Photo by Alina Grubnyak on Unsplash

Introduction to Deep Learning

Why Deep Learning

✓ A neural network layer is a relatively simple component

✓ Easy to scale up by stacking layers

✓ Generality: can approximate any function

✓ Works well for high dimensions

✓ Available hardware optimization

✓ Strong community (and open-sourced code)

X Computationally expensive

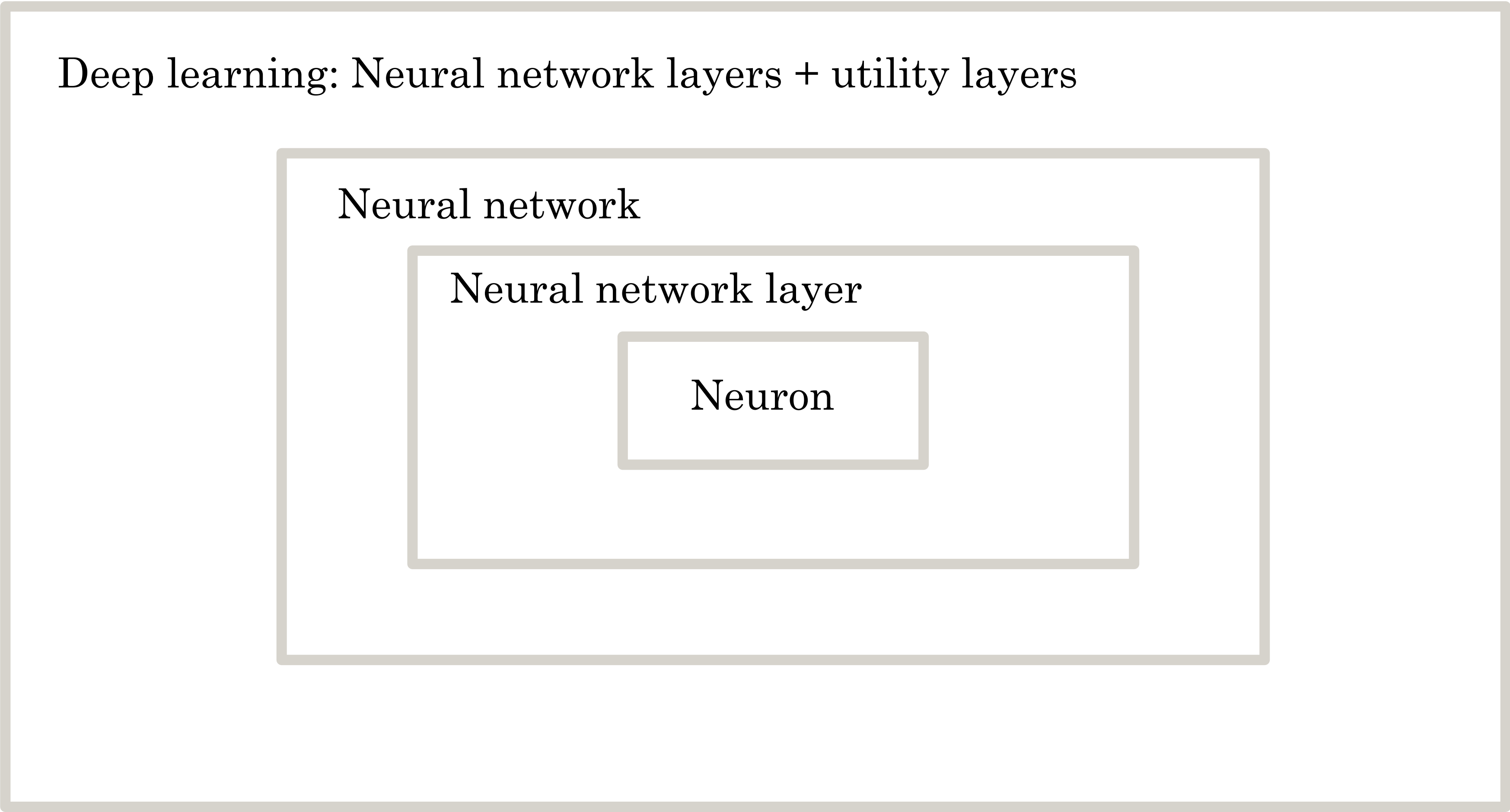
X Can be hard to finetune

X Hard to interpret (black box)

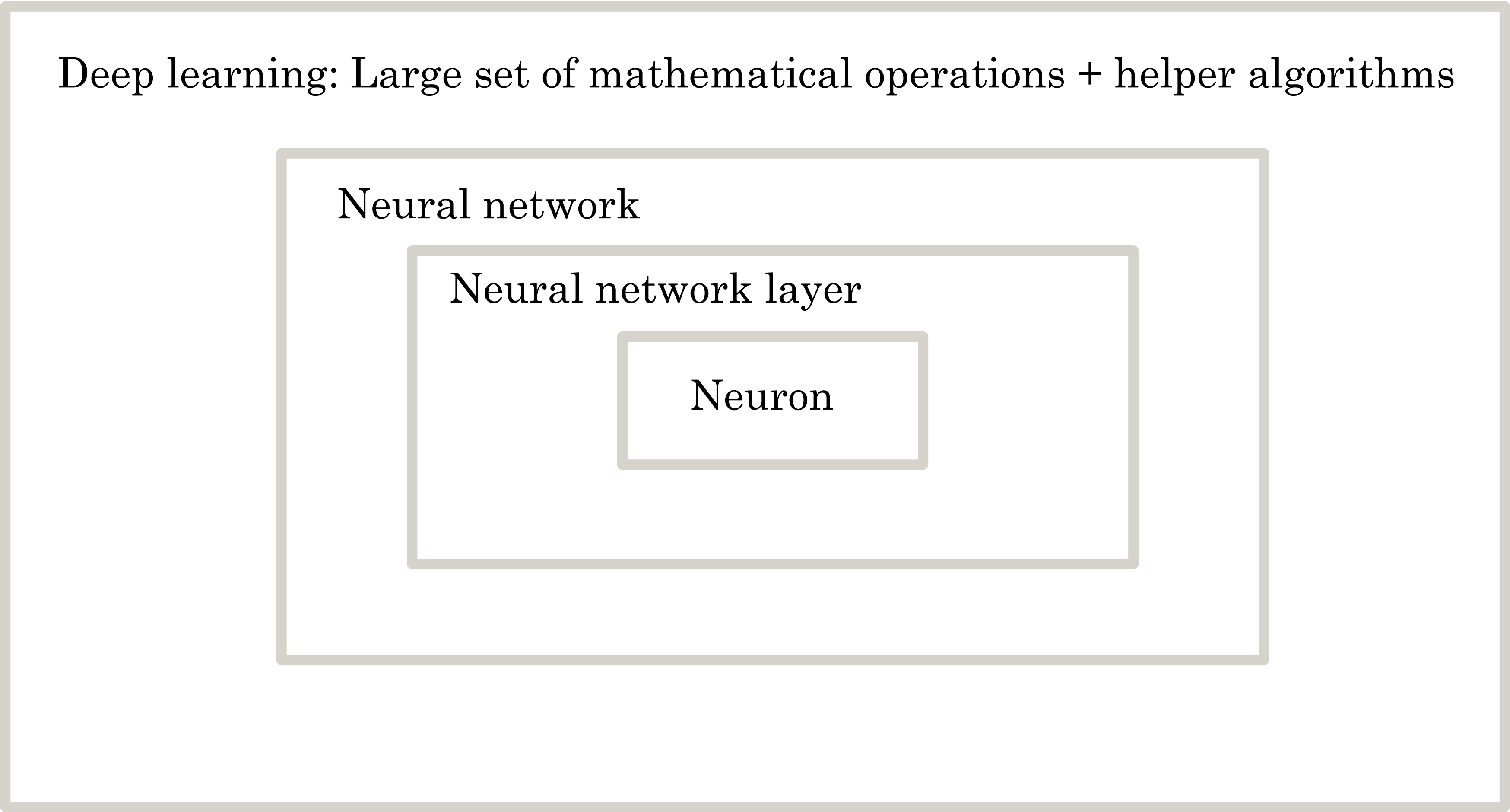
Yann LeCun

Wikimedia (retrieved 2023-04-08)

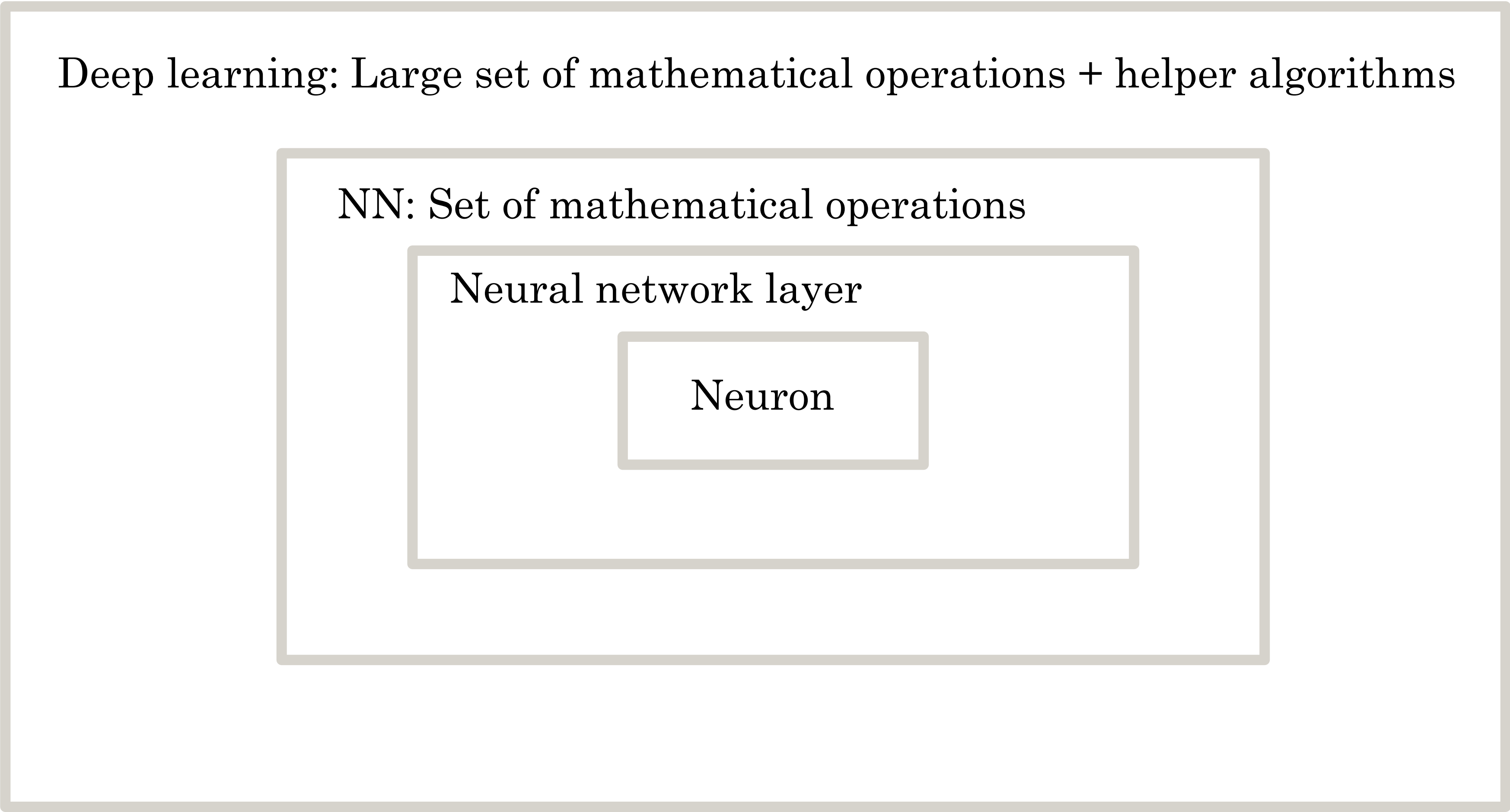
Deep learning components



Deep learning components



Deep learning components



Deep learning components

Deep learning: Large set of mathematical operations + helper algorithms

NN: Set of mathematical operations

NN layer: Set of non-linear functions

Neuron

Deep learning components

Deep learning: Large set of mathematical operations + helper algorithms

NN: Set of mathematical operations

NN layer: Set of non-linear functions

Neuron: non-linear function

Neuron

- A neuron is a function: $z = f(x, w) = f(\sum_i w_i x_i)$

Neuron

- A neuron is a <u>function</u>: z $= f(x, w) = f(\sum_i w_i x_i)$

Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)

Neuron

- A neuron is a function: $z = f(x, w) = f(\sum_i w_i x_i)$
- Activation functions:
  - Sigmoid

$$f(x) = \frac{e^x}{e^x + 1}$$

Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)

Neuron

- A neuron is a function: $z = f(x, w) = f(\sum_i w_i x_i)$
- Activation functions:
  - Sigmoid $$f(x) = \frac{e^x}{e^x + 1}$$
  - ReLU $$f(x) = \begin{cases} x \text{ if } x > 0 \\ 0 \end{cases}$$

Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)

Neuron

- A neuron is a <u>function</u>: $z = f(x,w) = f(\sum_i w_i x_i)$
- Activation functions:
  - Sigmoid $$f(x) = \frac{e^x}{e^x + 1}$$
  - ReLU $$f(x) = \begin{cases} x \ if \ x > 0 \\ 0 \end{cases}$$
  - GELU $$f(x) = x + P(X \leq x)$$

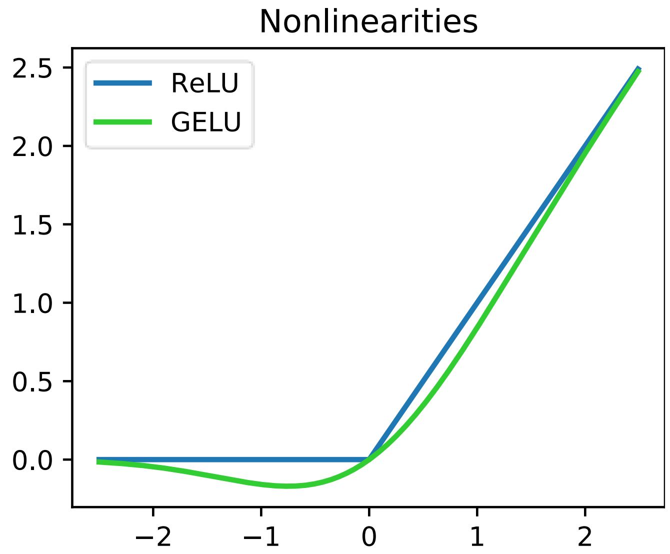
Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)

Neuron

- A neuron is a function: $z = f(x, w) = f(\sum_i w_i x_i)$
- Activation functions:
  - Sigmoid $$f(x) = \frac{e^x}{e^x + 1}$$
  - ReLU $$f(x) = \begin{cases} x \; if \; x > 0 \\ 0 \end{cases}$$
  - GELU $$f(x) = x + P(X \leq x)$$

CDF of the normal distribution

Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)



Wikimedia (retrieved 2023-06-25)

Neuron

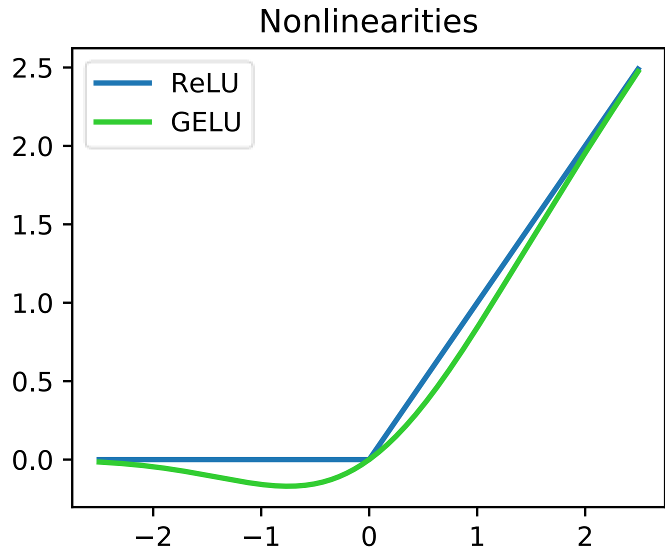
- In summary, a neuron is a non-linear function:

Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)

$$z = f(\sum_i w_i x_i)$$

$$z_{ReLU} = f(\sum_i w_i x_i) = \begin{cases} \sum_i w_i x_i & if \quad \sum_i w_i x_i > 0 \\ 0 & \end{cases}$$

$$z_{GELU} = \sum_i w_i x_i + P(X \leq \sum_i w_i x_i)$$



Wikimedia (retrieved 2023-06-25)

Neuron

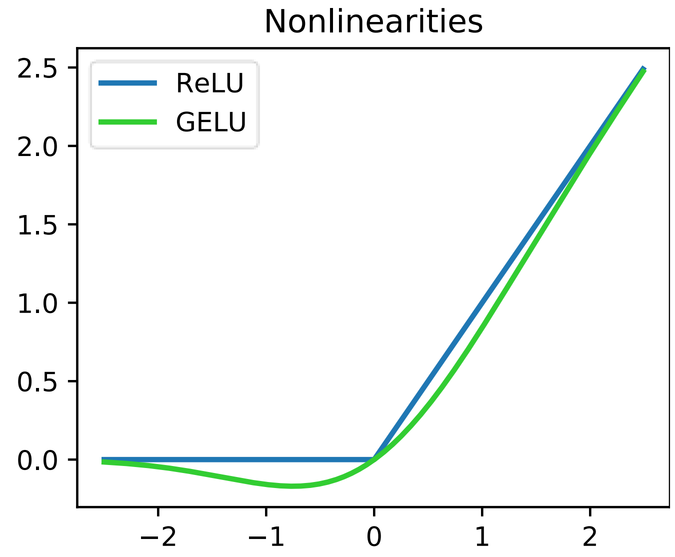
- In summary, a neuron is a non-linear function:

$$\text{Z} \qquad z = f(\sum_i w_i x_i)$$

$$z_{ReLU} = f(\sum_i w_i x_i) = \begin{cases} \sum_i w_i x_i & if \quad \sum_i w_i x_i > 0 \\ & 0 \end{cases}$$

$$z_{GELU} = \sum_i w_i x_i + P(X \leq \sum_i w_i x_i)$$

Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)



Wikimedia (retrieved 2023-06-25)

Neural network layer

- In summary, a neuron is a non-linear function:

$$z = f\left(\sum_{i}^{4} w_i x_i\right)$$

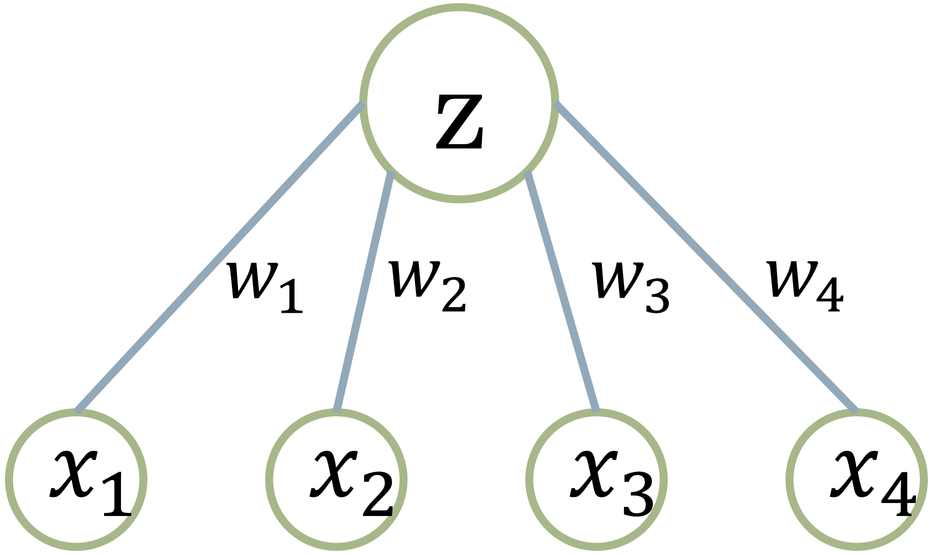


Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)

Neural network layer

- In summary, a neuron is a non-linear function

$$z = f(\sum_{i}^{4} w_i x_i)$$

- A NN layer is a set of non-linear functions

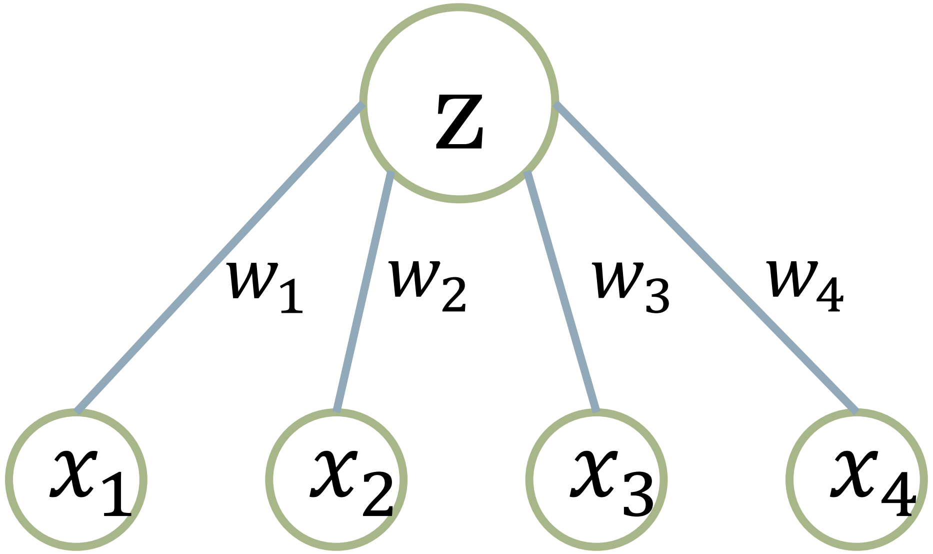


Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)

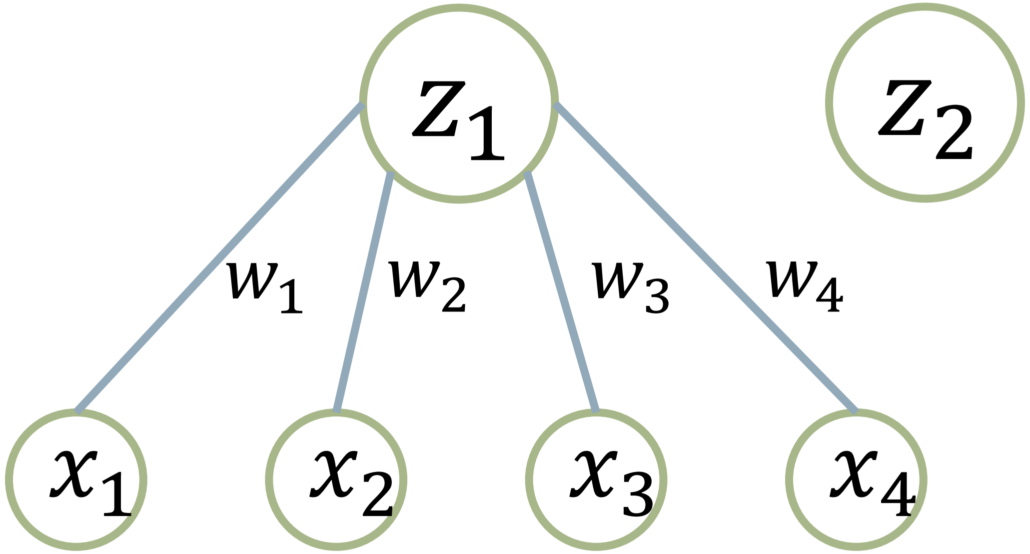
Neural network layer

- In summary, a neuron is a non-linear function

$$z_1 = f(\sum_{i}^{4} w_i x_i)$$

- A NN layer is a set of non-linear functions

Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)



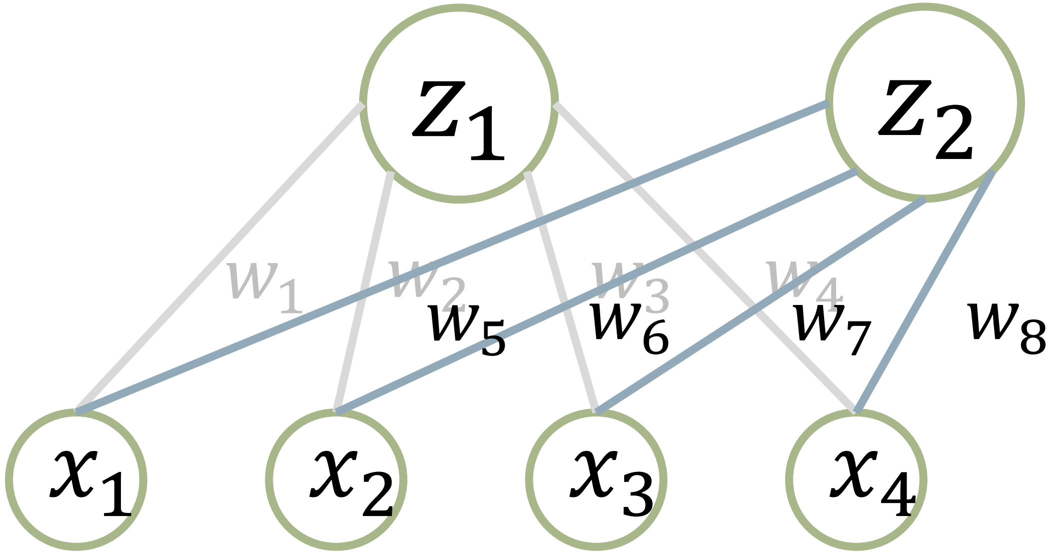
Neural network layer

- In summary, a neuron is a non-linear function

$$z_1 = f(\sum_i^4 w_i x_i)$$

- A NN layer is a set of non-linear functions

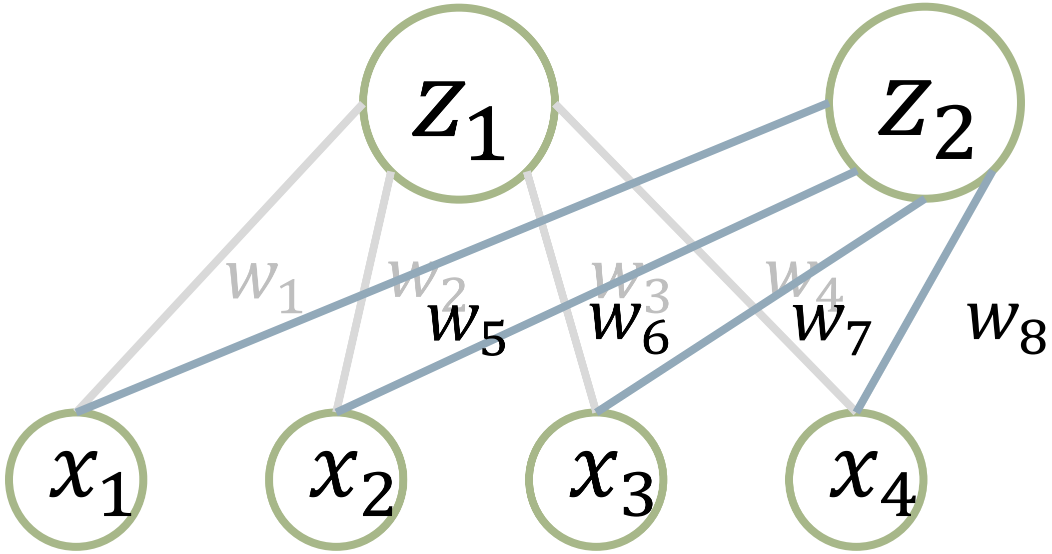
Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)



Neural network

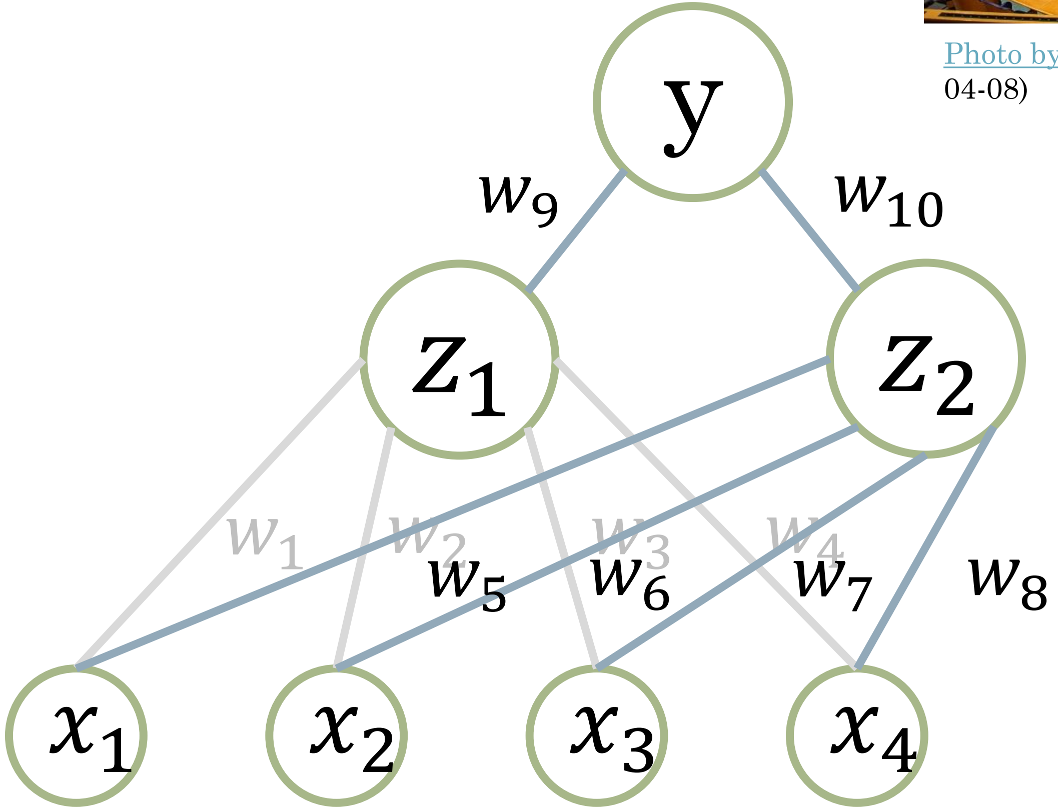
- A Neural Network is a set of NN layers = set of mathematical operations

Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)



Neural network

- A Neural Network is a set of NN layers = set of mathematical operations

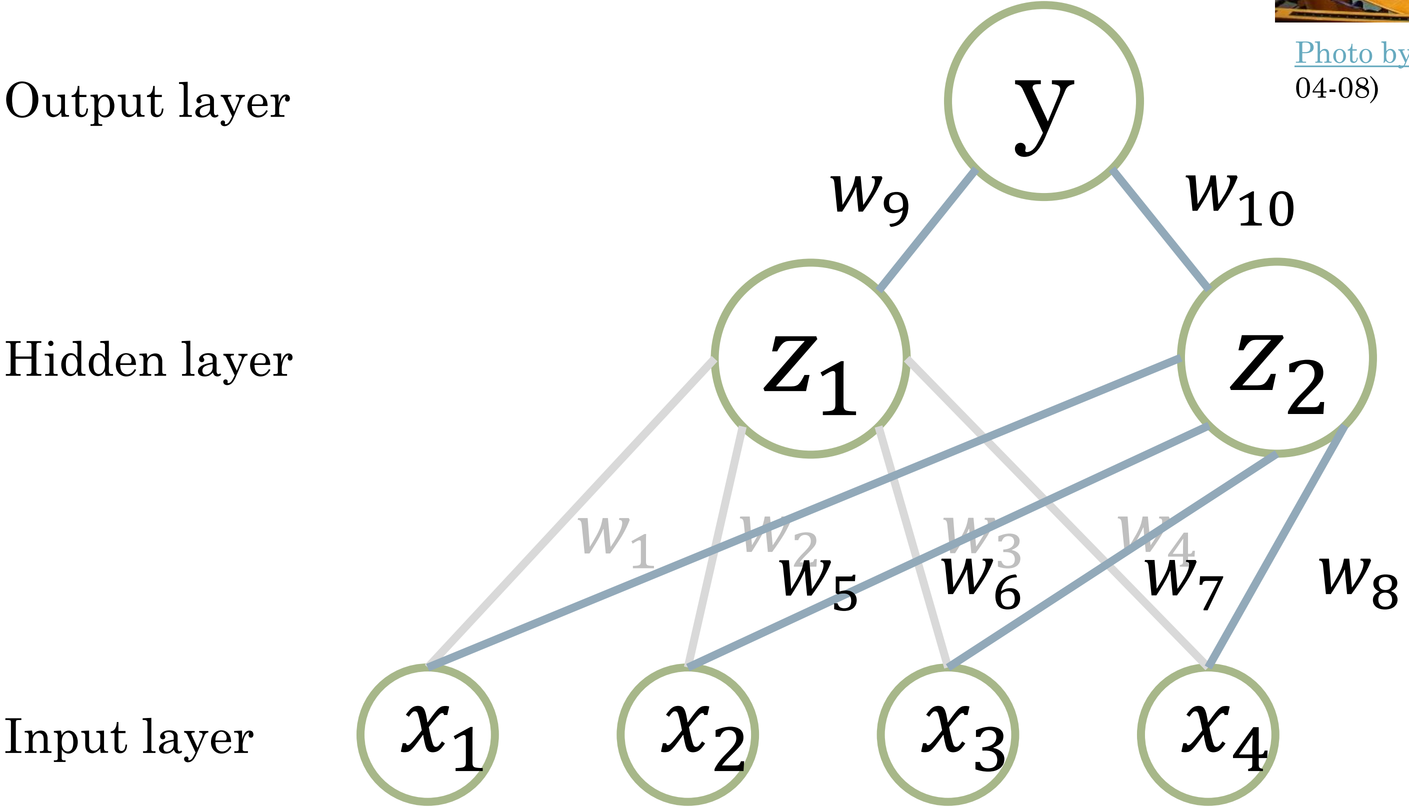


[Photo by Jan van der Wolf from Pexels](#) (retrieved 2023-04-08)

Neural network

- A Neural Network is a set of NN layers = set of mathematical operations

Photo by Jan van der Wolf from Pexels (retrieved 2023-04-08)

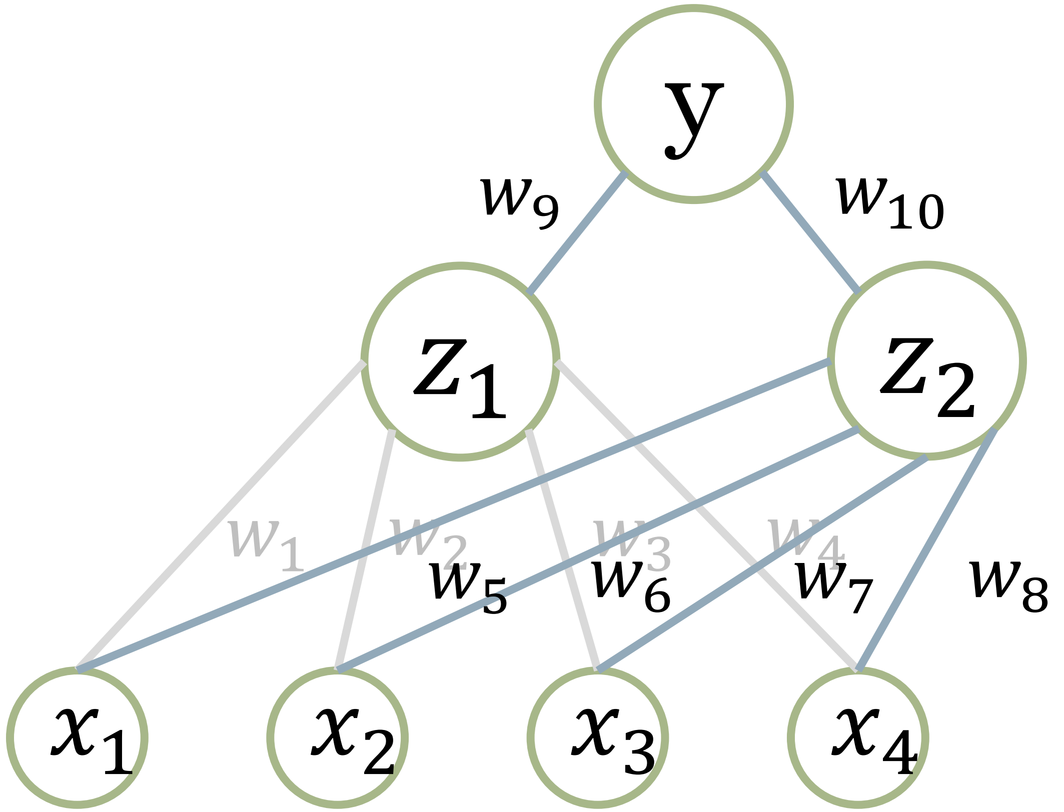


NN Generality

Learning / training

$$E_B(w) = \sum_{i \in B_k} |y_i - \hat{y}(w)_i|$$

- Goal: find the **weights *w*** that **minimize** the **error** function.



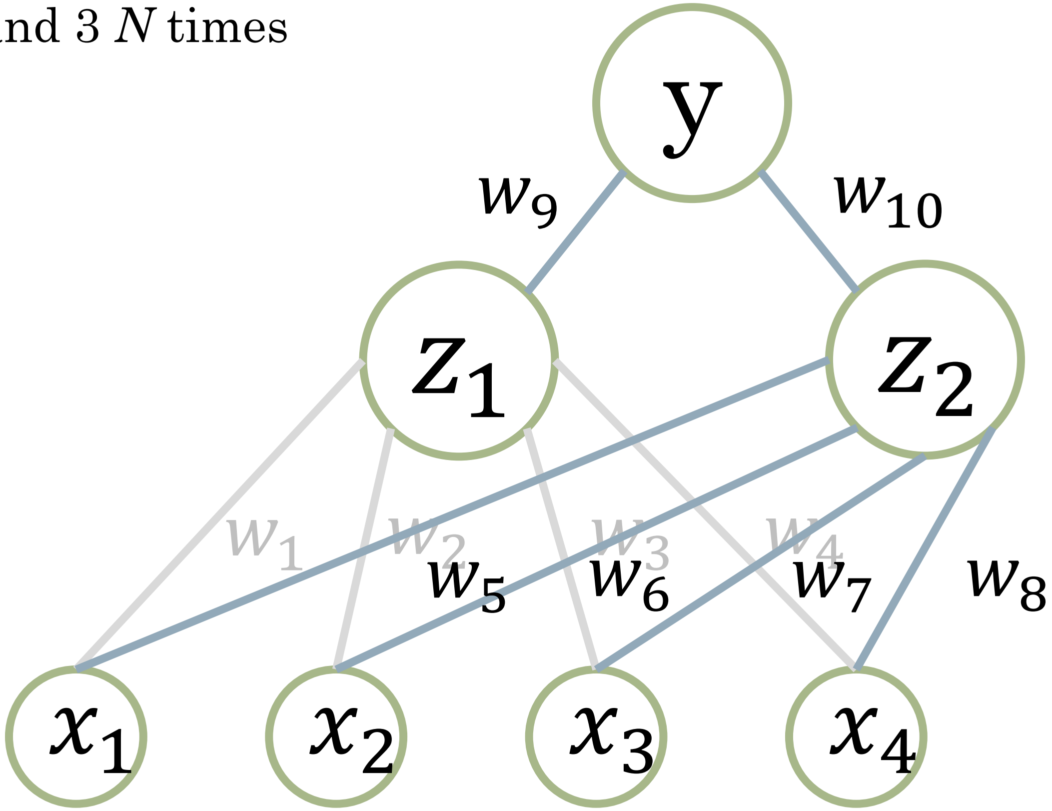
Learning / training

1. Initialize the weights randomly
2. Calculate the gradient of the error function
3. Update weights

Repeat steps 2 and 3 *N* times

SGD

1. $w_{j,0} = random$
2. $v_{j,t} = L\nabla_{w_j} E_B(w_{j,t})$
3. $w_{j,t+1} = w_{j,t} - v_{j,t}$



Introduction to Deep Learning

Learning / training

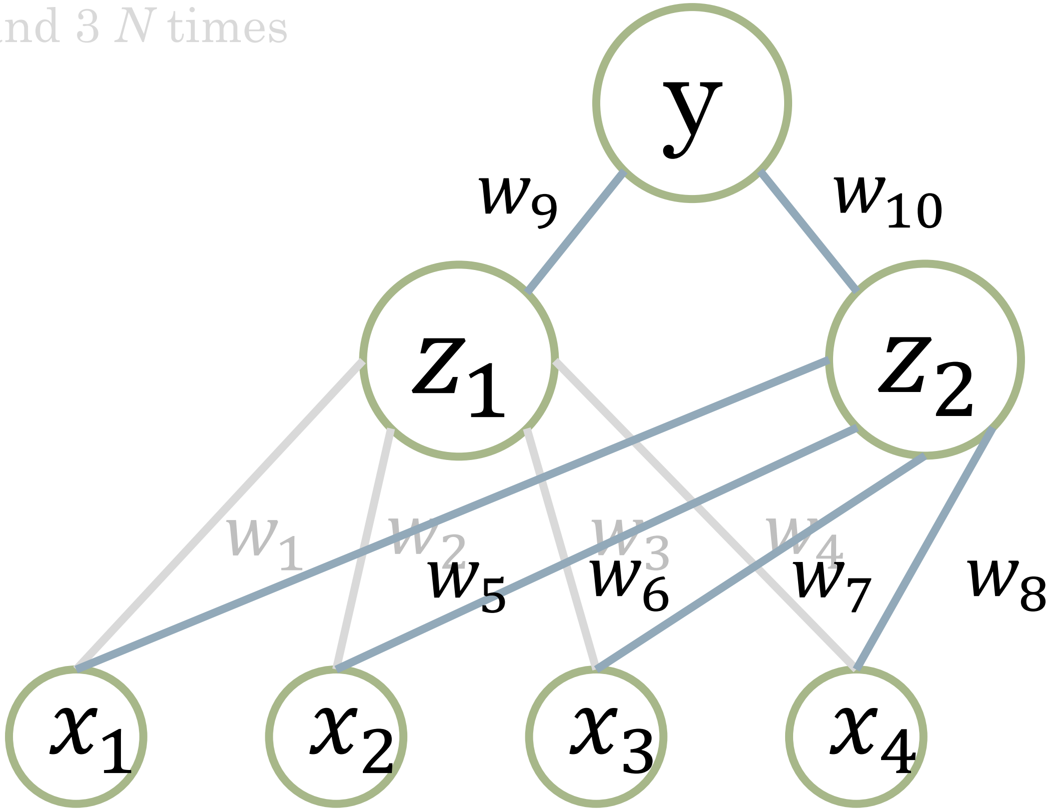
1. Initialize the weights randomly
2. Calculate the gradient of the error function
3. Update weights

Repeat steps 2 and 3 N times

SGD

1. $w_{j,0} = random$
2. $v_{j,t} = L\nabla_{w_j} E_B(w_{j,t})$
3. $w_{j,t+1} = w_{j,t} - v_{j,t}$

?



Backpropagation

SGD

1. $w_{j,0} = random$
2. $v_{j,t} = L\nabla_{w_j} E_B(w_{j,t})$
3. $w_{j,t+1} = w_{j,t} - v_{j,t}$

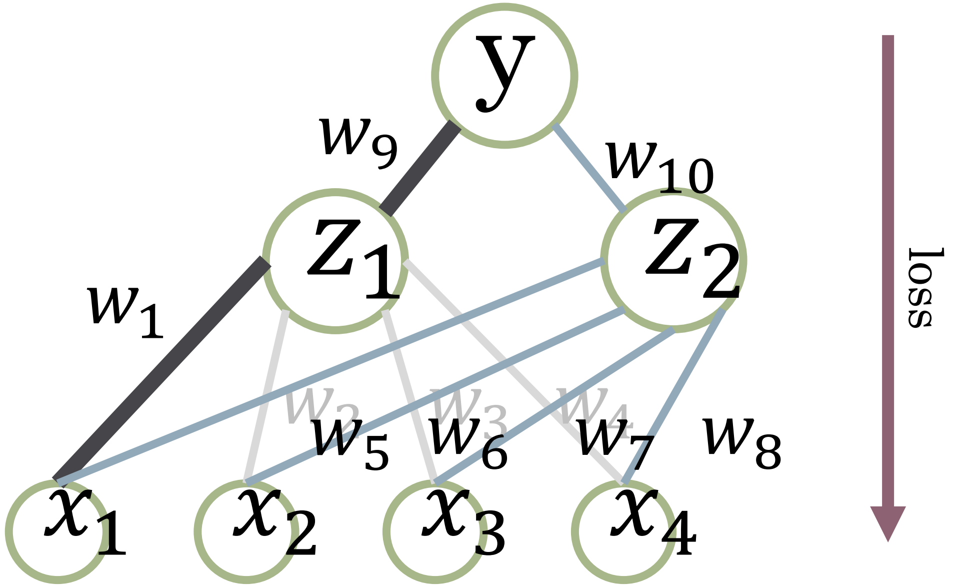
- How do we calculate the derivative of the error function?

- Chain rule

$$E_B(w) = \sum_{i \in B_k} |y_i - \hat{y}(w)_i|$$
$$\Rightarrow E = \frac{1}{2}(y - \hat{y}(w))^2 = \frac{1}{2}\Delta^2$$

$z = f(\sum_i w_i x_i)$

$$\partial_{w_1} E = \Delta\, \partial_{w_1} \hat{y} =$$
$$\Delta\, w_9\ \partial_{w_1} z_1 =$$
$$\Delta\, w_9\, x_1$$



Backpropagation

SGD

1. $w_{j,0} = random$
2. $v_{j,t} = L\nabla_{w_j} E_B(w_{j,t})$
3. $w_{j,t+1} = w_{j,t} - v_{j,t}$

- How do we calculate the derivative of the error function?

- Chain rule + Matrix multiplication

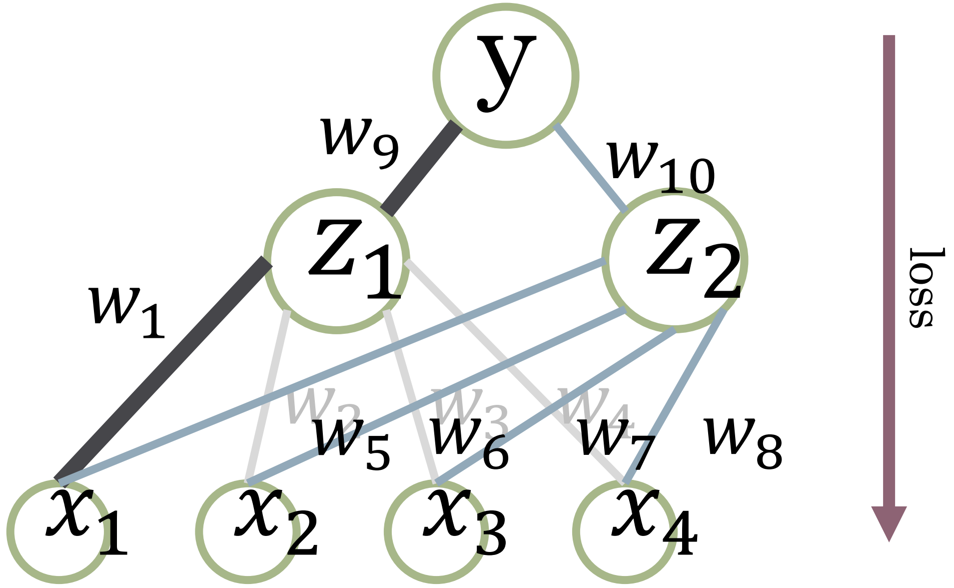
$$E_B(w) = \sum_{i \in B_k} |y_i - \hat{y}(w)_i|$$

$$\Rightarrow E = \frac{1}{2}(y - \hat{y}(w))^2 = \frac{1}{2}\Delta^2$$

$$z = f(\sum_i w_i x_i)$$

$$\partial_{w_1} E = \Delta\, \partial_{w_1} \hat{y} =$$
$$\Delta\, w_9\ \partial_{w_1} z_1 =$$
$$\Delta\, w_9\, x_1$$

With many weights and layers
⇒ **use matrix multiplication**



Backpropagation

SGD

1. $w_{j,0} = random$
2. $v_{j,t} = L\nabla_{w_j} E_B(w_{j,t})$
3. $w_{j,t+1} = w_{j,t} - v_{j,t}$

- How do we calculate the derivative of the error function?

- Chain rule + Matrix multiplication + storing the intermediate results

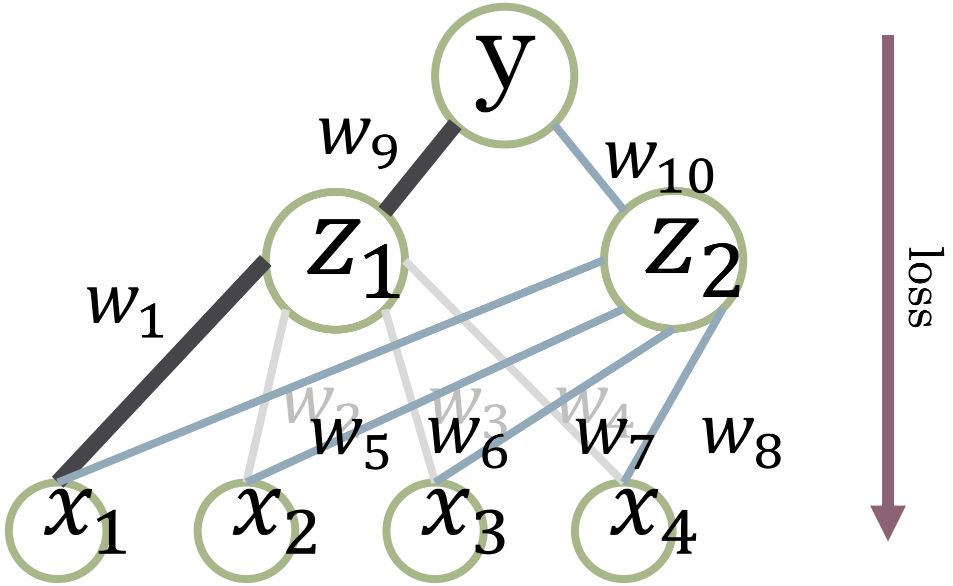
$$E_B(w) = \sum_{i \in B_k} |y_i - \hat{y}(w)_i|$$

$$\Rightarrow E = \frac{1}{2}(y - \hat{y}(w))^2 = \frac{1}{2}\Delta^2$$

$z = f(\sum_i w_i x_i)$

$$\partial_{w_1} E = \Delta\, \partial_{w_1} \hat{y} =$$
$$\Delta\, w_9\ \partial_{w_1} z_1 =$$
$$\Delta\, w_9\, x_1$$

With many weights and layers
⇒ **use matrix multiplication**



Backpropagation

SGD

1. $w_{j,0} = random$
2. $v_{j,t} = L\nabla_{w_j} E_B(w_{j,t})$
3. $w_{j,t+1} = w_{j,t} - v_{j,t}$

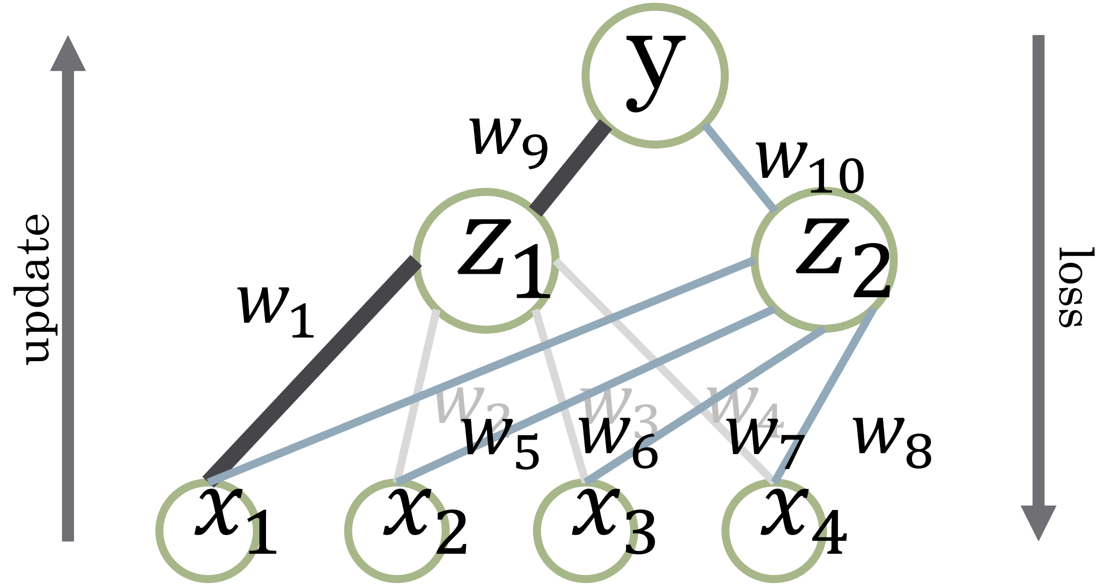
- How do we calculate the derivative of the error function?
- Chain rule + Matrix multiplication + storing the intermediate results
- Think of "loss" (i.e. step 2) and "update" (i.e. step 3) as two agents working together to train the network.

$$E_B(w) = \sum_{i \in B_k} |y_i - \hat{y}(w)_i|$$

$$\Rightarrow E = \frac{1}{2}(y - \hat{y}(w))^2 = \frac{1}{2}\Delta^2$$

$z = f(\sum_i w_i x_i)$

$$\partial_{w_1} E = \Delta\, \partial_{w_1} \hat{y} = \Delta\, w_9\ \partial_{w_1} z_1 = \Delta\, w_9\, x_1$$



Backpropagation

SGD

1. $w_{j,0} = random$
2. $v_{j,t} = L\nabla_{w_j} E_B(w_{j,t})$
3. $w_{j,t+1} = w_{j,t} - v_{j,t}$

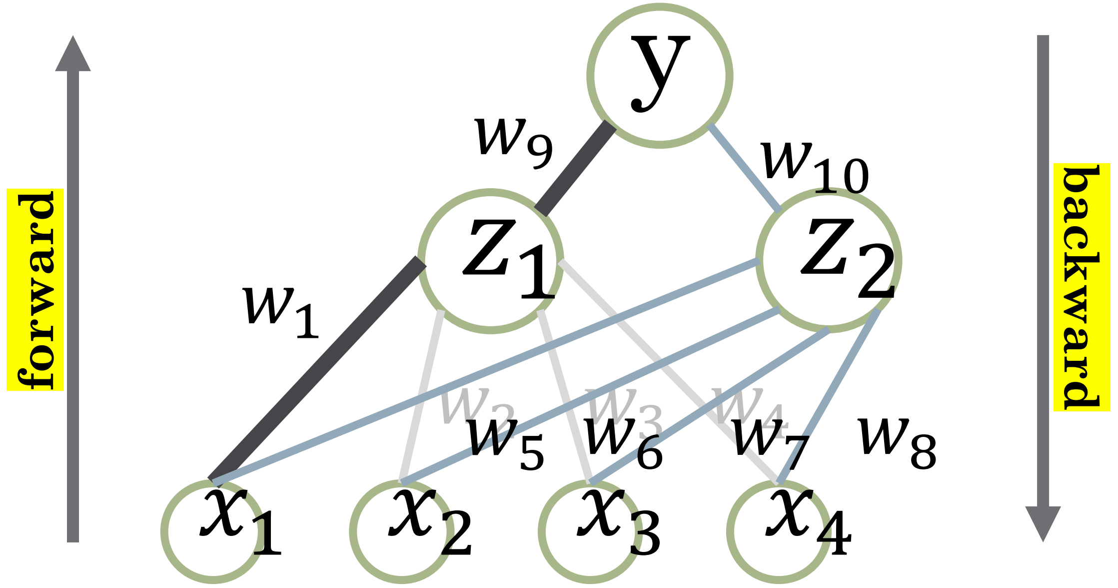
- How do we calculate the derivative of the error function?
- Chain rule + Matrix multiplication + storing the intermediate results
- Think of “loss” (i.e. step 2) and “update” (i.e. step 3) as two agents working together to train the network.

$$E_B(w) = \sum_{i \in B_k} |y_i - \hat{y}(w)_i|$$

$$\Rightarrow E = \frac{1}{2}(y - \hat{y}(w))^2 = \frac{1}{2}\Delta^2$$

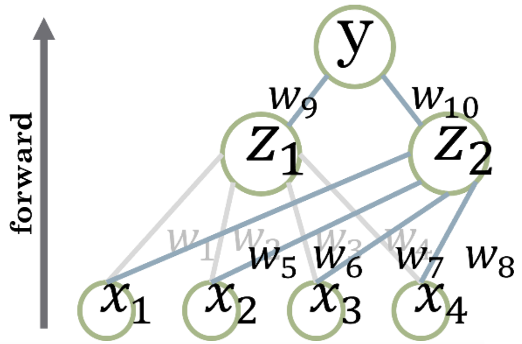
$z = f(\sum_i w_i x_i)$

$$\partial_{w_1} E = \Delta\, \partial_{w_1} \hat{y} = \Delta\, w_9\ \partial_{w_1} z_1 = \Delta\, w_9\, x_1$$



Code example: forward

Implementation in the "real-world": **PyTorch**

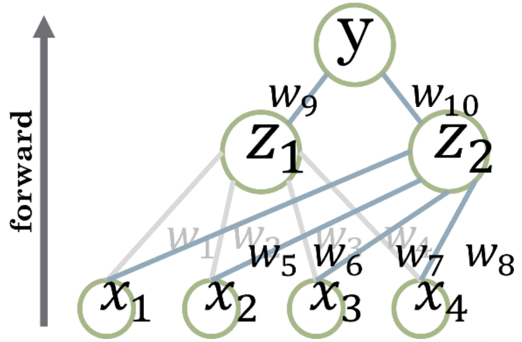


Define the Class

We define our neural network by subclassing `nn.Module`, and initialize the neural network layers in `__init__`. Every `nn.Module` subclass implements the operations on input data in the `forward` method.

```
[ ]  class NeuralNetwork(nn.Module):
         def __init__(self):
             super().__init__()
             self.flatten = nn.Flatten()
             self.linear_relu_stack = nn.Sequential(
                 nn.Linear(28*28, 512),
                 nn.ReLU(),
                 nn.Linear(512, 512),
                 nn.ReLU(),
                 nn.Linear(512, 10),
             )

         def forward(self, x):
             x = self.flatten(x)
             logits = self.linear_relu_stack(x)
             return logits
```

Code example: forward

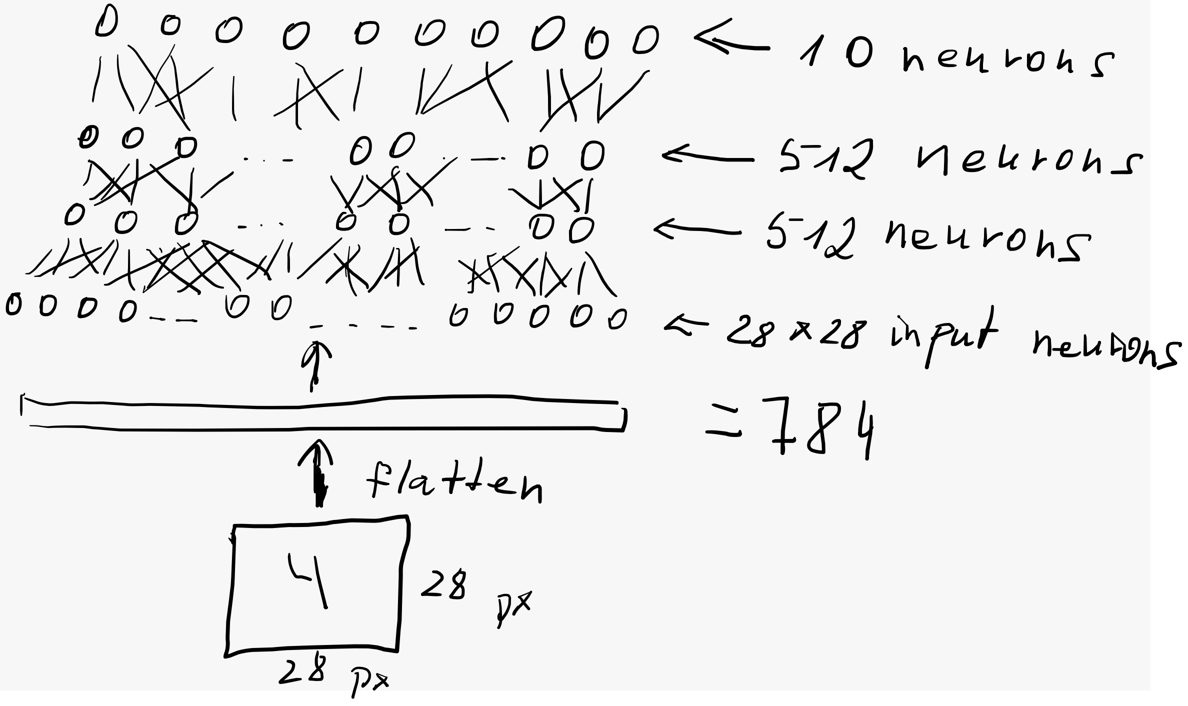


Implementation in the "real-world": **PyTorch**

Define the Class

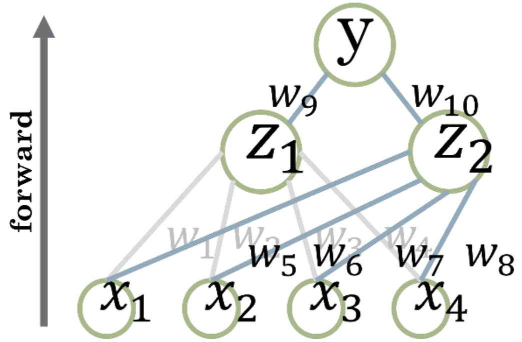
We define our neural network by subclassing `nn.Module`, and initialize the neural network layers in `__init__`. Every `nn.Module` subclass implements the operations on input data in the `forward` method.

```
[ ]  class NeuralNetwork(nn.Module):
         def __init__(self):
             super().__init__()
             self.flatten = nn.Flatten()
             self.linear_relu_stack = nn.Sequential(
                 nn.Linear(28*28, 512),
                 nn.ReLU(),
                 nn.Linear(512, 512),
                 nn.ReLU(),
                 nn.Linear(512, 10),
             )

         def forward(self, x):
             x = self.flatten(x)
             logits = self.linear_relu_stack(x)
             return logits
```



Code example: forward

Implementation in the "real-world": **PyTorch**



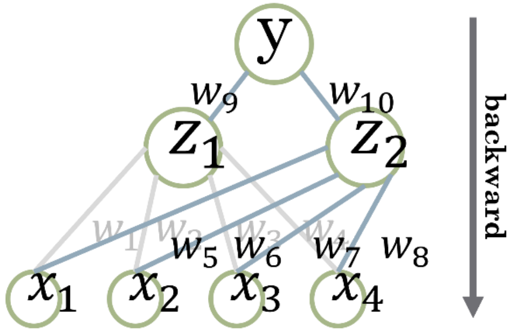
To use the model, we pass it the input data. This executes the model's `forward`, along with some [background operations](). Do not call `model.forward()` directly!

Calling the model on the input returns a 2-dimensional tensor with dim=0 corresponding to each output of 10 raw predicted values for each class, and dim=1 corresponding to the individual values of each output. We get the prediction probabilities by passing it through an instance of the `nn.Softmax` module.

```
[ ] X = torch.rand(1, 28, 28, device=device)
    logits = model(X)
    pred_probab = nn.Softmax(dim=1)(logits)
    y_pred = pred_probab.argmax(1)
    print(f"Predicted class: {y_pred}")
```

Code example: backward

Implementation in the "real-world": **PyTorch**



Consider the simplest one-layer neural network, with input `x`, parameters `w` and `b`, and some loss function. It can be defined in PyTorch in the following manner:

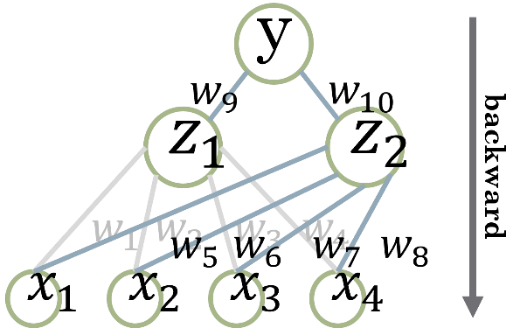
```
[2]  import torch

     x = torch.ones(5)  # input tensor
     y = torch.zeros(3)  # expected output
     w = torch.randn(5, 3, requires_grad=True)
     b = torch.randn(3, requires_grad=True)
     z = torch.matmul(x, w)+b
     loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

Computing Gradients

To optimize weights of parameters in the neural network, we need to compute the derivatives of our loss function with respect to parameters, namely, we need $\frac{\partial loss}{\partial w}$ and $\frac{\partial loss}{\partial b}$ under some fixed values of `x` and `y`. To compute those derivatives, we call `loss.backward()`, and then retrieve the values from `w.grad` and `b.grad`:

```
[3]  loss.backward()
     print(w.grad)
     print(b.grad)
```

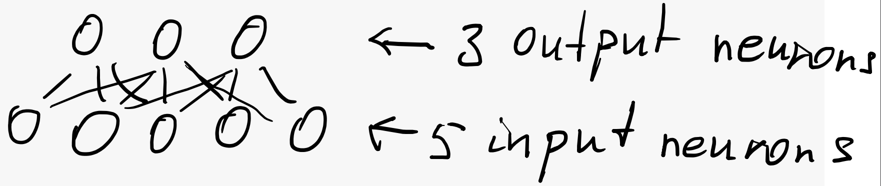
Code example: backward



Implementation in the “real-world”: **PyTorch**

Consider the simplest one-layer neural network, with input `x`, parameters `w` and `b`, and some loss function. It can be defined in PyTorch in the following manner:

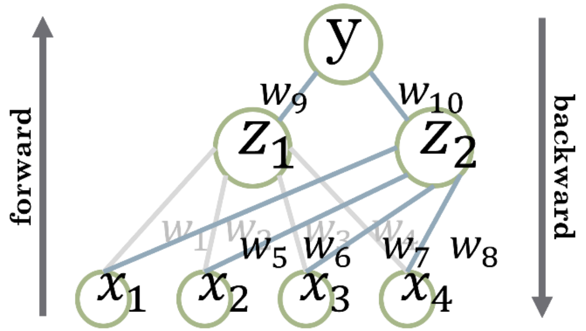
```
[2] import torch

    x = torch.ones(5)  # input tensor
    y = torch.zeros(3)  # expected output
    w = torch.randn(5, 3, requires_grad=True)
    b = torch.randn(3, requires_grad=True)
    z = torch.matmul(x, w)+b
    loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```



Computing Gradients

To optimize weights of parameters in the neural network, we need to compute the derivatives of our loss function with respect to parameters, namely, we need $\frac{\partial loss}{\partial w}$ and $\frac{\partial loss}{\partial b}$ under some fixed values of `x` and `y`. To compute those derivatives, we call `loss.backward()`, and then retrieve the values from `w.grad` and `b.grad`:

```
[3] loss.backward()
    print(w.grad)
    print(b.grad)
```



Code example: full implementation

Implementation in the "real-world": **PyTorch**

```
[ ] # Initialize the loss function
    loss_fn = nn.CrossEntropyLoss()
    # Initialize the optimizer
    optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

    epochs = 10
    for t in range(epochs):
        train_loop(train_dataloader, model, loss_fn, optimizer)
        test_loop(test_dataloader, model, loss_fn)
    print("Done!")
```

$$\boxed{E = -\sum_i y_i \log(f)(1 - y_i)\log(1 - f)}$$

Cross-entropy loss

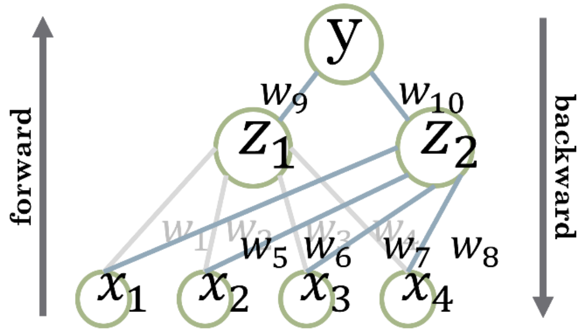
1. $w_0 = random$
2. $v_t = L\nabla_w E_B(w_t)$
3. $w_{t+1} = w_t - v_t$

SGD optimizer

Code example: full implementation

Implementation in the "real-world": **PyTorch**

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss = 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()

    test_loss /= num_batches
    print(f"Avg loss: {test_loss:>8f} \n")
```



$$\mathrm{E} = -\sum_i y_i \log(f)(1-y_i)\log(1-f)$$

Cross-entropy loss

1. $w_{j,0} = random$
2. $v_{j,t} = L\nabla_{w_j} E_B(w_{j,t})$
3. $w_{j,t+1} = w_{j,t} - v_{j,t}$

SGD optimizer

Deep learning components

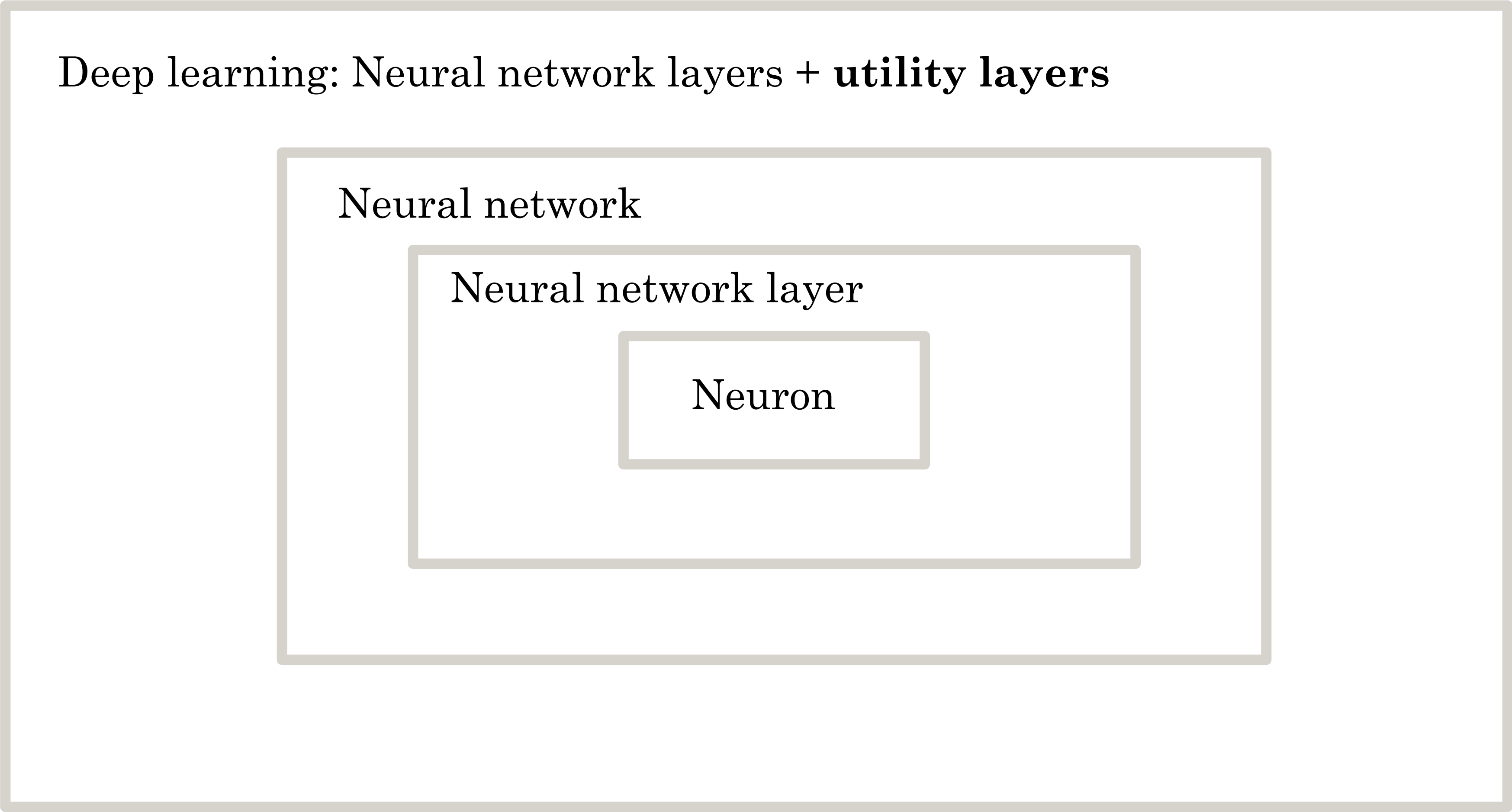
Deep learning: Large set of mathematical operations + helper algorithms

NN: Set of mathematical operations

NN layer: Set of non-linear functions

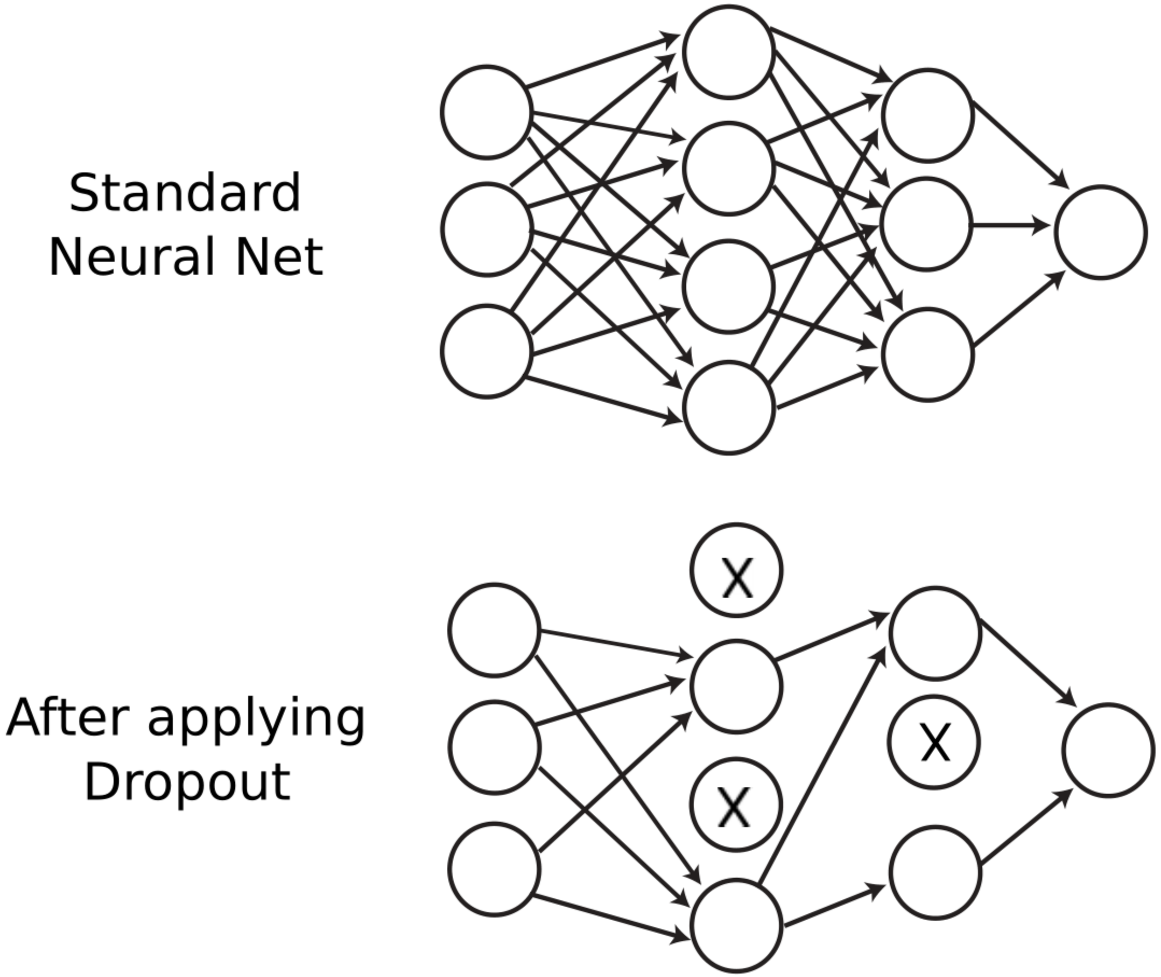
Neuron: non-linear function

Deep learning components



Dropout

Deep learning: Neural network layers + **utility layers**

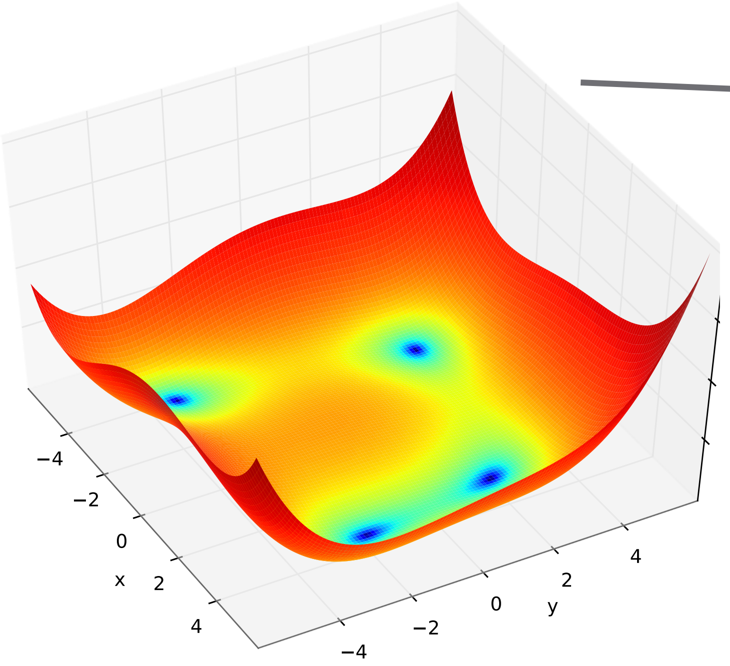


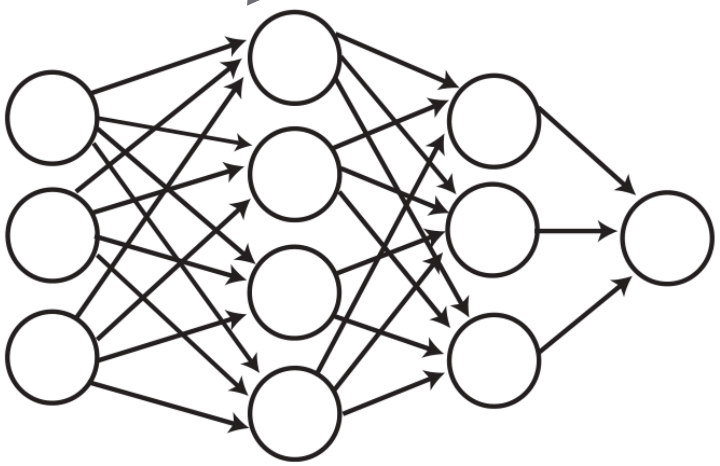
Mehta *et al.*, Physics reports **810**, 1-124 (2019)

Batch normalization

Deep learning: Neural network layers + **utility layers**

- Training speed-up AND regularization
- Prevents neurons from saturating and gradients from vanishing in deep nets
- Prevents changes in lower layers to significantly impact higher layers





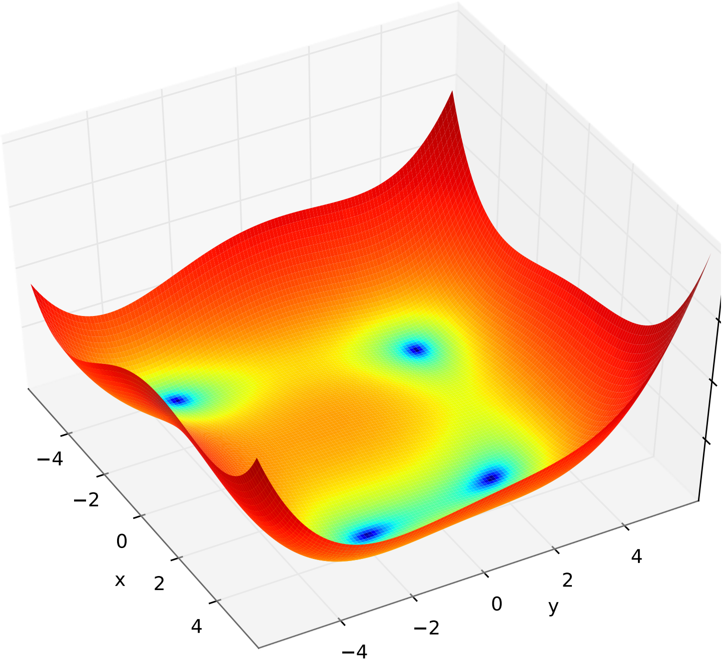
Wikimedia (retrieved 2023-04-07)

Mehta *et al.*, Physics reports **810**, 1-124 (2019)

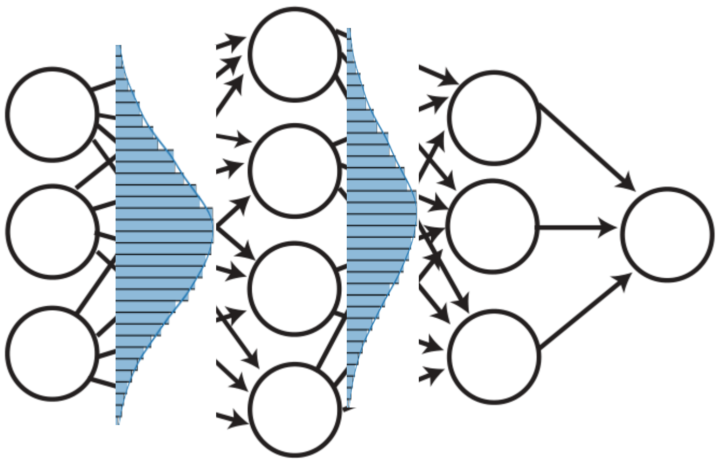
Batch normalization

Deep learning: Neural network layers + **utility layers**

- Training speed-up AND regularization
- Prevents neurons from saturating and gradients from vanishing in deep nets
- Prevents changes in lower layers to significantly impact higher layers



Wikimedia (retrieved 2023-04-07)



Mehta *et al.*, Physics reports **810**, 1-124 (2019)