4. Deep learning

Outline

From linear to non-linear models

- In linear regression, the prediction

$$h(x;\theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n,$$

  is a linear (strictly speaking, affine) function of the feature vector x.

- In logistic regression,

$$\log \frac{P(y=1|x;\theta)}{1-P(y=1|x;\theta)} = h(x;\theta)$$

- Such linear models are simple and easy to interpret, but not flexible enough to capture complex patterns in various applications.

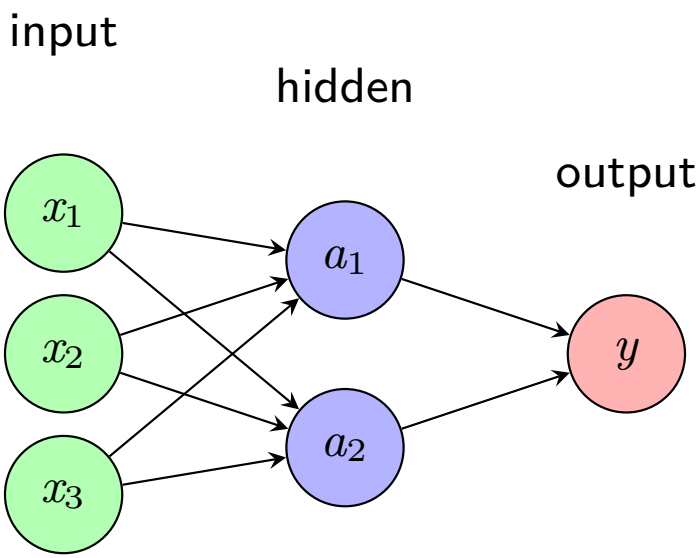
- It is desirable to make $h(x;\theta)$ a non-linear function of x.

Non-linear models

- Many choices of non-linear functions for $h(x;\theta)$, e.g.:
  - Polynomial regression
  - Kernel methods
  - Decision trees, random forests, boosting
  - Neural networks
- Models based on neural networks have proven to be very effective.
- Besides neural network architectures, various computing techniques are developed to train them.
- The combinations of neural network architectures and computing techniques are often referred to as **deep learning**.

Outline

Neural networks



- values of hidden units

$$a_1 = \mathrm{ReLU}(w_{11}^{[1]} x_1 + w_{12}^{[1]} x_2 + w_{13}^{[1]} x_3 + b_1^{[1]})$$
$$a_2 = \mathrm{ReLU}(w_{21}^{[1]} x_1 + w_{22}^{[1]} x_2 + w_{23}^{[1]} x_3 + b_2^{[1]})$$

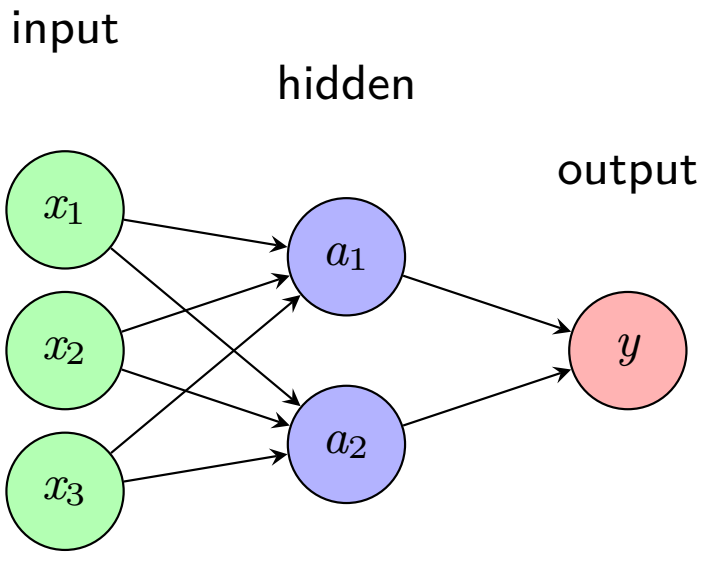
- value of output unit
  - regression:

$$y = w_1^{[2]} a_1 + w_2^{[2]} a_2 + b^{[2]}$$

  - classification:

$$y = \mathrm{Sigmoid}(w_1^{[2]} a_1 + w_2^{[2]} a_2 + b^{[2]})$$

Neural networks in matrix notation



- values of hidden units

$$a = \mathrm{ReLU}(W^{[1]}x + b^{[1]})$$

- value of output unit
  - regression:

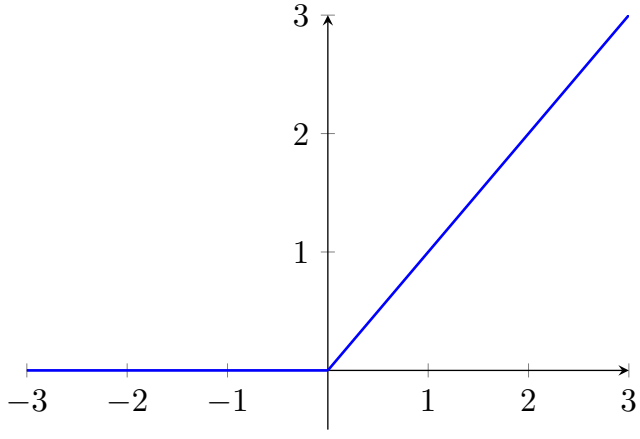
$$y = W^{[2]}a + b^{[2]}$$

  - classification:

$$y = \mathrm{Sigmoid}(W^{[2]}a + b^{[2]})$$

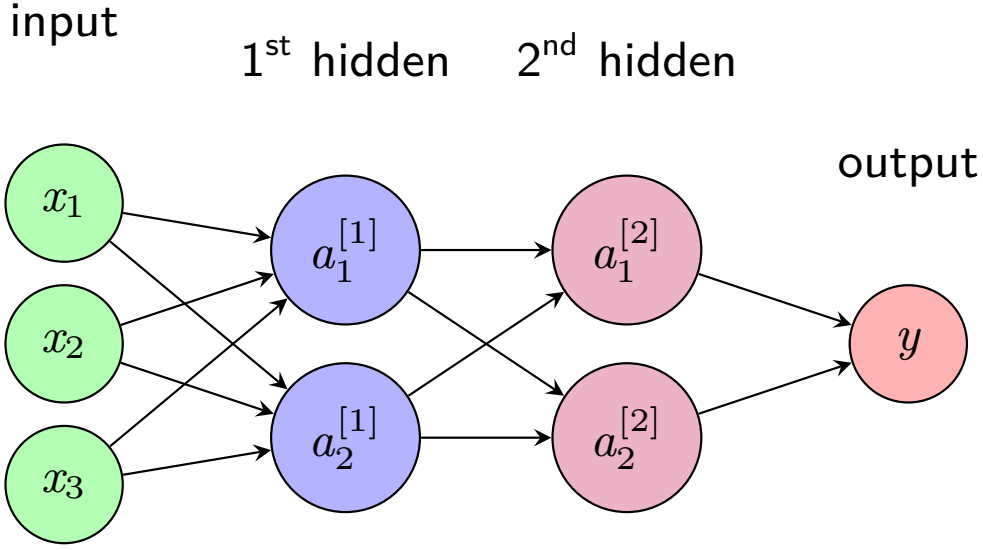
Activation functions

- Used to introduce non-linearity to the neural network
- $\mathrm{ReLU}(x) = \max(0, x)$



- Other activation functions: sigmoid, tanh, leaky ReLU, ELU, etc
- See `https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions` for more

Neural networks with multiple hidden layers



- values of hidden units

$$a^{[1]} = \mathrm{ReLU}(\, W^{[1]} x + b^{[1]})$$
$$a^{[2]} = \mathrm{ReLU}(\, W^{[2]} a^{[1]} + b^{[2]})$$

- value of output unit
  - regression:

$$y = W^{[3]} a^{[2]} + b^{[3]}$$

  - classification:

$$y = \mathrm{Sigmoid}(\, W^{[3]} a^{[2]} + b^{[3]})$$

Neural networks with multiple hidden layers

- input layer: x
- hidden layers:

$$
\begin{aligned}
a^{[1]} &= \mathrm{ReLU}(\, W^{[1]} x + b^{[1]}) \\
a^{[2]} &= \mathrm{ReLU}(\, W^{[2]} a^{[1]} + b^{[2]}) \\
&\vdots \\
a^{[L]} &= \mathrm{ReLU}(\, W^{[L]} a^{[L-1]} + b^{[L]})
\end{aligned}
$$

- output layer:
  - regression:

$$y = W^{[L+1]} a^{[L]} + b^{[L+1]}$$

  - classification:

$$y = \mathrm{Sigmoid}(\, W^{[L+1]} a^{[L]} + b^{[L+1]})$$

Multi-layer perceptrons

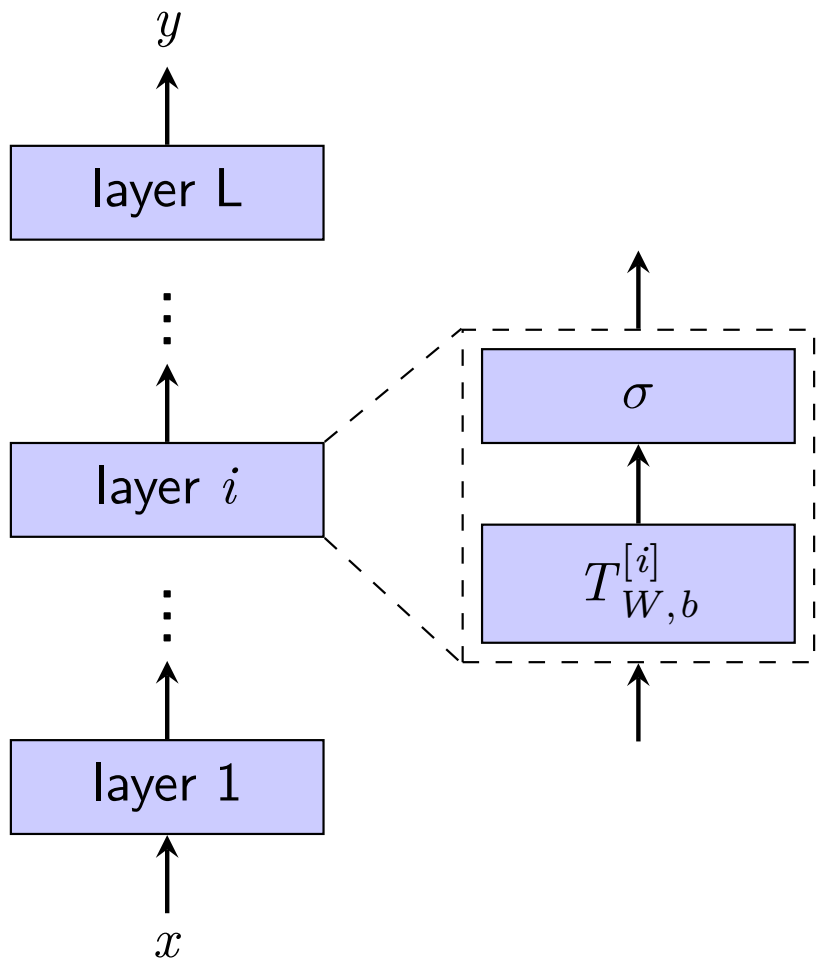
- Such neural network architectures are often referred to as **multi-layer perceptrons (MLPs)**.
- A layer corresponds to the transformation of

$$z_{\text{out}} = \sigma(Wz_{\text{in}} + b) = \sigma(T_{W,b}(z))$$

where σ is an activation function, W and b are the weight and bias parameters, respectively, and $T_{W,b}$ is the affine transformation $T_{W,b}(z) = Wz + b$.

- Each layer has its own weight and bias parameters.

Multi-layer perceptrons



Outline

Residue connections

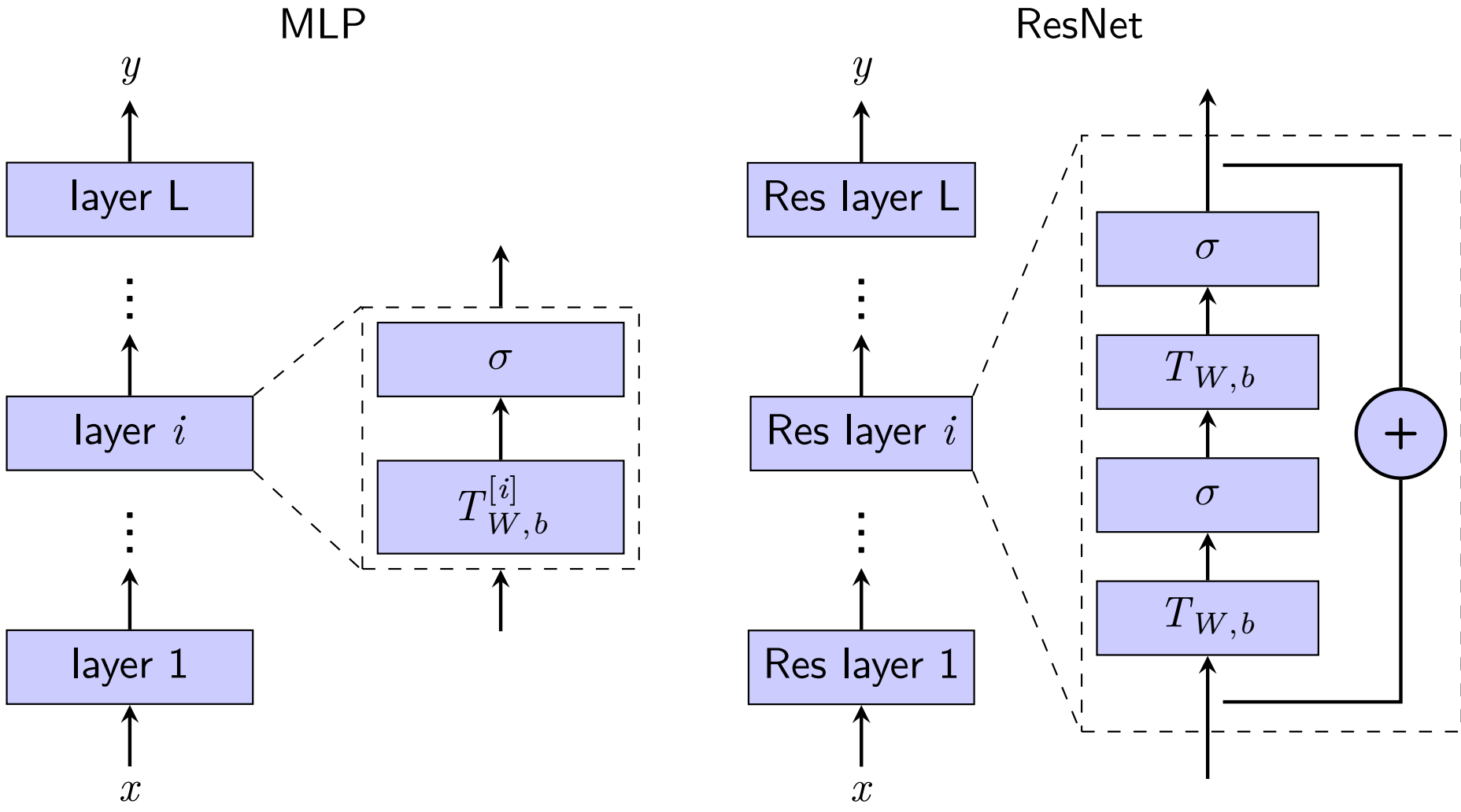
- An influential architecture that makes it easier to train networks with many layers
- A residue layer

$$\mathrm{Res}(z) = z + \sigma(T(\sigma(T(z))))$$

- A residue network is a sequence of residue layers

$$\mathrm{ResNet(z)} = T(\mathrm{Res}(\ldots \mathrm{Res}(z)))$$

MLP and Residue network



Layer normalization

- Maps a vector z to a more normalized vector $\mathrm{LN}(z)$

$$\mathrm{LN}(z) = \beta + \gamma \cdot \frac{z - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}} = \begin{bmatrix} \beta + \gamma \cdot \frac{z_1 - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}} \\ \beta + \gamma \cdot \frac{z_2 - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}} \\ \vdots \\ \beta + \gamma \cdot \frac{z_n - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}} \end{bmatrix}$$

- Both $\hat{\mu}$ and $\hat{\sigma}^2$ depend on the input z

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} z_i \text{ and } \hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} (z_i - \hat{\mu})^2$$

- β and γ are learnable parameters

Convolutional layer

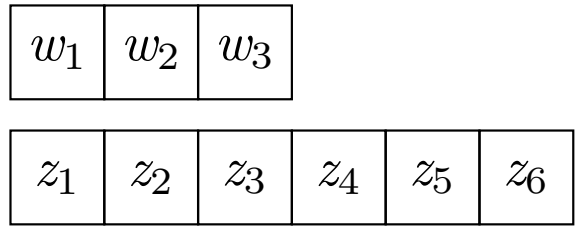
- particularly useful for computer vision tasks
- applies a convolution operation to the input
- defined for inputs of any dimensions
- captures local patterns that are translation invariant

Convolutional layer in 1D

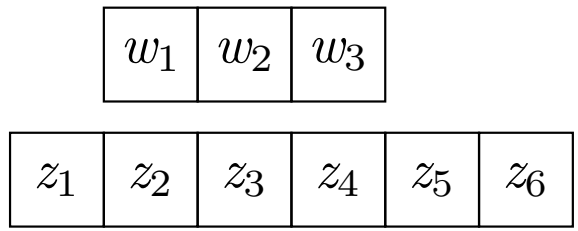
- input $z = [z_1, z_2, \ldots, z_n]$
- filter/kernel $w = [w_1, w_2, \ldots, w_k]$ with $k < n$
- output $y = [y_1, y_2, \ldots, y_{n-k+1}]$
- $y_i = \sum_{j=1}^{k} w_j z_{i+j-1}$

Example of convolutional layer in 1D

- input $z = [z_1, z_2, \ldots, z_6]$, filter/kernel $w = [w_1, w_2, w_3]$
- output $y = [y_1, y_2, y_3, y_4]$
- $y_1 = w_1 z_1 + w_2 z_2 + w_3 z_3$



- $y_2 = w_1 z_2 + w_2 z_3 + w_3 z_4$



- the output dimension is different from the input dimension

Example of convolutional layer in 1D with zero padding

- input $z = [z_1, z_2, \ldots, z_6]$, filter/kernel $w = [w_1, w_2, w_3]$
- output $y = [y_1, y_2, \ldots, y_6]$
- $y_1 = w_1 z_0 + w_2 z_1 + w_3 z_2$

w_1	w_2	w_3					
$z_0 = 0$	z_1	z_2	z_3	z_4	z_5	z_6	$z_7 = 0$

- $y_6 = w_1 z_5 + w_2 z_6 + w_3 z_7$

					w_1	w_2	w_3
$z_0 = 0$	z_1	z_2	z_3	z_4	z_5	z_6	$z_7 = 0$

- the output dimension is the same as the input dimension

Convolution as matrix multiplication

- convolution as matrix multiplication

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} w_2 & w_3 & 0 & 0 & 0 & 0 \\ w_1 & w_2 & w_3 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & w_3 \\ 0 & 0 & 0 & 0 & w_1 & w_2 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \end{bmatrix}$$

- sparse matrix with shared weights
- often use multiple filters to capture different patterns

Softmax layer

- used for multi-class classification
- maps a vector z to a probability distribution

$$\mathrm{Softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

- the output is a vector of probabilities
- has no learnable parameters

Pooling layer

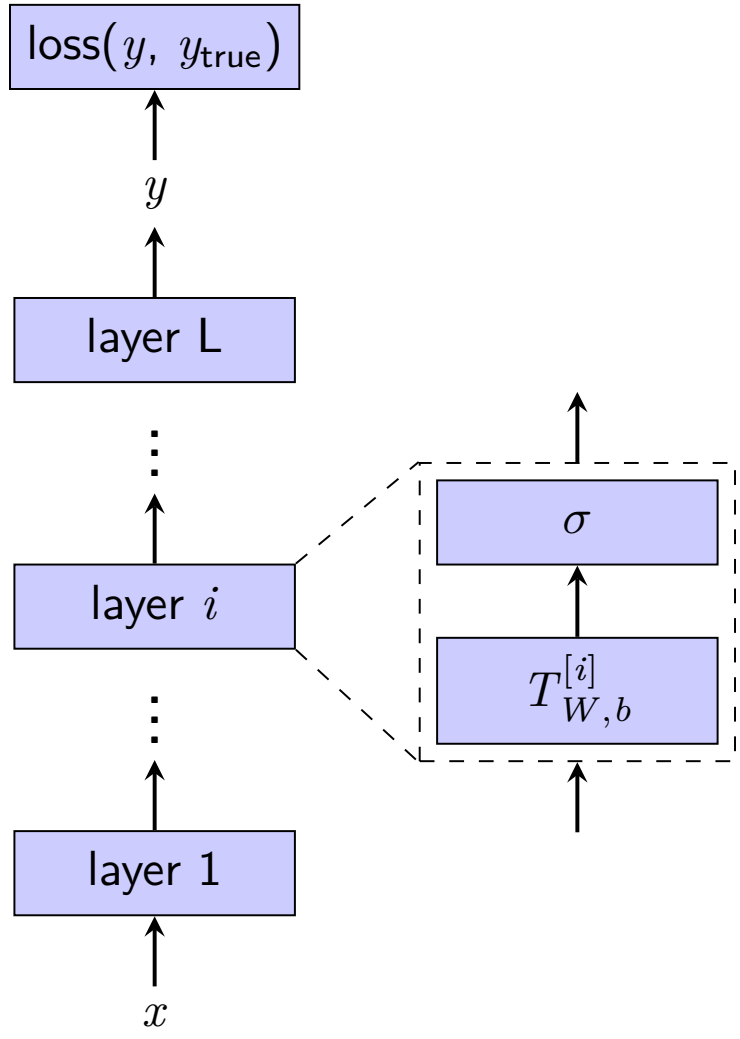
- reduces the spatial dimensions of the input
- often used after convolutional layers
- max pooling: takes the maximum value in a region
- average pooling: takes the average value in a region
- reduces the number of parameters and computation
- widely used in computer vision tasks

Outline

Compute the loss and its gradient

- training neural networks requires computing the gradient of the loss function with respect to parameters
- use the chain rule to compute the gradient
- also known as the *backpropagation* algorithm
- libraries like JAX and PyTorch provide automatic differentiation to compute the gradient

Compute the loss and its gradient



Stochastic gradient descent

- neural networks often have many layers and parameters
- expensive to compute the loss function and its gradient on the entire dataset that are required by gradient descent
- use stochastic gradient descent (SGD) instead
- update the parameters based on the gradient computed on a mini-batch of data

Stochastic gradient descent

Algorithm 1 Stochastic gradient descent for neural networks

input: neural network model f_θ, loss function $\mathcal{L}$, learning rate η, dataset $\mathcal{D}$, number of epochs T
initialize model parameters θ randomly
for epoch $t = 1$ to T **do**
    shuffle dataset $\mathcal{D}$ randomly
    for each mini-batch $\mathcal{B} \subset \mathcal{D}$ **do**
        compute gradient: $\nabla_\theta \mathcal{L}(\mathcal{B}, \theta)$
        update parameters: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\mathcal{B}, \theta)$
    end for
end for
output: trained model parameters θ

Stochastic gradient descent

- variations of SGD: vanilla SGD, momentum, RMSprop, Adam, etc
- advanced optimizers often perform better than vanilla SGD
- the adam optimizer is widely used and is a good default choice
- see the following link for more
  `https://pytorch.org/docs/stable/optim.html#algorithms`