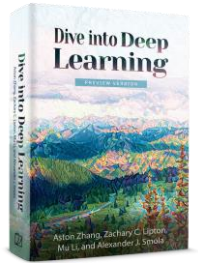
深度學習實作與應用
Deep learning and its applications

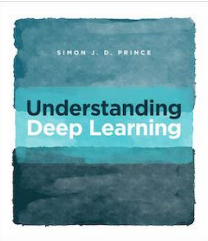
2. Basic Neural Networks (4)

IM5062, Spring 2024

黃意婷



Ch4 and CH11



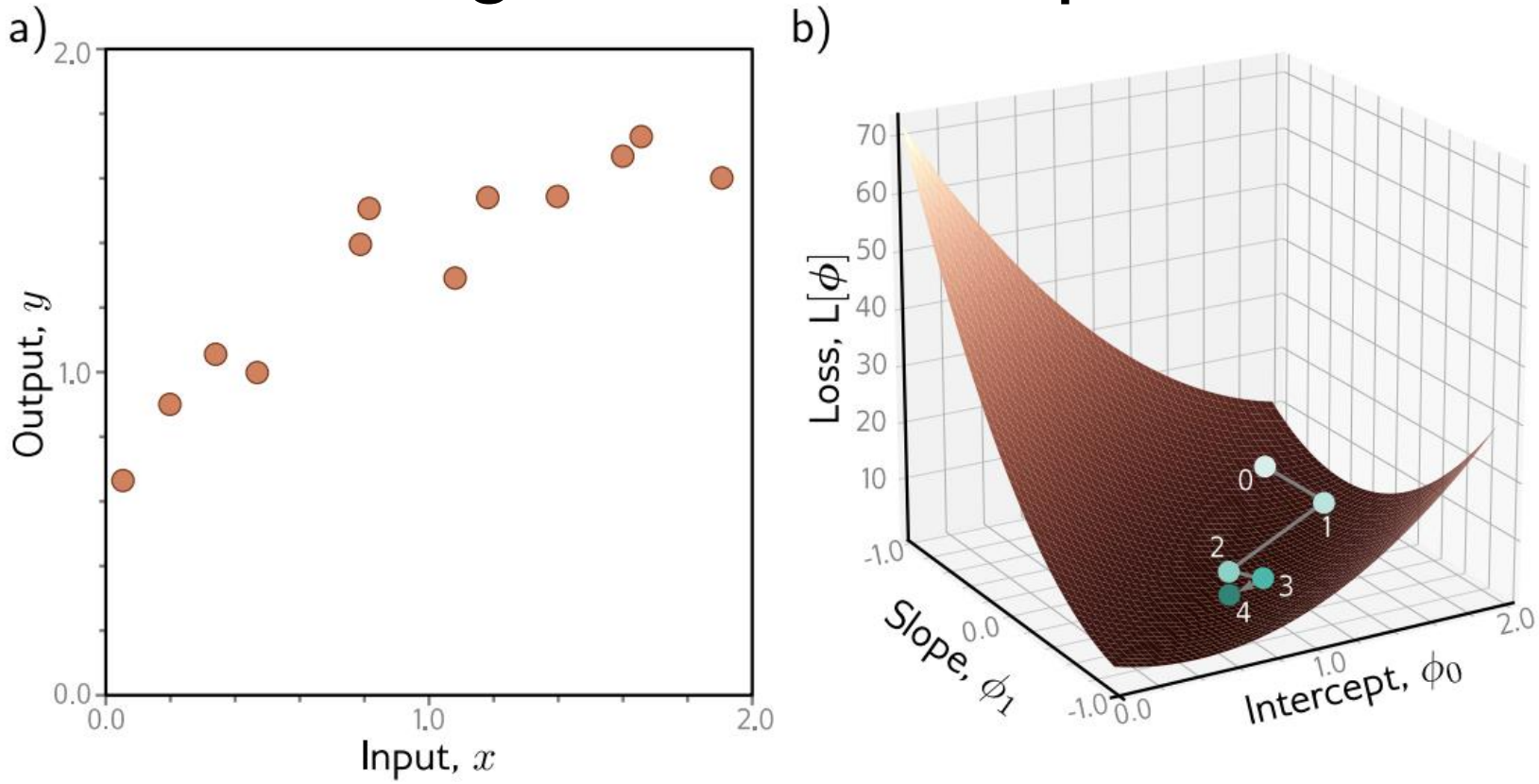
CH8

d2l: some slides from Alex Smola and Mu Li
UDL: some slides from Simon J.D. Prince

Outline

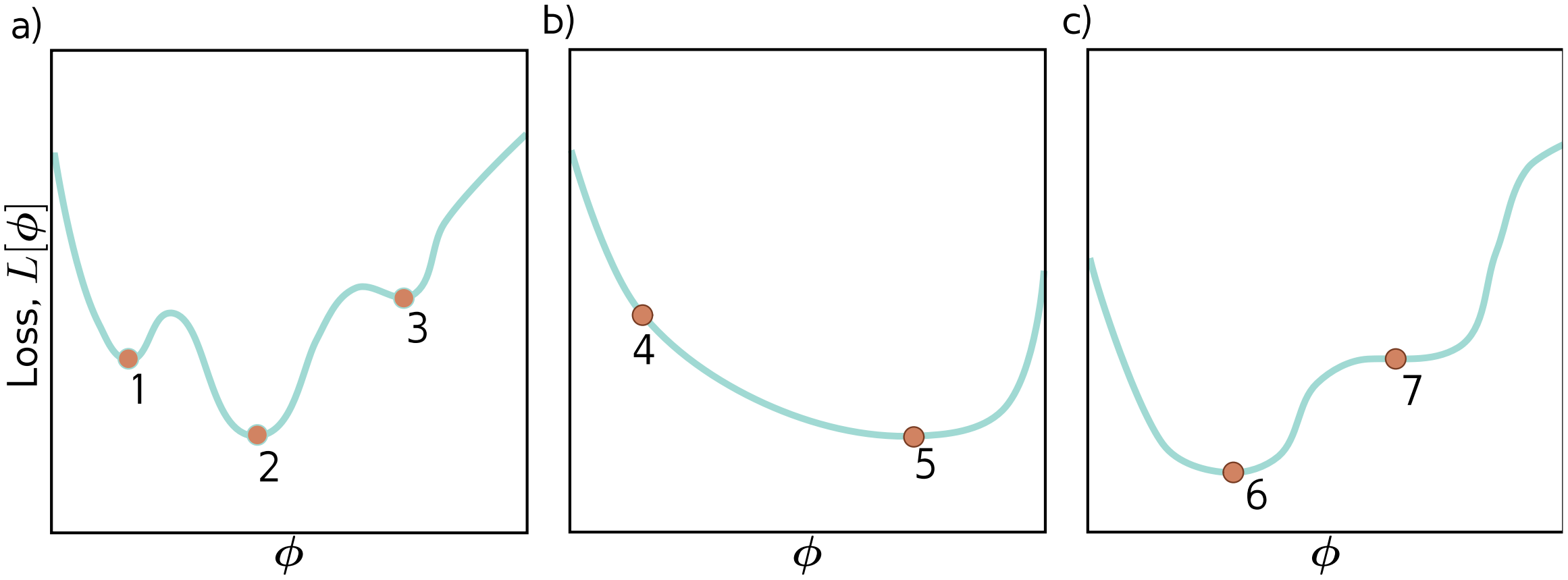
- Optimizer (d2l.ch11)
- Vanishing/Exploding Gradients (d2l.ch4.8)
- Parameter Initialization (d2l.ch4.8)
- Evaluation (d2l.ch4.4)
  - Dataset
  - Metrics
  - Overfitting or underfitting
- Regularization (d2l.ch4.5)
- Dropout (d2l.ch4.6)

Recall the linear regression example



a) Training set of *I* = 12 input/output pairs *{xi, yi}*.
b) Loss function showing iterations of gradient descent. We start at point 0 and move in the steepest downhill direction until we can improve no further to arrive at point 1. We then repeat this procedure. We measure the gradient at point 1 and move downhill to point 2 and so on.

Convex problems



Non convex

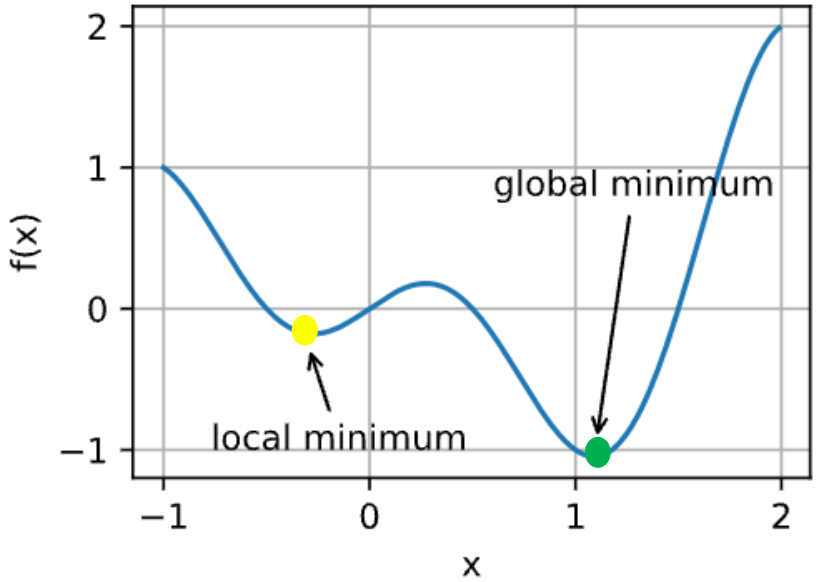
Convex

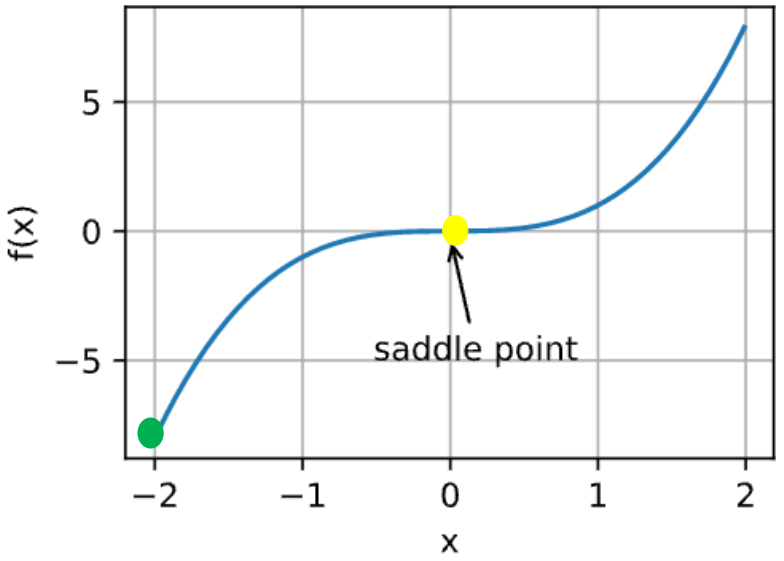
Non-Convex

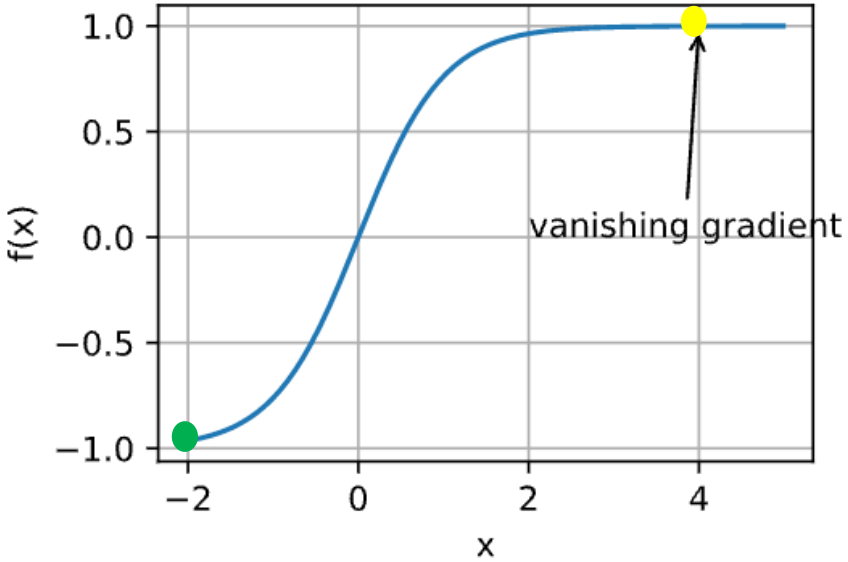
Optimization Challenges in Deep Learning

- Local minima
- Saddle Points
- Vanishing Gradients

output, i.e. loss value





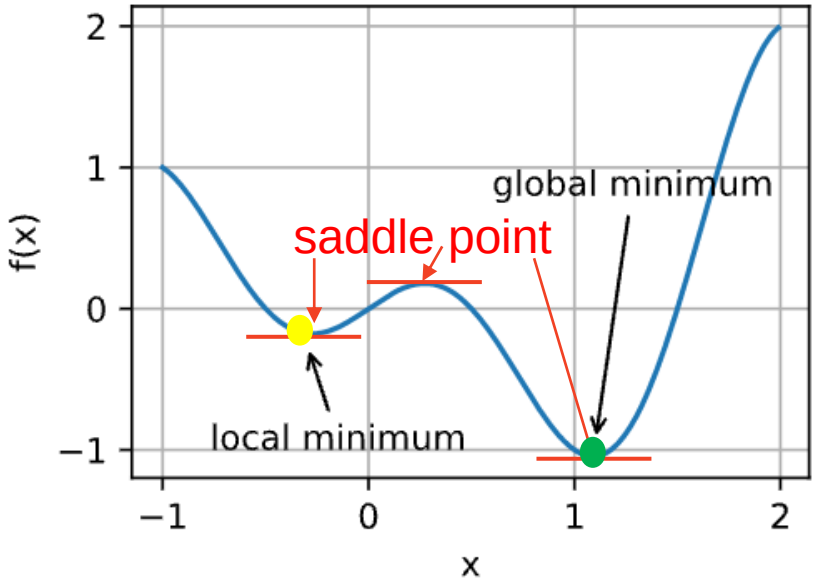


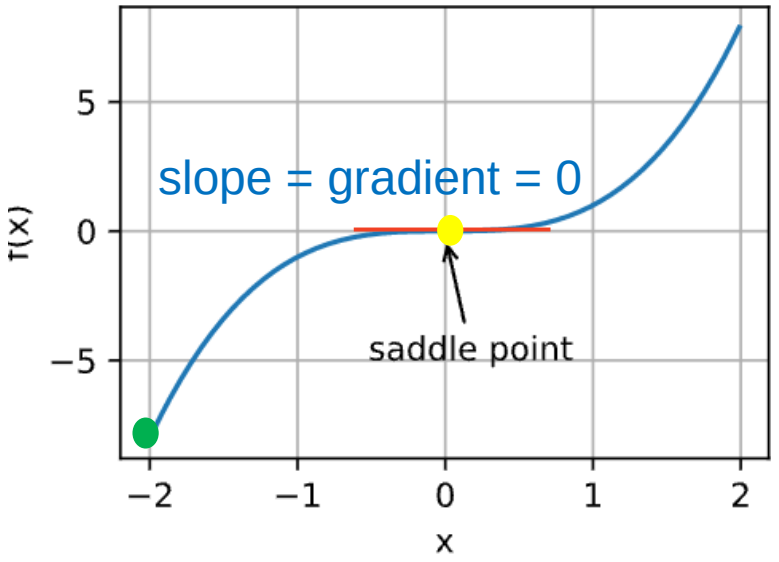
input, i.e., network prediction

Optimization Challenges in Deep Learning

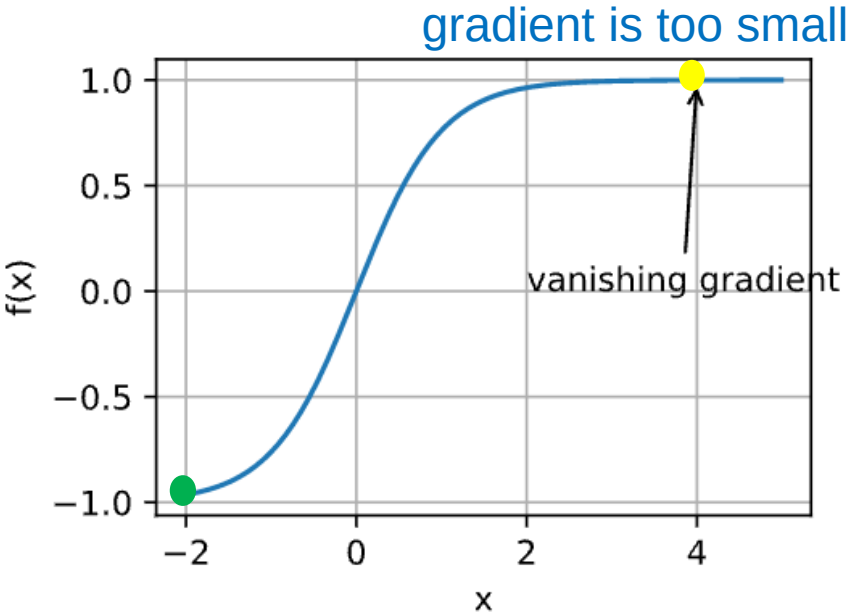
- Local minima
- Saddle Points
- Vanishing Gradients

output, i.e. loss value





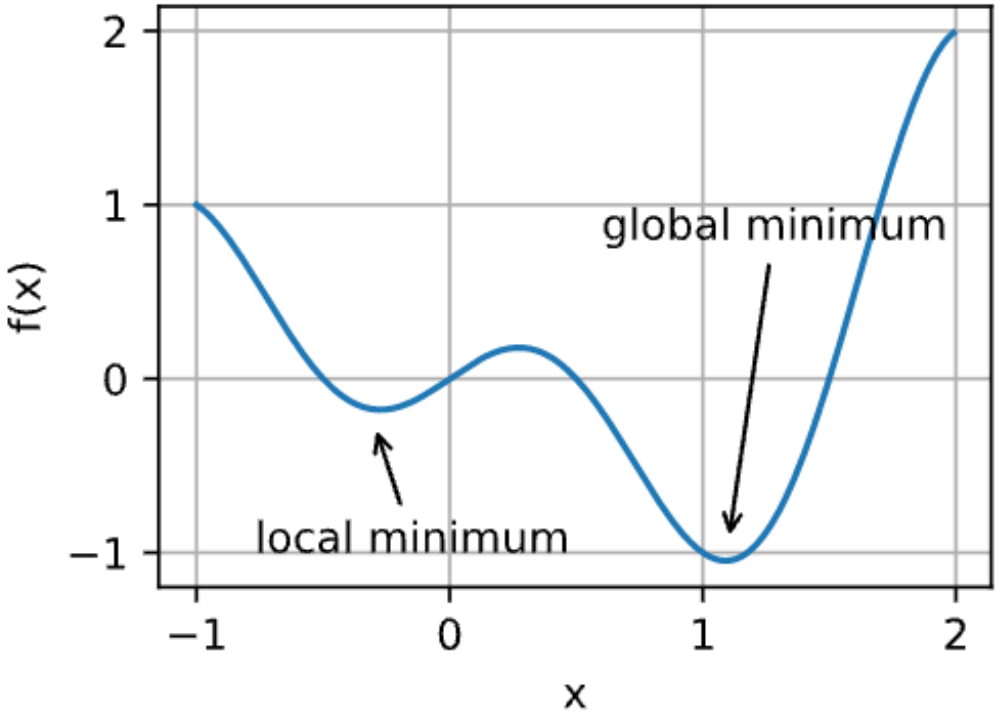
gradient is too small



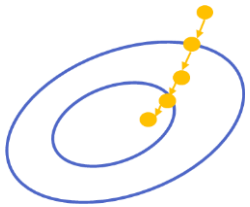
input, i.e., network prediction

Some issues

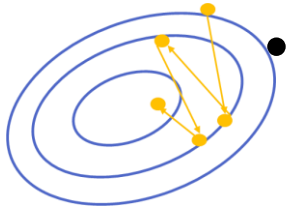
The Gradient Descent process can exhibit significant variability depending on the randomly selected starting point.



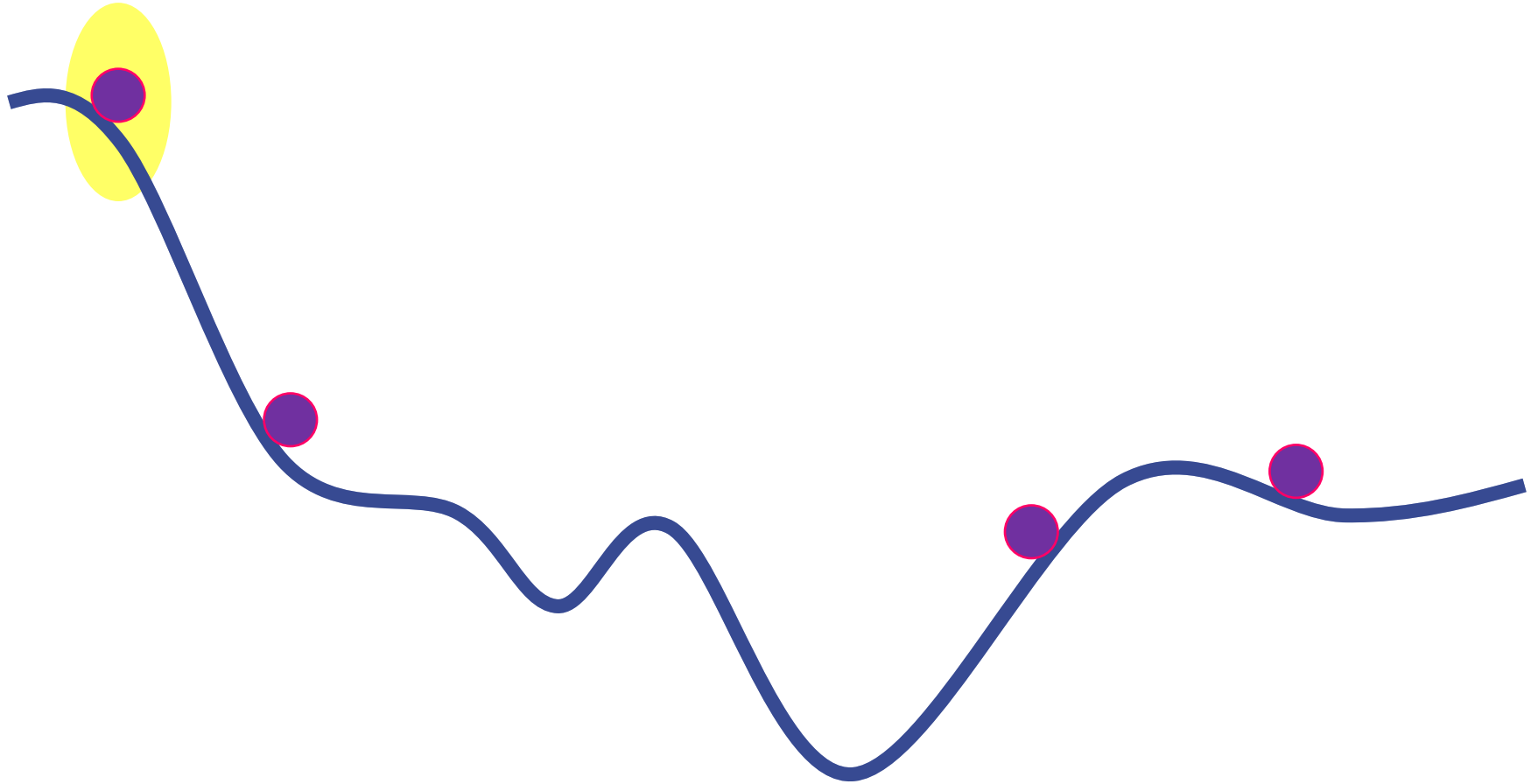
- Sensitive to learning rate
  - η is too small,
    - no progress
    - local minima

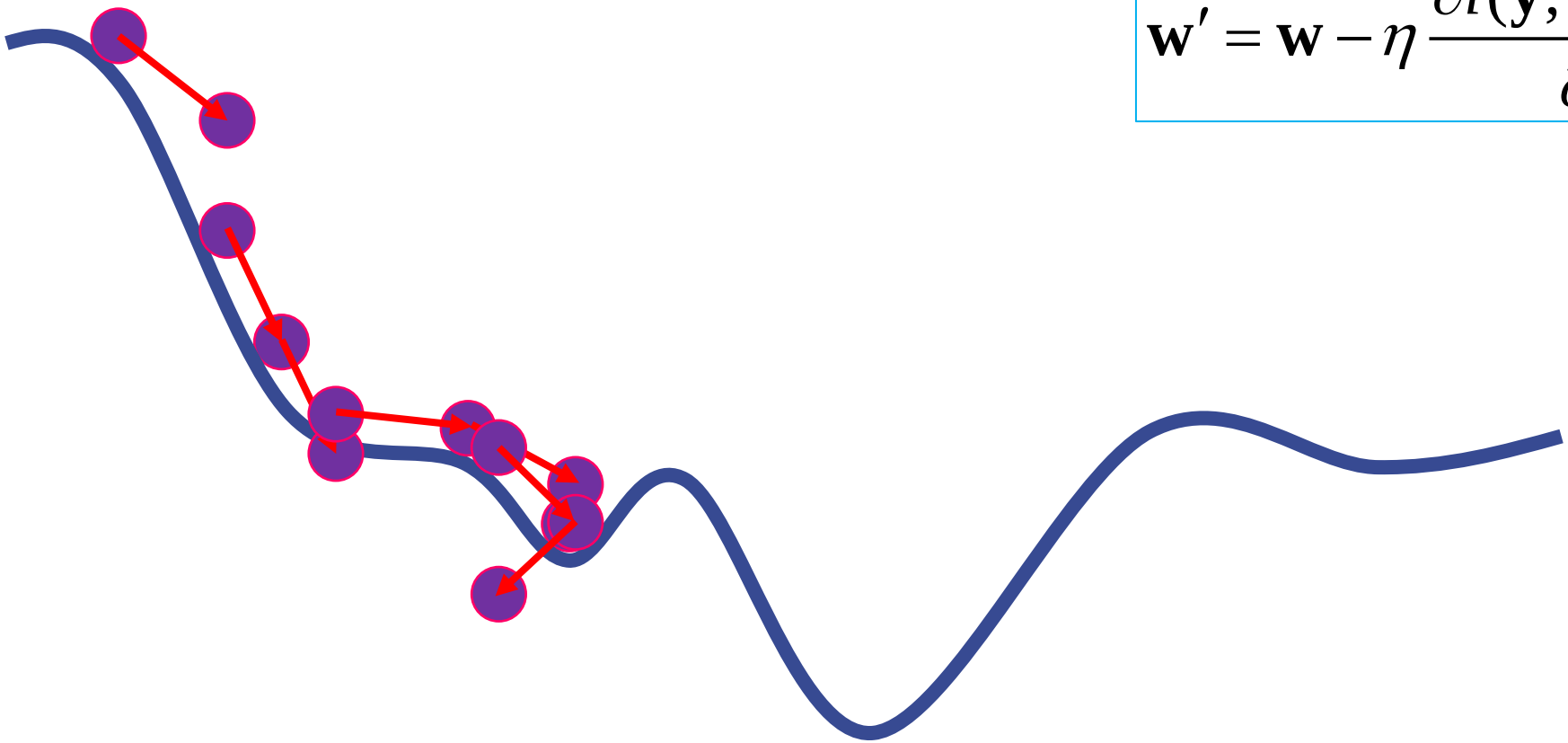


  - η is too large, unstable



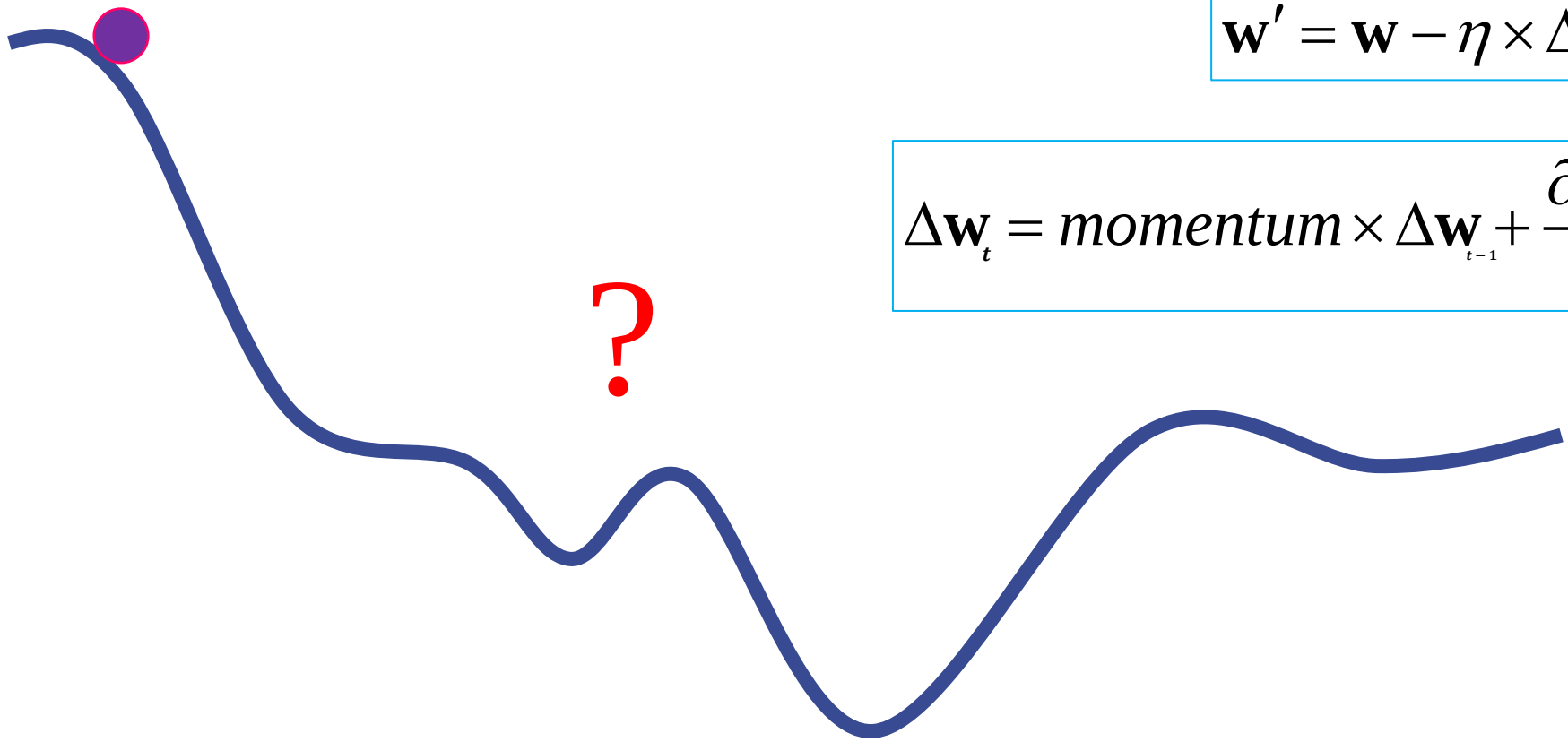
  - May need to change learning rate dramatically during training





$$\mathbf{w}' = \mathbf{w} - \eta \frac{\partial l(\mathbf{y}, \mathbf{wx} + \mathbf{b})}{\partial \mathbf{w}}$$

momentum

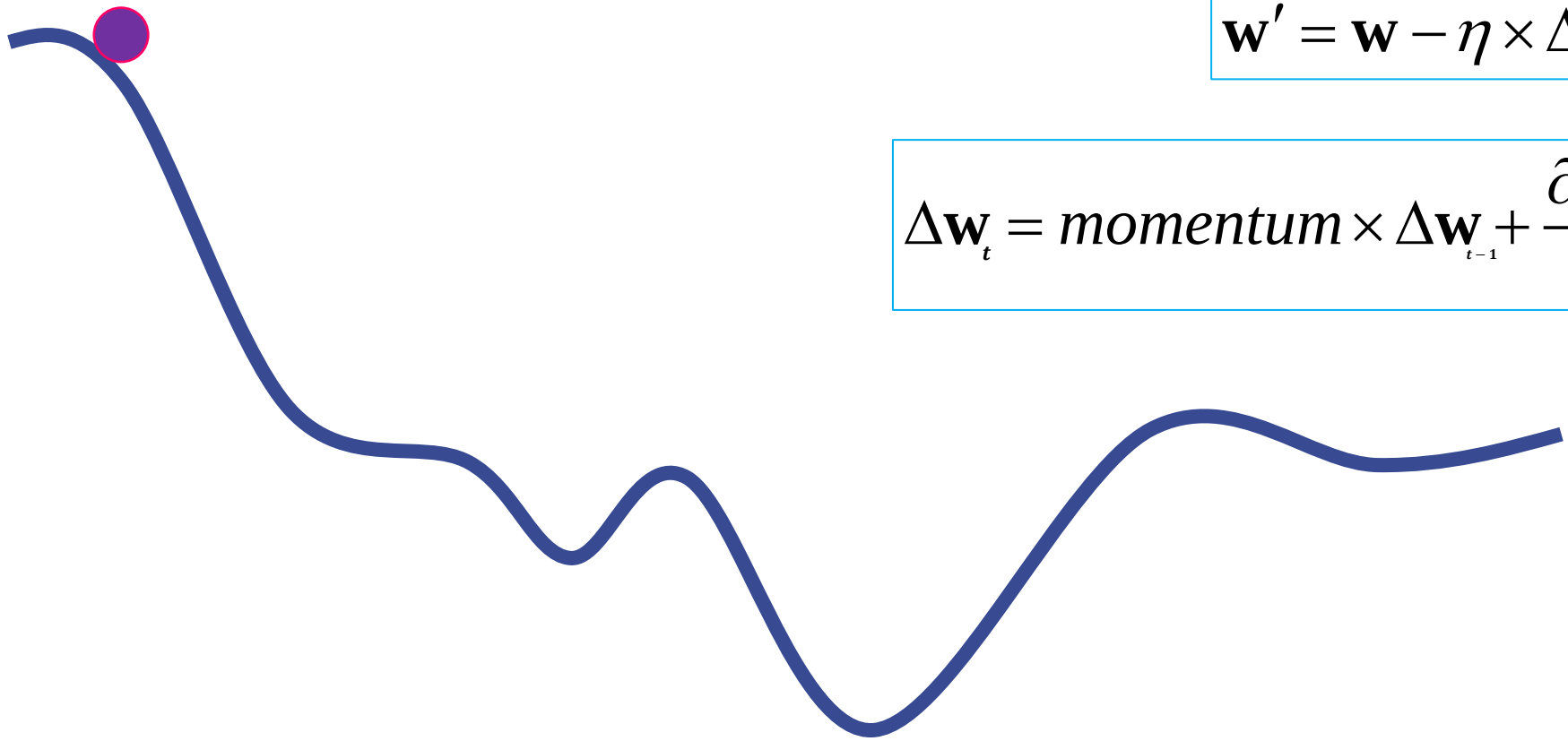


$$\mathbf{w}' = \mathbf{w} - \eta \times \Delta\mathbf{w}$$

$$\Delta\mathbf{w}_t = momentum \times \Delta\mathbf{w}_{t-1} + \frac{\partial l(\mathbf{y}, \mathbf{wx} + \mathbf{b})}{\partial \mathbf{w}}$$

momentum

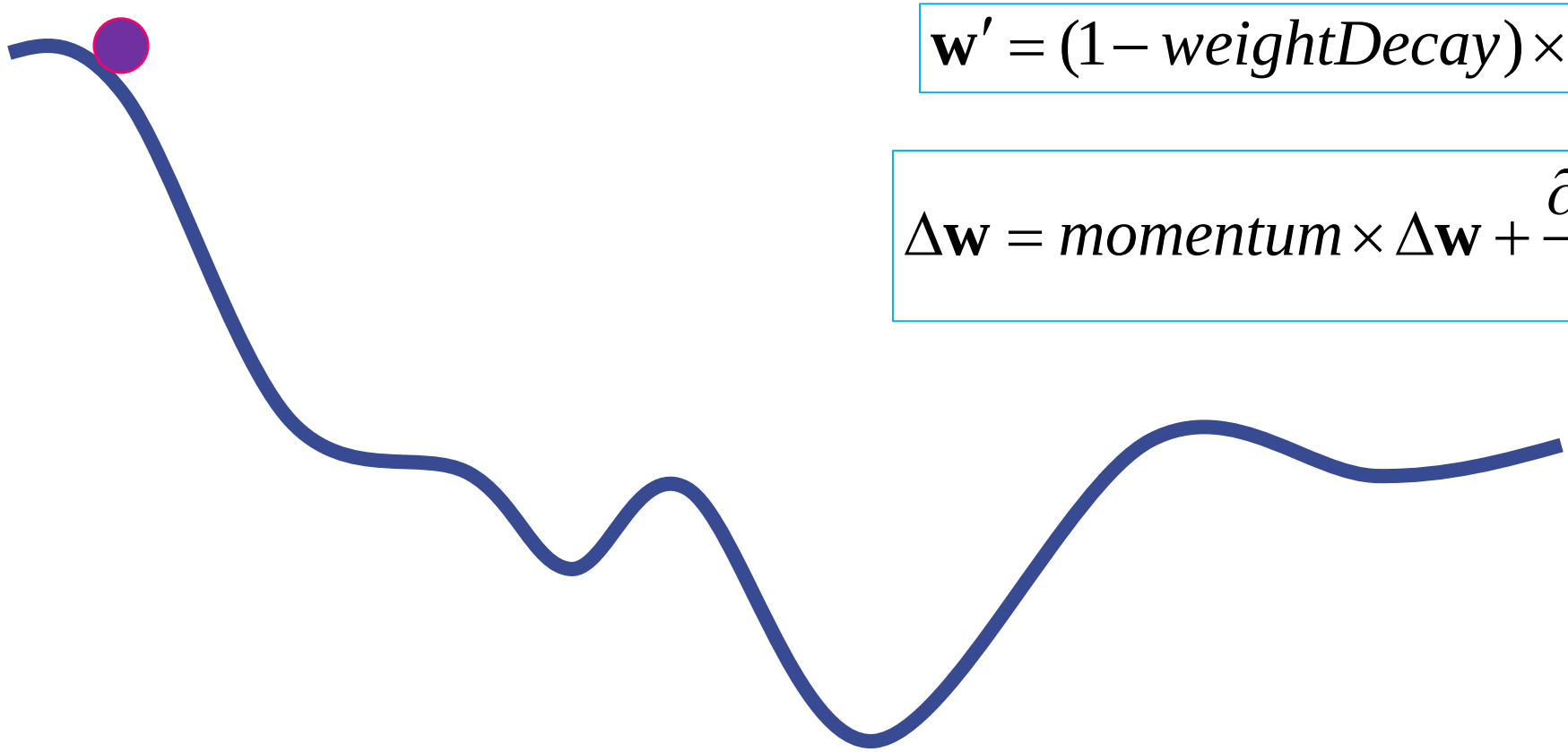
$$\Delta w_0 = 0$$
$$\Delta w_1 = \Delta w_0 + g_0 = 0 + g_0$$
$$\Delta w_2 = \Delta w_1 + g_1 = 0 + g_0 + g_1$$
$$\dots$$
$$\Delta w_{t-1} = 0 + g_0 + g_1 + \cdots + g_{t-2}$$



$$\mathbf{w}' = \mathbf{w} - \eta \times \Delta \mathbf{w}$$

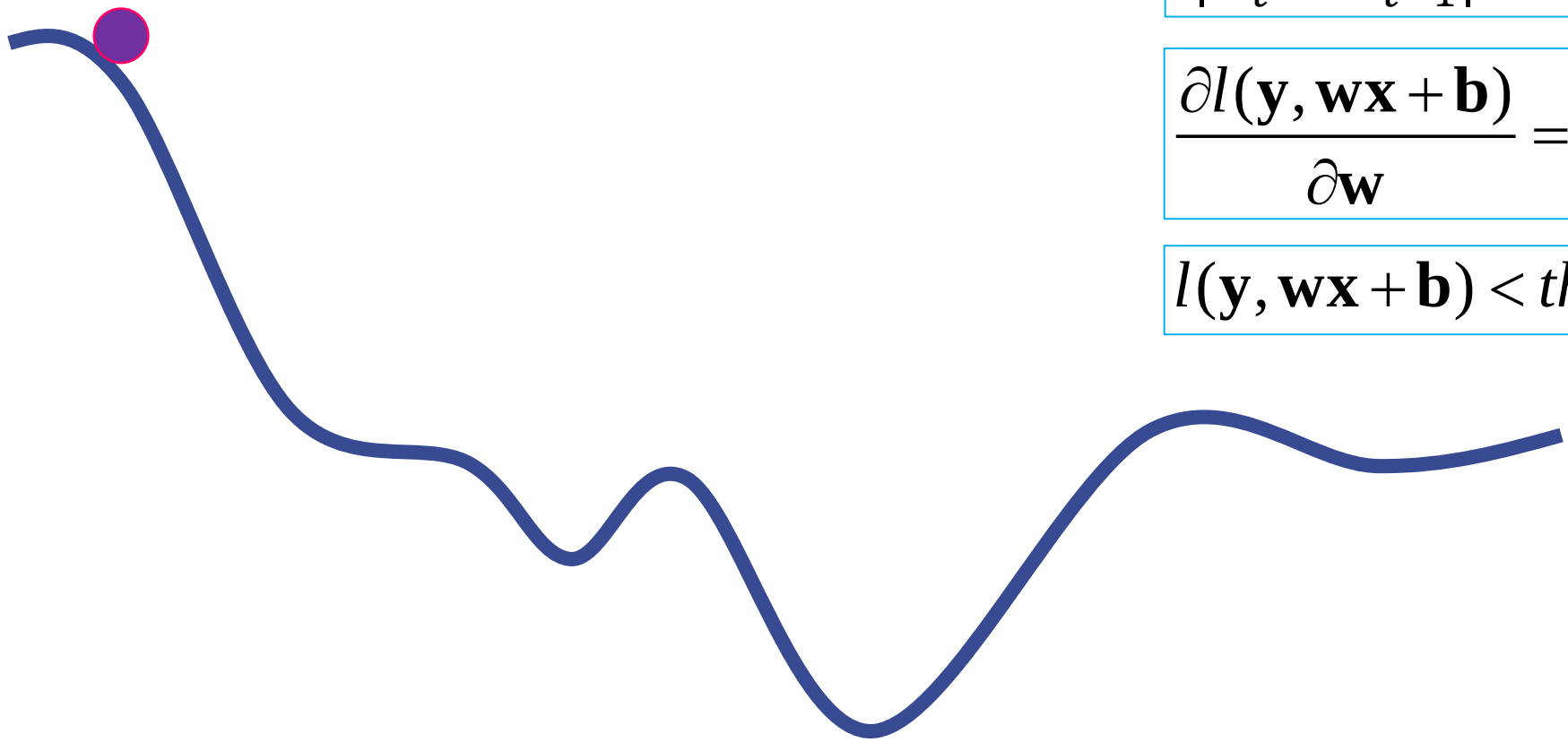
$$\Delta \mathbf{w}_t = momentum \times \Delta \mathbf{w}_{t-1} + \frac{\partial l(\mathbf{y}, \mathbf{wx} + \mathbf{b})}{\partial \mathbf{w}}$$

weight decay



$$\mathbf{w}' = (1 - weightDecay) \times \mathbf{w} - \eta \times \Delta\mathbf{w}$$

$$\Delta\mathbf{w} = momentum \times \Delta\mathbf{w} + \frac{\partial l(\mathbf{y}, \mathbf{wx} + \mathbf{b})}{\partial \mathbf{w}}$$

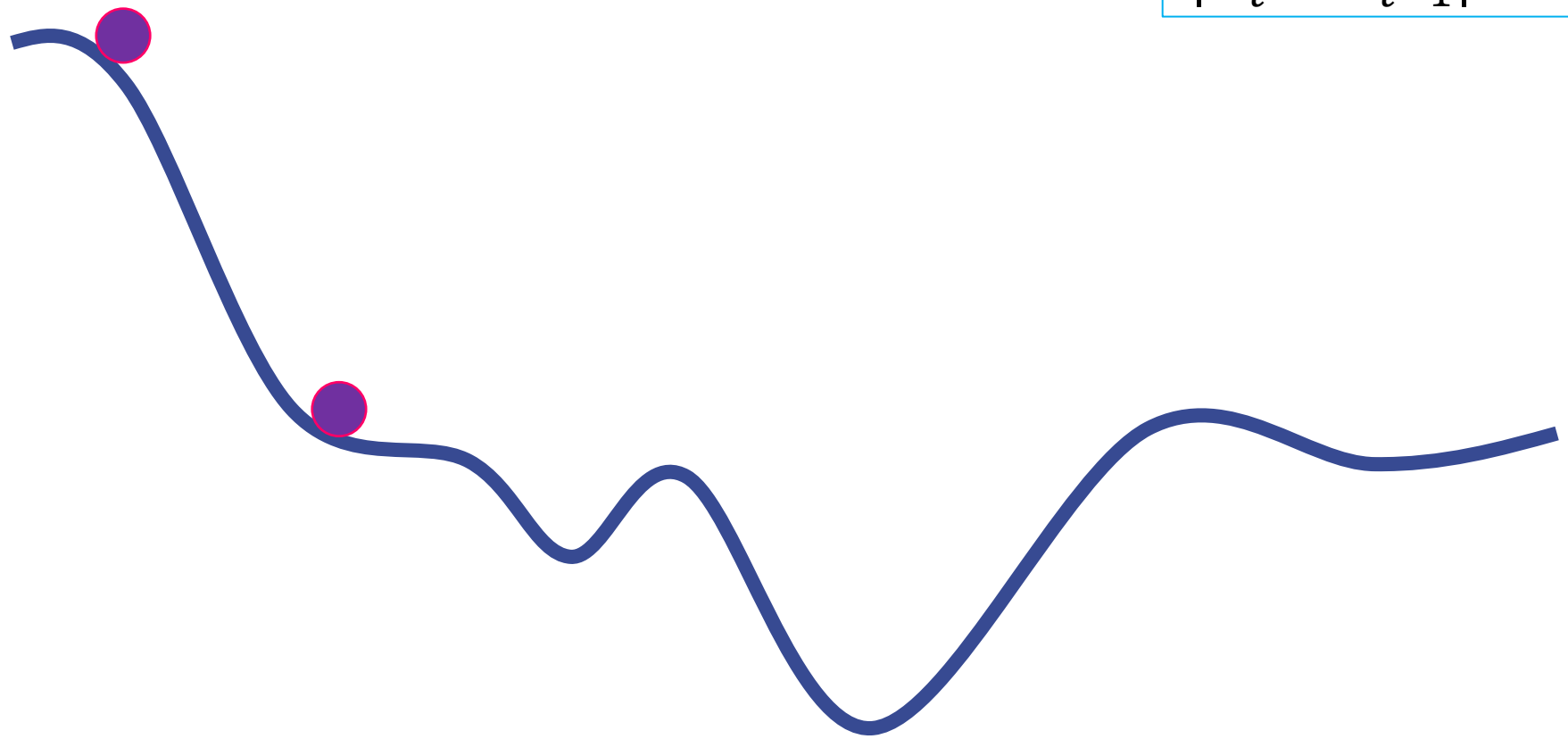


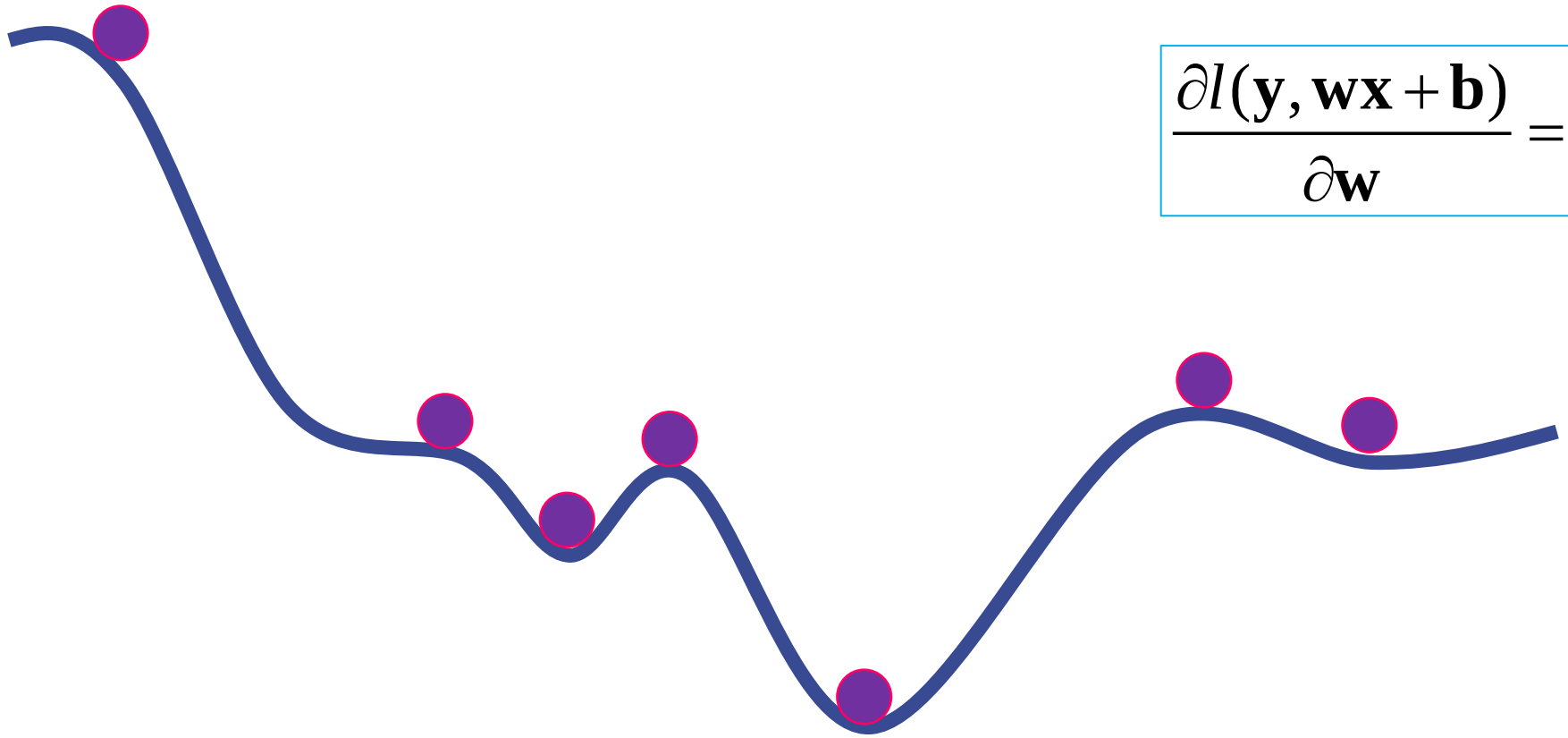
$$|w_t - w_{t-1}| \approx 0$$

$$\frac{\partial l(\mathbf{y}, \mathbf{wx}+\mathbf{b})}{\partial \mathbf{w}} = 0$$

$$l(\mathbf{y}, \mathbf{wx}+\mathbf{b}) < threshold$$

$$|w_t - w_{t-1}| \approx 0$$

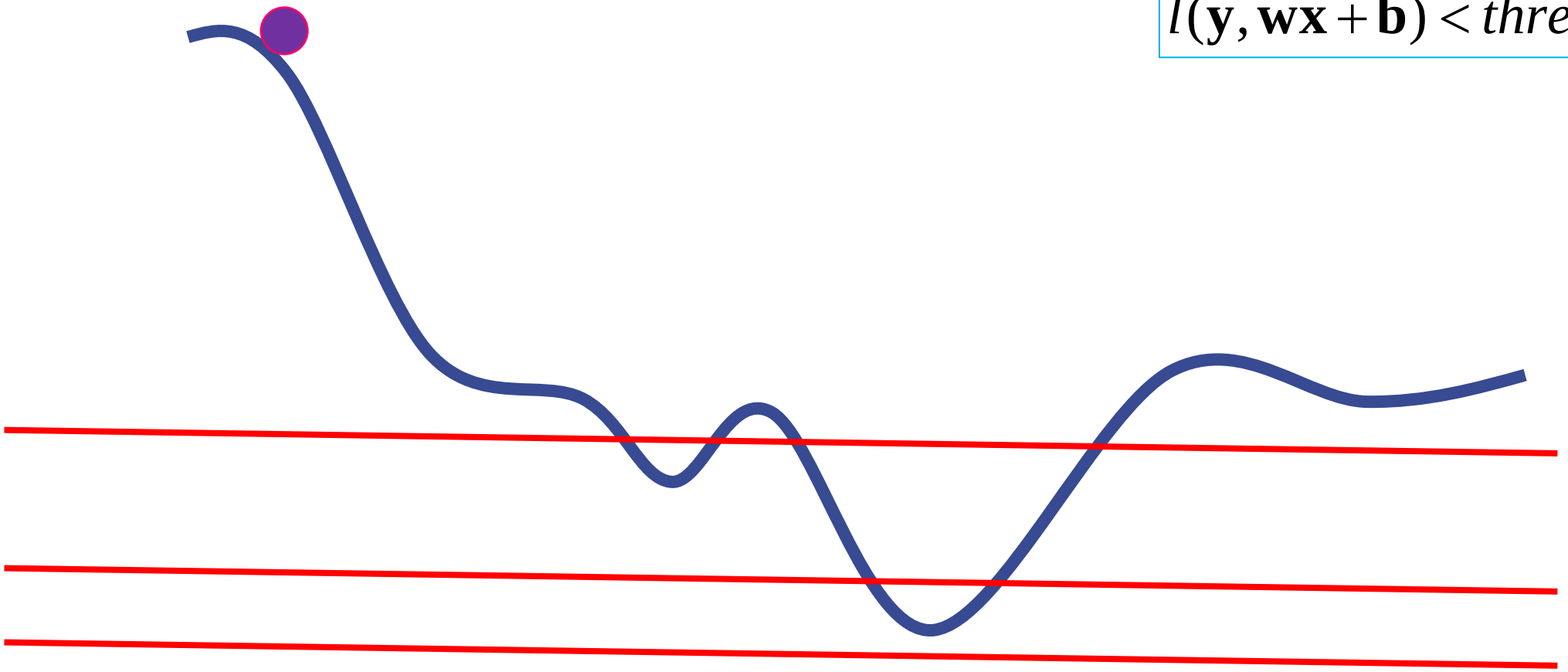


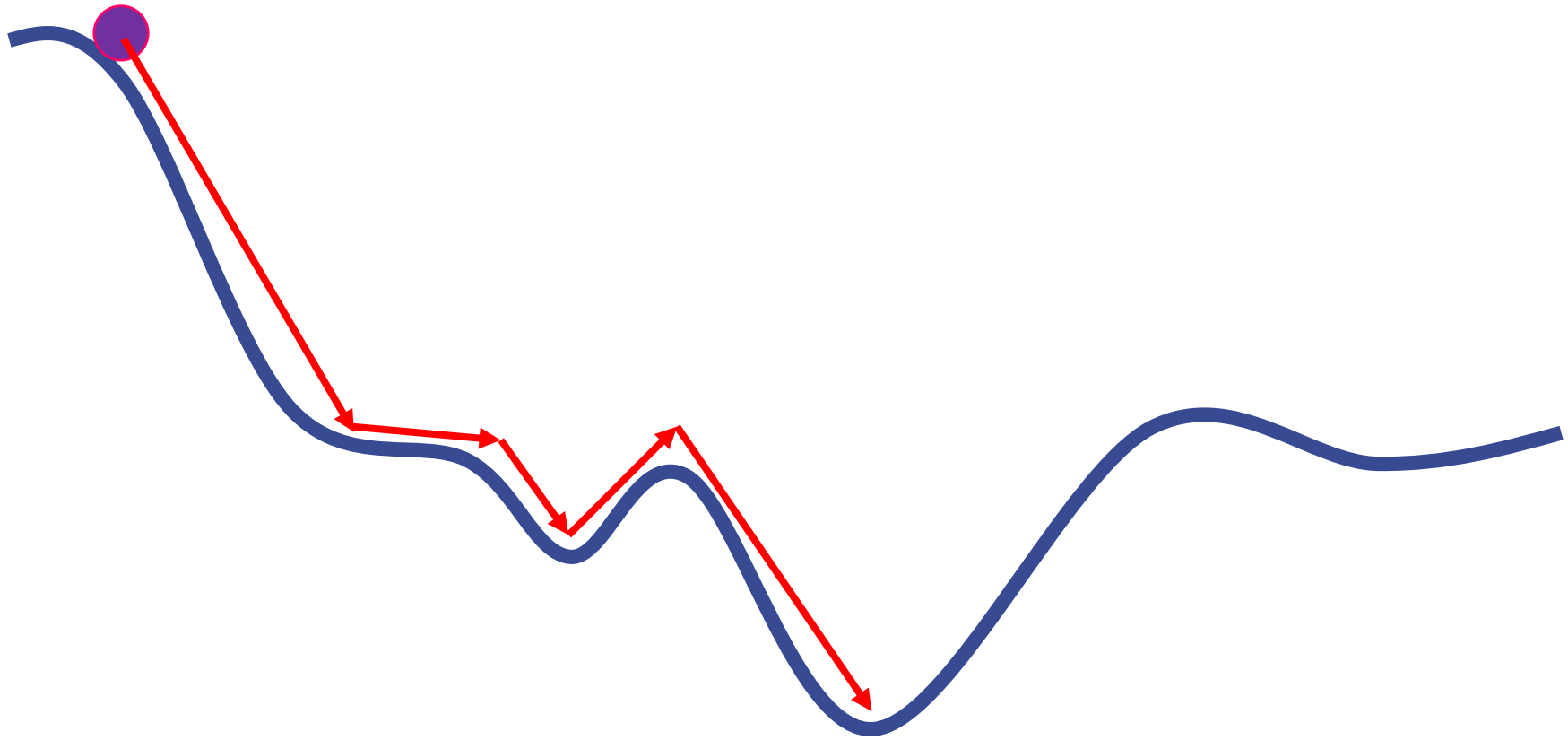


$$\frac{\partial l(\mathbf{y}, \mathbf{wx}+\mathbf{b})}{\partial \mathbf{w}} = 0$$

threshold

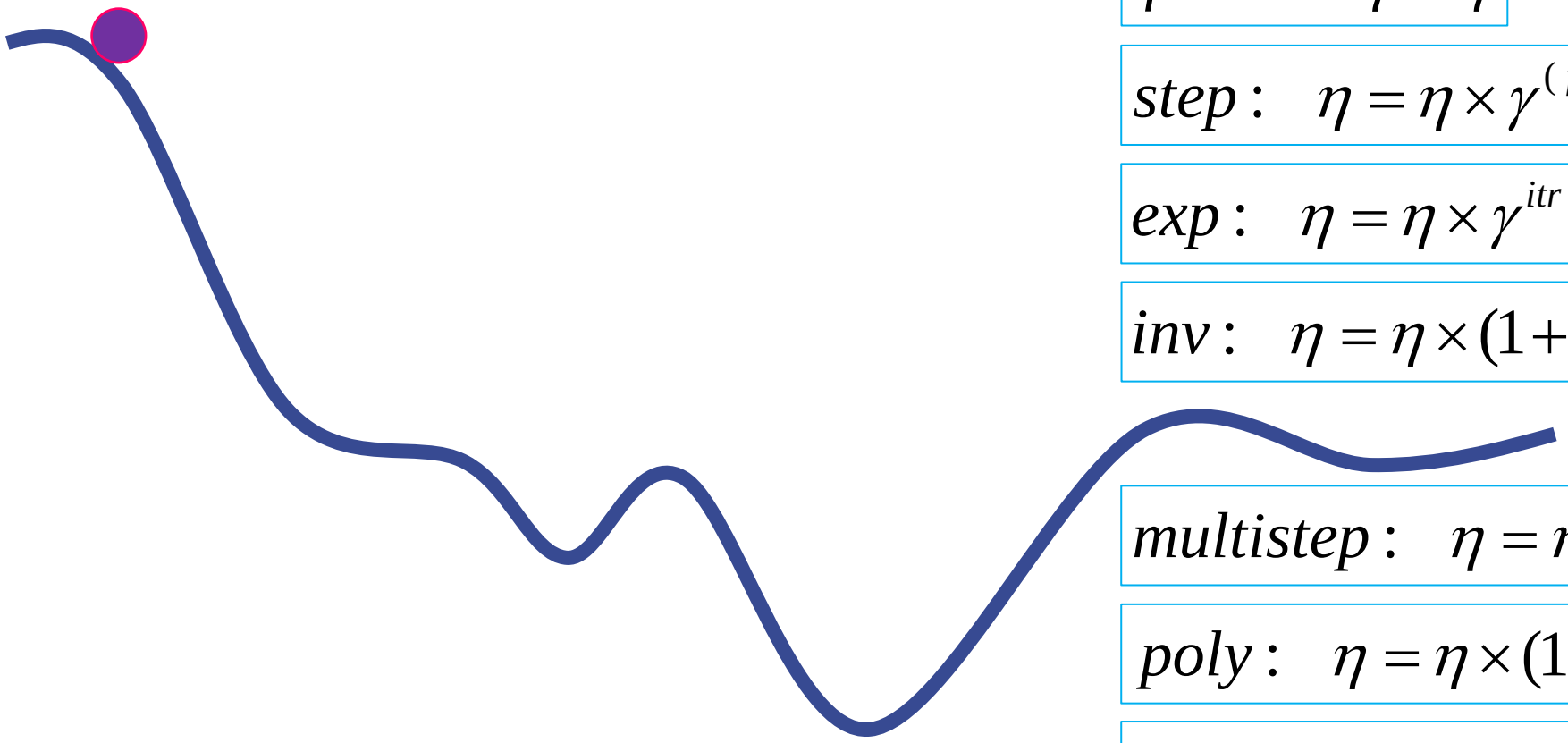
$$l(\mathbf{y}, \mathbf{wx} + \mathbf{b}) < threshold$$





Learning rate decay policy

$$\mathbf{w}' = \mathbf{w} - \eta \frac{\partial l(\mathbf{y}, \mathbf{wx} + \mathbf{b})}{\partial \mathbf{w}}$$



$$fixed: \quad \eta = \eta$$

$$step: \quad \eta = \eta \times \gamma^{(floor(itr/step))}$$

$$exp: \quad \eta = \eta \times \gamma^{itr}$$

$$inv: \quad \eta = \eta \times (1 + \gamma \times itr)^{-power}$$

$$multistep: \quad \eta = \eta \times \gamma^{(floor(itr/step))}$$

$$poly: \quad \eta = \eta \times (1 - itr/maxitr)^{power}$$

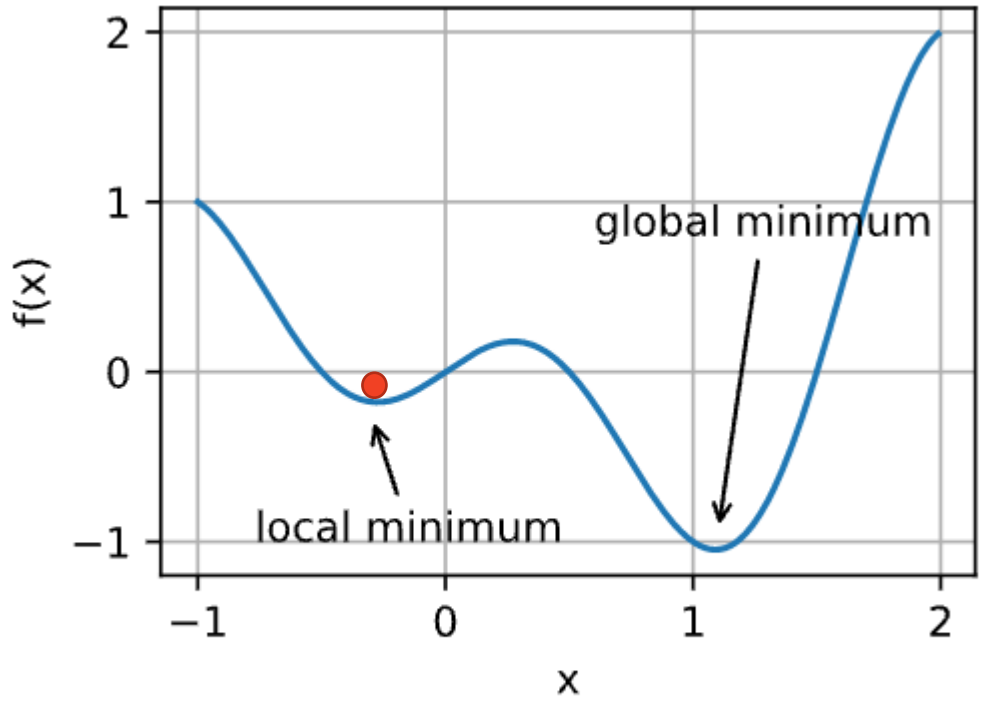
$$sigmoid: \quad \eta = \eta \times (1/(1 + e^{(-\gamma \times (itr - stepsize))}))$$

Optimizer

- Momentum
- Adaptive Gradient (AdaGrad)
- RMSprop (RMSProp)
- Adam (Adam)

Momentum

- local minima issue.



$$\mathbf{w}' = \mathbf{w} - \eta \times \Delta\mathbf{w}$$

$$\Delta\mathbf{w} = momentum \times \Delta\mathbf{w} + \frac{\partial l(\mathbf{y}, \mathbf{wx} + \mathbf{b})}{\partial \mathbf{w}}$$

$$v_t = \alpha v_{t-1} - \eta_t \mathrm{g_t}$$

Last step

gradient at present

Common values of α used in practice include 0.5, 0.9, and 0.99.

$$v_0 = 0, 0 \leq \alpha \leq 1$$

$$\theta_t \leftarrow \theta_{t-1} + v_t$$

AdaGrad

- Learning rates η matter → Algorithms with Adaptive Learning Rates
- One idea: $\eta^t = \frac{\eta}{\sqrt{t+1}}$, another idea: $\eta^t = \frac{\eta}{\sqrt{\text{acc. gradient}}}$
- AdaGrad $w_t = w_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot g_t$

Accumulating squared gradients

$$s_t = s_{t-1} + g_t^2$$

- An additive constant that ensures that we do not divide by 0.
- It is typically set to 10^{-6}

RMSprop

- Adagrad decays the learning rate that may lead slow progress at end.

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot g_t$$

$$s_t = \gamma s_{t-1} + (1-\gamma) g_t^2$$

$0 \leq \gamma < 1$

- An additive constant that ensures that we do not divide by 0.
- It is typically set to 10^{-6}

Adam (is mostly used) = RMSProp + Momentum

Similar to Momentum $v_t \leftarrow \beta_1 v_{t-1} + (1-\beta_1) g_t$

Similar to RMSProp $s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t^2$

- Common choices for them are $\beta_1 = 0.9, \beta_2 = 0.99$.
- That is, the variance estimate moves *much more slowly* than the momentum term.

- Fix bias $\hat{v}_t = \frac{v_t}{1-\beta_1^t}$ $\hat{s}_t = \frac{s_t}{1-\beta_2^t}$ $w_t = w_{t-1} - \frac{\eta}{\sqrt{\hat{s}_t + \varepsilon}} \hat{v}_t$

Outline

- Optimizer (d2l.ch11)
- Vanishing/Exploding Gradients (d2l.ch4.8)
- Parameter Initialization (d2l.ch4.8)
- Evaluation (d2l.ch4.4)
  - Dataset
  - Metrics
  - Overfitting or underfitting
- Regularization (d2l.ch4.5)
- Dropout (d2l.ch4.6)

Gradients for Neural Networks

- Consider a network with *d* layers

$$\mathbf{h}^t = f_t(\mathbf{h}^{t-1}) \quad \text{and} \quad y = f_d(f_{d-1}(f_{d-2} \cdots f_1(x)))$$

- Compute the gradient of the loss ℓ w.r.t. $\mathbf{W}_t$

$$\frac{\partial \ell}{\partial \mathbf{W}^t} = \frac{\partial \ell}{\partial \mathbf{h}^d} \underbrace{\frac{\partial \mathbf{h}^d}{\partial \mathbf{h}^{d-1}} \cdots \frac{\partial \mathbf{h}^{t+1}}{\partial \mathbf{h}^t}} \frac{\partial \mathbf{h}^t}{\partial \mathbf{W}^t}$$

Multiplication of *d-t* matrices

Two Issues for Deep Neural Networks

$$\prod_{i=t}^{d-1} \frac{\partial \mathbf{h}^{i+1}}{\partial \mathbf{h}^{i}}$$

Gradient Exploding

$$1.5^{100} \approx 4 \times 10^{17}$$

Gradient Vanishing

$$0.8^{100} \approx 2 \times 10^{-10}$$

Issues with gradient exploding

- Value out of range:
  - larger numbers and result in a NaN value
- Sensitive to learning rate
  - Not small enough LR -> large weights -> larger gradients
  - Too small LR -> No progress
- Model becomes unstable and give large changes in every update and never leads to the desired model.

Example: MLP

- Assume MLP (without bias for simplicity)

$$f_t(\mathbf{h}^{t-1}) = \sigma(\mathbf{W}^t\mathbf{h}^{t-1}) \qquad \sigma \text{ is the activation function}$$

$$\frac{\partial \mathbf{h}^t}{\partial \mathbf{h}^{t-1}} = \text{diag}\left(\sigma'(\mathbf{W}^t\mathbf{h}^{t-1})\right)(W^t)^T \qquad \sigma' \text{ is the gradient function of } \sigma$$

$$\prod_{i=t}^{d-1} \frac{\partial \mathbf{h}^{i+1}}{\partial \mathbf{h}^i} = \prod_{i=t}^{d-1} \text{diag}\left(\sigma'(\mathbf{W}^i\mathbf{h}^{i-1})\right)(W^i)^T$$

Gradient Exploding

- Use ReLU as the activation function

$$\sigma(x) = \max(0,x) \quad \text{and} \quad \sigma'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Elements of $\prod_{i=t}^{d-1} \frac{\partial \mathbf{h}^{i+1}}{\partial \mathbf{h}^{i}} = \prod_{i=t}^{d-1} \text{diag}\left(\sigma'(\mathbf{W}^i \mathbf{h}^{i-1})\right)(W^i)^T$ may from $\prod_{i=t}^{d-1} (W^i)^T$

0 or 1

  - Leads to large values when *d*-*t* is large

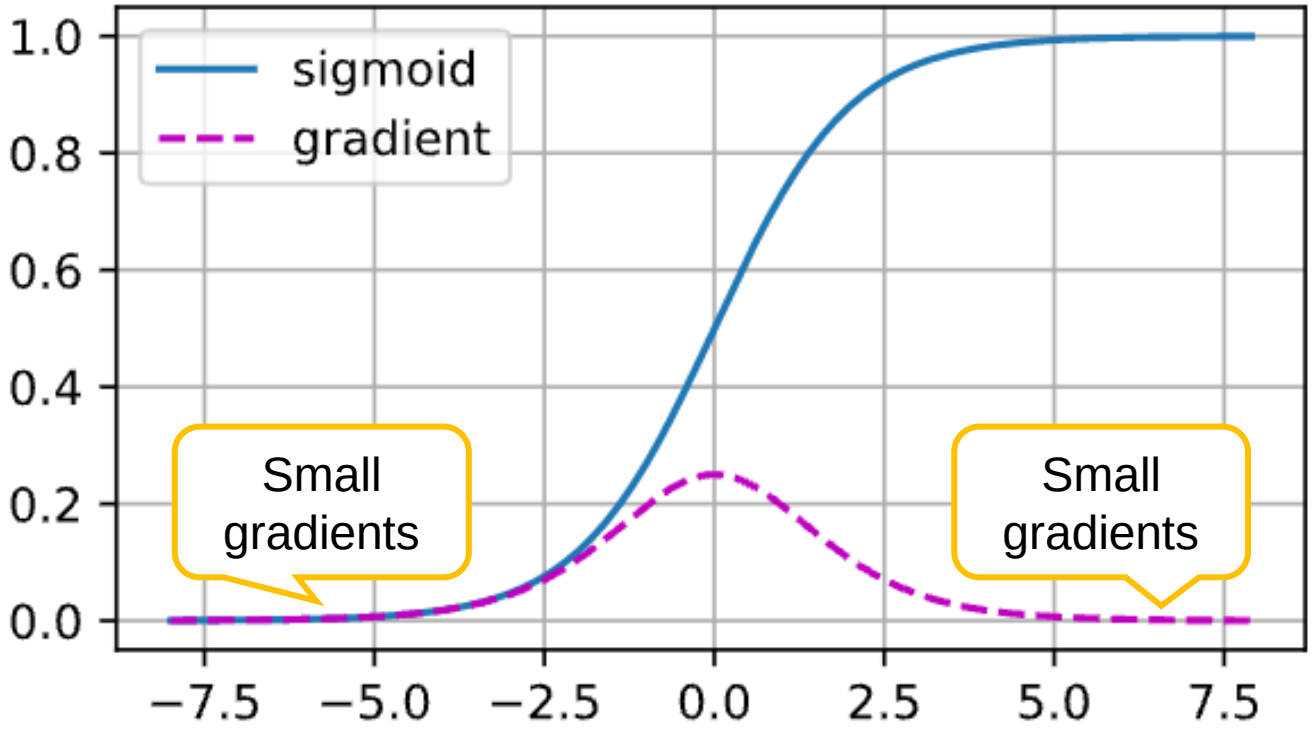
$$1.5^{100} \approx 4 \times 10^{17}$$

Gradient vanishing

$$\mathbf{w}' = \mathbf{w} - \eta \frac{\partial l(\mathbf{y}, \mathbf{wx} + \mathbf{b})}{\partial \mathbf{w}}$$

- Use sigmoid as the activation function

$$\theta_{t+1} = \theta_t - \eta g_t$$



Gradient vanishing

- Use sigmoid as the activation function

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

- Elements $\prod_{i=t}^{d-1} \frac{\partial \mathbf{h}^{i+1}}{\partial \mathbf{h}^{i}} = \prod_{i=t}^{d-1} \text{diag}\left(\sigma'(\mathbf{W}^{i}\mathbf{h}^{i-1})\right)(W^{i})^{T}$ are products of *d-t* small values

(0, 1)

$$0.8^{100} \approx 2 \times 10^{-10}$$

Issues with gradient vanishing

$$\theta_{t+1} = \theta_t - \eta g_t$$

- Gradients with value 0 (such as 2×10^{-10})
- No progress in training
  - No matter how to choose learning rate
- Severe with bottom layers
  - Only top layers are well-trained (close to loss function)
  - No benefit to make networks deeper
  - Make learning slow especially of front layers in the network (close to input layer).

Stabilize Training

- Goal: make sure gradient values are in a proper range
  - E.g. in [1e-6, 1e3]
  - Random weights initialization
- Other networks:
  - ResNet, LSTM (introduced in the following chapter)
- Normalize (introduced in the following chapter)
  - Batch Normalization, Gradient clipping
- Proper **weight initialization** and activation functions

Weight Initialization

- **Initialize weights with random values** in a proper range
- The beginning of training easily suffers to numerical instability
  - The surface **far** away from an optimal can be complex
  - Near optimal may be **flatter**
- Initializing according to uniform or normal distribution (such as N(0, 0.01)) can work well for small networks, but not guarantee for deep neural networks.

Xavier initialization (for tanh, sigmoid)

- Idea: $U\left(-\sqrt{\frac{1}{n}}, \sqrt{\frac{1}{n}}\right)$ and consider variance of input/output

$$U\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right) \qquad N\left(0, \sqrt{\frac{2}{n_{in}+n_{out}}}\right)$$

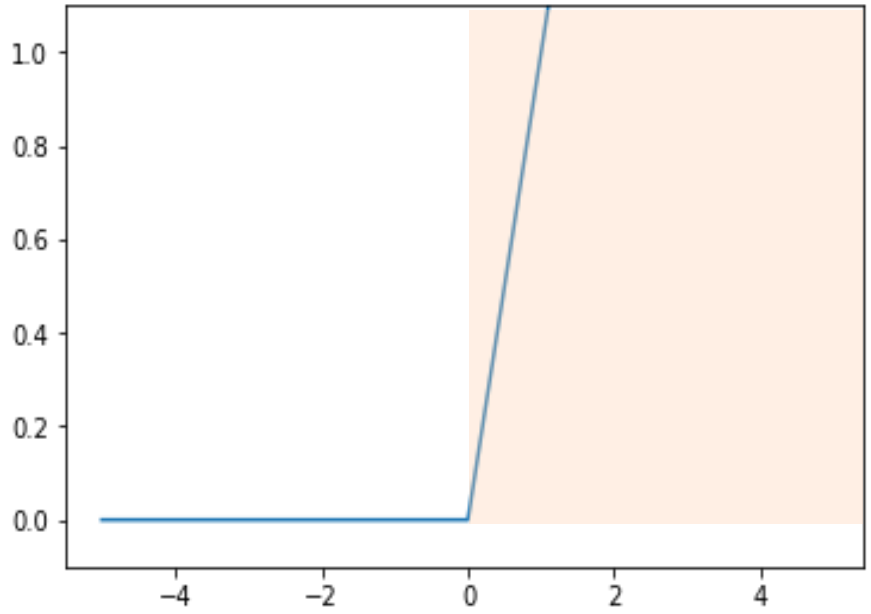
where n_{in} is the number of neurons in the input layer, n_{out} is the number of neurons in the output layer.

He initialization (for ReLU)

$$U\left(-\sqrt{\frac{6}{n}}, \sqrt{\frac{6}{n}}\right)$$

$$N\left(0, \frac{2}{n}\right)$$

$$ReLU(x) = \begin{cases} x & if\ x > 0 \\ 0 & if\ x \leq 0 \end{cases}$$



where n is the number of neurons in the input/output layer

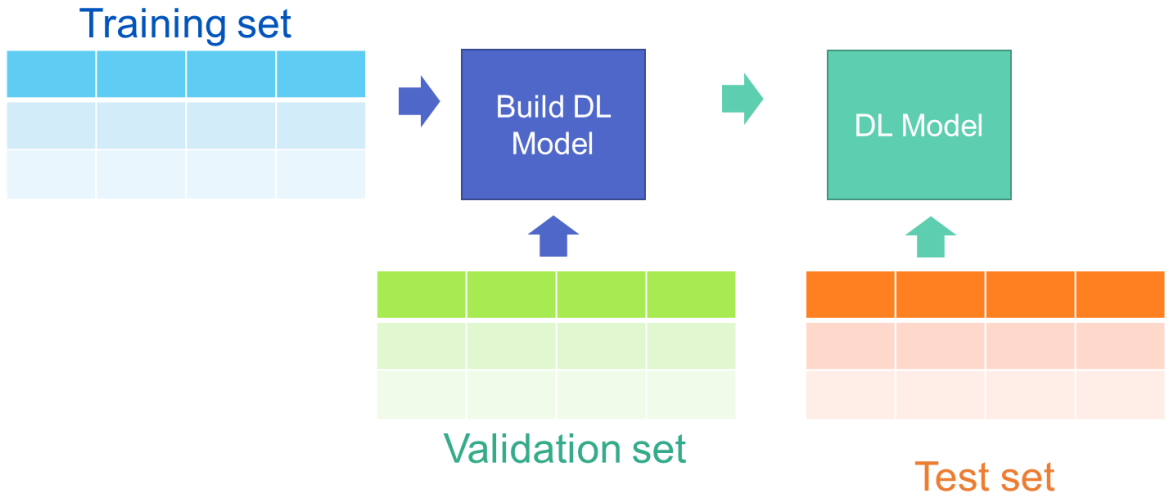
Outline

- Optimizer (d2l.ch11)
- Vanishing/Exploding Gradients (d2l.ch4.8)
- Parameter Initialization (d2l.ch4.8)
- Evaluation (d2l.ch4.4)
  - Dataset
  - Metrics
  - Overfitting or underfitting
- Regularization (d2l.ch4.5)
- Dropout (d2l.ch4.6)

Training error and generalization error

- Training error: model error on the training data
- Generalization error: model error on new data
- Example: practice a future exam with past exams
  - Doing well on past exams (training error) doesn’t guarantee a good score on the future exam (generalization error)
  - Student A gets 0 error on past exams by rote learning
  - Student B understands the reasons for given answers

Dataset



- Training set: a dataset used to learn the model
- Validation set (development set): a dataset used to evaluate the model
  - E.g. Take out 50% of the training data
  - Should not be mixed with the training data
  - This avoids overfitting
- Test set: a dataset can be used once
  - E.g. a future exam, dataset used in private leaderboard in Kaggle
- **Train on training set, tune on dev set, report on test set**

Dataset

- Want as much data as possible for training, and as much for dev; how to split?
  - (80%, 10%, 10%), (80%, 15%, 5),(90%, 5%, 5%)

- Notice that follows **the same probability distribution** as the training data set.

Evaluation metrics for continuous outputs

- Mean Square Error(MSE)/Root Mean Square Error(RMSE)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \qquad RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

- Mean Absolute Error(MAE)

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

Evaluation metrics for discrete outputs

- Precision: % of items the system detected (i.e., items the system labeled as positive) that are in fact positive (according to the human gold labels)

$$Precision = \frac{|correct|}{|prediction|}$$

- Recall: % of items actually present in the input that were correctly identified by the system.

$$Recall = \frac{|correct|}{|actual\ correct|}$$

Evaluation: Accuracy

- Why don't we use **accuracy** as our metric?
- Imagine we saw 1 million tweets
  - 100 of them talked about Delicious Pie Co.
  - 999,900 talked about something else
- We could build a dumb classifier that just labels every tweet "not about pie"
  - It would get 99.99% accuracy!!! Wow!!!!
  - But useless! Doesn't return the comments we are looking for!
  - That's why we use **precision** and **recall** instead

A combined measure: F

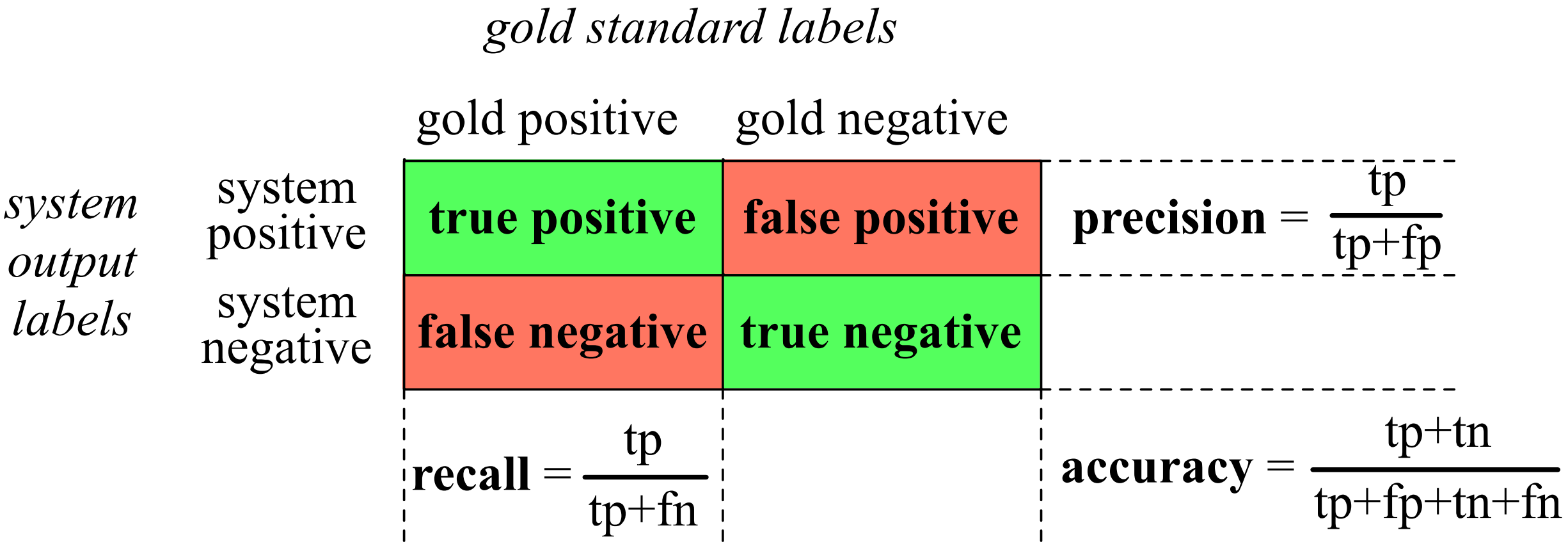
- F measure: a single number that combines P and R:

$$F_\beta = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}$$

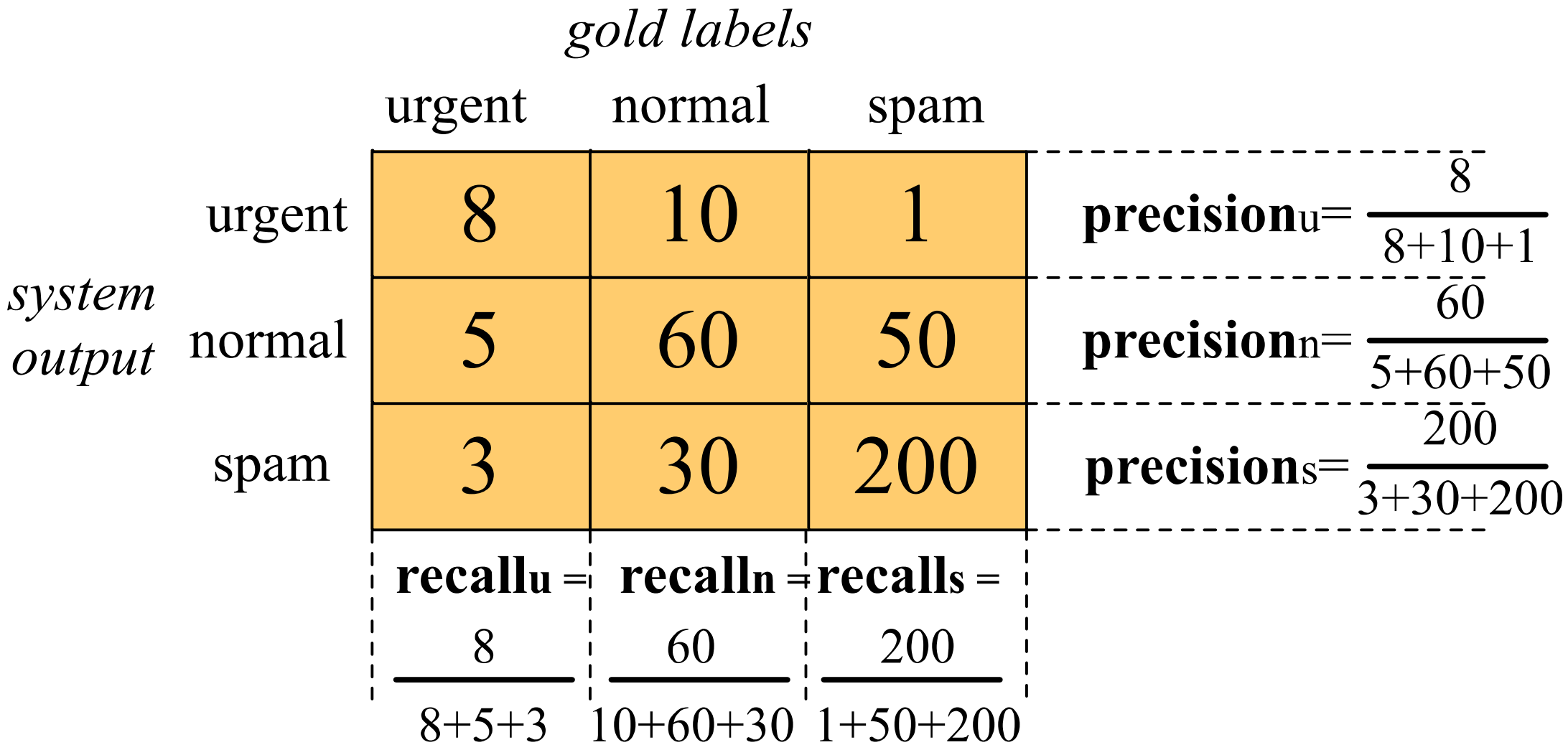
- We almost always use balanced F_1 (i.e., $\beta = 1$)

$$\mathrm{F}_1 = \frac{2PR}{P + R}$$

The 2-by-2 confusion matrix



Confusion Matrix for 3-class classification



How to combine P/R from 3 classes to get one metric

- Macro-averaging:
  - compute the performance for **each class**, and then average over classes

- Micro-averaging:
  - collect decisions for **all classes** into one confusion matrix
  - compute precision and recall from that table.

Macro-averaging and Micro-averaging

Class 1: Urgent

	true urgent	true not
system urgent	8	11
system not	8	340

$$\text{precision} = \frac{8}{8+11} = .42$$

Class 2: Normal

	true normal	true not
system normal	60	55
system not	40	212

$$\text{precision} = \frac{60}{60+55} = .52$$

Class 3: Spam

	true spam	true not
system spam	200	33
system not	51	83

$$\text{precision} = \frac{200}{200+33} = .86$$

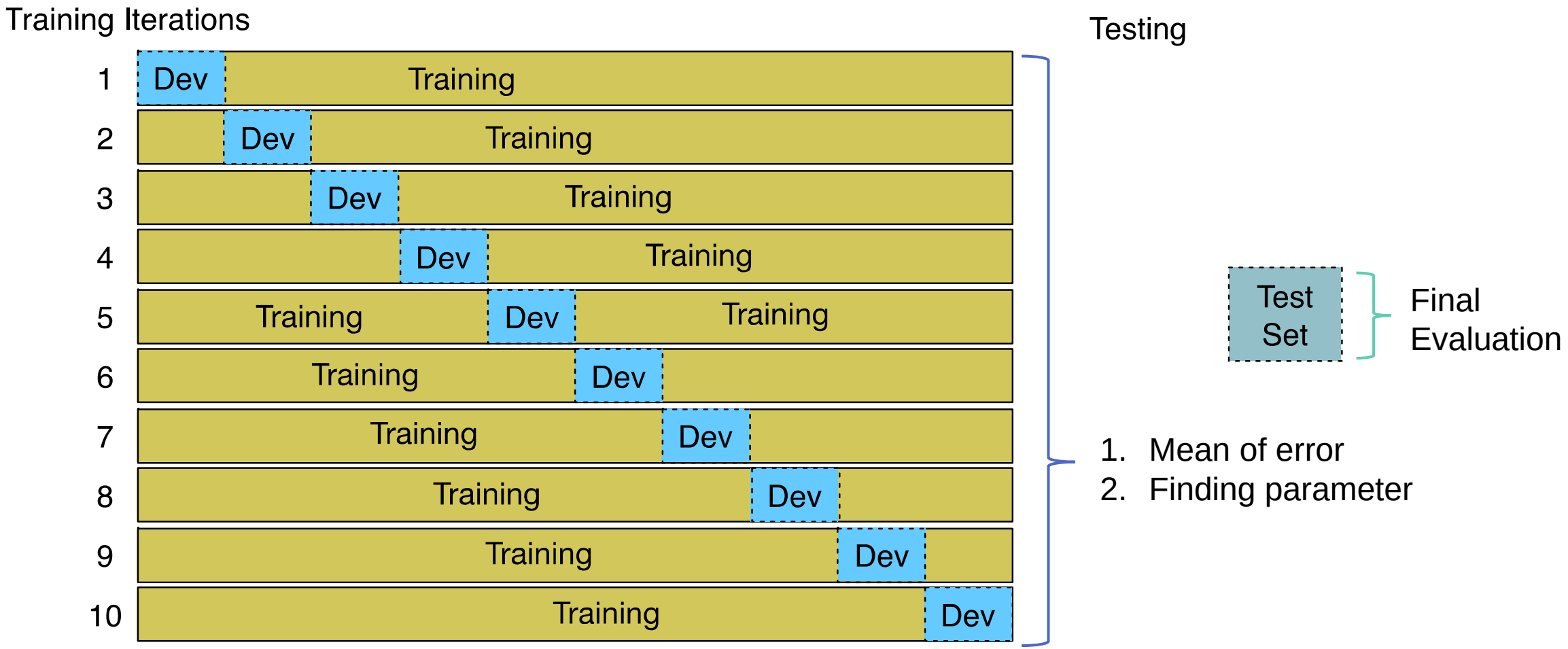
Pooled

	true yes	true no
system yes	268	99
system no	99	635

$$\text{microaverage precision} = \frac{268}{268+99} = \mathbf{.73}$$

$$\text{macroaverage precision} = \frac{.42+.52+.86}{3} = \mathbf{.60}$$

Cross-validation: multiple splits



K-fold Cross-validation

- Algorithm:
  - Partition the training data into K parts
  - For i = 1, …, K
    - Use the i-th part as the validation set, the rest for training
  - Report the averaged the K validation errors
- Popular choices: K = 3 or 5 or 10

Outline

- Optimizer (d2l.ch11)
- Vanishing/Exploding Gradients (d2l.ch4.8)
- Parameter Initialization (d2l.ch4.8)
- Evaluation (d2l.ch4.4)
  - Dataset
  - Metrics
  - Overfitting or underfitting
- Regularization (d2l.ch4.5)
- Dropout (d2l.ch4.6)

Underfitting & Overfitting

- # of examples
- # of elements in each example
- time/space structure
- diversity

Data complexity

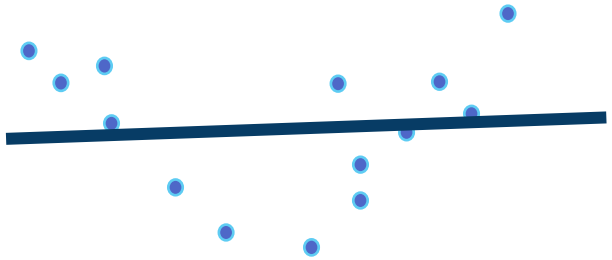
Model capacity

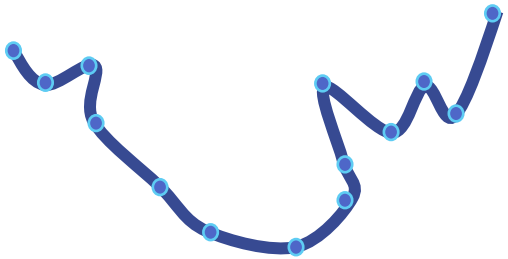
The number of parameters

	Simple	Complex
Low	Normal	Underfitting
High	Overfitting	Normal

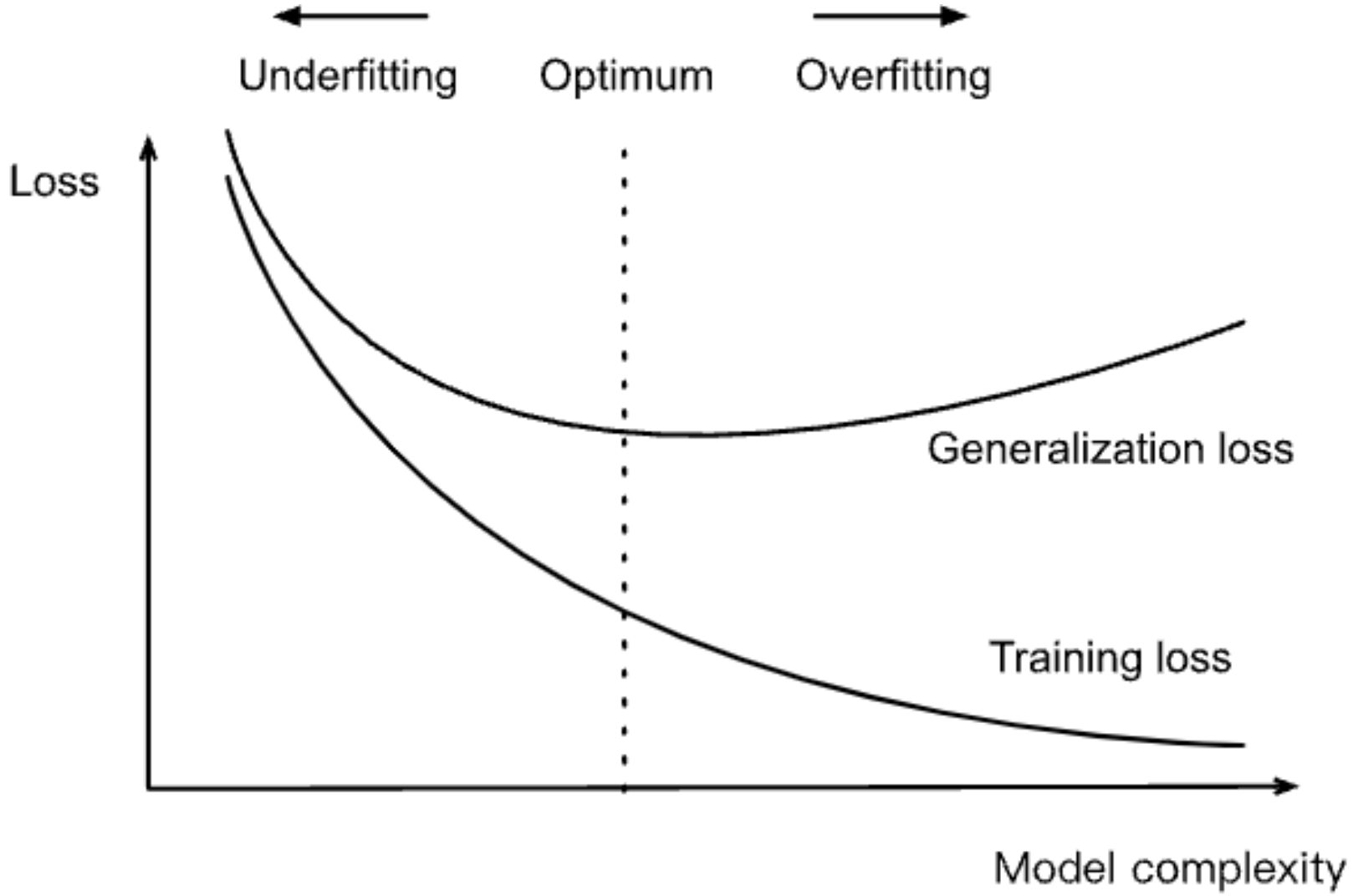
Model capacity

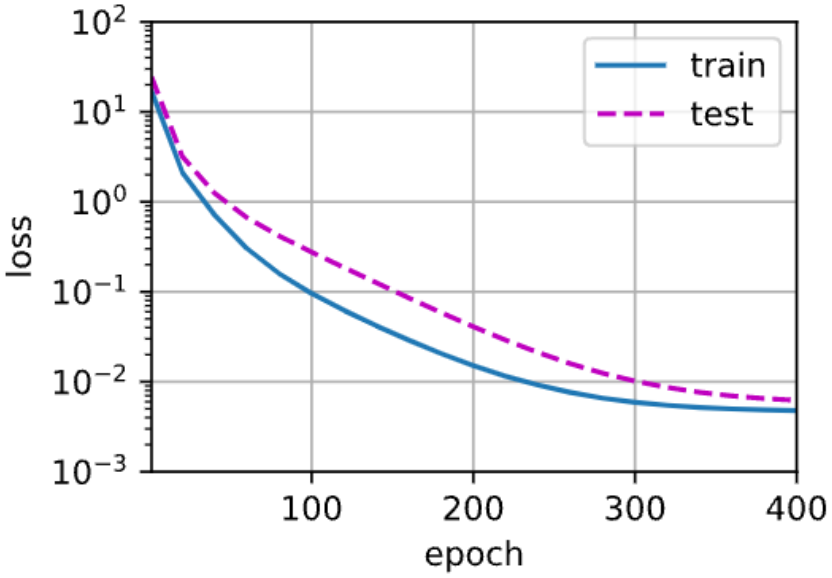
- The ability to fit variety of functions
- Low capacity models struggles to fit training set
  - Underfitting
- High capacity models can memorize the training set
  - Overfitting



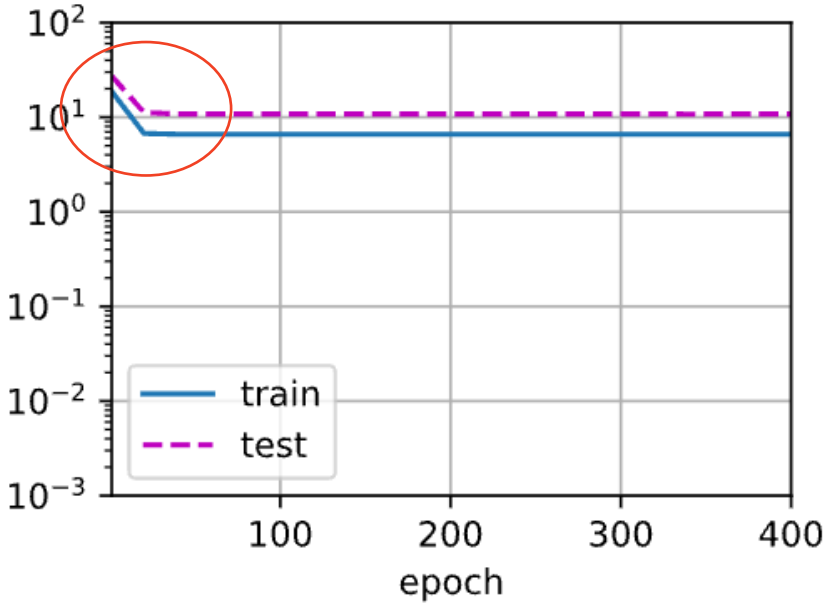


Influence of model complexity

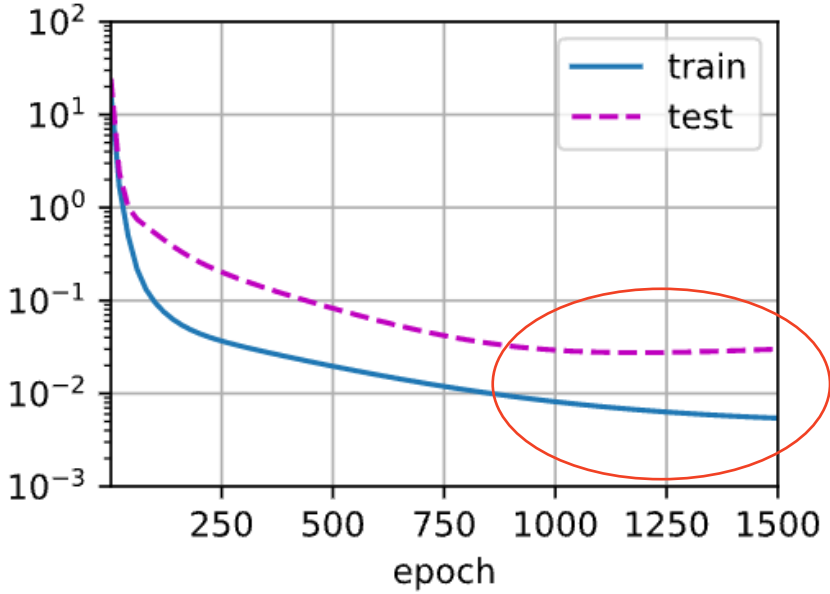




Normal



Underfitting



Overfitting

Overfitting

- A model that **perfectly** match the training data has a problem.

- It will also **overfit** to the data, modeling noise
  - Failing to generalize to a test set without this input.

- A good model should be able to **generalize**

Example

\+

This movie drew me in, and it'll do the same to you.

\-

I can't tell you how much I hated this movie. It sucked.

Useful or harmless features

X1 = "this"
X2 = "movie
X3 = "hated"
X4 = "drew me in"

4gram features that just "memorize" training set and might cause problems

X5 = "the same to you"
X7 = "tell you how much"

Overfitting

- 4-gram model on tiny data will just **memorize the data**
  - 100% accuracy on the training set
- But it will be surprised by the novel 4-grams in the test data
  - Low accuracy on test set
- Models that are too powerful can **overfit** the data
  - Fitting the details of the training data so exactly that the model doesn't generalize well to the test set
    - How to avoid overfitting?
      - Regularization
      - Dropout

Outline

- Optimizer (d2l.ch11)
- Vanishing/Exploding Gradients (d2l.ch4.8)
- Parameter Initialization (d2l.ch4.8)
- Evaluation (d2l.ch4.4)
  - Dataset
  - Metrics
  - Overfitting or underfitting
- Regularization (d2l.ch4.5)
- Dropout (d2l.ch4.6)

Regularization

- A solution for overfitting
- Idea: choose an $R(\theta)$ that **penalizes** large weights
  - fitting the data well with lots of **big weights** not as good as fitting the data a little less well, with small weights
- Add a regularization term $R(\theta)$ to the loss function
- A new loss function to be minimized.
  - Find a set of weight not only minimizing original loss but also close to zero.

$$L'(\theta) = L(\theta) + \frac{\lambda}{2} R(\theta)$$

L2 Regularization

- The sum of the squares of the weights
- The name is because this is the (square of the) **L2 norm** $||\theta||_2$.

$$R(\theta) = \left|\left|\theta\right|\right|_2 = \sum_{j=1}^{n} w_j^2$$

Dropout

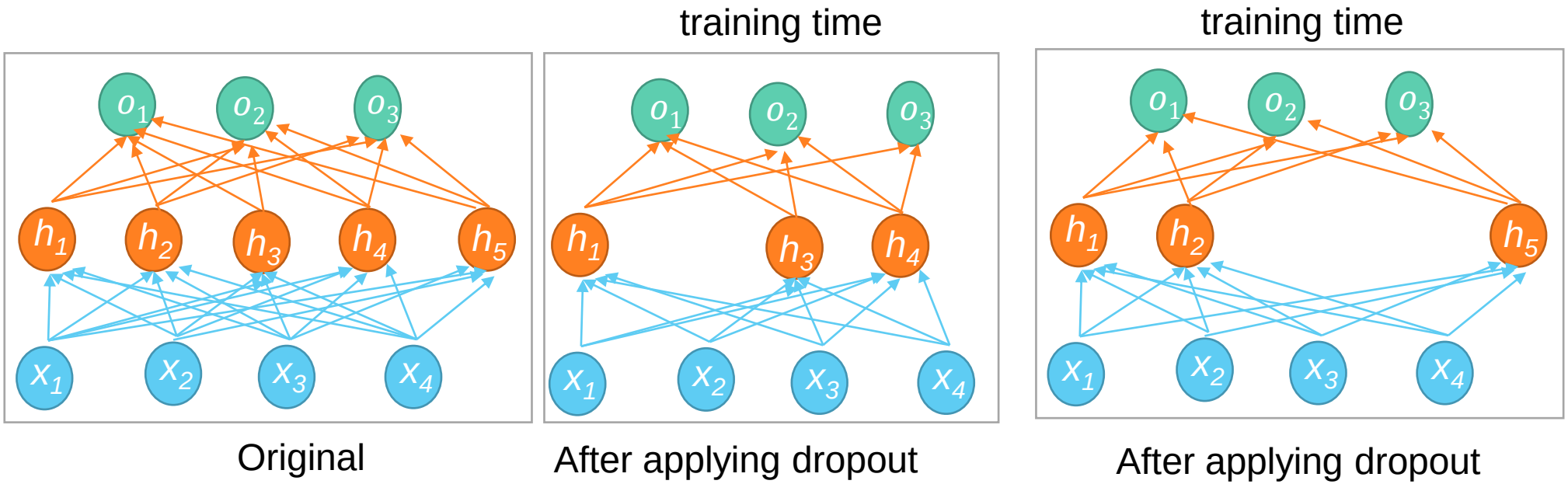
- Motivation: A good model should be **robust** under modest changes in the input.
- Dropout: **inject noises** into internal layers

- Add noise into *h* to get *h'*, we hope:

$$E[h'] = h$$

Dropout

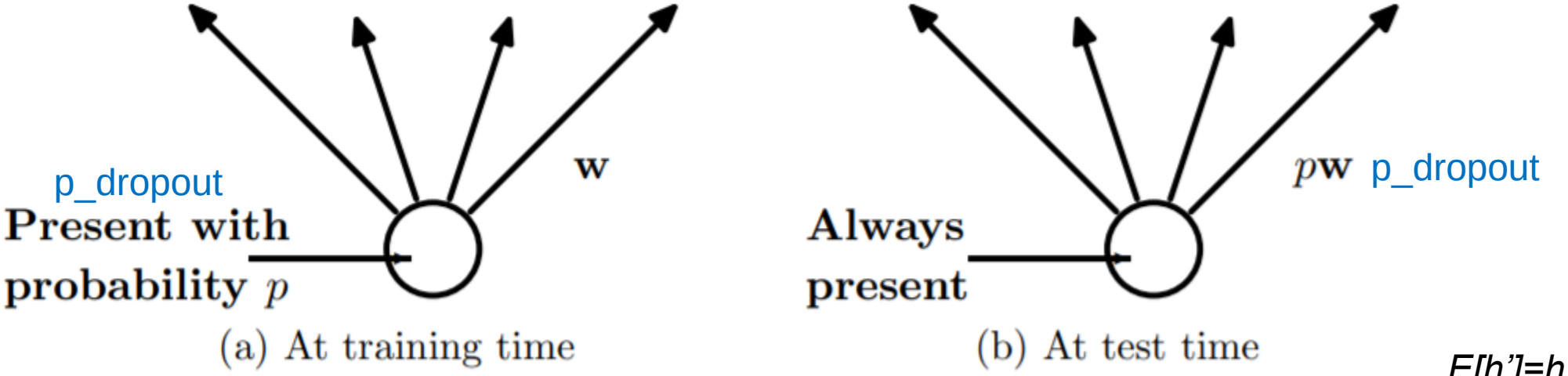
- Dropout has been widely used in deep learning to prevent overfitting.
- During training time, dropout randomly sets node values to zero.
- During inference time, dropout does not kill node values



Dropout during training and testing

reference

- Notice: dropout is only used in training



E[h']=h

$h = \{1,2,3,4,5\} \rightarrow h' = \{1,0,3,4,5\}$

Keep p% = 0.8
Dropout (1-p)% = 0.2

$h = \{1,2,3,4,5\}$
$\rightarrow h' = \{\ 0.8\ ,\ 1.6\ ,\ 2.4\ ,\ 3.2\ ,\ 4.0\ \}$

- Weights should multiply p% (Keep rate) while the weights remain unchanged.

Dropout during training and testing

- To avoid doing work during inference time
- During training,

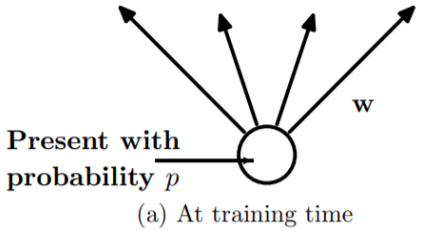
p_dropout

$$h' = \begin{cases} 0 & \text{with dropout probability } p \\ \dfrac{h}{1-p} & \text{otherwise} \end{cases}$$

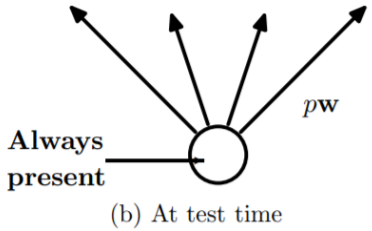
p_keep

$h = \{1,2,3,4,5\}$ $\quad h' = \{1.25,0,3.75,5,6.25\}$

p_keep=0.8
p_dropout=0.2



(a) At training time



(b) At test time

- During testing, h and weights remain unchanged.

Review of MLP

- Input Layer:
  - Preprocessing
  - Continuous/discrete value
- Hidden Layers
  - How many layers and neurons?
- Output Layer
  - neurons, functions

Input Layer: Data Preprocessing

- Miss values:
  - Replacing all missing values by the corresponding input's mean.
- Numerical inputs
  - Standardize: rescaling features to zero mean and unit variance

$$x = \frac{x - \mu}{\sigma}$$

    - It proves convenient for optimization.

- Min-Max Normalization

$$x = \frac{x - \min}{\max - \min}$$

- It does not handle outliers very well.

Hidden Layer

- **Size**: The number of nodes in the model.
- **Width**: The number of nodes in a specific layer.
- **Depth**: The number of layers in a neural network.
- **Capacity**: The type or structure of functions that can be learned by a network configuration

Data complexity

Model capacity	Simple	Complex
Low	Normal	Underfitting
High	Overfitting	Normal

How many layers?

- **Intuition**
- **Borrow ideas**
- Experiment
  - The performance improves when adding additional hidden layers.

Data complexity

Model capacity	Simple	Complex
Low	Normal	Underfitting
High	Overfitting	Normal

How many nodes?

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

Core Training Step (Pytroch)

- output_batch = model(train_batch)
  # compute model output
- loss = loss_fn(output_batch, labels_batch)
  # calculate loss
- optimizer.zero_grad() # clear previous gradients
- loss.backward()
  # compute gradients of all variables wrt loss
- optimizer.step()
  # perform updates using calculated gradients

Source: https://cs230.stanford.edu/blog/pytorch/

Loss Function

- nn.MSELoss
- nn.BCEWithLogitsLoss
- nn.CrossEntropyLoss

```
loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(out, target)
```

Optimizer

- #pick an SGD optimizer
- optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)

- #or pick ADAM
- optimizer = torch.optim.Adam(model.parameters(), lr = 0.0001)

PyTorch code

- Define a neural network
- Initialize params with He initialization
- Define loss function
- Choose optimization algorithm
- Choose initial learning rate
- Choose learning rates schedule
- Make some random data
- Train for 100 batches

```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from torch.optim.lr_scheduler import StepLR

# define input size, hidden layer size, output size
D_i, D_k, D_o = 10, 40, 5
# create model with two hidden layers
model = nn.Sequential(
   nn.Linear(D_i, D_k),
   nn.ReLU(),
   nn.Linear(D_k, D_k),
   nn.ReLU(),
   nn.Linear(D_k, D_o))

# He initialization of weights
def weights_init(layer_in):
   if isinstance(layer_in, nn.Linear):
      nn.init.kaiming_uniform(layer_in.weight)
      layer_in.bias.data.fill_(0.0)
model.apply(weights_init)

# choose least squares loss function
criterion = nn.MSELoss()
# construct SGD optimizer and initialize learning rate and momentum
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
# object that decreases learning rate by half every 10 epochs
scheduler = StepLR(optimizer, step_size=10, gamma=0.5)

# create 100 dummy data points and store in data loader class
x = torch.randn(100, D_i)
y = torch.randn(100, D_o)
data_loader = DataLoader(TensorDataset(x,y), batch_size=10, shuffle=True)

# loop over the dataset 100 times
for epoch in range(100):
   epoch_loss = 0.0
   # loop over batches
   for i, data in enumerate(data_loader):
      # retrieve inputs and labels for this batch
      x_batch, y_batch = data
      # zero the parameter gradients
      optimizer.zero_grad()
      # forward pass
      pred = model(x_batch)
      loss = criterion(pred, y_batch)
      # backward pass
      loss.backward()
      # SGD update
      optimizer.step()
      # update statistics
      epoch_loss += loss.item()
   # print error
   print(f'Epoch {epoch:5d}, loss {epoch_loss:.3f}')
   # tell scheduler to consider updating learning rate
   scheduler.step()
```