Deep Learning

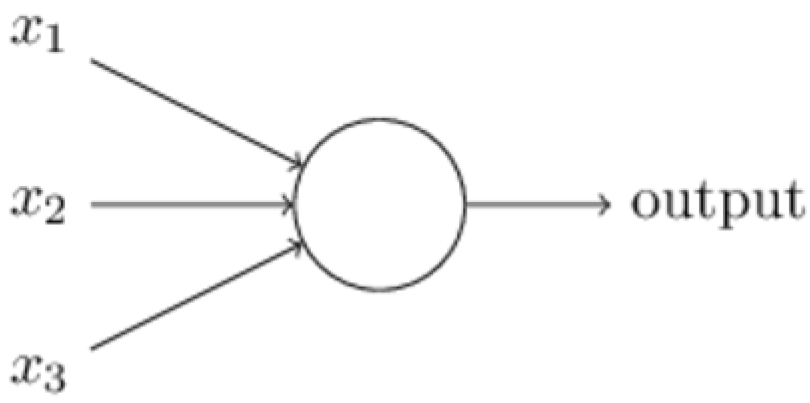
Training neural networks

Danijel Skočaj
University of Ljubljana
Faculty of Computer and Information Science

Academic year: 2022/23

Perceptron

- Rosenblatt, 1957
- Binary inputs and output
- Weights
- Threshold
- Bias
- Very simple!

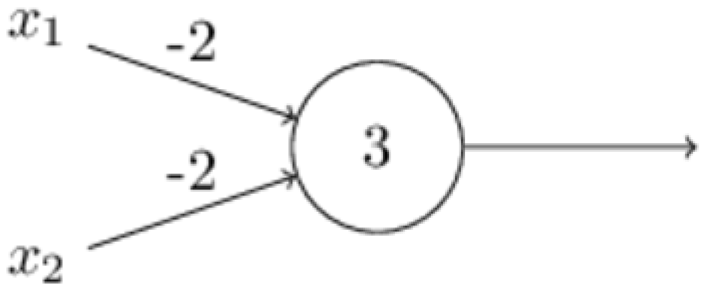


$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

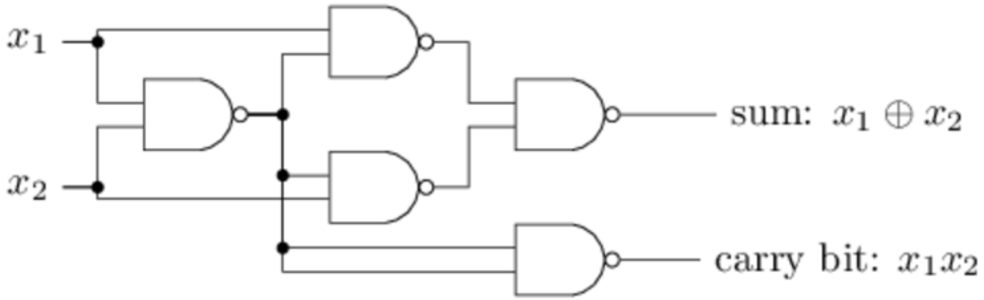
$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

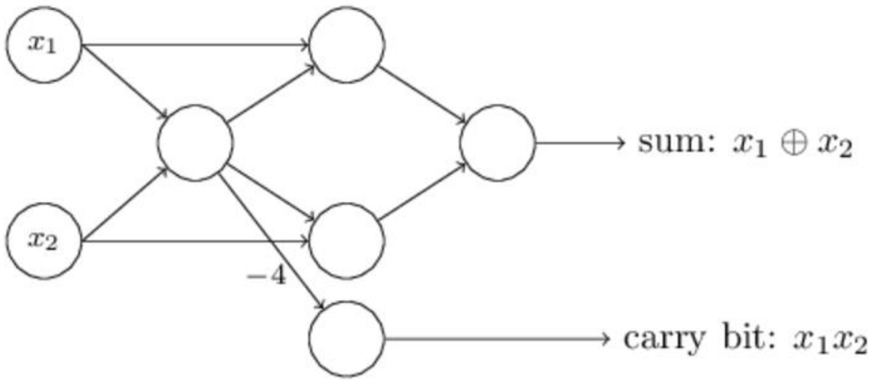
Example: logical functions

- NAND gate:



- Addition circuit:

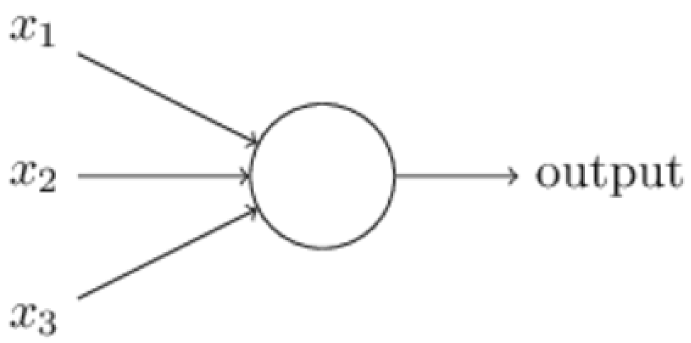


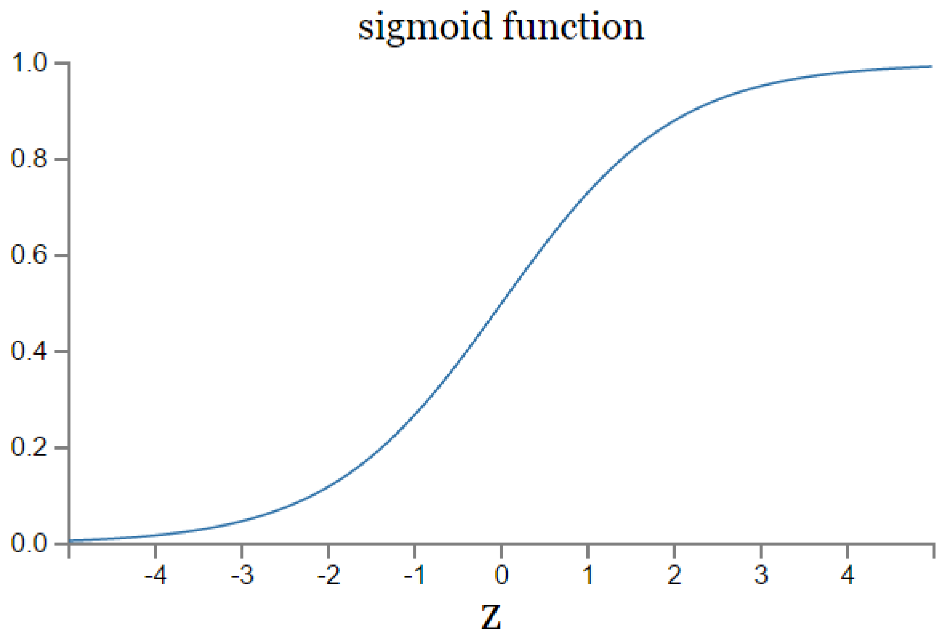


- Go beyond binary inputs/outputs
- Learn weights!

Sigmoid neurons

- Real inputs and outputs from interval [0,1]





- Activation function: sigmoid function

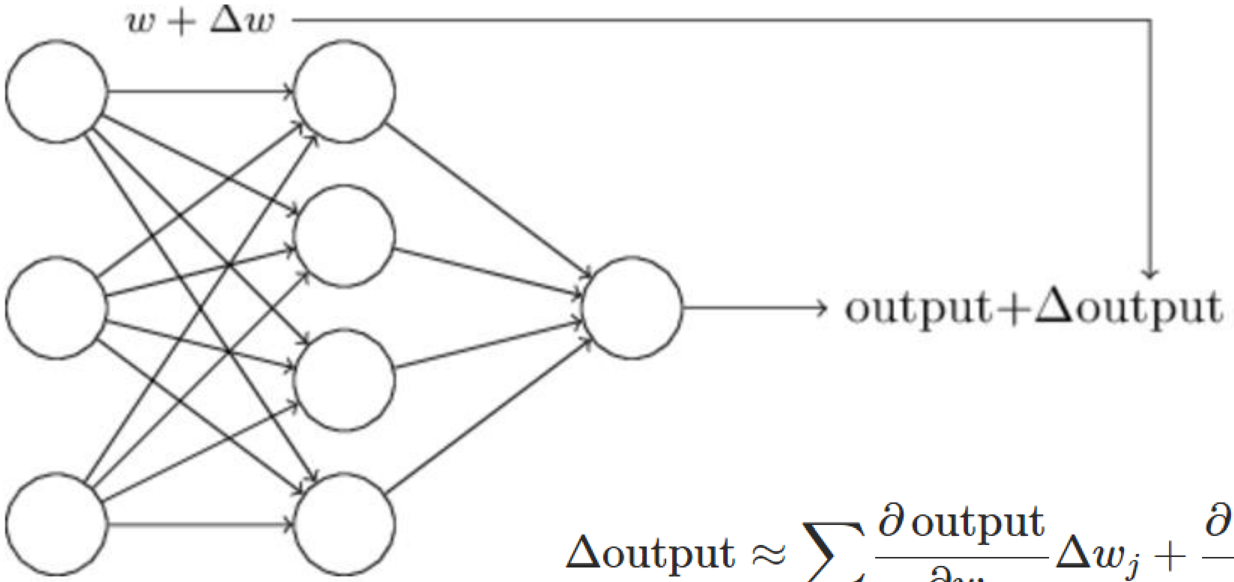
- *output* = $\dfrac{1}{1+\exp(-\sum_j w_j x_j - b)}$

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$

$$\sigma(w \cdot x + b)$$

Sigmoid neurons

- Small changes in weights and biases causes small change in output

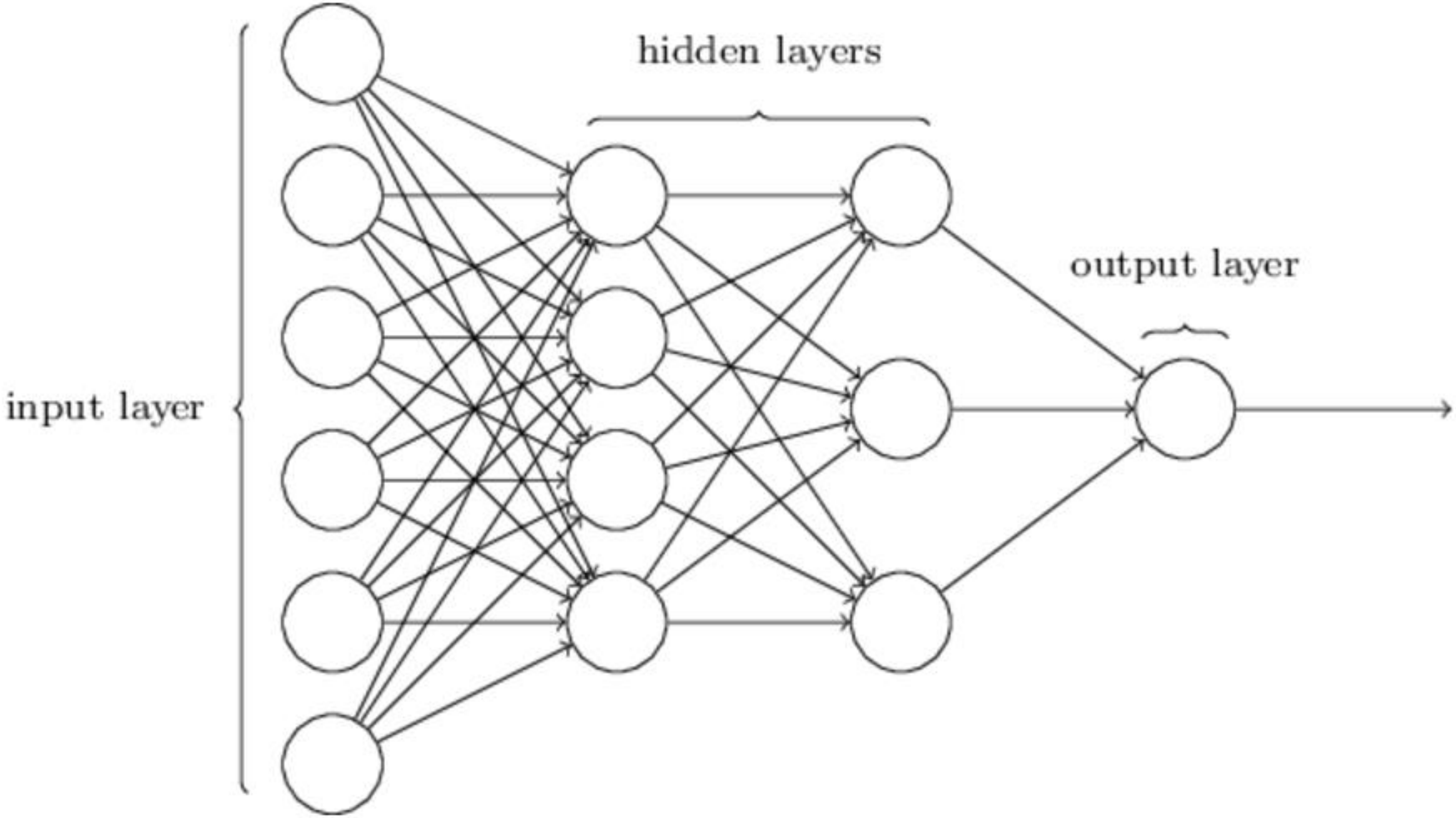


$$\Delta \text{output} \approx \sum_j \frac{\partial\, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial\, \text{output}}{\partial b} \Delta b$$

- Enables learning!

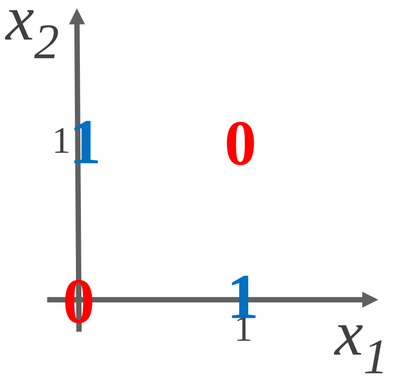
Feedfoward neural networks

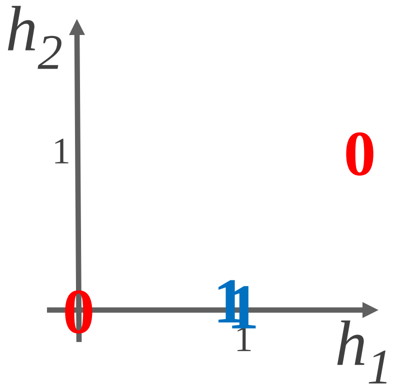
- Network architecture:



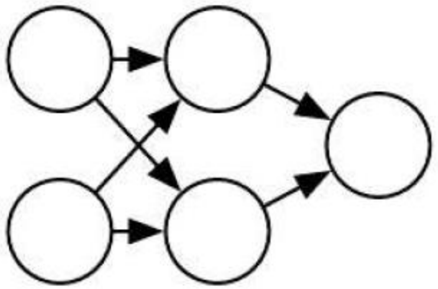
Example: XOR

- Not linearly separable function!



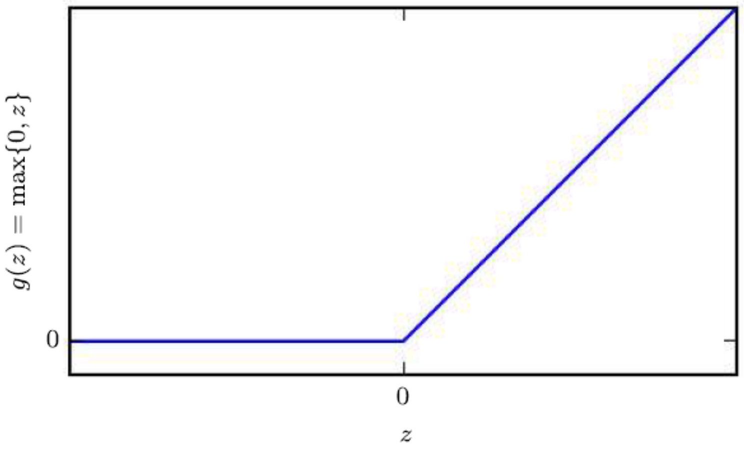


- Hidden neuron needed:



- Linearly separable in feature space!

- Activation function: ReLU



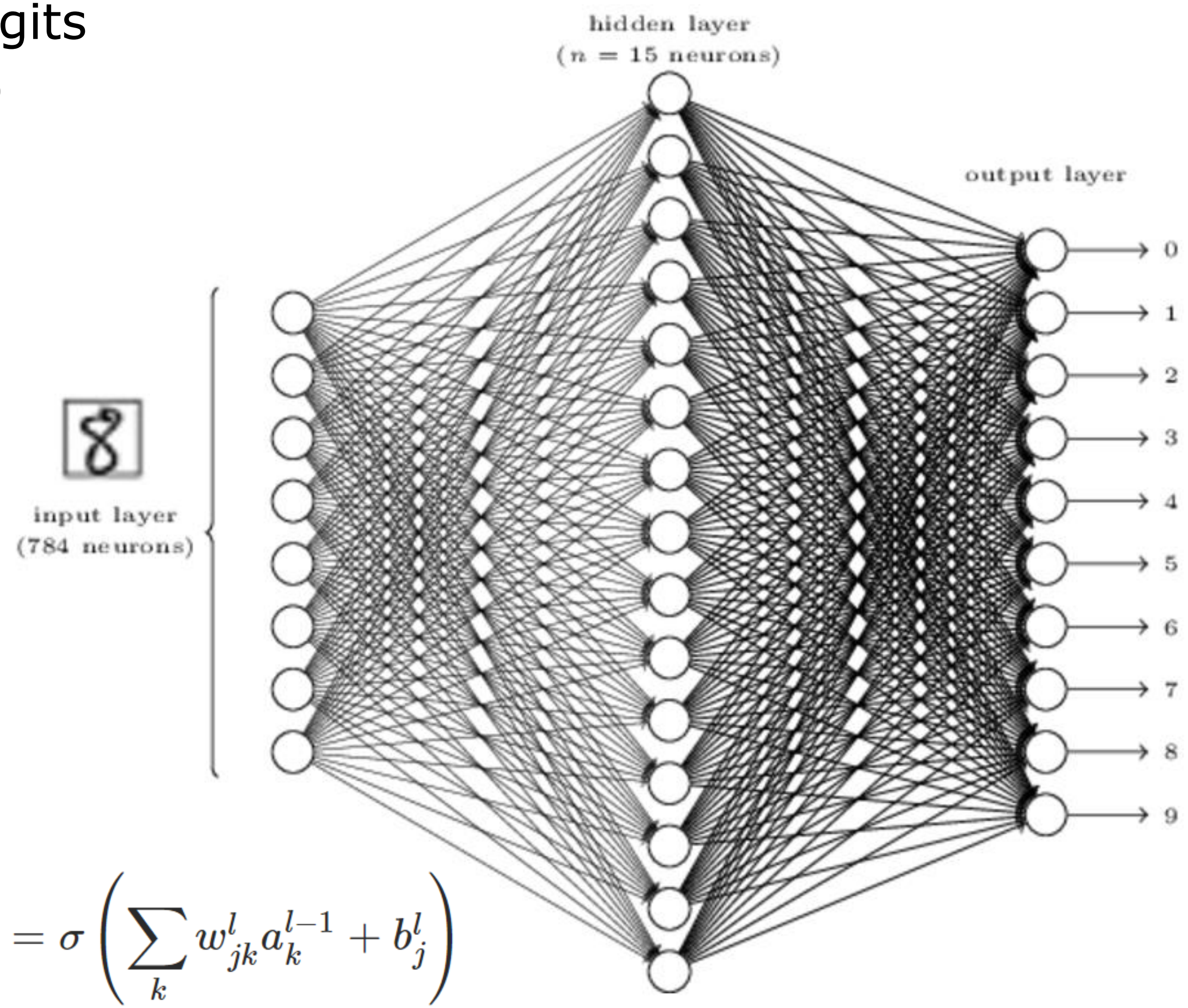
$$h = f^{(1)}(x; w^{(1)}, b^{(1)}) = max(0, w^{(1)} + b^{(1)})$$

$$a = f^{(2)}(f^{(1)}(x)) = f^{(2)}(h; w^{(2)}, b^{(2)}) =$$
$$= w^{(2)} max(0, w^{(1)} + b^{(1)}) + b^{(2)}$$

$$w^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \quad w^{(2)} = \begin{bmatrix} 1 & -2 \end{bmatrix} \quad b^{(2)} = 0$$

Example: recognizing digits

- MNIST database of handwritten digits
  - 28x28 pixes (=784 input neurons)
  - 10 digits
  - 50.000 training images
  - 10.000 validation images
  - 10.000 test images



$$
\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}
$$

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

Example code: Feedforward

- Code from http://neuralnetworksanddeeplearning.com/
  or https://github.com/mnielsen/neural-networks-and-deep-learning
- or https://github.com/MichalDanielDobrzanski/DeepLearningPython35 (for Python 3)

Nielsen, 2015

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a


def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

```
net = network.Network([784, 30, 10])
net.SGD(training_data, 5, 10, 3.0, test_data=test_data)
```

```
In [55]: x,y=test_data[0]

In [56]: net.feedforward(x)
Out[56]:
array([[  1.83408119e-03],
       [  5.94472468e-08],
       [  1.84785949e-03],
       [  6.85718810e-04],
       [  1.41399919e-05],
       [  5.40491233e-06],
       [  4.74332685e-09],
       [  9.97920007e-01],
       [  8.19370561e-05],
       [  6.65086583e-05]])

In [57]: y
Out[57]: 7
```

Loss function

- Given:

$y\Big($  $\Big) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ for all training images

- Loss function: $C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$
  - (mean sqare error – quadratic loss function)

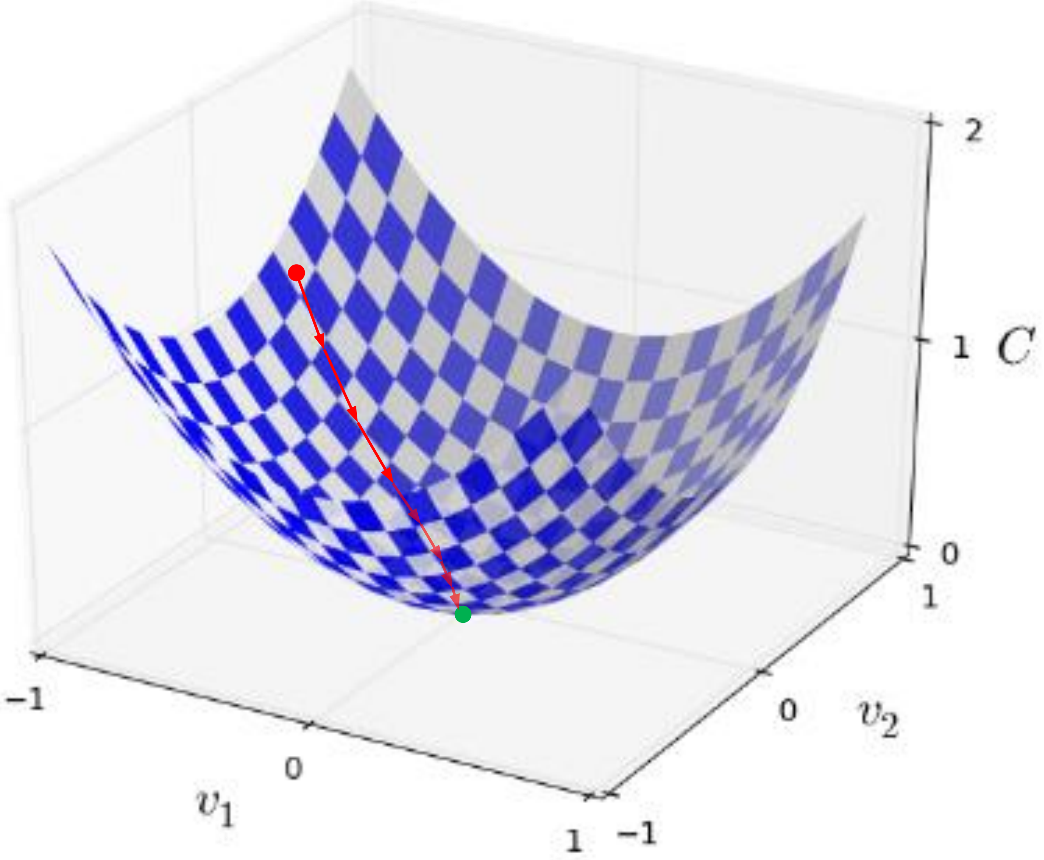
- Find weigths w and biases b that for given input x produce output a that minimizes Loss function C

Gradient descend

- Find minimum of $C(v_1, v_2)$
- Change of C: $\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 = \nabla C \cdot \Delta v = -\eta\|\nabla C\|^2$
- Gradient of C: $\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$
- *Change v in the opposite direction of the gradient:* $\Delta v = -\eta \nabla C$

Learning rate

- Algorithm:
  - Initialize v
  - Until stopping criterium riched
    - Apply udate rule $v \rightarrow v' = v - \eta \nabla C.$



Gradient descend in neural networks

- Loss function $C(w,b)$

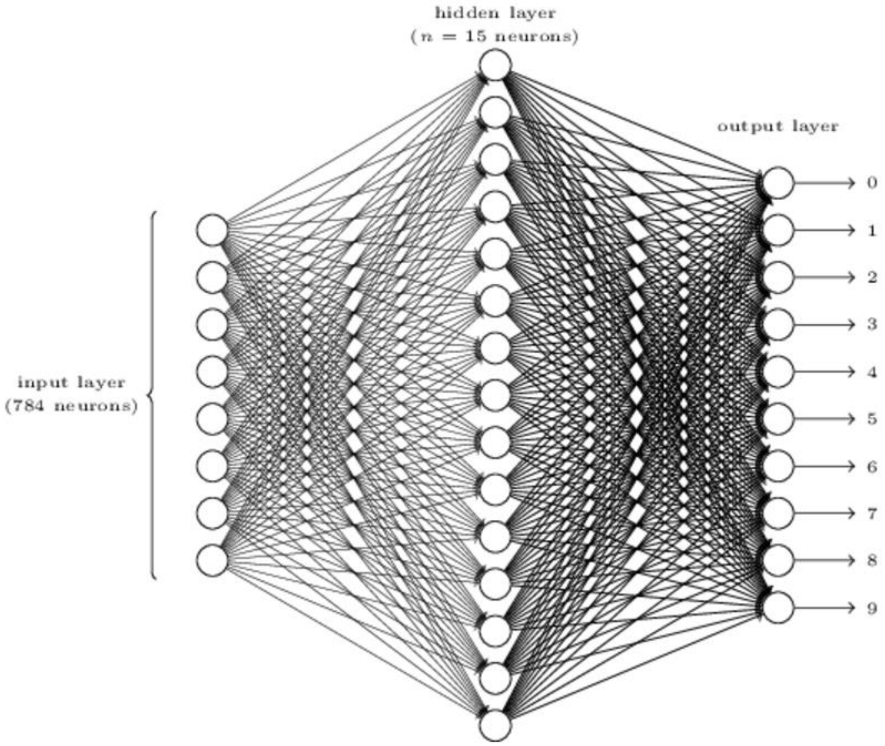
- Update rules:

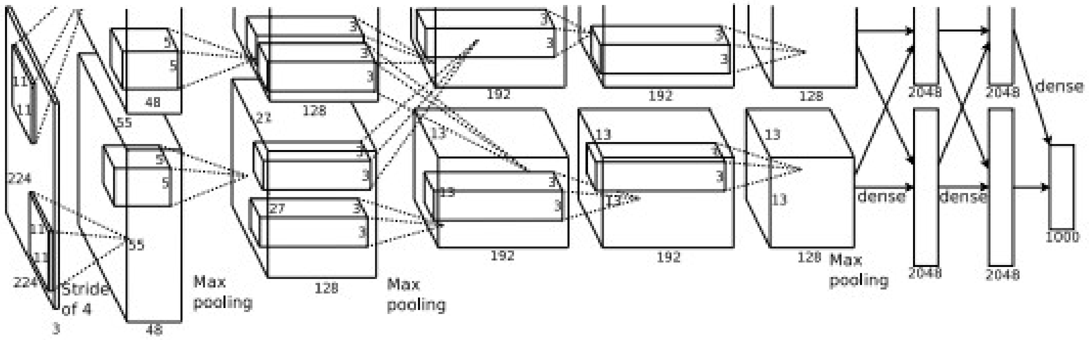
$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

- Consider all training samples
- Very many parameters
  => computationaly very expensive

- Use Stochastic gradient descend instead





Stochastic gradient descend

- Compute gradient only for a subset of m training samples:
  - *Mini-batch*: $X_1, X_2, \ldots, X_m$
  - Approximate gradient: $\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C$ $\qquad \nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}$
  - Update rules:

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

- Training:
  1. Initialize w and b
  2. In one *epoch* of training keep randomly selecting one mini-batch of m samples at a time (and train) until all training images are used
  3. Repeat for several epochs

Example code: SGD

```
def SGD(self, training_data, epochs, mini_batch_size, eta):
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
```

```
def update_mini_batch(self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

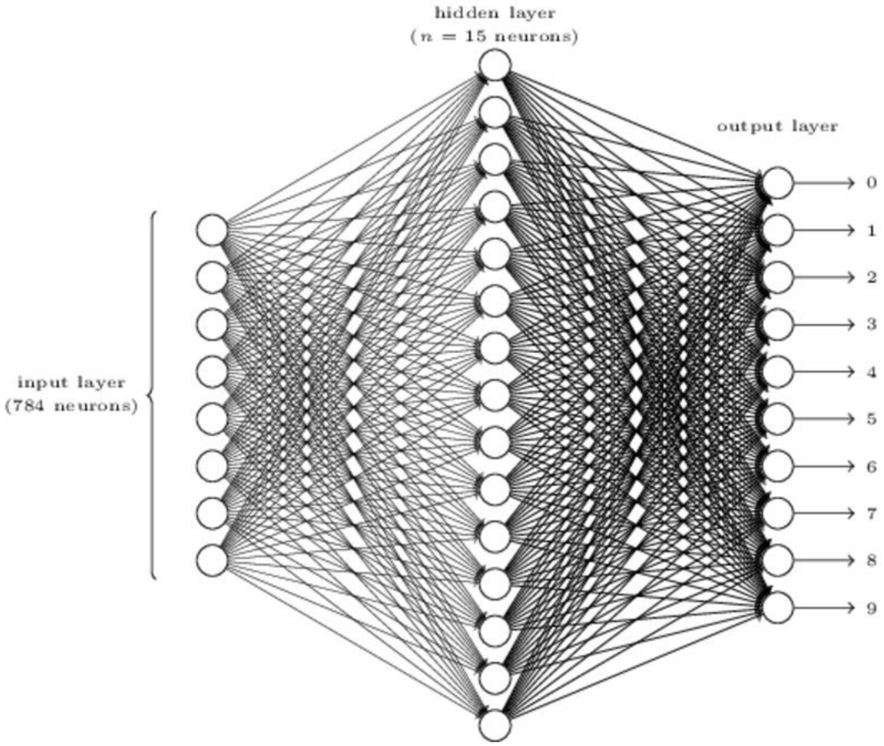
$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

Backpropagation

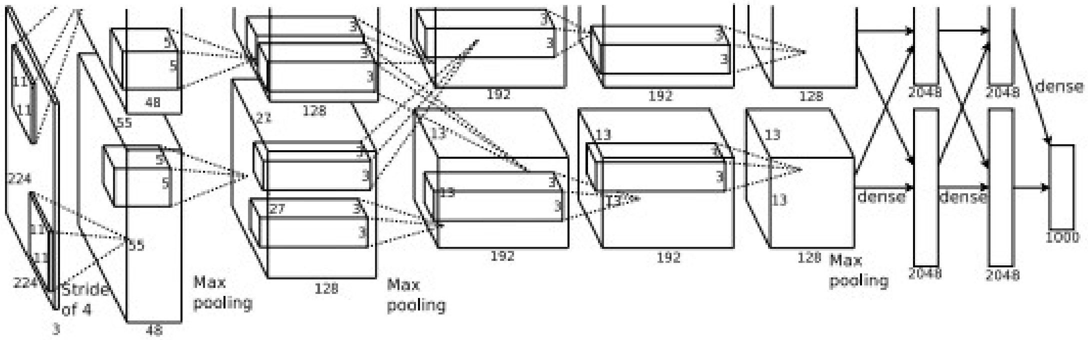
- All we need is gradient of loss function ∇C
  - Rate of change of C wrt. to change in any weigt
  - Rate of change of C wrt. to change in any bias

$$\frac{\partial C}{\partial b^l_j} \qquad \frac{\partial C}{\partial w^l_{jk}}$$



- How to compute gradient?
  - Numericaly
    - Simple, approximate, extremely slow ☹
  - Analyticaly for entire C
    - Fast, exact, nontractable ☹
  - Chain individual parts of network
    - Fast, exact, doable ☺

Backpropagation!



Backpropagation in computational graph

$$f(x,y,z) = (x+y)(y-z) = wv$$

$$\frac{\partial f}{\partial x} = y - z = 1$$

$$\frac{\partial f}{\partial y} = y - z + x + y = x + 2y - z = 6$$

$$\frac{\partial f}{\partial z} = -(x+y) = -5$$

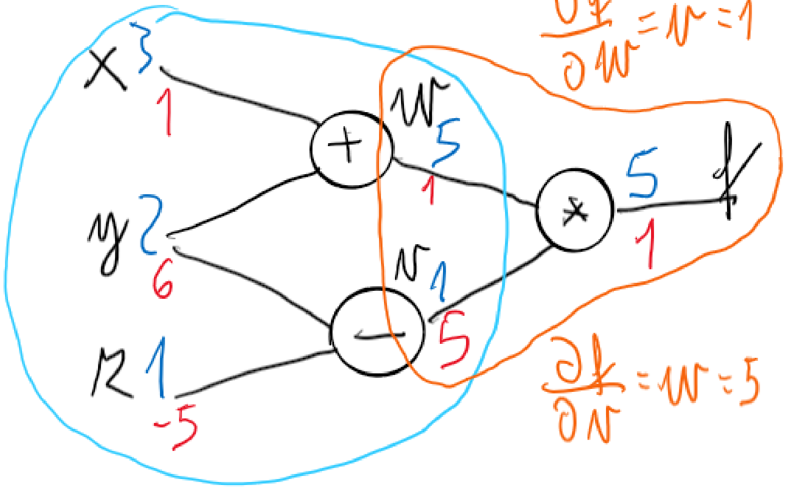
$$w = x + y \qquad f = wv$$

$$\frac{\partial w}{\partial x} = 1 \qquad \frac{\partial f}{\partial x} = \frac{\partial f}{\partial w}\frac{\partial w}{\partial x} = 1 \cdot 1 = 1$$

$$\frac{\partial w}{\partial y} = 1$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial w}\frac{\partial w}{\partial y} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial y} = 1 \cdot 1 + 5 \cdot 1 = 6$$

$$\frac{\partial f}{\partial w} = v = 1 \qquad \frac{\partial f}{\partial v} = w = 5 \qquad \frac{\partial f}{\partial f} = 1$$

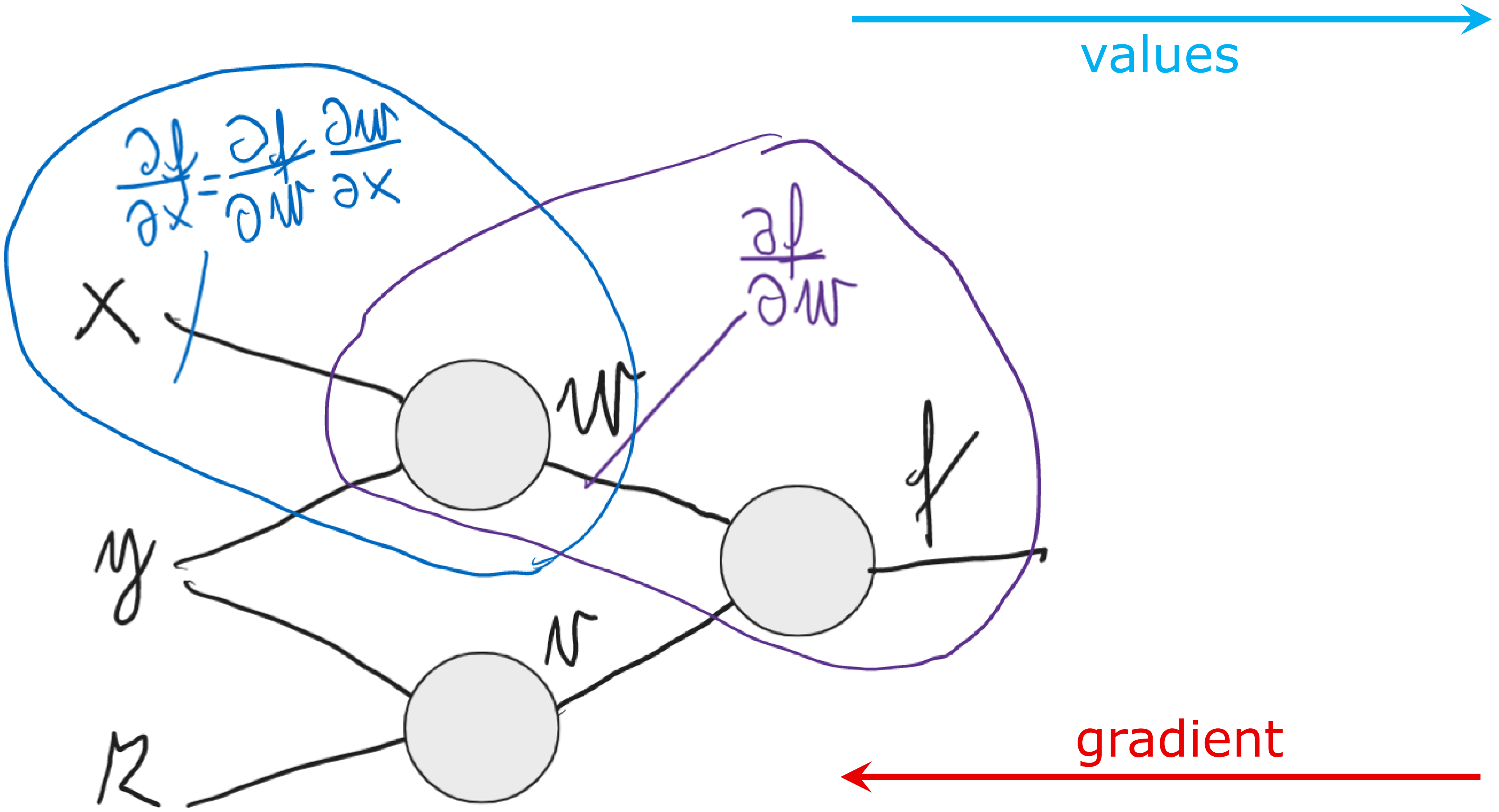


$$v = y - z$$

$$\frac{\partial v}{\partial y} = 1$$

$$\frac{\partial v}{\partial z} = -1 \qquad \frac{\partial f}{\partial z} = \frac{\partial f}{\partial v}\frac{\partial v}{\partial z} = 5 \cdot (-1) = -5$$

Locality of computation



Gradient backward flow

- Addition
  - Unchanged gradient value travels back

$$f(x,y) = x + y \qquad \frac{\partial f}{\partial x} = 1 \qquad \frac{\partial f}{\partial y} = 1$$

- Multiplication
  - Gradient multiplies with switched values

$$f(x,y) = xy \qquad \frac{\partial f}{\partial x} = y \qquad \frac{\partial f}{\partial y} = x$$

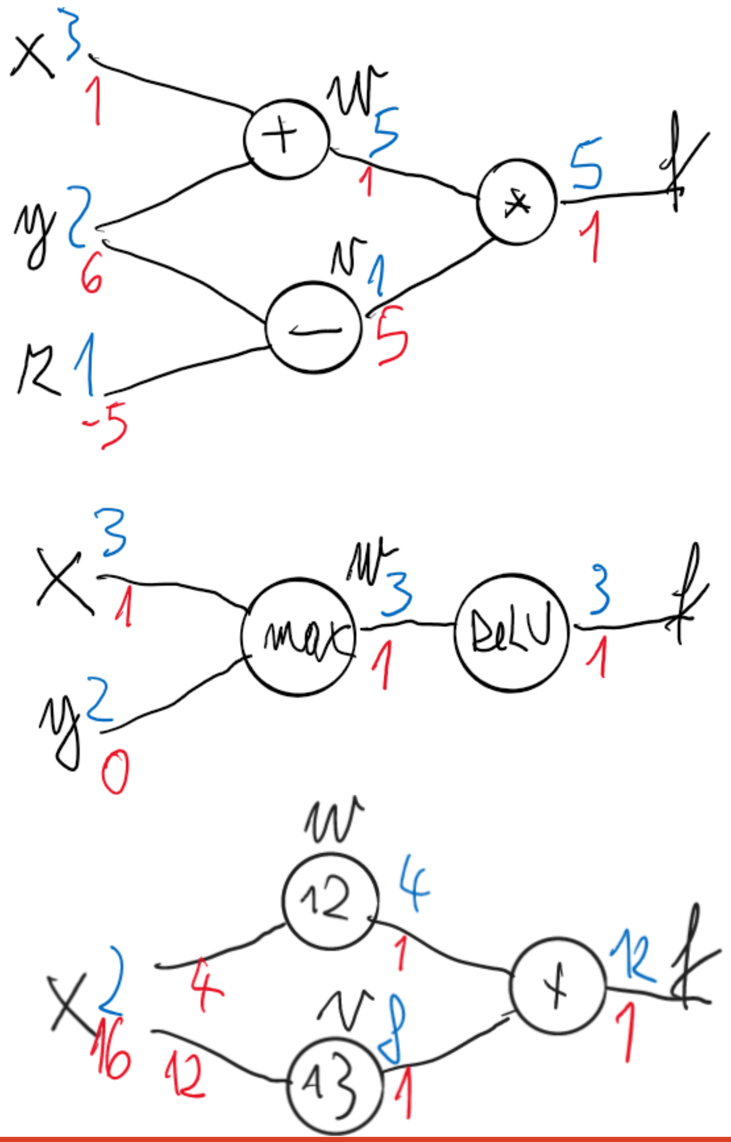
- Maximisation
  - Gradient routes back through the max. branch

$$f(x,y) = \max(x,y) \quad \frac{\partial f}{\partial x} = 1(x >= y) \quad \frac{\partial f}{\partial y} = 1(y >= x)$$

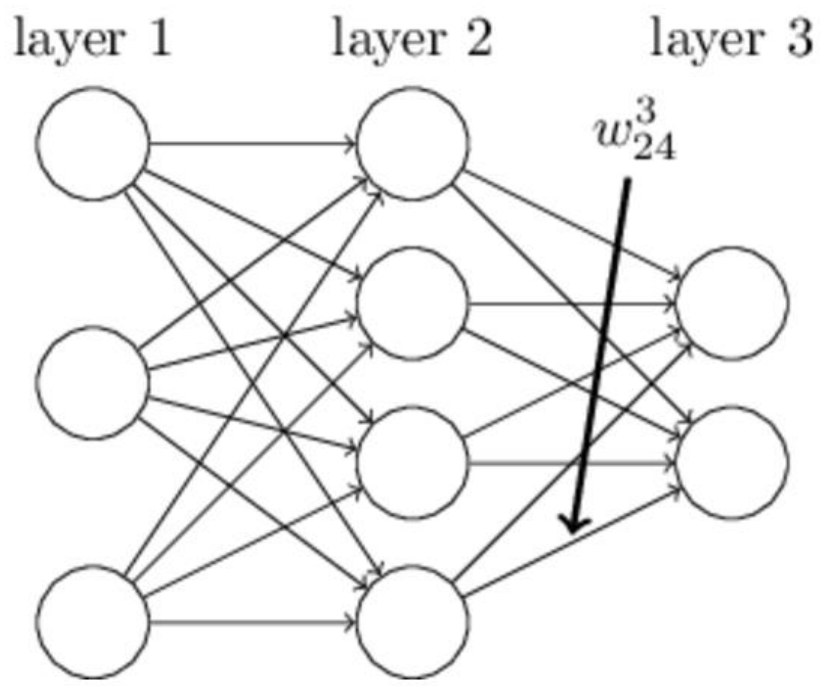
- ReLU
  - Gradient flows back for positive and stops if negative
- Branching

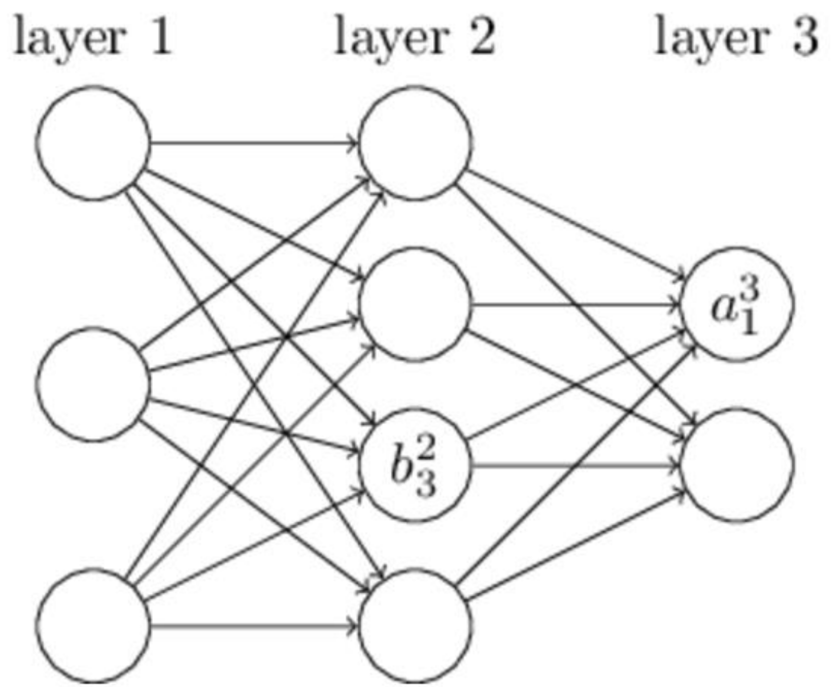
$$f(x) = ReLU(x) \qquad \frac{\partial f}{\partial x} = 1(x > 0)$$

  - Gradients of all branches added



Notation: *w, b*





- w^l_{jk} is the weight
  from the k-th neuron in the (l-1) layer
  to the j-th neuron in the l-th layer
- w^l : weigth matrix for the l-th layer
- b^l_j is the bias
  of the j-th neuron in the l-th layer
- b^l bias vector for the l-th layer

Nielsen, 2015

Notation: *a, z*

- *Activation* of the *j*-the neuron in the *l*-th level:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

- Activation vector at the *l*-th layer:

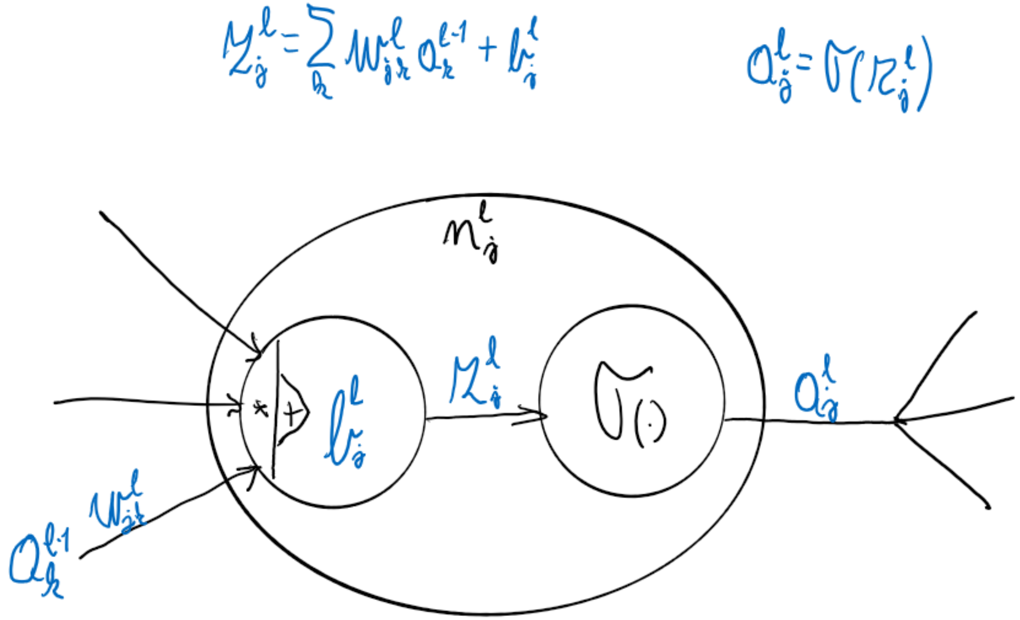
$$a^l = \sigma(w^l a^{l-1} + b^l) = \sigma(z^l)$$

- *Weighted input* to the *j*-the neuron in the *l*-th level:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

- Vector of weighted inputs at the *l*-th layer:

$$z^l \equiv w^l a^{l-1} + b^l$$

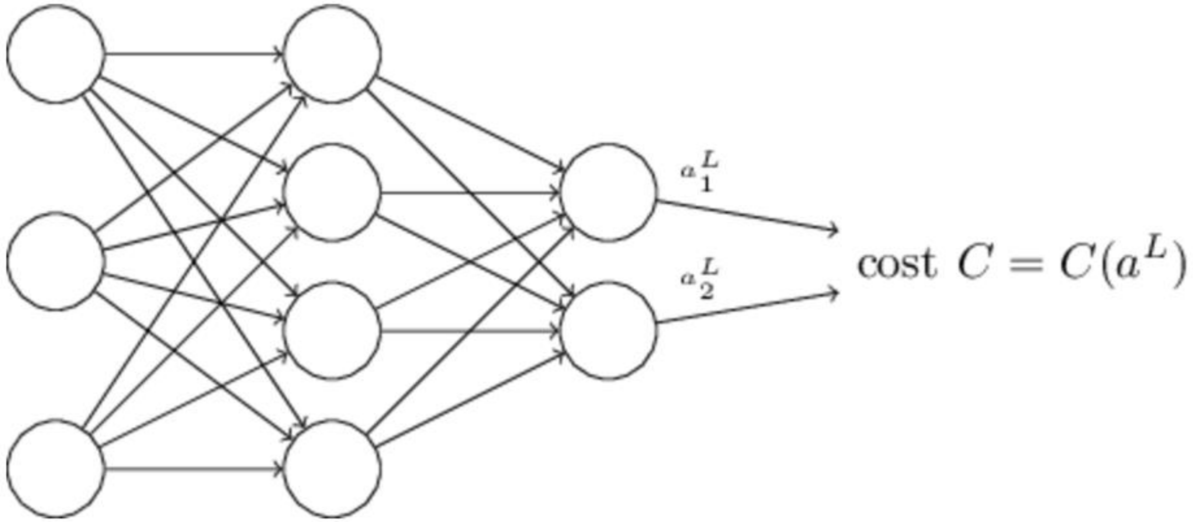


Assumptions about loss function

- Two assumptions about loss function:

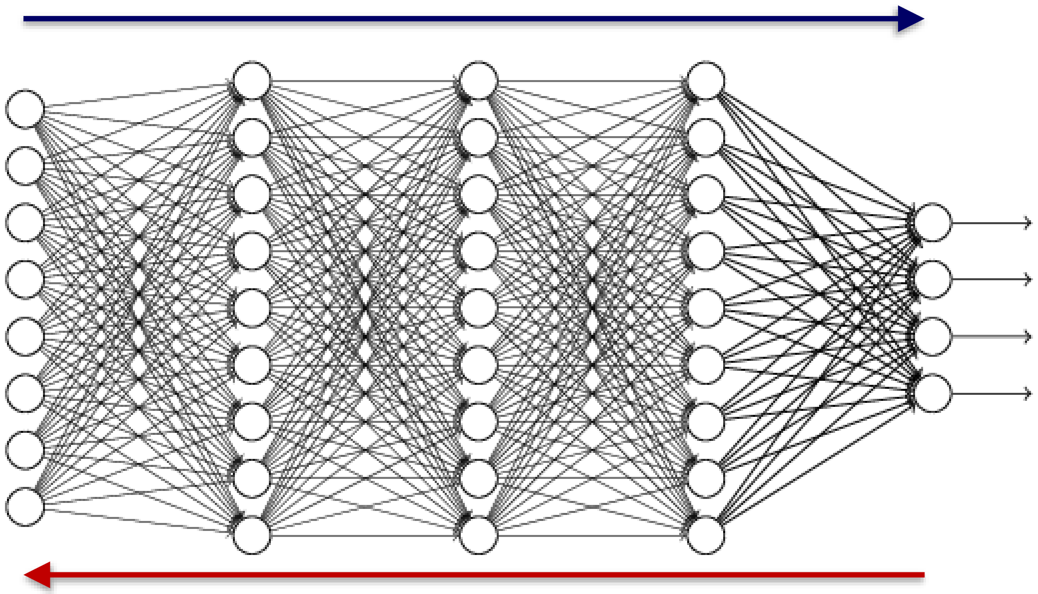
1. The loss function C can be written as an average over cost functions C_x for individual images x

2. The loss function C can be written as a function of the outputs from the neural network



Main principle

- We need the gradient of the Loss function ∇C $\frac{\partial C}{\partial b_j^l}$ $\frac{\partial C}{\partial w_{jk}^l}$
- Two phases:
  - Forward pass; propagation: the input sample is propagated through the network and the error at the final layer is obtained



  - Backward pass; weight update: the error is backpropagated to the individual levels, the contribution of the individual neuron to the error is calculated and the weights are updated accordingly

Chain rule

$$f\left(g(x)\right) \qquad \frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1}$$

$$a_j^l = \sigma(z_j^l) \qquad a_j^{l+1} = \sigma(z_j^{l+1})$$

$$\frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial a_k^{l+1}}\frac{\partial a_k^{l+1}}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l}$$

$$= \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l}$$

$$z_k^{l+1} = \sum_j w_{kj}^{l+1}\sigma(z_j^l) + b_k^{l+1}$$

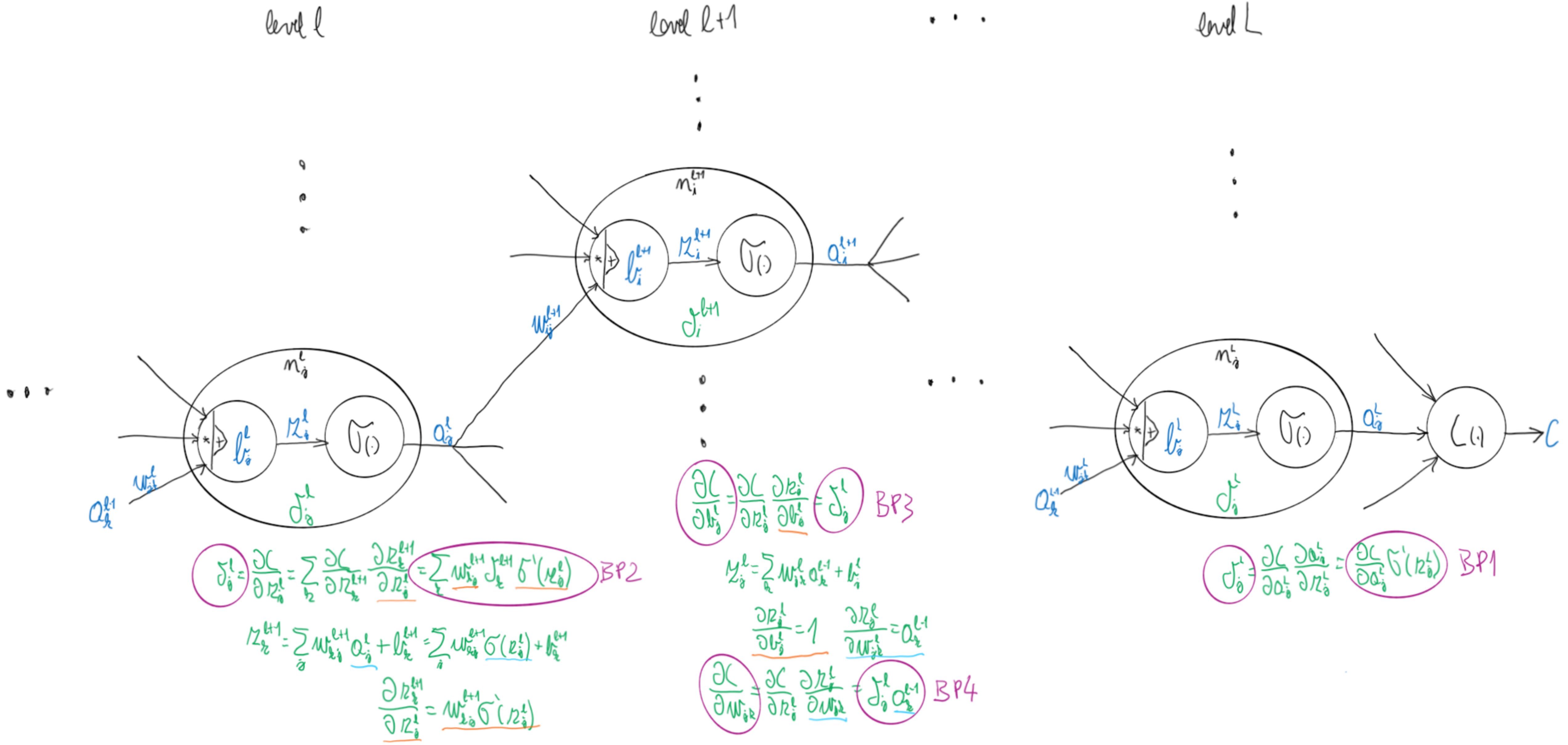
Learning strategy

- To obtain the gradient of the Loss function ∇C : $\frac{\partial C}{\partial b_j^l}$ $\frac{\partial C}{\partial w_{jk}^l}$
  - For every neuron in the network calculate the error of this neuron

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

    - This error propagates through the network causing the final error
    - Backpropagate the final error to get all δ_j^l
  - Obtain all $\frac{\partial C}{\partial b_j^l}$ and $\frac{\partial C}{\partial w_{jk}^l}$ from δ_j^l

Derivation of backpropagation



Equations of backpropagation

Nielsen, 2015

- BP1: Error in the output layer:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \qquad \delta^L = \nabla_a C \odot \sigma'(z^L)$$

- BP2: Error in terms of the error in the next layer:

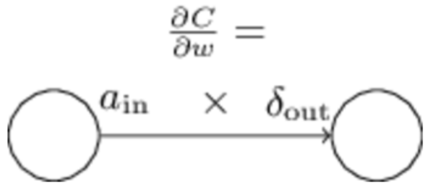
$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \qquad \delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

- BP3: Rate of change of the cost wrt. to any bias:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad \frac{\partial C}{\partial b} = \delta$$

- BP4: Rate of change of the cost wrt. to any weight:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \qquad \frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}}$$



Backpropagation algorithm

- **Input** x: Set the corresponding activation a^1 for the input layer

- **Feedforward**: For each $l = 2, 3, \ldots, L$

  compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$

- **Output error** δ^L: Compute the output error $\delta^L = \nabla_a C \odot \sigma'(z^L)$

- **Backpropagate the error**:

  For each $l = L-1, L-2, \ldots, 2$

  compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

- **Output the gradient**:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \frac{\partial C}{\partial b^l_j} = \delta^l_j$$

Backpropagation and SGD

For a number of **epochs**

Until all training images are used

Select a **mini-batch** of m training samples

For each training sample x in the mini-batch

Input: set the corresponding activation $a^{x,1}$

Feedforward: for each $l = 2, 3, \ldots, L$

compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$

Output error: compute $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$

Backpropagation: for each $l = L-1, L-2, \ldots, 2$

compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$

Gradient descend: for each $l = L, L-1, \ldots, 2$ and x update:

$$w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$$

$$b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$$

Example code: Backpropagation

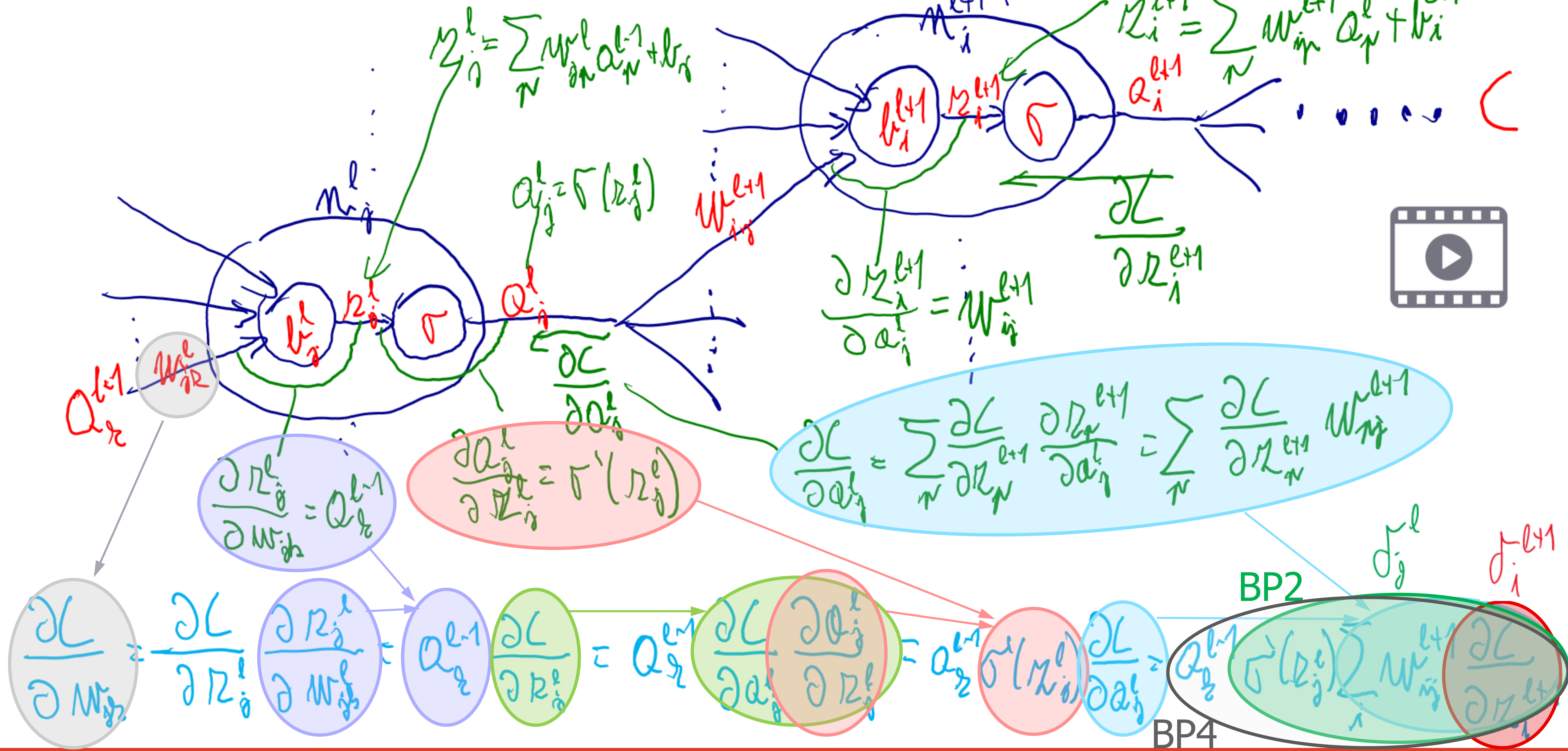
```
def backprop(self, x, y):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)
```

```
def cost_derivative(self, output_activations, y):
    return (output_activations-y)
```

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

```
def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```

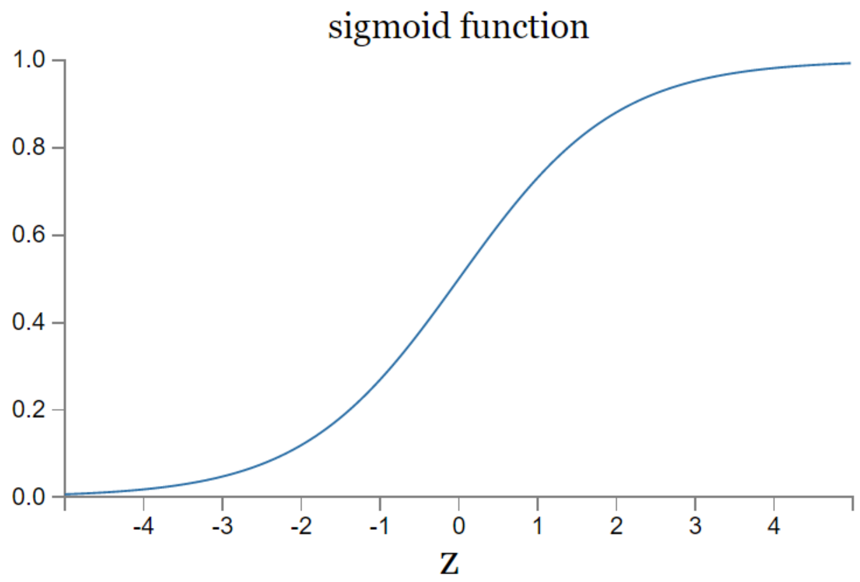
Backprop summary



Quadratic (L2) loss function

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

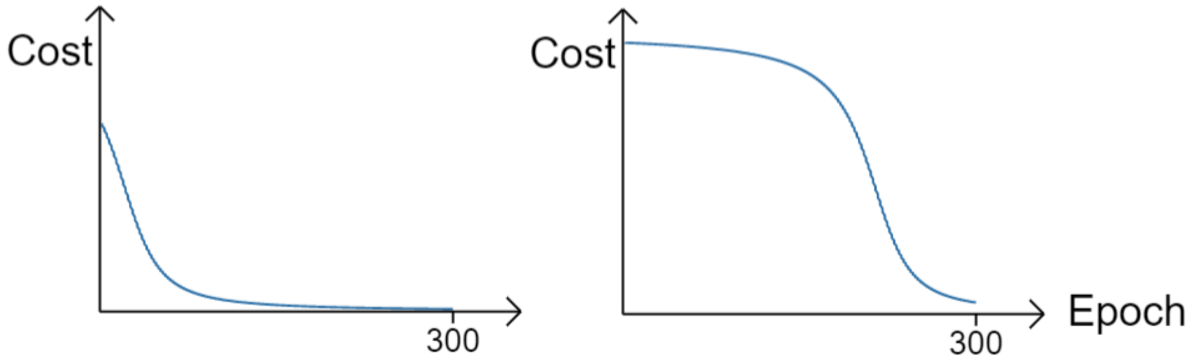
$$C = \frac{(y-a)^2}{2}$$



- Partial derivatives depend on σ'

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x$$

$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z)$$



- In case of **sigmoid** activation function and small or large activations -> slow learning!

Quadratic loss function

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

- In case of **linear** neurons in the output layer: $a_j^L = z_j^L$
- Partial derivatives:

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n} \sum_x a_k^{L-1} (a_j^L - y_j)$$

$$\frac{\partial C}{\partial b_j^L} = \frac{1}{n} \sum_x (a_j^L - y_j)$$

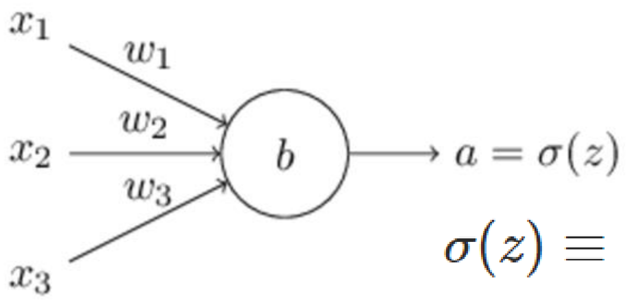
- Error in the output layer:

$$\delta^L = a^L - y$$

Cross-entropy loss function

- For one neuron with **sigmoid** activation function:

$$C = -\frac{1}{n}\sum_x \left[y \ln a + (1-y)\ln(1-a)\right]$$

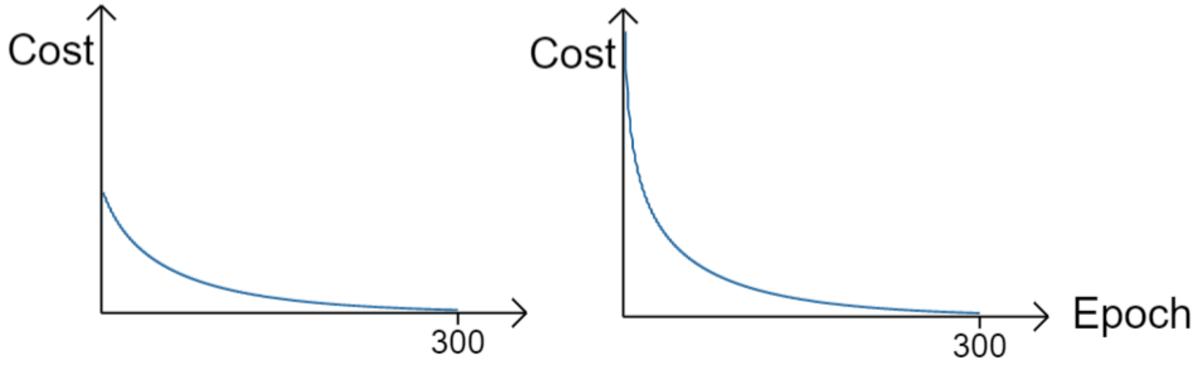


$$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$

- Partial derivatives do not depend on σ' any more!

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y)$$

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z) - y)$$



- Slow learning problem avoided

Cross-entropy loss function

$$C = -\frac{1}{n}\sum_{x}\left(y\ln(\sigma(z)) + (1-y)\ln(1-\sigma(z))\right)$$

$$\frac{\partial C}{\partial w_j} = \frac{\partial C}{\partial z}\frac{\partial z}{\partial w_j} = -\frac{1}{n}\sum_{x}\left(\frac{y}{\sigma(z)}\sigma'(z) - \frac{1-y}{1-\sigma(z)}\sigma'(z)\right)x_j =$$

$$= -\frac{1}{n}\sum_{x}\frac{y(1-\sigma(z)) - (1-y)\sigma(z)}{\sigma(z)(1-\sigma(z))}\sigma'(z)\,x_j =$$

$$\sigma(z)(1-\sigma(z)) \rightarrow \sigma'(z)$$

$$= -\frac{1}{n}\sum_{x}\frac{y - y\sigma(z) - \sigma(z) + y\sigma(z)}{\sigma'(z)}\sigma'(z)\,x_j =$$

$$= \frac{1}{n}\sum_{x}x_j(\sigma(z) - y)$$

$$\rightarrow \delta^L = a^L - y$$

Cross-entropy loss function

- For many neurons:

$$C = -\frac{1}{n}\sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right]$$

- Partial derivatives in the output layer:

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n}\sum_x a_k^{L-1}(a_j^L - y_j) \qquad \frac{\partial C}{\partial b_j^L} = \frac{1}{n}\sum_x (a_j^L - y_j)$$

- Error in the output layer:

$$\delta^L = a^L - y$$

- Categorical cross-entropy loss: $C = -\frac{1}{n}\sum_x \sum_j y_j \ln a_j^L$

Softmax layer

- The activation function is defined as:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \qquad\qquad z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$$

- The activations sum to 1:

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1$$

=> the activations could be considered as probabilities
the output layer can be considered as a probability distribution

- Properties of Softmax:
  - Monotonic function: increasing z_j^L increases a_j^L
  - Any output activation a_j^L depends on all the weighted inputs

Categorical Cross-entropy loss function

- Loss function for **Softmax** output layer:

$$C \equiv -\sum_j y_j \ln a_j^L$$

- Partial derivatives in the output layer:

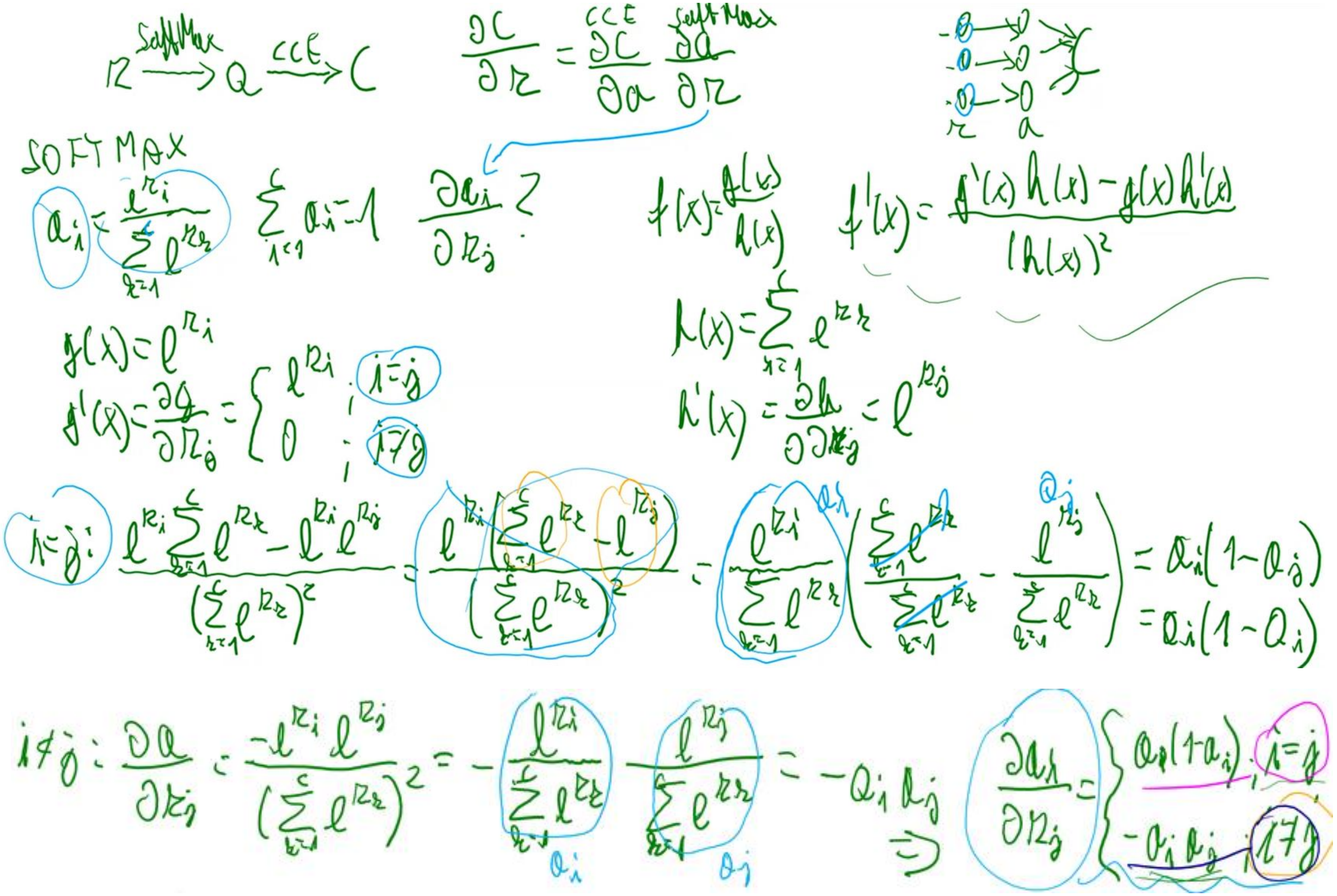
$$\frac{\partial C}{\partial b_j^L} = a_j^L - y_j$$

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}(a_j^L - y_j)$$

- Error in the output layer:

$$\delta_j^L = a_j^L - y_j$$

Categorical Cross-entropy loss function



Categorical Cross-entropy loss function

CCE:

$$C = -\sum_{k=1}^{C} y_k \ln a_k$$

$$\frac{\partial C}{\partial z_i} = -\sum_{k=1}^{C} y_k \frac{\partial(\ln a_k)}{\partial a_k}\frac{\partial a_k}{\partial z_i} = -\sum_{k=1}^{C} \frac{y_k}{a_k}\frac{\partial a_k}{\partial z_i} =$$

$$= -\frac{y_i}{a_i} a_i(1-a_i) + \sum_{\substack{k=1 \\ k\neq i}}^{C} \frac{y_k}{a_k} \cdot a_i a_k = -y_i + y_i a_i + \sum_{\substack{k=1 \\ k\neq i}}^{C} y_k a_i =$$

$$= a_i\left(y_i + \sum_{\substack{k=1 \\ k\neq i}}^{C} y_k\right) - y_i = a_i \underbrace{\sum_{k=1}^{C} y_k}_{=1} - y_i = a_i - y_i$$

$$\delta^L = a^L - y$$

Activation and loss functions

Activation function	Loss function
Linear $$a_j^L = z_j^L$$	Quadratic $$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$
Sigmoid $$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$	Cross-entropy $$C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a_j^L + (1-y_j) \ln(1-a_j^L) \right]$$
Softmax $$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$	Categorical Cross-entropy $$C = -\frac{1}{n} \sum_x \sum_j y_j \ln a_j^L$$

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}(a_j^L - y_j) \qquad \frac{\partial C}{\partial b_j^L} = a_j^L - y_j \qquad \delta_j^L = a_j^L - y_j$$

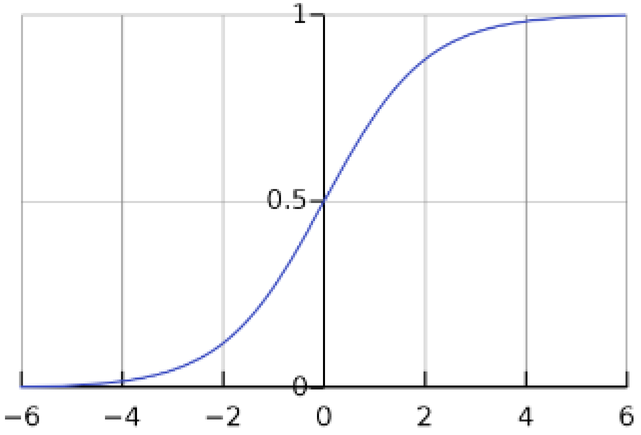
Activation functions

Method	Papers
ReLU	8096
Sigmoid Activation	5363
GELU Gaussian Error Linear Units (GELUs)	5285
Tanh Activation	4936
Leaky ReLU	915
GLU Language Modeling with Gated Convolutional Networks	372
Swish Searching for Activation Functions	254
Softplus	204
Mish	183
SELU Self-Normalizing Neural Networks	178
PReLU Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification	86
ReLU6 MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications	58
Hard Swish Searching for MobileNetV3	54
Maxout Maxout Networks	45
ELU Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)	34

[https://paperswithcode.com]

Sigmoid

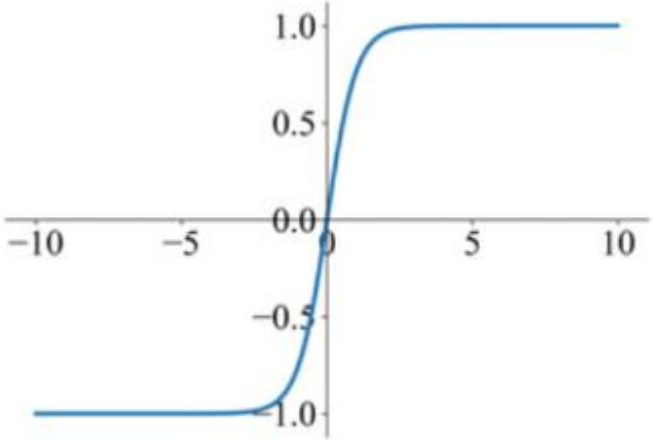
$$f(x) = \frac{1}{(1 + \exp(-x))}$$



- Continuous values from 0 to 1
- Saturated neurons slow down learning
- Not zero-centered
- Not very fast to compute

tanh

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

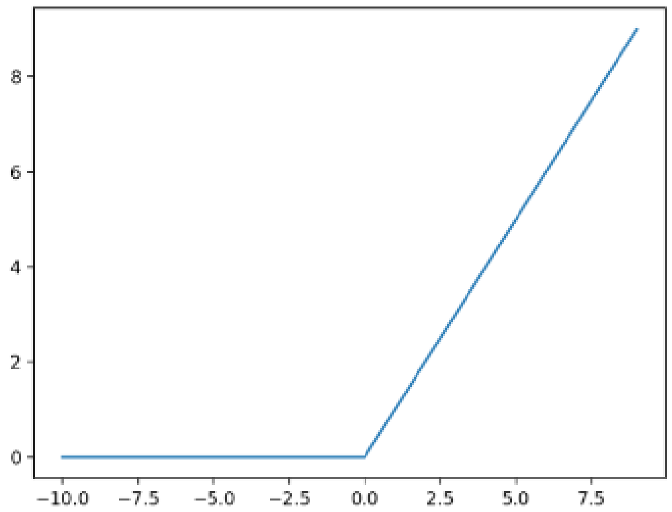


- Continuous values from -1 to 1
- Zero-centered
- Saturated neurons slow down learning
- Not very fast to compute

[LeCun et al., 1990]

ReLU	GELU

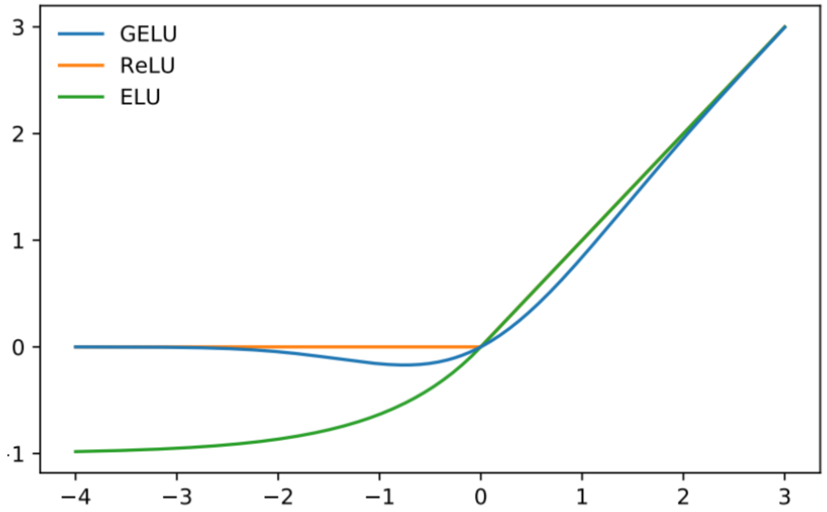
$$f(x) = \max(0, x)$$



- Rectified linear unit
- Do not saturate for x>0
- Computationally very efficient
- Faster convergence
- Dead neurons for x<0
- Not zero-centered

$$f(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}(x/\sqrt{2})\right]$$

$$X \sim \mathcal{N}(0,1)$$



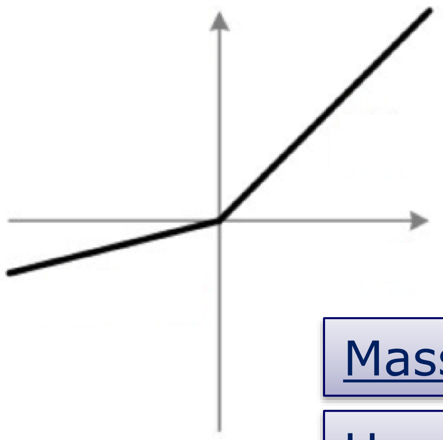
$$f(x) \approx x\sigma(1.702x)$$

- Gaussian Error Linear Unit
- Weights inputs by their percentile
- Smoother ReLU
- Less saturated neurons
- Not zero-centered
- Often use in Transfromers

Hendrycks, et al., 2016

Leaky ReLU

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases}$$



Mass et al., 2013

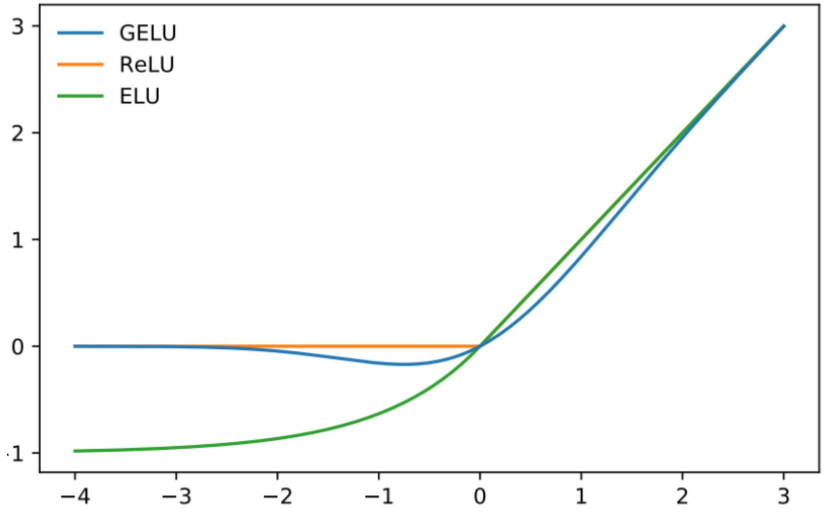
He et al., 2015

- ReLU with non-zero output for x<0
- Slope for x<0 controllable with α
  - It can be learned in PReLU
- Do not saturate
- More zero-centered
- Very fast to compute

ELU

$$f(x) = x \text{ if } x > 0$$

$$\alpha(\exp(x) - 1) \text{ if } x \leq 0$$

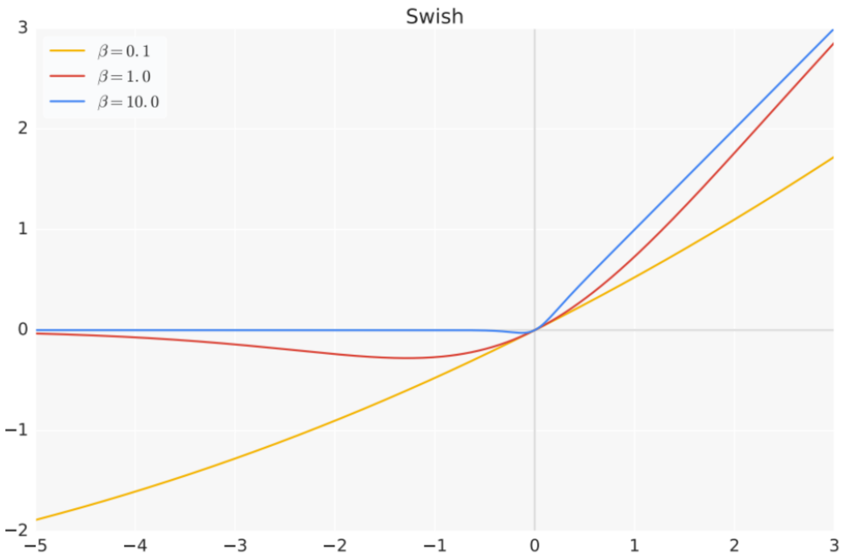


- Exponential Linear Unit
- More zero-centered
- Less saturated neurons
- Not very fast to compute

Clevert et al., 2015

SWISH

$$f(x) = x \cdot \text{sigmoid}(\beta x)$$

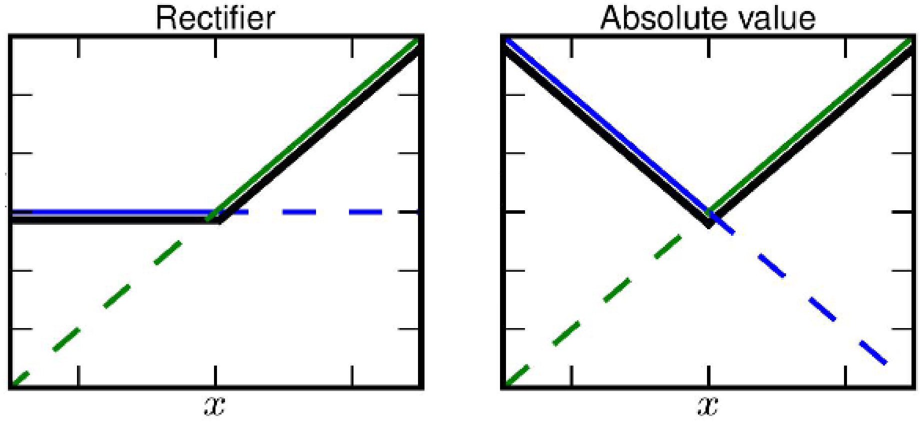


- Learnable parameter β
- Can be fixed to 1 → $f(x) = x\sigma(x)$
  - Sigmoid Linear Unit – SiLU
- More zero-centered
- Less saturated neurons

Ramachandran et al., 2017 | Elfwing et al., 2017

Maxout

$$f(x) = \max(w_1^T x + b_1, w_2^T x + b_2)$$



- Piecewise linear function
- Neurons do not saturate
- Two sets of parameters
- Computationally expensive

Goodfellow et al., 2013

Activation functions recap

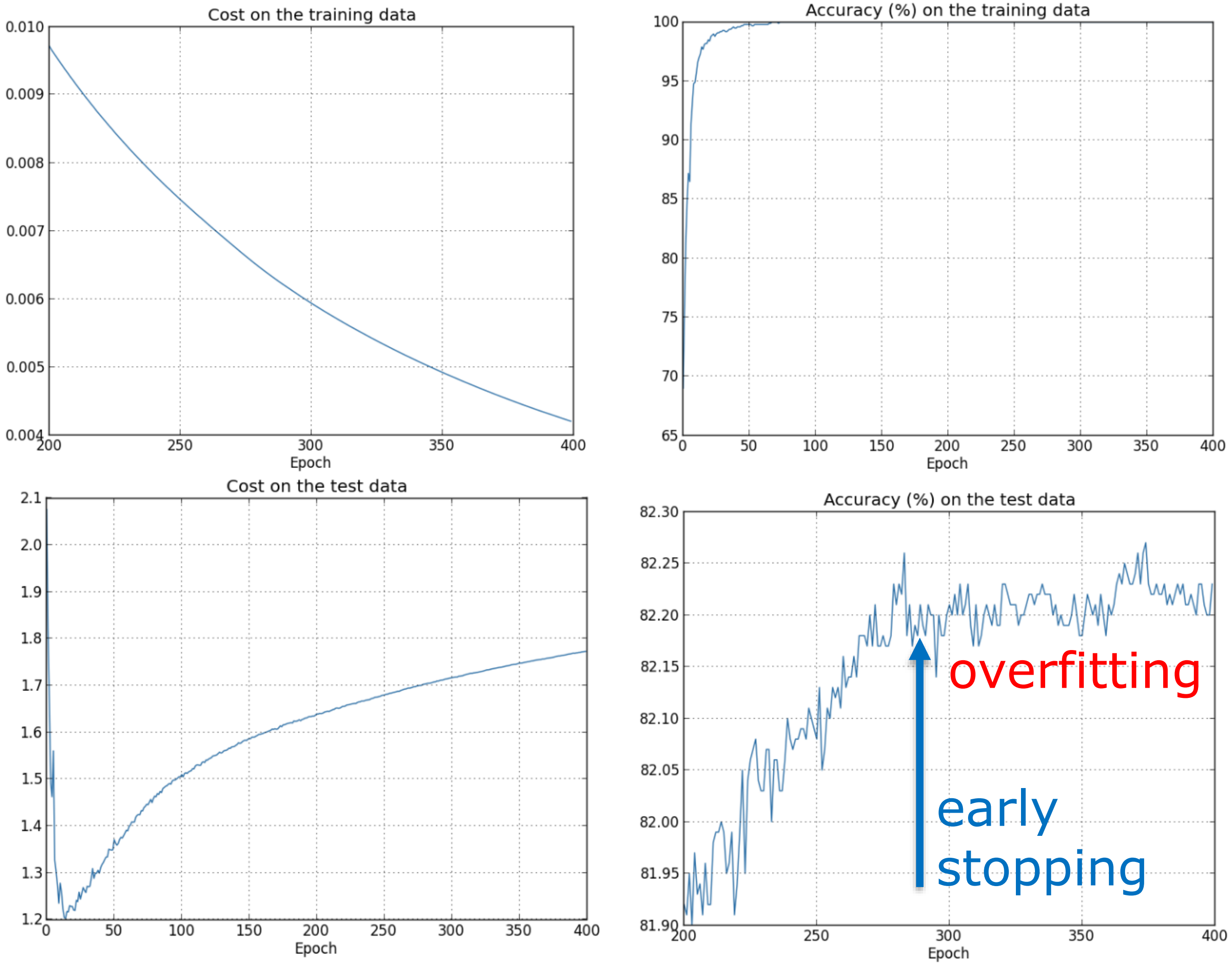
- ReLU usually suffices – the first choice
- Do not use sigmoid and tanh in hidden layers, use ReLU instead

- Select the activation function in the hidden layers according to the type of the neural network:
  - ReLU for CNNs (or Leaky ReLU, or ELU, etc.)
  - Sigmoid or tanh for RNNs
  - GELU for Transformers

- Select the activation function in the output layer according to the loss function:
  - Linear for L2 loss (regression)
  - Sigmoid for Cross-entropy (binary classification, multilabel classification)
  - Softmax for Categorical cross-entropy (multiclass classification)

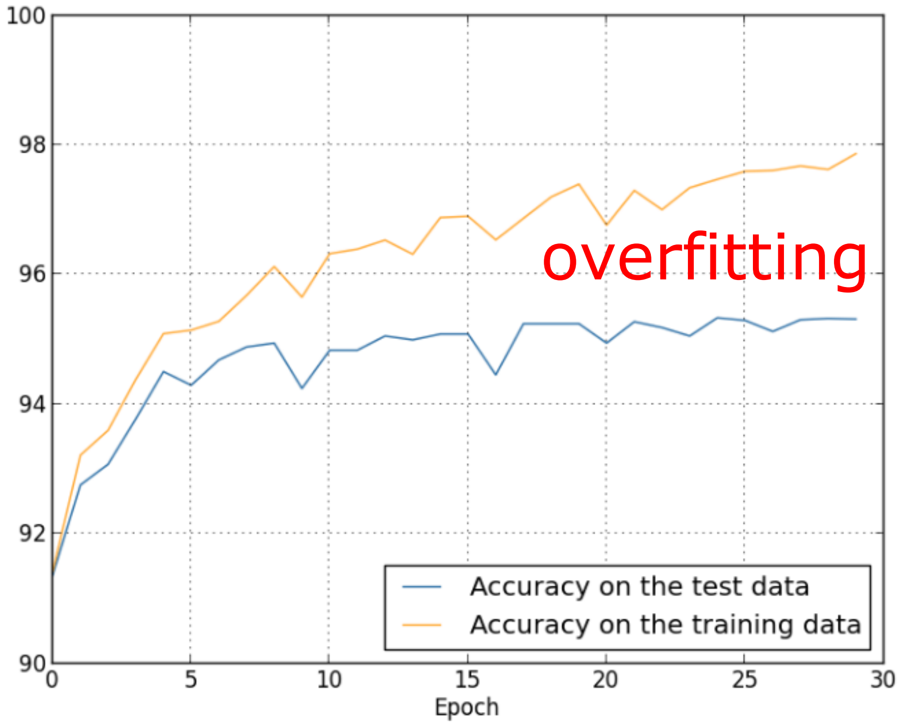
- Experiment for the best choice

Overfitting



1,000 MNIST training images

- Huge number of parameters -> danger of overfitting
- Use validation set to determine overfitting and early stopping
  - Hold out method



50,000 MNIST training images

Regularization

- How to avoid overfitting:
  - Increase the number of training images ☹
  - Decrease the number of parameters ☹
  - Regularization ☺

- Regularization:
  - L2 regularization
  - L1 regularization
  - Dropout
  - Data augmentation

L2 regularisation

- Add the regularisation term in the loss function
  - L_2 norm

Regularisation parameter

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1 - y_j)\ln(1 - a_j^L)\right] + \frac{\lambda}{2n}\sum_w w^2$$

Regularisation term

$$C = \frac{1}{2n}\sum_x \|y - a^L\|^2 + \frac{\lambda}{2n}\sum_w w^2$$

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

Weight decay

- Loss function:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

- Partial derivatives:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \qquad \frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$$

- Update rules:

$$w \rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w = \underbrace{\left(1 - \frac{\eta \lambda}{n}\right)}_{\text{Weight decay}} w - \eta \frac{\partial C_0}{\partial w} \qquad b \rightarrow b - \eta \frac{\partial C_0}{\partial b}$$

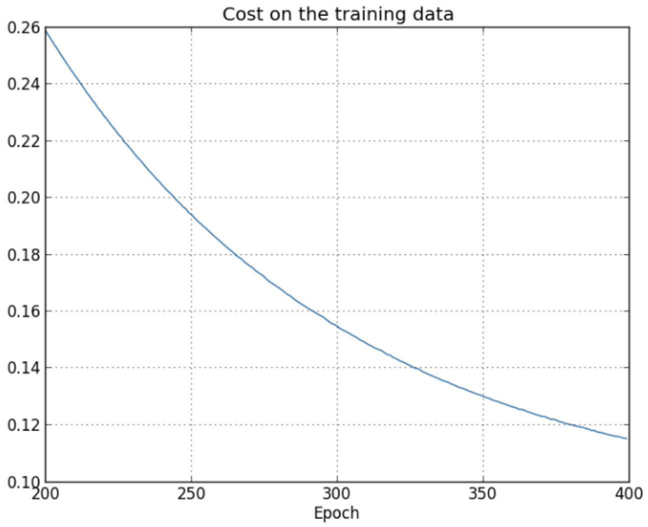
Weight decay

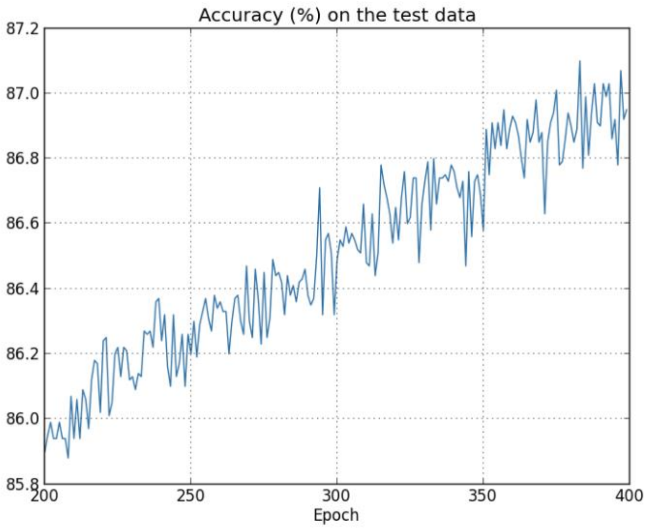
Regularised SGD

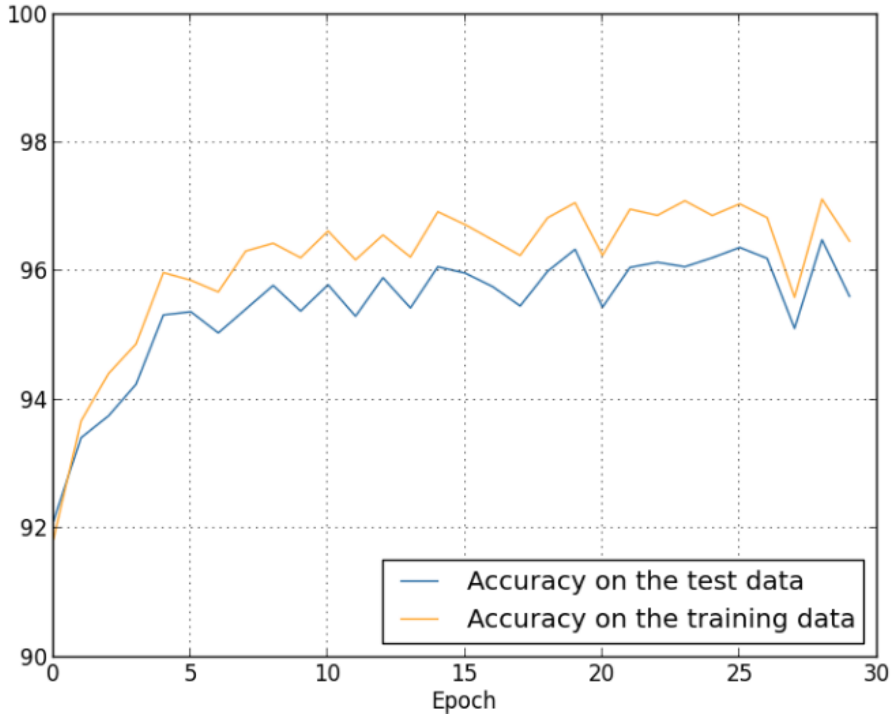
- Regularized learning rules for SGD:

$$w \rightarrow \left(1 - \frac{\eta\lambda}{n}\right) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w} \qquad b \rightarrow b - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial b}$$

- Improved performance!
  - Overfitting decreased







L1 regularization

- L_1 regularization term

$$C = C_0 + \frac{\lambda}{n}\sum_w |w|$$

- Partial derivatives:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}\operatorname{sgn}(w)$$

- Update rule:

$$w \rightarrow w' = w - \underbrace{\frac{\eta\lambda}{n}\operatorname{sgn}(w)}_{\text{Shrinking term}} - \eta\frac{\partial C_0}{\partial w}$$

Shrinking term

- Concentrate on relatively small number of high-importance connections

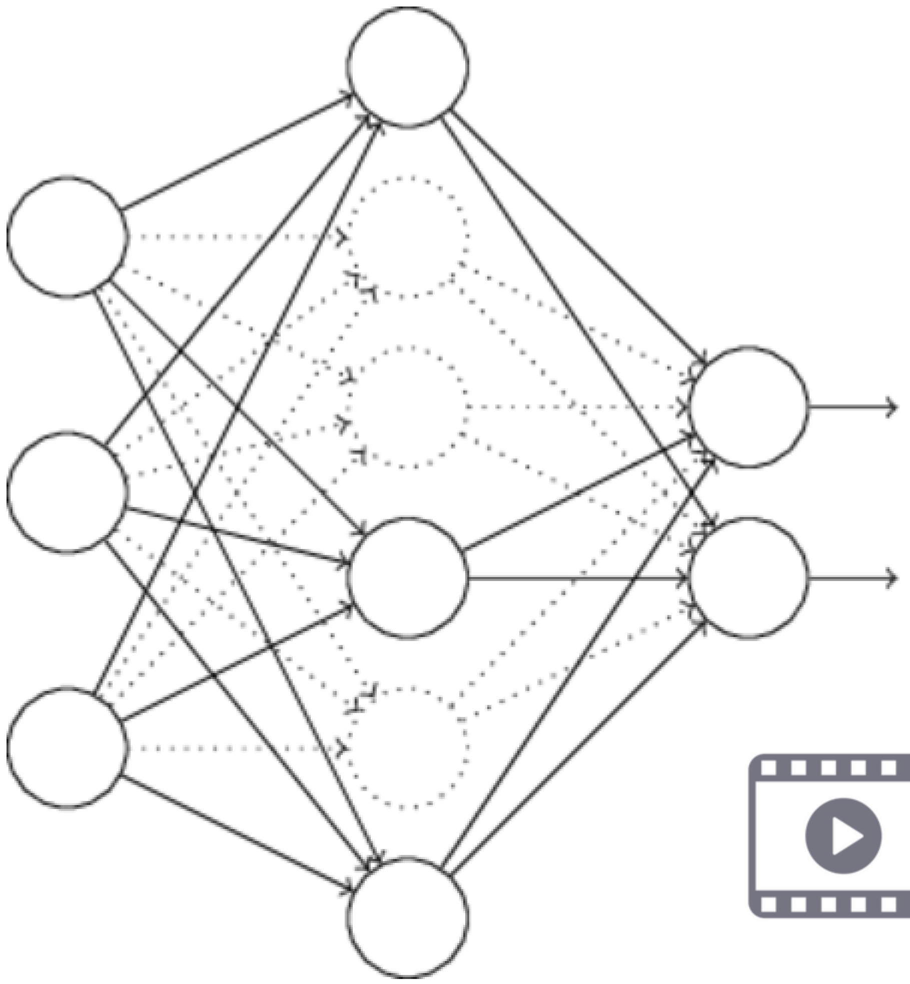
Dropout

- Randomly (and temporarily) delete half (or *p*) hidden neurons in the network
- Then restore the neurons and repeat the process

- Halve the weights when running the full network in test time
- Or double the weights during learning

- Ensemble learning: training multiple networks and averaging the results
- Reduces complex co-adaptations of neurons
- Smaller models harder to overfit

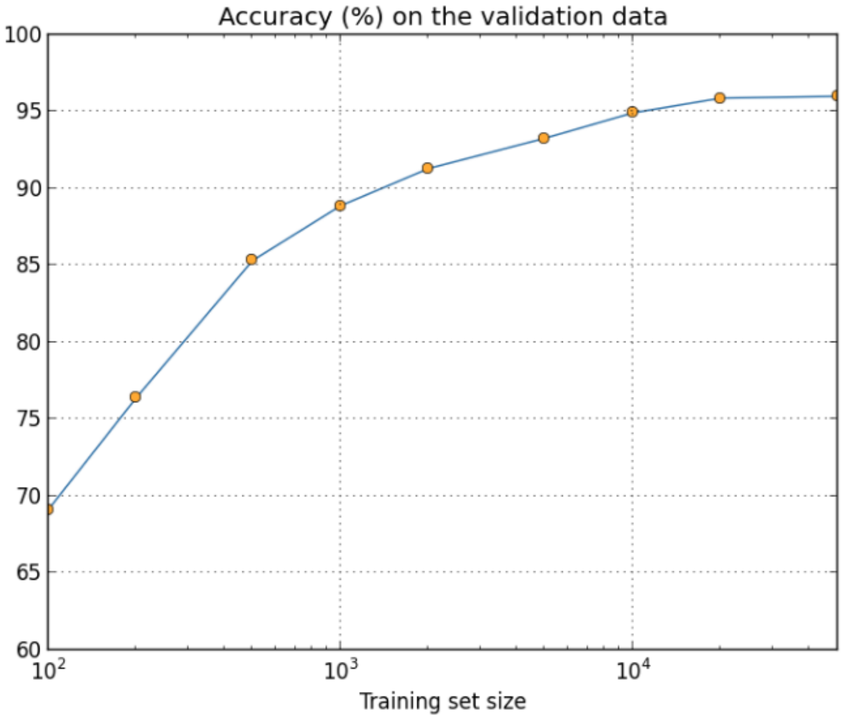
- Usually significantly improves the results



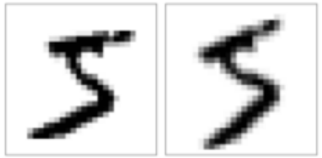
Srivastava et al., 2014

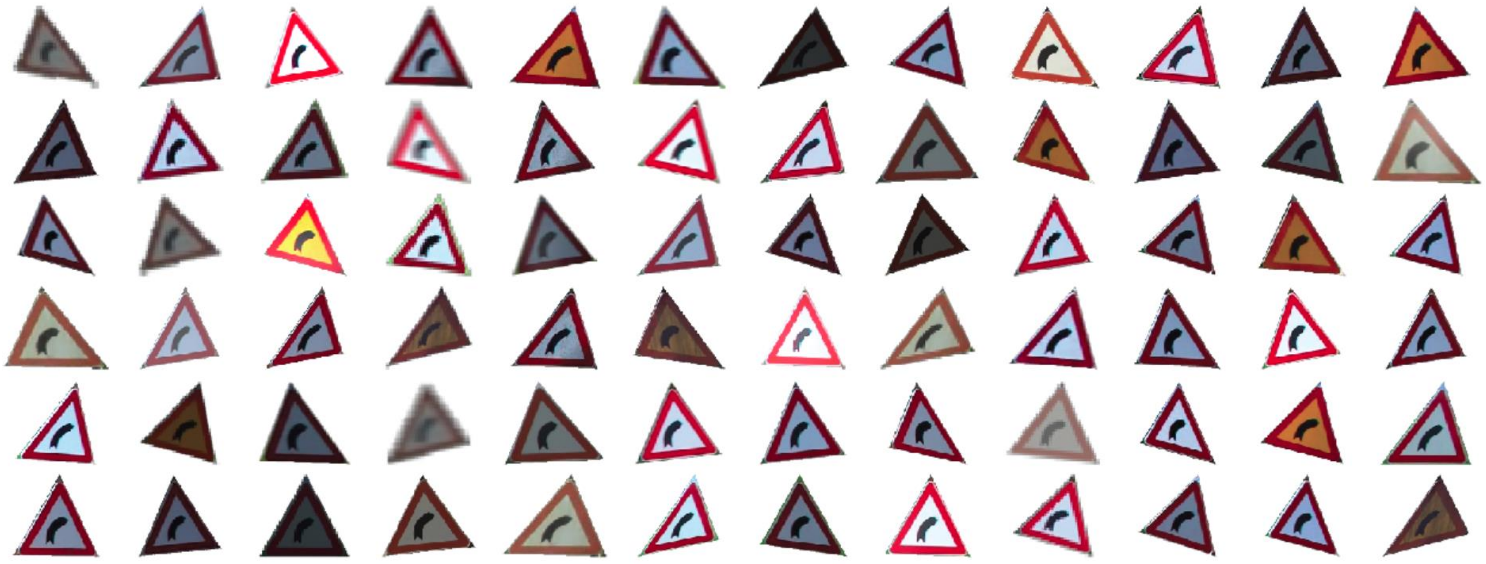
Data augmentation

- Use more data!



- Synthetically generate new data
- Apply different kinds of transformations: translations, rotations, elastic distortions, appearance modifications (intensity, blur)
- Operations should reflect real-world variation





Data preprocessing

- Curate the dataset
  - Identify/deal with missing values
  - Identify/deal with outliers
  - Data cleaning
  - Data engineering
  - Trash in – trash out

- Data reduction
  - Data selection
  - Dimensionality reduction
- Data normalisation
  - Data scaling
  - Mean-centering
  - Transforming to unit variance

- Same on train and test data!

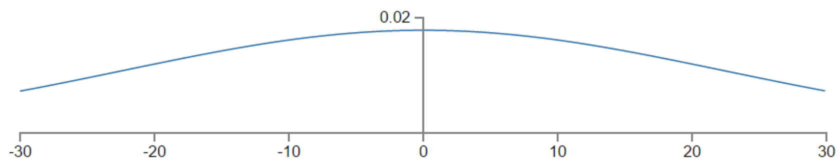
Tlamelo et al., 2021

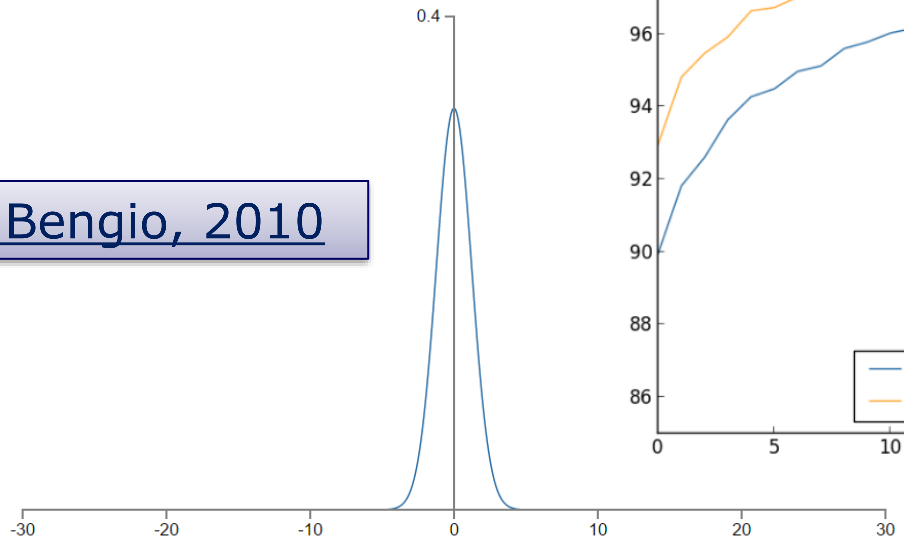
Weight initialization

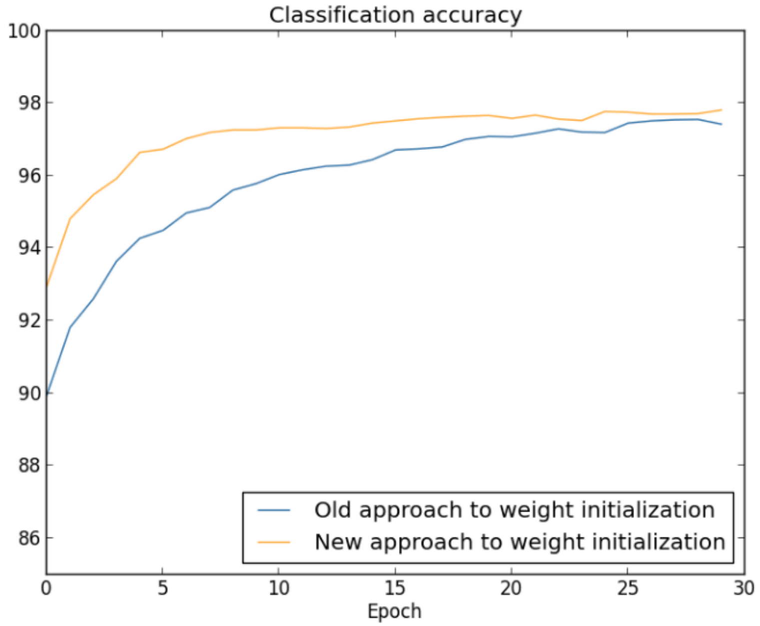
- Ad-hoc normalization
  - Initialize weights with $N(0,1)$
  - Variance is growing with n_{in}
  - Many large z
    => many saturated neurons
  - Slow learning
- Better initialization
  - Normalize variance with $1/\sqrt{n_{\text{in}}}$
  - Initialize weights with $N(0,1/n_{in})$
  - Total variance is limited
  - Faster learning!
- In case of ReLU:
  - ReLU halves the variance
  - Init with $N(\,0,\,1/(n_{in}/2)\,)$

Glorot & Bengio, 2010

He et al., 2015



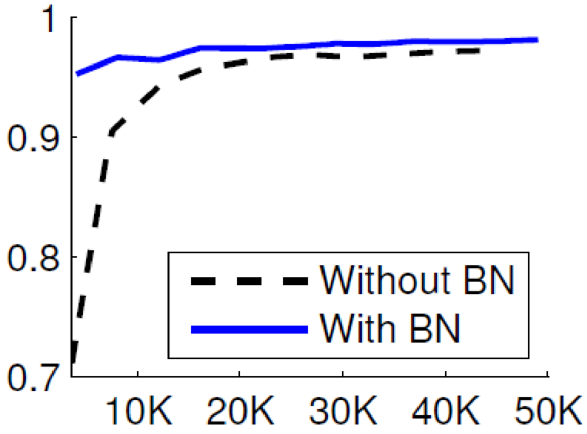




Batch normalisation

Ioffe and Szegedy, 2015

- Reducing internal covariate shift
- Normalising (whitening) layer inputs for each training mini-batch
  - Normalising with per-dimension mean and variance
- Speeds up learning
- Improves the gradient flow
- Regularisation
- Allows
  - Using higher learning rates
  - Less careful initialisation
  - Less dropout



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Parameter-update optimizers

- Different schemes for updating the weights
  - Gradient descend
  - Momentum update
  - AdaGrad update
  - RMSProp update
  - Adam update

  - Learning rate decay

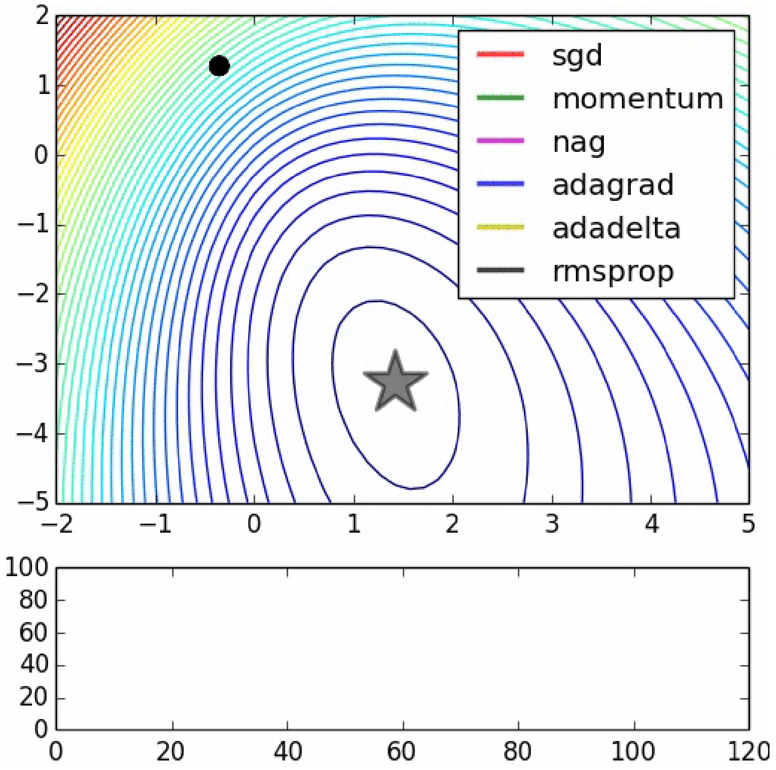


Image credit: Alec Radford

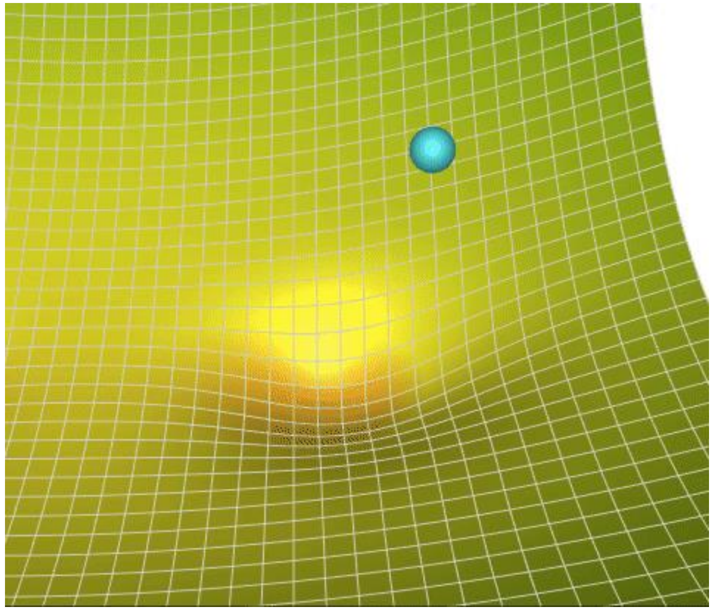
Gradient descend

Algorithm:

- Initialize v
- Until stopping criterium riched
  - Apply udate rule $v \to v' = v - \eta \nabla C.$

$$\Delta = -\eta \nabla$$

$$\Theta = \Theta + \Delta$$



Video credit to Lili Jiang:
https://github.com/lilipads/gradient_descent_viz

- Vanilla gradient descend can be very inefficient
- Take into account different slopes in different dimensions

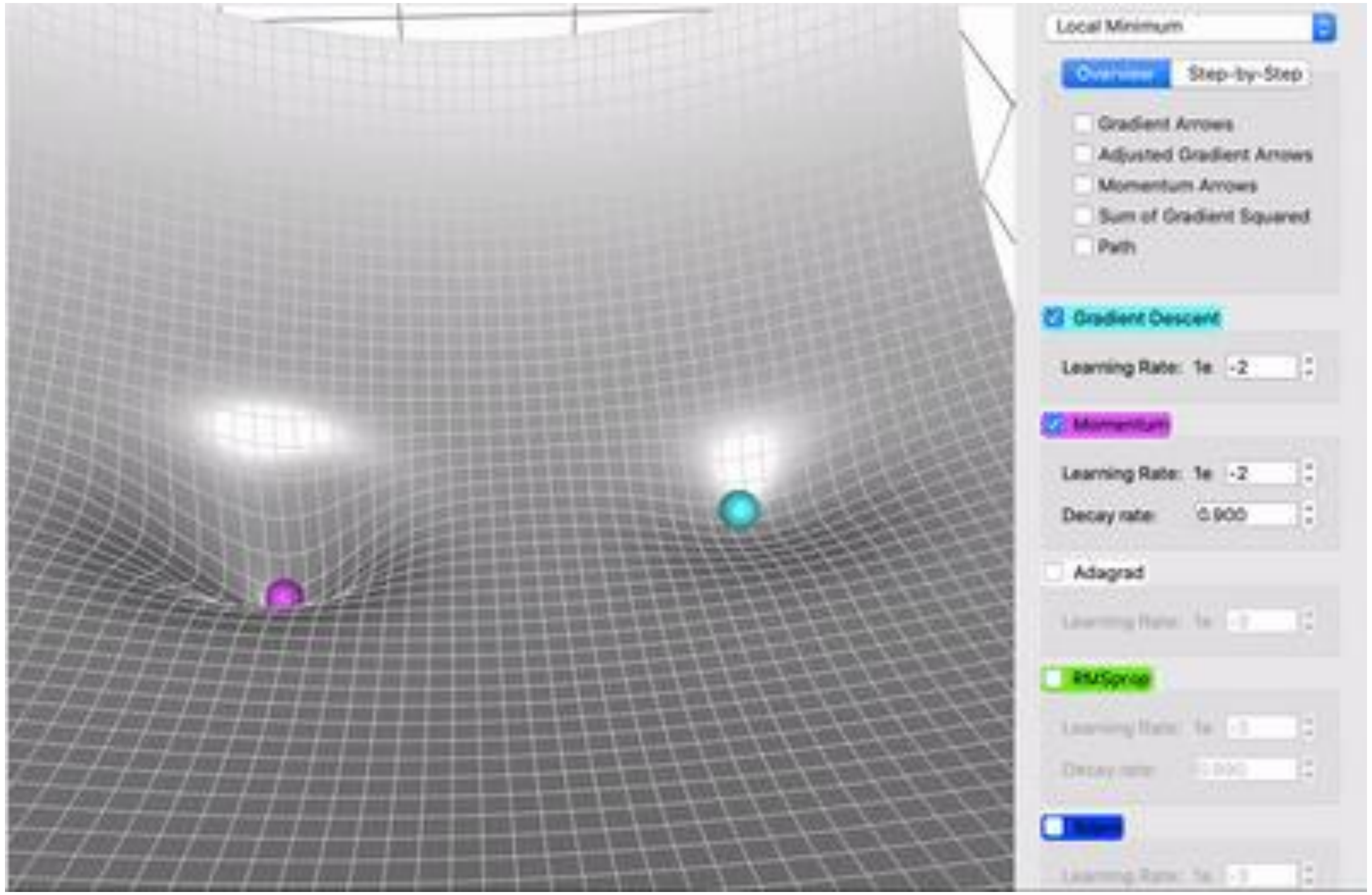
Momentum update

- Accumulate speed in the individual dimensions

$$\Delta = -\eta\nabla + \beta\Delta$$

$$\Sigma_\nabla = \beta\Sigma_\nabla + \nabla$$
$$\Delta = -\eta\Sigma_\nabla$$

- Cancels the oscillation in steep dimensions
- Builds up speed in shallow dimensions
- Faster convergence
- It may avoid local minima



Video credit to Lili Jiang: https://github.com/lilipads/gradient_descent_viz

AdaGrad and RMSProp updates

- Different learning rates for different dimensions
  - Scaling gradient in the individual dimensions
- Normalising the changes with the accumulated magnitudes of changes in the individual dimensions

- AdaGrad:

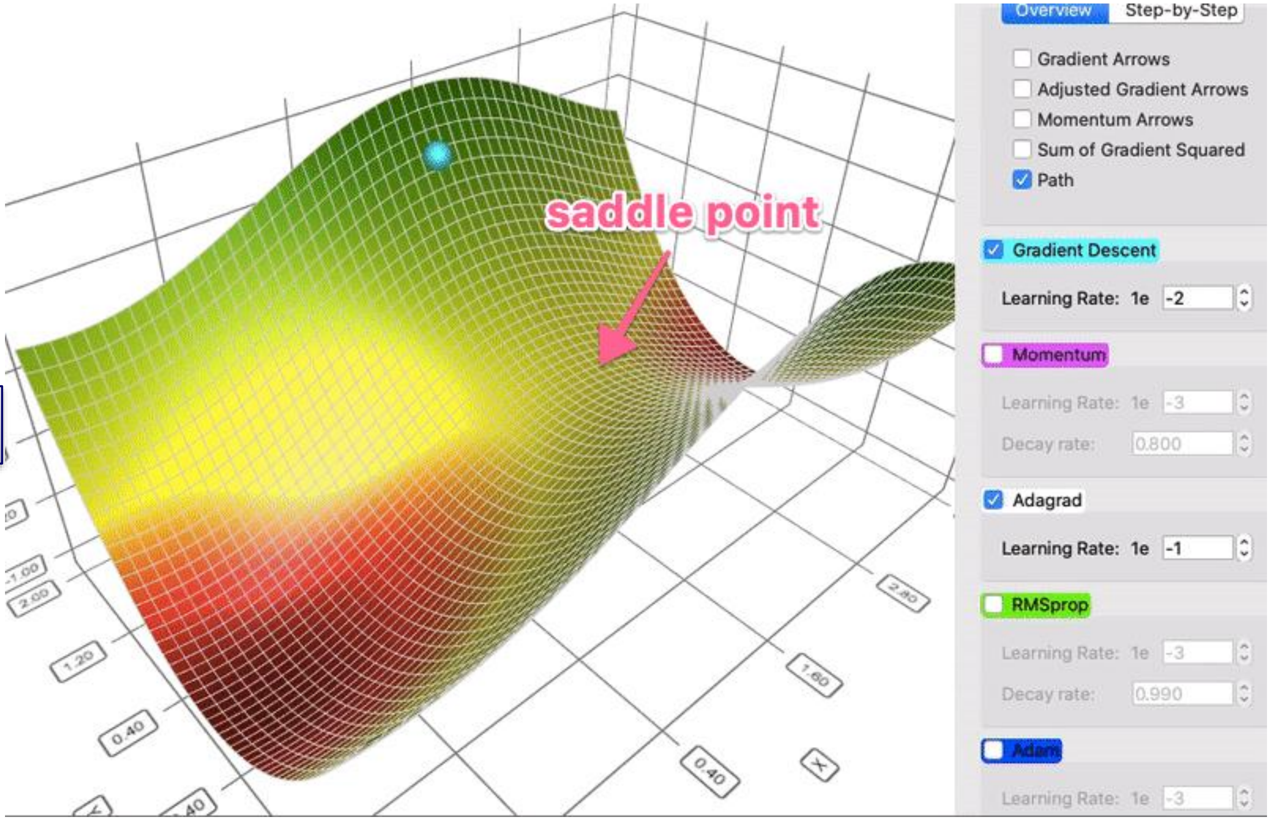
$$\Sigma^2_\nabla = \Sigma^2_\nabla + \nabla^2$$
$$\Delta = \frac{-\eta\nabla}{\sqrt{\Sigma^2_\nabla}}$$

Duchi et al., 2011

- RMSProp:

$$\Sigma^2_\nabla = \beta\Sigma^2_\nabla + (1-\beta)\nabla^2$$
$$\Delta = \frac{-\eta\nabla}{\sqrt{\Sigma^2_\nabla}}$$

Tieleman and Hinton, 2011



Video credit to Lili Jiang:
https://github.com/lilipads/gradient_descent_viz

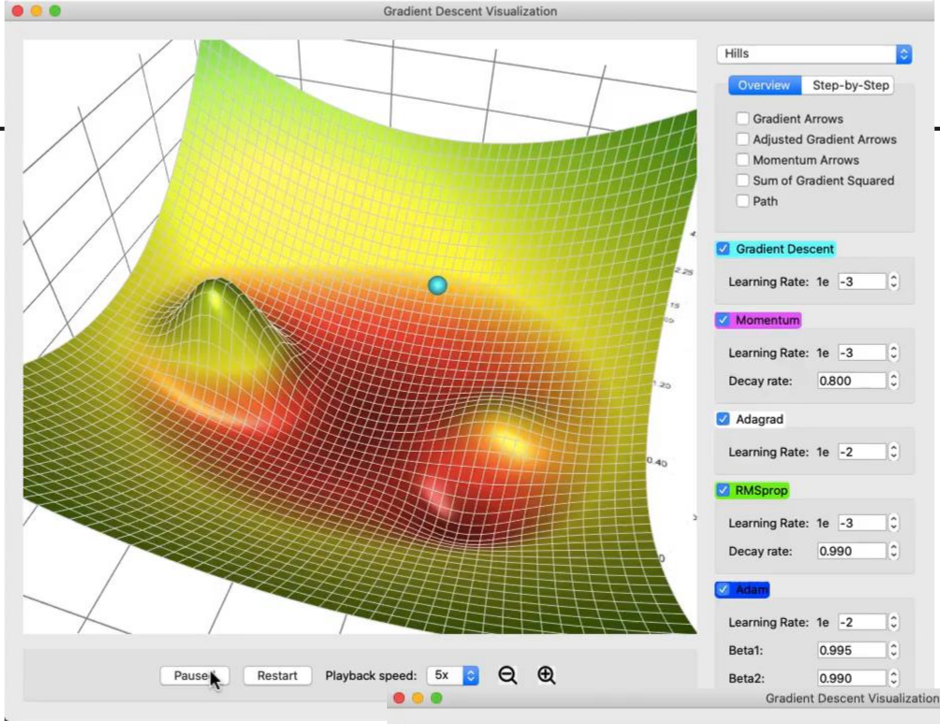
Adam update

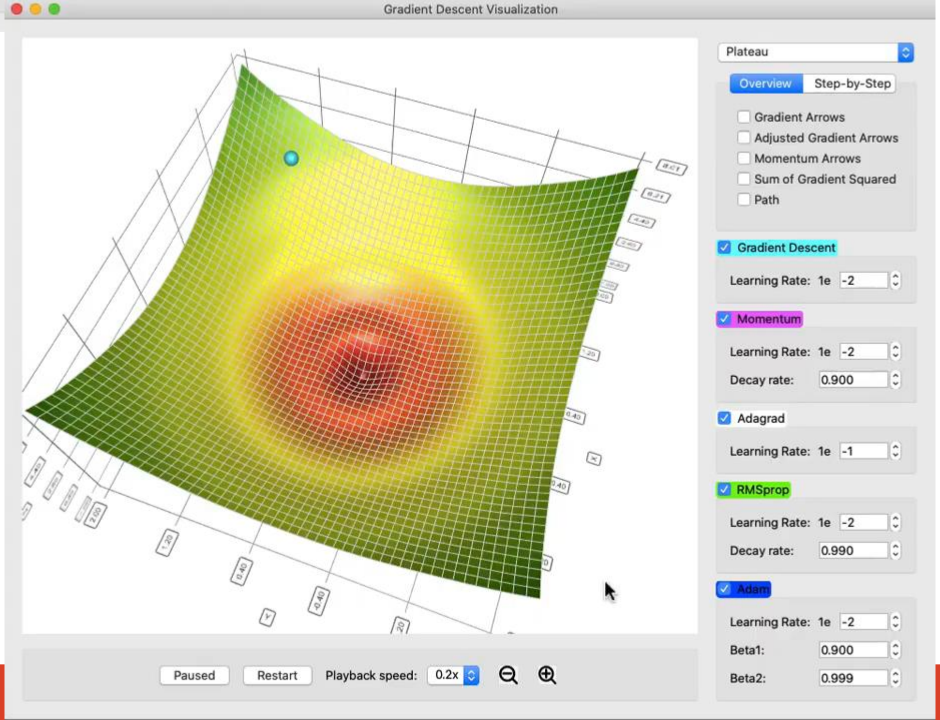
- Considering both ideas:
  - Keeping momentum
  - Adaptive learning rate

- ADAptive Moment estimation

$$\Sigma_{\nabla} = \beta_1 \Sigma_{\nabla} + (1 - \beta_1)\nabla$$
$$\Sigma^2_{\nabla} = \beta_2 \Sigma^2_{\nabla} + (1 - \beta_2)\nabla^2$$
$$\Delta = \frac{-\eta \Sigma_{\nabla}}{\sqrt{\Sigma^2_{\nabla}}}$$

- Usually works fine
  - The default choice





Video credit to Lili Jiang: https://github.com/lilipads/gradient_descent_viz

Kingma and Ba, 2014

Parameter-update optimizers

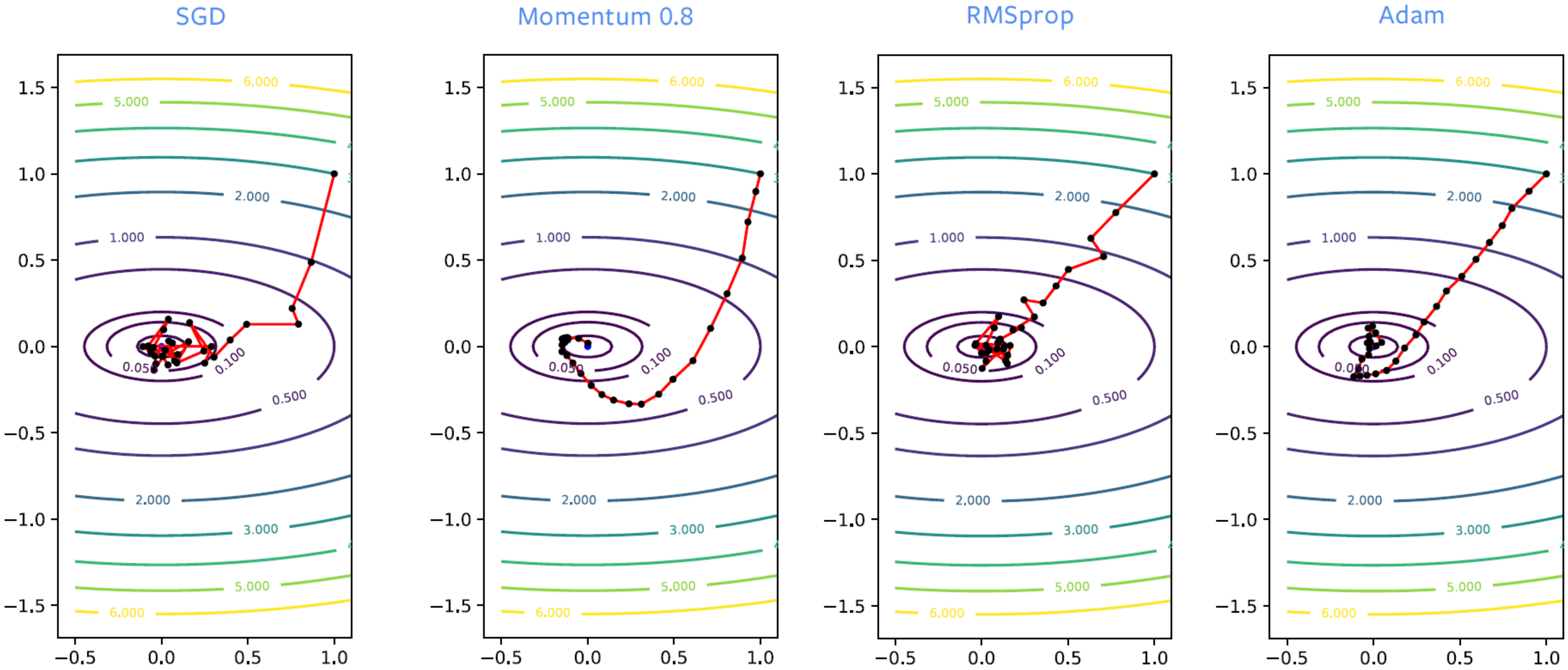
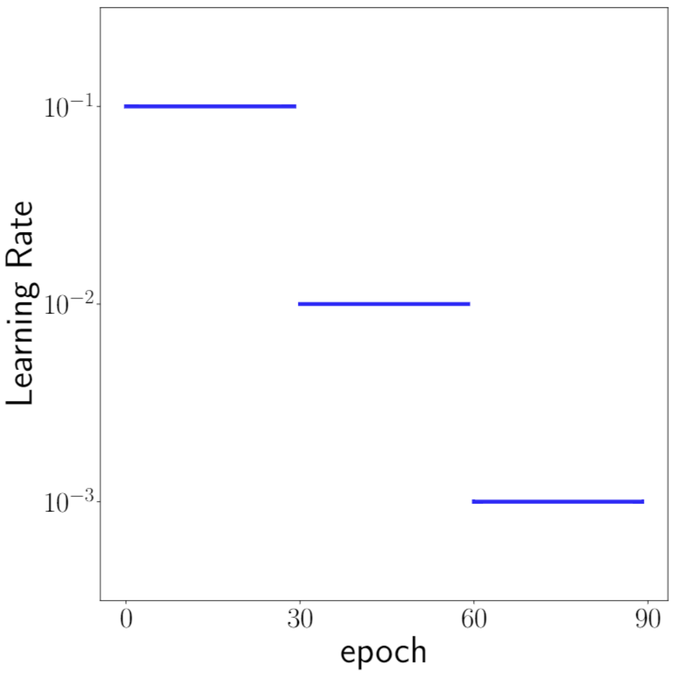


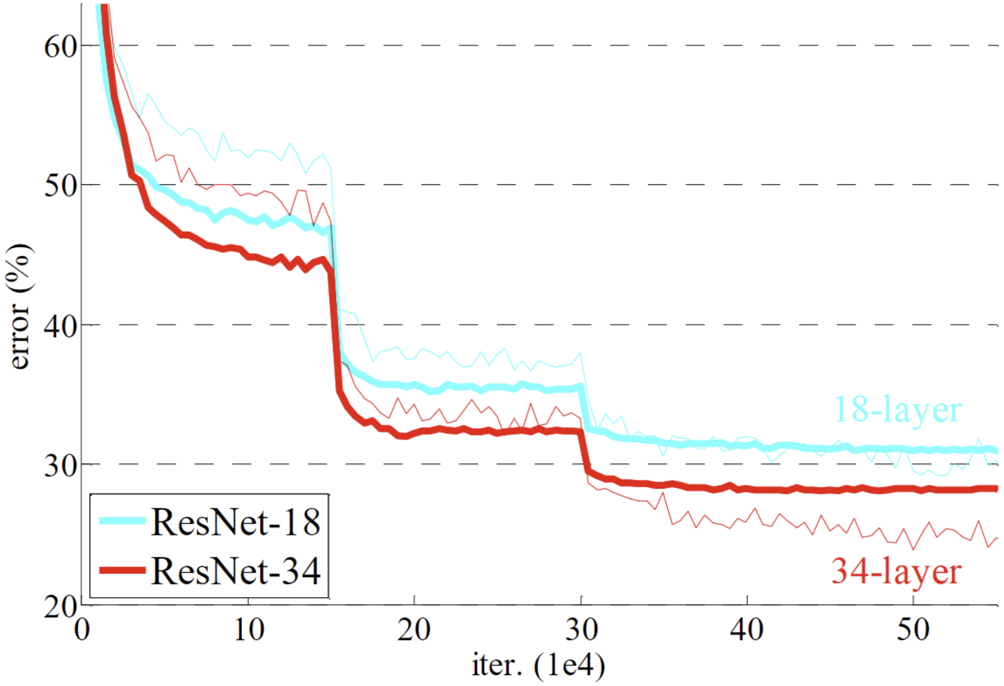
Image credit: Y. LeCun

Learning rate decay

- Start with a large learning rate
  - Escape spurious local minima
  - Suppresses the network from memorizing noisy data
- and decay it multiple times
  - Refine the solution and avoid oscillation
  - Improves the learning of complex patterns

- Learning rate schedule:
  - Step decay
  - Linear decay
  - Exponential decay
  - Inverse
  - Inverse sqrt

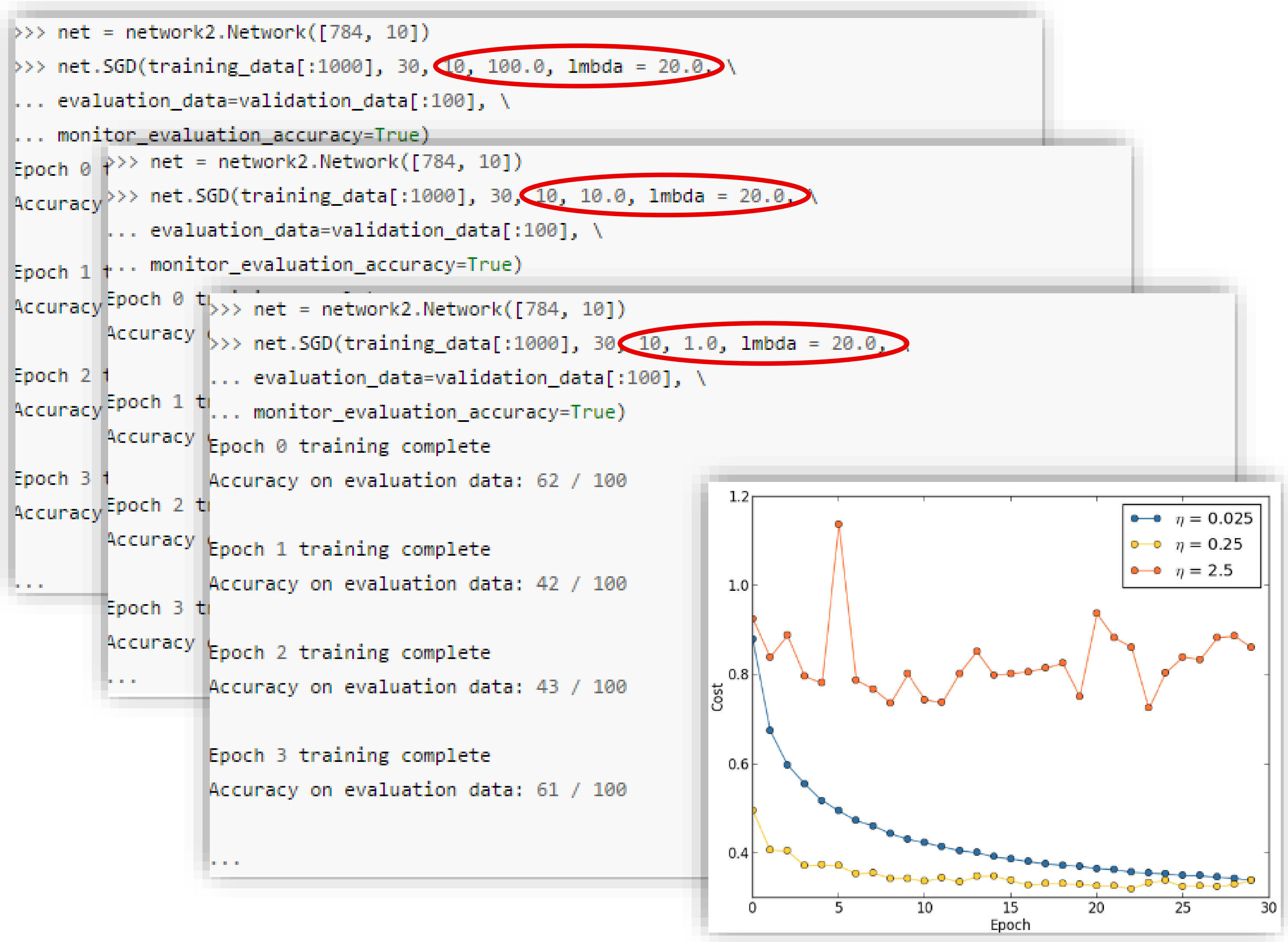




You et.al, 2019

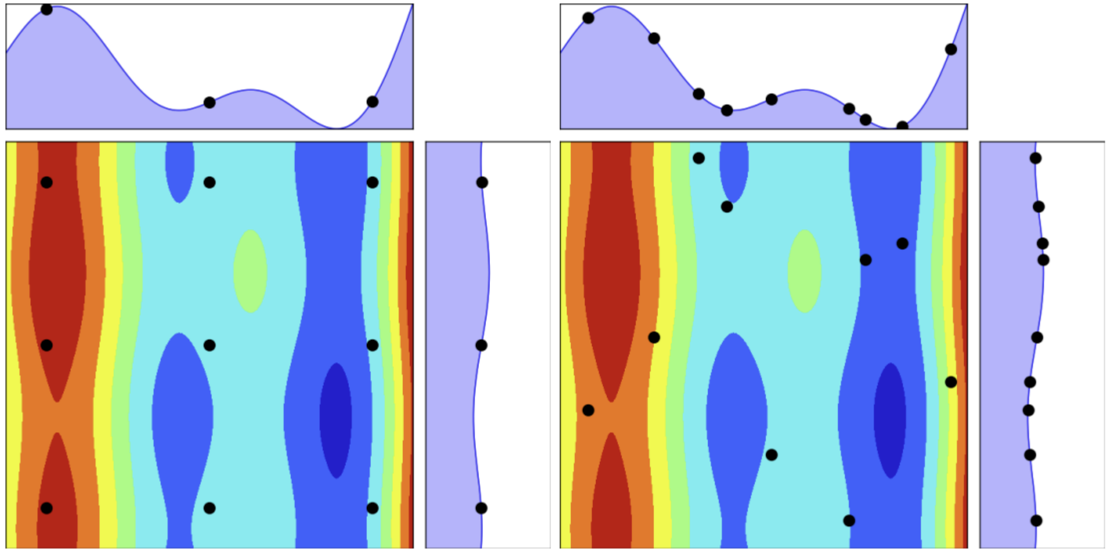
Setting up the network

- Set up the network
- Get any non-trivial learning
  - Even on a smaller problem to speed up the process
  - Can overfit to training data
  - Then scale up the data
- Monitor progress
- Set up reasonable η
  - You may define learning rate schedule
- Define regularization param.
  - Start with $\lambda=0$, increase it
- Use early stopping
  - To decrease number of epochs
- Cross-validate
- Automate the process of determining parameters



Hyperparameter optimisation

- Cross-validation of multiple parameters
- Coarse to fine cross-validation
  - First for a few epochs, coarse search
  - Then for more epochs, finer search
- Automated parameter sampling
  - Grid search
  - Random sampling of parameters
    - Sample in log space
- Run multiple validations simultaneously
- Actively observe the learning progress
  - visualise the loss curve, observe the results
- Hyperparameters to optimize:
  - Network architecture (architecture, number of layers, kernel sizes, loss function, etc.)
  - Learning rate, decay schedule, optimiser
  - Regularisation parameters (L2, dropout)
- Automated parameter search - NAS



Bertrand, 2019