Neural Networks

Introduction, multilayer perceptron, optimization techniques

Machine Learning and Data Mining, 2021

Artem Maevskiy

National Research University Higher School of Economics

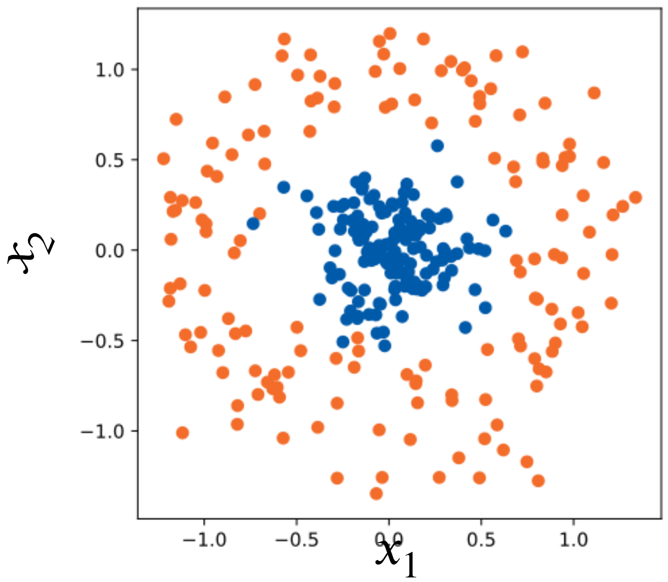
LAMBDA • HSE

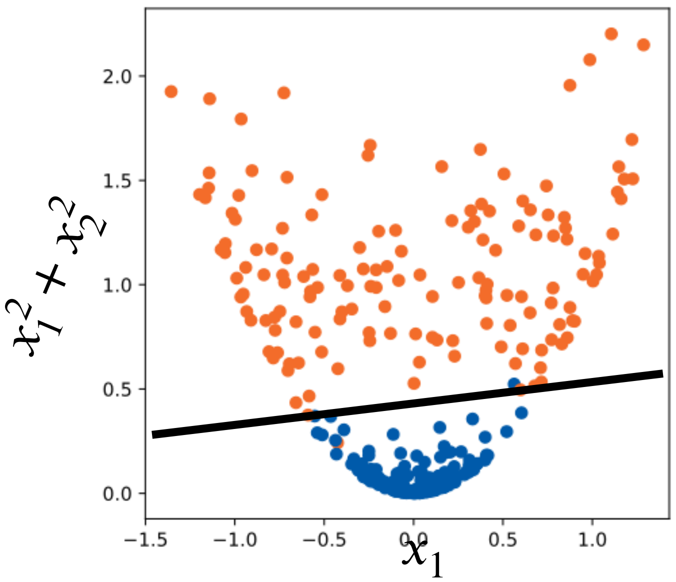
November 10, 2021

From linear model to a neural network

Linear models + feature expansion recap

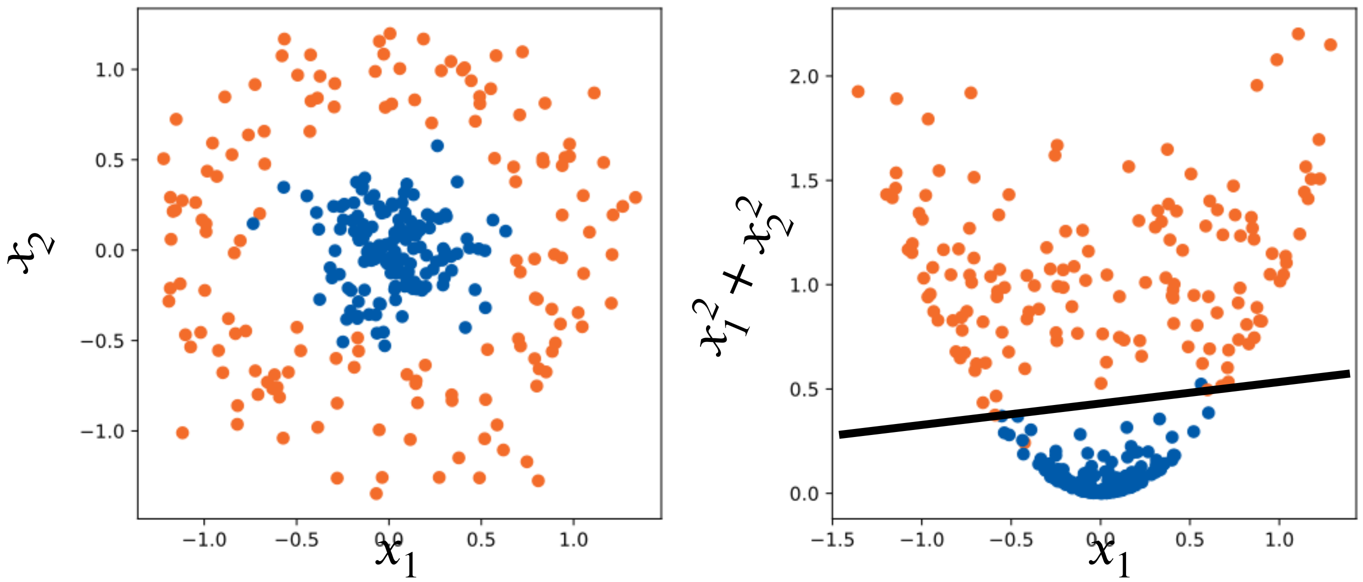
- Recall how, for linear models, we introduced **new features** to make the model **more powerful**
- Finding good features (aka feature engineering) is a **highly non-trivial task**





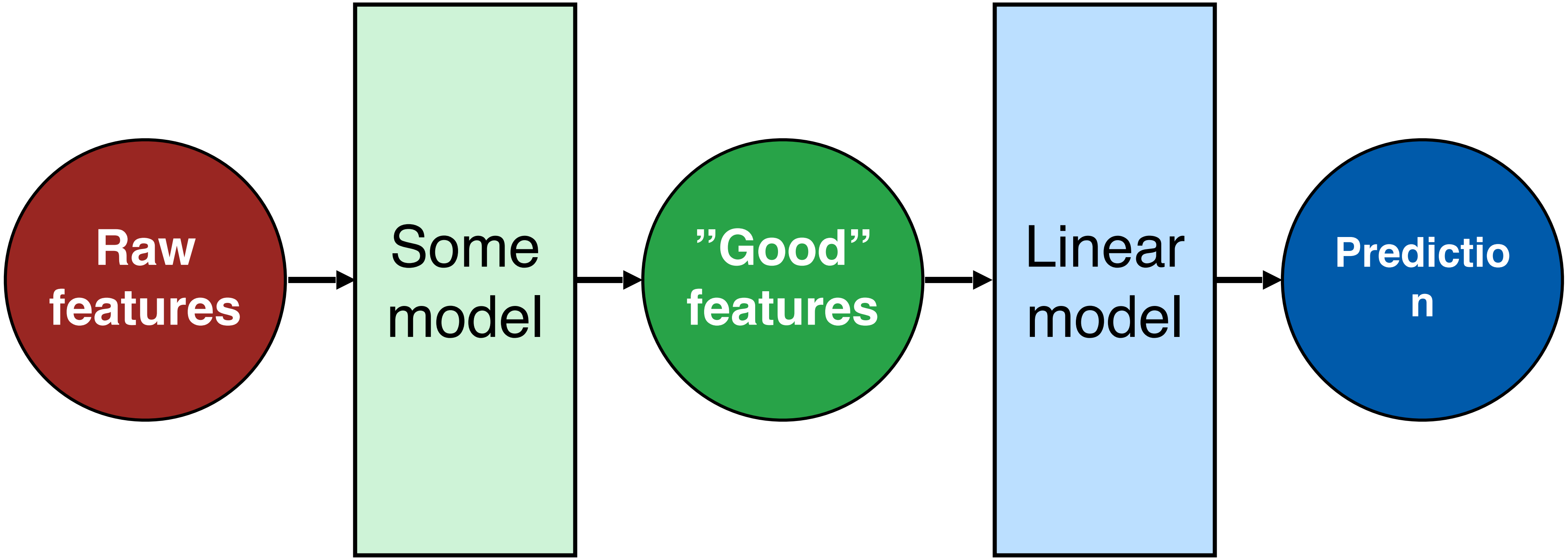
Linear models + feature expansion recap

- Recall how, for linear models, we introduced **new features** to make the model **more powerful**
- Finding good features (aka feature engineering) is a **highly non-trivial task**



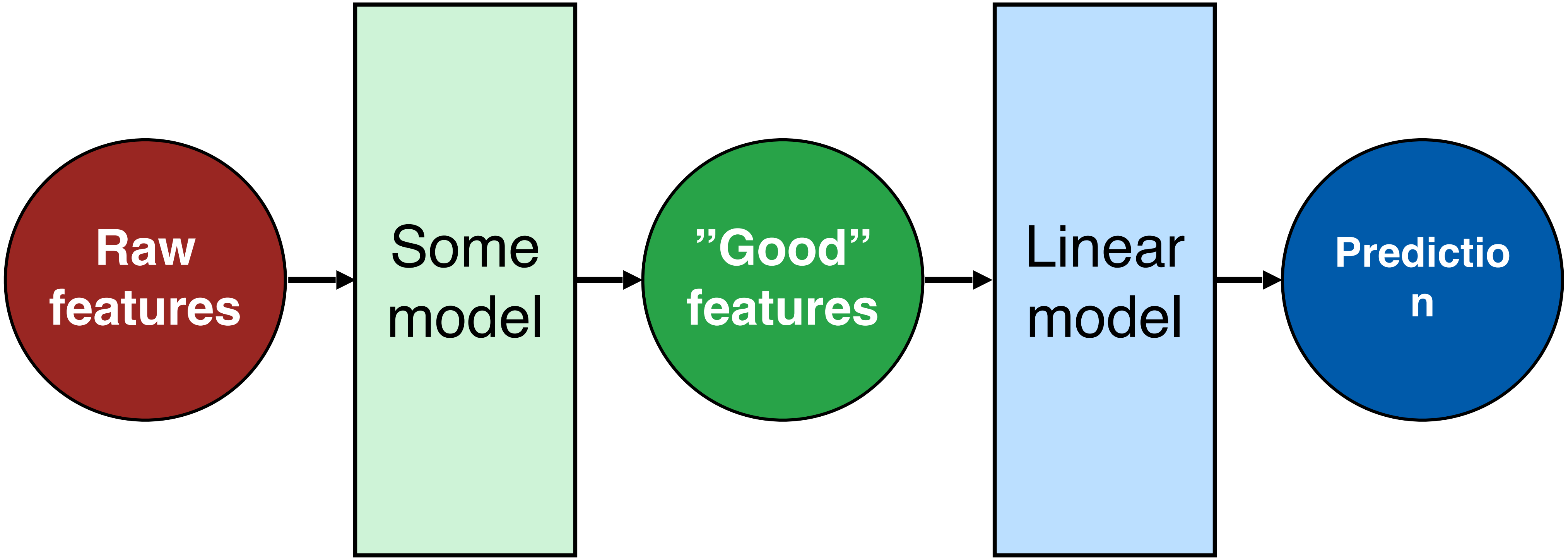
Can we automate feature engineering? ☺

Idea: add another model



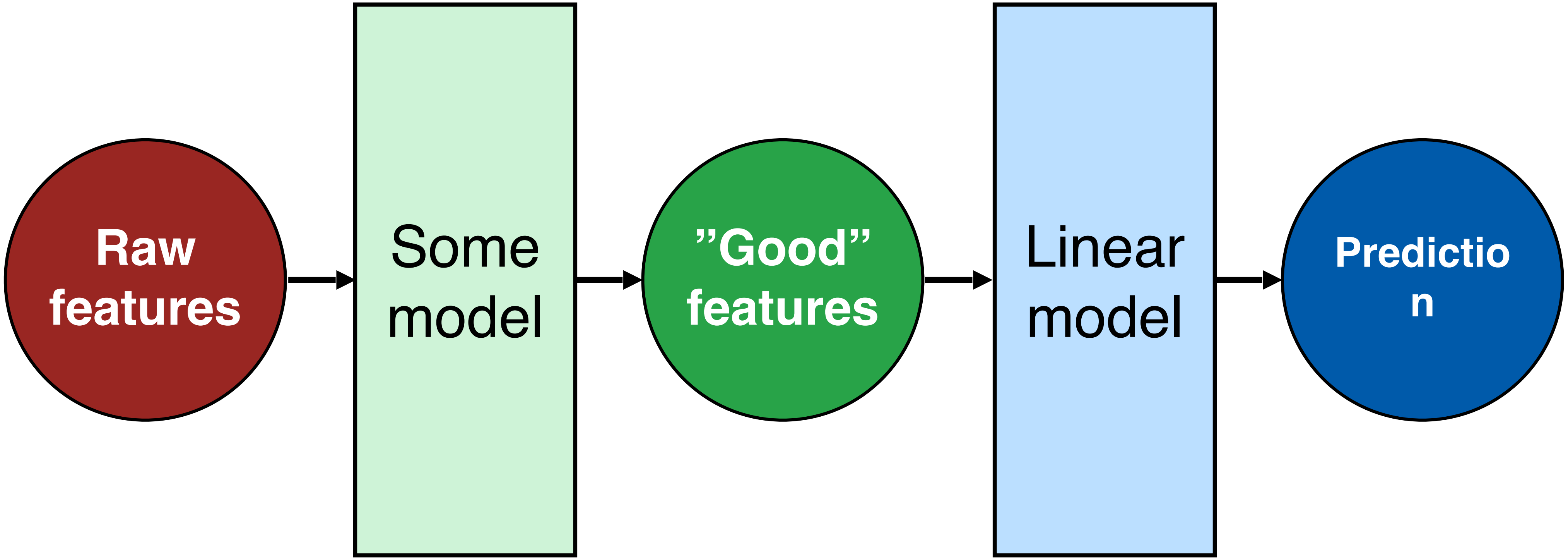
- Add another model

Idea: add another model



- Add another model
- Train everything simultaneously
  - Can use gradient descent if both models are **differentiable**

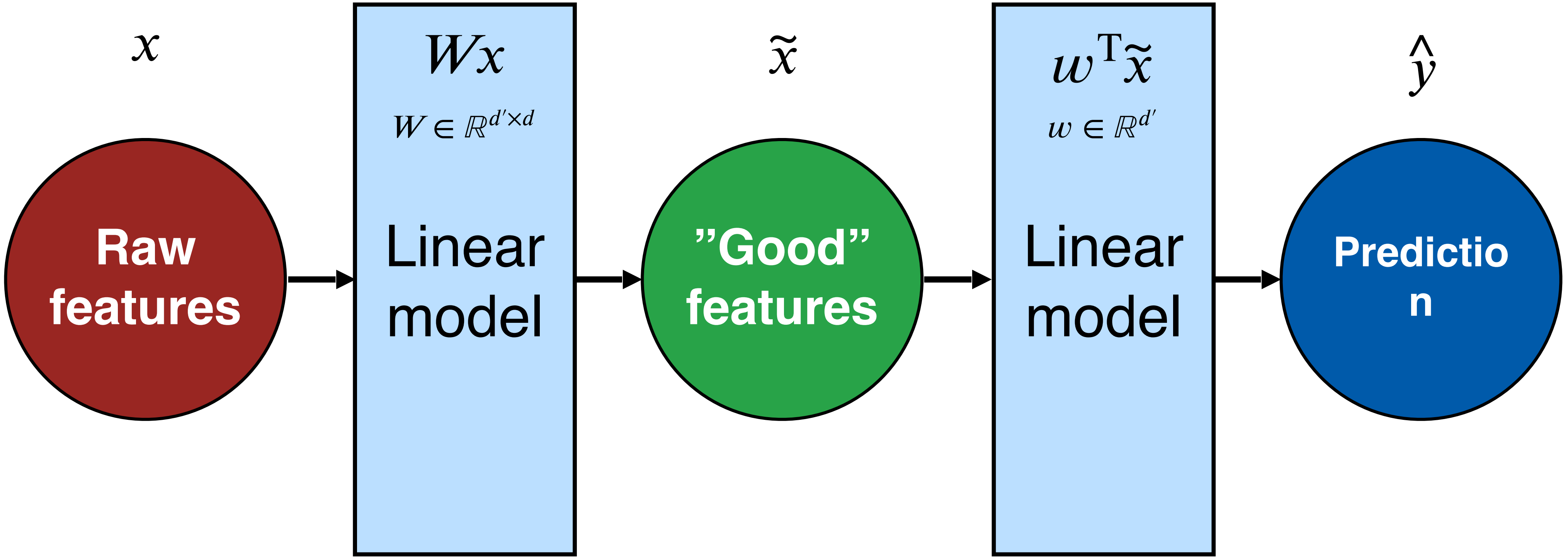
Idea: add another model



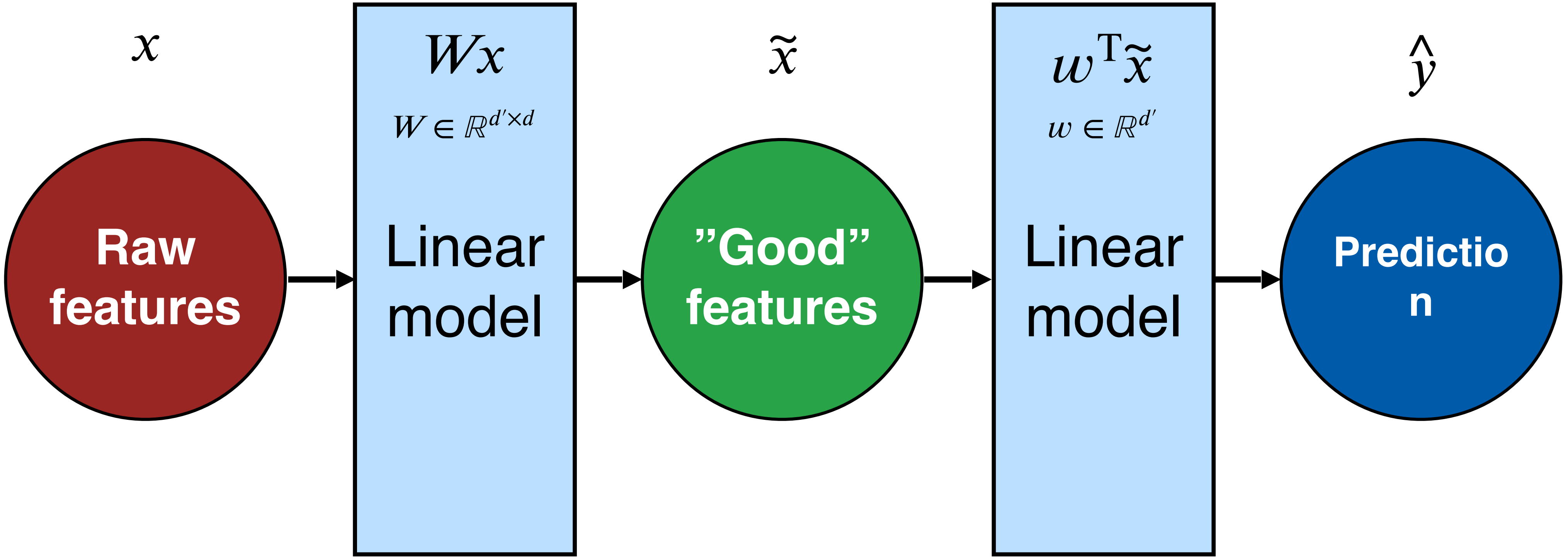
- Add another model
- Train everything simultaneously
  - Can use gradient descent if both models are **differentiable**

Note: stacking models like this likely makes the problem non-convex
⇒ **no convergence guarantees**

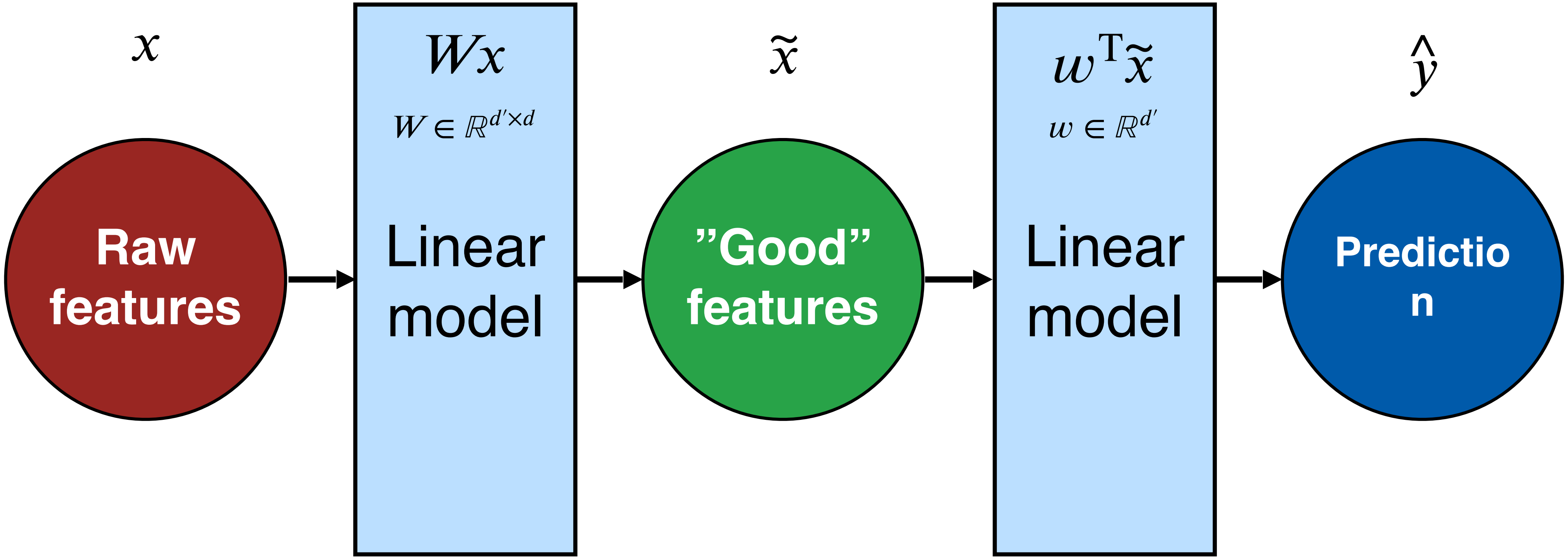
Can it be another linear model?



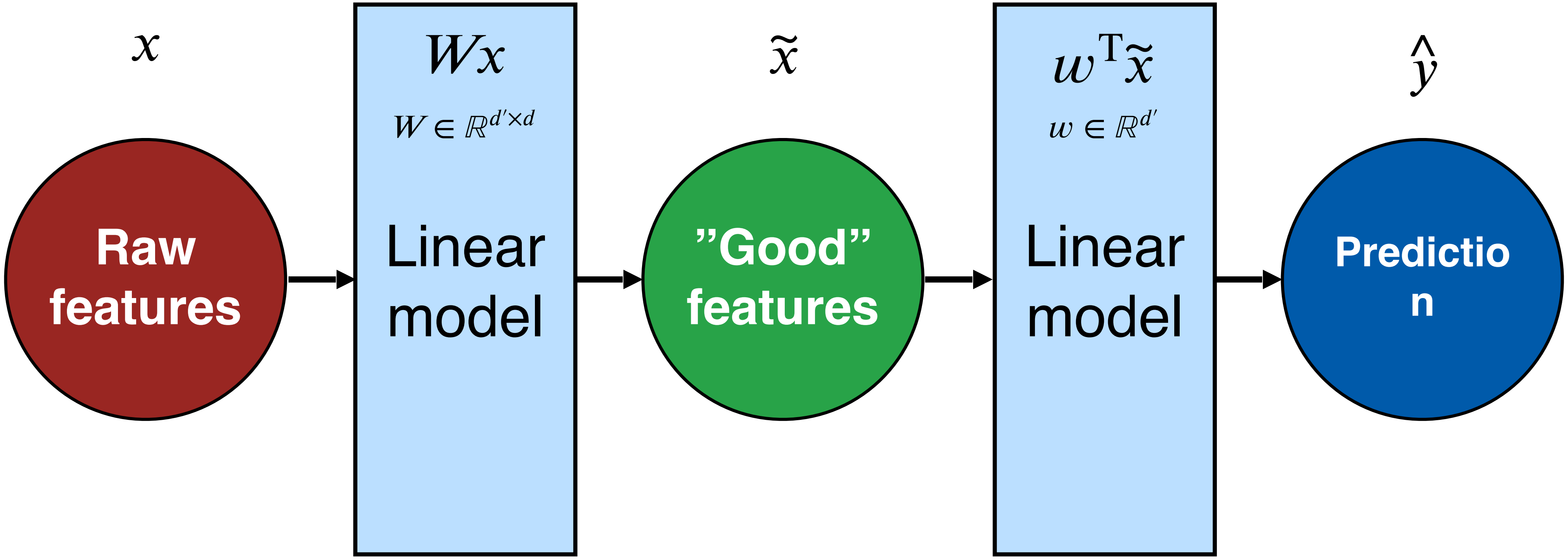
Can it be another linear model?



Can it be another linear model?



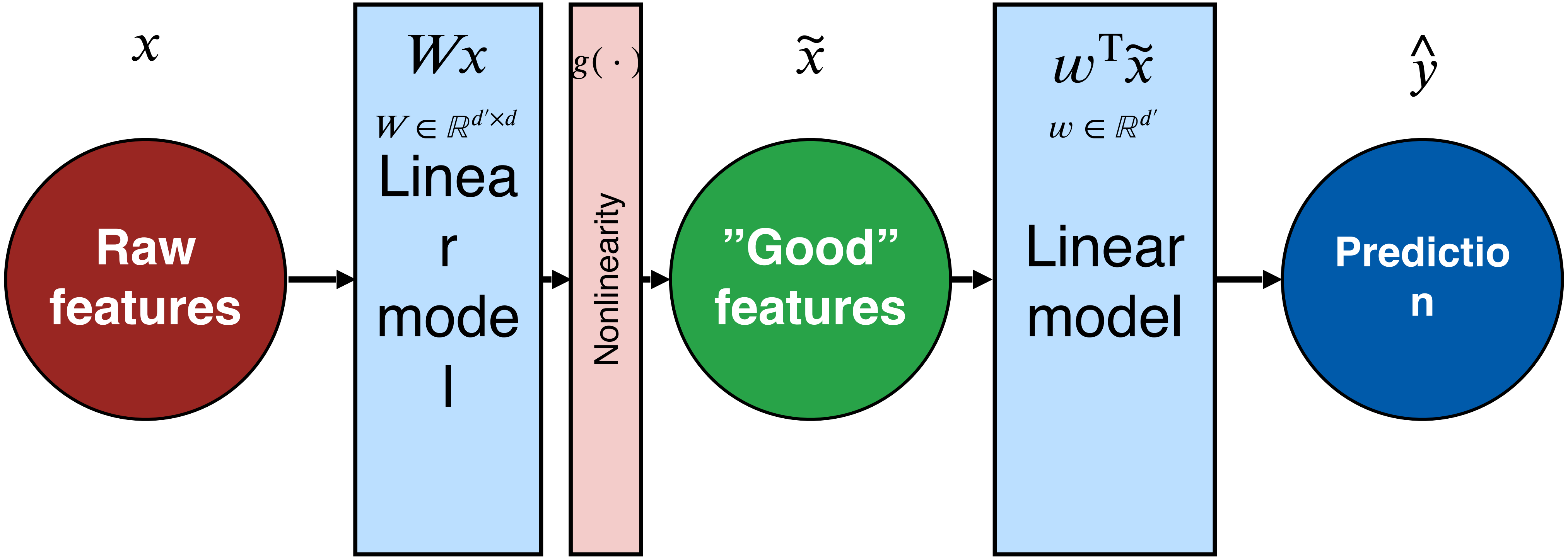
Can it be another linear model?



$$\hat{y} = w^{\mathrm{T}}\tilde{x} = w^{\mathrm{T}}(Wx) = \left(w^{\mathrm{T}}W\right)x = w'^{\mathrm{T}}x$$

– turns everything into just a single linear model

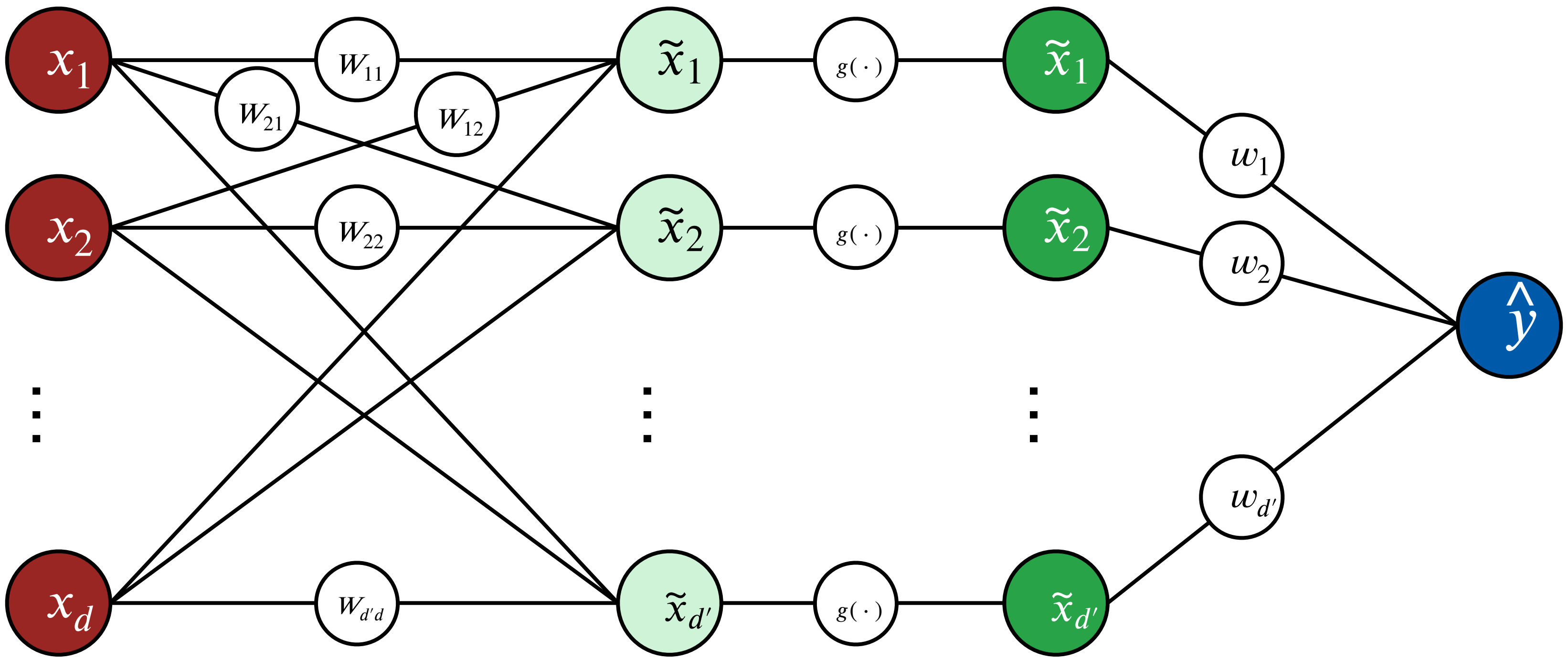
Fix: just introduce a nonlinearity



$$\hat{y} = w^{\mathrm{T}}\tilde{x} = w^{\mathrm{T}}g(Wx)$$

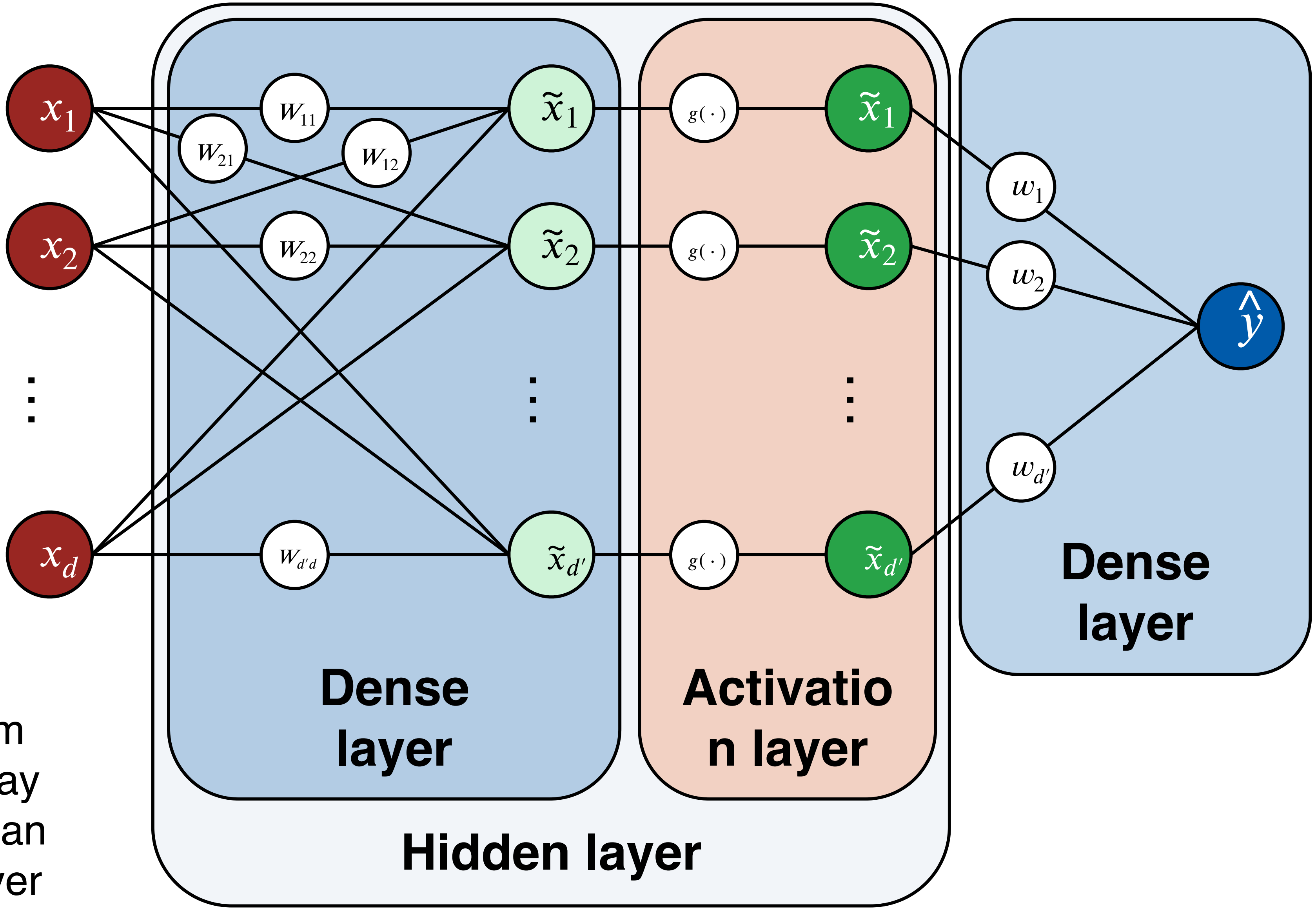
$g(\cdot)$ – some **nonlinear** scalar function (applied elementwise)

In greater detail



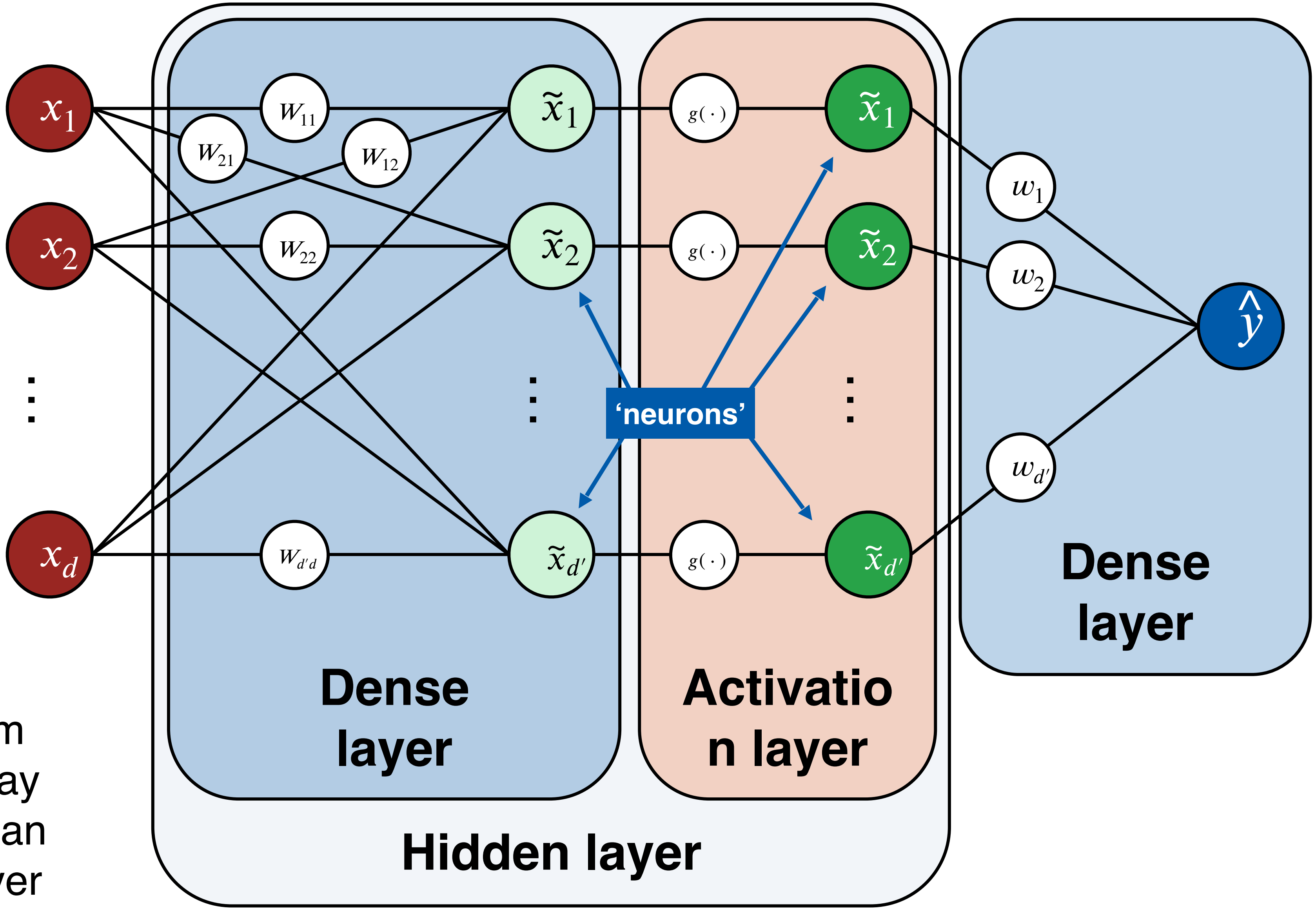
$$\hat{y} = w^{\mathrm{T}}\tilde{x} = w^{\mathrm{T}}g(Wx) = \sum_j \left[w_j g\left(\sum_i W_{ji} x_i \right) \right]$$

Some terminology



Note: the term "activation" may also stand for an output of a layer

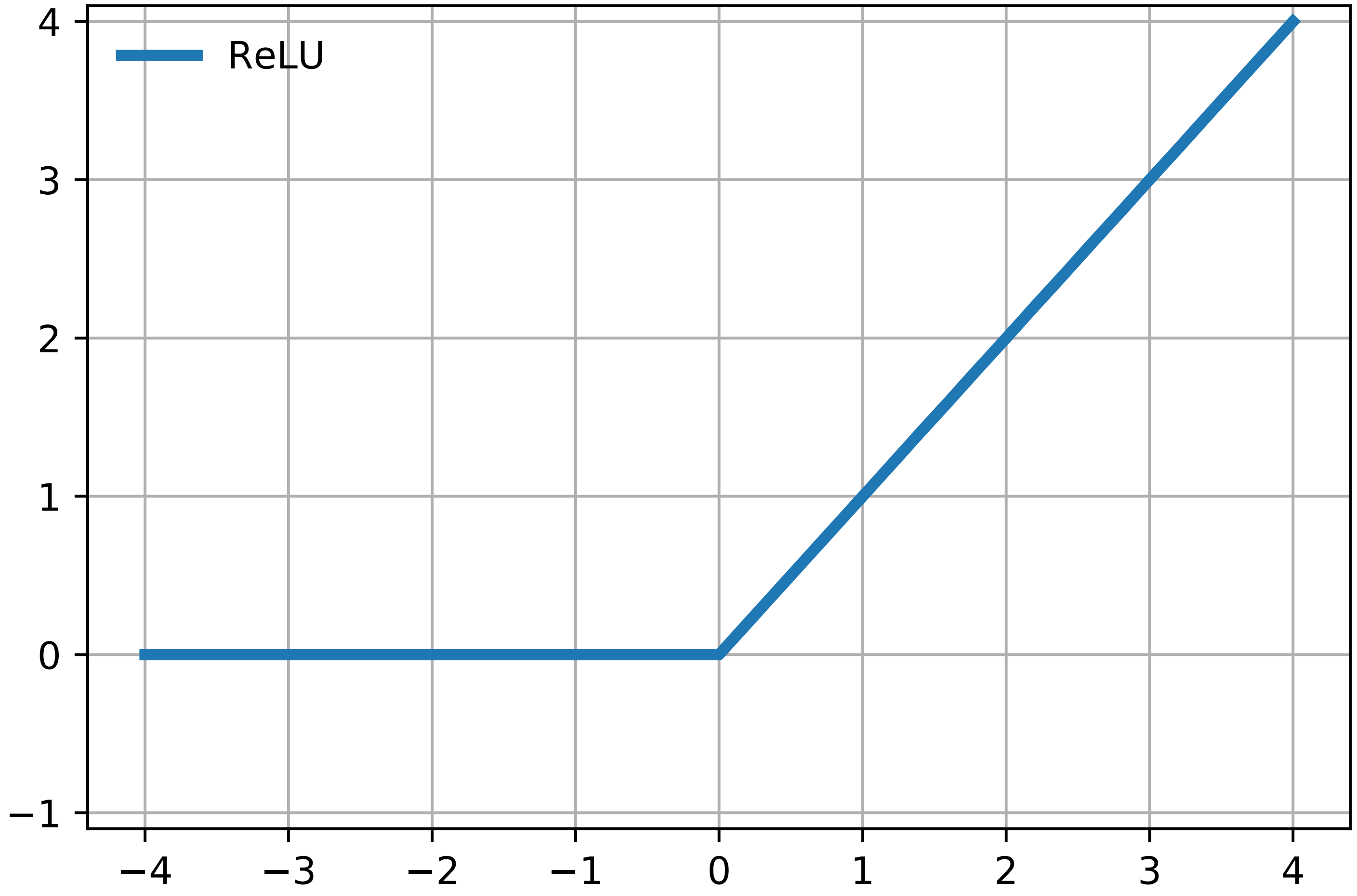
Some terminology



Note: the term "activation" may also stand for an output of a layer

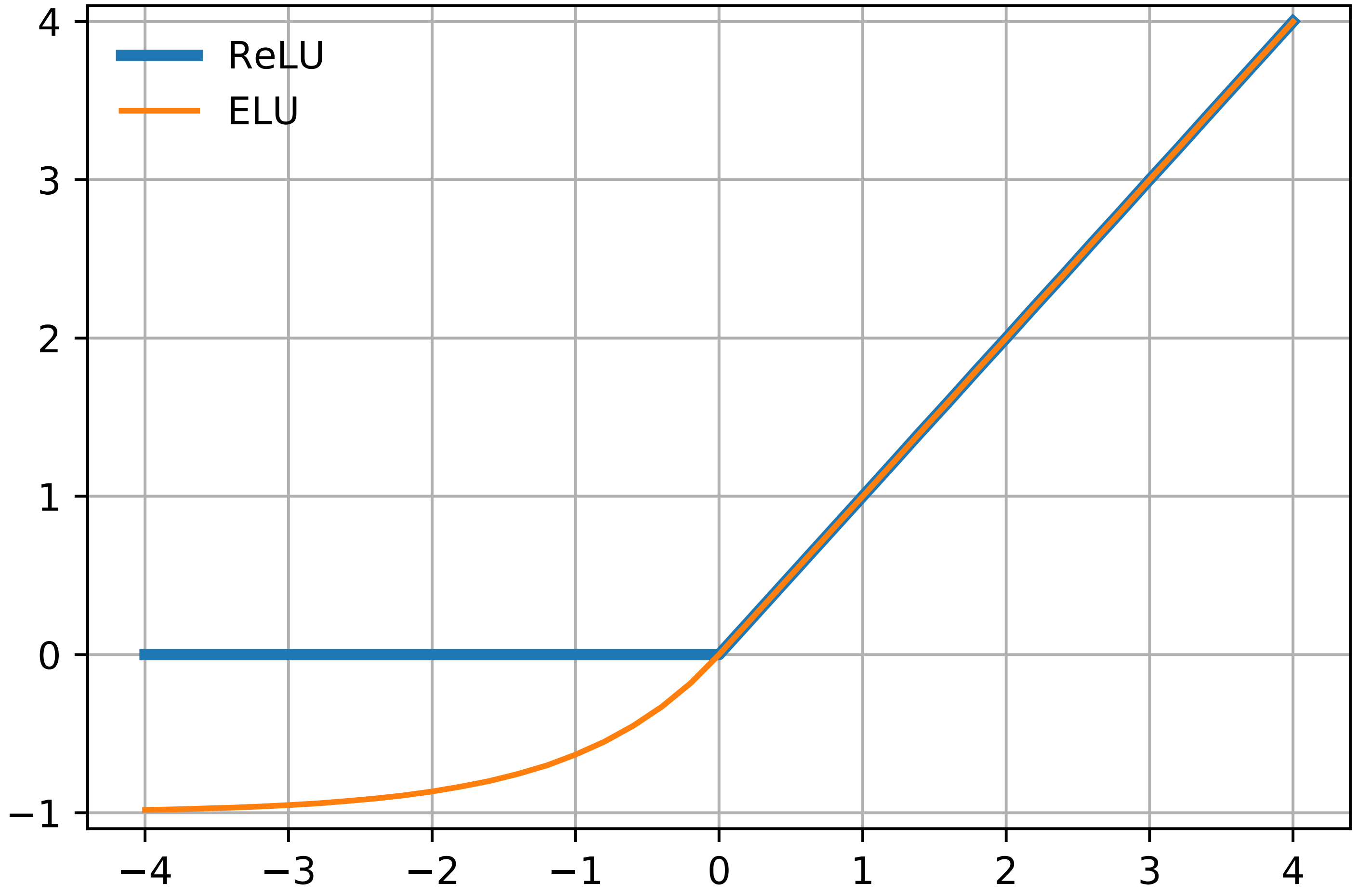
Activation functions

Activation functions



$$\mathrm{ReLU}(x) = \max(0, x)$$

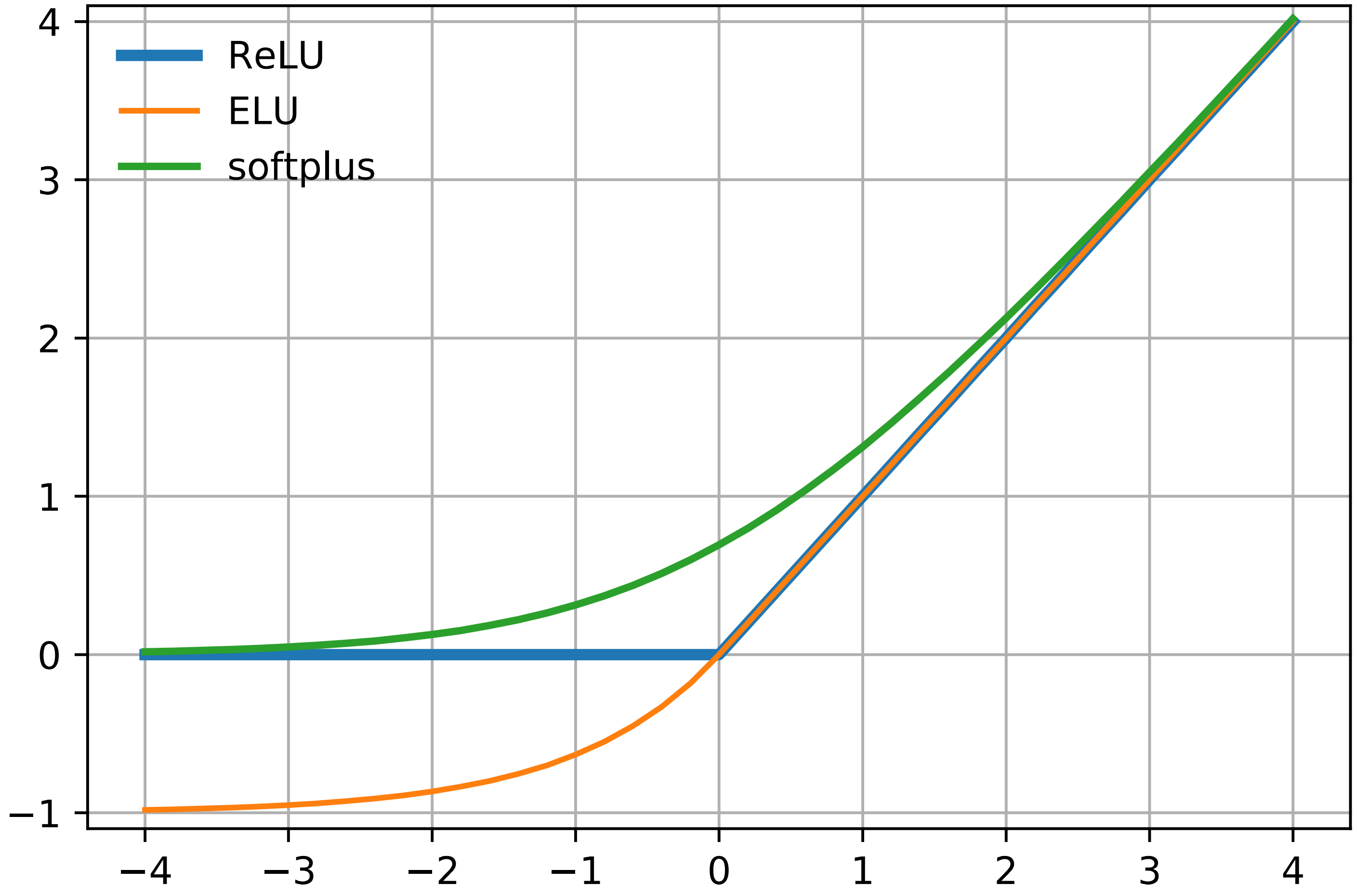
Activation functions



$$\mathrm{ReLU}(x) = \max(0, x)$$

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

Activation functions

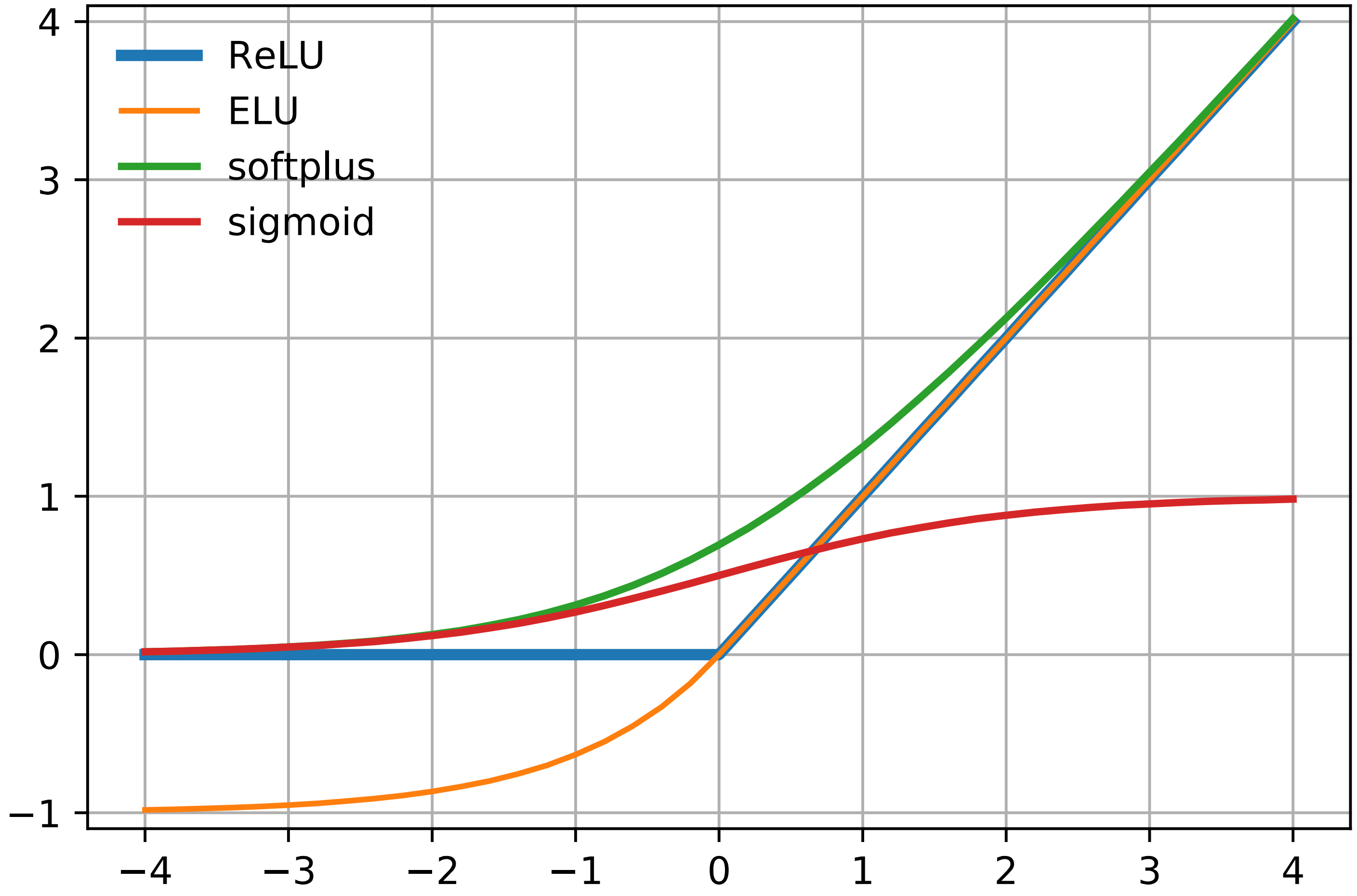


$$\mathrm{ReLU}(x) = \max(0, x)$$

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

$$\mathrm{softplus}(x) = \log\left(1 + e^x\right)$$

Activation functions



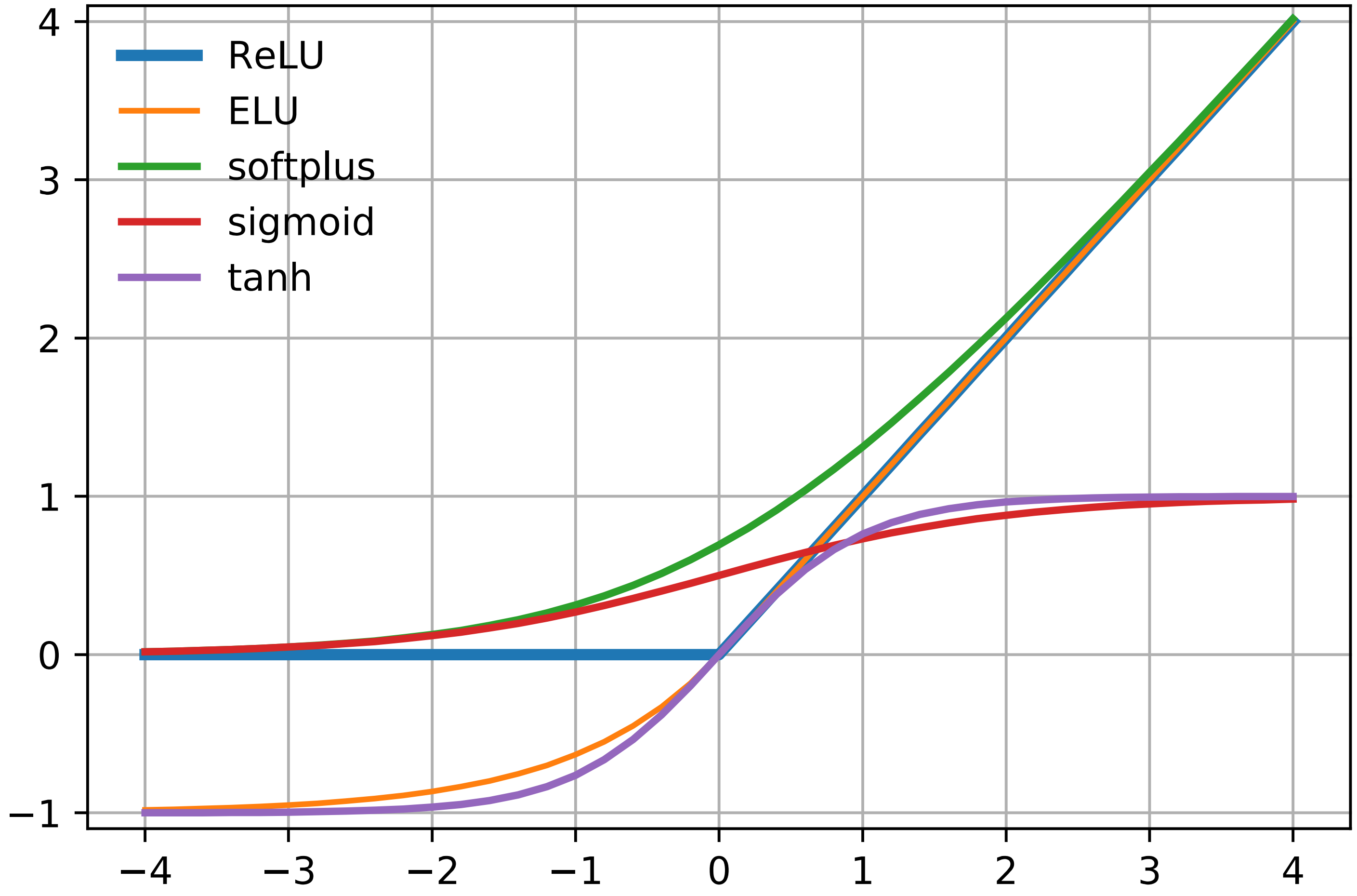
$$\mathrm{ReLU}(x) = \max(0, x)$$

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

$$\mathrm{softplus}(x) = \log\left(1 + e^x\right)$$

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

Activation functions



$$\mathrm{ReLU}(x) = \max(0, x)$$

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

$$\mathrm{softplus}(x) = \log\left(1 + e^x\right)$$

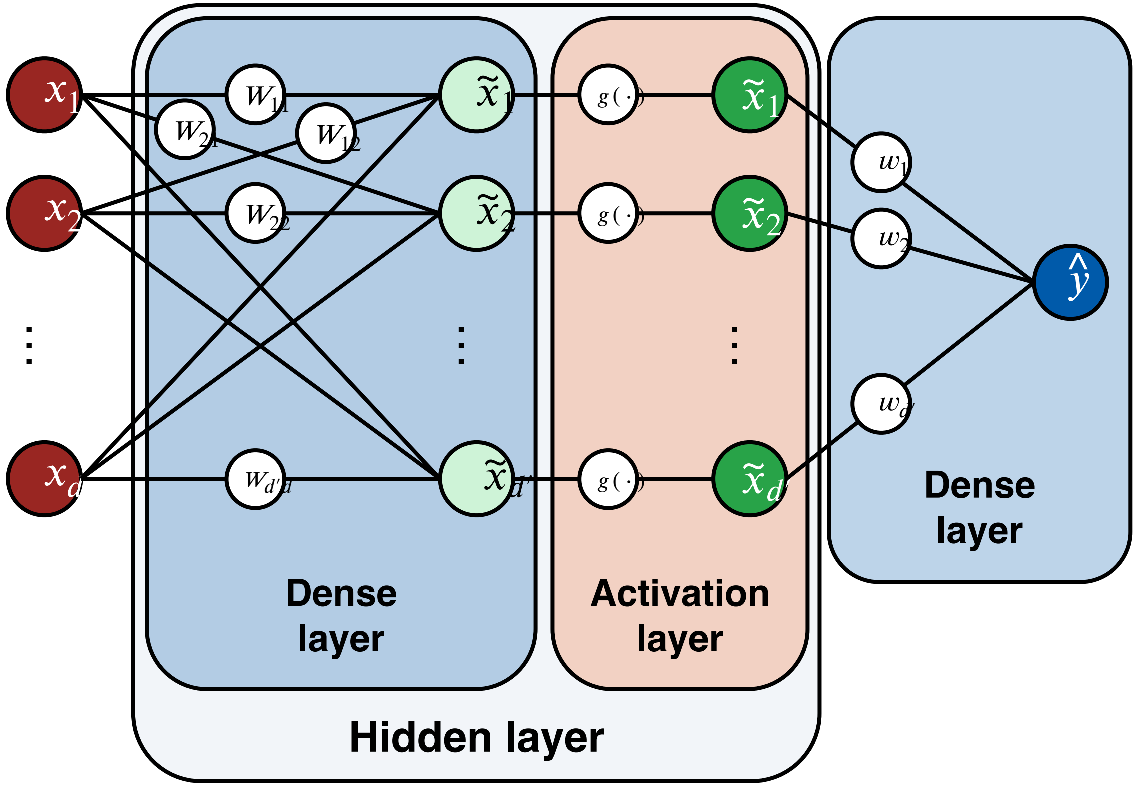
$$\mathrm{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

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

Universal approximator

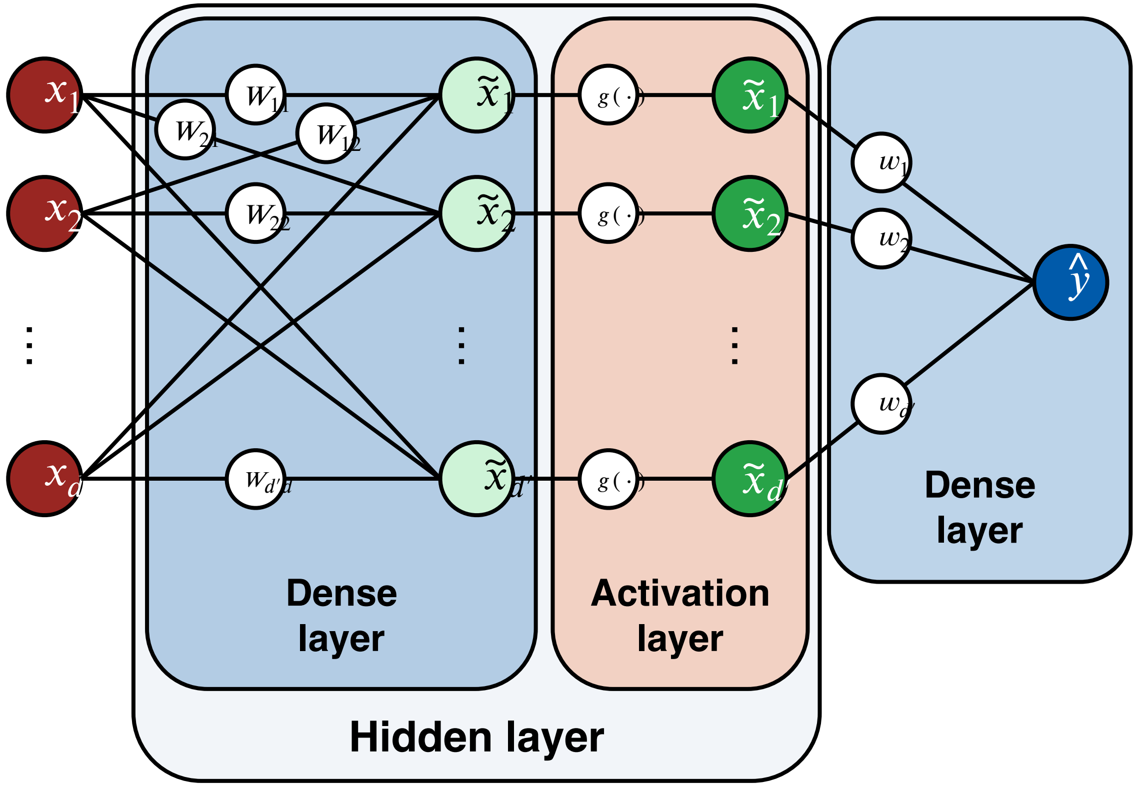
Universal approximator

- Just a single hidden layer with a nonlinearity makes this model a universal approximator



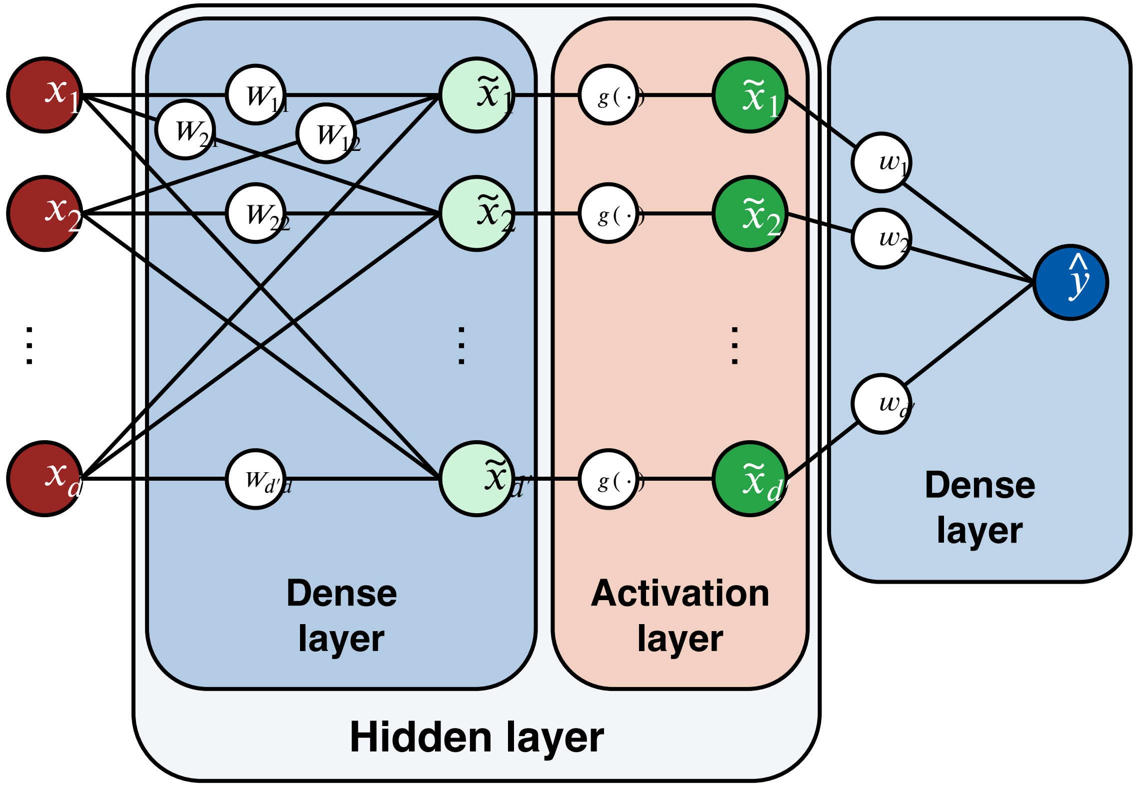
Universal approximator

- Just a single hidden layer with a nonlinearity makes this model a universal approximator
  - any function can be approximated **arbitrarily close** given wide enough hidden layer (large enough d')

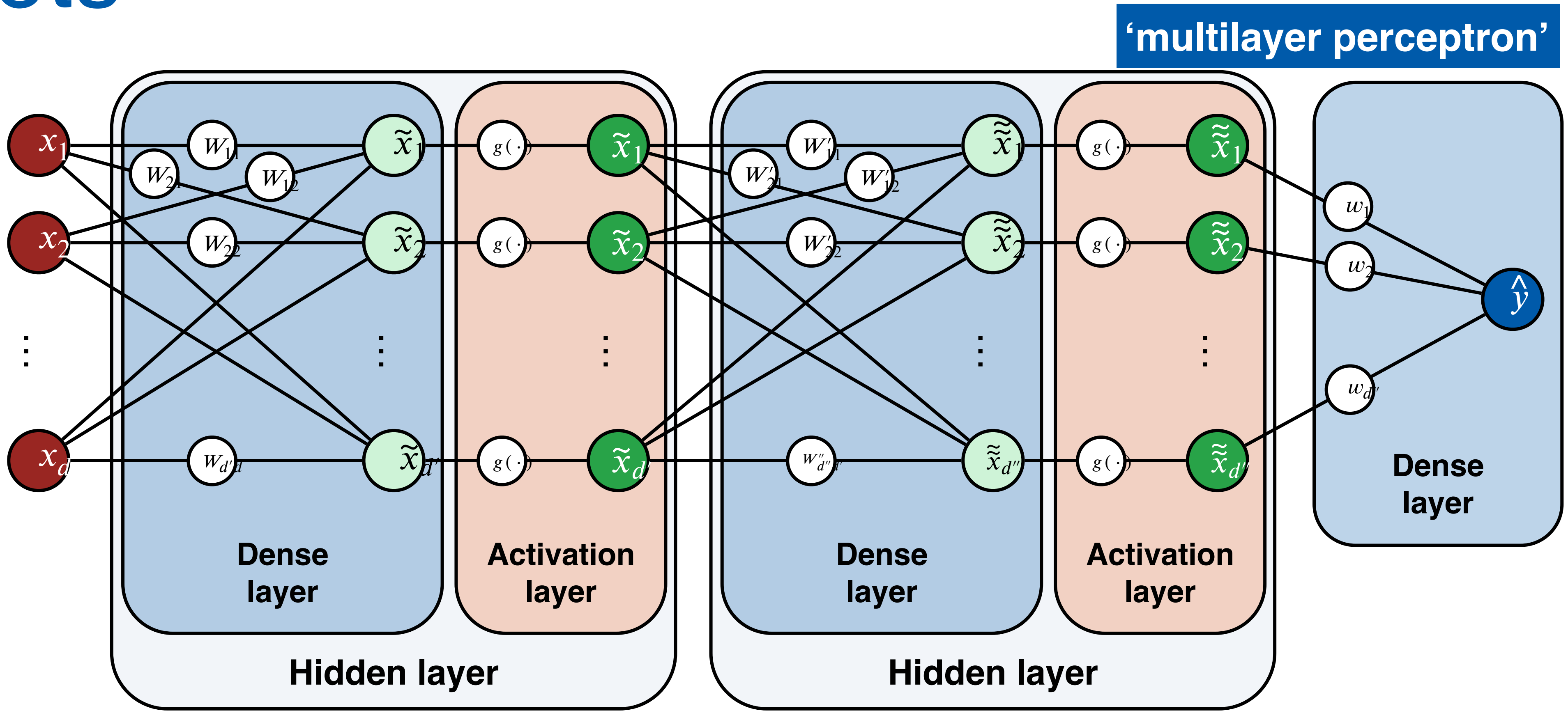


Universal approximator

- Just a single hidden layer with a nonlinearity makes this model a universal approximator
  - any function can be approximated **arbitrarily close** given wide enough hidden layer (large enough d')
  - Note: in practice we might not be able to find this approximation
    - e.g. due to heavily non-convex loss function, infeasibly large d', overfitting



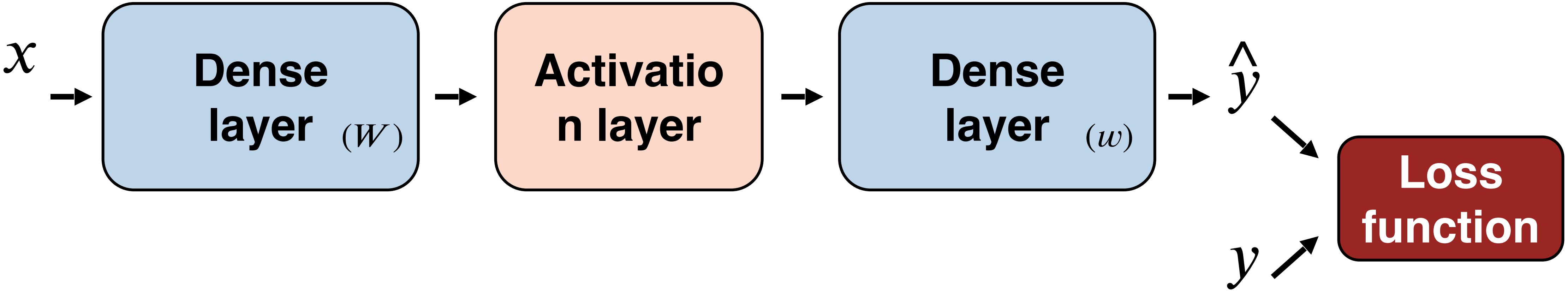
Deeper nets



- In practice, stacking more hidden layers often reduces the number of neurons required to represent a given function

Backpropagation

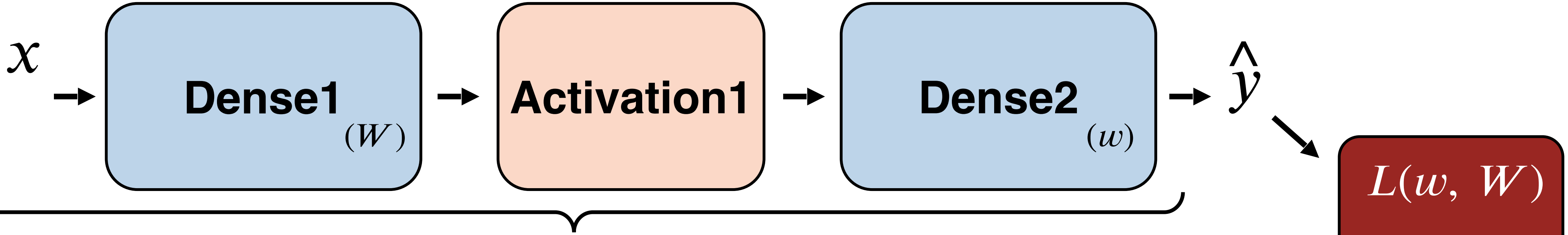
Loss function



- E.g. mean squared error:

$$L = \frac{1}{N} \sum_{i=1\ldots N} \left(y_i - w^{\mathrm{T}} g\left(W x_i \right) \right)^2$$

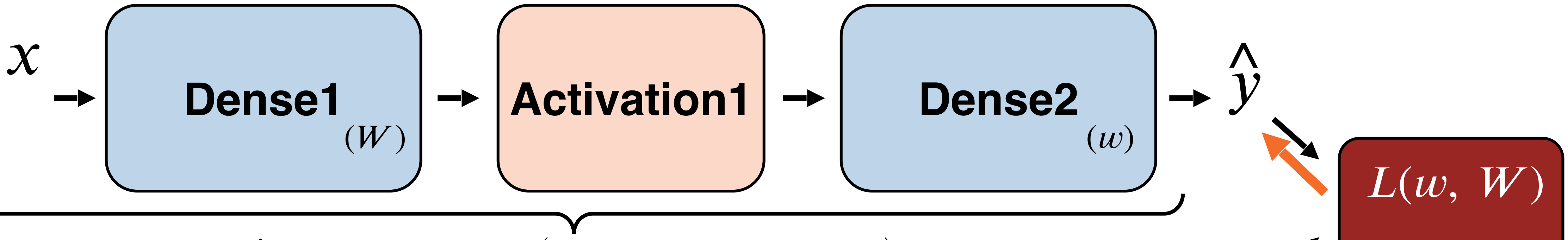
Calculating derivatives



$$\hat{f}(x; w, W) = \mathbf{Dense2}\Big(\mathbf{Activation1}\big(\mathbf{Dense1}(x)\big)\Big)$$

$$L(w,\ W) \equiv L\Big(y, \hat{f}(x; w, W)\Big)$$

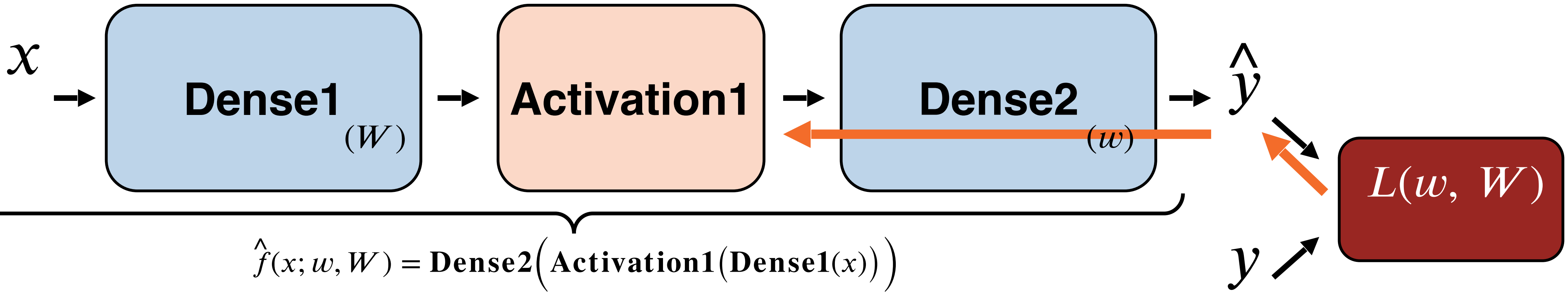
Calculating derivatives



$$\hat{f}(x; w, W) = \mathbf{Dense2}\Big(\mathbf{Activation1}\big(\mathbf{Dense1}(x)\big)\Big)$$

$$L(w,\ W) \equiv L\Big(y, \hat{f}(x; w, W)\Big)$$

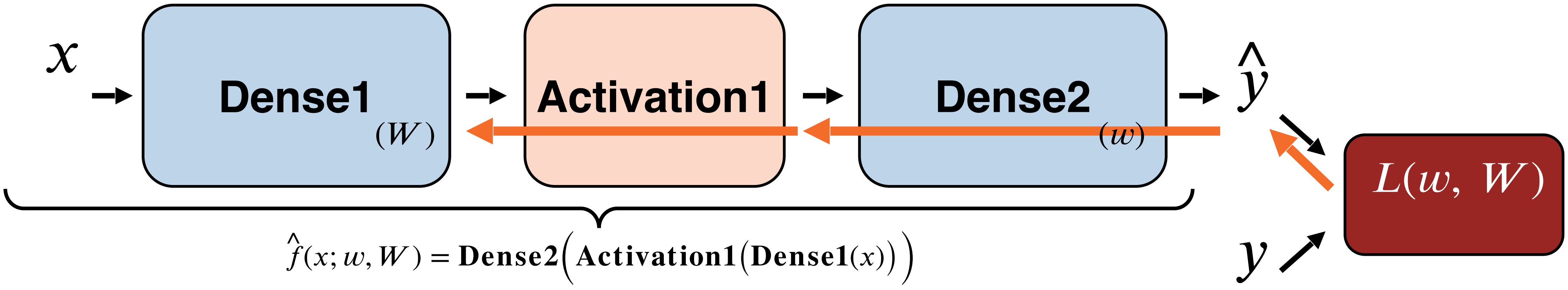
Calculating derivatives



$$\hat{f}(x; w, W) = \mathbf{Dense2}\Big(\mathbf{Activation1}\big(\mathbf{Dense1}(x)\big)\Big)$$

$$L(w,\ W) \equiv L\Big(y, \hat{f}(x; w, W)\Big)$$

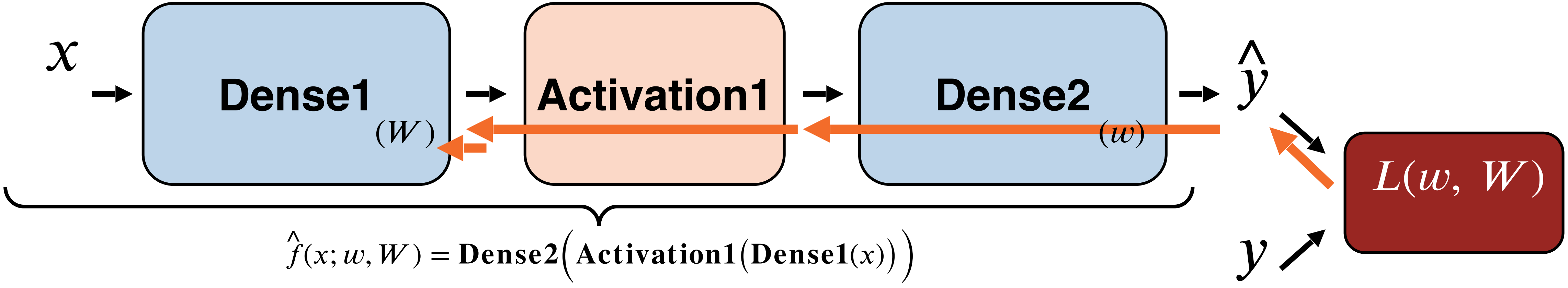
Calculating derivatives



$\hat{f}(x; w, W) = \mathbf{Dense2}\Big(\mathbf{Activation1}\big(\mathbf{Dense1}(x)\big)\Big)$

$$L(w,\ W) \equiv L\Big(y, \hat{f}(x; w, W)\Big)$$

Calculating derivatives

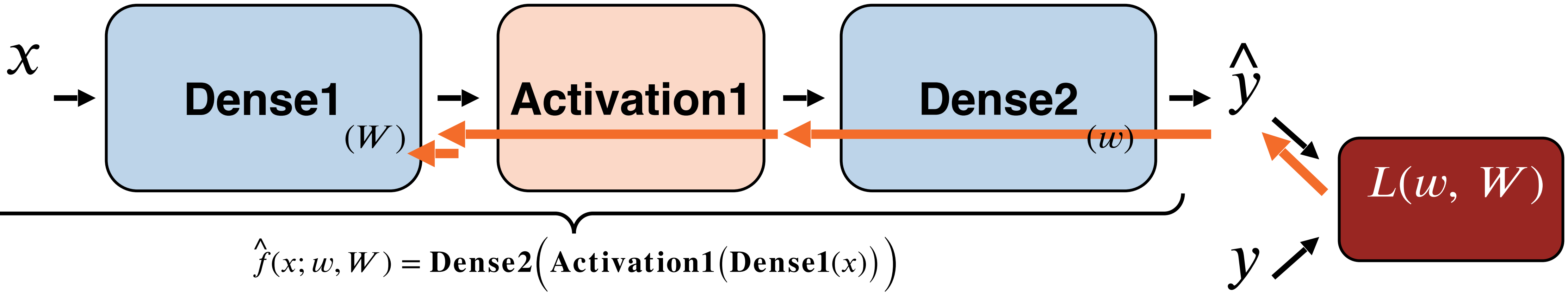


$$L(w,\ W) \equiv L\Big(y, \hat{f}(x; w, W)\Big)$$

$$\frac{\partial L}{\partial W} = \frac{\partial L\left(y, \hat{f}\right)}{\partial \hat{f}} \cdot \frac{\partial \hat{f}}{\partial W} = \frac{\partial L\left(y, \hat{f}\right)}{\partial \hat{f}} \cdot \frac{\partial \mathbf{Dense2}}{\partial \mathbf{Activation1}} \cdot \frac{\partial \mathbf{Activation1}}{\partial \mathbf{Dense1}} \cdot \frac{\partial \mathbf{Dense1}}{\partial W}$$

- Backpropagation algorithm ≈ applying the chain rule
  - The actual algorithm states how to do it efficiently

Calculating derivatives



$$\hat{f}(x; w, W) = \mathbf{Dense2}\Big(\mathbf{Activation1}\big(\mathbf{Dense1}(x)\big)\Big)$$

$$L(w,\ W) \equiv L\left(y, \hat{f}(x; w, W)\right)$$

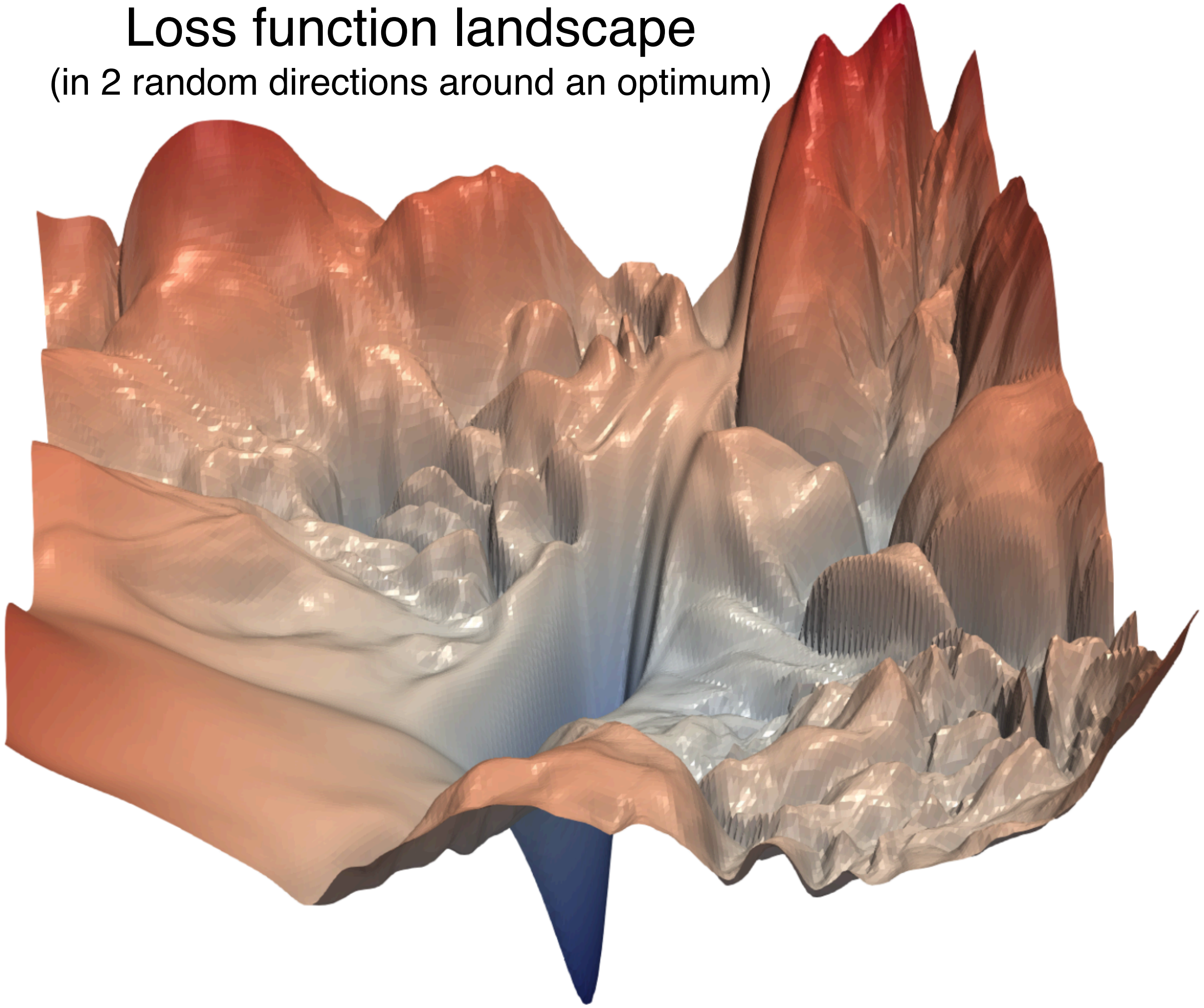
$$\frac{\partial L}{\partial W} = \frac{\partial L\left(y, \hat{f}\right)}{\partial \hat{f}} \cdot \frac{\partial \hat{f}}{\partial W} = \underbrace{\frac{\partial L\left(y, \hat{f}\right)}{\partial \hat{f}}}_{\text{scalar}} \cdot \underbrace{\frac{\partial \mathbf{Dense2}}{\partial \mathbf{Activation1}}}_{d'\text{-vector}} \cdot \underbrace{\frac{\partial \mathbf{Activation1}}{\partial \mathbf{Dense1}}}_{\substack{d' \times d'\text{-matrix} \\ \text{(diagonal)}}} \cdot \underbrace{\frac{\partial \mathbf{Dense1}}{\partial W}}_{d' \times d' \times d\text{-tensor} = d' \times d\text{-matrix}}$$

- Backpropagation algorithm ≈ applying the chain rule
  - The actual algorithm states how to do it efficiently

Optimization techniques

How to optimize such functions?

Loss function landscape
(in 2 random directions around an optimum)

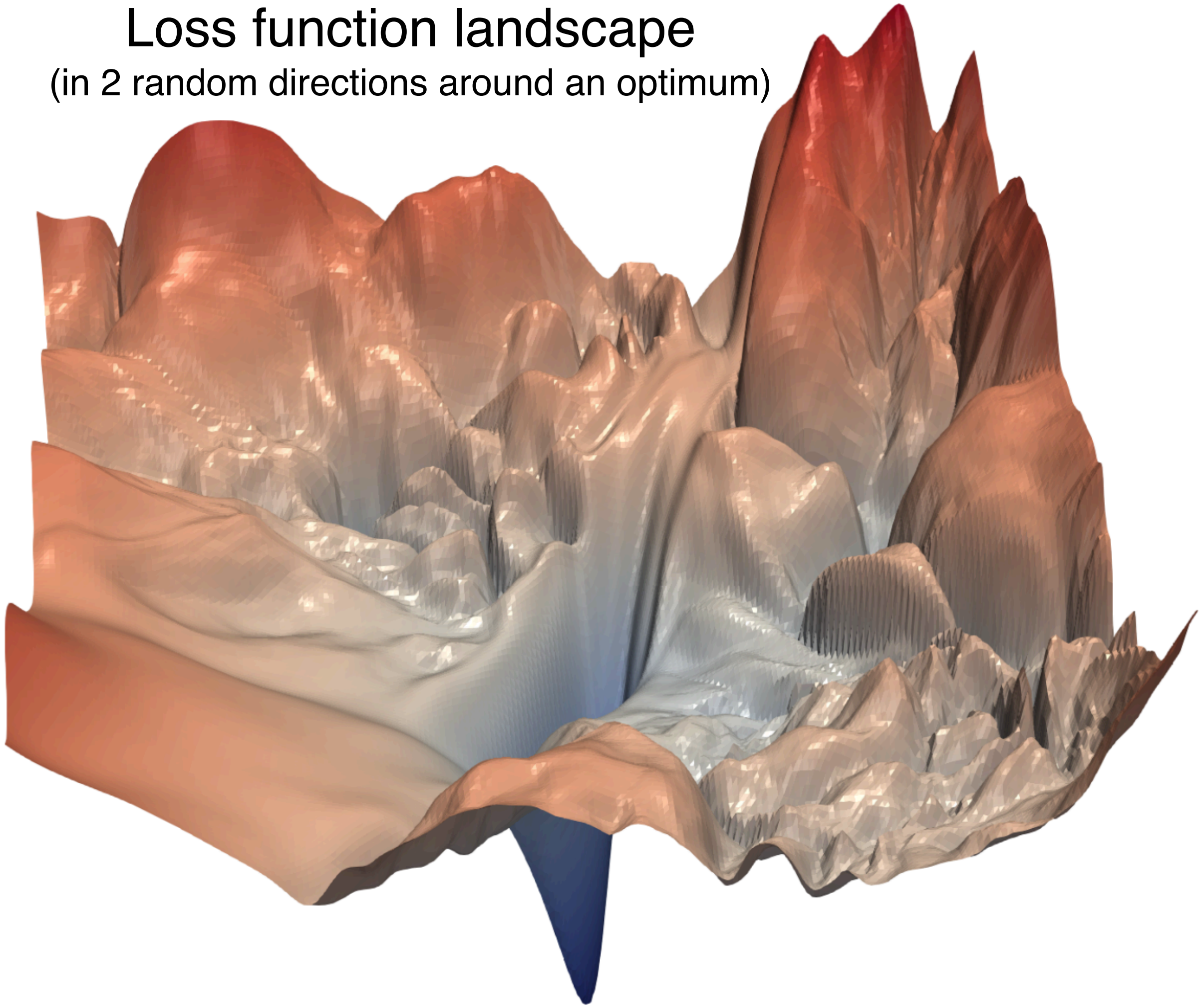


https://papers.nips.cc/paper/7875-visualizing-the-loss-landscape-of-neural-nets

- No convergence guarantees for the stochastic gradient descent

How to optimize such functions?

Loss function landscape
(in 2 random directions around an optimum)

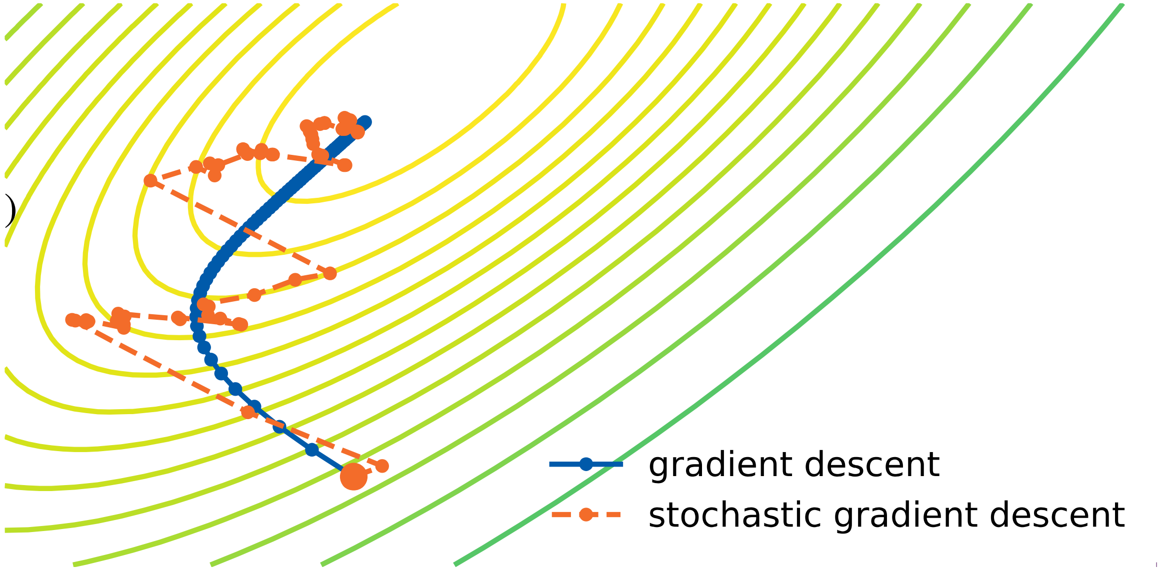


https://papers.nips.cc/paper/7875-visualizing-the-loss-landscape-of-neural-nets

- No convergence guarantees for the stochastic gradient descent
- There's a number of modifications to improve training

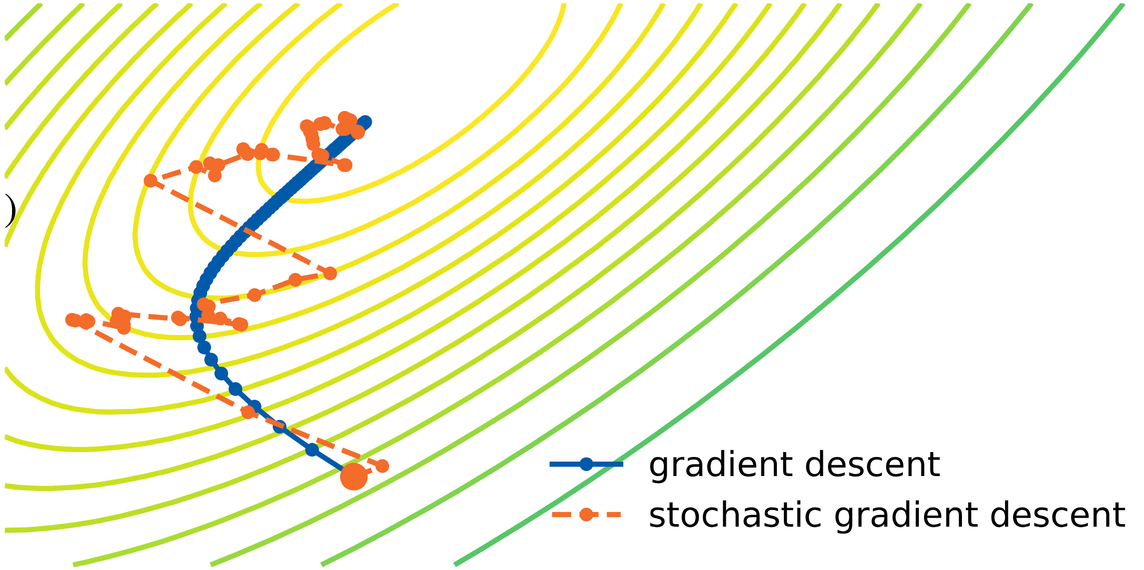
SGD on mini-batches

- SGD:
  - At each step k pick $l_k \in \{1, \ldots, N\}$ at random, then update:
  - $\theta^{(k)} \longleftarrow \theta^{(k-1)} - \eta \nabla_\theta \mathscr{L}\left(y_{l_k}, \hat{f}_\theta\left(x_{l_k}\right)\right)\Big|\theta = \theta^{(k-1)}$



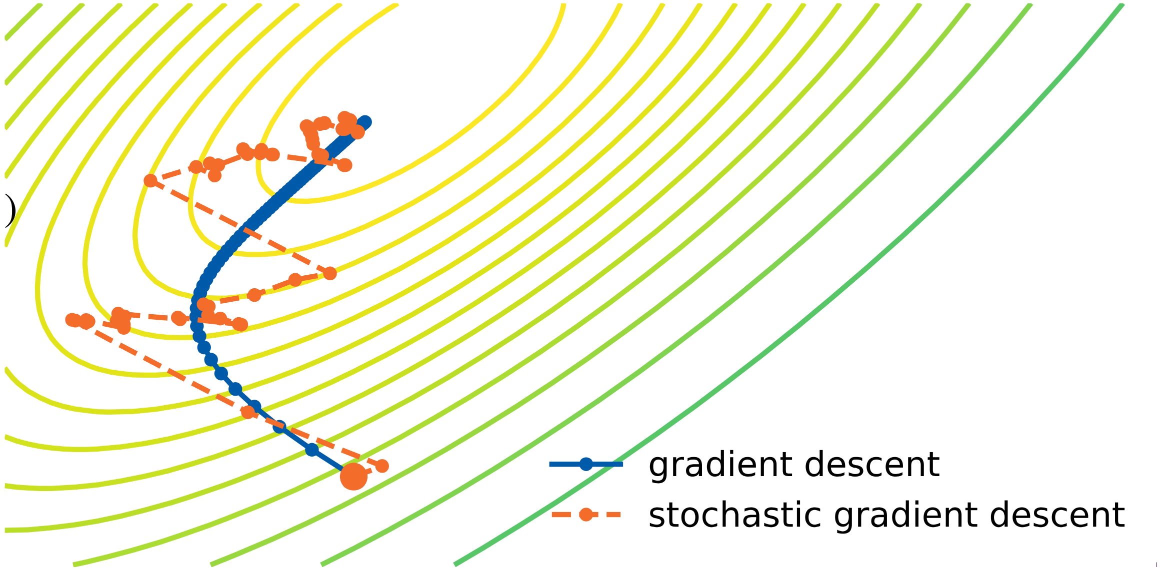
SGD on mini-batches

- SGD:
  - At each step k pick $l_k \in \{1, \ldots, N\}$ at random, then update:
  - $\theta^{(k)} \longleftarrow \theta^{(k-1)} - \eta \nabla_\theta \mathscr{L}\left(y_{l_k}, \hat{f}_\theta\left(x_{l_k}\right)\right)\Big|\theta = \theta^{(k-1)}$
- Mini-batch SGD:
  - Shuffle the training set, then iterate through it in chunks (batches) of fixed size



SGD on mini-batches

- SGD:
  - At each step k pick $l_k \in \{1, \ldots, N\}$ at random, then update:
  - $\theta^{(k)} \longleftarrow \theta^{(k-1)} - \eta \nabla_\theta \mathscr{L}\left(y_{l_k}, \hat{f}_\theta\left(x_{l_k}\right)\right)\Big|\theta = \theta^{(k-1)}$

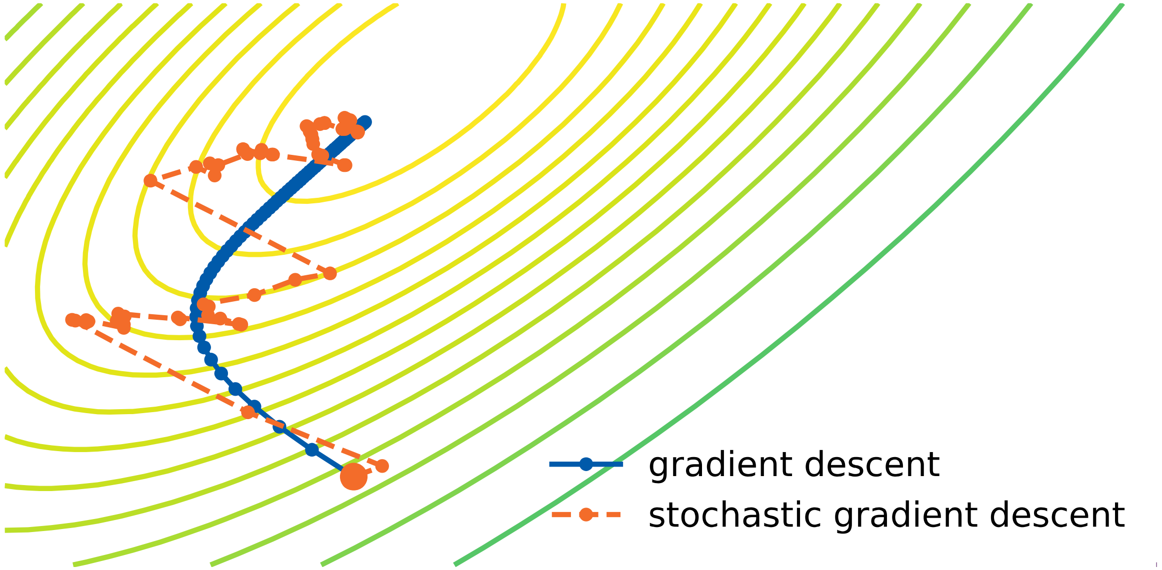


- Mini-batch SGD:
  - Shuffle the training set, then iterate through it in chunks (batches) of fixed size
  - At each iteration evaluate the loss gradients on the given chunk B:

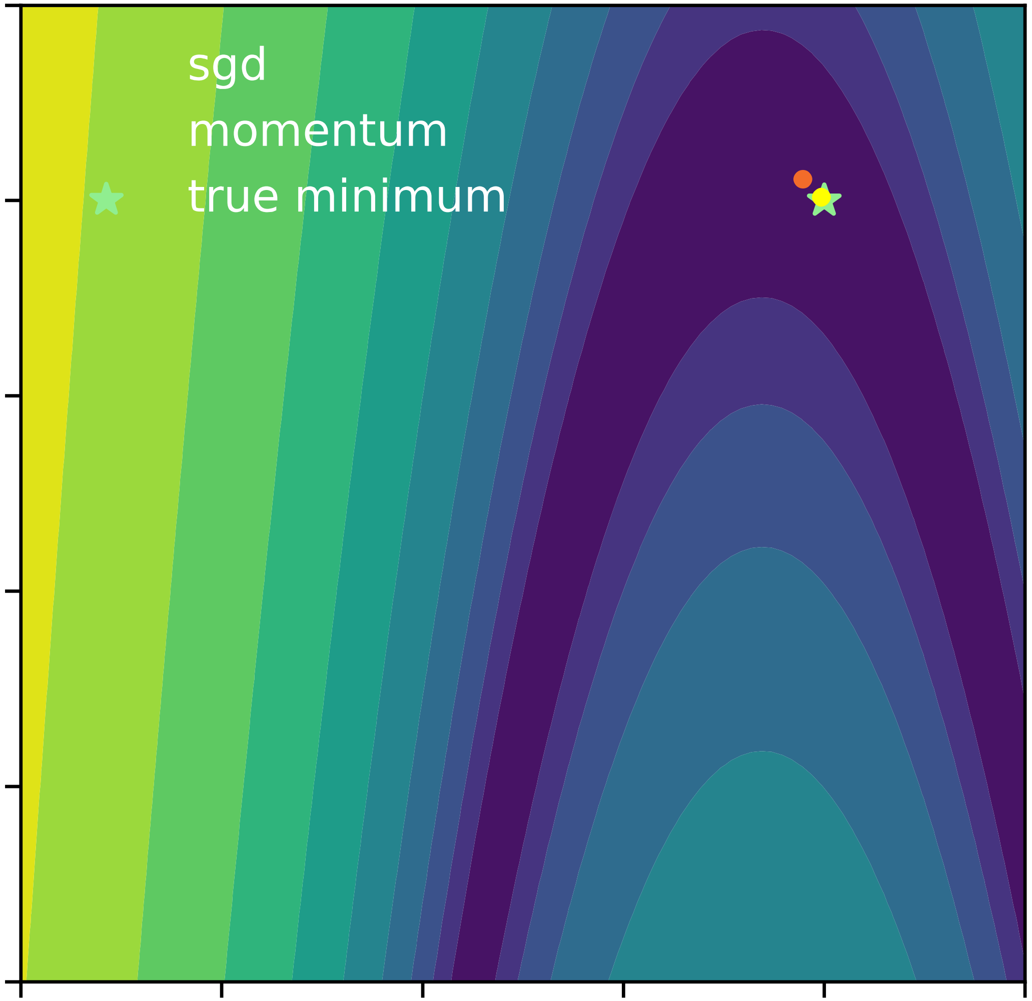
$$g = \sum_{i \in B} \nabla_\theta \mathscr{L}\left(y_i, \hat{f}_\theta\left(x_i\right)\right)$$

SGD on mini-batches

- SGD:
  - At each step k pick $l_k \in \{1, \ldots, N\}$ at random, then update:
  - $\theta^{(k)} \longleftarrow \theta^{(k-1)} - \eta \nabla_\theta \mathscr{L}\left(y_{l_k}, \hat{f}_\theta\left(x_{l_k}\right)\right)\Big|\theta = \theta^{(k-1)}$
- Mini-batch SGD:
  - Shuffle the training set, then iterate through it in chunks (batches) of fixed size
  - At each iteration evaluate the loss gradients on the given chunk B: $g = \sum_{i \in B} \nabla_\theta \mathscr{L}\left(y_i, \hat{f}_\theta(x_i)\right)$
  - Update the model parameters: $\theta^{(k)} \longleftarrow \theta^{(k-1)} - \eta \cdot g$



Momentum SGD

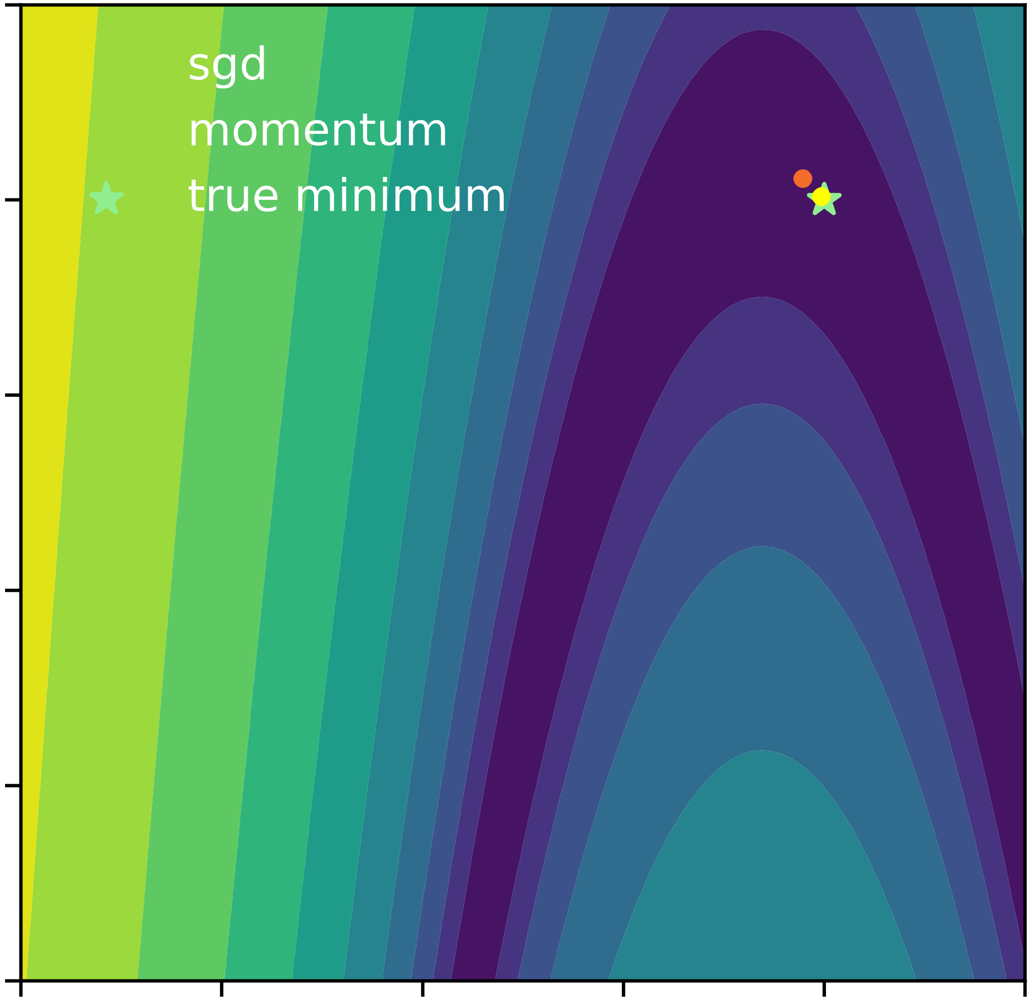


- Idea: introduce inertia (like a ball rolling down a hill)

$$m^{(k)} \longleftarrow \beta \cdot m^{(k-1)} + \left(1 - \beta\right) \cdot \left.\frac{\partial L}{\partial \theta}\right|_{\theta=\theta^{(k-1)}}$$

$$\theta^{(k)} \longleftarrow \theta^{(k-1)} - \eta \cdot m^{(k)}$$

Momentum SGD

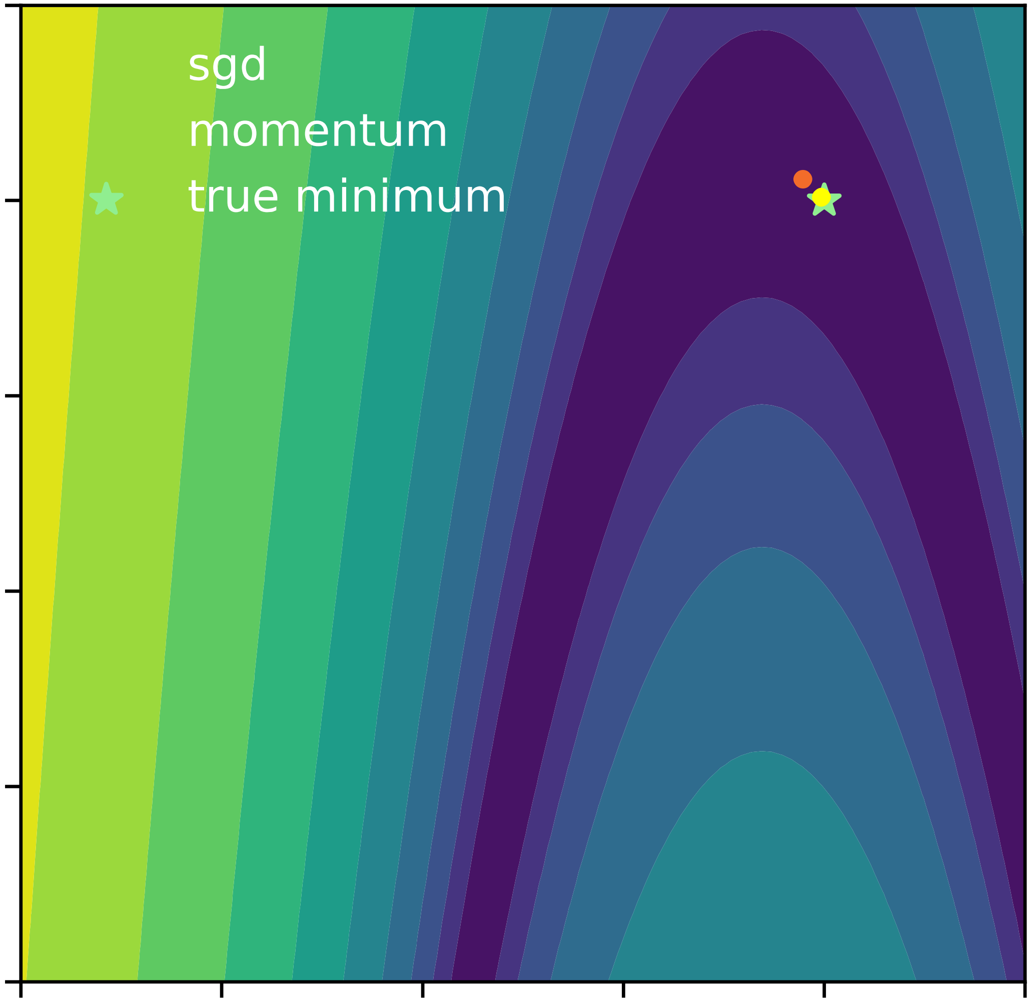


- Idea: introduce inertia (like a ball rolling down a hill)
  - Smooths out fast oscillations

$$m^{(k)} \longleftarrow \beta \cdot m^{(k-1)} + \left(1 - \beta\right) \cdot \left.\frac{\partial L}{\partial \theta}\right|_{\theta=\theta^{(k-1)}}$$

$$\theta^{(k)} \longleftarrow \theta^{(k-1)} - \eta \cdot m^{(k)}$$

Momentum SGD

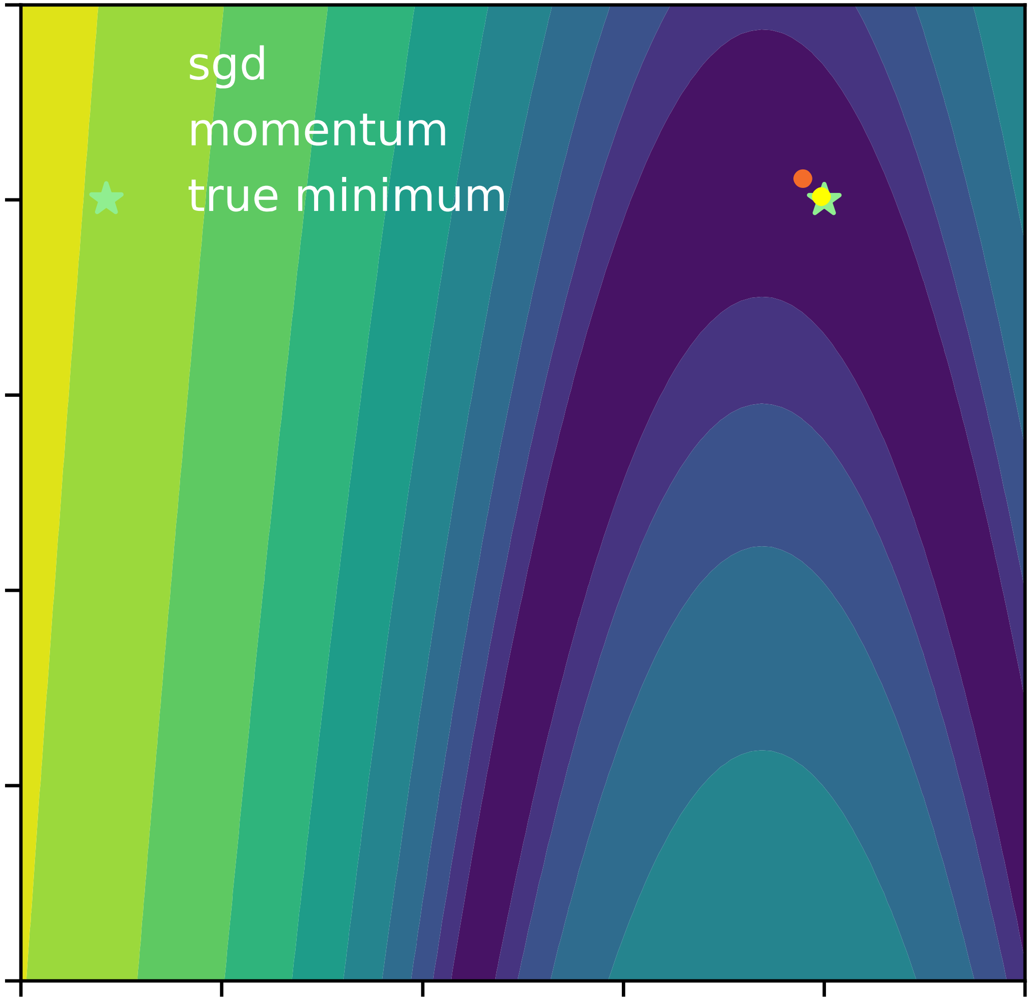


- Idea: introduce inertia (like a ball rolling down a hill)
  - Smooths out fast oscillations
  - Helps getting out of small local minima

$$m^{(k)} \longleftarrow \beta \cdot m^{(k-1)} + \left(1 - \beta\right) \cdot \left.\frac{\partial L}{\partial \theta}\right|_{\theta=\theta^{(k-1)}}$$

$$\theta^{(k)} \longleftarrow \theta^{(k-1)} - \eta \cdot m^{(k)}$$

Momentum SGD



- Idea: introduce inertia (like a ball rolling down a hill)
  - Smooths out fast oscillations
  - Helps getting out of small local minima
  - Allows for larger range of learning rates*

$$m^{(k)} \longleftarrow \beta \cdot m^{(k-1)} + \left(1 - \beta\right) \cdot \left.\frac{\partial L}{\partial \theta}\right|_{\theta=\theta^{(k-1)}}$$

$$\theta^{(k)} \longleftarrow \theta^{(k-1)} - \eta \cdot m^{(k)}$$

* https://distill.pub/2017/momentum/

RMSprop

- Idea: adjust learning rate separately for different components of the parameter vector
  – Gradients getting smaller $\Rightarrow$ increase the learning rate (scale by inverse running RMS of the gradient)

RMSprop

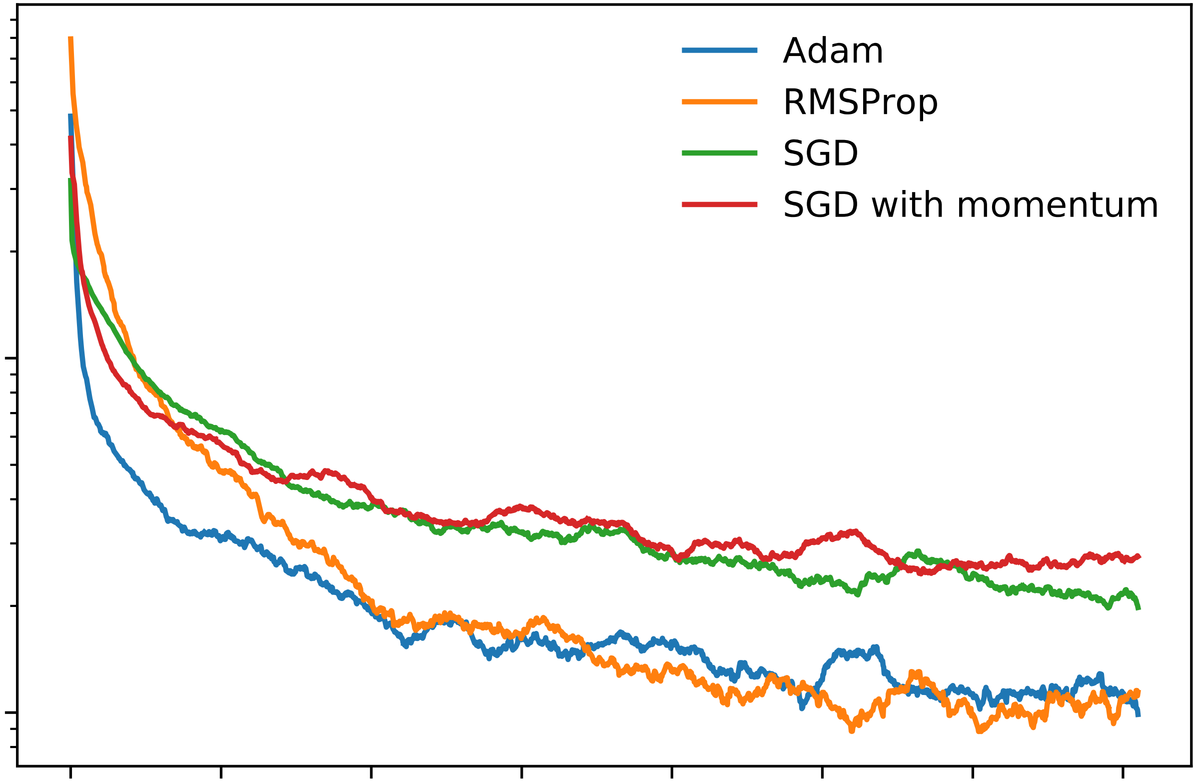
- Idea: adjust learning rate separately for different components of the parameter vector
  - Gradients getting smaller ⇒ increase the learning rate (scale by inverse running RMS of the gradient)

$$\mathbb{E}\left[g^2\right]_{(k)} \longleftarrow \beta \cdot \mathbb{E}\left[g^2\right]_{(k-1)} + (1-\beta) \cdot \left(\frac{\partial L}{\partial \theta}\right)^2 \Bigg|_{\theta=\theta^{(k-1)}}$$

$$\theta^{(k)} \longleftarrow \theta^{(k-1)} - \frac{\eta}{\sqrt{\mathbb{E}\left[g^2\right]_{(k)} + \varepsilon}} \cdot \frac{\partial L}{\partial \theta}\Bigg|_{\theta=\theta^{(k-1)}}$$

Adam

- Combine both ideas (momentum + RMSprop)
- Typically a good first choice for an optimizing algorithm



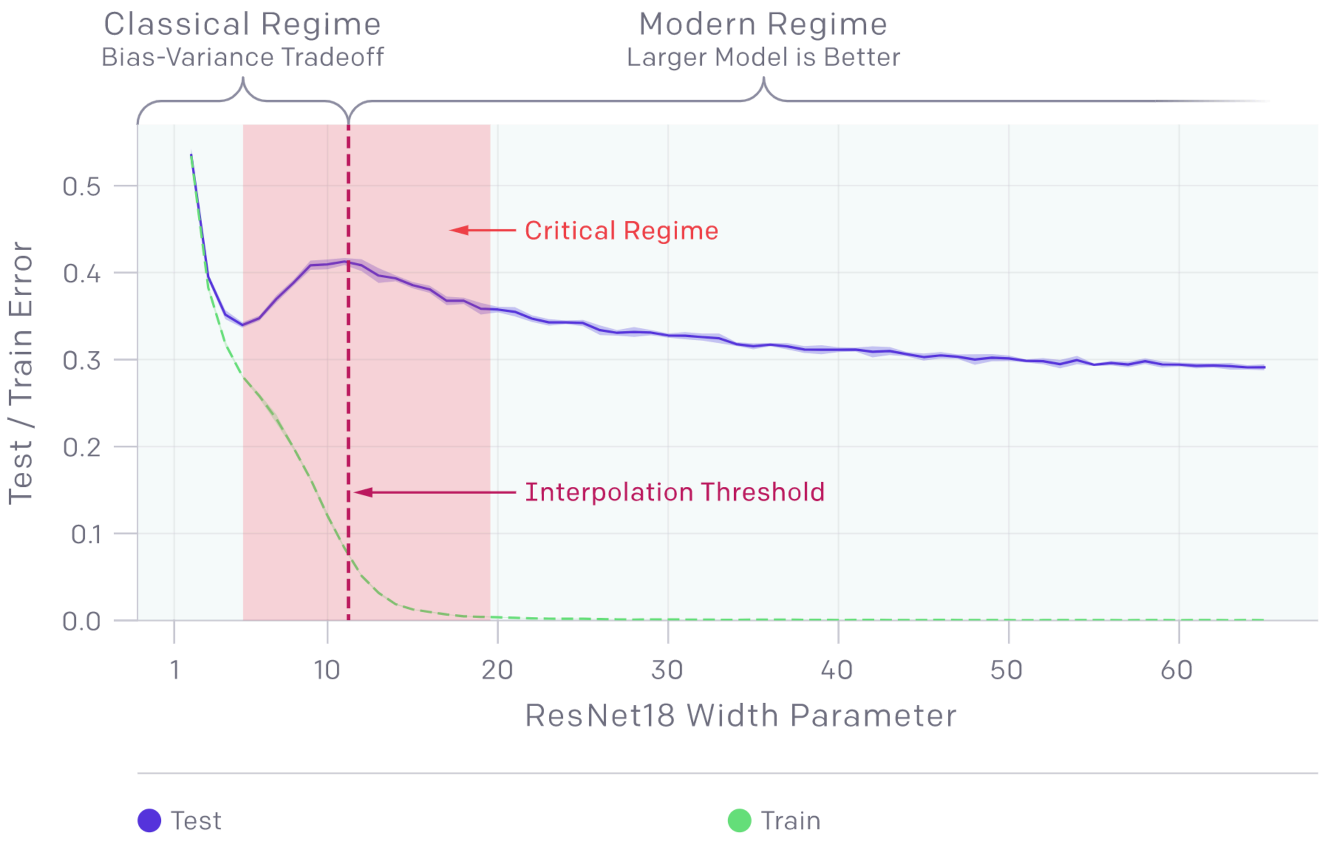
NN generalization

Why deep neural nets generalize well?

- Number of parameters is often well above the size of the training dataset
- Would expect heavy overfitting according to “classical ML” theory
- In practice, test error often decreases with the size of the model

Deep Double Descent

Img source: https://openai.com/blog/deep-double-descent/



- In fact, the dependence of the test error from the model size is more complicated
- Often, the effect of **double descent** is observed
- Not understood well
  - See this review
  - Probably, cannot be explained by the implicit regularization from the optimization technique (see, e.g., 2109.14119, 2104.14421)
  - Moreover: happens in simpler models, like linear regression (2109.02355)

Summary

- Neural networks are essentially **stacked linear models** with scalar nonlinearities in between

Summary

- Neural networks are essentially **stacked linear models** with scalar nonlinearities in between
- Earlier layers extract useful features s.t. the problem **becomes solvable** with a linear model (the last layer)

Summary

- Neural networks are essentially **stacked linear models** with scalar nonlinearities in between
- Earlier layers extract useful features s.t. the problem **becomes solvable** with a linear model (the last layer)
- Neural networks can approximate any function arbitrarily well, given they are deep/wide enough

Summary

- Neural networks are essentially **stacked linear models** with scalar nonlinearities in between
- Earlier layers extract useful features s.t. the problem **becomes solvable** with a linear model (the last layer)
- Neural networks can approximate any function arbitrarily well, given they are deep/wide enough
- Loss functions typically become highly **non-convex** for neural networks
  - this makes the optimization process harder

Summary

- Neural networks are essentially **stacked linear models** with scalar nonlinearities in between
- Earlier layers extract useful features s.t. the problem **becomes solvable** with a linear model (the last layer)
- Neural networks can approximate any function arbitrarily well, given they are deep/wide enough
- Loss functions typically become highly **non-convex** for neural networks
  - this makes the optimization process harder
- A variety of **SGD modifications** are available to mitigate this problem

Summary

- Neural networks are essentially **stacked linear models** with scalar nonlinearities in between
- Earlier layers extract useful features s.t. the problem **becomes solvable** with a linear model (the last layer)
- Neural networks can approximate any function arbitrarily well, given they are deep/wide enough
- Loss functions typically become highly **non-convex** for neural networks
  - this makes the optimization process harder
- A variety of **SGD modifications** are available to mitigate this problem
- Food for thought: being the ’universal approximators’, can neural nets really solve every possible supervised learning problem?

Thank you!

amaevskij@hse.ru

SiLiKhon

Artem Maevskiy