ECE 1508S2: Applied Deep Learning

Chapter 2: Advancing Our Toolbox I

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Winter 2024

What's Next

Right now, we are familiar with FNN and how to train them

↳ *We potentially can also handle some other architectures*

But, if we try to implement them from scratch, we could get into trouble

We still need to learn more tricks for having a working implementation

What's Next

Right now, we are familiar with FNN and how to train them

↳ *We potentially can also handle some other architectures*

But, if we try to implement them from scratch, we could get into trouble

We still need to learn more tricks for having a working implementation

---

In this chapter, we advance our bag of tools in four respects

1. *We learn more about optimizers*
   ↳ *more advanced tricks to make gradient descent work*
2. *We learn about hyperparameter tuning*
3. *We learn about data preprocessing*
   ↳ *how to handle data in practice*
4. *Tricks to make training faster and more robust*

Back to Gradient Descent

Let's take a look at *training* in *abstract form* once again

$$\min_{\mathbf{w}} \hat{R}(\mathbf{w}) \qquad \text{(Training)}$$

Recall that $\mathbf{w}$ *includes all weights and biases*; for instance,

In FNN of Assignmet 2, $\mathbf{w}$ has 89,610 entries! In practice much higher!

Back to Gradient Descent

Let's take a look at *training* in *abstract form* once again

$$\min_{\mathbf{w}} \hat{R}(\mathbf{w}) \qquad \text{(Training)}$$

Recall that $\mathbf{w}$ *includes all weights and biases*; for instance,

> *In FNN of Assignmet 2,* $\mathbf{w}$ *has 89,610 entries! In practice much higher!*

Also recall the gradient descent

1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
2: Choose some small ϵ and η, and set $t = 1$
3: **while** $\Delta > \epsilon$ **do**
4: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta \nabla \hat{R}(\mathbf{w}^{(t-1)})$
5: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
6: **end while**

Keep in mind: we almost always use (mini-batch) SGD $\rightsquigarrow$ *let's call it SGD*

Convergence Rate of Optimizers

Recall: we can make gradient descent converging to local minimum if

we set the learning rate η small enough

So is it also with SGD. *But, how fast does the algorithm converge?*

Convergence Rate of Optimizers

Recall: we can make gradient descent converging to local minimum if

we set the learning rate η small enough

So is it also with SGD. *But, how fast does the algorithm converge?*

Speed of an optimization algorithm is evaluated by convergence rate

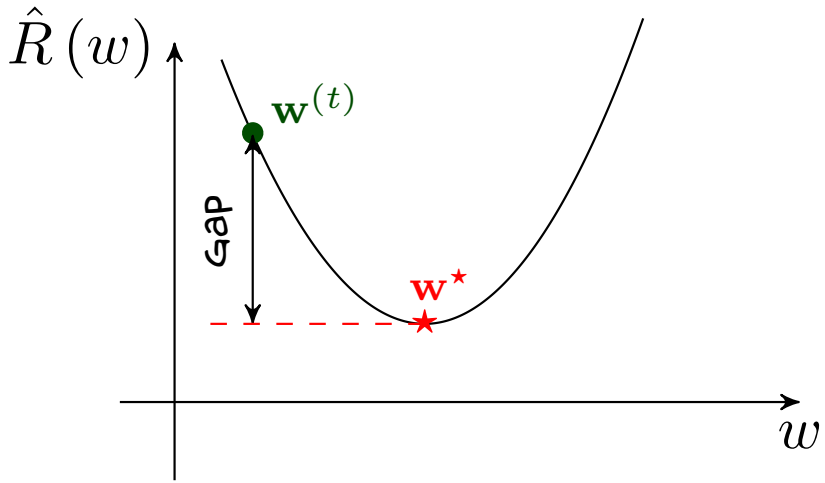
We can understand the its meaning from the eyes of *optimality gap*

Optimality Gap in Iteration t

Optimality gap is the gap between $\mathbf{w}^{(t)}$ and local minimizer $\mathbf{w}^\star$, i.e.,

$$\text{optimality gap in iteration } t = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^\star)|$$

Convergence Rate of Optimizers



An optimizer *shrinks this gap gradually*, i.e., *we can approximately say*[1]

$$|\hat{R}(\mathbf{w}^{(t+1)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant \ldots \leqslant |\hat{R}(\mathbf{w}^{(0)}) - \hat{R}(\mathbf{w}^{\star})|$$

In practice, this drop can occur *with different speeds in terms of* t

[1] It's just approximately correct: *gap may increase in one iteration only! That's no problem*

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

This intuitively means that *for each iteration t, we see*

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^{\star})|$$

for some $\alpha < 1$:

[2] We don't really care how large C is. It's going to shrink at the end

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

This intuitively means that *for each iteration t, we see*

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^{\star})|$$

for some $\alpha < 1$: in this case, we can say

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^{\star})|$$

[2] We don't really care how large C is. It's going to shrink at the end

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

This intuitively means that *for each iteration t, we see*

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^\star)| \leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^\star)|$$

for some $\alpha < 1$: in this case, we can say

$$\begin{aligned} |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^\star)| &\leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^\star)| \\ &\leqslant \alpha(\alpha |\hat{R}(\mathbf{w}^{(t-2)}) - \hat{R}(\mathbf{w}^\star)|) \end{aligned}$$

[2] We don't really care how large C is. It's going to shrink at the end

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

This intuitively means that *for each iteration t, we see*

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^\star)| \leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^\star)|$$

for some $\alpha < 1$: in this case, we can say

$$\begin{aligned} |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^\star)| &\leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^\star)| \\ &\leqslant \alpha(\alpha |\hat{R}(\mathbf{w}^{(t-2)}) - \hat{R}(\mathbf{w}^\star)|) = \alpha^2 |\hat{R}(\mathbf{w}^{(t-2)}) - \hat{R}(\mathbf{w}^\star)| \\ &\leqslant \dots \end{aligned}$$

[2] We don't really care how large C is. It's going to shrink at the end

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

This intuitively means that *for each iteration t, we see*

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^\star)| \leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^\star)|$$

for some $\alpha < 1$: in this case, we can say

$$\begin{aligned}
|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^\star)| &\leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^\star)| \\
&\leqslant \alpha(\alpha |\hat{R}(\mathbf{w}^{(t-2)}) - \hat{R}(\mathbf{w}^\star)|) = \alpha^2 |\hat{R}(\mathbf{w}^{(t-2)}) - \hat{R}(\mathbf{w}^\star)| \\
&\leqslant \ldots \leqslant \alpha^t |\hat{R}(\mathbf{w}^{(0)}) - \hat{R}(\mathbf{w}^\star)|
\end{aligned}$$

[2] We don't really care how large C is. It's going to shrink at the end

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

This intuitively means that *for each iteration t, we see*

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^{\star})|$$

for some $\alpha < 1$: in this case, we can say

$$\begin{aligned}
|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| &\leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^{\star})| \\
&\leqslant \alpha(\alpha |\hat{R}(\mathbf{w}^{(t-2)}) - \hat{R}(\mathbf{w}^{\star})|) = \alpha^2 |\hat{R}(\mathbf{w}^{(t-2)}) - \hat{R}(\mathbf{w}^{\star})| \\
&\leqslant \ldots \leqslant \alpha^t |\hat{R}(\mathbf{w}^{(0)}) - \hat{R}(\mathbf{w}^{\star})|
\end{aligned}$$

Now say that $|\hat{R}(\mathbf{w}^{(0)}) - \hat{R}(\mathbf{w}^{\star})| = C$.[2] So, we can say

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant C\alpha^t \rightsquigarrow \mathcal{O}(\alpha^t)$$

[2] We don't really care how large C is. It's going to shrink at the end

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

If we wish to *end up somewhere in ϵ-neighborhood of* $\mathbf{w}^\star$; then, we need

$$C\alpha^t \leqslant \epsilon$$

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

If we wish to *end up somewhere in ϵ-neighborhood of $\mathbf{w}^\star$*; then, we need

$$C\alpha^t \leqslant \epsilon$$

For this to happen, *we need to have at least*

$$t \geqslant \frac{\log 1/\epsilon + \log C}{\log 1/\alpha} \rightsquigarrow \mathcal{O}(\log 1/\epsilon)$$

iterations:

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

If we wish to *end up somewhere in ϵ-neighborhood of $\mathbf{w}^\star$*; then, we need

$$C\alpha^t \leqslant \epsilon$$

For this to happen, *we need to have at least*

$$t \geqslant \frac{\log 1/\epsilon + \log C}{\log 1/\alpha} \rightsquigarrow \mathcal{O}(\log 1/\epsilon)$$

iterations: the closer we need to get, the more we should iterate

For this required time, we say that the optimizer converges linearly

It's a fast rate, since number of iterations is proportional to logarithm of $1/\epsilon$

Convergence Rate of Optimizers

+ *You said ideal! Isn't gradient descent always converging at this rate?*
– Well! Only when empirical risk is *strongly convex and we do full-batch training*! You can guess it happens almost never for us!
+ *But how it works with realistic NNs and SGD?*

[3] To be rigorous: when it's Lipschitz continuous, but we don't really need details on that

Convergence Rate of Optimizers

+ *You said ideal! Isn't gradient descent always converging at this rate?*
– Well! Only when empirical risk is *strongly convex and we do full-batch training*! You can guess it happens almost never for us!
+ *But how it works with realistic NNs and SGD?*

In general, it's hard to characterize exact convergence; however, we know that *when empirical risks are rather smooth functions*[3], *we have*

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^\star)| \leqslant \frac{L}{t\eta}$$

for some L *that gets larger as* $\mathbf{w}$ *becomes larger*. So, in practice, we have

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^\star)| = \mathcal{O}(1/t)$$

[3] To be rigorous: when it's Lipschitz continuous, but we don't really need details on that

Convergence Rate of Optimizers

In practice, we have

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| = \mathcal{O}(1/t)$$

So, if we want to end up somewhere in ϵ-neighborhood of $\mathbf{w}^{\star}$, we need

$$t = \mathcal{O}(1/\epsilon)$$

Convergence Rate of Optimizers

In practice, we have

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| = \mathcal{O}(1/t)$$

So, if we want to end up somewhere in ϵ-neighborhood of $\mathbf{w}^{\star}$, we need

$$t = \mathcal{O}(1/\epsilon)$$

This can potentially take long! We say in this case that

the optimizer converges sub-linearly

+ *How bad can it be?*
- Just set $\epsilon = 10^{-3}$: *with linear convergence we need time in order of* 3; *with sub-linear one, we need in order of* 1000!

Alternative Optimizers

Moral of Story

Vanilla gradient descent is not what we can use in practice!

In practice, we employ improved versions of gradient descent:

Alternative Optimizers

Moral of Story

Vanilla gradient descent is not what we can use in practice!

In practice, we employ improved versions of gradient descent: *there is a long list of them, but we check a few important ones that are typically used*

- Gradient descent with *learning rate scheduling*
- Gradient descent with *momentum*
- *Rprop: Resilient backpropagation*
- *RMSprop: Root mean square propagation*
- *Adam: Adaptive moment estimation*

Alternative Optimizers

Moral of Story

Vanilla gradient descent is not what we can use in practice!

In practice, we employ improved versions of gradient descent: *there is a long list of them, but we check a few important ones that are typically used*

- Gradient descent with *learning rate scheduling*
- Gradient descent with *momentum*
- *Rprop: Resilient backpropagation*
- *RMSprop: Root mean square propagation*
- *Adam: Adaptive moment estimation*

\+ *If gradient descent is not used in practice, why we did backpropagation?*

– No worries! They *all use gradient!* This is why these algorithms are commonly referred to as *gradient-based training algorithms*

SGD with Learning Rate Scheduling

We had it in simple words in Chapter 1: *we vary learning rate through time*

$$\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta^{(t-1)} \nabla \hat{R}(\mathbf{w}^{(t-1)})$$

It's called *learning rate scheduling*: *start at large* $\eta^{(0)} = \eta$ *and reduce it with* t

SGD with Learning Rate Scheduling

We had it in simple words in Chapter 1: *we vary learning rate through time*

$$\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta^{(t-1)} \nabla \hat{R}(\mathbf{w}^{(t-1)})$$

It's called *learning rate scheduling*: *start at large* $\eta^{(0)} = \eta$ *and reduce it with* t

---

We may schedule learning rate with various approaches

- We could have *linear decay*

$$\eta^{(t)} = \frac{\eta}{t+1}$$

- We could have *polynomial decay* with power P

$$\eta^{(t)} = \frac{\eta}{(t+1)^P}$$

- We could have *exponential decay* with some exponent rate $\kappa > 0$

$$\eta^{(t)} = \eta \, e^{-\kappa t}$$

SGD with Learning Rate Scheduling

There is also a *more practical trick* for learning rate scheduling:

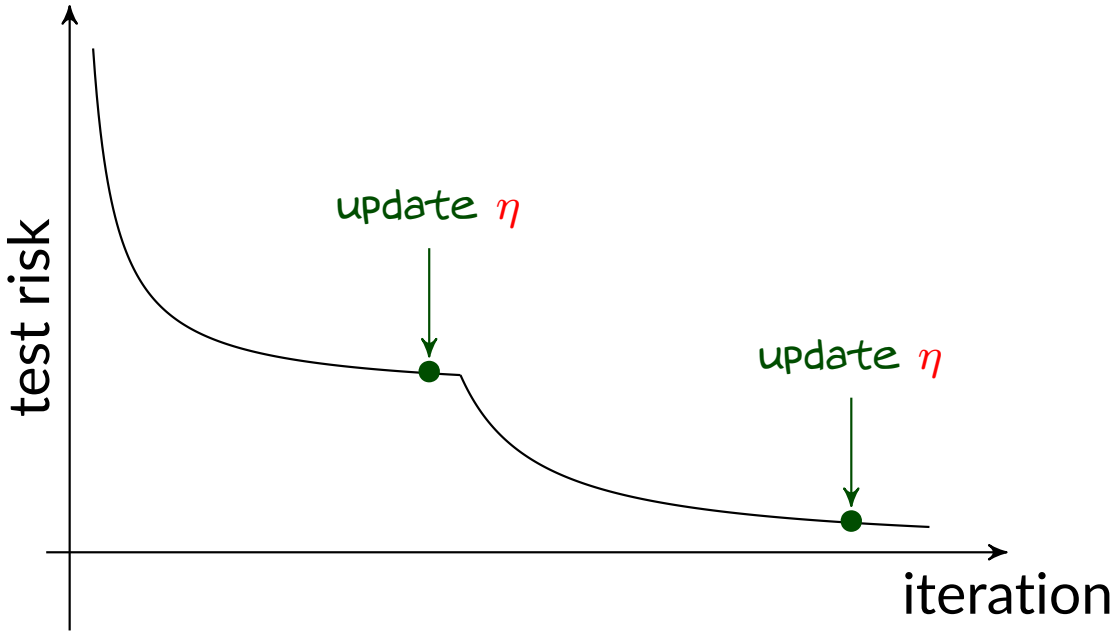
1. *Use a fixed learning rate η until the test risk saturates*
2. *Reduce the learning rate as $\eta \leftarrow \alpha\eta$ for $\alpha < 1$*
3. *Repeat the above steps until test risk arrives at a desired level*

SGD with Learning Rate Scheduling

There is also a *more practical trick* for learning rate scheduling:

1. *Use a fixed learning rate η until the test risk saturates*
2. *Reduce the learning rate as $\eta \leftarrow \alpha\eta$ for $\alpha < 1$*
3. *Repeat the above steps until test risk arrives at a desired level*

A good choice of α is around $\boxed{\alpha = 0.1}$



SGD with Learning Rate Scheduling

Learning rate scheduling can lead us to a better local minima; however,

it does not change the convergence rate

It is nevertheless a *good approach for easy problems*

We can access pre-implemented scheduling techniques in PyTorch

```
>> import torch

>> torch.optim.lr_scheduler
```

Momentum: *Moving Average*

Momentum is one of the key approaches to robust SGD

The idea is simple: *we replace the gradient*[4] *with its moving average*

[4] Which can largely variate, especially with small mini-batches

Momentum: *Moving Average*

Momentum is one of the key approaches to robust SGD

The idea is simple: *we replace the gradient*[4] *with its moving average*

Say we are in iteration t and let the computed gradient to be $\mathbf{g}^{(t)}$, i.e.,

$$\mathbf{g}^{(t)} = \texttt{estimator}\left\{\nabla \hat{R}(\mathbf{w}^{(t)})\right\}$$

e.g., we computed by SGD

With standard gradient descent we update as

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta^{(t)}\mathbf{g}^{(t)}$$

In momentum approach, we replace $\mathbf{g}^{(t)}$ with its moving average

[4] Which can largely variate, especially with small mini-batches

Momentum: *Moving Average*

Moving Average $\sim$ Momentum

Moving average with factor β in iteration t is

$$\mathbf{m}^{(t)} = \beta \mathbf{m}^{(t-1)} + (1 - \beta)\, \mathbf{g}^{(t)}$$

Momentum: *Moving Average*

Moving Average $\sim$ Momentum

Moving average with factor β in iteration t is

$$\mathbf{m}^{(t)} = \beta \mathbf{m}^{(t-1)} + (1 - \beta)\, \mathbf{g}^{(t)}$$

Moving average has less fluctuations; hence, *it's an estimator of true gradient with less variance*

Momentum: *Moving Average*

Moving Average $\sim$ Momentum

Moving average with factor β in iteration t is

$$\mathbf{m}^{(t)} = \beta \mathbf{m}^{(t-1)} + (1-\beta)\,\mathbf{g}^{(t)}$$

Moving average has less fluctuations; hence, *it's an estimator of true gradient with less variance*

SGD with momentum does the following update

Initiate $\mathbf{m}^{(0)}$
for $t = 1, \ldots$ **do**
$\ldots$
$\mathbf{m}^{(t)} = \beta \mathbf{m}^{(t-1)} + (1-\beta)\,\mathbf{g}^{(t)}$
$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta^{(t)} \mathbf{m}^{(t)}$
$\ldots$
end for

Nesterov Momentum: *Accelerated Gradient Computation*

Nesterov approach adds an *intermediate step: when we compute gradient, we use the already-calculated momentum to estimate future weights*

Initiate $\mathbf{m}^{(0)}$
for $t = 1, \ldots$ **do**
  $\ldots$
  $\hat{\mathbf{w}} = \mathbf{w}^{(t)} - \beta \mathbf{m}^{(t-1)}$ `# approximate next point`
  Compute gradient for weights $\hat{\mathbf{w}}$: call it $\hat{\mathbf{g}}^{(t)}$
  $\mathbf{m}^{(t)} = \beta \mathbf{m}^{(t-1)} + (1 - \beta)\, \hat{\mathbf{g}}^{(t)}$
  $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta^{(t)} \mathbf{m}^{(t)}$
  $\ldots$
end for

You can combine these lines into a single line of update

that may looks a bit more complicated, but it's the same thing!

SGD with Momentum: *Implementation*

In PyTorch, SGD is already implemented with momentum

```
>> import torch

>> torch.optim.SGD()
```

When using this implementation, we can *specify momentum factor*,*i.e.*, β, and choose *whether Nesterov being applied or not*

Typical choice of momentum factor is $\beta = 0.9$, *and remember*

with $\beta = 0$, *we return to the standard SGD*

This is the default value in PyTorch

Rprop: *Resilient Backpropagation*

Rprop was introduced by *Riedmiller and Braun* in 1992; *check the paper here*

> *Riedmiller and Braun noticed that gradient descent can be improved if we could have individual learning rate in each dimension of* $\mathbf{w}$*: you may recall the first question in Assignment 1!*

They hence came up with Rprop: *let's see first one-dimensional case*

Rprop: *Resilient Backpropagation*

Rprop was introduced by *Riedmiller and Braun* in 1992; *check the paper here*

> *Riedmiller and Braun noticed that gradient descent can be improved if we could have individual learning rate in each dimension of* $\mathbf{w}$*: you may recall the first question in Assignment 1!*

They hence came up with Rprop: *let's see first one-dimensional case*

```
Rprop():
    Initiate η^(0) and choose μ^+ > 1, μ^- < 1, η_max and η_min
    for t = 1, ... do
        ...
        Compute gradient at w^(t) and call it g^(t)
        Update learning rate as η^(t) ← Rprop_Scheduler(η^(t-1), g^(t), g^(t-1))
        Update weight w^(t+1) = w^(t) − η^(t) sign(g^(t))  # only sign of gradient
        ...
    end for
```

Rprop: *Learning Rate Scheduler*

They key point of Rprop is its scheduler

```
Rprop_Scheduler():
```

Use $\mu^+ > 1$ and $\eta_{\max}$, as well as $\mu^- < 1$ and $\eta_{\min}$

if $\text{sign}(g^{(t)}) = \text{sign}(g^{(t-1)})$ **then**

Update learning rate $\eta^{(t)} = \min\left\{\mu^+\eta^{(t-1)}, \eta_{\max}\right\}$ `# go faster`

else

Update learning rate $\eta^{(t)} = \max\left\{\mu^-\eta^{(t-1)}, \eta_{\min}\right\}$ `# slow down`

end if

Rprop: *Learning Rate Scheduler*

They key point of Rprop is its scheduler

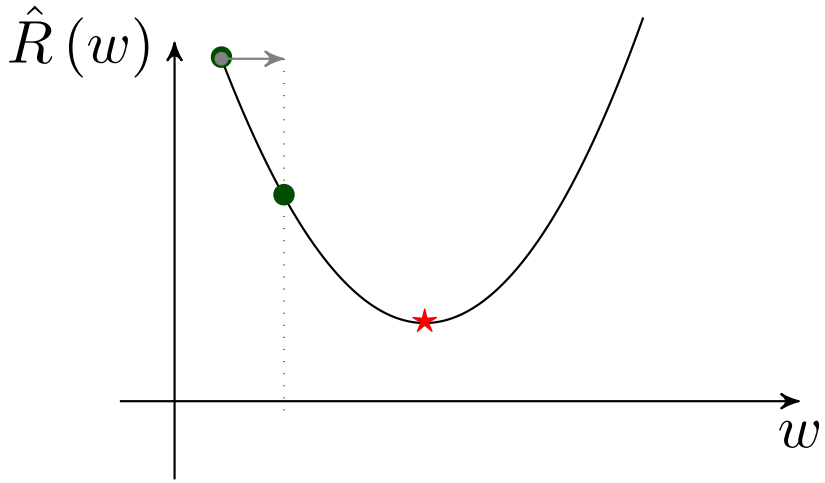
```
Rprop_Scheduler():
    Use μ⁺ > 1 and η_max, as well as μ⁻ < 1 and η_min
    if sign(g^(t)) = sign(g^(t−1)) then
        Update learning rate η^(t) = min{μ⁺ η^(t−1), η_max}        # go faster
    else
        Update learning rate η^(t) = max{μ⁻ η^(t−1), η_min}        # slow down
    end if
```

It follows an intuitive strategy

- *Keep going faster as the sign of gradient is not changing*
- *Slow down only when the sign changes: you have passed the minimum!*

Rprop: *Resilient Backpropagation*

Let's look at it geometrically

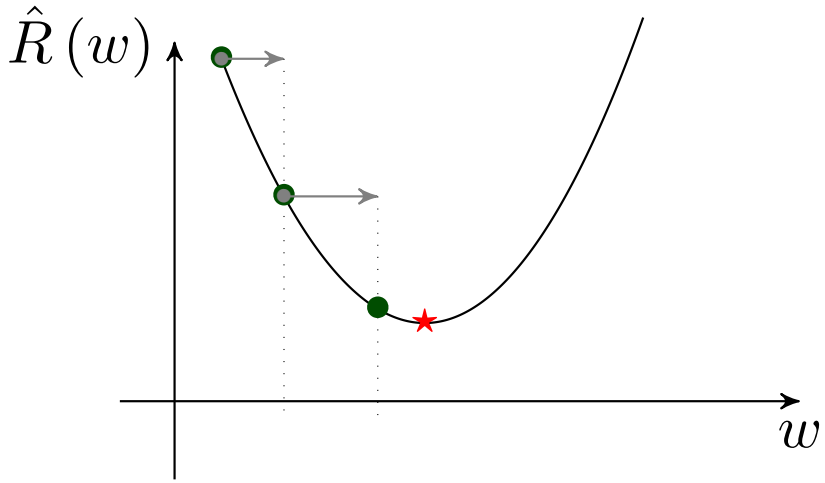


In simple words, with Rprop

we keep on increasing until we pass the minimum

Rprop: *Resilient Backpropagation*

Let's look at it geometrically

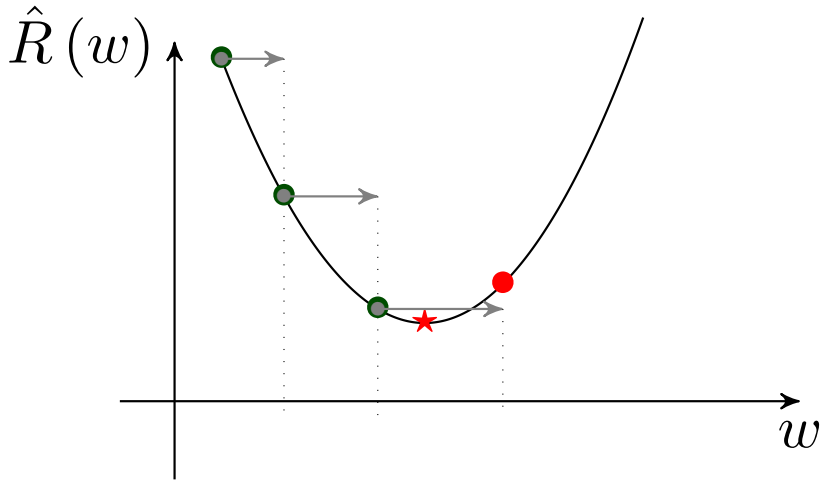


In simple words, with Rprop

we keep on increasing until we pass the minimum

Rprop: *Resilient Backpropagation*

Let's look at it geometrically

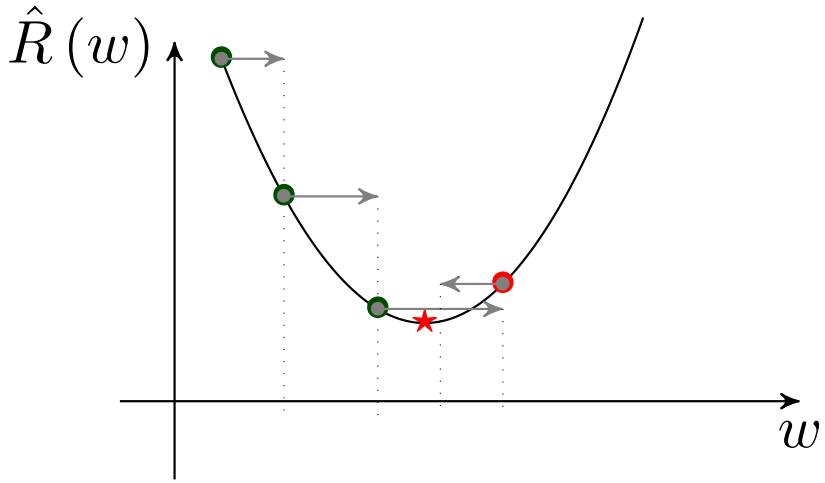


In simple words, with Rprop

we keep on increasing until we pass the minimum

Rprop: *Resilient Backpropagation*

Let's look at it geometrically



In simple words, with Rprop

we keep on increasing until we pass the minimum

Rprop: *Resilient Backpropagation*

+ *How does it extend to multi-dimensional case that we have in training?*
– We apply this idea on every entry individually

Rprop: *Resilient Backpropagation*

+ *How does it extend to multi-dimensional case that we have in training?*
– We apply this idea on every entry individually

Initiate η and choose μ^+, μ^-, $\eta_{\max}$ and $\eta_{\min}$
for $t = 1, \ldots$ **do**
  ...
  Compute gradient $\mathbf{g}^{(t)}$
  for every entry i of $\mathbf{g}^{(t)}$ **do**
    Use the sign of entry and update $\eta_i^{(t)}$ via `Rprop_Scheduler()`
    Apply one-dimensional `Rprop()` to update entry i
  end for
  ...
end for

Rprop: *Resilient Backpropagation*

Typical choices of parameters for this algorithm are

- initial learning rate $\eta = 0.01$
- factors $\mu^+ = 1.2$ and $\mu^- = 0.5$
- $\eta_{\max}$ and $\eta_{\min}$ are less important: *they get automatically regulated*

It's better to avoid choices that $1/\mu^+ = \mu^-$, because if we pass the minimum, we don't like to move exactly the previous point

---

We can again access Rprop through module `optim` *in PyTorch*

```
>> import torch

>> torch.optim.Rprop()
```

RMSprop: *Root Mean Square Propagation*

It turns out the Rprop only works fine with full-batch training

↳ *it's because, it ignores the magnitude of gradient*

To understand why this happens, consider the *following dummy example*

Consider a one-dimensional case, i.e., $\mathbf{w} = w$*: we break the full batch into 4 mini-batches and come up with the following derivatives calculated in each step of mini-batch SGD*

(1) $\leftrightsquigarrow 0.1$ *(2)* $\leftrightsquigarrow 0.1$ *(3)* $\leftrightsquigarrow 0.1$ *(4)* $\leftrightsquigarrow -0.5$

RMSprop: *Root Mean Square Propagation*

It turns out the Rprop only works fine with full-batch training

↳ *it's because, it ignores the magnitude of gradient*

To understand why this happens, consider the *following dummy example*

Consider a one-dimensional case, i.e., $\mathbf{w} = w$*: we break the full batch into 4 mini-batches and come up with the following derivatives calculated in each step of mini-batch SGD*

$$(1) \leftrightsquigarrow 0.1 \qquad (2) \leftrightsquigarrow 0.1 \qquad (3) \leftrightsquigarrow 0.1 \qquad (4) \leftrightsquigarrow -0.5$$

We may approximately say that the derivative of full-batch risk, at the very first choice of w*, was close to zero or negative;*

RMSprop: *Root Mean Square Propagation*

It turns out the Rprop only works fine with full-batch training

↳ *it's because, it ignores the magnitude of gradient*

To understand why this happens, consider the *following dummy example*

Consider a one-dimensional case, i.e., $\mathbf{w} = w$*: we break the full batch into 4 mini-batches and come up with the following derivatives calculated in each step of mini-batch SGD*

$$(1) \leftrightsquigarrow 0.1 \qquad (2) \leftrightsquigarrow 0.1 \qquad (3) \leftrightsquigarrow 0.1 \qquad (4) \leftrightsquigarrow -0.5$$

We may approximately say that the derivative of full-batch risk, at the very first choice of w*, was close to zero or negative; however, Rprop*

1. *takes first three steps with larger and larger learning rates*
2. *only comes back with smaller step at last iteration*

RMSprop: *Root Mean Square Propagation*

It turns out the Rprop only works fine with full-batch training

↳ *it's because, it ignores the magnitude of gradient*

To understand why this happens, consider the *following dummy example*

Consider a one-dimensional case, i.e., $\mathbf{w} = w$*: we break the full batch into 4 mini-batches and come up with the following derivatives calculated in each step of mini-batch SGD*

$$(1) \leftrightsquigarrow 0.1 \qquad (2) \leftrightsquigarrow 0.1 \qquad (3) \leftrightsquigarrow 0.1 \qquad (4) \leftrightsquigarrow -0.5$$

We may approximately say that the derivative of full-batch risk, at the very first choice of w*, was close to zero or negative; however, Rprop*

1. *takes first three steps with larger and larger learning rates*
2. *only comes back with smaller step at last iteration*

It could be hence already lost at the last iteration!

RMSprop: *Root Mean Square Propagation*

Geoffrey Hinton in his *lecture notes*[5] came up with a solution: *we can use the idea of moving average to further normalize the learning rate according to average gradient magnitude*

this way we do not completely ignore the magnitude of gradient

[5] Click to check out the lecture notes!

RMSprop: *Root Mean Square Propagation*

Geoffrey Hinton in his *lecture notes*[5] came up with a solution: *we can use the idea of moving average to further normalize the learning rate according to average gradient magnitude*

this way we do not completely ignore the magnitude of gradient

---

Hinton looks differently at update rule of Rprop: recall the update rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \boldsymbol{\eta}^{(t)} \odot \text{sign}(\mathbf{g}^{(t)})$$

We can write alternatively as

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \boldsymbol{\eta}^{(t)} \odot \frac{\mathbf{g}^{(t)}}{|\mathbf{g}^{(t)}|}$$

where $|\cdot|$ operates entry-wise

[5] Click to check out the lecture notes!

RMSprop: *Root Mean Square Propagation*

Geoffrey Hinton in his *lecture notes*[5] came up with a solution: *we can use the idea of moving average to further normalize the learning rate according to average gradient magnitude*

this way we do not completely ignore the magnitude of gradient

---

Hinton looks differently at update rule of Rprop: recall the update rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \boldsymbol{\eta}^{(t)} \odot \text{sign}(\mathbf{g}^{(t)})$$

We can write alternatively as

learning rates are updated entry-wise

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \boldsymbol{\eta}^{(t)} \odot \frac{\mathbf{g}^{(t)}}{|\mathbf{g}^{(t)}|}$$

where $|\cdot|$ operates entry-wise

[5] Click to check out the lecture notes!

RMSprop: *Root Mean Square Propagation*

Hinton suggests that *we replace the denominator with*

moving average of root mean square of the gradients

This mean: *starting with some* $\mathbf{v}^{(0)}$ *we determine in iteration* t

$$\mathbf{v}^{(t)} = \beta \mathbf{v}^{(t-1)} + (1-\beta) \left|\mathbf{g}^{(t)}\right|^2$$

for some $\beta < 1$ *and normalize* $\mathbf{g}^{(t)}$ *with* $\sqrt{\mathbf{v}^{(t)}}$

Initiate $\mathbf{v}^{(0)}$
for $t = 1, \ldots$ **do**
  $\ldots$
  $\mathbf{v}^{(t)} = \beta \mathbf{v}^{(t-1)} + (1-\beta) \left|\mathbf{g}^{(t)}\right|^2$ `# compute moving average`
  $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{\mathbf{g}^{(t)}}{\sqrt{\mathbf{v}^{(t)}}}$ `# normalize by RMS`
  $\ldots$
end for

RMSprop: *Root Mean Square Propagation*

We may note that RMSprop can be observed as

SGD with Rprop-inspired learning rate scheduling

Note that the exact form of RMSprop has more details!

In typical implementations of RMSprop, we set

- learning rate to a some constant: *typical choice* $\eta = 0.01$
- β to be close to 1: *typical choice* $\beta > 0.9$

The complete form of RMSprop is also available in module `optim` *of PyTorch*

```
>> import torch

>> torch.optim.RMSprop()
```

Adaptive Momentum Estimation

Most recent implementations use the optimizer

Adaptive Momentum Estimation: Adam

that was proposed by *Kingma and Ba in 2015*[6]

[6] Click to check out the original paper!

Adaptive Momentum Estimation

Most recent implementations use the optimizer

Adaptive Momentum Estimation: Adam

that was proposed by *Kingma and Ba in 2015*[6]

The idea of Adam is straightforward: *it combines RMSprop with momentum*

it combines the strength of both approaches

In simple words: *Adam suggests that we use*

- *momentum for updating the weights*
- *Rprop-inspired approach for normalization* $\equiv$ *scheduling*

[6] Click to check out the original paper!

Adaptive Momentum Estimation

We can think of Adam as below

Initiate $\mathbf{m}^{(0)}$ and $\mathbf{v}^{(0)}$
for $t = 1, \ldots$ **do**
$\ldots$
$\mathbf{m}^{(t)} = \beta_1 \mathbf{m}^{(t-1)} + (1 - \beta_1)\, \mathbf{g}^{(t)}$ # compute momentum
$\mathbf{v}^{(t)} = \beta_2 \mathbf{v}^{(t-1)} + (1 - \beta_2)\, |\mathbf{g}^{(t)}|^2$ # compute RMS
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \dfrac{\mathbf{m}^{(t)}}{\sqrt{\mathbf{v}^{(t)}}}$ # move with normalized momentum
$\ldots$
end for

Adaptive Momentum Estimation

We can think of Adam as below

Initiate $\mathbf{m}^{(0)}$ and $\mathbf{v}^{(0)}$
for $t = 1, \ldots$ **do**
$\ldots$
$\mathbf{m}^{(t)} = \beta_1 \mathbf{m}^{(t-1)} + (1 - \beta_1)\, \mathbf{g}^{(t)}$ # compute momentum
$\mathbf{v}^{(t)} = \beta_2 \mathbf{v}^{(t-1)} + (1 - \beta_2)\, |\mathbf{g}^{(t)}|^2$ # compute RMS
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \dfrac{\mathbf{m}^{(t)}}{\sqrt{\mathbf{v}^{(t)}}}$ # move with normalized momentum
$\ldots$
end for

Attention

Pseudo codes in this lecture give the main ideas: there are further details and numerical tricks to make these algorithms robust and efficient in practice

Adaptive Momentum Estimation

In typical implementations of Adam, we set

- learning rate to a some constant: *typical choice* $0.001 < \eta < 0.01$
- β_1 to be close to 1: *typical choice* $\beta_1 = 0.9$
- β_2 to be closer to 1: *typical choice* $\beta_2 = 0.99$

Just check out the PyTorch implementation in the module `optim`

```
>> import torch

>> torch.optim.Adam()
```

Other Optimizers: *First Order vs Second Order*

There is a long list of modified gradient descents

Press Tab after typing `torch.optim.` *to see how long it is!*

In some particular applications, we may need to learn a new one:

Other Optimizers: *First Order vs Second Order*

There is a long list of modified gradient descents

Press Tab after typing `torch.optim.` *to see how long it is!*

In some particular applications, we may need to learn a new one: *it is hence good to know these two terms*

- First-order optimizers *that use only gradient, i.e., first-order derivatives*
  - ↳ *What we had in this section were all first-order*

Other Optimizers: *First Order vs Second Order*

There is a long list of modified gradient descents

Press Tab after typing `torch.optim.` *to see how long it is!*

In some particular applications, we may need to learn a new one: *it is hence good to know these two terms*

- First-order optimizers *that use only gradient, i.e., first-order derivatives*
  - ↳ *What we had in this section were all first-order*
- Second-order optimizers *also use Hessian, i.e., second-order derivatives*
  - ↳ These approaches are inspired by Newton's method *that shows convergence of gradient descent is boosted if we multiply gradient with inverse of Hessian*
  - ↳ They have typically *better convergence behavior*
  - ↳ Finding Hessian is a huge computation: *practical algorithms usually approximate Hessian; but, they still need high computation*

Tuning Hyperparameters

A problem that we intuitively discussed but *left open in Chapter 1* was

How can we tune the hyperparameters of a model?

Tuning Hyperparameters

A problem that we intuitively discussed but *left open in Chapter 1* was

How can we tune the hyperparameters of a model?

In this section, we are going to *get the genie out of the bottle!*

+ *Shouldn't we set hyperparameters as much as we could?! Make NNs as deep and wide as our computer let?*
– Not really! We don't need very deep and wide NNs always

Tuning Hyperparameters

A problem that we intuitively discussed but *left open in Chapter 1* was

How can we tune the hyperparameters of a model?

In this section, we are going to *get the genie out of the bottle!*

+ *Shouldn't we set hyperparameters as much as we could?! Make NNs as deep and wide as our computer let?*
– Not really! We don't need very deep and wide NNs always
+ *But, what is we are computationally strong?! Then, we are fine! Right?!*
– No! In fact with large NNs we can do the so-called *overfitting!*

Tuning Hyperparameters

A problem that we intuitively discussed but *left open in Chapter 1* was

How can we tune the hyperparameters of a model?

In this section, we are going to *get the genie out of the bottle!*

+ *Shouldn't we set hyperparameters as much as we could?! Make NNs as deep and wide as our computer let?*
- Not really! We don't need very deep and wide NNs always
+ *But, what is we are computationally strong?! Then, we are fine! Right?!*
- No! In fact with large NNs we can do the so-called *overfitting!*

Let's see a very simple example!

Example: *Fitting Polynomial from Noisy Data*

We start by a *classical example* which is *not that of NN* we expect

We have a machine which gets real-valued x *and returns*

$$y = x^2 + 3x + 3$$

We however don't know this relation: the only thing that we know is that the inputs and labels are related via a polynomial

We invoke ML to learn this machine

Example: *Fitting Polynomial from Noisy Data*

We start by a *classical example* which is *not that of NN* we expect

We have a machine which gets real-valued x and returns

$$y = x^2 + 3x + 3$$

We however don't know this relation: the only thing that we know is that the inputs and labels are related via a polynomial

We invoke ML to learn this machine

Let's start with making the ML components, i.e.,

1. Dataset
2. Model
3. Loss

Example: *Polynomial Fitting - Dataset*

We start by collecting data: *we give input x_b to this machine and measure its output for a batch of inputs*.

Example: *Polynomial Fitting - Dataset*

We start by collecting data: *we give input x_b to this machine and measure its output for a batch of inputs. Our measurements are however noisy, i.e.,*

$$v_b = {x_b}^2 + 3x_b + 3 + \varepsilon_b$$

where ε_b is noise with bounded magnitude, i.e., $|\varepsilon_b| \leqslant \alpha$ for some constant α

Example: *Polynomial Fitting - Dataset*

We start by collecting data: *we give input x_b to this machine and measure its output for a batch of inputs. Our measurements are however noisy, i.e.,*

$$v_b = {x_b}^2 + 3x_b + 3 + \varepsilon_b$$

where ε_b is noise with bounded magnitude, i.e., $|\varepsilon_b| \leqslant \alpha$ for some constant α

We make our dataset as

$$\mathbb{D} = \{(x_b, v_b) : i = 1, \ldots, B\}$$

Example: *Polynomial Fitting - Model*

We know that machine is polynomial: *we assume a polynomial model*

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_P x^P$$

for some integer order P

Example: *Polynomial Fitting - Model*

We know that machine is polynomial: *we assume a polynomial model*

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_P x^P$$

for some integer order P

We can write it down as

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_P x^P$$

$$= \underbrace{\begin{bmatrix} w_0 & w_1 & w_2 & \ldots & w_P \end{bmatrix}}_{\mathbf{w}^\mathsf{T}} \underbrace{\begin{bmatrix} x^0 \\ x^1 \\ x^2 \\ \ldots \\ x^P \end{bmatrix}}_{\mathbf{h}}$$

Example: *Polynomial Fitting - Model*

We know that machine is polynomial: *we assume a polynomial model*

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_P x^P$$

for some integer order P

We can write it down as

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_P x^P$$

$$= \underbrace{\begin{bmatrix} w_0 & w_1 & w_2 & \ldots & w_P \end{bmatrix}}_{\mathbf{w}^\mathsf{T}} \underbrace{\begin{bmatrix} x^0 \\ x^1 \\ x^2 \\ \ldots \\ x^P \end{bmatrix}}_{\mathbf{h}} = \mathbf{w}^\mathsf{T}\mathbf{h}$$

Example: *Polynomial Fitting - Model*

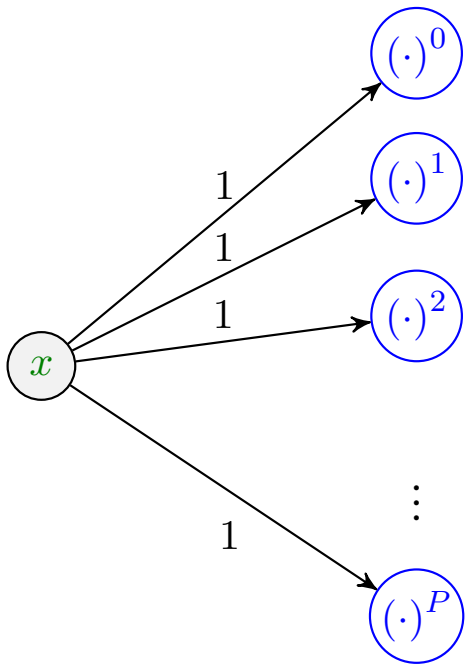
We can look at our model as *an NN with dummy neurons*

$\mathbf{h}$ *is what we get from hidden layer and* $\mathbf{w}$ *includes weights of output layer*

Example: *Polynomial Fitting - Model*

We can look at our model as *an NN with dummy neurons*

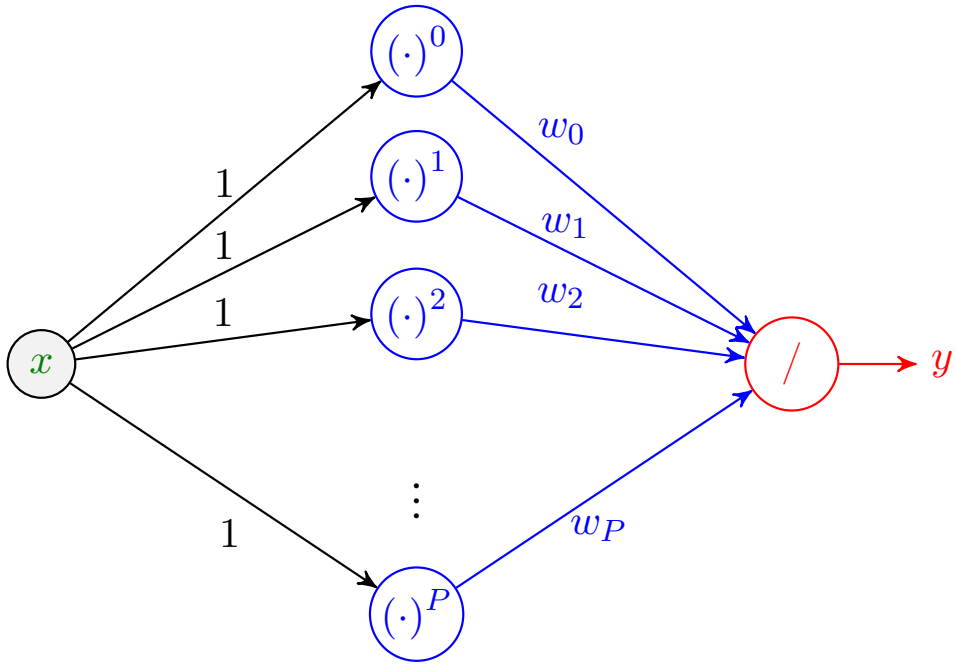
$\mathbf{h}$ *is what we get from hidden layer and* $\mathbf{w}$ *includes weights of output layer*



Example: *Polynomial Fitting - Model*

We can look at our model as *an NN with dummy neurons*

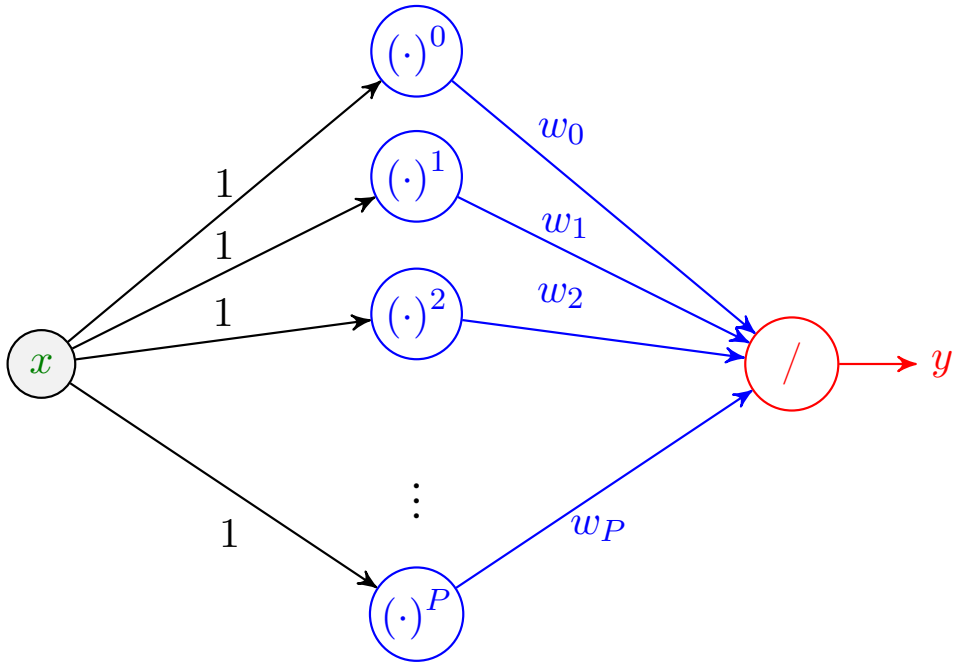
$\mathbf{h}$ *is what we get from hidden layer and* $\mathbf{w}$ *includes weights of output layer*



Example: *Polynomial Fitting - Model*

We can look at our model as *an NN with dummy neurons*

$\mathbf{h}$ *is what we get from hidden layer and* $\mathbf{w}$ *includes weights of output layer*



The key hyperparameter in this network is P

Example: *Polynomial Fitting - Loss*

We have a regression problem: *recall that*

in regression, the labels are real-valued

Example: *Polynomial Fitting - Loss*

We have a regression problem: *recall that*

in regression, the labels are real-valued

We use *squared error as the loss function, i.e.,*

$$\mathcal{L}(y, v) = (y - v)^2$$

for measurement v *and NN's output* y

Example: *Polynomial Fitting - Loss*

We have a regression problem: *recall that*

in regression, the labels are real-valued

We use *squared error as the loss function, i.e.,*

$$\mathcal{L}(y, v) = (y - v)^2$$

for measurement v and NN's output y

Now, the components are ready

let's start training

Example: *Polynomial Fitting - Training*

For training, we follow what *we already learned in previous lectures*

1. *We split $\mathbb{D}$ into a training dataset and test dataset*
2. *We start use gradient descent to train over the training dataset*
3. *We test our trained model over the test dataset*

Example: *Polynomial Fitting - Training*

For training, we follow what *we already learned in previous lectures*

1. *We split $\mathbb{D}$ into a training dataset and test dataset*
2. *We start use gradient descent to train over the training dataset*
3. *We test our trained model over the test dataset*

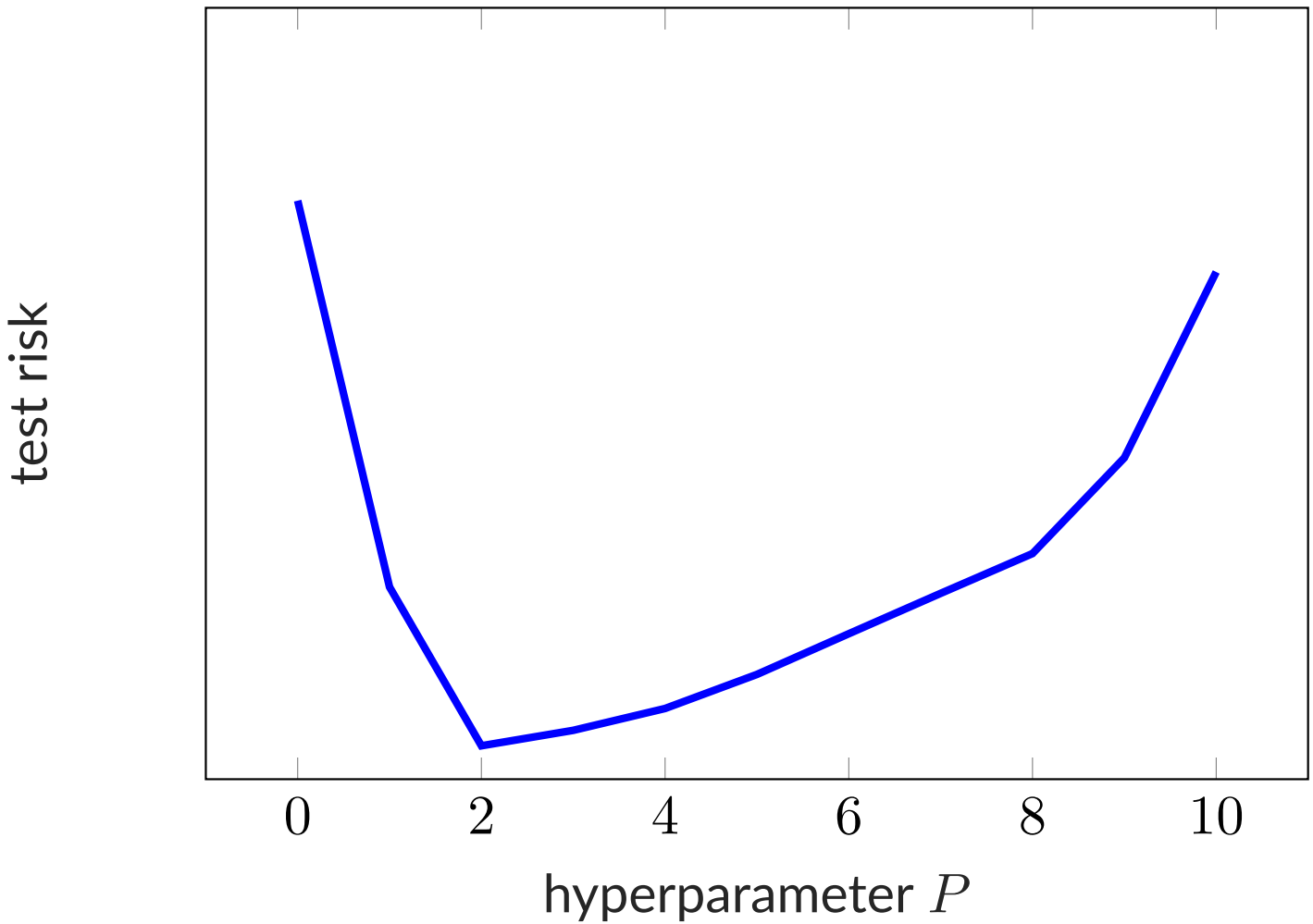
Before we get go on with training, let's take a look back

> *Naive conclusion was that making the NN large is always good; if so*
>
> *when we increase P, we should always see lowertest risk*
>
> *However, it's not the case!*

Let's see *how the test risk changes against hyperparameter P*

Example: *Polynomial Fitting*

Test risk against the hyperparameter P looks like the curve below!



Over and Underfitted Model

+ *What is happening here?*
- As we pass $P = 2$, we are overfitting!

Over and Underfitted Model

+ *What is happening here?*
- As we pass $P = 2$, we are overfitting!

Overfitting

Overfitting occurs when training fits the model, i.e., NN, to the training dataset, so that it does not generalize to new data-points

Over and Underfitted Model

+ *What is happening here?*
- As we pass $P = 2$, we are overfitting!

Overfitting

Overfitting occurs when training fits the model, i.e., NN, to the training dataset, so that it does not generalize to new data-points

We may also pay attention to the term *generalize* in this definition

Generalization

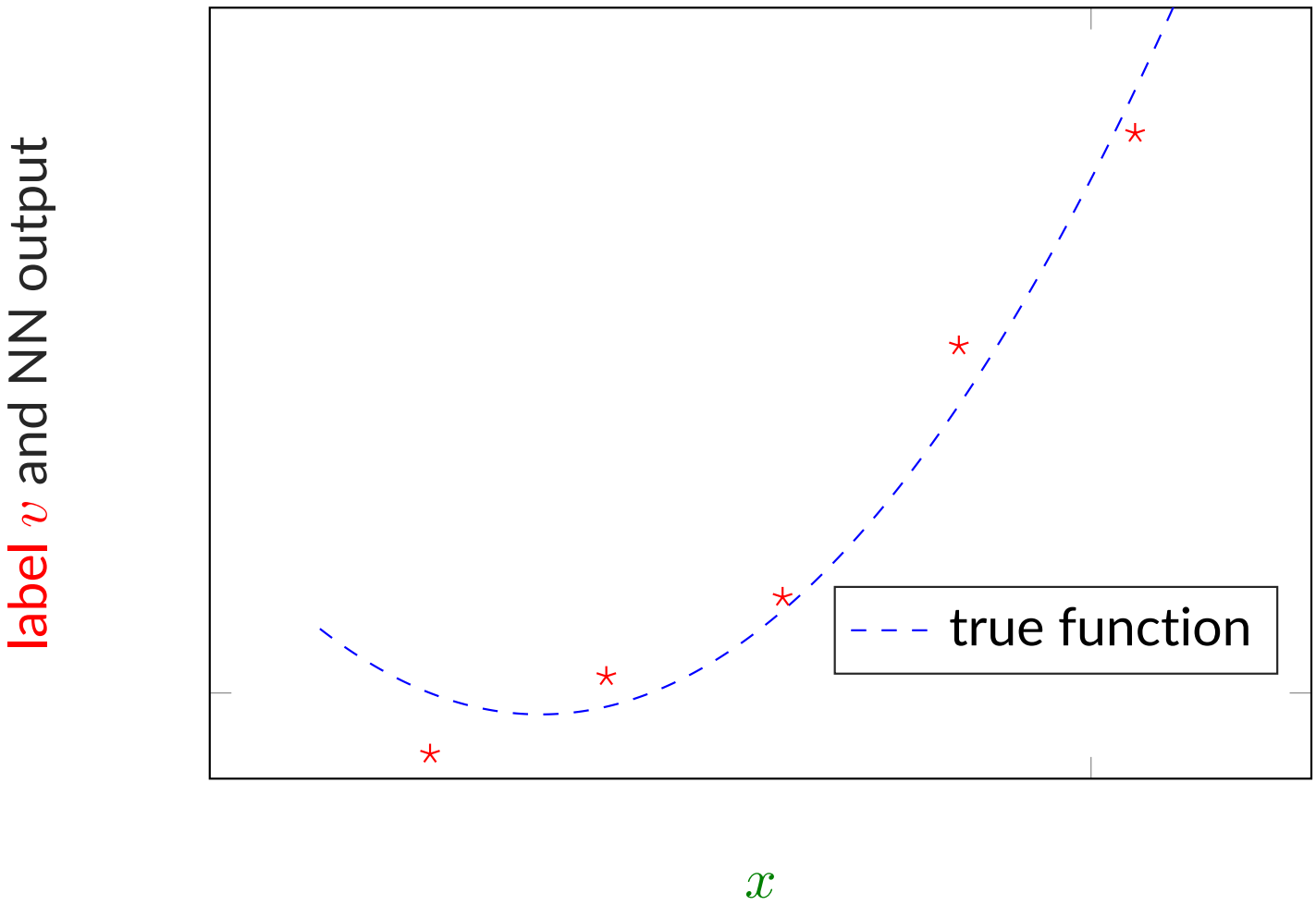
We say a model, i.e., NN, generalizes well if not only its empirical risk, but also its test risk is small

In simple words:

trained NN generalizes $\equiv$ *it does what we want on new data*

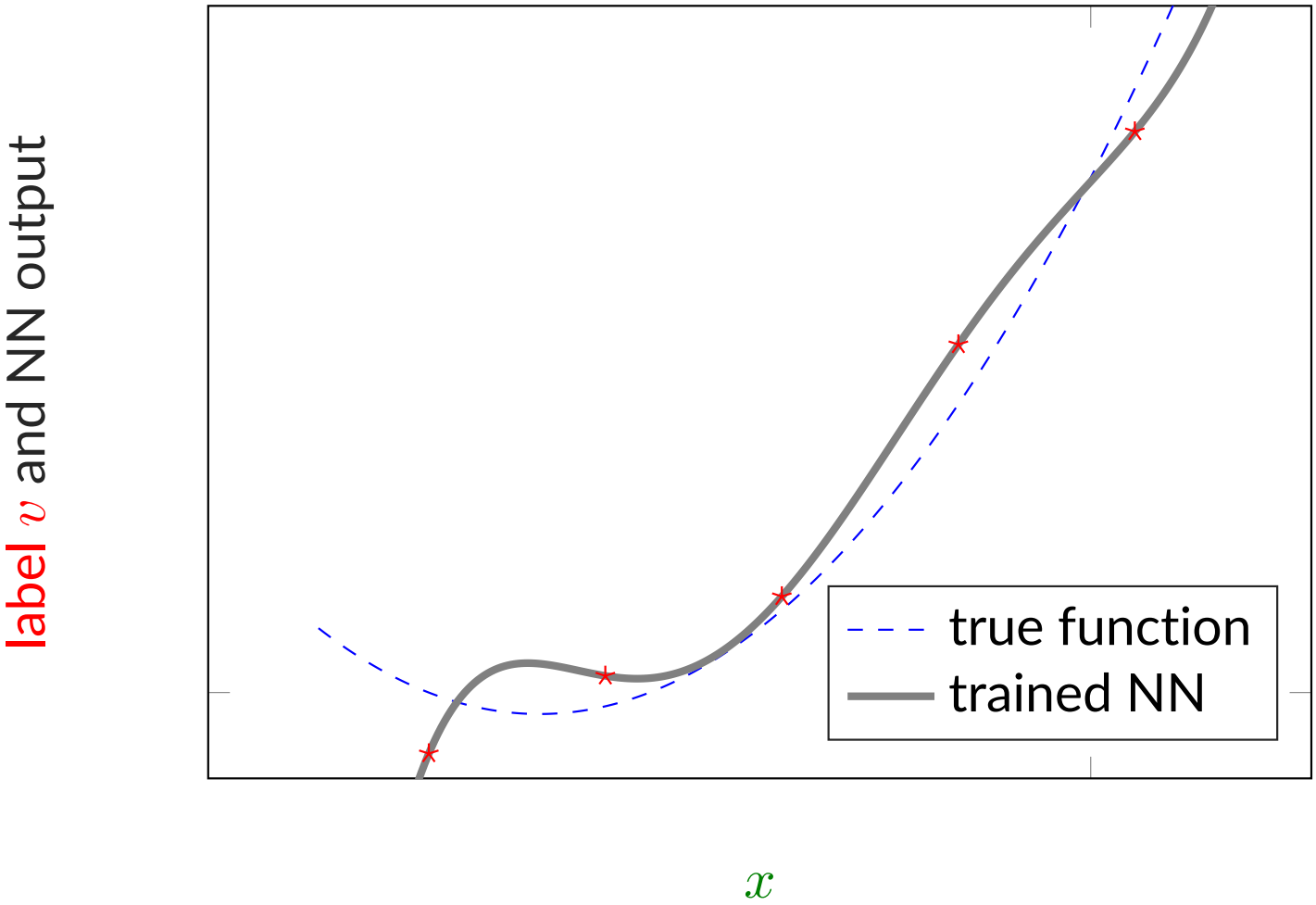
Overfitting: *Polynomial Fitting*

Let's take a look back on polynomial fitting example: *for large P the NN fits very well to the training data, but it deviates greatly from the true function*



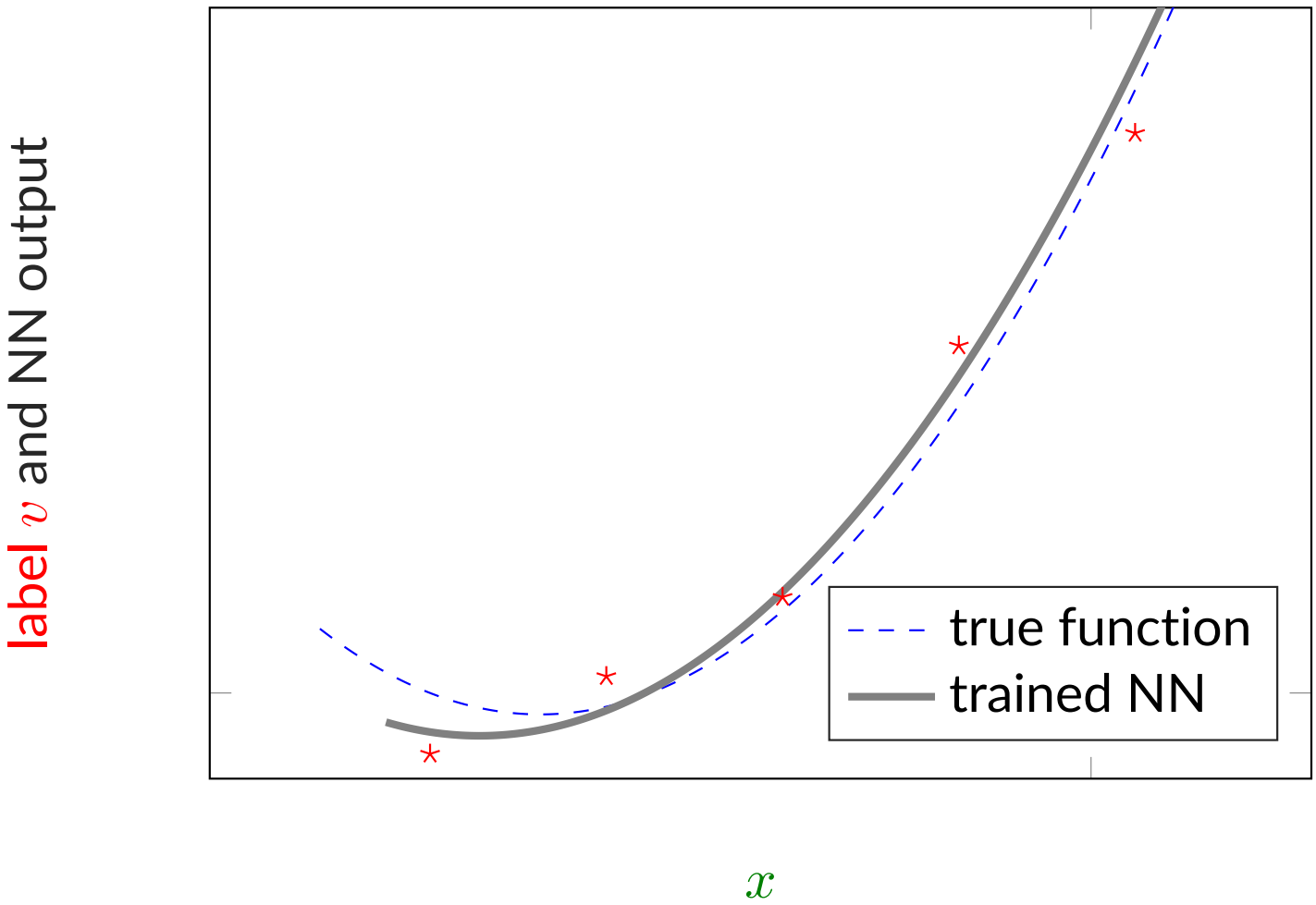
Overfitting: *Polynomial Fitting*

Let's take a look back on polynomial fitting example: *for large P the NN fits very well to the training data, but it deviates greatly from the true function*



Overfitting: *Polynomial Fitting*

We can see the importance of hyperparameter tuning: *if we set P to a right choice; then, our NN generalizes well, i.e., it closely track the true function*



Underfitting

The other side of the coin is *underfitting*: *it happens when our NN does not have enough parameters to train*

Underfitting

The other side of the coin is *underfitting*: *it happens when our NN does not have enough parameters to train*

Underfitting

Underfitting occurs when the model, i.e., NN, neither fits to the training dataset, nor generalizes to new data

Underfitting

The other side of the coin is *underfitting*: *it happens when our NN does not have enough parameters to train*

Underfitting

Underfitting occurs when the model, i.e., NN, neither fits to the training dataset, nor generalizes to new data

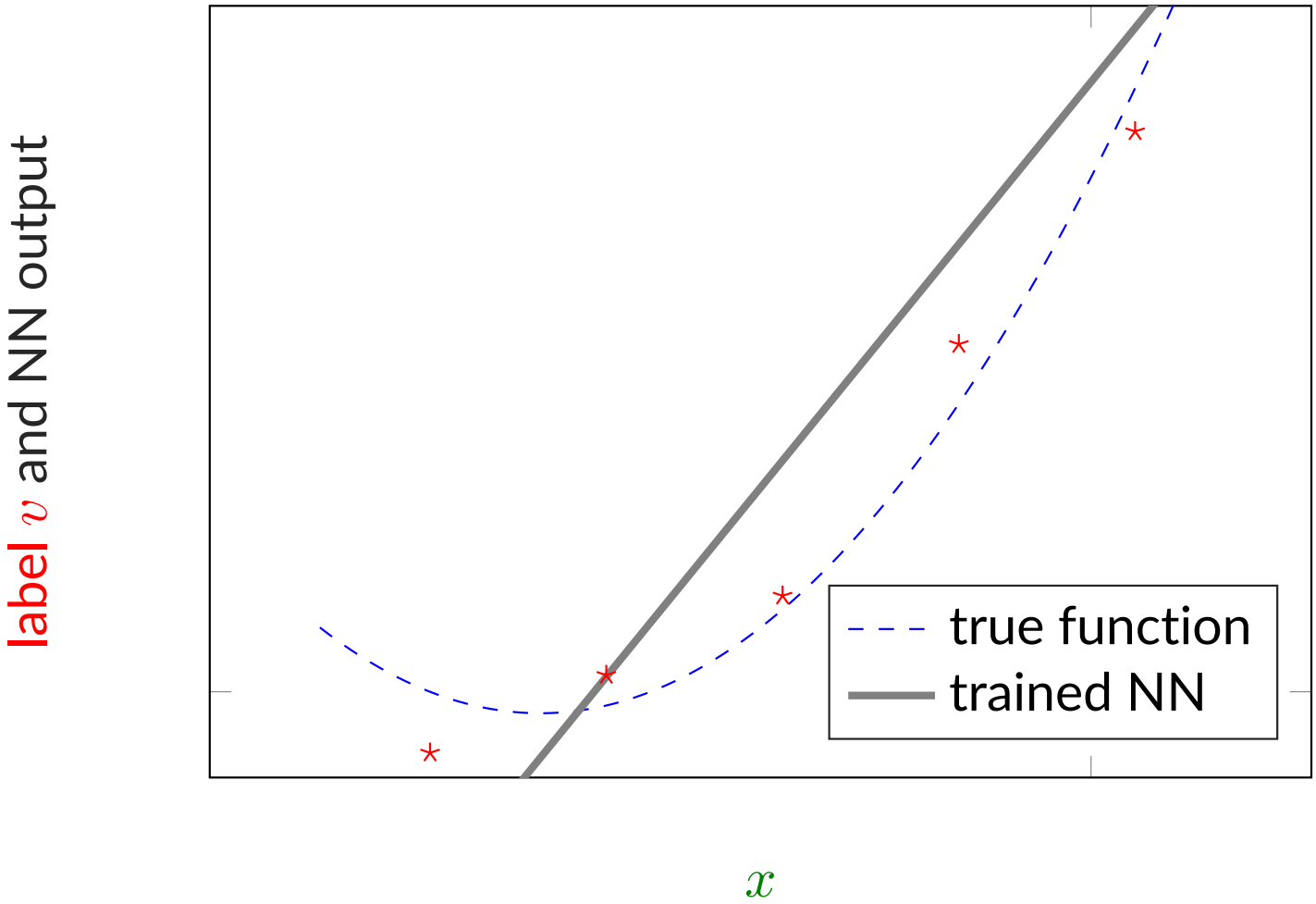
Se would also need to prevent underfitting; however,

with current NNs, underfitting can hardly occur

This is why *it's less discussed in the literature*

Underfitting: *Polynomial Fitting*

A linear model underfits in our example: *setting $P = 1$ will lead to a line that can never fit our training dataset*



Validation: *First Step Against Overfitting*

+ *What is connection to our main task, i.e., hyperparameter tuning?*
– Well! Before everything, we need to tune the hyperparameters right to avoid over or underfitting

Validation: *First Step Against Overfitting*

+ *What is connection to our main task, i.e., hyperparameter tuning?*
– Well! Before everything, we need to tune the hyperparameters right to avoid over or underfitting

---

Hyperparameter tuning is done by *validation*: *we change hyperparameters among possible choices and for each choice, we validate our model*

> *By validation we mean that we train the NN with the specified hyperparameters and then test it*

Validation: *First Step Against Overfitting*

+ *What is connection to our main task, i.e., hyperparameter tuning?*
– Well! Before everything, we need to tune the hyperparameters right to avoid over or underfitting

---

Hyperparameter tuning is done by *validation*: *we change hyperparameters among possible choices and for each choice, we validate our model*

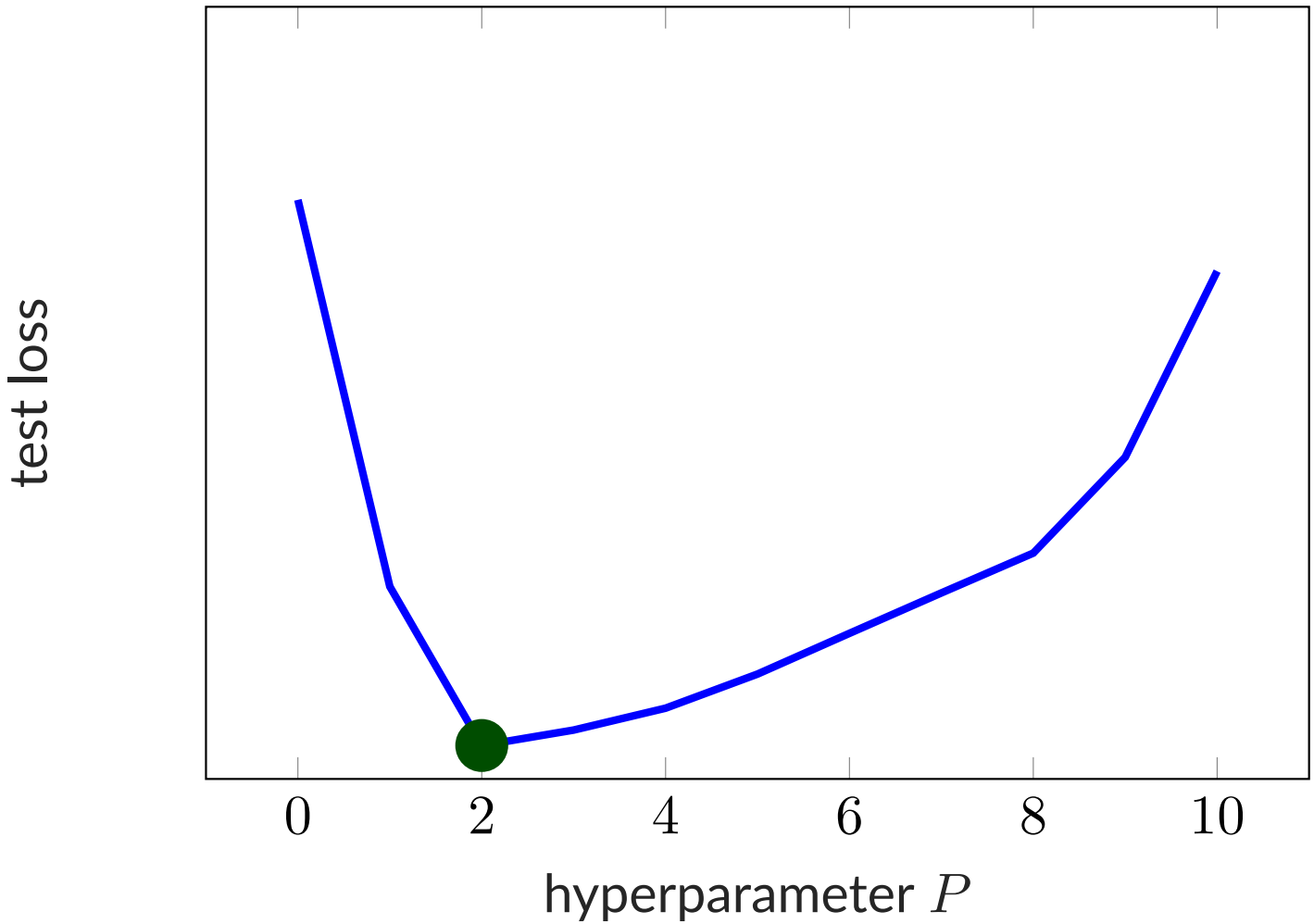
> *By validation we mean that we train the NN with the specified hyperparameters and then test it*

We set hyperparameters to the choice that

$$\textit{gives minimal test risk} \equiv \textit{generalizes the best}$$

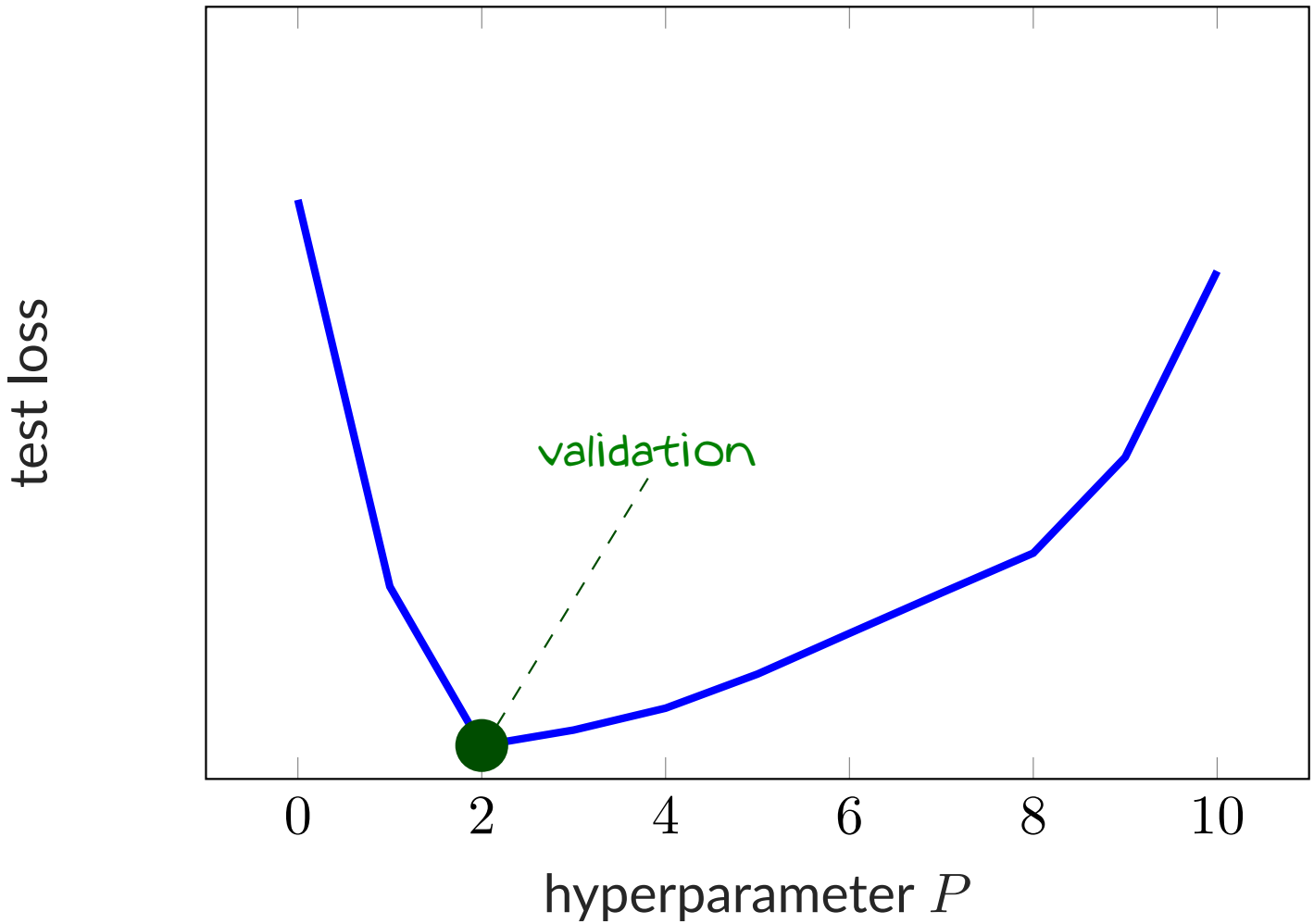
Validation: *Polynomial Fitting*

In fact what we did in our dummy example was validation



Validation: *Polynomial Fitting*

In fact what we did in our dummy example was validation



Why Overfitting Happens

+ *But can we really do hyperparameter tuning in a very deep NN?*
– Not really! We may tune some general hyperparameters like number of layers, but *cannot really do a complete validation*

Why Overfitting Happens

+ *But can we really do hyperparameter tuning in a very deep NN?*
– Not really! We may tune some general hyperparameters like number of layers, but *cannot really do a complete validation*

In NNs, we invoke other approaches as well to combat overfitting

- *Regularization*
- *Dropout*
- *Data Augmentation*
- . . .

Why Overfitting Happens

+ *But can we really do hyperparameter tuning in a very deep NN?*
– Not really! We may tune some general hyperparameters like number of layers, but *cannot really do a complete validation*

In NNs, we invoke other approaches as well to combat overfitting

- *Regularization*
- *Dropout*
- *Data Augmentation*
- . . .

To understand these approaches, we should first answer the following question

When does overfitting happen in a NN?

Let's take a look!

Why Overfitting Happens: *Model Capacity*

We know the initial answer: *in our dummy example, it happened because*

we assumed large polynomial order

In other words

our model was too complex for our learning task

Why Overfitting Happens: *Model Capacity*

We know the initial answer: *in our dummy example, it happened because*

we assumed large polynomial order

In other words

our model was too complex for our learning task

We can extend this to NNs: *overfitting happens when the model is too complex, i.e., it's suited for learning complicated functions*

When does overfitting happen in a NN? *It happens when*

1. *the model has a large capacity*

Why Overfitting Happens: *Model Capacity*

We know the initial answer: *in our dummy example, it happened because*

we assumed large polynomial order

In other words

our model was too complex for our learning task

We can extend this to NNs: *overfitting happens when the model is too complex, i.e., it's suited for learning complicated functions*

When does overfitting happen in a NN? *It happens when*

1. *the model has a large capacity*

Though *model capacity* has a concrete definition, for our purpose

model capacity $\equiv$ *ability of model to learn different functions*

Why Overfitting Happens: *Dataset Size*

+ *But how can we find this out? It does not seem to be easy!*
– That's right! This is why we look into *other reasons as well*

Why Overfitting Happens: *Dataset Size*

+ *But how can we find this out? It does not seem to be easy!*
– That's right! This is why we look into *other reasons as well*

Let's get back to our dummy example: *this time we check it a bit differently*

In our polynomial example, we consider an overfitted model with $P = 5$ *and train it on two randomly generated datasets*

1. *a dataset with 20 data-points*
2. *a dataset with only 4 data-points*

Why Overfitting Happens: *Dataset Size*

+ *But how can we find this out? It does not seem to be easy!*
– That's right! This is why we look into *other reasons as well*

Let's get back to our dummy example: *this time we check it a bit differently*

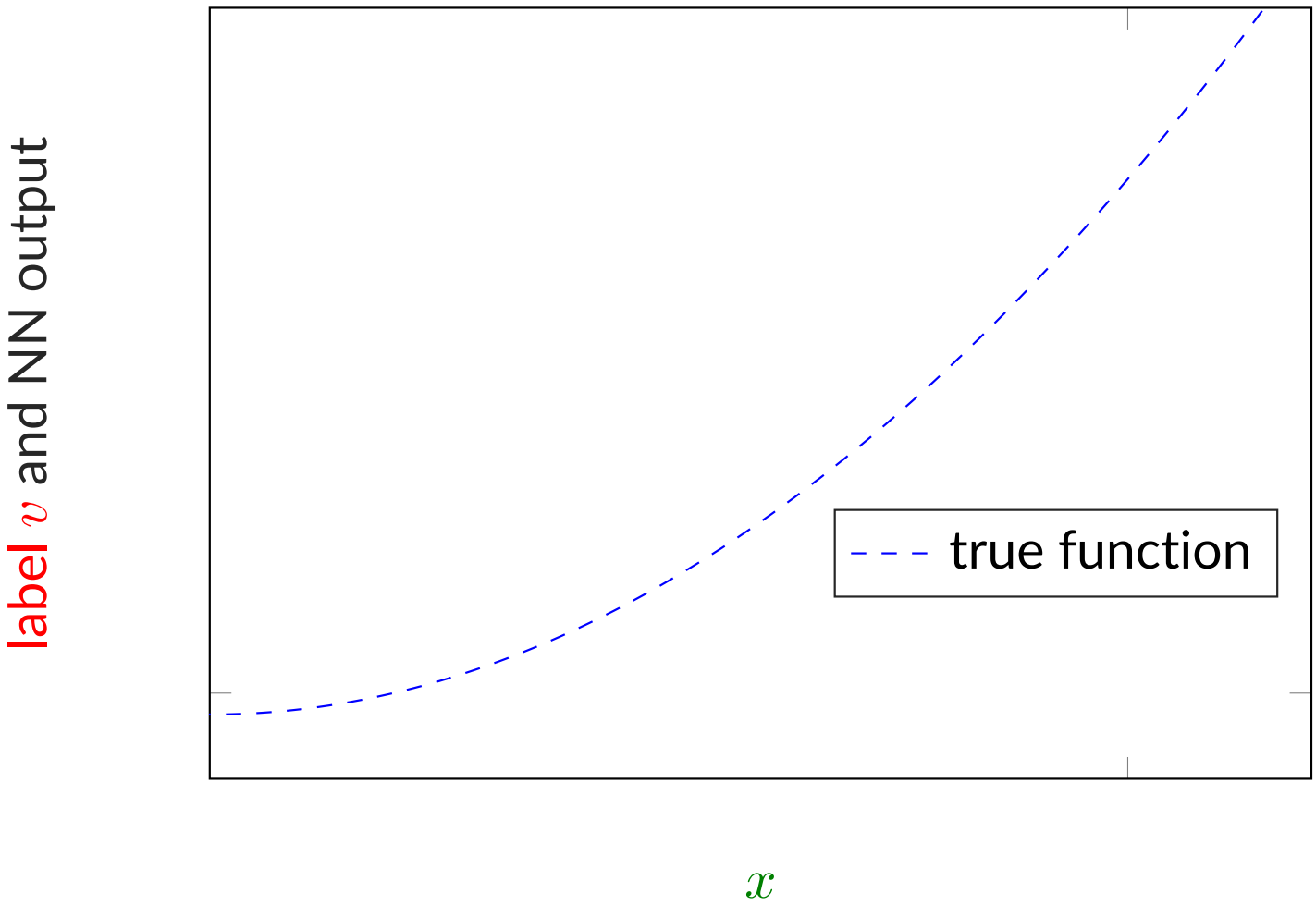
In our polynomial example, we consider an overfitted model with $P = 5$ *and train it on two randomly generated datasets*

1. *a dataset with 20 data-points*
2. *a dataset with only 4 data-points*

After training: *we compare trained models with the true function*

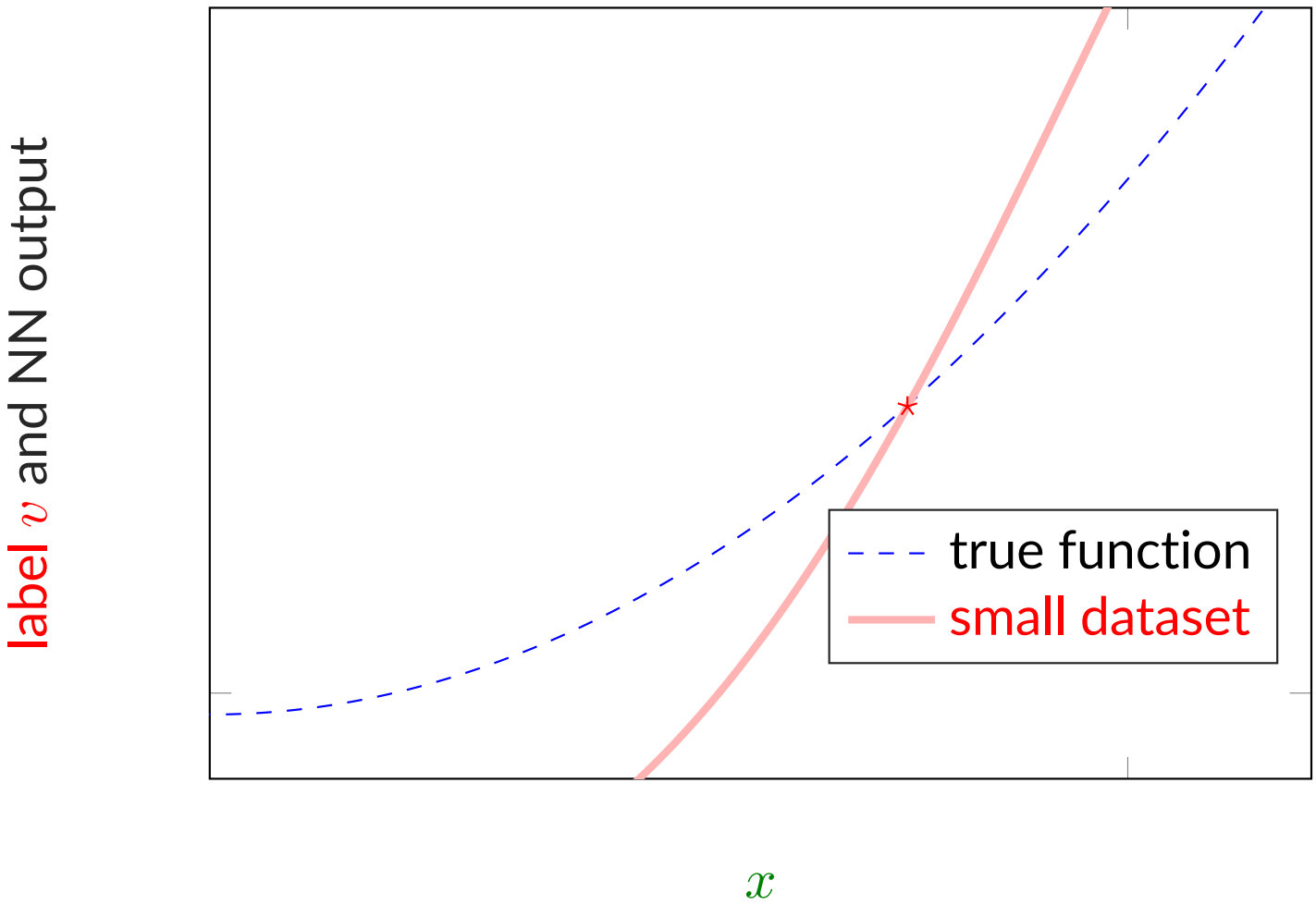
Why Overfitting Happens: *Dataset Size*

As we can see: *overfitting* is *reduced* as we *increase the number of data-points*



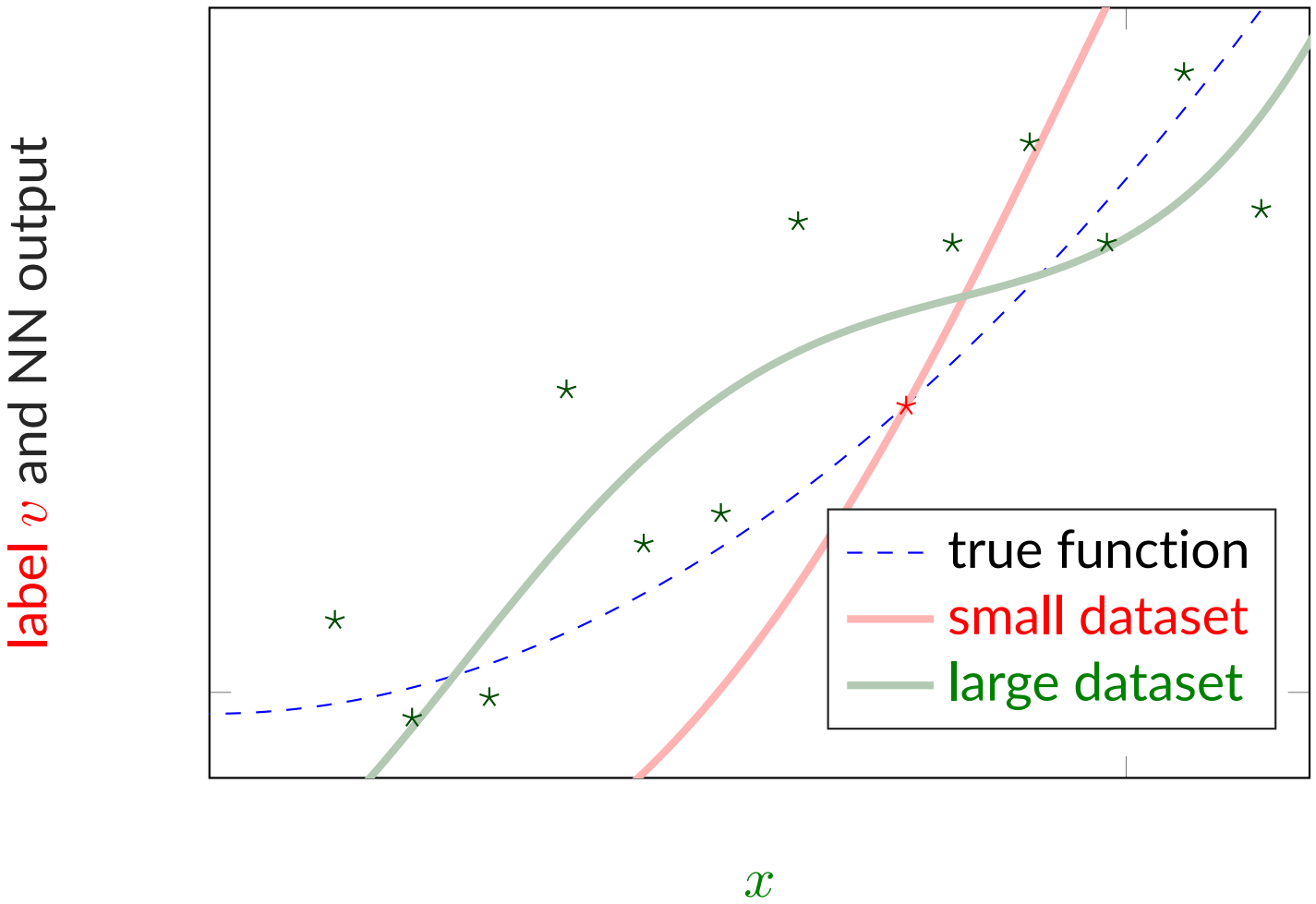
Why Overfitting Happens: *Dataset Size*

As we can see: *overfitting* is *reduced* as we *increase the number of data-points*



Why Overfitting Happens: *Dataset Size*

As we can see: *overfitting* is *reduced* as we *increase the number of data-points*



Why Overfitting Happens: *Dataset Size*

This is a general behavior: *if we have a large enough dataset the model cannot really overfits too much*

- \+ *How large it should be?*
- – It depends on the NN

Why Overfitting Happens: *Dataset Size*

This is a general behavior: *if we have a large enough dataset the model cannot really overfits too much*

- \+ *How large it should be?*
- – It depends on the NN

A general rule is that *the more learnable parameters the model has, the larger the training dataset should be*

Why Overfitting Happens: *Dataset Size*

This is a general behavior: *if we have a large enough dataset the model cannot really overfits too much*

- \+ *How large it should be?*
- – It depends on the NN

> A general rule is that *the more learnable parameters the model has, the larger the training dataset should be*

So, we can add to our answers

> When does overfitting happen in a NN? *It happens when*
>
> 1. *the model has a large capacity*
> 2. *our training dataset is small*

Why Overfitting Happens: *Co-Adaptation*

Another way to see overfitting is to look at *how model parameters change as optimizer iterates*. To see it, *let's get back to our dummy polynomial-fitting NN*

> Consider the following setting: *we have a high-capacity NN with* $P = 5$ *and dataset with 8 noisy samples. We train this NN using full-batch SGD*

We now take a look at the trained NN at different iterations: *recall that the vector of model parameters is*

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_5 \end{bmatrix}$$

We start with vector of all zeros and keep on going

Why Overfitting Happens: *Co-Adaptation*

Recall that *the ground truth* $\mathbf{w}^\star$ for our polynomial machine

$$\mathbf{w}^\star = \begin{bmatrix} 3 \\ 3 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Why Overfitting Happens: *Co-Adaptation*

Recall that *the ground truth* $\mathbf{w}^\star$ for our polynomial machine

$$\mathbf{w}^\star = \begin{bmatrix} 3 \\ 3 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Now, let's look at few iterations

$$\mathbf{w}^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightsquigarrow \mathbf{w}^{(t_1)} = \begin{bmatrix} 2.61 \\ 2.36 \\ 0.71 \\ 0.01 \\ 0.02 \\ 0.01 \end{bmatrix} \rightsquigarrow \mathbf{w}^{(t_2)} = \begin{bmatrix} 3.03 \\ 2.97 \\ 0.98 \\ 0.21 \\ 0.12 \\ 0.09 \end{bmatrix}$$

Why Overfitting Happens: *Co-Adaptation*

Recall that *the ground truth* $\mathbf{w}^\star$ for our polynomial machine

$$\mathbf{w}^\star = \begin{bmatrix} 3 \\ 3 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Now, let's look at few iterations

$$\mathbf{w}^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightsquigarrow \mathbf{w}^{(t_1)} = \begin{bmatrix} 2.61 \\ 2.36 \\ 0.71 \\ 0.01 \\ 0.02 \\ 0.01 \end{bmatrix} \rightsquigarrow \mathbf{w}^{(t_2)} = \begin{bmatrix} 3.03 \\ 2.97 \\ 0.98 \\ 0.21 \\ 0.12 \\ 0.09 \end{bmatrix}$$

$\mathbf{w}^{(t_2)}$ *looks good! But, what if we keep on training*

$$\mathbf{w}^{(t_2)} = \begin{bmatrix} 3.03 \\ 2.97 \\ 0.98 \\ 0.21 \\ 0.12 \\ 0.09 \end{bmatrix} \rightsquigarrow \cdots \rightsquigarrow \mathbf{w}^{(t_3)} = \begin{bmatrix} 2.36 \\ 4.43 \\ 3.13 \\ -2.1 \\ 1.98 \\ -1.2 \end{bmatrix}$$

Why Overfitting Happens: *Co-Adaptation*

Let's formulate what we observed

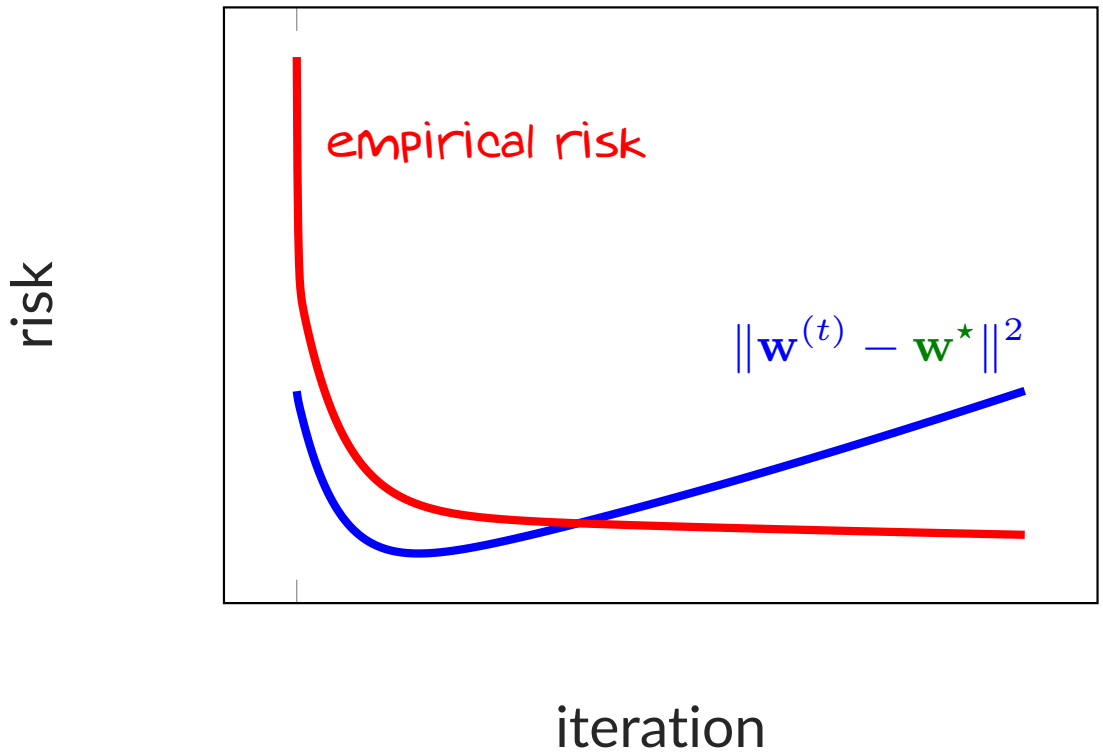
> *Weights start to get close to what we want up to intermediate number of iterations t_2. But, by further training they start to deviate $\equiv$ overfit*

Why Overfitting Happens: *Co-Adaptation*

Let's formulate what we observed

> *Weights start to get close to what we want up to intermediate number of iterations t_2. But, by further training they start to deviate $\equiv$ overfit*

We can also see this behavior in the figure below



Why Overfitting Happens: *Co-Adaptation*

This behavior is co-adaptation of the parameters

Why Overfitting Happens: *Co-Adaptation*

This behavior is co-adaptation of the parameters

In initial iterations, NN fits to the true model: *since data come from a quadratic function, the first iterations of SGD*

update majorly w_0, w_1 *and* w_2

Why Overfitting Happens: *Co-Adaptation*

This behavior is co-adaptation of the parameters

In initial iterations, NN fits to the true model: *since data come from a quadratic function, the first iterations of SGD*

update majorly w_0, w_1 *and* w_2

After NN has gone close to ground truth, it starts to overfit: *due to noise, quadratic model can't perfectly fit; thus,*

w_3, w_4 *and* w_5 *try to co-adapt, i.e., compensate the gap caused by noise*

Why Overfitting Happens: *Co-Adaptation*

This behavior is co-adaptation of the parameters

In initial iterations, NN fits to the true model: *since data come from a quadratic function, the first iterations of SGD*

update majorly w_0, w_1 *and* w_2

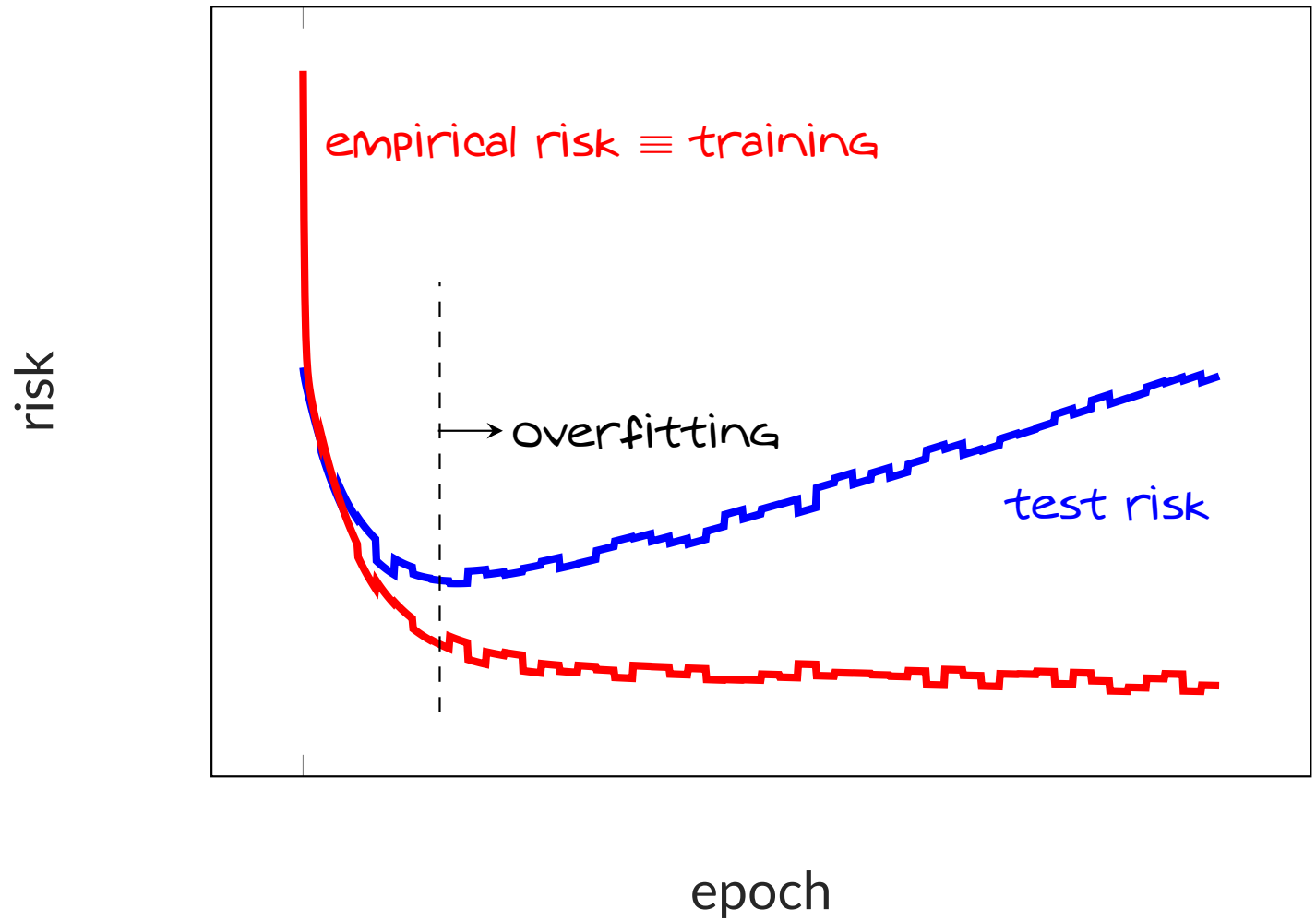
After NN has gone close to ground truth, it starts to overfit: *due to noise, quadratic model can't perfectly fit; thus,*

w_3, w_4 *and* w_5 *try to co-adapt, i.e., compensate the gap caused by noise*

In simple words: *in first iterations NN learns true function; however, at some point it starts to learn noise!*

Co-adaptation is the most observable source of overfitting with NNs

Co-Adaptation: *Typical Learning Curve*



Why Overfitting Happens: *Final List*

So, let's complete the answer list

When does overfitting happen in NNs? *It happens when*

1. *the model has a large capacity*
2. *our training dataset is small*
3. *due to large number of training iterations co-adaptation occurs*

Why Overfitting Happens: *Final List*

So, let's complete the answer list

When does overfitting happen in NNs? *It happens when*

1. *the model has a large capacity*
2. *our training dataset is small*
3. *due to large number of training iterations co-adaptation occurs*

Attention: *sources are mutually related*

If we have a very large model capacity, i.e., very deep with too many neurons; then, training it by a small dataset leads to overfitting, especially if we keep on training for too many epochs!

\+ *Now, can we do anything to avoid overfitting?*

– Yes! Depending on what we see as source, we use different tricks

Classical Solutions to Overfitting

Overfitting happens when

1. *the model has a large capacity*
2. *our training dataset is small*
3. *due to large number of training iterations co-adaptation occurs*

The key tricks to address overfitting in each of these cases are

1. *We can tune the hyperparameters to restrict the NN's capacity*
   - ↳ *For instance, we can validate our FNN with 2, 3 and 4 hidden layers and choose the model with minimal test risk*

Classical Solutions to Overfitting

Overfitting happens when

1. *the model has a large capacity*
2. *our training dataset is small*
3. *due to large number of training iterations co-adaptation occurs*

The key tricks to address overfitting in each of these cases are

1. *We can tune the hyperparameters to restrict the NN's capacity*
   - ↳ *For instance, we can validate our FNN with 2, 3 and 4 hidden layers and choose the model with minimal test risk*
2. *We can increase our dataset by the so-called data augmentation*
   - ↳ *For instance, we can add rotated and shifted versions of images inside the dataset with the same label: a rotated image of a dog is still a dog!*

Classical Solutions to Overfitting

Overfitting happens when

1. *the model has a large capacity*
2. *our training dataset is small*
3. *due to large number of training iterations co-adaptation occurs*

The key tricks to address overfitting in each of these cases are

1. *We can tune the hyperparameters to restrict the NN's capacity*
   - ↳ *For instance, we can validate our FNN with 2, 3 and 4 hidden layers and choose the model with minimal test risk*
2. *We can increase our dataset by the so-called data augmentation*
   - ↳ *For instance, we can add rotated and shifted versions of images inside the dataset with the same label: a rotated image of a dog is still a dog!*
3. *We can regularize our empirical risk to penalize co-adapted solutions*
   - ↳ *For instance, we can drop out randomly some neurons in each mini-batch*

Classical Solutions to Overfitting

Overfitting happens when

1. *the model has a large capacity*
2. *our training dataset is small*
3. *due to large number of training iterations co-adaptation occurs*

The key tricks to address overfitting in each of these cases are

Done

1. *We can tune the hyperparameters to restrict the NN's capacity*
   - ↳ *For instance, we can validate our FNN with 2, 3 and 4 hidden layers and choose the model with minimal test risk*

Wait

2. *We can increase our dataset by the so-called data augmentation*
   - ↳ *For instance, we can add rotated and shifted versions of images inside the dataset with the same label: a rotated image of a dog is still a dog!*

Next

3. *We can regularize our empirical risk to penalize co-adapted solutions*
   - ↳ *For instance, we can drop out randomly some neurons in each mini-batch*

Training by Penalized Risk: *Regularization*

Regularization aims to resolve overfitting by treating co-adaptation

Let's recall co-adaptation in our dummy polynomial fitting NN: *we set* $P = 5$ *and train our NN via the noisy samples inside training dataset; clearly,*

as training proceeds, empirical risk drops

In our particular example with

$$\mathbf{w}^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightsquigarrow \mathbf{w}^{(t_2)} = \begin{bmatrix} 3.03 \\ 2.97 \\ 0.98 \\ 0.21 \\ 0.12 \\ 0.09 \end{bmatrix} \rightsquigarrow \mathbf{w}^{(t_3)} = \begin{bmatrix} 2.36 \\ 4.43 \\ 3.13 \\ -2.1 \\ 1.98 \\ -1.2 \end{bmatrix}$$

This means that $\hat{R}(\mathbf{w}^{(t_3)}) \leqslant \hat{R}(\mathbf{w}^{(t_2)}) \leqslant \hat{R}(\mathbf{w}^{(0)})$

Training by Penalized Risk: *Regularization*

Regularization follows this idea: *can we modify empirical risk, such that it stops dropping after* t_2*?*

Let's continue with our example: *assume risk's value at each* $\mathbf{w}$ *is*

$$\hat{R}(\mathbf{w}^{(t_3)}) = 0.001 \qquad \hat{R}(\mathbf{w}^{(t_2)}) = 0.01 \qquad \hat{R}(\mathbf{w}^{(0)}) = 100$$

We may note that as training progresses: vector $\mathbf{w}^{(t)}$ *becomes larger. So, what if we add a penalty to risk that is proportional to* $\|\mathbf{w}^{(t)}\|^2$*: this way when risk becomes too small, this penalty becomes large and thus the sum increases.* Let's look at this sum at *different iterations*

$$\tilde{R}(\mathbf{w}^{(0)}) = \hat{R}(\mathbf{w}^{(0)}) + \|\mathbf{w}^{(0)}\|^2 = 100$$
$$\tilde{R}(\mathbf{w}^{(t_2)}) = \hat{R}(\mathbf{w}^{(t_2)}) + \|\mathbf{w}^{(t_2)}\|^2 = 19.04$$
$$\tilde{R}(\mathbf{w}^{(t_3)}) = \hat{R}(\mathbf{w}^{(t_3)}) + \|\mathbf{w}^{(t_3)}\|^2 = 44.761$$

Training by Penalized Risk: *Regularization*

Penalized risk shows a different behavior

$$\tilde{R}(\mathbf{w}^{(0)}) = 100 \qquad \tilde{R}(\mathbf{w}^{(t_2)}) = 19.04 \qquad \tilde{R}(\mathbf{w}^{(t_3)}) = 44.761$$

From above values, we can say: *if we apply SGD to minimize penalized risk we may get from* $\mathbf{w}^{(0)}$ *to* $\mathbf{w}^{(t_2)}$*; however, we will not get from* $\mathbf{w}^{(t_2)}$ *to* $\mathbf{w}^{(t_3)}$

This idea is called regularization which can prevent NNs from overfitting

Training by Penalized Risk: *Regularization*

Penalized risk shows a different behavior

$$\tilde{R}(\mathbf{w}^{(0)}) = 100 \qquad \tilde{R}(\mathbf{w}^{(t_2)}) = 19.04 \qquad \tilde{R}(\mathbf{w}^{(t_3)}) = 44.761$$

From above values, we can say: *if we apply SGD to minimize penalized risk we may get from* $\mathbf{w}^{(0)}$ *to* $\mathbf{w}^{(t_2)}$*; however, we will not get from* $\mathbf{w}^{(t_2)}$ *to* $\mathbf{w}^{(t_3)}$

This idea is called regularization which can prevent NNs from overfitting

Regularization

In training with regularization, we minimize a penalized (regularized) form of the empirical risk, i.e.,

$$\min_{\mathbf{w}} \hat{R}(\mathbf{w}) + \Pi(\mathbf{w}) \qquad \text{(Regularized Training)}$$

$\Pi(\mathbf{w})$ *is a penalty that describes the behavior of* $\mathbf{w}$ *in the case of overfitting*

Classical Regularization Approaches

There are various regularization penalties: *some important ones are*

- ℓ_2 *or Tikhonov regularization in which we add a term proportional to* $\|\mathbf{w}\|^2$

$$\Pi(\mathbf{w}) = \lambda \|\mathbf{w}\|^2$$

↳ *This way we avoid very large weights*
↳ *This prevents perfect fit to training dataset reducing chance of overfitting*

Classical Regularization Approaches

There are various regularization penalties: *some important ones are*

- ℓ_2 *or Tikhonov regularization in which we add a term proportional to* $\|\mathbf{w}\|^2$

$$\Pi(\mathbf{w}) = \lambda \|\mathbf{w}\|^2$$

  - ↳ *This way we avoid very large weights*
  - ↳ *This prevents perfect fit to training dataset reducing chance of overfitting*

- ℓ_1 *or Lasso regularization in which we add a term proportional to* $\|\mathbf{w}\|_1$

$$\Pi(\mathbf{w}) = \lambda \|\mathbf{w}\|_1 = \lambda \sum_{i=1}^{D} |w_i|$$

  - ↳ *This way enforce* $\mathbf{w}$ *to be sparse, i.e., to have to many zeros*
  - ↳ *This way we reduce the capacity of NN and thus prevent overfitting*

Regularizing by *Dropout*

A less conventional regularization approach *is dropout* that was proposed by *Hinton et al.* first in *their 2012 paper* and then in *their 2014 paper*:[7] *the idea is at the same time easy and effective*

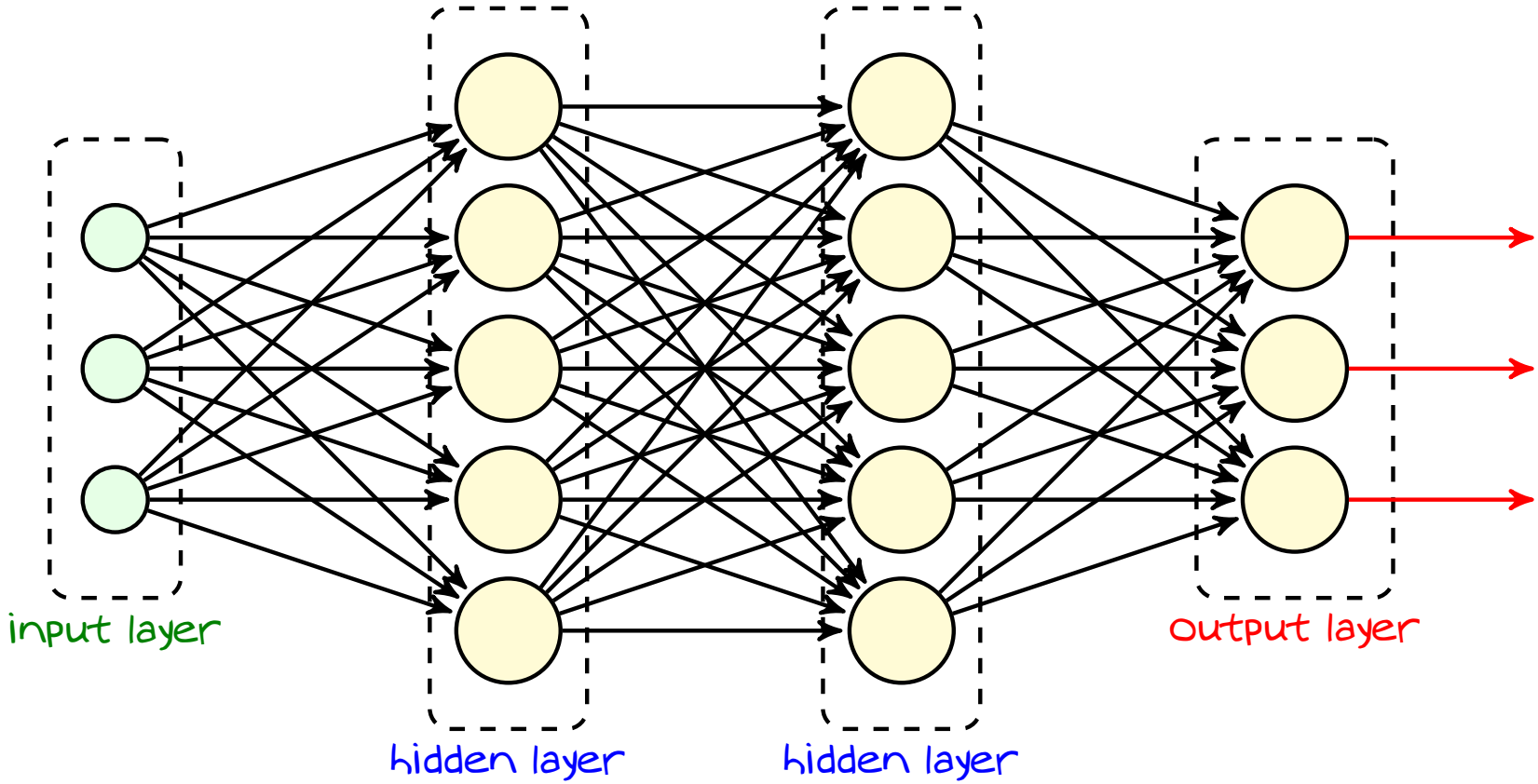
> *for each training iteration, we deactivate some nodes of NN at random*

or in other words we drop them out

[7] Click to check out the papers!

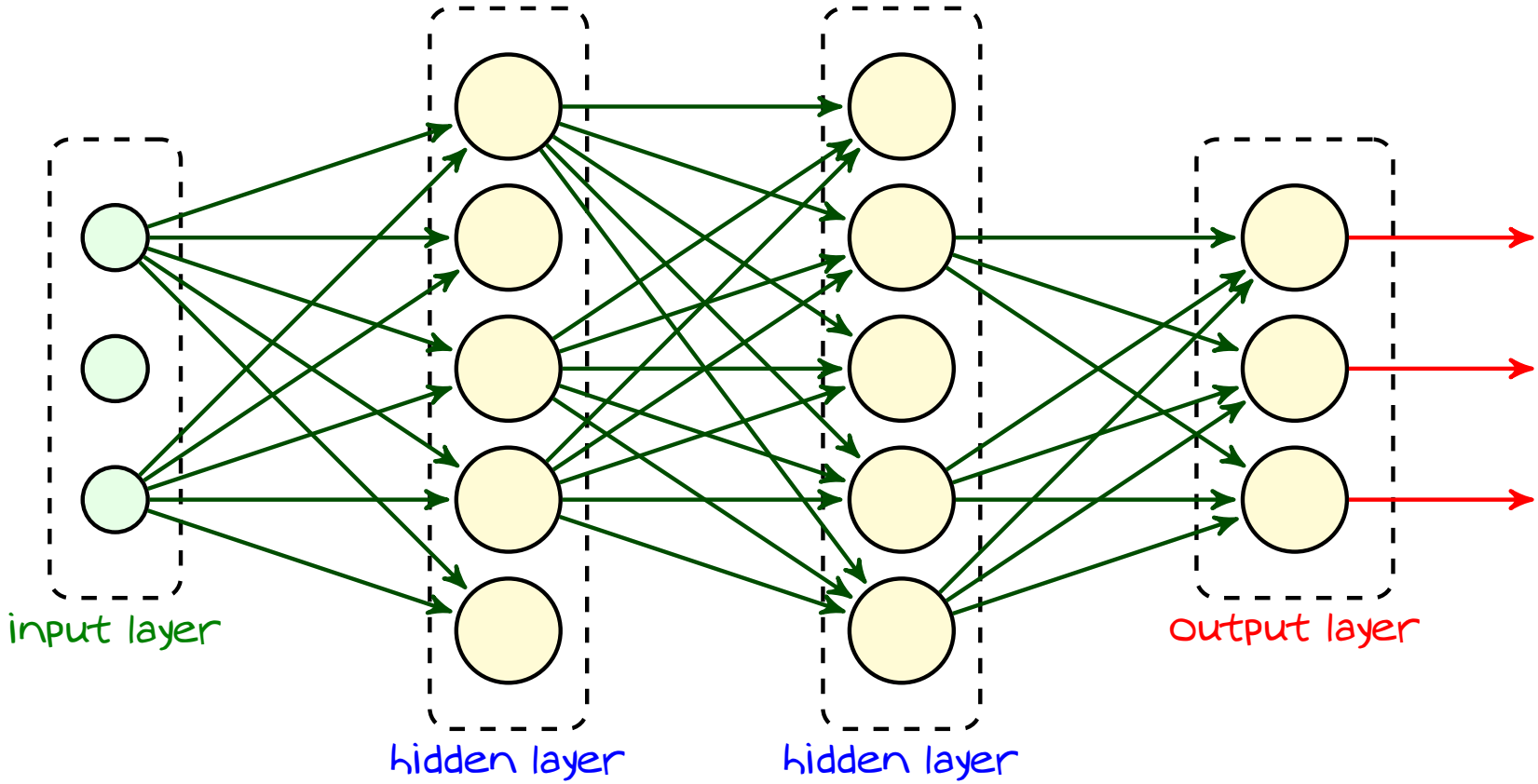
Dropout: *Schematic*

Let's say this is the dense NN



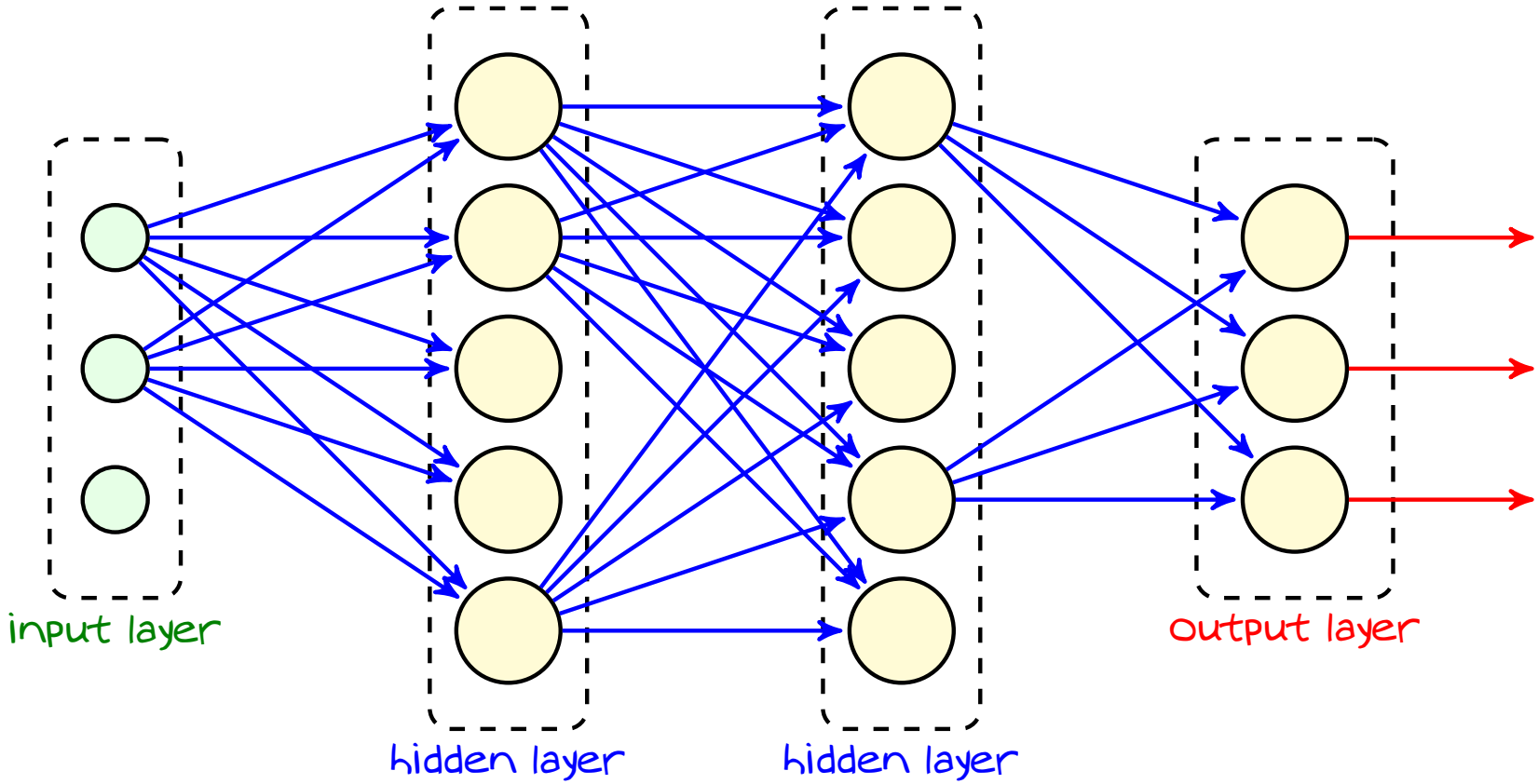
Dropout: *Schematic*

For first forward-backward we select few nodes in each layer



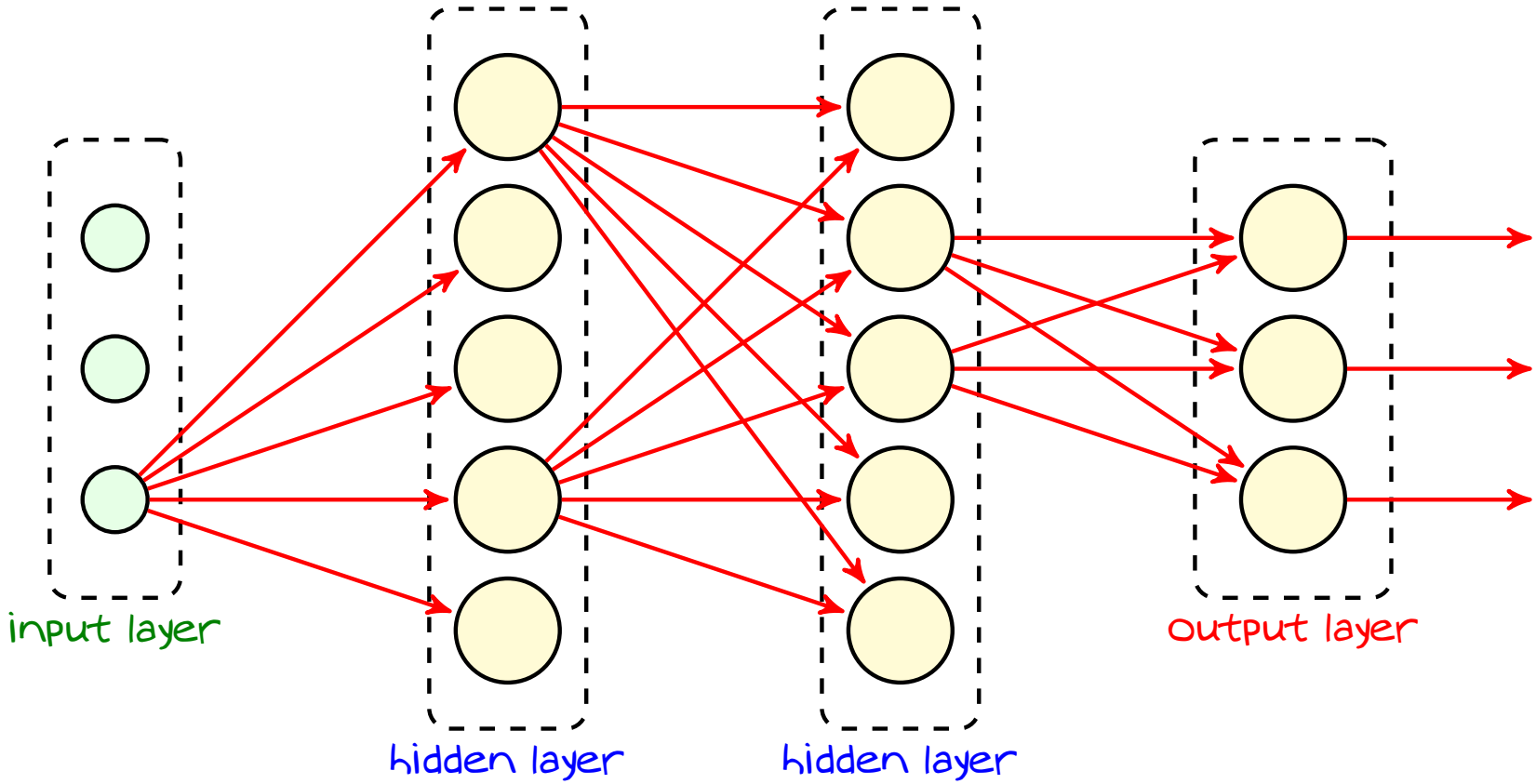
Dropout: *Schematic*

For second forward-backward we select new nodes in each layer at random



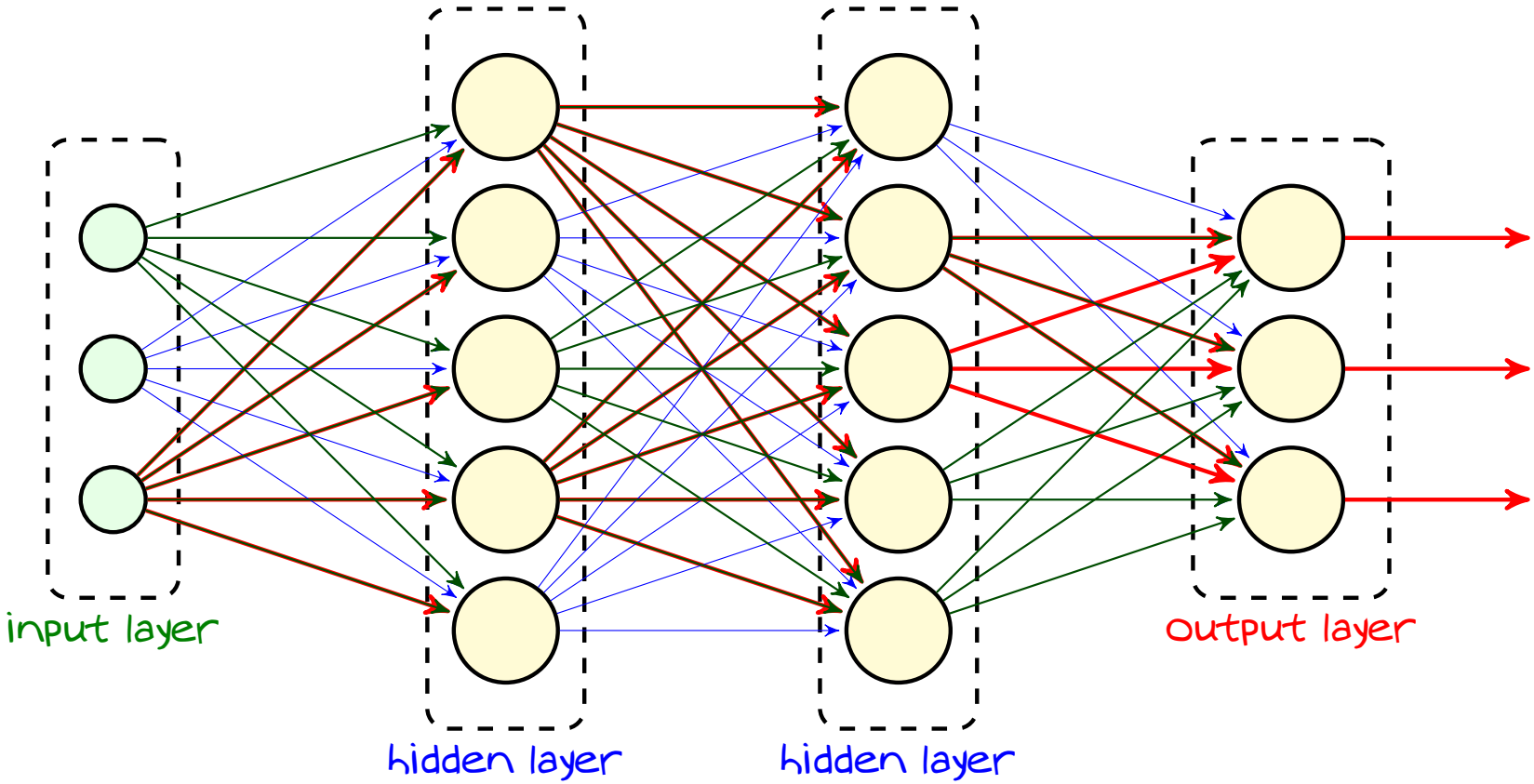
Dropout: *Schematic*

For next forward-backward we select again few nodes in each layer at random



Dropout: *Schematic*

At the end, we average the gradients determined over these reduced NNs



Dropout: *Intuition*

+ *But, why does dropout work?*
– We can explain it *heuristically*

Recall that *observing overfitting means that NN is larger than required*. With dropout, *in each iteration we train a smaller version of NN*

- *We randomly switch among these smaller versions*
- *Many of these smaller versions do not overfit*

We can look at the training loop with dropout as an averaged training of these smaller NNs; hence, training loop gets less chances to overfit

Dropout: *Intuition*

+ *But, why does dropout work?*
– We can explain it *heuristically*

Recall that *observing overfitting means that NN is larger than required*. With dropout, *in each iteration we train a smaller version of NN*

- *We randomly switch among these smaller versions*
- *Many of these smaller versions do not overfit*

We can look at the training loop with dropout as an averaged training of these smaller NNs; hence, training loop gets less chances to overfit

---

Dropout: *Intuition*

+ *But, why does dropout work?*
– We can explain it *heuristically*

Recall that *observing overfitting means that NN is larger than required*. With dropout, *in each iteration we train a smaller version of NN*

- *We randomly switch among these smaller versions*
- *Many of these smaller versions do not overfit*

We can look at the training loop with dropout as an averaged training of these smaller NNs; hence, training loop gets less chances to overfit

---

+ *Why do we do it randomly? Why not sticking to one smaller version?*
– Not all smaller NNs are good, and we cannot check all of them: *it's exponentially hard to check all smaller NNs*

Dropout: *Training Loop*

How does training change with dropout? *Training with dropout is exactly as before.*

Dropout: *Training Loop*

How does training change with dropout? *Training with dropout is exactly as before. Say we use mini-batches: for each mini-batch*

- *we compute the gradient by forward and backpropagation*
- *we give the gradient to the optimizer to apply the next iteration*

Dropout: *Training Loop*

How does training change with dropout? *Training with dropout is exactly as before. Say we use mini-batches: for each mini-batch*

- *we compute the gradient by forward and backpropagation*
- *we give the gradient to the optimizer to apply the next iteration*

The only thing that is different now is that we set the output of some nodes to zero in the forward pass $\equiv$ *only forward propagation changes*

Dropout: *Training Loop*

How does training change with dropout? *Training with dropout is exactly as before. Say we use mini-batches: for each mini-batch*

- *we compute the gradient by forward and backpropagation*
- *we give the gradient to the optimizer to apply the next iteration*

> *The only thing that is different now is that we set the output of some nodes to zero in the forward pass* $\equiv$ *only forward propagation changes*

Let's make it concrete: *when we pass forward, we generate random masks for each layer* $\ell = 0, \ldots, L$. *Mask of layer* ℓ *is a vector whose length is layer's width and entries are 0 or 1, i.e.,* $\mathbf{s}_\ell \in \{0, 1\}^{\mathcal{W}_\ell}$. *Entries of* $\mathbf{s}_\ell$ *are generated randomly*

$$\text{each entry of } \mathbf{s}_\ell = \begin{cases} 1 & \text{with probability } p_\ell \\ 0 & \text{with probability } 1 - p_\ell \equiv \text{dropout probability} \end{cases}$$

Dropout: *Forward Propagation*

Let's show generation of random mask $\mathbf{s}_\ell$ by following notation

$$\mathbf{s}_\ell = \texttt{mask}\left(\mathcal{W}_\ell | p_\ell\right)$$

Dropout: *Forward Propagation*

Let's show generation of random mask $\mathbf{s}_\ell$ by following notation

$$\mathbf{s}_\ell = \texttt{mask}\left(\mathcal{W}_\ell | p_\ell\right)$$

We are going to do forward propagation for each data-point as

`DropoutForwardProp()`:

1: Initiate with $\mathbf{y}_0 = \boldsymbol{x}$
2: **for** $\ell = 0, \ldots, L$ **do**
3: Generate $\mathbf{s}_\ell = \texttt{mask}\left(\mathcal{W}_\ell | p_\ell\right)$ `# random mask`
4: Set $\mathbf{y}_\ell = \mathbf{y}_\ell \odot \mathbf{s}_\ell$ `# dropout nodes`
5: Add $y_\ell[0] = 1$ and determine $\mathbf{z}_{\ell+1} = \mathbf{W}_{\ell+1}\mathbf{y}_\ell$ `# forward affine`
6: Determine $\mathbf{y}_{\ell+1} = f_{\ell+1}(\mathbf{z}_{\ell+1})$ `# forward activation`
7: **end for**
8: **for** $\ell = 1, \ldots, L+1$ **do**
9: Return $\mathbf{y}_\ell$ and $\mathbf{z}_\ell$
10: **end for**

Dropout: *Backpropagation*

The backpropagation goes exactly as before: *of course those outputs that were dropped out participate with value zero in gradient computation*

Dropout: *Backpropagation*

The backpropagation goes exactly as before: *of course those outputs that were dropped out participate with value zero in gradient computation*

One final piece of trick

After training is over, we scale weights of each layer with its retain probability p_ℓ: say T is the last iteration of training loop; then, we finally do

$$\mathbf{W}_\ell^{(T)} \leftarrow p_\ell \mathbf{W}_\ell^{(T)}$$

+ *Why do we do that?*
– Well! It's practically understood; however, we can justify it as follows: *each weight could be what has been computed with probability p_ℓ and zero with probability $1 - p_\ell$. We hence compute the average*

Dropout: *Implementation*

Dropout is implemented in almost all deep learning libraries

```
>> import torch

>> torch.nn.Dropout()
```

Typical choices of retain probability p_ℓ are

- *for input layer, i.e., layer 0,* $p_\ell = 0.8$
- *for hidden layers* $p_\ell = 0.5$

It's generally suggested to drop out more at hidden layers

Data Preparation

Frankly speaking, preparing data for training and testing is

most time-consuming part of a practical project

+ *How hard it could be? Really harder than finding right hyperparameters, regularizing or adjusting the optimizer?*
– Sure! We get used to such *design tasks* and start to have feeling about NNs, as they repeat so much. *The main thing that is new is data*

Data Preparation

Frankly speaking, preparing data for training and testing is

most time-consuming part of a practical project

+ *How hard it could be? Really harder than finding right hyperparameters, regularizing or adjusting the optimizer?*
– Sure! We get used to such *design tasks* and start to have feeling about NNs, as they repeat so much. *The main thing that is new is data*

How to prepare data that *works well for our purpose* is an individual topic discussed in *courses on data science*: we only briefly touch it in this section

Data Preparation

Procedure of processing raw data into a form suitable for underlying model

Data Preprocessing Procedures

There is a long list of techniques for data preparation

- *Data augmentation which we do when training dataset is too small*
- *Data cleaning that we do to either remove or modify unwanted samples in training dataset*
  - ↳ Samples like outliers, duplicates and nulls
- *Data transform which aims to transform data into a form that lead to more robust training*
- *Dimensionality reduction which reduces the redundancy by extracting lower-dimensional features with minimal confusion from samples*
- . . .

We discuss the first two items in this section: *let's start with the first one that we already have some idea about*

Expanding Dataset via *Augmentation*

Recall that one conclusion from overfitting was that dataset is too small

In practice, we may expanding our dataset by *augmenting* it

+ *Say we have a set of cat and dog images! How can we expand it?*
– Well! Let's take a look at this simple example!

Say we have a dataset of cat and dog N-pixel images. We write it as

$$\mathbb{D} = \{(\boldsymbol{x}_b, v_b) : b = 1, \ldots, B\} \qquad v_b = 0 : \texttt{cat} \qquad v_b = 1 : \texttt{dog}$$

Say $\boldsymbol{x}_1$ is pixel-vector of a cat image. We are given function $\mathcal{A} : \mathbb{R}^N \mapsto \mathbb{R}^N$: it gets $\boldsymbol{x}_1$ and returns $\hat{\boldsymbol{x}} = \mathcal{A}(\boldsymbol{x}_1)$ which satisfies two conditions

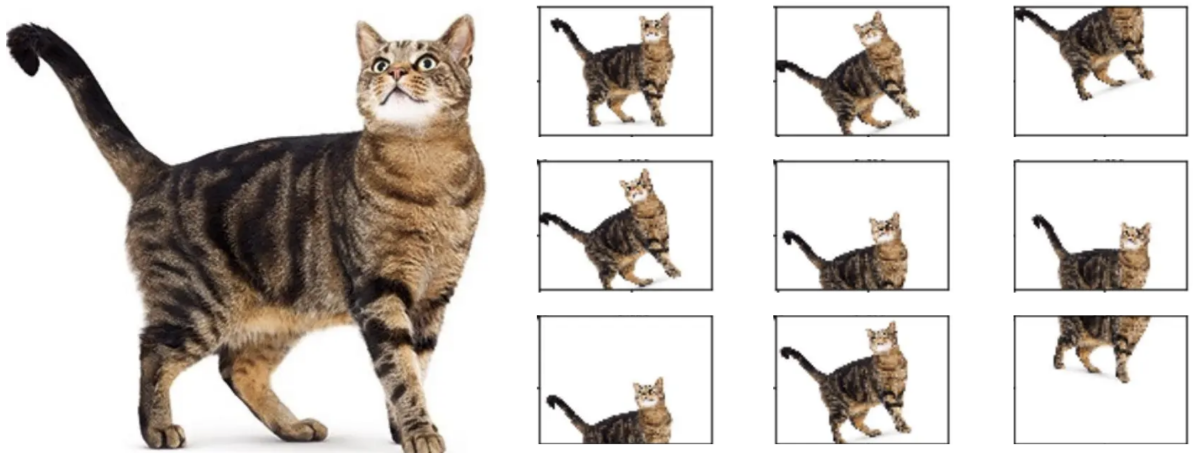
1. *After plotting $\hat{\boldsymbol{x}}$ we still see a cat*
2. *This new cat image does not belong to the dataset, i.e., $(\hat{\boldsymbol{x}}, 0) \notin \mathbb{D}$*

We can then expand our dataset as $\mathbb{D} \leftarrow \{(\hat{\boldsymbol{x}}, 0)\} \cup \mathbb{D}$!

Data Augmentation: *Example*

- \+ *But, how could we know such a function $\mathcal{A}(\cdot)$?*
- – For images we know some!

We can simply *rotate*, *shift*, *zoom-in*, *zoom-out*, *change intensity* and so on!



How can we rotate an image? Multiply it by a rotation matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$

These are examples of data augmentation *by engineering*

we can find $\mathcal{A}(\cdot)$ analytically using properties of our data

Data Augmentation by *Engineering*

Data augmentation by *engineering depends* on *data* and *learning task*

- If we are classifying a set of images
  - ↳ *We can apply geometric transformations, e.g., random rotating, flipping, stretching, zooming, and cropping*
  - ↳ *We may use kernel filters to make random filtering, e.g., changing sharpness or blurring*
  - ↳ *We can apply random color-space transformations, e.g., changing intensity or brightness, and exchanging RGB channels*
  - ↳ *We can randomly remove pixels, i.e., set their value to some reference value*
  - ↳ *We can randomly mix images, e.g., get multiple cat images and make a new one out of them*

Note that *we may train a separate NN to do any of those transforms for us: just think about the last one!*

Data Augmentation by *Engineering*

Data augmentation by *engineering depends* on *data* and *learning task*

- If we are working with audio signals
  - ↳ *We can apply noise injection, e.g., add background noise*
  - ↳ *We may change sampling rate to change the speed of audio data*
  - ↳ *We can apply random shifts, e.g., sample signals at a bit deviated points*
- If we are dealing with text data
  - ↳ *We may apply random replacements, e.g., replace a word with its synonyms*
  - ↳ *We may manipulate syntax-tree, e.g., rephrase a sentence*
  - ↳ *We can do random shuffling is some applications*
  - ↳ *We may apply removal and insertion of redundant words, e.g., so*

Synthetic Data Generation

An alternative approach to expand a small dataset is to

generate synthetic data

We did this in Assignment 1 for the dummy projectile example

Recall we had a projectile with velocity v and height h: we knew by Newton's laws that the hitting distance d is given by

$$d = 0.45v\sqrt{h}$$

To make dataset, we generated lots of velocities v_i and heights h_i at random and for each pair we determined d_i by above equation

Synthetic Data vs Augmented Data

When we generate synthetic data

- *we need to know the process of data being generated from a seed*
- *we make new data by simulating the process with a random seed*

When we augment data

- *we need to know transforms that keep data-points inside dataset*
- *we apply those transforms on the existing data*

+ *It sounds that synthetic data is only feasible in scientific problems, where we know physics! Right?!*
– Until few years ago the answer was Yes! But, currently No! Nowadays, *we can generate images of what we want from noise using generative adversarial networks (GANs) or diffusion models!*

Data Augmentation: *Formulation*

$\mathcal{A}(\cdot)$ *gets data-point* x *and returns new data-point* $\hat{x} = \mathcal{A}(x)$

In general, we do not need $\mathcal{A}(\cdot)$ operate on single data-point

$\mathcal{A}(\cdot)$ *can get multiple samples and generate a new one*

For instance, *it combines multiple cat images and makes a new one*

Also, $\mathcal{A}(\cdot)$ is not enforced to return data-points with same labels

label of what $\mathcal{A}(\cdot)$ *returns is only required to be a valid label*

For instance, *it gets multiple cat images and makes a new dog image; however, if our dataset includes only cats and dogs it should not return a horse image*

We can now formulate data augmentation more precisely

Augmentation and Synthetic Generation: *Formulation*

Data Augmentation

A data augmentation technique $\mathcal{A}(\cdot)$ takes the training dataset as the input and returns new samples from data distribution

We can think of it as the following block

$$\text{training dataset } \mathbb{D} \rightsquigarrow \boxed{\mathcal{A}(\cdot)} \rightsquigarrow (\boldsymbol{x}_{\text{new},1}, \upsilon_{\text{new},1}), (\boldsymbol{x}_{\text{new},2}, \upsilon_{\text{new},2}), \ldots$$

Synthetic Data Generation

A synthetic data generator $\mathcal{S}(\cdot)$ takes a random seed as the input and returns samples from data distribution

We can think of it as the following block

$$\text{random seed } \mathbf{s} \rightsquigarrow \boxed{\mathcal{S}(\cdot)} \rightsquigarrow (\boldsymbol{x}_1, \upsilon_1), (\boldsymbol{x}_2, \upsilon_2), \ldots$$

Augmentation and Synthetic Generation: *Formulation*

+ *There is one point left bothering in definitions!*

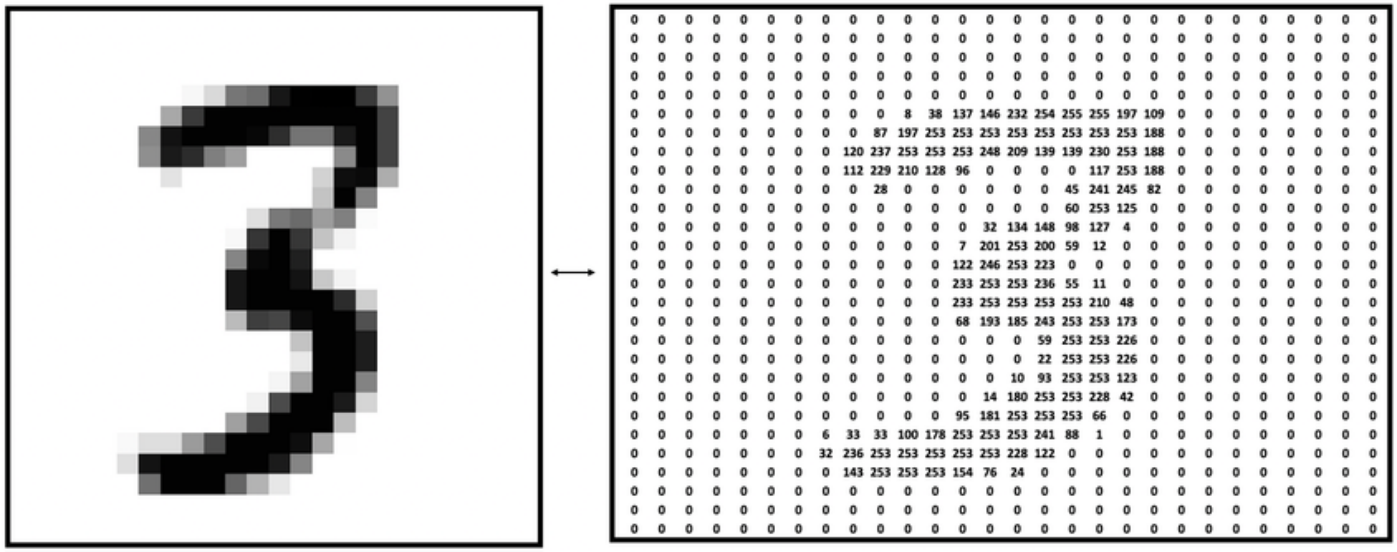
What does data distribution mean?

– In simple words it means an *abstract machine that generates only data-points that we need*

+ *But, why we call it distribution*

– Well! Let's get it clear!

Possible Samples of Data



Let's start with MNIST dataset: *in MNIST we have 28 × 28 pixel images*

- *They are 8-bit images: each pixel value is an integer between 0 and 255*
- *These images are all hand-written numbers*

But, we note that they are

- *neither all 28 × 28 pixel 8-bit images images*
- *nor all possible images of hand-written numbers*

Possible Samples of Data

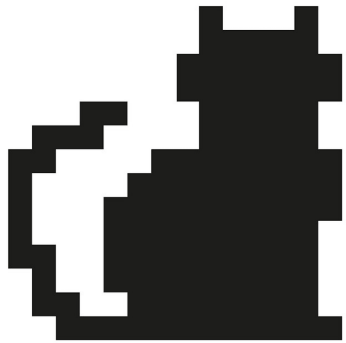
MNIST images are not all 28 × 28 pixel 8-bit images

A 28 × 28 pixel image has in total 784 pixels with each pixel being one of 256 different possible values values: in total we have

$$\textit{total number of images} = 256^{784} = 2^{6272} > 10^{1881}$$

MNIST has only $70,000 < 10^5$ *images!*

We also note that *not all of those* 2^{6272} *can get into MNIST!* For instance,



Possible Samples of Data

MNIST images are not all possible images of hand-written numbers

We can imagine that *among those* 2^{6272} *images there are much more than only* $70,000$ *images of hand-written numbers: just take an image of my handwriting and convert it into a* 28×28 *pixel image!*

Space of Possible Data *(Data Space)*

Space of possible data is the set of all labeled data-points whose labels are valid

In our example, *the space of possible data is*

$$\mathbb{X} = \left\{ \boldsymbol{x} \in \{0, \dots, 255\}^{784} : \textit{image of } \boldsymbol{x} \textit{ is classified as hand-written number} \right\}$$

Of course space of possible data is only a definition: most of the time, it is impossible to specify it explicitly like above example

Data-Point as *Sample of Random Object*

We obviously see that a dataset is a subset of data space: *MNIST is a subset of $\mathbb{X}$ defined in the last slide and it is much much smaller*

In machine learning, we have a specific way to look at datasets

dataset is collection of samples drawn randomly from the data space

This can be easily understood as the example below

Recall the data space $\mathbb{X}$ defined in last slide

- *We assume that there exist a machine with a button*
  - ↳ *each time we push this button the machine randomly generates data-point x from $\mathbb{X}$*

MNIST is then generated by pushing this button for $70,000$ times

Data Distribution

Data Distribution

Data distribution is the probability *distribution* by which the *dataset* has been generated from the *data space*

+ *Do we know this distribution?!*
– No! We can *neither fully* specify the *space of possible data nor* the *data distribution!* They are mainly abstract definitions!

But, we can have a partial understanding: *assume I say such a sentence*

> "*MNIST* contains 70,000 samples drawn from *data distribution* $p(\boldsymbol{x})$"

We cannot find out what $p(\boldsymbol{x})$ *is, but we know for sure*

 $\equiv \boldsymbol{x}_1 \rightsquigarrow p(\boldsymbol{x}_1) \neq 0$

 $\equiv \boldsymbol{x}_2 \rightsquigarrow p(\boldsymbol{x}_2) = 0$

Data Distribution

From now on, if we get into such a sentence in a paper

> the learner has access to a *small reference dataset* $S_T := \{(x_{T,1}, y_{T,1}), \ldots, (x_{T,m_T}, y_{T,m_T})\}$ of m_T samples drawn i.i.d. from a target distribution $\mathcal{D}_T$

we know what it means!

Last note: although we do not have access to data distribution

we can in practice approximate it from samples that we have collected

For instance, if data-points are heights of different people: we can plot the histogram to approximate data distribution

Data Distribution: *Practical Aspects*

In practice, this looking at data-points as samples of a random process lets us *use statistical methods to preprocess data*

We can use these methods to

- *realize if our dataset is a good representative of data space*
  - ↳ *Maybe we have too many non-typical data-points*
- *transform our dataset into a better representative of data space*
  - ↳ *Maybe we should add, remove or change some data-points*

This is what we call *data cleaning*: *this can be a separate course! So, we make it very short by discussing only few practical techniques*

Data Cleaning: *Duplicates*

Duplication impacts model training: *assume we have training dataset*

$$\mathbb{D} = \{(\boldsymbol{x}_b, v_b) : b = 1, \ldots, B\}$$

Without any duplication, our training loop solves

$$\min_{\mathbf{w}} \frac{1}{B} \sum_{b=1}^{B} \mathcal{L}\left(y_b, v_b\right) : y_b \text{ output of NN with weights } \mathbf{w} \text{ and input } \boldsymbol{x}_b$$

Now assume that we copy $(\boldsymbol{x}_1, v_1)$ *by mistake* M *times: the training on this duplicated dataset is*

$$\min_{\mathbf{w}} \frac{1}{B+M} \sum_{b=1}^{B} \mathcal{L}\left(y_b, v_b\right) + \frac{M}{B+M} \mathcal{L}\left(y_1, v_1\right)$$

which is not the same thing!

Data Cleaning: *Duplicates*

Having the same data-points in dataset does not necessarily mean duplication: *consider the following two simple examples*

1. *We are training an NN that takes age, education and place of birth as input and returns number of children as output*
   - ↳ Dataset has too many $\boldsymbol{x}_b = [22, \text{Bachelor}, \text{Toronto}]$ and $v_b = 0$
   - ↳ These are not duplicates since they come from independent samples
2. *We are training an NN that takes age and height as input and returns weight as output*
   - ↳ It is not likely to have $\boldsymbol{x}_b = [48, 176.42]$ and $v_b = 73.31$ in dataset
   - ↳ They most probably come from the same person and are duplicates

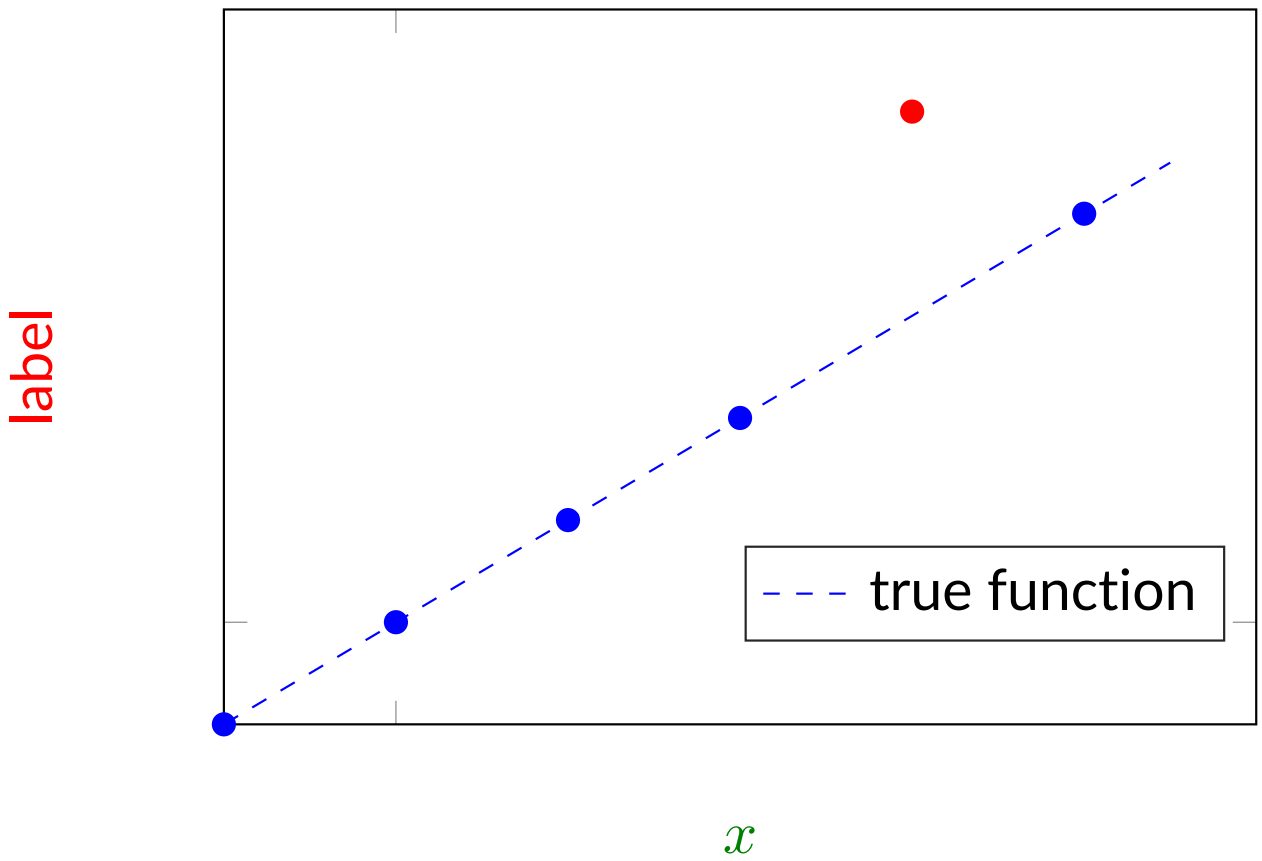
---

Bingo! *Terms like "come from independent samples" and "not likely to have" are used since we look at data-points as samples of a random process*

Data Cleaning: *Outliers*

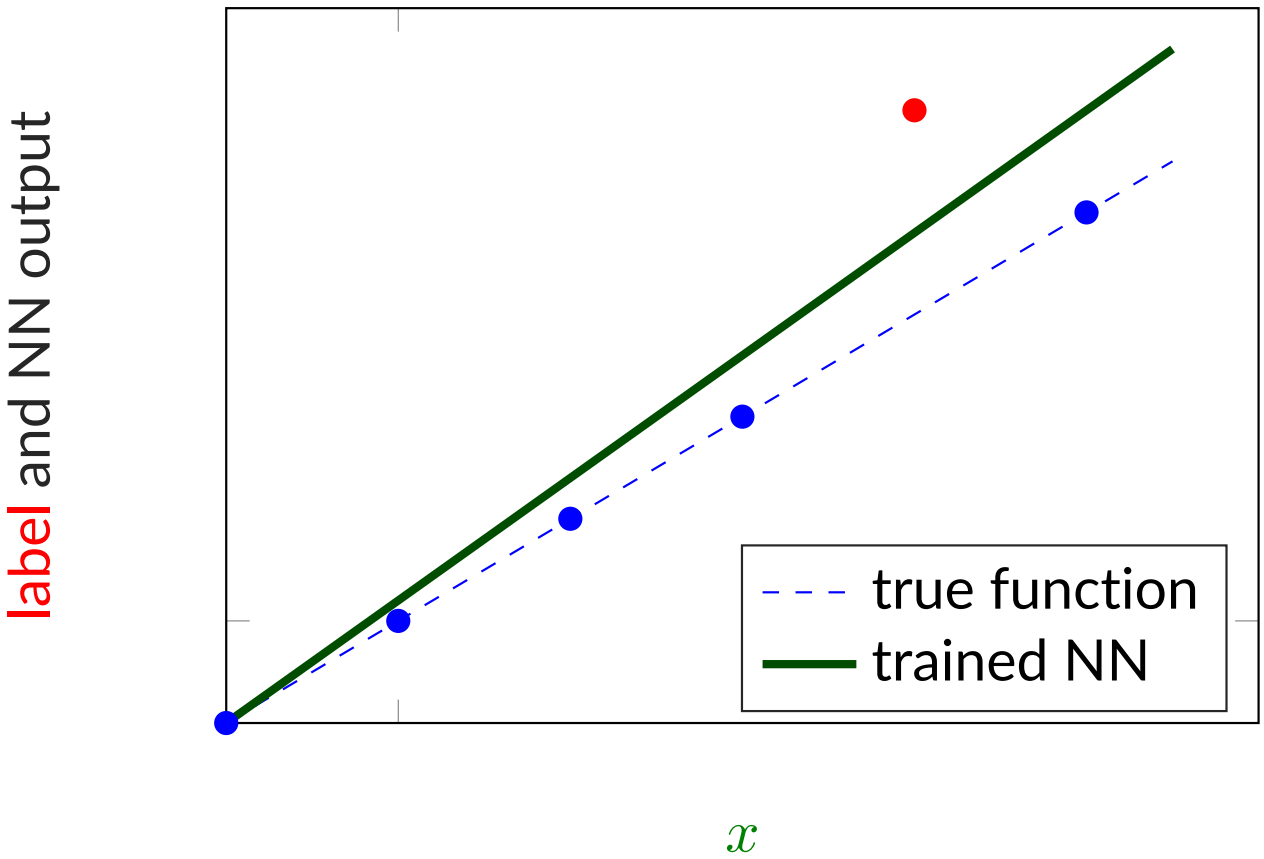
Outliers

Outliers are data-points that lie an abnormal distance from other samples



Data Cleaning: *Outliers*

Outliers can hinder training even with good tuning and regularization



Data Cleaning: *Outliers*

Finding outliers is required for training of a model that generalizes well

---

There are two types of outliers is a dataset

1. *Univariate outliers which are detected from their marginal distributions*
   - ↳ *These data-points are understood to be outliers from an individual variable in them without comparing to other variables*

We collect heights and weights: Sultan Kösen[a] is among our samples with height 2.51 m! Without checking weights, we can say this is an outlier

[a] Tallest alive person in the world

Data Cleaning: *Outliers*

Finding outliers is required for training of a model that generalizes well

---

There are two types of outliers is a dataset

2. *Multivatiate outliers which are detected from their joint distributions*
   - ↳ *These are understood to be outliers by comparing different variables*

We collect heights and weights: a sample with height 1.82 m and another with weight 24 kg are individually normal; however a sample with height 1.82 m and weight 24 kg is an outlier

Data Cleaning: *Outliers*

+ *How can we handle outliers?*
- Well! It depends

Conventional approaches to handle outliers are to

1. *remove them from training dataset*
   - ↳ *It might be a good idea for small NNs with low model capacity*
   - ↳ *It can hinder generalization of our model if we have detected outliers based on poor statistics, e.g., not enough samples to understand data distribution*
2. *use loss functions that are robust against outliers*
   - ↳ *They give higher weights to typical data-points*
   - ↳ *They give less weights to outliers*
   - ↳ *An example is to use Minkowski error instead of squared error for regression*

Take a look at Python library `Pandas` if you need to do any data cleaning

Standardization

In Assignment 2 *we implemented forward and backward pass of an FNN for MNIST classification: giving image pixels to NN can return huge features!*

Feature

Outputs of hidden layers, i.e., $\mathbf{y}_\ell$

This is a *typical observation in practice*; however, *it does not make trouble only in forward pass: it can also impact severely the training, i.e., backward pass*

Standardization

In Assignment 2 *we implemented forward and backward pass of an FNN for MNIST classification: giving image pixels to NN can return huge features!*

Feature

Outputs of hidden layers, i.e., $\mathbf{y}_\ell$

This is a *typical observation in practice*; however, *it does not make trouble only in forward pass: it can also impact severely the training, i.e., backward pass*

The solution to this issue is to *standardize the inputs and features*

> *Standardization means making variables look the same in all directions*

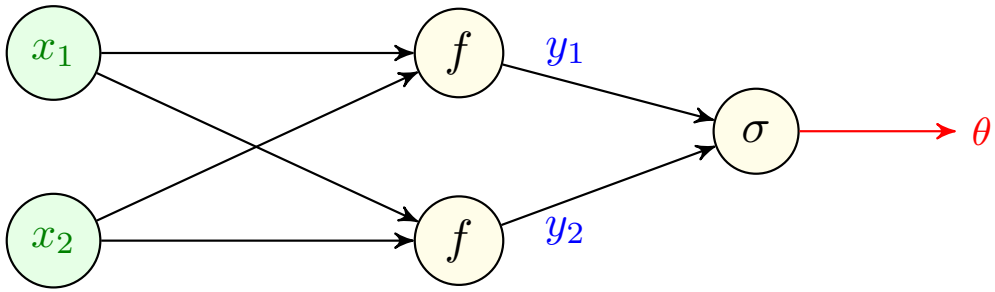
This is done by two particular techniques in practice

- *input standardization*
- *batch normalization*

Let's understand them *through an easy example*

Standardization: *Example*

Consider the following simple NN: *here we have a two-dimensional input, one hidden layer with a two-dimensional feature and a single output*



The inputs could be anything, for instance

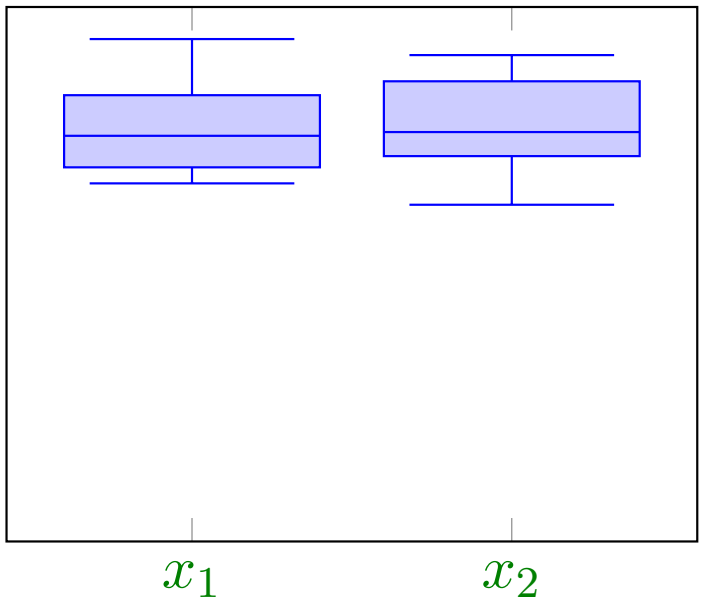
- ***Example A:*** *both are heights in centimeters*
  - ↳ *they are in the same range*
  - ↳ *distributions of variables have close means and variances*
- ***Example B:*** *x_1 is height centimeters and x_2 is number of children*
  - ↳ *they are in the widely-different range*
  - ↳ *distributions of variables have strongly-different means and variances*

Standardization: *Example*

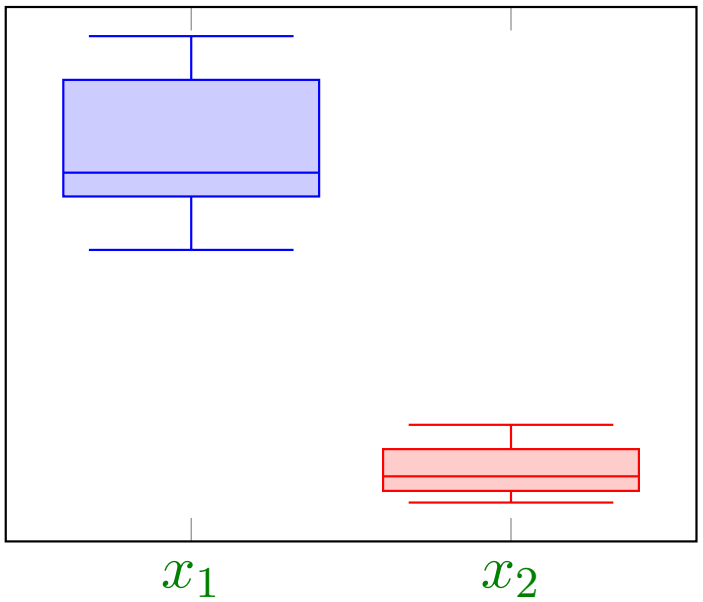
We expect *different* *behavior* *in* *forward pass*:

- *in* ***Example A*** *all variables are in the same scale*
- *in* ***Example B*** *each variable is in different scale*

Example A



Example B



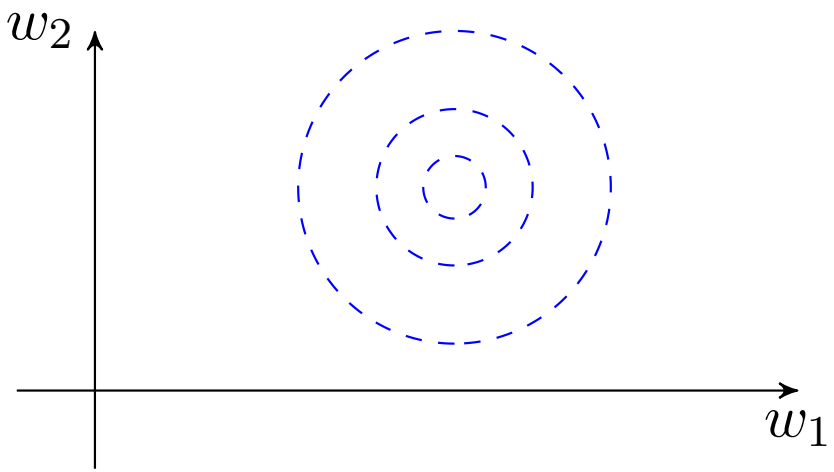
Standardization: *Example*

It also leads to different behavior in backward pass:

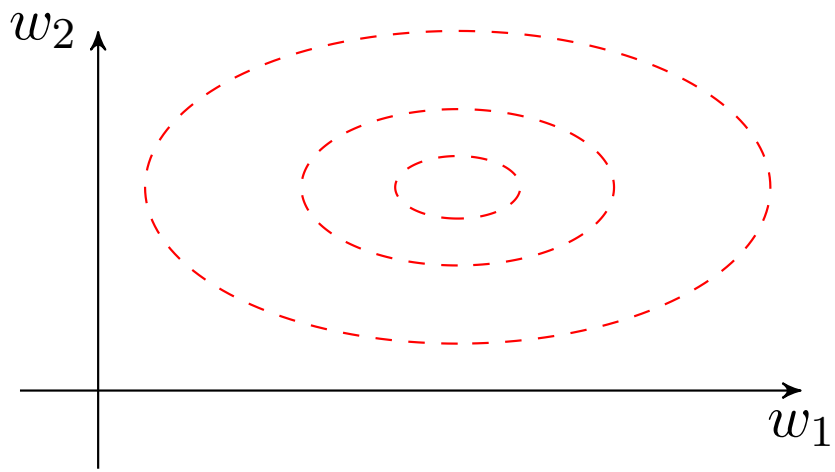
- *in **Example A** empirical risk's curvature is similar in all directions*
- *in **Example B** empirical risk's curvature varies from one direction to another*

For instance, if we assume NN has only two weights to train, the counters of empirical loss can look as below

Example A

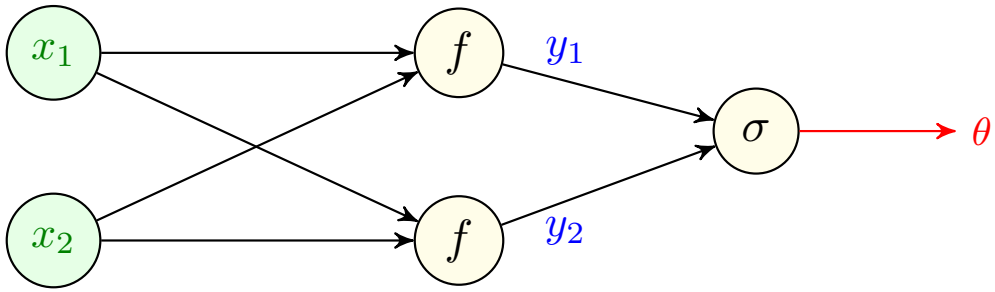


Example B



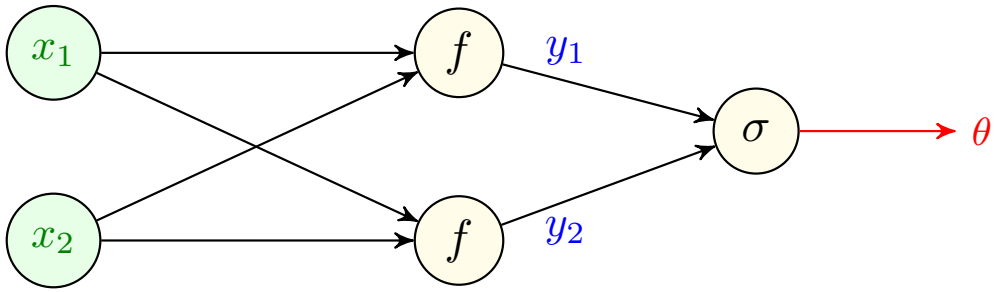
Standardization: *Different Layers*

Such behavior is *not specific to the input layer: we could also see the same behavior at hidden layers*

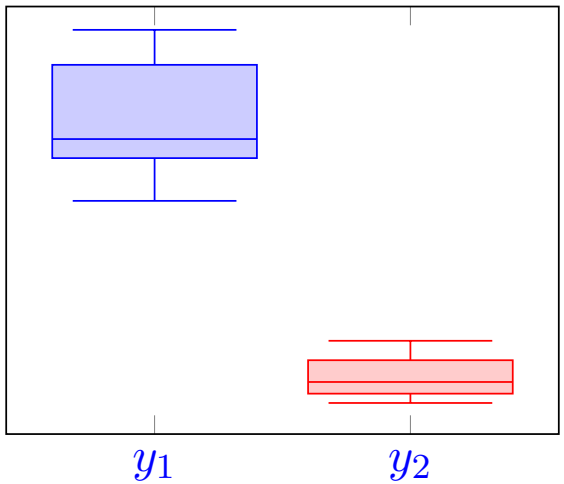


Standardization: *Different Layers*

Such behavior is *not specific* to the *input layer*: *we could also see the same behavior* at *hidden layers*



For instance in our example, even with properly-scaled inputs, features may evolve differently through training: after multiple iterations we end up with



Standardization via *Normalization*

The solution for *any layer* is to *shift* and *scale every sample input such that*

it becomes *zero-mean* and *unit-variance*

Then, we work with the standardized input

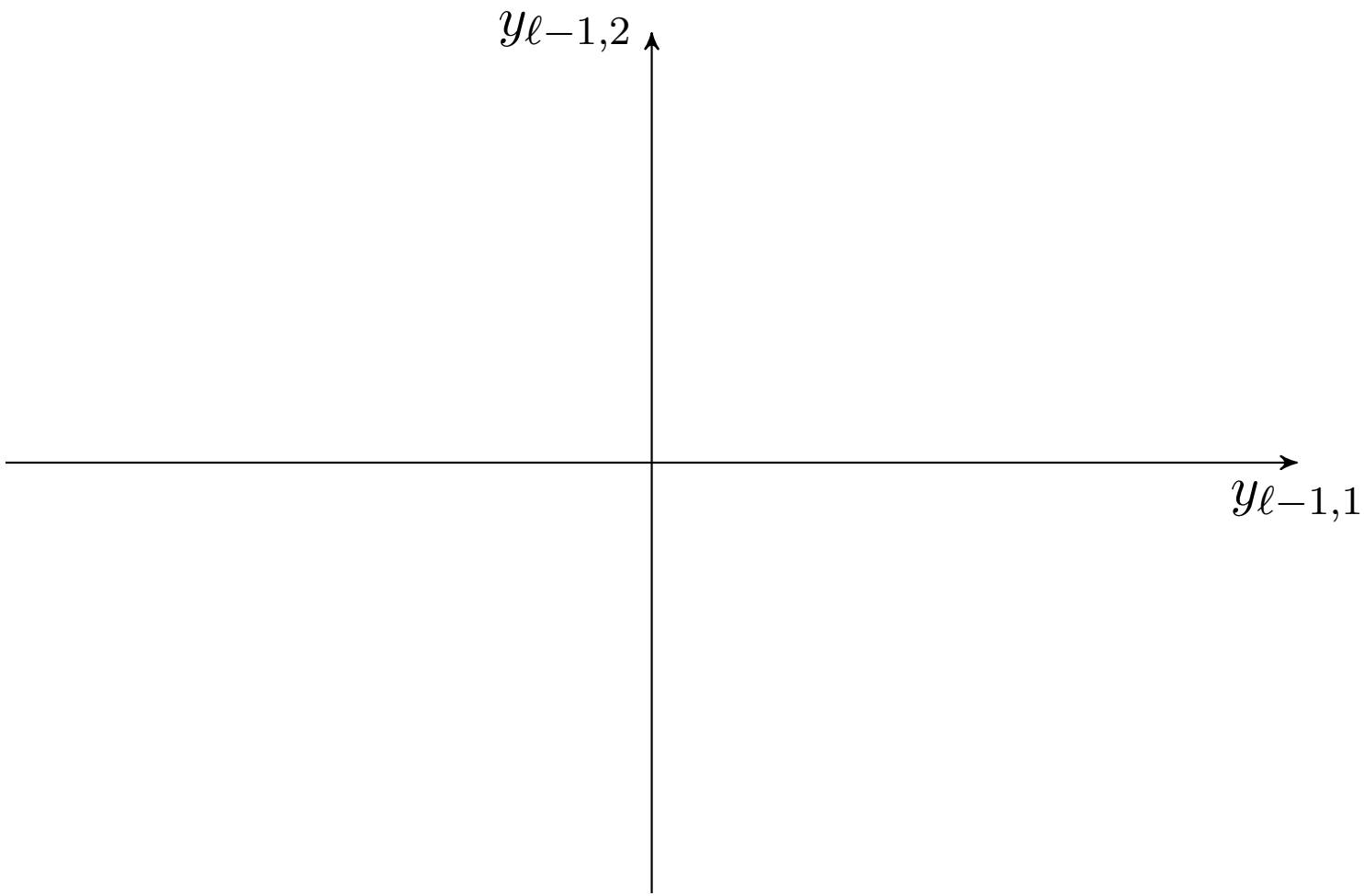
Say we are at layer ℓ: a sample input is $\mathbf{y}_{\ell-1,b}$, so we compute

$$\bar{\mathbf{y}}_{\ell-1,b} = \frac{\mathbf{y}_{\ell-1,b} - \mathbb{E}\left\{\mathbf{y}_{\ell-1}\right\}}{\sqrt{\mathbb{V}\mathrm{ar}\left\{\mathbf{y}_{\ell-1}\right\}}}$$

and then perform all further operations on $\bar{\mathbf{y}}_{\ell-1,b}$

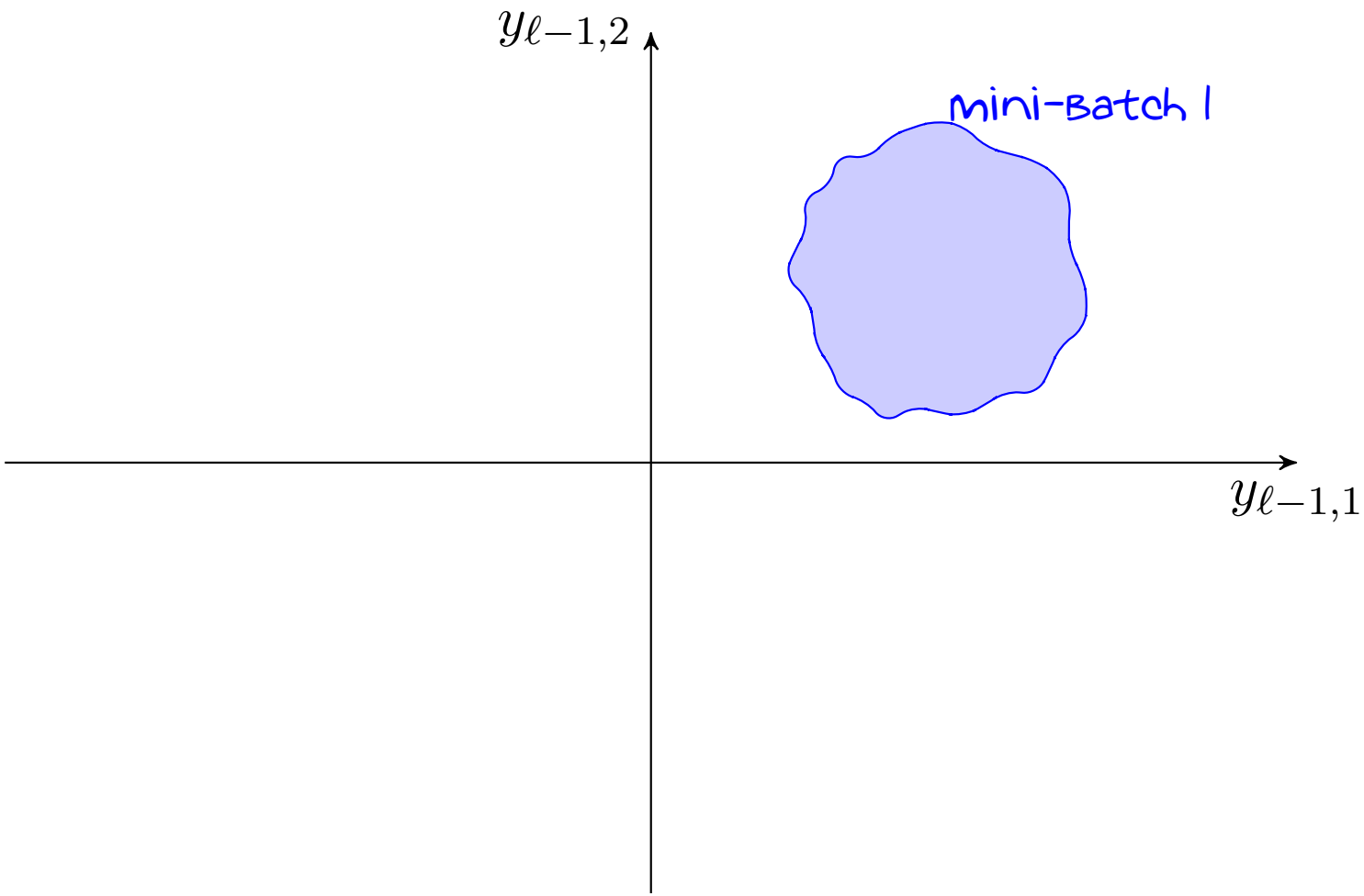
Visualizing *Normalization*

We can visualize *normalization* as below *for two-dimensional inputs*



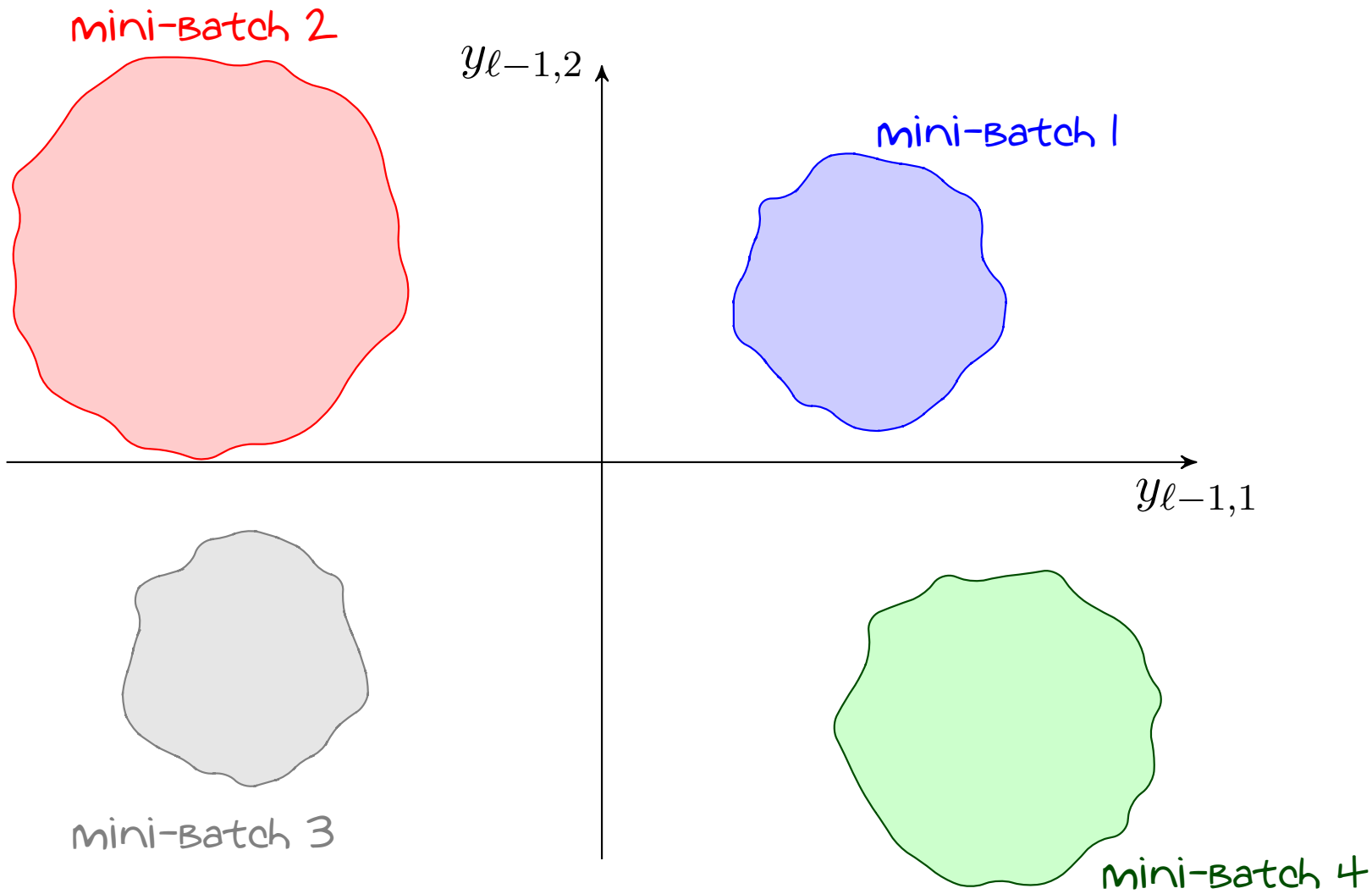
Visualizing *Normalization*

We can visualize *normalization* as below *for two-dimensional inputs*



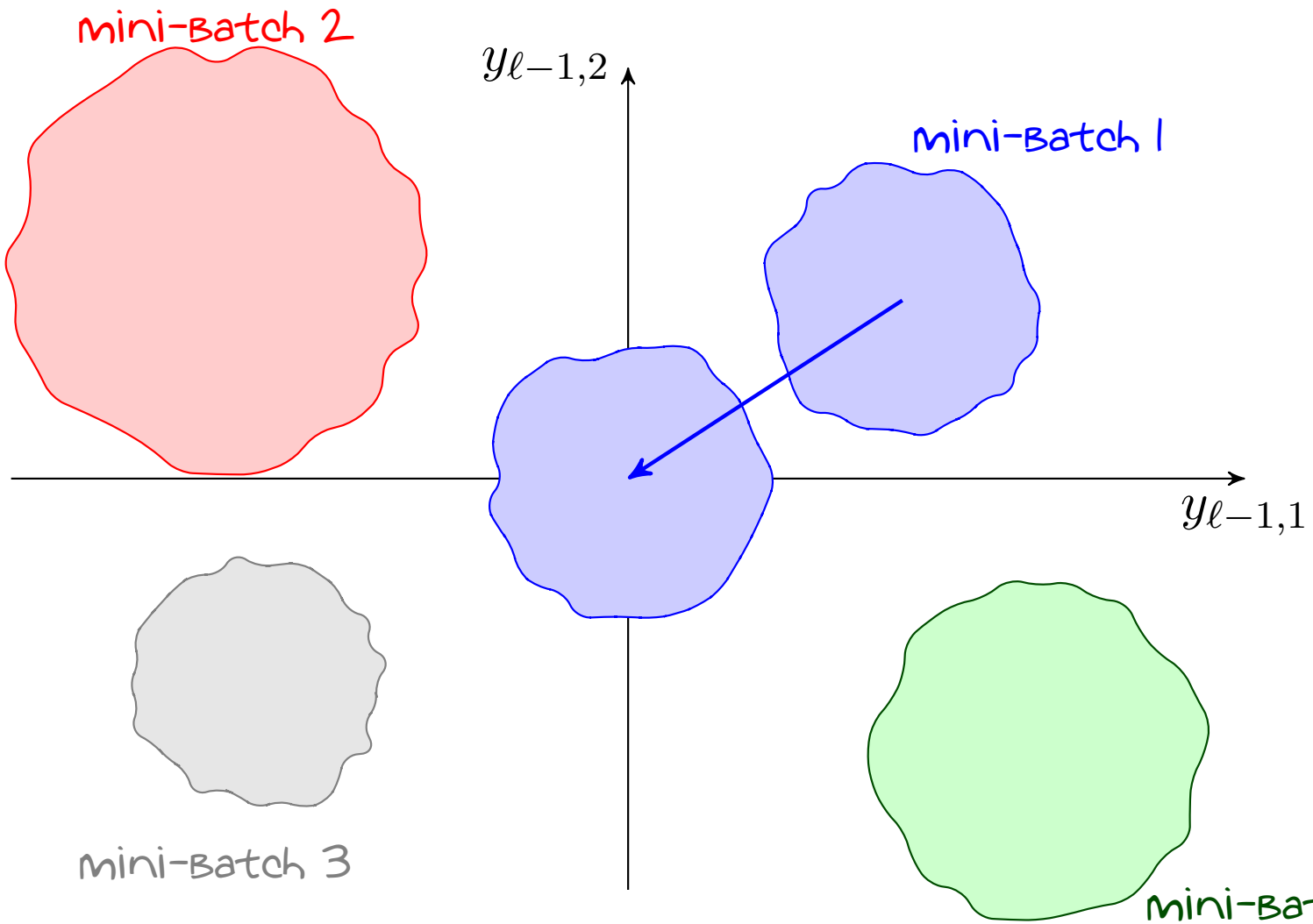
Visualizing *Normalization*

We can visualize *normalization* as below *for two-dimensional inputs*



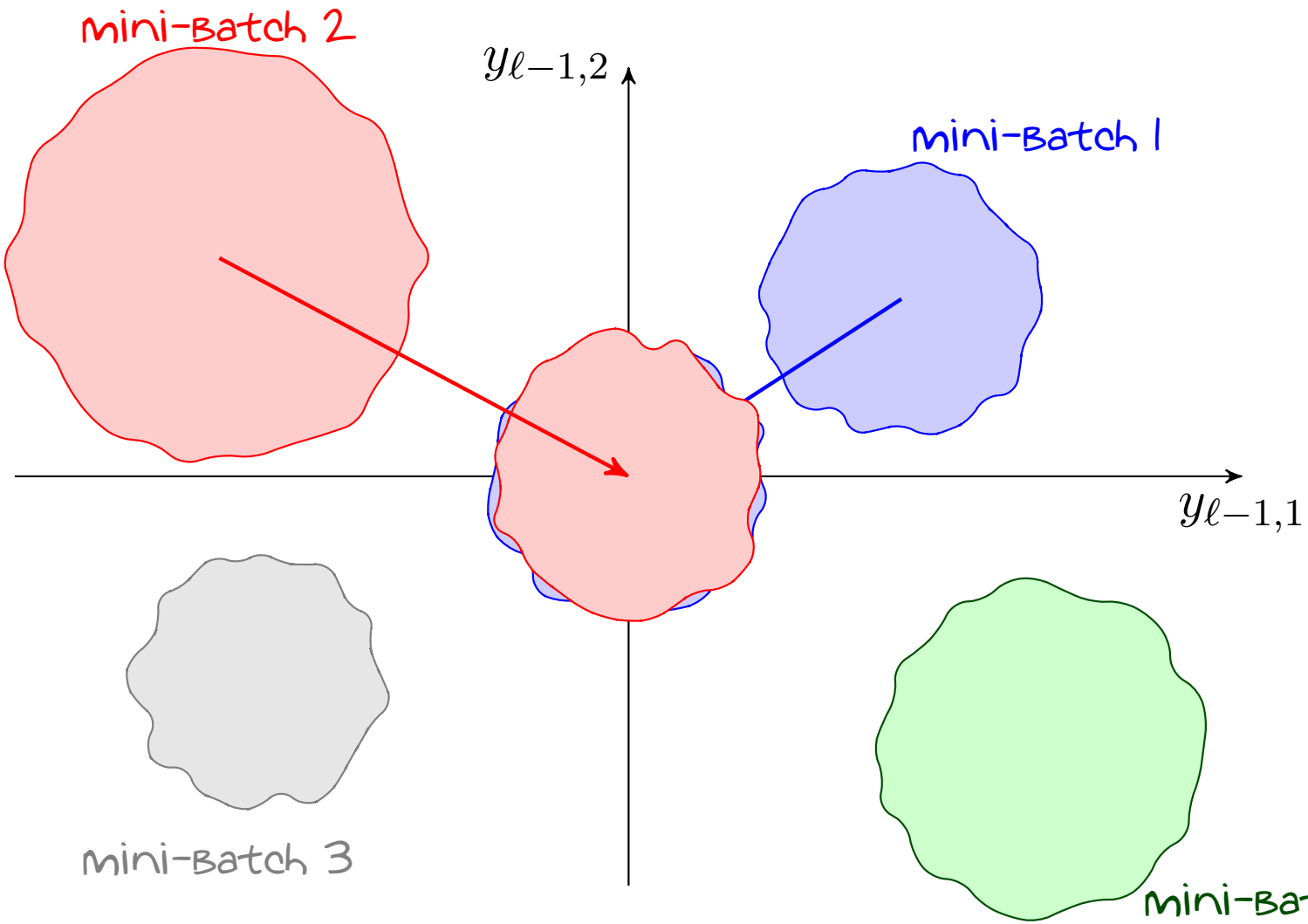
Visualizing *Normalization*

We can visualize *normalization* as below *for two-dimensional inputs*



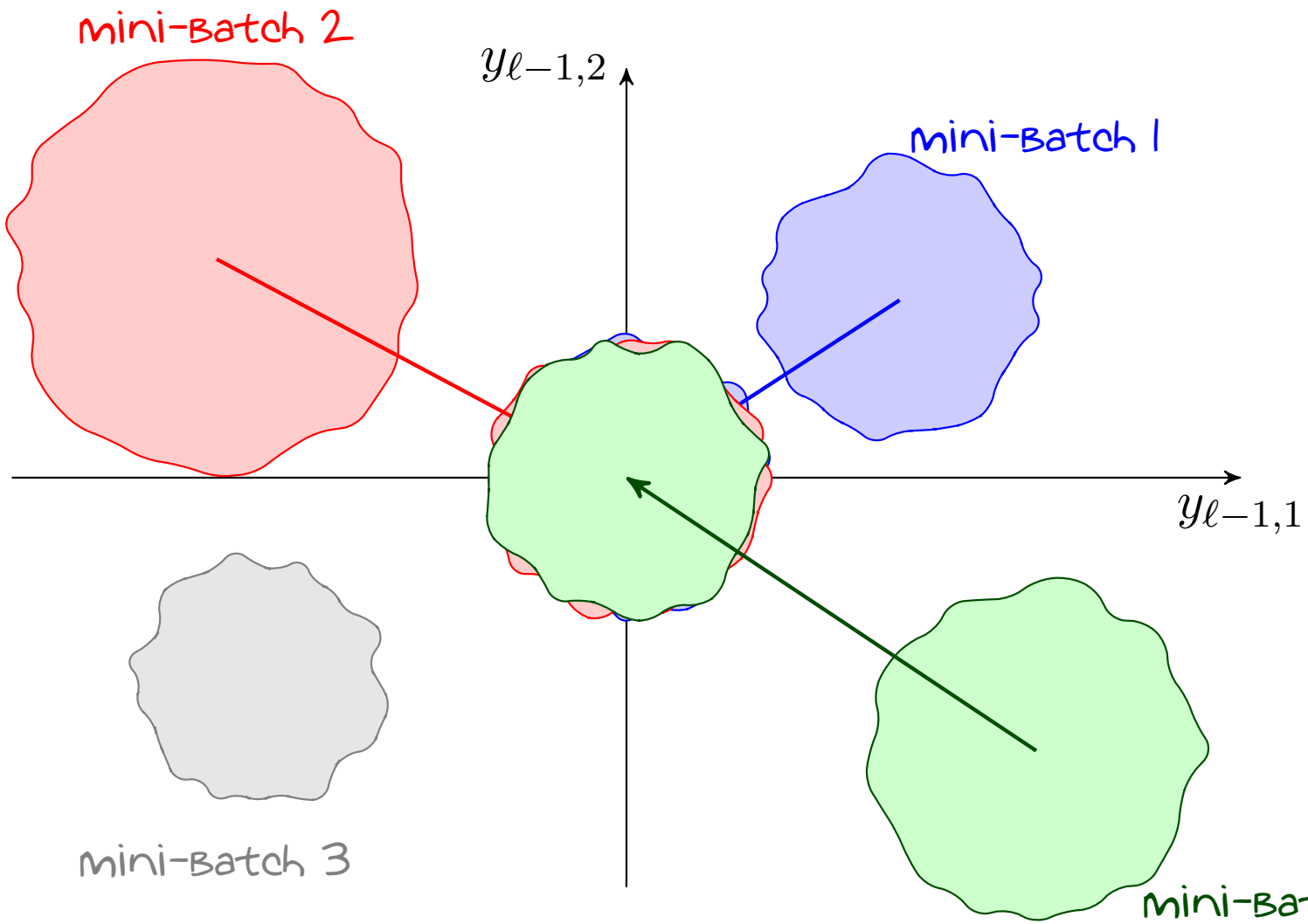
Visualizing *Normalization*

We can visualize *normalization* as below *for two-dimensional inputs*



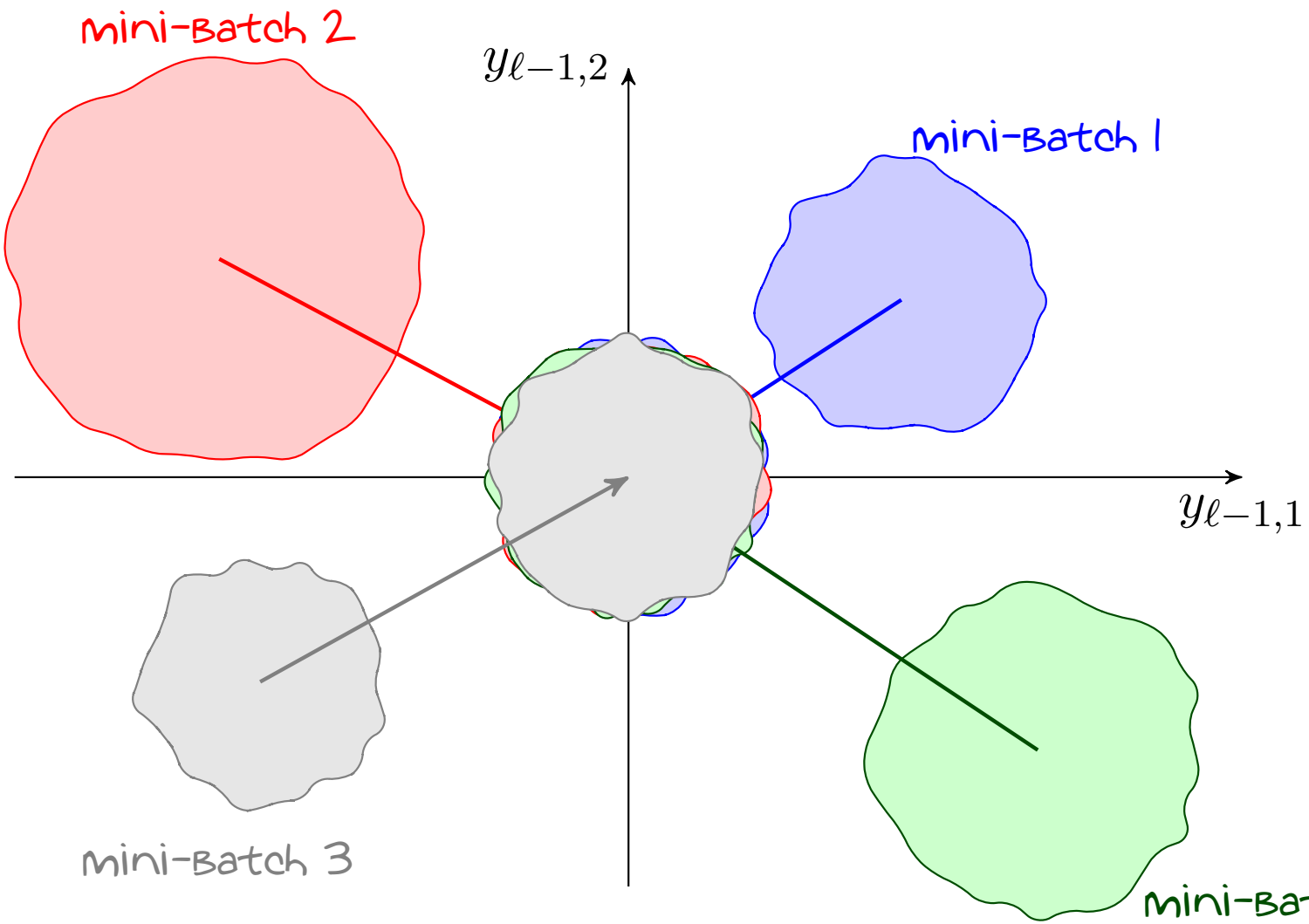
Visualizing *Normalization*

We can visualize *normalization* as below *for two-dimensional inputs*



Visualizing *Normalization*

We can visualize *normalization* as below *for two-dimensional inputs*



Standardization via *Normalization*

$$\bar{\mathbf{y}}_{\ell-1,b} = \frac{\mathbf{y}_{\ell-1,b} - \mathbb{E}\left\{\mathbf{y}_{\ell-1}\right\}}{\sqrt{\mathbb{V}\mathrm{ar}\left\{\mathbf{y}_{\ell-1}\right\}}}$$

+ *How do we compute mean and variance? We don't know data distribution!*
– Well, as always: *we can approximate them from the dataset*

Standardization via *Normalization*

$$\bar{\mathbf{y}}_{\ell-1,b} = \frac{\mathbf{y}_{\ell-1,b} - \mathbb{E}\left\{\mathbf{y}_{\ell-1}\right\}}{\sqrt{\mathbb{V}\mathrm{ar}\left\{\mathbf{y}_{\ell-1}\right\}}}$$

+ *How do we compute mean and variance? We don't know data distribution!*
– Well, as always: *we can approximate them from the dataset*

Say our dataset has B data-points: we approximate mean and variance as

$$\mathbb{E}\left\{\mathbf{y}_{\ell-1}\right\} \approx \frac{1}{B}\sum_{b=1}^{B} \mathbf{y}_{\ell-1,b} \equiv \boldsymbol{\mu}_{\ell-1}$$

$$\mathbb{V}\mathrm{ar}\left\{\mathbf{y}_{\ell-1}\right\} \approx \frac{1}{B}\sum_{b=1}^{B} \left(\mathbf{y}_{\ell-1,b} - \boldsymbol{\mu}_{\ell-1}\right)^2 \equiv \boldsymbol{\sigma}^2_{\ell-1}$$

Let's check this idea for each layer!

Normalization: *Input Layer*

With input layer, i.e., $\ell = 1$, this approximation works well: *we apply normalization just once and work with normalized data from then on, i.e.,*

$$\bar{\boldsymbol{x}}_b = \frac{\boldsymbol{x}_b - \boldsymbol{\mu}_0}{\boldsymbol{\sigma}_0}$$

for $\boldsymbol{\mu}_0$ *and* $\boldsymbol{\sigma}_0$ *that are computed from the dataset as*

$$\boldsymbol{\mu}_0 = \frac{1}{B}\sum_{b=1}^{B} \boldsymbol{x}_b \qquad \boldsymbol{\sigma}_0 = \sqrt{\frac{1}{B}\sum_{b=1}^{B} (\boldsymbol{x}_b - \boldsymbol{\mu}_0)^2}$$

Let's define the operator $\mathcal{U}(\cdot)$ as the *standardizer*, *i.e.*,

$$\bar{\boldsymbol{x}}_b = \mathcal{U}(\boldsymbol{x}_b | \mathbb{D})$$

where $\mathbb{D}$ *is the training dataset*

Forward Propagation with *Input Normalization*

```
ForwardProp():
```
1: Initiate with $\mathbf{y}_0 = \mathcal{U}(\boldsymbol{x}_b|\mathbb{D})$
2: **for** $\ell = 0, \ldots, L$ **do**
3: Add $y_\ell[0] = 1$ and determine $\mathbf{z}_{\ell+1} = \mathbf{W}_{\ell+1}\mathbf{y}_\ell$ `# forward affine`
4: Determine $\mathbf{y}_{\ell+1} = f_{\ell+1}(\mathbf{z}_{\ell+1})$ `# forward activation`
5: **end for**
6: **for** $\ell = 1, \ldots, L+1$ **do**
7: Return $\mathbf{y}_\ell$ and $\mathbf{z}_\ell$
8: **end for**

+ *Shall we do the same for every layer?*
– Well, we could try, *but we end up with computation complexity close to full-batch training!*

Normalization: *Hidden Layers*

At hidden layers, i.e., $\ell > 1$, *the input depends on the wights and biases of previous layer: for instance, after normalizing input we compute*

$$\mathbf{z}_1 = \mathbf{W}_1 \bar{\boldsymbol{x}} \rightsquigarrow \mathbf{y}_1 = f_1(\mathbf{z}_1)$$

if we approximate mean and variance with same approach, we should compute

$$\boldsymbol{\mu}_1 = \frac{1}{B}\sum_{b=1}^{B} \mathbf{y}_{1,b} \qquad \boldsymbol{\sigma}_1 = \sqrt{\frac{1}{B}\sum_{b=1}^{B} (\mathbf{y}_{1,b} - \boldsymbol{\mu}_1)^2}$$

These parameters depend on $\mathbf{W}_1$! This means after each update of weights at the end of each mini-batch, we should repeat this for the entire training dataset!

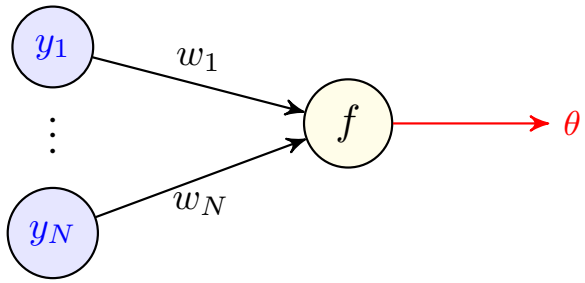
This is not the only issue: *since the normalization of these layers depends on weights, the gradient of loss with respect to weights also changes*

Batch Normalization: *Training with Normalization*

Batch normalization extends the training loop of mini-batch SGD to incorporate also normalization of hidden layers: the idea is simple

- *approximate mean and variance using the mini-batch*
- *compute derivatives by taking normalization into account*

Let's focus first on a *single neuron in a hidden layer*



Say the output neuron has no bias, i.e., θ is given by

$$z = \sum_{n=1}^{N} w_n y_n = \mathbf{w}^\mathsf{T}\mathbf{y} \qquad \rightsquigarrow \qquad \theta = f(z)$$

Also assume we train via mini-batches with batch size Ω

Recap: *Basic Forward and Backward Pass*

Without batch normalization, we pass forward $\mathbf{y}_\omega$ for every $\omega = 1, \ldots, \Omega$

- *by first computing the affine transform*

$$z_\omega = \mathbf{w}^\mathsf{T} \mathbf{y}_\omega$$

- *then activating as* $\theta_\omega = f\left(z_\omega\right)$

Once we get to the output: *we backpropagate by*

- *first computing derivative with respect to* z_ω*, i.e.,*

$$\frac{\mathrm{d}}{\mathrm{d}z_\omega}\mathcal{L} = \left(\frac{\mathrm{d}}{\mathrm{d}\theta_\omega}\mathcal{L}\right) \dot{f}\left(z_\omega\right)$$

- *and then tracking back to* $\mathbf{y}_\omega$*, i.e.,*

$$\nabla_{\mathbf{y}}\mathcal{L} = \mathbf{w}\left(\frac{\mathrm{d}}{\mathrm{d}z_\omega}\mathcal{L}\right)$$

Batch Normalization: *Forward Pass*

With batch normalization *in forward pass, we wait till the whole mini-batch is over: wait till we have* $\mathbf{y}_\omega$ *for* $\omega = 1, \ldots, \Omega$. *We then*

- *approximate mean and variance as*

$$\boldsymbol{\mu} = \frac{1}{\Omega} \sum_{\omega=1}^{\Omega} \mathbf{y}_\omega \qquad \boldsymbol{\sigma} = \sqrt{\frac{1}{\Omega} \sum_{\omega=1}^{\Omega} (\mathbf{y}_\omega - \boldsymbol{\mu})^2}$$

- *normalize* $\mathbf{y}_\omega$ *for* $\omega = 1, \ldots, \Omega$ *as*

$$\mathbf{u}_\omega = \frac{\mathbf{y}_\omega - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

$\rightarrow$ *scale and shift to a common place*

$$\bar{\mathbf{y}}_\omega = \boldsymbol{\gamma} \odot \mathbf{u}_\omega + \boldsymbol{\beta}$$

- *compute the affine transforms* $z_\omega = \mathbf{w}^\mathsf{T} \bar{\mathbf{y}}_\omega$ *and activate as* $\theta_\omega = f(z_\omega)$

Batch Normalization: *Learnable Shift and Scale*

+ *Why do we scale and shift after normalization?*
– This is not guaranteed that center with unit variance is the best place: *we can learn the best place through training!*

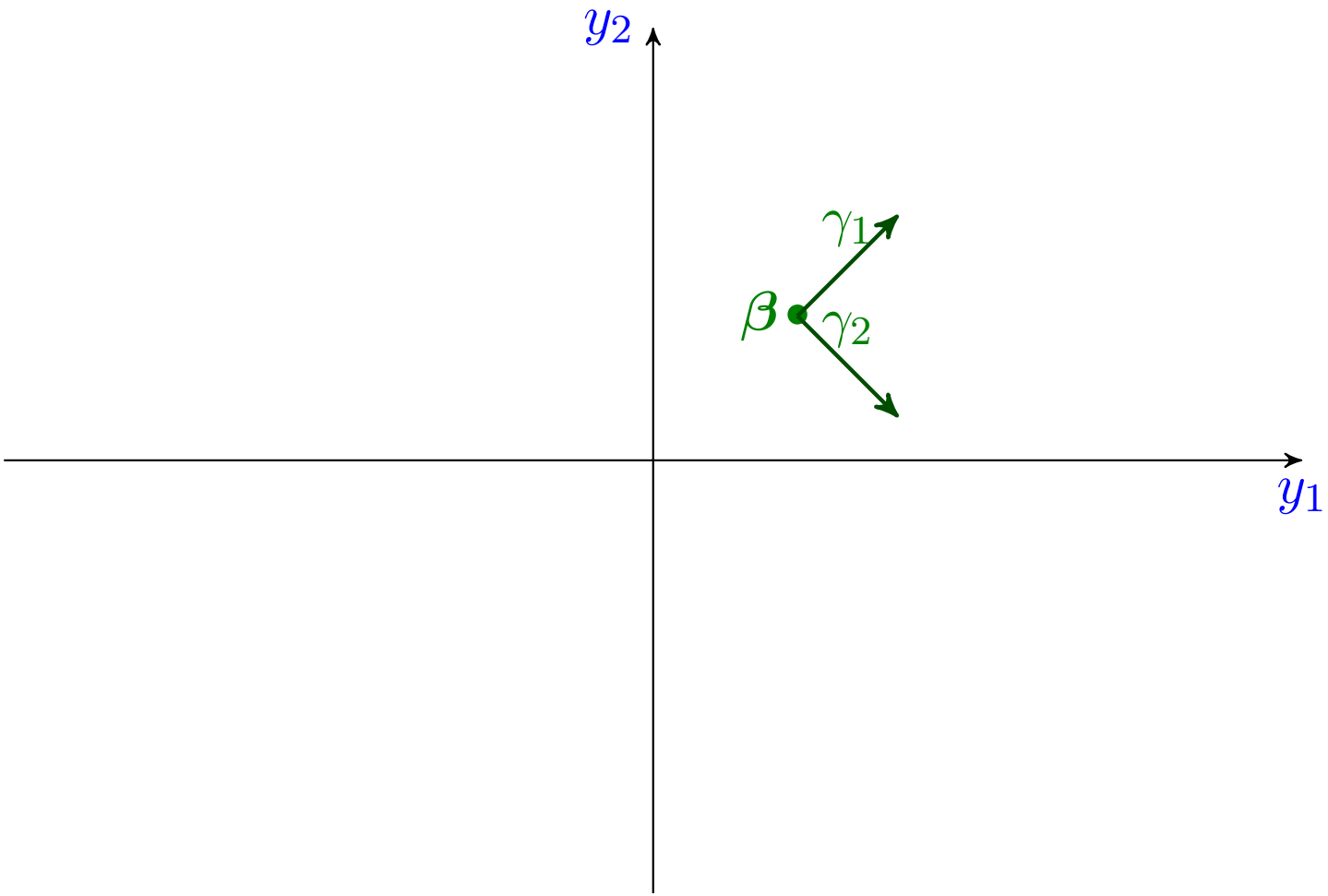
Depending on activation, it might be better *to center all features at another place with some different variances*

- *It seems hard to engineer this point and the variance*
- *We hence introduce a general point* $\boldsymbol{\beta}, \boldsymbol{\gamma} \in \mathbb{R}^N$
  - ↳ *We treat these vectors as learnable parameters*
  - ↳ *We update them in addition to weight matrices* $\mathbf{W}_\ell$ *in backward pass*

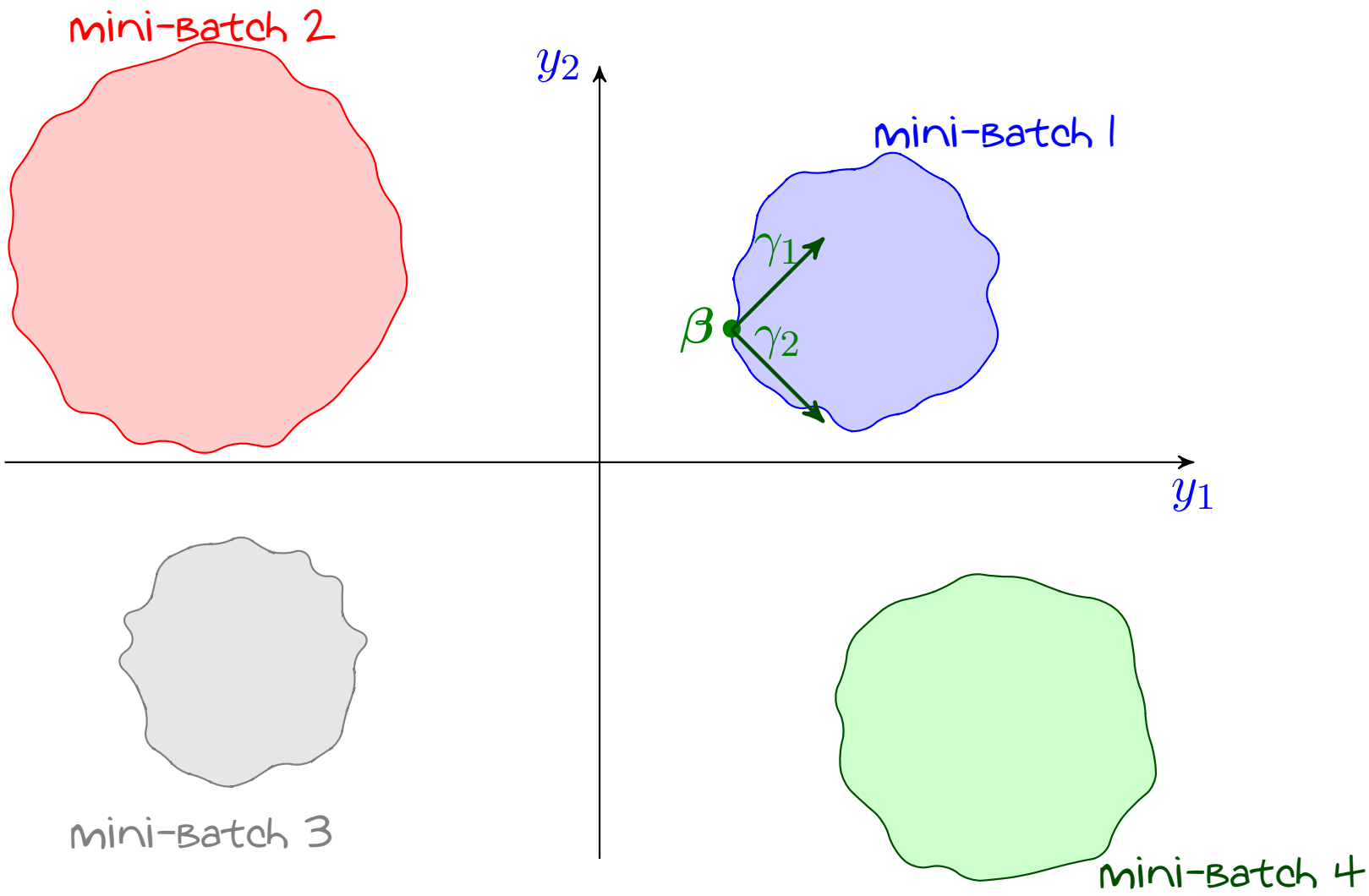
We denote this operation by

$$\mathcal{B}_{\mathcal{N}}\left(\mathbf{u}|\boldsymbol{\gamma}, \boldsymbol{\beta}\right) = \boldsymbol{\gamma} \odot \mathbf{u} + \boldsymbol{\beta}$$

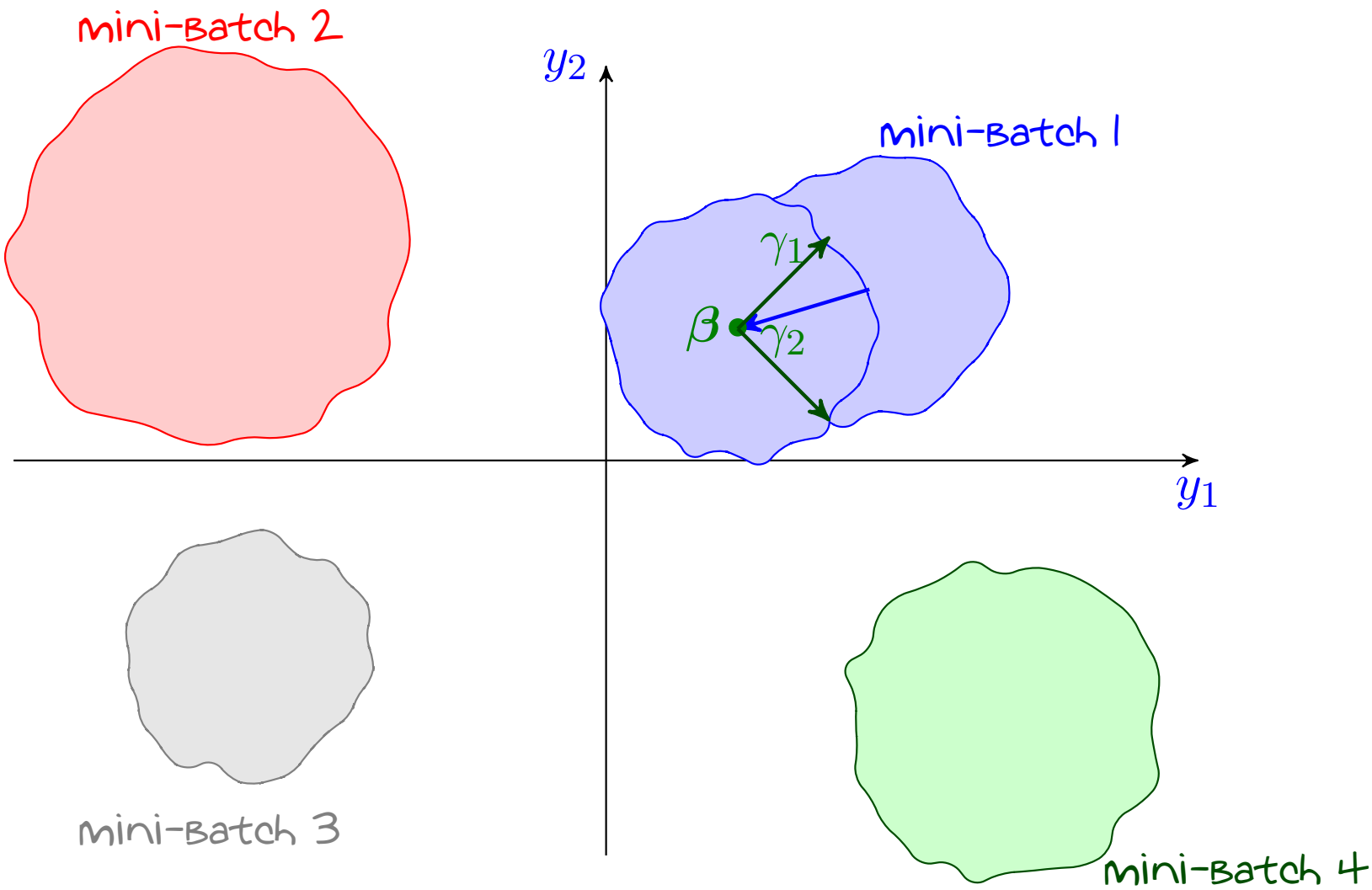
Batch Normalization: *Learnable Shift and Scale*



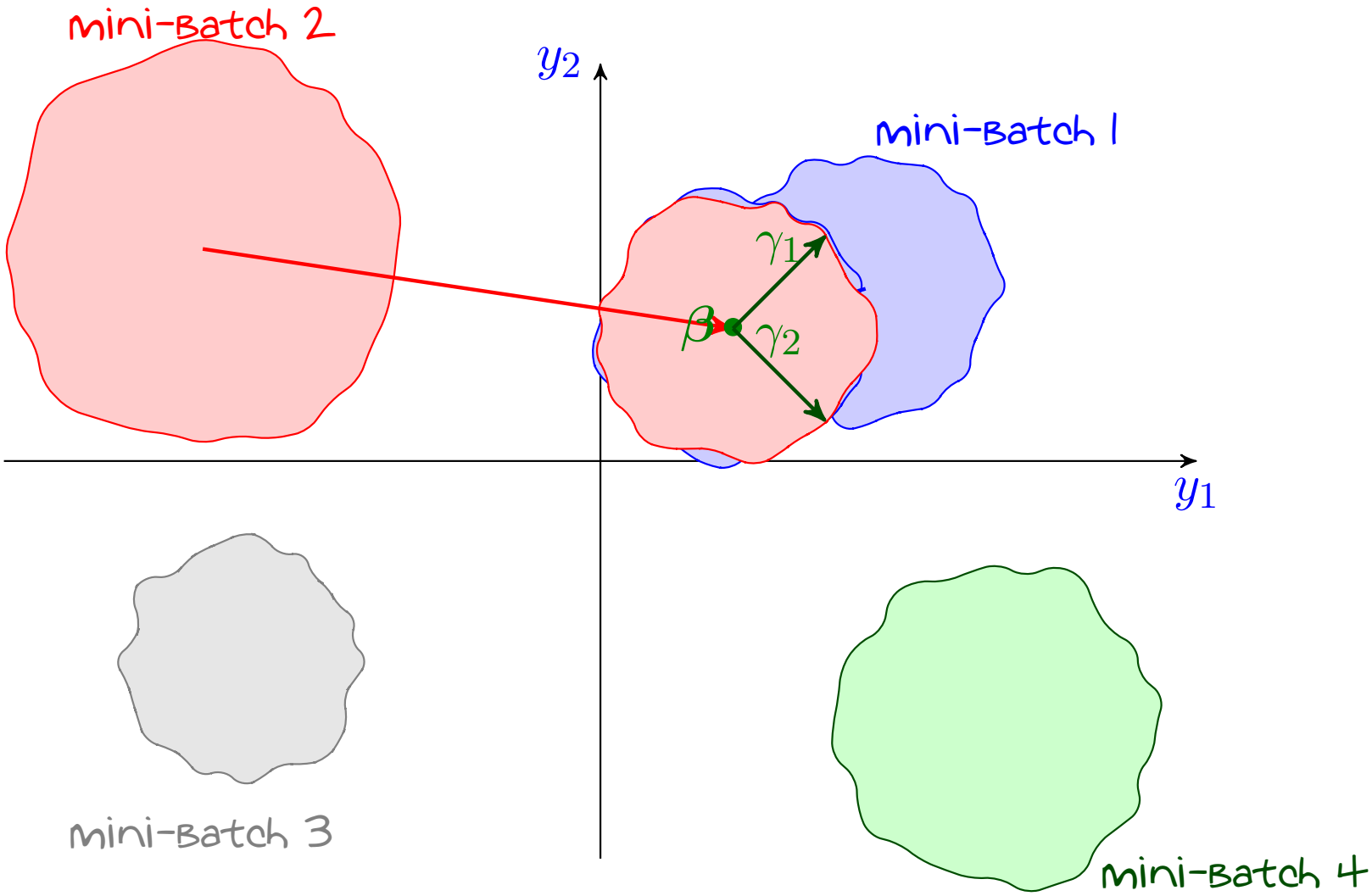
Batch Normalization: *Learnable Shift and Scale*



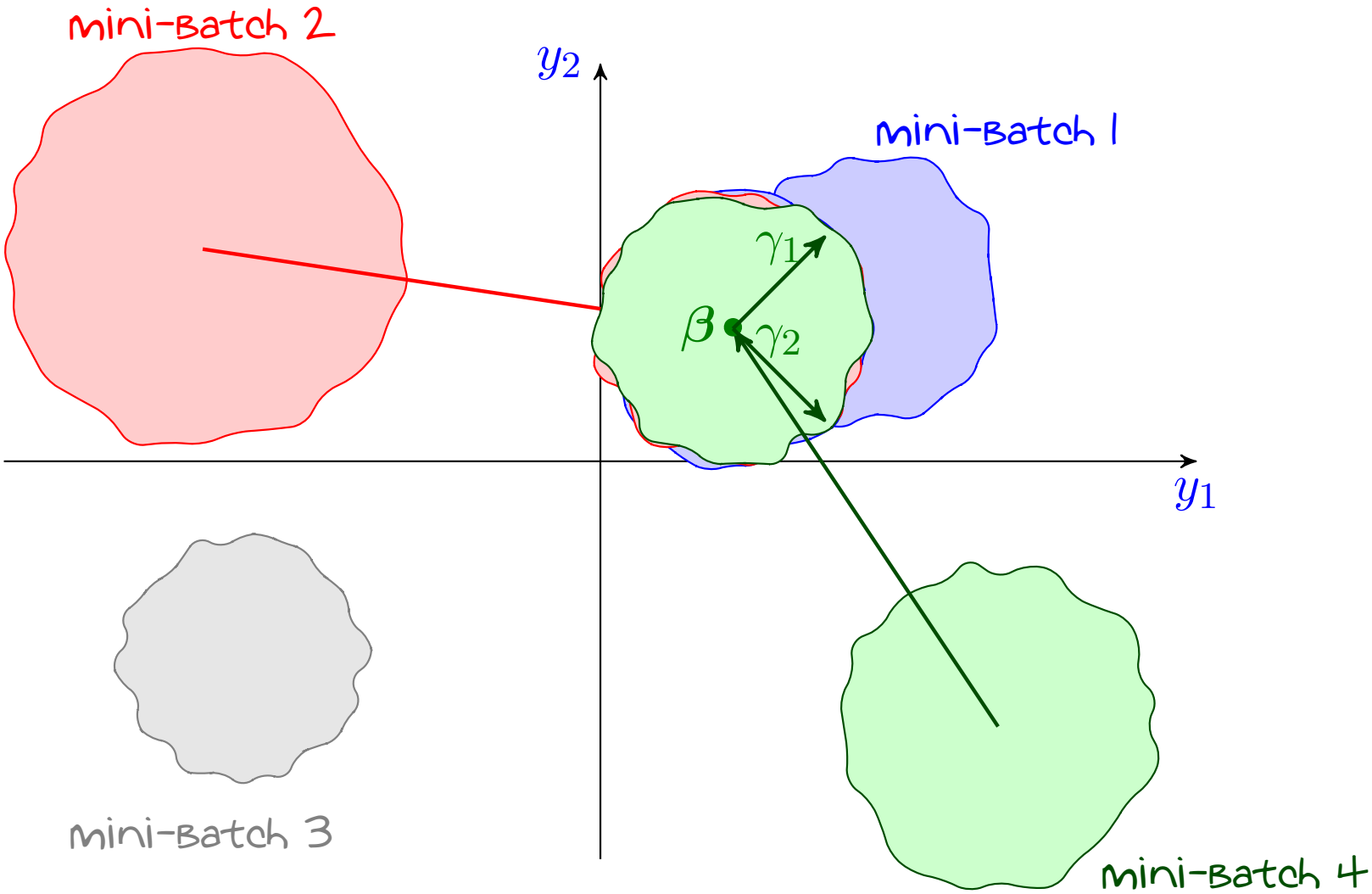
Batch Normalization: *Learnable Shift and Scale*



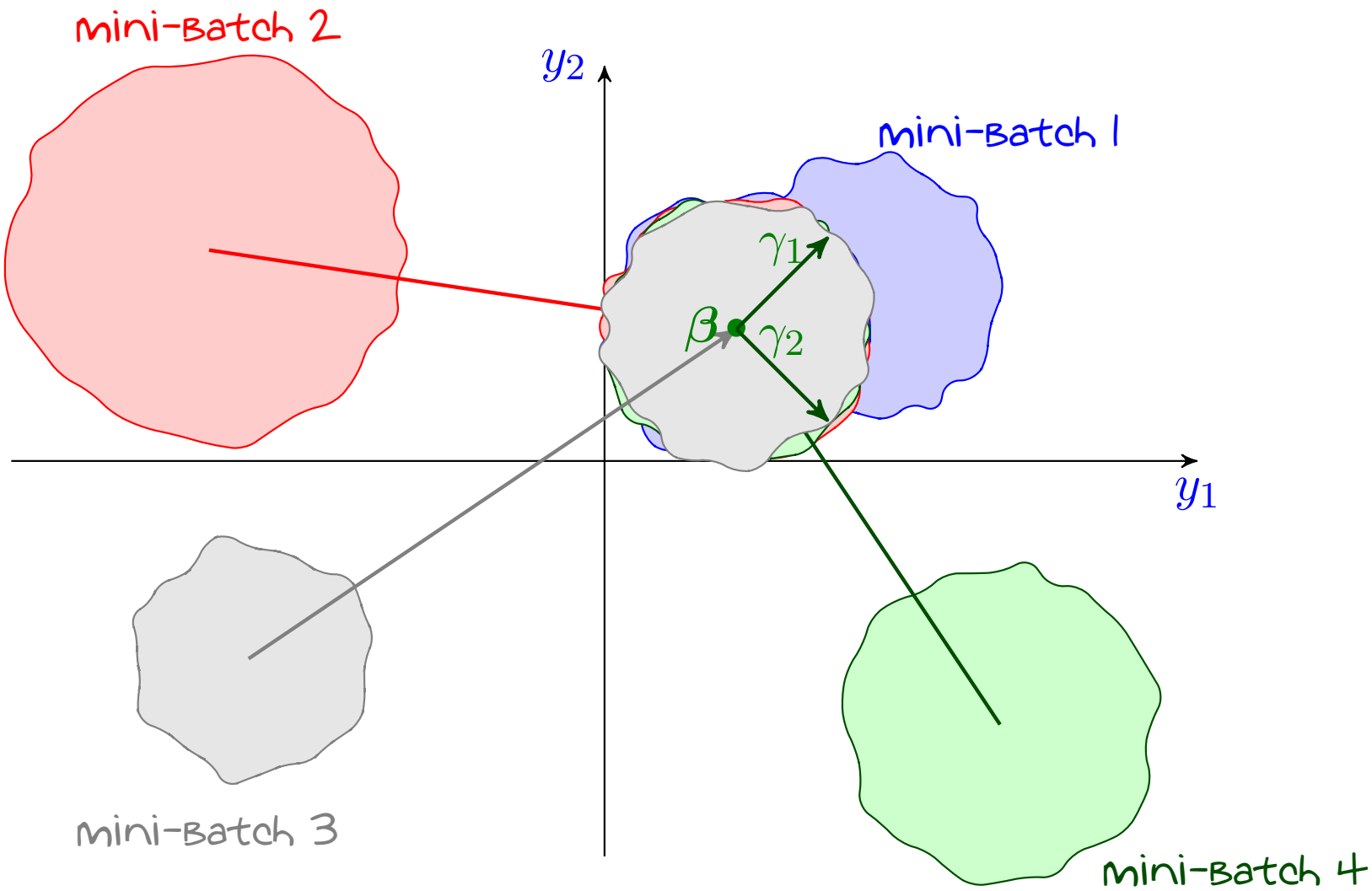
Batch Normalization: *Learnable Shift and Scale*



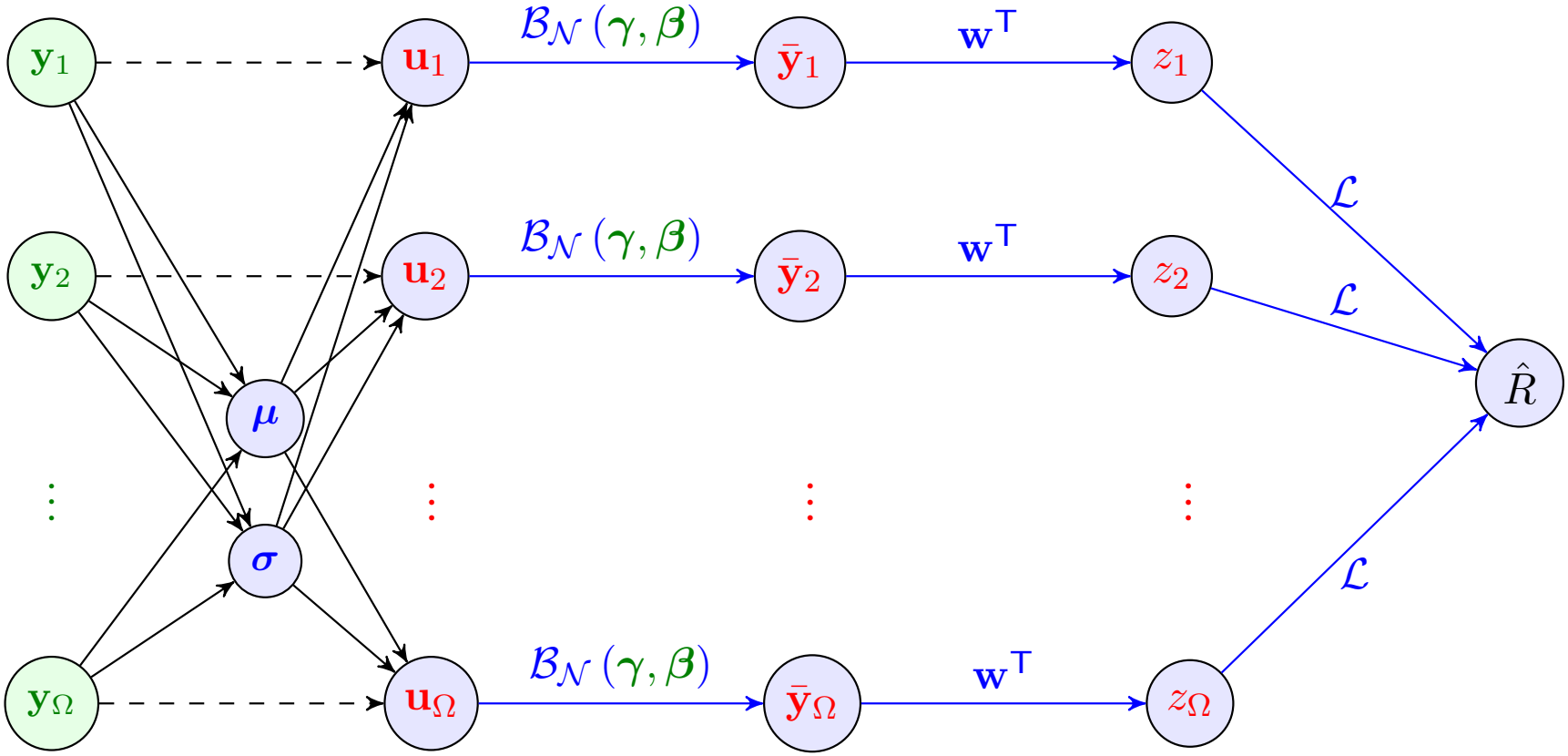
Batch Normalization: *Learnable Shift and Scale*



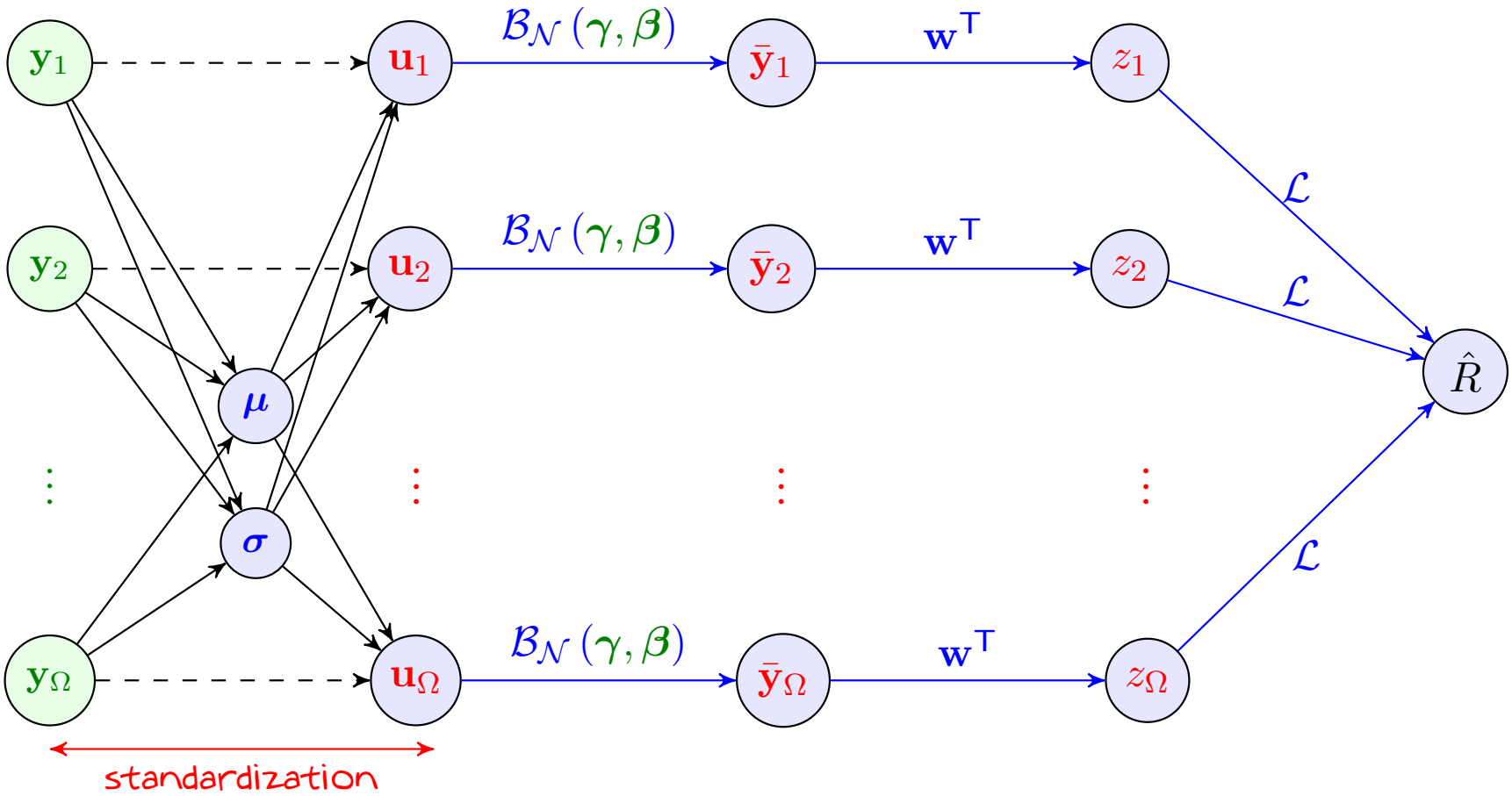
Batch Normalization: *Learnable Shift and Scale*



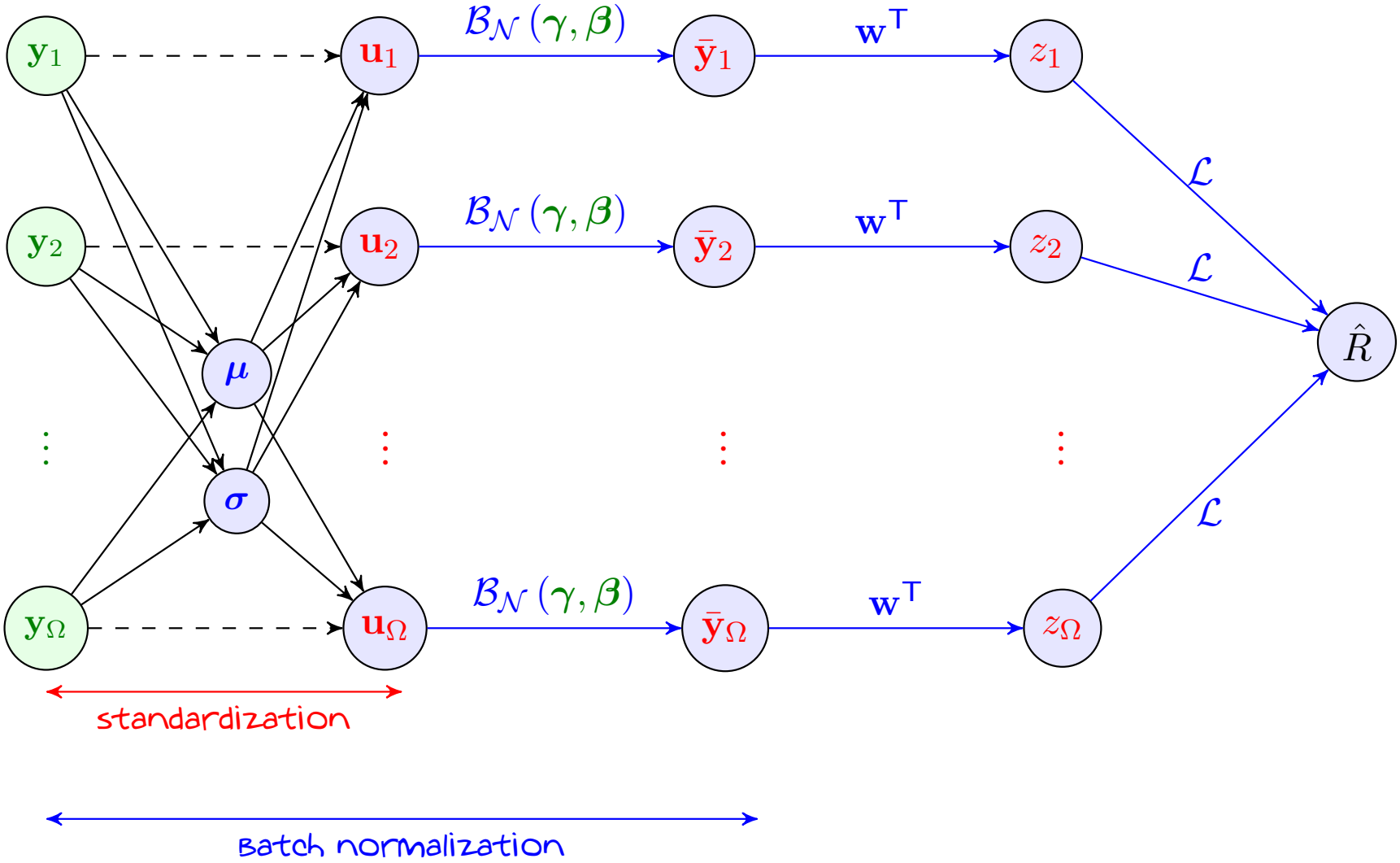
Batch Normalization: *Computation Graph*



Batch Normalization: *Computation Graph*



Batch Normalization: *Computation Graph*



Batch Normalization: *Backpropagation*

Now assume that *forward pass* *is over for the complete mini-batch*

we could easily get back to normalized variables, i.e., $\bar{\mathbf{y}}_\omega$

First, we note that *our empirical risk reads*

$$\hat{R} = \frac{1}{\Omega} \sum_{\omega=1}^{\Omega} \underbrace{\mathcal{L}\left(\theta_\omega, v_\omega\right)}_{\hat{R}_\omega}$$

Batch Normalization: *Backpropagation*

Now assume that *forward pass* *is over for the complete mini-batch*

we could easily get back to normalized variables, i.e., $\bar{\mathbf{y}}_\omega$

First, we note that *our empirical risk reads*

$$\hat{R} = \frac{1}{\Omega} \sum_{\omega=1}^{\Omega} \underbrace{\mathcal{L}\left(\theta_\omega, v_\omega\right)}_{\hat{R}_\omega} = \frac{1}{\Omega} \sum_{\omega=1}^{\Omega} \hat{R}_\omega$$

Batch Normalization: *Backpropagation*

Now assume that *forward pass* is over for the *complete mini-batch*

we could easily get back to normalized variables, i.e., $\bar{\mathbf{y}}_\omega$

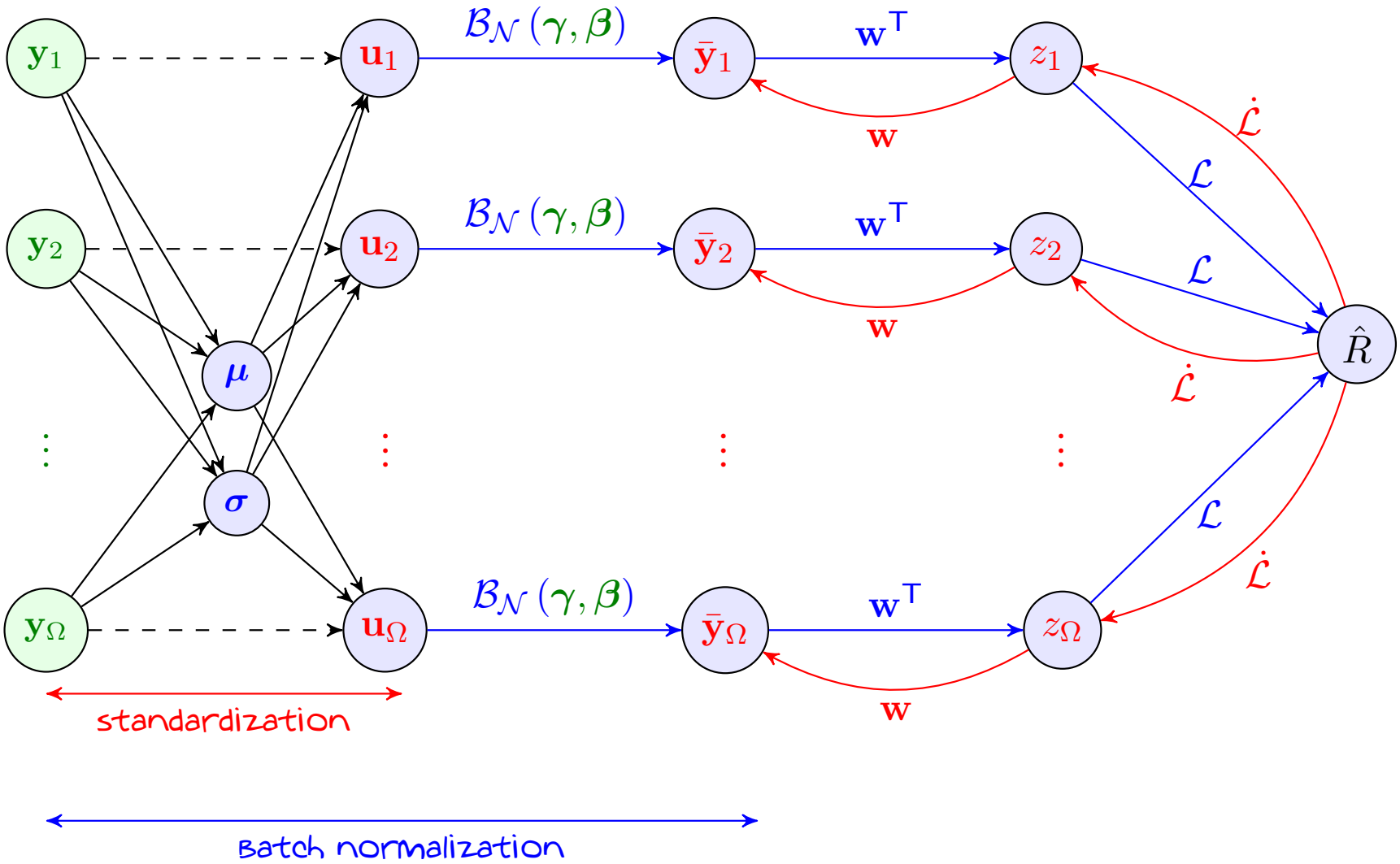
First, we note that *our empirical risk reads*

$$\hat{R} = \frac{1}{\Omega} \sum_{\omega=1}^{\Omega} \underbrace{\mathcal{L}\left(\theta_\omega, v_\omega\right)}_{\hat{R}_\omega} = \frac{1}{\Omega} \sum_{\omega=1}^{\Omega} \hat{R}_\omega$$

- *We compute* $\nabla_{z_\omega} \hat{R}_\omega$ *using* $\dot{\mathcal{L}}\left(\cdot\right)$ *and* $\dot{f}\left(\cdot\right)$
- *We compute* $\nabla_{\bar{\mathbf{y}}_\omega} \hat{R}_\omega$ *from* $\nabla_{z_\omega} \hat{R}_\omega$
  - ↳ *We just need to apply standard backward pass of linear transforms*

$$\nabla_{\bar{\mathbf{y}}_\omega} \hat{R}_\omega = \left(\nabla_{z_\omega} \hat{R}_\omega\right) \mathbf{w} = \left(\frac{\mathrm{d}}{\mathrm{d} z_\omega} \hat{R}_\omega\right) \mathbf{w}$$

Batch Normalization: *Backpropagation*



Batch Normalization: *Backpropagation*

By now, *we have* $\nabla_{\bar{\mathbf{y}}_\omega} \hat{R}_\omega$ *for* $\omega = 1, \ldots, \Omega$; next, we move backward

from scaled and shifted variables to standard ones

Batch Normalization: *Backpropagation*

By now, *we have* $\nabla_{\bar{\mathbf{y}}_\omega} \hat{R}_\omega$ *for* $\omega = 1, \ldots, \Omega$; next, we move backward

from scaled and shifted variables to standard ones

$\mathcal{B}_{\mathcal{N}}(\boldsymbol{\gamma}, \boldsymbol{\beta})$ *scales and shifts entry-wise*

$$\begin{bmatrix} \bar{y}_{1,\omega} \\ \vdots \\ \bar{y}_{N,\omega} \end{bmatrix} = \begin{bmatrix} \gamma_1 u_{1,\omega} + \beta_1 \\ \vdots \\ \gamma_N u_{N,\omega} + \beta_N \end{bmatrix}$$

Batch Normalization: *Backpropagation*

By now, *we have* $\nabla_{\bar{\mathbf{y}}_\omega} \hat{R}_\omega$ *for* $\omega = 1, \ldots, \Omega$; next, we move backward

from scaled and shifted variables to standard ones

$\mathcal{BN}(\boldsymbol{\gamma}, \boldsymbol{\beta})$ scales and shifts entry-wise

$$\begin{bmatrix} \bar{y}_{1,\omega} \\ \vdots \\ \bar{y}_{N,\omega} \end{bmatrix} = \begin{bmatrix} \gamma_1 u_{1,\omega} + \beta_1 \\ \vdots \\ \gamma_N u_{N,\omega} + \beta_N \end{bmatrix}$$

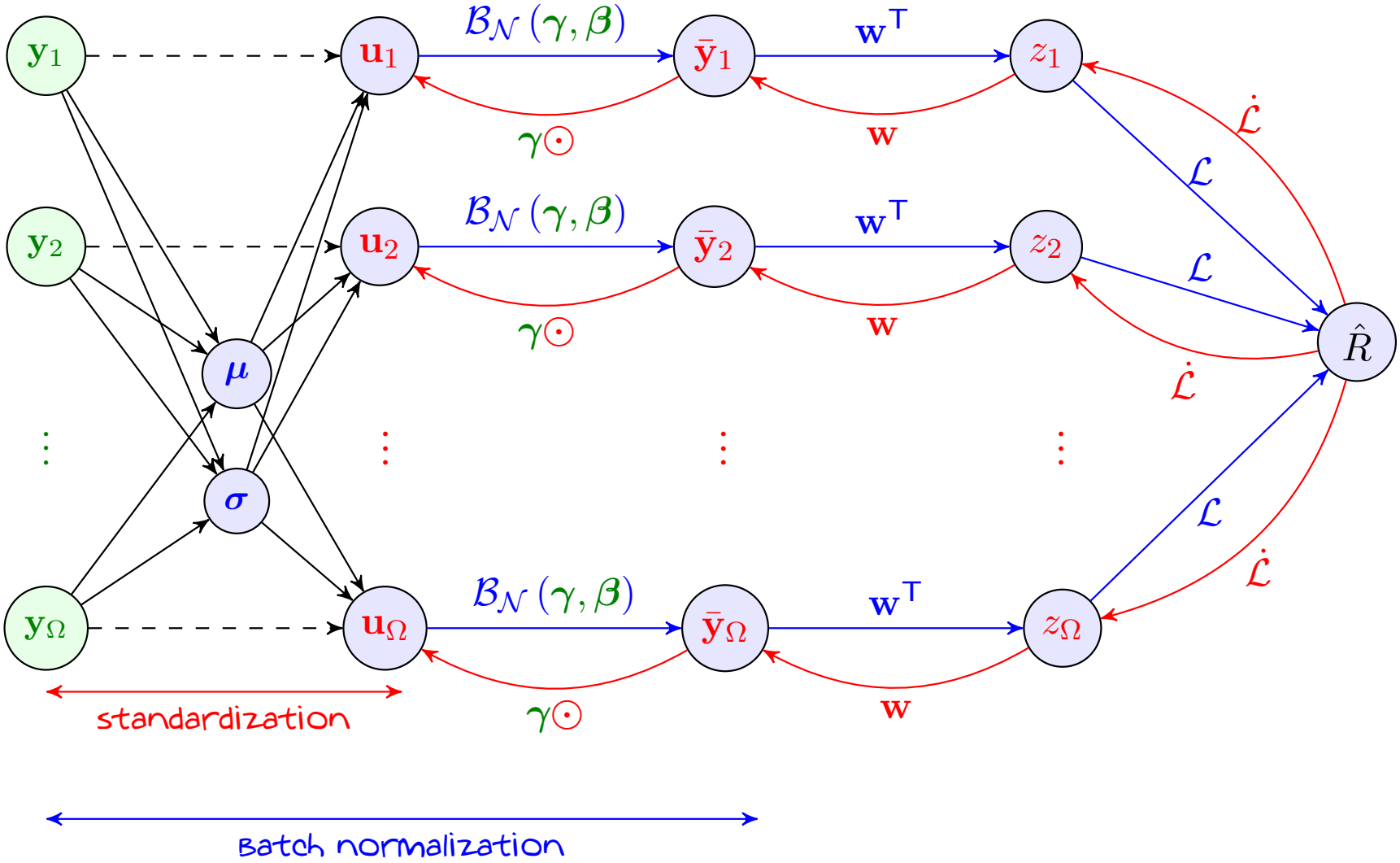
So, we can say

$$\frac{\partial}{\partial u_{i,\omega}} \hat{R}_\omega = \gamma_i \frac{\partial}{\partial \bar{y}_{i,\omega}} \hat{R}_\omega$$

The backward pass is hence for $\omega = 1, \ldots, \Omega$ *is*

$$\nabla_{\mathbf{u}_\omega} \hat{R}_\omega = \boldsymbol{\gamma} \odot \nabla_{\bar{\mathbf{y}}_\omega} \hat{R}_\omega$$

Batch Normalization: *Backpropagation*



Batch Normalization: *Backpropagation*

At this point, we can also *compute the gradients with respect to* γ *and* β

For $\omega = 1, \ldots, \Omega$, *we have*

$$\nabla_{\gamma} \hat{R}_{\omega} = \mathbf{u}_{\omega} \odot \nabla_{\bar{\mathbf{y}}_{\omega}} \hat{R}_{\omega}$$

$$\nabla_{\beta} \hat{R}_{\omega} = \nabla_{\bar{\mathbf{y}}_{\omega}} \hat{R}_{\omega}$$

Batch Normalization: *Backpropagation*

At this point, we can also *compute the gradients with respect to* γ *and* β

For $\omega = 1, \ldots, \Omega$, *we have*

$$\nabla_{\gamma} \hat{R}_{\omega} = \mathbf{u}_{\omega} \odot \nabla_{\bar{\mathbf{y}}_{\omega}} \hat{R}_{\omega}$$
$$\nabla_{\beta} \hat{R}_{\omega} = \nabla_{\bar{\mathbf{y}}_{\omega}} \hat{R}_{\omega}$$

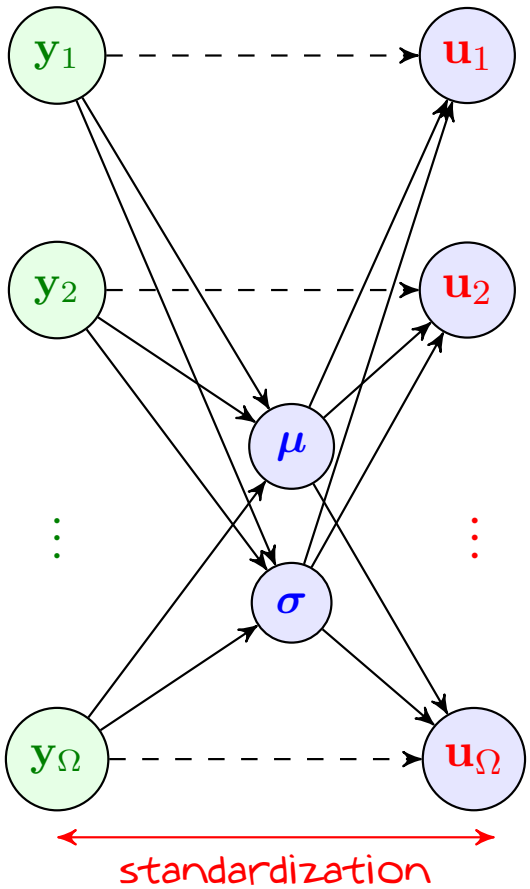
In our optimizer, we then update γ and β using $\nabla_{\gamma} \hat{R}_{\omega}$ and $\nabla_{\beta} \hat{R}_{\omega}$: *for instance with standard SGD we have*

$$\gamma^{(t+1)} = \gamma^{(t)} - \frac{\eta}{\Omega} \sum_{\omega=1}^{\Omega} \nabla_{\gamma} \hat{R}_{\omega}$$

$$\beta^{(t+1)} = \beta^{(t)} - \frac{\eta}{\Omega} \sum_{\omega=1}^{\Omega} \nabla_{\beta} \hat{R}_{\omega}$$

Batch Normalization: *Backpropagation*

The most challenging block is standardization: we open expand everything

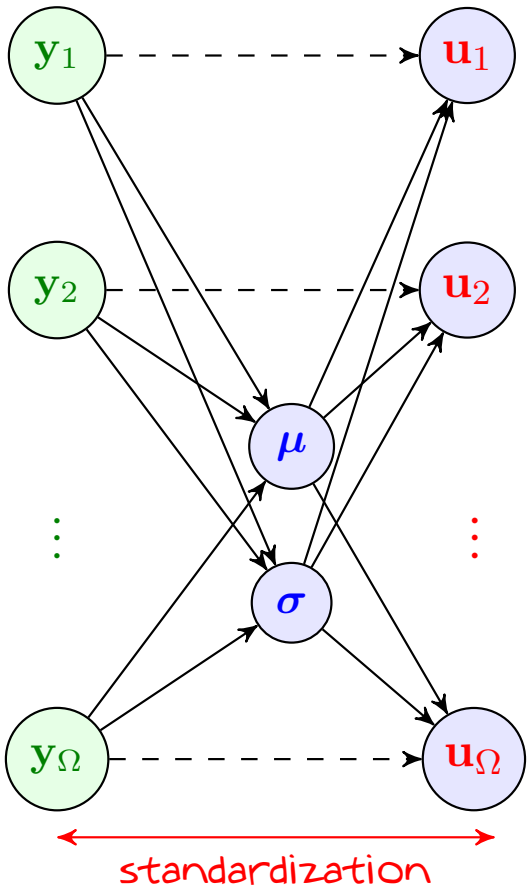


Let's write again forward pass

$$\mathbf{u}_\omega = \frac{\mathbf{y}_\omega - \boldsymbol{\mu}\left(\mathbf{y}_1, \ldots, \mathbf{y}_\Omega\right)}{\boldsymbol{\sigma}\left(\mathbf{y}_1, \ldots, \mathbf{y}_\Omega\right)}$$

Batch Normalization: *Backpropagation*

The most challenging block is standardization: we open expand everything



Let's write again forward pass

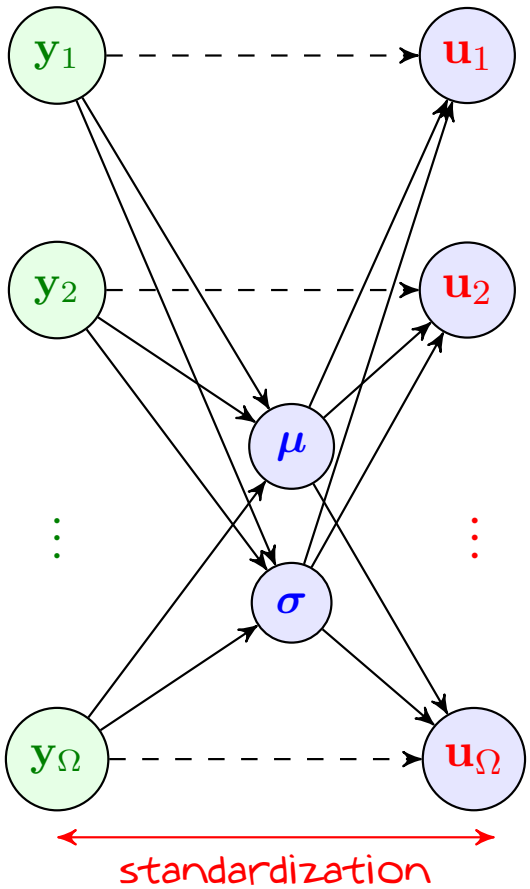
$$\mathbf{u}_\omega = \frac{\mathbf{y}_\omega - \boldsymbol{\mu}\left(\mathbf{y}_1, \ldots, \mathbf{y}_\Omega\right)}{\boldsymbol{\sigma}\left(\mathbf{y}_1, \ldots, \mathbf{y}_\Omega\right)}$$

Here, any $\hat{R}_\tau$ depends on all $\mathbf{y}_\omega$'s, i.e.,

$$\hat{R}_\tau\left(\mathbf{y}_1, \ldots, \mathbf{y}_\Omega\right)$$

Batch Normalization: *Backpropagation*

The most challenging block is standardization: we open expand everything



Let's write again forward pass

$$\mathbf{u}_\omega = \frac{\mathbf{y}_\omega - \boldsymbol{\mu}\left(\mathbf{y}_1, \ldots, \mathbf{y}_\Omega\right)}{\boldsymbol{\sigma}\left(\mathbf{y}_1, \ldots, \mathbf{y}_\Omega\right)}$$

Here, any $\hat{R}_\tau$ depends on all $\mathbf{y}_\omega$'s, i.e.,

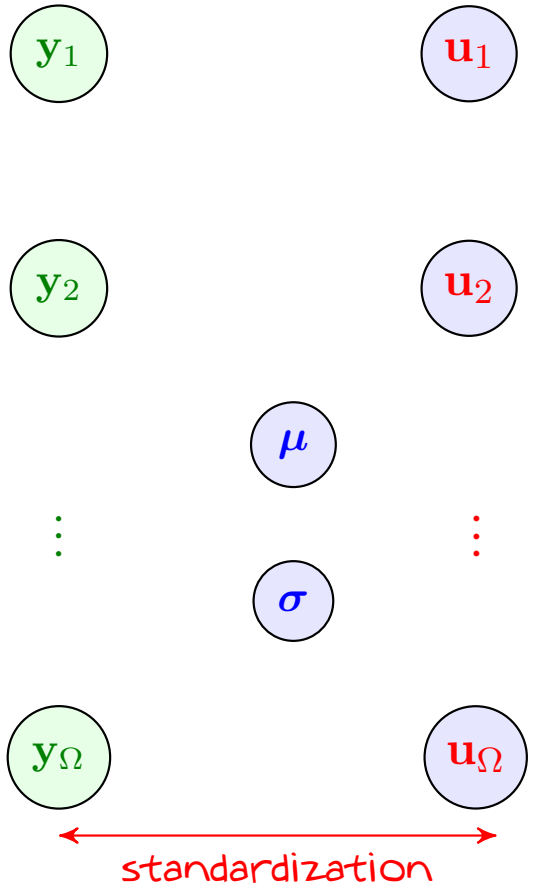
$$\hat{R}_\tau\left(\mathbf{y}_1, \ldots, \mathbf{y}_\Omega\right)$$

We now focus on a single entry of $\mathbf{y}_\omega$

$$\frac{\partial \hat{R}_\tau}{\partial y_{i,\omega}} = \frac{\partial \hat{R}_\tau}{\partial u_{i,\tau}} \frac{\partial u_{i,\tau}}{\partial y_{i,\omega}}$$

since all operations are entry-wise

Batch Normalization: *Backpropagation*

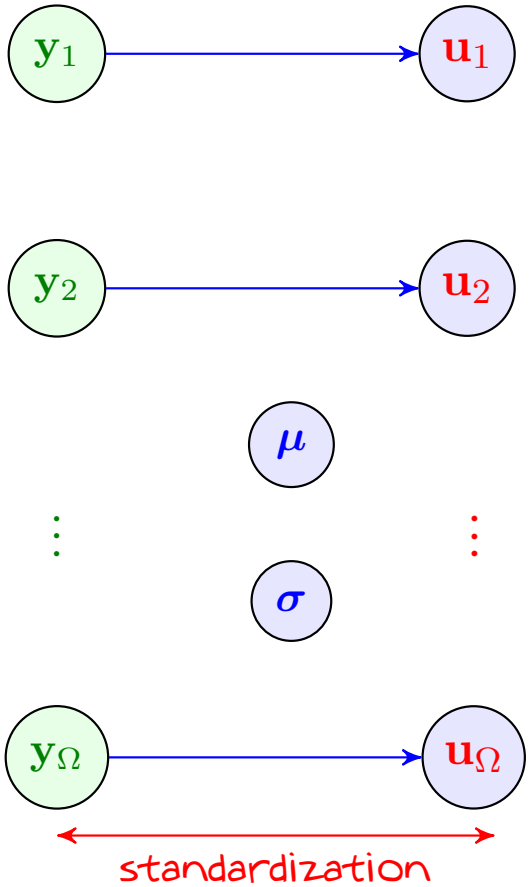


$$u_{i,\tau} = \frac{y_{i,\omega} - \mu_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}{\sigma_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}$$

If we set $\tau = \omega$; then, we have

$$\frac{\partial u_{i,\omega}}{\partial y_{i,\omega}} =$$

Batch Normalization: *Backpropagation*

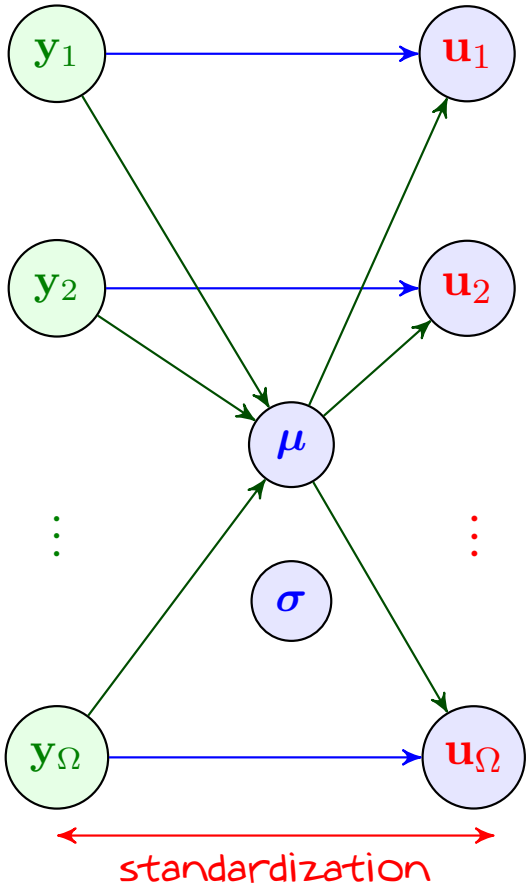


$$u_{i,\tau} = \frac{y_{i,\omega} - \mu_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}{\sigma_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}$$

If we set $\tau = \omega$; then, we have

$$\frac{\partial u_{i,\omega}}{\partial y_{i,\omega}} = \frac{1}{\sigma_i}$$

Batch Normalization: *Backpropagation*

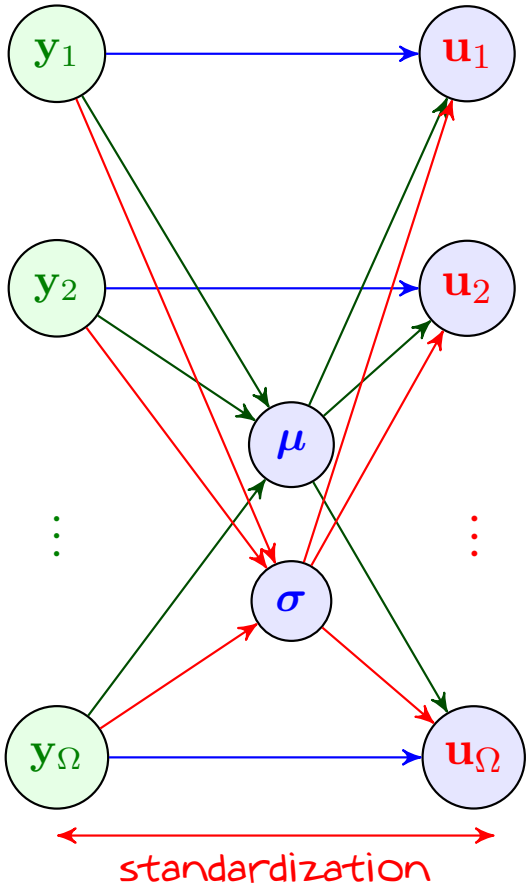


$$u_{i,\tau} = \frac{y_{i,\omega} - \mu_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}{\sigma_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}$$

If we set $\tau = \omega$; then, we have

$$\frac{\partial u_{i,\omega}}{\partial y_{i,\omega}} = \frac{1}{\sigma_i} + \frac{\partial u_{i,\omega}}{\partial \mu_i}\frac{\partial \mu_i}{\partial y_{i,\omega}}$$

Batch Normalization: *Backpropagation*

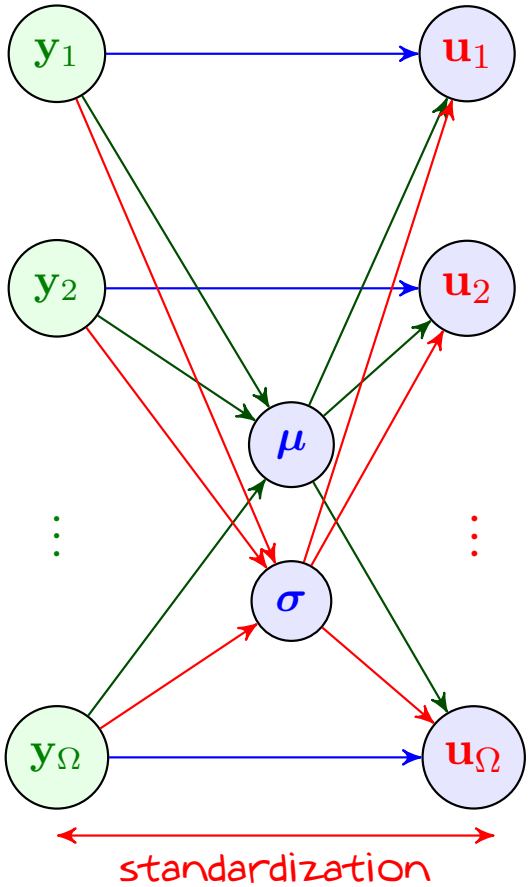


$$u_{i,\tau} = \frac{y_{i,\omega} - \mu_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}{\sigma_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}$$

If we set $\tau = \omega$; then, we have

$$\frac{\partial u_{i,\omega}}{\partial y_{i,\omega}} = \frac{1}{\sigma_i} + \frac{\partial u_{i,\omega}}{\partial \mu_i}\frac{\partial \mu_i}{\partial y_{i,\omega}} + \frac{\partial u_{i,\omega}}{\partial \sigma_i}\frac{\partial \sigma_i}{\partial y_{i,\omega}}$$

Batch Normalization: *Backpropagation*



$$u_{i,\tau} = \frac{y_{i,\omega} - \mu_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}{\sigma_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}$$

If we set $\tau = \omega$; then, we have

$$\begin{aligned}\frac{\partial u_{i,\omega}}{\partial y_{i,\omega}} &= \frac{1}{\sigma_i} + \frac{\partial u_{i,\omega}}{\partial \mu_i}\frac{\partial \mu_i}{\partial y_{i,\omega}} + \frac{\partial u_{i,\omega}}{\partial \sigma_i}\frac{\partial \sigma_i}{\partial y_{i,\omega}} \\ &= \frac{1}{\sigma_i} - \frac{1}{\sigma_i}\frac{\partial \mu_i}{\partial y_{i,\omega}} - \frac{y_{i,\omega} - \mu_i}{\sigma_i^2}\frac{\partial \sigma_i}{\partial y_{i,\omega}}\end{aligned}$$

Batch Normalization: *Backpropagation*

Now, let us determine partial derivatives

$$\mu_i = \frac{1}{\Omega} \sum_{\omega=1}^{\Omega} y_{i,\omega}$$

So, we can write

$$\frac{\partial \mu_i}{\partial y_{i,\omega}} = \frac{1}{\Omega}$$

Batch Normalization: *Backpropagation*

Now, let us determine partial derivatives

$$\mu_i = \frac{1}{\Omega}\sum_{\omega=1}^{\Omega} y_{i,\omega}$$

So, we can write

$$\frac{\partial \mu_i}{\partial y_{i,\omega}} = \frac{1}{\Omega}$$

$$\sigma_i = \sqrt{\frac{1}{\Omega}\sum_{\omega=1}^{\Omega}\left(y_{i,\omega} - \mu_i\right)^2}$$

Here, we should do it in two steps

$$\frac{\partial \sigma_i}{\partial y_{i,\omega}} = \frac{1}{2\sigma_i}\left(\frac{2}{\Omega}\left(y_{i,\omega} - \mu_i\right)\right) + \frac{\partial \sigma_i}{\partial \mu_i}\frac{\partial \mu_i}{\partial y_{i,\omega}}$$

Batch Normalization: *Backpropagation*

Now, let us determine partial derivatives

$$\mu_i = \frac{1}{\Omega}\sum_{\omega=1}^{\Omega} y_{i,\omega}$$

So, we can write

$$\frac{\partial \mu_i}{\partial y_{i,\omega}} = \frac{1}{\Omega}$$

$$\sigma_i = \sqrt{\frac{1}{\Omega}\sum_{\omega=1}^{\Omega}\left(y_{i,\omega}-\mu_i\right)^2}$$

Here, we should do it in two steps

$$\begin{aligned}\frac{\partial \sigma_i}{\partial y_{i,\omega}} &= \frac{1}{2\sigma_i}\left(\frac{2}{\Omega}\left(y_{i,\omega}-\mu_i\right)\right) + \frac{\partial \sigma_i}{\partial \mu_i}\frac{\partial \mu_i}{\partial y_{i,\omega}} \\ &= \frac{1}{\Omega\sigma_i}\left(y_{i,\omega}-\mu_i\right) + \underbrace{\frac{\partial \sigma_i}{\partial \mu_i}}_{0}\frac{\partial \mu_i}{\partial y_{i,\omega}}\end{aligned}$$

Batch Normalization: *Backpropagation*

Now, let us determine partial derivatives

$$\mu_i = \frac{1}{\Omega}\sum_{\omega=1}^{\Omega} y_{i,\omega}$$

So, we can write

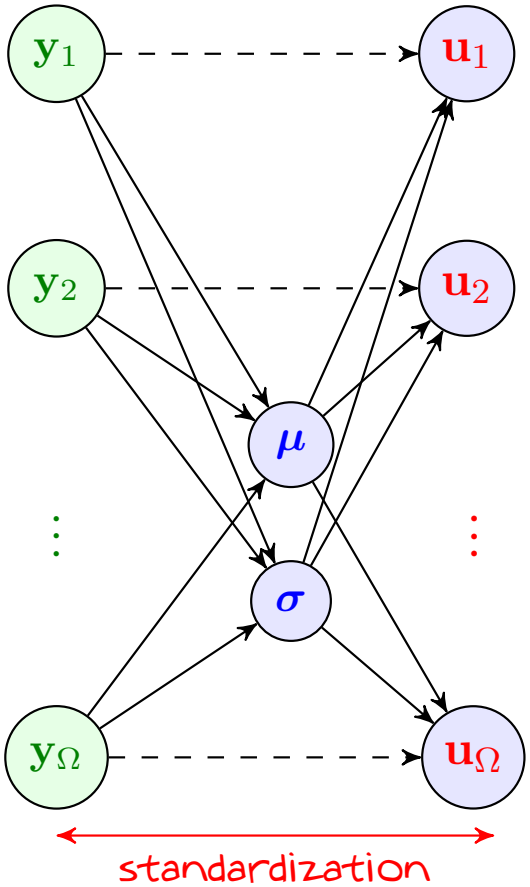
$$\frac{\partial \mu_i}{\partial y_{i,\omega}} = \frac{1}{\Omega}$$

$$\sigma_i = \sqrt{\frac{1}{\Omega}\sum_{\omega=1}^{\Omega}\left(y_{i,\omega}-\mu_i\right)^2}$$

Here, we should do it in two steps

$$\begin{aligned}\frac{\partial \sigma_i}{\partial y_{i,\omega}} &= \frac{1}{2\sigma_i}\left(\frac{2}{\Omega}\left(y_{i,\omega}-\mu_i\right)\right) + \frac{\partial \sigma_i}{\partial \mu_i}\frac{\partial \mu_i}{\partial y_{i,\omega}} \\ &= \frac{1}{\Omega\sigma_i}\left(y_{i,\omega}-\mu_i\right) + \underbrace{\frac{\partial \sigma_i}{\partial \mu_i}}_{0}\frac{\partial \mu_i}{\partial y_{i,\omega}} \\ &= \frac{1}{\Omega\sigma_i}\left(y_{i,\omega}-\mu_i\right)\end{aligned}$$

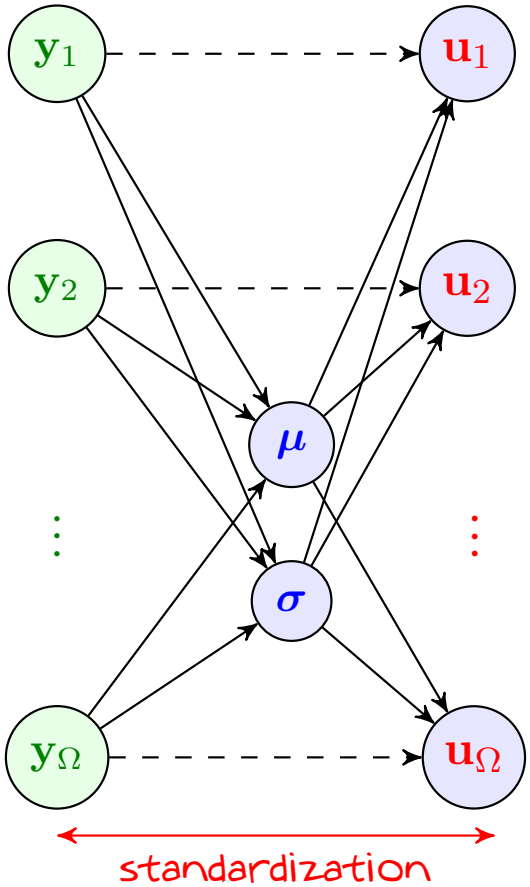
Batch Normalization: *Backpropagation*



Let's replace for the case $\tau = \omega$

$$\begin{aligned}
\frac{\partial u_{i,\tau}}{\partial y_{i,\omega}} &= \frac{1}{\sigma_i} + \frac{\partial u_{i,\omega}}{\partial \mu_i}\frac{\partial \mu_i}{\partial y_{i,\omega}} + \frac{\partial u_{i,\omega}}{\partial \sigma_i}\frac{\partial \sigma_i}{\partial y_{i,\omega}} \\
&= \frac{1}{\sigma_i} - \frac{1}{\sigma_i}\frac{\partial \mu_i}{\partial y_{i,\omega}} - \frac{y_{i,\omega} - \mu_i}{\sigma_i^2}\frac{\partial \sigma_i}{\partial y_{i,\omega}} \\
&= \frac{1}{\sigma_i} - \frac{1}{\Omega\sigma_i} - \frac{(y_{i,\omega} - \mu_i)^2}{\Omega\sigma_i^3}
\end{aligned}$$

Batch Normalization: *Backpropagation*



$$u_{i,\tau} = \frac{y_{i,\omega} - \mu_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}{\sigma_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}$$

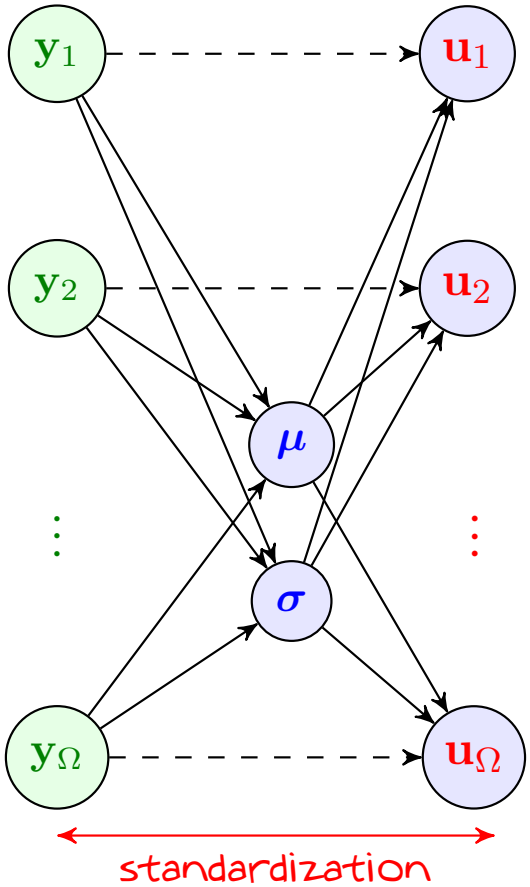
If we set $\tau \neq \omega$; then, we have

$$\frac{\partial u_{i,\tau}}{\partial y_{i,\omega}} = 0 + \frac{\partial u_{i,\tau}}{\partial \mu_i}\frac{\partial \mu_i}{\partial y_{i,\omega}} + \frac{\partial u_{i,\tau}}{\partial \sigma_i}\frac{\partial \sigma_i}{\partial y_{i,\omega}}$$

Everything as before

↳ *Just the first term drops*

Batch Normalization: *Backpropagation*



$$u_{i,\tau} = \frac{y_{i,\omega} - \mu_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}{\sigma_i\left(y_{i,1}, \ldots, y_{i,\Omega}\right)}$$

If we set $\tau \neq \omega$; then, we have

$$\begin{aligned}\frac{\partial u_{i,\tau}}{\partial y_{i,\omega}} &= 0 + \frac{\partial u_{i,\tau}}{\partial \mu_i}\frac{\partial \mu_i}{\partial y_{i,\omega}} + \frac{\partial u_{i,\tau}}{\partial \sigma_i}\frac{\partial \sigma_i}{\partial y_{i,\omega}} \\ &= -\frac{1}{\sigma_i}\frac{\partial \mu_i}{\partial y_{i,\omega}} - \frac{y_{i,\omega} - \mu_i}{\sigma_i^2}\frac{\partial \sigma_i}{\partial y_{i,\omega}} \\ &= -\frac{1}{\Omega\sigma_i} - \frac{\left(y_{i,\omega} - \mu_i\right)^2}{\Omega\sigma_i^3}\end{aligned}$$

Everything as before

↳ *Just the first term drops*

Batch Normalization: *Backpropagation*

The last piece of derivation is to relate these partial derivatives to gradient of empirical risk determined over the whole mini-batch

$$\nabla_{\mathbf{y}_\omega} \hat{R} = \nabla_{\mathbf{y}_\omega} \frac{1}{\Omega} \sum_{\tau=1}^{\Omega} \hat{R}_\tau = \frac{1}{\Omega} \sum_{\tau=1}^{\Omega} \nabla_{\mathbf{y}_\omega} \hat{R}_\tau$$

Batch Normalization: *Backpropagation*

The last piece of derivation is to relate these partial derivatives to gradient of empirical risk determined over the whole mini-batch

$$\nabla_{\mathbf{y}_\omega} \hat{R} = \nabla_{\mathbf{y}_\omega} \frac{1}{\Omega} \sum_{\tau=1}^{\Omega} \hat{R}_\tau = \frac{1}{\Omega} \sum_{\tau=1}^{\Omega} \nabla_{\mathbf{y}_\omega} \hat{R}_\tau$$

From above derivation we have

$$\nabla_{\mathbf{y}_\omega} \hat{R}_\tau = \frac{\mathbb{1}\{\tau = \omega\}}{\boldsymbol{\sigma}} - \frac{1}{\Omega\boldsymbol{\sigma}} - \frac{(\mathbf{y}_\omega - \boldsymbol{\mu})^2}{\Omega\boldsymbol{\sigma}^3}$$

with all operations being entry-wise!

Batch Normalization: *Backpropagation*

The last piece of derivation is to relate these partial derivatives to gradient of empirical risk determined over the whole mini-batch

$$\nabla_{\mathbf{y}_\omega} \hat{R} = \nabla_{\mathbf{y}_\omega} \frac{1}{\Omega} \sum_{\tau=1}^{\Omega} \hat{R}_\tau = \frac{1}{\Omega} \sum_{\tau=1}^{\Omega} \nabla_{\mathbf{y}_\omega} \hat{R}_\tau$$

From above derivation we have

$$\nabla_{\mathbf{y}_\omega} \hat{R}_\tau = \frac{\mathbb{1}\{\tau = \omega\}}{\boldsymbol{\sigma}} - \frac{1}{\Omega\boldsymbol{\sigma}} - \frac{(\mathbf{y}_\omega - \boldsymbol{\mu})^2}{\Omega\boldsymbol{\sigma}^3}$$

with all operations being entry-wise! We then derive $\nabla_{\mathbf{W}_\ell} \hat{R}$ exactly as in sample-wise backpropagation

Batch Normalization: *Backpropagation*

The last piece of derivation is to relate these partial derivatives to gradient of empirical risk determined over the whole mini-batch

$$\nabla_{\mathbf{y}_\omega} \hat{R} = \nabla_{\mathbf{y}_\omega} \frac{1}{\Omega} \sum_{\tau=1}^{\Omega} \hat{R}_\tau = \frac{1}{\Omega} \sum_{\tau=1}^{\Omega} \nabla_{\mathbf{y}_\omega} \hat{R}_\tau$$

From above derivation we have

$$\nabla_{\mathbf{y}_\omega} \hat{R}_\tau = \frac{\mathbb{1}\{\tau = \omega\}}{\boldsymbol{\sigma}} - \frac{1}{\Omega\boldsymbol{\sigma}} - \frac{(\mathbf{y}_\omega - \boldsymbol{\mu})^2}{\Omega\boldsymbol{\sigma}^3}$$

with all operations being entry-wise! We then derive $\nabla_{\mathbf{W}_\ell} \hat{R}$ exactly as in sample-wise backpropagation

Suggestion

Try to write the complete backpropagation with batch normalization

Batch Normalization: *Testing*

+ *Say we trained our NN! Now how we test it for single new point? We do not have any mini-batch anymore!*
- Good point! In practice, we use *moving average*

Throughout training, we compute moving averages $\bar{\boldsymbol{\mu}}_\ell$ and $\bar{\boldsymbol{\sigma}}_\ell$

At each layer ℓ, we start with some initial $\bar{\boldsymbol{\mu}}_\ell$ and $\bar{\boldsymbol{\sigma}}_\ell$ and compute

$$\bar{\boldsymbol{\mu}}_\ell = \alpha \bar{\boldsymbol{\mu}}_\ell + (1-\alpha)\,\boldsymbol{\mu}_\ell$$
$$\bar{\boldsymbol{\sigma}}_\ell = \alpha \bar{\boldsymbol{\sigma}}_\ell + (1-\alpha)\,\boldsymbol{\sigma}_\ell$$

after each iteration for some $0 < \alpha < 1$: typically close to 1

We use these values for normalization after we are over with training

Batch Normalization: *Final Points*

Few points that you may observe *in implementation of batch normalization*

Batch Normalization: *Final Points*

Few points that you may observe *in implementation of batch normalization*

- *We usually perturb variance with a small constant for numerical stability*

$$\boldsymbol{\sigma} = \sqrt{\frac{1}{\Omega}\sum_{\omega=1}^{\Omega}\left(\mathbf{y}_{\omega} - \boldsymbol{\mu}\right)^2 + \boldsymbol{\epsilon}}$$

Batch Normalization: *Final Points*

Few points that you may observe *in implementation of batch normalization*

- *We usually perturb variance with a small constant for numerical stability*

$$\boldsymbol{\sigma} = \sqrt{\frac{1}{\Omega}\sum_{\omega=1}^{\Omega}(\mathbf{y}_\omega - \boldsymbol{\mu})^2 + \boldsymbol{\epsilon}}$$

- *We could normalize before or after activation*
  - ↳ *Here, we did it after activation*
  - ↳ *In the original proposal is was before activation*
  - ↳ *In general, it is not fully known which one is better*
    - ⇝ *We may try both and see which one gives better result*