Neural Networks
CS 550: Machine Learning
Classifiers and Discriminant Functions

(revisited)

- A classifier is represented with a set of discriminant functions $g_j(x)$ for $j = 1, 2, \ldots, C$

- A given instance $x$ is then classified with the class $C_j$ for which the discriminant function $g_j(x)$ is the maximum

1. Likelihood-based approaches

2. Discriminant-based approaches
Likelihood-Based Approaches

(revisited)

- They estimate class probabilities on the training samples and then use them to define the discriminant functions

\[
g_j(x) = \hat{P}(C_j \mid x) \\
= \frac{\hat{P}(x \mid C_j) \cdot \hat{P}(C_j)}{\hat{P}(x)} \\
= \hat{P}(x \mid C_j) \cdot \hat{P}(C_j)
\]

For each class, estimate the likelihood and the prior from the training samples that belong to this class
Discriminant-Based Approaches

(revisited)

- They learn discriminant functions directly on the training samples

- They make an assumption on the form of the discriminant functions and learn their parameters from the training samples without estimating the class probabilities

- **Linear discriminants assume that each discriminant function is a linear combination of the input features**
Linear Discriminants

- They define $g_j(x)$ as a linear combination of the input features

$$g_j(x \mid W_j) = \sum_{i=1}^{d} W_{ij} x_i + W_{0j}$$

Number of the input dimensions

Weight vector for the $j$-th class

Let's define $x_0 = 1$

$$g_j(x \mid W_j) = \sum_{i=0}^{d} W_{ij} x_i$$

Learning involves learning the parameters (weights) $W_j$ for each class $C_j$ from the training samples.

For that, we will define a criterion function and learn the weights that minimize/maximize this function.
Linear Discriminants

- They yield hyperplane decision boundaries
- Consider the 2-class classification problem

\[ g_1(x) = \sum_{i=1}^{d} W_{i1} x_i + W_{01} \]
\[ g_1(x) = W_1^T x + W_{01} \]
\[ g_2(x) = W_2^T x + W_{02} \]
\[ g(x) = g_1(x) - g_2(x) \]

choose \( C_1 \) if \( g(x) \geq 0 \)
choose \( C_2 \) otherwise

\[ g(x) = (W_1 - W_2)^T x + (W_{01} - W_{02}) \]
\[ g(x) = W^T x + W_0 \]

\( \text{This is another linear function} \)
Let's take two points on the decision plane

\[ g(x^a) = g(x^b) \]

\[ W^T x^a + W_0 = W^T x^b + W_0 \]

\[ W^T (x^a - x^b) = 0 \]

*W* determines the hyperplane’s orientation (\(W\) is normal to any vector on the hyperplane)

*W*\(_0\) determines the hyperplane’s location with respect to the origin.
Linear Discriminants

We consider the multiclass classification as

1. one-against-one     OR     2. one-against-all

\[
g_{kj}(x) = \begin{cases} 
\geq 0 & \text{if } x \in C_k \\
< 0 & \text{if } x \notin C_j \\
\text{don't care otherwise}
\end{cases}
\]

\[
g_k(x) = \begin{cases} 
\geq 0 & \text{if } x \in C_k \\
< 0 & \text{if } x \notin C_k
\end{cases}
\]

To resolve ambiguities, we may assign \(x\) to the class for which the discriminant is highest (it is also possible to reject classification or combine the discriminants in a different way).
How to Learn

Although we will use linear discriminants for classification, let’s first consider a linear regression problem.

Construct a linear model on the following data points:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3.1</td>
</tr>
<tr>
<td>4</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Construct a linear model:

\[ f(x) = W x + W_0 \]

Define a criterion function:

\[ \text{loss}(W, W_0) = \frac{1}{2} \sum_t \left( f(x^t) - y^t \right)^2 \]

Sum of squared errors

Select \( W \) and \( W_0 \) that minimize this error on the training samples:

\[ \frac{\partial \text{loss}}{\partial W} = 0 \quad \frac{\partial \text{loss}}{\partial W_0} = 0 \]
Analytical Solution

\[ \text{loss}(W, W_0) = \frac{1}{2} \sum_t \left( f(x^t) - y^t \right)^2 \]
\[ f(x) = W x + W_0 \]

\[ \frac{\partial \text{loss}}{\partial W} = \frac{1}{2} 2 \sum_t \left( W x^t + W_0 - y^t \right) x^t = W \sum_t x^{t^2} + W_0 \sum_t x^t + \sum_t x^t y^t = 0 \]

\[ \frac{\partial \text{loss}}{\partial W_0} = \frac{1}{2} 2 \sum_t \left( W x^t + W_0 - y^t \right) = W \sum_t x^t + W_0 N + \sum_t y^t = 0 \]

**In our example**

\[ \sum x^t = 10 \]
\[ \sum y^t = 9.2 \]
\[ \sum x^{t^2} = 30 \]
\[ \sum x^t y^t = 26.1 \]
\[ N = 4 \]

\[ 30 W + 10 W_0 - 26.1 = 0 \]
\[ 10 W + 4 W_0 - 9.2 = 0 \]

\[ \begin{cases} 2 \text{ unknowns} & \Rightarrow \begin{cases} W = 0.62 \\ W_0 = 0.75 \end{cases} \\ 2 \text{ equations} \end{cases} \]

\[ f(x) = 0.62 x + 0.75 \]

In many cases, there is no analytical solution (If the linear system has a singular matrix, no solution or multiple solutions exist)

**ITERATIVE OPTIMIZATION METHODS**
Gradient Descent Algorithm

- One commonly used iterative optimization method
- Goal is to find the parameters that minimize the loss
  - Starting with random parameters, it iteratively updates them in the direction of the steepest descent (in the opposite direction of the gradient) until the gradient is zero (or small enough)

\[
\begin{align*}
\text{start with random weights } W_i \\
do \\
\Delta W_i &= -\eta \frac{\partial \text{loss}}{\partial W_i} \quad \text{for all } i \\
W_i &= W_i + \Delta W_i \quad \text{for all } i \\
\text{until convergence}
\end{align*}
\]

- It finds the nearest minimum, which could be local
- It does not guarantee to find the global minimum

\(\eta\) is the learning rate, which determines how much to move in the direction of the steepest descent

→ if it is too small, convergence is slow
→ if it is too large, we may overshoot the minimum (divergence might occur)
Regression

Let’s derive the update rules for regression

\[
\Delta W_i = -\eta \frac{\partial \text{loss}_{\text{ALL}}(W)}{\partial W_i}
\]

\[
\frac{\partial \text{loss}}{\partial W_i} = \frac{\partial \text{loss}}{\partial \text{net}} \frac{\partial \text{net}}{\partial W_i}
\]

\[
\frac{\partial \text{loss}}{\partial W_i} = \delta x_i
\]

\[
\delta = (\text{net} - y)
\]

\[
\Delta W_i = -\eta \sum_t \delta x_i
\]

\[
\Delta W_i = \eta \sum_t (y^t - \text{net}^t) x'^t_i
\]
Classification (Logistic Regression)

Let’s derive the update rules for 2-class classification

\[ y = 1 \text{ if } x \in C_1 \]
\[ y = 0 \text{ if } x \in C_2 \]

\[ f(x) = \sigma(\text{net}) \]

\[ \text{net} = \sum_i x_i W_i \]

\[ \sigma(\text{net}) = \frac{1}{1 + \exp(-\text{net})} \]

\[ \sigma'(\text{net}) = \sigma(\text{net}) (1 - \sigma(\text{net})) \]

\[ f(x) = a \tanh(b \cdot x) = a \left[ \frac{\exp(b \cdot x) - \exp(-b \cdot x)}{\exp(b \cdot x) + \exp(-b \cdot x)} \right] \]

Logarithmic sigmoid function

Hyperbolic tangent sigmoid function
Classification (Logistic Regression)

Let’s derive the update rules for 2-class classification

$$f(x) = \sigma(\text{net})$$
$$\text{net} = \sum_i x_i W_i$$

$$\frac{\partial \text{loss}}{\partial W_i} = \frac{\partial \text{loss}}{\partial \text{net}} \frac{\partial \text{net}}{\partial W_i}$$

When squared error is used

$$\delta = (\sigma(\text{net}) - y) \sigma'(\text{net})$$

$$\Delta W_i = \eta \sum_i \left( y^t - \sigma(\text{net}^t) \right) \sigma(\text{net}^t) (1 - \sigma(\text{net}^t)) x^t_i$$
Classification

Let’s derive the update rules for multiclass classification

\[ y_j = 1 \text{ if } x \in C_j \]
\[ y_j = 0 \text{ if } x \notin C_j \]

Define output as a C-dimensional vector

\[ f_j(x) = \text{softmax}(\text{net}_j) \]
\[ \text{net}_j = \sum_i x_i W_{ij} \]

\[ \text{softmax}(\text{net}_j) = \frac{\exp(\text{net}_j)}{\sum_m \exp(\text{net}_m)} \]

\[ \frac{\partial \text{softmax}(\text{net}_k)}{\partial \text{net}_j} = \text{softmax}(\text{net}_j) \left( \delta_{jk} - \text{softmax}(\text{net}_k) \right) \]

\[ \delta_{jk} = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases} \]

Kronecker delta
Classification

Let’s derive the update rules for multiclass classification

\[ f_j(x) = \text{softmax}(\text{net}_j) \]

\[ \text{net}_j = \sum_i x_i W_{ij} \]

\[ \frac{\partial \text{loss}}{\partial W_{ij}} = \frac{\partial \text{loss}}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial W_{ij}} \]

\[ \frac{\partial \text{loss}}{\partial \text{net}_j} = \delta_j x_i \]

When squared error is used

\[ \delta_j = \sum_k \left( \text{softmax}(\text{net}_k) - y_k \right) \frac{\partial \text{softmax}(\text{net}_k)}{\partial \text{net}_j} \]

\[ \Delta W_{ij} = \eta \sum_t \sum_k \left( y_k^t - s(\text{net}_k^t) \right) s(\text{net}_j^t) \left( \delta_{jk}^t - s(\text{net}_k^t) \right) x_i^t \]

Cross entropy

\[ \text{loss} = -\sum_k y_k \log f_k(x) \]

Squared error

\[ \text{loss} = \frac{1}{2} \sum_k \left( f_k(x) - y_k \right)^2 \]
start with random weights $W_{ij}$
do
  compute $f_j(x^t) = \text{softmax}\left(\sum_i x^t_i W_{ij}\right)$ for all $t$ and $j$
  compute $\Delta W_{ij} = -\eta \sum_t \delta_j x^t_i$ for all $i$ and $j$
  update $W_{ij} = W_{ij} + \Delta W_{ij}$ for all $i$ and $j$
until convergence

**Batch learning algorithm**

**Stochastic learning algorithm**

Mini-batch stochastic learning algorithm is a good tradeoff
Batch or Stochastic Learning?

- **Batch learning allows using** some second-order techniques that
  - Cannot be easily incorporated into stochastic learning

- **Stochastic training is preferred** for most applications
  - Especially when datasets are highly redundant
  - It is typically faster than batch training

- Mini-batch stochastic learning is a good tradeoff
Adding Nonlinearity

- Linear discriminants yield hyperplane decision boundaries
- If they are not sufficient to construct a “good” model
  1. We may transform the space into a new one using nonlinear mappings and construct linear discriminants on the transformed space \( \rightarrow \text{SVMs} \)
  2. We may learn the nonlinearity at the same time as the linear discriminants \( \rightarrow \text{ANNs} \)
XOR problem

Support vector machines use the idea of nonlinear mapping to find a linearly separable space.
XOR problem

\[ net_1 = W_{12} \cdot x_2 + W_{11} \cdot x_1 + W_{10} \]
\[ h_1 = \text{sign}(net_1) \]

\[ net_2 = W_{22} \cdot x_2 + W_{21} \cdot x_1 + W_{20} \]
\[ h_2 = \text{sign}(net_2) \]

Neural networks learn the nonlinearity at the same time as the linear discriminants (learn all the weights at the same time)
Multilayer Perceptrons

- Also contain hidden layers in addition to input and output layers

In this network
1. Each hidden unit computes its net activation
   \[ net_j = \sum_i x_i W_{ij} \]
2. Each hidden unit emits an output that is a nonlinear function of its activation
   \[ h_j = \sigma(net_j) \]
3. Each output unit computes its net activation
   \[ net_k = \sum_j h_j W_{jk} \]
4. Each output unit emits an output
   \[ y_k = g(net_k) \]

Hidden units \( h_j \)'s can be viewed as new "features" obtained by combining \( x_i \)'s

A deeper architecture with nonlinear activations is more expressive than a shallow one
How to Learn?

- In linear discriminants, we select the weights to minimize a loss function defined on the difference between the actual and computed output values.

- In multilayer structures, we can also select the hidden-to-output-layer weights to minimize a loss function defined on the actual and computed output values.

- However, we cannot select the input-to-hidden-layer weights in a similar way since we do not know the actual values of the hidden units.

- Thus, to learn the input-to-hidden-layer weights, we propagate the loss function (defined on the output values) from the output layer to the corresponding hidden layer. → BACKPROPAGATION ALGORITHM
Backpropagation Algorithm

Let’s derive the update rules for multiclass classification

\[
\text{net}_j = \sum_i x_i W_{ij} \\
h_j = \sigma(\text{net}_j) \\
\text{net}_k = \sum_j h_j W_{jk} \\
f_k(x) = \text{softmax}(\text{net}_k) \\
\text{loss}_{\text{ALL}}(W) = \sum_t \text{loss}'_t \\
\text{loss} = \frac{1}{2} \sum_k \left(f_k(x) - y_k\right)^2
\]

\[
\frac{\partial \text{loss}}{\partial W_{jk}} = \frac{\partial \text{loss}}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial W_{jk}} \\
\frac{\partial \text{loss}}{\partial W_{jk}} = \delta_k h_j \\
\delta_k = \sum_m \left(\text{softmax}(\text{net}_m) - y_m\right) \frac{\partial \text{softmax}(\text{net}_m)}{\partial \text{net}_k}
\]

Hidden-to-output-layer weights
Backpropagation Algorithm

Let’s derive the update rules for multiclass classification

\[
\begin{align*}
\text{net}_j &= \sum_i x_i W_{ij} \\
\text{net}_k &= \sum_j h_j W_{jk} \\
\text{loss}_{\text{ALL}}(W) &= \sum_t \text{loss}' \\
\text{loss} &= \frac{1}{2} \sum_k \left( f_k(x) - y_k \right)^2 \\
\Delta W_{ij} &= -\eta \frac{\partial \text{loss}_{\text{ALL}}(W)}{\partial W_{ij}} \\

\frac{\partial \text{loss}}{\partial W_{ij}} &= \frac{\partial \text{loss}}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial W_{ij}} \\
\frac{\partial \text{loss}}{\partial W_{ij}} &= \delta_j x_i \\
\delta_j &= \sum_k \frac{\partial \text{loss}}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} = \left[ \sum_k \delta_k W_{jk} \right] \sigma'(\text{net}_j)
\end{align*}
\]

Squared error

Exercise: Derive the update rules for regression
More hidden layers

\[ \delta_j \text{ may vanish after repeated multiplication} \]
This makes deep architectures hard to train
(when the initial values of the weights are not “good” enough)

\[ \frac{\partial \text{loss}}{\partial W_{ij}} = \frac{\partial \text{loss}}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial W_{ij}} \]
\[ \frac{\partial \text{loss}}{\partial W_{ij}} = \delta_j x_i \]
\[ \delta_j = \sum_{(j+1)} \frac{\partial \text{loss}}{\partial \text{net}_{(j+1)}} \frac{\partial \text{net}_{(j+1)}}{\partial \text{net}_j} \]
\[ \delta_j = \left[ \sum_{(j+1)} \delta_{(j+1)} W_{j(j+1)} \right] \sigma'(\text{net}_j) \]
Activation function

A number of properties we seek for activation function $\sigma(.)$

- $\sigma(.)$ must be **nonlinear**
  - Otherwise a hidden layer provides no further computational power

- $\sigma(.)$ must be **saturate** (it has min and max values)
  - This will keep the weights and activations bounded
  - It is also desirable for classification
  - Of course, it may not be desirable in networks used for regression

- $\sigma(.)$ and $\sigma'(.)$ must be **defined throughout the range of their argument** (for backpropagation algorithm)

- $\sigma(.)$ should be **linear for small values of net activation**
  - This will enable the system to implement a linear model if it is adequate for low error
Activation function

- One class of such functions is sigmoid
  - Logarithmic sigmoid function
    \[ f(x) = \frac{1}{1 + \exp(-x)} \]
  - Hyperbolic tangent sigmoid function
    \[ f(x) = a \tanh(b \cdot x) = a \left[ \frac{\exp(b \cdot x) - \exp(-b \cdot x)}{\exp(b \cdot x) + \exp(-b \cdot x)} \right] \]
Target output values

- In classification, a target value can be represented by
  - 0/1 target values
    - Outputs represent posterior probabilities
    - Using the softmax function, the maximum output is transformed towards 1.0 and all others reduced to 0.0
  - -1/+1 target values
    - Outputs do not represent posterior properties

- A proper activation function should be used in the output layer
  - Depending on the selected target value representation
When to Stop

- When the change in the loss function is smaller than some preset value $\theta$

$$\left\| \nabla \text{loss}_{ALL}(W) \right\| \leq \theta$$

- When a minimum is reached on the validation set
  
  - Training error ultimately reaches an asymptotic value
  
  - The error on an independent test set is virtually always higher
    - Although it usually decreases, it can also increase or oscillate
Initializing Weights

- We cannot initialize the weights to zero
- We want to have uniform learning
  - All weights reach their final equilibrium at about the same time
  - For that, with standardized data, we choose weights randomly from a uniform distribution $-\omega < w < \omega$
  - When the sigmoid function is used (for calculating an output and/or for defining hidden units in MLPs)
    - If $\omega$ is chosen too small, the net will be too small
    - If $\omega$ is chosen too large, sigmoid may saturate even before learning begins
    - Set $\omega$ such that sigmoid is in its linear range
Learning Rate

- In principle, if the learning rate is small enough, it ensures the convergence
  - Its value determines only the speed
  - Not the final weight values

- In practice, the learning rate can indeed affect the quality of the final network
  - Since networks are not fully trained most of the time
Learning Rate

- The optimal learning rate leads to the local minimum in one step
- The optimal rate is found as
  \[ \eta_{opt} = \left( \frac{\partial^2 \text{loss}_{ALL}(W)}{\partial W^2} \right)^{-1} \]
- The system converges for \( \eta < \eta_{opt} \) and \( \eta_{opt} < \eta < 2\eta_{opt} \)
  - But training is needlessly slow
- It is found that the system diverge if \( \eta > 2\eta_{opt} \)
Learning Rate

- Thus, in order to have rapid and uniform learning
  - For each weight, calculate $\frac{\partial^2 \text{loss}_{\text{ALL}}(W)}{\partial W^2}$ and set the optimal learning rate separately (not so much practical)

- For typical networks that use sigmoid functions
  - $\eta=0.1$ is a good choice to start with
    - It should be lowered if the loss function diverges
    - It should be raised if learning seems unduly slow

- During training, it is also possible to change the learning rate $\eta$ as a function of time (epoch number)
Regularization

- Adding regularization term to the loss function reduces sensitivity to training samples and decreases the risk of overfitting.

\[
loss_{ALL}(W) = \sum_t loss^t + \|W\|
\]

\[
loss_{ALL}(W) = \frac{1}{2} \sum_t \sum_k \left( f_k(x^t) - y_k^t \right)^2 + \|W\|
\]

- Sum of the squared error
- Regularization term
Momentum

- Error surfaces often have plateaus
  - Regions in which the derivative is very small
  - Such plateaus may arise especially when there are too many weights such that the loss function only weakly depends on any of them

- Momentum allows to learn more quickly when there are such plateaus
Momentum

- For stochastic learning algorithm, we include some fraction of the previous weight updates into the learning rule

\[
\begin{align*}
    w^{(t+1)} &= w^{(t)} + (1 - \alpha) \Delta w^{(t)} + \alpha \Delta w^{(t-1)}
\end{align*}
\]

- Parameter \( \alpha \) should be nonnegative and less than 1
- If \( \alpha = 0 \), it is the same as the standard gradient descent
- If \( \alpha = 1 \), the weight vector moves with constant velocity
- Values typically used are \( \alpha \approx 0.9 \)

- The use of momentum increases stability
  - Thus, it can speed up the learning process
Network topology

The number of hidden units

- It controls the expressive power of the network
  - Thus, the complexity of the decision boundary

- There is no foolproof method to set the number of hidden units before training
  - If samples are well-separated
    - Few hidden units are enough
  - If samples have complicated densities
    - More hidden units may be necessary
Network topology

- If too much hidden units,
  - The network is tuned to the particular dataset (overfitting)
    - Training error can become small, but test error is unacceptably high
- If too few hidden units,
  - The network does not have enough free parameters to fit the training data well
    - Training and test errors are high