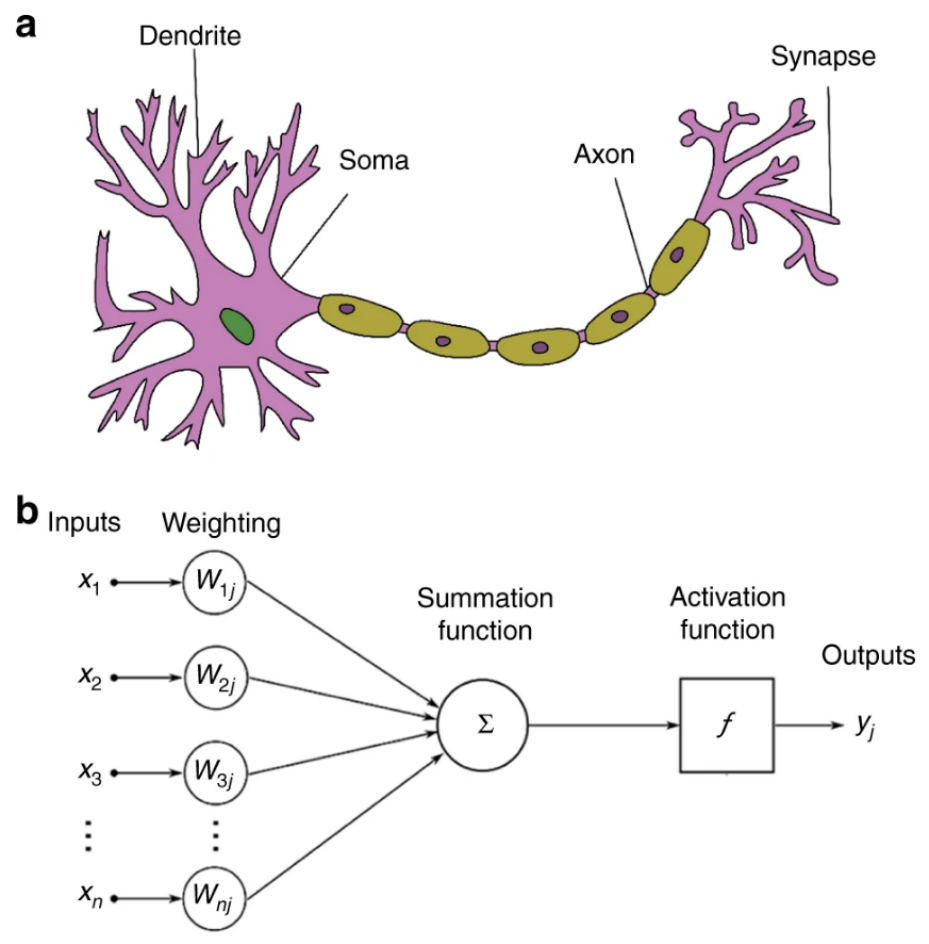
Neural Networks II

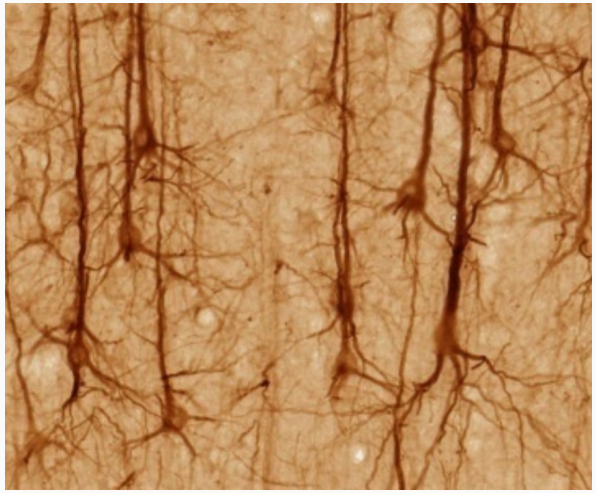
Machine Learning 1 — Lecture 8

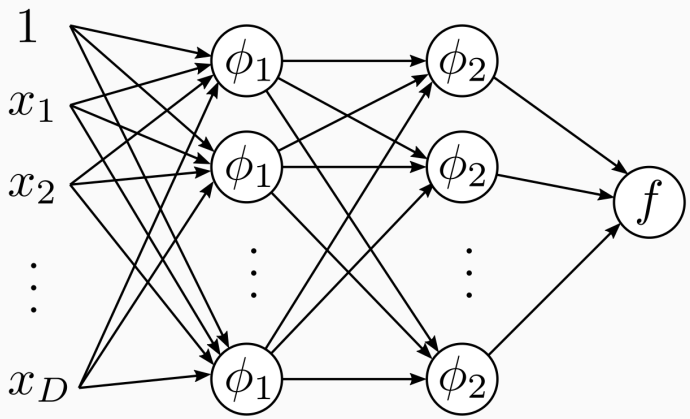
3rd May 2022

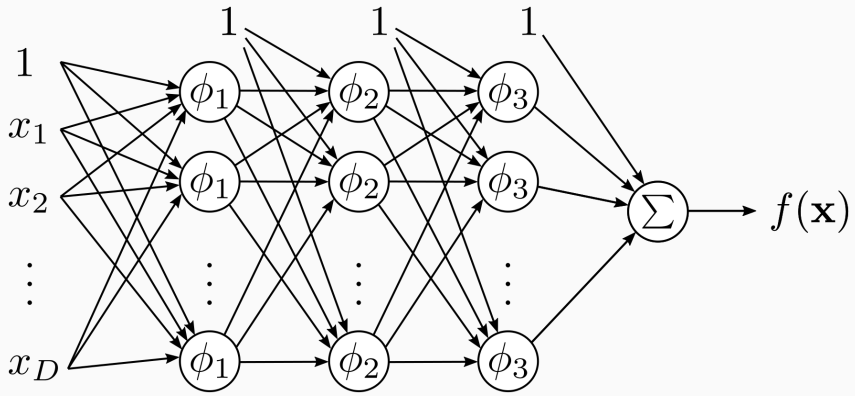
Robert Peharz
Institute of Theoretical Computer Science
Graz University of Technology

Biological vs. Artificial Neural Networks

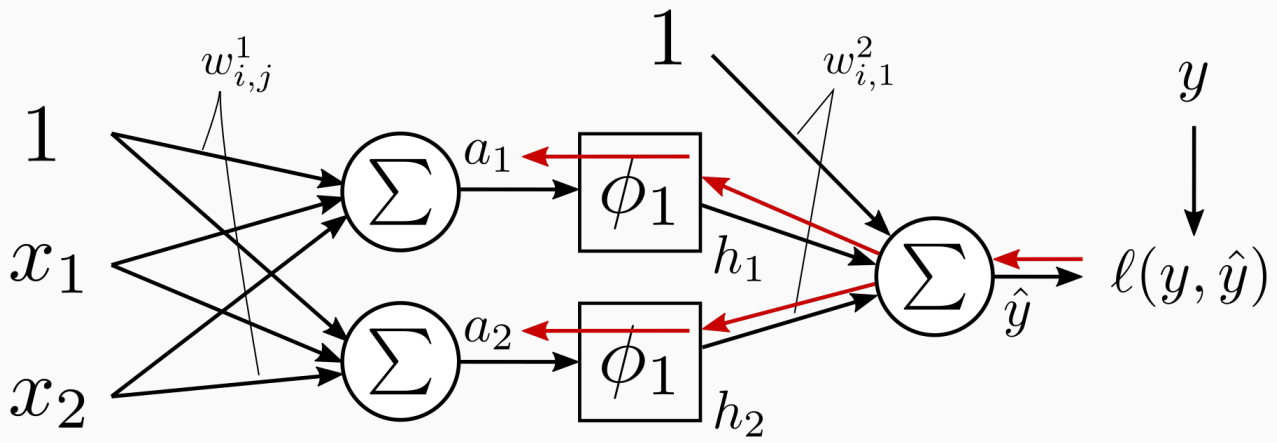








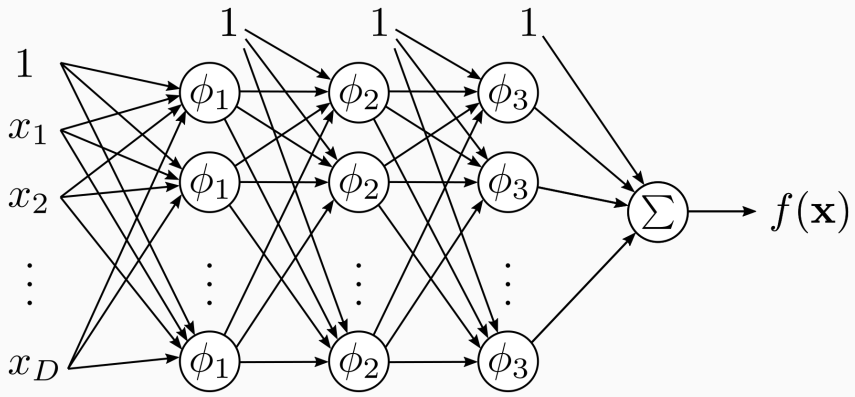
- **Shallow networks** (1-layer)
  - **universal approximators**
  - However, some function classes require **exponentially** many neurons, in the number of inputs
- **Deep networks**
  - can often represent represent the same functions with only **polynomial size**
  - This phenomenon is called **expressive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



- $\frac{\partial \ell}{\partial \hat{y}} =: B_1$ (closed form for most losses)
- $\frac{\partial \ell}{\partial h_i} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_i} = B_1 w^2_{i,1} =: B_{2,i}$ $(\hat{y} = \sum_{i=0}^{2} w^2_{i,1} h_i)$
- $\frac{\partial \ell}{\partial a_i} = \frac{\partial \ell}{\partial h_i} \frac{\partial h_i}{\partial a_i} = B_{2,i}\, \phi_1'(a_i) =: B_{3,i}$ (closed form ϕ_1')
- $\frac{\partial \ell}{\partial w^2_{i,1}} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w^2_{i,1}} = B_1 h_i$ $(\hat{y} = \sum_{i=0}^{2} w^2_{i,1} h_i)$
- $\frac{\partial \ell}{\partial w^1_{i,j}} = \frac{\partial \ell}{\partial a_j} \frac{\partial a_j}{\partial w^1_{i,j}} = B_{3,j} x_i$ $(a_j = \sum_{i=0}^{2} w^1_{i,j} x_i)$

Matrix-Vector Notation for MLPs

Matrix-Vector Notation for MLPs



- Consider an MLP with L layers, where the first $L-1$ layers are "**hidden layers**" and layer L is the **output layer**
- The layers have $H_1, H_2, \ldots, H_L$ units
- Let h_j^i be the output of the j^{th} unit in the i^{th} layer
- Collect all outputs of the i^{th} layer in a vector (including a constant "dummy" feature for the bias):

$$\boldsymbol{h}^i = (1, h_1^i, h_2^i, \ldots, h_{H_i}^i)^\top$$

Matrix-Vector Notation for MLPs cont'd

- We can also interpret the input features as "zeroth layer:"

$$\boldsymbol{h}^0 = (1, x_1, x_2, \ldots, x_D)^\top$$

- For $i \geq 1$, the output of unit h_j^i is given as

$$h_j^i = \phi_i(a_j^i)$$

where ϕ_i is the non-linear **activation function** for layer i and the **activation** a_j^i is given as

$$a_j^i = \sum_{k=1}^{H_{i-1}} w_{k,j}^i \, h_k^{i-1}$$

- Here $w_{k,j}^i$ is the weight on the connection from k^{th} unit in layer $i-1$ to j^{th} unit in layer i.

Vector Notation for MLPs cont'd

- The activation is simply an inner product

$$a_j^i = {\boldsymbol{w}_j^i}^T \boldsymbol{h}^{i-1}$$

  where $\boldsymbol{w}_j^i = (w_{0,j}^i, w_{1,j}^i, \ldots, w_{H_{i-1},j}^i)^T$ is the vector of all weights connecting to unit j.
- We collect all $\boldsymbol{w}_j^i$ in a $H_{i-1} \times H_i$ **weight matrix** W^i:

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

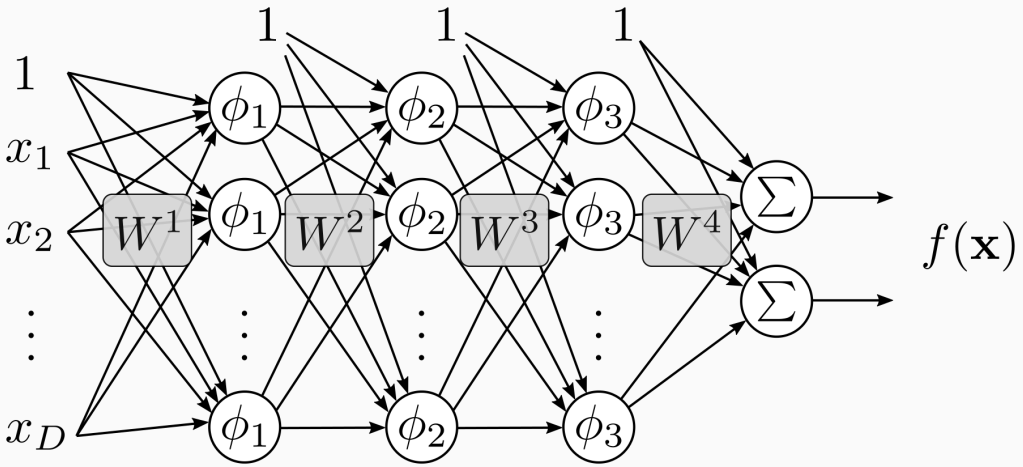
- All activations in layer i obtained with a simple matrix multiplication:

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

  where $\boldsymbol{a}^i = \left(a_1^i, a_2^i, \ldots, a_{H_i}^i\right)^T$
- The outputs of layer i are then given as

$$\boldsymbol{h}^i = \left(1, \phi_i(a_1^i), \phi_i(a_2^i), \ldots, \phi_i(a_{H_i}^i)\right)^T$$ (i.e., ϕ_i acts element-wise)

Writing a 3-layer MLP in One Line



$$\boldsymbol{a}^4 = \hat{\boldsymbol{y}} = f(\boldsymbol{x})$$

$$\boldsymbol{h}^3$$

$$\boldsymbol{a}^3$$

$$\boldsymbol{h}^2$$

$$\boldsymbol{a}^2$$

$$\boldsymbol{h}^1$$

$$\boldsymbol{a}^1$$

$$f(\boldsymbol{x}) = \mathrm{W}^{4\,T}\,\phi_3\Big(\mathrm{W}^{3\,T}\,\phi_2\Big(\mathrm{W}^{2\,T}\,\phi_1\Big(\overbrace{\mathrm{W}^{1\,T}\boldsymbol{x}}\Big)\Big)\Big)$$

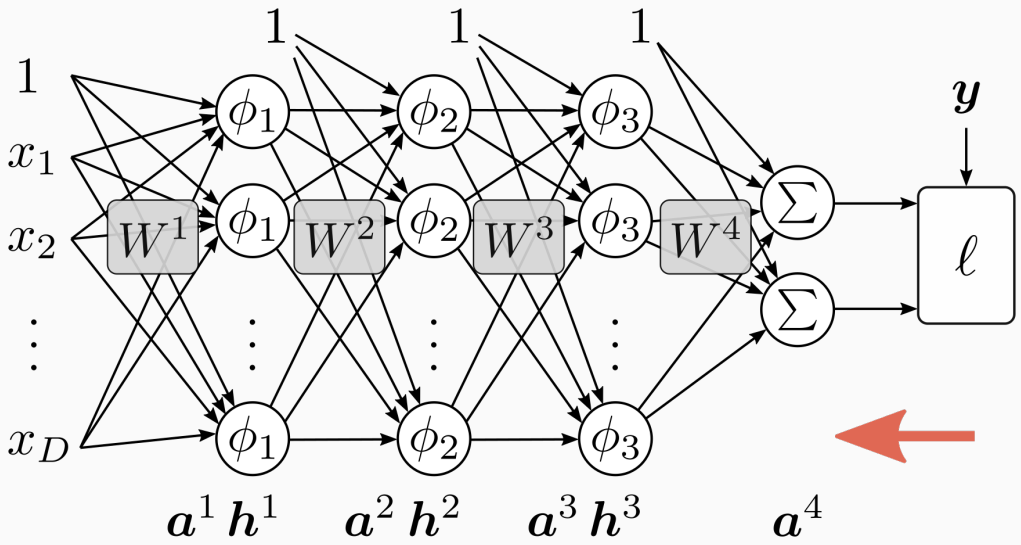
Backprop in Matrix-Vector Form

Backprop: recursively compute the gradient of the loss ℓ with respect to $\boldsymbol{a}^i$, $\boldsymbol{h}^i$ and W^i using the chain rule of calculus. We need three types of derivatives:

- Gradient of loss w.r.t. output layer
- Jacobian between $\boldsymbol{h}^{i-1}$ and $\boldsymbol{a}$. Recall from Lecture 2 that the Jacobian of a linear function $A\boldsymbol{x}$ is A.
- Jacobian between $\boldsymbol{a}^i$ and $\boldsymbol{h}^i$. Recall from Lecture 2 that the Jacobian of an element-wise function $\phi(\boldsymbol{x})$ is

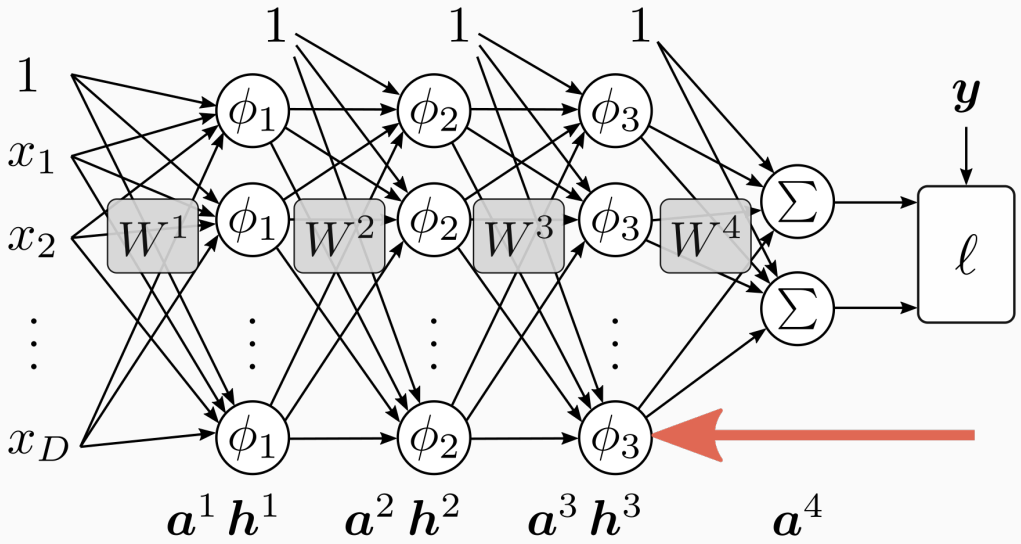
$$J = \begin{pmatrix} \frac{\partial \phi}{\partial x}(x_1) & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{\partial \phi}{\partial x}(x_D) \end{pmatrix}$$

Backprop in Matrix-Vector Form



The gradient of the loss w.r.t. the output layer $\nabla_{\boldsymbol{a}^4} \ell$ is available in closed form for all losses used in practice (e.g., squared loss, cross-entropy)

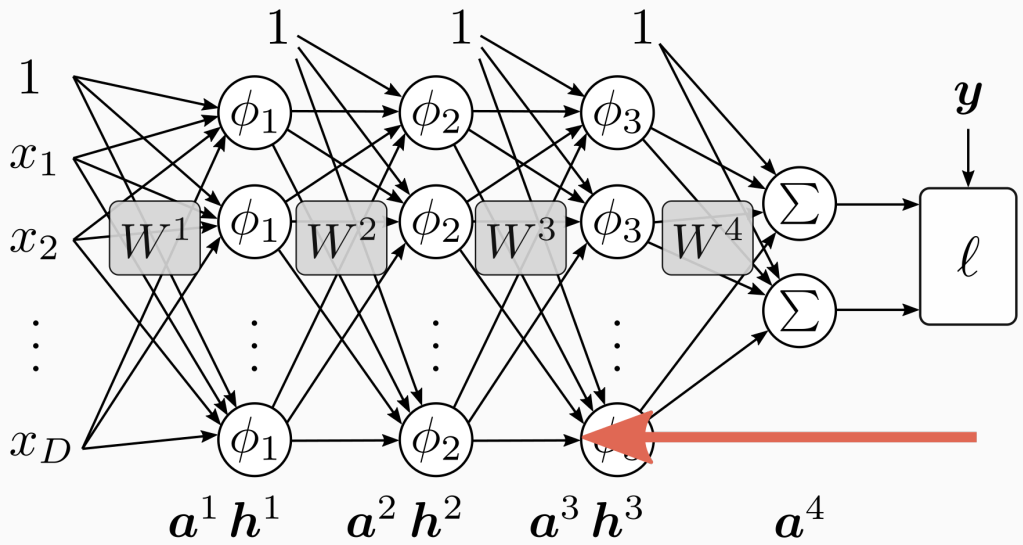
Backprop in Matrix-Vector Form



Since the activation is a linear function of the previous outputs, $\boldsymbol{a}^4 = W^{4\,T}\boldsymbol{h}^3$, we have

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

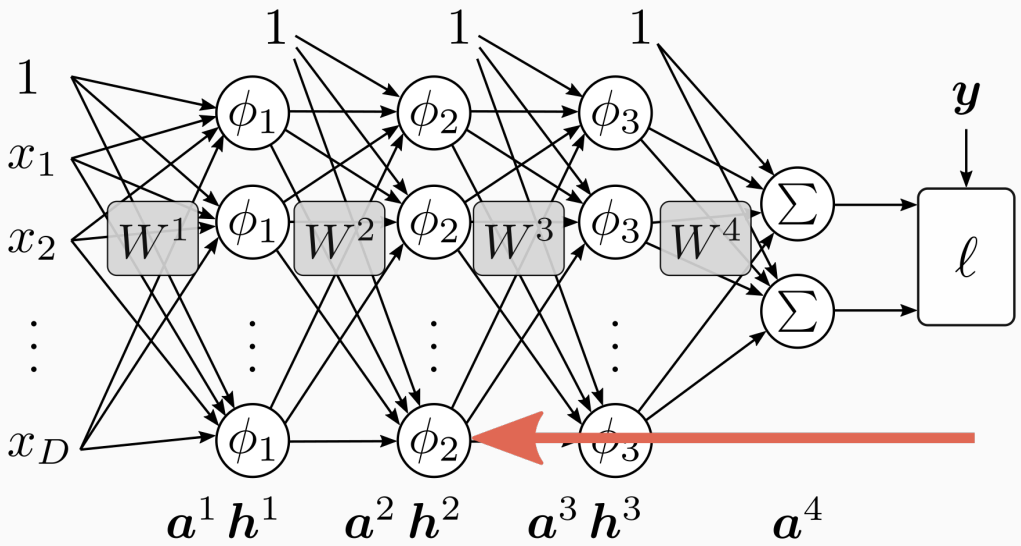
Backprop in Matrix-Vector Form



Since $\boldsymbol{h}^3 = \phi_3(\boldsymbol{a}^3)$ (element-wise function) we have

$$\nabla_{\boldsymbol{a}^3} \ell = \mathit{diag}(\phi_3'(\boldsymbol{a}^3)) \nabla_{\boldsymbol{h}^3} \ell$$

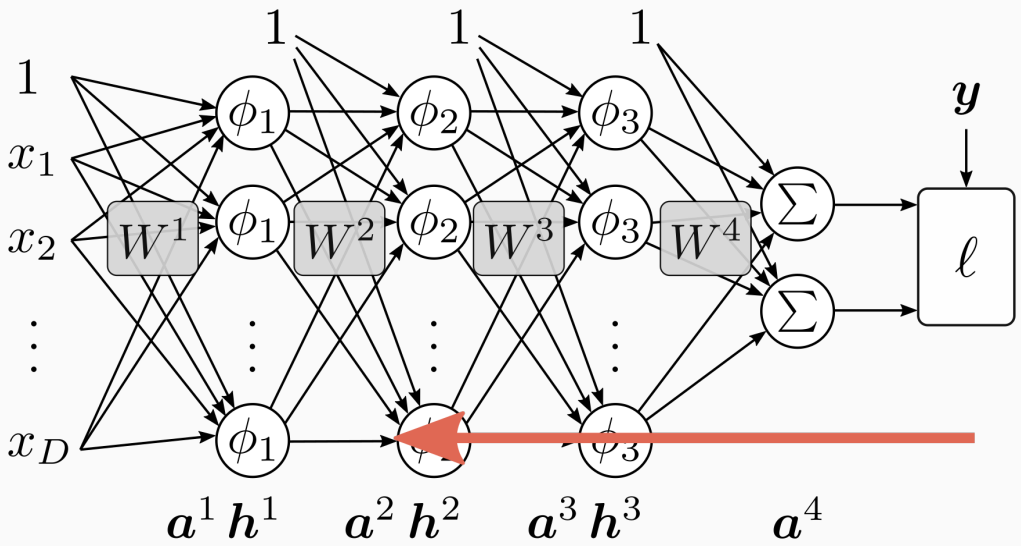
Backprop in Matrix-Vector Form



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

and so on . . .

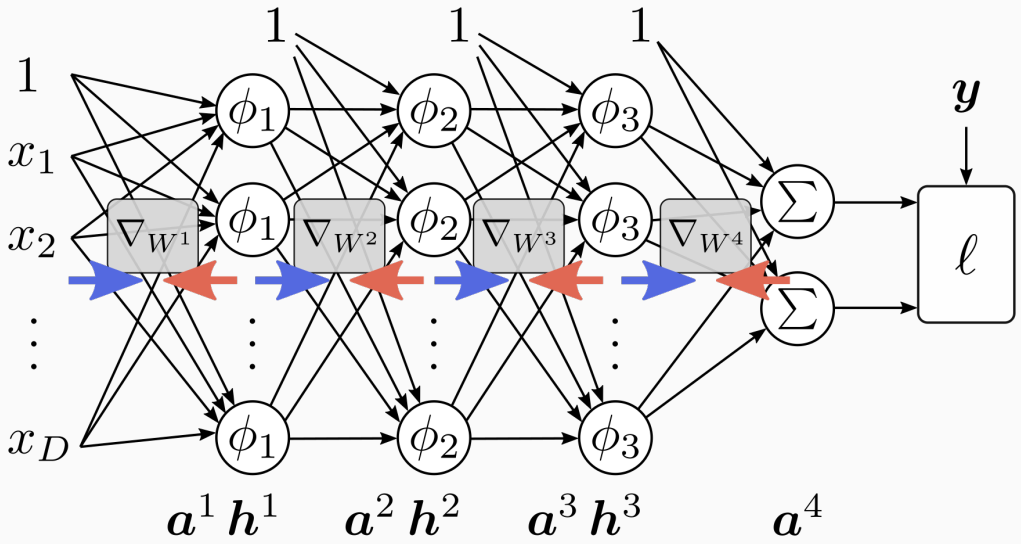
Backprop in Matrix-Vector Form



$$\nabla_{\boldsymbol{a}^2} \ell = \mathit{diag}(\phi_2'(\boldsymbol{a}^2)) \, \nabla_{\boldsymbol{h}^2} \ell$$

and so on . . .

Backprop in Matrix-Vector Form



Since each weight occurs in a multiplicative way in $a_j^i = \sum_{k=1}^{H_{i-1}} w_{k,j}^i \, h_k^{i-1}$, its partial derivative is

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

Classification with MLPs

Modeling Multi-class Problems cont'd

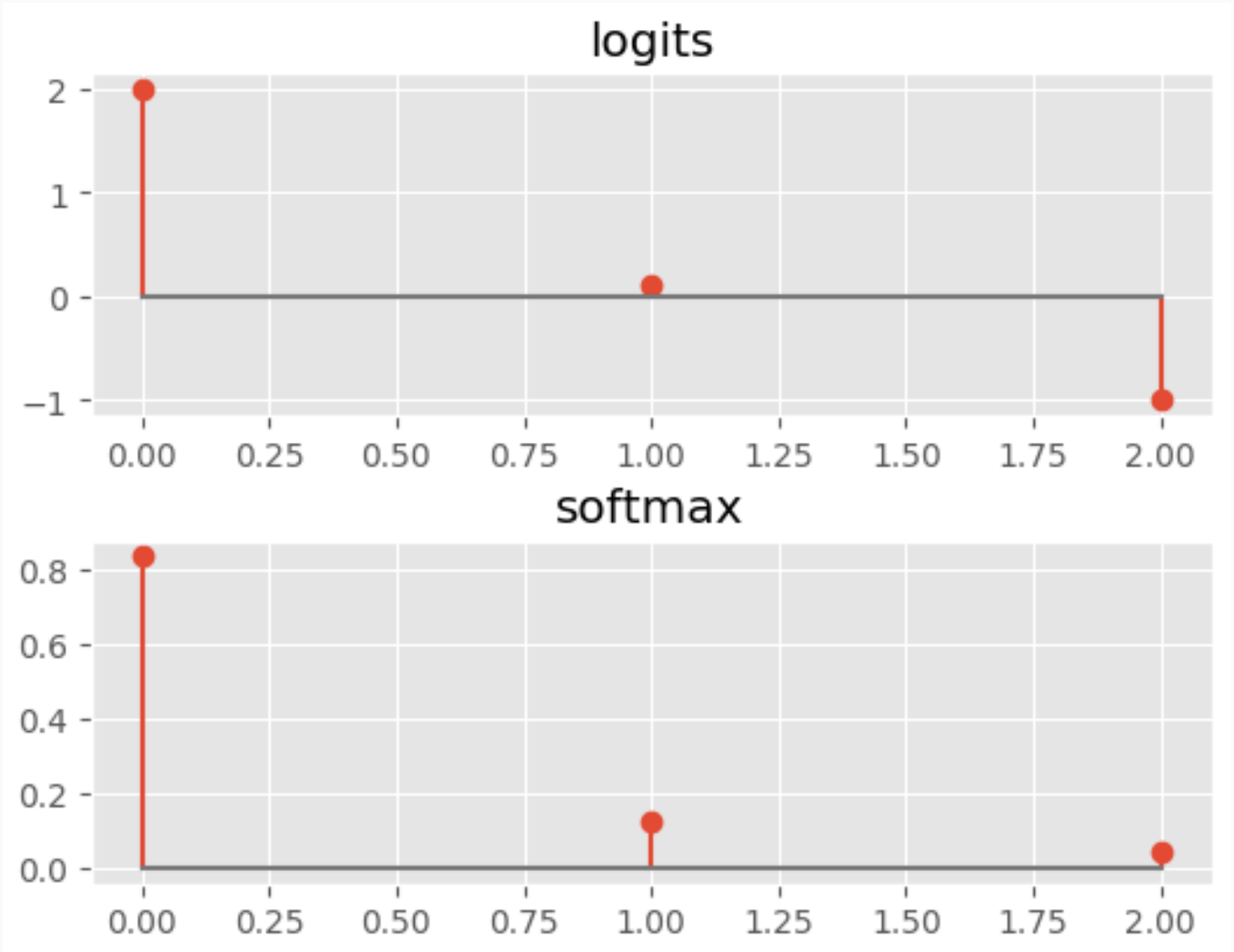
- Let C be the number of classes (e.g. $C = 3$ if classes are $\mathcal{C} = \{\text{'dog'}, \text{'cat'}, \text{'rabbit'}\}$)
- Introduce one output a_c per class and let $\boldsymbol{a}$ be the C-dimensional output vector
- Convert $\boldsymbol{a} = (a_1, \ldots, a_C)^T$ into a probability distribution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

$$\hat{y}_c = softmax_c(\boldsymbol{a}) = \frac{\exp(a_c)}{\sum_{c'=1}^{C} \exp(a_{c'})}$$

- Evidently, $\hat{\boldsymbol{y}} = (\hat{y}_1, \ldots, \hat{y}_C)^T$ is non-negative and sums to one, i.e. a probability distribution
- In this context, $\boldsymbol{a}$ are often called the **logits** of the softmax

Softmax Function

Example



- Let $\boldsymbol{a}$ be the logits output of the model (neural net, logistic regression) and $\hat{\boldsymbol{y}} = (\hat{y}_1, \ldots, \hat{y}_C)^T = \mathit{softmax}(\boldsymbol{a})$
- Assume that classes are encoded with integers $1 \ldots C$
- Then the **cross-entropy** for one sample $\boldsymbol{x}, y$ is defined as

$$CE = -\log \hat{y}_y$$

  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

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

Gradient of Cross-Entropy

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

$$CE = -\log \hat{y}_y \qquad \hat{y}_i = \frac{\exp(a_i)}{\sum_{c'=1}^{C} \exp(a_{c'})}$$

Case 1: $i = y$

$$\frac{\partial CE}{\partial a_i} = -\frac{1}{\hat{y}_y}\left[\overbrace{\frac{\exp(a_i)}{\sum_{c'=1}^{C}\exp(a'_c)}}^{\hat{y}_y} - \overbrace{\frac{\exp^2(a_i)}{\left(\sum_{c'=1}^{C}\exp(a'_c)\right)^2}}^{\hat{y}_y^2}\right] = (\hat{y}_i - 1)$$

Case 2: $i \neq y$

$$\frac{\partial CE}{\partial a_i} = -\frac{1}{\hat{y}_y}\left[-\overbrace{\frac{\exp(a_y)\exp(a_i)}{\left(\sum_{c'=1}^{C}\exp(a'_c)\right)^2}}^{\hat{y}_y\hat{y}_i}\right] = \hat{y}_i$$

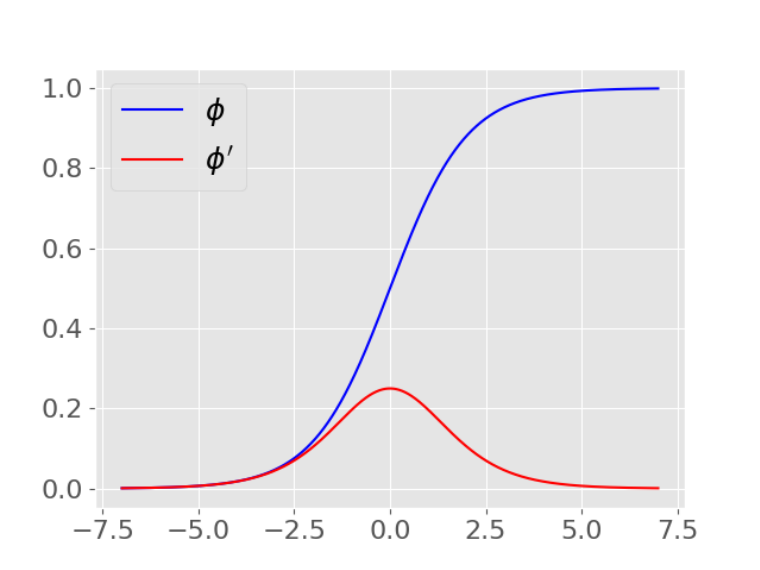
Derivatives of Common Activation Functions

The Jacobian between $\boldsymbol{a}^i$ and $\boldsymbol{h}^i$ is $diag(\phi'(\boldsymbol{a}^i))$, i.e. the diagonal matrix containing local derivatives of the activation function ϕ. This is easy, since ϕ are typically picked so that ϕ' is known.

Sigmoid

$\phi(x) = 1/(1 + exp(-x))$

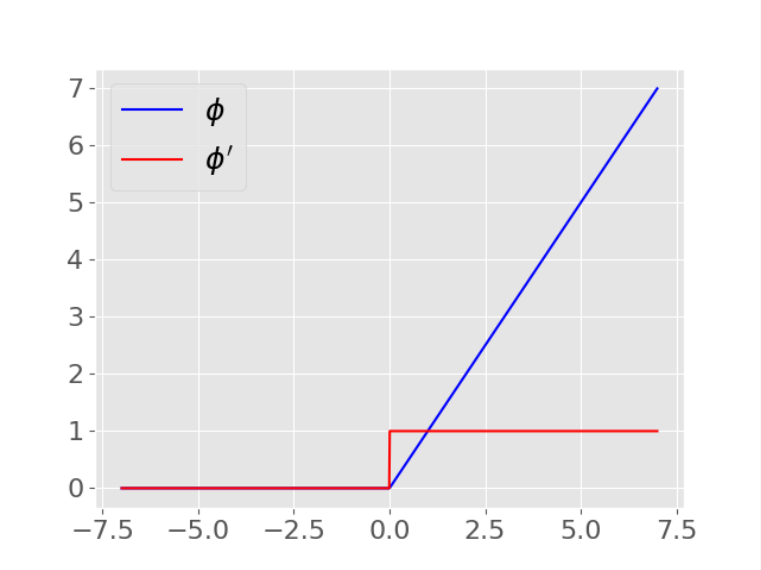
$\phi'(x) = \phi(x)\,(1 - \phi(x))$



Rectified Linear Unit (ReLU)

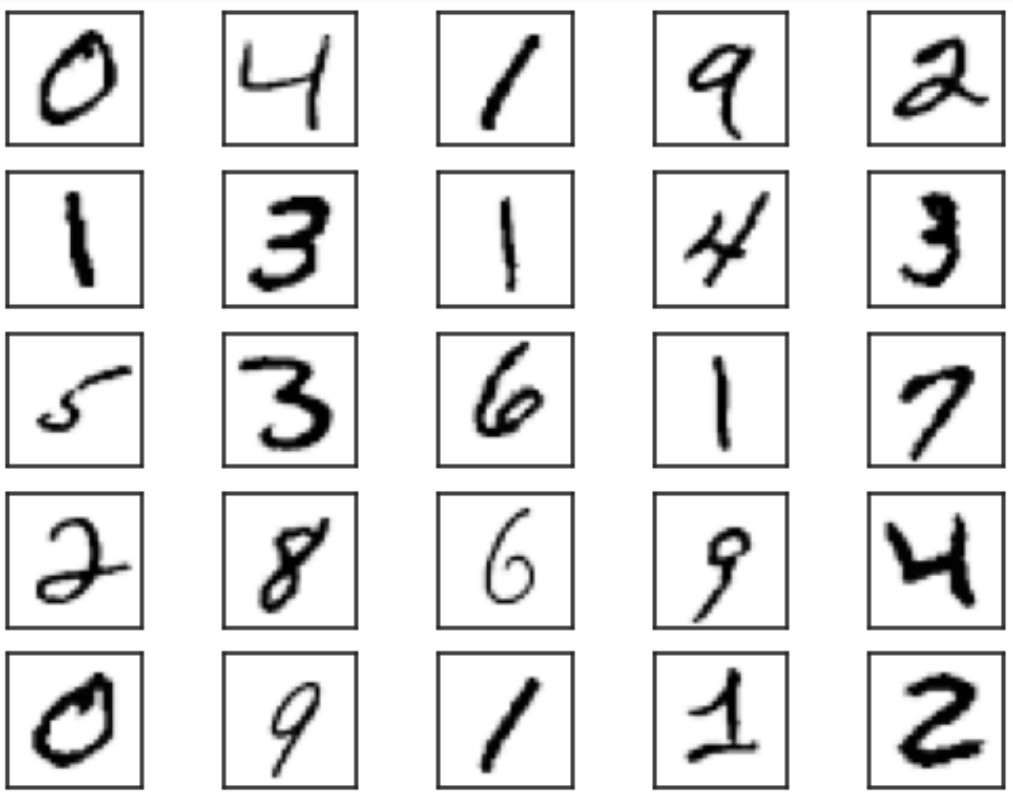
$\phi(x) = \max(0, x)$

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



MNIST Digit Classification

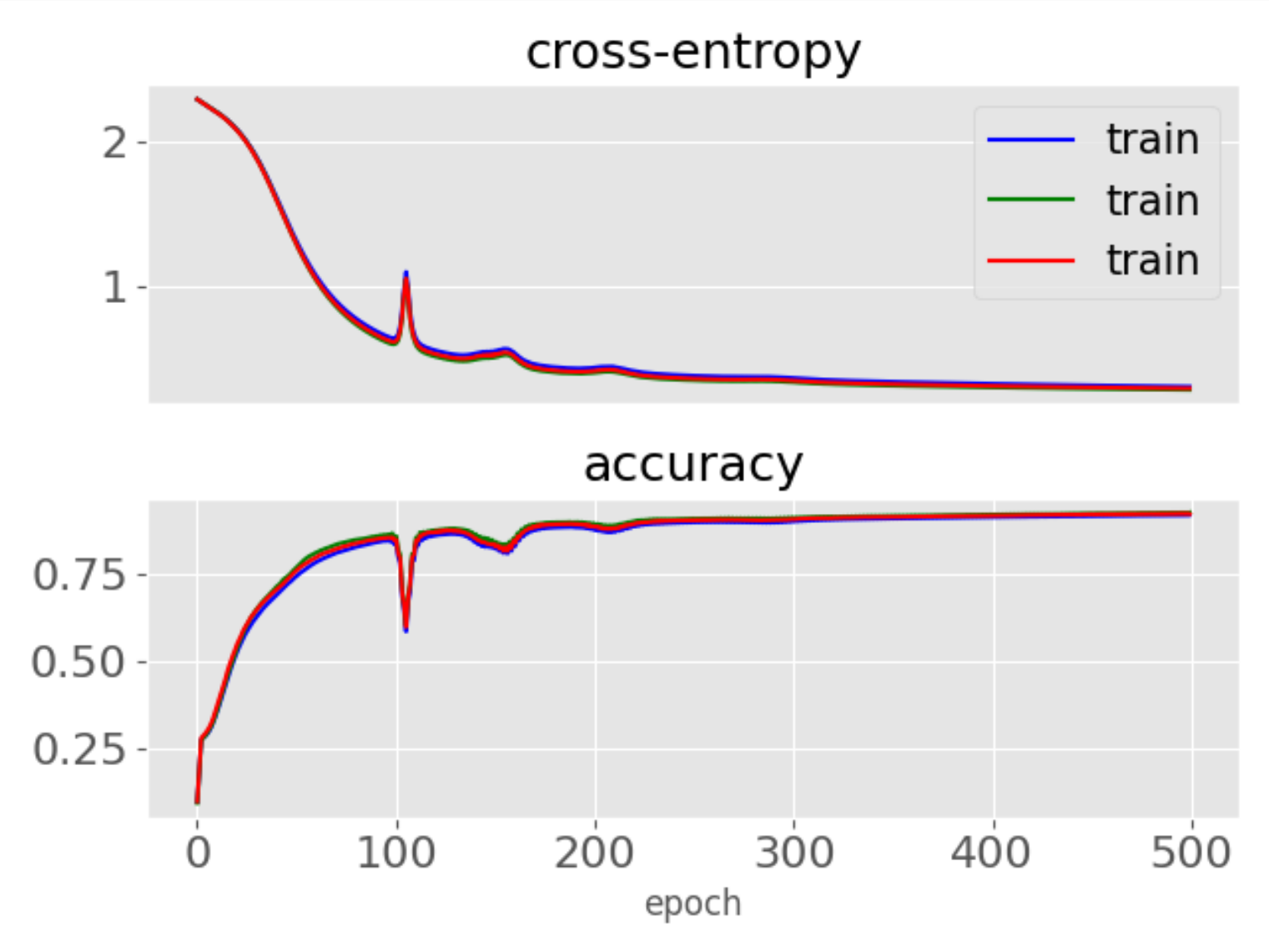
- $60k$ training images, $10k$ test images
- Split training images into $50k$ training examples and $10k$ validation examples
- 28×28 pixels, yields 784-dimensional feature vectors



MNIST Digit Classification cont'd

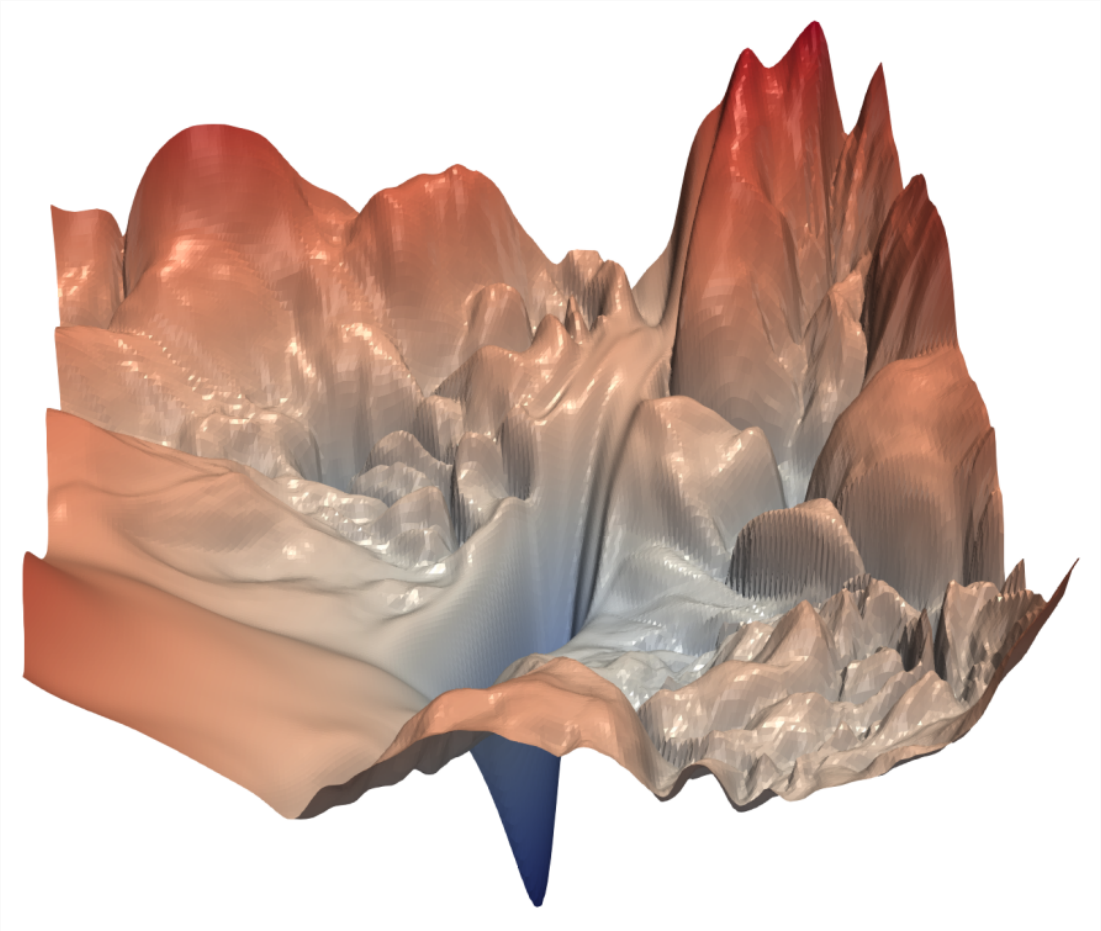
- MLP with two hidden layers, 1000 neurons each
- 10 outputs, one for each class
- ReLU activations for hidden layers
- Outputs are processed by softmax
- Using cross-entropy loss
- Initialize parameters randomly (small uniform noise)
- 500 iterations of gradient descent (often called **epochs**)
- Step-size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

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

Automatic Differentiation

- Backpropagation amounts to the chain rule of calculus
- This process can be automated, yielding **automatic differentiation (autodiff)**
- Only the forward pass needs to be specified, the gradient is computed automatically
- *Theano* (2007) was one of the first widely used packages facilitating autodiff
- Common ML packages facilitating autodiff include nowadays *Tensorflow*, *PyTorch* and *Jax*
- Autodiff is convenient, but one has to understand how it works!

Parallelism and Hardware Accelerators

- Main bottleneck in neural networks: matrix multiplications
- Also "more advanced" architectures require linear algebra routines as core computational blocks
- Linear algebra computations benefit dramatically from parallelized implementation on specialized hardware, such as **Graphics Processing Units (GPUs)**
- The last decade has seen tremendous increase in hardware and software technology for implementing neural networks, e.g. **Tensor Processing Units (TPUs)**
- Common ML frameworks (Tensorflow, PyTorch) facilitate CPU/GPU/TPU computation

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 – Motivation

- Gradient descent on the loss function computes

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

- 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 $\boldsymbol{w}$ with gradient descent step:

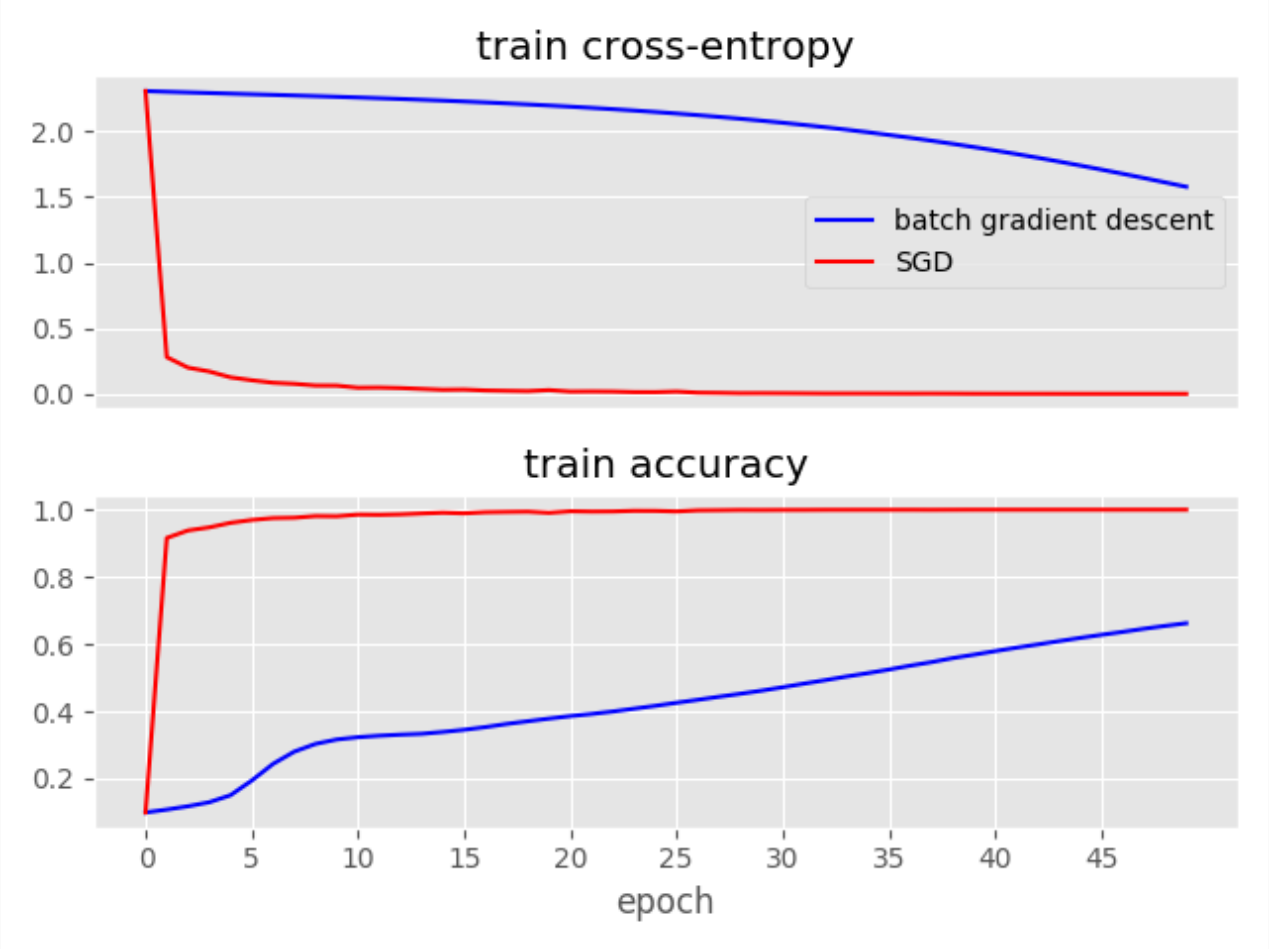
$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta_k \nabla_{\boldsymbol{w}} \mathcal{L}_{batch}$$

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

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

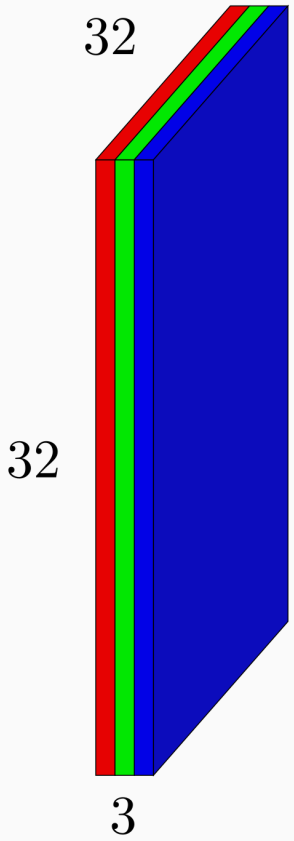


Convolutional Neural Networks — Le Cun et al. 1989

- Convolutional Neural Networks (ConvNets)
- Use extensively **2D-convolutions** which are reminiscent to low-level computation on the visual cortex
- State-of-the-art in image classification (and other visual tasks)
- Krizhevsky et al., 2012 kicked off most of the current excitement about Deep Learning, by achieving new state-of-the-art on ImageNet

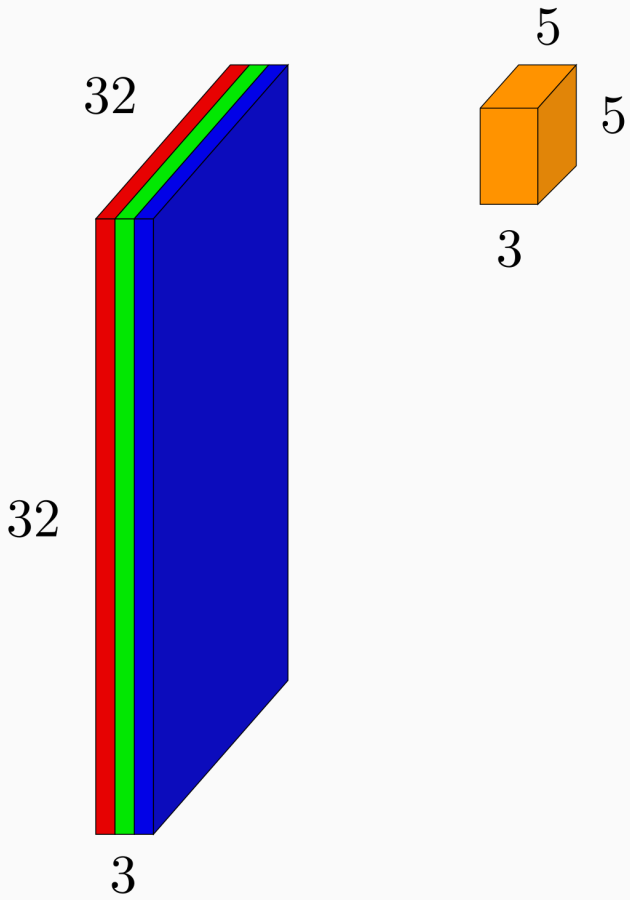
Convolutional Layers

- Start with an image of dimensions $W^0 \times H^0 \times C^0$
- Here $W^0 = H^0 = 32$ and $C^0 = 3$ (RGB image)
- Image interpreted as a **feature map** $\boldsymbol{h}^0$



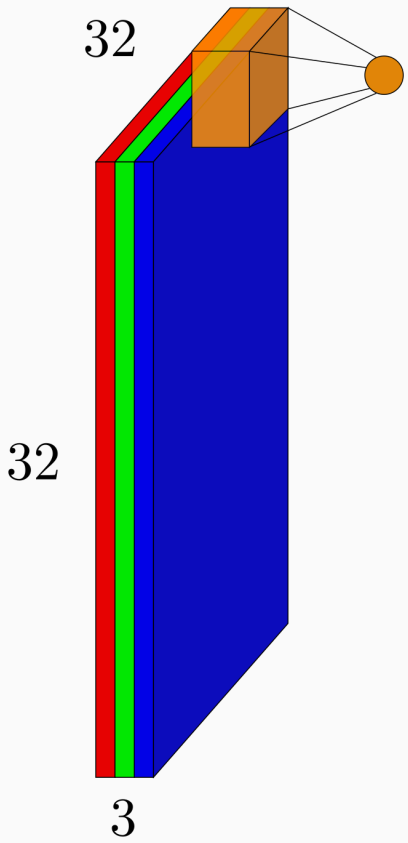
Convolutional Layers

- The **convolutional kernel (filter)** W is a tensor of dimensions $K^1 \times K^1 \times C^0$ (one filter per channel)
- Here, **width of the kernel** is $K^1 = 5$



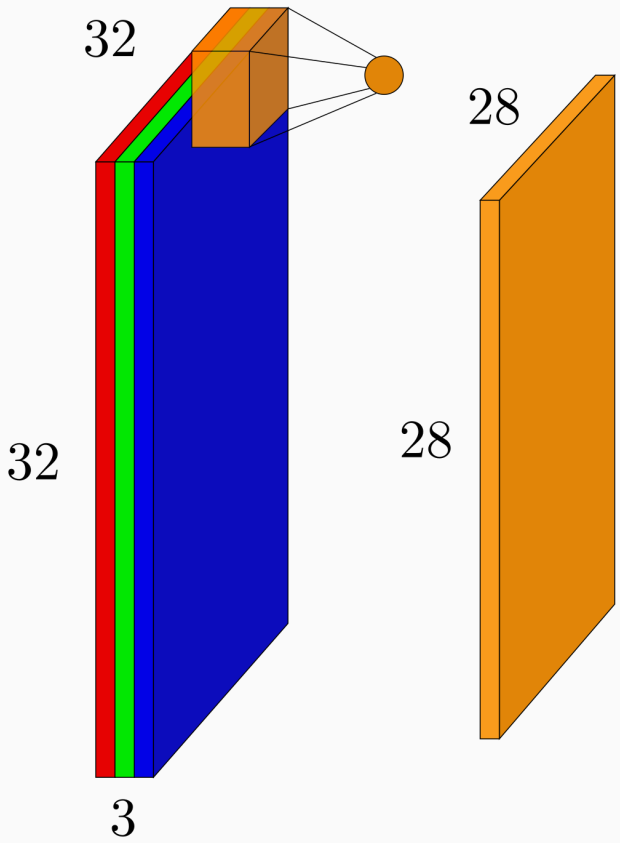
Convolutional Layers

- Perform 2D-convolution, i.e. slide the kernel over the image
- Local image patch is multiplied with the kernel weights and summed up
- Amounts to a local feature extractor



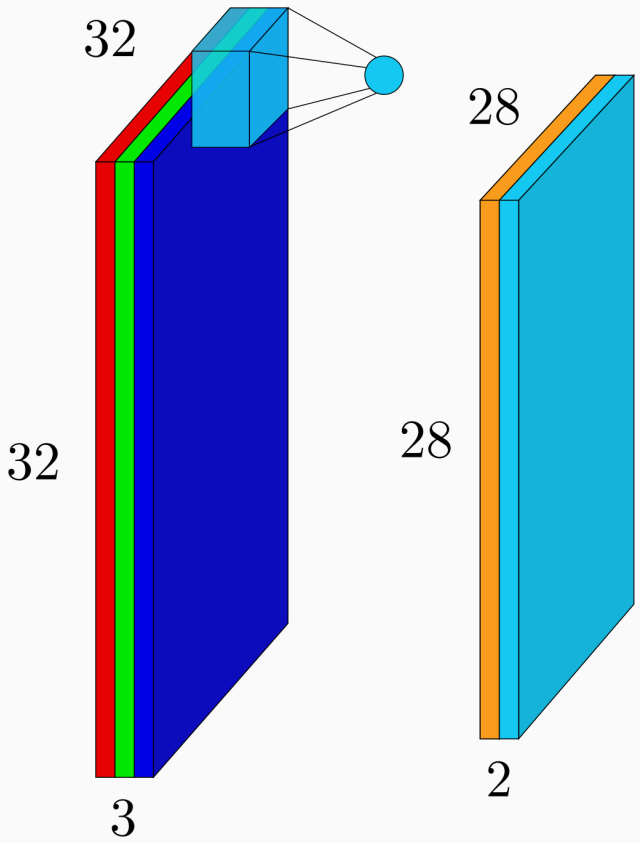
Convolutional Layers

- The convolution extracts a new feature map, here of shape 28×28



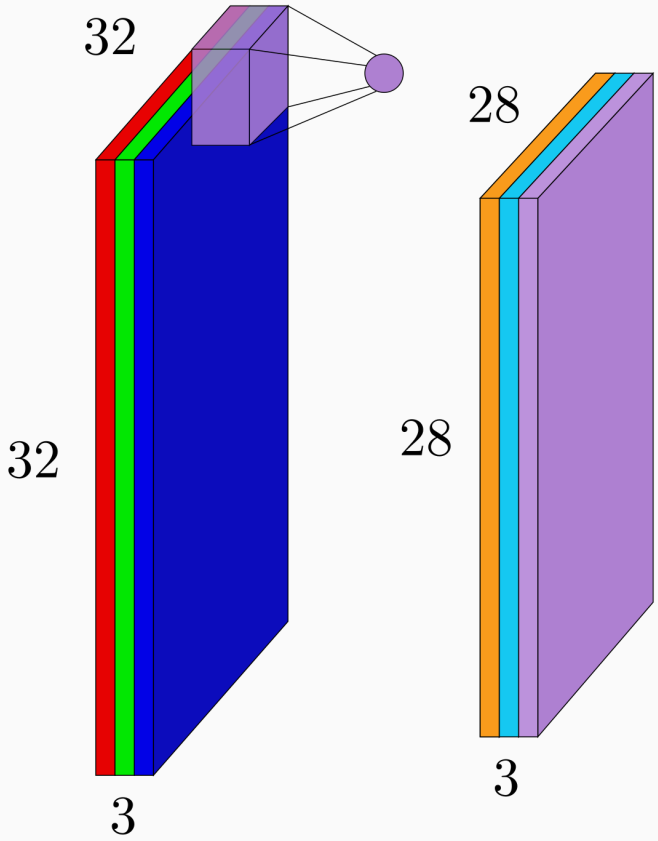
Convolutional Layers

- Repeat with another kernel
- Stacking with the first result yields a feature map with 2 channels



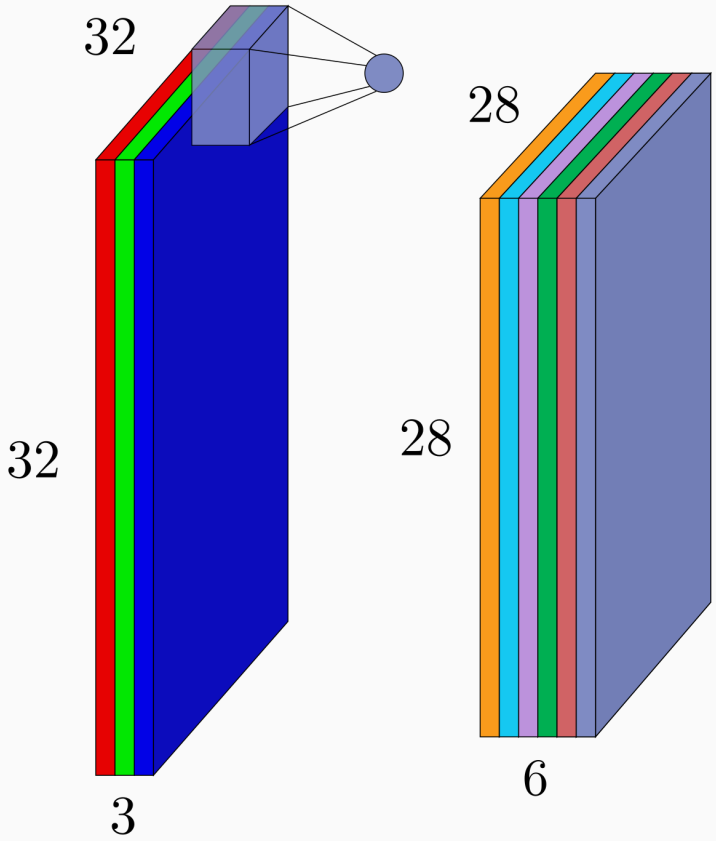
Convolutional Layers

- Another kernel, feature map with 3 channels. . .



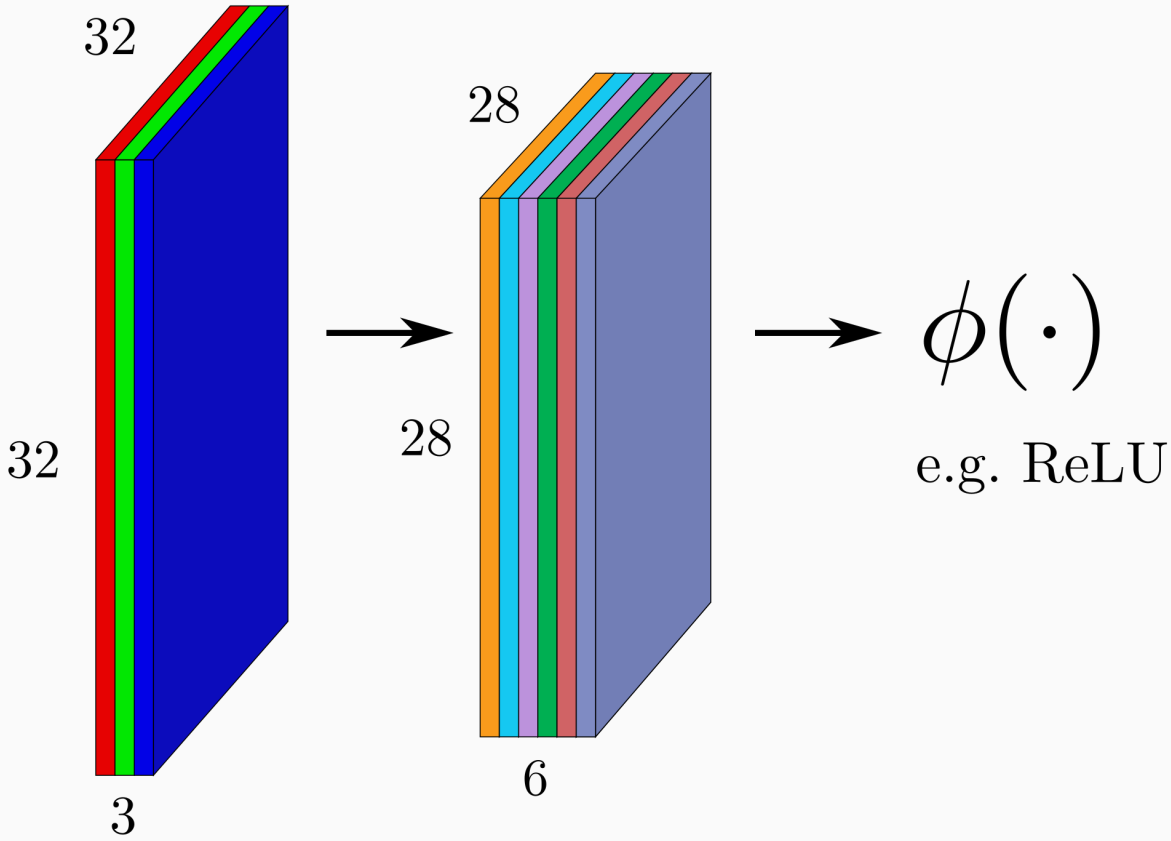
Convolutional Layers

- . . . and so on
- Number of output channels is a hyperparameter



Convolutional Layers

- After convolution an element-wise non-linearity is applied, e.g. ReLU
- After that we can add more convolutional layers or standard linear layers, e.g. to yield an output layer for classification

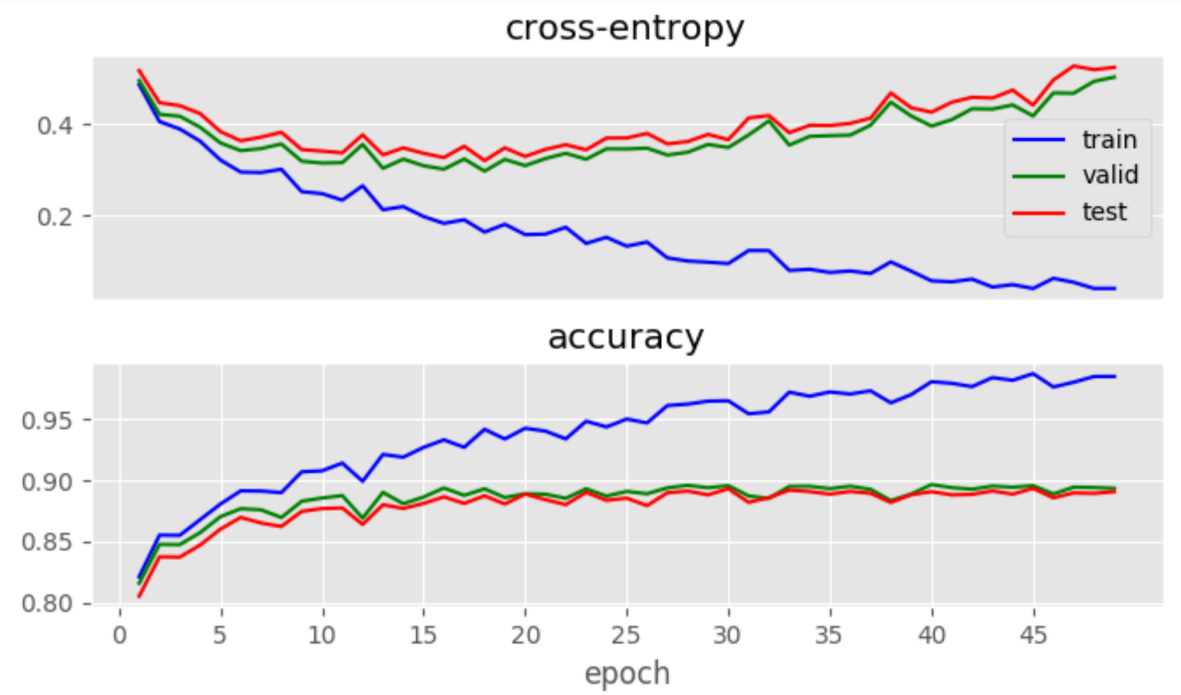


Avoiding Overfitting

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

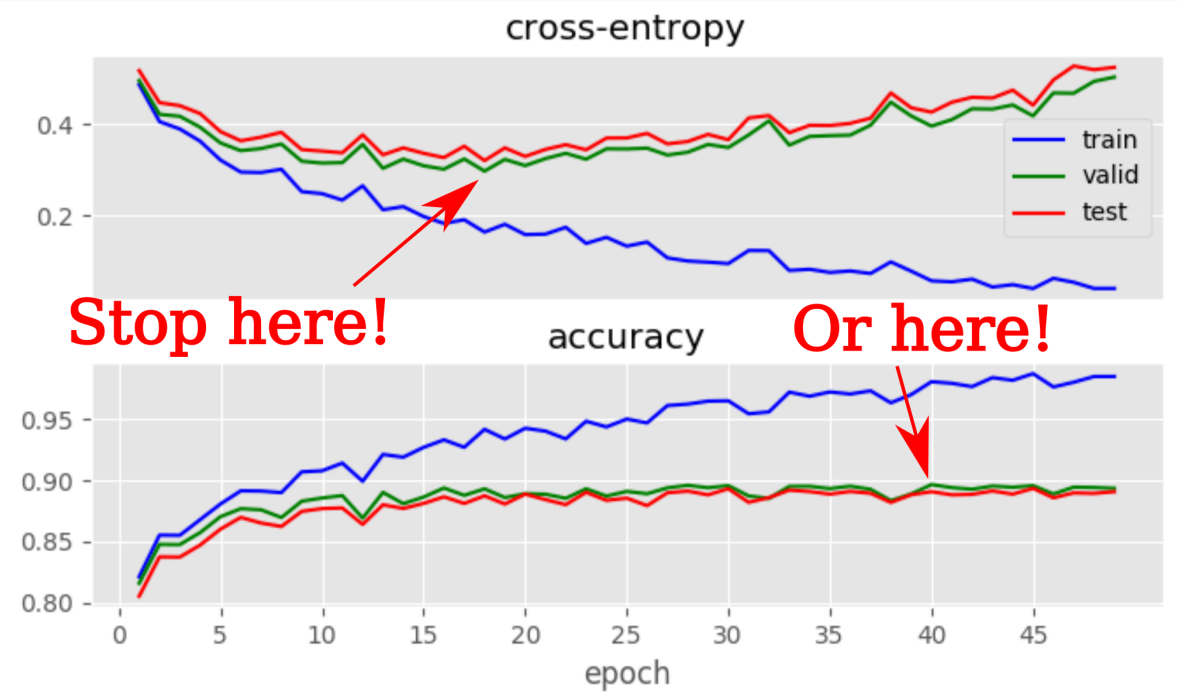
- Early stopping
- Regularization
- Dropout

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 $\mathcal{L}_{val,min}$

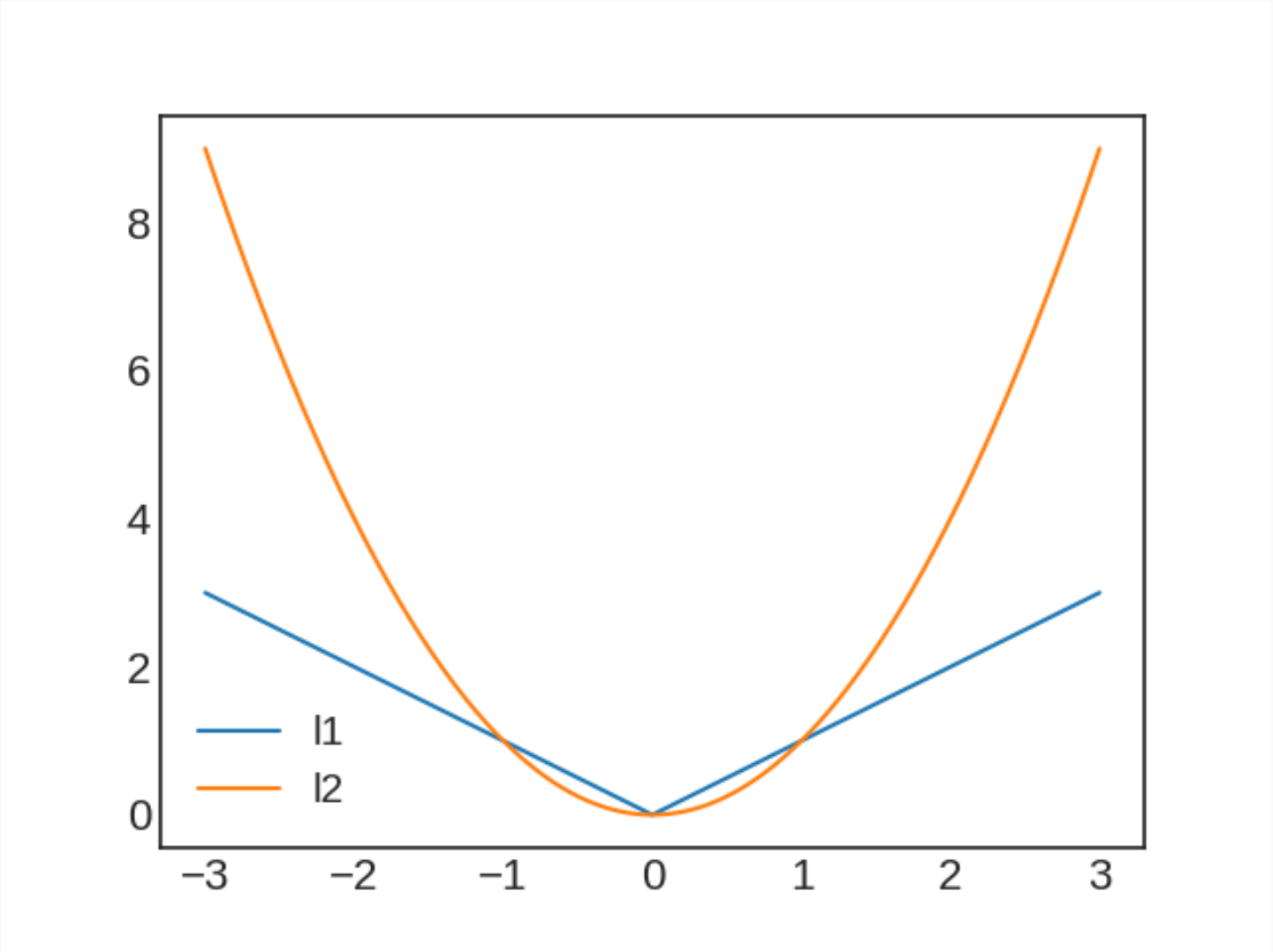
Regularization

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

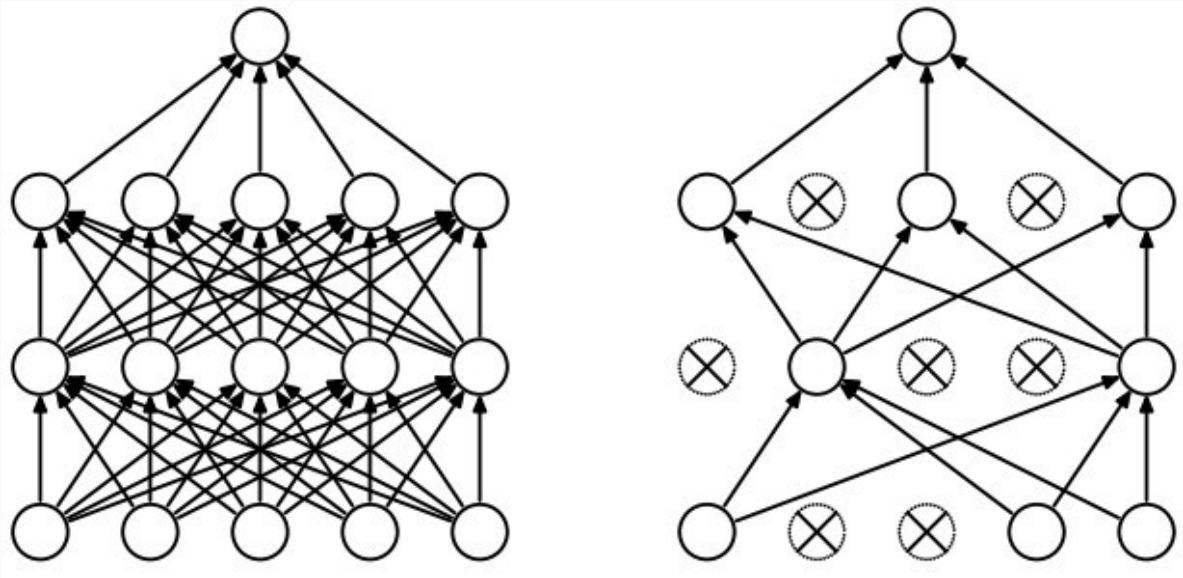
$$\mathcal{L}_{train} = \frac{1}{N}\sum_{i=1}^{N} \ell(y^{(i)}, f(\boldsymbol{x}^{(i)}; \boldsymbol{w})) + \lambda R(\boldsymbol{w})$$

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

L1 vs. L2-Regularization

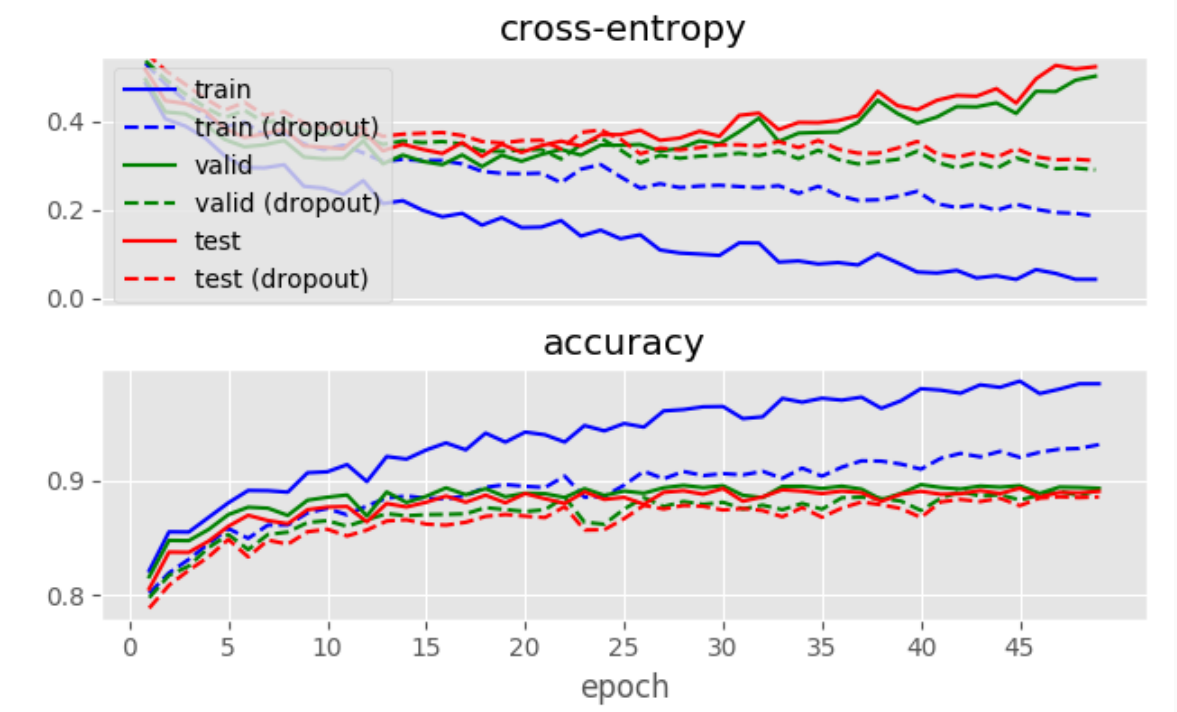


Dropout [Srivastava et al., JMLR 2014]



- 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: just a stack of matrix multiplications and element-wise non-linearities
- Backprop applies the chain rule and evaluates the neural network backwards
- Autodiff simplifies this a lot
- Stochastic gradient descent for quicker updates
- Regularization, early stopping, dropout
- Convolutional neural nets for image data
- See further: "Deep Learning" course with Legenstein, Brglez & Özdenizci