Deep Neural Networks (3)
Computational Graphs, Learning Algorithms, Initialisation

Hakan Bilen

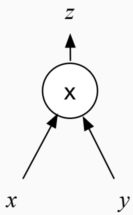
Machine Learning Practical — MLP Lecture 5
15 October 2019

Computational Graphs

Computational graphs

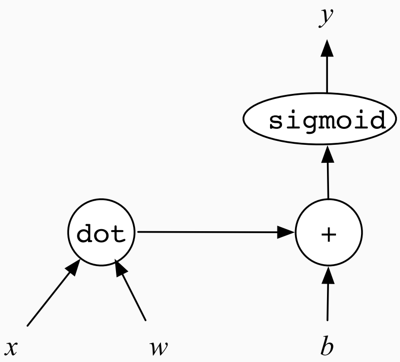
- Each node is an operation
- Data flows between nodes (scalars, vectors, matrices, tensors)
- More complex operations can be formed by composing simpler operations

Computational graph example 1



Graph for $\times$ to compute $z = xy$

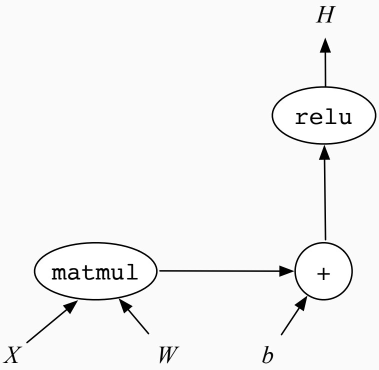
Computational graph example 2



Graph for logistic regression:

$y = \text{sigmoid}(\mathbf{w}^\mathsf{T}\mathbf{x} + b)$

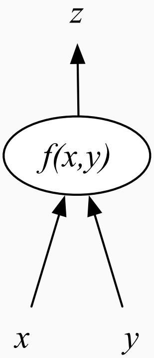
Computational graph example 3



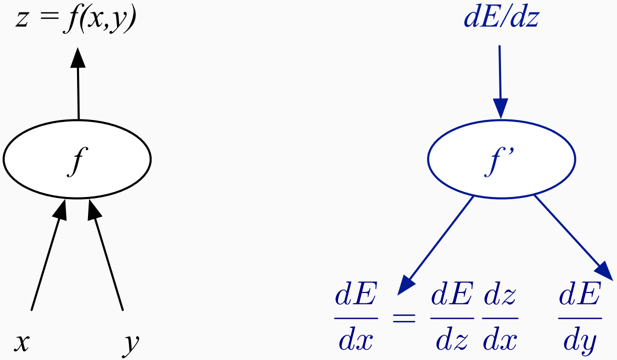
Graph for ReLU layer:

$\mathbf{H} = \text{relu}(\mathbf{WX} + \mathbf{b})$

Computational graphs and back-propagation



Computational graphs and back-propagation



$$\frac{dE}{dx} = \frac{dE}{dz}\frac{dz}{dx} \qquad \frac{dE}{dy} = \frac{dE}{dz}\frac{dz}{dy}$$

Chain rule of differentiation as the backward pass through the computational graph

Computational graphs

- Each node is an operation
- Data flows between nodes (scalars, vectors, matrices, tensors)
- More complex operations can be formed by composing simpler operations
- Implement chain rule of differentiation as a backward pass through the graph
- Back-propagation: Multiply the local gradient of an operation with an incoming gradient (or sum of gradients)
- See `http://colah.github.io/posts/2015-08-Backprop/`

How to set the learning rate?

Weight Updates

- Let $d_i(t) = \partial E / \partial w_i(t)$ be the gradient of the error function E with respect to a weight w_i at update time t
- "Vanilla" gradient descent updates the weight along the negative gradient direction:

$$\Delta w_i(t) = -\eta d_i(t)$$
$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

Hyperparameter η - *learning rate*

- Initialise η, and update as the training progresses (learning rate schedule)

Learning Rate Schedules

- Proofs of convergence for stochastic optimisation rely on a learning rate that reduces through time (as $1/t$) - Robbins and Munro (1951)
- Learning rate schedule – typically initial larger steps followed by smaller steps for fine tuning: Results in *faster convergence* and *better solutions*

Learning Rate Schedules

- Proofs of convergence for stochastic optimisation rely on a learning rate that reduces through time (as $1/t$) - Robbins and Munro (1951)
- Learning rate schedule – typically initial larger steps followed by smaller steps for fine tuning: Results in *faster convergence* and *better solutions*
- **Time-dependent** schedules

$$\Delta w_i(t) = -\eta(t) d_i(t)$$

  - **Piecewise constant**: pre-determined η for each epoch
  - **Exponential**: $\eta(t) = \eta(0) \exp(-t/r)$ ($r \sim$ training set size)
  - **Reciprocal**: $\eta(t) = \eta(0)(1 + t/r)^{-c}$ ($c \sim 1$)

Learning Rate Schedules

- Proofs of convergence for stochastic optimisation rely on a learning rate that reduces through time (as $1/t$) - Robbins and Munro (1951)
- Learning rate schedule – typically initial larger steps followed by smaller steps for fine tuning: Results in *faster convergence* and *better solutions*
- **Time-dependent** schedules

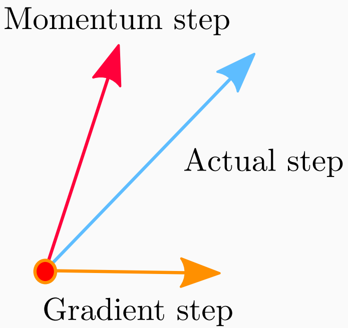
$$\Delta w_i(t) = -\eta(t) d_i(t)$$

  - **Piecewise constant**: pre-determined η for each epoch
  - **Exponential**: $\eta(t) = \eta(0) \exp(-t/r)$ ($r \sim$ training set size)
  - **Reciprocal**: $\eta(t) = \eta(0)(1 + t/r)^{-c}$ ($c \sim 1$)
- **Performance-dependent** η – e.g. "NewBOB": fixed η until validation set stops improving, then halve η each epoch (i.e. constant, then exponential)

Training with Momentum

$$\Delta w_i(t) = -\eta d_i(t) + \alpha \Delta w_i(t-1)$$

- $\alpha \sim 0.9$ is the *momentum* hyperparameter
- Weight changes start by following the gradient
- After a few updates they start to have *velocity* – no longer pure gradient descent
- Momentum term encourages the weight change to go in the previous direction
- Damps the random directions of the gradients, to encourage weight changes in a consistent direction



Adaptive Learning Rates

- Tuning learning rate (and momentum) parameters can be expensive (hyperparameter grid search) – it works, but we can do better
- Adaptive learning rates and per-weight learning rates
  - AdaGrad – normalise the update for each weight
  - RMSProp – AdaGrad forces the learning rate to always decrease, this constraint is relaxed with RMSProp
  - Adam – "RMSProp with momentum"

Well-explained by Andrej Karpathy at
http://cs231n.github.io/neural-networks-3/

AdaGrad

- Separate, normalised update for each weight
- Normalised by the sum squared gradient S

$$
\begin{aligned}
S_i(0) &= 0 \\
S_i(t) &= S_i(t-1) + d_i(t)^2 \\
\Delta w_i(t) &= \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} d_i(t)
\end{aligned}
$$

$\epsilon \sim 10^{-8}$ is a small constant to prevent division by 0 errors

AdaGrad

- Separate, normalised update for each weight
- Normalised by the sum squared gradient S

$$S_i(0) = 0$$

$$S_i(t) = S_i(t-1) + d_i(t)^2$$

$$\Delta w_i(t) = \boxed{\frac{-\eta}{\sqrt{S_i(t)} + \epsilon}} d_i(t)$$

$\epsilon \sim 10^{-8}$ is a small constant to prevent division by 0 errors

- The update step for a parameter w_i is normalised by the (square root of) the sum squared gradients for that parameter
  - Weights with larger gradient magnitudes will have smaller effective learning rates
  - S_i cannot get smaller, so the effective learning rates monotonically decrease
- AdaGrad can decrease the effective learning rate too aggressively in deep networks

AdaGrad

- Separate, normalised update for each weight
- Normalised by the sum squared gradient S

$$S_i(0) = 0$$
$$S_i(t) = S_i(t-1) + d_i(t)^2$$
$$\Delta w_i(t) = \boxed{\frac{-\eta}{\sqrt{S_i(t)} + \epsilon}} d_i(t)$$

  $\epsilon \sim 10^{-8}$ is a small constant to prevent division by 0 errors
- The update step for a parameter w_i is normalised by the (square root of) the sum squared gradients for that parameter
  - Weights with larger gradient magnitudes will have smaller effective learning rates
  - S_i cannot get smaller, so the effective learning rates monotonically decrease
- AdaGrad can decrease the effective learning rate too aggressively in deep networks

Duchi et al, http://jmlr.org/papers/v12/duchi11a.html

RMSProp

- RProp (Riedmiller & Braun, http://dx.doi.org/10.1109/ICNN.1993.298623) is a method for batch gradient descent with an adaptive learning rate for each parameter, and uses only the sign of the gradient (which is equivalent to normalising by the gradient)
- RMSProp can be viewed as a stochastic gradient descent version of RProp normalised by a moving average of the squared gradient (Hinton, http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf) – similar to AdaGrad, but replacing the sum by a moving average for S:

$$S_i(t) = \beta S_i(t-1) + (1-\beta)\, d_i(t)^2$$

$$\Delta w_i(t) = \frac{-\eta}{\sqrt{S_i(t)} + \epsilon}\, d_i(t)$$

$\beta \sim 0.9$ is the decay rate

RMSProp

- RProp (Riedmiller & Braun, http://dx.doi.org/10.1109/ICNN.1993.298623) is a method for batch gradient descent with an adaptive learning rate for each parameter, and uses only the sign of the gradient (which is equivalent to normalising by the gradient)
- RMSProp can be viewed as a stochastic gradient descent version of RProp normalised by a moving average of the squared gradient (Hinton, http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf) – similar to AdaGrad, but replacing the sum by a moving average for S:

$$S_i(t) = \beta S_i(t-1) + (1-\beta)\, d_i(t)^2$$

$$\Delta w_i(t) = \boxed{\frac{-\eta}{\sqrt{S_i(t)} + \epsilon}}\, d_i(t)$$

$\beta \sim 0.9$ is the decay rate

- Effective learning rates no longer guaranteed to decrease

Adam

- Hinton commented about RMSProp: "Momentum does not help as much as it normally does"

Adam

- Hinton commented about RMSProp: "Momentum does not help as much as it normally does"
- Adam (Kingma & Ba, https://arxiv.org/abs/1412.6980) can be viewed as addressing this – it is a variant of RMSProp with momentum:

$$M_i(t) = \alpha M_i(t-1) + (1-\alpha) d_i(t)$$

$$S_i(t) = \beta S_i(t-1) + (1-\beta) d_i(t)^2$$

$$\Delta w_i(t) = \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} M_i(t)$$

Here a momentum-smoothed gradient is used for the update in place of the gradient. Kingma and Ba recommend $\alpha \sim 0.9$, $\beta \sim 0.999$

Adam

- Hinton commented about RMSProp: "Momentum does not help as much as it normally does"
- Adam (Kingma & Ba, https://arxiv.org/abs/1412.6980) can be viewed as addressing this – it is a variant of RMSProp with momentum:

$$M_i(t) = \alpha M_i(t-1) + (1-\alpha) d_i(t)$$

$$S_i(t) = \beta S_i(t-1) + (1-\beta) d_i(t)^2$$

$$\Delta w_i(t) = \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} M_i(t)$$

Here a momentum-smoothed gradient is used for the update in place of the gradient. Kingma and Ba recommend $\alpha \sim 0.9$, $\beta \sim 0.999$

Adam

- Hinton commented about RMSProp: "Momentum does not help as much as it normally does"
- Adam (Kingma & Ba, https://arxiv.org/abs/1412.6980) can be viewed as addressing this – it is a variant of RMSProp with momentum:

$$
\begin{aligned}
\boxed{M_i(t)} &= \alpha M_i(t-1) + (1-\alpha) d_i(t) \\
S_i(t) &= \beta S_i(t-1) + (1-\beta) d_i(t)^2 \\
\Delta w_i(t) &= \boxed{\frac{-\eta}{\sqrt{S_i(t)} + \epsilon}} \boxed{M_i(t)}
\end{aligned}
$$

Here a momentum-smoothed gradient is used for the update in place of the gradient. Kingma and Ba recommend $\alpha \sim 0.9$, $\beta \sim 0.999$

Adam

- Hinton commented about RMSProp: "Momentum does not help as much as it normally does"
-

Many hyperparameters:
batch-size, learning-rate, momentum,
learning-decay-rate, num-layers, num-units, ...

How to set them?

Here a momentum-smoothed gradient is used for the update in place of the gradient. Kingman and Ba recommend $\alpha \sim 0.9$, $\beta \sim 0.999$

Coursework 1

http://www.inf.ed.ac.uk/teaching/courses/mlp/coursework-2019.html

- Build a baseline using the EMNIST dataset
- Implement/compare various activation functions
- Explore different multi-layer network architectures

Coursework 1

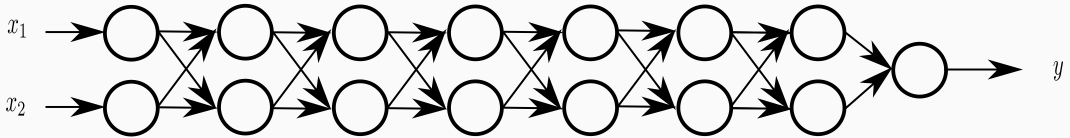
http://www.inf.ed.ac.uk/teaching/courses/mlp/coursework-2019.html

- Build a baseline using the EMNIST dataset
- Implement/compare various activation functions
- Explore different multi-layer network architectures

Main aims of the coursework

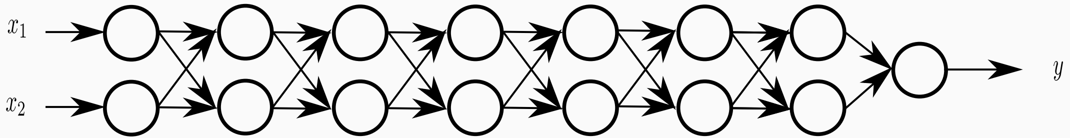
- Implement recent activation functions in Python, carry out experiments to address research questions
- Write a clear, concise, correct report that includes
  - **What** you did
  - **Why** you did it
  - and provides an **interpretation** of your results, and some **conclusions**

Vanishing/exploding gradients



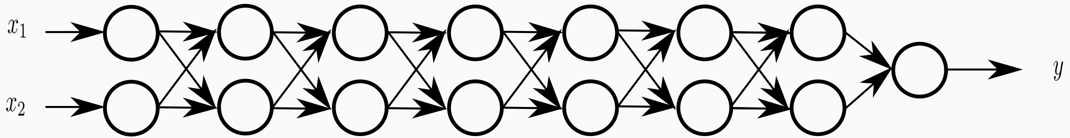
- $z^{(1)} = W^{(1)}x, \quad h^{(1)} = f(z^{(1)})$ and $y = h^{(L)}$
- Assuming f is identity mapping, $y = W^{(L)}W^{(L-1)} \ldots W^{(2)}W^{(1)}x$

Vanishing/exploding gradients



- $z^{(1)} = W^{(1)}x, \quad h^{(1)} = f(z^{(1)})$ and $y = h^{(L)}$
- Assuming f is identity mapping, $\quad y = W^{(L)}W^{(L-1)} \dots W^{(2)}W^{(1)}x$
- $W^{(l)} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \rightarrow \quad y = W^{(L)} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}^{L-2} W^{(1)}x$

Vanishing/exploding gradients



- $z^{(1)} = W^{(1)}x, \quad h^{(1)} = f(z^{(1)})$ and $y = h^{(L)}$
- Assuming f is identity mapping, $\quad y = W^{(L)}W^{(L-1)}\dots W^{(2)}W^{(1)}x$
- $W^{(l)} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \rightarrow \quad y = W^{(L)}\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}^{L-2} W^{(1)}x$
- $W^{(l)} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \quad \rightarrow \quad y = W^{(L)}\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{L-2} W^{(1)}x$

Is it a good idea
to initialize weights with zero?

Summary

- Computational graphs
- Learning rate schedules and gradient descent algorithms
- Initialising the weights
- **Reading**
    - Goodfellow et al, sections 6.5, 8.3, 8.5
    - Olah, "Calculus on Computational Graphs: Backpropagation", http://colah.github.io/posts/2015-08-Backprop/
    - Andrej Karpathy, CS231n notes (Stanford) http://cs231n.github.io/neural-networks-3/
- **Additional Reading**
    - Kingma and Ba, "Adam: A Method for Stochastic Optimization", ICLR-2015 https://arxiv.org/abs/1412.6980
    - Glorot and Bengio, "Understanding the difficulty of training deep feedforward networks", AISTATS-2010.