About Training

Outline

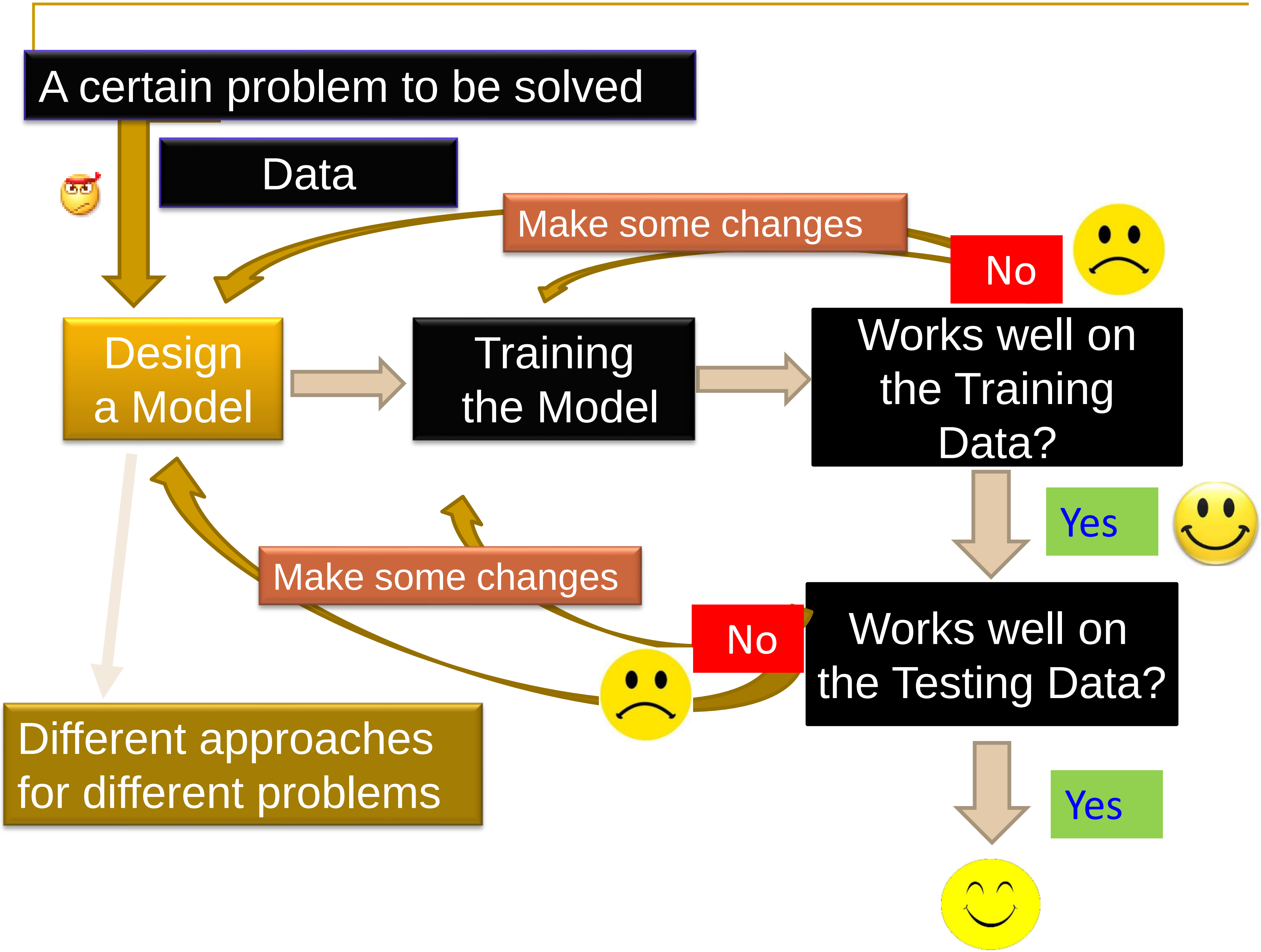
- Hyperparameters & Parameters
- Setting up the data
- Gradient Descent
- Learning rate
- Batch Normalization
- Early stopping
- Regularization
- Dropout
- Hyperparameter tuning

Supervised Learning Task - Review

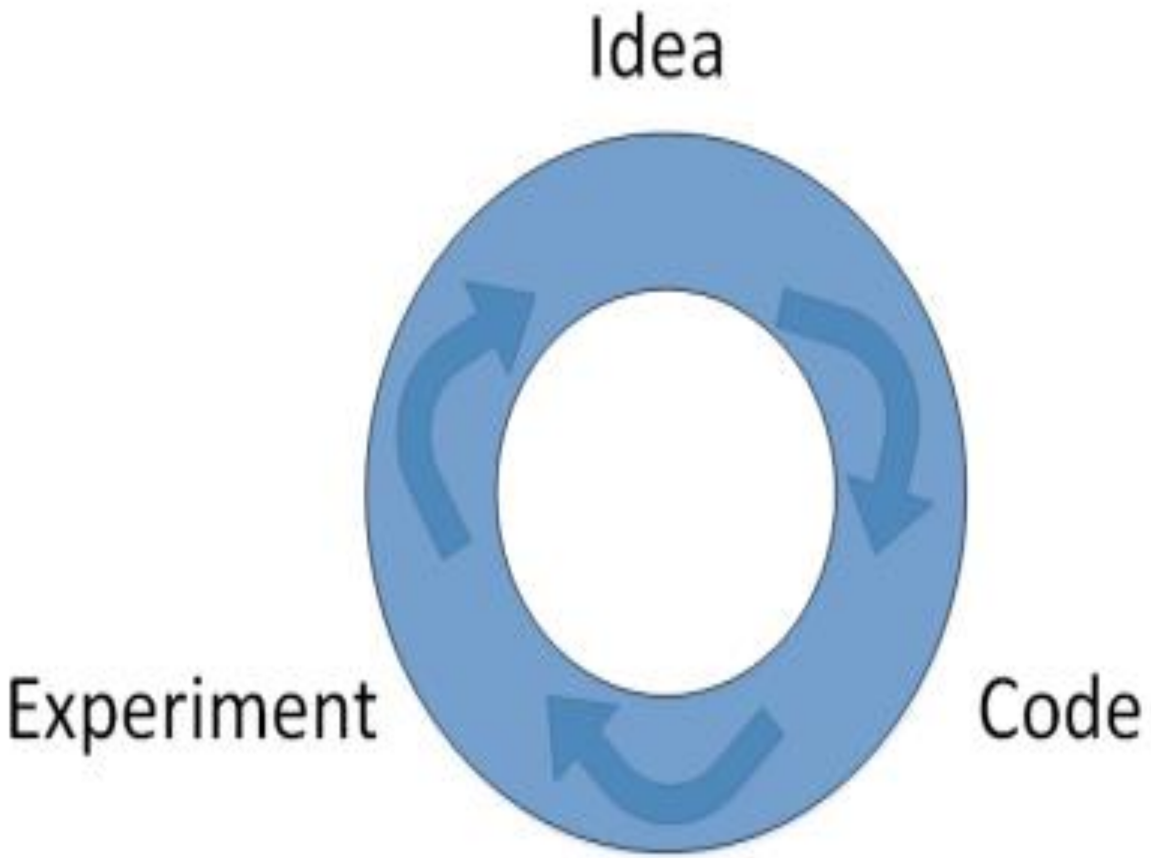
- Given: training data $\{(x_i, y_i), i = 1, \ldots, n\}$ i.i.d. from distribution D
- Find $y = f(x) \in \mathcal{H}$
- S.t. f works well on *test* data i.i.d. from distribution D
  - Find y=$f(x)$∈$\mathcal{H}$ that minimizes

$$\hat{L}(f) = \frac{1}{n}\sum_{i=1}^{n} l(f, x_i, y_i)$$

Empirical loss


A certain problem to be solved
Data
Make some changes
No
Design a Model
Training the Model
Works well on the Training Data?
Yes
Make some changes
No
Works well on the Testing Data?
Yes
Different approaches for different problems

The whole process is a cycle



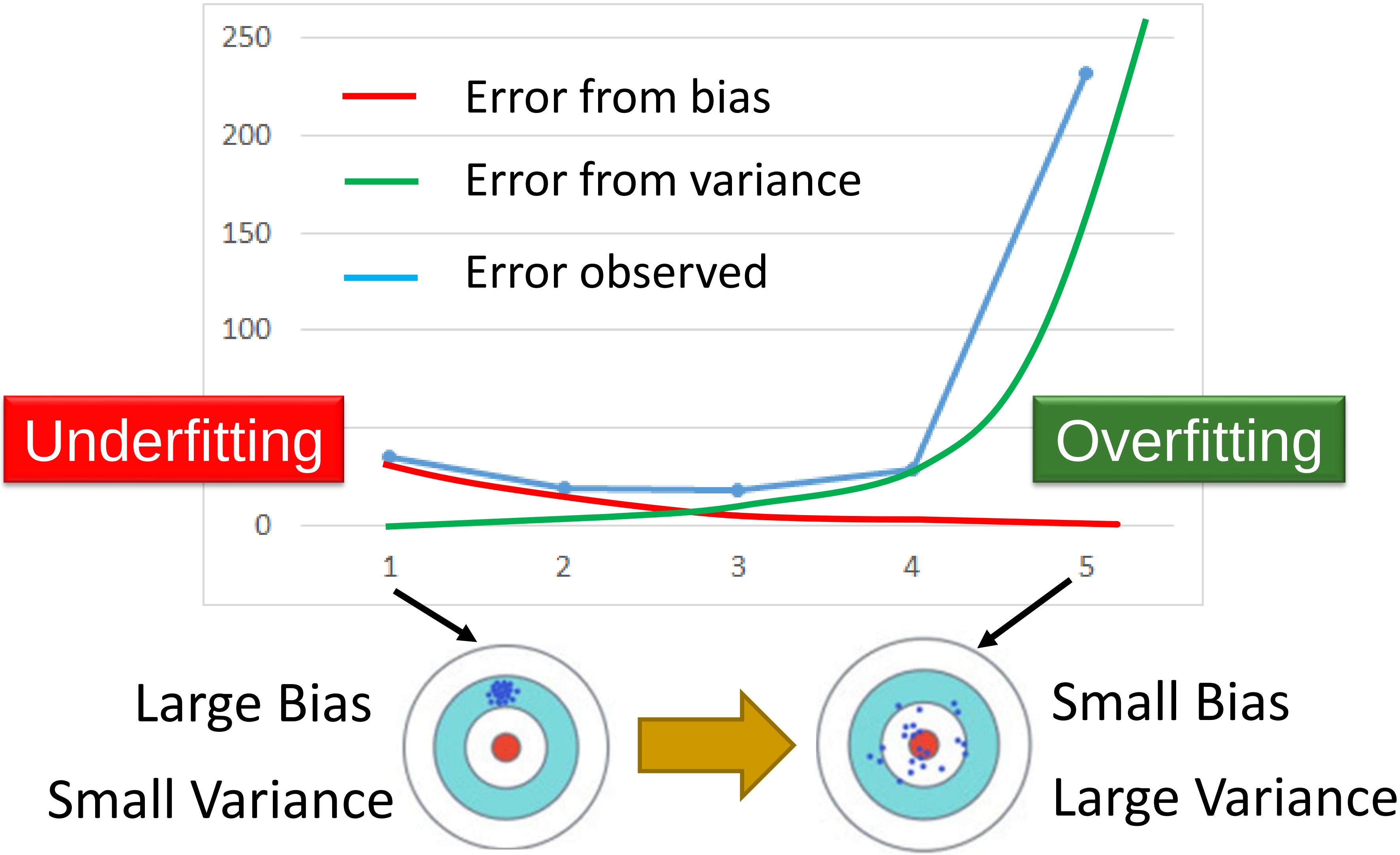
Underfitting & Overfitting

- Diagnosis：
  - If your model cannot even fit the training examples, then you have **large bias** Not train well

    Underfitting
  - If you can fit the training data, but with large error on the testing data, then you probably have large variance

    Overfitting
- For large bias:
  - Add more features as inputs
  - Use a more complex model
- For large variance:
  - Use more data: Very effective, but not always practical
  - Regularization: May increase bias

Bias v.s. Variance



Hyperparameters & Parameters

(Model Design + Hyperparameters) → Model Parameters

The building blocks:

- \# Layers
- Activations
- Optimizers

...

The knobs that you can turn:

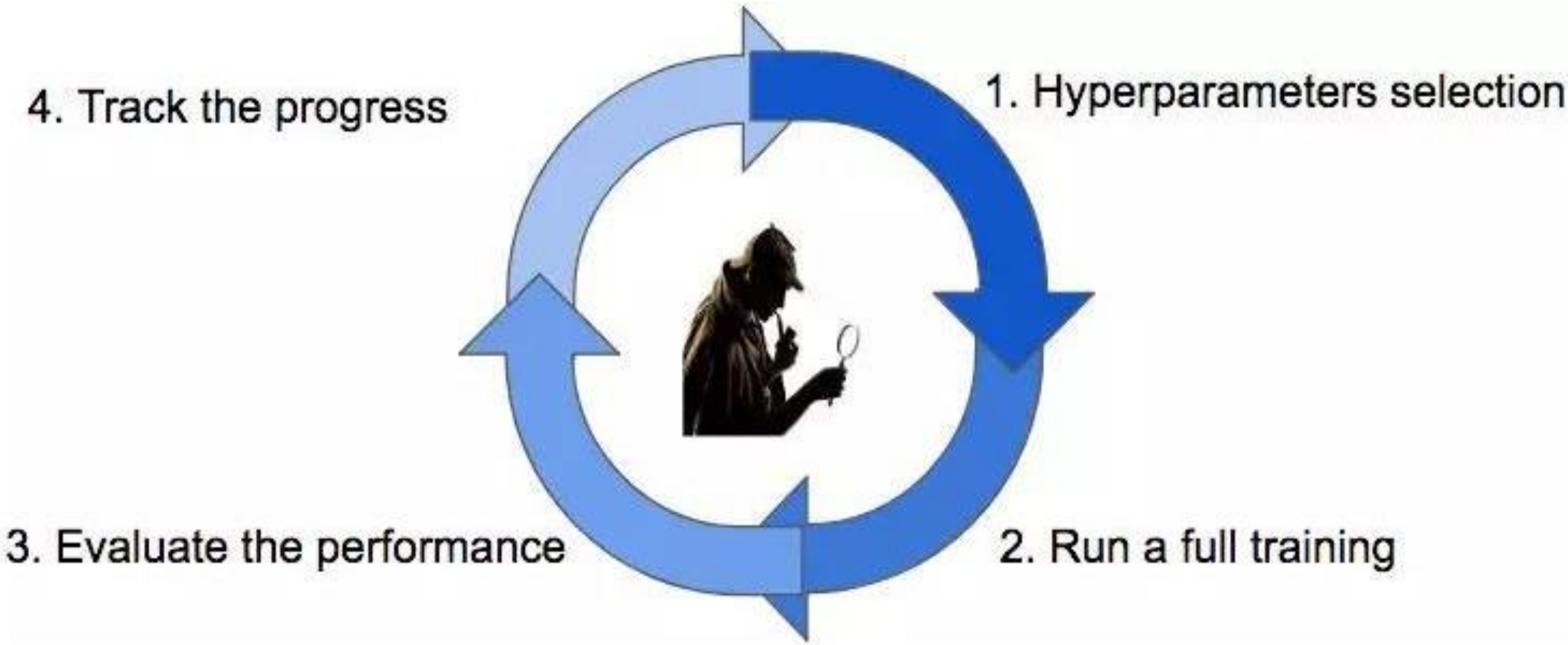
- Learning Rate
- Dropout

...

The variables learned from the data:

- weights

...

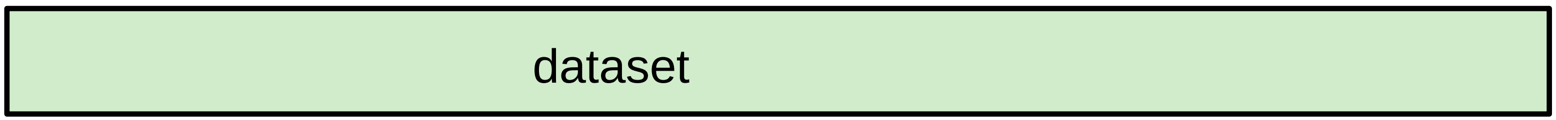

1. Hyperparameters selection
2. Run a full training
3. Evaluate the performance
4. Track the progress

Outline

- Hyperparameters & Parameters
- Setting up the data
- Gradient Descent
- Learning rate
- Batch Normalization
- Early stopping
- Regularization
- Dropout
- Hyperparameter tuning

Setting the dataset

- **Idea #1:** Choose hyperparameters that work best on the data



- BAD: Always works perfectly on the training data

- **Idea #2:** Split data into train and test, choose hyperparameters that work best on the test data

- BAD: The test set is a proxy for the generalization performance! No idea how algorithm will perform on new data

Setting the dataset

- Idea #3: Split data into train, validation, and test; choose hyperparameters on the validation data and evaluate on the test data.

- Better!

Setting the dataset

- Idea #4: Cross-Validation: Split data into folds, try each fold as validation and average the results

fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test

  - Especially useful for small datasets

Hyperparameters Setting

- Choose hyperparameters using the validation set
- Only run on the test set once at the very end!
  - Measuring the generalization of the designed model

Outline

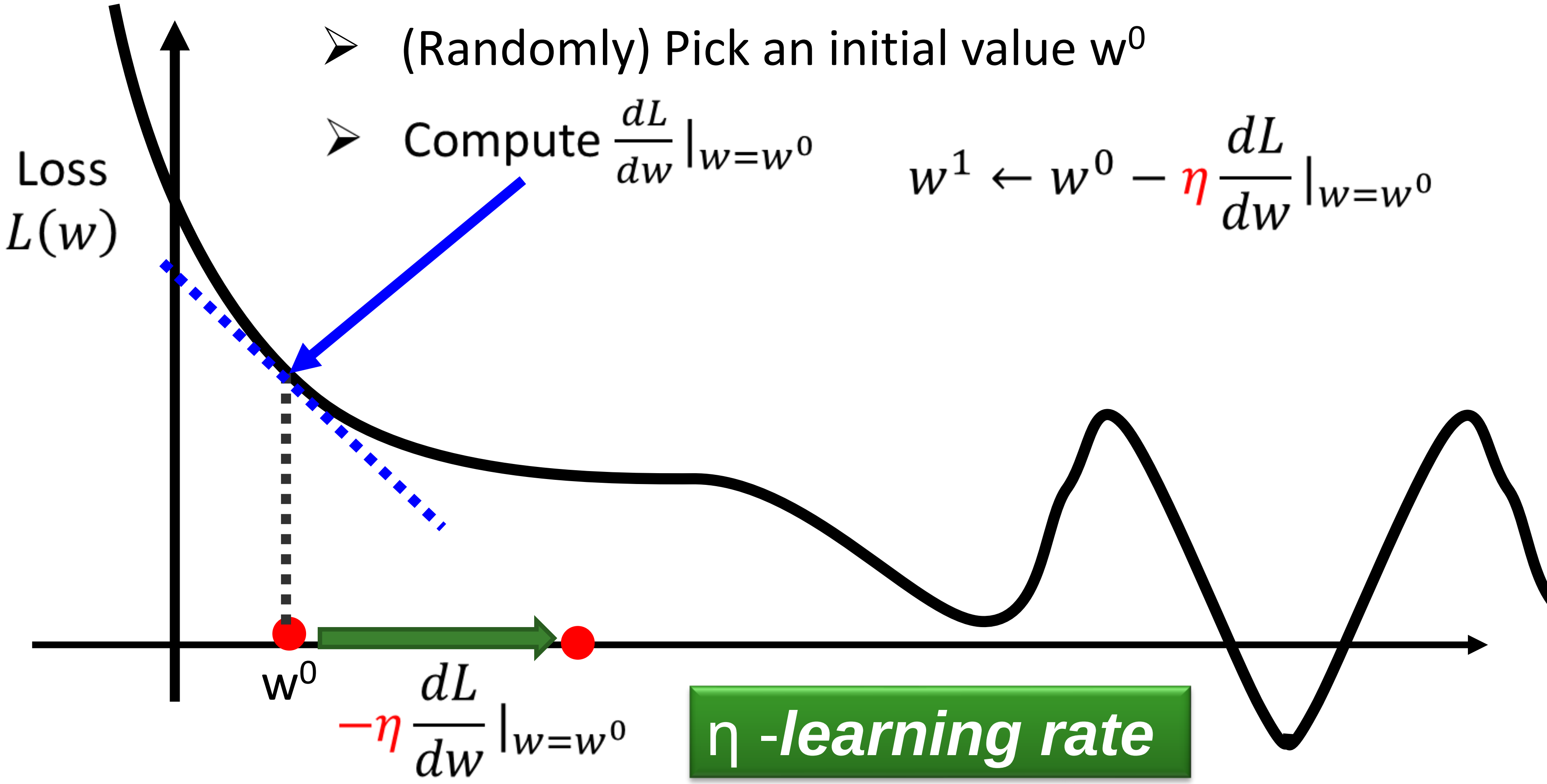
- Hyperparameters & Parameters
- Setting up the data
- Gradient descent
- Learning rate
- Batch normalization
- Early stopping
- Regularization
- Dropout
- Hyperparameter tuning

Gradient Descent

$$w^* = arg\min_w L(w)$$

- Consider loss function $L(w)$ with one parameter w:
  - (Randomly) Pick an initial value w^0
  - Compute $\frac{dL}{dw}|_{w=w^0}$

$$w^1 \leftarrow w^0 - \eta \frac{dL}{dw}|_{w=w^0}$$



η -learning rate

Gradient Descent

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix}$$ gradient

- How about two parameters?

$$w^*, b^* = arg \min_{w,b} L(w, b)$$

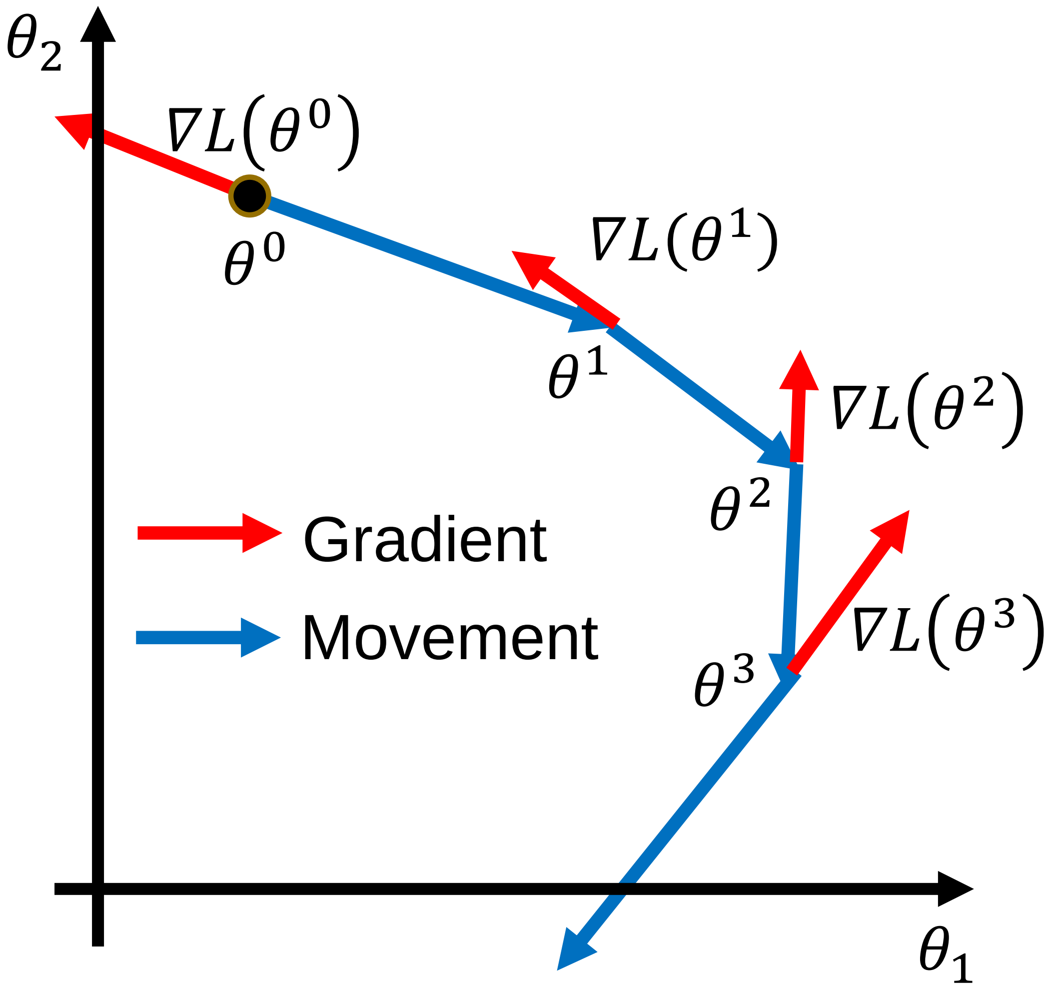
  - (Randomly) Pick an initial value w^0, b^0
  - Compute $\frac{\partial L}{\partial w}|_{w=w^0,b=b^0}$, $\frac{\partial L}{\partial b}|_{w=w^0,b=b^0}$

$$w^1 \leftarrow w^0 - \eta \frac{\partial L}{\partial w}|_{w=w^0,b=b^0} \qquad b^1 \leftarrow b^0 - \eta \frac{\partial L}{\partial b}|_{w=w^0,b=b^0}$$

  - Compute $\frac{\partial L}{\partial w}|_{w=w^1,b=b^1}$, $\frac{\partial L}{\partial b}|_{w=w^1,b=b^1}$

$$w^2 \leftarrow w^1 - \eta \frac{\partial L}{\partial w}|_{w=w^1,b=b^1} \qquad b^2 \leftarrow b^1 - \eta \frac{\partial L}{\partial b}|_{w=w^1,b=b^1}$$

Gradient Descent



Start at position θ^0

Compute gradient at θ^0

Move to $\theta^1 = \theta^0 - \eta \nabla L(\theta^0)$

Compute gradient at θ^1

Move to $\theta^2 = \theta^1 - \eta \nabla L(\theta^1)$

⋮ ⋮

Stop until $\nabla L(\theta^t) \approx 0$

Stochastic Gradient Descent

$$L = \sum_n \left(\hat{y}^n - \left(b + \sum w_i x_i^n \right) \right)^2$$

Loss is the summation over all training examples

◆ ***Gradient Descent***

$$\theta^i = \theta^{i-1} - \eta \nabla L(\theta^{i-1})$$

◆ ***Stochastic Gradient Descent***

Faster!

Pick an example x^n

Loss for only one example

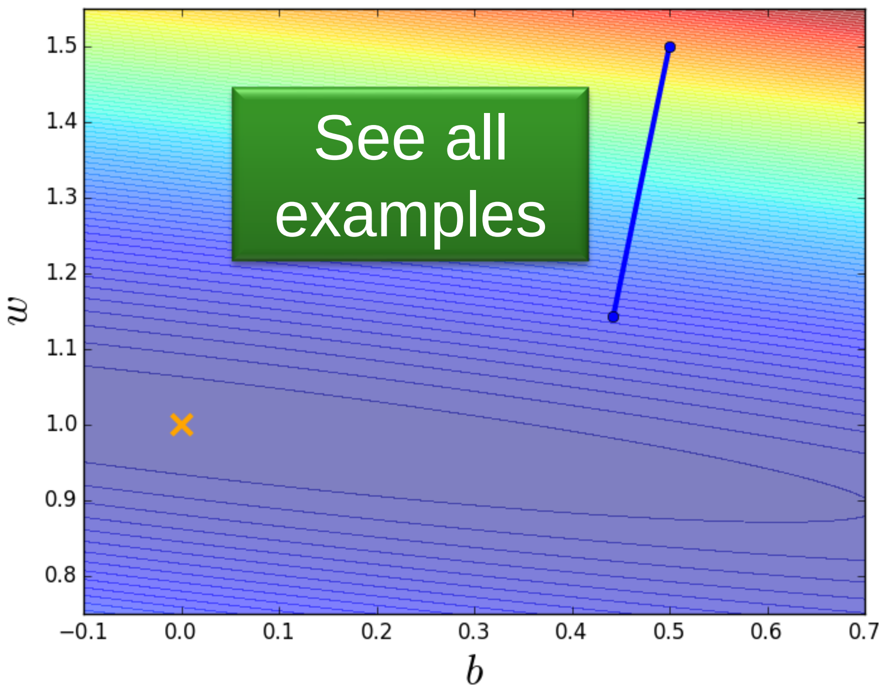
$$L^n = \left(\hat{y}^n - \left(b + \sum w_i x_i^n \right) \right)^2$$

$$\theta^i = \theta^{i-1} - \eta \nabla L^n(\theta^{i-1})$$

Stochastic Gradient Descent

Gradient Descent

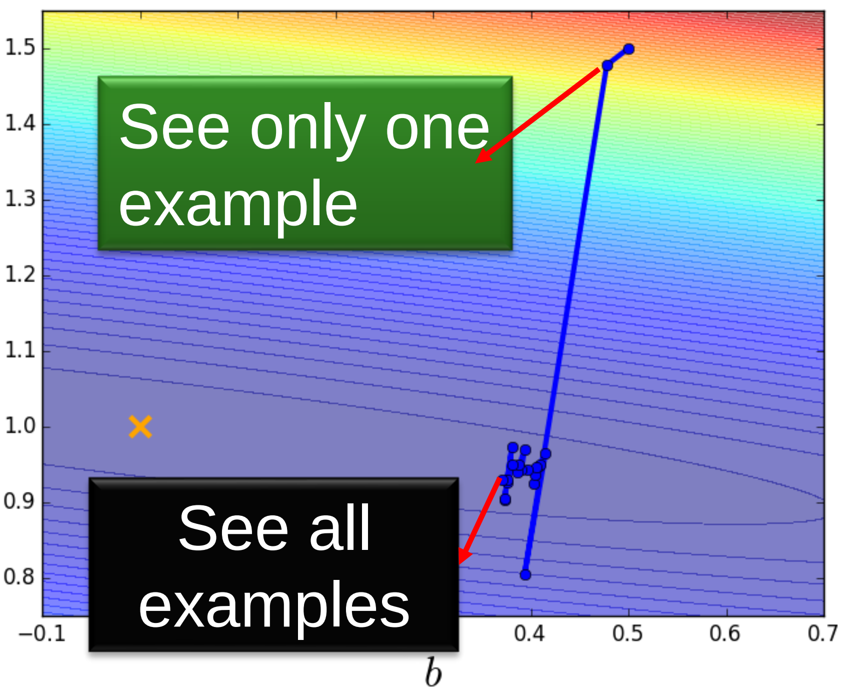
Update after seeing all examples



Stochastic Gradient Descent

Update for each example

If there are 20 examples, 20 times faster.



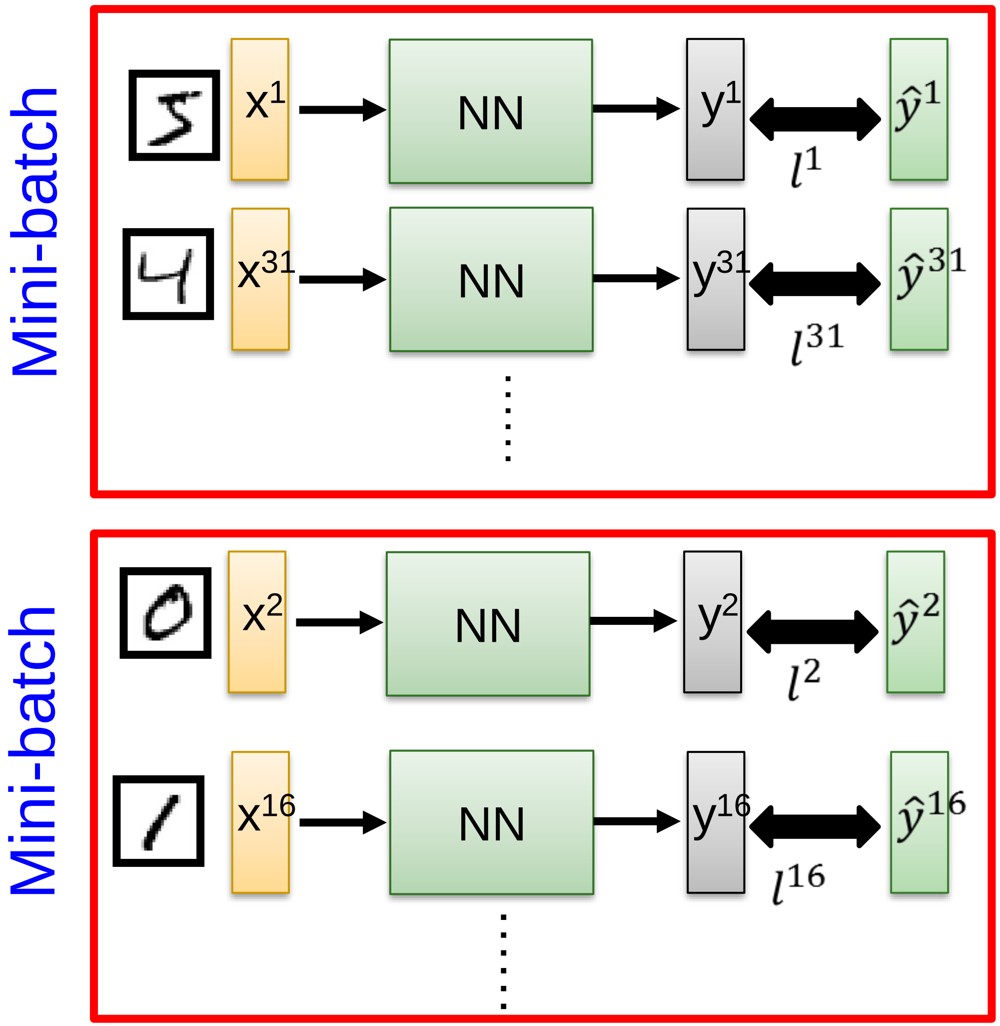
Mini-batch SGD

- Iterate over epochs
  - Iterate over dataset mini-batches $(x_1, y_1), \ldots, (x_b, y_b)$
    - Compute gradient of the mini-batch loss:
      $$\nabla \hat{L} = \frac{1}{b} \sum_{i=1}^{b} \nabla l(w, x_i, y_i)$$
    - Update parameters:
      $$w \leftarrow w - \eta \nabla \hat{L}$$
  - Check for convergence, decide whether to decay learning rate

- What are the hyperparameters?
  - Mini-batch size, learning rate decay schedule, deciding when to stop

Mini-batch

We do not really minimize total loss!



- Randomly initialize network parameters
- Pick the 1st batch

  $L' = l^1 + l^{31} + \cdots$

  Update parameters once
- Pick the 2nd batch

  $L'' = l^2 + l^{16} + \cdots$

  Update parameters once

  ⋮
- Until all mini-batches have been picked

one epoch

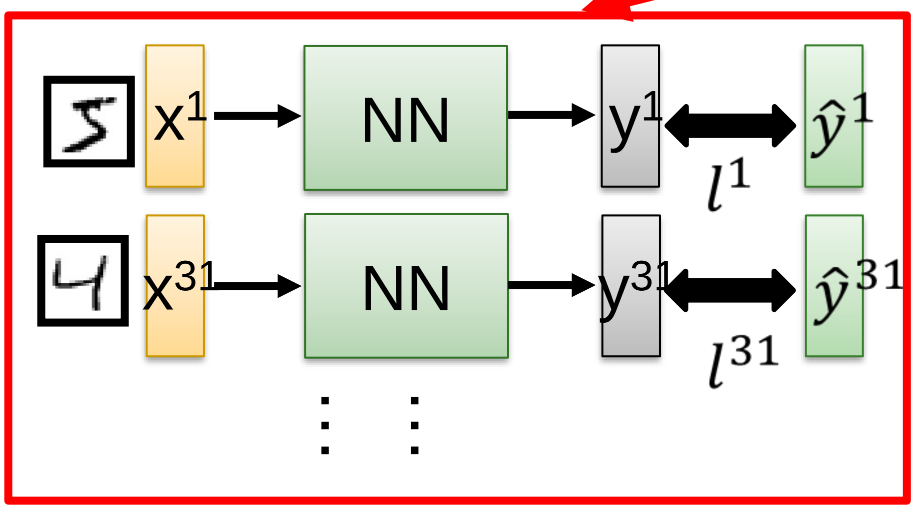
Repeat the above process

Mini-batch

Batch size influences both ***speed*** and ***performance***. You have to tune it.

```
model.fit(x_train, y_train, batch_size=100, nb_epoch=20)
```

Mini-batch



100 examples in a mini-batch

Batch size = 1 ➡ Stochastic gradient descent

- Pick the 1st batch

  $L' = l^1 + l^{31} + \cdots$

  Update parameters once

- Pick the 2nd batch

  $L'' = l^2 + l^{16} + \cdots$

  Update parameters once

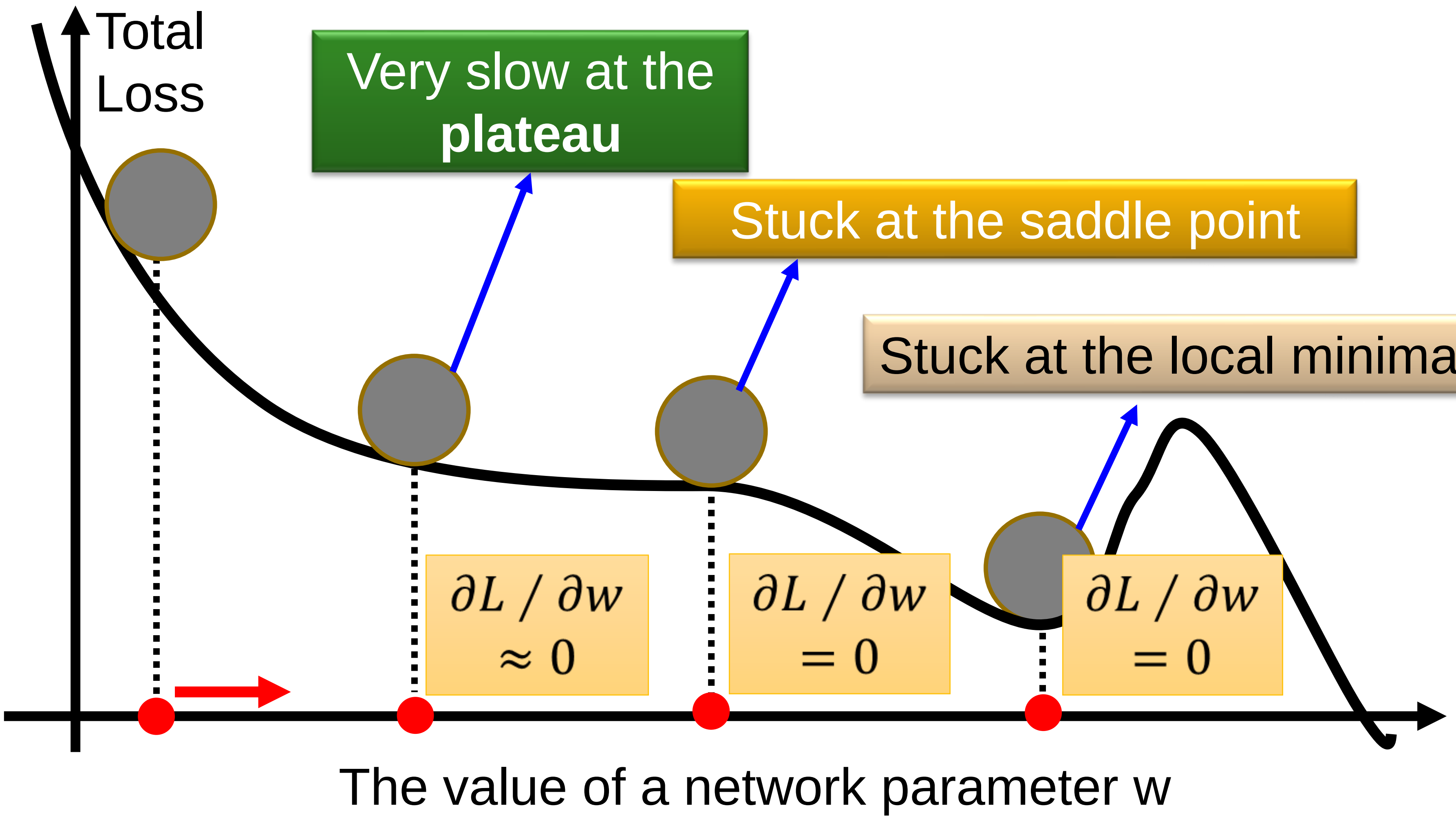
  ⋮

- Until all mini-batches have been picked

one epoch

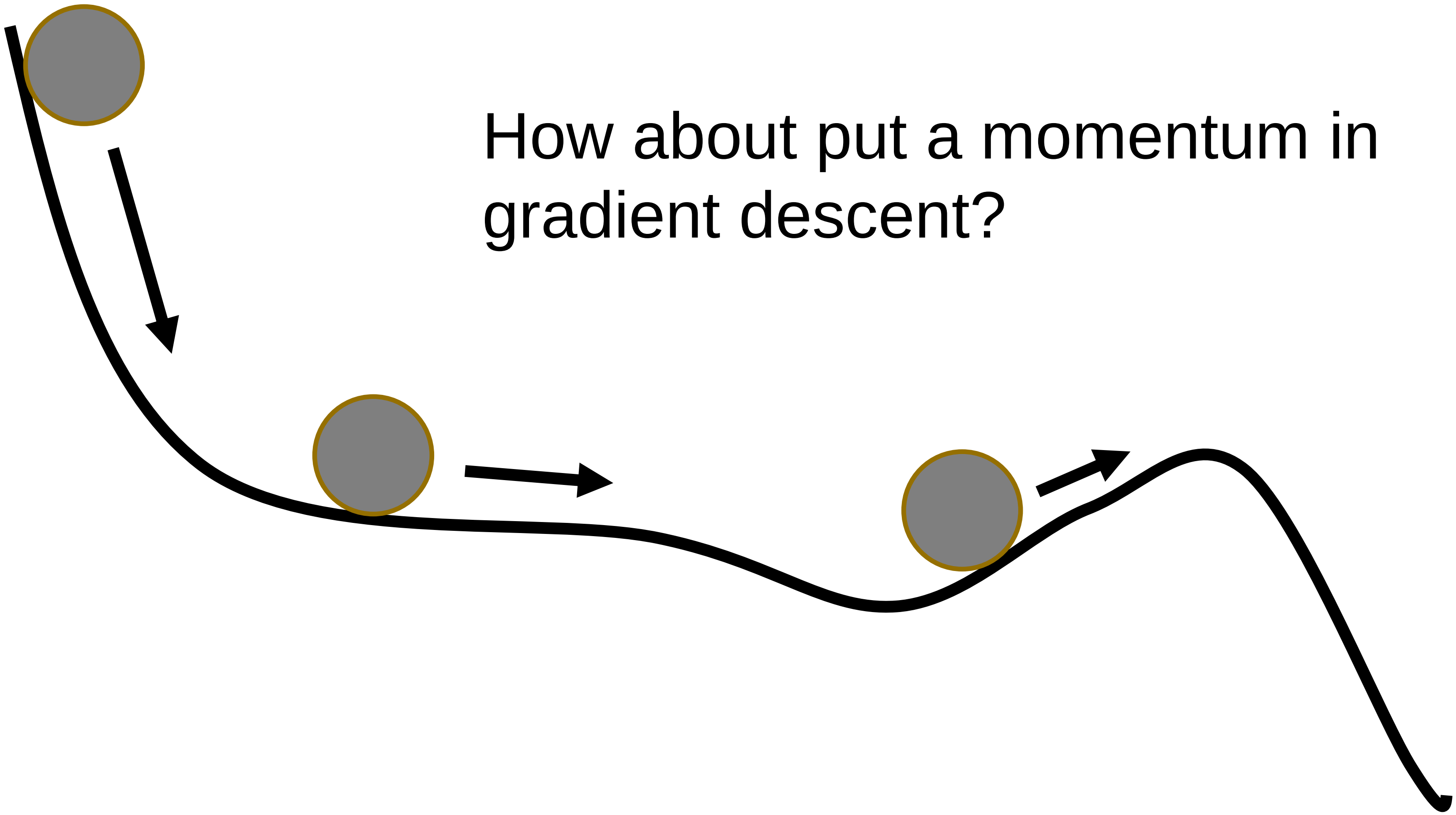
Repeat 20 times

Hard to find optimal network parameters



In physical world ……

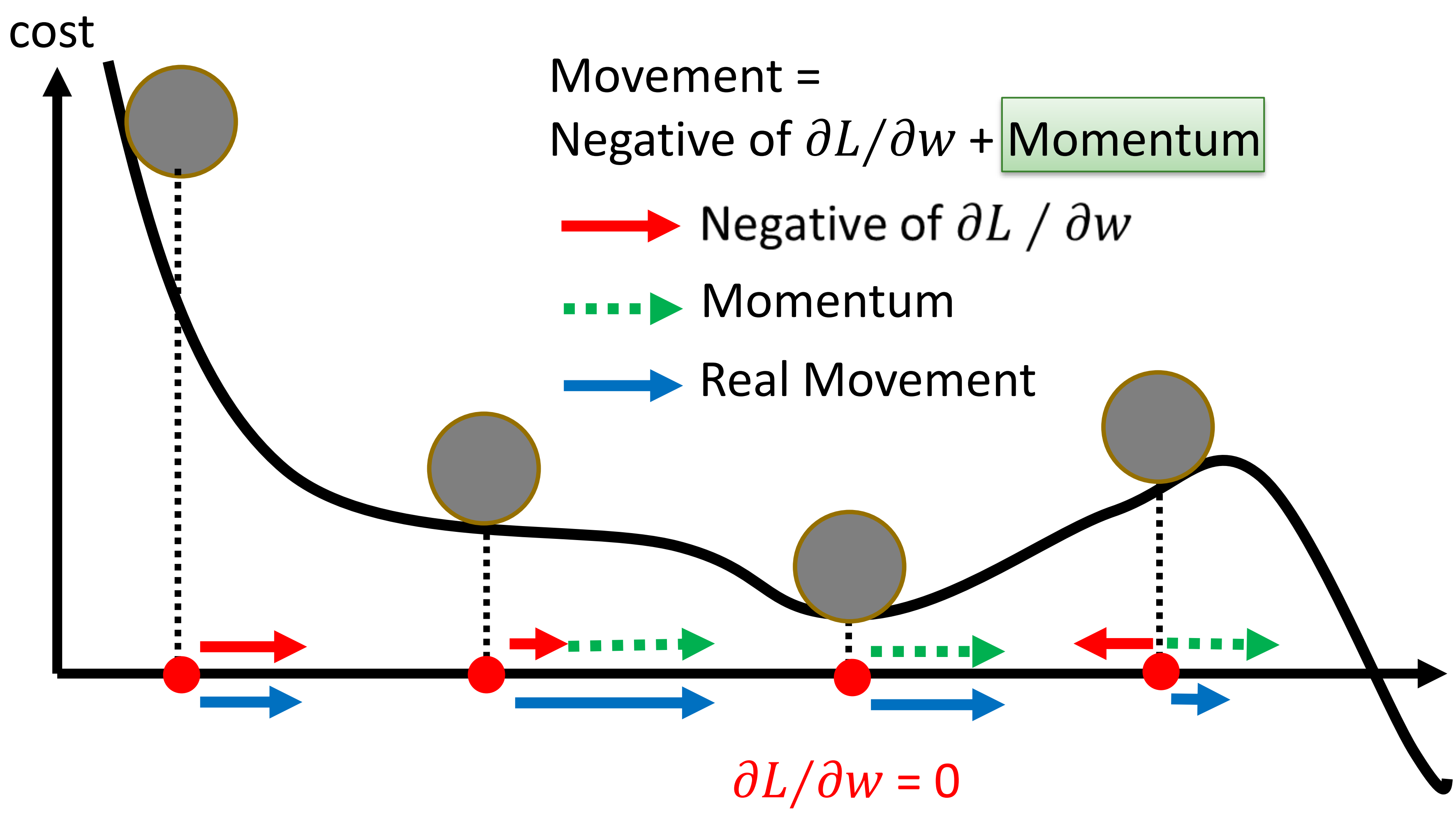
- Momentum



How about put a momentum in gradient descent?

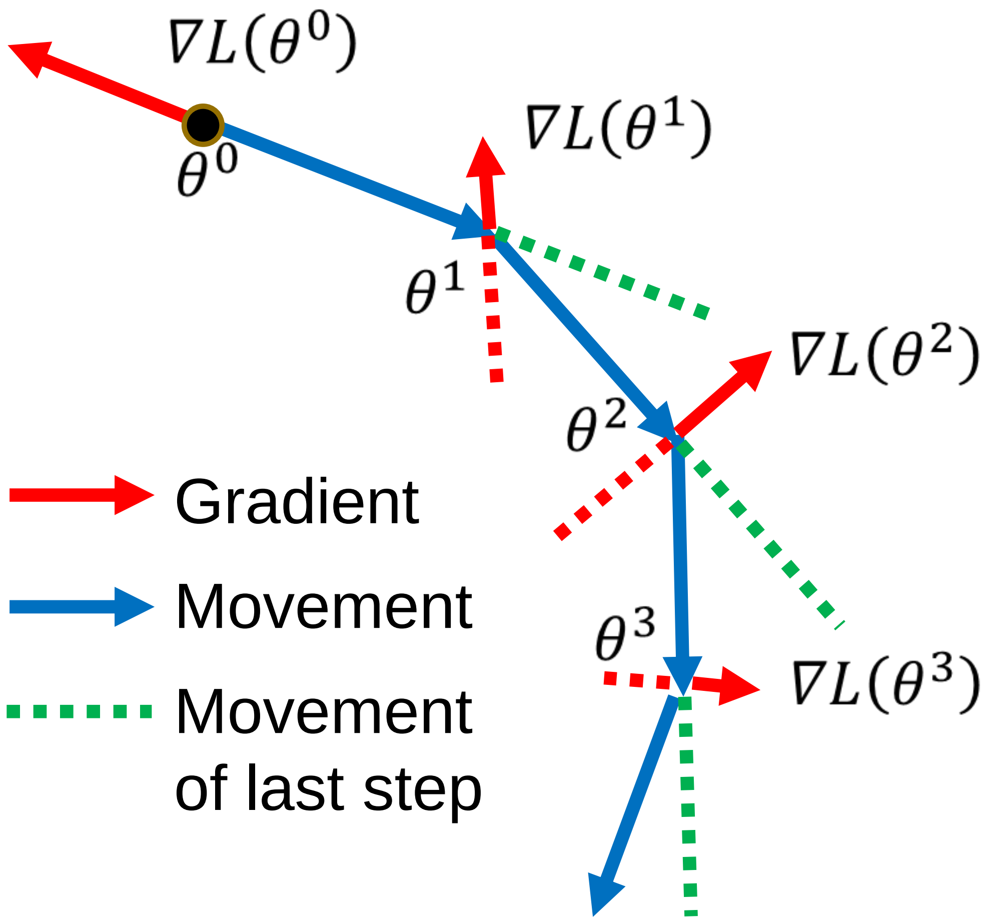
Momentum

Still not guarantee reaching global minima, but give some hope ……



Momentum

Movement: movement of the last step minus the gradient at present



Start at point θ^0

Movement $v^0=0$

Compute gradient at θ^0

Movement $v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at θ^1

Movement $v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

Movement not just based on the gradient, but also on the previous movement

Momentum

Movement: movement of last step minus the gradient at present

v^i is actually the weighted sum of all the previous gradient: $\nabla L(\theta^0)$, $\nabla L(\theta^1)$, … $\nabla L(\theta^{i-1})$

$v^0 = 0$

$v^1 = -\eta \nabla L(\theta^0)$

$v^2 = -\lambda \eta \nabla L(\theta^0) - \eta \nabla L(\theta^1)$

⋮

Start at point θ^0

Movement $v^0=0$

Compute gradient at θ^0

Movement $v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at θ^1

Movement $v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

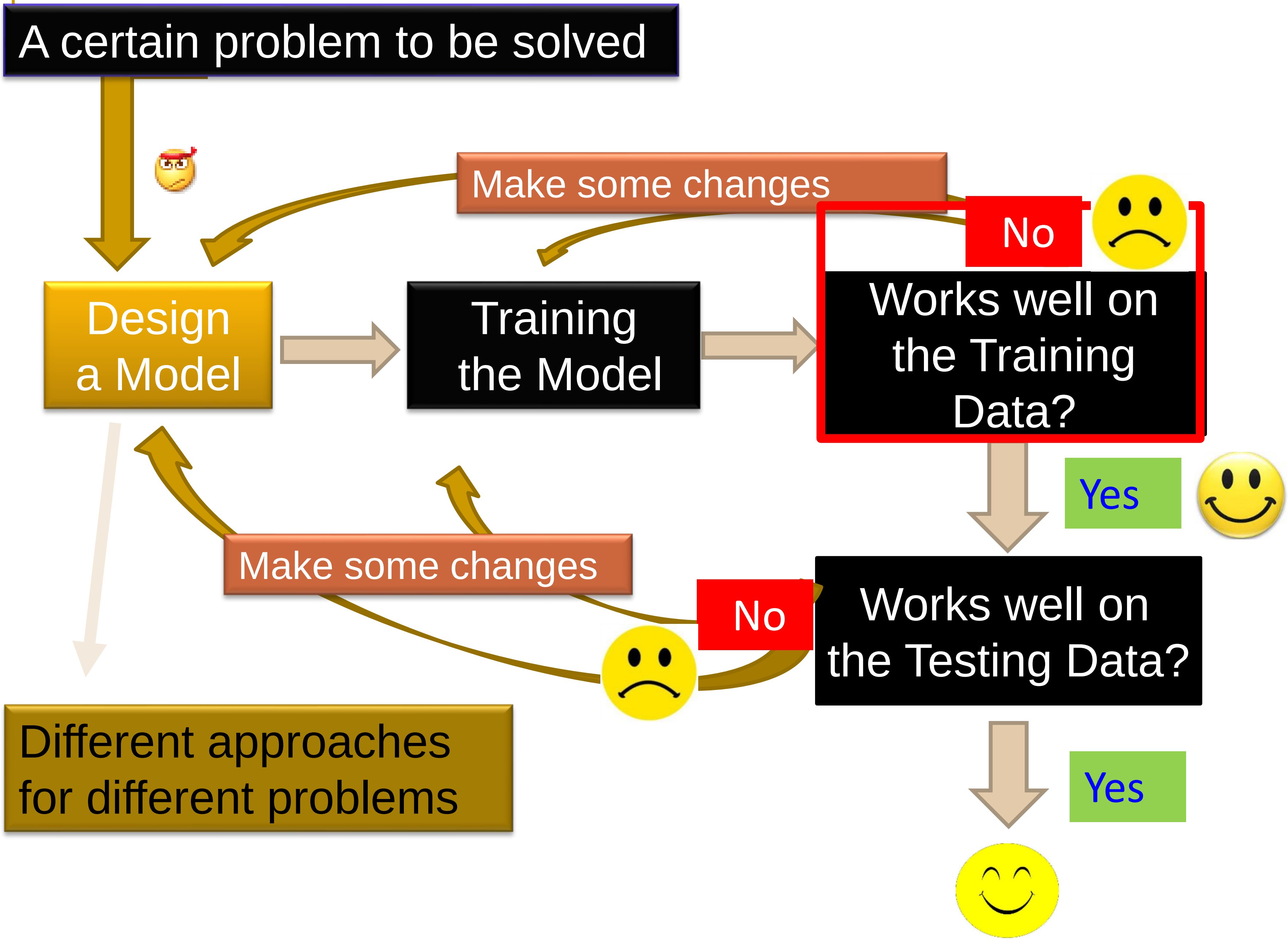
Momentum

$$v_{dw} = \beta v_{dw} + (1 - \beta)dW$$

$$v_{db} = \beta v_{db} + (1 - \beta)db$$

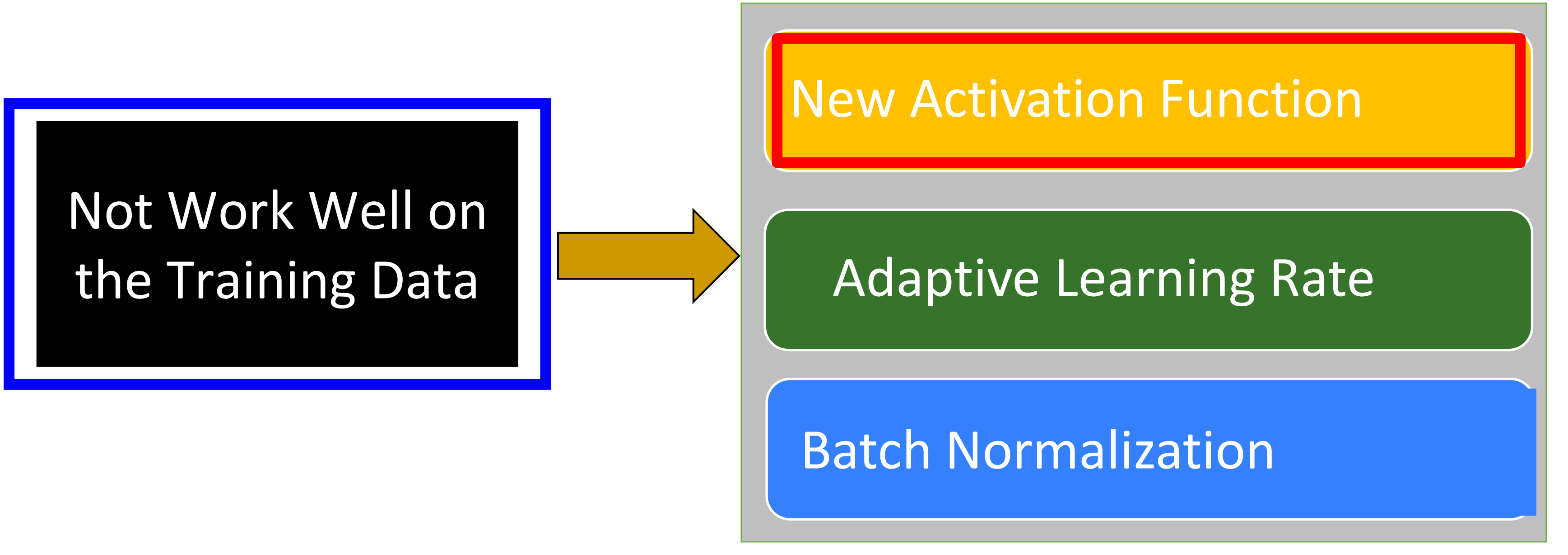
$$W = W - \alpha v_{dw}$$

$$b = b - \alpha v_{db}$$



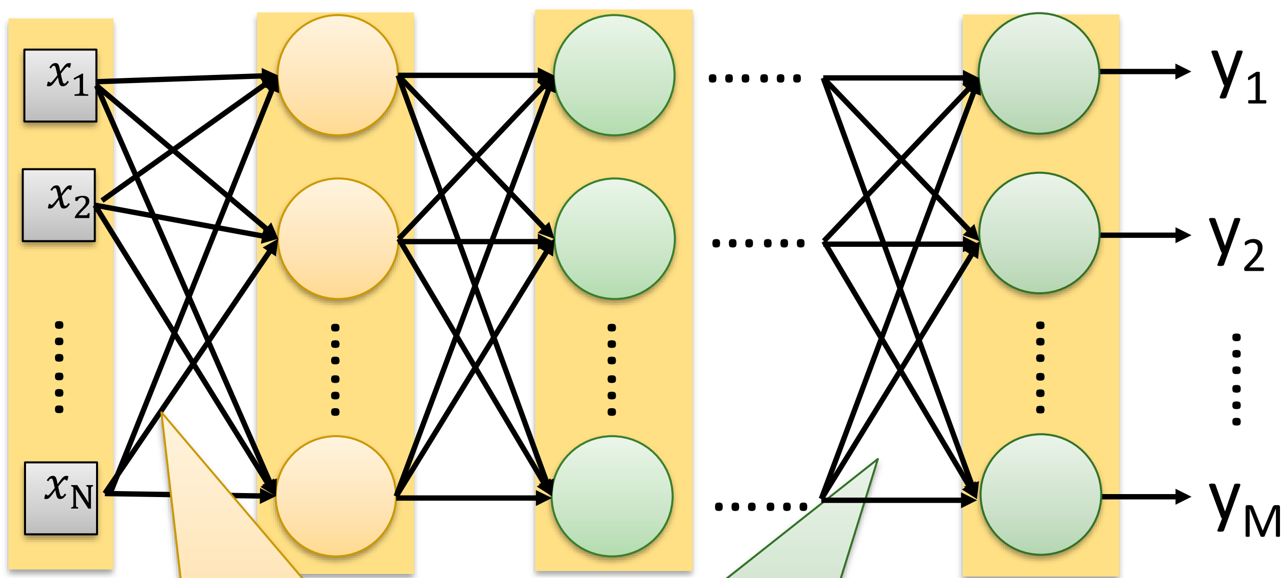
A certain problem to be solved
Make some changes
No
Design a Model
Training the Model
Works well on the Training Data?
Yes
Make some changes
No
Works well on the Testing Data?
Yes
Different approaches for different problems

Not Work Well on the Training Data



Deeper usually does not imply better

Vanishing Gradient Problem



Smaller gradients

Learn very slow

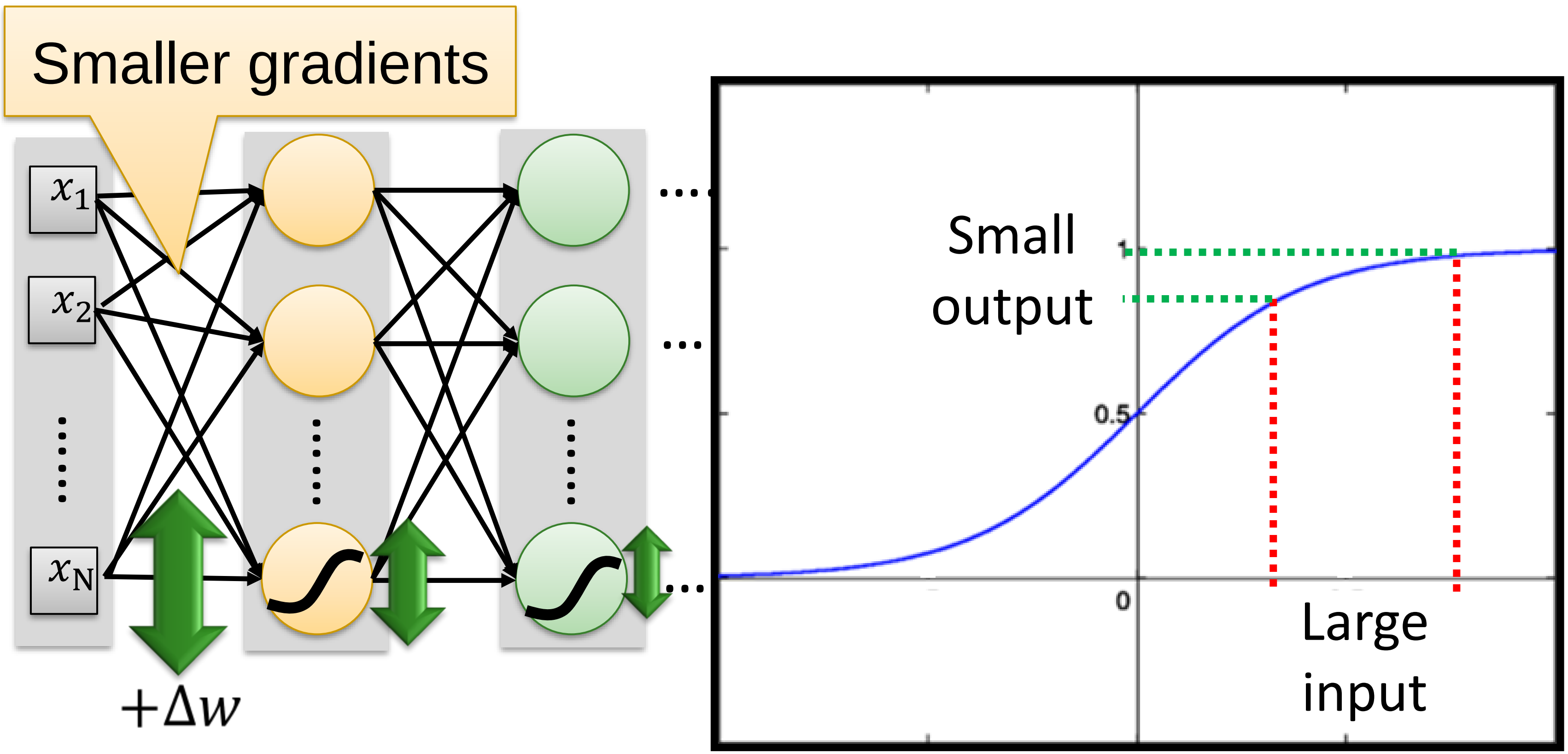
Almost random

Larger gradients

Learn very fast

Already converge

Vanishing Gradient Problem



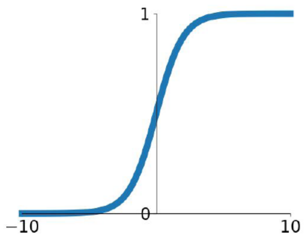
An intuitive way to compute the derivatives …

$$\frac{\partial l}{\partial w} =? \ \frac{\Delta l}{\Delta w}$$

Some Activation Functions

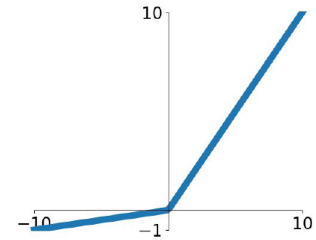
Sigmoid

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



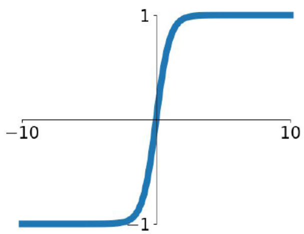
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$

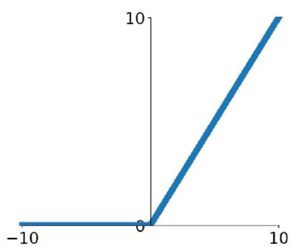


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

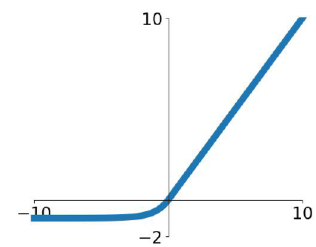
ReLU

$\max(0, x)$



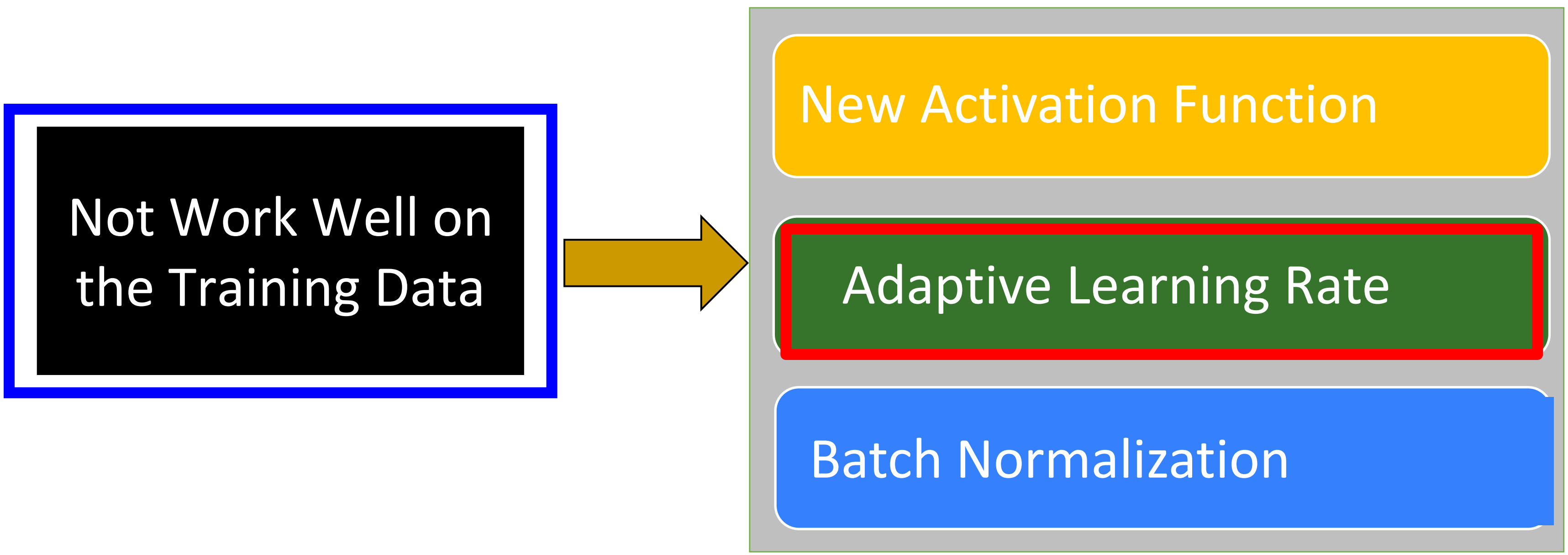
ELU

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



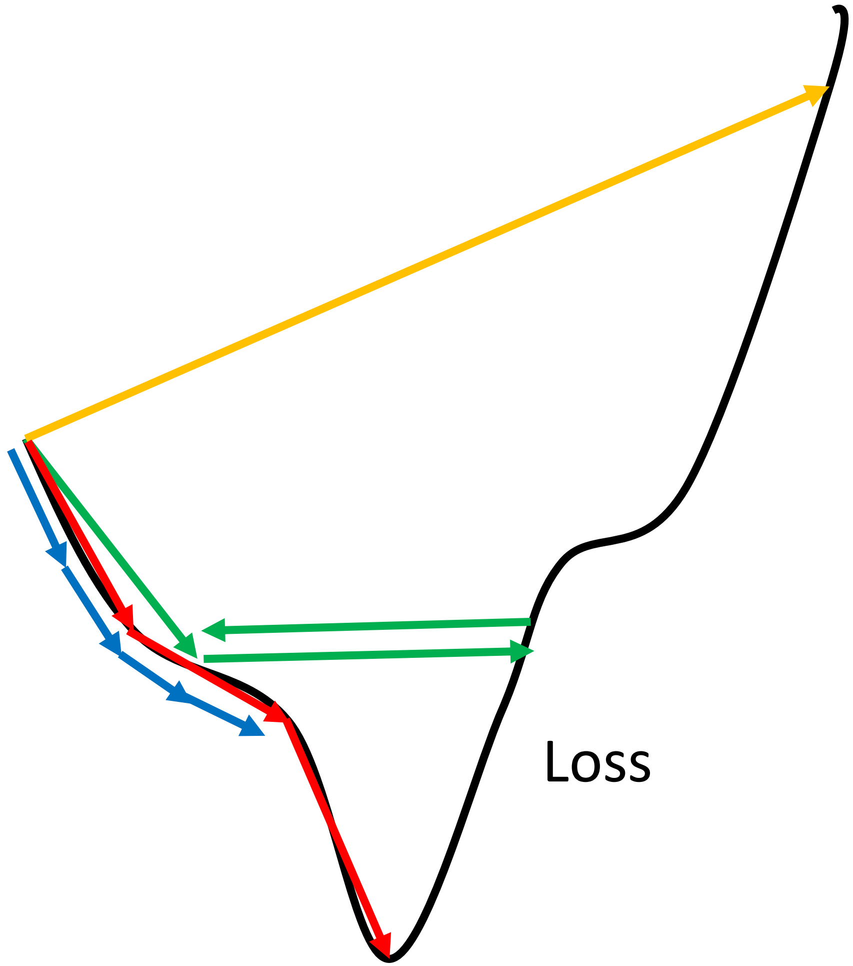
……

Not Work Well on the Training Data



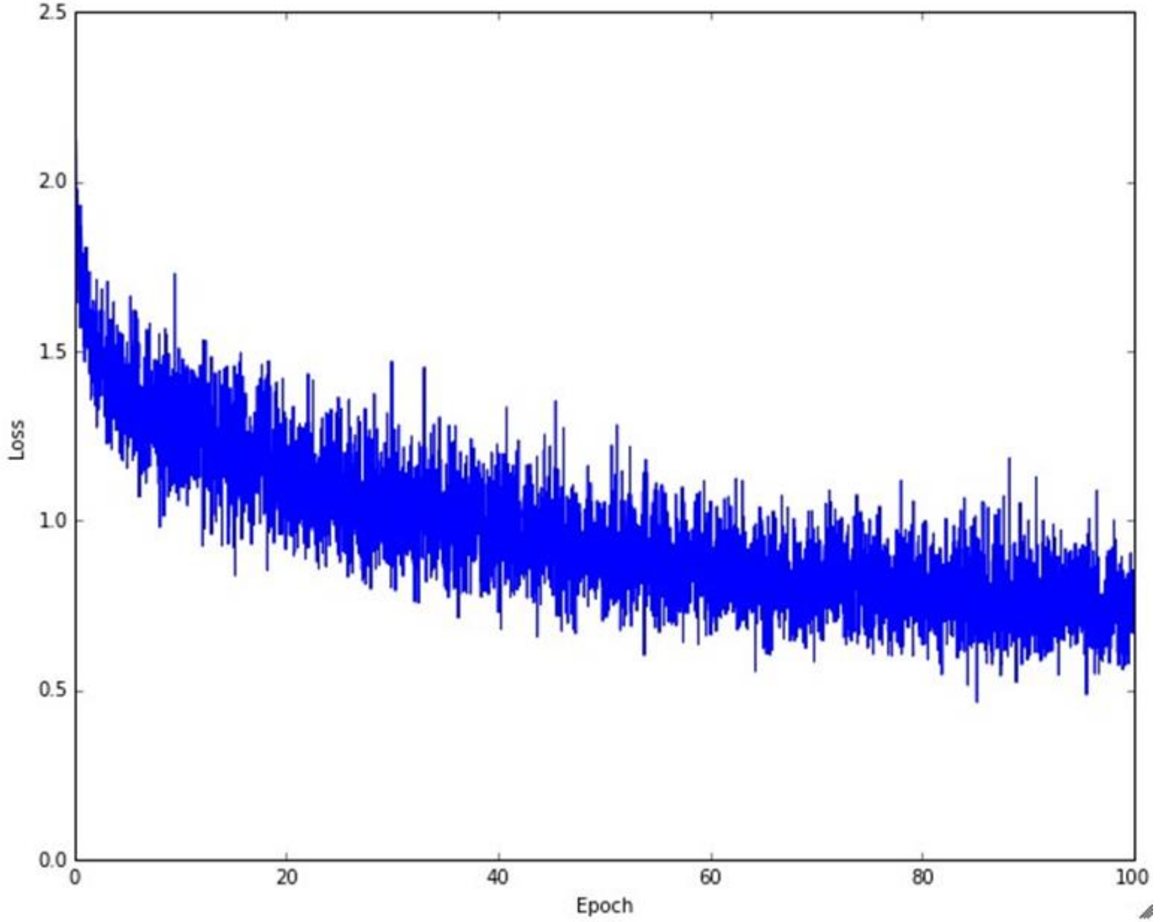
Learning Rate

$\theta^i = \theta^{i-1} - \eta \nabla L(\theta^{i-1})$



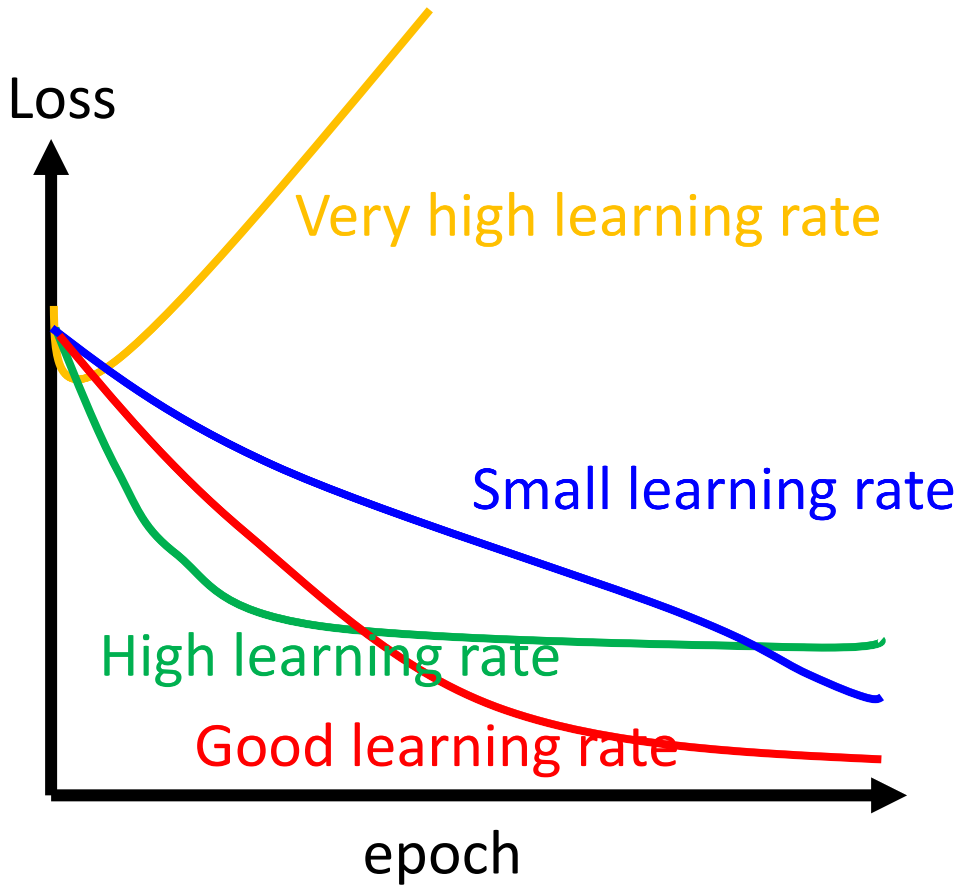
Set the learning rate η carefully

We can always visualize this.



- The loss function looks reasonable
- Might indicate a slightly too small learning rate based on its speed of decay
- The batch size might be a little too low (since the loss is a little too noisy)

The effects of different learning rates



Adaptive Learning Rates

- Popular & Simple Idea: Reduce the learning rate by some factor every few epochs.
  - At the beginning, we are far from the destination, so we use larger learning rate
  - After several epochs, we are close to the destination, so we reduce the learning rate
  - E.g. 1/t decay: $\eta^t = \eta / \sqrt{t+1}$
- Learning rate cannot be one-size-fits-all
  - Giving different parameters different learning rates

Adagrad

$$\eta^t = \frac{\eta}{\sqrt{t+1}} \qquad g^t = \frac{\partial L(\theta^t)}{\partial w}$$

- Divide the learning rate of each parameter by the ***root mean square of its previous deviation***

Vanilla Gradient descent

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

Adagrad

$$w^{t+1} \leftarrow w^t - \frac{\eta^t}{\sigma^t} g^t$$

Parameter dependent

Adagrad

- Divide the learning rate of each parameter by the ***root mean square of its previous derivatives***

$$\eta^t = \frac{\eta}{\cancel{\sqrt{t+1}}}$$

1/t decay

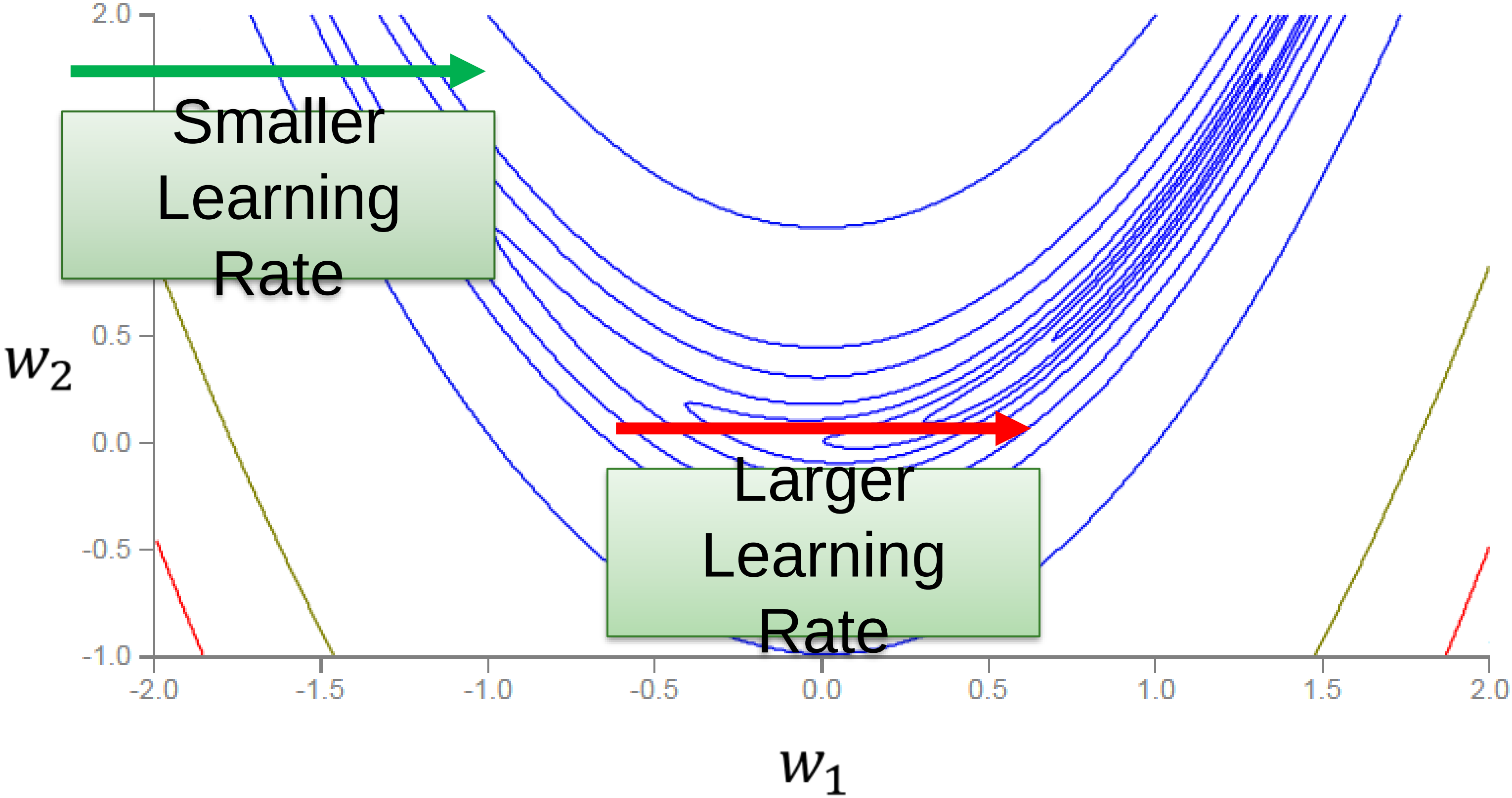
$$w^{t+1} \leftarrow w^t - \frac{\boxed{\eta^t}}{\boxed{\sigma^t}} g^t$$

$$\sigma^t = \sqrt{\cancel{\frac{1}{t+1}} \sum_{i=0}^{t} (g^i)^2}$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^{t} (g^i)^2}} g^t$$

RMSProp

Error Surface can be very complex when training NN.



RMSProp

$$w^1 \leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 \qquad \sigma^0 = g^0$$

$$w^2 \leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 \qquad \sigma^1 = \sqrt{\alpha(\sigma^0)^2 + (1-\alpha)(g^1)^2}$$

$$w^3 \leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 \qquad \sigma^2 = \sqrt{\alpha(\sigma^1)^2 + (1-\alpha)(g^2)^2}$$

$$\vdots$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}$$

Root Mean Square of the gradients with previous gradients being decayed

RMSProp

$$s_{dw} = \beta s_{dw} + (1 - \beta) dW^2$$

$$s_{db} = \beta s_{db} + (1 - \beta) db^2$$

$$W = W - \alpha \frac{dW}{\sqrt{s_{dw}} + \varepsilon}$$

$$b = b - \alpha \frac{db}{\sqrt{s_{db}} + \varepsilon}$$

Adam

RMSProp + Momentum

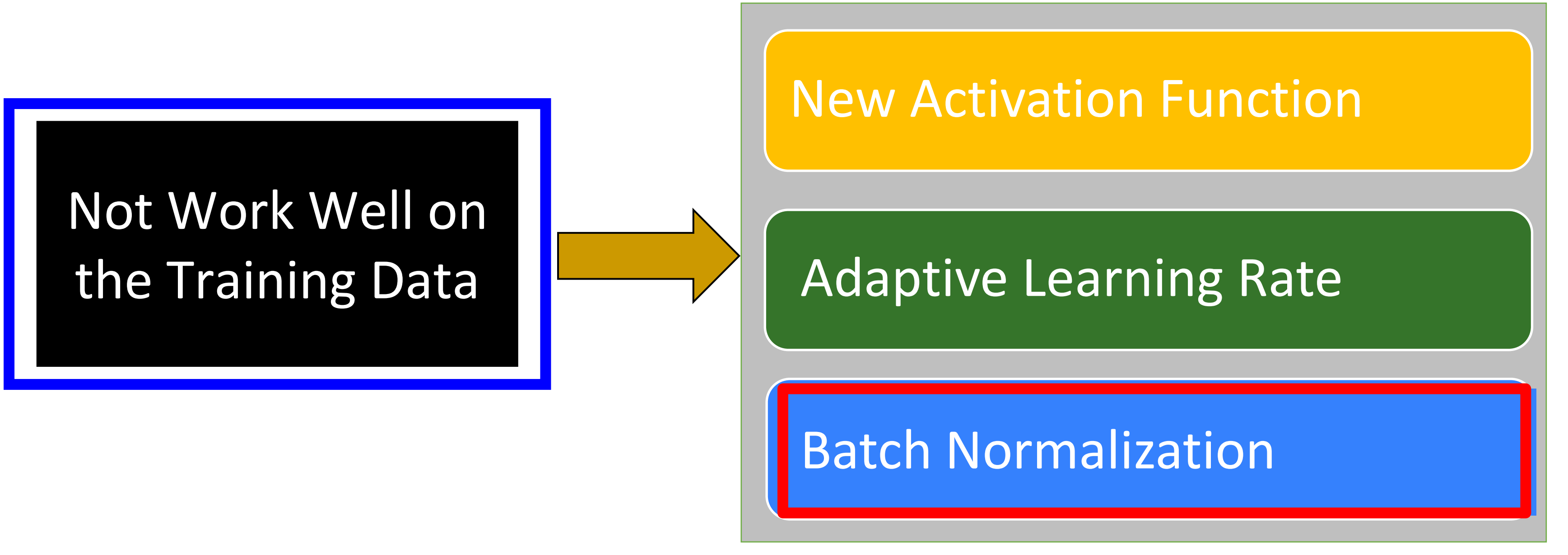
Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

for momentum

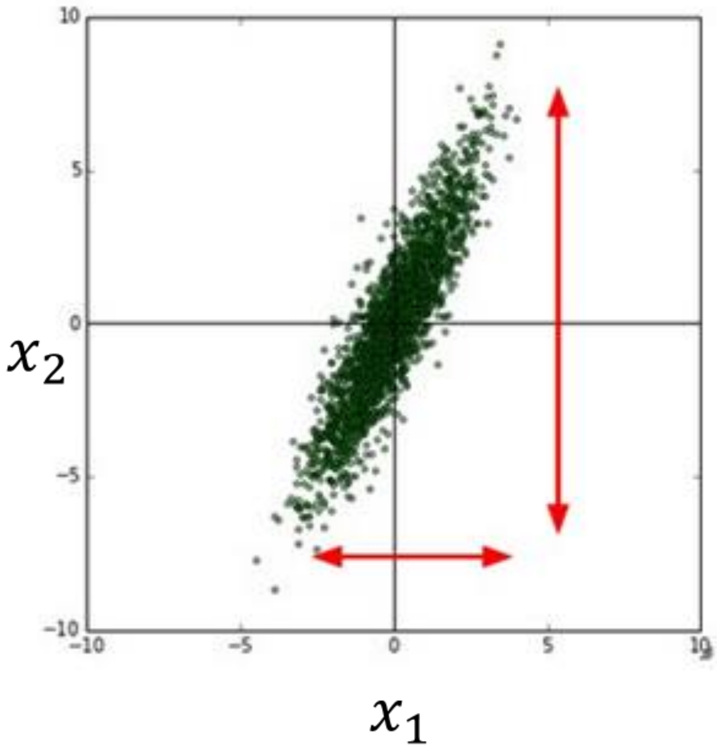
for RMSProp

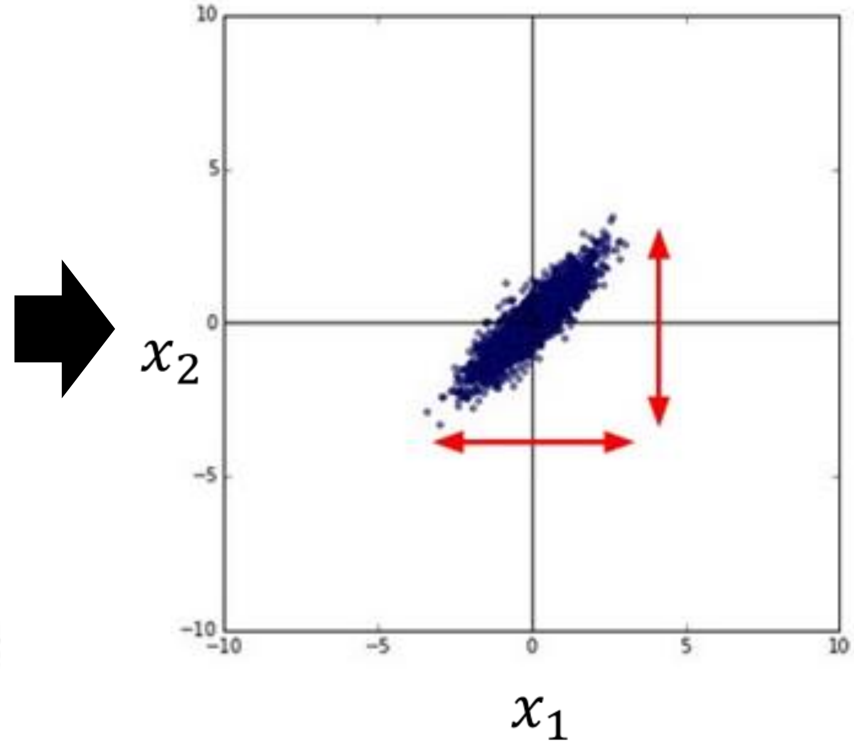
Not Work Well on the Training Data



Feature Scaling

$$y = w_1 x_1 + w_2 x_2 + b$$

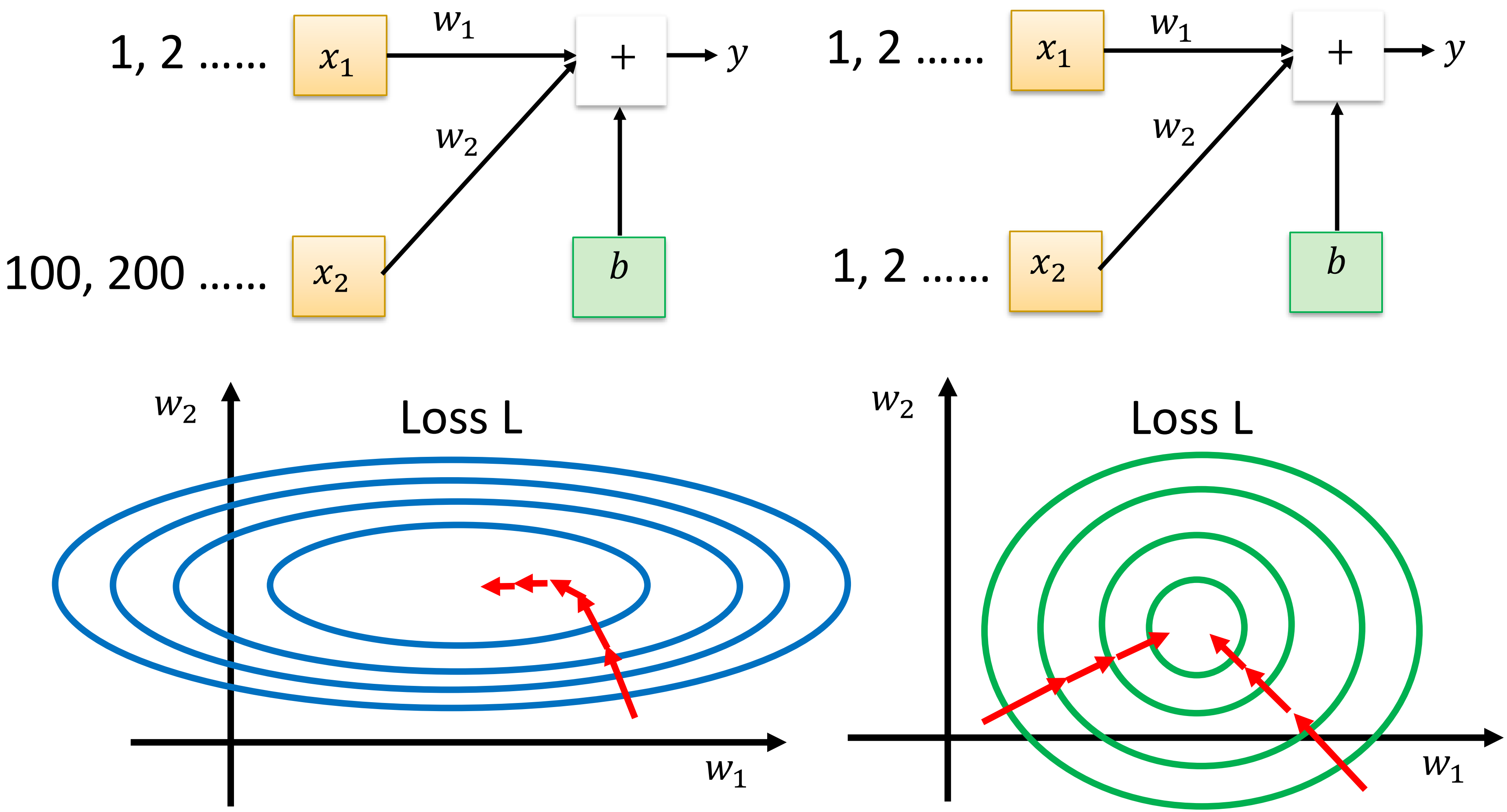




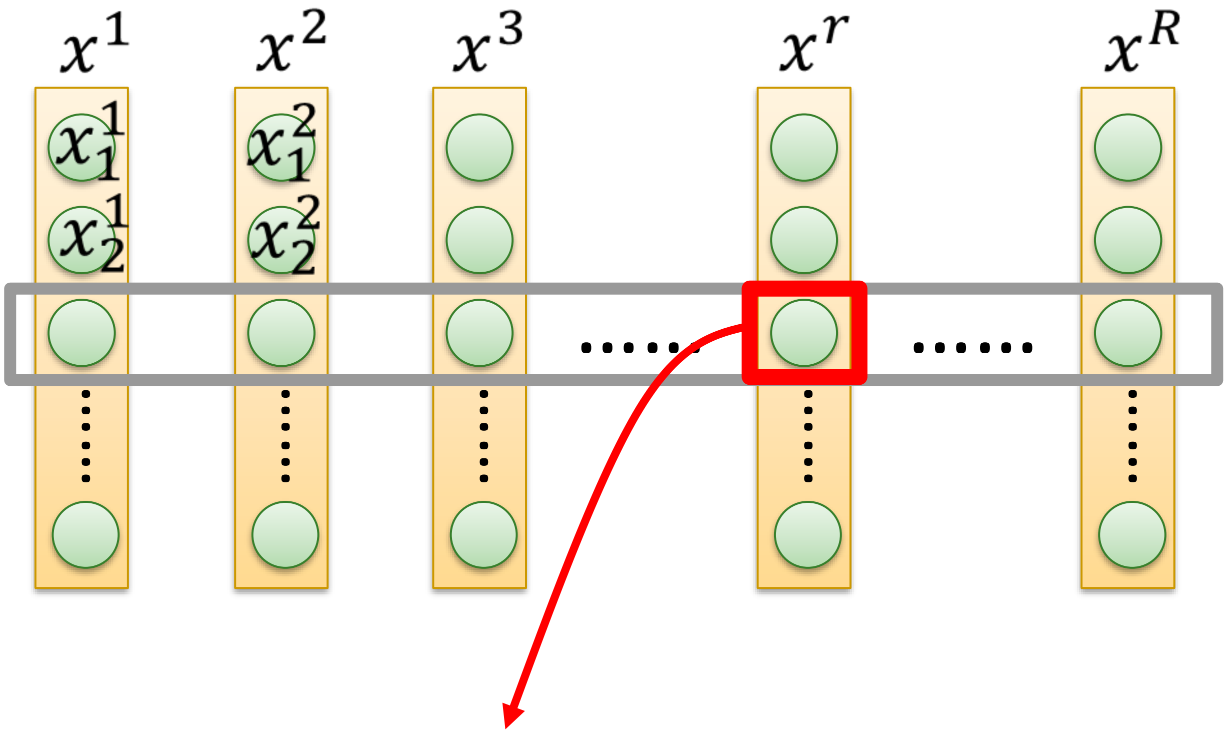
Make different features have the same scaling.

Feature Scaling

$$y = w_1 x_1 + w_2 x_2 + b$$



Feature Scaling



For each dimension i:

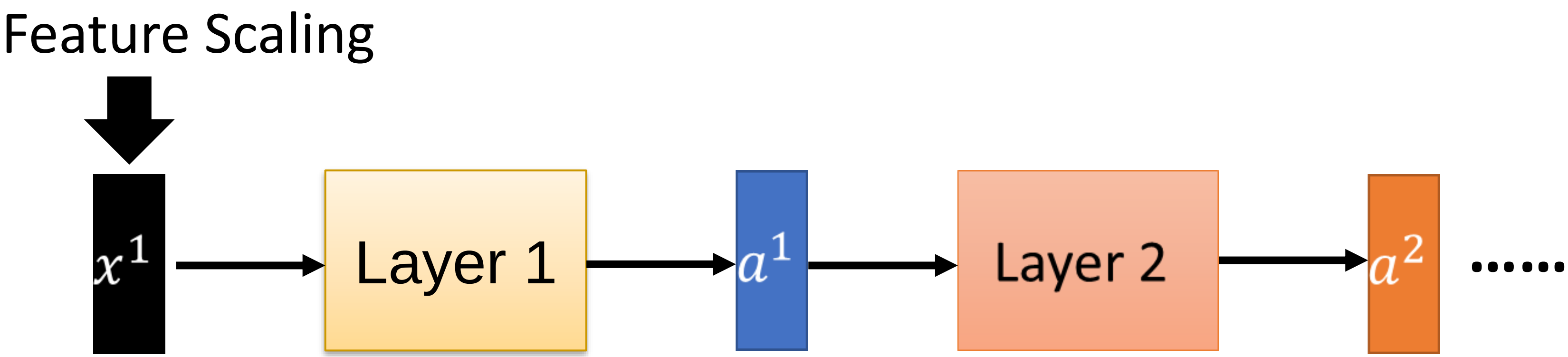
mean: m_i

standard deviation: σ_i

$$x_i^r \leftarrow \frac{x_i^r - m_i}{\sigma_i}$$

In general, gradient descent converges much faster with feature scaling.

How about Hidden Layer?



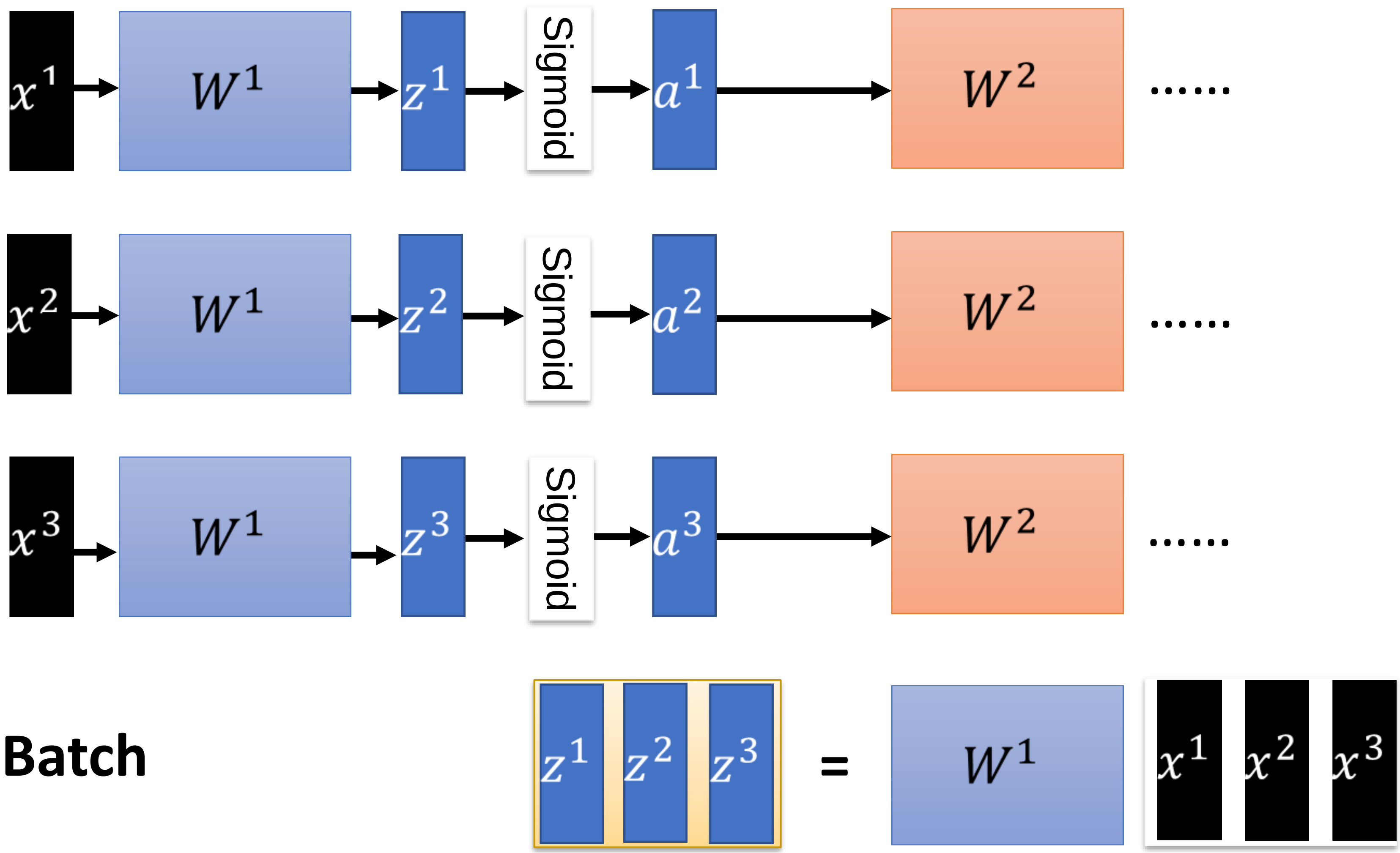
Internal Covariate Shift

Smaller learning rate can be helpful,
but the training would be slower.

Batch normalization

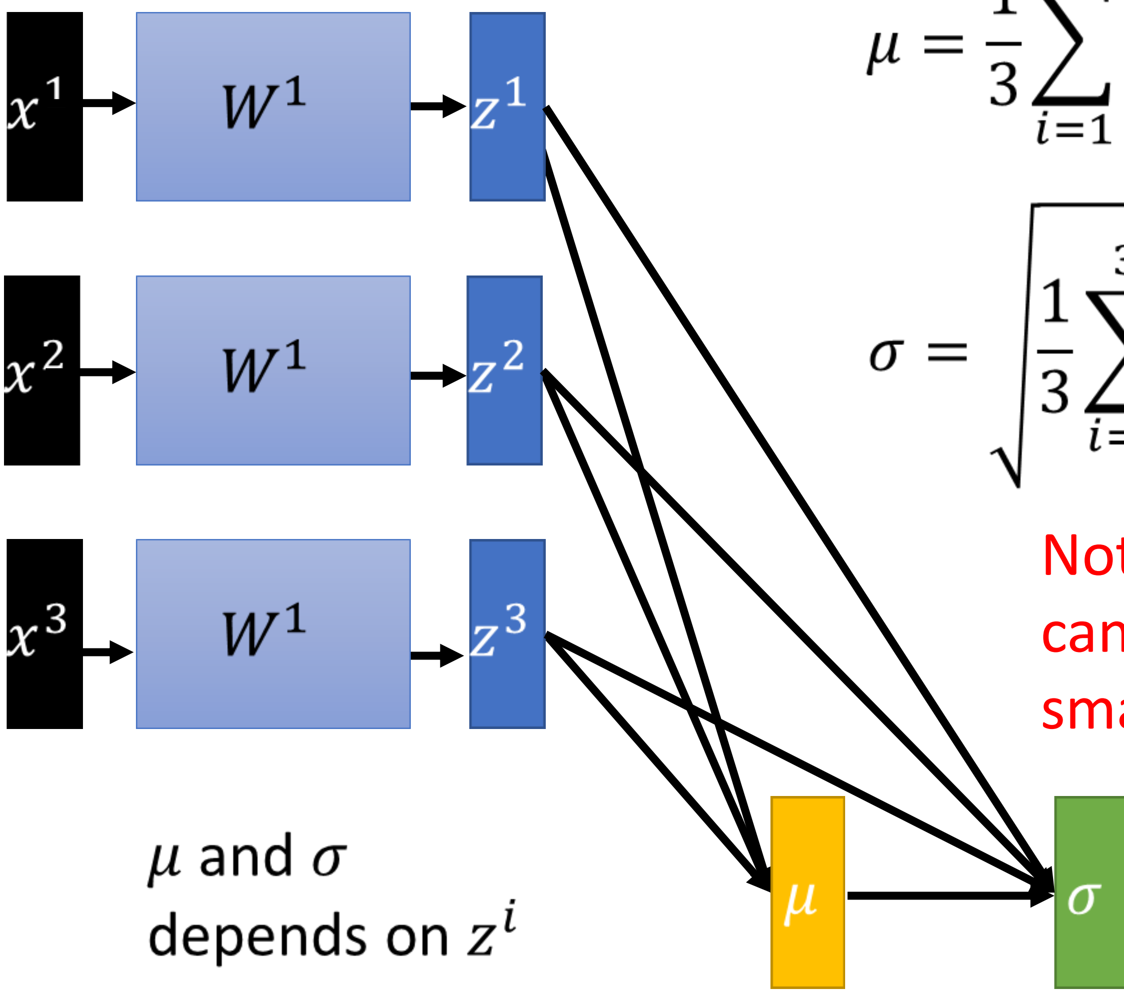
Sergey Ioffe，Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. 2015

Batch



x^1
W^1
z^1
Sigmoid
a^1
W^2
……
x^2
W^1
z^2
Sigmoid
a^2
W^2
……
x^3
W^1
z^3
Sigmoid
a^3
W^2
……
Batch
z^1 z^2 z^3 = W^1 x^1 x^2 x^3

Batch normalization



$$\mu = \frac{1}{3}\sum_{i=1}^{3} z^i$$

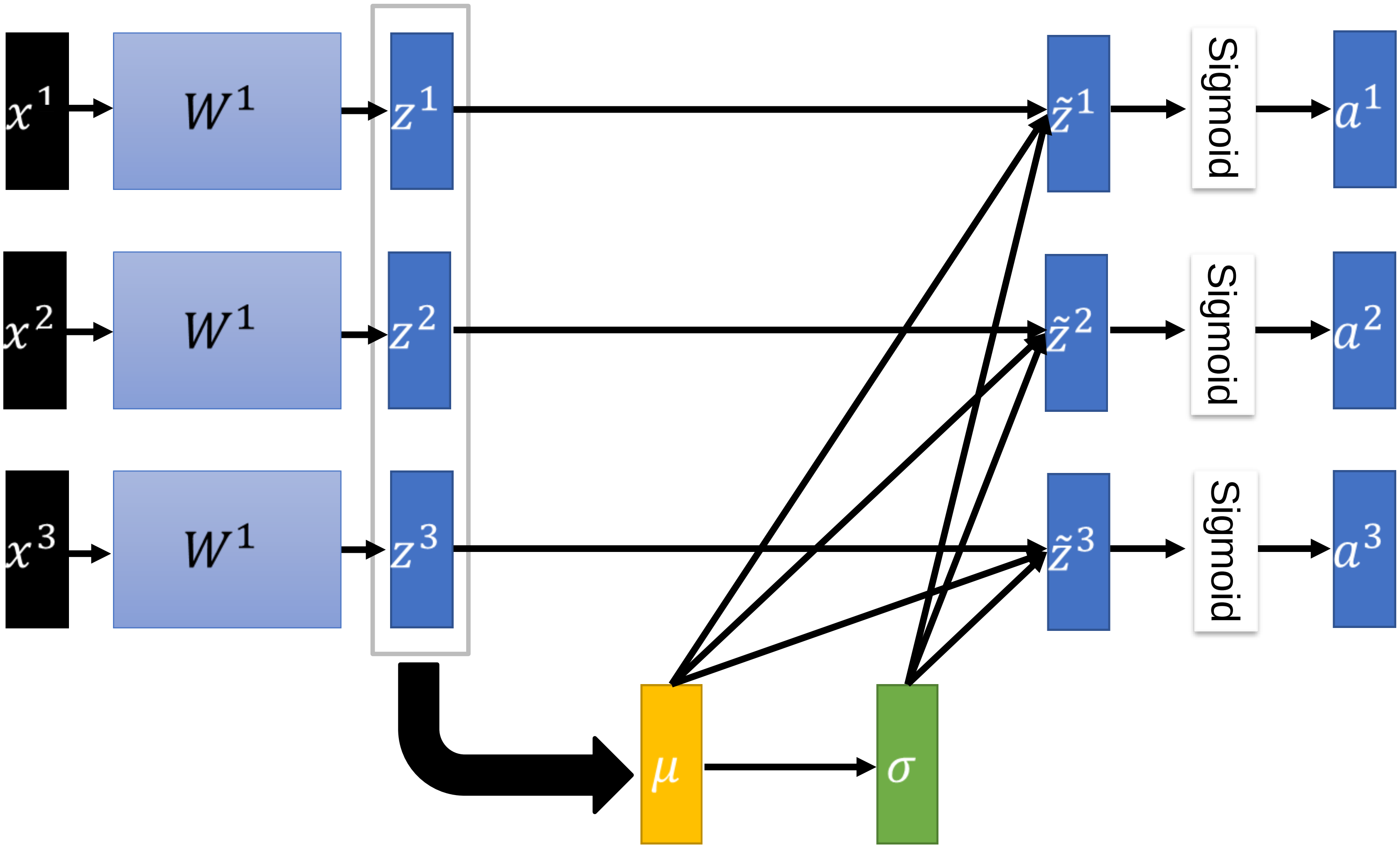
$$\sigma = \sqrt{\frac{1}{3}\sum_{i=1}^{3} (z^i - \mu)^2}$$

Note: Batch normalization cannot be applied on small batch.

μ and σ depends on z^i

Batch normalization

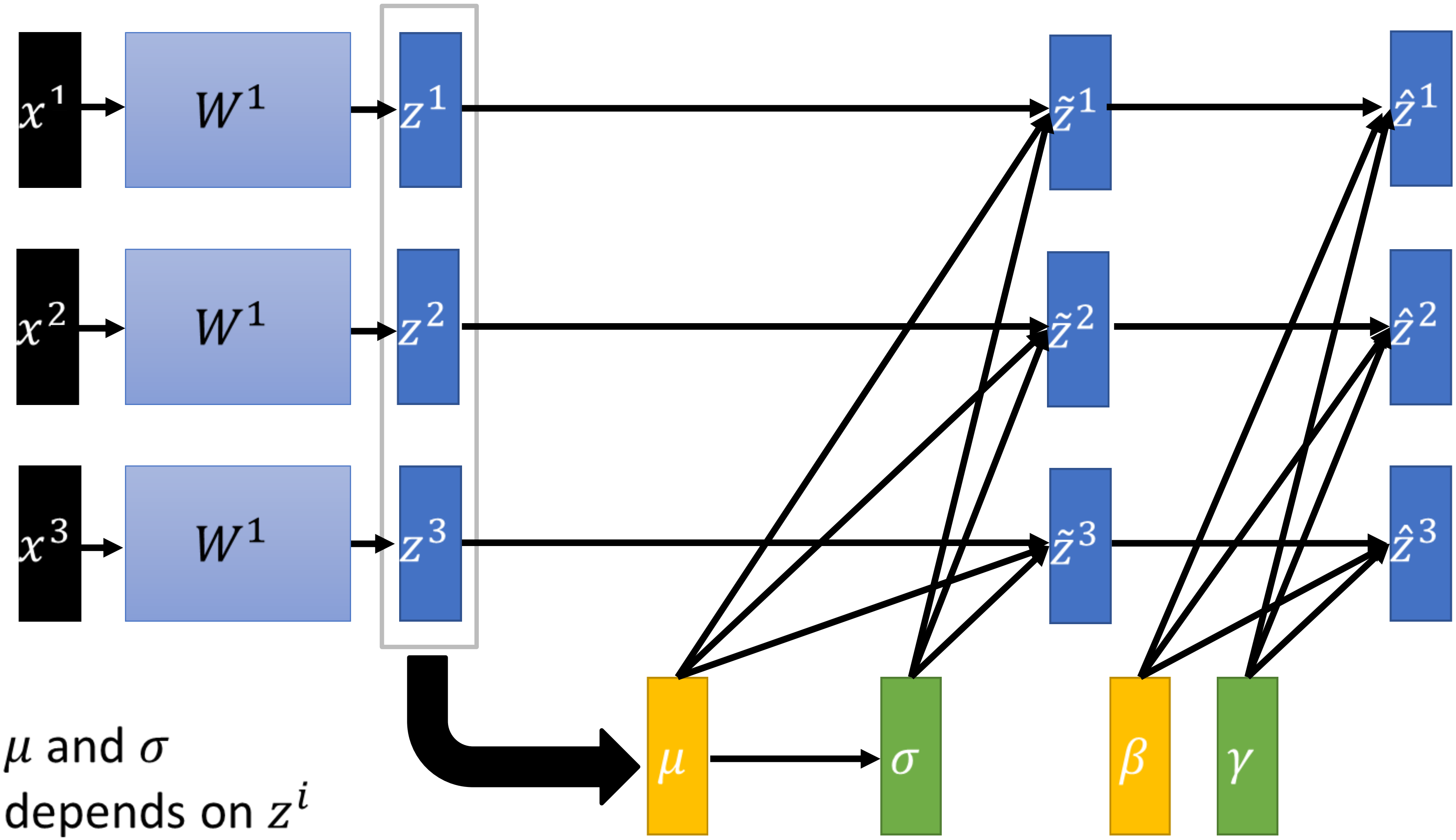
$$\tilde{z}^i = \frac{z^i - \mu}{\sigma}$$



Batch normalization

$$\tilde{z}^i = \frac{z^i - \mu}{\sigma}$$

$$\hat{z}^i = \gamma \odot \tilde{z}^i + \beta$$



μ and σ depends on z^i

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

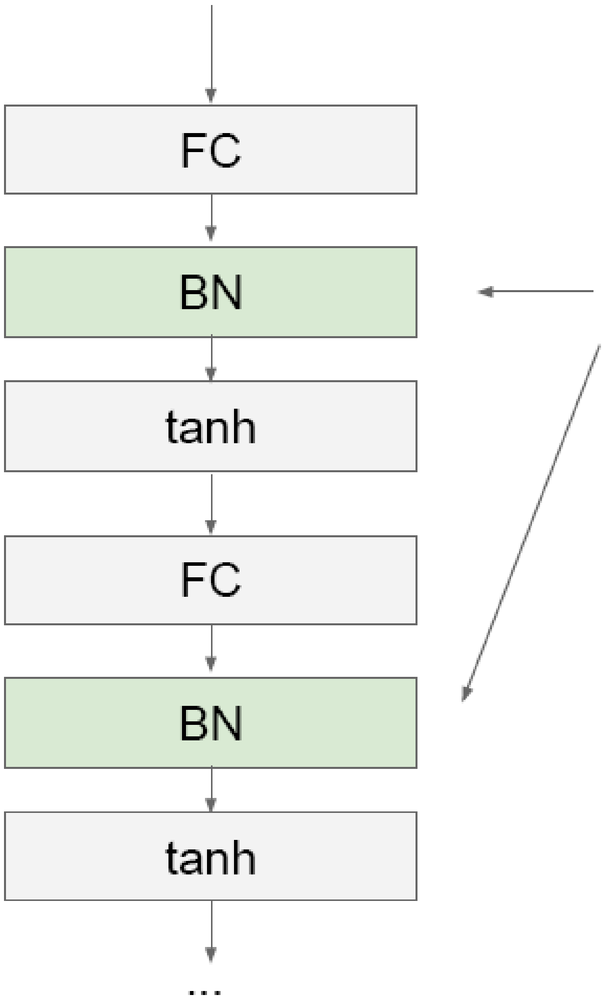
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

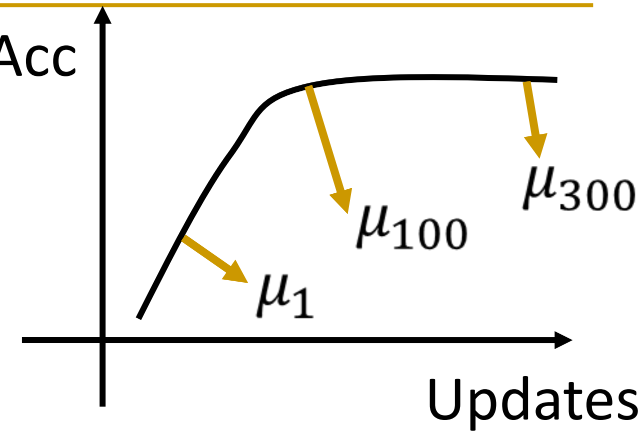
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization

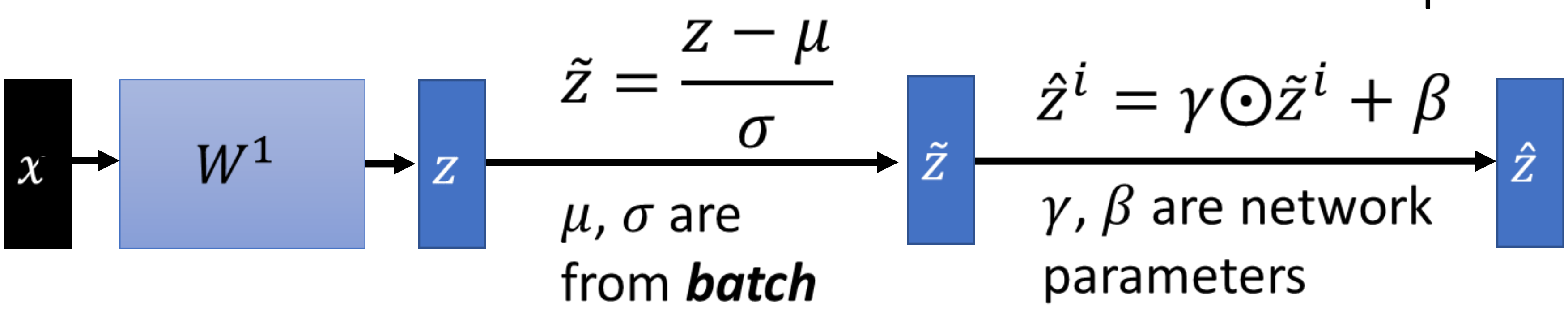


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Batch normalization



- At testing stage:



We do not have ***batch*** at testing stage.

Ideal solution:

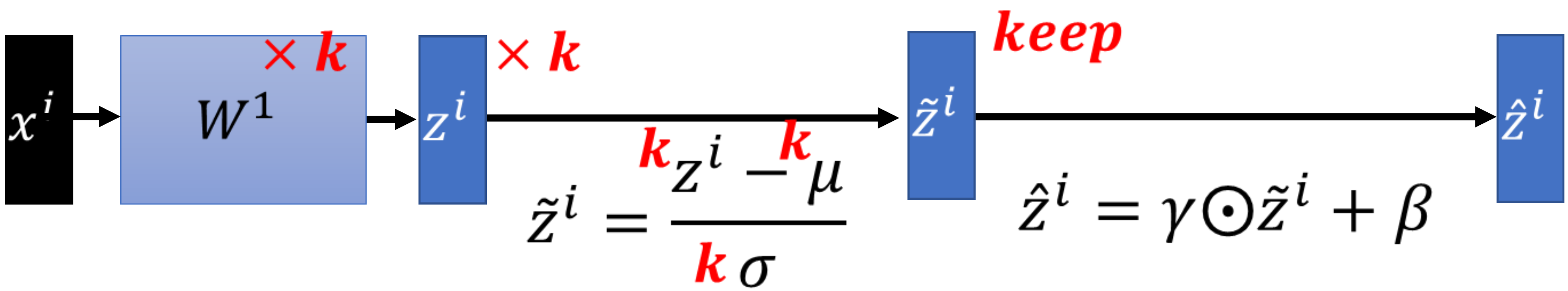
Computing μ and σ using the whole training dataset.

Practical solution:

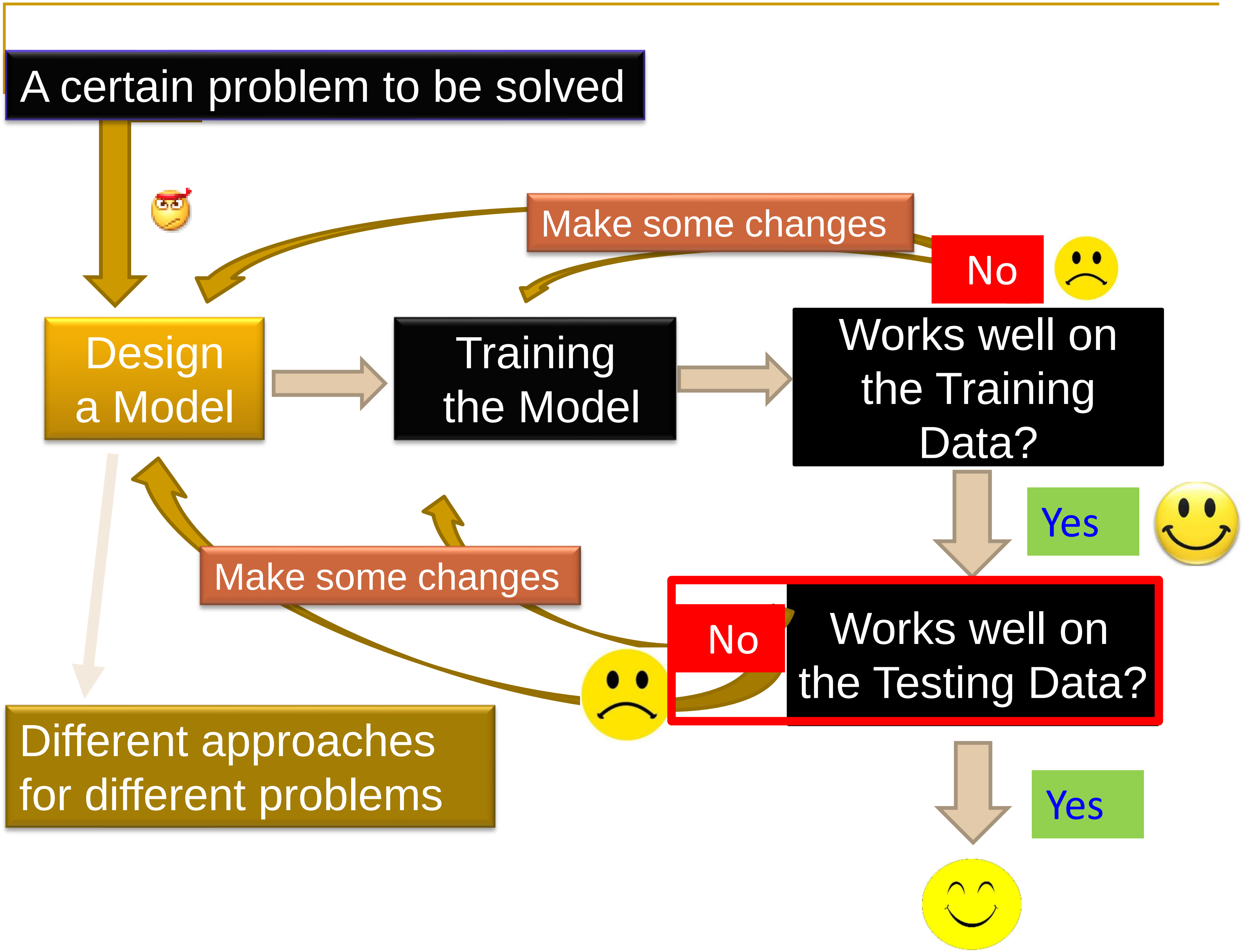
Computing the moving average of μ and σ of the batches during training.

Batch normalization - Benefit

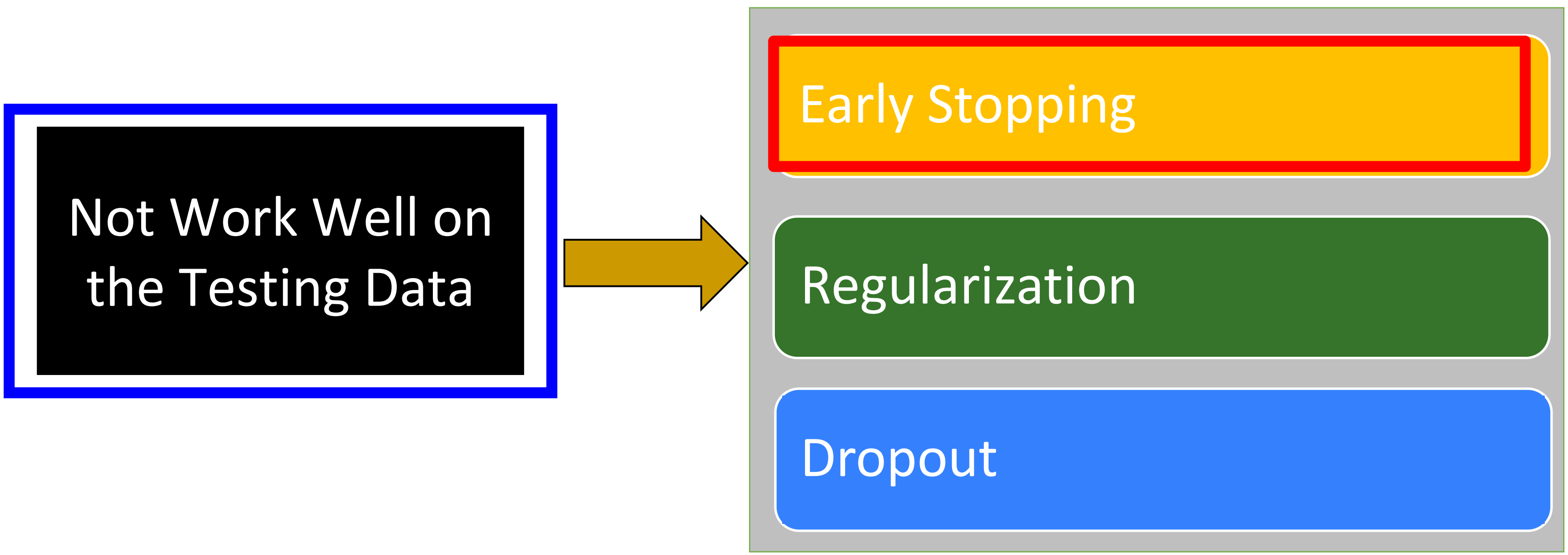
- BN reduces training times, and make very deep net trainable.
  - Because of less Covariate Shift, we can use larger learning rates.
  - Less exploding/vanishing gradients
    - Especially effective for sigmoid, tanh, etc.
- Learning is less affected by initialization.



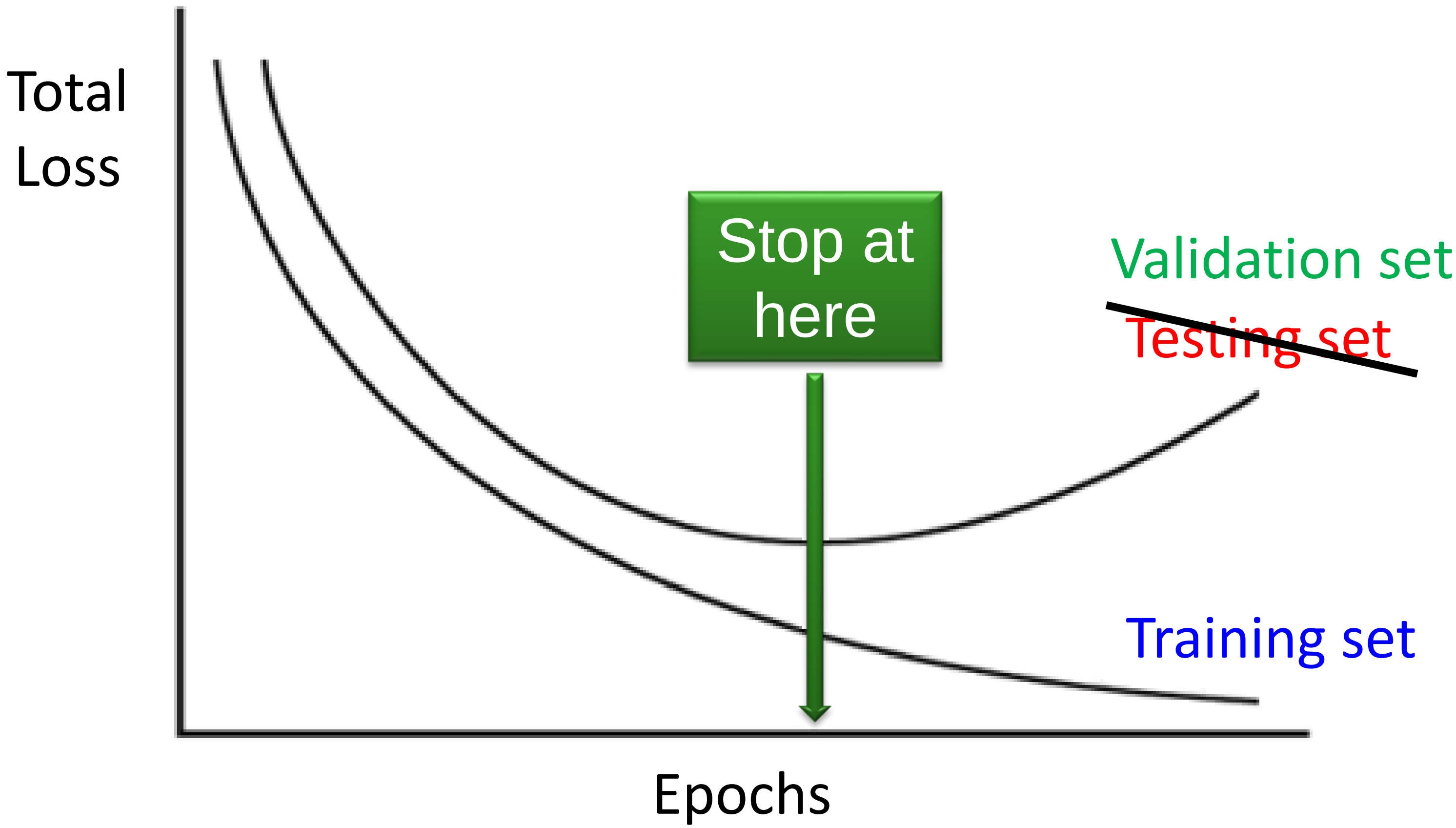
- BN reduces the demand for regularization.


A certain problem to be solved
Make some changes
No
Design
a Model
Training
the Model
Works well on
the Training
Data?
Yes
Make some changes
No
Works well on
the Testing Data?
Different approaches
for different problems
Yes

Not Work Well on the Testing Data

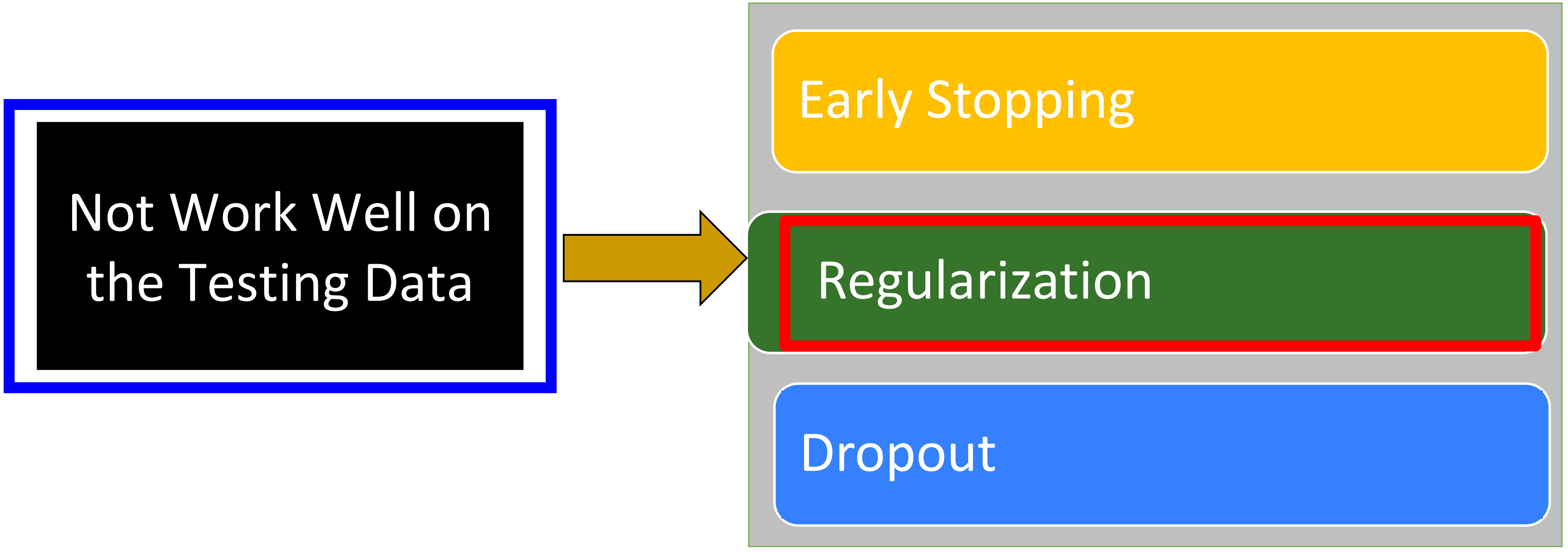


Early Stopping



Keras: http://keras.io/getting-started/faq/#how-can-i-interrupt-training-when-the-validation-loss-isnt-decreasing-anymore

Not Work Well on the Testing Data



Regularization

- New loss function to be minimized
  - Find a set of weight not only minimizing original cost but also close to zero

$$\mathrm{L}'(\theta) = \underbrace{L(\theta)}_{\text{Original loss (e.g. minimize square error, cross entropy ...)}} + \lambda \frac{1}{2} \|\theta\|_2$$

→ Regularization term

Original loss
(e.g. minimize square
error, cross entropy ...)

$$\theta = \{w_1, w_2, \ldots\}$$

L2 regularization:

$$\|\theta\|_2 = (w_1)^2 + (w_2)^2 + \cdots$$

(usually not consider biases)

Regularization

L2 regularization:

$$\|\theta\|_2 = (w_1)^2 + (w_2)^2 + \cdots$$

- New loss function to be minimized

$$\mathrm{L}'(\theta) = L(\theta) + \lambda \frac{1}{2} \|\theta\|_2$$

Gradient: $\frac{\partial \mathrm{L}'}{\partial w} = \frac{\partial \mathrm{L}}{\partial w} + \lambda w$

Update: $w^{t+1} \leftarrow w^t - \eta \frac{\partial \mathrm{L}'}{\partial w} = w^t - \eta \left(\frac{\partial \mathrm{L}}{\partial w} + \lambda w^t \right)$

$$= \underbrace{(1 - \eta\lambda) w^t}_{\text{Close to zero}} - \eta \frac{\partial \mathrm{L}}{\partial w}$$

Weight Decay

Regularization

L1 regularization:

$$\|\theta\|_1 = |w_1| + |w_2| + \cdots$$

- New loss function to be minimized

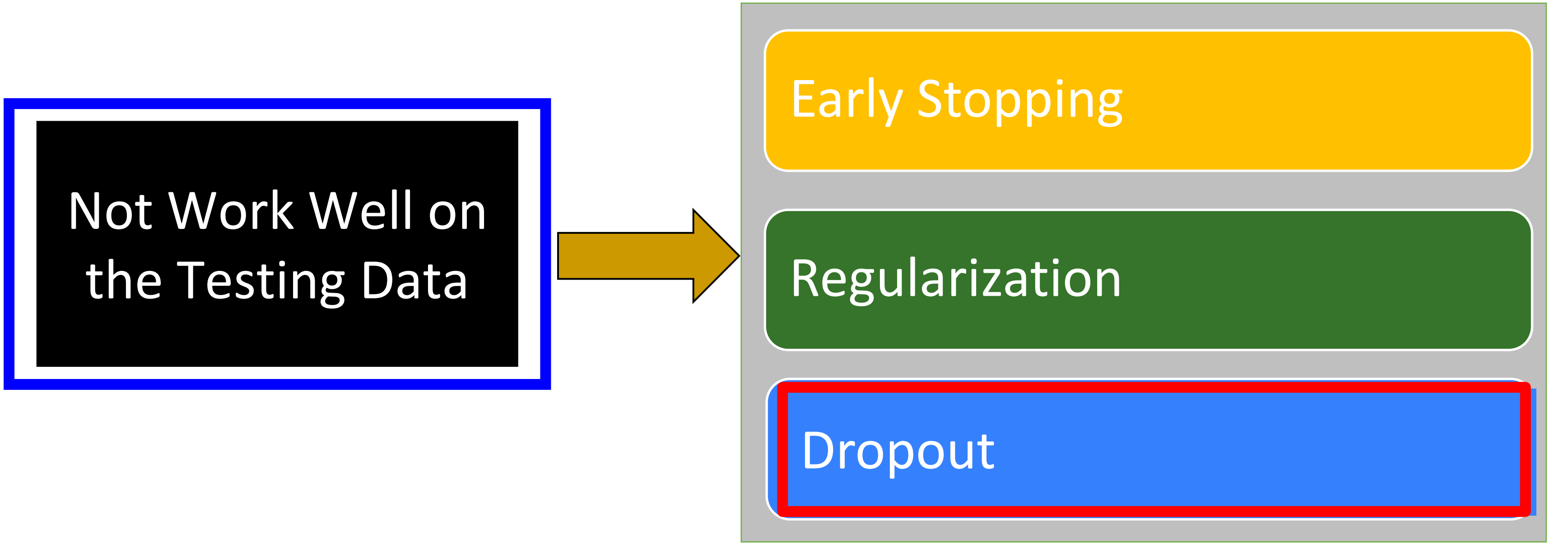
$$\mathrm{L}'(\theta) = L(\theta) + \lambda\|\theta\|_1 \qquad \frac{\partial \mathrm{L}'}{\partial w} = \frac{\partial \mathrm{L}}{\partial w} + \lambda\,\mathrm{sgn}(w)$$

Update:

$$w^{t+1} \leftarrow w^t - \eta\frac{\partial \mathrm{L}'}{\partial w} = w^t - \eta\left(\frac{\partial \mathrm{L}}{\partial w} + \lambda\,\mathrm{sgn}(w^t)\right)$$

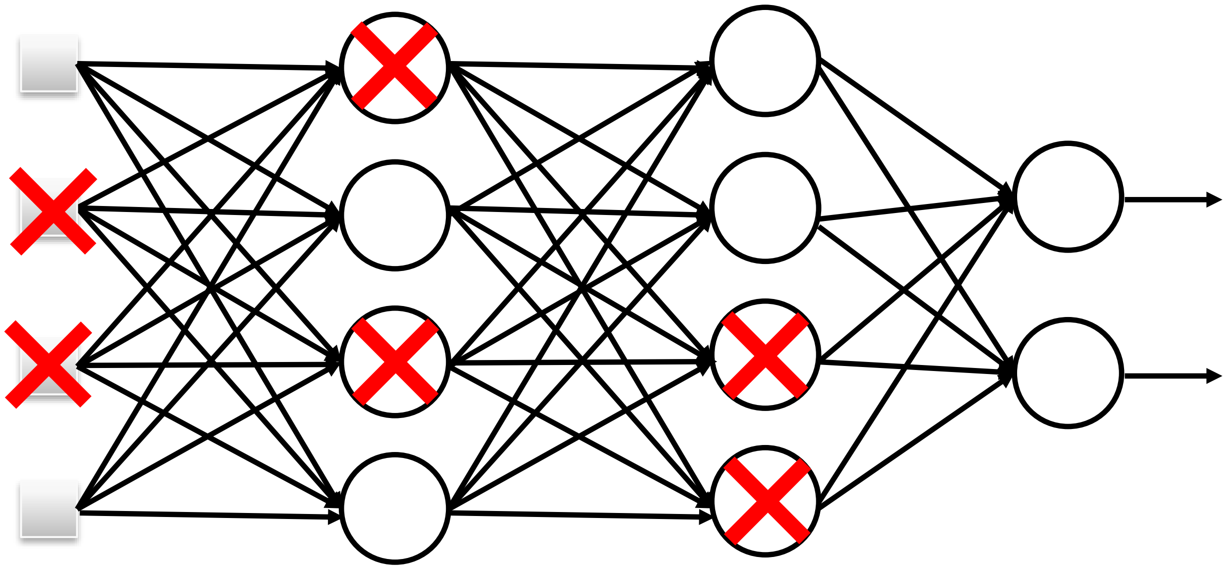
$$= \underline{w^t - \eta\lambda\,\mathrm{sgn}(w^t)} - \eta\frac{\partial \mathrm{L}}{\partial w}$$

Not Work Well on the Testing Data



Dropout

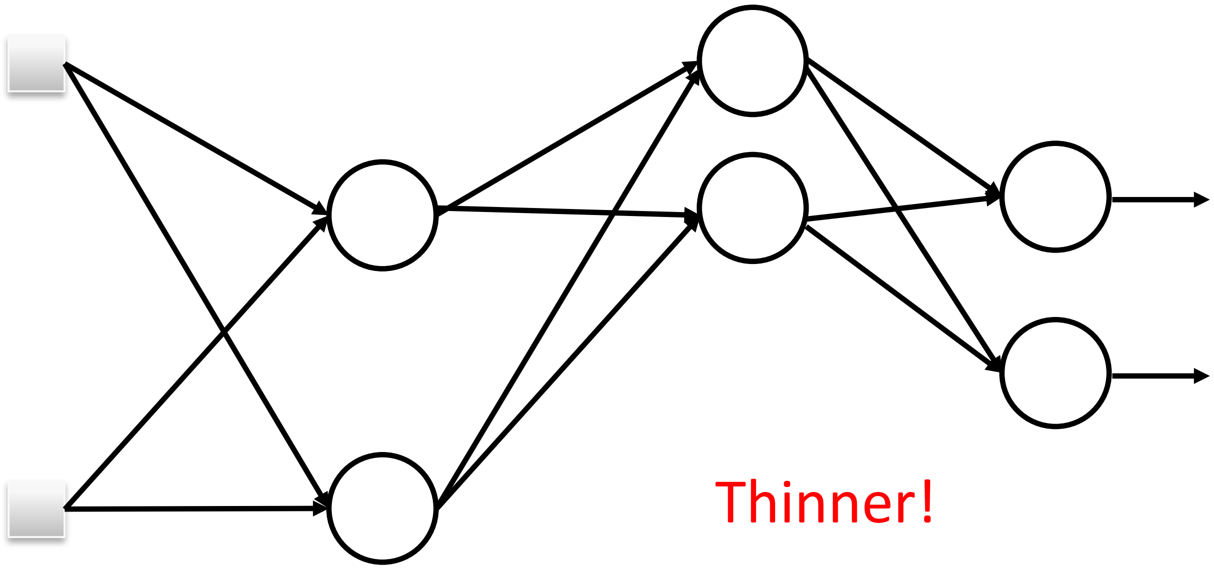
Training:



- **Each time before updating the parameters**
  - Each neuron has a probability of *p* to be dropouted

Dropout

Training:



➢ **Each time before updating the parameters**

- Each neuron has a probability of p to be dropouted

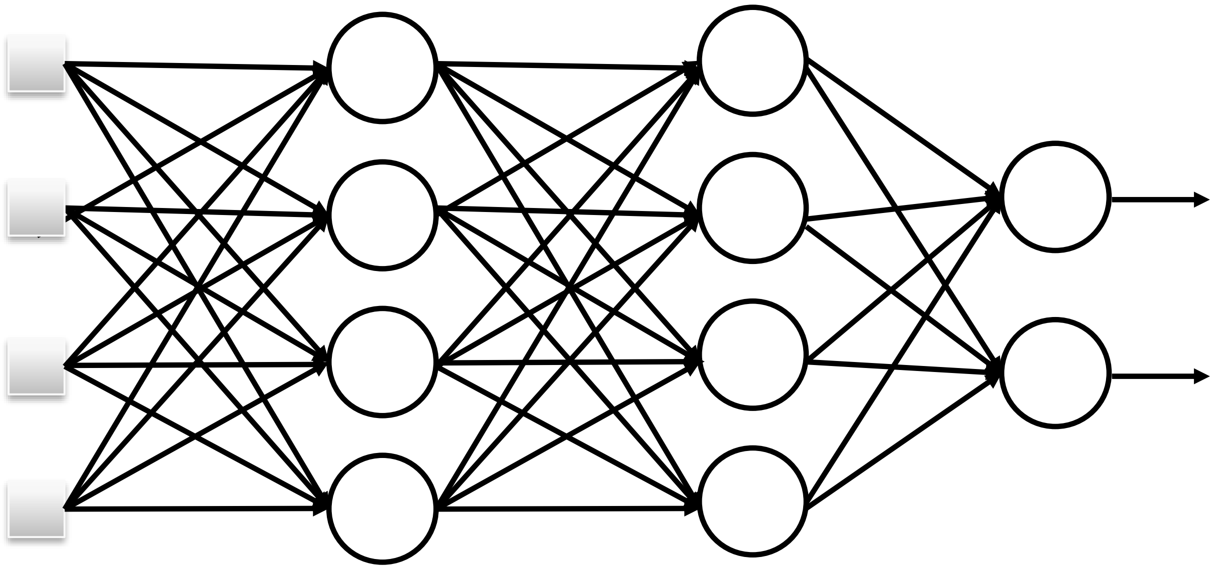
  ⇒ **The structure of the network is changed.**

- Using the new network for training

For each mini-batch, we resample the dropout neurons.

Dropout

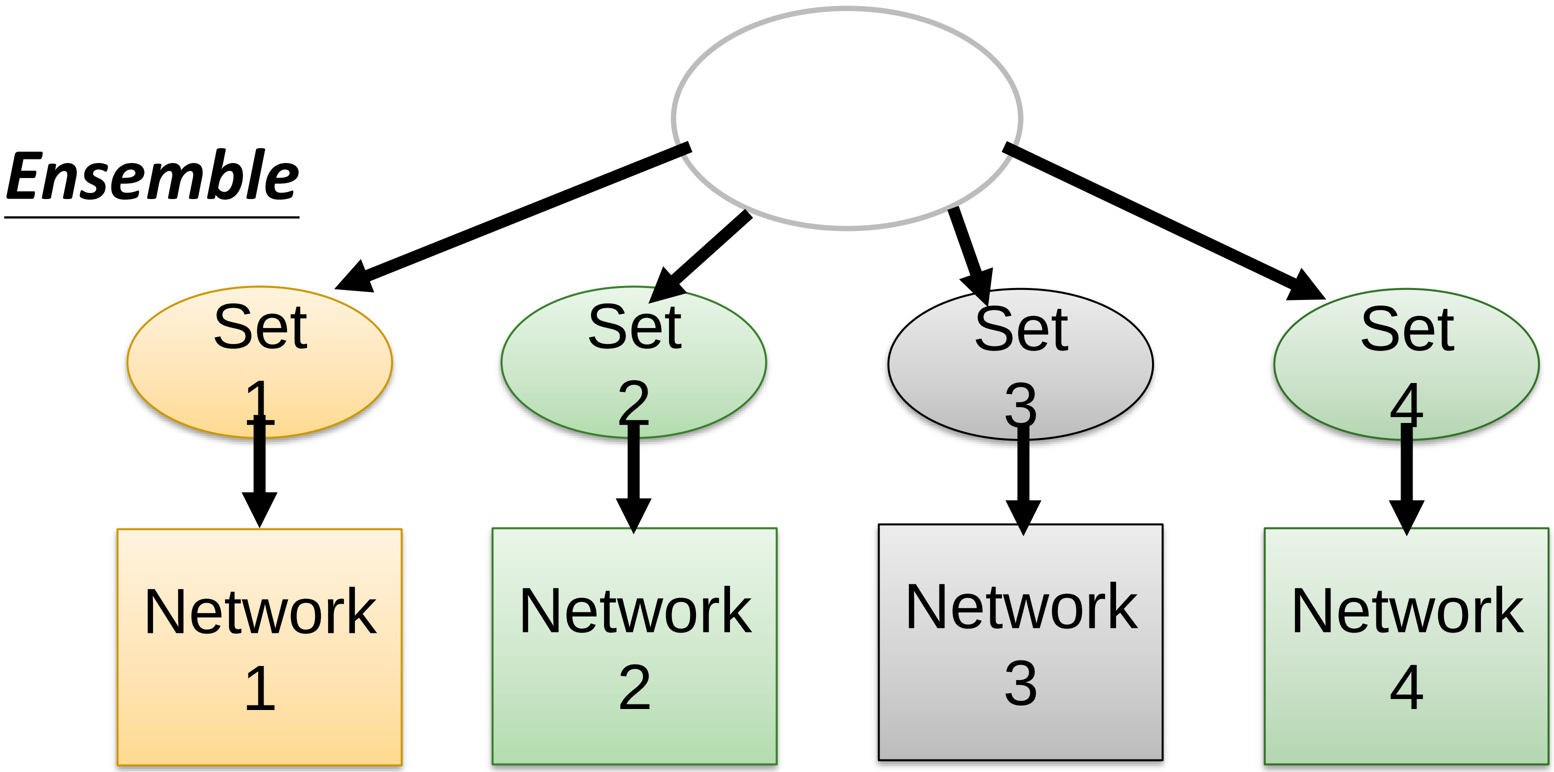
Testing:



➢ **No dropout**

- If the dropout rate at training is p, all the weights times 1-p
- Assume that the dropout rate is 0.5
  If a weight $w = 1$ by training, set $w = 0.5$ for testing.

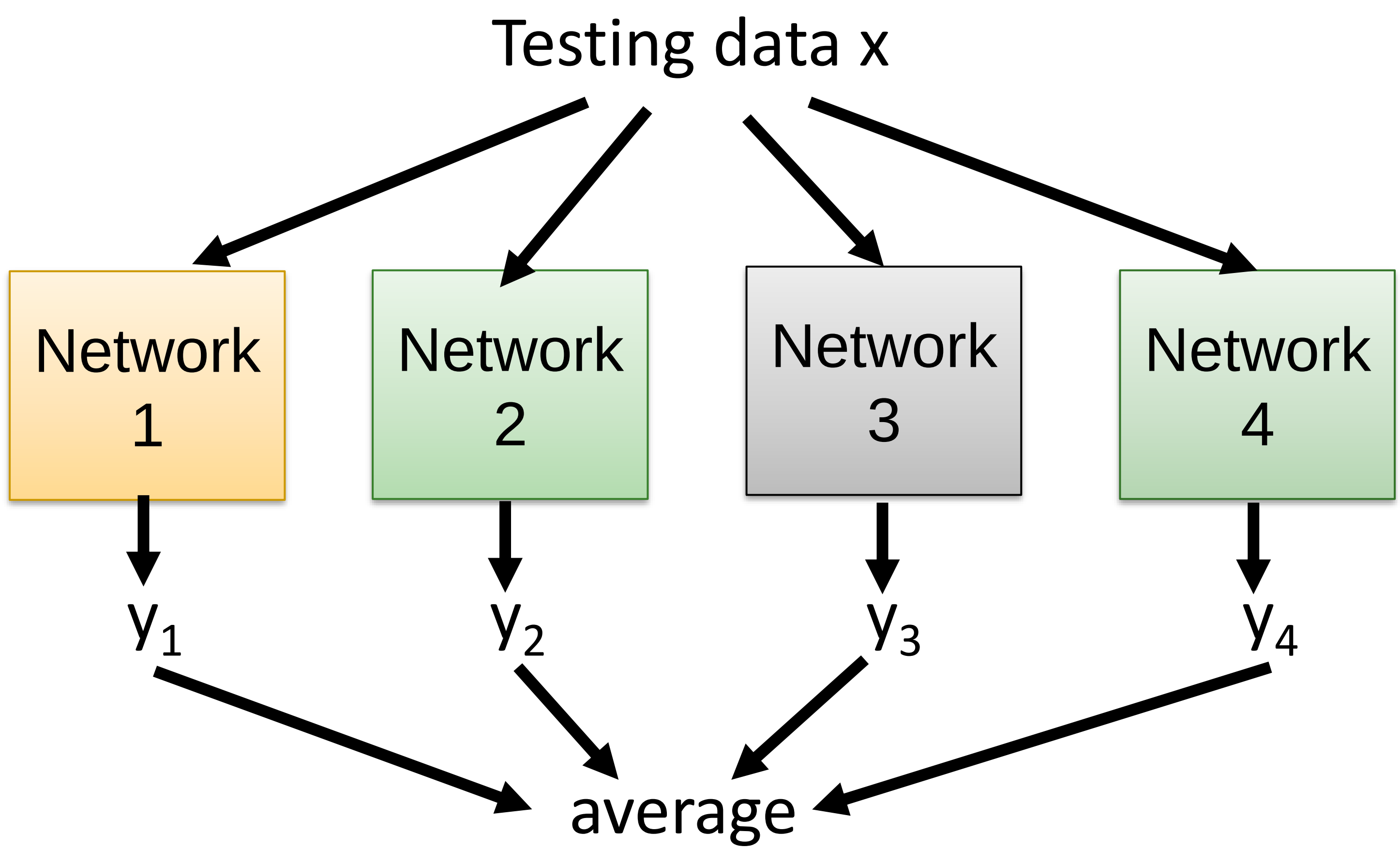
Dropout is a kind of ensemble.



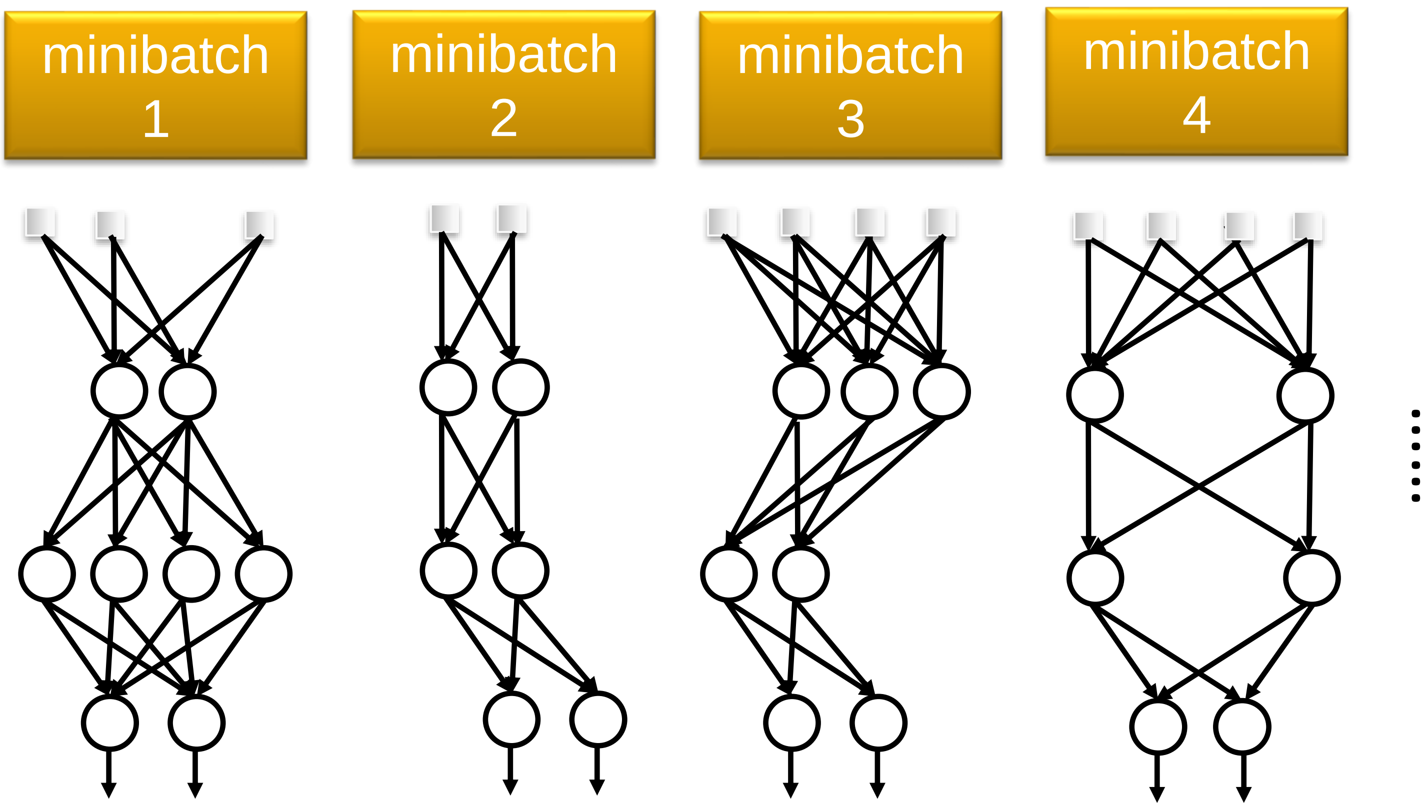
Train a bunch of networks with different structures

Dropout is a kind of ensemble.

Ensemble



Dropout is a kind of ensemble.



Training of Dropout

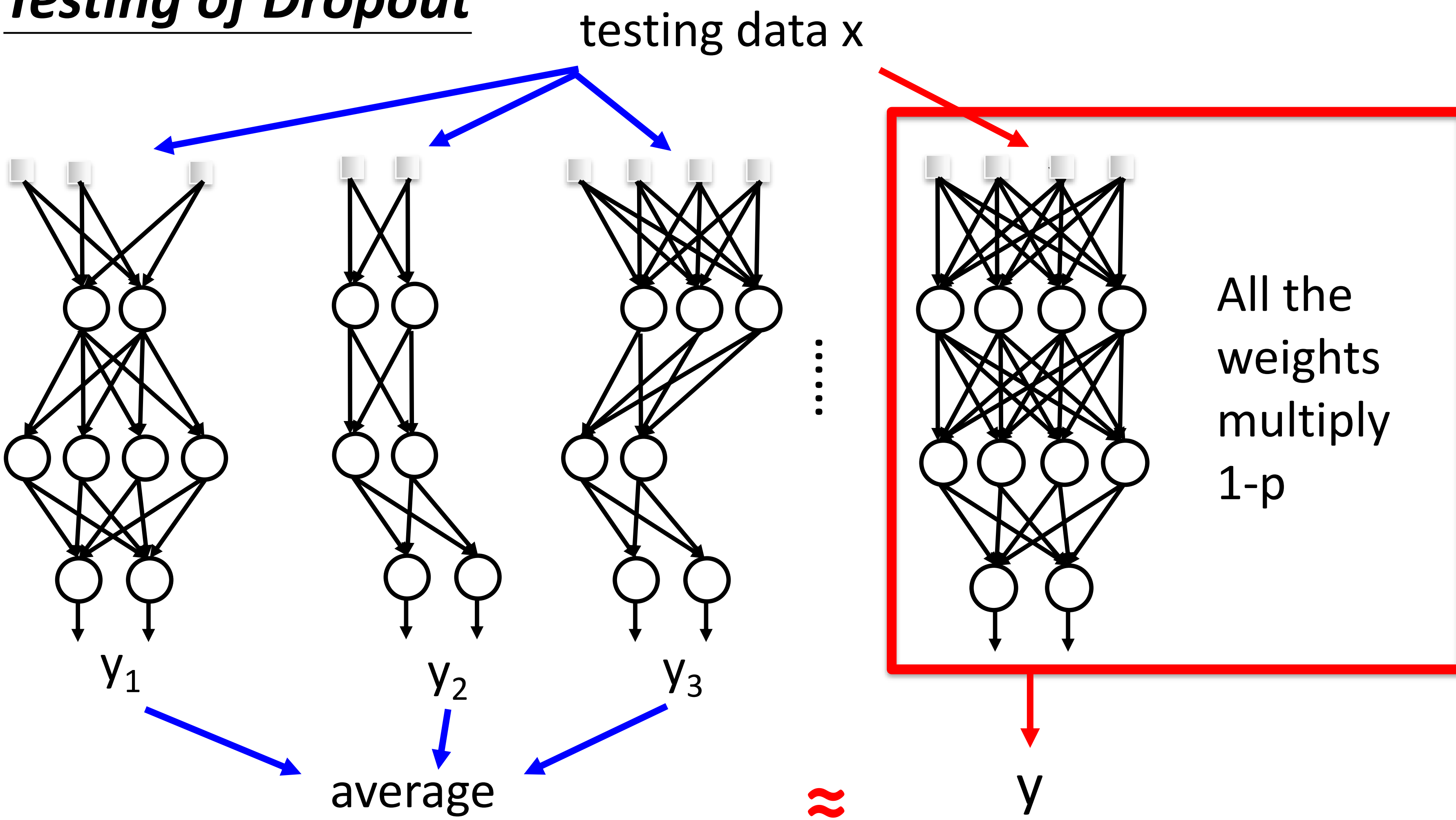
n neurons

2^n possible networks

- Using one mini-batch to train one network
- Some parameters in the network are shared

Dropout is a kind of ensemble.

Testing of Dropout



Outline

- Hyperparameters & Parameters
- Setting up the data
- Gradient Descent
- Learning rate
- Batch Normalization
- Early stopping
- Regularization
- Dropout
- Hyperparameter tuning

Hyperparameter Tuning

- Choices about the algorithm that we set rather than learn
- Come up very often in the design of many Machine Learning algorithms that learn from data.
- Often not obvious & Very problem-dependent : (

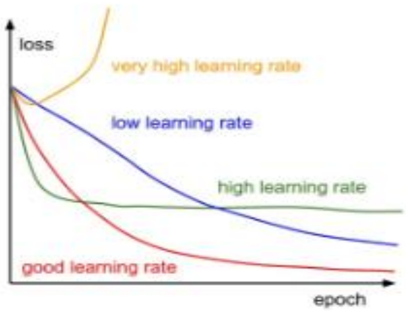
- Just **try** and **try** and **try** and see what works best（e.g. the predicted class scores are consistent with the ground truth labels）

Four main strategies for searching for the best configuration

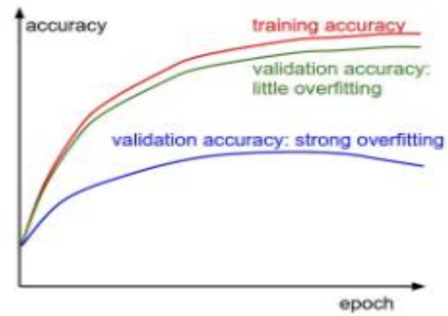
- Babysitting
- Grid Search
- Random Search
- Automatic Hyperparameter Tuning

Babysitting: Manually

Act/Grad/Filter

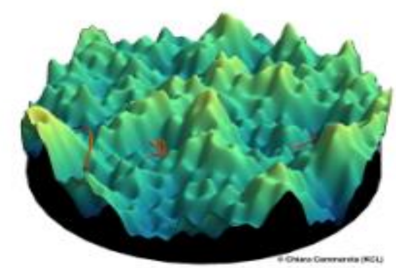


Loss



Metric

Data

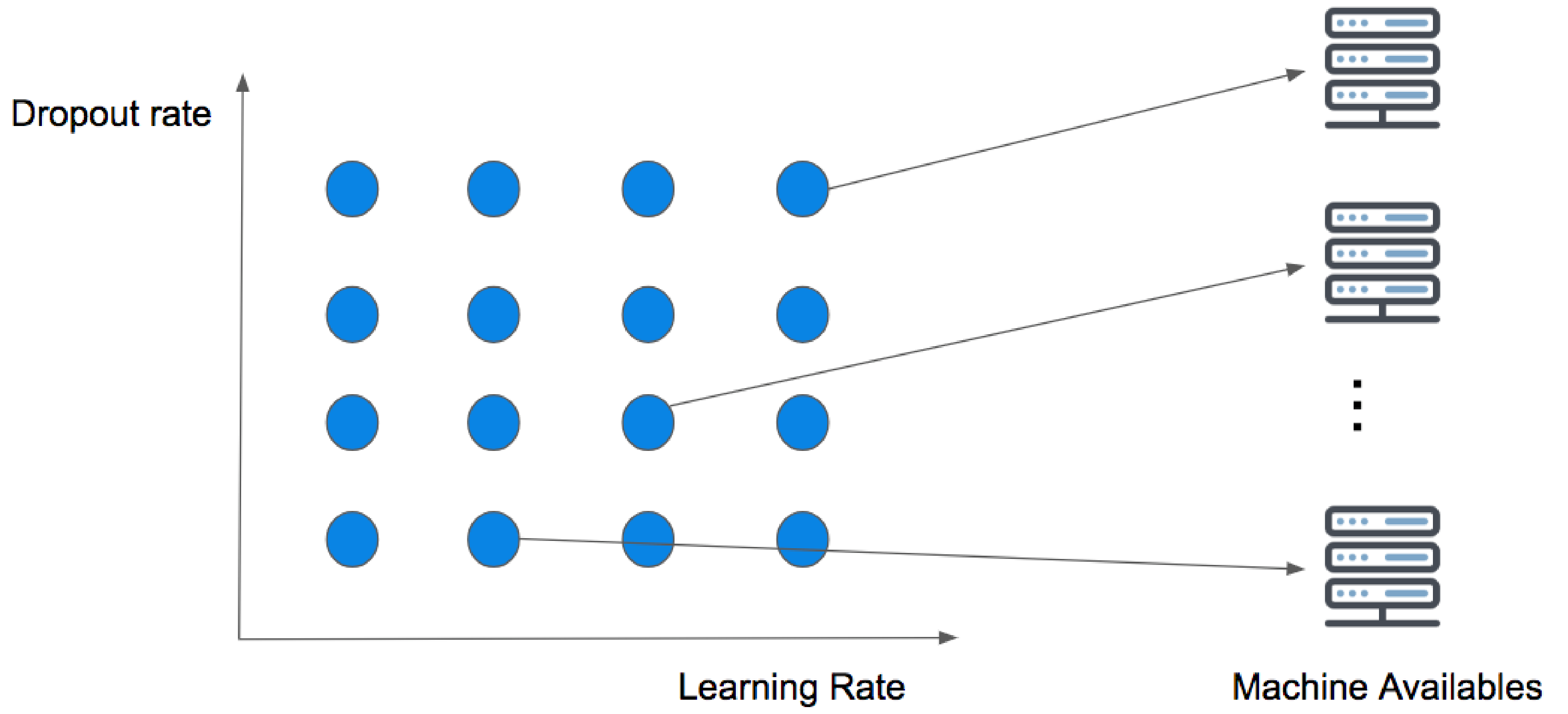


Space

Grid Search

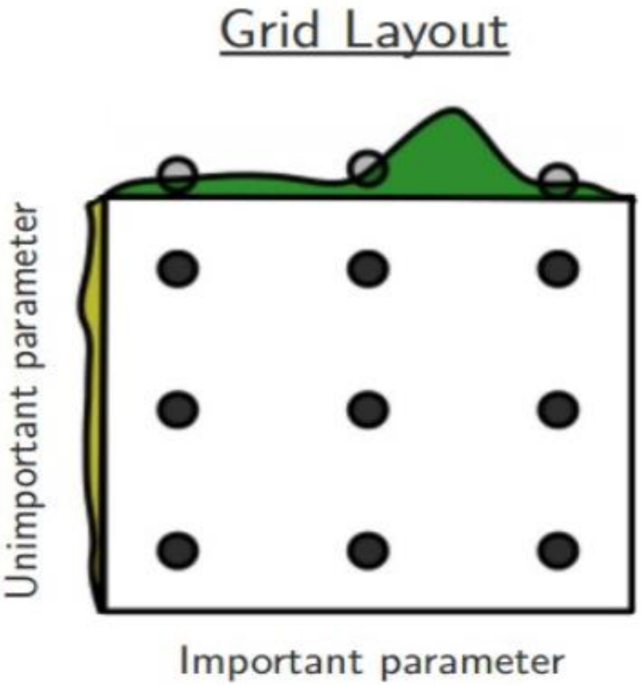
- Workflow:
  - Define a grid on *n* dimensions, where each of these maps is for an hyperparameter.
    - e.g. *n* = (learning_rate, dropout_rate, batch_size)
  - For each dimension, define the range of possible values:
    - e.g. batch_size = [4, 8, 16, 32, 64, 128, 256]
  - Search for all the possible configurations and wait for the results to establish the best one:
    - e.g. C1 = (0.1, 0.3, 4) -> acc = 92%, C2 = (0.1, 0.35, 4) -> acc = 92.3%, etc...

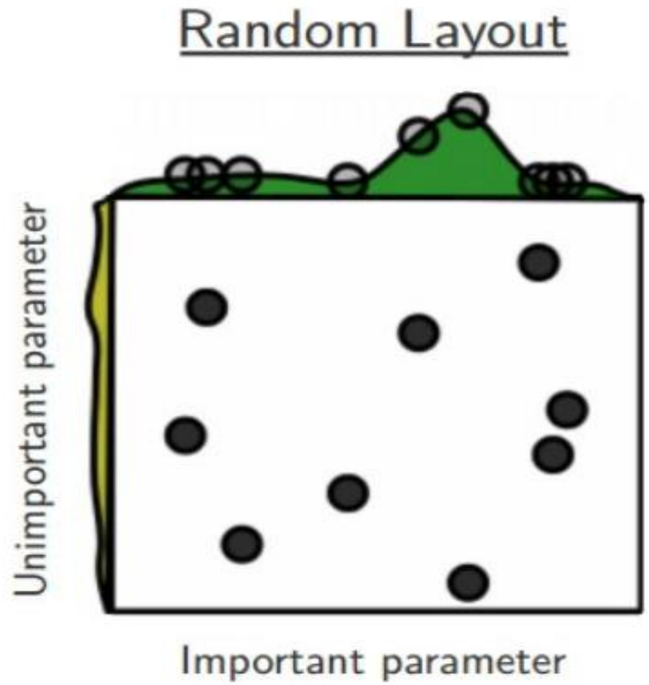
❖ A simple grid search on two dimensions for the Dropout and Learning rate



- This strategy is embarrassingly parallel because it doesn't take into account the computation history.
- Pain point: the curse of dimensionality
  - It's common to use this approach when the dimensions are less than or equal to 4

Grid Search vs Random Search





Bad on high spaces

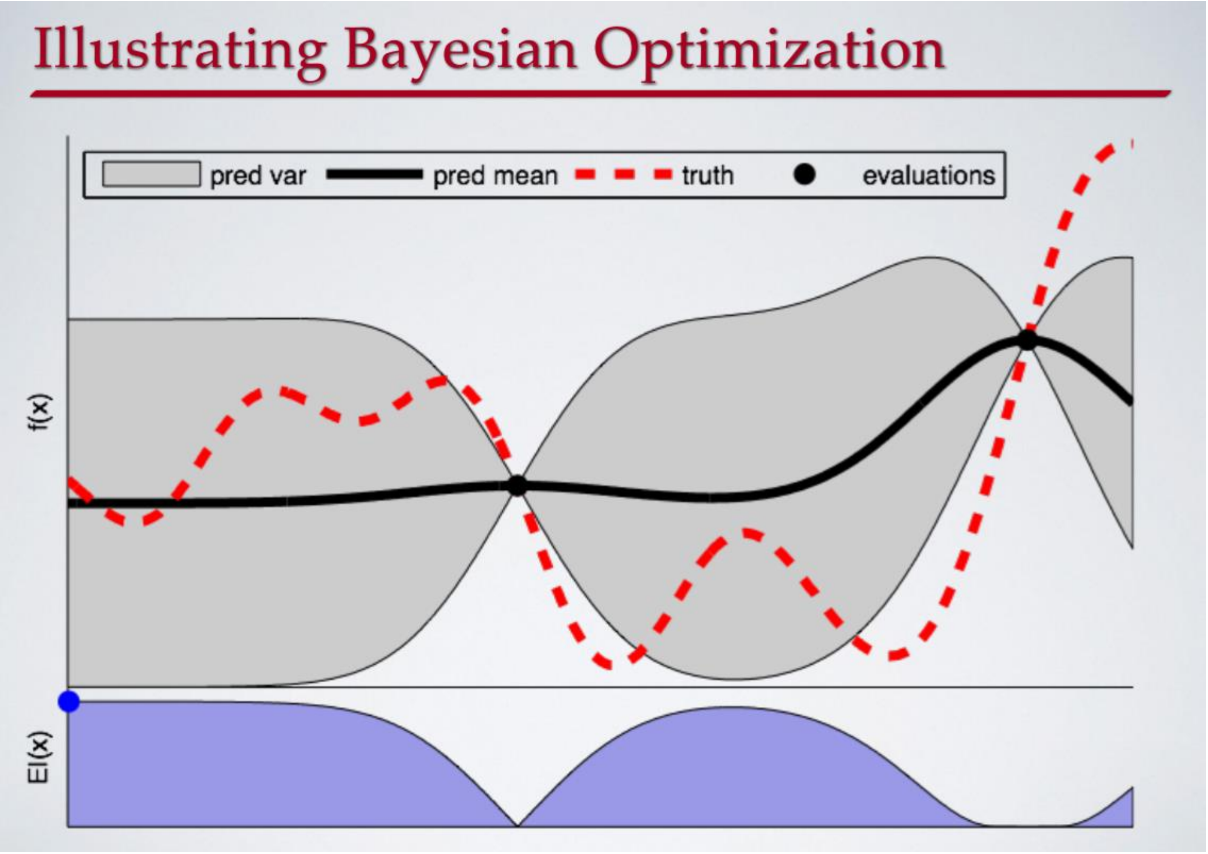
It doesn’t guarantee to find the best hyperparameters

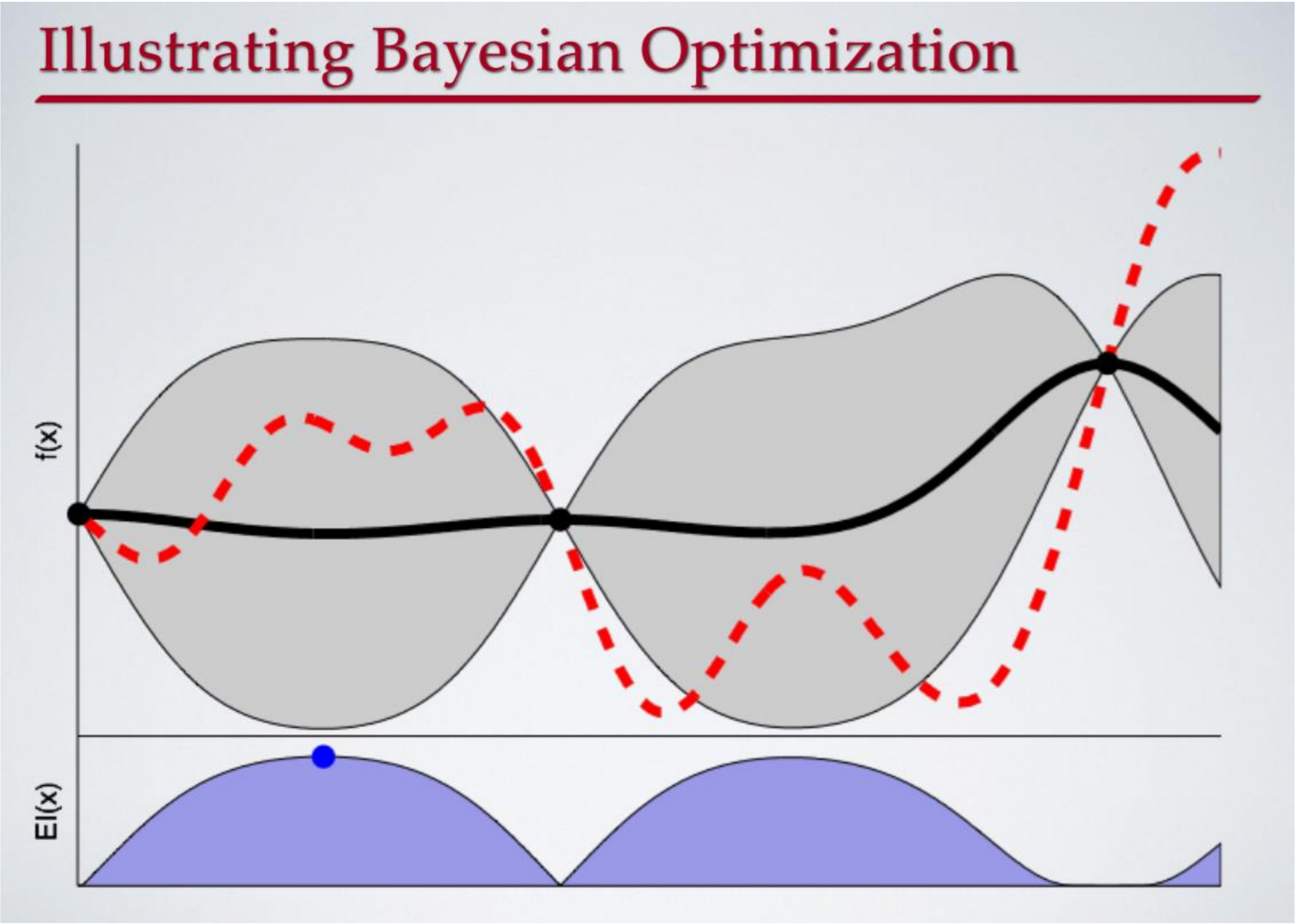
Good on high spaces

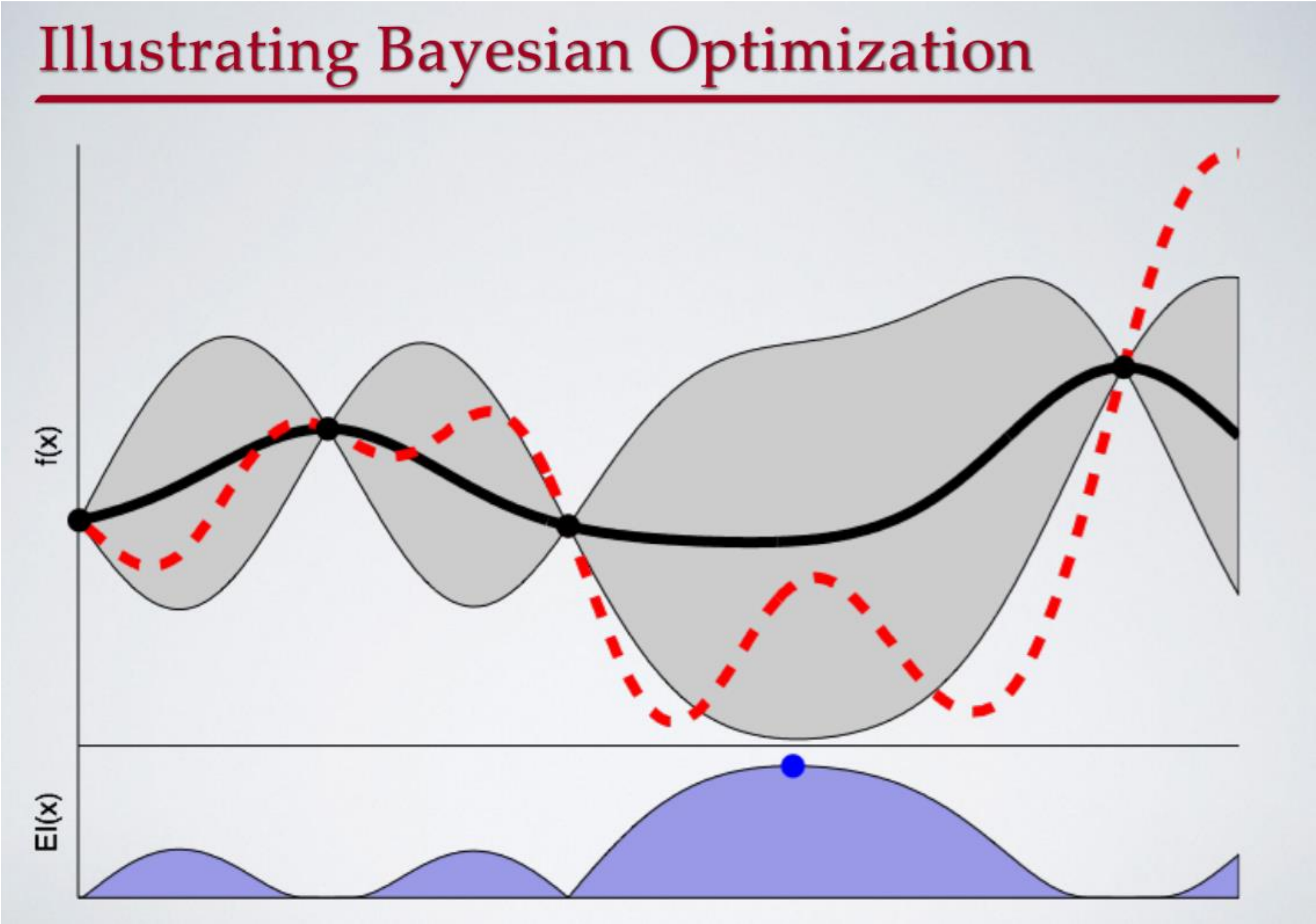
Give better results in less iterations

Bayesian Optimization

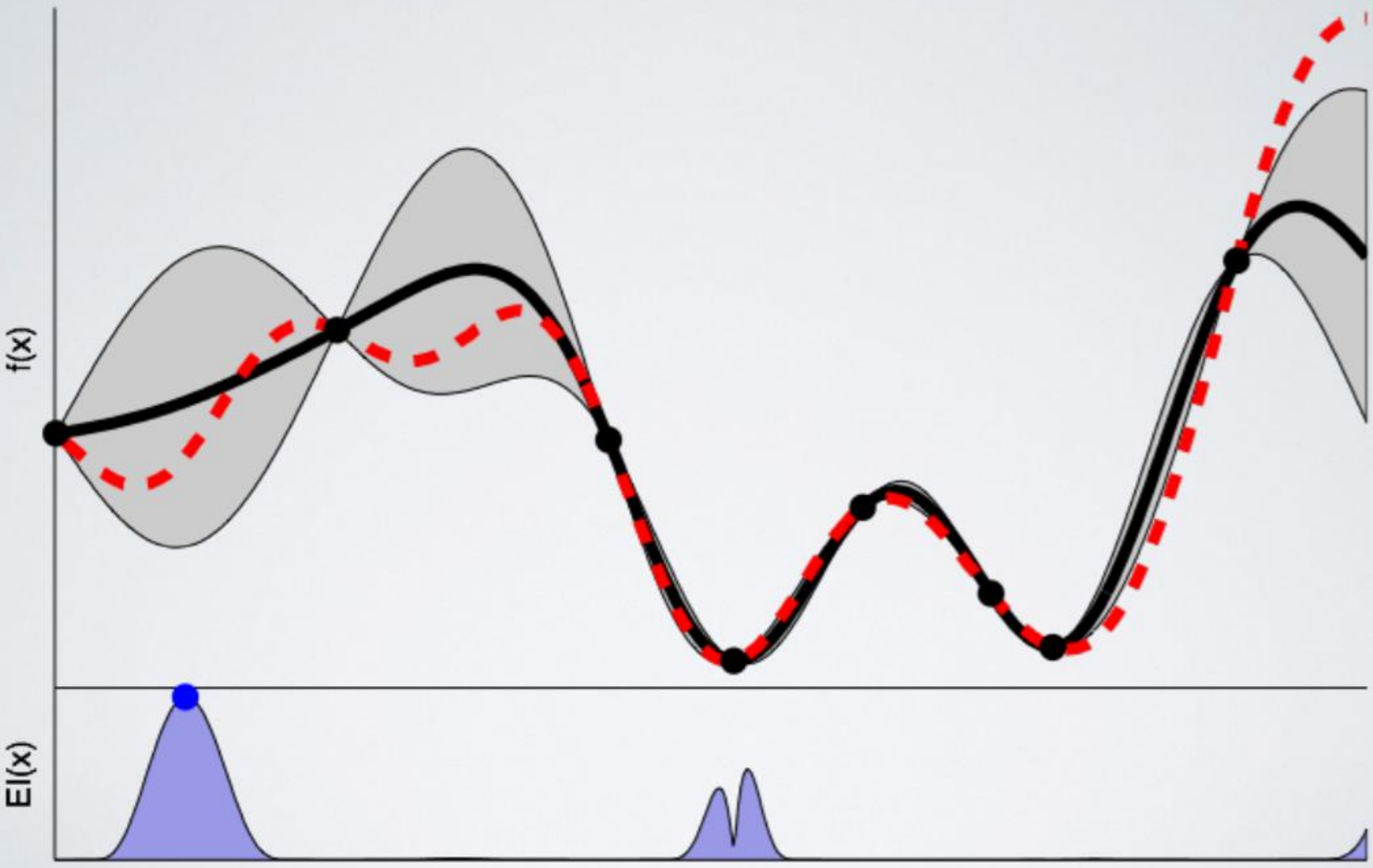
-Automatic Hyperparameter Tuning




Illustrating Bayesian Optimization
f(x)
EI(x)


Illustrating Bayesian Optimization
f(x)
EI(x)

Illustrating Bayesian Optimization



课程部分材料来自他人

和网络，仅限教学使用

，请勿传播，谢谢！