Neural Networks II

Machine Learning 1 — Lecture 9

16th May 2023

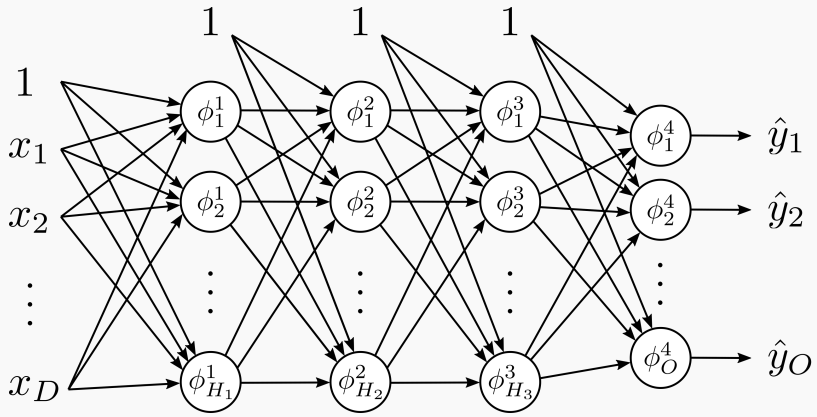
Robert Peharz
Institute of Theoretical Computer Science
Graz University of Technology

Neuron, Unit

The **neuron** (**unit**) is the basic building block of neural networks:

$$\hat{y} = \phi(a) = \phi\left(\sum_{i=0}^{D} w_i x_i\right)$$

- D: number of inputs
- x_i: inputs (features, outputs from previous neurons)
- w_i: weights
- w_0: bias term ($x_0 \equiv 1$)
- ϕ: non-linear **activation function** (also called **non-linearity**)
- a: **activation** $\sum_{i=0}^{D} w_i x_i$
- $\hat{y}$: output $\phi(a)$
- **Affine function of x, followed by non-linearity ϕ**



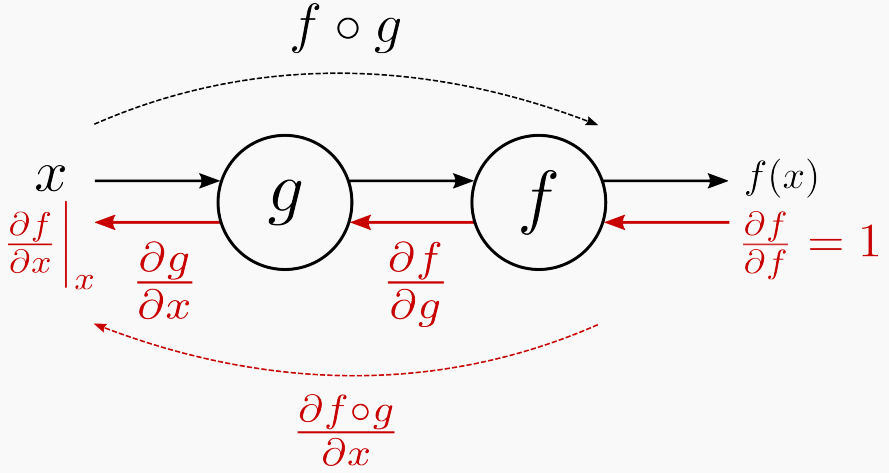
- **Shallow networks** (1-layer)
  - **universal approximators** for many non-linearities
  - However, some function classes require **exponentially** many neurons, in the number of inputs
- **Deep networks**
  - can often represent represent the same functions with only **polynomial size**
  - This phenomenon is called **expressive efficiency** – Deep networks can represent functions more efficiently than shallow ones

Automatic Differentiation (Backprop)

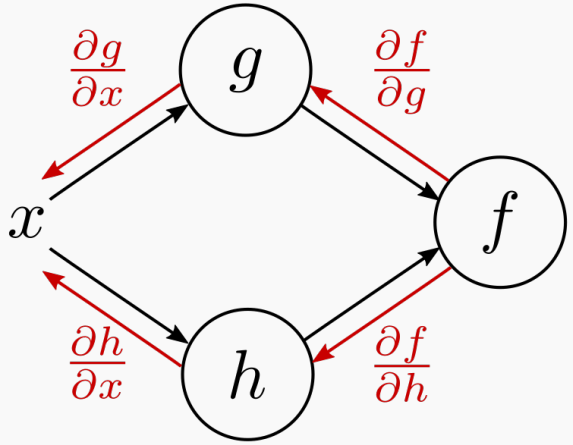
Automatic Differentiation (AD): systematic use of the chain rule

---

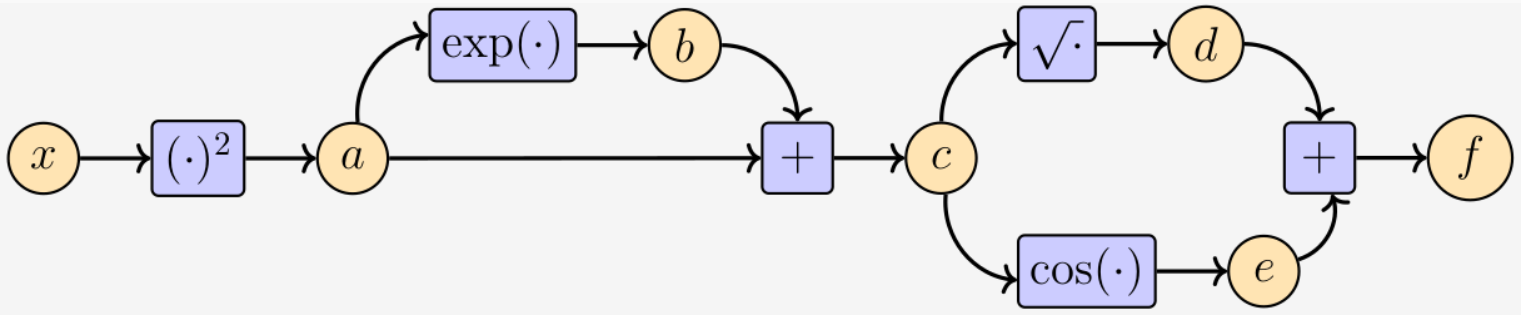
$$\frac{\partial f \circ g}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$



$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x}$$

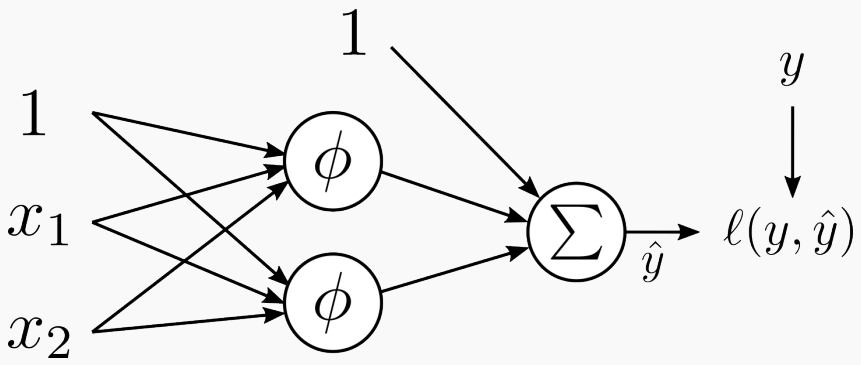


Backwards Pass (Autodiff, Backprop)

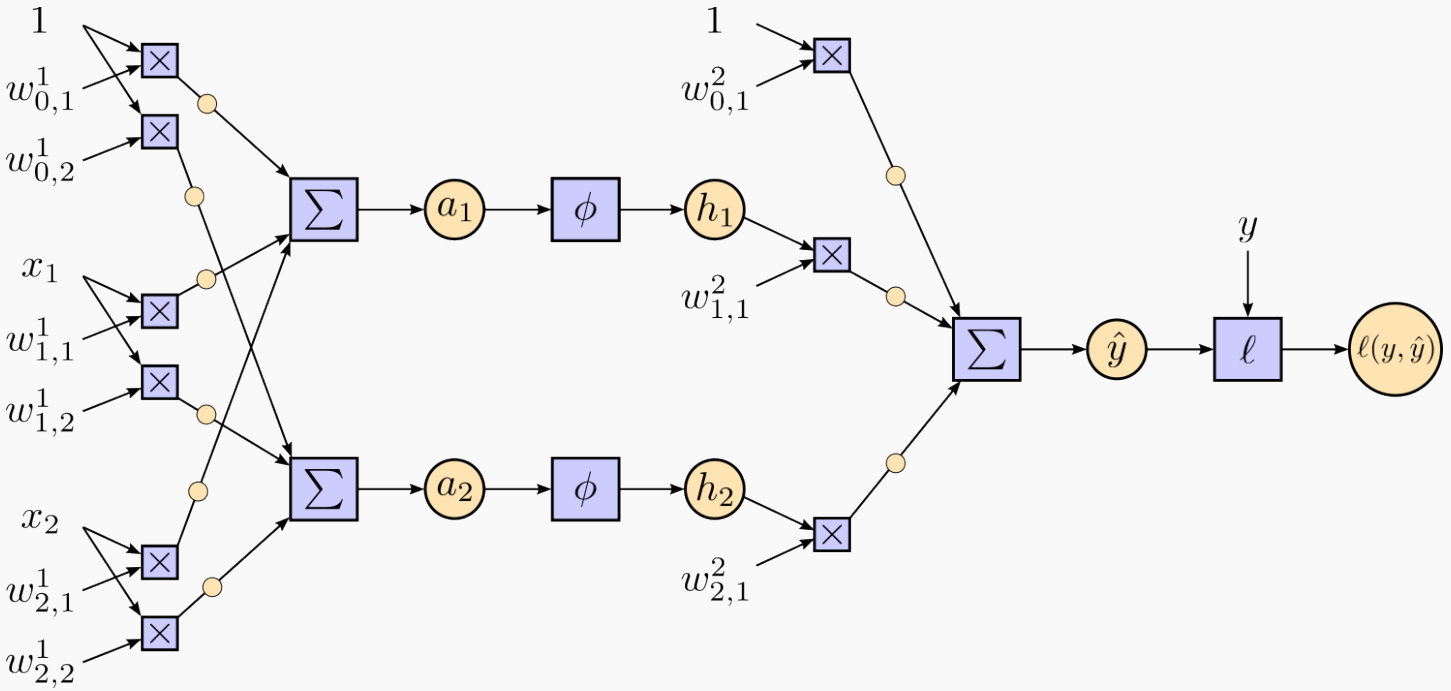


- $\frac{\partial f}{\partial f} = 1$
- $\frac{\partial f}{\partial e} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial e} = 1 \times 1 = 1$
- $\frac{\partial f}{\partial d} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial d} = 1 \times 1 = 1$
- $\frac{\partial f}{\partial c} = \frac{\partial f}{\partial e}\frac{\partial e}{\partial c} + \frac{\partial f}{\partial d}\frac{\partial d}{\partial c} = 1 \times -\sin(c) + 1 \times \frac{1}{2}c^{-\frac{1}{2}} = -0.822$
- $\frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} = -0.822 \times 1 = -0.822$
- $\frac{\partial f}{\partial a} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} = -0.822 \times 1 + -0.822 \times \exp(a) = -45.717$
- $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} = -45.717 \times 2x = -182.87$
- Note we get derivatives for **all nodes**, not only x!

- Let a computational graph be given
- Compute a **forward pass** with given inputs, delivering the output and all intermediate results, which are stored in their respective nodes
- **Reverse-mode autodiff (backprop)** starts with the trivial derivative 1 at the output node o
- **Backward-pass** computes for any **intermediate or input node** n the partial derivative $\frac{\partial o}{\partial n}$
- Recursive principle: the partial derivatives $\frac{\partial s_i}{\partial n}$ for all **successors** s_i of n have already been computed (stored in s_i)
- Hence, the partial derivative is given as $\frac{\partial o}{\partial n} = \sum_i \frac{\partial o}{\partial s_i} \frac{\partial s_i}{\partial n}$
- $\frac{\partial o}{\partial n}$ is stored in n; continue with backward-pass



- One hidden layer with two neurons, sigmoid activation function
- Loss function $\ell(y, \hat{y})$, with squared loss (regression problem)
- Weights $w^l_{i,j}$ sitting on input edges to each neuron
- Parameter vector $\boldsymbol{\theta} = (w^1_{0,1}, w^1_{0,2}, \ldots, w^2_{2,1})^T$
- Goal: compute $\nabla_{\boldsymbol{\theta}} \ell = \left(\frac{\partial \ell}{\partial w^1_{0,1}}, \frac{\partial \ell}{\partial w^1_{0,2}}, \ldots, \frac{\partial \ell}{\partial w^2_{2,1}} \right)^T$ for given $\boldsymbol{x}, y$
- Gradient for training loss is just the sum over all training examples
- Resulting gradient is used for learning via gradient descent



- The MLP corresponds to the **computational graph** above
- Note that also weights $w^l_{i,j}$ and target y are inputs
- The loss ℓ is the output node of the graph
- We use standard autodiff do compute $\frac{\partial \ell}{\partial w^l_{i,j}}$, for all l, i, j

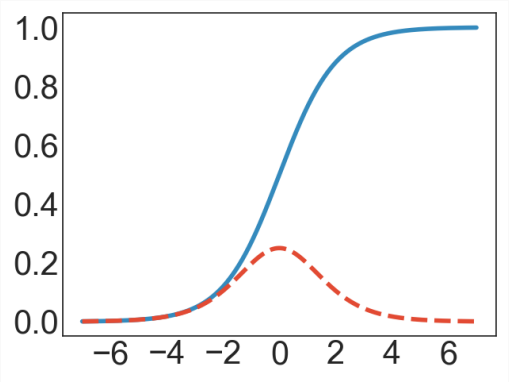
Recap:

- For **squared loss** $\ell(y, \hat{y}) = (\hat{y} - y)^2$ we have

$$\frac{\partial \ell}{\partial \hat{y}} = 2(\hat{y} - y)$$

- For **sigmoid** $\phi(z) = \frac{1}{1+\exp(-z)}$ we have

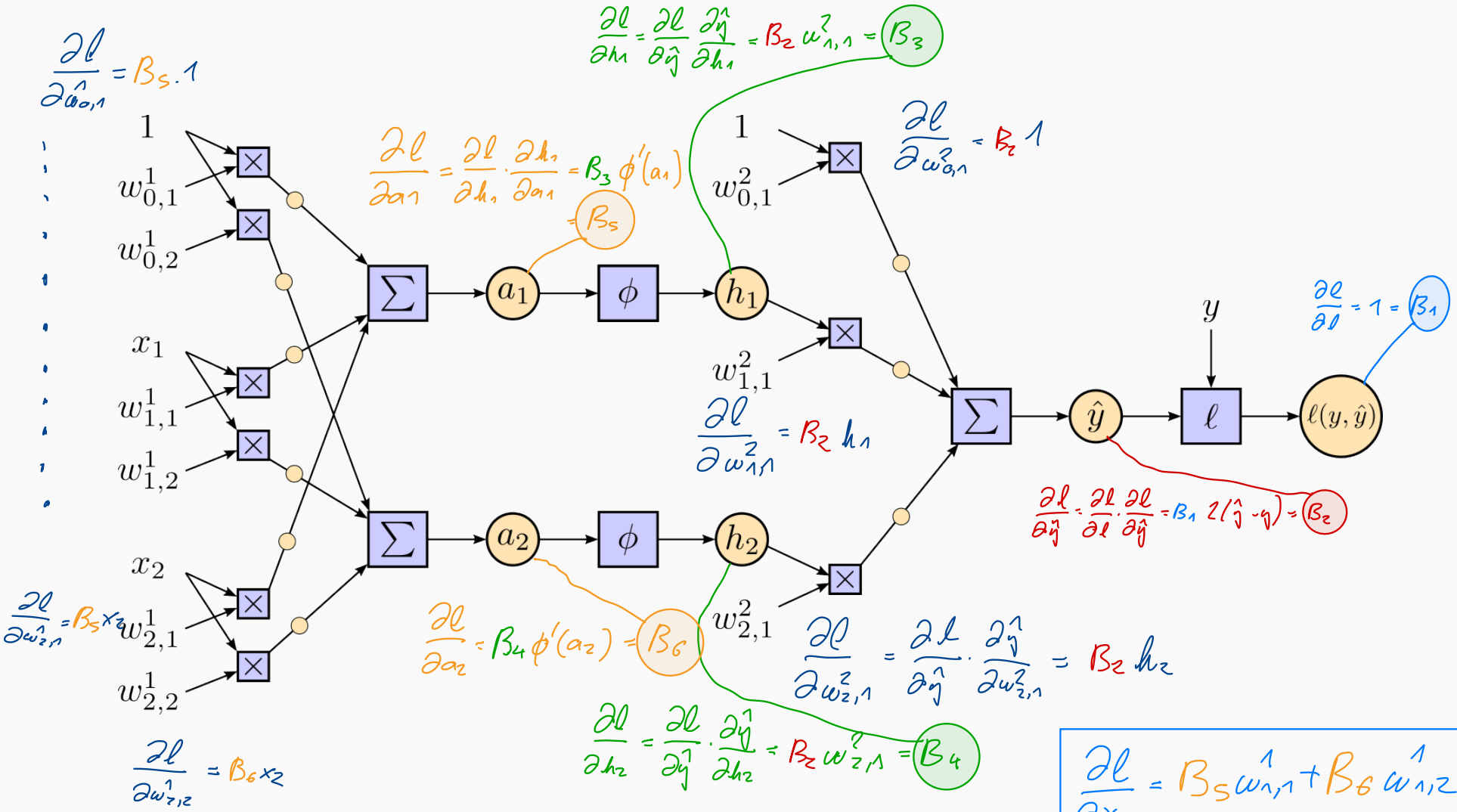
$$\frac{\partial \phi}{\partial z} = \sigma(z)\sigma(-z) = \sigma(z)(1 - \sigma(z))$$



- Generally, loss and activation functions used in NNs have "easy" analytic derivatives

Autodiff in Neural Nets

Example

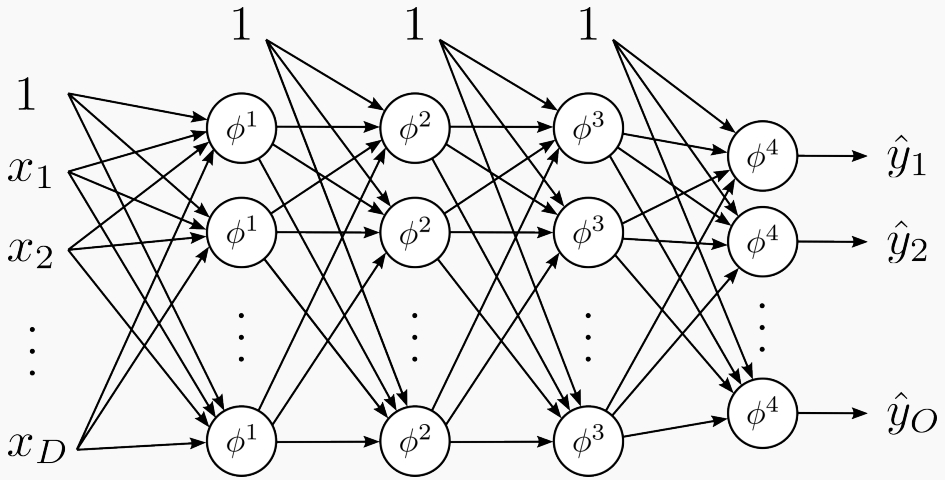


sigmoid: $\phi'(z) = \phi(z)\phi(-z) = \phi(z)(1 - \phi(z))$

$$\frac{\partial \ell}{\partial x_1} = B_5 w^1_{1,1} + B_6 w^1_{1,2}$$

$$\frac{\partial \ell}{\partial x_2} = B_5 w^1_{2,1} + B_6 w^1_{2,2}$$

Matrix-Vector Notation for MLPs



- $L+1$ layers, where the first L layers are the **hidden layers**
- The hidden layers have $H_1, H_2, \ldots, H_L$ units
- Assume we use the same **activation function** ϕ^l across the l^{th} layer (often linear activation function is used for the output layer)
- Let $h_j^0 = x_j$
- $a_j^l = \sum_{k=1}^{H_{l-1}} w_{k,j}^l \, h_k^{l-1}$ is the **activation** of the j^{th} unit in the l^{th} layer
- $h_j^l = \phi^l(a_j^l)$ is the output of the j^{th} unit in the l^{th} layer

Matrix-Vector Notation for MLPs

- Collect all outputs of the l^{th} layer in a vector (including a constant "dummy" feature for the bias):

$$\boldsymbol{h}^0 = (1, x_1, x_2, \ldots, x_D)^\top$$
$$\boldsymbol{h}^l = (1, h_1^l, h_2^l, \ldots, h_{H_l}^l)^\top$$

- Consider now the j^{th} unit in the l^{th} layer
- We collect all incoming weights in a vector $\boldsymbol{w}_j^l$:

$$\boldsymbol{w}_j^l = (w_{0,j}^l, w_{1,j}^l, \ldots, w_{H_{l-1},j}^l)^T$$

- Activation a_j^l is just an inner product between $\boldsymbol{w}_j^l$ and $\boldsymbol{h}^{l-1}$:

$$a_j^l = \sum_{i=1}^{H_{l-1}} w_{i,j}^l \, h_i^{l-1} = {\boldsymbol{w}_j^l}^T \boldsymbol{h}^{l-1}$$

Vector Notation for MLPs cont'd

- $a_j^l = {\boldsymbol{w}_j^l}^T \boldsymbol{h}^{l-1}$
- Next we collect all $\boldsymbol{w}_j^l$ in a $H_{l-1} \times H_l$ **weight matrix** W^l:

$$\mathrm{W}^l = \left(\boldsymbol{w}_1^l, \boldsymbol{w}_2^l, \ldots, \boldsymbol{w}_{H_l}^l\right)$$

- All activations $\boldsymbol{a}^l = \left(a_1^l, a_2^l, \ldots, a_{H_l}^l\right)^T$ in layer l are now obtained obtained with a simple matrix multiplication:

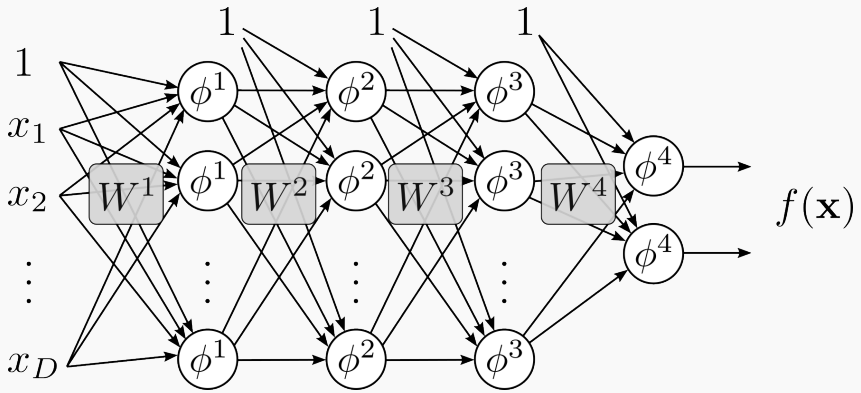
$$\boldsymbol{a}^l = {\mathrm{W}^l}^T \boldsymbol{h}^{l-1}$$

- The outputs of layer l are then given as

$$\boldsymbol{h}^l = \left(1, \phi^l(a_1^l), \phi^l(a_2^l), \ldots, \phi^l(a_{H_l}^l)\right)^T$$

  i.e., ϕ^l acts **element-wise** on $\boldsymbol{a}^l$

Writing a 4-layer MLP in One Line*



$$f(\boldsymbol{x}) = \phi^4\Big(\mathrm{W}^{4\,T}\,\bar{\phi}^3\Big(\mathrm{W}^{3\,T}\,\bar{\phi}^2\Big(\mathrm{W}^{2\,T}\,\bar{\phi}^1\Big(\overbrace{\mathrm{W}^{1\,T}\boldsymbol{x}}^{\boldsymbol{a}^1}\Big)\Big)\Big)\Big)$$

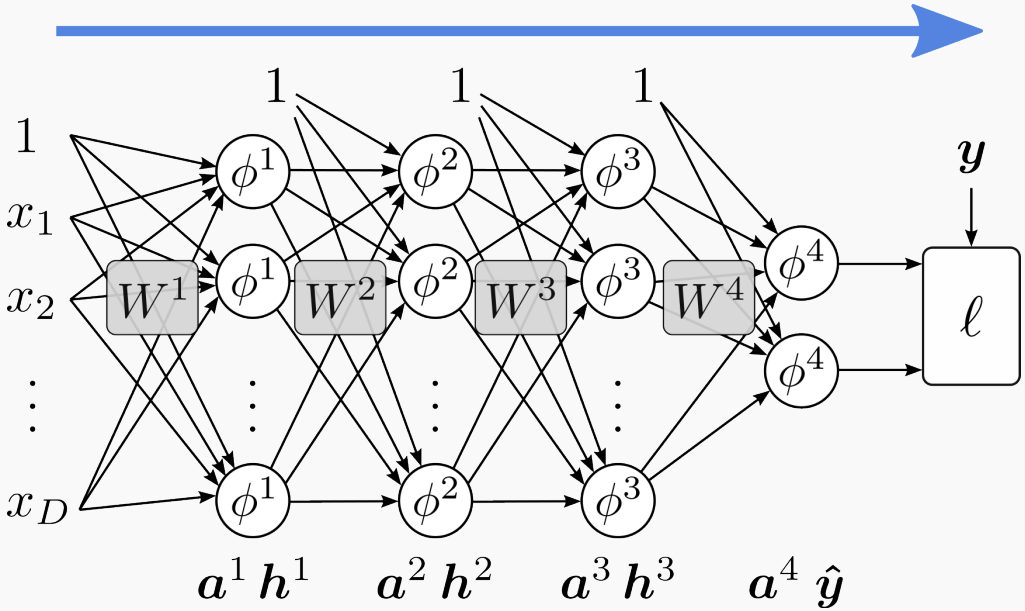
$\boldsymbol{a}^4$, $\boldsymbol{h}^3$, $\boldsymbol{a}^3$, $\boldsymbol{h}^2$, $\boldsymbol{a}^2$, $\boldsymbol{h}^1$, $\boldsymbol{a}^1$

$\bar{\phi}$ denotes adding a "dummy feature" 1.

Advantages of Vector Notation

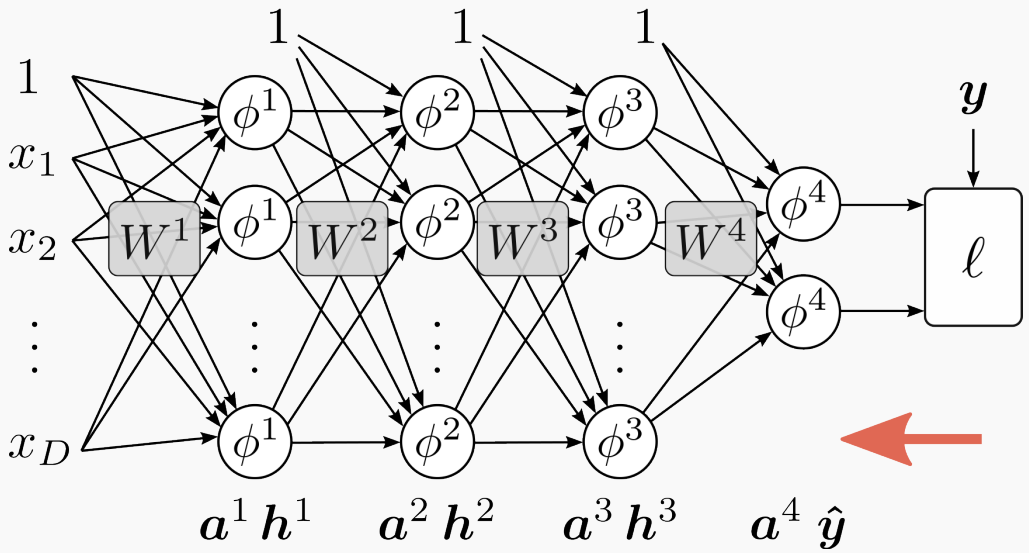
- More compact representation of neural networks
- Math becomes actually easier
- Vectorized implementations are typically faster, especially on dedicated hardware (GPUs, TPUs, etc.)

Autodiff in Matrix-Vector Form



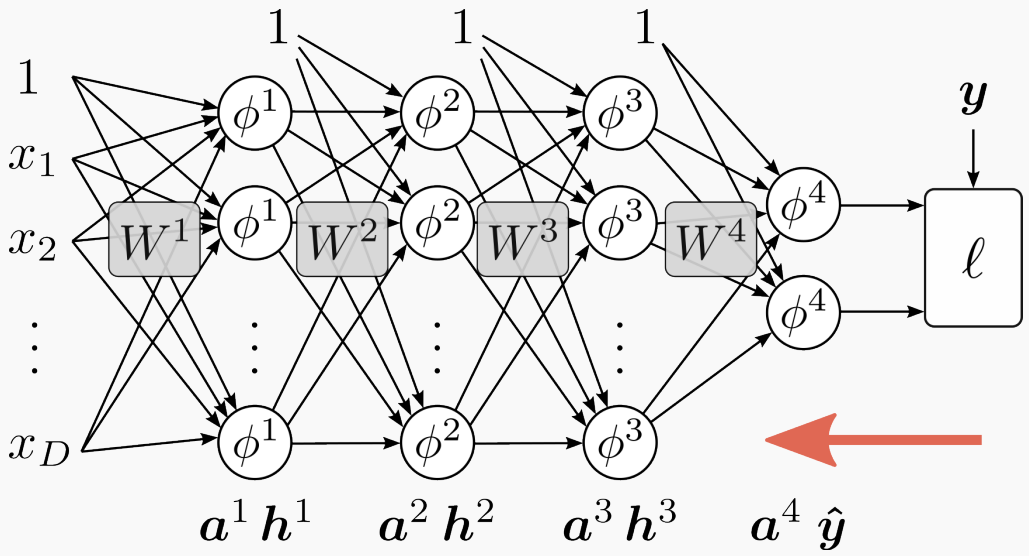
Forward evaluation with input $\boldsymbol{x}$, computing loss $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y})$

Autodiff in Matrix-Vector Form



The gradient of the loss w.r.t. the output layer $\nabla_{\hat{\boldsymbol{y}}^4}\ell = (\frac{\partial \ell}{\partial \hat{y}_1}, \ldots, \frac{\partial \ell}{\partial \hat{y}_O})^T$ is available in closed form for all losses used in practice (e.g., squared loss, cross-entropy)

Autodiff in Matrix-Vector Form

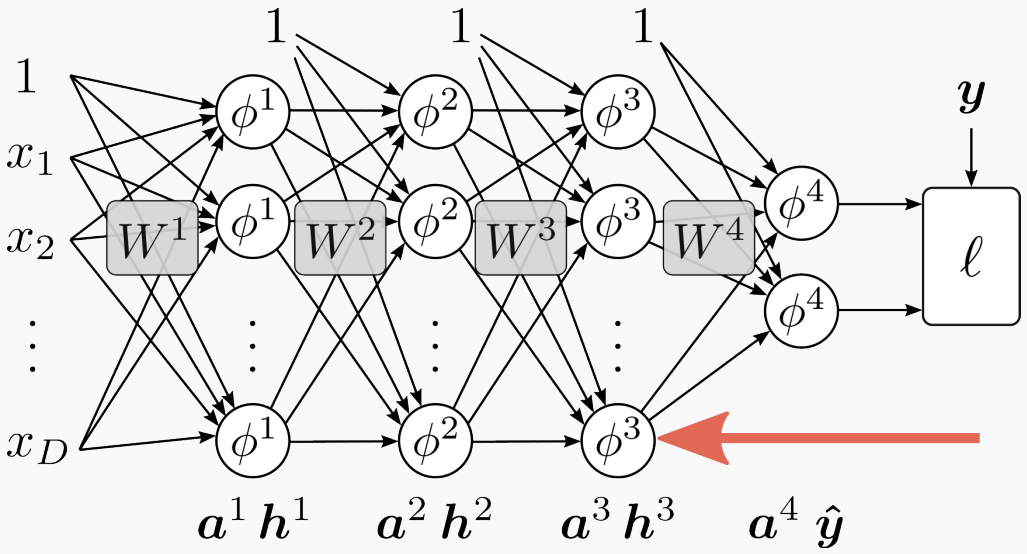


From autodiff we know $\frac{\partial \ell}{\partial a_i^4} = \frac{\partial \ell}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial a_i^4} = \frac{\partial \ell}{\partial \hat{y}_i} \phi^{4\prime}(a_i^4)$, where $\phi^{4\prime}$ is the derivative of ϕ^4. In short

$$\nabla_{\boldsymbol{a}^4} \ell = \mathit{diag}(\phi^{4\prime}(\boldsymbol{a}^4)) \nabla_{\hat{\boldsymbol{y}}^4} \ell$$

Here $\mathit{diag}(\boldsymbol{z})$ is the diagonal matrix whose diagonal is $\boldsymbol{z}$.

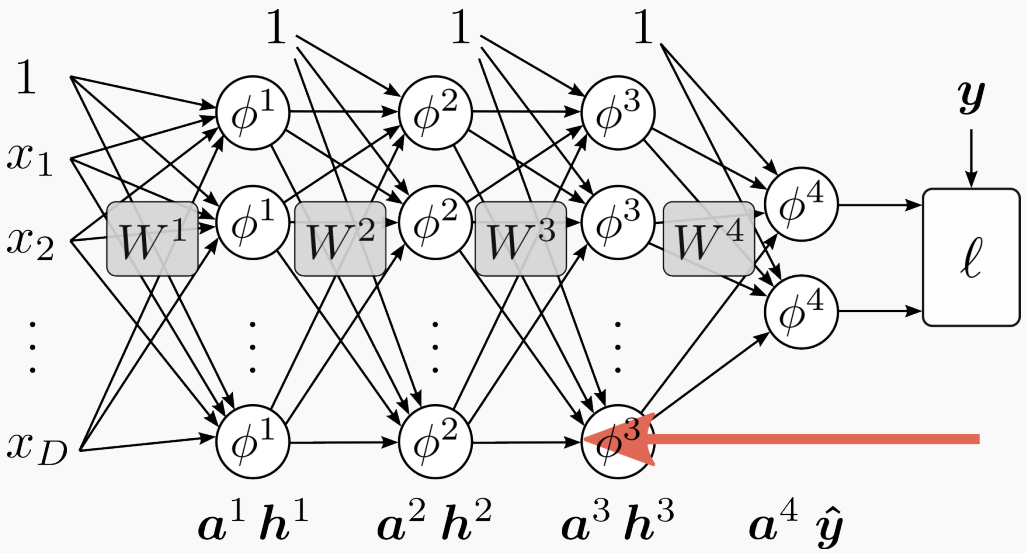
Autodiff in Matrix-Vector Form



From autodiff we know $\frac{\partial \ell}{\partial h_i^3} = \sum_{j=1}^{O} \frac{\partial \ell}{\partial a_j^4} \frac{\partial a_j^4}{\partial h_i^3} = \sum_{j=1}^{O} \frac{\partial \ell}{\partial a_j^4} w_{i,j}^4$ which is an inner product between $\nabla_{\boldsymbol{a}^4} \ell$ and the i^{th} row of W^4. In short

$$\nabla_{\boldsymbol{h}^3} \ell = W^4 \nabla_{\boldsymbol{a}^4} \ell$$

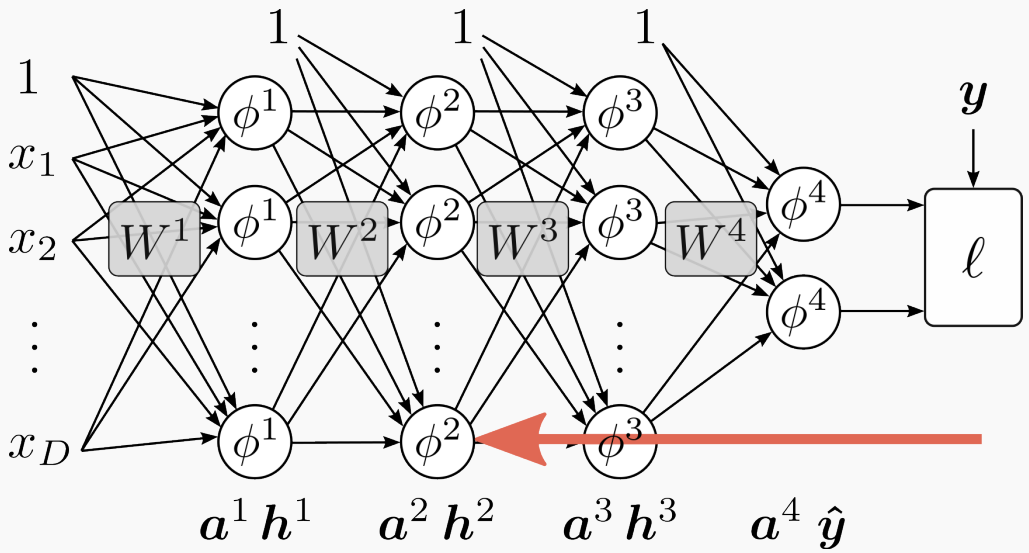
Autodiff in Matrix-Vector Form



and so on . . .

$$\nabla_{\boldsymbol{a}^3}\ell = \mathit{diag}(\phi^{3\prime}(\boldsymbol{a}^3))\,\nabla_{\boldsymbol{h}^3}\ell$$

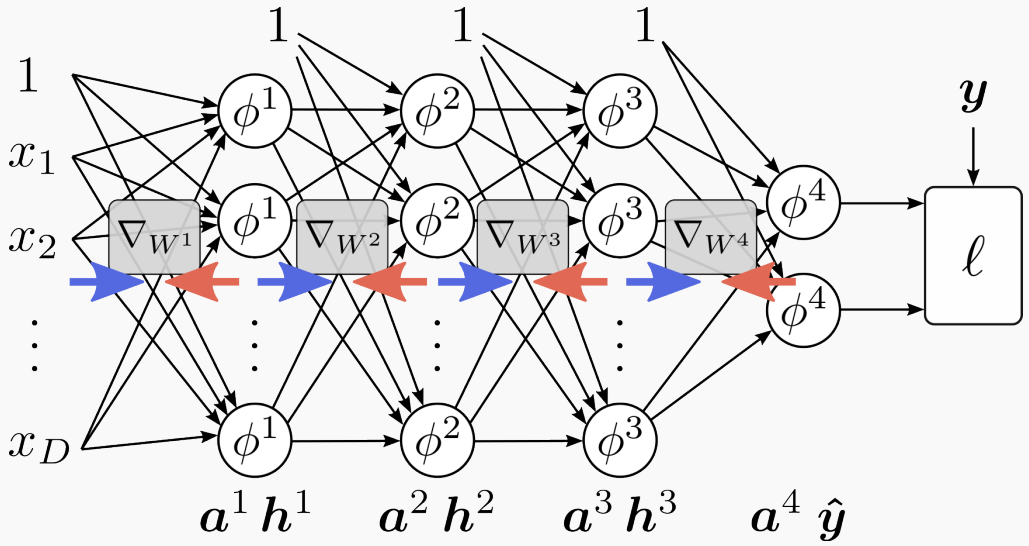
Autodiff in Matrix-Vector Form



and so on . . .

$$\nabla_{\boldsymbol{h}^2} \ell = W^3 \nabla_{\boldsymbol{a}^3} \ell$$

Autodiff in Matrix-Vector Form



With just two rules, we get the derivatives w.r.t. all activations a_i^l and outputs h_i^l.

The derivatives w.r.t. the weights are then given as

$$\frac{\partial \ell}{\partial w_{i,j}^l} = \frac{\partial \ell}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{i,j}^l} = \overbrace{\frac{\partial \ell}{\partial a_j^l}}^{\text{backprop}} \overbrace{h_i^{l-1}}^{\text{forward eval}}$$

Classification with MLPs

Modeling Multi-class Problems

- Recall that in Lecture 5, we discussed **binary classification** for logistic regression
- We used a linear model $f(\boldsymbol{x}) = \boldsymbol{w}^T \phi(\boldsymbol{x})$ (ϕ is a vector of fixed non-linearities) and converted its output to a probability with the sigmoid function:

$$\pi = \sigma\left(\boldsymbol{w}^T \phi(\boldsymbol{x})\right)$$

  π (respectively $1 - \pi$) is interpreted as the model's belief that $y = 1$ (respectively $y = 0$).
- How to generalize this to arbitrary many classes, i.e. to **multi-class** problems?

Modeling Multi-class Problems

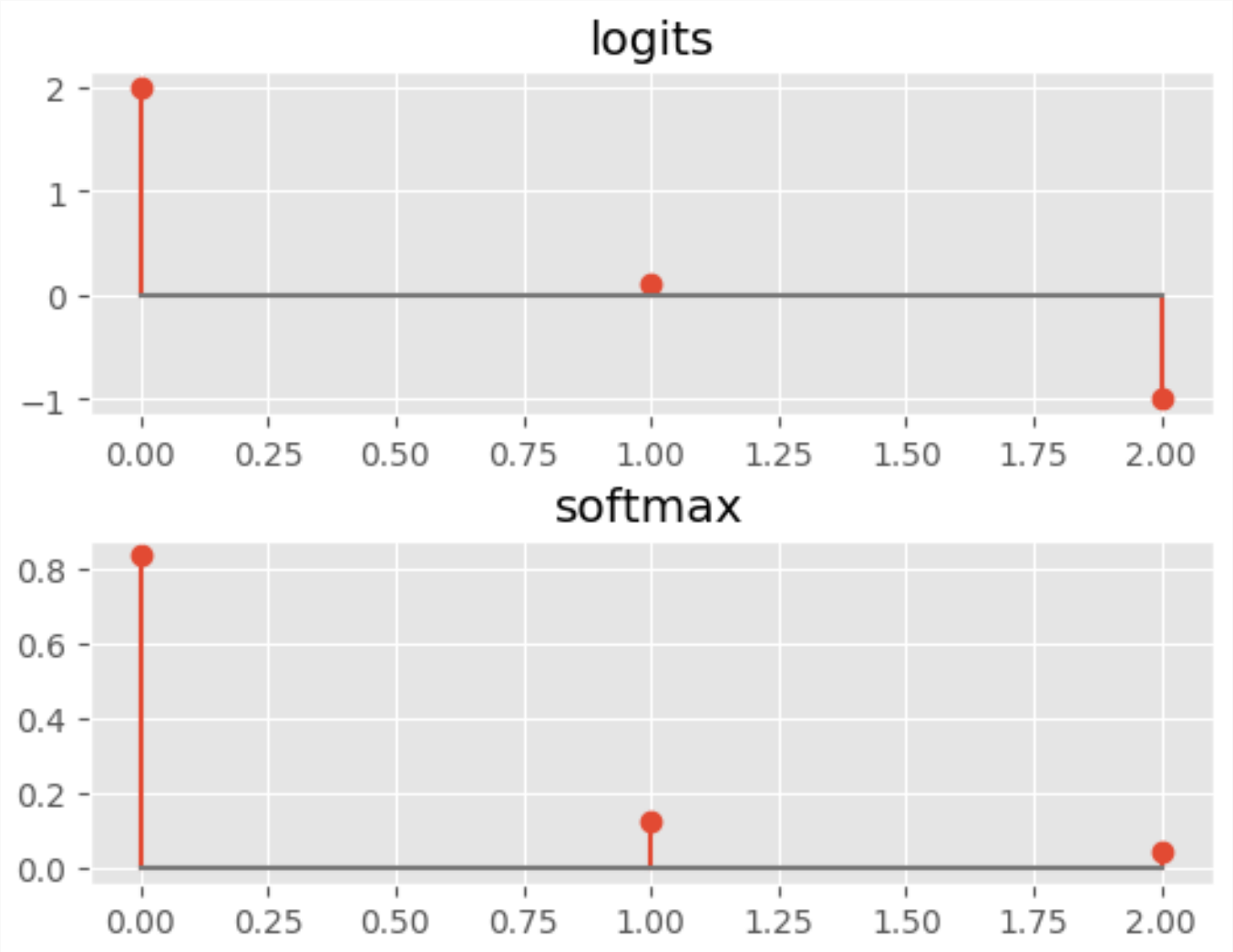
- Let C be the number of classes (e.g. $C = 3$ if classes are 'dog', 'cat', 'rabbit').
- Introduce one output (neuron) a_c per class and let $\boldsymbol{a} = (a_1, \ldots, a_C)^T$ be the C-dimensional output vector
- Convert $\boldsymbol{a} = (a_1, \ldots, a_C)^T$ into a probability distribution over labels. This is achieved with the so-called **softmax function**:

$$\boldsymbol{\pi} = \begin{pmatrix} \pi_1 \\ \vdots \\ \pi_C \end{pmatrix} = softmax(\boldsymbol{a}) = \begin{pmatrix} \frac{\exp(a_1)}{Z} \\ \vdots \\ \frac{\exp(a_C)}{Z} \end{pmatrix}, \; Z = \sum_{i=1}^{C} \exp(a_i)$$

- Evidently, $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_C)^T$ is non-negative and sums to one
- In this context, $\boldsymbol{a}$ are often called the **logits** of the softmax

Softmax Function

Example



- Let $\boldsymbol{a}(\boldsymbol{\theta})$ be the outputs of the model and $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_C)^T = \mathit{softmax}(\boldsymbol{a})$
- Assume that classes are encoded with integers i.e., $y \in \{1, \ldots, C\}$
- Then the **cross-entropy** for one sample $\boldsymbol{x}, y$ is defined as

$$\ell_{CE}(\boldsymbol{\pi}, y) = -\log \pi_y$$

  i.e., the negative of the model's log-probability assigned to the true class (negative **log-likelihood**).
- The **cross-entropy loss** over the whole training set is

$$\mathcal{L}_{CE} = -\frac{1}{N} \sum_{i=1}^{N} \log \pi_{y^{(i)}}^{(i)}$$

- Compute $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{CE}$ with autodiff and perform gradient descent

Gradient of Cross-Entropy

First, we require the gradient of $CE(softmax(\boldsymbol{a}))$.

$$CE = -\log \pi_y \qquad \pi_c = \frac{\exp(a_c)}{\sum_{c'=1}^{C} \exp(a_{c'})}$$

Case 1: $c = y$

$$\frac{\partial CE}{\partial a_c} = -\frac{1}{\pi_y}\left[\overbrace{\frac{\exp(a_c)}{\sum_{c'=1}^{C}\exp(a'_c)}}^{\pi_y} - \overbrace{\frac{\exp^2(a_c)}{\left(\sum_{c'=1}^{C}\exp(a'_c)\right)^2}}^{\pi_y^2}\right] = (\pi_c - 1)$$

Case 2: $i \neq y$

$$\frac{\partial CE}{\partial a_c} = -\frac{1}{\pi_y}\left[-\overbrace{\frac{\exp(a_y)\exp(a_c)}{\left(\sum_{c'=1}^{C}\exp(a'_c)\right)^2}}^{\pi_y \pi_c}\right] = \pi_c$$

After that, just use standard autodiff.

Detailed derivation (not for exam)

(1) derivative of cross-entropy by π_c

$$\frac{\partial CE}{\partial \pi_c} = \begin{cases} -\frac{1}{\pi_c} & \text{if } y = c \\ 0 & \text{otherwise} \end{cases}$$

(2) derivative of softmax

Note that $\frac{\exp(a_c)}{\sum_{c'=1}^{C} \exp(a_{c'})} = \exp(a_c) \left(\sum_{c'=1}^{C} \exp(a_{c'}) \right)^{-1}$

Hence by the product rule

$$\frac{\partial \pi_c}{\partial a_d} = \frac{\partial \exp(a_c)}{\partial a_d} \left(\sum_{c'=1}^{C} \exp(a_{c'}) \right)^{-1} + \exp(a_c) \frac{\partial \left(\sum_{c'=1}^{C} \exp(a_{c'}) \right)^{-1}}{\partial a_d}$$

$$= \frac{\frac{\partial \exp(a_c)}{\partial a_d}}{\sum_{c'=1}^{C} \exp(a_{c'})} - \frac{\exp(a_c) \frac{\partial \sum_{c'=1}^{C} \exp(a_{c'})}{\partial a_d}}{\left(\sum_{c'=1}^{C} \exp(a_{c'}) \right)^2}$$

$$\frac{\frac{\partial \exp(a_c)}{\partial a_d}}{\sum_{c'=1}^{C} \exp(a_{c'})} = \begin{cases} \frac{\exp(a_c)}{\sum_{c'=1}^{C} \exp(a_{c'})} & \text{if } c = d \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\exp(a_c) \frac{\partial \sum_{c'=1}^{C} \exp(a_{c'})}{\partial a_d}}{\left(\sum_{c'=1}^{C} \exp(a_{c'})\right)^2} = \frac{\exp(a_c)\exp(a_d)}{\left(\sum_{c'=1}^{C} \exp(a_{c'})\right)^2}$$

Hence, if $c = d$:

$$\frac{\partial \pi_c}{\partial a_d} = \frac{\exp(a_c)}{\sum_{c'=1}^{C} \exp(a_{c'})} - \frac{\exp^2(a_c)}{\left(\sum_{c'=1}^{C} \exp(a_{c'})\right)^2}$$

and if $c \neq d$:

$$\frac{\partial \pi_c}{\partial a_d} = -\frac{\exp(a_c)\exp(a_d)}{\left(\sum_{c'=1}^{C} \exp(a_{c'})\right)^2}$$

$$\frac{\partial CE}{\partial a_c} = \sum_d \overbrace{\frac{\partial CE}{\partial \pi_d}}^{\text{0, except if } d=y} \frac{\partial \pi_d}{\partial a_c} = -\frac{1}{\pi_y} \frac{\partial \pi_y}{\partial a_c}$$

Case 1: $c = y$

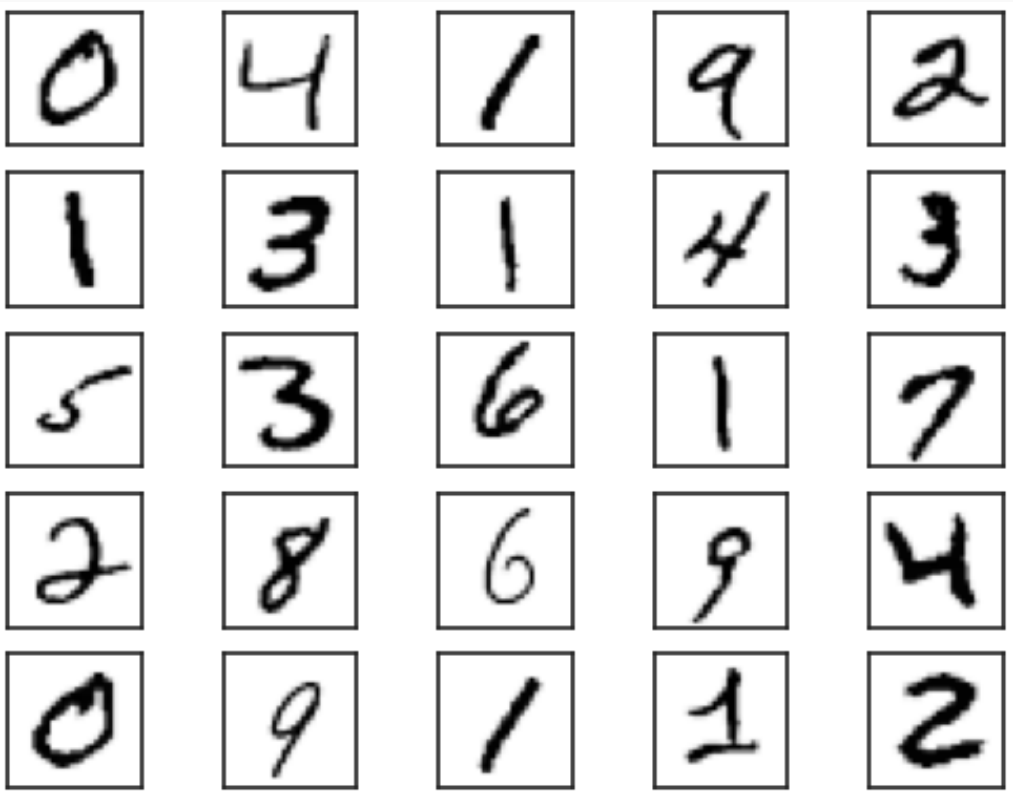
$$\frac{\partial CE}{\partial a_c} = -\frac{1}{\pi_y} \left[\overbrace{\frac{\exp(a_c)}{\sum_{c'=1}^{C} \exp(a'_c)}}^{\pi_y} - \overbrace{\frac{\exp^2(a_c)}{\left(\sum_{c'=1}^{C} \exp(a'_c)\right)^2}}^{\pi_y^2} \right] = (\pi_c - 1)$$

Case 2: $i \neq y$

$$\frac{\partial CE}{\partial a_c} = -\frac{1}{\pi_y} \left[-\overbrace{\frac{\exp(a_y)\exp(a_c)}{\left(\sum_{c'=1}^{C} \exp(a'_c)\right)^2}}^{\pi_y \pi_c} \right] = \pi_c$$

MNIST Digit Classification

- $60k$ training images, $10k$ test images
- Split training images into $50k$ training examples and $10k$ validation examples
- 28×28 pixels, yields 784-dimensional feature vectors



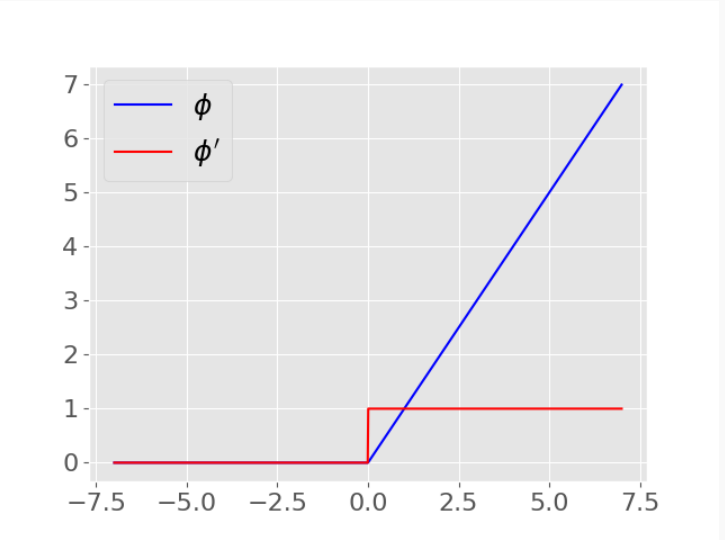
MNIST Digit Classification cont'd

- MLP with two hidden layers, 1000 neurons each
- 10 outputs, one for each class
- ReLU activations for hidden layers
- Outputs are processed by softmax
- Using cross-entropy loss
- Initialize parameters randomly (small uniform noise)
- 500 iterations of gradient descent (often called **epochs**)
- Step-size $\eta = 0.1$

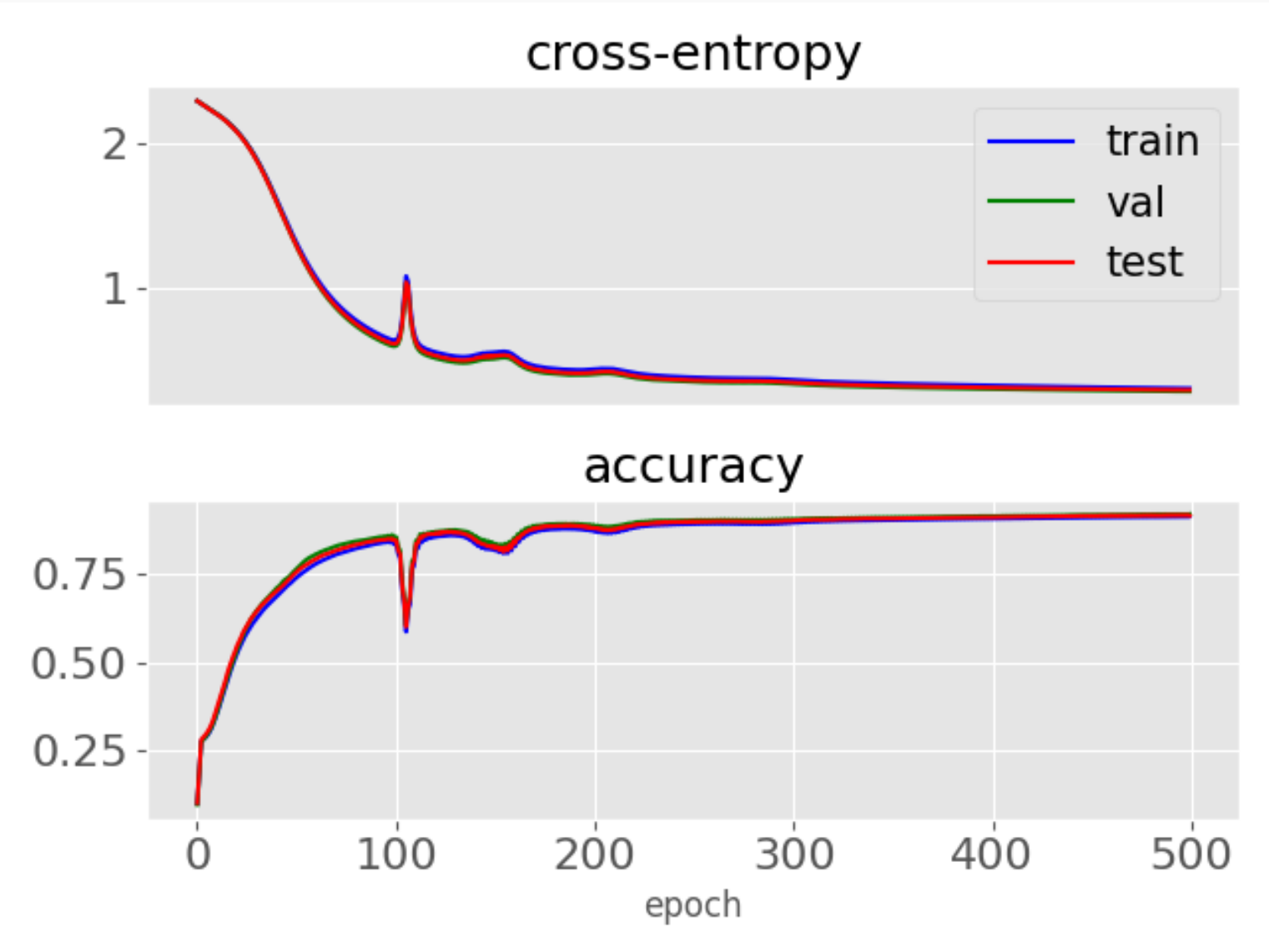
Rectified Linear Unit (ReLU)

$\phi(x) = \max(x, 0)$

$\phi'(x) = \mathbb{1}(x > 0)$

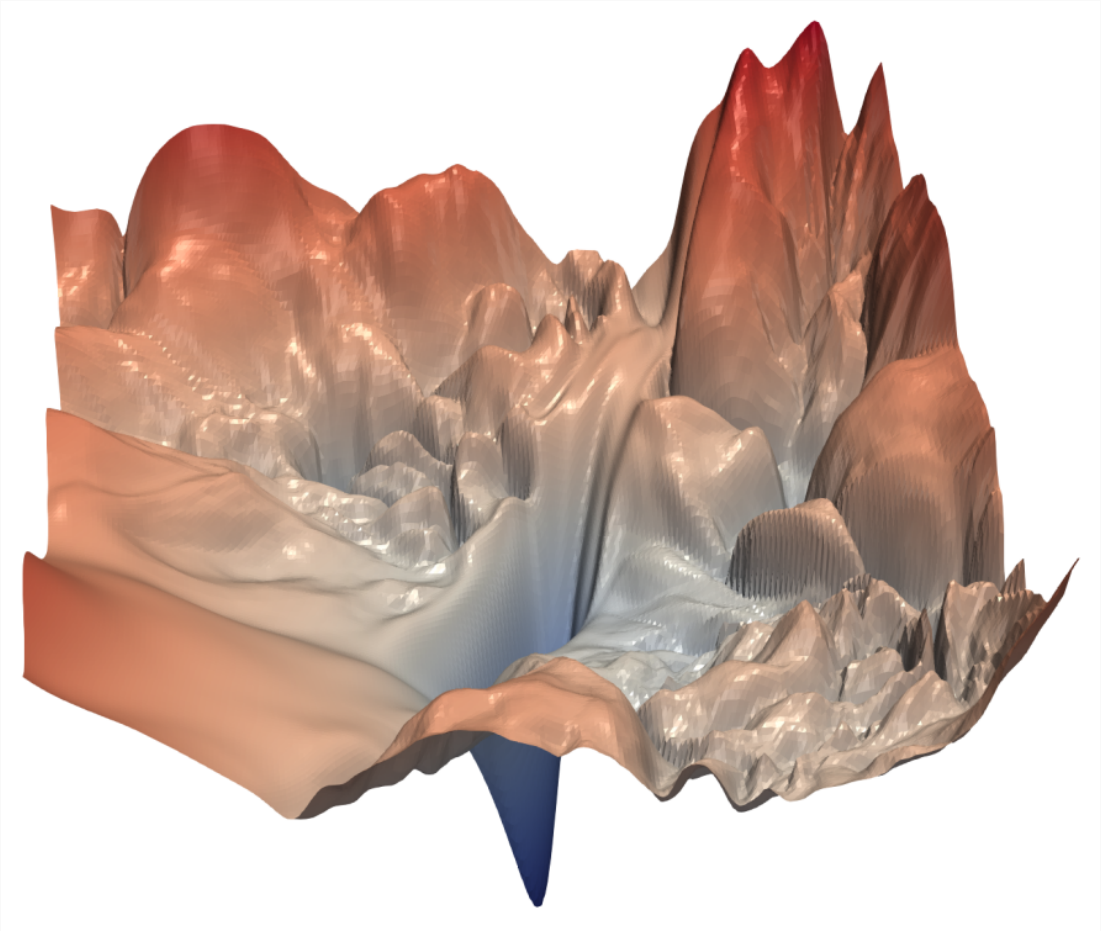


MNIST Digit Classification cont'd



Loss Landscape of Neural Networks (Non-Convex)

Training loss in neural networks is highly non-convex, and we can find only local minima. However, in practice they work very well.



Source: Li et al., *Visualizing the Loss Landscape of Neural Nets*, NeurIPS 2018.

Neural Networks in Practice

Frameworks

- Many ML frameworks for training neural nets exist
- **Theano** (2007) was one of the first widely used packages
- Todays common ML packages include **Tensorflow**, **PyTorch** and **Jax**
- Only forward pass needs to be specified—gradients are obtained via autodiff
- These packages also have various gradient-based optimizers implemented (adaptive versions of standard gradient-descent)
- These frameworks also work well with dedicated hardware, i.e. **Graphics Processing Units (GPUs)**, **Tensor Processing Units (TPUs)**, etc., allowing dramatic speedups

Pre-processing

Input features might live on very different **scales**. For example, assume that some features are measured in *mm* while others are measured in *km*. Thus, good practice is to **normalize** features before using them in neural networks.

Often the pre-processing is computed from the training set. The same pre-processing, however, is also applied to validation and test sets.

Pre-processing cont'd

Zero-Mean/Unit-Variance Normalization: Let $\bar{\boldsymbol{x}}$ and $\boldsymbol{s}$ be the empirical mean and standard deviation vectors, i.e. $\bar{\boldsymbol{x}} = \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \boldsymbol{x}^{(i)}$ and $\boldsymbol{s} = \sqrt{\frac{1}{N_{train}} \sum_{i=1}^{N_{train}} (\boldsymbol{x}^{(i)} - \bar{\boldsymbol{x}})^2}$ (here, the square acts element-wise). Then the zero-mean, unit-variance normalization is given as:

$$\boldsymbol{x}^{(i)} \leftarrow \frac{\boldsymbol{x}^{(i)} - \bar{\boldsymbol{x}}}{\boldsymbol{s}}$$

Here the division acts element-wise. The transformed data has zero mean and unit standard deviation (unit variance) in each dimension.

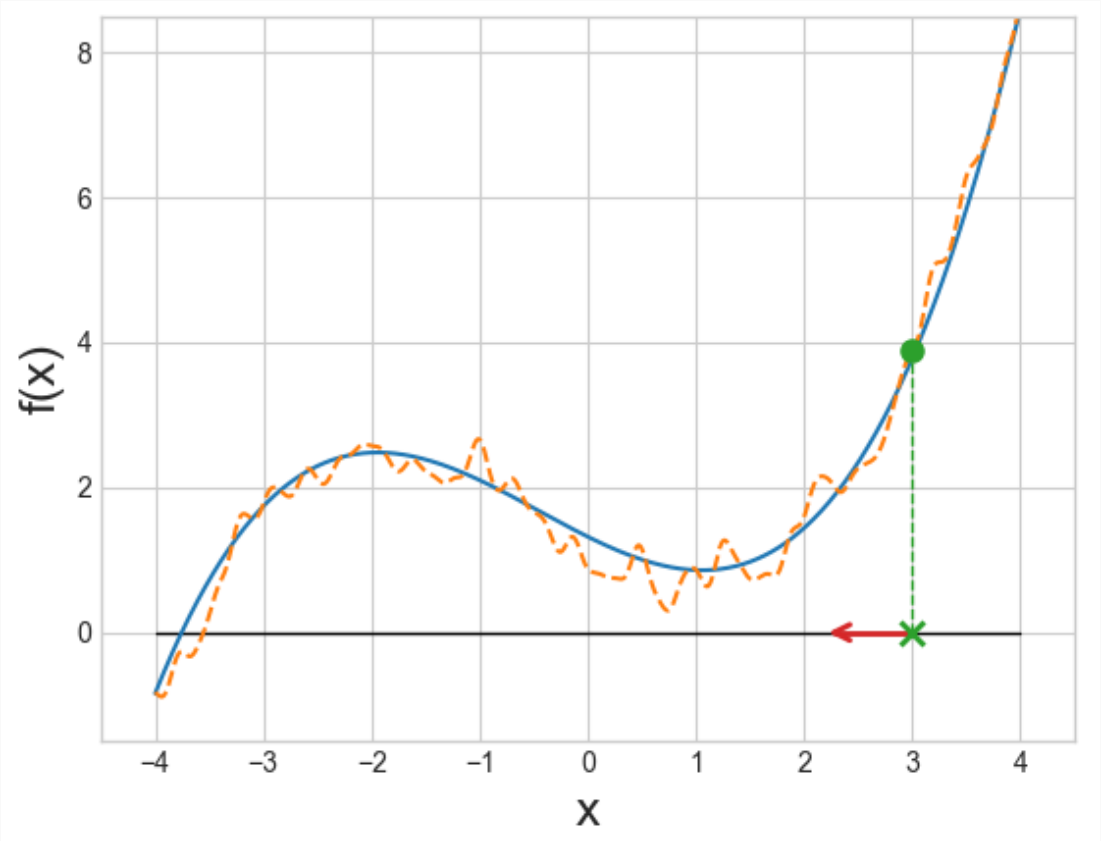
Pre-processing cont'd

Min-Max Normalization: Let $\boldsymbol{x}_{min}$ and $\boldsymbol{x}_{max}$ be the vectors containing minima and maxima over the training set for each dimension, i.e. $x_{min,i} = \min_{i=1\ldots N_{train}} x^{(i)}$, $x_{max,i} = \max_{i=1\ldots N_{train}} x^{(i)}$ Then the min-max is given as:

$$\boldsymbol{x}^{(i)} \leftarrow \frac{\boldsymbol{x}^{(i)} - \boldsymbol{x}_{min}}{\boldsymbol{x}_{max} - \boldsymbol{x}_{min}}$$

Here the division acts element-wise. The transformed data is scaled to interval $[0, 1]$ in each dimension. Additionally, one might subtract 0.5, so that the data is scaled to interval $[-0.5, 0.5]$.

Stochastic Gradient Descent (SGD)



`stochastic_gradient_descent_demo.py`

Stochastic Gradient Descent – Motivation

- Gradient of training loss

$$\nabla_{\theta}\mathcal{L}_{train} = \frac{1}{N}\sum_{i=1}^{N}\nabla_{\theta}\ell(y^{(i)}, \hat{y}^{(i)})$$

  requires a whole pass over train data (epoch), **for each single gradient update**

- Inefficient, due to **data redundancies**
- Extreme example of data redundancy: train data duplicated 10 times yields same gradient $\Rightarrow$ wasted computation
- Redundancies in real data are not as extreme, but still lead to wasteful computation
- **Let's do quicker updates**

- At the beginning of each epoch, shuffle train samples and divide them into **mini-batches** of size B
- In each epoch, iterate over mini-batches
  - compute loss $\mathcal{L}_{batch} = \frac{1}{B} \sum_{i=1}^{B} \ell(y^{(i)}, \hat{y}^{(i)})$ for current mini-batch
  - update $\boldsymbol{\theta}$ with gradient descent step:

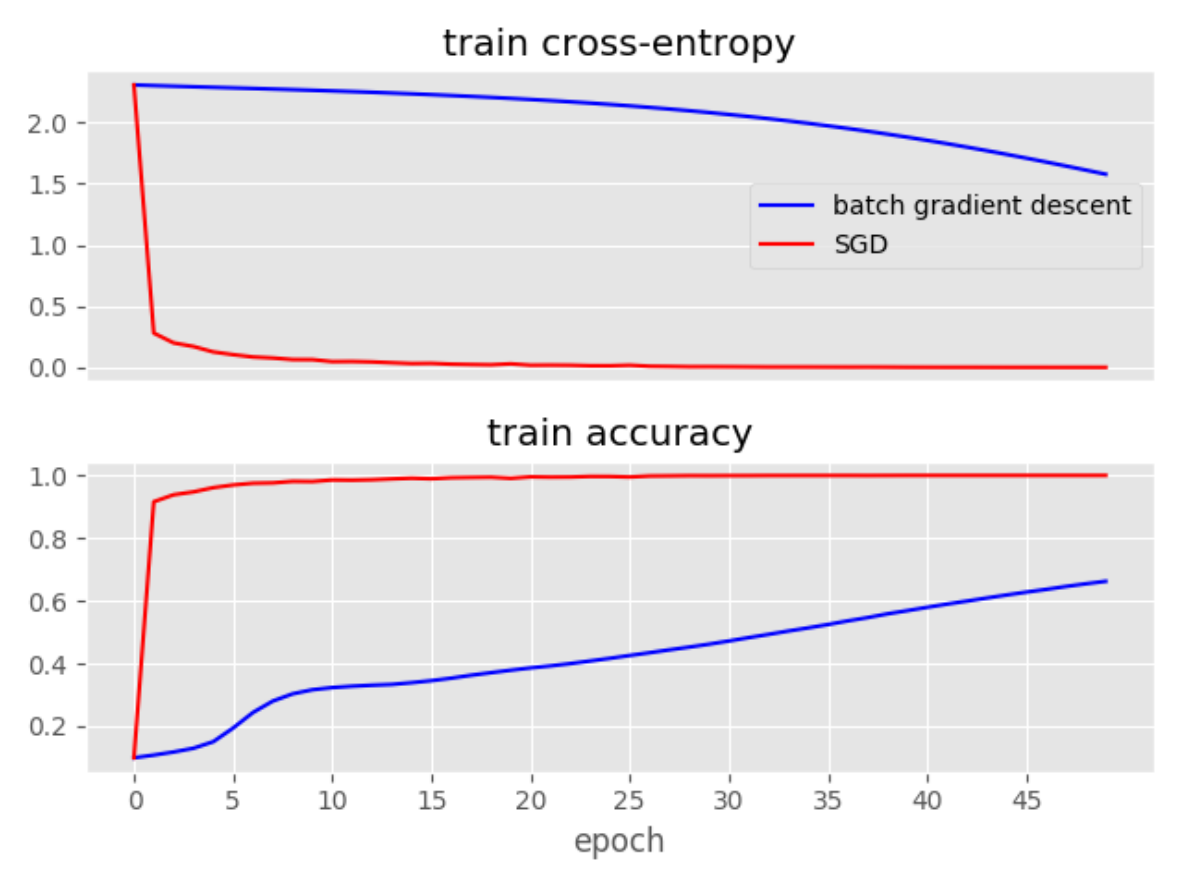
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta_k \nabla_{\boldsymbol{\theta}} \mathcal{L}_{batch}$$

SGD is a Sound Optimizer

- Noisy gradients, but unbiased (correct expectation)
- If $\sum_k \eta_k = \infty$ and $\sum_k \eta_k^2 < \infty$, where k is the epoch counter, SGD converges to a local minimum of $\mathcal{L}_{train}$! (*)
- In contrast to SGD, standard gradient descent is often called batch gradient descent ((!) don't confuse with mini-batches)

*In practice, one often uses a simple stepsize schedule

- MNIST data
- MLP with 2 hidden layers, 200 units each, ReLU activation
- Batch size $B = 100$ (600 batches per epoch)
- Stepsize $\eta = 0.1$

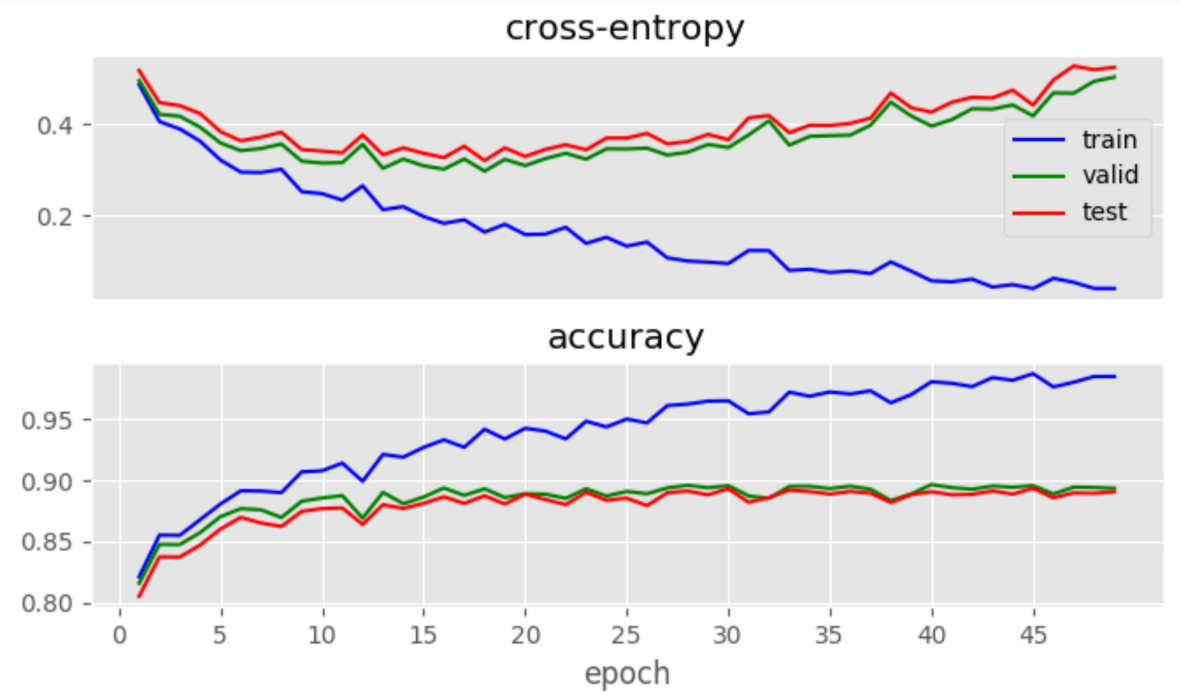


Avoiding Overfitting

Neural networks can have millions and billions of parameters and are thus prone to overfitting. Some common techniques to avoid overfitting are

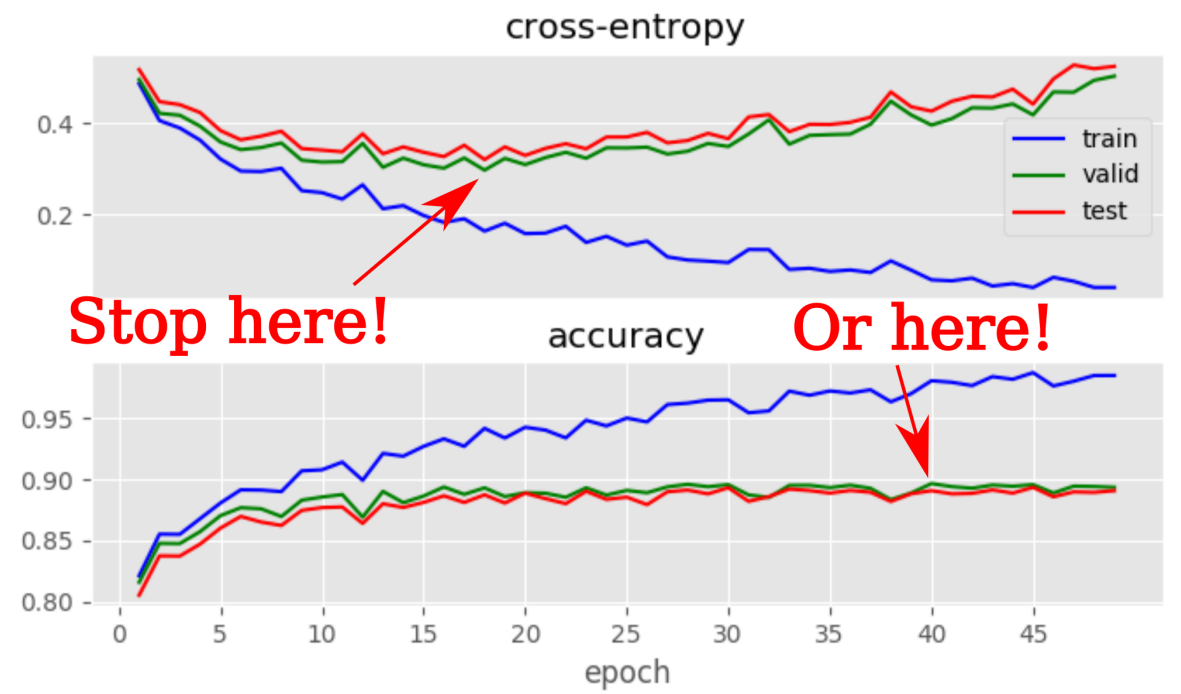
- **Early stopping**
- **Regularization**
- **Dropout**

Early Stopping



- Stop, when validation error starts to increase
- Epoch number as "hyper parameter"
- Save **checkpoint**, whenever observing new minimum $\mathcal{L}_{val,min}$
- Optionally: stop training when K_{fail} epochs have passed without decreasing $\mathcal{L}_{val,min}$

Early Stopping



- Stop, when validation error starts to increase
- Epoch number as "hyper parameter"
- Save **checkpoint**, whenever observing new minimum $\mathcal{L}_{val,min}$
- Optionally: stop training when K_{fail} epochs have passed without decreasing $\mathcal{L}_{val,min}$

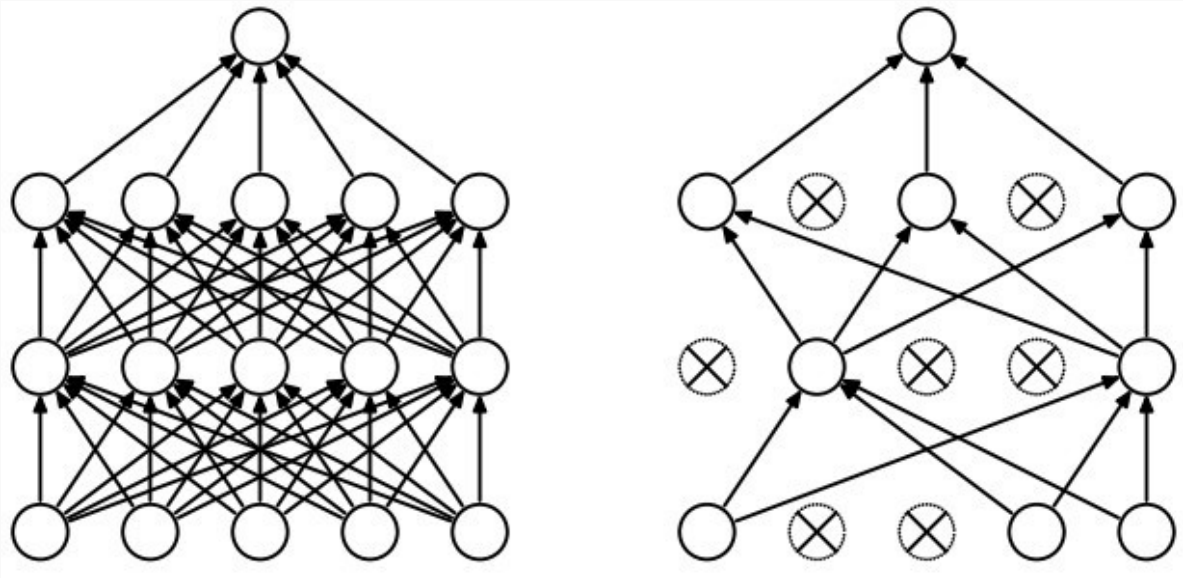
Regularization

- Neural nets with smaller weights yield "simpler functions"
- Add weight regularizer to loss

$$\mathcal{L}_{train}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \ell(y^{(i)}, f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})) + \lambda R(\boldsymbol{\theta})$$

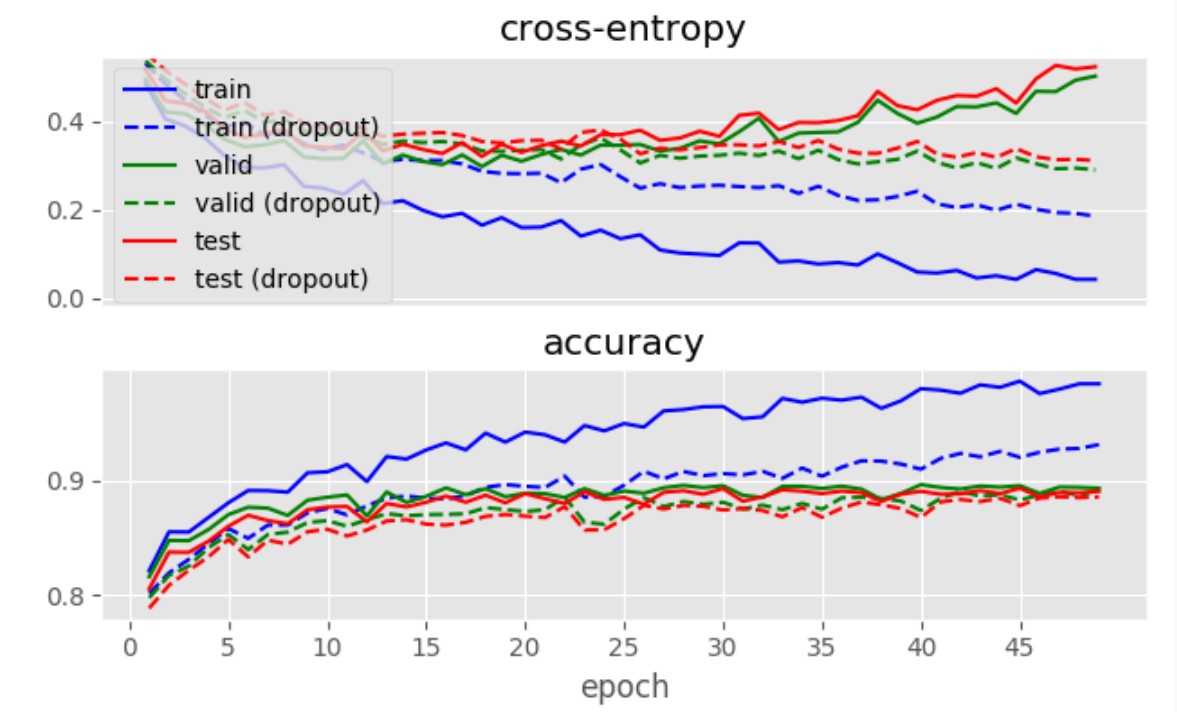
- **Trade-off parameter** λ balances between data fit and simplicity
- λ can be set via cross-validation
- Typical regularization terms:
  - $L2$**-regularizer** $R = \|\boldsymbol{\theta}\|_2^2$
  - $L1$**-regularizer** $R = \|\boldsymbol{\theta}\|_1$
- $L1$-regularizer induces **sparsity**, i.e. many weights will be exactly zero and can be pruned

Dropout [Srivastava et al., JMLR 2014]



- Randomly drop out inputs and/or hidden units
- Avoids co-adaption of units
- Can be interpreted as implicit ensemble
- Dropout probabilities are hyperparameters
- During testing, no dropout is applied, but outputs are scaled by dropout probability to compensate for the additional input

- Fashion-MNIST
- Train/validation/test split: 50k/10k/10k
- 3-layer MLP with 1000 hidden units each
- SGD, stepsize 0.1, 50 epochs
- Dropout at inputs $p = 0.5$



Summary

- MLPs: stacked matrix multiplications and element-wise non-linearities
- Autodiff dramatically simplifies learning neural nets
- Stochastic gradient descent for quicker updates
- Regularization, early stopping, dropout
- See further: "Deep Learning" course with Legenstein & Özdenizci
  - specialized architectures (convolutional neural nets, recurrent neural nets, transformers, . . .)
  - advanced learning techniques
  - advanced applications