# Variance-based Stochastic Gradient Descent (vSGD):

No More Pesky Learning Rates

Schaul et al., ICML13

# The idea

- Remove need for setting learning rates by updating them optimally from the Hessian values.

**vSGD-l** uses *local* gradient variance terms and the local diagonal Hessian estimates, leading to $\eta_i^* = (\overline{g_i})^2/(\overline{h_i} \cdot \overline{v_i})$,

**vSGD-g** uses a *global* gradient variance term and an upper bound on diagonal Hessian terms: $\eta^* = \sum(\overline{g_i})^2/(h^+ \cdot \bar{l})$,

**vSGD-b** operates like vSGD-g, but being only global across multiple (architecture-specific) *blocks* of parameters, with a different learning rate per block. Similar ideas are adopted in TONGA (Le Roux et al., 2008). In the experiments, the parameters connecting every two layers of the network are regard as a block, with the corresponding bias parameters in separate blocks.

**Algorithm 1:** Stochastic gradient descent with adaptive learning rates (element-wise, vSGD-l).

**repeat**

draw a sample $c^{(j)}$, compute the gradient $\nabla_{\boldsymbol{\theta}}^{(j)}$, and compute the diagonal Hessian estimates $h_i^{(j)}$ using the "bbprop" method

**for** $i \in \{1, \ldots, d\}$ **do**

update moving averages

$$\overline{g_i} \leftarrow (1 - \tau_i^{-1}) \cdot \overline{g_i} + \tau_i^{-1} \cdot \nabla_{\theta_i}^{(j)}$$

$$\overline{v_i} \leftarrow (1 - \tau_i^{-1}) \cdot \overline{v_i} + \tau_i^{-1} \cdot \left(\nabla_{\theta_i}^{(j)}\right)^2$$

$$\overline{h_i} \leftarrow (1 - \tau_i^{-1}) \cdot \overline{h_i} + \tau_i^{-1} \cdot \left|\text{bbprop}(\boldsymbol{\theta})_i^{(j)}\right|$$

estimate learning rate $\eta_i^* \leftarrow \frac{(\overline{g_i})^2}{\overline{h_i} \cdot \overline{v_i}}$

update memory size

$\tau_i \leftarrow \left(1 - \frac{(\overline{g_i})^2}{\overline{v_i}}\right) \cdot \tau_i + 1$

update parameter $\theta_i \leftarrow \theta_i - \eta_i^* \nabla_{\theta_i}^{(j)}$

**end**

**until** *stopping criterion is met*

| | vSGD-l | vSGD-b | vSGD-g | SGD | AdaGrad | SMD | Amari | Almeida |
|---|---|---|---|---|---|---|---|---|
| M0 | **6.72%** | 7.63% | 8.20% | 7.05% | 6.97% | 7.02% | 7.33% | 11.80% |
| M1 | **0.18%** | 0.78% | 3.50% | 0.30% | 0.58% | 0.40% | 2.91% | 8.49% |
| M2 | **0.05%** | 0.33% | 2.91% | 0.46% | 0.41% | 0.55% | 1.68% | 7.16% |
| C0 | **45.61%** | 52.45% | 56.16% | 54.78% | 54.36% | – | – | – |
| C1 | **33.16%** | 45.14% | 54.91% | 47.12% | 45.20% | – | – | – |
| CR | 10.64 | 10.13 | 15.37 | **9.77** | **9.80** | – | – | – |

*Table 2.* Final classification error (and reconstruction error for CIFAR-2R) on the **training** set, obtained after 6 epochs of training, and averaged over ten random initializations. Variants are marked in bold if they don't differ statistically significantly from the best one ($p = 0.01$). Note that the tuning parameters of SGD, AdaGrad, SMD, Amari and Almeida are different for each benchmark (see Table 1). We observe the best results with the full element-wise learning rate adaptation ('vSGD-l'), almost always significantly better than the best-tuned SGD or best-tuned AdaGrad.

| | vSGD-l | vSGD-b | vSGD-g | SGD | AdaGrad | SMD | Amari | Almeida |
|---|---|---|---|---|---|---|---|---|
| M0 | **7.50%** | 7.89% | 8.20% | **7.60%** | **7.52%** | **7.57%** | **7.69%** | 11.13% |
| M1 | **2.42%** | **2.44%** | 4.14% | **2.34%** | 2.70% | **2.37%** | 3.95% | 8.39% |
| M2 | **2.16%** | **2.05%** | 3.65% | **2.15%** | 2.34% | **2.18%** | 2.97% | 7.32% |
| C0 | 66.05% | 61.70% | **61.10%** | **61.06%** | **61.25%** | – | – | – |
| C1 | **57.72%** | 59.55% | 60.62% | 58.85% | 58.67% | – | – | – |
| CR | 11.05 | 10.57 | 15.71 | **10.29** | **10.33** | – | – | – |
| #settings | 1 | 1 | 1 | 68 | 17 | 476 | 119 | 119 |

*Table 3.* Final classification error (and reconstruction error for CIFAR-2R) on the **test** set, after 6 epochs of training, averaged over ten random initializations. Variants are marked in bold if they don't differ statistically significantly from the best one ($p = 0.01$). Note that the parameters of SGD, AdaGrad, SMD, Amari and Almeida were finely tuned, on this same test set, and were found to be different for each benchmark (see Table 1); the last line gives the total number of parameter settings over which the tuning was performed. Compared to training error, test set performance is more balanced, with vSGD-l being better or statistically equivalent to the best-tuned SGD in 4 out of 6 cases. The main outlier (C0) is a case where the more aggressive element-wise learning rates led to overfitting (compare training error in Table 2).

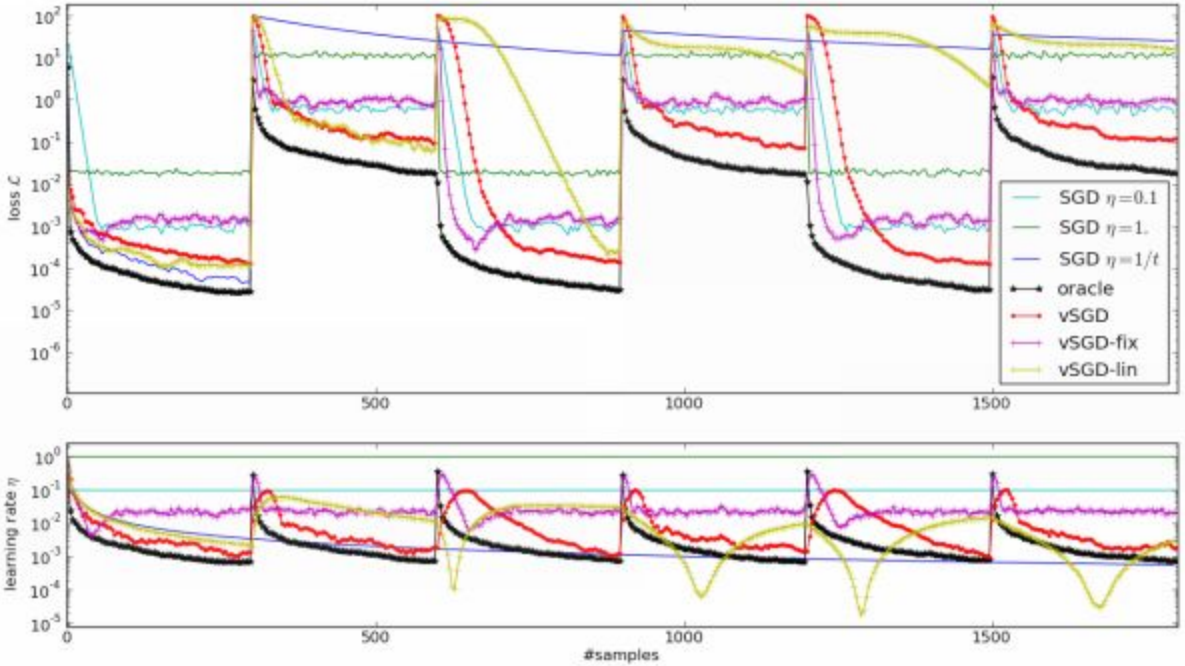



*Figure 4.* Non-stationary loss. The loss is quadratic but now the target value ($\mu$) changes abruptly every 300 time-steps. Above: loss as a function of time, below: corresponding learning rates. This illustrates the limitations of SGD with fixed or decaying learning rates (full lines): any fixed learning rate limits the precision to which the optimum can be approximated (progress stalls); any cooling schedule on the other hand cannot cope with the non-stationarity. In contrast, our adaptive setting ('vSGD', red circles), as closely resembles the optimal behavior (oracle, black dashes). The learning rate decays like $1/t$ during the static part, but increases again after each abrupt change (with just a very small delay compared to the oracle). The average loss across time is substantially better than for any SGD cooling schedule.

# ADAM:

A Method For Stochastic Optimization

Kingma & Ba, arXiv14

# The idea

- Establish and update trust region where the gradient is assumed to hold.
- Attempts to combine the robustness to sparse gradients of AdaGrad and the robustness of RMSProp to non-stationary objectives.

**Algorithm 1:** *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. $g_t^2$ indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With $\beta_1^t$ and $\beta_2^t$ we denote $\beta_1$ and $\beta_2$ to the power $t$.

**Require:** $\alpha$: Stepsize
**Require:** $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  **while** $\theta_t$ not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
  **end while**
  **return** $\theta_t$ (Resulting parameters)

# Alternative form: AdaMax

- The second moment is calculated as a sum of squares and its square root is used in the update in ADAM.
- Changing that from power of two to power of p as p goes to infinity yields AdaMax.

**Algorithm 2:** *AdaMax*, a variant of Adam based on the infinity norm. See section 7.1 for details. Good default settings for the tested machine learning problems are $\alpha = 0.002$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. With $\beta_1^t$ we denote $\beta_1$ to the power $t$. Here, $(\alpha/(1-\beta_1^t))$ is the learning rate with the bias-correction term for the first moment. All operations on vectors are element-wise.

**Require:** $\alpha$: Stepsize
**Require:** $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $u_0 \leftarrow 0$ (Initialize the exponentially weighted infinity norm)
  $t \leftarrow 0$ (Initialize timestep)
  **while** $\theta_t$ not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $u_t \leftarrow \max(\beta_2 \cdot u_{t-1}, |g_t|)$ (Update the exponentially weighted infinity norm)
    $\theta_t \leftarrow \theta_{t-1} - (\alpha/(1-\beta_1^t)) \cdot m_t/u_t$ (Update parameters)
  **end while**
  **return** $\theta_t$ (Resulting parameters)

# Results

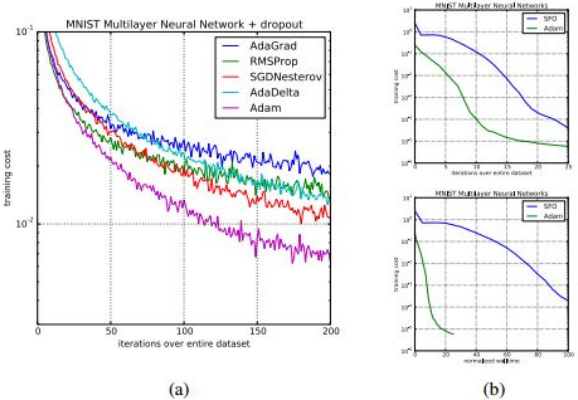



Figure 2: Training of multilayer neural networks on MNIST images. (a) Neural networks using dropout stochastic regularization. (b) Neural networks with deterministic cost function. We compare with the sum-of-functions (SFO) optimizer (Sohl-Dickstein et al., 2014)

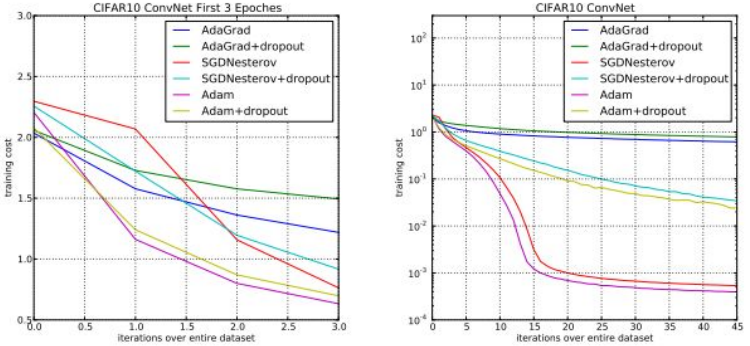



Figure 3: Convolutional neural networks training cost. (left) Training cost for the first three epochs. (right) Training cost over 45 epochs. CIFAR-10 with c64-c64-c128-1000 architecture.

# AdaGrad:

Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

Duchi et al., COLT10

# The idea

- Decrease the update over time by penalizing quickly moving values.

$$\Delta x_t = -\frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}}\, g_t$$

# The problem

- The learning rate only ever decreases.
- Complex problems may need more freedom.

# Precursor to

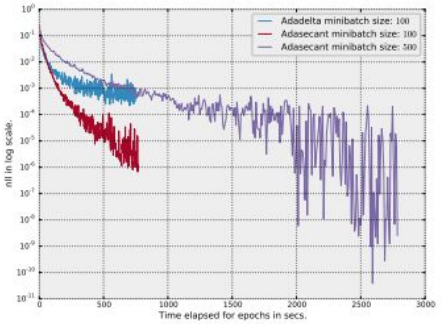
- AdaDelta (Zeiler, ArXiv12)
  - Uses the square root of exponential moving average of squares instead of just accumulating.
  - Approximate a Hessian correction using the same moving impulse over the weight updates.
  - Removes need for learning rate

$$E[g^2]_t = \rho\, E[g^2]_{t-1} + (1-\rho)\, g_t^2 \qquad \mathrm{RMS}[g]_t = \sqrt{E[g^2]_t + \epsilon}$$

$$\Delta x_t = -\frac{\mathrm{RMS}[\Delta x]_{t-1}}{\mathrm{RMS}[g]_t}\, g_t$$

- AdaSecant (Gulcehre et al., ArXiv14)
  - Uses expected values to reduce variance.

$$\tilde{g}_i = \frac{g_i + \gamma_i \mathrm{E}[g_i]}{1+\gamma_i} \qquad \mathrm{E}[\tilde{g}_i] = \mathrm{E}[g_i] \text{ and } \mathrm{Var}(\tilde{g}_i) = \frac{1}{(1+\gamma_i)^2}\mathrm{Var}(g_i)$$



# Comparisons

- https://cs.stanford.edu/people/karpathy/convnetjs/demo/trainers.html
- Doesn’t have ADAM in the default run, but ADAM is implemented and can be added.
- Doesn’t have Batch Normalization, vSGD, AdaMax, or AdaSecant.

# Questions?