AutoEncoders & Kernels

CMSC 422

SOHEIL FEIZI

sfeizi@cs.umd.edu

Slides adapted from MARINE CARPUAT and GUY GOLAN

Today's topics

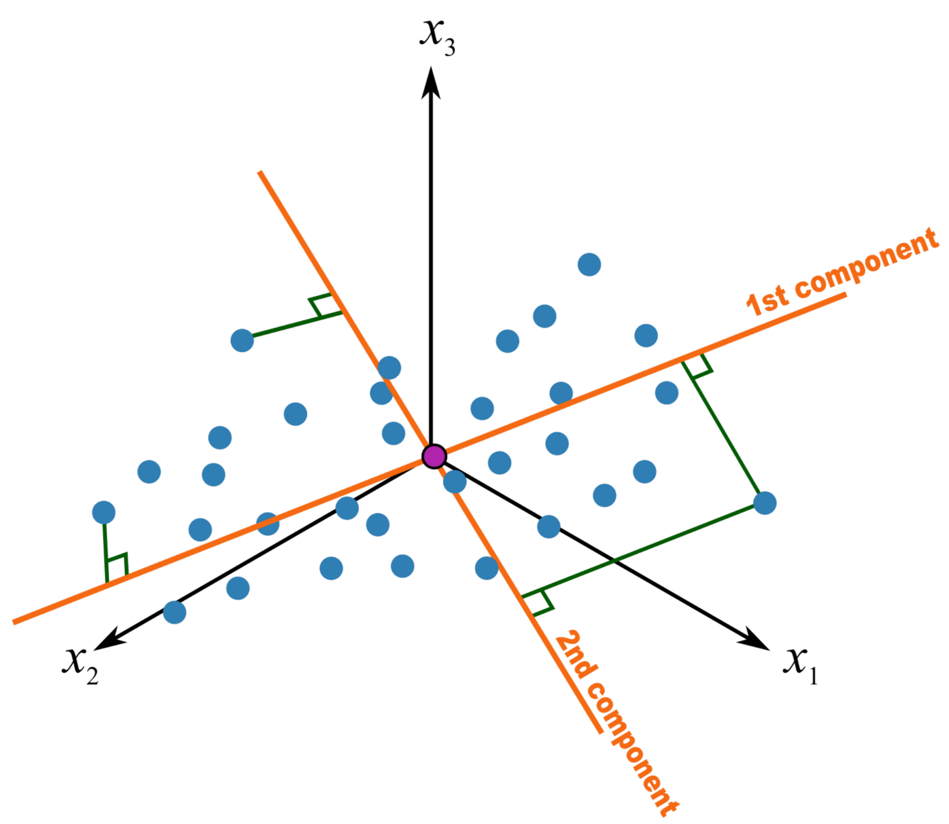
- Nonlinear dimensionality reduction

- Kernel methods

PCA – Principal Component analysis

- Statistical approach for data compression and visualization
- Invented by Karl Pearson in 1901

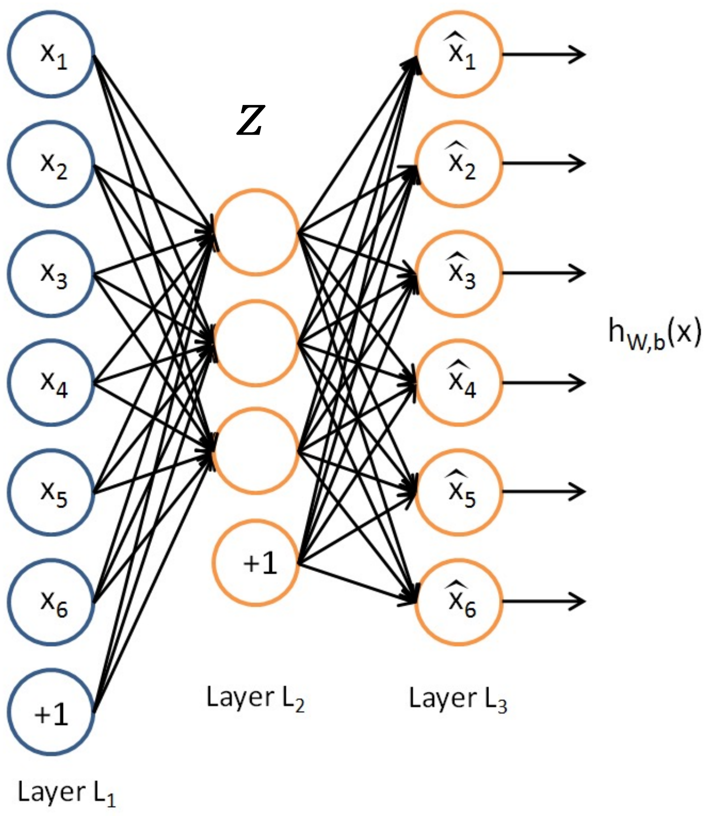
- Weakness: linear components only.



Autoencoder

- Unlike the **PCA** now we can use activation functions to achieve non-linearity.

- It has been shown that an AE without activation functions achieves the **PCA** capacity.



Uses

- The autoencoder idea was a part of NN history for decades (LeCun et al, 1987).

- Traditionally an autoencoder is used for dimensionality reduction and feature learning.

- Recently, the connection between autoencoders and latent space modeling has brought autoencoders to the front of generative modeling

Simple Idea

- Given data x (no labels) we would like to learn the functions f (encoder) and g (decoder) where:

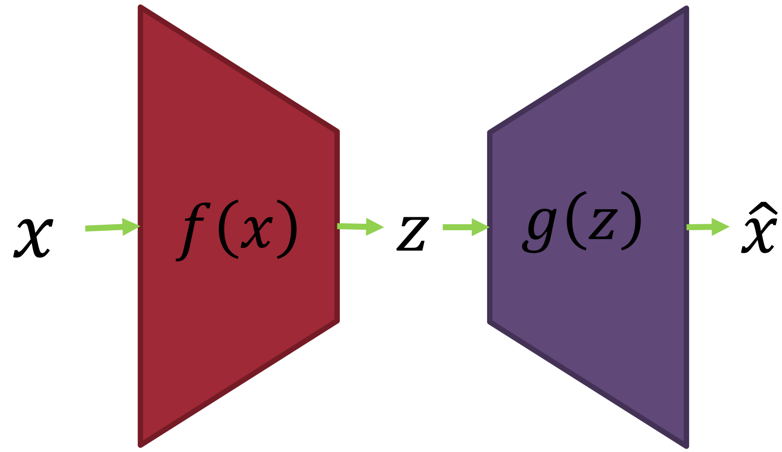
$$f(x) = s(wx + b) = z$$

and

$$g(z) = s(w'z + b') = \hat{x}$$

s.t $h(x) = g\big(f(x)\big) = \hat{x}$

where h is an **approximation** of the identity function.



(z is some **latent** representation or **code** and s is a non-linearity such as the sigmoid)

($\hat{x}$ is x's reconstruction)

Simple Idea

Learning the identity function seems trivial, but with added constraints on the network (such as limiting the number of hidden neurons or regularization) we can learn information about the structure of the data.

Trying to capture the distribution of the data (data specific!)

Training the AE

Using **Gradient Descent** we can simply train the model as any other FC NN with:

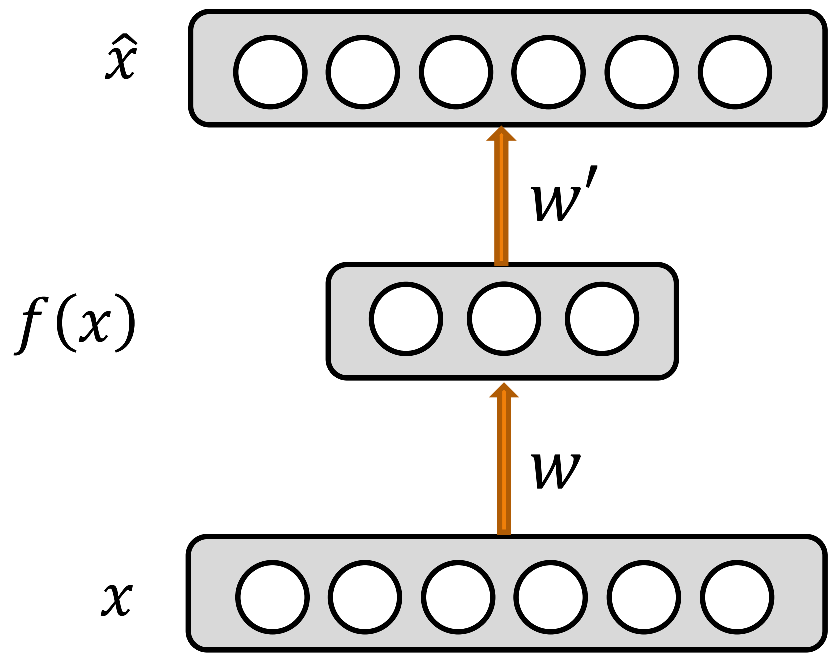
- Traditionally with *squared error* loss function

$$L(x, \hat{x}) = \|x - \hat{x}\|^2$$

- Why?

AE Architecture

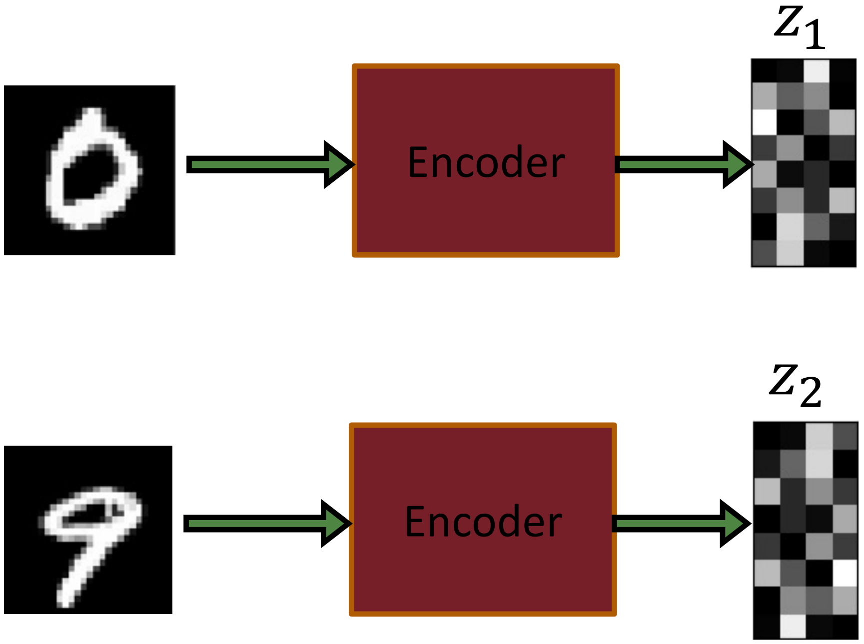
- Hidden layer is **Undercomplete** if smaller than the input layer
  - ❑Compresses the input
  - ❑Compresses well only for the training dist.
- Hidden nodes will be
  - ❑Good features for the training distribution.
  - ❑Bad for other types on input



Deep Autoencoder Example

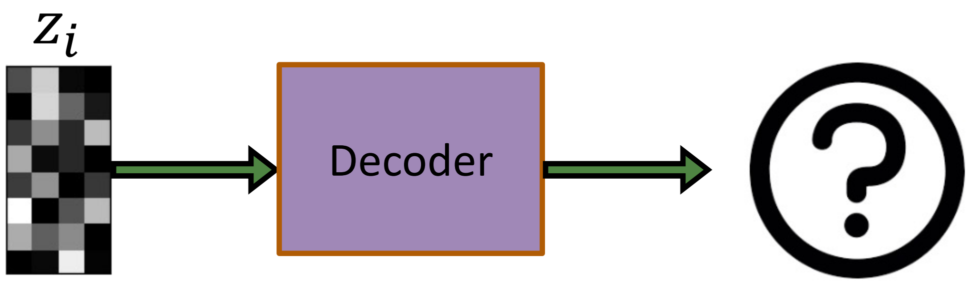
- https://cs.stanford.edu/people/karpathy/convnetjs/demo/autoencoder.html - By Andrej Karpathy

Simple latent space interpolation

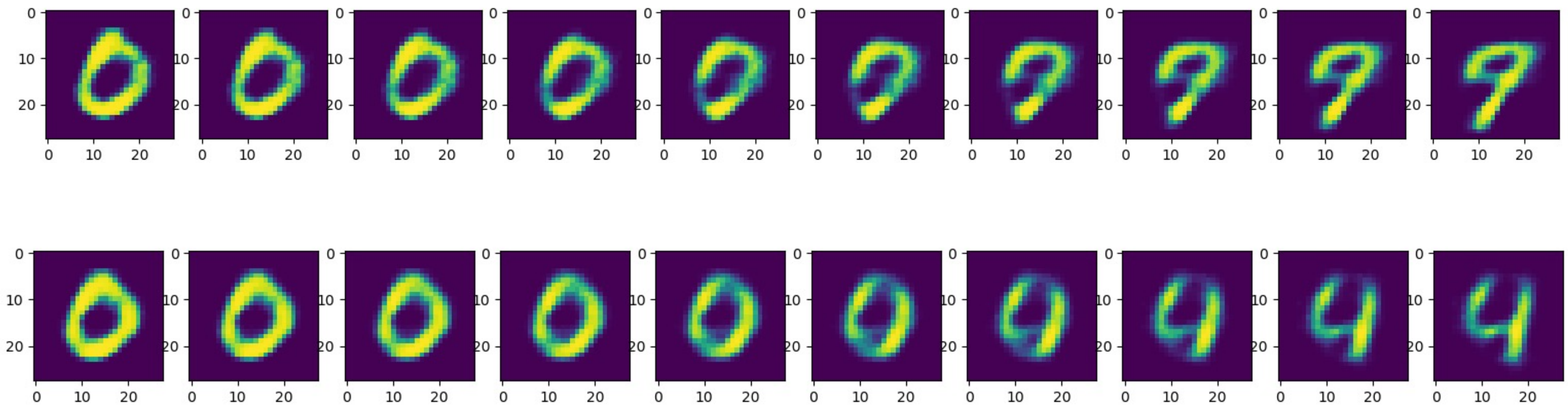


Simple latent space interpolation

$$z_i = \alpha \underset{}{z_1} + (1 - \alpha) z_2$$



Simple latent space interpolation



Kernel Methods

Beyond linear classification

- Problem: linear classifiers
  - Easy to implement and easy to optimize
  - But limited to linear decision boundaries

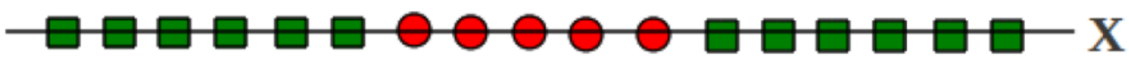
- What can we do about it?
  - Neural networks
    - Very expressive but harder to optimize (non-convex objective)
  - Today: Kernels

Kernel Methods

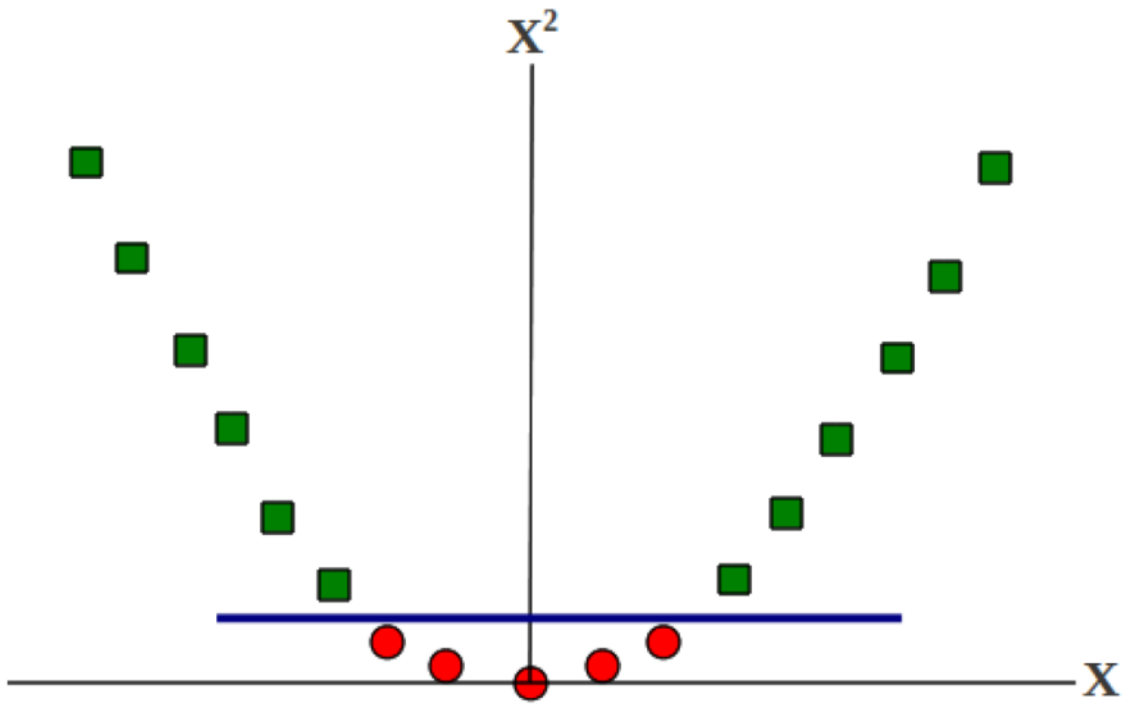
- Goal: keep advantages of linear models, but make them capture non-linear patterns in data!

- How?
  - By mapping data to higher dimensions where it exhibits linear patterns

Classifying non-linearly separable data with a linear classifier: examples



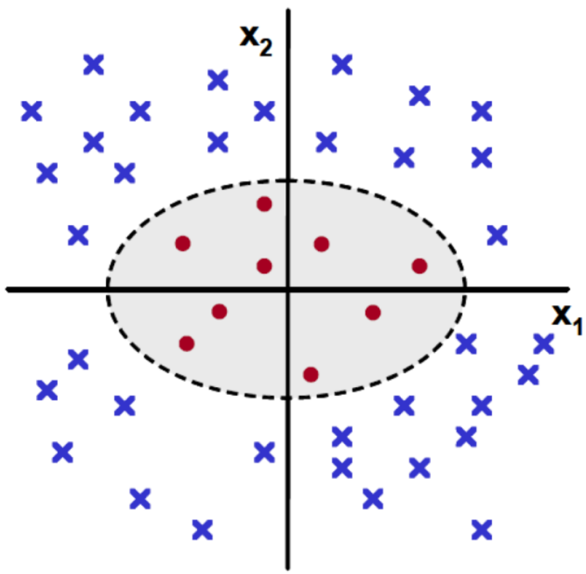
Non-linearly separable data in 1D



Becomes linearly separable in new 2D space
defined by the following mapping:

$$x \rightarrow \{x, x^2\}$$

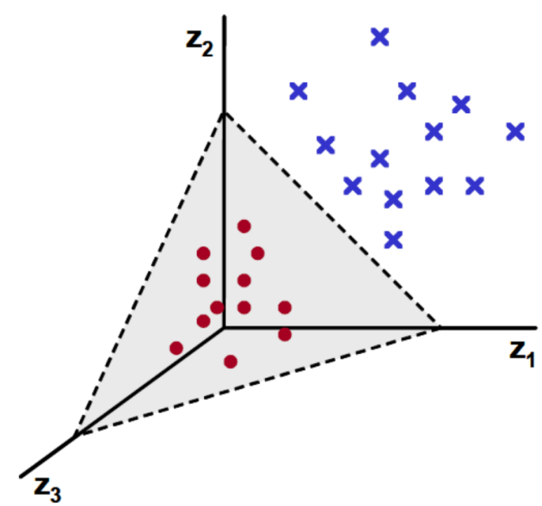
Classifying non-linearly separable data with a linear classifier: examples



Non-linearly separable data in 2D

Becomes linearly separable in the 3D space defined by the following transformation:

$$\mathbf{x} = \{x_1, x_2\} \rightarrow \mathbf{z} = \{x_1^2, \sqrt{2}x_1x_2, x_2^2\}$$



Defining feature mappings

- Map an original feature vector $\boldsymbol{x} = \langle x_1, x_2, x_3, \ldots, x_D \rangle$ to an expanded version $\phi(\boldsymbol{x})$

- Example: quadratic feature mapping represents feature combinations

$$\begin{aligned}
\phi(\boldsymbol{x}) = \langle 1, 2x_1, 2x_2, 2x_3, \ldots, 2x_D, \\
& x_1^2, x_1x_2, x_1x_3, \ldots, x_1x_D, \\
& x_2x_1, x_2^2, x_2x_3, \ldots, x_2x_D, \\
& x_3x_1, x_3x_2, x_3^2, \ldots, x_2x_D, \\
& \ldots, \\
& x_Dx_1, x_Dx_2, x_Dx_3, \ldots, x_D^2 \rangle
\end{aligned}$$

Feature Mappings

- Pros: can help turn non-linear classification problem into linear problem

- Cons: "feature explosion" creates issues when training linear classifier in new feature space
  - More computationally expensive to train
  - More training examples needed to avoid overfitting

Kernel Methods

- Goal: keep advantages of linear models, but make them capture non-linear patterns in data!
- How?
  - By mapping data to higher dimensions where it exhibits linear patterns
  - **By rewriting linear models so that the mapping never needs to be explicitly computed**

The Kernel Trick

- Rewrite learning algorithms so they only depend on **dot products between two examples**

- Replace dot product $\phi(\mathbf{x})^\top \phi(\mathbf{z})$
  by **kernel function** $k(\mathbf{x}, \mathbf{z})$
  which computes the dot product **implicitly**

Example of Kernel function

Consider two examples $\mathbf{x} = \{x_1, x_2\}$ and $\mathbf{z} = \{z_1, z_2\}$

Let's assume we are given a function k (kernel) that takes as inputs $\mathbf{x}$ and $\mathbf{z}$

$$
\begin{aligned}
k(\mathbf{x}, \mathbf{z}) &= (\mathbf{x}^\top \mathbf{z})^2 \\
&= (x_1 z_1 + x_2 z_2)^2 \\
&= x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 x_2 z_1 z_2 \\
&= (x_1^2, \sqrt{2} x_1 x_2, x_2^2)^\top (z_1^2, \sqrt{2} z_1 z_2, z_2^2) \\
&= \phi(\mathbf{x})^\top \phi(\mathbf{z})
\end{aligned}
$$

The above k implicitly defines a mapping ϕ to a higher dimensional space

$$\phi(\mathbf{x}) = \{x_1^2, \sqrt{2} x_1 x_2, x_2^2\}$$

Another example of Kernel Function (from CIML)

$$\begin{aligned}\phi(\boldsymbol{x}) = \langle 1, 2x_1, 2x_2, 2x_3, \dots, 2x_D,\\ x_1^2, x_1x_2, x_1x_3, \dots, x_1x_D,\\ x_2x_1, x_2^2, x_2x_3, \dots, x_2x_D,\\ x_3x_1, x_3x_2, x_3^2, \dots, x_2x_D,\\ \dots,\\ x_Dx_1, x_Dx_2, x_Dx_3, \dots, x_D^2\rangle\end{aligned}$$

What is the function k(x,z) that can implicitly compute the dot product $\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z})$?

$$\begin{aligned}\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z}) &= 1 + x_1z_1 + x_2z_2 + \dots + x_Dz_D + x_1^2z_1^2 + \dots + x_1x_Dz_1z_D + \\ &\quad \dots + x_Dx_1z_Dz_1 + x_Dx_2z_Dz_2 + \dots + x_D^2z_D^2 && (9.2)\\ &= 1 + 2\sum_d x_dz_d + \sum_d\sum_e x_dx_ez_dz_e && (9.3)\\ &= 1 + 2\boldsymbol{x}\cdot\boldsymbol{z} + (\boldsymbol{x}\cdot\boldsymbol{z})^2 && (9.4)\\ &= (1 + \boldsymbol{x}\cdot\boldsymbol{z})^2 && (9.5)\end{aligned}$$

Kernels: Formally defined

Recall: Each kernel k has an associated feature mapping ϕ

ϕ takes input $\mathbf{x} \in \mathcal{X}$ (input space) and maps it to $\mathcal{F}$ ("feature space")

Kernel $k(\mathbf{x}, \mathbf{z})$ takes two inputs and gives their similarity in $\mathcal{F}$ space

$$
\begin{aligned}
\phi &: \mathcal{X} \to \mathcal{F} \\
k &: \mathcal{X} \times \mathcal{X} \to \mathbb{R}, \quad k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})
\end{aligned}
$$

$\mathcal{F}$ needs to be a *vector space* with a *dot product* defined on it

Also called a *Hilbert Space*

Kernels: Mercer's condition

- Can *any* function be used as a kernel function?
  - No! it must satisfy Mercer's condition.

For k to be a kernel function

- There must exist a Hilbert Space $\mathcal{F}$ for which k defines a dot product
- The above is true if K is a positive definite function

$$\int d\mathbf{x} \int d\mathbf{z} f(\mathbf{x}) k(\mathbf{x}, \mathbf{z}) f(\mathbf{z}) > 0$$

For all square integrable functions f

Kernels: Constructing combinations of kernels

Let k_1, k_2 be two kernel functions then the following are as well

- $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z}) + k_2(\mathbf{x}, \mathbf{z})$: direct sum
- $k(\mathbf{x}, \mathbf{z}) = \alpha k_1(\mathbf{x}, \mathbf{z})$: scalar product
- $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z}) k_2(\mathbf{x}, \mathbf{z})$: direct product

Commonly Used Kernel Functions

Linear (trivial) Kernel:

$$k(\mathbf{x}, \mathbf{z}) = \mathbf{x}^\top \mathbf{z} \text{ (mapping function } \phi \text{ is identity - no mapping)}$$

Quadratic Kernel:

$$k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z})^2 \quad \text{or} \quad (1 + \mathbf{x}^\top \mathbf{z})^2$$

Polynomial Kernel (of degree d):

$$k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z})^d \quad \text{or} \quad (1 + \mathbf{x}^\top \mathbf{z})^d$$

Radial Basis Function (RBF) Kernel:

$$k(\mathbf{x}, \mathbf{z}) = \exp[-\gamma ||\mathbf{x} - \mathbf{z}||^2]$$

The Kernel Trick

- Rewrite learning algorithms so they only depend on **dot products between two examples**

- Replace dot product $\phi(\mathbf{x})^\top \phi(\mathbf{z})$ by **kernel function** $k(\mathbf{x}, \mathbf{z})$ which computes the dot product **implicitly**

"Kernelizing" the perceptron

- Naïve approach: let's explicitly train a perceptron in the new feature space

Algorithm 28 PerceptronTrain($\mathbf{D}$, *MaxIter*)

```
w ← 0, b ← 0                              // initialize weights and bias
for iter = 1 ... MaxIter do
    for all (x,y) ∈ D do
        a ← w · φ(x) + b                  // compute activation for this example
        if ya ≤ 0 then
            w ← w + y φ(x)                // update weights
            b ← b + y                     // update bias
        end if
    end for
end for
return w, b
```

Can we apply the Kernel trick?
Not yet, we need to rewrite the algorithm using dot products between examples

"Kernelizing" the perceptron

- Perceptron Representer Theorem

"During a run of the perceptron algorithm, the weight vector w can always be represented as a linear combination of the expanded training data"

Proof by induction

(in CIML)

"Kernelizing" the perceptron

- We can use the perceptron representer theorem to compute activations as a **dot product** between examples

$$\boldsymbol{w} \cdot \phi(\boldsymbol{x}) + b = \left(\sum_n \alpha_n \phi(\boldsymbol{x}_n) \right) \cdot \phi(\boldsymbol{x}) + b \qquad \text{definition of } \boldsymbol{w} \tag{9.6}$$

$$= \sum_n \alpha_n \left[\phi(\boldsymbol{x}_n) \cdot \phi(\boldsymbol{x}) \right] + b \qquad \text{dot products are linear} \tag{9.7}$$

"Kernelizing" the perceptron

Algorithm 29 KernelizedPerceptronTrain($\mathbf{D}$, *MaxIter*)

```
α ← 0, b ← 0                                  // initialize coefficients and bias
for iter = 1 ... MaxIter do
    for all (x_n, y_n) ∈ D do
        a ← Σ_m α_m φ(x_m) · φ(x_n) + b       // compute activation for this example
        if y_n a ≤ 0 then
            α_n ← α_n + y_n                   // update coefficients
            b ← b + y                         // update bias
        end if
    end for
end for
return α, b
```

- Same training algorithm, but
doesn't explicitly refers to weights w anymore
only depends on dot products between examples

- We can apply the kernel trick!

Kernel Methods

- Goal: keep advantages of linear models, but make them capture non-linear patterns in data!
- How?
  - By **mapping data to higher dimensions** where it exhibits linear patterns
  - By **rewriting linear models** so that the **mapping never needs to be explicitly computed**

Discussion

- Other algorithms can be kernelized:
  - See CIML for K-means

- Do Kernels address all the downsides of "feature explosion"?
  - Helps reduce computation cost during training
  - But overfitting remains an issue

What you should know

- Kernel functions
  - What they are, why they are useful, how they relate to feature combination
- Kernelized perceptron
  - You should be able to derive it and implement it