Neural Networks.

Petr Pošík

Czech Technical University in Prague

Faculty of Electrical Engineering

Dept. of Cybernetics

Introduction and Rehearsal

Notation

In *supervised learning*, we work with

- an observation described by a vector $x = (x_1, \ldots, x_D)$,

- the corresponding true value of the dependent variable y, and

- the prediction of a model $\widehat{y} = f_w(x)$, where the model parameters are in vector w.

Notation

In *supervised learning*, we work with

- an observation described by a vector $x = (x_1, \ldots, x_D)$,

- the corresponding true value of the dependent variable y, and

- the prediction of a model $\widehat{y} = f_w(x)$, where the model parameters are in vector w.

- Very often, we use *homogeneous coordinates* and matrix notation, and represent the whole training data set as $T = (X, y)$, where

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

Notation

In *supervised learning*, we work with

- an observation described by a vector $\boldsymbol{x} = (x_1, \ldots, x_D)$,

- the corresponding true value of the dependent variable y, and

- the prediction of a model $\widehat{y} = f_{\boldsymbol{w}}(\boldsymbol{x})$, where the model parameters are in vector $\boldsymbol{w}$.

- Very often, we use *homogeneous coordinates* and matrix notation, and represent the whole training data set as $T = (\boldsymbol{X}, \boldsymbol{y})$, where

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

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

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

Multiple linear regression

Multiple linear regression model:

$$\widehat{y} = f_{\boldsymbol{w}}(\boldsymbol{x}) = w_1 x_1 + w_2 x_2 + \ldots + w_D x_D = \boldsymbol{x}\boldsymbol{w}^T$$

The minimum of

$$J_{MSE}(\boldsymbol{w}) = \frac{1}{|T|} \sum_{i=1}^{|T|} \left(y^{(i)} - \widehat{y}^{(i)} \right)^2 ,$$

is given by

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

or found by numerical optimization.

Multiple linear regression

Multiple linear regression model:

$$\widehat{y} = f_{\boldsymbol{w}}(\boldsymbol{x}) = w_1 x_1 + w_2 x_2 + \ldots + w_D x_D = \boldsymbol{x} \boldsymbol{w}^T$$

The minimum of

$$J_{MSE}(\boldsymbol{w}) = \frac{1}{|T|} \sum_{i=1}^{|T|} \left(y^{(i)} - \widehat{y}^{(i)} \right)^2 ,$$

is given by

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

or found by numerical optimization.

Multiple regression as a **linear neuron**:

Logistic regression

Logistic regression model:

$$\widehat{y} = f(\boldsymbol{w}, \boldsymbol{x}) = g(\boldsymbol{x}\boldsymbol{w}^T),$$

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

Logistic regression model:

$$\widehat{y} = f(\boldsymbol{w}, \boldsymbol{x}) = g(\boldsymbol{x}\boldsymbol{w}^T),$$
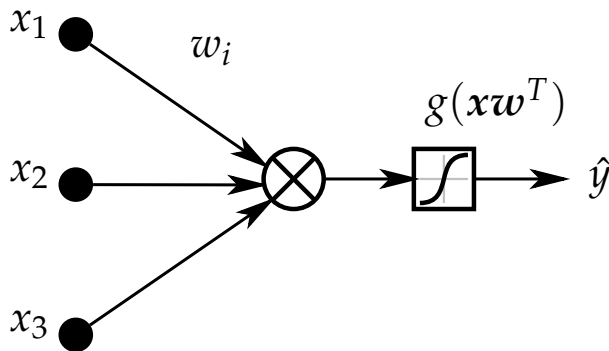
where

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

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 $J(w)$ that should be minimized,

■ start with a guess of w, and change it so that $J(w)$ decreases, i.e.

■ update our current guess of w by taking a step in the direction opposite to the gradient:

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

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

where all w_ds are updated simultaneously and α is a **learning rate** (step size).

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

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

we can concentrate on a single training example because
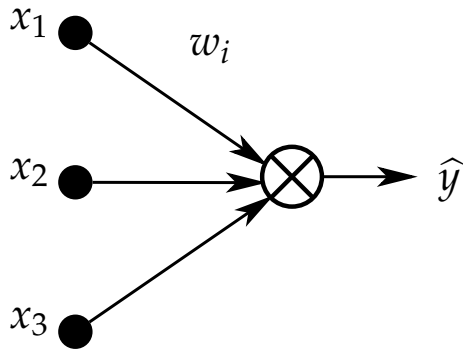
$$\frac{\partial}{\partial w_d} J(w) = \sum_{i=1}^{|T|} \frac{\partial}{\partial w_d} E(w, x^{(i)}, y^{(i)}),$$

and we can drop the indices over training data set:

$$E = E(w, x, y).$$

Example: Gradient for multiple regression and squared loss

Assuming the squared error loss

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

we can compute the derivatives using the chain rule as

$$\frac{\partial E}{\partial w_d} = \frac{\partial E}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial w_d}, \text{where}$$

$$\frac{\partial E}{\partial \widehat{y}} = \frac{\partial}{\partial \widehat{y}} \frac{1}{2}(y - \widehat{y})^2 = -(y - \widehat{y}), \text{and}$$

$$\frac{\partial \widehat{y}}{\partial w_d} = \frac{\partial}{\partial w_d} \boldsymbol{x}\boldsymbol{w}^T = x_d,$$

and thus

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

Nonlinear *activation* function:

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

Note that

$$g'(a) = g(a)\left(1 - g(a)\right).$$

Example: Gradient for logistic regression and crossentropy loss
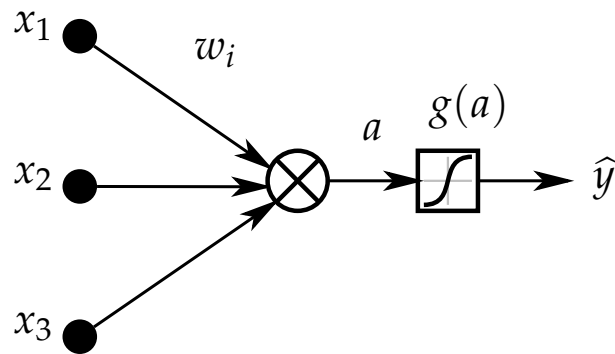
Nonlinear *activation* function:

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

Note that

$$g'(a) = g(a)\left(1 - g(a)\right).$$

Assuming the crossentropy loss

$$E(\boldsymbol{w}, \boldsymbol{x}, y) = -y\log\widehat{y} - (1-y)\log(1-\widehat{y}), \qquad \text{where } \widehat{y} = g(a) = g(\boldsymbol{x}\boldsymbol{w}^T),$$

we can compute the derivatives using the chain rule as

$$\frac{\partial E}{\partial w_d} = \frac{\partial E}{\partial \widehat{y}}\frac{\partial \widehat{y}}{\partial a}\frac{\partial a}{\partial w_d}, \text{ where}$$

$$\frac{\partial E}{\partial \widehat{y}} = -\frac{y}{\widehat{y}} + \frac{1-y}{1-\widehat{y}} = -\frac{y - \widehat{y}}{\widehat{y}(1-\widehat{y})},$$

$$\frac{\partial \widehat{y}}{\partial a} = \widehat{y}(1 - \widehat{y}), \text{ and } \frac{\partial a}{\partial w_d} = \frac{\partial}{\partial w_d}\boldsymbol{x}\boldsymbol{w}^T = x_d,$$

and thus

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

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!

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:

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

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 $\mathcal{R}^D \longrightarrow \mathcal{R}^C$, 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).
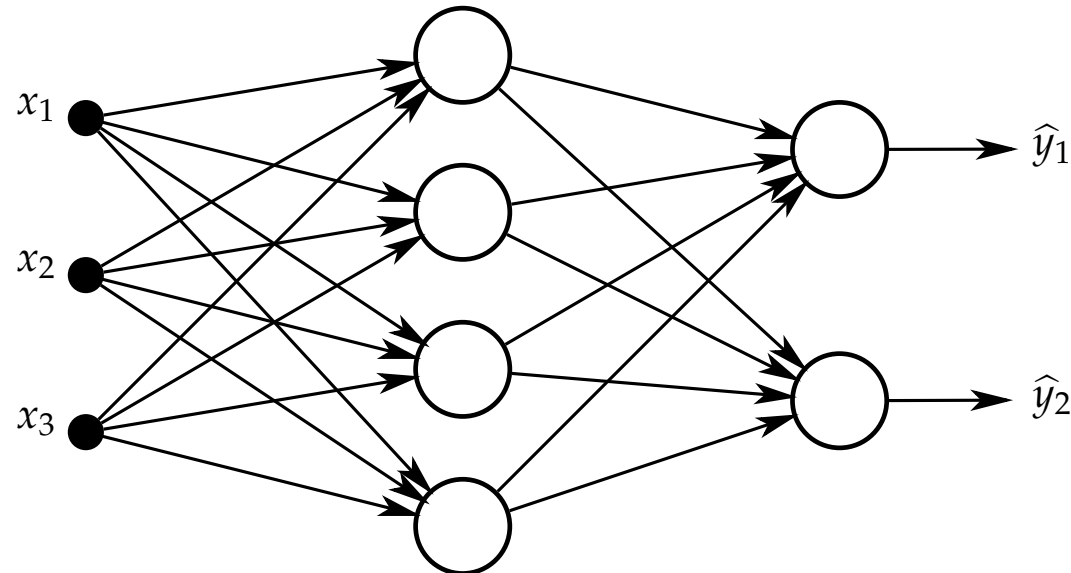
MLP: A look inside
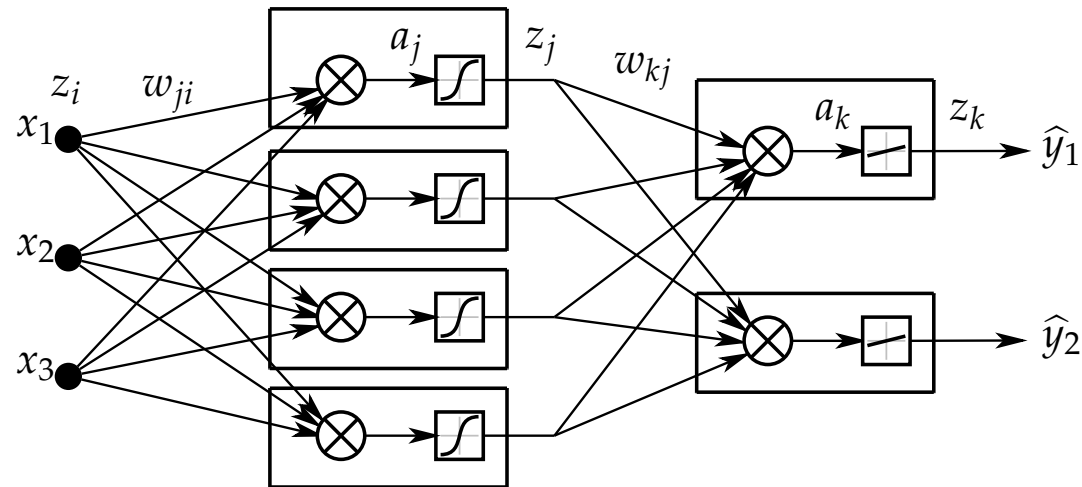
Forward propagation:

- 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 $\widehat{y}$ by iteratively evaluating in individual layers:

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

$$z_j = g(a_j) \tag{2}$$

- 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 the *outputs of hidden layers neurons* or the *outputs* $\widehat{y}_k$.

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)?

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:

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

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

$|T|$ 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:

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

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

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

How to compute the individual derivatives?

Error backpropagation

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

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

$$\frac{\partial J}{\partial w_d} = \sum_n \frac{\partial}{\partial w_d} E(\boldsymbol{w}, \boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}).$$
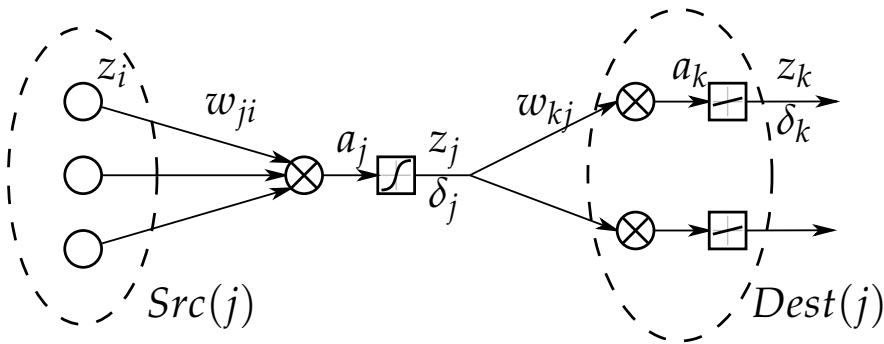
E depends on w_{ji} only via a_j:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} \tag{3}$$

Let's introduce the so called *error* δ_j:

$$\delta_j = \frac{\partial E}{\partial a_j} \tag{4}$$

From (1) we can derive:

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \tag{5}$$

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

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i, \tag{6}$$

where

δ_j is the error of the neuron on the output of the

z_i is the input of the edge $i \to j$.

"The more we excite edge $i \to j$ (big z_i) and the larger is the error of the neuron on its output (large δ_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 δ_j.

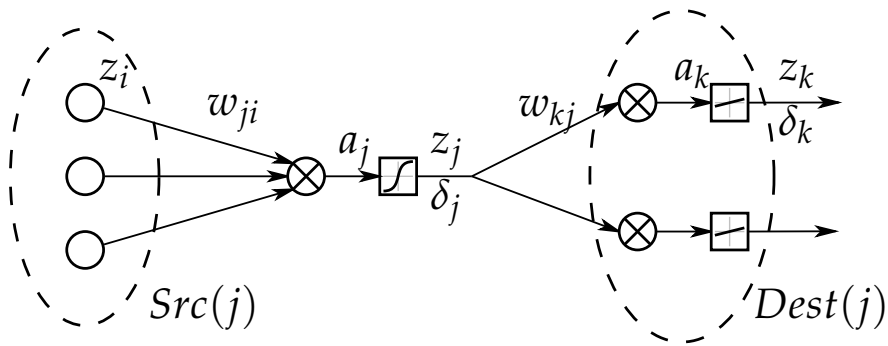# Error backpropagation (cont.)

We need to compute the *errors* δ_j.

$Src(j)$ $Dest(j)$

For the output layer:

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

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

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

For the hidden layers:

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

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

$$\delta_j = \frac{\partial E}{\partial a_j} = \sum_{k \in Dest(j)} \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial a_j} =$$

$$= \sum_{k \in Dest(j)} \delta_k \frac{\partial a_k}{\partial a_j} =$$

$$= g'(a_j) \sum_{k \in Dest(j)} w_{kj}\delta_k, \qquad (8)$$

because

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

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

"The error δ_k is distributed to δ_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 δ_k for the output layer (using Eq. 7):

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

4   Using Eq. 8, propagate the errors δ_k back to get all the remaining δ_j:

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

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: $E = \dfrac{1}{2} \sum_{k=1}^{C} (y_k - \widehat{y}_k)^2$

- Activation func. in the output layer: identity $g_k(a_k) = a_k, \quad g_k'(a_k) = 1$

- Activation func. in the hidden layer: sigmoidal $g_j(a_j) = \dfrac{1}{1 + e^{-a_j}}, \quad g_j'(a_j) = z_j(1 - z_j)$

Error backpropagation: Example

NN with a single hidden layer:

- Squared error loss: $E = \dfrac{1}{2} \sum_{k=1}^{C} (y_k - \widehat{y}_k)^2$

- 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 $g_j(a_j) = \dfrac{1}{1 + e^{-a_j}}$, $g_j'(a_j) = z_j(1 - z_j)$

Computing the errors δ:

- Output layer: $\delta_k = g_k'(a_k) \dfrac{\partial E}{\partial \widehat{y}_k} = -(y_k - \widehat{y}_k)$

Error backpropagation: Example

NN with a single hidden layer:

- Squared error loss: $E = \dfrac{1}{2} \sum_{k=1}^{C} (y_k - \widehat{y}_k)^2$

- 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 $g_j(a_j) = \dfrac{1}{1 + e^{-a_j}}$, $g'_j(a_j) = z_j(1 - z_j)$

Computing the errors δ:

- Output layer: $\delta_k = g'_k(a_k) \dfrac{\partial E}{\partial \widehat{y}_k} = -(y_k - \widehat{y}_k)$

- Hidden layer: $\delta_j = g'_j(a_j) \sum_{k \in Dest(j)} w_{kj} \delta_k = z_j(1 - z_j) \sum_{k \in Dest(j)} w_{kj} \delta_k$

Error backpropagation: Example

NN with a single hidden layer:

- Squared error loss: $E = \dfrac{1}{2} \sum\limits_{k=1}^{C} (y_k - \widehat{y}_k)^2$

- 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 $g_j(a_j) = \dfrac{1}{1 + e^{-a_j}},\ \ g'_j(a_j) = z_j(1 - z_j)$

Computing the errors δ:

- Output layer: $\delta_k = g'_k(a_k) \dfrac{\partial E}{\partial \widehat{y}_k} = -(y_k - \widehat{y}_k)$

- Hidden layer: $\delta_j = g'_j(a_j) \sum\limits_{k \in Dest(j)} w_{kj} \delta_k = z_j(1 - z_j) \sum\limits_{k \in Dest(j)} w_{kj} \delta_k$

Computation of all the partial derivatives:

$$\frac{\partial E}{\partial w_{ji}} = \delta_j x_i \qquad\qquad\qquad\qquad \frac{\partial E}{\partial w_{kj}} = \delta_k z_j$$

Error backpropagation: Example

NN with a single hidden layer:

- Squared error loss: $E = \dfrac{1}{2} \sum\limits_{k=1}^{C} (y_k - \widehat{y}_k)^2$

- 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 $g_j(a_j) = \dfrac{1}{1 + e^{-a_j}}$, $g'_j(a_j) = z_j(1 - z_j)$

Computing the errors δ:

- Output layer: $\delta_k = g'_k(a_k) \dfrac{\partial E}{\partial \widehat{y}_k} = -(y_k - \widehat{y}_k)$

- Hidden layer: $\delta_j = g'_j(a_j) \sum\limits_{k \in Dest(j)} w_{kj} \delta_k = z_j(1 - z_j) \sum\limits_{k \in Dest(j)} w_{kj} \delta_k$

Computation of all the partial derivatives:

$$\frac{\partial E}{\partial w_{ji}} = \delta_j x_i \qquad\qquad \frac{\partial E}{\partial w_{kj}} = \delta_k z_j$$

Online learning:

$$w_{ji} \longleftarrow w_{ji} - \eta \delta_j x_i$$
$$w_{kj} \longleftarrow w_{kj} - \eta \delta_k z_j$$

Batch learning:

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

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

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: $J = -\sum_{i=1}^{|T|} \left[y^{(i)} \log \widehat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \widehat{y}^{(i)}) \right]$
Multinomial classification	Multinomial cross-entropy: $J = -\sum_{i=1}^{|T|} \sum_{k=1}^{C} I(y^{(i)} = k) \log \widehat{y}_k^{(i)}$
Regression	Squared error: $J = \sum_{i=1}^{|T|} (y^{(i)} - \widehat{y}^{(i)})^2$
Multi-output regression	Squared error: $J = \sum_{i=1}^{|T|} \sum_{k=1}^{C} (y_k^{(i)} - \widehat{y}_k^{(i)})^2$

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.

$$w^* = \arg\min_{w} J(w; X, y)$$

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 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:

$$v^{(t+1)} = \mu v^{(t)} + \eta^{(t)} \nabla J(w^{(t)})$$
$$w^{(t+1)} = w^{(t)} + v^{(t+1)}$$

■ 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.

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:

$$v^{(t+1)} = \mu v^{(t)} + \eta^{(t)} \nabla J(w^{(t)})$$
$$w^{(t+1)} = w^{(t)} + v^{(t+1)}$$

■ 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:

$$v^{(t+1)} = \mu v^{(t)} + \eta^{(t)} \nabla J(w^{(t)} + \mu v^{(t)})$$
$$w^{(t+1)} = w^{(t)} + v^{(t+1)}$$

■ Classic momentum corrects the velocity using gradient at $w^{(t)}$; Nesterov uses gradient at $w^{(t)} + \mu v^{(t)}$ which is more similar to $w^{(t+1)}$.

■ Stronger theoretical convergence guarantees; slightly better in practice.

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.

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.

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.

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.

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).

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:

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

■ Modified weight update in GD:

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

where η is the learning rate, α 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.

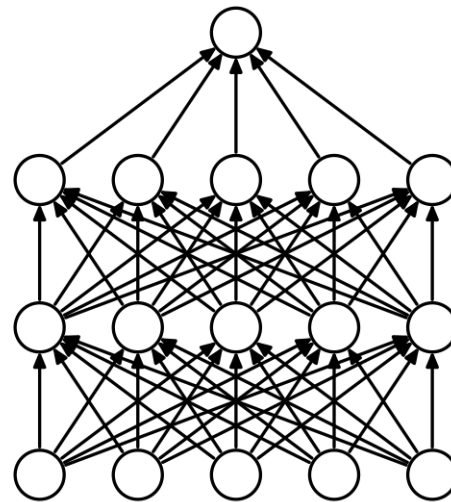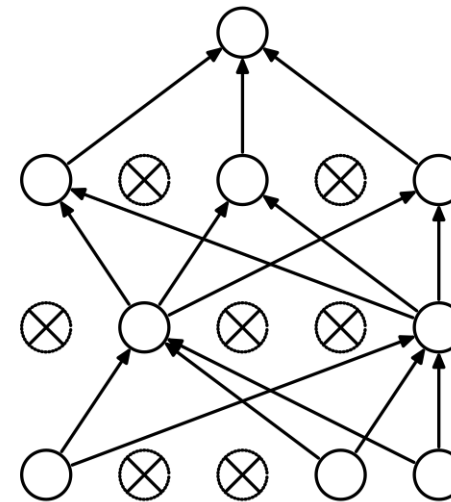(a) Standard Neural Net   (b) After applying dropout.

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.

- . . .

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.

- …

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.

- ...

Summary

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.