# Lecture 8: spectra of deep net (Pennington et al. 18'[1])

$$h^{(\ell)} = \phi(\hat{h}^{(\ell)}) \quad \text{with} \quad \hat{h}^{(\ell)} = W^{(\ell)} h^{(\ell-1)} + b^{(\ell)}$$

has input to output Jacobian given by

$$J = \frac{\partial h^{(L)}}{\partial x^{(0)}} = \Pi_{\ell=1}^{L} D^{(\ell)} W^{(\ell)}$$

where $D^{(\ell)}$ is diagonal with entries $D_{ii}^{(\ell)} = \phi'(\hat{h}_i^{(\ell)})$.
Stability: $\|H(x+\delta;\theta) - H(x;\theta)\| \leq \|\delta\| \max \|J\|$.

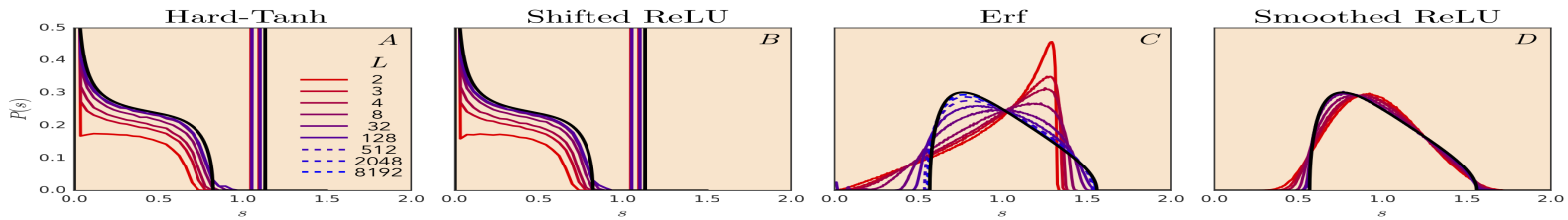



Figure 4: Two limiting universality classes of Jacobian spectra. Hard Tanh and Shifted ReLU fall into one class, characterized by Bernoulli-distributed $\phi'(h)^2$, while Erf and Smoothed ReLU fall into a second class, characterized by a smooth distribution for $\phi'(h)^2$. The black curves are theoretical predictions for the limiting distributions with variance $\sigma_0^2 = 1/4$. The colored lines are emprical spectra of finite-depth width-1000 orthogonal neural networks. The empirical spectra converge to the limiting distributions in all cases. The rate of convergence is similar for Hard-Tanh and Shifted ReLU, whereas it is significantly different for Erf and Smoothed Relu, which converge to the same limiting distribution along distinct trajectories. In all cases, the solid colored lines go from shallow $L = 2$ networks (red) to deep networks (purple). In all cases but Erf the deepest networks have $L = 128$. For Erf, the dashed lines show solutions to (15) for very large depth up to $L = 8192$.

[1] https://arxiv.org/pdf/1802.09979.pdf

# Outline for today

- Backpropogation and the Pennington spectra
- Glorot (Xavier) initialisation to maintain variance through depth
- Batch normalization as a learned way to control saturation
- Methods for learning the net parameters:
  - Stochastic gradient descent (SGD)
  - Polyak and Nesterov momentum
  - AdaGrad and variants toward Adam

# Optimizing (learning) a deep net

The output of the net is $H(x_\mu; \theta) = \hat{h}_\mu$ and we measure the value of the net through the average sum of squares:

$$\mathcal{L}(\theta; X, Y) = (2m)^{-1} \sum_{\mu=1}^{m} \sum_{i=1}^{n} (\hat{h}_{i,\mu} - y_{i,\mu})^2$$

Central to the success of deep nets is the ability to learn the parameters $\theta$ of the network to achieve good training error while also avoiding overfitting so as to generalize well.

Backpropogation allows an efficient calculation of $\text{grad}_\theta L(\theta; X, Y)$.

# Backpropogation

$$\mathcal{L}(\theta; X, Y) = (2m)^{-1} \sum_{\mu=1}^{m} \sum_{i=1}^{n} (\hat{h}_{i,\mu} - y_{i,\mu})^2$$

Letting $\delta_\ell := \frac{\partial \mathcal{L}}{\partial \hat{h}^{(\ell)}}$ and as before $D^{(\ell)}$ the diagonal matrix with $D^{(\ell)}_{ii} = \phi'(\hat{h}^{(\ell)}_i)$ we have

$$\delta_\ell = D^{\ell}(W^{(\ell)})^T \delta_{\ell+1} \quad \text{and} \quad \delta_L = D^{(L)} \text{grad}_{h^{(L)}} \mathcal{L}.$$

which gives the formula for computing the $\delta_\ell$ for each layer as

$$\delta_\ell = \left( \Pi_{k=\ell}^{L-1} D^{(k)} (W^{(k)})^T \right) D^{(L)} \text{grad}_{h^{(L)}} \mathcal{L}.$$

and the resulting gradient $\text{grad}_\theta \mathcal{L}$ with entries as

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \delta_{\ell+1} \cdot h_\ell^T \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial b^{(\ell)}} = \delta_{\ell+1}$$

# Distribution of Jacobian spectra (Pennington et al. 18'[2])

Recall the Jacobian of the input-output map

$$J = \frac{\partial h^{(L)}}{\partial x^{(0)}} = \Pi_{\ell=1}^{L} D^{(\ell)} W^{(\ell)}$$

and contrast with gradient $\delta_\ell$

$$\delta_\ell = \left( \Pi_{k=\ell}^{L-1} D^{(k)} (W^{(k)})^T \right) D^{(L)} \text{grad}_{h^{(L)}} \mathcal{L}.$$

we see the matrices with the same spectra and limiting distributions.

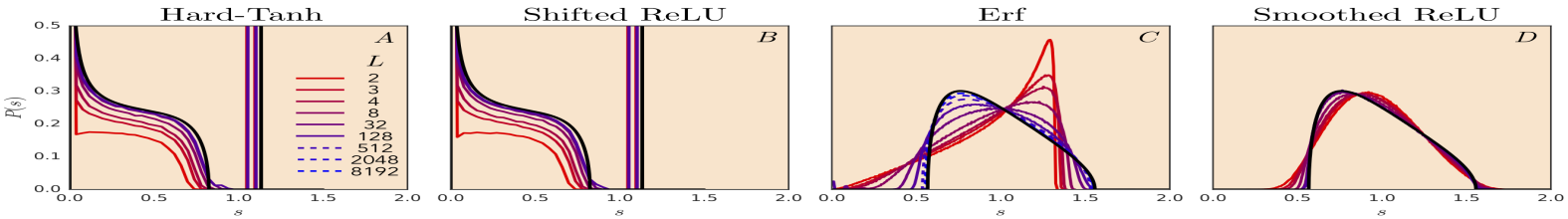



Figure 4: Two limiting universality classes of Jacobian spectra. Hard Tanh and Shifted ReLU fall into one class, characterized by Bernoulli-distributed $\phi'(h)^2$, while Erf and Smoothed ReLU fall into a second class, characterized by a smooth distribution for $\phi'(h)^2$. The black curves are theoretical predictions for the limiting distributions with variance $\sigma_0^2 = 1/4$. The colored lines are emprical spectra of finite-depth width-1000 orthogonal neural networks. The empirical spectra converge to the limiting distributions in all cases. The rate of convergence is similar for Hard-Tanh and Shifted ReLU, whereas it is significantly different for Erf and Smoothed Relu, which converge to the same limiting distribution along distinct trajectories. In all cases, the solid colored lines go from shallow $L = 2$ networks (red) to deep networks (purple). In all cases but Erf the deepest networks have $L = 128$. For Erf, the dashed lines show solutions to (15) for very large depth up to $L = 8192$.

[2] https://arxiv.org/pdf/1802.09979.pdf

# Backpropogation: weight initialization (Glorot et al.' 10[3])

An earlier, more empirical, perspective on the same issue of vanishing/exploding gradients was considered by Xavier Glorot and Yoshua Bengio (2010). They considered the same model assumptions as Pennington, $\hat{h}^{\ell}$ being approximately $\mathcal{N}(0, \sigma_{\ell}^2)$ and:
*"Our objective heres is to understand why standard gradient descent from random initialization is doing so poorly with deep neural networks.... Finally, we study how activations and gradients vary across layers and during training, with the idea that training may be more difficult when the singular values of the Jacobian associated with each layer are far from 1."*

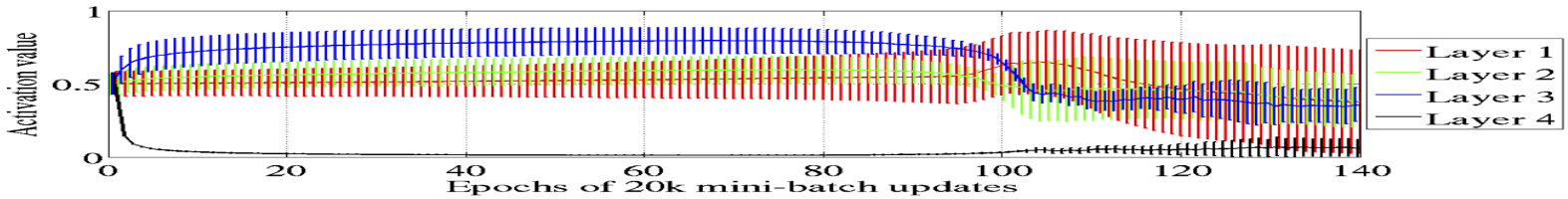



Figure 2: *Mean and standard deviation (vertical bars) of the activation values (output of the sigmoid) during supervised learning, for the different hidden layers of a deep architecture. The top hidden layer quickly saturates at 0 (slowing down all learning), but then slowly desaturates around epoch 100.*

[3] http://proceedings.mlr.press/v9/glorot10a.html

# Backpropogation: weight initialization (Glorot et al.' 10[4])

Glorot initialization follows from seeking the variance of both the backpropogation gradient and forward activations to maintain the same variance per layer: for $W^{(\ell)} \in \mathbb{R}^{n \times n}$ need $\sigma_w^2 = 1/3n$.

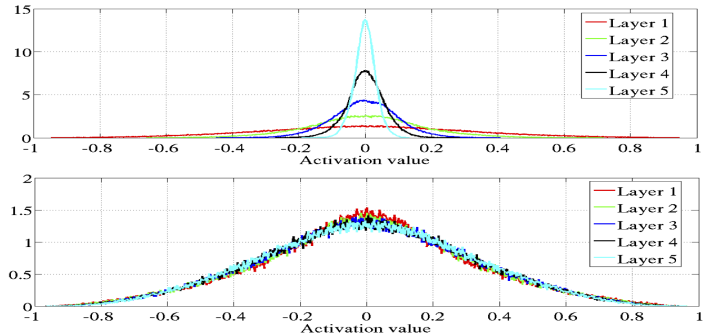



Figure 6: *Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.*

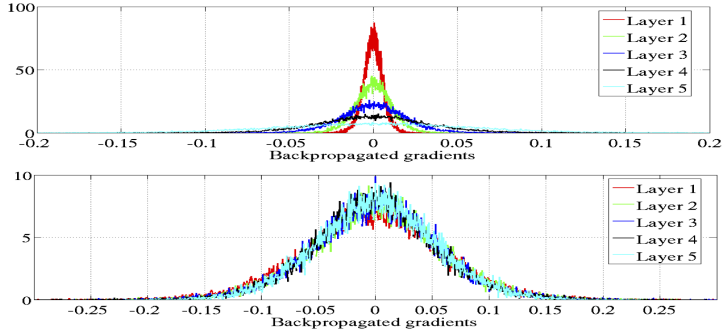



Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

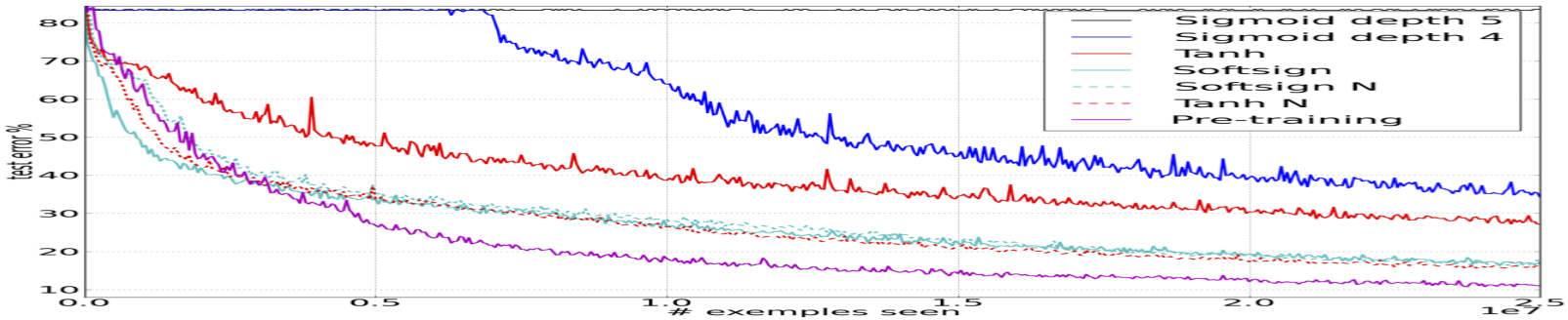



**Figure 11:** *Test error during online training on the* **Shapeset-3 × 2** *dataset, for various activation functions and initialization schemes (ordered from top to bottom in decreasing final error). N after the activation function name indicates the use of normalized initialization.*

[4] http://proceedings.mlr.press/v9/glorot10a.html

# Batch normalization (Ioffe et al. 15'[5])

Saturation in a deep net can alternatively be controlled in a more learned manner by trying to shape the input to a nonlinear activation to be with a learned mean and variance:

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma, \beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.

Typically included after a fully connected layer, before the nonlinear activation. Performed independently per nonlinear activation, with the $\gamma$ and $\beta$ learned as part of the net parameters $\theta$.

[5] https://arxiv.org/pdf/1502.03167.pdf

# Batch normalization experiment (Ioffe et al. 15'[6])

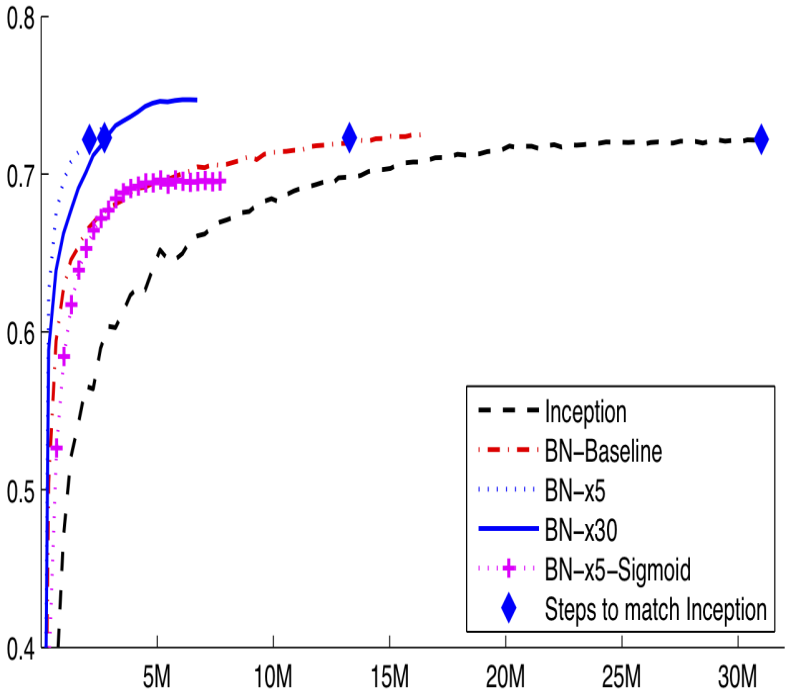



Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

| 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.*

[6] `https://arxiv.org/pdf/1502.03167.pdf`

# Stochastic gradient descent (SGD)

Given a loss function $\mathcal{L}(\theta; X, Y)$, gradient descent is given by

$$\theta^{(k+1)} = \theta^{(k)} - \eta \cdot \mathrm{grad}_\theta \mathcal{L}(\theta, X, Y)$$

where $\eta$ is referred to as the stepsize, or in deep learning the "learning rate."

Recall, we typically have a loss function which is the sum of $n$ individual loss functions, independent for each data point: $\mathcal{L}(\theta; X, Y) = n^{-1} \sum_{\mu=1}^{n} l(\theta; x_\mu, y_\mu)$

For $n \gg 1$ gradient descent is computationally too costly and instead one can break appart the $n$ loss functions into "mini-batches" and repeatedly solve

$$\theta^{(k+1)} = \theta^{(k)} - \eta |\Lambda_k|^{-1} \mathrm{grad}_\theta \sum_{\mu \in \Lambda_k} l(\theta; x_\mu, y_\mu).$$

This is referred to as stochastic gradient descent as typically $\Lambda_k$ is chosen in some randomized method, usually as a partition of $[n]$ and a sequence of $\Lambda_k$ which cover $[n]$ is referred to as an "epoch."

# Stochastic gradient descent: challenges and benefits

$$\theta^{(k+1)} = \theta^{(k)} - \eta|\Lambda_k|^{-1}\text{grad}_\theta \sum_{\mu\in\Lambda_k} l(\theta; x_\mu, y_\mu).$$

- SGD is preferable for large $n$ as it reduces the per iteration computational cost dependence on $n$ to instead depend on $|\Lambda_k|$ which can be set by the user as opposed to $n$ which is given by the data set.
- SGD, and gradient descent, require selection of a learning rate (stepsize) which in deep learning is typically selected using some costly trial and error heuristics.
- The learning rate is typically chosen adaptively in a way that satisfies $\sum_{k=1}^{\infty}\eta_k = \infty$ and $\sum_{k=1}^{\infty}\eta_k^2 < \infty$; in particular as $\eta_k \sim k^{-1}$.
- The optimal selection of learning weight, and selection of $\Lambda$, depends on the unknown local Lipschitz constant $\|\text{grad}l(\theta_1; x_\mu, y_\mu) - \text{grad}l(\theta_2; x_\mu, y_\mu)\| \leq L_\mu\|\theta_1 - \theta_2\|$.

# SGD improvements: momentum

There are many improvements of stochastic gradient descent typically used in practise for deep learning; particularly popular is Polyak momentum:

$$\theta^{(k+1)} = \theta^{(k)} + \beta(\theta^{(k)} - \theta^{(k-1)}) - \alpha \cdot \text{grad}_\theta \mathcal{L}\left(\theta^{(k)}\right)$$

or Nesterov's accelerated gradient:

$$\begin{aligned} \hat{\theta}^k &= \theta^{(k)} + \beta(\theta^{(k)} - \theta^{(k-1)}) \\ \theta^{(k+1)} &= \hat{\theta}^{(k)} - \alpha \cdot \text{grad}_\theta \mathcal{L}\left(\hat{\theta}^{(k)}\right) \end{aligned}$$

These acceleration methods give substantial improvements in the linear convergence rate for convex problems; linear convergence rates are: Normal GD $\frac{\kappa-1}{\kappa+1}$, Polyak $\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$ and NAG $\sqrt{\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}}}$.

# SGD improv. : Adaptive sub-gradients (Duchi et al. 11'[7])

A standard method for improving the convergence rate of line-search methods is preconditioning, Adaptive sub-gradients (AdaGrad) is such a method.
Let $g^{(k)}(\theta^{(k)}) =: \text{grad}_\theta \mathcal{L}(\theta^{(k)})$ be the gradient of the training loss function at iteration $k$, then an efficient method for preconditioning is via a diagonal matrix $B$ with entries

$$B_k(i,i) = \left( \sum_{j=1}^{k} \left( g^{(j)}(\theta^{(j)})(i) \right)^2 \right)^{1/2}$$

which is the diagonal of the square-root of the sum of prior gradient outer-products. AdaGrad is the gradient descent method

$$\theta^{(k+1)} = \theta^{(k)} - \eta |\Lambda_k|^{-1} (B^{(k)} + \epsilon I)^{-1} \text{grad}_\theta \sum_{\mu \in \Lambda_k} l(\theta; x_\mu, y_\mu).$$

$\epsilon I > 0$ added to avoid poor scaling of small values of $B^{(k)}(i,i)$.

[7] http://jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

# AdaGrad improvements: RMSProp and AdaDelta

AdaGrad preconditions with the inverse of
$B_k(i,i) = \left(\sum_{j=1}^{k} \left(g^{(j)}(\theta^{(j)})(i)\right)^2\right)^{1/2}$.
RMSProp (Hinton) gives more weight to the current gradient

$$B_k^{RMS}(i,i) = \gamma \sum_{j=1}^{k-1} \left(g^{(j)}(\theta^{(j)})(i)\right)^2 + (1-\gamma)\left(g^{(k)}(\theta^{(k)})(i)\right)^2$$

for some $\gamma \in [0,1]$ and updates as

$$\theta^{(k+1)} = \theta^{(k)} - \eta|\Lambda_k|^{-1}(B^{(k)} + \epsilon I)^{-1/2}\text{grad}_\theta \sum_{\mu\in\Lambda_k} l(\theta; x_\mu, y_\mu).$$

AdaDelta (Zeiler 12'[8]) extends AdaGrad using a similar preconditioner as $B_k^{RMS}$, but also estimates the stepsize using an average difference in $\theta^{(k)} - \theta^{(k-1)}$.

[8] https://arxiv.org/abs/1212.5701

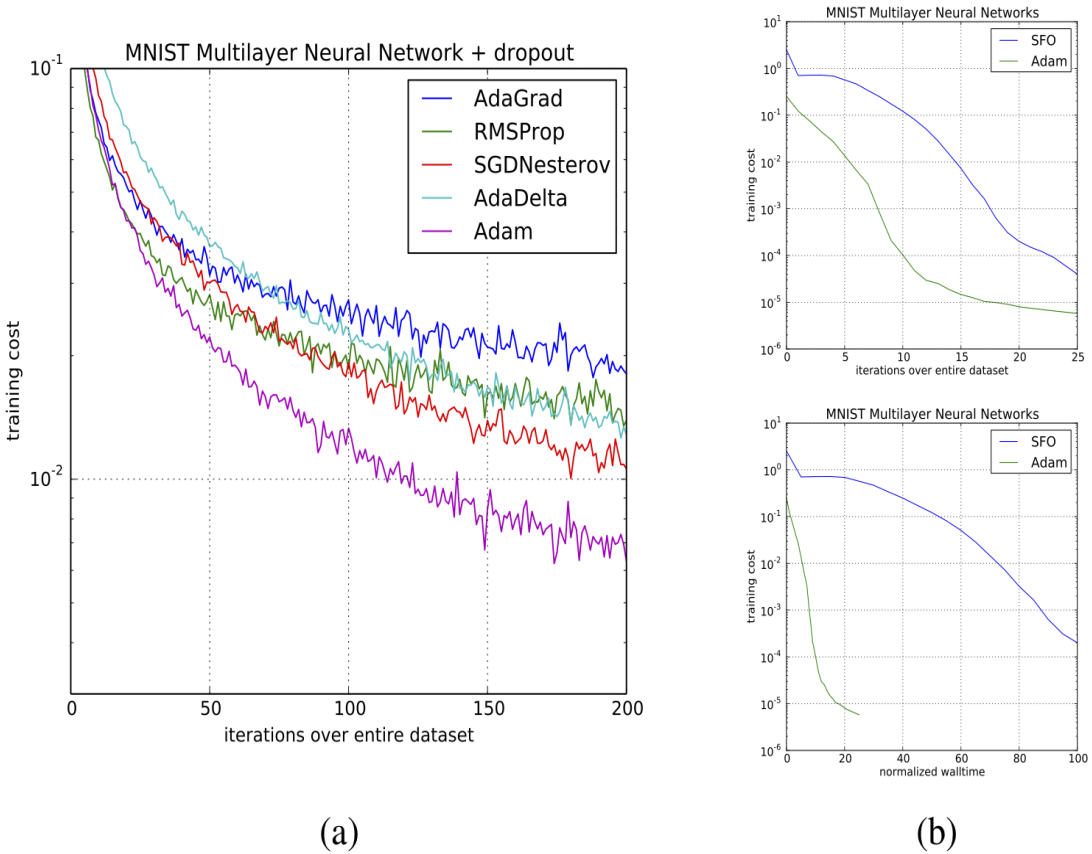
# Adaptive moment estimation (Adam) (Kingma et al. 15'[9])

**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)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  **end while**
  **return** $\theta_t$ (Resulting parameters)

[9] https://arxiv.org/pdf/1412.6980.pdf

# Adaptive moment estimation (Adam) (Kingma et al. 15'[10])




(a) (b)

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)

[10] `https://arxiv.org/pdf/1412.6980.pdf`

# Adaptive moment estimation (Adam) (Kingma et al. 15'[11])

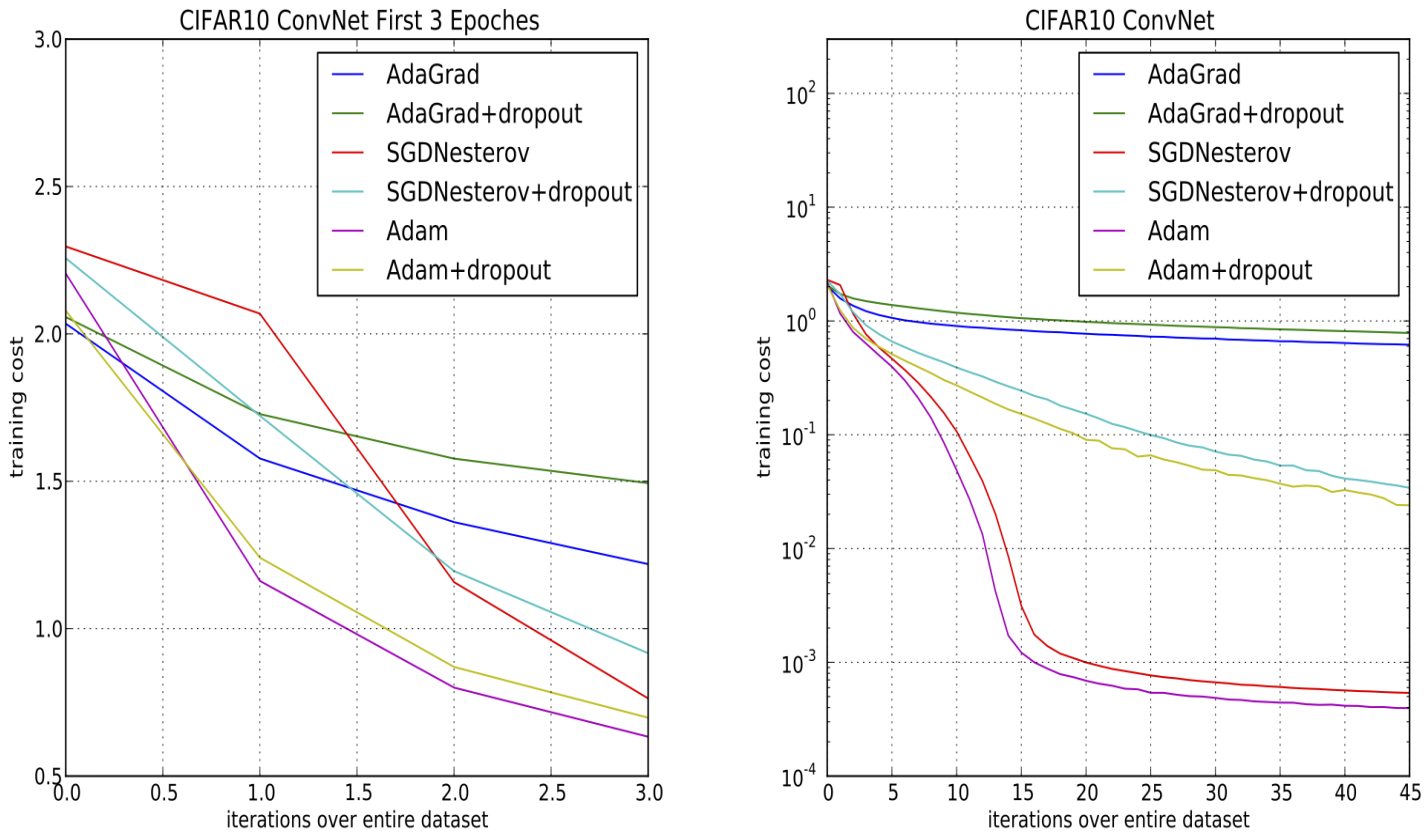



Figure 3: Convolutional neural networks training cost. (left) Training cost for the first three epochs. (right) Training cost over 45 epochs. CIFAR-10 with c64-c64-c128-1000 architecture.

[11] https://arxiv.org/pdf/1412.6980.pdf