Contents:

1. What is learning a function by optimization?
2. What are feed-forward neural networks?
3. How are they applied?
4. How are they trained?
5. How does PyTorch help in function learning with neural networks?
6. What alternatives exist?

Note. code examples `torch.nn`.

Let $f: \mathbb{X} \longrightarrow \mathbb{Y}$ be a function we want to learn from observations $\mathscr{D} \in \mathbb{X} \times \mathbb{Y}$

x	$y \simeq f(x)$
x_1	y_1
$\vdots$	$\vdots$
x_n	y_n

The function f can be learned from (noisy) observations $\mathscr{D}$ by optimization if

1. we have a suitably expressive function class $\mathscr{F} := \{f(x;\theta) | \theta \in \Theta\}$ such that $f(x) \simeq f(x;\theta^\star)$ for some parameter (vector) $\theta^\star$ and some quality criterion "$\simeq$".
2. we have a practicable algorithm $A: (\mathscr{D}, \mathscr{F}, \mathscr{H}) \longrightarrow \theta^\star$.

$\longrightarrow$ rich model class, quality criterion and practical inference algorithm $\longleftarrow$

rich model class. (deep) feedforward neural networks

quality criterion. low *test* error: empirical risk minimization + regularization

learning algorithm. descent along gradients calculated by backpropagation

Feedforward neural networks became useful function approximators with concurrent, often interdependent improvements in

- expressivity (depth, conv/pool layers)
- regularization (dropout, early stopping, explicit cost penalties…)
- inference speed/stability (activations, gradient rules, backpropagation…).

networks. graphical expression of function composition

feedforward. no cycles – network is a DAG

neural. each node calculates $h_i = g(x^\top W_{:,i} + c_i)$. The linear combination followed by nonlinear threshold/saturation function resembles a *very* stylized neuron

deep. more layers facilitate training of very expressive networks

`torch.nn.[Conv2d|Linear|Dropout2d|...]`,
`torch.nn.functional.[max_pool2d,|relu|...]`

Note: $H = g(XW + c)$ batch matrix multiplication $\rightarrow$ leverage GPU acceleration

`torch.[.cuda].Tensor`, `a_tensor.to_device(<dev>)`

As our functions are parameterised, we turn this into an optimisation problem:

$$\theta^\star = \text{argmin}_\theta \text{cost}(f(x;\theta), y)$$

Two terms contribute to the cost:

i. loss: penalizes bad predictions, i.e. some idea of *training error*

ii. regularization: penalizes complex f, leading to *generalization error*

→ this is the difference between pure optimization (fit = minimize training error) and a machine *learning* algorithm.

`torch.nn.*loss`

- Find how the cost J depends on each of the parameters θ_i

  $\rightarrow$ *find* gradients, (reverse) differentiation $\rightarrow$ backpropagation

- and adjust the parameters to minimize it

  $\rightarrow$ *use* gradients, learning rule

$$\theta^{[k]} = \theta^{[k-1]} - \eta \nabla_\theta J(\theta), \qquad \eta \in \mathbb{R}^+$$

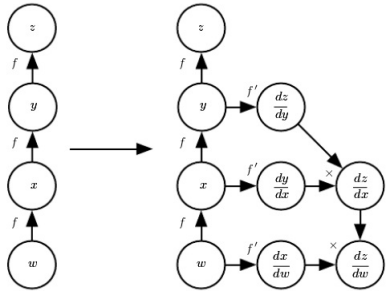
- only local minima – but some of these surprisinly useful in NNs
- since J usually additive on samples, i.e. $J = J_i$ we can perform gradient updates with fixed batch size $\ll$ dataset size $\rightarrow$ *minibatches* (uniform sampling)
- Adadelta, Adagrad, Adam, momentum, RMSprop…

`torch.optim.*`

- gradient descent may need or benefit from learning rate adaptation

`torch.optim.lr_scheduler.*`

Where do the gradients come from?




```
torch.Tensor(..., require_grad=True)
```

```
a_tensor.backward()
```

```
torch.autograd.
```

- dynamic computation graphs vs. declaration/execution phases
- distributed, multimachine training → `torch.distributed`
- C++ model serving → tensorflow.serving
- checkpointing → `torch.utils.checkpoint`
- monitoring / debugging / optimizing → tensorflow.tensorboard
- model porting (to production / other frameworks) → ONNX
- fast forward mode embedded / mobile → tensorflow lite

Alternatives

- caffe 2 (production, mobile) → PyTorch 1.0
- tensorflow: docs, community, tensorboard, serving, lite
- keras (tf, cntk, theano): prototyping, spark,
- mxnet: language agnostic
- cntk: spark, azure