



S&DS 365 / 565  
Intermediate Machine Learning

# Kernels and Neural Networks

September 16

Yale

# Reminders

- Assignment 1 out; due September 25 (week from this Wed)
- Quiz 2 posted Wednesday, material up to today
- Check Canvas/EdD for office hours—please join us!

# Today: Kernels and Neural nets

- 1 Recap/discussion of RKHS concepts
- 2 Basic architecture of feedforward neural nets
- 3 Backpropagation
- 4 Examples: TensorFlow
- 5 Next time: NTK and double descent

# **1: Mercer kernel recap**

# Summary from last time

- Smoothing methods compute local averages, weighting points by a kernel. The details of the kernel don't matter much
- Mercer kernels using penalization rather than smoothing
- Defining property: Matrix  $\mathbb{K}$  is always positive semidefinite
- Equivalent to a type of ridge regression in function space
- The curse of dimensionality limits use of both approaches

# Mercer Kernels: The big picture

Instead of using local smoothing, we can optimize the fit to the data subject to regularization (penalization). Choose  $\hat{m}$  to minimize

$$\sum_i (Y_i - \hat{m}(X_i))^2 + \lambda \text{penalty}(\hat{m})$$

where  $\text{penalty}(\hat{m})$  is a *roughness penalty*.

$\lambda$  is a parameter that controls the amount of smoothing.

How do we construct a penalty that measures roughness? One approach is: *Mercer Kernels* and *RKHS = Reproducing Kernel Hilbert Spaces*.

# What is a Mercer Kernel?

A kernel is a bivariate function  $K(x, x')$ . We think of this as a measure of “similarity” between points  $x$  and  $x'$ .

# What is a Mercer Kernel?

A kernel is a bivariate function  $K(x, x')$ . We think of this as a measure of “similarity” between points  $x$  and  $x'$ .

A Mercer kernel has a special property: For any set of points  $x_1, \dots, x_n$  the  $n \times n$  matrix

$$\mathbb{K} = [K(x_i, x_j)]$$

is positive semidefinite (no negative eigenvalues)



# What is a Mercer Kernel?

A kernel is a bivariate function  $K(x, x')$ . We think of this as a measure of “similarity” between points  $x$  and  $x'$ .

A Mercer kernel has a special property: For any set of points  $x_1, \dots, x_n$  the  $n \times n$  matrix

$$\mathbb{K} = [K(x_i, x_j)]$$

is positive semidefinite (no negative eigenvalues)

This property has many important (and beautiful!) mathematical consequences. It is a characterization of Mercer kernels.

# Mercer Kernels: Key example

A Gaussian gives us a Mercer kernel:

$$K(x, x') = e^{-\frac{\|x - x'\|^2}{2h^2}}$$

Note: Here we fix the bandwidth  $h$ .

# Basis functions

We can create a set of *basis functions* based on  $K$ .

Fix  $z$  and think of  $K(z, x)$  as a function of  $x$ . That is,

$$K(z, x) = K_z(x)$$

is a function of the second argument, with the first argument fixed.

# Defining an inner product (geometry)

Because of the positive semidefinite property, we can create an *inner product* and *norm* over the span of these functions

If  $f(x) = \sum_r \alpha_r K_{z_r}(x)$ ,  $g(x) = \sum_s \beta_s K_{y_s}(x)$ , the inner product is

$$\begin{aligned}\langle f, g \rangle_K &= \sum_r \sum_s \alpha_r \beta_s K(z_r, y_s) \\ &= \alpha^T \mathbb{K} \beta\end{aligned}$$

where  $\mathbb{K} = [K(z_r, y_s)]$

# Defining an inner product (geometry)

Because of the positive semidefinite property, we can create an *inner product* and *norm* over the span of these functions

The norm is

$$\begin{aligned}\|f\|_K^2 &= \langle f, f \rangle_K = \sum_r \sum_s \alpha_r \alpha_s K(z_r, z_s) \\ &= \alpha^T \mathbb{K} \alpha \geq 0\end{aligned}$$

*In fact  $\|f\|_K = 0$  if and only if  $f = 0$*  (see notes)

Assignment 1 will solidify your understanding of Mercer kernels and kernel ridge regression!

# Reducing to finite dimensions

## Representer Theorem

Let  $\hat{m}$  minimize

$$J(m) = \sum_{i=1}^n (Y_i - m(X_i))^2 + \lambda \|m\|_K^2.$$

Then

$$\hat{m}(x) = \sum_{i=1}^n \alpha_i K(X_i, x)$$

for some  $\alpha_1, \dots, \alpha_n$ .

So, we only need to find the coefficients

$$\alpha = (\alpha_1, \dots, \alpha_n).$$

# Gradient descent

The gradient descent updates to  $\alpha$  are

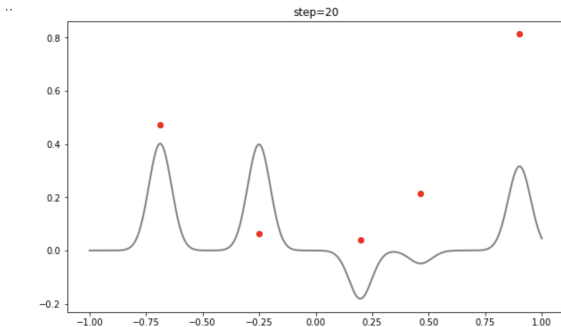
$$\alpha \longleftarrow \alpha + \eta (\mathbb{K}(\mathbf{y} - \mathbb{K}\alpha) - \lambda \mathbb{K}\alpha)$$

where  $\mathbb{K}$  is the  $n \times n$  Gram matrix and  $\eta > 0$  is a step size.

# Demo

```
if step % 10 == 0:  
    plot_function_and_data(x, f, X, y, t=step, sleeptime=.5,  
alpha = alpha + stepsize * (K.T @ (y - K @ alpha) - lam * K
```

6] 6.4s





## **2: Neural net basics**

# Recall :-)

## What does “Intermediate” imply?

- A second course in machine learning
- Assume familiar with things like PCA, bias/variance, maximum likelihood, **basics of neural nets**
- Have experimented with basic ML methods on some data sets
- Previous exposure to Python
- More on this later...

# Starting with regression

For linear regression, our loss function for an example  $(x, y)$  is

$$\begin{aligned}\mathcal{L}(x, y) &= \frac{1}{2}(y - \beta^T x - \beta_0)^2 \\ &= \frac{1}{2}(y - f(x))^2\end{aligned}$$

where  $f(x) = \beta^T x + \beta_0$ .

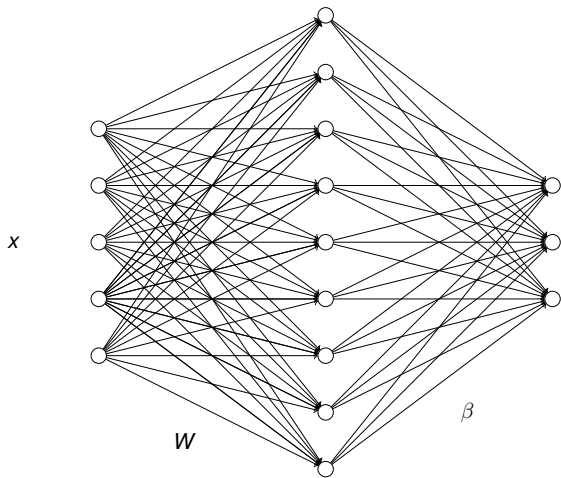
# Adding a layer

Loss is

$$\mathcal{L} = \frac{1}{2}(y - f(x))^2$$

where now  $f(x) = \beta^T h(x) + \beta_0$  where  $h(x) = Wx + b$ .

This can be viewed graphically.



# Equivalent to linear model

But this is just another linear model

$$f(x) = \tilde{\beta}^T x + \tilde{\beta}_0$$

We get a reparameterization of a linear model; nothing new.

Need to add *nonlinearities*

# Nonlinearities

Add nonlinearity

$$h(x) = \varphi(Wx + b)$$

applied component-wise.

For regression, the last layer is just linear:

$$f(x) = \beta^T h(x) + \beta_0$$

# Nonlinearities

Commonly used nonlinearities:

$$\varphi(u) = \tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$$

$$\varphi(u) = \text{sigmoid}(u) = \frac{e^u}{1 + e^u}$$

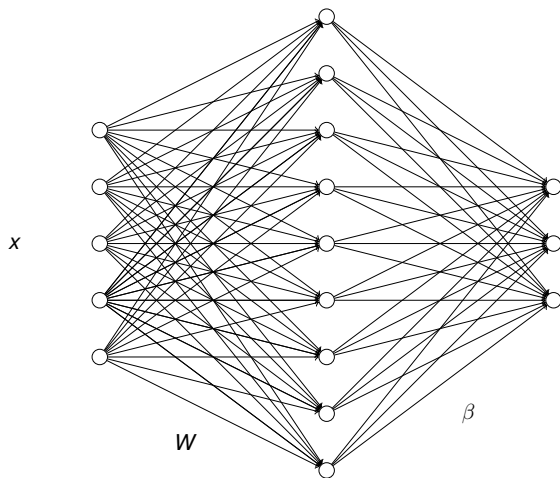
$$\varphi(u) = \text{relu}(u) = \max(u, 0)$$



# Nonlinearities

So, a neural network is nothing more than a parametric regression model with a restricted type of nonlinearity

# Two-layer dense network (multi-layer perceptron)

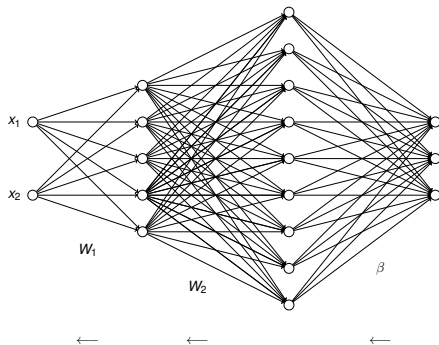


### **3: Backprop**

# Training

- The parameters are trained by stochastic gradient descent.
- To calculate derivatives we just use the chain rule, working our way backwards from the last layer to the first.

# High level idea



Start at last layer, send error information back to previous layers

# Start simple

Loss is

$$\mathcal{L} = \frac{1}{2}(y - f)^2$$

The change in loss due to making a small change in output  $f$  is

$$\frac{\partial \mathcal{L}}{\partial f} = (f - y)$$

We now send this backward through the network

# Example

So if  $f = Wx + b$  then

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W} &= \frac{\partial \mathcal{L}}{\partial f} x^T \\ &= (f - y) x^T\end{aligned}$$

# Example

So if  $f = Wx + b$  then

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial f} \\ &= (f - y)\end{aligned}$$



# Two layers

Now add a layer:

$$f = W_2 h + b_2$$

$$h = W_1 x + b_1$$

Then we have

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_2} &= \frac{\partial \mathcal{L}}{\partial f} h^T \\ &= (f - y) h^T\end{aligned}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial h} &= W_2^T \frac{\partial \mathcal{L}}{\partial f} \\ &= W_2^T (f - y)\end{aligned}$$

## Two layers

Now send this back (backpropagate) to the first layer:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_1} &= \frac{\partial \mathcal{L}}{\partial h} x^T \\ &= W_2^T \frac{\partial \mathcal{L}}{\partial f} x^T \\ &= W_2^T (f - y) x^T\end{aligned}$$

# Adding a nonlinearity

Remember, this just gives a linear model! Need a nonlinearity:

$$h = \varphi(W_1 x + b_1)$$

$$f = W_1 h + b_2$$

# Adding a nonlinearity

If  $\varphi(u) = \text{ReLU}(u) = \max(u, 0)$  then this just becomes

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_1} &= \mathbb{1}(h > 0) \frac{\partial \mathcal{L}}{\partial h} x^T \\ &= \mathbb{1}(h > 0) W_2^T \frac{\partial \mathcal{L}}{\partial f} x^T \\ &= \mathbb{1}(h > 0) W_2^T (f - y) x^T\end{aligned}$$

where

$$\mathbb{1}(u) = \begin{cases} 1 & u > 0 \\ 0 & \text{otherwise} \end{cases}$$

See notes on backpropagation for details

# Classification

For classification we use softmax to compute probabilities

$$(p_1, p_2, p_3) = \frac{1}{e^{f_1} + e^{f_2} + e^{f_3}} (e^{f_1}, e^{f_2}, e^{f_3})$$

The loss function is

$$\mathcal{L} = -\log P(y | x) = \log (e^{f_1} + e^{f_2} + e^{f_3}) - f_y$$

So, we have

$$\frac{\partial \mathcal{L}}{\partial f_k} = p_k - \mathbb{1}(y = k)$$

## **4: Demos**

# Interactive examples

`https://playground.tensorflow.org/`

# What's going on?

- These models are curiously robust to overfitting
- Why is this?
- Some insight: Kernels and double descent

Next time!



# Summary

- Neural nets are layered linear models with nonlinearities added
- Trained using stochastic gradient descent with backprop
- Can be automated to train complex networks (with no math!)