Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Implementing a Neural Network from Scratch

Christian Bartz, Joseph Bethge

What We Will Do With You

- tasks for this exercise
- LENGTHy introduction
- time to hack
- outlook

- at home: finish any remaining tasks until next time (two weeks)!

Why Are We Doing This?

- we want to make sure that we understand the theory
- and we can use the knowledge to implement it in practice
- we can claim to have (partially) implemented a neural network from scratch!

- use the Internet (stackoverflow, google, ...) as scarcely as possible

Preparations

- clone the prepared code from:

```
https://github.com/HPI-DeepLearning/length
```

- install all requirements with:

```
pip install -r requirements.txt
```

You have **2 minutes**.

Tasks For Today

1. data loading
2. fully connected layer
3. mean squared error
4. SGD

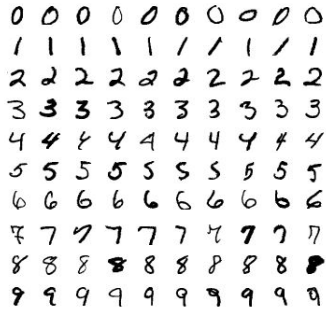
Bonus:

1. convolution including tests!
2. pooling functions (max_pooling, average_pooling) including tests!

Task 1: Data Loading (length/data_sets/mnist_like.py)

- MNIST (70 000 handwritten digits)
  - “Hello world!” equivalent of machine learning
  - classify which digit was written
- Fashion-MNIST (70 000 fashion article images)
  - “drop in” replacement
  - classify between e.g. shoe, sweater, t-shirt, ...
- how to read the data (see “THE IDX FILE FORMAT”):

  `http://yann.lecun.com/exdb/mnist/`



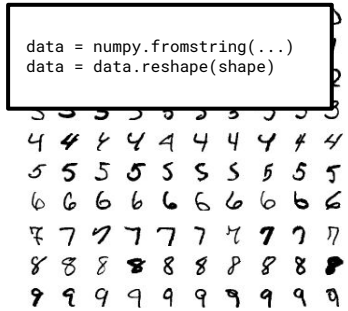
Task 1: Data Loading (length/data_sets/mnist_like.py)

can be read directly into numpy array

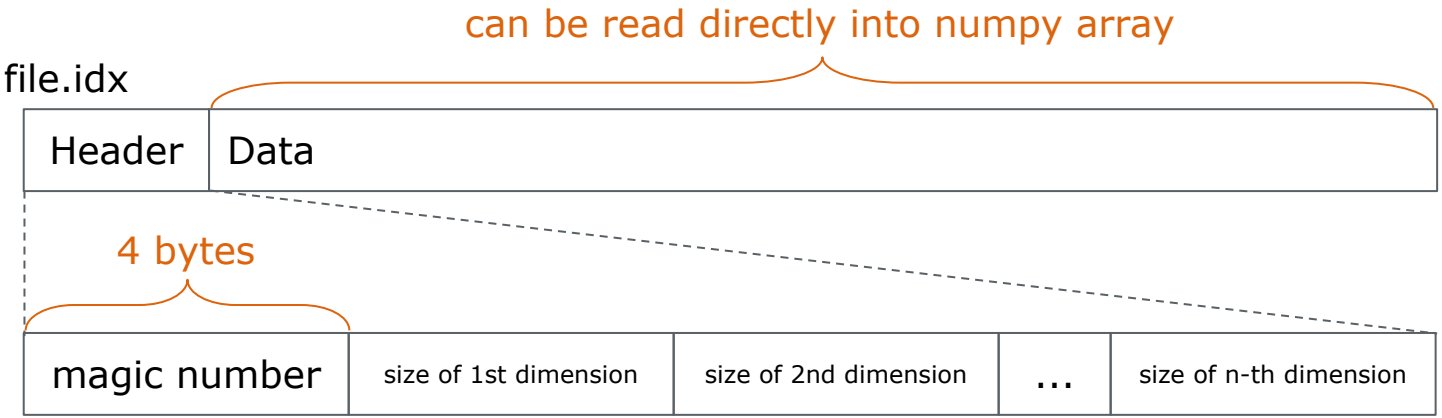
file.idx

Header	Data

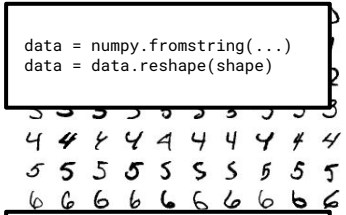
```
data = numpy.fromstring(...)
data = data.reshape(shape)
```



Task 1: Data Loading (length/data_sets/mnist_like.py)

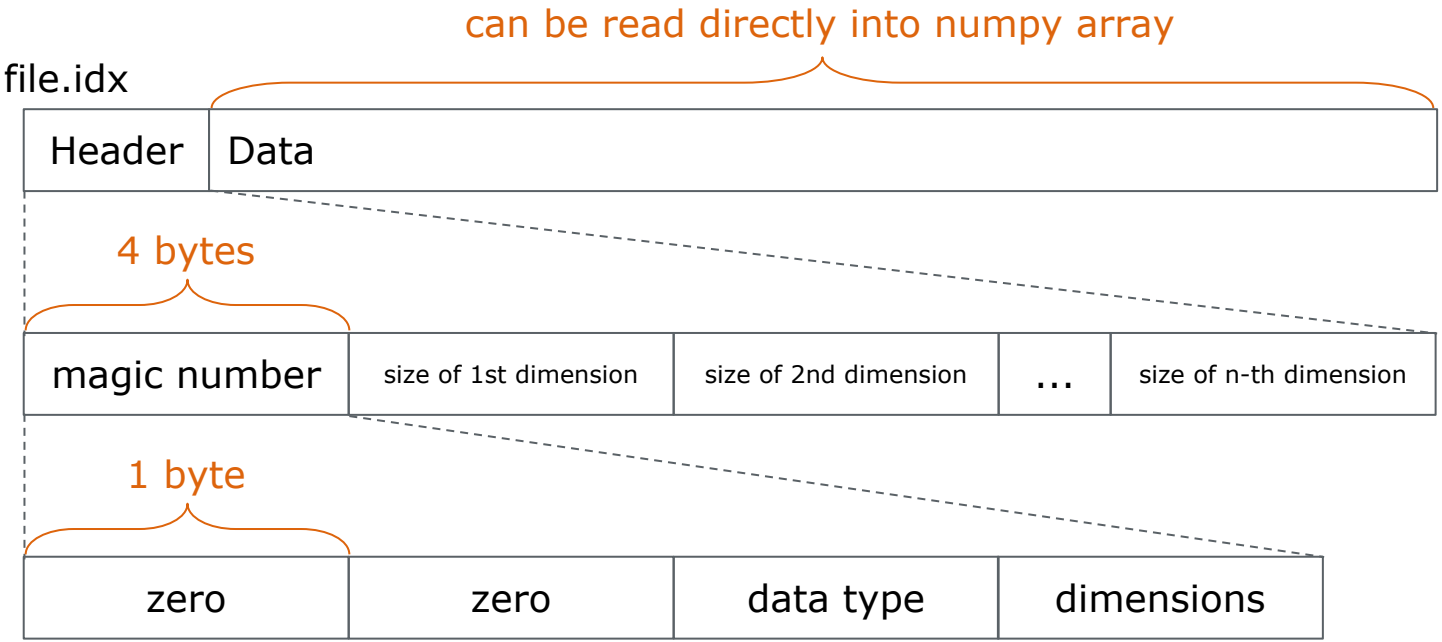



```
data = numpy.fromstring(...)
data = data.reshape(shape)
```



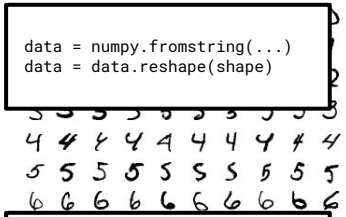
```
h = stream.read(4).hex()
i = int(h, 16)
```

Task 1: Data Loading (length/data_sets/mnist_like.py)

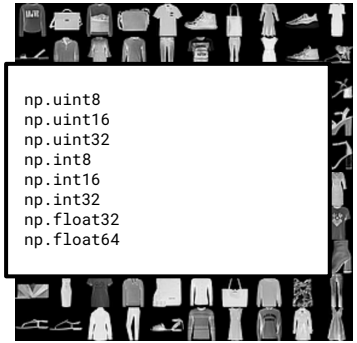


0x08: unsigned byte
0x09: signed byte
0x0B: short (2 bytes)
0x0C: int (4 bytes)
0x0D: float (4 bytes)
0x0E: double (8 bytes)

```
data = numpy.fromstring(...)
data = data.reshape(shape)
```

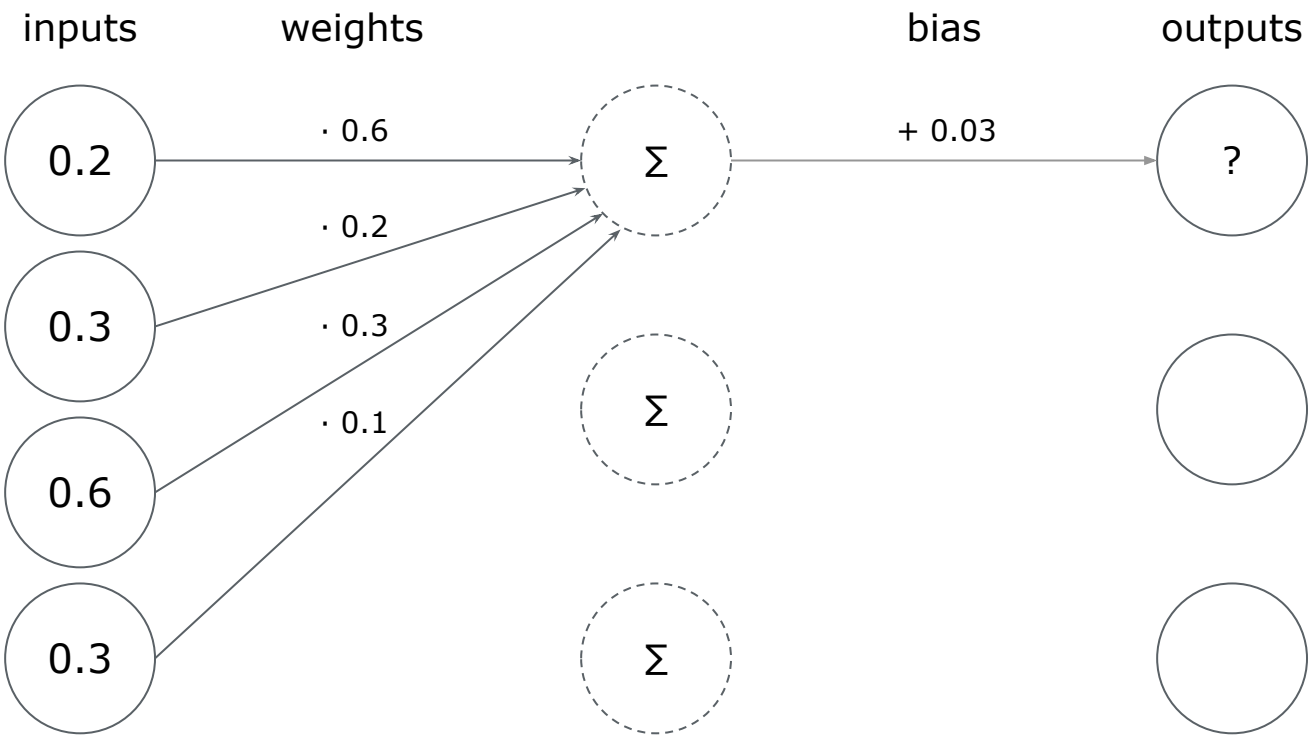


```
h = stream.read(4).hex()
i = int(h, 16)
```

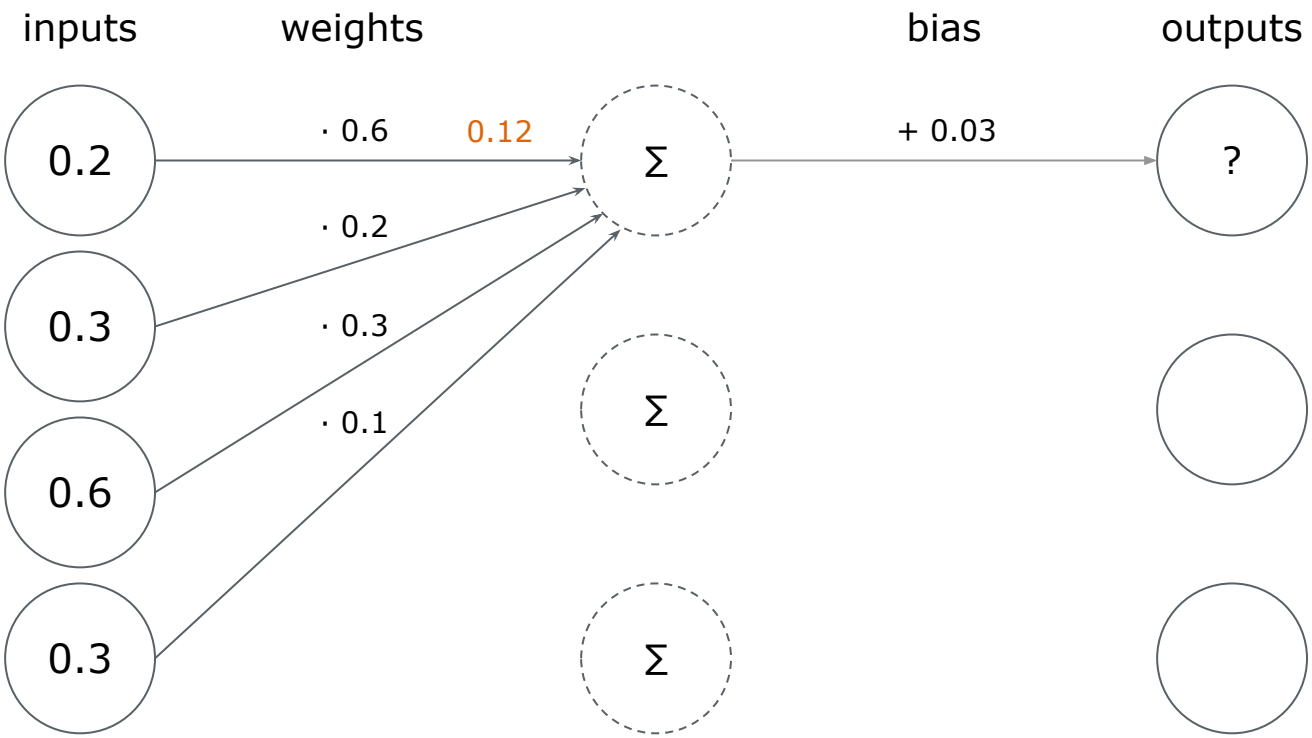


```
np.uint8
np.uint16
np.uint32
np.int8
np.int16
np.int32
np.float32
np.float64
```

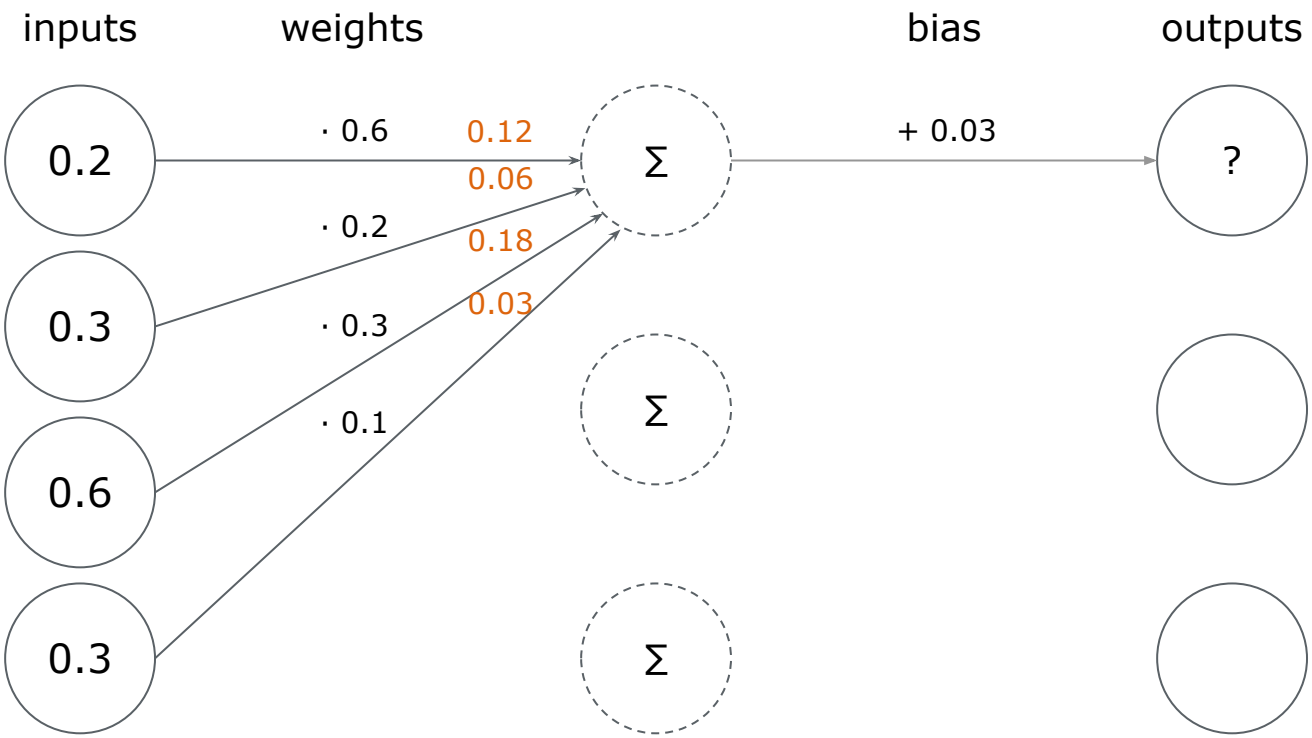
Task 2: Fully Connected Layer (length/layers/fully_connected.py)



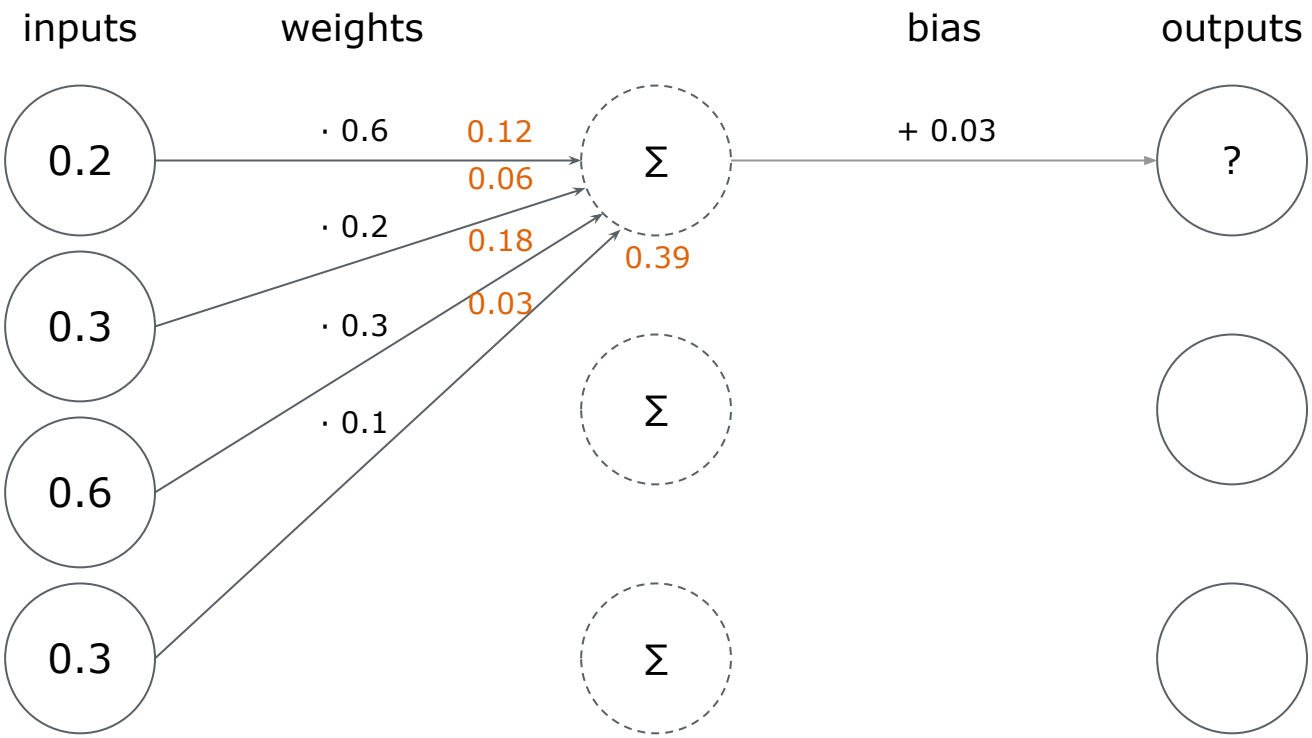
Task 2: Fully Connected Layer (length/layers/fully_connected.py)



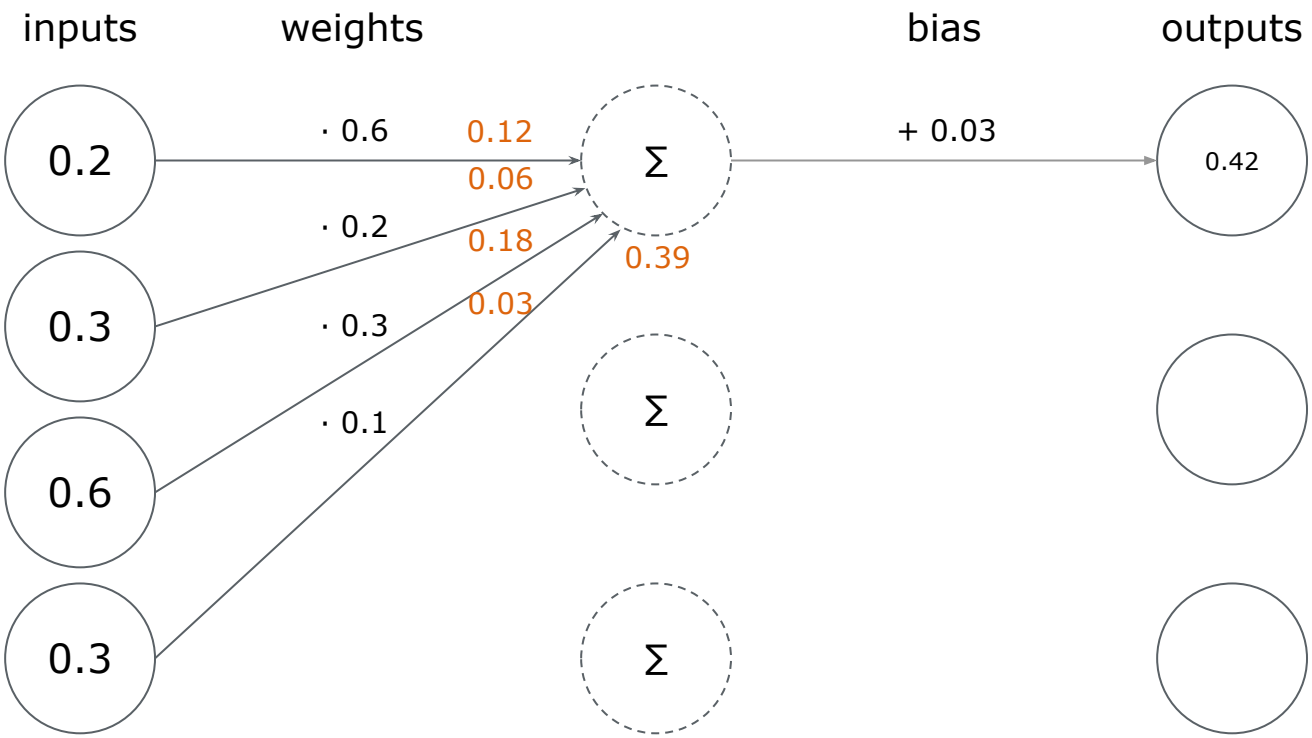
Task 2: Fully Connected Layer (length/layers/fully_connected.py)



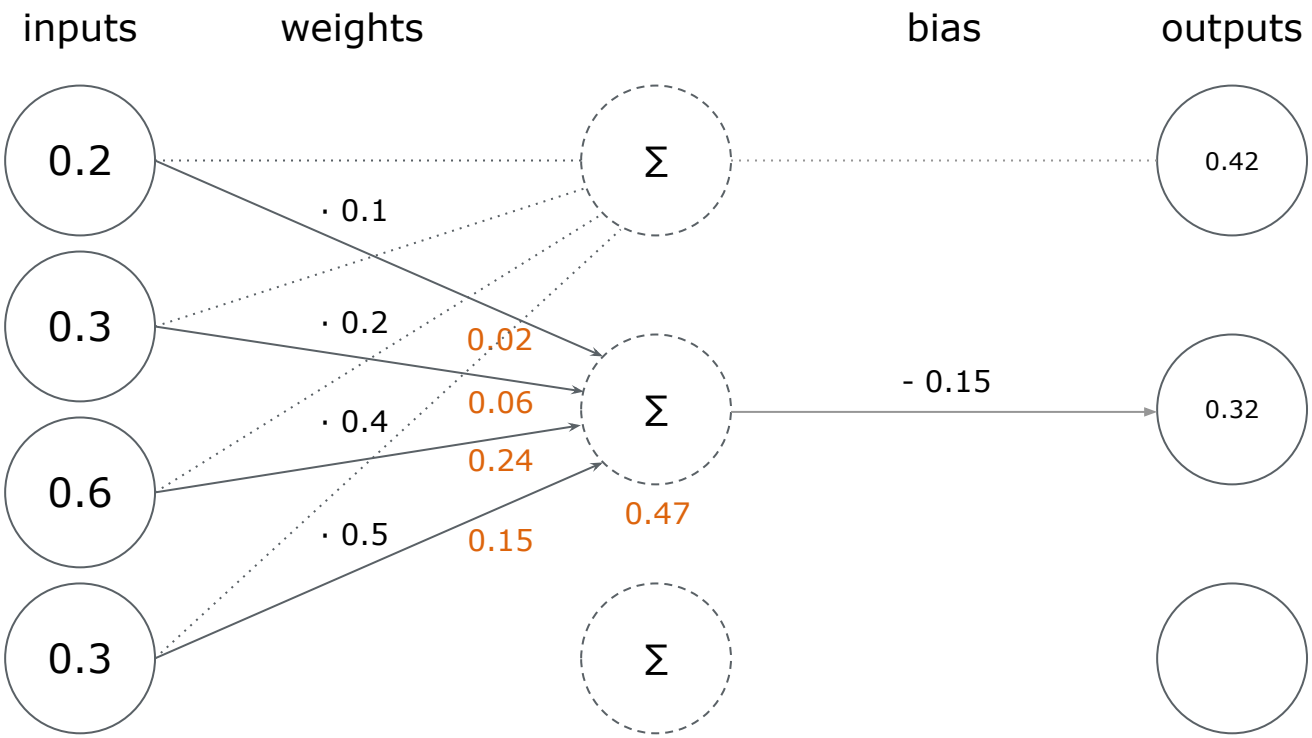
Task 2: Fully Connected Layer (length/layers/fully_connected.py)



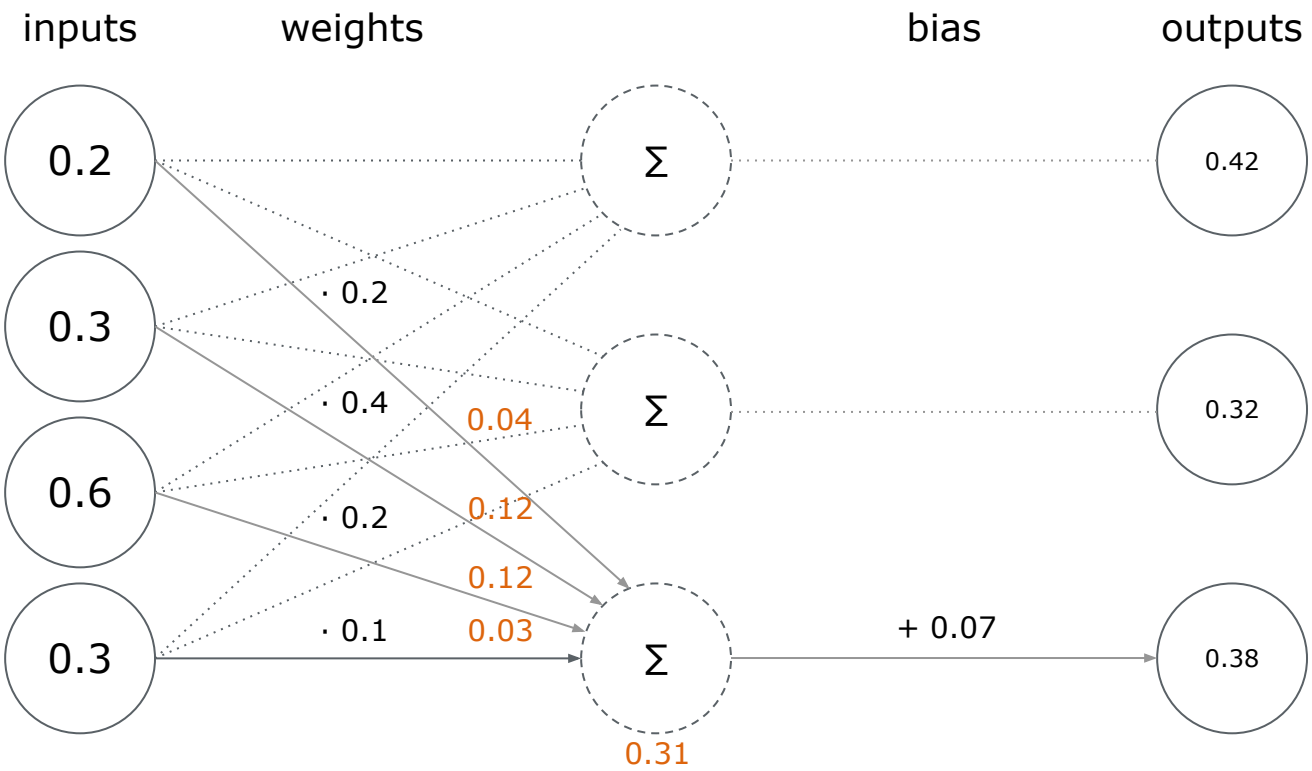
Task 2: Fully Connected Layer (length/layers/fully_connected.py)



Task 2: Fully Connected Layer (length/layers/fully_connected.py)



Task 2: Fully Connected Layer (length/layers/fully_connected.py)

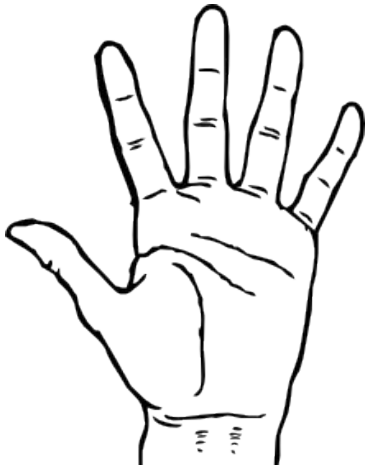


Task 2: Fully Connected Layer (length/layers/fully_connected.py)

- a fully connected layer provides dense connections between each input and output
  - parameters: `i` (number of inputs), `o` (number of outputs)
  - $\text{output}_j = \sum(\text{input}_i * \text{weight}_{i,j}) + \text{bias}_j$ over all inputs

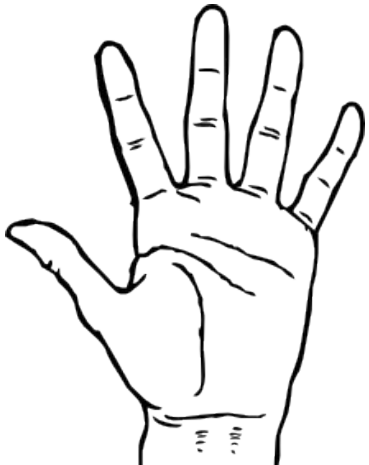
Task 2: Fully Connected Layer (length/layers/fully_connected.py)

- a fully connected layer provides dense connections between each input and output
  - parameters: `i` (number of inputs), `o` (number of outputs)
  - $\text{output}_j = \sum(\text{input}_i * \text{weight}_{i,j}) + \text{bias}_j$ over all inputs
- how many weights do we need?
- how many values for the bias do we need?
- how do we implement this efficiently?



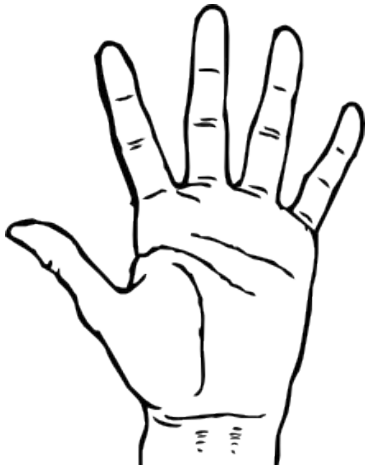
Task 2: Fully Connected Layer (length/layers/fully_connected.py)

- a fully connected layer provides dense connections between each input and output
  - parameters: `i` (number of inputs), `o` (number of outputs)
  - $\text{output}_j = \sum(\text{input}_i * \text{weight}_{i,j}) + \text{bias}_j$ over all inputs
- how many weights do we need? `i*o`
- how many values for the bias do we need?
- how do we implement this efficiently?



Task 2: Fully Connected Layer (length/layers/fully_connected.py)

- a fully connected layer provides dense connections between each input and output
  - parameters: `i` (number of inputs), `o` (number of outputs)
  - $\text{output}_j = \sum(\text{input}_i * \text{weight}_{i,j}) + \text{bias}_j$ over all inputs
- how many weights do we need? `i*o`
- how many values for the bias do we need? `o`
- how do we implement this efficiently?



Task 2: Fully Connected Layer (length/layers/fully_connected.py)

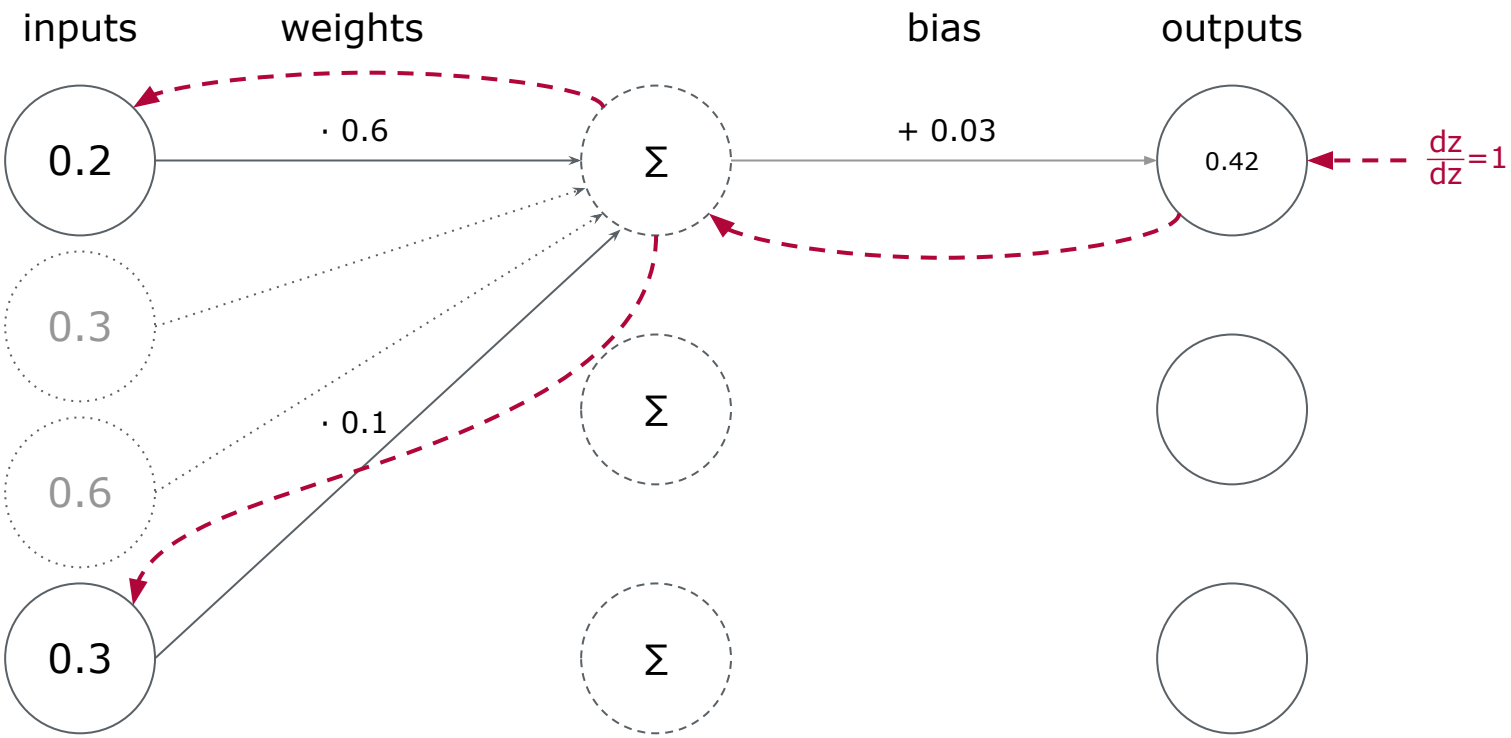
- a fully connected layer provides dense connections between each input and output
  - parameters: `i` (number of inputs), `o` (number of outputs)
  - $\text{output}_j = \sum(\text{input}_i * \text{weight}_{i,j}) + \text{bias}_j$ over all inputs
- how many weights do we need? `i*o`
- how many values for the bias do we need? `o`
- how do we implement this efficiently? dot product

Task 2: Fully Connected Layer (length/layers/fully_connected.py)

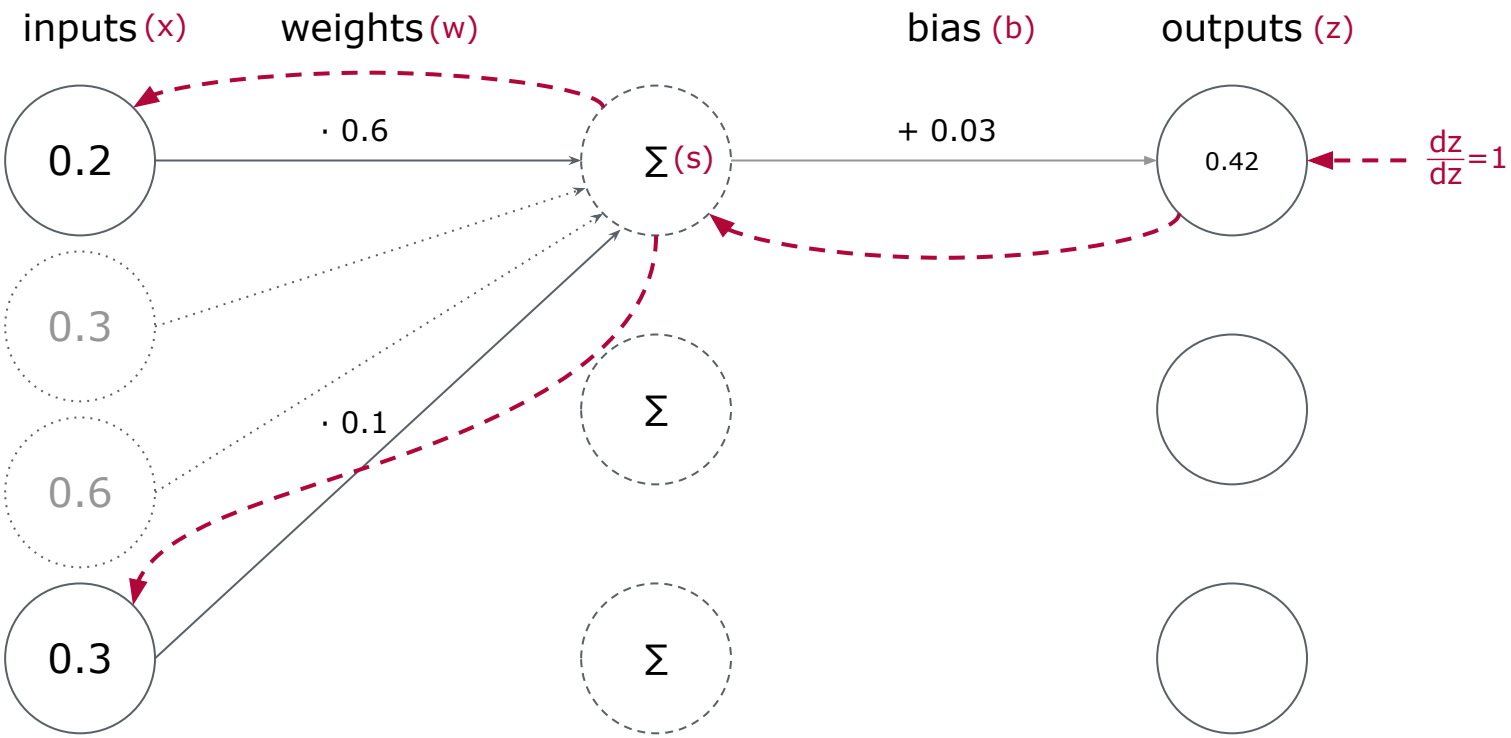
- a fully connected layer provides dense connections between each input and output
  - parameters: `i` (number of inputs), `o` (number of outputs)
  - $\text{output}_j = \sum(\text{input}_i * \text{weight}_{i,j}) + \text{bias}_j$ over all inputs
- how many weights do we need? `i*o`
- how many values for the bias do we need? `o`
- how do we implement this efficiently? dot product

- **watch out:** batch processing!

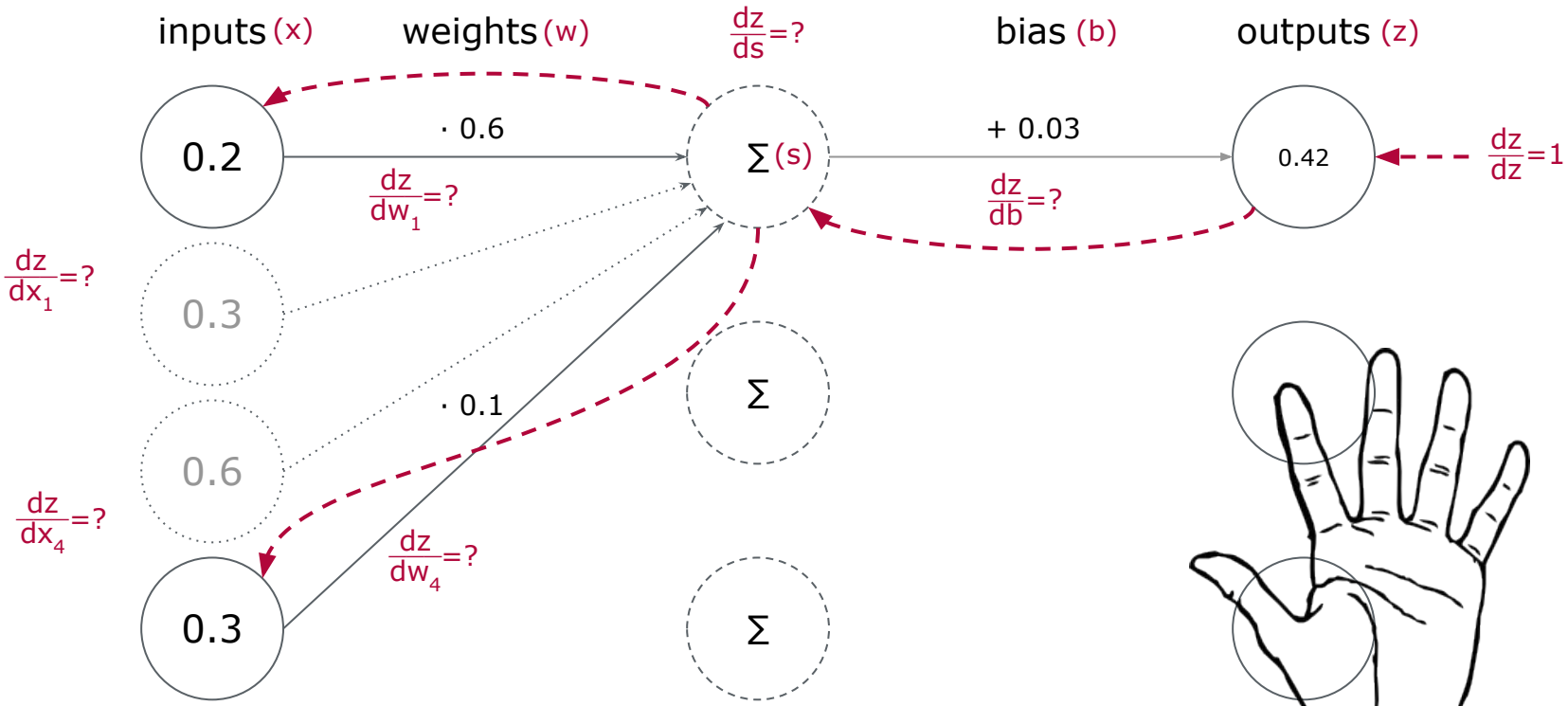
Task 2: Fully Connected Layer (length/layers/fully_connected.py)



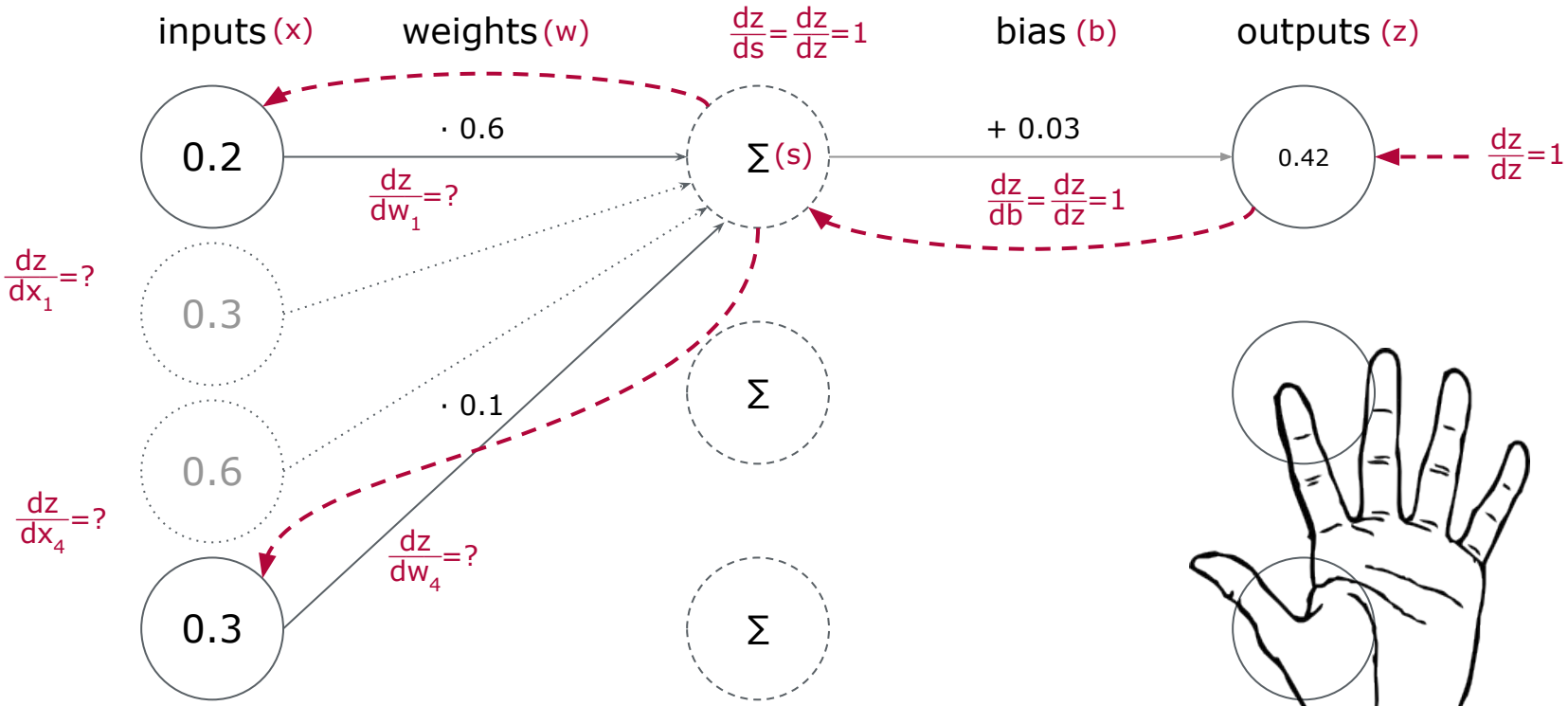
Task 2: Fully Connected Layer (length/layers/fully_connected.py)



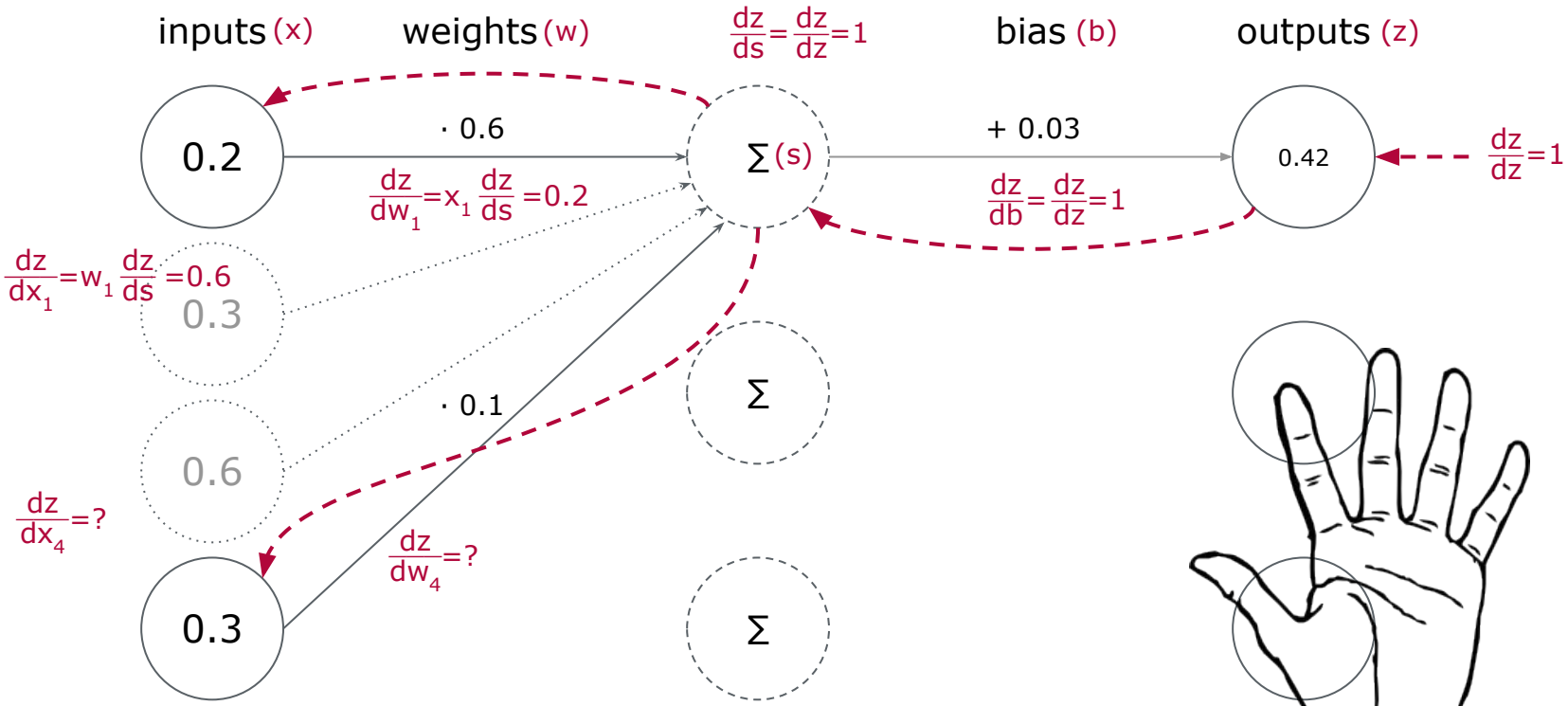
Task 2: Fully Connected Layer (length/layers/fully_connected.py)



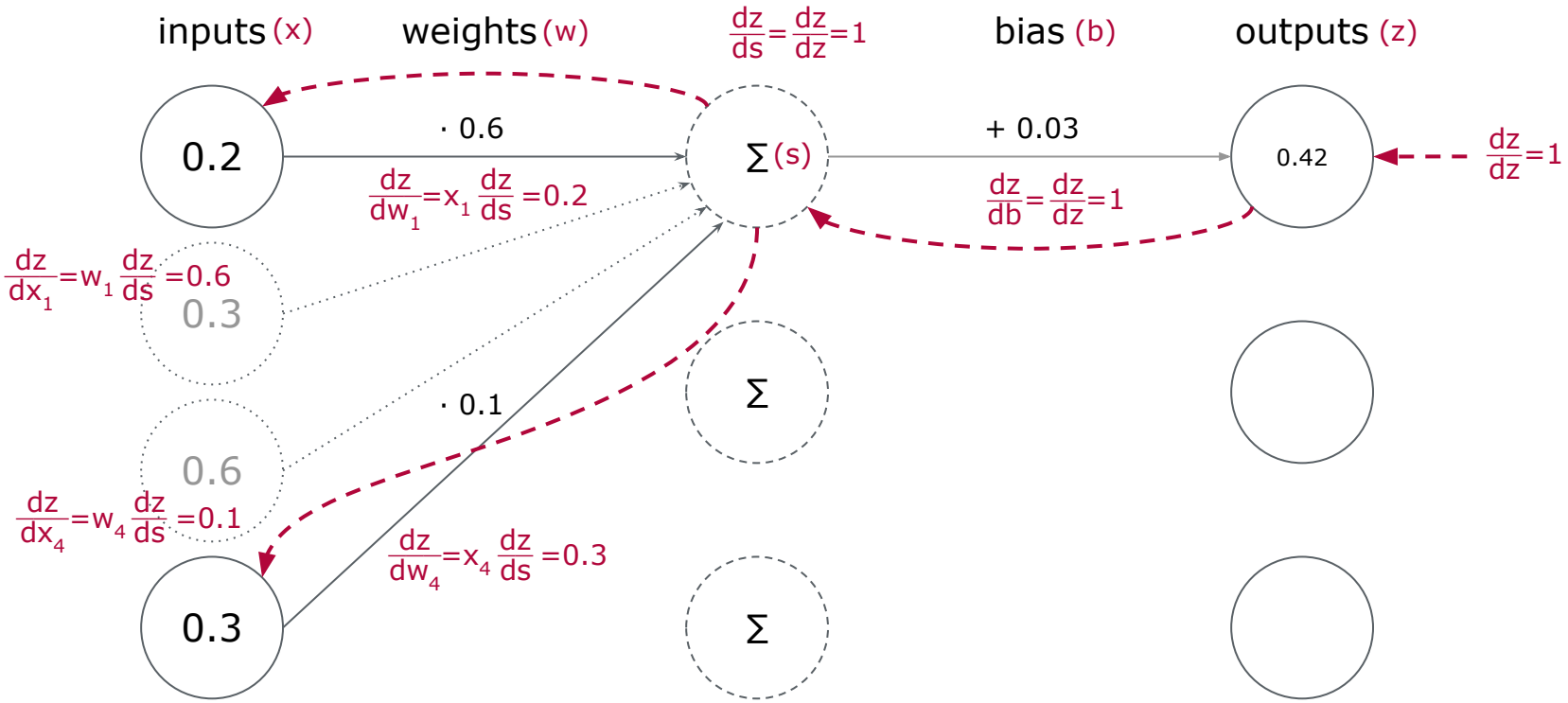
Task 2: Fully Connected Layer (length/layers/fully_connected.py)



Task 2: Fully Connected Layer (length/layers/fully_connected.py)



Task 2: Fully Connected Layer (length/layers/fully_connected.py)



Task 2: Fully Connected Layer (length/layers/fully_connected.py)

- a fully connected layer provides dense connections between each input and output
  - parameters: `i` (number of inputs), `o` (number of outputs)

    $\text{output}_j = \sum(\text{input}_i * \text{weight}_{i,j} + \text{bias}_j)$ over all inputs
- backward pass:
  - need to compute values for $\frac{dL}{dx}$, $\frac{dL}{dw}$, and $\frac{dL}{db}$
  - first we compute $\frac{dz}{dx}$, $\frac{dz}{dw}$, and $\frac{dz}{db}$, then use chain rule
- **hint:** we can use dot product again, but what could be the input to that?
- **watch out:** batch processing and chain rule!

$$\frac{dL}{dv} = \frac{dL}{dz}\frac{dz}{dv}$$

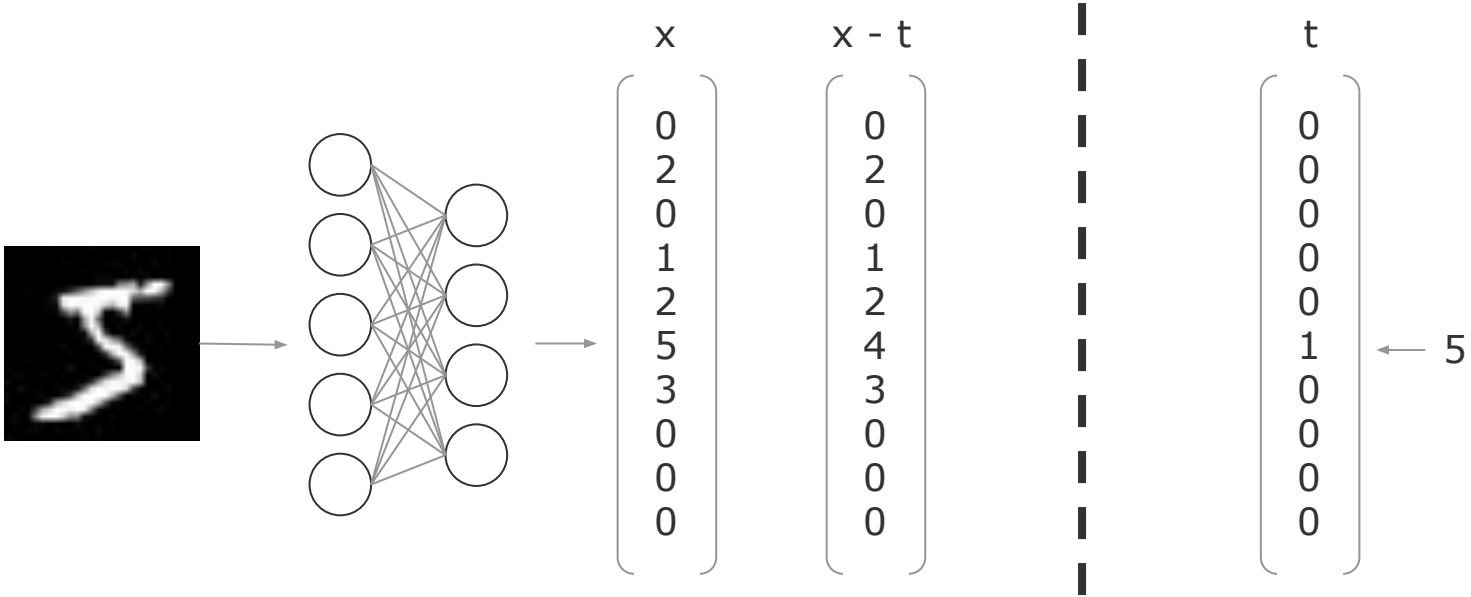
Task 3: Mean Squared Error (MSE) (length/functions/mean_squared_error.py)

- MSE computes the dissimilarity of two given vectors
- provides a measure for the quality of our prediction → loss

$$MSE = \frac{1}{n}\sum_{1}^{n}(x - t)^2$$

- **watch out:** two Inputs! (x and t)

Task 3: Mean Squared Error (MSE) (length/functions/mean_squared_error.py)



Task 3: Mean Squared Error (MSE) (length/functions/mean_squared_error.py)

x	t	x - t	$(x - t)^2$	$sum((x - t)^2)$	$mean(sum((x - t)^2))$
0	0	0	0		
2	0	2	4		
0	0	0	0		
1	0	1	1		
2	0	2	4	34	3.4
5	1	4	16		
3	0	3	9		
0	0	0	0		
0	0	0	0		
0	0	0	0		

Task 3: Mean Squared Error (MSE) (length/functions/mean_squared_error.py)

$$MSE = \frac{1}{n}\sum_{1}^{n}(x-t)^2$$

Backpropagation

- we need to compute

$$\frac{\partial MSE}{\partial x} \quad \text{and} \quad \frac{\partial MSE}{\partial t}$$

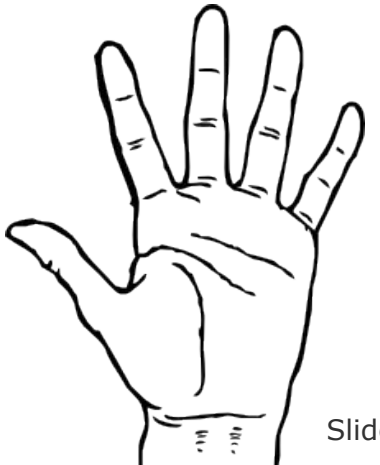
Task 4: Stochastic Gradient Descent (SGD) (length/optimizers/sgd.py)

- the layers provide us their calculated gradients for each input
  - we know how they influence the result (loss)
  - we want the loss to be close to zero
- how do we do this?

$\frac{dL}{dx_1}=0.12$

$\frac{dL}{dx_2}=-0.32$

$\frac{dL}{dx_3}=0.97$



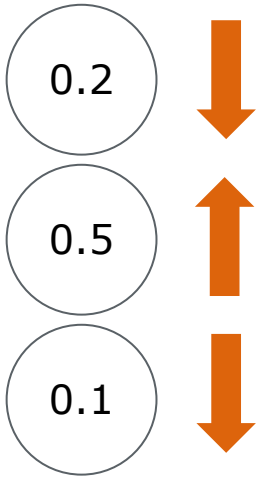
Task 4: Stochastic Gradient Descent (SGD) (length/optimizers/sgd.py)

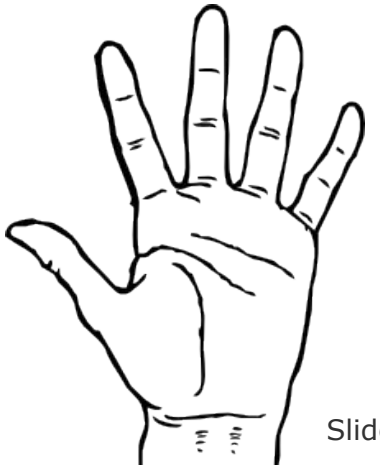
- the layers provide us their calculated gradients for each input
  - we know how they influence the result (loss)
  - we want the loss to be close to zero
- how do we do this?

$\frac{dL}{dx_1}=0.12$

$\frac{dL}{dx_2}=-0.32$

$\frac{dL}{dx_3}=0.97$





Task 4: Stochastic Gradient Descent (SGD) (length/optimizers/sgd.py)

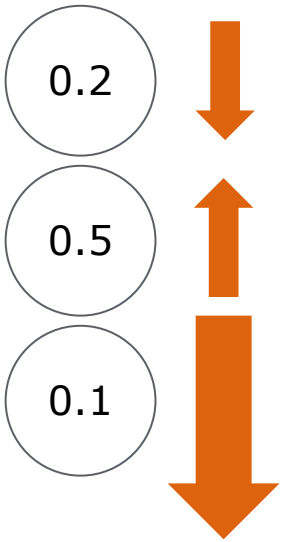
- the layers provide us their calculated gradients for each input
  - we know how they influence the result (loss)
  - we want the loss to be close to zero
- how do we do this?
- scale (multiply) with the learning rate!

- **hint:** also check again how you implemented the update method in your fully connected layer

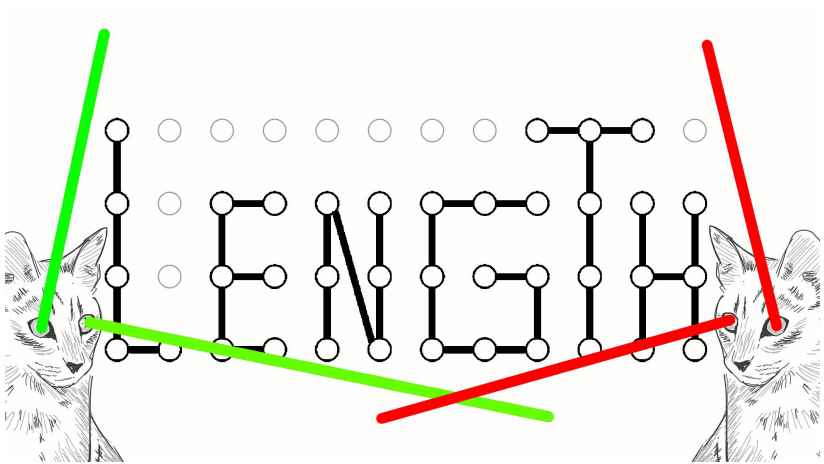
$\frac{dL}{dx_1}=0.12$

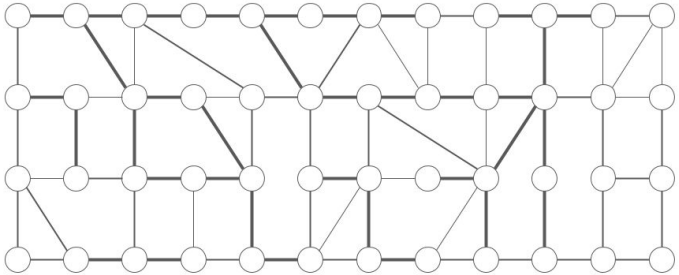
$\frac{dL}{dx_2}=-0.32$

$\frac{dL}{dx_3}=0.97$



Hasso Plattner Institut
Digital Engineering · Universität Potsdam

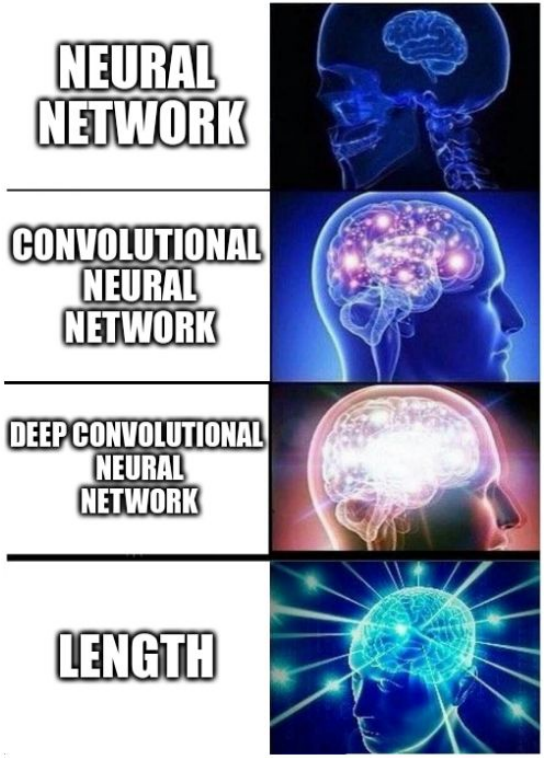




LENGTH - **L**ightning-fast **E**xtensible **N**eural-network **G**uarding **T**he **H**PI

LENGTH - Introduction

- very simple neural network implementation based on Chainer
  - entirely written in Python using Numpy
  - simple, object oriented API
  - uses dynamic computational graph



LENGTH - Functions (length/function.py and length/functions/add.py)

- functions are base building blocks for neural network in LENGTH

```python
from length.function import Function

class Add(Function):
    name = "Add"

    def internal_forward(self, inputs):
        x, y = inputs
        return x + y,

    def internal_backward(self, inputs, gradients):
        gradient, = gradients
        return gradient, gradient
```

LENGTH - Functions (length/function.py and length/functions/add.py)

- functions are base building blocks for neural network in LENGTH

```python
from length.function import Function

class Add(Function):
    name = "Add"

    def internal_forward(self, inputs):
        x, y = inputs
        return x + y,

    def internal_backward(self, inputs, gradients):
        gradient, = gradients
        return gradient, gradient
```

```
>>> c = add(a, b)
>>> c.visualize()
id | layer       | next
 1 | input (1,) |    3
 2 | input (1,) |    3
 3 | Add (1,)   |    4
```

LENGTH - Functions (length/function.py and length/functions/add.py)

- functions are base building blocks for neural network in LENGTH

```python
from length.function import Function

class Add(Function):
    name = "Add"

    def internal_forward(self, inputs):
        x, y = inputs
        return x + y,

    def internal_backward(self, inputs, gradients):
        gradient, = gradients
        return gradient, gradient
```

inputs = tuple of numpy arrays

returns a tuple of outputs
→ in this case the element-wise sum

LENGTH - Functions (length/function.py and length/functions/add.py)

- functions are base building blocks for neural network in LENGTH

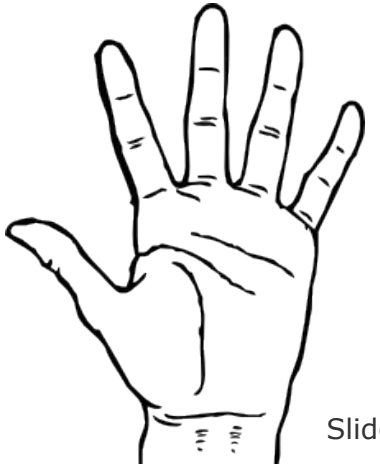
```python
from length.function import Function

class Add(Function):
    name = "Add"

    def internal_forward(self, inputs):
        x, y = inputs
        return x + y,

    def internal_backward(self, inputs, gradients):
        gradient, = gradients
        return gradient, gradient
```

inputs = tuple of numpy arrays
gradients = tuple of numpy array, representing input gradients from other functions

returns a tuple of outputs
→ in this case the gradient with respect to each input

LENGTH - Layers

- a layer is a function that has to keep track of optimizable parameters
- can you think of an example for a layer?



LENGTH - Layers

- a layer is a function that has to keep track of optimizable parameters
- can you think of an example for a layer?
- FullyConnected is a layer
  - contains optimizable parameters **`W`** and **`b`**

LENGTH - Layers (length/layer.py and length/layers/)

```
class Layer(Function):
    needs_optimizer = True
    name = "Layer"

    def internal_update(self, parameter_deltas):
        raise NotImplementedError
```

updates internal parameters (like W and b) with the deltas computed by the optimizer

LENGTH - Optimizer (length/optimizer.py)

```
class Optimizer:

    def run_update_rule(self, gradients, layer):
        """
        Does the actual optimization step and calculates the update
        :param gradients: the gradients of the layer that are to be used to optimize the parameters
        (a tuple of numpy arrays)
        :param layer: the layer to which the gradients belong
        :return: the deltas that are to be applied to the parameters
        """
        raise NotImplementedError
```

called during backward pass, if we compute gradients for a layer

LENGTH - Model (length/models/mlp.py)

```
class MLP:
    def __init__(self):
        self.fully_connected_1 = FullyConnected(784, 512)
        self.fully_connected_2 = FullyConnected(512, 512)
        self.fully_connected_3 = FullyConnected(512, 10)
        self.loss = None
        self.predictions = None

    def forward(self, batch, train=True):
        hidden = self.fully_connected_1(batch.data)
        hidden = self.fully_connected_2(hidden)
        self.predictions = self.fully_connected_3(hidden)
        self.loss = F.mean_squared_error(self.predictions, batch.labels)

    def backward(self, optimizer):
        self.loss.backward(optimizer)
```

LENGTH - Model (length/models/mlp.py)

```
class MLP:
    def __init__(self):
        self.fully_connected_1 = FullyConnected(784, 512)
        self.fully_connected_2 = FullyConnected(512, 512)
        self.fully_connected_3 = FullyConnected(512, 10)
        self.loss = None
        self.predictions = None

    def forward(self, batch, train=True):
        hidden = self.fully_connected_1(batch.data)
        hidden = self.fully_connected_2(hidden)
        self.predictions = self.fully_connected_3(hidden)
        self.loss = F.mean_squared_error(self.predictions, batch.labels)

    def backward(self, optimizer):
        self.loss.backward(optimizer)
```

create all layers we want to train and initialize them

LENGTH - Model (length/models/mlp.py)

```
class MLP:
    def __init__(self):
        self.fully_connected_1 = FullyConnected(784, 512)
        self.fully_connected_2 = FullyConnected(512, 512)
        self.fully_connected_3 = FullyConnected(512, 10)
        self.loss = None
        self.predictions = None

    def forward(self, batch, train=True):
        hidden = self.fully_connected_1(batch.data)
        hidden = self.fully_connected_2(hidden)
        self.predictions = self.fully_connected_3(hidden)
        self.loss = F.mean_squared_error(self.predictions, batch.labels)

    def backward(self, optimizer):
        self.loss.backward(optimizer)
```

do forward pass with given data batch and compute loss

LENGTH - Model (length/models/mlp.py)

```
class MLP:
    def __init__(self):
        self.fully_connected_1 = FullyConnected(784, 512)
        self.fully_connected_2 = FullyConnected(512, 512)
        self.fully_connected_3 = FullyConnected(512, 10)
        self.loss = None
        self.predictions = None

    def forward(self, batch, train=True):
        hidden = self.fully_connected_1(batch.data)
        hidden = self.fully_connected_2(hidden)
        self.predictions = self.fully_connected_3(hidden)
        self.loss = F.mean_squared_error(self.predictions, batch.labels)

    def backward(self, optimizer):
        self.loss.backward(optimizer)
```

do backward pass → compute gradients for each function we applied and also optimize parameters of each layer.

Task Overview - Time to Hack!

1. data loading
   - `length/data_sets/mnist_like.py`
2. fully connected layer
   - `length/layers/fully_connected.py`
3. mean squared error
   - `length/functions/mean_squared_error.py`
4. stochastic gradient descent
   - `length/optimizers/sgd.py`

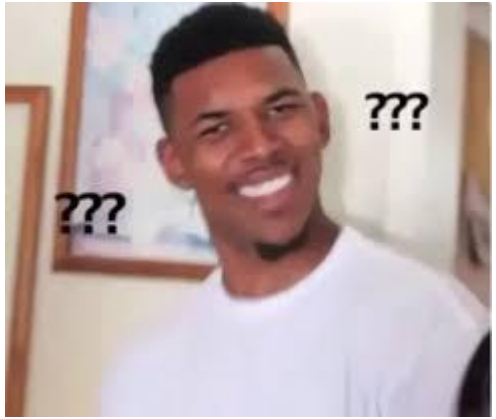
```
$ python train.py
train: epoch:  0, loss:  0.10, accuracy 0.19, iteration:  900
running test set...
 test: epoch:  0, loss:  0.10, accuracy 0.20
train: epoch:  1, loss:  0.09, accuracy 0.34, iteration:  900
running test set...
 test: epoch:  1, loss:  0.09, accuracy 0.35
train: epoch:  2, loss:  0.09, accuracy 0.48, iteration:  900
running test set...
 test: epoch:  2, loss:  0.09, accuracy 0.47
train: epoch:  3, loss:  0.09, accuracy 0.50, iteration:  900
running test set...
 test: epoch:  3, loss:  0.09, accuracy 0.53
train: epoch:  4, loss:  0.09, accuracy 0.53, iteration:  900
running test set...
 test: epoch:  4, loss:  0.09, accuracy 0.57
train: epoch:  5, loss:  0.09, accuracy 0.55, iteration:  900
running test set...
 test: epoch:  5, loss:  0.09, accuracy 0.59
train: epoch:  6, loss:  0.09, accuracy 0.55, iteration:  900
running test set...
 test: epoch:  6, loss:  0.09, accuracy 0.60
train: epoch:  7, loss:  0.09, accuracy 0.56, iteration:  900
running test set...
 test: epoch:  7, loss:  0.09, accuracy 0.62
train: epoch:  8, loss:  0.09, accuracy 0.58, iteration:  900
running test set...
 test: epoch:  8, loss:  0.09, accuracy 0.63
train: epoch:  9, loss:  0.09, accuracy 0.61, iteration:  900
running test set...
 test: epoch:  9, loss:  0.09, accuracy 0.63
```

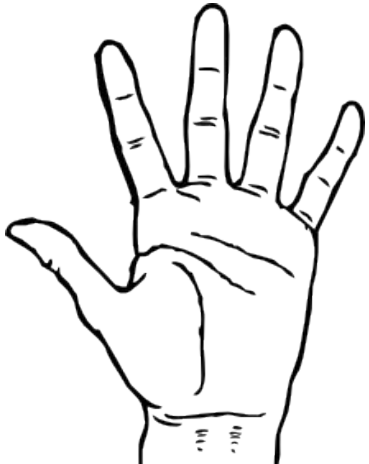
Run test with: `pytest`

Run actual training: `python train.py` (this will only work after you are finished)

Debriefing

- why are our results pretty bad?





```
$ python train.py
train: epoch:  0, loss:  0.10, accuracy 0.19, iteration:  900
running test set...
 test: epoch:  0, loss:  0.10, accuracy 0.20
train: epoch:  1, loss:  0.09, accuracy 0.34, iteration:  900
running test set...
 test: epoch:  1, loss:  0.09, accuracy 0.35
train: epoch:  2, loss:  0.09, accuracy 0.48, iteration:  900
running test set...
 test: epoch:  2, loss:  0.09, accuracy 0.47
train: epoch:  3, loss:  0.09, accuracy 0.50, iteration:  900
running test set...
 test: epoch:  3, loss:  0.09, accuracy 0.53
train: epoch:  4, loss:  0.09, accuracy 0.53, iteration:  900
running test set...
 test: epoch:  4, loss:  0.09, accuracy 0.57
train: epoch:  5, loss:  0.09, accuracy 0.55, iteration:  900
running test set...
 test: epoch:  5, loss:  0.09, accuracy 0.59
train: epoch:  6, loss:  0.09, accuracy 0.55, iteration:  900
running test set...
 test: epoch:  6, loss:  0.09, accuracy 0.60
train: epoch:  7, loss:  0.09, accuracy 0.56, iteration:  900
running test set...
 test: epoch:  7, loss:  0.09, accuracy 0.62
train: epoch:  8, loss:  0.09, accuracy 0.58, iteration:  900
running test set...
 test: epoch:  8, loss:  0.09, accuracy 0.63
train: epoch:  9, loss:  0.09, accuracy 0.61, iteration:  900
running test set...
 test: epoch:  9, loss:  0.09, accuracy 0.63
```

Debriefing

- why are our results pretty bad?
  - linear functions only
  - initialization
  - SGD is not necessarily optimal
  - length of training
  - number of weights?

```
$ python train.py
train: epoch:  0, loss:  0.10, accuracy 0.19, iteration:  900
running test set...
 test: epoch:  0, loss:  0.10, accuracy 0.20
train: epoch:  1, loss:  0.09, accuracy 0.34, iteration:  900
running test set...
 test: epoch:  1, loss:  0.09, accuracy 0.35
train: epoch:  2, loss:  0.09, accuracy 0.48, iteration:  900
running test set...
 test: epoch:  2, loss:  0.09, accuracy 0.47
train: epoch:  3, loss:  0.09, accuracy 0.50, iteration:  900
running test set...
 test: epoch:  3, loss:  0.09, accuracy 0.53
train: epoch:  4, loss:  0.09, accuracy 0.53, iteration:  900
running test set...
 test: epoch:  4, loss:  0.09, accuracy 0.57
train: epoch:  5, loss:  0.09, accuracy 0.55, iteration:  900
running test set...
 test: epoch:  5, loss:  0.09, accuracy 0.59
train: epoch:  6, loss:  0.09, accuracy 0.55, iteration:  900
running test set...
 test: epoch:  6, loss:  0.09, accuracy 0.60
train: epoch:  7, loss:  0.09, accuracy 0.56, iteration:  900
running test set...
 test: epoch:  7, loss:  0.09, accuracy 0.62
train: epoch:  8, loss:  0.09, accuracy 0.58, iteration:  900
running test set...
 test: epoch:  8, loss:  0.09, accuracy 0.63
train: epoch:  9, loss:  0.09, accuracy 0.61, iteration:  900
running test set...
 test: epoch:  9, loss:  0.09, accuracy 0.63
```

Next Time

We will improve our network with non-linear functions and more.

1. Initialization
2. Sigmoid
3. ReLU
4. Adam
5. Dropout

Send an email or visit us anytime with questions!

Christian: christian.bartz@hpi.de H-1.11

Joseph: joseph.bethge@hpi.de H-1.21

Bitte bringen Sie die Studenten dazu den Raum zu verlassen, um die Präsentation zu beenden.

All Tasks

1. Data Loading
2. Initialization
3. Fully Connected Layer
4. Mean Squared Error
5. SGD
6. Sigmoid
7. ReLU
8. Adam
9. Dropout

Bonus Bonus:

1. tanh

Gleich kommen Backup Slides!

Task 3: Mean Squared Error (MSE) (length/functions/mean_squared_error.py)

Watch out: x is a vector!

$$MSE = \frac{1}{n}\sum_{1}^{n}(x-t)^2$$

$$\frac{\partial MSE}{\partial x_i} = \frac{1}{n}(x_i - t_i)^2$$

$$= \frac{2}{n}(x_i - t_i) \cdot 1$$

Watch out: outer and inner derivative

Task 3: Mean Squared Error (MSE) (length/functions/mean_squared_error.py)

Watch out: t is a vector!

$$MSE = \frac{1}{n}\sum_{1}^{n}(x-t)^2$$

$$\frac{\partial MSE}{\partial t_i} = \frac{1}{n}(x_i - t_i)^2$$

$$= \frac{2}{n}(x_i - t_i) \cdot -1$$

Watch out: outer and inner derivative