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:

\[ y = \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b) \]
Graph for ReLU layer:
\[ H = \text{relu}(WX + b) \]
Computational graphs and back-propagation

\[ f(x, y) \]

\[ z \]

\[ f(x, y) \rightarrow z \]

\[ x \]

\[ y \]
Computational graphs and back-propagation

\[ z = f(x, y) \]

\[ \frac{dE}{dz} \]

\[ \frac{dE'}{dx} = \frac{dE}{dz} \frac{dz}{dx} \quad \frac{dE'}{dy} = \frac{dE}{dz} \frac{dz}{dy} \]

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)
How to set the learning rate?
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 $\eta$ - *learning rate*

Initialise $\eta$, and update as the training progresses (learning rate schedule)
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 $\eta$ 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 $\eta$ 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** $\eta$ – e.g. “NewBOB”: fixed $\eta$ until validation set stops improving, then halve $\eta$ 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”

AdaGrad

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

$$S_i(0) = 0$$
$$S_i(t) = S_i(t-1) + d_i(t)^2$$

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

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

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

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

$\epsilon \sim 10^{-8}$ 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 \( S \)

\[
S_i(0) = 0
\]
\[
S_i(t) = S_i(t - 1) + d_i(t)^2
\]

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

\( \epsilon \sim 10^{-8} \) 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
  
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
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.

**Effective learning rates** no longer guaranteed to decrease
Adam commented about RMSProp: “Momentum does not help as much as it normally does”
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:

\[
\begin{align*}
M_i(t) & = \alpha M_i(t-1) + (1 - \alpha)d_i(t) \\
S_i(t) & = \beta S_i(t-1) + (1 - \beta)d_i(t)^2 \\
\Delta w_i(t) & = \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} M_i(t)
\end{align*}
\]

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

\[
\begin{align*}
M_i(t) &= \alpha M_i(t - 1) + (1 - \alpha) d_i(t) \\
S_i(t) &= \beta S_i(t - 1) + (1 - \beta) d_i(t)^2 \\
\Delta w_i(t) &= \frac{-\eta}{\sqrt{S_i(t) + \epsilon}} M_i(t)
\end{align*}
\]

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

\[
M_i(t) = \alpha M_i(t-1) + (1-\alpha) d_i(t)
\]

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

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

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

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

How to set them?
Coursework 1


- Build a baseline using the EMNIST dataset
- Compare RMSProp and Adam with SGD
- Cosine annealing learning rate scheduler
- L2 regularization / weight decay with Adam and cosine annealing
- Inspired by Loshchilov and Hutter, Fixing Weight Decay Regularization in Adam

Main aims of the coursework

- Read and understand a recent paper in the area
- Take the ideas from a paper, implement them 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
How should we initialise deep networks?
Random weight initialisation

- Initialise weights to small random numbers \( r \), sampling weights independently from a Gaussian or from a uniform distribution
  - control the initialisation by setting the mean (typically to 0) and variance of the weight distribution
- Biases may be initialised to 0
  - output (softmax) biases can be normalised to \( \log(p(c)) \), log of prior probability of the corresponding class \( c \)
Random weight initialisation

- Initialise weights to small random numbers $r$, sampling weights independently from a Gaussian or from a uniform distribution
  - control the initialisation by setting the mean (typically to 0) and variance of the weight distribution
- Biases may be initialised to 0
  - output (softmax) biases can be normalised to $\log(p(c))$, log of prior probability of the corresponding class $c$
- Calibration – variance of the input to a unit independent of the number of incoming connections (“fan-in”, $n_{in}$)
- Heuristic: $w_i \sim U(-\sqrt{1/n_{in}}, \sqrt{1/n_{in}})$ [U is uniform distribution]
  - Corresponds to a variance $\text{Var}(w_i) = 1/(3n_{in})$
  - (Since, if $x \sim U(a, b)$, then $\text{Var}(x) = (b-a)^2/12$
    so if $x \sim U(-n, n)$, then $\text{Var}(x) = n^2/3$)
Why $\text{Var}(w) \sim 1/n$?

Consider a linear unit:

$$y = \sum_{i=1}^{n_{in}} w_i x_i$$

if $w$ and $x$ are zero-mean, then

$$\text{Var}(y) = \text{Var}(\sum_{i=1}^{n_{in}} w_i x_i) = n_{in} \text{Var}(x) \text{Var}(w)$$

if $w$ and $x$ are iid (independent and identically distributed)

So, if we want variance of inputs $x$ and outputs $y$ to be the same, set

$$\text{Var}(w_i) = \frac{1}{n_{in}}$$

We would like to constrain the variance of each layer to be $1/n_{in}$, thus

$$w_i \sim U(-\sqrt{3/n_{in}}, \sqrt{3/n_{in}})$$

However we need to take the backprop into account, hence we would also like $\text{Var}(w_i) = 1/n_{out}$

As compromise set the variance to be $\text{Var}(w_i) = 2/(n_{in} + n_{out})$

This corresponds to Glorot and Bengio’s normalised initialisation

$$w_i \sim U\left(-\sqrt{6/(n_{in} + n_{out})}, \sqrt{6/(n_{in} + n_{out})}\right)$$


http://www.jmlr.org/proceedings/papers/v9/glorot10a.html
Summary

- Computational graphs
- Learning rate schedules and gradient descent algorithms
- Initialising the weights
- Reading
  - Goodfellow et al, sections 6.5, 8.3, 8.5
- Additional Reading