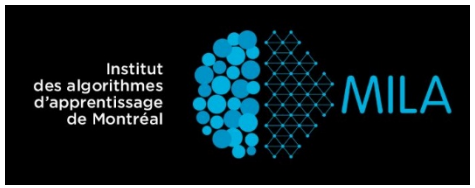
Regularization and Optimization

Jian Tang

tangjianpku@gmail.com



What is regularization

- The goal of machine learning algorithm is to perform well on the training data and generalize well to new data
- Regularization are the techniques to improve the generalization ability
  - i.e., avoid overfitting

Outline

- Regularization
  - Parameter Norm Penalties
  - Data set Augmentation
  - Noise Robustness
  - Semi-supervised Learning
  - Multi-task Learning
  - Early Stopping
  - Dropout

Parameter Norm Penalties

- Adding a parameter norm penalty $\Omega(\theta)$ to the objective function J. The regularized objective function is denoted as:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

- $\alpha \in [0, \infty)$ is a hyperparameter that controls the weights of the regularization term
- For regularization neural networks
  - Only the weights of the linear transformation at each layer are regularized
  - The biases are not regularized

L^2 Parameter Regularization

- $\Omega(\theta) = \frac{1}{2}\left|\left|\mathbf{w}\right|\right|^2$, also know as weight decay or ridge regression
- The objective function:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$$

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}).$$

- Update w with SGD:

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

- Push w towards zero

L^1 Parameter Regularization

- $\Omega(\theta) = \left|\left|\mathbf{w}\right|\right|_1 = \sum_i w_i,$
- The objective function:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha||\boldsymbol{w}||_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{w})$$

- Compare to L2 regularization, L1 regularization results in a solution that is more sparse
  - Some parameters have an optimal value of zero

L1 Regularization

- L1 regularization:

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j |W_{i,j}^{(k)}|$$

- Gradient:

$$\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = \text{sign}(\mathbf{W}^{(k)})$$

$$\text{sign}(\mathbf{W}^{(k)})_{i,j} = 1_{\mathbf{W}_{i,j}^{(k)} > 0} - 1_{\mathbf{W}_{i,j}^{(k)} < 0}$$

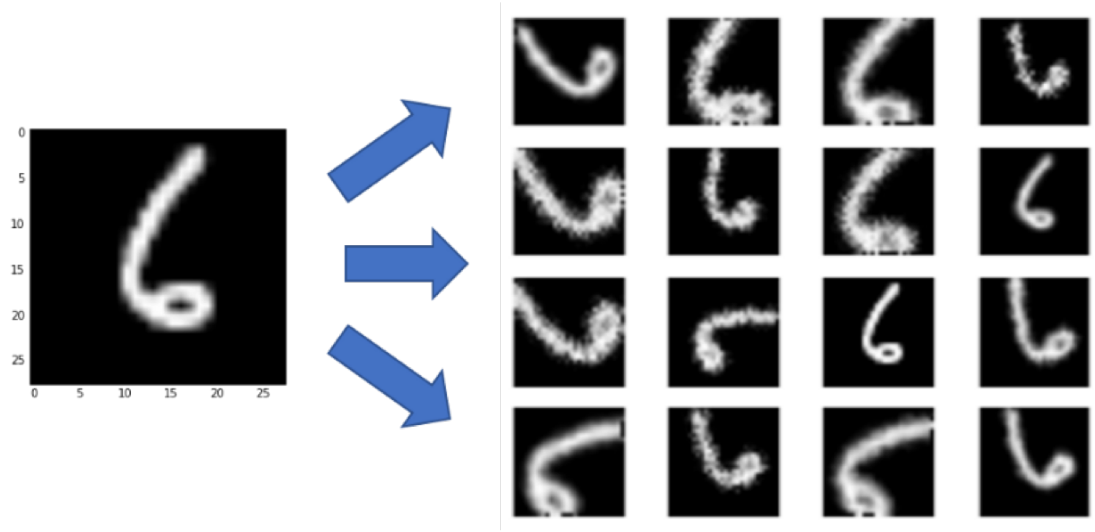
  - Only applies to weights, not biases (weigh decay)
  - Can be interpreted as having a Laplace prior over the weights, while performing MAP estimation.
  - Unlike L2, L1 will push some weights to be exactly 0.

Outline

- Regularization
  - Parameter Norm Penalties
  - Dataset Augmentation
  - Noise Robustness
  - Semi-supervised Learning
  - Multi-task Learning
  - Early Stopping
  - Dropout

Data Augmentation

- Best way to improve the performance of machine learning
  - Train it with more data
- Create fake data and add it to the training data
  - Translation
  - Rotation
  - Random crops
  - Inject noise
  - …



Outline

- Regularization
  - Parameter Norm Penalties
  - Dataset Augmentation
  - Noise Robustness
  - Semi-supervised Learning
  - Multi-task Learning
  - Early Stopping
  - Dropout

Noise Robustness

- Adding noise to the weights
  - Push the model into regions where the model is relatively insensitive to small variations in the weights
  - Find points that are not merely minima, but minima surrounded by flat regions.
- Adding noise at the output targets
  - Most data sets have some amount of mistakes in the output labels: y
  - Explicitly model the noise on the labels
  - For example, the training label y is correct with probability $1 - \epsilon$, and any of the other labels with probability ϵ

Outline

- Regularization
  - Parameter Norm Penalties
  - Dataset Augmentation
  - Noise Robustness
  - Semi-supervised Learning
  - Multi-task Learning
  - Early Stopping
  - Dropout

Semi-supervised Learning

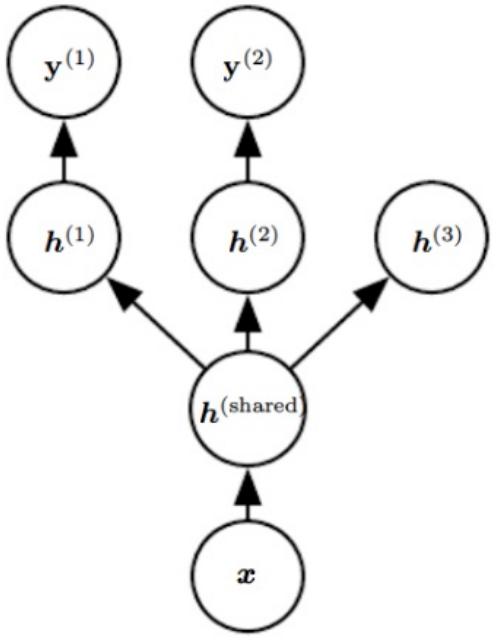
- Semi-supervised learning: both unlabeled examples from p(x) and labeled examples p(x,y) are used to estimate p(y|x)
- Share parameters between the unsupervised objective p(x) and supervised objective p(y|x)
  - E.g., for both objectives, the goal is to learn a representation h = f(x), which can be shared across the two objectives

Outline

- Regularization
  - Parameter Norm Penalties
  - Dataset Augmentation
  - Noise Robustness
  - Semi-supervised Learning
  - Multi-task Learning
  - Early Stopping
  - Dropout

Multi-task Learning

- Jointly learning multi-tasks by sharing the same inputs and some intermediate representations, which capture a common pool of factors

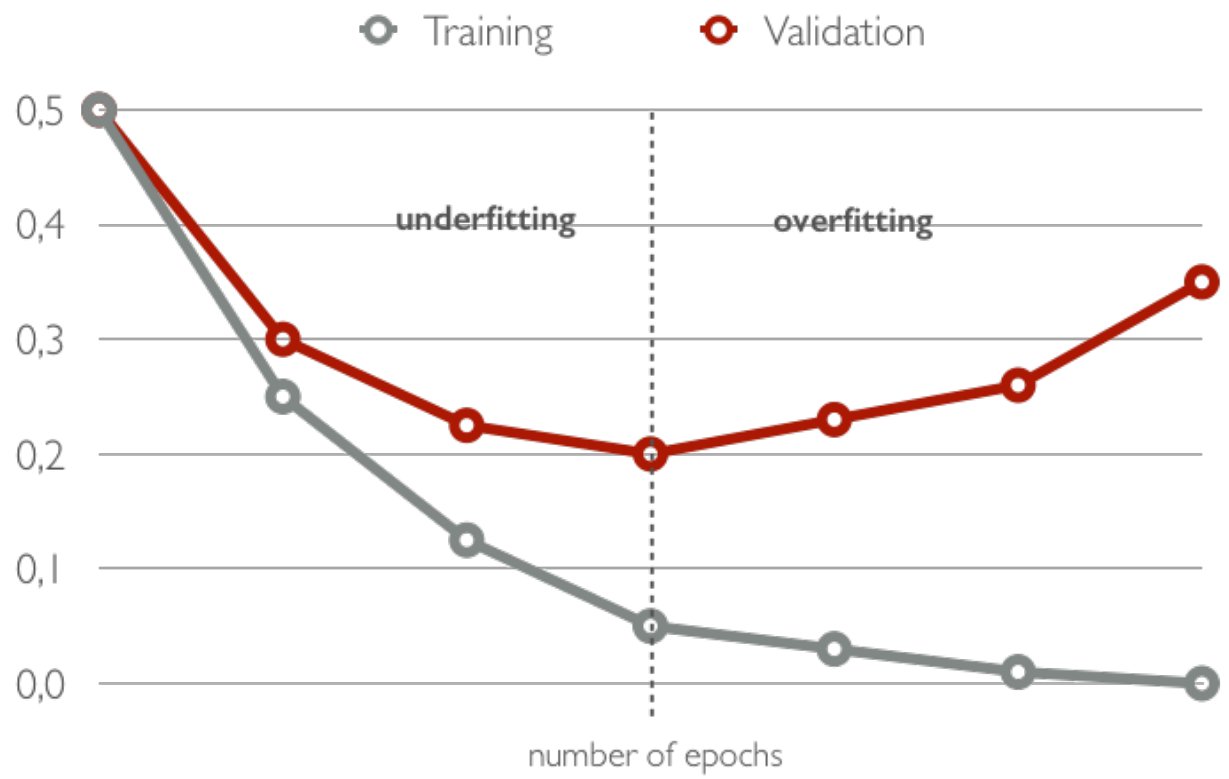


Outline

- Regularization
  - Parameter Norm Penalties
  - Dataset Augmentation
  - Noise Robustness
  - Semi-supervised Learning
  - Multi-task Learning
  - Early Stopping
  - Dropout

Early Stopping

• To select the number of epochs, stop training when validation set error increases (with some look ahead).



Outline

- Regularization
  - Parameter Norm Penalties
  - Dataset Augmentation
  - Noise Robustness
  - Semi-supervised Learning
  - Multi-task Learning
  - Early Stopping
  - Dropout

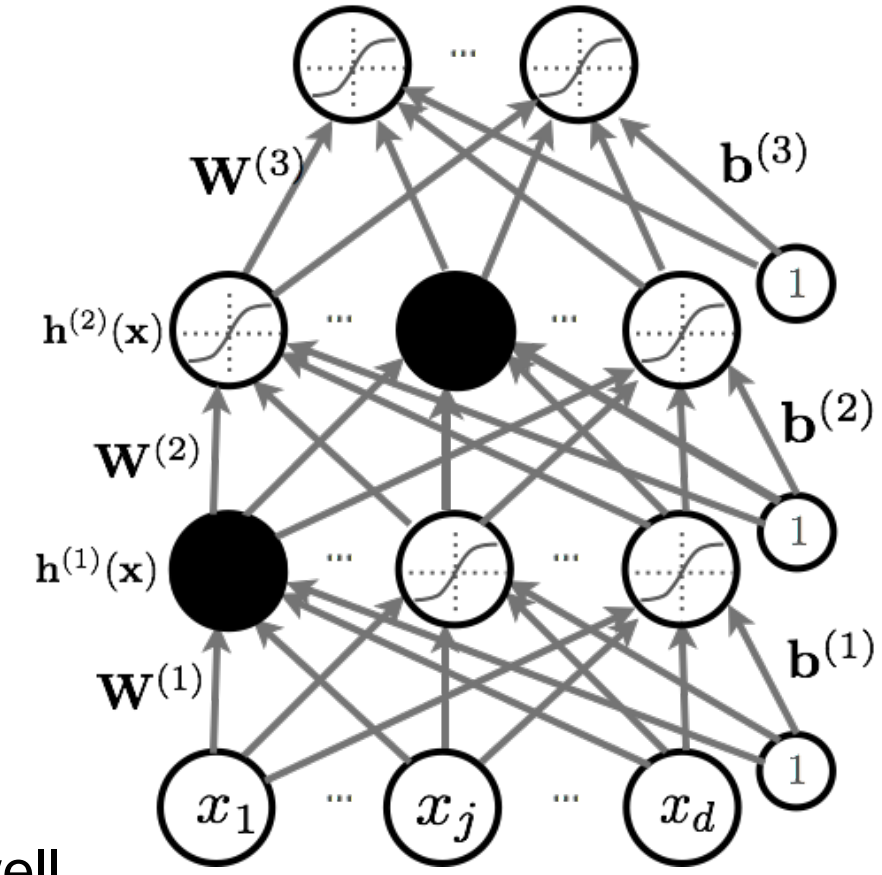
Dropout

- Overcome overfitting by a ensemble of multiple different models
  - Trained with different architectures
  - Trained on different data sets
- Too expensive on deep neural networks
- Dropout:
  - Training multiple networks together by parameter sharing

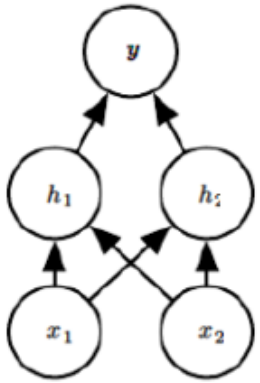
Dropout

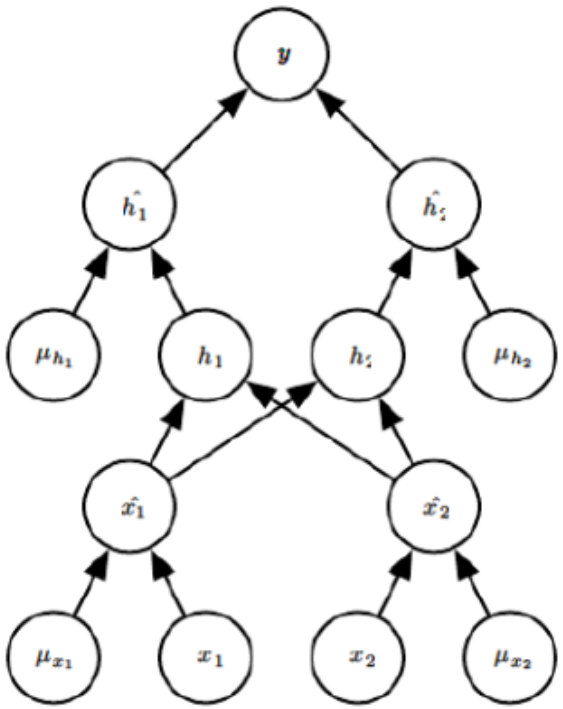
• Key idea: Cripple neural network by removing hidden units stochastically

- each hidden unit is set to 0 with probability 0.5
- hidden units cannot co-adapt to other units
- hidden units must be more generally useful



• Could use a different dropout probability, but 0.5 usually works well


y
h_1
h_2
x_1
x_2


y
$\hat{h_1}$
$\hat{h_2}$
μ_{h_1}
h_1
h_2
μ_{h_2}
$\hat{x_1}$
$\hat{x_2}$
μ_{x_1}
x_1
x_2
μ_{x_2}

Dropout

- Use random binary masks $m^{(k)}$
  - layer pre-activation for k>0

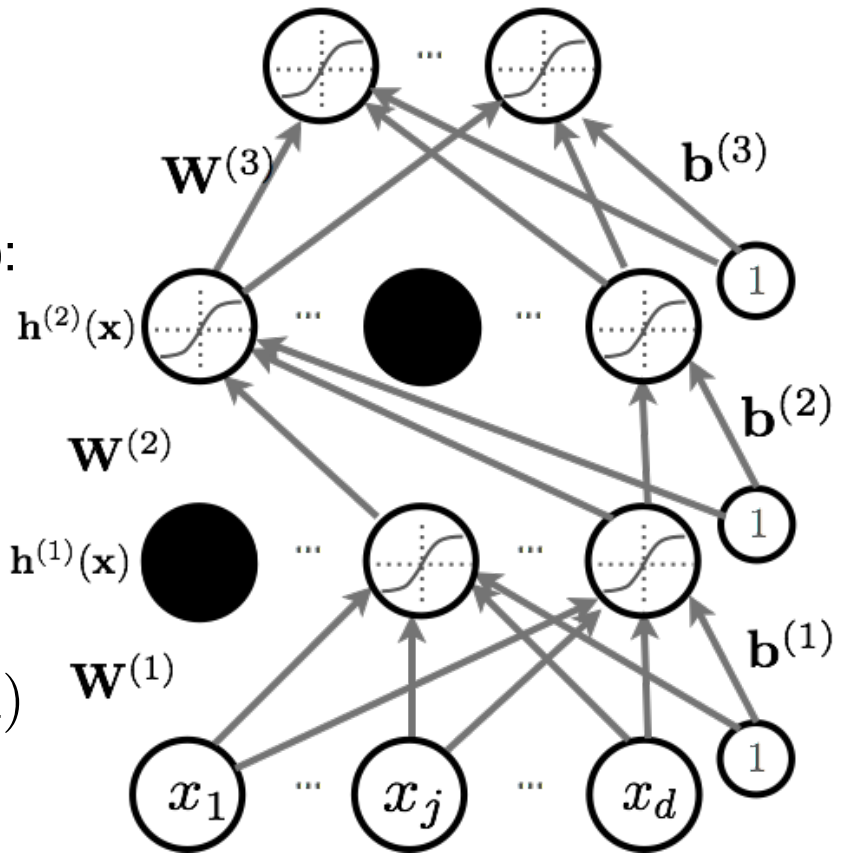
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - hidden layer activation (k=1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x})) \odot \mathbf{m}^{(k)}$$

  - Output activation (k=L+1)

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



Dropout at Test Time

- At test time, we replace the masks by their expectation
  - This is simply the constant vector 0.5 if dropout probability is 0.5
  - For single hidden layer: equivalent to taking the geometric average of all neural networks, with all possible binary masks
- Can be combined with unsupervised pre-training
- Beats regular backpropagation on many datasets
- Ensemble: Can be viewed as a geometric average of exponential number of networks.

Outline

- Optimization
  - Parameter Initialization Strategies
  - Momentum
  - Adaptive Learning Rates (AdaGrad, RMSProp, Adam)
  - Batch Normalization

Parameter Initialization (Glorot and Bengio, 2010)

- For a fully connected network with *m* inputs and *n* outputs, the weights are sampled according to:

$$\mathrm{W}_{i,j} \sim U(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}).$$

- Which aims to tradeoff between the goal of initializing all layers to have the same **activation variance** and the goal of initializing all layers to have the same **gradient variance**

Tricks of the Trade

- Normalizing your (real-valued) data:
  - for each dimension x_i subtract its training set mean
  - divide each dimension x_i by its training set standard deviation
  - this can speed up training

- Decreasing the learning rate: As we get closer to the optimum, take smaller update steps:

  i. start with large learning rate (e.g. 0.1)
  ii. maintain until validation error stops improving
  iii. divide learning rate by 2 and go back to (ii)

Mini-batch, Momentum

• Make updates based on a mini-batch of examples (instead of a single example):

- the gradient is the average regularized loss for that mini-batch
- can give a more accurate estimate of the gradient
- can leverage matrix/matrix operations, which are more efficient

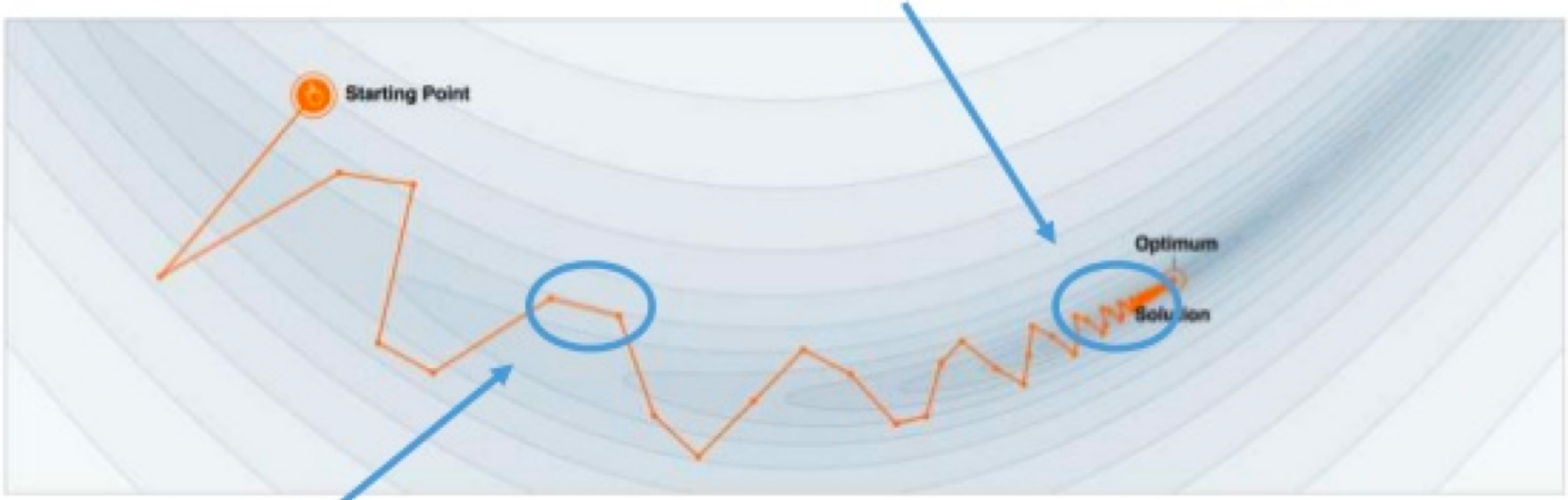
• Momentum: Can use an exponential average of previous gradients:

$$\overline{\nabla}_{\boldsymbol{\theta}}^{(t)} = \nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)}) + \beta \overline{\nabla}_{\boldsymbol{\theta}}^{(t-1)}$$

- can get pass plateaus more quickly, by "gaining momentum"

Why Momentum really works?

The momentum term **reduces updates for dimensions whose gradients change directions.**



The momentum term **increases for dimensions whose gradients point in the same directions.**

Demo : http://distill.pub/2017/momentum/

Adapting Learning Rates

• Updates with adaptive learning rates ("one learning rate per parameter")

- Adagrad: learning rates are scaled by the square root of the cumulative sum of squared gradients

$$\gamma^{(t)} = \gamma^{(t-1)} + \left(\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})\right)^2 \quad \overline{\nabla}_\theta^{(t)} = \frac{\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

- RMSProp: instead of cumulative sum, use exponential moving average

$$\gamma^{(t)} = \beta\gamma^{(t-1)} + (1-\beta)\left(\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})\right)^2$$

- Adam: essentially combines RMSProp with momentum

$$\overline{\nabla}_\theta^{(t)} = \frac{\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

Batch Normalization

- Internal covariate shift
  - Covariate shift: Changes of input distribution to a learning system

$$\ell = F(x, \theta)$$

  - Internal covariate shift: Extension to the deep network

$$\begin{aligned} \ell &= F_2(F_1(u, \theta_1), \theta_2) \\ &= F_2(x, \theta_2) \end{aligned}$$

- Normalizing the inputs will speed up training (Lecun et al. 1998)
  - ➢ could normalization be useful at the level of the hidden layers?

Batch Normalization

- Normalizing the inputs will speed up training (Lecun et al. 1998)
  - could normalization be useful at the level of the hidden layers?

- Batch normalization is an attempt to do that (Ioffe and Szegedy, 2014)
  - each unit's pre-activation is normalized (mean subtraction, stddev division)
  - during training, mean and stddev is computed for each minibatch
  - backpropagation takes into account the normalization
  - at test time, the global mean / stddev is used

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Learned linear transformation to adapt to non-linear activation function (γ and β are trained)

Batch Normalization

- Why normalize the pre-activation?
  - can help keep the pre-activation in a non-saturating regime (though the linear transform $y_i \leftarrow \gamma \widehat{x}_i + \beta$ could cancel this effect)

- Why use minibatches?
  - since hidden units depend on parameters, can't compute mean/ stddev once and for all
  - adds stochasticity to training, which might regularize

Batch Normalization

- How to take into account the normalization in backdrop?
  - derivative w.r.t. x_i depends on the partial derivative of both: the mean and stddev
  - must also update γ and β

- Why use the global mean and stddev at test time?
  - removes the stochasticity of the mean and stddev
  - requires a final phase where, from the first to the last hidden layer
    - propagate all training data to that layer
    - compute and store the global mean and stddev of each unit

References

- Chapter 7-8, Deep Learning book