The Mathematics of Machine Learning 2025

Lecture 11

Tim van Erven

University of Amsterdam

Outline

Deep Learning / Neural Networks

1. Neural Networks
2. Fully Connected Layers
3. Activation Functions
4. Number of Parameters
5. Applications

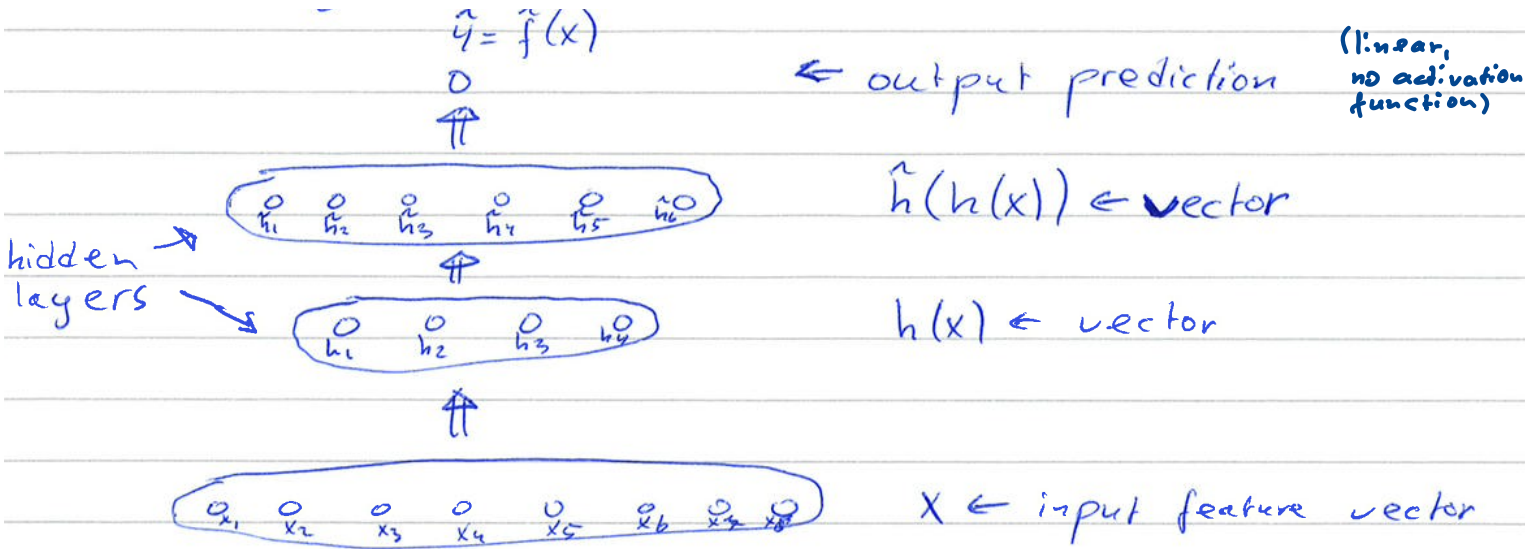
Optimizing Neural Networks

6. Stochastic Gradient Descent
7. SGD for Neural Networks
8. Backpropagation

Summary and Preview

Deep Learning / Neural Networks

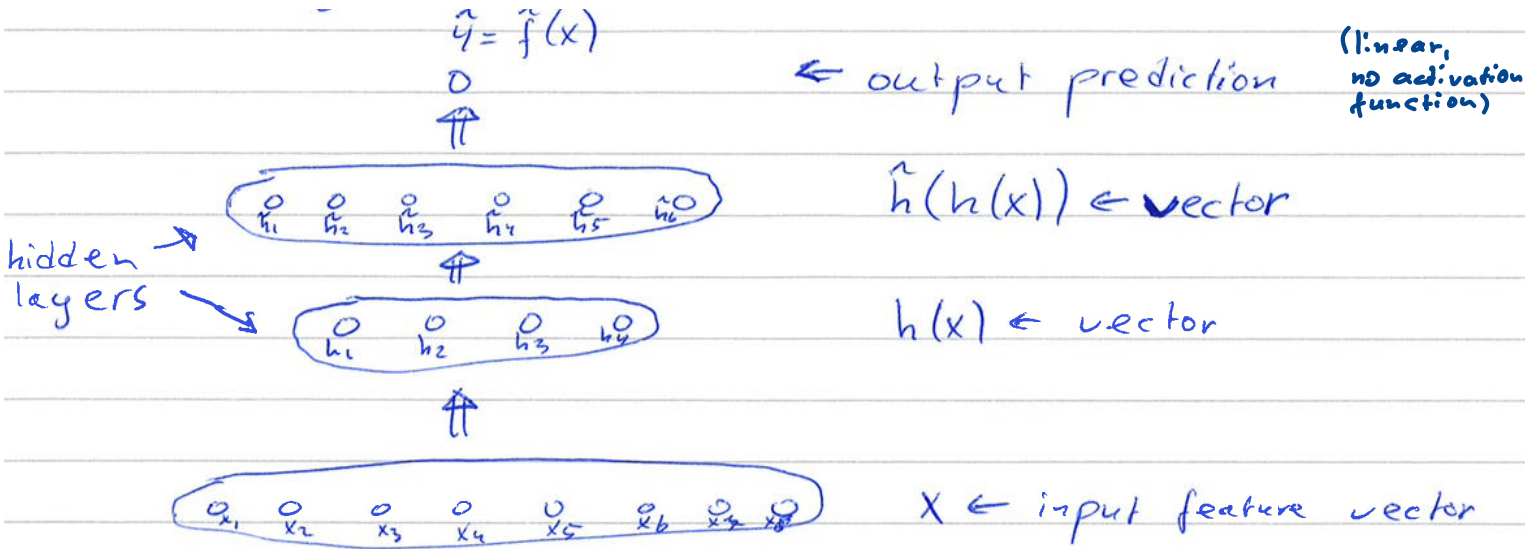
Neural networks are models that generalize the linear model to a series of transformations called **layers**:



- Map from last layer to prediction typically linear $\Rightarrow$ View $\tilde{h}(h(X))$ as **feature transformation that is learned from the data**.
- Contrast to SVMs, where a single feature transformation $h(X)$ maps to a very high-dimensional space, but is a fixed part of the model.

Deep Learning / Neural Networks

Neural networks are models that generalize the linear model to a series of transformations called **layers**:

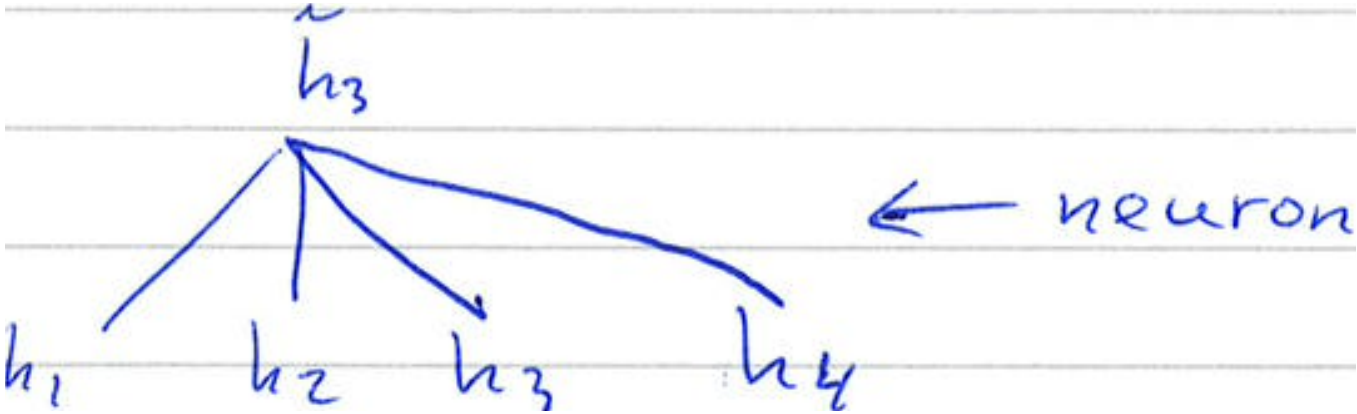


Remarks:

- Can have > 3 layers: "deep" learning
- Loss $L(Y, \hat{Y})$ can be anything. Common choices are logistic loss for classification, and squared error for regression.

Fully Connected Layers

Zooming in:



activation

$$\tilde{a}_3 = \sum_{i=1}^{4} w_{h_i \to \tilde{h}_3} h_i + b_{\tilde{h}_3}$$

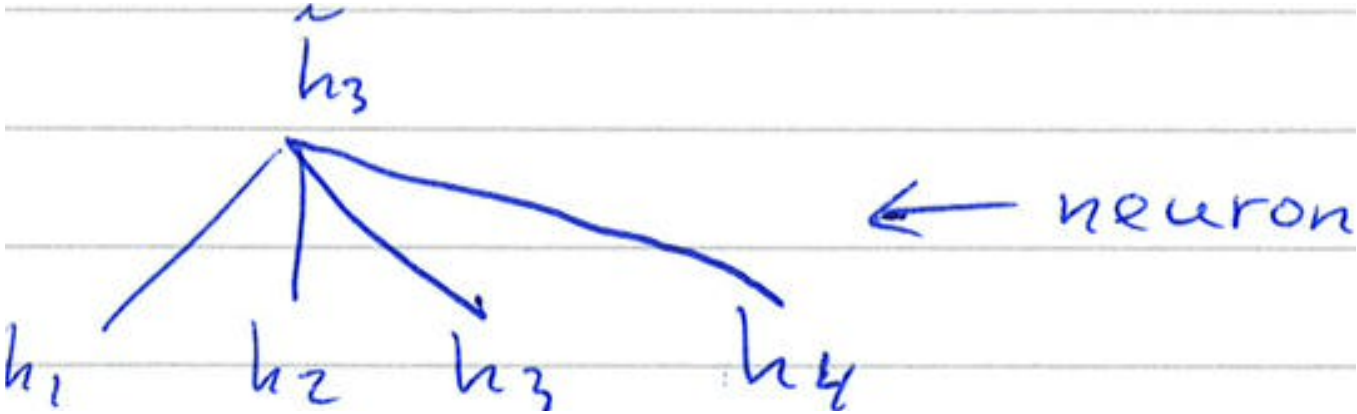
$$\tilde{h}_3 = \sigma(\tilde{a}_3)$$

activation function (non-linear)

Terminology note: w for "weights" and b for "bias"

Fully Connected Layers

Zooming in:



linear function with coefficients $w_{h_i \to \tilde{h}_3}$ and intercept $b_{\tilde{h}_3}$

activation

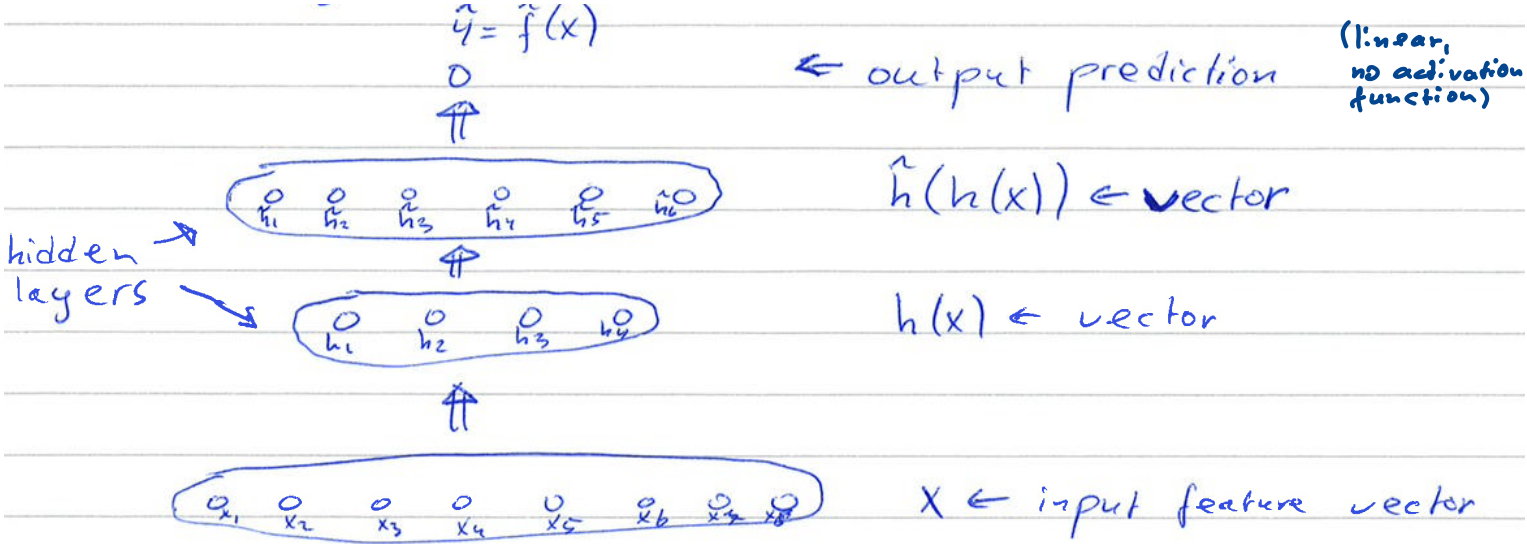
$$\tilde{a}_3 = \sum_{i=1}^{4} w_{h_i \to \tilde{h}_3} h_i + b_{\tilde{h}_3}$$

$$\tilde{h}_3 = \sigma(\tilde{a}_3)$$

activation function (non-linear)

Terminology note: w for "weights" and b for "bias"

Fully Connected Layers



In matrix form (with parameters $\widetilde{W}_{i,j} = w_{h_i \to \tilde{h}_j}$ and $\tilde{b}_j = b_{\tilde{h}_j}$):

$$\begin{pmatrix} \tilde{a}_1 \\ \vdots \\ \tilde{a}_6 \end{pmatrix} = \widetilde{W} \begin{pmatrix} h_1 \\ \vdots \\ h_4 \end{pmatrix} + \tilde{b} \qquad \text{(linear)}$$

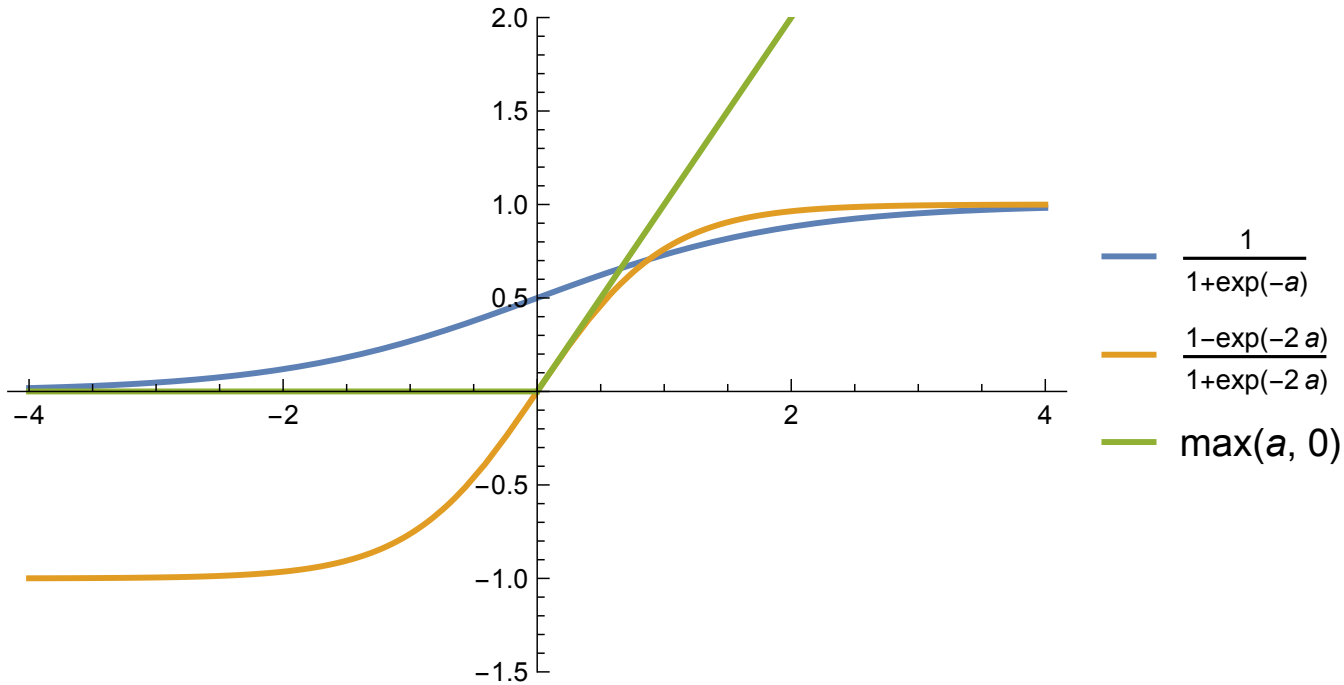
$$\begin{pmatrix} \tilde{h}_1 \\ \vdots \\ \tilde{h}_6 \end{pmatrix} = \begin{pmatrix} \sigma(\tilde{a}_1) \\ \vdots \\ \sigma(\tilde{a}_6) \end{pmatrix} \qquad \text{(non-linear)}$$

Activation Functions

sigmoid: $\sigma(a) = \dfrac{1}{1+e^{-a}}$

tanh: $\sigma(a) = \dfrac{1-e^{-2a}}{1+e^{-2a}} = 2\sigma_{\text{sigmoid}}(2a) - 1$

rectified linear unit (ReLu): $\sigma(a) = \max\{a, 0\}$



Activation Functions

$$\text{sigmoid:} \quad \sigma(a) = \frac{1}{1+e^{-a}}$$

$$\text{tanh:} \quad \sigma(a) = \frac{1-e^{-2a}}{1+e^{-2a}} = 2\sigma_{\text{sigmoid}}(2a) - 1$$

$$\text{rectified linear unit (ReLu):} \quad \sigma(a) = \max\{a, 0\}$$

- Sigmoid/tanh: historically first
- ReLu:
  - Currently most popular (by far)
  - Not differentiable at $a = 0$, but differentiable almost everywhere
  - Piece-wise linear: made optimization easier

Number of Parameters

Example: A network with the following fully-connected layers:

output:	1 output
layer 2:	100 hidden units
layer 1:	100 hidden units
inputs:	256×256 pixels for each of 3 colors (red, green, blue)

Number of Parameters

Example: A network with the following fully-connected layers:

output:	1 output
layer 2:	100 hidden units
layer 1:	100 hidden units
inputs:	256×256 pixels for each of 3 colors (red, green, blue)

Number of Parameters:

Input dimension: $256 \times 256 \times 3 = 196\,608$

Number of Parameters

Example: A network with the following fully-connected layers:

output:	1 output
layer 2:	100 hidden units
layer 1:	100 hidden units
inputs:	256×256 pixels for each of 3 colors (red, green, blue)

Number of Parameters:

Input dimension: $256 \times 256 \times 3 = 196\,608$

Number of parameters per layer:

layer 1:	$196\,608$	$\times 100$	$+100$
layer 2:	100	$\times 100$	$+100$
output:	100	$\times 1$	$+1$
total:		$19\,671\,101 \approx$	**20 million parameters!**

Number of Parameters

Example: A network with the following fully-connected layers:

output:	1 output
layer 2:	100 hidden units
layer 1:	100 hidden units
inputs:	256×256 pixels for each of 3 colors (red, green, blue)

Number of Parameters:

Input dimension: $256 \times 256 \times 3 = 196\,608$

Number of parameters per layer:

layer 1:	$196\,608$	$\times 100$	$+100$
layer 2:	100	$\times 100$	$+100$
output:	100	$\times 1$	$+1$
total:		$19\,671\,101 \approx$ **20 million parameters!**	

Discussion:

- Risk of overfitting!
- Optimization challenge: need very fast algorithms!
- We will see alternative layers with fewer (but still many) parameters later

Applications

Important applications of neural networks:

- Image recognition, image generation and other computer vision tasks
- Speech recognition and generation
- Natural language processing (ChatGPT, ...)
- Protein folding (Alpha-fold)
- Game playing (Go, chess, Atari games, StarCraft, ...)

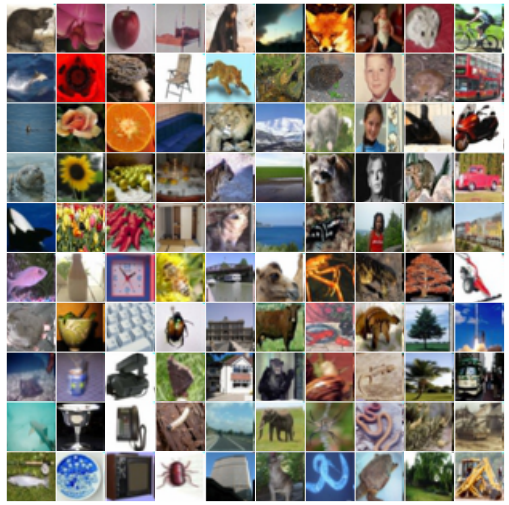


Figure 18.8 Examples of natural images. The `CIFAR-100` database consists of 100 color image classes, with 600 examples in each class (500 train, 100 test). Each image is $32 \times 32 \times 3$ (red, green, blue). Here we display a randomly chosen image from each class. The classes are organized by hierarchical structure, with 20 coarse levels and five subclasses within each. So, for example, the first five images in the first column are `aquatic mammals`, namely `beaver`, `dolphin`, `otter`, `seal` and `whale`.

Common properties of these applications: **huge data sets** with millions of data points available, which allows estimating very large number of parameters.

Limitations:

- Neural networks usually **not state of the art for smaller data sets** (less than 100 000 data points may still be "small").

Outline

Deep Learning / Neural Networks

1. Neural Networks
2. Fully Connected Layers
3. Activation Functions
4. Number of Parameters
5. Applications

Optimizing Neural Networks

6. Stochastic Gradient Descent
7. SGD for Neural Networks
8. Backpropagation

Summary and Preview

Stochastic Gradient Descent

$$F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{\beta}) \qquad \text{for some (convex) functions } f_1, \dots, f_N$$

$$\nabla F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(\boldsymbol{\beta})$$

Stochastic Gradient Descent

$$F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{\beta}) \qquad \text{for some (convex) functions } f_1, \dots, f_N$$

$$\nabla F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(\boldsymbol{\beta})$$

Pick one data point at random and do gradient descent step based on only that data point instead of the whole data:

Choose i_t randomly from $\{1, \dots, N\}$

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \eta_t \nabla f_{i_t}(\boldsymbol{\beta}_t)$$

Stochastic Gradient Descent

$$F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{\beta}) \qquad \text{for some (convex) functions } f_1, \dots, f_N$$

$$\nabla F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(\boldsymbol{\beta})$$

Pick one data point at random and do gradient descent step based on only that data point instead of the whole data:

$$\text{Choose } i_t \text{ randomly from } \{1, \dots, N\}$$

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \eta_t \nabla f_{i_t}(\boldsymbol{\beta}_t)$$

- $\nabla f_{i_t}(\boldsymbol{\beta}_t)$ is **unbiased** estimate of $\nabla F(\boldsymbol{\beta}_t)$:

$$\mathop{\mathbb{E}}_{i_t}[\nabla f_{i_t}(\boldsymbol{\beta}_t)] = \sum_{i_t=1}^{N} \frac{1}{N} \nabla f_{i_t}(\boldsymbol{\beta}_t) = \nabla F(\boldsymbol{\beta}_t).$$

- So we go in **right direction on average** with much less computation

Stochastic Gradient Descent

$$F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{\beta}) \qquad \text{for some (convex) functions } f_1, \dots, f_N$$

$$\nabla F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(\boldsymbol{\beta})$$

Minibatches with B data points:

$$F(\boldsymbol{\beta}) = \frac{1}{N/B}\sum_{i=1}^{N/B} f_i(\boldsymbol{\beta}) \text{ where } f_i(\boldsymbol{\beta}) = \frac{1}{B}\sum_{j=(i-1)B+1}^{iB} L(Y_j, f_{\boldsymbol{\beta}}(X_j))$$

Stochastic Gradient Descent

$$F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{\beta}) \qquad \text{for some (convex) functions } f_1, \dots, f_N$$

$$\nabla F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(\boldsymbol{\beta})$$

Minibatches with B data points:

$$F(\boldsymbol{\beta}) = \frac{1}{N/B}\sum_{i=1}^{N/B} f_i(\boldsymbol{\beta}) \text{ where } f_i(\boldsymbol{\beta}) = \frac{1}{B}\sum_{j=(i-1)B+1}^{iB} L(Y_j, f_{\boldsymbol{\beta}}(X_j))$$

- Interpolates between case above ($B = 1$) and non-stochastic gradient descent ($B = N$)

Stochastic Gradient Descent

$$F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{\beta}) \qquad \text{for some (convex) functions } f_1, \dots, f_N$$

$$\nabla F(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(\boldsymbol{\beta})$$

Minibatches with B data points:

$$F(\boldsymbol{\beta}) = \frac{1}{N/B}\sum_{i=1}^{N/B} f_i(\boldsymbol{\beta}) \text{ where } f_i(\boldsymbol{\beta}) = \frac{1}{B}\sum_{j=(i-1)B+1}^{iB} L(Y_j, f_{\boldsymbol{\beta}}(X_j))$$

- Interpolates between case above ($B = 1$) and non-stochastic gradient descent ($B = N$)
- Motivations:
  - Computational: modern hardware can compute $\nabla L(Y_j, f_{\boldsymbol{\beta}}(X_j))$ **in parallel** for m data points, so computing gradient on minibatch of size $B = m$ as fast as gradient for $B = 1$.
  - Statistical: larger B estimates gradient $\nabla F(\boldsymbol{\beta})$ with less variance

SGD for Neural Networks

- $\boldsymbol{\beta}$ is vector with all parameters (e.g. 20 million)
- Loss function $L(Y_j, f_{\boldsymbol{\beta}}(X_j))$ **not convex** as a function of $\boldsymbol{\beta}$, so can get stuck in local minima or in very flat regions
- Big open question for theory: why does it find good local minima in practice? $\Leftarrow$ optimization appears to work better when number of parameters is much larger than N!

SGD for Neural Networks

- $\boldsymbol{\beta}$ is vector with all parameters (e.g. 20 million)
- Loss function $L(Y_j, f_{\boldsymbol{\beta}}(X_j))$ **not convex** as a function of $\boldsymbol{\beta}$, so can get stuck in local minima or in very flat regions
- Big open question for theory: why does it find good local minima in practice? $\Leftarrow$ optimization appears to work better when number of parameters is much larger than N!

Minor deviations from SGD setup in practice:

- **Pick minibatches sequentially** instead of uniformly at random: $i_t = 1, 2, \ldots, N/B$, and restart when come to the end of the data. One pass through the data is called an **epoch**.
- $f_i(\boldsymbol{\beta}) = \sum_{j=(i-1)B+1}^{iB} L(Y_j, f_{\boldsymbol{\beta}}(X_j))$ without division by B! Equivalent if we divide η_t by B instead.

Backpropagation

How to compute gradients $\nabla_{\boldsymbol{\beta}} L(Y_i, f_{\boldsymbol{\beta}}(X_i))$?

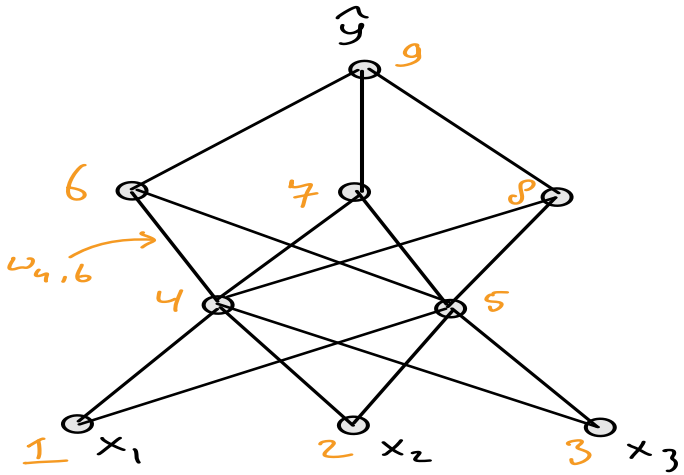
- 20 million parameters: only have time for a fixed number of computations per parameter!

Backpropagation

How to compute gradients $\nabla_{\boldsymbol{\beta}} L(Y_i, f_{\boldsymbol{\beta}}(X_i))$?

- 20 million parameters: only have time for a fixed number of computations per parameter!
- Solution: **Backpropagation**
  $\leftarrow$ use chain rule for derivatives many times: $\frac{\mathrm{d}a}{\mathrm{d}b} = \frac{\mathrm{d}a}{\mathrm{d}c}\frac{\mathrm{d}c}{\mathrm{d}b}$

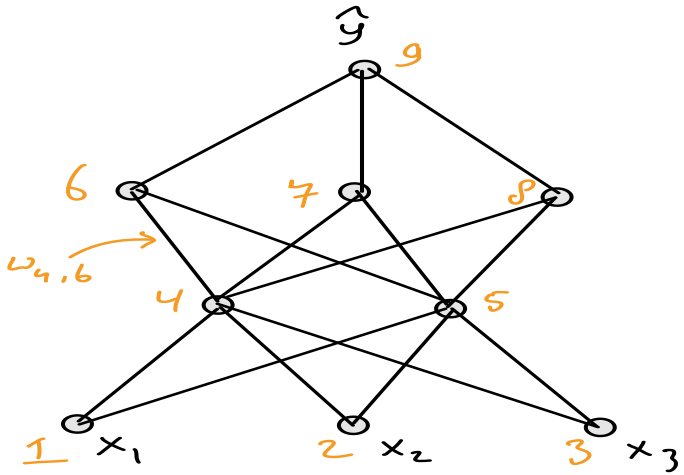
Backpropagation



$$\hat{Y} = f_{\boldsymbol{\beta}}(X) = a_9$$

$$o_m = \sigma(a_m) \qquad \text{for } m = 1, \ldots, 8$$

Backpropagation

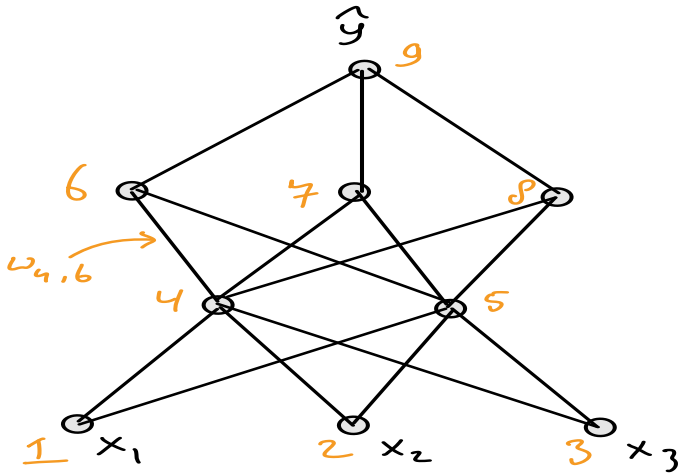


$$\hat{Y} = f_{\boldsymbol{\beta}}(X) = a_9$$

$$o_m = \sigma(a_m) \qquad \text{for } m = 1, \ldots, 8$$

1. $\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}\hat{Y}}$ is derivative of the loss

Backpropagation



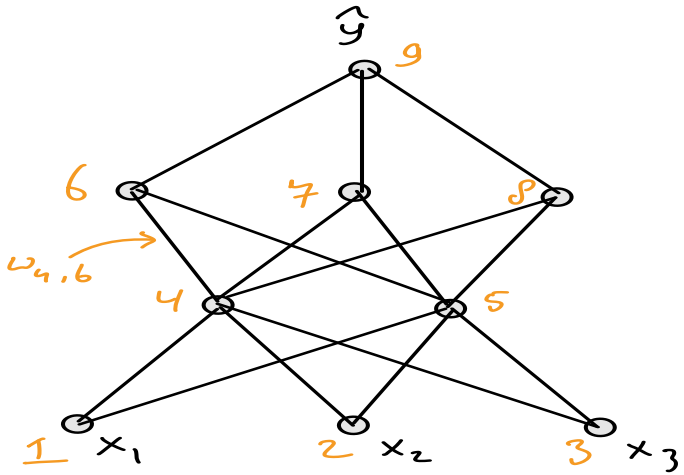
$$\hat{Y} = f_{\boldsymbol{\beta}}(X) = a_9$$

$$o_m = \sigma(a_m) \qquad \text{for } m = 1, \ldots, 8$$

1. $\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}\hat{Y}}$ is derivative of the loss
2. Suppose we have $\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}o_6}$. Then we can efficiently compute

$$\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}a_6} = \frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}o_6}\frac{\mathrm{d}o_6}{\mathrm{d}a_6} = \frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}o_6}\sigma'(a_6)$$

Backpropagation



$$\hat{Y} = f_{\boldsymbol{\beta}}(X) = a_9$$
$$o_m = \sigma(a_m) \qquad \text{for } m = 1, \ldots, 8$$

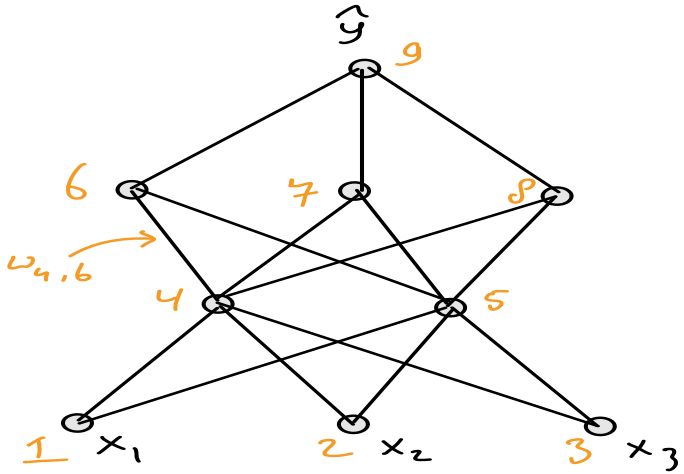
1. $\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}\hat{Y}}$ is derivative of the loss
2. Suppose we have $\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}o_6}$. Then we can efficiently compute

$$\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}a_6} = \frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}o_6}\frac{\mathrm{d}o_6}{\mathrm{d}a_6} = \frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}o_6}\sigma'(a_6)$$

3. Suppose we have e.g. $\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}a_6}$, then we can efficiently compute derivatives w.r.t. all incoming weights to unit 6:

$$\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}w_{4,6}} = \frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}a_6}\frac{\mathrm{d}\sum_{i=4}^{5} w_{i,6}o_i + b_6}{\mathrm{d}w_{4,6}} = \frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}a_6}o_4$$

Backpropagation



$$\hat{Y} = f_{\boldsymbol{\beta}}(X) = a_9$$
$$o_m = \sigma(a_m) \qquad \text{for } m = 1, \ldots, 8$$

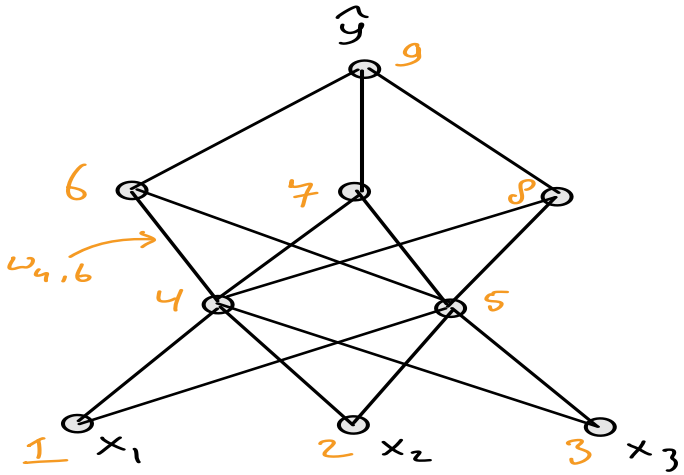
1. $\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}\hat{Y}}$ is derivative of the loss
2. Suppose we have $\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}o_6}$. Then we can efficiently compute
3. S

Summary so far: if we could efficiently compute $\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}o_m}$ for all $m = 1, \ldots, 8$, then we could efficiently compute derivatives w.r.t. all weights.

derivatives w.r.t. all incoming weights to unit 6:

$$\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}w_{4,6}} = \frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}a_6} \frac{\mathrm{d}\sum_{i=4}^{5} w_{i,6} o_i + b_6}{\mathrm{d}w_{4,6}} = \frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}a_6} o_4$$

Backpropagation

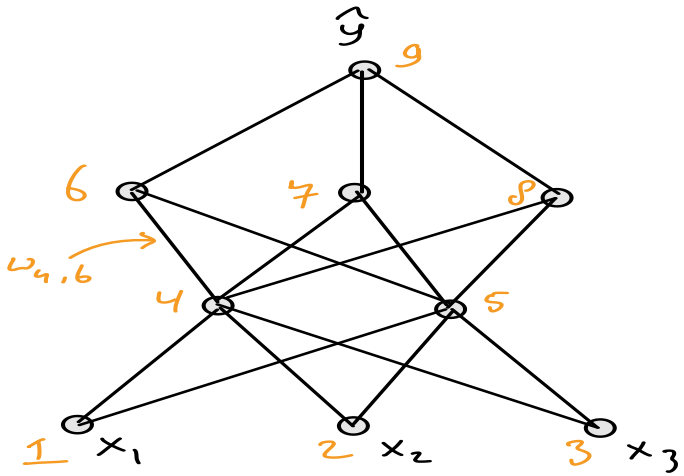


$$\hat{Y} = f_{\boldsymbol{\beta}}(X) = a_9$$

$$o_m = \sigma(a_m) \qquad \text{for } m = 1, \ldots, 8$$

4. Work backwards from output to inputs to compute derivatives w.r.t. each o_m if we already have these derivatives higher up in the network.

Backpropagation

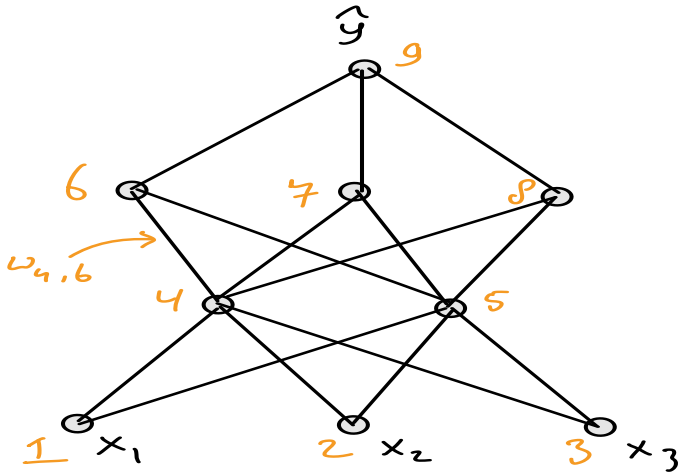


$$\hat{Y} = f_{\boldsymbol{\beta}}(X) = a_9$$
$$o_m = \sigma(a_m) \qquad \text{for } m = 1, \ldots, 8$$

4. Work backwards from output to inputs to compute derivatives w.r.t. each o_m if we already have these derivatives higher up in the network.

Vector chain rule: $$\frac{\mathrm{d}L(Y, \hat{Y})}{\mathrm{d}o_5} = \sum_{i=6}^{8} \frac{\mathrm{d}L(Y, \hat{Y})}{\mathrm{d}o_i} \frac{\mathrm{d}o_i}{\mathrm{d}o_5}$$

Backpropagation



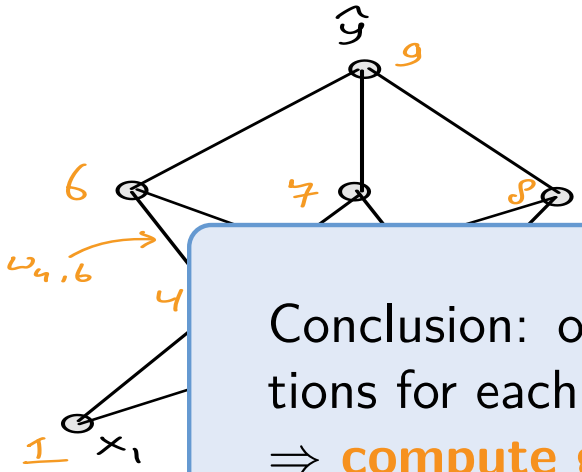
$$\hat{Y} = f_{\boldsymbol{\beta}}(X) = a_9$$
$$o_m = \sigma(a_m) \qquad \text{for } m = 1, \ldots, 8$$

4. Work backwards from output to inputs to compute derivatives w.r.t. each o_m if we already have these derivatives higher up in the network.

Vector chain rule: $$\frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}o_5} = \sum_{i=6}^{8} \frac{\mathrm{d}L(Y,\hat{Y})}{\mathrm{d}o_i} \frac{\mathrm{d}o_i}{\mathrm{d}o_5}$$

$$\frac{\mathrm{d}o_6}{\mathrm{d}o_5} = \frac{\mathrm{d}o_6}{\mathrm{d}a_6}\frac{\mathrm{d}a_6}{\mathrm{d}o_5} = \sigma'(a_6)\frac{\mathrm{d}\sum_{i=4}^{5} w_{i,6}o_i + b_6}{\mathrm{d}o_5} = \sigma'(a_6)w_{i,5}$$

Backpropagation



$1, \dots, 8$

4. V

e

n

w.r.t.

V

Conclusion: only need to do fixed number of operations for each element of the network
$\Rightarrow$ **compute gradient vector quickly**, in number of computation steps that scales **linearly in the number of parameters**

- Programming all these derivatives is very challenging
- Modern software libraries do all this differentiation automatically under the hood!

$$\frac{\mathrm{d}o_6}{\mathrm{d}o_5} = \frac{\mathrm{d}o_6}{\mathrm{d}a_6}\frac{\mathrm{d}a_6}{\mathrm{d}o_5} = \sigma'(a_6)\frac{\mathrm{d}\sum_{i=4} w_{i,6}o_i + b_6}{\mathrm{d}o_5} = \sigma'(a_6)w_{i,5}$$

Outline

Deep Learning / Neural Networks

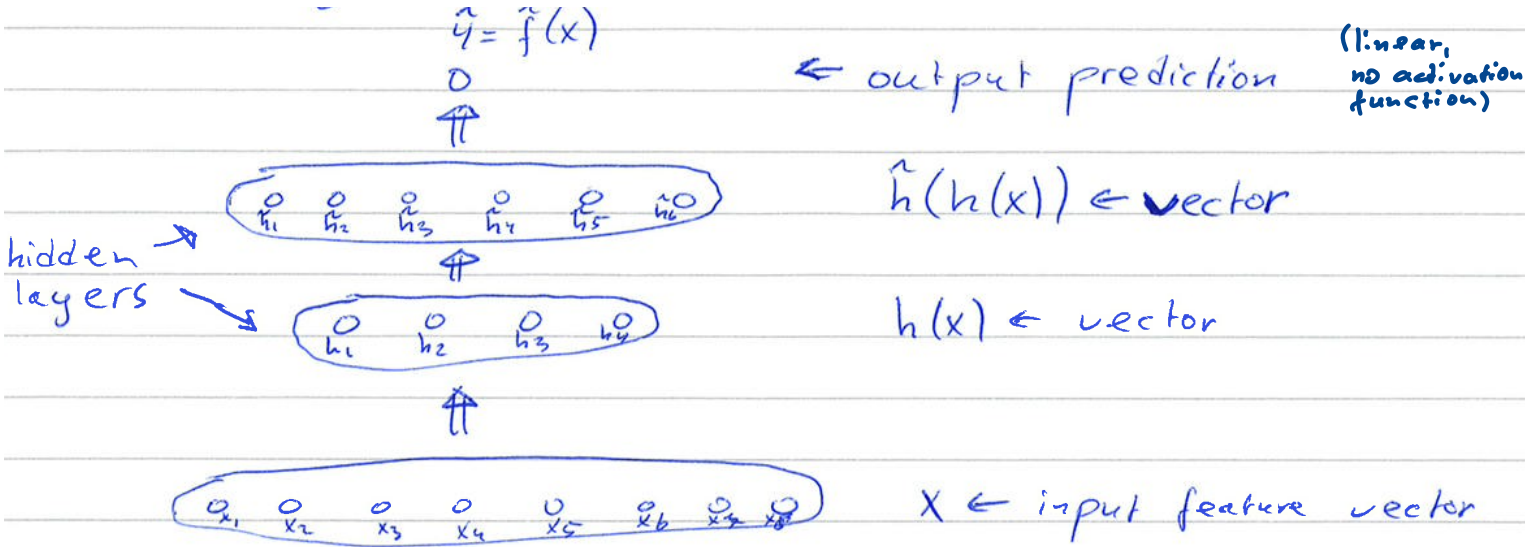
1. Neural Networks
2. Fully Connected Layers
3. Activation Functions
4. Number of Parameters
5. Applications

Optimizing Neural Networks

6. Stochastic Gradient Descent
7. SGD for Neural Networks
8. Backpropagation

Summary and Preview

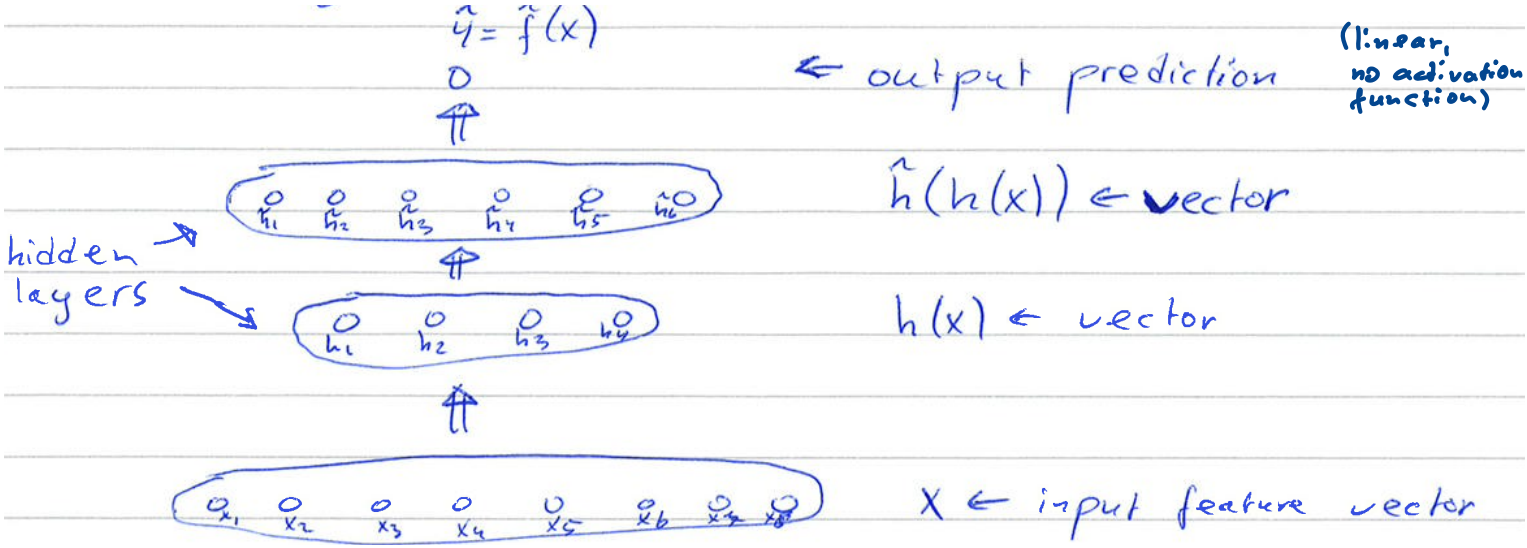
Summary and Preview



Summary:

- Neural networks consist of a **sequence of layers** that each transform the output of the previous layer
- Their number of parameters is typically very large
- These are optimized efficiently using stochastic gradient descent
- Optimization challenges:
  - Non-convexity: open question why SGD still often finds good local minima
  - Computing gradients: **backpropagation** does this in time linear in the number of parameters

Summary and Preview



Preview for next lecture:

1. **Convolutional layers:**
   - Assume some parameters have same value ⇒ greatly reduces number of parameters
   - Important for images and audio
2. **Transformers:**
   - Important usage: predict next word in a sequence of words (e.g. ChatGPT)
   - Learn which previous words in sequence are most relevant for prediction