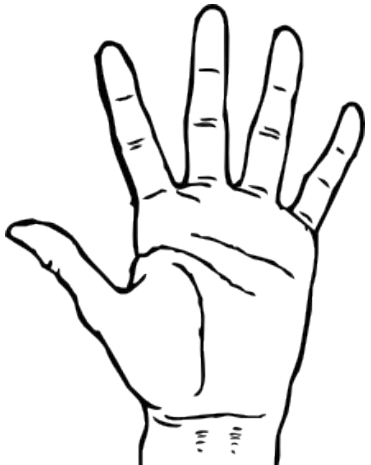
Implementing a Neural Network from Scratch #2

Christian Bartz, Joseph Bethge

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

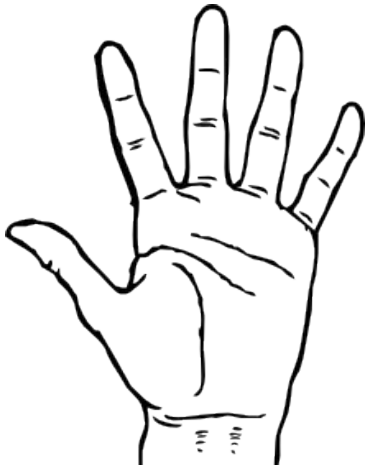
Last Exercise: Feedback

- how long did it take?



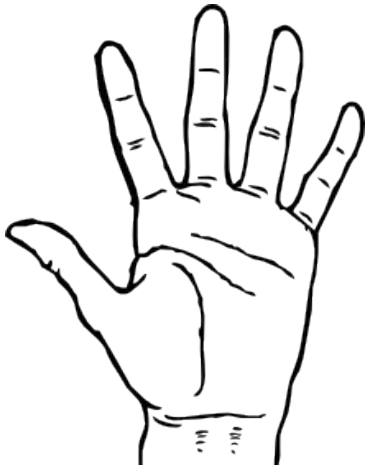
Last Exercise: Feedback

- how long did it take?
- most difficult/easy task?



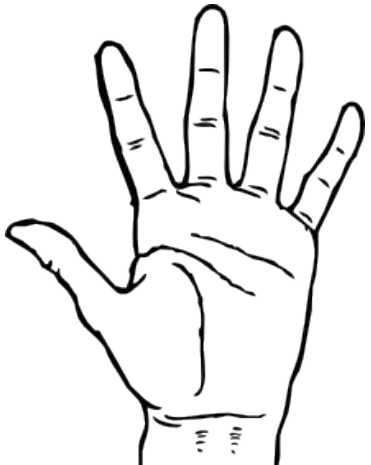
Last Exercise: Feedback

- how long did it take?
- most difficult/easy task?
- favorite/most disliked task?



Last Exercise: Feedback

- how long did it take?
- most difficult/easy task?
- favorite/most disliked task?
- more suggestions, comments?



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 (three weeks)!

Prepare your Environment

- we added some tests for today's tasks
- stash your changes

  ```
  git stash
  ```

- fetch the updates from Github

  ```
  git fetch
  ```

- rebase your current branch on our master

  ```
  git rebase origin/master
  ```

- apply stash

  ```
  git stash apply
  ```

Tasks for today

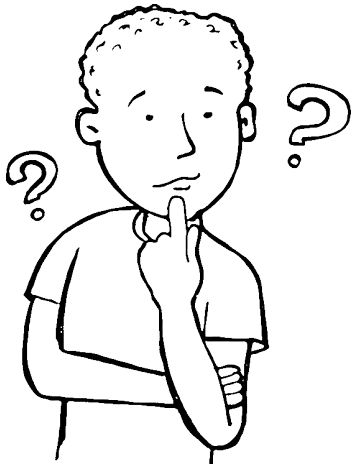
1. Initialization - Christian
2. Sigmoid - Joseph
3. ReLU - Christian
4. Adam - Joseph
5. Dropout - Christian

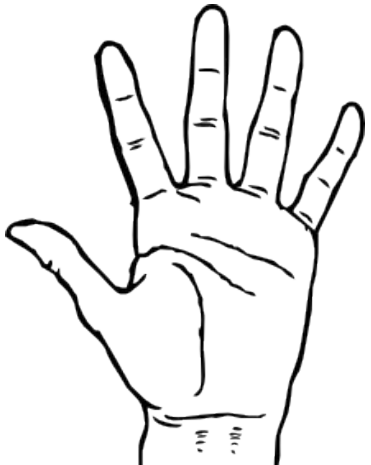
Bonus:

1. Convolution including tests!
2. Pooling functions (max_pooling, average_pooling) including tests!
3. Tanh

Task 1: Initialization

Should we initialize a network with zeros everywhere?

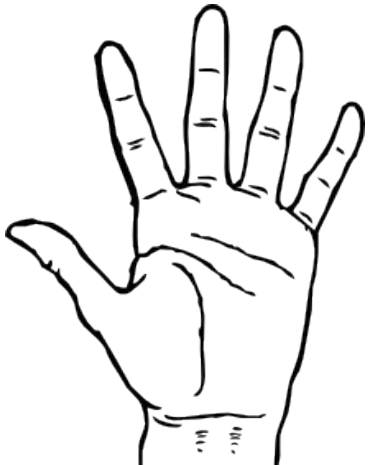




Task 1: Initialization (length/initializers/xavier.py)

- it is important to have a good initialization
  - allows convergence
  - enables faster convergence
- why do we care about initialization and don't just take:

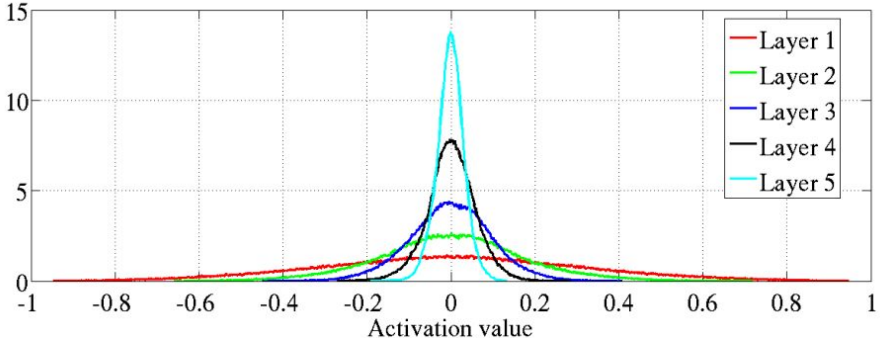
`W = np.random.randn(n)` with n being the number of inputs?



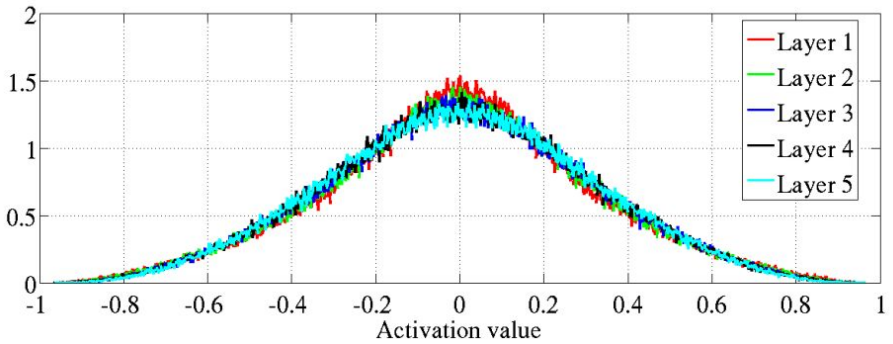
Task 1: Initialization (length/initializers/xavier.py)

- using the right initialization we get evenly distributed activations
  - makes training easier
  - mitigates saturation of activation functions and vanishing gradient

naive initialization



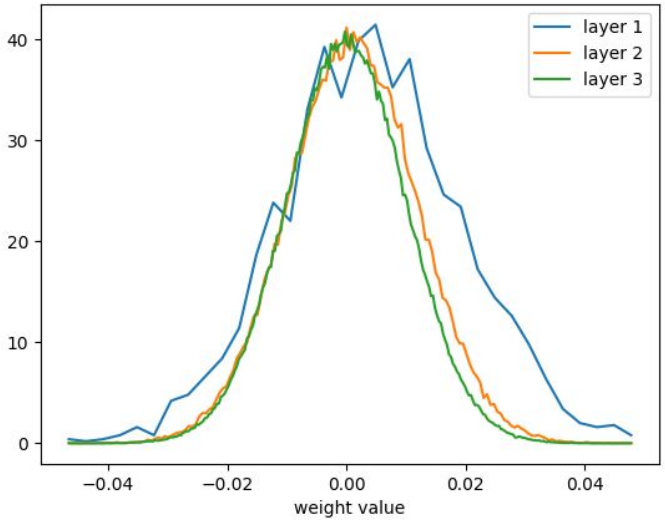
normalized initialization



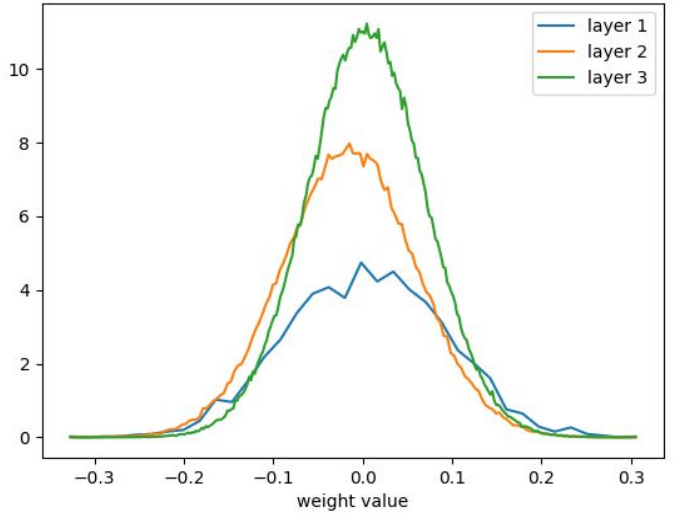
Task 1: Initialization (length/initializers/xavier.py)

- we can get evenly distributed activation values by scaling the random weights: $\sqrt{\frac{2}{n_{in}+n_{out}}}$

naive initialization

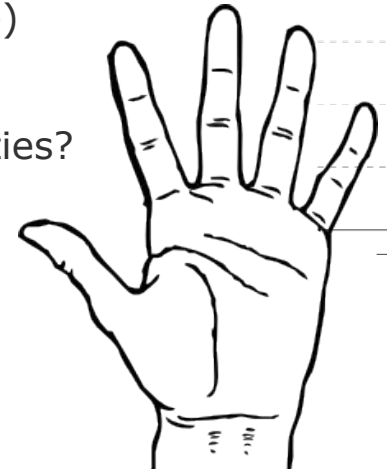


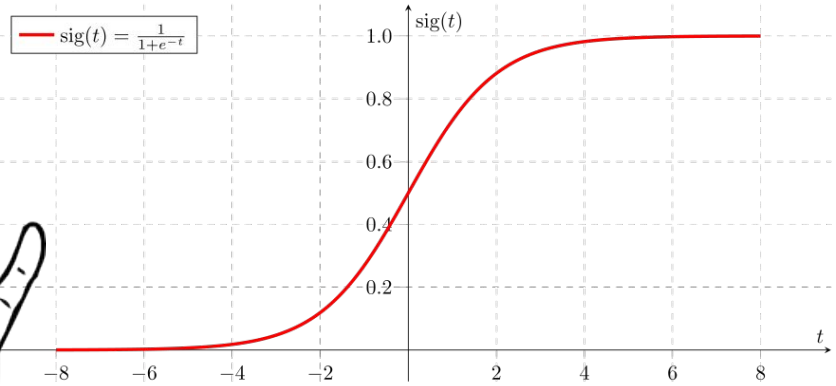
normalized initialization



Task 2: Sigmoid (length/functions/sigmoid.py)

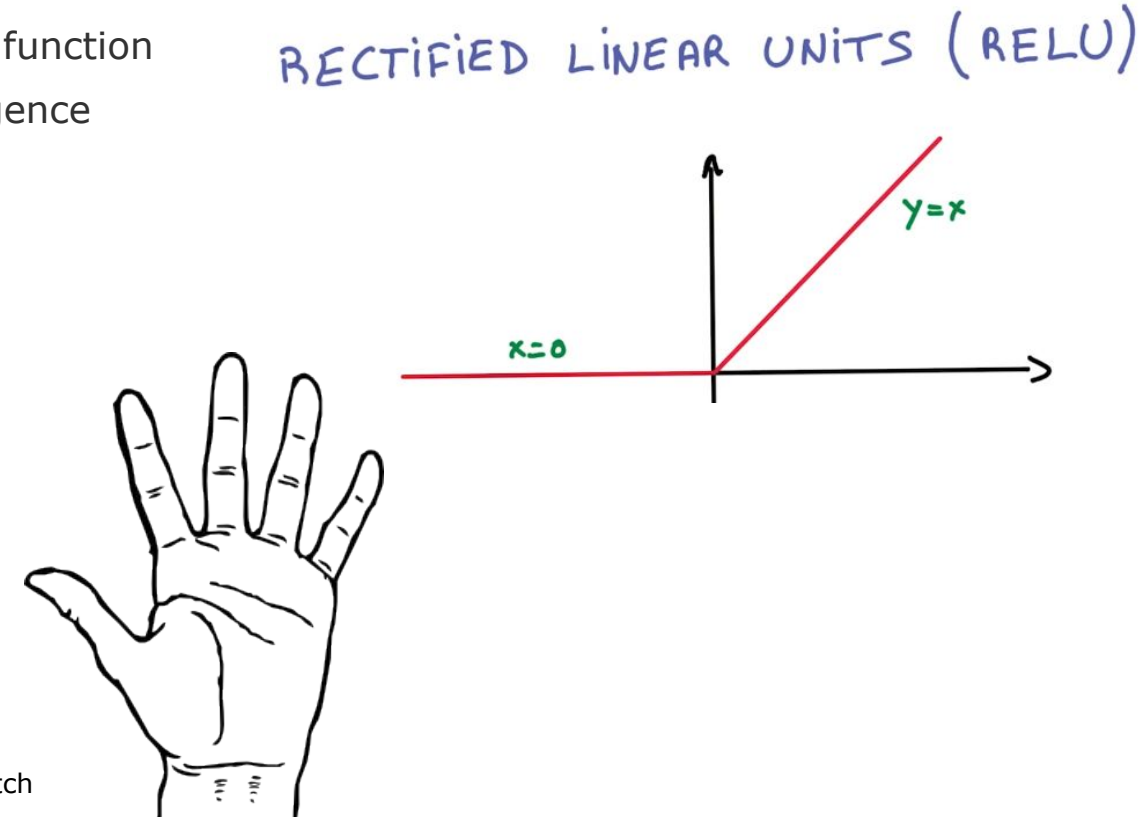
- forward pass:
  - trivial to implement for one value
  - batch processing: use numpy methods instead
- backward pass:
  - stepwise derivatives
  - direct derivative (lecture)
  - chain rule!
- why are we using non-linearities?





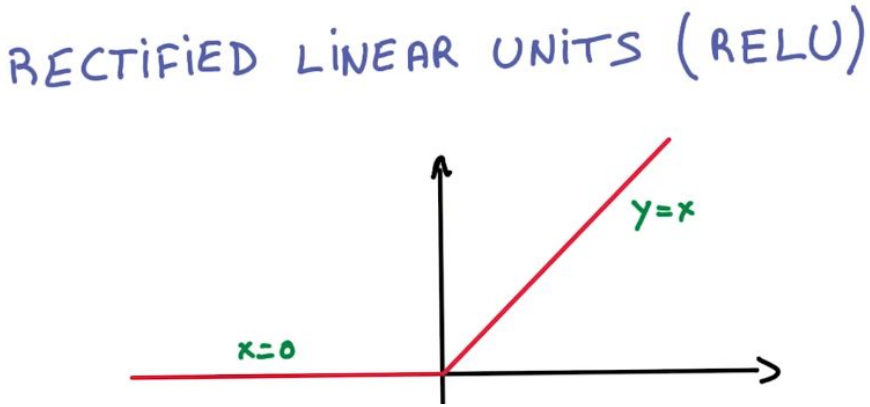
Task 3: ReLU (length/functions/relu.py)

- very simple activation function
- enables faster convergence
  - why?



Task 3: ReLU (length/functions/relu.py)

- very simple activation function
- enables faster convergence
  - does not saturate
  - stable gradient
- forward pass:
  - element-wise maximum
- backward pass:
  - only **sub**-differentiable!
  - think about every case

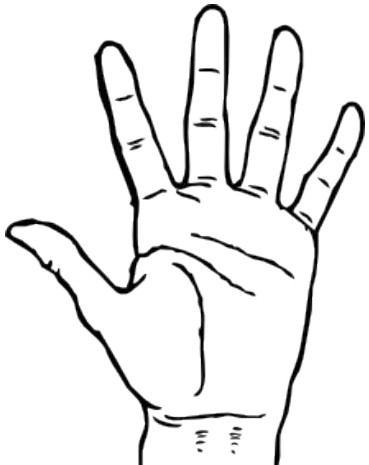


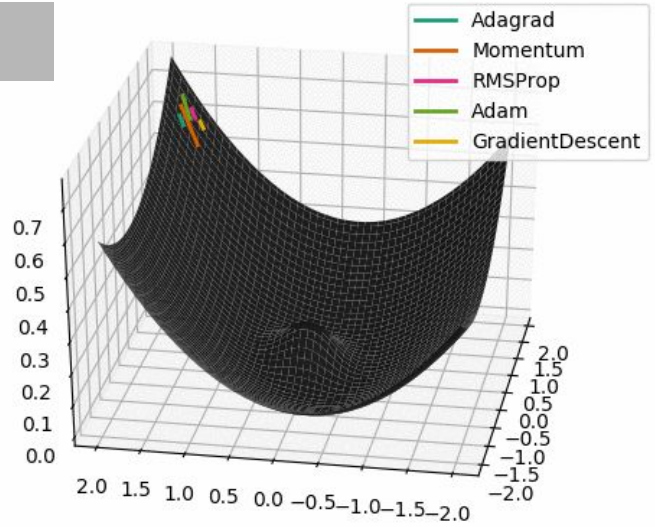
Task 4: Adam (length/optimizers/adam.py)

- baseline: SGD - “man walking the steepest way down”

```
param_deltas = [self.lr * grad for grad in gradients]
```

  - does anyone know how adam works?





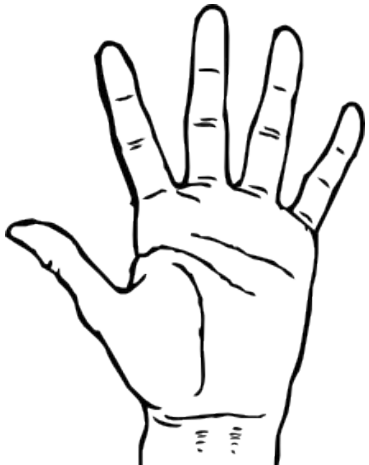
$$z = \frac{1}{20}\left(x^2 + x + y^2 + y + 3^{1-(5x^2+5y^2)}\right)$$

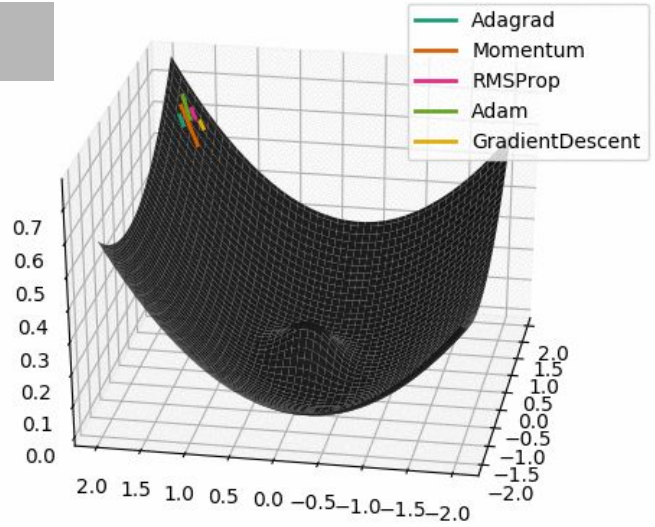
Task 4: Adam (length/optimizers/adam.py)

- baseline: SGD - “man walking the steepest way down”

```
param_deltas = [self.lr * grad for grad in gradients]
```

  - adam - “ball rolling down the hill”
  - how could we implement this?



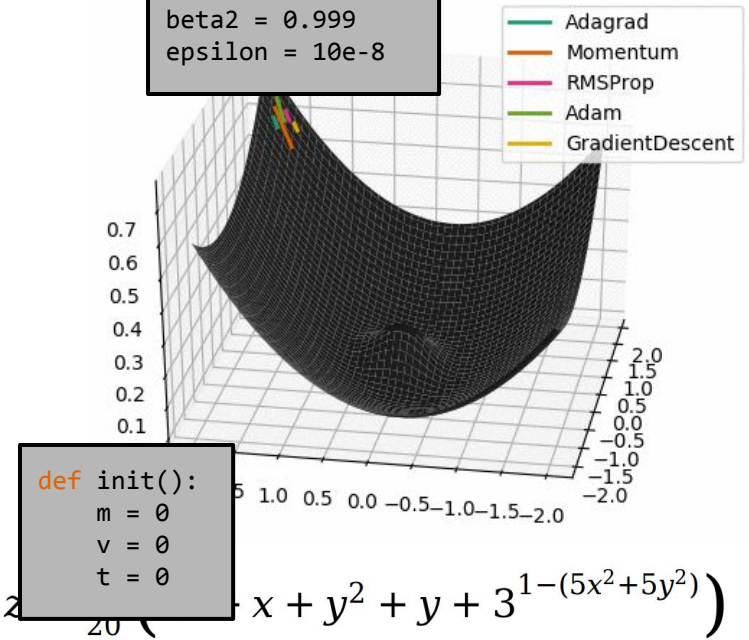


$$z = \frac{1}{20}\left(x^2 + x + y^2 + y + 3^{1-(5x^2+5y^2)}\right)$$

Task 4: Adam (length/optimizers/adam.py)

```
def adam(self, gradient):
    self.t += 1
    self.m = beta1 * self.m + (1 - beta1) * gradient
    self.v = beta2 * self.v + (1 - beta2) * (gradient ** 2)
    m_corrected = m / (1 - (beta1 ** self.t))
    v_corrected = v / (1 - (beta2 ** self.t))
    delta = alpha * m_corrected / ((v_corrected ** 0.5) + epsilon)
    return delta
```

```
alpha = 0.001
beta1 = 0.9
beta2 = 0.999
epsilon = 10e-8
```



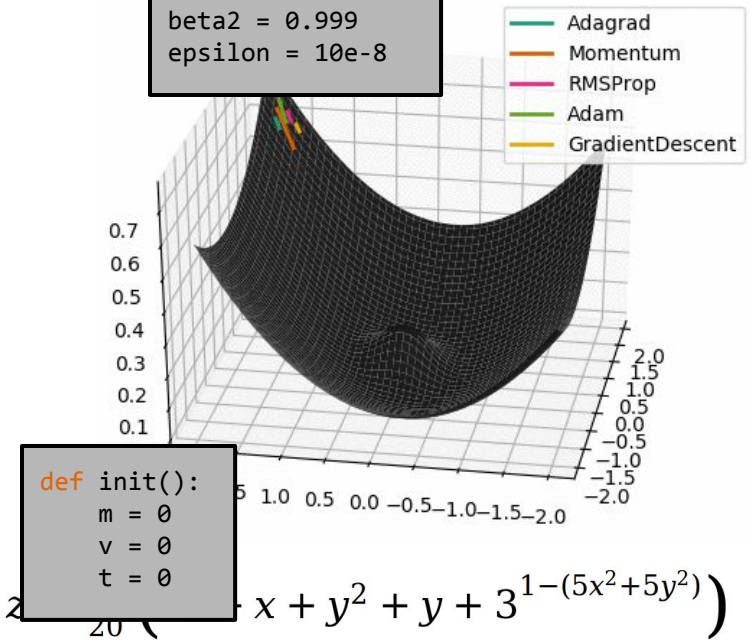

```
def init():
    m = 0
    v = 0
    t = 0
```

Task 4: Adam (length/optimizers/adam.py)

```
def adam(self, gradient):
    self.t += 1
    self.m = beta1 * self.m + (1 - beta1) * gradient
    self.v = beta2 * self.v + (1 - beta2) * (gradient ** 2)
    m_corrected = m / (1 - (beta1 ** self.t))
    v_corrected = v / (1 - (beta2 ** self.t))
    delta = alpha * m_corrected / ((v_corrected ** 0.5) + epsilon)
    return delta
```

increase timestep
(needed for bias correction)

```
alpha = 0.001
beta1 = 0.9
beta2 = 0.999
epsilon = 10e-8
```




```
def init():
    m = 0
    v = 0
    t = 0
```

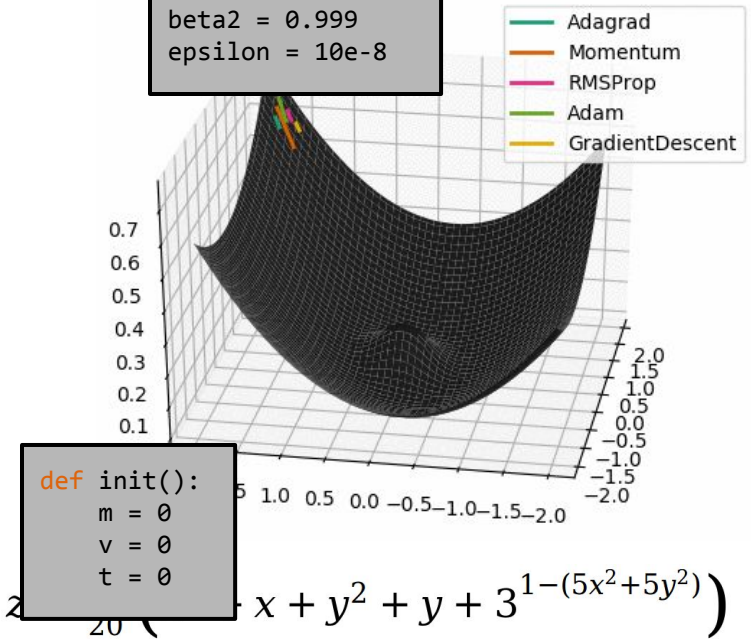
$$x + y^2 + y + 3^{1-(5x^2+5y^2)}$$

Task 4: Adam (length/optimizers/adam.py)

```
def adam(self, gradient):
    self.t += 1
    self.m = beta1 * self.m + (1 - beta1) * gradient
    self.v = beta2 * self.v + (1 - beta2) * (gradient ** 2)
    m_corrected = m / (1 - (beta1 ** self.t))
    v_corrected = v / (1 - (beta2 ** self.t))
    delta = alpha * m_corrected / (sqrt(v_corrected) + epsilon)
    return delta
```

```
adapt first order momentum (mean)
90 % - previous momentum
10 % - new gradients
```

```
alpha = 0.001
beta1 = 0.9
beta2 = 0.999
epsilon = 10e-8
```



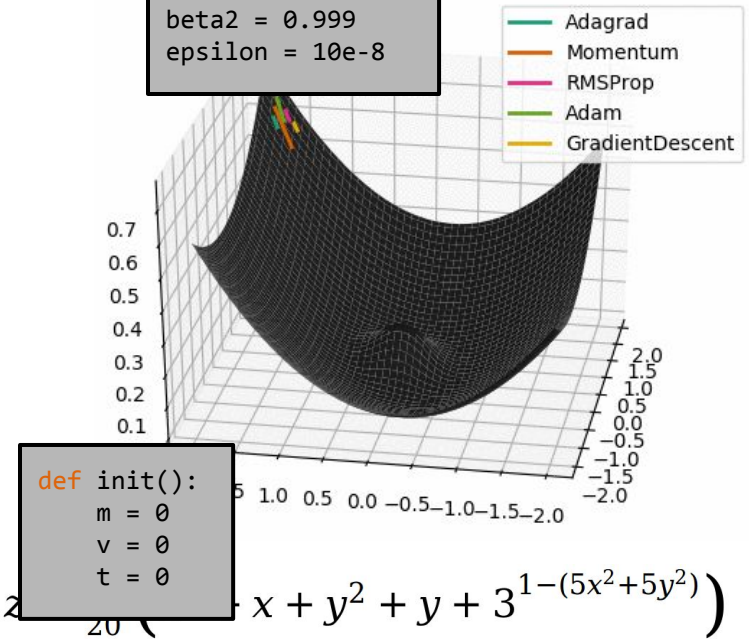

```
def init():
    m = 0
    v = 0
    t = 0
```

Task 4: Adam (length/optimizers/adam.py)

```
def adam(self, gradient):
    self.t += 1
    self.m = beta1 * self.m + (1 - beta1) * gradient
    self.v = beta2 * self.v + (1 - beta2) * (gradient ** 2)
    m_corrected = m / (1 - (beta1 ** self.t))
    v_corrected = v / (1 - (beta2 ** self.t))
    delta = alpha * m_corrected / (sqrt(v_corrected) + epsilon)
    return delta
```

```
adapt second order momentum (variance)
99.9 % - previous momentum
 0.1 % - element-wise square
         of new gradients
```

```
alpha = 0.001
beta1 = 0.9
beta2 = 0.999
epsilon = 10e-8
```




```
def init():
    m = 0
    v = 0
    t = 0
```

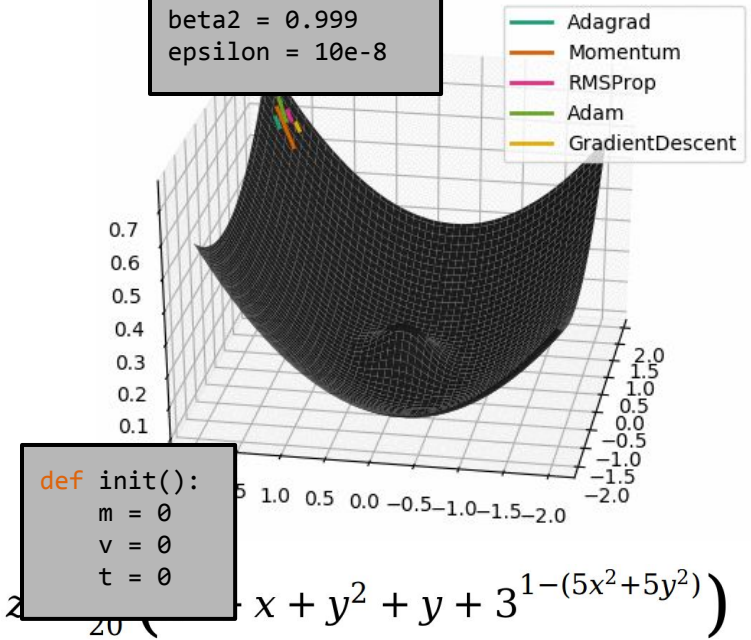
$$x + y^2 + y + 3^{1-(5x^2+5y^2)}\Big)$$

Task 4: Adam (length/optimizers/adam.py)

```python
def adam(self, gradient):
    self.t += 1
    self.m = beta1 * self.m + (1 - beta1) * gradient
    self.v = beta2 * self.v + (1 - beta2) * (gradient ** 2)
    m_corrected = m / (1 - (beta1 ** self.t))
    v_corrected = v / (1 - (beta2 ** self.t))
    delta = alpha * m_corrected / (sqrt(v_corrected) + epsilon)
    return delta
```

bias correction - most relevant for the first iterations (m and v were initalized with zero)

```
alpha = 0.001
beta1 = 0.9
beta2 = 0.999
epsilon = 10e-8
```




```python
def init():
    m = 0
    v = 0
    t = 0
```

$$x + y^2 + y + 3^{1-(5x^2+5y^2)}$$

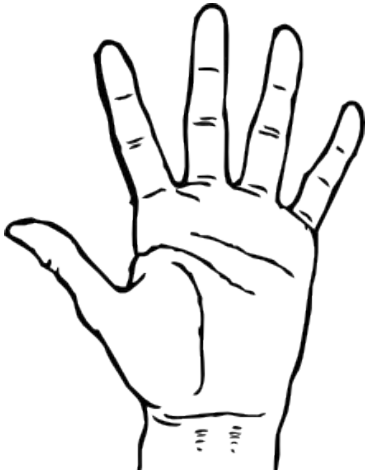
Task 4: Adam (length/optimizers/adam.py)

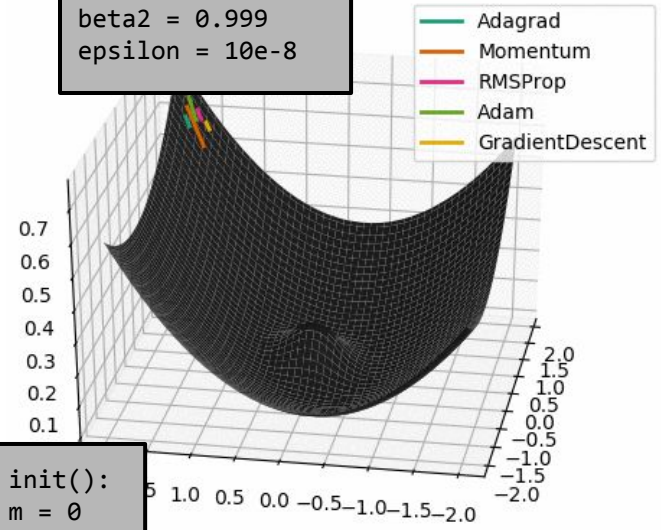
```
def adam(self, gradient):
    self.t += 1
    self.m = beta1 * self.m + (1 - beta1) * gradient
    self.v = beta2 * self.v + (1 - beta2) * (gradient ** 2)
    m_corrected = m / (1 - (beta1 ** self.t))
    v_corrected = v / (1 - (beta2 ** self.t))
    delta = alpha * m_corrected / (sqrt(v_corrected) + epsilon)
    return delta
```

```
alpha = 0.001
beta1 = 0.9
beta2 = 0.999
epsilon = 10e-8
```

calculate parameter delta
(alpha = learning rate)

what is the influence of v?






```
def init():
    m = 0
    v = 0
    t = 0
```

$$x + y^2 + y + 3^{1-(5x^2+5y^2)}$$

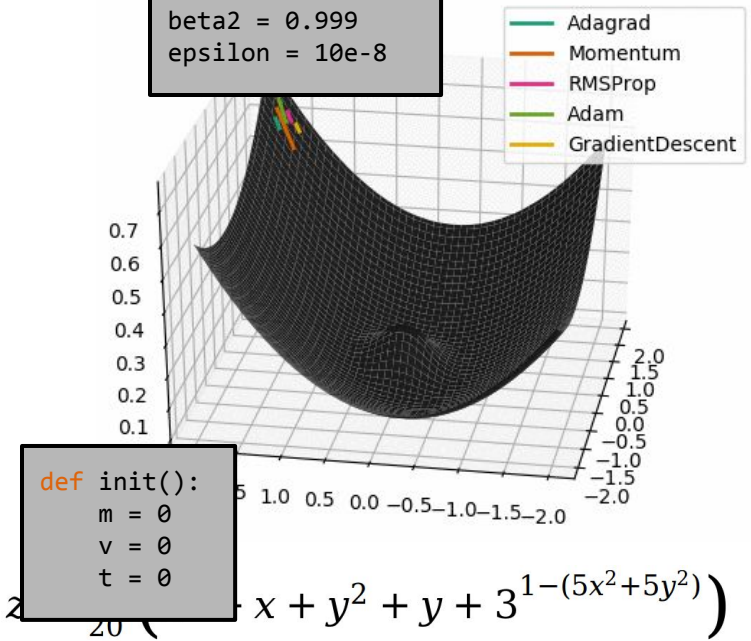
Task 4: Adam (length/optimizers/adam.py)

```
def adam(self, gradient):
    self.t += 1
    self.m = beta1 * self.m + (1 - beta1) * gradient
    self.v = beta2 * self.v + (1 - beta2) * (gradient ** 2)
    m_corrected = m / (1 - (beta1 ** self.t))
    v_corrected = v / (1 - (beta2 ** self.t))
    delta = alpha * m_corrected / (sqrt(v_corrected) + epsilon)
    return delta
```

```
calculate parameter delta
(alpha = learning rate)

v - decreases delta on
    alternating gradients

abs(delta) <= alpha
```

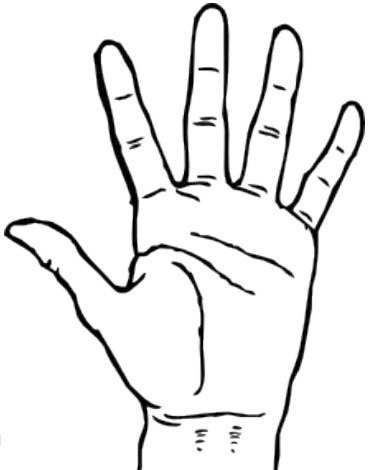
```
alpha = 0.001
beta1 = 0.9
beta2 = 0.999
epsilon = 10e-8
```

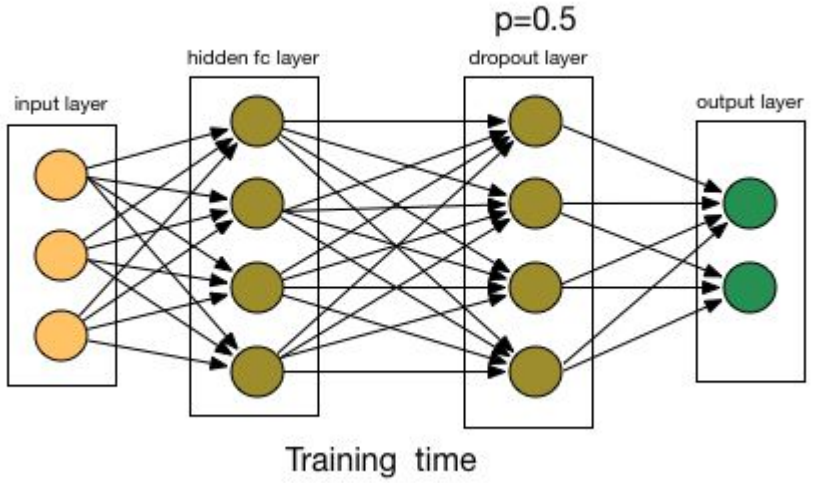
```
def init():
    m = 0
    v = 0
    t = 0
```



Task 5: Dropout (length/functions/dropout.py)

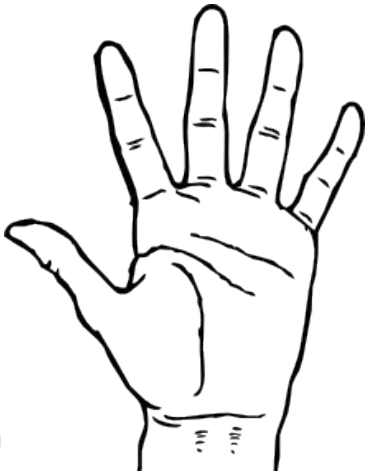
- regularization function that randomly drops units
- why does this help the training of the network?

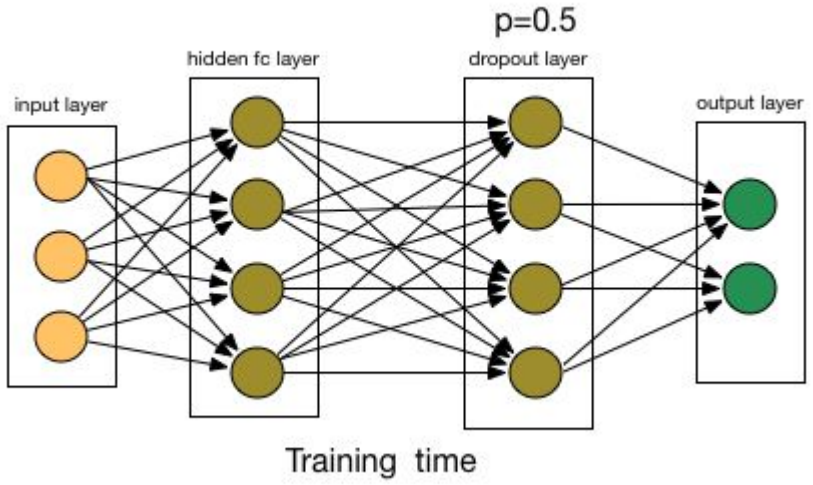




Task 5: Dropout (length/functions/dropout.py)

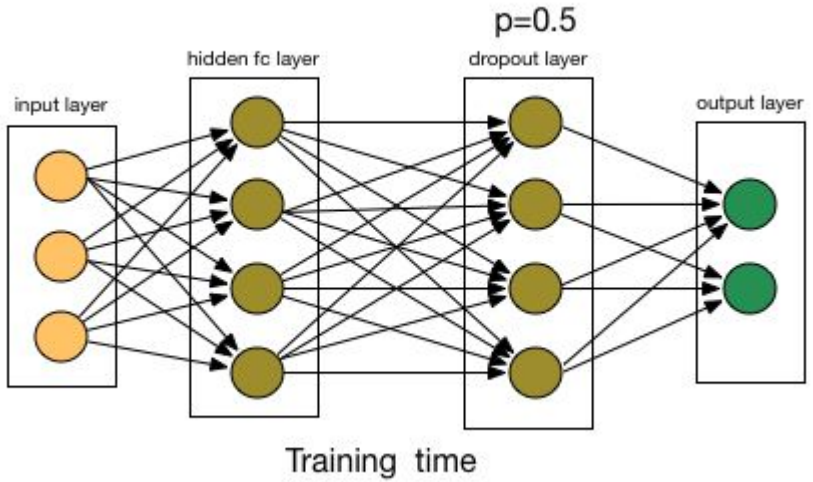
- regularization function that randomly drops units
- why does this help the training of the network?
  - forces network to find meaningful features
- anything we have to think of?





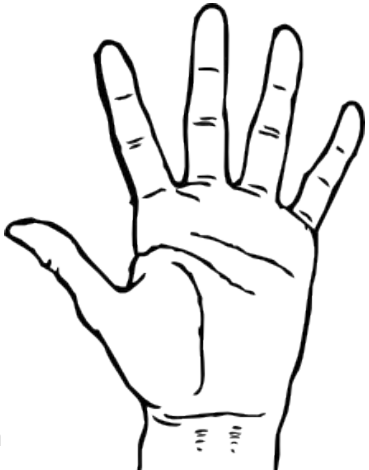
Task 5: Dropout (length/functions/dropout.py)

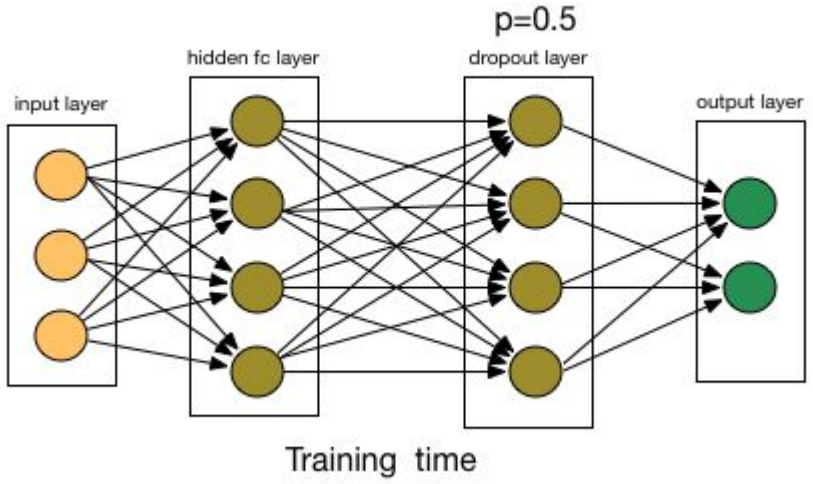
- regularization function that randomly drops units
- why does this help the training of the network?
  - forces network to find meaningful features
- anything we have to think of?
  - **no** dropout at testing time!
  - scaling necessary!



Task 5: Dropout (length/functions/dropout.py)

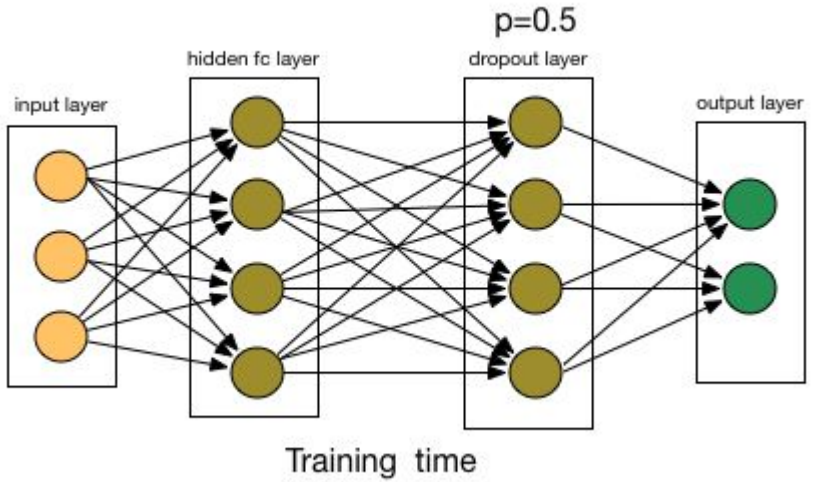
- forward pass:
  - drop a value in input with probability p
  - scale outputs of functions by probability p
- backward pass:
  - anyone an idea?



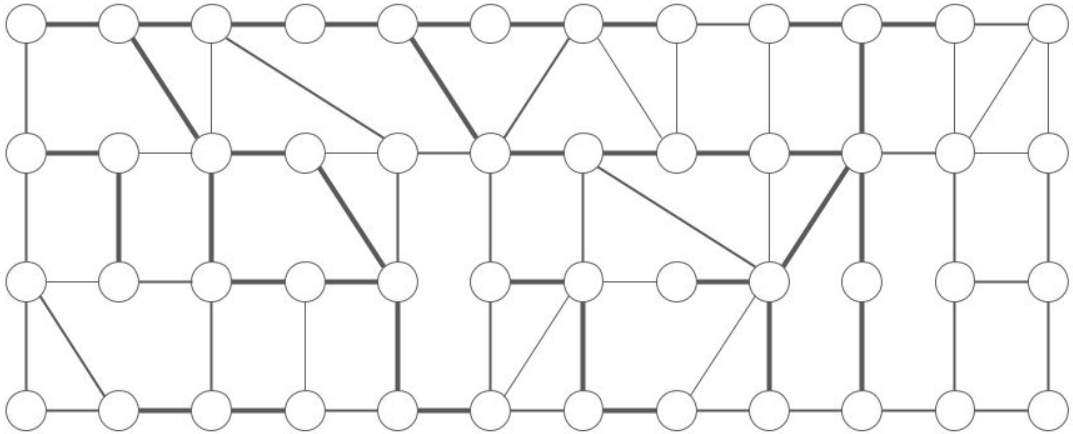


Task 5: Dropout (length/functions/dropout.py)

- forward pass:
  - drop a value in input with probability `p`
  - scale outputs of functions by probability `p`
- backward pass:
  - set gradients of dropped units to `0`
- testing time:
  - do nothing



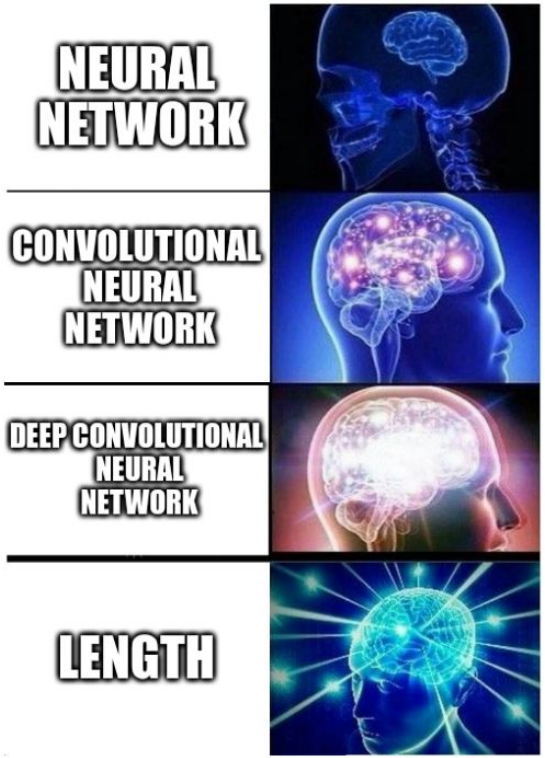
Hasso Plattner Institut
Digital Engineering · Universität Potsdam



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

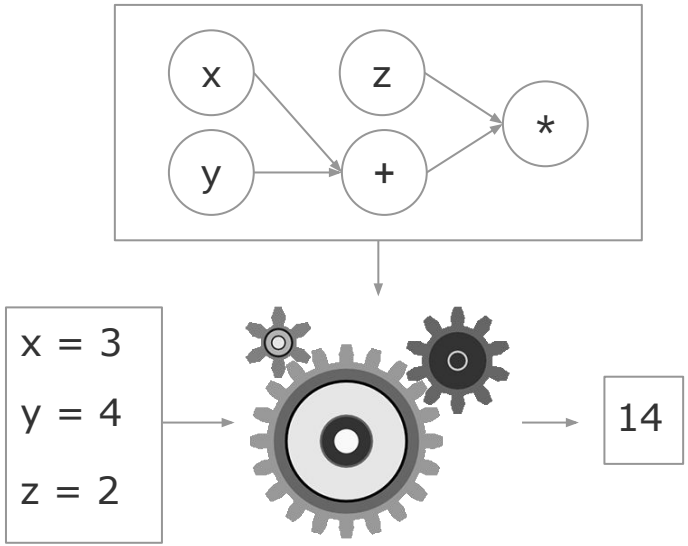
LENGTH - Recap

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



LENGTH - Computational Graph

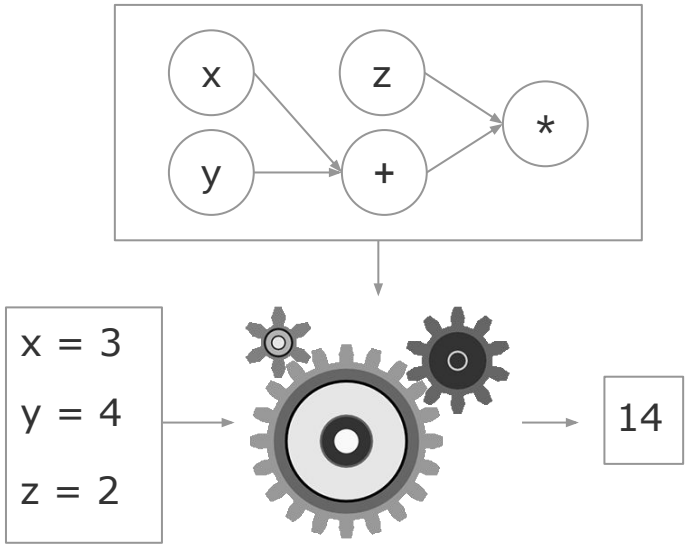
Static Computational Graph
(define and run)



Dynamic Computational Graph
(define by run)

LENGTH - Computational Graph

Static Computational Graph
(define and run)



Dynamic Computational Graph
(define by run)

LENGTH - Backward Computation

```
import numpy as np
import length.functions as F
from length.graph import Graph
```

```
x = Graph(np.array([3], dtype=np.float32))
y = Graph(np.array([4], dtype=np.float32))
z = Graph(np.array([2], dtype=np.float32))
```

create input data and prepare computational graph

```
h = F.add(x, y)
out = F.multiply(h, z)
```

LENGTH - Backward Computation

```
import numpy as np
import length.functions as F
from length.graph import Graph


x = Graph(np.array([3], dtype=np.float32))
y = Graph(np.array([4], dtype=np.float32))
z = Graph(np.array([2], dtype=np.float32))
```

```
h = F.add(x, y)
out = F.multiply(h, z)
```

```
perform computation and keep track of
computational graph

>>> out.visualize()
id | layer          | next
 1 | input (1,)     |    4
 2 | input (1,)     |    4
 3 | input (1,)     |    5
 4 | Add (1,)       |    5
 5 | Multiply (1,)  |    6
```

LENGTH - Backward Computation (length/graph.py)

- `out.backward(optimizer)` → starts computation of gradients and update of learnable parameters

```
def backward(self, optimizer):
    if self.data.size == 1 and self.grad is None:
        self.grad = np.ones((1,), dtype=constants.DTYPE)

    candidate_layers = []
    seen_layers = set()

    def add_candidate_layer(candidate):
        if candidate is not None and candidate not in seen_layers:
            candidate_layers.append(candidate)
            seen_layers.add(candidate)

    add_candidate_layer(self)
```

```
df
-- = 1
df
```

LENGTH - Backward Computation (length/graph.py)

- `out.backward(optimizer)` → starts computation of gradients and update of learnable parameters

```
def backward(self, optimizer):
        if self.data.size == 1 and self.grad is None:
            self.grad = np.ones((1,), dtype=constants.DTYPE)

        candidate_layers = []
        seen_layers = set()

        def add_candidate_layer(candidate):
            if candidate is not None and candidate not in seen_layers:
                candidate_layers.append(candidate)
                seen_layers.add(candidate)

        add_candidate_layer(self)
```

prepare gradient computation for each function in computational graph

LENGTH - Backward Computation (length/graph.py)

```
def backward(self, optimizer):
    [...]
    while candidate_layers:
        candidate_layer = candidate_layers.pop()
        if candidate_layer.creator is None:
            continue

        if candidate_layer.creator.needs_optimizer:
            candidate_layer.creator.optimizer = optimizer

        gradients = candidate_layer.creator.backward(candidate_layer.grad)

        for predecessor, gradient in zip(candidate_layer.predecessors, gradients):
            predecessor.grad = gradient
            if gradient is not None:
                # the gradient flows to another layer (does not happen with loss layers)
                add_candidate_layer(predecessor)
```

as long as we are not at the top of the computational graph, we go on

LENGTH - Backward Computation (length/graph.py)

```
def backward(self, optimizer):
        [...]
        while candidate_layers:
            candidate_layer = candidate_layers.pop()
            if candidate_layer.creator is None:
                continue

            if candidate_layer.creator.needs_optimizer:
                candidate_layer.creator.optimizer = optimizer

            gradients = candidate_layer.creator.backward(candidate_layer.grad)

            for predecessor, gradient in zip(candidate_layer.predecessors, gradients):
                predecessor.grad = gradient
                if gradient is not None:
                    # the gradient flows to another layer (does not happen with loss layers)
                    add_candidate_layer(predecessor)
```

set optimizer if necessary

LENGTH - Backward Computation (length/graph.py)

```
def backward(self, optimizer):
        [...]
        while candidate_layers:
            candidate_layer = candidate_layers.pop()
            if candidate_layer.creator is None:
                continue

            if candidate_layer.creator.needs_optimizer:
                candidate_layer.creator.optimizer = optimizer

            gradients = candidate_layer.creator.backward(candidate_layer.grad)

            for predecessor, gradient in zip(candidate_layer.predecessors, gradients):
                predecessor.grad = gradient
                if gradient is not None:
                    # the gradient flows to another layer (does not happen with loss layers)
                    add_candidate_layer(predecessor)
```

compute gradients of this layer/function

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

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

    def internal_update(self, parameter_deltas):
        raise NotImplementedError

    def backward(self, gradients):
        gradients = super().backward(gradients)
        input_gradient = gradients[:len(self.inputs)]
        parameter_gradients = gradients[len(self.inputs):]
        if len(parameter_gradients) > 0:
            parameter_deltas = self.optimizer.run_update_rule(parameter_gradients, self)
            self.internal_update(parameter_deltas)
        return input_gradient
```

compute gradients with respect to inputs and parameters of the layer

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

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

    def internal_update(self, parameter_deltas):
        raise NotImplementedError

    def backward(self, gradients):
        gradients = super().backward(gradients)
        input_gradient = gradients[:len(self.inputs)]
        parameter_gradients = gradients[len(self.inputs):]
        if len(parameter_gradients) > 0:
            parameter_deltas = self.optimizer.run_update_rule(parameter_gradients, self)
            self.internal_update(parameter_deltas)
        return input_gradient
```

use optimizer to compute updates for internal parameters, based on computed gradients

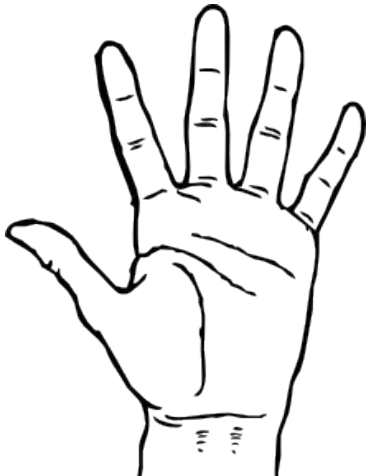
LENGTH - Backward Computation (length/graph.py)

```
def backward(self, optimizer):
        [...]
        while candidate_layers:
            candidate_layer = candidate_layers.pop()
            if candidate_layer.creator is None:
                continue

            if candidate_layer.creator.needs_optimizer:
                candidate_layer.creator.optimizer = optimizer

            gradients = candidate_layer.creator.backward(candidate_layer.grad)

            for predecessor, gradient in zip(candidate_layer.predecessors, gradients):
                predecessor.grad = gradient
                if gradient is not None:
                    # the gradient flows to another layer (does not happen with loss layers)
                    add_candidate_layer(predecessor)
```

find next functions to compute gradients for and scatter gradients to them

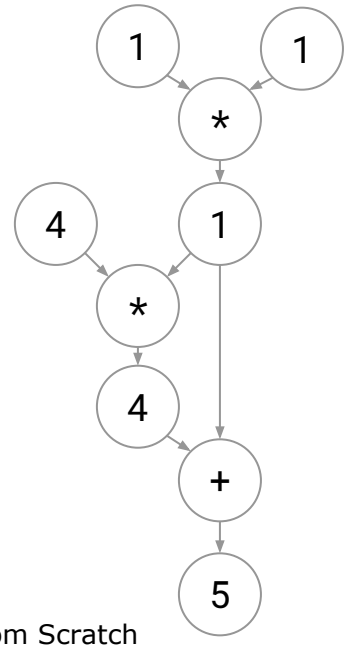
LENGTH - Backward Computation

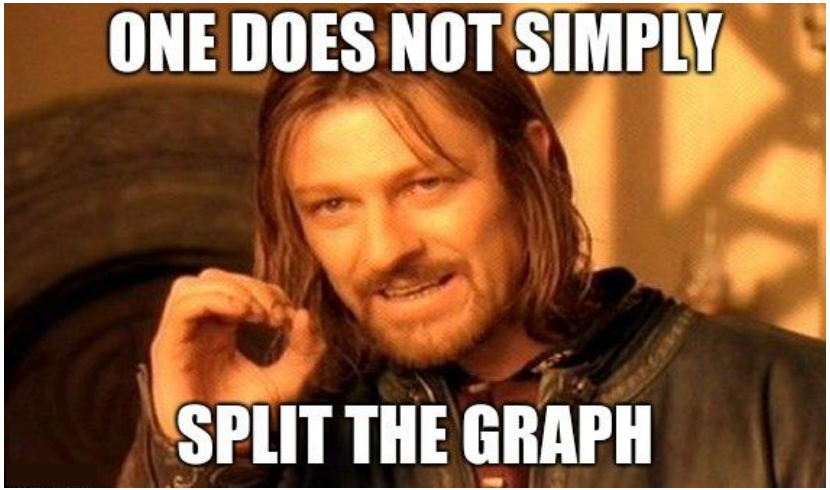
- do you see any problems with this backward implementation?



LENGTH - Backward Computation

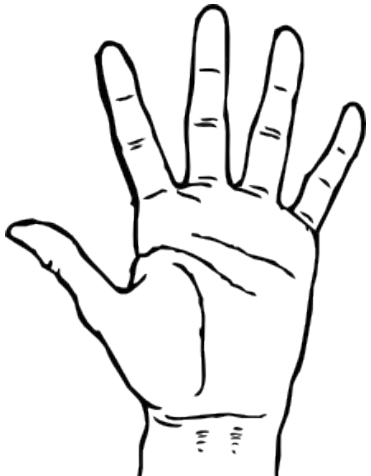
- do you see any problems with this backward implementation?
  - can not handle networks with graphs that split at a certain point





How Can We Improve Our Results?

```
def __init__(self):
    self.fully_connected_1 = FullyConnected(784, 512)
    self.fully_connected_2 = FullyConnected(512, 512)
    self.fully_connected_3 = FullyConnected(512, 10)
    [...]

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)
```



How Can We Improve Our Results?

```
def __init__(self):
    self.fully_connected_1 = FullyConnected(784, 512)
    self.fully_connected_2 = FullyConnected(512, 512)
    self.fully_connected_3 = FullyConnected(512, 10)
    [...]

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)
```

increase layer size

use adam

```
python train.py --optimizer adam
```

add relu/sigmoid

add dropout

replace `mean_squared_error` with `softmax_cross_entropy`

Task Overview - Time to Hack!

1. Initialization
   - `length/initializers/xavier.py`
2. Sigmoid
   - `length/functions/sigmoid.py`
3. ReLU
   - `length/functions/relu.py`
4. Adam
   - `length/optimizers/adam.py`
5. Dropout
   - `length/functions/dropout.py`

```
$ python train.py --optimizer adam
train: epoch:  0, loss:  0.12, accuracy 0.94, iteration:  900
running test set...
 test: epoch:  0, loss:  0.18, accuracy 0.96
train: epoch:  1, loss:  0.14, accuracy 0.94, iteration:  900
running test set...
 test: epoch:  1, loss:  0.11, accuracy 0.98
train: epoch:  2, loss:  0.09, accuracy 0.97, iteration:  900
running test set...
 test: epoch:  2, loss:  0.08, accuracy 0.99
train: epoch:  3, loss:  0.02, accuracy 0.98, iteration:  900
running test set...
 test: epoch:  3, loss:  0.11, accuracy 0.98
train: epoch:  4, loss:  0.03, accuracy 1.00, iteration:  900
running test set...
 test: epoch:  4, loss:  0.10, accuracy 0.98
train: epoch:  5, loss:  0.10, accuracy 0.97, iteration:  900
running test set...
 test: epoch:  5, loss:  0.08, accuracy 0.99
train: epoch:  6, loss:  0.04, accuracy 0.98, iteration:  900
running test set...
 test: epoch:  6, loss:  0.08, accuracy 0.99
train: epoch:  7, loss:  0.05, accuracy 0.98, iteration:  900
running test set...
 test: epoch:  7, loss:  0.08, accuracy 0.99
train: epoch:  8, loss:  0.00, accuracy 1.00, iteration:  900
running test set...
 test: epoch:  8, loss:  0.07, accuracy 1.00
train: epoch:  9, loss:  0.00, accuracy 1.00, iteration:  900
running test set...
 test: epoch:  9, loss:  0.08, accuracy 0.99
```

Run test with: `pytest`

Run actual training: `python train.py --optimizer [sgd,adam]`

Next Time

We use a real framework for inference with a trained model.

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