S&DS 365 / 565

Intermediate Machine Learning

Kernels and Neural Networks

September 19

Reminders

- Assignment 1 out; due September 28 (week from this Wed)
- Quiz 2 posted Wednesday, material up to today
- Check Canvas/EdD for office hours—please join us!

Today: Neural nets

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

1: Mercer kernel recap

2: Neural net basics

Starting with regression

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

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

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

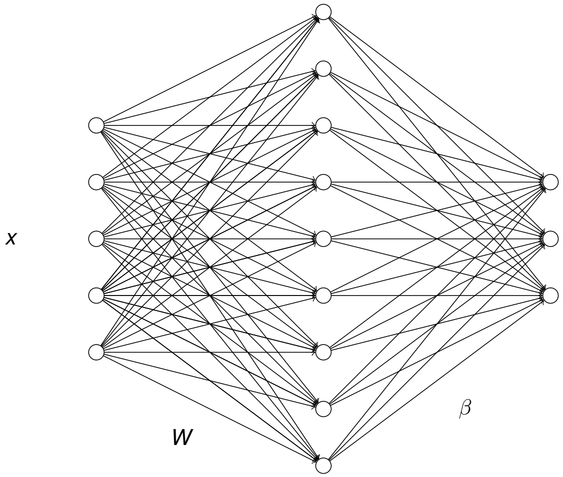
Adding a layer

Loss is

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

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

This can be viewed graphically.


x
W
β

Equivalent to linear model

But this is just a linear model

$$f = \widetilde{\beta}^T x + \widetilde{\beta}_0$$

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

Need to add *nonlinearities*

Nonlinearities

Add nonlinearity

$$h = \phi(Wx + b)$$

applied component-wise.

For regression, the last layer is just linear:

$$f = \beta^T h + \beta_0$$

Nonlinearities

Commonly used nonlinearities:

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

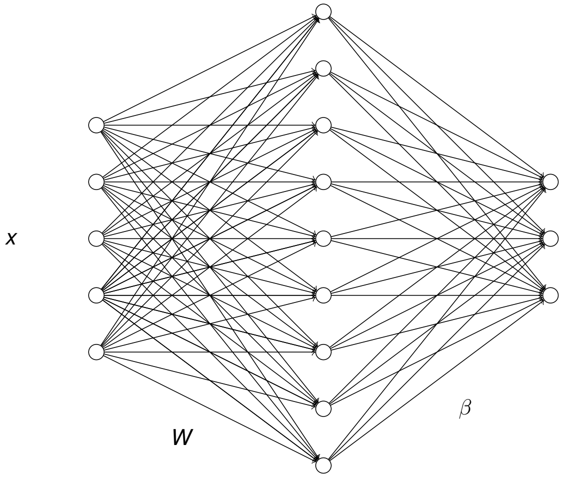
$$\phi(u) = \text{sigmoid}(u) = \frac{e^u}{1 + e^u}$$

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

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)

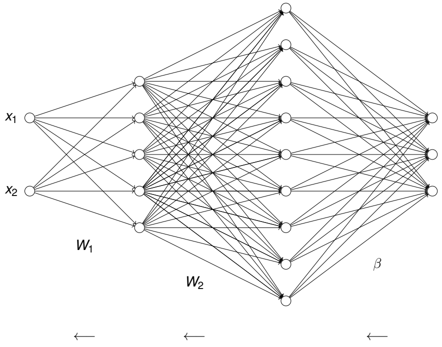


3: Backprop

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:

$$\begin{aligned} f &= W_2 h + b_2 \\ h &= W_1 x + b_1 \end{aligned}$$

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 $\varphi(u) = ReLU(u) = \max(u, 0)$ 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

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

The loss function is

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

So, we have

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

4: Demos

Demos

```
https://colab.research.google.com/github/YData123/
sds265-fa21/blob/master/demos/neural-nets/
neural-nets-regress.ipynb
```

```
https://colab.research.google.com/github/YData123/
sds265-fa21/blob/master/demos/neural-nets/neural-nets.
ipynb
```

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

5: Kernels and double descent

Neural tangent kernel

There is a kernel view of neural networks that has been useful in understanding the dynamics of stochastic gradient descent for neural networks.

This is based on something called the *neural tangent kernel (NTK)*

Parameterized functions

Suppose we have a parameterized function $f_\theta(x) \equiv f(x;\theta)$

Almost all machine learning takes this form — for classification and regression, these give us estimates of the regression function

For neural nets, the parameters θ are all of the weight matrices and bias (intercept) vectors across the layers.

Feature maps

Suppose we have a parameterized function $f_\theta(x) \equiv f(x;\theta)$

We then define a *feature map*

$$x \mapsto \varphi(x) = \nabla_\theta f(x;\theta) = \begin{pmatrix} \frac{\partial f(x;\theta)}{\partial \theta_1} \\ \frac{\partial f(x;\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(x;\theta)}{\partial \theta_p} \end{pmatrix}$$

This defines a Mercer kernel

$$K(x,x') = \varphi(x)^T \varphi(x') = \nabla_\theta f(x;\theta)^T \nabla_\theta f(x';\theta)$$

NTK and SGD

- The NTK has been used to study the dynamics of stochastic gradient descent
- Upshot: As the number of neurons in the layers grows, the parameters in the network barely change during training, even though the training error quickly decreases to zero

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!)
- Kernel connection: NTK
- Minimal norm linear model, closed form
- Key to understanding risk properties: Double descent