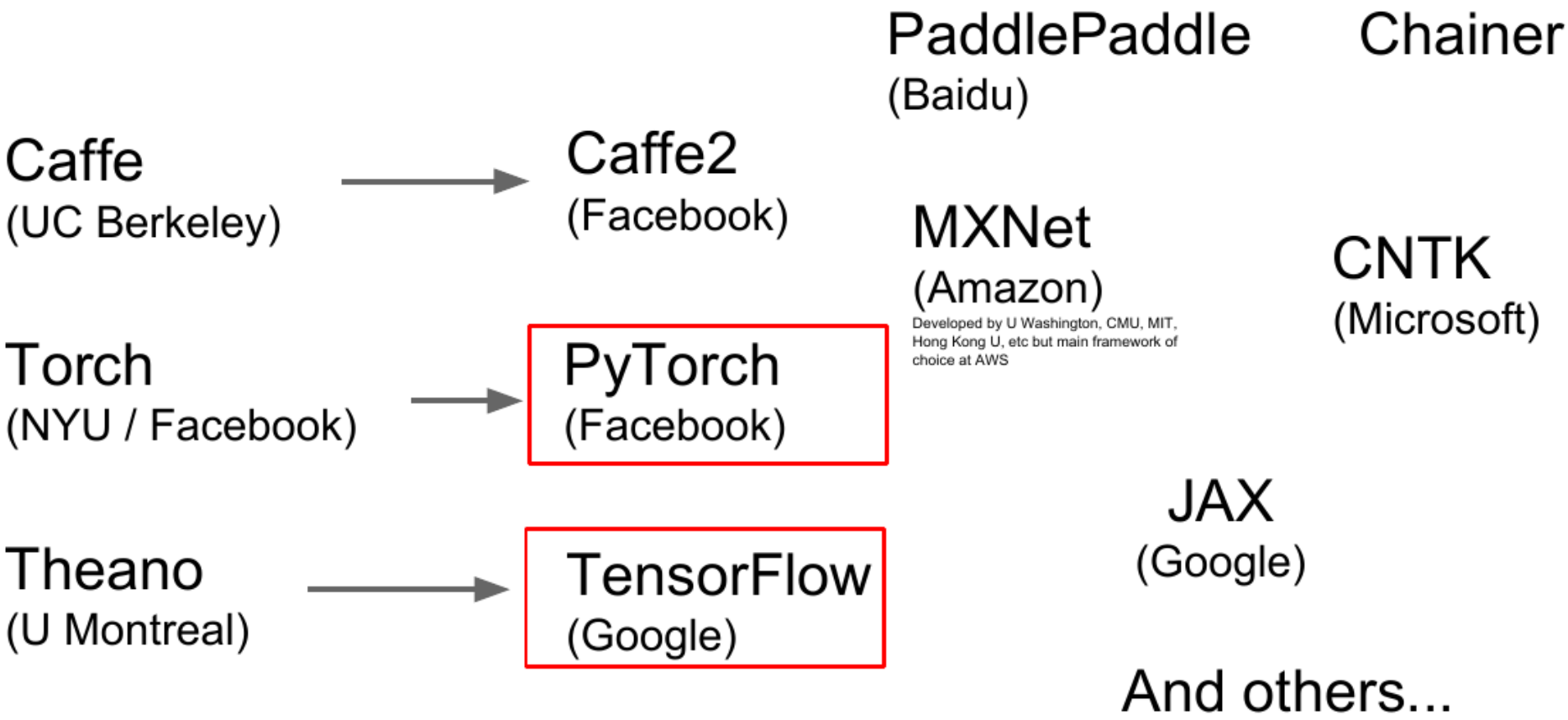
Typical Neural Network



CNN example


POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

Development frameworks



Training may be done on GPU environment Colab (Google).

References

- Deep Learning book (https://www.deeplearningbook.org/) (www.dropbox.com/s/oj3olyzxvnchrqs/SGP%202018.pdf?dl=0)
- Yannis Avrithis, Deep learning for computer vision (6h short course) (http://erga.di.uoa.gr/meetings/Slides_Avrithis.zip)
- J.J. Allaire, Machine Learning with TensorFlow and R (https://beta.rstudioconnect.com/ml-with-tensorflow-and-r/)