Lecture 2 (?)

Now just need a family of fit fns $f(x;\theta)$

→ this is where deep NNs come in!

→ extremely expressive family of fns "universal approximating fns"

→ scale well to very high dims

→ generalize unreasonably well beyond training data

$x \in \mathbb{R}^d$ $d \sim 10^3$, 10^4, ...

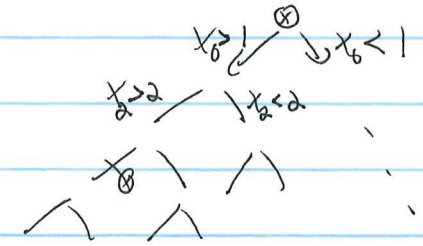
~~"deep learning"~~

CAN CUT

Aside

"shallow ML"

→ previous standard for ML now field: BDTs



↓ primary for classification

NNs much broader range of applications

ensemble & "boost" many weak classifiers in the form of decision trees

- limited to small # of features
- not as expressive as NNs so generally subopt
- more interpretable (just ~~its~~)

NNs → "deep learning" vs BDTs "shallow ML"

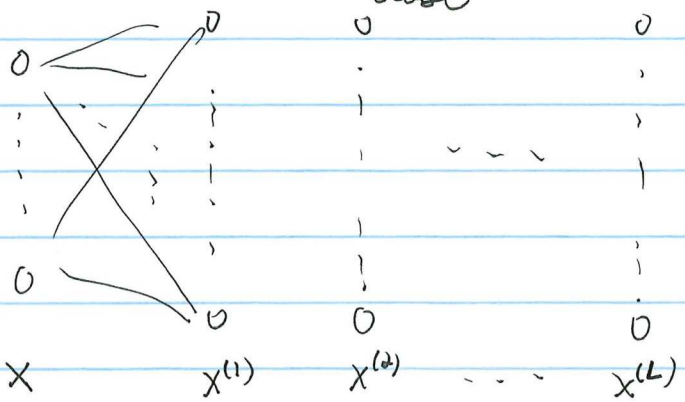
CAN CUT

Simplest NN: "feed-forward"

"MLP"

"DNN" "fully connected"

dense

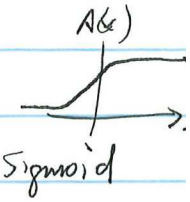


input features — hidden layers — outputs

$$x^{(1)} = A^{(1)}\left(W^{(1)}x + b^{(1)}\right)$$

activation → nonlinearity — weights — biases

$$A(x) = \frac{e^x}{1+e^x}$$



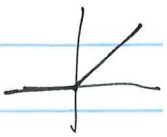
vanishing gradients →

$\tanh x$

$\{W, b\} \to \theta$

NN trainable pars

most popular, "last" choice → ReLU $\begin{cases} 0 & x<0 \\ x & x>0 \end{cases}$



$$x^{(n)} = A^{(n)}\left(W^{(n)}x^{(n-1)} + b^{(n)}\right)$$

[Note: A acts elementwise]

$$z = A^{(L+1)}\left(W^{(L+1)}x^{(L)} + b^{(L+1)}\right)$$

usually pick $A^{(1)} \ldots A^{(L)}$ ReLU

→ output $A^{(L+1)}$ diff. depending on application

Ex: binary classification

- This defines a NN architecture

  Many "hyperparameters" → need to optimize wrt hyperparam

  - L: # of hidden layers
  - $n_h^{(l)} = \dim x^{(l)}$: # of hidden nodes in layer l
  - choice of activations
  - ⋮

- DNN basically mtx mult + nonlinearity → GPUs optimized for this!

  Deep learning on GPU → major breakthrough ~2010?

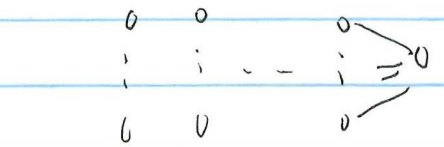
- typical DNN for classif. could start w/ 100 or 1000 (100 or 1000) inputs

  ex w/ 10^5 hidden nodes

  #pars ~

  might have

  Ex: 100 hidden nodes and ~5 layers

  ~10^4 weights ×5 ~ 50,000 parameters

  can easily get to 1M pars or 10M pars

  ↓

  still trainable on GPU

Training NNs

- How to fit $f(x;\theta)$ w/ 50k or 1M θ's to date??

No time to go into any detail. Just focus on rough, key concepts

— Most obvious approach: Newton method

$\theta_0 = \theta_0 + \delta\theta$ ← initial guess

$$L(\theta) = \sum_{i \in \text{data}} \mathcal{L}(f(x_i;\theta))$$

$$\approx L(\theta_0) + (\theta - \theta_0)\frac{\partial L}{\partial \theta}(\theta_0) + \frac{1}{2}(\theta-\theta_0)^2 \frac{\partial^2 L}{\partial \theta^2}$$

problem: the θ is 1M dim'l

→ cannot invert 1M x 1M Hessian mtx!

$$\frac{\partial L}{\partial \theta} + \delta\theta \frac{\partial^2 L}{\partial \theta^2} + \cdots = 0$$

$$\delta\theta = \frac{\partial L}{\partial \theta} \Big/ \frac{\partial^2 L}{\partial \theta^2}$$

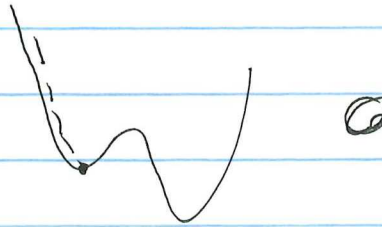
- Instead NNs are optimized w/ 1st order methods

"Gradient descent"

"Gradient update" ← $\delta\theta = -\eta \frac{\partial L}{\partial \theta}$

"learning rate" — hyperparameter

Hope that $\frac{\partial L}{\partial \theta}$ at least points in right direction



- Problem: can get stuck in local minima

Soln: Stochastic gradient descent: introduce noisy gradient

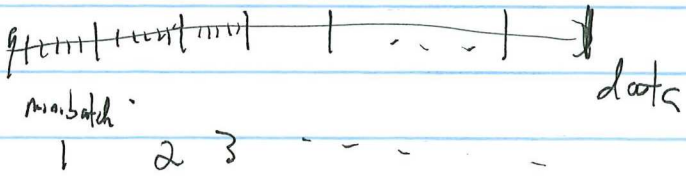
- don't calc $\frac{\partial L}{\partial \theta}$ w/ full data set

$$\frac{\partial L}{\partial \theta} = \sum_{i=1}^{N} \frac{\partial \ell}{\partial \theta}(f(x_i;\theta)) = N\left\langle \frac{\partial \ell}{\partial \theta} \right\rangle$$

$\frac{\partial \ell}{\partial \theta}(f(x_i;\theta))$ evaluate on single data instance — extreme version of SGD

~~loop over data~~

evaluate on subset of data — "minibatch"



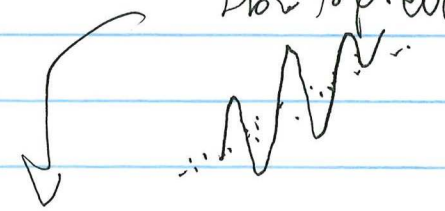
Train: loop over ~~data~~ data in minibatches → 1 epoch

~~shuffle~~ shuffle and repeat → N epochs.

- In practice people use fancier versions of SGD
  Adam, Adadelta → think of as SGD w/ adaptive learning rate γ.

- Overfitting: NNs have so many parameters

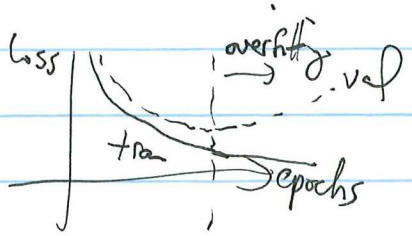
  How to prevent overfitting — or "memorizing" data



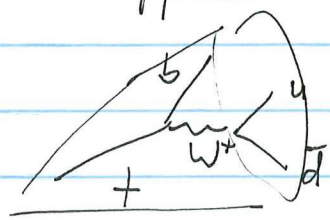
main tool: train/val/test split

Split data into

- perform SGD on train data
- check loss on val data
  stop if val loss increases
- final evaluate on test data.

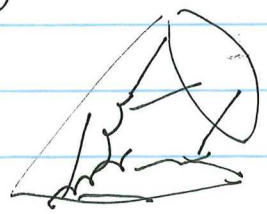


Application to HEP: Top Tagging



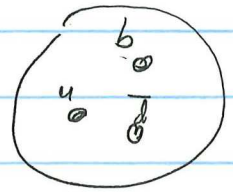
"fat jet"

vs



QCD

boosted



tops
3 prong substructure

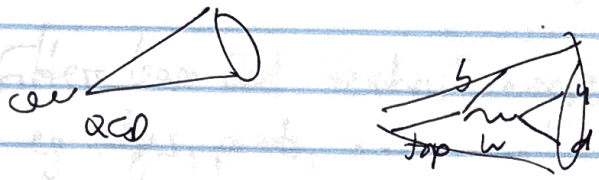
QCD
no substructure (1-prong)

Good setting for developing ML classifier methods
- may handles, rich substructure
- accurate simulations

Traditional approach: physics features ⊗ "high level features"

II. Data representations

- Work example: Boosted Top Tagging



high p_T
both are "fat" jets
but tops have a lot of substructure.

Distinguishing them is classic MLHEP task "top tagging"

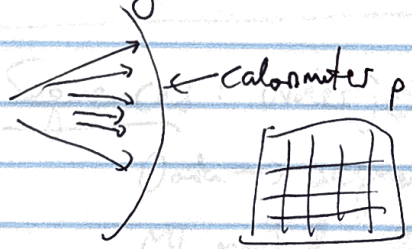
↳ walking through diff. approaches — great overview of standard data representations

"Raw" data (reduced)

$jet = \{(p_{Ti}, \eta_i, \phi_i) | i = 1, \dots N\}$
all the constituent 4-vectors
(could also include b-tagging, particle flow, ...)

i) Early days: High level features $m_t, m_W, \tau_{32}, \dots$
Data → "summary statistics"
Classify w/ BDT, ~~MLP~~, MLP, simple cuts

ii) Jet images



Σp_{Ti} in each η, ϕ pixel
→ jet image!

Data → image

New method: Convolutional NN (CNN)
instead of fully connected MLP:

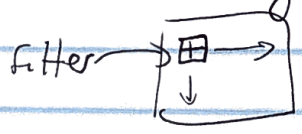


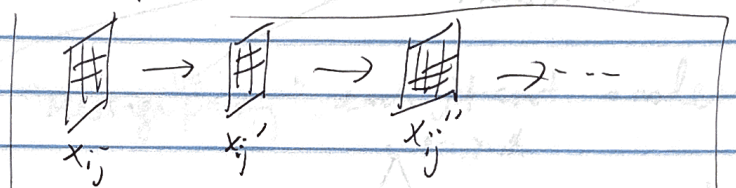
image: x_{ij} filter w_{ab} (2×2 for example)
$i = 1, \dots N_{pixels}$ $a, b = 1, \dots 2$

$x_{ij} \to A\left(\sum_{a,b} x_{i+a, j+b} w_{ab} + b\right) = x'_{ij}$

Produces hidden layer that's also an image!

Fully connected weights → 2×2 web weights of filter
$N^2_{pixel} \times N_{hidden}$ "weight sharing"

Idea: Filters learn 2d "features" in image
eg edges, points, ...
Those features can apear anywhere in image
Can repeat!



Basic structure of CNN

Also: channels "color"
pooling
:

iii) Sequences: order jet constituents somehow eg $p_{T1} > p_{T2} > \cdots$
Data → sequence
ML methods: Recursive NNs (RNNs) } idea:
LSTMs ... } weight sharing along sequence
Skip for lack of time
"in time" not 2d