Heuristics for faster training

disadvantage of NNs: take much longer to train than other methods
* for any of these solutions, cross validation/empirical analysis required

Solutions:
1) fix vanishing grad. problem
2) SGD: faster than BGD on large, redundant datasets
   - BGD: walks downhill on 1 cost fn
   - SGD: very short step downhill on one point's lost fn. and then so on...

(cost fn: sum of loss functions over all sample pts.
↳ one batch step ≃ n stochastic steps, roughly same amount of time

drawback of SGD: quick to learn redundant info, e.g. a ton of 9's in MNIST
i.e. better than BGD

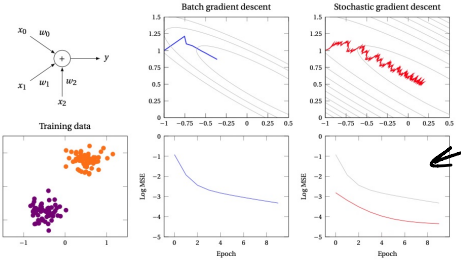


Figure 19.1: Comparison between stochastic gradient descent and batched gradient descent for a simple one-layer neural network

- one epoch presents every training pt. once
- training usually takes multiple epochs
  - if large sample size (w/ high redundancy), then training can take less than one epoch

Sol. 3) normalizing data: center each feature so $\mu = 0$, scale so $\sigma = 1$

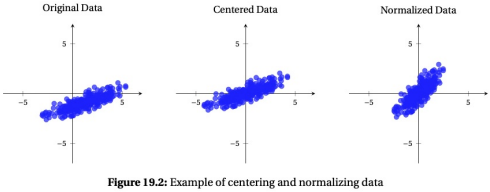


Figure 19.2: Example of centering and normalizing data

empirically, centering data → easier for hidden units to get in good operating region of Sigmoid & reLU

Scaling features → better conditioned objective fn. → gradient descent converges faster
- Skewed data → objective fn w/ ill-conditioned/highly eccentric hessian H
  → Slow GD convergence
    Sol: Scaling or Momentum (soon)

4) "centering" the hidden units
   ↳ replace $S(\cdot)$ w/ $\tanh r = \frac{e^r - e^{-r}}{e^r + e^{-r}} = 2s(2r) - 1$
   ↳ now, range of $(-1, 1)$ instead of $(0, 1)$

   - new derivative: $S' = 1 - \tanh^2 r$
   - good output target values: $(-0.7, 0.7)$

5) different learning rates for each layer of weights
   - earlier layers have smaller gradients → larger learning rates

6) emphasizing schemes: have network learn/emphasize uncommon examples before the redundant & common examples
- present examples from rare classes more often or w/ bigger LR
  - can do same for misclassified examples

Warning: backfires if there are bad outliers

7) 2nd-order optimization
- Newton's method is impractical since Hessian too large & expensive to compute

nonlinear conjugate gradient method: Works well for small networks w/ small datasets using regression
- Should only be used w/ BGD since this is slow w/ redundant data

Stochastic Levenberg Marquardt: approximates a diagonal Hessian
- authors claim 3x faster convergence vs SGD

8) acceleration schemes e.g. RMSprop, Adam, AMSGrad

Heuristics for avoiding bad local minima

1) fix vanishing grad. problem
2) SGD: random motion lets you avoid bad/shallow local minima
- even if at local minima of cost fn., loss fn. of each pt. will try to push you in a different direction
3) Momentum: GD can be modified to change a velocity $\nabla \vec{w}$ slowly

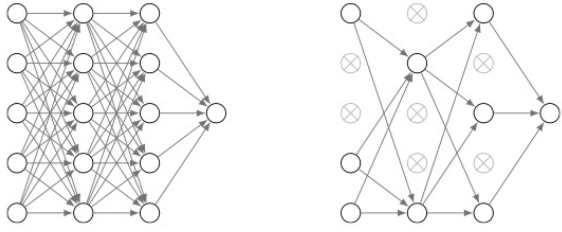
$\beta < 1$, how much momentum persists between iterations

$$\Delta \vec{w} \leftarrow -\varepsilon \nabla J(\vec{w})$$
repeat
$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$
$$\Delta \vec{w} \leftarrow -\varepsilon \nabla J(\vec{w}) + \beta \Delta \vec{w}$$

- performs well for BGD & SGD
- Choosing β: Conflicting advice
  - Some set $\beta = 0.9$, others $\beta \approx 0$
  - Hinton: start at 0.5 & increase to 0.9 as grad becomes small
  - large $\beta \rightarrow$ small LR to compensate
    - can still use large LR at first for reasonable initial velocity
- problem w/ momentum: once it finds a minimum, it oscillates around it
  - that said, more likely to find good minimum in 1st place

Heuristics to avoid overfitting:

1) ensemble of NNs w/ random initial weights & bagging
   - slow to train
2) L2 regularization/Weight decay: add $\|\vec{w}\|^2$ to cost fn
   $b \cdot \frac{dJ}{dw_i}$ has extra $-2\varepsilon\gamma w_i$ term
3) Dropout "emulates an ensemble in one network"
   - temporarily disable random subset of units during training, as well as all edges in and out of those units
   - seems good if P(hidden units) = 1/2, P(input) < P(hidden units)
   - w/ SGD, frequently change which random subset of units are disabled
   - authors claim better than L2 regularization
   - some benefits of ensemble, but faster



(a) Network without dropout (b) Network with dropout

Figure 19.3: Example of a network with and without dropout

4) fewer hidden units
   - # hidden units is a hyperparameter used to adjust B/V tradeoff
     - too few → high bias
     - too many → low B, high V

Convolutional Neural Networks

NNs are overparameterized: too many weights, not enough data

- If you have a ton of weights → need more data, slow compute

2 new properties of NN design:

- **local connectivity**: hidden unit (in earlier layers) only connect to small patch of units from the prev. layer
  - improves overparameterization problem
  - speeds up training & classification
- **shared weights**: groups of hidden units share same mask/filter/kernel
  - each mask operates on every patch of an image
  - # units in 1st hidden layer = # masks · # patches
  - if NN learns to detect edges on one patch, then every patch has an edge detector since the mask that detects edges is applied to every patch

Convolution: same linear transformation applied to different parts of input by shifting

- Shared weights: suppose gradient descent starts to develop an edge detector
  - ↳ this edge detector is being trained on every part of every image, not just one spot
    - ↳ good since edges appear on different patches of diff. images
      - ↳ loc. no longer matters

NNs: hidden units are features that we learn

CNNs: learn features from multiple patches simultaneously and then applying those features everywhere