Karlsruher Institut für Technologie

Steinbuch Centre for Computing

NHR SummerSchool 2023

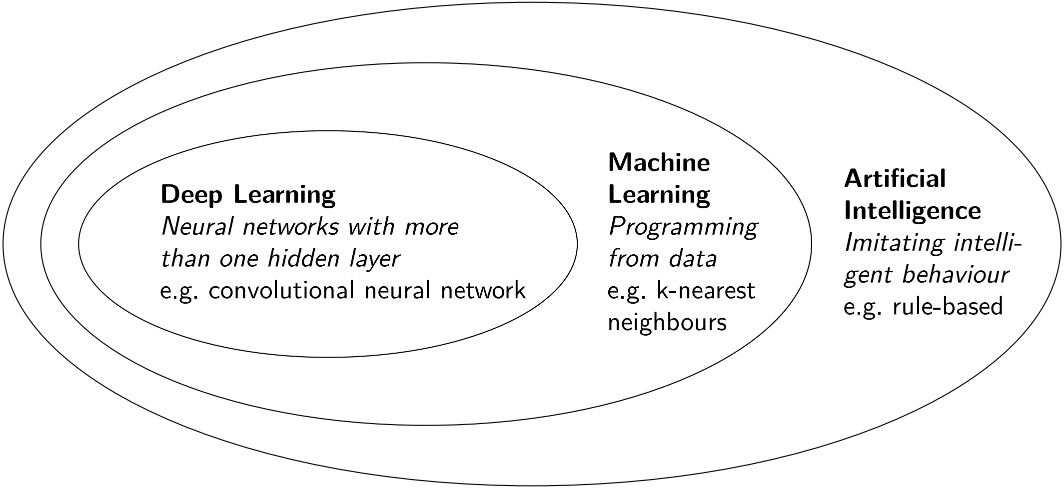
Data-Parallel Neural Networks

Dr. Charlotte Debus and Dr. Marie Weiel | June 15th, 2023

Part I

Introduction to Neural Networks

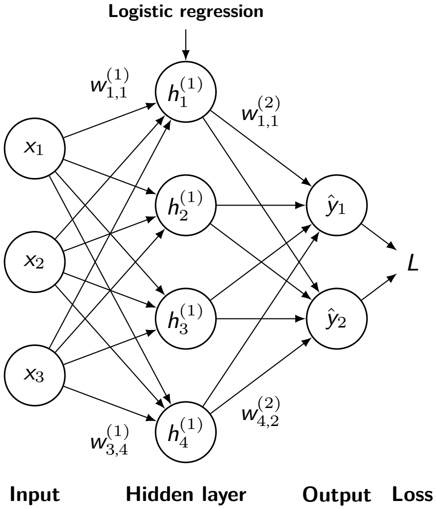
Artificial Intelligence



Artificial Neural Networks

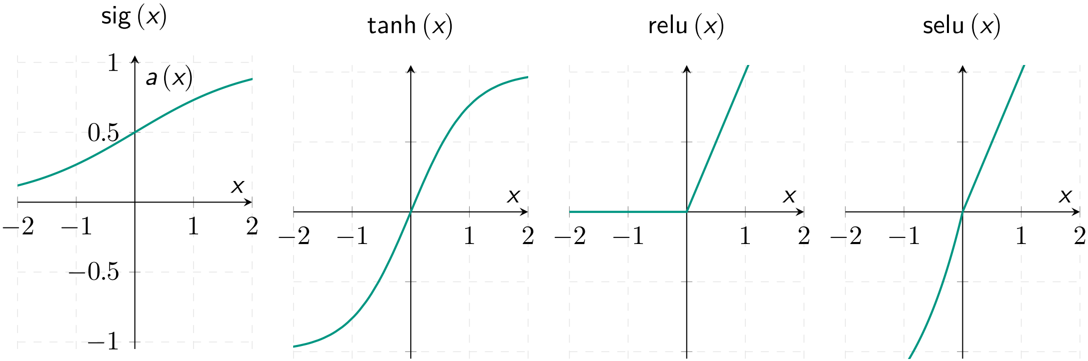
- Neural networks imitate biological behaviors [1].
  - **Neurons: smallest processing unit**
  - Graph of arithmetic operations
- **Weights** W: neuron connections, **free parameters** of the network
- Mathematical notation
  - $w_{i,j}^{(l)}$ – weight of input i wrt. neuron j, layer l
  - $a^{(l)}$ – activation function in layer l
  - $n_i^{(l)}$ – neural activation in layer l and neuron i
  - $h_j^{(l)}$ – hidden layer l, neuron j

$$h_j^{(l)} = a^{(l)}\left(n_i^{(l)}\right) = a^{(l)}\left(\Sigma_{i=1}^{n} w_{i,j}^{(l)} \cdot x_i\right)$$



Activation functions

- Activation function $a(x)$: **non-linearity** in neural networks.
- Improved learning capabilities

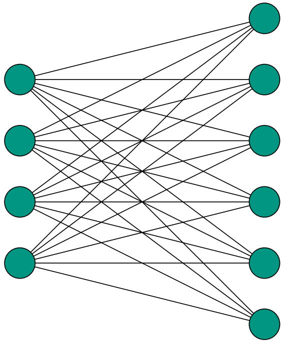


Layer Types

Fully Connected Layer

- Fully connected (FC): every neuron in layer i is connected to every neuron in layer $i+1$.
- Activation of a neuron: dot-product of input with weight (plus bias), passed through the activation function f:

$$a_i^{(h+1)} = f\left(\sum_j w_{ji} a_j^{(h)} + b_j\right)$$



Layer Types

Convolutional Layer

- Activation of a neuron via discrete convolution:
  - Filter of size *k* is applied to input via "sliding window" technique with *stride s* (inner product).
  - → Local surrounding of input determines activation
  - Filter size: receptive field
- Model weights = filter values (kernel).
- Fewer free (trainable) parameters than FC layer
- Image recognition: filter maps local image features (edges).

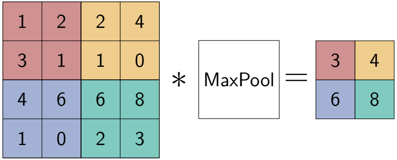
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} * \begin{bmatrix} 7 & 6 & 5 & 5 & 6 & 7 \\ 6 & 4 & 3 & 3 & 4 & 6 \\ 5 & 3 & 2 & 2 & 3 & 5 \\ 5 & 3 & 2 & 2 & 3 & 5 \\ 6 & 4 & 3 & 3 & 4 & 6 \\ 7 & 6 & 5 & 5 & 6 & 7 \end{bmatrix} = \begin{bmatrix} 2 & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

$$\begin{aligned} &0 \cdot 7 - 1 \cdot 6 + 0 \cdot 5 \\ &-1 \cdot 6 + 5 \cdot 4 - 1 \cdot 3 \\ &+0 \cdot 5 - 1 \cdot 3 + 0 \cdot 2 \\ &= 2 \end{aligned}$$

Layer Types

Pooling Layer

- Condensation of neurons, remove redundant information
- Stencil operations: Several neurons (pixels) are merged via a function.
  - Max-pooling: maximum value within receptive field is passed on



- Generally no loss of accuracy through "loss of information"
- Pooling layers have multiple advantages:
  - Reduced memory footprint, faster computation

  → Deeper networks for more complex tasks

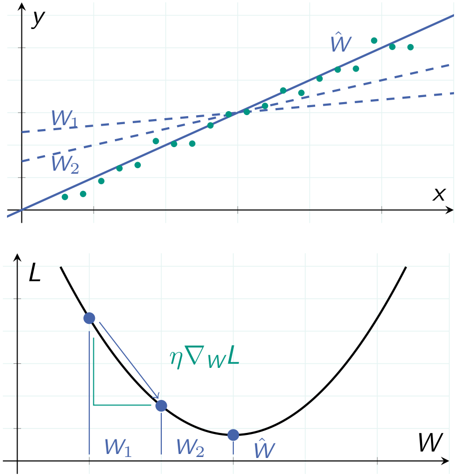
  - Receptive field grows automatically towards deeper layers, without increase of kernel size
  - Prevents overfitting

Determining W – Gradient Descent

- **Iterative approach** to determine W
$$W_{i+1} = W_i - \eta \nabla_W L$$

- **Random initial state**, e.g. $W_0 \neq 0$
- η is step size, called **learning rate**.

- Extensions and variants
  - **Standard** – gradient descent after every sample, batch size $B = 1$
  - **Stochastic** – randomized sample, $B = 1$
  - **Batch** – all samples, $B = |X|$
  - **Mini-Batch** – sample subset, $1 \leq B \leq |X|$



Backpropagation

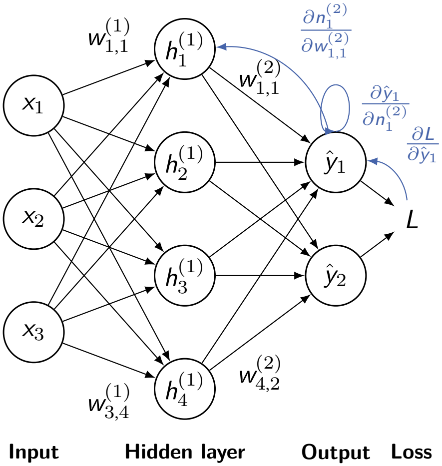
- **Algorithms** for calculating **gradients**
- Idea 1: **Divide** into **subproblems** – every weight with partial gradient.

$$\nabla_w L = \left(\frac{\partial L}{\partial w_{1,1}^{(1)}}, \frac{\partial L}{\partial w_{1,2}^{(1)}}, \dots, \frac{\partial L}{\partial w_{i,j}^{(l)}} \right)$$

- Idea 2: "denesting" of neurons via **chain rule**

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

- Solution from output to input



Backpropagation: Example I

- Example: partial derivative for weight $w_{1,1}^{(2)}$

$$\frac{\partial L}{\partial w_{1,1}^{(2)}} = \frac{\partial L}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial n_1^{(2)}} \frac{\partial n_1^{(2)}}{\partial w_{1,1}^{(2)}}$$

- Step 1: calculate $\frac{\partial L}{\partial \hat{y}_1}$.

$$\frac{\partial L}{\partial \hat{y}_1} = \frac{\partial}{\partial \hat{y}_1} \frac{1}{2} \sum_{i=1}^{2} (y_i - \hat{y}_i)^2$$

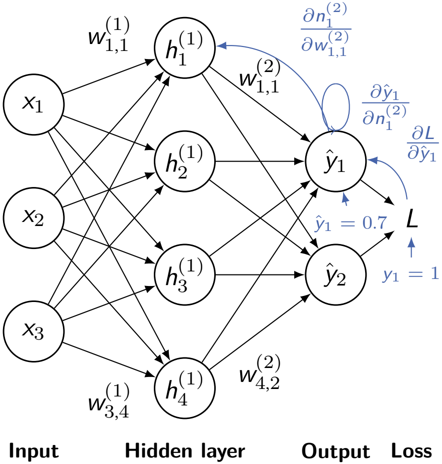
$$= \frac{\partial}{\partial \hat{y}_1} \left(\frac{1}{2} (y_1 - \hat{y}_1)^2 + \frac{1}{2} (y_2 - \hat{y}_2)^2 \right)$$

Derivatives for sums and constants

$$\frac{\partial L}{\partial \hat{y}_1} = \frac{\partial}{\partial \hat{y}_1} \frac{1}{2} (y_1 - \hat{y}_1)^2 + 0$$

Chain rule

$$\frac{\partial L}{\partial \hat{y}_1} = 2 \cdot \frac{1}{2} (y_1 - \hat{y}_1)^{2-1} \cdot (-1) = -(y_1 - \hat{y}_1) = -0.3$$



Backpropagation: Example I

- Example: partial derivative for weight $w_{1,1}^{(2)}$

$$\frac{\partial L}{\partial w_{1,1}^{(2)}} = \frac{\partial L}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial n_1^{(2)}} \frac{\partial n_1^{(2)}}{\partial w_{1,1}^{(2)}}$$

- Step 1: calculate $\frac{\partial L}{\partial \hat{y}_1}$.

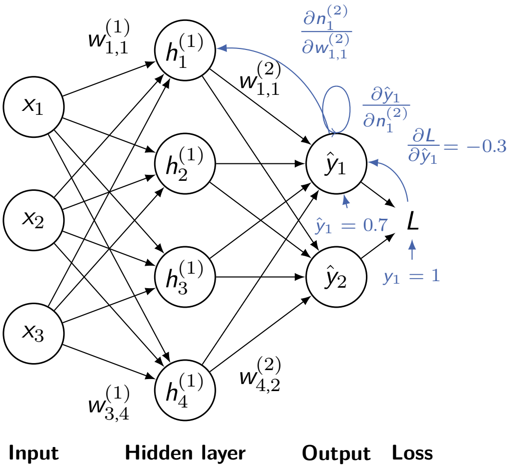
$$\frac{\partial L}{\partial \hat{y}_1} = \frac{\partial}{\partial \hat{y}_1} \frac{1}{2} \sum_{i=1}^{2} (y_i - \hat{y}_i)^2$$
$$= \frac{\partial}{\partial \hat{y}_1} \left(\frac{1}{2} (y_1 - \hat{y}_1)^2 + \frac{1}{2} (y_2 - \hat{y}_2)^2 \right)$$

Derivatives for sums and constants

$$\frac{\partial L}{\partial \hat{y}_1} = \frac{\partial}{\partial \hat{y}_1} \frac{1}{2} (y_1 - \hat{y}_1)^2 + 0$$

Chain rule

$$\frac{\partial L}{\partial \hat{y}_1} = 2 \cdot \frac{1}{2} (y_1 - \hat{y}_1)^{2-1} \cdot (-1) = -(y_1 - \hat{y}_1) = -0.3$$



Backpropagation: Example II

- Step 2: Calculate $\frac{\partial \hat{y}_1}{\partial n_1^{(2)}}$ with sigmoid activation function:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial \hat{y}_1}{\partial n_1^{(2)}} = \frac{\partial}{\partial n_1^{(2)}} \frac{1}{1+e^{-n_1^{(2)}}}$$

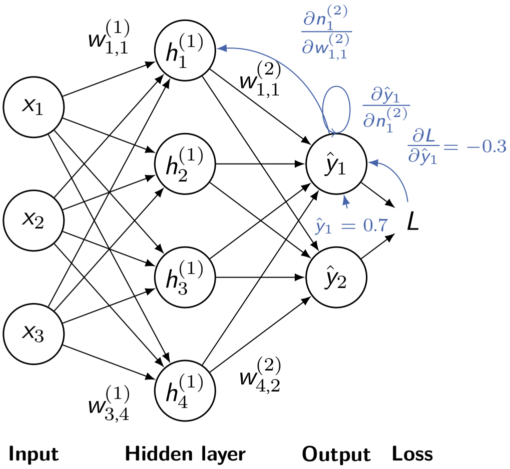
Inverse, chain rule

$$\frac{\partial \hat{y}_1}{\partial n_1^{(2)}} = \frac{\partial}{\partial n_1^{(2)}} \left(1+e^{-n_1^{(2)}}\right)^{-1}$$

$$= \left(1+e^{-n_1^{(2)}}\right)^{-2} \left(-e^{-n_1^{(2)}}\right)$$

$$\frac{\partial \hat{y}_1}{\partial n_1^{(2)}} = \sigma\left(n_1^{(2)}\right) * \left(1-\sigma\left(n_1^{(2)}\right)\right) = \hat{y}_1 * (1-\hat{y}_1)$$

$$\frac{\partial \hat{y}_1}{\partial n_1^{(2)}} = 0.7 \cdot (1-0.7) = 0.21$$



Backpropagation: Example II

- Step 2: Calculate $\frac{\partial \hat{y}_1}{\partial n_1^{(2)}}$ with sigmoid activation function:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial \hat{y}_1}{\partial n_1^{(2)}} = \frac{\partial}{\partial n_1^{(2)}} \frac{1}{1+e^{-n_1^{(2)}}}$$

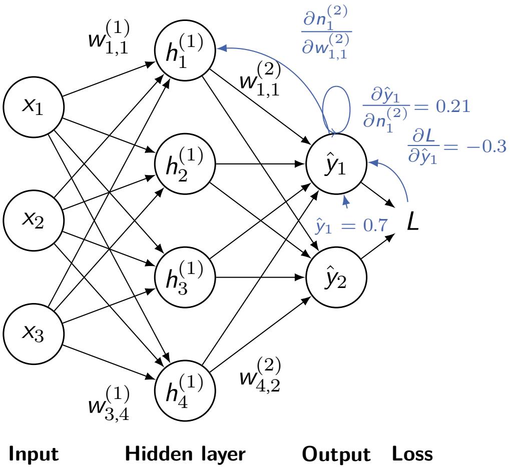
Inverse, chain rule

$$\frac{\partial \hat{y}_1}{\partial n_1^{(2)}} = \frac{\partial}{\partial n_1^{(2)}} \left(1+e^{-n_1^{(2)}}\right)^{-1}$$

$$= \left(1+e^{-n_1^{(2)}}\right)^{-2} \left(-e^{-n_1^{(2)}}\right)$$

$$\frac{\partial \hat{y}_1}{\partial n_1^{(2)}} = \sigma\left(n_1^{(2)}\right) * \left(1-\sigma\left(n_1^{(2)}\right)\right) = \hat{y}_1 * (1-\hat{y}_1)$$

$$\frac{\partial \hat{y}_1}{\partial n_1^{(2)}} = 0.7 \cdot (1-0.7) = 0.21$$



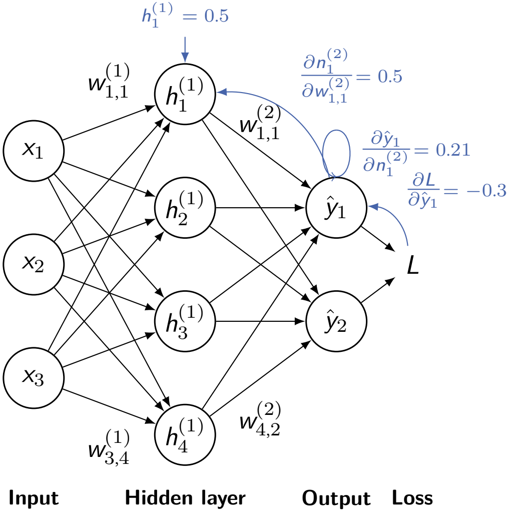
Backpropagation: Example III

- Step 3: Calculate $\frac{\partial n_1^{(2)}}{\partial w_{1,1}^{(2)}}$.

$$\frac{\partial n_1^{(2)}}{\partial w_{1,1}^{(2)}} = \frac{\partial}{\partial w_{1,1}^{(2)}} \sum_{i=1}^{4} w_{i,1}^{(2)} * h_i^{(1)}$$

Derivatives for sums and constants

$$\frac{\partial n_1^{(2)}}{\partial w_{1,1}^{(2)}} = \frac{\partial}{\partial w_{1,1}^{(2)}} w_{1,1}^{(2)} * h_1^{(1)} = h_1^{(1)} = 0.5$$



Backpropagation: Example III

- Step 3: Calculate $\frac{\partial n_1^{(2)}}{\partial w_{1,1}^{(2)}}$.

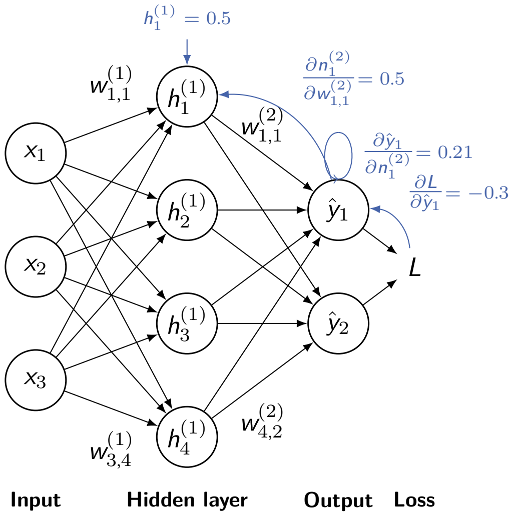
$$\frac{\partial n_1^{(2)}}{\partial w_{1,1}^{(2)}} = \frac{\partial}{\partial w_{1,1}^{(2)}} \sum_{i=1}^{4} w_{i,1}^{(2)} * h_i^{(1)}$$

Derivatives for sums and constants

$$\frac{\partial n_1^{(2)}}{\partial w_{1,1}^{(2)}} = \frac{\partial}{\partial w_{1,1}^{(2)}} w_{1,1}^{(2)} * h_1^{(1)} = h_1^{(1)} = 0.5$$

- Step 4: merge partial results.

$$\frac{L}{w_{1,1}^{(2)}} = \frac{\partial L}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial n_1^{(2)}} \frac{\partial n_1^{(2)}}{\partial w_{1,1}^{(2)}} = -0.3 \cdot 0.21 \cdot 0.5 = -0.0315$$



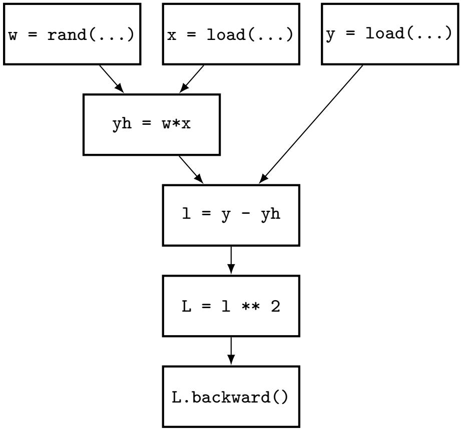
Optimizers

- Learning rate η is essential for convergence and generalization
- Variations of gradient descent: adaptive η (heuristic).
- In practice: faster training, but SGD with suitable LR yields better performance

Method	Weight update	Definitions
Gradient Descent	$w_{t+1} = w_t - \eta \nabla_{w_t} L$	
Adaptive Learning Rate	$w_{t+1} = w_t - \eta_t \nabla_{w_t} L$	
Momentum [2]	$w_{t+1} = w_t + \mu \left(w_t - w_{t-1} \right) - \eta \nabla_{w_t} L$	
Adagrad [3]	$w_{i,t+1} = w_{i,t} + \eta \frac{\nabla_{w_{i,t}} L}{\sqrt{A_{i,t}} + \varepsilon}$	$A_{i,t} = \sum \tau = 0^t \left(\nabla_{w_{i,t-\tau}} L \right)^2$
RMSprop [4]	$w_{i,t+1} = w_{i,t} + \eta \frac{\nabla_{w_{i,t} L} L}{\sqrt{B_{i,t}} + \varepsilon}$	$B_{i,t} = \beta B_{i,t-1} + (1 - \beta) \left(\nabla_{w_{i,t}} L \right)^2$
Adam [5]	$w_{i,t+1} = w_{i,t} + \eta \frac{C_{i,t}^{(1)}}{\sqrt{C_{i,t}^{(2)}} + \varepsilon}$	$C_{i,t}^{(m)} = \frac{\beta_m C_{i,t-1} + (1-\beta_m) \left(\nabla_{w_{i,t}} L \right)^M}{1 - \beta_m^t}$

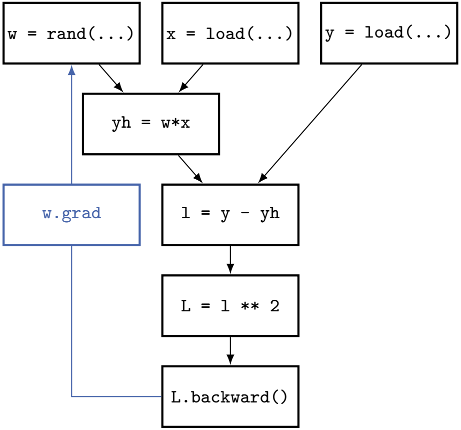
Automatic Differentiation (AD)

- Practical application of backpropagation
  - Efficient, since it only propagates partial derivatives
  - Multiple weights in a layer as matrices
- Backpropagation by hand is prone to errors.
- **Automatic Differentiation (AD)**: technique to generate derivatives in a program.
  - Atomic operations $(+, -, *, /)$ and certain functions $(\sin, \exp, \max)$ with explicit gradients.
  - Combination via chain rule.
  - Common implementations: `TensorFlow` [6], `PyTorch` [7],...



Automatic Differentiation (AD)

- Practical application of backpropagation
  - Efficient, since it only propagates partial derivatives
  - Multiple weights in a layer as matrices
- Backpropagation by hand is prone to errors.
- **Automatic Differentiation (AD)**: technique to generate derivatives in a program.
  - Atomic operations $(+, -, *, /)$ and certain functions $(\sin, \exp, \max)$ with explicit gradients.
  - Combination via chain rule.
  - Common implementations: `TensorFlow` [6], `PyTorch` [7],...



How to Train a Neural Network in PyTorch

1. **The data**
   - Can be loaded via PyTorch's DataSet and DataLoader class (in torch.utils.data)
   - Common datasets in torchvision

```
import torch
import torchvision

trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True,download=
    True)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=batch_size,
    shuffle=True, num_workers=2)
```

How to Train a Neural Network in PyTorch

2. The Model

- Different layer types via `torch.nn`
- `init` and `forward` need to be implemented by user

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc = nn.Linear(16 * 5 * 5, 120)
        self.out = nn.Linear(120, 2)


    def forward(self, x):
        x = self.pool(F.relu(self.conv(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc(x))
        x = self.out(x)
        return x
```

How to Train a Neural Network in PyTorch

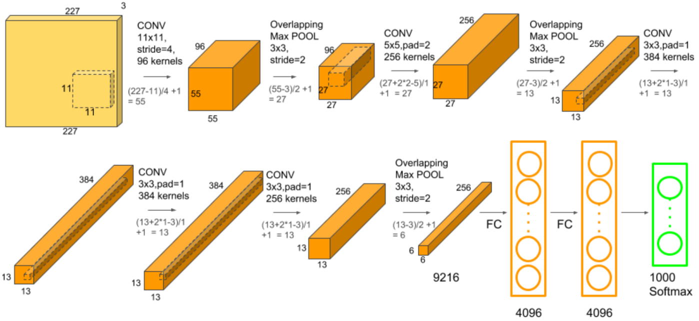
3. The training loop

- Define loss function and optimizer
- Loop over epochs
- For each mini-batch in DataLoader (torch.utils.data)
  - Initialize gradients (optimizer.zero_grad())
  - Pass samples through model
  - Calculate loss between model output and targets
  - Calculate gradients (loss.backwards())
  - Optimizer updates model weights based on gradients (optimizer.step())

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr
    =0.001, momentum=0.9)

for epoch in range(2):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

AlexNet

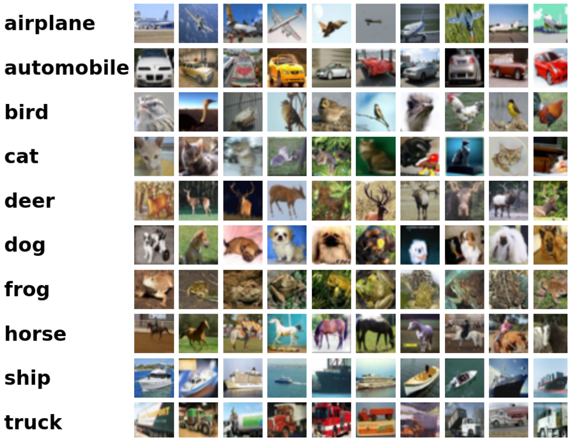
- Classification model
- Winner of the ImageNet Challenge 2012
- Convolutional Neural Network
  - 5 convolutional layers (incl. max-pooling)
  - 3 fully connected layers



Source: https://learnopencv.com/understanding-alexnet/

CIFAR-10 Dataset

- $60\,000$ colored images (RGB) with 32×32 pixels
- 10 classes, $6\,000$ images per class
- $50\,000$ training samples
- $10\,000$ test samples



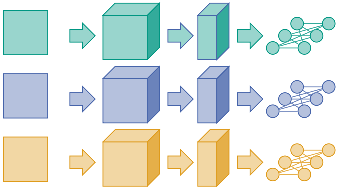
Source: https://www.cs.toronto.edu/~kriz/cifar.html

Part II

Scalable Deep Learning

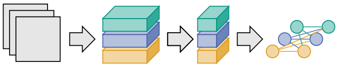
Parallelization of Neural Networks

Data Parallelism



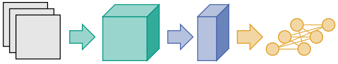
- Copy of the model on each processor
- Data $\mathcal{X}$ disjoint subsets $S_p = \left\{\vec{x}_{p\cdot N/P}, ..., \vec{x}_{(p+1)\cdot N/P}\right\}$ distributed across processors

Model Parallelism



- Model is distributed, i.e. each processor holds subset W_p of model weights

Pipelining



- Special type of model parallelism
- Connected parts of the model (e.g. full layer) are distributed across processors

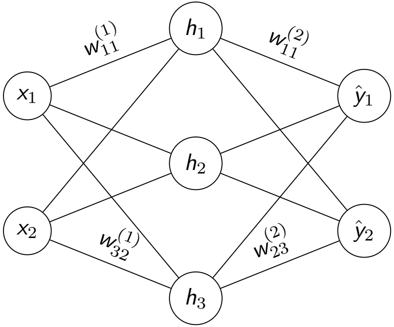
A Simple Neural Network

- 2 input neurons, 3 hidden neurons, 2 output neurons

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \vec{h} = \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} \quad \vec{y}_{\text{pred}} = \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \end{pmatrix}$$

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} & b_1^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & b_2^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & b_3^{(1)} \end{pmatrix}$$

$$W^{(2)} = \begin{pmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} & b_1^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} & w_{23}^{(2)} & b_2^{(2)} \end{pmatrix}$$



$$\vec{h} = \sigma\left(\vec{n}^{(1)}\right) = \sigma\left(W^{(1)} \cdot \vec{x}\right)$$

$$\vec{y}_{\text{pred}} = \sigma\left(\vec{n}^{(2)}\right) = \sigma\left(W^{(2)} \cdot \vec{h}\right)$$

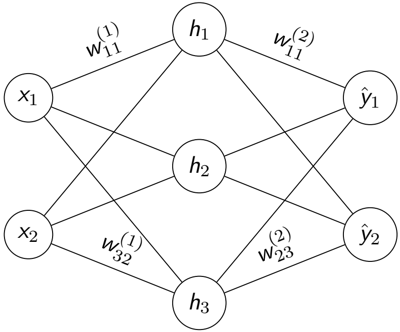
Gradient Calculation for Weight Updates

$$\frac{\partial L}{\partial \hat{y}_1} = -(\hat{y}_1 - y_1), \qquad \frac{\partial \hat{y}_1}{\partial n_1^{(2)}} = \hat{y}_1 \cdot (1 - \hat{y}_1)$$

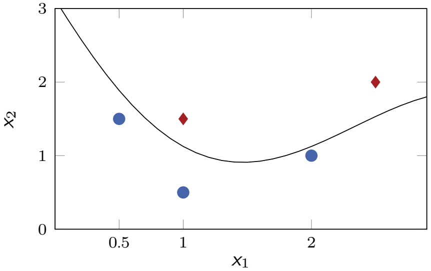
$$\frac{\partial n_1^{(2)}}{\partial h_1} = w_{11}^{(2)}, \qquad \frac{\partial h_1}{\partial n_1^{(1)}} = h_1 \cdot (1 - h_1)$$

$$\frac{\partial n_1^{(1)}}{\partial w_{11}^{(1)}} = x_1$$

$$\frac{\partial L}{\partial w_{11}^{(1)}} = -(\hat{y}_1 - y_1) \cdot \hat{y}_1 \cdot (1 - \hat{y}_1) \cdot w_{11}^{(2)} \cdot h_1 \cdot (1 - h_1) \cdot x_1$$



Example: Gradient Calculation for Weight Updates



$$X = \begin{pmatrix} 0.5 & 1.5 \\ 2 & 1 \\ 1 & 1.5 \\ 1 & 0.5 \\ 2.5 & 2 \end{pmatrix} \quad Y = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$W^{(1)} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$$\vec{y}_{pred} = \begin{pmatrix} 0.941 & 0.873 \\ 0.950 & 0.868 \\ 0.943 & 0.860 \\ 0.948 & 0.868 \\ 0.951 & 0.876 \end{pmatrix} \quad \frac{\partial L_i}{\partial w_{11}^{(1)}} = \begin{pmatrix} -0.00134 \\ -0.00461 \\ -0.00279 \\ 0.06524 \\ 0.10746 \end{pmatrix}$$

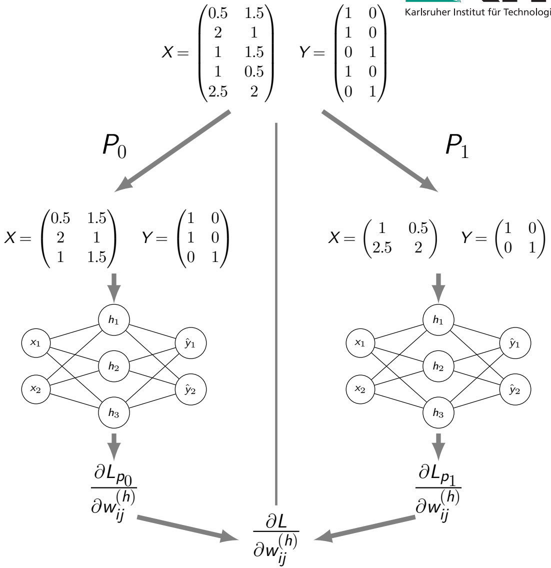
$$\frac{\partial L}{\partial w_{11}^{(1)}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L_i}{\partial w_{11}^{(1)}} = \mathbf{0.03281}$$

Data-Parallel Neural Networks

- Data are distributed subsets (chunks) on processors.
- Each process conducts forward-backward pass on its data with local copy of the network.
- After each forward-backward pass, model weights are synchronized across all processes.

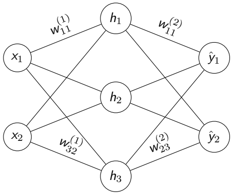
→ Communication, averaging of gradients

- After synchronization all local copies of the model are identical.



Data-Parallel Neural Networks

Process P_0

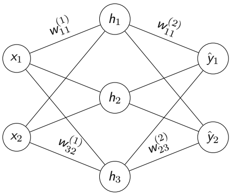


$$X = \begin{pmatrix} 0.5 & 1.5 \\ 2 & 1 \\ 1 & 1.5 \end{pmatrix}$$

$$Y = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\Rightarrow \frac{\partial L_{p_0}}{\partial w_{11}^{(1)}} = -0.00288$$

Process P_1



$$X = \begin{pmatrix} 1 & 0.5 \\ 2.5 & 2 \end{pmatrix}$$

$$Y = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\Rightarrow \frac{\partial L_{p_1}}{\partial w_{11}^{(1)}} = 0.08635$$

$\Longrightarrow$ Communication: $\frac{\partial L_p}{\partial w_{11}^{(1)}}$

$$\frac{\partial L}{\partial w_{11}^{(1)}} = \frac{1}{2}\left(\frac{\partial L_{p_0}}{\partial w_{11}^{(1)}} + \frac{\partial L_{p_1}}{\partial w_{11}^{(1)}}\right) = \mathbf{0.04173}$$

Data-Parallel Neural Networks

Karlsruher Institut für Technologie

1. Initialize network weights.
   - Root process creates initial model weights and broadcasts them to all other processes.
   - All processes create initial model weights using the same random seed.
2. Load data.
   - Each process loads disjoint subset = mini-batch.
3. Shuffle data.
   - Distributed shuffling (independent and identically distributed data)
4. For each epoch until converged:

   4.1 Local forward-backward pass
   - Calculate local gradients based on local mini-batch.

   4.2 Synchronize gradients and update model weights.
   - Overall gradient is average across all processes

   $\rightarrow$ Via **parameter server** (worker-master scheme)

   $\rightarrow$ Via MPI **Allreduce**

Algorithm: Data-Parallel Neural Networks

Algorithm 1: Dataparallel Training.

Input : X, Y – Data matrix with N samples, P – Number of process, p – process rank

```
model = NeuralNetwork();
model.weights.initialize(seed);
start = p * N // P;
stop = (p+1) * N // P;
chunk = (X[start : stop], Y[start : stop]);
local_data = shuffle_data(chunk);
while not converged(nn.weights) do
    prediction = model.forward(local_data);
    loss = L(prediction, Y);
    dw_p = loss.backwards();
    dW = MPI.Allreduce(dw_p);
    model.weights.update(dW);
```

PyTorch DistributedDataParallel

Karlsruher Institut für Technologie

- PyTorch module for data-parallel neural networks with multiple processes
  - Not the same as torch.DataParallel, which provides threading.
- torch.distributed for PyTorch support and basic communication for multi-process-parallelism (multi-node)
- torch.DistributedDataParallel builds on torch.distributed for *synchronous* distributed training of PyTorch models
  - Wrapper of existing PyTorch model
  - Automatic synchronization of gradients
- User has to start copy of main on each process.
- Each process has its own optimizer and performs complete update step in every iteration.

PyTorch DistributedDataParallel

Back-end	gloo		mpi		nccl	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✗	✓	?	✗	✗
recv	✓	✗	✓	?	✗	✗
broadcast	✓	✓	✓	?	✗	✓
allreduce	✓	✓	✓	?	✗	✓
reduce	✓	✗	✓	?	✗	✓
allgather	✓	✗	✓	?	✗	✓
gather	✓	✗	✓	?	✗	✗
scatter	✓	✗	✓	?	✗	✗
reduce_scatter	✗	✗	✗	✗	✗	✓
alltoall	✗	✗	✓	?	✗	✓
barrier	✓	✗	✓	?	✗	✓

PyTorch DistributedDataParallel

Karlsruher Institut für Technologie

- Initialization of multi-process environment:
  `dist.init_process_group(backend, rank, world_size)`

→ Each process must know
  1. Which process is the root.
  2. Which rank it has.
  3. How many processes there are in total.
  4. Which communication back-end is used (Gloo, MPI, NCCL).
- Clean-up at the end of training:
  `dist.destroy_process_group()`
- Wrapper for single-process model:
  `model = nn.parallel.DistributedDataParallel(model, device_ids)`
- For data-parallelism load distributed mini-batch:
  `torch.utils.data.distributed.DistributedSampler`

- **CAUTION**: `mp.spawn` should NOT be used on the cluster, `srun` starts script on each process (double spawning)

Summary

Karlsruher Institut für Technologie

Data-parallel training

- is the most widely used form of scalable neural network training
  - Sample/batch-parallel (most common)
  - Feature-Parallelism (Domain parallelsim)
- is utilized if
  a. Individual data samples are very large →**Memory constraints**
  b. Training takes a long time due to a large number of samples → **Speed-up**
- is an approximation of serial single-process training
  - IID
  - Large mini-batch effects

Each process

- holds a **model instance** (copy of the same model)
  → Model needs to fit into memory of single process
- trains its model instance on a subset of the data (**distributed mini-batch**)

After each forward-backward pass **gradients are averaged across all processes**

- Parameter server: Asynchronous SGD, Stale Gradients
- `Allreduce`: Tree-Allreduce vs. Ring-Allreduce

→ Communication after each batch/epoch

Literatur I

Karlsruher Institut für Technologie

[1] W. Ertel und N. T. Black, *Grundkurs Künstliche Intelligenz*. Springer, 2016.

[2] N. Qian, „On the momentum term in gradient descent learning algorithms,“ *Neural networks*, Jg. 12, Nr. 1, S. 145–151, 1999.

[3] J. Duchi, E. Hazan und Y. Singer, „Adaptive subgradient methods for online learning and stochastic optimization.,“ *Journal of machine learning research*, Jg. 12, Nr. 7, 2011.

[4] G. Hinton, N. Srivastava und K. Swersky, „Neural networks for machine learning lecture 6a overview of mini-batch gradient descent,“ *Cited on*, Jg. 14, Nr. 8, S. 2, 2012.

[5] D. P. Kingma und J. Ba, „Adam: A method for stochastic optimization,“ *arXiv preprint arXiv:1412.6980*, 2014.

[6] Martin Abadi, Ashish Agarwal, Paul Barham u. a., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, [accessed at 2021-08-04], 2015. Adresse: `http://tensorflow.org/`.

Literatur II

[7] A. Paszke, S. Gross, F. Massa, A. Lerer u. a., „PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, S. 8024–8035. Adresse: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.