ML Notes

Reinforcement learning (**RL**) is an area of machine learning concerned with how software agents ought **to** take actions **in** an environment so as **to** maximize some notion of cumulative reward. **Reinforcement learning** is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

Reinforcement Learning is a type of **Machine Learning**, and thereby also a branch of Artificial Intelligence. It allows machines and software agents to automatically determine the ideal behaviour within a specific context, in order to maximize its performance.

Supervised: Training data contain values for variable that model has to predict

- Classification: categorial variable
- Regression: continuous variable
- Ordinal regression, finite, ordered set of values
- Rankings: ordering of elements
- Structured prediction: sequence, tree, graph, …
- Recommendation: Item-by-user matrix

- The loss function measures the goodness-of-fit a model has to the observed training data.
  - How bad is it if the model predicts value $y\boldsymbol{\theta}\ \mathbf{x}i$ when the true value of the target variable is yi?
    - **Zero-One Loss – [1 if match else 0]**
    - **Quadric Error – Regression, distance bw target and prediction**
  - **Empirical risk –** Overall loss from Training data

- The regularization function measures, whether the model is likely according to our prior knowledge. The higher the regularization term is for a model, the less likely the model is.
  - L0 - Count of the non-zero weights – Difficult minimize
  - L1 Lasso - Sum of the attribute weights
    - It set the weights to 0 for irrelevant attributes

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

  - L2 Ridge (Squared Euclidean norm) - Sum of the squared attribute weights. Used to avoid Overfiiting.

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

        - It punish and reduce the weight of the irrelevant attributes
    - If lambda is low, overfit. If lambda is high, can cause the under fitting.
- **Risk**: expected loss under distribution p ($\mathbf{x}$, y). Training & Test data set. N-fold cross validation
- The optimization criterion is a (weighted) sum of the losses for the training data and the regularizer.
- We seek the model that minimizes the optimization criterion.

TF-IDF – it shows the words that occurs once have higher TD IDF Score than those that occur more than once, not just in the same documents. But across all the documents. So it foces the **Rare words** than frequent one.

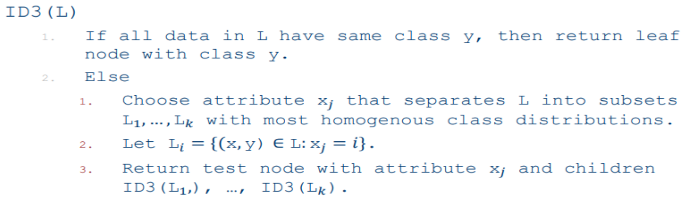
TF= the number of times each term occurs in each document
IDF = (total number of documents) / (the number of documents containing the term)

TF - IDF = Multiplying TF & IDF

- Min/Max normalization: $x^{new} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}(x_{\max}^{new} - x_{\min}^{new}) + x_{\min}^{new}$
- Z-Score normalisierung: $x^{new} = \frac{x - \mu_x}{\sigma_x}$
- Decimal scaling: $x^{new} = |x| \cdot 10^a \quad a = \max_x\{i \in \mathbb{Z} \mid |x| \cdot 10^i < 1\}$
- Logarithmic scaling: $x^{new} = \log_a x$

Decision trees

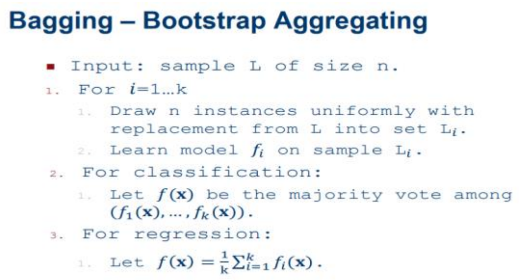
Good design the small tree. Greedy algorithm that finds a small tree (instead of the smallest tree) but is polynomial in the number of attributes



```
ID3(L)
  1. If all data in L have same class y, then return leaf
     node with class y.
  2. Else
       1. Choose attribute $x_j$ that separates L into subsets
          $L_1, \ldots, L_k$ with most homogenous class distributions.
       2. Let $L_i = \{(x,y) \in L: x_j = i\}$.
       3. Return test node with attribute $x_j$ and children
          ID3($L_1$), …, ID3($L_k$).
```

- Using Information Gain to find best homogenous (Information Gain based on entropy which is splits based on bits size).

- Classification with categorical features: ID3 – Recursive algorithm
  - Use information gain, gain ratio, or Gini index
  - ID3 constructs a branch for every value of the selected attribute . Only for **discrete independent variables**.
- Classification with continuous features: C4.5
  - For **continuous independent** variables.
  - Algorithm designed to choose the classes based on the less than or greater than
- Pruning (**regularization**)
  - Remove test nodes whose leaves have less than τ instances.
  - Collect in new leaf node that is labeled with the majority class
- Regression: CART
  - Same algorithm as above, but mean values are find for final leaf nodes
- Model trees
  - Same algorithm as CART, but at the end, linear regression applied on leaf nodes instead of the mean.
- Bagging
  - Sample random subsets of training instances
  - The bootstrapping procedure draws k instances from a set of n instances with replacement.



Bagging – Bootstrap Aggregating

```
▪ Input: sample L of size n.
1. For $i$=1…k
    1. Draw n instances uniformly with
       replacement from L into set $L_i$.
    2. Learn model $f_i$ on sample $L_i$.
2. For classification:
    1. Let $f(\mathbf{x})$ be the majority vote among
       $(f_1(\mathbf{x}), \dots, f_k(\mathbf{x}))$.
3. For regression:
    1. Let $f(\mathbf{x}) = \frac{1}{k}\sum_{i=1}^{k} f_i(\mathbf{x})$.
```

- Random forests
  - Sample random subsets of training instances **and random subsets of features**

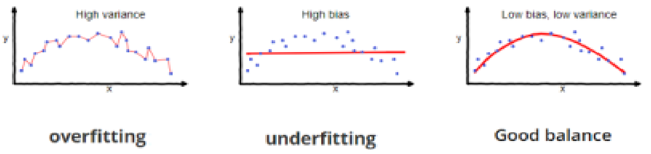
- Boosting
  - Iteratively draw subsets of the training instances such that instances that are misclassified by the current ensemble receive a higher weight.

Linear Classification Models

- Decision function: $f\boldsymbol{\theta}\ \mathbf{x} = \mathbf{x}^{\mathsf{T}}\boldsymbol{\theta} + \theta^0$
- Binary classifier, $y \in \{+1, -1\}$
- Gradient descent – first-order iterative optimization algorithm. Finding the minimum loss using different step functions
- Stochastic Gradient descent – Same as GD. But here, we use subset of training set to compute the best minimize function for good performance.

- Parallel Stochastic Gradient - distribute data over multiple nodes; perform computation in parallel on these nodes.

Loss functions for classification:



overfitting underfitting Good balance

ıss is not convex
lt to minimize!

- Zero-one loss:

  $\ell_{0/1}(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = \begin{cases} 1 & - \\ 0 & - \end{cases}$

  si

  $\mathrm{sign}(f_{\boldsymbol{\theta}}(\mathbf{x}_i)) = y_i$

- Logistic loss:

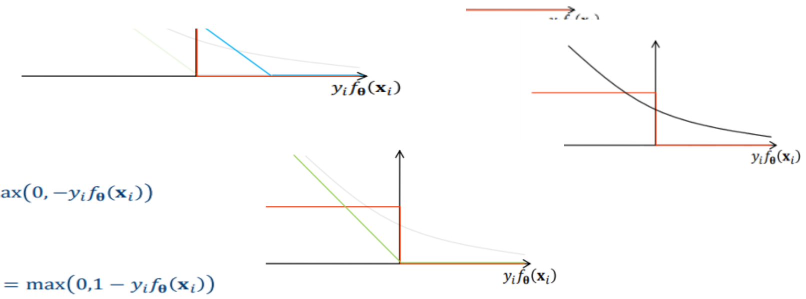
  $\ell_{log}(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = \log(1 + e^{-y_i f_{\boldsymbol{\theta}}(\mathbf{x}_i)})$

- Perceptron loss:

  $\ell_p(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = \begin{cases} -y_i f_{\boldsymbol{\theta}}(\mathbf{x}_i) & -y_i f_{\boldsymbol{\theta}}(\mathbf{x}_i) > 0 \\ 0 & -y_i f_{\boldsymbol{\theta}}(\mathbf{x}_i) \leq 0 \end{cases} = \max(0, -y_i f_{\boldsymbol{\theta}}(\mathbf{x}_i))$

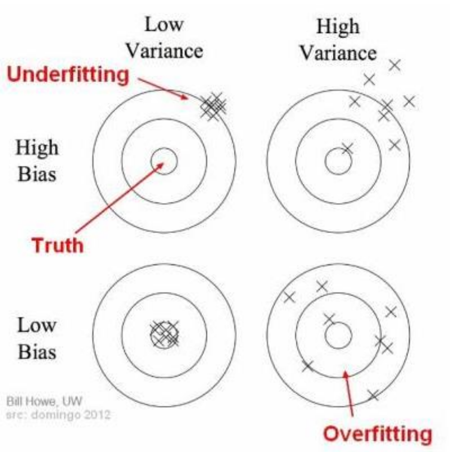
- Hinge loss:

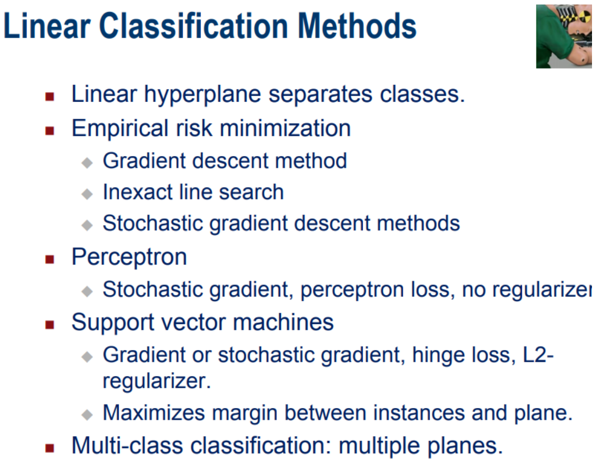
  $\ell_h(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = \begin{cases} 1 - y_i f_{\boldsymbol{\theta}}(\mathbf{x}_i) & 1 - y_i f_{\boldsymbol{\theta}}(\mathbf{x}_i) > 0 \\ 0 & 1 - y_i f_{\boldsymbol{\theta}}(\mathbf{x}_i) \leq 0 \end{cases} = \max(0{,}1 - y_i f_{\boldsymbol{\theta}}(\mathbf{x}_i))$



- **Perceptron algorithm** minimizes the sum of the perceptron loss over all samples. No Regularizer. Separates a hyper plane (converge only if hyper plane is exist)
- **SVM**
  - Class $y \in -1, +1$.
  - Hinge loss is used.
  - $\boldsymbol{L(\theta)}$ can be minimized using (stochastic) gradient descent method
  - L2 Norm is very popular.
  - Using a Margins (Cost or lambda)
    - The regularization parameter (lambda) serves as a degree of importance that is given to miss-classifications. Then when lambda tends to infinite the solution tends to the hard-margin (allow no miss-classification). When lambda tends to 0 (without being 0) the more the miss-classifications are allowed.
  - Gamma – Gaussian Radial Function
    - A small gamma means a Gaussian with a large variance so the influence of x_j is more, i.e. if x_j is a support vector; a small gamma implies the class of this support vector will have influence on deciding the class of the vector x_i even if the distance between them is large. If gamma is large, then variance is small implying the support vector does not have wide-spread influence. Technically speaking, large gamma leads to high bias and low variance (under fitting) models, and vice-versa.

- Multi-class classification - Each class y has a separate function $f\boldsymbol{\theta}\ \mathbf{x}, y$ that is used to predict how likely y is given $\mathbf{x}$.





Linear Classification Methods

- Linear hyperplane separates classes.
- Empirical risk minimization
  - Gradient descent method
  - Inexact line search
  - Stochastic gradient descent methods
- Perceptron
  - Stochastic gradient, perceptron loss, no regularizer
- Support vector machines
  - Gradient or stochastic gradient, hinge loss, L2-regularizer.
  - Maximizes margin between instances and plane.
- Multi-class classification: multiple planes.

Linear Regression Models

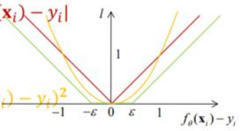
Loss Functions for Regression

- Absolute loss:

$$\ell_{\text{abs}}(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = |f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i|$$

- Squared loss:

$$\ell_2(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = (f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2$$



- ε-insensitive loss:

$$\ell_\varepsilon(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = \begin{cases} |f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i| - \varepsilon & |f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i| - \varepsilon > 0 \\ 0 & |f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i| - \varepsilon \le 0 \end{cases}$$

Regularizer for Regression

- L1 regularization:

$$\Omega_1(\boldsymbol{\theta}) \propto \|\boldsymbol{\theta}\|_1 = \sum_{j=1}^{m} |\theta_j|$$

- L2 regularization:

$$\Omega_2(\boldsymbol{\theta}) \propto \|\boldsymbol{\theta}\|_2^2 = \sum_{j=1}^{m} {\theta_j}^2$$

Special Cases

- Lasso: squared loss + L1 regularization

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} \ell_2(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) + \lambda\|\boldsymbol{\theta}\|_1$$

- Ridge regression: squared loss + L2 regularization

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} \ell_2(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) + \lambda\|\boldsymbol{\theta}\|_2^2$$

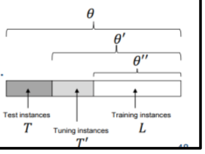
- Elastic net: squared loss, L1 + L2 regularization

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} \ell_2(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) + \lambda\|\boldsymbol{\theta}\|_2^2 + \lambda'\|\boldsymbol{\theta}\|_1$$

Evaluation

Triple Cross Validation

- Iterate over all values of the hyperparameters λ (grid search)
  - Train model $f_{\theta''}^{\lambda}$ on L.
  - Evaluate $f_{\theta''}^{\lambda}$on T' by calculating $\hat{R}_{T'}(f_{\theta''}^{\lambda})$
- Use hyperparameter λ^* that gave lowest $\hat{R}_{T'}(f_{\theta''}^{\lambda^*})$.
- Train model $f_{\theta'}^{\lambda^*}$ on $L \cup T'$.
- Determine $\hat{R}_T(\theta')$.
- Train model $f_{\theta}^{\lambda^*}$ on $L \cup T' \cup T$.
- Return model $f_{\theta}^{\lambda^*}$ and estimate $\hat{R}_T(f_{\theta'}^{\lambda^*})$.



True-positive rate (recall): $r_{TP} = \frac{n_{TP}}{n_{TP}+n_{FN}}$

- Rate of true positives among all positive instances
- Answers: “How many of the positive instances does the classifier detect?”

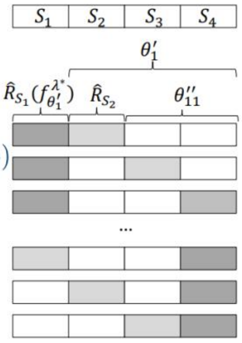
False-positive rate: $r_{FP} = \frac{n_{FP}}{n_{FP}+n_{TN}}$

- Rate of false positives among all instances that are really negatives.
- Answers: “How many of the negative instances does the classifier misclassify as positive?”

- True positives:
  - Patient has disease ($y_i = +1$), classifier recognizes ($y_\theta(\mathbf{x}_i) = +1$)
- False positives:
  - Patient is healthy ($y_i = -1$), but classifier diagnoses disease ($y_\theta(\mathbf{x}_i) = +1$)
- True negatives:
  - Patient is healthy ($y_i = -1$), classifier recognizes ($y_\theta(\mathbf{x}_i) = -1$)
- False negatives:
  - Patient has disease ($y_i = +1$), classifier misses ($y_\theta(\mathbf{x}_i) = -1$)

Nested Cross Validation

- For $i = 1 \ldots k$
  - Iterate over values λ
    - For $j = 1 \ldots k \setminus i$
      - Train $f_{\theta_{ij}}^{\lambda}$ on $S \setminus S_i \setminus S_j$
      - Determine $\hat{R}_{S_j}\left(f_{\theta_{ij}}^{\lambda}\right)$
    - Average $\hat{R}_{S_j}$ to determine $\hat{R}_{S\setminus S_i}\left(f_{\theta_i'}^{\lambda}\right)$
  - Choose λ_i^* that minimizes $\hat{R}_{S\setminus S_i}\left(f_{\theta_i'}^{\lambda}\right)$
  - Train $f_{\theta_i}^{\lambda_i^*}$ on $S \setminus S_i$
  - Determine $\hat{R}_{S_i}\left(f_{\theta_i'}^{\lambda_i^*}\right)$
- Average $\hat{R}_{S_i}\left(f_{\theta_i}^{\lambda_i^*}\right)$ to determine $\hat{R}_S(f_{\theta^*}^{\lambda^*})$
- Determine λ^* by averaging λ_i^*
- Train $f_\theta^{\lambda^*}$ on S
- Return $f_\theta^{\lambda^*}$ and $\hat{R}_S(f_{\theta^*}^{\lambda^*})$



50

- Precision: $P = \frac{n_{TP}}{n_{TP}+n_{FP}}$
  - Rate of true positives among all instances that are classified as positives
  - Answers: "How accurate is classifier when it says +1?"
- Recall: $R = \frac{n_{TP}}{n_{TP}+n_{FN}}$
  - Rate of true positives among all positive instances
  - Answers: "How many of the positive instances does the classifier detect?"

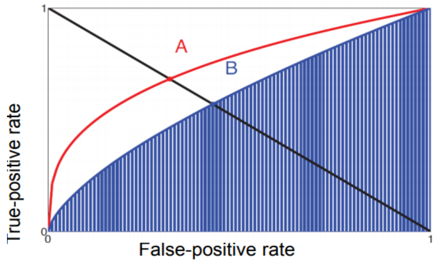
F Measures

- F_α measures combine precision and recall values into single value:

$$F_\alpha = \frac{n_{TP}}{\alpha(n_{TP}+n_{FP}) + (1-\alpha)(n_{TP}+n_{FN})}$$

Summary

- Risk: expected loss over input distribution $p(\mathbf{x}, y)$.
- Empirical risk: estimate of risk on data.
- Evaluation protocols:
  - Hold-out testing: good for large samples
  - K-fold Cross Validation: good for small samples.
- Model selection: tune model hyperparameters.
  - Triple cross validation: good for large samples.
  - Nested cross validation: good for small samples.
- Precision-recall curves and ROC curves characterize decision function. Each point on curve is classifier for some threshold θ_0.

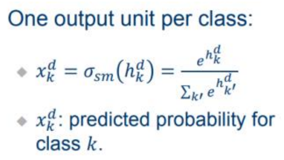


$$r_{TP} = \frac{n_{TP}}{n_{TP}+n_{FN}}$$

$$r_{FP} = \frac{n_{FP}}{n_{FP}+n_{TN}}$$

Neural Network :

- ***Deep Learning***
  - **Weights & Biases**: Numerical values - A weight is applied to input of each of the neuron to compute an output. Biases are also numerical values which are added once weights are applied to inputs. Hence weights and biases make neural networks self-learning algorithms.
  - **Activation Function** Essentially activation functions smooth or normalise the output before it is passed on to the next or previous neurons When neurons compute weighted sum of inputs, they are passed to the activation function which checks if the computed value is above the required threshold.
    - **Sigmoid:** 1/1 + exp(x) which produces a S-shaped curve. Although it is non-linear in nature but it does not capture slight changes within inputs hence variations in inputs will yield similar results.
    - **Hyperbolic Tangent Functions (Tanh)**: (1- (exp(-2(x))/(1 + exp(-2x)). It is a superior function when compared to Sigmoid. However it does not capture relationships better and is slower at converging.
    - **Rectified Linear Units (ReLu)**: This function converges faster, optimises and produces the objective value quicker. It is by far the most popular activation function used within the hidden layers. prevents the gradient from vanishing for deep networks.
    - **Softmax** (cross-entropy loss): Used in output layer because it reduces dimensions and can represent categorical distribution.
    - **Linear Activation**



One output unit per class:

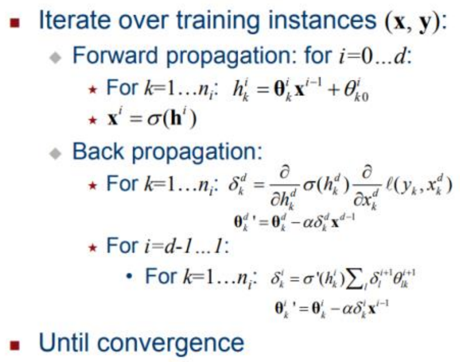
- $x_k^d = \sigma_{sm}(h_k^d) = \frac{e^{h_k^d}}{\sum_{k'} e^{h_{k'}^d}}$
- x_k^d: predicted probability for class k.

  - **How it works?**
    - For each neuron in a layer, multiply input to weight.
    - Then for each layer, sum all input x weights of neurons together.
    - Finally, apply activation function on the output to compute new output.

- ***Feed-forward networks***
  - These models are called **feed forward** because information flows through the function being evaluated from **x**, through the intermediate computations used to define f, and finally to the output y (i.e. do not form cycles (like in recurrent nets)). There are no feedback connections in which outputs of the model are fed back into itself.
    - **Cost function:** cost function like the quadratic cost it turns out to be easy to figure out how to make small changes in the weights and biases so as to get an improvement in the cost. That’s why we focus first on minimizing the quadratic cost, and only after that will we examine the classification accuracy

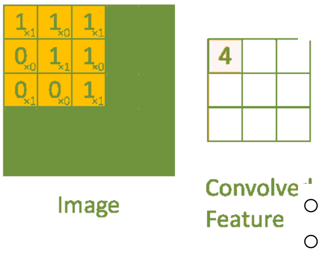
$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

    - **(Stochastic) Gradient descent & loss function for learnings**
      - Loss functions to become non-convex since using iterative.

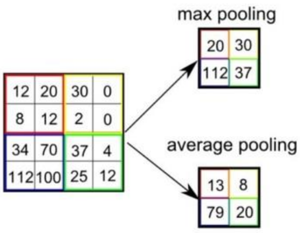


- Iterate over training instances $(\mathbf{x}, \mathbf{y})$:
  - Forward propagation: for i=0...d:
    - For k=1…n_i: $h_k^i = \boldsymbol{\theta}_k^i \mathbf{x}^{i-1} + \theta_{k0}^i$
    - $\mathbf{x}^i = \sigma(\mathbf{h}^i)$
  - Back propagation:
    - For k=1…n_i: $\delta_k^d = \frac{\partial}{\partial h_k^d}\sigma(h_k^d)\frac{\partial}{\partial x_k^d}\ell(y_k, x_k^d)$
 $\boldsymbol{\theta}_k^d{}' = \boldsymbol{\theta}_k^d - \alpha\delta_k^d \mathbf{x}^{d-1}$
    - For i=d-1…1:
      - For k=1…n_i: $\delta_k^i = \sigma'(h_k^i)\sum_l \delta_l^{i+1}\theta_{lk}^{i+1}$
 $\boldsymbol{\theta}_k^i{}' = \boldsymbol{\theta}_k^i - \alpha\delta_k^i \mathbf{x}^{i-1}$
- Until convergence

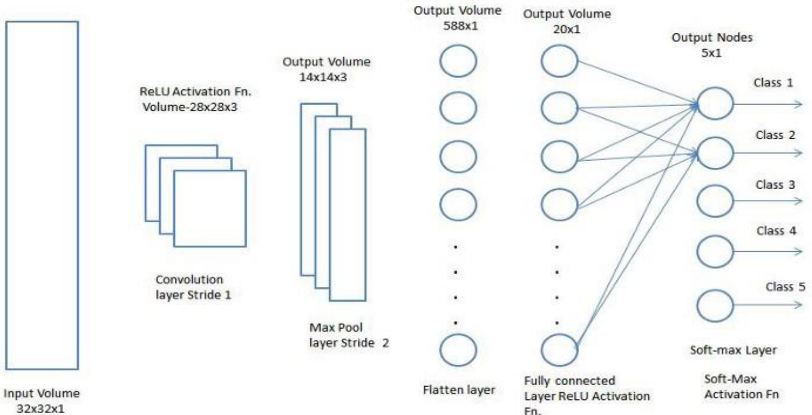
- Gradient descent can be viewed as a way of taking small steps in the direction which does the most to immediately decrease C.
- Many challenges in training gradient based learning - very large this can take a long time. So idea called ***stochastic gradient descent*** (Mini-Batches : small number m of randomly chosen training inputs)
- Stochastic gradient descent applied to non-convex loss functions has **no such convergence guarantee**, and is sensitive to the values of the initial parameters.
  - Local minima can still be arbitrarily good.
  - Many local minima can be equally good.
  - Supervised learning often works with hundreds of layers and millions of training instances.
- For feed forward neural networks, it is important to **initialize all weights** to small random values. The biases may be initialized to zero or to small positive value



  - **Back Propagation:** To **solve is to update weight and biases** such that our cost function can be minimised. **Back propagation** is a training algorithm consisting of 2 steps:
    - Feed forward the values
    - Calculate the error and propagate it back to the earlier layers. So to be precise, forward-propagation is part of the back propagation algorithm but comes before back-propagating.
  - Regularizer - Weight Decay (L2 Regularizer), Dropouts
  - Input: Normalized (0 to 1) as it is difficult to adjust with large absolute value of input vector with gradient.
  - Initialization weight – Not 0. Use Uniform or Normal (have lower variance). Bcus weights in all neural must be similar
- Both forward and backward propagation can be made much faster **by parallel computation**
  - Forward- and backward-propagation can be written as matrix multiplications**.**
  - Columns of the weight matrix can be processed in parallel.
- **Convolutional Layers** - Image & Video recognition, Image Analysis & Classification, Media Recreation, Recommendation Systems, Natural Language Processing



  - Output of Conv. Feature increase the size as it is added a padding.
  - Pooling Layers - This is to **decrease the computational power required to process the data** through *dimensionality reduction.*
    - **Max Pooling** returns the **maximum value** from the portion of the image covered by the Kernel.
    - **Average Pooling** returns the **average of all the values** from the portion of the image covered by the Kernel.

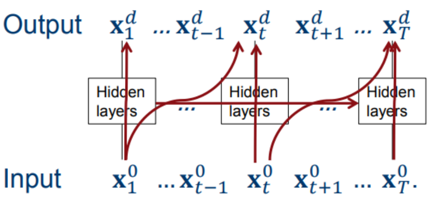


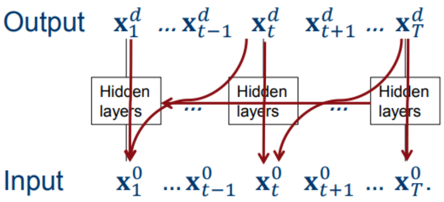
- **Recurrent Neural Networks**: RNNs learn similarly while training, in addition, they remember things learnt from prior input(s) while generating output(s).
    - A limitation of Vanilla Neural Networks (and also Convolutional Networks) is that their API is too constrained: they accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g. probabilities of different classes). Not only that: These models perform this mapping using a fixed amount of computational steps (e.g. the number of layers in the model).

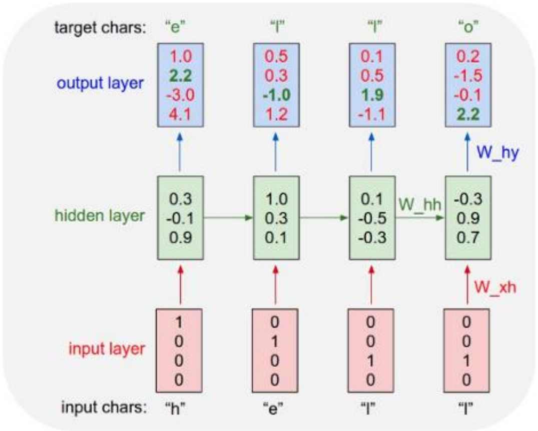
Forward propagation

- Input: time series $\mathbf{x}_1^0, \ldots, \mathbf{x}_T^0$.
- Output can be:
    - One output (vector) for entire time series: $\mathbf{x}^d$
    - One output at each time step: $\mathbf{x}_1^d, \ldots, \mathbf{x}_T^d$.

Back propagation through time.

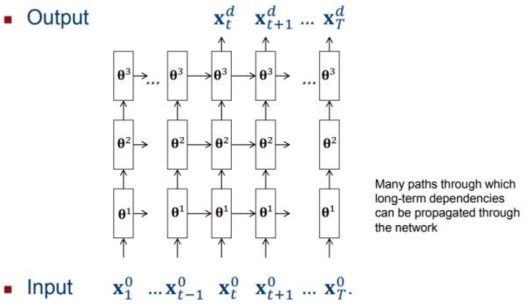

Output
$\mathbf{x}_1^d$ … $\mathbf{x}_{t-1}^d$ $\mathbf{x}_t^d$ $\mathbf{x}_{t+1}^d$ … $\mathbf{x}_T^d$
Hidden layers
…
Hidden layers
…
Hidden layers
Input
$\mathbf{x}_1^0$ … $\mathbf{x}_{t-1}^0$ $\mathbf{x}_t^0$ $\mathbf{x}_{t+1}^0$ … $\mathbf{x}_T^0$.


Output
$\mathbf{x}_1^d$ … $\mathbf{x}_{t-1}^d$ $\mathbf{x}_t^d$ $\mathbf{x}_{t+1}^d$ … $\mathbf{x}_T^d$
Hidden layers
…
Hidden layers
…
Hidden layers
Input
$\mathbf{x}_1^0$ … $\mathbf{x}_{t-1}^0$ $\mathbf{x}_t^0$ $\mathbf{x}_{t+1}^0$ … $\mathbf{x}_T^0$.



An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters "hell" as input. The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o"); We want the green numbers to be high and red numbers to be low.

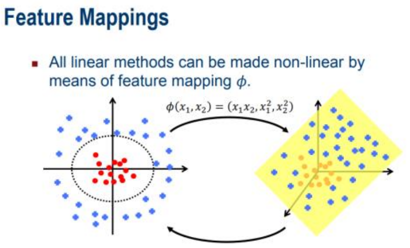
Deep Recurrent Neural Networks


Output
$\mathbf{x}_t^d$ $\mathbf{x}_{t+1}^d$ … $\mathbf{x}_T^d$
θ^3
θ^2
θ^1
Many paths through which long-term dependencies can be propagated through the network
Input
$\mathbf{x}_1^0$ … $\mathbf{x}_{t-1}^0$ $\mathbf{x}_t^0$ $\mathbf{x}_{t+1}^0$ … $\mathbf{x}_T^0$.

Summary

- Computational model of neural information processing.
- Feed-forward networks: layer-wise matrix multiplication + activation function.
- Back propagation: stochastic gradient descent. Gradient computation by layer-wise matrix multiplication + derivative of activation function.
- Convolutional neural networks: layer-wise application of local filters.
- Recurrent neural networks: state information passed on to next time step.

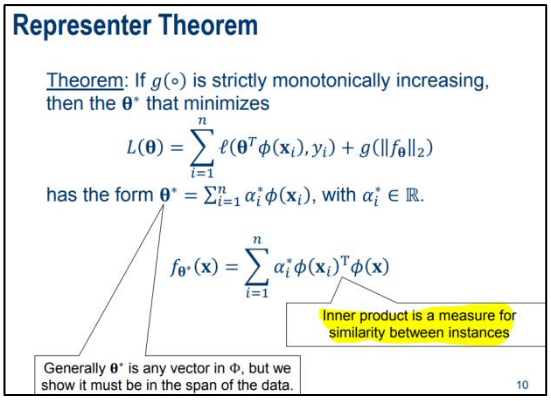
Kernel Methods



Feature Mappings

- All linear methods can be made non-linear by means of feature mapping ϕ.

- ν Find hyper plane in higher-dimensional space, corresponds to non-linear surface in feature space
- ν Feature space (X) need not be represented explicitly, can be infinite-dimensional.
- ν **Representer Theorem -** This is because popular kernels possess the problem of infinite dimensional space which may seem mathematically feasible but not practically viable. A **representer theorem** is any of several related results stating that a minimizer *f** of a regularized empirical risk functional defined over a reproducing kernel Hilbert space can be represented as a finite linear combination of kernel products evaluated on the input points in the training set data.



Representer Theorem

Theorem: If $g(\circ)$ is strictly monotonically increasing, then the $\boldsymbol{\theta}^*$ that minimizes

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} \ell(\boldsymbol{\theta}^T\phi(\mathbf{x}_i), y_i) + g(\|f_{\boldsymbol{\theta}}\|_2)$$

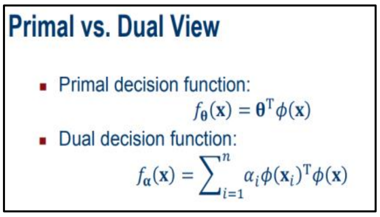
has the form $\boldsymbol{\theta}^* = \sum_{i=1}^{n} \alpha_i^*\phi(\mathbf{x}_i)$, with $\alpha_i^* \in \mathbb{R}$.

$$f_{\boldsymbol{\theta}^*}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i^*\phi(\mathbf{x}_i)^{\mathrm{T}}\phi(\mathbf{x})$$

Inner product is a measure for similarity between instances

Generally $\boldsymbol{\theta}^*$ is any vector in Φ, but we show it must be in the span of the data.

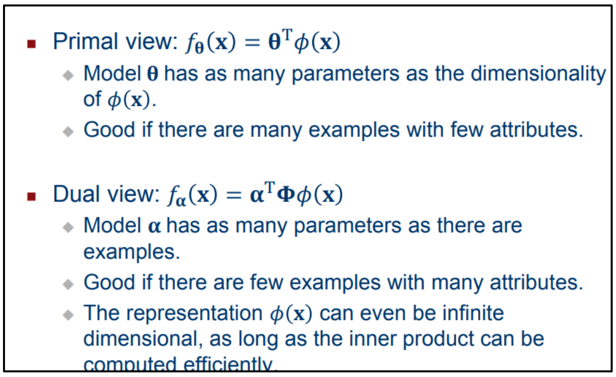
10



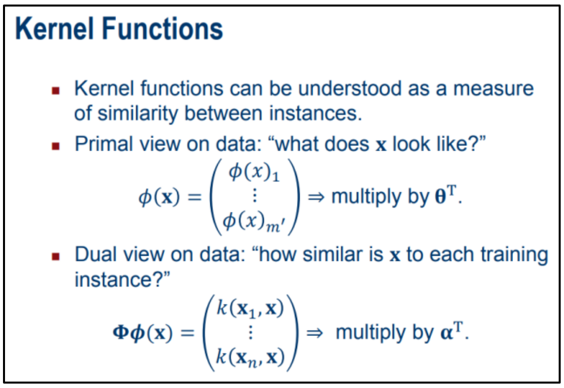
Primal vs. Dual View

- Primal decision function:
  $$f_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^{\mathrm{T}}\phi(\mathbf{x})$$
- Dual decision function:
  $$f_{\boldsymbol{\alpha}}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i\phi(\mathbf{x}_i)^{\mathrm{T}}\phi(\mathbf{x})$$

- ○ **Dual –** linear Program
- ○ **Primal –** Relation to its Dual is termed Primal. (The dual of the problem is again the primal problem) The solution to the dual problem provides a lower bound to the solution of the primal (minimization) problem



- Primal view: $f_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^{\mathrm{T}}\phi(\mathbf{x})$
  - Model $\boldsymbol{\theta}$ has as many parameters as the dimensionality of $\phi(\mathbf{x})$.
  - Good if there are many examples with few attributes.
- Dual view: $f_{\boldsymbol{\alpha}}(\mathbf{x}) = \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi}\phi(\mathbf{x})$
  - Model $\boldsymbol{\alpha}$ has as many parameters as there are examples.
  - Good if there are few examples with many attributes.
  - The representation $\phi(\mathbf{x})$ can even be infinite dimensional, as long as the inner product can be computed efficiently.



Kernel Functions

- Kernel functions can be understood as a measure of similarity between instances.
- Primal view on data: "what does $\mathbf{x}$ look like?"
  $$\phi(\mathbf{x}) = \begin{pmatrix} \phi(x)_1 \\ \vdots \\ \phi(x)_{m'} \end{pmatrix} \Rightarrow \text{multiply by } \boldsymbol{\theta}^{\mathrm{T}}.$$
- Dual view on data: "how similar is $\mathbf{x}$ to each training instance?"
  $$\boldsymbol{\Phi}\phi(\mathbf{x}) = \begin{pmatrix} k(\mathbf{x}_1,\mathbf{x}) \\ \vdots \\ k(\mathbf{x}_n,\mathbf{x}) \end{pmatrix} \Rightarrow \text{multiply by } \boldsymbol{\alpha}^{\mathrm{T}}.$$

- ν **Kernel Ridge Regression**

Minimize

$$L(\boldsymbol{\theta}) = (\boldsymbol{\Phi\theta} - \mathbf{y})^{\mathrm{T}}(\boldsymbol{\Phi\theta} - \mathbf{y}) + \lambda\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{\theta}$$

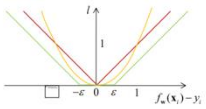
By the representer theorem:

$$\boldsymbol{\theta} = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\alpha}$$

Dual regularized empirical risk:

$$L(\boldsymbol{\alpha}) = \left(\boldsymbol{\Phi\Phi}^{\mathrm{T}}\boldsymbol{\alpha} - \mathbf{y}\right)^{\mathrm{T}}\left(\boldsymbol{\Phi\Phi}^{\mathrm{T}}\boldsymbol{\alpha} - \mathbf{y}\right) + \lambda\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi\Phi}^{\mathrm{T}}\boldsymbol{\alpha}$$

- Squared loss:
  $$\ell_2(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = (f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2$$
- L2 regularization:
  $$\Omega_2(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_2^2$$



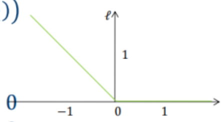
- Dual regularized empirical risk:
  $$\begin{aligned} L(\boldsymbol{\alpha}) &= \left(\boldsymbol{\Phi\Phi}^{\mathrm{T}}\boldsymbol{\alpha} - \mathbf{y}\right)^{\mathrm{T}}\left(\boldsymbol{\Phi\Phi}^{\mathrm{T}}\boldsymbol{\alpha} - \mathbf{y}\right) + \lambda\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi\Phi}^{\mathrm{T}}\boldsymbol{\alpha} \\ &= \boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi\Phi}^{\mathrm{T}}\boldsymbol{\Phi\Phi}^{\mathrm{T}}\boldsymbol{\alpha} - 2\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi\Phi}^{\mathrm{T}}\mathbf{y} - \mathbf{y}^{\mathrm{T}}\mathbf{y} \\ &+ \lambda\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Phi\Phi}^{\mathrm{T}}\boldsymbol{\alpha} \end{aligned}$$
- Define gram matrix (or kernel matrix) as $\mathbf{K} = \boldsymbol{\Phi\Phi}^{\mathrm{T}}$.
  $$L(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^{\mathrm{T}}\mathbf{KK}\boldsymbol{\alpha} - 2\boldsymbol{\alpha}^{\mathrm{T}}\mathbf{Ky} - \mathbf{y}^{\mathrm{T}}\mathbf{y} + \lambda\boldsymbol{\alpha}^{\mathrm{T}}\mathbf{K}\boldsymbol{\alpha}$$
- Setting the derivative to zero
  $$\frac{\partial}{\partial\boldsymbol{\alpha}}L(\boldsymbol{\alpha}) = \mathbf{0}$$
- Gives the solution
  $$\boldsymbol{\alpha} = (\mathbf{K} + \lambda\boldsymbol{I})^{-1}\mathbf{y}$$

- Kernel (gram) matrix: $\mathbf{K} = \boldsymbol{\Phi\Phi}^{\mathrm{T}}$
  $$\mathbf{K} = \begin{pmatrix} - & \phi(\mathbf{x}_1)^{\mathrm{T}} & - \\ \vdots & \ddots & \vdots \\ - & \phi(\mathbf{x}_n)^{\mathrm{T}} & - \end{pmatrix}\begin{pmatrix} | & \dots & | \\ \phi(\mathbf{x}_1) & \ddots & \phi(\mathbf{x}_n) \\ | & \dots & | \end{pmatrix} = \begin{pmatrix} k(\mathbf{x}_1,\mathbf{x}_1) & \dots & k(\mathbf{x}_1,\mathbf{x}_n) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_n,\mathbf{x}_1) & \dots & k(\mathbf{x}_n,\mathbf{x}_n) \end{pmatrix}$$
- $\mathbf{K}_{ij} = k(\mathbf{x}_i,\mathbf{x}_j)$

- Regression method that uses kernel functions
- Works with any nonlinear embedding $\boldsymbol{\phi}$ as long as there is a kernel function that computes the inner product: $k(\mathbf{x}_i,\mathbf{x}) = \phi(\mathbf{x}_i)^{\mathrm{T}}\phi(\mathbf{x})$.
- Kernel matrix $\mathbf{K}$ of size $n \times n$ has to be inverted, works only for modest sample sizes.
- Solution dependent on $\mathbf{K}_{ij} = k(\mathbf{x}_i,\mathbf{x}_j)$, but otherwise independent of $\boldsymbol{\Phi}$.
- For large sample size, use numeric optimization (e.g., stochastic gradient descent method).

- **Kernel Perceptron :** i.e. non-linear classifiers that employ a kernel function to compute the similarity of unseen samples to training samples

Loss function:
$\ell_p(f_{\boldsymbol\theta}(\mathbf{x}_i), y_i) = \max(0, -y_i f_{\boldsymbol\theta}(\mathbf{x}_i))$
No regularizer.
Primal stochastic gradient:
$$\nabla L_{\mathbf{x}_i}(\boldsymbol\theta) = \begin{cases} -y_i\mathbf{x}_i & -y_i f_{\boldsymbol\theta}(\mathbf{x}_i) > 0 \\ 0 & -y_i f_{\boldsymbol\theta}(\mathbf{x}_i) < 0 \end{cases}$$



Kernel Perceptron Algorithm

```
Perceptron(Instances $\{(\mathbf{x}_i, y_i)\}$)
    Set $\boldsymbol\alpha = \mathbf{0}$
    DO
        FOR $i = 1, \dots, n$
            IF      $y_i f_{\boldsymbol\alpha}(\mathbf{x}_i) \le 0$
            THEN    $\alpha_i = \alpha_i + y_i$
        END
    WHILE $\boldsymbol\alpha$ changes
    RETURN $\boldsymbol\alpha$
```

- Decision function:
$$f_{\boldsymbol\alpha}(\mathbf{x}) = \boldsymbol\alpha^{\mathrm{T}}\boldsymbol\Phi\phi(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x})$$

Stochastic gradient update step:

```
IF      $y_i f_{\boldsymbol\theta}(\mathbf{x}_i) \le 0$
THEN    $\boldsymbol\theta' = \boldsymbol\theta + y_i\mathbf{x}_i$
```

$$\boldsymbol\theta' = \boldsymbol\theta + y_i\phi(\mathbf{x}_i)$$
$$\Leftrightarrow \sum_{j=1}^{n} \alpha'_j \phi(\mathbf{x}_j) = \sum_{j=1}^{n} \alpha_j \phi(\mathbf{x}_j) + y_i\phi(\mathbf{x}_i)$$
$$\Leftarrow \alpha'_i\phi(\mathbf{x}_i) = \alpha_i\phi(\mathbf{x}_i) + y_i\phi(\mathbf{x}_i),$$
$$\forall j \neq i: \alpha'_j = \alpha_j$$
$$\Leftarrow \alpha'_i = \alpha_i + y_i$$

Dual stochastic gradient update step:

```
IF      $y_i f_{\boldsymbol\alpha}(\mathbf{x}_i) \le 0$
THEN    $\alpha_i = \alpha_i + y_i$
```

Kernel Perceptron

- Perceptron loss, no regularizer
- Dual form of the decision function:
$$f_{\boldsymbol\alpha}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x})$$
- Dual form of the update rule:
  - If $y_i f_{\boldsymbol\alpha}(\mathbf{x}_i) \le 0$, then $\alpha_i = \alpha_i + y_i$
- Equivalent to the primal form of the perceptron
- Advantageous to use instead of the primal perceptron if there are few samples and $\phi(\mathbf{x})$ is high dimensional.

- **Kernel Support Vector Machine :** *Solving Primal Problem by Dual formulation*

- Primal: $\min_{\boldsymbol\theta} \left[\sum_{i=1}^{n} \max(0, 1 - y_i\phi(\mathbf{x}_i)^{\mathrm{T}}\boldsymbol\theta) + \frac{1}{2\lambda}\boldsymbol\theta^{\mathrm{T}}\boldsymbol\theta\right]$
- Equivalent optimization problem with side constraints:
$$\min_{\boldsymbol\theta,\boldsymbol\xi} \left[\lambda\sum_{i=1}^{n} \xi_i + \frac{1}{2}\boldsymbol\theta^{\mathrm{T}}\boldsymbol\theta\right]$$
such that
$$y_i\phi(\mathbf{x}_i)^{\mathrm{T}}\boldsymbol\theta \ge 1 - \xi_i \text{ and } \xi_i \ge 0$$
- Goal: dual formulization of the optimization problem

- Optimization problem with side constraints:
$$\min_{\boldsymbol\theta,\boldsymbol\xi} \left[\lambda\sum_{i=1}^{n} \xi_i + \tfrac{1}{2}\boldsymbol\theta^{\mathrm{T}}\boldsymbol\theta\right]$$
such that
$$y_i\phi(\mathbf{x}_i)^{\mathrm{T}}\boldsymbol\theta \ge 1 - \xi_i \text{ and } \xi_i \ge 0$$

Goal function:	$Z(\boldsymbol\theta, \boldsymbol\xi)$
Side constraints:	$g(\boldsymbol\theta, \boldsymbol\xi) \ge 0$
Lagrange function:	$Z(\boldsymbol\theta, \boldsymbol\xi) - \beta g(\boldsymbol\theta, \boldsymbol\xi)$

- Lagrange function with Lagrange-Multipliers $\boldsymbol\beta \ge \mathbf{0}$ and $\boldsymbol\beta^0 \ge \mathbf{0}$ for the side constraints:
$$L(\boldsymbol\theta, \boldsymbol\xi, \boldsymbol\beta, \boldsymbol\beta^0) = \lambda\sum_{i=1}^{n} \xi_i + \frac{\boldsymbol\theta^{\mathrm{T}}\boldsymbol\theta}{2} - \sum_{i=1}^{n} \beta_i\left(y_i\phi(\mathbf{x}_i)^{\mathrm{T}}\boldsymbol\theta - 1 + \xi_i\right) - \sum_{i=1}^{n} \beta_i^0\xi_i$$
- Optimization problem without side constraints:
$$\min_{\boldsymbol\theta,\boldsymbol\xi} \max_{\boldsymbol\beta,\boldsymbol\beta^0} L(\boldsymbol\theta, \boldsymbol\xi, \boldsymbol\beta, \boldsymbol\beta^0)$$

$$\boldsymbol\theta = \sum_{i=1}^{n} \beta_i y_i \phi(\mathbf{x}_i)$$
$$\lambda = \beta_i + \beta_i^0$$

- Optimization criterion of the dual SVM:
$$\max_{\boldsymbol\beta} \sum_{i=1}^{n} \beta_i - \frac{1}{2}\sum_{i,j=1}^{n} \beta_i\beta_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$$
such that
$$0 \le \beta_i \le \lambda$$

Large if $\beta_i, \beta_j > 0$ for similar instances of different classes.

L1-Regularizer of $\boldsymbol\beta$ (sparse)

- Optimization criterion of the dual SVM:
$$\max_{\boldsymbol\beta} \sum_{i=1}^{n} \beta_i - \frac{1}{2}\sum_{i,j=1}^{n} \beta_i\beta_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$$
- Optimization over parameters $\boldsymbol\beta$.
- Solution found with QP-Solver in $O(n^2)$.
- Sparse solution.
- Samples only appear as pairwise inner products.

- Primal and dual optimization problem have the same solution.
$$\boldsymbol\theta = \sum_{\mathbf{x}_i \in SV} \beta_i y_i \phi(\mathbf{x}_i)$$
Support Vectors: $\beta_i > 0$
- Dual form of the decision function:
$$f_{\boldsymbol\beta}(\mathbf{x}) = \sum_{\mathbf{x}_i \in SV} \beta_i y_i k(\mathbf{x}_i, \mathbf{x})$$
- Primal SVM:
  - Solution is a Vector $\boldsymbol\theta$ in the space of the attributes.
- Dual SVM:
  - The same solution is represented as weights β_i of the samples.

Kernels

- Kernel matrices are symmetric:
$$\mathbf{K} = \mathbf{K}^{\mathrm{T}}$$
- Kernel matrices $\mathbf{K} \in \mathbb{R}^{n \times n}$ are positive semidefinite:
$$\exists \boldsymbol\Phi \in \mathbb{R}^{n \times m}: \mathbf{K} = \boldsymbol\Phi\boldsymbol\Phi^{\mathrm{T}}$$
- Kernel function $k(\mathbf{x}, \mathbf{x}')$ is positive semidefinite if $\mathbf{K}$ is positive semidefinite for every data set.
- For every positive definite function k there is at least one mapping $\phi(\mathbf{x})$ such that $k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^{\mathrm{T}}\phi(\mathbf{x}')$ for all $\mathbf{x}$ and $\mathbf{x}'$.

Constructing Kernels

- Design embedding $\phi(\mathbf{x})$, then obtain resulting kernel function $k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^{\mathrm{T}}\phi(\mathbf{x}')$.
- Or: just define kernel function (any similarity measure) $k(\mathbf{x}, \mathbf{x}')$ directly, don't bother with embedding.
- For which functions k does there exist a mapping $\phi(\mathbf{x})$, so that k represents an inner product?

- **Mercer Map :** provides a feature mapping

$$\mathbf{K} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{-1}, \text{ with } \boldsymbol{\Lambda} = \begin{pmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_n \end{pmatrix} \text{ \& } \mathbf{U} = \begin{pmatrix} | & & | \\ \mathbf{u}_1 & \cdots & \mathbf{u}_n \\ | & & | \end{pmatrix}$$

  - Based on Eigenvalue decomposition
  - Useful if a learning problem is given as a kernel function but learning should take place in the primal.

Feature mapping for used training data can then be defined as

$$\begin{pmatrix} | & & | \\ \phi(\mathbf{x}_1) & \cdots & \phi(\mathbf{x}_n) \\ | & & | \end{pmatrix} = \left(\mathbf{U}\boldsymbol{\Lambda}^{1/2}\right)^{\mathrm{T}}$$

Kernel matrix between training and test data

$$\begin{aligned} \mathbf{K}_{test} &= \Phi(\mathbf{X}_{train})^{\mathrm{T}}\Phi(\mathbf{X}_{test}) \\ &= \left(\mathbf{U}\boldsymbol{\Lambda}^{1/2}\right)\Phi(\mathbf{X}_{test}) \end{aligned}$$

Equation results in a mapping of the test data:

$$\Phi(\mathbf{X}_{test}) = \left(\mathbf{U}\boldsymbol{\Lambda}^{1/2}\right)^{-1}\mathbf{K}_{test}$$

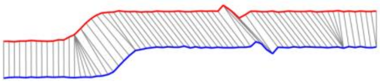
$$\Phi(\mathbf{X}_{test}) = \boldsymbol{\Lambda}^{-1/2}\mathbf{U}^{\mathrm{T}}\mathbf{K}_{test}$$

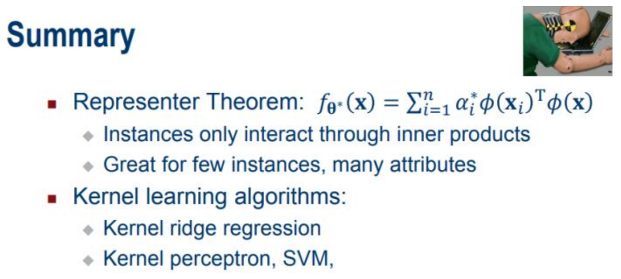
$$\boxed{\mathbf{U}^{\mathrm{T}} = \mathbf{U}^{-1}}$$

- **Kernel Functions**
  - Polynomial kernels $k_{poly}(\mathbf{x}_i, \mathbf{x}_j) = \left(\mathbf{x}_i^{\mathrm{T}}\mathbf{x}_j + 1\right)^p$
  - Radial basis functions $k_{RBF}(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|}$
    - Returns based on the nearby points
  - Sigmoid kernels
  - Dynamic time-warping kernels (DTW) – Based on similarity
  - String kernels - substrings / subsequence's
  - Graph kernels
    - How similar are two Graphs?
      - Counting the number of "common" paths in the graph.
    - How similar are two nodes within a Graph?
      - number of "common" paths in their product graph
    - Shortest-Path Kernel
    - Subtree-Kernel

- DTW distance is squared distance between matched sequences:

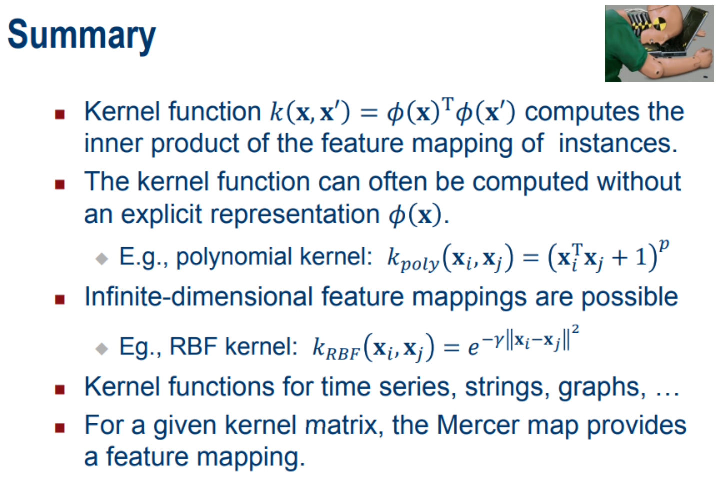
$$k_{DTW}(\mathbf{x}, \mathbf{x}') = e^{-\left(\min \sum_{k=1}^{T}\left(\mathbf{x}_{\pi_{\mathbf{x}(k)}} - \mathbf{x}'_{\pi_{\mathbf{x}'(k)}}\right)^2\right)}$$





Summary

- Representer Theorem: $f_{\boldsymbol{\theta}^*}(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i^* \phi(\mathbf{x}_i)^{\mathrm{T}}\phi(\mathbf{x})$
  - Instances only interact through inner products
  - Great for few instances, many attributes
- Kernel learning algorithms:
  - Kernel ridge regression
  - Kernel perceptron, SVM,



Summary

- Kernel function $k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^{\mathrm{T}}\phi(\mathbf{x}')$ computes the inner product of the feature mapping of instances.
- The kernel function can often be computed without an explicit representation $\phi(\mathbf{x})$.
  - E.g., polynomial kernel: $k_{poly}(\mathbf{x}_i, \mathbf{x}_j) = \left(\mathbf{x}_i^{\mathrm{T}}\mathbf{x}_j + 1\right)^p$
- Infinite-dimensional feature mappings are possible
  - Eg., RBF kernel: $k_{RBF}(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2}$
- Kernel functions for time series, strings, graphs, …
- For a given kernel matrix, the Mercer map provides a feature mapping.