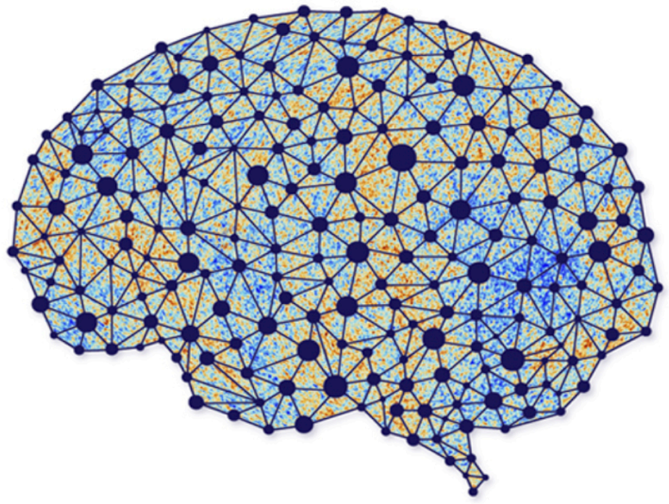
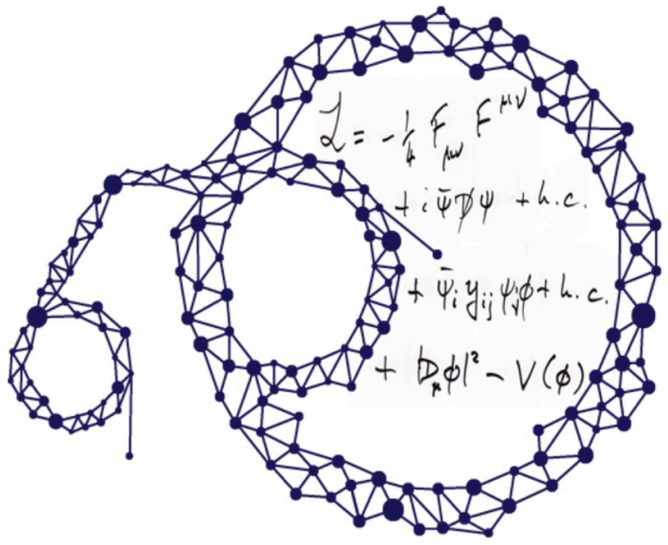
Physics 361 - Machine Learning in Physics

Lecture 6 – Basics of Machine Learning

Feb. 6th 2025



AI
∩
Universe



Moritz Münchmeyer (with Slides from Gary Shiu)

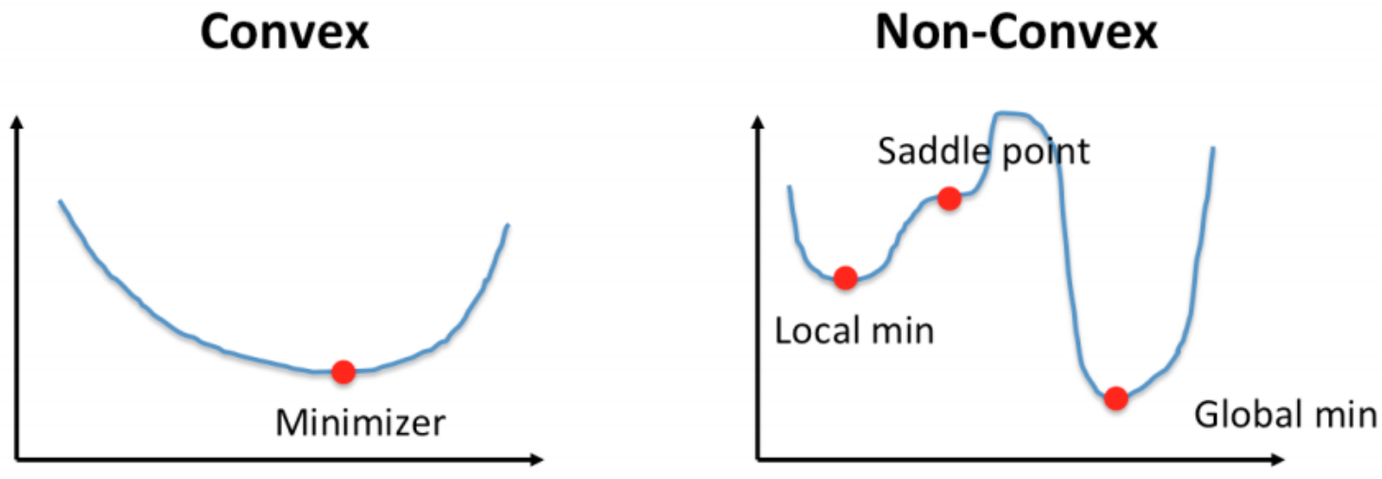
Unit 2: Machine Learning Basics

2.5 Optimization

Resources:
- https://arxiv.org/pdf/1803.08823 A high-bias, low-variance introduction to Machine Learning for physicists
- deeplearningbook.org chapter 8

Optimizers

- ML problems are mostly about minimizing a cost function. This can be a hard problem because:
  - The function depends on many parameters, say $\mathcal{O}(10^6)$ and hence the minimization is over a huge parameter space.
  - It becomes numerically expensive to evaluate the cost function, its gradient and higher derivatives.
  - Non-convex loss function → multiple minima



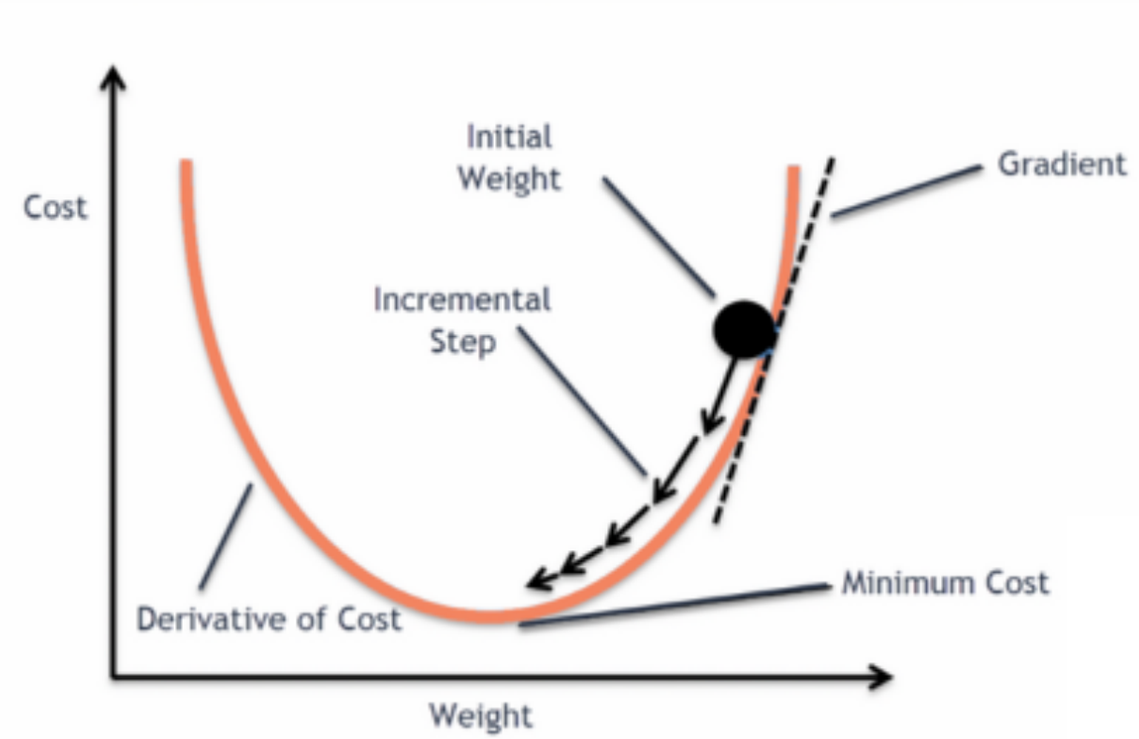
- Common method: **gradient descent** & variations.

Gradient Descent

- The “energy” we want to minimize is the cost function (loss function):

$$E(\boldsymbol{\theta}) = \sum_{i=1}^{n} e_i(\mathbf{x}_i, \boldsymbol{\theta}).$$

can often be written as a sum over data points, e.g., mean-square error or cross-entropy (classification).



- **Idea:** adjust parameters in the direction where the gradient of $E(\theta)$ is large and negative. Gradually shifting towards a local minimum.

learning rate

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t),$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t$$

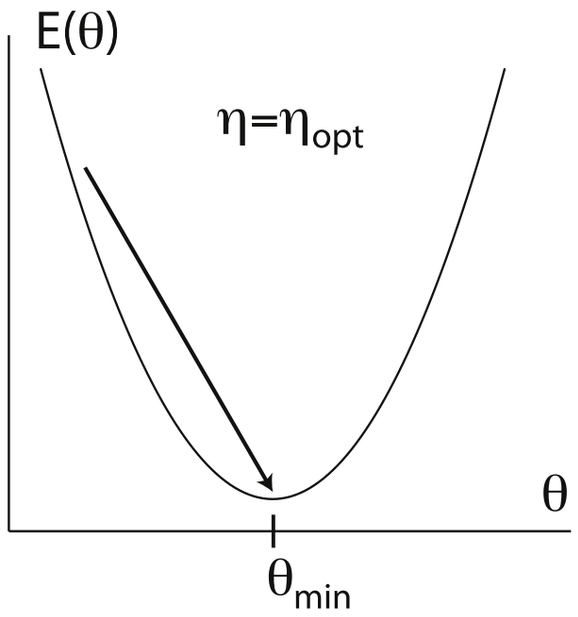
Newton's Method

- Inspiration for many widely used optimization methods.

- Choose the step $\mathbf{v}$ for the parameter θ to minimize a 2nd order Taylor expansion:

$$E(\boldsymbol{\theta}+\mathbf{v}) \approx E(\boldsymbol{\theta}) + \nabla_\theta E(\boldsymbol{\theta})\mathbf{v} + \frac{1}{2}\mathbf{v}^T H(\boldsymbol{\theta})\mathbf{v},$$

where $H(\theta)$ is the Hessian. Differentiate w.r.t. $\mathbf{v}$, noting that for the optimal value $\mathbf{v}_{opt}$, $\nabla_{\mathbf{v}} E(\theta+\mathbf{v})\big|_{\mathbf{v}=\mathbf{v}_{opt}} = 0$:



$$\Rightarrow \quad 0 = \nabla_\theta E(\boldsymbol{\theta}) + H(\theta)\mathbf{v}_{\text{opt}}$$

$$\Rightarrow \quad \mathbf{v}_t = H^{-1}(\boldsymbol{\theta}_t)\nabla_\theta E(\boldsymbol{\theta}_t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t.$$

Gradient Descent vs Newton's Method

- Newton's method requires knowledge of 2nd derivatives (n^2 component Hessian) which is computationally expensive.

- Calculating inverse of the Hessian is expensive especially for millions of parameters (common in neural network applications).

  ⇒ Newton's method unfeasible for typical ML systems.

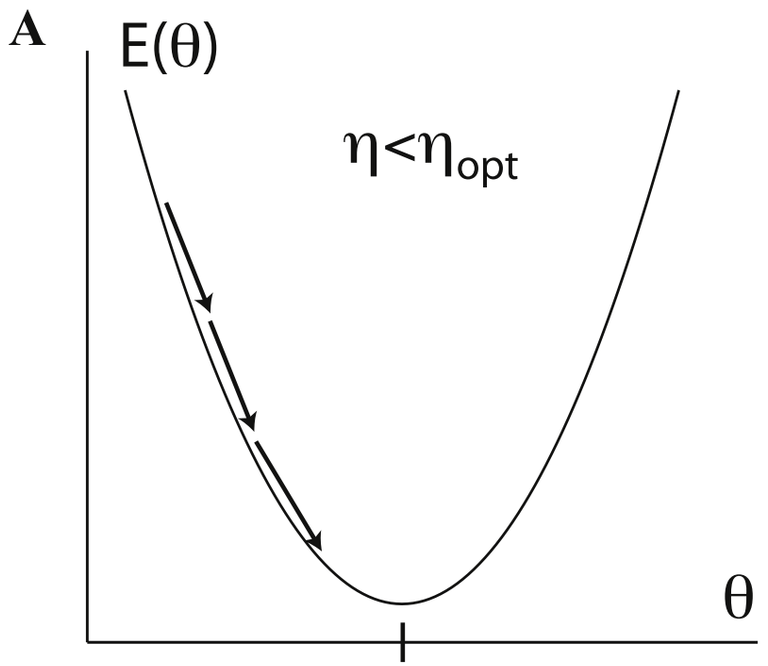
- However, useful to get intuition how to choose the **learning rate:**

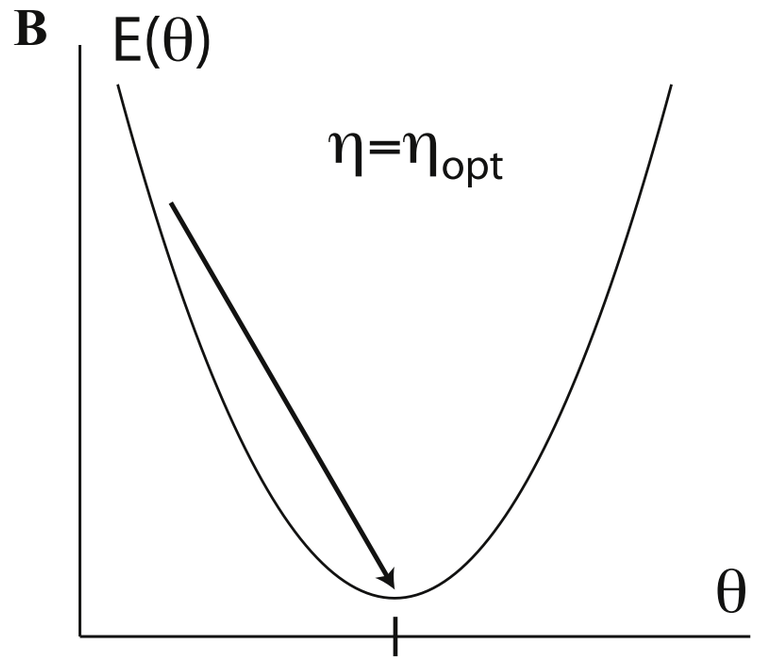
$$\eta_{\text{opt}} = [\partial_\theta^2 E(\theta)]^{-1} \qquad \textbf{(1-dim)}$$

- Newton's method automatically **adjusts the learning rate**: takes larger steps in flat directions and smaller steps in steep directions.

Regimes of Learning Rate (for a quadratic loss)

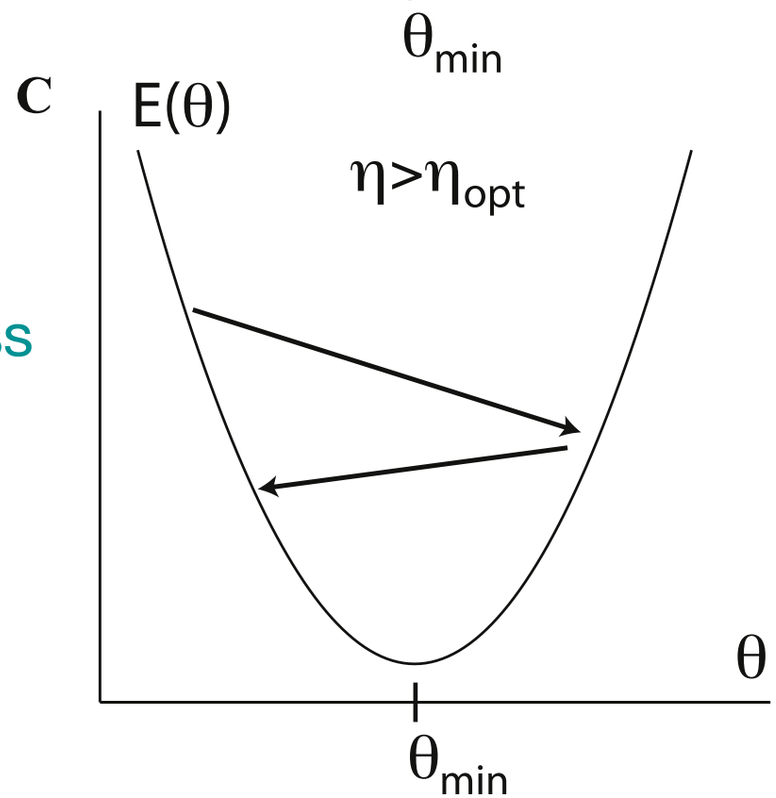
multiple small steps

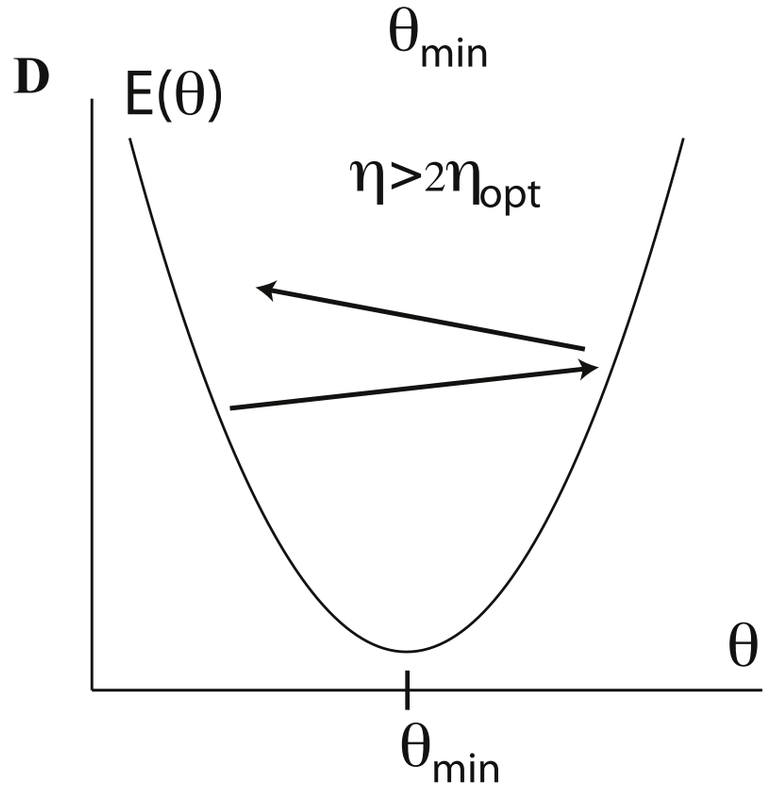




single step

oscillate across both sides





diverges

LeCun et al, 1998

Convergence in Higher Dimensions

- Natural generalization of $\partial_\theta^2 E(\theta)$ is the Hessian.

- Perform a singular value decomposition of the Hessian matrix:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^{\mathrm{T}},$$

  where U and V are orthogonal matrices and D is diagonal with eigenvalues $\{\lambda_{min}, \ldots, \lambda_{max}\}$.

- Convergence of gradient descent requires:

$$\eta < \frac{2}{\lambda_{\max}},$$

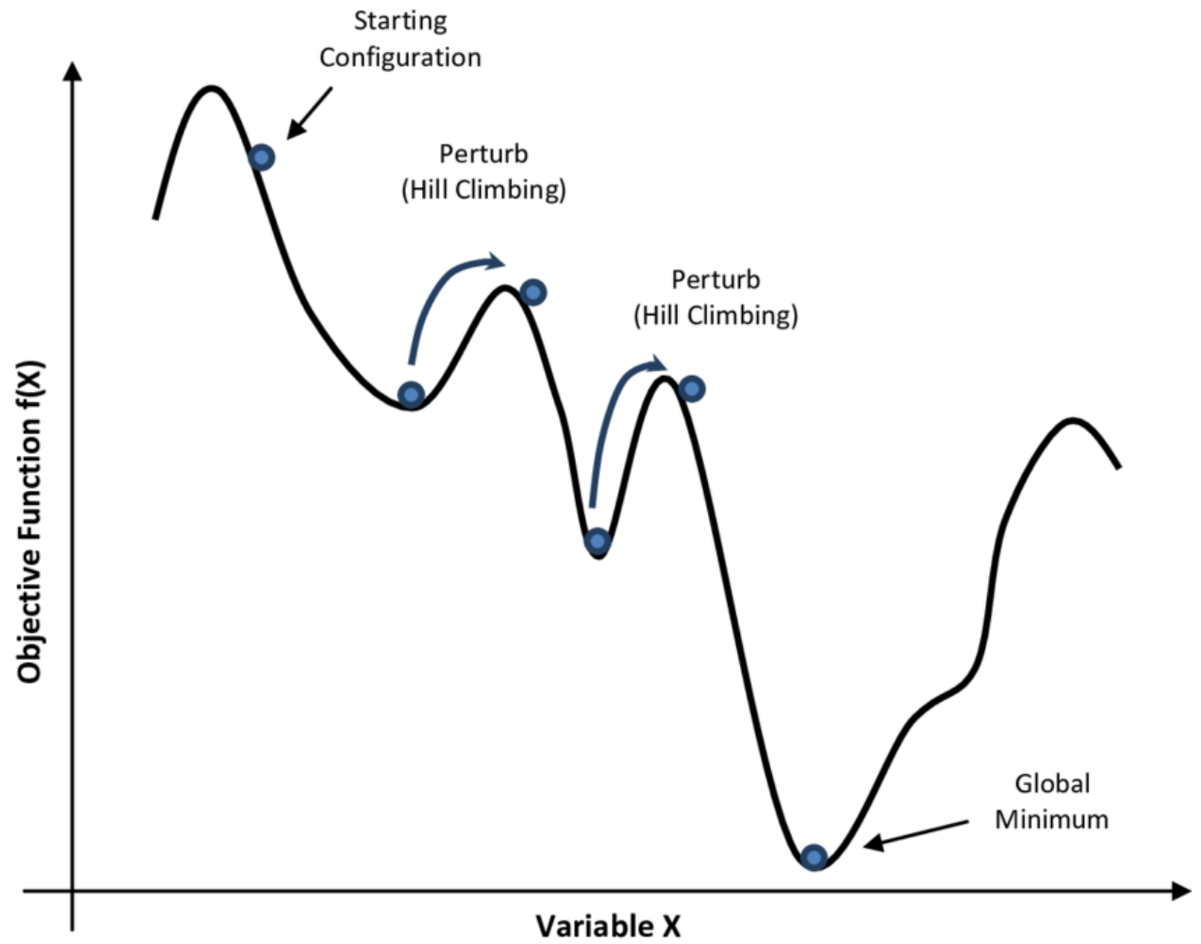
- If $\lambda_{min} \ll \lambda_{max}$, convergence is slow in the λ_{min} direction. Convergence time scale scales with $\kappa = \lambda_{max}/\lambda_{min}$.

Gradient Descent — Limitations

- Finds **local minima**: Needs a "temperature" (stochasticity) to tunnel over energy barriers.
- **Sensitive to initial conditions** (which local minimum depends on starting point)

  → important to consider sensible **initialization** of training process.
- Gradients computationally expensive for large datasets

  → calculate gradient using small subset of data: **"mini-batches"** (gives stochasticity)



Stochastic Gradient Descent (SGD)

Gradient Descent — Limitations

- **Sensitive to choice of learning rates** (too small would take a long time to train, too large would diverge from minima).

  → Furthermore need to adaptively choose learning rate.

- **Treats all directions uniformly. In steep directions a large learning rate can cause instability, while in flat directions a small learning rate is inefficient.**

  → We would like to take larger steps in flat directions, smaller steps in steep directions

  → second derivatives needed to account for “curvature effects”.

- Takes exponential amount of time to **escape a saddle point**.

You are encouraged to experiment with gradient descent and its variants using the Juypter notebook on:

https://physics.bu.edu/%7Epankajm/MLnotebooks.html

SGD with Mini-batches

- **Stochasticity** by approximating gradient on subset of data, so-called **mini-batches**, denoted as B_k (size varies ~10-100):

$$D \rightarrow B_1, B_2, \ldots, B_n$$

- **Speed up gradient computation:**

$$\nabla_\theta E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \nabla_\theta e_i(\mathbf{x}_i, \boldsymbol{\theta}) \longrightarrow \sum_{i \in B_k} \nabla_\theta e_i(\mathbf{x}_i, \boldsymbol{\theta})$$

- Perform gradient descent:

$$\mathbf{v}_t = \eta_t \nabla_\theta E^{MB}(\boldsymbol{\theta}),$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t.$$

- Cycle through mini-batches. One entire cycle is known as an **epoch.**

- Bonus: works effectively as a natural regularizer that **prevents overfitting** in deep, isolated minima **(Bishop 1995)**.

GD with Momentum (GDM)

- **Idea:** add memory of the direction we move in parameter space

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t,$$

by introducing a **momentum parameter** γ, with $0 \leq \gamma \leq 1$

- The step taken $\mathbf{v}$ is a running average of recently encountered gradients with the characteristic time scale for the memory set by γ.

- To get some physics intuition, consider a massive particle in viscous medium with viscous damping coefficient μ, and potential $E(\theta)$:

$$m\frac{d^2\mathbf{w}}{dt^2} + \mu\frac{d\mathbf{w}}{dt} = -\nabla_w E(\mathbf{w}).$$

GD with Momentum (GDM)

- Discrete version of this EOM:

$$m\frac{\mathbf{w}_{t+\Delta t}-2\mathbf{w}_t+\mathbf{w}_{t-\Delta t}}{(\Delta t)^2}+\mu\frac{\mathbf{w}_{t+\Delta t}-\mathbf{w}_t}{\Delta t}=-\nabla_w E(\mathbf{w}).$$

- Bringing it to a form of a GDM:

$$\Delta\mathbf{w}_{t+\Delta t}=-\frac{(\Delta t)^2}{m+\mu\Delta t}\nabla_w E(\mathbf{w})+\frac{m}{m+\mu\Delta t}\Delta\mathbf{w}_t.$$

- The momentum parameter and the learning rate are then identified:

$$\gamma=\frac{m}{m+\mu\Delta t},\qquad \eta=\frac{(\Delta t)^2}{m+\mu\Delta t}.$$

GD with Momentum (GDM)

- **Gain speed** in directions with persistent but small gradient, while **suppressing oscillations** in high curvature directions.

- Especially useful when $E(\theta)$ is shallow and flat in some directions, and narrow and steep in others.

- More useful during the **transient phase** than the fine-tuning phase.

- Slight modification: Nesterov accelerated gradient (NAG) descent (update at expected value of parameters with current momentum):

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t + \gamma \mathbf{v}_{t-1}) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_t . \end{aligned}$$

improves stability and prevents excessive overshooting.

Using the 2nd Moment

- Ideally calculate/approximate Hessian and normalize learning rates accordingly.

- In addition to keeping a running average of the first moment of the gradient (momentum), we also keep track of the **second moment**:

$$\mathbf{S}_t = \mathbb{E}[\mathbf{g}_t^2]$$

- Methods include: AdaGrad (2011), AdaDelta (2012), **RMS-Prop (2012), ADAM (2014).**

- **RMS-Prop** update rules:

$$\begin{aligned}
\mathbf{g}_t &= \nabla_\theta E(\boldsymbol{\theta}) \\
\mathbf{s}_t &= \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2 \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}},
\end{aligned}$$

RMS-Prop

- RMS-Prop update rules:

$\beta \approx 0.9$ controls the averaging time of the 2nd moment

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta})$$
$$\mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}},$$

$\epsilon \approx 10^{-8}$ regularizes divergences

- Learning rate is reduced in directions where the norm of the gradient is consistently large.

- Speeds up convergence by allowing us to use a larger learning rate for flat directions.

ADAM optimizer

- Using a running average of both the 1st and 2nd moments:

$$
\begin{aligned}
\mathbf{g}_t &= \nabla_\theta E(\boldsymbol{\theta}) \\
\mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \qquad\qquad \mathbf{m}_t = \mathbb{E}[\mathbf{g}_t] \\
\mathbf{s}_t &= \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \\
\hat{\mathbf{m}}_t &= \frac{\mathbf{m}_t}{1-(\beta_1)^t} \\
\hat{\mathbf{s}}_t &= \frac{\mathbf{s}_t}{1-(\beta_2)^t} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon},
\end{aligned}
$$

- Memory lifetimes of the 1st and 2nd moment are typically:

$$\beta_1 = 0.9,\ \beta_2 = 0.99$$

- Adam works very well for a wide domain of problems.

Which optimizer to use?

- There is no absolute superior optimizer; one should experiment which optimizer and which hyperparameters are suitable for the problem at hand.

- Standard tools: mini-batches, momentum, randomize your batches, transform input to get uniform loss landscape

- Often people use Adam by default and only experiment with the learning rate.

Example of recent related work in physics: Energy-Conserving Optimizer

Improving Energy Conserving Descent for Machine Learning: Theory and Practice #1

G. Bruno De Luca, Alice Gatti, Eva Silverstein (Jun 1, 2023)

e-Print: 2306.00352 [cs.LG]

pdf cite claim reference search 2 citations

Microcanonical Hamiltonian Monte Carlo #2

Jakob Robnik, G. Bruno De Luca, Eva Silverstein, Uroš Seljak (Dec 16, 2022)

e-Print: 2212.08549 [stat.CO]

pdf cite claim reference search 5 citations

Born-Infeld (BI) for AI: Energy-Conserving Descent (ECD) for Optimization #3

G. Bruno De Luca (Stanford U., ITP), Eva Silverstein (Stanford U., ITP) (Jan 26, 2022)

Published in: *PMLR* 162 (2022) 4918 • e-Print: 2201.11137 [cs.LG]

pdf cite claim reference search 5 citations

Unit 2: Machine Learning Basics

2.6 Understanding Shallow and Deep Neural Networks

Resources:

- Based on Simon Prince “Understanding Deep Learning”. Free online: https://udlbook.github.io/udlbook/
- Figures from that book unless otherwise stated

More details on MLPs

- In previous lectures we have proposed stacks of linear layers and activation functions (called **Multilayer Perceptron or MLP**) and used them on SUSY data. MLPs are also sometimes called **Fully Connected Neural Networks**.

- In the following we will understand these functions in some more detail.

- The following section does not introduce any new neural networks technology, but helps interpret the one we have been using.

- We will first understand "**shallow neural networks**", which have one activation function and two linear transformations. Then we will extrapolate to **deep neural networks** which stack many more layers.

Learning a function

- A NN is a parametrization of “big” (multivariate, non-linear) functions.
- Shallow NNs parametrize **piecewise linear functions** and are already expressive enough to approximate arbitrarily complex relationships between multi-dimensional inputs and outputs.

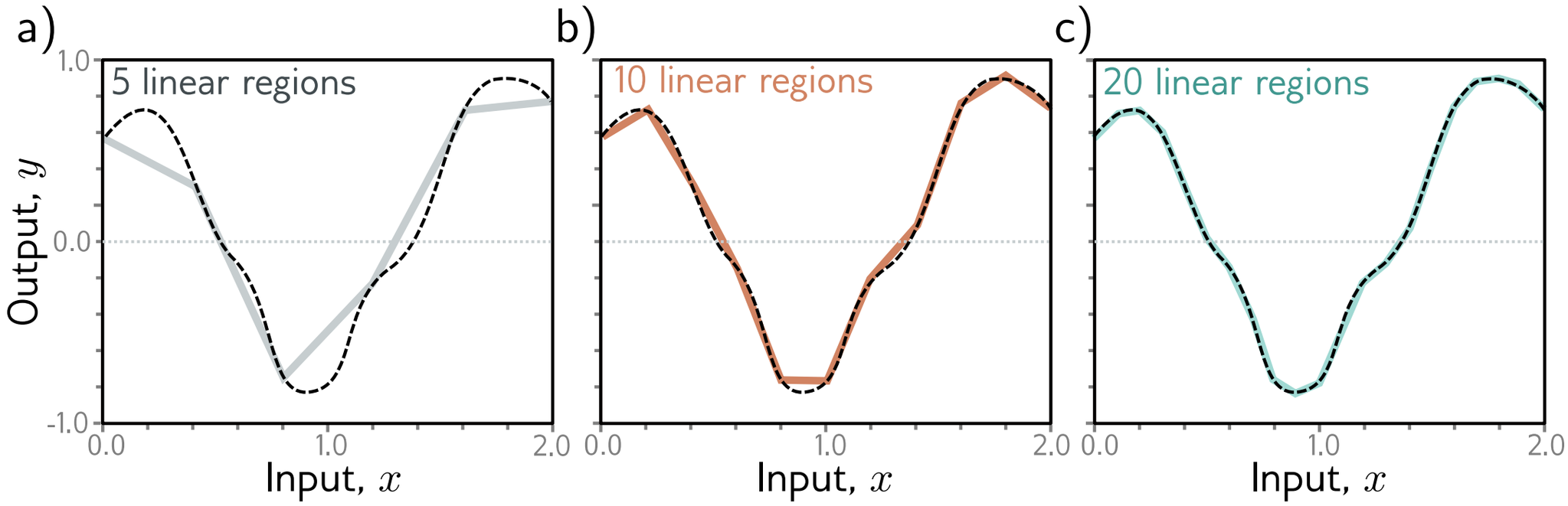


Figure from Simon Prince “Understanding Deep Learning"

Shallow Neural Networks

- Shallow neural networks are functions $\mathbf{y} = f(\mathbf{x}, \boldsymbol{\phi})$ with parameters $\boldsymbol{\phi}$ that map multivariate inputs $\mathbf{x}$ to multivariate outputs $\mathbf{y}$.

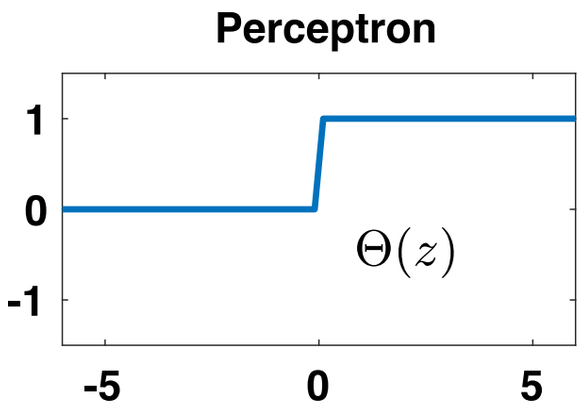
- As a warmup, consider $f(x, \boldsymbol{\phi})$ that maps a scalar input x to a scalar output y and has ten parameters $\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$:

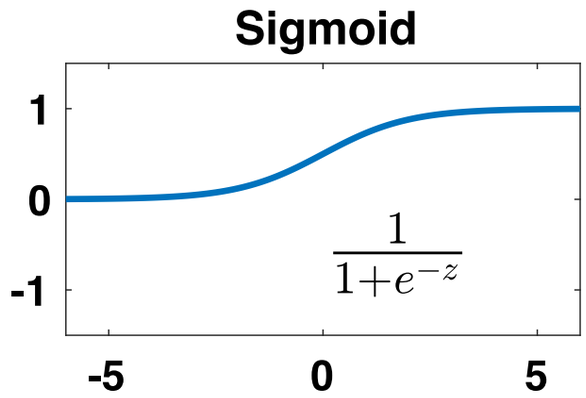
$$\begin{aligned} y &= \mathrm{f}[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x]. \end{aligned}$$

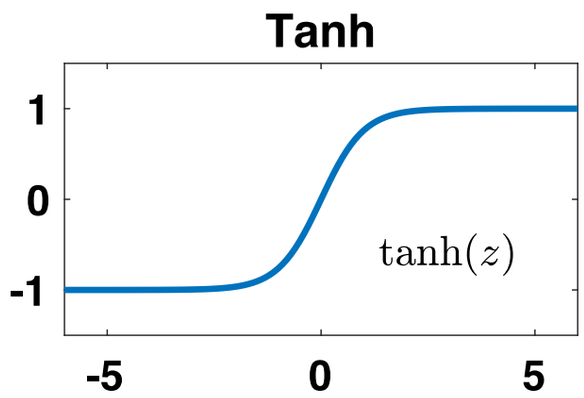
- $a[\,\cdot\,]$ is known as the **activation function**. It cannot be a linear function in order for the NN to go beyond linear regression.

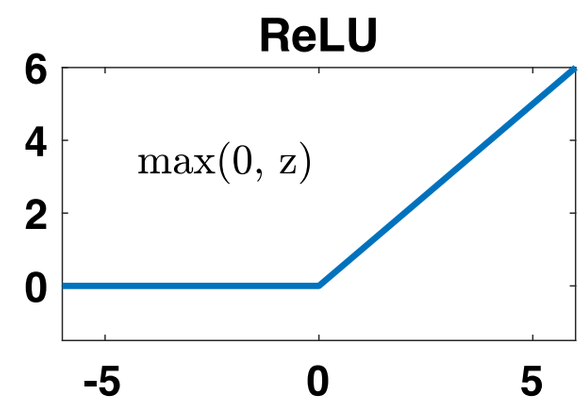
- Given a training dataset $\{x_i, y_i\}_{i=1}^{I}$, we can define a least squares loss function $L[\boldsymbol{\phi}]$ to measure how effectively the model describes this dataset. To train the model, we find $\hat{\boldsymbol{\phi}}$ that minimizes $L[\boldsymbol{\phi}]$.

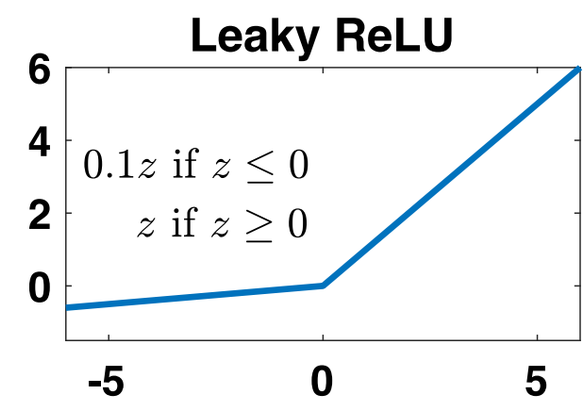
Activation Functions

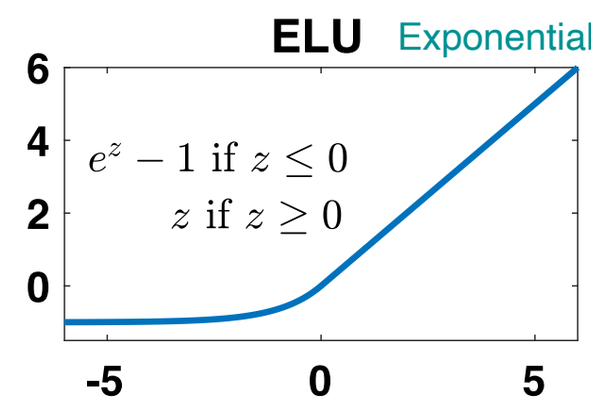












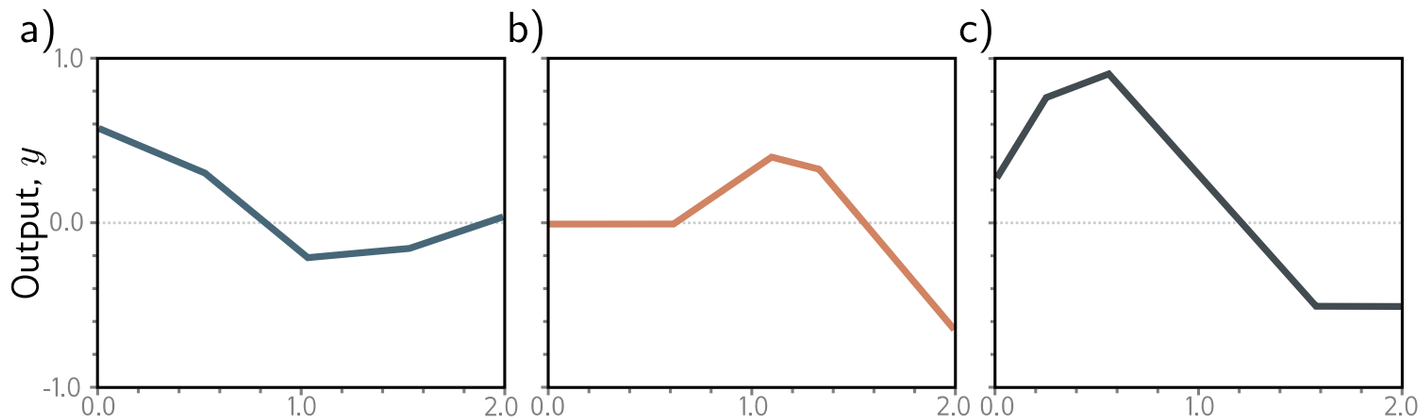
Exponential Linear Unit

- For illustrative purpose, we consider the most common choice known as the **rectified linear unit** or ReLU:

$$\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}.$$

NN Intuition

- In the ten-parameter example, we model the dataset with a family of continuous piecewise linear functions with up to 4 linear regions.



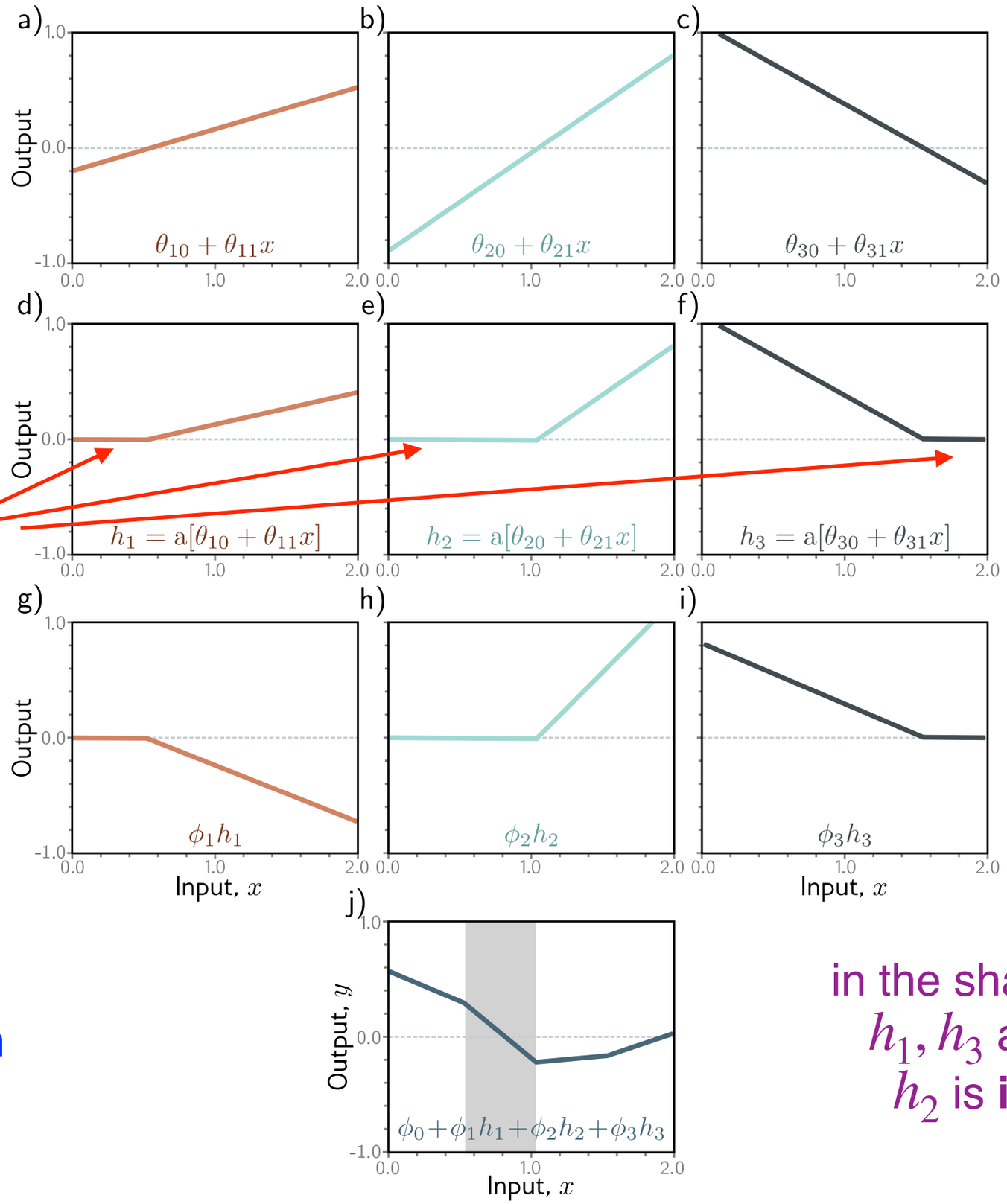
- To see why, we define the intermediate quantities as **hidden units**:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x], \end{aligned}$$

- The output is given by combining the hidden units w/ a linear function:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3.$$

Activation Pattern



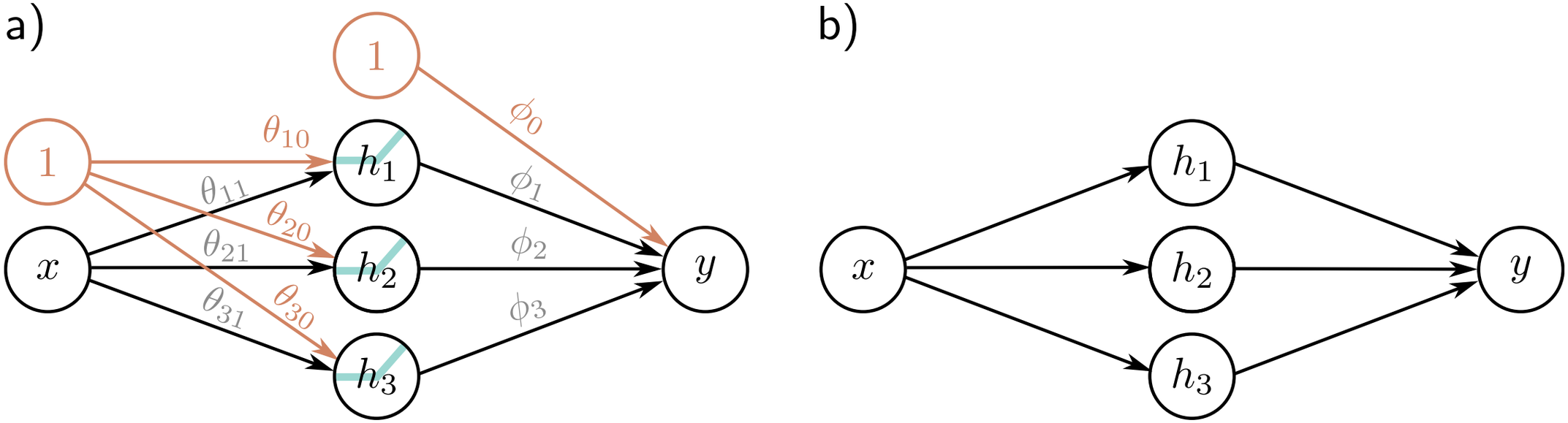
clipped by ReLU

define locations
of the 3 joints
$\Rightarrow$ 4 linear regions

[Only 3 of the slopes
are independent; the
4-th is either zero
or sum of slopes from
the other regions.]

in the shaded region
h_1, h_3 are **active**
h_2 is **inactive**.

Depicting Neural Networks



inputs **hidden units** **outputs** **inputs** **hidden units** **outputs**

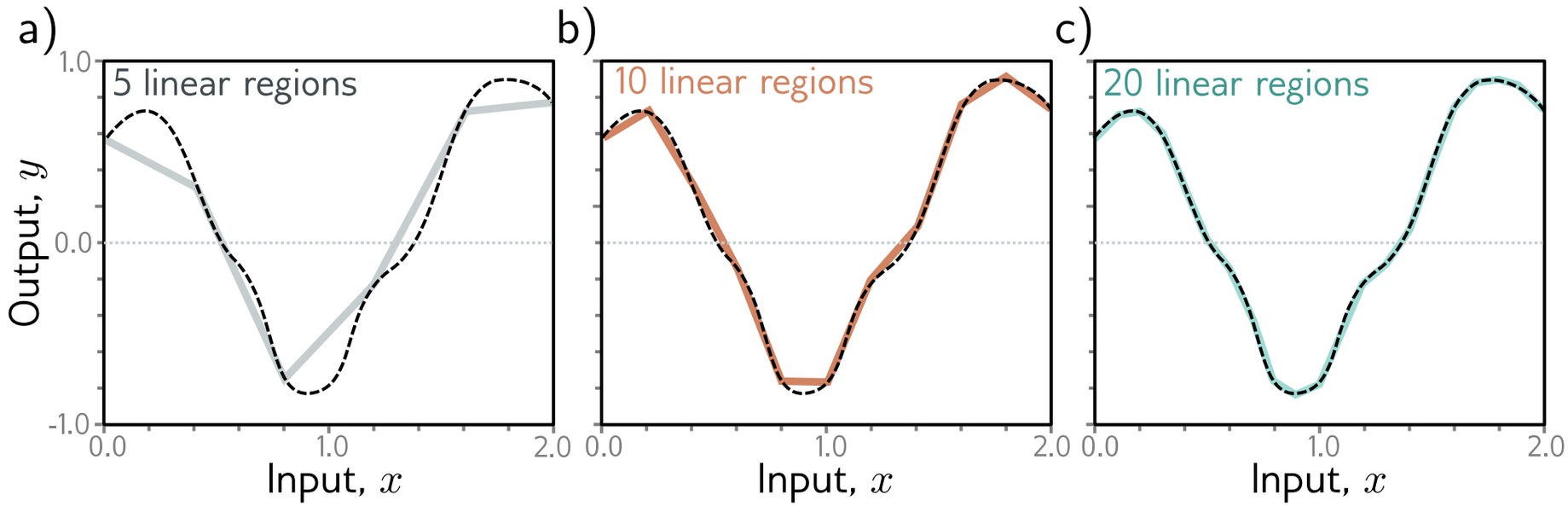
- The intercepts (known as **biases**) are usually not shown in the NN architecture, the NN is simplified to the picture on the right.

Universal Approximation Theorem

- Generalizing to D hidden units:

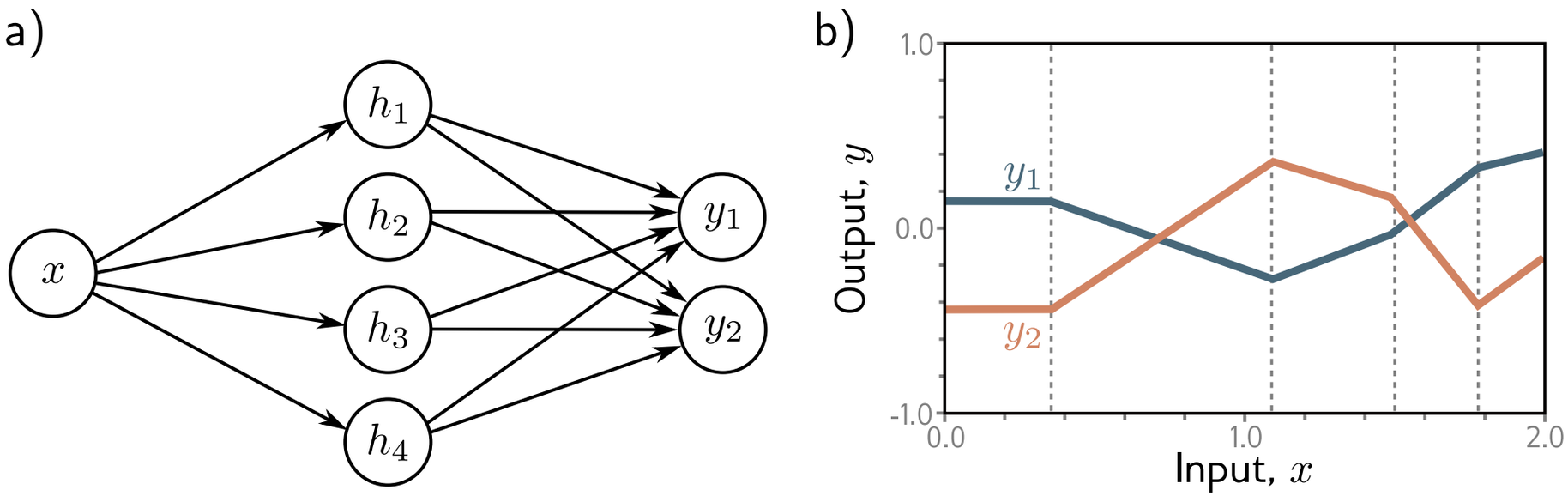
$$h_d = \mathrm{a}[\theta_{d0} + \theta_{d1}x], \qquad y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d.$$

- D = **network capacity**; there are D joints and $D + 1$ linear regions.
- **Universal approximation theorem:** ∀ continuous function, ∃ a shallow network that can approximate it to any specified precision; holds for networks that map multivariate inputs to multivariate outputs.



Multivariate Outputs

- For example, $\mathbf{y} = [y_1, y_2]^T$:



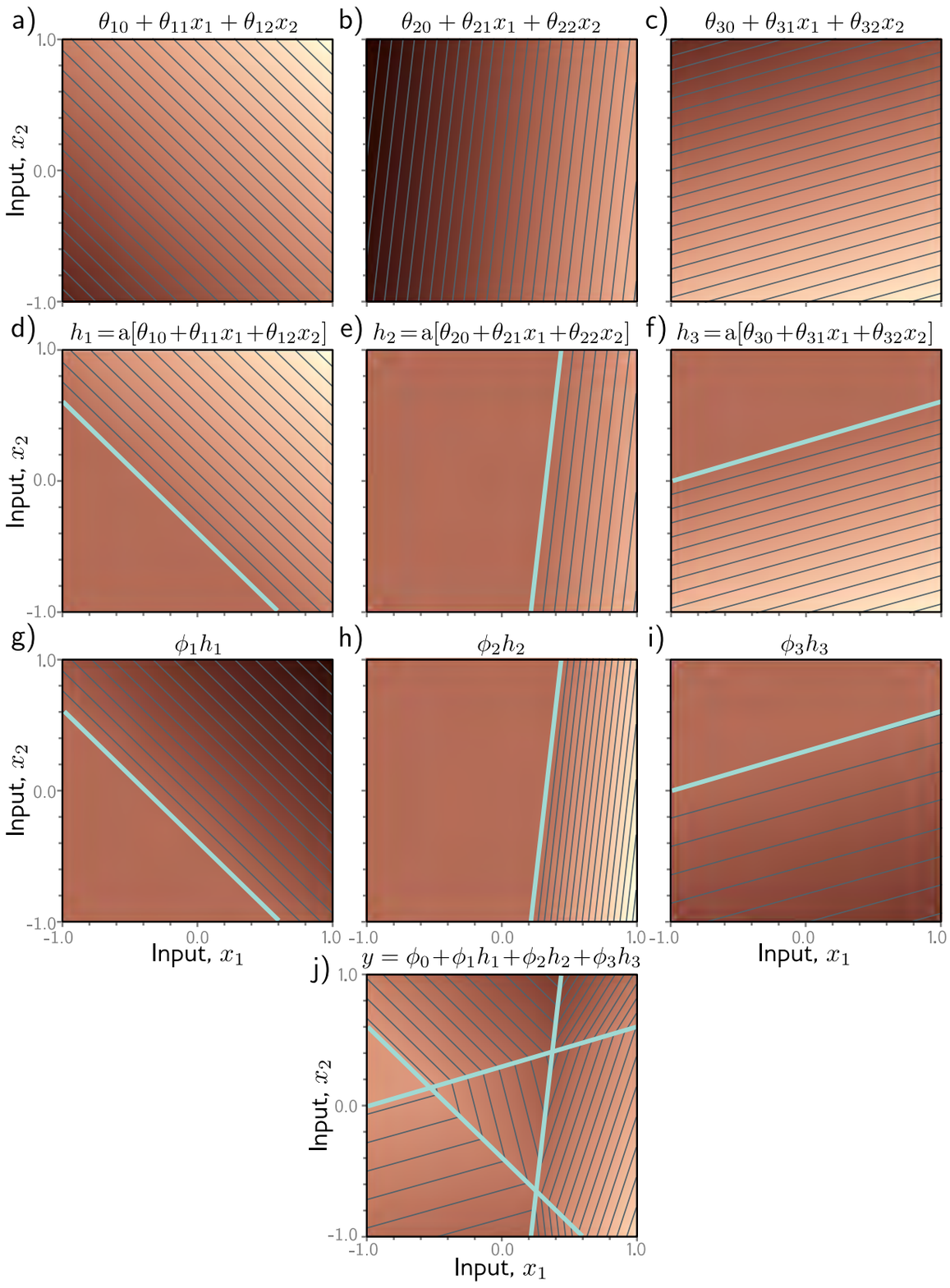
- The hidden units for both outputs are the same:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x] \\ h_4 &= \mathrm{a}[\theta_{40} + \theta_{41}x], \end{aligned}$$

- The joints are the same but the slopes of the linear regions and the vertical offsets can differ:

$$\begin{aligned} y_1 &= \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4 \\ y_2 &= \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4. \end{aligned}$$

Multivariate Inputs



The hidden units depend on both inputs

$$
\begin{aligned}
h_1 &= \mathrm{a}[\theta_{10}+\theta_{11}x_1+\theta_{12}x_2] \\
h_2 &= \mathrm{a}[\theta_{20}+\theta_{21}x_1+\theta_{22}x_2] \\
h_3 &= \mathrm{a}[\theta_{30}+\theta_{31}x_1+\theta_{32}x_2],
\end{aligned}
$$

They create a continuous piecewise linear surface consisting of convex polygonal regions, each with a different activation patten.

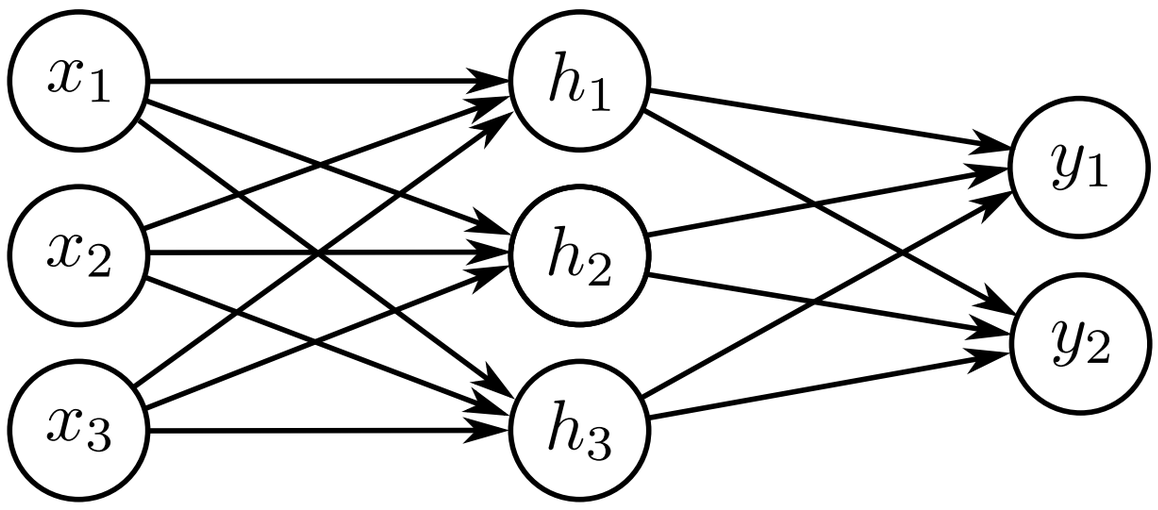
This depicts 2 inputs and 1 output. Generalizable to more than 2 inputs but difficult to visualize such cases.

General Case

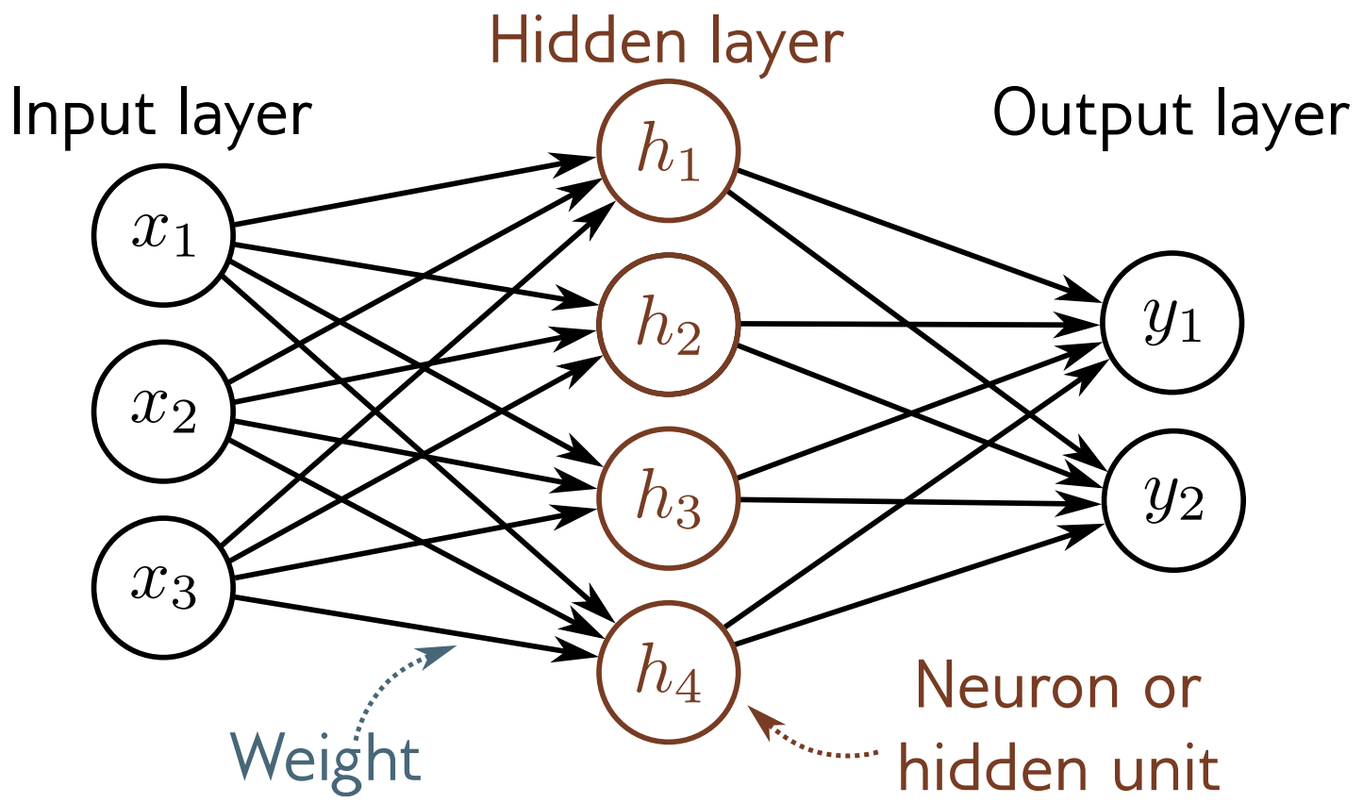
- In general, a shallow NN is a function $\mathbf{y} = f(\mathbf{x}, \boldsymbol{\phi})$ that maps a multi-dimensional input $\mathbf{x} \in \mathbb{R}^{D_i}$ to a multi-dimensional output $\mathbf{y} \in \mathbb{R}^{D_0}$ using $\mathbf{h} \in \mathbb{R}^{D}$ hidden units:

$$h_d = \mathrm{a}\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right], \qquad y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d,$$

- Graphically, a shallow NN is depicted as e.g.



Terminology

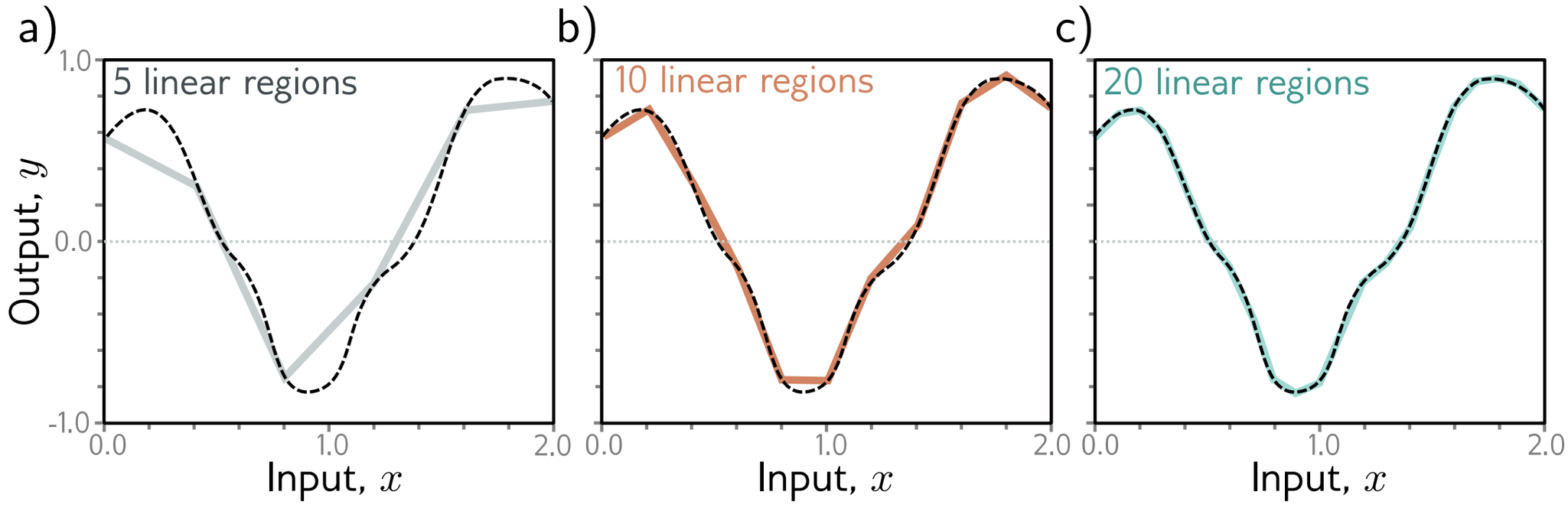


- Any NN with at least one hidden layer is called a multi-layer perceptron, or MLP.
- NNs with one hidden layer are called shallow NNs. NNs with multiple hidden layers are called deep NNs.
- NNs with connections form an acyclic graph (a graph w/0 loops) are feedforward NNs.
- Every element in one layer connects to every element in the next: fully connected NNs.

Why going deep?

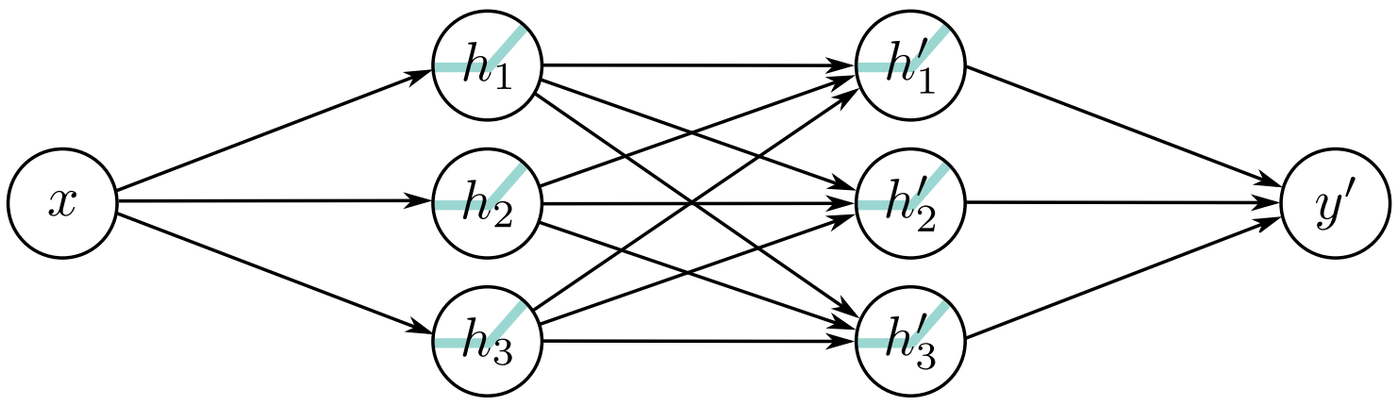
- A shallow NN with only a single hidden layer can already approximate any continuous function to a specific precision, using piecewise linear functions.

- However, the network capacity (# hidden units) may be impractically large. A deep NN can produce more linear regions for a given # parameters.



Deep Neural Networks

- The composition of 2 shallow networks results in a 2-layer network:



- First layer:

$$\begin{aligned} h_1 &= \mathrm{a}[\theta_{10} + \theta_{11}x] \\ h_2 &= \mathrm{a}[\theta_{20} + \theta_{21}x] \\ h_3 &= \mathrm{a}[\theta_{30} + \theta_{31}x] \end{aligned}$$

- Second layer:

$$\begin{aligned} h'_1 &= \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\ h'_2 &= \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\ h'_3 &= \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3] \end{aligned}$$

- Output:

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3$$

Deep Neural Networks

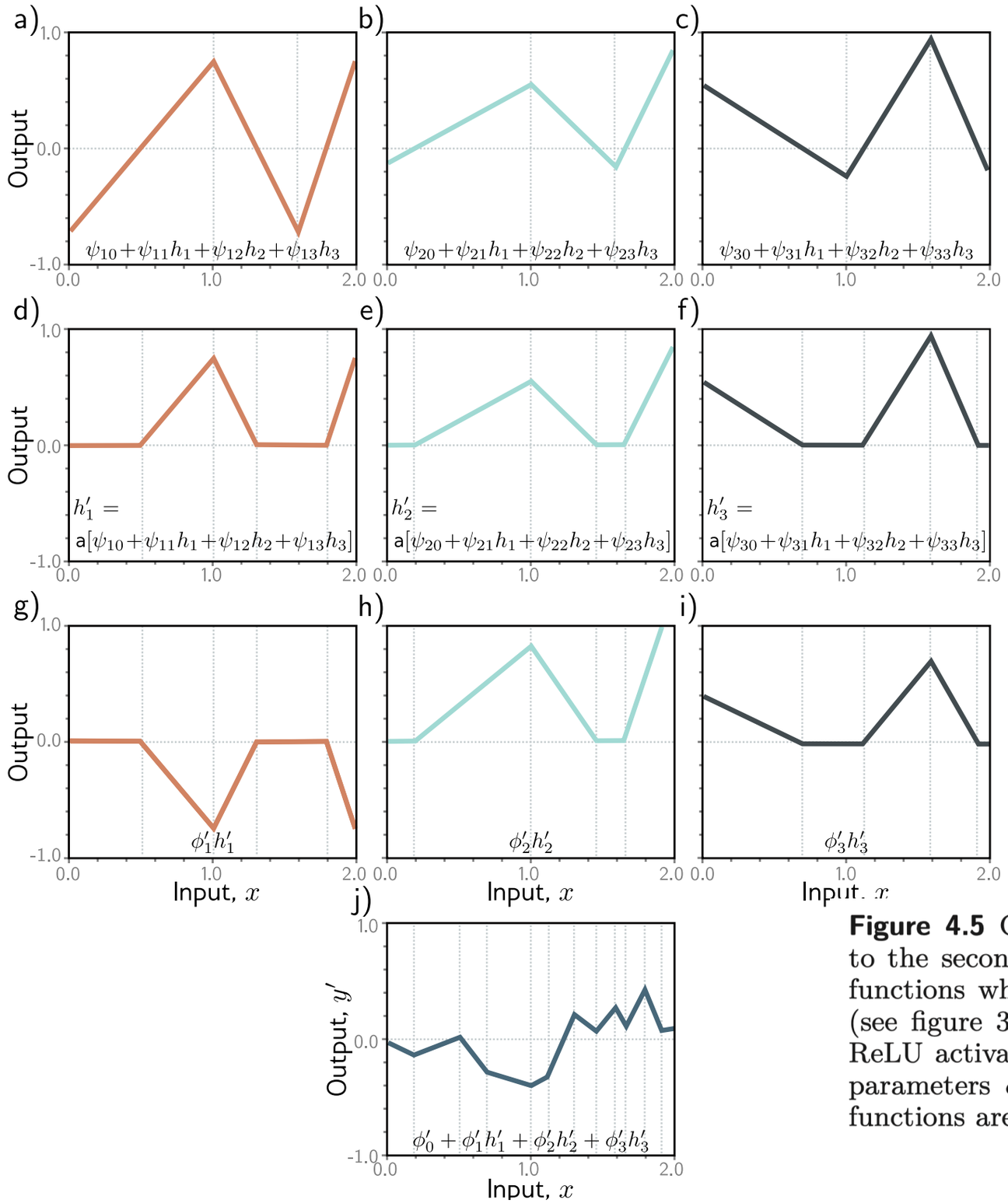
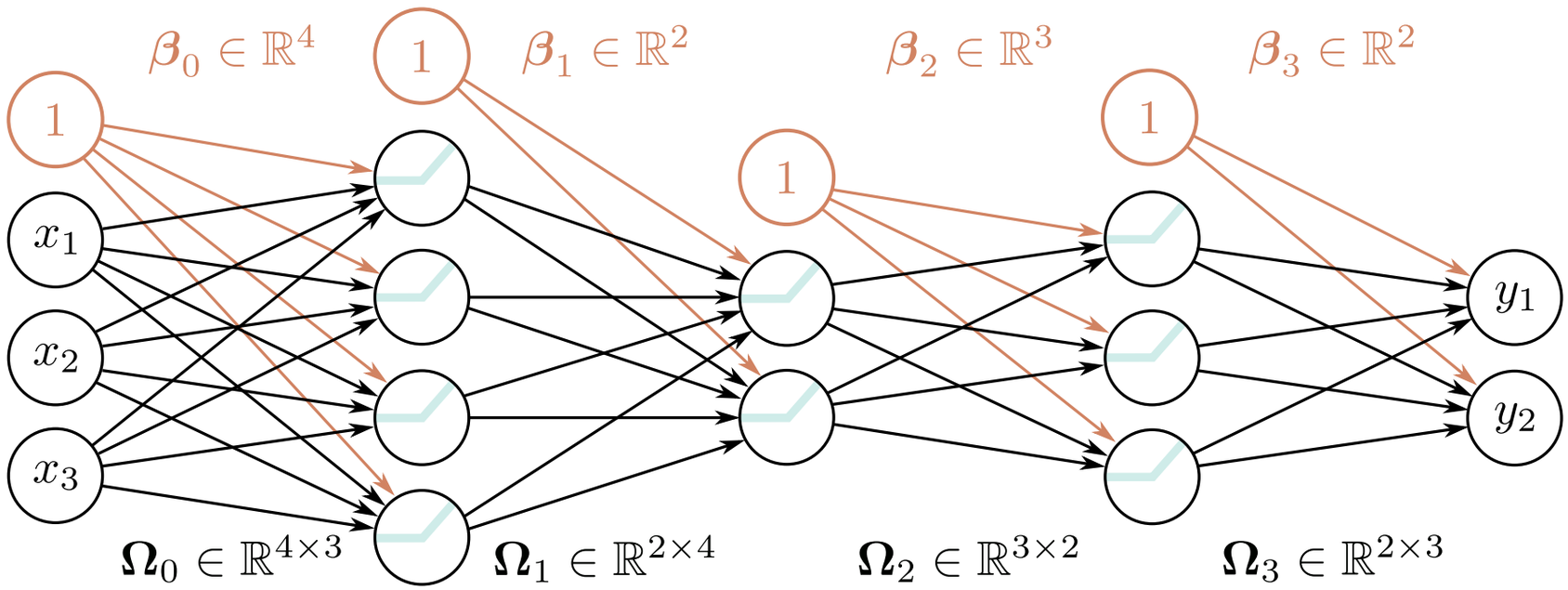


Figure 4.5 Computation for the deep network in figure 4.4. a–c) The inputs to the second hidden layer (i.e., the pre-activations) are three piecewise linear functions where the "joints" between the linear regions are at the same places (see figure 3.6). d–f) Each piecewise linear function is clipped to zero by the ReLU activation function. g–i) These clipped functions are then weighted with parameters ϕ'_1, ϕ'_2, and ϕ'_3, respectively. j) Finally, the clipped and weighted functions are summed and an offset ϕ'_0 that controls the overall height is added.

A complicated (piece-wise linear) function emerges.

Hyperparameters



Input, $\mathbf{x}$	Hidden layer, $\mathbf{h}_1$	Hidden layer, $\mathbf{h}_2$	Hidden layer, $\mathbf{h}_3$	Output, $\mathbf{y}$
$D_i = 3$	$D_1 = 4$	$D_2 = 2$	$D_3 = 3$	$D_o = 2$

- Modern deep NNs can have $\mathcal{O}(10)$ to $\mathcal{O}(10^2)$ layers with $\mathcal{O}(10^3)$ of hidden units in each layer.
- The number of layers K = **depth**, & the number of hidden units in each layer (=**width**) $D_1, D_2, \ldots, D_K$ are hyperparameters. The network **capacity** = # number of hidden units.
- For fixed hyperparameters, the model describes a family of functions, and the parameters θ (known as weights) determine a specific function.

Back to matrix notation

- We can express a 2-layer network in **matrix notation**:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \theta_{10} \\ \theta_{20} \\ \theta_{30} \end{bmatrix} + \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \end{bmatrix} x\right],$$

$$\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix} = \mathbf{a}\left[\begin{bmatrix} \psi_{10} \\ \psi_{20} \\ \psi_{30} \end{bmatrix} + \begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{31} & \psi_{32} & \psi_{33} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}\right],$$

and

$$y' = \phi'_0 + \begin{bmatrix} \phi'_1 & \phi'_2 & \phi'_3 \end{bmatrix} \begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix},$$

$$\begin{aligned} \mathbf{h} &= \mathbf{a}\left[\boldsymbol{\theta}_0 + \boldsymbol{\theta} x\right] \\ \mathbf{h}' &= \mathbf{a}\left[\boldsymbol{\psi}_0 + \boldsymbol{\Psi}\mathbf{h}\right] \\ y' &= \phi'_0 + \boldsymbol{\phi}'\mathbf{h}', \end{aligned}$$

- More generally, a K-layer network:

$$\begin{aligned} \mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2] \\ &\vdots \\ \mathbf{h}_K &= \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} \mathbf{h}_{K-1}] \\ \mathbf{y} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{h}_K. \end{aligned}$$

Hyperparameters:

$$K, D_1, D_2, \ldots, D_K$$

Parameters: biases and matrices

Shallow vs Deep

- **Universal approximation theorem:** deep NNs can approximate any continuous function arbitrarily closely given sufficient capacity.
  - We can reproduce a shallow network if all but one layer is the identity function. Since we showed that a shallow NN can approximate any continuous function, deep NNs also work.
- **More expressive** (more linear regions per parameter):
  - A shallow NN with 1 input, 1 output, $D > 2$ hidden units (in 1 layer) can create up to $D + 1$ linear regions using $3D + 1$ parameters.
  - A deep NN with 1 input, 1 output, $D > 2$ hidden units in K layers of same dimension D can create up to $(D + 1)^K$ linear regions using $3D + 1 + (K - 1)D(D + 1)$ parameters.

This exponential growth in linear regions is what makes deep NN more expressive.

Shallow vs Deep

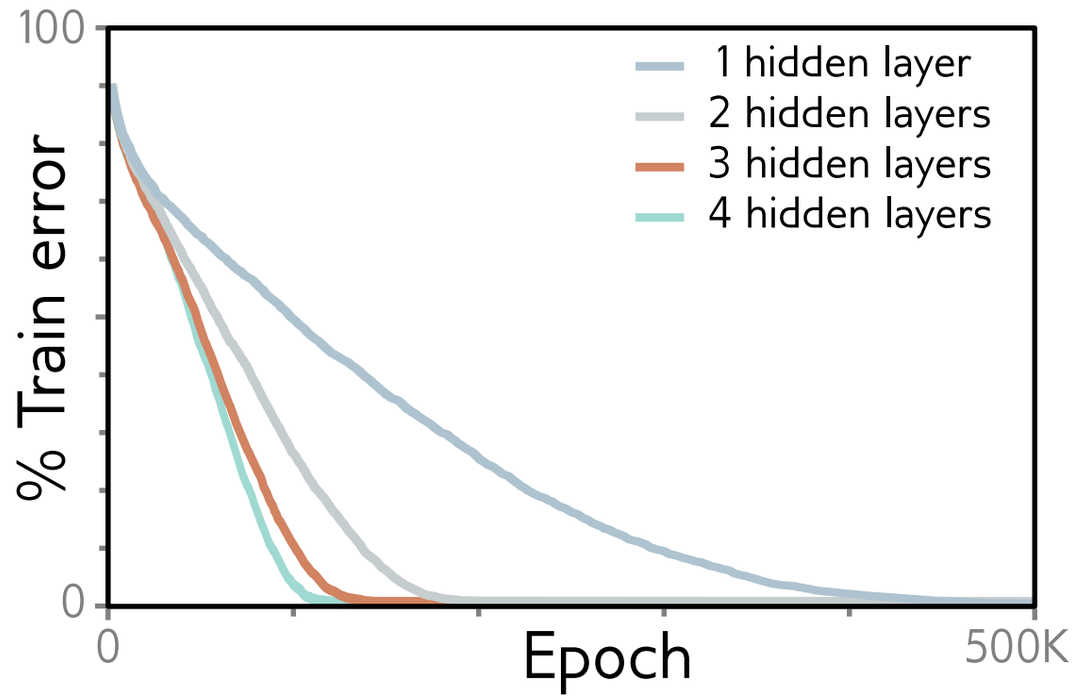
- The counting of parameters for shallow NNs goes as follows:
  - **There are D hidden units, each has two parameters (bias, weight). The output layer has D weights and one bias. # parameter=2D+D+1=3D+1.**
- The counting of parameters for deep NNs goes as follows:
  - **There are D weights between the input and the first hidden layer, $K-1$ lots of $D \times D$ inputs between adjacent hidden layers, and D weights between the last hidden layer and the output. There are D biases at each of the K hidden layers and 1 bias for the output. This gives $D + (K-1)D^2 + D + KD + 1 = \quad 3D + (K-1)D^2 + (K-1)D + 1$ parameters.**

Shallow vs Deep

- Deep NNs create much more linear regions for a fixed parameter budget, but they contain complex dependence and symmetries.
- The greater number of regions is an advantage if:
    1. there are similar symmetries in the function to approximate;
    2. the input→output map is a composition of simpler functions.
- **Depth efficiency** refers to the phenomenon that a shallow NN needs exponentially more hidden units to achieve an equivalent approximation to that of a deep NN.

Shallow vs Deep

- **Training and generalization:** It is easier to train moderately deep networks than to train shallow ones.



- Deep NNs also seem to generalize to new data better than shallow ones.
- Empirically, one finds best results for most tasks using networks with a few (or more) hidden layers layers.

Course logistics

- **Reading for this lecture:**
  - https://arxiv.org/pdf/1803.08823 (Optimization)
  - Simon Prince “Understanding Deep Learning” (Shallow and Deep NN sections). Free online: https://udlbook.github.io/udlbook/

- **Problem set**: Second problem due next Wednesday