

# Neural Networks.

Petr Pošík

Czech Technical University in Prague  
Faculty of Electrical Engineering  
Dept. of Cybernetics

<b>Intro</b>	<b>2</b>
Notation .....	3
Multiple regression.....	4
Logistic regression .....	5
Gradient descent.....	6
Ex: Grad. for MR.....	7
Ex: Grad. for LR .....	8
Relations to NN .....	9
<b>Multilayer FFN</b>	<b>10</b>
MLP .....	11
MLP: A look inside.....	12
Activation functions.....	13
MLP: Learning .....	14
BP .....	15
BP algorithm .....	17
BP: Example.....	18
BP Efficiency .....	19
Loss functions .....	20
<b>Gradient Descent</b>	<b>21</b>
Learning rate.....	22
Weights update.....	23
Momentum .....	24
GD improvements .....	25
<b>Regularization</b>	<b>26</b>
Ridge.....	27
Dropout .....	28
<b>Other NNs</b>	<b>29</b>
Beyond MLPs .....	30
<b>Summary</b>	<b>31</b>
Competencies .....	32

**Notation**

In *supervised learning*, we work with

- an observation described by a vector  $\mathbf{x} = (x_1, \dots, x_D)$ ,
- the corresponding true value of the dependent variable  $y$ , and
- the prediction of a model  $\hat{y} = f_w(\mathbf{x})$ , where the model parameters are in vector  $\mathbf{w}$ .
- Very often, we use *homogeneous coordinates* and matrix notation, and represent the whole training data set as  $T = (\mathbf{X}, \mathbf{y})$ , where

$$\mathbf{X} = \begin{pmatrix} 1 & \mathbf{x}^{(1)} \\ \vdots & \vdots \\ 1 & \mathbf{x}^{(|T|)} \end{pmatrix}, \quad \text{and} \quad \mathbf{y} = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(|T|)} \end{pmatrix}.$$

*Learning* then amounts to finding such model parameters  $\mathbf{w}^*$  which minimize certain loss (or energy) function:

$$\mathbf{w}^* = \arg \min_{\mathbf{w}} J(\mathbf{w}, T)$$

**Multiple linear regression**

Multiple linear regression model:

$$\hat{y} = f_w(\mathbf{x}) = w_1x_1 + w_2x_2 + \dots + w_Dx_D = \mathbf{x}\mathbf{w}^T$$

The minimum of

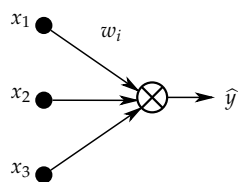
$$J_{MSE}(\mathbf{w}) = \frac{1}{|T|} \sum_{i=1}^{|T|} (y^{(i)} - \hat{y}^{(i)})^2,$$

is given by

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y},$$

or found by numerical optimization.

Multiple regression as a **linear neuron**:



## Logistic regression

Logistic regression model:

$$\hat{y} = f(\mathbf{w}, \mathbf{x}) = g(\mathbf{x}\mathbf{w}^T),$$

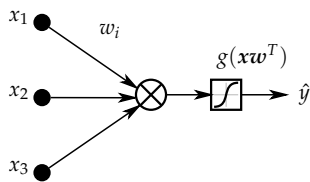
where

$$g(z) = \frac{1}{1 + e^{-z}}$$

is the **sigmoid** (a.k.a **logistic**) function.

- No explicit equation for the optimal weights.
- The only option is to find the optimum numerically, usually by some form of gradient descent.

Logistic regression as a **non-linear neuron**:



## Gradient descent algorithm

- Given a function  $J(\mathbf{w})$  that should be minimized,
- start with a guess of  $\mathbf{w}$ , and change it so that  $J(\mathbf{w})$  decreases, i.e.
- update our current guess of  $\mathbf{w}$  by taking a step in the direction opposite to the gradient:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J(\mathbf{w}), \text{ i.e.}$$

$$w_d \leftarrow w_d - \eta \frac{\partial}{\partial w_d} J(\mathbf{w}),$$

where all  $w_d$ s are updated simultaneously and  $\eta$  is a **learning rate** (step size).

- For cost functions given as the sum across the training examples

$$J(\mathbf{w}) = \sum_{i=1}^{|\mathcal{T}|} E(\mathbf{w}, \mathbf{x}^{(i)}, y^{(i)}),$$

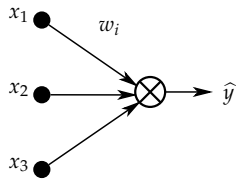
we can concentrate on a single training example because

$$\frac{\partial}{\partial w_d} J(\mathbf{w}) = \sum_{i=1}^{|\mathcal{T}|} \frac{\partial}{\partial w_d} E(\mathbf{w}, \mathbf{x}^{(i)}, y^{(i)}),$$

and we can drop the indices over training data set:

$$E = E(\mathbf{w}, \mathbf{x}, y).$$

### Example: Gradient for multiple regression and squared loss



Assuming the squared error loss

$$E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(y - \mathbf{x}\mathbf{w}^T)^2,$$

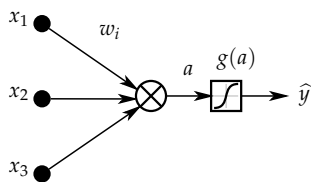
we can compute the derivatives using the chain rule as

$$\begin{aligned} \frac{\partial E}{\partial w_d} &= \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_d}, \text{ where} \\ \frac{\partial E}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}} \frac{1}{2}(y - \hat{y})^2 = -(y - \hat{y}), \text{ and} \\ \frac{\partial \hat{y}}{\partial w_d} &= \frac{\partial}{\partial w_d} \mathbf{x}\mathbf{w}^T = x_d, \end{aligned}$$

and thus

$$\frac{\partial E}{\partial w_d} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_d} = -(y - \hat{y})x_d.$$

### Example: Gradient for logistic regression and crossentropy loss



Nonlinear *activation* function:

$$g(a) = \frac{1}{1 + e^{-a}}$$

Note that

$$g'(a) = g(a)(1 - g(a)).$$

Assuming the crossentropy loss

$$E(\mathbf{w}, \mathbf{x}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}), \quad \text{where } \hat{y} = g(a) = g(\mathbf{x}\mathbf{w}^T),$$

we can compute the derivatives using the chain rule as

$$\begin{aligned} \frac{\partial E}{\partial w_d} &= \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a} \frac{\partial a}{\partial w_d}, \text{ where} \\ \frac{\partial E}{\partial \hat{y}} &= -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} = -\frac{y - \hat{y}}{\hat{y}(1 - \hat{y})}, \\ \frac{\partial \hat{y}}{\partial a} &= \hat{y}(1 - \hat{y}), \text{ and } \frac{\partial a}{\partial w_d} = \frac{\partial}{\partial w_d} \mathbf{x}\mathbf{w}^T = x_d, \end{aligned}$$

and thus

$$\frac{\partial E}{\partial w_d} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a} \frac{\partial a}{\partial w_d} = -(y - \hat{y})x_d.$$

## Relations to neural networks

- Above, we derived training algorithms (based on gradient descent) for linear regression model and linear classification model.
- Note the similarity with the *perceptron algorithm* (“just add certain part of a misclassified training example to the weight vector”).
- Units like those above are used as **building blocks** for more complex/flexible models!

A more complex/flexible model:

$$\hat{y} = g^{OUT} \left( \sum_{k=1}^K w_k^{HID} g_k^{HID} \left( \sum_{d=1}^D w_{kd}^{IN} x_d \right) \right),$$

which is

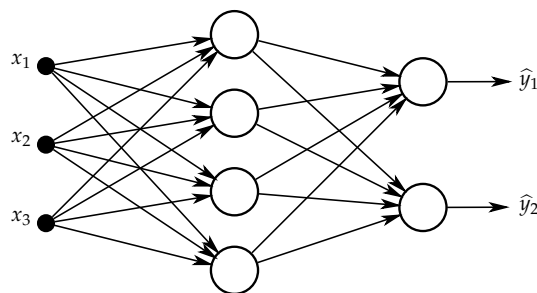
- a nonlinear function of
  - a linear combination of
    - nonlinear functions of
      - linear combinations of inputs.

## Multilayer Feedforward Networks

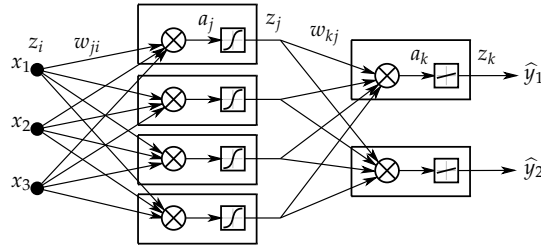
### MLP

#### Multilayer perceptron (MLP)

- Multilayer feedforward network:
  - the „signal“ is propagated from inputs towards outputs; no feedback connections exist.
- It realizes mapping from  $\mathcal{R}^D \rightarrow \mathcal{R}^C$ , where  $D$  is the number of object features, and  $C$  is the number of output variables.
  - For binary classification and regression, a single output is sufficient.
  - For classification into multiple classes, 1-of-N encoding is usually used.
- **Universal approximation theorem:** A MLP with a single hidden layer with sufficient (but finite) number of neurons can approximate any continuous function arbitrarily well (under mild assumptions on the activation functions).



## MLP: A look inside



### Forward propagation:

- Given all the weights  $w$  and activation functions  $g$ , we can for a single input vector  $x$  easily compute the estimate of the output vector  $\hat{y}$  by iteratively evaluating in individual layers:

$$a_j = \sum_{i \in \text{Src}(j)} w_{ji} z_i \quad (1)$$

$$z_j = g(a_j) \quad (2)$$

- Note that
  - $z_i$  in (1) may be the *outputs of hidden layers neurons* or the *inputs*  $x_i$ , and
  - $z_j$  in (2) may be the *outputs of hidden layers neurons* or the *outputs*  $\hat{y}_k$ .

## Activation functions

- Identity:  $g(a) = a$
- Binary step:  $g(a) = \begin{cases} 0 & \text{for } a < 0, \\ 1 & \text{for } a \geq 0 \end{cases}$
- Logistic (sigmoid):  $g(a) = \sigma(a) = \frac{1}{1+e^{-a}}$
- Hyperbolic tangent:  $g(a) = \tanh(a) = 2\sigma(a) - 1$
- Rectified Linear unit (ReLU):  $g(a) = \max(0, a) = \begin{cases} 0 & \text{for } a < 0, \\ a & \text{for } a \geq 0 \end{cases}$
- Leaky ReLU:  $g(a) = \begin{cases} 0.01a & \text{for } a < 0, \\ a & \text{for } a \geq 0 \end{cases}$
- ...

## MLP: Learning

How to train a NN (i.e. find suitable  $w$ ) given the training data set  $(X, y)$ ?

In principle, MLP can be trained in the same way as a single-layer NN using a gradient descent algorithm:

- Define the loss function to be minimized, e.g. squared error loss:

$$J(w) = \sum_{i=1}^{|T|} E(w, x^{(i)}, y^{(i)}) = \frac{1}{2} \sum_{i=1}^{|T|} \sum_{k=1}^C (y_{ik} - \hat{y}_{ik})^2, \quad \text{where}$$

$$E(w, x, y) = \frac{1}{2} \sum_{k=1}^C (y_k - \hat{y}_k)^2.$$

$|T|$  is the size of the training set, and  $C$  is the number of outputs of NN.

- Compute the gradient of the loss function w.r.t. individual weights:

$$\nabla E(w) = \left( \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_W} \right).$$

- Make a step in the direction opposite to the gradient to update the weights:

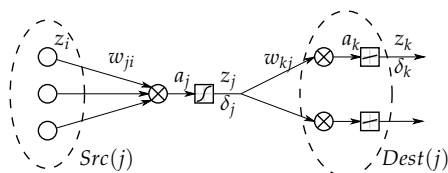
$$w_d \leftarrow w_d - \eta \frac{\partial E}{\partial w_d} \quad \text{for } d = 1, \dots, W.$$

How to compute the individual derivatives?

## Error backpropagation

**Error backpropagation (BP)** is the algorithm for computing  $\frac{\partial E}{\partial w_d}$ .

Consider only  $\frac{\partial E}{\partial w_d}$  because  $\frac{\partial J}{\partial w_d} = \sum_n \frac{\partial}{\partial w_d} E(w, x^{(n)}, y^{(n)})$ .



$E$  depends on  $w_{ji}$  only via  $a_j$ :

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} \quad (3)$$

Let's introduce the so called **error**  $\delta_j$ :

$$\delta_j = \frac{\partial E}{\partial a_j} \quad (4)$$

From (1) we can derive:

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \quad (5)$$

Substituting (4) and (5) into (3):

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i, \quad (6)$$

where

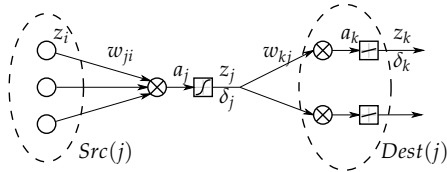
$\delta_j$  is the error of the neuron on the output of the edge  $i \rightarrow j$ , and  $z_i$  is the input of the edge  $i \rightarrow j$ .

"The more we excite edge  $i \rightarrow j$  (big  $z_i$ ) and the larger is the error of the neuron on its output (large  $\delta_j$ ), the more sensitive is the loss function  $E$  to the change of  $w_{ji}$ ."

- All values  $z_i$  are known from forward pass,
- to compute the gradient, we need to compute all  $\delta_j$ .

## Error backpropagation (cont.)

We need to compute the *errors*  $\delta_j$ .



**For the output layer:**

$$\delta_k = \frac{\partial E}{\partial a_k}$$

$E$  depends on  $a_k$  only via  $\hat{y}_k = g(a_k)$ :

$$\delta_k = \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k} = g'(a_k) \frac{\partial E}{\partial \hat{y}_k} \quad (7)$$

**For the hidden layers:**

$$\delta_j = \frac{\partial E}{\partial a_j}$$

$E$  depends on  $a_j$  via all  $a_k$ ,  
 $k \in \text{Dest}(j)$ :

$$\begin{aligned} \delta_j &= \frac{\partial E}{\partial a_j} = \sum_{k \in \text{Dest}(j)} \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \\ &= \sum_{k \in \text{Dest}(j)} \delta_k \frac{\partial a_k}{\partial a_j} = \\ &= g'(a_j) \sum_{k \in \text{Dest}(j)} w_{kj} \delta_k, \end{aligned} \quad (8)$$

because

$$a_k = \sum_{j \in \text{Src}(k)} w_{kj} z_j = \sum_{j \in \text{Src}(k)} w_{kj} g(a_j),$$

$$\text{and thus } \frac{\partial a_k}{\partial a_j} = w_{kj} g'(a_j)$$

“The error  $\delta_k$  is distributed to  $\delta_j$  in the lower layer according to the weight  $w_{kj}$  (which is the speed of growth of the linear combination  $a_k$ ) and according to the size of  $g'(a_j)$  (which is the speed of growth of the activation function).”

## Error backpropagation algorithm

**Algorithm 1:** Error Backpropagation: the computation of derivatives  $\frac{\partial E}{\partial w_d}$ .

1 **begin**

2     Perform a forward pass for observation  $x$ . This will result in values of all  $a_j$  and  $z_j$  for the vector  $x$ .

3     Evaluate the error  $\delta_k$  for the output layer (using Eq. 7):

$$\delta_k = g'(a_k) \frac{\partial E}{\partial \hat{y}_k}$$

4     Using Eq. 8, propagate the errors  $\delta_k$  back to get all the remaining  $\delta_j$ :

$$\delta_j = g'(a_j) \sum_{k \in \text{Dest}(j)} w_{kj} \delta_k$$

5     Using Eq. 6, evaluate all the derivatives to get the whole gradient:

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i$$



## Error backpropagation: Example

NN with a single hidden layer:

- Squared error loss:  $E = \frac{1}{2} \sum_{k=1}^C (y_k - \hat{y}_k)^2$
- Activation func. in the output layer: identity  $g_k(a_k) = a_k$ ,  $g'_k(a_k) = 1$
- Activation func. in the hidden layer: sigmoidal  $g_j(a_j) = \frac{1}{1 + e^{-a_j}}$ ,  $g'_j(a_j) = z_j(1 - z_j)$

Computing the errors  $\delta$ :

- Output layer:  $\delta_k = g'_k(a_k) \frac{\partial E}{\partial \hat{y}_k} = -(y_k - \hat{y}_k)$
- Hidden layer:  $\delta_j = g'_j(a_j) \sum_{k \in Dest(j)} w_{kj} \delta_k = z_j(1 - z_j) \sum_{k \in Dest(j)} w_{kj} \delta_k$

Computation of all the partial derivatives:

$$\frac{\partial E}{\partial w_{ji}} = \delta_j x_i$$

$$\frac{\partial E}{\partial w_{kj}} = \delta_k z_j$$

Online learning:

$$w_{ji} \leftarrow w_{ji} - \eta \delta_j x_i$$

$$w_{kj} \leftarrow w_{kj} - \eta \delta_k z_j$$

Batch learning:

$$w_{ji} \leftarrow w_{ji} - \eta \sum_{n=1}^{|T|} \delta_j^{(n)} x_i^{(n)}$$

$$w_{kj} \leftarrow w_{kj} - \eta \sum_{n=1}^{|T|} \delta_k^{(n)} z_j^{(n)}$$

## Error backpropagation efficiency

Let  $W$  be the number of weights in the network (the number of parameters being optimized).

- The evaluation of  $E$  for a single observation requires  $\mathcal{O}(W)$  operations (evaluation of  $w_{ji} z_i$  dominates, evaluation of  $g(a_j)$  is neglected).

We need to compute  $W$  derivatives for each observation:

- Classical approach:
  - Find explicit equations for  $\frac{\partial E}{\partial w_{ji}}$ .
  - To compute each of them  $\mathcal{O}(W)$  steps are required.
  - In total,  $\mathcal{O}(W^2)$  steps for a single training example.
- Backpropagation:
  - Requires only  $\mathcal{O}(W)$  steps for a single training example.

## Loss functions

Task	Suggested loss function
Binary classification	Cross-entropy: $J = - \sum_{i=1}^{ T } [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$
Multinomial classification	Multinomial cross-entropy: $J = - \sum_{i=1}^{ T } \sum_{k=1}^C I(y^{(i)} = k) \log \hat{y}_k^{(i)}$
Regression	Squared error: $J = \sum_{i=1}^{ T } (y^{(i)} - \hat{y}^{(i)})^2$
Multi-output regression	Squared error: $J = \sum_{i=1}^{ T } \sum_{k=1}^C (y_k^{(i)} - \hat{y}_k^{(i)})^2$

Note: often, mean errors are used.

- Computed as the average w.r.t. the number of training examples  $|T|$ .
- The optimum is in the same point, of course.

## Gradient Descent

### Learning rate annealing

**Task:** find such parameters  $w^*$  which minimize the model cost over the training set, i.e.

$$w^* = \arg \min_w J(w; X, y)$$

Gradient descent:  $w^{(t+1)} = w^{(t)} - \eta^{(t)} \nabla J(w^{(t)})$ ,  
where  $\eta^{(t)} > 0$  is the **learning rate** or **step size** at iteration  $t$ .

**Learning rate decay:**

- Decrease the learning rate in time.
- **Step decay:** reduce the learning rate every few iterations by certain factor, e.g.  $\frac{1}{2}$ .
- **Exponential decay:**  $\eta^{(t)} = \eta_0 e^{-kt}$
- **Hyperbolic decay:**  $\eta^{(t)} = \frac{\eta_0}{1+kt}$

## Weights update

When should we update the weights?

### ■ Batch learning:

- Compute the gradient w.r.t. all the training examples (epoch).
- Several epochs are required to train the network.
- Inefficient for redundant datasets.

### ■ Online learning:

- Compute the gradient w.r.t. a single training example only.
- **Stochastic Gradient Descent (SGD)**
- Converges almost surely to local minimum when  $\eta^{(t)}$  decreases appropriately in time.

### ■ Mini-batch learning:

- Compute the gradient w.r.t. a small subset of the training examples.
- A compromise between the above 2 extremes.

## Momentum

### Momentum

- Perform the update in an analogy to physical systems: a particle with certain mass and velocity gets acceleration from the gradient (“force”) of the loss function:

$$\begin{aligned} \mathbf{v}^{(t+1)} &= \mu \mathbf{v}^{(t)} + \eta^{(t)} \nabla J(\mathbf{w}^{(t)}) \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + \mathbf{v}^{(t+1)} \end{aligned}$$

- SGD with momentum tends to keep traveling in the same direction, preventing oscillations.
- It builds the velocity in directions with consistent (but possibly small) gradient.

### Nesterov’s Momentum

- Slightly different update equations:

$$\begin{aligned} \mathbf{v}^{(t+1)} &= \mu \mathbf{v}^{(t)} + \eta^{(t)} \nabla J(\mathbf{w}^{(t)} + \mu \mathbf{v}^{(t)}) \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + \mathbf{v}^{(t+1)} \end{aligned}$$

- Classic momentum corrects the velocity using gradient at  $\mathbf{w}^{(t)}$ ; Nesterov uses gradient at  $\mathbf{w}^{(t)} + \mu \mathbf{v}^{(t)}$  which is more similar to  $\mathbf{w}^{(t+1)}$ .
- Stronger theoretical convergence guarantees; slightly better in practice.

## Further gradient descent improvements

### Resilient Propagation (Rprop)

- $\frac{\partial J}{\partial w_d}$  may differ a lot for different parameters  $w_d$ .
- Rprop does not use the value, only its *sign* to adapt the step size for each weight separately.
- Often, an order of magnitude faster than basic GD.
- Does not work well for mini-batches.

### Adaptive Gradient (Adagrad)

- Idea: Reduce learning rates for parameters having high values of gradient.

### Root Mean Square Propagation (RMSprop)

- Similar to AdaGrad, but employs a moving average of the gradient values.
- Can be seen as a generalization of Rprop, can work also with mini-batches.

### Adaptive Moment Estimation (Adam)

- Improvement of RMSprop.
- Uses moving averages of gradients and their second moments.

See also:

- <http://sebastianruder.com/optimizing-gradient-descent/>
- <http://cs231n.github.io/neural-networks-3/>
- <http://cs231n.github.io/assets/nn3/opt2.gif>, <http://cs231n.github.io/assets/nn3/opt1.gif>

## Regularization

### Overfitting and regularization

*Overfitting in NN* is often characterized by weight values that are very large in magnitude. How to deal with it?

- Get more data.
- Use a simpler model (less hidden layers, less neurons, different activation functions).
- Use *regularization* (penalize the model complexity).

#### Ridge regularization:

- Modified loss function, e.g. for squared error:

$$J'(w) = J(w) + \text{penalty} = \frac{1}{2m} \sum_{i=1}^m (y^{(i)} - \mathbf{x}^{(i)} w^T)^2 + \frac{\alpha}{m} \sum_{d=1}^D w_d^2.$$

- Modified weight update in GD:

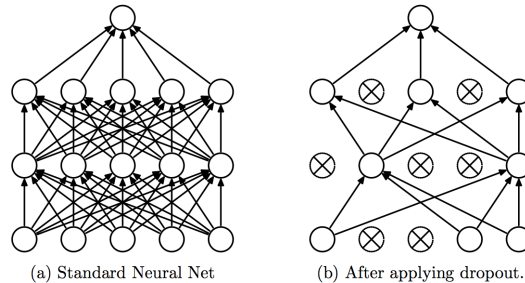
$$w_d \leftarrow w_d - \eta \frac{\partial J'}{\partial w_d} = \underbrace{\left(1 - \frac{\eta \alpha}{m}\right)}_{\text{weight decay}} w_d - \eta \frac{\partial J}{\partial w_d},$$

where  $\eta$  is the learning rate,  $\alpha$  is the regularization strength,  $m$  is the number of examples in the batch.

- The biases (weights connected to constant 1) should not be regularized!

## Dropout

- Idea: Average many NNs, share weights to make it computationally feasible.
- For each training example, omit each neuron with certain probability (often  $p = 0.5$ ).
- This is like sampling from  $2^N$  networks where  $N$  is the number of units.
- Only a small part of the  $2^N$  networks is actually sampled.
- Prevents coadaptation of feature vectors.



(a) Standard Neural Net (b) After applying dropout.  
Srivastava et al.: A Simple Way to Prevent Neural Networks from Overfitting, 2014

## Other types of Neural Networks

### Beyond MLPs

MLPs are only one type of neural networks. Other types of FFNNs include:

- **Radial basis functions (RBF) nets.** Neurons contain prototypes, forward propagation resembles a (smoothed) nearest neighbors method.
- **Autoencoders.** Learn a compact representation of the input data.
- **Convolutional nets.** Replace the fully-connected layer with a convolutional layer that has smaller number of weights and reuses them for many input variables. Aimed at image processing.
- ...

**Recurrent nets** contain also *feedback connections*.

- They preserve a kind of *state* of the network.
- Simple recurrent architectures: **Jordan, Elman.** Network output or state used together with input in the next iteration.
- **Hopfield net.** Used as associative memory.
- **Long short-term memory (LSTM).** Suitable for processing data sequences in time.
- ...

Other architectures:

- **Kohonen's self-organizing maps (SOM).** Used for unsupervised learning.
- **Neural gas.** Used e.g. to approximately solve the traveling salesperson problem.
- ...

**Competencies**

After this lecture, a student shall be able to ...

- describe the model of a simple neuron, and explain its relation to multivariate regression and logistic regression;
- explain how to find weights of a single neuron using gradient descent (GD) algorithm;
- derive the update equations used in GD to optimize the weights of a single neuron for various loss functions and various activation functions;
- describe a multilayer feedforward network and discuss its usage and characteristics;
- compare the use of GD in case of a single neuron and in case of NN, discuss similarities and differences;
- explain the error backpropagation (BP) algorithm — its purpose and principle;
- implement BP algorithm for a simple NN, and suggest how the implementation should be modified to allow application for complex networks;
- discuss the purpose of various modifications of GD algorithm (learning rate decay, weight update schedule, momentum, ...);
- discuss the regularization options for NN (weight decay, dropout);
- be aware of other types of NNs, not only feedforward nets.