Backpropagation: the principle

\[
\frac{\partial C}{\partial w_*} = ?
\]
Backpropagation: the principle

\[ \frac{\partial C}{\partial w_*} = ? \]

\[ y_{\text{out}} - F(y_{\text{in}}) \]

(omitting indices, should be clear from figure)
Backpropagation: the principle

\[
\frac{\partial C}{\partial w_*} =? \quad y^{\text{out}} - F(y^{\text{in}}) \quad f'(z)
\]

(omitting indices, should be clear from figure)
Backpropagation: the principle

\[
\frac{\partial C}{\partial w_*} = ?
\]

\[
y_{\text{out}} - F(y_{\text{in}})
\]

\[
f'(z)
\]

\[
f'(z)
\]
\[
\frac{\partial C}{\partial w_*} = ?
\]

(omitting indices, should be clear from figure)
\[
\frac{\partial C}{\partial w_*} = y^\text{out} - F(y^\text{in})
\]
\[
\frac{\partial C}{\partial w_*} = ?
\]

and now: sum over ALL possible paths!

similar to Feynman sum over paths (path integral)

efficient implementation: repeated matrix/vector multiplication

(omitting indices, should be clear from figure)
def net_f_df(z):  # calculate f(z) and f'(z)
    val=1/(1+exp(-z))
    return(val,exp(-z)*(val**2))  # return both f and f'

def forward_step(y,w,b):  # calculate values in next layer
    z=dot(y,w)+b  # w=weights, b=bias vector for next layer
    return(net_f_df(z))  # apply nonlinearity

def apply_net(y_in):  # one forward pass through the network
    global Weights, Biases, NumLayers
    global y_layer, df_layer  # store y-values and df/dz
    y=y_in  # start with input values
    y_layer[0]=y
    for j in range(NumLayers):  # loop through all layers
        y,df=forward_step(y,Weights[j],Biases[j])
        df_layer[j]=df  # store f'(z)
        y_layer[j+1]=y  # store f(z)
    return(y)

def backward_step(delta,w,df):
    return(dot(delta,transpose(w)) * df)

def backprop(y_target):  # one backward pass
    global y_layer, df_layer, Weights, Biases, NumLayers
    global dw_layer, db_layer  # dCost/dw and dCost/db
    #(w,b=weights,biases)
    global batchsize
    delta=(y_layer[-1]-y_target)*df_layer[-1]
    dw_layer[-1]=dot(transpose(y_layer[-2]),delta)/batchsize
    db_layer[-1]=delta.sum(0)/batchsize
    for j in range(NumLayers-1):
        delta=backward_step(delta,Weights[-1-j],df_layer[-2-j])
        dw_layer[-2-j]=dot(transpose(y_layer[-3-j]),delta)/batchsize
        db_layer[-2-j]=delta.sum(0)/batchsize
Neural networks: the ingredients

General purpose algorithm: feedforward & backpropagation (implement once, use often)

**Problem-specific:**
Choose network layout (number of layers, number of neurons in each layer, type of nonlinear functions, maybe specialized structures of the weights) “Hyperparameters”

Generate training (& validation & test) samples: load from big databases (that have to be compiled from the internet or by hand!) or produce by software

Monitor/optimize training progress (possibly choose learning rate and batch size or other parameters, maybe try out many combinations) “Hyperparameters”
Example: Learning a 2D function

Evaluate at sample points

See notebook (on website): **MultiLayerBackProp**
Example: Learning a 2D function

see notebook (on website): **MultiLayerBackProp**

```python
# pick batchsize random positions in the 2D square
def make_batch():
    global batchsize

    inputs = random.uniform(low=-0.5, high=+0.5, size=[batchsize, 2])
    targets = zeros([batchsize, 1])  # must have right dimensions
    targets[:, 0] = myFunc(inputs[:, 0], inputs[:, 1])
    return (inputs, targets)

eta = 0.1
batchsize = 1000
batches = 2000
costs = zeros(batches)

for k in range(batches):
    y_in, y_target = make_batch()
    costs[k] = train_net(y_in, y_target, eta)
```
Example: Learning a 2D image

see notebook (on website): MultiLayer_ImageCompression
Network layers: 2, 150, 150, 100, 1 neurons
(after about 2min of training, ~4 Mio. samples)
Reminder: ReLU (rectified linear unit)

\[ f(z) = \begin{cases} 
  z & \text{for } z > 0 \\
  0 & \text{for } z \leq 0 
\end{cases} \]

\[ z = wy + b \]
à la Franz Marc?
Try to understand how the network operates!
Switching on only a single neuron of the last hidden layer,

Image shows results of switching on individually each of 100 neurons.
Weights from last hidden layer to output

- Deleted first 50 weights
- Deleted last 50 weights
- Kept only 10 out of 100

Weights from 2nd hidden layer to last hidden layer

- Deleted first 75
- Deleted last 75
- Kept only 10 out of 150
Weights from 1st hidden layer to 2nd hidden layer

- deleted first 75
- deleted last 75
- kept only 10 out of 150
Influence of learning rate (stepsize) 

eta=0.1

(batchsize=1000)
Influence of learning rate (stepsize)

(batchsize=1000)

Cost vs. batch for different learning rates:
- eta=0.1
- eta=0.2
Influence of learning rate (stepsize)

(batchsize=1000)
Influence of learning rate (stepsize)

(batchsize=1000)

Cost

eta=0.1
eta=0.2
eta=0.5
eta=1.0
Randomness (initial weights, learning samples)

Learning is a stochastic, nonlinear process!

(batchsize=1000)  all: $\eta=1.0$
Influence of batch size / learning rate

Small batch size and large learning rate together are problematic!

(batchsize=20)

Cost

eta=0.1
eta=0.2
eta=0.5
eta=1.0
Influence of batch size / learning rate

\[ C(w - \eta \nabla_w C) \approx C(w) - \eta (\nabla_w C)(\nabla_w C) + \ldots \]

new weights
\[ \text{always} \ > 0 \]
decrease in C!
\[ \text{higher order in } \eta \]
Influence of batch size / learning rate

\[ C(w - \eta \nabla_w C) \approx C(w) - \eta (\nabla_w C)(\nabla_w C) + \ldots \]

always >0

decrease in C!

Potential problems:

- step too large: need higher-order terms
  [will not be a problem near minimum of C]
- approx. of C bad [small batch size: approx. C fluctuates]

Sufficiently small learning rate: multiple training steps (batches) add up, and their average is like a larger batch
Programming a general multilayer neural network & backpropagation was not so hard (once you know it!)
Could now go on to image recognition etc. with the same program!
Programming a general multilayer neural network & backpropagation was not so hard (once you know it!)
Could now go on to image recognition etc. with the same program!

But: want more flexibility and added features!

For example:
• Arbitrary nonlinear functions for each layer
• Adaptive learning rate
• More advanced layer structures (such as convolutional networks)
• etc.
• Convenient neural network package for python
• Set up and training of a network in a few lines
• Based on underlying neural network / symbolic differentiation package [which also provides run-time compilation to CPU and GPU]: either ‘theano’ or ‘tensorflow’ [User does not care]
“Keras is a high-level neural networks API, written in Python and capable of running on top of either TensorFlow or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.”
from keras import *
from keras.models import Sequential
from keras.layers import Dense

Defining a network

layers with 2, 150, 150, 100, 1 neurons

net = Sequential()
net.add(Dense(150, input_shape=(2,), activation='relu'))
net.add(Dense(150, activation='relu'))
net.add(Dense(100, activation='relu'))
net.add(Dense(1, activation='relu'))

‘Compiling’ the network

net.compile(loss='mean_squared_error',
            optimizer=optimizers.SGD(lr=0.1),
            metrics=['accuracy'])
from keras import *
from keras.models import Sequential
from keras.layers import Dense

Defining a network

“Sequential”: the usual neural network, with several layers

net=Sequential()
net.add(Dense(150, input_shape=(2,), activation='relu'))
net.add(Dense(150, activation='relu'))
net.add(Dense(100, activation='relu'))
net.add(Dense(1, activation='relu'))

“Dense”: “fully connected layer” (all weights there)
input_shape: number of input neurons

‘Compiling’ the network

SGD=stoch. gradient descent
net.compile(loss='mean_squared_error',
optimizer=optimizers.SGD(lr=0.1),
metrics=['accuracy'])

‘relu’: rectified linear unit
‘loss’=cost
lr=learning rate=stepsize
Training the network

```
batchsize=20
batches=200
costs=zeros(batches)

for k in range(batches):
    y_in, y_target = make_batch()
    costs[k] = net.train_on_batch(y_in, y_target)[0]
```

*y_in*    array dimensions ‘batchsize’ x 2
*y_target*  array dimensions ‘batchsize’ x 1
(just like before, for our own python code)
Predicting with the network

```python
y_out = net.predict_on_batch(y_in)
```

- `y_in` array dimensions ‘batchsize’ x 2
- `y_out` array dimensions ‘batchsize’ x 1

(just like before, for our own python code)
Explore how well the network can reproduce various features of target images, and how that depends on the network layout!

Aspects to consider (& I do not claim to know all the answers!):

How good are other nonlinear functions? [e.g. sigmoids or your own favorite $f(z)$]

Given a fixed total number of weights, is it better to go deep (many layers) or shallow?

Bonus: After training, try to ‘prune’ the network, i.e. delete neurons whose deletion does not increase the cost function too much!