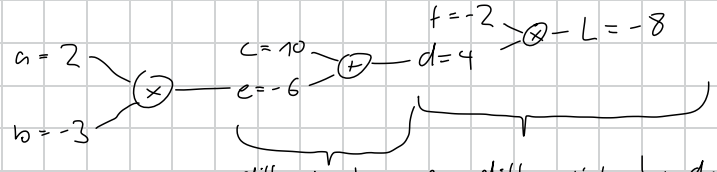
Backpropagation

↳ figure out how much changing certain inputs changes outputs

Example:



differentiate by e. eg.: differentiate by d:

$f(e) = f \cdot (c+e)$ $\quad f(d) = d \cdot f \quad \Rightarrow f'(d) = f = -2$

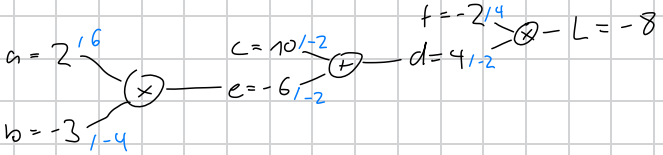
$\Rightarrow g(e) = c+e \quad h(e) = f \cdot e$

$\Rightarrow f(e) = h(g(e))$

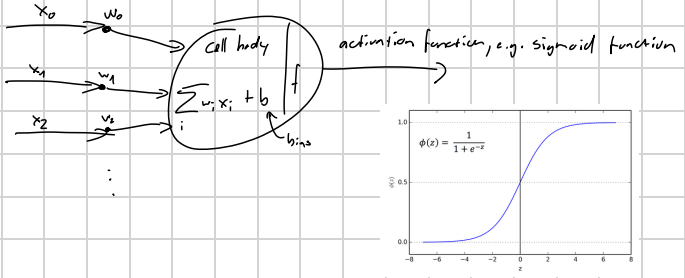
$\Rightarrow f'(e) = \underbrace{h'(g(e))}_{f} \cdot \underbrace{g'(e)}_{1} = -2$ by chain rule

Like this we can backpropagate through the entire network and figure out the gradients by multiplying the local gradients by global previous (from backward direction) gradients.

For example above:



Single neuron

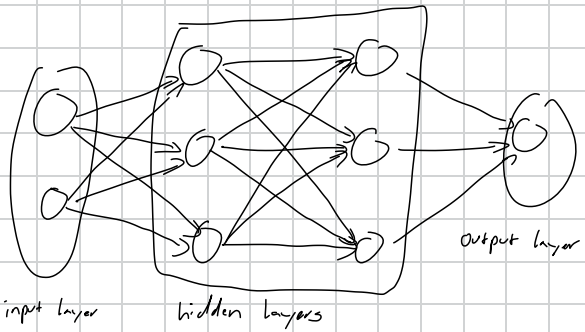


forward pass: calculate output depending on inputs → prediction

backward pass: backpropagate and find gradients → learning

MLP: Multi-layer-perceptron

each layer fully connected to previous layer




```python
class Neuron:
  def __init__(self, nIn):
    self.w = [Value(random.uniform(-1, 1)) for _ in range(nIn)]
    self.b = Value(random.uniform(-1, 1))
  def __call__(self, x):
    # calculate w * x + b
    act = sum((xi * wi for xi, wi in zip(self.w, x)), self.b)
    out = act.tanh()
    return out

class Layer:
  def __init__(self, nIn, nOut):
    self.neurons = [Neuron(nIn) for _ in range(nOut)]

  def __call__(self, x):
    outs = [n(x) for n in self.neurons]
    return outs

class MLP:
  def __init__(self, nin, nout):
    sz = [nin] + nout #list with n inputs, nout is list with outputs per level
    self.layers = [Layer(sz[i], sz[i + 1]) for i in range(len(nout))]

  def __call__(self, x):
    for layer in self.layers:
      x = layer(x)
    return x

x = [2.0, 3.0, -1.0]
n = MLP(3, [4, 4, 1])
n(x)
```

3 inputs, hidden layers = 4,4, outputs = 1

Gradient descent

through backpropagation we figured out the gradients of each parameter (= weights + biases of neurons in MLP). In gradient descent we then set data to data += -step_size · grad, we then iteratively recalculate gradients and adjust the parameters slightly. Through this we try to find the minimum of parameters for loss.

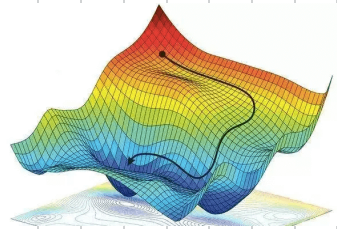
```python
#gradient descent
# reset all gradients to zero
for k in range(20):

  # forward pass
  ypred = [n(x) for x in xs]
  loss = sum((yi - predi)**2 for yi, predi in zip(ys, ypred))

  # backward pass
  for p in n.parameters():
    p.grad = 0.0 #need to reset since otherwise backwards accumulates => huge step size
  loss.backward()

  # update
  for p in n.parameters():
    p.data += -0.05 * p.grad #step size 0.05
```

Visually:



Batch: On tiny networks we can go through all data in the forward pass, however for actual useful networks, the training data is too large ⇒ we take random subset of the data and use that for forward pass.

Learning rate decay: As the network stabilizes, we decrease learning rate per step to avoid overshooting.