Visual Computing Institute
RWTH AACHEN UNIVERSITY

Elements of Machine Learning & Data Science

Winter semester 2023/24

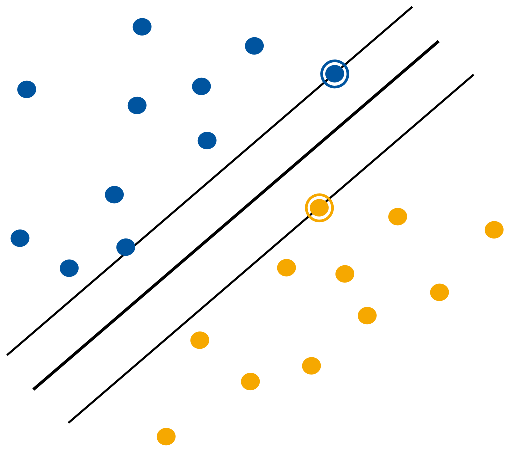
Lecture 19 – Neural Networks Basics

19.12.2023

Prof. Bastian Leibe

Machine Learning Topics

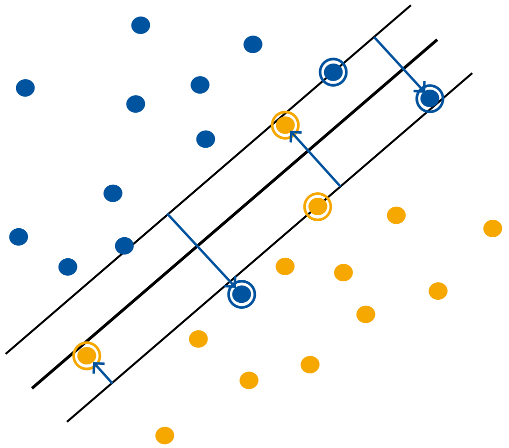
1. Introduction to ML
2. Probability Density Estimation
3. Linear Discriminants
4. Linear Regression
5. Logistic Regression
6. **Support Vector Machines**
7. (AdaBoost)
8. Neural Network Basics



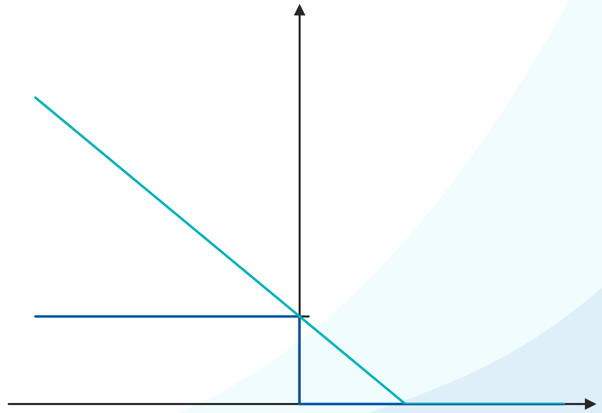
Maximum Margin Classification

$$L_p(\mathbf{w}, b, \mathbf{a})$$
$$L_d(\mathbf{a})$$

Primal & Dual Form



Soft-Margin SVM



Hinge Loss

Recap: Soft-Margin SVM

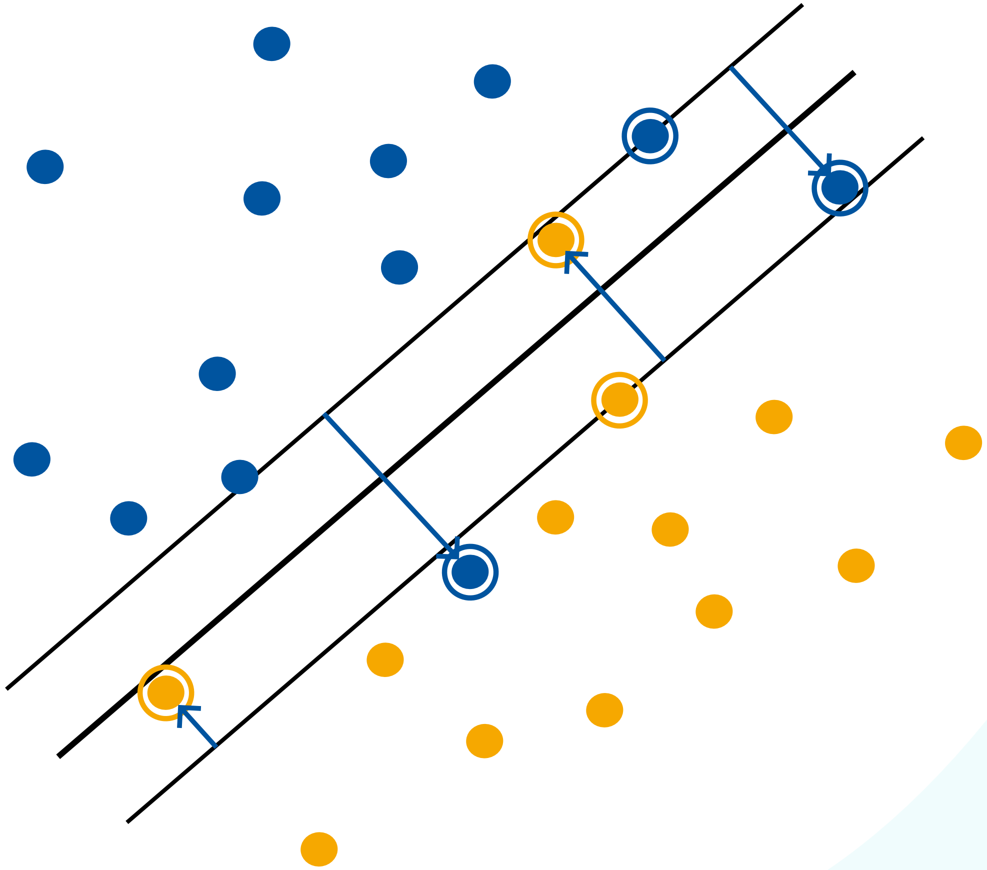
- Idea: Introduce slack variables $\xi_n \geq 0$
  - One slack variable ξ_n for each training point

- Effect
  - $\xi_n = 0$ for points on the correct side.
  - Linear penalty for all other points:
    $\xi_n = |t_n - y(\mathbf{x}_n)|$

- Slack variables are jointly optimized with $\mathbf{w}$:

$$\min \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{n=1}^{N}\xi_n$$

where C is a tradeoff parameter.



Recap: Soft-Margin SVM Primal Form

- Minimize

$$L_p = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{n=1}^{N}\xi_n - \underbrace{\sum_{n=1}^{N} a_n \left[t_n(y(\mathbf{x}_n) - 1 + \xi_n\right]}_{\text{Constraint}} - \underbrace{\sum_{n=1}^{N}\mu_n\xi_n}_{\text{Constraint}}$$

Constraint $t_n y(\mathbf{x}_n) \geq 1 - \xi_n$

Constraint $\xi_n \geq 0$

- KKT conditions

$$\begin{aligned} a_n &\geq 0 & \mu_n &\geq 0 \\ t_n y(\mathbf{x}_n) - 1 + \xi_n &\geq 0 & \xi_n &\geq 0 \\ a_n \left[t_n y(\mathbf{x}_n) - 1 + \xi_n\right] &= 0 & \mu_n\xi_n &= 0 \end{aligned}$$

Recap: Soft-Margin SVM Dual Form

- Maximize

$$L_d(\mathbf{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} a_n a_m t_n t_m (\mathbf{x}_m^{\mathsf{T}} \mathbf{x}_n)$$

- Under the side conditions

$$0 \leq a_n \leq C \quad \forall n$$

$$\sum_{n=1}^{N} a_n t_n = 0$$

This is the only difference to before.

Recap: Non-linear SVM Formulation

- Maximize

$$L_d(\mathbf{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} a_n a_m t_n t_m k(\mathbf{x}_m, \mathbf{x}_n)$$

under the constraints $\quad 0 \leq a_n \leq C \quad \forall n$

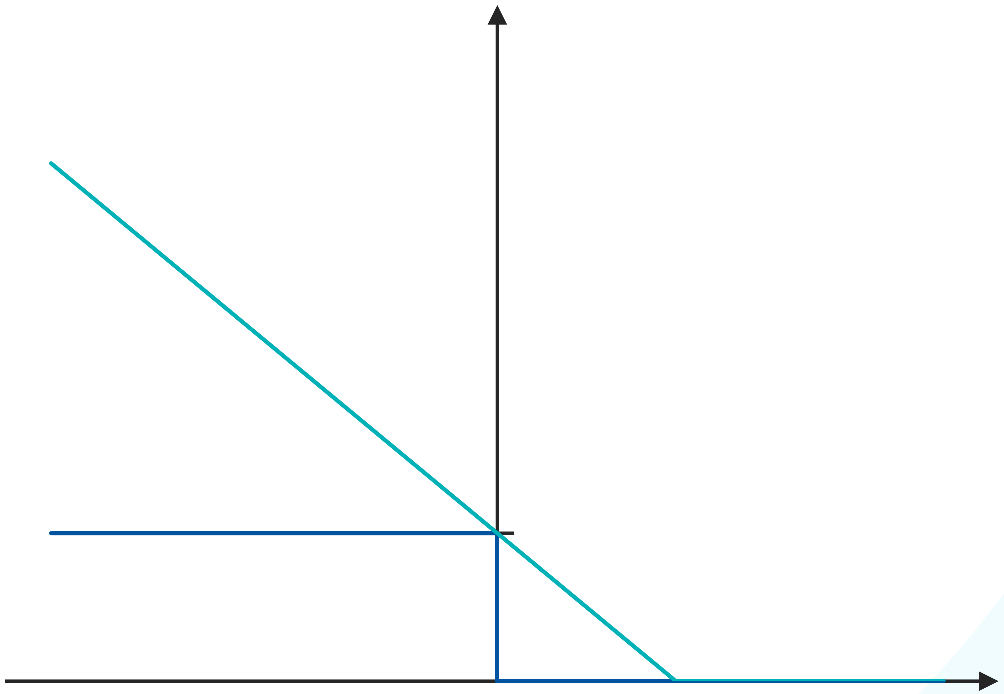
$$\sum_{n=1}^{N} a_n t_n = 0$$

- Classify new data points using

$$y(\mathbf{x}) = \sum_{n=1}^{N} a_n t_n k(\mathbf{x}_n, \mathbf{x}) + b$$

Support Vector Machines

1. Maximum Margin Classification
2. Primal Formulation
3. Dual Formulation
4. Soft-Margin SVMs
5. Non-linear SVMs
6. **Error Function Analysis**



Error Function Analysis

- We know how to formulate and optimize an SVM as a convex optimization problem:

$$\underset{\mathbf{w},\, b,\, \xi_n \in \mathbb{R}^+}{\arg\min} \; \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{n=1}^{N} \xi_n$$

subject to the constraints

$$t_n y(\mathbf{x}_n) \geq 1 - \xi_n$$

Error Function Analysis

- We know how to formulate and optimize an SVM as a convex optimization problem:

$$\underset{\mathbf{w},\, b,\, \xi_n \in \mathbb{R}^+}{\arg\min} \; \frac{1}{2}\|\mathbf{w}\|^2 + C \sum_{n=1}^{N} \xi_n$$

subject to the constraints

$$t_n y(\mathbf{x}_n) \geq 1 - \xi_n$$

But what error function does this correspond to?

- Integrate the constraints into the objective function:
  - Rewrite as $\xi_n \geq 1 - t_n y(\mathbf{x}_n)$
  - Thus, we obtain

$$\min_{\mathbf{w},\, b} E(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 + C \sum_{n=1}^{N} [1 - t_n y(\mathbf{x}_n)]_+$$

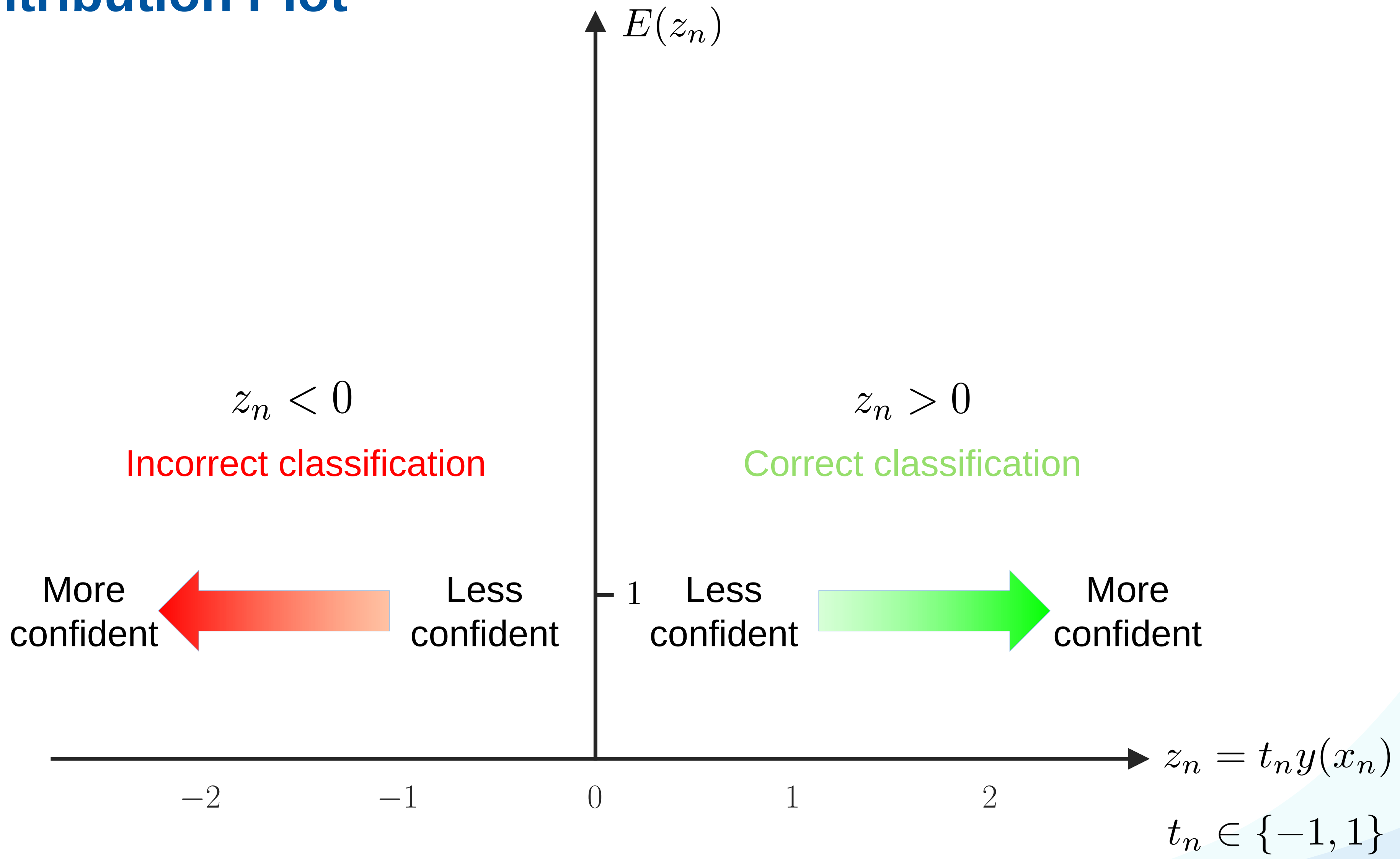
$$[x]_+ \equiv \max\{x, 0\}$$

The Hinge Loss

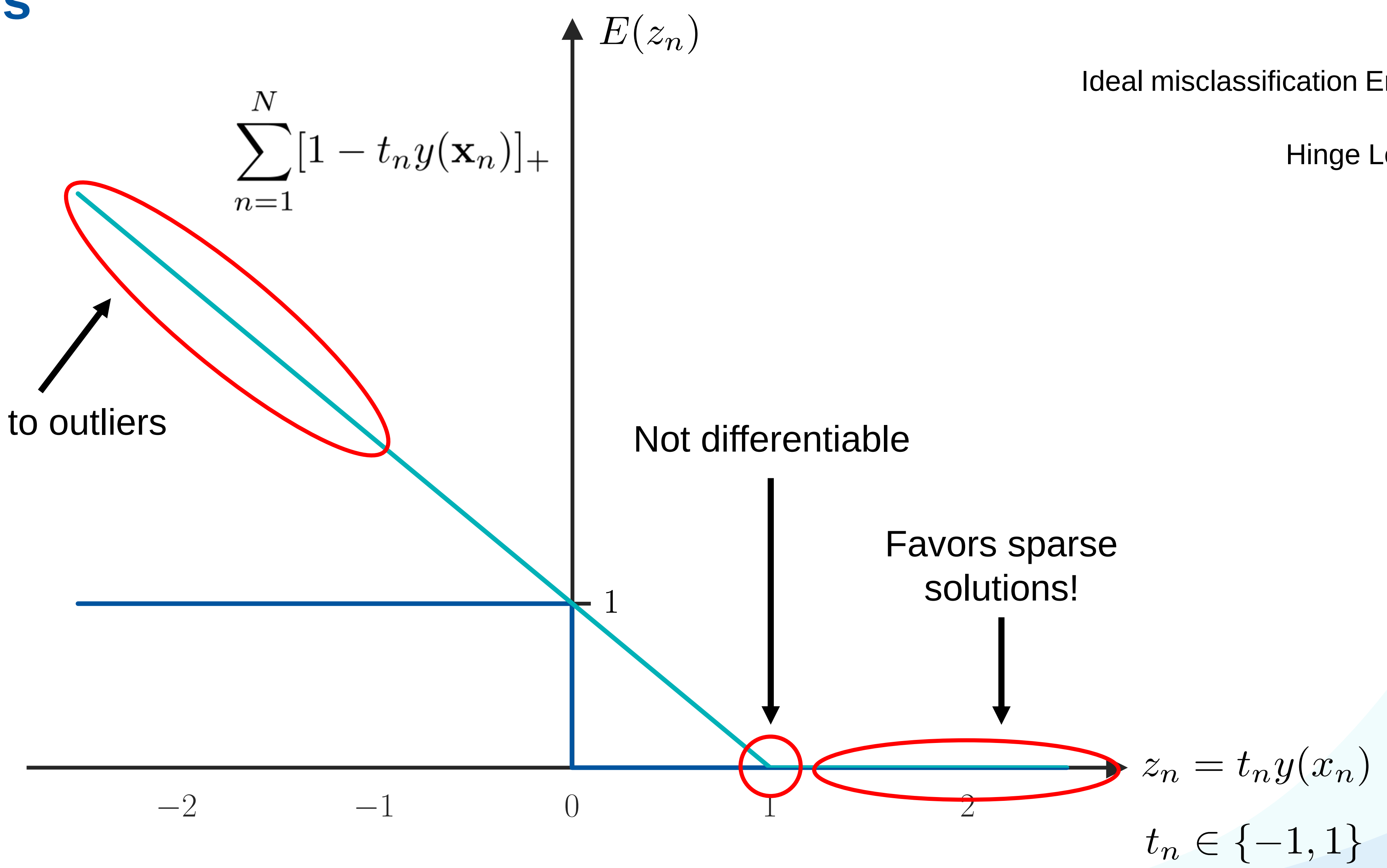
$$E(\mathbf{w}) = \underbrace{\frac{1}{2}\|\mathbf{w}\|^2}_{\text{L}_2 \text{ regularization}} + \underbrace{C\sum_{n=1}^{N}[1 - t_n y(\mathbf{x}_n)]_+}_{\text{Hinge loss}}$$

- Regularization bounds parameter size.
- Hinge Loss enforces sparsity:
  - Only a subset of training data points actually influences the decision boundary.
  - Still, all input dimensions are used.
- This formulation corresponds to an unconstrained optimization of a non-differentiable function.
  - Very efficient: stochastic (sub-)gradient descent.

Error Contribution Plot

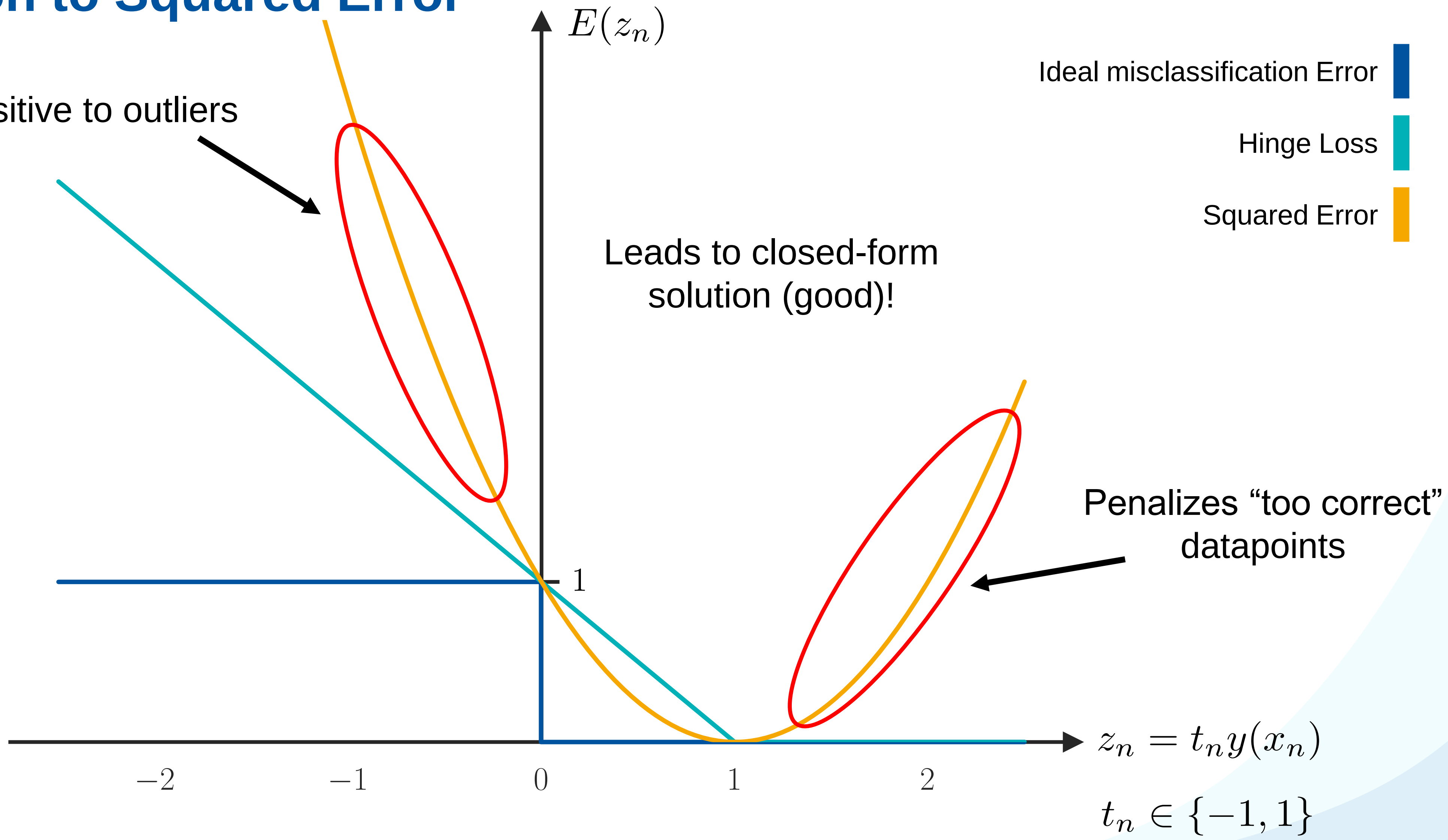


Hinge Loss

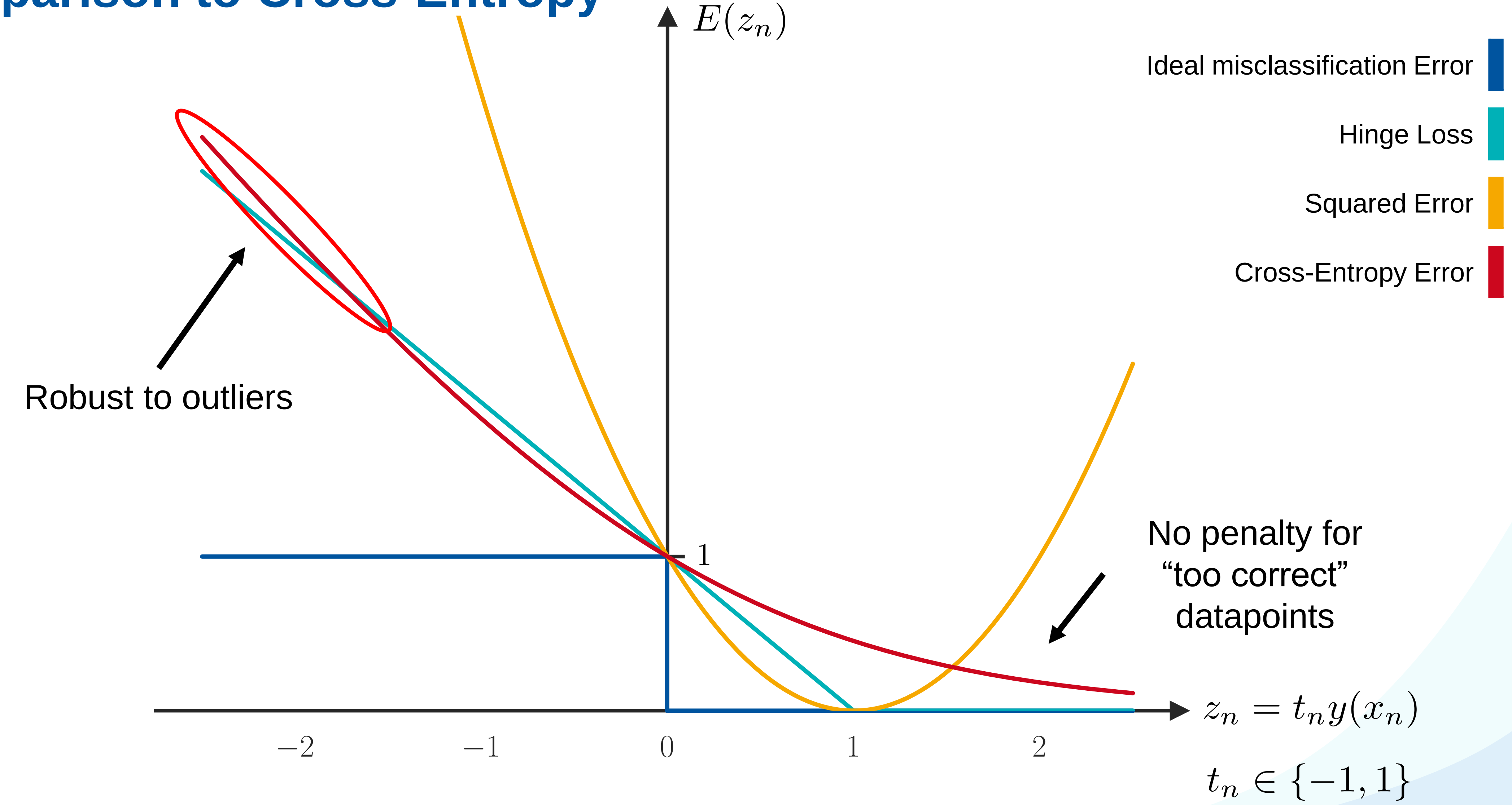


$E(z_n)$
Ideal misclassification Error
Hinge Loss
$\sum_{n=1}^{N}[1 - t_n y(\mathbf{x}_n)]_+$
Robust to outliers
Not differentiable
Favors sparse solutions!
1
-2
-1
0
1
2
$z_n = t_n y(x_n)$
$t_n \in \{-1, 1\}$

Comparison to Squared Error



Comparison to Cross-Entropy



Discussion: Hinge Loss

Advantages

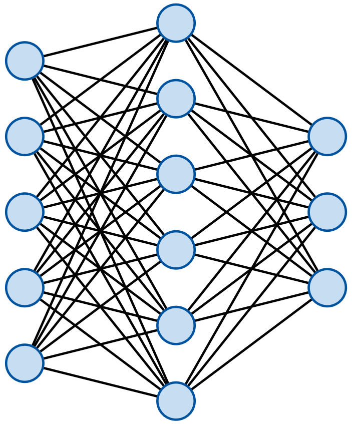
- Favors sparse solutions that only depend on a subset of training data points.
- Robust to outliers (only a linear penalty for misclassified points).
- Convex function, unique minimum exists.

Limitations

- Not differentiable (cannot minimize this loss using standard gradient descent, but need to use subgradient descent).

Machine Learning Topics

1. Introduction to ML
2. Probability Density Estimation
3. Linear Discriminants
4. Linear Regression
5. Logistic Regression
6. Support Vector Machines
7. (AdaBoost)
8. **Neural Network Basics**



Multi-Layer Perceptrons

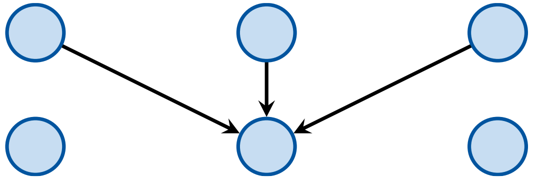
$$\frac{1}{2}(y(\mathbf{x})-t)^2$$

$$[1-ty(\mathbf{x})]_+$$

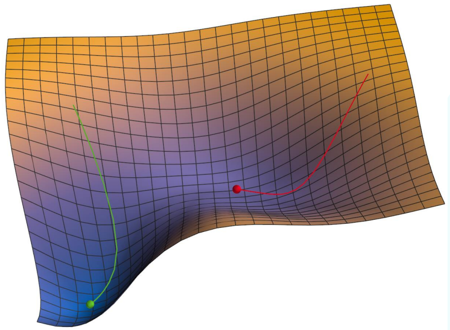
$$-\sum_k \left(\mathbb{I}(t=k)\ln \frac{\exp(y_k(\mathbf{x}))}{\sum_j \exp(y_j(\mathbf{x}))}\right)$$

$$\frac{1}{2}\|\mathbf{w}\|^2$$

Losses & Regularizers



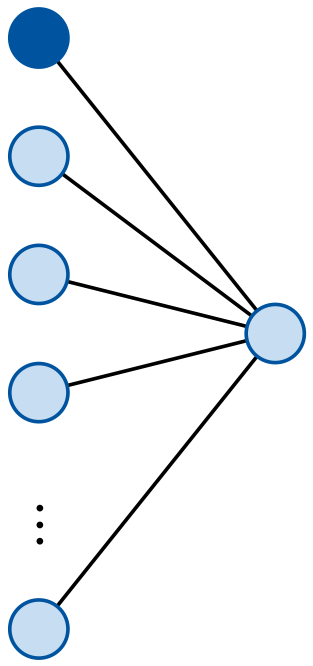
Backpropagation



Stochastic Gradient Descent

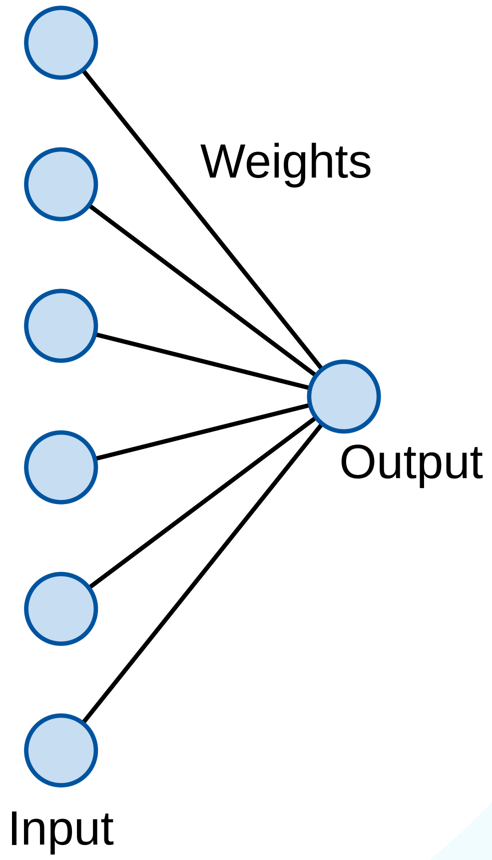
Neural Network Basics

1. **Perceptrons**
2. Multi-Layer Perceptrons
3. Loss Functions
4. Backpropagation
5. Stochastic Gradient Descent



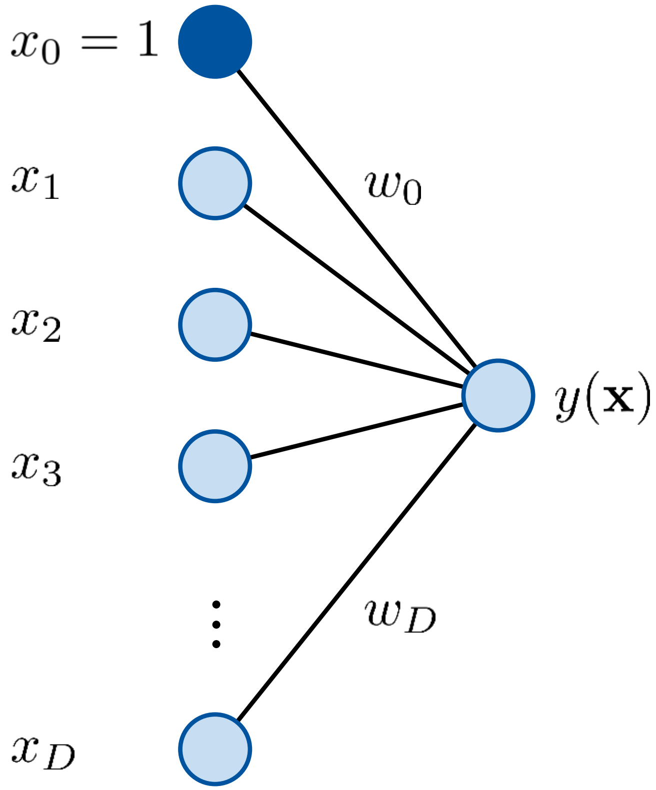
Perceptrons

- Inspired by biological neurons.
- The output is determined by the activation of the input nodes and a set of weights connecting input and output layers.



Basic Perceptron

- Input Layer:
  - Hand-designed features
- Outputs:
  - Linear outputs
    $$y(\mathbf{x}) = \mathbf{w}^\mathsf{T}\mathbf{x} + w_0$$
  - Logistic outputs
    $$y(\mathbf{x}) = \sigma(\mathbf{w}^\mathsf{T}\mathbf{x} + w_0)$$
- Learning: finding optimal weights $\mathbf{w}$.



Multi-Class Networks

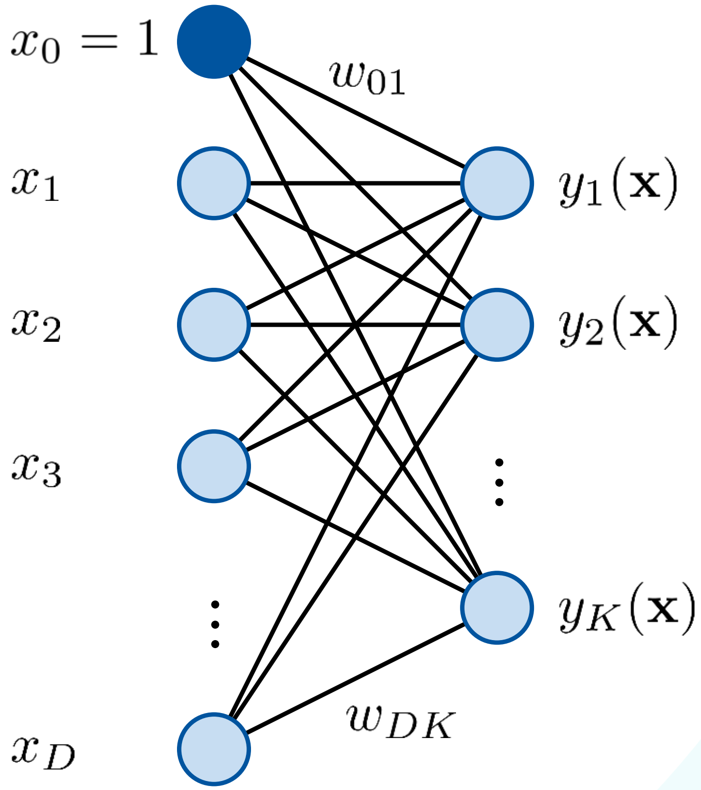
- One output node per class:
  - Linear outputs

$$y_k(\mathbf{x}) = \sum_{i=0}^{D} w_{ki}\mathbf{x}_i$$

  - Logistic outputs

$$y_k(\mathbf{x}) = \sigma\left(\sum_{i=0}^{D} w_{ki}\mathbf{x}_i\right)$$

- We can do multidimensional linear regression or multiclass classification this way.



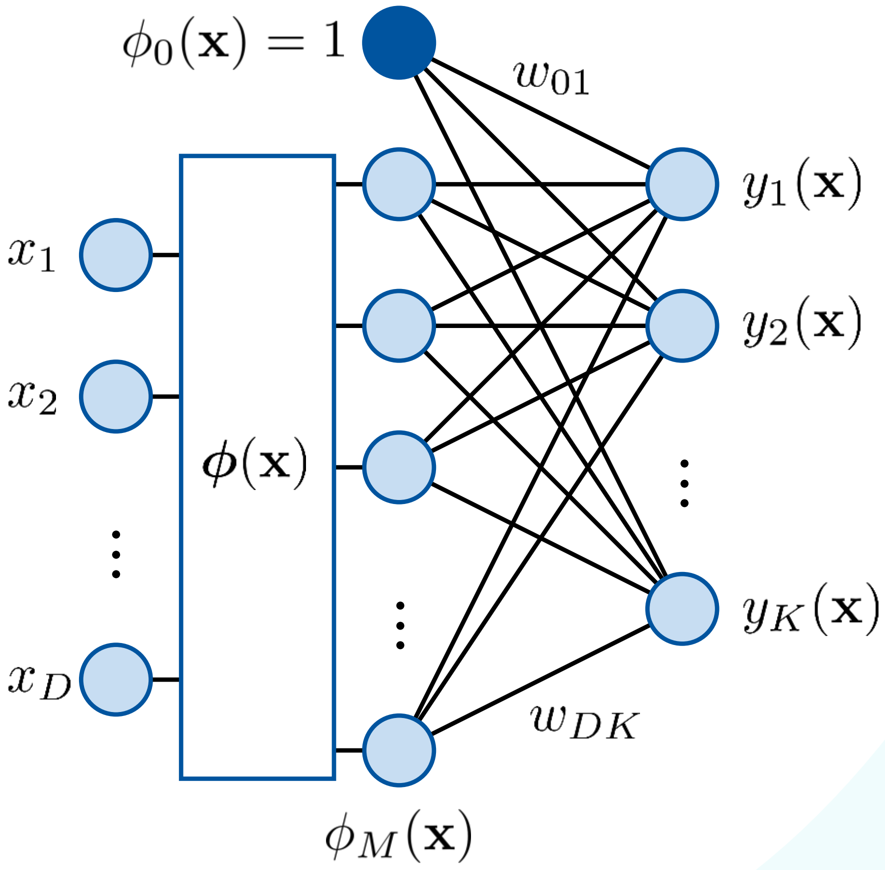
Non-Linear Basis Functions

- Apply a (fixed) mapping $\phi(\mathbf{x})$ to inputs:
  - Linear outputs

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x})$$

  - Logistic outputs

$$y_k(\mathbf{x}) = \sigma\left(\sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x})\right)$$



Connections to linear discriminants

- All of this should feel very familiar.
- Perceptrons are generalized linear discriminants!

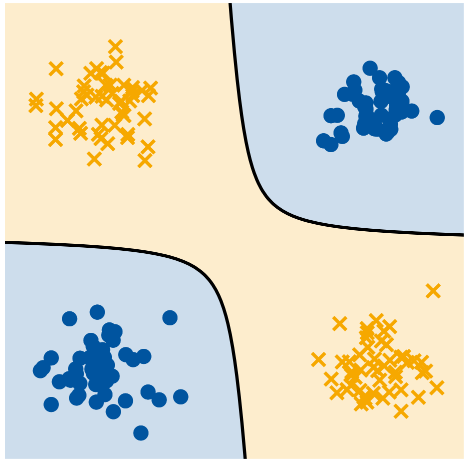
- What does that mean?
  - We have the same limitations as before.
  - Can model any separable function perfectly, given the right input features.
  - For some tasks, this may require an exponential number of input features.

$\Rightarrow$ *It is the feature design that solves the task!*

Limitations so far

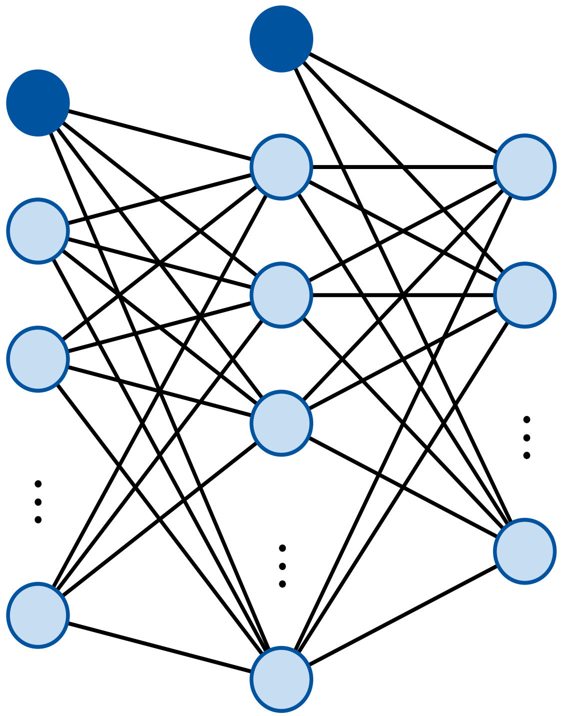
- Generalized linear discriminants (including perceptrons) are very limited.
  - A linear classifier cannot solve certain problems (e.g., XOR).
  - However, with a non-linear classifier based on suitable features, the problem becomes solvable.
  - So far, we have designed the features and kernels by hand.

$\Rightarrow$ *How can we learn good feature representations?*



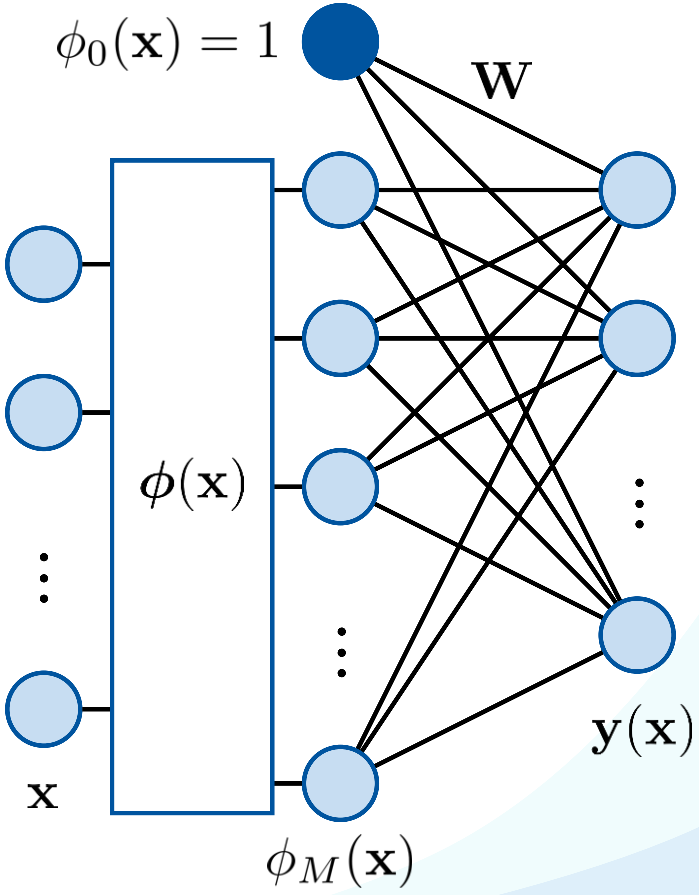
Neural Network Basics

1. Perceptrons
2. **Multi-Layer Perceptrons**
3. Loss Functions
4. Backpropagation
5. Stochastic Gradient Descent



Multi-Layer Perceptrons

- Perceptrons are limited by having a fixed input mapping.

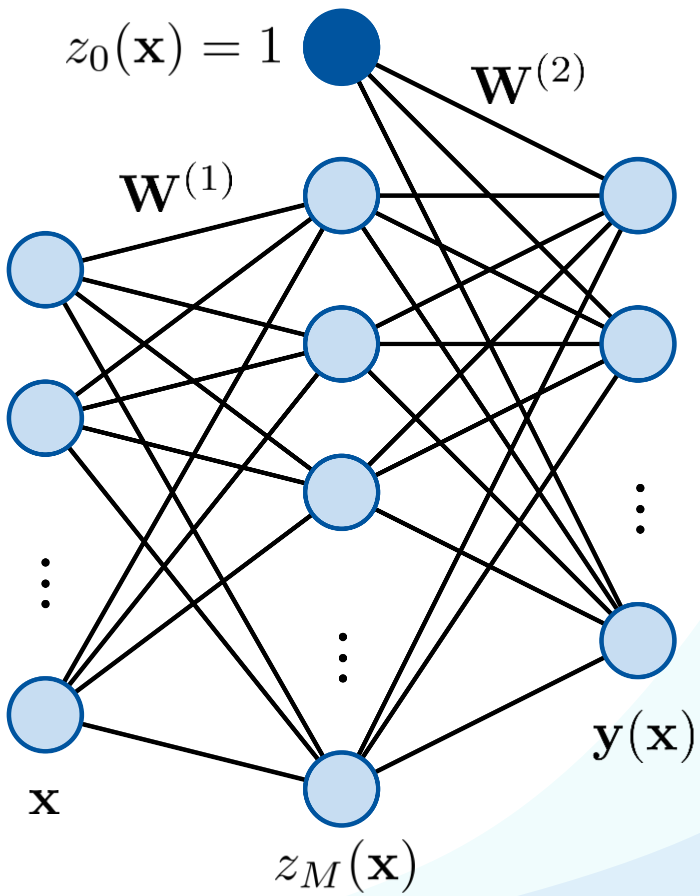


Multi-Layer Perceptrons

- Perceptrons are limited by having a fixed input mapping.
- Replace it with a hidden layer that learns suitable features.
- Output of hidden layer is input to next layer.
- Each layer computes a matrix multiplication and applies an elementwise activation function $g(\cdot)$:

$$\mathbf{z}(\mathbf{x}) = g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x}\right)$$
$$\mathbf{y}(\mathbf{x}) = g^{(2)}\left(\mathbf{W}^{(2)}\mathbf{z}(\mathbf{x})\right)$$

- Key step: *Now we also make the earlier layer learnable!*
- This is known as a Multi-Layer Perceptron (MLP).

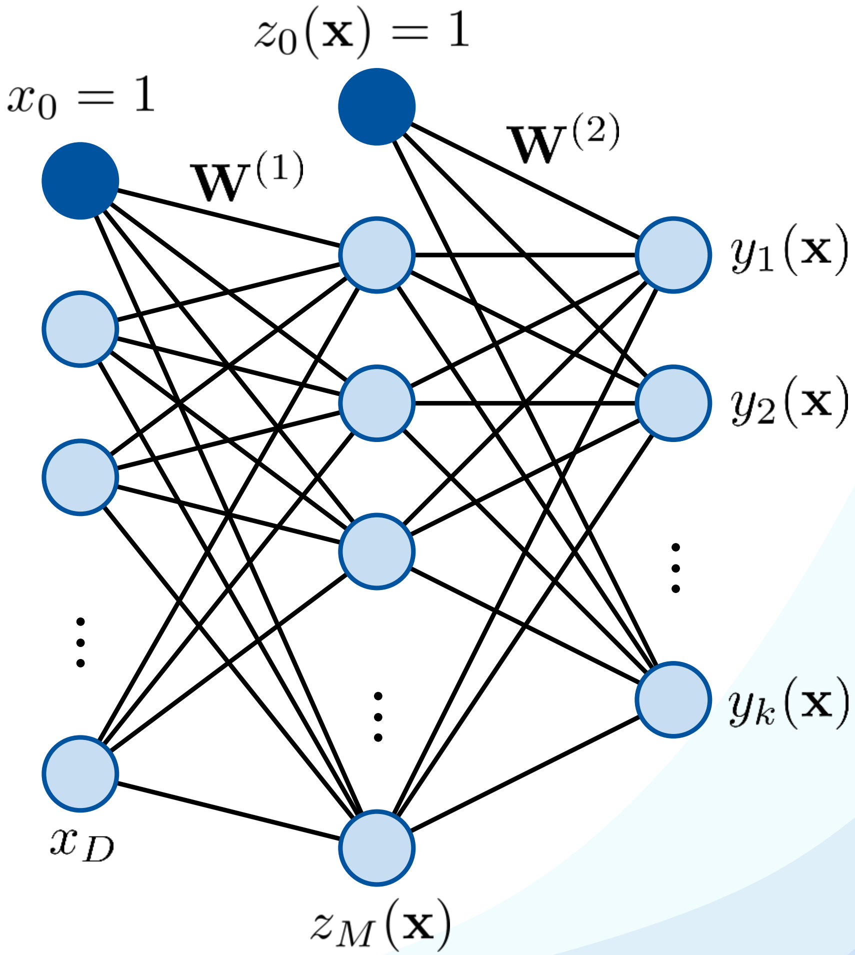


General Network Structure

- Multi-Layer Perceptron model:

$$y_k(\mathbf{x}) = g^{(2)}\left(\sum_{i=0}^{M} w_{ki}^{(2)} g^{(1)}\left(\sum_{j=0}^{D} w_{ij}^{(1)} x_j\right)\right)$$

- Usually, each layer adds a bias term.
- Activation functions between layers should be non-linear.
  - For example: $g^{(2)}(a) = \sigma(a),\ g^{(1)}(a) = \max\{a, 0\}$
  - With linear activations, successive layers would still compute a linear function.
- The hidden layer can have an arbitrary number of nodes.
- There can also be multiple hidden layers.

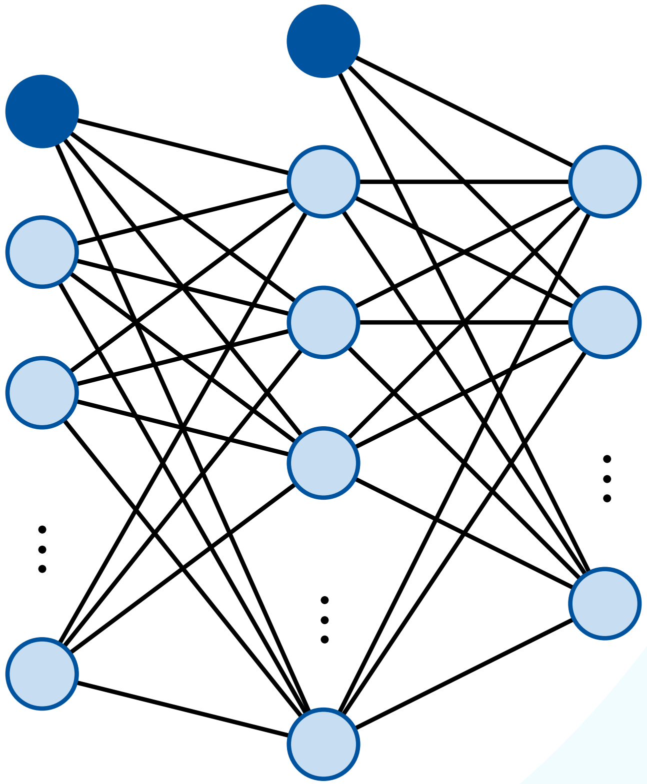


MLPs are Universal Approximators

$$y_k(\mathbf{x}) = g^{(2)}\left(\sum_{i=0}^{M} w_{ki}^{(2)} g^{(1)}\left(\sum_{j=0}^{D} w_{ij}^{(1)} x_j\right)\right)$$

- Universal Approximator Theorem:
  - A network with one hidden layer can approximate any continuous function of a compact domain arbitrarily well (assuming sufficient hidden nodes).

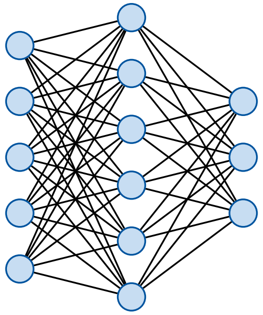
$\Rightarrow$ *Way more powerful than linear models!*



Learning with Hidden Units

- We now have a model that contains multiple layers of adaptive non-linear hidden units.
- How can we train such models?
  - Need to train *all* weights, not just last layer.
  - Learning the weights to the hidden units = learning features.
  - We don't know what the hidden units should do.
- Basic Idea: Gradient Descent.

This is the main challenge of deep learning!

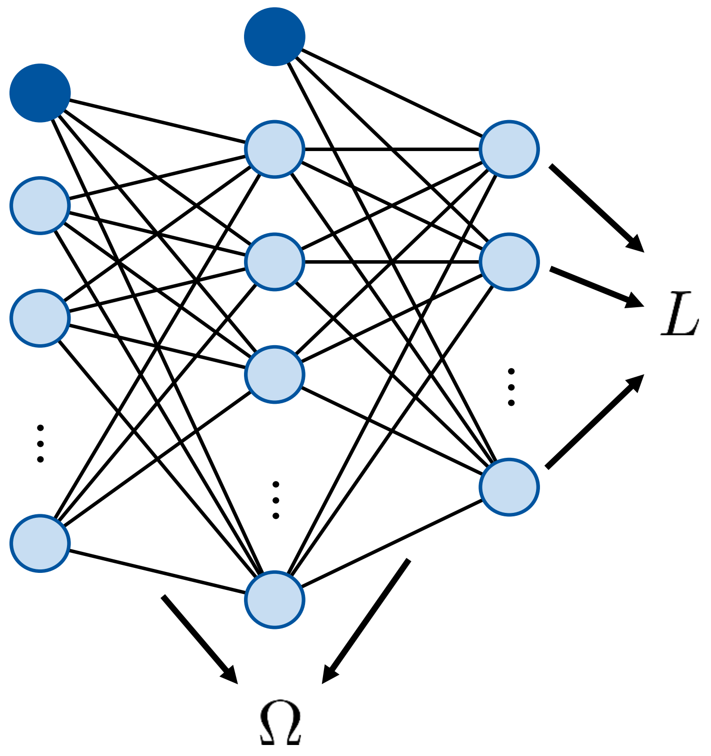


$E(\mathbf{w})$ → $\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ → $\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ → Repeat until convergence.

Apply Error Function | Calculate Gradients | Update Weights

Neural Network Basics

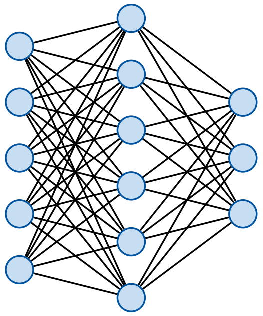
1. Perceptrons
2. Multi-Layer Perceptrons
3. **Loss Functions**
4. Backpropagation
5. Stochastic Gradient Descent



Loss Functions

- We train Neural Networks by minimizing an error function
  - In principle, any differentiable objective function can be used here.
  - Typically, we use a combination of a loss function $L(t, y(\mathbf{x}))$ and a regularizer $\Omega(\mathbf{w})$:

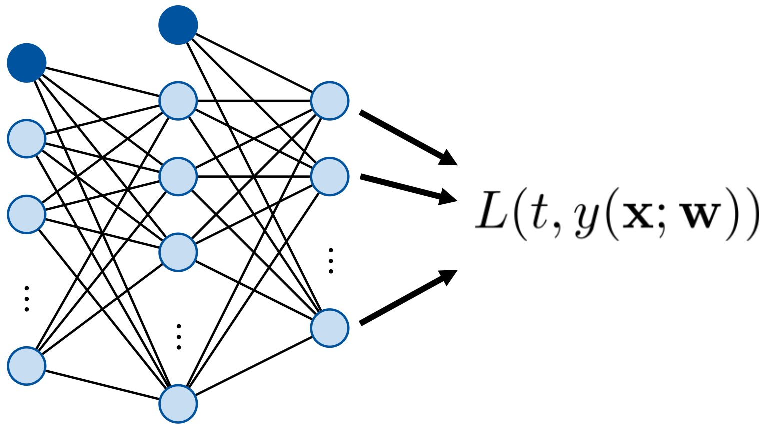
$$E(\mathbf{w}) = \sum_{n=1}^{N} L(t_n, y(\mathbf{x}_n; \mathbf{w})) + \lambda \Omega(\mathbf{w})$$



$E(\mathbf{w})$ → $\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ → $\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$

Apply Error Function | Calculate Gradients | Update Weights

Examples of Loss Functions



- We can use any of the loss functions we have seen so far to achieve different effects:
  - L_2 loss (Squared Error)

    $$L(t, y(\mathbf{x})) = \frac{1}{2}(y(\mathbf{x}) - t)^2$$

    $\Rightarrow$ Least-squares regression / classif.
  - Binary Cross-Entropy loss

    $$L(t, y(\mathbf{x})) = -(t \ln \sigma(y(\mathbf{x})) + (1-t)\ln(1 - \sigma(y(\mathbf{x}))))$$

    $\Rightarrow$ Logistic regression
  - Hinge loss

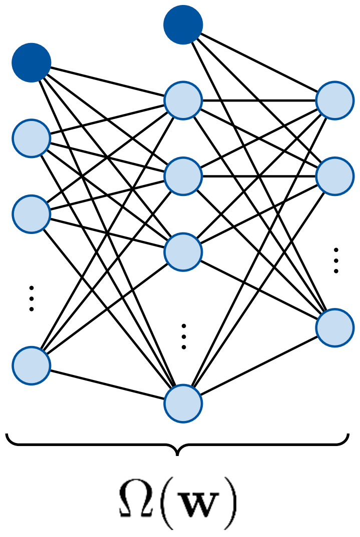
    $$L(t, y(\mathbf{x})) = [1 - t y(\mathbf{x})]_+$$

    $\Rightarrow$ SVM classification
  - Multi-Class Cross-Entropy loss

    $$L(t, \mathbf{y}(\mathbf{x})) = -\sum_k \left(\mathbb{I}(t = k) \ln \frac{\exp(y_k(\mathbf{x}))}{\sum_j \exp(y_j(\mathbf{x}))} \right)$$

    $\Rightarrow$ Multi-class probabilistic classification

Examples of Regularization Terms



- Similarly, we can use any of the regularization terms we have seen so far:
  - L_2 regularizer ("Weight Decay")

$$\Omega(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_2^2$$

  - L_1 regularizer

$$\Omega(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_1$$

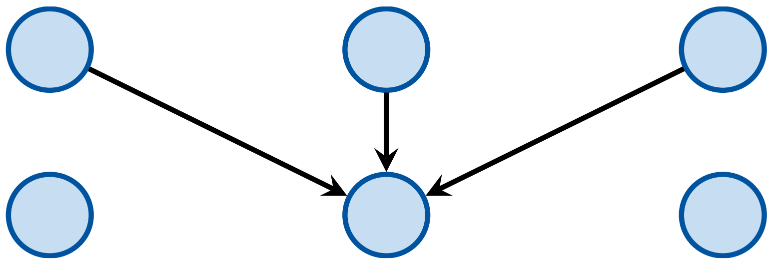
$\Rightarrow$ Prevents overfitting

$\Rightarrow$ Enforces sparsity (feature selection)

- Since Neural Networks have many parameters, regularization becomes an important consideration.
  - Many of the more advanced NN "training tricks" can also be understood as a form of regularization

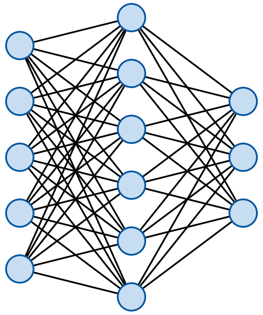
Neural Network Basics

1. Perceptrons
2. Multi-Layer Perceptrons
3. Loss Functions
4. **Backpropagation**
5. Stochastic Gradient Descent



Backpropagation

- We know a flexible model that is able to learn features.
- We also know how to compute an error estimate.
- Now we need to compute the gradients with respect to our parameters.



$E(\mathbf{w})$

Apply Error Function

$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$

Calculate Gradients

?

$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$

Update Weights

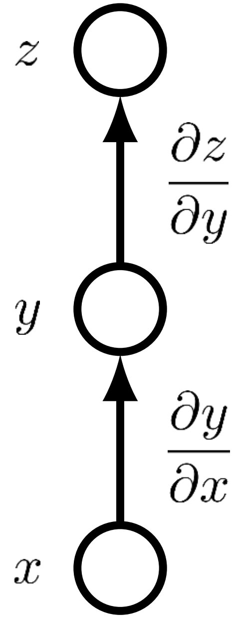
Approach 1: Naïve Analytical Differentiation

- Compute the gradients of each variable analytically.
- Scalar case is straightforward:

$$\Delta z = \frac{\partial z}{\partial y} \Delta y \qquad \Delta y = \frac{\partial y}{\partial x} \Delta x$$

$$\Delta z = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \Delta x$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

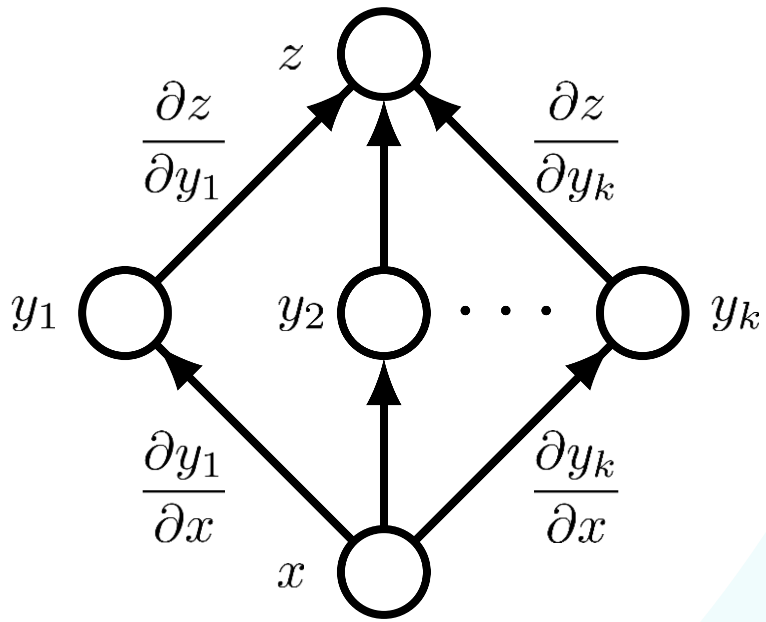


Approach 1: Naïve Analytical Differentiation

- Compute the gradients of each variable analytically.
- Scalar case is straightforward.
- Multi-dimensional case: Total derivative
  - Need to sum over all paths to target variable:

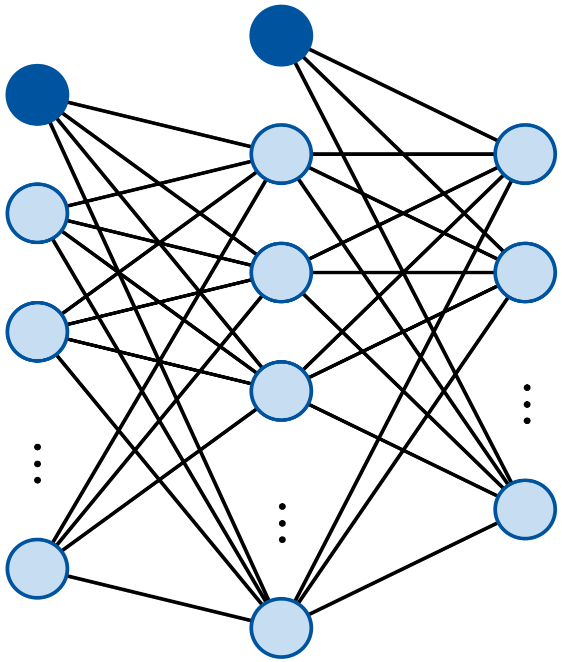
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x} + \ldots = \sum_{i=1}^{k} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

  - With increasing depth, there will be exponentially many paths!



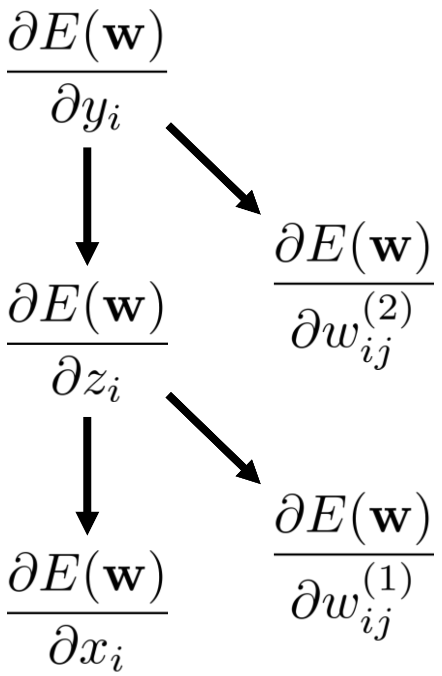
Approach 2: Numerical Differentiation

- Given the current state $\mathbf{w}^{(\tau)}$, we can evaluate $E(\mathbf{w}^{(\tau)})$.
- Idea: Make small changes to $\mathbf{w}^{(\tau)}$ and accept those that improve $E(\mathbf{w}^{(\tau)})$.
- Need several forward passes for each weight – over the whole dataset.
- This is horribly inefficient!

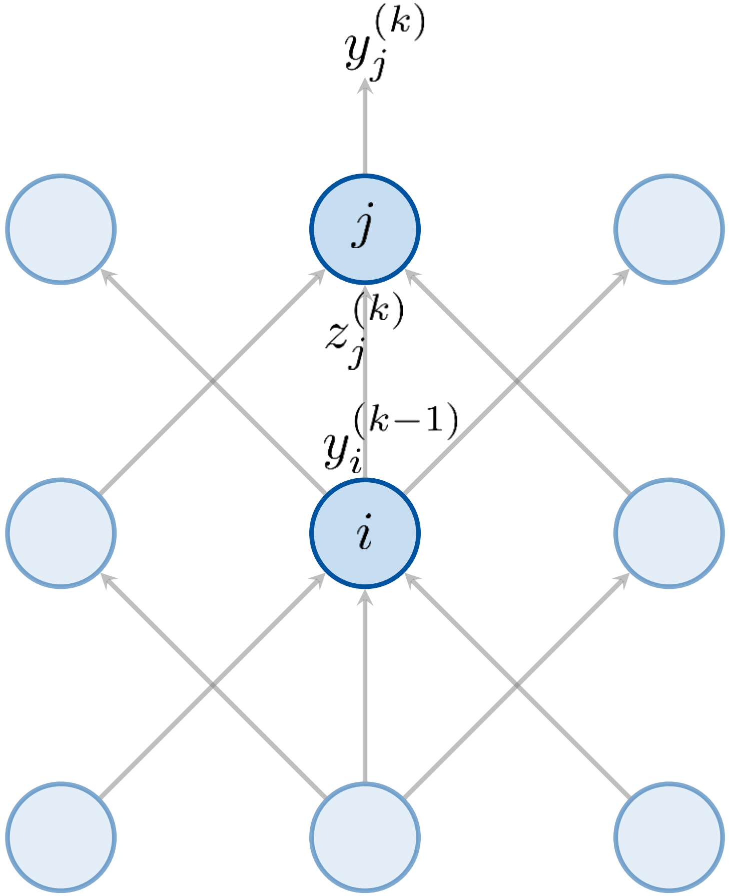


Approach 3: Incremental Analytical Differentiation

- Idea: compute the gradients layer by layer.
- Each layer below builds upon the results of the layer above.
- The gradient is propagated backwards through the layers.
- This is the backpropagation algorithm.



Example: Backpropagation for MLPs



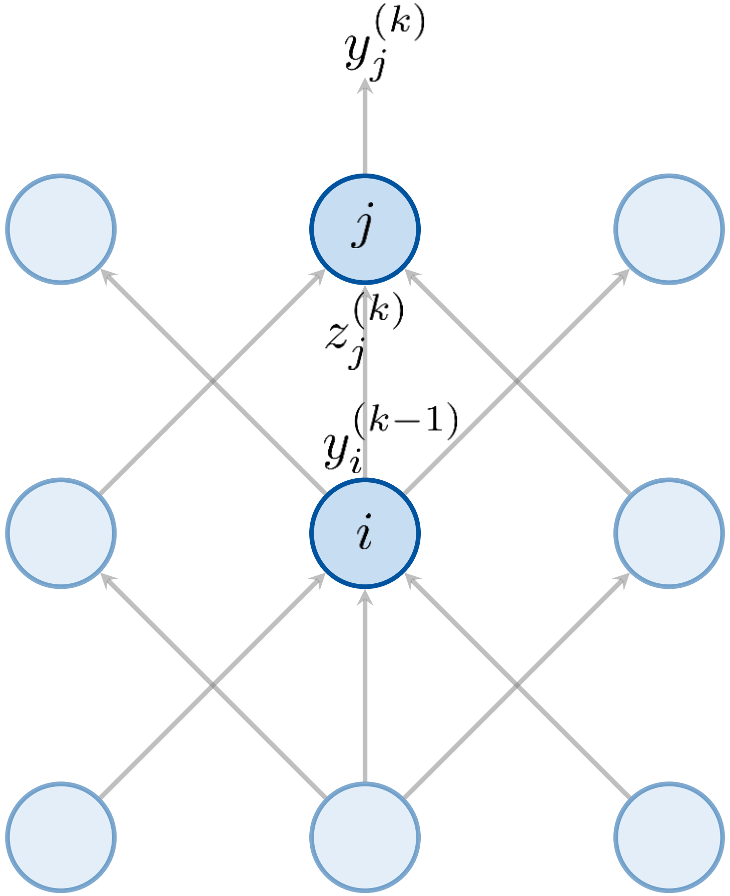
$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}}$$

Input of layer k: $z_j^{(k)} = \sum_i w_{ji}^{(k-1)} y_i^{(k-1)}$

Output of layer k: $y_j^{(k)} = g\left(z_j^{(k)}\right)$

Example: Backpropagation for MLPs



$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

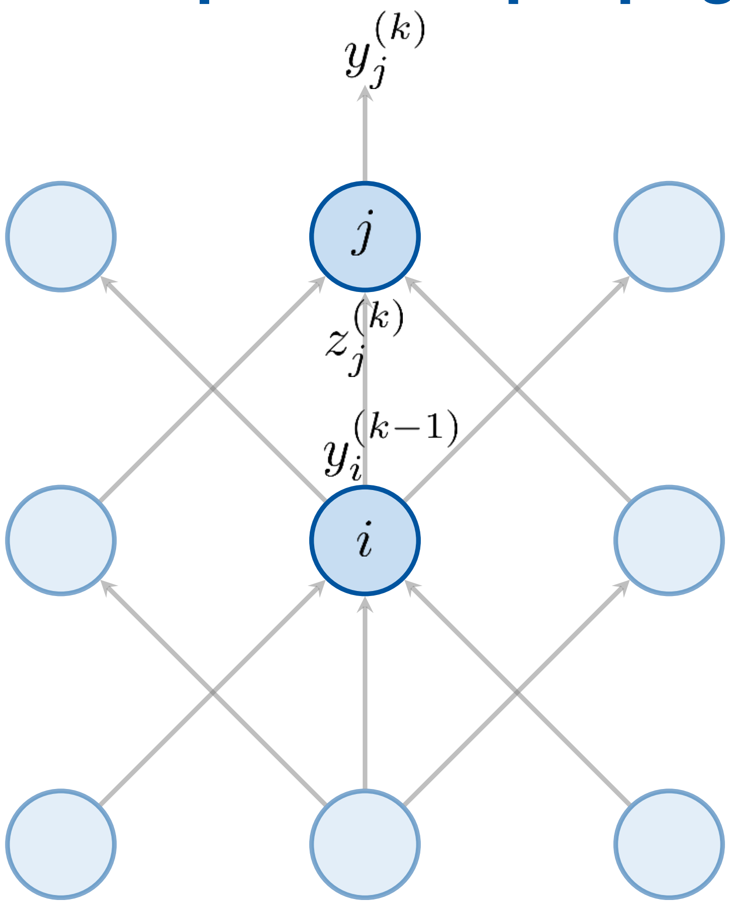
$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = \sum_j w_{ji}^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

Input of layer k: $z_j^{(k)} = \sum_i w_{ji}^{(k-1)} y_i^{(k-1)}$

Output of layer k: $y_j^{(k)} = g\left(z_j^{(k)}\right)$

$$\frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} = w_{ji}^{(k-1)}$$

Example: Backpropagation for MLPs



$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

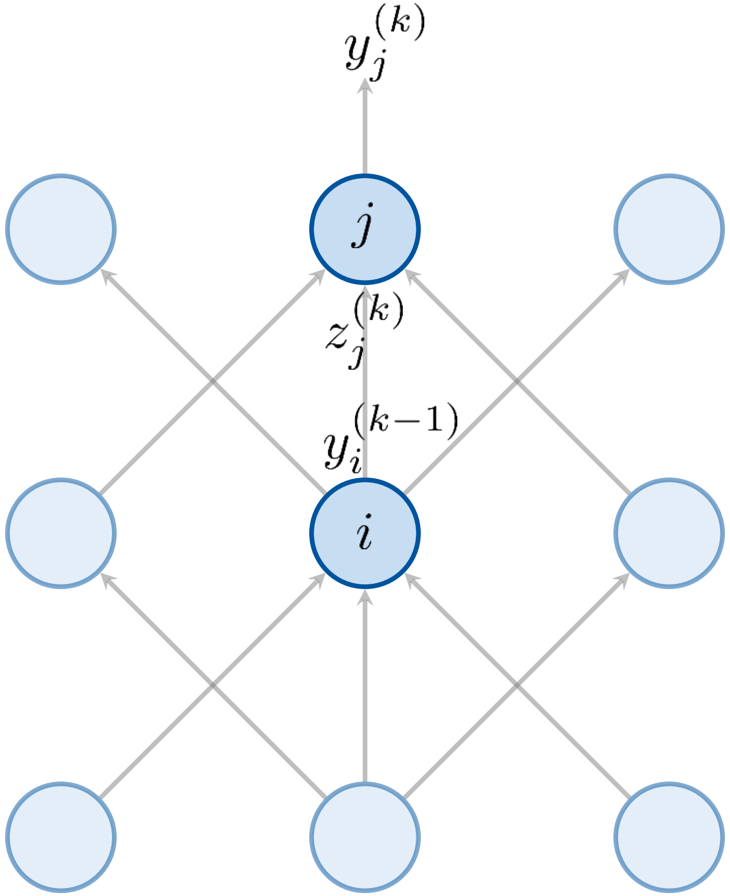
$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = \sum_j w_{ji}^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

$$\frac{\partial E}{\partial w_{ji}^{(k-1)}} = \frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}}$$

Input of layer k: $z_j^{(k)} = \sum_i w_{ji}^{(k-1)} y_i^{(k-1)}$

Output of layer k: $y_j^{(k)} = g\left(z_j^{(k)}\right)$

Example: Backpropagation for MLPs



$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = \sum_j w_{ji}^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

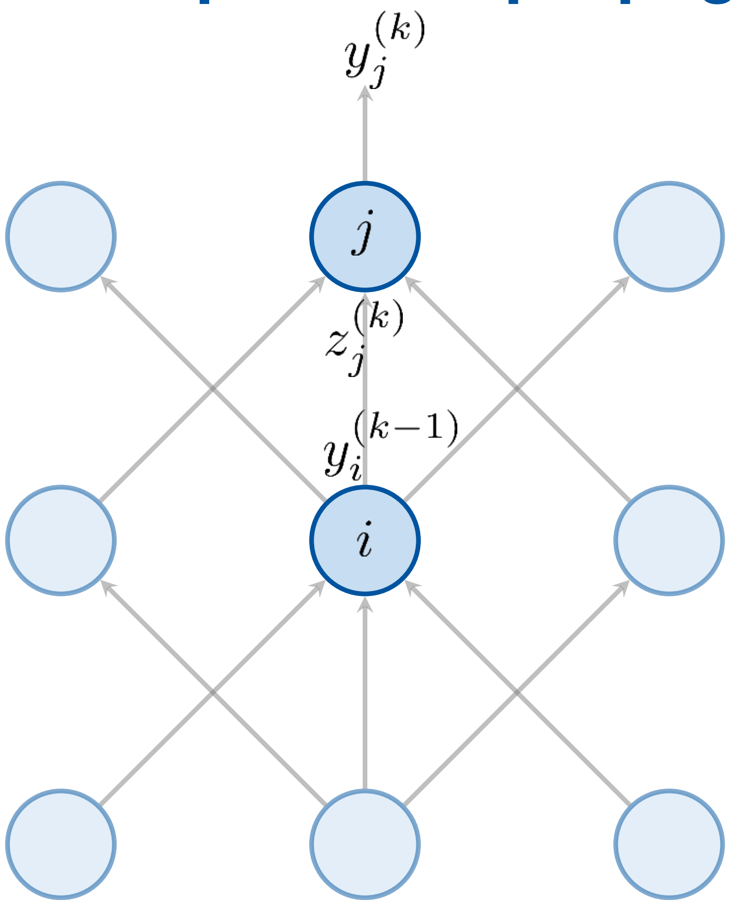
$$\frac{\partial E}{\partial w_{ji}^{(k-1)}} = \frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = y_i^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

Input of layer k: $z_j^{(k)} = \sum_i w_{ji}^{(k-1)} y_i^{(k-1)}$

Output of layer k: $y_j^{(k)} = g\left(z_j^{(k)}\right)$

$$\frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k-1)}} = y_i^{(k-1)}$$

Example: Backpropagation for MLPs



Input of layer k: $z_j^{(k)} = \sum_i w_{ji}^{(k-1)} y_i^{(k-1)}$

Output of layer k: $y_j^{(k)} = g\left(z_j^{(k)}\right)$

$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = \sum_j w_{ji}^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

$$\frac{\partial E}{\partial w_{ji}^{(k-1)}} = \frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = y_i^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

Discussion Backpropagation

Advantages

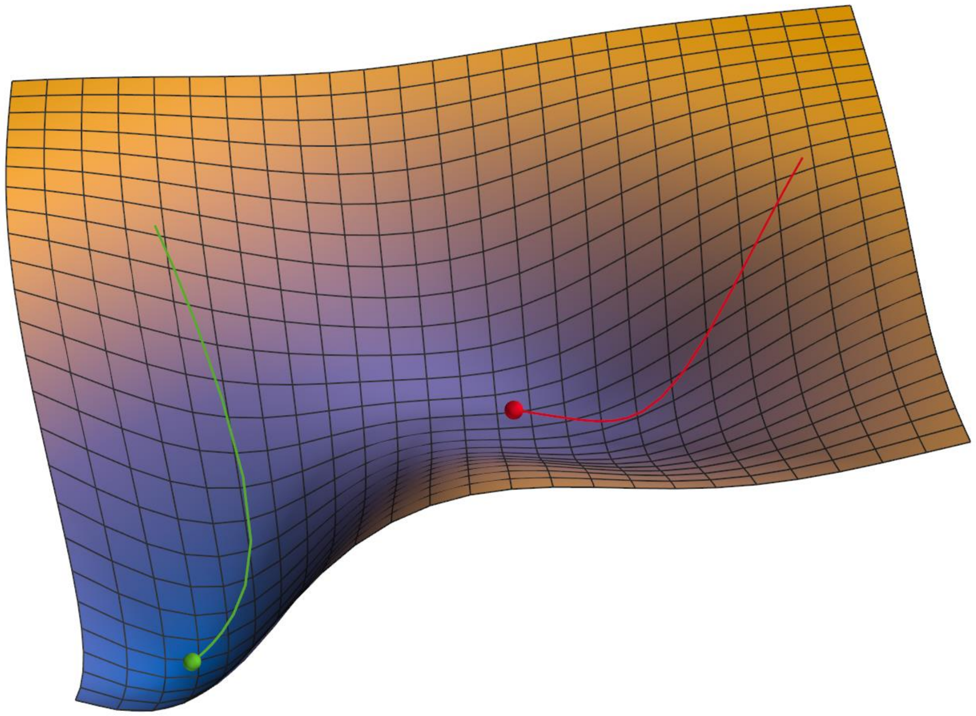
- Very general algorithm, widely used.
- Efficiently computes all gradients in the network using dynamic programming.
- The same concept can be applied to any differentiable function.
  - This makes it possible to define other types of layers.

Limitations

- Efficient evaluation of backpropagation requires storing all unit activations from forward pass.
  - The amount of memory necessary for this imposes a practical limit on the size of the network.
- Successful learning relies on the gradients to be propagated to the early network layers.
  - Numerical challenges may arise here.

Neural Network Basics

1. Perceptrons
2. Multi-Layer Perceptrons
3. Loss Functions
4. Backpropagation
5. **Stochastic Gradient Descent**

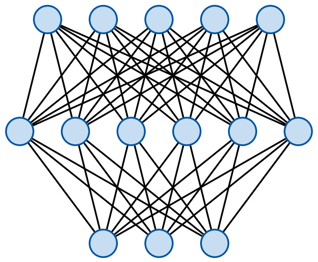


Stochastic Gradient Descent

- Now that we have the gradients, we need to update the weights.
- We already know the basic equation for this
  - (1st-order) Gradient Descent

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta \left. \frac{\partial E(\mathbf{w})}{\partial w_{kj}} \right|_{\mathbf{w}^{(\tau)}}$$

- Remaining Questions:
  - On what data do we want to apply this?
  - How should we choose the learning rate η ?



Apply Error Function

Calculate Gradients

Update Weights

Stochastic vs. Batch Learning

- Batch Learning
  - Process the full dataset in one batch.
  - Compute the gradient based on all training examples.

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta \left. \frac{\partial E(\mathbf{w})}{\partial w_{kj}} \right|_{\mathbf{w}^{(\tau)}}$$

- Stochastic Learning
  - Choose a single example from the training set.
  - Compute the gradient only based on this example.
  - This estimate will generally be noisy, which has some advantages.

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta \left. \frac{\partial E_n(\mathbf{w})}{\partial w_{kj}} \right|_{\mathbf{w}^{(\tau)}}$$

Batch Learning

- Conditions of convergence are well understood.
- Many acceleration techniques only work in batch learning.
- Theoretical analysis of the weight dynamics and convergence rates are simpler.

Stochastic Learning

- Usually much faster than batch learning.
- Often results in better solutions.
- Can be used for tracking changes when the target distribution shifts.

Middle ground: Minibatches

Minibatches

- Idea
  - Process only a small batch of training examples together.
  - Start with a small batch size & increase it as training proceeds.

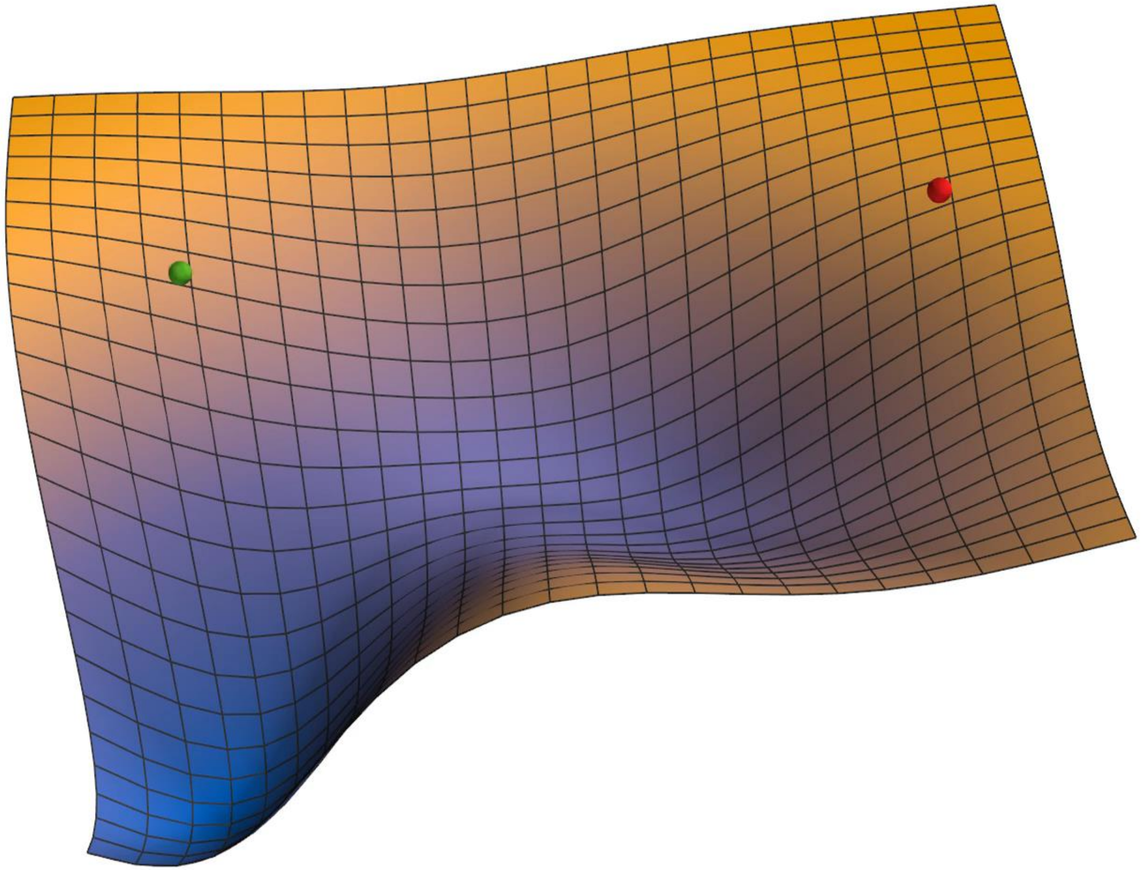
$$\mathcal{B} = \{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_B, t_B)\} \subset \mathcal{D}$$

- Advantages
  - Gradients will be more stable than for stochastic gradient descent, but still faster to compute than with batch learning.
  - Take advantage of redundancies in the training set.
  - Matrix operations are more efficient than vector operations.

- Caveat
  - Need to normalize error function by the minibatch size to use the same learning rate between minibatches

$$E(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} L(t_n, y(\mathbf{x}_n; \mathbf{w})) + \frac{\lambda}{N} \Omega(\mathbf{w})$$

Example



Choosing the Right Learning Rate

- Consider a simple 1D example:

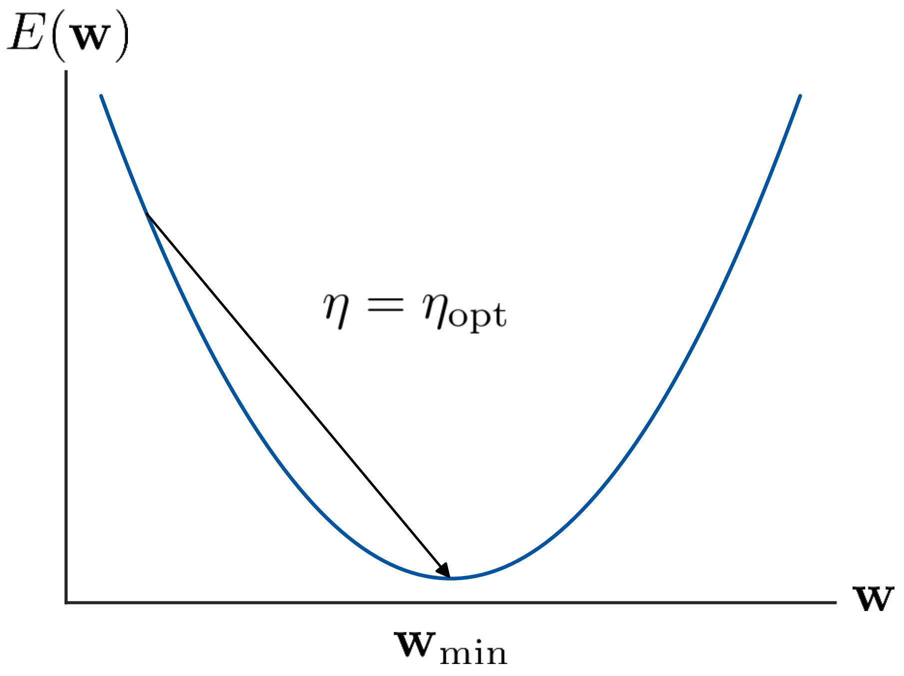
$$w^{(\tau+1)} = w^{(\tau)} - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$$

- What is the optimal learning rate η_{opt}?

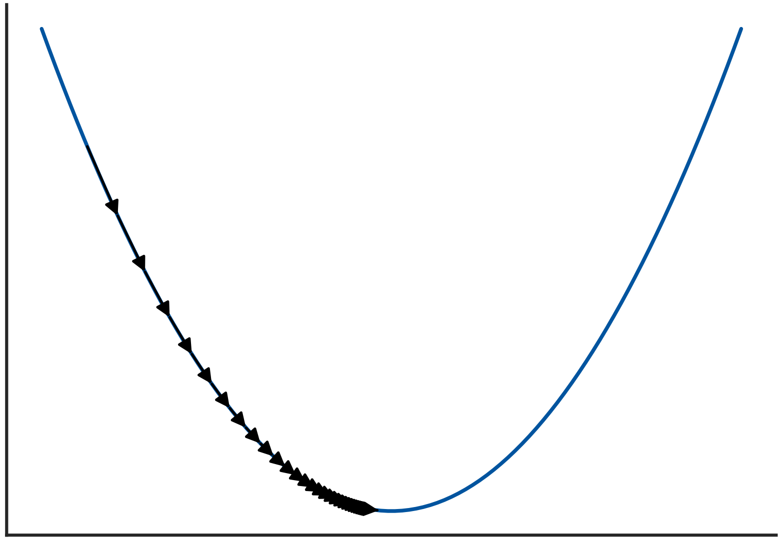
- If E is quadratic, the optimal learning rate is given by the inverse of the Hessian:

$$\eta_{\mathrm{opt}} = \left(\frac{\partial^2 E(\mathbf{w}^{(\tau)})}{\partial \mathbf{w}^2} \right)^{-1}$$

- *For neural networks, the Hessian is usually infeasible to compute.*

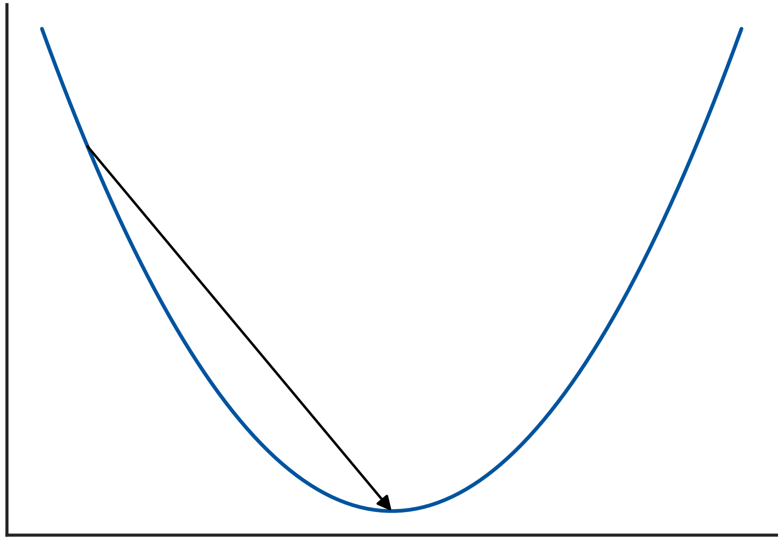


η too small



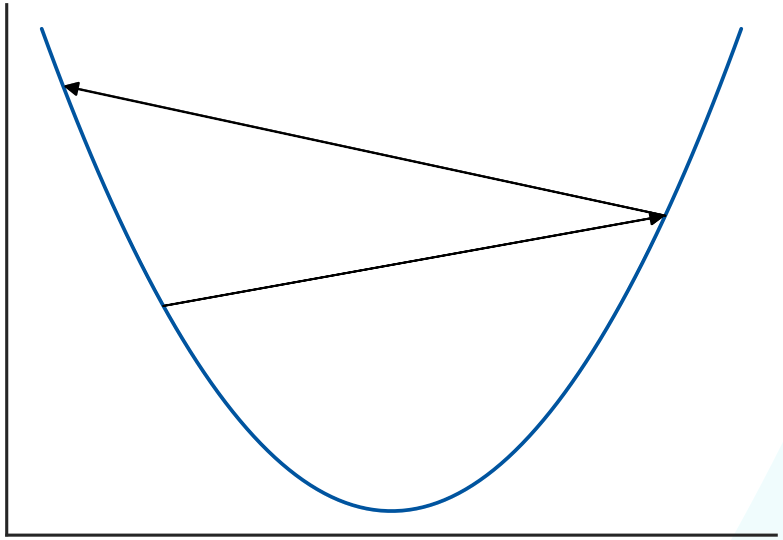
Convergence is slow

η_{opt}



Converges ideally in a single step

η too large



Might not converge

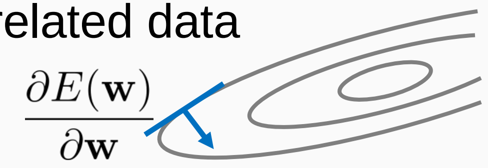
Discussion: Stochastic Gradient Descent

Advantages

- Very simple, but still quite robust method.
- Minibatches offer a compromise between stability and faster computation.
- Stochasticity in minibatches is often beneficial for learning

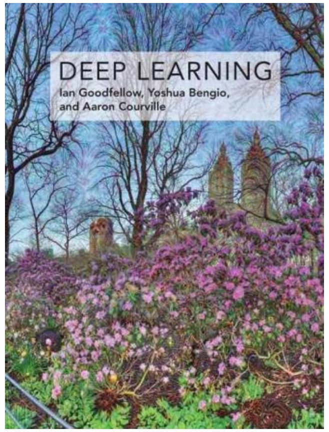
Limitations

- Finding a good setting for the learning rate is very important for fast convergence.
  - Choosing the right learning rate is a challenge and requires experience.
  - A different learning rate may be optimal for different parts of the network
- Following the direction of steepest descent is not always the fastest way to the optimum
  - E.g., in highly correlated data



References and Further Reading

- More information about Neural Networks and Deep Learning is available in the following book.



I. Goodfellow, Y. Bengio, A. Courville
Deep Learning
MIT Press, 2016

https://www.deeplearningbook.org/