Bayesian Classification

Let Ω be a sample space (a universum) of all objects that can be classified. We assume a probability P on Ω.

We consider the problem of **binary classification**:

- Let Y be the random variable for the category which takes values in $\{\mathbf{0}, \mathbf{1}\}$.
- Let X be the random vector describing n features of a given instance, i.e., $X = (X_1, \ldots, X_n)$
  - Denote by $\vec{x} \in \mathbb{R}^n$ values of X,
  - and by $x_i \in \mathbb{R}$ values of X_i.

Bayes classifier: Given a vector of feature values $\vec{x}$,

$$C^{Bayes}(\vec{x}) := \begin{cases} \mathbf{1} & \text{if } P(Y = \mathbf{1} \mid X = \vec{x}) \geq P(Y = \mathbf{0} \mid X = \vec{x}) \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

Intuitively, C^{Bayes} assigns to $\vec{x}$ the most probable category it might be in.

Bayesian Classification

Determine the category for $\vec{x}$ by computing

$$P(Y = y \mid X = \vec{x}) = \frac{P(Y = y) \cdot P(X = \vec{x} \mid Y = y)}{P(X = \vec{x})}$$

for both $y \in \{\mathbf{0}, \mathbf{1}\}$ and deciding whether or not the following holds:

$$P(Y = \mathbf{1} \mid X = \vec{x}) \geq P(Y = \mathbf{0} \mid X = \vec{x})$$

So in order to make the classifier we need to compute:

- The prior $P(Y = \mathbf{1})$ (then $P(Y = \mathbf{0}) = 1 - P(Y = \mathbf{1})$)
- The conditionals $P(X = \vec{x} \mid Y = y)$ for $y \in \{\mathbf{0}, \mathbf{1}\}$ and for every $\vec{x}$

Naive Bayes

- We assume that features are (conditionally) independent *given the category*. That is for all $\vec{x} = (x_1, \ldots, x_n)$ and $y \in \{\mathbf{0}, \mathbf{1}\}$ we **assume**:

$$P(X = x \mid Y = y) = P(X_1 = x_1, \cdots, X_n = x_n \mid Y)$$
$$= \prod_{i=1}^{n} P(X_i = x_i \mid Y = y)$$

- Therefore, we only need to specify $P(X_i = x_i \mid Y = y)$ for each possible pair of a feature-value x_i and $y \in \{\mathbf{0}, \mathbf{1}\}$.

  Note that if all X_i are binary (values in $\{0, 1\}$), this requires specifying only $2n$ parameters:

$$P(X_i = 1 \mid Y = \mathbf{1}) \text{ and } P(X_i = 1 \mid Y = \mathbf{0}) \text{ for each } X_i$$

as $P(X_i = 0 \mid Y = y) = 1 - P(X_i = 1 \mid Y = y)$ for $y \in \{\mathbf{0}, \mathbf{1}\}$.

Compared to specifying 2^n parameters without any independence assumption.

Linear Function Approximation

- Given a set D of training examples:

  $$D = \{(\vec{x}_1, f(\vec{x}_1)), (\vec{x}_2, f(\vec{x}_2)), \ldots, (\vec{x}_p, f(\vec{x}_p))\}$$

  Here $\vec{x}_k = (x_{k1} \ldots, x_{kn}) \in \mathbb{R}^n$ and $f_k(\vec{x}) \in \mathbb{R}$.

  In what follows we use f_k to denote $f(\vec{x}_k)$.

  Our goal: Find $\vec{w}$ so that $h[\vec{w}](\vec{x}) = \vec{w} \cdot \tilde{\mathrm{x}}$ approximates the function f some of whose values are given by the training set.
  Recall that $\tilde{\mathrm{x}}_k = (x_{k0}, x_{k1} \ldots, x_{kn})$.

- **Squared Error Function:**

  $$E(\vec{w}) = \frac{1}{2}\sum_{k=1}^{p}(\vec{w} \cdot \tilde{\mathrm{x}}_k - f_k)^2 = \frac{1}{2}\sum_{k=1}^{p}\left(\sum_{i=0}^{n} w_i x_{ki} - f_k\right)^2$$

Gradient of the Error Function

Consider the **gradient** of the error function:

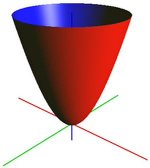
$$\nabla E(\vec{w}) = \left(\frac{\partial E}{\partial w_0}(\vec{w}), \ldots, \frac{\partial E}{\partial w_n}(\vec{w})\right) = \sum_{k=1}^{p} (\vec{w} \cdot \tilde{\mathrm{x}}_k - f_k) \cdot \tilde{\mathrm{x}}_k$$

What is the gradient $\nabla E(\vec{w})$? It is a vector in $\mathbb{R}^{n+1}$ which points in the direction of the steepest *ascent* of E (it's length corresponds to the steepness). Note that here the vectors $\tilde{\mathrm{x}}_k$ are *fixed* parameters of E!

Fakt

If $\nabla E(\vec{w}) = \vec{0} = (0, \ldots, 0)$, *then* $\vec{w}$ *is a global minimum of* E.

This follows from the fact that E is a convex paraboloid that has a unique extreme which is a minimum.



Function Approximation – Learning

Gradient Descent:

- Weights $\vec{w}^{(0)}$ are initialized randomly close to $\vec{0}$.
- In $(t+1)$-th step, $\vec{w}^{(t+1)}$ is computed as follows:

$$\begin{aligned}\vec{w}^{(t+1)} &= \vec{w}^{(t)} - \varepsilon \cdot \nabla E(\vec{w}^{(t)}) \\ &= \vec{w}^{(t)} - \varepsilon \cdot \sum_{k=1}^{p} \left(\vec{w}^{(t)} \cdot \tilde{\mathrm{x}}_k - f_k \right) \cdot \tilde{\mathrm{x}}_k \\ &= \vec{w}^{(t)} - \varepsilon \cdot \sum_{k=1}^{p} \left(h[\vec{w}^{(t)}](\vec{x}_k) - f_k \right) \cdot \tilde{\mathrm{x}}_k\end{aligned}$$

Here $k = (t \mod p) + 1$ and $0 < \varepsilon \leq 1$ is the learning rate.

Note that the algorithm is almost similar to the batch perceptron algorithm!

Tvrzení

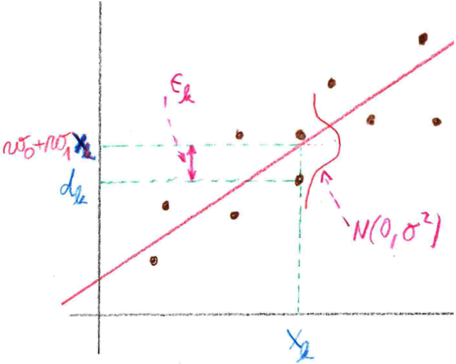
For sufficiently small $\varepsilon > 0$ the sequence $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$ converges (component-wisely) to the global minimum of E.

Maximum Likelihood (GOOD STUDENTS)

Fix a training set $D = \{(x_1, f_1), (x_2, f_2), \ldots, (x_p, f_p)\}$
Assume that each f_k has been generated randomly by

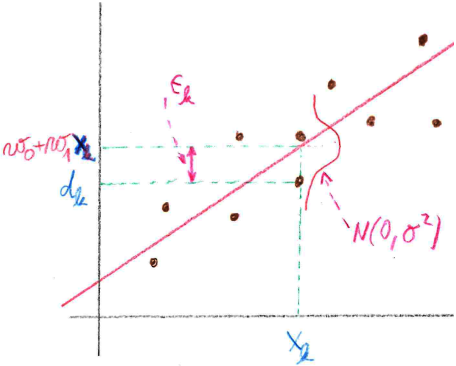
$$f_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$$

where w_0, w_1 are **unknown weights**, and ϵ_k are independent, normally distributed noise values with mean 0 and some variance σ^2



How "probable" is it to generate the correct $f_1, \ldots, f_p$?

Maximum Likelihood (GOOD STUDENTS)

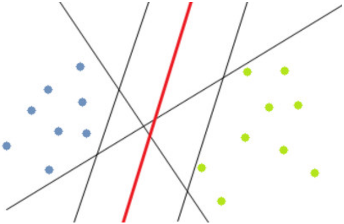


How "probable" is it to generate the correct $f_1, \ldots, f_p$?

The following conditions are equivalent:

- w_0, w_1 minimize the squared error E
- w_0, w_1 maximize the likelihood (i.e., the "probability") of generating the correct values $f_1, \ldots, f_p$ using $f_k = (w_0 + w_1 \cdot x_k) + \epsilon_k$

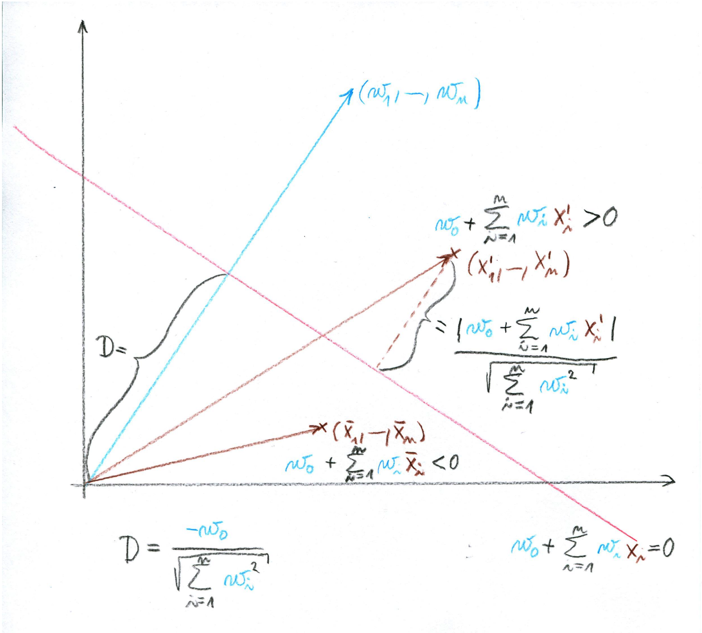
SVM Idea – Which Linear Classifier is the Best?



Benefits of maximum margin:

- Intuitively, maximum margin is good w.r.t. generalization.
- Only the *support vectors* (those on the magin) matter, others can, in principle, be ignored.

Linear Model – Geometry



Support Vector Machines (SVM)

Notation:

- $\vec{w} = (w_0, w_1, \ldots, w_n)$ a vector of weights,
- $\underline{\vec{w}} = (w_1, \ldots, w_n)$ a vector of all weights except w_0,
- $\vec{x} = (x_1, \ldots, x_n)$ a (generic) feature vector.

Consider a linear classifier:

$$h[\vec{w}](\vec{x}) := \begin{cases} 1 & w_0 + \sum_{i=1}^{n} w_i \cdot x_i = w_0 + \underline{\vec{w}} \cdot \vec{x} \geq 0 \\ -1 & w_0 + \sum_{i=1}^{n} w_i \cdot x_i = w_0 + \underline{\vec{w}} \cdot \vec{x} < 0 \end{cases}$$

The *signed distance* of $\vec{x}$ from the decision boundary determined by $\vec{w}$ is

$$d[\vec{w}](\vec{x}) = \frac{w_0 + \underline{\vec{w}} \cdot \vec{x}_k}{\|\underline{\vec{w}}\|}$$

Here $\|\underline{\vec{w}}\| = \sqrt{\sum_{i=1}^{n} w_i^2}$ is the Euclidean norm of $\underline{\vec{w}}$.

$|d[\vec{w}](\vec{x})|$ is the distance of $\vec{x}$ from the decision boundary.
$d[\vec{w}](\vec{x})$ is positive for $\vec{x}$ on the side to which $\underline{\vec{w}}$ points and negative on the opposite side.

Support Vectors & Margin

- Given a training set

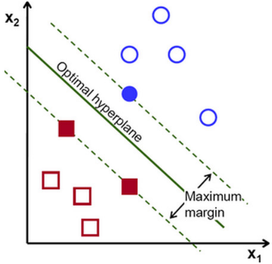
$$D = \{(\vec{x}_1, y(\vec{x}_1)), (\vec{x}_2, y(\vec{x}_2)), \ldots, (\vec{x}_p, y(\vec{x}_p))\}$$

Here $\vec{x}_k = (x_{k1} \ldots, x_{kn}) \in X \subseteq \mathbb{R}^n$ and $y(\vec{x}_k) \in \{-1, 1\}$.

We write y_k instead of $y(\vec{x}_k)$.

- Assume that D is linearly separable, let $\vec{w}$ *be consistent with* D.

- *Support vectors* are those $\vec{x}_k$ that minimize $|d[\vec{w}](\vec{x}_k)|$.
- *Margin* $\rho[\vec{w}]$ of $\vec{w}$ is twice the distance between support vectors and the decision boundary.



Our goal is to find $\vec{w}$ that maximizes the margin $\rho[\vec{w}]$.

Maximizing the Margin (GOOD STUDENTS)

For $\vec{w}$ consistent with D (such that no $\vec{x}_k$ lies on the decision boundary) we have

$$\rho[\vec{w}] = 2 \cdot \frac{|w_0 + \underline{\vec{w}} \cdot \vec{x}_k|}{\|\underline{\vec{w}}\|} = 2 \cdot \frac{y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k)}{\|\underline{\vec{w}}\|} > 0$$

where $\vec{x}_k$ is a support vector.

We may safely consider only $\vec{w}$ such that $y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) = 1$ for the support vectors.

Just adjust the length of $\vec{w}$ so that $y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) = 1$, the denominator $\|\underline{\vec{w}}\|$ will compensate.

Then maximizing $\rho[\vec{w}]$ is equivalent to maximizing $2/\|\underline{\vec{w}}\|$.

(In what follows we use a bit looser constraint:

$$y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) \geq 1 \text{ for all } \vec{x}_k$$

However, the result is the same since even with this looser condition, the support vectors always satisfy $y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) = 1$ whenever $2/\|\underline{w}\|$ is maximal.)

SVM – Optimization (BETTER STUDENTS)

Margin maximization can be formulated as a *quadratic optimization problem:*

Find $\vec{w} = (w_0, \ldots, w_n)$ such that

$$\rho = \frac{2}{\|\underline{\vec{w}}\|} \text{ is maximized}$$

and for all $(\vec{x}_k, y_k) \in D$ we have $y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) \geq 1$.

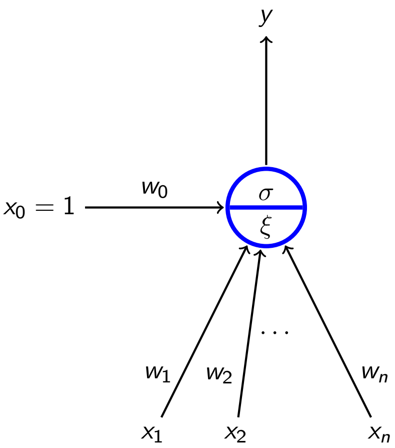
which can be reformulated as:

Find $\vec{w}$ such that

$$\Phi(\vec{w}) = \|\underline{\vec{w}}\|^2 = \underline{\vec{w}} \cdot \underline{\vec{w}} \text{ is minimized}$$

and for all $(\vec{x}_k, y_k) \in D$ we have $y_k \cdot (w_0 + \underline{\vec{w}} \cdot \vec{x}_k) \geq 1$.

Formal neuron

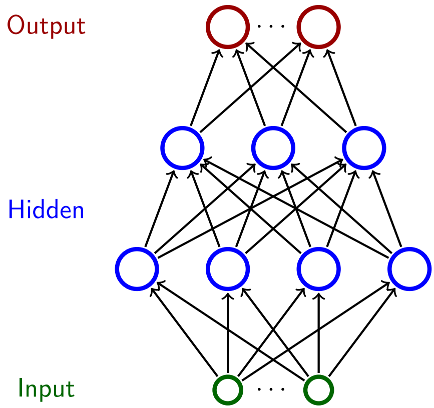


- $x_1, \ldots, x_n$ real *inputs*
- x_0 special input, always 1
- $w_0, w_1, \ldots, w_n$ real *weights*
- $\xi = w_0 + \sum_{i=1}^{n} w_i x_i$ *inner potential*;
  In general, other potentials are considered (e.g. Gaussian), more on this in PV021.
- y *output* defined by $y = \sigma(\xi)$ where σ is an *activation function*.
  We consider several activation functions.

  e.g., *linear threshold function*

$$\sigma(\xi) = sgn(\xi) = \begin{cases} 1 & \xi \geq 0\,; \\ 0 & \xi < 0. \end{cases}$$

Multilayer Perceptron (MLP)



- Neurons are organized in *layers* (input layer, output layer, possibly several hidden layers)
- Layers are numbered from 0; the input is 0-th
- Neurons in the ℓ-th layer are connected with all neurons in the $\ell + 1$-th layer

Intuition: The network computes a function as follows: Assign input values to the input neurons and 0 to the rest. Proceed upwards through the layers, one layer per step. In the ℓ-th step consider output values of neurons in $\ell - 1$-th layer as inputs to neurons of the ℓ-th layer. Compute output values of neurons in the ℓ-th layer.

Expressive Power of MLP

Cybenko's theorem:

- Two layer networks with a single output neuron and a single layer of hidden neurons (with the logistic sigmoid as the activation function) are able to
  - approximate with arbitrarily small error any "reasonable" boundary

    a given input is classified as 1 iff the output value of the network is $\geq 1/2$.
  - approximate with arbitrarily small error any "reasonable" function from $[0, 1]$ to $(0, 1)$.

  Here "reasonable" means that it is pretty tough to find a function that is not reasonable.

So multi-layer perceptrons are suffuciently powerful for any application.

But for a long time, at least throughout 60s and 70s, nobody well-known knew any efficient method for training multilayer networks!

... then an efficient way of using the gradient descent was published in 1986!

MLP – Notation

- X set of input neurons
- Y set of output neurons
- Z set of all neurons (tedy $X, Y \subseteq Z$)
- individual neurons are denoted by indices, e.g., i, j.
- ξ_j is the inner potential of the neuron j when the computation is finished.
- y_j is the output value of the neuron j when the computation is finished.

  (we formally assume $y_0 = 1$)
- w_{ji} is the weight of the arc **from** the neuron i **to** the neuron j.
- $j_{\leftarrow}$ is the set of all neurons from which there are edges to j
  (i.e. $j_{\leftarrow}$ is the layer directly below j)
- $j^{\rightarrow}$ is the set of all neurons to which there are edges from j.
  (i.e. $j^{\rightarrow}$ is the layer directly above j)

MLP – Notation

- Inner potential of a neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

- A value of a non-input neuron $j \in Z \setminus X$ when the computation is finished is

$$y_j = \sigma_j(\xi_j)$$

  Here σ_j is an activation function of the neuron j.

  (y_j is determined by weights $\vec{w}$ and a given input $\vec{x}$, so it's sometimes written as $y_j[\vec{w}](\vec{x})$)

- Fixing weights of all neurons, the network computes a function $F[\vec{w}] : \mathbb{R}^{|X|} \rightarrow \mathbb{R}^{|Y|}$ as follows: Assign values of a given vector $\vec{x} \in \mathbb{R}^{|X|}$ to the input neurons, evaluate the network, then $F[\vec{w}](\vec{x})$ is the vector of values of the output neurons.

  Here we implicitly assume a fixed orderings on input and output vectors.

MLP – Learning

- Given a set D of training examples:

$$D = \left\{ \left(\vec{x}_k, \vec{d}_k\right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

  Here $\vec{x}_k \in \mathbb{R}^{|X|}$ and $\vec{d}_k \in \mathbb{R}^{|Y|}$. We write d_{kj} to denote the value in $\vec{d}_k$ corresponding to the output neuron j.

- **Error Function:** $E(\vec{w})$ where $\vec{w}$ is a vector of all weights in the network. The choice of E depends on the solved task (classification vs regression etc.).
  Example (Squared error): $E(\vec{w}) = \sum_{k=1}^{p} E_k(\vec{w})$ where

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j[\vec{w}](\vec{x}_k) - d_{kj}\right)^2$$

  GOOD STUDENTS: Distinguish regression (identity output activation & squared error) and classification (logistic sigmoid output activation & cross-entropy error).

MLP – Batch Gradient Descent

The algorithm computes a sequence of weights $\vec{w}^{(0)}, \vec{w}^{(1)}, \ldots$.

- weights $\vec{w}^{(0)}$ are initialized randomly close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$) is $\vec{w}^{(t+1)}$ computed as follows:

  $$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

  where

  $$\Delta w_{ji}^{(t)} = -\varepsilon(t) \cdot \frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$$

  is the weight change w_{ji} and $0 < \varepsilon(t) \leq 1$ is the learning rate in the step $t+1$.

Note that $\frac{\partial E}{\partial w_{ji}}(\vec{w}^{(t)})$ is a component of ∇E, i.e. the weight change in the step $t+1$ can be written as follows: $\vec{w}^{(t+1)} = \vec{w}^{(t)} - \varepsilon(t) \cdot \nabla E(\vec{w}^{(t)})$.

MLP – Gradient Computation

For every weight w_{ji} we have (obviously)

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

So now it suffices to compute $\frac{\partial E_k}{\partial w_{ji}}$, that is the error for a fixed training example $(\vec{x}_k, d_k)$.

Applying the chain rule we obtain

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

where (more applications of the chain rule)

$\frac{\partial E_k}{\partial y_j}$ is computed directly for the output neurons $j \in Y$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj} \qquad \text{for } j \in Z \smallsetminus (Y \cup X)$$

(Here $y_r = y[\vec{w}](\vec{x}_k)$ where $\vec{w}$ are the current weights and $\vec{x}_k$ is the input of the k-th training example.)

Multilayer Perceptron – Backpropagation

Input: A training set $D = \left\{ \left(\vec{x}_k, \vec{d}_k\right) \quad \middle| \quad k = 1, \ldots, p \right\}$ and the current vector of weights $\vec{w}$.

Note that the backprop. is repeated in every iteration of the gradient descent!

- Evaluate all values y_i of neurons using the standard bottom-up procedure with the input $\vec{x}_k$.
- For every training example $(\vec{x}_k, \vec{d}_k)$ compute $\frac{\partial E_k}{\partial y_j}$ using *backpropagation* through layers top-down :
  - For all $j \in Y$ compute $\frac{\partial E_k}{\partial y_j}$ by taking the derivative of the error.
    e.g., in the case of the squared error we have $\frac{\partial E_k}{\partial y_j} = y_j - d_{kj}$.
  - In the layer ℓ, assuming that $\frac{\partial E_k}{\partial y_r}$ has been computed for all neurons r in the layer $\ell + 1$, compute
    $$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_j} \cdot \sigma_r'(\xi_r) \cdot w_{rj}$$
    for all j from the ℓ-th layer. Here σ_r' is the derivative of σ_r.
  - Put $\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$

Output: $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$.