ARTIFICIAL INTELLIGENCE NEEDS REAL INTELLIGENCE

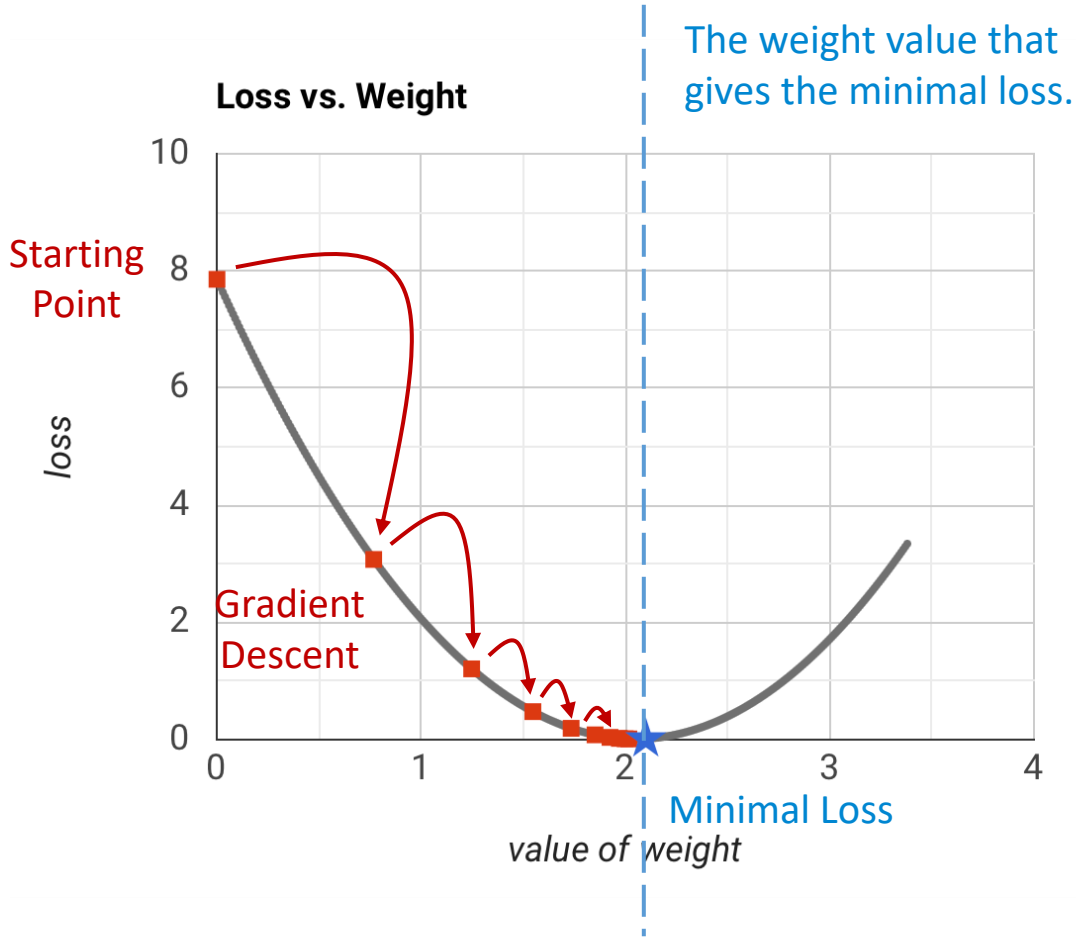
Artificial Neural Network III

Professor: Tianyuan ZHANG
tianyuan.zhang@kedgebs.com

KEDGE BUSINESS SCHOOL

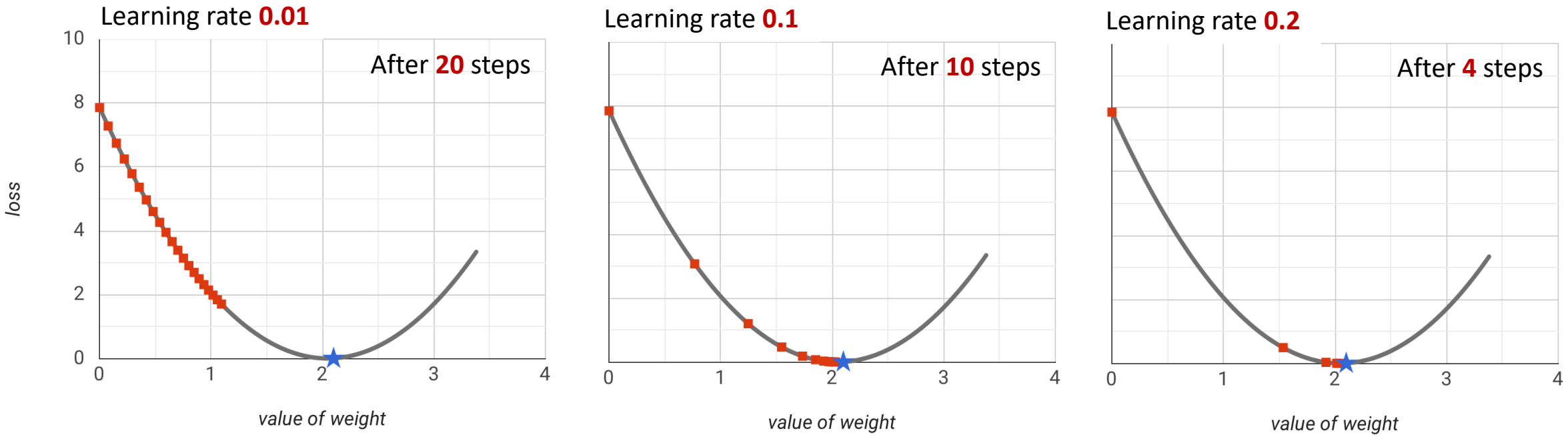
Recap of Previous Session

- **Gradient Descent**
  - An iterative algorithm
  - Take repeated steps to update the parameter values and reduce the loss function
  - For each step:
    - $p \leftarrow p - \eta \times \nabla p$
    - η is the learning rate to scale the step size
    - The direction is opposite to the gradient direction
    - The step size is determined by the magnitude of the gradient and the learning rate



Recap of Previous Session

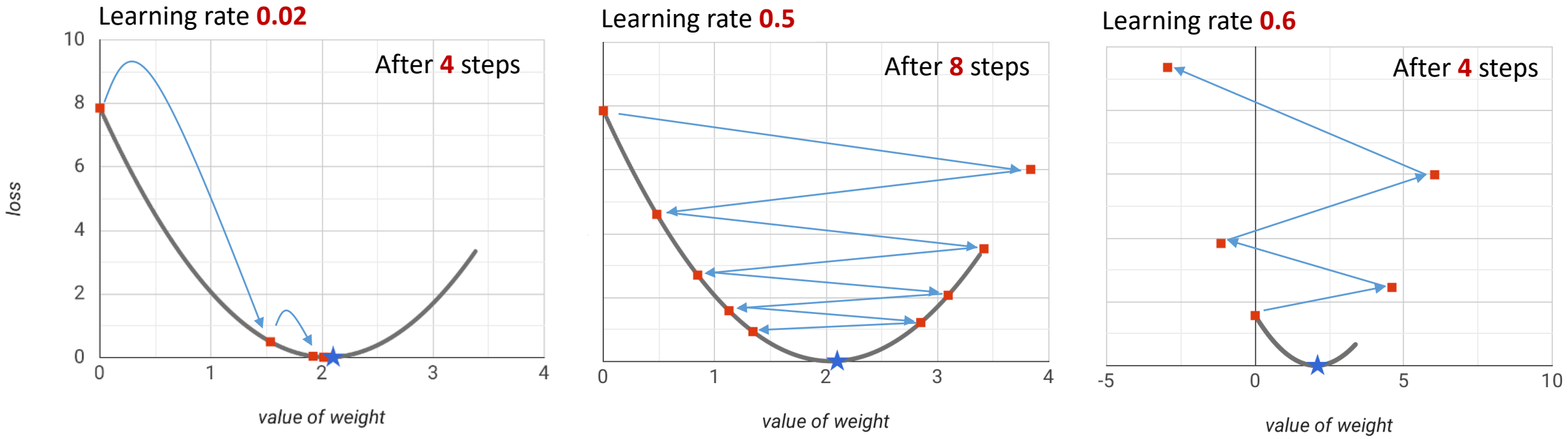
- **Learning rate η affects the step size.**



- A small learning rate will lead to more iterations, less efficient learning process.

Recap of Previous Session

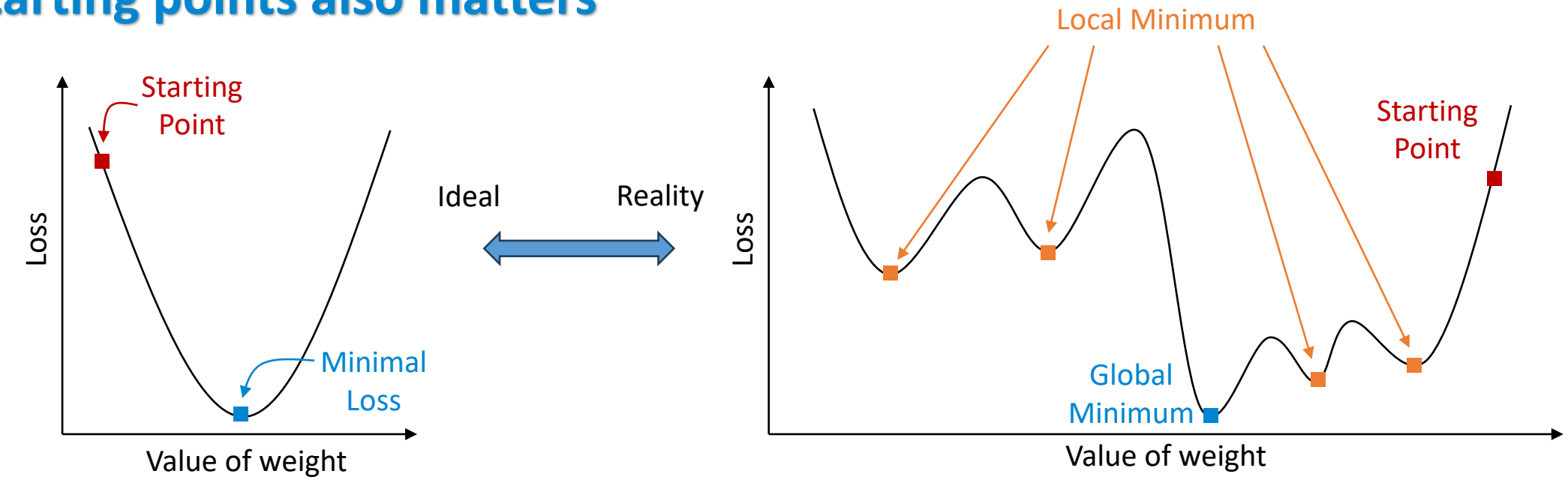
- **Learning rate η affects the step size.**



- A large learning rate may cross the minimal loss
- If learning rate is too large, it never reaches the minimal loss → Divergence

Recap of Previous Session

- **Starting points also matters**



- The starting point is randomly selected when initialize the ANN
- There can be many local minimum, if the algorithm get stuck in one of them, try to re-initialize the network or adjust the learning rate

Recap of Previous Session

- **Gradient descent**
  - Use **all** data samples in the training set to compute the gradient in each iteration
    - More stable, but computationally expensive
- **Stochastic gradient descent**
  - Use a **single** sample in the training set to compute the gradient in each iteration
    - The randomness may lead less accurate gradient directions
    - Although each iteration is faster, it may require more iterations for optimization
- **Mini-batch gradient descent**
  - Use a **batch** of training samples to compute the gradient in each iteration
    - Reduce the randomness in SGD, trade-off between speed and stability
    - The batch size is usually 2^n, smaller is faster but might lead to a local minimal

Recap of Previous Session

- **Backpropagation**
  - Gradient descent + **Chain rule**
  - In each iteration, compute the gradients and update the parameters in ANN:
    - **Layer by layer**
    - **Iterating backward** from the last layer to the input layer
    - Only a new partial derivative needs to be computed for updating a parameter
    - Allows to train very **deep** neural network with many layers
      - No need to update all parameters at the same time
      - Thus, reducing requirements of the computation resources

Recap of Previous Session

- **Implement an ANN**
  1. Build the data pipeline
     - torch.utils.data.Dataset
     - torch.utils.data.DataLoader
  2. Create the artificial neural network
     - Define a custom neural network class
  3. Training by gradient descent
     - Define a train( ) function
  4. Save and load a trained model
  5. Make predictions and evaluation
     - Define a test( ) function

- *Network structure & activation functions*
- *Forward propagation method*

- *Loss function*
- *Optimizer*
- *Learning rate*
- *Number of epochs to train*

Recap of Previous Session

1-layer Linear Neural Network:

```python
# define a custom neural network class
class NeuralNetwork(nn.Module):
    def __init__(self, n_features, n_labels):
        super().__init__()
        self.net = nn.Linear(n_features, n_labels)
    def forward(self, X):
        return self.net(X)
```

Act as a linear regression model

Multi-layer Non-linear Neural Network:

```python
# define a custom neural network class
class NeuralNetwork(nn.Module):
    def __init__(self, n_features, n_labels):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_features, 16),
            nn.ReLU(),
            nn.Linear(16, 8),
            nn.ReLU(),
            nn.Linear(8, 4),
            nn.ReLU(),
            nn.Linear(4, n_labels)
        )
    def forward(self, X):
        return self.net(X)
```

Embed multiple layers in a sequential container

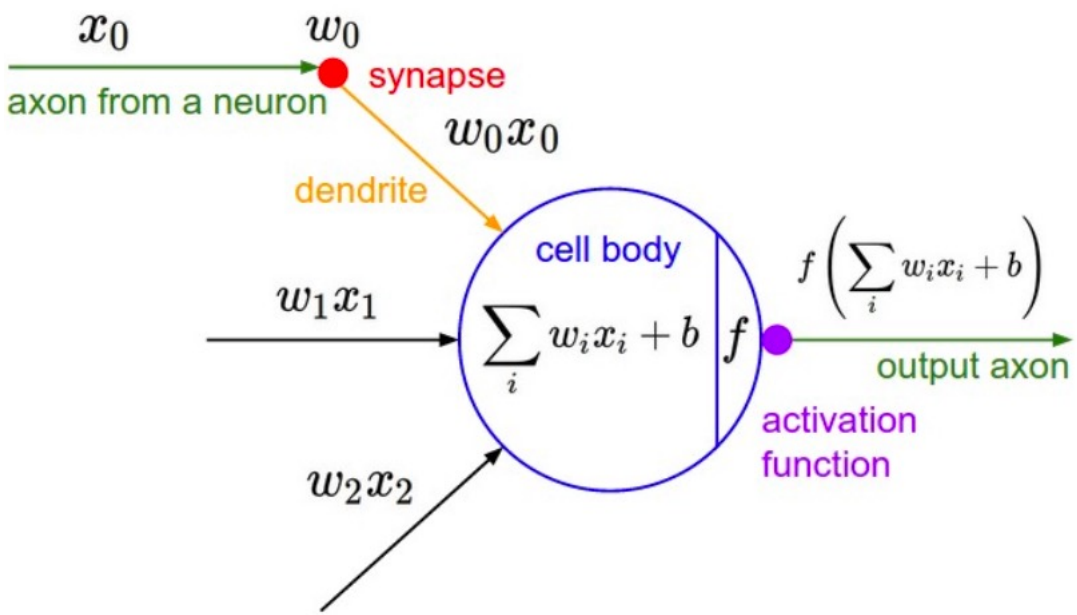
Introduce non-linearity by adding non-linear activation functions

Outline

- **ANN for binary classification**
  - Logistic regression
  - Generalization and Regularization
    - Early stopping
    - Weight decay
    - Learning decay
- ANN for multi-class classification
  - Logits vs. Probabilities vs. Predicted classes
  - Dropout
  - Batch normalization

ANN for binary classification

- ANN as a **logistic regression** model
  - If the activation function is the sigmoid function:
    - $f(z) = 1/(1 + e^{-z})$
  - The output of the artificial neuron:
    - $f(x) = 1/\left(1 + e^{-(\sum_i w_i x_i + b)}\right)$
  - We can use a single artificial neuron with sigmoid function as a logistic regression model



ANN for binary classification

- ANN as a **logistic regression** model
  - Input labels:

Class	Dog	Cat	Cat	…	Dog
Label	1	0	0	…	1

    - Binary classification problem
    - Label is either 1 or 0, 1 represents the positive class, 0 represents the negative class
  - Output:

Prob	0.9	0.2	0.1	…	0.75
Pred	1	0	0	…	1

    - Probabilities: The probability of the input sample belonging to the positive class
    - Use a threshold (0.5) to convert the probabilities into predicted classes

ANN for binary classification

- ANN for binary classification problems:
  - A single artificial neuron with **sigmoid** function → A logistic regression model
  - A multi-layer ANN → For more complex binary classification problems
    - The activation function of the output neuron needs to be the **sigmoid** function
      - Make sure the output of the network can be seen as the probability of the input sample belongs to the positive class
  - The loss function
    - The Binary Cross Entropy between the label and the output probabilities
    - torch.nn.BCELoss()
    - $L = \sum_n l_n = -(y\log(p) + (1-y)\log(1-p))$

Outline

- ANN for binary classification
  - Logistic regression
  - **Generalization and Regularization**
    - Early stopping
    - Weight decay
    - Learning decay
- ANN for multi-class classification
  - Logits vs. Probabilities vs. Predicted classes
  - Dropout
  - Batch normalization

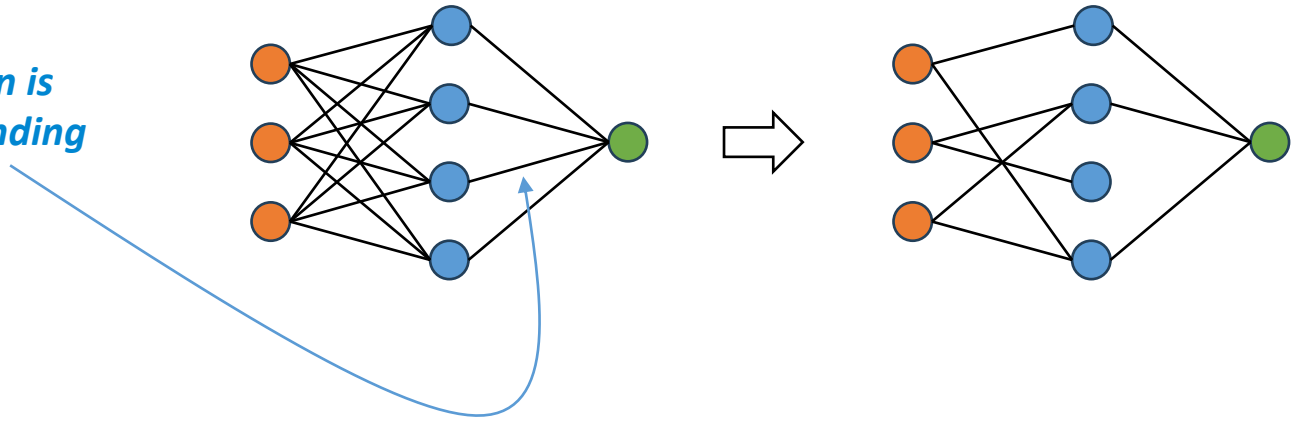
Generalization and Regularization

- We prefer a model to:
  - Have **good fitness** to the training data
    - Minimize the loss on the training set
    - If the network is too simple, the fitness may be poor, resulting in an under-fitted model
      - **Increase the capacity** of the network → Use a deeper one
      - Introduce **non-linearity** into the network → Use non-linear activation function
  - **Generalize well** on unseen data
    - Apart from the training and set, we need a **validation set** to monitor the generalizability
      - In addition to the training loss, **the validation loss also need to be reduced**
    - If the network is too complex, it may over-fit to the training data, resulting in poor generalizability
      - We can decrease the capacity of the network
      - More commonly, we have other techniques to prevent over-fitting → **Regularization** techniques

Generalization and Regularization

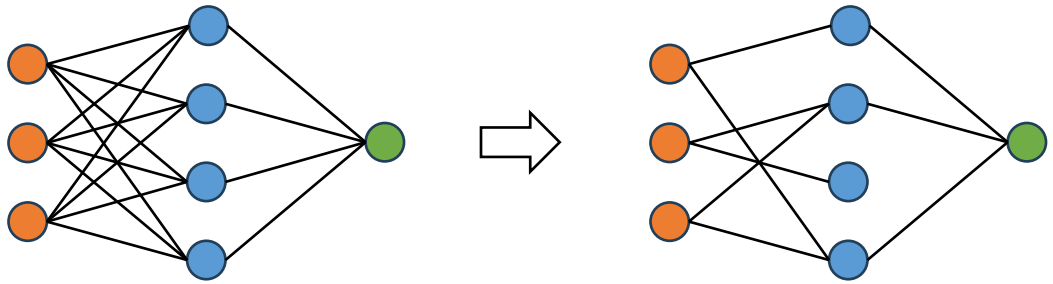
- **Regularization**
  - **Add a penalty on the complexity of the network to the loss function**
    - $Loss = CrossEntropyLoss + \boldsymbol{Penalty(Complexity)}$
  - When we try to reduce the loss value in the training process, we also reduce the penalty term, resulting in a simpler model
  - The network structure cannot be changes during the training process
  - But we can **deactivate some of the connections**

Deactivate this connection is equal to set the corresponding weight to zero



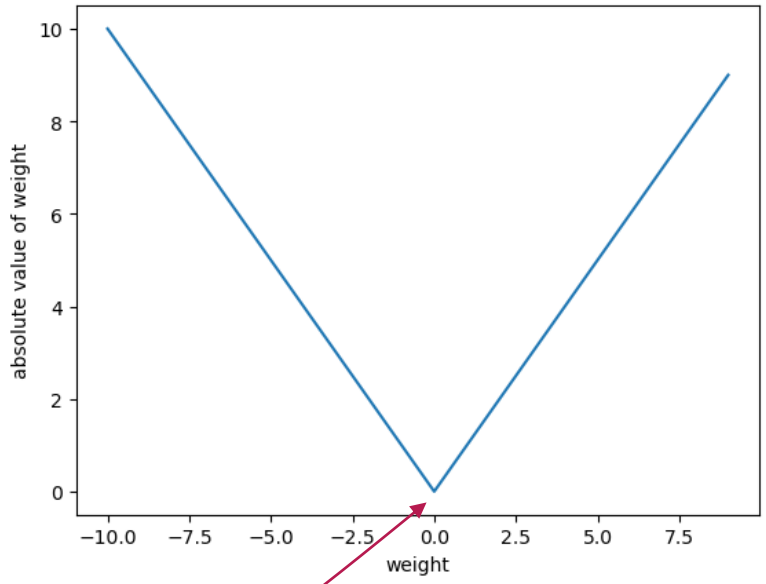
Generalization and Regularization

- **L1 Norm Regularization**
  - Add a **penalty on the L1 Norm of the weights** of the network to the loss function
    - $Loss = CrossEntropyLoss + \lambda \sum |weight|$
    - $|weight|$ is the absolute value of the $weight$
    - λ is used to adjust the strength of this penalty term
  - When we reduce the Loss value, we also reduce the L1 Norm of the weights
    - $\boldsymbol{|weight| \rightarrow 0}$
    - This will deactivate the connection
  - **We don't use it in practice.**



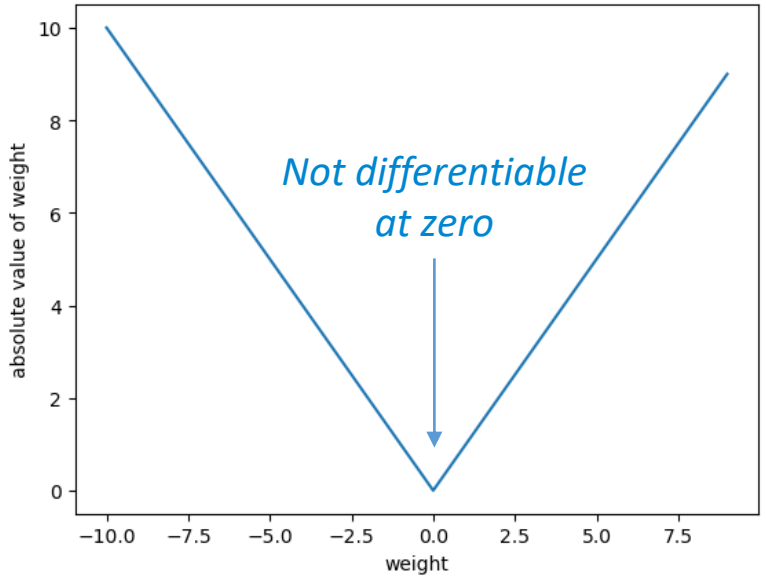
Generalization and Regularization

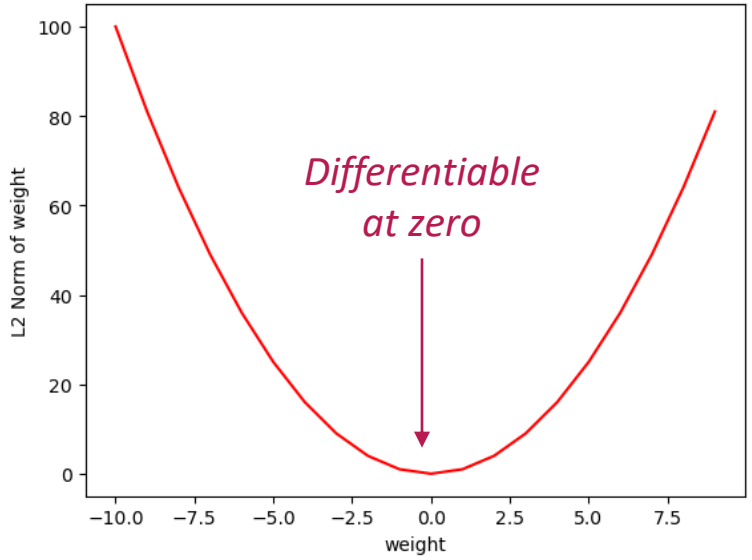
- **L1 Norm Regularization**
  - Add a penalty on the L1 Norm of the weights of the network to the loss function
    - $Loss = CrossEntropyLoss + \lambda \sum |weight|$
  - When we reduce the Loss value
    - $|weight| \rightarrow 0$
    - This will deactivate the connection
  - We don't use it in practice.
    - **The L1 Norm is not differentiable.**
    - **We can't apply gradient descent directly.**



Generalization and Regularization

- **L2 Norm Regularization**
  - Add **a penalty on the L2 Norm of the weights** of the network to the loss function
    - $Loss = CrossEntropyLoss + \lambda \sum |weight|^2$
  - The L2 Norm is differentiable.



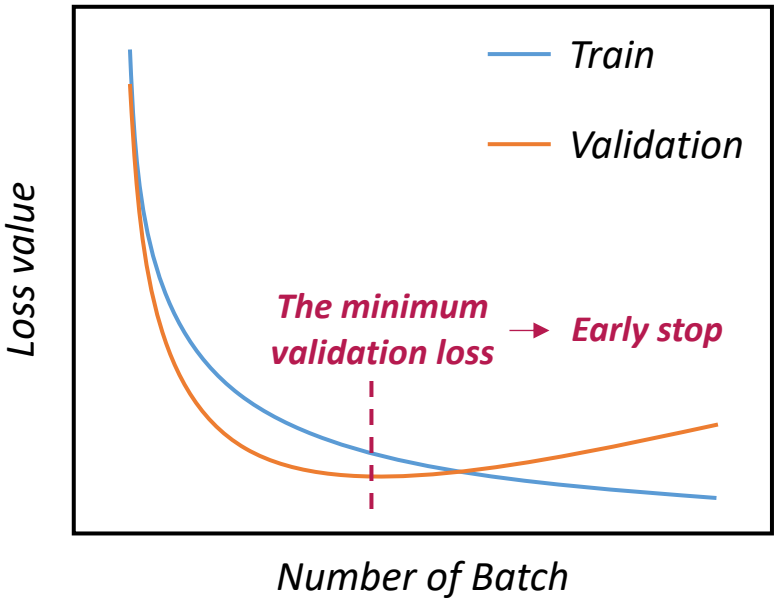


Generalization and Regularization

- **L2 Norm Regularization**
  - Add **a penalty on the L2 Norm of the weights** of the network to the loss function
    - $Loss = CrossEntropyLoss + \lambda \sum |weight|^2$
  - The L2 Norm is differentiable.
  - Reduce the loss function will reduce the weights → **Weight decay**
  - In PyTorch, we have a weight_decay parameter for the optimizer
    - torch.optim.SGD(weight_decay)
      - **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

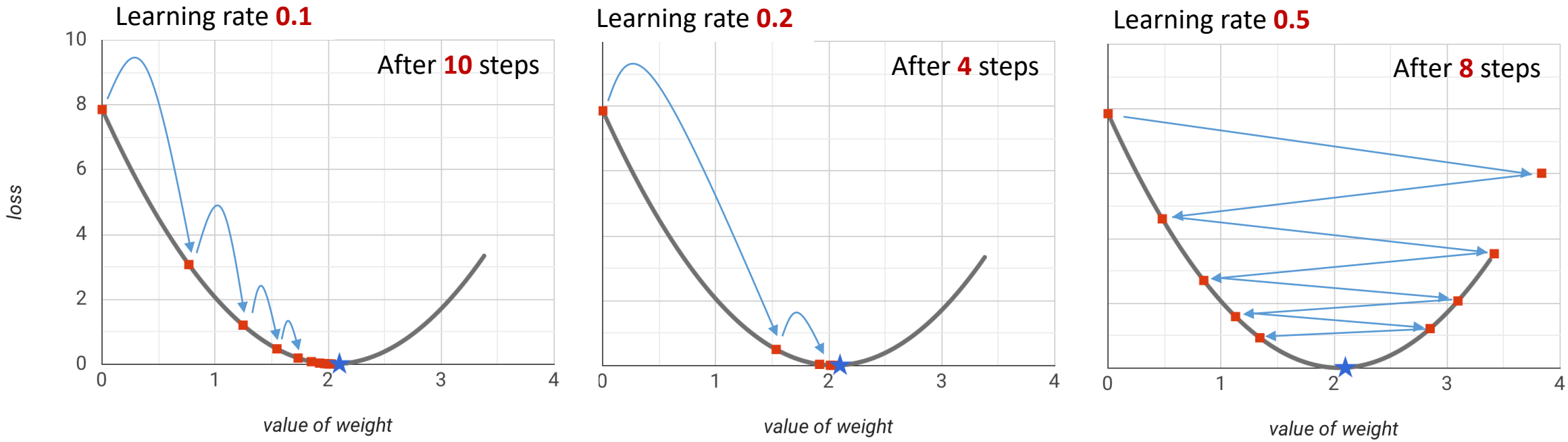
Generalization and Regularization

- Other regularization technique
  - **Early stopping**
    - Suppose we have **a network with enough capacity to be an over-fitting model**
    - In the training process:
      - The train loss will be reduced continuously
      - The validation loss:
        - First decrease
        - Then increase → The network starts to overfit
    - We can perform early stopping when the validation loss starts to increase to prevent the model from over-fitting
      - Doing this, we don't need to decrease the capacity
      - So we can always start with a complex network



Generalization and Regularization

- Other regularization technique
  - **Learning rate decay**
    - Learning rate η is used to scale the step size in gradient descent.



Generalization and Regularization

- Other regularization technique
  - **Learning rate decay**
    - Learning rate η is used to scale the step size in gradient descent.
    - At the beginning → Far away from the minimal loss → A large learning rate → Approach quickly
    - At the end → Close to the minimal loss → A small learning rate → Don't cross the minimal loss
    - We can start with a large learning rate and gradually reduce it.

CLASS torch.optim.lr_scheduler.ReduceLROnPlateau(*optimizer, mode='min', factor=0.1, patience=10, threshold=0.0001, threshold_mode='rel', cooldown=0, min_lr=0, eps=1e-08, verbose='deprecated')* [SOURCE]

Reduce learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This scheduler reads a metrics quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

Monitor the validation loss to prevent over-fitting

Hands-on Exercise

- **Exercise 05 ANN for Binary Classification – Instruction**
- Exercise 05 ANN for Multi-class Classification - Instruction
- Exercise 05 ANN for Classification - Assignment

Outline

- ANN for binary classification
  - Logistic regression
  - Generalization and Regularization
    - Early stopping
    - Weight decay
    - Learning decay
- **ANN for multi-class classification**
  - Logits vs. Probabilities vs. Predicted classes
  - Dropout
  - Batch normalization

ANN for multi-class classification

- For multi-class problems:
  - Input labels:

Class	Dog	Cat	Monkey	Monkey	Dog
Label	1	2	3	3	1

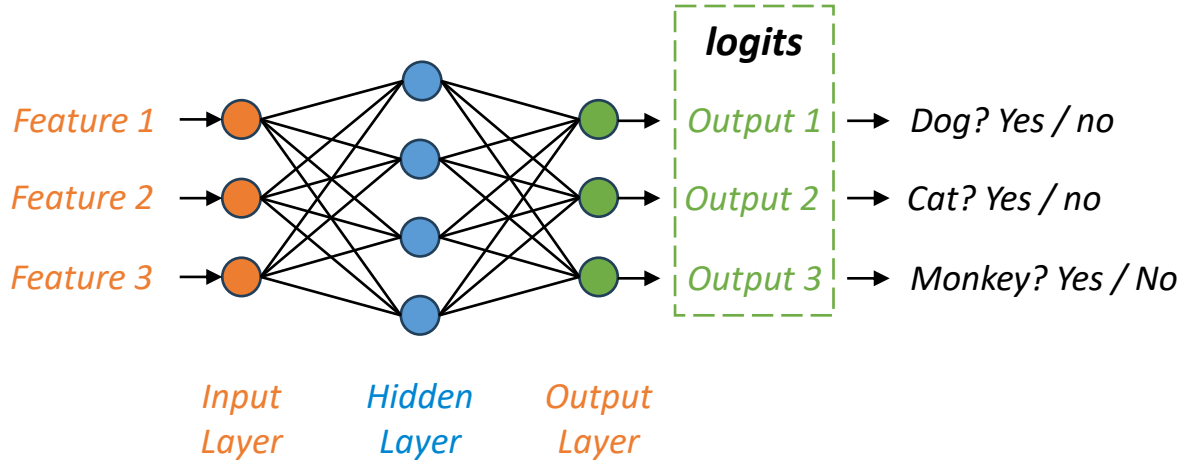
  - We need to encode the input labels using **one-hot encoding**:

Dog	1	0	0	0	1
Cat	0	1	0	0	0
Monkey	0	0	1	1	0

  - Instead of having one output neuron that predict the class directly
  - We need multiple output neurons to estimate the probabilities of the input sample belonging to each class

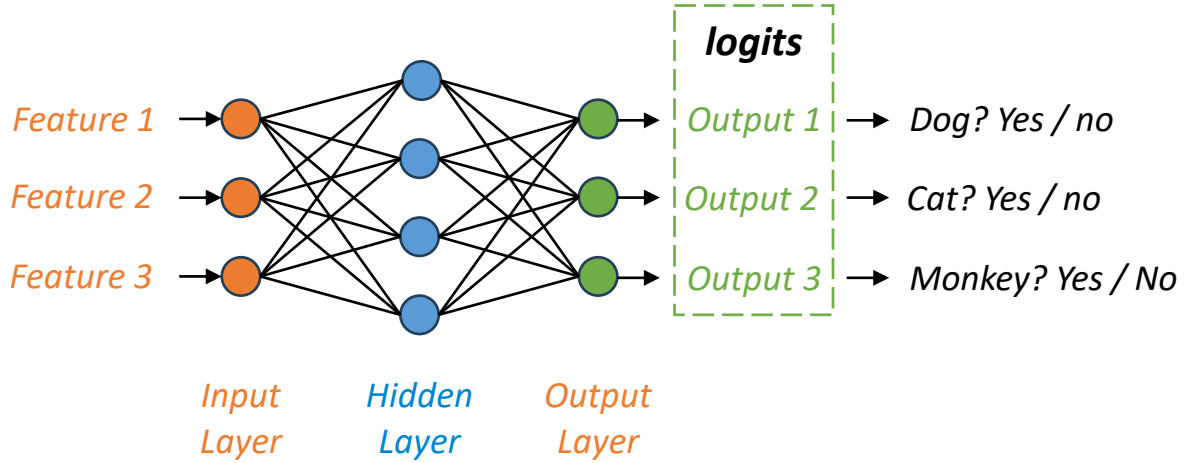
ANN for multi-class classification

- For multi-class problems:
  - We need multiple output neurons to estimate the probabilities of the input sample belonging to each class
    - **The number of output neurons is equal to the number of classes**
  - By default, **the activation function of the output layer is linear**
    - Not sigmoid function like binary classification
    - We call the outputs the **logits**



ANN for multi-class classification

- Logits:
  - The activation function of the output neuron is linear
  - There is no limit to the logit value:
    - Between 0 and 1
    - **Or greater than 1**
    - **Or negative**
  - We can use logits to calculate the loss
    - Cross Entropy Loss
  - We can't regard the logits as the probabilities

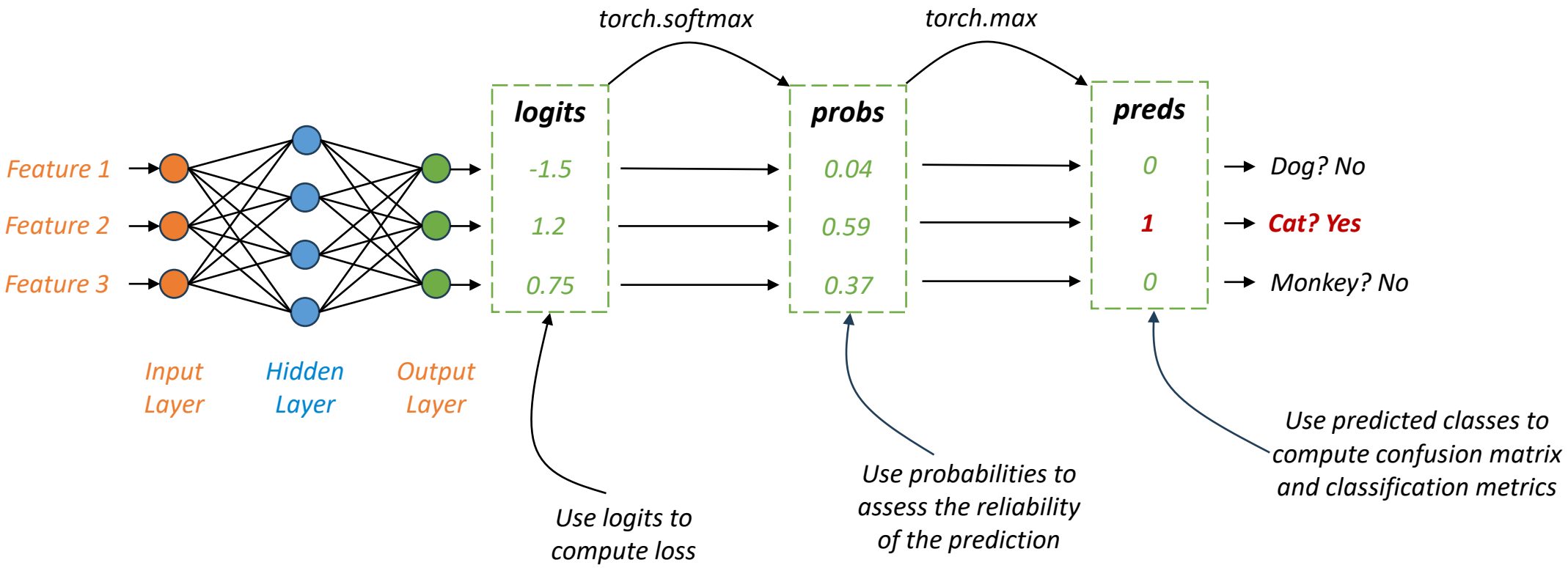


ANN for multi-class classification

- Logits:
  - There is no limit to the logit value
  - We can't regard the logits as the probabilities
- Softmax:
  - Convert logits to probabilities
  - $\hat{p}_i = \exp(o_i)/\sum_i \exp(o_i)$
    - Guarantee the sum of the probabilities is 1
    - Guarantee the probability is positive
  - torch.softmax( )

ANN for multi-class classification

- Logits vs. Probabilities vs. Predicted classes

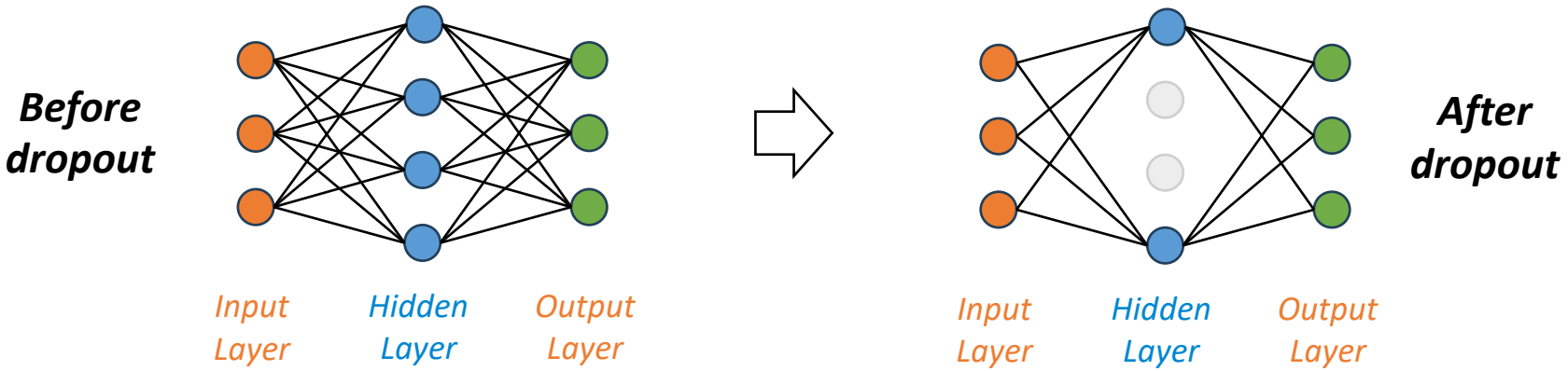


Outline

- ANN for binary classification
  - Logistic regression
  - Generalization and Regularization
    - Early stopping
    - Weight decay
    - Learning decay
- ANN for multi-class classification
  - Logits vs. Probabilities vs. Predicted classes
  - **Dropout**
  - Batch normalization

Dropout

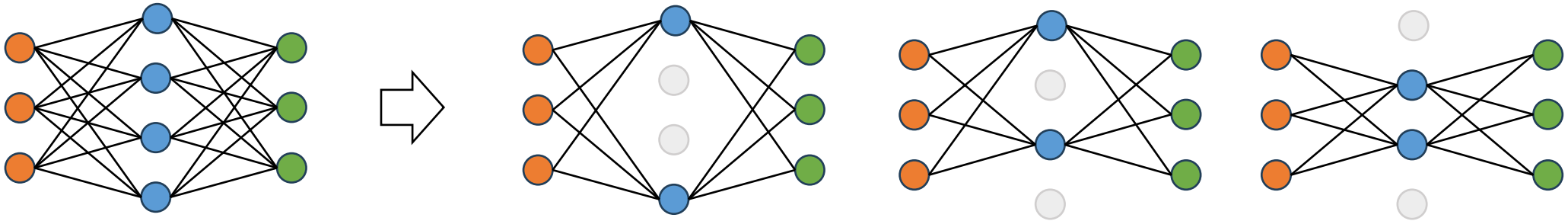
- A **regularization** technique
- **In each training iteration, randomly drop some neurons** based on the drop rate (the probability to drop each neuron)



- Reduce the complexity of the network in each training iteration
- Prevent the network from over-fitting

Dropout

- In different training iterations, drop different neurons based on the drop rate



  - This means each batch in the training phase will essentially train a different network, with different neurons dropped out.
- During inference on validation or test set, dropout is not applied.
  - All neurons are retained to make the predictions.
  - Neurons' outputs are scaled down with the drop rate to balance the fact that more neurons are active than during training.

Dropout

CLASS torch.nn.Dropout(*p=0.5, inplace=False*) [SOURCE]

During training, randomly zeroes some of the elements of the input tensor with probability `p`.

The zeroed elements are chosen independently for each forward call and are sampled from a Bernoulli distribution.

- The dropout layer will have different behaviors in different process
  - For the training process, drop neurons randomly
  - For the validation and test set, use all neurons to make predictions
- We need to tell the network when is the training process, and when is not
  - Use `model.train( )` to set the network in the training mode
  - Use `model.eval( )` to set the network in the evaluation mode

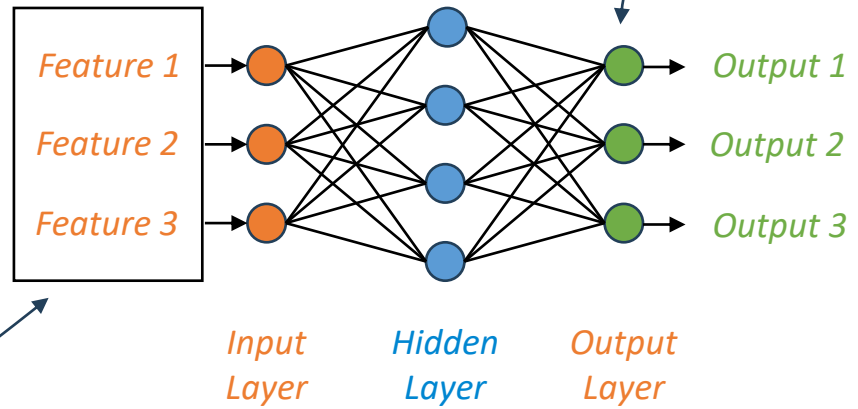
Outline

- ANN for binary classification
  - Logistic regression
  - Generalization and Regularization
    - Early stopping
    - Weight decay
    - Learning decay
- ANN for multi-class classification
  - Logits vs. Probabilities vs. Predicted classes
  - Dropout
  - **Batch normalization**

Batch normalization

- A technique to **accelerate the convergence** of the training process
- Also **prevent over-fitting** as a regularization technique
- **Normalize the inputs of a layer** by re-centering and re-scaling **for each batch**

We observe that feature scaling can make training faster and lead to better fitness



The idea behind batch normalization is similar: for other layers, scale the inputs may also be beneficial

- *This is an assumption of Sergey Ioffe and Christian Szegedy, they did the experiment in 2015 and find the assumption is true.*
- *But why it works? It's still under discussion and we still don't know for sure.*

Batch normalization

- How to normalize the inputs of a layer for each batch?
  - Given an input value x_i in a batch $(x_1, x_2, \dots, x_n)$
  - The normalized input value $N(x_i) = \frac{x_i - \mu_{batch}}{\sigma_{batch}}$
    - $N(x_i)$ follows the Standard Gaussian Distribution
  - Scaling and shifting after normalization
    - $BN(x_i) = \gamma N(x_i) + \beta$
    - γ and β are two learning parameters of the batch normalization layer
    - If $\gamma = \sigma_{batch}$ and $\beta = \mu_{batch}$, then $BN(x_i) = x_i$
      - This means the batch normalization layer doesn't change the input value
      - If the batch normalization is not beneficial, the network can learn to undo it during the training process

CLASS torch.nn.BatchNorm1d(*num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True, device=None, dtype=None*) [SOURCE]

Hands-on Exercise

- Exercise 05 ANN for Binary Classification – Instruction
- **Exercise 05 ANN for Multi-class Classification - Instruction**
- Exercise 05 ANN for Classification - Assignment