Deep Learning and Applications UEC630

Regularization

By

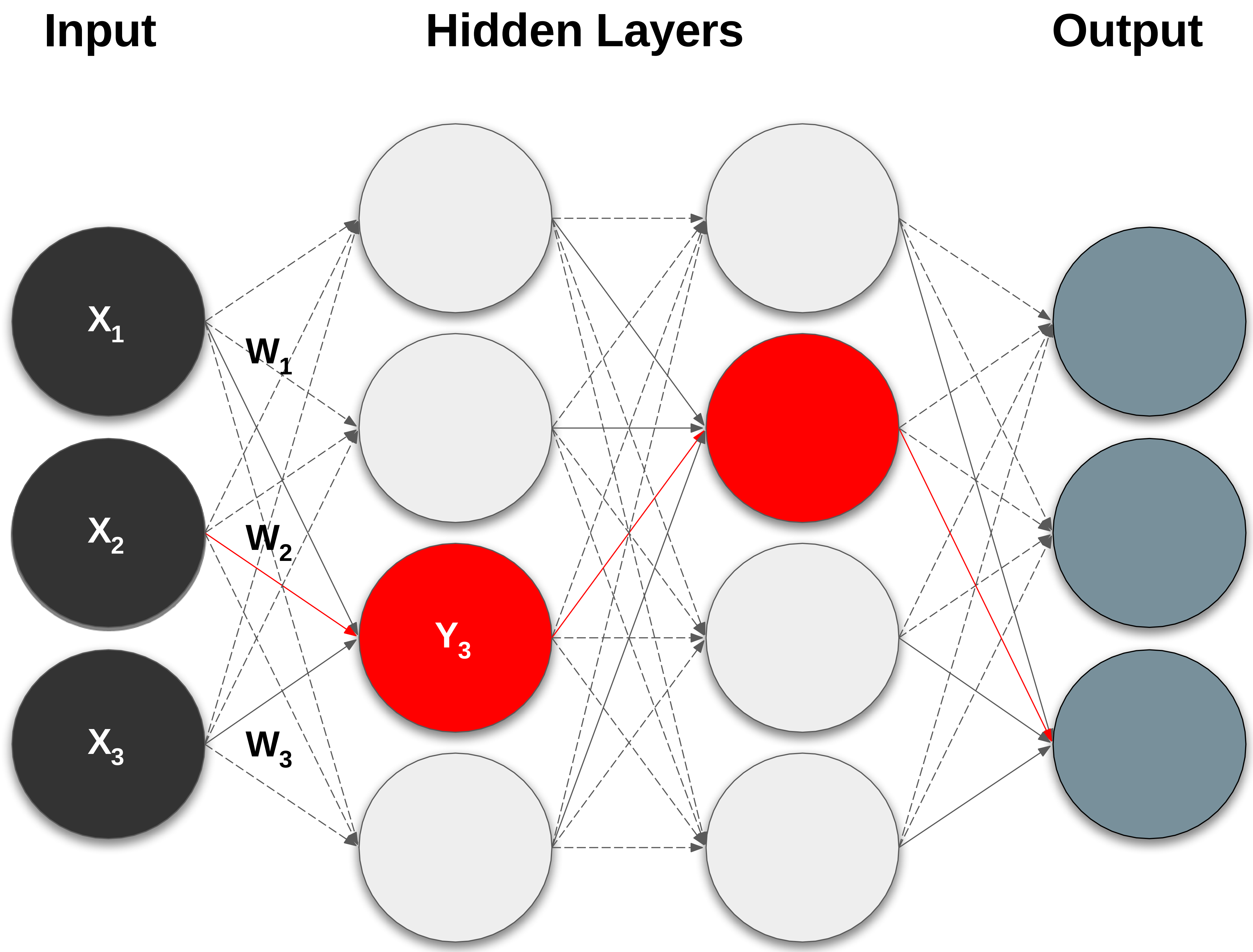
Dr. Ashu

Supervised Deep Learning with Neural Networks

From one layer to the next

$$Y_j = f\left(\sum_i W_i X_i + b_i\right)$$

f **is the activation function,** W_i **is the weight, and** b_i **is the bias.**



Training - Minimizing the Loss

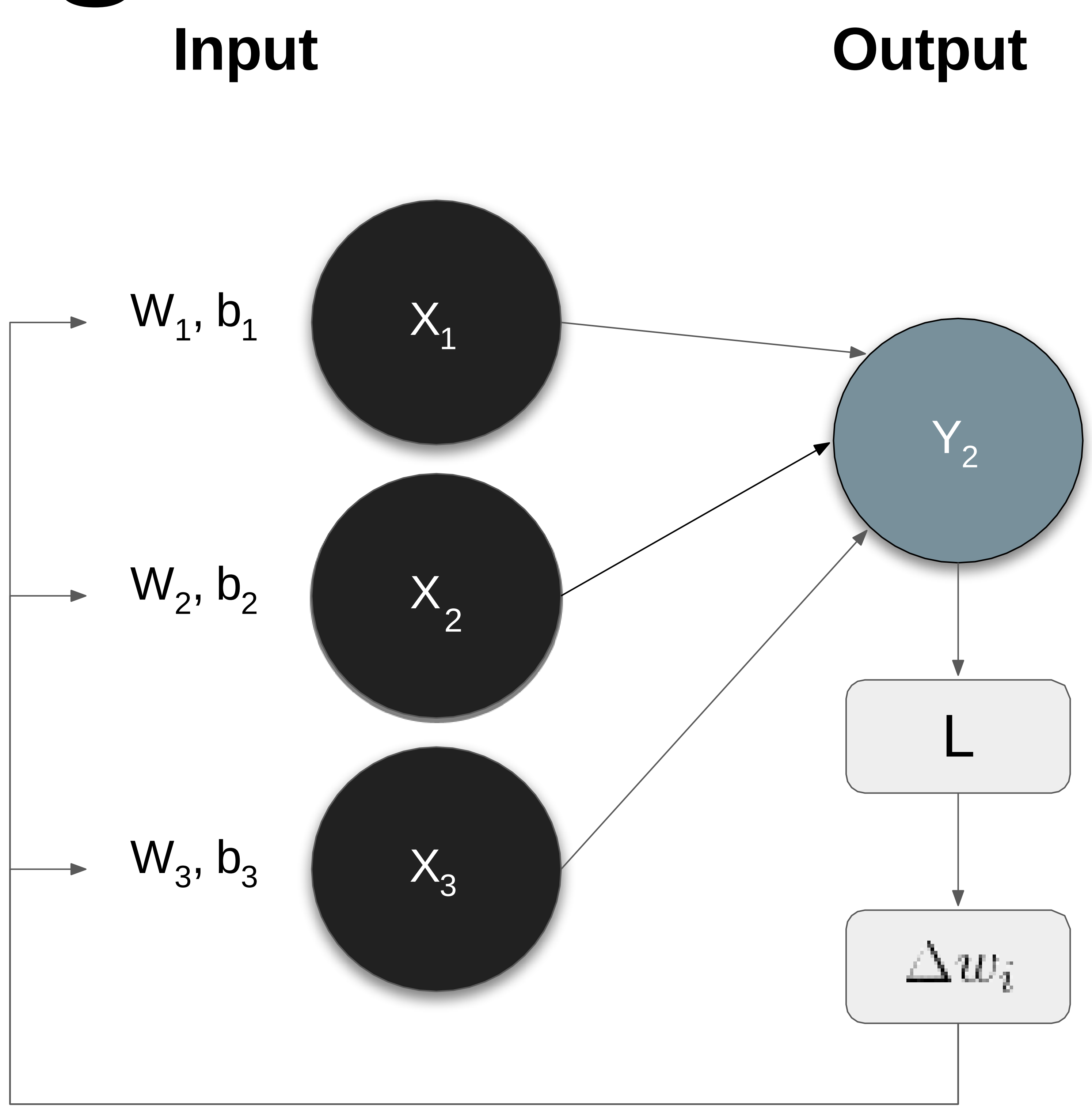
The loss function with regard to weights and biases can be defined as

$$L(\mathbf{w},\mathbf{b}) = \frac{1}{2}\sum_i (\mathbf{Y}(\mathbf{X},\mathbf{w},\mathbf{b}) - \mathbf{Y}'(\mathbf{X},\mathbf{w},\mathbf{b}))^2$$

The weight update is computed by moving a step to the opposite direction of the cost gradient.

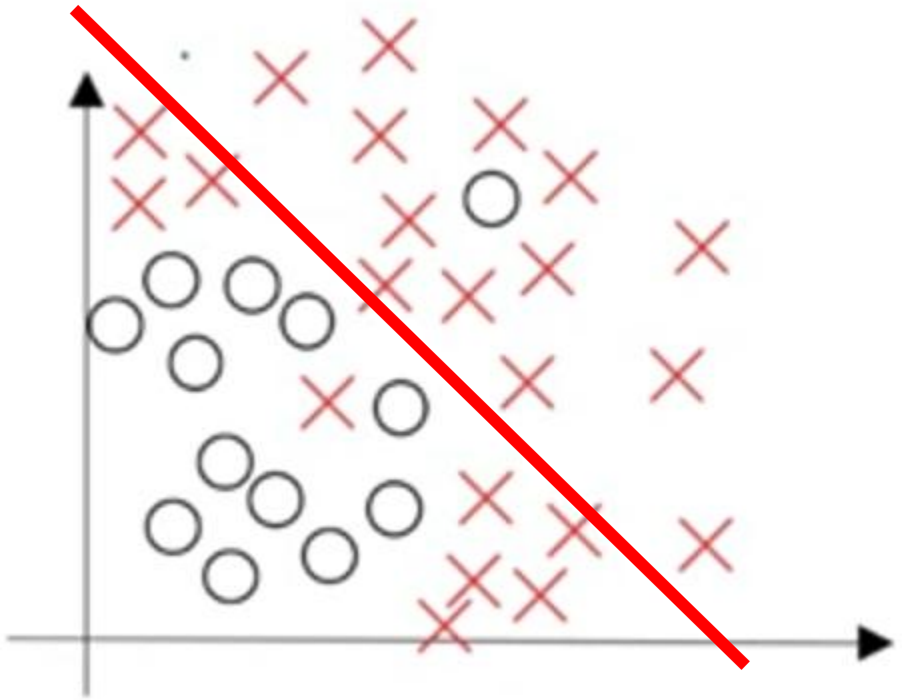
$$\Delta w_i = -\alpha \frac{\partial L}{\partial w_i}$$

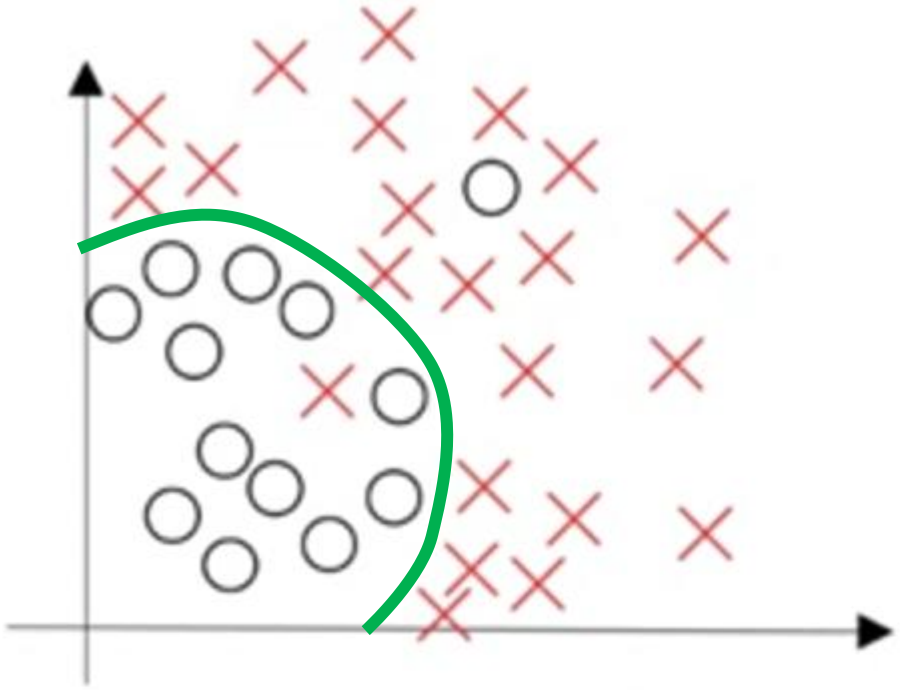
Iterate until L stops decreasing.

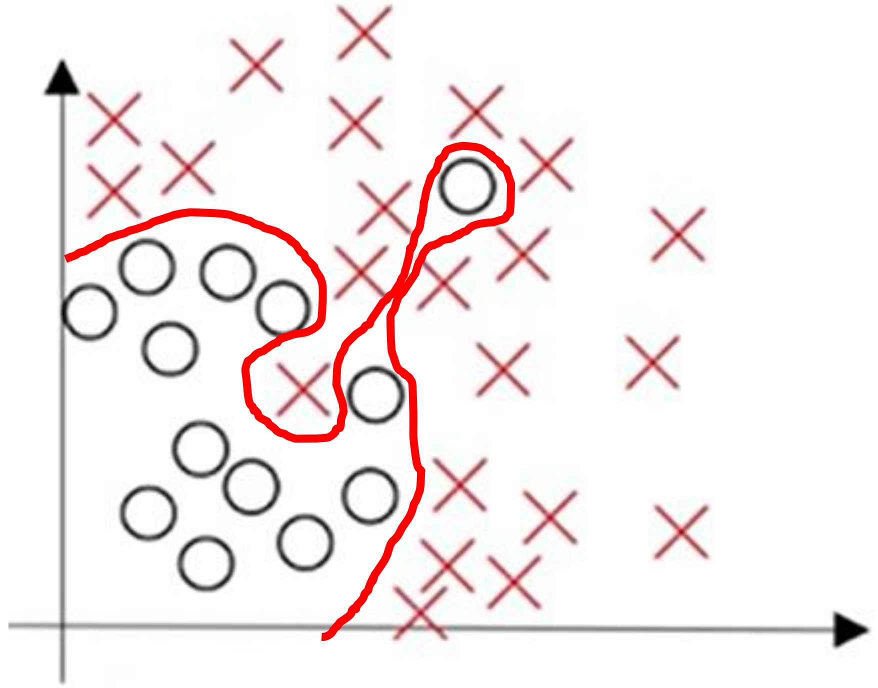


Regularization

- Bias
- Variance







Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



Cat	**3.2**	1.3	2.2
Car	5.1	**4.9**	2.5
Frog	-1.7	2.0	**-3.1**

A **loss function** tells how good our current classifier is

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^{N}$$

Where x_i is image and
y_i is (integer) label

Loss over the dataset is a average of loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

Regularization

$$\underbrace{L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}$$

Data loss: Model predictions should match training data

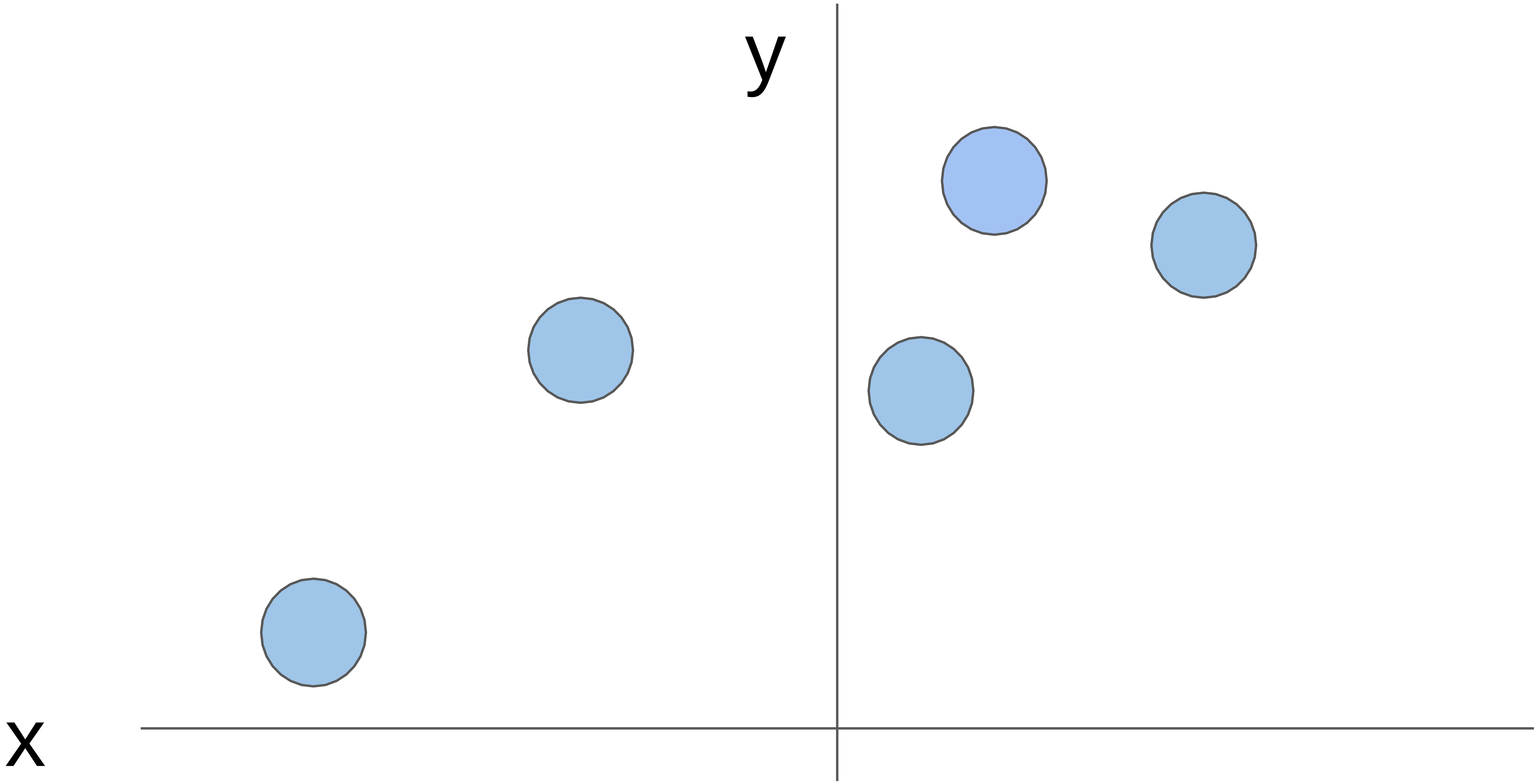
Regularization

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

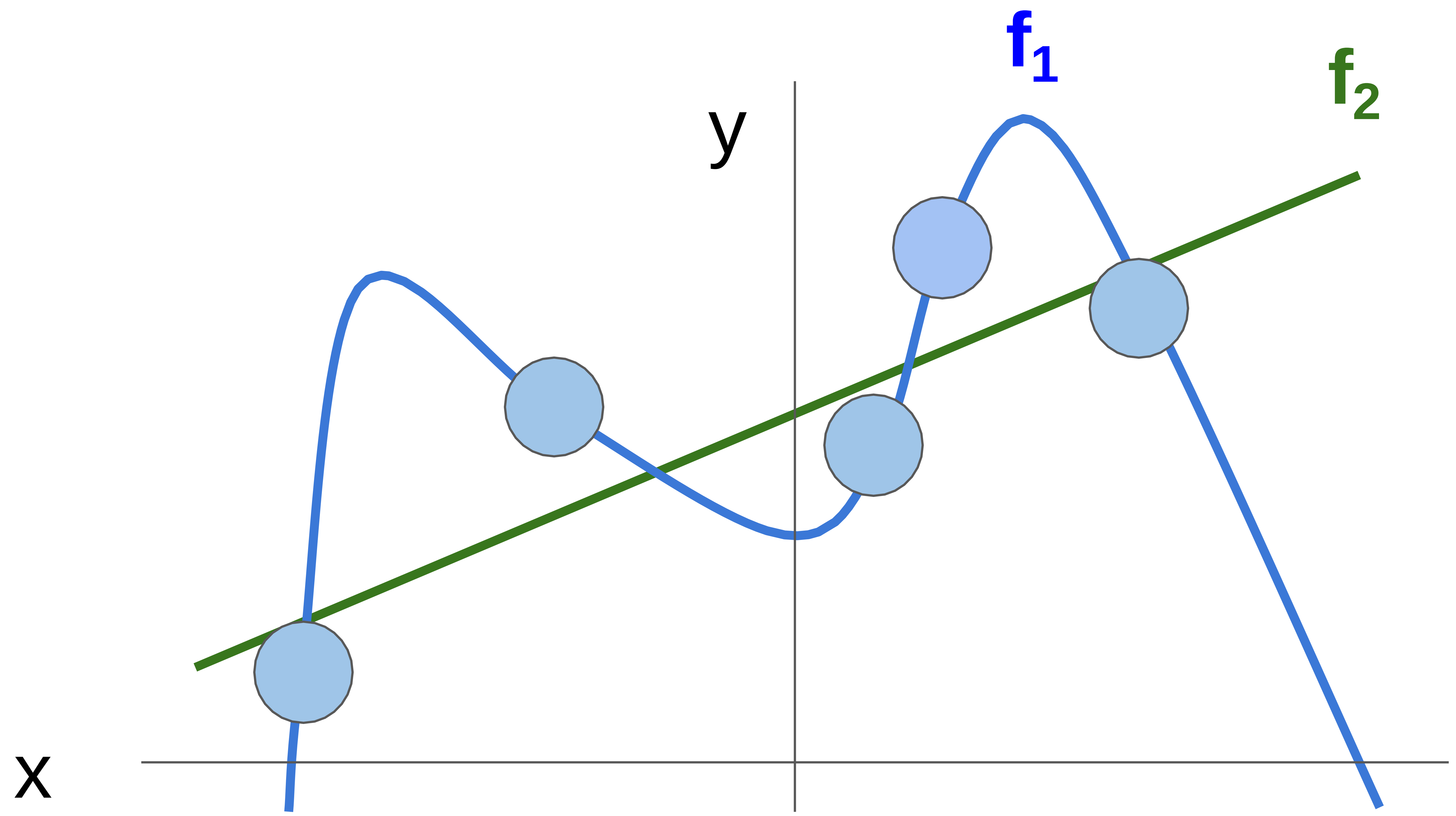
Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

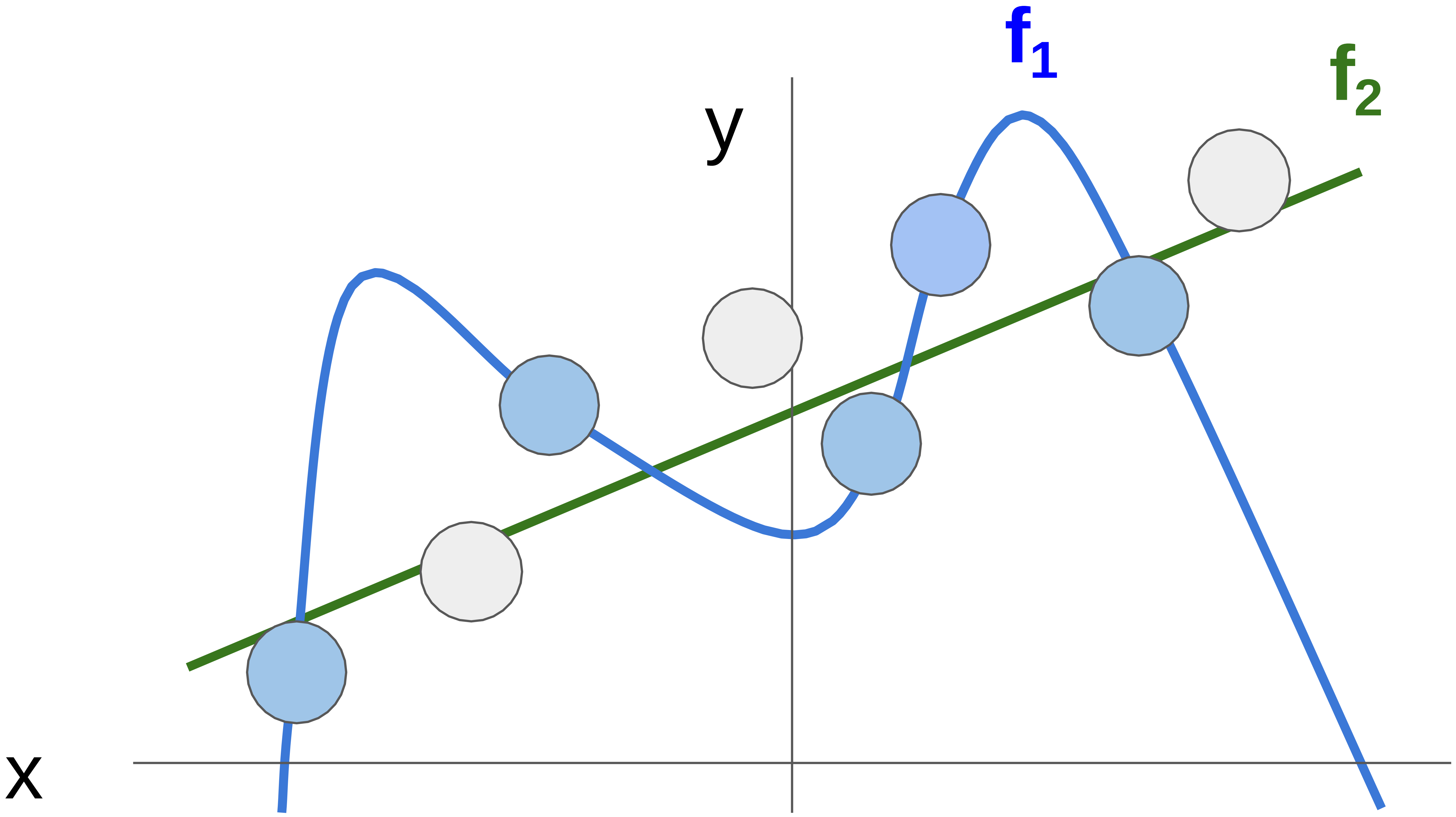
Regularization intuition: toy example training data



Regularization intuition: Prefer Simpler Models



Regularization: Prefer Simpler Models



Regularization pushes against fitting the data *too* well so we don't fit noise in the data

Regularization

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \underbrace{\lambda R(W)}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Occam's Razor: Among multiple competing hypotheses, the simplest is the best, William of Ockham 1285-1347

Regularization

λ = regularization strength (hyperparameter)

$$\underbrace{L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Regularization

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \underbrace{\lambda R(W)}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{} + \underbrace{\lambda R(W)}_{}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

More complex:

Dropout

Batch normalization

Stochastic depth, fractional pooling, etc

Regularization

λ = regularization strength (hyperparameter)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Why regularize?

- Express preferences over weights
- Make the model *simple* so it works on test data
- Improve optimization by adding curvature

Regularization

- L1 Regularization (Lasso Regression) we add a penalty equal to the L1 norm of the weights to the loss function.
- The loss function with L1 regularization is:

$$\text{Loss} = \text{MSE} + \lambda \sum_{i=1}^{n} |w_i|$$

- This can be seen as minimizing the MSE while keeping the sum of the absolute values of the weights below a certain threshold.

Regularization

- L2 Regularization (Ridge Regression) we add a penalty equal to the L2 norm of the weights to the loss function.
- The loss function with L2 regularization is:

$$\text{Loss} = \text{MSE} + \lambda \sum_{i=1}^{n} w_i^2$$

- This can be seen as minimizing the MSE while keeping the sum of the squares values of the weights below a certain threshold.

Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$
$$w_1 = [1, 0, 0, 0]$$
$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Which of w1 or w2 will the L2 regularizer prefer?

Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = \boxed{[0.25, 0.25, 0.25, 0.25]}$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Which of w1 or w2 will the L2 regularizer prefer?

L2 regularization likes to "spread out" the weights

Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = \boxed{[0.25, 0.25, 0.25, 0.25]}$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Which of w1 or w2 will the L2 regularizer prefer?

L2 regularization likes to "spread out" the weights

Which one would L1 regularization prefer?

Regularization

- Problem Statement:
  - Consider a simple neural network with one input feature, one hidden layer with one neuron, and one output neuron. We are given a small dataset and need to perform one step of gradient descent with L2 regularization on the weights.
- Given Data
  - Input feature $x = 2$
  - True output $y = 5$
  - Initial weight from input to hidden layer $w_1 = 0.5$
  - Initial weight from hidden to output layer $w_2 = 0.3$
  - Learning rate $\eta = 0.01$
  - Regularization parameter $\lambda = 0.1$
- Neural Network Structure
  - Input layer: 1 feature
  - Hidden layer: 1 neuron
    - Activation function: Identity (for simplicity)
  - Output layer: 1 neuron
    - Activation function: Identity (for simplicity)

Regularization

- Forward Pass (without regularization)
  1. Calculate hidden layer output:

$$h = x \cdot w_1 = 2 \cdot 0.5 = 1$$

  2. Calculate output:

$$\hat{y} = h \cdot w_2 = 1 \cdot 0.3 = 0.3$$

  3. Compute the loss (Mean Squared Error):

$$L = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(0.3 - 5)^2 = \frac{1}{2} \cdot 22.09 = 11.045$$

Regularization

- Adding L2 Regularization
  - Regularized Loss Function:

$$L_{\text{reg}} = L + \frac{\lambda}{2}(w_1^2 + w_2^2)$$

  - Calculating the regularized loss:

$$L_{\text{reg}} = 11.045 + \frac{0.1}{2}(0.5^2 + 0.3^2)$$

$$= 11.045 + 0.005 \cdot (0.25 + 0.09) = 11.045 + 0.005 \cdot 0.34$$

$$= 11.045 + 0.0017 = 11.0467$$

Regularization

- Backward Pass (Gradient Calculation)
  - 1. Gradient of the loss (before regularization):

$$\frac{\partial L}{\partial w_2} = (\hat{y} - y) \cdot h = (0.3 - 5) \cdot 1 = -4.7$$

  - Including L2 regularization term:

$$\frac{\partial L_{\text{reg}}}{\partial w_2} = \frac{\partial L}{\partial w_2} + \lambda w_2 = -4.7 + 0.1 \cdot 0.3 = -4.7 + 0.03 = -4.67$$

  - Gradient of the loss (before regularization):

$$\frac{\partial L}{\partial w_1} = (\hat{y} - y) \cdot w_2 \cdot x = (0.3 - 5) \cdot 0.3 \cdot 2 = -4.7 \cdot 0.3 \cdot 2 = -2.82$$

  - Including L2 regularization term:

$$\frac{\partial L_{\text{reg}}}{\partial w_1} = \frac{\partial L}{\partial w_1} + \lambda w_1 = -2.82 + 0.1 \cdot 0.5 = -2.82 + 0.05 = -2.77$$

Regularization

- Gradient Descent Step
  1. Update w_1:

$$w_2 := w_2 - \eta \cdot \frac{\partial L_{\text{reg}}}{\partial w_2} = 0.3 - 0.01 \cdot (-4.67) = 0.3 + 0.0467 = 0.3467$$

  2. Update w_2 :

$$w_1 := w_1 - \eta \cdot \frac{\partial L_{\text{reg}}}{\partial w_1} = 0.5 - 0.01 \cdot (-2.77) = 0.5 + 0.0277 = 0.5277$$

- Updated Weights
  - $w_1 = 0.5277$
  - $w_2 = 0.3467$

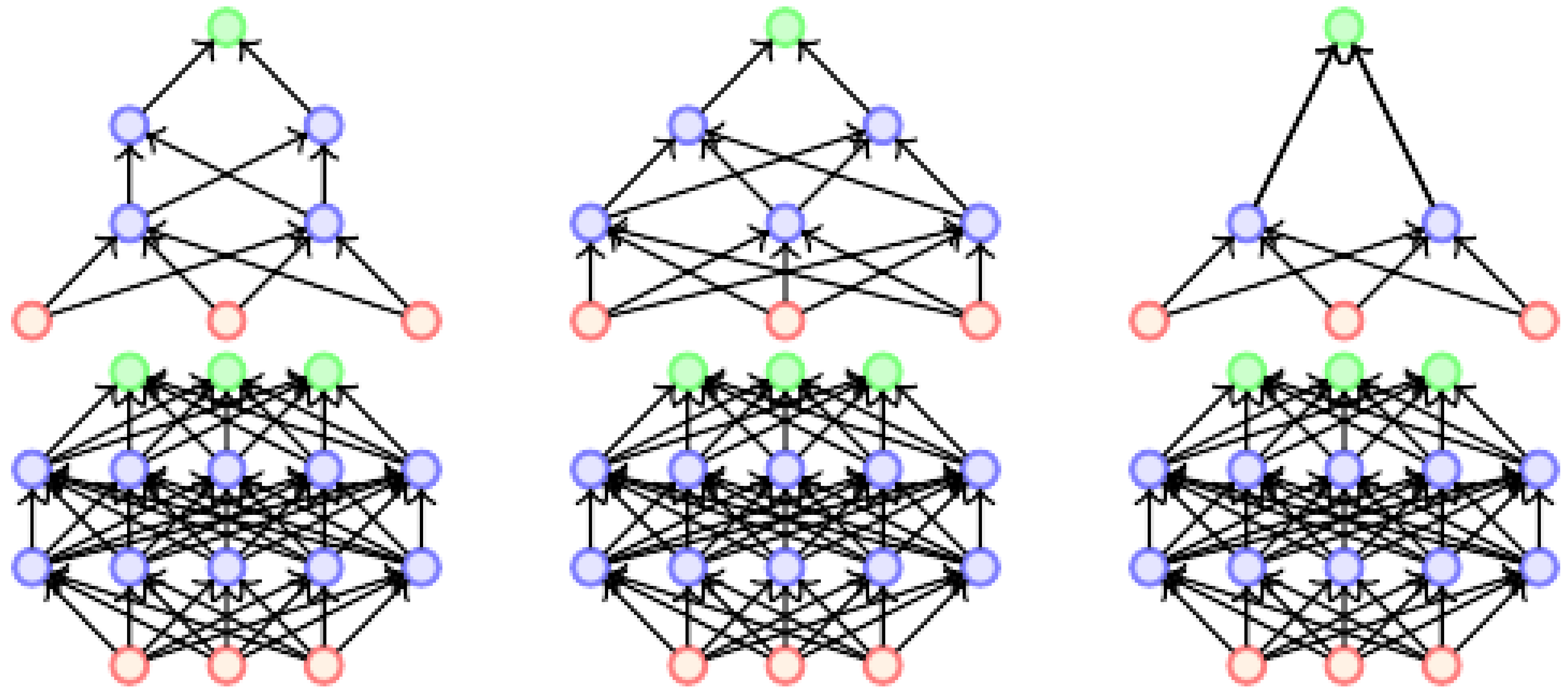
Regularization

- Techniques for Regularization:
  - Norm Penalties (L1/L2 Norm)
  - Dropout
  - Data Augmentation
  - Parameter Sharing and tying
  - Early stopping
  - Batch Normalization

Dropout

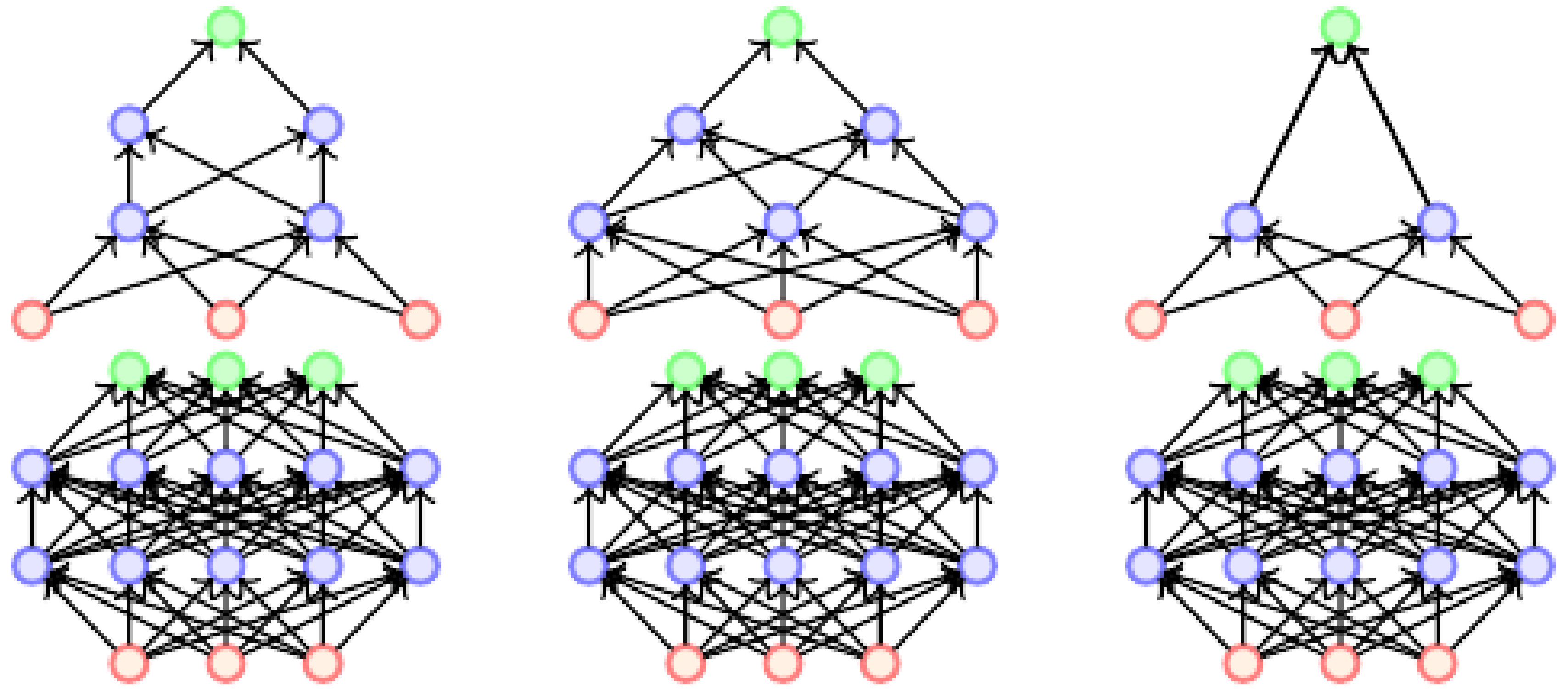
- Randomly select weights to update
- More precisely, in each update step
  - Randomly sample a different binary mask to all the input and hidden units
  - Multiple the mask bits with the units and do the update as usual
- Typical dropout probability: 0.2 for input and 0.5 for hidden units

Dropout



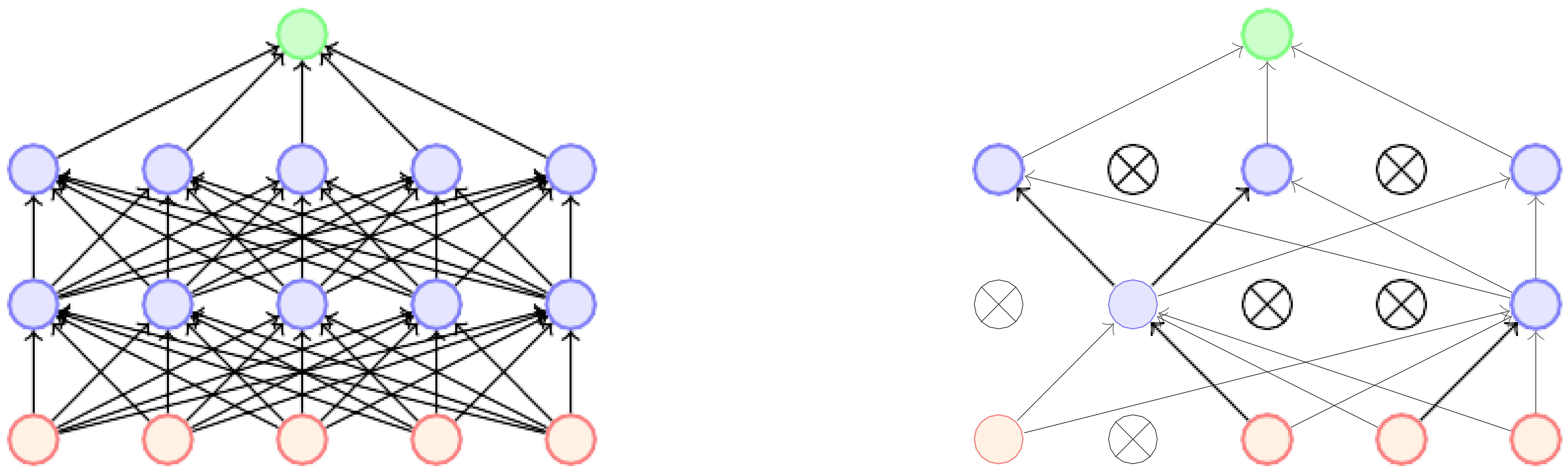
- Typically model averaging(bagging ensemble) always helps
- Training several large neural net- works for making an ensemble is pro- hibitively expensive
- Option 1: Train several neural networks having different architec- tures(obviously expensive)
- Option 2: Train multiple instances of the same network using different training samples (again expensive)
- Even if we manage to train with op- tion 1 or option 2, combining several models at test time is infeasible in real time applications

Dropout



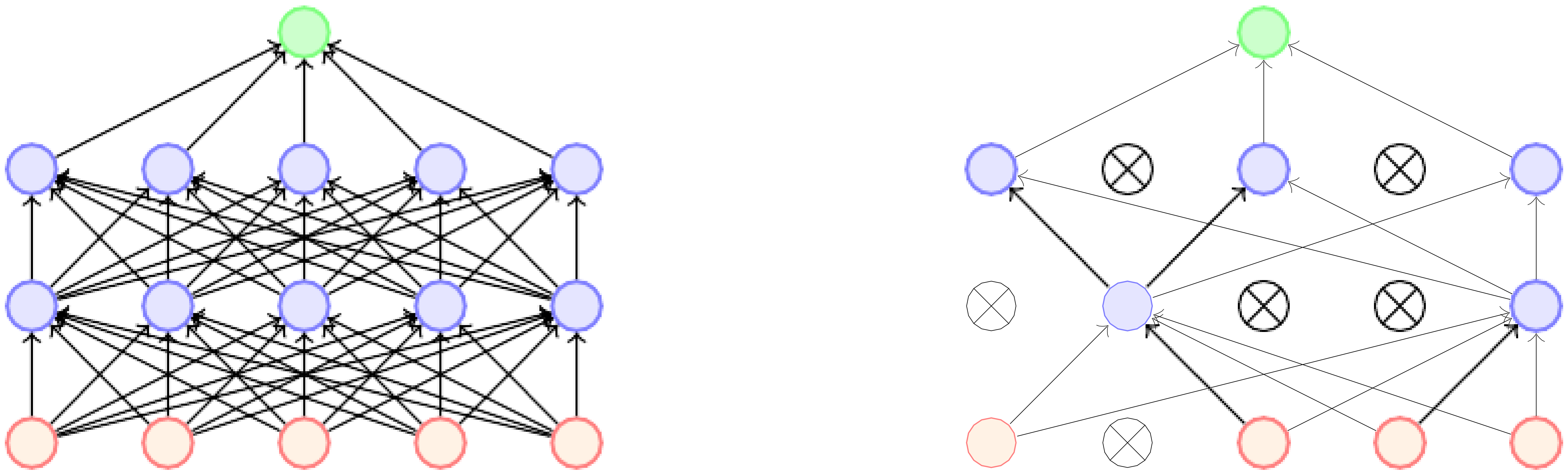
- Dropout is a technique which addresses both these issues.
- Effectively it allows training several neural networks without any significant computational overhead.
- Also gives an efficient approximate way of combining exponentially many different neural networks.

Dropout



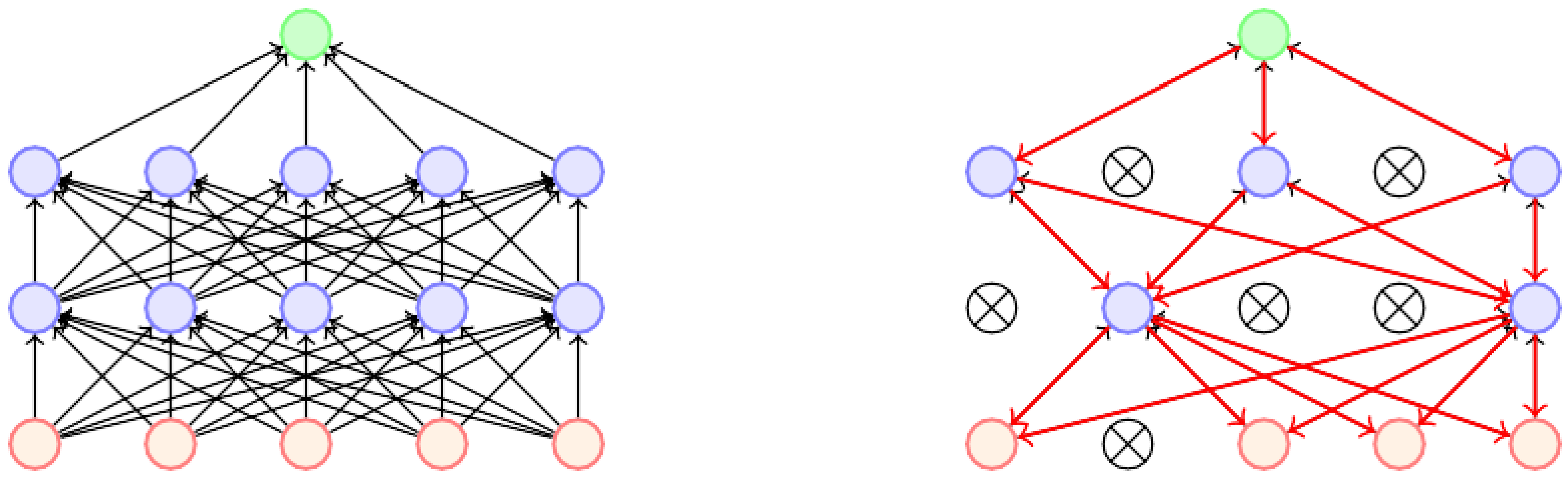
- Dropout refers to dropping out units
- Temporarily remove a node and all its incoming/outgoing connections resulting in a thinned network
- Each node is retained with a fixed probability (typically $p = 0.5$) for hidden nodes and $p = 0.8$ for visible nodes

Dropout



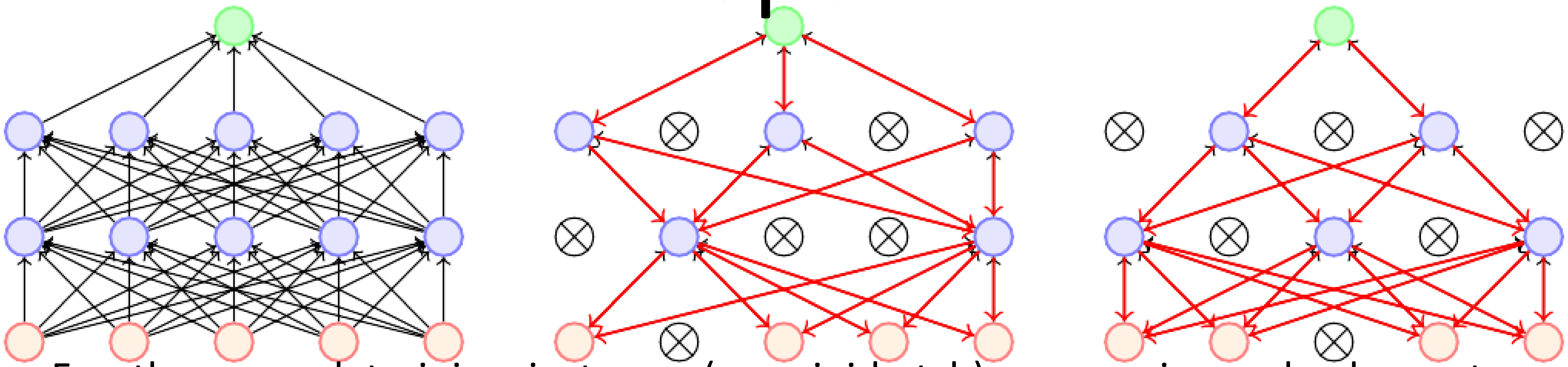
- Suppose a neural network has n nodes
- Using the dropout idea, each node can be retained or dropped
- For example, in the above case we drop 5 nodes to get a thinned network Given a total of n nodes, what are the total number of thinned networks that can be formed? 2^n
- Of course, this is prohibitively large and we cannot possibly train so many networks
- **Trick:** (1) Share the weights across all the networks
  (2) Sample a different network for each training instance
- Let us see how?

Dropout



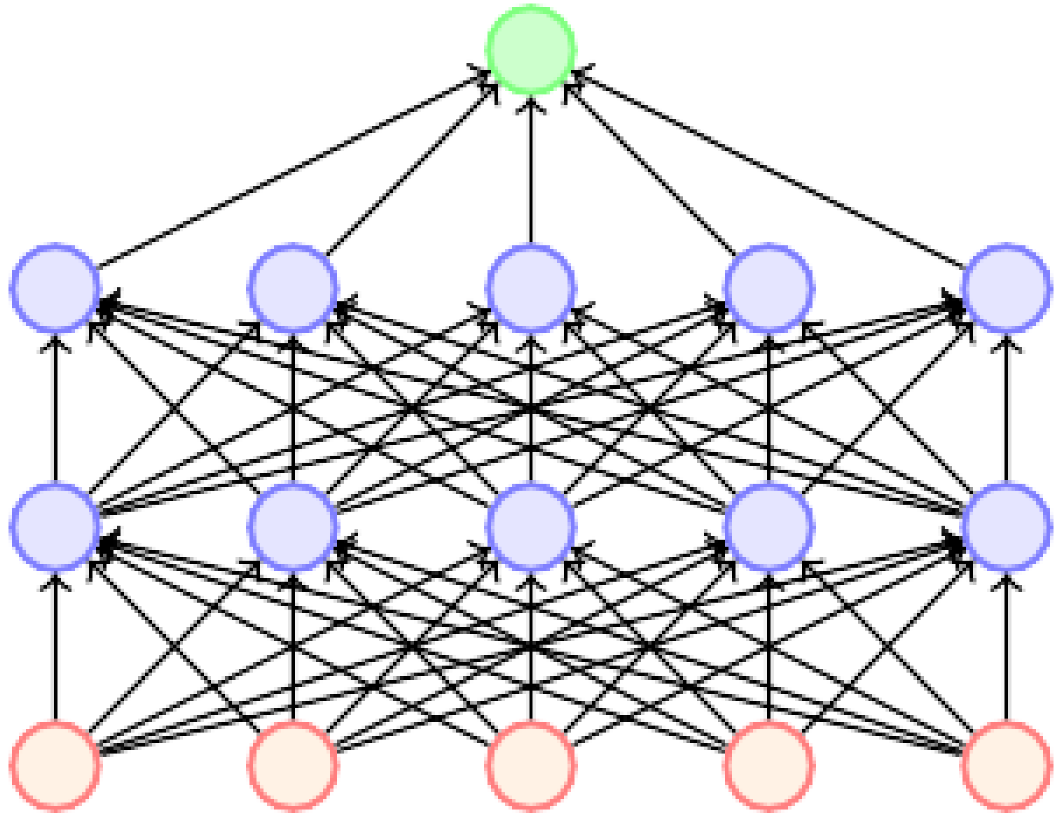
- We initialize all the parameters (weights) of the network and start training
- For the first training instance (or mini-batch), we apply dropout resulting in the thinned network
- We compute the loss and backpropagate
- Which parameters will we update? Only those which are active

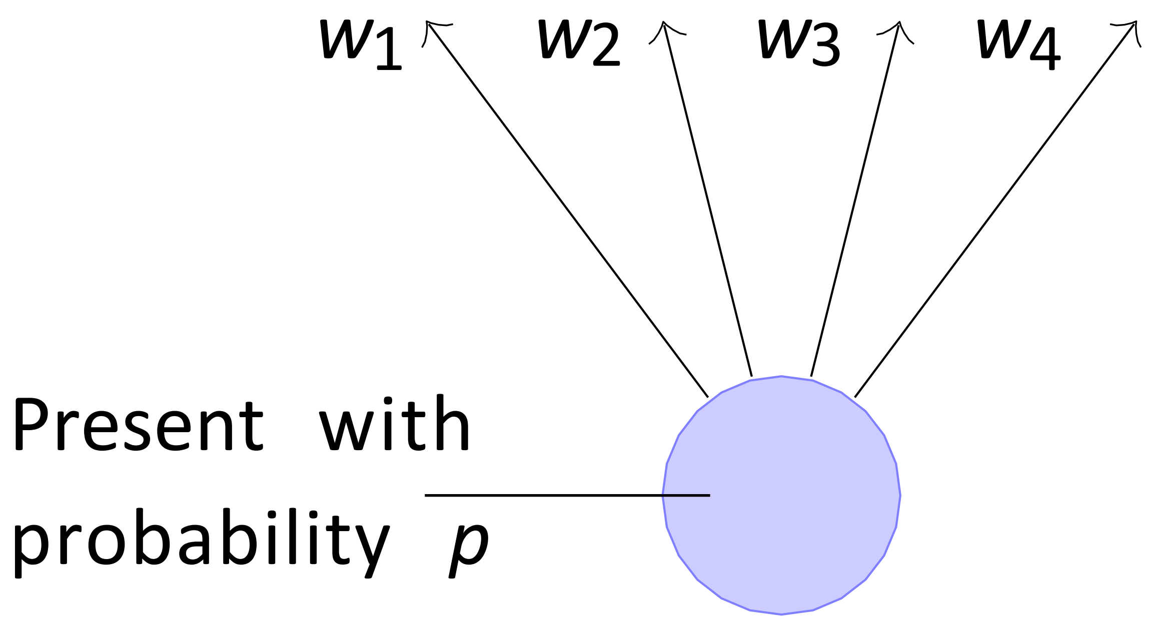
Dropout



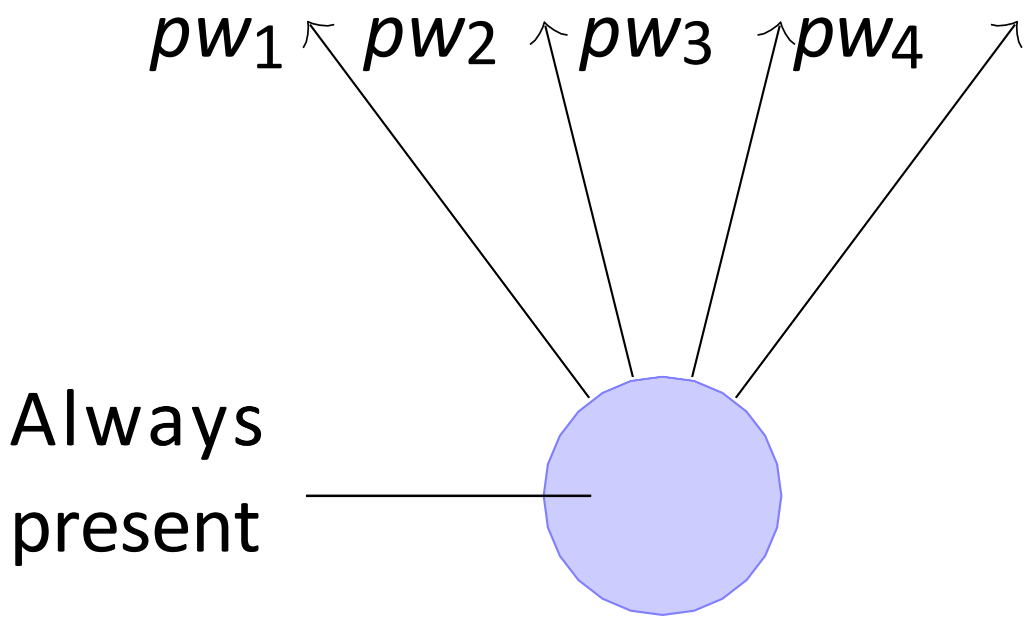
- For the second training instance (or mini-batch), we again apply dropout resulting in a different thinned network
- We again compute the loss and backpropagate to the active weights
- If the weight was active for both the training instances then it would have received two updates by now
- If the weight was active for only one of the training instances then it would have received only one updates by now
- Each thinned network gets trained rarely (or even never) but the parameter sharing ensures that no model has untrained or poorly trained parameters

Dropout





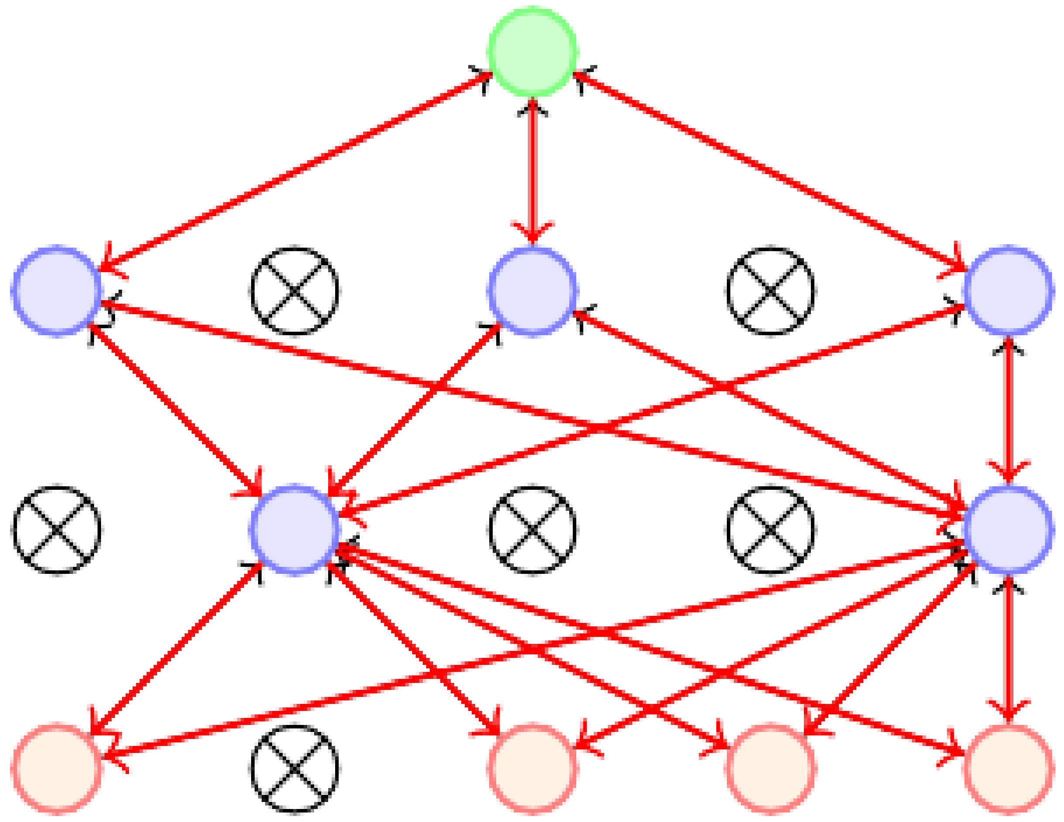
At training time



At test time

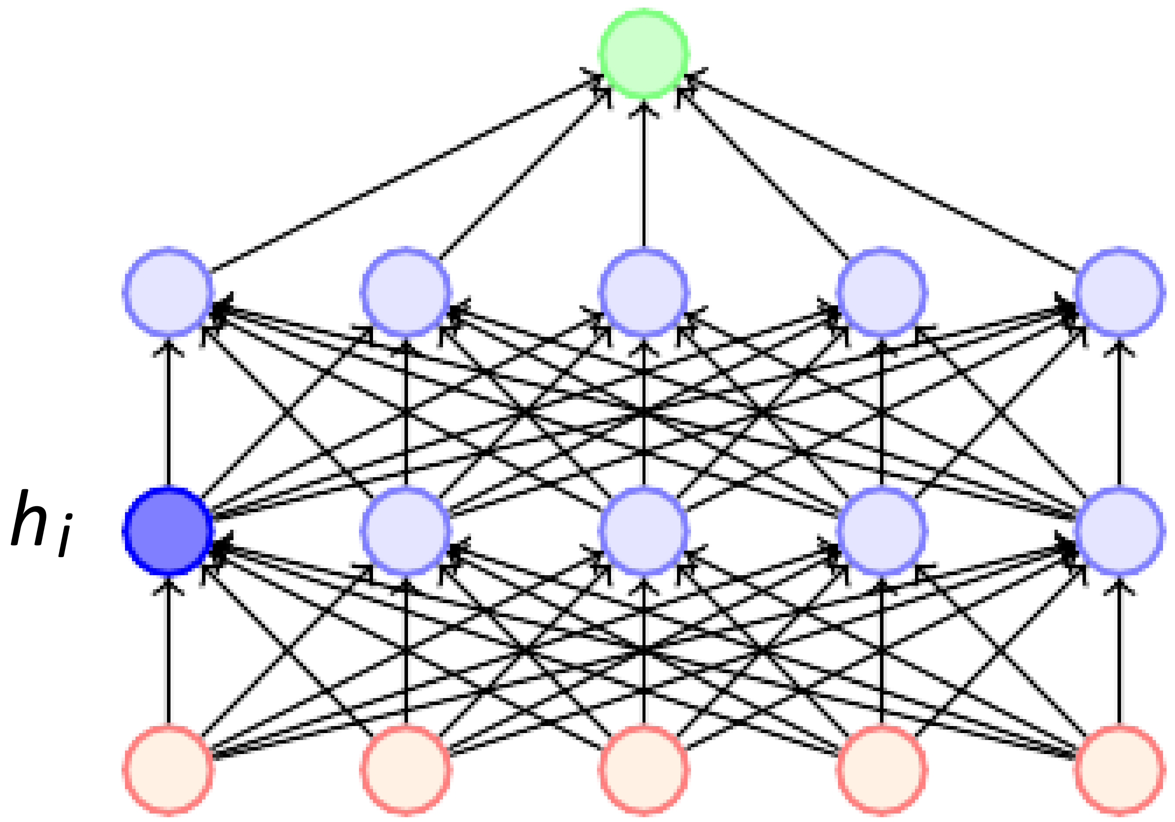
- What happens at test time?
- Impossible to aggregate the outputs of 2^n thinned networks
- Instead we use the full Neural Network and scale the output of each node by the fraction of times it was on during training

Dropout



- Dropout essentially applies a masking noise to the hidden units
- Prevents hidden units from co-adapting
- Essentially a hidden unit cannot rely too much on other units as they may get dropped out any time
- Each hidden unit has to learn to be more robust to these random dro-pouts

Dropout

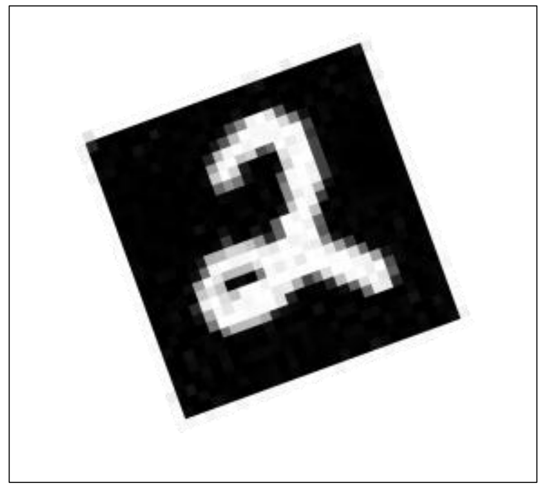


- Here is an example of how dropout helps in ensuring redundancy and robustness
- Suppose h_i learns to detect a face by firing on detecting a nose
- Dropping h_i then corresponds to erasing the information that a nose exists
- The model should then learn another h_i which redundantly encodes the presence of a nose
- Or the model should learn to detect the face using other features

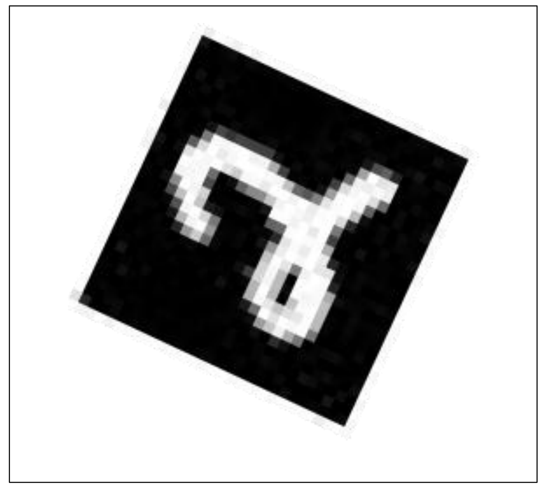
Data Augmentation

label = 2

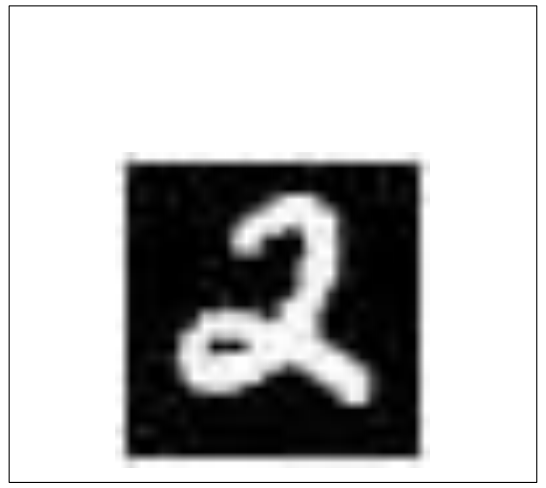
[given training data]
We exploit the fact that certain transformations to the image do not change the label of the image.



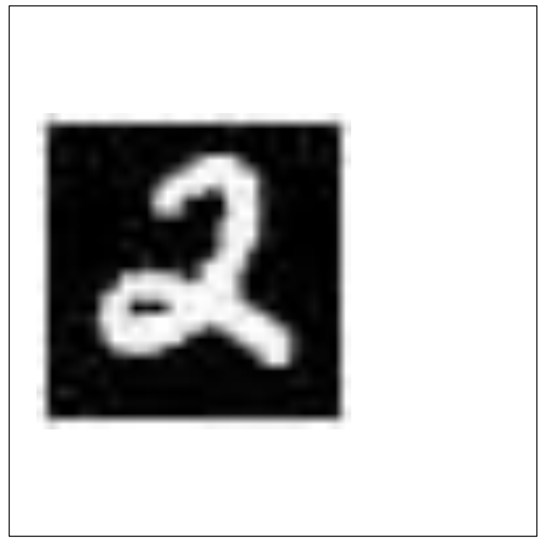
rotated by 20°



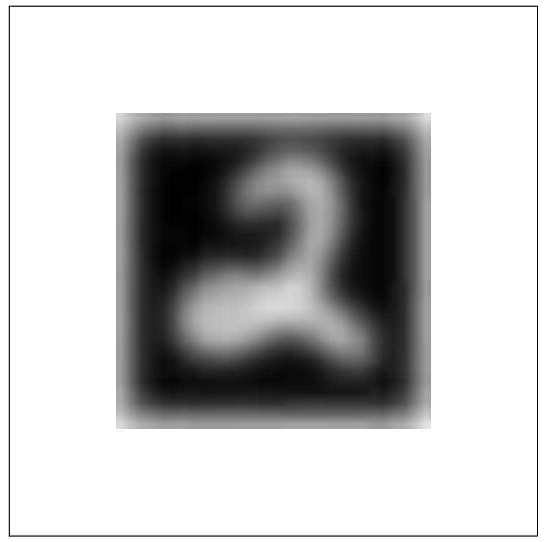
rotated by 65°



shifted vertically



shifted horizontally



blurred

changed some pixels

label = 2

[augmented data = created using some knowledge of the task]

Data Augmentation

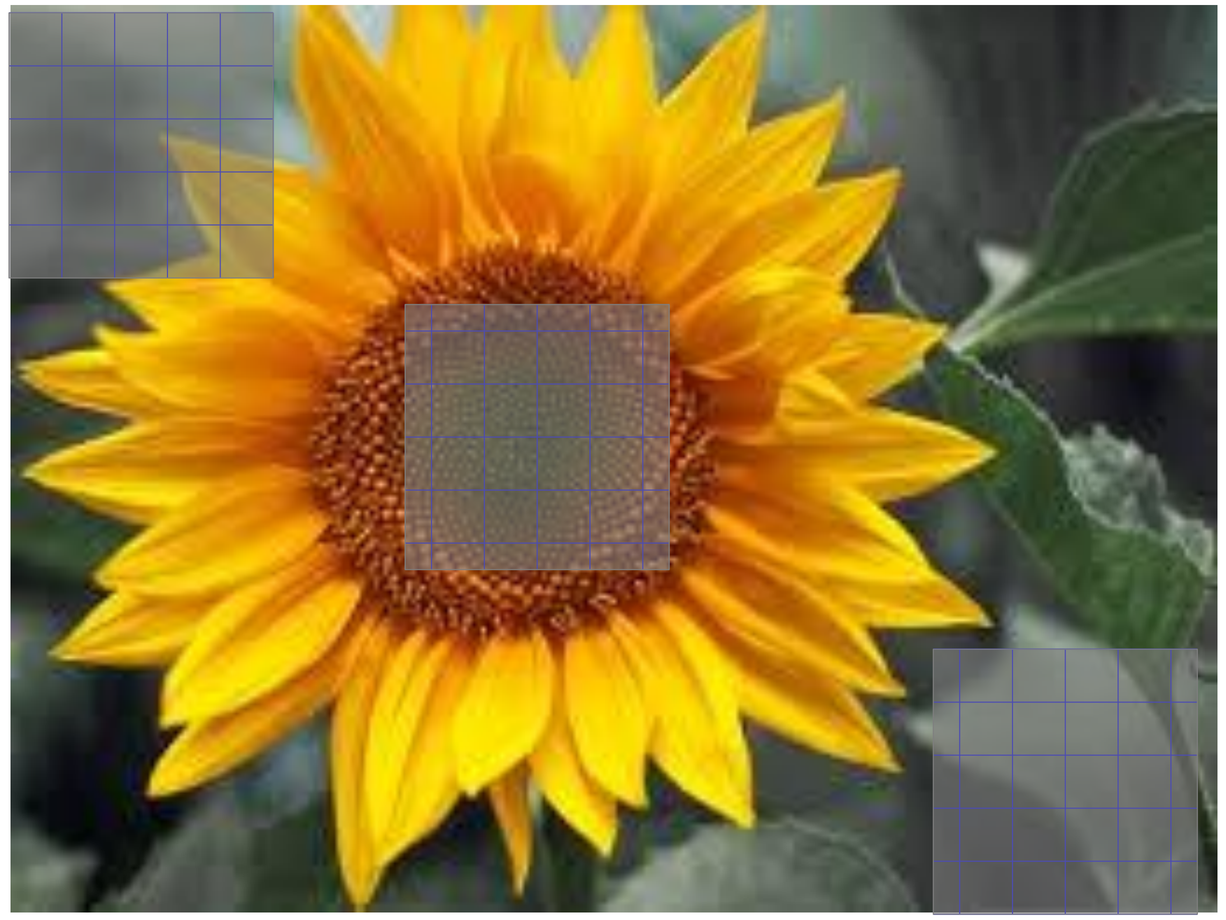
- Typically, More data = better learning
- Works well for image classification / object recognition tasks
- Also shown to work well for speech
- For some tasks it may not be clear how to generate such data

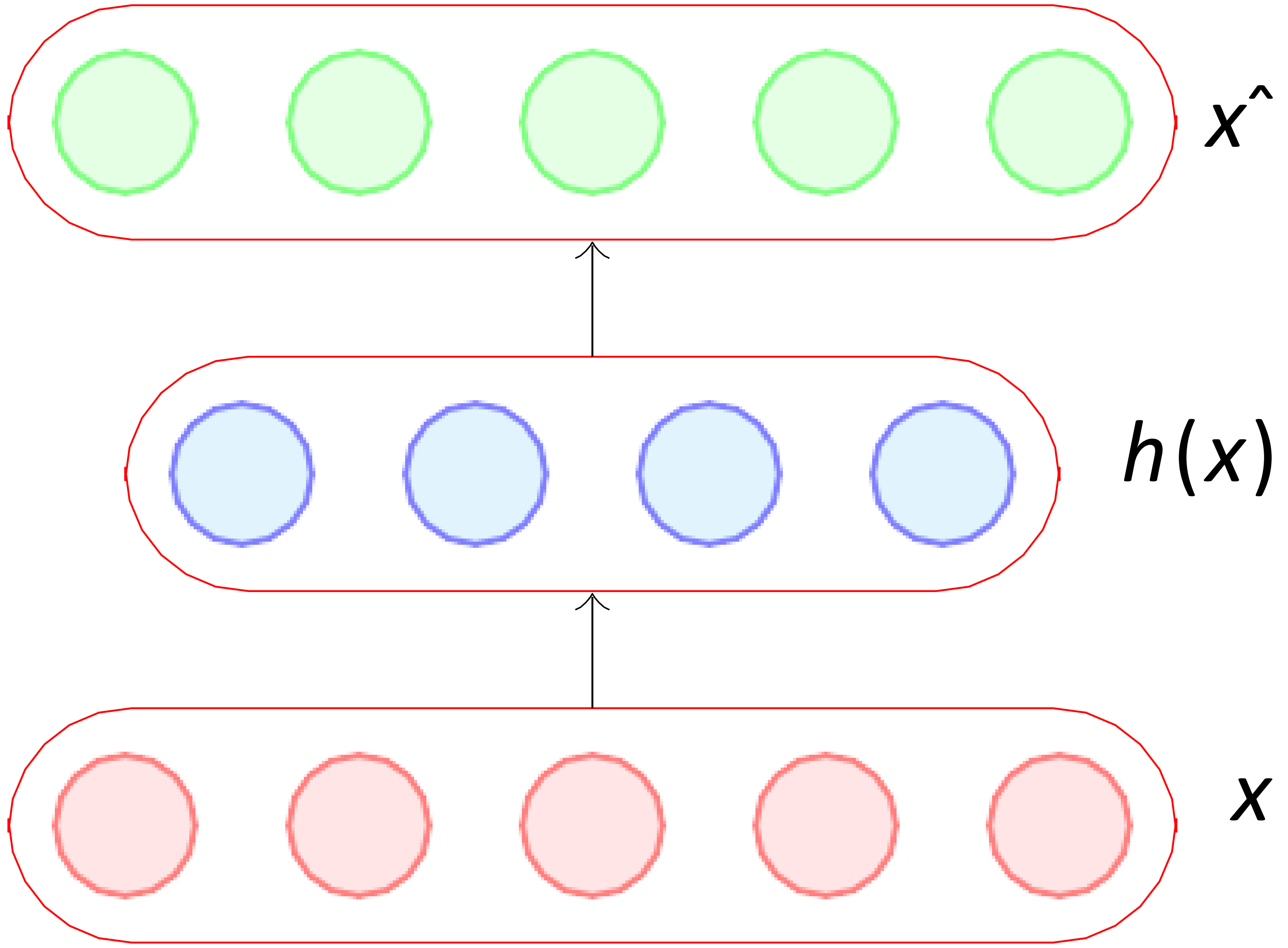
Data Augmentation

- Adding noise to the input: a special kind of augmentation

- Be careful about the transformation applied:
  - Example: classifying ‘b’ and ‘d’
  - Example: classifying ‘6’ and ‘9’

Parameter Sharing and tying





Parameter Sharing

- Used in CNNs
- Same filter applied at different positions of the image
- Or same weight matrix acts on different input neurons

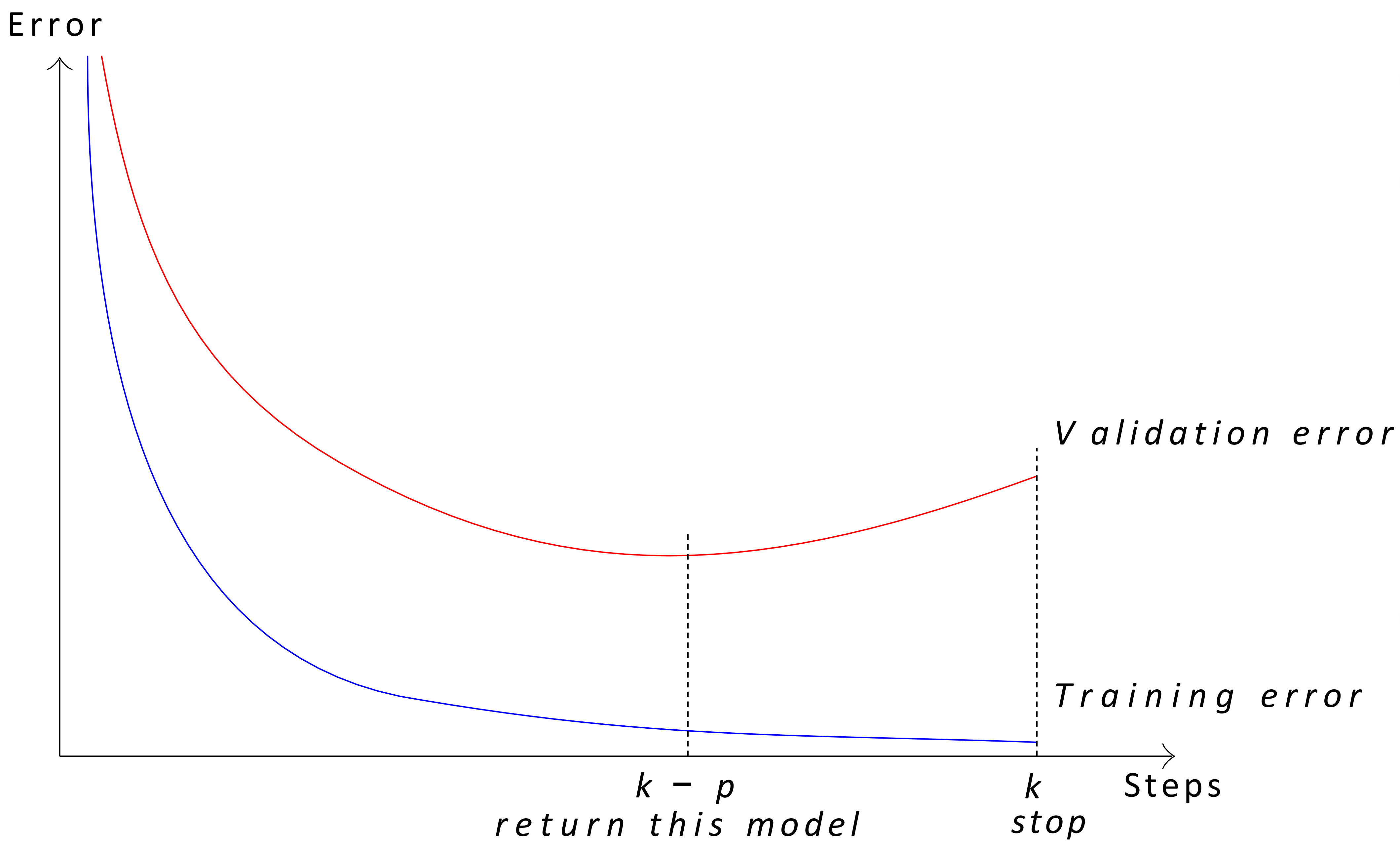
Parameter Tying

- Typically used in autoencoders
- The encoder and decoder weights are tied.

Early stopping

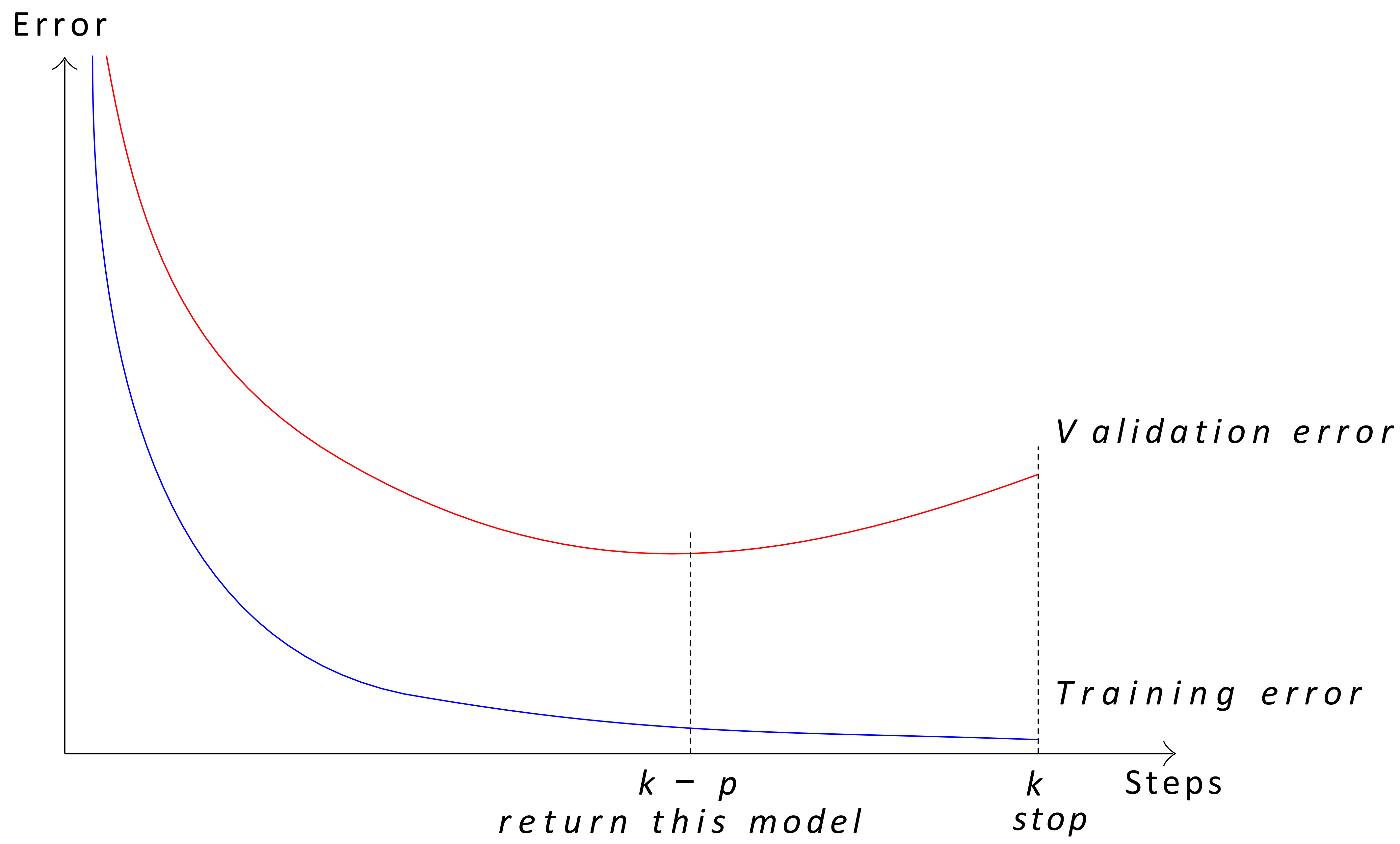
- **Idea:** don’t train the network to too small training error
- Recall overfitting: Larger the hypothesis class, easier to find a hypothesis that fits the difference between the two
- Prevent overfitting: do not push the hypothesis too much; use validation error to decide when to stop

Early stopping



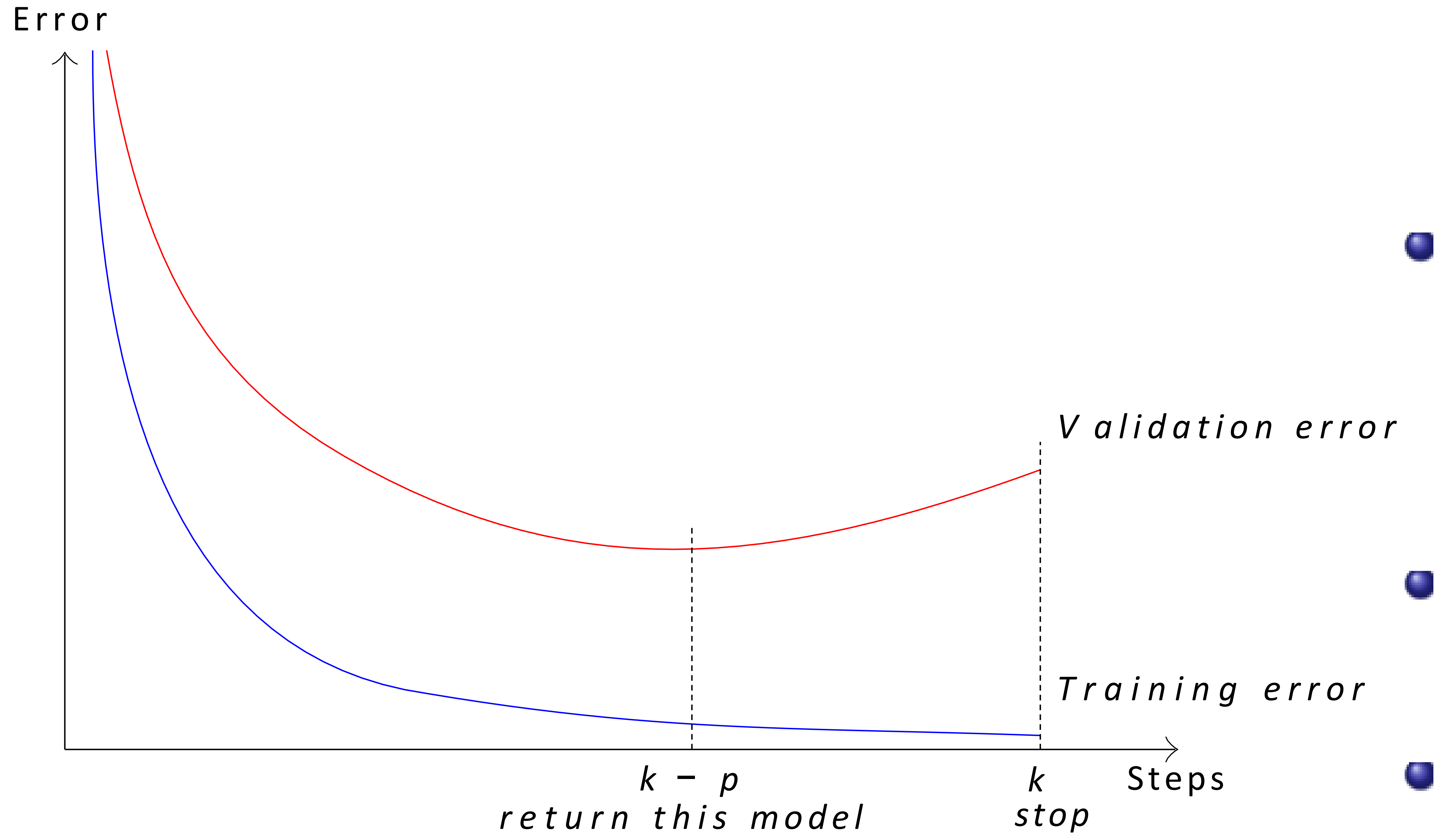
- Track the validation error
- Have a patience parameter p
- If you are at step k and there was no improvement in validation error in the previous p steps then stop training and return the model stored at step $k - p$
- Basically, stop the training early before it drives the training error to 0 and blows up the validation error

Early stopping



- Very effective and the mostly widely used form of regularization
- Can be used even with other regularizers (such as l_2)
- How does it act as a regularizer ?
- We will first see an intuitive explanation and then a mathematical analysis

Early stopping



- Recall that the update rule in SGD (Stochastic Gradient Descen) is

$$w_{t+1} = w_t - \eta \nabla w_t$$
$$= w_0 - \eta \sum_{i=1}^{t} \nabla w_i$$

- Let τ be the maximum value of ∇w_i then

$$|w_{t+1} - w_0| \leq \eta t |\tau|$$

- Thus, t controls how far w_t can go from the initial w_0
- In other words it controls the space of exploration

Early stopping

- When training, also output validation error
- Every time validation error improved, store a copy of the weights
- When validation error not improved for some time, stop
- Return the copy of the weights stored

Early stopping

- hyperparameter selection: training step is the hyperparameter

- **Advantage**
  - Efficient: along with training; only store an extra copy of weights
  - Simple: no change to the model/ algo

- **Disadvantage:** need validation data

Batch Normalization

- If outputs of earlier layers are uniform or change greatly on one round for one mini batch, then neurons at next levels can’t keep up: they output all high (or all low) values

- Next layer doesn’t have ability to change its outputs with learning rate sized changes to its input weights

- We say the layer has “saturated”

Batch Normalization

- Another View of the problem:
  - In ML, we assume future data will be drawn from same probability distribution as training data.
  - For a hidden unit, after training, the earlier layers have new weights and hence generate input data for this hidden unit from a ***new*** distribution.
  - Want to reduce this ***internal covariate shift*** for the benefit of later layers.