Introduction to Neural Networks

Philipp Koehn

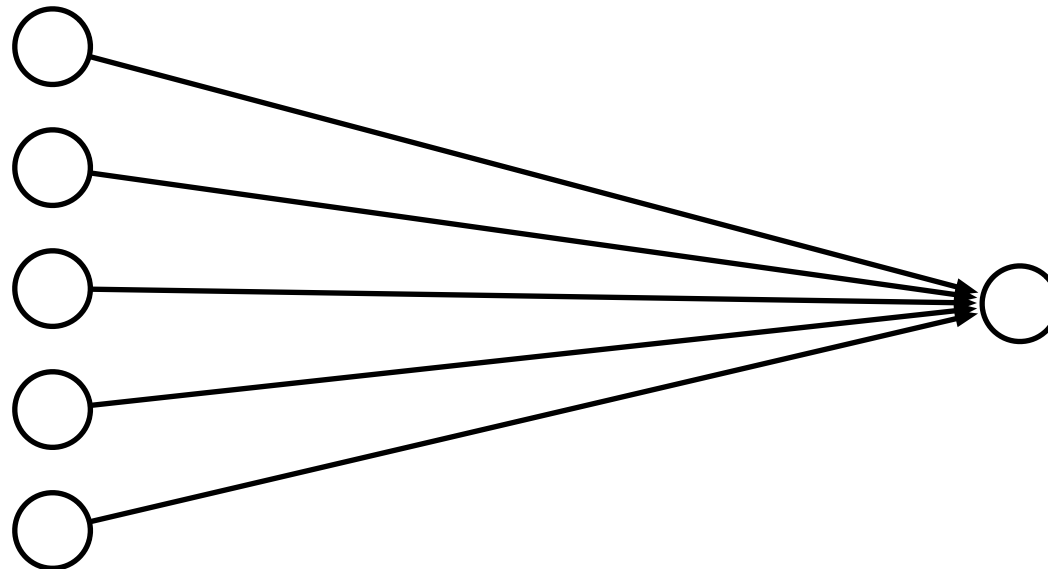
21 September 2023

Linear Models

- We used before weighted linear combination of feature values h_j and weights λ_j

$$\text{score}(\lambda, \mathbf{d}_i) = \sum_j \lambda_j \, h_j(\mathbf{d}_i)$$

- Such models can be illustrated as a "network"



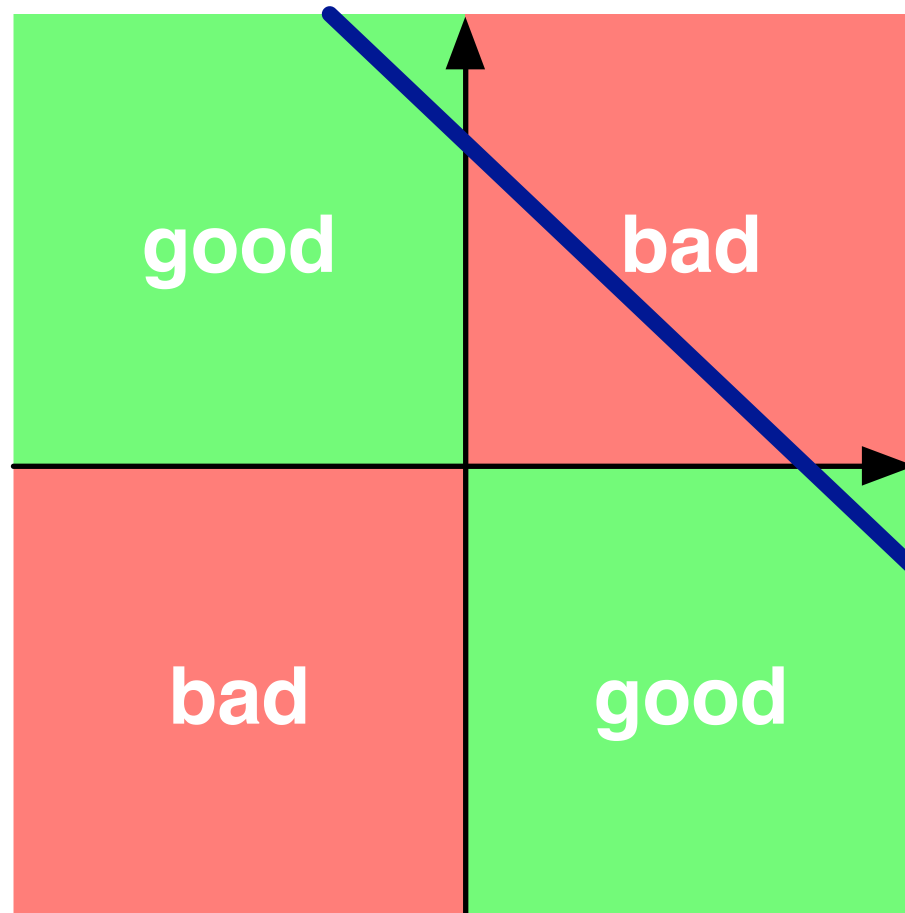
Limits of Linearity

- We can give each feature a weight

- But not more complex value relationships, e.g,
  - any value in the range [0;5] is equally good
  - values over 8 are bad
  - higher than 10 is not worse

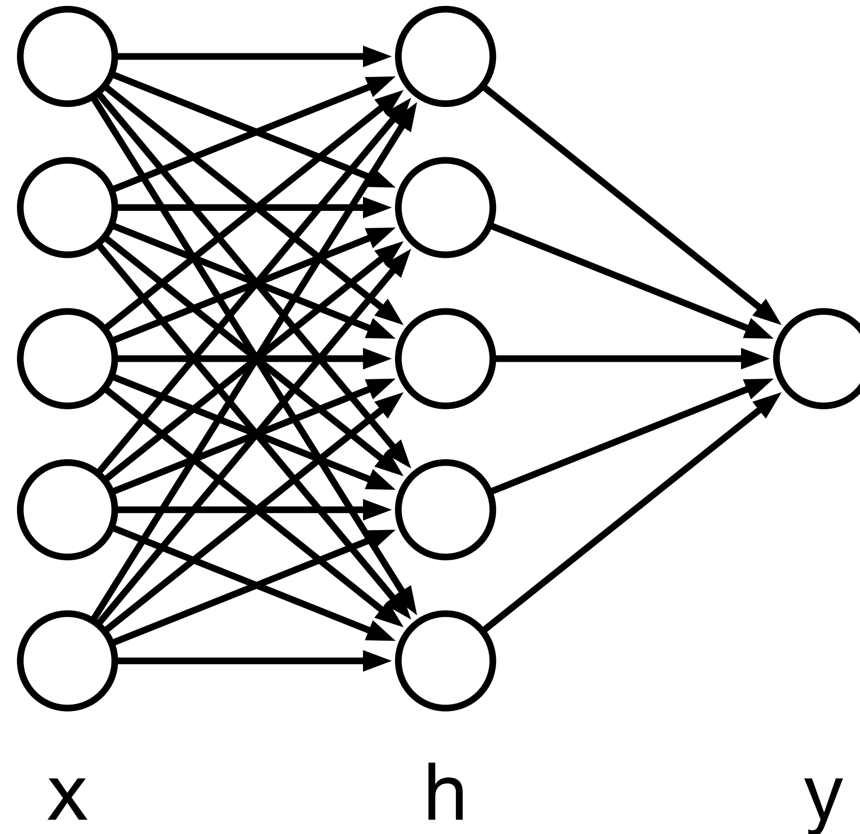
XOR

- Linear models cannot model XOR



Multiple Layers

- Add an intermediate ("hidden") layer of processing
  (each arrow is a weight)



- Have we gained anything so far?

Non-Linearity

- Instead of computing a linear combination

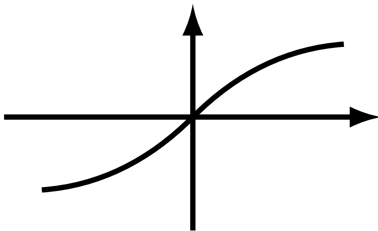
$$\text{score}(\lambda, \mathbf{d}_i) = \sum_j \lambda_j \; h_j(\mathbf{d}_i)$$

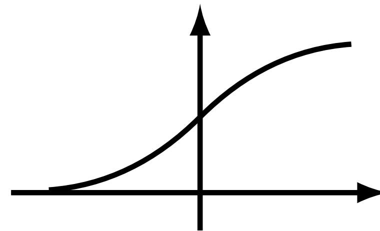
- Add a non-linear function

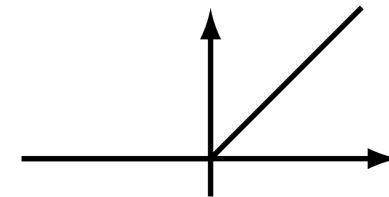
$$\text{score}(\lambda, \mathbf{d}_i) = f\Big(\sum_j \lambda_j \; h_j(\mathbf{d}_i)\Big)$$

- Popular choices

tanh(x) sigmoid(x) = $\frac{1}{1+e^{-x}}$ relu(x) = max(0,x)



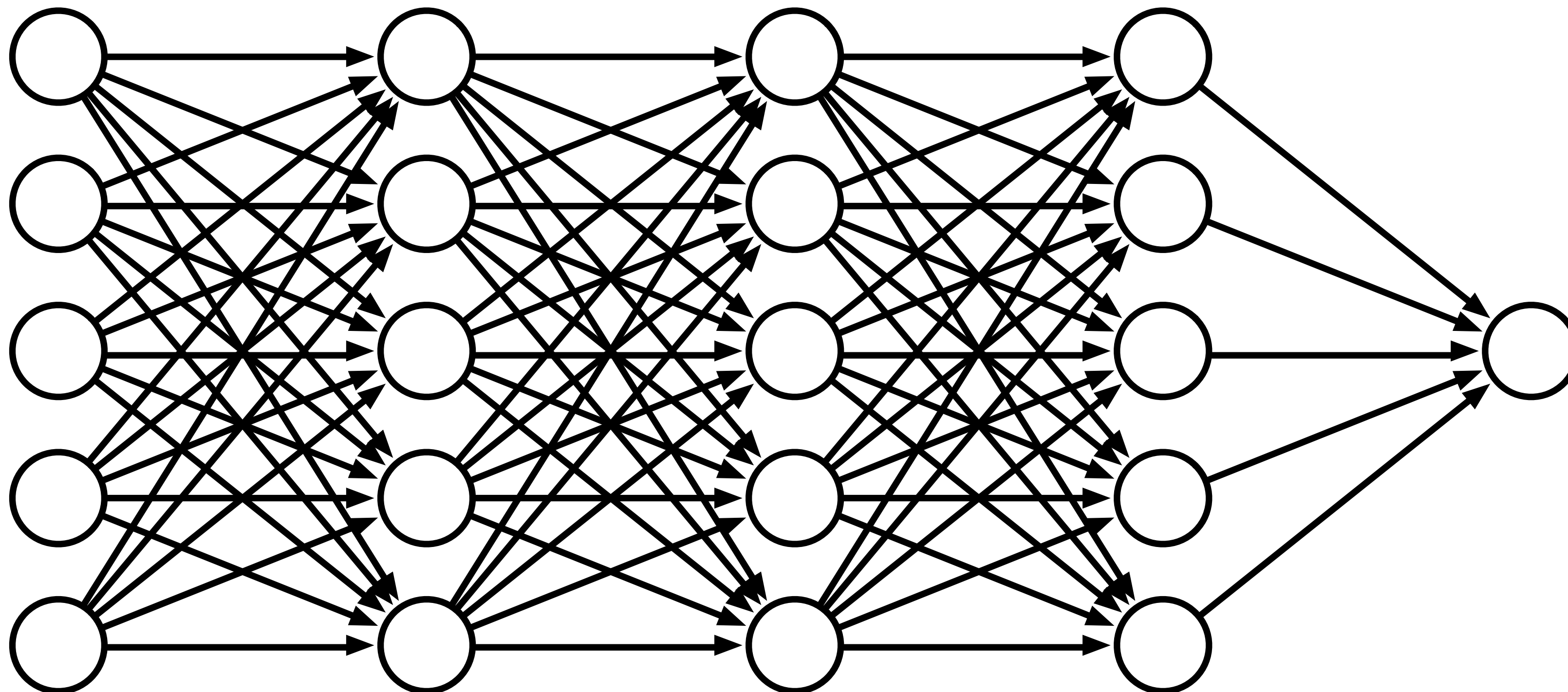




(sigmoid is also called the "logistic function")

Deep Learning

- More layers = deep learning

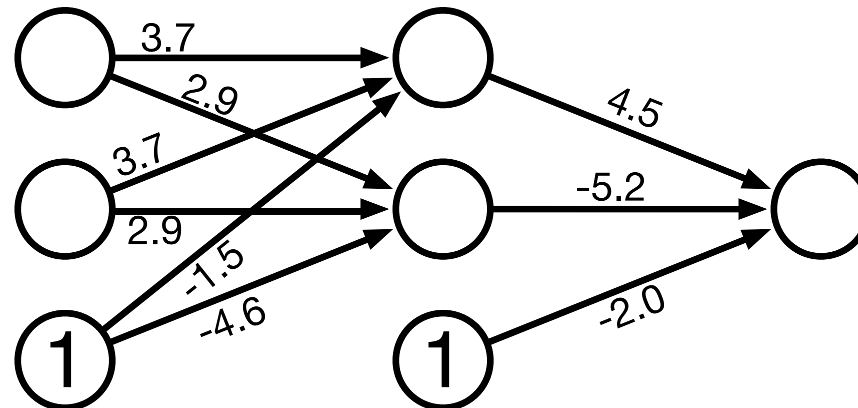


What Depth Enables

- Each layer is a processing step
- Having multiple processing steps allows complex functions
- Metaphor: NN and computing circuits
  - computer = sequence of Boolean gates
  - neural computer = sequence of layers
- Deep neural networks can implement complex functions
  e.g., sorting on input values

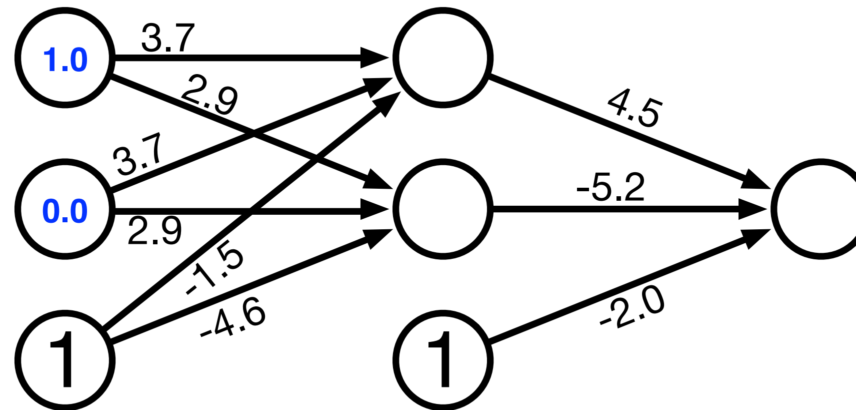
example

Simple Neural Network



- One innovation: bias units (no inputs, always value 1)

Sample Input

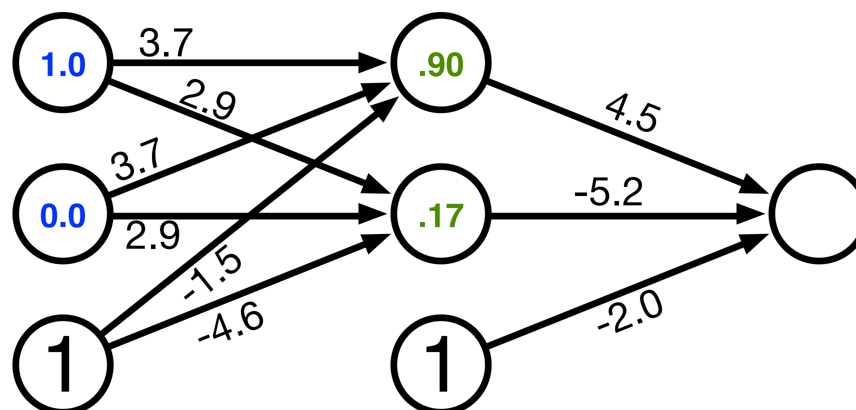


- Try out two input values
- Hidden unit computation

$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1 + e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1 + e^{1.6}} = 0.17$$

Computed Hidden

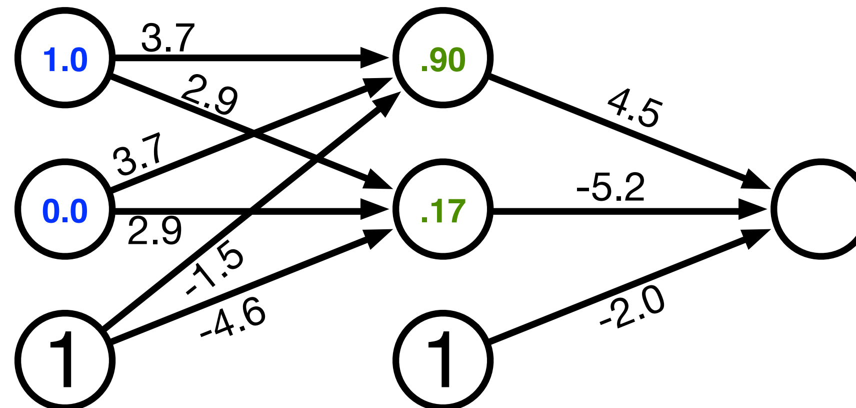


- Try out two input values
- Hidden unit computation

$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1 + e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1 + e^{1.6}} = 0.17$$

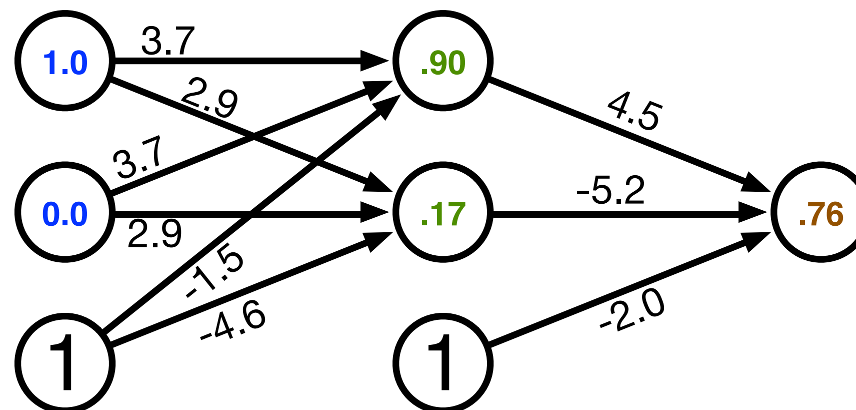
Compute Output



- Output unit computation

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76$$

Computed Output



- Output unit computation

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76$$

Output for all Binary Inputs

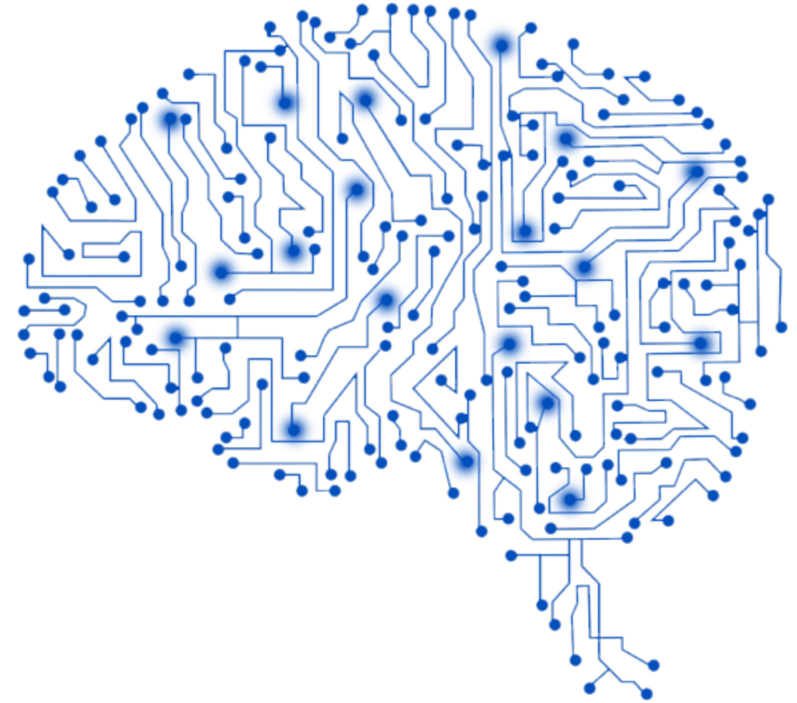
Input x_0	Input x_1	Hidden h_0	Hidden h_1	Output y_0
0	0	0.12	0.02	$0.18 \rightarrow 0$
0	1	0.88	0.27	$0.74 \rightarrow 1$
1	0	0.73	0.12	$0.74 \rightarrow 1$
1	1	0.99	0.73	$0.33 \rightarrow 0$

- Network implements XOR
  - hidden node h_0 is OR
  - hidden node h_1 is AND
  - final layer operation is $h_0 - -h_1$

- Power of deep neural networks: chaining of processing steps
  just as: more Boolean circuits $\rightarrow$ more complex computations possible

why ”neural” networks?

The Brain vs. Artificial Neural Networks

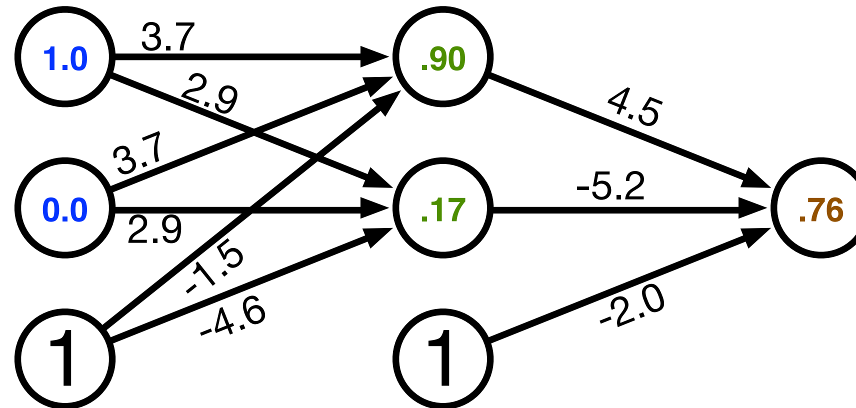


- Similarities
  - Neurons, connections between neurons
  - Learning = change of connections, not change of neurons
  - Massive parallel processing

- But artificial neural networks are much simpler
  - computation within neuron vastly simplified
  - discrete time steps
  - typically some form of supervised learning with massive number of stimuli

back-propagation training

Error



- Computed output: $y = .76$

- Correct output: $t = 1.0$

⇒ How do we adjust the weights?

Key Concepts

- Gradient descent
  - error is a function of the weights
  - we want to reduce the error
  - gradient descent: move towards the error minimum
  - compute gradient $\rightarrow$ get direction to the error minimum
  - adjust weights towards direction of lower error

- Back-propagation
  - first adjust last set of weights
  - propagate error back to each previous layer
  - adjust their weights

Hidden Layer Update

- In a hidden layer, we do not have a target output value
- But we can compute how much each node contributed to downstream error
- Definition of error term of each node

$$\delta_j = (t_j - y_j)\ y'_j$$

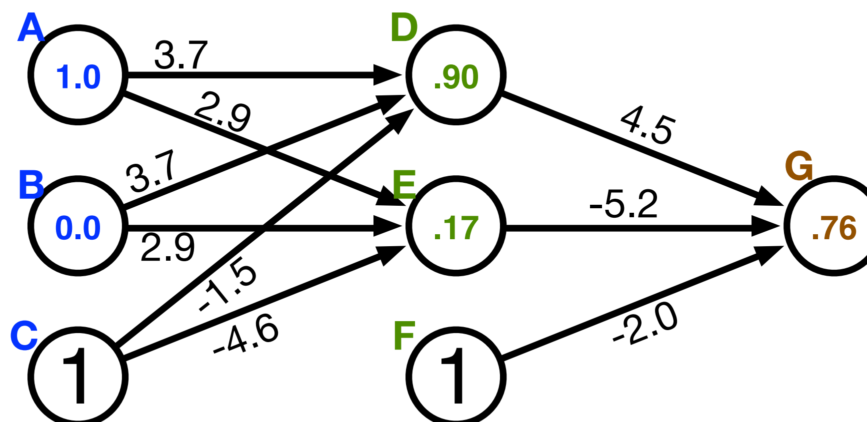
- Back-propagate the error term
  (why this way? there is math to back it up...)

$$\delta_i = \Big(\sum_j w_{j \leftarrow i} \delta_j \Big)\ y'_i$$

- Universal update formula

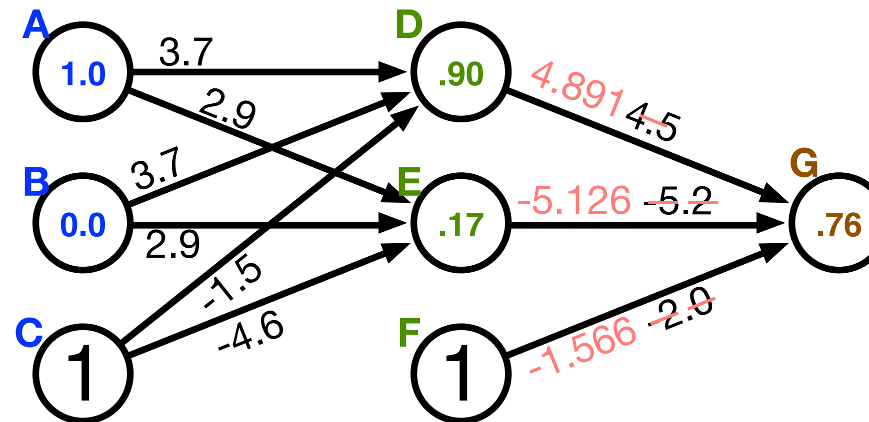
$$\Delta w_{j \leftarrow k} = \mu\ \delta_j\ h_k$$

Our Example



- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$)
  - $\delta_{\text{G}} = (t - y)\ y' = (1 - .76)\ 0.181 = .0434$
  - $\Delta w_{\text{GD}} = \mu\ \delta_{\text{G}}\ h_{\text{D}} = 10 \times .0434 \times .90 = .391$
  - $\Delta w_{\text{GE}} = \mu\ \delta_{\text{G}}\ h_{\text{E}} = 10 \times .0434 \times .17 = .074$
  - $\Delta w_{\text{GF}} = \mu\ \delta_{\text{G}}\ h_{\text{F}} = 10 \times .0434 \times 1 = .434$

Our Example



- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$)
  - $\delta_{\mathsf{G}} = (t - y)\, y' = (1 - .76)\; 0.181 = .0434$
  - $\Delta w_{\mathsf{GD}} = \mu\; \delta_{\mathsf{G}}\; h_{\mathsf{D}} = 10 \times .0434 \times .90 = .391$
  - $\Delta w_{\mathsf{GE}} = \mu\; \delta_{\mathsf{G}}\; h_{\mathsf{E}} = 10 \times .0434 \times .17 = .074$
  - $\Delta w_{\mathsf{GF}} = \mu\; \delta_{\mathsf{G}}\; h_{\mathsf{F}} = 10 \times .0434 \times 1 = .434$

Batches

- Each training example yields a set of weight updates Δw_i.
- Batch up several training examples
  - sum up their updates
  - apply sum to model
- Mostly done or speed reasons

computational aspects

Vector and Matrix Multiplications

- Forward computation: $\vec{s} = W\vec{h}$
- Activation function: $\vec{y} = \text{sigmoid}(\vec{h})$
- Error term: $\vec{\delta} = (\vec{t} - \vec{y}) \text{ sigmoid}'(\vec{s})$
- Propagation of error term: $\vec{\delta}_i = W\vec{\delta}_{i+1} \cdot \text{sigmoid}'(\vec{s})$
- Weight updates: $\Delta W = \mu\vec{\delta}\vec{h}^T$

GPU

- Neural network layers may have, say, 200 nodes
- Computations such as $W\vec{h}$ require $200 \times 200 = 40,000$ multiplications
- Graphics Processing Units (GPU) are designed for such computations
  - image rendering requires such vector and matrix operations
  - massively multi-core but lean processing units
  - example: NVIDIA H100 GPU provides 18,432 CUDA cores
- Extensions to C to support programming of GPUs, such as CUDA

Explosion of Deep Learning Toolkits

- University of Montreal: Theano (early, now defunct)
- Google: Tensorflow
- Facebook: Torch, pyTorch
- Microsoft: CNTK
- Amazon: MX-Net
- CMU: Dynet
- AMU/Edinburgh/Microsoft: Marian
- ... and many more

Toolkits

- Machine learning architectures around computations graphs very powerful
  - define a computation graph
  - provide data and a training strategy (e.g., batching)
  - toolkit does the rest
  - seamless support of GPUs

Example: PyTorch

- Installation

```
pip install torch
```

- Usage

```
import torch
```

Some Data Types

- PyTorch data type for parameter vectors, matrices etc., called `torch.tensor`

```
W = torch.tensor([[3,4],[2,3]], requires_grad=True, dtype=torch.float)
b = torch.tensor([-2,-4], requires_grad=True, dtype=torch.float)
W2 = torch.tensor([5,-5], requires_grad=True, dtype=torch.float)
b2 = torch.tensor([-2], requires_grad=True, dtype=torch.float)
```

- Definition of variables includes
  - specification of their basic data type (`float`)
  - indication to compute gradients (`requires_grad=True`)

- Input and output

```
x = torch.tensor([1,0], dtype=torch.float)
t = torch.tensor([1], dtype=torch.float)
```

Computation Graph

- Computation graph

```
s = W.mv(x) + b
h = torch.nn.Sigmoid()(s)

z = torch.dot(W2, h) + b2
y = torch.nn.Sigmoid()(z)

error = 1/2 * (t - z) ** 2
```

- Note
  - PyTorch sigmoid function `torch.nn.Sigmoid()`
  - multiplication between matrix `W` and vector `x` is `mv`
  - multiplication between two vectors `W2` and `h` is `torch.dot`.

Backward Computation

- Here it is:

```
error.backward()
```

- No need to derive gradients — all is done automatically

- We can look up computed gradients

```
>>> W2.grad
tensor([-0.0360, -0.0059])
```

- Note
  - when you run this code multiple times, then gradients accumulate
  - reset them with, e.g., `W2.grad.data.zero_()`

Training Data

- Our training set consists of the four examples of binary XOR operations.

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

- Placed into array

```
training_data =
    [ [ torch.tensor([0.,0.]), torch.tensor([0.])  ],
      [ torch.tensor([1.,0.]), torch.tensor([1.])  ],
      [ torch.tensor([0.,1.]), torch.tensor([1.])  ],
      [ torch.tensor([1.,1.]), torch.tensor([0.])  ]  ]
```

Training Loop: Forward

```
mu = 0.1

for epoch in range(1000):
  total_error = 0

  for item in training_data:
    x = item[0]
    t = item[1]

    # forward computation
    s = W.mv(x) + b
    h = torch.nn.Sigmoid()(s)
    z = torch.dot(W2, h) + b2
    y = torch.nn.Sigmoid()(z)
    error = 1/2 * (t - y) ** 2
    total_error = total_error + error
```

Training Loop: Backward and Updates


```
    # backward computation
    error.backward()

    # weight updates
    W.data  = W  - mu * W.grad.data
    b.data  = b  - mu * b.grad.data
    W2.data = W2 - mu * W2.grad.data
    b2.data = b2 - mu * b2.grad.data

    W.grad.data.zero_()
    b.grad.data.zero_()
    W2.grad.data.zero_()
    b2.grad.data.zero_()

  print("error: ", total_error/4)
```

Batch Training

- We computed gradients for each training example, update model immediately
- More common: process examples in batches, update after batch processed
- Instead

```
error.backward()
```

- Run back-propagation on accumulated error

```
total_error.backward()
```

Training Data Batch

```
x = torch.tensor([ [0.,0.], [1.,0.], [0.,1.], [1.,1.]  ])
t = torch.tensor([ 0., 1., 1., 0.  ])
```

- Change to computation graph (input now a matrix, output a vector)

```
s = x.mm(W) + b
h = torch.nn.Sigmoid()(s)
z = h.mv(W2) + b2
y = torch.nn.Sigmoid()(z)
```

- Convert error vector into single number

```
error = 1/2 * (t - y) ** 2
mean_error = error.mean()
mean_error.backward()
```

Parameter Updates (Optimizer)

- Our code has explicit parameter update computations

```
# weight updates
W.data  = W  - mu * W.grad.data
b.data  = b  - mu * b.grad.data
W2.data = W2 - mu * W2.grad.data
b2.data = b2 - mu * b2.grad.data
```

- But fancier optimizers are typically used (Adam, etc.)
- This requires more complex implementation

torch.nn.Module

- Neural network model is defined as class derived from `torch.nn.Module`

```
class ExampleNet(torch.nn.Module):

  def __init__(self):
    super(ExampleNet, self).__init__()
    self.layer1 = torch.nn.Linear(2,2)
    self.layer2 = torch.nn.Linear(2,1)
    self.layer1.weight = torch.nn.Parameter(torch.tensor([[3.,2.],[4.,3.]]))
    self.layer1.bias = torch.nn.Parameter(torch.tensor([-2.,-4.]))
    self.layer2.weight = torch.nn.Parameter(torch.tensor([[5.,-5.]]))
    self.layer2.bias = torch.nn.Parameter(torch.tensor([-2.]))

  def forward(self, x):
    s = self.layer1(x)
    h = torch.nn.Sigmoid()(s)
    z = self.layer2(h)
    y = torch.nn.Sigmoid()(z)
    return y
```

Optimizer Definition

- Instantiation of neural network object

```
net = ExampleNet()
```

- Optimizer definition

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
```

Training Loop


```
for iteration in range(1000):
  optimizer.zero_grad()
  out = net.forward( x )
  error = 1/2 * (t - out) ** 2
  mean_error = error.mean()
  print("error:  ",mean_error.data)
  mean_error.backward()
  optimizer.step()
```

code available on web page for textbook

http://www.statmt.org/nmt-book/