Deep Learning Systems: Algorithms and Implementation

“Manual” Neural Networks

Tim Dettmers (this time) and Tianqi Chen
Carnegie Mellon University

Outline

From linear to nonlinear hypothesis classes

Neural networks

Backpropagation (i.e., computiing gradients)

Outline

From linear to nonlinear hypothesis classes

Neural networks

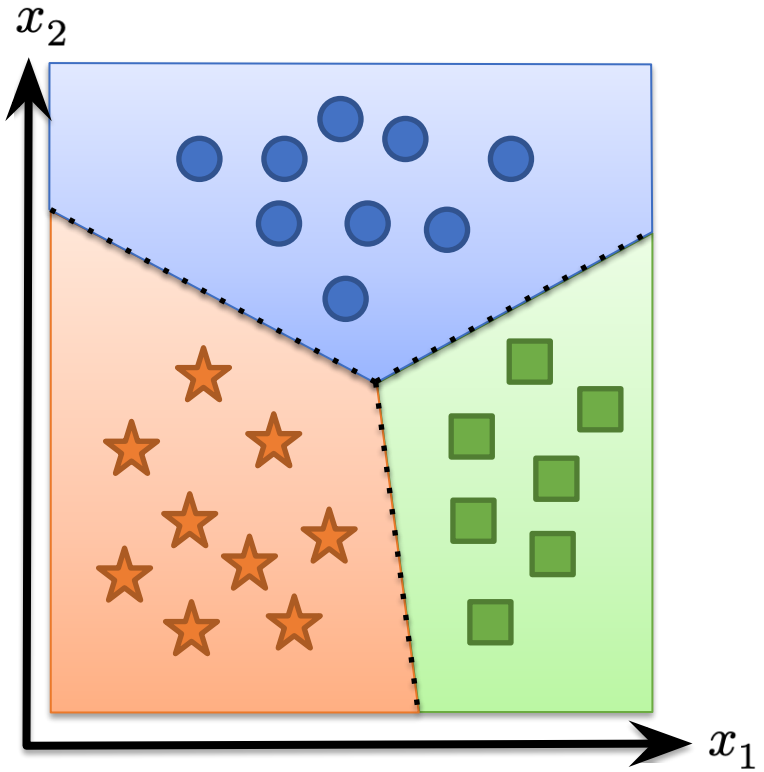
Backpropagation (i.e., computiing gradients)

The trouble with linear hypothesis classes

Recall that we needed a hypothesis function to map inputs in $\mathbb{R}^n$ to outputs (class logits) in $\mathbb{R}^k$, so we initially used the linear hypothesis class

$$h_\theta(x) = \theta^T x, \qquad \theta \in \mathbb{R}^{n \times k}$$

This classifier essentially forms k linear functions of the input and then predicts the class with the largest value: equivalent to partitioning the input into k linear regions corresponding to each class



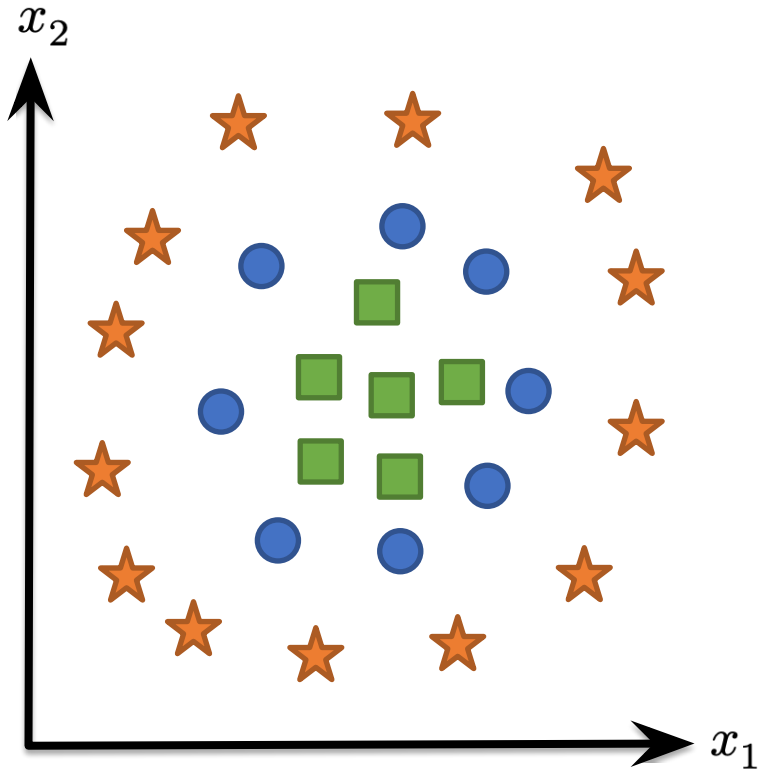
What about nonlinear classification boundaries?

What if we have data that cannot be separated by a set of linear regions?

We want some way to separate these points via a nonlinear set of class boundaries

One idea: apply a linear classifier to some (potentially higher-dimensional) *features* of the data

$$h_\theta(x) = \theta^T \phi(x)$$
$$\theta \in \mathbb{R}^{d \times k}, \phi: \mathbb{R}^n \rightarrow \mathbb{R}^d$$



How do we create features?

How can we create the feature function ϕ?

1. Through manual engineering of features relevant to the problem (the “old” way of doing machine learning)
2. In a way that itself is learned from data (the “new” way of doing ML)

First take: what if we just again use a linear function for ϕ?

$$\phi(x) = W^T x$$

Doesn’t work, because it is just equivalent to another linear classifier

$$h_\theta(x) = \theta^T \phi(x) = \theta^T W^T x = \tilde{\theta} x$$

Nonlinear features

But what does work? … essentially *any* nonlinear function of linear features

$$\phi(x) = \sigma(W^T x)$$

where $W \in \mathbb{R}^{n \times d}$, and $\sigma: \mathbb{R}^d \to \mathbb{R}^d$ is essentially *any* nonlinear function

Example: let W be a (fixed) matrix of random Gaussian samples, and let σ be the cosine function $\Longrightarrow$ “random Fourier features” (work great for many problems)

But maybe we want to train W to minimize loss as well? Or maybe we want to compose multiple features together?

Outline

From linear to nonlinear hypothesis classes

Neural networks

Backpropagation (i.e., computiing gradients)

Neural networks / deep learning

A *neural network* refers to a particular type of hypothesis class, consisting of **multiple, parameterized differentiable functions (a.k.a. "layers") composed together in any manner to form the output**

The term stems from biological inspiration, but at this point, literally any hypothesis function of the type above is referred to as a neural network

"Deep network" is just a synonym for "neural network," and "deep learning" just means "machine learning using a neural network hypothesis class" (let's cease pretending that there is any requirements on depth beyond "just not linear")

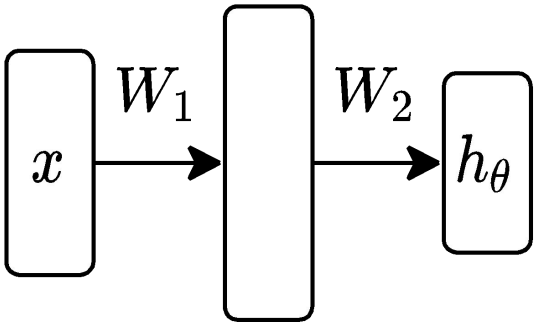
- But it's also true that modern neural networks involve composing together a *lot* of functions, so "deep" is typically an appropriate qualifier

The “two layer” neural network

We can begin with the simplest form of neural network, basically just the nonlinear features proposed earlier, but where both sets of weights are learnable parameters

$$h_\theta(x) = W_2^T \sigma(W_1^T x)$$
$$\theta = \{W_1 \in \mathbb{R}^{n \times d}, W_2 \in \mathbb{R}^{d \times k}\}$$

where $\sigma: \mathbb{R} \to \mathbb{R}$ is a nonlinear function applied *elementwise* to the vector (e.g. sigmoid, ReLU)



Written in batch matrix form

$$h_\theta(X) = \sigma(XW_1)W_2$$

Universal function approximation

Theorem (1D case): Given any smooth function $f: \mathbb{R} \to \mathbb{R}$, closed region $\mathcal{D} \subset \mathbb{R}$, and $\epsilon > 0$, we can construct a one-hidden-layer neural network $\hat{f}$ such that

$$\max_{x \in \mathcal{D}} \left| f(x) - \hat{f}(x) \right| \leq \epsilon$$

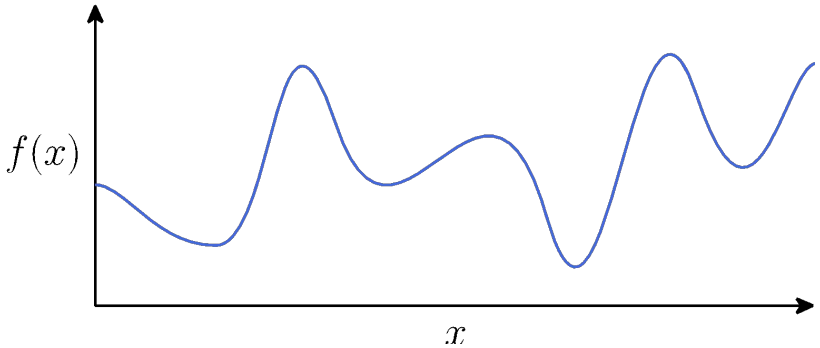
Proof: Select some dense sampling of points $\left(x^{(i)}, f\left(x^{(i)}\right)\right)$ over $\mathcal{D}$. Create a neural network that passes exactly through these points (see next slide). Because the neural network function is piecewise linear, and the function f is smooth, by choosing the $x^{(i)}$ close enough together, we can approximate the function arbitrarily closely.

Universal function approximation

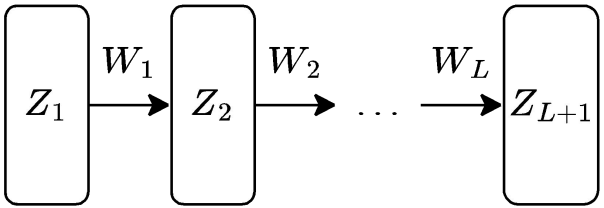
Assume one-hidden-layer ReLU network (w/ bias):

$$\hat{f}(x) = \sum_{i=1}^{d} \pm \max\{0, w_i x + b_i\}$$

Visual construction of approximating function.



Fully-connected deep networks



A more generic form of a L-layer neural network – a.k.a. "Multi-layer perceptron" (MLP), feedforward network, fully-connected network – (in batch form)

$$Z_{i+1} = \sigma_i(Z_i W_i), i = 1, \ldots, L$$
$$Z_1 = X,$$
$$h_\theta(X) = Z_{L+1}$$
$$[Z_i \in \mathbb{R}^{m \times n_i},\ W_i \in \mathbb{R}^{n_i \times n_{i+1}}]$$

with nonlinearities $\sigma_i : \mathbb{R} \to \mathbb{R}$ applied elementwise, and parameters

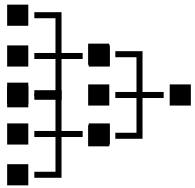
$$\theta = \{W_1, \ldots, W_L\}$$

(Can also optionally add bias term)

Why deep networks?

They work like the brain!

… no they don't!



Deep circuits are provably more efficient!

… at representing functions neural networks cannot actually learn (e.g. parity)!

Empirically it seems like they work better for a fixed parameter count!

… okay!

Outline

From linear to nonlinear hypothesis classes

Neural networks

Backpropagation (i.e., computiing gradients)

Neural networks in machine learning

Recall that neural networks just specify one of the "three" ingredients" of a machine learning algorithm, also need:

- Loss function: still cross entropy loss, like last time
- Optimization procedure: still SGD, like last time

In other words, we still want to solve the optimization problem

$$\underset{\theta}{\text{minimize}} \ \frac{1}{m}\sum_{i=1}^{m} \ell_{ce}(h_\theta(x^{(i)}), y^{(i)})$$

using SGD, just with $h_\theta(x)$ now being a neural network

Requires computing the gradients $\nabla_\theta \ell_{ce}(h_\theta(x^{(i)}), y^{(i)})$ for each element of θ

The gradient(s) of a two-layer network

Let's work through the derivation of the gradients for a simple two-layer network, written in batch matrix form, i.e.,

$$\nabla_{\{W_1,W_2\}} \ell_{ce}(\sigma(XW_1)W_2, y)$$

The gradient w.r.t. W_2 looks identical to the softmax regression case:

$$\frac{\partial \ell_{ce}(\sigma(XW_1)W_2, y)}{\partial W_2} = \frac{\partial \ell_{ce}(\sigma(XW_1)W_2, y)}{\partial \sigma(XW_1)W_2} \cdot \frac{\partial \sigma(XW_1)W_2}{\partial W_2}$$
$$= (S - I_y) \cdot (\sigma(XW_1))$$
$$[S = \text{normalize}(\exp(\sigma(XW_1)W_2)]$$

so (matching sizes) the gradient is

$$\nabla_{W_2} \ell_{ce}(\sigma(XW_1)W_2, y) = \sigma(XW_1)^T(S - I_y)$$

The gradient(s) of a two-layer network

Deep breath and let's do the gradient w.r.t. W_1…

$$\frac{\partial \ell_{ce}(\sigma(XW_1)W_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(\sigma(XW_1)W_2, y)}{\partial \sigma(XW_1)W_2} \cdot \frac{\partial \sigma(XW_1)W_2}{\partial \sigma(XW_1)} \cdot \frac{\partial \sigma(XW_1)}{\partial XW_1} \cdot \frac{\partial XW_1}{\partial W_1}$$
$$= (S - I_y) \cdot (W_2) \cdot \left(\sigma'(XW_1)\right) \cdot (X)$$

and so the gradient is

$$\nabla_{W_1} \ell_{ce}(\sigma(XW_1)W_2, y) = X^T \left((S - I_y)W_2^T \circ \sigma'(XW_1)\right)$$

where $\circ$ denotes elementwise multiplication

… having fun yet?

Backpropagation "in general"

There *is* a method to this madness … consider our fully-connected network:

$$Z_{i+1} = \sigma_i(Z_i W_i), \qquad i = 1, \dots, L$$

Then (now being a bit terse with notation)

$$\frac{\partial \ell(Z_{L+1}, y)}{\partial W_i} = \underbrace{\frac{\partial \ell}{\partial Z_{L+1}} \cdot \frac{\partial Z_{L+1}}{\partial Z_L} \cdot \frac{\partial Z_{L-1}}{\partial Z_{L-2}} \cdot \dots \cdot \frac{\partial Z_{i+2}}{\partial Z_{i+1}}}_{G_{i+1} = \frac{\partial \ell(Z_{L+1}, y)}{\partial Z_{i+1}}} \cdot \frac{\partial Z_{i+1}}{\partial W_i}$$

Then we have a simple "backward" iteration to compute the G_i's

$$G_i = G_{i+1} \cdot \frac{\partial Z_{i+1}}{\partial Z_i} = G_{i+1} \cdot \frac{\partial \sigma_i(Z_i W_i)}{\partial Z_i W_i} \cdot \frac{\partial Z_i W_i}{\partial Z_i} = G_{i+1} \cdot \sigma'(Z_i W_i) \cdot W_i$$

Computing the real gradients

To convert these quantities to “real” gradients, consider matrix sizes

$$G_i = \frac{\partial \ell(Z_{L+1}, y)}{\partial Z_i} = \nabla_{Z_i} \ell(Z_{L+1}, y) \in \mathbb{R}^{m \times n_i}$$

so with “real” matrix operations

$$G_i = G_{i+1} \cdot \sigma'(Z_i W_i) \cdot W_i = \Big(G_{i+1} \circ \sigma'(Z_i W_i)\Big) W_i^T$$

Similar formula for actual parameter gradients $\nabla_{W_i} \ell(Z_{L+1}, y) \in \mathbb{R}^{n_i \times n_{i+1}}$

$$\frac{\partial \ell(Z_{L+1}, y)}{\partial W_i} = G_{i+1} \cdot \frac{\partial \sigma_i(Z_i W_i)}{\partial Z_i W_i} \cdot \frac{\partial Z_i W_i}{\partial W_i} = G_{i+1} \cdot \sigma'(Z_i W_i) \cdot Z_i$$

$$\implies \nabla_{W_i} \ell(Z_{L+1}, y) = Z_i^T \Big(G_{i+1} \circ \sigma'(Z_i W_i)\Big)$$

Backpropagation: Forward and backward passes

Putting it all together, we can efficiently compute *all* the gradients we need for a neural network by following the procedure below

1. Initialize: $Z_1 = X$
   Iterate: $Z_{i+1} = \sigma_i(Z_i W_i), \quad i = 1, \dots, L$ } Forward pass
2. Initialize: $G_{L+1} = \nabla_{Z_{L+1}} \ell(Z_{L+1}, y) = S - I_y$
   Iterate: $G_i = \left(G_{i+1} \circ \sigma_i'(Z_i W_i)\right) W_i^T, \quad i = L, \dots, 1$ } Backward pass

And we can compute all the needed gradients along the way

$$\nabla_{W_i} \ell(Z_{k+1}, y) = Z_i^T \left(G_{i+1} \circ \sigma_i'(Z_i W_i)\right)$$

"Backpropagation" is just chain rule + intelligent caching of intermediate results

A closer look at these operations

What is really happening with the backward iteration?

$$\frac{\partial \ell(Z_{L+1}, y)}{\partial W_i} = \underbrace{\frac{\partial \ell}{\partial Z_{L+1}} \cdot \frac{\partial Z_{L+1}}{\partial Z_L} \cdots \cdot \frac{\partial Z_{i+2}}{\partial Z_{i+1}}}_{G_{i+1}} \cdot \frac{\partial Z_{i+1}}{\partial W_i}$$

Derivative of the ith layer

Each layer needs to be able to multiply the "incoming backward" gradient G_{i+1} by its derivatives, $\frac{\partial Z_{i+1}}{\partial W_i}$, an operation called the "vector Jacobian product"

This process can be generalized to arbitrary computation graphs: this is exactly the process of automatic differentiation we will discuss in then next lecture