ROYAL INSTITUTE
OF TECHNOLOGY

EP3260: Machine Learning Over Networks

Lecture 8: Deep Neural Networks

Hossein S. Ghadikolaei, Hadi Ghauch, and Carlo Fischione

Division of Network and Systems Engineering
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology, Stockholm, Sweden

`https://sites.google.com/view/mlons/home`

March 2019

Learning outcomes

- Recap of optimization solution approaches for ML
- The hardness of DNN optimization landscape
- Inside of the black-box optimizers of the existing libraries
- Examples of how to apply your optimization knowledge

Outline

1. Recap
2. Basics of DNNs
3. Training DNN
4. Learning and Inference Over Networks

Smooth convex setting

Problem: $\underset{\boldsymbol{w}}{\text{minimize}}\ f(\boldsymbol{w}) = \frac{1}{N}\sum_{i\in[N]} f_i(\boldsymbol{w})$

Generic solver: $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha_k \boldsymbol{g}(\boldsymbol{w}_k)$

- **Preconditioning**

  GD: Steepest descent (SD) on Euclidean norm $\boldsymbol{g}(\boldsymbol{w}) = -\nabla f(\boldsymbol{w}_k)^T$

  Euclidean norm is agnostic to the geometry of the sub-level sets

  Newton method: SD on Mahalanobis norm $\boldsymbol{g}(\boldsymbol{w}) = -\nabla^2 f(\boldsymbol{w}_k)^{-1}\nabla f(\boldsymbol{w}_k)$

- **Over-parametrization**

  Improves expressiveness of the model

  Adding regularization terms to promote sparsity, smaller norms, etc.

- **Big datasets**

  SGD: gradient for one sample $\boldsymbol{g}(\boldsymbol{w}_k) = -\nabla f_{\zeta_k}(\boldsymbol{w}_k)$

  Mini-batch GD: $\boldsymbol{g}(\boldsymbol{w}_k) = -\frac{1}{N_k}\sum_{i\in[N_k]} \nabla f_{\zeta_k^i}(\boldsymbol{w}_k)$

Nonsmooth/nonconvex setting

- **Nonsmooth**

  Generalize gradient: subgradient or proximal methods

  Smooth then optimize

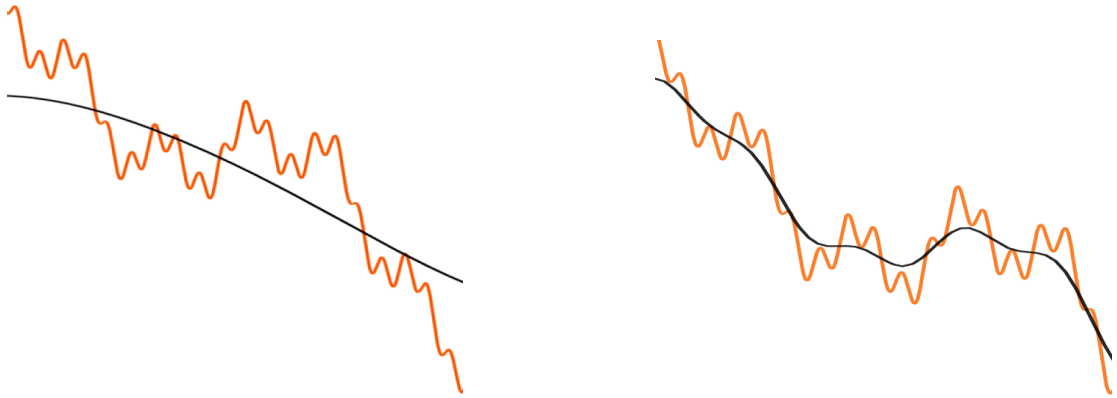
  Successive smooth upper-bound minimization

- **Nonconvex**

  Lack of global optimality, convergence to stationary points

  GD, successive convex approximation, coordinate descent, and BSUM

  Escaping saddle points (using 1st and 2nd order info)



Learning and inference over networks

Importance of communication graph

Parallel computing and edge computing

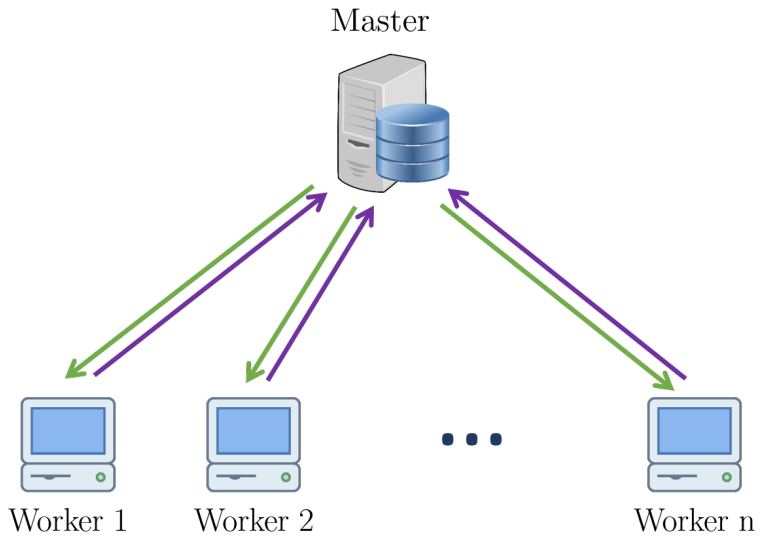
primal vs dual domains

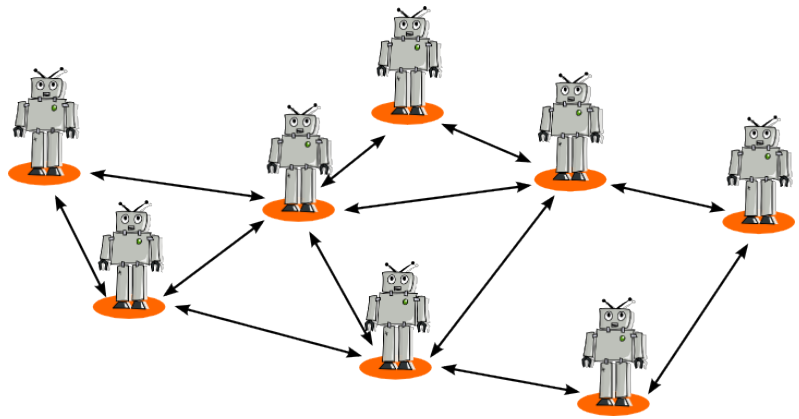
Privacy and distributed dataset

Decentralized vs distributed settings

Limited resources

communications, computation, storage





Outline

1. Recap
2. Basics of DNNs
3. Training DNN
4. Learning and Inference Over Networks

Percepteron

Linear mapping + activation function σ

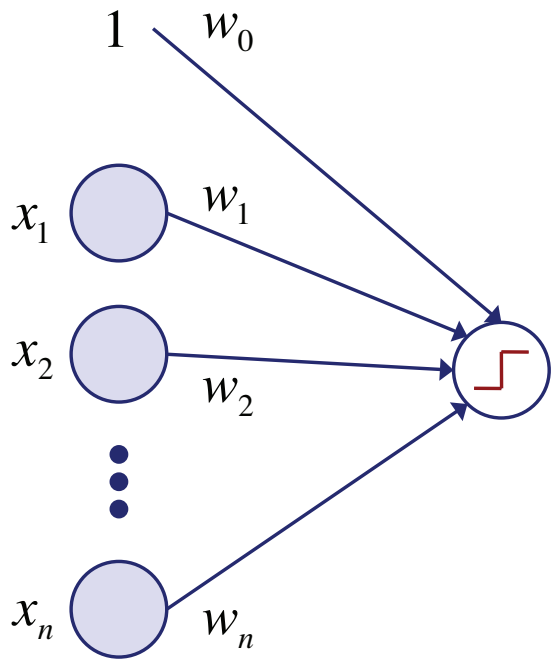
One weight per input

Output: $z = \sigma(\boldsymbol{w}^T \boldsymbol{x})$

Binary classification:

$\sigma(t) = 1$ if $t \geq 0$, and 0 otherwise

Other examples of σ: sigmoid, tanh, ReLU, ...

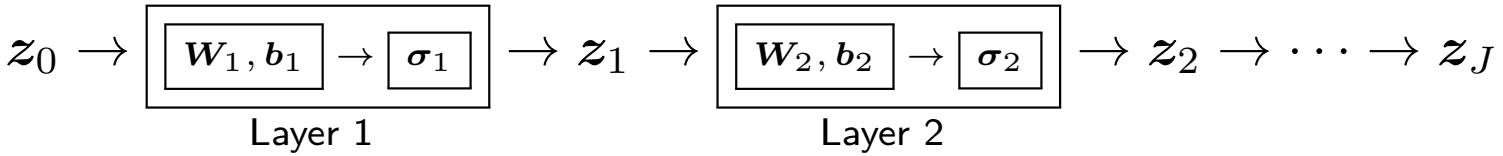


Do we need nonlinearity σ?

yes, in old days!

what about today?

Deep neural networks



Compactly: $\boldsymbol{z}_j = \boldsymbol{\sigma}_j(\boldsymbol{W}_j \boldsymbol{z}_{j-1} + \boldsymbol{b}_j)$

- $\boldsymbol{z}_{j-1} \in \mathbb{R}^{d_{j-1}}$: input to layer j, $(\boldsymbol{z}_0, \boldsymbol{z}_J) = (\boldsymbol{x}, \boldsymbol{y})$: data
- $\boldsymbol{W}_j \in \mathbb{R}^{d_j \times d_{j-1}}$: weights for layer j
- $\boldsymbol{b}_j \in \mathbb{R}^{d_j}$: bias for layer j
- $\boldsymbol{\sigma}_j() \in \mathbb{R}^{d_j \times d_j}$: activation function for layer j (element-wise operator)

Example architectures

Feed-forward NN: communication from input to output

Convolutional NN: $\boldsymbol{W}_j$ is circulant (convolutional) matrix

Fully-connected NN: no zeros in $\boldsymbol{W}_j$

Recurrent NN: allow for loops in network

Deep linear networks: $\boldsymbol{\sigma}_j = \boldsymbol{I}$ for all $j \in [J]$, so $\boldsymbol{y} = \boldsymbol{W}_k \ldots \boldsymbol{W}_2 \boldsymbol{W}_1 \boldsymbol{x}$

Established results on DNNs

Universal Approximation Theorem [Hornik-89, Cybenko-90]

A single hidden layer NN with an activation function can approximate any continuous function arbitrarily well, given enough hidden units

Are we done?

NP-hardness of the training [Blum-Rivest-89]

Training a 3-node NN ($\boldsymbol{W}_1 \in \mathbb{R}^{2\times d}, \boldsymbol{W}_2 \in \mathbb{R}^{1\times 2}$) is NP-complete

Violation of "given enough hidden units"

Limited resources

insufficient data (overfit), lack of computational power (underfit)!

Shall we give up?

Well, lets have a closer look at DNN optimization landscape

Established results on DNNs

Universal Approximation Theorem [Hornik-89, Cybenko-90]

A single hidden layer NN with an activation function can approximate any continuous function arbitrarily well, given enough hidden units

Are we done?

NP-hardness of the training [Blum-Rivest-89]

Training a 3-node NN ($\boldsymbol{W}_1 \in \mathbb{R}^{2\times d}, \boldsymbol{W}_2 \in \mathbb{R}^{1\times 2}$) is NP-complete

Violation of "given enough hidden units"

Limited resources

insufficient data (overfit), lack of computational power (underfit)!

Shall we give up?

Well, lets have a closer look at DNN optimization landscape

Established results on DNNs

Universal Approximation Theorem [Hornik-89, Cybenko-90]

A single hidden layer NN with an activation function can approximate any continuous function arbitrarily well, given enough hidden units

Are we done?

NP-hardness of the training [Blum-Rivest-89]

Training a 3-node NN ($\boldsymbol{W}_1 \in \mathbb{R}^{2\times d}, \boldsymbol{W}_2 \in \mathbb{R}^{1\times 2}$) is NP-complete

Violation of "given enough hidden units"

Limited resources

insufficient data (overfit), lack of computational power (underfit)!

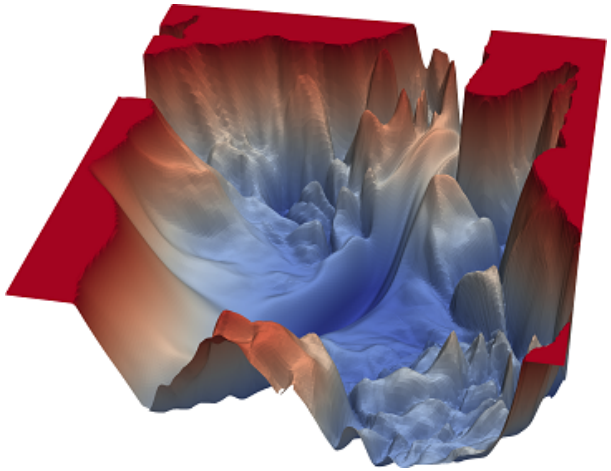
Shall we give up?

Well, lets have a closer look at DNN optimization landscape

Loss surface of DNNs

Abstract DNN training for $a > 0$:

$$\underset{\boldsymbol{w}}{\text{minimize}} \;\; f(\boldsymbol{w}) := \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{w}) + ar(\boldsymbol{w})$$



[Goodfellow-2015]

- $\boldsymbol{w}$: concatenation of all the weights in network
- $f()$: regularized non-convex loss function
- $r()$: regularization function
- $f_i()$: e.g., loss function for sample $i \in [N]$

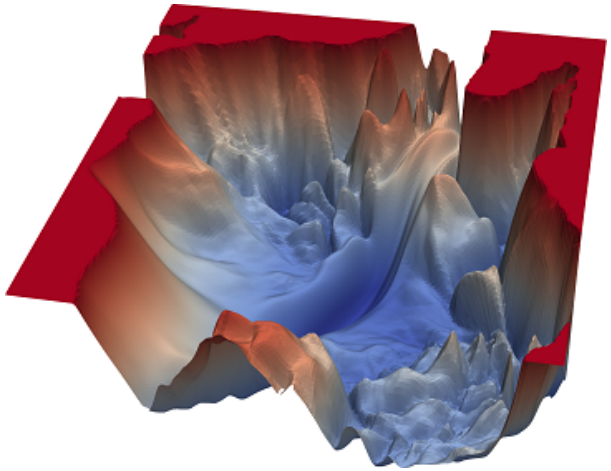
We've learned how to address it! 😊

GD, SGD, BCD, escaping saddle points, 2oN points, ...

Loss surface of DNNs

Abstract DNN training for $a > 0$:

$$\underset{\boldsymbol{w}}{\text{minimize}} \;\; f(\boldsymbol{w}) := \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{w}) + ar(\boldsymbol{w})$$



[Goodfellow-2015]

$\boldsymbol{w}$: concatenation of all the weights in network

$f()$: regularized non-convex loss function

$r()$: regularization function

$f_i()$: e.g., loss function for sample $i \in [N]$

We've learned how to address it! 🙂

GD, SGD, BCD, escaping saddle points, 2oN points, ...

Outline

1. Recap
2. Basics of DNNs
3. Training DNN
4. Learning and Inference Over Networks

Main optimization tricks

Globally optimal parameters $\rightarrow$ give it up!

Parameters optimization $\rightarrow$ run GD: $\boldsymbol{w}_{k+1} = \boldsymbol{x}_k - \frac{\alpha_k}{N} \sum_{i \in [N]} \nabla f_i(\boldsymbol{w}_k)$

Gradient evaluation $\rightarrow$ use backpropagation, watch

`https://www.youtube.com/watch?v=tIeHLnjs5U8`

`https://www.youtube.com/watch?v=GlcnxUlrtek`

Heavy computation on big dataset $\rightarrow$ run SGD or mini-batch GD: $\frac{1}{N_k} \sum_{i \in [N_k]} \nabla f_{\zeta_k^i}(\boldsymbol{w}_k)$

Escape saddle points $\rightarrow$ a bonus to replace GD with SGD!

Apply coordinate descent, so optimize over weights $\{\boldsymbol{W}_j\}_j$ individually

Main optimization tricks

Globally optimal parameters $\rightarrow$ give it up!

Parameters optimization $\rightarrow$ run GD: $\boldsymbol{w}_{k+1} = \boldsymbol{x}_k - \frac{\alpha_k}{N} \sum_{i \in [N]} \nabla f_i(\boldsymbol{w}_k)$

Gradient evaluation $\rightarrow$ use backpropagation, watch

`https://www.youtube.com/watch?v=tIeHLnjs5U8`

`https://www.youtube.com/watch?v=GlcnxUlrtek`

Heavy computation on big dataset $\rightarrow$ run SGD or mini-batch GD: $\frac{1}{N_k} \sum_{i \in [N_k]} \nabla f_{\zeta_k^i}(\boldsymbol{w}_k)$

Escape saddle points $\rightarrow$ a bonus to replace GD with SGD!

Apply coordinate descent, so optimize over weights $\{\boldsymbol{W}_j\}_j$ individually

Main optimization tricks

Globally optimal parameters $\rightarrow$ give it up!

Parameters optimization $\rightarrow$ run GD: $\boldsymbol{w}_{k+1} = \boldsymbol{x}_k - \frac{\alpha_k}{N} \sum_{i \in [N]} \nabla f_i(\boldsymbol{w}_k)$

Gradient evaluation $\rightarrow$ use backpropagation, watch

https://www.youtube.com/watch?v=tIeHLnjs5U8

https://www.youtube.com/watch?v=GlcnxUlrtek

Heavy computation on big dataset $\rightarrow$ run SGD or mini-batch GD: $\frac{1}{N_k} \sum_{i \in [N_k]} \nabla f_{\zeta_k^i}(\boldsymbol{w}_k)$

Escape saddle points $\rightarrow$ a bonus to replace GD with SGD!

Apply coordinate descent, so optimize over weights $\{\boldsymbol{W}_j\}_j$ individually

Main optimization tricks

Globally optimal parameters $\rightarrow$ give it up!

Parameters optimization $\rightarrow$ run GD: $\boldsymbol{w}_{k+1} = \boldsymbol{x}_k - \frac{\alpha_k}{N} \sum_{i \in [N]} \nabla f_i(\boldsymbol{w}_k)$

Gradient evaluation $\rightarrow$ use backpropagation, watch

`https://www.youtube.com/watch?v=tIeHLnjs5U8`

`https://www.youtube.com/watch?v=GlcnxUlrtek`

Heavy computation on big dataset $\rightarrow$ run SGD or mini-batch GD: $\frac{1}{N_k} \sum_{i \in [N_k]} \nabla f_{\zeta_k^i}(\boldsymbol{w}_k)$

Escape saddle points $\rightarrow$ a bonus to replace GD with SGD!

Apply coordinate descent, so optimize over weights $\{\boldsymbol{W}_j\}_j$ individually

Main optimization tricks

Globally optimal parameters $\rightarrow$ give it up!

Parameters optimization $\rightarrow$ run GD: $\boldsymbol{w}_{k+1} = \boldsymbol{x}_k - \frac{\alpha_k}{N} \sum_{i \in [N]} \nabla f_i(\boldsymbol{w}_k)$

Gradient evaluation $\rightarrow$ use backpropagation, watch

`https://www.youtube.com/watch?v=tIeHLnjs5U8`

`https://www.youtube.com/watch?v=GlcnxUlrtek`

Heavy computation on big dataset $\rightarrow$ run SGD or mini-batch GD: $\frac{1}{N_k} \sum_{i \in [N_k]} \nabla f_{\zeta_k^i}(\boldsymbol{w}_k)$

Escape saddle points $\rightarrow$ a bonus to replace GD with SGD!

Apply coordinate descent, so optimize over weights $\{\boldsymbol{W}_j\}_j$ individually

Main optimization tricks

Globally optimal parameters $\rightarrow$ give it up!

Parameters optimization $\rightarrow$ run GD: $\boldsymbol{w}_{k+1} = \boldsymbol{x}_k - \frac{\alpha_k}{N} \sum_{i \in [N]} \nabla f_i(\boldsymbol{w}_k)$

Gradient evaluation $\rightarrow$ use backpropagation, watch

`https://www.youtube.com/watch?v=tIeHLnjs5U8`

`https://www.youtube.com/watch?v=GlcnxUlrtek`

Heavy computation on big dataset $\rightarrow$ run SGD or mini-batch GD: $\frac{1}{N_k} \sum_{i \in [N_k]} \nabla f_{\zeta_k^i}(\boldsymbol{w}_k)$

Escape saddle points $\rightarrow$ a bonus to replace GD with SGD!

Apply coordinate descent, so optimize over weights $\{\boldsymbol{W}_j\}_j$ individually

Main optimization tricks

Globally optimal parameters $\rightarrow$ give it up!

Parameters optimization $\rightarrow$ run GD: $\boldsymbol{w}_{k+1} = \boldsymbol{x}_k - \frac{\alpha_k}{N} \sum_{i \in [N]} \nabla f_i(\boldsymbol{w}_k)$

Gradient evaluation $\rightarrow$ use backpropagation, watch

`https://www.youtube.com/watch?v=tIeHLnjs5U8`

`https://www.youtube.com/watch?v=GlcnxUlrtek`

Heavy computation on big dataset $\rightarrow$ run SGD or mini-batch GD: $\frac{1}{N_k} \sum_{i \in [N_k]} \nabla f_{\zeta_k^i}(\boldsymbol{w}_k)$

Escape saddle points $\rightarrow$ a bonus to replace GD with SGD!

Apply coordinate descent, so optimize over weights $\{\boldsymbol{W}_j\}_j$ individually

Back-propagation

Uses chain rule over a computational graph

Forward-mode differentiation

one pass gives the gradients of all the outputs w.r.t. one parameter

Reverse-mode differentiation (backprop)

one pass gives the gradients of one output w.r.t. all the parameters

computationally efficient for large NNs with millions of parameters

Example 1: find forward- and reverse-mode differentiations on $f = (a + b)(b + 1)$

Example 2: run backprop on $f = (y - a_j)^2$ where $a_j = \sigma(z_j)$, $z_j = w_j a_{j-1} + b_j)$, for three layers

Poor condition

Curvature of f is not even across dimensions

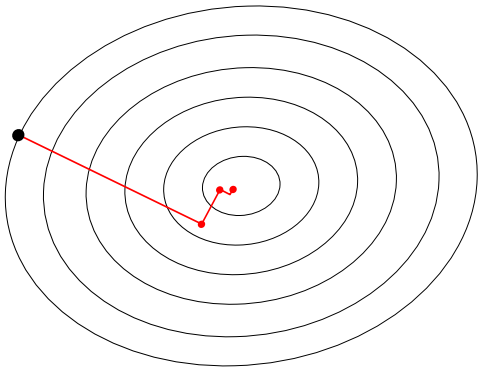
Replace $\nabla f(\boldsymbol{w}_k)$ by $\boldsymbol{B}_k \nabla f(\boldsymbol{w}_k)$ (preconditioning)

What about Newton method $\boldsymbol{B}_k = \nabla^2 f(\boldsymbol{w}_k)^{-1}$?

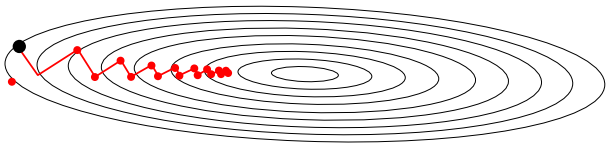
How about changing perspective? different step-size per dimension:

α_k chosen a priori (constant or diminishing) independent of $f(\boldsymbol{w}_k)$

works like a diagonal preconditioner matrix



small L/μ



large L/μ

Adaptive step sizes

AdaGrad

$$B_k = \epsilon + \sqrt{\operatorname{diag}\left(\sum_{i\in[k]} \nabla f_{\zeta_i}(\boldsymbol{w}_i)\nabla f_{\zeta_i}(\boldsymbol{w}_i)^T\right)}$$

Diagonal element for coordinate j (namely $[\boldsymbol{w}]_j$) is $\sum_{i\in[k]} \left([\nabla f_{\zeta_i}(\boldsymbol{w}_i)]_j\right)^2$

It is the l_2 norm of previous derivatives

- dampening frequent parameters
- boosting infrequent (rarely occurring) parameters
- useful when the data is sparse and features have different frequencies
- also, no further manual tune of the learning rate (stepsize)!

Drawback: vanishing stepsize $\rightarrow$ use exponentially decaying average (**RMSProp**)

Use bias correction terms $\rightarrow$ Adaptive Moment Estimation (ADAM)

Adaptive step sizes

AdaGrad

$$B_k = \epsilon + \sqrt{\operatorname{diag}\left(\sum_{i\in[k]} \nabla f_{\zeta_i}(\boldsymbol{w}_i)\nabla f_{\zeta_i}(\boldsymbol{w}_i)^T\right)}$$

Diagonal element for coordinate j (namely $[\boldsymbol{w}]_j$) is $\sum_{i\in[k]} \left([\nabla f_{\zeta_i}(\boldsymbol{w}_i)]_j\right)^2$

It is the l_2 norm of previous derivatives

dampening frequent parameters

boosting infrequent (rarely occurring) parameters

useful when the data is sparse and features have different frequencies

also, no further manual tune of the learning rate (stepsize)!

Drawback: vanishing stepsize $\rightarrow$ use exponentially decaying average (**RMSProp**)

Use bias correction terms $\rightarrow$ Adaptive Moment Estimation (ADAM)

Adaptive step sizes

AdaGrad

$$B_k = \epsilon + \sqrt{\operatorname{diag}\left(\sum_{i\in[k]} \nabla f_{\zeta_i}(\boldsymbol{w}_i)\nabla f_{\zeta_i}(\boldsymbol{w}_i)^T\right)}$$

Diagonal element for coordinate j (namely $[\boldsymbol{w}]_j$) is $\sum_{i\in[k]} \left([\nabla f_{\zeta_i}(\boldsymbol{w}_i)]_j\right)^2$

It is the l_2 norm of previous derivatives

dampening frequent parameters

boosting infrequent (rarely occurring) parameters

useful when the data is sparse and features have different frequencies

also, no further manual tune of the learning rate (stepsize)!

Drawback: vanishing stepsize $\rightarrow$ use exponentially decaying average (**RMSProp**)

Use bias correction terms $\rightarrow$ Adaptive Moment Estimation (ADAM)

Too many parameters!

Good expressiveness power

Hard to optimize (both statistically and algorithmically)

How to address

Data augmentation: add new data, augment after linear maps

Regularization: add $\|\boldsymbol{w}\|_1$ term

Dropout: randomly dropping some neurons (works magically well!)

Early stop: monitor test error

Batch normalization

Input normalization to bring all the features on the same scale

improves learning

Why not applying this idea to every layer? two new parameters to learn per layer (γ_j and β_j), same training pipeline, almost same backprop.

Batch normalization for layer j [Ioffe-Szegedy-2015]

Consider a minibatch of size N_j, $\mathcal{B} = \{\boldsymbol{z}_j^{[i]}\}_{i \in [N_j]}$

Compute batch mean $m_j^{[\mathcal{B}]}$ and batch standard deviation $\Sigma_j^{[\mathcal{B}]}$

$$m_j^{[\mathcal{B}]} = \frac{1}{N_j} \sum_{i \in \mathcal{B}} \boldsymbol{z}_j^{[i]} \quad , \quad \Sigma_j^{[\mathcal{B}]} = \frac{1}{N_j} \sum_{i \in \mathcal{B}} \left(\boldsymbol{z}_j^{[i]} - m_j^{[\mathcal{B}]} \right)^2$$

For every $i \in \mathcal{B}$, normalize the outputs by $\boldsymbol{z}_{\text{norm},j}^{[i]} = \dfrac{\boldsymbol{z}_j^{[i]} - m_j^{[\mathcal{B}]}}{\sqrt{\Sigma_j^{[\mathcal{B}]} + \epsilon}}$

Use the new output: $\hat{\boldsymbol{z}}_j^{[i]} = \gamma_j \boldsymbol{z}_{\text{norm},j}^{[i]} + \beta_j$

Do we need activation function σ?

No clear answer! most ideas are empirical, chosen due to their superior performance

Some existing hypotheses on existing algorithms may be wrong! e.g., look for why batch normalization works [Santurkar-2018]

Activation function may increase expressiveness

unless we add some nonlinear functions after 1st layer:

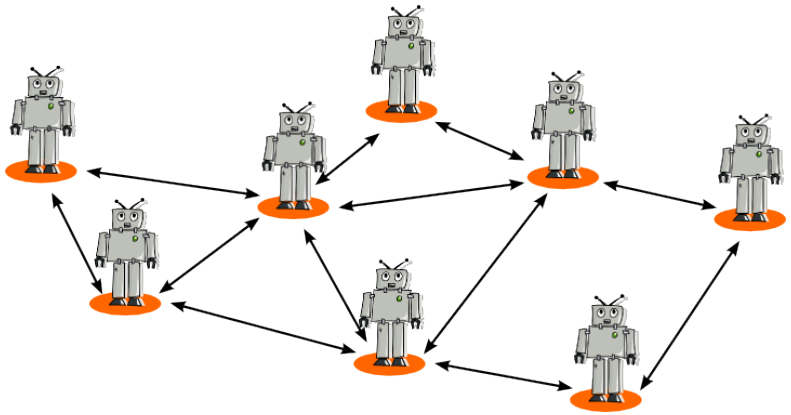
$$\min_{\boldsymbol{W}} \|\boldsymbol{Y} - \boldsymbol{W}\boldsymbol{X}\|^2 \leq \min_{\boldsymbol{W}_1, \boldsymbol{W}_2} \|\boldsymbol{Y} - \boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{X}\|^2$$

Today's NNs have many nonlinear operations (e.g., batch normalization), so technically we may not need nonlinear activations anymore

Outline

1. Recap
2. Basics of DNNs
3. Training DNN
4. Learning and Inference Over Networks

Learning and inference over networks



Mathematically, similar to Lectures 5, 6 and 7

Given a trained model, can we apply them on cheap devices?

not memory/coputational efficient (AlexNet has 62 million parameters and 240 MB size)

end-device may have lower-precision!

Cloud computing vs edge computing for inference?

Memory/precision limited devices

Apply lossy compression techniques to a trained model

- assume over-parametrization, so feel free to drop some weights and retrain
- lower the precision of weights, from 32 to 8
- cluster weights and use source coding techniques to compress them

AlexNet with 240 MB $\rightarrow$ 0.47 MB, almost same accuracy [Iandola-2017]

Many other approaches exist

Train with low-precision numbers

How to make the solution robust to change in weight precisions?

Think of the local geometry around your solution. Can it tolerate some errors in weights? Use sensitivity analysis.

Do you favor solutions in steep valleys? Can you reject them?

Many more challenges

Edge computing

Which functions (classification) should be run locally and which globally

Collaborative inference

Classification with sub-sampled inputs (for communication-efficiency)

Adversarial examples

Faulty communication links

Latency in communication links

CA 6: Deep neural networks

Consider "MNIST" dataset. Consider a DNN with J layers and $\{N_j\}$ neurons on layer j.

a) Train DNN using SGD and your choices of hyper-parameters, L, and $\{N_j\}_{j\in[J]}$. Report the convergence rate on the training as well as the generalization performance. Feel free to change SGD to any other solver of your choice (give explanation for the choice).

b) Repeat part a with mini-batch GD of your choice of the mini-batch size, retrain DNN, and show the performance measures. Compare the training performance (speed, accuracy) using various adaptive learning rates (constant, diminishing, AdaGrad, RMSProp).

c) Consider design of part a. Fix $\sum_j N_j$. Investigate shallower networks (smaller J) each having potentially more neurons versus deeper network each having fewer neurons per layer, and discuss pros and cons of these two DNN architectures.

d) Split the dataset to 6 random disjoint subsets, each for one worker, and repeat part a on master-worker computational graph.

e) Consider a two-star topology with communication graph (1,2)-3-4-(5,6). Repeat part a using your choice of distributed optimization solver. You can add communication-efficiency to the iterations, if you like!

f) To promote sparse solutions, you may use l_1 regularization or a so-called dropout technique. Explain how you incorporate each of these approaches in the training? Compare their training performance and the size of the final trained models.

g) Improving the smoothness of an optimization landscape may substantially improve the convergence properties of first-order iterative algorithms. Batch-normalization is a relatively simple technique to smoothen the landscape [Santurkar-2018]. Using the materials of the course, propose an alternative approach to improve the smoothness. Provide numerical justification for the proposed approach.

Some references

- A. Goodfellow, Y. Bendgio, A Courville, “Deep learning”, MIT Press, Chap. 6.

- K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” Neural Networks, 1989.

- G. Cybenko, “Approximation by superpositions of a sigmoidal function,” Mathematics of control, signals and systems, 1989.

- A. L. Blum and R. L. Rivest, “Training a 3-node neural network is NP-complete,” Machine learning: From Theory to Applications, 1993.

- S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift," ICML, 2015.

- S. Santurkar, D. Tsipras, A. Ilyas, A. Madry, “How does batch normalization help optimization?” NIPS, 2018.

- F. N. Iandola *et. al.*, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size,” ICLR, 2017.