11-785 11/05/2020

L6 - convergence in neural networks

- HW - educational, not punitive exercises (by working together)

Training a network — L5 slides — continues L5 slides S. 2019

$$L(W) = \frac{1}{N_X} \sum_X div(f(X;W), D(X))$$ ✓

$$\hat{W} = \underset{W}{argmin}\ L(W)$$ ✓

$$W_k = W_{k-1} - \eta \nabla_W L(W)^T$$ ↗ you've seen this already

(*) Gradient of total loss → average of gradients of loss for indiv instances

$$L(W) = \frac{1}{N_X} \sum_X div(f(X;W), D(X))$$

$$\nabla_W L(W) = \frac{1}{N_X} \sum_X \nabla_W div(f(X;W), D(X))$$

computed via backprop

(*) Backpropagation → an algorithm for computing derivatives

(*) Differentiable divergence is a proxy for class. error

- minimising loss is expected, but not guaranteed, to minimise class. error

(36705 analogies?)

(Issues with gradient descent)

(*) multivariate function of many param - gradient descent takes step 'against derivative' (①) (#1)

(#1): size of step/learning rate — same in every component of the param

(Blunt tool)

(*) loss function has eccentricities in different directions

2nd order methods

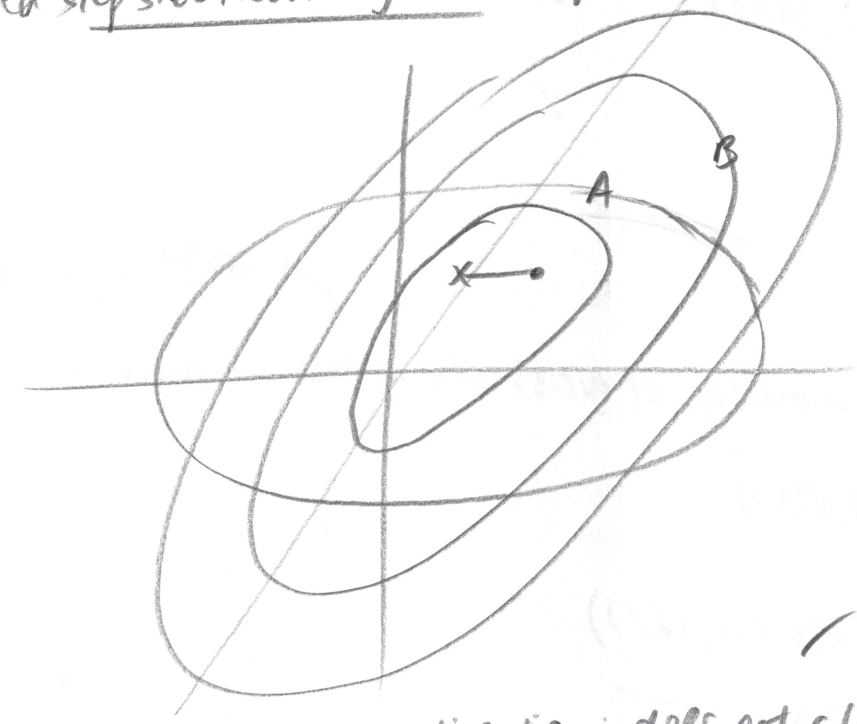
- Normalise variation along comp to mitigate diff opt. learning rates for diff comp.

- Comput. infeasible
- 2nd order approx + step - away from minimum (non-convex)

(*) See slides → (A) (AE): Review

- Release fixed step size / learning rate requirement

BR:



clarify

- In example A: moving in x-direction, does not affect y-direction (in axis aligned case)

- In example B: Have to move along recalibrated x-y axes for these contours for independence of dimensions x and y

- Heuristic illustration of fixed step size, and when it may not be a good thing
i.e.
- when there is 'dependence in direction'

- derivative inspired algos

  - Rprop
  - quickprop

- Resilient propagation

  - Adaptive to sign of derivative
  - Not considering value of deriv

(A) Accounts for overshooting
- introduce scaling factor

(*) Algorithmic explanation of Rprop
- independently for every direction

(W)(A2): Review logic
(A3): Algo
- use intuition to us.
- ensure you understood how formalised.

(*) Rprop pseudocode

(Q) does this require backprop?
- Yes; for $\frac{dErr(w_{i,j})}{dw_{i,j}}$

(BR): Much more efficient than gradient descent

- (*) NO convexity

quickprop - Scott Fahlman
- Newton updates (compute of 2nd deriv.) (empirically derived)
- instab. for non-convex objectives
- Better than Rprop (taste)
- Newton: multiply by inverse Hessian (normalis) (W)(A4): review

(*) Tying step sizes for all dimension is subopt. (S11)
- Treat each dimension separately

-Rprop, quickprop issues: -
- ignores dependence between dimensions → (W): suggests we should find a more nuanced way of addressing (S11)
- unexpect. behaviour
- slow → (W): use 16 s. 2019 slides here

(*) Close look at convergence

(*) BR in 2-dim map learning rates
↳ converges in some dirs; diverges in others
- steps; in directions of convergence (smoothly)
  - steps are pointing in same direction
- steps; in directions of divergence/oscillation
  - steps not pointing in same dir (net distance travelled given a direction is low)

/ vector quantity with dir

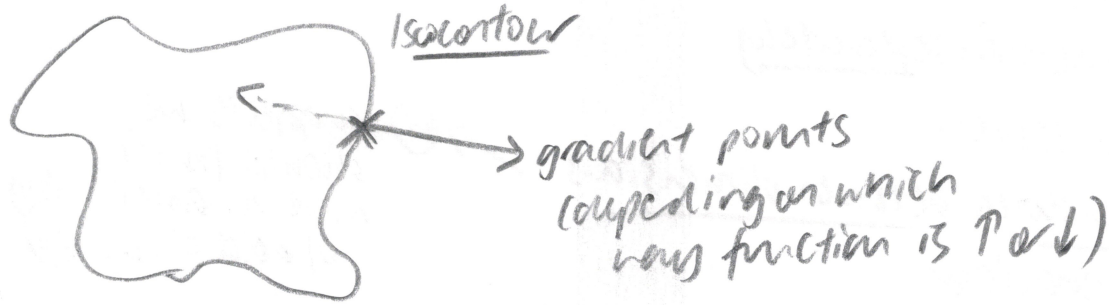
(*) Proposal - keep track of oscillations
- Bouncing around (encoded as frequent sign changes?) → decrease step size

- smooth convergence - increase step size.
- another way of seeing it is changing the nature of the feedback in for an adaptive learning rate - time series analogy kicks in?

(*) momentum methods
- maintain running av. of all past steps
(w):
- can encode differences between smooth conv / oscillation / divergence
- smooth conv → running average will have large value
- oscill / divergence → running average will be small (cancellation of the and -ve swings) (eliminate fluctuations, reduce step size)

- update with running average, rather than exact gradient.

(*) momentum update
- BR: Remember gradient is orthogonal to level set

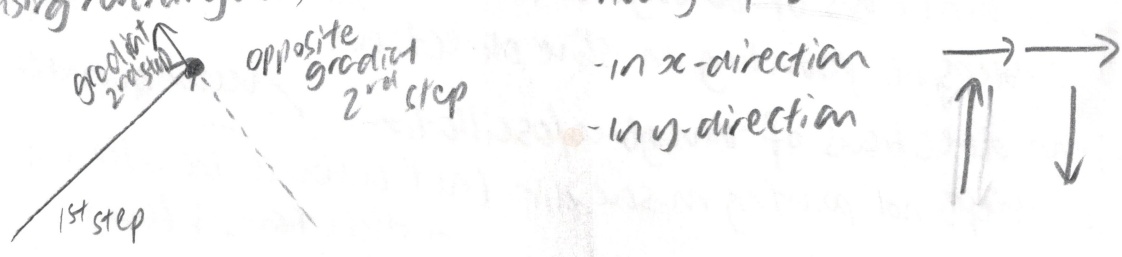


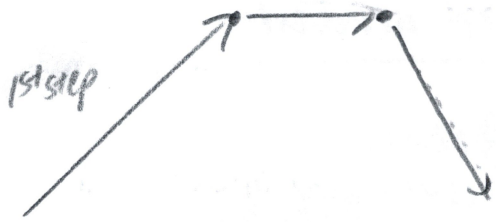
(*) Plain gradient update
- BR: "Gradient always pointing immediately downwards (rather than towards centre of ellipse)"
- BR: "Top-bottom (y-direction) - oscillating back and forth
left-right (x-direction) - moving gradually rightwards"

Final update is average - momentum intuit.

(*) Example (using running av.)
Heuristic
- average of 2
- in x-direction
- in y-direction

step 1 step 2





under momentum (using running average of gradient)

(W)(A3): Review - check you understand heuristically

(*) $\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err(W^{(k-1)})$ → (?) (A4)

$W^{(k)} = W^{(k-1)} + \Delta W^{(k)} \Rightarrow W^{(k)} - W^{(k-1)} = \Delta W^{(k)}$

(W): distinguish layer subscripts and iteration no. superscripts

---

(*) Gradient descent / training WITH momentum

- initialise all weights: $W_1, W_2, \dots, W_K$
- Do
  - $\forall$ layers k, init $\nabla_{W_k} Err = 0$ $\Delta W_k = 0$
  - $\forall$ $t = 1:T$ (training instances)
    - $\forall$ every layer k
      - compute gradient $\nabla_{W_k} Div(Y_t, d_t)$
      - $\nabla_{W_k} Err \mathrel{+}= \frac{1}{T} \nabla_{W_k} Div(Y_t, d_t)$
  - $\forall$ every layer k
    - $\Delta W_k = \beta \Delta W_k - \eta \nabla_{W_k} Err$
    - $W_k = W_k + \Delta W_k$

(?) (W) (A5): algorithmic flow

(*) Refer to previous, as no iteration superscripts.

- momentum update

- (W) - Analyse intuitively the eq:-

$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err$ (update)

- At every iteration: to compute current step:-
  1. compute gradient at current loc ($\nabla_W Err(W^{(k-1)})$)
  2. Add in scaled previous step (running average) ($\beta \Delta W^{(k-1)}$)

(*) (A6): check you understand this in the hill-climbing language ✓

- use hill-climbing analogy :-
- Reverse order of operation → Nesterov's accelerated gradient

---

(*) Nesterov's accelerated gradient

(W)(A7) Revier - check you understood hill-climbing; vectors

- At any iteration, to compute current step :-
  - Extend previous step
  - Compute gradient step at result. pos.
  - Add the 2 to obtain final
- BR: Still maintaing running average, but not computing grad. at current loc; rather, ...

(W)(A8): - see
- Provably faster than simple grad. descent

(*) Nesterov's Accelerated gradient update

- $\Delta \underline{W}^{(k)} = \beta \Delta \underline{W}^{(k-1)} - \eta \nabla_{\underline{W}} Err\left(\underline{W}^{(k-1)} + \beta \Delta W^{(k-1)}\right)$

$\underline{W}^{(k)} = \underline{W}^{(k-1)} + \Delta \underline{W}^{(k)}$

(D11)

BR: Nesterov's method is faster than momentum
- For convex fns; optimum

(*) Gradient descent pseudocode with Nesterov method.

- initialise all weights $\underline{W}_1, \underline{W}_2, \ldots$
- Do:
  - $\forall$ layers k, init $\nabla_{\underline{W}_k} Err = 0$, $\Delta \underline{W}_k = 0$
  - $\forall$ every layer k :-

$$\underline{W}_k = \underline{W}_k + \beta \Delta \underline{W}_k$$

  - $\forall t = 1, \ldots, T$ (training instance)
    - $\forall$ every layer k :
      - compute gradient $\nabla_{\underline{W}_k} Err(\underline{Y}_t, \underline{d}_t)$
      - $\nabla_{\underline{W}_k} Err \mathrel{+}= \frac{1}{T} \nabla_{\underline{W}_k} Err(\underline{Y}_t, \underline{d}_t)$

- For every layer k:-

$$W_k = W_k - \eta \nabla_{W_k} Err$$

$$\Delta W_k = \beta \Delta W_k - \eta \nabla_{W_k} Err$$

- until Err converged

(W) (AIO): review pseudocode (particularly notation) - check understood.

(*) issue here

(*) see story so far - convergence

(*) Incremental updates

(*) training formulation

(W) graph illustration

- Simple input-output pairs
- Adjust parameters only at input-output pairs } gradient descent

ISSUE: (*) Process all training points before making a single adjustment

- Batch update

- compute loss over all training points (average)

(*) Process entire training set in one-batch, getting errors, adjusting param.

(*) 'Handkerchief grabbing'

(*) Incremental update

- Adjust function at one training point at a time
- Processed all training points → adjusted entire function.

(*) Incremental updates: ~~SGD~~ (pseudocode)

(W) (AIO): How is this diff vs gradient descent (batch)

- Given $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- Initialise all weights $W_1, W_2, \ldots, W_K$
- Do:- (until loss converged) (I)

(II) {
- For all $t = 1, 2, \ldots, T$:-
  - For every layer k:-
    - compute $\nabla_{W_k} Div(Y_t, d_t)$ (via backprop)
    - update: $W_k = W_k - \eta \nabla_{W_k} Div(Y_t, d_t)^T$ (III)

(*) Adjust param. after every training inst.

(*) Iterations → multiple passes our training data

- single pass through entire training data → epoch

(I) → our multiple epochs

(II) - one epoch

(III) → one update

(*) A heuristic illustration of why behaviour

- insert stochasticity i.e. randomly

(*) SGD - insert random permutation of training data between epochs

~~random~~ i.e. change order in which you present training instances

---

(Q): why does process of incremental updates work?

(Q): under what conditions

- Heuristic, simplistic exp.

(*) use extreme example

- use expected behaviour of gradient

- Batch vs SGD

(Ans) - review argument for batch vs SGD ;

(Q): under what conditions do incremental updates work?

- caveats - learning rate

(W) (Ans) - Review heuristic argument for why learning rate must decrease with iterations.

(*) incremental update, SGD, changing step size (with iteration)

(*) supplement to existing (W)(A14): review

SGD convergence

(R): SGD converges 'almost surely' to a global/local minimum for most functions.

sufficient condit.

$$\sum_k \eta_k = \infty \quad , \quad \sum_k \eta_k^2 < \infty$$

- eventually entire param space can be searched
- steps shrink (finite sum of squared step sizes finite)

(?) (W)(A15)

- Fastest converging series that satisfies both above:-

$$\eta_k \propto \frac{1}{k}$$

- optimal rate of shrinking step size for convex fns.
- learning rates determined heuristically
- convex loss, SGD → optimum sol
- non-convex loss; SGD → local minimum.

(W)(A16): Harmonic series

SGD convergence

(R) some key results here (W)(A17) - review + digest convergence results

- strong convexity assumptn
- generically / weakly convex

---

batch gradient convergence

- some distinctions as above

SGD example

- K-means -
  - (*) vertical bars - variation of error across runs
  - (*) SGD large variation, faster convergence, higher error local minima
  - (*) Batch gradient descent

(R)(HY)

- modelling a function
- distinction between
  empirical est. of error vs expected gen. error
  (over empirical distn) (over true distn)
- (✗) empirical error is an unbiased estimate of expected error (loss) (loss)

BR: Hence we can compute/minimise loss as we have done

(Q) How about variance

(*) $\text{Var}\left(\frac{1}{N}\sum_i \text{div}(x_i)\right) = \frac{1}{N^2}\sum_i \text{Var}(\text{div}) = \frac{N\,\text{Var}(\text{div})}{N^2} = \frac{\text{Var}(\text{div})}{N}$

(variance of empirical risk)

(*) compare variance of empirical/sample error (W)(A16): review mathematically
mini batch and SGD

(*) SGD - examines error/divergence for a single instance

↳ $L = \text{div}(x_i)$ $\mathbb{E}[L] = \mathbb{E}[\text{div}(x_i)]$

- SGD minimises an ~~un~~ unbiased estimator (loss) of actual function we want to minimise

- How about variance for SGD?

- Graphical illustration of variance issues/charact
- sample estimate/empirical loss estimates area using average length of distances
- this average distance depends on where training samples are taken
- Increase no. of samples
- BR: "large variance depending on n" - variation between epoch? will be large if you have a small no. of training samples, but increase no. of training samples, variation ↓

(*) Refers to estimated $\hat{w}$?

(W) (A17): Find the relevant formalism to bind these notations (which are well pres.)

OR: creating a tension between SGD and batch updating to motivate mini-batch

---

(*) Minibatch updates
- not one instance at a time (SGD) nor
- whole training set at a time (batch)
- use subset of training samples at a time

- very clear motivation for mini-batch

(*) Incremental update: minibatch update (pseudocode)
- see this as building on gradient descent (stochastic) with shrinking learning rate.

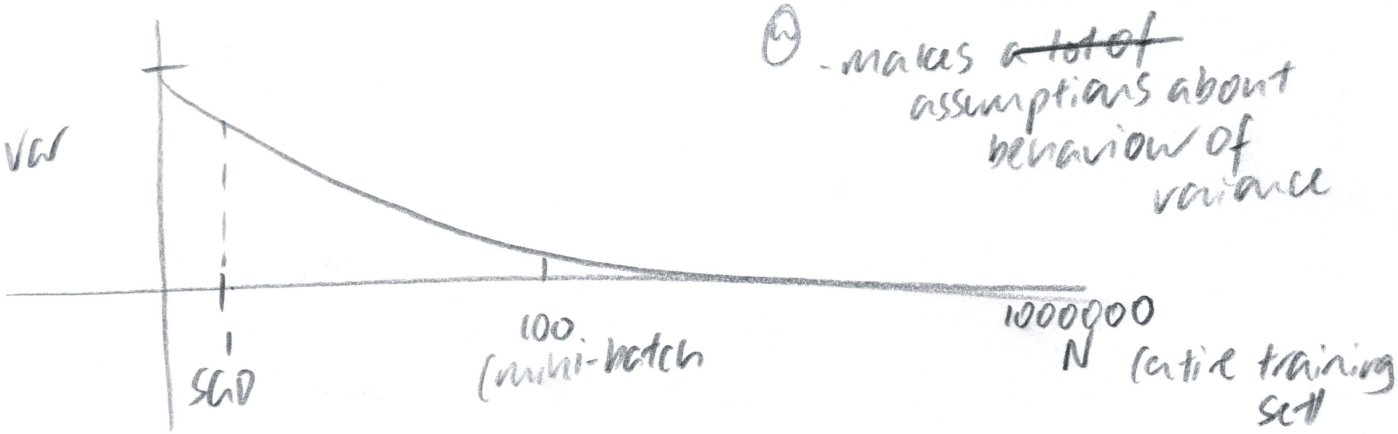
(W) (A18): (R) - check you understand algorithmically
- Randomly select subset of training samples
- Permute instances between epochs
- After permutation, run over mini-batch
- Compute average derivative of over minibatch; before making updates

---

(*) Batchloss is an unbiased estimate of expected loss
variance of batch error smaller than sample error variance in SGD.

- variance against batch size

(W) - makes assumptions about behaviour of variance



- minibatch convergence
  - GPU becomes important here
  - time taken to process instance similar to processing batch

---

(**) Empirical comparison of SGD, mini-batch, batch.

(*) Measuring loss

- Relies on estimating loss over entire training set (figures)

---

training and mini-batches

- mini-batch size in practice → hyperparameter
- convergence depends on learning rate
  - simple technique → start from fixed/exogenous parameters (e.g. Nesterov and 1st moment of gradient)
  - other methods
    - learning rate part of estimation