Data Science Bootcamp, 12th January 2017

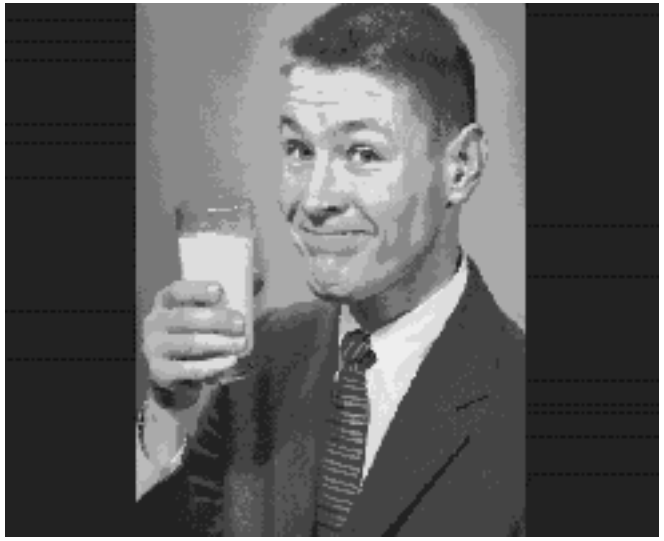
Neural Networks

Francisco J. R. Ruiz

Columbia University
Data Science Institute

Introduction

Introduction

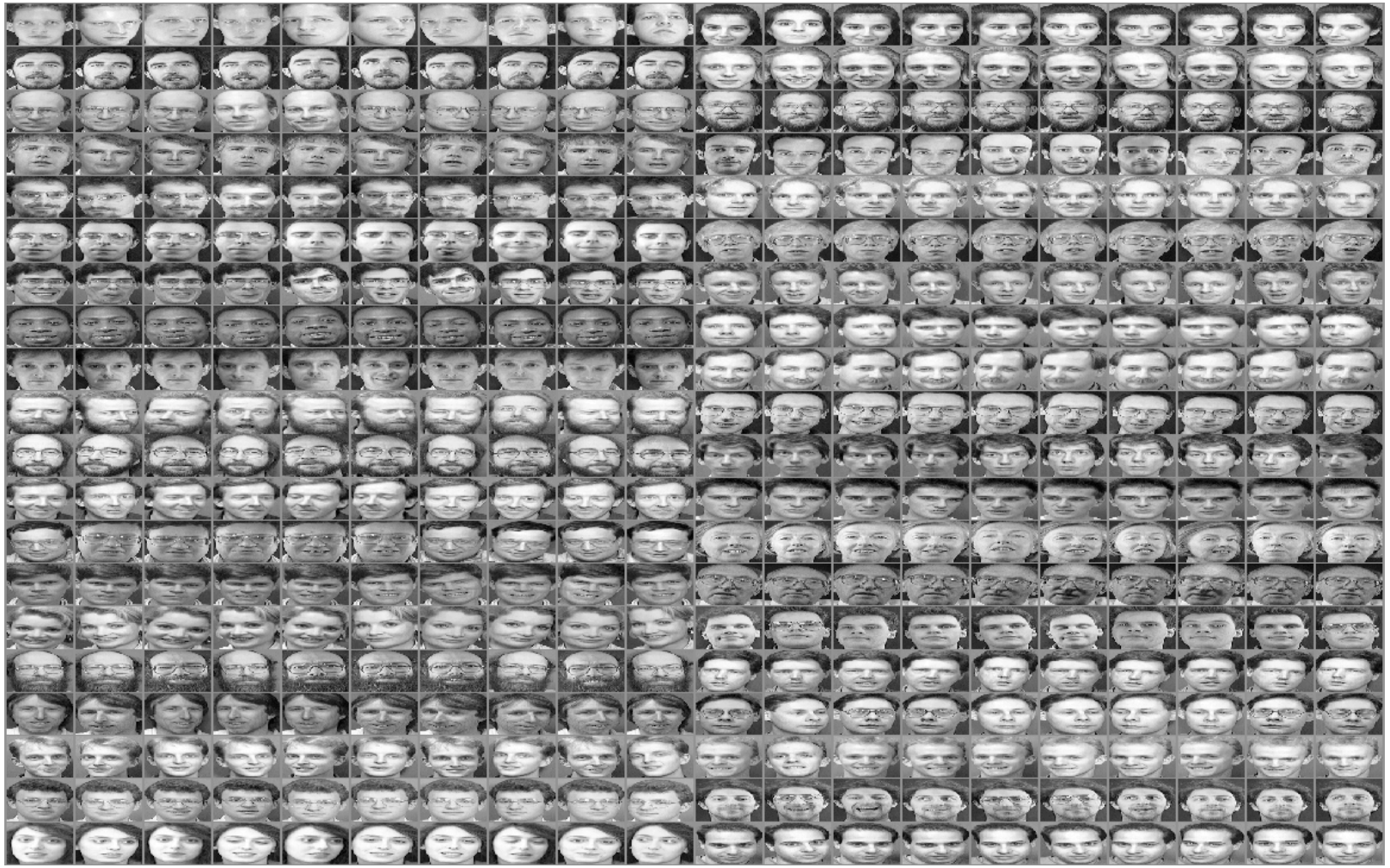


Introduction

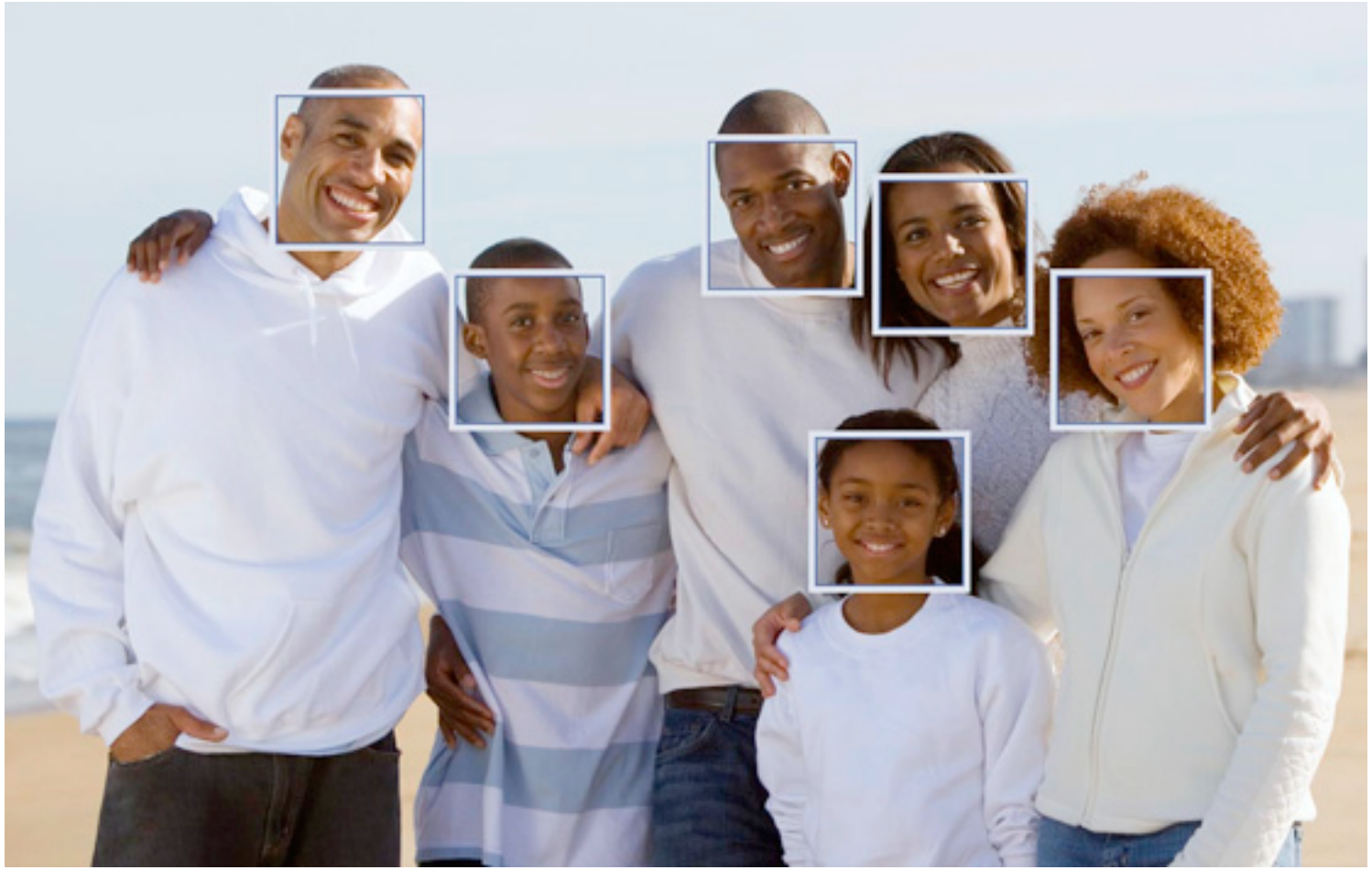
- We call this change blindness
- Problem trivial for computers
- As humans, we get high-level descriptors (“man+glass”)
- But descriptors (features) are challenging for computers

Introduction

Introduction

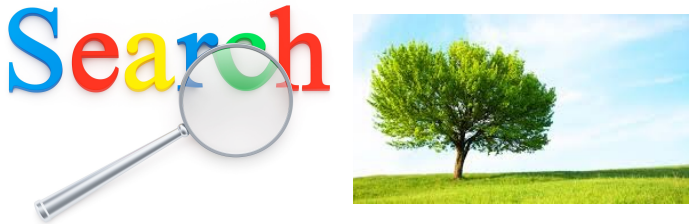


Introduction



Introduction

Deep Learning: Applications

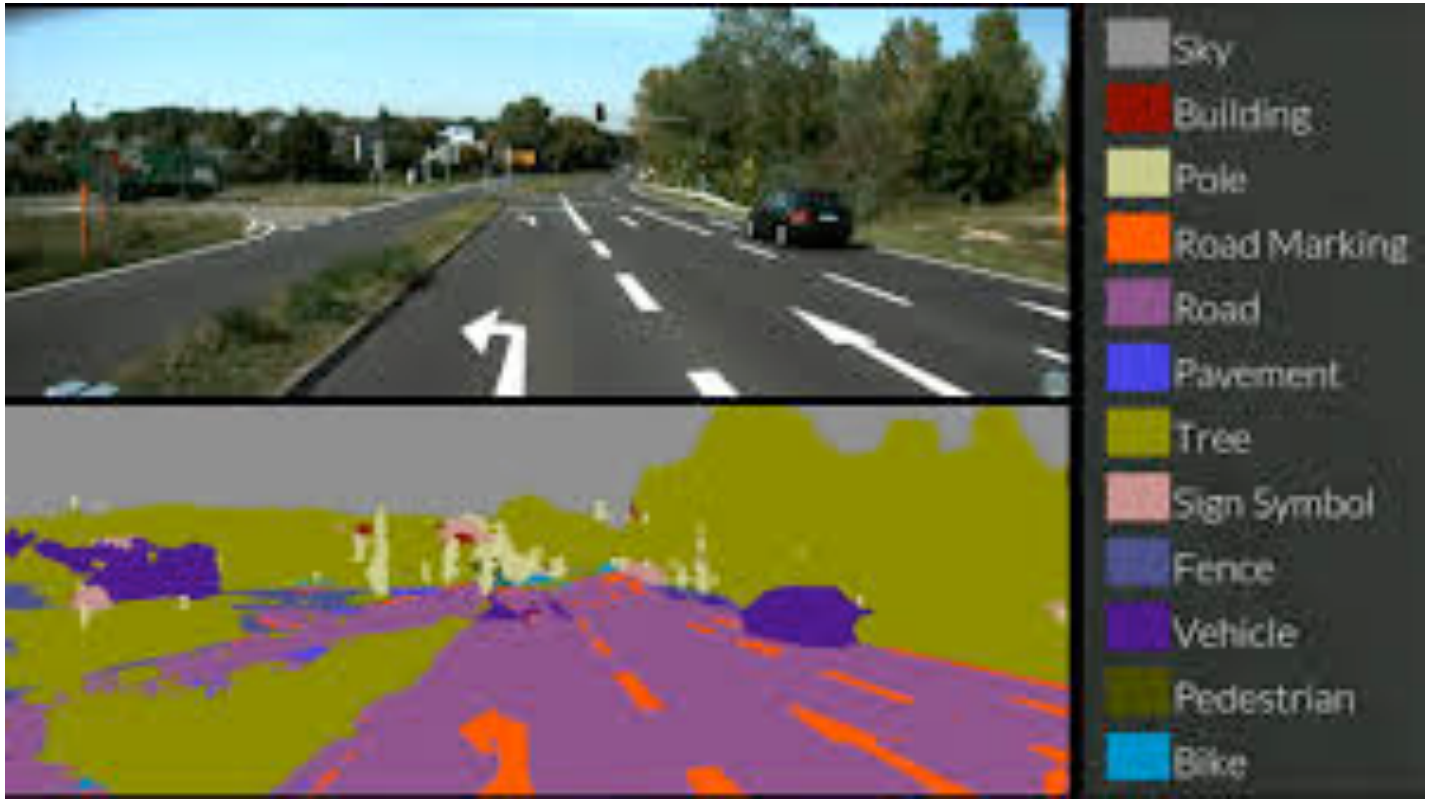


Visually similar images

Deep Learning: Applications

Deep Learning: Applications

Deep Learning: Applications



Deep Learning: Applications

Deep Learning: Applications

DeepDrumpf
@DeepDrumpf

DeepDrumpf @DeepDrumpf · 8 nov. 2016
[I told Ohio] my promise to the American voter: If I am elected President, I will grow your money. $500 billion a year to be a Republican.

DeepDrumpf
@DeepDrumpf

Seguir

[Math is a] common democrat lie. It can't make the budget great. I'll have the best economy. #debatenight

Deep Learning: Brief History

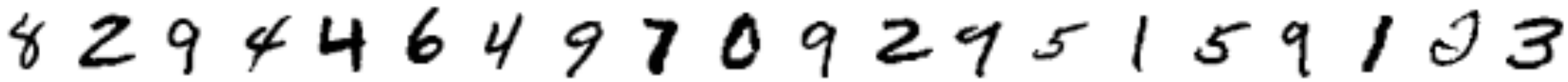
- Deep learning has become a hot topic recently…

  … but it dates back to the 70s

- It has been "re-discovered" recently due to
  - Massive amounts of data
  - Computational capabilities

Neural Networks

- Running example: Classification

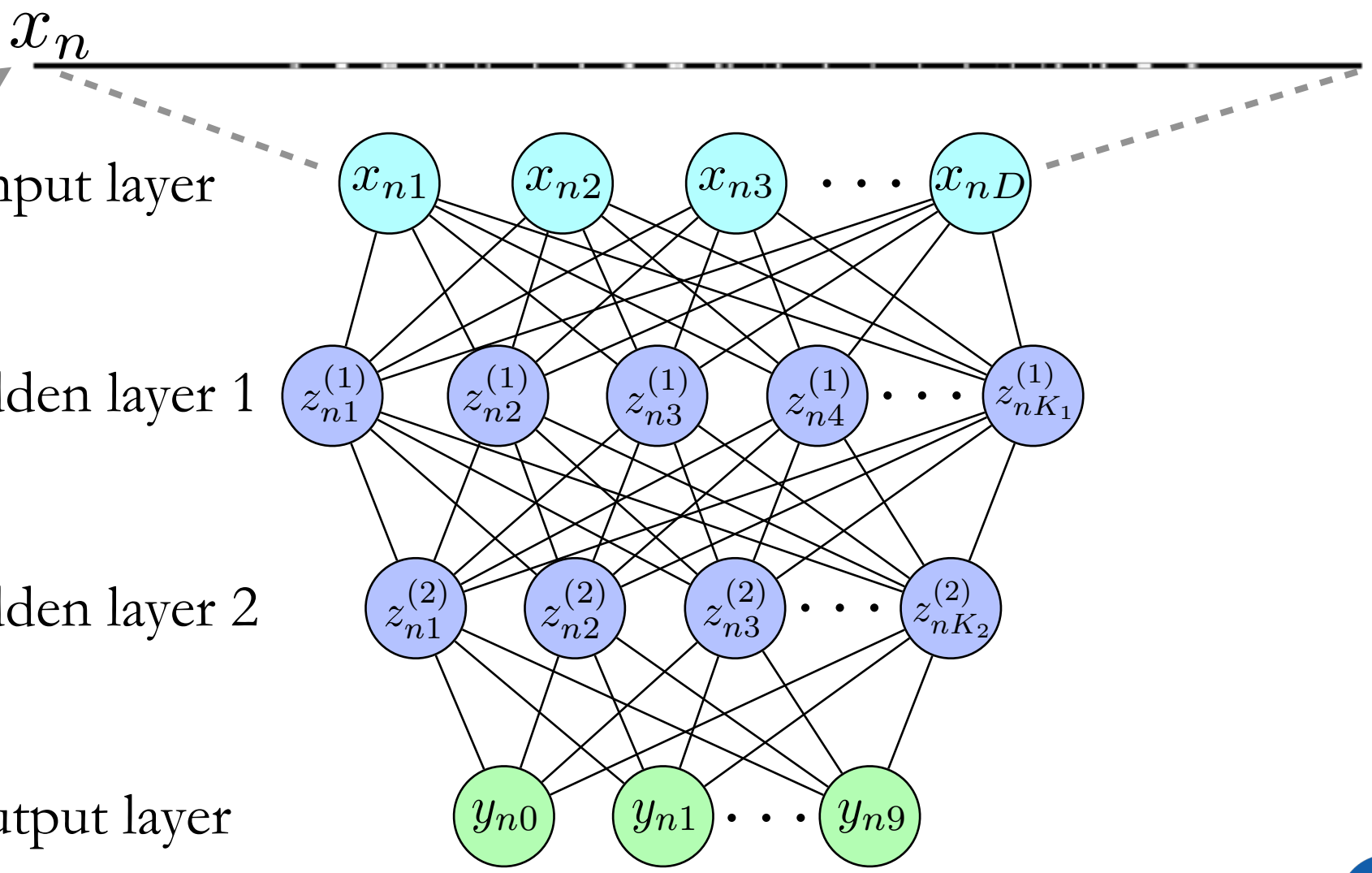


- NNs can also be applied for regression (and even for unsupervised learning)

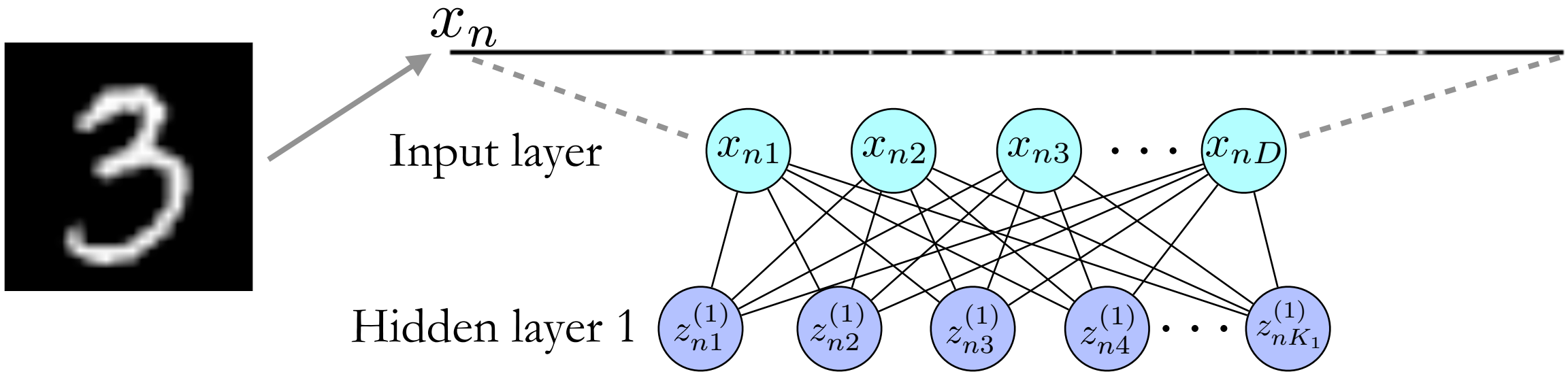
Neural Networks

- Cascade of layers from the input to the output
    1. Input layer
    2. Hidden layers (any number)
    3. Output layer

Neural Networks

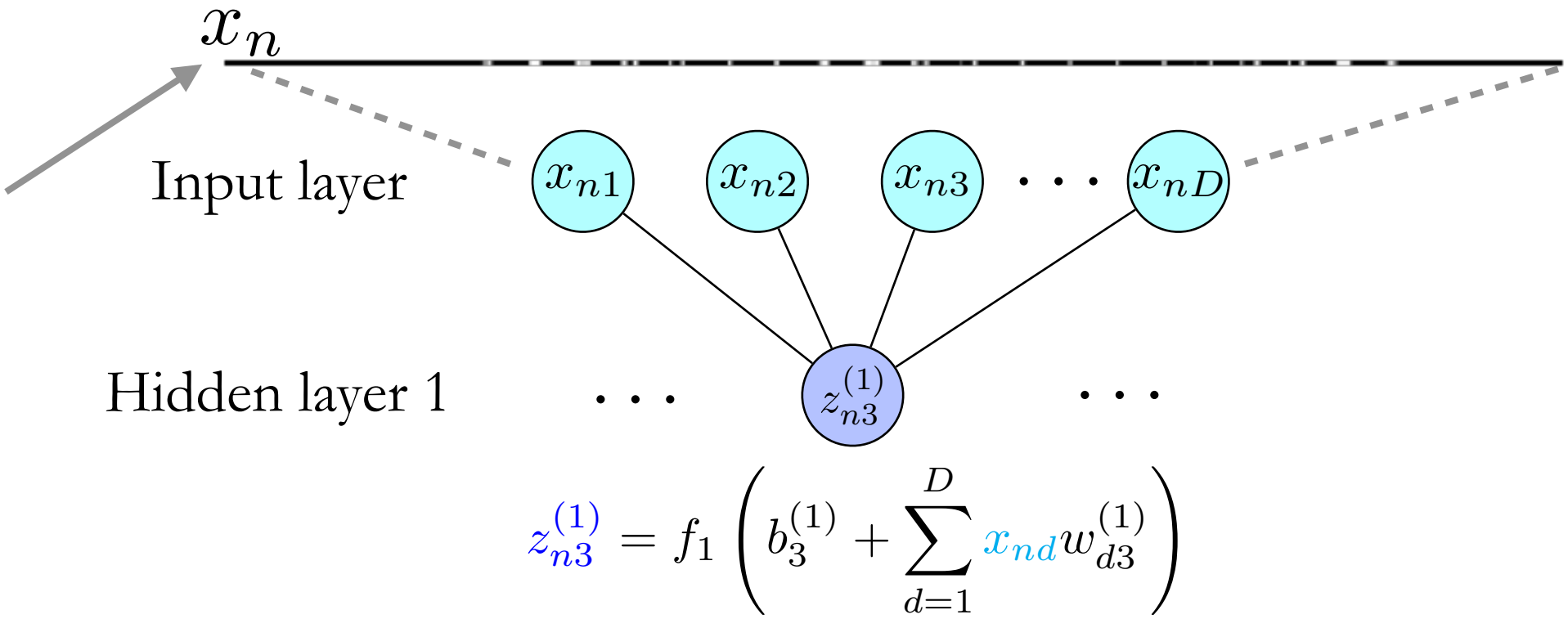


Neural Networks



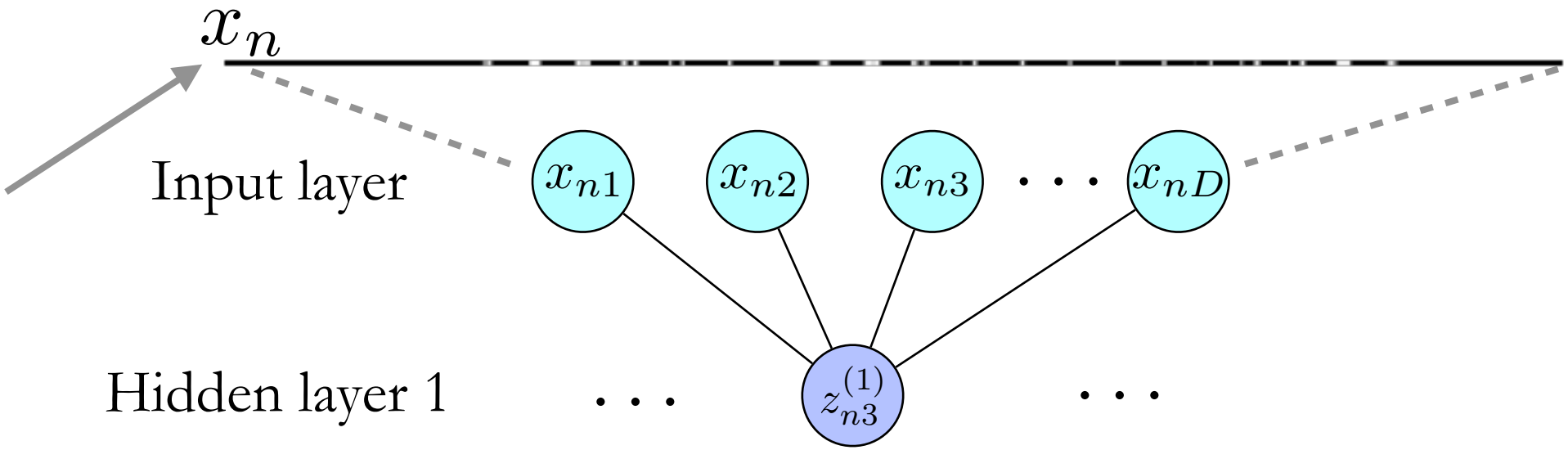
- Each layer:
  - Dot product & Non-linear function:

Neural Networks



$$z_{n3}^{(1)} = f_1\left(b_3^{(1)} + \sum_{d=1}^{D} x_{nd} w_{d3}^{(1)}\right)$$

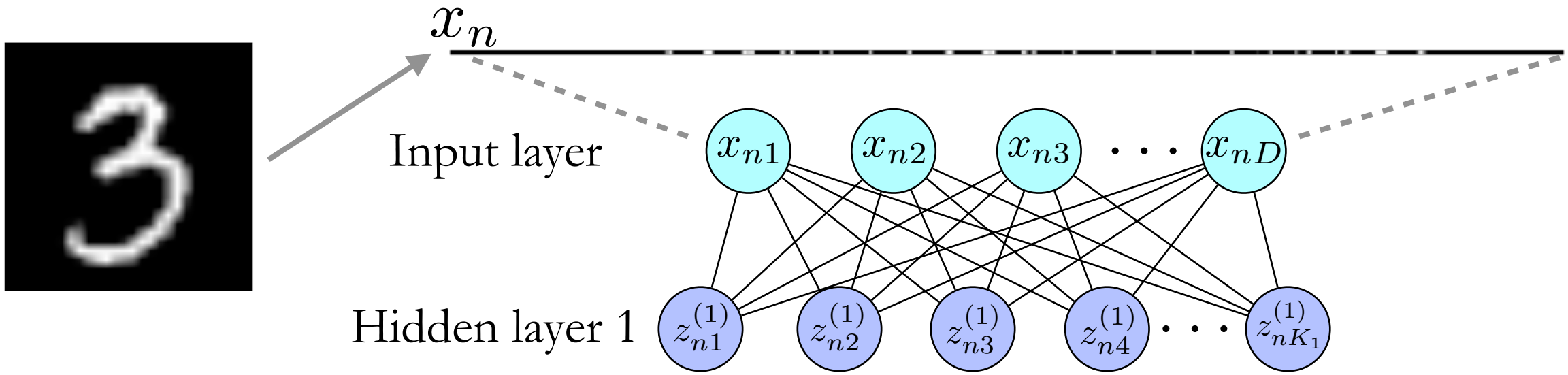
Neural Networks



$$z_{n3}^{(1)} = f_1\left(b_3^{(1)} + \sum_{d=1}^{D} x_{nd} w_{d3}^{(1)}\right)$$

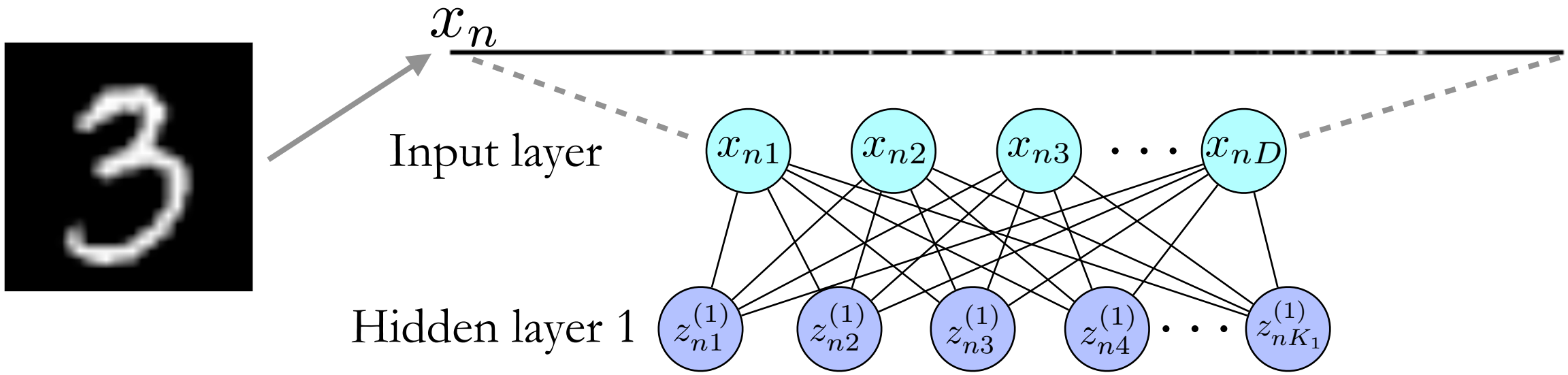
$$\mathbf{z}_n^{(1)} = f_1(\mathbf{x}_n \mathbf{W}^{(1)} + \mathbf{b}^{(1)})$$

Neural Networks



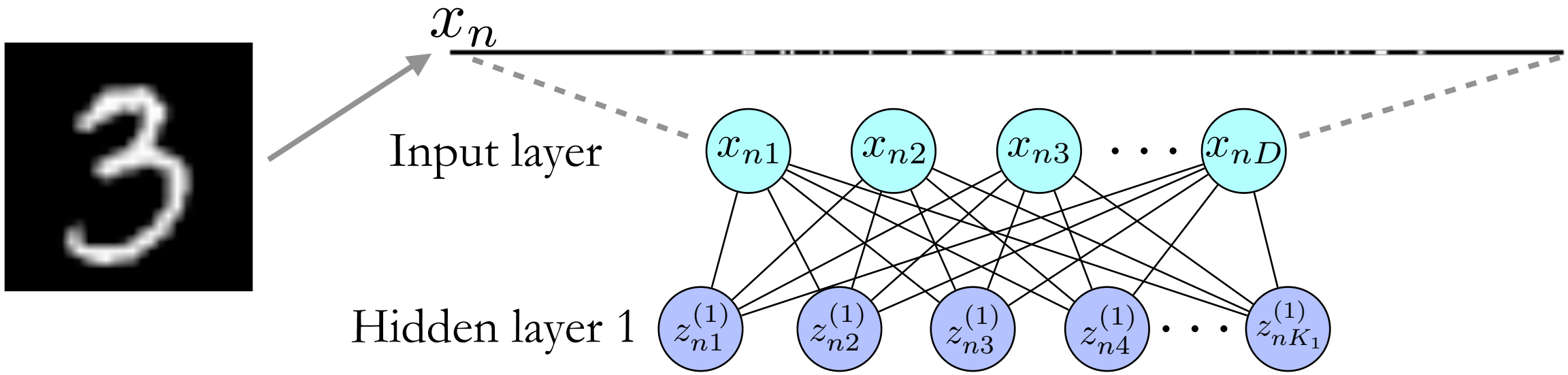
- Each layer:
  - Dot product & Non-linear function: $\mathbf{z}_n^{(1)} = f_1(\mathbf{x}_n\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$
  - Number of weights?

Neural Networks



- Each layer:
  - Dot product & Non-linear function: $\mathbf{z}_n^{(1)} = f_1(\mathbf{x}_n\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$
  - Number of weights? $D \times K_1$

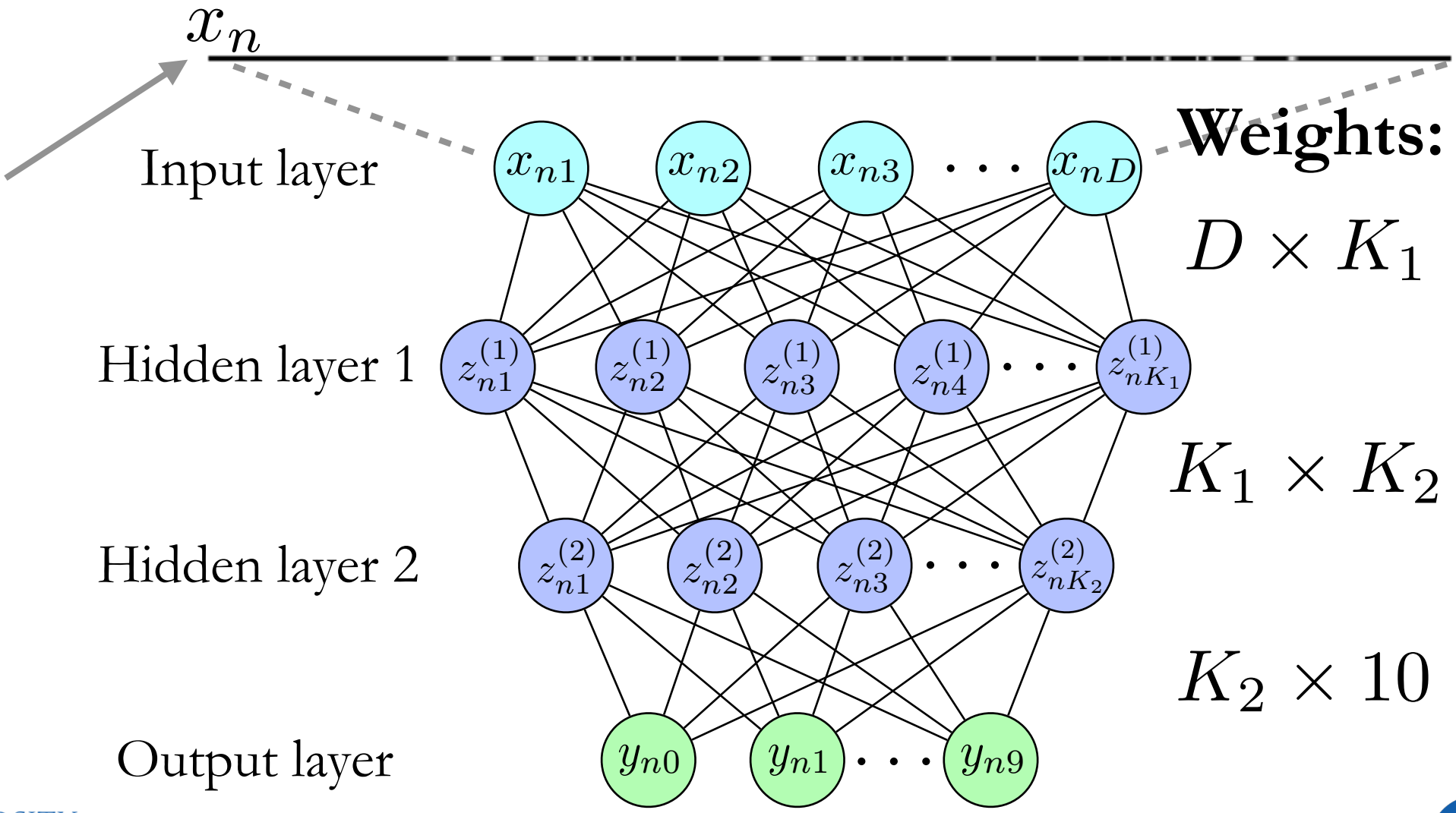
Neural Networks



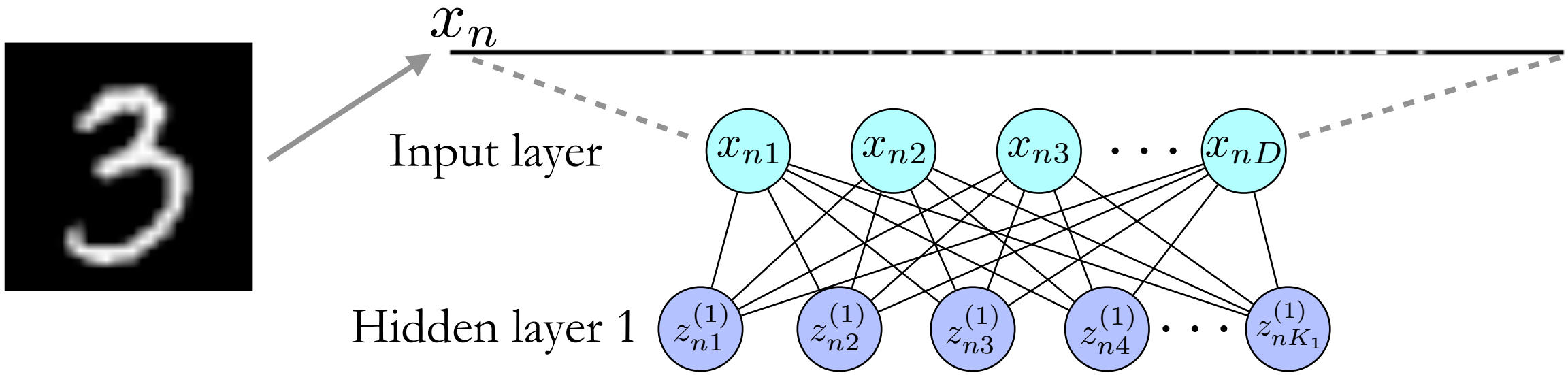
- Number of weights for each layer:

  Input dimension $\times$ Output dimension

Neural Networks



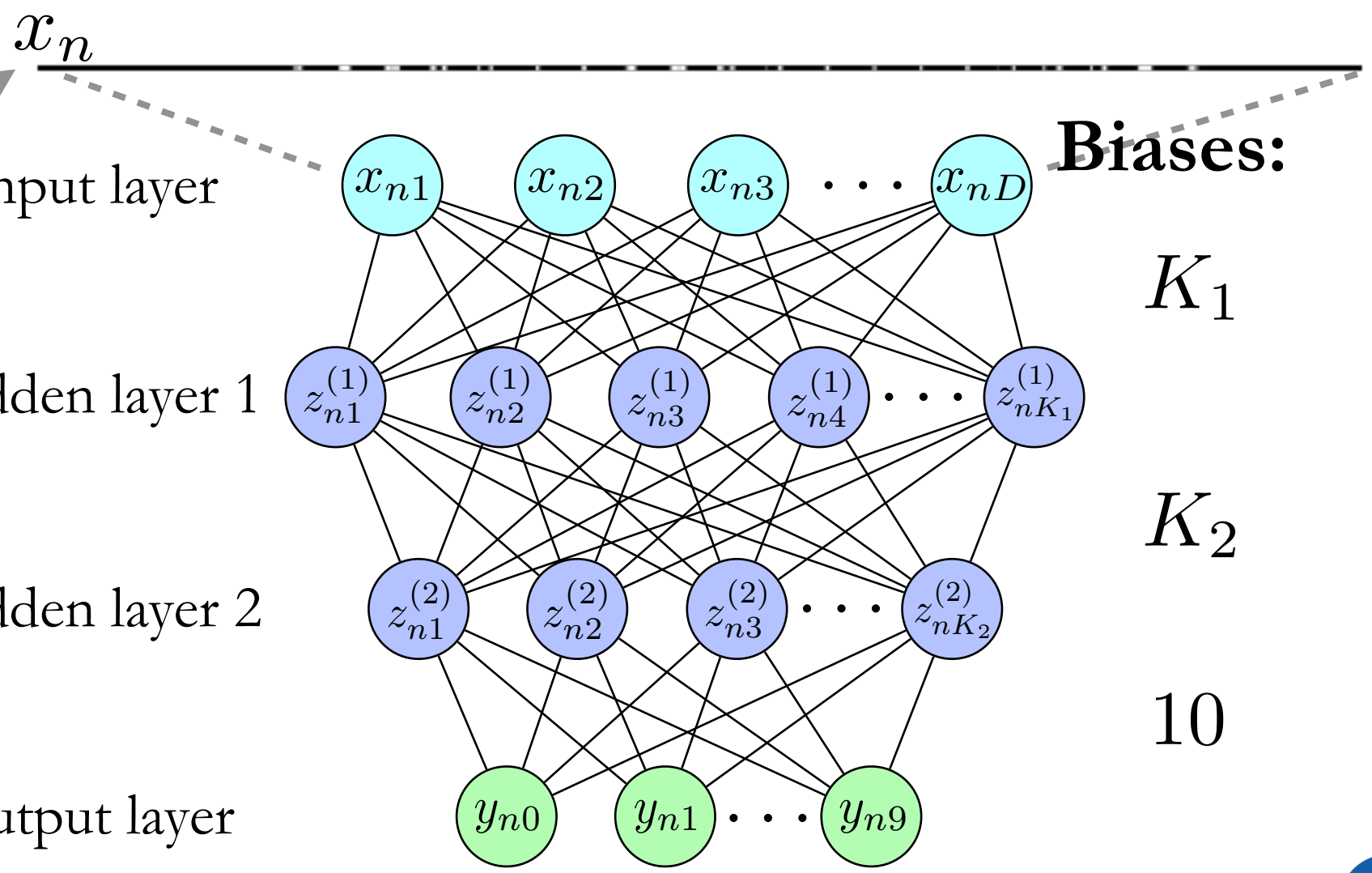
Neural Networks



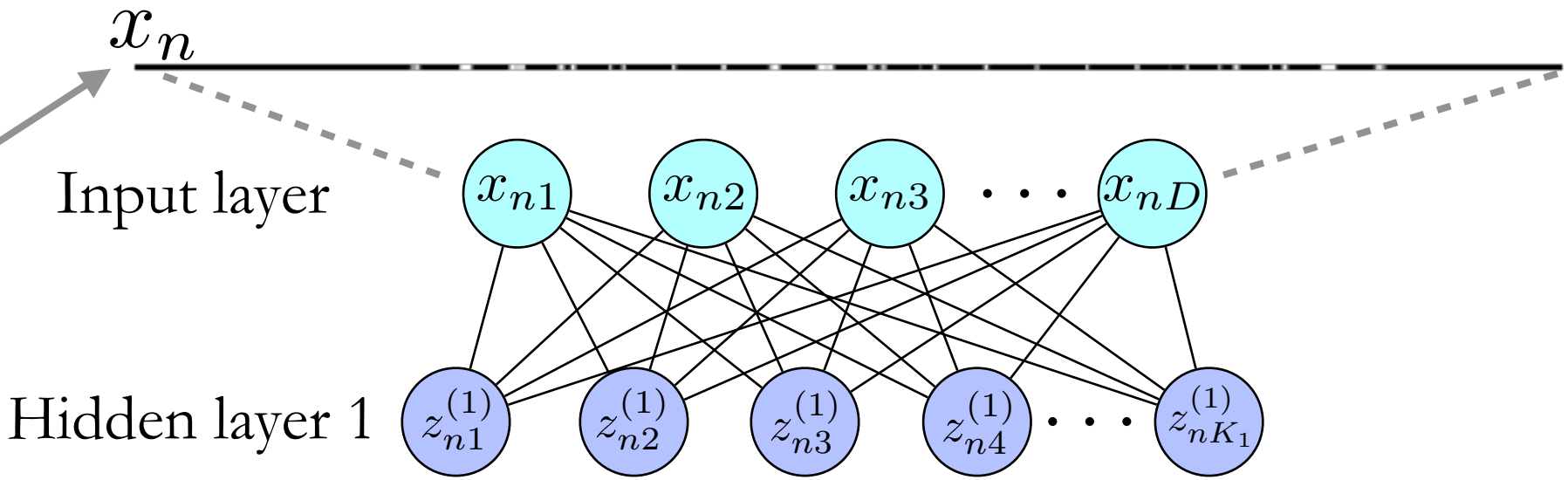
- Number of biases for each layer:

  Output dimension

Neural Networks



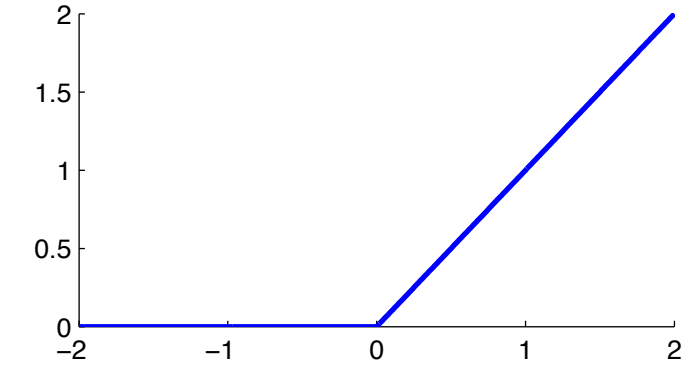
Neural Networks



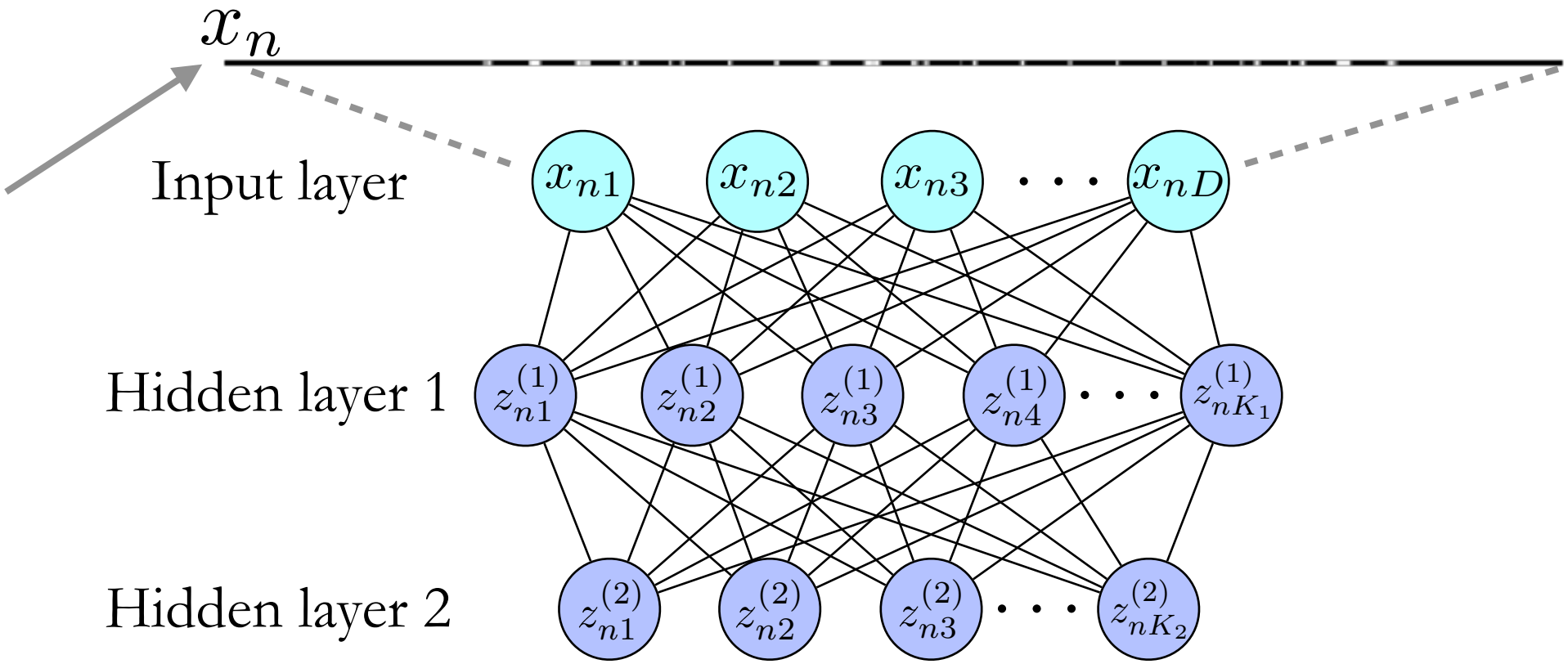
$$\mathbf{z}_n^{(1)} = f_1(\mathbf{x}_n \mathbf{W}^{(1)} + \mathbf{b}^{(1)})$$

- Non-linear function: ReLU

$$f_1(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

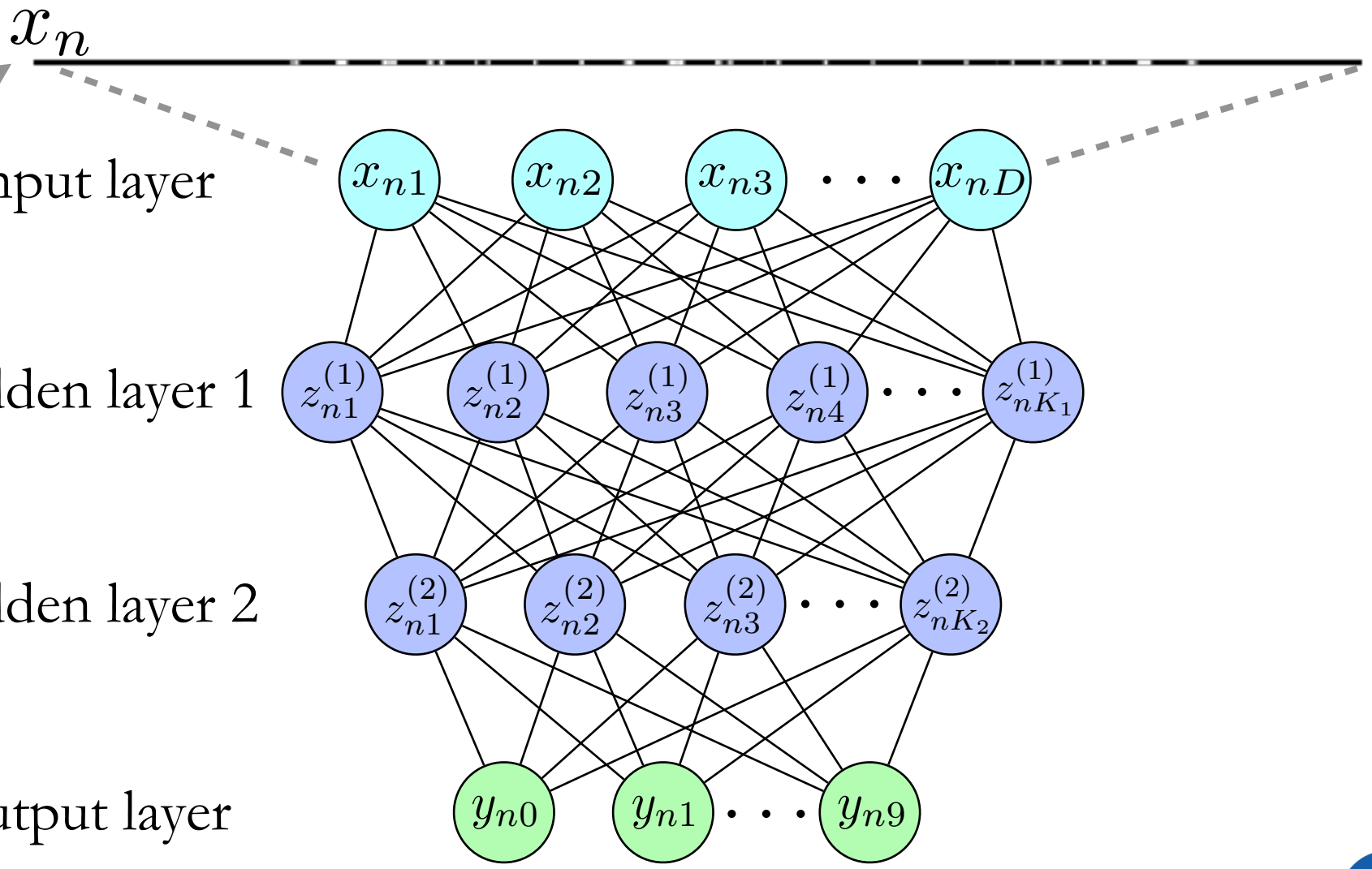


Neural Networks



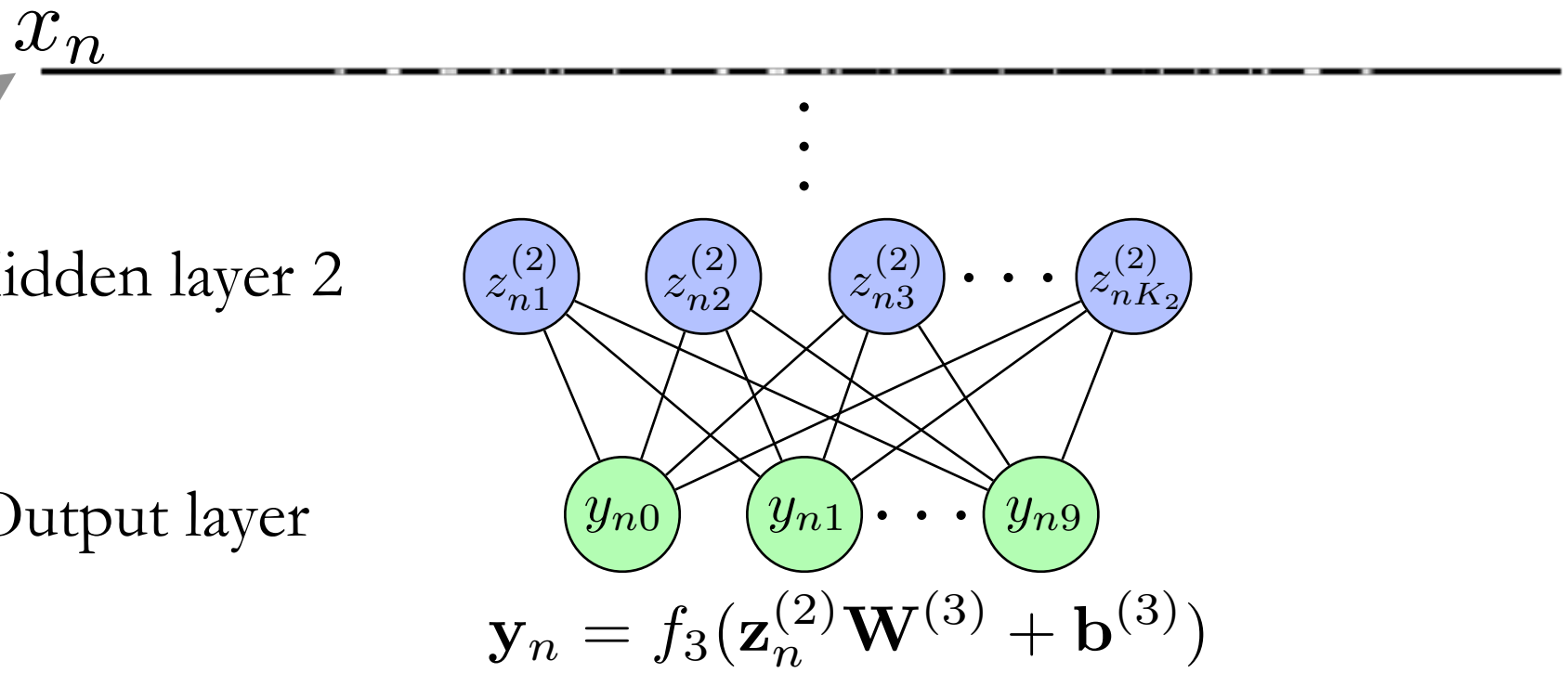
$$\mathbf{z}_n^{(2)} = f_2(\mathbf{z}_n^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$$

Neural Networks



$$\mathbf{y}_n = f_3(\mathbf{z}_n^{(2)}\mathbf{W}^{(3)} + \mathbf{b}^{(3)})$$

Neural Networks

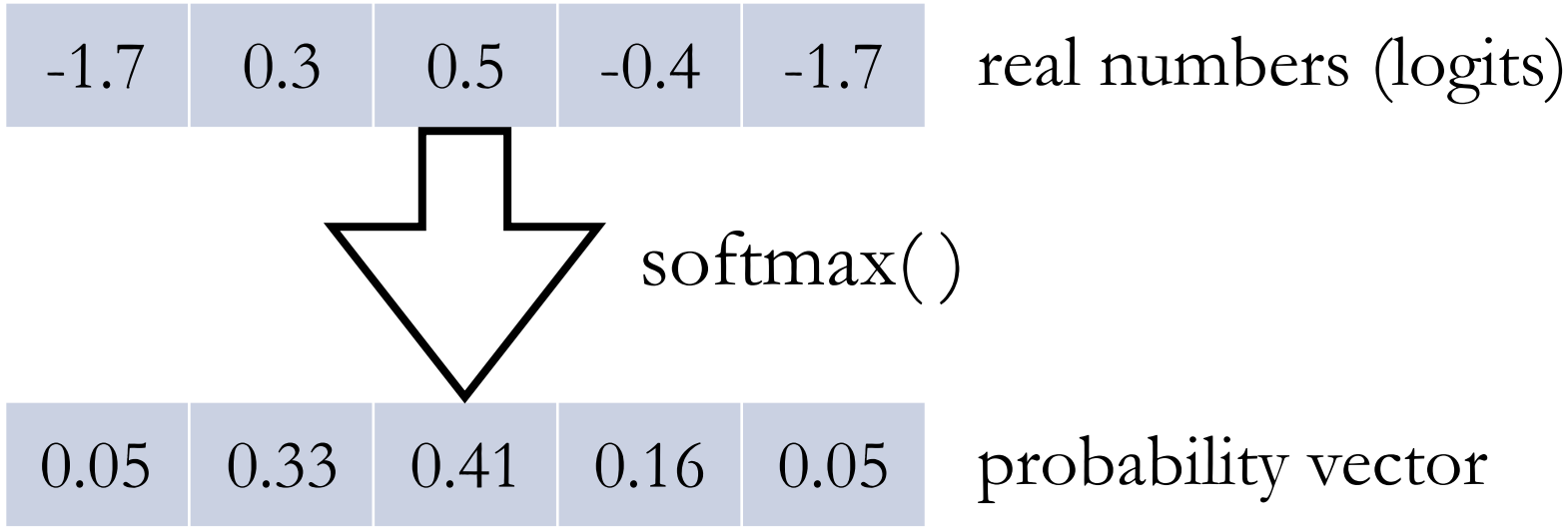


The output should be the probability for each class

- The non-linear function is a softmax

An Aside: Softmax

- The softmax is a function that inputs a vector of *reals* and outputs a *probability* vector

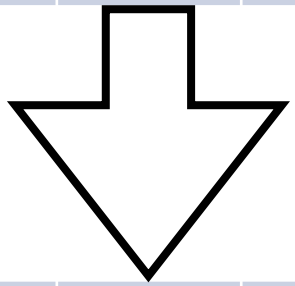


An Aside: Softmax

- The softmax is a function that inputs a vector of *reals* and outputs a *probability* vector

-1.7	0.3	0.5	-0.4	-1.7

real numbers (logits)



softmax( )

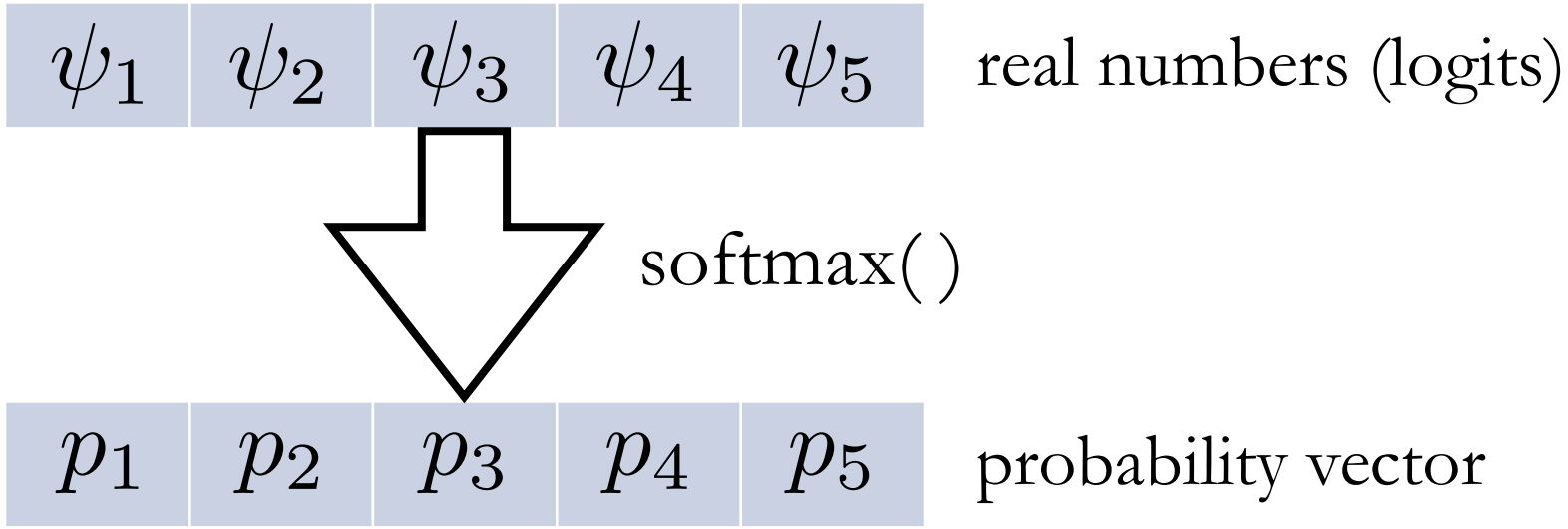
0.05	0.33	0.41	0.16	0.05

probability vector

$$0.41 = \frac{e^{0.5}}{e^{-1.7} + e^{0.3} + e^{0.5} + e^{-0.4} + e^{-1.7}}$$

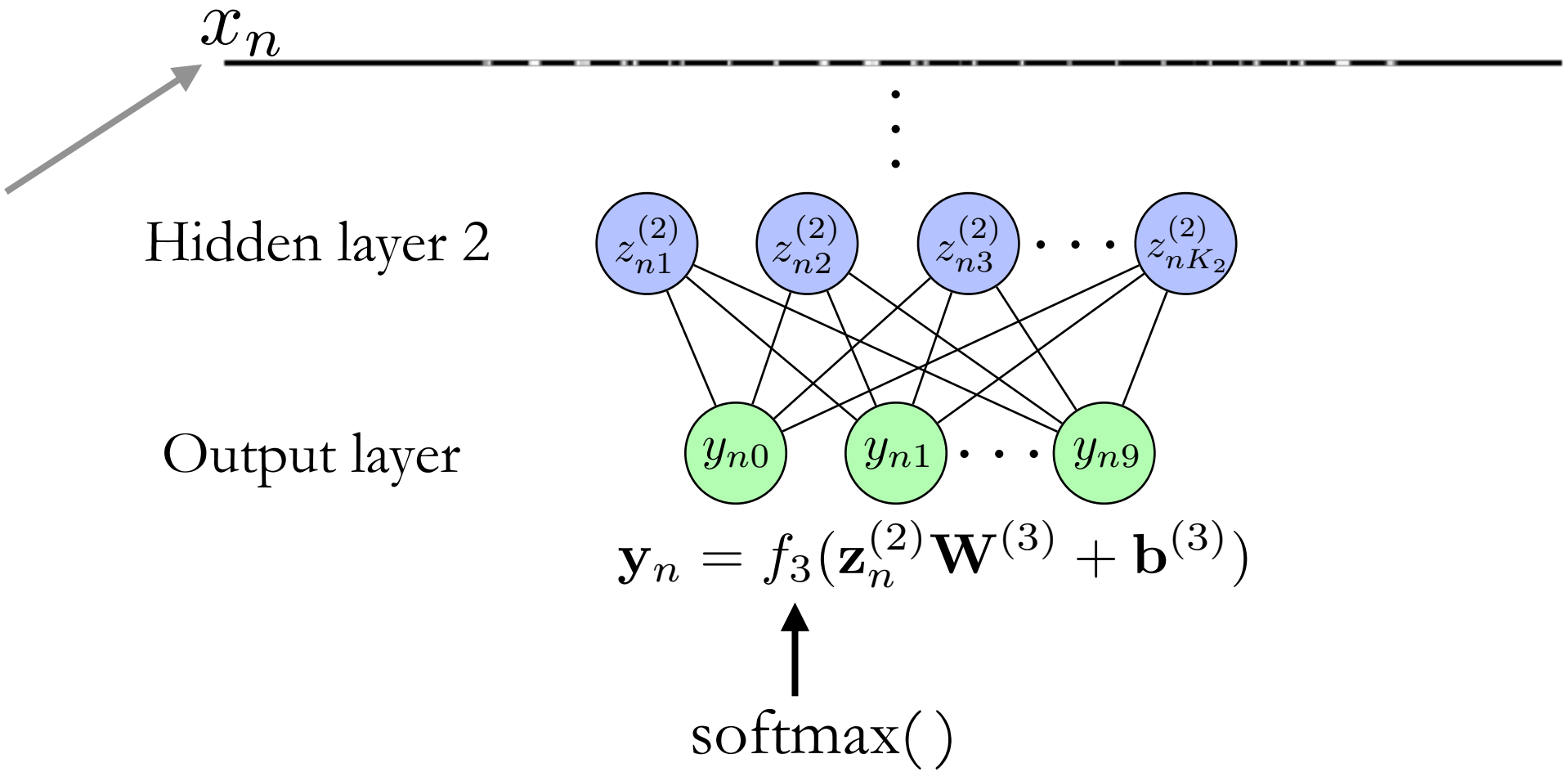
An Aside: Softmax

- The softmax is a function that inputs a vector of *reals* and outputs a *probability* vector



$$p_i = \frac{e^{\psi_i}}{\sum_j e^{\psi_j}}$$

Neural Networks



Inference

- To fit the model, we need to learn
  - The weights of all layers
  - The biases (intercepts) of all layers

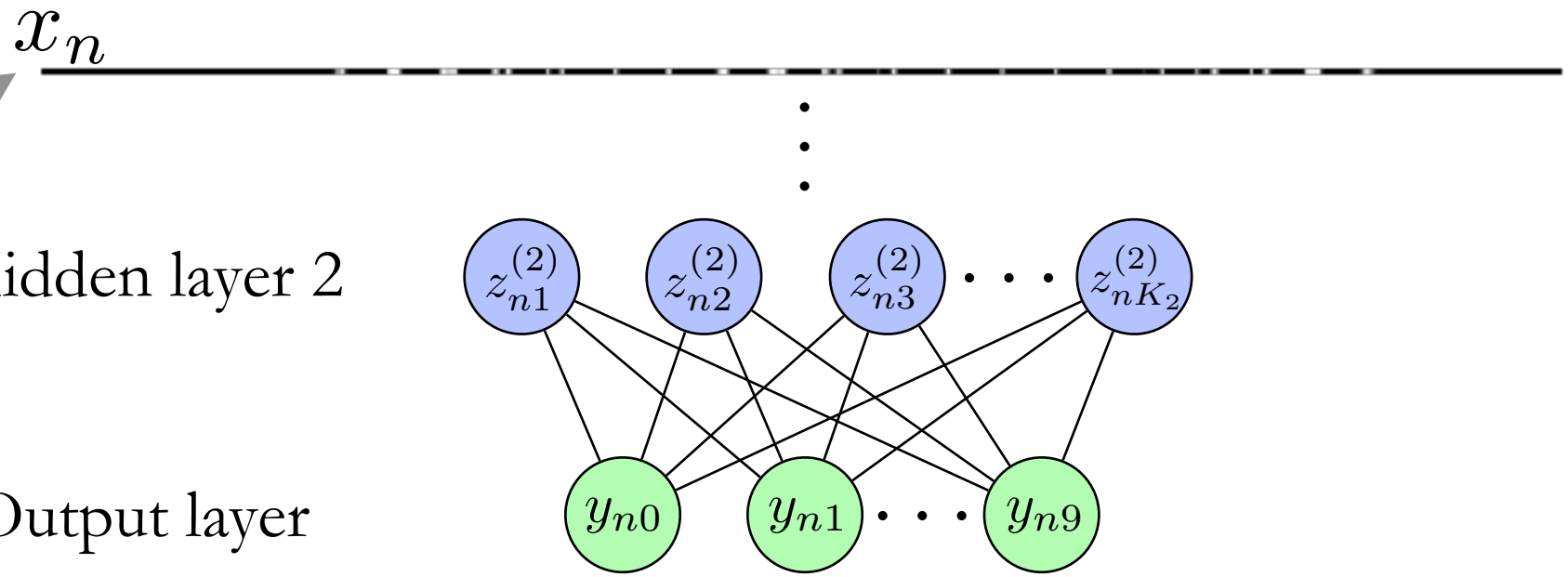
Inference

- To fit the model, we need to learn
  - The weights of all layers
  - The biases (intercepts) of all layers
- Define your error (loss) function on the training data

$$\mathcal{L} = \sum_{n=1}^{N} \log(\hat{y}_n \mid \mathbf{W}, \mathbf{b})$$

predicted probability for the observed class

Neural Networks



$$\mathcal{L} = \sum_{n=1}^{N} \log(\hat{y}_n \mid \mathbf{W}, \mathbf{b})$$

The n-th digit was #3, so $\hat{y}_n = y_{n3}$

Inference

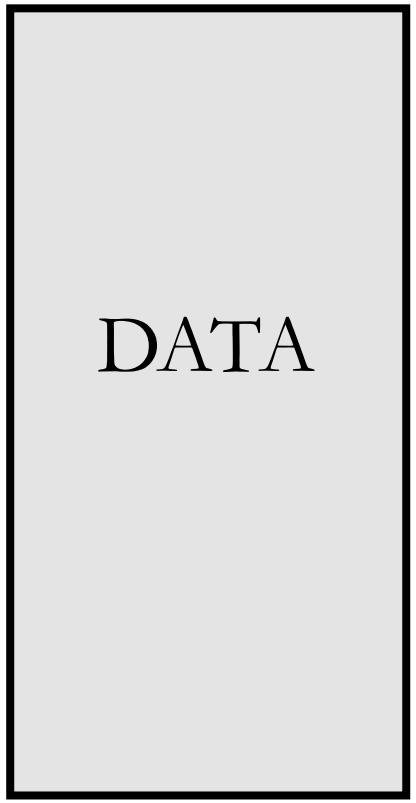
- We maximize the loss with respect to weights and biases
- Gradient ascent

$$\nabla \mathcal{L} = \sum_{n=1}^{N} \nabla \log(\hat{y}_n \mid \mathbf{W}, \mathbf{b})$$

  - Backpropagation
  - Too expensive: N can be large

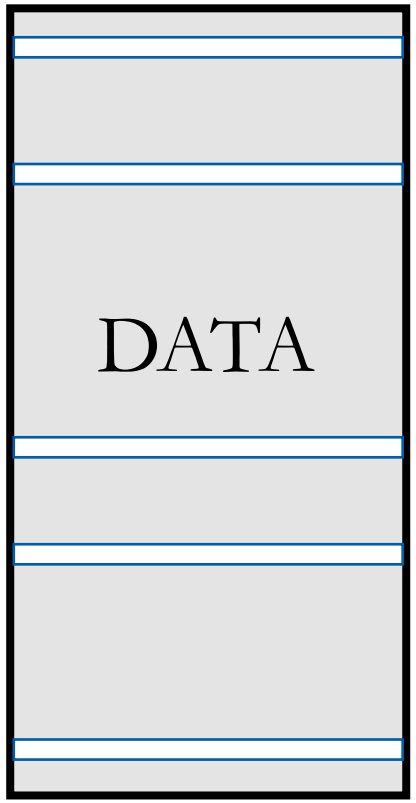
Inference: SGD

- Stochastic gradient ascent



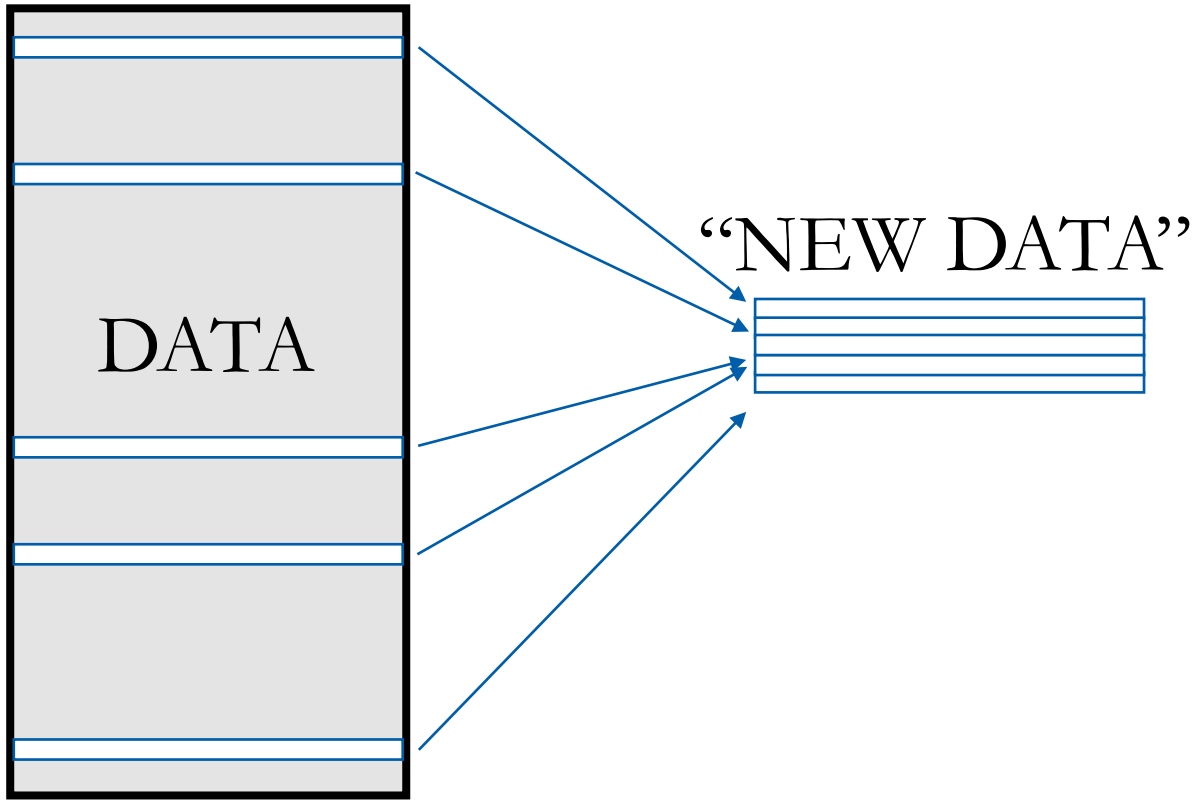
Inference: SGD

- Stochastic gradient ascent



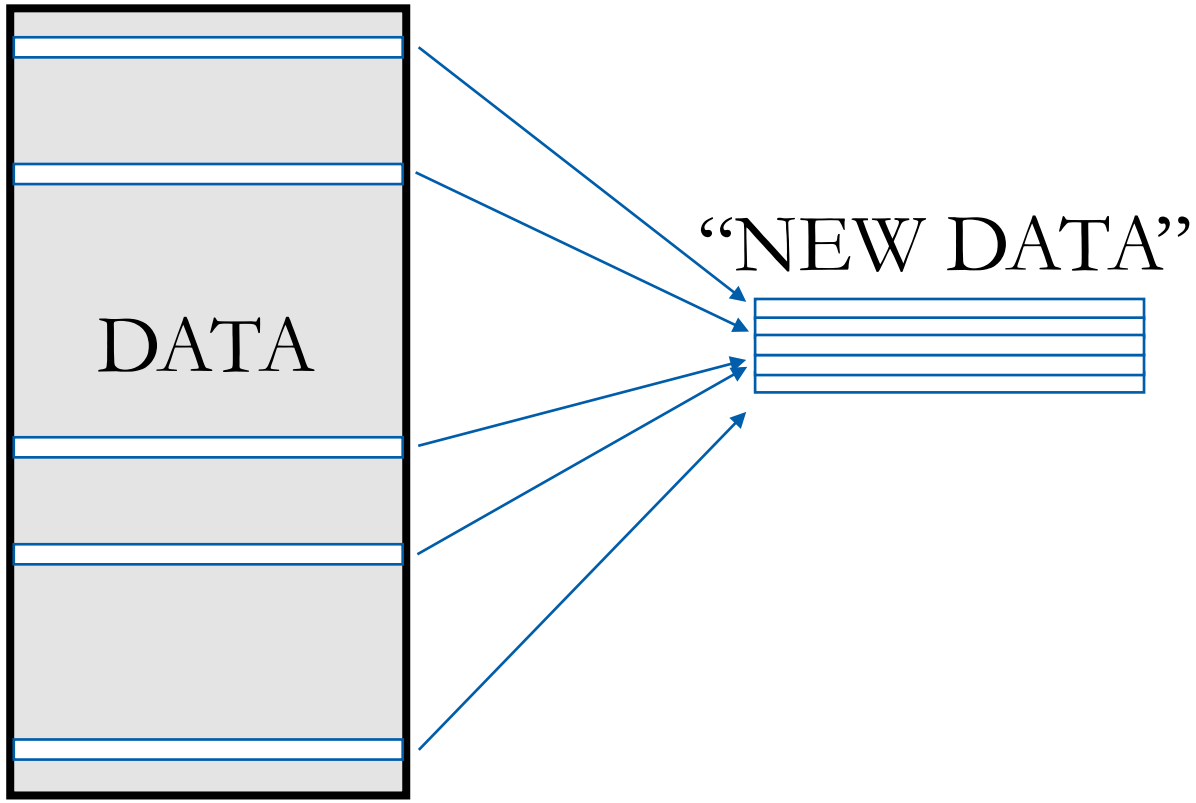
Inference: SGD

- Stochastic gradient ascent



Inference: SGD

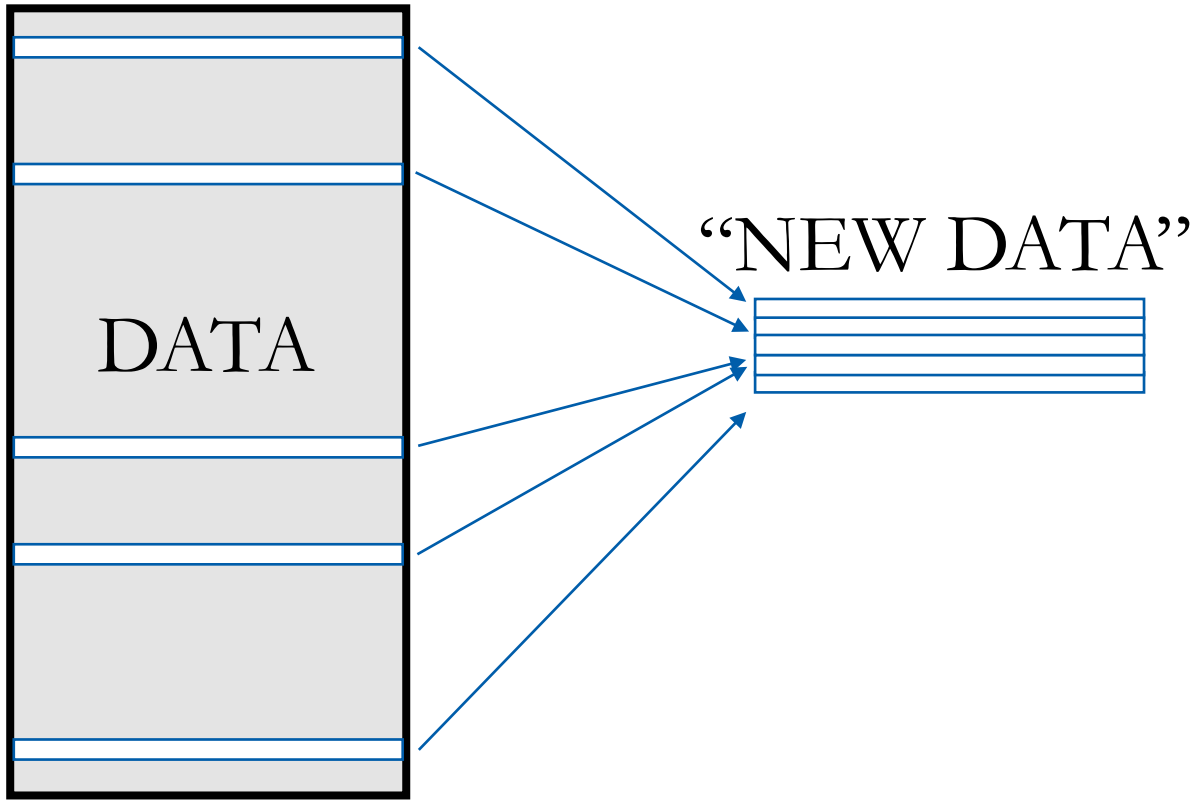
- Stochastic gradient ascent



- Subsample a **minibatch** of data
- Pretend that you dataset consists of this minibatch

Inference: SGD

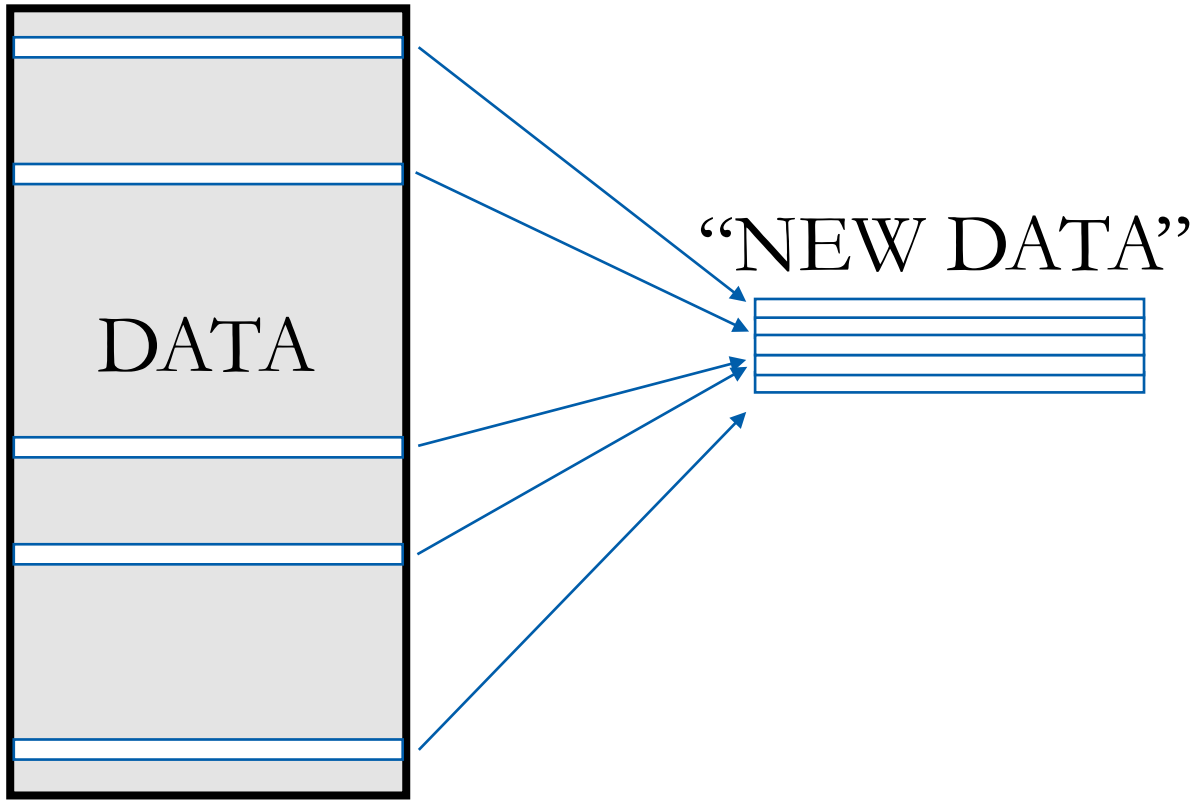
- Stochastic gradient ascent



- At each iteration of gradient descent, subsample a new minibatch
- Eventually, we end up using the entire dataset

Inference: SGD

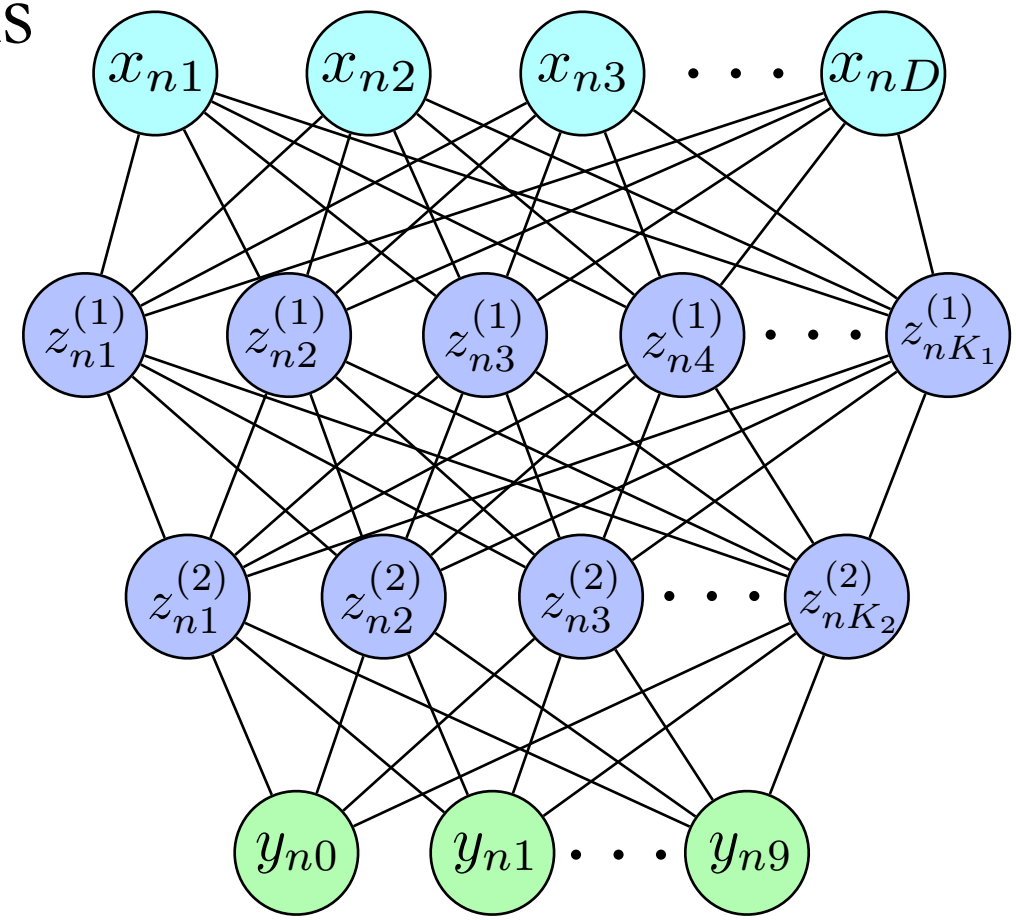
- Stochastic gradient ascent



- Requires more advanced step sizes (RMSProp, AdaGrad, Adam, …)

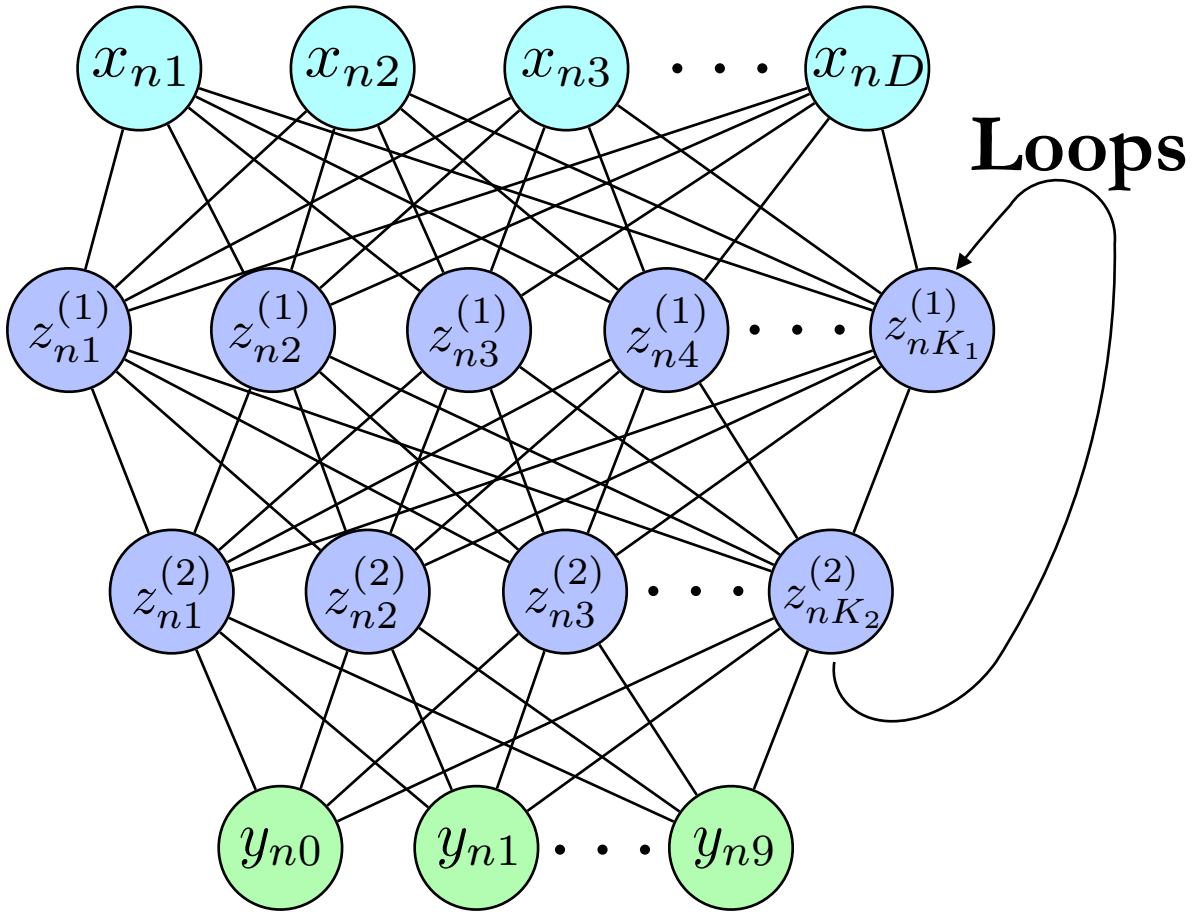
Types of Neural Networks

- **Feed-forward** neural networks
  - Fully-connected networks
  - Convolutional networks
- Recurrent neural networks



Other Types of Neural Networks

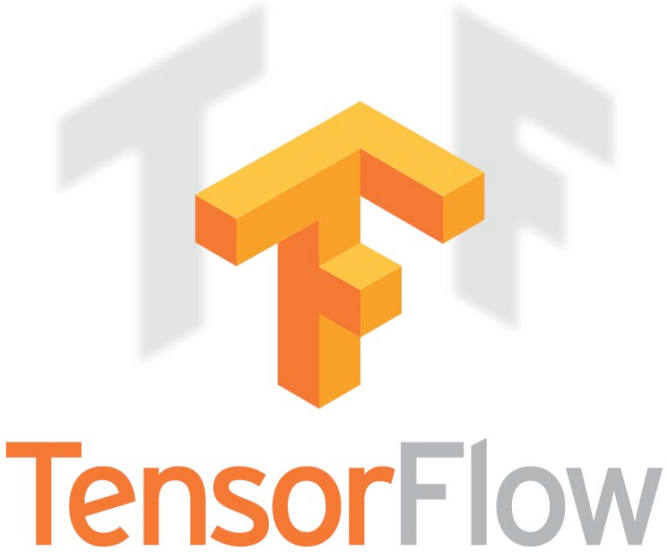
- Feed-forward neural networks
  - Fully-connected networks
  - Convolutional networks
- **Recurrent** neural networks



Criticism

- NNs are powerful and promising, but…
  1. Black-box flavor
  2. They need **a lot** of data (humans don't need that much)
  3. Do not reflect how real neurons function
  4. High computational requirements
  5. No posterior uncertainty

TensorFlow

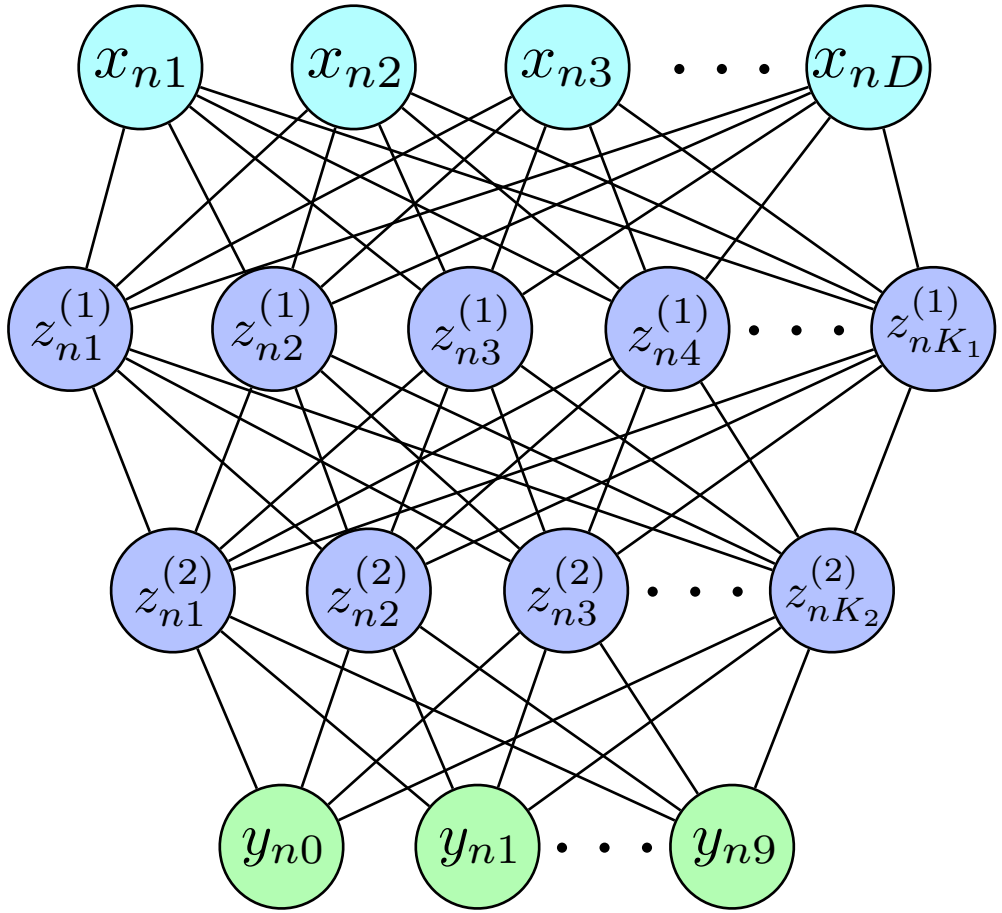


Computational Graph

- The computational graph defines:
  A. Data
  B. Variables
  C. Computations (network architecture, loss)
  D. Optimizer
  E. Other tasks (e.g., predictions on test)

Computational Graph

- The computational graph
  - does not compute anything
  - does not hold any values
  - just specifies the model and variables



Sessions

- A session allows to
  - execute graph (or part of the graph)
  - allocate memory to hold variables
  - do computations

Computational Graph & Sessions

```
import tensorflow as tf
```


Computational Graph & Sessions

```
import tensorflow as tf
```

```
# Declare a graph
graph_name = tf.Graph()
with graph_name.as_default():
    # Declare inputs, variables, computations, ...
    aux_variable = tf.Variable( tf.constant(42.0) )
```


Computational Graph & Sessions

```
import tensorflow as tf
```

```
# Declare a graph
graph_name = tf.Graph()
with graph_name.as_default():
    # Declare inputs, variables, computations, ...
    aux_variable = tf.Variable( tf.constant(42.0) )
```

```
with tf.Session(graph=graph_name) as session_name:
    # Initialize the variable
    tf.initialize_all_variables().run()
    # Print its value
    print(aux_variable.eval())
```


Placeholders

- A placeholder
  - declares a variable with no specified value
  - “promises” to specify the value later

Placeholders

```
import tensorflow as tf
```


Placeholders

```
import tensorflow as tf
```

```
# Declare a graph
graph_name = tf.Graph()
with graph_name.as_default():
    # Declare a placeholder
    aux_var1 = tf.placeholder(tf.float32)
    # Define another variable
    aux_var2 = aux_var1 + 1.0
```


Placeholders

```
import tensorflow as tf
```

```
# Declare a graph
graph_name = tf.Graph()
with graph_name.as_default():
    # Declare a placeholder
    aux_var1 = tf.placeholder(tf.float32)
    # Define another variable
    aux_var2 = aux_var1 + 1.0
```

```
with tf.Session(graph=graph_name) as session_name:
    # Create a feeder
    feed_dict = {aux_var1: 3.0}
    # Print the second variable
    print( aux_var2.eval(feed_dict=feed_dict) )
```


COLUMBIA UNIVERSITY
Data Science Institute