02456 – Week 2
Learning

Jes Frellsen

Technical University of Denmark

08 September 2025

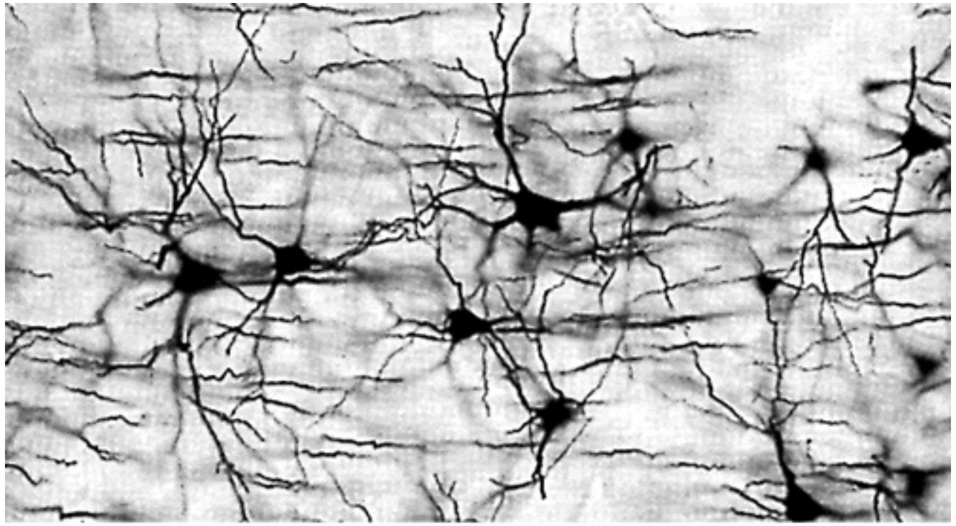
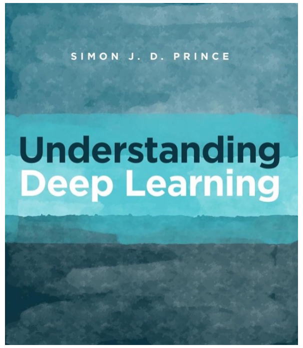


Image from: https://alanturing.net

Slides codeveloped by Ole Winther, Tapani Raiko and Jes Frellsen

Menu of the day



- Week 1: Neural nets
- Week 2: Learning
- Week 3: Tricks of The Trade
- Week 4: CNNs
- Week 5: RNNs
- Week 6: Transformers
- Week 7: Unsupervised
- Week 8: Mini-project

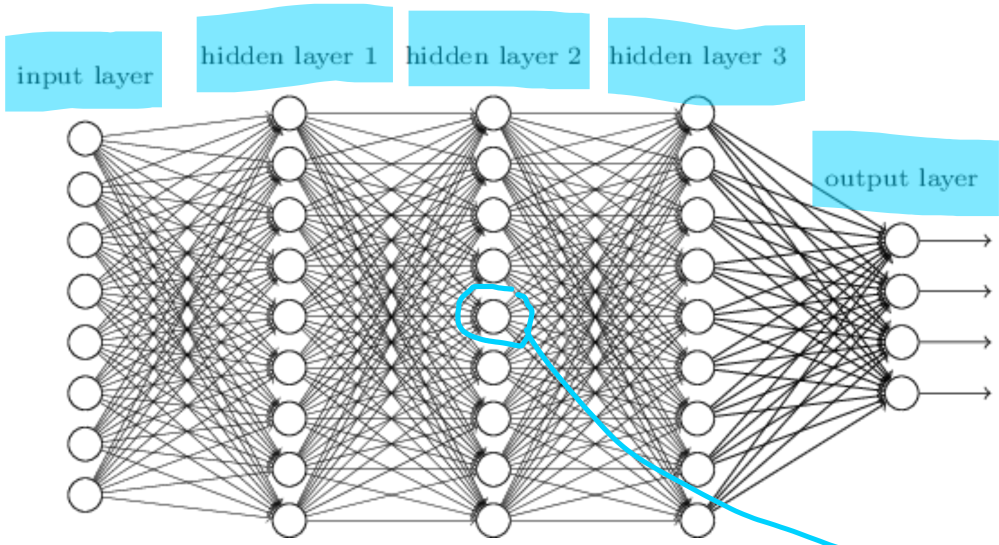
Lecture

- Loss functions (ch 5)
- Fitting models (ch 6)
- Gradients and initial. (c 7)

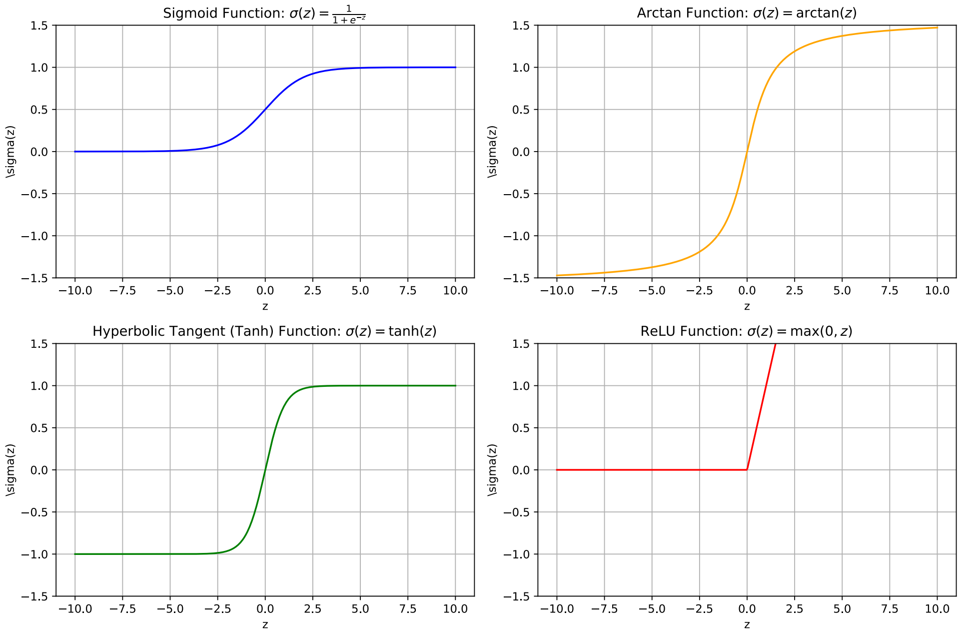
Exercises

- **Notebook**:
  - `2.1 FNN AutoDif Nanograd.ipynb`
  - Try to code autodiff yourself
- **Problems:** 5.9, 6.5, 7.10

Recap: Neural networks



- Node
- Artificial neuron



We can write the **output of layer ℓ** as

$$f^{(\ell)}(\mathbf{h}) = \sigma\left(W^{(\ell)}\mathbf{h} + \mathbf{b}^{(\ell)}\right).$$

The **joint function** is then

$$f_\phi(\mathbf{x}) = \mathbf{y} = f^{(4)}\left(f^{(3)}\left(f^{(2)}\left(f^{(1)}(\mathbf{x})\right)\right)\right)$$

Terminology:
- Fully connected neural network
- Feed forward neural network (FFN)
- Multilayer perceptron (MLP)

Training criterion

- For training data $\{(\mathbf{x}_i, y_i)\}_i$, we want to find neural net parameters

$$\phi = \{W^{(\ell)}, \mathbf{b}^{(\ell)}\}_\ell$$

  s.t. we **minimize the mismatch** between $f_\phi(\mathbf{x}_i)$ and y_i

- We defined a **loss function** $L(\phi)$ capturing this **mismatch**

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}}\, L(\phi)$$

- How do we define such a **loss function**?
  - Last week we saw mean squared error $L(\phi) = \frac{1}{n}\sum_{i=1}^{n}\left(f_\phi(\mathbf{x}_i) - y_i\right)^2$
  - *Is there some principled framework we can use?*

Maximum likelihood estimation (MLE)

Find **parameters** that maximises the probability of the data $\{(\mathbf{x}_i, y_i)\}_i$

$$\hat{\phi} = \arg\max_{\phi} \prod_i p(y_i|\mathbf{x}_i, \phi)$$

We parametrize the probability with a neural net

$$p(y_i|f_\phi(\mathbf{x}_i))$$

Normally we do this in log-space

$$\hat{\phi} = \arg\max_{\phi} \sum_i \log p(y_i|f_\phi(\mathbf{x}_i))$$

Log-likelihood function

We can use the **negative log-likelihood** as a loss

$$L(\phi) = -\sum_i \log p(y_i|f_\phi(\mathbf{x}_i))$$

$$\hat{\phi} = \arg\min_{\phi} L(\phi)$$

Why is MLE a good framework?

- Strong theoretical foundation, e.g.,
- **Consistency**: converges to the true parameter value as $n \to \infty$
- **Efficient**: Achieving the lowest possible variance as $n \to \infty$

Probabilistic inference (predictions)

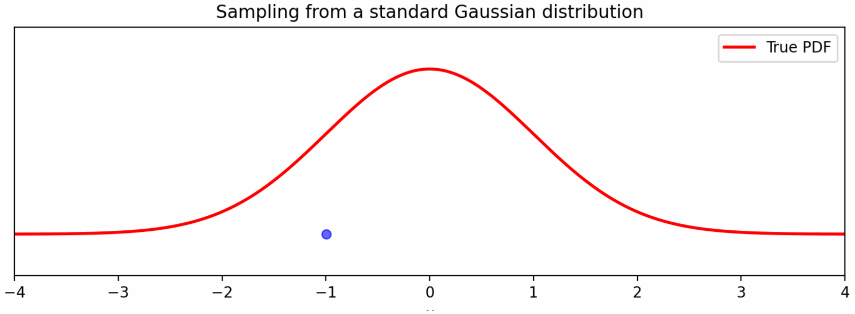
- For **learned** $\hat{\phi}$, how can I make **predictions** using $p\left(y \middle| f_{\hat{\phi}}(\mathbf{x})\right)$?
- For a given $\mathbf{x}$, I my **predictions** can be:
  - The most probable value: $\hat{y} = \arg\max_y p\left(y \middle| f_{\hat{\phi}}(\mathbf{x})\right)$
  - The expected value: $\hat{y} = \mathbb{E}_{y \sim p\left(y \middle| f_{\hat{\phi}}(\mathbf{x})\right)}[y]$
  - A sample: $y \sim p\left(y \middle| f_{\hat{\phi}}(\mathbf{x})\right)$

Mostly used

For a Gaussian $\mathcal{N}(y|\mu, \sigma)$ these are both μ

So, if $\mu = f_{\hat{\phi}}(\mathbf{x})$, what would the prediction be?

$f_{\hat{\phi}}(\mathbf{x})$



Gaussian distribution (regressions)

- If we assume y is Gaussian distributed

$$p(y|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

- The loss (negative log-likelihood) becomes

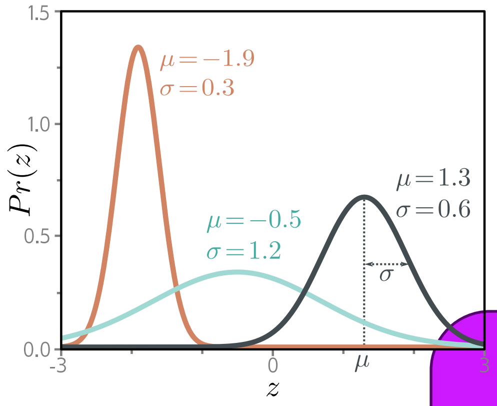
$$L(\phi) = -\sum_i \log p\big(y_i \big| f_\phi(\mathbf{x}), \sigma\big)$$

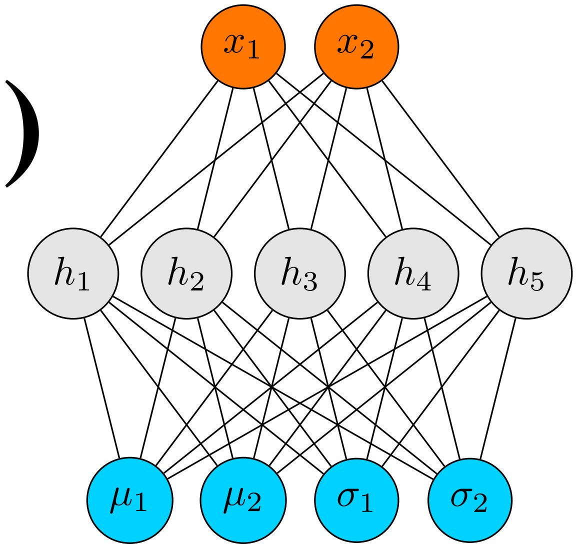
$$= -\sum_i \left(-\frac{(y_i-\mu)^2}{\cancel{2\sigma^2}} - \cancel{\frac{1}{2}\log 2\pi\sigma^2}\right)$$

- Assuming, $\mu = f_\phi(\mathbf{x})$ we have

$$\hat{\phi} = \arg\min_{\phi} L(\phi) = \sum_i \big(y_i - f_\phi(\mathbf{x})\big)^2$$

Mean squared error!





$$\mathcal{N}\left(\begin{bmatrix}\mu_1\\ \mu_2\end{bmatrix}, \begin{bmatrix}\sigma_1 & 0\\ 0 & \sigma_2\end{bmatrix}\right)$$

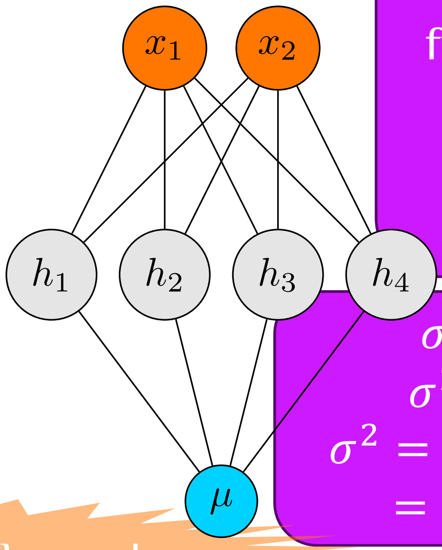
Which activation function can make the output no-negative?

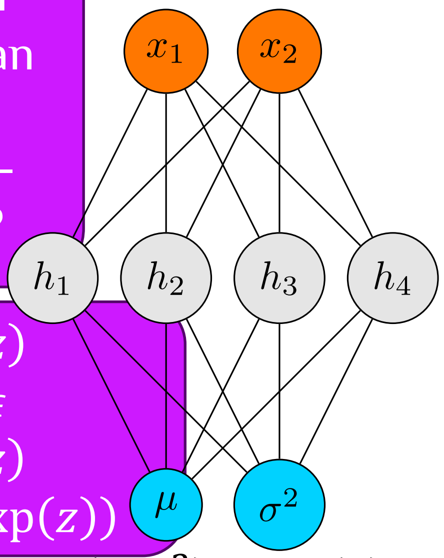
$\sigma^2 = \exp(z)$

$\sigma^2 = z^2 + \epsilon$

$\sigma^2 = \mathrm{softplus}(z)$

$= \log(1+\exp(z))$



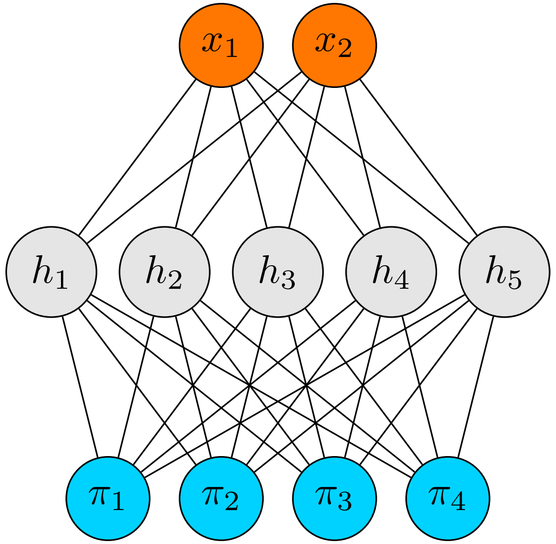


$(\mu, \sigma^2) = f_\phi(\mathbf{x})$

Plot from: Prince SJD. Understanding deep learning. MIT press, 2023.

Categorial distribution (multiclass classification)

For four classes this is
$0 \rightarrow (1,0,0,0)^T$
$1 \rightarrow (0,1,0,0)^T$
$2 \rightarrow (0,0,1,0)^T$
$3 \rightarrow (0,0,0,1)^T$



- If y categorial and is one-hot encoded then

$$p(y|\pi) = \prod_d \pi_d^{y_d}$$

where $\pi = f_\phi(\mathbf{x})$

- The loss (negative log-likelihood) becomes

$$L(\phi) = -\sum_i \sum_d y_{id} \log \pi_d$$

Sum over data points

Sum over dimensions

- **Softmax activation function** converts neural network outputs into probabilities

$$\pi = \frac{\exp(z_d)}{\sum_d \exp(z_d)}$$

Also know as **cross-entropy**, which is between p and q

$$H(p,q) = \sum_x p(x) \log q(x)$$

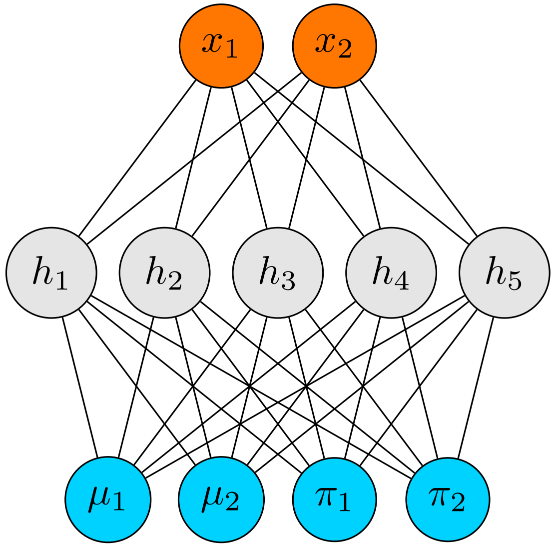
Combinations: Multiple outputs

- If $\mathbf{y}$ is multiple dimensional and dimensions are independent given $\mathbf{x}$,

$$p(\mathbf{y}|f_\phi(\mathbf{x})) = \prod_d p(y_d|f_\phi(\mathbf{x})_d)$$

and the loss becomes

$$L(\phi) = -\sum_i \sum_d p(y_{id}|f_\phi(\mathbf{x}_i)_d)$$



$$p(\mathbf{y}|f_\phi(\mathbf{x})) = \\ \mathcal{N}\left(\mathbf{y}_{1:2}|f_\phi(\mathbf{x})_{1:2}, \mathrm{diag}(\sigma_1, \sigma_2)^T\right) \cdot \mathrm{Cat}(\mathbf{y}_{3:4}|f_\phi(\mathbf{x})_{3:4})$$

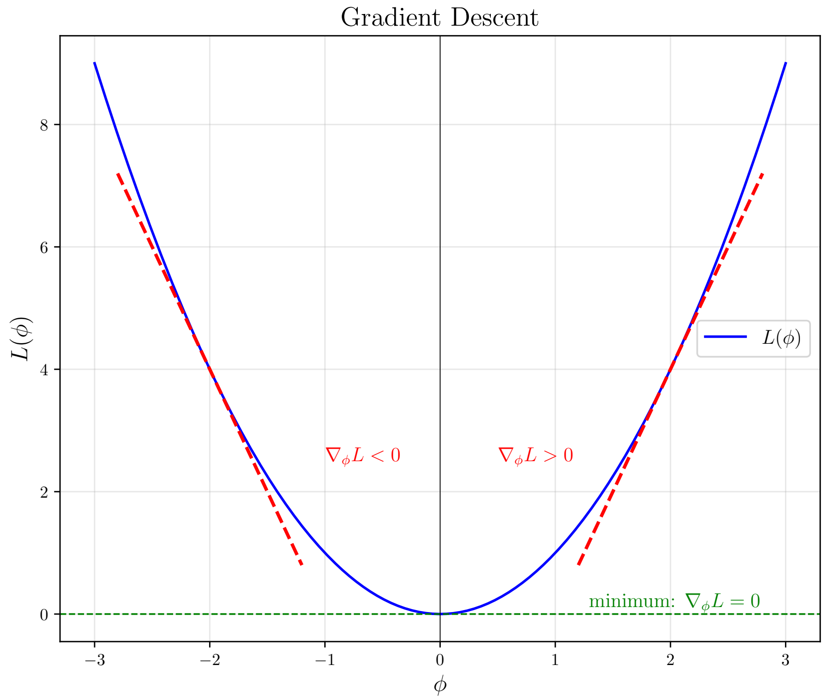
Learning: gradient decent

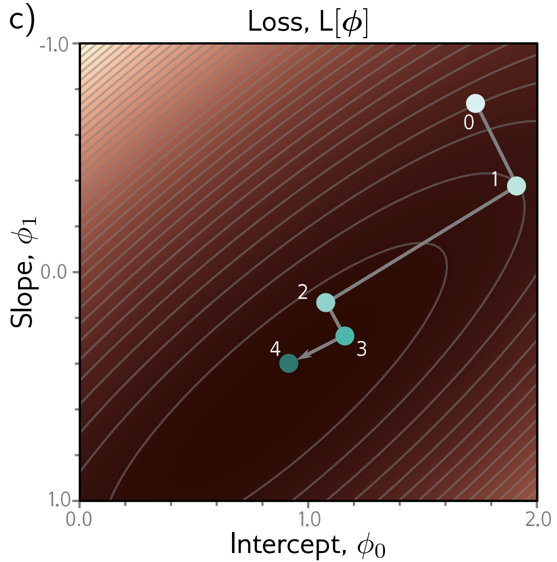
We want to find $\hat{\phi} = \underset{\phi}{\arg\min}\, L(\phi)$

- Initialise $\phi^{(0)}$ randomly (more about this later)
- Iterate (for $t \in 1, \dots, k$)
    - Step 1 (gradient): Iterate $\nabla_\phi L\left(\phi^{(t)}\right) = \begin{pmatrix} \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_D} \end{pmatrix}$
    - Step 2 (update parameters): $\phi^{(t+1)} = \phi^{(t)} - \eta \nabla_\phi L\left(\phi^{(t)}\right)$

Where

- k is the number of iterations (steps)
- η is the step-size or learning rate

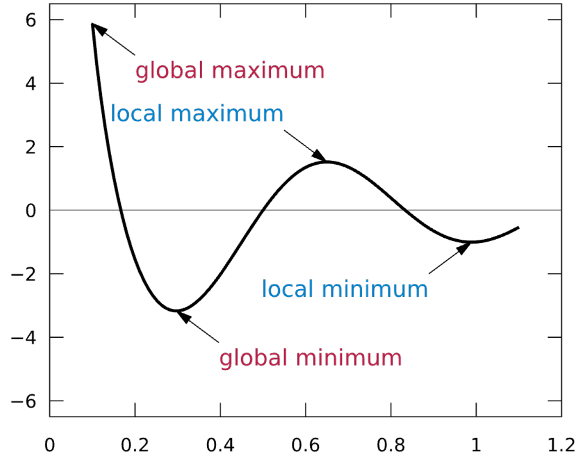


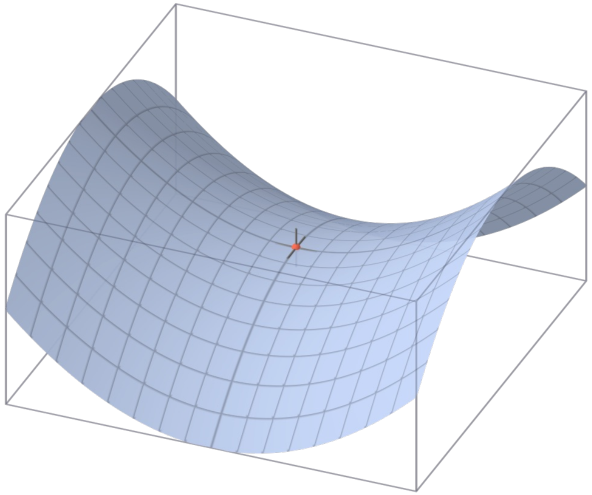


Bottom figure from: Prince SJD. Understanding deep learning. MIT press, 2023.

Local minima and saddle points

- Learning staps a critical points ϕ, for which $\nabla_\phi L(\phi) = 0$
  - **Local minimal**
    - All eigenvalues of Hessian H_L are positive
  - **Saddle point**
    - Hessian H_L has both positive and negative eigenvalues
- Gradient decent **can get stuck** in local minima
- Can **escape** saddle points
  - Local minima dominate in low-dimensional
  - Saddle points dominate in high dimensions (Dauphin et al., 2014, Choromanska et al., 2015)





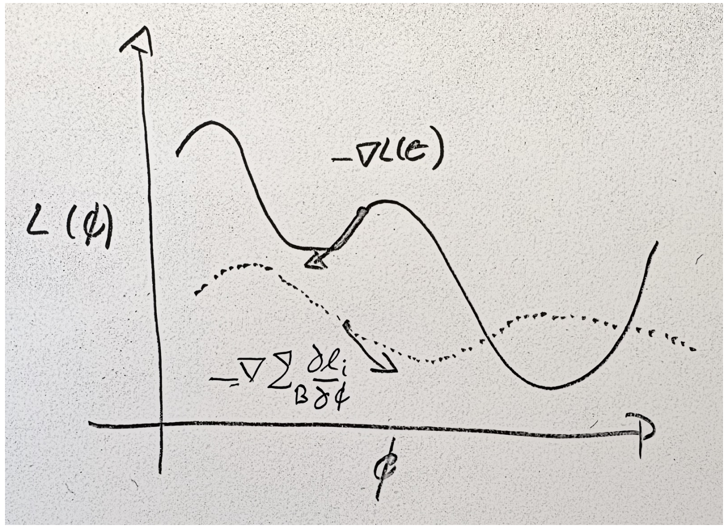
Figures from Wikipedia

Stochastic gradient descent (minibatch)

- At each step, the gradient is calculated on a minibatch

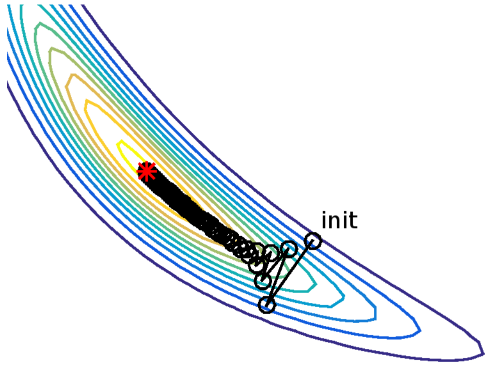
$$\phi^{(t+1)} = \phi^{(t)} - \sum_{i \in \mathcal{B}_t} \frac{\partial l_i\left(\phi^{(t)}\right)}{\partial \phi}$$

  - The batch $\mathcal{B}_t \subseteq \{1, \dots, n\}$ index-set is drawn stochasticallys
    - Usually **w/o replacement** and a full pass of the data is called an **epoch**
  - $l_i\left(\phi^{(t)}\right)$ is the **loss** on $(\mathbf{x}_i, y_i)$ assuming that $L(\phi) = \sum_{i=1}^{n} l_i(\phi)$
- Properties:
  - Uses an **unbiased** estimate of the gradient
    - The gradient is correct on average
    - Each training point contribute equally
  - **Less computation expensive**
  - Can **escape** local minima and saddle points
  - May help the network **generalise** better

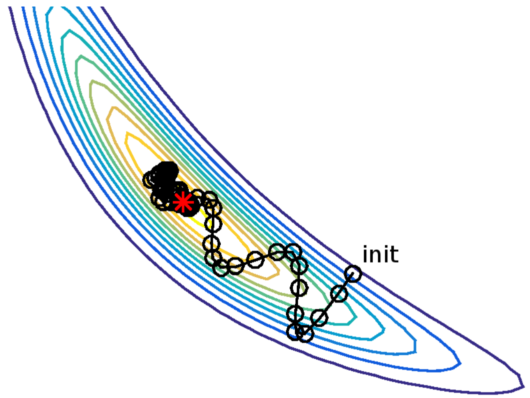


Momentum

- We can use a weighted (decaying) average of previous gradients
  - $m^{(t+1)} = \beta m^{(t)} + (1-\beta)\sum_{i\in\mathcal{B}_t} \frac{\partial l_i(\phi^{(t)})}{\partial \phi}$
  - $\phi^{(t+1)} = \phi^{(t)} - \eta m^{(t+1)}$
    where $\beta \in [0,1)$ controls the smoothing



- **Smoother** the trajectory
- Reduces **oscillations**



Adam (Adaptive moment estimation)

- In Adam, we normalise the gradients by their variance
- Estimate the first and second moments (weighted) of the gradients
  - $m^{(t+1)} = \beta m^{(t)} + (1-\beta)\nabla_{\phi} L\left(\phi^{(t)}\right)$
  - $v^{(t+1)} = \gamma v^{(t)} + (1-\gamma)\left(\nabla_{\phi} L\left(\phi^{(t)}\right)\right)^2$

Mini-batched

- Compensate for initial values close to zero
  - $\tilde{m}^{(t+1)} = \frac{m^{(t+1)}}{1-\beta^{t+1}}$ and $\tilde{v}^{(t+1)} = \frac{v^{(t)}}{1-\gamma^{t+1}}$
- Update the parameters
  - $\phi^{(t+1)} = \phi^{(t)} - \eta \frac{\tilde{m}^{(t+1)}}{\sqrt{\tilde{v}^{(t+1)}}+\epsilon}$

Acts as signal-to-noise ratio

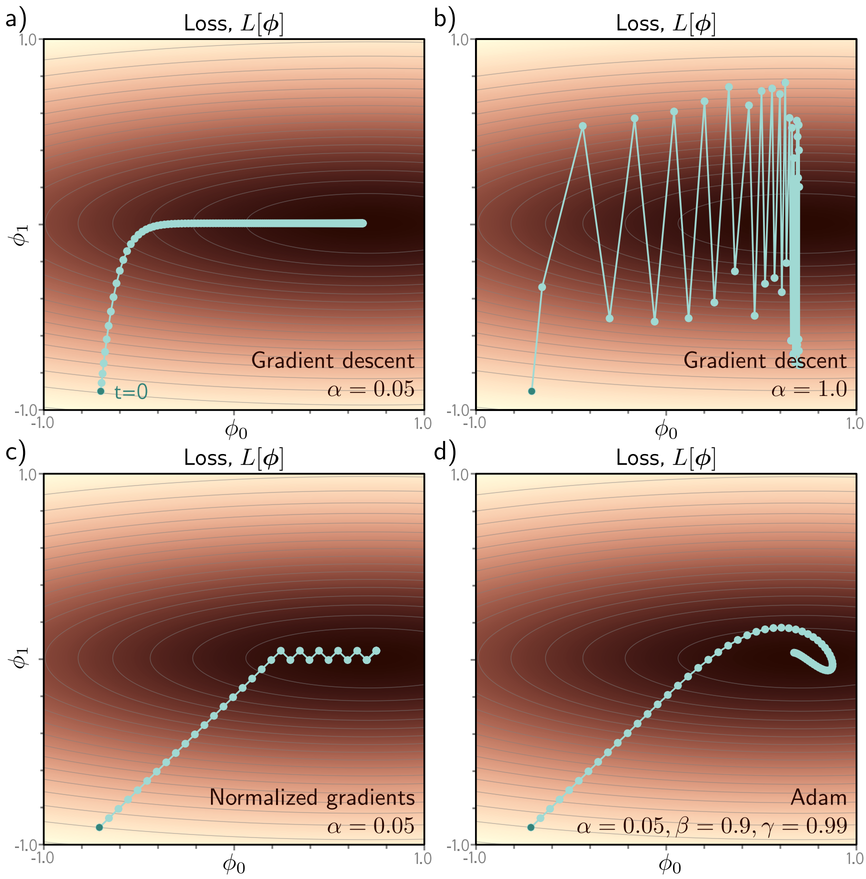


Figure from: Prince SJD. Understanding deep learning. MIT press, 2023.

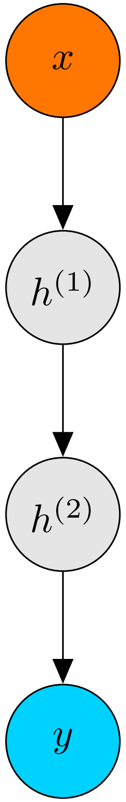
Hyperparameters

- **Hyperparameters** are distinct from the **model parameters ϕ**
- **Architecture** hyperparameters
  - Size of the hidden layers
  - Number of hidden layers
- **Training algorithm** hyperparameters
  - Choices of learning algorithm
  - Batch size
  - Learning rate (schedule)
- Next week will will talk about tuning them

Computing derivatives

- We use **(stochastic) gradient decent** to find $\arg\min_{\phi} L(\phi)$
  - Each step in the algorithm requires $\nabla_{\phi} L(\phi)$
- We use **backpropagation** to calculate gradients $\nabla_{\phi} L(\phi)$

Backpropagation: scalar architecture



- Consider a scalar only architecture

$$h^{(1)} = \sigma_1(w_1 x)$$
$$h^{(2)} = \sigma_2(w_2 x)$$
$$y = w_3 h^{(2)}$$

  and some loss function
  - We can calculate these values in a **forward pass**

- Using the chain rule, calculate derivative in a **backward pass**

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial w_3}$$

$$\frac{\partial L}{\partial h^{(2)}}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial w_2}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial h^{(1)}}\frac{\partial h^{(1)}}{\partial w_1}$$

Recall chain rule for

$z = f(y)$

and

$y = f(x)$

is

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

Parameter initialisation

- We random initialise the weights, e.g., $\phi_i \sim \mathcal{N}(0, \sigma^2)$
- How to choose σ^2?
  - If σ^2 is too **small**, the signal **vanishes** as it passes thought the network
  - If σ^2 is too **big**, the signal **grows** as it passes thought the network
- A reasonable **criterion** for **keeping the information** flow for all i, i'

$$\mathrm{Var}\left[h_i^{(\ell)}\right] = \mathrm{Var}\left[h_{i'}^{(\ell-1)}\right]$$

- Fr ReLU activation, this implies that

$$\sigma^2 = \frac{2}{D}$$

where D is the dimension of the layer

Today's exercises!

- A Python notbook (`2.1 FNN AutoDif Nanograd.ipynb`)
  - Implement AutoDiff yourself
- Three **problems** from to book on covered material

Thank you!