ADAPTIVE LEARNING RATE

-SRAVYA REDDY
Motivation

- Basic methods have a fixed learning rate.

- Challenges of using learning rate scheduler
  - Dependency of type of model and problem.
  - Same learning rate is applied on different parameters.

- **Solution:** Adaptive learning rate.

- Adaptive learning rate is a method by which the performance of the model on the training dataset can be monitored by the learning algorithm and the learning rate can be adjusted in response.
AdaGrad

- The AdaGrad (Adaptive Gradient) algorithm, individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square-root of the sum of all of their historical squared values.
- Adagrad uses a different learning rate for every parameter $\theta_i$ at every time step $t$.
- It has an improved performance over SGD.
- Mostly used in natural language processing and image recognition.
AdaGrad Algorithm

**Require:** Global learning rate

**Require:** Initial parameter

**Require:** small constant \((\text{suggested value: } 10^{-7})\)

Initialize gradient accumulation variable \(r = 0\)

while stopping criterion not met do

Sample a mini-batch of \(m\) examples from the training set \(\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}\) with corresponding targets \(y^{(i)}\).

Compute gradient:

Accumulate squared gradient:

Compute update:

Apply update

end while
RMS Prop

- Problem with AdaGrad is its nature of radically diminishing learning rates and hence RMS Prop algorithm.
- RMSProp would deal with this problem and it is similar to gradient descent with momentum.

- RMSProp as well divides the learning rate by an exponentially decaying average of squared gradients.
- Usual values for $\beta$ is 0.9 or 0.95.
- RMSProp converges faster than AdaGrad to the convex bowl.
- It is useful when dealing with sparse data or noisy data.
RMSProp Algorithm

Require: Global learning rate, \textit{decay rate}

Require: Initial parameter

Require: small constant .

Initialize gradient accumulation variable \( r = 0 \)

\[
\text{while stopping criterion not met do}
\]

Sample a mini-batch of \( m \) examples from the training set \( \{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\} \) with corresponding targets \( y^{(i)} \).

Compute gradient:

\textbf{Accumulate squared gradient}:

Compute update:

Apply update

\textbf{end while}
Adam

- The name is derived from the phrase “Adaptive moments”.
- It estimates the first moment and the second moment of the gradients and hence the name of the method.
- It is a combines the advantages of AdaGrad and RMSProp.
  - *Inspired from AdaGrad, it maintains the per-parameter learning rate that improves performance of problems with sparse gradients.*
  - *Inspired from RMSProp, it stores the exponential decay of average of the past squared gradients.*
Adam Algorithm

**Require:** step size (suggested default: 0.001)

**Require:** Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.(Suggested defaults: 0.9 and 0.999 respectively)

**Require:** Small constant used for numerical stabilization. (Suggested default: $10^{-8}$)

**Require:** Initial parameters

Initialize $1^{st}$ and $2^{nd}$ moment variables $s = 0, r = 0$

Initialize time step $t = 0$

**while** stopping criterion not met **do**

Sample a mini-batch of $m$ examples from the training set $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

Compute gradient:

Update biased first moment estimate:

Update biased second moment estimate:

Correct bias in first moment:

Correct bias in second moment:

Compute update:

Apply update:

**end while**
Choosing the right optimization algorithm

- It is highly problem dependent.
- If the input data is highly sparse than adaptive learning rates are recommended.
- Adaptive models are used when training a deep or complex neural network or when faster convergence is expected.
- Adam is the mostly used optimizer.
Example
gradient descent (cyan), momentum (magenta), AdaGrad (white), RMSProp (green), Adam (blue).
Second-Order methods

- It provides an addition curvature information of an objective function that adaptively estimate the step-length of optimization trajectory in training phase of neural network.

- This involves computing or approximating the matrix of second-order derivatives, i.e. the Hessian, in the context of exact deterministic optimization.

\[
H_f = \begin{bmatrix}
\frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
\frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2}
\end{bmatrix}
\]

- While second-order methods often have significantly better convergence properties than first-order methods, the size of typical problems prohibits their use in practice, as they require quadratic storage and cubic computation time for each gradient update.

- In all the methods here empirical risk is assumed to be
Newton’s Method

■ It is a widely used second-order gradient method.

■ It is an optimization method based on second order Taylor series expansion of $J()$ at some point $\theta_0$ after ignoring the higher orders.

■ This method added computational burden of calculation of inverse of the matrix and also not recommended for a function with saddle points.

■ For non-quadratic surfaces, as long as $H$ remains PD, Newton’s method can be applied.

■ Newton’s method would require the inversion of a $k \times k$ matrix—with computational complexity of $O(k^3)$. 
Newton’s method Algorithm

**Require:** Initial parameter

**Require:** Training set of $m$ examples

while stopping criterion not met do

  Compute gradient:

  Compute Hessian:

  Compute Hessian inverse: $H^{-1}$

  Compute update: $g$

  Apply update

end while
Conjugate gradients

Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending conjugate directions.

Motivation of this approach is line searches applied iteratively in the direction associated with the gradient.

At each step, next step is made in the direction using

Two directions are conjugate if

Two methods of calculation are:

Fletcher-Reeves:

$$
\beta_t = \frac{\nabla_{\theta} J(\theta_t)^T \nabla_{\theta} J(\theta_t)}{\nabla_{\theta} J(\theta_{t-1})^T \nabla_{\theta} J(\theta_{t-1})}
$$

Polak-Ribière:

$$
\beta_t = \frac{(\nabla_{\theta} J(\theta_t) - \nabla_{\theta} J(\theta_{t-1}))^T \nabla_{\theta} J(\theta_t)}{\nabla_{\theta} J(\theta_{t-1})^T \nabla_{\theta} J(\theta_{t-1})}
$$
Conjugate gradient method

Require: Initial parameters $\theta_0$

Require: Training set of $m$ examples

Initialize

Initialize $g_0 = 0$

Initialize $t = 1$

while stopping criterion not met do

Initialize the gradient $g_t = 0$

Compute gradient:

Compute

Compute search direction:

Perform line search to find:

Apply update:

end while
Outline

1. Batch Normalization
2. Supervised Pretraining
3. Designing Models to Aid Optimization
4. Continuation Methods and Curriculum Learning
5. Coordinate Descent and Polyak Averaging
Outline

1. Batch Normalization
2. Supervised Pretraining
3. Designing Models to Aid Optimization
4. Continuation Methods and Curriculum Learning
5. Coordinate Descent and Polyak Averaging
Batch Normalization – Motivation

- $\hat{y} = xw_1 w_2 \cdots w_k \ (\text{all } \in \mathbb{R})$
- gradient of $w_1$ derived under the assumption that $w_2, \ldots, w_k$ fixed
- BUT: update rule $w \leftarrow w - \epsilon g$
- expected: $\hat{y}$ decreases by $\epsilon g^T g$ (first-order Taylor approximation)
- actually: various higher-order effects, e.g. $\epsilon^2 g_1 g_2 \prod_{i=3}^{k} w_i$
Batch Normalization

**problem:** unwanted side-effects when applying gradient-descent to networks with many layers

**solutions**
- correction via higher-order methods
- very small learning rate
- batch normalization

**batch normalization:**
- normalize activation values after each linear transformation
- applicable to all layers except for the output layer
Batch Normalization

**Input:** $m$ examples $e^{(i)} \in \mathbb{R}^n$ ($n$ nodes in layer)

1. calculate $n$ means $\mu_j = \frac{1}{m} \sum_{i=1}^{m} e_j^{(i)}$

2. calculate $n$ standard deviations $\sigma_j = \sqrt{\delta + \frac{1}{m} \sum_{i=1}^{m} (e_j^{(i)} - \mu_j)^2}$ with a small positive $\delta$ (e.g. $10^{-8}$)

3. replace $e^{(i)}$ by $e'(i) = \left( \frac{e_1^{(i)} - \mu_1}{\sigma_1}, \frac{e_2^{(i)} - \mu_2}{\sigma_2}, \ldots, \frac{e_n^{(i)} - \mu_n}{\sigma_n} \right)^T$

**training:** back-propagate through this normalization operation

$\Rightarrow$ gradient will never propose changing mean or standard deviation

**test:** use averages over $\mu, \sigma$ collected during training
Batch Normalization – Example

\[ \hat{y} = x w_1 w_2 \cdots w_k \]

\( x \sim \mathcal{N}(0, 1) \)

linear transformation: \( x w_1 \sim \mathcal{N}(0, \sigma^2) \)

\( \sigma \) removed by batch normalization \( \Rightarrow x w_1 \sim \mathcal{N}(0, 1) \)

\( \Rightarrow \) only \( w_k \) has an (linear) effect on the output value \( \hat{y} \)
batch normalization removes linear effects of hidden layers, but preserves non-linear effects.

advantages [2]

- learning becomes more stable
- higher learning rates possible
- models less sensitive to initialization values
- regularization effect, similar to Dropout in some cases

⇒ faster and better training results
Outline

1. Batch Normalization
2. Supervised Pretraining
3. Designing Models to Aid Optimization
4. Continuation Methods and Curriculum Learning
5. Coordinate Descent and Polyak Averaging
Supervised Pretraining

sometimes training a model directly for a task is not possible

problems
  - hard to optimize model (e.g. very deep networks)
  - difficult task

goal: additional guidance for parameters in deep networks
Supervised Pretraining

- greedy supervised pretraining
  - multiple trainings before actual training
  - each trains only a subset of layers
  - assumption: pre-trained weights provide guidance for hidden layer parameters

adapted from [1, p. 324]
Supervised Pretraining

teacher student learning

- shallow and wide network (teacher) aids training of deep and thin network (student)
- secondary objective: predict middle-layer values of teacher network
  \[ \Rightarrow \] guidance on how to use the hidden layers
- example: student outperforms teacher on CIFAR-10 with 90% less parameters [5]
Supervised Pretraining

transfer learning

- train network on one task
- use weights to initialize training on a similar task
- *assumption*: networks learn some general abstraction that is useful for many tasks
Outline

1. Batch Normalization
2. Supervised Pretraining
3. Designing Models to Aid Optimization
4. Continuation Methods and Curriculum Learning
5. Coordinate Descent and Polyak Averaging
Designing Models to Aid Optimization

an easy to optimize model family is more important than a powerful optimization algorithm. [1, p. 326]

goal: local gradient information useful for reaching distant solution ⇒ as much (near-)linearity as possible, e.g. ReLu instead of sigmoids
Designing Models to Aid Optimization

challenge: ensure useful gradient information on low layers in deep networks

skip connections [6]
  - “highways” passing unchanged activation over several layers

auxiliary heads [7, 3]
  - additional nodes at hidden layers
  - trained to perform like output nodes, discarded after training
Outline

1. Batch Normalization
2. Supervised Pretraining
3. Designing Models to Aid Optimization
4. Continuation Methods and Curriculum Learning
5. Coordinate Descent and Polyak Averaging
Continuation Methods

- generate series of cost functions
  \( J^{(0)}, J^{(1)}, \ldots, J^{(n)} = J \) with increasing difficulty

\[ \Rightarrow \]
- keep local optimization in well-behaved regions
- often created by smoothing/blurring \( J \)
- intuition: non-convex function might become convex, but still preserve global minima

from [4]
Curriculum Learning

- first learn simple concept, then proceed to more complicated ones
- interpretation: series of $J^{(i)}$; cost functions with lower index depend on simpler examples
- successful in natural language processing and computer vision
Outline

1. Batch Normalization
2. Supervised Pretraining
3. Designing Models to Aid Optimization
4. Continuation Methods and Curriculum Learning
5. Coordinate Descent and Polyak Averaging
Coordinate Descent

concept: only update a subset of parameters at each optimization step

example: k-means

objective function:

\[ J(\mu_1, \ldots, \mu_k, S_1, \ldots, S_k) = \sum_{i=1}^{k} \sum_{x_j \in S_i} \|x_j - \mu_i\|^2 \quad (S_i \text{ partition}) \]

Lloyd’s algorithm:

1. Select \( k \) random cluster centers
2. repeat until convergence
   1. assign each instance to closest center ⇒ optimize w.r.t. \( S_i \)
   2. compute centers of this new clusters ⇒ optimize w.r.t. \( \mu_i \)

limitation: not applicable with strong dependencies between variables.
Polyak Averaging

- average over all locations visited by an optimization algorithm

\[ \hat{\theta}(t) = \frac{1}{t} \sum_{i=1}^{t} \theta(i) \]

\[ \Rightarrow \] strong convergence guarantees for some problem classes

- for neural networks: no guarantees, but performs well in practice
- common modification: exponentially decaying running average

adapted from [8]
Outline

1. Batch Normalization
2. Supervised Pretraining
3. Designing Models to Aid Optimization
4. Continuation Methods and Curriculum Learning
5. Coordinate Descent and Polyak Averaging


