Deep Learning

Lecture 3 – Deep Neural Networks

Sebastian Bordt
Theory of Machine Learning Group
University of Tübingen







Exam Quizzes

Quiz	Opens	Closes
Exam Quiz 1	Wed 28.10.	Wed 04.11., 14:00
Exam Quiz 2	Wed 11.11.	Wed 25.11., 14:00
Exam Quiz 3	Wed 02.12.	Wed 16.12., 14:00
Exam Quiz 4	Wed 13.01.	Wed 20.01., 14:00
Exam Quiz 5	Wed 20.01.	Wed 27.01., 14:00

- Each exam quiz opens after the lecture and is on **Ilias**
- **To take the exam, you must participate in at least 3 of the 5 exam quizzes!**

Agenda

3.1 Backpropagation with Tensors

3.2 Multi-Layer Perceptrons

3.1
Backpropagation with Tensors

Recap: Backpropagation with Scalars

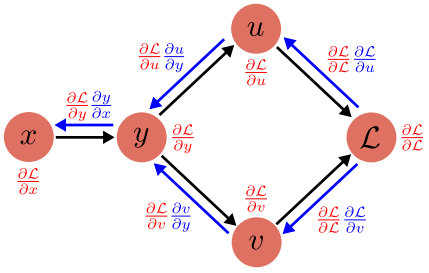
Forward Pass:

$$
\begin{aligned}
&(1) \quad y = y(x) \\
&(2) \quad u = u(y) \\
&(2) \quad v = v(y) \\
&(3) \quad \mathcal{L} = \mathcal{L}(u, v)
\end{aligned}
$$

Backward Pass:

$$
\begin{aligned}
&(3) \quad \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial u} \\
&(3) \quad \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial v} \\
&(2) \quad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial y} \\
&(1) \quad \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial x}
\end{aligned}
$$

Loss: $\mathcal{L}(\, u(y(x)),\, v(y(x))\,)$



Step through it with numbers:

example “Fan-out > 1” on

Course website → Backprop 1D

Recap: Backpropagation with Scalars

Forward Pass:

$$
\begin{aligned}
&(1) \quad y = y(x) \\
&(2) \quad u = u(y) \\
&(2) \quad v = v(y) \\
&(3) \quad \mathcal{L} = \mathcal{L}(u, v)
\end{aligned}
$$

Backward Pass:

$$
\begin{aligned}
&(3) \quad \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial u} \\
&(3) \quad \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial v} \\
&(2) \quad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial y} \\
&(1) \quad \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial x}
\end{aligned}
$$

Implementation: Each variable/node is an object and has attributes `x.value` and `x.grad`. Values are computed **forward** and gradients **backward:**

$$
\begin{aligned}
&\texttt{x.value} = \texttt{Input} \\
&\texttt{y.value} = y(\texttt{x.value}) \\
&\texttt{u.value} = u(\texttt{y.value}) \\
&\texttt{v.value} = v(\texttt{y.value}) \\
&\texttt{L.value} = \mathcal{L}(\texttt{u.value}, \texttt{v.value})
\end{aligned}
$$

Recap: Backpropagation with Scalars

Forward Pass:

$$
\begin{aligned}
&(1) \quad y = y(x) \\
&(2) \quad u = u(y) \\
&(2) \quad v = v(y) \\
&(3) \quad \mathcal{L} = \mathcal{L}(u, v)
\end{aligned}
$$

Backward Pass:

$$
\begin{aligned}
&(3) \quad \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial u} \\
&(3) \quad \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial v} \\
&(2) \quad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial y} \\
&(1) \quad \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial x}
\end{aligned}
$$

Implementation: Each variable/node is an object and has attributes `x.value` and `x.grad`. Values are computed **forward** and gradients **backward:**

$$\texttt{x.grad} = \texttt{y.grad} = \texttt{u.grad} = \texttt{v.grad} = 0$$

$$\texttt{L.grad} = 1$$

$$\texttt{u.grad} \mathrel{+}= \texttt{L.grad} * (\partial \mathcal{L}/\partial u)(\texttt{u.value}, \texttt{v.value})$$

$$\texttt{v.grad} \mathrel{+}= \texttt{L.grad} * (\partial \mathcal{L}/\partial v)(\texttt{u.value}, \texttt{v.value})$$

$$\texttt{y.grad} \mathrel{+}= \texttt{u.grad} * (\partial u/\partial y)(\texttt{y.value})$$

$$\texttt{y.grad} \mathrel{+}= \texttt{v.grad} * (\partial v/\partial y)(\texttt{y.value})$$

$$\texttt{x.grad} \mathrel{+}= \texttt{y.grad} * (\partial y/\partial x)(\texttt{x.value})$$

Scalar vs. Matrix Operations

So far we have considered computations on **scalars:**

$$y = \sigma(w_1 x + w_0)$$

We now consider computations on **vectors** and **matrices:**

$$\mathbf{y} = \sigma(\mathbf{A}\mathbf{x} + \mathbf{b}), \qquad \mathbf{x} \in \mathbb{R}^D,\ \mathbf{A} \in \mathbb{R}^{M \times D},\ \mathbf{b}, \mathbf{y} \in \mathbb{R}^M$$

- σ is applied **elementwise:** $y_i = \sigma(u_i + b_i)$ with $\mathbf{u} = \mathbf{A}\mathbf{x}$
- Matrix $\mathbf{A}$ and vector $\mathbf{b}$ are objects with attributes `value` and `grad`
- `A.grad` stores $\nabla_{\mathbf{A}} \mathcal{L}$ and has the same shape as `A.value` (since $\mathcal{L}$ is scalar)

Backpropagation on Loops

The matrix/vector computation

$$\mathbf{y} = \sigma(\underbrace{\mathbf{A}\mathbf{x}}_{=\mathbf{u}} + \mathbf{b}), \qquad \mathbf{x} \in \mathbb{R}^D,\ \mathbf{A} \in \mathbb{R}^{M \times D},\ \mathbf{b}, \mathbf{u}, \mathbf{y} \in \mathbb{R}^M$$

can be written as **loops over scalar operations:**

for i $\quad \texttt{u.value}[i] = 0$

for i,j $\quad \texttt{u.value}[i]\ \texttt{+=}\ \texttt{A.value}[i, j] * \texttt{x.value}[j]$

for i $\quad \texttt{y.value}[i] = \sigma(\texttt{u.value}[i] + \texttt{b.value}[i])$

Backpropagation on Loops

The backpropagated gradients, in **reverse order** of the forward loops:

Backward of the last loop, $\texttt{y.value}[i] = \sigma(\texttt{u.value}[i] + \texttt{b.value}[i])$:

$$\text{for i} \quad \texttt{u.grad}[i] \mathrel{+}= \texttt{y.grad}[i] * \sigma'(\texttt{u.value}[i] + \texttt{b.value}[i])$$

$$\text{for i} \quad \texttt{b.grad}[i] \mathrel{+}= \texttt{y.grad}[i] * \sigma'(\texttt{u.value}[i] + \texttt{b.value}[i])$$

Backward of the matrix product, $\texttt{u.value}[i] \mathrel{+}= \texttt{A.value}[i,j] * \texttt{x.value}[j]$:

$$\text{for i,j} \quad \texttt{A.grad}[i,j] \mathrel{+}= \texttt{u.grad}[i] * \texttt{x.value}[j]$$

$$\text{for i,j} \quad \texttt{x.grad}[j] \mathrel{+}= \texttt{u.grad}[i] * \texttt{A.value}[i,j]$$

- **Red:** back-propagated gradients
- **Blue:** local gradients

Backpropagation on Loops

In practice, all deep learning operations can be written using loops over scalar assignments. Example for a **higher-order tensor:**

$$\text{for h,i,j,k} \quad \texttt{U.value}[h,i,j] \texttt{ += } \texttt{A.value}[h,i,k] * \texttt{B.value}[h,j,k]$$
$$\text{for h,i,j} \quad \texttt{Y.value}[h,i,j] = \sigma(\texttt{U.value}[h,i,j])$$

(for each h a matrix product $\mathbf{U}_h = \mathbf{A}_h \mathbf{B}_h^\top$: a *batched* matrix product)

Backpropagation loops:

$$\text{for h,i,j} \quad \texttt{U.grad}[h,i,j] \texttt{ += } \texttt{Y.grad}[h,i,j] * \sigma'(\texttt{U.value}[h,i,j])$$
$$\text{for h,i,j,k} \quad \texttt{A.grad}[h,i,k] \texttt{ += } \texttt{U.grad}[h,i,j] * \texttt{B.value}[h,j,k]$$
$$\text{for h,i,j,k} \quad \texttt{B.grad}[h,j,k] \texttt{ += } \texttt{U.grad}[h,i,j] * \texttt{A.value}[h,i,k]$$

Backpropagation on Loops: Matrix Form

$$\mathbf{y} = \sigma(\mathbf{A}\mathbf{x} + \mathbf{b}), \qquad \mathbf{x} \in \mathbb{R}^D,\ \mathbf{A} \in \mathbb{R}^{M \times D},\ \mathbf{b}, \mathbf{u}, \mathbf{y} \in \mathbb{R}^M$$

Each backward loop is **one matrix operation** on the whole array:

for i $\quad \texttt{u.grad}[i]\ \texttt{+=}\ \texttt{y.grad}[i] * \sigma'(\texttt{u.value}[i] + \texttt{b.value}[i])$

$\Rightarrow \quad \texttt{u.grad}\ \texttt{+=}\ \texttt{y.grad} \odot \sigma'(\texttt{u.value} + \texttt{b.value})$ elementwise; same for `b.grad`

for i,j $\quad \texttt{A.grad}[i, j]\ \texttt{+=}\ \texttt{u.grad}[i] * \texttt{x.value}[j]$

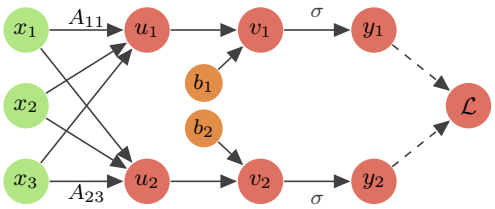
$\Rightarrow \quad \texttt{A.grad}\ \texttt{+=}\ \texttt{u.grad}\ \texttt{x.value}^\top$ outer product, $M \times D$

for i,j $\quad \texttt{x.grad}[j]\ \texttt{+=}\ \texttt{u.grad}[i] * \texttt{A.value}[i, j]$

$\Rightarrow \quad \texttt{x.grad}\ \texttt{+=}\ \texttt{A.value}^\top\ \texttt{u.grad}$ matrix $\times$ vector, sums over i

Scalar vs. Matrix Computation Graph

Graph of scalars: one node per number

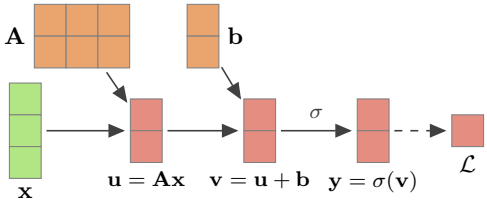


$$\frac{\partial \mathcal{L}}{\partial u_i} = \frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial v_i} = \sigma'(v_i) \frac{\partial \mathcal{L}}{\partial y_i}$$

$$\frac{\partial \mathcal{L}}{\partial A_{ij}} = \frac{\partial \mathcal{L}}{\partial u_i} x_j$$

$$\frac{\partial \mathcal{L}}{\partial x_j} = \sum_i A_{ij} \frac{\partial \mathcal{L}}{\partial u_i}$$

Graph of matrices: one node per array



$$\nabla_{\mathbf{u}} \mathcal{L} = \nabla_{\mathbf{b}} \mathcal{L} = \nabla_{\mathbf{v}} \mathcal{L} = \sigma'(\mathbf{v}) \odot \nabla_{\mathbf{y}} \mathcal{L}$$

$$\nabla_{\mathbf{A}} \mathcal{L} = \nabla_{\mathbf{u}} \mathcal{L}\, \mathbf{x}^\top$$ outer product

$$\nabla_{\mathbf{x}} \mathcal{L} = \mathbf{A}^\top \nabla_{\mathbf{u}} \mathcal{L}$$ matrix × vector

Minibatching

Source code has two components:

- Slow part: Sequential operations (Python)
- Fast part: Vector/matrix operations (NumPy, BLAS, CUDA)

Goal:

- Fast part should dominate computation (wall clock time)
- Reduce the number of slow sequential operations (e.g., Python loops) by running the fast vector/matrix operations on several data points jointly
- This is called **minibatching** and used in stochastic gradient descent

Minibatching

Affine + Sigmoid: (applied to N data points simultaneously)

$$\mathbf{Y} = \sigma(\underbrace{\mathbf{X}\mathbf{A}}_{=\mathbf{U}} + \mathbf{B})$$

- Each row in $\mathbf{X} \in \mathbb{R}^{N \times D}$ is a data point, bias $\mathbf{b} \in \mathbb{R}^{M}$ is broadcast to $\mathbf{B} \in \mathbb{R}^{N \times M}$

Loops now include **batch index b:**

for b,i $\quad \texttt{U.value}[b, i] = 0$

for b,i,j $\quad \texttt{U.value}[b, i] \texttt{ += X.value}[b, j] * \texttt{A.value}[j, i]$

for b,i $\quad \texttt{Y.value}[b, i] = \sigma(\texttt{U.value}[b, i] + \texttt{B.value}[i])$

- Only inputs and outputs depend on batch index b, not the parameters (e.g., `A`, `B`)
- By convention, the gradients are averaged over the batch

Implementation

Affine Transformation: (applied to N data points simultaneously)

$$\mathbf{Y} = \mathbf{X}\mathbf{A} + \mathbf{B}$$

- Each row in $\mathbf{X} \in \mathbb{R}^{N \times D}$ is a data point, bias $\mathbf{b} \in \mathbb{R}^{M}$ is broadcast to $\mathbf{B} \in \mathbb{R}^{N \times M}$

Implementation in EDF:

```
def forward(self):
    self.value = np.matmul(self.x.value,self.w.A.value) + self.w.b.value

def backward(self):
    self.x.addgrad(np.matmul(self.grad,self.w.A.value.transpose()))
    self.w.b.addgrad(self.grad)
    self.w.A.addgrad(self.x.value[:,:,np.newaxis] * self.grad[:,np.newaxis,:])
```

- Computation graphs are easy to understand using the loop notation
- Efficient implementation using NumPy primitives not always obvious

3.2
Multi-Layer Perceptrons

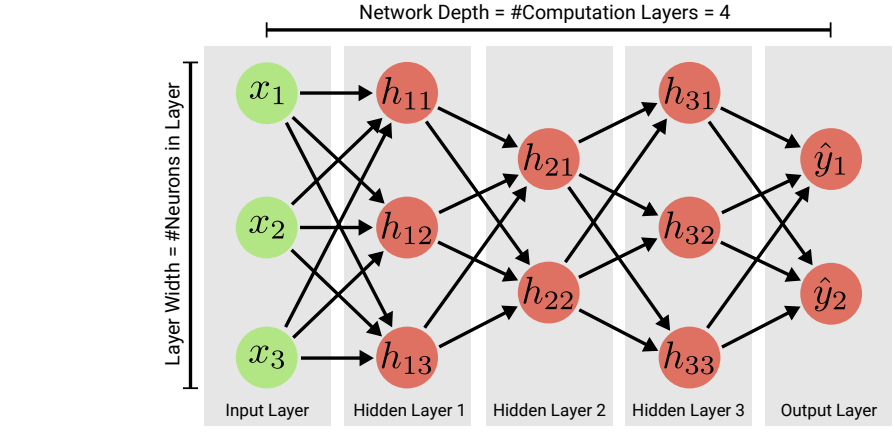
Multi-Layer Perceptrons

- MLPs are **feedforward** neural networks (no feedback connections)
- They **compose** several non-linear functions $\mathbf{f}(\mathbf{x}) = \hat{\mathbf{y}}(\mathbf{h}_3(\mathbf{h}_2(\mathbf{h}_1(\mathbf{x}))))$
  where $\mathbf{h}_i(\cdot)$ are called **hidden layers** and $\hat{\mathbf{y}}(\cdot)$ is the **output layer**
- The data specifies only the behavior of the output layer (thus the name “hidden”)
- Each layer i comprises multiple **neurons** j which are implemented as **affine transformations** $(\mathbf{a}^\top \mathbf{x} + \mathbf{b})$ followed by non-linear **activation functions** (g):

$$h_{ij} = g(\mathbf{a}_{ij}^\top \mathbf{h}_{i-1} + \mathbf{b}_{ij})$$

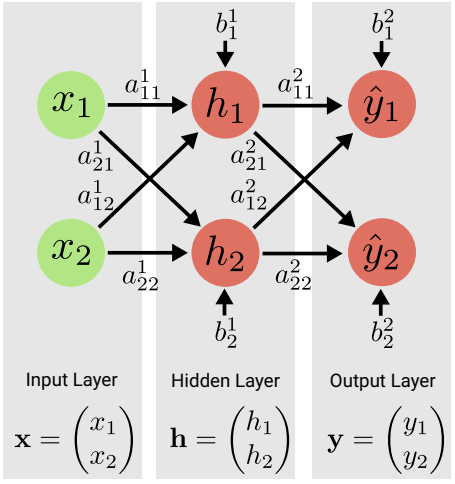
- Each neuron in each layer is **fully connected** to all neurons of the previous layer
- The overall length of the chain is the **depth** of the model $\Rightarrow$ “Deep Learning”
- MLPs appear in modern networks as **Linear** or **Dense** layers (e.g., in transformers)

MLP Network Architecture



- Neurons are grouped into layers, each neuron **fully connected** to all prev. ones
- **Hidden layer** $\mathbf{h}_i = g(\mathbf{A}_i\mathbf{h}_{i-1} + \mathbf{b}_i)$ with **activation function** $g(\cdot)$ and weights $\mathbf{A}_i, \mathbf{b}_i$
- **Output layer** $\mathbf{y} = h(\mathbf{A}_L\mathbf{h}_{L-1} + \mathbf{b}_L)$ with **activ. function** $h(\cdot)$ and weights $\mathbf{A}_L, \mathbf{b}_L$

MLP Network Architecture: Example



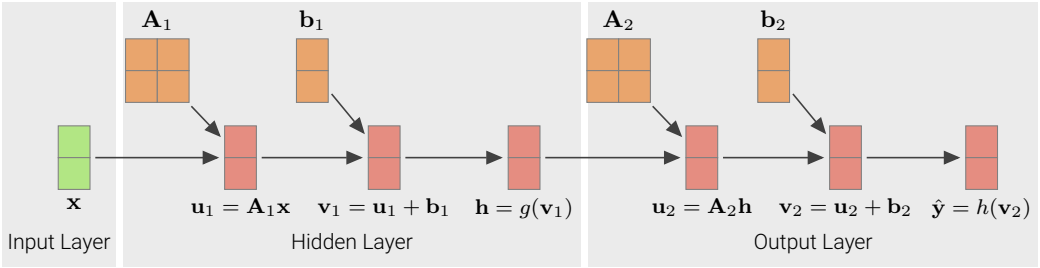
$$\mathbf{h} = g(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{y} = h(\mathbf{A}_2\mathbf{h} + \mathbf{b}_2)$$

$$\mathbf{A}_l = \begin{pmatrix} a_{11}^l & a_{12}^l \\ a_{21}^l & a_{22}^l \end{pmatrix}$$

$$\mathbf{b}_l = \begin{pmatrix} b_1^l \\ b_2^l \end{pmatrix}$$

- Example MLP with $L = 2$ layers of width 2 and parameters $\{\mathbf{A}_1, \mathbf{A}_2, \mathbf{b}_1, \mathbf{b}_2\}$

MLP Network Architecture: Example in Matrix Form

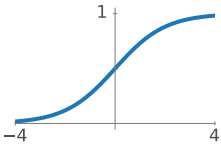


- The MLP from the previous slide as a graph with **one node per array**
- Each layer has three steps: **matrix product**, **bias**, **elementwise activation**
- All layers have width 2: $\mathbf{A}_1, \mathbf{A}_2 \in \mathbb{R}^{2\times 2}$ and $\mathbf{b}_1, \mathbf{b}_2, \mathbf{h}, \hat{\mathbf{y}} \in \mathbb{R}^2$

Activation Functions g(·)

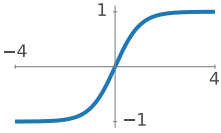
Sigmoid (~1986)

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



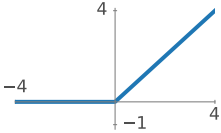
tanh (~1989)

$$\tanh(x)$$



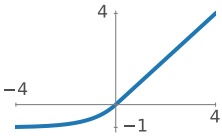
ReLU (~2010)

$$\max(0, x)$$



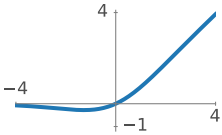
ELU (~2015)

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



SiLU (~2017)

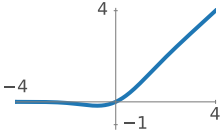
$$x\,\sigma(x)$$



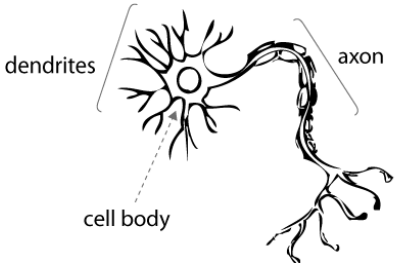
GELU (~2016)

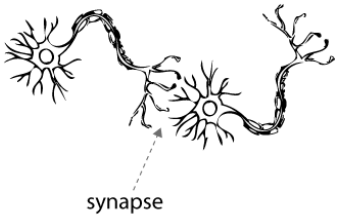
$$x\,\Phi(x)$$

Φ: Gaussian CDF



Neural Motivation





- Neurons in the brain are structured in layers
- They receive input from many other units and compute their own activation
- How brain-like modern networks are is an **open research question**
- Our goal is not to model the brain, but to build great models

Training

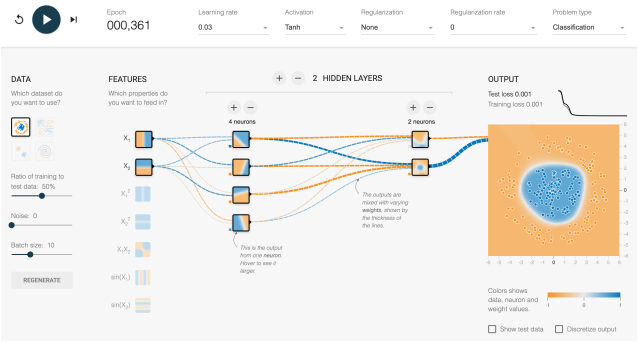
Algorithm for training an MLP using (stochastic) gradient descent:

1. Initialize weights $\mathbf{w}$, pick learning rate η and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw (random) minibatch $\mathcal{X}_{\text{batch}} \subseteq \mathcal{X}$
3. For all elements $(\mathbf{x}, \mathbf{y}) \in \mathcal{X}_{\text{batch}}$ of minibatch (in parallel) do:
   3.1 Forward propagate $\mathbf{x}$ through network to calculate $\mathbf{h}_1, \mathbf{h}_2, \ldots, \hat{\mathbf{y}}$
   3.2 Backpropagate gradients through network to obtain $\nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{|\mathcal{X}_{\text{batch}}|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{X}_{\text{batch}}} \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
5. If validation error decreases, go to step 2, otherwise stop

Remarks:

- Large datasets typically do not fit into GPU memory $\Rightarrow |\mathcal{X}_{\text{batch}}| < |\mathcal{X}|$
- Our examples on the next slides are small $\Rightarrow |\mathcal{X}_{\text{batch}}| = |\mathcal{X}|$

Interactive: TensorFlow Playground



Note: Keep the inputs X_1, X_2 and switch off the hand-crafted features $(X_1^2, \sin(X_1), \ldots)$.

Try it yourself!

Train an MLP in the browser:

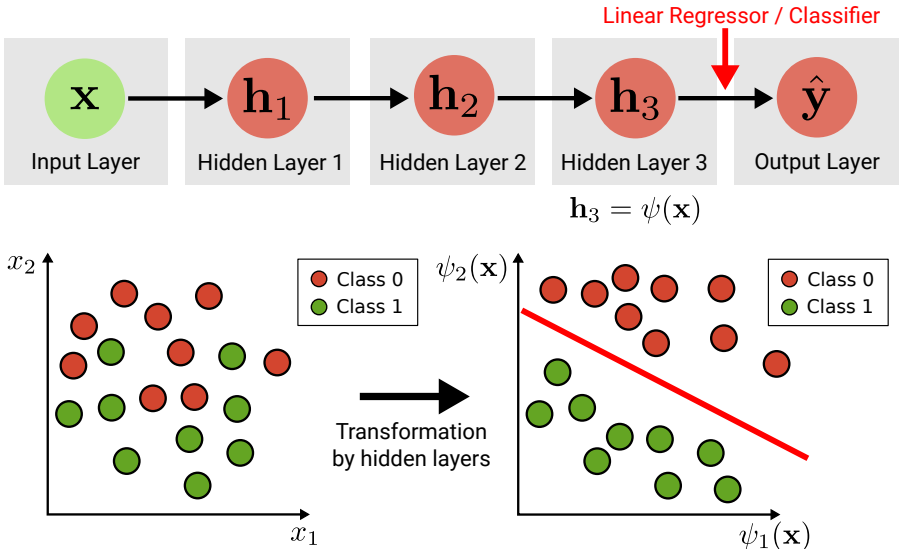
- Four 2D datasets
- Layers and neurons
- Activation, learning rate

playground.tensorflow.org

Stretching Break

5 Minutes

Feature Learning Perspective



Expressiveness

This following two-layer MLP

$$\mathbf{h} = g(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{y} = g(\mathbf{A}_2\mathbf{h} + \mathbf{b}_2)$$

can be written as

$$\mathbf{y} = g(\mathbf{A}_2\, g(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$$

What if we would be using a linear activation function $g(\mathbf{x}) = \mathbf{x}$?

$$\mathbf{y} = \mathbf{A}_2\left(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1\right) + \mathbf{b}_2 = \mathbf{A}_2\mathbf{A}_1\mathbf{x} + \mathbf{A}_2\mathbf{b}_1 + \mathbf{b}_2 = \mathbf{A}\mathbf{x} + \mathbf{b}$$

- With linear activations, a multi-layer network can only express linear functions
- What is the model capacity of MLPs with non-linear activation functions?

Universal Approximation Theorem

Theorem 1

Let σ be any continuous discriminatory function. Then finite sums of the form

$$G(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{a}_j^\top \mathbf{x} + b_j)$$

are dense in the space of continuous functions $C(I_n)$ on the n-dimensional unit cube I_n. In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $G(\mathbf{x})$ for which

$$|G(\mathbf{x}) - f(\mathbf{x})| < \epsilon \quad \text{for all} \quad \mathbf{x} \in I_n$$

Remark: Has been proven for various activation functions (e.g., Sigmoid, ReLU).

Cybenko: Approximation by superposition of a sigmoidal function. Mathematics of Control, Signals, and Systems, 1989.

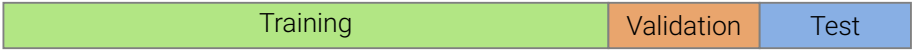
Generalization

- Universal approximation is a statement about **expressiveness**:
  A network that approximates the function f **exists**
- It does not tell us how large N must be, whether training finds this network,
  or how the network behaves on **unseen inputs**
- Lecture 1: A model that is flexible enough to fit any training set tends to **overfit**
- Modern networks have far more parameters than training points,
  and can even fit **random labels** perfectly
- Yet, when trained on real data, they **generalize** remarkably well
- **Why** neural networks generalize so well is still **not understood** today

Zhang, Bengio, Hardt, Recht and Vinyals: Understanding deep learning requires rethinking generalization. ICLR, 2017.

Train / Validation / Test Split

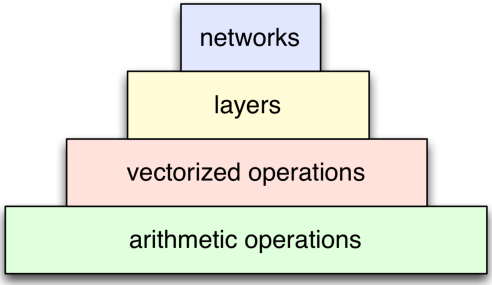
In machine learning, we always **split our dataset** into three disjoint parts:



- **Training set:** fit the parameters $\mathbf{w}$ with gradient descent
- **Validation set:** choose hyperparameters (architecture, learning rate, …) and decide when to stop training
- **Test set:** estimate the error on unseen data, used **once** at the very end
- Never tune on the test set: it then no longer measures generalization

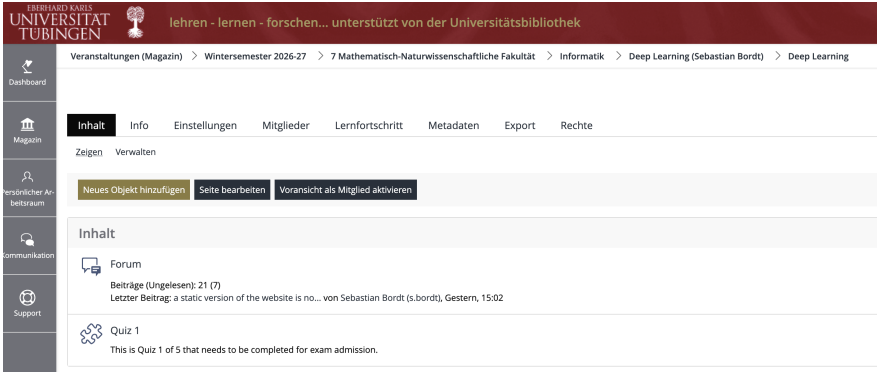
New to this? A visual explainer:
MLU-Explain → Train, Test, and Validation Sets

Levels of Abstraction



- When designing neural networks and machine learning algorithms, you'll need to simultaneously think at multiple level's of abstraction
- "The psychological profiling [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large." [Donald E. Knuth]

Exam Quiz 1 is Now Live on Ilias



- The first of 5 exam quizzes, in the Ilias course *Deep Learning*
- You have **one week** to participate
- You need to participate in at least 3 of the 5 exam quizzes for exam admission

Break

- **Now: 30 minute break**
  - Complete **Exam Quiz 1** on **Ilias**
  - Vote for the exercises that you want discussed in the tutorial
- **After that: Tutorial**
  - Discussion of exercise sheet 1



Vote for exercises!

Exercise discussion survey