Discretization

$$entropy(S) = -\sum_i P_i . \log_2 P_i$$

64	65	68	69	70	71	72	73	74	75	80	81	83	85
yes	no	yes	yes	yes	no	no	no	yes	yes	no	yes	yes	no

$$entropy(S_{left}) = -\frac{4}{5}\log_2\frac{4}{5} - \frac{1}{5}\log_2\frac{1}{5} = 0.722\,bits$$

$$entropy(S_{right}) = -\frac{4}{9}\log_2\frac{4}{9} - \frac{5}{9}\log_2\frac{5}{9} = 0.991\,bits$$

$$totalEntropy = \sum_i^n w_i\, entropy(S_i)$$

Standardization vs. Normalization

Normalization
(also called min-max scaling):

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Standardization:

$$x' = \frac{x - \mu(x)}{\sigma(x)}$$

x – original value
x' – new value

x – all values of the attribute; a vector
min(x) and max(x) – min and max values of the attribute (of the vector x)
$\mu(x)$ - mean value of the attribute
$\sigma(x)$ - standard deviation of the attribute

Distance measures for numeric attributes

- A, B – examples with attribute values $a_1, a_2, \ldots, a_n$ & $b_1, b_2, \ldots, b_n$
- E.g. A= [1, 3, 5], B=[1, 6, 9]

Euclidean distance (L2 norm) – most frequently used

$$D(A,B) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2}$$

D(A,B) = sqrt $((1-1)^2+(3-6)^2+(5-9)^2)=5$

Manhattan distance (L1 norm)

$$D(A,B) = |a_1 - b_1| + |a_2 - b_2| + \ldots + |a_n - b_n|$$

D(A,B)=|1-1|+|3-6|+|5-9|=7

Minkowski distance – generalization of Euclidean & Manhattan

$$D(A,B) = (|a_1 - b_1|^q + |a_2 - b_2|^q + \ldots + |a_n - b_n|^q)^{1/q}$$

q – positive integer

Weighted distance – each attribute is assigned a weight according to its importance (requires domain knowledge)

- Weighted Euclidean:

$$D(A,B) = \sqrt{w_1|a_1 - b_1|^2 + w_2|a_2 - b_2|^2 + \ldots + w_n|a_n - b_n|^2}$$

Similarity Measure

$$\cos(A, B) = \frac{A \bullet B}{\|A\| \|B\|}$$

$$\text{corr}(\mathbf{x}, \mathbf{y}) = \frac{\text{covar}(\mathbf{x}, \mathbf{y})}{\text{std}(\mathbf{x})\,\text{std}(\mathbf{y})}$$

where:

$$mean(\mathbf{x}) = \frac{\sum_{k=1}^{n} x_k}{n}$$

$$std(\mathbf{x}) = \sqrt{\frac{\sum_{k=1}^{n} (x_k - mean(\mathbf{x}))^2}{n-1}}$$

$$\text{co var}(\mathbf{x}, \mathbf{y}) = \frac{1}{n-1} \sum_{k=1}^{n} (x_k - mean(x))(y_k - mean(y))$$

- Range: [-1, 1]
  - -1: perfect negative correlation
  - +1: perfect positive correlation
  - 0: no correlation

Given: *N* examples with dimensionality *m* (i.e. *m* features)

Find: *m* new axes $Z_1,\ldots, Z_m$ orthogonal to each other such that

$Var(Z_1) > Var(Z_2)\ldots. > Var(Z_m)$

Z1,…, Zm are called principal components

The principal components are vectors that define a new coordinate system

They are ordered based on how much variance they capture

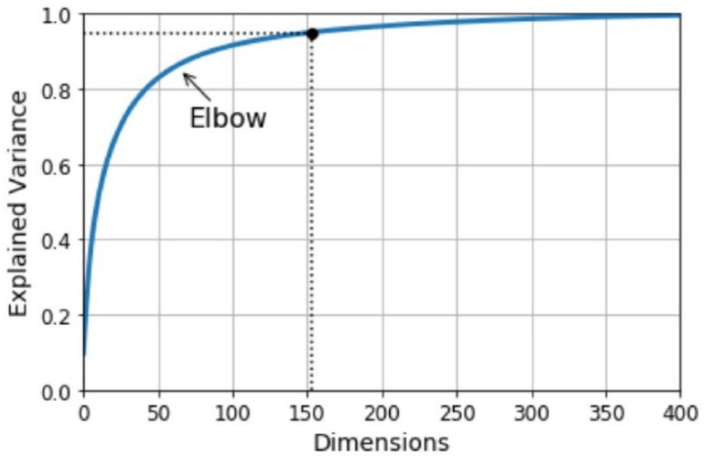
- The first axis goes in the direction of the highest variance in the data
- The second axis is orthogonal to the first one and goes in the direction of the second highest variance
- The third one is orthogonal to both the first and second and goes in the direction of the third highest variance, and so on

Method 1: Set min % of variance that should be preserved, e.g. 95%

- Choose *k* such that Z_1, Z_2,…Z_k capture 95% of the variance

Method 2: (Elbow method)

- Plot number of dimensions as a function of variance
- There is usually an elbow in the curve where the variance stops growing fast



- 95% variance is at 153 dimensions
- Elbow (subjective) - e.g. 100 dimensions

K-Nearest Neighbor is very sensitive to to the value of k

- rule of thumb: k ≤ sqrt(#training_examples)
- commercial packages typically use k=10

Using more nearest neighbors increases the robustness to noisy examples

K-Nearest Neighbor can be used not only for classification, but also for regression

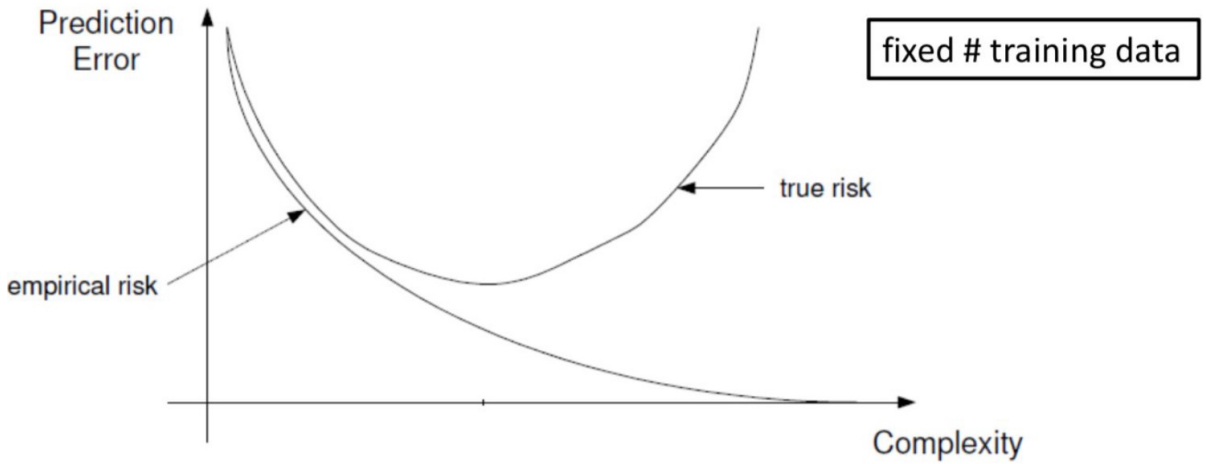
- The prediction will be the average value of the class values (numerical) of the k nearest neighbors

Step 1: Compute distance to other training records (e.g. euclidean distance).

Step 2: Identity k nearest neighbours.

Step 3: Use class labels of nearest neighbours to determine the class label of unknown records (using majority vote, weight the vote according to distance)

Overfitting and generalization



A FIRST APPROACH

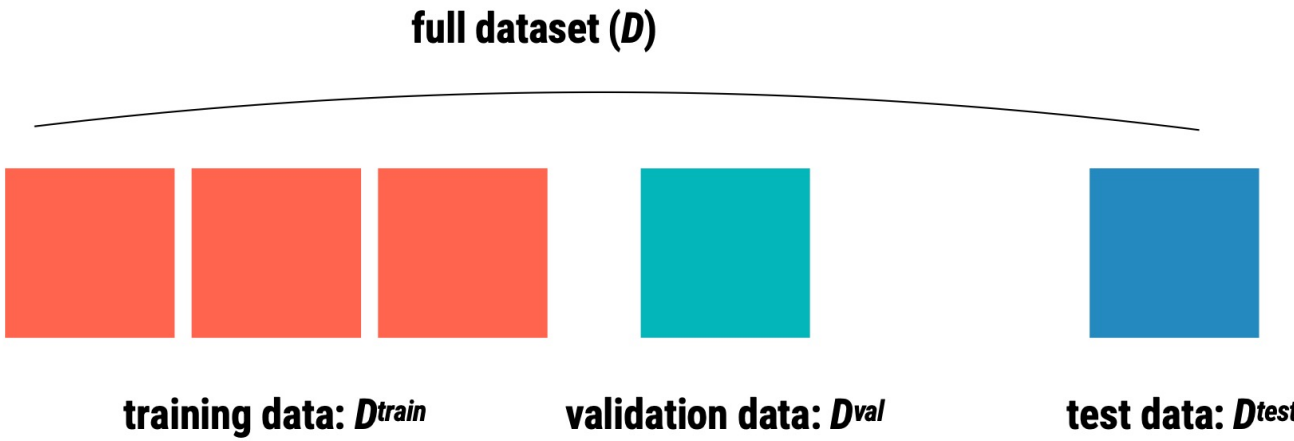
Tuning with a validation set

Training data: $D^{train} = \{(x_1, y_1), \ldots (x_n, y_n)\}$
- Used to learn the ERM predictor

Test data: $D^{test} = \{(x_1, y_1), \ldots (x_m, y_m)\}$
- Used to assess the performance of the ERM predictor

Validation data: $D^{val} = \{(x_1, y_1), \ldots (x_L, y_L)\}$
- Used to optimize hyperparameter(s)

training, validation, and test data should not overlap!



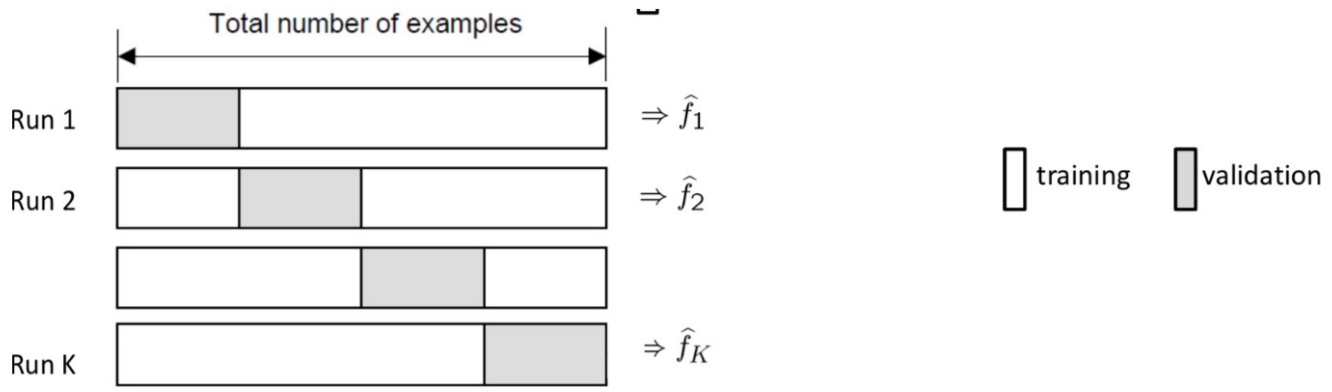
Cross-validation

Split the training data into K equal parts

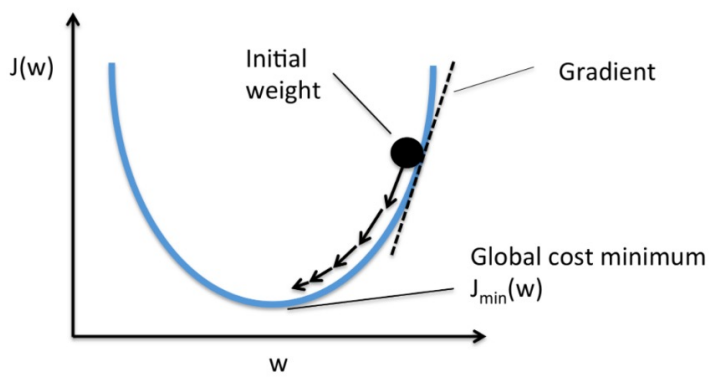
Use each part in turn as a validation dataset, with the remaining for training

Choose the hyperparameter such that the model performs the best (e.g., on average) across the K validation datasets

Special case: when K=n, leave-one-out (LOO) cross-validation



LR and GD



Cost function: Sum of Squared Errors

$$J(w) = \frac{1}{2}\sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

Step 1: Take partial derivative of the cost function with respect to each weight wj (gradient)

$$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

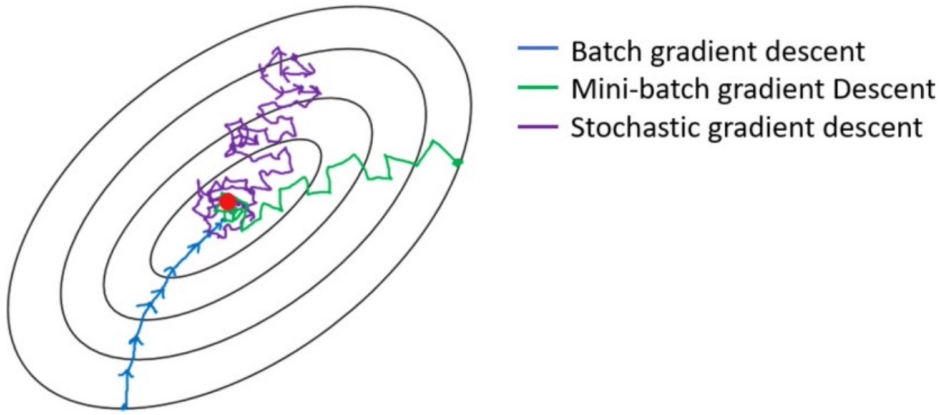
Step 2: Update weights by taking a step away from the gradient

$$w := w + \Delta w$$

Step 3: The weight change is defined as the negative gradient multiplied by the learning rate

$$\Delta w = -\eta \Delta J(w)$$

Gradient Descent



Objective we want to solve: $\min_{\mathbf{w}} f(\mathbf{w})$ where $f(\mathbf{w}) := \sum_{j=1}^{n} f_j(\mathbf{w})$

Gradient Descent Update: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f(\mathbf{w}_i)$

Stochastic Gradient Descent (SGD) Update: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f_j(\mathbf{w}_i)$

with j sampled at random

Mini-batch SGD Update: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f_{B_i}(\mathbf{w}_i)$

with mini-batch $\mathbf{B}_i \subseteq \{1,\dots n\}$ *sampled at random*

Mini-batch SGD

Pros

- In contrast to SGD: parallelizable
- Less computation than GD (might lead to faster overall convergence)
- Can tune computation vs. communication depending on batch size

Cons

- In terms of iterations: slower convergence than gradient descent
- Another parameter to tune (batch size)
- Still might be too much communication …

- Batch gradient descent:
  1. Go through the entire training set on every iteration. Cost long time on every iteration
  2. The cost function decreases on every single iteration

- Stochastic gradient descent:
  1. Cost function won't ever converge, it will always just kind of oscillate and wander around the region of the minimum. But it won't ever just head to the minimum and stay there.
  2. Lose almost all your speed up from vectorisation.

- Mini-batch gradient descent:
  1. Choose 1 < size < m can give you in practice the fastest learning. You do get a lot of vectorisation
  2. Cost function's decrease is in between of previous two.

Naïve Bayes

Week 4 Lecture Example

Assumption: the features are conditionally independent given the class label.

$p(\mathbf{x} \mid y = c, \theta) = \prod_{j=1}^{D} p(x_j \mid y = c, \theta_{jc})$

The form of class-conditional density depends on the type of each feature.

- Real-valued features

  $p(\mathbf{x} \mid y = c, \theta) = \prod_{j=1}^{D} \mathcal{N}(x_j \mid \mu_{jc}, \sigma_{jc}^2) = \prod_{j=1}^{D} \{\frac{1}{\sigma_{jc}\sqrt{2\pi}} \exp^{-\frac{(x_j - \mu_{jc})^2}{2\sigma_{jc}^2}}\}$

- Binary features ($x_j \in \{0, 1\}$)

  $p(\mathbf{x} \mid y = c, \theta) = \prod_{j=1}^{D} \mathrm{Ber}(x_j \mid p_{jc}) = \prod_{j=1}^{D} \{p_{jc}^{x_j}(1 - p_{jc})^{1-x_j}\}$

- Categorical features ($x_j \in \{1, \ldots, K\}$)

  $p(\mathbf{x} \mid y = c, \theta) = \prod_{j=1}^{D} \mathrm{Cat}(x_j \mid \mu_{jc})$ where μ_{jc} is a histogram over the possible values for x_j in class c.

sum rule $p(X) = \sum_Y p(X, Y)$ $\quad p(x) = \int p(x, y)\, \mathrm{d}y$

product rule $p(X, Y) = p(Y|X)p(X).$ $\quad p(x, y) = p(y|x)p(x).$

Bayes Rule

$$p(X, Y) = p(Y|X)p(X).$$

$$\downarrow$$

$$p(X, Y) = p(Y, X)$$

$$\downarrow$$

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}$$

$$\downarrow$$

$$p(X) = \sum_Y p(X|Y)p(Y).$$

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})} \qquad p(\mathcal{D}) = \int p(\mathcal{D}|\mathbf{w})p(\mathbf{w})\, \mathrm{d}\mathbf{w}.$$

$$\text{posterior} \propto \text{likelihood} \times \text{prior}$$

Decision Tree

$$T1 = H(S) = I(\frac{9}{14}, \frac{5}{14}) = 0.940\ bits$$

$$T2 = H(S \mid outlook) = \frac{5}{14} \cdot H(S_1) + \frac{4}{14} \cdot H(S_2) + \frac{5}{14} \cdot H(S_3)$$

$$H(S \mid outlook = sunny) = I(\frac{2}{5}, \frac{3}{5}) = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.971 bits$$

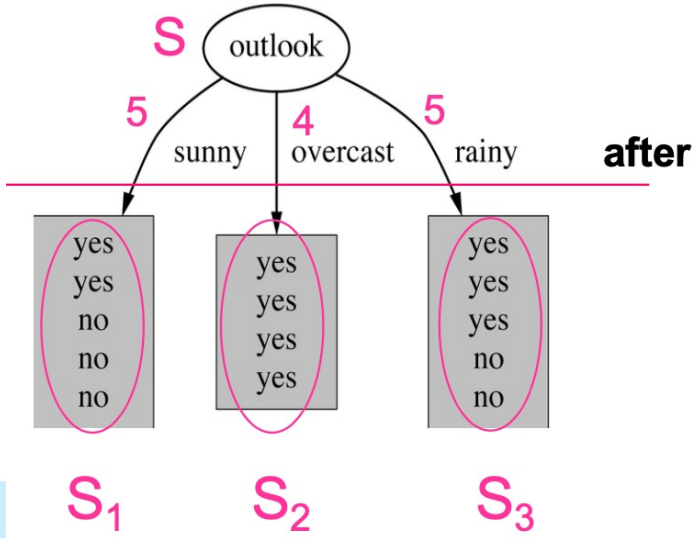
$$H(S \mid outlook = overcast) = I(\frac{4}{4}, \frac{0}{4}) = -\frac{4}{4}\log_2\frac{4}{4} - \frac{0}{4}\log_2\frac{0}{4} = 0 bits$$

$$H(S \mid outlook = rainy) = I(\frac{3}{5}, \frac{2}{5}) = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0.971 bits$$

$$H(S \mid outlook) = \frac{5}{14} \cdot 0.971 + \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot 0.971 = 0.693 bits$$

Gain(S|outlook)=H(S)-H(S|outlook)= 0.940-0.693=0.247 bits

Before splitting: 9 yes & 5 no



Gain(S|outlook)=H(S)-H(S|outlook)= 0.940-0.693=0.247 bits

- Similarly, the information gain for the other three attributes is:

Gain(S|temperature)=0.029 bits
Gain(S|humidity)=0.152 bits
Gain(S|windy)=0.048 bits

- => we select outlook as it has the highest information gain

Ensemble Method
Bagging vs. Boosting

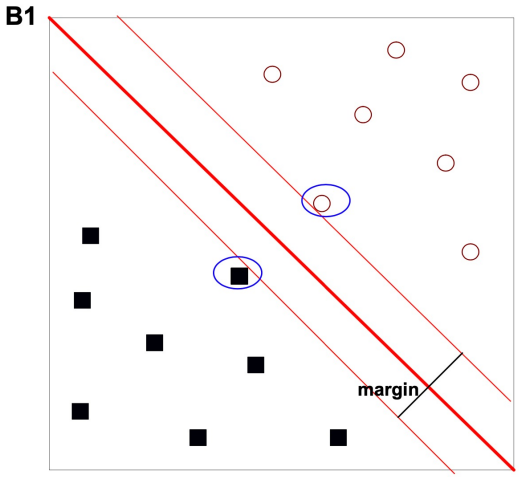
Similarities

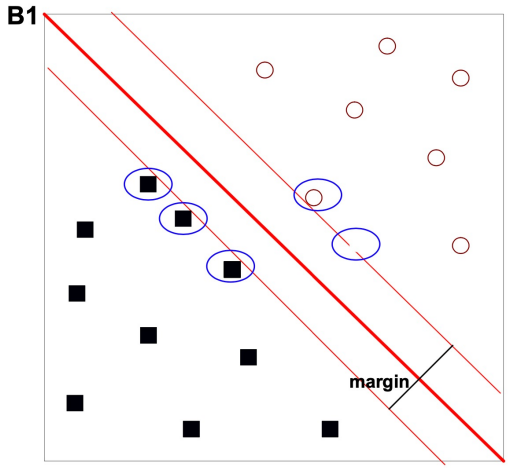
- Use voting (for classification) and averaging (for prediction) to combine the outputs of the individual learners
- Combine classifiers of the same type, typically trees – e.g. decision stumps or decision trees

Differences

- Creating base classifiers:
  - Bagging – separately
  - Boosting – iteratively – the new ones are encouraged to become experts for the misclassified examples by the previous base learners (complementary expertise)
- Combination method
  - Bagging – equal weighs to all base learners
  - Boosting (AdaBoost) – different weights based on the performance on training data

SVM – hard margin





The support vectors just touch the margin of the decision boundary

It is possible to have more than 1 support vector for each class

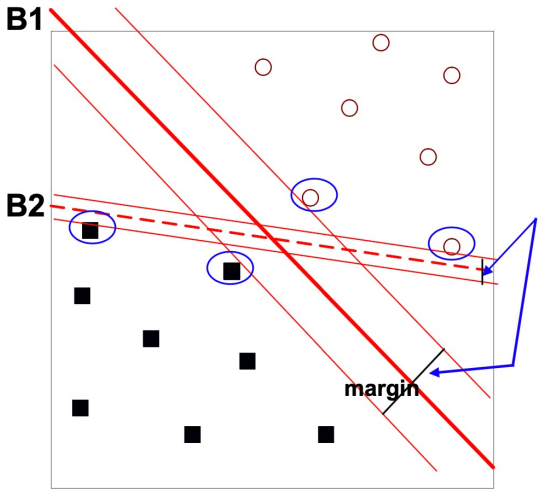
For our example: 5 support vectors, 3 for class square and 2 for class circle

Support vectors are the examples (data points) that lie closest to the decision boundary; they are circled

Margin – the separation between the boundary and the closest examples

The boundary is in the middle of the margin

Which hyperplane should we select - B1 or B2?
Which one is likely to classify more accurately new data**?**



- The hyperplane with the bigger margin, B1

SVM selects the maximum margin hyperplane

SVM – soft margin

We can modify our method to allow some misclassifications, i.e. by considering the trade-off between the margin width and the number of misclassifications

The optimisation problem formulation is similar but there is an additional parameter *C* in the definition of the optimization function

C is a hyper-parameter that allows for a trade-off between maximizing the margin and minimizing the training error

- Large *C*: more emphasis on minimizing the training error than maximizing the margin

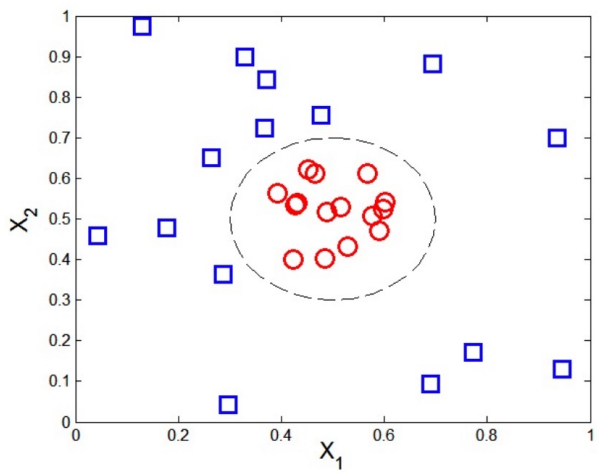
SVM – nonlinear

Transform the data from its original feature space to a new space where a linear boundary can be used to separate the data

If the transformation is non-linear and to a higher dimensional space, it is more likely than a linear decision boundary can be found in it

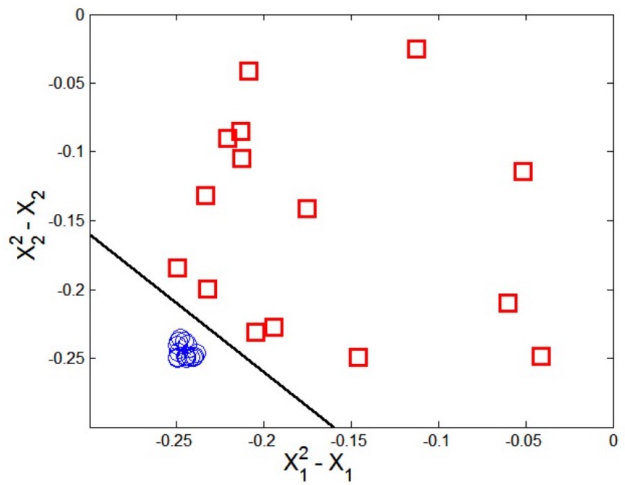
The learned linear decision boundary in the new feature space is mapped back to the original feature space, resulting in a non-linear decision boundary in the original space

- Non-linearly separable data in the original space



ϕ

- Becomes linearly separable in the new space



transformation from old to new space: $\phi = (x_1, x_2) \rightarrow ({x_1}^2 - x_1, {x_2}^2 - x_2)$

SVM – kernel trick

$$\Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j)$$

$$1)\ \mathbf{x}_i \xrightarrow{\Phi} \Phi(\mathbf{x}_i), \mathbf{x}_j \xrightarrow{\Phi} \Phi(\mathbf{x}_j)$$

$$2)\ \Phi(\mathbf{x}_i).\Phi(\mathbf{x}_j)$$

$$\Phi : (x_1, x_2) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$

$$\mathbf{u} \xrightarrow{\Phi} \Phi(\mathbf{u}), \mathbf{v} \xrightarrow{\Phi} \Phi(\mathbf{v})$$

$$\Phi(\mathbf{u})\cdot\Phi(\mathbf{v}) = (u_1^2, \sqrt{2}u_1u_2, u_2^2)\cdot(v_1^2, \sqrt{2}v_1v_2, v_2^2) =$$
$$= u_1^2v_1^2 + 2u_1u_2v_1v_2 + u_2^2v_2^2 = (u_1v_1)^2 + (u_2v_2)^2 + 2u_1u_2v_1v_2 =$$
$$= (u_1v_1 + u_2v_2)^2 = (\mathbf{u}\cdot\mathbf{v})^2$$

$$\Phi(\mathbf{u})\cdot\Phi(\mathbf{v}) = (\mathbf{u}\cdot\mathbf{v})^2$$

$$K(\mathbf{u},\mathbf{v}) = \Phi(\mathbf{u})\cdot\Phi(\mathbf{v}) = (\mathbf{u}\cdot\mathbf{v})^2$$

$$K(\mathbf{x},\mathbf{y}) = (\mathbf{x}\cdot\mathbf{y}+1)^p - \textit{polynomial kernel}$$

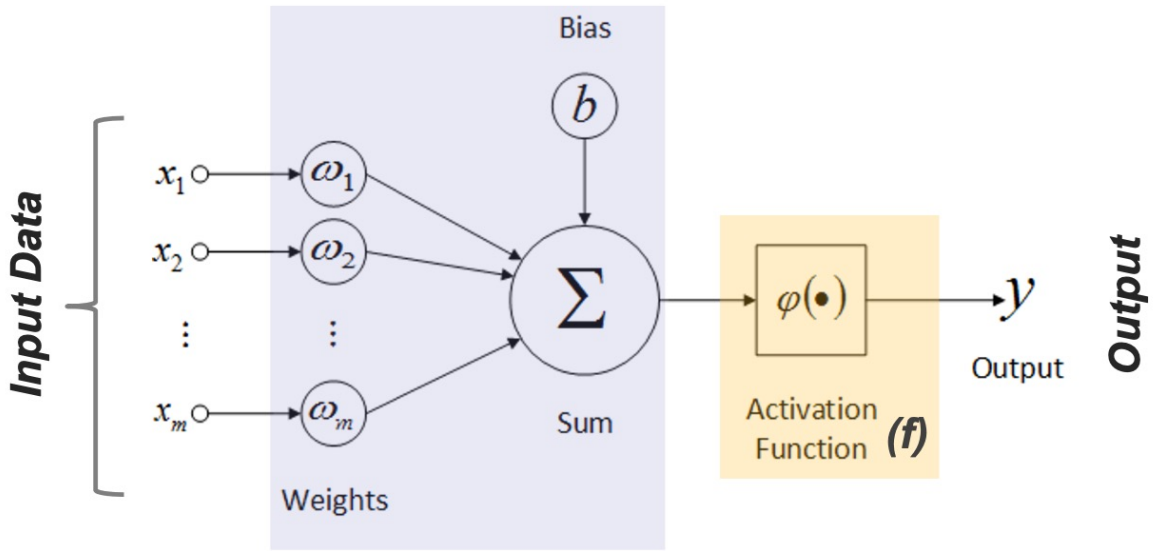
$$K(\mathbf{x},\mathbf{y}) = e^{-\frac{\|x-y\|^2}{2\sigma^2}} - RBF$$

$$K(\mathbf{x},\mathbf{y}) = \tanh(k\mathbf{x}\cdot\mathbf{y} - \theta) - \textit{tanget hyperbolic}$$

$$(\textit{satisfies Mercer's Th. only for some } k \textit{ and } \theta)$$

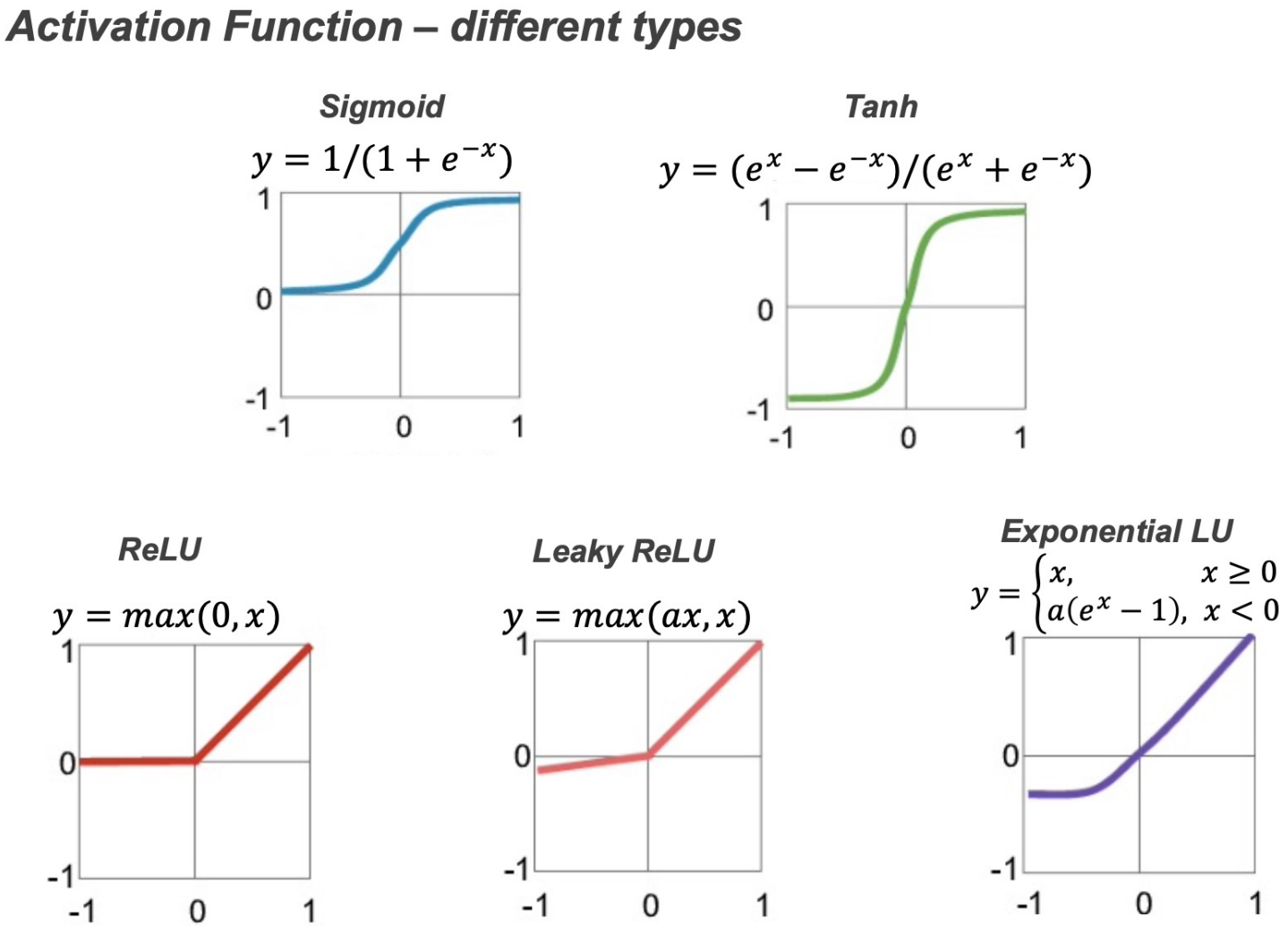
Deep Learning

MLP – feedforward and backpropagation



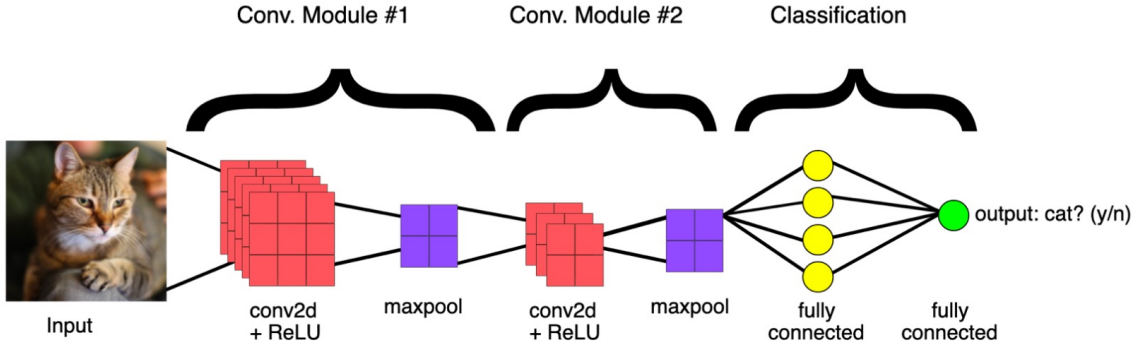
Deep Learning

Activation function



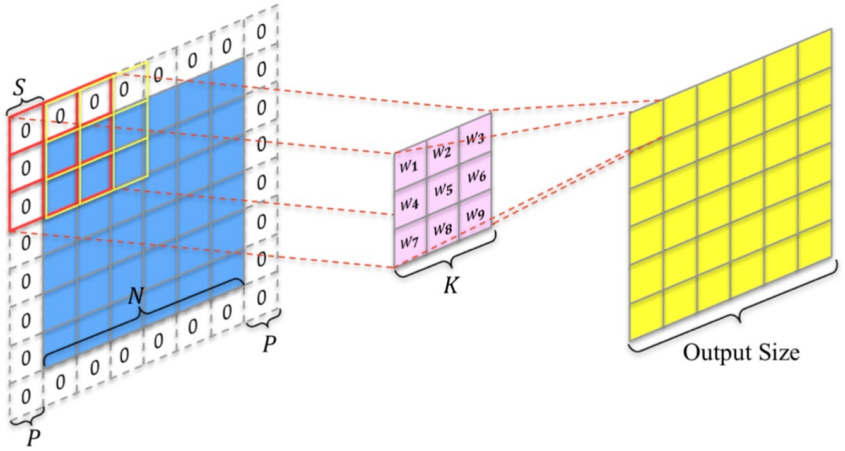
Deep Learning

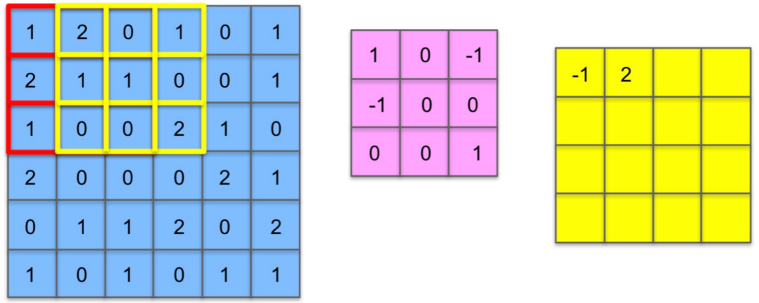
CNN



Basic CNN structure:
- Convolutional Layer
- Pooling
- Fully-connected Layer

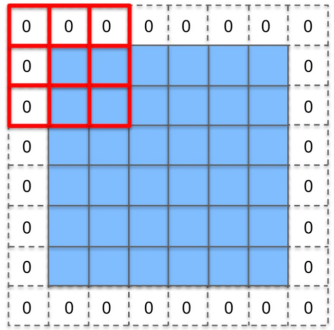
$$\text{Output Size} = \frac{N+2P-K}{S} + 1$$





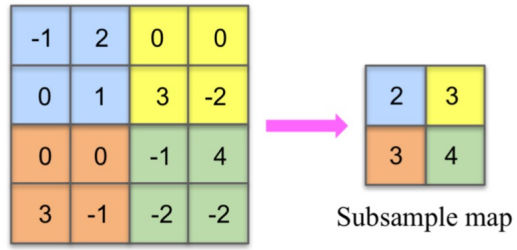
Stride = 1

Zero padding (pad = 1)



improves performance by keeping information at the borders

Max pooling
- Filter size: (2,2)
- Stride: (2,2)
- Pooling ops: max(·)

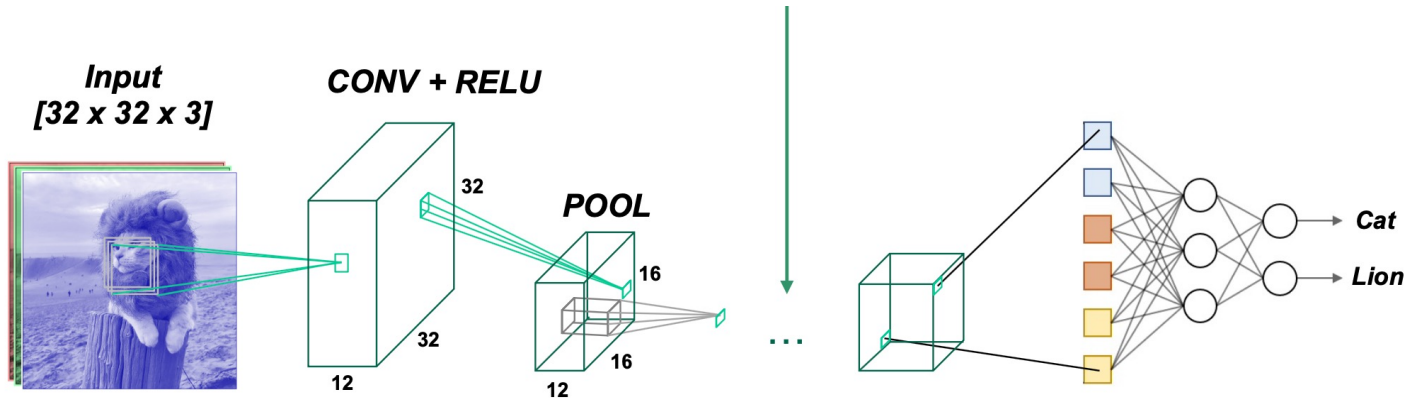


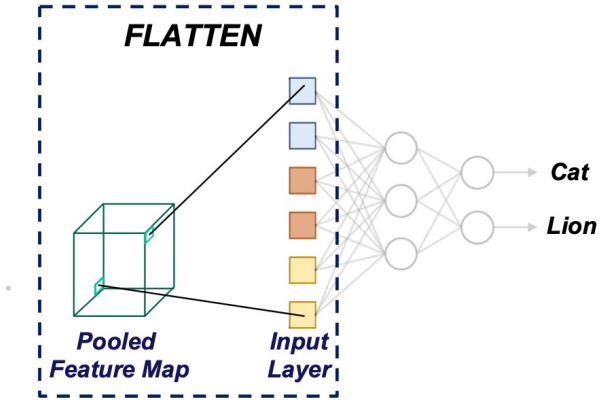
Deep Learning

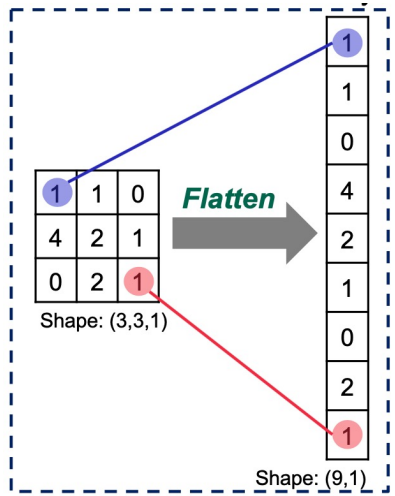
CNN

FLATTEN (Flattening) Layer

In between the convolutional layer and the fully connected layer, there is a '***Flatten***' layer. Flattening transforms a multi-dimensional matrix of features into a vector that can be fed into a fully connected neural network classifier.

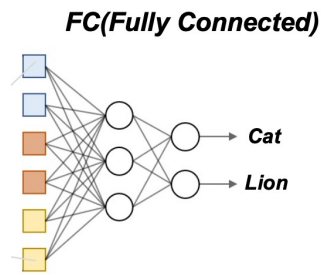






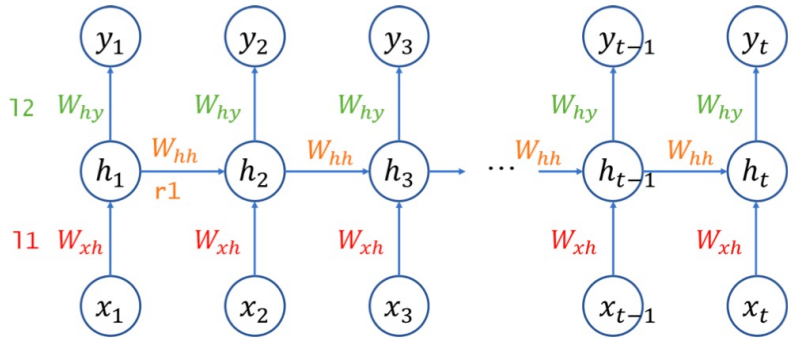
FC (Full Connected) layer

Compute the class scores, each of the 2 categories correspond to a class score, such as among the 2 categories of our dataset. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume



Deep Learning

RNN vs. LSTM

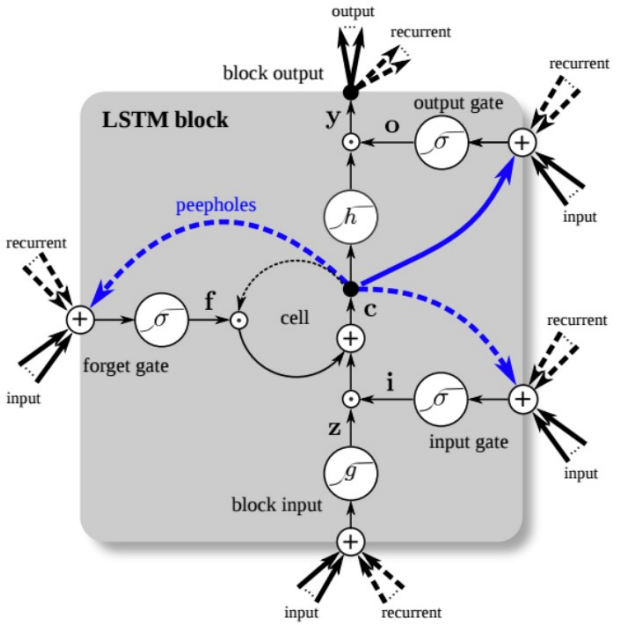


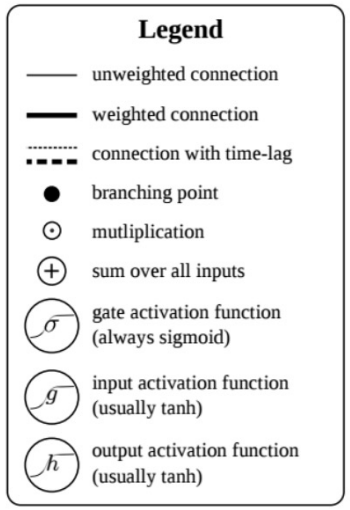
$$h_t = \tanh(\, l1(x_t) + r1(h_{t-1})\,)$$
$$y_t = l2(h_t)$$

The output of hidden layer are stored in memory. Memory can be considered as another input.

So information cycles through a loop.

1. current input
2. what it has leaned from the inputs it received previously.





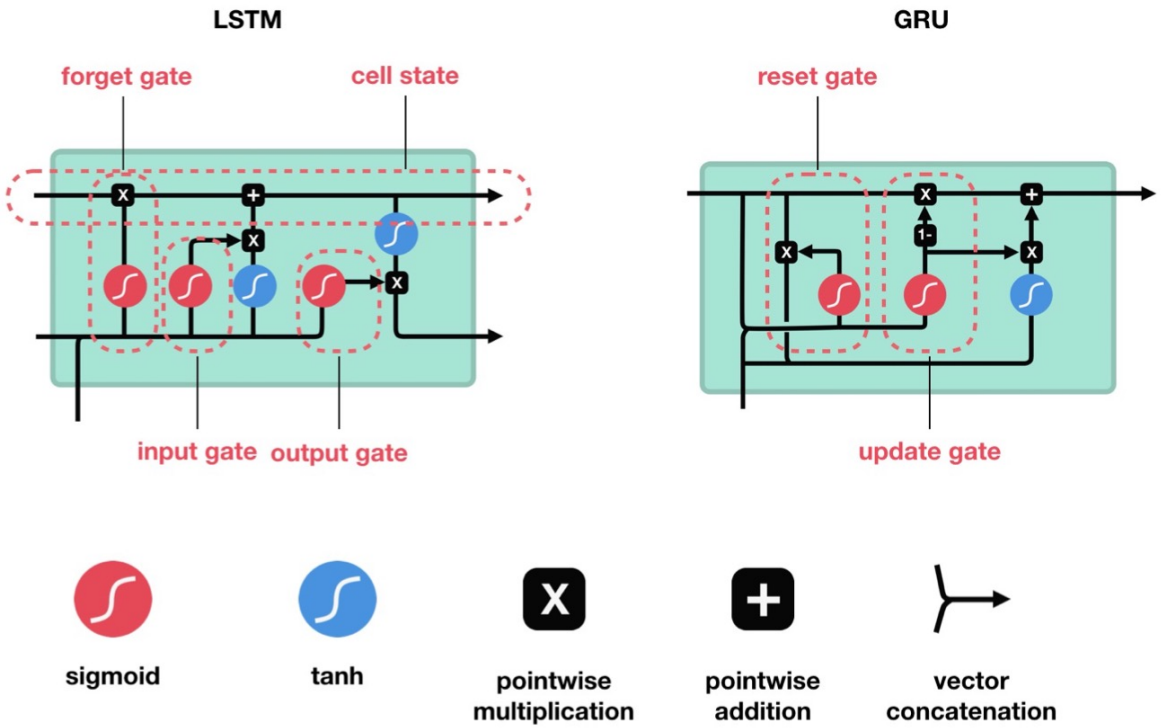
LSTM is an extension for RNN.

The memory in LSTM can be seen as a gated cell (store or delete based on the importance it assigns to the information).

Three gates:

1. input gate: whether or not let new input in
2. forget gate: delete information
3. output gate: output at the current time step

RNN vs. LSTM



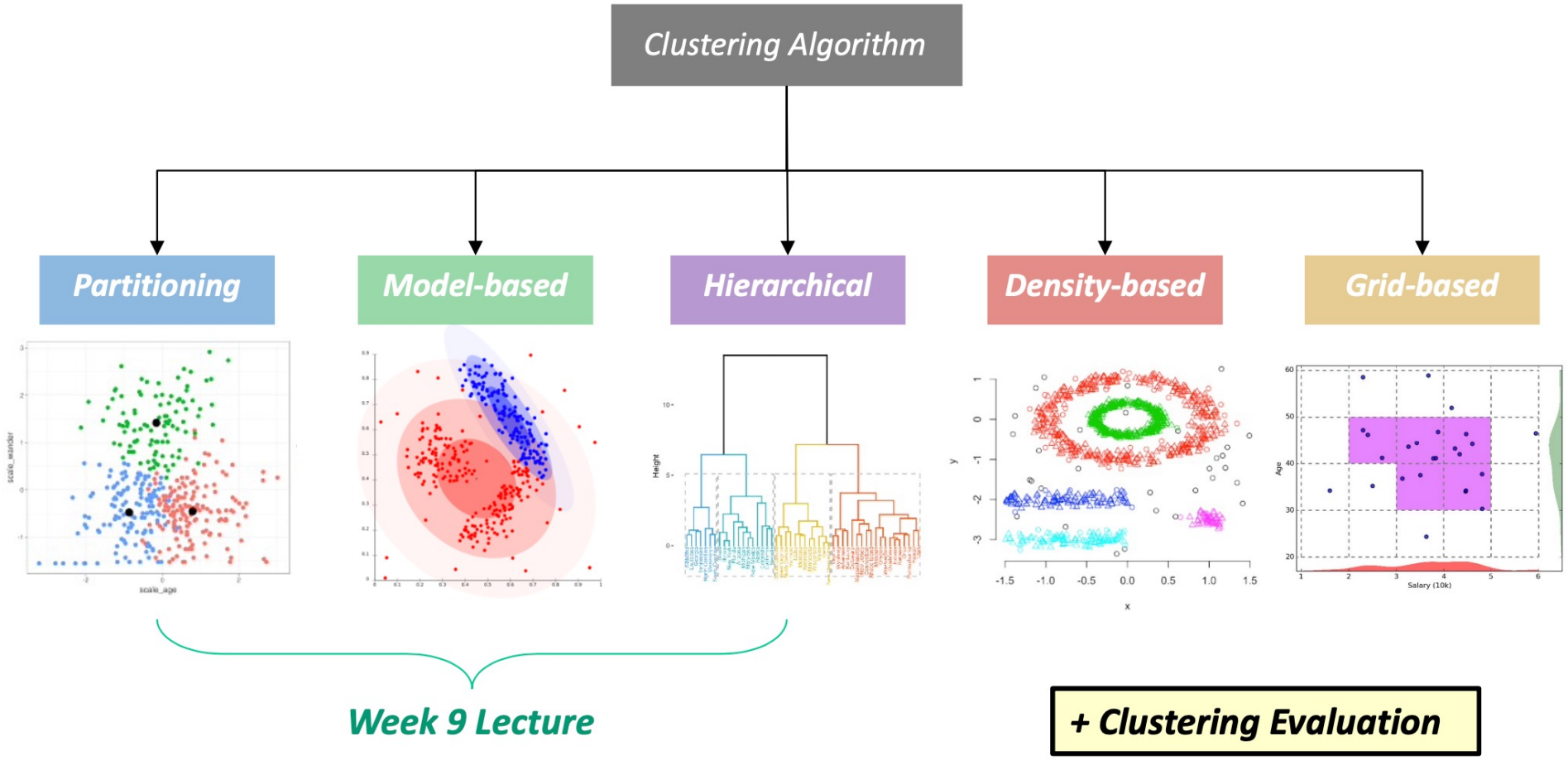
GRU first computes an **update gate** based on **current input word vector** and **hidden state**

Compute reset gate similarly but with different weights

- If reset gate unit is ~0, then this ignores previous memory and only stores the new word information

Final memory at time step combines current and previous time steps

Clustering



Clustering evaluation week 10 slides

HMM: The three main questions

Evaluation

- *What is the **probability** of the observed sequence?* Forward algorithm

Decoding

- *Given a model and a sequence of observations, what is the most likely state observation?*

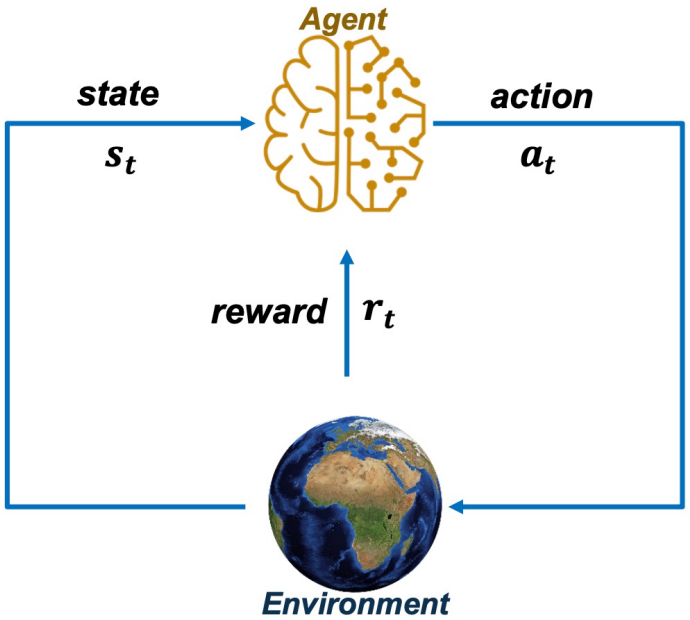
Viterbi algorithm

Learning

- *Under what parameterization is the observed sequence most probable?*

EM algorithm

Reinforcement Learning



At each step t the agent:
- *Executes action* a_t
- *Receives state* s_t
- *Receives scalar reward* r_t

The environment:
- *Receives action* a_t
- *Emits state* s_{t+1}
- *Emits scalar reward* r_{t+1}

t increments at env. step

$(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

$\mathcal{S}$: set of possible states
$\mathcal{A}$: set of possible actions
$\mathcal{R}$: distribution of reward given (state, action) pair
$\mathbb{P}$: **transition probability**
i.e. distribution over next state given (state, action) pair

γ : discount factor

Our governing equation looks like this:

$$Q_{i+1}(state, action) = (1 - \alpha)Q_i(state, action) + \alpha(reward + \gamma \max_{a \in actions}[Q_i(nextstate, a)])$$

We have three hyperparameters:

α - a learning rate, how quickly do we want to change Q from step to step

γ - a discount factor, assigns an importance to the effect of future steps on evolution of Q

During training quality values will be learned for each action in each state. Should the system always make the action with the highest quality at each state? Consider whether this would encourage a form of overfitting, or sticking in local minima. We'll introduce a third parameter in training to balance between taking the current highest quality move (exploitation) or a different action (exploration).

ϵ - a weighting between exploration and exploitation