Lecture Recording

- **Note: These lectures will be recorded and posted onto the IMPRS website**

- Dear participants,
- We will record all lectures on *"Making sense of data: introduction to statistics for gravitational wave astronomy"*, including possible Q&A after the presentation, and we will make the recordings publicly available on the IMPRS lecture website at:

  - https://imprs-gw-lectures.aei.mpg.de/2021-making-sense-of-data/

- By participating in this Zoom meeting, you are giving your explicit consent to the recording of the lecture and the publication of the recording on the course website.

Making sense of data: introduction to statistics for gravitational-wave astronomy

Lecture 11: Deep Learning

AEI IMPRS Lecture Course
Stephen Green stephen.green@aei.mpg.de

Machine learning algorithms

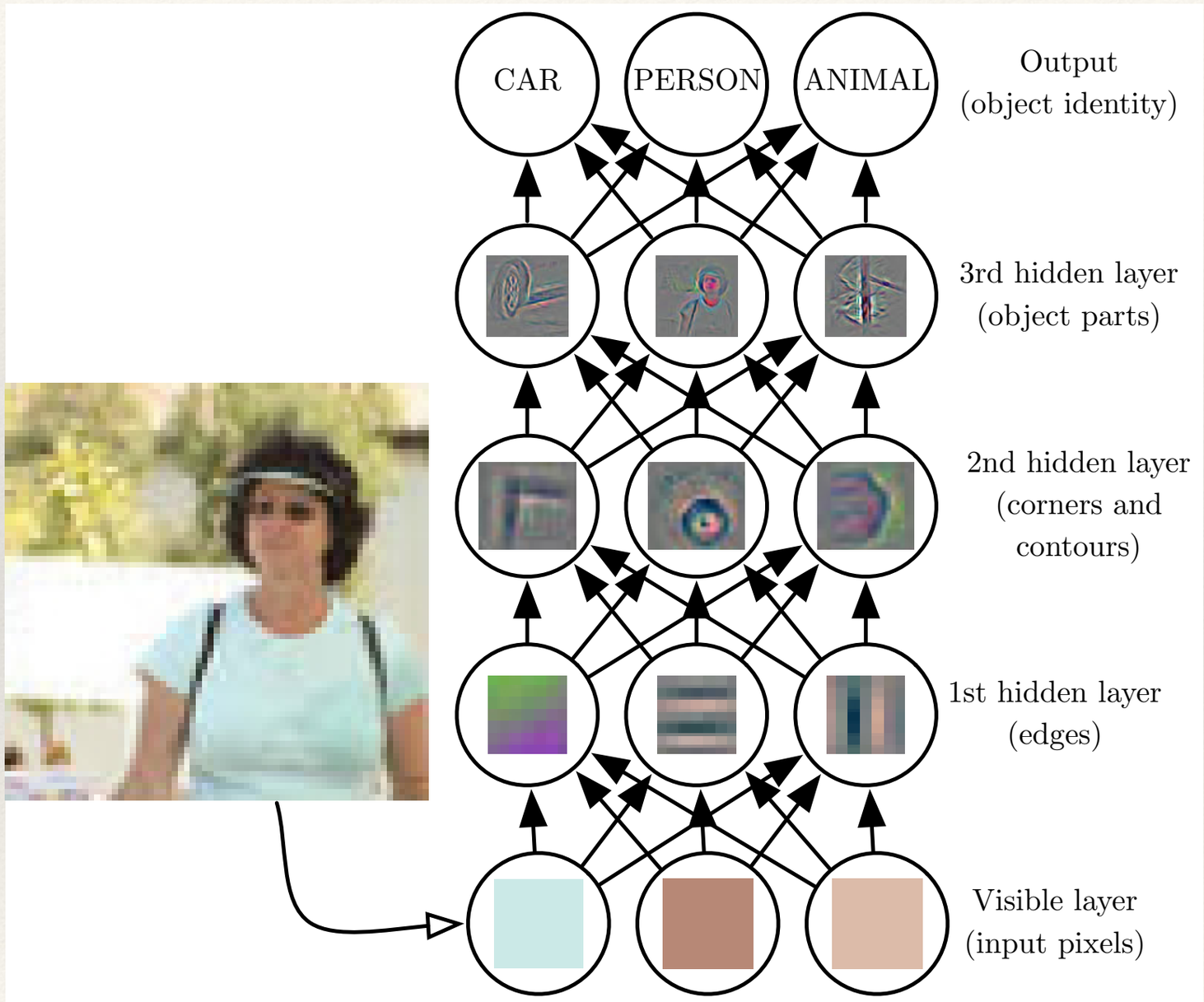
- A machine learning algorithm requires the following:

1. **dataset** — $\{\boldsymbol{x}^{(i)}, y^{(i)}\}$ (supervised) or $\{\boldsymbol{x}^{(i)}\}$ (unsupervised)
2. **model** — E.g., linear regression $p_{\text{model}}(y \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{\theta}^\top \boldsymbol{x}, 1)(y)$
3. **loss function** — E.g., $J(\theta) = -\mathbb{E}_{p_{\text{data}}(\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{x})$
4. **optimization algorithm** — E.g., stochastic gradient descent

Introduction to deep learning

- For many machine learning algorithms, the input $\boldsymbol{x}$ is not the raw data. Rather, one must choose a small number of high-level **features** that are useful for predicting the output.
  - E.g., the color of a car rather than the image of a car.
- This **representation** is often specified by hand, but it can also be learned from lower-level features, or raw data.
- Deep learning seeks to learn higher level representations in terms of lower level ones by composing functions.

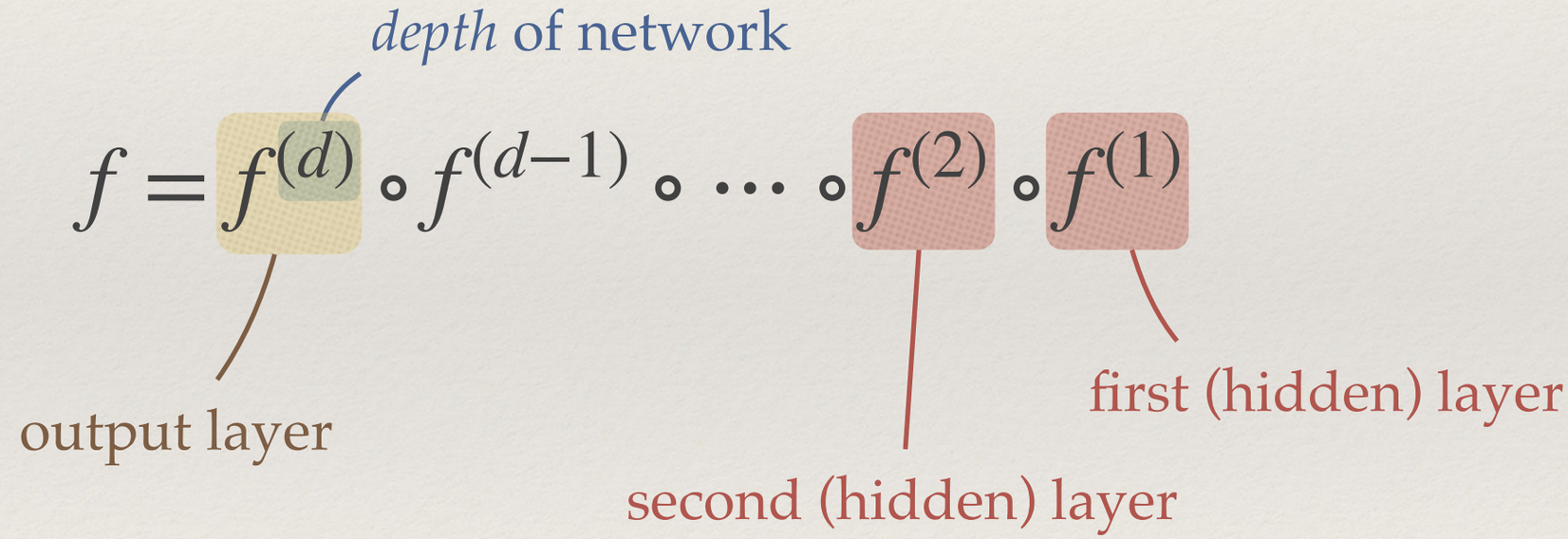
Introduction to deep learning



Goodfellow *et al* (2016)

Feedforward neural networks

- Feedforward neural networks or multilayer perceptrons (MLPs) are the classic deep learning model.
- Defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ as a composition of simpler mappings:



- "Feed-forward" because there is no feedback of later layers on earlier ones.

Feedforward neural networks

$$f = f^{(d)} \circ f^{(d-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}$$

- Each layer is of the form

$$f^{(j)}(\boldsymbol{h}) = \sigma_j\left(\boldsymbol{W}_j^\top \boldsymbol{h} + \boldsymbol{b}_j\right)$$

weight matrix, bias vector

activation function
- often nonlinear

linear mapping $\boldsymbol{z}_j = \boldsymbol{W}_j^\top \boldsymbol{h} + \boldsymbol{b}_j$

- Weights and biases are the parameters defining the model $\boldsymbol{\theta} \equiv \{\boldsymbol{W}_j, \boldsymbol{b}_j\}_{j=1}^d$. These are tuned during training.

Feedforward neural networks

$$f = f^{(d)} \circ f^{(d-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}$$

$$f^{(j)}(\boldsymbol{h}) = \sigma_j\left(\boldsymbol{W}_j^\top \boldsymbol{h} + \boldsymbol{b}_j\right)$$

- The MLP is therefore defined by
  - **depth** (number of layers)
  - **widths** (dimensions of hidden layers)
  - **choice of activation functions**
- Training uses stochastic gradient descent with gradients calculated using **back-propagation** (the chain rule).

Output layer

- The activation function for the output layer is determined by the nature of the output and the distribution we are modeling.
- Example 1: For **regression,** typically take the output to be the **mean** of a Gaussian distribution

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}\left(\boldsymbol{f}(\boldsymbol{x}), \boldsymbol{I}\right)(\boldsymbol{y})$$

- Since the mean is unconstrained, use a **linear** output layer

$$\sigma_{\text{linear}}(z) = z$$

- Maximum likelihood gives the mean squared error loss (as before).

Output layer

- Example 2: For **binary classification,** use a **Bernoulli** distribution.
  - Take a 1-dimensional output, which gives the probability that $y = 1$.
  - Needs to lie between 0 and 1. Use a **logistic sigmoid** activation,

$$\sigma_{\text{sigmoid}}(z) = \frac{1}{1 + e^{-z}}$$

- The probability of y is then given by [check this!]

$$p(y|\boldsymbol{x}) = \sigma_{\text{sigmoid}}\left((2y - 1)z\right)$$

- Finally, the maximum likelihood loss involves the **softplus** function

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\log p(y|\boldsymbol{x}) \\ &= \zeta\left((1 - 2y)z\right) \end{aligned}$$

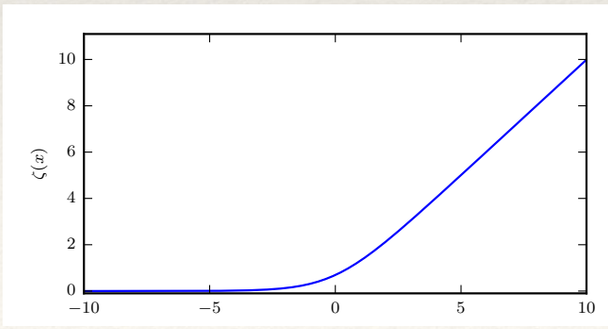


Figure 3.4: The softplus function.

Output layer

- Example 3: For general classification (with possible outputs $y \in \{1, \ldots, k\}$) use a **Multinoulli** distribution.
  - Take a k-dimensional output layer, where each output i is the probability that $y = i$.
  - Use a **softmax** activation function $\sigma_{\text{softmax}}(z)_i = \dfrac{\exp(z_i)}{\sum_j \exp(z_j)}$

    Then $p(y = i \,|\, \boldsymbol{x}) = \sigma_{\text{softmax}}(z)_i$
  - Maximum likelihood loss is $J(\boldsymbol{\theta}) = -z_i + \log \sum_j \exp(z_j)$

Output layer

- **Remarks:**
  - These examples show how specifying a model as a probability distribution and using maximum likelihood estimation automatically yields the correct loss function for training.
  - All loss functions constructed are more-or-less "linear" (not exponential) in $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$ from the previous layer:

MSE loss

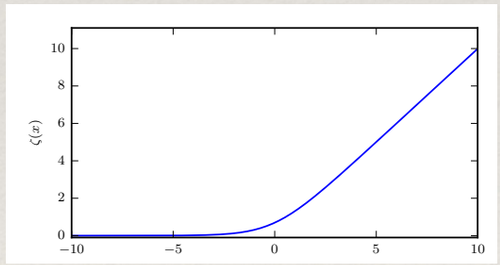


Figure 3.4: The softplus function.

$$J(\boldsymbol{\theta}) = -z_i + \log \sum_j \exp(z_j)$$

This is part of the construction, so that that gradients do not vanish or blow up during training.

Output layer

- We can also construct **more complicated distributions.**
- Example: Gaussian distribution where we also fit for the **covariance matrix**

$$p(\boldsymbol{y}\,|\,\boldsymbol{x}) = \mathcal{N}\left(\boldsymbol{\mu}(\boldsymbol{x}), \boldsymbol{\Sigma}(\boldsymbol{x})\right)$$

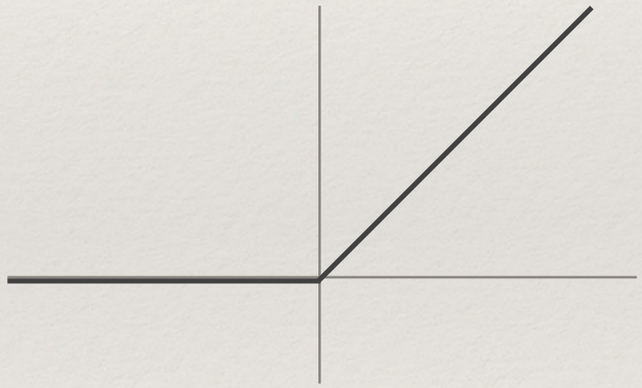
  - For diagonal covariance, better to use the precision matrix, and enforce positivity using, e.g., softplus activation.
- Example: **Mixture density networks**
  - Several Gaussian distributions and a multinouilli distribution
- Example: **Normalizing flow** can model much more complicated distributions.
- In all cases, maximum likelihood loss gives an appropriate loss function.

Hidden layers

$$f = f^{(d)} \circ f^{(d-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}$$

- Activation functions needed to **introduce nonlinearity.**
- Most common activation: **Rectified Linear Unit (ReLU)**

$$\sigma_i(z) = \begin{cases} z_i, & z_i > 0 \\ 0, & z_i = 0 \end{cases}$$



  - Linear part good for gradients. Runs the risk of not activating for some inputs.
- Other possibilities: leaky ReLU, ELU, logistic sigmoid, tanh, sin, …

Hidden layers

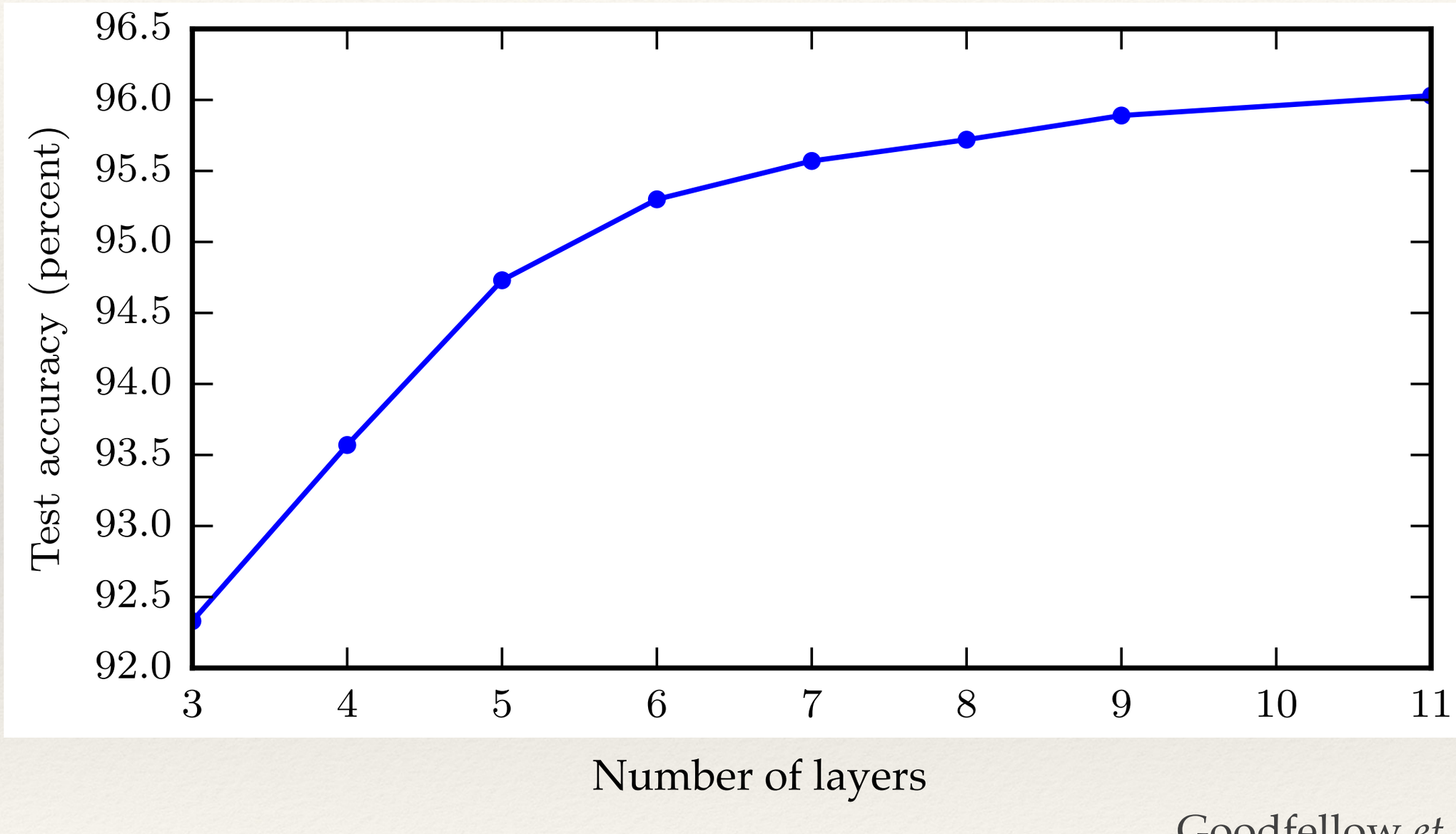
- In addition to activations, it is necessary to choose hyperparameters for **depth and width** of network. Deeper and wider will give more representational capacity, although it may be harder to train. Choice comes from experimentation.
- **Universal approximation theorem:**

> A feedforward network with a linear output layer and at least one hidden layer can—given a wide enough hidden layer—approximate any reasonable function to arbitrary accuracy.

  - May not learn or generalize very well, and may require a huge hidden layer.

- Deeper networks will usually require fewer total parameters than one very wide network.

Effect of depth



Goodfellow *et al* (2016)

Back-propagation

- To train the network using some form of gradient descent, it is necessary to be able to **efficiently compute gradients** with respect to all of the network parameters (weights and biases).

- This is accomplished using a form of automatic differentiation call **back-propagation.**

- Relies on **compositional nature** of neural networks

$$f = f^{(d)} \circ f^{(d-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}$$

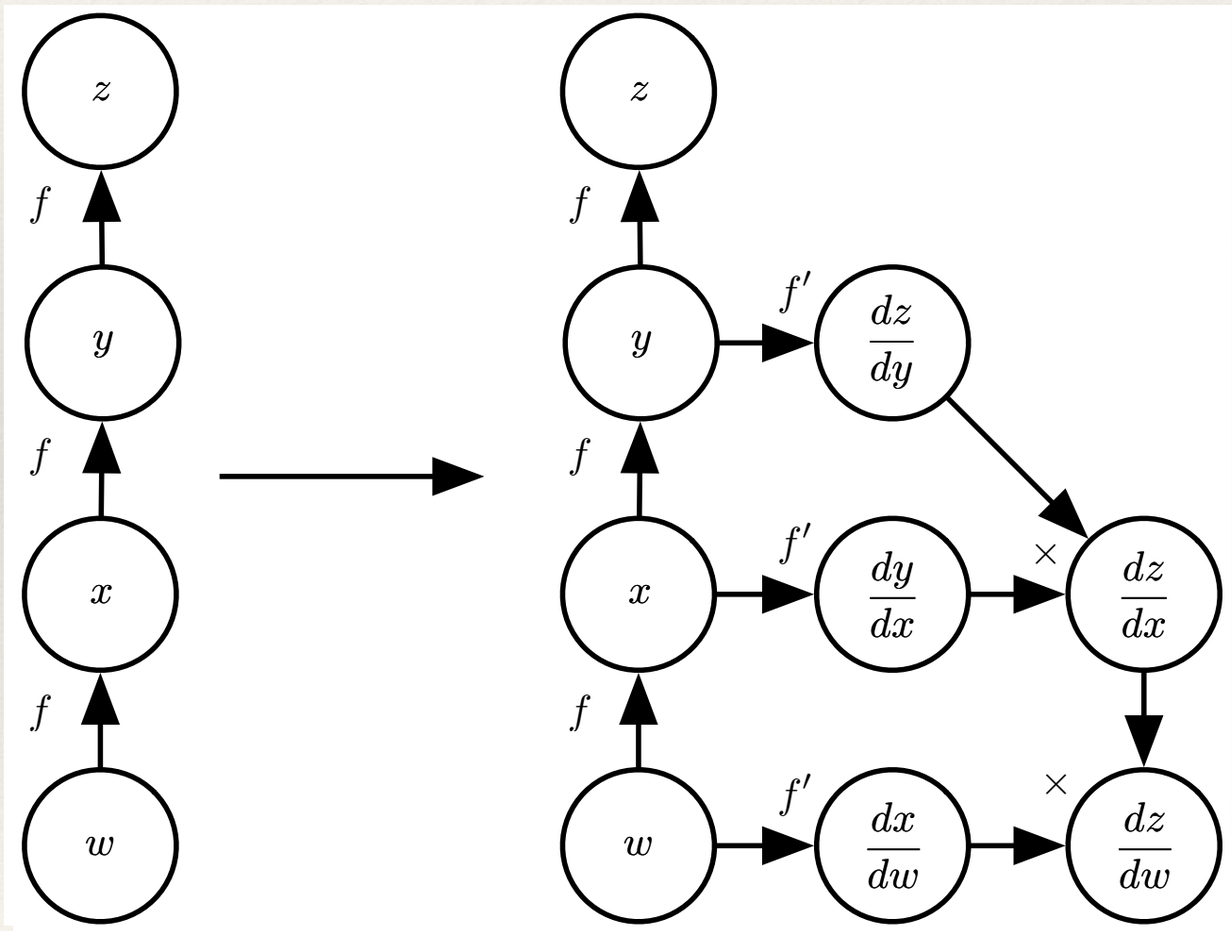
plus the **chain rule of calculus** and differentiability of all operations.

Back-propagation

- It is important to organize the calculation in an **efficient** way, and not carry out the same calculation multiple times.

$$\begin{aligned}\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\ &= f'(y)f'(x)f'(z) \\ &= f'(f(f(w)))f'(f(w))f'(w)\end{aligned}$$

Efficient implementations in every deep-learning framework (PyTorch, TensorFlow, JAX, …)



Goodfellow et al (2016)

Training

- Stochastic gradient descent is well suited to training on GPU hardware.
- **Challenges:**
  - Local minima, plateaus, saddle points in the loss landscape
  - Exploding gradients
  - Very deep networks
- Various approaches to deal with these, including other architectural choices, choice of initial parameter values, and improvements upon stochastic gradient descent.
- Improved optimizers include **momentum** — a moving average of the gradient.
- Best bet is to use the **Adam** optimizer.

Architectural choices

- So far we have described the most basic of neural network architectures, the fully-connected feed-forward network.
- **Possible modifications:**
  - Sparse connectivity (sparse weight matrices)
  - Shared weights
  - Connections between non-adjacent layers
  - Recursive or recurrent connections
- Usage will depend on characteristics of the data, amount of computer resources,

Convolutional networks

- Convolutional networks are well suited to data that has **translation invariance**
  - 2d images, 1d audio, 3d medical scans, …
- Use local convolutions instead of general matrix multiplication.

$$(K * I)_{ij} = \sum_{mn} K_{mn} I_{i-m,j-n}$$

output kernel input

  - Weight matrix uses **parameter sharing** and **sparsity** (kernel is local)
  - Typically also add an additional dimension: **channel**
- Can also handle inputs of variable size

Convolutional networks

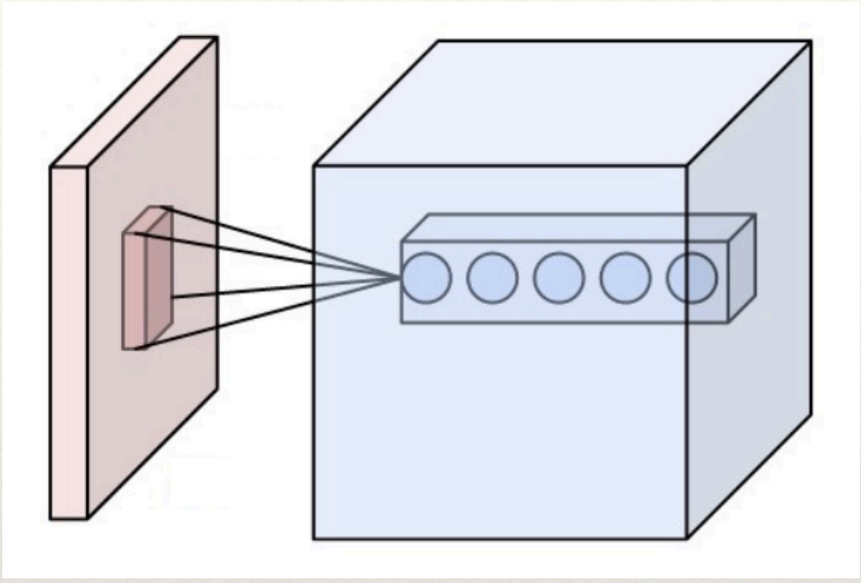
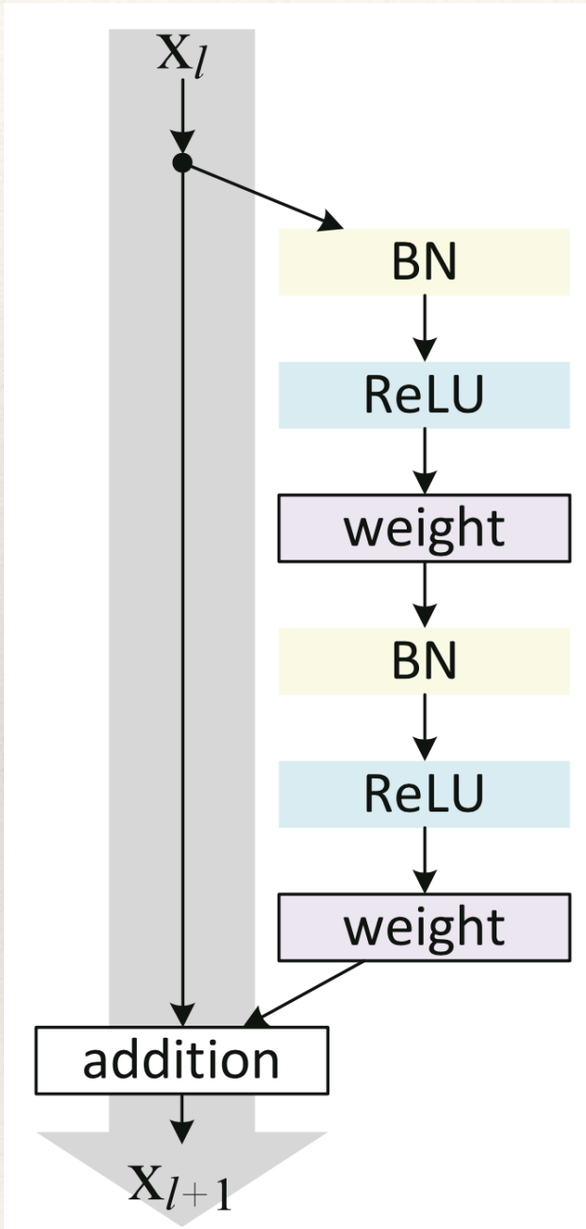


Image: Aphex34

- Network constructed by combining various types of layers
  - Layers for convolving in 1D, 2D, … (choices for stride, kernel size, number of channels, padding, …)
  - Layers for changing dimension (pooling, reshape)
  - Convolutional arithmetic needed to fit layers together

Residual networks

- Beyond a small number of layers, training deep neural networks becomes very difficult.
- One very successful architecture is the **residual network** (He *et al*, 2016)
  - Preserves an **identity mapping** all the way through the network.
  - Learns the **residual** $\mathbf{x}_{l+1} = \mathbf{x}_l + \mathcal{F}(\mathbf{x}_l, \mathcal{W}_l)$
  - Enables successful training with over **1000 layers**.



Summary

- **Things we discussed:**
  - Feedforward networks:
    - Output layers, hidden layers, architecture considerations
    - Loss functions
    - Back-propagation
    - Convolutional and residual networks
- Many other topics were not discussed, e.g., recurrent architectures
- **Next lecture:** GW applications