Imperial College London

Neural Network Optimisation

Deep Learning Course

Kevin Webster, Pierre Harvey Richemond

Course outline

1. Introduction to deep learning
2. Neural networks optimisation
3. Convolutional neural networks
4. Introduction to reinforcement learning - part 1
5. Introduction to reinforcement learning - part 2
6. Sequence models
7. Generative adversarial networks (GANs)
8. Variational autoencoders (VAEs)
9. Normalising flows
10. Theories of deep learning

Outline

Network optimisation

- Backpropagation

Initialisation

- Vanishing and exploding gradients
- Xavier initialisation
- Orthogonal initialisation

Optimisers

- Gradient descent variants
- Momentum
- Optimisation algorithms

Batch normalization

Outline

Network optimisation

- Backpropagation

Initialisation

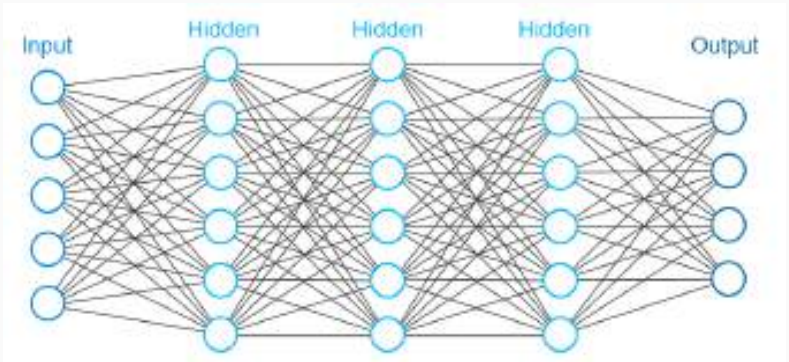
- Vanishing and exploding gradients
- Xavier initialisation
- Orthogonal initialisation

Optimisers

- Gradient descent variants
- Momentum
- Optimisation algorithms

Batch normalization

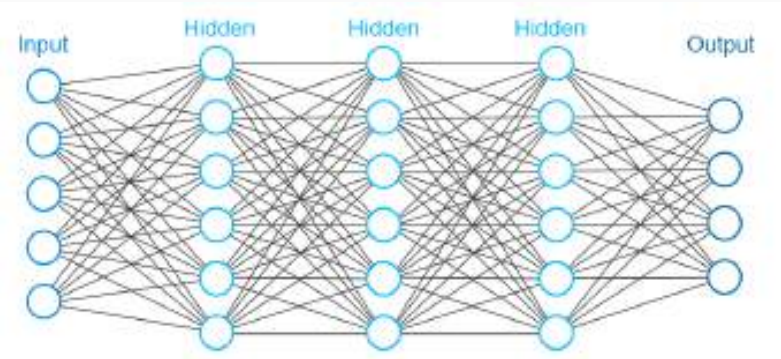
Network optimisation



Recall the typical setting for supervised training of a neural network:

- We have a labelled training set $(\mathbf{x}^{(j)}, y^{(j)})_{j=1}^{m}$
- A neural network is designed to predict y from $\mathbf{x}$, i.e. $y = f(\mathbf{x})$
- The network is typically built in layers $h_1, \ldots, h_N$ $(h_i \in \mathbb{R}^{n_i})$ with input $\mathbf{x} = h_1$ and output $\mathbf{y} = h_N$
- A suitable cost function is chosen $J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{j=1}^{m} L(\mathbf{x}^{(j)}, y^{(j)}, \boldsymbol{\theta})$
- We train the network by minimising the cost function on the training set, while testing it on a separate held-out test set

Backpropagation



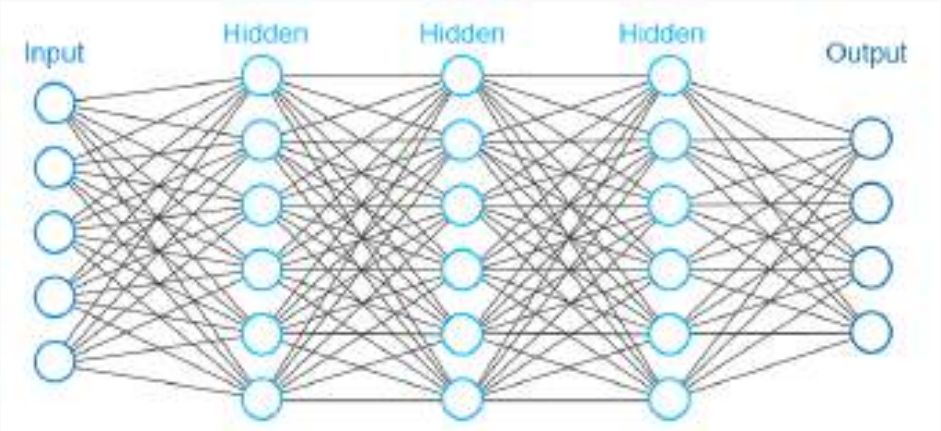
Most algorithms to optimise the weights of a neural network involve computing the gradient

$$\nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(j)}, y^{(j)}, \boldsymbol{\theta}).$$

This calculation is commonly referred to as **backpropagation**.

It involves the use of the chain rule to propagate the gradient of the cost function with respect to the parameters back through the network from the output to the input.

Backpropagation



Neural network with N layers $h_1, \ldots, h_N$, where $h_i \in \mathbb{R}^{n_i}$. Input $\mathbf{x} = h_1$ and output $\mathbf{y} = h_N$. For $i = 1, \ldots, N-1$,

$$\hat{h}_{i+1} = W^{(i)} h_i + b^{(i)}, \qquad \text{(pre-activation)}$$

where $W^{(i)} \in \mathbb{R}^{n_{i+1} \times n_i}$ and $b^{(i)} \in \mathbb{R}^{n_{i+1}}$.

$$h_{i+1} = \sigma(\hat{h}_{i+1}), \qquad \text{(post-activation)}$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is an **activation function** that is applied element-wise.

Backpropagation

We introduce the following notation for the gradient with respect to the pre-activations $\hat{h}_i$:

$$\delta_i := \frac{\partial L}{\partial \hat{h}_i} \in \mathbb{R}^{n_i},$$

and the diagonal matrix of activation function derivates evaluated at $\hat{h}_i$:

$$\Sigma'(\hat{h}_i) := \mathrm{diag}(\sigma'(\hat{h}_i)) \in \mathbb{R}^{n_i \times n_i}$$

Backpropagation

Then it is easy to show that

$$\delta_i = \Sigma'(\hat{h}_i)(W^{(i)})^T \delta_{i+1},$$

and in addition,

$$\delta_N = \Sigma'(\hat{h}_N)\nabla_{h_N=\mathbf{y}} L.$$

Combining the two equations above gives

$$\delta_i = \left(\prod_{k=i}^{N-1} \Sigma'(\hat{h}_k)(W^{(k)})^T \right) \Sigma'(\hat{h}_N)\nabla_{h_N=\mathbf{y}} L.$$

Backpropagation

Finally, the total derivative of the cost function with respect to the parameters of the network is given by

$$\begin{aligned} \frac{\partial L}{\partial W^{(i)}} &= \delta_{i+1} \, . \, h_i^T \\ \frac{\partial L}{\partial b^{(i)}} &= \delta_{i+1} \end{aligned}$$

These equations for backpropagation show how the information is first propagated forward through the network, before the error is calculated and gradients are then propagated backwards through the network and parameter updates are made.

Outline

Network optimisation

- Backpropagation

Initialisation

- Vanishing and exploding gradients
- Xavier initialisation
- Orthogonal initialisation

Optimisers

- Gradient descent variants
- Momentum
- Optimisation algorithms

Batch normalization

Vanishing and exploding gradients

Recall the equation from the backpropagation calculation:

$$\delta_i = \left(\prod_{k=i}^{N-1} \Sigma'(\hat{h}_k)(W^{(k)})^T \right) \Sigma'(\hat{h}_N) \nabla_{h_N = \mathbf{y}} L.$$

highlights one of the main issues with training neural networks; that of **vanishing** or **exploding** gradients.

- Some activation functions (such as sigmoid) can saturate
- Small activations lead to slow-learning weights
- Deep networks can lead to vanishing or exploding gradients if weight matrices have large or small eigenvalues
- This problem is tackled through network design and initialisation strategies

Xavier initialisation

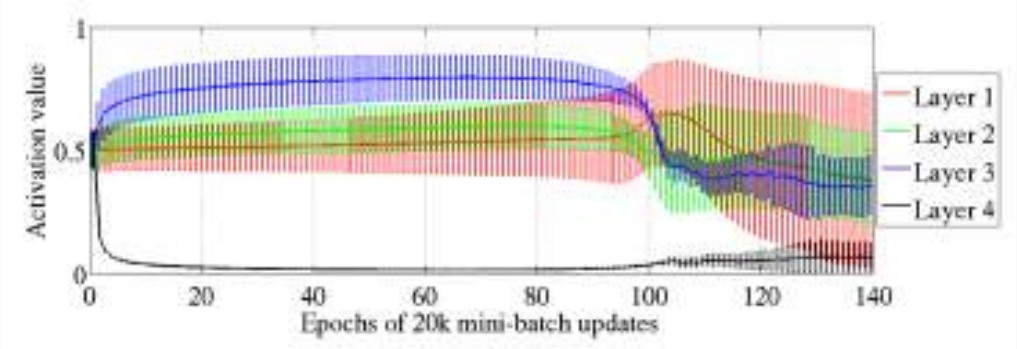
Initialisation of the weights of a neural network is important to ensure the forward and backward signals can propagate through the network during training.

One method that has become popular is commonly referred to as **Xavier** or **Glorot initialisation**.

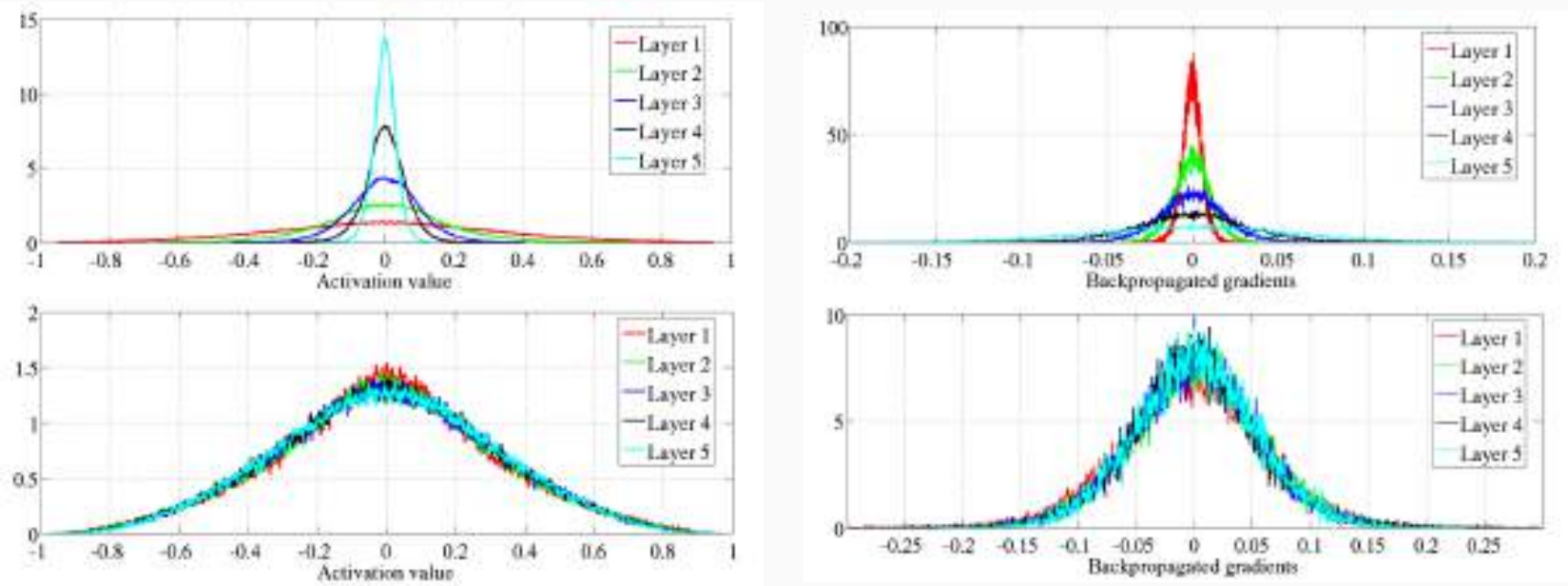
The aim of this initialisation is to specify basic statistical properties of the weight distribution to ensure signal propagation.

- Fix a zero mean for the weight distribution
- We want to find the optimal variance of the weights in each layer

Xavier initialisation



Mean and standard deviation of activation values. Note the quick saturation of the top layer.



Activation & gradient histograms. Top: Unnormalised initialisation. Bottom: Xavier initialisation.

source: [Glorot and Bengio, 2010]

Xavier initialisation

Assume:

- The activations are in the linear region, $\Sigma'(\hat{h}_i) \approx I$.
- The network inputs x_k are zero mean and i.i.d.
- The weight distribution is i.i.d with zero mean for each layer

Then the variance for each unit in the i-th layer h_i is given by

$$\mathrm{Var}[h_i] = \mathrm{Var}[x] \prod_{i'=1}^{i-1} n_{i'} \mathrm{Var}[W^{(i')}],$$

where $\mathrm{Var}[x]$ and $\mathrm{Var}[W^{(i)}]$ denote the shared variances throughout the input layer $\mathbf{x}$ and the weight matrix $W^{(i)}$ respectively.

Xavier initialisation

Also we can show that

$$\mathrm{Var}[\delta_i] = \mathrm{Var}[\nabla_{h_n=\mathbf{y}} L]] \prod_{i'=i}^{N-1} n_{i'+1} \mathrm{Var}[W^{(i')}],$$

where recall $\delta_i := \frac{\partial L}{\partial \hat{h}_i}$.

To preserve the signal through the network in both the forward and backward passes, we want

$$\begin{aligned} \mathrm{Var}[h_i] \approx \mathrm{Var}[h_{i'}] \quad &\Rightarrow \quad n_i \mathrm{Var}[W^{(i)}] = 1 \quad \forall i \\ \mathrm{Var}[\delta_i] \approx \mathrm{Var}[\delta_{i'}] \quad &\Rightarrow \quad n_{i+1} \mathrm{Var}[W^{(i)}] = 1 \quad \forall i \end{aligned}$$

Xavier initialisation

As a compromise between these two constraints, the suggested **Xavier initialisation** scheme is given by

$$\mathrm{Var}[W^{(i)}] = \frac{2}{n_i + n_{i+1}} \quad \forall i = 1, \ldots, N-1.$$

An example weight distribution to use for initialisation with this scheme is:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}\right]$$

Orthogonal initialisation

The repeated application of the weight matrix $W^{(i)}$ in the forward pass and in the equation

$$\delta_i = \left(\prod_{k=i}^{N-1} \Sigma'(\hat{h}_k)(W^{(k)})^T \right) \Sigma'(\hat{h}_N) \nabla_{h_N = \mathbf{y}} L$$

suggests an orthogonal initialisation to prevent the forward and backward signal from vanishing/exploding.

In practice, this can be implemented by randomly initialising a weight matrix $\overline{W}$ and computing the singular value decomposition

$$\overline{W} = U \Sigma V^T$$

and use the matrix V^T to initialise the weights.

Outline

Network optimisation

- Backpropagation

Initialisation

- Vanishing and exploding gradients
- Xavier initialisation
- Orthogonal initialisation

Optimisers

- Gradient descent variants
- Momentum
- Optimisation algorithms

Batch normalization

Gradient descent

Batch gradient descent is the most straightforward variant of gradient descent, where the gradient of the loss function is taken over the entire training set and the parameters adjusted accordingly:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}),$$

where η is the learning rate.

- Very slow, due to the use of the whole dataset
- Intractable for large datasets
- Inefficient use of the data

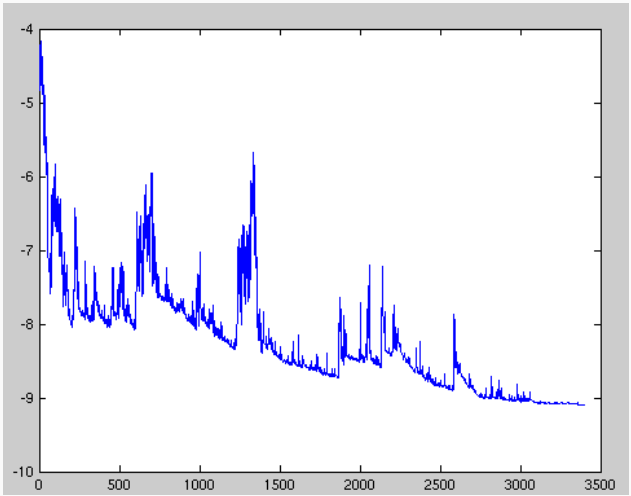
Stochastic gradient descent

Stochastic gradient descent divides the dataset into smaller batches, and computes the gradient for each update on this smaller batch:

$$\begin{aligned} \mathbf{g} &= \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(n_i)}, y^{(n_i)}, \boldsymbol{\theta}), \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta \mathbf{g}, \end{aligned}$$

where again η is the learning rate.

- Reduces redundancy in gradient computation
- Faster (and usually better) convergence
- Can be used online



Some challenges

- Convergence with SGD can be very slow
- Setting the learning rate can be difficult, and often involves trial and error
- Learning rate schedules can be used to reduce the learning rate over the course of training
- Different weights might operate on different scales, and require different rates of learning
- Local minima, and/or saddle points

Several optimisation algorithms have been proposed to help treat these problems.

Momentum

One common tweak to accelerate the slow convergence of the SGD algorithm is to add momentum:

$$\begin{aligned} \mathbf{g}^k &= \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(n_i)}, y^{(n_i)}, \boldsymbol{\theta}^k) \\ \boldsymbol{z}^{k+1} &= \beta \boldsymbol{z}^k + \mathbf{g}^k \\ \boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k - \eta \boldsymbol{z}^{k+1} \end{aligned}$$

Setting $\beta = 0$ recovers SGD.

In practice, typical β values when using momentum is around 0.9.

Convex quadratic example with gradient descent

The effect of momentum can be studied in a simple convex quadratic example:

$$L(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^T A\boldsymbol{\theta} - b^T\boldsymbol{\theta},$$

for symmetric $A \in \mathbb{R}^{p\times p}$ and $b \in \mathbb{R}^p$. The optimal solution is $\boldsymbol{\theta}^* = A^{-1}b$ and the gradient is $\nabla_{\boldsymbol{\theta}} L = A\boldsymbol{\theta} - b$.

The gradient descent update is therefore

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \eta(A\boldsymbol{\theta}^k - b).$$

Convex quadratic example with gradient descent

Let $\lambda_1, \ldots, \lambda_p$ be the eigenvalues of A in ascending order, and $\mathbf{q}_1, \ldots, \mathbf{q}_p$ be the corresponding eigenvectors. Then the gradient descent algorithm can be viewed in closed form:

$$\boldsymbol{\theta}^k - \boldsymbol{\theta}^* = \sum_{i=1}^{p} \varphi_i^0 (1 - \eta\lambda_i)^k \mathbf{q}_i,$$

where $\boldsymbol{\varphi}^k := Q^T(\boldsymbol{\theta}^k - \boldsymbol{\theta}^*)$ and φ_i^0 is the i-th coordinate of $\boldsymbol{\varphi}^0$, with $Q = [\mathbf{q}_1, \ldots, \mathbf{q}_p]$.

Therefore, for convergence, we need $|1 - \eta\lambda_i| < 1$, or $0 < \eta\lambda_i < 2$, for all i. The rate of convergence is determined by $\max\{|1 - \eta\lambda_1|, |1 - \eta\lambda_p|\}$.

$$\begin{aligned} \text{optimal } \eta &= \frac{2}{\lambda_1 + \lambda_p}, \\ \text{optimal rate} &= \frac{\kappa - 1}{\kappa + 1}, \qquad \left(\kappa := \frac{\lambda_p}{\lambda_1}\right). \end{aligned}$$

Convex quadratic example with momentum

The momentum update on the same examples is given by

$$\begin{aligned} \boldsymbol{z}^{k+1} &= \beta \boldsymbol{z}^k + (A\boldsymbol{\theta}^k - b) \\ \boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k - \eta \boldsymbol{z}^{k+1}. \end{aligned}$$

With the change of basis $\boldsymbol{\varphi}^k := Q^T(\boldsymbol{\theta}^k - \boldsymbol{\theta}^*)$ and $\boldsymbol{\omega} := Q^T \mathbf{z}^k$ the updates decouple as

$$\begin{aligned} \omega_i^{k+1} &= \beta \omega_i^k + \lambda_i \varphi_i^k \\ \varphi_i^{k+1} &= \varphi_i^k - \eta \omega_i^k \end{aligned}$$

and we can write

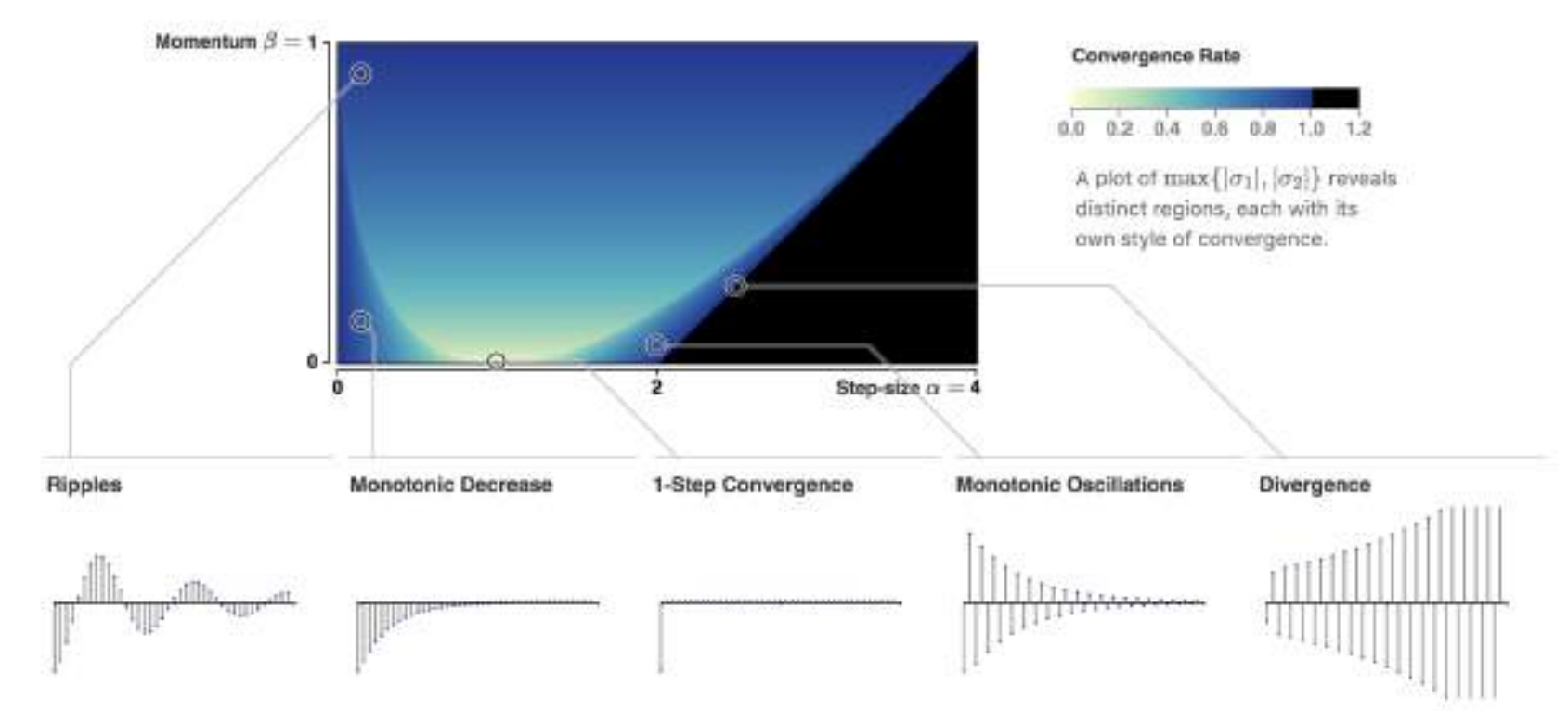
$$\begin{pmatrix} \omega_i^{k+1} \\ \varphi_i^{k+1} \end{pmatrix} = R^k \begin{pmatrix} \omega_i^0 \\ \varphi_i^0 \end{pmatrix}, \qquad R = \begin{pmatrix} \beta & \lambda_i \\ -\eta\beta & 1 - \eta\lambda_i \end{pmatrix}$$

Convex quadratic example with momentum

We can write R^k in terms of its eigenvalues σ_1 and σ_2:

$$R^k = \begin{cases} \sigma_1^k R_1 - \sigma_2^k R_2 & \sigma_1 \neq \sigma_2 \\ \sigma_1^k (kR/\sigma_1 - (k-1)I) & \sigma_1 = \sigma_2 \end{cases}$$

where $R_j := \frac{R - \sigma_j I}{\sigma_1 - \sigma_2}$, $j = 1, 2$. The convergence rate is $\max\{|\sigma_1|, |\sigma_2|\}$.



source: distill.pub/2017/momentum

Convex quadratic example with momentum

For convergence ($\max\{|\sigma_1|, |\sigma_2|\} < 1$) we need

$$0 < \eta\lambda_i < 2 + 2\beta, \qquad \text{for} \quad 0 \leq \beta < 1,$$

so momentum allows us to double the learning rate compared with gradient descent before diverging.

What are the best choices of β and η? Recall the dynamical system

$$\begin{aligned} \omega_i^{k+1} &= \beta\omega_i^k + \lambda_i\varphi_i^k \\ \varphi_i^{k+1} &= \varphi_i^k - \eta\omega_i^k \end{aligned}$$

Convex quadratic example with momentum

For a fixed learning rate η, the critical damping occurs at

$$\beta = (1 - \sqrt{\eta\lambda_i})^2$$

and gives a convergence rate of $1 - \sqrt{\eta\lambda_i}$ (cf. $1 - \eta\lambda_i$ for gradient descent).

Optimising both η and β to get a global convergence rate gives

$$\eta = \left(\frac{2}{\sqrt{\lambda_1} + \sqrt{\lambda_p}}\right)^2, \quad \beta = \left(\frac{\sqrt{\lambda_p} - \sqrt{\lambda_1}}{\sqrt{\lambda_p} + \sqrt{\lambda_1}}\right)^2$$

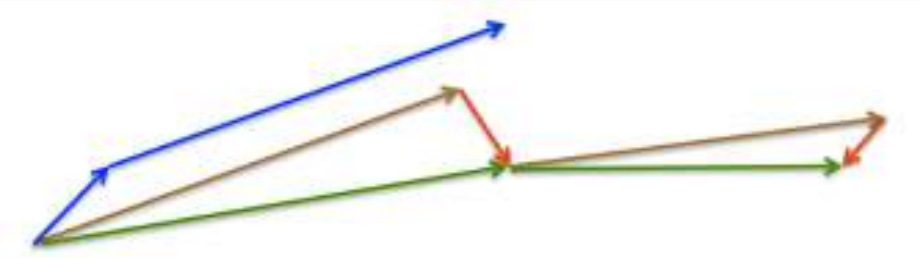
with convergence rate $\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$ (cf. $\frac{\kappa-1}{\kappa+1}$ for gradient descent).

Nesterov momentum

A common variant of momentum is to use Nesterov momentum, which computes the gradient correction after the accumulated gradient, instead of before:

$$\left.\begin{aligned} \mathbf{z}^{k+1} &= \beta \mathbf{z}^k + \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k) \\ \boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k - \eta \mathbf{z}^{k+1} \end{aligned}\right\} \quad \text{(Momentum update)}$$

$$\left.\begin{aligned} \mathbf{z}^{k+1} &= \beta \mathbf{z}^k + \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k - \eta\beta \mathbf{z}^k) \\ \boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k - \eta \mathbf{z}^{k+1} \end{aligned}\right\} \quad \text{(Nesterov momentum)}$$



blue: SGD with momentum

green: SGD with Nesterov momentum

source: G. Hinton's Coursera lectures

Adagrad

- Adapts the learning rate for each parameter
- Less frequent (active) parameters receive larger updates
- Well suited to sparse data
- Used to train GloVe word embeddings

The update rule is (division and square root performed element-wise):

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} - \frac{\eta}{\sqrt{G^{k} + \epsilon}} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^{k}),$$

where $G^k \in \mathbb{R}^{p \times p}$ is a diagonal matrix where the diagonal elements G^k_{ii} are the sum of squares of gradients with respect to θ_i up to time step k.

Note that the resulting learning rates per parameter are monotonically decreasing, and eventually the algorithm effectively stops learning.

RMSProp

- Aims to resolve the vanishing learning rates of Adagrad
- Unpublished method, appears in G. Hinton's Coursera class
- Uses a decaying average of past squared gradients

The update rule is:

$$\begin{aligned} \mathbb{E}[\mathbf{g}^2]^k &= \gamma \mathbb{E}[\mathbf{g}^2]^{k-1} + (1-\gamma)(\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k))^2 \\ \boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k - \frac{\eta}{\sqrt{\mathbb{E}[\mathbf{g}^2]^k + \epsilon}} \odot \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k), \end{aligned}$$

As before, the division and square root are performed element-wise, and $\odot$ is the Hadamard product. The γ term is typically set similar to momentum, around 0.9.

Adadelta

- Independently developed around the same time as RMSProp
- Also aims to resolve the vanishing learning rates of Adagrad
- Removes the need to set a default learning rate

The update rule is:

$$
\begin{aligned}
\mathbb{E}[\mathbf{g}^2]^k &= \gamma \mathbb{E}[\mathbf{g}^2]^{k-1} + (1-\gamma)(\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k))^2 \\
\mathbb{E}[(\Delta\boldsymbol{\theta})^2]^k &= \gamma \mathbb{E}[(\Delta\boldsymbol{\theta})^2]^{k-1} + (1-\gamma)(\boldsymbol{\theta}^k - \boldsymbol{\theta}^{k-1})^2 \\
\boldsymbol{\theta}^{k+1} &= \boldsymbol{\theta}^k - \frac{\sqrt{\mathbb{E}[(\Delta\boldsymbol{\theta})^2]^k + \epsilon}}{\sqrt{\mathbb{E}[\mathbf{g}^2]^k + \epsilon}} \odot \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k),
\end{aligned}
$$

The γ term is typically set similar to momentum, around 0.9.

Adam

- Adaptive moment estimation
- Also computes adaptive learning rates per parameter
- Estimates first and second moments of the gradients

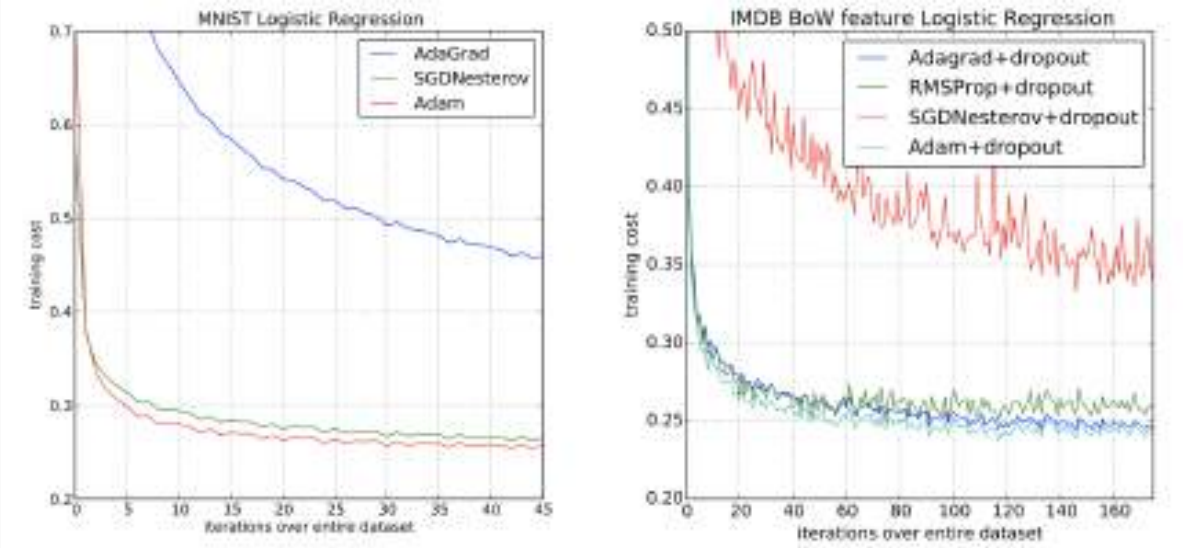
The update rule is:

$$\begin{aligned} \mathbb{E}[\mathbf{g}]^k &= \beta_1 \mathbb{E}[\mathbf{g}]^{k-1} + (1-\beta_1)\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k), & \mathbf{m}^k &= \mathbb{E}[\mathbf{g}]^k/(1-\beta_1) \\ \mathbb{E}[\mathbf{g}^2]^k &= \beta_2 \mathbb{E}[\mathbf{g}^2]^{k-1} + (1-\beta_2)(\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k))^2, & \mathbf{v}^k &= \mathbb{E}[\mathbf{g}^2]^k/(1-\beta_2) \end{aligned}$$

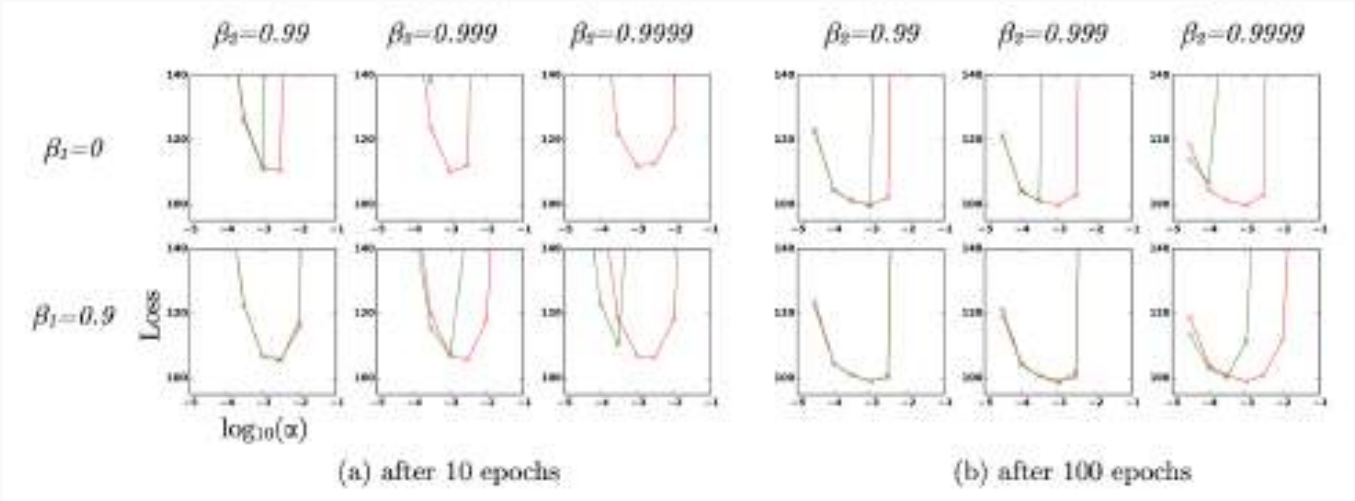
$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \frac{\eta}{\sqrt{\mathbf{v}^k + \epsilon}} \odot \mathbf{m}^k$$

- $\mathbf{m}^k$ and $\mathbf{v}^k$ correct for an initial bias towards zero
- Typical values are $\beta_1 \approx 0.9$ and $\beta_2 \approx 0.999$ and $\epsilon \approx 10^{-8}$.

Adam



Logistic regression model training with different optimisers.



Comparing bias correction (red) to without (green). y-axis is the cost, x-axis is learning rate.

Optimisers summary

- Adagrad introduces adaptive learning rate; best suited for sparse data
- RMSProp resolves the vanishing learning rates by using decaying averages
- Adadelta is similar to RMSProp but does not require a learning rate
- Adam adds bias-correction and momentum
- SGD is still often used and tends to find a good minimiser and generalise well
- Further work on optimisers includes warm restarts, switching and cyclic learning rates - see references

Outline

Network optimisation

- Backpropagation

Initialisation

- Vanishing and exploding gradients
- Xavier initialisation
- Orthogonal initialisation

Optimisers

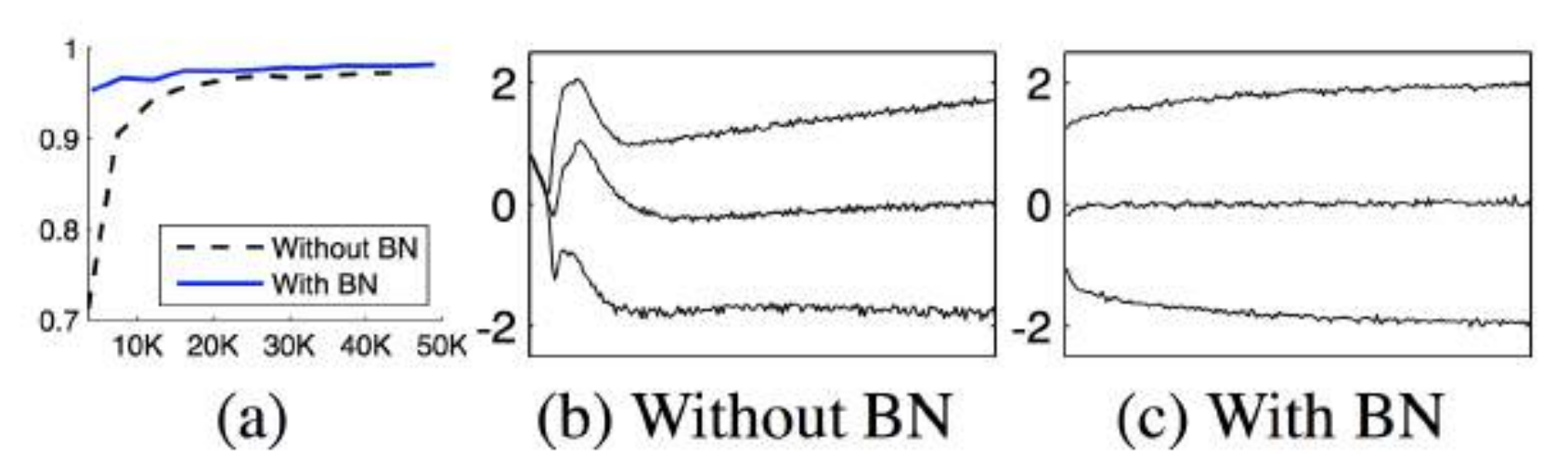
- Gradient descent variants
- Momentum
- Optimisation algorithms

Batch normalization

Batch normalization

- Widely used method in training deep neural networks
- Normalises the distribution of activations throughout the network
- Original paper defines *internal covariate shift* as the change in distribution of a network layer's activations as parameters are changed.
- Ideally we would like to whiten each layer's activations, but this is too expensive
  - Instead, normalize each feature independently in each layer
  - Use minibatch statistics to approximate the statistics of the training set

Batch normalization



(a) Test accuracy on a network trained on MNIST against training iterations

(b, c) Evolution of typical distributions to a sigmoid layer, shown as {15, 50, 85}th percentiles

source: [Ioffe and Szegedy, 2015]

Batch normalization

Note that we must take account of any normalization in the gradient computation:

- Suppose $x = u + b$ for some input u
- Normalize $\hat{x} = x - \mathbb{E}[x]$, with the expectation taken over the training set
- Compute update $\Delta b \propto -\partial L / \partial \hat{x}$
- $b \leftarrow b + \Delta b$
- Then $u + (b + \Delta b) - \mathbb{E}[u + (b + \Delta b)] = u + b - \mathbb{E}[u + b]$
- $\Rightarrow$ Layer output doesn't change
- Normalization needs to be part of the network structure, and activations always follow the desired distribution

Batch normalization

For a layer with d-dimensional input $\mathbf{x} = (x^{(1)}, \ldots, x^{(d)})$, normalize each dimension

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

In order to maintain full expressive power of the network, we make sure the final transformation can represent the identity:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Here, $\gamma^{(k)}$ and $\beta^{(k)}$ are additional learned parameters. Note that setting $\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$ and $\beta^{(k)} = \mathbb{E}[x^{(k)}]$ recovers the original activations.

Batch normalization

Statistics $\mathbb{E}[x^{(k)}]$ and $\mathrm{Var}[x^{(k)}]$ are computed over each minibatch $\mathcal{B} = \{x_1, \ldots, x_m\}$.

$$
\begin{aligned}
\mu_{\mathcal{B}} &= \frac{1}{m}\sum_{i=1}^{m} x_i \\
\sigma^2_{\mathcal{B}} &= \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \\
\hat{x}_i &= \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \\
y_i &= \gamma\hat{x}_i + \beta =: BN_{\gamma,\beta}(x_i)
\end{aligned}
$$

Population statistics are also computed and are used for inference in place of the minibatch statistics.

Batch normalization

For backpropagation, the normalization is taken into account:

$$
\begin{aligned}
\frac{\partial L}{\partial \hat{x}_i} &= \frac{\partial L}{\partial y_i} \cdot \gamma \\
\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} &= \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \sigma_{\mathcal{B}}^2} \\
\frac{\partial L}{\partial \mu_{\mathcal{B}}} &= \left(\sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \mu_{\mathcal{B}}} \right) + \frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mu_{\mathcal{B}}} \\
\frac{\partial L}{\partial x_i} &= \left(\frac{\partial L}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial x_i} \right) + \left(\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\partial \sigma_{\mathcal{B}}^2}{\partial x_i} \right) + \left(\frac{\partial L}{\partial \mu_{\mathcal{B}}} \cdot \frac{\partial \mu_{\mathcal{B}}}{\partial x_i} \right) \\
\frac{\partial L}{\partial \gamma} &= \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} . \hat{x}_i \\
\frac{\partial L}{\partial \beta} &= \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}
\end{aligned}
$$

References i


Duchi, J., Hazan, E., and Singer, Y. (2011).
Adaptive subgradient methods for online learning and stochastic optimization.
J. Mach. Learn. Res., 12:2121–2159.


Glorot, X. and Bengio, Y. (2010).
Understanding the difficulty of training deep feedforward neural networks.
In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*.


Goh, G. (2017).
Why momentum really works.
https://distill.pub/2017/momentum/.

References ii

He, K., Zhang, X., Ren, S., and Sun, J. (2015).
Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.
In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 1026–1034, Washington, DC, USA. IEEE Computer Society.

Ioffe, S. and Szegedy, C. (2015).
Batch normalization: Accelerating deep network training by reducing internal covariate shift.
In Bach, F. R. and Blei, D. M., editors, *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 448–456. JMLR.org.

References iii


Keskar, N. S. and Socher, R. (2017).
Improving generalization performance by switching from adam to SGD.
CoRR, abs/1712.07628.


Kingma, D. P. and Ba, J. (2014).
Adam: A method for stochastic optimization.
CoRR, abs/1412.6980.


Loshchilov, I. and Hutter, F. (2017).
Sgdr: Stochastic gradient descent with warm restarts.
In *International Conference on Learning Representations (ICLR) 2017 Conference Track*.

References iv


Nesterov, Y. (2014).
Introductory Lectures on Convex Optimization: A Basic Course.
Springer Publishing Company, Incorporated, 1 edition.


Ruder, S. (2017).
An overview of gradient descent optimization algorithms.
ArXiv e-prints.


Saxe, A. M., McClelland, J. L., and Ganguli, S. (2013).
Exact solutions to the nonlinear dynamics of learning in deep linear neural networks.
cite arxiv:1312.6120.

References v


Su, W., Boyd, S., and Candes, E. (2015).
A Differential Equation for Modeling Nesterov's Accelerated Gradient Method: Theory and Insights.
ArXiv e-prints.


Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013).
On the importance of initialization and momentum in deep learning.
In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1139–III–1147. JMLR.org.


Zeiler, M. D. (2012).
Adadelta: An adaptive learning rate method.
CoRR, abs/1212.5701.