Imperial College London

Lecture 7: Neural network optimisation

Introduction to machine learning

Kevin Webster

Department of Mathematics
Imperial College London

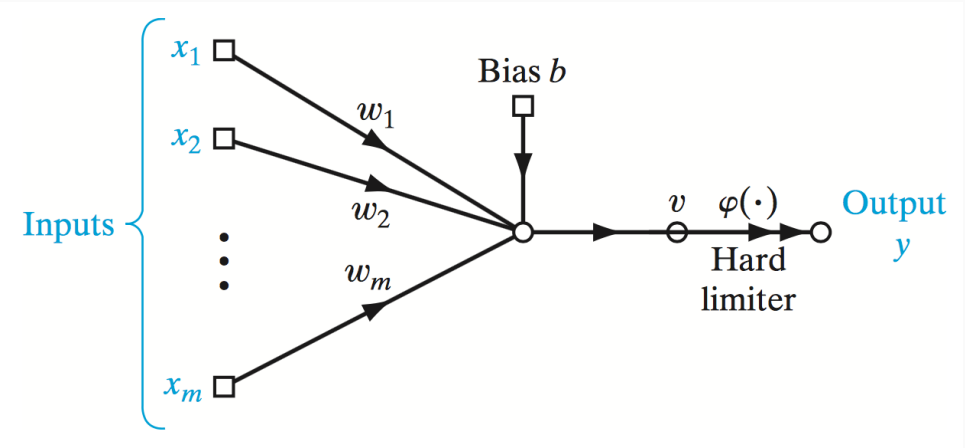
Outline

Outline

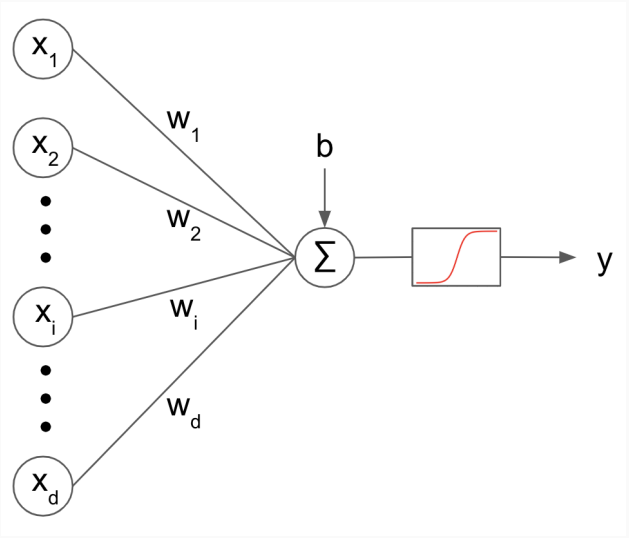
Review: mathematical neuron

We have introduced the concept of a mathematical neuron, and seen examples with the perceptron classifier and logistic regression.

The perceptron

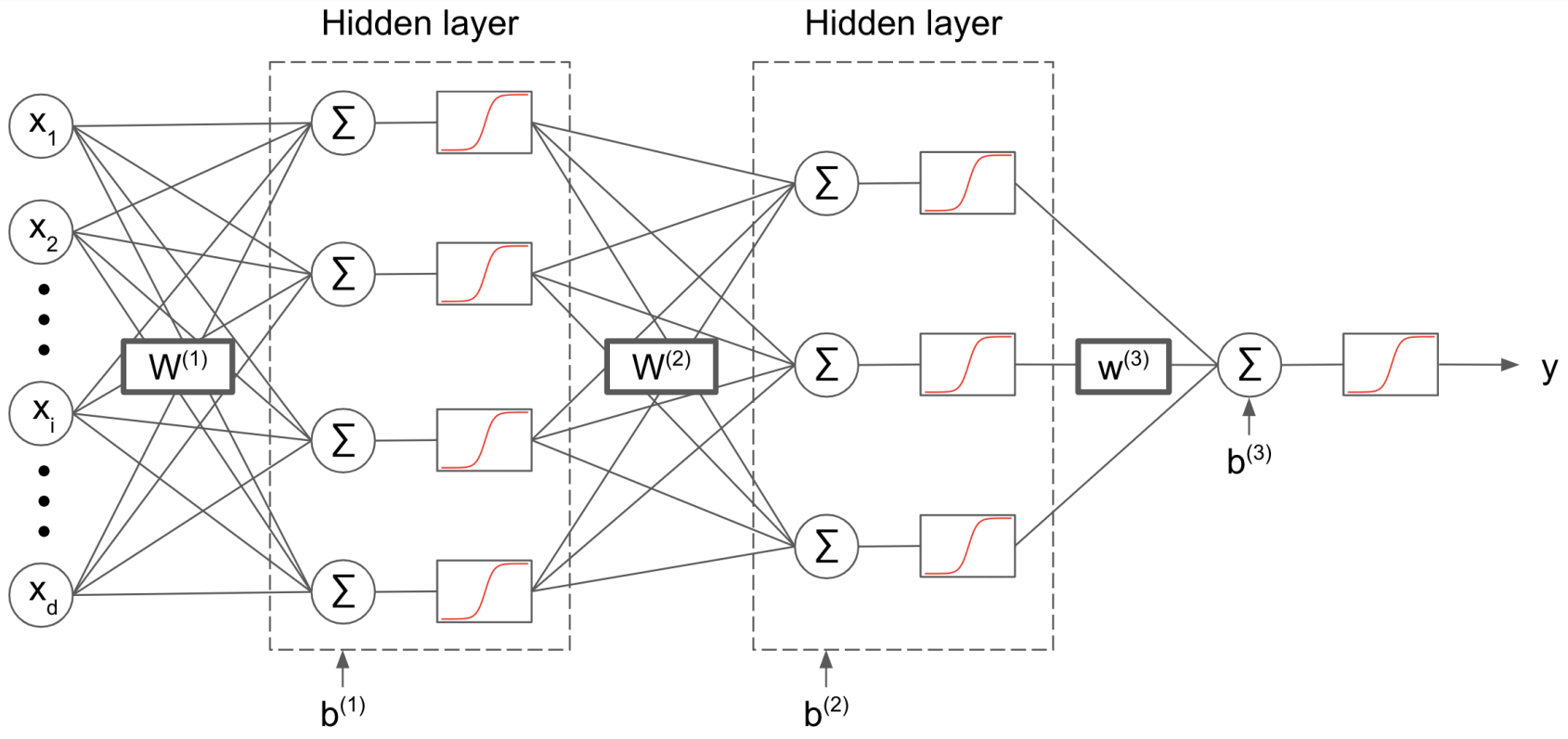


Logistic regression



A key difference between these classifiers is that logistic regression uses a differentiable activation function. That enables the model to be trained using gradient-based methods.

Review: neural network



A neural network can be thought of as a model that composes many neurons together in layers. The parameters of the model are the weight matrices $W^{(j)}$ and biases $b^{(j)}$.

Review: neural network training

We typically use maximum likelihood principles to derive a loss function $E(\boldsymbol{\theta})$, which is a function of the model parameters (we use the notation $\boldsymbol{\theta}$ to represent all weights and biases).

We assume the loss function decomposes as a sum over the training examples:

$$E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} E_i(x_i, y_i, \boldsymbol{\theta})$$

Therefore to apply gradient-based methods for updating the parameters $\boldsymbol{\theta}$, we need to compute $\nabla_{\boldsymbol{\theta}} E_i(x_i, y_i, \boldsymbol{\theta})$. Thus there are two stages:

1. Computation of the derivatives of the loss function with respect to the model parameters (backpropagation)
2. Use the computed gradient to update the parameters

Review: backpropagation algorithm

In the formulation of the backpropagation algorithm, we denote pre-activations in the network as a_j and (post)-activations by z_j.

Note that some of the z_j will be input neurons, and some of the z_j will be output neurons.

The loss function can be seen as a function of the output neurons and the data labels, where the activations of the output neurons have been computed by propagating the inputs through the network in the forward pass.

Review: backpropagation algorithm

$$\text{Forward pass:} \quad a_k = \sum_{j \in \uparrow k} w_{kj} h(a_j) + b_k$$

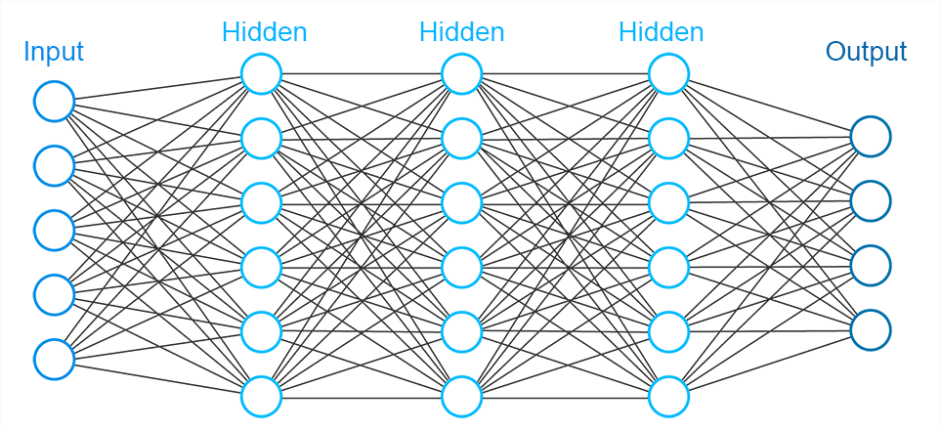
$$\text{Backward pass:} \quad \delta_j = h'(a_j) \sum_{k \in \downarrow j} w_{kj} \delta_k$$

$$\text{Derivatives:} \quad \frac{\partial E_i}{\partial w_{jk}} = \delta_j z_k, \qquad \frac{\partial E_i}{\partial b_j} = \delta_j$$

1. First, propagate the signal forwards by passing an input vector x_i to the network and computing all pre-activations and activations
2. Evaluate $\delta_k = \frac{\partial E_i}{\partial a_k}$ for the output neuron pre-activations
3. Backpropagate the errors δ to compute δ_j for each hidden unit
4. Compute the derivatives using the errors δ_j

Feedforward network / MLP

We can rewrite the backpropagation equations in matrix-vector form for the feedforward network / MLP architecture:



Suppose our network has L layers. Denote the j-th pre/post-activation in the k-th layer as $a_j^{(k)}$ and $z_j^{(k)}$ respectively, and $a^{(k)}, z^{(k)}, \delta^{(k)} \in \mathbb{R}^{n_k}$ are the collections of pre-activations/activations/errors in the k-th layer.

The input layer is $z^{(1)} \equiv x_i$ and output layer is $z^{(L)} \equiv \hat{y}_i$.

Feedforward network / MLP

Denote the weight matrices and biases that map from layer k to $k+1$ as $W^{(k)}$ and $b^{(k)}$ respectively.

The forward and backward pass equations become:

$$\begin{aligned} \text{Forward pass:} \quad a^{(k+1)} &= W^{(k)} z^{(k)} + b^{(k)} \\ z^{(k)} &= \sigma(a^{(k)}) \\ \text{Backward pass:} \quad \delta^{(k)} &= \boldsymbol{\sigma}'(a^{(k)}) \left(W^{(k)}\right)^T \delta^{(k+1)} \end{aligned}$$

where $\boldsymbol{\sigma}'(a^{(k)}) = \text{diag}[\sigma'(a^{(k)})]$, and the derivatives are given by

$$\begin{aligned} \frac{\partial L}{\partial W^{(k)}} &= \delta^{(k+1)} \left(z^{(k)}\right)^T \\ \frac{\partial L}{\partial b^{(k)}} &= \delta^{(k+1)} \end{aligned}$$

Feedforward network / MLP

In the final layer, the error is

$$\delta^{(L)} = \boldsymbol{\sigma}'(a^{(L)}) \frac{\partial E_i}{\partial z^{(L)}}(\hat{y}_i)$$

and so we find that

$$\delta^{(k)} = \left(\prod_{j=k}^{L-1} \boldsymbol{\sigma}'(a^{(j)}) \, (W^{(j)})^T \right) \boldsymbol{\sigma}'(a^{(L)}) \frac{\partial E_i}{\partial z^{(L)}}(\hat{y}_i) \tag{1}$$

Outline

Vanishing and exploding gradients

Equation (1) highlights one of the main issues with training neural networks; that of **vanishing** or **exploding** gradients.

- Some activation functions (such as sigmoid) can saturate, leading to small values in the diagonal matrix $\boldsymbol{\sigma}'(a^{(j)})$
- This can lead to a slow down in the learning of weights
- Deep networks can suffer from vanishing or exploding gradients if weight matrices have large or small eigenvalues
- This problem can be tackled through network design and initialisation strategies

Xavier initialisation

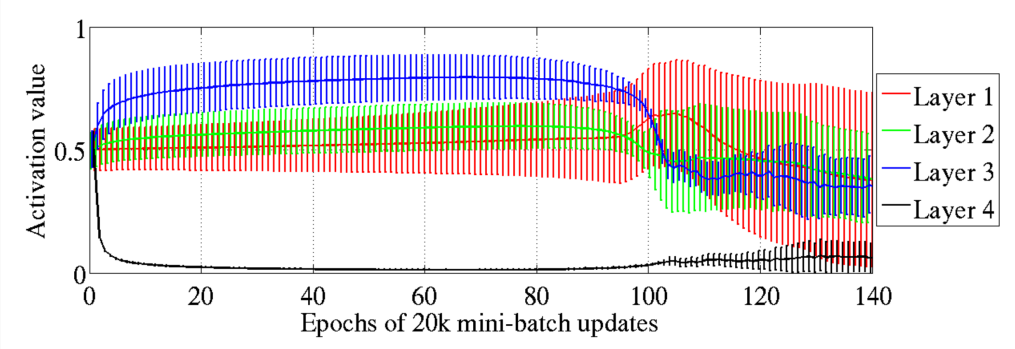
Initialisation of the weights of a neural network is important to ensure the forward and backward signals can propagate through the network during training.

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

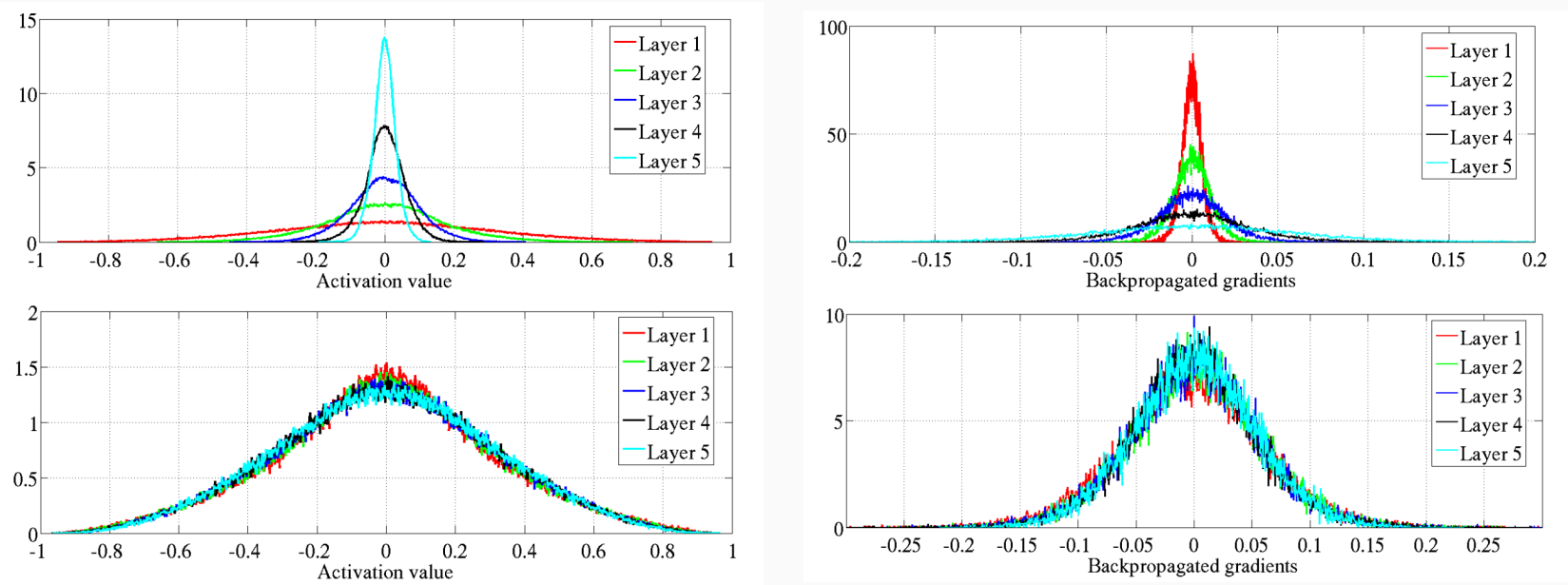
The aim of this initialisation is to specify basic statistical properties of the weight distribution to ensure signal propagation.

- 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: X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", 2010.

Xavier initialisation

Assume:

- The activations are in the linear region, $\sigma'(a_j) \approx I$.
- The network input features $x_i^{(j)}$ 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 k-th layer $z^{(k)}$ is given by

$$\mathrm{Var}[z^{(k)}] = \mathrm{Var}[x_i] \prod_{k'=1}^{k-1} n_{k'} \mathrm{Var}[W^{(k')}],$$

where $\mathrm{Var}[x_i]$ and $\mathrm{Var}[W^{(j)}]$ denote the shared variances throughout the input layer $z^{(1)} = x_i$ and the weight matrix $W^{(j)}$ respectively.

Xavier initialisation

Also we can show that

$$\mathrm{Var}[\delta^{(k)}] = \mathrm{Var}[\nabla_{h_L=y_i} E_i]] \prod_{k'=k}^{L-1} n_{k'+1} \mathrm{Var}[W^{(k')}],$$

where recall $\delta^{(k)} := \frac{\partial E_i}{\partial a^{(k)}}$.

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

$$\begin{aligned} \mathrm{Var}[z^{(k)}] \approx \mathrm{Var}[z^{(k')}] \quad &\Rightarrow \quad n_k \mathrm{Var}[W^{(k)}] = 1 \quad \forall k \\ \mathrm{Var}[\delta^{(k)}] \approx \mathrm{Var}[\delta^{(k')}] \quad &\Rightarrow \quad n_{k+1} \mathrm{Var}[W^{(k)}] = 1 \quad \forall k \end{aligned}$$

Xavier initialisation

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

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

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

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

Outline

Neural network optimisers

Recall the two stages of training neural networks:

1. Computation of the derivatives of the loss function with respect to the model parameters (backpropagation)
2. Use the computed gradient to update the parameters

We have looked at the first stage and the backpropagation algorithm.

We have also seen how vanishing or exploding gradients can occur, and how we can tackle them with good initialisation strategies.

We now turn to the second stage of neural network training, and examine some popular optimisation algorithms.

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:

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

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

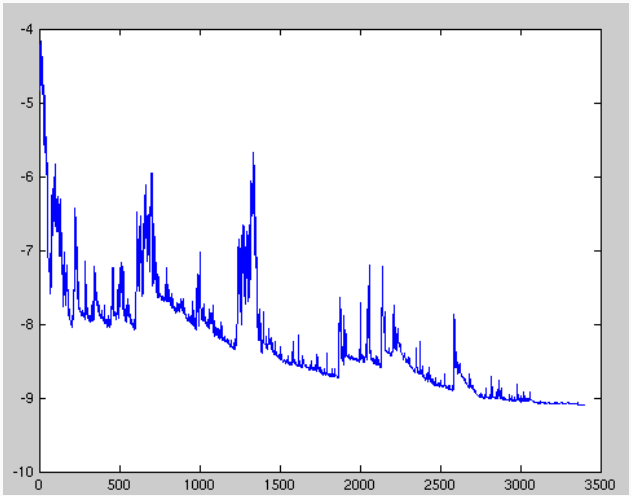
Stochastic gradient descent

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

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

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:

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

Setting $\beta = 0$ recovers SGD.

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

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):

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

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:

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

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:

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

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

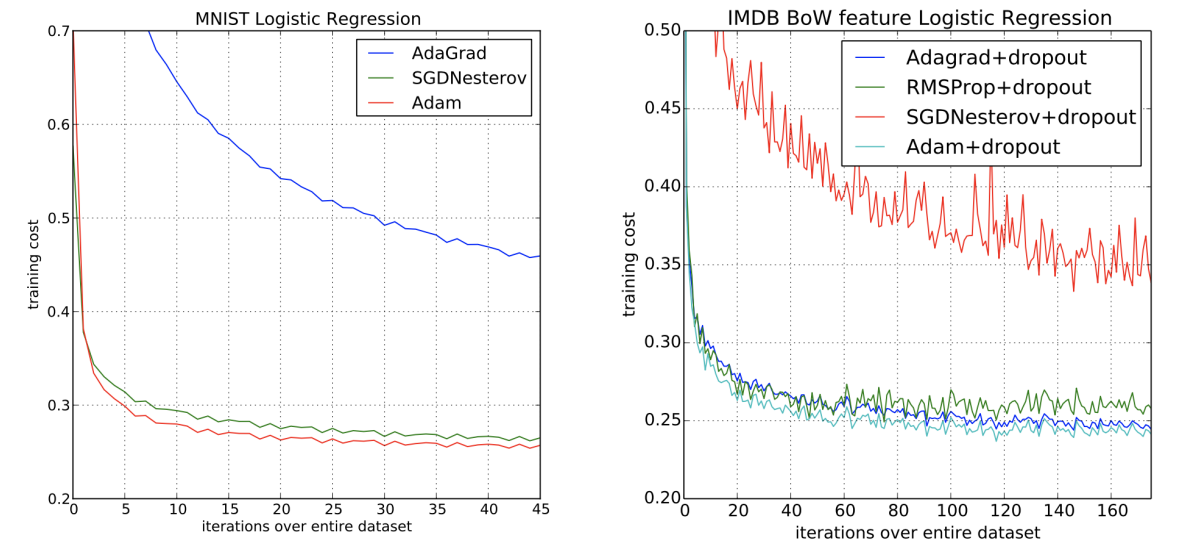
The update rule is:

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

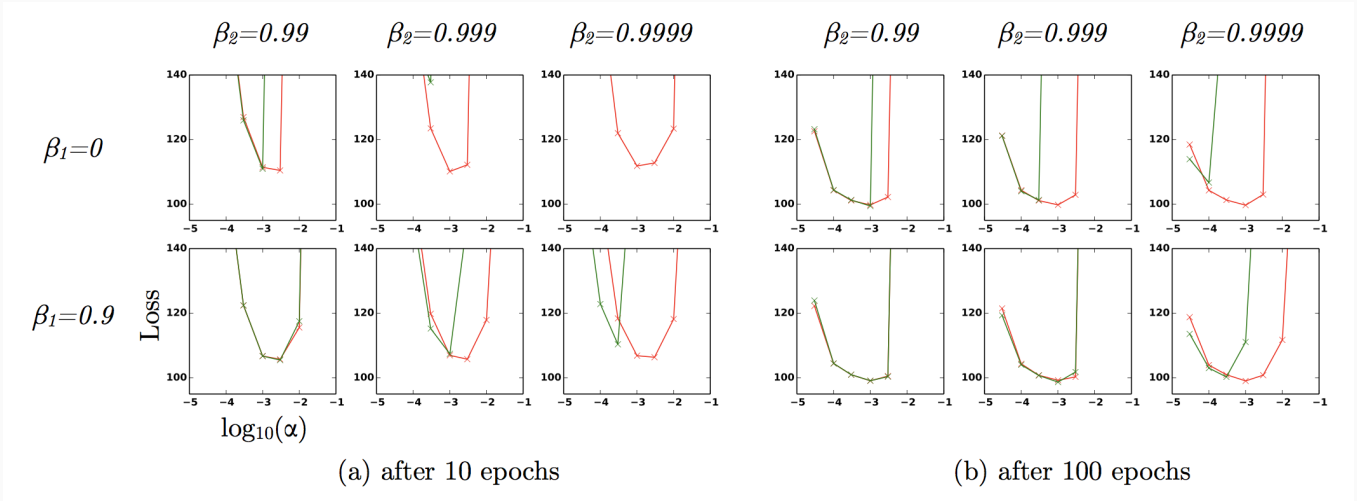
$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \frac{\eta}{\sqrt{\mathbf{v}^k + \epsilon}} \odot \mathbf{m}^k$$

- $\mathbf{m}^k$ 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