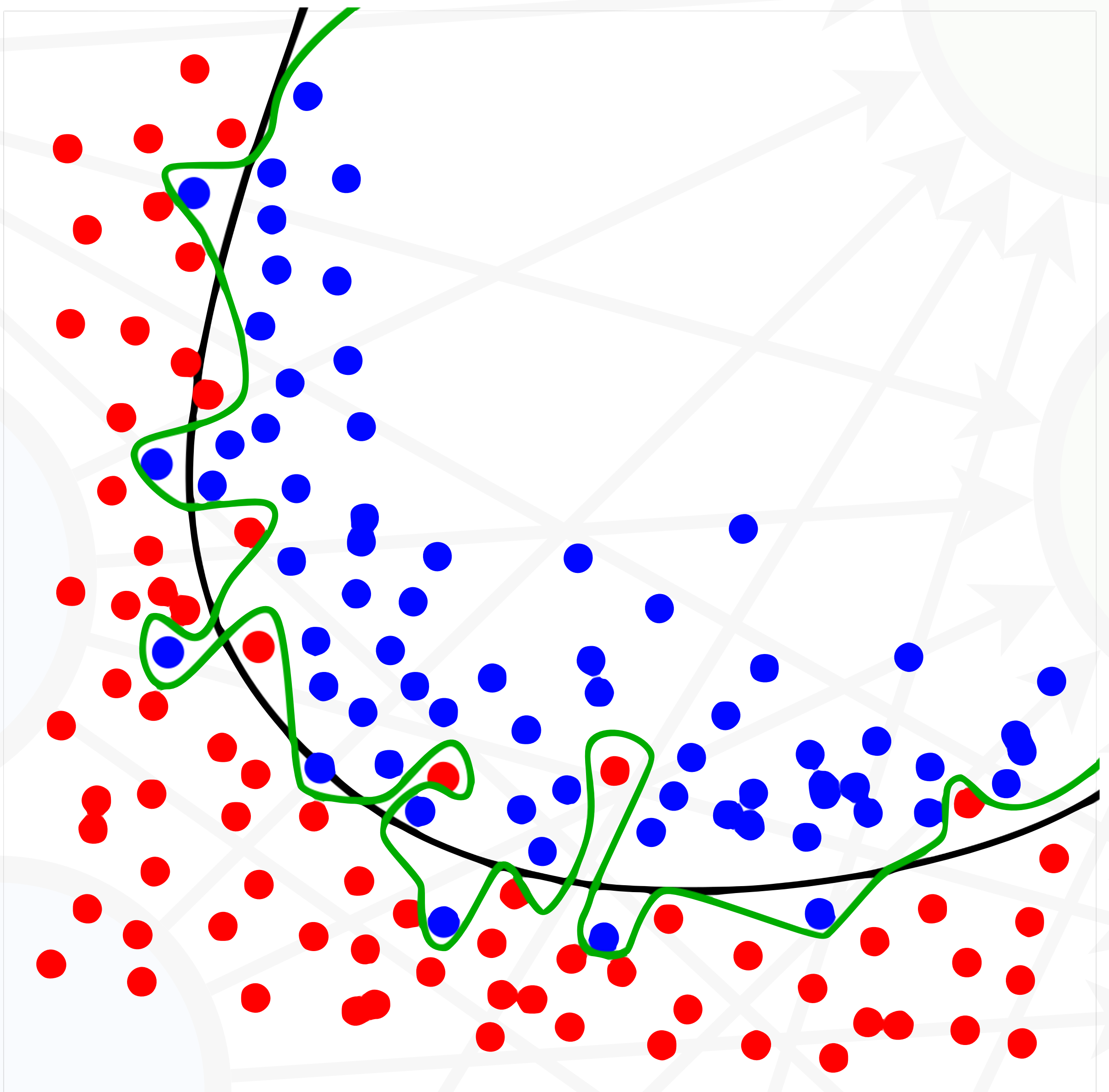
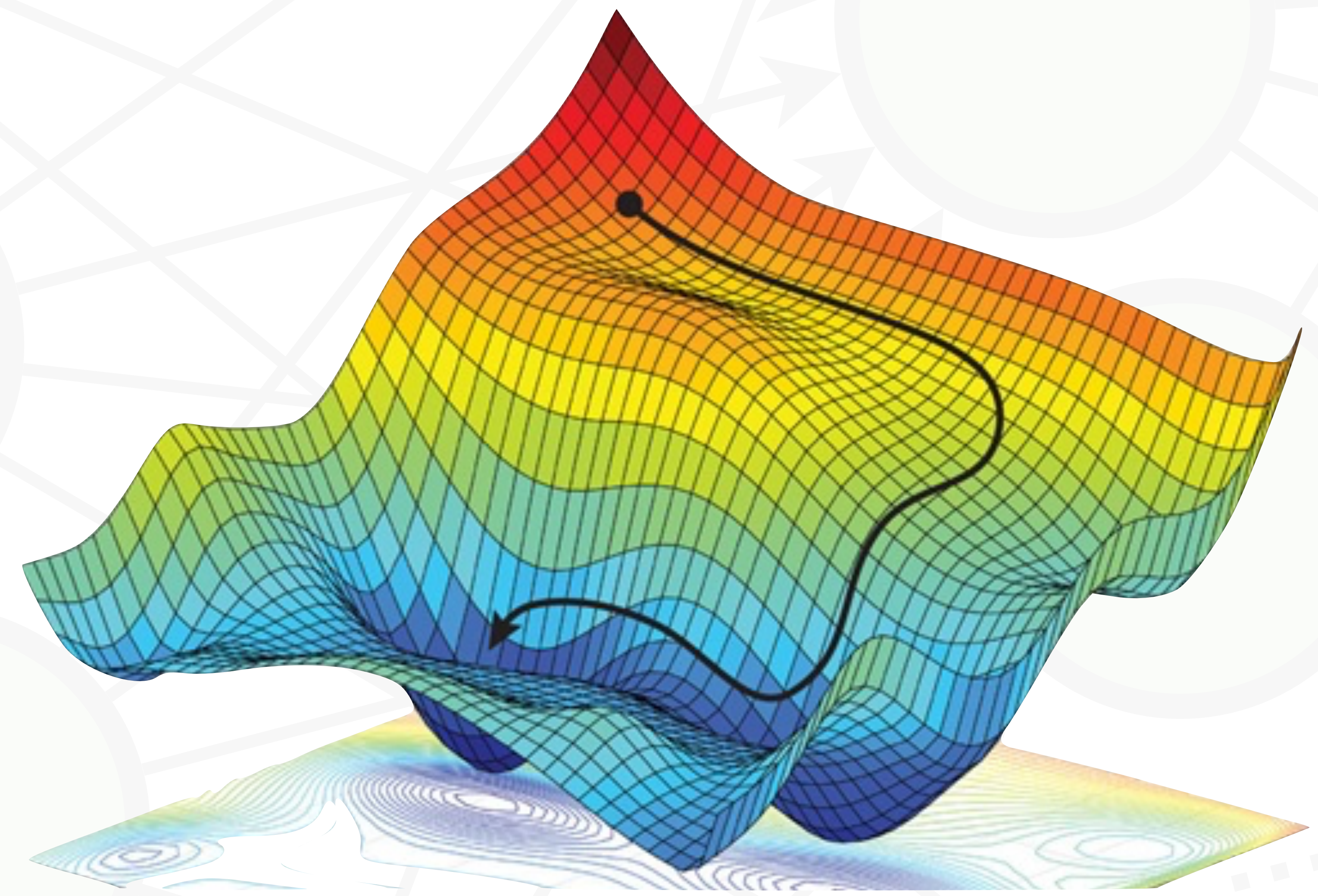




Mastering Model Building

Docent: *Marcel Rieger (UHH)*
Tutors: *Bogdan Wiederspan (UHH), Boyang Yu (LMU), Lars Sowa (KIT)*

Deep Learning School — "Basic Concepts"

09.08.2022

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

ERUM DATA HUB

"Neural Network Building Blocks" — Dennis Noll

↓

"Mastering model building"

- **Goals**
  - Extend your intuition on effective *model building*
  - Learn practical concepts that guide through *model optimization*
  - Fill your box of tools that help you identify the *do's & don'ts*

- **Contents**
  - Variants of and improvements in fully-connected networks
  - Numerical insights & considerations
  - Overtraining suppression and regularization
  - Optimization techniques
  - Technical insights to TensorFlow and Keras

↓

"Convolutional Neural Networks" — Judith Reindl

Today
14:30 - 16:00

20" **1. Variants of and improvements in fully-connected networks (FCNs)**
- Gradient calculation (recap), vanishing gradients, ResNet, ensemble learning, multi-purpose networks

30" **2. Numerical insights & considerations**
- Domains, feature & output scaling, batch normalization, SELU, categorical embedding, class imbalance

40" **3. Techniques 1/2 & hands-on**
- Keras functional API, custom Keras layer, computing gradients

Today
16:30 - 18:00

25" **4. Regularization & overtraining suppression**
- Overtraining & generalization, capacity & capability, regularization, dataset splitting

25" **5. Model optimization**
- Optimizer choices, class-importance, hyper-parameters, search strategies

40" **6. Techniques 2/2 & hands-on**
- Compute architecture, TensorFlow eager and graph, custom training loop, tensorboard

Tomorrow
09:00 - 10:30

10" **7. Exercise introduction: Identifying Jets in Particle Collider Experiments**
- Problem statement, input data & features, objective(s)

70" **8. Hands-on!**
- Classification task, implementing newly learned techniques, extension to multi-purpose network

10" **9. Exercise summary and tips**
- Example wrap-up, additional practical tips

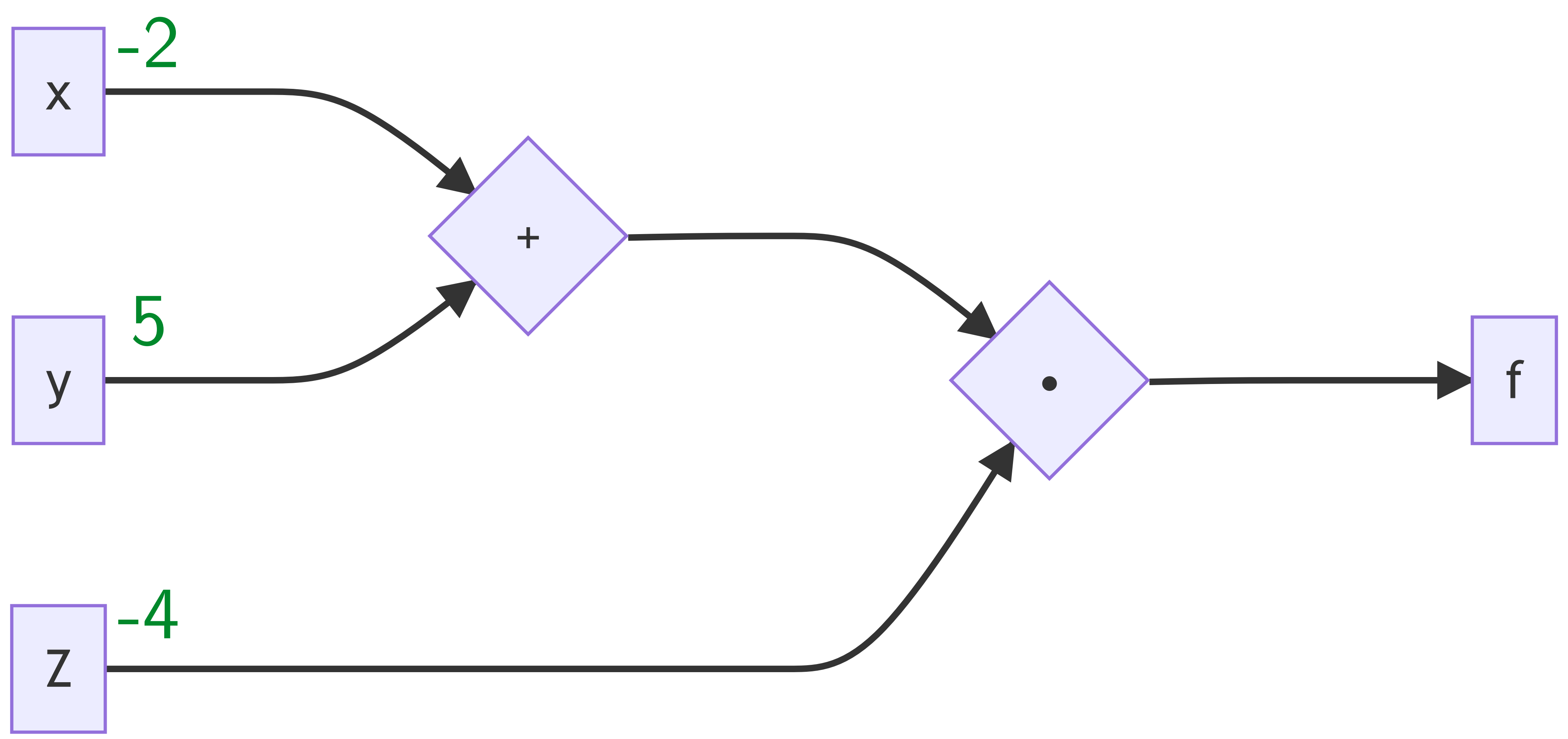
Before we start ...

- **Ask questions!**
  - Feel free to interrupt as more people might have the exact same question that's worth discussing
  - *Learning* to discuss ML topics is an important goal of this school
- **Time for hands-on parts is deliberately generous**
  - Technical insights and practical hands-on experience are essential for mastering ML
  - You should be able to fully understand and digest presented concepts & code examples
  - Best ideas emerge from just "playing around"
- **I have a particle-physicist's bias ...**
  - Chosen examples might reflect that
  - I'll try to keep them as simple as possible

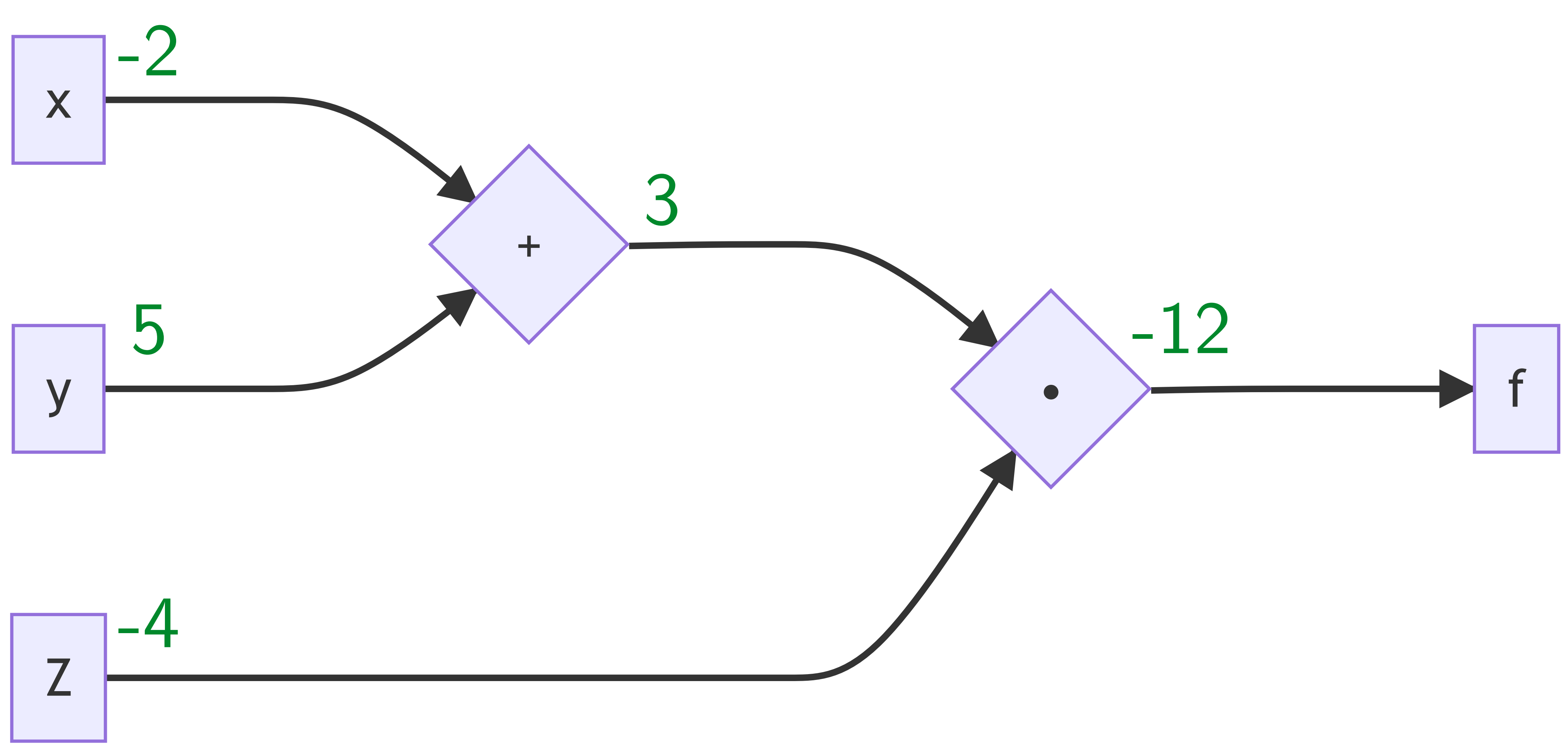
1. Variants of and improvements in fully-connected networks (FCNs)

- Consider $f(x, y, z) = (x + y) \cdot z$ as a computational graph that is too complicated to derive directly

- Perform the forward pass and back-propagation for x = -2, y = 5, z = -4



- Consider $f(x, y, z) = (x + y) \cdot z$ as a computational graph that is too complicated to derive directly
- Perform the forward pass and back-propagation for x = -2, y = 5, z = -4

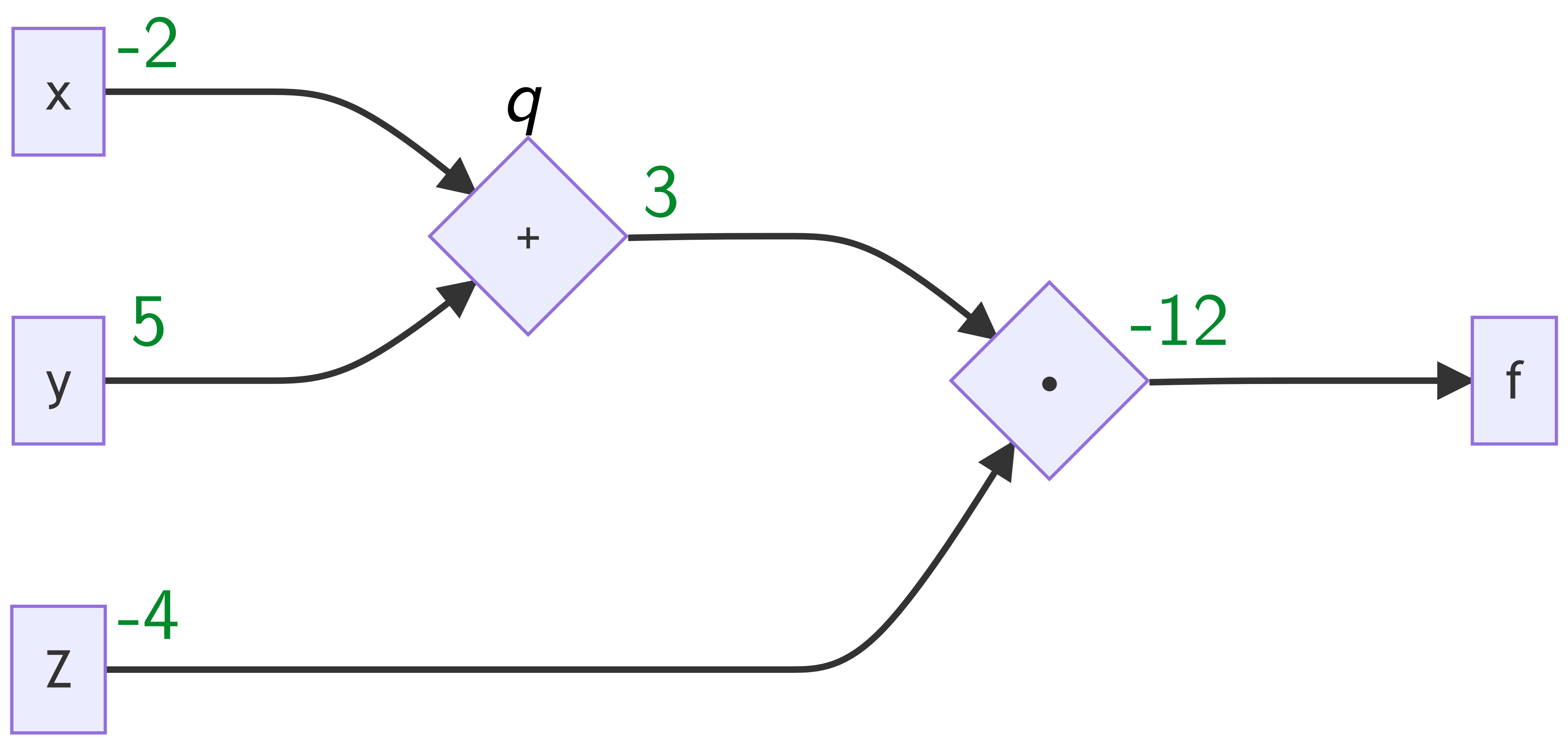


- Consider $f(x, y, z) = (x + y) \cdot z$ as a computational graph that is too complicated to derive directly

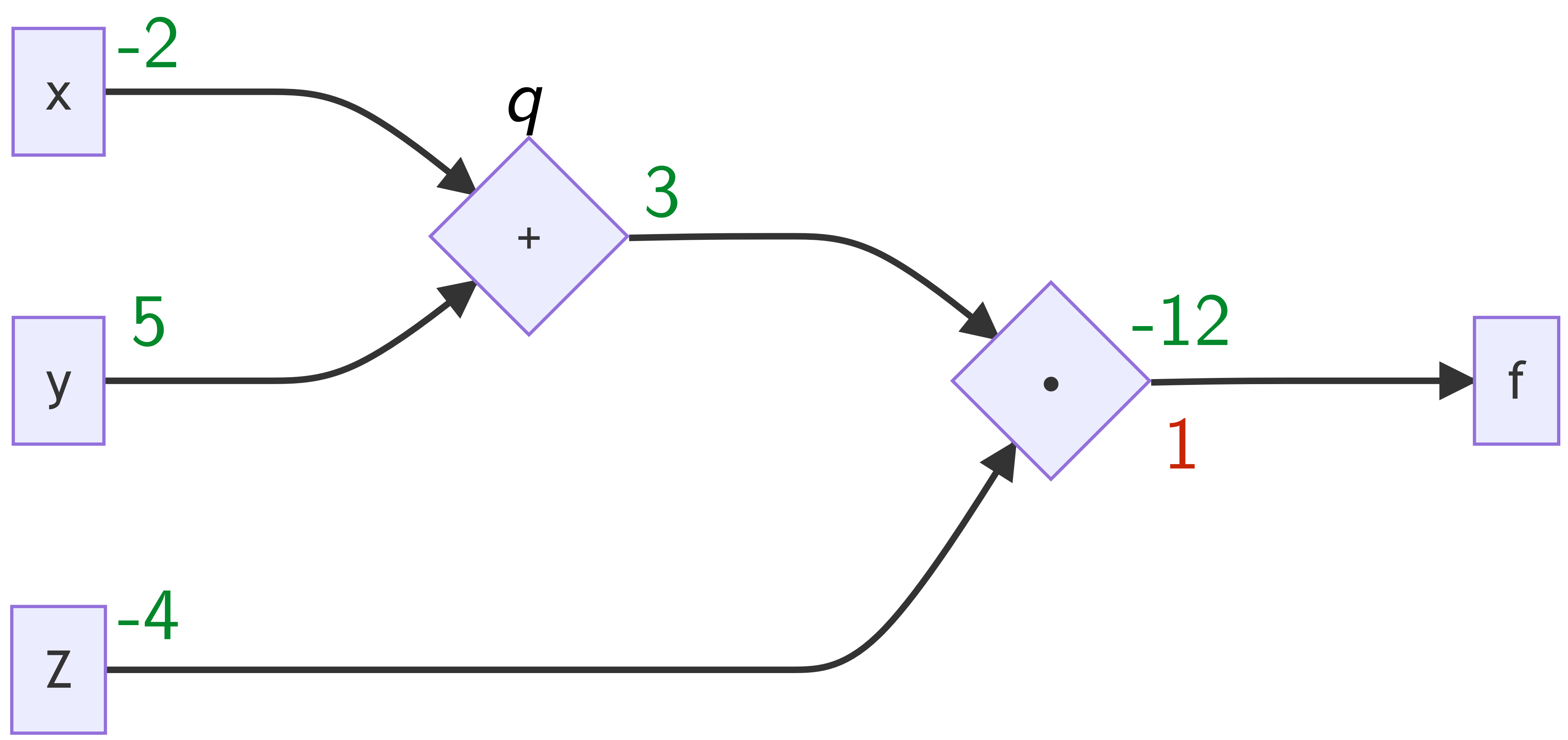
- Perform the forward pass and back-propagation for x = -2, y = 5, z = -4

- Introduce $q = x + y \rightarrow f = q \cdot z$

- Partial derivatives $\frac{\partial f}{\partial z} = q, \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial q}{\partial x} = \frac{\partial q}{\partial y} = 1$

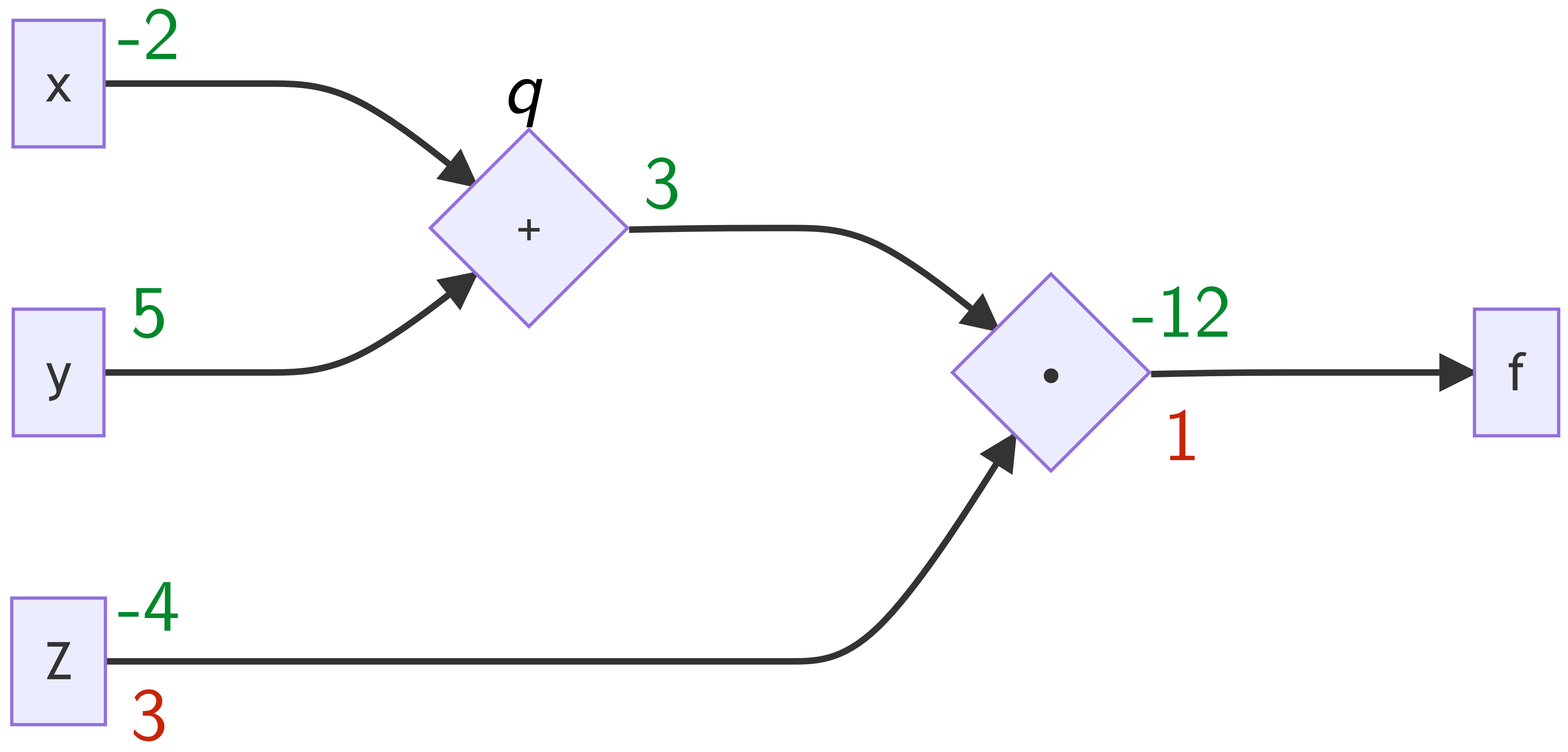


- Consider $f(x, y, z) = (x + y) \cdot z$ as a computational graph that is too complicated to derive directly
- Perform the forward pass and back-propagation for x = -2, y = 5, z = -4
- Introduce $q = x + y \rightarrow f = q \cdot z$
- Partial derivatives $\frac{\partial f}{\partial z} = q, \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial q}{\partial x} = \frac{\partial q}{\partial y} = 1$

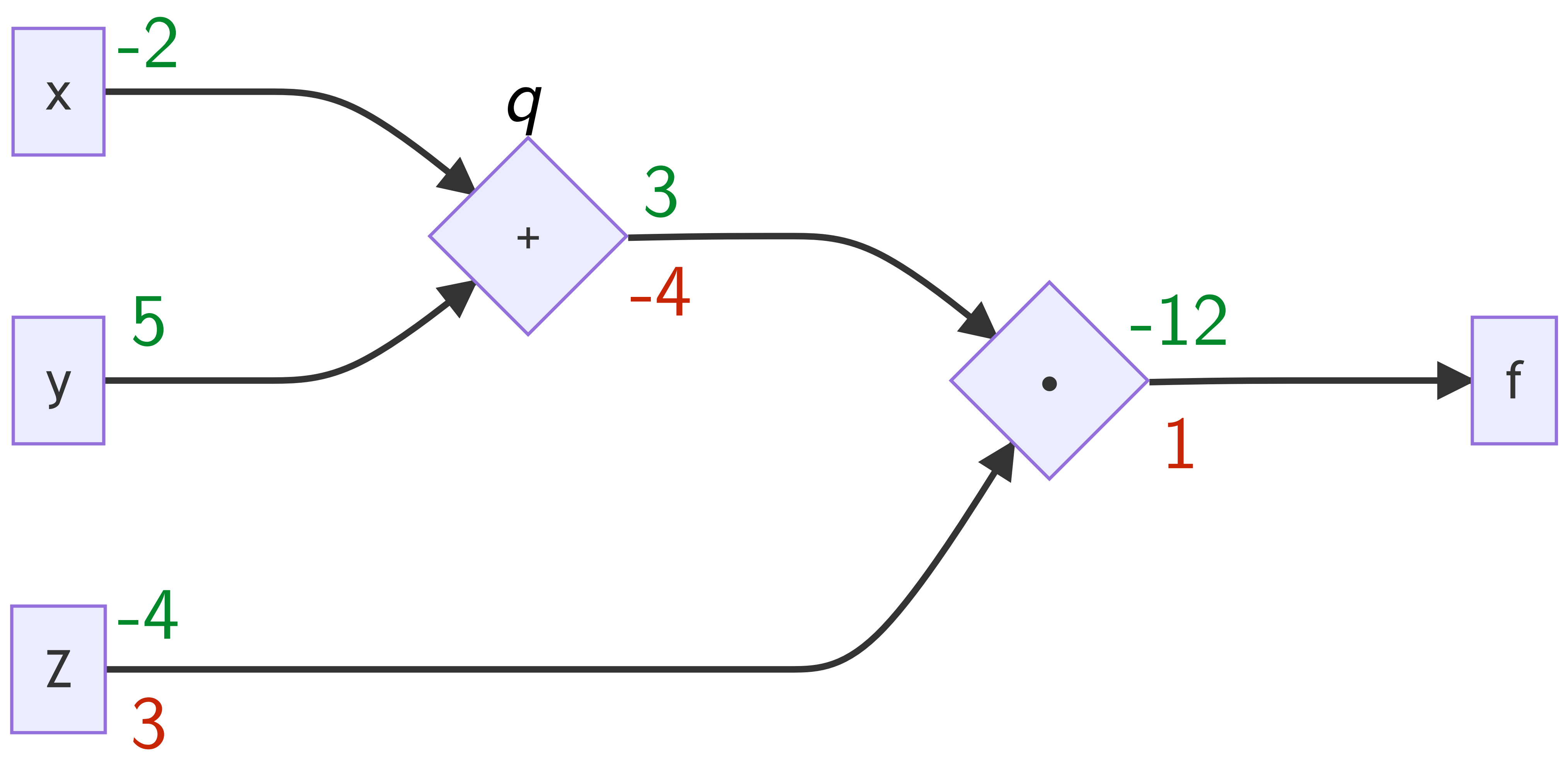


Gradients by example (1)

- Consider $f(x, y, z) = (x + y) \cdot z$ as a computational graph that is too complicated to derive directly
- Perform the forward pass and back-propagation for x = -2, y = 5, z = -4
- Introduce $q = x + y \rightarrow f = q \cdot z$
- Partial derivatives $\frac{\partial f}{\partial z} = q, \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial q}{\partial x} = \frac{\partial q}{\partial y} = 1$

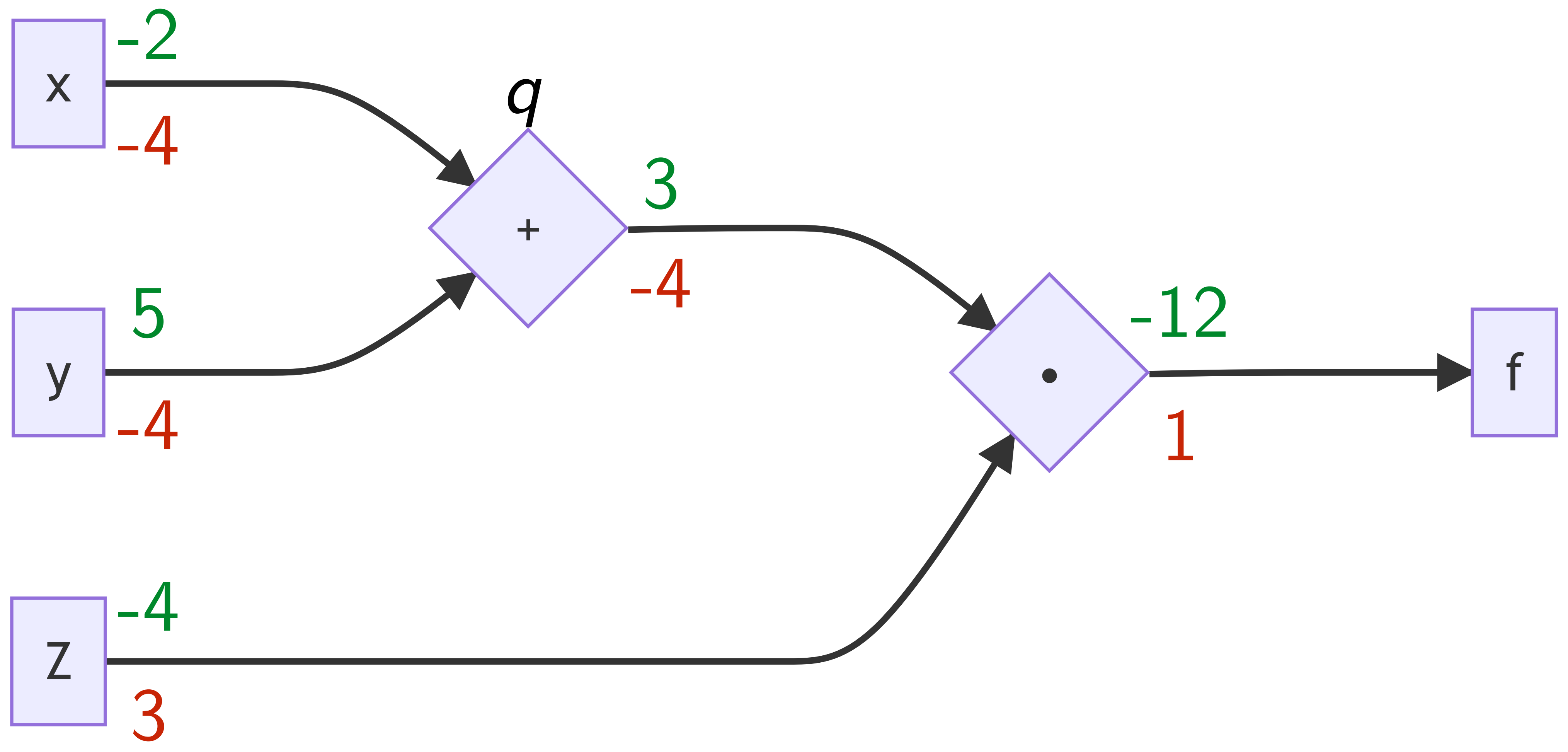


- Consider $f(x, y, z) = (x + y) \cdot z$ as a computational graph that is too complicated to derive directly
- Perform the forward pass and back-propagation for x = -2, y = 5, z = -4
- Introduce $q = x + y \rightarrow f = q \cdot z$
- Partial derivatives $\frac{\partial f}{\partial z} = q, \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial q}{\partial x} = \frac{\partial q}{\partial y} = 1$



Gradients by example (1)

- Consider $f(x, y, z) = (x + y) \cdot z$ as a computational graph that is too complicated to derive directly
- Perform the forward pass and back-propagation for x = -2, y = 5, z = -4
- Introduce $q = x + y \rightarrow f = q \cdot z$
- Partial derivatives $\frac{\partial f}{\partial z} = q, \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial q}{\partial x} = \frac{\partial q}{\partial y} = 1$
- Obtain $\frac{\partial f}{\partial x}$ through **chain rule** (same for y)
  - $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x}$



- Consider $f(x, y, z) = (x + y) \cdot z$ as a computational graph that is too complicated to derive directly

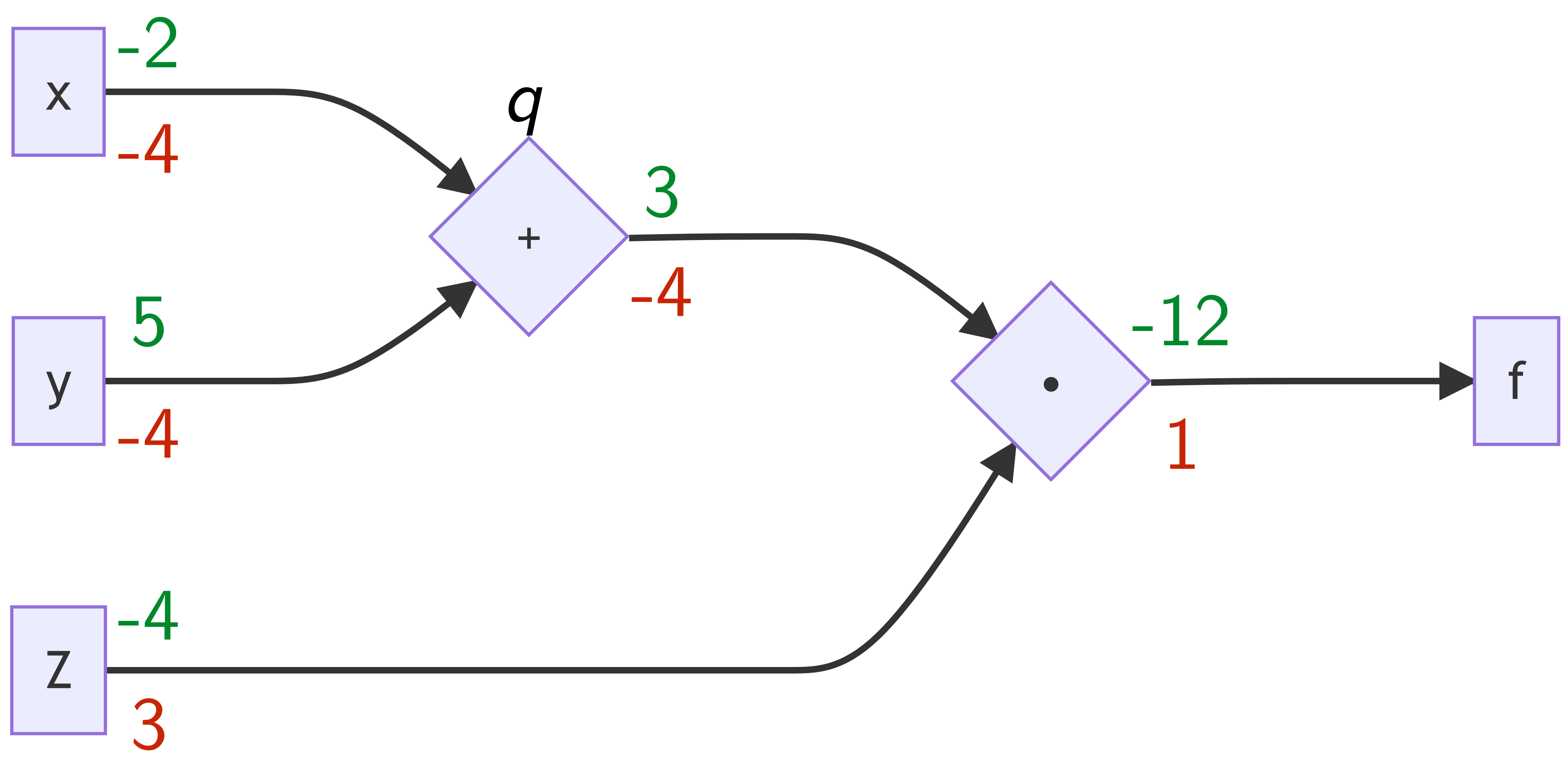
- Perform the forward pass and back-propagation for x = -2, y = 5, z = -4

- Introduce $q = x + y \rightarrow f = q \cdot z$

- Partial derivatives $\frac{\partial f}{\partial z} = q, \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial q}{\partial x} = \frac{\partial q}{\partial y} = 1$

- Obtain $\frac{\partial f}{\partial x}$ through **chain rule** (same for y)
  - $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x}$

- Trivial in this example, but important implications
  - When an input is changed by Δ, f changes by "Δ × gradient"



- Consider $f(x, y, z) = (x + y) \cdot z$ as a computational graph that is too complicated to derive directly

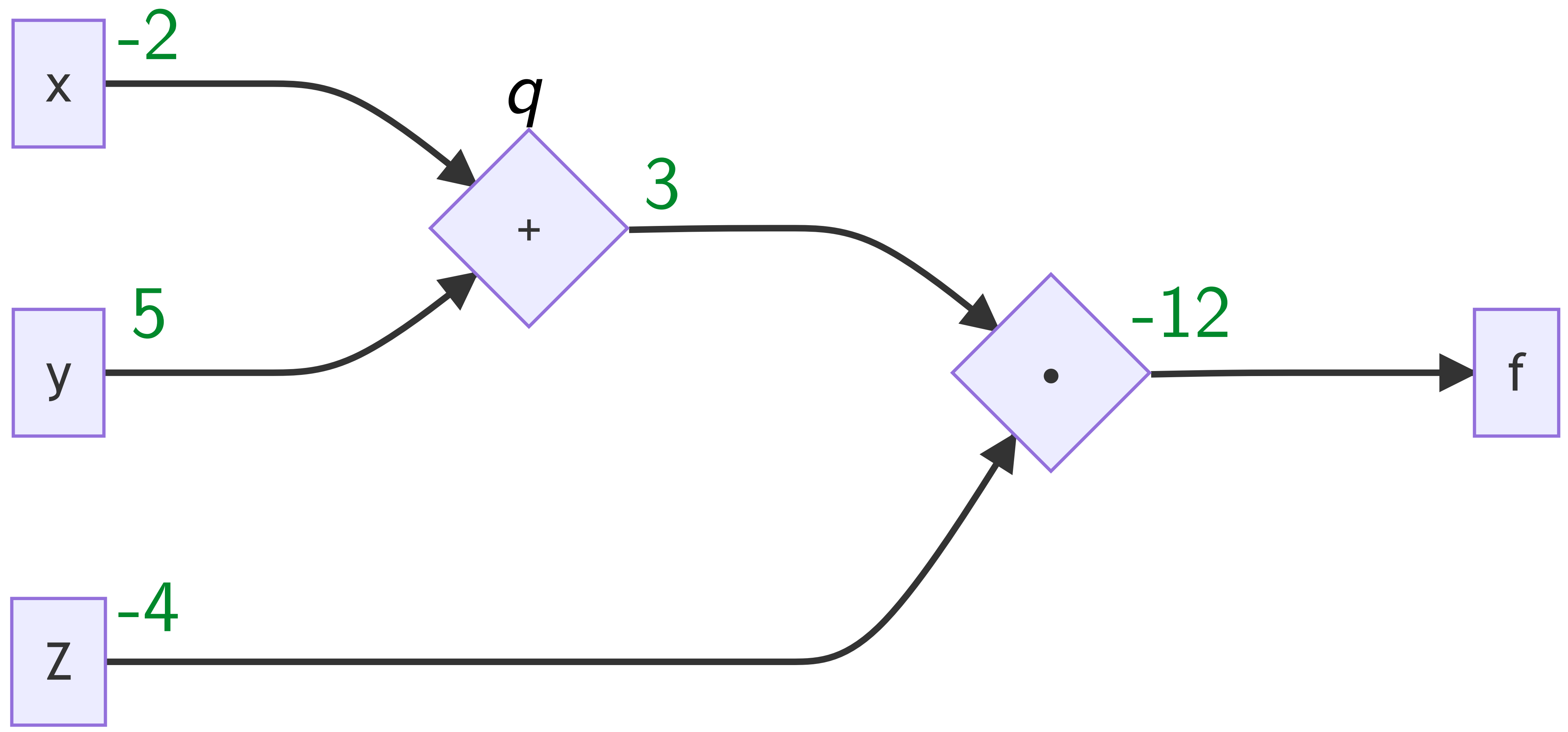
- Perform the forward pass and back-propagation for x = -2, y = 5, z = -4

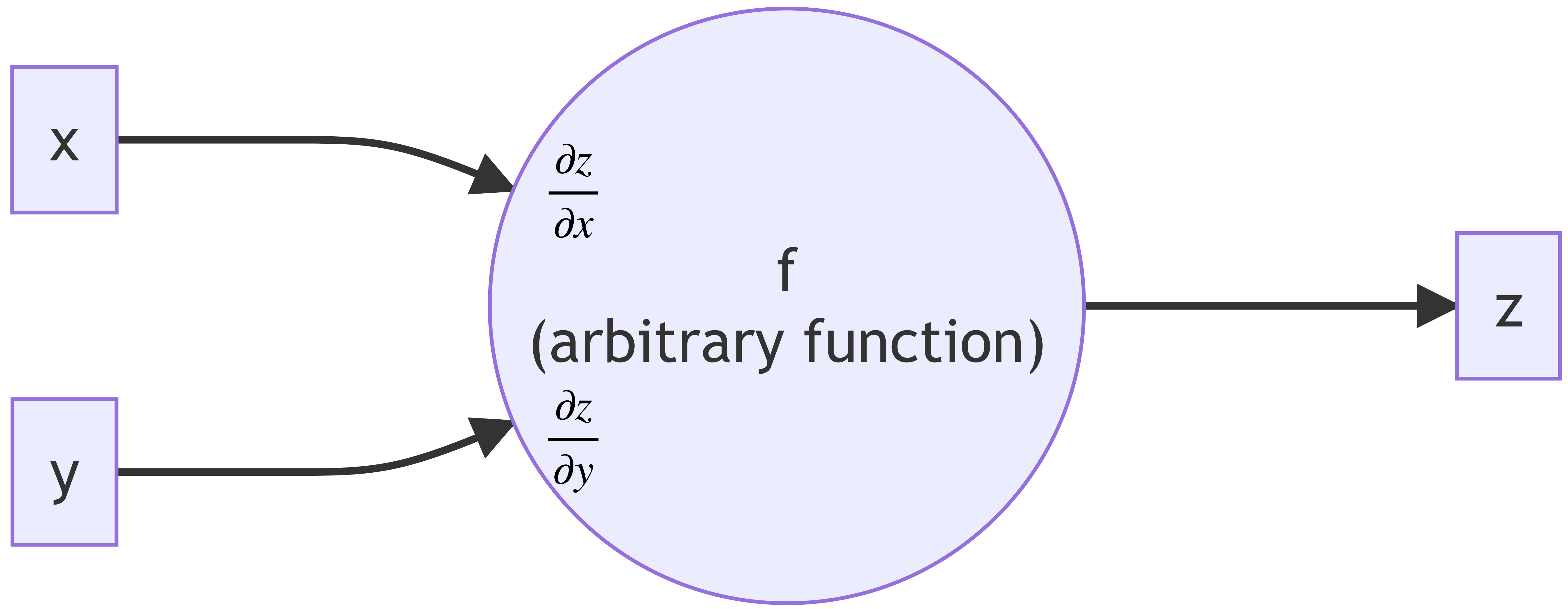
- Introduce $q = x + y \rightarrow f = q \cdot z$

- Partial derivatives $\frac{\partial f}{\partial z} = q, \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial q}{\partial x} = \frac{\partial q}{\partial y} = 1$

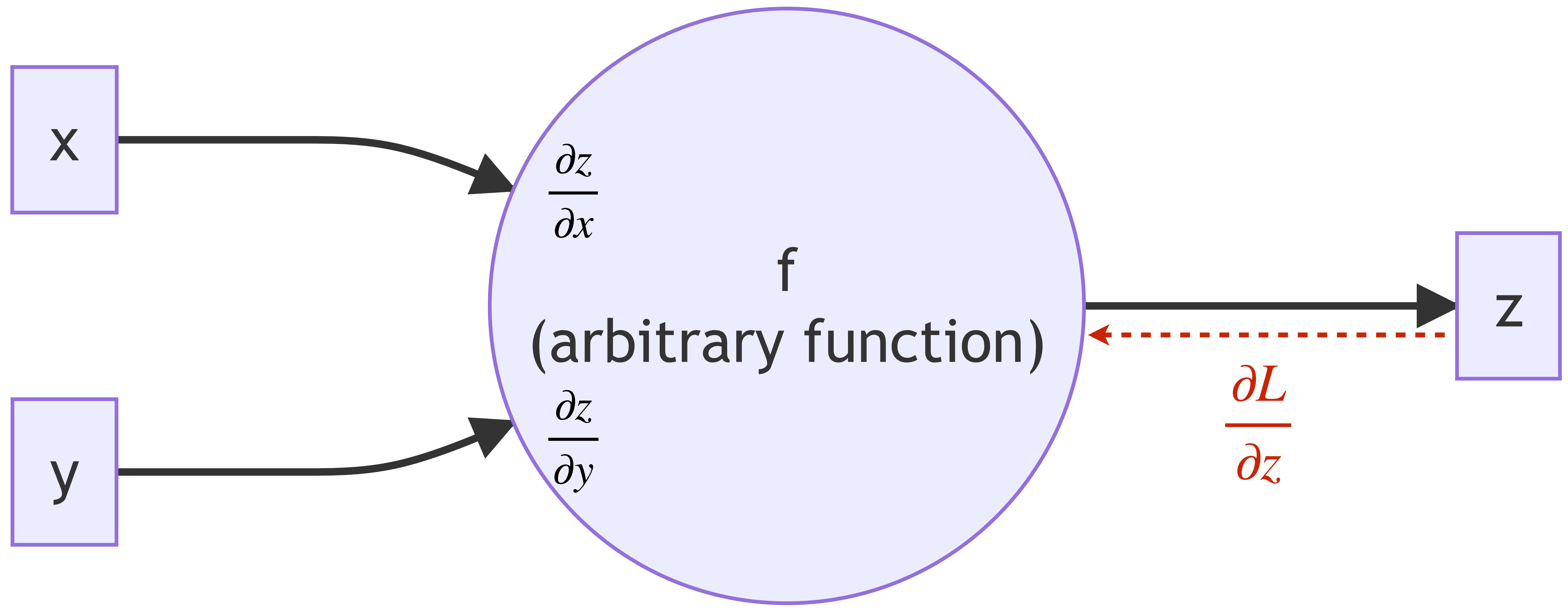
- Obtain $\frac{\partial f}{\partial x}$ through **chain rule** (same for y)
  - $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x}$

- Trivial in this example, but important implications
  - When an input is changed by Δ, f changes by "Δ × gradient"

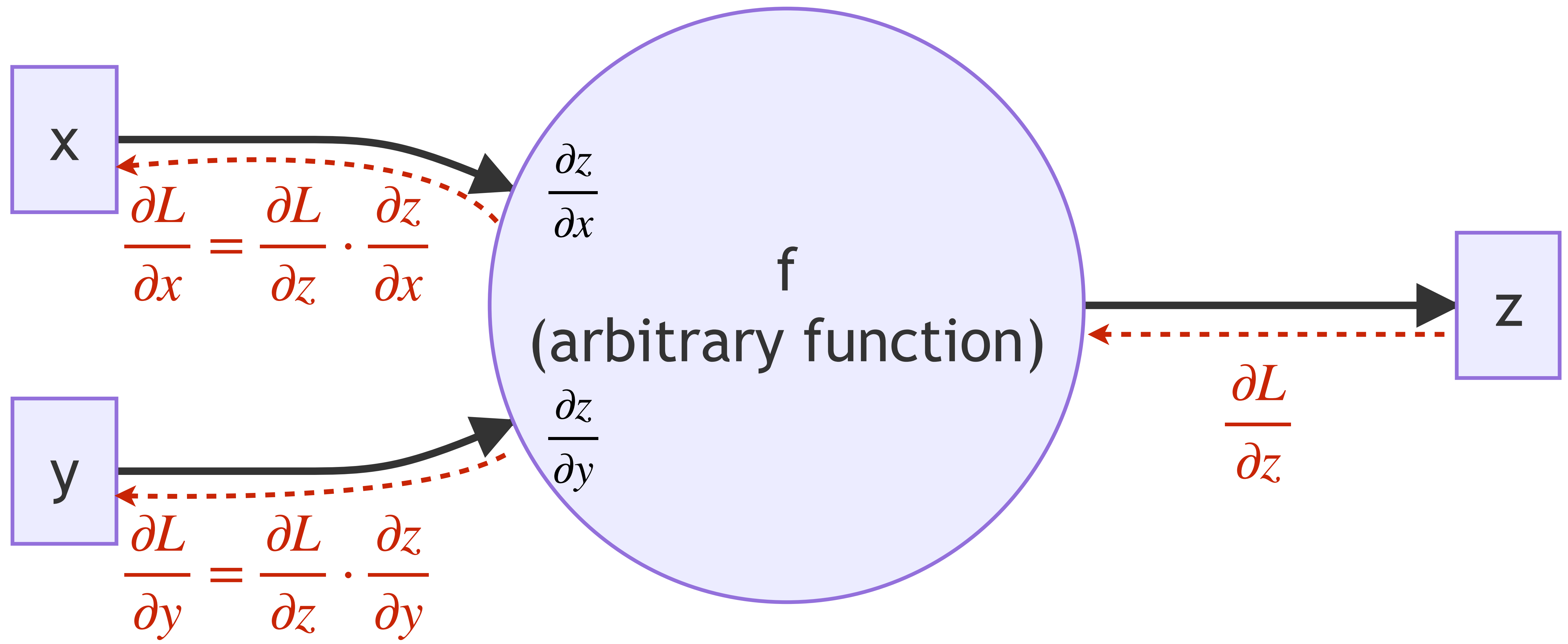




- During forward pass, "operation" f can **already** compute so-called **local** gradients of its output
  - $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$
- Upon back-propagation, global gradient is simply computed by means of back-propagation via multiplication

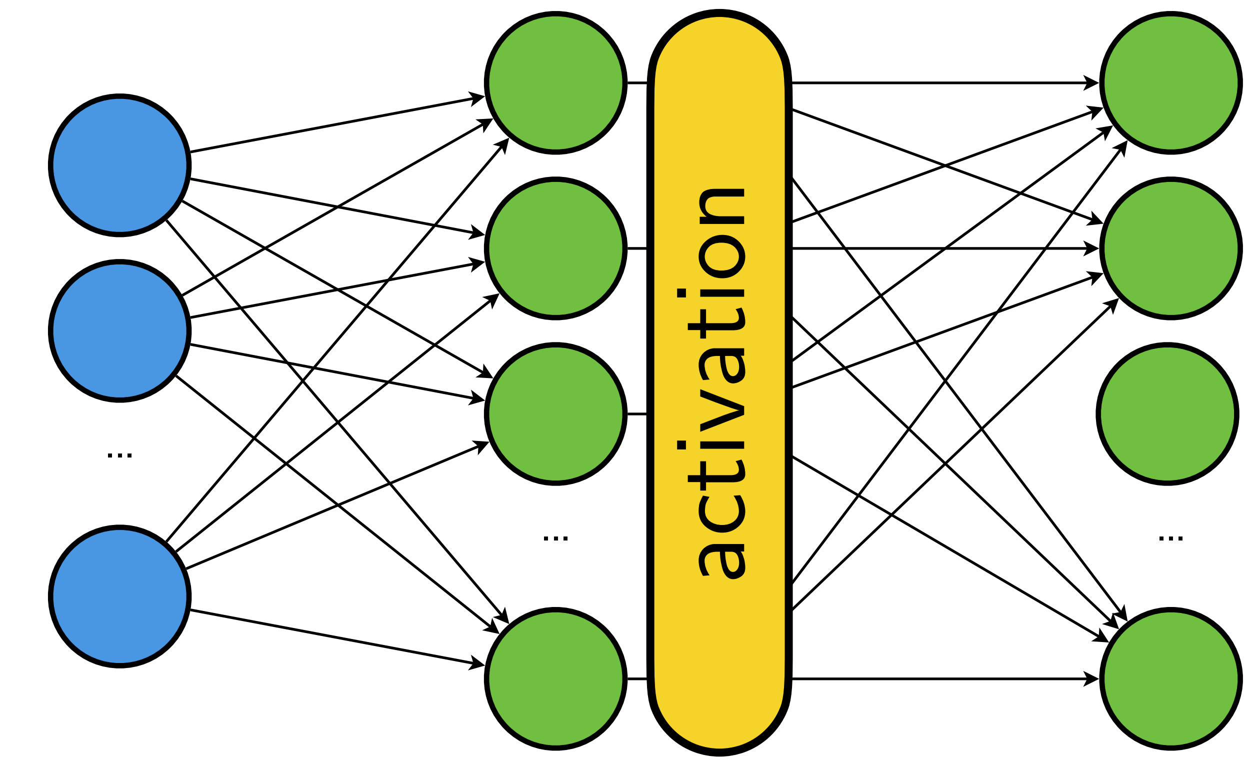


- During forward pass, "operation" f can **already** compute so-called **local** gradients of its output
  - $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$
- Upon back-propagation, global gradient is simply computed by means of back-propagation via multiplication



- During forward pass, "operation" f can **already** compute so-called **local** gradients of its output
  - $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$
- Upon back-propagation, global gradient is simply computed by means of back-propagation via multiplication

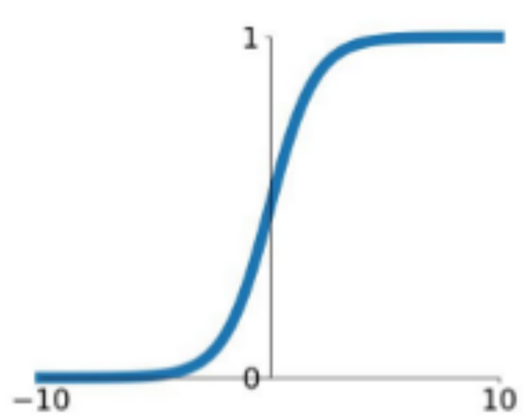
- Activation functions add non-linear behavior to a network layer
  - Allows finding more complex inner representations (hidden features) within fewer layers
  - **But**: need to control input space to prevent vanishing gradients!



- **Examples**

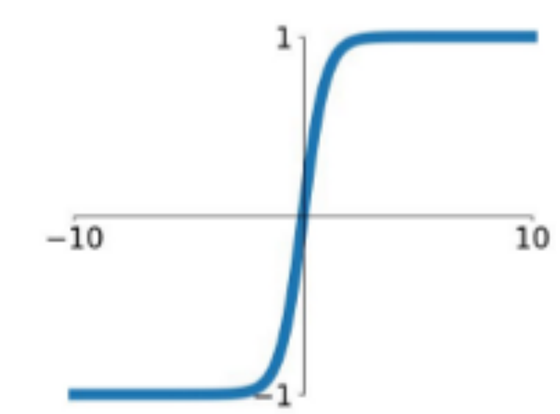
Sigmoid

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



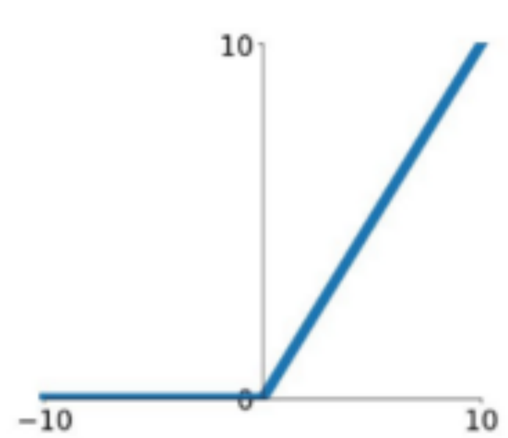
tanh

$\tanh(x)$



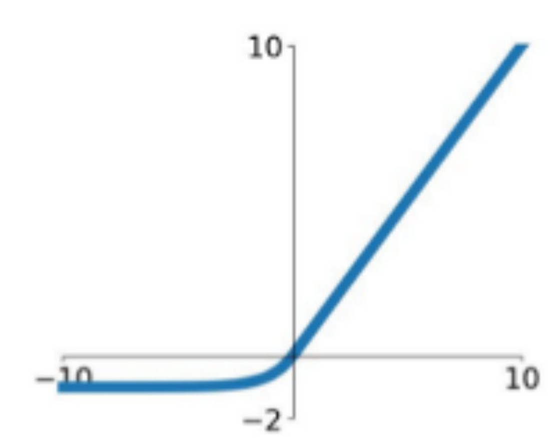
ReLU

$\max(0, x)$



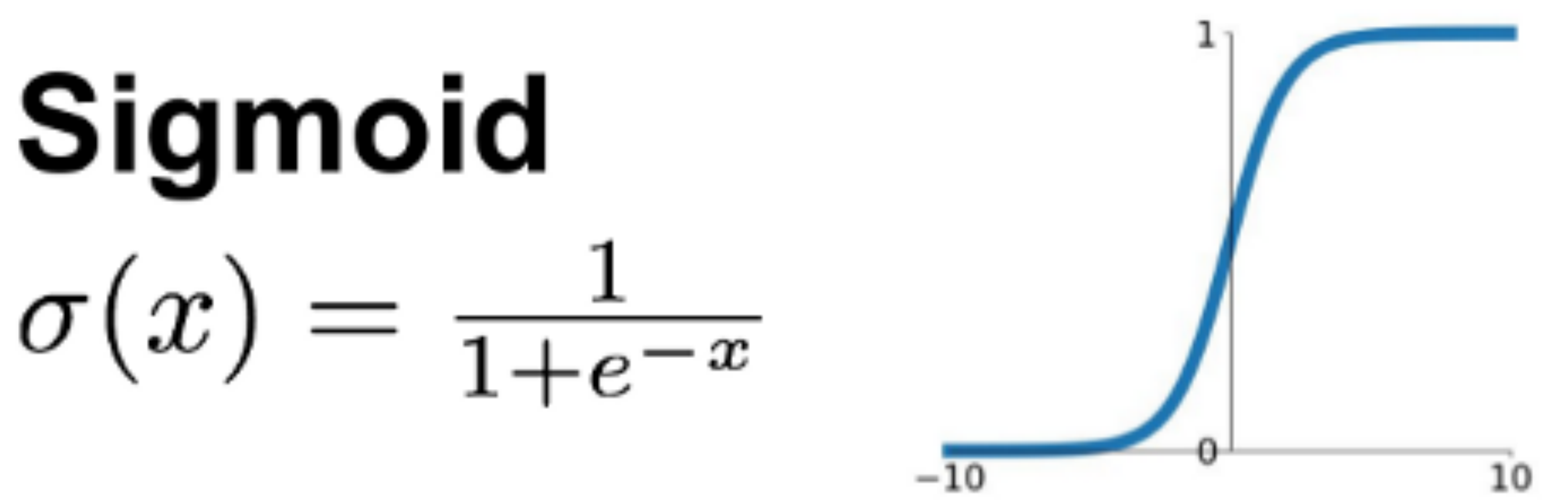
ELU

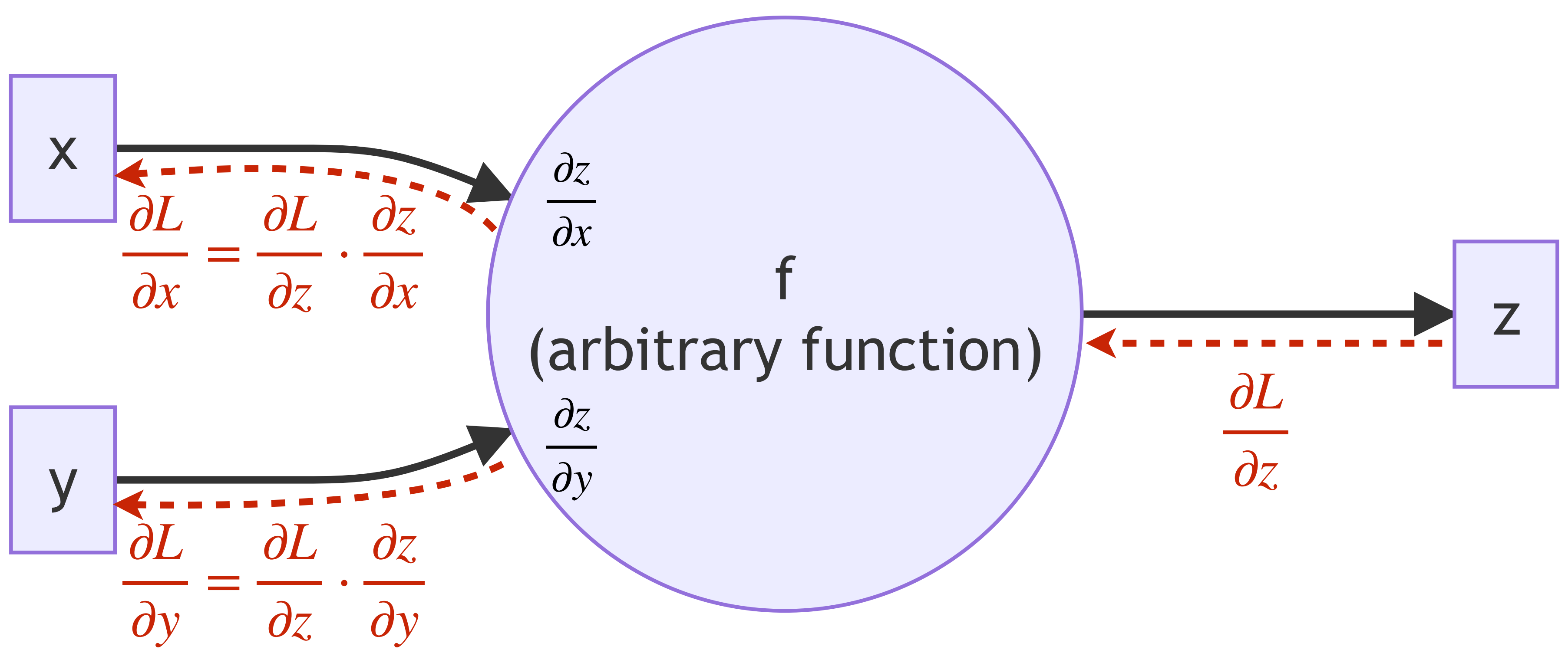
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



and many more ...

- **Example: sigmoid**
  - Local gradient $\frac{\partial \sigma}{\partial x} \equiv \sigma'$ yields asymptotically vanishing behavior
    - ▻ $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$
    - ▻ Gradient vanishes for small and large x
  - Two possible solutions:
    a) Manually enforce $x \in [-2,2]$ (keeps gradient above 0.1)
       - Not trivial since x is usually scalar product $W_i^T \cdot x_i + b_i$
       - See "Numerical insights"
    b) Use different activation

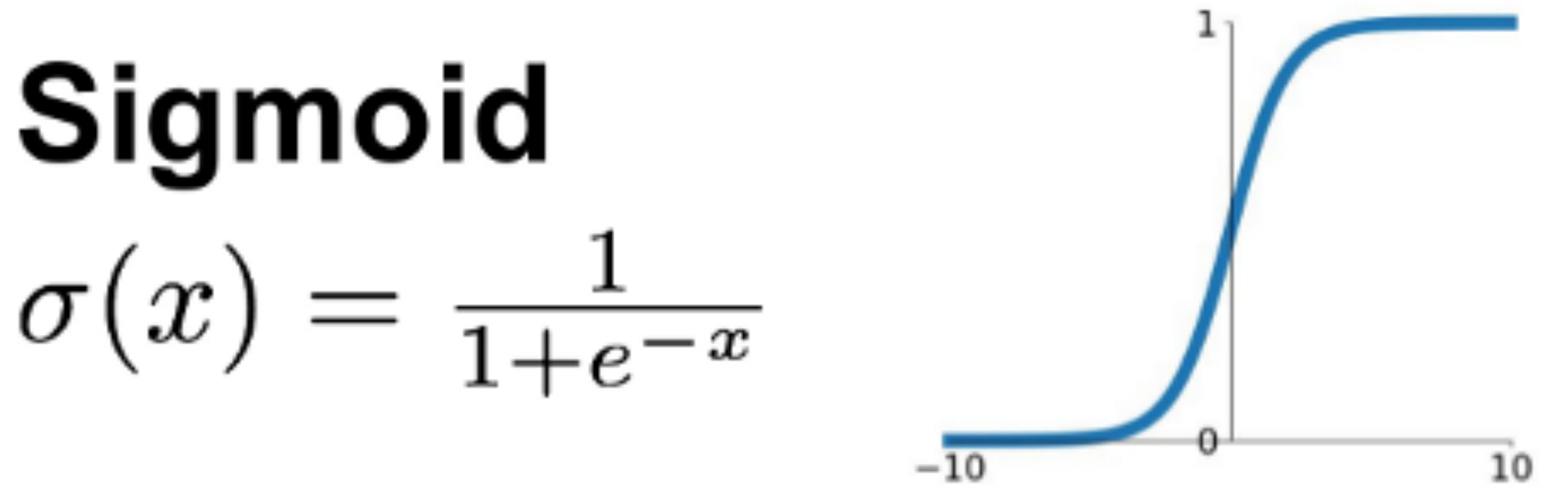


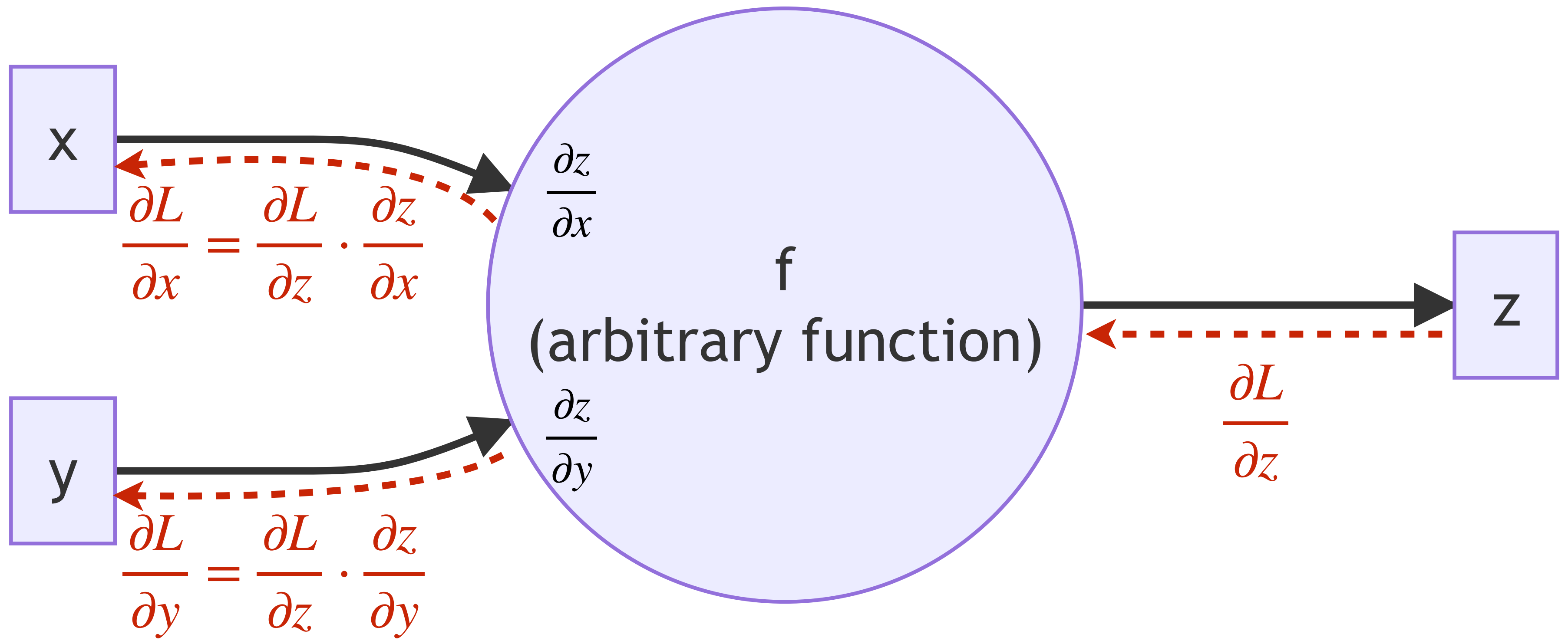


- **Example: sigmoid**
  - Local gradient $\frac{\partial\sigma}{\partial x} \equiv \sigma'$ yields asymptotically vanishing behavior
    - ▹ $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$
    - ▹ Gradient vanishes for small and large x
  - Two possible solutions:

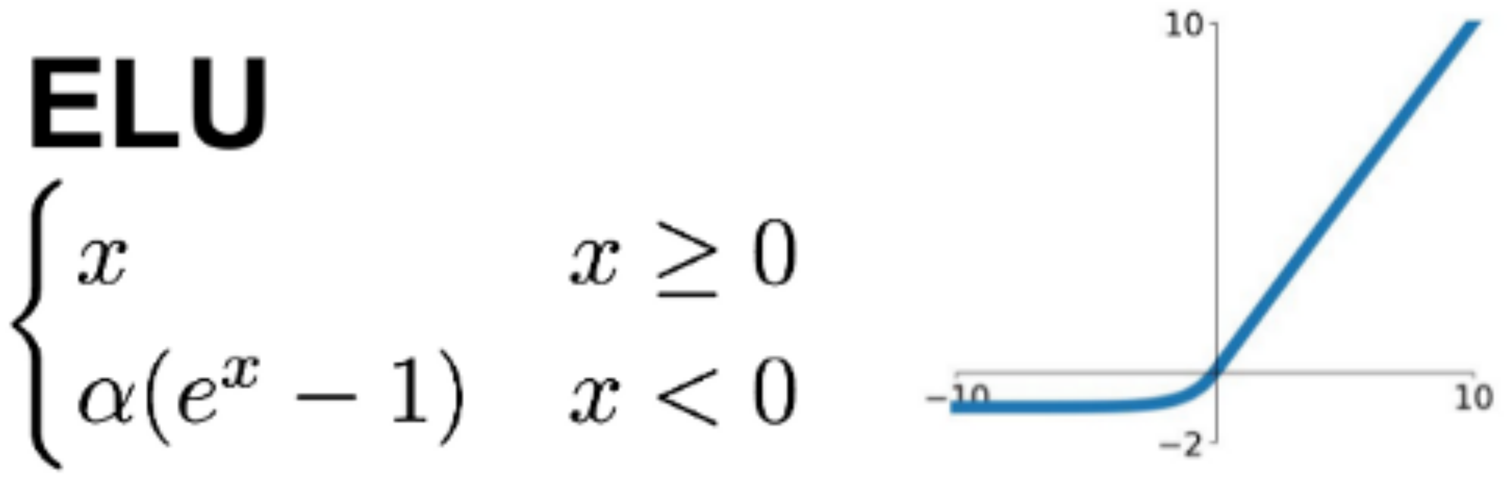
    a) Manually enforce $x \in [-2,2]$ (keeps gradient above 0.1)
       - Not trivial since x is usually scalar product $W_i^T \cdot x_i + b_i$
       - See "Numerical insights"

    b) Use different activation



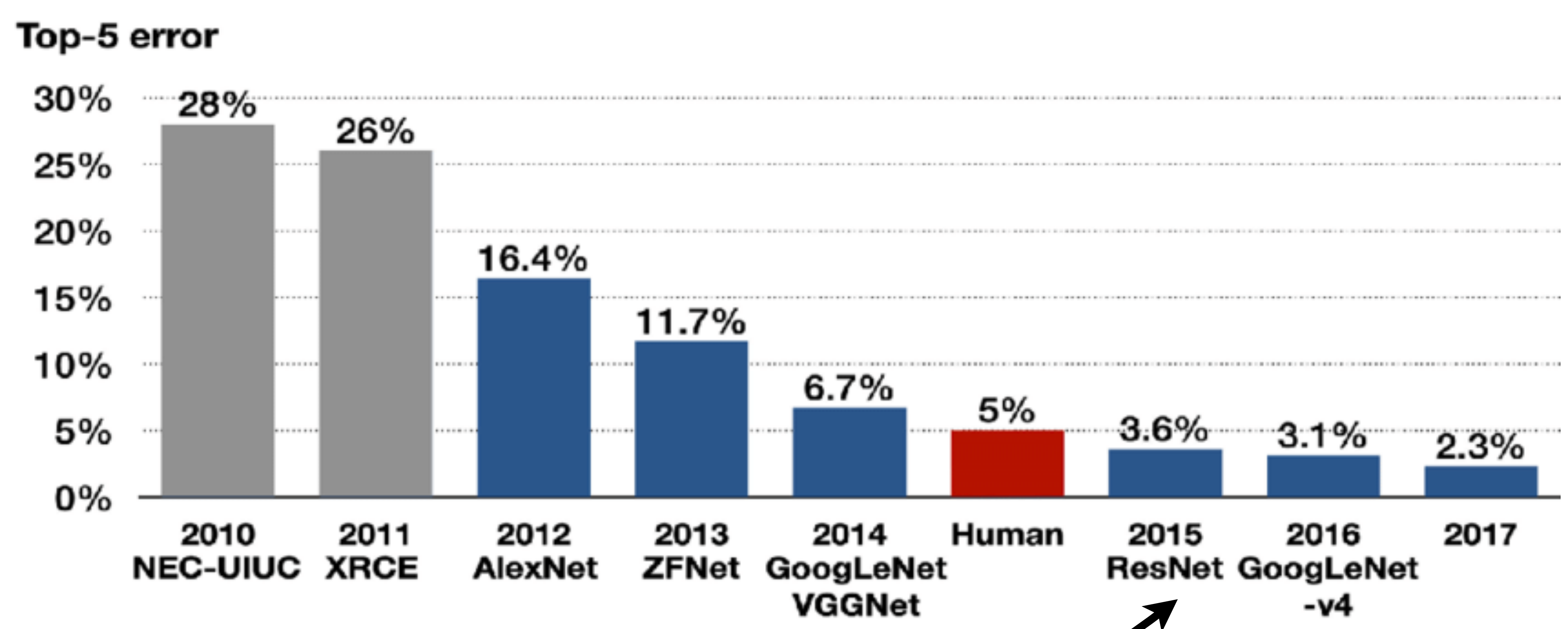


- Better: Re**LU**/ E**LU** / ...
  - Gradient **always** present (in fact, 1) for x ≥ 0
  - ReLU: unit dead once x < 0 (but can be desired, see CNNs)
  - ELU: units can recover over time after x < 0, accelerated by α

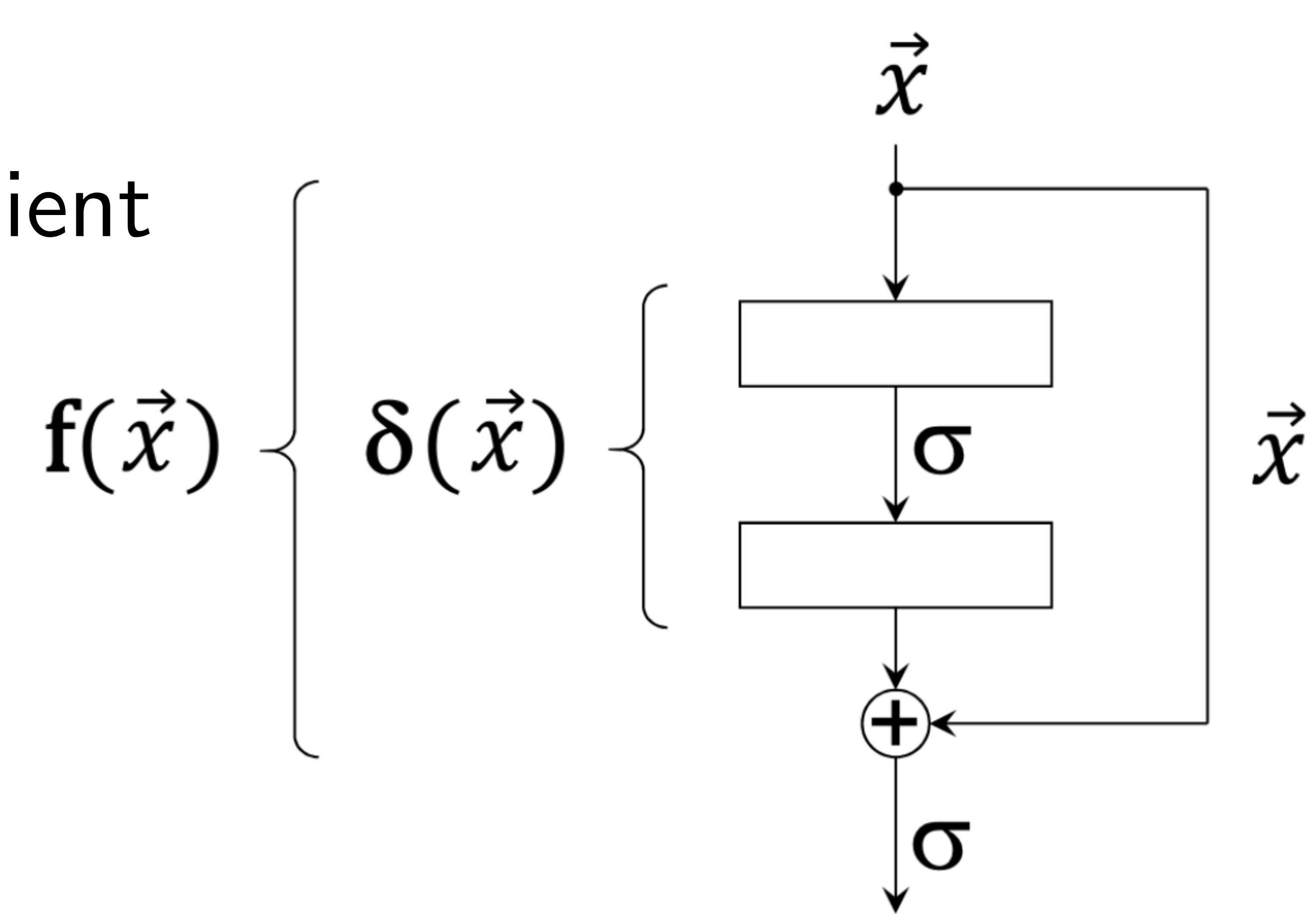


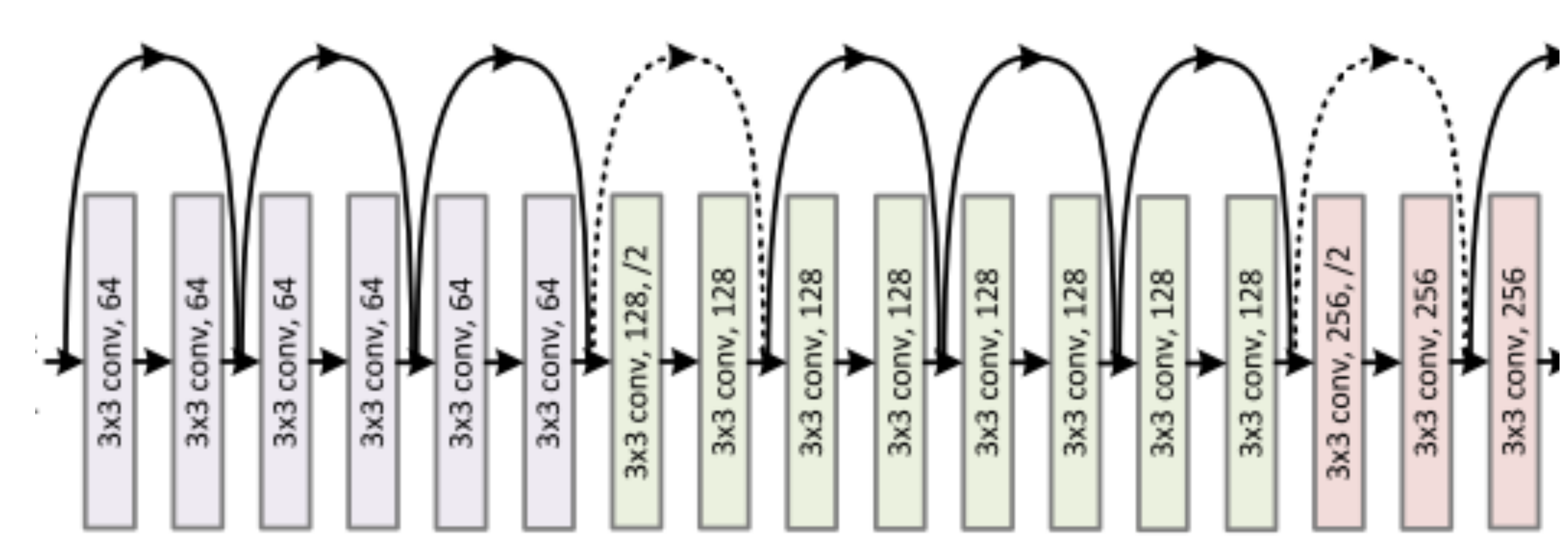
Beyond FCNs: ResNet

- **ImageNet (Large Scale Recognition) challenge**
  - Image recognition challenge that was driving the advancement of ML research
  - 1.3M training images, 0.1M test images, 1000(!) classes



- **ResNet** became the first architecture to beat human recognition performance (7 years ago)
  - Residual learning → predict target & add additional layers to learn residual differences
  - $f(\vec{x}) = \vec{x} + \delta(\vec{x})$
  - Benefits convergence and fast gradient propagation through deep NNs!

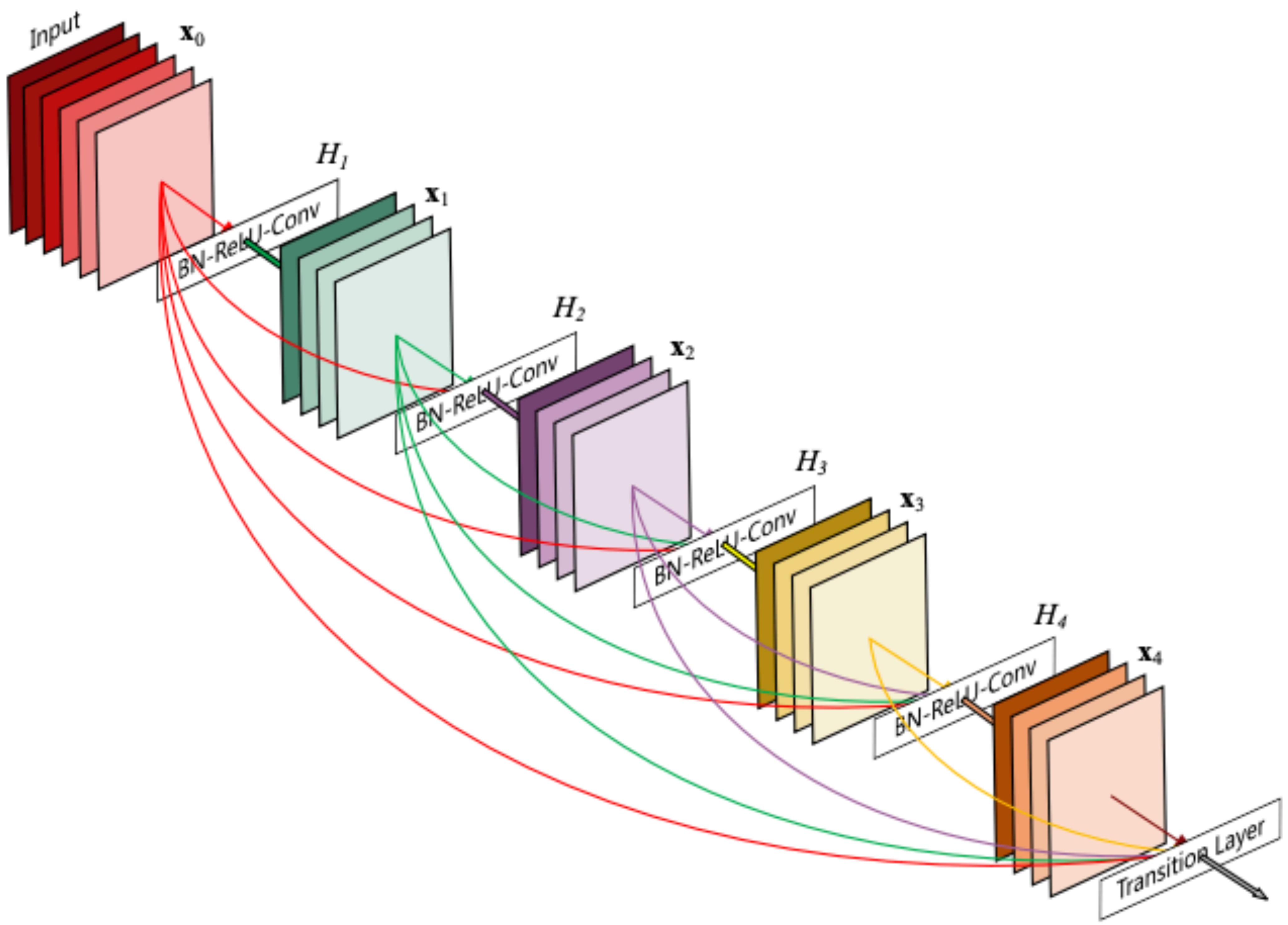


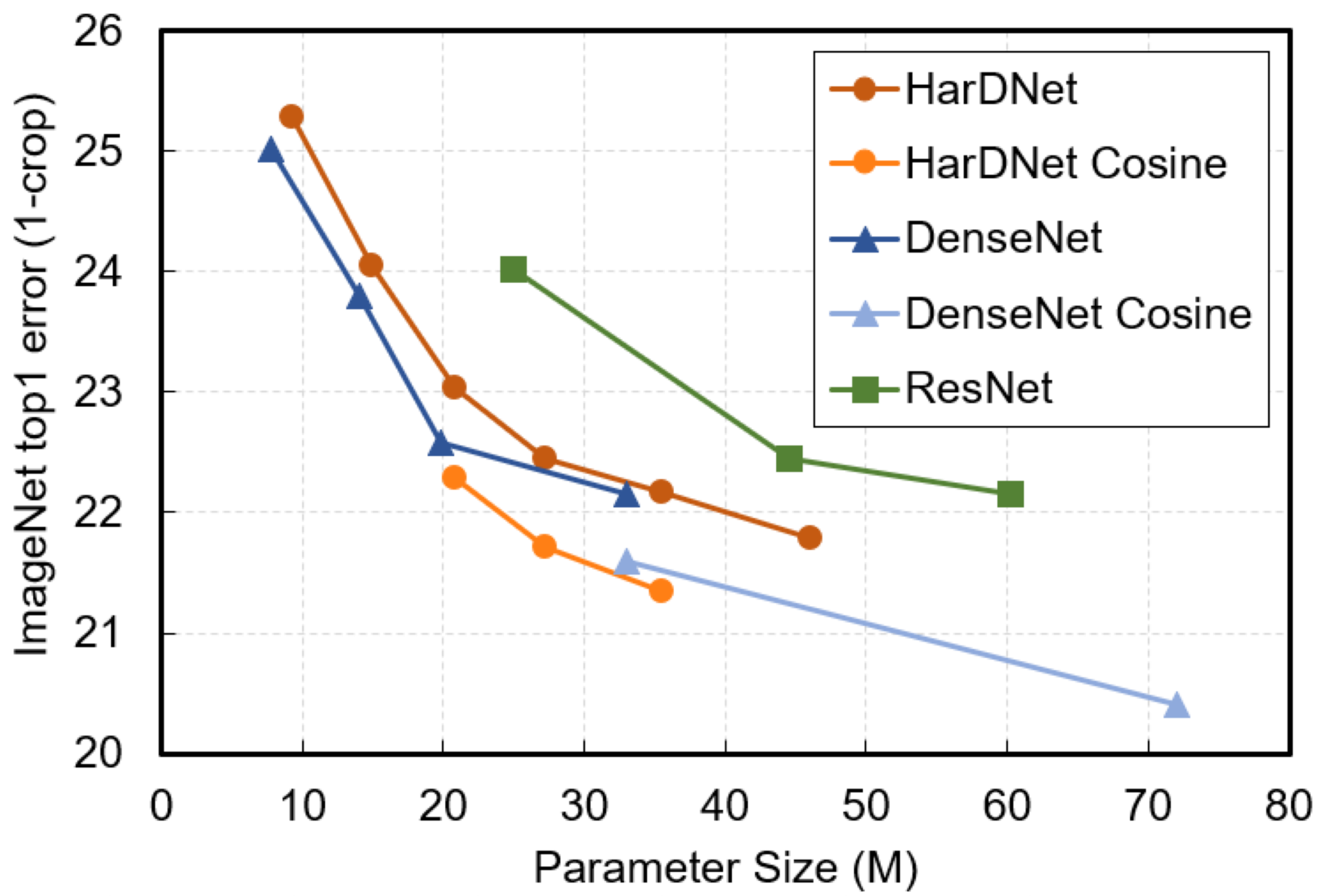


Many layers

from "Deep learning in Physics Research", Erdmann *et al.*

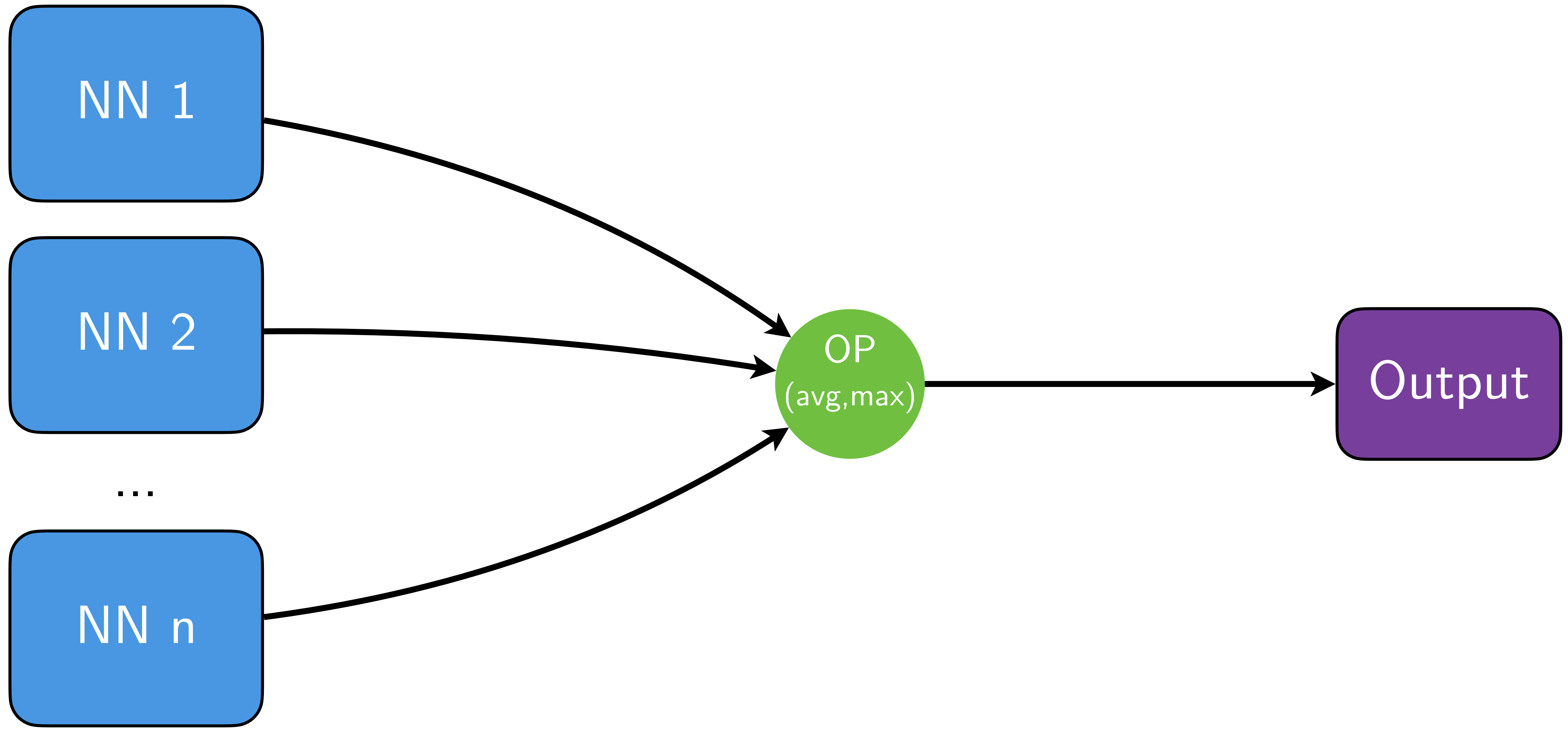
- **DenseNet**
  - Pass on layer outputs as additional inputs to all subsequent layers
  - Less weights required to reach equal performance compared to ResNet





from arXiv:1909.00948

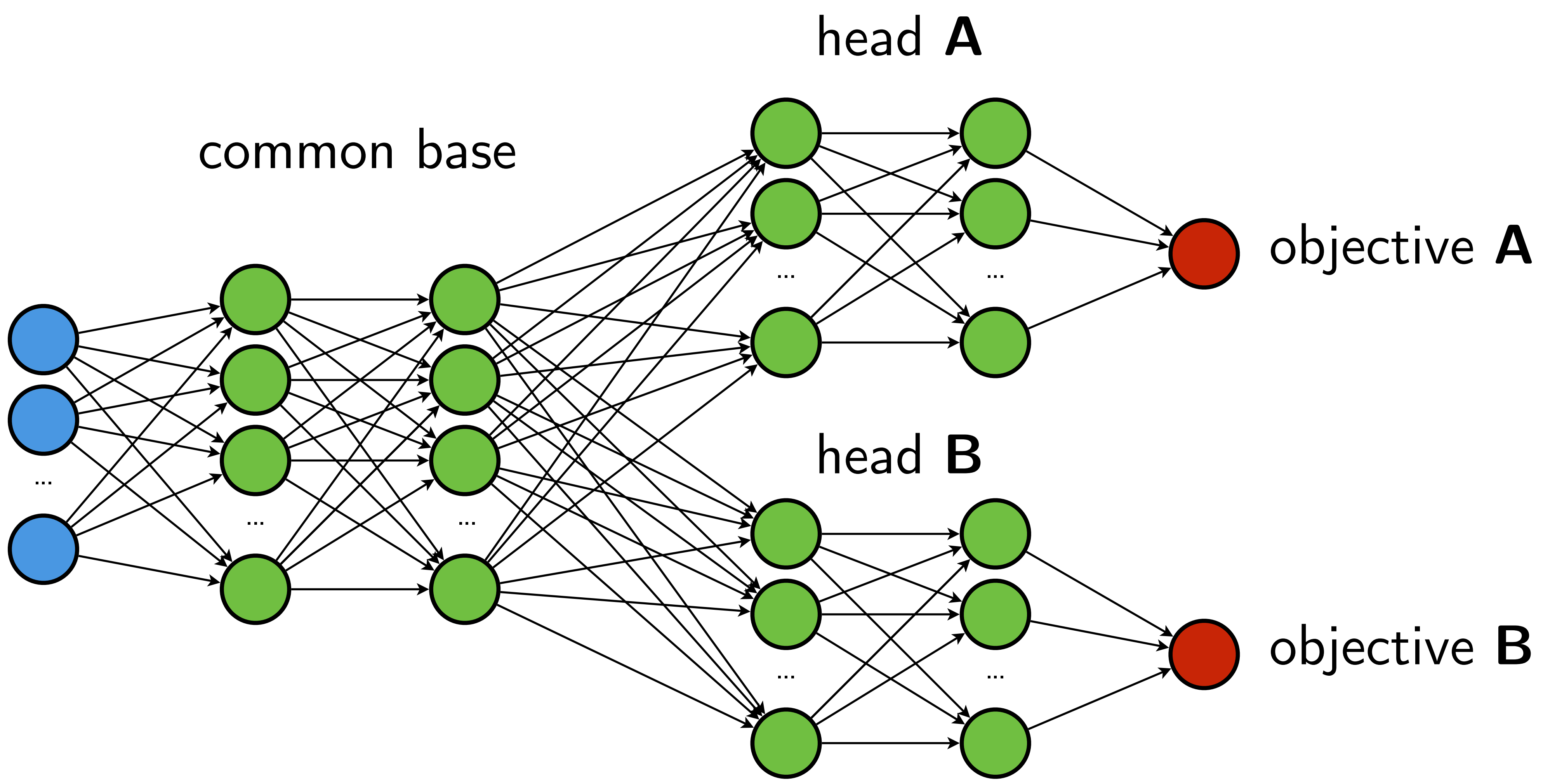
- The predictive power of multiple networks can be combined



- **Benefits**
  - Performance usually improved (many Kaggle challenges won this way)
  - Less prone to fluctuations in input data, that a single NN might have picked up

- **Variants**
  a) Ensemble can be trained as one, with *different initial weights* per NN
  b) Same as a) *plus* use different subsets of data

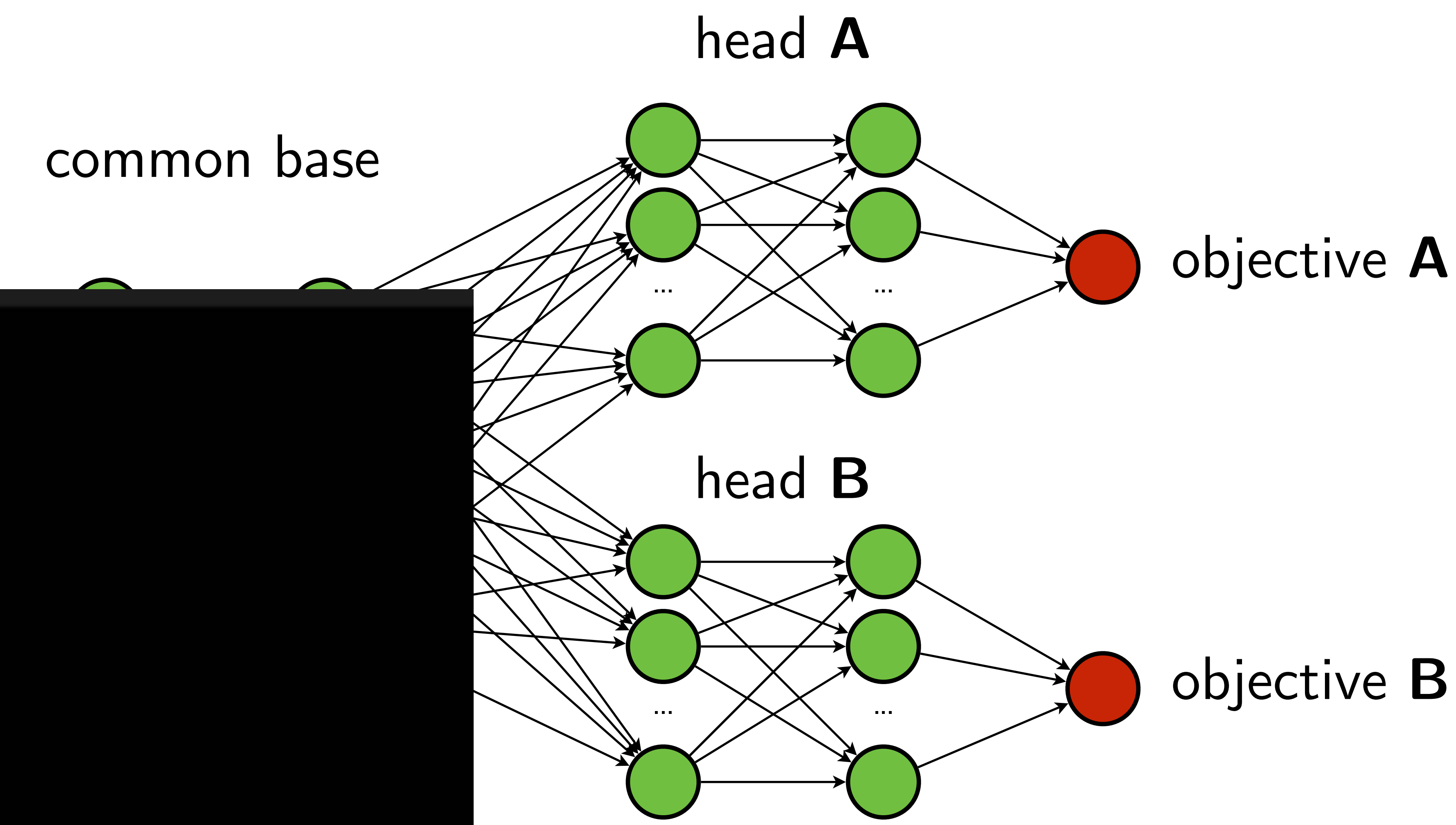
- **A network can serve multiple tasks**
  - Common base layers
  - Specific "heads" per output
  - Multiple objectives to be connected through loss function
    - $L = L_A + \lambda \cdot L_B$
    - λ balances importance of **A** and **B** to overall loss
  - Countless variants
    - E.g. for >2 tasks, add several common bases

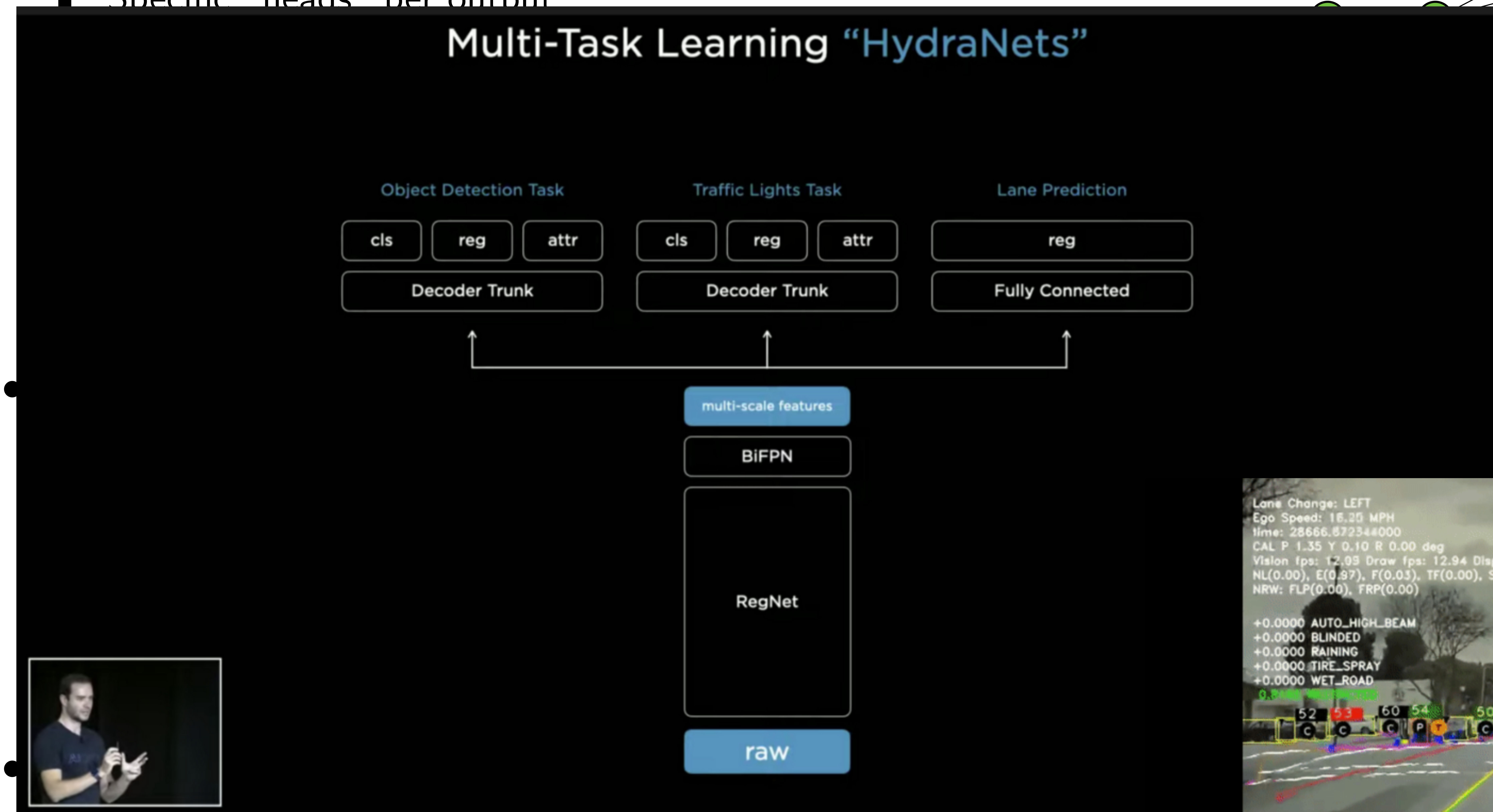


- Allows performing several tasks at once while profiting from
  - Single training process
  - Joined learning of inner representations important to both tasks
  - Constructive mutual influence
    - Updates propagated back to common base from A can improve B in next forward pass
    - Effectively, more ground truth information is used

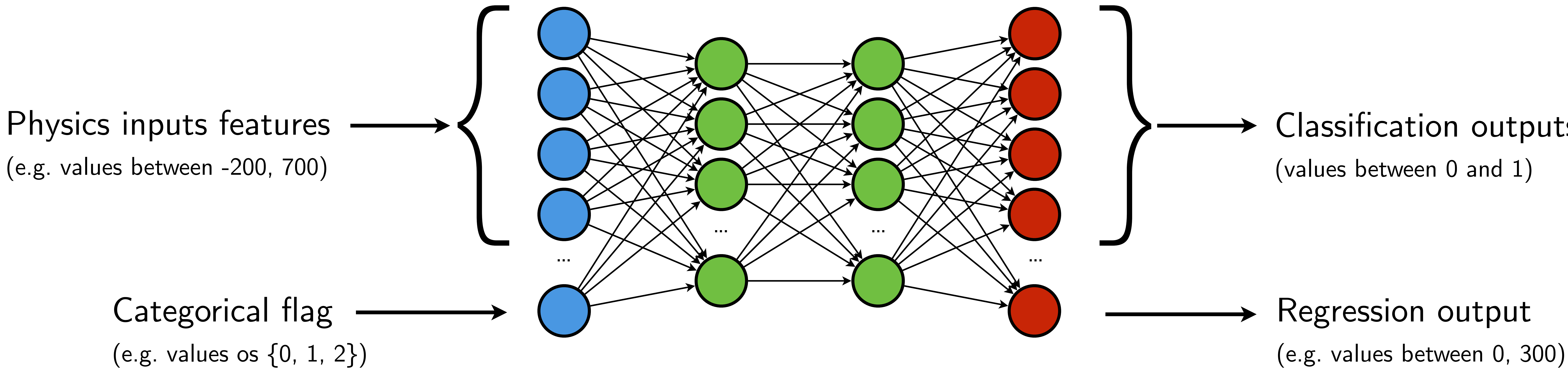
- **Used in real-life applications** (e.g. self-driving vehicles)

- **A network can serve multiple tasks**
  - Common base layers
  - Specific "heads" per output

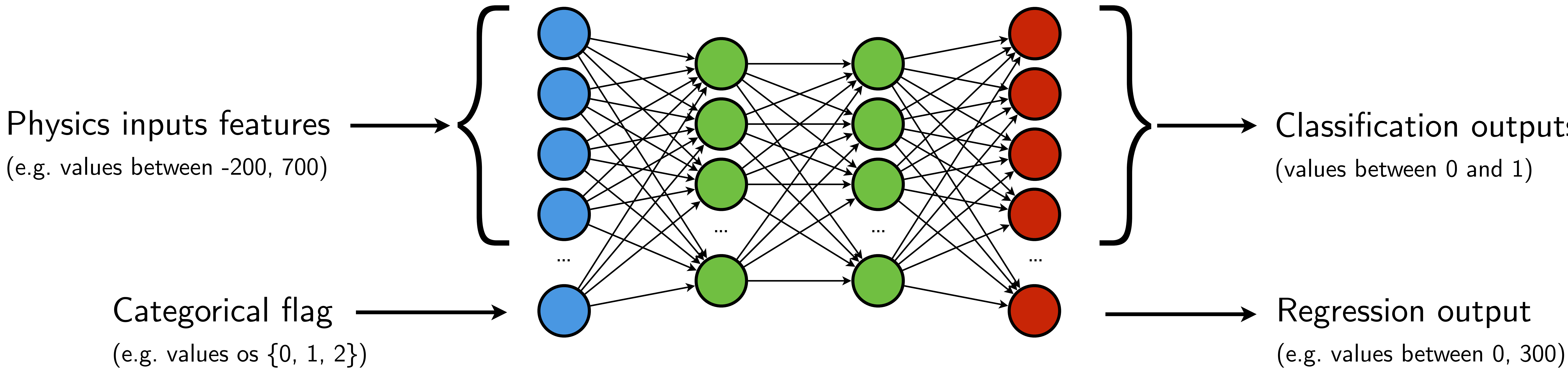




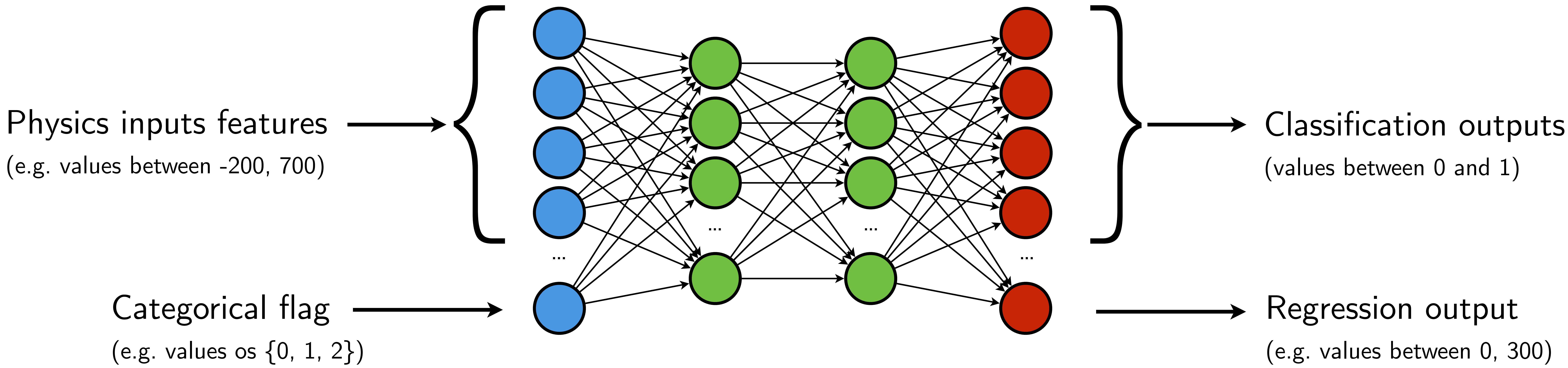
2. Numerical insights & considerations



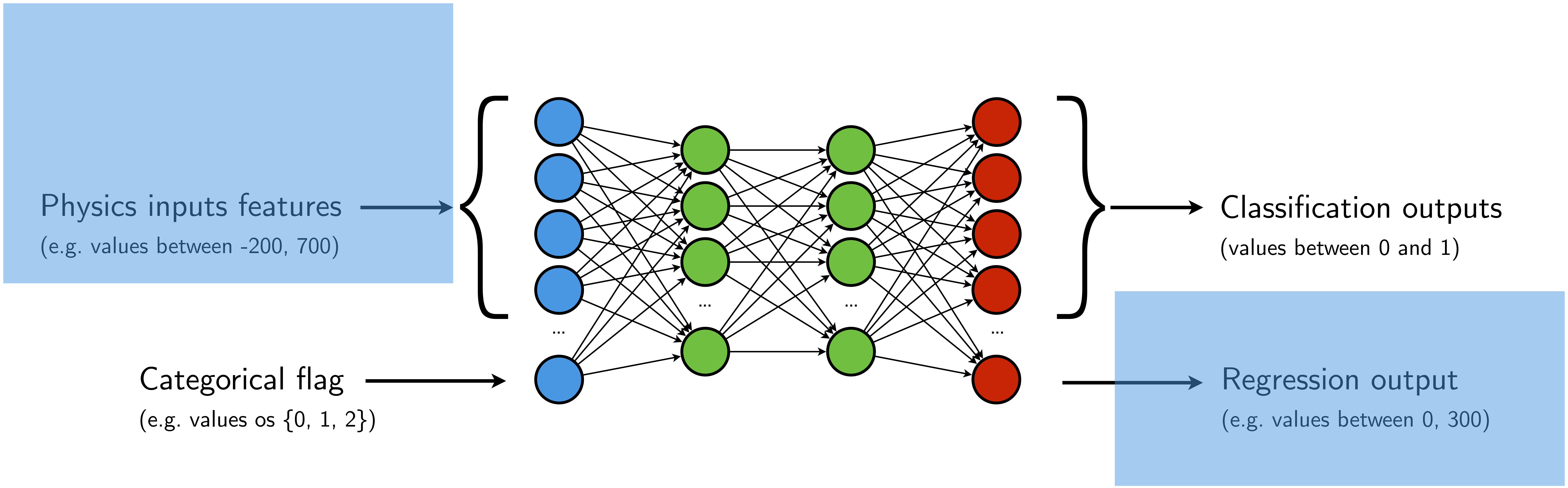
- **Questions**
  - What happens when large input features are fed into the network?
  - What happens during back-propagation when large outputs are expected?



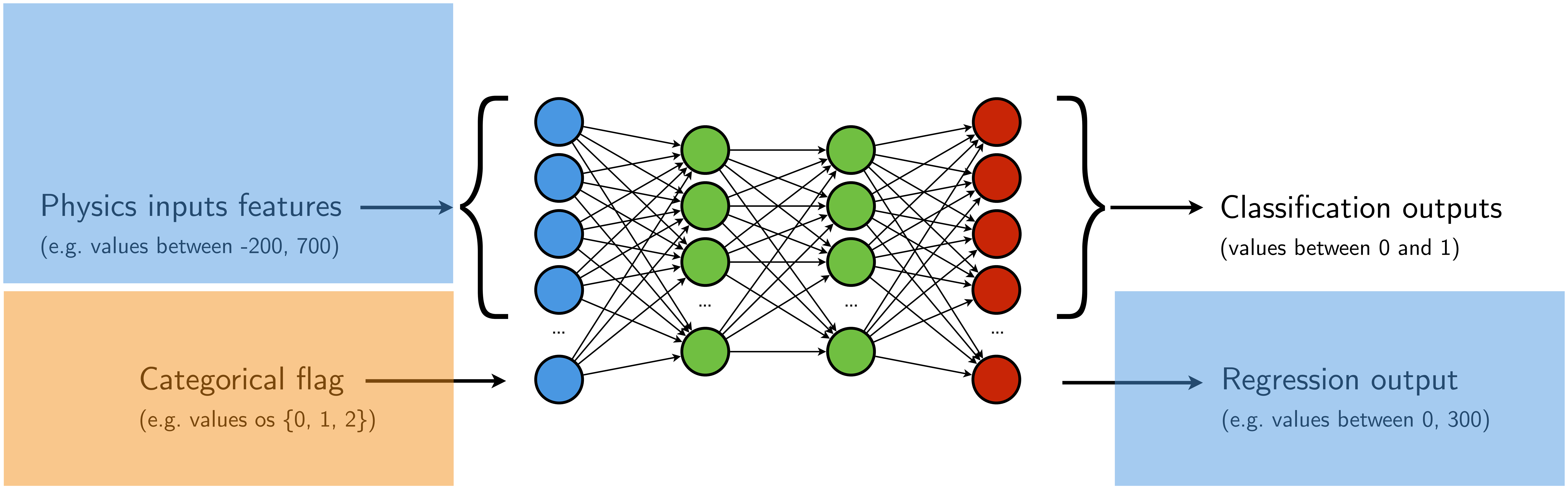
- **Questions**
  - What happens when large input features are fed into the network?
    - Inputs to activations might fall into regimes with vanishing gradient / dead units!
  - What happens during back-propagation when large outputs are expected?



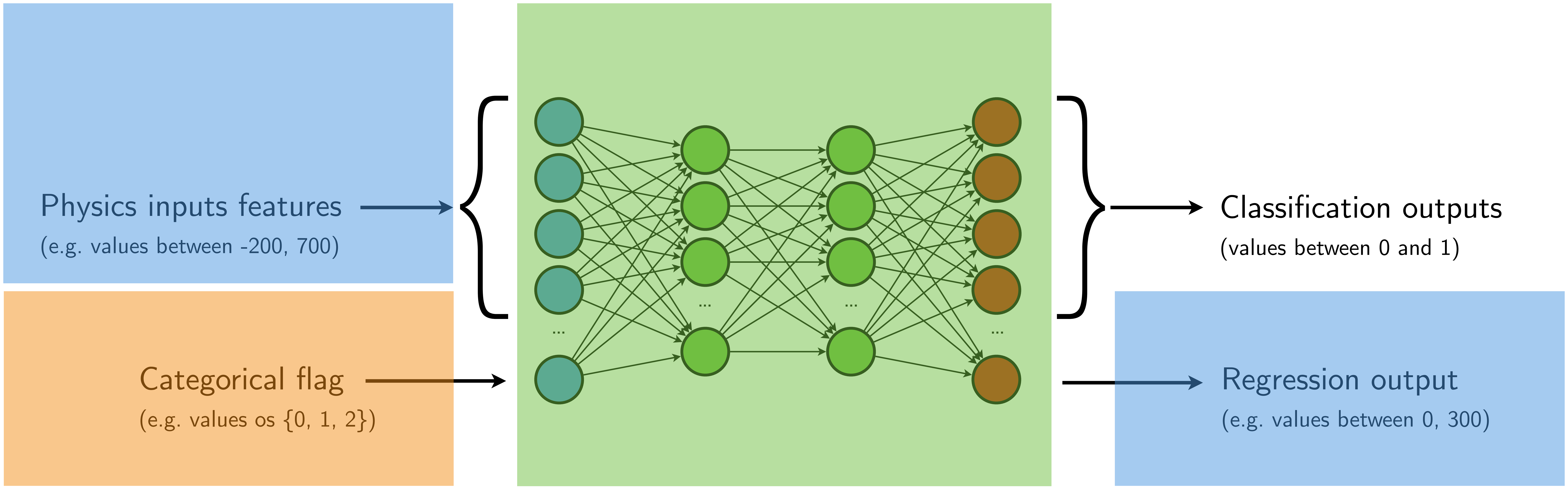
- **Questions**
  - What happens when large input features are fed into the network?
    - Inputs to activations might fall into regimes with vanishing gradient / dead units!
  - What happens during back-propagation when large outputs are expected?
    - Weights are likely increased right-to-left, to reach large regression output
    - Weights are likely increased left-to-right, to bring inputs down to *usable* ranges



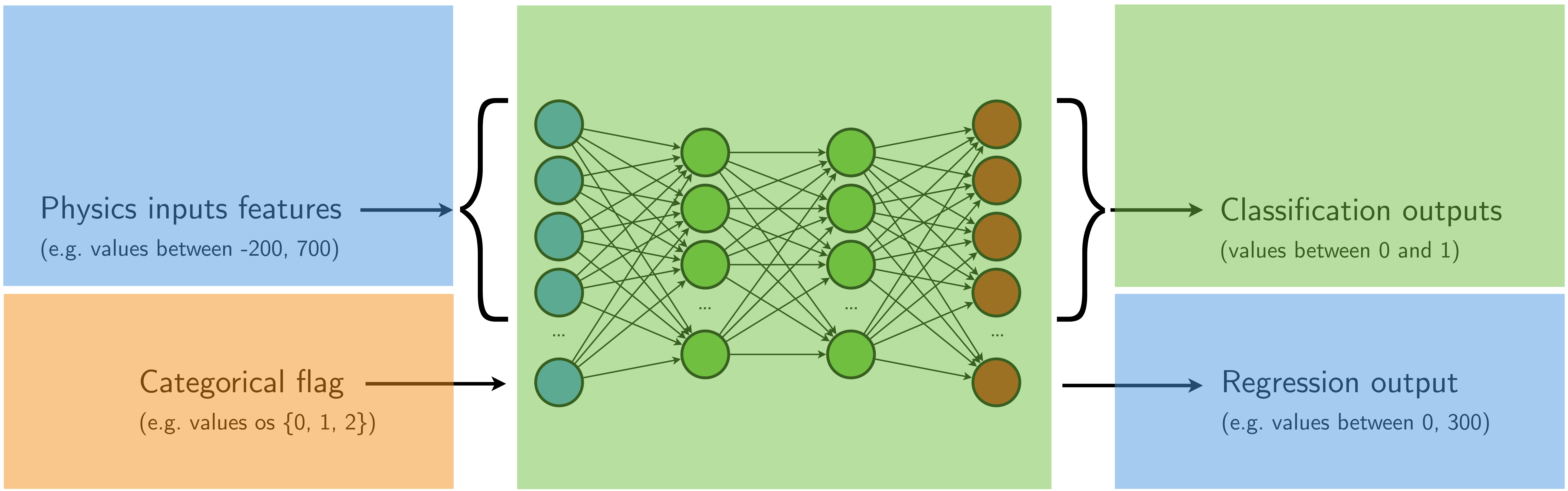
- **Questions**
  - What happens when large input features are fed into the network?
    - Inputs to activations might fall into regimes with vanishing gradient / dead units!
  - What happens during back-propagation when large outputs are expected?
    - Weights are likely increased right-to-left, to reach large regression output
    - Weights are likely increased left-to-right, to bring inputs down to *usable* ranges



- **Questions**
  - What happens when large input features are fed into the network?
    - Inputs to activations might fall into regimes with vanishing gradient / dead units!
  - What happens during back-propagation when large outputs are expected?
    - Weights are likely increased right-to-left, to reach large regression output
    - Weights are likely increased left-to-right, to bring inputs down to *usable* ranges

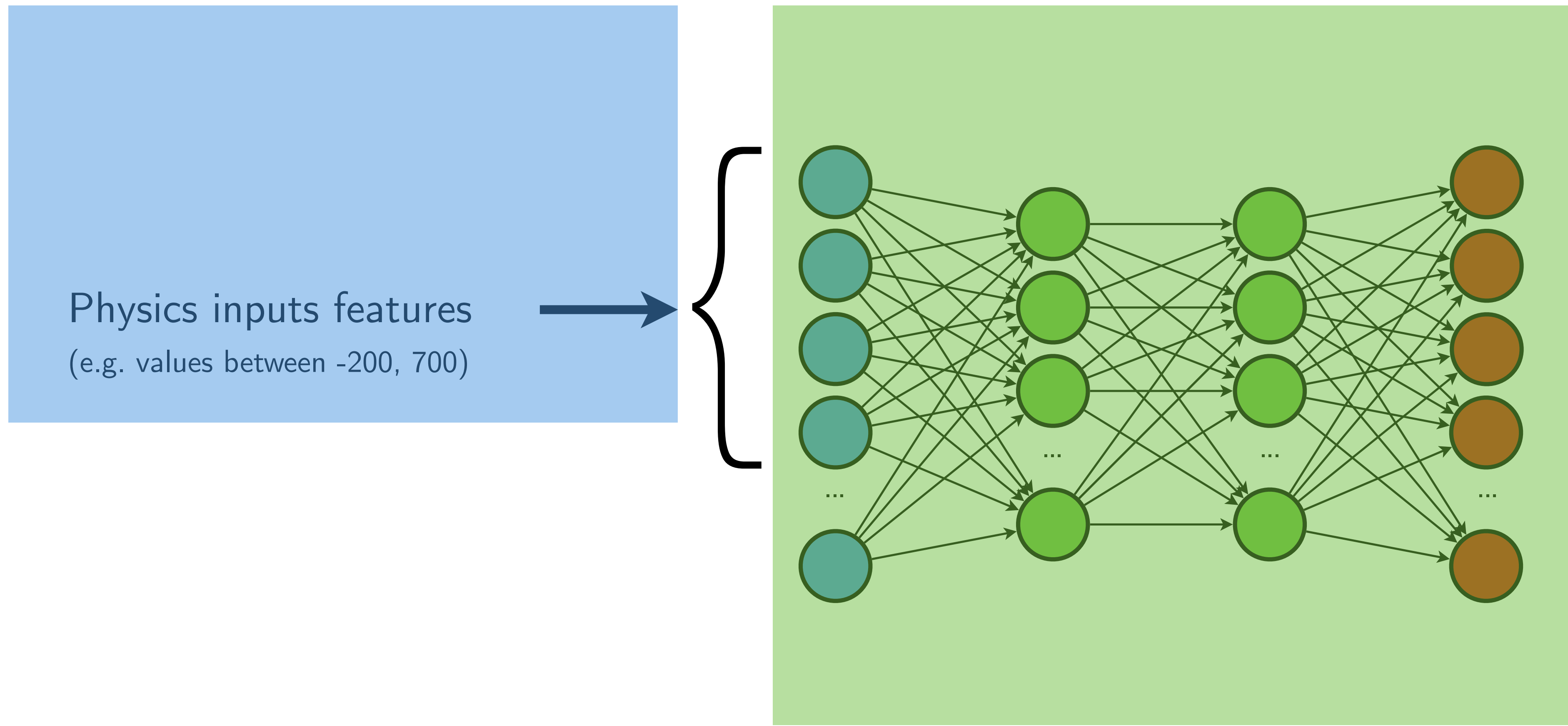


- **Questions**
  - What happens when large input features are fed into the network?
    - Inputs to activations might fall into regimes with vanishing gradient / dead units!
  - What happens during back-propagation when large outputs are expected?
    - Weights are likely increased right-to-left, to reach large regression output
    - Weights are likely increased left-to-right, to bring inputs down to *usable* ranges

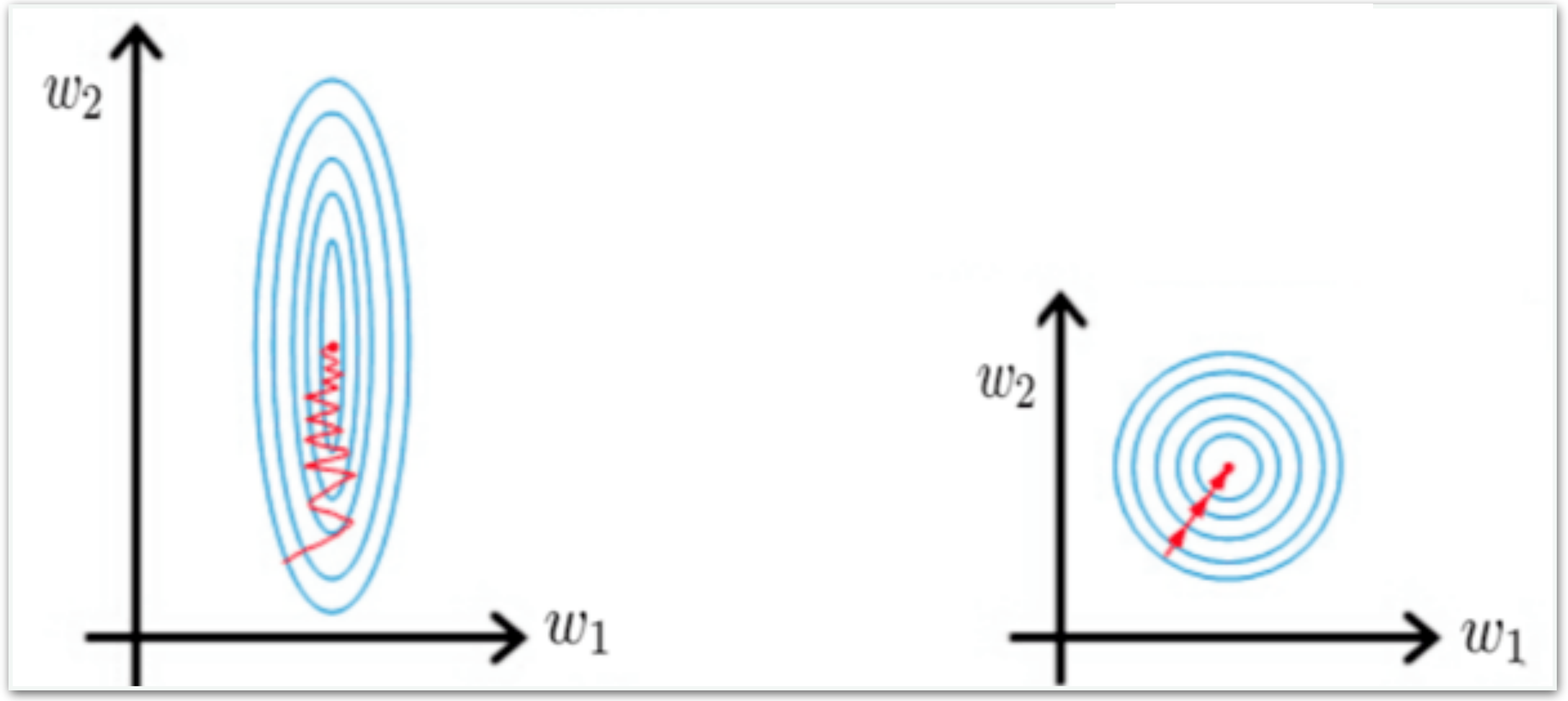


- **Questions**
  - What happens when large input features are fed into the network?
    - ▻ Inputs to activations might fall into regimes with vanishing gradient / dead units!
  - What happens during back-propagation when large outputs are expected?
    - ▻ Weights are likely increased right-to-left, to reach large regression output
    - ▻ Weights are likely increased left-to-right, to bring inputs down to *usable* ranges

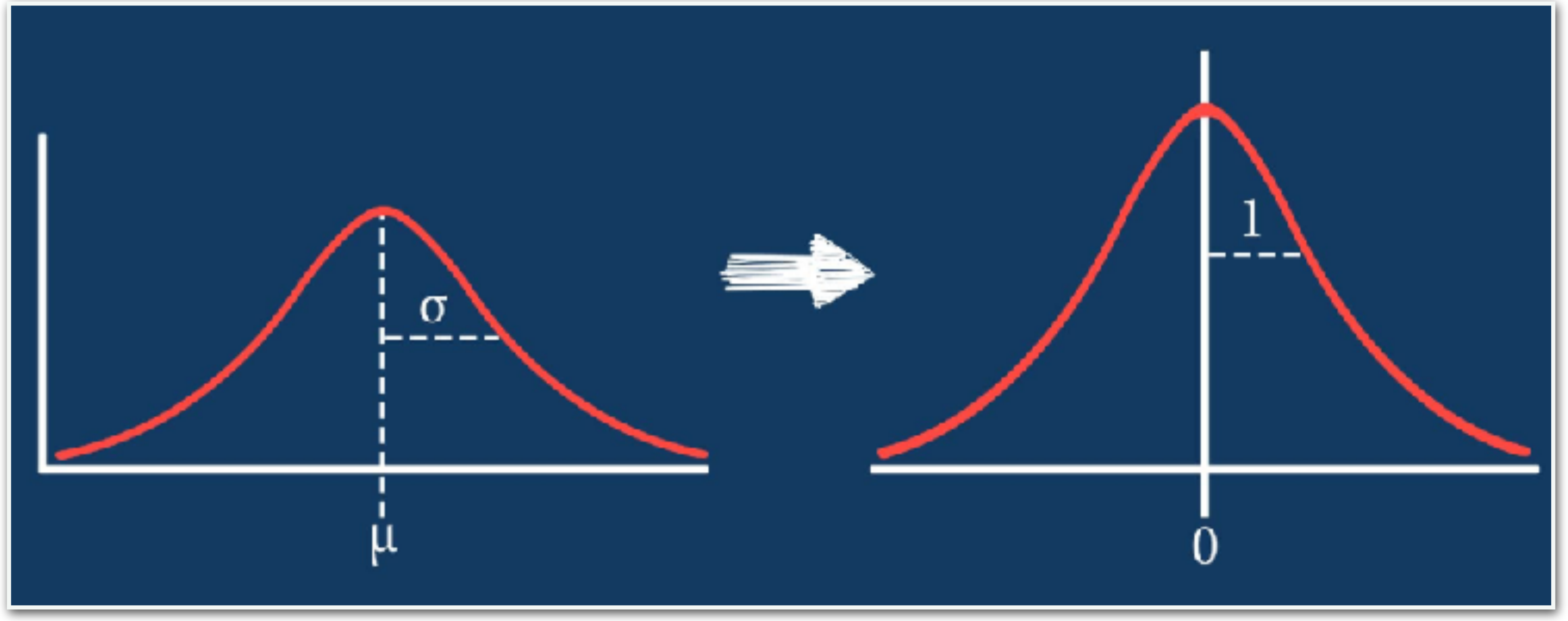
- **Goal**
  - Transform "physics" input features such that their range fits the numerical domain of the network (typically $[-1,1]$) **while preserving all corrections**



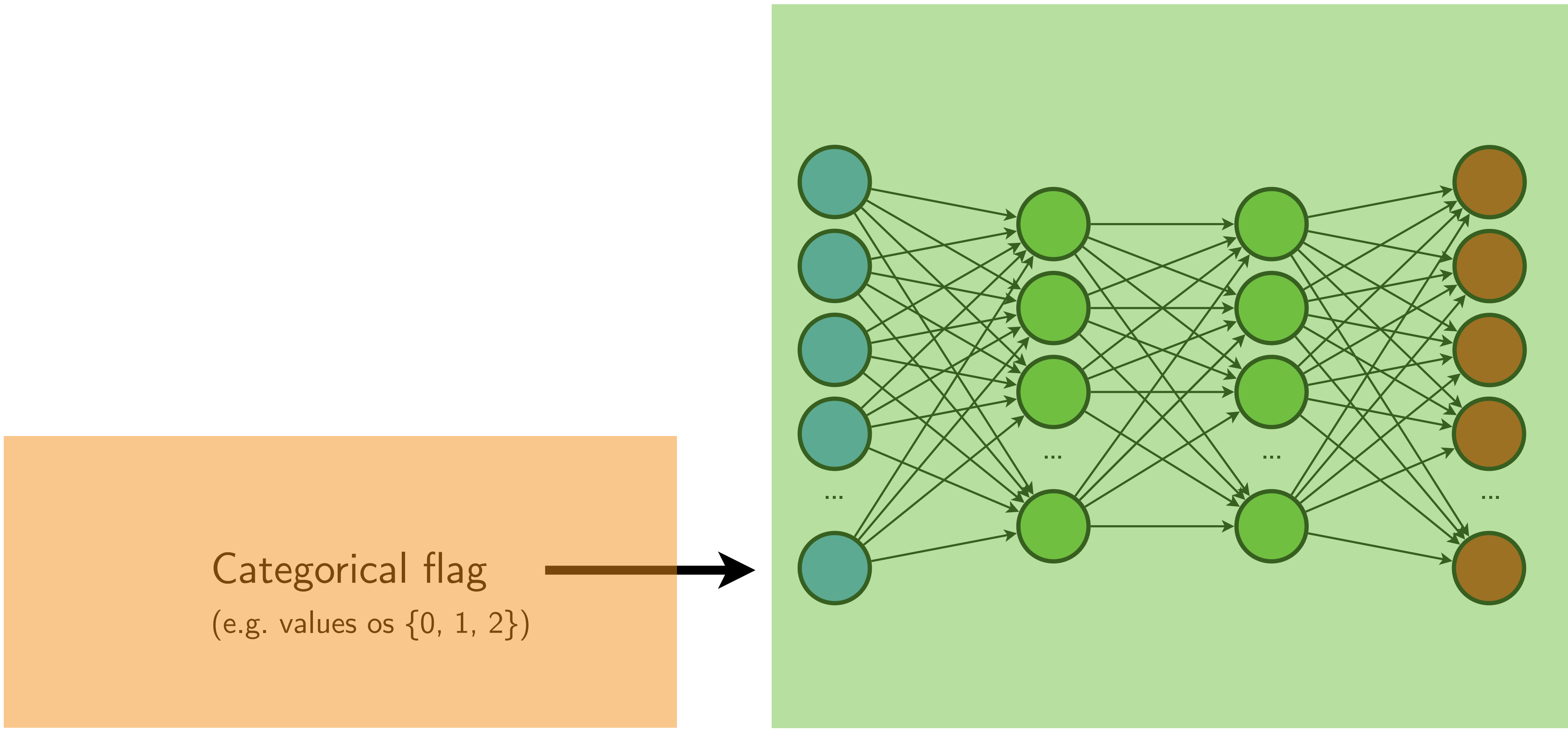
- **Benefits**
  - Vanishing gradients less likely
  - Speed-up due to homogeneous loss "landscape"



- **Simple "shift & scale" approach**
  - For each feature f, apply $f \to f' = \dfrac{f-\mu}{\sigma}$
    - μ: mean
    - σ: $\sqrt{}$variance
  - To be performed once for each feature before training **and** needs to be applied to inputs **before evaluation**
  - **Hint**: create initial layer with constant, non-trainable scaling parameters per feature to avoid having to remember those values



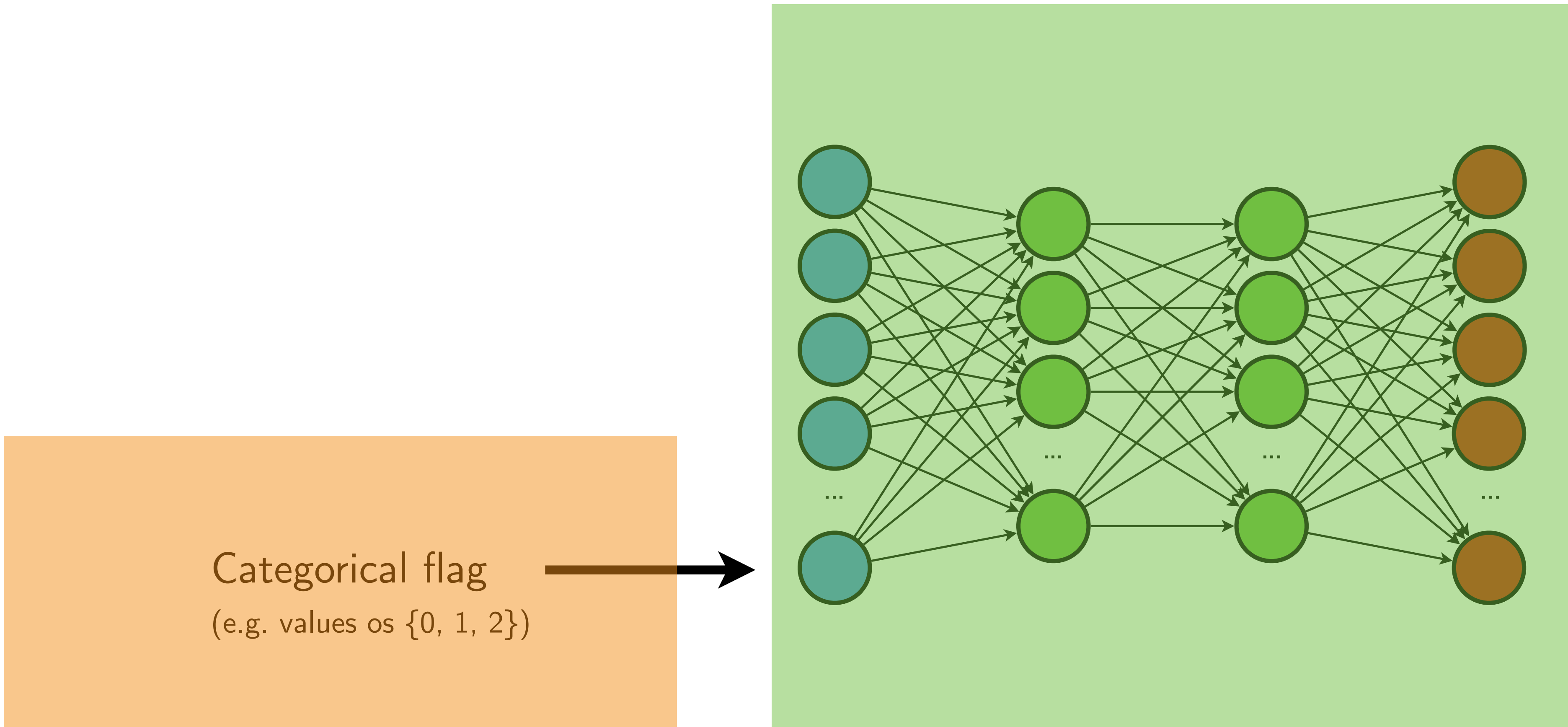
- Categorical flags constitute a common source of input features
  - Example: gender → {0, 1, 2, ...} (flag)
  - **Not** an example: age → [0, 99] (simple integer-value input)
  - **Difference**
    - ▻ Adjacency between two categorical values does **not** carry additional information: "0" is equally far apart from "1" than "2"
    - ▻ Adjacency between integer values **does**: age "50" is closer "49" than "10"
  - → Categorical flags require further treatment as numerical proximity matters to networks!



- **One-hot encoding**
  - Encode flags with (e.g.) three realizations through three separate inputs, each being either 0 or 1
    - ▻ flag 0 → (1, 0, 0)
    - ▻ flag 1 → (0, 1, 0)
    - ▻ flag 2 → (0, 0, 1)
  - Bit-like mixtures such as (0, 1, 1) *can* also work but rather use embedding layers which optimize this

Categorical feature embedding (2)

- Categorical flags constitute a common source of input features
  - Example: gender → {0, 1, 2, ...} (flag)
  - **Not** an example: age → [0, 99] (simple integer-value input)



- **Embedding layers**
  - Useful in case of two or more categorical features whose values form the full "vocabulary"
  - Influenced by speech recognition where words are flags and inputs would be "*sentence length* x *vocabulary length*"
  - Instead
    1. Build random weight matrix shaped $N_{vocabulary}$ x N_{weight}
    2. Given N_f input flags, lookup indices in vocabulary
    3. Select N_w weights from matrix per input flag index
    4. Construct N_f x N_w matrix
    5. Flatten it to (N_f • N_w) vector and use it as input

Index	Word
0	are
1	you
2	ok
3	how

N_v

Index	Weights dim 1	Weights dim 2
0	0.1	0.3
1	-0.4	0.3
2	0.9	-1.0
3	0.7	0.4

N_w

"How are you?"

Indices

```
[3, 1, 2]
```

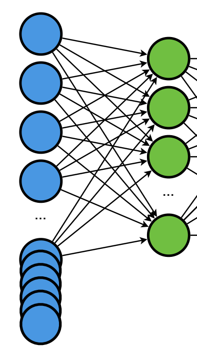
Select weights

```
[[ 0.7,  0.4],
 [-0.4,  0.3],
 [ 0.9, -1.0]]
```

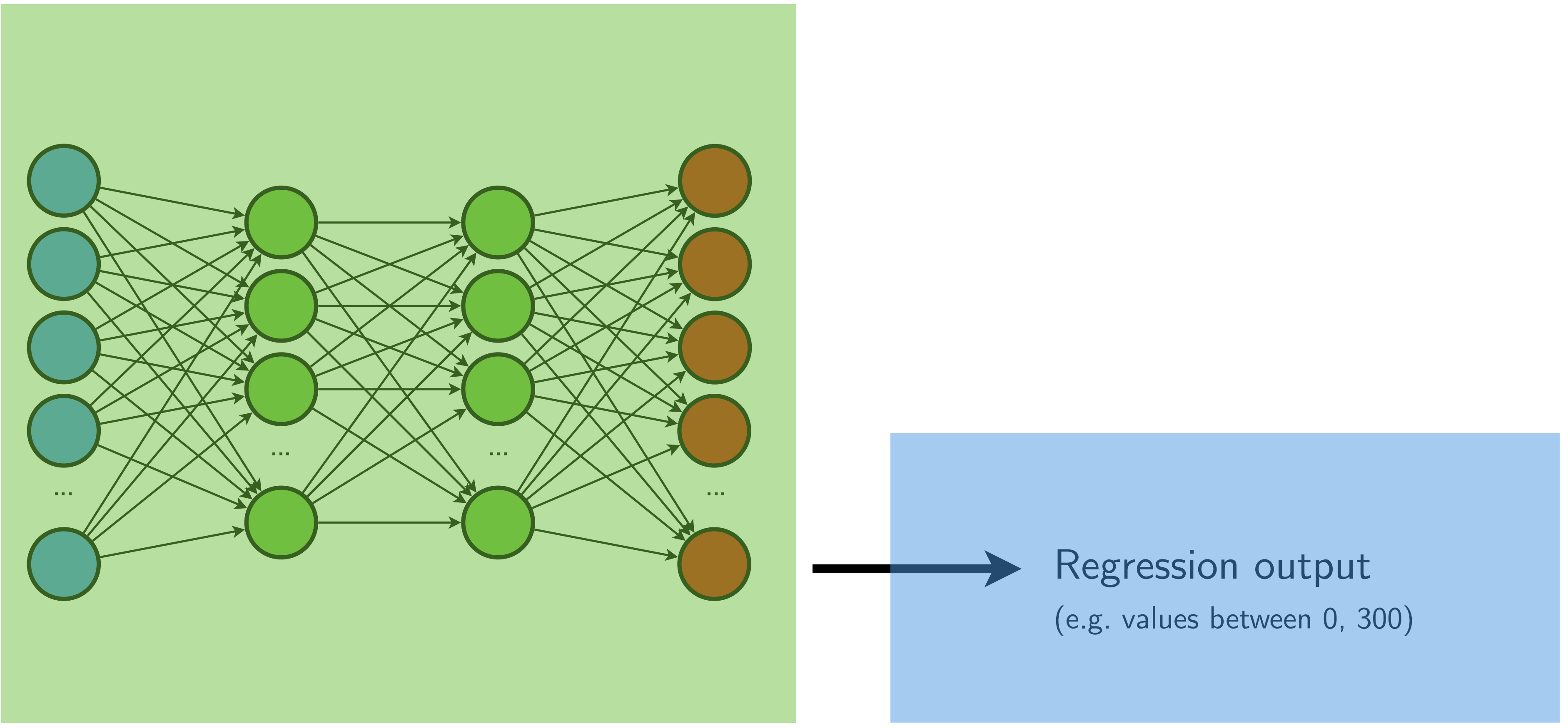
Flatten

```
[0.7, 0.4, -0.4,
 0.3, 0.9, -1.0]
```

→

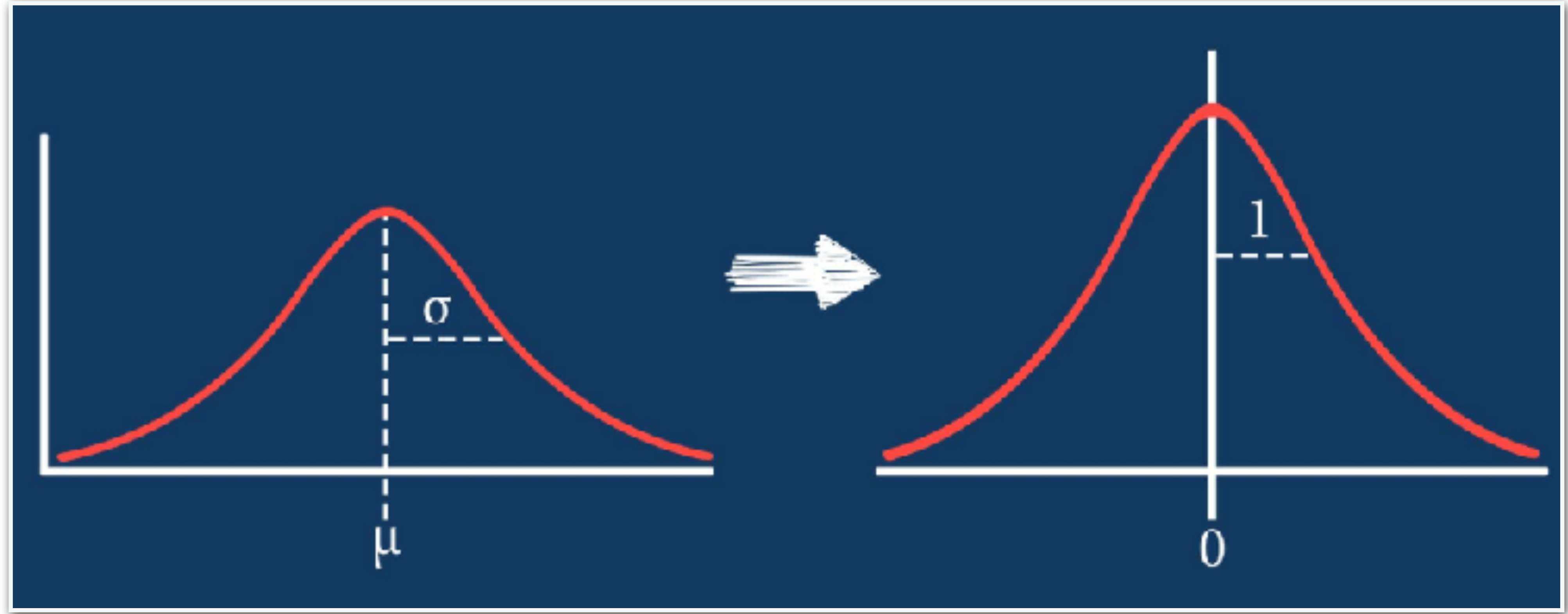


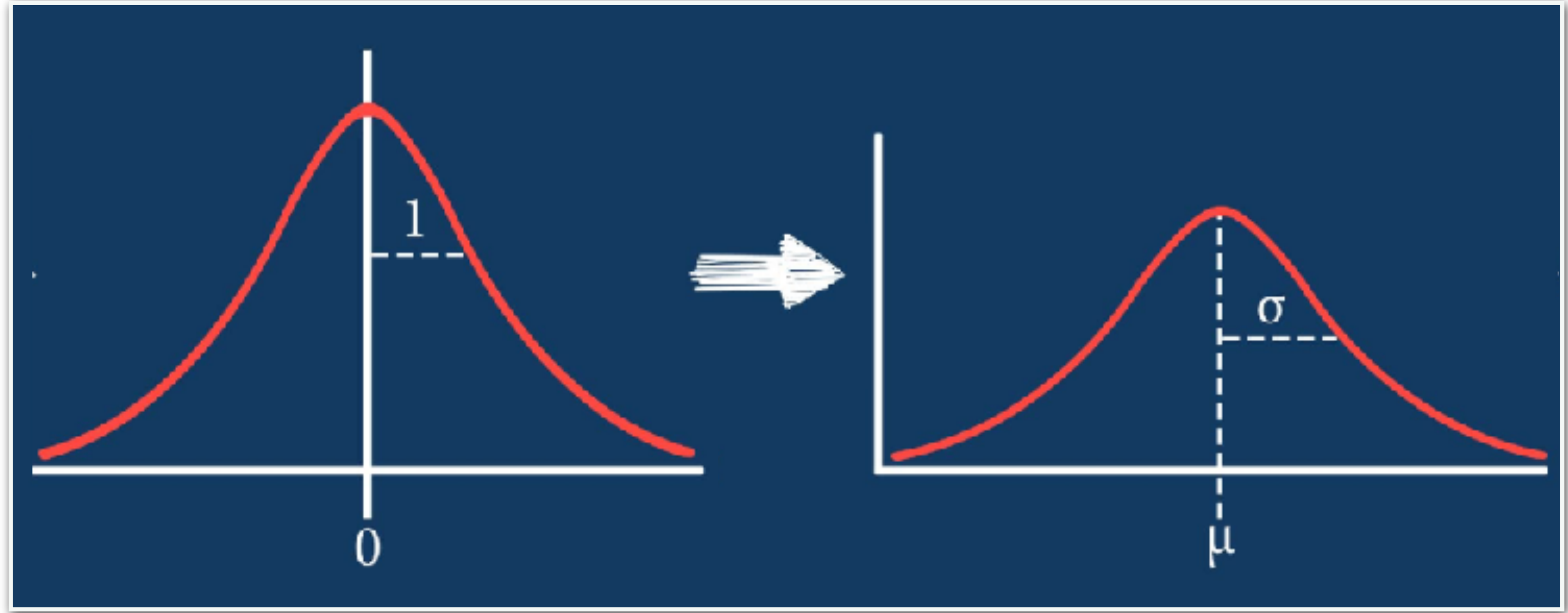
- **Goal**
  - Transform "physics" output target such that the network prediction remains in the network domain (typically $[-1,1]$)



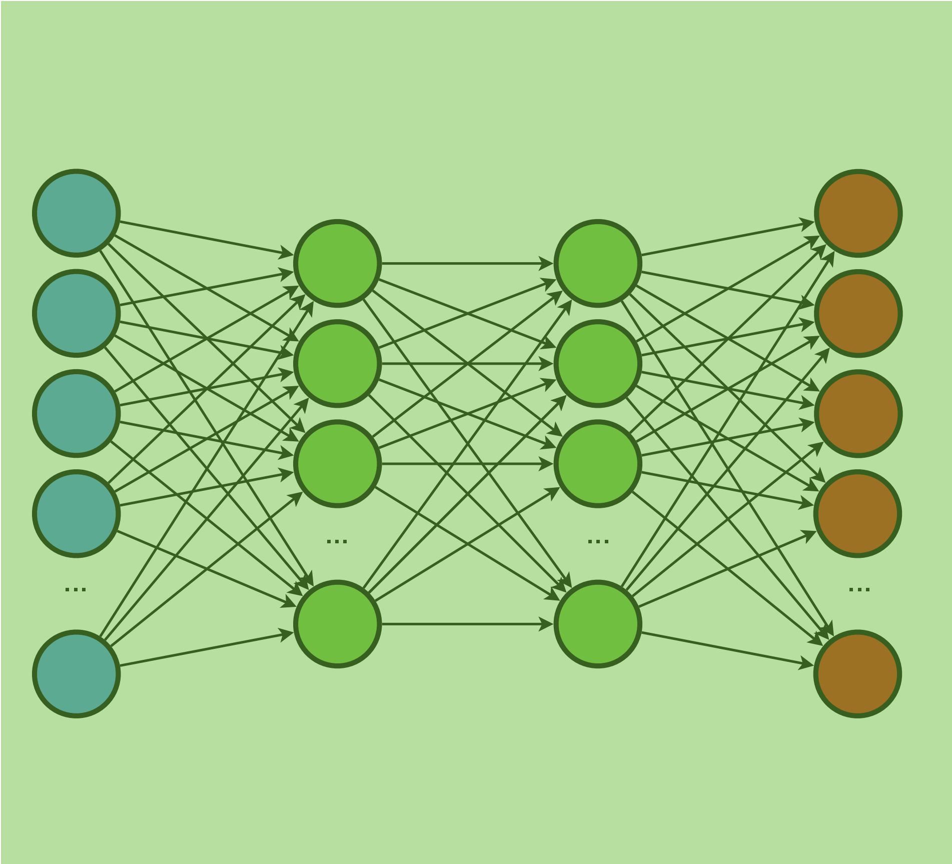
- **Benefits**
  - Large weights do not propagate back into the network
  - Similar to input feature scaling: loss landscape does not stretch

- **Simple "shift & scale" approach for prediction & ground truth**
  - For each target t, apply $t \to t' = \dfrac{t-\mu}{\sigma}$
    - μ: mean
    - σ: $\sqrt{}$variance
  - Need to **retransform** NN output to get actual physics output
    - $t' \to t = t' \cdot \sigma + \mu$
  - **Hint**: create final layer with constant, non-trainable scaling parameters per target to avoid having to remember those values



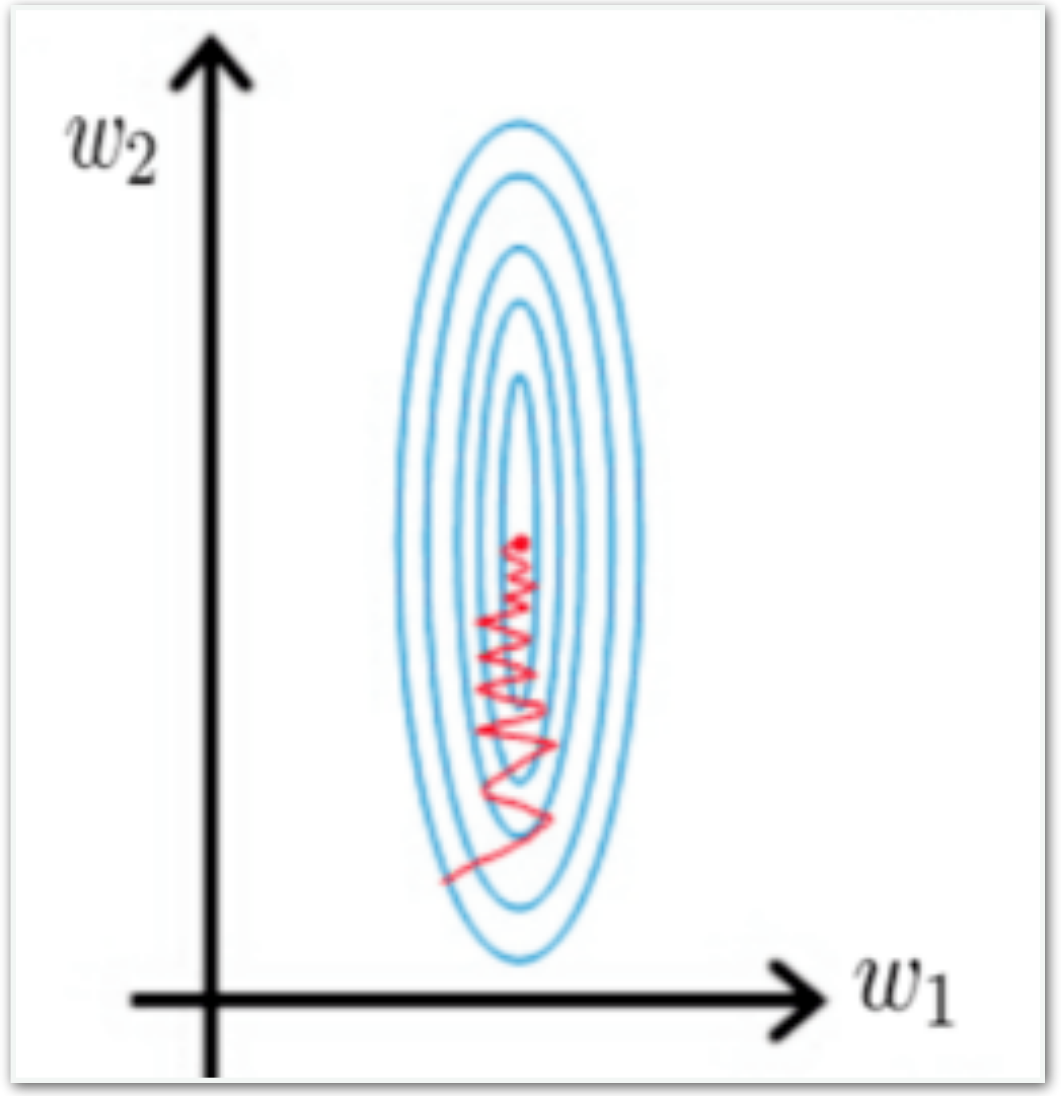


- **Caveats with large network weights**
  1. Volatile training steps in inhomogeneous losses
  2. Higher chance of vanishing gradients (dep. in activation)
  3. Network prone to so-called overtraining
     - ▻ Discussed in later today



- Input feature and regression target scaling mitigate large weights to some extent, but still needs consideration
  → Batch normalization and self-normalizing networks

Difficult loss landscape



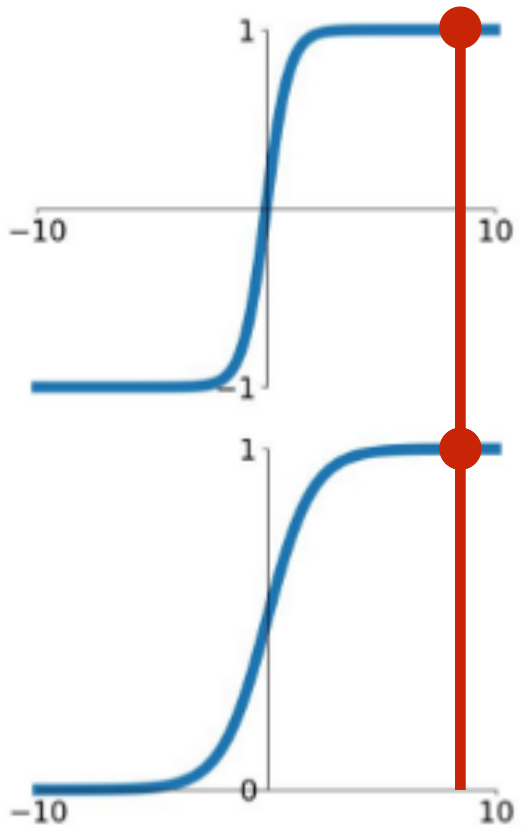
Vanishing gradients

tanh

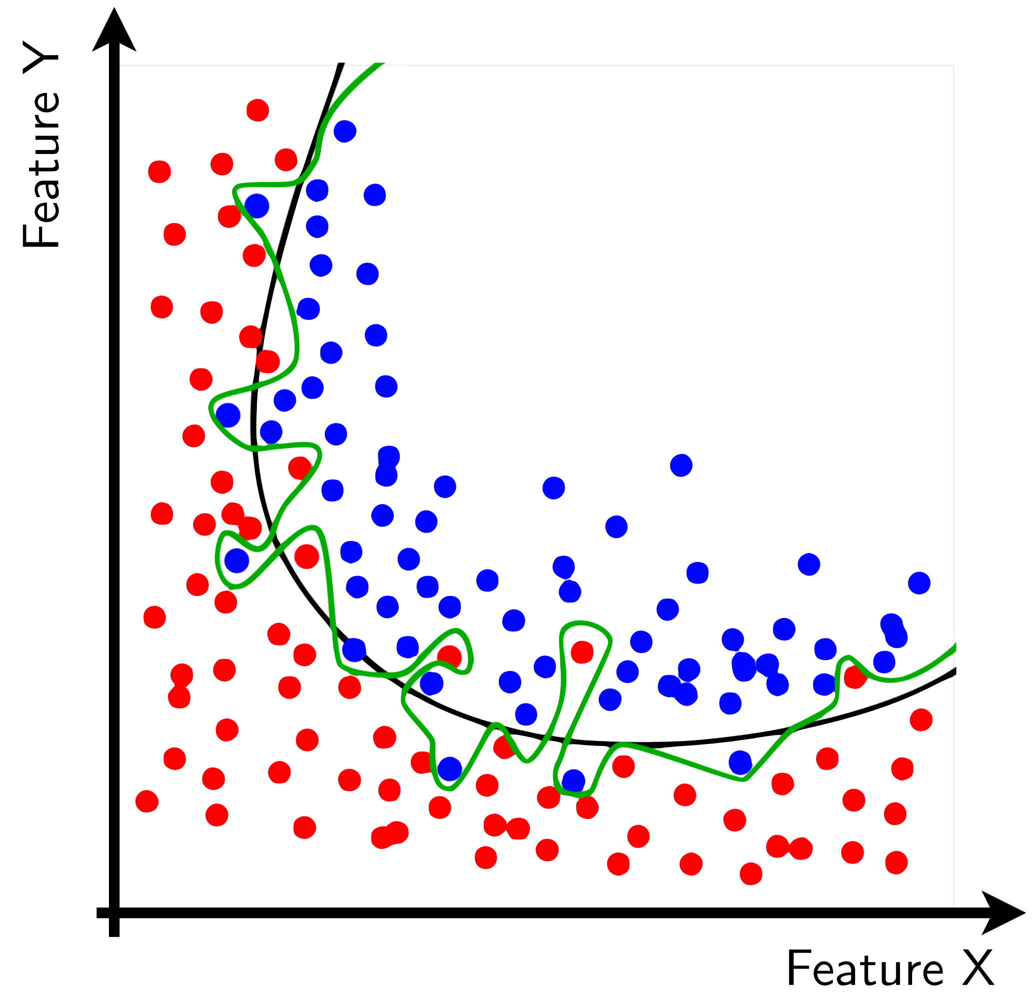
$\tanh(x)$

Sigmoid

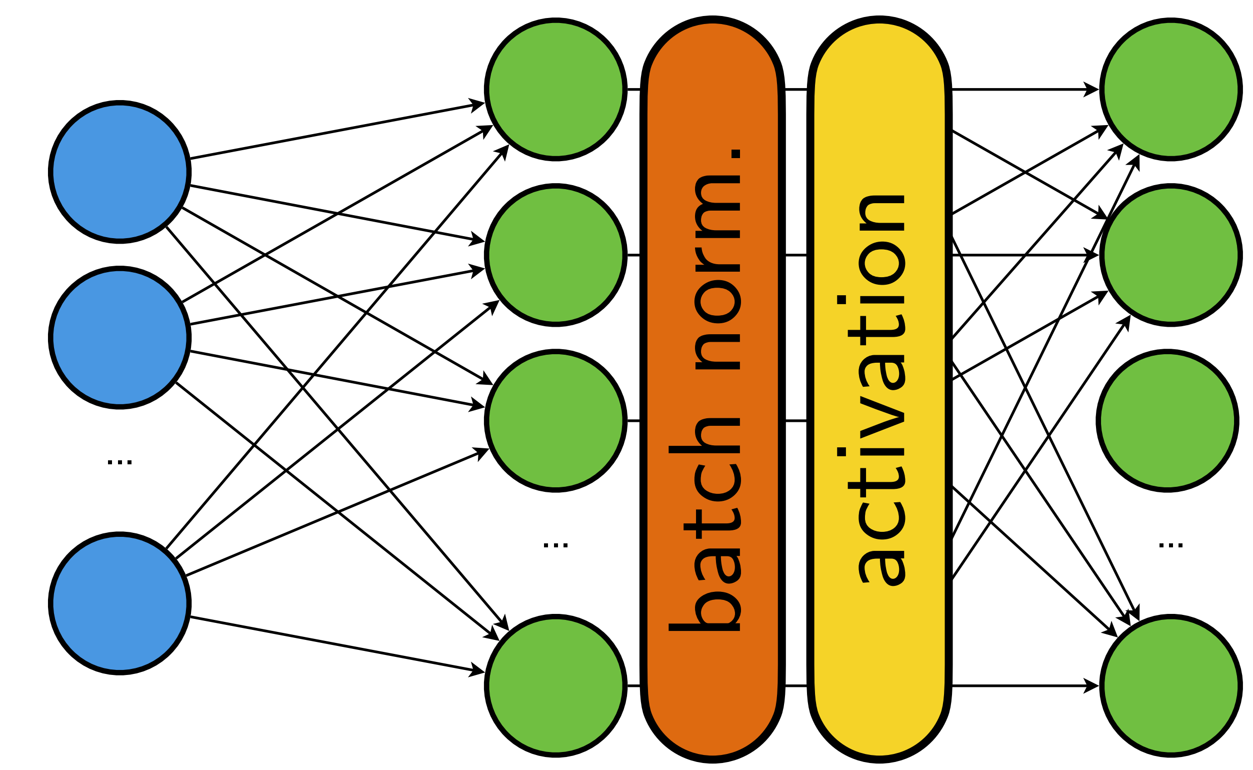
$\sigma(x) = \frac{1}{1+e^{-x}}$



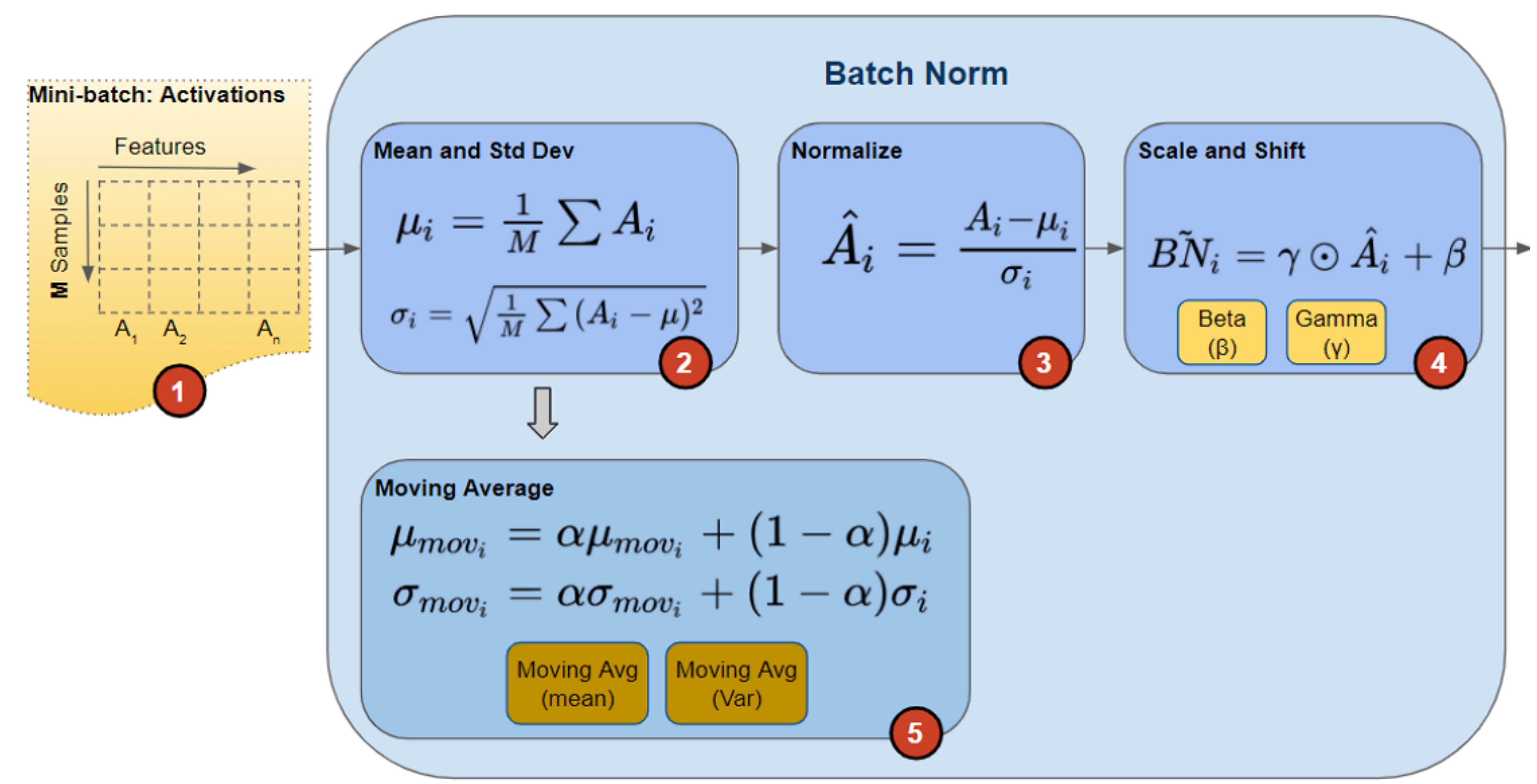
Overtraining



- **Idea**
  - Provided that the batch size is sufficiently large, the input feature scaling could be automatically evaluated and applied per batch!
  - **Moreover**, this normalization can be added between all layers (typically before activations)



- **Batch normalization**
  - During the first forward pass, compute mean μ_f and variance$^{0.5}$ σ_f for each feature f
  - Apply the scaling as before, $f' = \dfrac{f - \mu_f}{\sigma_f}$
  - Introduce **trainable** parameters γ_f and β_f that can adjust dispersion and shift again, if deemed desirable during training
  - μ_f and σ_f are *moving averages* ($\hat{\mu}_f$ and $\hat{\sigma}_f$)
    - For the next forward pass, use α-averaged quantities
      - $\hat{\mu}_f = \alpha\hat{\mu}_f + (1-\alpha)\hat{\mu}_f$
      - $\hat{\sigma}_f = \alpha\hat{\sigma}_f + (1-\alpha)\hat{\sigma}_f$



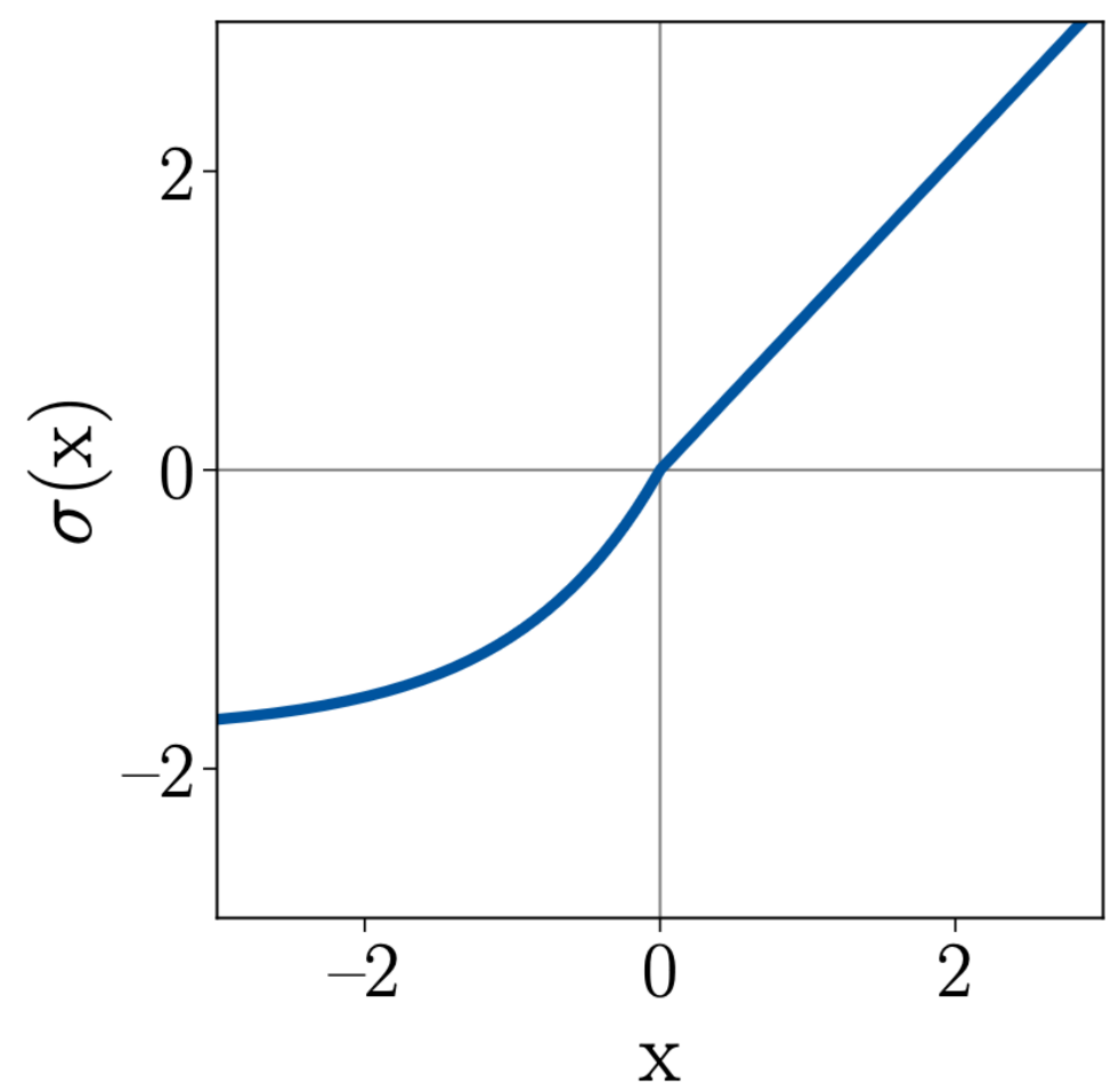
  - When just **evaluating** the network, use the last known averages and do not move them

- The mean and variance of layer activations can be intentionally constrained
  - Either with batch normalization, or
  - Scaled exponential linear units (**SELU**) activation

- Numerical stability reached in a way similar to beam focussing with F and D quadrupole magnets
  - (De)focussing in x(y) followed by (de)focussing in y(x), but when placed in perfect distance(*), overall effect is focussing in both planes

- **SELU**
  - Mean and variance per layer map to next layer such that they slightly alternate, but always remain in a defined region (proof)
  - Require fine tuned(*) scaling parameters λ and α

  - Alternative to batch-normalization (feel free to test)

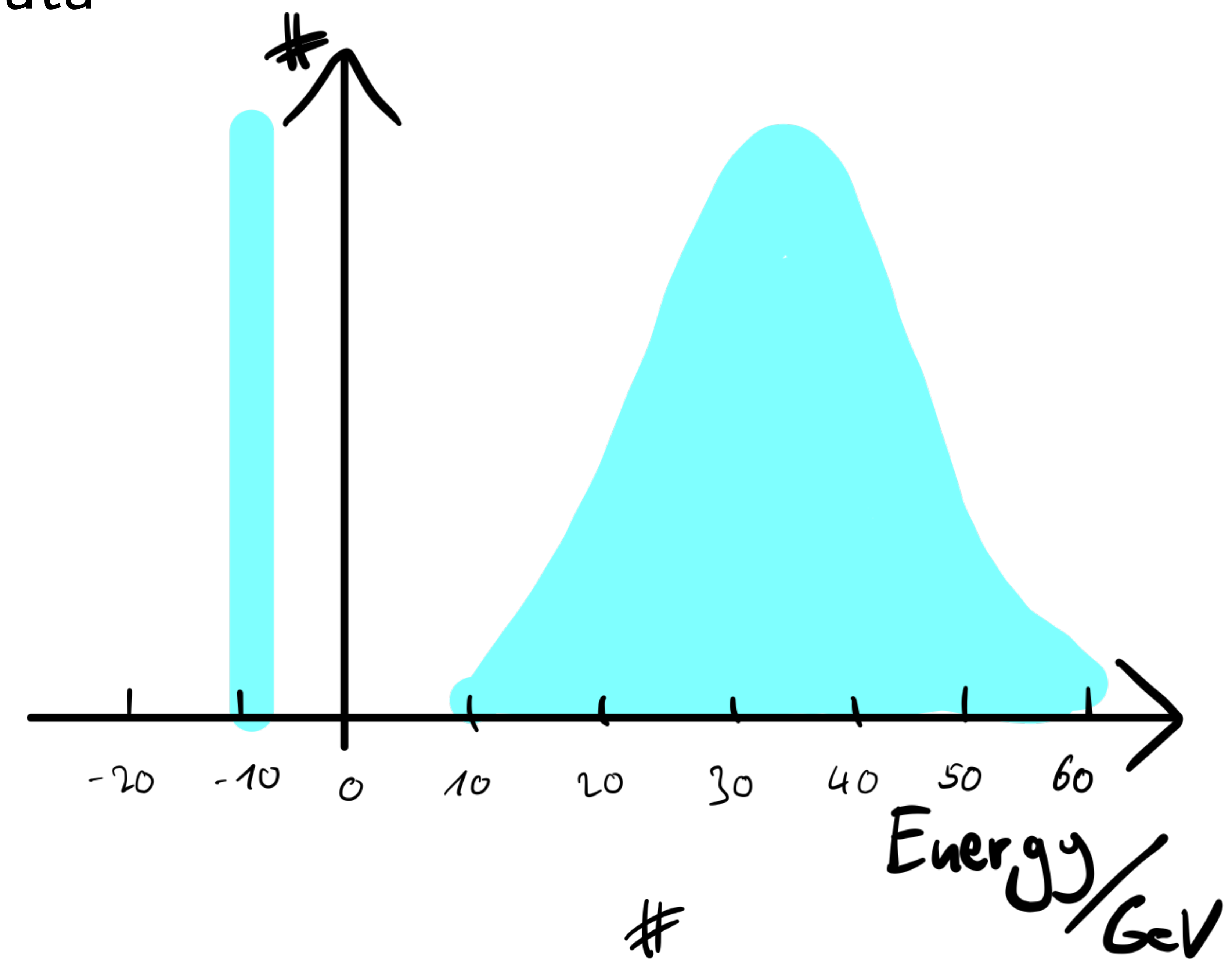


$\alpha \approx 1.6733$

$\lambda \approx 1.0507$

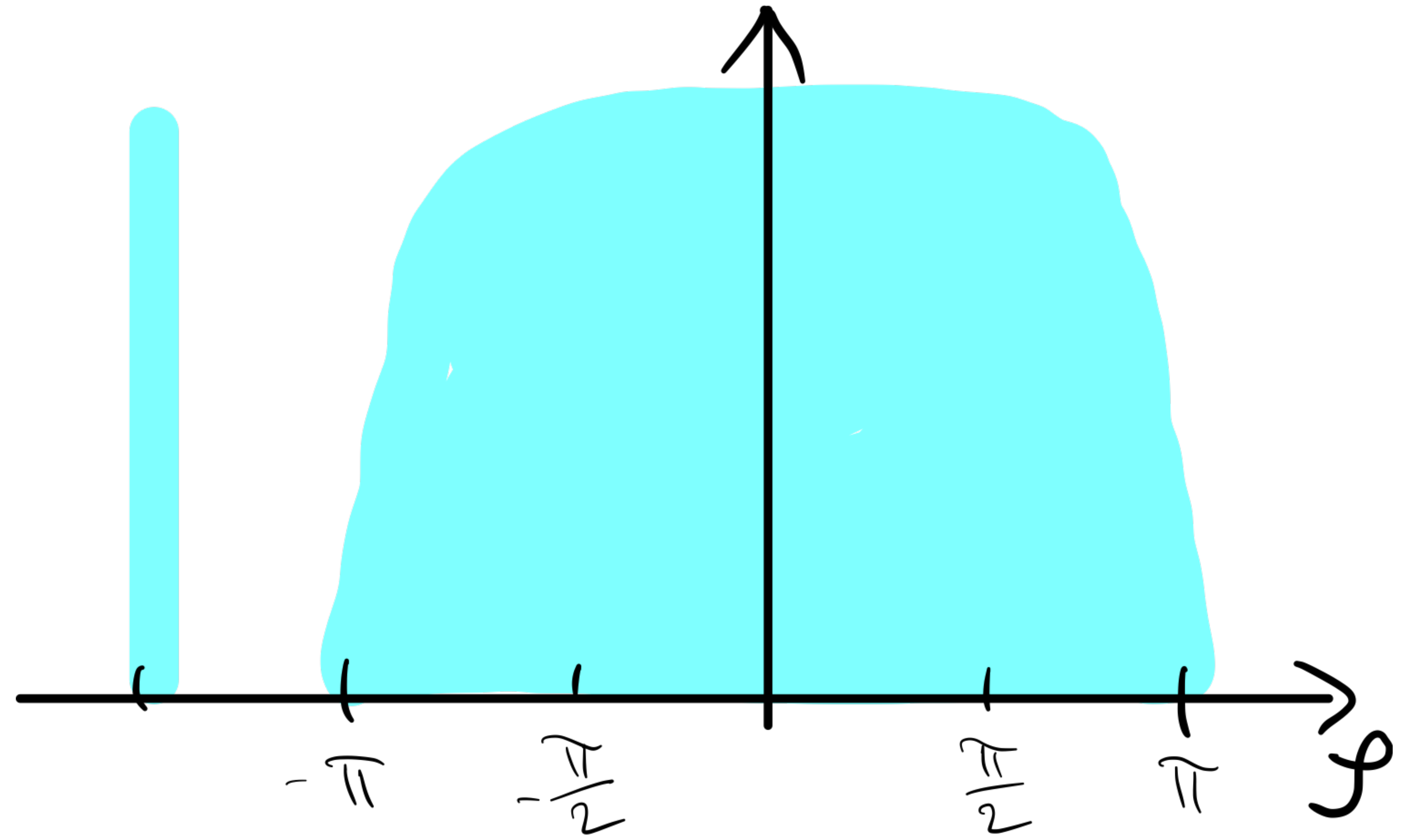
$$\mathrm{SELU}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha\left(e^{x}-1\right) & \text{if } x \leq 0 \end{cases}$$

- In some scenarios, input features might be missing for *some* samples in your data



- **Example**
  - Feeding four-momenta of leading 4 jets into network
  - But, in some samples (events) there are only three jets

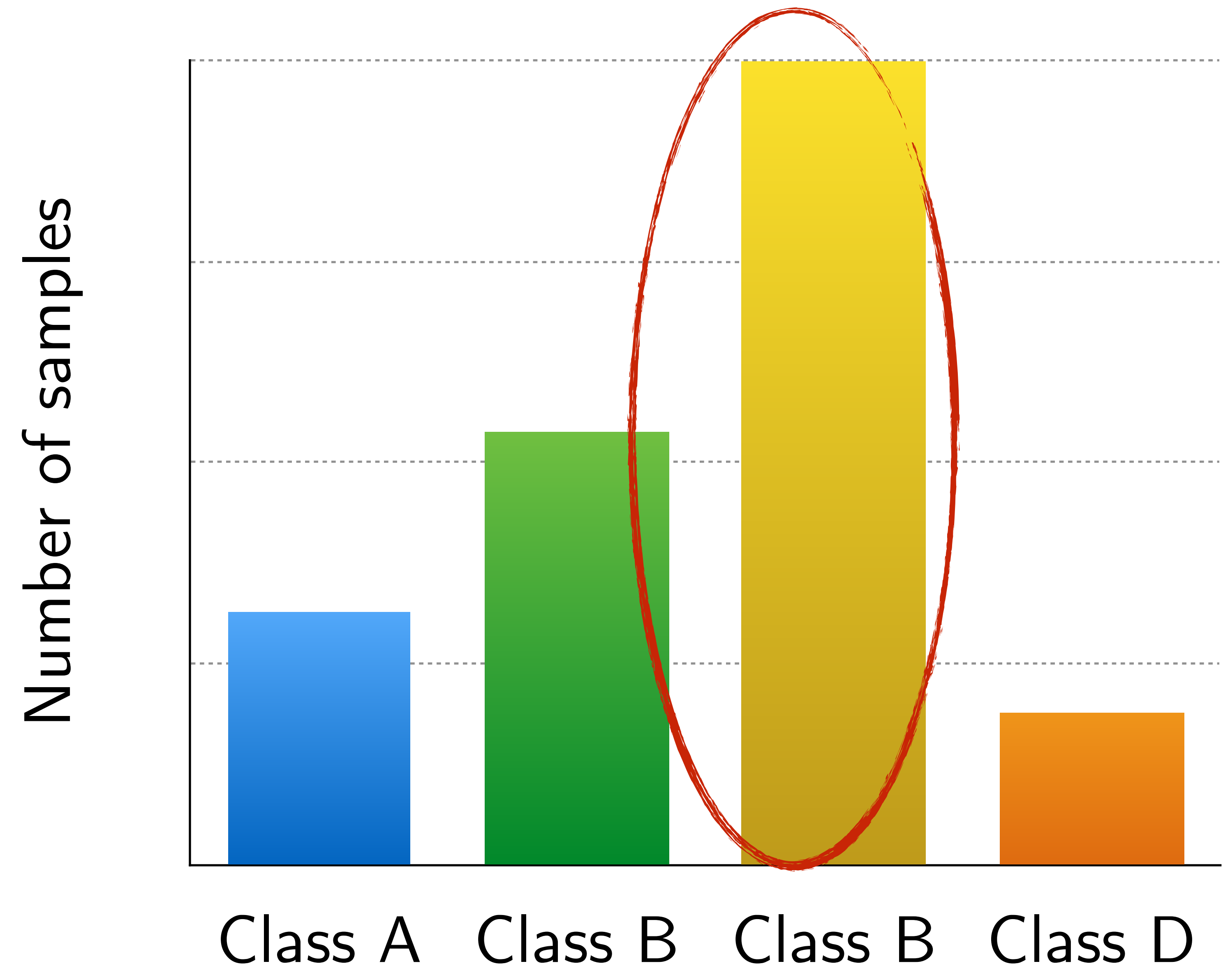
- **Common approaches**
  - Train separate networks with only existing features for these cases
    - ▻ Only beneficial if many inputs are affected
    - ▻ Otherwise discouraged
      - – Requires multiple trainings
      - – Each with fewer samples → less predictive power



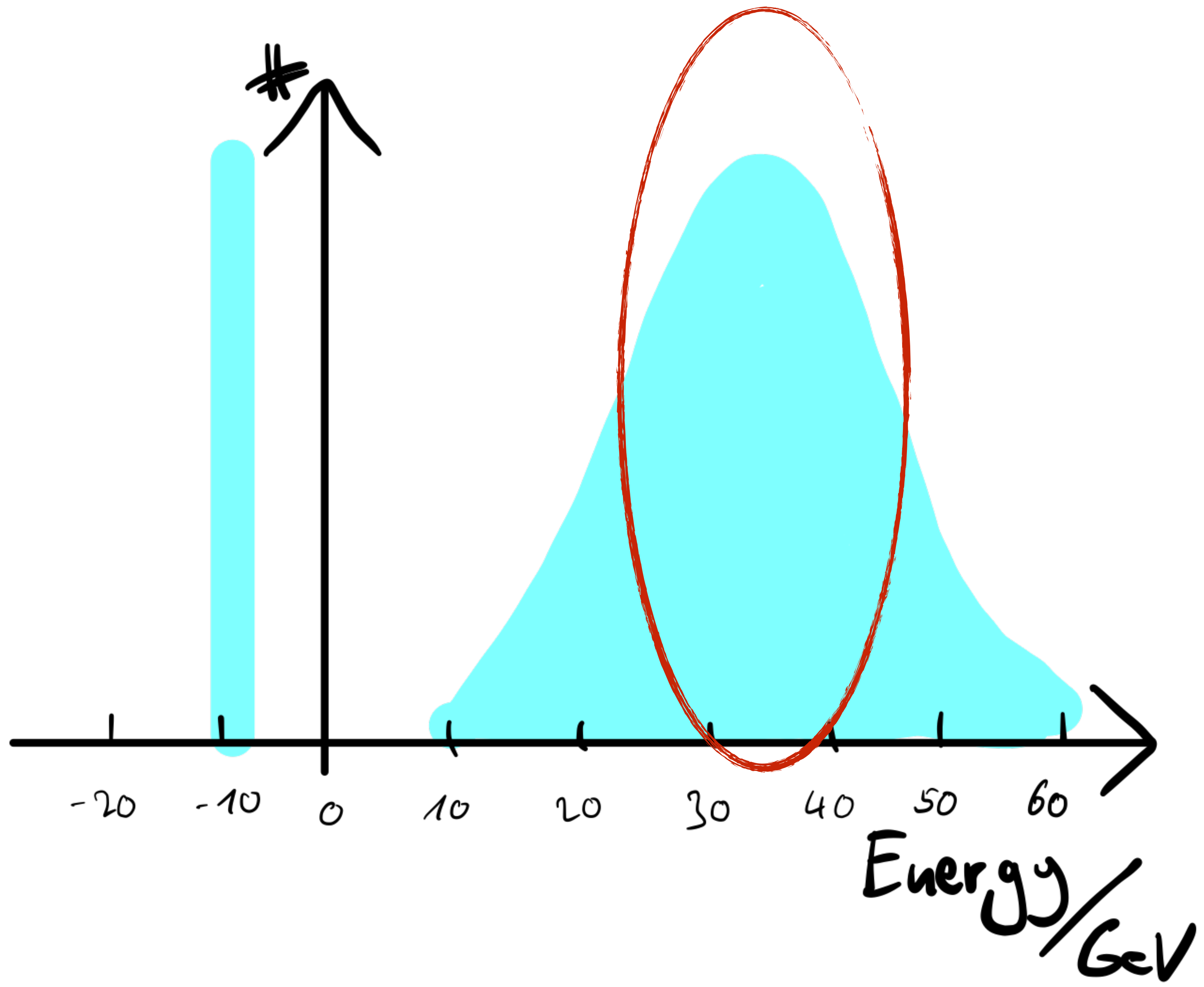
  - **Better**: Encode these cases with *null* values
    - ▻ Missing values constitute *additional* information on their own!
    - ▻ Actual *null* value definition depends on feature distribution
      - – Proximity to bulk of distribution would imply numerical relation!
      - → At least ~ 3σ from center μ of distribution for networks to *see* gap

❗ Attention: consider skipping these samples when deriving parameters for input feature scaling

Class frequency for classification

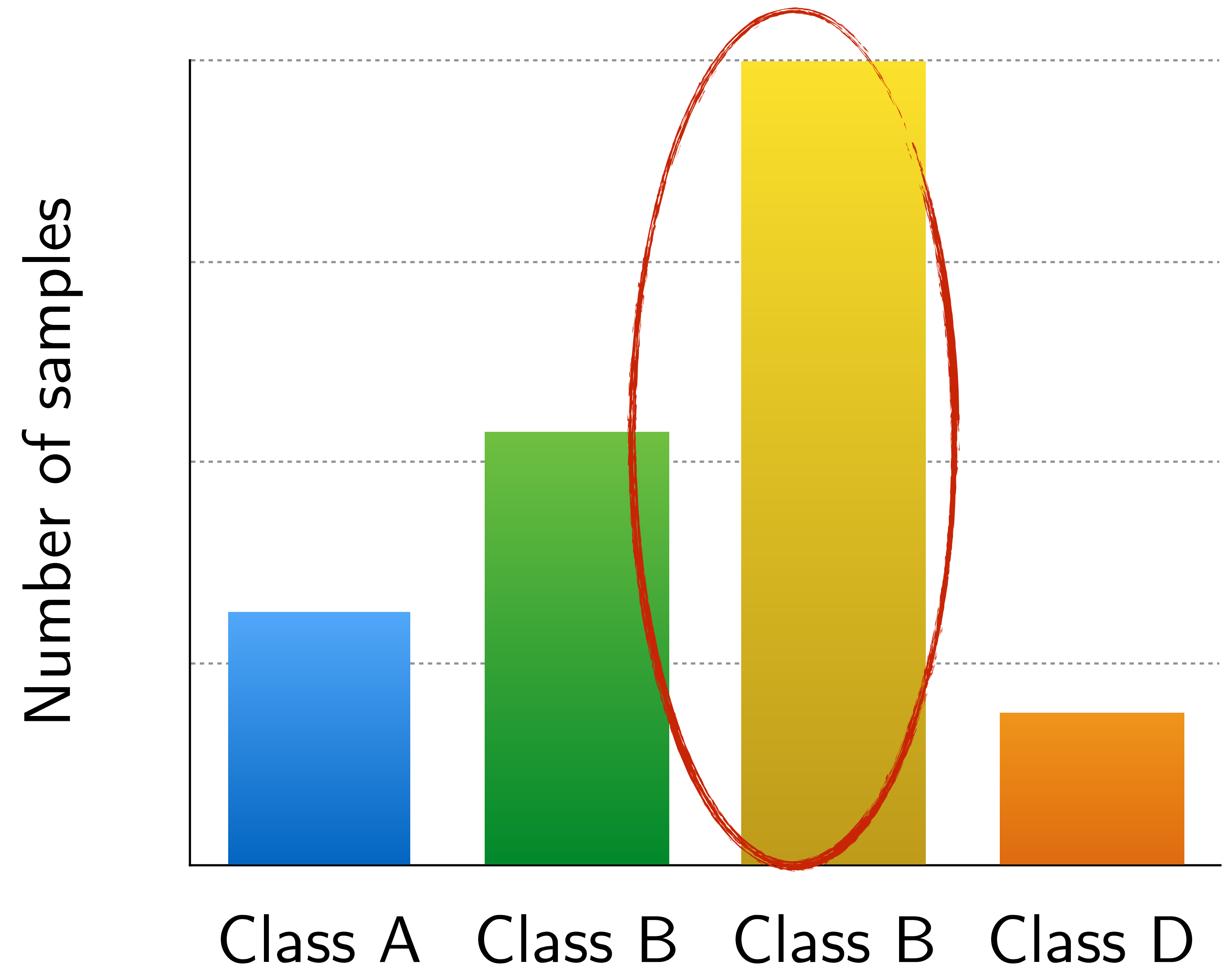


Distribution of regression target

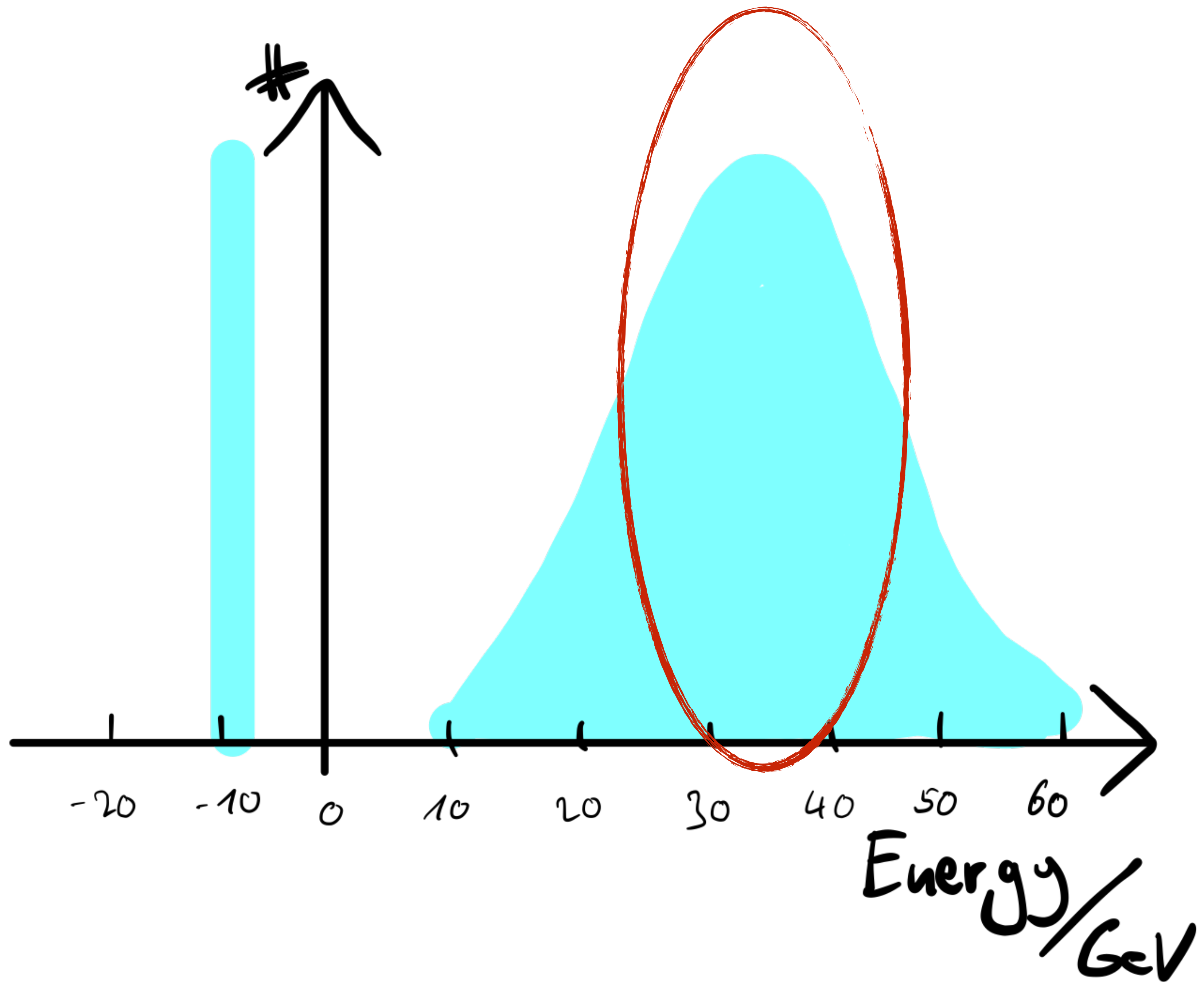


- **What will happen during training?**

Class frequency for classification



Distribution of regression target



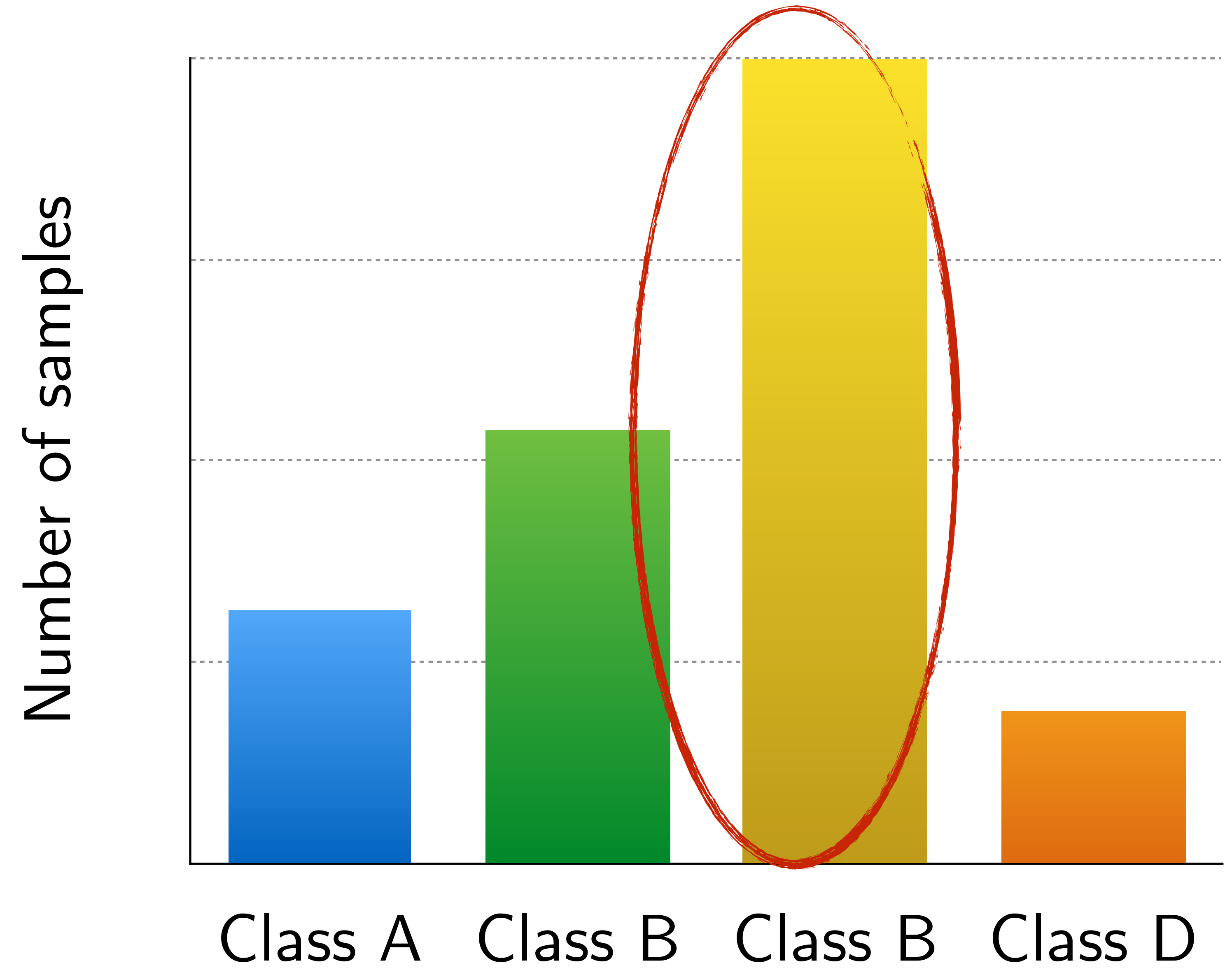
- **What will happen during training?**
  - Networks will *focus more on over-represented* classes (regions) than on the unpopulated ones

  → Not *necessarily* what you want

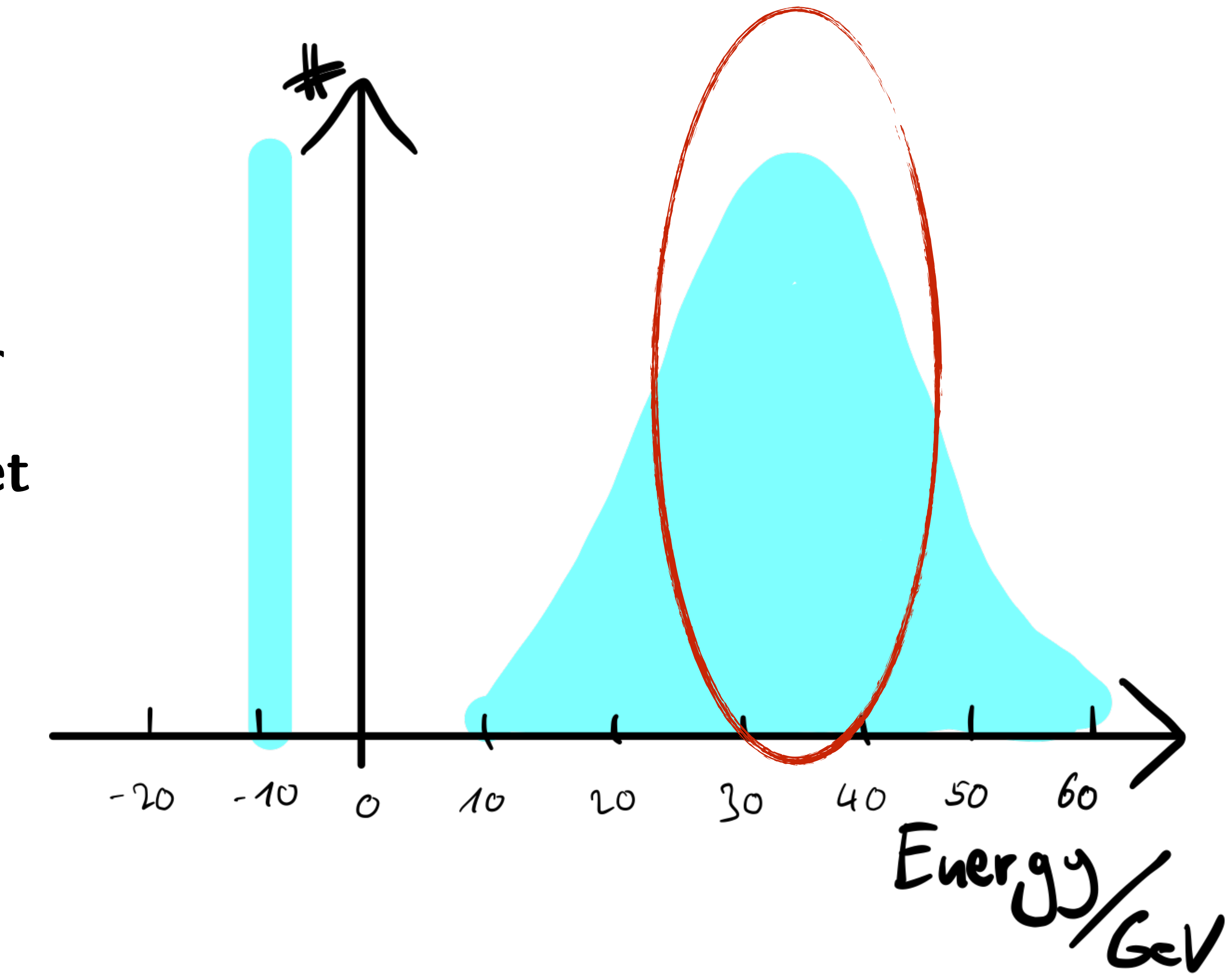
- **Possible approaches**
  1. Down-sampling: remove samples in over-represented classes (generally discouraged, esp. when statistics is an issue)
  2. Collocation: let training batch consist of equal amount of classes (only complicates the definition of "epoch")
  3. Sample weights: loss functions support per-sample weights to control sample / class *importance*

Classification: weight samples of class c by $\frac{< N >_{classes}}{N_c}$ **Regression**: bin distribution and weight as for classification

Class frequency for classification



Distribution of regression target



- **What will happen during training?**
  - Networks will *focus more on over-represented* classes (regions) than on the unpopulated ones

  → Not *necessarily* what you want **... or do you?**

- **Possible approaches**
  1. Down-sampling: remove samples in over-represented classes (generally discouraged, esp. when statistics is an issue)
  2. Collocation: let training batch consist of equal amount of classes (only complicates the definition of "epoch")
  3. Sample weights: loss functions support per-sample weights to control sample / class *importance*

Classification: weight samples of class c by $\frac{<N>_{classes}}{N_c}$ **Regression**: bin distribution and weight as for classification

3. Techniques 1/2 & hands-on

- Keras **sequential model** known from Dennis' lectures

```
import tensorflow as tf

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(128, input_dim=32))
model.add(tf.keras.layers.Dense(128))
model.add(tf.keras.layers.Dense(128))
model.add(tf.keras.layers.Dense(128))
model.add(tf.keras.layers.Softmax(2))
...
```

documentation

- More freedom and options in **functional** API

```
import tensorflow as tf

x = tf.keras.Input(shape=(32,))
a1 = tf.keras.layers.Dense(128)(x)
a2 = tf.keras.layers.Dense(128)(a1)
a3 = tf.keras.layers.Dense(128)(a2)
a4 = tf.keras.layers.Dense(128)(a3)
y = tf.keras.layers.Dense(2, activation="softmax")(a4)
model = tf.keras.Model(inputs=x, outputs=y)
```

documentation

- Custom layers need to implement 5 methods

documentation

```python
import tensorflow as tf

class FeatureScaling(tf.keras.layers.Layer):

    def __init__(self, means, stddevs):
        """
        Constructor. Stores arguments as instance members.
        """
        super(FeatureScaling, self).__init__(trainable=False)

        self.means = means
        self.stddevs = stddevs

    def get_config(self):
        """
        Method that is required for model cloning and saving. It
        should return a mapping of instance member names to the
        actual members.
        """
        return {"means": self.means, "stddevs": self.stddevs}

    def compute_output_shape(self, input_shape):
        """
        Method that, given an input shape, defines the shape of
        the output tensor. This way, the entire model can be
        built without actually calling it.
        """
        return (input_shape[0], input_shape[1] * input_shape[2])

    def build(self, input_shape):
        """
        Any variables defined by this layer should be
        created inside this method. This helps Keras to
        defer variable registration to the point where it
        is needed the first time, and in particular not at
        definition time.
        """
        # nothing to do here as our feature scaling
        # has no trainable parameters

    def call(self, c_vectors):
        """
        Payload of the layer that takes inputs and computes
        the requested output whose shape should match what
        is defined in compute_output_shape.
        """
        ... implementation missing :)

        return features
```

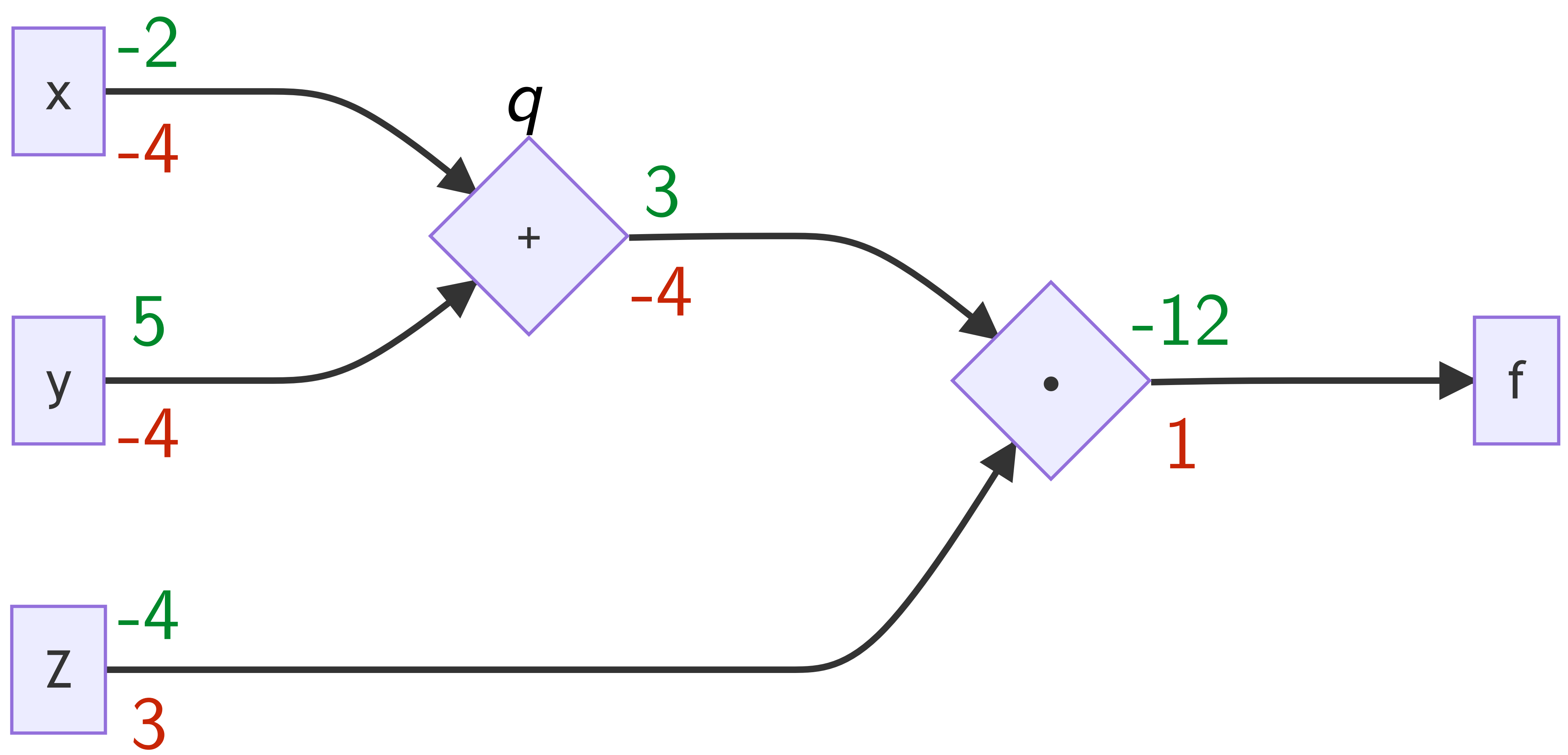
- Quick introduction to gradients

```python
import tensorflow as tf

@tf.function
def example():
  a = tf.constant(2.)
  b = 3 * a
  return tf.gradients(a + b, [a, b], stop_gradients=[a, b])

example()
# [4.0, 1.0]
```

documentation

- **Your tasks**

1. Gradients (colab notebook, 15")
   a) Repeat the gradient computation to the right
   b) Play around with more complex computational graphs and verify your results (e.g. sin, cos, exp, ^{2}, ...)
2. Keras' functional API (colab notebook, 25")
   a) Build your own model
   b) Write a custom layer that performs feature scaling
   c) Extend your model to create a multi-purpose network



Today
14:30 - 16:00

20″ 1. **Variants of and improvements in fully-connected networks (FCNs)** ✔
- Gradient calculation (recap), vanishing gradients, ResNet, ensemble learning, multi-purpose networks

30″ 2. **Numerical insights & considerations** ✔
- Domains, feature & output scaling, batch normalization, SELU, categorical embedding, class imbalance

40″ 3. **Techniques 1/2 & hands-on** ✔
- Keras functional API, custom Keras layer, computing gradients

Today
16:30 - 18:00

25″ 4. **Regularization & overtraining suppression**
- Overtraining & generalization, capacity & capability, regularization, dataset splitting

25″ 5. **Model optimization**
- Optimizer choices, class-importance, hyper-parameters, search strategies

40″ 6. **Techniques 2/2 & hands-on**
- Compute architecture, TensorFlow eager and graph, custom training loop, tensorboard

Tomorrow
09:00 - 10:30

10″ 7. **Exercise introduction: Identifying Jets in Particle Collider Experiments**
- Problem statement, input data & features, objective(s)

70″ 8. **Hands-on!**
- Classification task, implementing newly learned techniques, extension to multi-purpose network

10″ 9. **Exercise summary and tips**
- Example wrap-up, additional practical tips

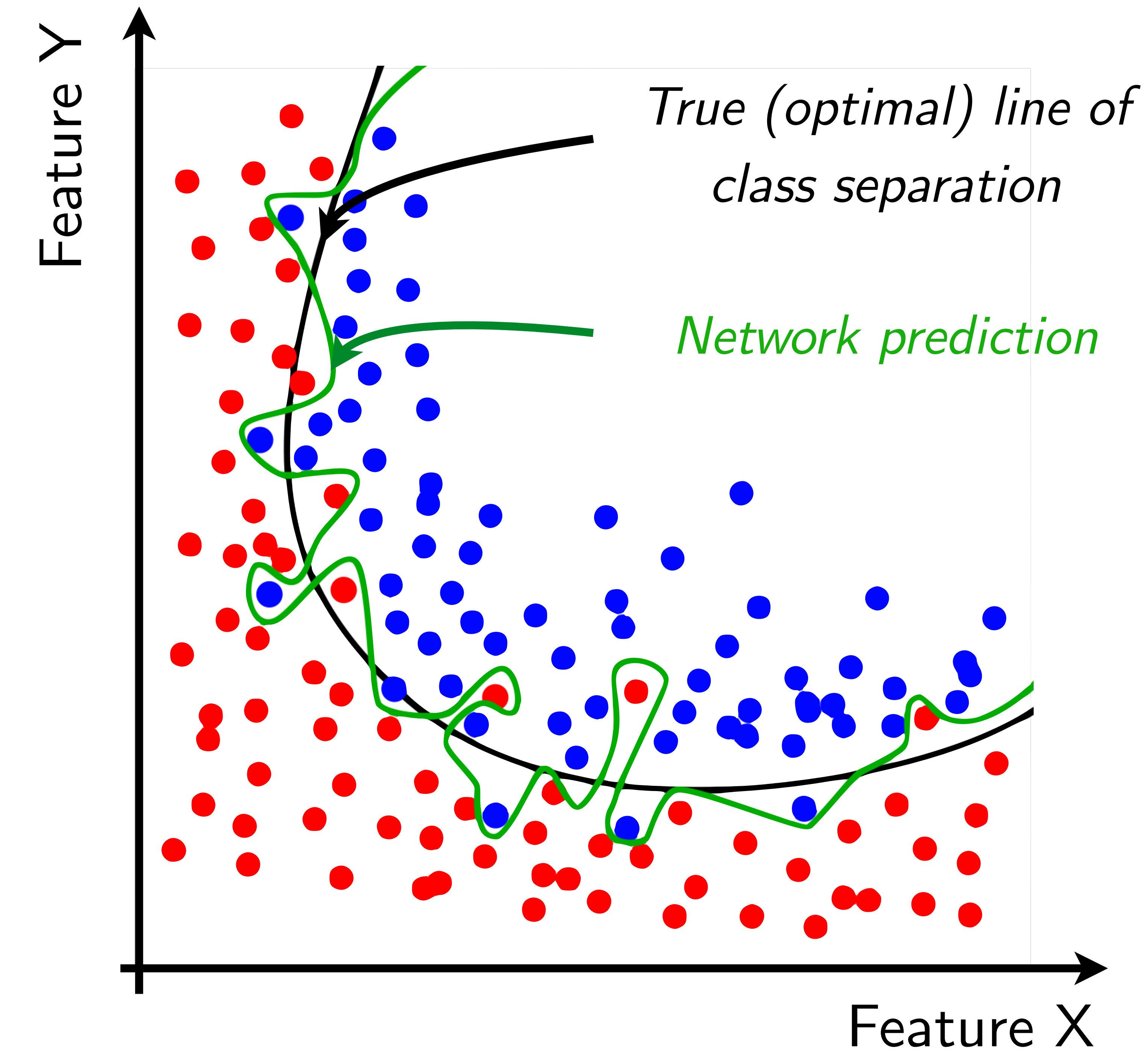
4. Regularization & overtraining suppression

TensorFlow playground

TensorFlow playground

Overtraining

- Network learns training data and **fails to generalize** to underlying truth (*pdf*)

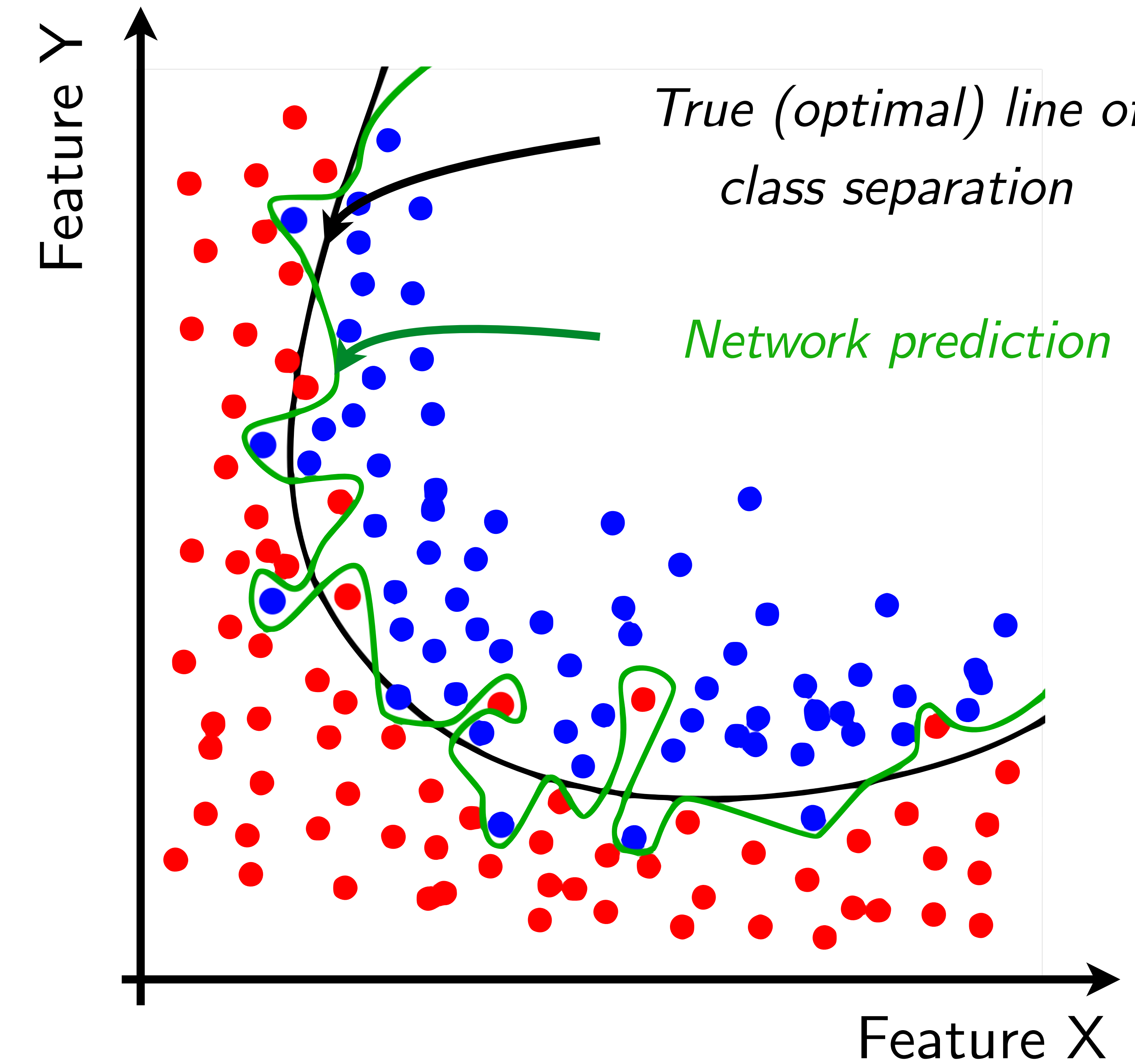


- **Most evident reasons**
  1. Insufficient training statistics
     - ▻ Training samples fail to represent truth with sufficient accuracy
       (*longer*: there will **always** be noise, but with enough statistics, it becomes less likely that random outliers shift the appearance of the full sample distribution)
     - ▻ Allows networks to learn particular training samples
       (*longer*: besides global trends, networks have enough **capacity** to also focus on local density fluctuations)

- Network learns training data and **fails to generalize** to underlying truth (*pdf*)

- **Most evident reasons**

  1. Insufficient training statistics
     - ▻ Training samples fail to represent truth with sufficient accuracy
       (*longer*: there will **always** be noise, but with enough statistics, it becomes less likely that random outliers shift the appearance of the full sample distribution)
     - ▻ Allows networks to learn particular training samples
       (*longer*: besides global trends, networks have enough **capacity** to also focus on local density fluctuations)



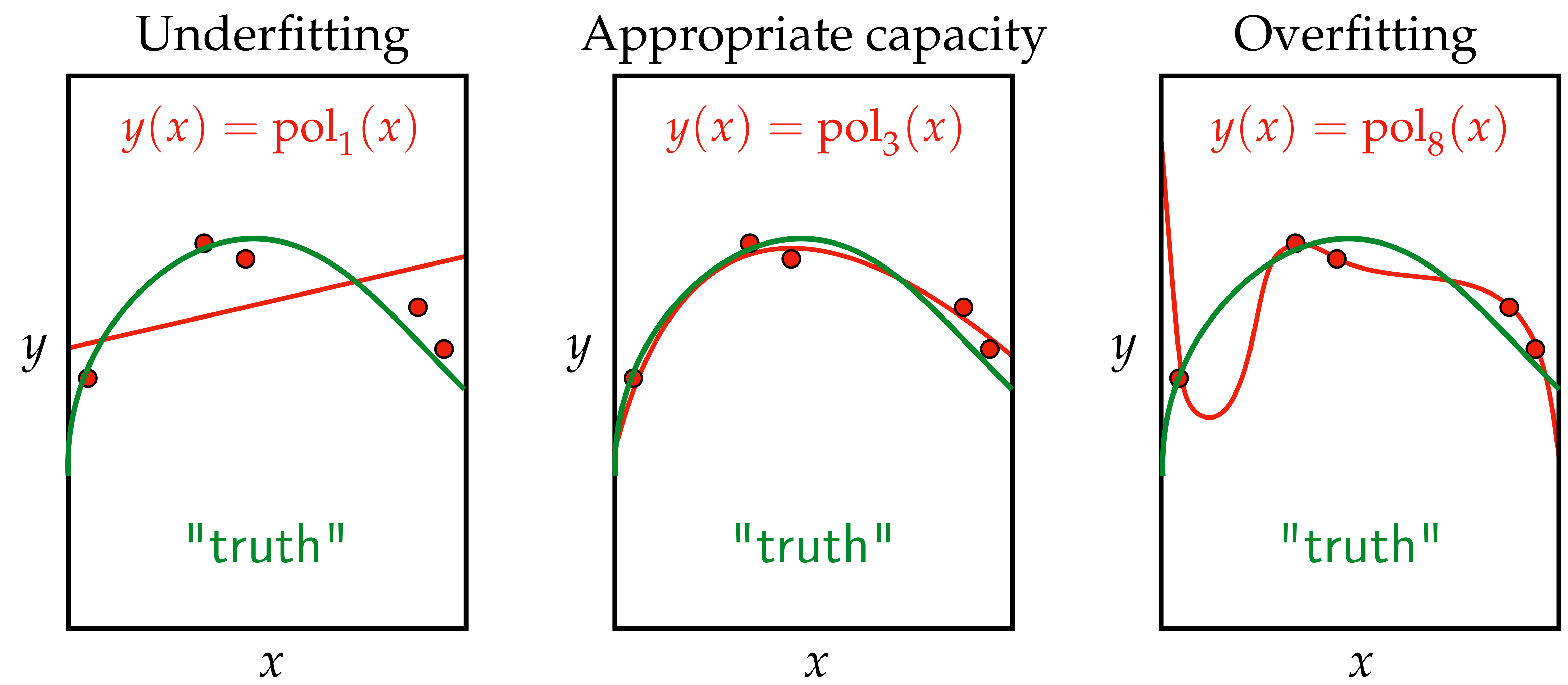
  2. Over-powered network (~*inverse* of 1.)
     - ▻ High capacity allows network to model (*remember*) higher amount of density changes
       (*longer*: model complex enough for prediction (green line) to become extremely volatile / "zig-zagy")
     - ▻ Network potentially **capable** to focus on non-representative regions
       (*longer*: even a few outliers can cause the network to move the decision boundary, trying to include them)

Model **capacity** Training **statistics**

Model capacity & capability

- Consider a simple ground truth (pdf) and a polynomial fit

too low capacity →

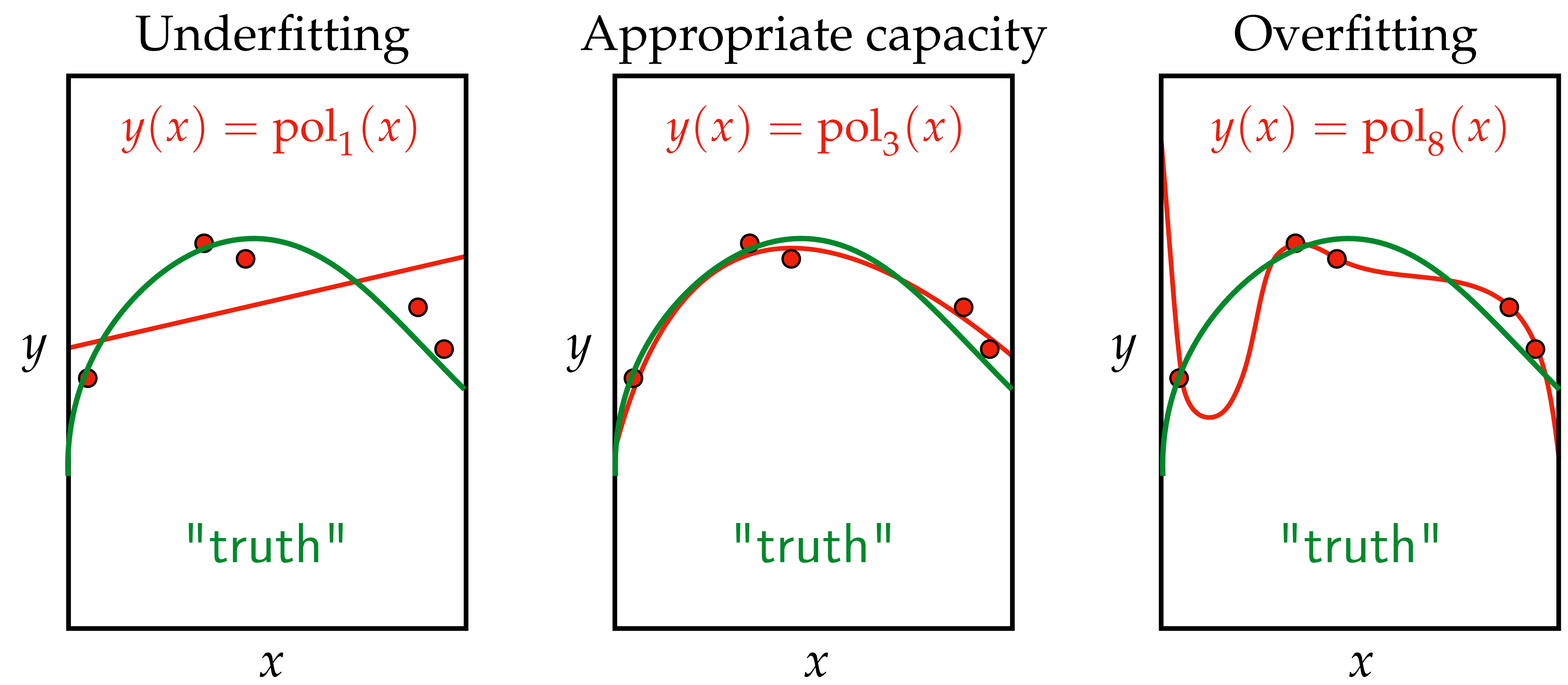


← *high* capacity

- *Which model would you pick?*

- Consider a simple ground truth (pdf) and a polynomial fit

too low capacity →

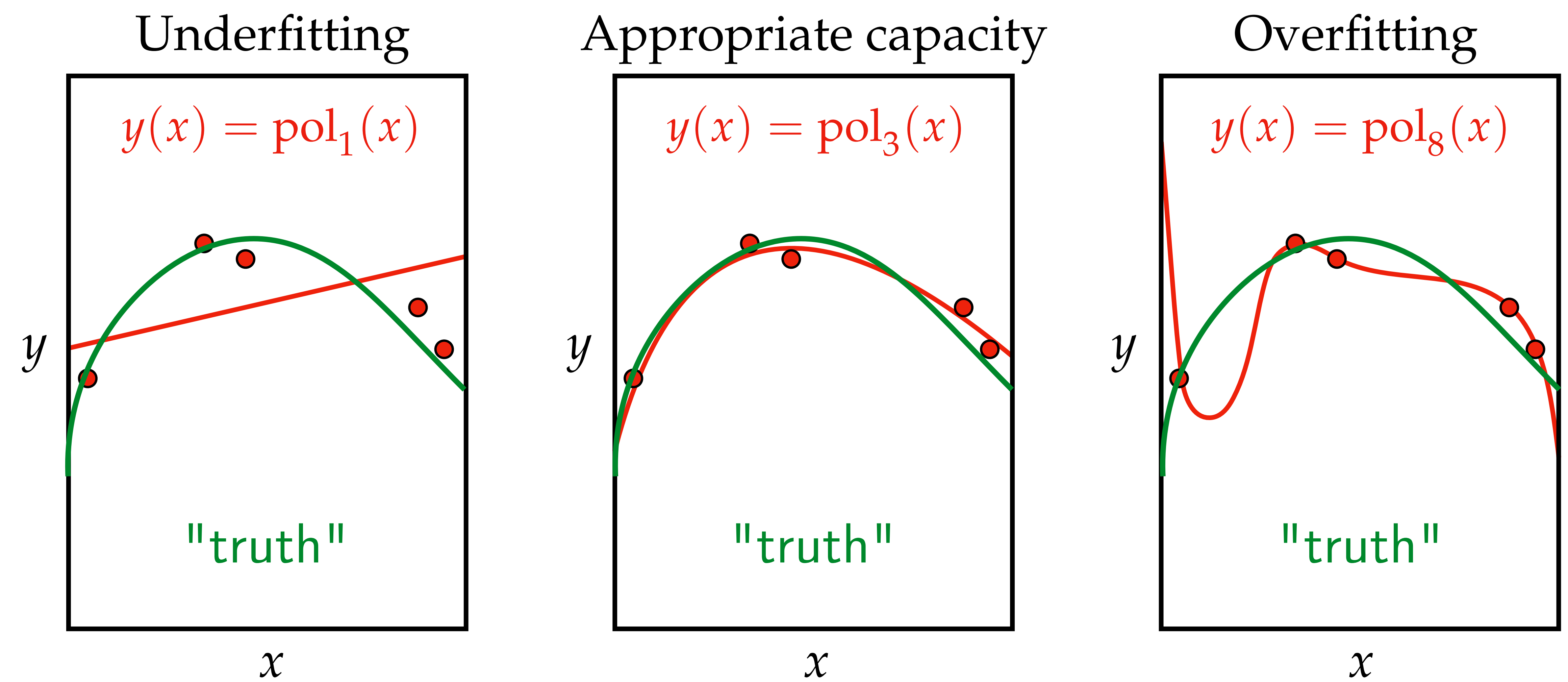


← *high* capacity

- *Which model would you pick?* 

- Consider a simple ground truth (pdf) and a polynomial fit

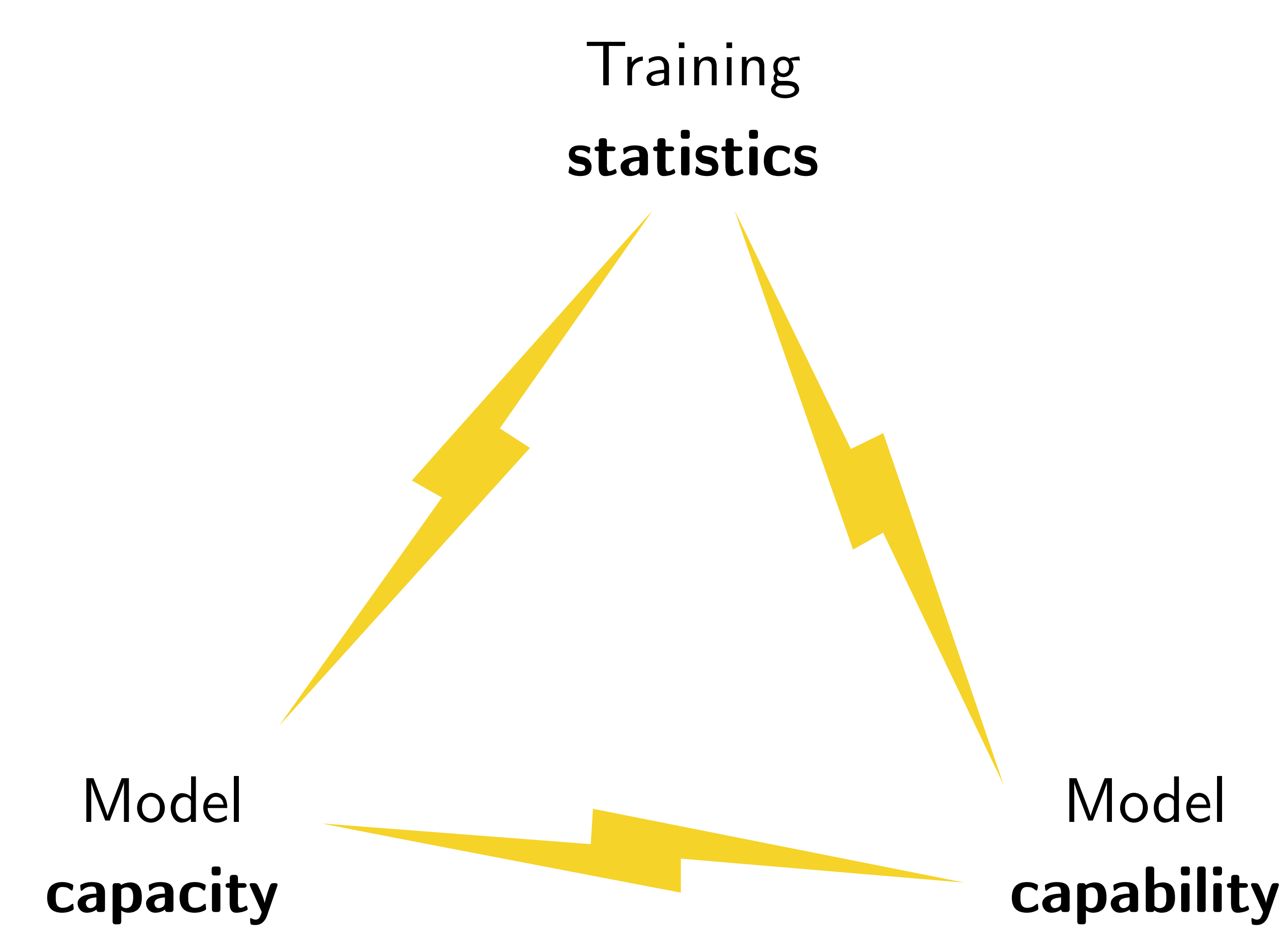
too low capacity →



← *high* capacity

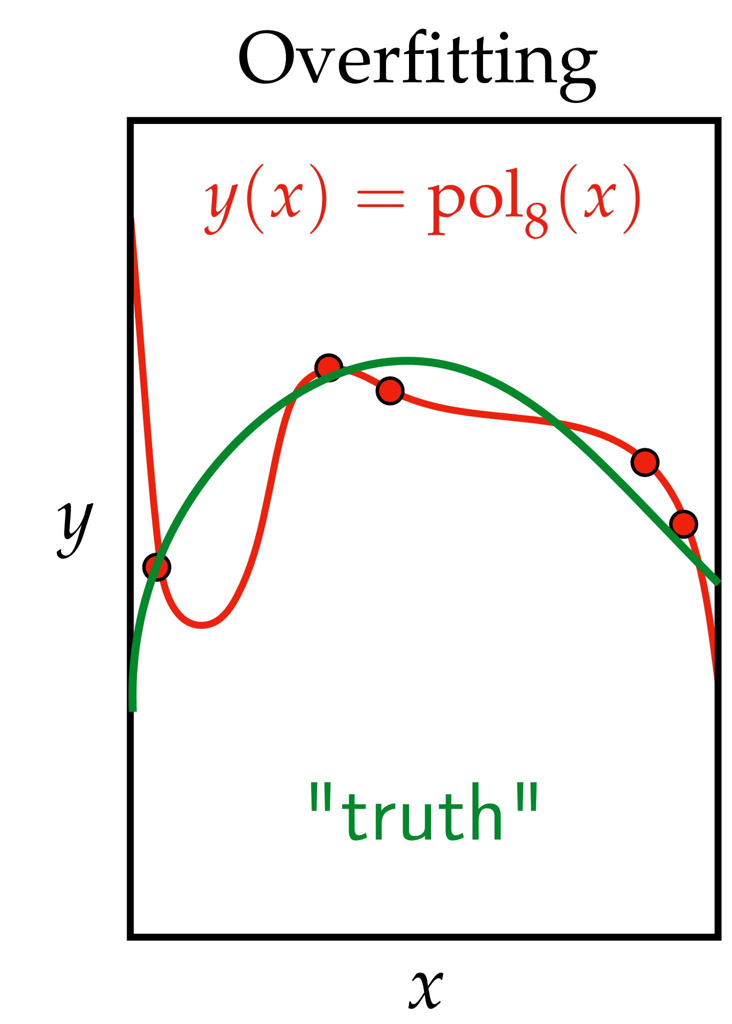
- *Which model would you pick?* 

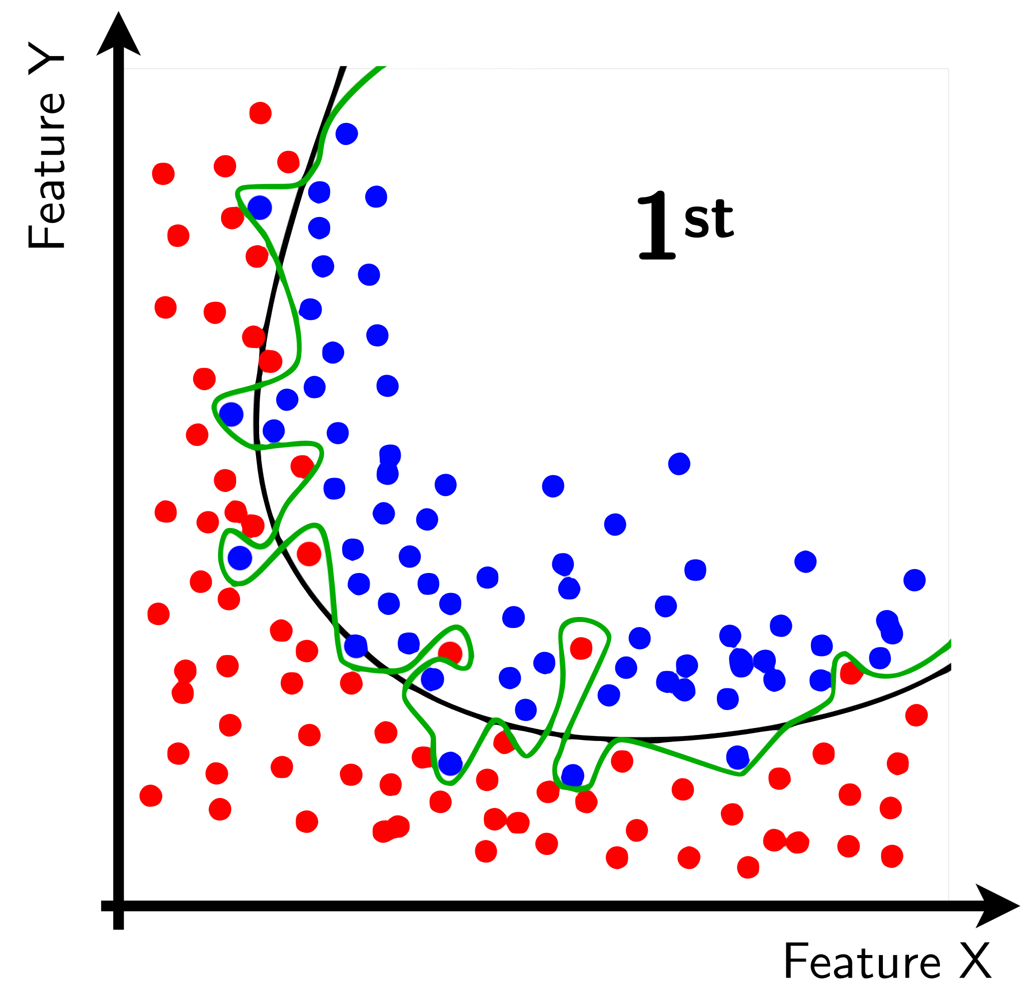
- **Caveats**
  - "Appropriate" capacity hardly known
  - Less **capacity** leads to less **capability**!
    - ▻ Complex-to-reconstruct hidden features require complex networks
      ("complex network" = large, well-designed, or both)
    - ▻ See particle mass regression example!



→ Always go with (reasonably) higher capacity. Don't sacrifice capability for overfitting suppression.

- **1st scenario: overtraining**
  - Small number of weights become large
    - $y(x) = 2.1 + 3.9x^4 - 4.6x^6 + 4.4x^7 + \ldots$ (example)
    - Should be avoided

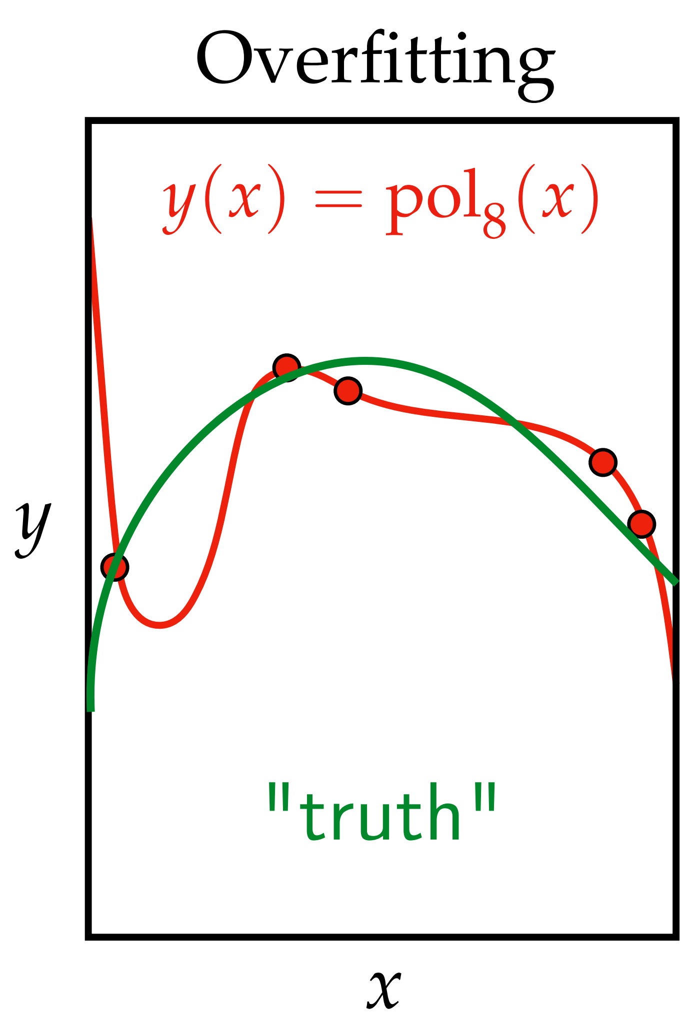


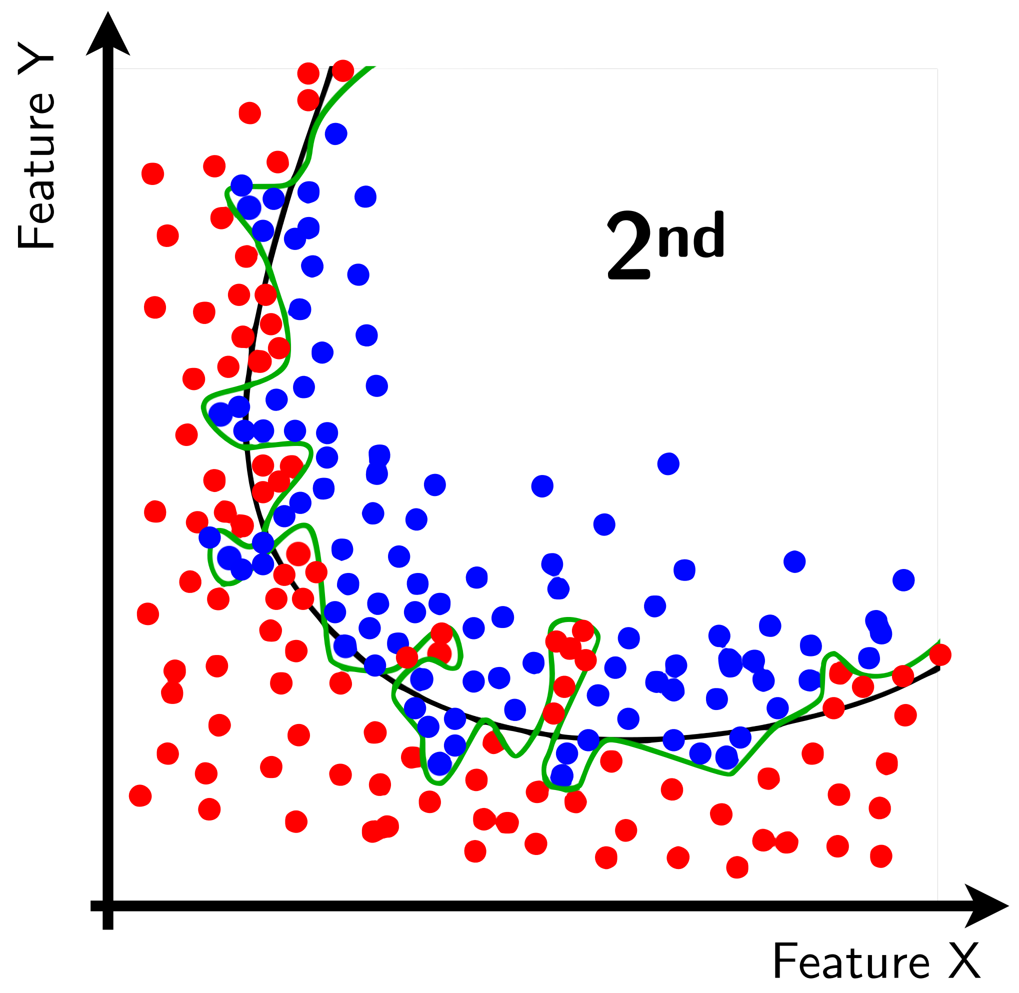


- **2nd scenario: volatile ground truth**
  - Network should remain capable to model that
    - *Some* weights need to be large

- **Goal**: "Don't depend on **few, large weights** to model volatile behavior,
  but rather allow network to increase **multiple weights moderately** if dictated by training data"

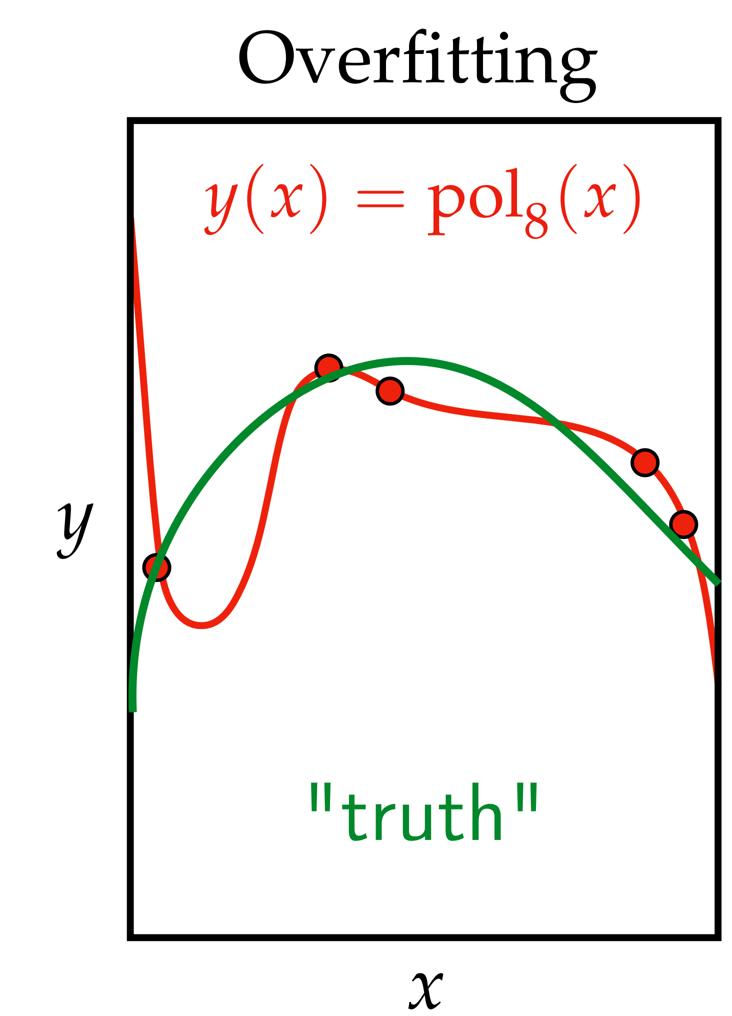
- **1st scenario: overtraining**
  - Small number of weights become large
    - $y(x) = 2.1 + 3.9x^4 - 4.6x^6 + 4.4x^7 + \ldots$ (example)
    - Should be avoided
- **2nd scenario: volatile ground truth**
  - Network should remain capable to model that
    - *Some* weights need to be large

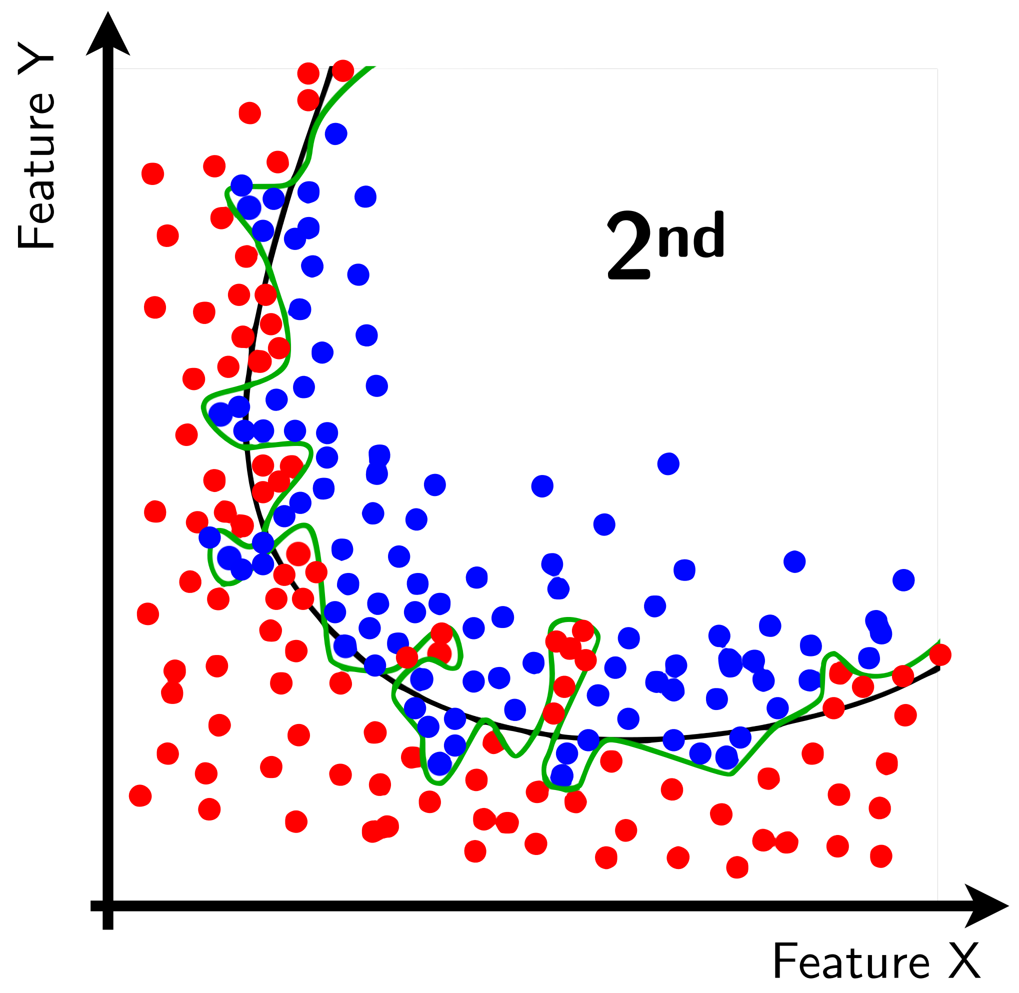




- **Goal**: "Don't depend on **few, large weights** to model volatile behavior,
  but rather allow network to increase **multiple weights moderately** if dictated by training data"

- **1st scenario: overtraining**
  - Small number of weights become large
    - $y(x) = 2.1 + 3.9x^4 - 4.6x^6 + 4.4x^7 + \ldots$ (example)
    - Should be avoided

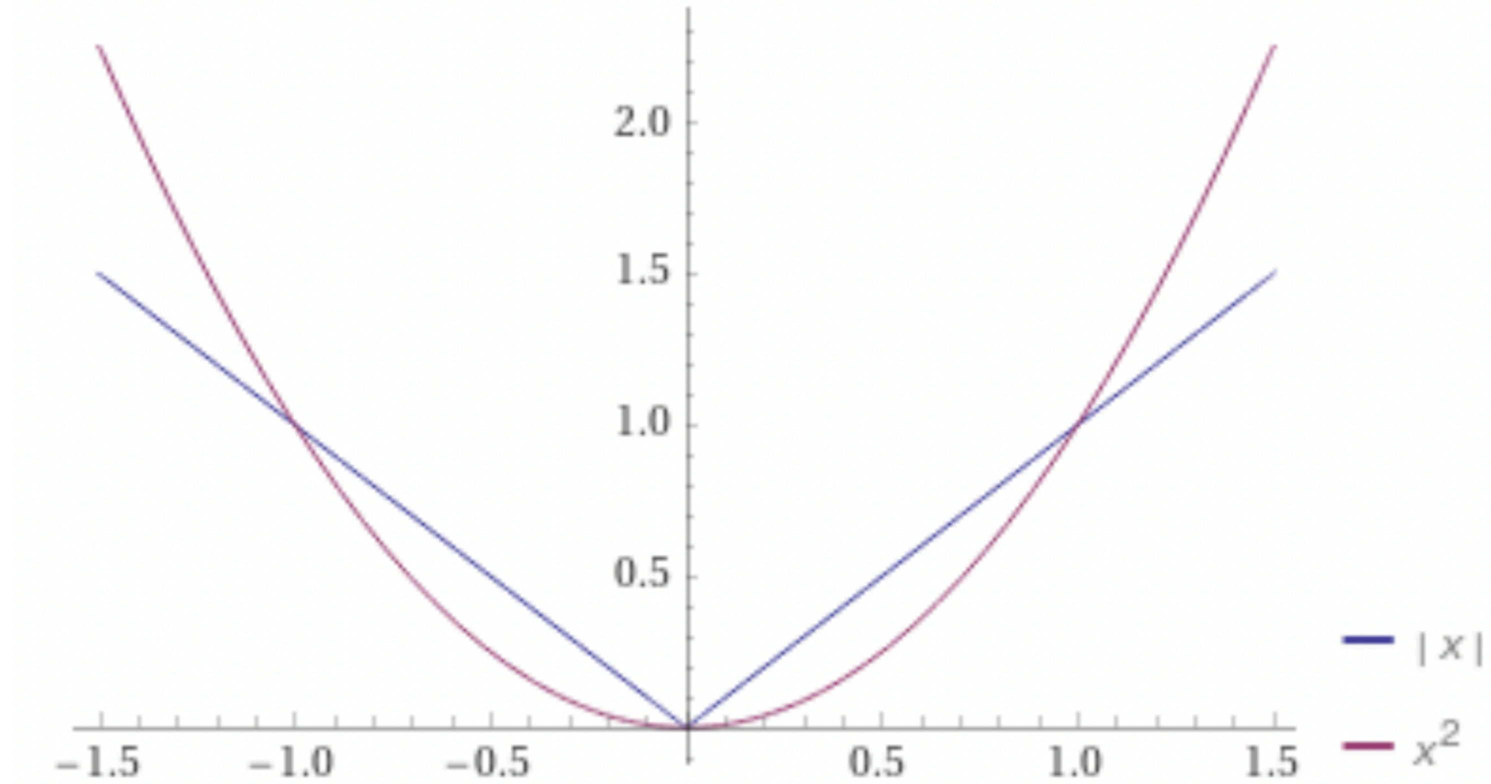




- **2nd scenario: volatile ground truth**
  - Network should remain capable to model that
    - *Some* weights need to be large

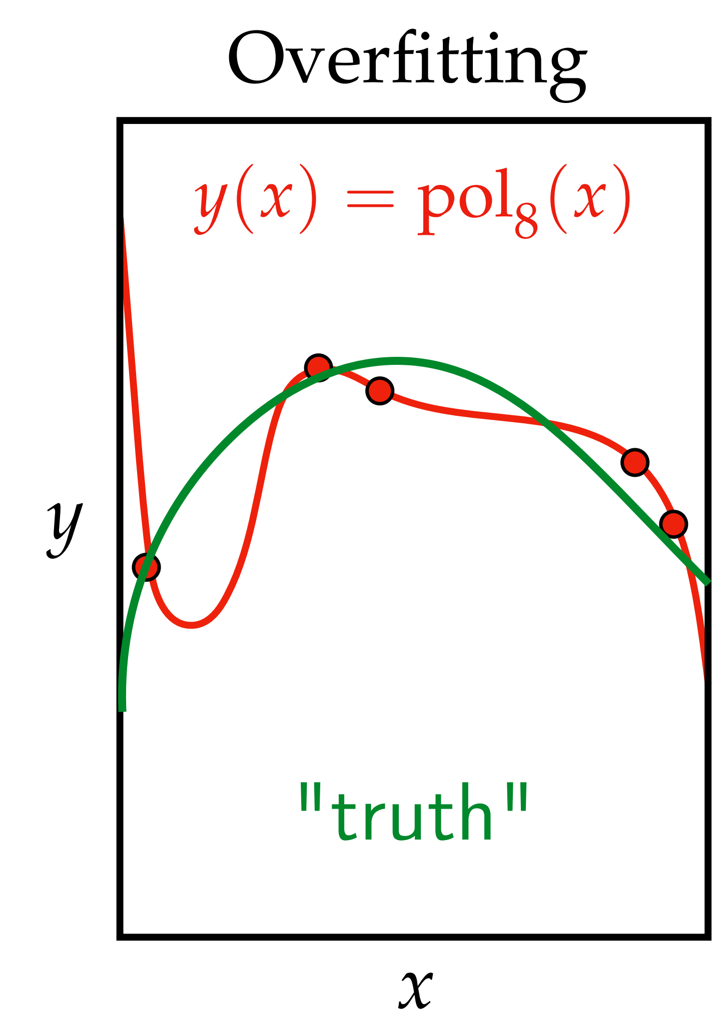
- **Goal**: "Don't depend on **few, large weights** to model volatile behavior,
  but rather allow network to increase **multiple weights moderately** if dictated by training data"

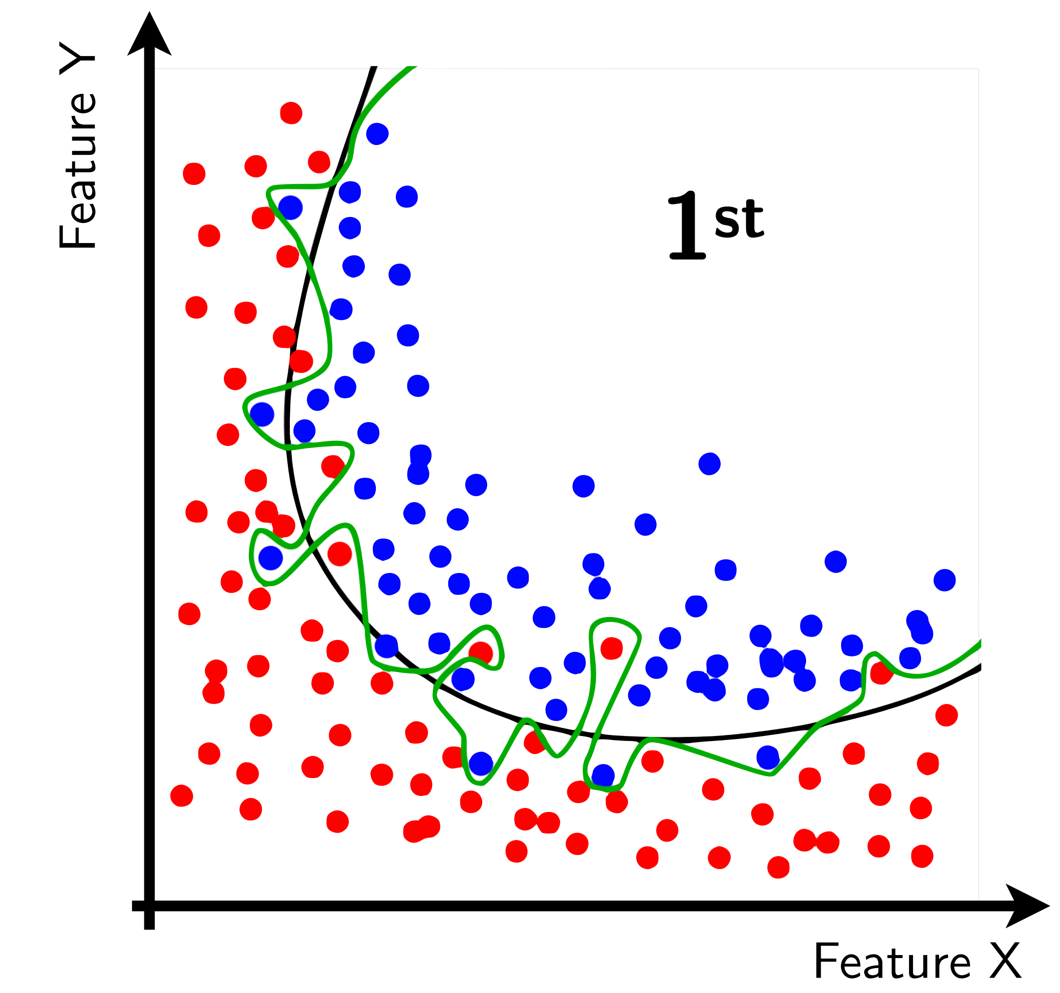
- **Approach**: regularization losses $L_1 = \sum_i^{weights} |\omega|$ or $L_2 = \sum_i^{weights} \omega^2$ $\longrightarrow$ Added to main loss, e.g. $L_{total} = L_{pred.\ vs.\ truth} + \lambda \cdot L_{1,2}$
  - λ is a free parameter that should mediate between regularization and main loss
  - We usually pick L_2
    - Gradients $\partial L_2 / \partial \omega$ still depend on ω, better feedback for updating ω's
    - Small penalty for $\omega < 1$, very high for $\omega > 1$



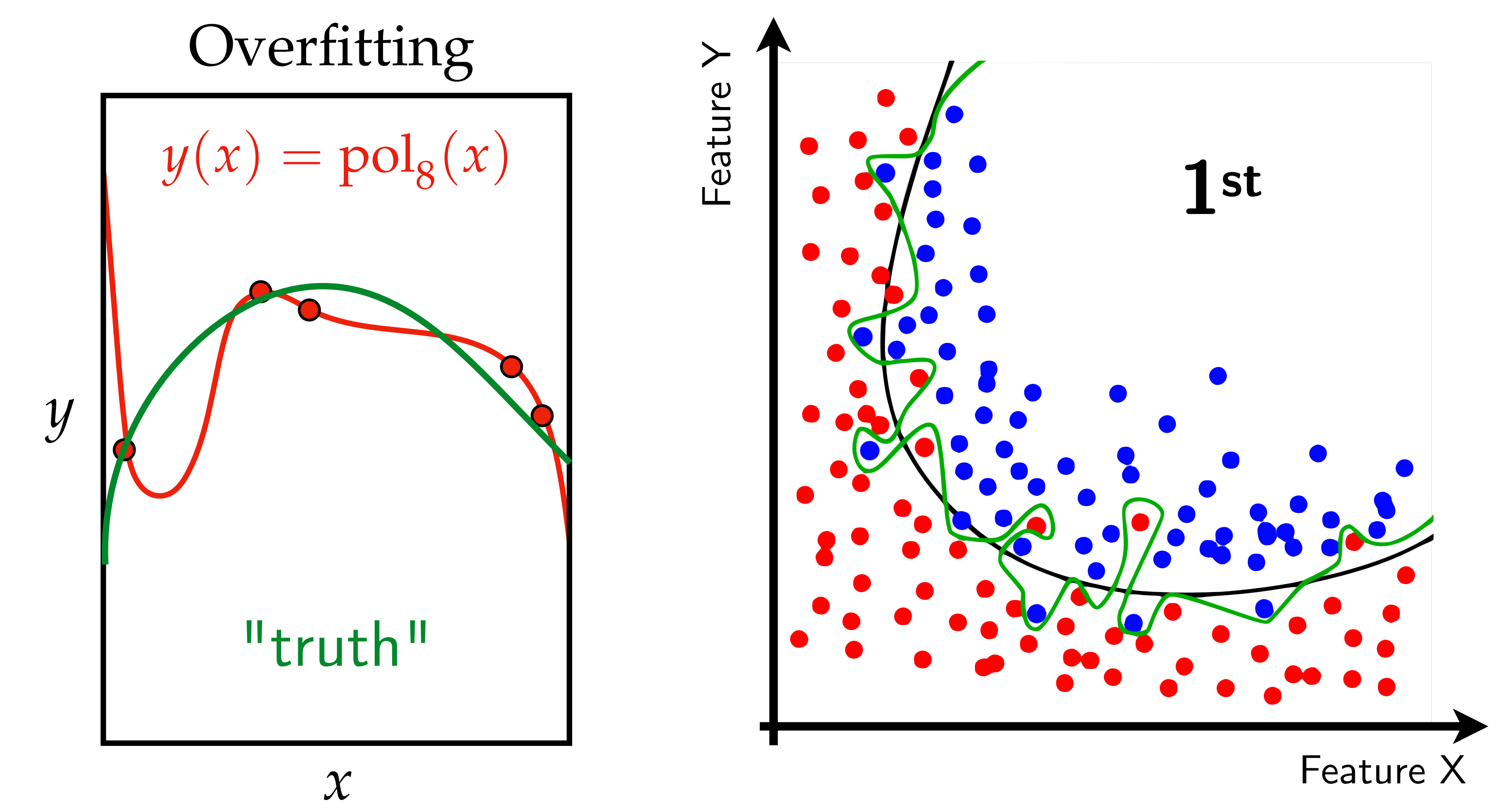
- **1st scenario: overtraining**
  - Small number of weights become large
  - **Also**, other weights adapt to that
    - E.g. when $\omega_1 \gg 1$, then $\omega_2 \ll 1$, and $\omega_3 \approx 0$, and ...
    - Fine tuning problem among weights

- **Goal**: "Introduce slightly stochastic behavior to the learning process to reduce weights' reliance on one another"



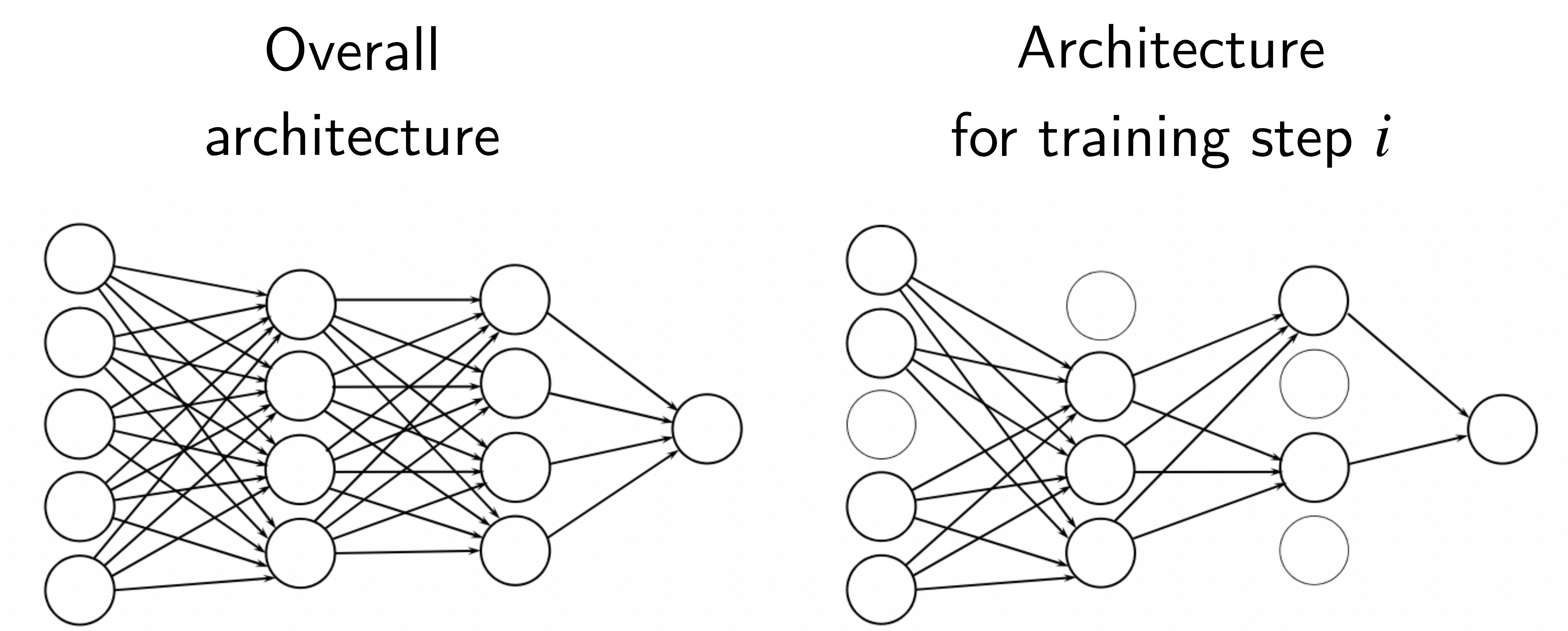


- **1st scenario: overtraining**
  - Small number of weights become large
  - **Also**, other weights adapt to that
    - E.g. when $\omega_1 \gg 1$, then $\omega_2 \ll 1$, and $\omega_3 \approx 0$, and ...
    - Fine tuning problem among weights



- **Goal**: "Introduce slightly stochastic behavior to the learning process to reduce weights' reliance on one another"

- **Approach**: random unit dropout
  - Only applied during training
  - During the forward pass of step i, randomly pick units with a dropout probability p that are not considered
  - However, input to each unit in the next layer scaled down
    - Scale back up by $1/(1-p)$ ✓
  - Repeat in next step $i+1$, picking **different** units



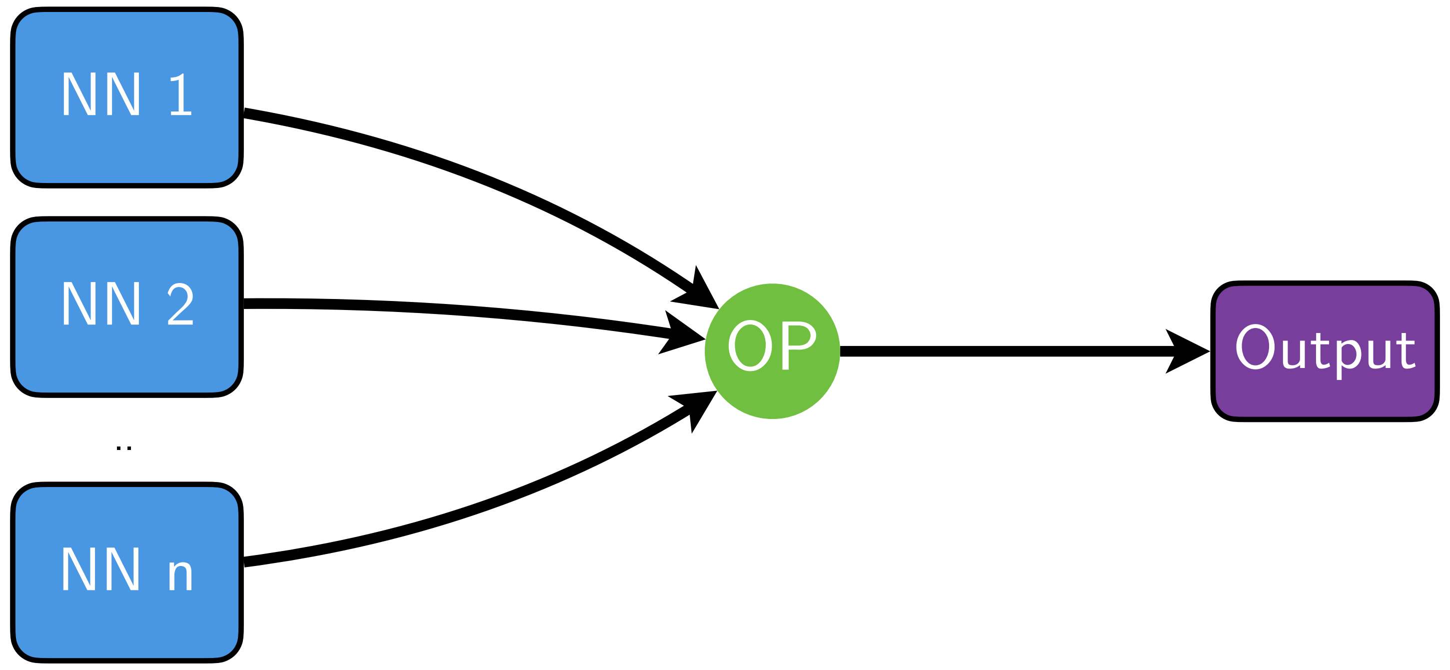
from "Deep learning in Physics Research", Erdmann *et al.*

- Typical dropout rates between 10% and 50%
- Side note: SELU activation requires "AlphaDropout" (preserves μ, σ^2)

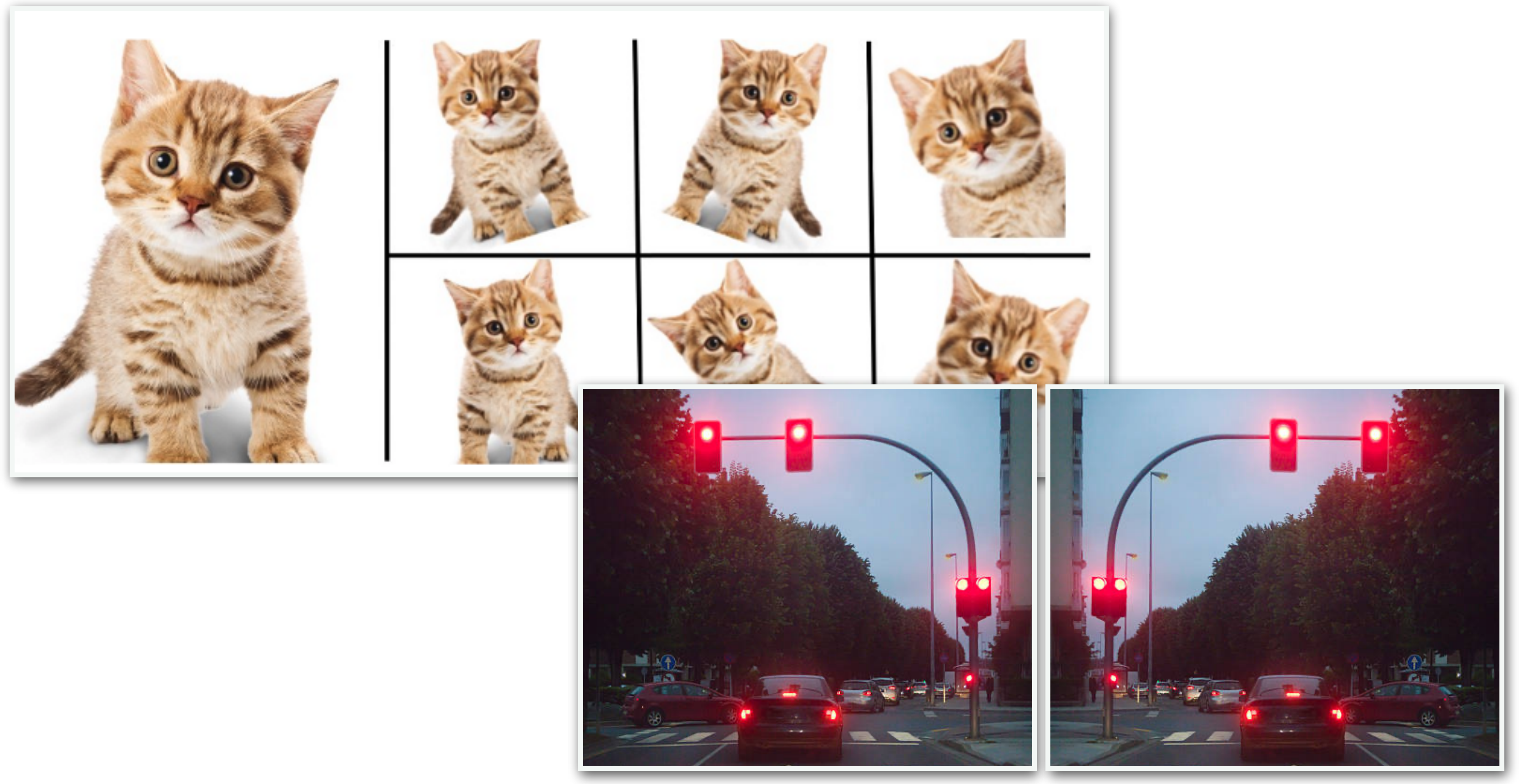
Low-hanging-🍇?

✗ • **More, representative data**
- Presumably not trivial

✓✓ • **Ensemble training**
- Averaging or "majority votes" across multiple networks



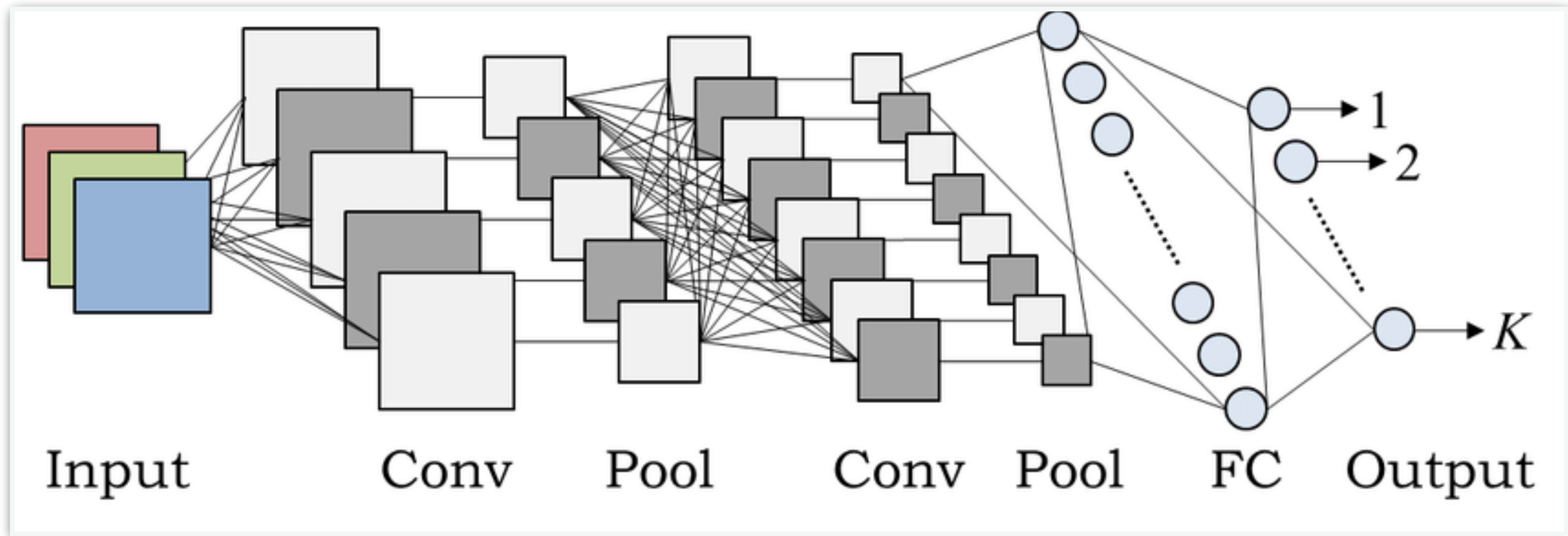
✗✗ • **Data augmentation**
- Artificially extend your dataset exploiting known symmetries
- Highly non-trivial as underlying *pdf*'s should be preserved



✗✗ • **Noise injection**
- Smears input feature *pdf*'s but choice non-trivial too
- Trade-off: overtraining vs. loss in performance

(✓) • **Fewer parameters, yet shared across network**
- Requires change in architecture

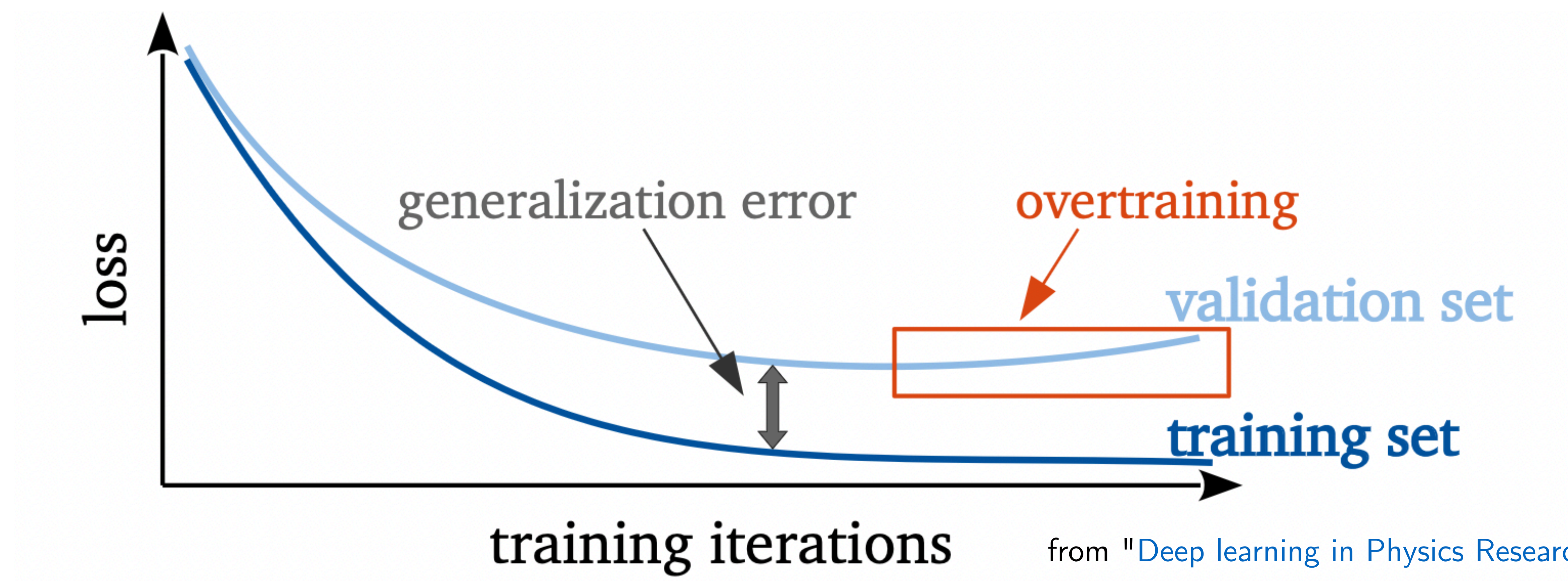
→ See CNN lectures



✓✓✓ • **Early stopping**
- Monitor generalization error and stop training at threshold

→ Next slides

- Suppose you split your data first into a "**training set**" and an **independent** "**validation set**"
  - Perform training with former
  - Every n batches / after each epoch / ..., check loss or other metrics on latter



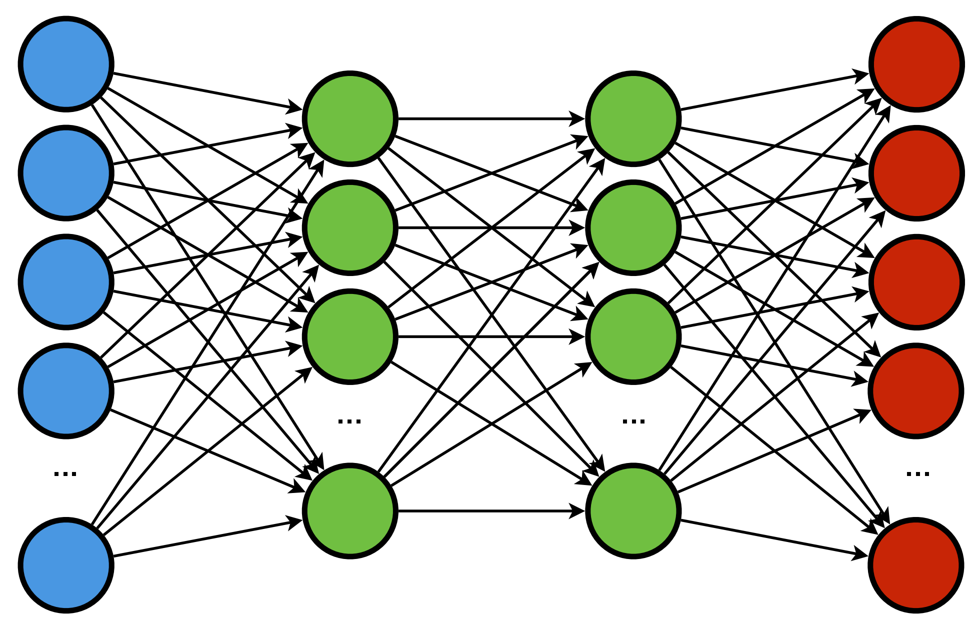
from "Deep learning in Physics Research", Erdmann *et al.*

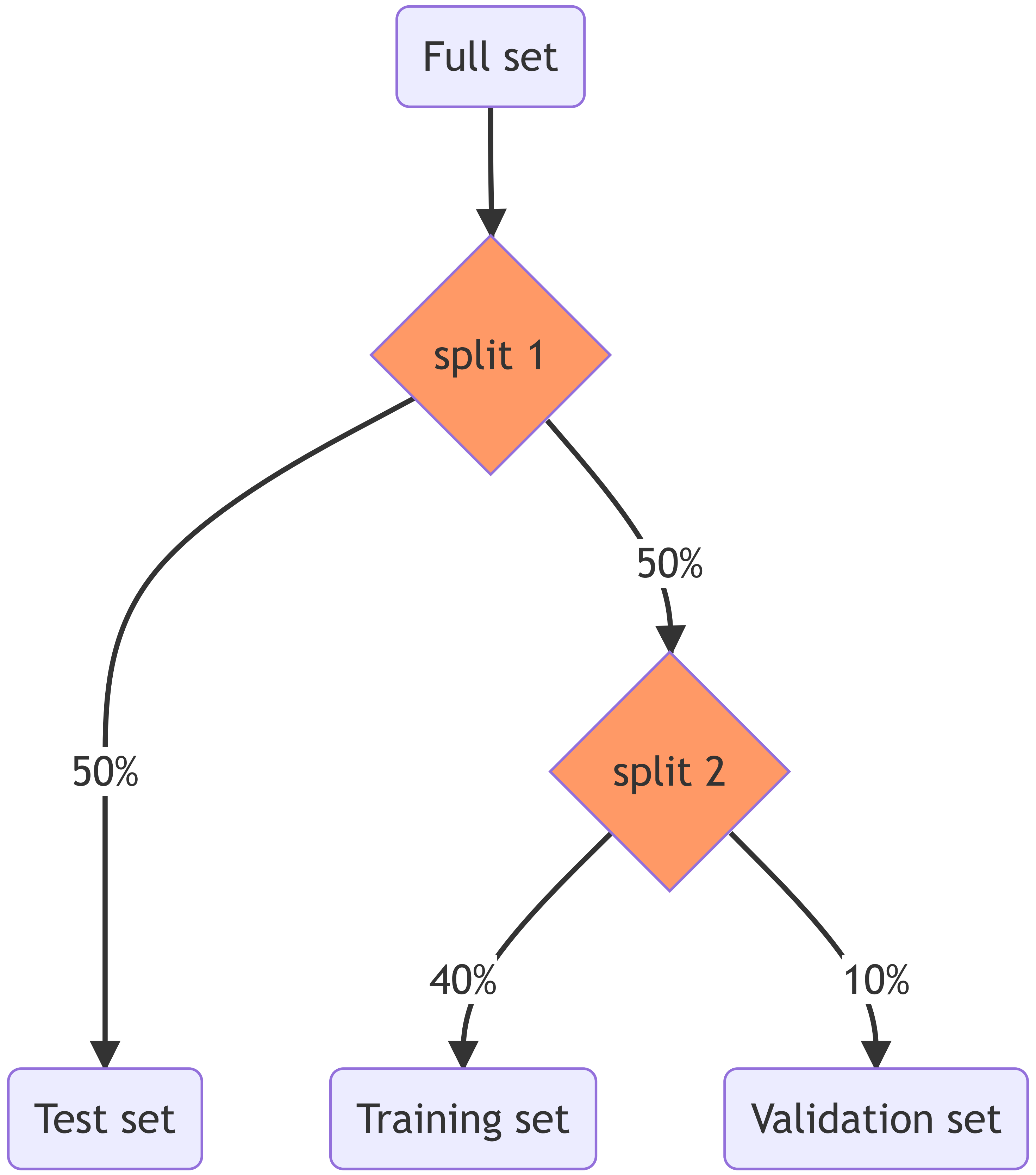
> ▻ A soon as metrics diverge above level of statistical fluctuations, overtraining occurred

- What now?
  - Difference between metrics define *generalization error*
  - Decide whether error is acceptable, and if not, **stop the training** and **save the best* model**

- Side note: to save resources, you can also stop the training early in case no improvement happened for some time

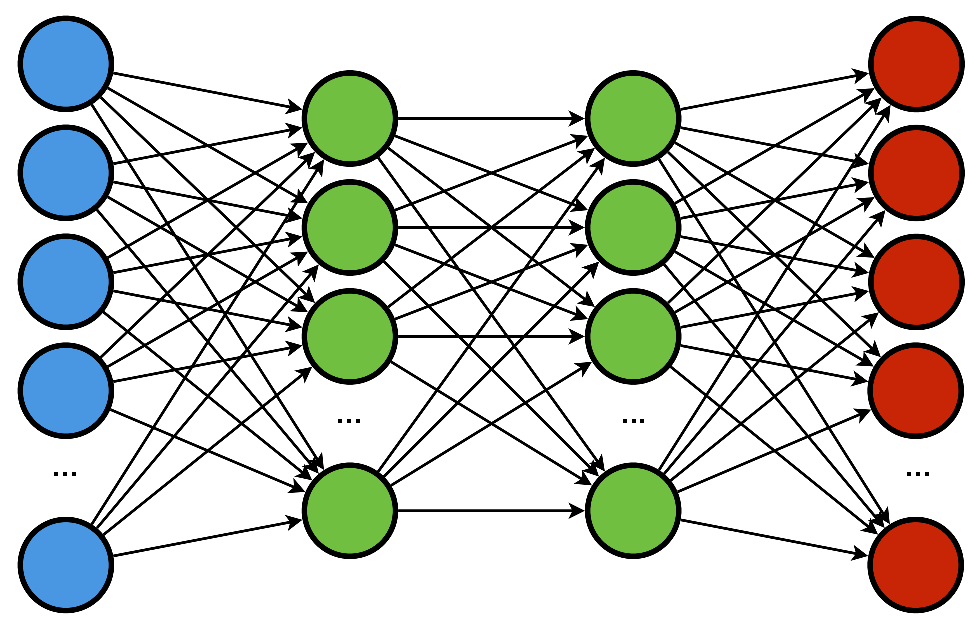
- **"training" ↔ "validation" ↔ "test"**
  - Divide your data into three, fully independent datasets
  - **training**
    - Samples used for *weight optimization* through back-propagation
  - **validation**
    - Samples used during training for immediate *validation* & possibly *hyper-parameter optimization* (e.g. architecture itself, see later)
  - **test**
    - After information content of **training** and **validation** set has been exploited, all results are to be reported on independent **test** set
    - Decouples your results from potential overestimation due to overtraining



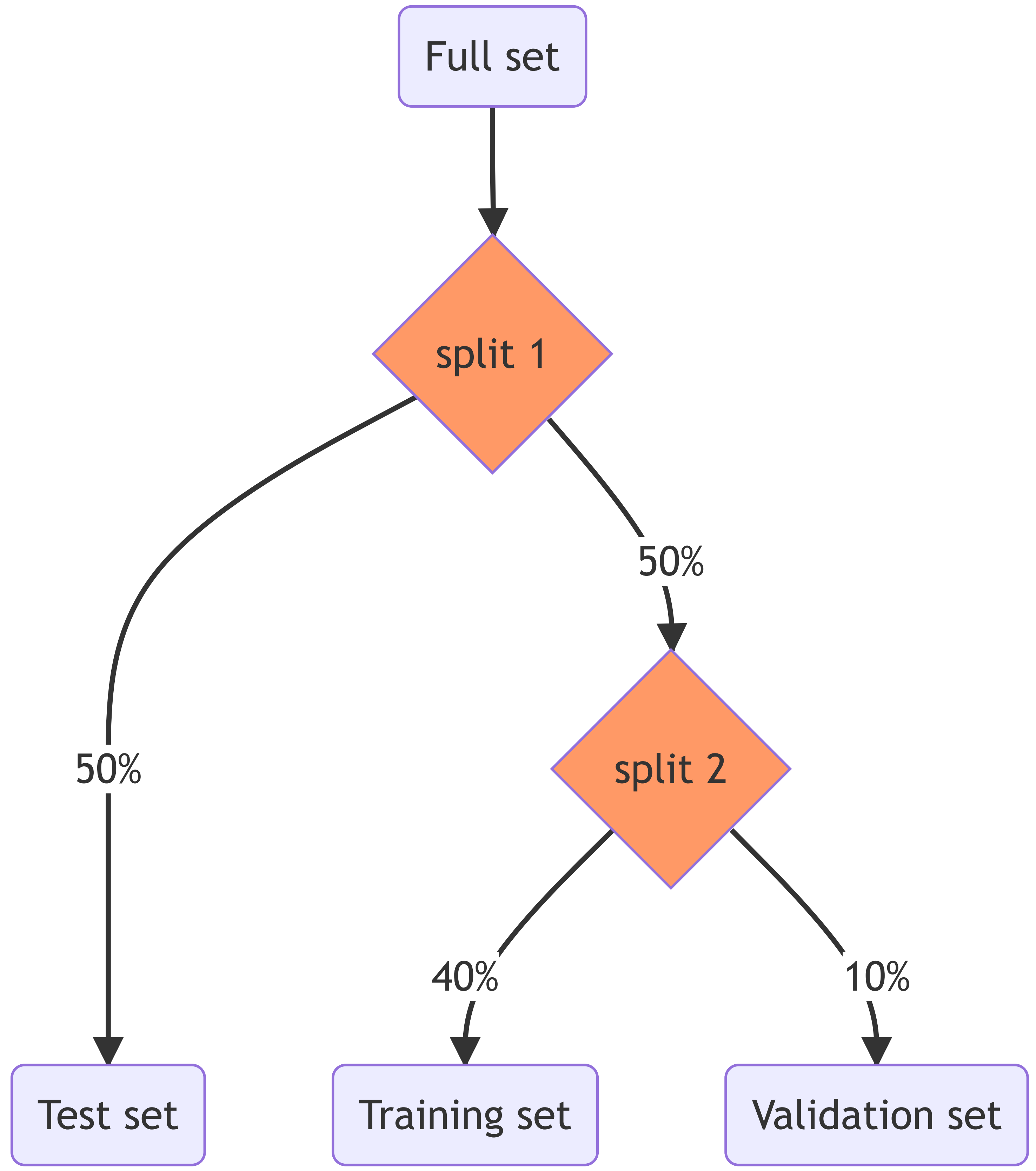


(example numbers)

- **"training" ↔ "validation" ↔ "test"**
  - Divide your data into three, fully independent datasets
  - **training**
    - ▻ Samples used for *weight optimization* through back-propagation
  - **validation**
    - ▻ Samples used during training for immediate *validation* & possibly *hyper-parameter optimization* (e.g. architecture itself, see later)
  - **test**
    - ▻ After information content of **training** and **validation** set has been exploited, all results are to be reported on independent **test** set
    - ▻ Decouples your results from potential overestimation due to overtraining

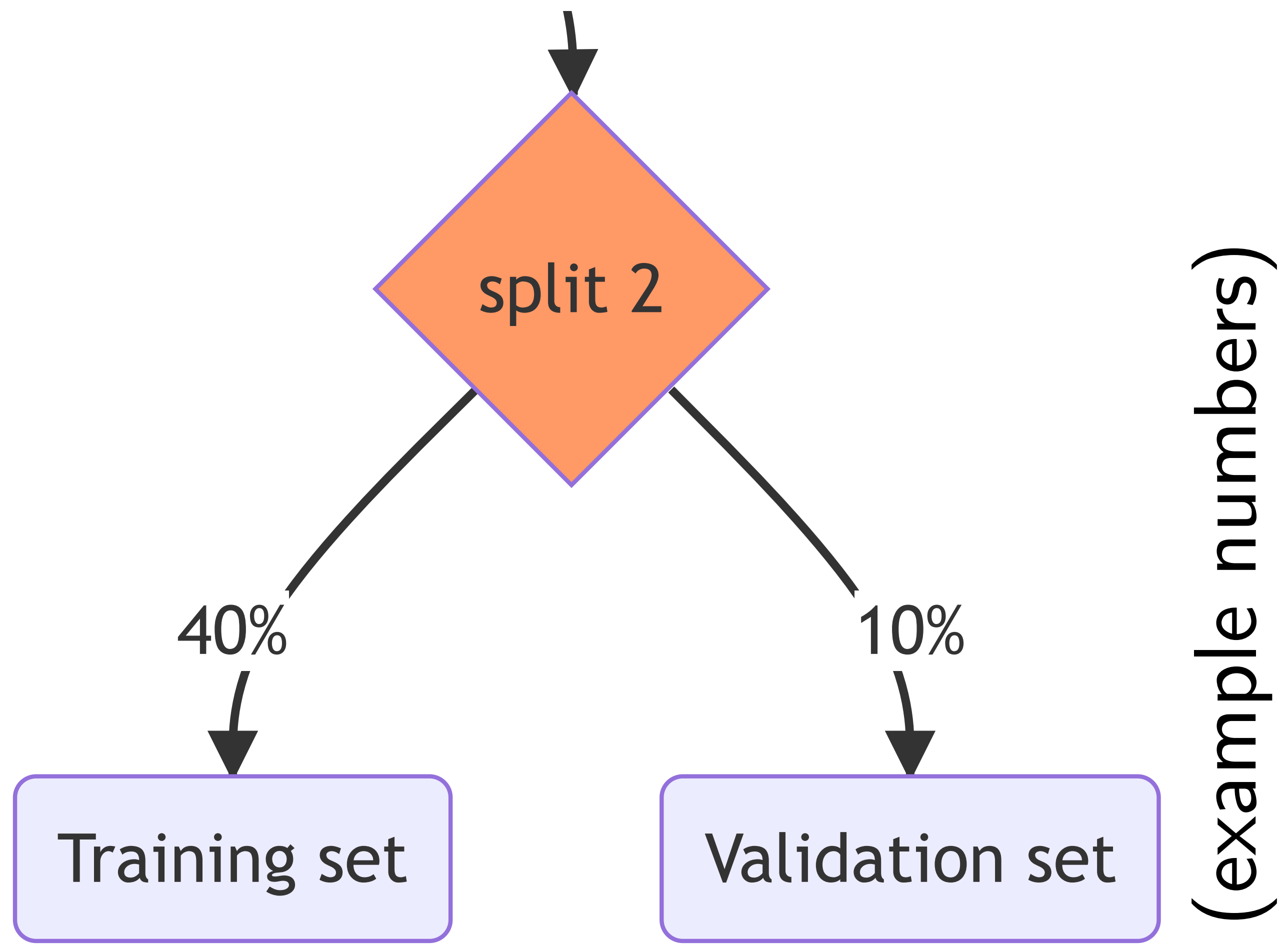


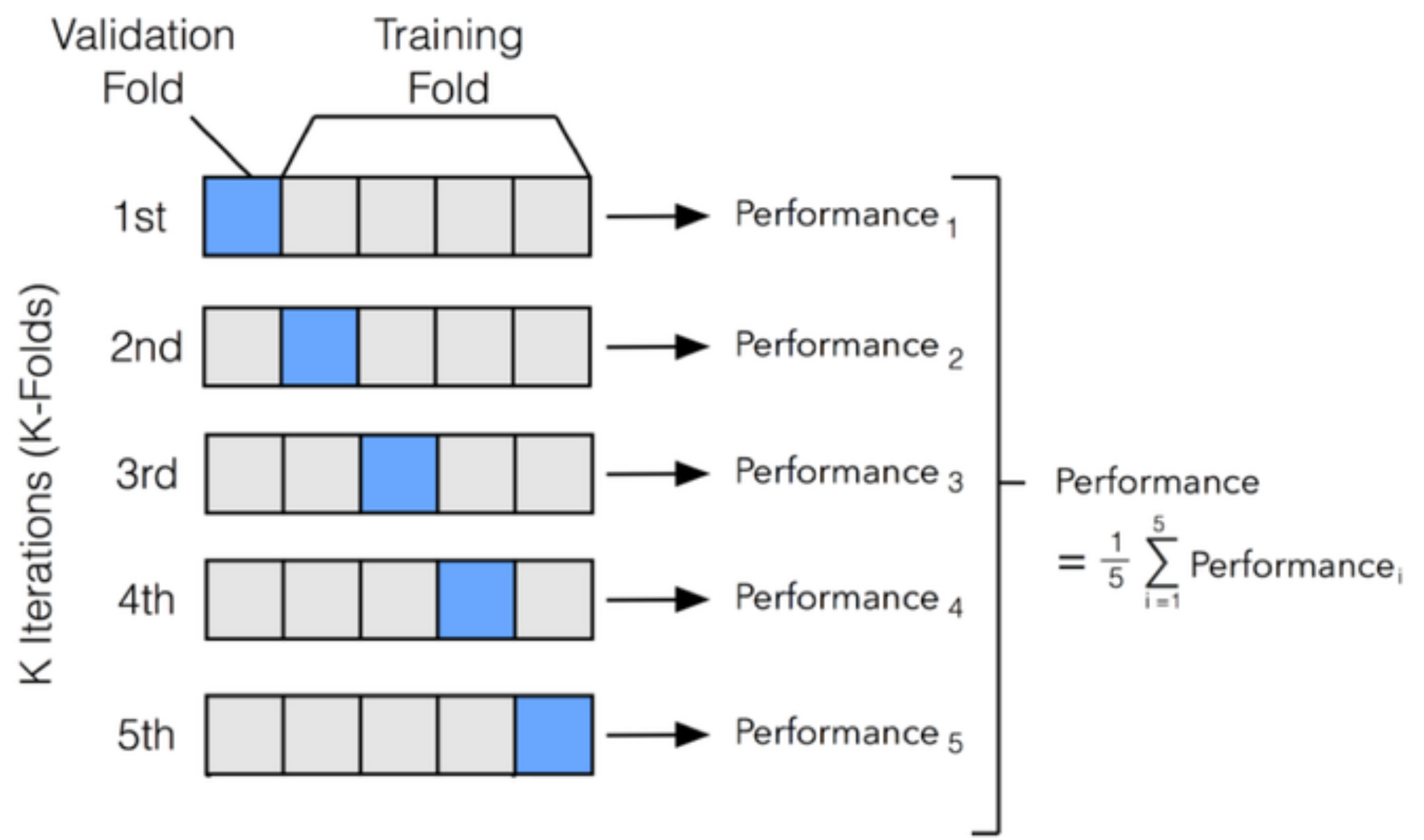
- *Practicalities*
  - Split randomly *once*, or *repeatedly but deterministic*
    - ▻ Avoids test samples being randomly used at any given time
  - Think about splitting fractions
    - ▻ All sets should be statistically significant, define fractions case-by-base
  - In low stat. scenarios, consider a second training with reversed splits
    - ▻ Requires dedicated treatment ... also, why stop at two?



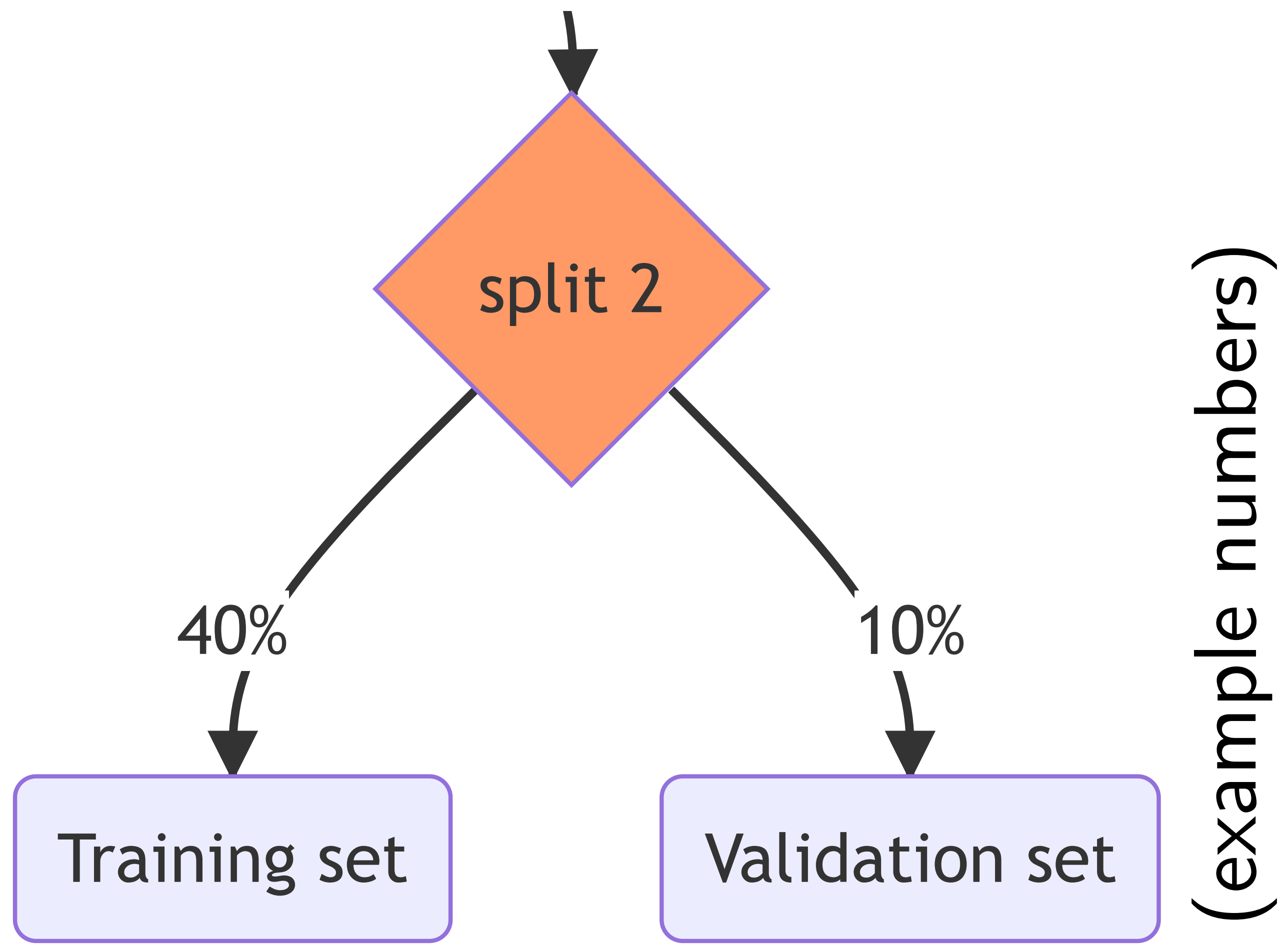
(example numbers)

- **Variant 1**: Reconsider the split into **training** and **validation** sets
  - If statistics is an issue, you can incorporate the validation set into the training
  - Example:
    - ▻ Split into 5 **folds** à 10%
    - ▻ Perform training on 4 folds, validate with remaining fold
    - ▻ Perform 5 trainings in total
    - ▻ Final model consists of ensemble of networks
  - ✓ Exploited **all non-test** samples
  - ✓ Increased overtraining robustness



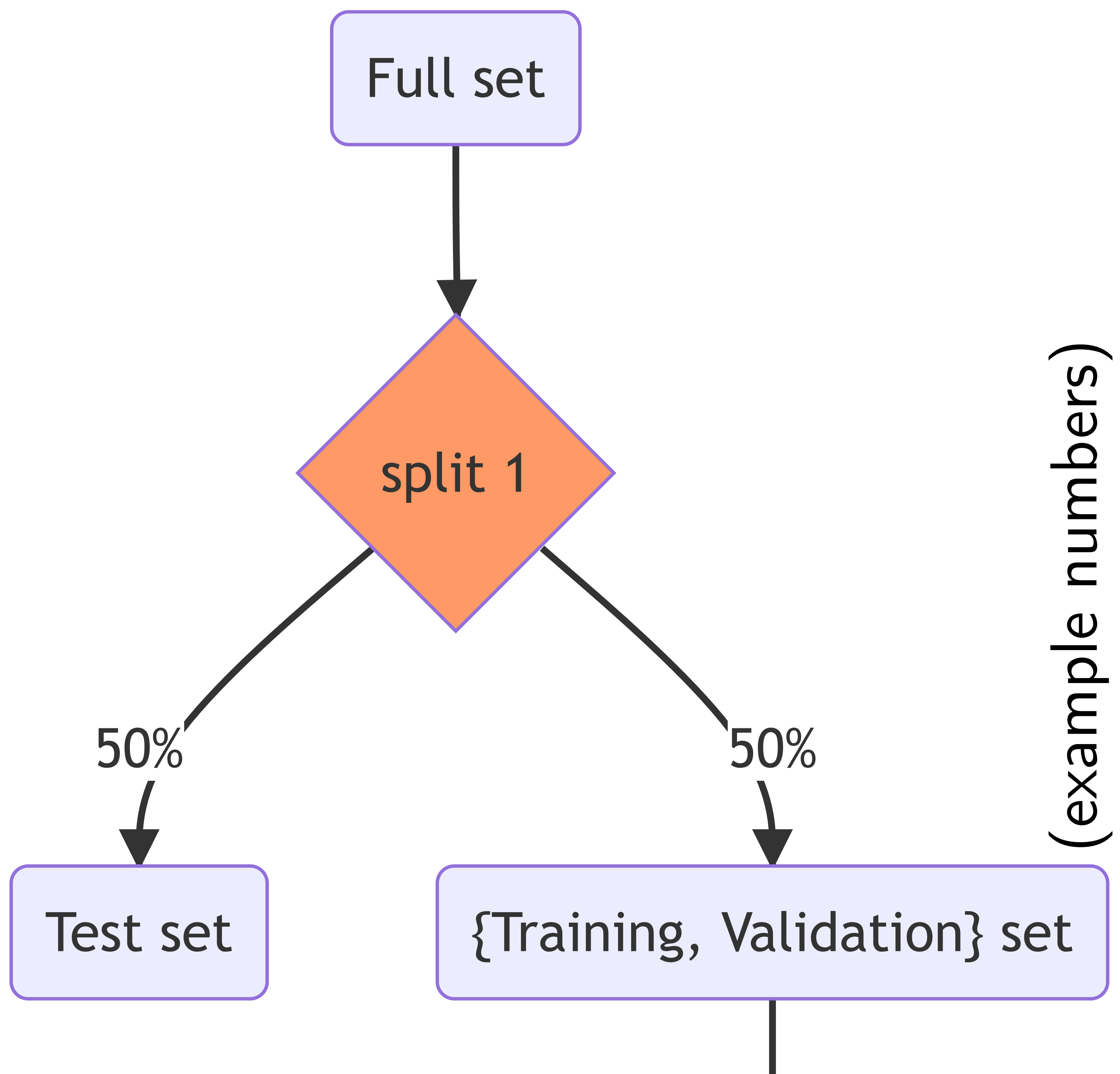


- **Variant 1**: Reconsider the split into **training** and **validation** sets
  - If statistics is an issue, you can incorporate the validation set into the training
  - Example:
    - Split into 5 **folds** à 10%
    - Perform training on 4 folds, validate with remaining fold
    - Perform 5 trainings in total
    - Final model consists of ensemble of networks
  - ✓ Exploited **all non-test** samples
  - ✓ Increased overtraining robustness



- **Variant 2**: Reconsider the split into **test** and {**training**,**validation**} set
  - If statistics is an even bigger issue, extend **folds** to **test** set
  - Example:
    - Split all samples into k=10 **folds** à 10%
    - Perform trainings on k-1 folds (1 for validation), *retain* remaining fold
    - Perform k trainings in total
    - For results, evaluate samples with network of opposite k-1th fold
    - For samples not seen during training, *random* / *ensemble* / *sub-ensembling*
  - ✓ Exploited **all** samples
  - ☞ Potential implications when real data involved

that's where it gets tricky



Let's revisit the initial example

Note: No regularization, no dropout

Note: No regularization, no dropout

- **Tasks**

1. Open playground.tensorflow.org
2. Pick the  dataset and create a network that visibly overtrains
3. Play around with the settings (except for the "data") to make the overtraining as drastic as possible
4. Step by step, start changing network parameters to suppress overtraining while
   - preserving a reasonable loss
   - maintaining a quick training process
5. Now, change some "data" settings. Increase the statistics ("ratio of training to test data") and perform two trainings with low and high noise settings. Are they susceptible to overtraining?
6. Change the data set and play.

5. Model optimization

Imagine you're on a *hike*

as **dense fog** rolls in ...

- Vision below ~ 1m
  - You only see the **ground below you**
- You want to get to the hut **in the valley**
- You want to get there **fast**!
  - It's getting dark
- Your phone can measure the elevation & slope at your loc.
  - Battery is dying
- What's your plan?

Imagine you're on a *hike* as **dense fog** rolls in ...

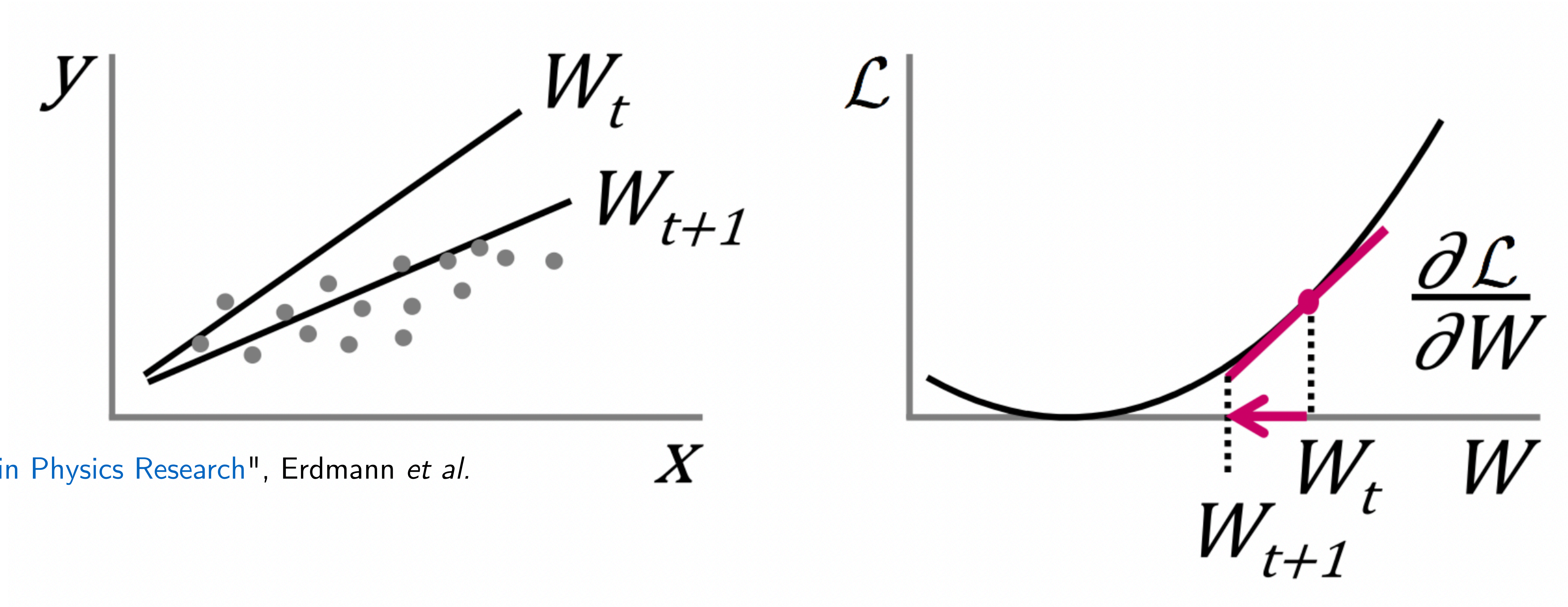
- *How to get down to the hut?*
  1. Measure local gradient
  2. Walk in direction of steepest for slope for 1min
  3. Repeat

  → **Gradient descent!**

- *But what if*
  - your battery won't hold?
  - you walk into a small well?
  - you feel like you run in circles?

ML: find weights ω_1 and ω_2 of a model f that minimize the loss $L(f(x\,|\,\omega_1, \omega_2), y)$ given data x and truth y

- From Dennis' lecture



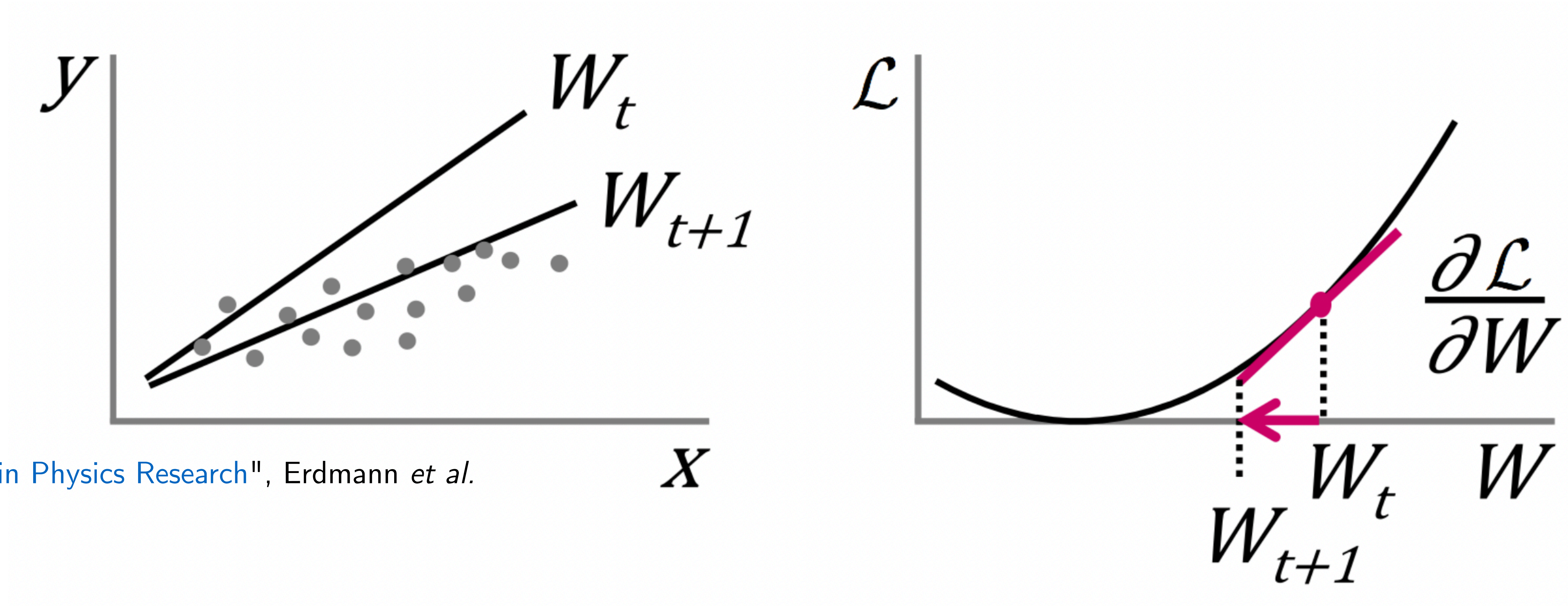
from "Deep learning in Physics Research", Erdmann *et al.*

Update rule

$$\omega_{t+1} = \omega_t - \alpha \cdot \frac{\partial L}{\partial \omega_t}$$

learning rate gradient

- From Dennis' lecture



from "Deep learning in Physics Research", Erdmann *et al.*

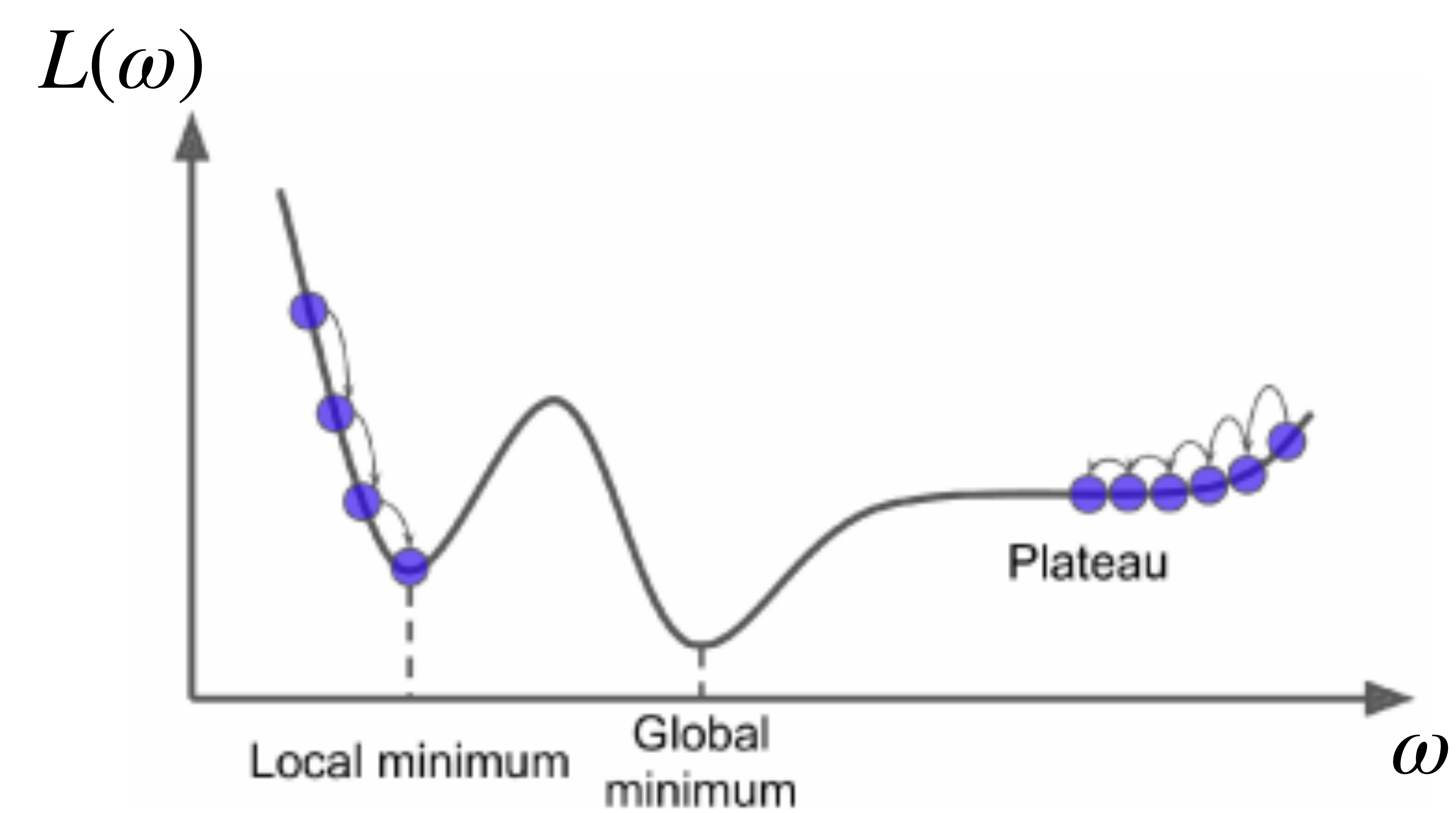
Update rule

$$\omega_{t+1} = \omega_t - \alpha \cdot \frac{\partial L}{\partial \omega_t}$$

learning rate

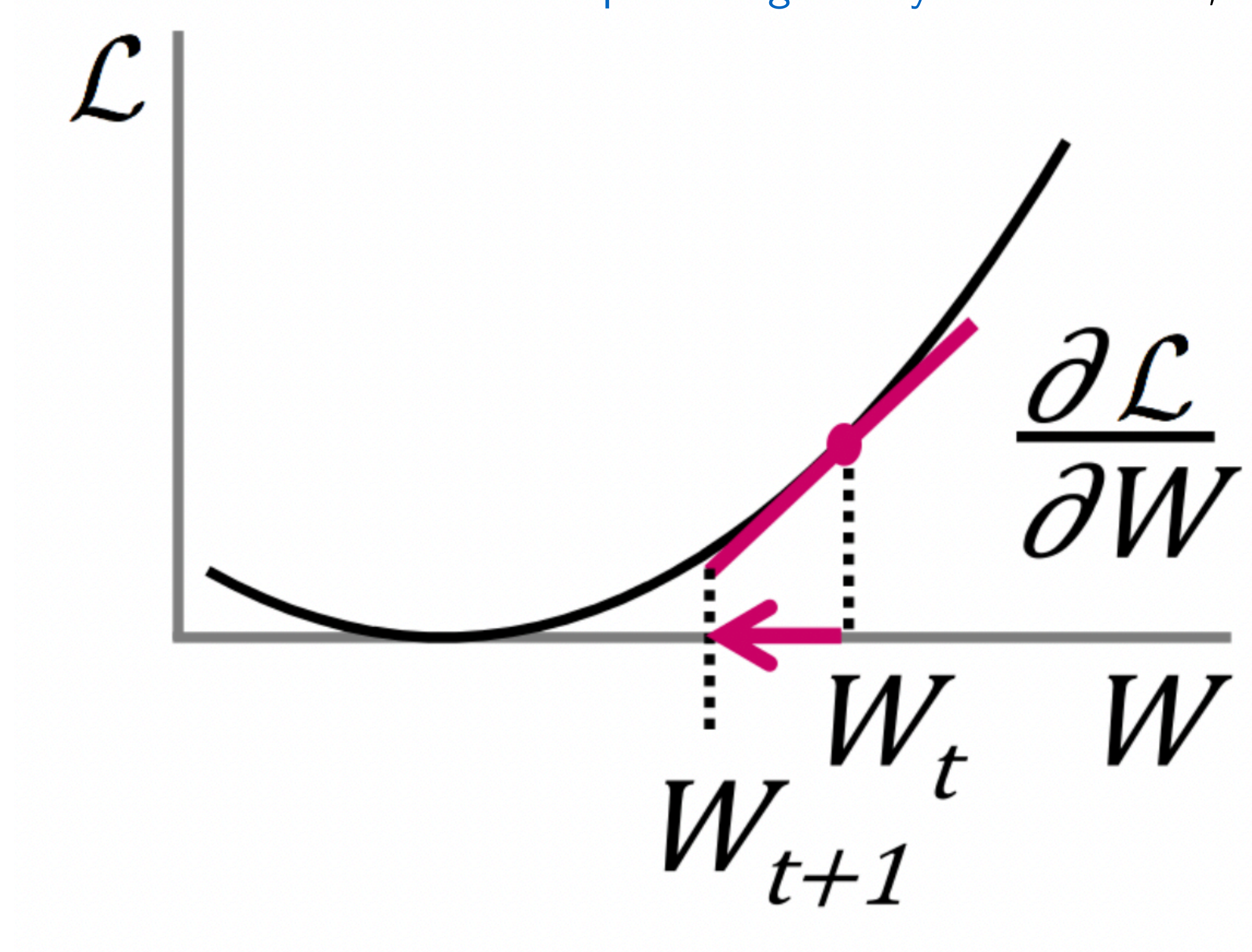
gradient

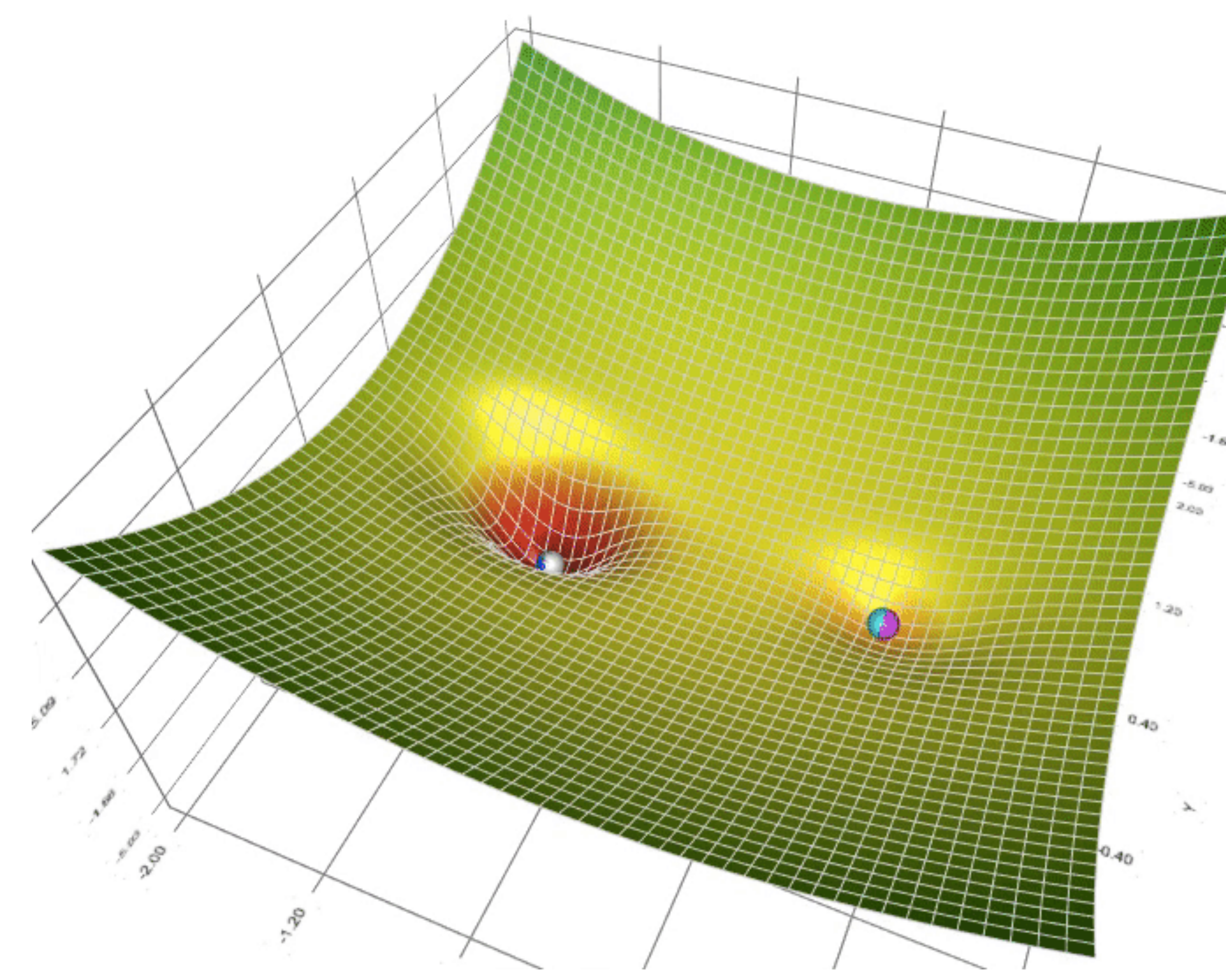
- **How to find the global minimum faster and avoid local minima?**
  - Several techniques available that amend the update rule
  - Influenced by the scenario of a moving object on a slope, introducing:
    - *Adaptive* learning rate per parameter ω
    - *Momentum*, to overcome local minima and fluctuations
    - *Friction*, to reduce momentum step-wise



1. **Standard update** of particular weight ω from $t \rightarrow t+1$
   - $\omega_{t+1} = \omega_t - \alpha \cdot \dfrac{\partial L}{\partial \omega_t} \quad \rightarrow \quad \Delta\omega_t = -\,\alpha \cdot \dfrac{\partial L}{\partial \omega_t}$

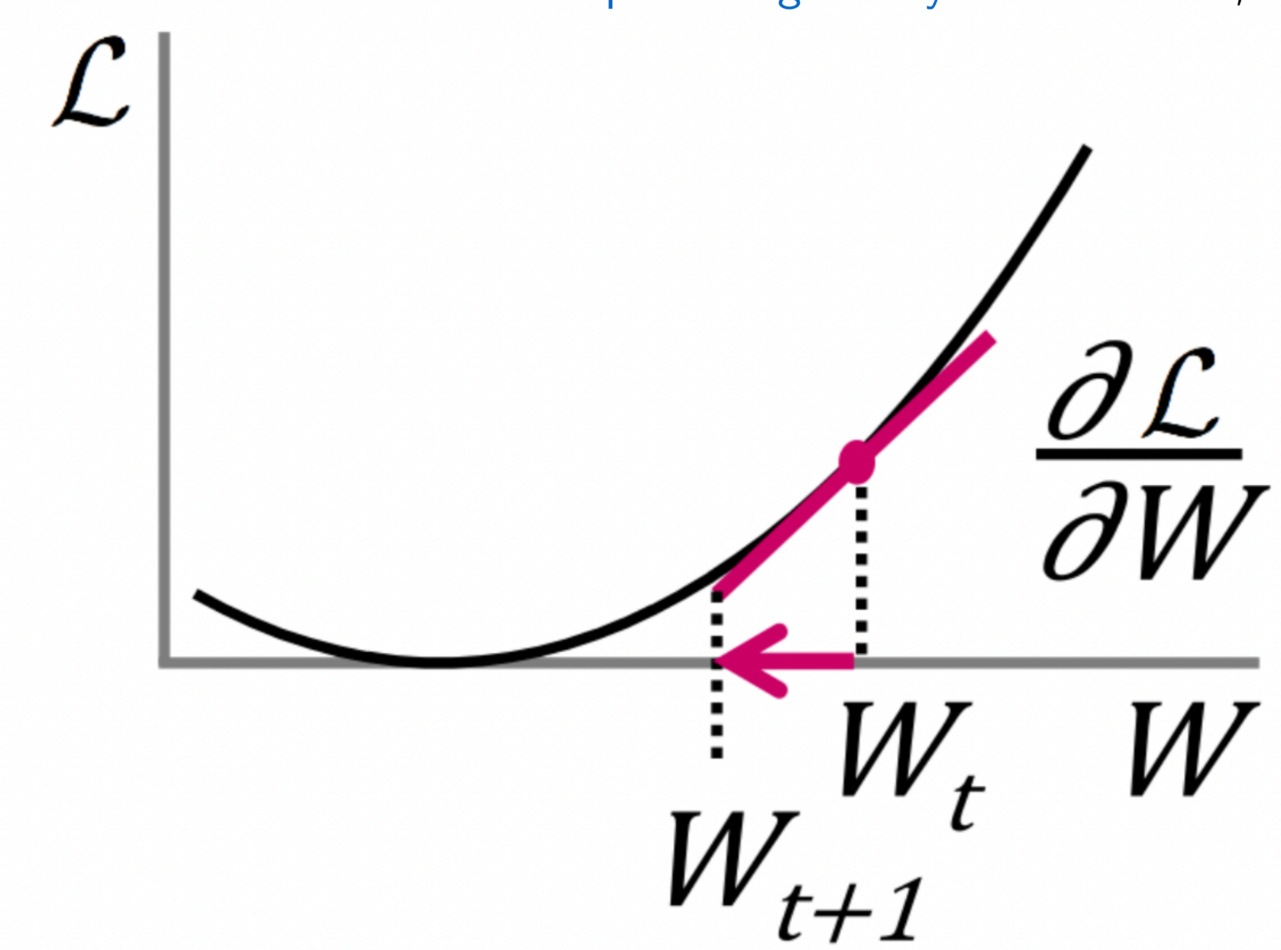
from "Deep learning in Physics Research", Erdmann *et al.*

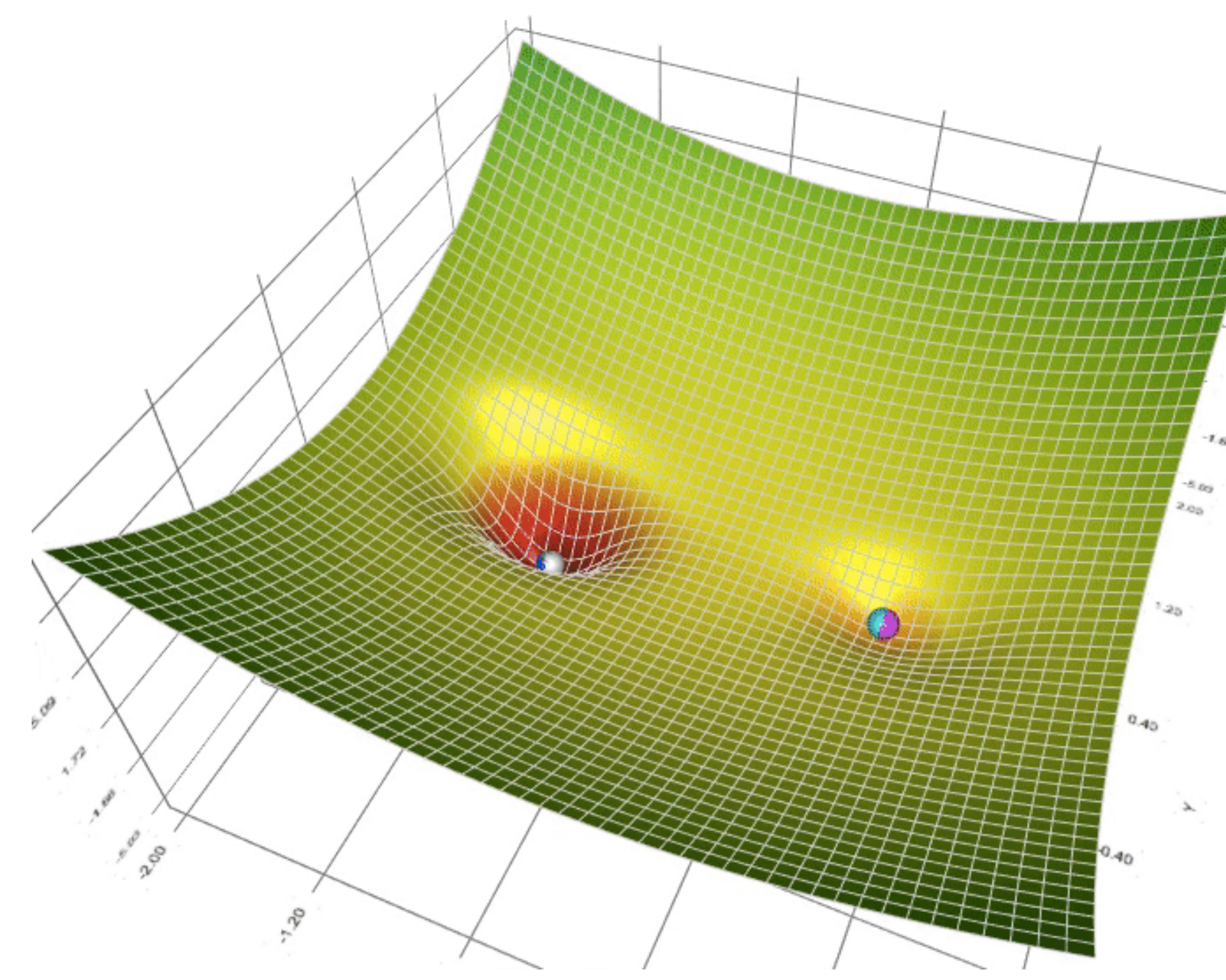




1. **Standard update** of particular weight ω from $t \rightarrow t+1$
   - $\omega_{t+1} = \omega_t - \alpha \cdot \frac{\partial L}{\partial \omega_t} \quad \rightarrow \quad \Delta\omega_t = -\,\alpha \cdot \frac{\partial L}{\partial \omega_t}$
2. **Adagrad**: Remember all past gradients and adapt $\alpha \rightarrow \alpha_t$
   - $\nu_t = \sum_{\tau=1}^{t} \left(\frac{\partial L}{\partial \omega_\tau} \right)^2$ and $\alpha_t = \frac{\alpha}{\sqrt{\nu_t} + \epsilon}$

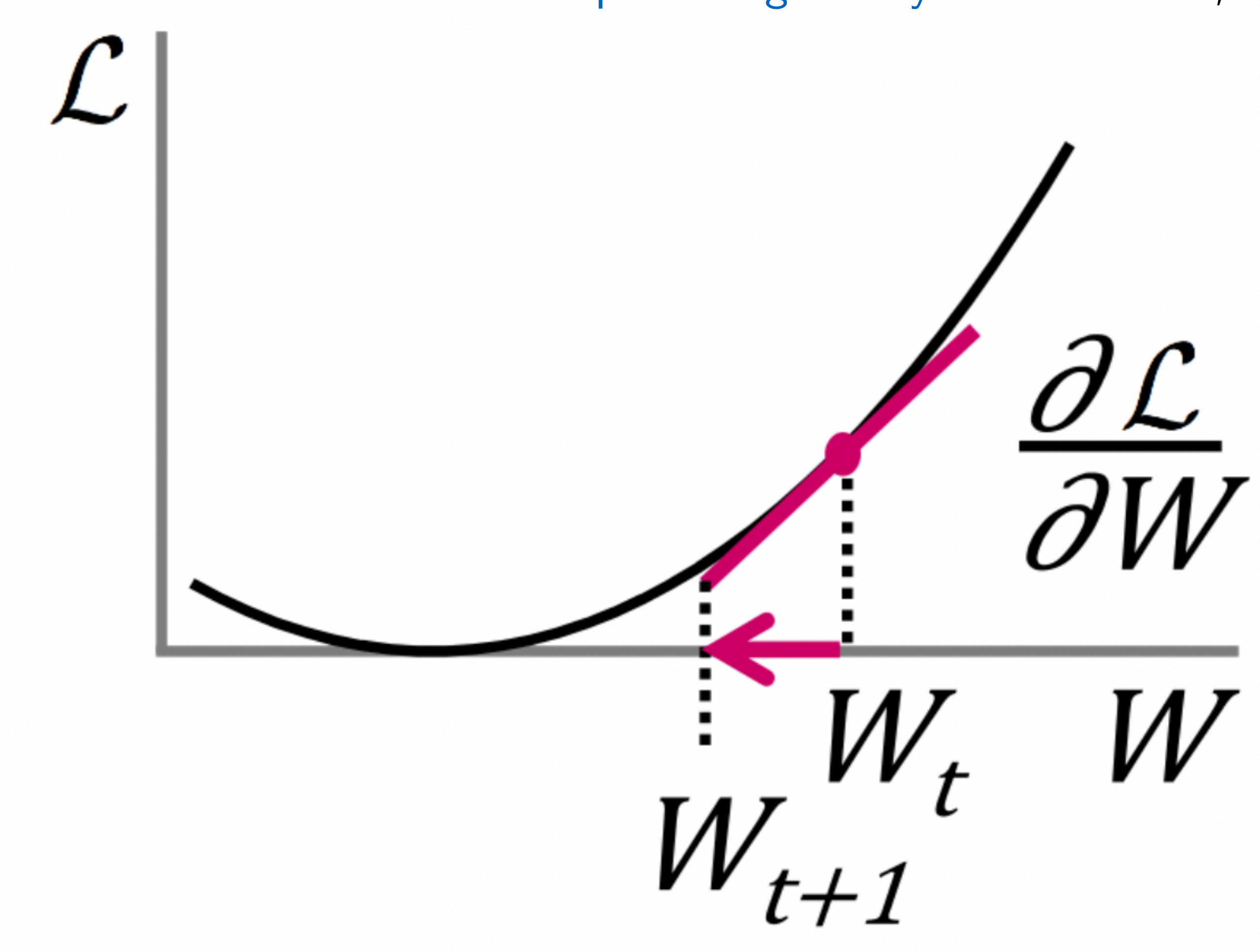
from "Deep learning in Physics Research", Erdmann *et al.*

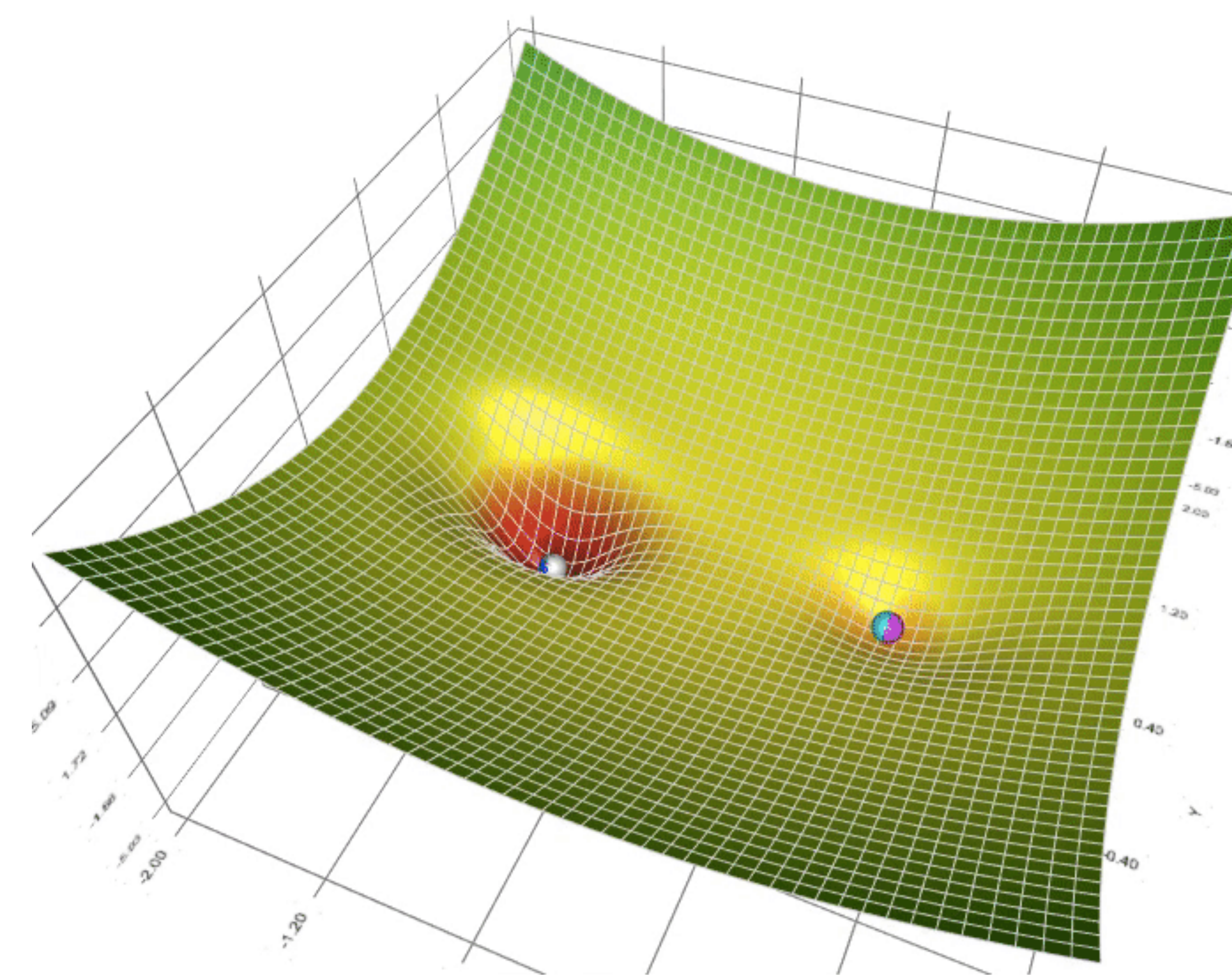




1. **Standard update** of particular weight ω from $t \to t+1$
   - $\omega_{t+1} = \omega_t - \alpha \cdot \frac{\partial L}{\partial \omega_t} \quad \to \quad \Delta\omega_t = -\alpha \cdot \frac{\partial L}{\partial \omega_t}$
2. **Adagrad**: Remember all past gradients and adapt $\alpha \to \alpha_t$
   - $\nu_t = \sum_{\tau=1}^{t} \left(\frac{\partial L}{\partial \omega_\tau} \right)^2$ and $\alpha_t = \frac{\alpha}{\sqrt{\nu_t} + \epsilon}$
3. **RMSprob**: Scale down or "decay" sum of past gradients by β
   - $\nu_t = \beta \cdot \nu_{t-1} + (1-\beta) \cdot \left(\frac{\partial L}{\partial \omega_t} \right)^2$

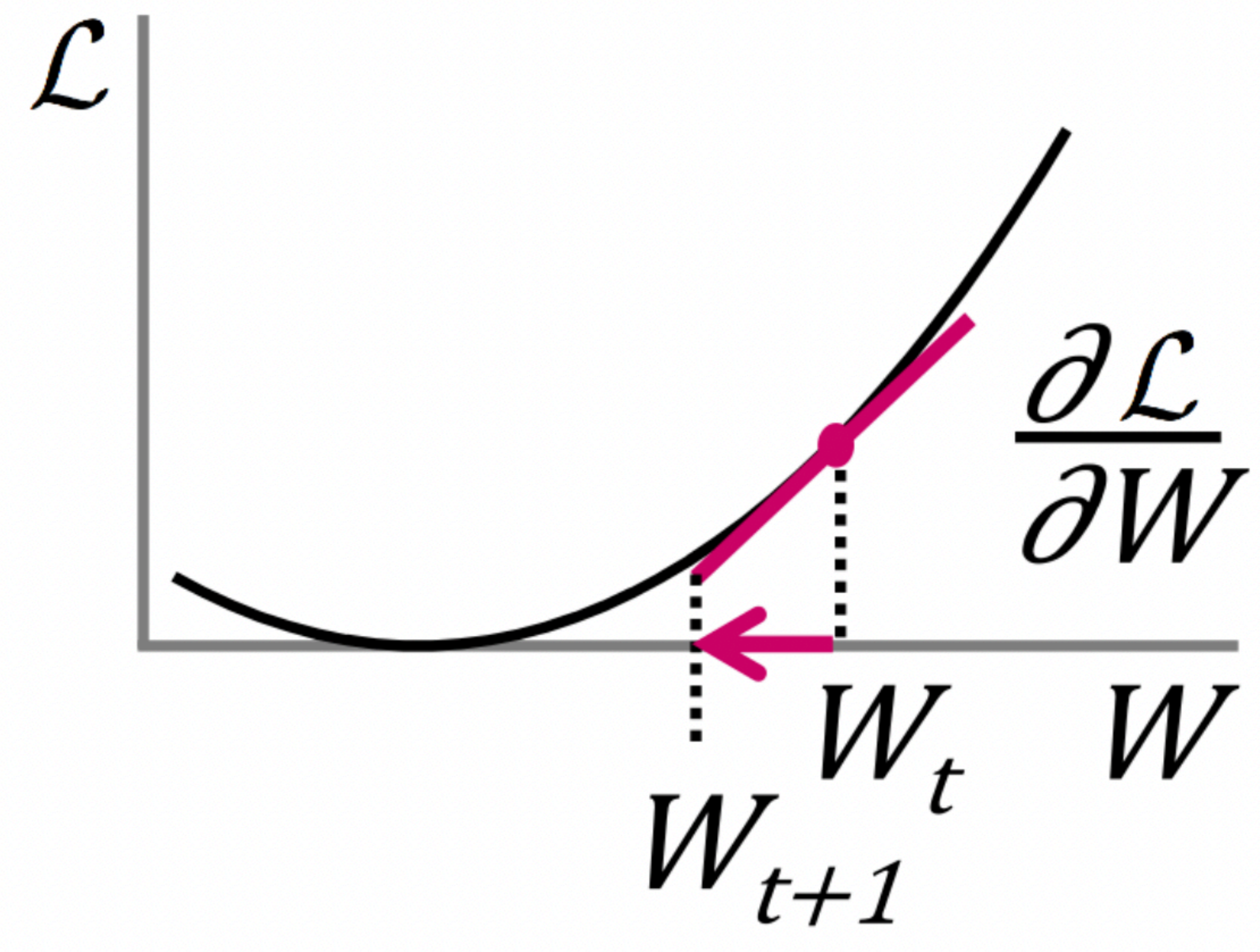
from "Deep learning in Physics Research", Erdmann *et al.*





1. **Standard update** of particular weight ω from $t \to t+1$
   - $\omega_{t+1} = \omega_t - \alpha \cdot \frac{\partial L}{\partial \omega_t} \quad \to \quad \Delta\omega_t = -\alpha \cdot \frac{\partial L}{\partial \omega_t}$

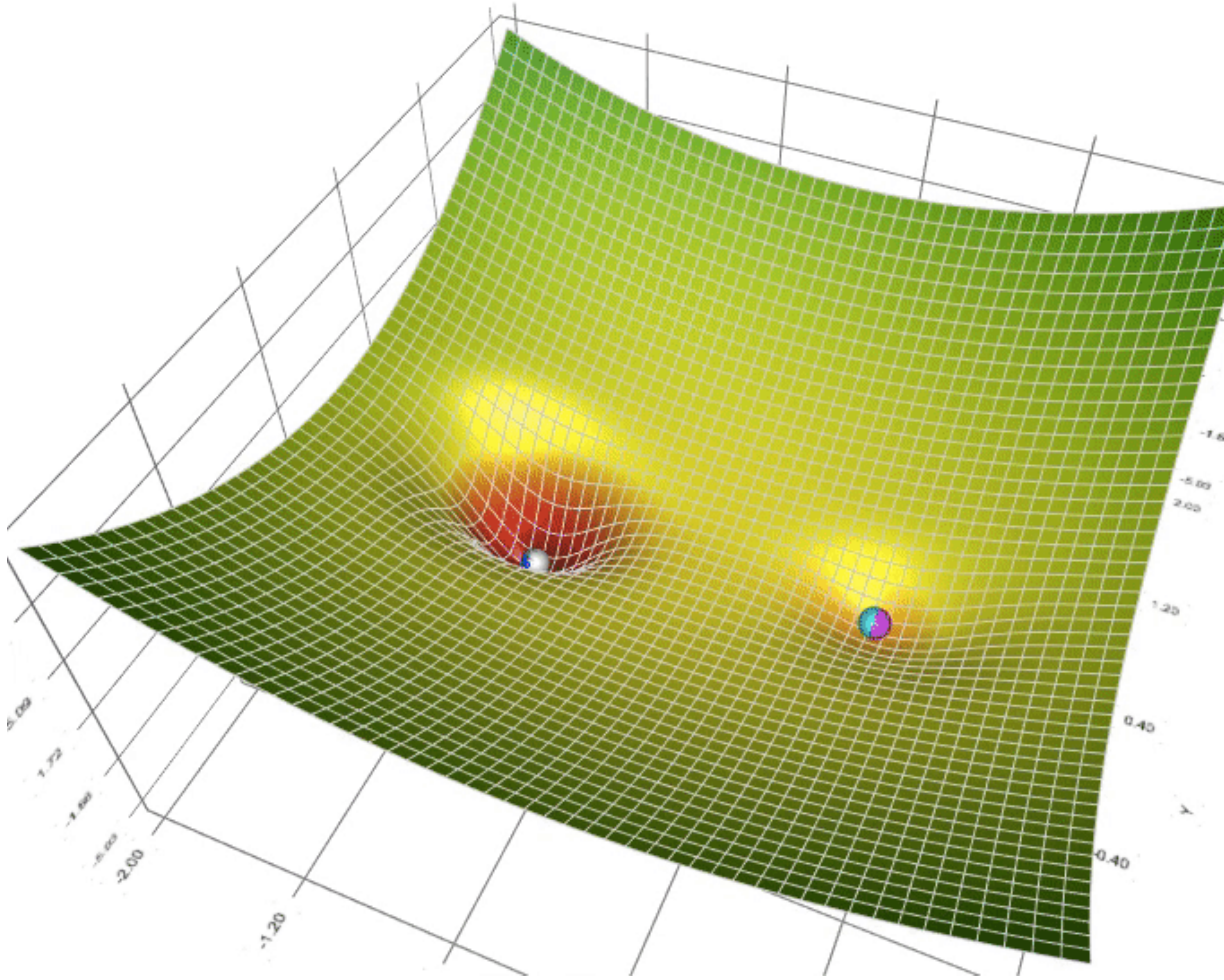
from "Deep learning in Physics Research", Erdmann *et al.*



2. **Adagrad**: Remember all past gradients and adapt $\alpha \to \alpha_t$
   - $\nu_t = \sum_{\tau=1}^{t} \left(\frac{\partial L}{\partial \omega_\tau} \right)^2$ and $\alpha_t = \frac{\alpha}{\sqrt{\nu_t} + \epsilon}$

3. **RMSprob**: Scale down or "decay" sum of past gradients by β
   - $\nu_t = \beta \cdot \nu_{t-1} + (1-\beta) \cdot \left(\frac{\partial L}{\partial \omega_t} \right)^2$

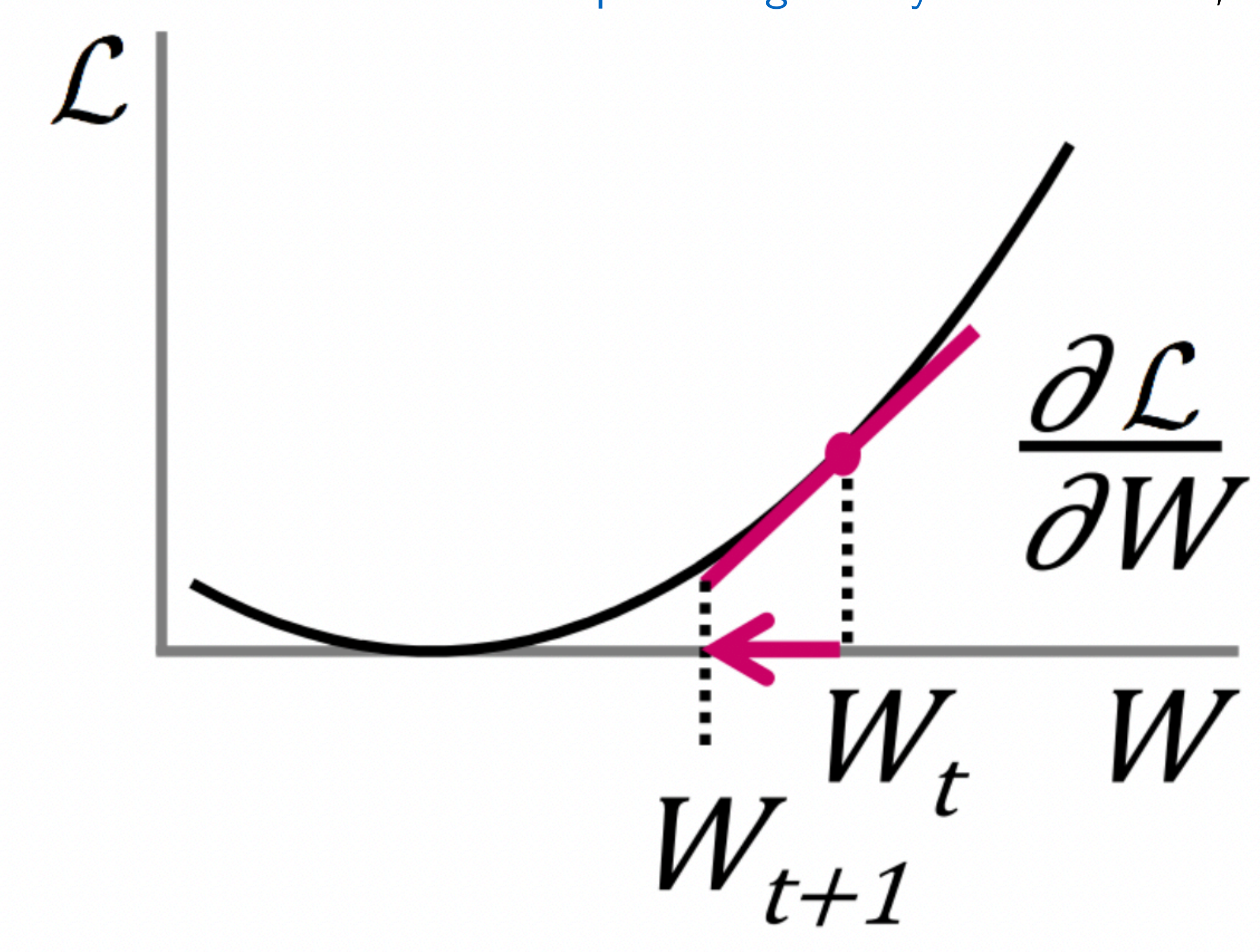
4. **Momentum**: Stabilize the direction, maintaining previous "velocity"
   - $\Delta\omega_t \overset{!}{=} v_t = \beta \cdot v_{t-1} - (1-\beta) \cdot \alpha \cdot \frac{\partial L}{\partial \omega_t}$

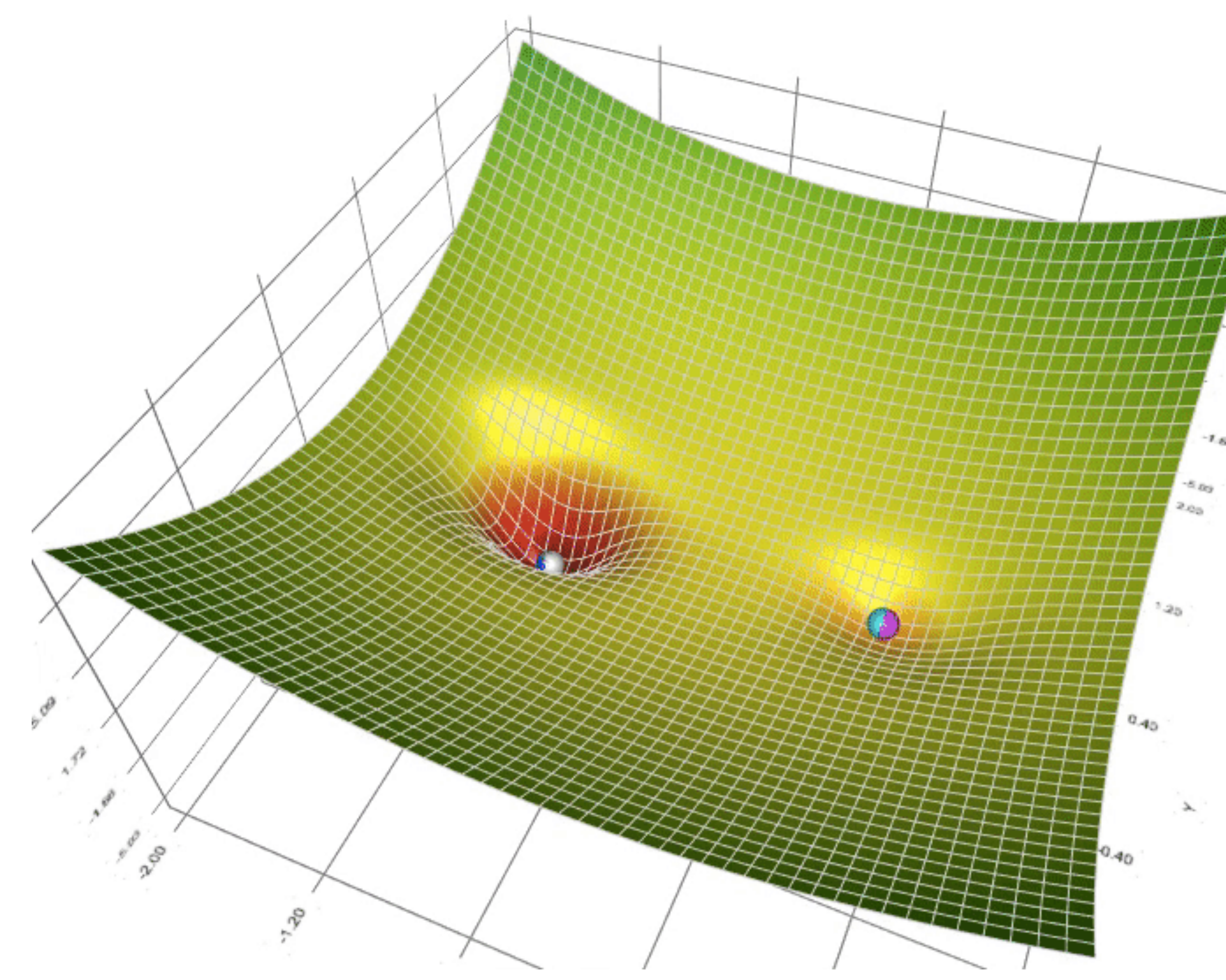


1. **Standard update** of particular weight ω from $t \rightarrow t+1$
   - $$\omega_{t+1} = \omega_t - \alpha \cdot \frac{\partial L}{\partial \omega_t} \quad \rightarrow \quad \Delta\omega_t = -\alpha \cdot \frac{\partial L}{\partial \omega_t}$$
2. **Adagrad**: Remember all past gradients and adapt $\alpha \rightarrow \alpha_t$
   - $$\nu_t = \sum_{\tau=1}^{t} \left(\frac{\partial L}{\partial \omega_\tau}\right)^2 \quad \text{and} \quad \alpha_t = \frac{\alpha}{\sqrt{\nu_t} + \epsilon}$$
3. **RMSprob**: Scale down or "decay" sum of past gradients by β
   - $$\nu_t = \beta \cdot \nu_{t-1} + (1-\beta) \cdot \left(\frac{\partial L}{\partial \omega_t}\right)^2$$
4. **Momentum**: Stabilize the direction, maintaining previous "velocity"
   - $$\Delta\omega_t \overset{!}{=} v_t = \beta \cdot v_{t-1} - (1-\beta) \cdot \alpha \cdot \frac{\partial L}{\partial \omega_t}$$
5. **Adam**: Combine $(\nabla L)^2$ of RMSprob and ∇L of momentum
   - $$\Delta\omega_t \overset{!}{=} v_t = -\alpha \frac{m_t}{\sqrt{\nu_t} + \epsilon} \quad \text{with}$$

$$m_t = \frac{1}{1-\gamma^t}\left[\gamma \cdot m_{t-1} + (1-\gamma) \cdot \frac{\partial L}{\partial \omega_t}\right] \qquad \nu_t = \frac{1}{1-\beta^t}\left[\beta \cdot \nu_{t-1} + (1-\beta) \cdot \left(\frac{\partial L}{\partial \omega_t}\right)^2\right]$$

from "Deep learning in Physics Research", Erdmann *et al.*



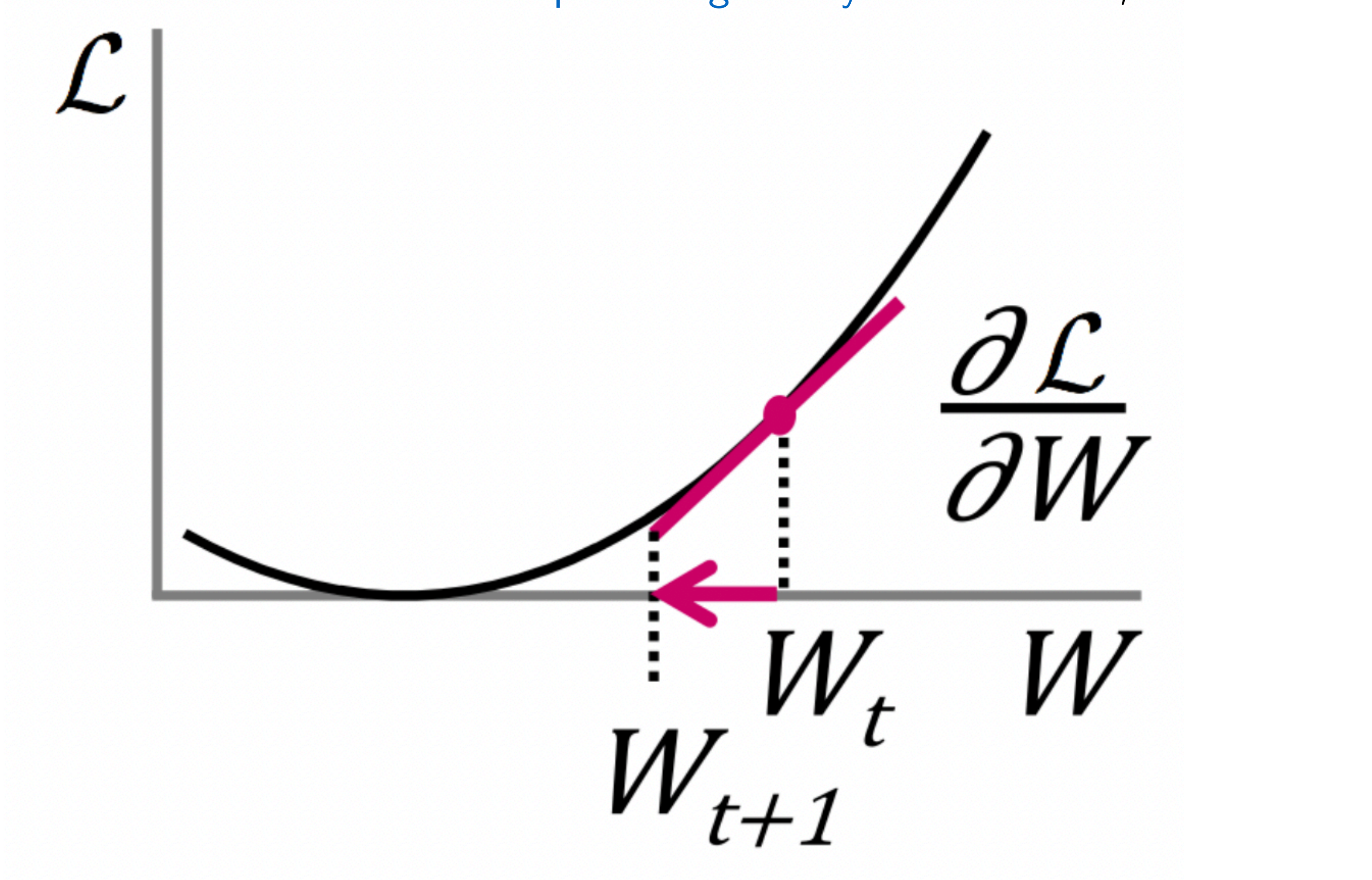


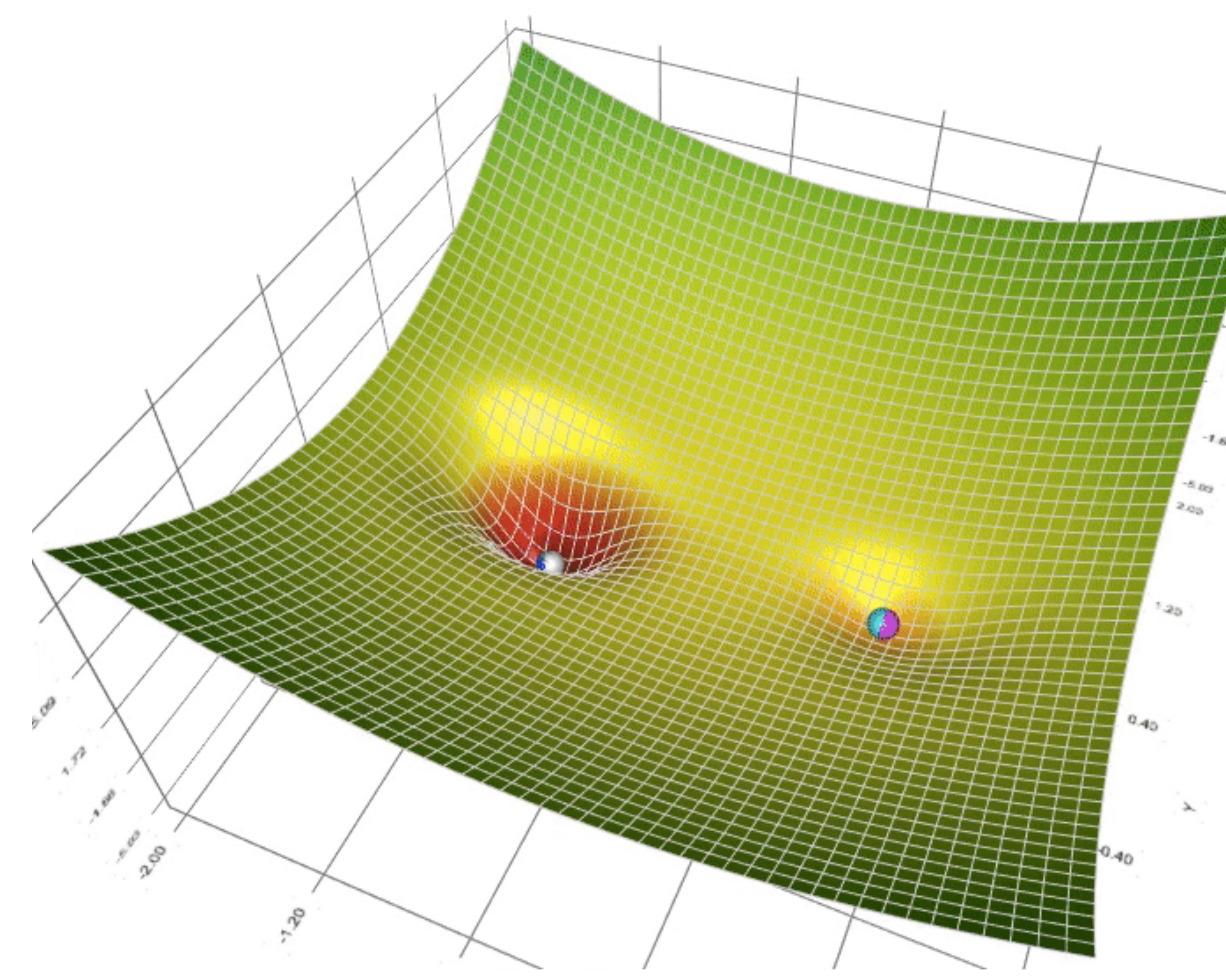
Adaptive & momentum learning in 5 steps

1. **Standard update** of particular weight ω from $t \to t+1$
   - $\omega_{t+1} = \omega_t - \alpha \cdot \dfrac{\partial L}{\partial \omega_t} \quad \to \quad \Delta\omega_t = -\alpha \cdot \dfrac{\partial L}{\partial \omega_t}$
2. **Adagrad**: Remember all past gradients and adapt $\alpha \to \alpha_t$
   - $\nu_t = \sum_{\tau=1}^{t} \left(\dfrac{\partial L}{\partial \omega_\tau}\right)^2$ and $\alpha_t = \dfrac{\alpha}{\sqrt{\nu_t} + \epsilon}$
3. **RMSprob**: Scale down or "decay" sum of past gradients by β
   - $\nu_t = \beta \cdot \nu_{t-1} + (1-\beta) \cdot \left(\dfrac{\partial L}{\partial \omega_t}\right)^2$
4. **Momentum**: Stabilize the direction, maintaining previous "velocity"
   - $\Delta\omega_t \overset{!}{=} v_t = \beta \cdot v_{t-1} - (1-\beta) \cdot \alpha \cdot \dfrac{\partial L}{\partial \omega_t}$
5. **Adam**: Combine $(\nabla L)^2$ of RMSprob and ∇L of momentum
   - $\Delta\omega_t \overset{!}{=} v_t = -\alpha \dfrac{m_t}{\sqrt{\nu_t} + \epsilon}$ with

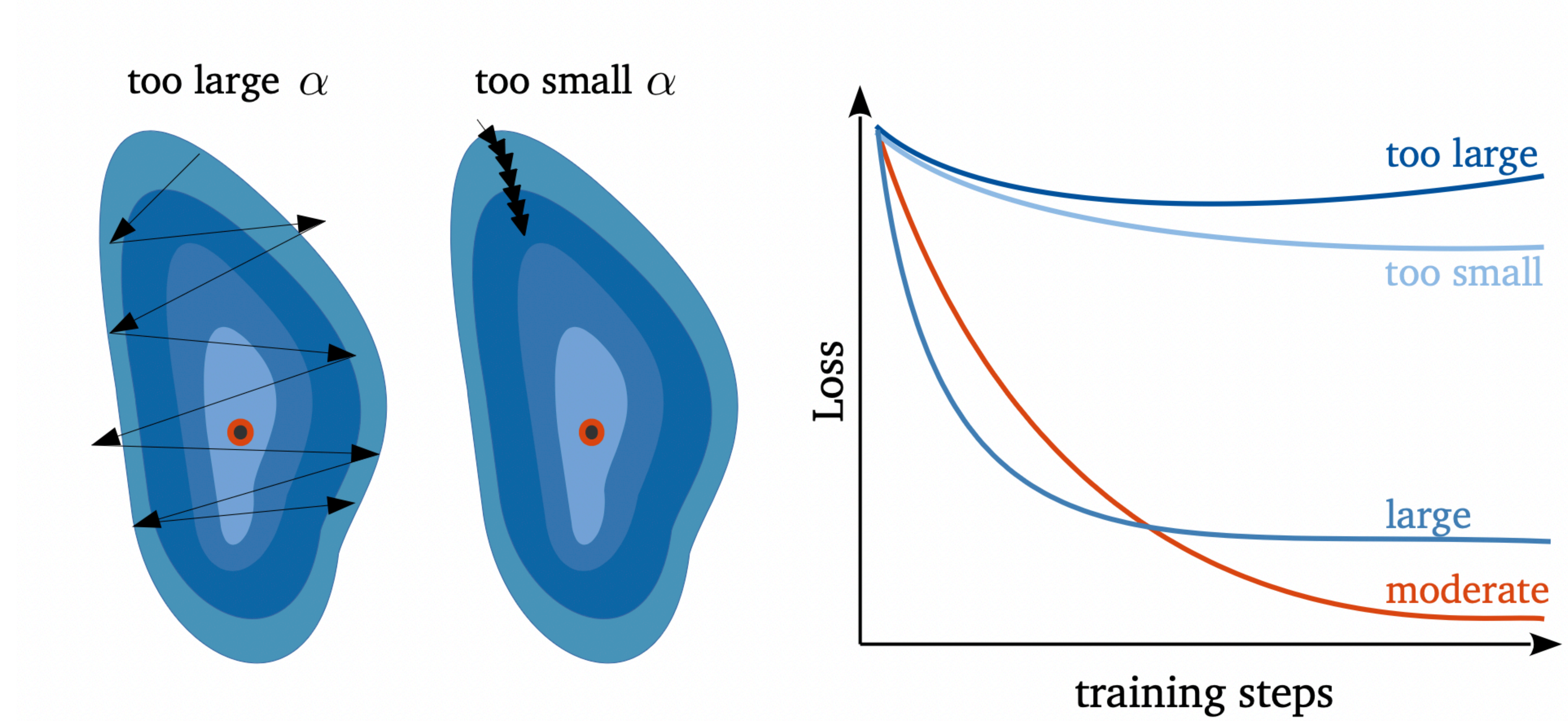
$$m_t = \frac{1}{1-\gamma^t}\left[\gamma \cdot m_{t-1} + (1-\gamma) \cdot \frac{\partial L}{\partial \omega_t}\right] \qquad \nu_t = \frac{1}{1-\beta^t}\left[\beta \cdot \nu_{t-1} + (1-\beta) \cdot \left(\frac{\partial L}{\partial \omega_t}\right)^2\right]$$

from "Deep learning in Physics Research", Erdmann *et al.*



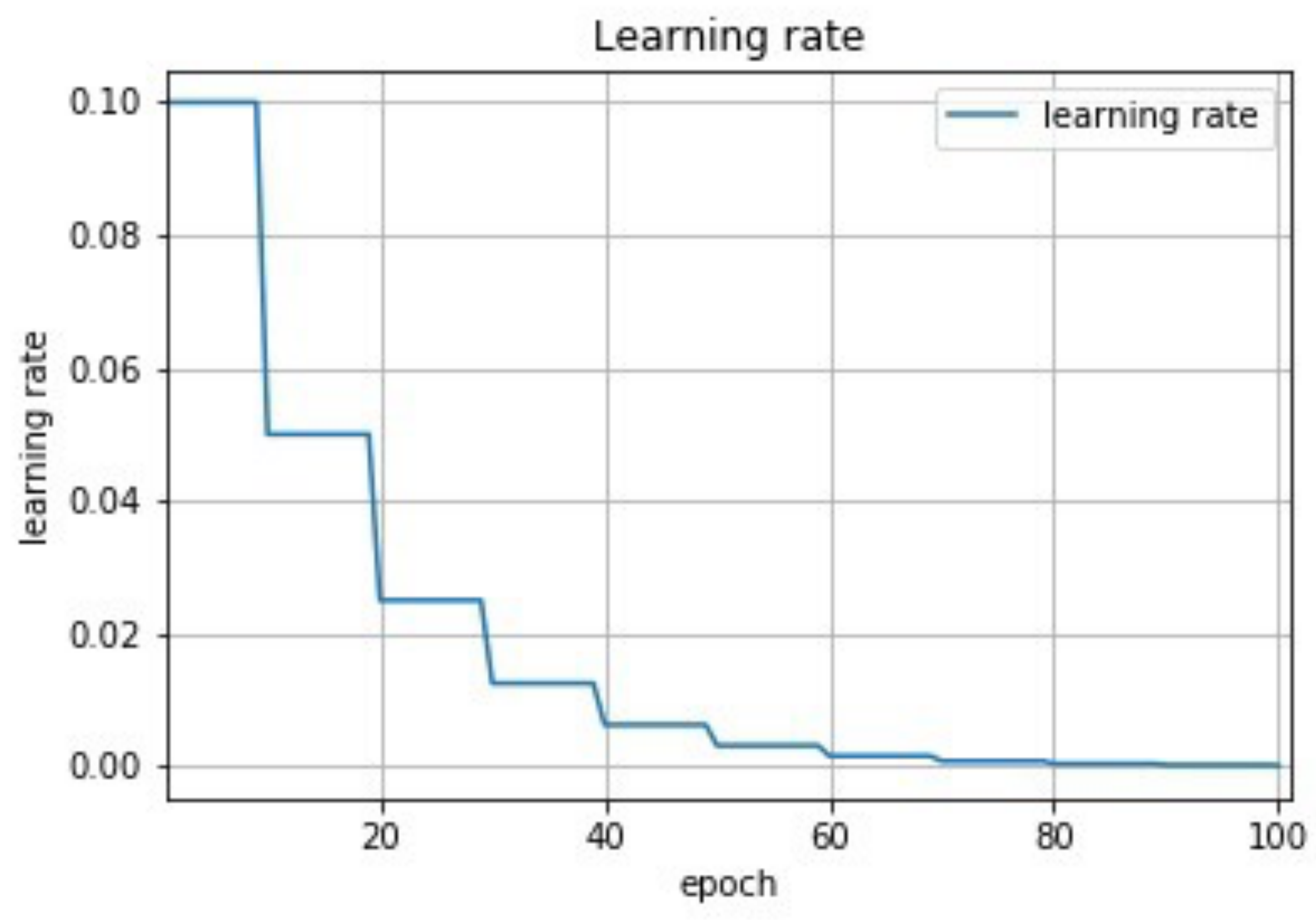


- Even with Adam, the final optimization steps can circle around the true minimum ("overshooting")
  → Detectable via noise after seemingly converging



from "Deep learning in Physics Research", Erdmann *et al.*

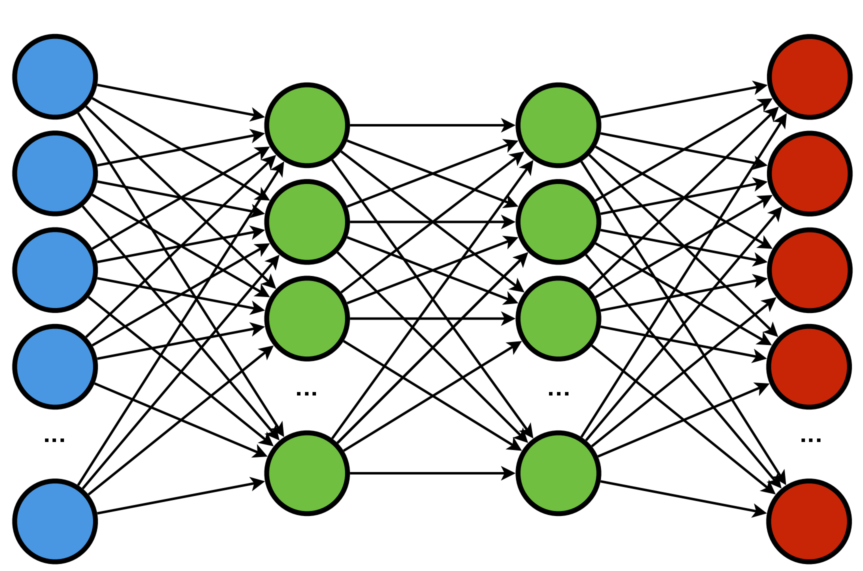
- Adjust the base learning rate α
  - **Over time**
    - ▻ Step wise according to some schedule
    - ▻ Exponential decay (e.g. multiply by $\rho = 0.9$every n steps)
  - **After reaching plateau**
    - ▻ Detect by keeping history of last n losses
    - ▻ Multiply α by $\rho = 0.5$, repeat k times



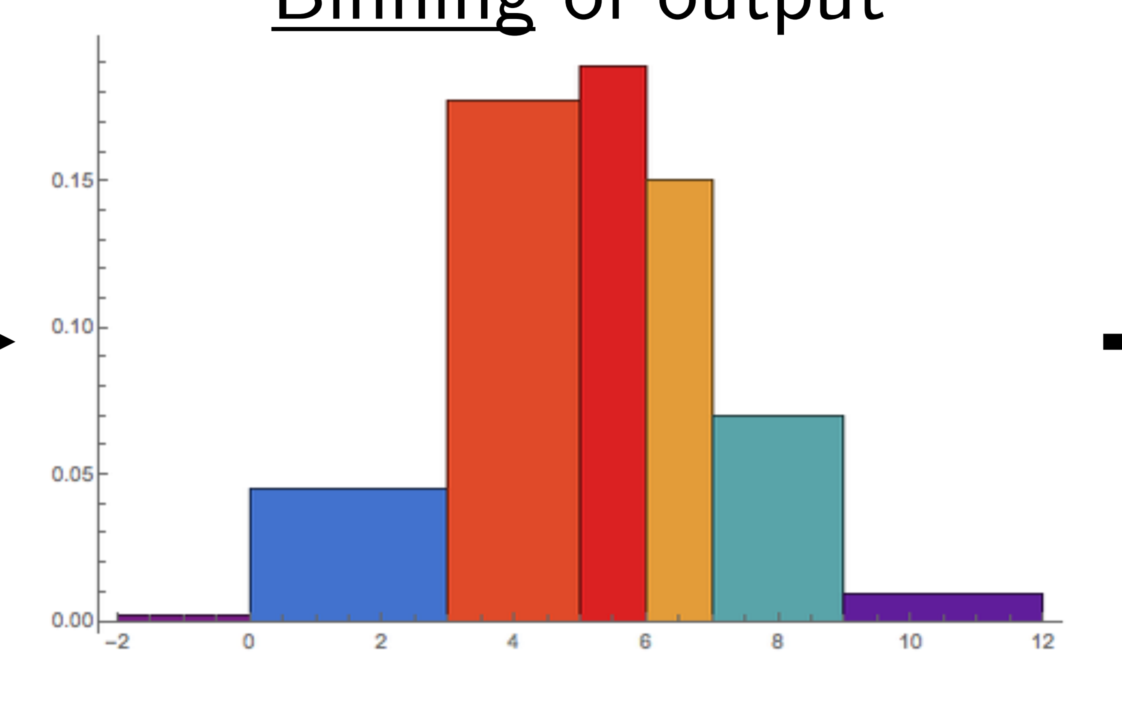
- **Which type of result do intend to extract and publish?**
  - Some central measurement of observable plus uncertainties
  - Theory / model exclusion intervals (CL)
  - Significance of measurement over some background-only hypothesis
  - ...

- **... did you tell your network?**
  - *Or*: Is your loss **±100% correlated** to your result quantity?
  - Often **not the case**
    - ▻ Neural network metrics are just **proxies**, entangled with assumptions!
    - ▻ Make sure it's a good one, *or* define an optimization process (brute-force scans)
    - ▻ Especially true if complex measurement techniques are employed after your ML algo.
    - ▻ **Example**

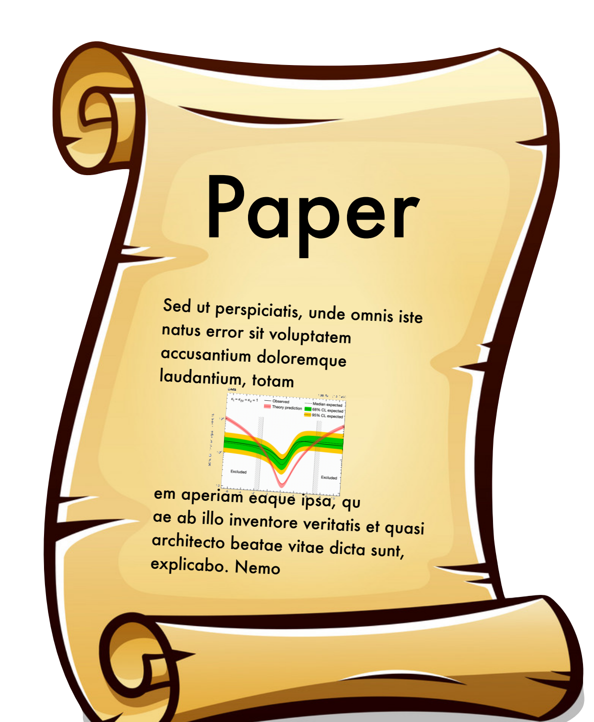
Network



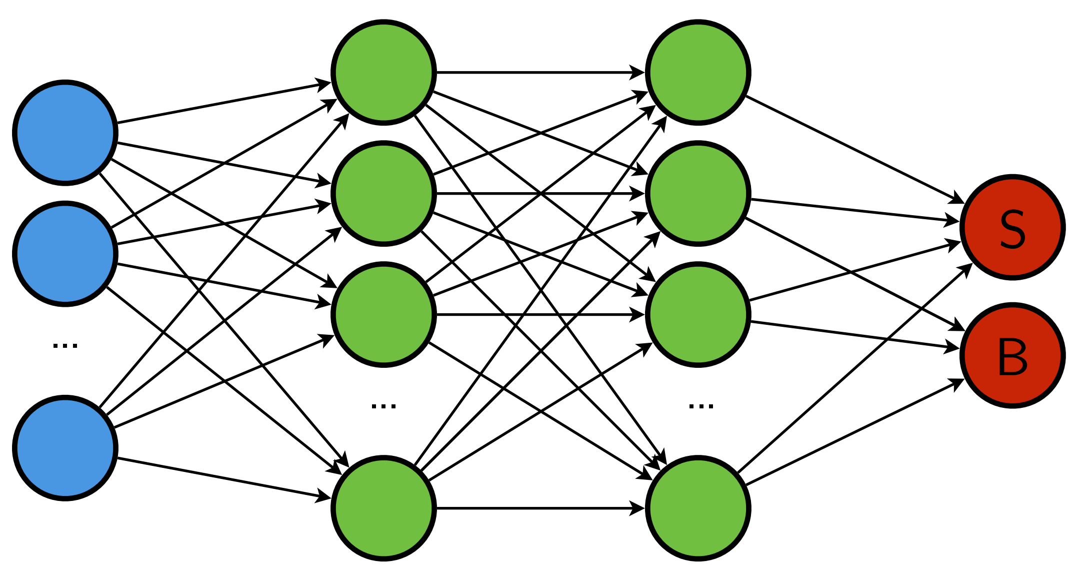
Binning of output

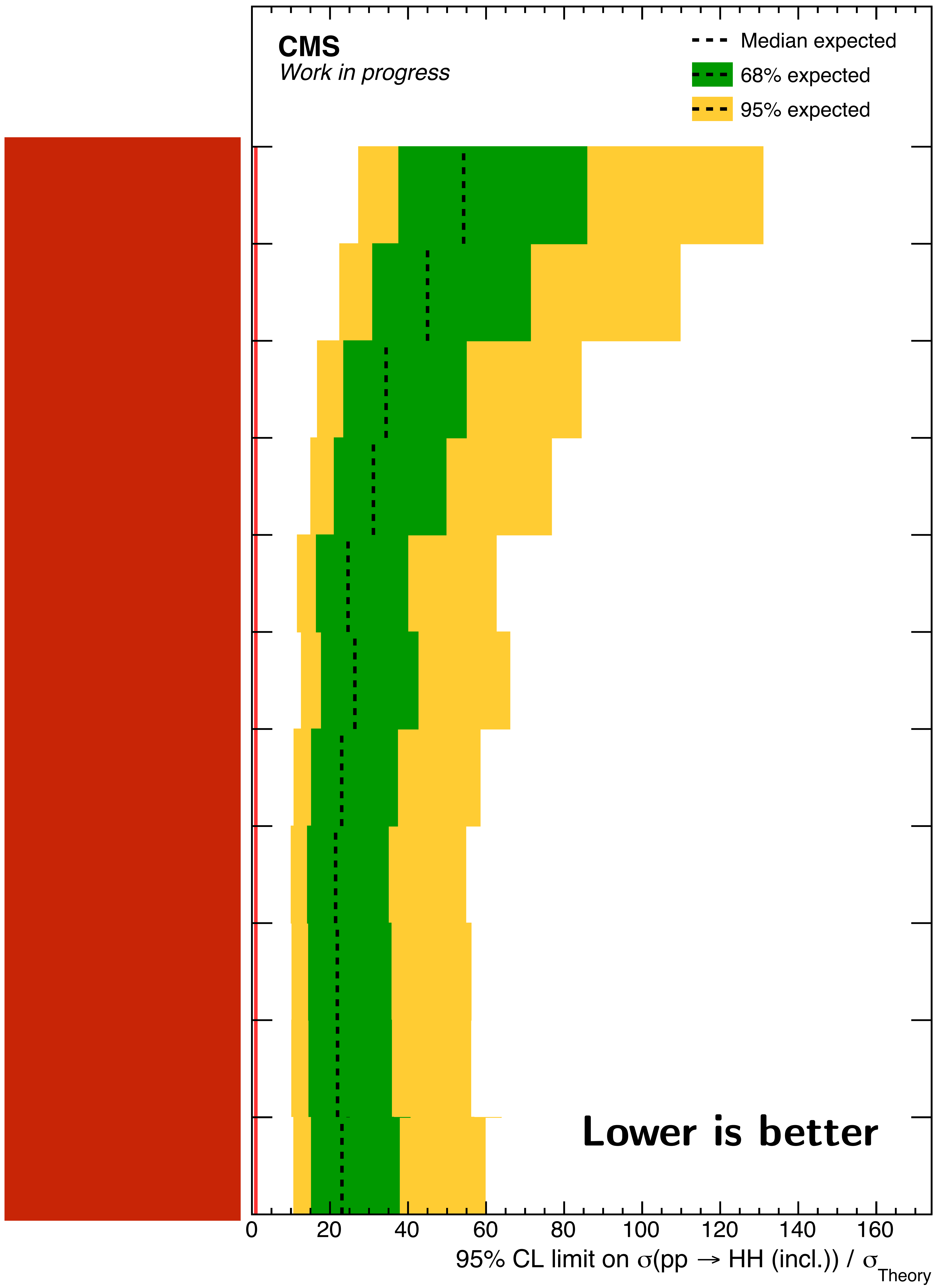


Results: Limits

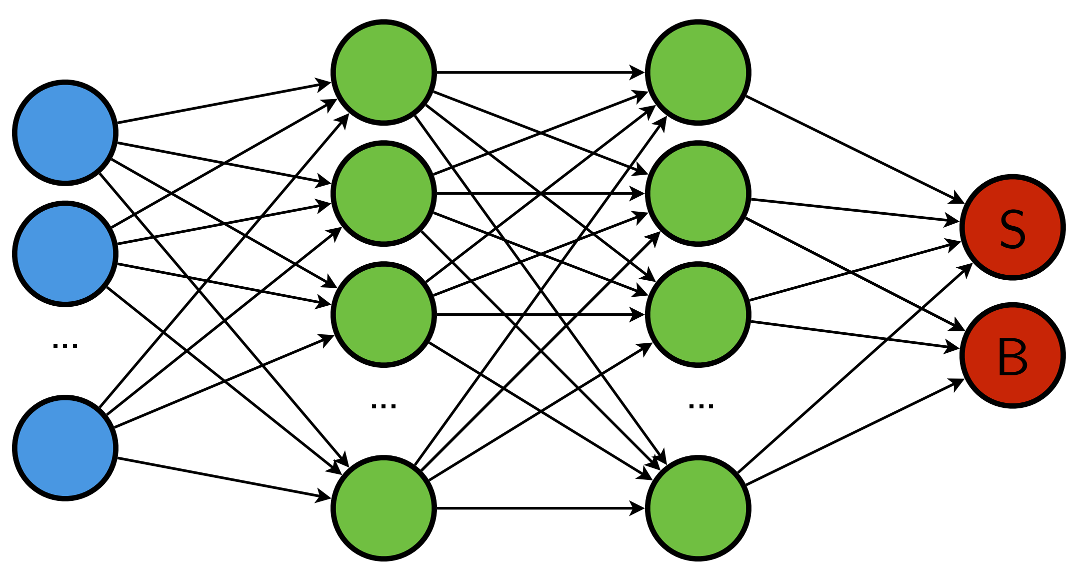


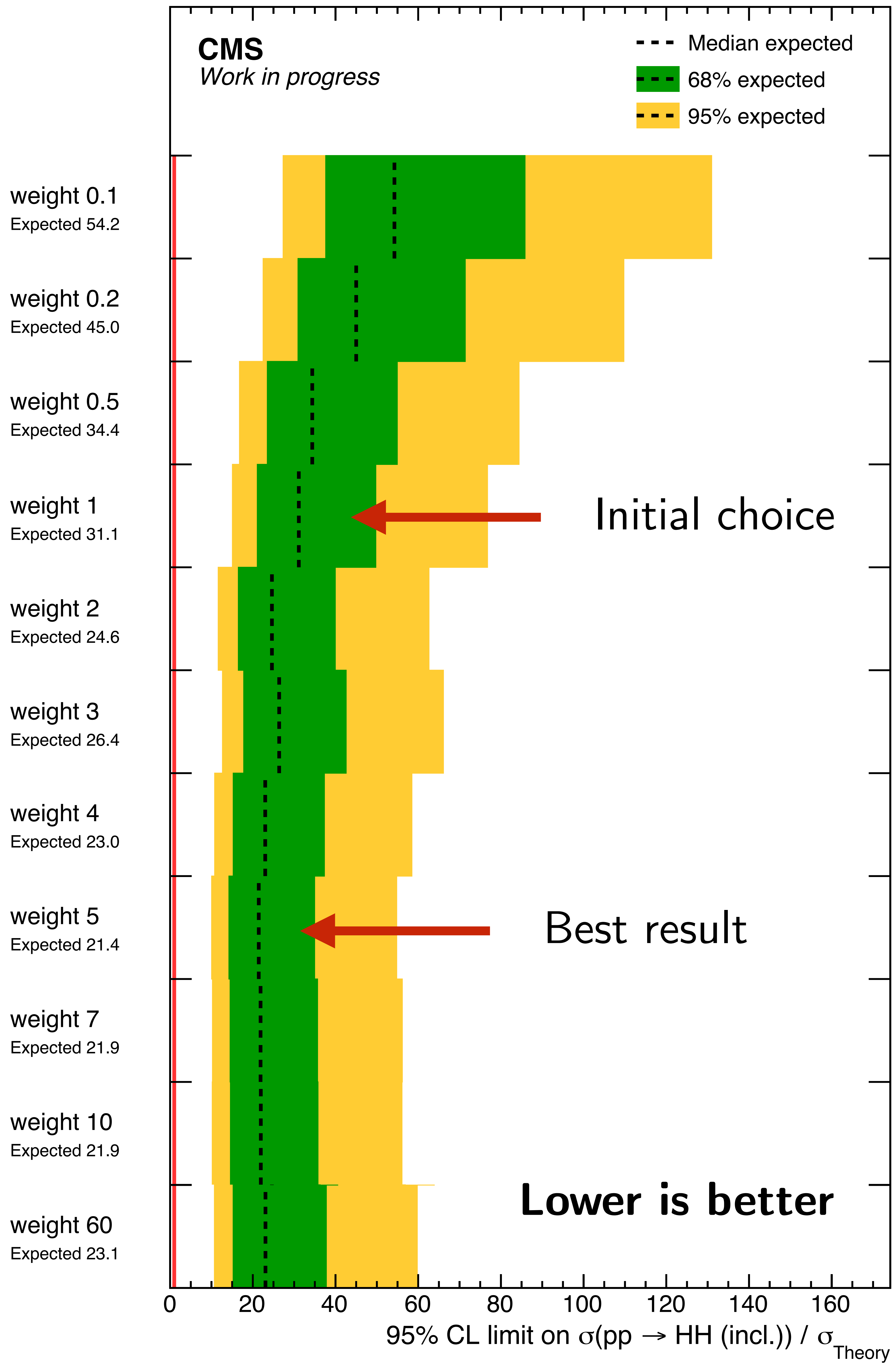
- Real-life example: Search for HH production
  - ML application: **S**ignal vs. **B**ackground separation
  - **Loss**
    - ▻ Cross entropy with **equal weight / importance** for all **S** and **B** events
    - ▻ Assumption!
  - **Test**
    - ▻ Vary the relative weight of **B** to **S**
    - ▻ Compute upper exclusion limit (brute force)





- Real-life example: Search for HH production
  - ML application: **S**ignal vs. **B**ackground separation
  - **Loss**
    - ▻ Cross entropy with **equal weight / importance** for all **S** and **B** events
    - ▻ Assumption!
  - **Test**
    - ▻ Vary the relative weight of **B** to **S**
    - ▻ Compute upper exclusion limit (brute force)





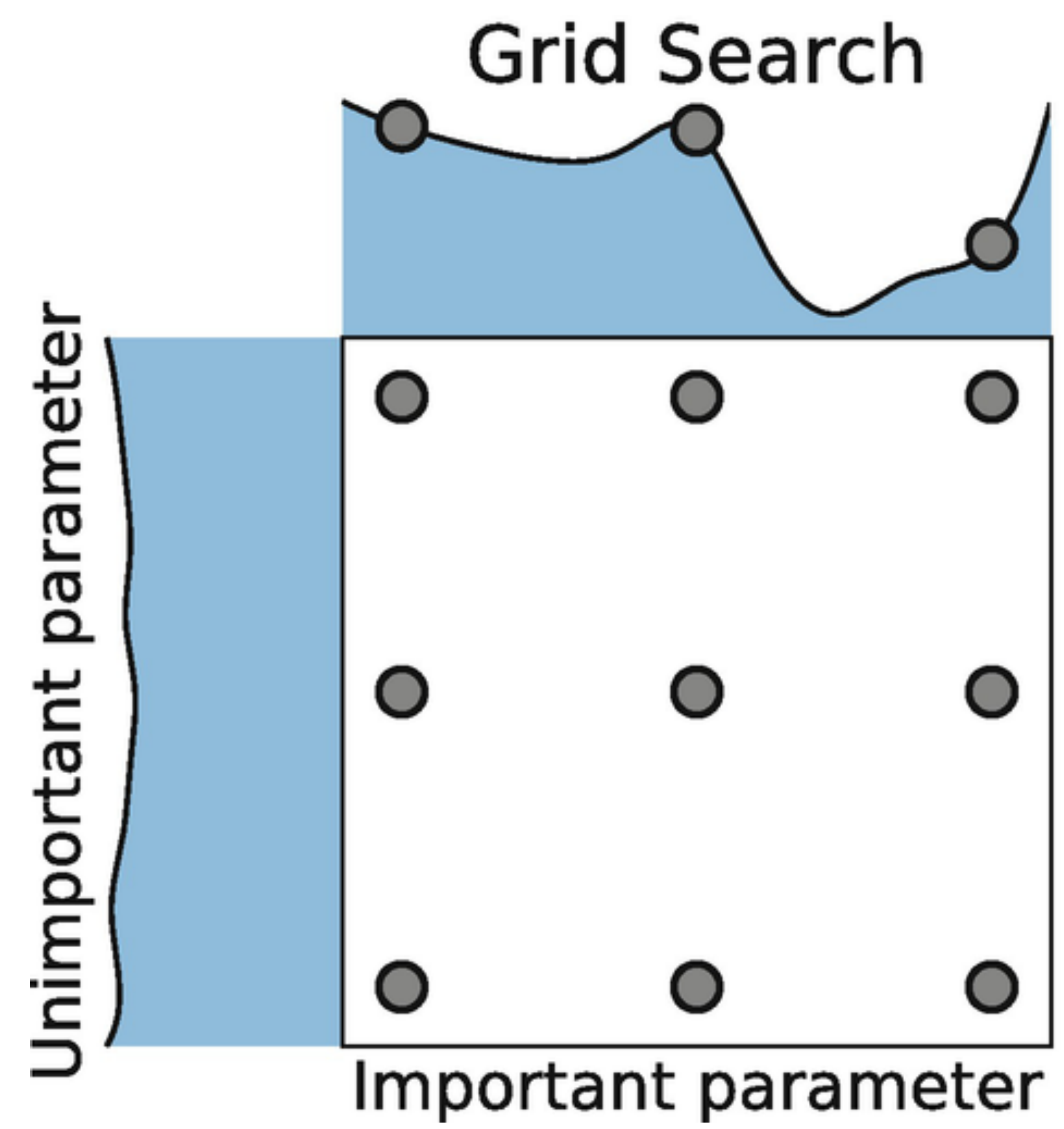
- Large list of hyper-parameters to optimize (here, for FCNs)
  - **Architecture**
    - Number of layers, n_l
    - Number of units per layer, n_u
    - Activation, $\sigma(x)$
    - Discrete choices
      - Dense/residual/classic
      - Weight initialization
      - Batch normalization
  - **Optimizer**
    - Learning rate + decay, α, ρ
    - Algorithm and its parameters (Adam: β, γ)
  - **Training**
    - Batch size, b
    - Splitting fractions, f_{train}, f_{valid}
    - Cross validation folds, k
    - Regularization factor, λ
    - Dropout rate, p

- Large list of hyper-parameters to optimize (here, for FCNs)
  - **Architecture**
    - Number of layers, n_l
    - Number of units per layer, n_u
    - Activation, $\sigma(x)$
    - Discrete choices
      - Dense/residual/classic
      - Weight initialization
      - Batch normalization
  - **Optimizer**
    - Learning rate + decay, α, ρ
    - Algorithm and its parameters (Adam: β, γ)
  - **Training**
    - Batch size, b
    - Splitting fractions, f_{train}, f_{valid}
    - Cross validation folds, k
    - Regularization factor, λ
    - Dropout rate, p

How to approach them?

1. Some have well-tested defaults
   - E.g. Adam parameters
2. Some can be grouped (divide & conquer)
   - E.g. batch size & learning rate
3. Some do not depend in each other
   - Optimize one after another
4. Hyper-parameter search for the remainder
   - "hyper-opt"

- Assume you have $n_p = 6$ hyper-parameters with $n_c = 5$ possible settings each
  - $n_c^{n_p} = 15625$ trainings
  - Even more when considering ensemble learning, k-fold cross validation, ...



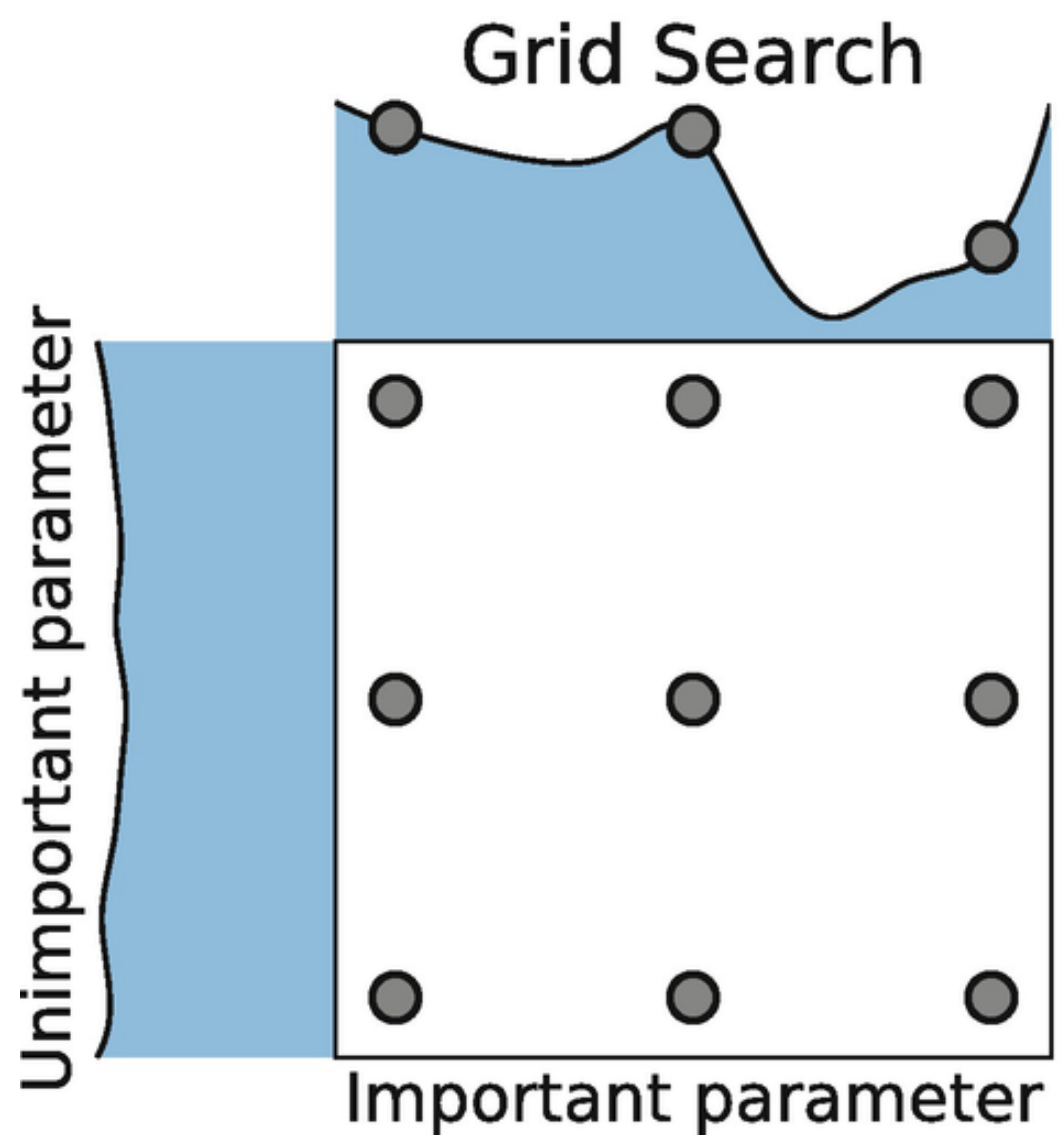
- **Grid search**
  - Iterate brute-force through the grid of points

  → Highly-resource intensive

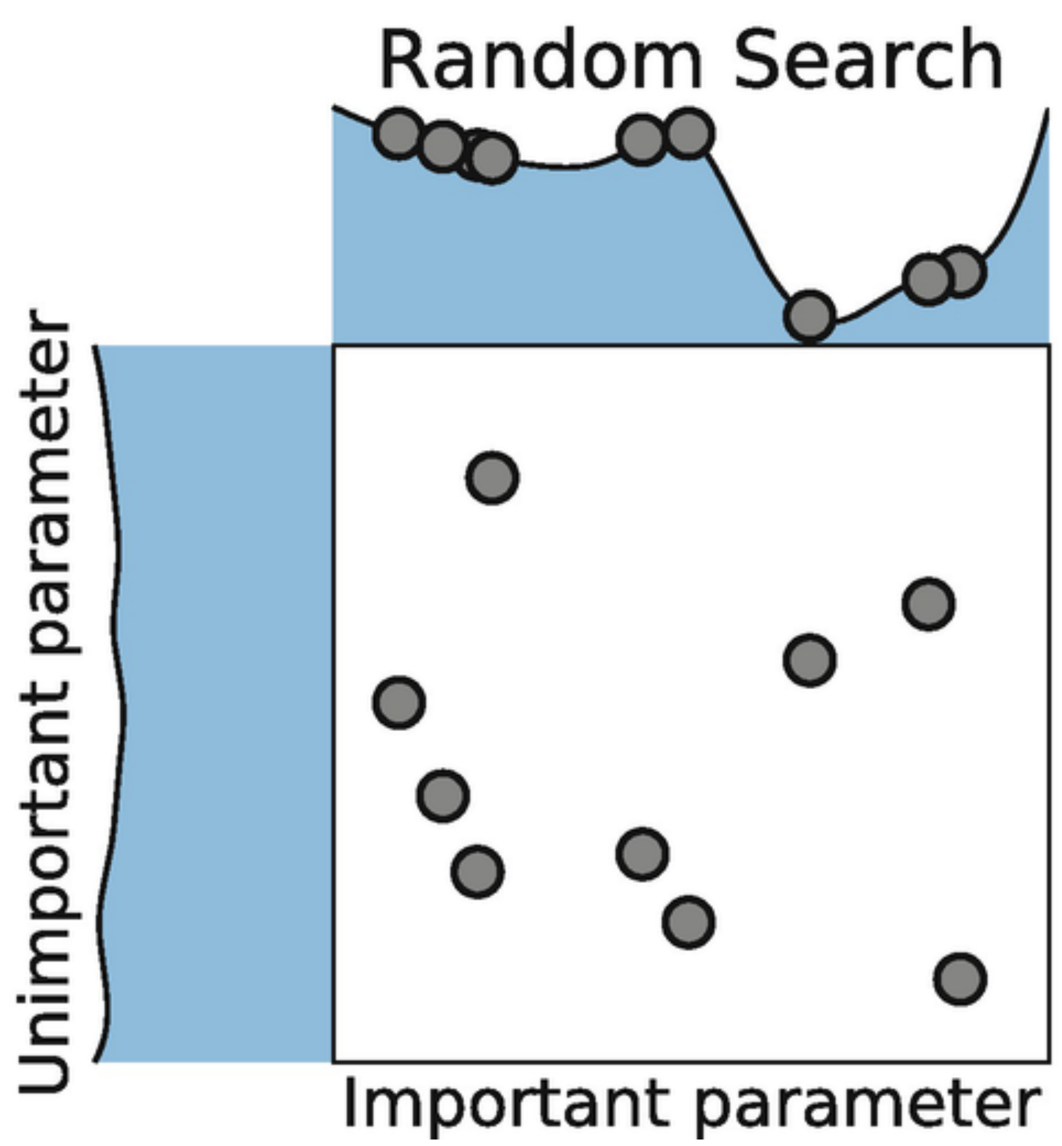
  → Risk of wasting resources on unimportant parameter choices

  → Chance of hitting the best parameters limited by grid granularity

- Assume you have $n_p = 6$ hyper-parameters with $n_c = 5$ possible settings each
  - $n_c^{n_p} = 15625$ trainings
  - Even more when considering ensemble learning, k-fold cross validation, ...



- **Grid search**
  - Iterate brute-force through the grid of points
  - → Highly-resource intensive
  - → Risk of wasting resources on unimportant parameter choices
  - → Chance of hitting the best parameters limited by grid granularity



- **Random search**
  - Define continuous ranges rather than fixed grid (if possible)
  - Sample r points and randomly search for best performance
  - → Less resources as r is usually $\ll n_c^{n_p}$
  - → Given that r is still sufficiently large, better chance of finding good points

→ Can we define a somewhat "informed" search?

- Many packages available, scikit-optimize (skopt), hyperopt
- From the skopt documentation:

Problem statement

We are interested in solving

loss **after training** with choice of x

hyper-parameters x

$$x^* = arg \min_x f(x)$$

optimal hyper-parameters x

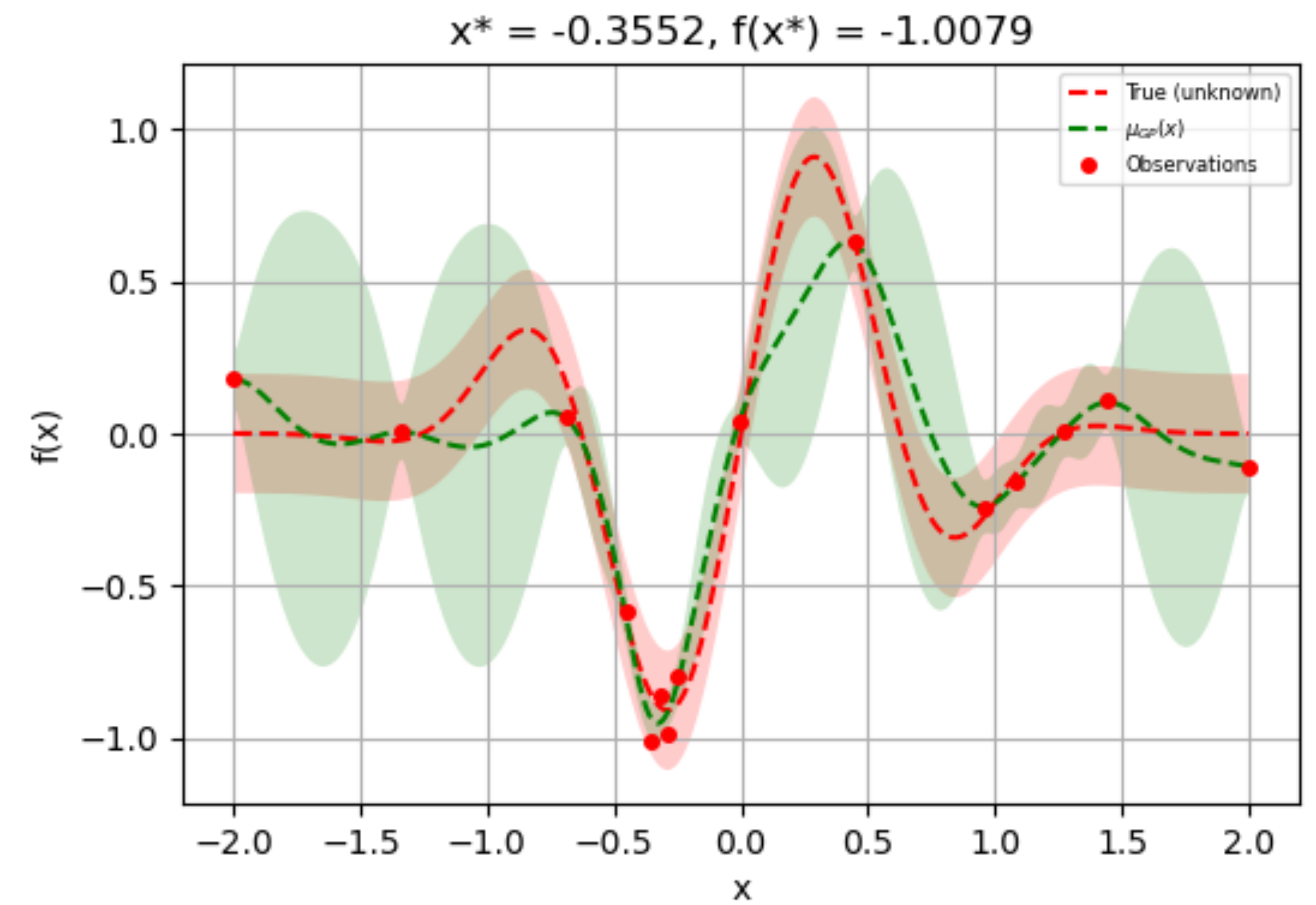
under the constraints that

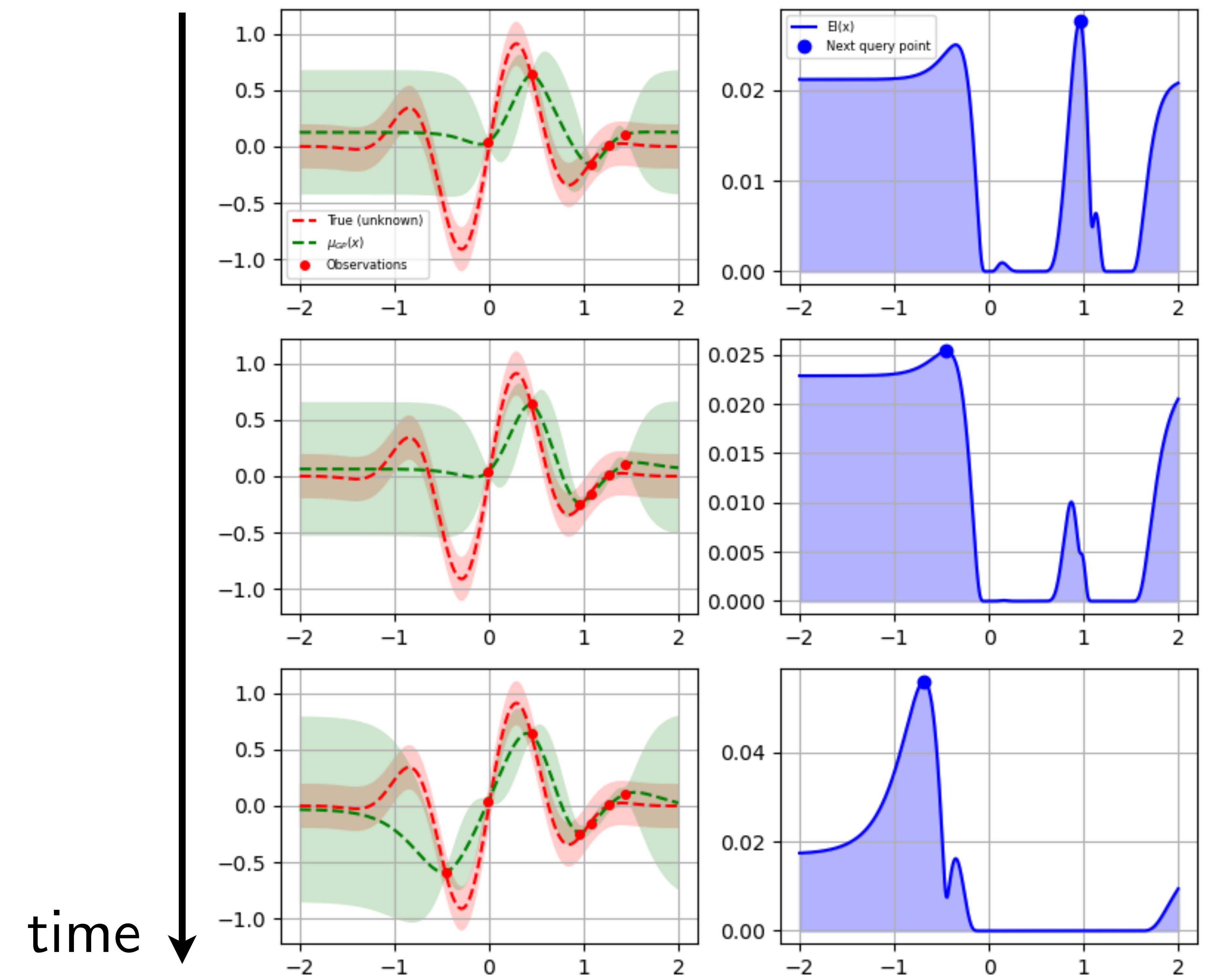
- f is a black box for which no closed form is known (nor its gradients); → yes
- f is expensive to evaluate; → yes
- and evaluations of $y = f(x)$ may be noisy. → yes

Disclaimer. If you do not have these constraints, then there is certainly a better optimization algorithm than Bayesian optimization.

- **Driven by Bayesian optimization**
  - Fit of an arbitrary, potentially non-differentiable function
  - Iterative procedure
    - Few starting points (observations) are required
    - Bayesian process predicts most likely function approximation **plus an uncertainty**
    - Suggest next point of observation at parameter x with highest uncertainty
    - Cycle
    - After sufficient sampling, stop and identify x minimizing f

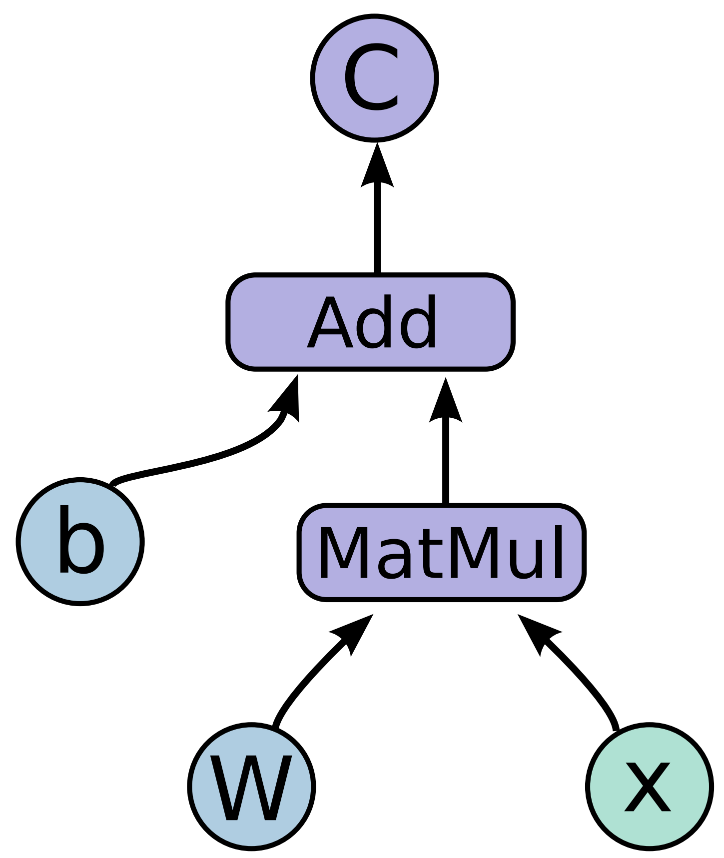
- **Practical workflow**
  - Perform an initial coarse grid or random search
  - Feed pairs of {loss, hyper-parameters} == ($f(x)$, x) into Bayesian optimizer
  - Get back predictions of best next hyper-parameters to test and perform new training
  - Cycle





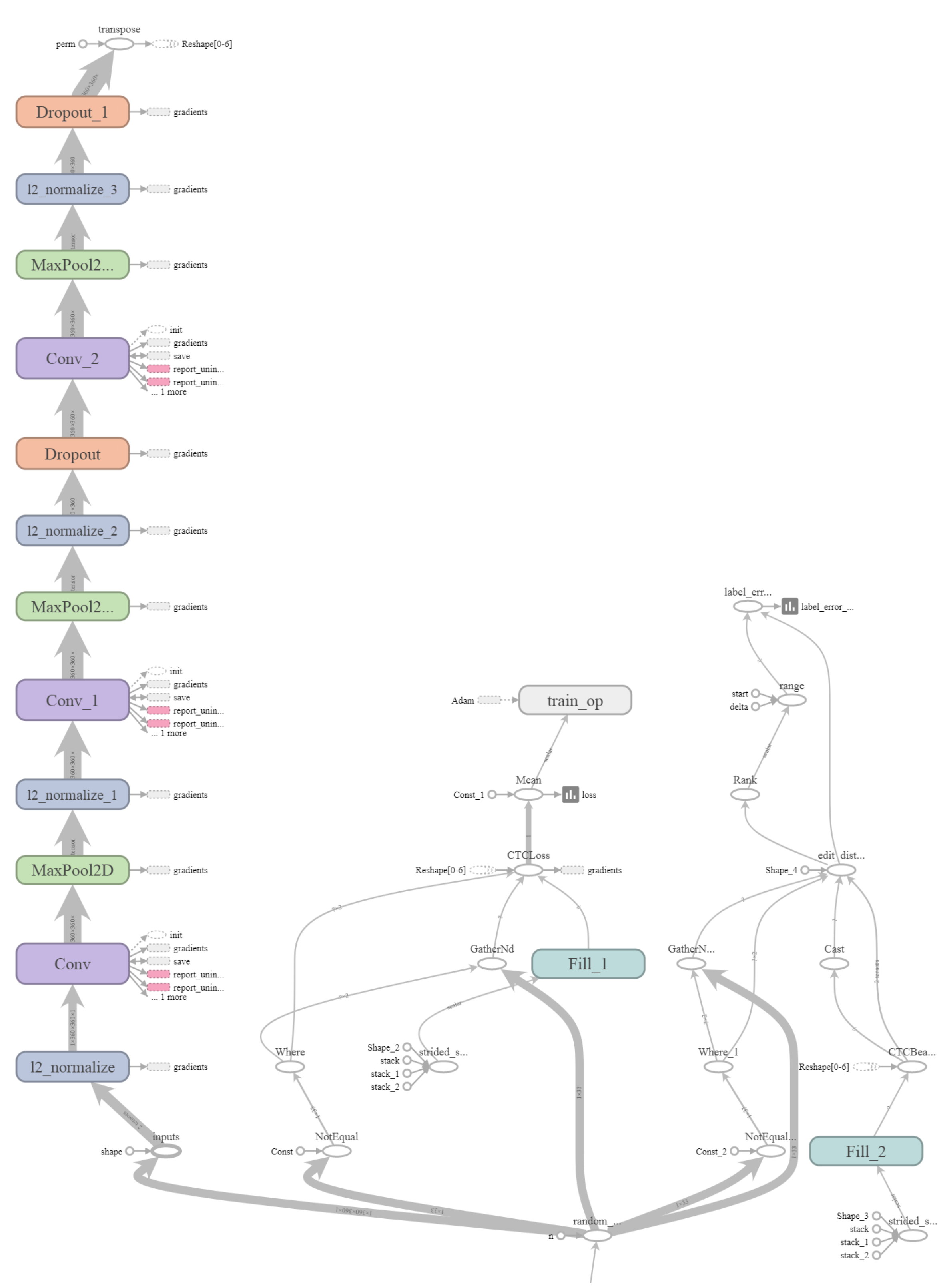
6. Techniques 2/2 & hands-on

- Consider the computation $c = W \cdot x + b$
  - Can be visualized in graph form
  - W, x and b are **inputs** to the graph, c is the **output**
  - Inputs, outputs and intermediate results $(W \cdot x)$ are denoted by **edges**
  - **Operations** ($+$ "*Add*", $\times$ "*MatMul*") are box-shaped **nodes**



- **Every neural network can be described as a directed, acyclic graph (DAG)**

- **Why is that helpful?**
  - Clear definition of forward-pass based on symbolic operation instructions
  - Identification of values (weights, output of operations, etc) being used multiple times
  - Clear definition of backward-propagation
  - Use graph theory to optimize processing (e.g. merging of nodes) (advanced)
  - ! Identify independent subgraphs for efficient, parallel processing



TensorFlow internals: Compute architecture

- Computing architecture

High-level interface (Python) **Low-level library (C++)** **Devices (CPU,GPU,...)**

- TensorFlow: eager execution (*instantly return results*)

```
import tensorflow as tf

x = tf.constant([1.0, 2.0, 3.0])
W = tf.constant([[0.5, 1.5, 0.8], [0.0, 2.1, 0.6]])
b = tf.constant([5.0, 5.0])

prod = W @ x  # @ -> matmul
c = prod + b

print(prod)
print(c)
```

while **developing**, *intermediate* results can help debugging

in **production**, we are usually not interested in *intermediate* results

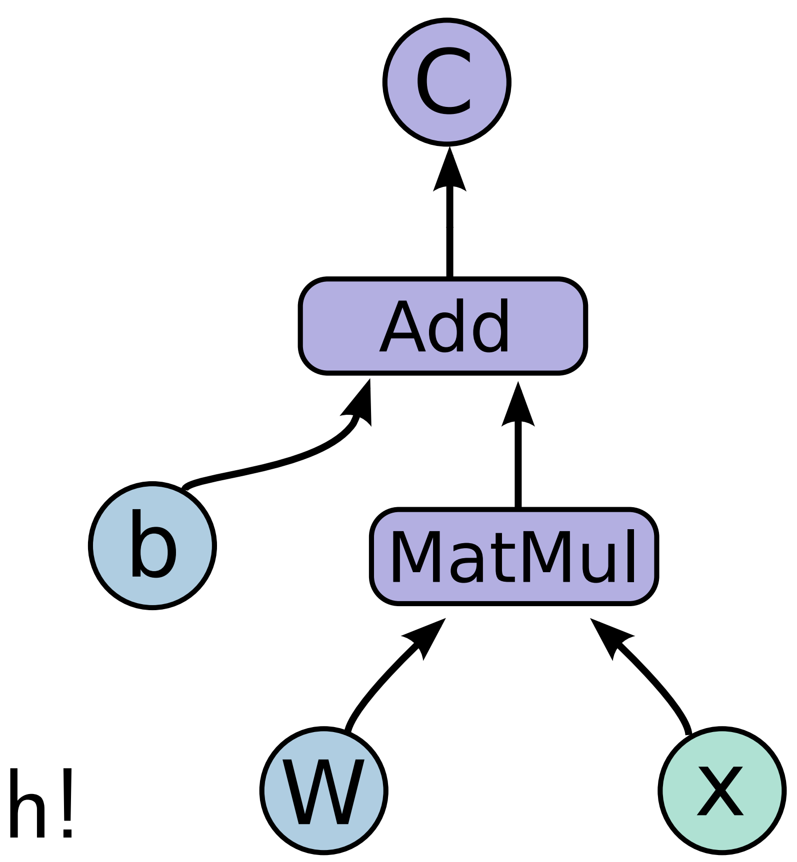
Question: how to tell TensorFlow?

- Functions decorated by `tf.function` become **templates for graphs**

```python
import tensorflow as tf

x = tf.constant([1.0, 2.0, 3.0])
W = tf.constant([[0.5, 1.5, 0.8], [0.0, 2.1, 0.6]])
b = tf.constant([5.0, 5.0])
```

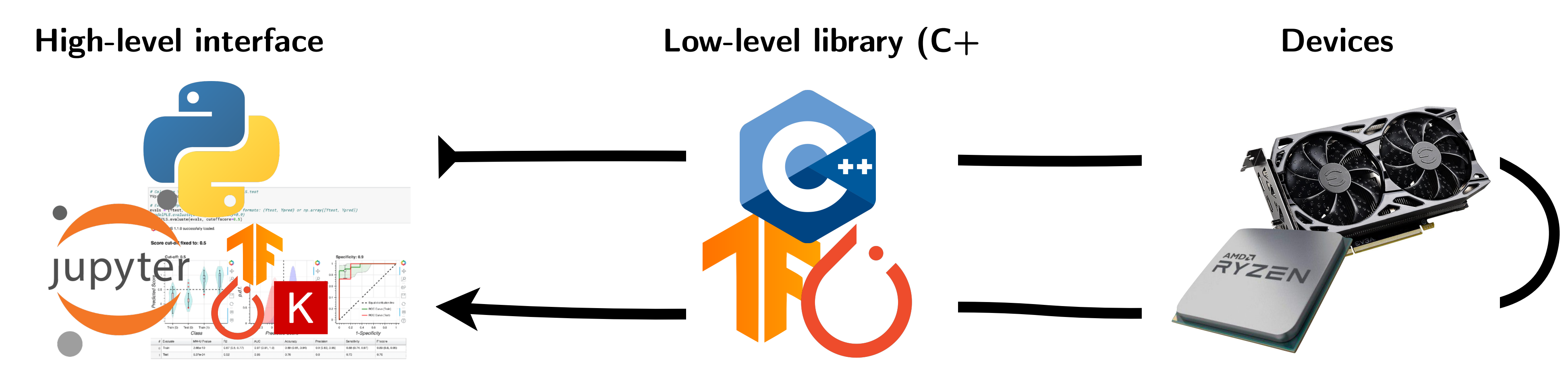
```python
@tf.function
def my_operation(W, x, b):
    prod = W @ x  # @ -> matmul
    print(prod)
    return prod + b
```

```python
c = my_operation(W, x, b)
print(c)
```



→ Inputs and output clearly defined → create graph!

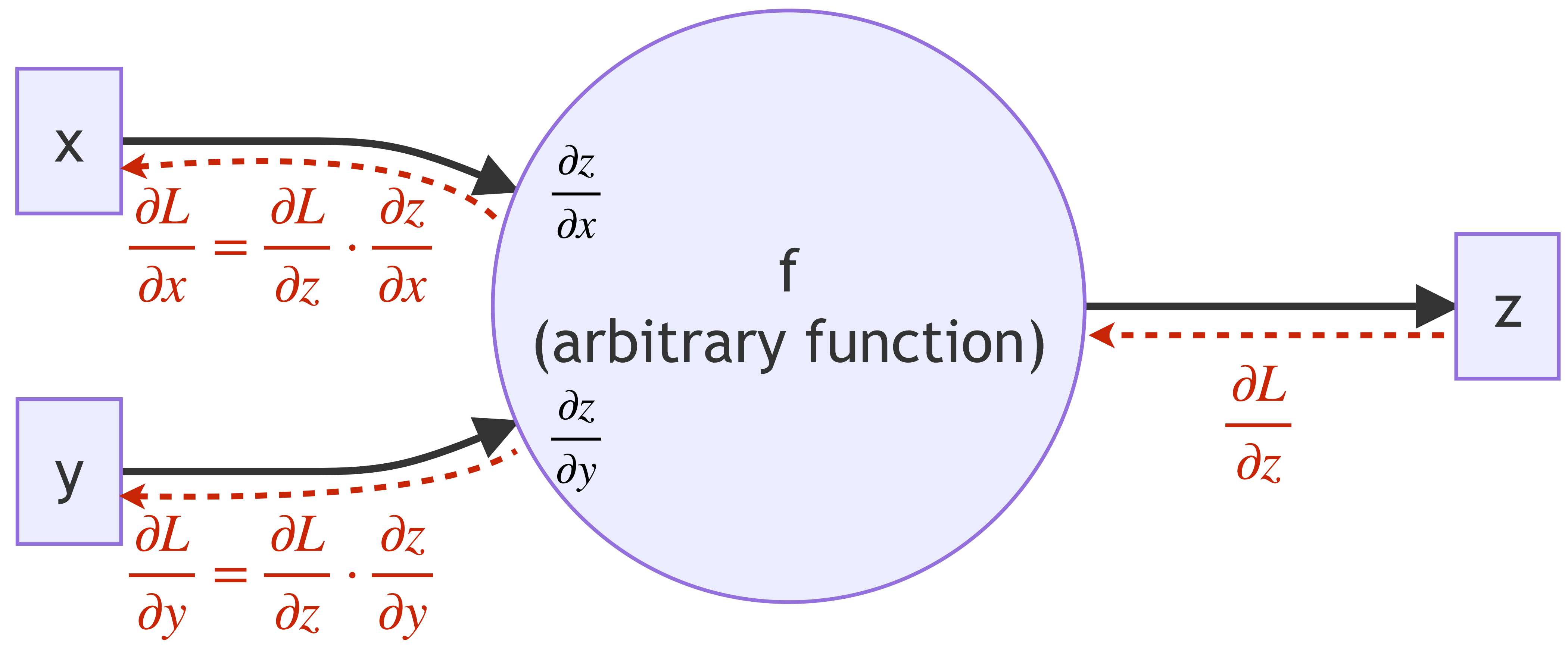
→ Intermediate objects neither transferred to Python interpreter **nor between devices!**



- More technical insights
  - `tf.function` converts any python code (for and while loops, if conditions, ...) to a graph and executes it in C++
  - Decorated functions are *polymorphic* → accept input tensor of any shape and type
  - A new graph is built every time yet unseen {shape, type} combinations of the parameters are used
    - ▻ Calling `my_operation(W, x, b)` **once** will also `print(prod)` (but only showing a symbolic tensor)
    - ▻ Calling `my_operation(W, x, b)` **a second time** won't! Graph is **already built** so no need to call it again.

TensorFlow gradient tape

- From a previous example: local gradients $\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}$ are already computed in the forward pass



- In TensorFlow, this can be instructed through a `tf.GradientTape`

```
# guard all execution with a gradient tape
with tf.GradientTape() as tape:
    # get predictions
    predictions = model(inputs, training=True)

    # compute the loss losses
    loss_value = loss(labels, predictions)

    # get and propagate gradients
    gradients = tape.gradient(loss_value, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

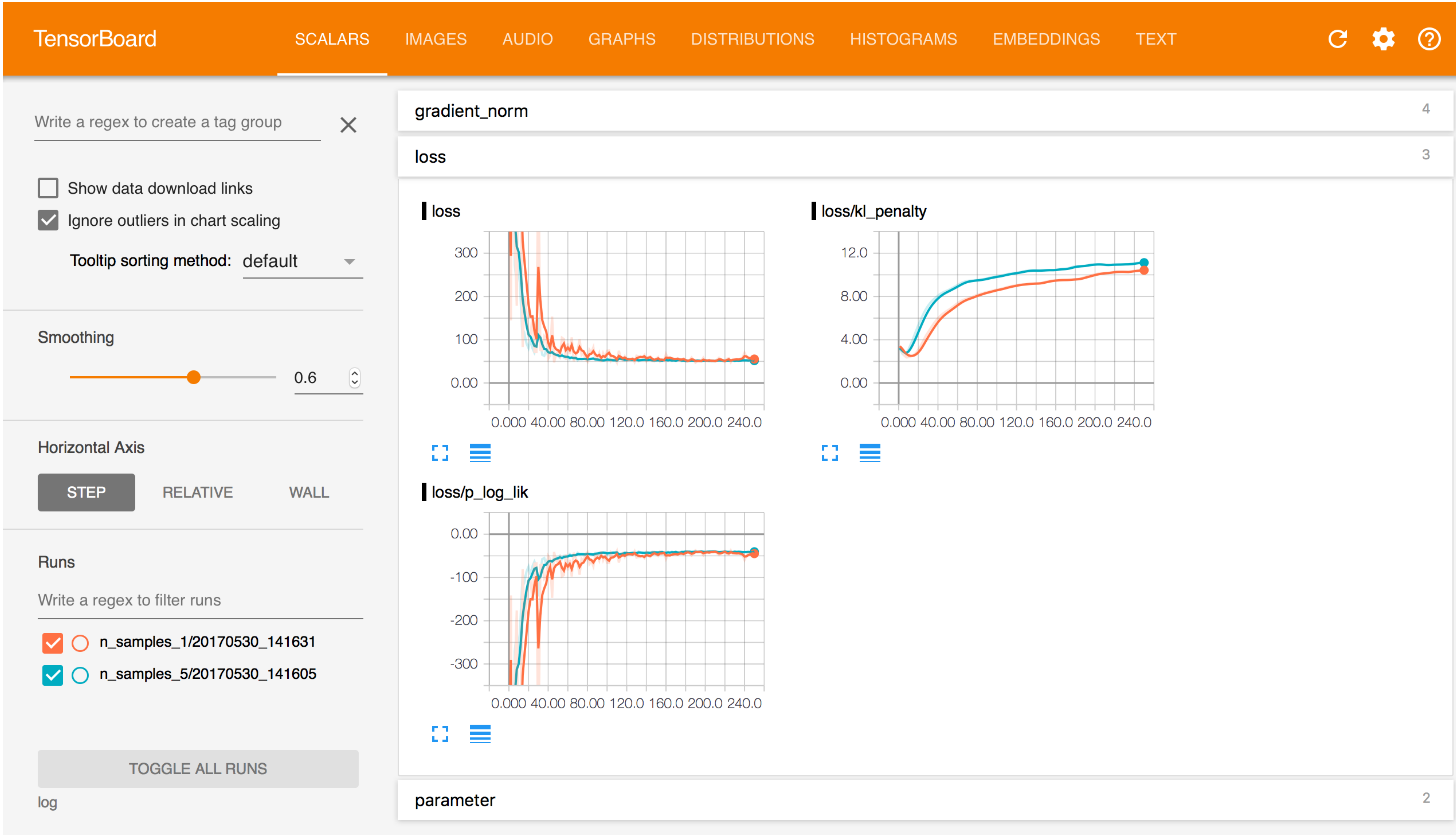
$y = f(x \,|\, \omega)$

$L = L(\hat{y}, y)$

$\frac{\partial L}{\partial \omega}$

back-propagation

- Access to live insights of your training(s) metrics via your browser

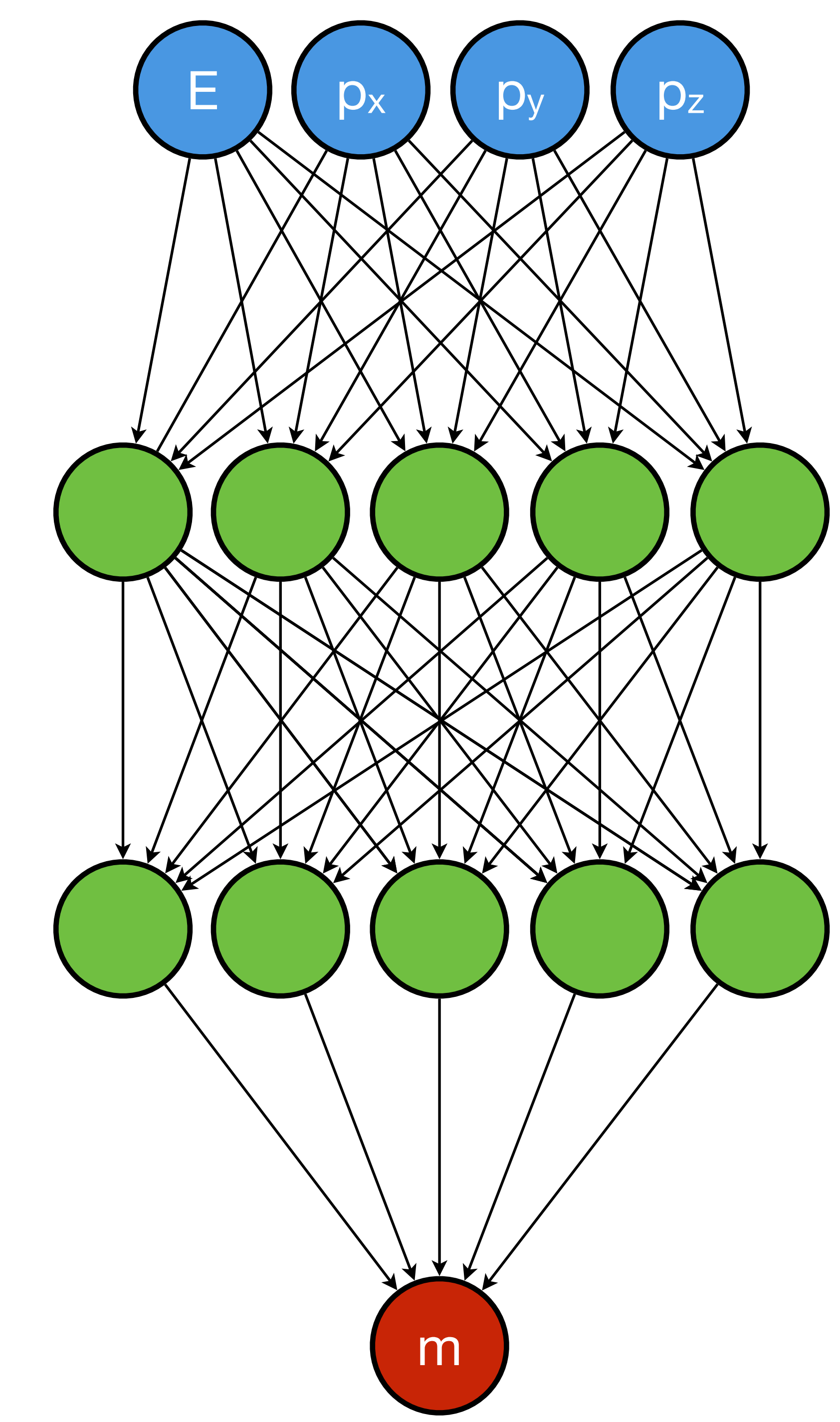


Hands-on!

- **Regression task**
  - You are given randomly generated particle four-vectors
    - They are generated on-the-fly, so no need for dataset splitting
    - You can choose the basis (E, p_x, p_y, p_z) or (E, p_T, η, φ)
  - The network should be trained to reconstruct the particle mass
    - "Simple" relativistic computation for us
    - Potentially hard for the network
      - Build four squares
      - Subtract correctly from one another
      - Extract the square root
  - Colab notebook
    - Complete and optimize the training

$$q = (E, p_x, p_y, p_z)^T$$

$$q = (E, p_x, p_y, p_z)^T$$



$$m = \sqrt{E^2 - p^2}$$

Yesterday
14:30 - 16:00

20" **1. Variants of and improvements in fully-connected networks (FCNs) ✔**
- Gradient calculation (recap), vanishing gradients, ResNet, ensemble learning, multi-purpose networks

30" **2. Numerical insights & considerations ✔**
- Domains, feature & output scaling, batch normalization, SELU, categorical embedding, class imbalance

40" **3. Techniques 1/2 & hands-on ✔**
- Keras functional API, custom Keras layer, computing gradients

Yesterday
16:30 - 18:00

25" **4. Regularization & overtraining suppression ✔**
- Overtraining & generalization, capacity & capability, regularization, dataset splitting

25" **5. Model optimization ✔**
- Optimizer choices, class-importance, hyper-parameters, search strategies

40" **6. Techniques 2/2 & hands-on ✔**
- Compute architecture, TensorFlow eager and graph, custom training loop, tensorboard

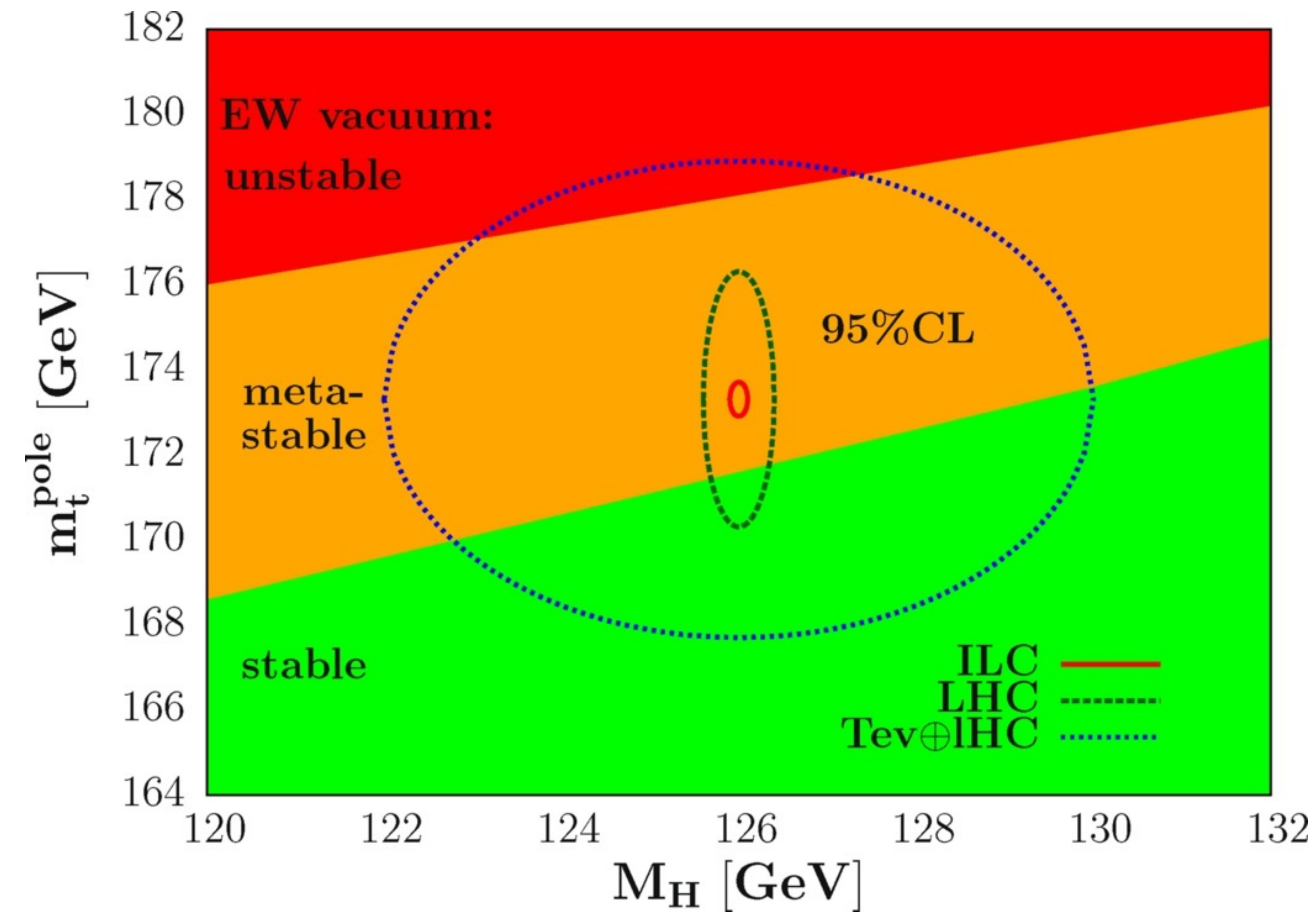
Today
09:00 - 10:30

10" **7. Exercise introduction: Identifying Jets in Particle Collider Experiments**
- Problem statement, input data & features, objective(s)

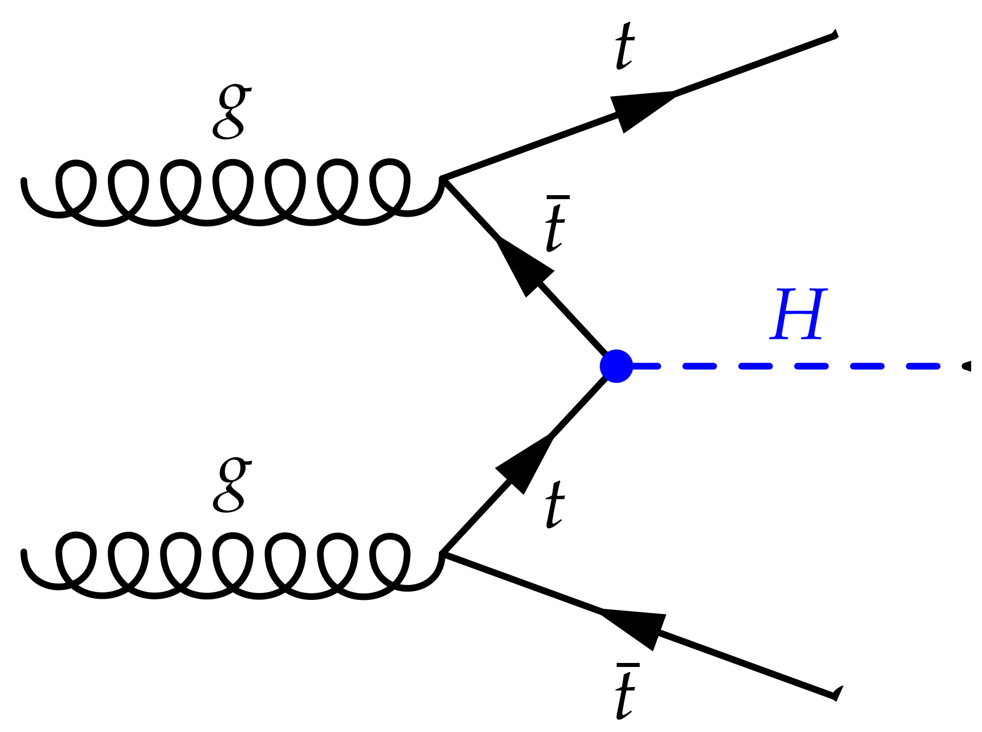
70" **8. Hands-on!**
- Classification task, implementing newly learned techniques, extension to multi-purpose network

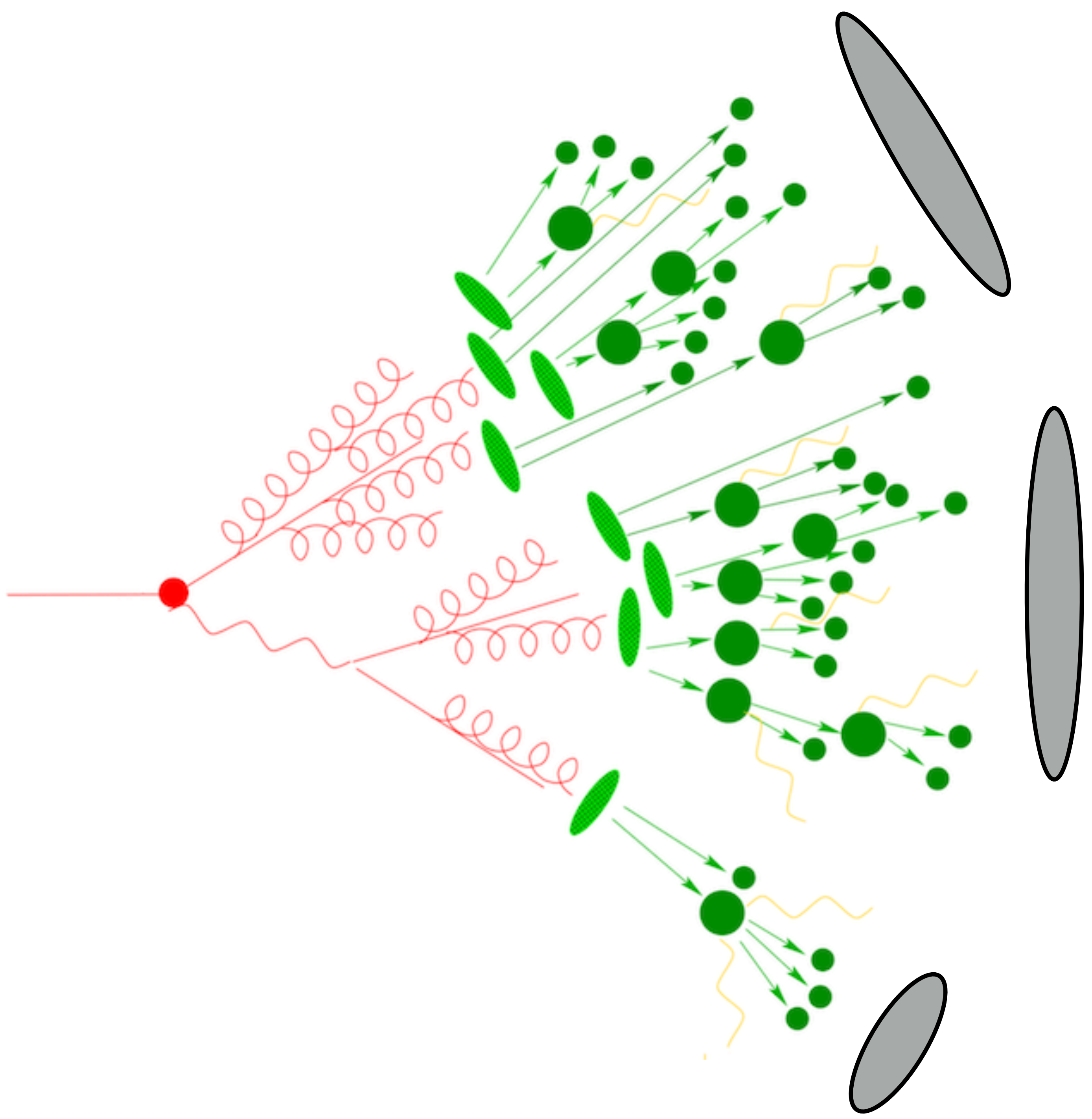
10" **9. Exercise summary and tips**
- Example wrap-up, additional practical tips

7. Exercise introduction:
Identifying Jets in Particle Collider Experiments

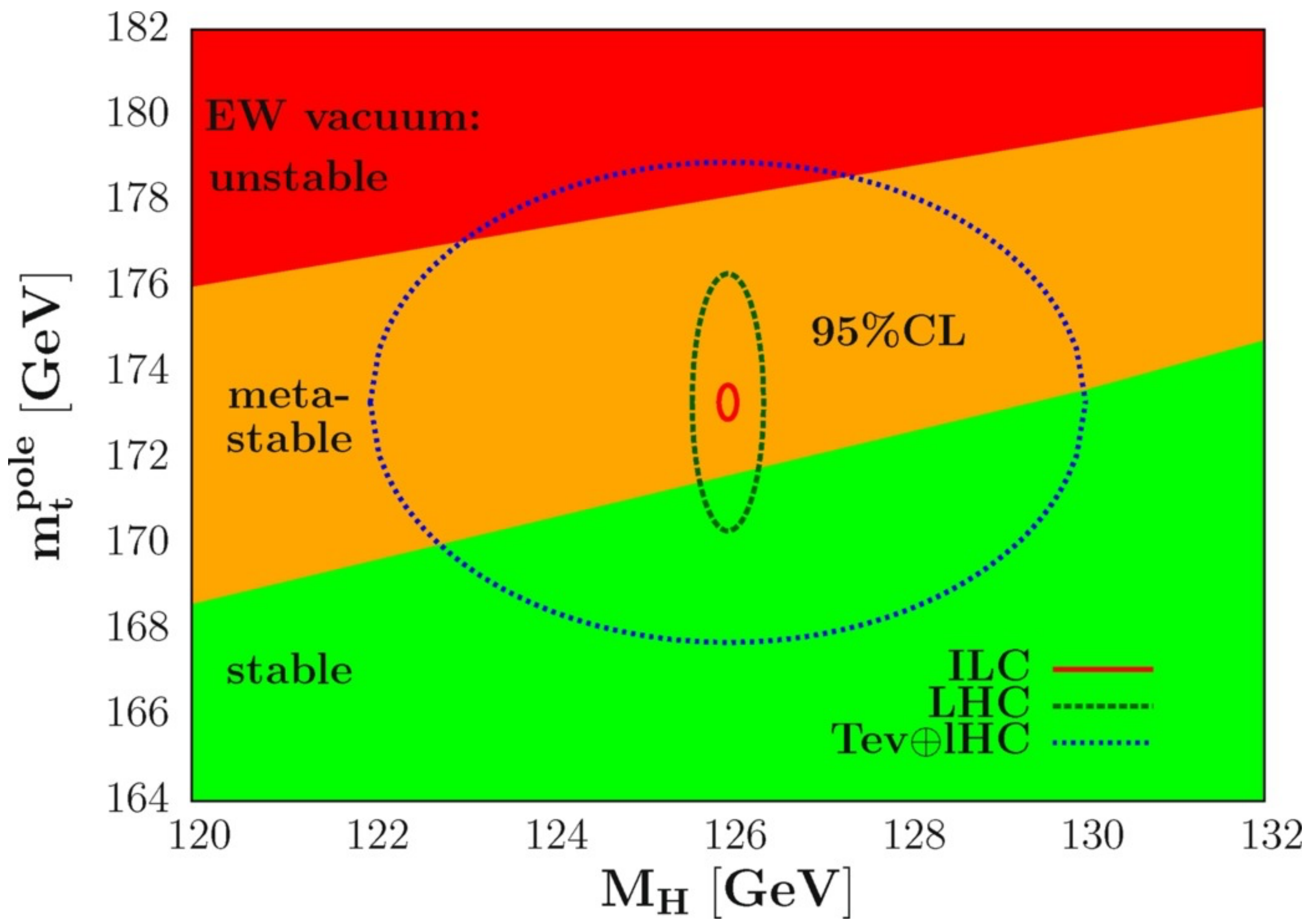


- Heaviest particle known to date (comparable to a tungsten atom)
- Exact knowledge of mass gives insights to electroweak vacuum stability
- High mass causes sizable strength of Higgs-top coupling
  - Example: "ttH" production

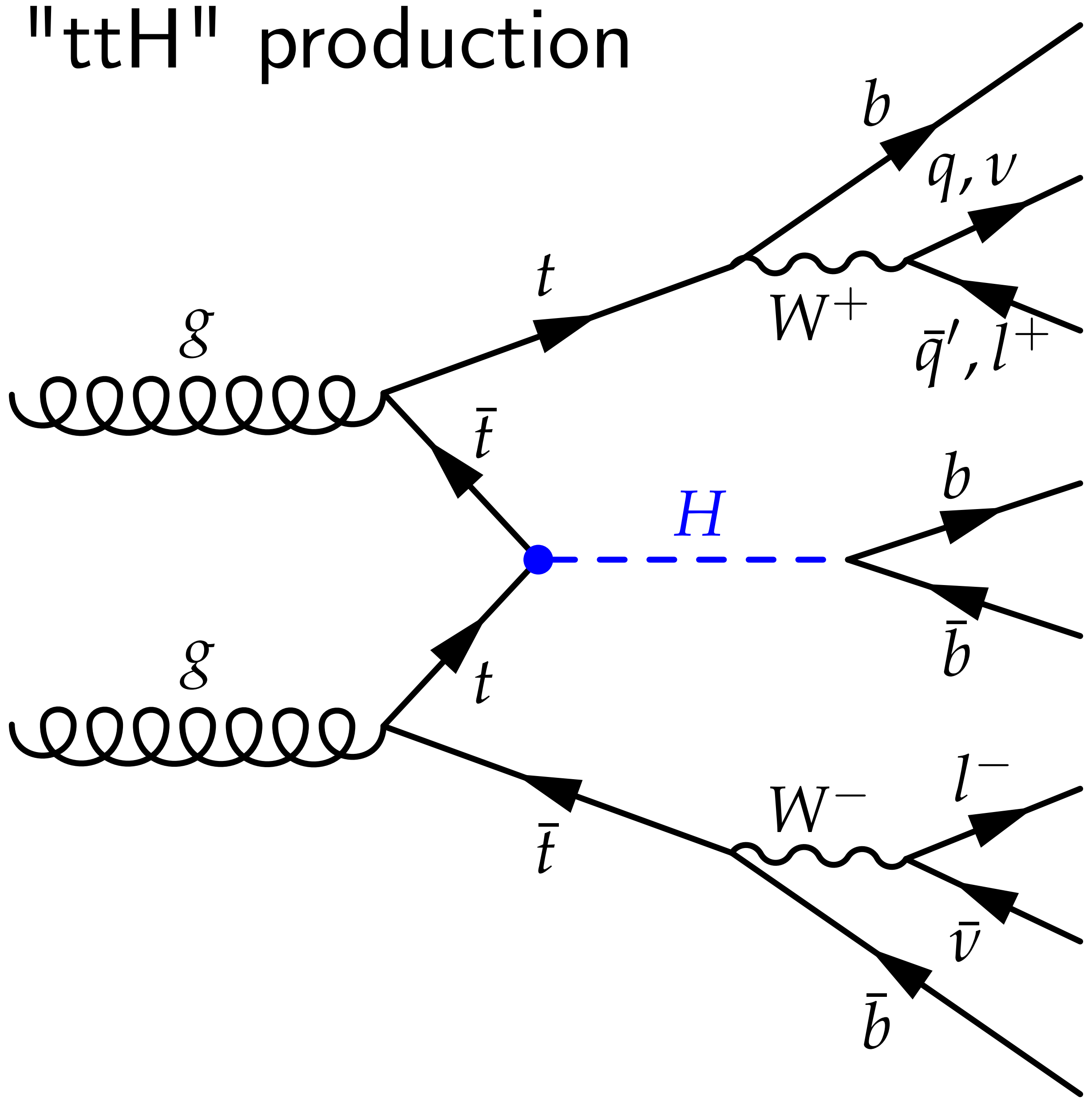


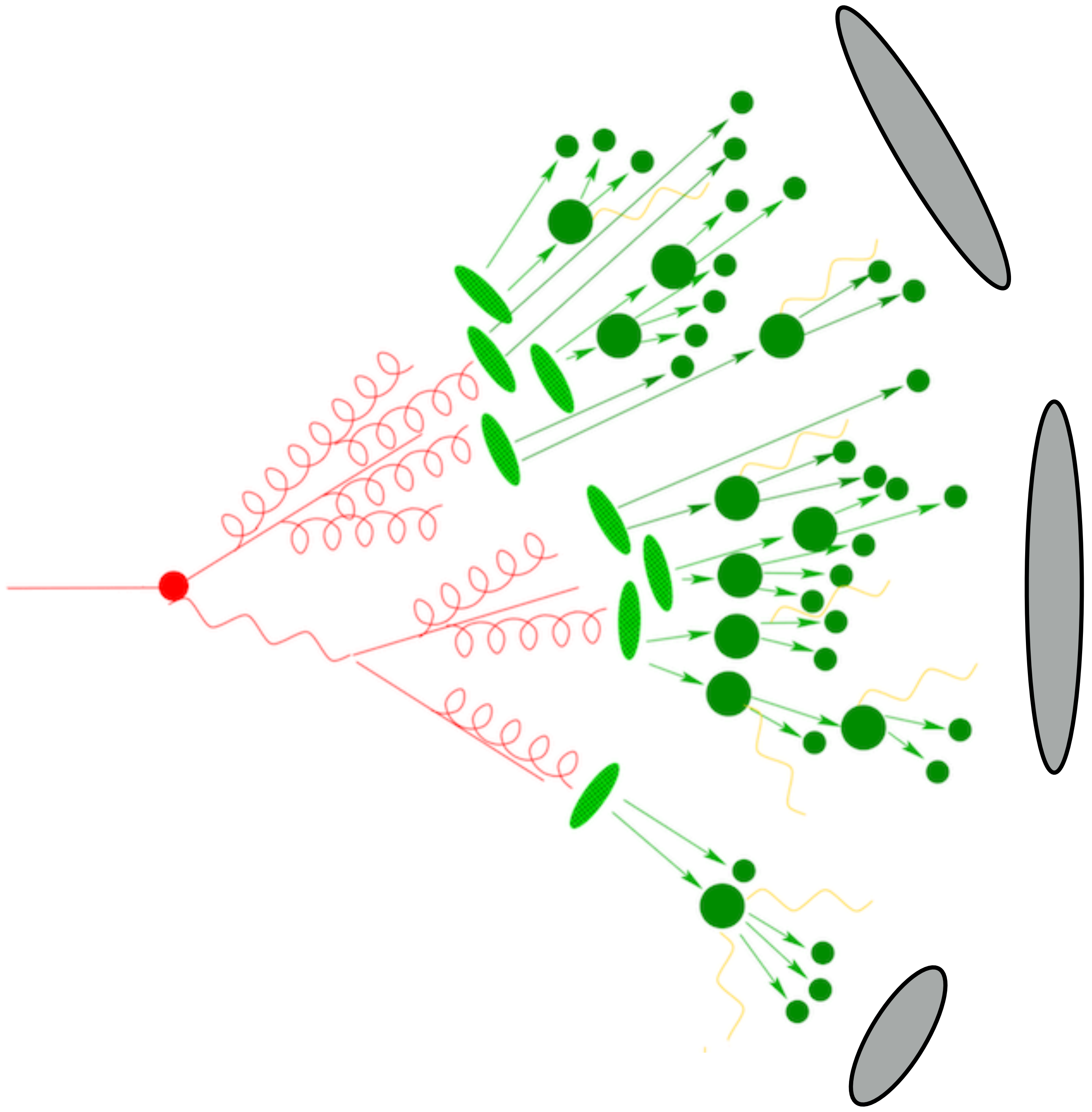


- Decay virtually exclusively into b quark and W boson
- Quarks form collimated jets of (many) stable particles in the detector
- Up to eight jets measurable!
  - **Clear identification of all jets of top decays desirable**

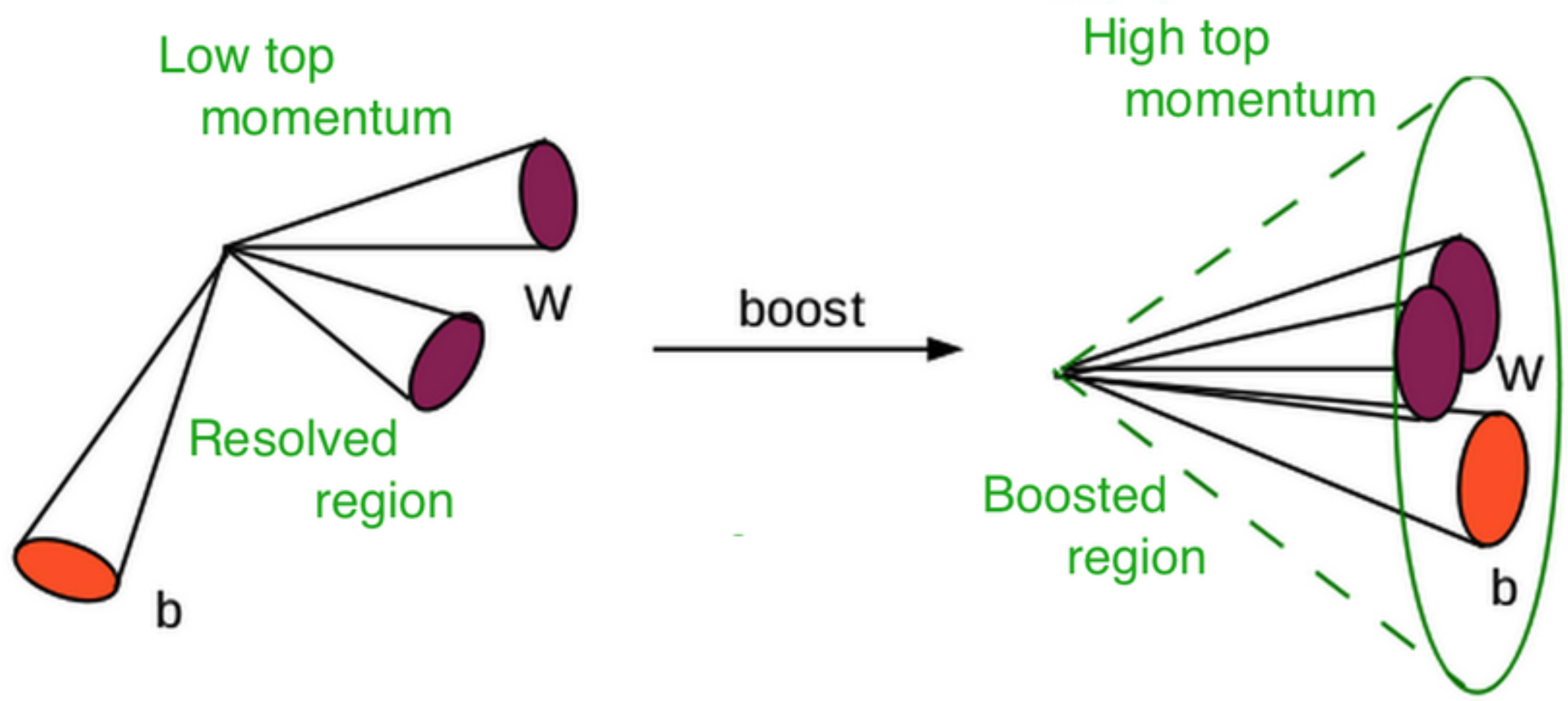


- Heaviest particle known to date (comparable to a tungsten atom)
- Exact knowledge of mass gives insights to electroweak vacuum stability
- High mass causes sizable strength of Higgs-top coupling
  - Example: "ttH" production



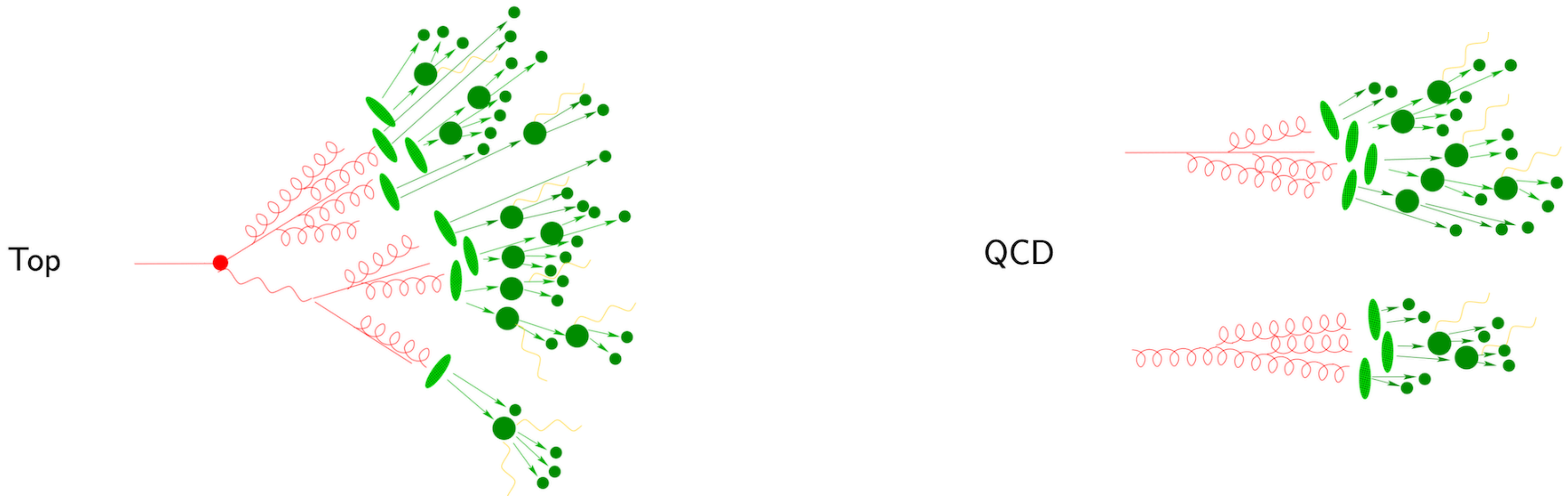


- Decay virtually exclusively into b quark and W boson
- Quarks form collimated jets of (many) stable particles in the detector
- Up to eight jets measurable!
  - **Clear identification of all jets of top decays desirable**

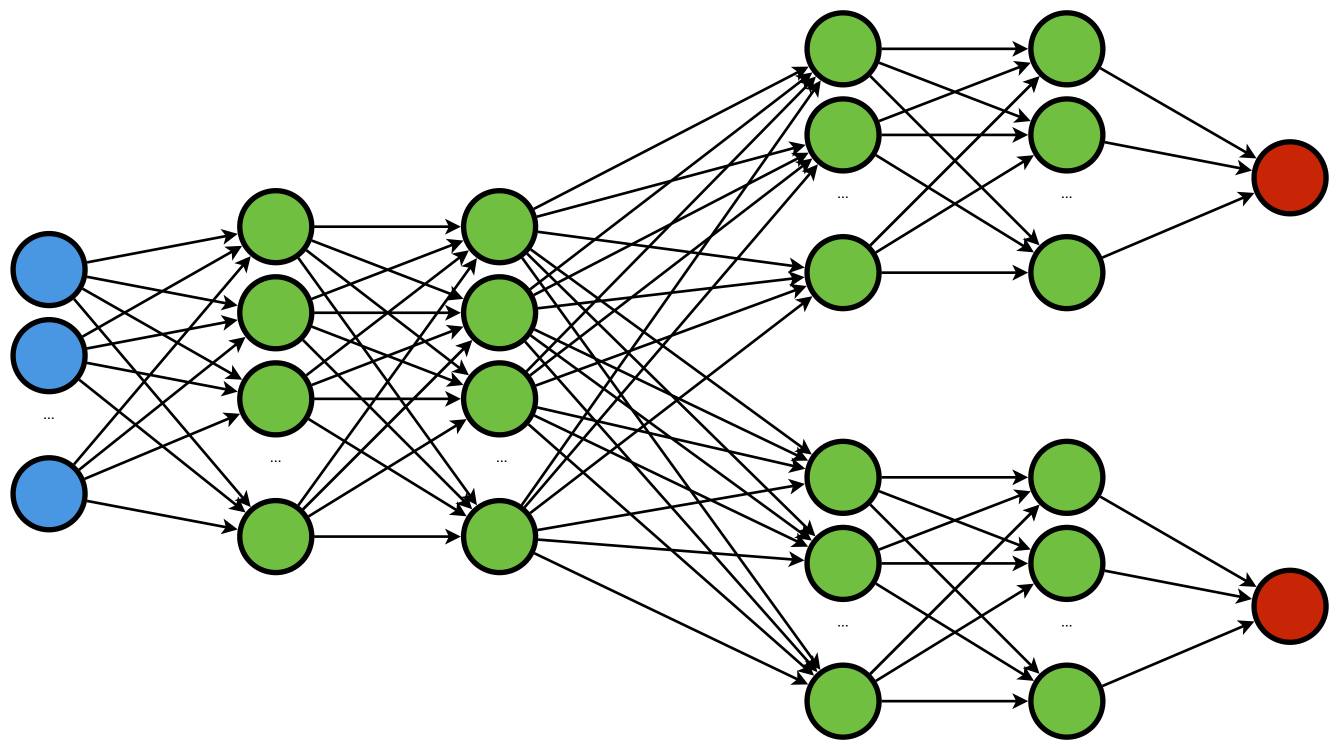


High top quark momentum leads to all decay products being collimated in a single, large jet ("**fat jet**")

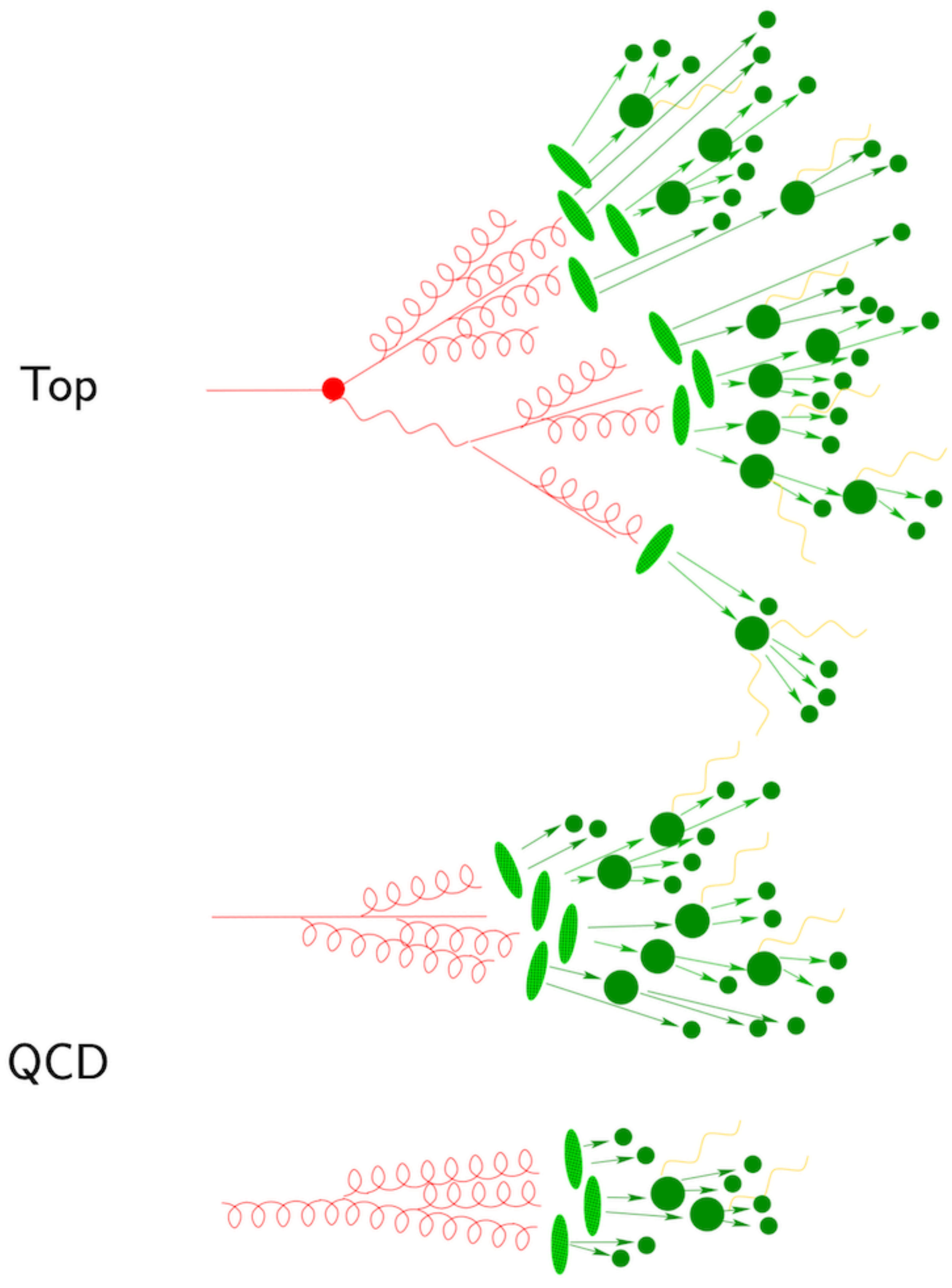
1. **Classification task**
   - Given four-momenta of up to 200 measured particles, distinguish between jets originating from top quarks (signal) or lighter quarks / gluons (background), so-called *QCD* jets



2. **Extension: top quark energy regression in a multi-purpose network**
   - Extend the network to perform a regression task simultaneously
   - Predict the **true** energy of the initial top quark for signal jets



- **Input features**
  - 1.8M jets in 20 "train" files, 8 "valid" files, and 8 "test" files
  - Per jet, you are given the four-vectors of up to 200 of its constituents
    - ▻ Total of up to 800 values per jet
    - ▻ Note that jets might have less constituents !
  - To spare you the trouble of working with uneven (so-called jagged) arrays, constituents vectors are padded with zeros

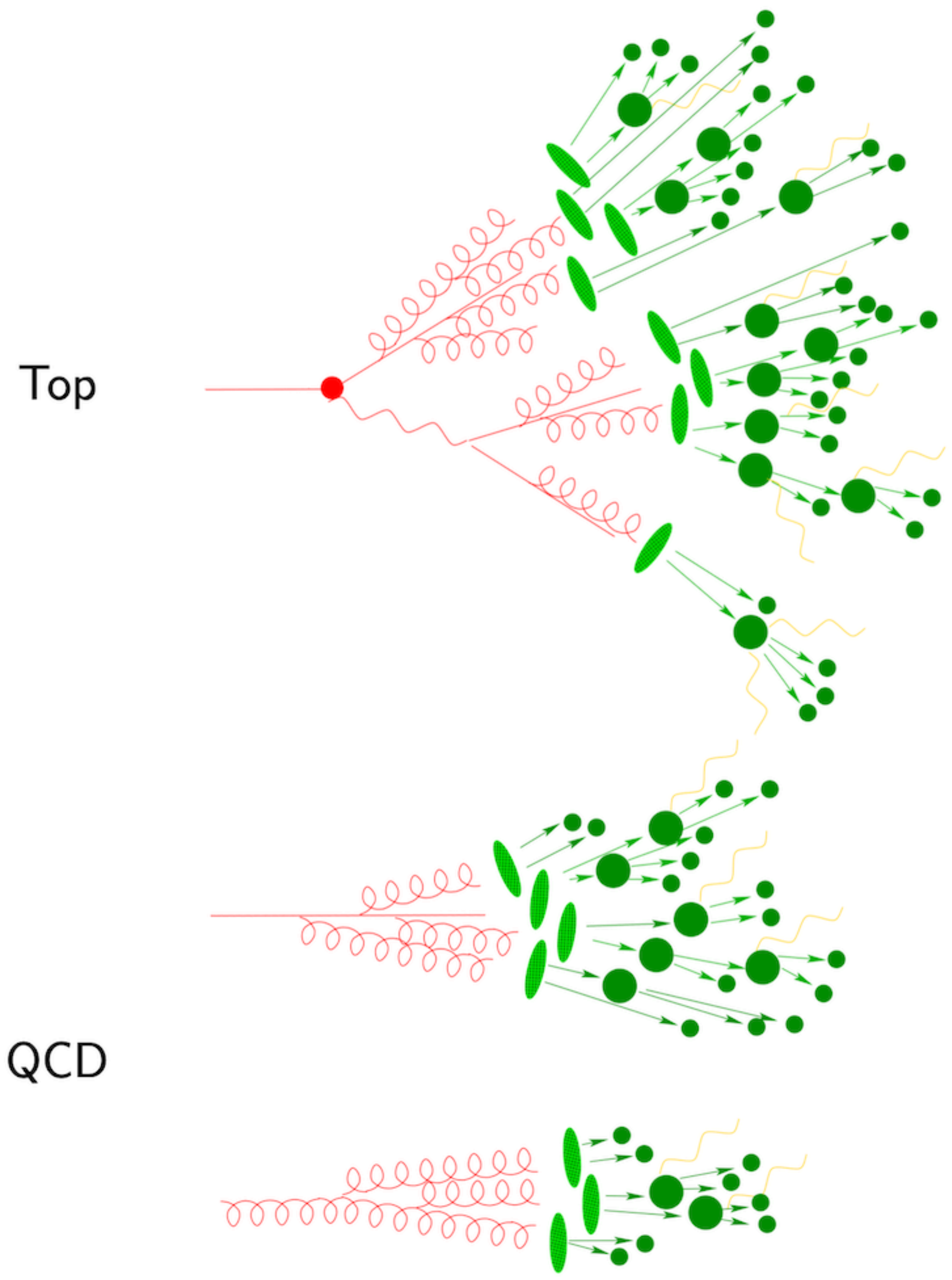


- **Training targets**
  - Per jet, you are provided 2 different training targets:
    - ▻ A flag that marks the true origin of the jet
      - – 1 for jets from top quark decays
      - – 0 for light jets from QCD events
  - The true four-vector of the initial particle (only for top quarks)

8. Hands-on!

- Colab notebook
  - Divided into 6 parts, with a lot of refreshers for easier starting point later on
    1. TensorFlow refresher
    2. Refresher of NN terminology
    3. The tutorial dataset
    4. Minimal training and evaluation workflow
    5. Advanced training loop
    6. (opt) Multi-purpose network

  - Work through it
  - Complete missing blocks
  - Perform your first trainings
  - Improve upon it
  - ! Ask questions and discuss



9. Exercise summary and tips

- **While prototyping new networks, use a "lab book"**
  - Manually via actual pen & paper, markdown file, spreadsheets, ...
  - Change one thing at a time and log finding
  - Automated (e.g. for hyper-opt.) via tensorboard, comet.ml, wandb.ai, mlflow.org, ...

  → You are part of the learning process! And things can get very complex very quickly

- **Know your data**
  - Maintain a script to create input feature plots (1d, 2d), means & variances, correlations, obtain class statistics, ...

  → Key to avoid various issues down the road upfront

- **Monitor your training**
  - To improve your network's performance, you need to understand what it does
  - Vanishing gradients? Overtraining? Dead units? Stuck in local minimum? Optimization process too slow? ...

  → Saves you a lot of time and helps making the guessing process more educated

- **Discuss with others**
  - Profit from experience of fellow colleagues and vice versa
  - Exchange new ideas and papers you found
  - ...

  → Helps to stay ahead of the "game"!