Optimization for Deep Networks

Ishan Misra

Overview

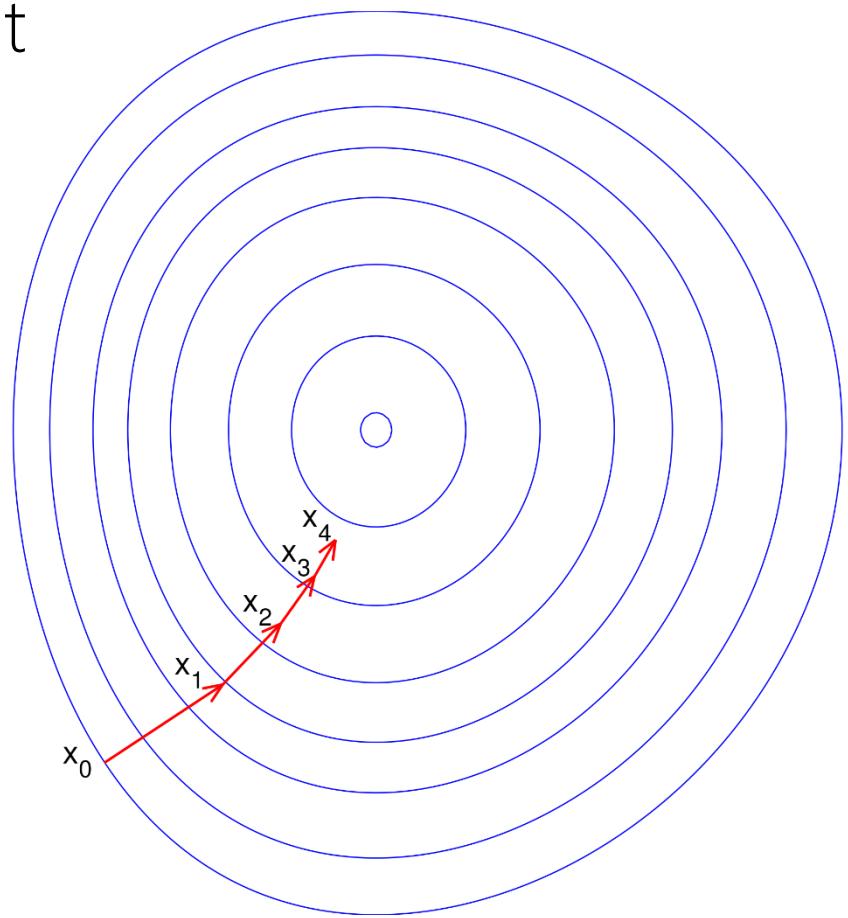
- Vanilla SGD
- SGD + Momentum
- NAG
- Rprop
- AdaGrad
- RMSProp
- AdaDelta
- Adam

More tricks

- Batch Normalization
- Natural Networks

Gradient (Steepest) Descent

- Move in the opposite direction of the gradient



Conjugate Gradient Methods

- See Moller 1993 [A scaled conjugate gradient algorithm for fast supervised learning], Martens et al., 2010 [Deep Learning via Hessian Free optimization]

Notation

θ Parameters of Network

f Function of network parameters

$v = f'$

Properties of Loss function for SGD

Loss function over all samples must decompose into a loss function per sample

$$f = \sum_{i=0}^{n} (x_i - y_i)^2$$

Vanilla SGD

$$v_{t+1} = \alpha f'(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

θ Parameters of Network

f Function of network parameters

$v = f'$

t Iteration number

α Step size/Learning rate

SGD + Momentum

- Plain SGD can make erratic updates on non-smooth loss functions
  - Consider an outlier example which “throws off” the learning process
- Maintain some history of updates
- Physics example
  - A moving ball acquires “momentum”, at which point it becomes less sensitive to the direct force (gradient)

SGD + Momentum

$$v_{t+1} = \alpha f'(\theta_t) + \mu v_t$$

$$\theta_{t+1} = \theta_t - v_{t+1}$$

θ	Parameters of Network
f	Function of network parameters
t	Iteration number
α	Step size/Learning rate
μ	Momentum

SGD + Momentum

- At iteration t you add updates from previous iteration t_{-n} by weight μ^n
- You effectively multiply your updates by $\frac{1}{1-\mu}$

Nesterov Accelerate Gradient (NAG)

- Ilya Sutskever, 2012

- First make a jump as directed by momentum
- Then depending on where you land, correct the parameters

NAG

$$\theta_{t+\frac{1}{2}} = \theta_t - \mu v_t$$

$$v_{t+1} = \alpha f'(\theta_{t+\frac{1}{2}})$$

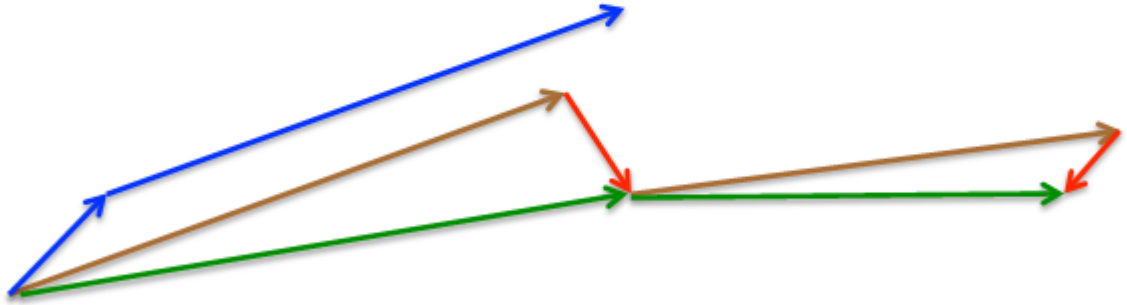
$$\theta_{t+1} = \theta_t - v_{t+1}$$

θ	Parameters of Network
f	Function of network parameters
t	Iteration number
α	Step size/Learning rate
μ	Momentum

NAG vs Standard Momentum

A picture of the Nesterov method

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Why anything new (beyond Momentum/NAG)?

- How to set learning rate and decay of learning rates?
- Ideally want adaptive learning rates

Why anything new (beyond Momentum/NAG)?

- Neurons in each layer learn differently
  - Gradient magnitudes vary across layers
  - Early layers get “vanishing gradients”
- Should ideally use separate adaptive learning rates
  - One of the reasons for having “gain” or lr multipliers in caffe
- Adaptive learning rate algorithms
  - Jacobs 1989 – agreement in sign between current gradient for a weight and velocity for that weight
  - Use larger mini-batches

Adaptive Learning Rates

Resilient Propagation (Rprop)

- Riedmiller and Braun 1993
- Address the problem of adaptive learning rate

- Increase the learning rate for a weight multiplicatively if signs of last two gradients agree
- Else decrease learning rate multiplicatively

Rprop Update

$$
\begin{aligned}
&\text{if} \quad f'_t f'_{t-1} > 0 \\
&\quad v_t = \eta^+ v_{t-1} \\
&\text{else if} \quad f'_t f'_{t-1} < 0 \\
&\quad v_t = \eta^- v_{t-1} \\
&\text{else} \\
&\quad v_t = v_t
\end{aligned}
$$

$$\theta_{t+1} = \theta_t - v_t$$

$$0 < \eta- < 1 < \eta+$$

Rprop Initialization

- Initialize all updates at iteration 0 to constant value
  - If you set both learning rates to 1, you get “Manhattan update rule”

$$v_0 = \delta$$

- Rprop effectively divides the gradient by its magnitude
  - You never update using the gradient itself, but by its sign

Problems with Rprop

- Consider a weight that gets updates of 0.1 in nine mini batches, and -0.9 in tenth mini batch
- SGD would keep this weight roughly where it started
- Rprop would increment weight nine times by δ, and then for the tenth update decrease the weight δ
  - Effective update $9\delta - \delta = 8\delta$
- Across mini-batches we scale updates very differently

Adaptive Gradient (AdaGrad)

- Duchi et al., 2010
- We need to scale updates across mini-batches similarly
- Use gradient updates as an indicator to scaling

$$r_t = \theta_t^2 + r_{t-1},$$

$$v_{t+1} = \frac{\alpha}{\sqrt{r_t}} f'(\theta_t),$$

$$\theta_{t+1} = \theta_t - v_{t+1}$$

Problems with AdaGrad

- Lowers the update size very aggressively

$$
\begin{aligned}
r_t &= \theta_t^2 + r_{t-1}, \\
v_{t+1} &= \frac{\alpha}{\sqrt{r_t}} f'(\theta_t), \\
\theta_{t+1} &= \theta_t - v_{t+1}
\end{aligned}
$$

RMSProp = Rprop + SGD

- Tieleman & Hinton et al., 2012 (Coursera slide 29, Lecture 6)
- Scale updates similarly across mini-batches
- Scale by **decaying average of squared gradient**
  - Rather than the sum of squared gradients in AdaGrad

$$r_t = (1-\gamma)\ f'(\theta_t)^2 + \gamma r_{t-1},$$

$$v_{t+1} = \frac{\alpha}{\sqrt{r_t}} f'(\theta_t),$$

$$\theta_{t+1} = \theta_t - v_{t+1}$$

RMSProp

- Has shown success for training Recurrent Models
- Using Momentum generally does not show much improvement

Fancy RMSProp

- “No more pesky learning rates” – Schaul et al.
  - Computes a diagonal Hessian and uses something similar to RMSProp
  - Diagonal Hessian computation requires an additional Forward-Backward pass
    - Double the time of SGD

Units of update

- SGD update is in terms of gradient

$$v_t = \alpha f',$$
$$\theta_{t+1} = \theta_t + v_t$$

$$f' \propto \frac{1}{\text{units of } \theta}$$

Unitless updates

- Updates are not in units of parameters

$$
\begin{aligned}
r_t &= (1-\gamma)\ f'(\theta_t)^2 + \gamma r_{t-1}, \\
v_{t+1} &= \frac{\alpha}{\sqrt{r_t}} f'(\theta_t), \\
\theta_{t+1} &= \theta_t - v_{t+1}
\end{aligned}
$$

Hessian updates

$$v_{t+1} = H^{-1} f',$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

Hessian gives correct units

$$
\begin{aligned}
v_{t+1} &= H^{-1} f', \\
\theta_{t+1} &= \theta_t - v_{t+1}
\end{aligned}
$$

$$
\begin{aligned}
v_{t+1} &= H^{-1} f', \\
&\propto \frac{f'}{f''} \\
&\propto \frac{1/\text{units of } \theta}{(1/\text{units of } \theta)^2} \\
&\propto \text{units of } \theta
\end{aligned}
$$

AdaDelta

- Zeiler et al., 2012
- Get updates that match units
- Keep properties from RMSProp
- Updates should be of the form

$$
\begin{aligned}
v_{t+1} &= H^{-1} f', \\
&\propto \frac{f'}{f''} \\
&\propto \frac{1/\text{units of } \theta}{(1/\text{units of } \theta)^2} \\
&\propto \text{units of } \theta
\end{aligned}
$$

AdaDelta

- Approximate denominator by **sqrt(decaying average of gradient squares)**
- Approximate numerator by **decaying average of squared updates**

$$v_{t+1} \propto \text{units of } \theta$$

$$\propto \text{gradient} \frac{\text{RMS(updates)}}{\text{RMS(gradient)}}$$

AdaDelta

- Approximate denominator by **sqrt(decaying average of gradient squares)**
- Approximate numerator by **decaying average of squared updates**

$$v_{t+1} \propto \text{units of } \theta$$

$$\propto \text{gradient} \frac{\text{RMS(updates)}}{\text{RMS(gradient)}}$$

$$p_{t+1} = (1 - \gamma_2) v_t^2 + \gamma_2 p_t$$

$$r_t = (1 - \gamma_1) f'(\theta_t)^2 + \gamma_1 r_{t-1}$$

AdaDelta Update Rule

$$
\begin{aligned}
r_t &= (1-\gamma_1) f'(\theta_t)^2 + \gamma_1 r_{t-1}, \\
v_t &= \alpha \frac{\sqrt{p_t}}{\sqrt{r_t}} f'(\theta_t) \\
\theta_{t+1} &= \theta_t - v_t \\
p_{t+1} &= (1-\gamma_2) v_t^2 + \gamma_2 v_t
\end{aligned}
$$

Problems with AdaDelta

- The moving averages are biased by initialization of decay parameters

$$p_{t+1} = (1 - \gamma_2) v_t^2 + \gamma_2 p_t$$
$$r_t = (1 - \gamma_1) f'(\theta_t)^2 + \gamma_1 r_{t-1}$$

Problems with AdaDelta

- Not the first intuitive shot at fixing “unit updates” problem
  - This just maybe me nitpicking
- Why do updates have a time delay?
  - There is some explanation in the paper, not convinced …

$$p_{t+1} = (1 - \gamma_2) v_t^2 + \gamma_2 p_t$$
$$r_t = (1 - \gamma_1) f'(\theta_t)^2 + \gamma_1 r_{t-1}$$

Adam

- Kingma & Ba, 2015
- Averages of gradient, or squared gradients
- Bias correction

$$
\begin{aligned}
r_t &= (1-\gamma_1)\ f'(\theta_t) + \gamma_1 r_{t-1}, \\
p_t &= (1-\gamma_2) f'(\theta_t)^2 + \gamma_2 p_{t-1} \\
\hat{r}_t &= \frac{r_t}{(1-(1-\gamma_1)^t)} \\
\hat{p}_t &= \frac{p_t}{(1-(1-\gamma_2)^t)}
\end{aligned}
$$

Adam update rule

$$
\begin{aligned}
r_t &= (1-\gamma_1)\ f'(\theta_t) + \gamma_1 r_{t-1}, \\
p_t &= (1-\gamma_2) f'(\theta_t)^2 + \gamma_2 p_{t-1} \\
\hat{r}_t &= \frac{r_t}{(1-(1-\gamma_1)^t)} \\
\hat{p}_t &= \frac{p_t}{(1-(1-\gamma_2)^t)} \\
v_t &= \alpha \frac{\hat{r}_t}{\sqrt{\hat{p}_t}} \\
\theta_{t+1} &= \theta_t - v_t
\end{aligned}
$$

Updates are not in the correct unit ☹

Simplification, does not have decay over gamma_1

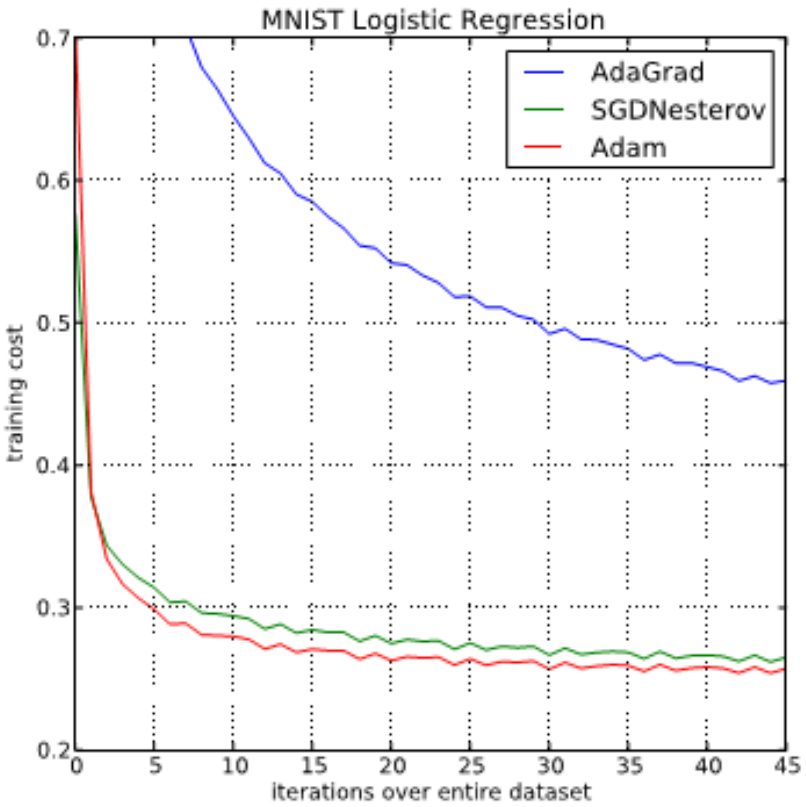
AdaMax

$$r_t = (1-\gamma_1)\ f'(\theta_t) + \gamma_1 r_{t-1},$$
$$p_t = \max(\gamma_2 p_{t-1}, |f'(\theta_t)|)$$
$$\hat{r}_t = \frac{r_t}{(1-(1-\gamma_1)^t)}$$
$$v_t = \alpha \frac{\hat{r}_t}{p_t}$$
$$\theta_{t+1} = \theta_t - v_t$$

Adam

$$r_t = (1-\gamma_1)\ f'(\theta_t) + \gamma_1 r_{t-1},$$
$$p_t = (1-\gamma_2) f'(\theta_t)^2 + \gamma_2 p_{t-1}$$
$$\hat{r}_t = \frac{r_t}{(1-(1-\gamma_1)^t)}$$
$$\hat{p}_t = \frac{p_t}{(1-(1-\gamma_2)^t)}$$
$$v_t = \alpha \frac{\hat{r}_t}{\sqrt{\hat{p}_t}}$$
$$\theta_{t+1} = \theta_t - v_t$$

Adam Results – Logistic Regression



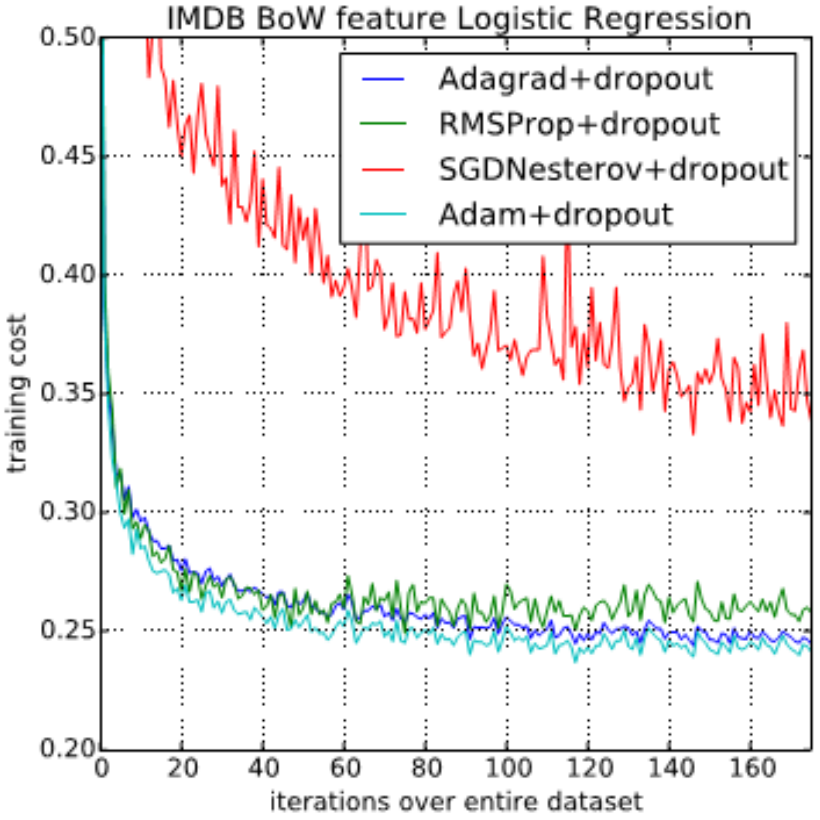
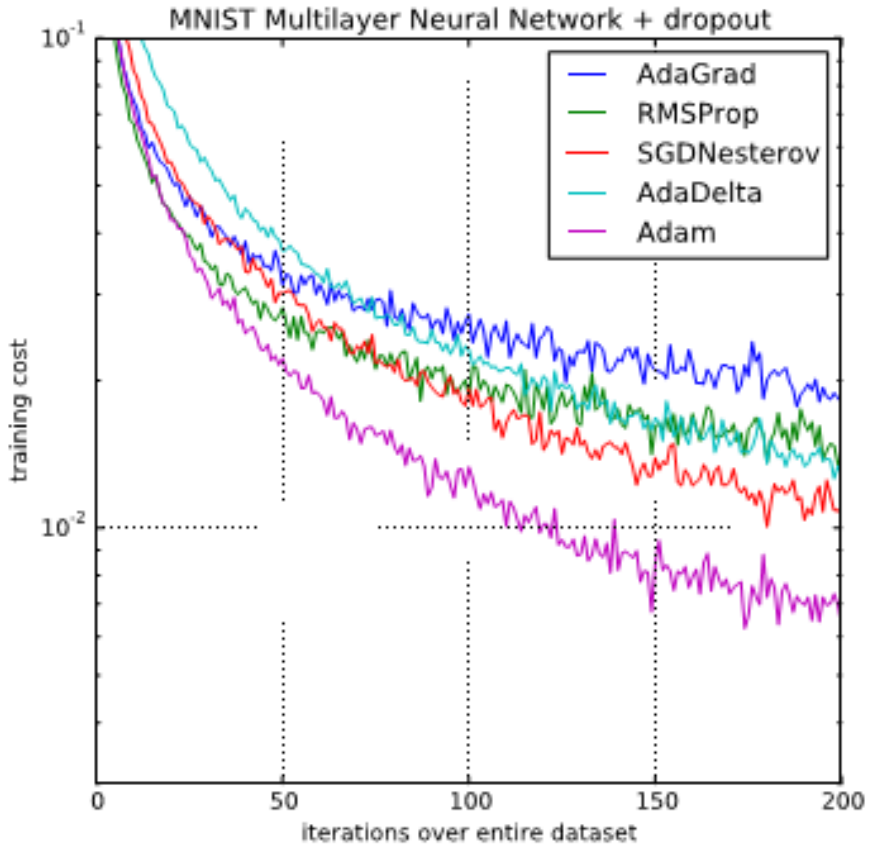


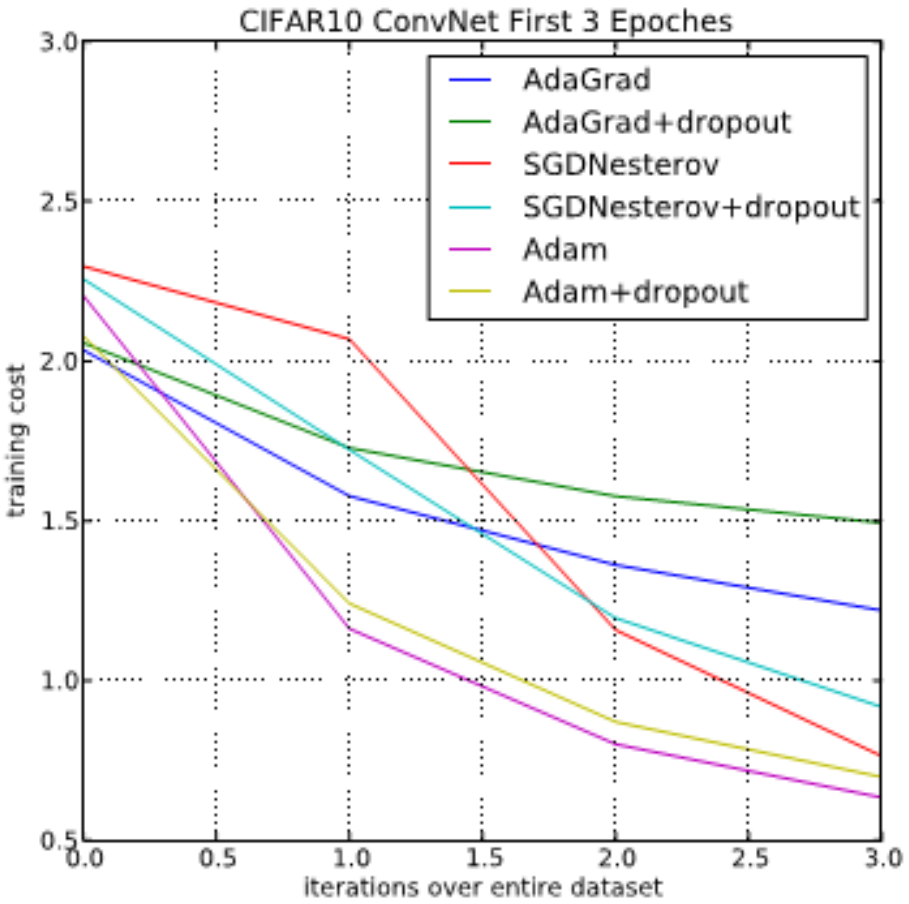
Figure 1: Logistic regression training negative log likelihood on MNIST images and IMDB movie reviews with 10,000 bag-of-words (BoW) feature vectors.

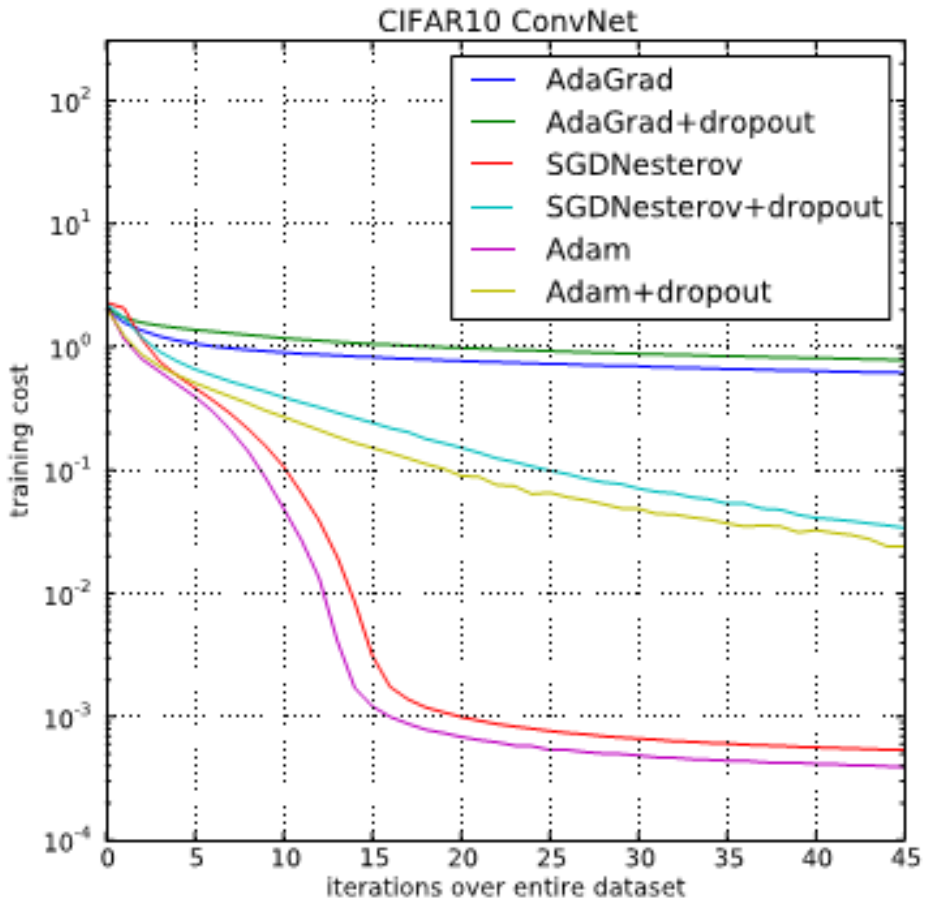
Adam Results - MLP



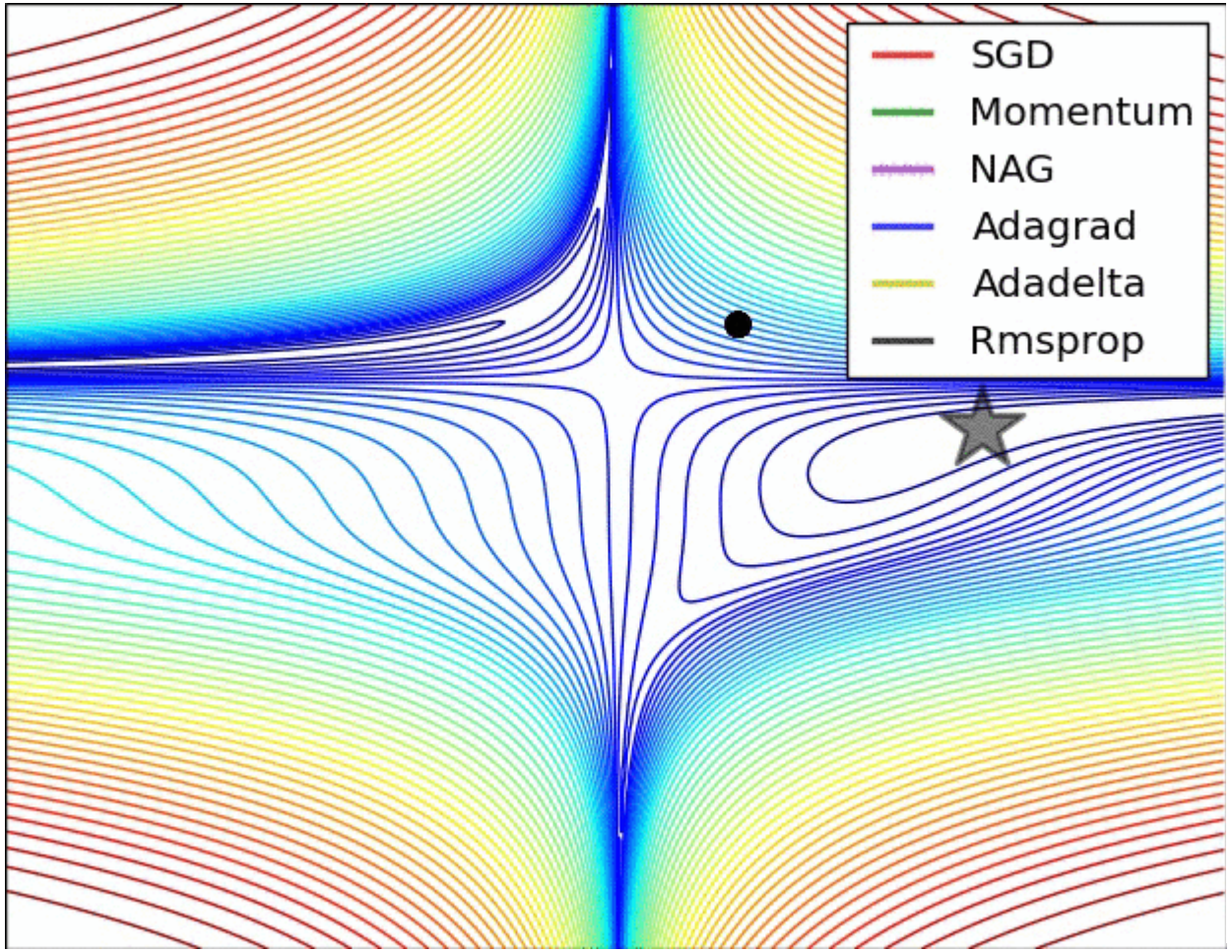
(a)

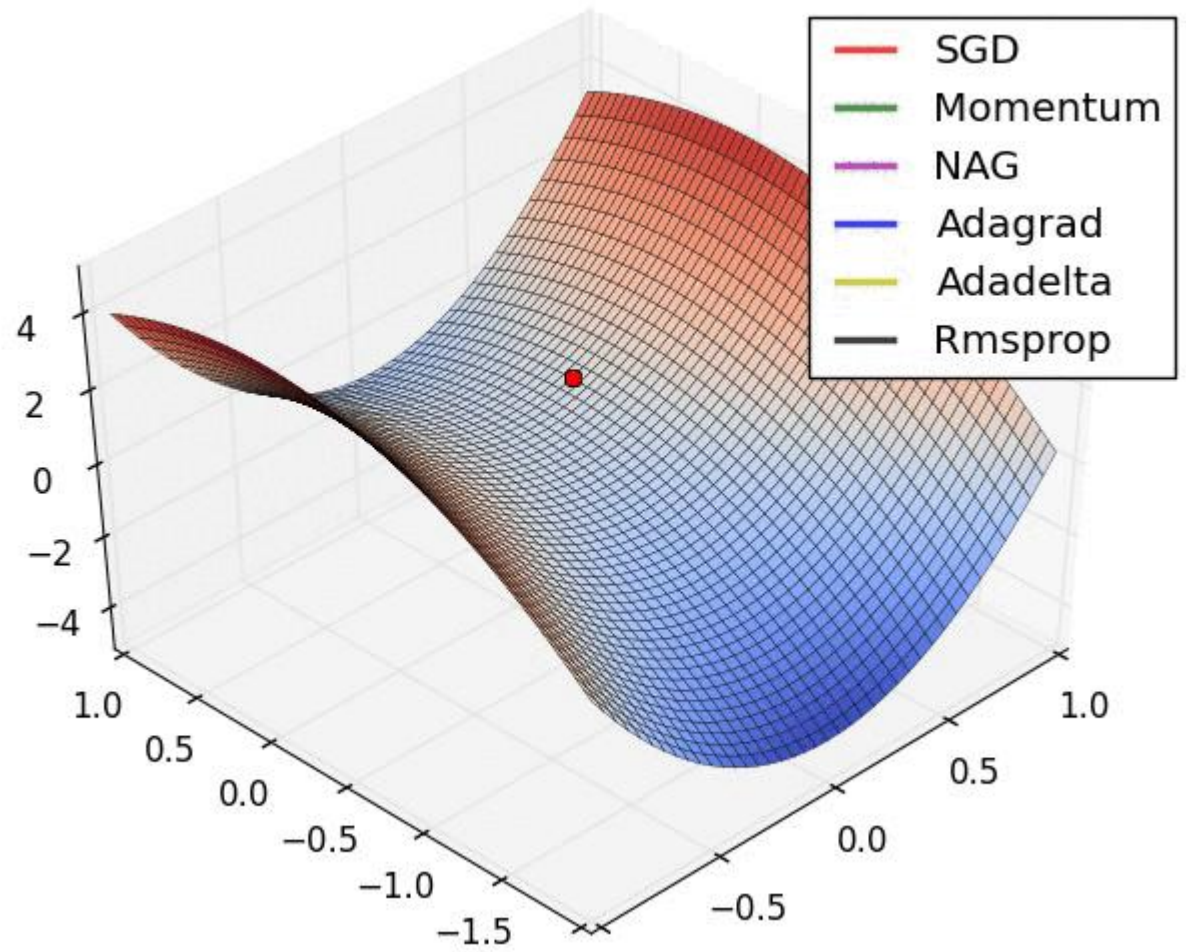
Adam Results – Conv Nets





Visualization





Alec Radford

Batch Normalization

Ioffe and Szegedy, 2015

Distribution of input

- Having a fixed input distribution is known to help training of linear classifiers
  - Normalize inputs for SVM
  - Normalize inputs for Deep Networks

Distribution of input at each layer

- Each layer would benefit if its input had constant distribution

Normalize the input to each layer!

$$\mu_x = \frac{1}{m}\sum_{i=0}^{m} x_i$$

$$\sigma_x^2 = \frac{1}{m}\sum_{i=0}^{m}(x_i - \mu_x)^2$$

$$\hat{x_i} = \frac{x_i - \mu_x}{\sigma_x}$$

Is this normalization a good idea?

- Consider inputs to a sigmoid layer
- If normalized, the sigmoid may not ever “saturate”

Modify normalization …

- Accommodate identity transform

$$\mu_x = \frac{1}{m}\sum_{i=0}^{m} x_i$$

$$\sigma_x^2 = \frac{1}{m}\sum_{i=0}^{m}(x_i - \mu_x)^2$$

$$\hat{x}_i = \frac{x_i - \mu_x}{\sigma_x}$$

$$y_i = \gamma\hat{x}_i + \beta$$

γ, β Learned parameters

Batch Normalization Results

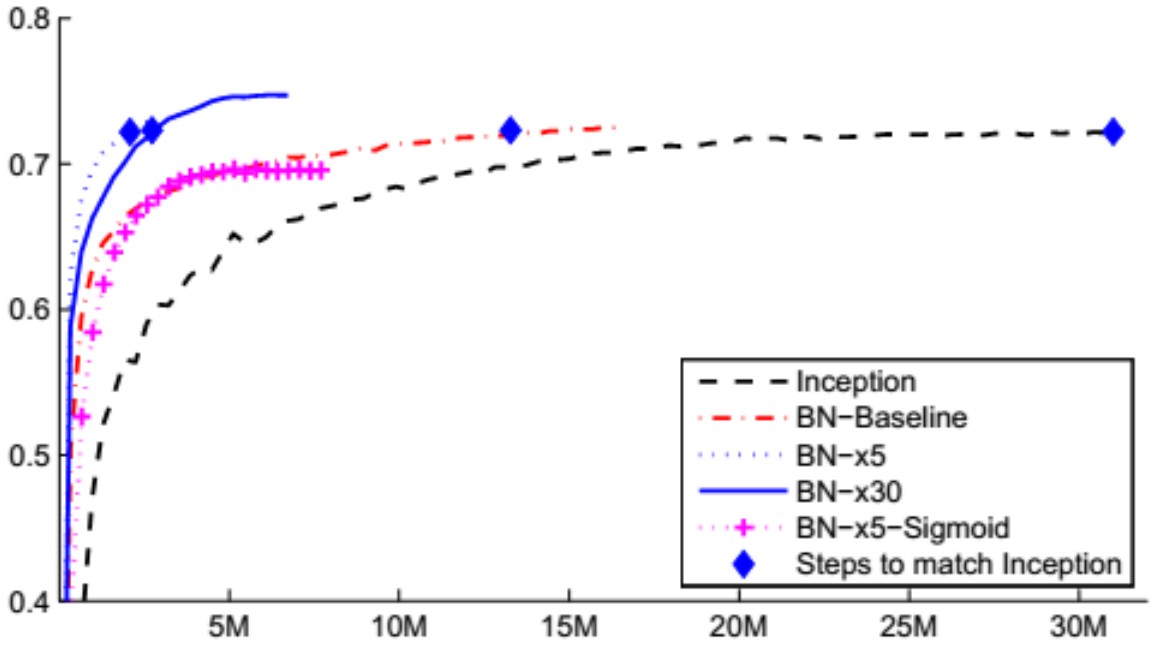


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Figure 3: *For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.*

Batch Normalization for ensembles

Model	Resolution	Crops	Models	Top-1 error	Top-5 error
GoogLeNet ensemble	224	144	7	-	6.67%
Deep Image low-res	256	-	1	-	7.96%
Deep Image high-res	512	-	1	24.88	7.42%
Deep Image ensemble	variable	-	-	-	5.98%
BN-Inception single crop	224	1	1	25.2%	7.82%
BN-Inception multicrop	224	144	1	21.99%	5.82%
BN-Inception ensemble	224	144	6	20.1%	**4.9%***

Figure 4: *Batch-Normalized Inception comparison with previous state of the art on the provided validation set comprising 50000 images. *BN-Inception ensemble has reached 4.82% top-5 error on the 100000 images of the test set of the ImageNet as reported by the test server.*

Natural Gradients

Related work: PRONG or Natural Neural Networks

Gradients and orthonormal coordinates

- In Euclidean space, we define length using orthonormal coordinates

$$\|\theta\|^2 = \sum_{i=0}^{n} (\theta_i)^2$$

What happens on a manifold?

- Use metric tensors (generalizes dot product to manifolds)

$$\|\theta\|^2 = \sum_{i,j} g_{ij}\theta_i\theta_j$$

G is generally a symmetric PSD matrix.
It is a tensor because G' = J^T G J, where J = jacobian(G)

Gradient descent revisited

- What exactly does a gradient descent update step do?

$$v_{t+1} = \alpha f'(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

Gradient descent revisited

- What exactly does a gradient descent update step do?

$$v_{t+1} = \alpha f'(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

Distance is measured in orthonormal coordinates

$$\theta_{t+1} = \arg\min_{\theta} \left(< \theta, f' > + \frac{1}{2\alpha_t} \|\theta - \theta_t\|_2^2 \right)$$

Relationship: Natural Gradient vs. Gradient

$$f'_N = F_\theta^{-1} f'$$

F is the Fisher information matrix

PRONG: Projected Natural Gradient Descent

- Main idea is to avoid computing inverse of Fisher matrix
- Reparametrize the network so that Fisher matrix is identity
  - Hint: Whitening …

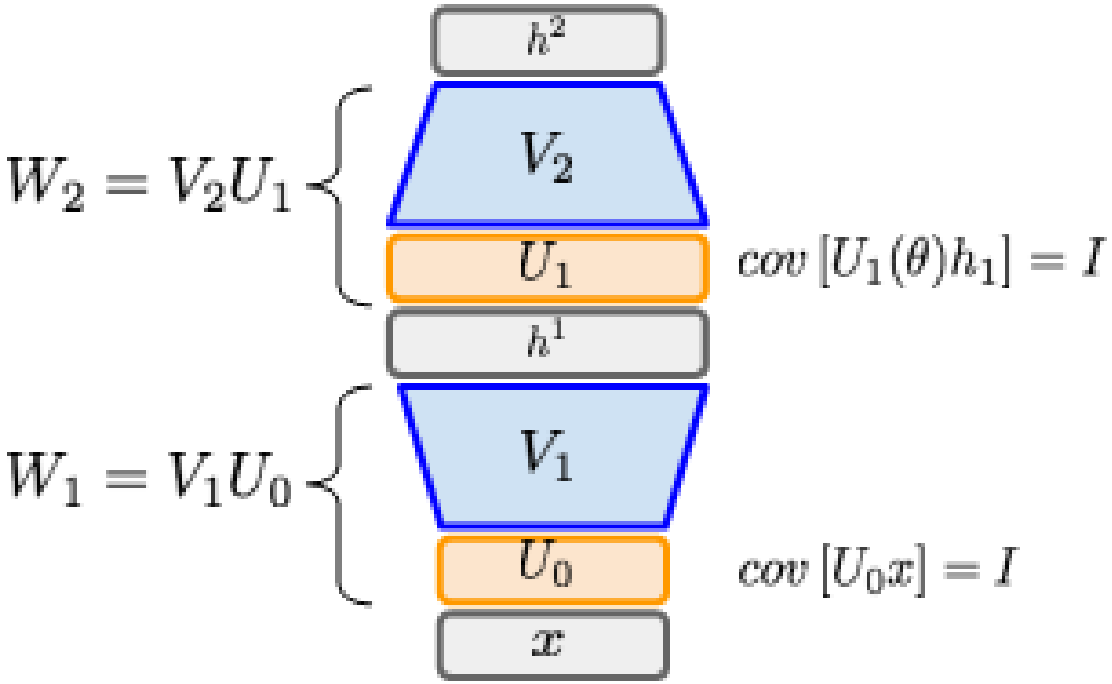
Reparametrization

- Each layer has two components

V_i The old θ contains all the weights; **Learned**

U_i Whitening matrix; **Estimated**

Reparametrization



$$h_i = f_i(V_iU_{i-1}(h_{i-1} - c_i) + d_i)$$

Exact forms of weights

- U_i is the normalized eigen-decomposition of $\boldsymbol{\Sigma}_i$

PRONG Update rule

Algorithm 1 Projected Natural Gradient Descent

Input: training set $\mathcal{D}$, initial parameters θ.
Hyper-parameters: reparam. frequency T, number of samples N_s, regularization term ϵ.
$U_i \leftarrow I; c_i \leftarrow 0; t \leftarrow 0$
repeat
  if $mod(t, T) = 0$ **then** ▷ amortize cost of lines [6-11]
    for all layers i **do**
      Compute canonical parameters $W_i = V_i U_{i-1}$; $b_i = d_i + W_i c_i$. ▷ proj. $P_\Phi^{-1}(\Omega)$
      Estimate μ_i and Σ_i, using N_s samples from $\mathcal{D}$.
      Update c_i from μ_i and U_i from eigen decomp. of $\Sigma_i + \epsilon I$. ▷ update Φ
      Update parameters $V_i \leftarrow W_i U_{i-1}^{-1}$; $c_i \leftarrow b_i - V_i c_i$. ▷ proj. $P_\Phi(\theta)$
    end for
  end if
  Perform SGD update wrt. Ω using samples from $\mathcal{D}$.
  $t \leftarrow t + 1$
until convergence

Similarity to Batch Normalization

The recently introduced batch normalization (BN) scheme [7] quite closely resembles a *diagonal* version of PRONG, the main difference being that BN normalizes the variance of activations *before* the non-linearity, as opposed to normalizing the latent activations by looking at the full covariance. Furthermore, BN implements normalization by modifying the feed-forward computations thus requiring the method to backpropagate through the normalization operator.

PRONG: Results

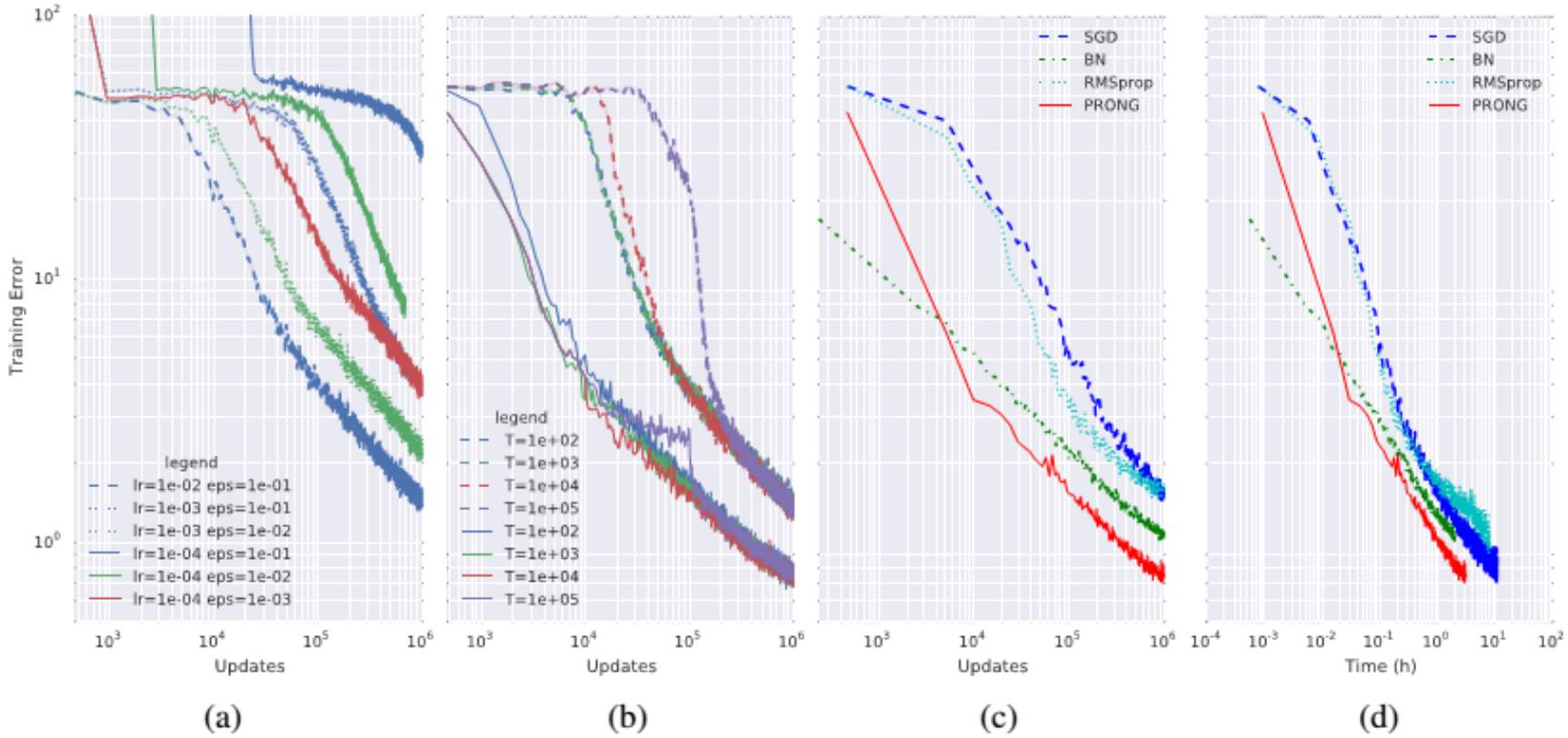


Figure 3: Optimizing a deep auto-encoder on MNIST. (a) Impact of eigenvalue regularization term ϵ. (b) Impact of amortization period T showing that initialization with the whitening reparametrization is important for achieving faster learning and better error rate. (c) Training error vs number of updates. (d) Training error vs cpu-time. Plots (c) and (d) show that PRONG achieves better error rate both in number of updates and wall clock time.

PRONG: Results

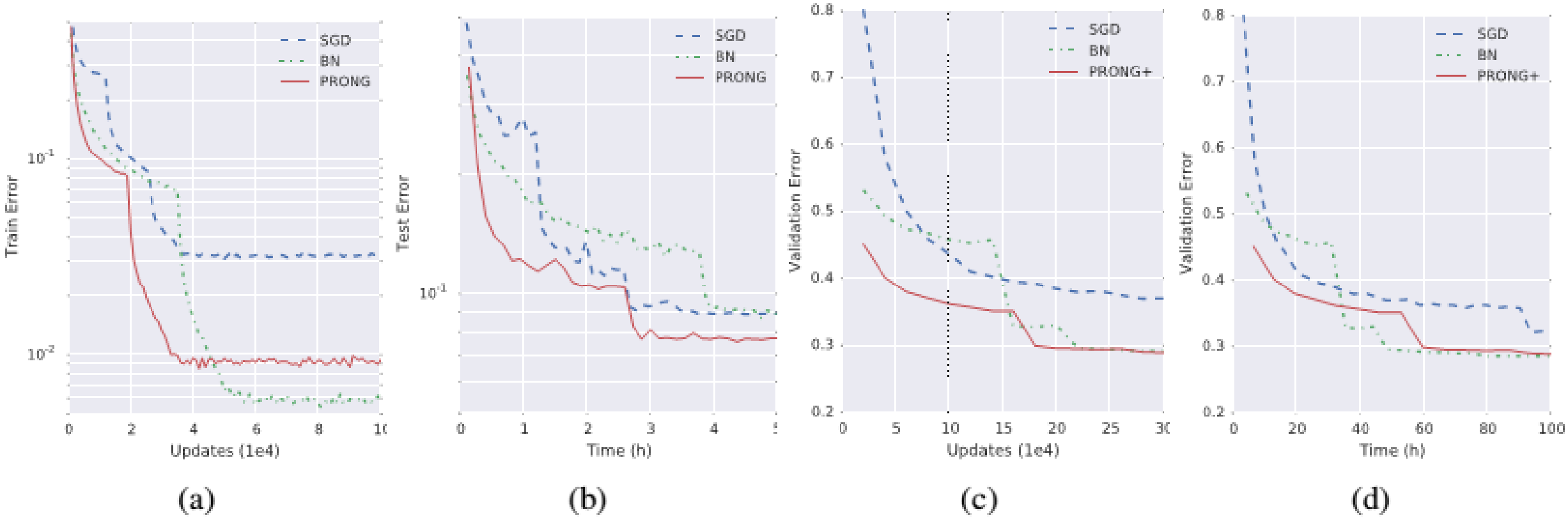


Figure 4: Classification error on CIFAR-10 (a-b) and ImageNet (c-d). On CIFAR-10, PRONG achieves better test error and converges faster. On ImageNet, PRONG$^+$ achieves comparable validation error while maintaining a faster covergence rate.

Thanks!

References

- http://climin.readthedocs.org/en/latest/rmsprop.html
- http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
- Training a 3-Node Neural Network is NP-Complete (Blum and Rivest, 1993)
- Equilibrated adaptive learning rates for non-convex optimization
- Practical Recommendations for Gradient-Based Training of Deep Architectures
- Efficient BackPropagation - Yann Le Cun 1989
- Stochastic Gradient Descent Tricks – Leon Bottou