Lecture 4: Learning neural network parameters

Arto Klami

21 March, 2025

SGD recap

Last lecture we learnt the stochastic gradient descent (SGD) method for optimization

Initialize parameters somehow

- Evaluate gradient $\nabla J(\boldsymbol{\theta})$ at current $\boldsymbol{\theta}$, based on a subset of observations
- Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mu \nabla J(\boldsymbol{\theta})$
- Stop when happy

From general optimization to learning NNs

Today we talk about how SGD is used to learn parameters of a neural network

We need essentially three things, all of which are somewhat non-trivial:

- How to compute the gradients for large/deep models
- How much to update the parameters: Choice of step-size μ and beyond
- How to do this well: Heuristics and conditions for finding a good local (or global) optimum

We will also discuss how optimizing a NN differs from many other optimization problems

About derivatives

Ways of computing derivatives:

- Analytic derivative: $\nabla x^2 = 2x$ or $\nabla \sin(x) = \cos(x)$
- Chain rule: $\nabla f(g(x)) = g'(x) f(g(x))$ or $\nabla \sin(x^2) = 2x \cos(x^2)$
- Finite difference: $\nabla f(x) \approx \frac{f(x+dx)-f(x)}{dx}$

These are fine for simple univariate functions $f(x)$ but bad or difficult for high-dimensional inputs and/or complex expressions

(Intentional misuse of notation: $\nabla f(x)$ means $\frac{\partial f(x)}{\partial x}$ as these are now scalar functions)

Automatic differentiation

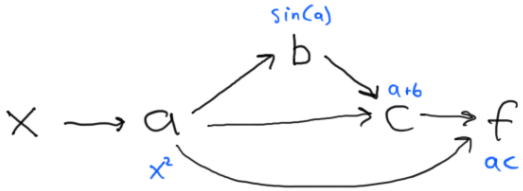
- For NNs we need a new technique, which is typically not covered in math courses: automatic differentiation
- Basic idea: Store intermediate results and partial derivatives during computation of the function, and use those to compute the gradient
- Heavily uses the basic chain rule, but in a different way than you are used to
- Computing the gradient is 'as fast' as evaluating the function itself, and we can do this also over control flow structures (while-loops etc)

See: Prince, Section 7.2-7.3

AD: Example

- Let us start with a simple scalar function $f(x) = (x^2 + \sin(x^2))x^2$
- Give names for intermediate steps:

$$
\begin{aligned}
a &= x^2 \\
b &= \sin(a) \\
c &= a + b \\
f &= ac
\end{aligned}
$$



See: Math exercise 1

AD: Example

- Let us start with very simple scalar function $f(x) = (x^2 + \sin(x^2))x^2$
- Give names for intermediate steps and compute the partial derivatives wrt to inputs for each step

$$
\begin{aligned}
a &= x^2 & \frac{\partial a}{\partial x} &= 2x \\
b &= \sin(a) & \frac{\partial b}{\partial a} &= \cos(a) \\
c &= a + b & \frac{\partial c}{\partial a} &= 1, \quad \frac{\partial c}{\partial b} = 1 \\
f &= ac & \frac{\partial f}{\partial a} &= c, \quad \frac{\partial f}{\partial c} = a
\end{aligned}
$$

Note that $c = a + b$ and $f = ca$ have two inputs so we have two partial derivatives

AD: Example

$f(x) = (x^2 + \sin(x^2))x^2 = ac$ with $a = x^2, b = \sin(a), c = a + b$

- Starting from $\frac{\partial f}{\partial f} = 1$, we can work backwards to form all partial derivatives

$$\begin{aligned}
\frac{\partial f}{\partial c} &= \frac{\partial f}{\partial f}\frac{\partial f}{\partial c} && = 1 \times a \\
\frac{\partial f}{\partial b} &= \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} && = a \times 1 \\
\frac{\partial f}{\partial a} &= \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} + \frac{\partial f}{\partial f}\frac{\partial f}{\partial a} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} && = a + c + a\cos(a) \\
\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} && = (a + c + a\cos(a)) \times 2x
\end{aligned}$$

- The result $\frac{\partial f}{\partial x} = 2x(a + c + a\cos(a))$ is expressed using the intermediate variables

Note that we use a in three computations, so $\frac{\partial f}{\partial a}$ needs to sum over them

AD: Example

For $x = 2$ we have $a = 4$, $b \approx -0.76$, $c \approx 3.24$ and $f \approx 12.97$

- Starting from $\frac{\partial f}{\partial f} = 1$, we can work backwards to form all partial derivatives

$$\begin{aligned}
\frac{\partial f}{\partial c} &= \frac{\partial f}{\partial f}\frac{\partial f}{\partial c} && = 4 \\
\frac{\partial f}{\partial b} &= \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} && = 4 \\
\frac{\partial f}{\partial a} &= \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} + \frac{\partial f}{\partial f}\frac{\partial f}{\partial a} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} && \approx 4.625 \\
\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} && \approx 18.50
\end{aligned}$$

- The result computed directly using the intermediate variables and partial derivatives, analogous to the original computation. We never form the expression!

AD: A bit more formally

The example was a scalar function $f : \mathbb{R} \to \mathbb{R}$ but we already covered cases where an intermediate variable had two (or more) inputs or outputs $\to$ no problem in generalising to multivariate functions

- Forward computation: Start from x and while computing the function store each intermediate result and its partial derivatives
- Backward computation: Start from $\frac{\partial f}{\partial f} = 1$ and compute the previous partial derivatives one by one, summing over the children

Purely mechanical process as long as we know

- The computational graph that gives inputs and outputs for each variable
- The partial derivatives for the individual computational steps

AD: A bit more formally

- This form of AD is called *reverse mode automatic differentiation* and it gives all partial derivatives of $f(x) : \mathbb{R}^D \to \mathbb{R}$
- Loss functions $J(\boldsymbol{\theta})$ are always of this form, since they map the parameter space to a single objective value
- A similar *forward mode AD* algorithm exists to compute all derivatives of a scalar to vector functions, but we mostly do not need it in DL
- Works for general program code if the control flow does not depend on x:

```
while i < y[i]:
        x = x*y[i]
        i += y[i+1]
```

AD in PyTorch

- `optimizer.zero_grad()` sets the gradients to zero
- `function.backward()` computes the gradient
- After `backward()` we can access the gradients (evaluated at the current values) with `param.grad`
- `optimizer.step()` updates the parameters
- Concepts to be aware of: `requires_grad=True`, `torch.no_grad()`, `model.train()` and `model.eval()`

Limitations of AD

- Non-differentiable computational steps: For example, selecting a maximum element of a list would not be okay
- Can replace them with continuous approximations (softmax instead of max)
- Works well for functions that are non-differentiable at isolated points (absolute value, ReLU): The derivative is well defined for most inputs and for the non-differentiable points any sub-derivative is okay
- Documentation tells which functions can be part of the computational graph and which cannot; may depend on the library

AD is NOT mere chain rule

- Even though we repeatedly use the chain rule within the AD algorithm, it is different from direct symbolic differentiation
- Symbolic differentiation of nested expressions very quickly results in complex expressions and often requires repeated re-computation of the same terms
- For the example function we get $f'(x) = (2x + 2x\cos(x^2))x^2 + 2(x^2 + \sin(x^2))x$ which could be further simplified but the simplification is algorithmically harder
- Chain rule over control flow structures?

Backpropagation: AD for NNs

- Unaware of the general AD principle, the backpropagation algorithm was 'invented' in 80s to compute derivatives of feedforward neural networks
- The algorithm is exactly the reverse-mode AD and even presented largely from the same perspective
- Worth going through as an example, even though we now happily rely on AD to compute gradients of much more complex expressions

See: Prince, Section 7.4

Backpropagation: AD for NNs

- Express a MLP as nested sequence of layers:
  $\mathbf{g}(\mathbf{x}) = \psi_K(\boldsymbol{W}_K(\psi_{K-1}(\boldsymbol{W}_{K-1} \dots \psi_1(\boldsymbol{W}_1\mathbf{x})))$
- What we want to compute is the gradient of a loss function $J(\boldsymbol{\theta})$ that compares the model output $\mathbf{y}_K$ against the ground truths $\mathbf{y}$
- For simplicity, we exclude the bias terms here
- We apply AD for computing the gradient of the loss, but computing the loss requires passing the data through the network and hence the whole model is part of the computation graph
- (You could also make various pre-processing steps part of the graph as well – if they have unknown parameters then you would be training also over them, making the pre-processing 'part of the NN')

Forward pass

- Following the general principle we introduce intermediate results and compute their partial derivatives

$$\mathbf{z}_1 = \boldsymbol{W}_1\mathbf{x} \qquad \frac{\partial \mathbf{z}_1}{\partial \boldsymbol{W}_1} = \mathbf{x}^T, \quad \frac{\partial \mathbf{z}_1}{\partial \mathbf{x}} = \boldsymbol{W}_1^T$$

$$\mathbf{y}_1 = \psi_1(\mathbf{z}_1) \qquad \frac{\partial \mathbf{y}_1}{\partial \mathbf{z}_1} = \psi'(\mathbf{z}_1)$$

- Lazy notation: $\frac{\partial \mathbf{y}_1}{\partial \mathbf{z}_1}$ is a vector with derivatives of the activation function for the different neurons (formally this should be a matrix, which is diagonal because the post-activation of one neuron cannot depend on pre-activation of other neurons)
- For subsequent layers we just replace $\mathbf{x}$ with $\mathbf{y}_{k-1}$
- Finally, we compute the loss and its gradient wrt to the model prediction $\mathbf{y}_K$.

Backward pass

- Start from the gradient of the loss, e.g. for MSE: $\boldsymbol{e}_K := \frac{\partial J}{\partial \mathbf{y}_K} = 2(\mathbf{y}_K - \mathbf{y})$
- Proceed backwards in the graph, computing

$$\frac{\partial J}{\partial \boldsymbol{W}_K} = \frac{\partial J}{\partial \mathbf{y}_K}\frac{\partial \mathbf{y}_K}{\partial \mathbf{z}_K}\frac{\partial \mathbf{z}_K}{\partial \boldsymbol{W}_K} = \left(\boldsymbol{e}_K \odot \psi'(\mathbf{z}_K)\right)\mathbf{z}_{K-1}^T$$

$$\frac{\partial J}{\partial \mathbf{y}_{K-1}} = \frac{\partial J}{\partial \mathbf{y}_K}\frac{\partial \mathbf{y}_K}{\partial \mathbf{z}_K}\frac{\partial \mathbf{z}_K}{\partial \mathbf{y}_{K-1}} = \left(\boldsymbol{e}_K \odot \psi'(\mathbf{z}_K)\right)\boldsymbol{W}_K^T =: \boldsymbol{e}_{K-1}$$

- We need the first one directly, and the latter leads to obvious recursion

$$\frac{\partial J}{\partial \boldsymbol{W}_{K-1}} = \frac{\partial J}{\partial \mathbf{y}_{K-1}}\frac{\partial \mathbf{y}_{K-1}}{\partial \mathbf{z}_{K-1}}\frac{\partial \mathbf{z}_{K-1}}{\partial \boldsymbol{W}_{K-1}} = \left(\boldsymbol{e}_{K-1} \odot \psi'(\mathbf{z}_{K-1})\right)\mathbf{z}_{K-2}^T$$

$$\frac{\partial J}{\partial \mathbf{y}_{K-2}} = \frac{\partial J}{\partial \mathbf{y}_{K-1}}\frac{\partial \mathbf{y}_{K-1}}{\partial \mathbf{z}_{K-1}}\frac{\partial \mathbf{z}_{K-1}}{\partial \mathbf{y}_{K-2}} = \left(\boldsymbol{e}_{K-1} \odot \psi'(\mathbf{z}_{K-1})\right)\boldsymbol{W}_{K-1}^T =: \boldsymbol{e}_{K-2}$$

Note: We get also $\frac{\partial J}{\partial \mathbf{x}}$ for free (e.g. for sensitivity analysis or for adversarial examples)

General optimization loop

Initialize parameters somehow

- DONE: Evaluate gradient $\nabla J(\boldsymbol{\theta})$ at current $\boldsymbol{\theta}$, based on a subset of observations
- Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mu \nabla J(\boldsymbol{\theta})$
- Stop when happy

General optimization loop

- Well, almost: Still need to decide which data points to use for computing the gradient
- Commonly, we use SGD so that randomly selected *mini-batch size* of B samples is used, instead of just one data point
- Both forward and backwards passes can be written also for a matrix of inputs, not just a single vector, with linear (but easy-to-parallize) computational cost
- Classical optimization theory would suggest using as large B as we can, to reduce gradient noise
- In practice, small B tend to be better: Allow making more updates in given time, and noise actually helps escaping local optima and generalization

For now use some B samples to estimate the gradients and worry about the choice later

Learning rates

- How about the step length μ?
- We mostly do not use SGD in the conventional sense where the step-lengths decrease with a pre-determined schedule (or even decreasing lengths in general – some methods cyclically increase μ)
- Instead, there are numerous SGD variants that adapt the learning rate during optimization
- Most have separate learning rate for each parameter: $\boldsymbol{\mu} \in \mathbb{R}^D$ instead of μ
- The actual methods are minor variants of each other; we go through one example

Motivation

Momentum and second-order methods:

- Newton's method has an automatic rule for determining the step length via $\mathbf{H}(\boldsymbol{\theta})^{-1}$, but we cannot compute (or invert) $\mathbf{H}$
- Computational cost: Let's use a diagonal matrix $\mathbf{H}$ instead, with fast inverse (element-wise division)
- Obtaining $\mathbf{H}$: Let's not compute the real Hessian but approximate it
- The absolute crudest approximation: Diagonal of $\mathbf{H} = \nabla J(\boldsymbol{\theta}) \nabla J(\boldsymbol{\theta})^T$, which is a vector of second powers of the gradient
- Momentum (averaging gradients over iterations) is also known to help: If gradients do not change we average out the noise, but if they change a lot then we slow down the speed

Adam

Basic idea:

- Compute gradient and denote it by g_t
- Form search direction as running average: $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$
- Form the fake diagonal Hessian: $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
- Update parameters: $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mu \frac{m_t}{\sqrt{v_t}}$

Note: All computations are here element-wise

See: Prince, Section 6.4

Adam

Actual algorithm:

- Compute gradient and denote it by g_t
- Form search direction as running average: $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$
- Form the fake diagonal Hessian: $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
- Fix bias in both estimates: $\hat{m}_t = m_t/(1-\beta_1^t)$ and $\hat{v}_t = v_t/(1-\beta_2^t)$
- Update parameters and safeguard against division by zero: $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mu \frac{m_t}{\sqrt{v_t}+\epsilon}$

Recommended settings: $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

General optimization loop

Initialize parameters somehow

- DONE: Evaluate gradient $\nabla J(\boldsymbol{\theta})$ at current $\boldsymbol{\theta}$, based on a subset of observations
- DONE: Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla J(\boldsymbol{\theta})$
- Stop when happy

Monitoring convergence

Some terminology

- Often the samples are selected without replacement (do not select the same sample again, until we have seen every sample at least once). This is not part of SGD theory, but quite reasonable thing to do
- After N/B iterations we have gone though the data once. This is called *epoch*
- Often people evaluate convergence in terms of epochs, but this is just a convention. E.g. for streaming data it may be better to think in terms of 'how many samples we have seen'

Monitoring convergence

- You should inspect *convergence plots* to see how your optimization works
- Plot the *average* loss per sample, not the sum (this reduces risk of messing up with scaling)
- Often useful to plot logarithm of the loss instead and even use log-scale for the epochs as well
- Not guaranteed to be monotonic, and may start improving again even after the loss seemed to plateau, so avoid making early conclusions
- For large models we may need counter-intuitively many epochs (we could analyse convergence rates, but we skip that on this course)

See: Computer assignment 1

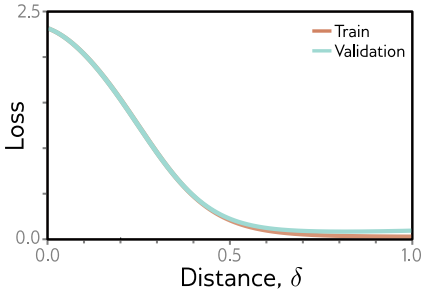
Global vs local convergence

- The loss surface of any reasonable NN is complex and has multiple local optima
- Already because of non-identifiability: We can e.g. permute the order of neurons
- More importantly: We can have multiple global optima that reach zero error but are clearly different outside the training examples
- Formally SGD and its variants find one of those
- Stochasticity helps in escaping poor local optima, as does momentum (Adam etc)

Global vs local convergence

Theoretical analysis of this is quite challenging but empirically SGD is able to optimize (large) NNs extremely well

- Most local minima are good, having almost the same loss function value: `https://arxiv.org/abs/1412.0233`
- Linear interpolation from initial solution to global optimum (sometimes) improves monotonically: `https://arxiv.org/abs/1412.6544`



See: Prince, Section 20.2-20.3

Source: Prince (2023), CC-BY-NC-ND

Global vs local convergence

- Recognizing global optima is in general difficult, but easy in one special case – when we reach the theoretically optimal value: Mean square error of zero, or classification accuracy of 100%, etc.
- Empirically we observe that we can do this in many cases and do not even need stochasticity for escaping local optima – full GD with no stochastic elements at all also finds the global optima for (some) large models
- We may not want the global optimum of the training loss (next lecture), but this should be a conscious decision (not failure of optimizing well enough)
- In practice: Learn to optimize your models so that you can reach the global optimum of zero loss (when sensible), or at least that you are sure you could not optimize it better

See: Computer assignment 1

General optimization loop

Initialize parameters somehow

- DONE: Evaluate gradient $\nabla J(\boldsymbol{\theta})$ at current $\boldsymbol{\theta}$, based on a subset of observations
- DONE: Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mu \nabla J(\boldsymbol{\theta})$
- DONE: Stop when you actually find the global optimum (BUT WE REVISIT THIS WHEN TALKING ABOUT GENERALIZATION)

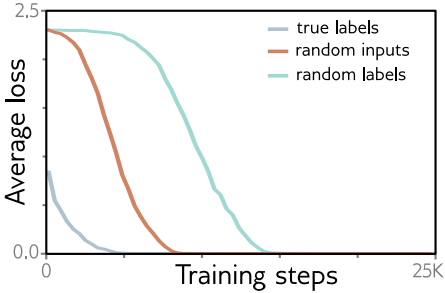
Making the learning easy

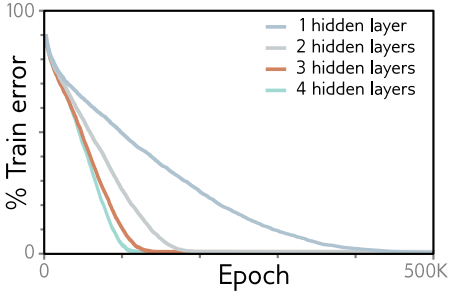
On convergence speed and difficulty:

- Complex dependencies between the model, the data and the algorithm
- Example: Fitting models on random inputs and outputs is slower, but possible
  - Try it out! Randomly permute the outputs or replace the inputs with random values in Computer Assignment 1 and see if it has an effect.
- Smoother surfaces are easier (classical GD convergence rate analysis is in terms of the Lipschitz constant of the objective)
- Various regularization techniques, residual connections etc smooth the loss surface
- Deeper networks tend to be faster to train (but why?)

See: Prince, Chapter 20

On convergence





Source: Prince (2023), CC-BY-NC-ND

Making the learning easy

- Despite the general results on good learnability, it is easy to mess things up
- For instance: SGD completely fails if the gradients are either effectively zero (no learning) or if they have huge magnitude (overshooting with any step length)
- These (vanishing and exploding gradients) used to be a major problem that prevented effective training of deeper neural networks
- Largely resolved today but only if you follow the good practices:
  - Good initialization to ensure informative gradients in the beginning
  - Batch normalization and its variants, to keep them informative (not covered on this course)
  - Residual connections and other architectural tricks (Lecture 7)

Vanishing gradients

- Sigmoid-like activation functions saturate for large inputs: $\psi'(z) = 0$ for $|z| \gg 1$
- NN with K layers of a single neuron: If any $|w_k| \gg 1$, we have either $\psi'(w_k y_{k-1}) = 0$ or $\psi(w_k y_{k-1}) = 0$, both killing the information flow
- ReLU (and variants) do not saturate, but do not resolve the problem alone
- Good initialization helps a lot, and was one of the key elements behind early DL success stories: both Glorot & Bengio (2010) and He et al. (2015) focusing on the initialization have 25k+ citations

Initialization

Simple initialization heuristics:

- Select the scale of initial weights so that the average magnitude of the values remains similar during the forward pass, the backward pass, or (ideally) both
- Ensures the network is trainable in the beginning, and hopefully also later
- Implemented in all libraries as default choices, but you should still learn to control the initialization

See: Mathematical exercise 2

General optimization loop

DONE: Initialize parameters somehow

- DONE: Evaluate gradient $\nabla J(\boldsymbol{\theta})$ at current $\boldsymbol{\theta}$, based on a subset of observations
- DONE: Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mu \nabla J(\boldsymbol{\theta})$
- DONE: Stop when you actually find the global optimum (BUT WE REVISIT THIS WHEN TALKING ABOUT GENERALIZATION)

Deep learning frameworks

The right way to think about DL frameworks is that they provide

- Automatic differentiation engine that gives derivatives for almost anything
- Highly polished optimization libraries with good enough heuristics
- Efficient computation (e.g. GPU computation)

You can (and should) use these also for other things besides deep learning!

In addition, the DL frameworks have very helpful practical tools

- Syntax for defining models, with lots of pre-implemented layers etc
- Data access, convergence monitoring etc
- Good tutorials and reference implementations of specific complex models

Use them, but you need to learn by yourself

Hyperparameters part 1

- For a given model, the optimization quality and speed depend on B, initialization, and the optimizer parameters
- Eventually we will want to select these *hyperparameters* so that we solve the problem well
- Convention: B often chosen amongst powers of two. Often in 'tens'.
- Need to try broad enough range of step lengths μ, so again try powers of two/ten/whatever (not a linear range)
- Roughly linear scaling between the two: If you multiply B by k then you can also multiply μ with k (but may not need to do it for the adaptive methods)
- But: The ratio will influence the kind of local minima we find, so we cannot make optimal choices based on training loss alone

Summary

- SGD and variants as the go-to tool for DL optimization
- Automatic differentiation gives all partial derivatives almost for free
- Do not implement your own optimizers (outside learning exercises); you will miss many details
- For large overparameterized models we can often find a global optimum with zero training error
- The difficulty of the learning problem depends on a lot of things: Initialization, model depth, the data, the architecture, ...
- Learn to recognize when you are optimising well and when not!