Dive into Deeplearning - Chapters 1→5

Introduction

Supervised Problem

1. The data we can learn from
2. A model of how to transform the data
3. Um objective function that quantifies how well (or badly) the model is doing
4. Um algorithm to adjust the model's parameters to optimize the objetive function

Preliminaries

→ Data manipulation

torch.arange(12)

x.shape

torch.numel(M) = total number of elements in M

x.sum(↑ if KEEPDIMS=True mantém dimensão de vetor unitário

x.numpy()

torch.from_numpy(M)

torch.mv(A, x) = matrix vector multiplication

torch.mm(A, B) = matrix matrix multiplication

torch.norm()

→ Linear Algebra

B = A.clone()

→ Probability

from torch.distributions import multinomial

My playlist today: #

Linear Regression

$\hat{y} = Xw + b$ linear model

Loss function = a measure of fitness

$$l(w,b) = \frac{1}{2}(\hat{y} - y)^2$$

When training we want to find the parameters (w^*, b^*) that minimize total loss across all training examples

$$w^*, b^* = \underset{w,b}{\operatorname{argmin}}\, l(w,b)$$

* Analytic solution: too much restrictive, computational cost
* Minibatch

The most naive application of gradient descent consist of taking the derivative of the loss function, which is an average of all losses computed of every single update → too slow often sample a minibatch

$$w = w - \eta \partial_w l(w)$$

Minibatch $\mathcal{B}$

$$(w,b) \leftarrow (w,b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(w,b)} l^{(i)}(w,b)$$

Concise implementation

From torch import nn

nn.Sequencial = pipeline of nn networks

nn.Linear(2, 1)

in dimension → out dimension

net[i] → layer i
net[0].weight, net[0].bias
nn.MSELoss()
trainer = torch.optim.SGD

Softmax regression

Regression = how much/how many?
Classification = which one

classification
- ↗ hard assignment: categories only
- ↳ soft assignment: also class probs

classification treated as one-hot encoding
$y \in \{(1,0,0), (0,1,0), (0,0,1)\}$
dog y_1 cat y_2 chicken y_3

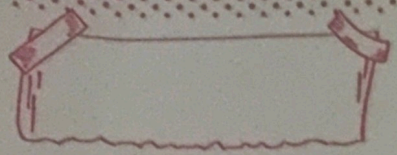
Three logits o_1, o_2, o_3 for each input

$o_1 = x_1 w_{11} + \ldots + x_n w_{1n} + b_1$
$o_2 = x_2 w_{21} + \ldots + x_n w_{2n} + b_2$
$o_3 = x_3 w_{31} + \ldots + x_n w_{3n} + b_3$

$o = wx + b$

Just as linear regression softmax is a single layer neural network and fully connected

You might be tempted to suggest that we interpret the logits "o" directectely as our outputs




of interest. However, there are some problems with directly interpreting the output of linear layers as probabilities; they also don't sum up to 1, can be negative. This violates the laws of probability

$$\hat{y} = \text{softmax}(o) \text{ where } \hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$$

$$\underset{j}{\text{argmax}}\ \hat{y}_j = \underset{j}{\text{argmax}}\ o_j$$

Note that the softmax operation does not change the ordering among the logits o, which are simply the pre-softmax values that determine the probabilities assigned to each class.

Vectorization of minibatch X

$$O = XW + b$$
$$\hat{Y} = \text{softmax}(O)$$

Loss function = log likelihood

$$P(Y, X) = \prod_{i=1} P(y^{(i)} \mid x^{(i)})$$

Maximum likelihood = minimize negative likelihood

$$l(y, \hat{y}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j$$

The image classification dataset

One of the widely used for image classification is the

MNIST dataset. While it had a good run as a benchmark dataset, even simple models by today's standards achieve classification accuracy over 95%, making it unsuitable for distinguishing between stronger models and weaker ones. Today, MNIST serves as more of sanity check than benchmark

Multilayer Perceptron

Linearity implies the weaker assumption of monotonicity
↑ feature ↑ model output

We can overcome these limitations of linear models and handle a more general class of functions by incorporating one or more hidden layers

> Multilayer Perceptron - MLP

Stack many fully connected layers one top of each other

$$H = \sigma(XW^{(1)} + b^{(1)}) \rightarrow \text{activation function}$$
$$O = HW^{(2)} + b^{(2)}$$

1. ReLu : Rectified Linear Unit

$$ReLU(x) = \max(x, 0)$$

2. $$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

My playlist today. #

3. $\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$

nn.Parameter
torch.zeros_like(matrix)

underfit ↓ train eval ↓ test eval
overfit ↑ train eval ↓ test eval

Weight decay
Limiting the number of the features is a popular technique to reduce overfitting.

Weight decay (commonly called L2 regularization)

Dropout

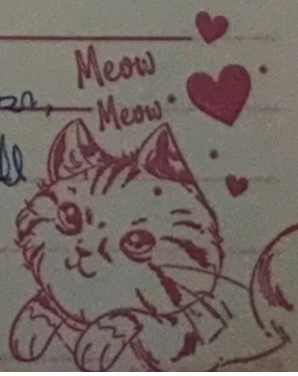
Faced with more features than examples, linear models tend to overfit. But given more examples than features, we can generally count on linear models not overfit

Unfortunately, the reliability with which linear models generalize come at cost. Naively applied, linear models do not take in account interactions among features. For every feature, a linear model must assign either a positive or a negative weight, ignoring context

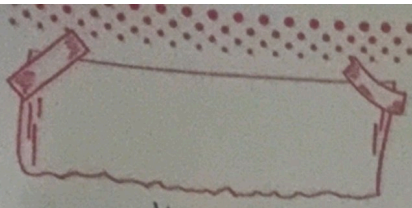
In traditional texts, this fundamental tension between generalizability and flexibility is described as bias-variance trade-off. Linear models have high bias, they can only represent a small class of functions. However, those models have low variance: they give similarity results across different samples of data

Deep neural networks inhabit the opposite end of the bias-variance spectrum. Unlike linear models, neural network are not confined to looking each feature individually Even when we have far more examples than features, neural network are capable of overfitting

Another useful notion of simplicity is smoothness, i.e., that function should not be sensitive to small change to its input



My playlist today. #




smoothness → dropout = support noise in examples
nn.Dropout

Propagation and computational graphs

Forward propagation (or forward pass) refers to the calculation and storage of intermediate (including output) for a neural network in order from the input layer to the output layer

Plotting computational graphs help us visualize the dependencies of operations and variables within calculations.

Backpropagation refers to the method of calculating the gradient of neural network parameters. In short, the method traverses the network in reverse order from the output to the input layer, according to the chain rule in calculus

~~Layers and blocks~~

nn.State_dict
nn.init.constant_(matrix, a)
nn.init.zeros_(matrix)