

# Neural Networks

Olga Razuvaeva<sup>1, 2</sup>, Sergey Korpachev<sup>3, 4</sup>, Stepan Zakharov<sup>5</sup>

<sup>1</sup>Institute for Theoretical and Experimental Physics

<sup>2</sup>National Research Nuclear University MEPhI (Moscow Engineering Physics Institute)

<sup>3</sup>Moscow Institute of Physics and Technology

<sup>4</sup>Lebedev Physical Institute of the Russian Academy of Sciences

<sup>5</sup>Novosibirsk State University



# Outline

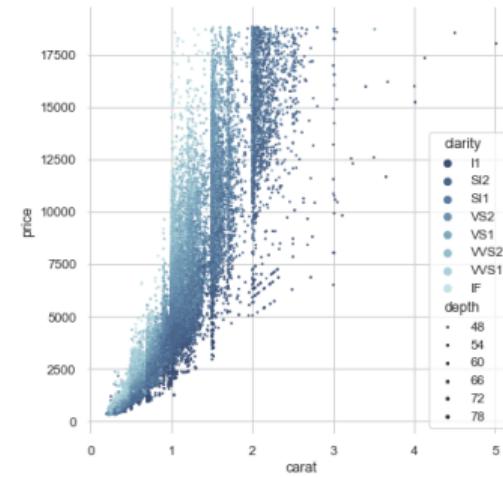
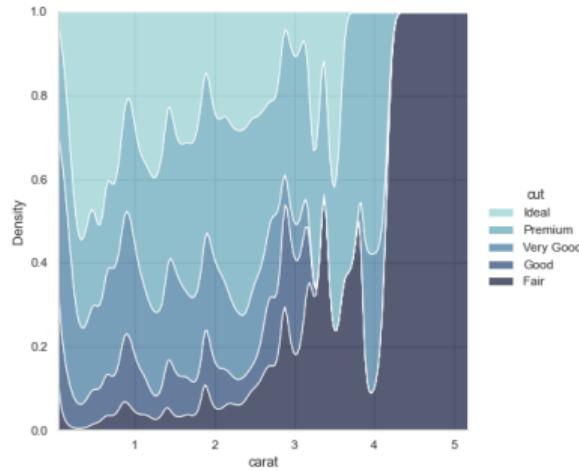
- Modeling nonlinearities
- Neural Network overview
- Training
- Deep neural networks
- Tackling overfitting

# Modelling nonlinearities

# Modelling nonlinearities

—○ they exist

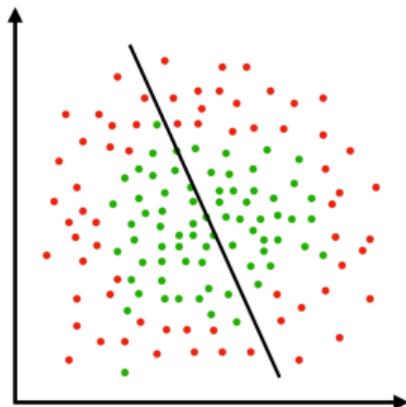
seaborn illustrations



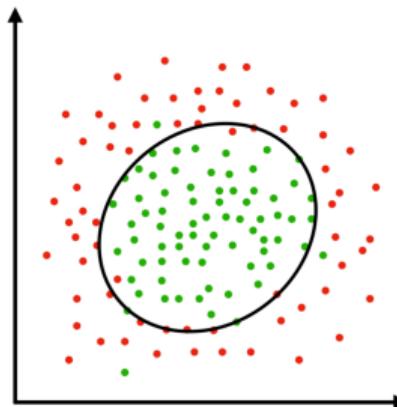
# Modelling nonlinearities

linear models

What we have



What we want

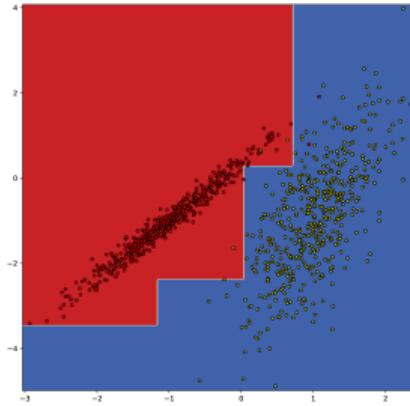
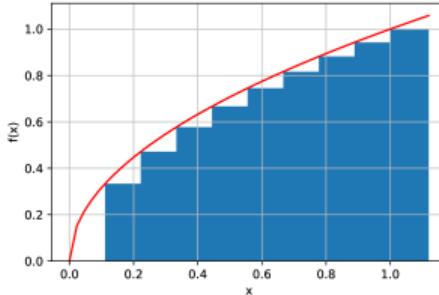


Linear models can't simply describe complex nonlinear data

# Modelling nonlinearities

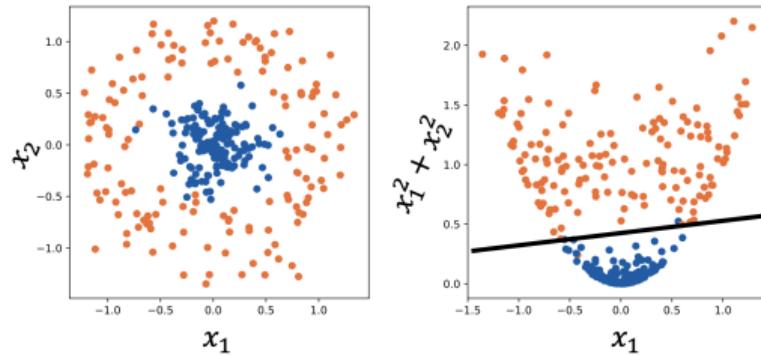
—○ trees

- (Ensembles of) Trees were designed to approximate nonlinearities and are **pretty good** in it + they are **fast and interpretable**
- But they are just “brute-force” algorithms – **don’t infer symmetries** in data by design
- **Ad-hoc, cut-based and piecewise approximations** of data at hand + not differentiable and smooth



# Modelling nonlinearities

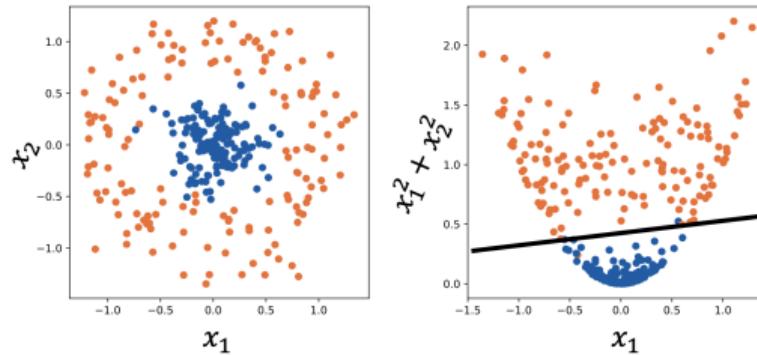
—○ feature engineering



- But sometimes we know *a priori* that there are transformations simplifying the problem → even linear model can do the job
- However, this **feature engineering** is non-trivial, requires domain knowledge and is time-consuming

# Modelling nonlinearities

—○ feature engineering



- But sometimes we know *a priori* that there are transformations simplifying the problem → even linear model can do the job
- However, this **feature engineering** is non-trivial, requires domain knowledge and is time-consuming

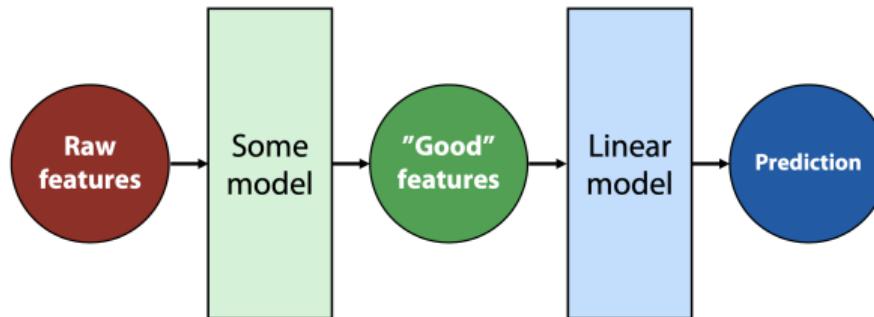
What if we design a model which could **automatically** feature-engineer itself?

# Neural Network

# Neural Network

—○ automating FE

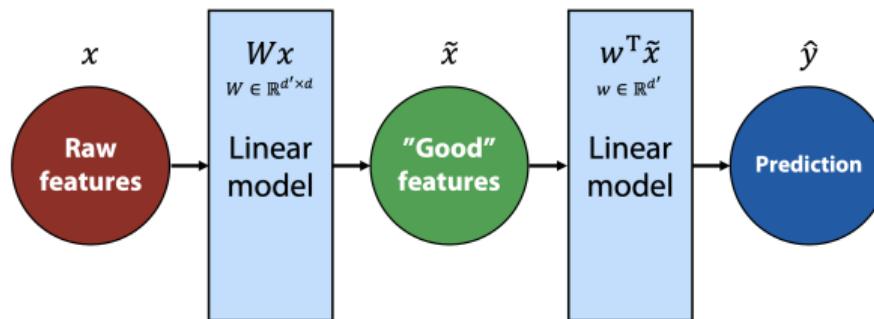
NN illustrations from  
ML in HEP 2020



- Let's use a **simple linear model** to solve our supervised problem
- Add a block to a linear model which will automatically **generate new features** for it
- Two blocks would work together as a **single model** ⇒ their parameters are updated simultaneously
- And **automatically**, by e.g. gradient descent (given their differentiability)

# Neural Network

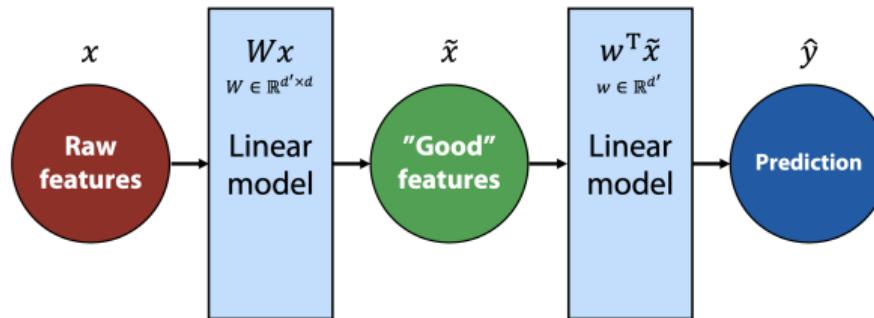
—○ automating FE



- Would a linear model work as a feature generating model?

# Neural Network

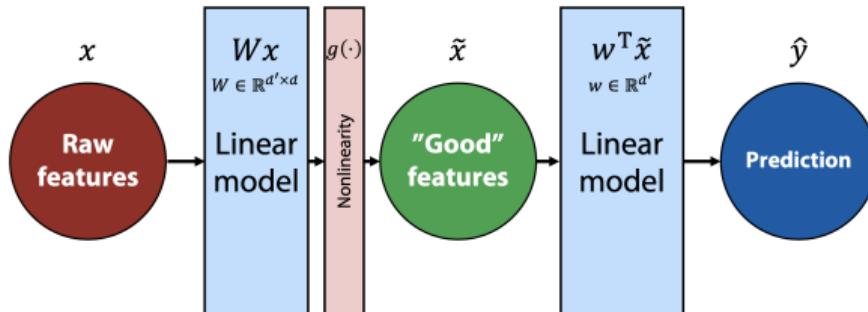
—○ automating FE



- Would a linear model work as a feature generating model? **No**
- $\hat{y} = w^T \tilde{x} = w^T (Wx) = (w^T W)x = w'^T x \Rightarrow$  it is still a linear model
- Input feature space has not changed, only the model weights

# Neural Network

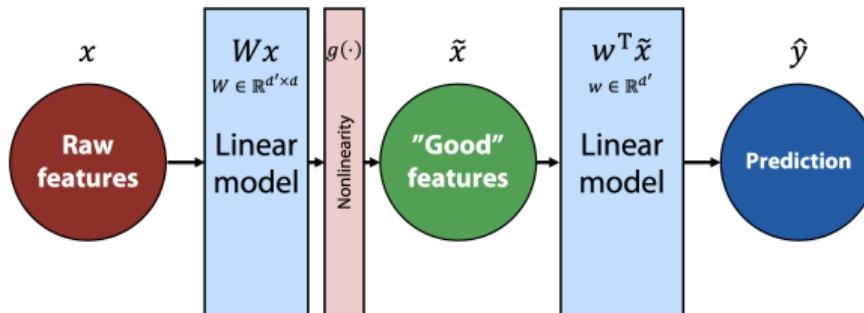
—○ automating FE



- Let's then introduce **nonlinearity** to our model  
→  $\hat{y} = w^T \tilde{x} = w^T g(Wx)$ ,  
where  $g(\cdot)$  – some nonlinear scalar function (applied elementwise)

# Neural Network

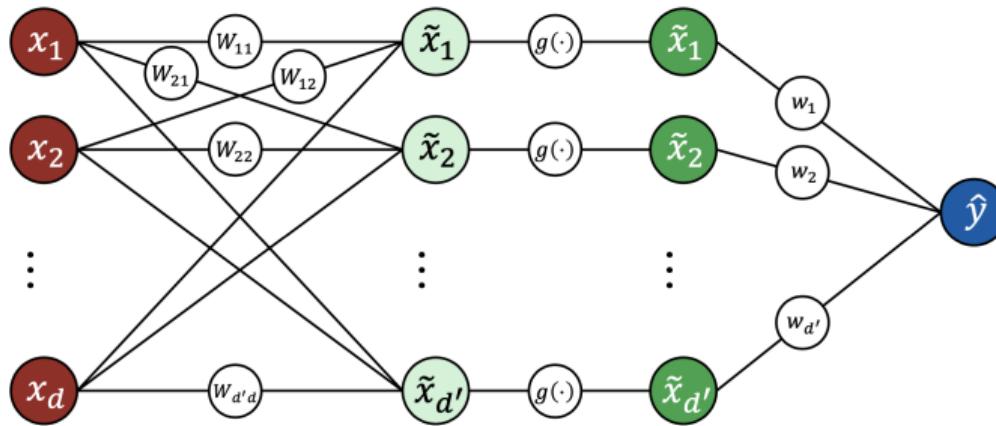
—○ automating FE



- Let's then introduce **nonlinearity** to our model
  - $\hat{y} = w^T \tilde{x} = w^T g(Wx)$ ,  
where  $g(\cdot)$  – some nonlinear scalar function (applied elementwise)
  - This is the simplest example of a **neural network**

# Neural Network

—○ architecture



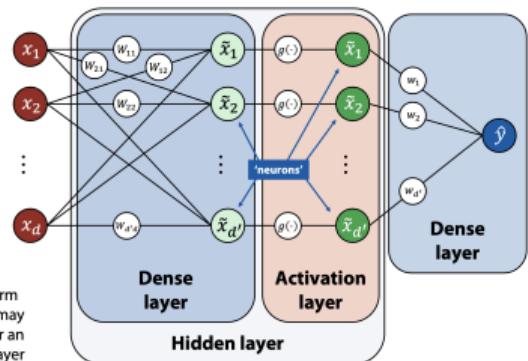
Essentially, NN is just a **composite function** that maps a set of  $X$  to a set of  $Y$

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

# Neural Network

 ——○ terminology

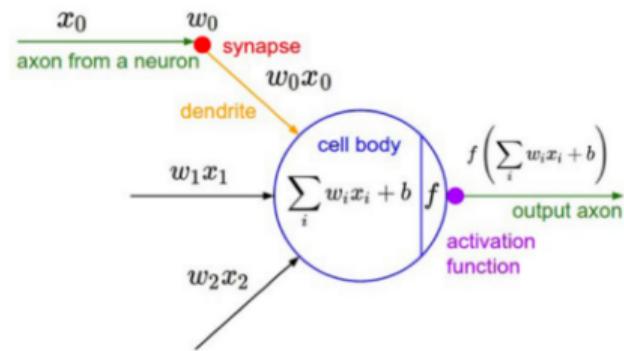
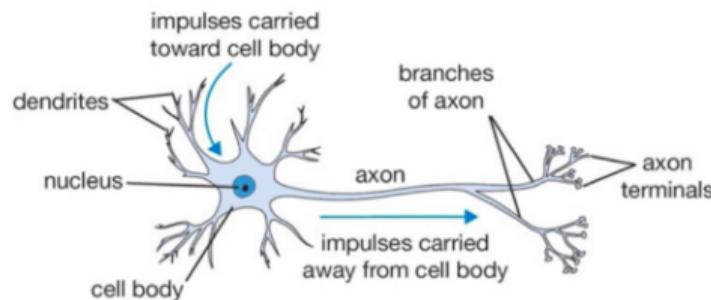
## Feed-forward network:



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

- Brown nodes  $x_1, x_2, \dots, x_d$  – features from an **input layer**
- Green nodes  $\tilde{x}_1, \tilde{x}_2, \dots, \tilde{x}_{d'}$  – **neurons** from a **hidden layer**
- Blue node  $\hat{y}$  – neuron from an **output layer**
- Straight lines (edges) between neurons – **weights**  $w_{ij}$
- $g(\cdot)$  – nonlinear **activation** function, e.g. sigmoid  $\sigma(\cdot)$
- **Important:** each neuron has additional **bias**  $b$  associated to it and added to other inputs (not illustrated)

# Neural Network —○ human brain



# Training

# Training —— o how to train?

- Since NN fundamentally is a parametrized differentiable model we can optimise the loss function and use **gradient descent** to train it:

$$\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \eta \cdot \nabla \mathcal{L}(\mathbf{w}_k)$$

- But **gradients** are hard to derive analytically – writing down all the derivatives is tough and tedious (especially for large NN)
- Note that NN is just a **composite model**  $\Rightarrow$  can use **chain rule** for differentiating it

# Recap



gradient descent

**Idea:** bring up the information about the direction to the minimum of  $Q$  using its **antigradient**

- Start with some vector  $\theta^0$

- $\hat{\theta} \leftarrow \theta^0$

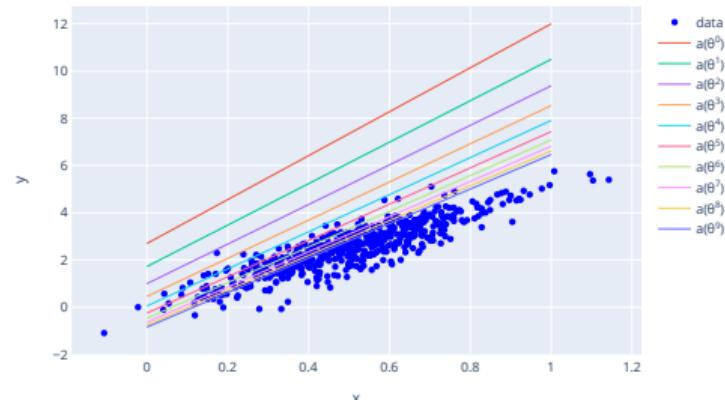
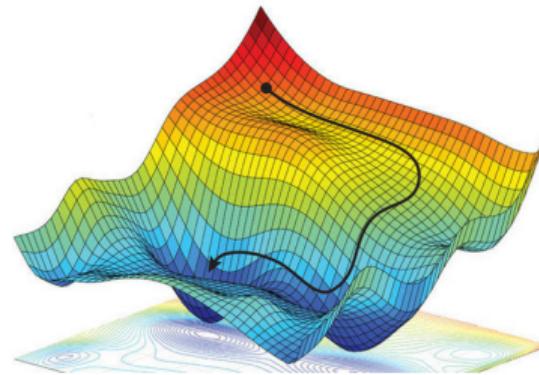
- Calculate direction to the minimum:

$$-\nabla \mathcal{L}(\hat{\theta}) = -\left[ \frac{\partial \mathcal{L}}{\partial \theta_0}, \frac{\partial \mathcal{L}}{\partial \theta_1} \right]_{\hat{\theta}}$$

- Move towards the minimum:

$$\hat{\theta} \leftarrow \hat{\theta} - \eta \nabla \mathcal{L}(\hat{\theta})$$

- Repeat until convergence



# Training —— o chain rule

- Computing derivative of a "base" function is simple  $\Rightarrow$  decompose composite function into a set of base ones and differentiate them one by one
- Let's recall the rule:

$$\frac{\partial f(t(x))}{\partial x} = \frac{\partial f(t)}{\partial t} \cdot \frac{\partial t(x)}{\partial x}$$

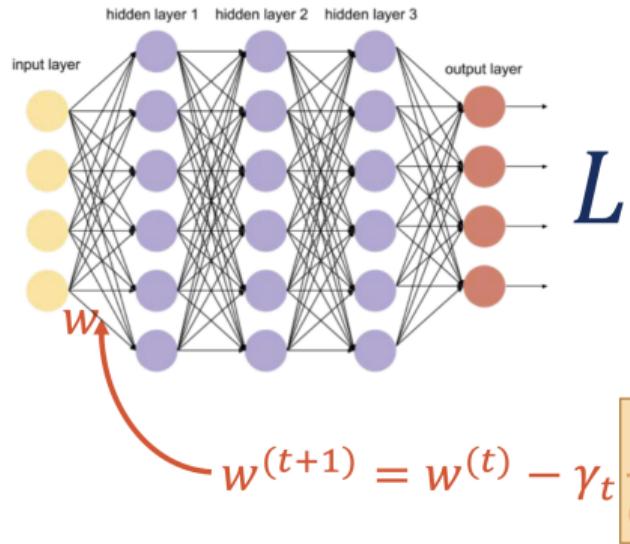
**Example:** Derivative of the sigmoid is simply expressed through the initial function:

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

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

# Training — o backpropagation

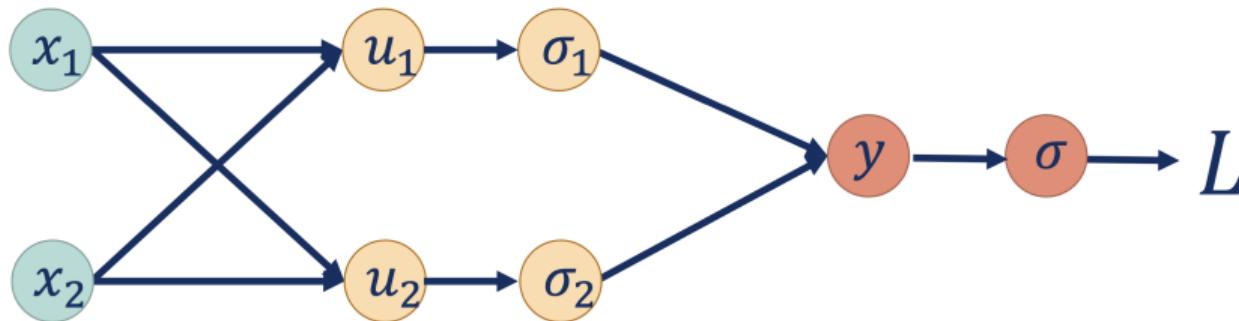
backprop illustrations from [DMIA](#)



- So how to find partial derivatives of loss function with respect to some weight?

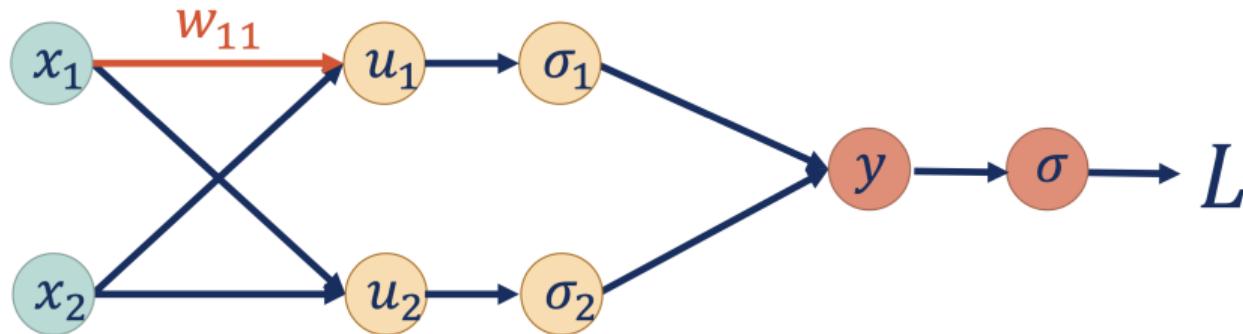
Training —○ backpropagation

backprop illustrations from [DMIA](#)



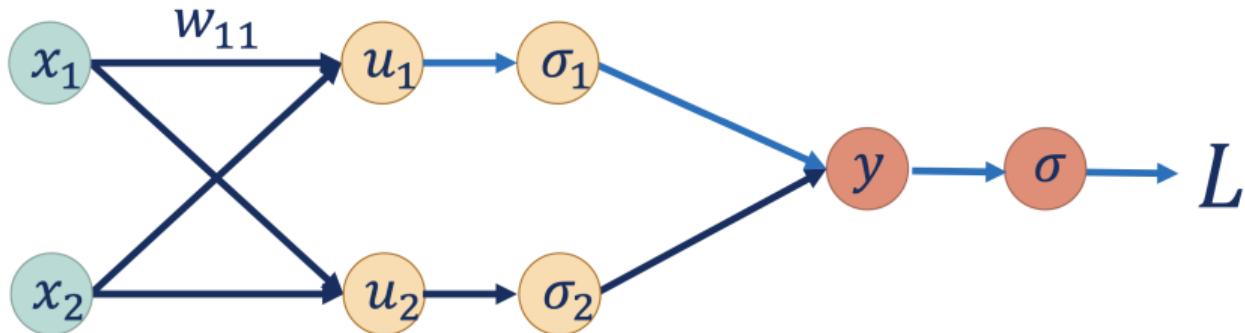
- Let's consider a simplified neural network
- Represent it in the form of a **computational graph**

Training —○ backpropagation



$$w_{11}^{(t+1)} = w_{11}^{(t)} - \gamma_t \frac{\partial L}{\partial w_{11}}$$

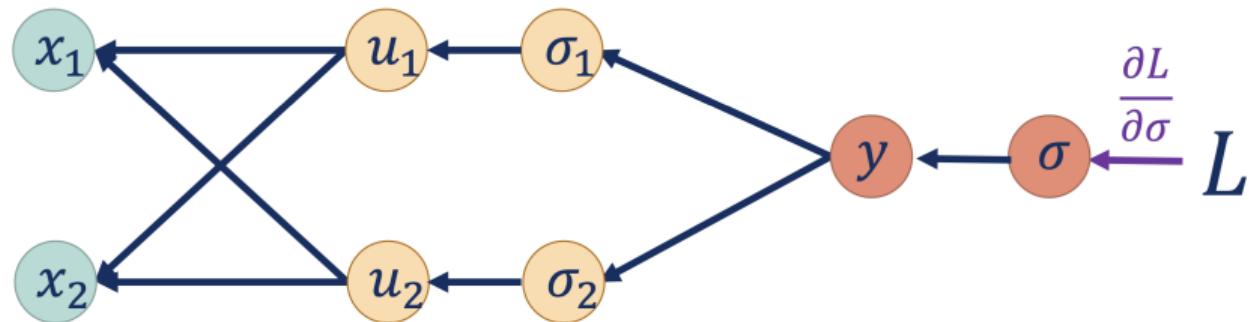
Training —○ backpropagation



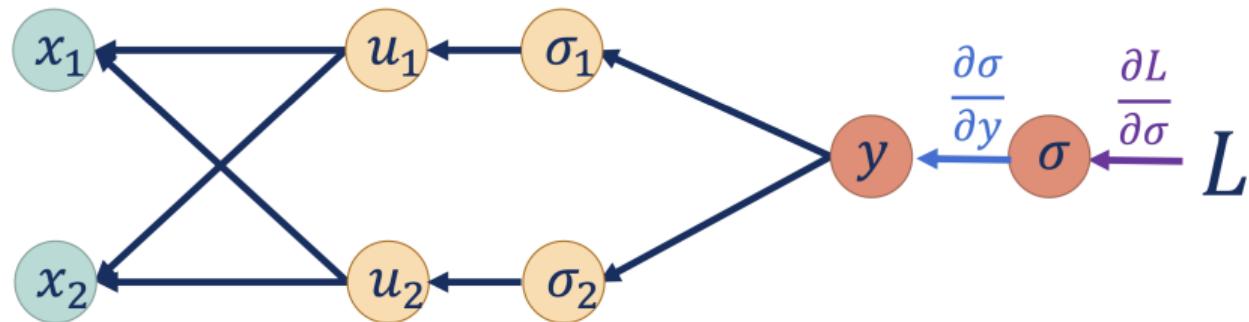
$$w_{11}^{(t+1)} = w_{11}^{(t)} - \gamma_t \frac{\partial L}{\partial w_{11}}$$

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial u_1} \frac{\partial u_1}{\partial w_{11}} = \frac{\partial L}{\partial u_1} x_1$$

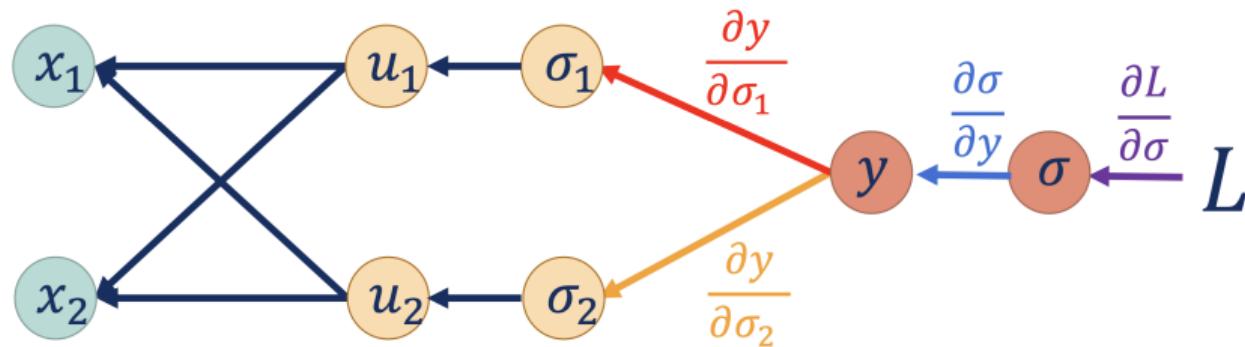
Training —○ backpropagation



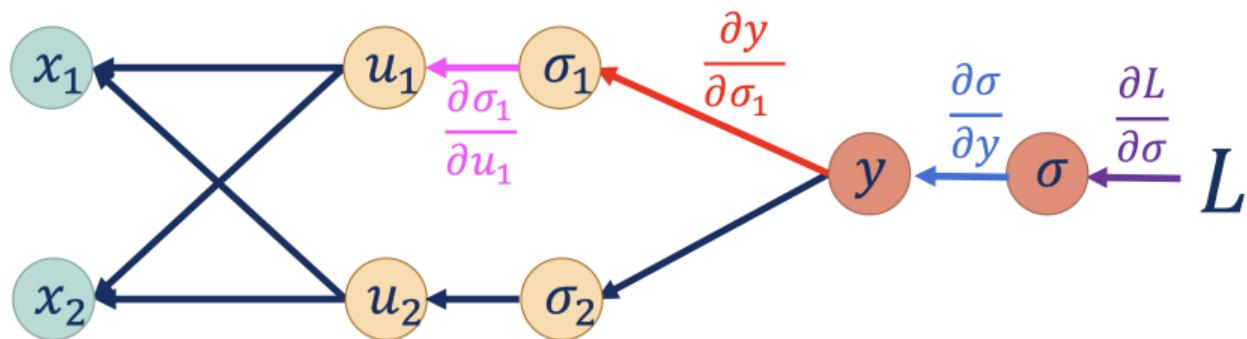
Training —○ backpropagation



Training —○ backpropagation

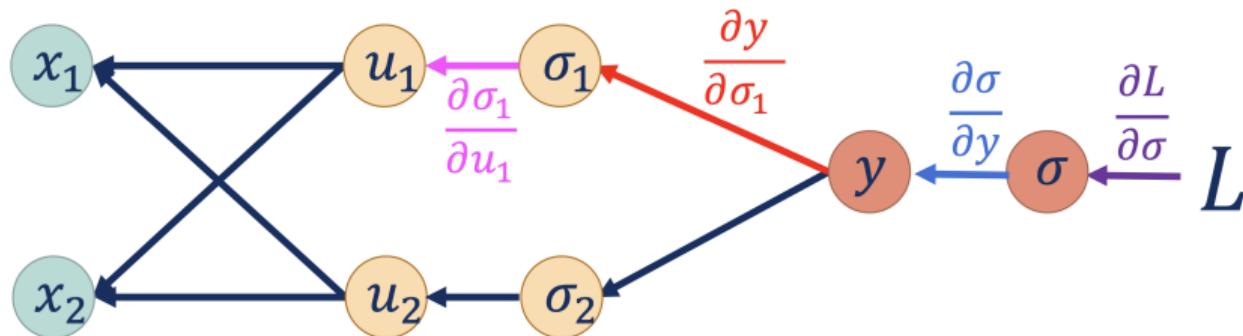


Training —○ backpropagation



$$\frac{\partial L}{\partial u_1} = \frac{\partial L}{\partial \sigma} \frac{\partial \sigma}{\partial y} \frac{\partial y}{\partial \sigma_1} \frac{\partial \sigma_1}{\partial u_1}$$

Training ——○ backpropagation



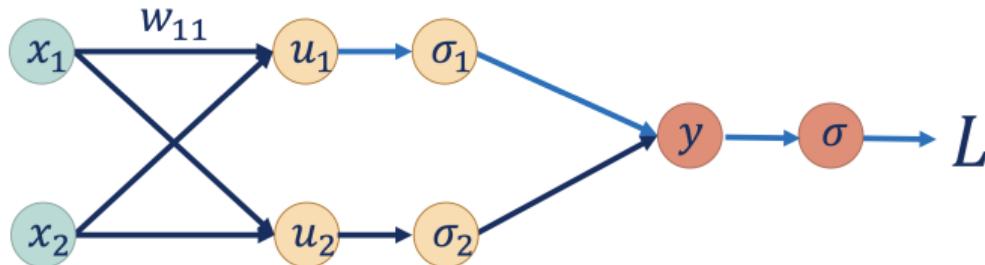
$$\frac{\partial L}{\partial u_1} = \frac{\partial L}{\partial \sigma} \frac{\partial \sigma}{\partial y} \frac{\partial y}{\partial \sigma_1} \frac{\partial \sigma_1}{\partial u_1}$$

- this procedure is called **backpropagation**
  - its idea is to **collect derivatives** at each step in the computational graph w/o recalculating them every single time for every weight

# Training

—○ wrap it up

train NN in your browser!

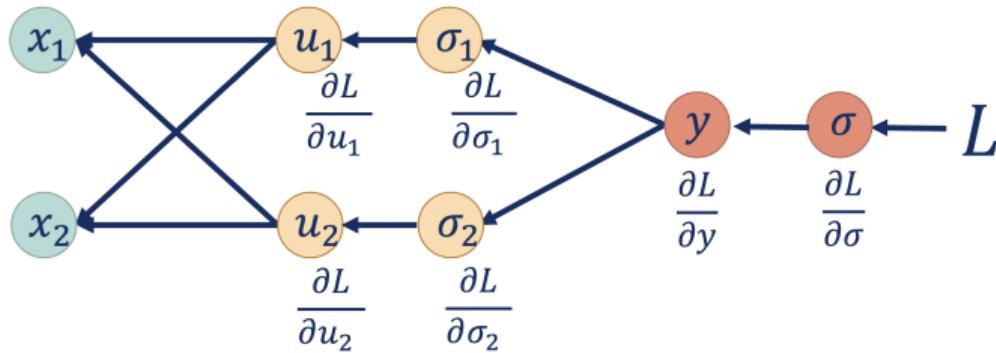


- 1 make **forward pass** through NN to calculate the output of each neuron and value of loss function

# Training

—○ wrap it up

train NN in your browser!

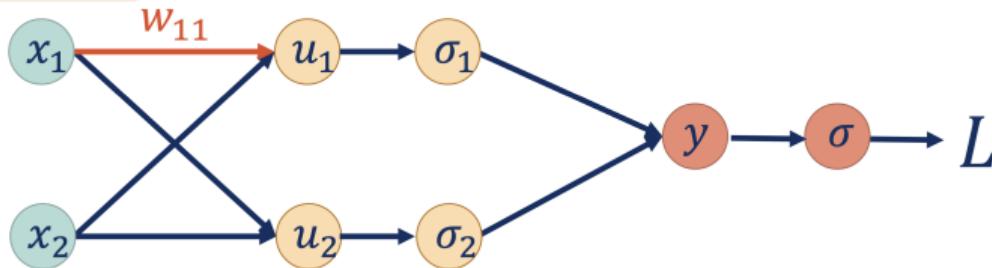


- 1 make **forward pass** through NN to calculate the output of each neuron and value of loss function
- 2 with **backward pass** go back through NN to calculate gradients for all weights

# Training

—○ wrap it up

train NN in your browser!



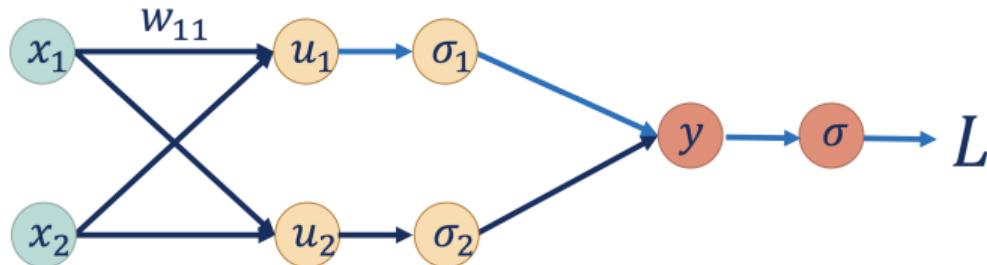
$$w_{11}^{(t+1)} = w_{11}^{(t)} - \gamma_t \frac{\partial L}{\partial w_{11}}$$

- 1 make **forward pass** through NN to calculate the output of each neuron and value of loss function
- 2 with **backward pass** go back through NN to calculate gradients for all weights
- 3 update weights with their gradients

# Training

—○ wrap it up

[train NN in your browser!](#)



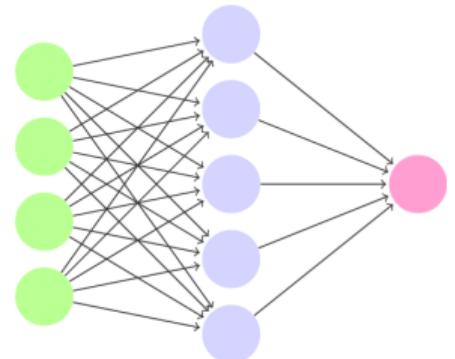
- 1 make **forward pass** through NN to calculate the output of each neuron and value of loss function
- 2 with **backward pass** go back through NN to calculate gradients for all weights
- 3 update weights with their gradients
- 4 repeat until convergence

\*one iteration (**epoch**) = forward and backward pass

# Going deeper

# Going deeper —— o universal approximation theorem

- Roughly speaking, any well-behaved function  $f$  can be approximated **arbitrarily close** with a 1-hidden layer NN, given wide enough hidden layer
- But in practice this is often not the case:
  - loss function is heavily non-convex
  - overfitting
  - not straightforward how to find this NN



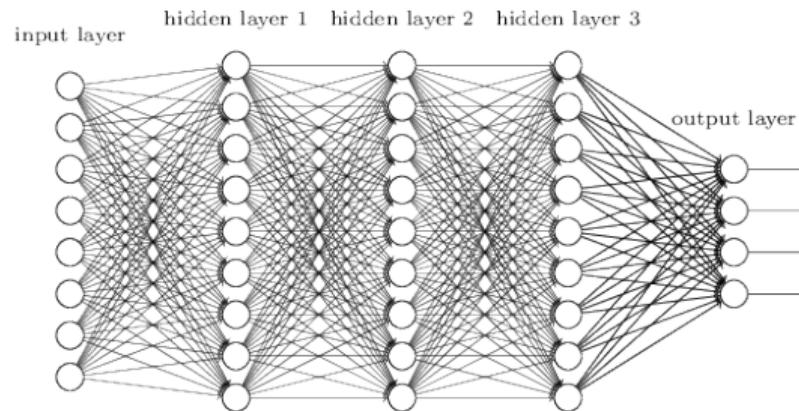
# Going deeper



stack more layers

Going deeper with convolutions

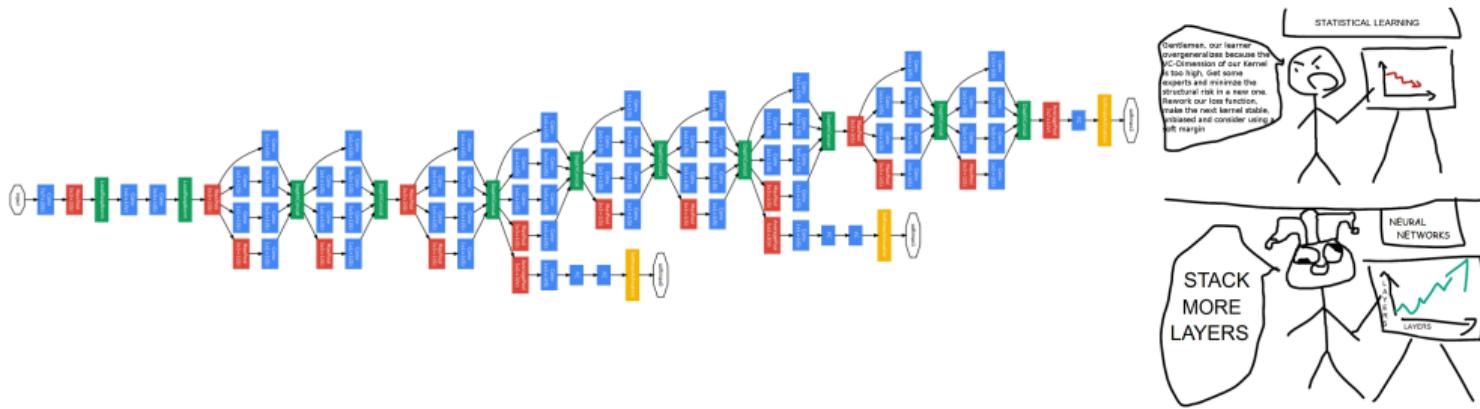
- In practice stacking more layers generally **improves performance**



Going deeper —○ stack more layers

## Going deeper with convolutions

- In practice stacking more layers generally **improves performance**
  - Much more layers...

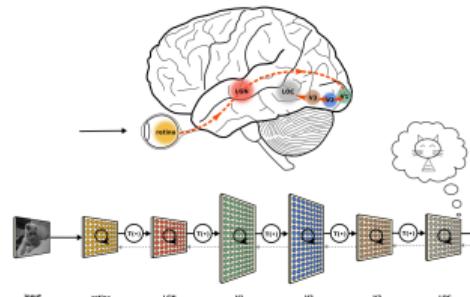


# Going deeper

—○ stack more layers

## Going deeper with convolutions

- In practice stacking more layers generally **improves performance**
- Much more layers...
- Which is reminiscent of the **brain structure**
- Signals travel through multiple areas of different organization
- This makes our perception system incredibly advanced in understanding reality



# Going deeper

—○ stack more layers

Going deeper with convolutions

- In practice stacking more layers generally **improves performance**
- Much more layers...
- Which is reminiscent of the **brain structure**
- Signals travel through multiple areas of different organization
- This makes our perception system incredibly advanced in understanding reality
- **but there are some problems...**

# Going deeper — vanishing gradients

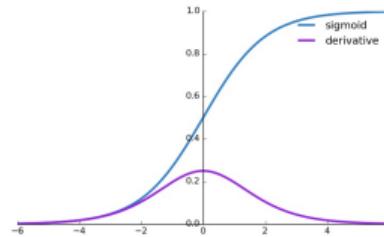
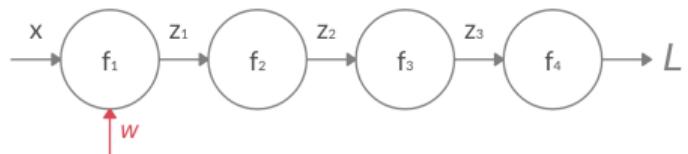
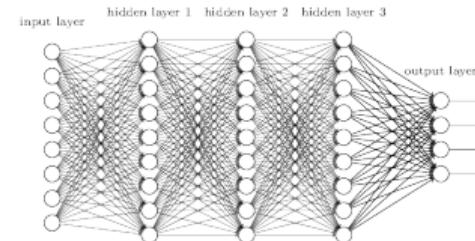
- Layer  $f_i(z_{i-1})$  takes the output  $z_{i-1}$  from the previous layer and returns  $z_i$
- Using chain rule and sigmoid activation we have:

$$\Delta w_j \sim \frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial \mathcal{L}}{\partial f_i} \frac{\partial f_i}{\partial f_{i-1}} \cdots \frac{\partial f_1}{\partial w_j}$$

$$\frac{\partial f_i}{\partial f_{i-1}} = \sigma(z_{i-1})(1 - \sigma(z_{i-1}))$$

$$\bullet \left| \frac{\partial f_i}{\partial f_{i-1}} \right| \leq \frac{1}{4} \Rightarrow \frac{\partial \mathcal{L}}{\partial w_j} \lesssim \left( \frac{1}{4} \right)^n \Rightarrow \Delta w_j \rightarrow 0, n \rightarrow \infty$$

→ there's no learning happening



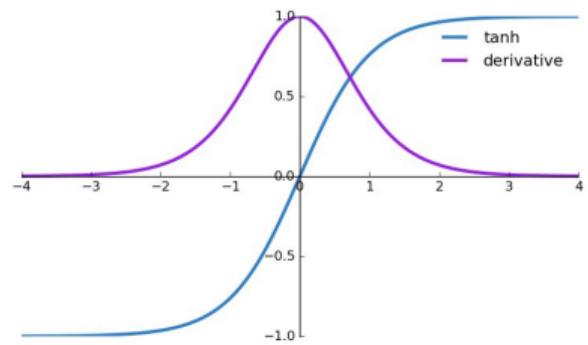
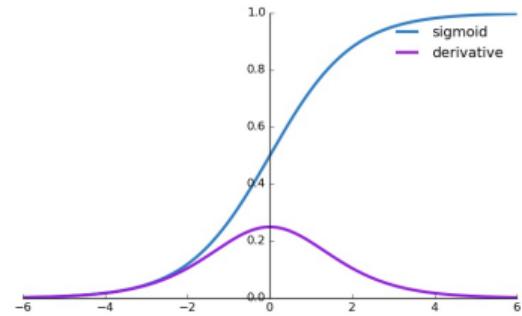
# Going deeper —○ activation functions

- Let's have a closer look at sigmoid activation:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- outputs are in  $[0,1]$  range  $\Rightarrow$  "neuron fired" intuition
- outputs are not zero-centered
- saturate at large  $|z|$   $\Rightarrow$  kill gradients ( $\rightarrow 0$ )**

- same applies to  $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

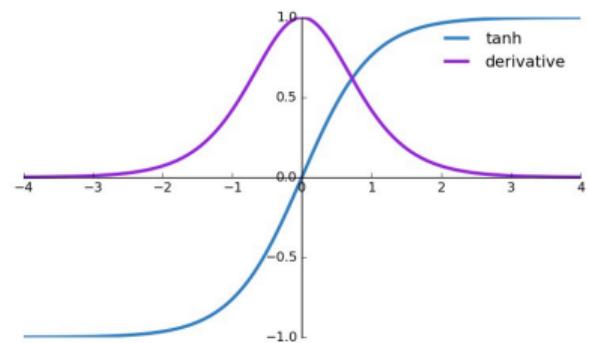
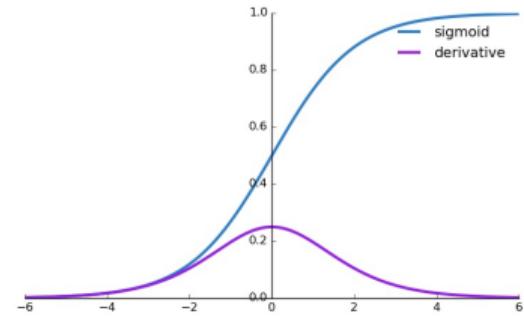


# Going deeper —○ activation functions

- Let's have a closer look at sigmoid activation:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- outputs are in  $[0,1]$  range  $\Rightarrow$  "neuron fired" intuition
- outputs are not zero-centered
- saturate at large  $|z|$   $\Rightarrow$  kill gradients ( $\rightarrow 0$ )**
- same applies to  $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- can we use other activation functions?

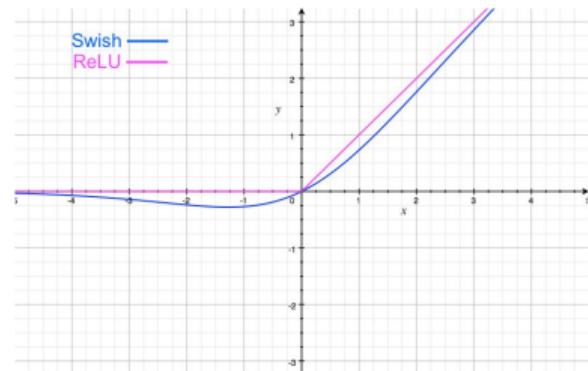


# Going deeper — o activation functions

## Searching for activation functions

- $\text{ReLU}(z) = \max(0, z)$ 
  - gradients don't vanish
  - simple implementation (derivative either 0 or 1)
  - not zero-centered and unbounded
  - neurons can "die"
  - there's more: Leaky ReLU, ELU, GELU, Softplus
- and even more:

$$\text{e.g., } \text{Swish}(z) = \frac{z}{1 + e^{-\beta \cdot z}} = z \cdot \sigma(\beta \cdot z)$$



# Going deeper — o weight initialisation

init playground

- But gradients can also **explode**

# Going deeper

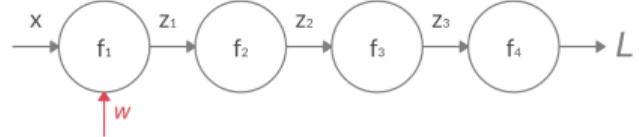
[init playground](#)

- But gradients can also **explode**
- Intuitively, weights are updated by:

$$\Delta w_j \sim \frac{\partial \mathcal{L}}{\partial f_i} \frac{\partial f_i}{\partial f_{i-1}} \dots \frac{\partial f_1}{\partial w_j} \sim \prod_i S_{ij}$$

$S_{ij}$  - scale of gradient at  $i$ -th layer

- if  $S_{ij} \ll 1$ , gradients vanish and weights don't update  $\Rightarrow$  learning is stuck
- if  $S_{ij} \gg 1$ , gradients explode  $\Rightarrow$  learning is extremely unstable



# Going deeper

[init playground](#)

- But gradients can also **explode**
- Intuitively, weights are updated by:

$$\Delta w_j \sim \frac{\partial \mathcal{L}}{\partial f_i} \frac{\partial f_i}{\partial f_{i-1}} \dots \frac{\partial f_1}{\partial w_j} \sim \prod_i S_{ij}$$

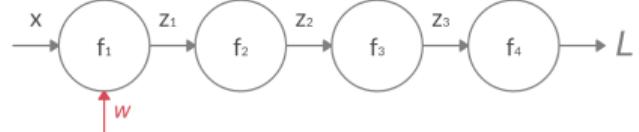
$S_{ij}$  - scale of gradient at  $i$ -th layer

- if  $S_{ij} \ll 1$ , gradients vanish and weights don't update  $\Rightarrow$  learning is stuck
- if  $S_{ij} \gg 1$ , gradients explode  $\Rightarrow$  learning is extremely unstable

**Idea:** constrain the scales at each layer to avoid exponential growth

→ one can show that clever weight initialisation can remedy this

→ e.g. Xavier:  $W_j \sim \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right)$  or He:  $W_j \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$  (for ReLU) initialisations\*



\*library implementations  
may slightly differ

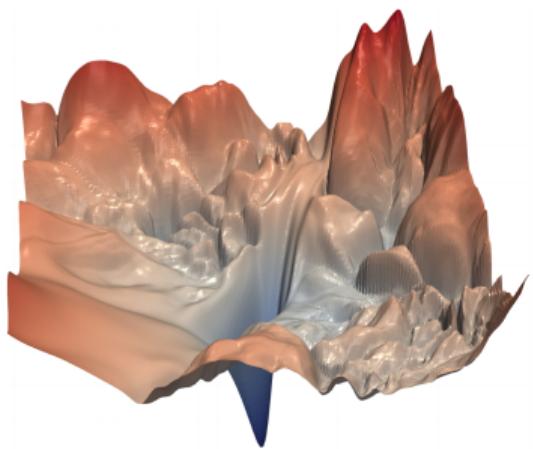
# Tackling overfitting

# Tackling overfitting

— o complexity

Visualizing the Loss Landscape

- NN are highly complicated models with  $> 1M$  weights being normal
- Therefore, optimisation task is extremely tough with loss function being non-convex
- This makes overfitting and getting trapped in a local minimum a piece of cake

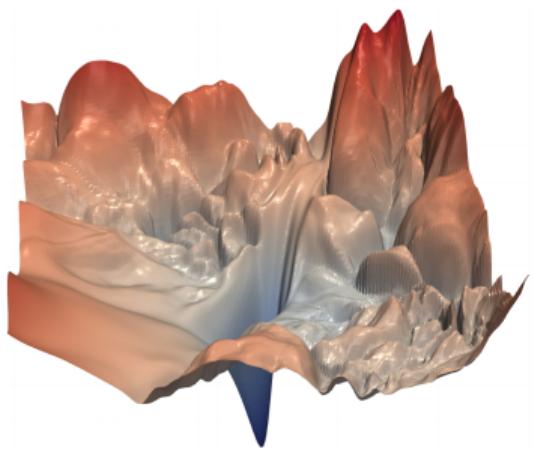


# Tackling overfitting

— o complexity

Visualizing the Loss Landscape

- NN are highly complicated models with  $> 1M$  weights being normal
  - Therefore, optimisation task is extremely tough with loss function being non-convex
  - This makes overfitting and getting trapped in a local minimum a piece of cake
- Improvements in optimisation methods are needed



# Tackling overfitting

—○ SGD recap

[source](#)

- **SGD:**

- At each step  $k$  pick random sample  $(x_l, y_l)$
- Update weights:

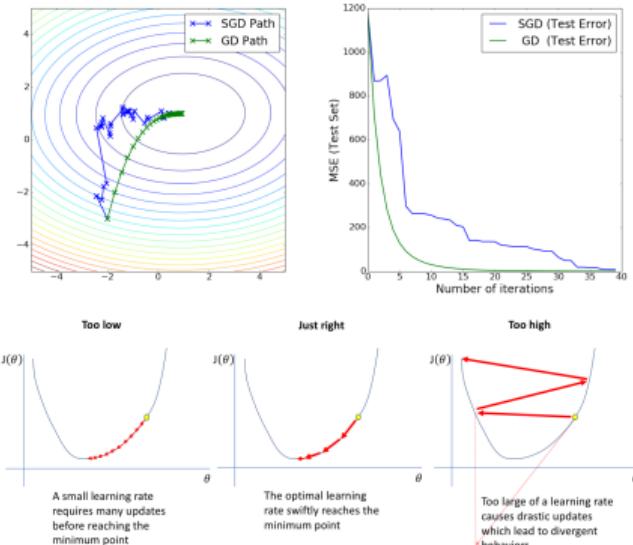
$$\mathbf{w}^{(k)} \leftarrow \mathbf{w}^{(k-1)} - \eta \nabla \mathcal{L}(y_l, \mathbf{x}_l) \Big|_{\mathbf{w}=\mathbf{w}^{(k-1)}}$$

- **Mini-batch SGD:**

- Iterate through the dataset in chunks (batches)
- Aggregate gradients over the chunk:

$$g = \sum_{l \in B} \nabla \mathcal{L}(y_l, \mathbf{x}_l)$$

- Update the weights:  $\mathbf{w}^{(k)} \leftarrow \mathbf{w}^{(k-1)} - \eta \cdot g$



# Tackling overfitting

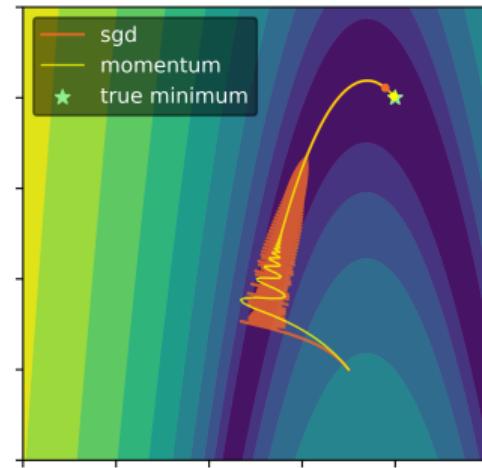
—○ momentum

check out more on [distill](#)

- Let's change perspective to a physical one and add a notion of **velocity**
- Example: a ball rolling down the hill  $\Rightarrow$  treat loss as **potential energy**
- If we build up velocity in a direction with consistent gradient, we can overcome local minima and smooth out rapid oscillations  $\Rightarrow$  **SGD with momentum**:

$$\boldsymbol{v}^{(k)} \leftarrow \beta \boldsymbol{v}^{(k-1)} - \eta \nabla \mathcal{L}(\boldsymbol{y}_l, \boldsymbol{x}_l) \Big|_{\mathbf{w}=\mathbf{w}^{(k-1)}}$$

$$\boldsymbol{\omega}^{(k)} \leftarrow \boldsymbol{\omega}^{(k-1)} + \boldsymbol{v}^{(k)}$$



[source](#)

# Tackling overfitting

—○ momentum

check out more on [distill](#)

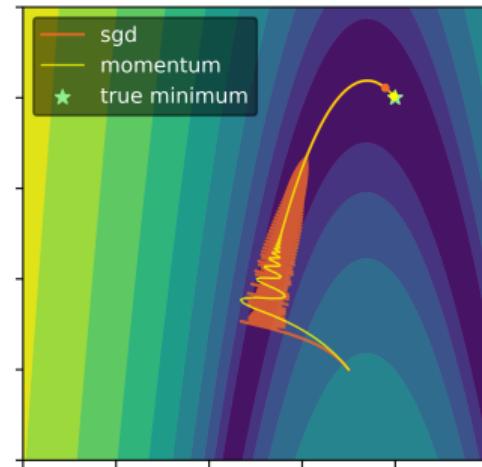
- Let's change perspective to a physical one and add a notion of **velocity**
- Example: a ball rolling down the hill  $\Rightarrow$  treat loss as **potential energy**
- If we build up velocity in a direction with consistent gradient, we can overcome local minima and smooth out rapid oscillations  $\Rightarrow$  **SGD with momentum**:

$$\mathbf{v}^{(k)} \leftarrow \beta \mathbf{v}^{(k-1)} - \eta \nabla \mathcal{L}(\mathbf{y}_l, \mathbf{x}_l) \Big|_{\mathbf{w}=\mathbf{w}^{(k-1)}}$$

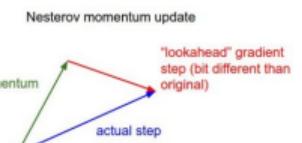
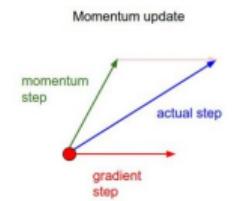
$$\mathbf{\omega}^{(k)} \leftarrow \mathbf{\omega}^{(k-1)} + \mathbf{v}^{(k)}$$

- Nesterov momentum** updates position with "lookahead"

$$\text{gradient } \nabla \mathcal{L}(\mathbf{y}_l, f_{\mathbf{w}}(\mathbf{x}_l)) \Big|_{\mathbf{w}=\mathbf{w}^{(k-1)} + \beta \mathbf{v}^{(k-1)}}$$



[source](#)



[source](#)

# Tackling overfitting — o adaptive LR

- Previously, we were manipulating learning rate  $\eta$  **globally and equally** for all parameters
- This sounds like a limitation, since gradient scales vary significantly and we could've gained from adjusting LRs for **each component independently**

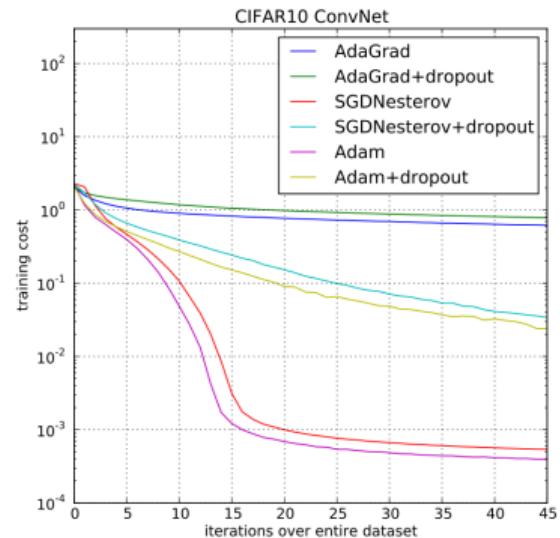
# Tackling overfitting

— o adaptive LR

- Previously, we were manipulating learning rate  $\eta$  **globally and equally** for all parameters
- This sounds like a limitation, since gradient scales vary significantly and we could've gained from adjusting LRs for **each component independently**
- So here comes RMSprop (Hinton's lecture notes):

$$\mathbf{w}^{(k)} \leftarrow \mathbf{w}^{(k-1)} - \frac{\eta}{\sqrt{\text{Var}[g^2]_{(k)} + \varepsilon}} \odot \left. \frac{\partial \mathcal{L}}{\partial \mathbf{w}} \right|_{\mathbf{w}=\mathbf{w}^{(k-1)}}$$

- Adam combines ideas of momentum and RMSprop
- Note:** there's also LR annealing and more sophisticated optimizers



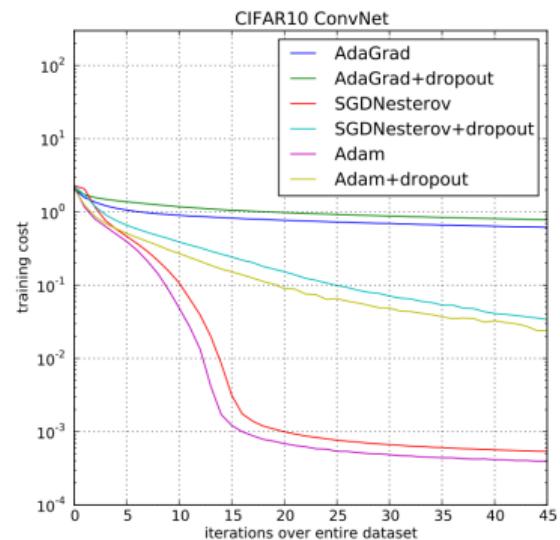
# Tackling overfitting

—○ adaptive LR

- Previously, we were manipulating learning rate  $\eta$  **globally and equally** for all parameters
- This sounds like a limitation, since gradient scales vary significantly and we could've gained from adjusting LRs for **each component independently**
- So here comes RMSprop (Hinton's lecture notes):

$$\mathbf{w}^{(k)} \leftarrow \mathbf{w}^{(k-1)} - \frac{\eta}{\sqrt{\text{Var}[g^2]_{(k)} + \varepsilon}} \odot \left. \frac{\partial \mathcal{L}}{\partial \mathbf{w}} \right|_{\mathbf{w}=\mathbf{w}^{(k-1)}}$$

- Adam combines ideas of momentum and RMSprop
- Note:** there's also LR annealing and more sophisticated optimizers



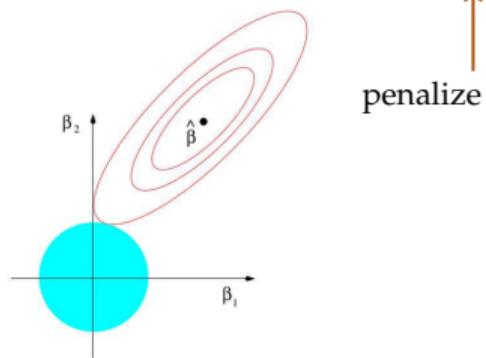
Nice moment to show this animation

# Tackling overfitting

—○ weight regularisation

## L2 regularization (Tikhonov)

$$Q(\omega) = \mathbb{E}_{p(x,y)} [\mathcal{L}(y, f(x, \omega))] + \lambda \sum_{j=1}^K \omega_j^2$$

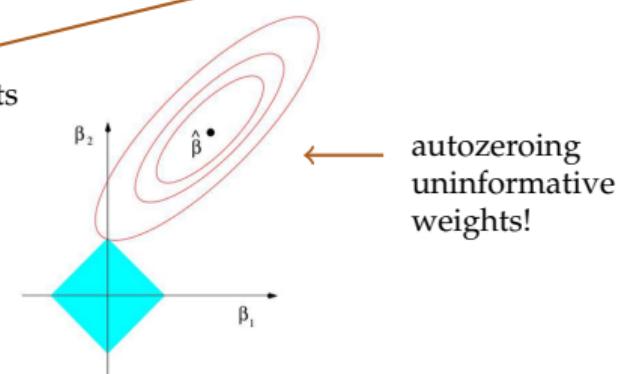


↑  
penalize model for too large weights

## L1 regularization (LASSO)

least absolute shrinkage and selection operator

$$Q(\omega) = \mathbb{E}_{p(x,y)} [\mathcal{L}(y, f(x, \omega))] + \lambda \sum_{j=1}^K |\omega_j|$$



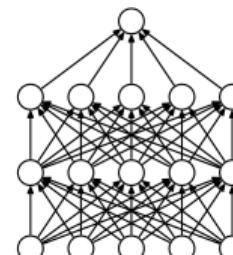
←  
autozeroing  
uninformative  
weights!

# Tackling overfitting

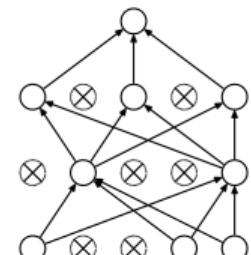
—○ dropout

[paper](#)

- Let's randomly **drop neurons with probability  $p$**  during the training
- Essentially, this would mean that at each iteration we train a *new subnetwork*
- This allows for **breaking co-adaptation** of neurons  $\Rightarrow$  neurons forced to learn useful features w/o relying on neighbouring ones
- And makes it very simple, elegant and **powerful regularization** technique



(a) Standard Neural Net



(b) After applying dropout.

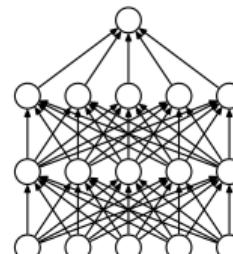
# Tackling overfitting

—○ dropout

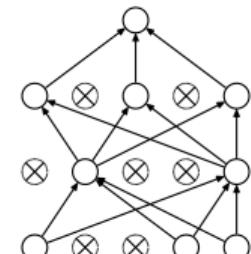
[paper](#)

- Let's randomly **drop neurons with probability  $p$**  during the training
- Essentially, this would mean that at each iteration we train a *new subnetwork*
- This allows for **breaking co-adaptation** of neurons  $\Rightarrow$  neurons forced to learn useful features w/o relying on neighbouring ones
- And makes it very simple, elegant and **powerful regularization** technique

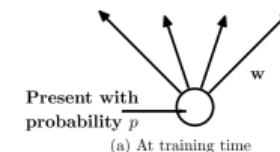
**Note:** during testing one needs to simply scale neurons' outputs with  $p$  to compensate *on average* for dropout during the training



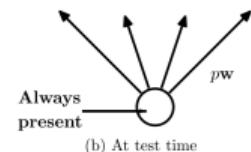
(a) Standard Neural Net



(b) After applying dropout.



Present with probability  $p$   
(a) At training time



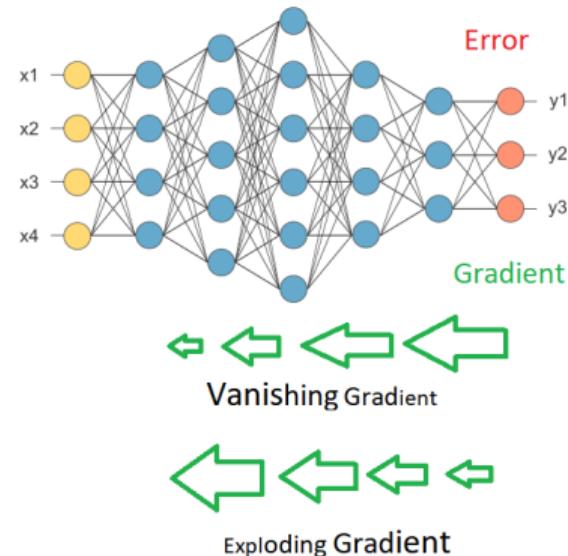
Always present  
(b) At test time

# Tackling overfitting

— o batch normalisation

[paper](#)

- As was mentioned earlier, training procedure is sensitive to the scale of gradients in NN
- Furthermore, the latter is connected to the inputs' scale of layers, which in turn tends to vary throughout the training (aka "**internal covariate shift**")
- This slows down the training and makes the procedure sensitive to weight initialisation



# Tackling overfitting

— o batch normalisation

[paper](#)

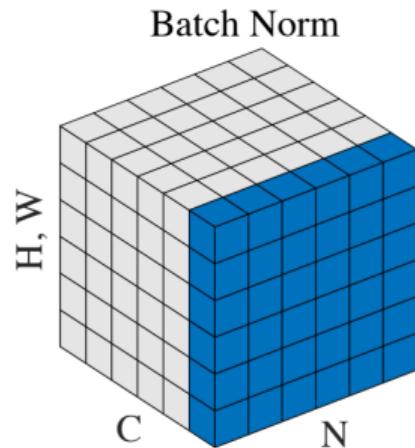
- It was proposed to approach this problem with **normalising layer inputs** over a batch ( $\gamma$  and  $\beta$  are learnable parameters):

$$\mu_B = \frac{1}{|B|} \sum_{i \in B} x_i, \quad \sigma_B^2 = \frac{1}{|B|} \sum_{i \in B} (x_i - \mu_B)^2$$

$$y_i = \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

- This turned out to improve performance, speed up and stabilize convergence (but didn't really remove internal covariate shift)

**Note:** there are other fancy ways to normalize inputs



# NN ZOO

*A mostly complete chart of*

# Neural Networks

[link](#)

- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool

©2016 Fjodor van Veen - [asimovinstitute.org](http://asimovinstitute.org)

Perceptron (P)



Feed Forward (FF)



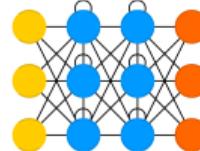
Radial Basis Network (RBF)



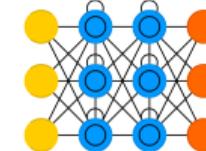
Deep Feed Forward (DFF)



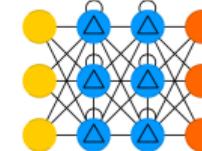
Recurrent Neural Network (RNN)



Long / Short Term Memory (LSTM)



Gated Recurrent Unit (GRU)



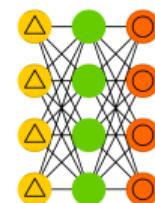
Auto Encoder (AE)



Variational AE (VAE)



Denoising AE (DAE)



Sparse AE (SAE)



# Summary

- Modelling nonlinearities
- Neural Network
  - automating feature engineering
  - architecture
  - terminology
- Training
  - chain rule
  - backpropagation
- Going deeper
  - universal approximation theorem
  - vanishing gradients
  - activation functions
  - weight initialisation
- Tackling overfitting
  - gradient descent modifications
  - weight regularization
  - dropout
  - batch normalisation
- NN zoo