Visual Computing Institute

RWTH AACHEN UNIVERSITY

Elements of Machine Learning & Data Science

Winter semester 2025/26

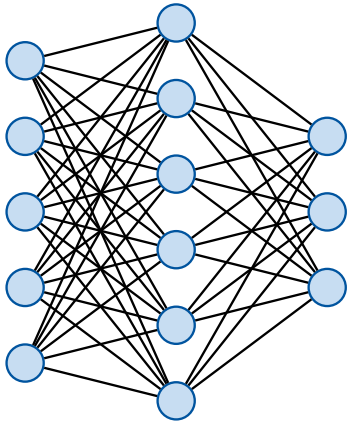
Lecture 16 – Neural Networks Basics

16.12.2025

Prof. Bastian Leibe

Machine Learning Topics

8. Introduction to ML
9. Probability Density Estimation
10. Linear Discriminants
11. Linear Regression
12. Logistic Regression
13. Support Vector Machines
14. **Neural Network Basics**



Multi-Layer Perceptrons

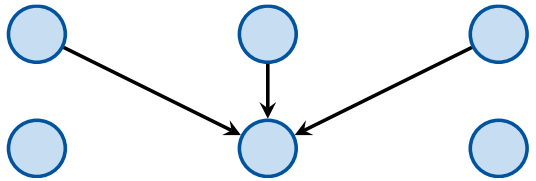
$$\frac{1}{2}(y(\mathbf{x}) - t)^2$$

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

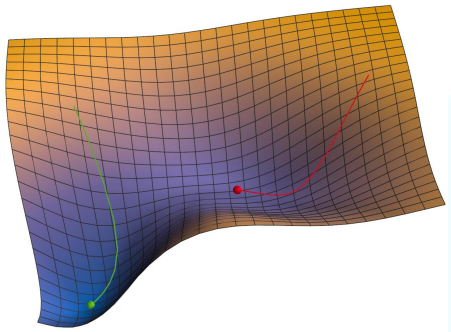
$$-\sum_k \left(\mathbb{I}(t = k) \ln \frac{\exp(y_k(\mathbf{x}))}{\sum_j \exp(y_j(\mathbf{x}))} \right)$$

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

Losses & Regularizers



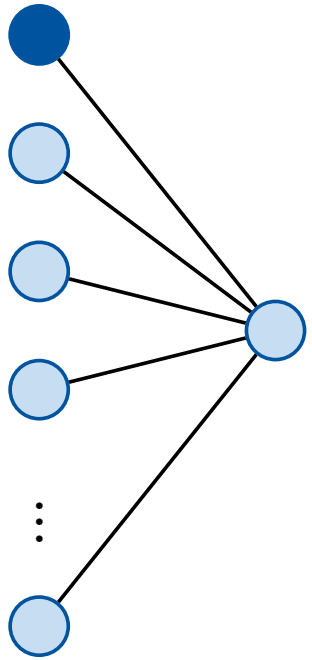
Backpropagation



Stochastic Gradient Descent

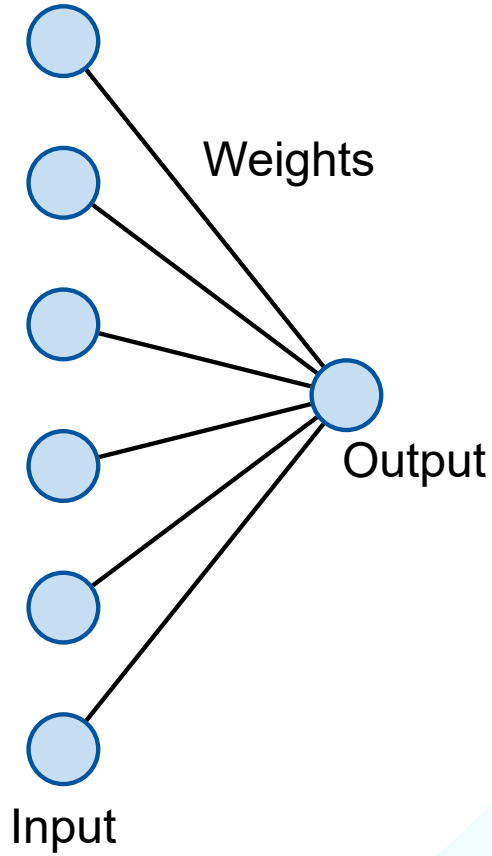
Neural Network Basics

1. **Perceptrons**
2. Multi-Layer Perceptrons
3. Loss Functions
4. Backpropagation
5. Computational Graphs
6. 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

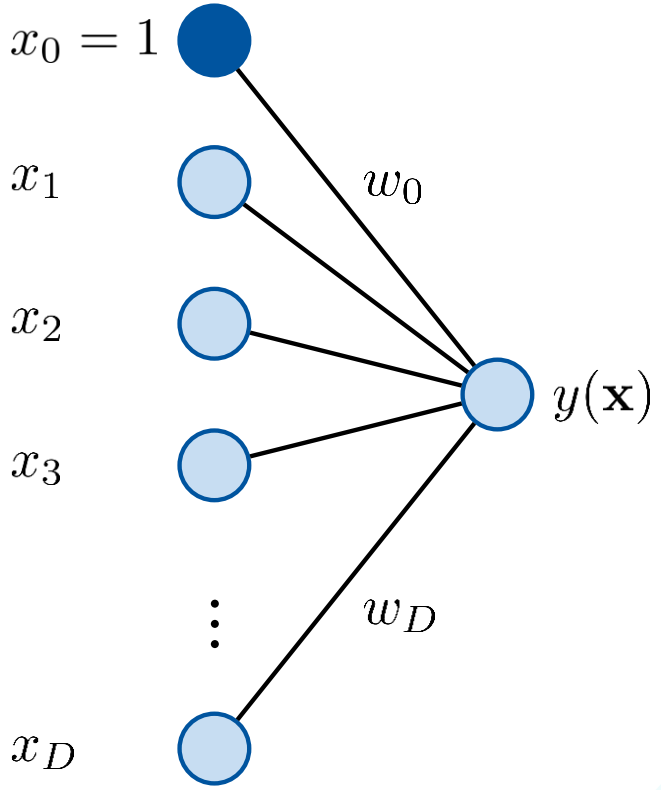
- Input Layer:
  - Hand-designed features
- Outputs:
  - Linear outputs

$$y(\mathbf{x}) = \mathbf{w}^{\mathsf{T}}\mathbf{x} + w_0$$

  - Logistic outputs

$$y(\mathbf{x}) = \sigma(\mathbf{w}^{\mathsf{T}}\mathbf{x} + w_0)$$

- Learning: finding optimal weights $\mathbf{w}$.



Multi-Class Networks

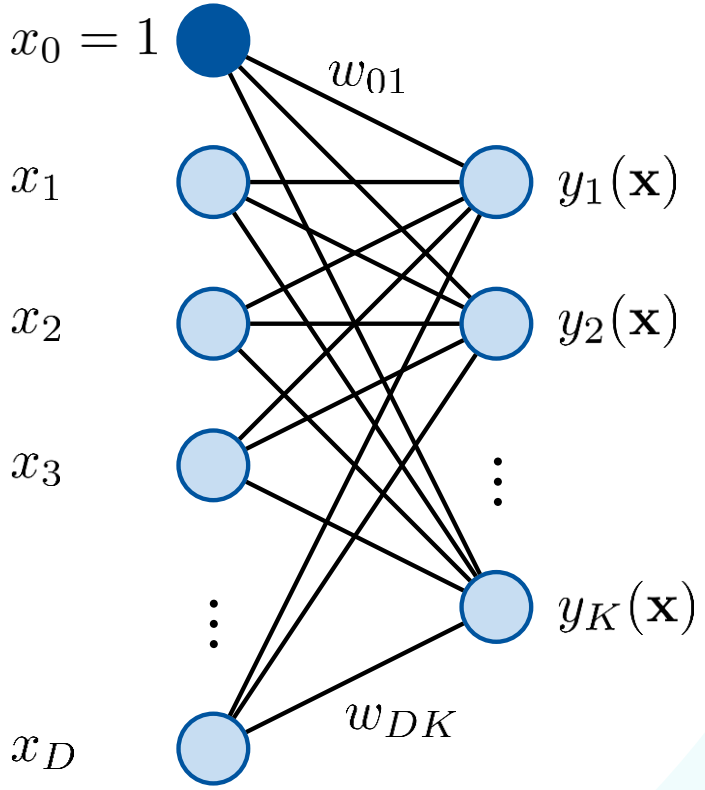
- One output node per class:
  - Linear outputs

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

  - 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.



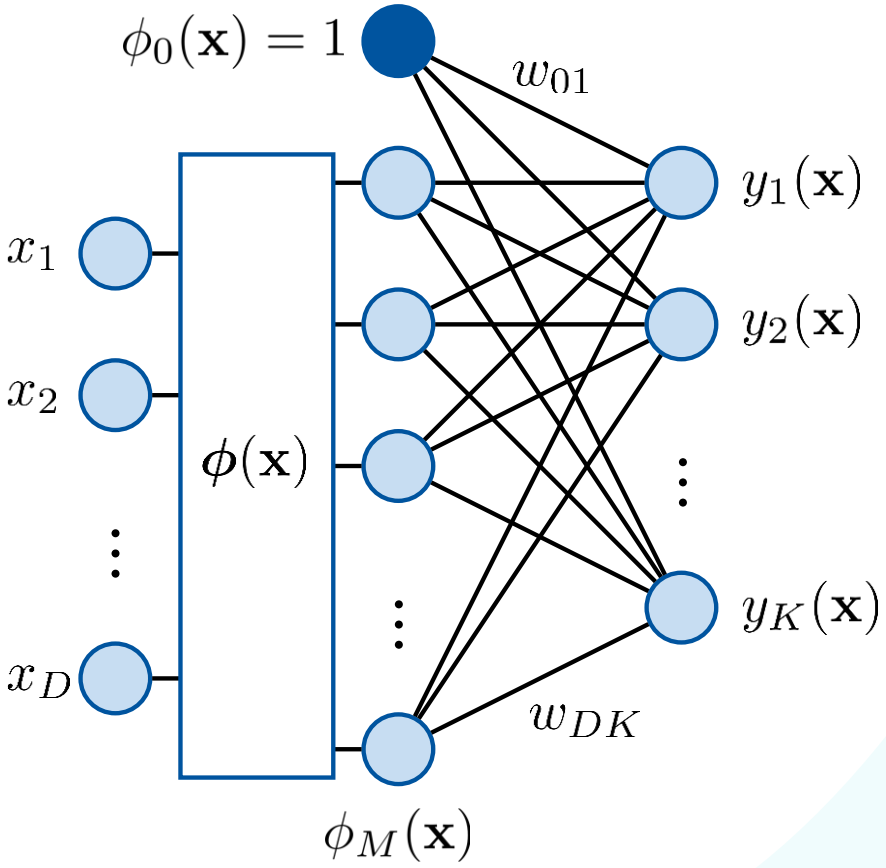
Non-Linear Basis Functions

- Apply a (fixed) mapping $\boldsymbol{\phi}(\mathbf{x})$ 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!

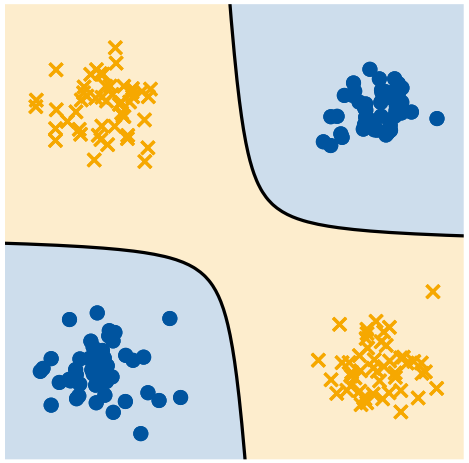
- What does that mean?
  - We have the same limitations as before.
  - Can model any separable function perfectly, given the right input features.
  - For some tasks, this may require an exponential number of input features.

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

Limitations so far

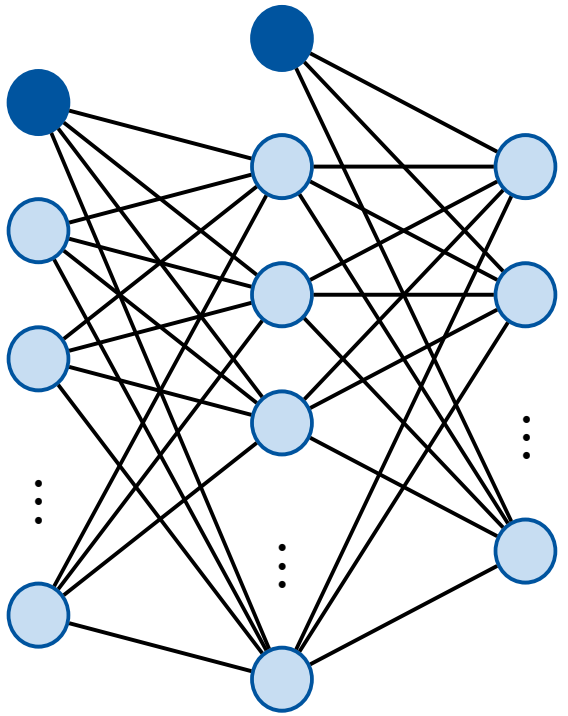
- Generalized linear discriminants (including perceptrons) are very limited.
  - A linear classifier cannot solve certain problems (e.g., XOR).
  - However, with a non-linear classifier based on suitable features, the problem becomes solvable.
  - So far, we have designed the features and kernels by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

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



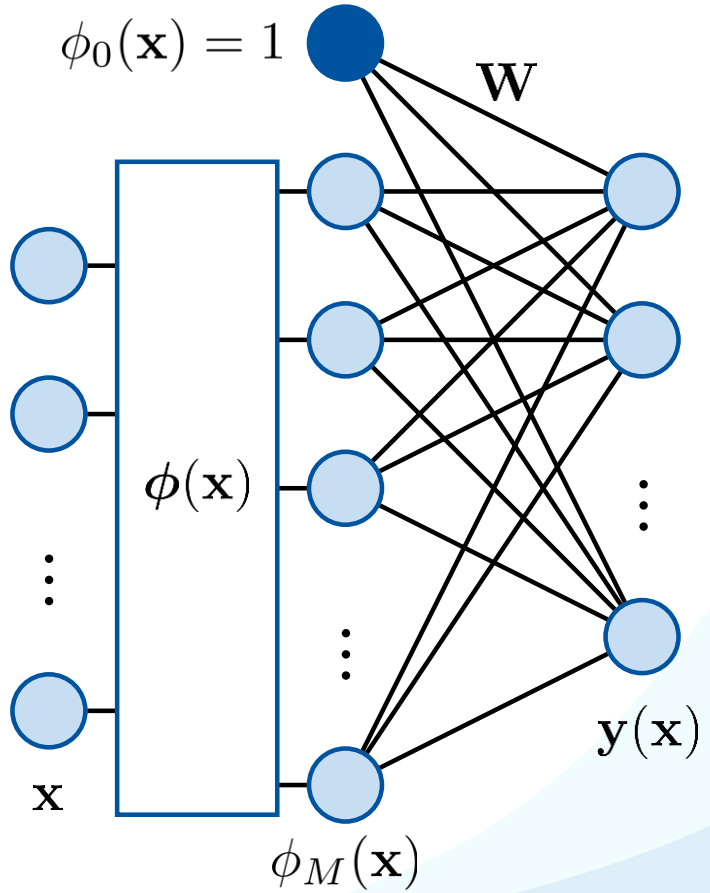
Neural Network Basics

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



Multi-Layer Perceptrons

- Perceptrons are limited by having a fixed input mapping.



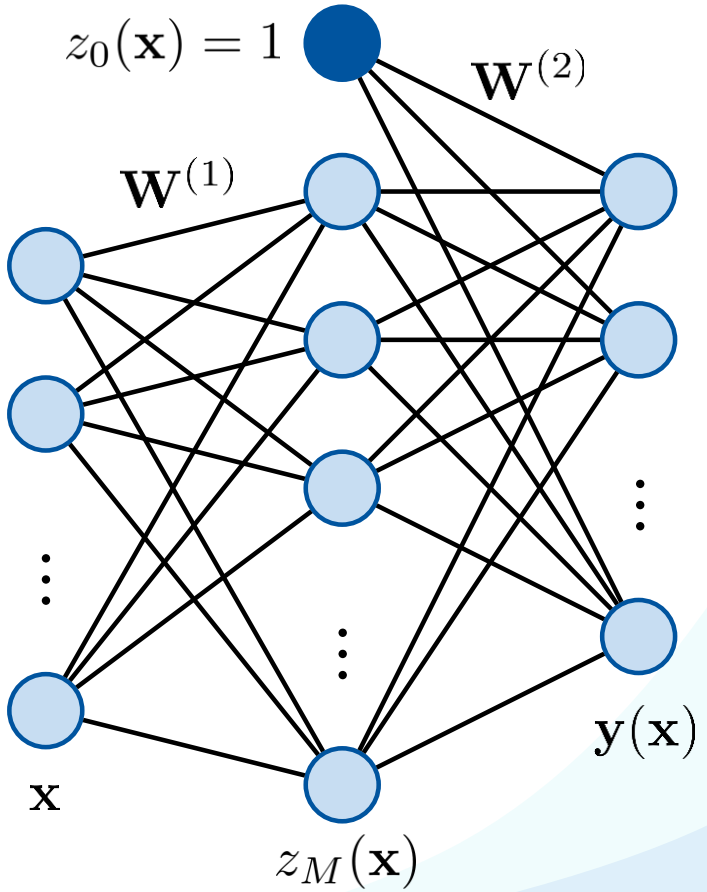
Multi-Layer Perceptrons

- 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).

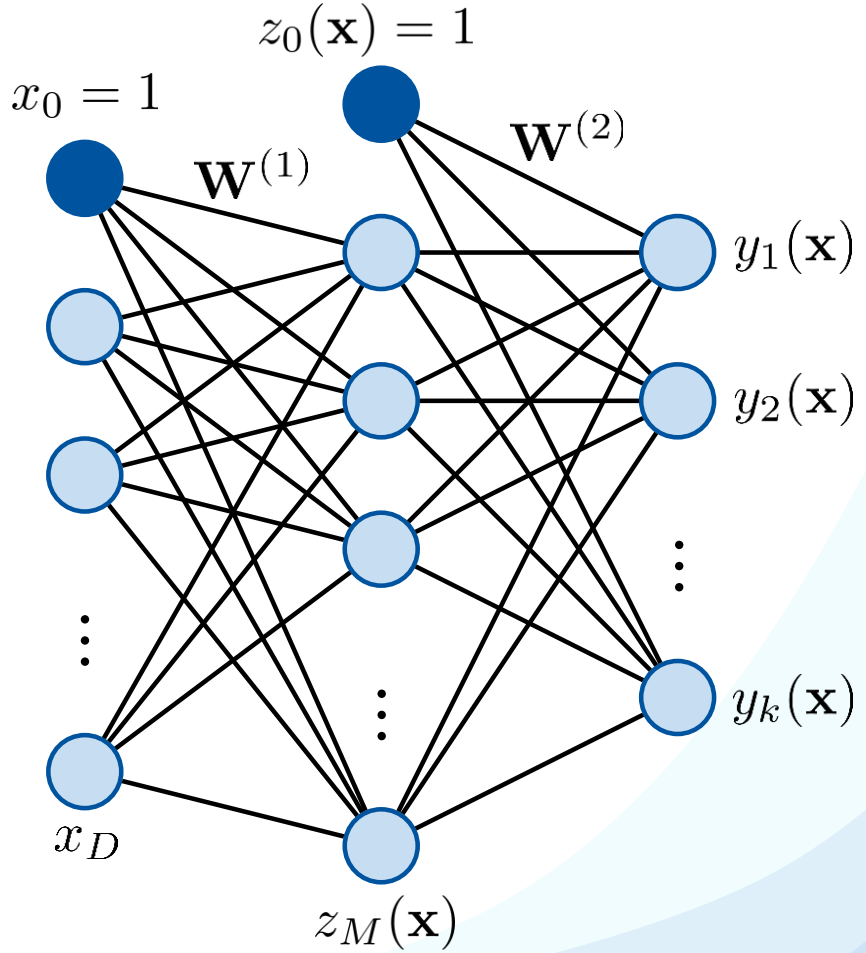


General Network Structure

- 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.

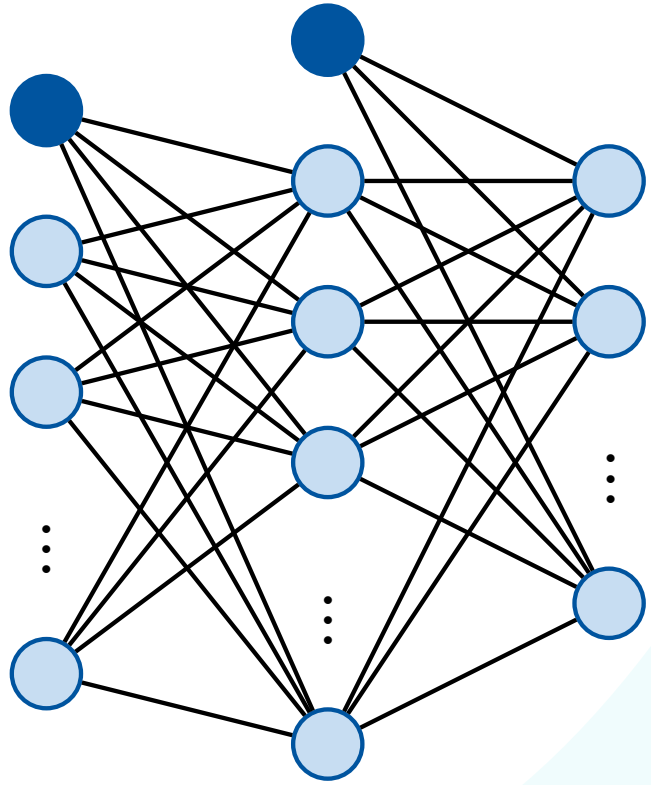


MLPs are Universal Approximators

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

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

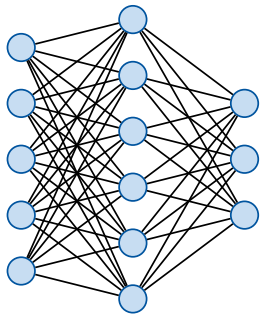
$\Rightarrow$ *Way more powerful than linear models!*



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!



$E(\mathbf{w})$

Apply Error Function

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

Calculate Gradients

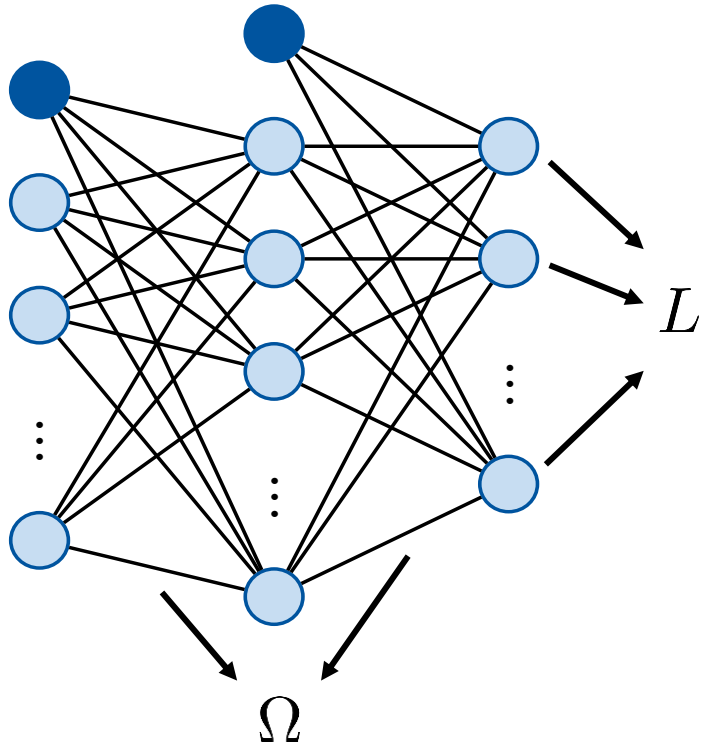
$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$

Update Weights

Repeat until convergence.

Neural Network Basics

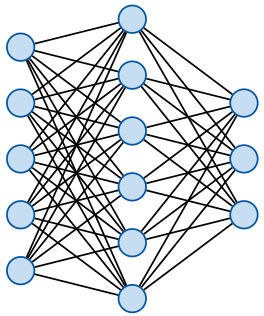
1. Perceptrons
2. Multi-Layer Perceptrons
3. **Loss Functions**
4. Backpropagation
5. Computational Graphs
6. Stochastic Gradient Descent



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 $\Omega(\mathbf{w})$:

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



$E(\mathbf{w})$

Apply Error Function

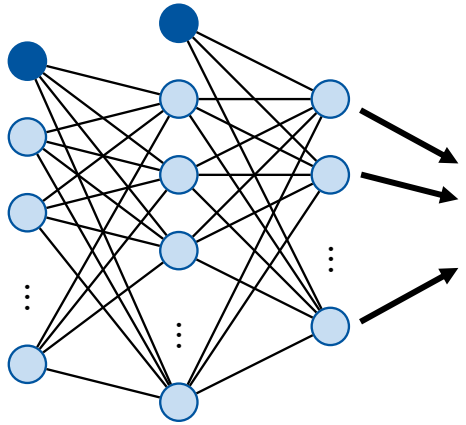
$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$

Calculate Gradients

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

Update Weights

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)

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

⇒ Least-squares regression / classif.

  - Binary Cross-Entropy loss

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

⇒ Logistic regression

  - Hinge loss

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

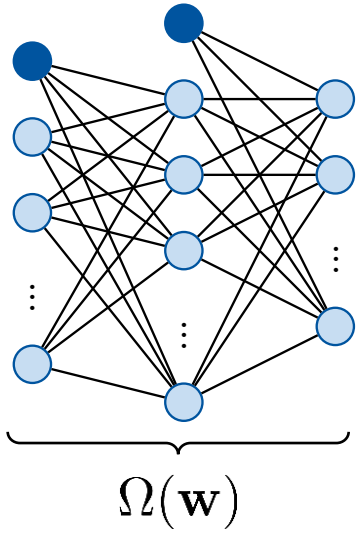
⇒ SVM classification

  - Multi-Class Cross-Entropy loss

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

⇒ 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")

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

⇒ Prevents overfitting

  - L_1 regularizer

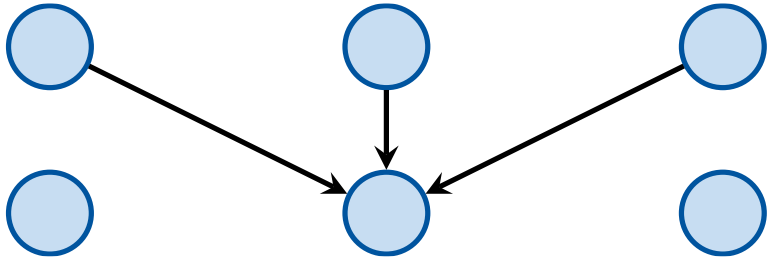
$$\Omega(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_1$$

⇒ 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

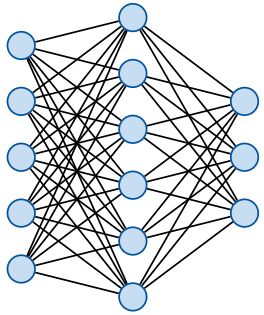
Neural Network Basics

1. Perceptrons
2. Multi-Layer Perceptrons
3. Loss Functions
4. **Backpropagation**
5. Computational Graphs
6. 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

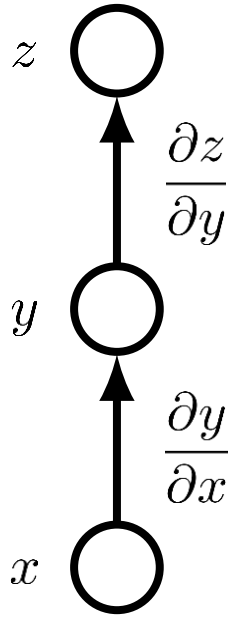
Approach 1: Naïve Analytical Differentiation

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

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

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

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

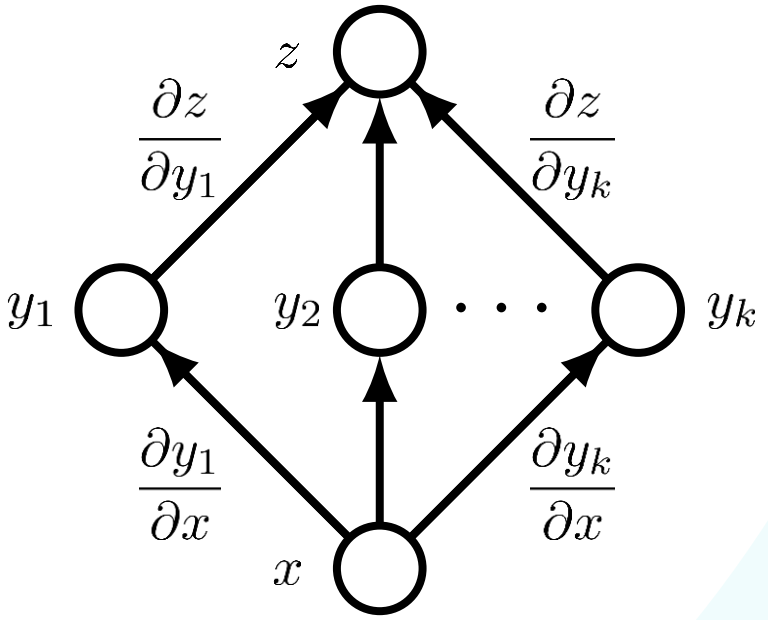


Approach 1: Naïve Analytical Differentiation

- 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:

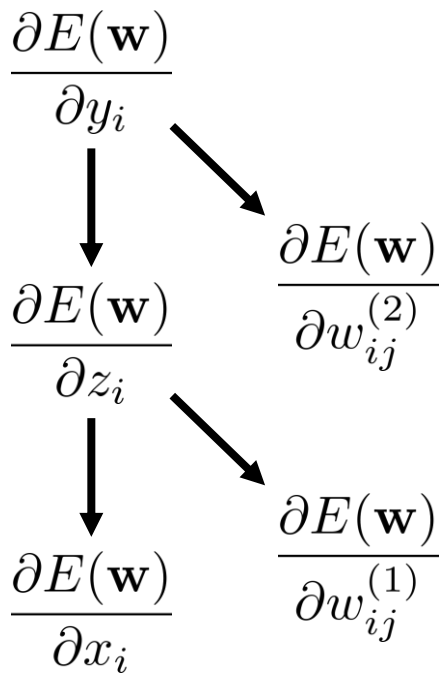
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x} + \ldots = \sum_{i=1}^{k} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

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

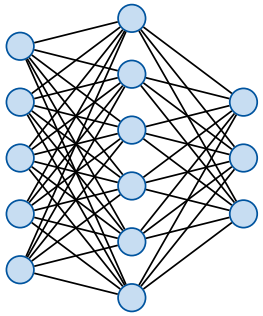


Approach 2: Incremental Analytical Differentiation

- 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.



Backpropagation



$E(\mathbf{w})$

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

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

Apply model to training data

Evaluate error function

Compute gradients

Update weights

Repeat

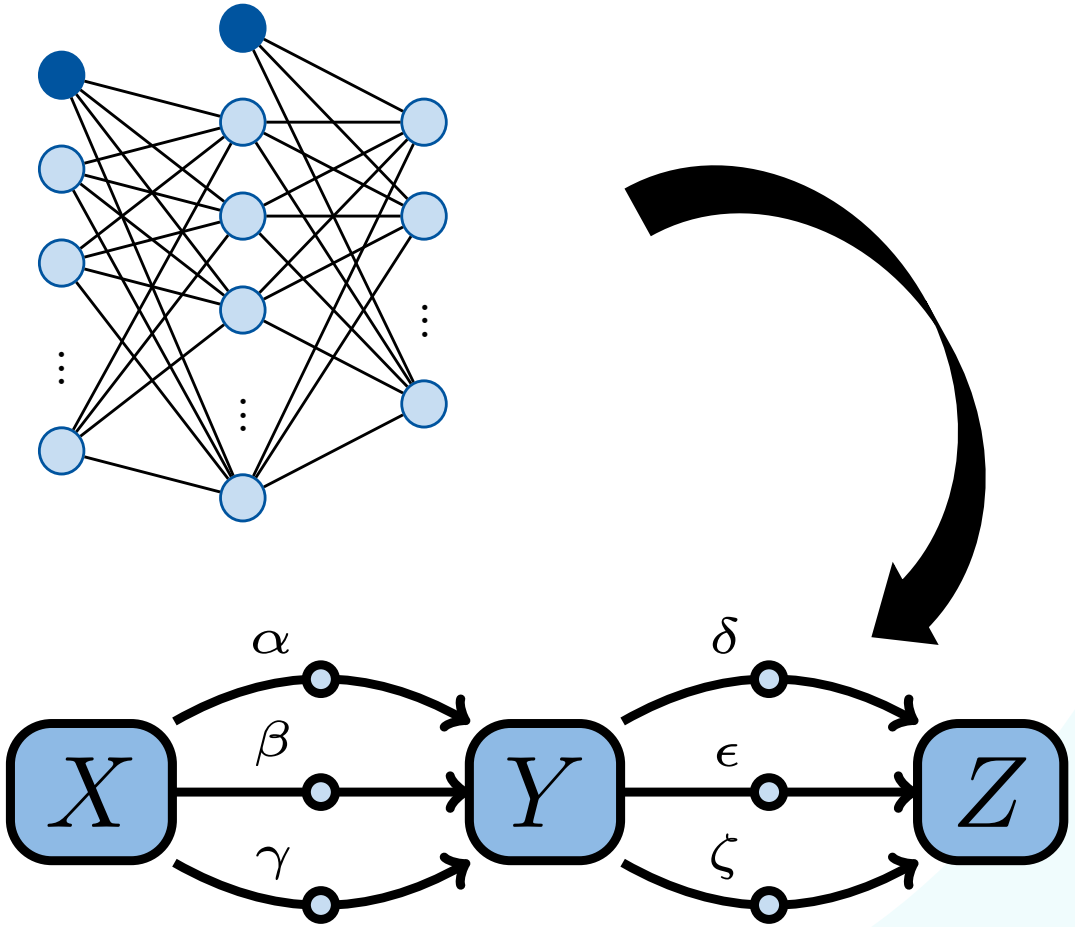
Forward pass

Backward pass

Implemented via
automatic differentiation

Automatic Differentiation

- Convert the network into a computational graph.
- Each layer/module just needs to specify how it affects the forward and backward passes.
- Apply backpropagation on the computational graph.
- Very general algorithm, used in today's Deep Learning packages.



Intuition

Let's assume we have a chain of functions

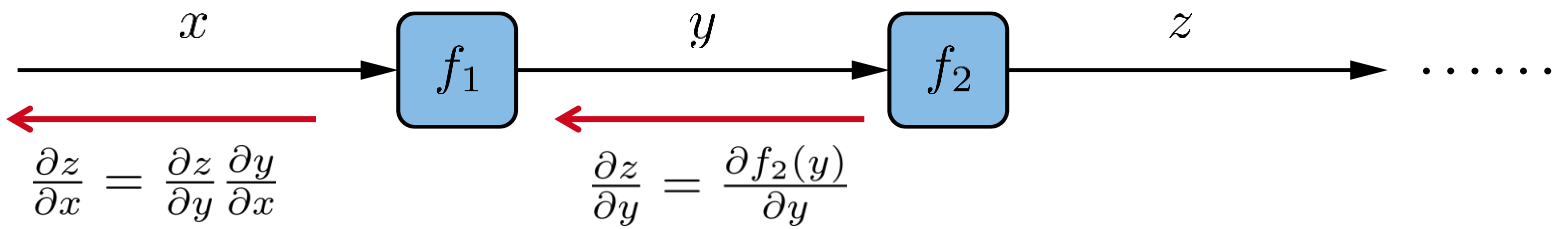
$$y = f_1(x)$$
$$z = f_2(y) = f_2(f_1(x))$$

How will the value of z change if we change x (i.e., $\frac{\partial z}{\partial x}$)? Use the chain rule!

Find out how y will affect z, then find out how x will affect y

Formula:

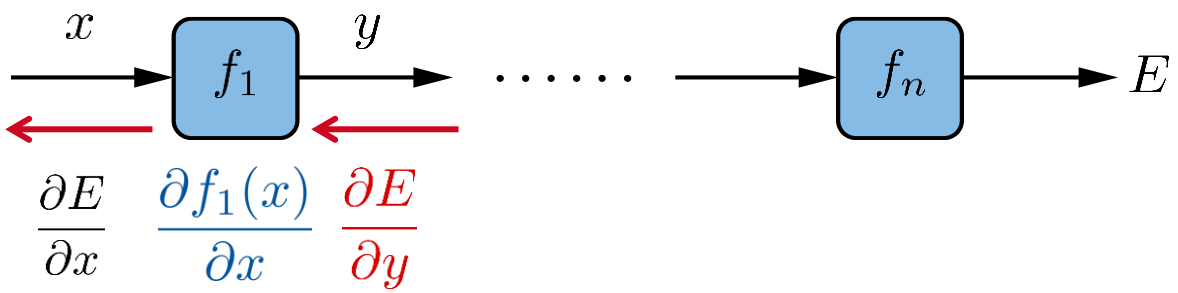
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$



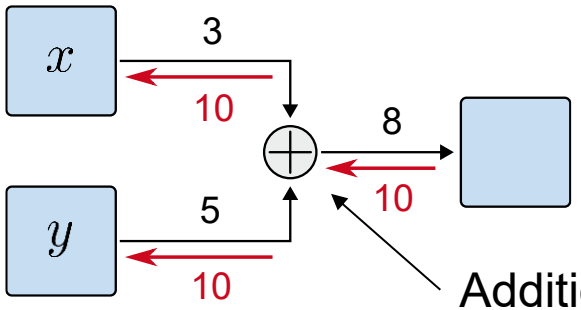
Backpropagation algorithm

Backpropagation calculates the gradient of the loss function with respect to each weight by propagating the error gradient backwards through the network.

$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial x} = \text{output gradient} \times \text{local gradient}$$



Example: Some Common Functions

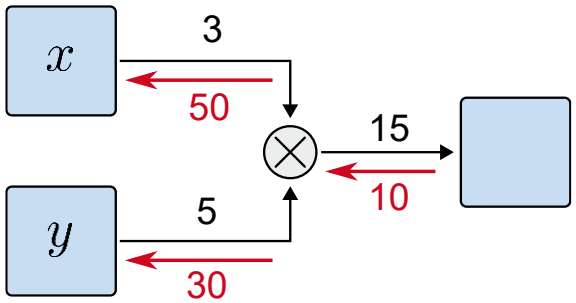


Value ■

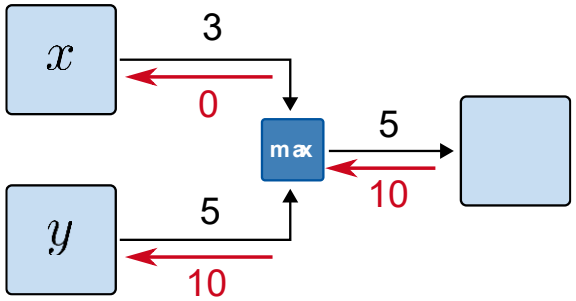
Gradient ■

$$f(x,y) = x + y \qquad \frac{\partial f}{\partial x} = 1,\ \frac{\partial f}{\partial y} = 1$$

Addition acts like a “gradient distributor” ⟶ Gradient flows through and is unchanged!



$$f(x,y) = xy \qquad \frac{\partial f}{\partial x} = y,\ \frac{\partial f}{\partial y} = x$$



$$f(x,y) = \max(x,y) \qquad \frac{\partial f}{\partial x} = \begin{cases} 1, & \text{if } x \geq y \\ 0, & \text{otherwise} \end{cases}, \quad \frac{\partial f}{\partial y} = \begin{cases} 1, & \text{if } x < y \\ 0, & \text{otherwise} \end{cases}$$

A Larger Example

$b = xy + z$

Goal: find $\frac{\partial b}{\partial x}$, $\frac{\partial b}{\partial y}$ and $\frac{\partial b}{\partial z}$ when

$x = 3,\ y = -5\ ,z = 2$

Step 1: Forward pass

Define intermediate values

$a = xy = -15$

$b = a + z = -13$

Step 2: Backward pass

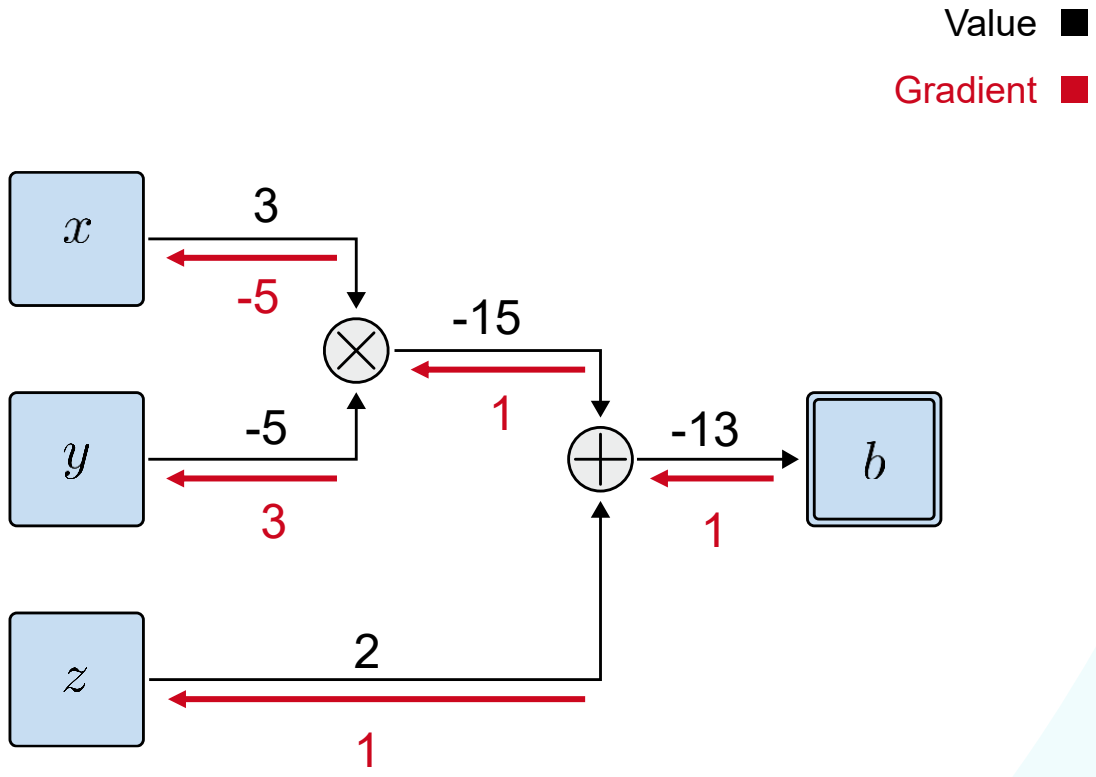
$\frac{\partial b}{\partial a} = 1$

$\frac{\partial b}{\partial z} = 1$

$\frac{\partial b}{\partial x} = \frac{\partial b}{\partial a}\frac{\partial a}{\partial x} = 1 \times (-5) = -5$

$\frac{\partial b}{\partial y} = \frac{\partial b}{\partial a}\frac{\partial a}{\partial y} = 1 \times 3 = 3$

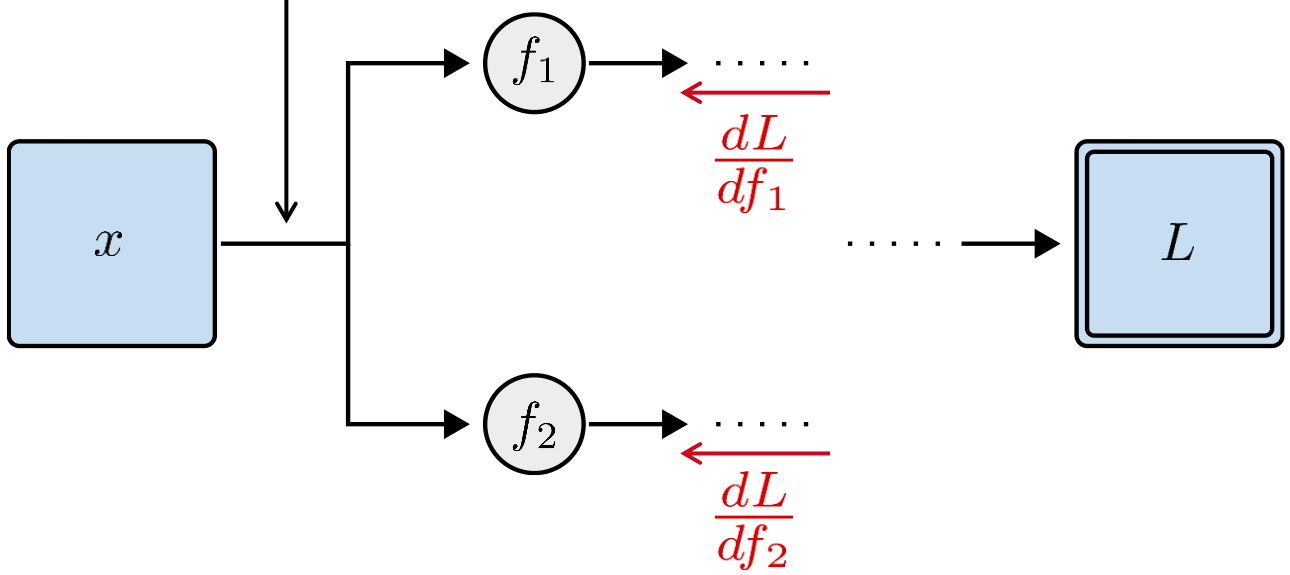
Done!



Multiple outputs

If there are more than one out-going edges at a node, we can just sum the gradients!

$$\frac{dL}{dx} = \frac{dL}{df_1} \cdot \frac{df_1}{dx} + \frac{dL}{df_2} \cdot \frac{df_2}{dx}$$



Discussion Backpropagation

Advantages

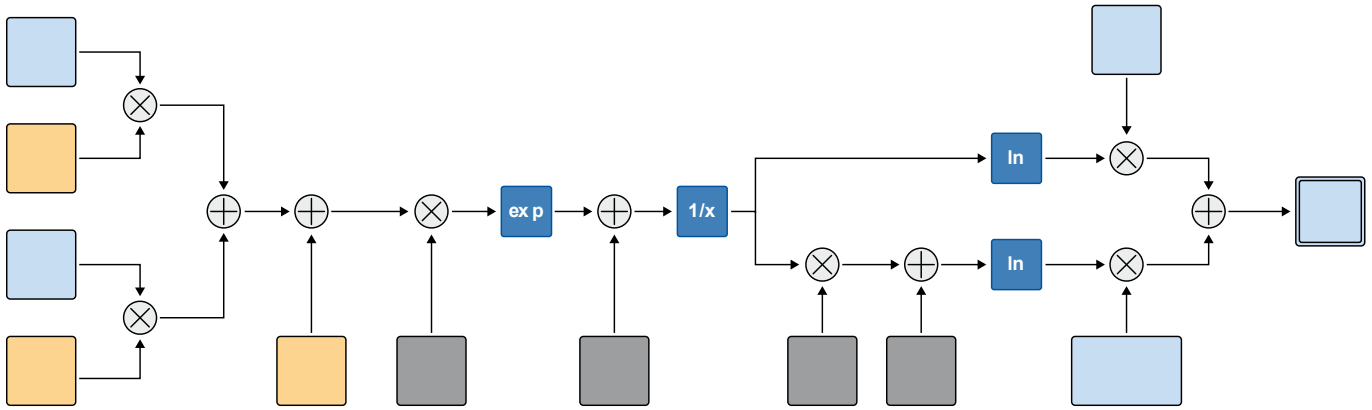
- Very general algorithm, widely used.
- Efficiently computes all gradients in the network using dynamic programming.
- The same concept can be applied to any differentiable function.
  - This makes it possible to define other types of layers.

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. **Computational Graphs**
6. Stochastic Gradient Descent

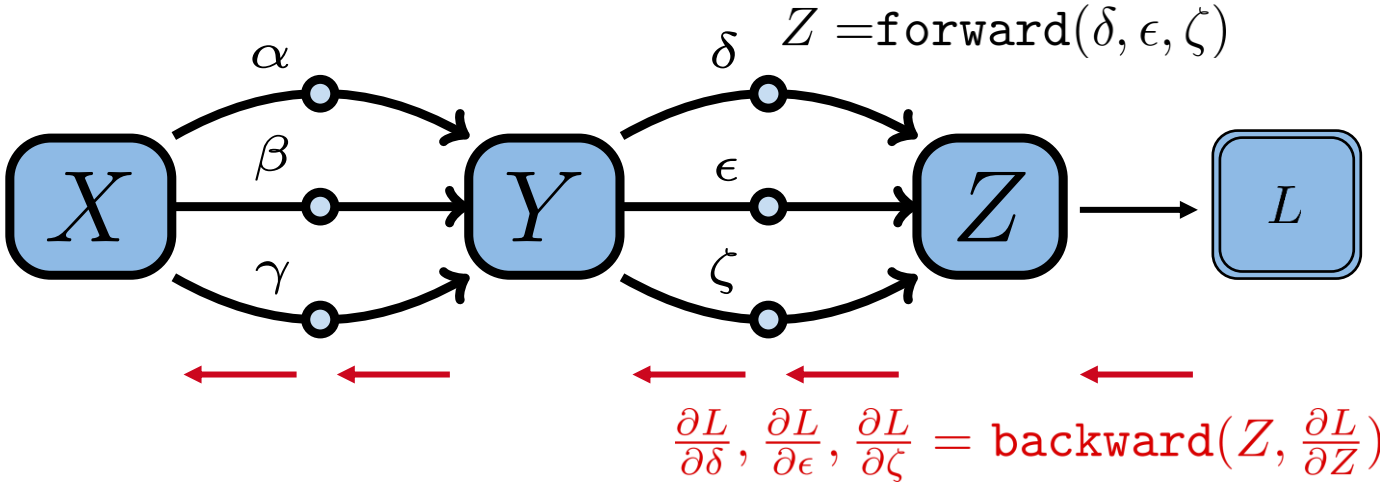


Backpropagation on Computational Graphs

- Neural networks define a computational graph computing some error L.
- Compute the weight's gradients via backpropagation.
- Only need to define two functions per node:

  $\texttt{forward}(\ldots)$ to compute the output.

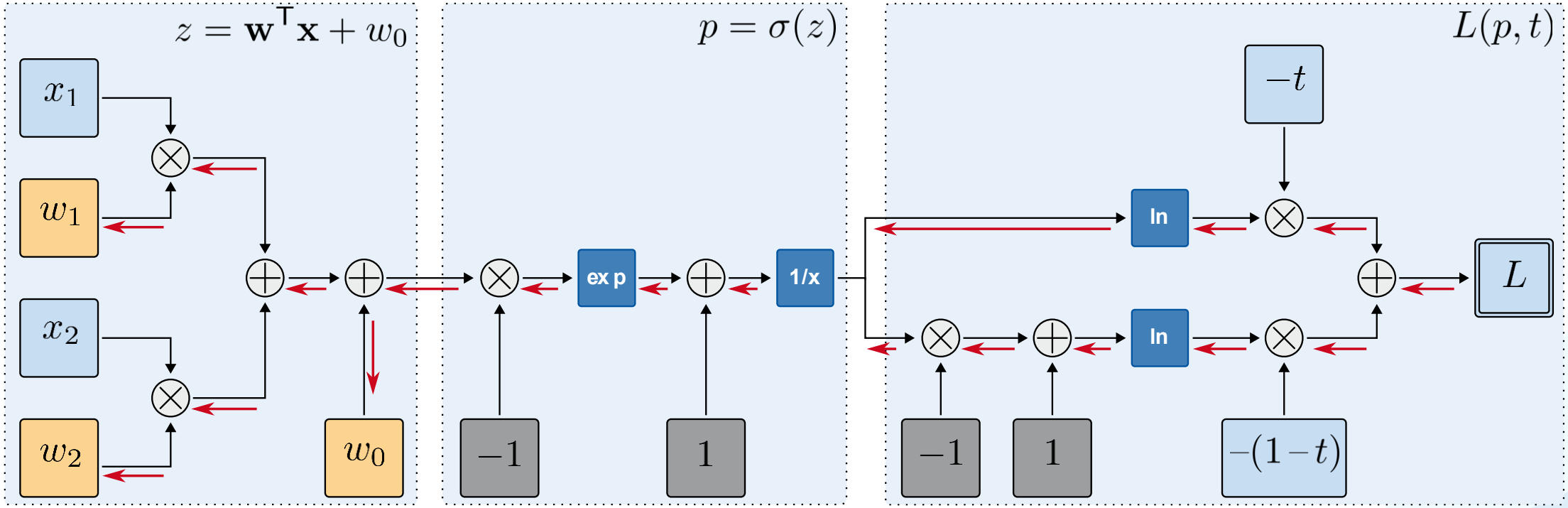
  $\texttt{backward}(\ldots)$ to compute the gradient.



Example: Logistic Regression

Model output: $p(\mathbf{x}) = \sigma(\mathbf{w}^\mathsf{T}\mathbf{x} + w_0) = 1/(1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)})$

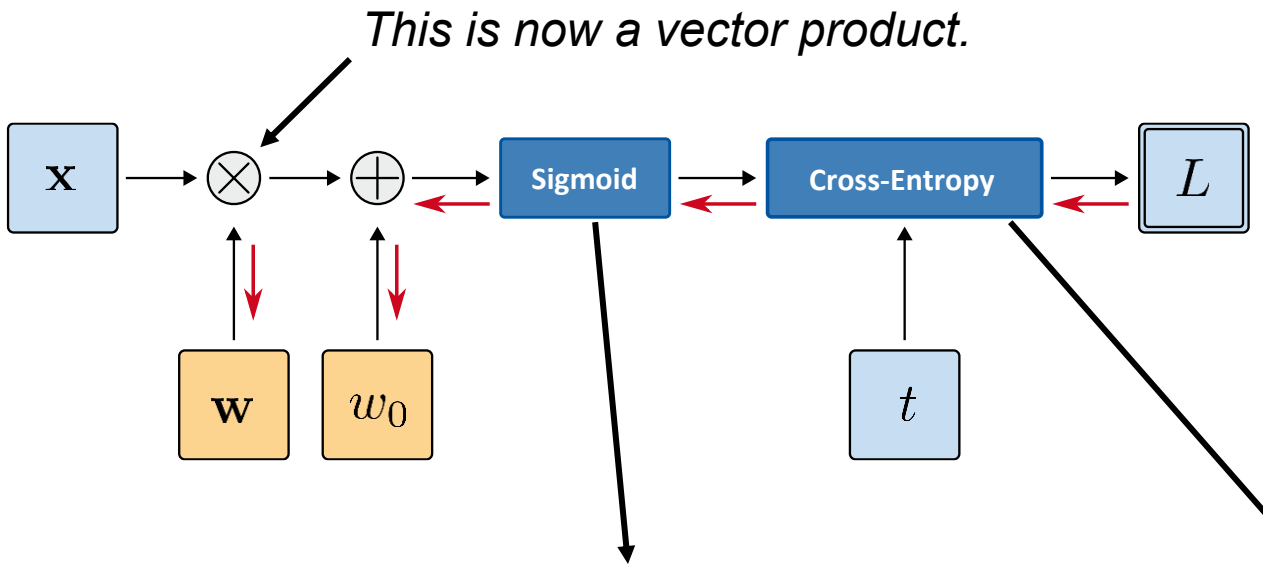
Loss function: $L(p, t) = \text{Cross-Entropy}(p, t) = -t \ln(p) - (1 - t) \ln(1 - p)$



Example: Logistic Regression

Model output: $p(\mathbf{x}) = \sigma(\mathbf{w}^\mathsf{T}\mathbf{x} + w_0)$

Loss function: $L(p, t) = \text{Cross-Entropy}(p, t)$



Grouping may simplify derivatives:

$$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1 - \sigma(a))$$

$$\frac{\partial L(p, t)}{\partial p} = -\frac{t}{p} + \frac{1 - t}{1 - p}$$

Example: Neural Network

With computational graphs for matrix and tensor operations, we can apply backpropagation to neural networks.

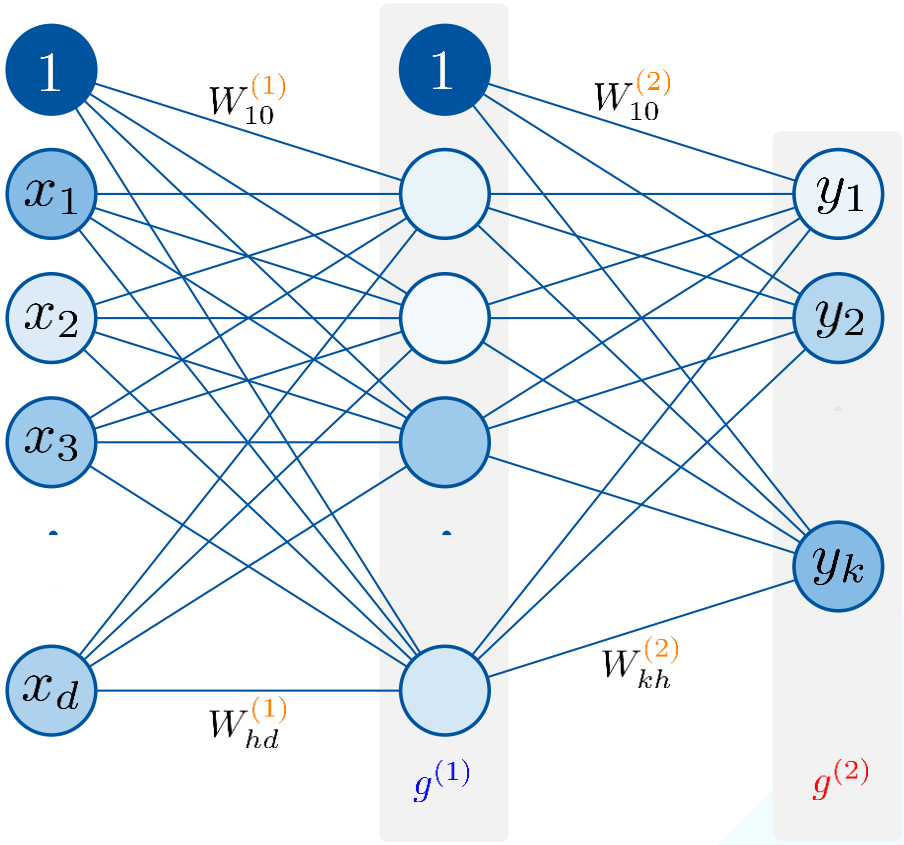
Recall the MLP example

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

where

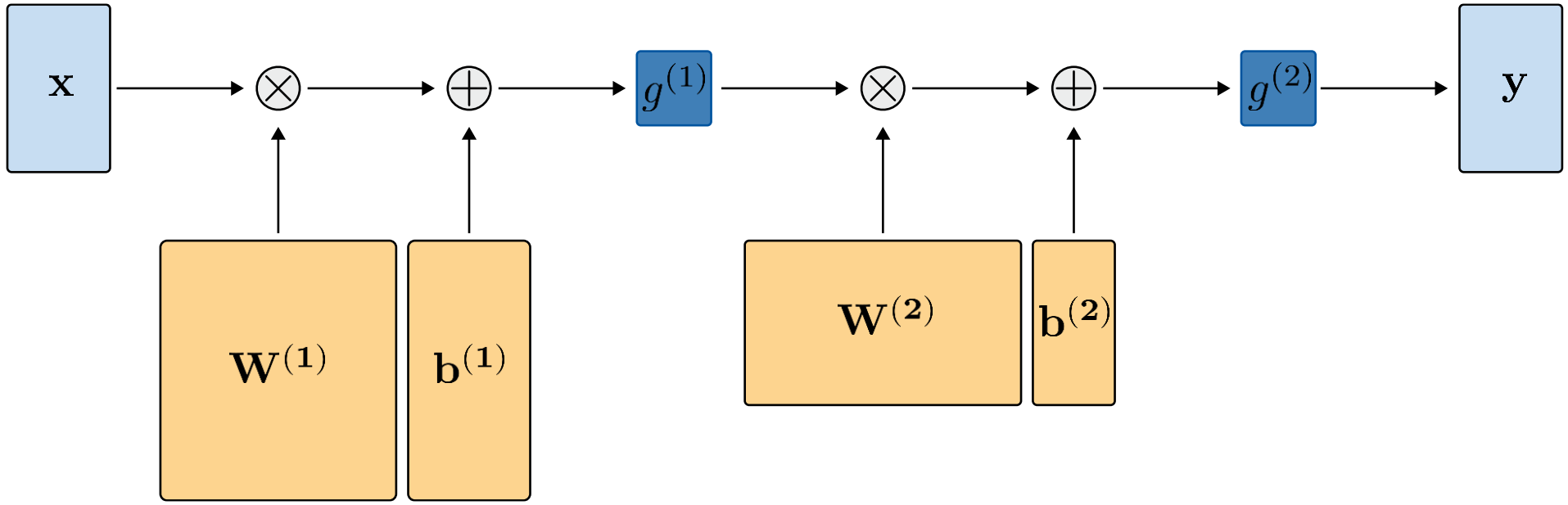
$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_k \end{bmatrix} \quad \mathbf{b}^{(1)} = \begin{bmatrix} W_{10}^{(1)} \\ \vdots \\ W_{h0}^{(1)} \end{bmatrix} \quad \mathbf{b}^{(2)} = \begin{bmatrix} W_{10}^{(2)} \\ \vdots \\ W_{k0}^{(2)} \end{bmatrix}$$

$$\mathbf{W}^{(1)} = \begin{bmatrix} W_{11}^{(1)} & \dots & W_{1d}^{(1)} \\ \vdots & & \vdots \\ W_{h1}^{(1)} & \dots & W_{hd}^{(1)} \end{bmatrix} \quad \mathbf{W}^{(2)} = \begin{bmatrix} W_{11}^{(2)} & \dots & W_{1h}^{(2)} \\ \vdots & & \vdots \\ W_{k1}^{(2)} & \dots & W_{kh}^{(2)} \end{bmatrix}$$



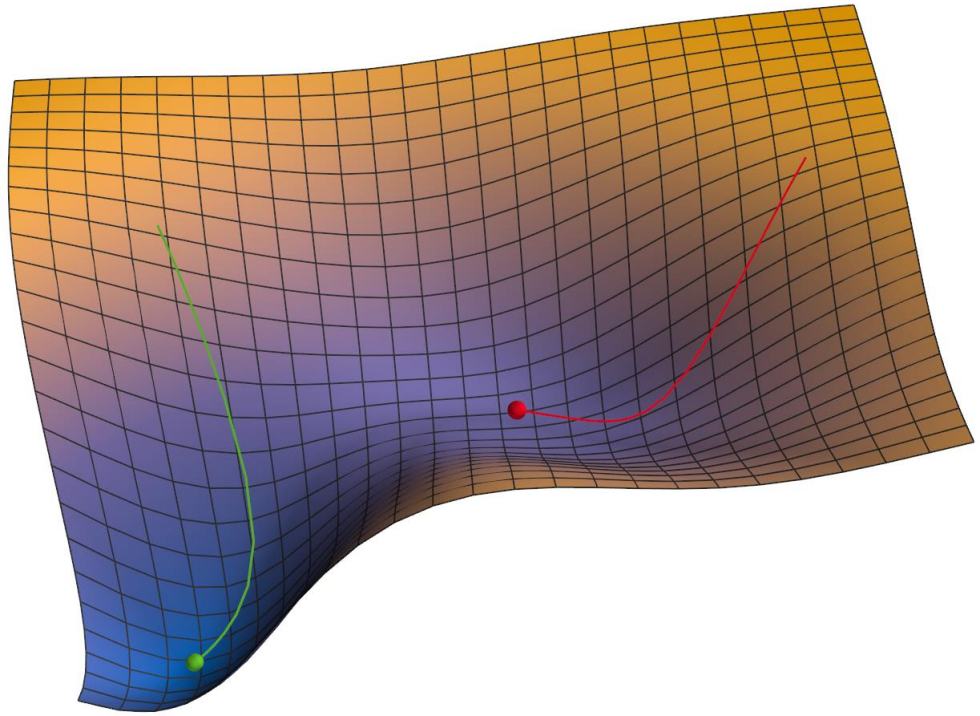
Example: Neural Network

The following computational graph represents such an MLP:



Neural Network Basics

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

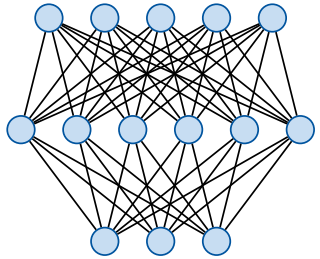


Stochastic Gradient Descent

- 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.

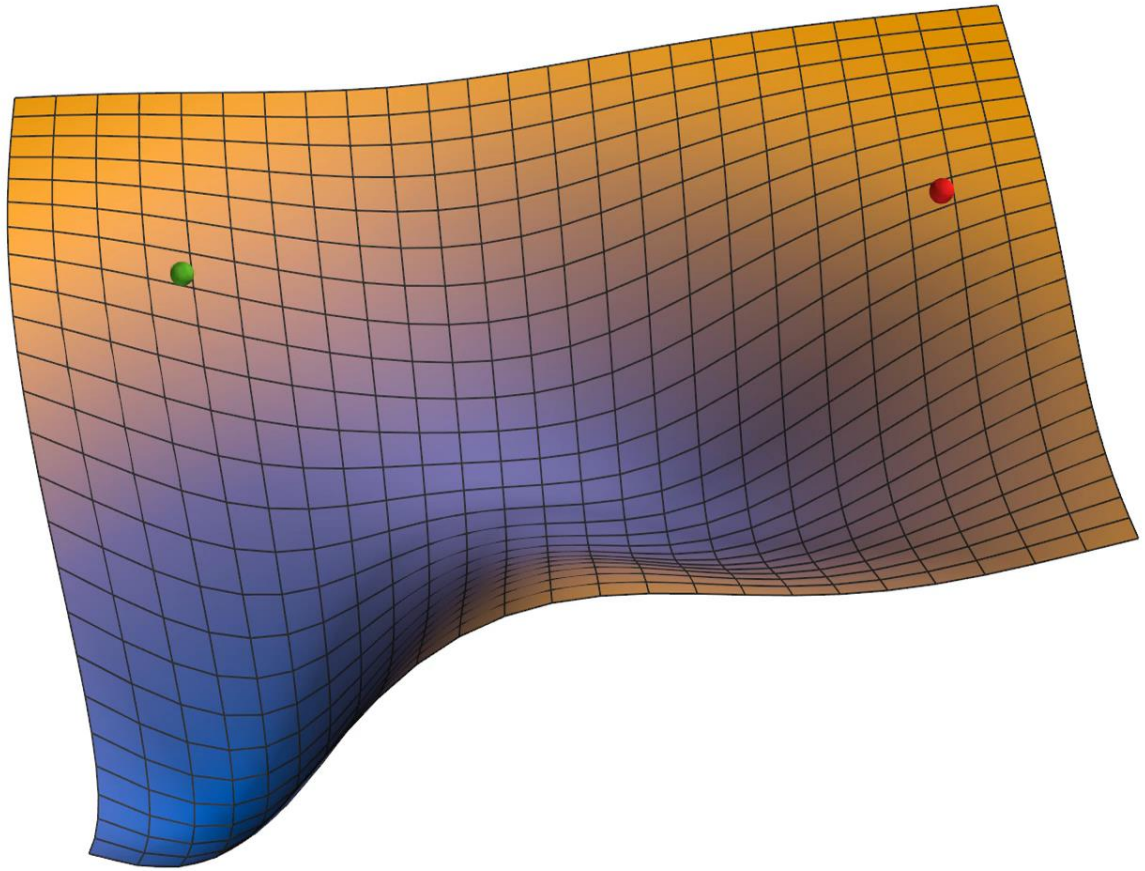
Middle ground: Minibatches

Minibatches

- Idea
  - Process only a small batch of training examples together. $\mathcal{B} = \{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_B, t_B)\} \subset \mathcal{D}$
  - Start with a small batch size & increase it as training proceeds.
- 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

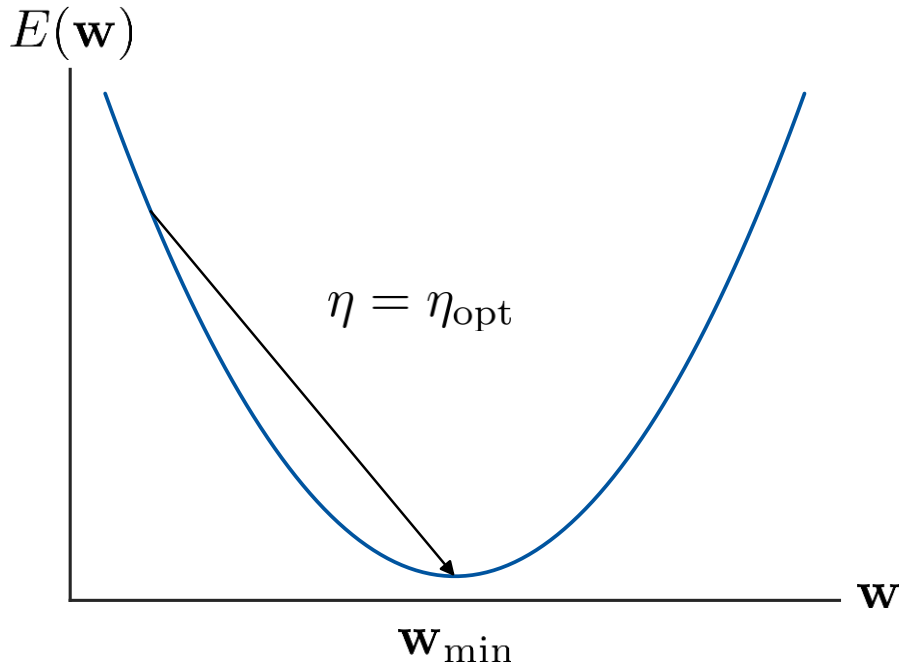


Choosing the Right Learning Rate

- 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}?

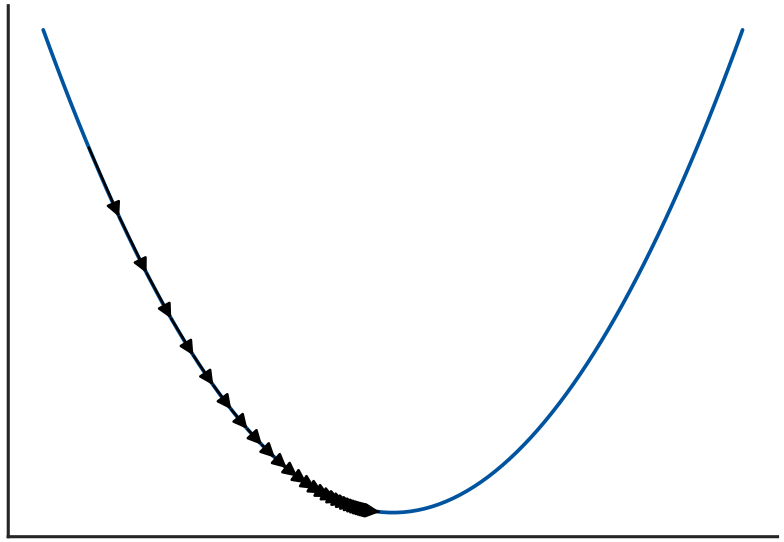


- If E is quadratic, the optimal learning rate is given by the inverse of the Hessian:

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

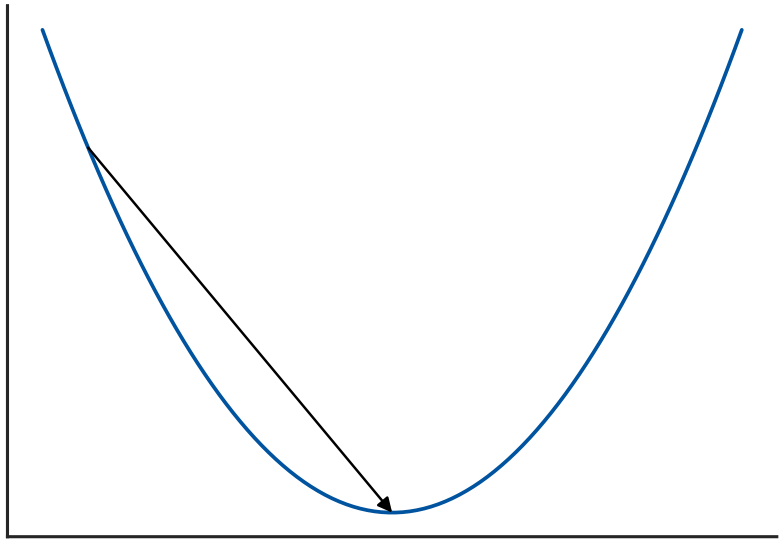
- *For neural networks, the Hessian is usually infeasible to compute.*

η too small



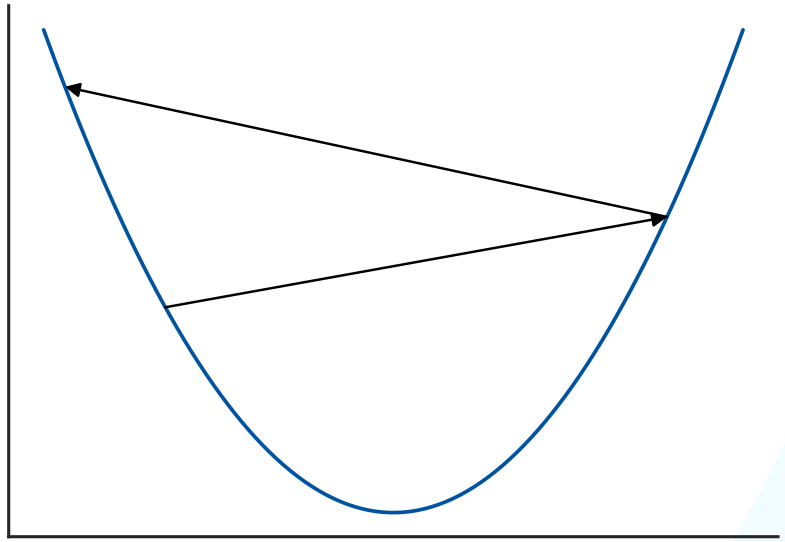
Convergence is slow

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/