Introduction to Machine Learning

Part II

Judith Katzy
Hamburg, September 2024

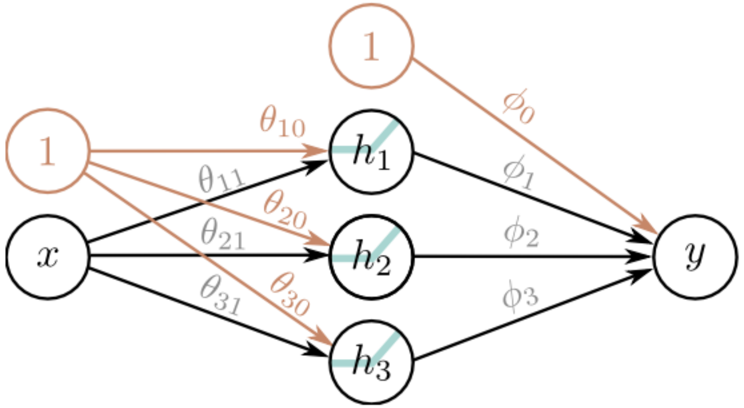
HELMHOLTZ

Outline

- Optimisation
- Stochastic learning
- Loss & Regularisation

Neural network family of functions

$$
\begin{aligned}
y &= \mathrm{f}[x, \boldsymbol{\phi}] \\
&= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x]
\end{aligned}
$$



input

hidden layer with nodes h_i

output

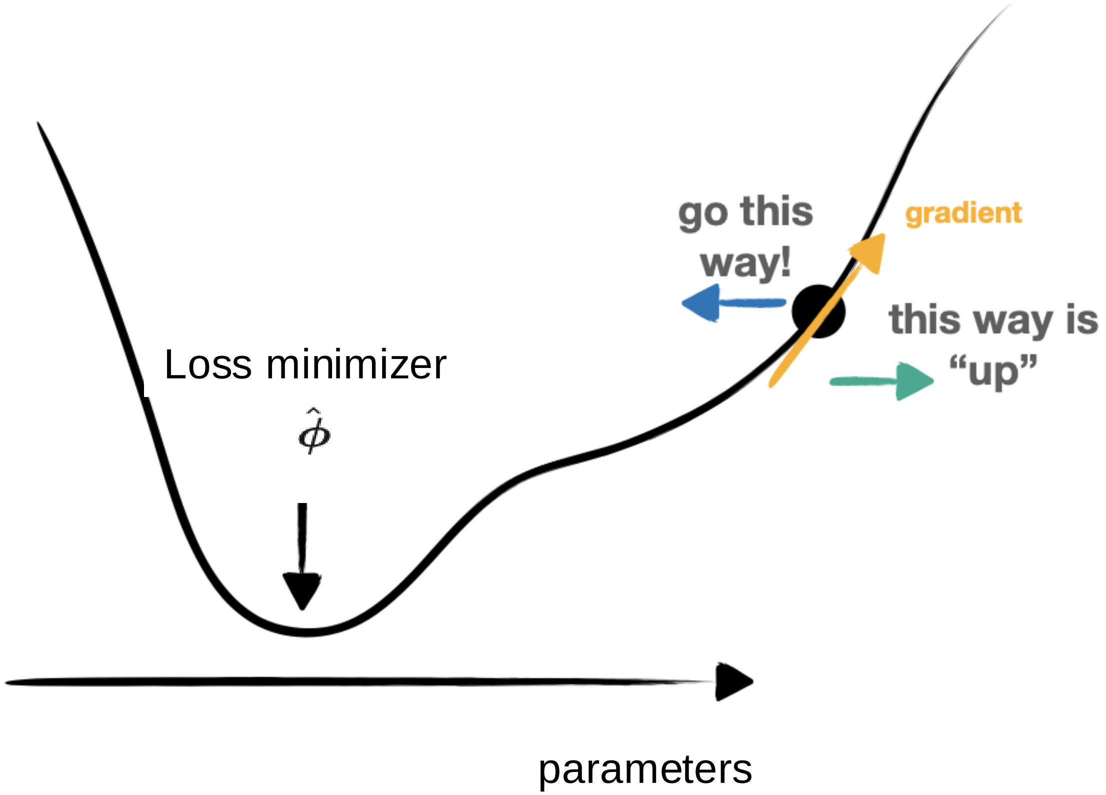
$$
\begin{aligned}
\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \Big[L[\boldsymbol{\phi}] \Big] \\
&= \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} (\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i)^2 \right]
\end{aligned}
$$

$$
\begin{aligned}
h_1 &= \mathrm{a}[\theta_{10} + \theta_{11} x] \\
h_2 &= \mathrm{a}[\theta_{20} + \theta_{21} x] \\
h_3 &= \mathrm{a}[\theta_{30} + \theta_{31} x],
\end{aligned}
$$

Iterative Optimisation

Gradient descent

A natural idea is to optimize by walking downhill

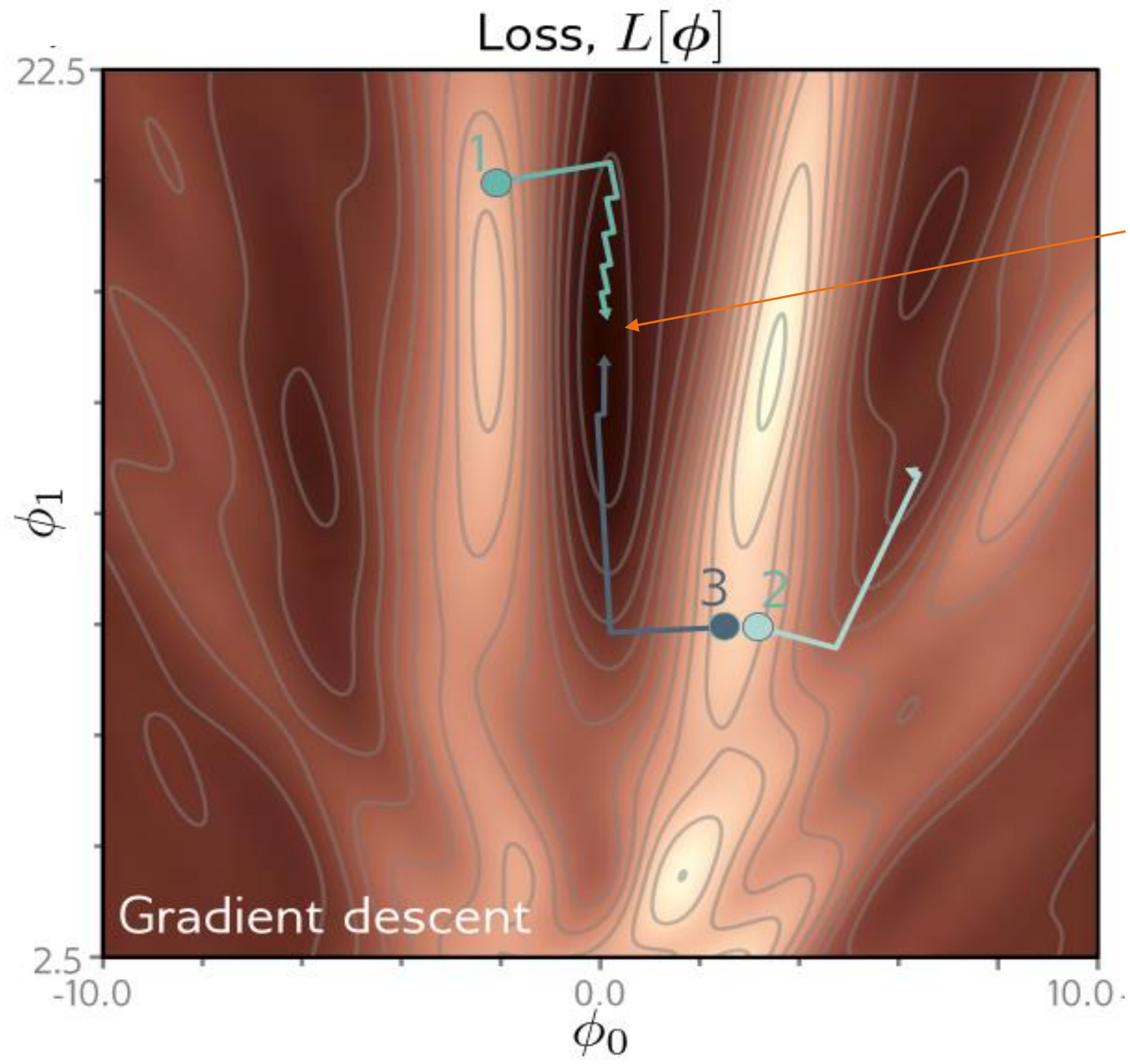


$$\phi \longleftarrow \phi - \alpha \cdot \frac{\partial L}{\partial \phi},$$

alpha: learning rate

$$\hat{\phi} = \underset{\phi}{\mathrm{argmin}} \Big[L[\phi] \Big]$$

Guaranteed to work for convex functions

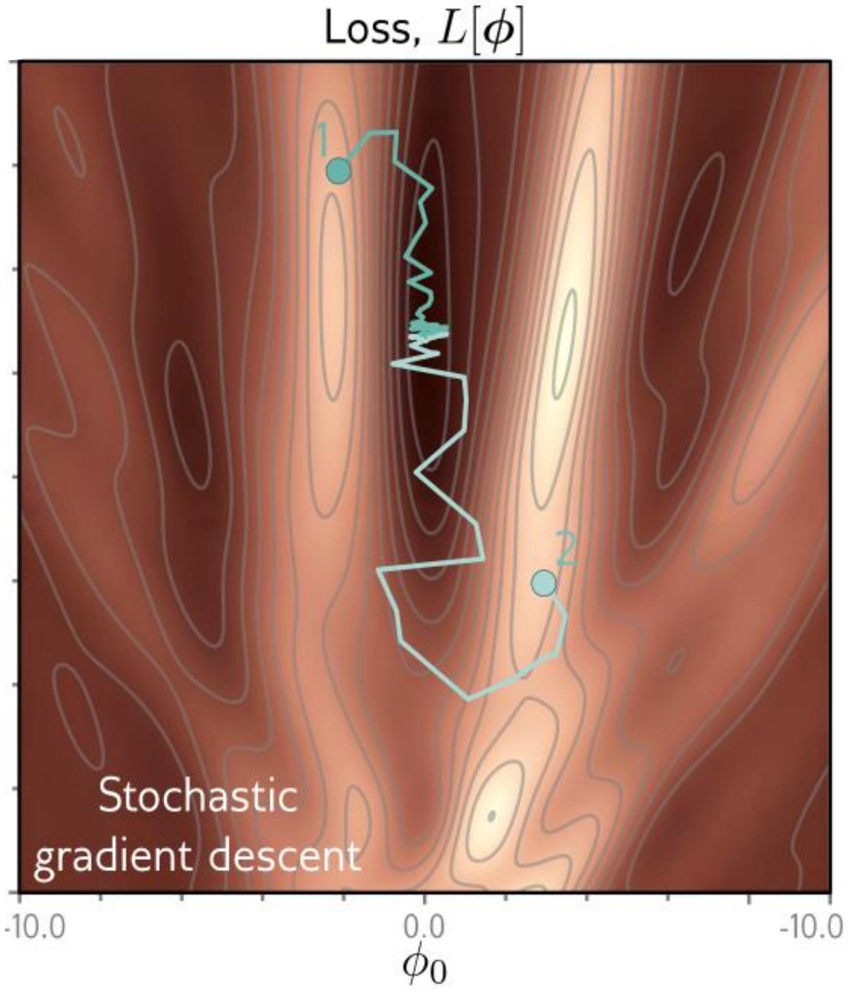


Global minimum

Stochastic gradient descent

Remember: goal is generalization not training loss

Evaluating the loss on a small “mini-batch” instead of the full data: useful noise to jump over e.g. local minima.

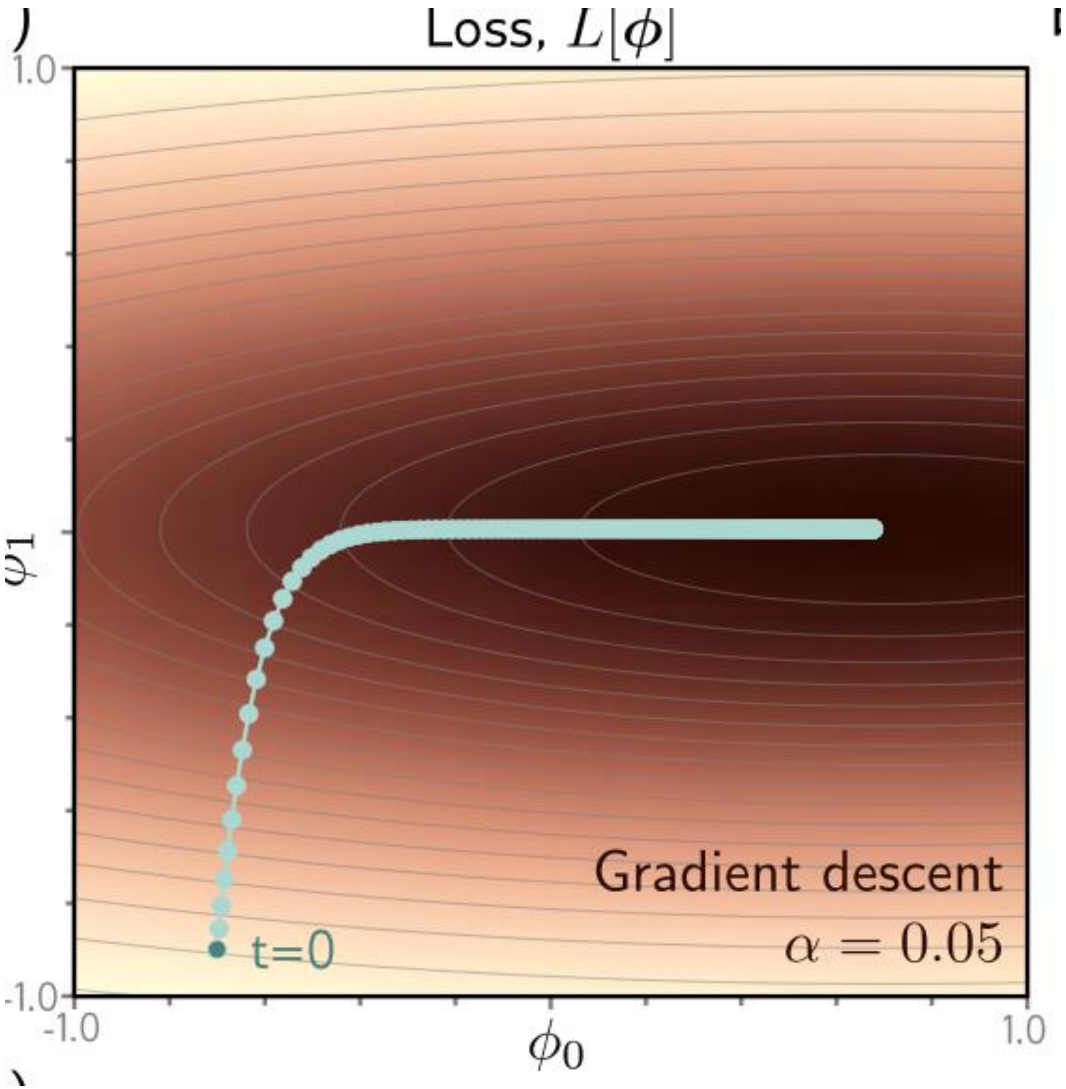


Update loss on (mini)batch

$$\phi_{t+1} \longleftarrow \phi_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

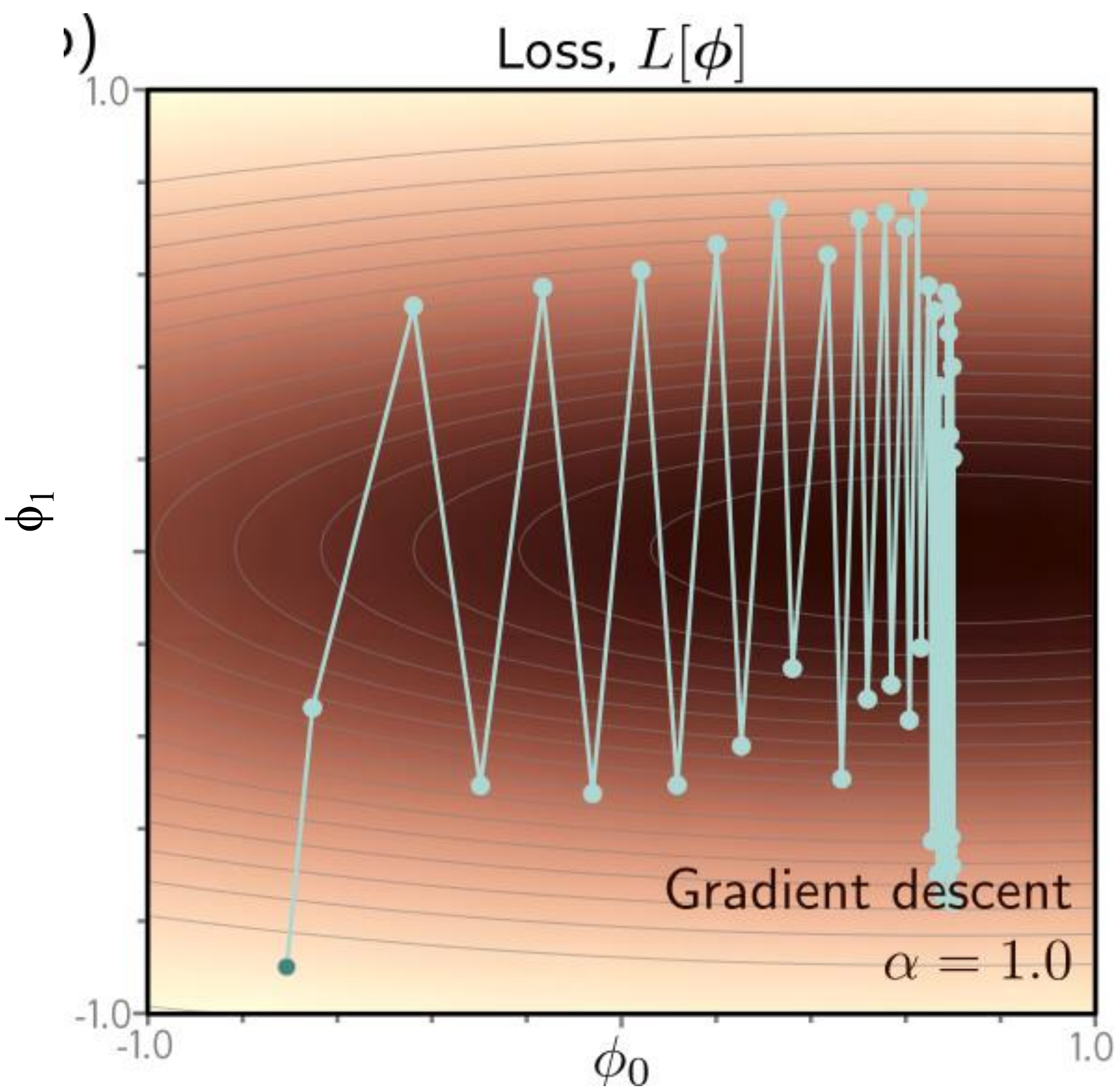
Additional benefit:
Less computationally expensive

Tuning the learning rate



$$\phi \longleftarrow \phi - \alpha \cdot \frac{\partial L}{\partial \phi},$$

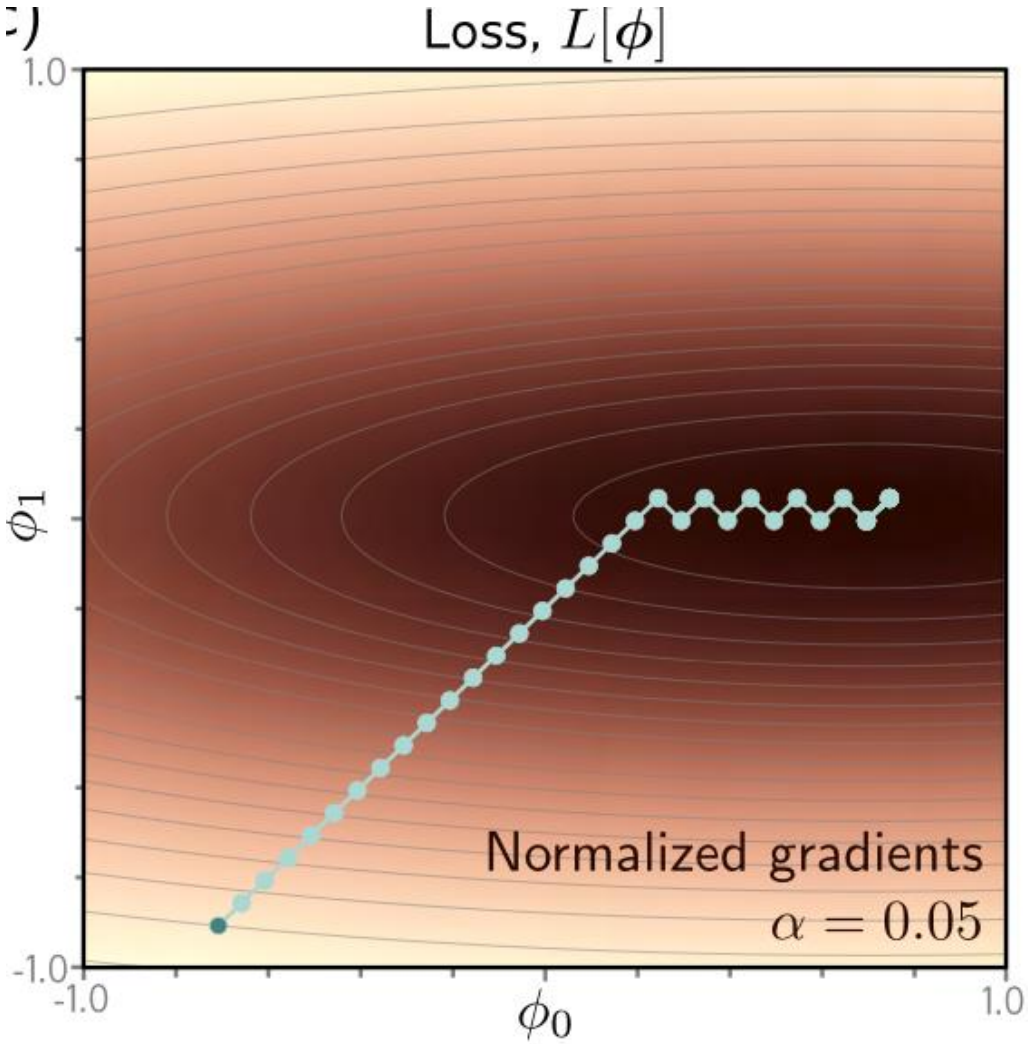
Tuning the learning rate



$$\phi \longleftarrow \phi - \alpha \cdot \frac{\partial L}{\partial \phi},$$

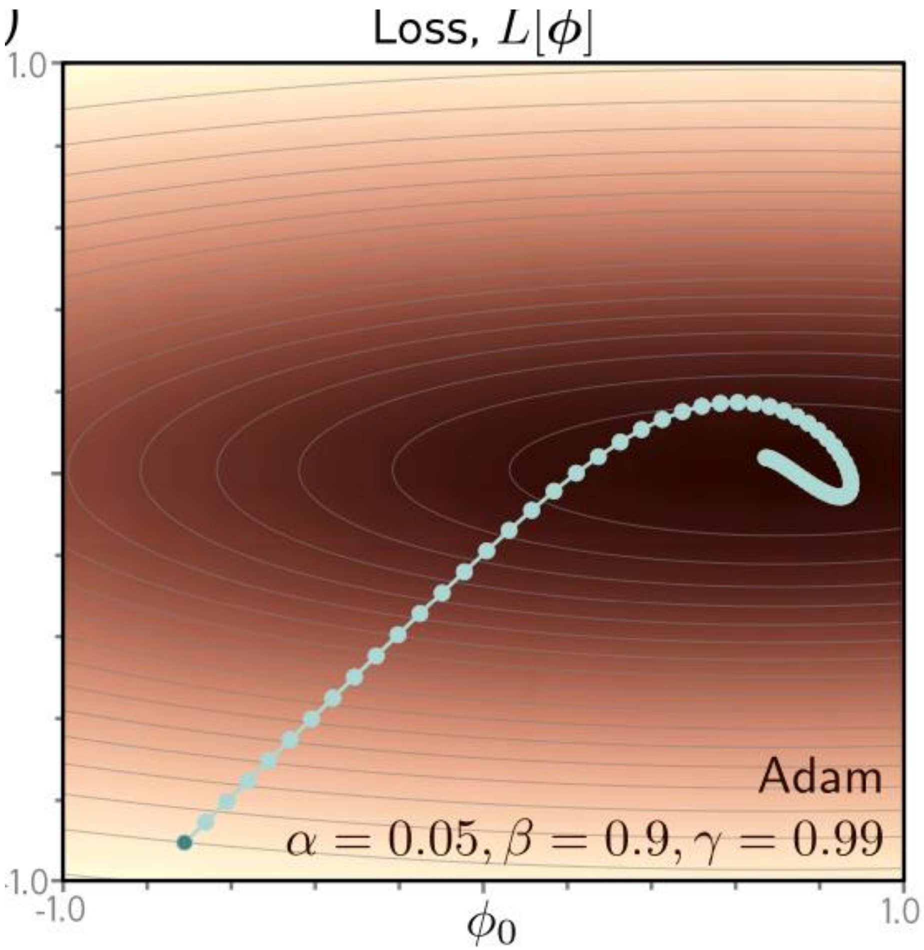
Tuning the learning rate

$$\phi \longleftarrow \phi - \alpha \cdot \frac{\partial L}{\partial \phi},$$



Adapting the learning rate dynamically

Many additional tricks & nuances in practical optimization algorithms to improve convergence for non-convex problems



Example of a good default: ADAM

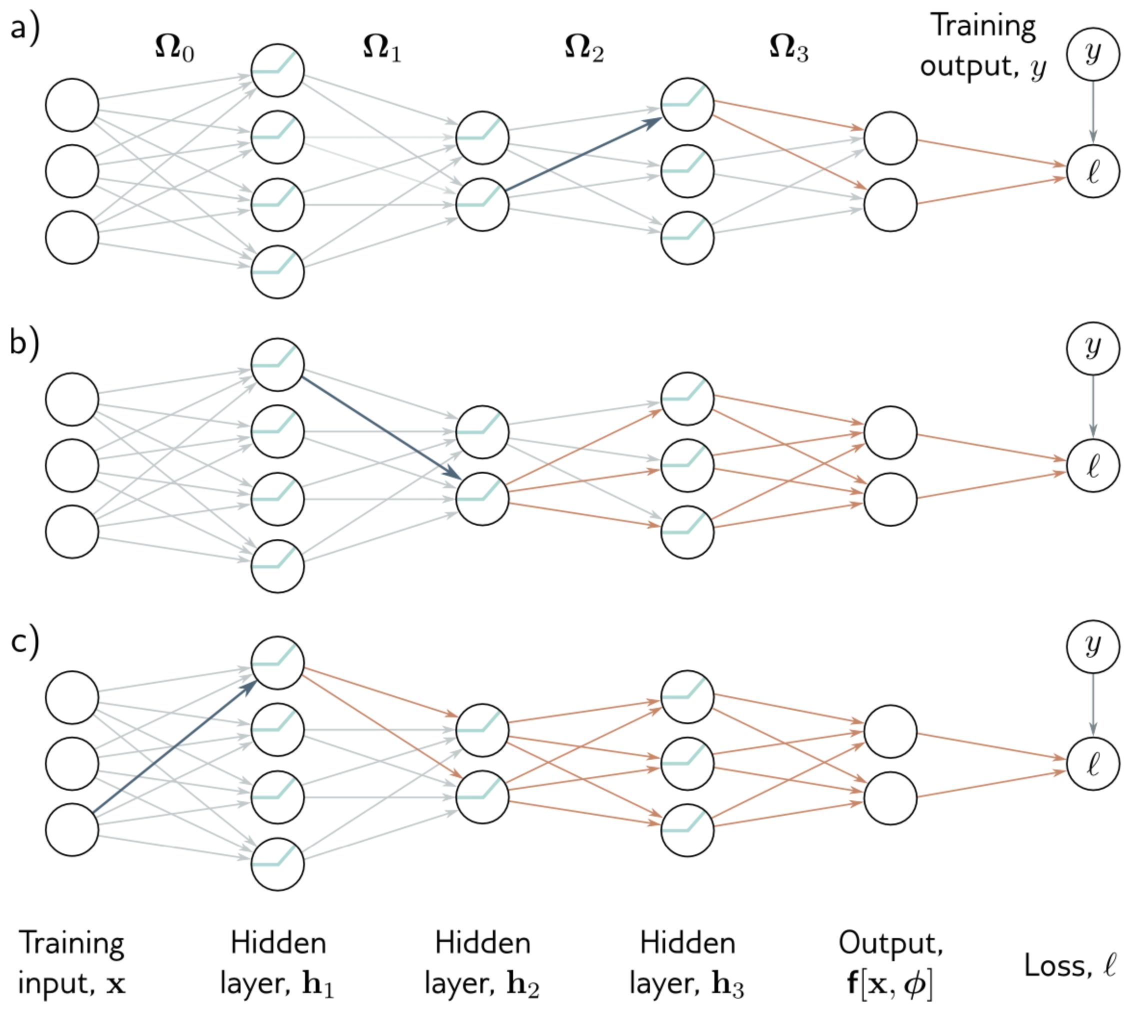
Weighted average over history

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\frac{\partial L[\phi_t]}{\partial \phi}$$

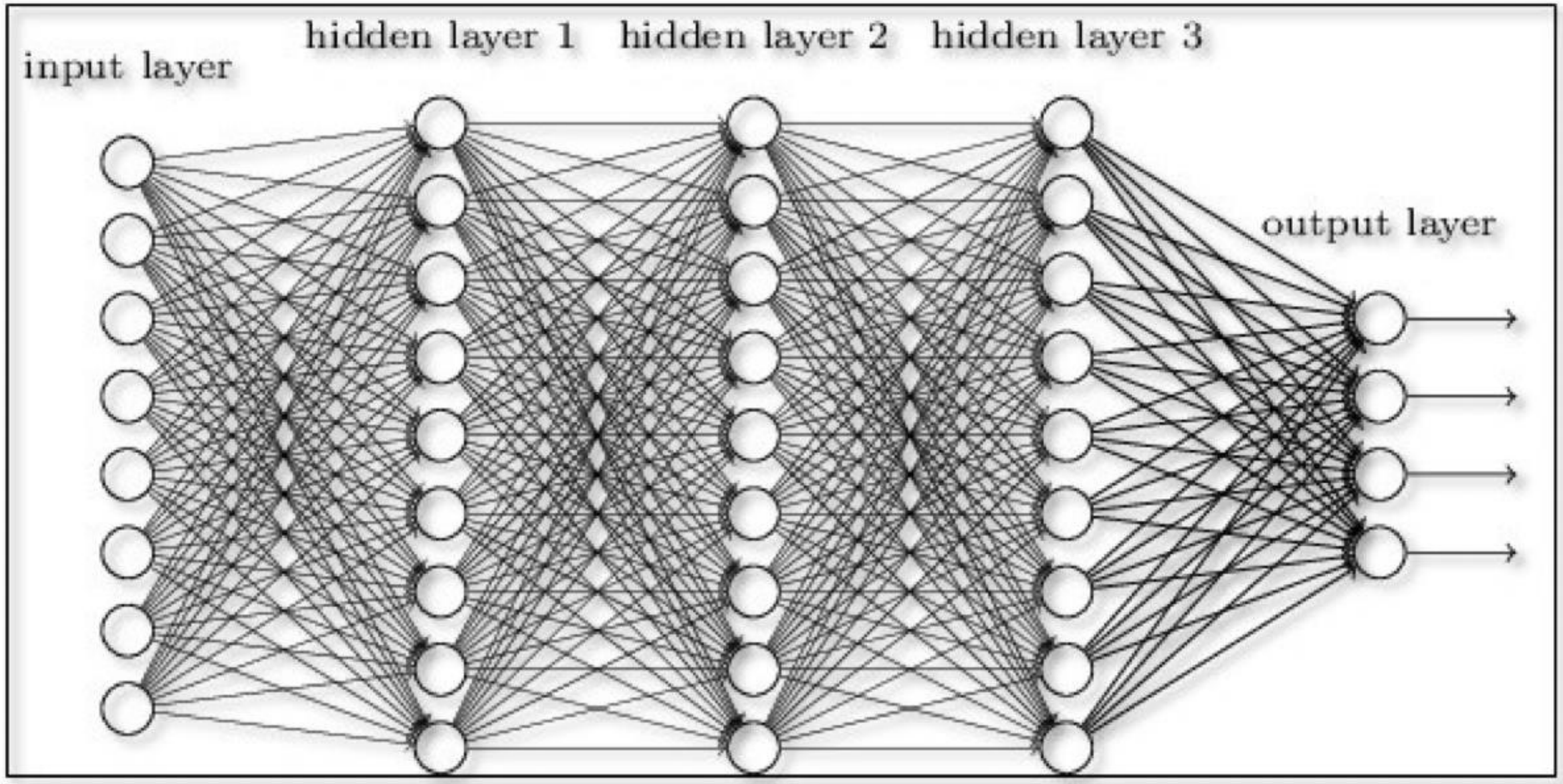
$$\mathbf{v}_{t+1} \leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\frac{\partial L[\phi_t]}{\partial \phi}\right)^2$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon},$$

Gradient descent needs…..GRADIENTS!



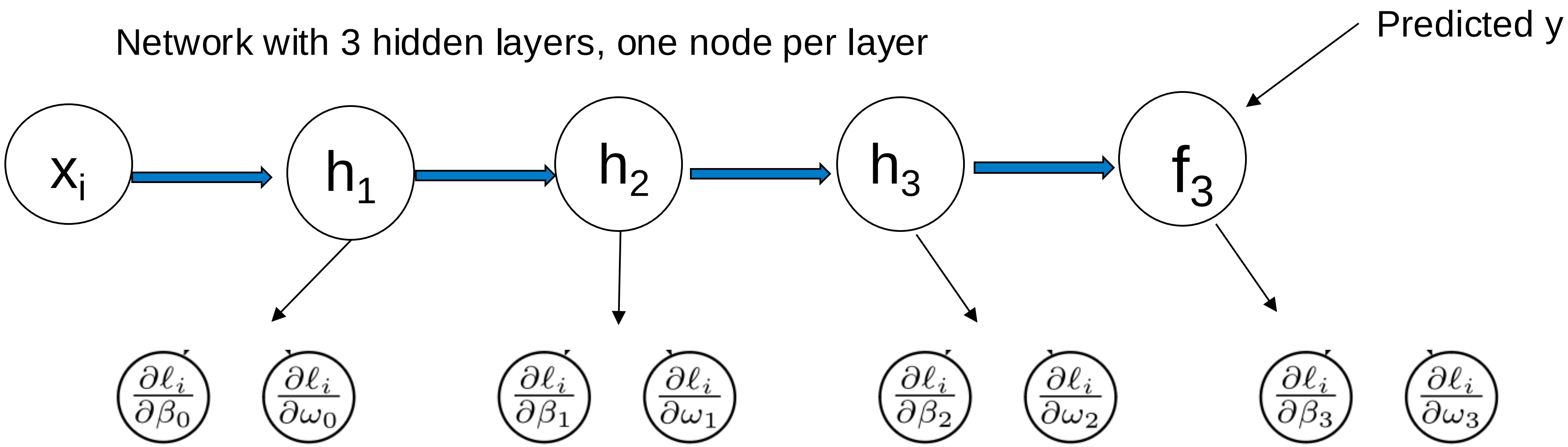
…and there are plenty of gradients



8*9 +9 + 9*9 +9 +9*9 +9 +9*4 +4 = 220 parameters

Biggest networks to date have up to ~10^{12} parameters (e.g. ChatGPT)

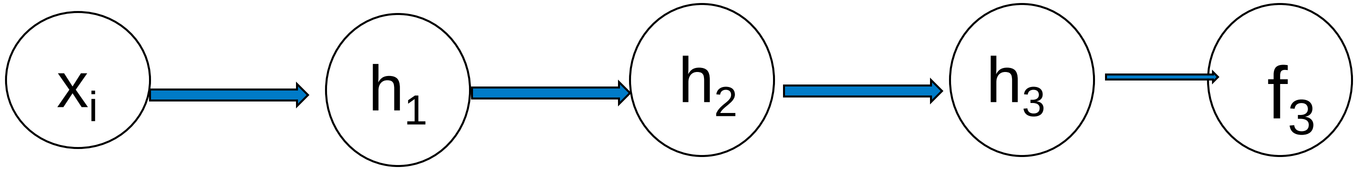
Let’s start simple:



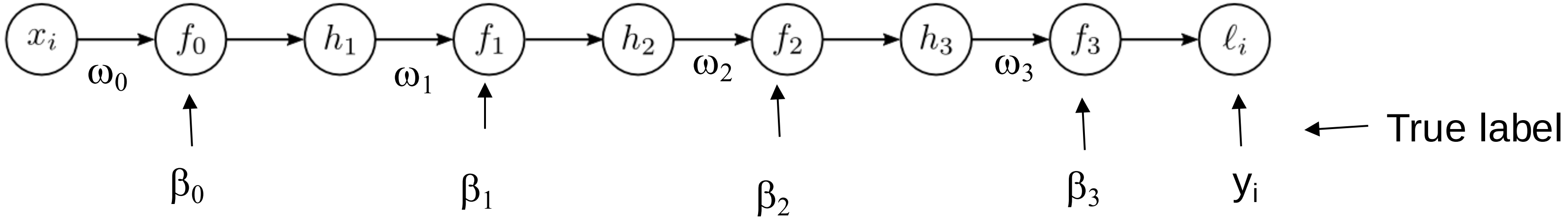
Our goal 8 gradients!

Let's start simple:

Network with 3 hidden layers, one node per layer



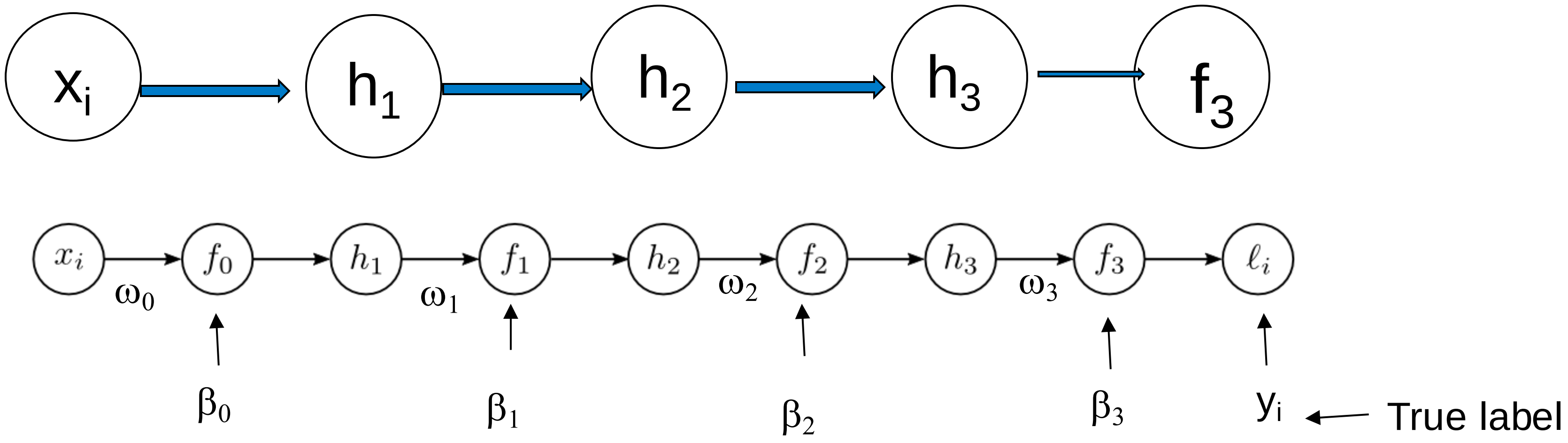
Break down in elements (algorithmic differentiation):



$$
\begin{aligned}
f_0 &= \beta_0 + \omega_0 \cdot x_i \\
h_1 &= a\,[f_0] \\
f_1 &= \beta_1 + \omega_1 \cdot h_1 \\
h_2 &= a\,[f_1] \\
f_2 &= \beta_2 + \omega_2 \cdot h_2 \\
h_3 &= a\,[f_2] \\
f_3 &= \beta_3 + \omega_3 \cdot h_3 \\
\ell_i &= (f_3 - y_i)^2.
\end{aligned}
$$

Let's start simple:

Network with 3 hidden layers, one node per layer

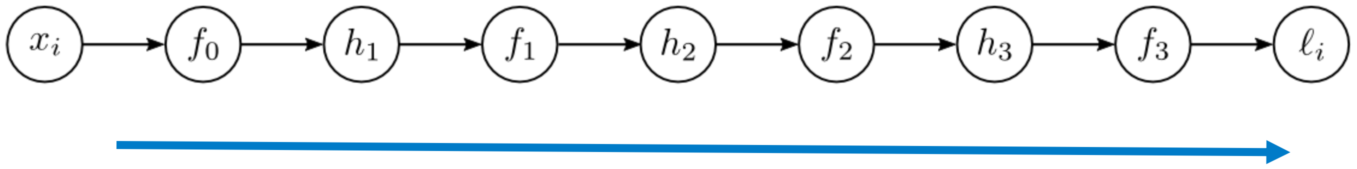


$$
\begin{aligned}
f_0 &= \beta_0 + \omega_0 \cdot x_i \\
h_1 &= a[f_0] \\
f_1 &= \beta_1 + \omega_1 \cdot h_1 \\
h_2 &= a[f_1] \\
f_2 &= \beta_2 + \omega_2 \cdot h_2 \\
h_3 &= a[f_2] \\
f_3 &= \beta_3 + \omega_3 \cdot h_3 \\
\ell_i &= (f_3 - y_i)^2.
\end{aligned}
$$

$$l_i(\omega_0,\beta_0,\omega_1,\beta_1,\omega_2,\beta_2,\omega_3,\beta_3) = (f_3(h_3(f_2(h_2(f_1(h_1(f_0(\omega_0,\beta_0))))))) - y_i)^2$$

Backpropagation: collect the ingredients for gradients

Forward pass



Store these intermediate values for later use

Backpropagation: collect ingredients for gradients

Calculate & store derivatives with respect to intermediate variables

$$l_i\,(\omega_0,\beta_0,\omega_1,\beta_1,\omega_2,\beta_2,\omega_3,\beta_3) = (f_3(h_3(f_2(h_2(\,f_1(h_1(f_0(\omega_0,\beta_0))))))) - y_i)^2$$

$$\frac{\partial \ell_i}{\partial f_3} = 2(f_3 - y_i).$$

$$\frac{\partial \ell_i}{\partial h_3} = \frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}.$$

$$\frac{\partial \ell_i}{\partial f_2} = \frac{\partial h_3}{\partial f_2}\left(\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right)$$

$$\frac{\partial \ell_i}{\partial h_2} = \frac{\partial f_2}{\partial h_2}\left(\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right)$$

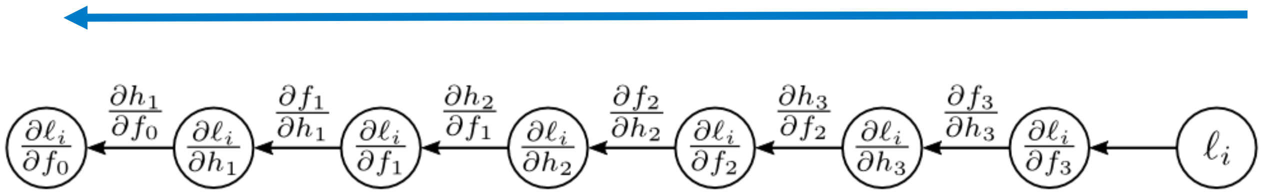
$$\frac{\partial \ell_i}{\partial f_1} = \frac{\partial h_2}{\partial f_1}\left(\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right)$$

$$\frac{\partial \ell_i}{\partial h_1} = \frac{\partial f_1}{\partial h_1}\left(\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right)$$

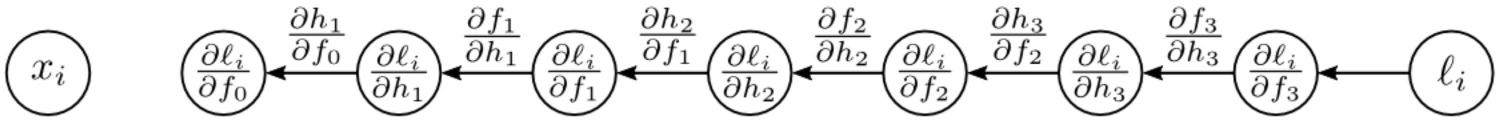
$$\frac{\partial \ell_i}{\partial f_0} = \frac{\partial h_1}{\partial f_0}\left(\frac{\partial f_1}{\partial h_1}\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right)$$

Reuse previously calculated derivative

Computationally very efficient!



Backpropagation: derivatives of simple functions

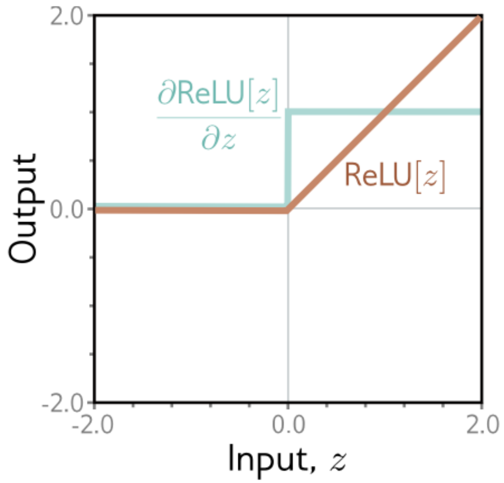


$$
\begin{aligned}
h_3 &= \mathrm{a}\,[f_2] \\
f_3 &= \beta_3 + \omega_3 \cdot h_3
\end{aligned}
$$

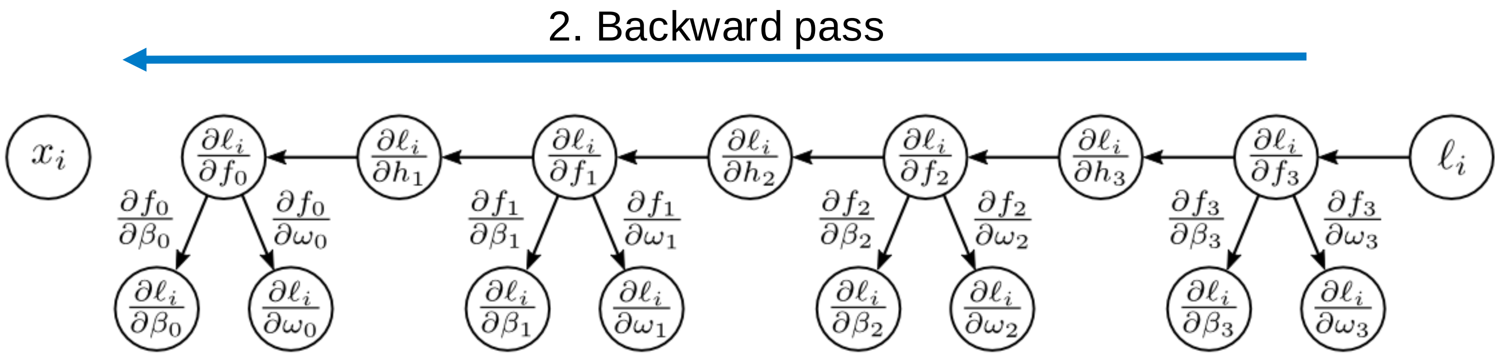
$$\frac{\partial h_3}{\partial f_2} = \begin{cases} 1 & \text{for } f_2>0 \\ 0 & \text{for } f_2<=0 \end{cases}$$

$$\frac{\partial f_3}{\partial h_3} = \omega_3$$

Activation $a[f_i]$



Backpropagation: collect ingredients



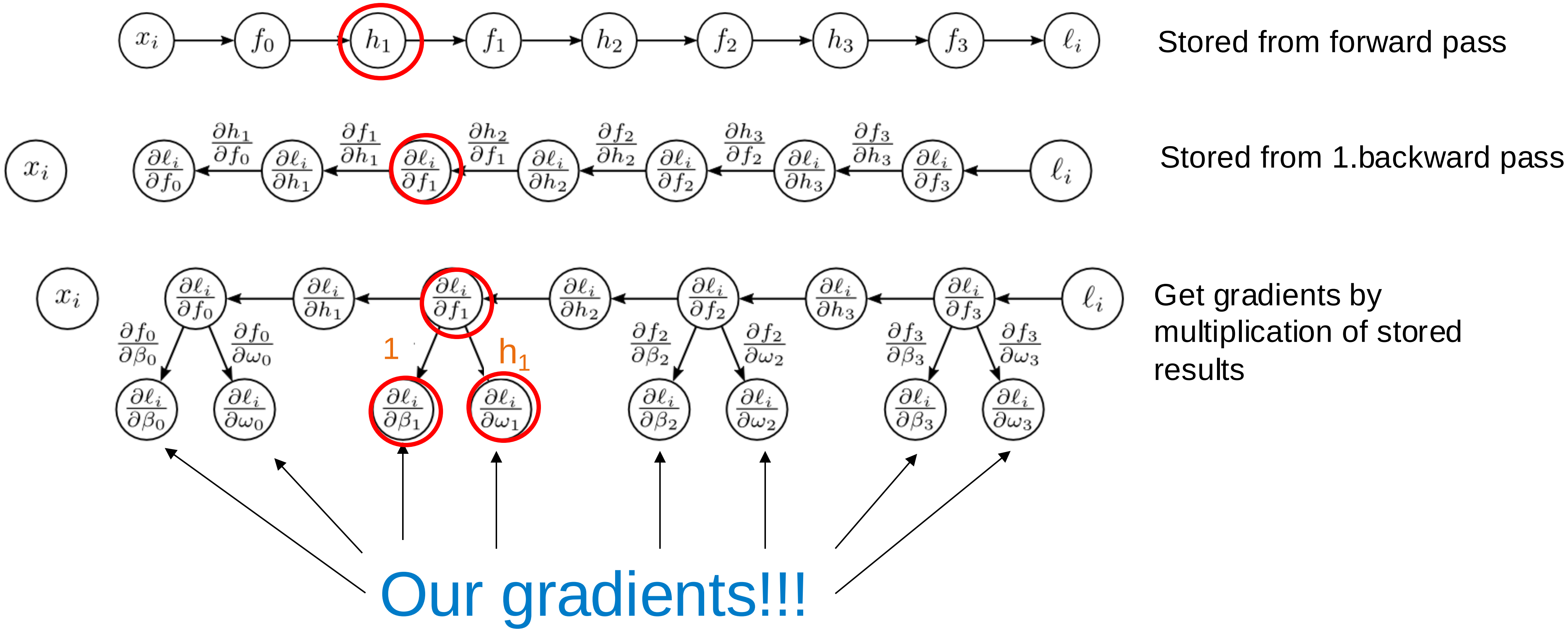
Finally gradient of loss with respect to weights and biases:

$$\frac{\partial f_k}{\partial \beta_k} = 1 \qquad \text{and} \qquad \frac{\partial f_k}{\partial \omega_k} = h_k.$$

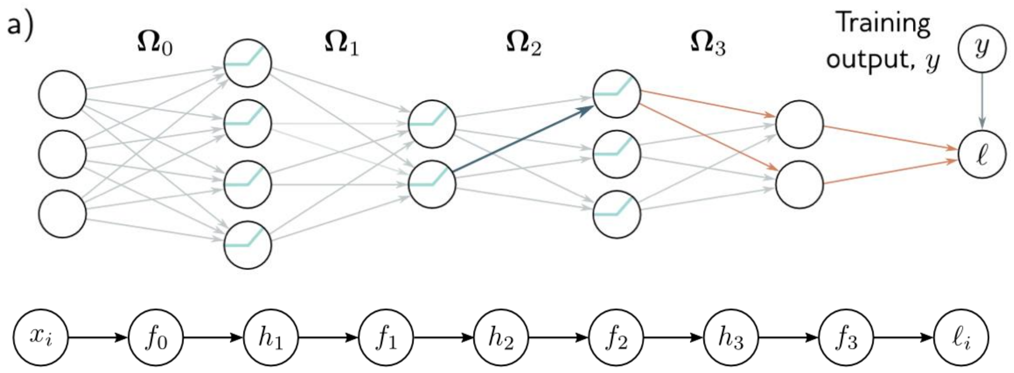
Recall: $f_k = \beta_k + \omega_k h_k$

Already stored in forward pass

Putting it all together…



Now complex network…forward pass

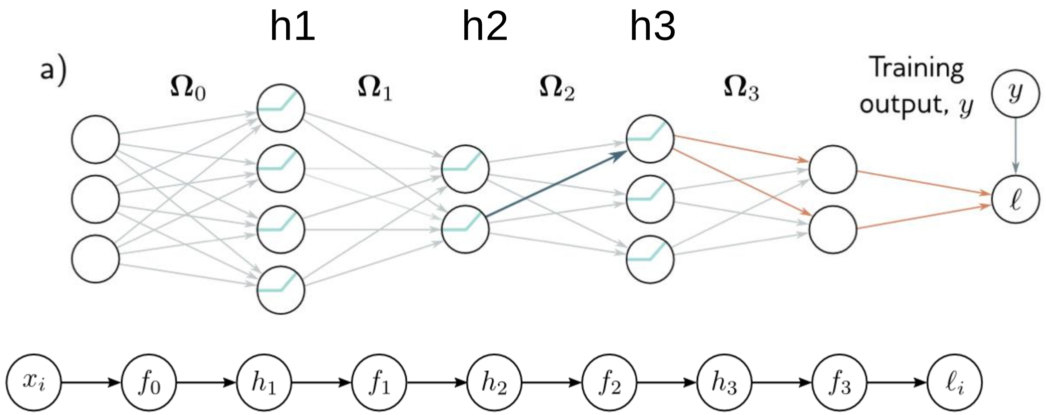


Ω_i: matrix, x_i, b_i, h_i, f_i: vector:

x_i		3x1
$\mathbf{f}_0$	$= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i$	4x1
$\mathbf{h}_1$	$= \mathbf{a}[\mathbf{f}_0]$	4x1
$\mathbf{f}_1$	$= \boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1$	2x1
$\mathbf{h}_2$	$= \mathbf{a}[\mathbf{f}_1]$	2x1
$\mathbf{f}_2$	$= \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2$	3x1
$\mathbf{h}_3$	$= \mathbf{a}[\mathbf{f}_2]$	3x1
$\mathbf{f}_3$	$= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3$	2x1
ℓ_i	$= \mathrm{l}[\mathbf{f}_3, y_i],$	1x1

$$\Omega_0 = \begin{pmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{21} & \omega_{22} & \omega_{23} \\ \omega_{31} & \omega_{32} & \omega_{33} \\ \omega_{41} & \omega_{42} & \omega_{43} \end{pmatrix} \begin{pmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \end{pmatrix}$$

Now complex model…backward pass



$$
\begin{array}{rcll}
\mathbf{f}_2 &=& \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2 & \text{3x1} \\
\mathbf{h}_3 &=& \mathbf{a}[\mathbf{f}_2] & \text{3x1} \\
\mathbf{f}_3 &=& \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3\mathbf{h}_3 & \text{2x1} \\
\ell_i &=& \mathrm{l}[\mathbf{f}_3, y_i], & \text{1x1}
\end{array}
$$

$$
\frac{\partial \ell_i}{\partial \mathbf{f}_2} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3}\frac{\partial \ell_i}{\partial \mathbf{f}_3}.
$$

Derivatives of vectors become matrices

3x3 3x2 2x1

and scalar multiplication turns into matrix multiplication

but there are still simplifications

$$\frac{\partial \ell_i}{\partial \mathbf{f}_2} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3} \frac{\partial \ell_i}{\partial \mathbf{f}_3}.$$

$$\frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3} = \frac{\partial}{\partial \mathbf{h}_3} \left(\boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3\right) = \boldsymbol{\Omega}_3^T.$$

proof with matrix calculus

$$\frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2}$$

diagonal matrix -> replace by vector $\mathbb{I}(f_2>0)$ and pointwise multiply
very compute efficient!

$$\boxed{\frac{\partial \ell_i}{\partial \mathbf{f}_2} = \mathbb{I}(f_2>0)\ \Omega_3^{\mathrm{T}} \frac{\partial \ell_i}{\partial \mathbf{f}_3}.}$$

Correspondingly for all other f_i, h_i

Summary backpropagation

Neural network $f(\mathbf{x}_i,\phi)$: K hidden layers, activation function (e.g. ReLu)

Loss: $$\ell_i = \mathrm{l}[\mathbf{f}[\mathbf{x}_i,\boldsymbol{\phi}],\mathbf{y}_i].$$

Forward pass: compute and store

$$\begin{aligned}
\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}_i \\
\mathbf{h}_k &= \mathbf{a}[\mathbf{f}_{k-1}] & k \in \{1,2,\ldots,K\} \\
\mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k. & k \in \{1,2,\ldots,K\}
\end{aligned}$$

Backward pass: compute and store

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} & k \in \{K,K-1,\ldots,1\} \\
\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k}\mathbf{h}_k^T & k \in \{K,K-1,\ldots,1\} \\
\frac{\partial \ell_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial \ell_i}{\partial \mathbf{f}_k}\right), & k \in \{K,K-1,\ldots,1\}
\end{aligned}$$

Finally first layer

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \\
\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0}\mathbf{x}_i^T
\end{aligned}$$

Point wise multiplication

Comments

- Backpropagation is super compute efficient but not memory efficient

  ➔ All intermediate values of the forward pass and all weight matrices are stored -> might limit size of model to be trained

- If memory allows, perform the forward and backward passes for the entire batch in parallel

  - Matrices and vectors get another index for the date event, i.,e, become a multi-dimensional tensor
  - “tensor” = generalization of matrix to arbitrary dimension (vector = 1d tensor, matrix=2d tenser, etc.)

- All of this is implemented in TensorFlow and PyTorch

  - After this school you will just use a single line, like loss.backward()

Vanishing and exploding gradients

Due to the matrix multiplication

- very small weights may get to 0 and effectively remove nodes from the network “vanishing gradient”

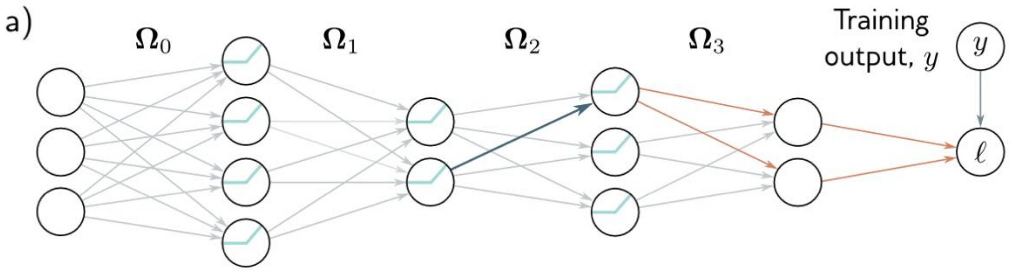
- Large weights might get exponentiated to values beyond the precision of floating point arithmetic

Initialisation

To avoid vanishing or exploding gradients sample from normal distribution with mean 0 and variance

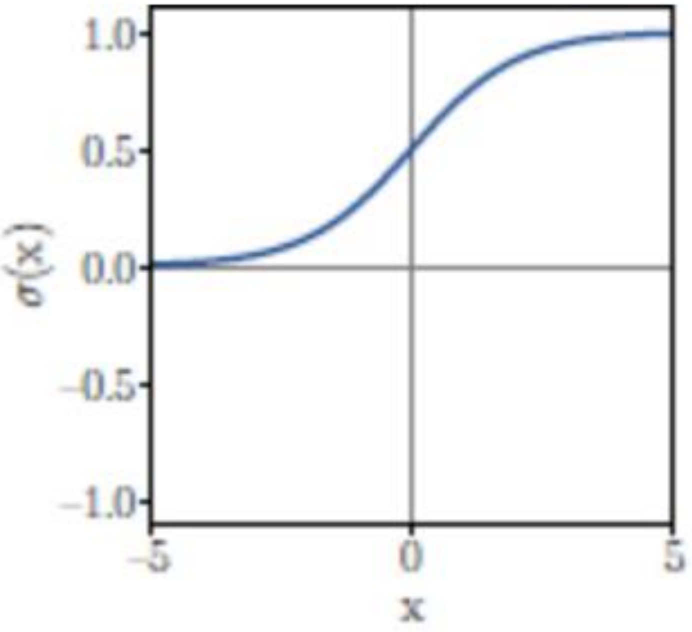
$$\sigma_\Omega^2 = \frac{4}{D_h + D_{h'}}.$$

D_h: dimension of layer where weights are applied
D_h': dimension of layer to which they are fed

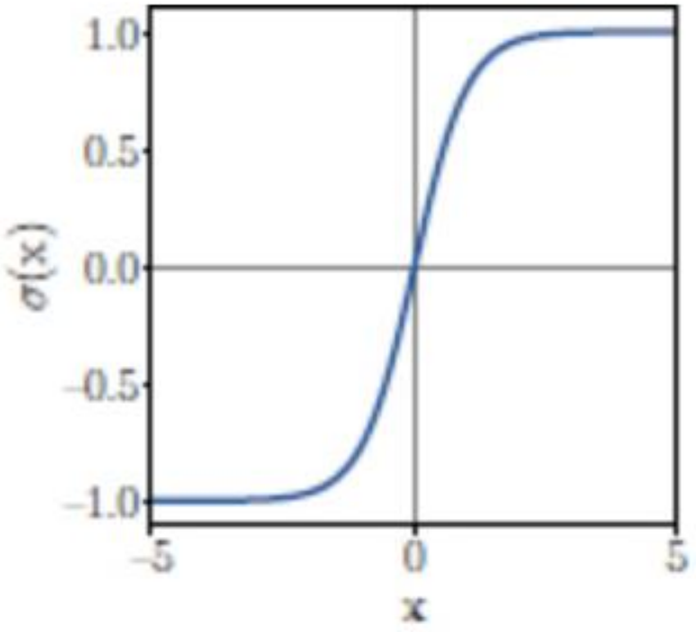


Complex network architectures may use separate simple networks to determine initialization for large complex networks

Gradients of other activation functions



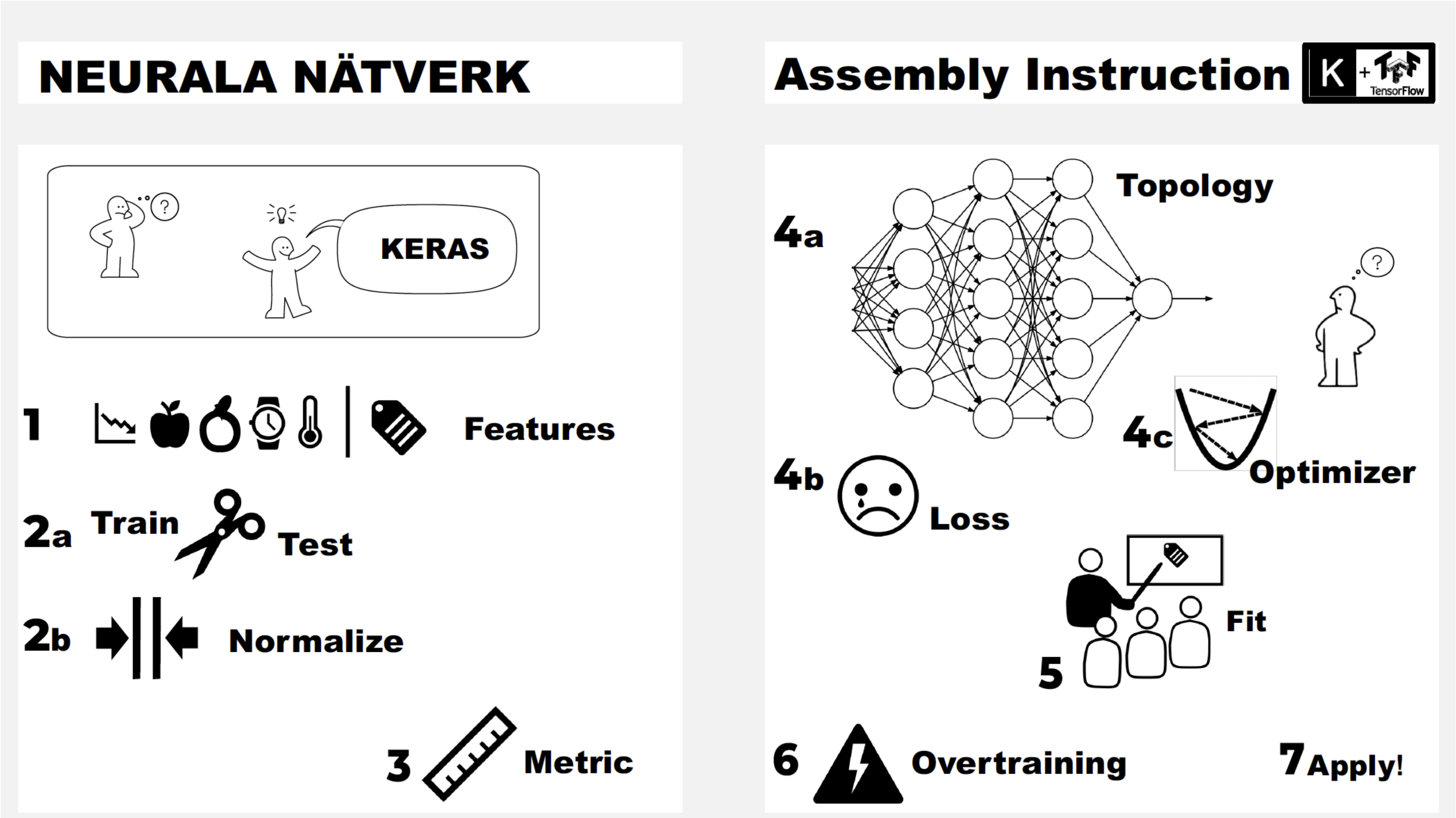
Logistic sigmoid



tanh

The full chain

ML frameworks like TensorFlow (with Keras API), PyTorch and JAX put a lot of the pieces together to provide a performant setup
-> See exercises with Peter



The network

```
def create_model(input_dim, output_dim):
  return torch.nn.Sequential(
      torch.nn.Linear(input_dim, 9),
      torch.nn.ReLU(),
      torch.nn.Linear(9, 9),
      torch.nn.ReLU(),
      torch.nn.Linear(9, 9),
      torch.nn.ReLU(),
      torch.nn.Linear(9, output_dim),
      [redacted]
  )
```

….improving step by step

Learn by revisiting the data often and adjusting

```
f = initial_guess()
for n in range(steps):
        examples ~ p(data)
        loss = evaluate(f, examples)
        adjustment = react(loss, f)
        f = new_hypo(f, adjustment)
```

A full training loop

Data $s \sim p(s)$

```
def learn(samples):
    features, labels = samples
    model = MyModel()
    loss_func = torch.nn.MSELoss()
    opt = torch.optim.Adam(
        model.parameters(), lr = 1e-3
    )

    for i in range(steps):
        predictions = model(samples)
        loss = loss_func(predictions, labels)
        loss.backward()
        opt.step()
        opt.zero_grad()
    return model
```

Hypothesis Set $f_\phi(x)$

Objective $L(s) = L(f_\phi(x), z)$

Learning Algorithm $\mathrm{argmin}_\phi L_{\mathrm{MC}}$

Gradient Computation $\nabla_\phi L_{\mathrm{MC}}(\phi)$

Trained Model $f_{\hat{\phi}}(x)$

Back-up