

Training Neural Networks

Generalization

M. Soleymani

Sharif University of Technology

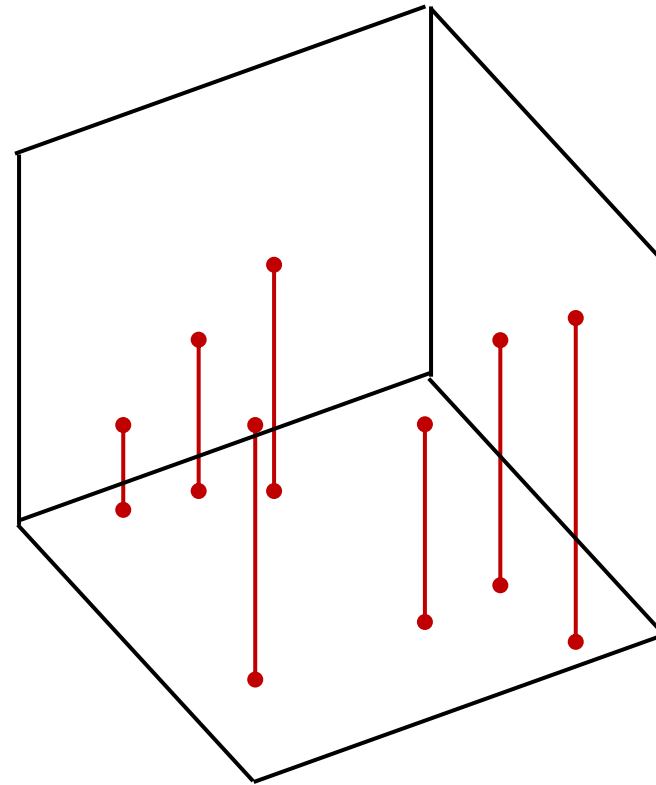
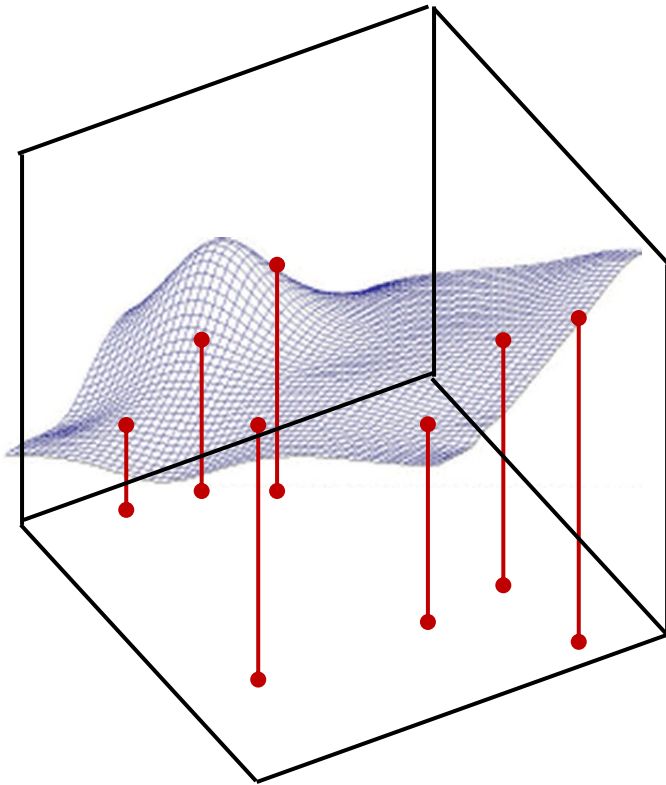
Spring 2026

Most slides have been adapted from Bhiksha Raj, 11-785, CMU
and some from Fei Fei Li et. al, cs231n, Stanford

Outline

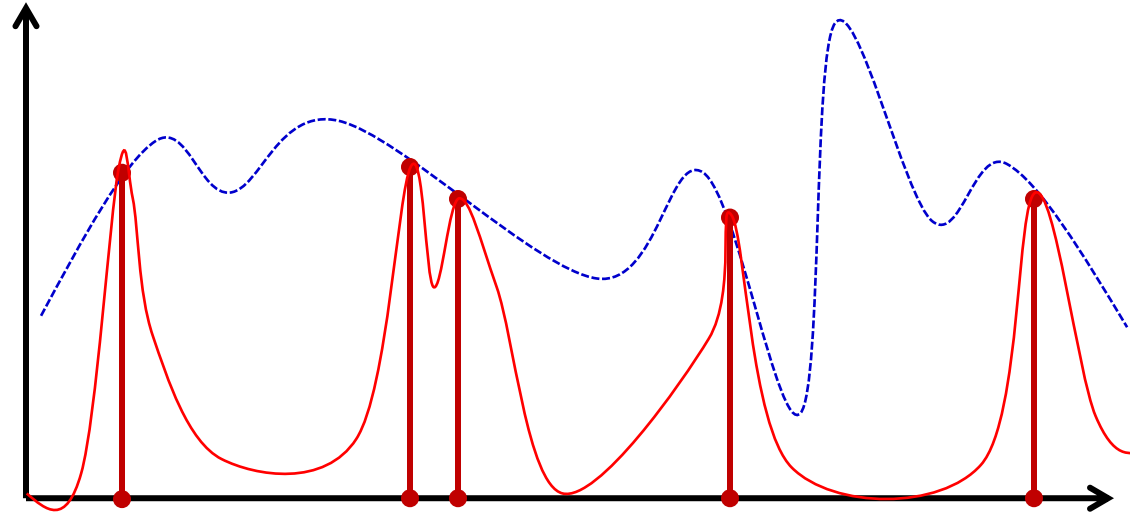
- Generalization
- Regularization
- Early stopping
- Model ensembles
- Drop-out
- Data Augmentation

Learning the network



- We attempt to learn an entire function from just a few *snapshots* of the target function

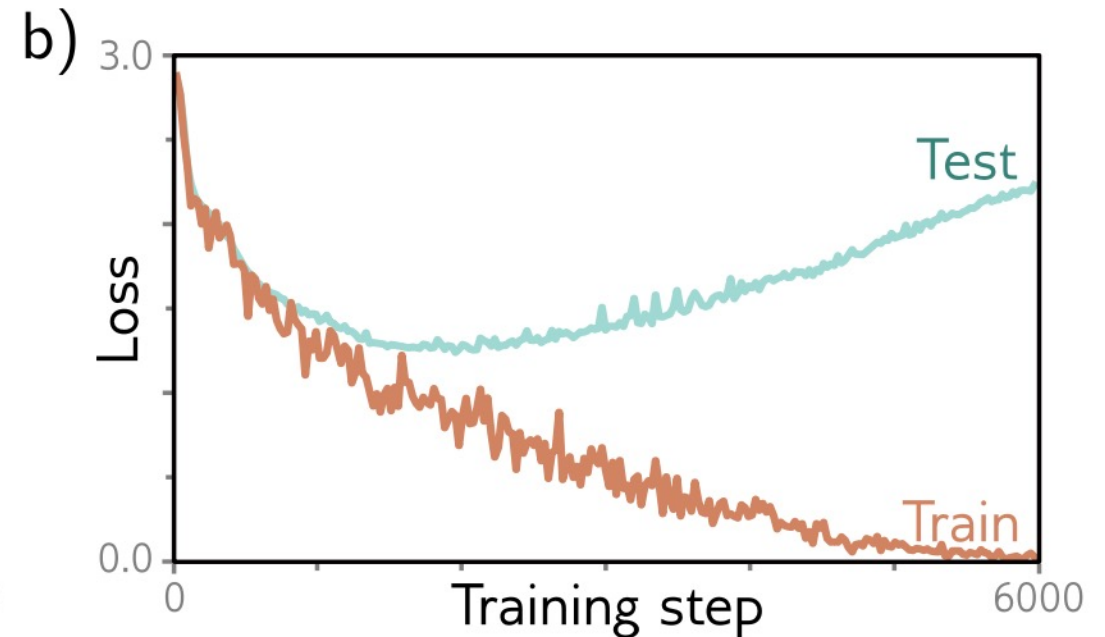
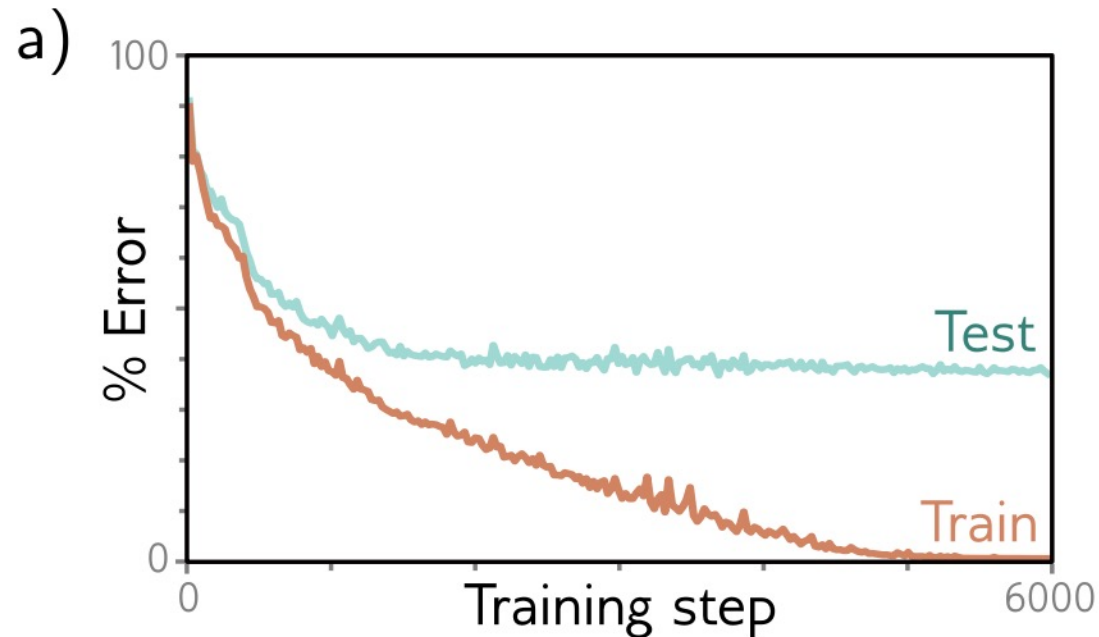
Overfitting



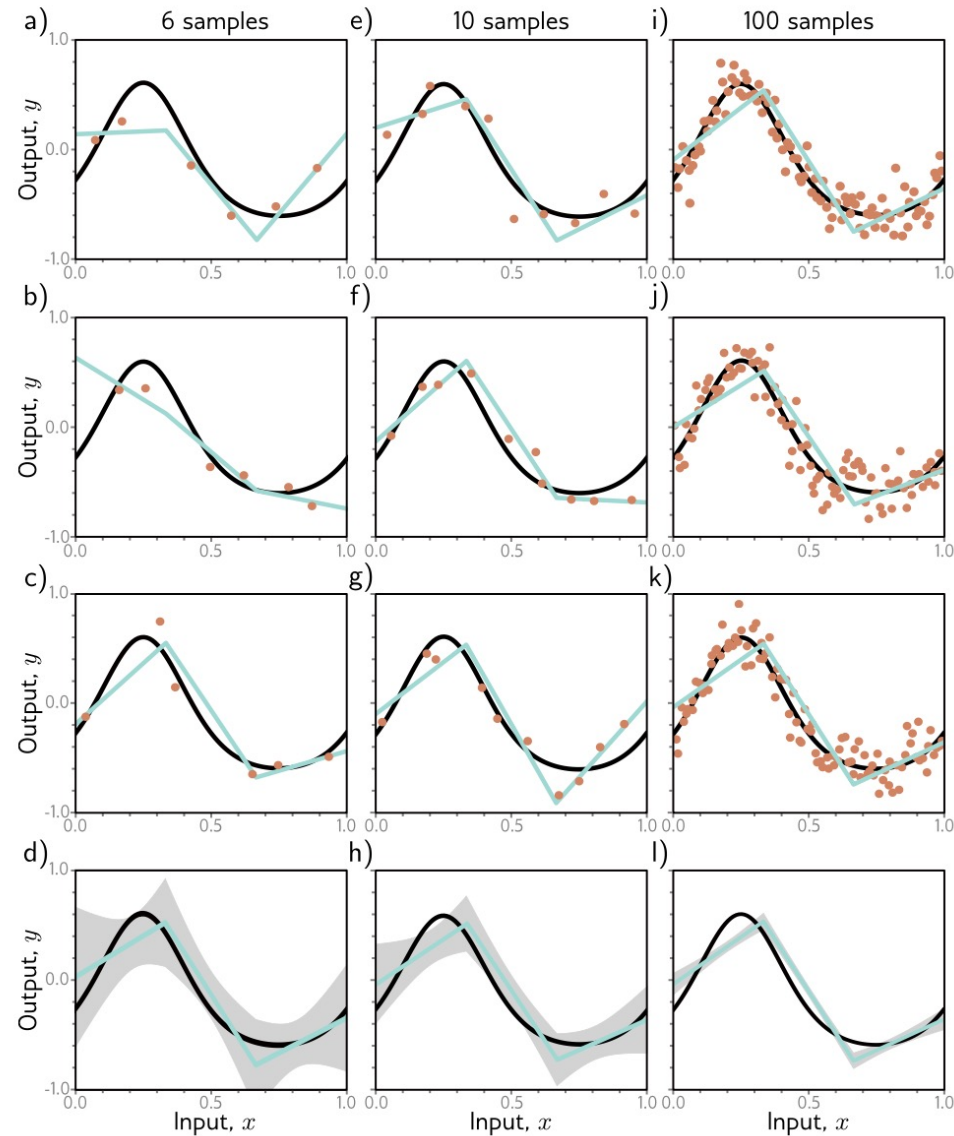
- Problem: Network may just learn the values at the inputs
 - Learn the red curve instead of the dotted blue one
 - Given only the red vertical bars as inputs

Beyond training error

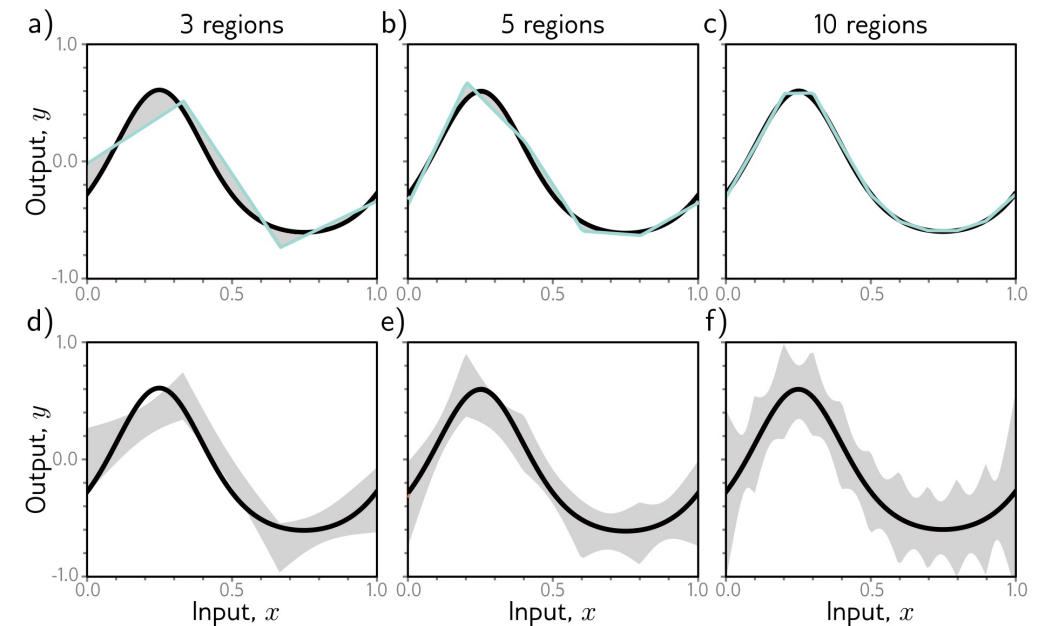
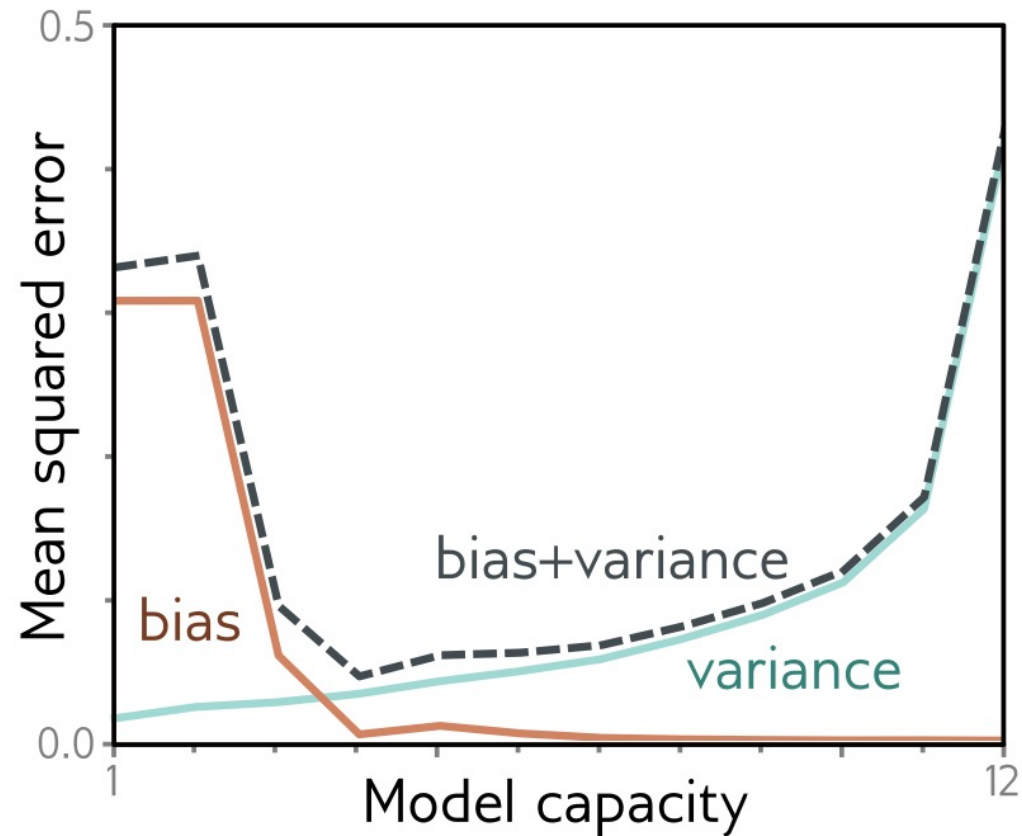
- Better optimization techniques help to reduce training loss
- Generalization gap: How to reduce generalization gap?



Variance of trained models on different sample sets



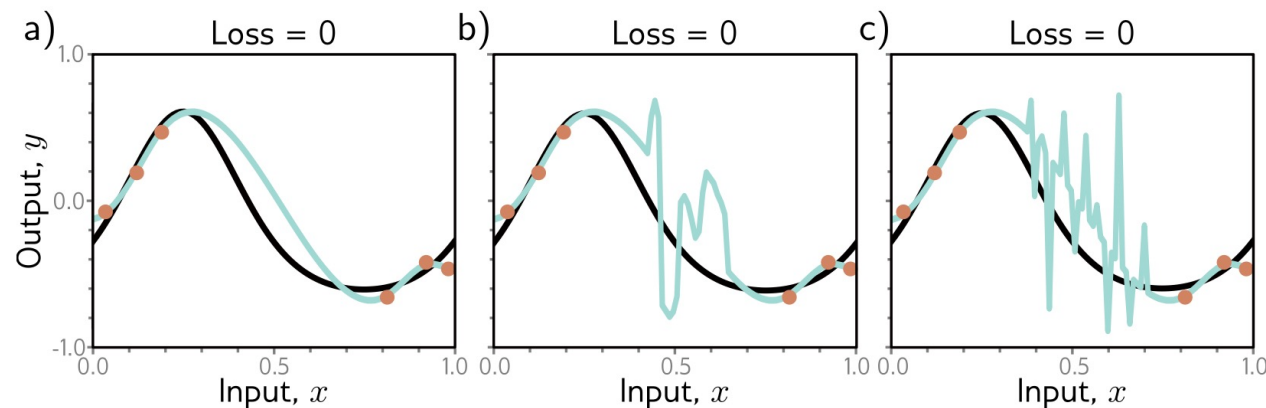
Bias-variance trade-off



Regularization

$$J(W) = \sum_{n=1}^N \text{loss}(y^{(n)}, f(y^{(n)}, W)) + \lambda R(W)$$

- How to reduce the generalization gap between training and test performance?
- Adding explicit terms to the loss function that inject our inductive biases



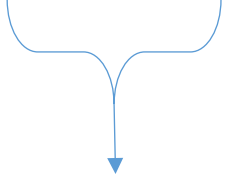
L2 regularization: Weight decay

$$L(W) = \sum_{n=1}^N \text{loss}(y^{(n)}, f(y^{(n)}, W))$$

$$J(W) = L(W) + \lambda \|W\|^2$$

$$W \leftarrow W - \alpha \nabla_W L(W) - 2\alpha\lambda W$$

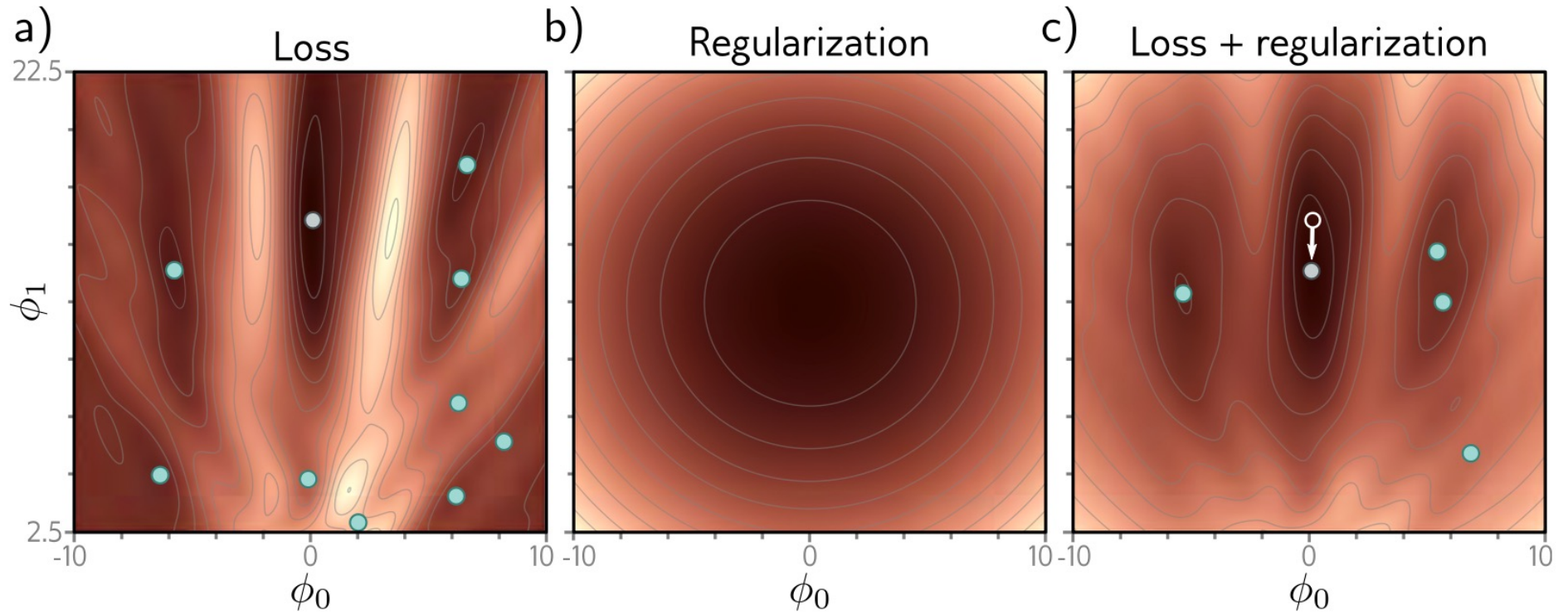
$$W \leftarrow W(1 - 2\alpha\lambda) - \alpha \nabla_W L(W)$$



Weight decay

It is usually applied to the weights but not the biases

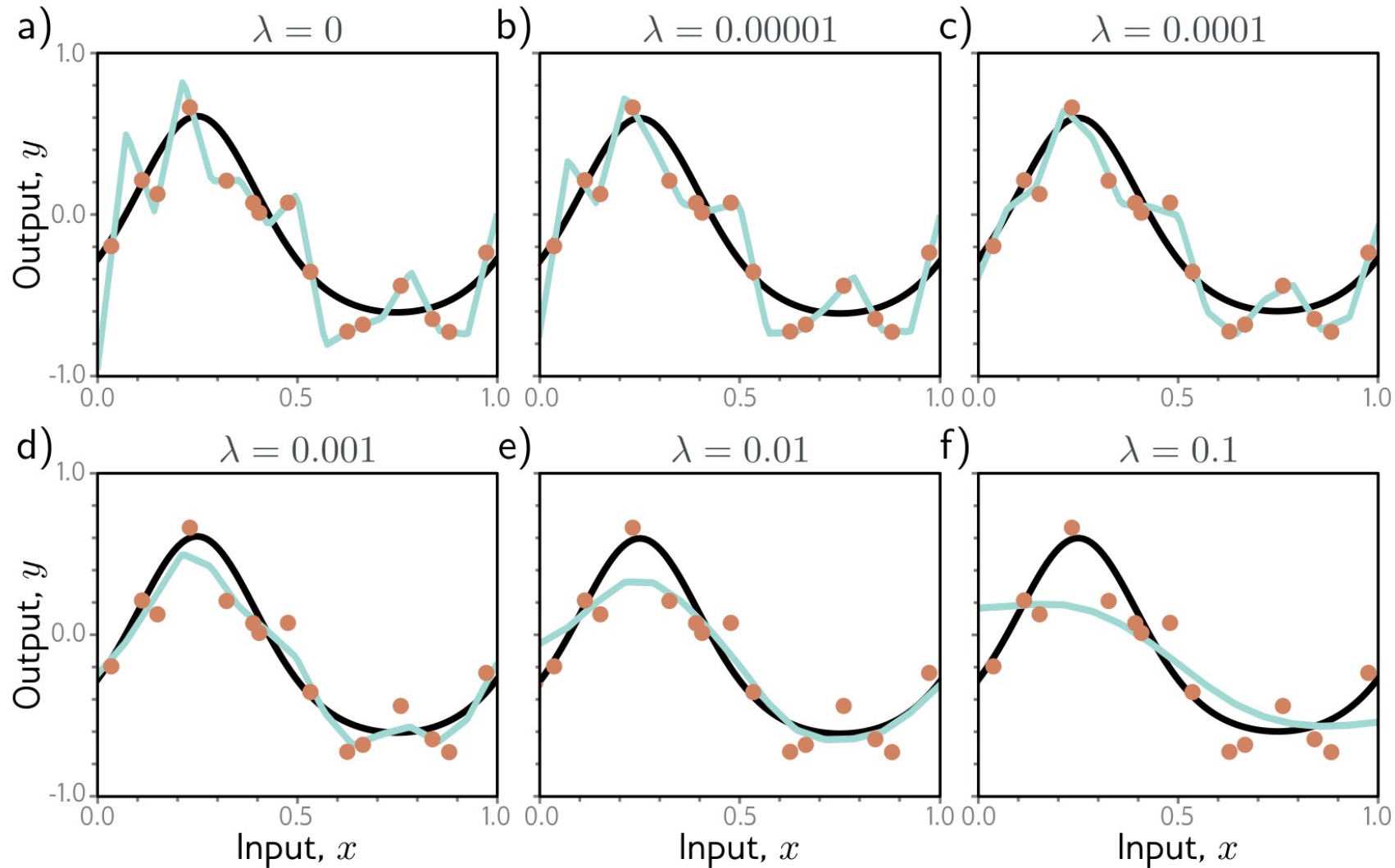
L2 regularization



Regularization parameter

- λ is the regularization parameter whose value depends on how important it is for us to want to impose the regularizer
- Increasing λ assigns greater importance to shrinking the weights
 - Make greater error on training data, to obtain a more acceptable network

Regularization parameter



Why L2 regularization?

- Variance reduction
 - With regularization, smoother cost vs. #epoch curve is obtained.
- Encouraging smaller weights, so the output function is smoother

Regularization: Add term to loss

$$J(W) = \frac{1}{N} \sum_{n=1}^N \text{loss}(y^{(n)}, f(x^{(n)}, W)) + \lambda R(W)$$

In common use:

L2 regularization

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

L1 regularization

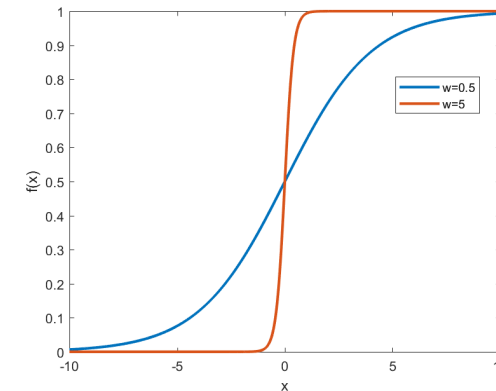
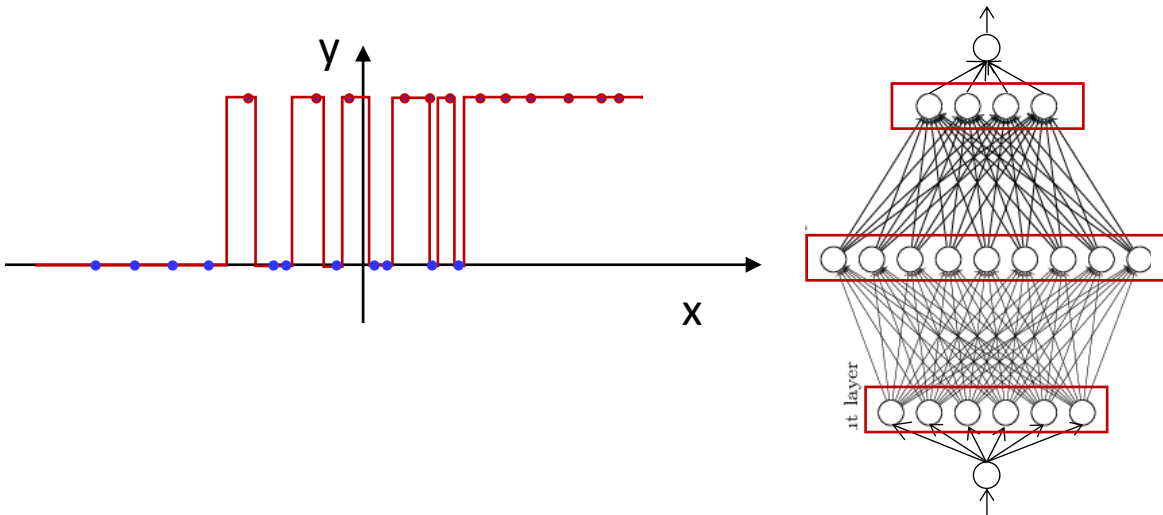
$$R(W) = \sum_k \sum_l |W_{k,l}|$$

Elastic net (L1 + L2)

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

Regularization and different activation functions

- ReLU: Lower slope for piecewise linear functions
- Sigmoid or tanh: Small W linear regime
 - A deep network with small W can also act as a near linear function
 - Steep changes facilitate overfitted responses by neurons with large w



As $|w|$ increases, the response becomes steeper

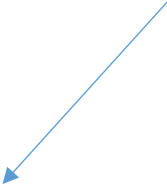
Probabilistic perspective of regularization

- MAP estimation instead of ML estimation

$$\operatorname{argmax}_W p(W|D) = \operatorname{argmax}_W p(D|W)P(W)$$

$$\operatorname{argmax}_W \log p(W) \prod_{n=1}^N p(y^{(n)}|f(x^{(n)}; W))$$

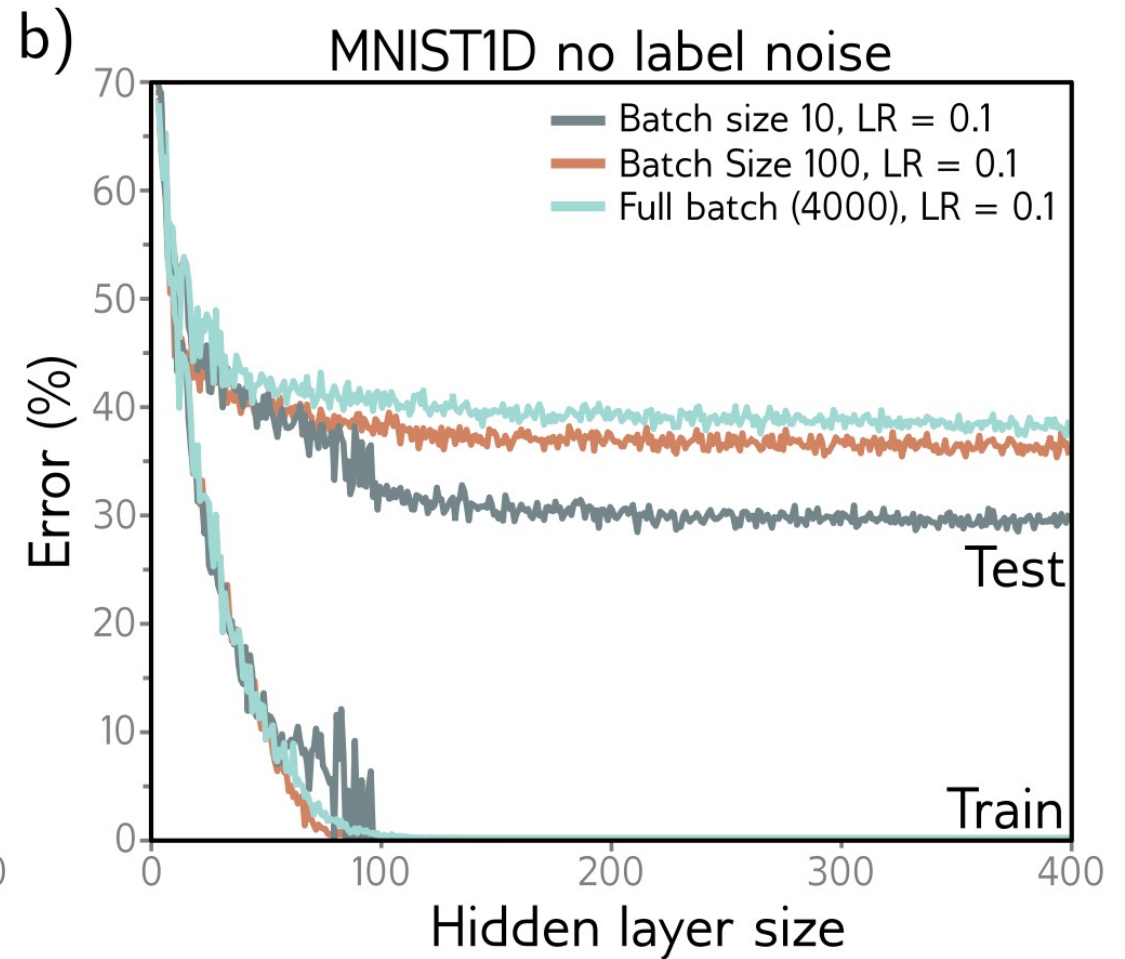
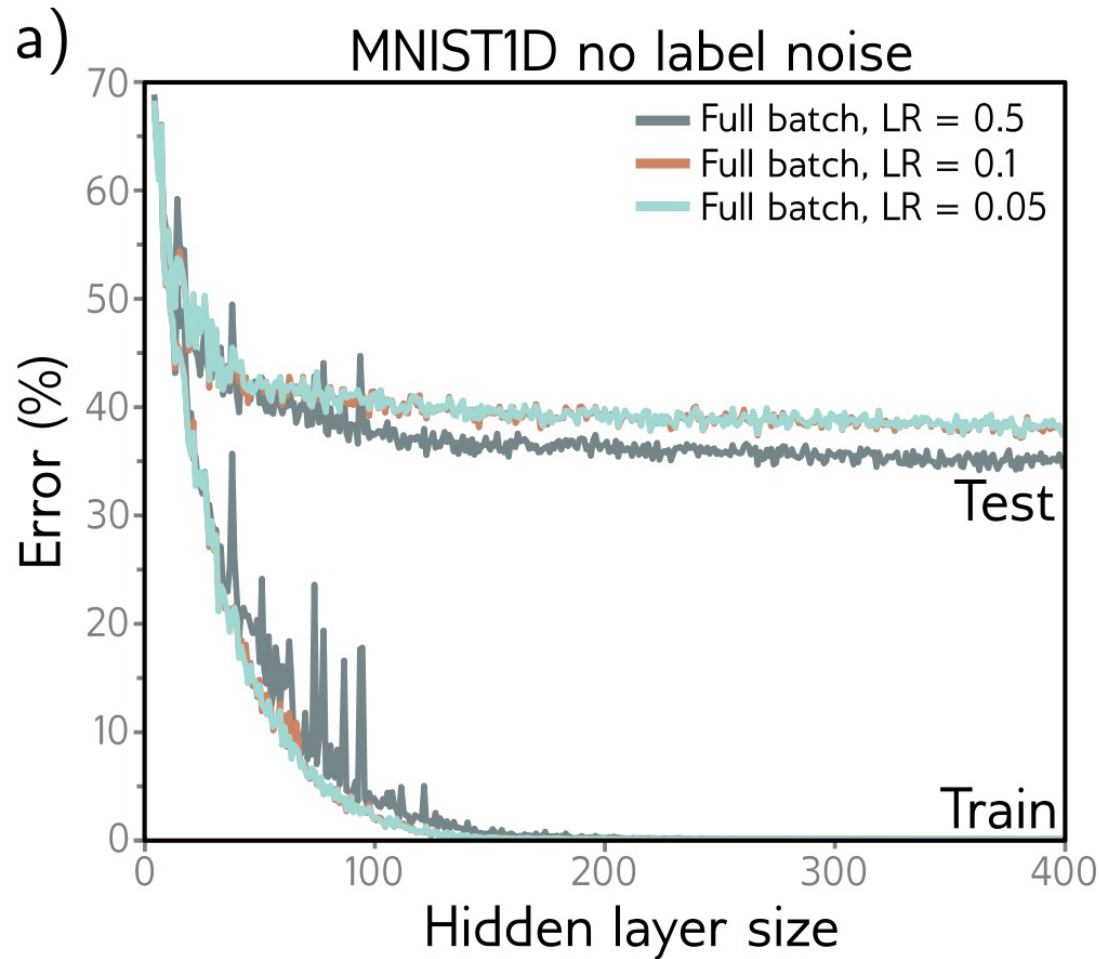
$$\operatorname{argmax}_{\phi} \log p(W) + \sum_{n=1}^N \log p(y^{(n)}|f(x^{(n)}; \phi))$$


$$p(W) = N(0, \alpha^2 I) = \frac{1}{(\sqrt{2\pi}\alpha)^{M/2}} \prod_{i=1}^M e^{-\frac{1}{2\alpha^2}w_i^2}$$

Implicit regularization

- SGD generalizes better than gradient descent
 - smaller batch sizes may perform better than larger ones
 - The inherent randomness allows to reach different parts of the loss function.
- Implicit regularization: a preference for some solutions over others
 - encourages solutions where all the data fits well (so the batch variance is small)
 - rather than solutions where some of data fit extremely well and other data less well
- SGD implicitly favors places where the gradients are stable (where all the batches agree on the slope)

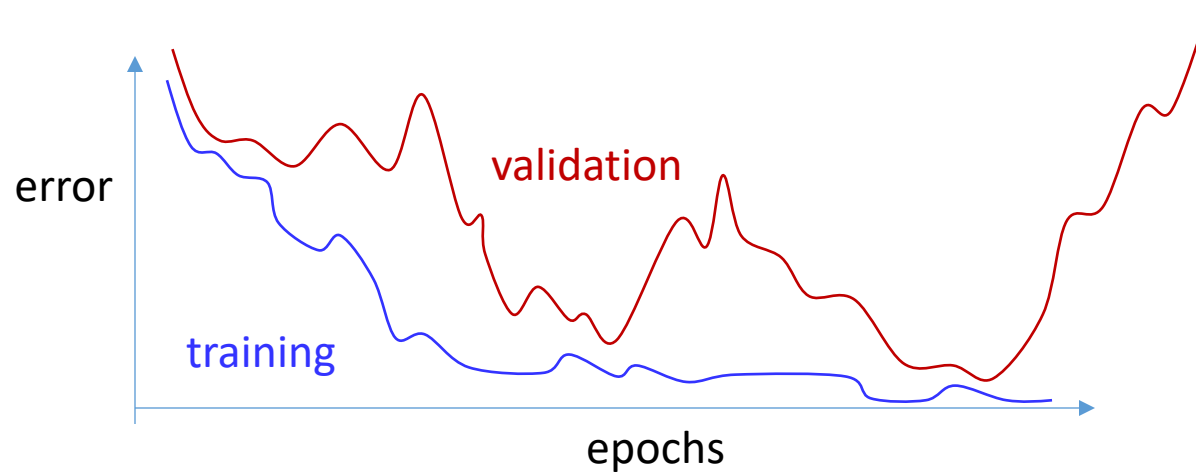
SGD: Generalization



Early stopping

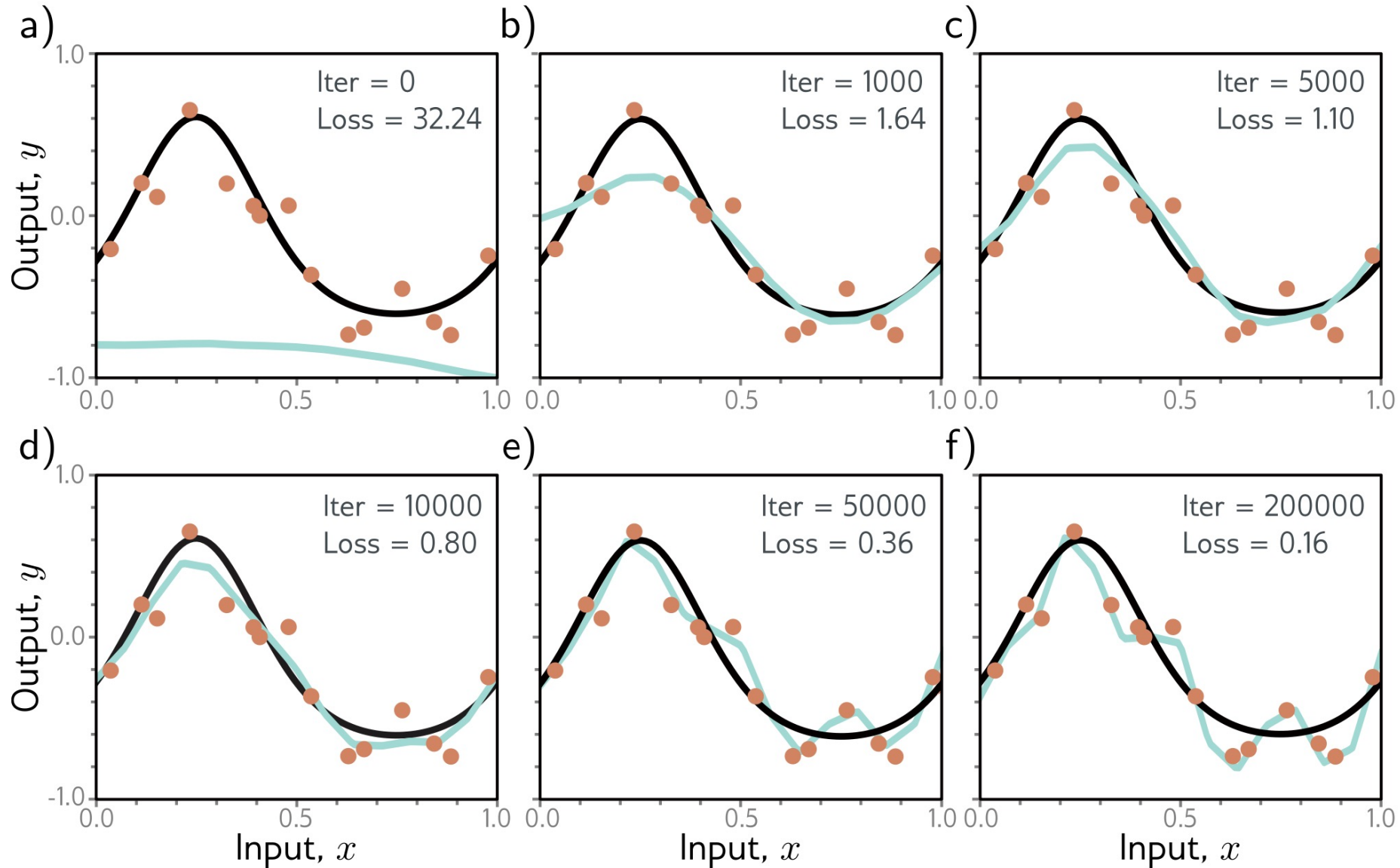
- Stopping the training procedure before it has fully converged
- One way of thinking about this is L2 regularization since the weights are initialized to small values
 - they simply don't have time to become large
- Early stopping has a single hyperparameter
 - the number of steps after which learning is terminated

Early stopping



- Continued training can result in over fitting to training data
 - Track performance on a held-out validation set
 - Apply one of several early-stopping criterion to terminate training when performance on validation set degrades significantly

Early stopping: Example



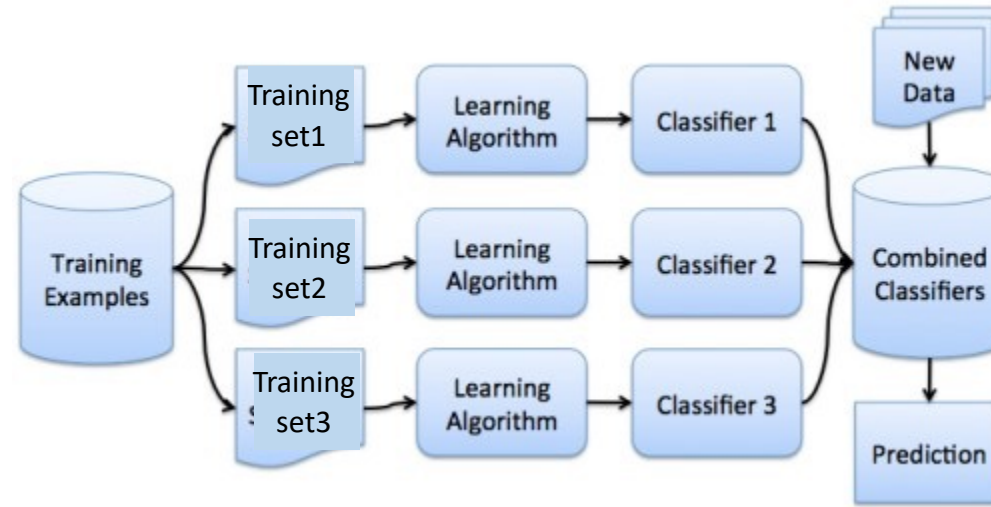
Model ensembles

1. Train multiple models
2. At test time average their results

Model ensembles

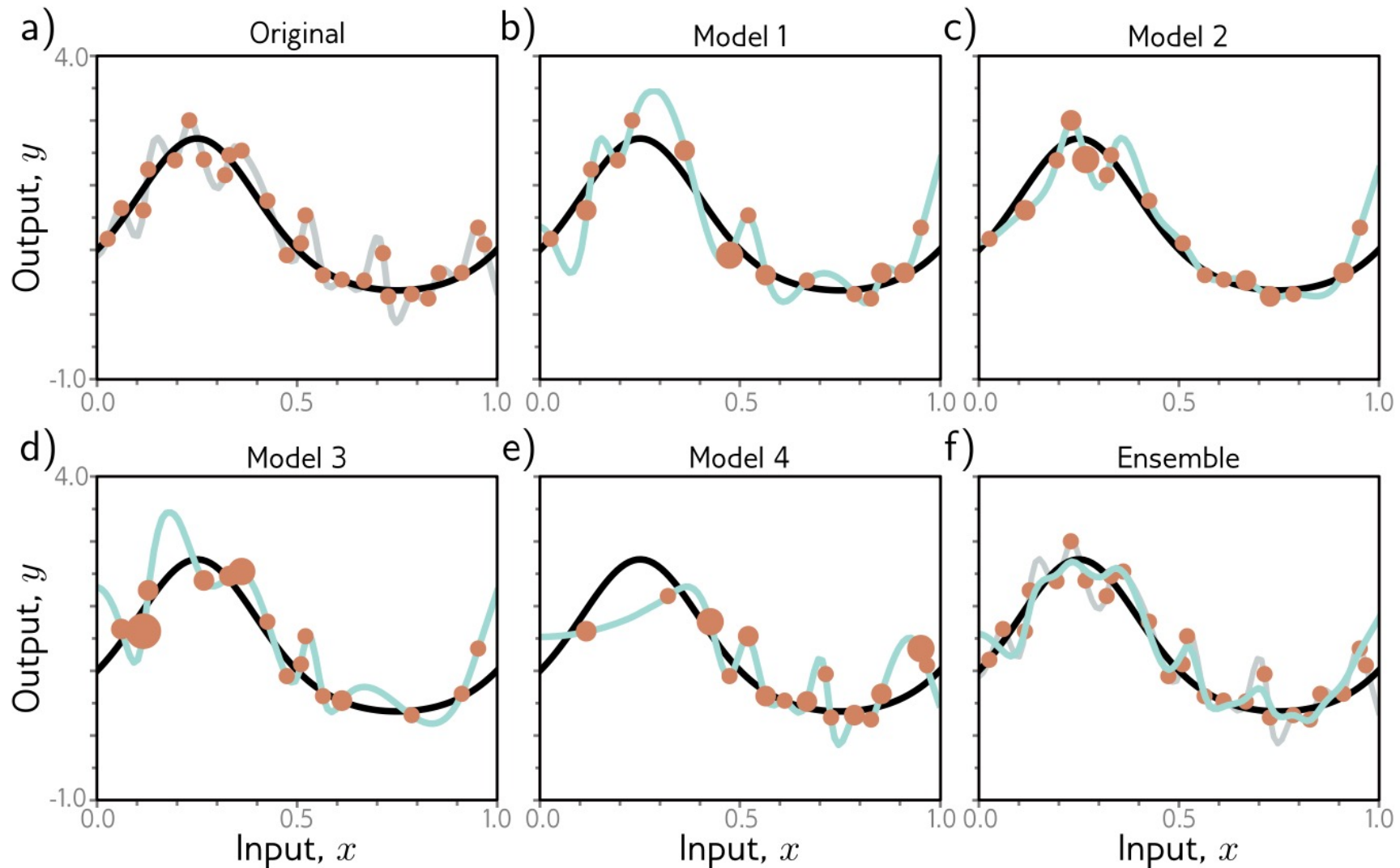
- Train multiple models, and at test time average their predictions.
 - Mean or median of the outputs (for regression problems)
 - the most frequent predicted class or mean of the pre-softmax activations (for classification problems)
- Candidates
 - **Training multiple models**
 - **Same model, different initializations** (after selecting the best hyperparameters)
 - **Top models discovered during cross-validation**
 - **Different checkpoints of a single model** (If training is very expensive)
 - maybe limited success to form an ensemble.
 - **Running average of parameters during training**
 - maintains an exponentially decaying sum of previous weights during training.

A brief detour.. Bagging



- Popular method proposed by Leo Breiman:
 - Sample training data and train several different classifiers
 - Classify test instance with entire ensemble of classifiers
 - Vote across classifiers for final decision
 - Empirically shown to improve significantly over training a single classifier from combined data

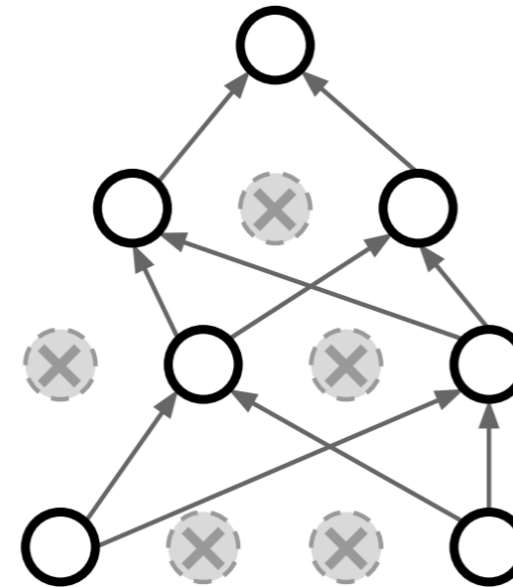
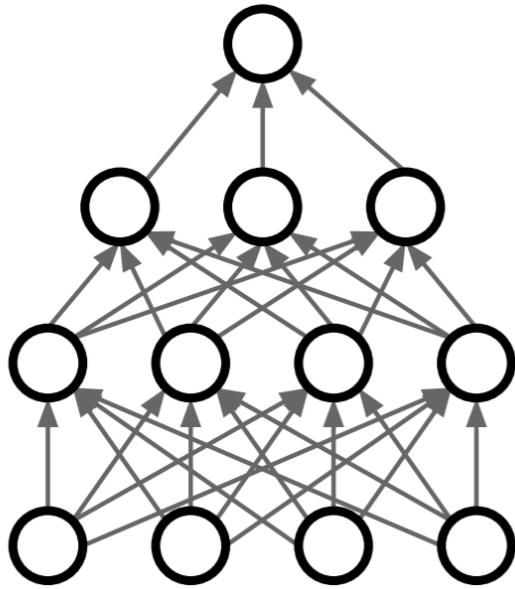
Ensemble models



Regularization: Dropout

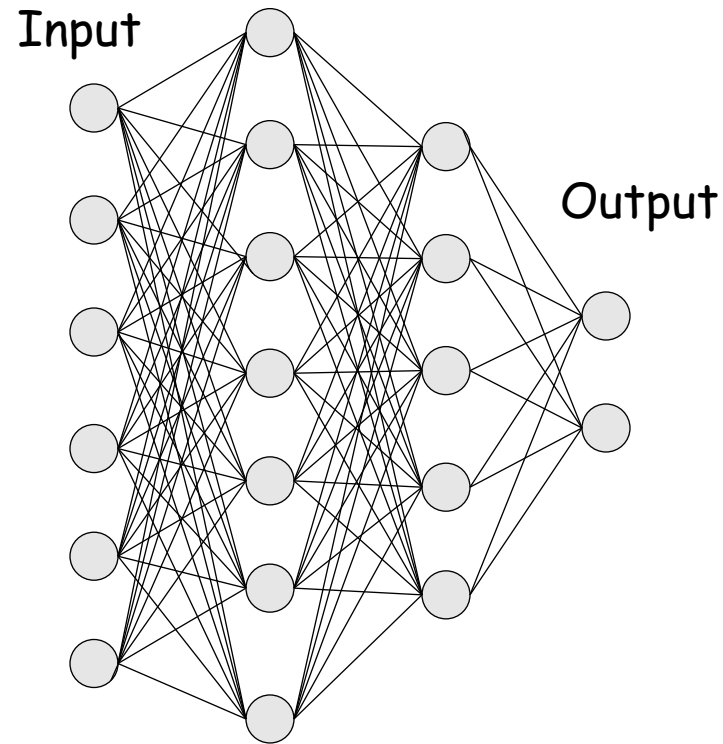
Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

- In each forward pass, randomly set some neurons to zero
 - Probability of dropping is a hyperparameter; 0.5 is common



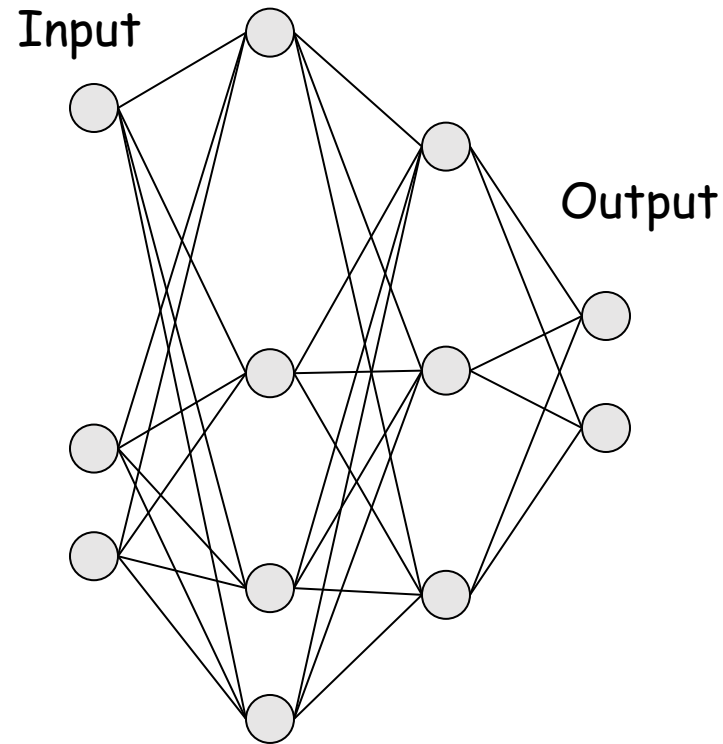
- Thus, a smaller network is trained for each sample
 - Smaller network provides a regularization effect

Dropout



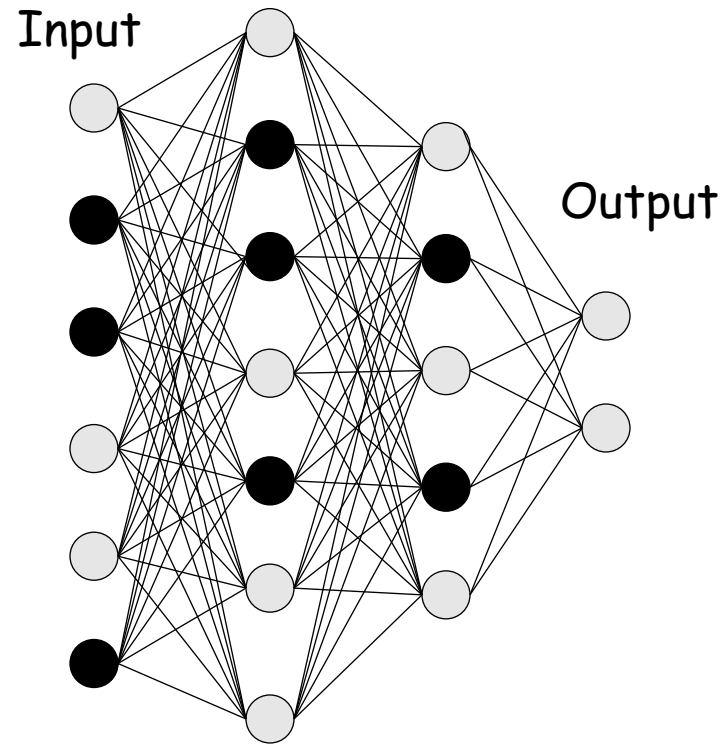
- **During training:** For each input, at each iteration, “turn off” each neuron with a probability $1-\alpha$
 - α shows keep probability

Dropout



- **During training:** For each input, at each iteration, “turn off” each neuron with a probability $1-\alpha$
 - Also can turn off inputs similarly

Dropout

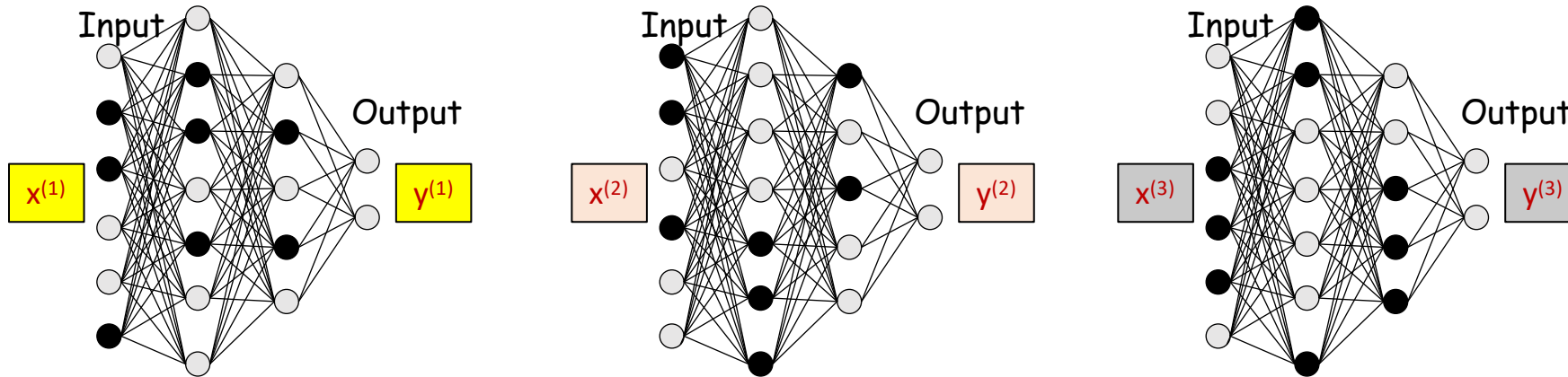


- **During training:** For each input, at each iteration, “turn off” each neuron (including inputs) with a probability $1-\alpha$
 - In practice, set them to 0 according to the success of a Bernoulli random number generator with success probability $1-\alpha$

Implementing dropout

- for $l=1,\dots,L$
 - $m^{[l]} \leftarrow I(\text{rand}(\#neurons^{[l]}, 1) < \text{keep_prob})$
 - $a^{[l]} \leftarrow a^{[l]} .* m^{[l]}$

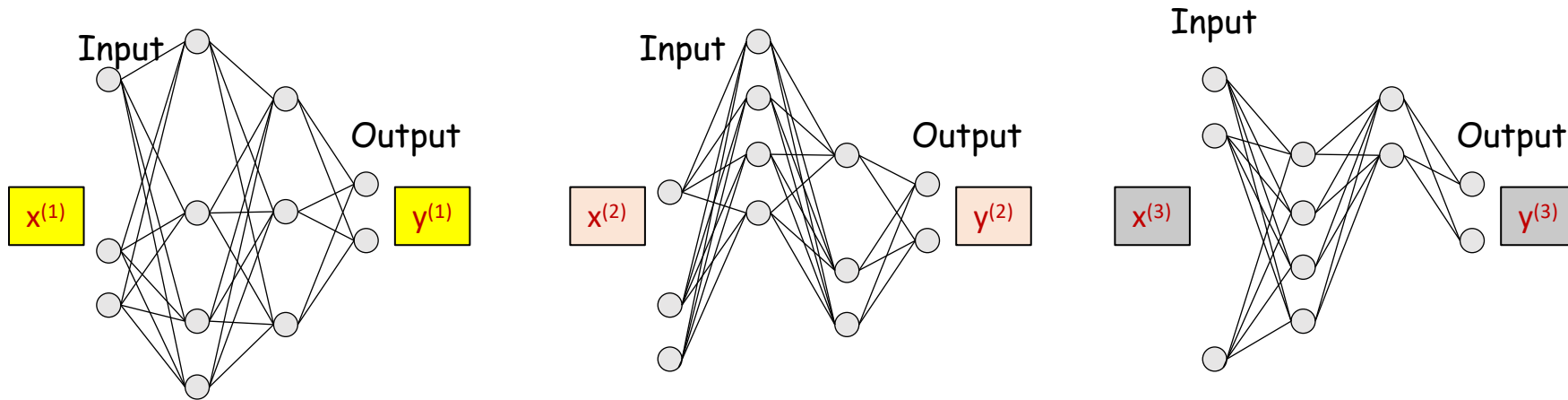
Dropout



The pattern of dropped nodes
changes for *each input*
i.e. in every pass through the net

- **During training:** For each input, at each iteration, “turn off” each neuron (including inputs) with a probability $1-\alpha$

Dropout

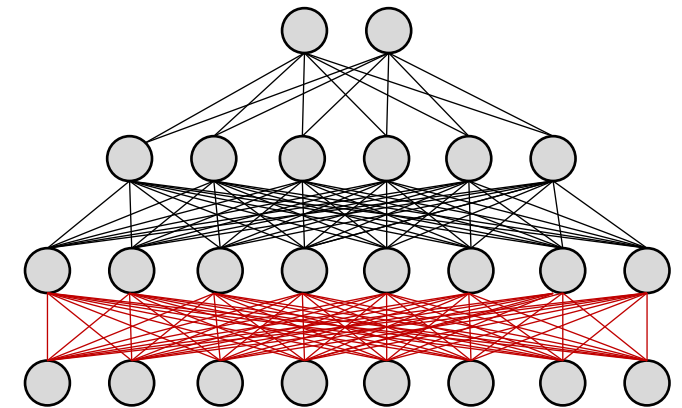
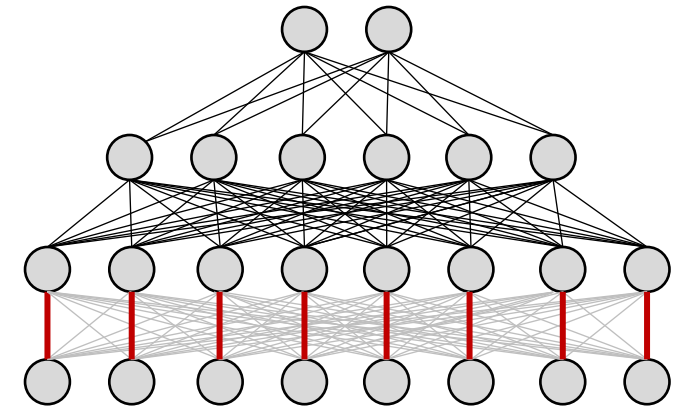


The pattern of dropped nodes changes for *each input* i.e. in every pass through the net

- **During training:** Backpropagation is effectively performed only over the remaining network
 - The effective network is different for different inputs
 - Gradients are obtained only for the weights and biases *from* “On” nodes *to* “On” nodes
 - For the remaining, the gradient is just 0

Dropout as a mechanism to increase pattern density

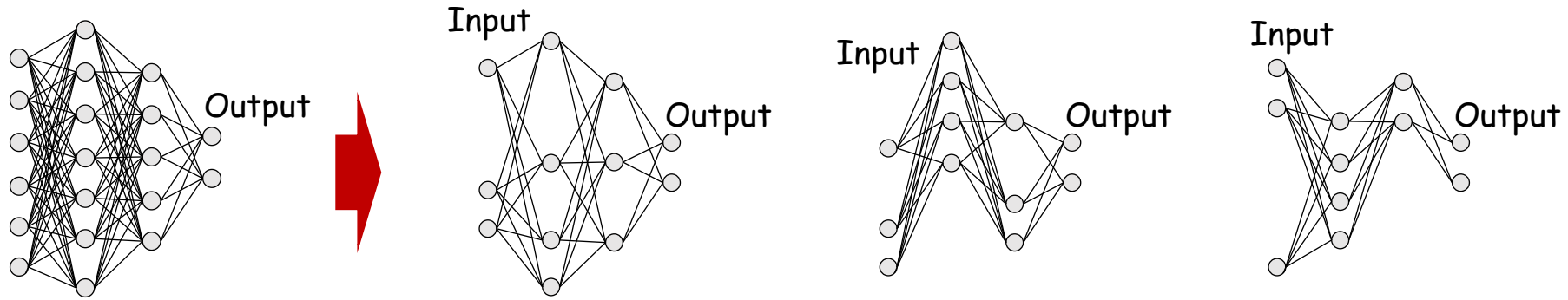
- Dropout forces the neurons to learn “rich” and redundant patterns
- E.g. without dropout, a non-compressive layer may just “clone” its input to its output
 - Transferring the task of learning to the rest of the network
- Dropout forces the neurons to learn denser patterns
 - With redundancy



Why does drop-out work?

- **Intuition I:** It cannot rely on any one feature
- This makes the network less dependent on any unit and thus encourages the weights to have smaller magnitudes
 - similar effect to the L_2 regularization

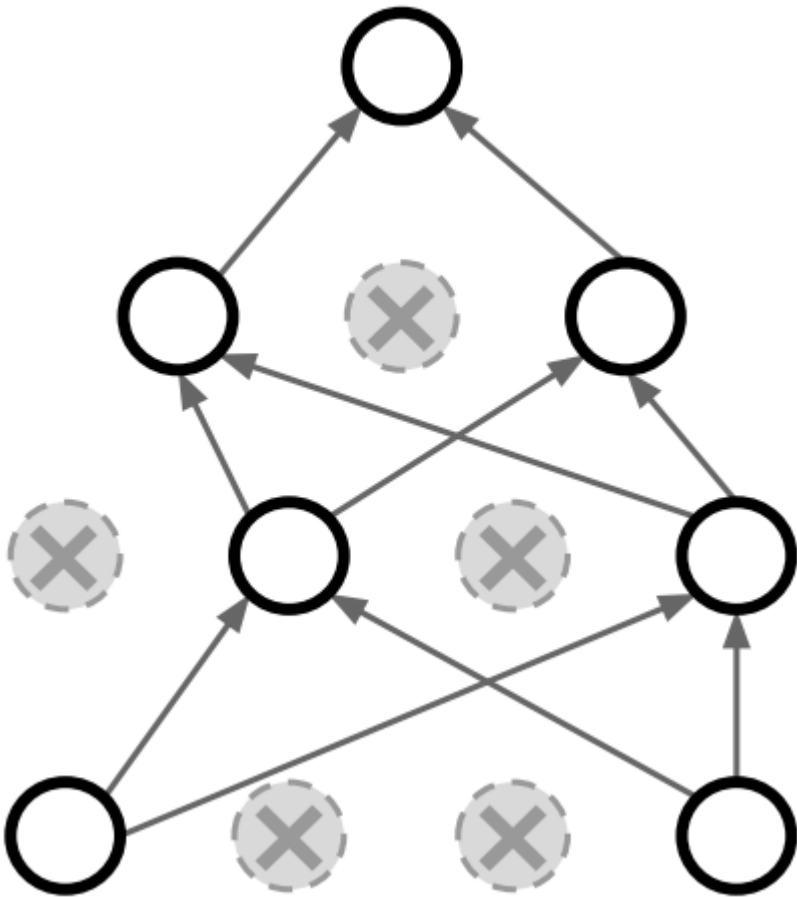
Why does drop-out work?



- **Intuition II:** Ensemble approach
- For a network with a total of M neurons, there are 2^M possible sub-networks
 - Obtained by choosing different subsets of nodes
 - Dropout *samples* over all 2^M possible networks
 - Effectively learns a network that *averages* over all possible networks
 - Bagging

Regularization: Dropout

- **Intuition II: Ensemble approach**



Dropout is training a **large ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!

Only $\sim 10^{82}$ atoms in the universe...

What each neuron computes

- Each neuron actually has the following activation:

$$a_i^{[l]} = D \phi \left(\sum_j w_{ji}^{[l]} a_j^{[l-1]} + b_i^{[l]} \right) \quad a^{[l]} = f(a^{[l-1]}, D)$$

- Where D is a Bernoulli variable that takes a value 1 with probability α
- D may be switched on or off for individual sub networks, but over the ensemble, the *expected output* of the neuron is

$$a_i^{[l]} = \alpha \phi \left(\sum_j w_{ji}^{[l]} a_j^{[l-1]} + b_i^{[l]} \right)$$

- During *test* time, we will use the *expected* output of the neuron
 - Which corresponds to the bagged average output
 - Consists of simply scaling the output of each neuron by α

Dropout: Test time

```
def predict(X):  
    # ensembled forward pass  
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations  
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations  
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
output at test time = expected output at training time

Dropout summary

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

Inverted dropout

- Since test-time performance is critical, it is preferable to use inverted dropout,
 - which performs the scaling at train time, leaving the forward pass at test time untouched.
 - Just divide the activation of each layer by keep-prob

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

test time is unchanged!



Backward Pass

- Output layer (L) :

$$- \frac{\partial \text{loss}(a_i^{[L]}, y)}{\partial a_i^{[L]}}$$

$$- \frac{\partial \text{loss}}{\partial z_i^{[L]}} = f_L' \left(z_i^{[L]} \right) \frac{\partial \text{loss}}{\partial a_i^{[L]}}$$

- For layer $l = L - 1$ *downto* 0

– For $i = 1 \dots d^{[l]}$

- if (not dropout layer OR $\text{mask}(l, i)$)

- $\frac{\partial \text{loss}}{\partial a_i^{[l]}} = \sum_j w_{ij}^{[l+1]} \frac{\partial \text{loss}}{\partial z_j^{[l+1]}} \text{mask}(l+1, j)$

- $\frac{\partial \text{loss}}{\partial z_i^{[l]}} = f_l' \left(z_i^{[l]} \right) \frac{\partial \text{loss}}{\partial a_i^{[l]}}$

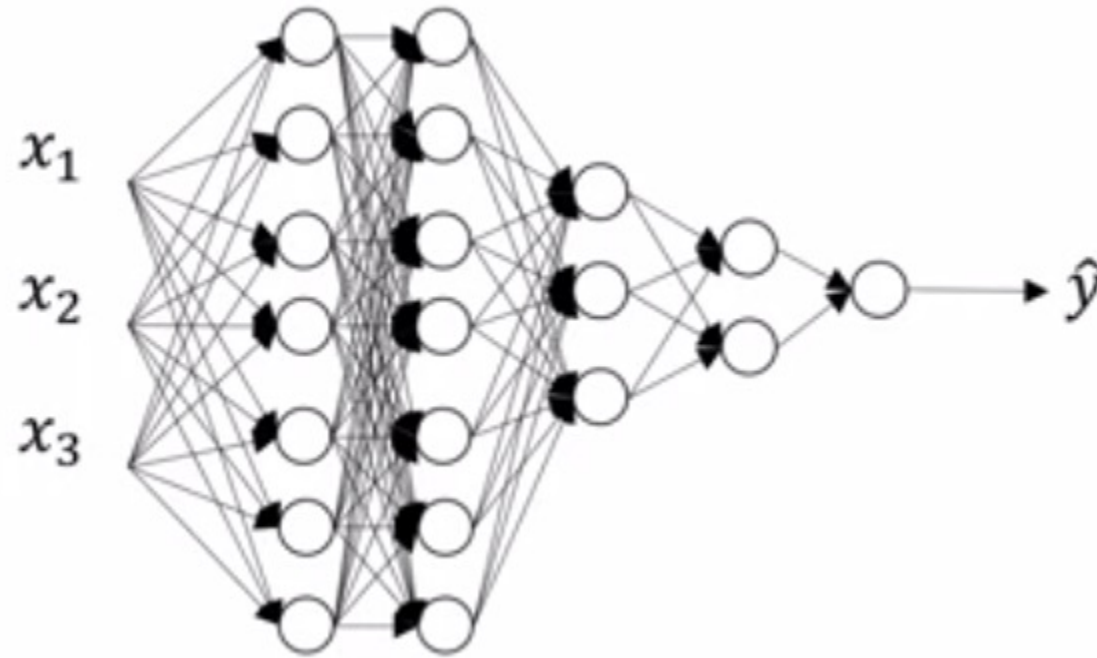
- $\frac{\partial \text{loss}}{\partial w_{ij}^{[l+1]}} = a_i^{[l]} \frac{\partial \text{loss}}{\partial z_j^{[l+1]}} \text{mask}(l+1, j) \quad \text{for } j = 1 \dots d^{[l+1]}$

- else

- $\frac{\partial \text{loss}}{\partial z_i^{[l]}} = 0$

Dropout on layers

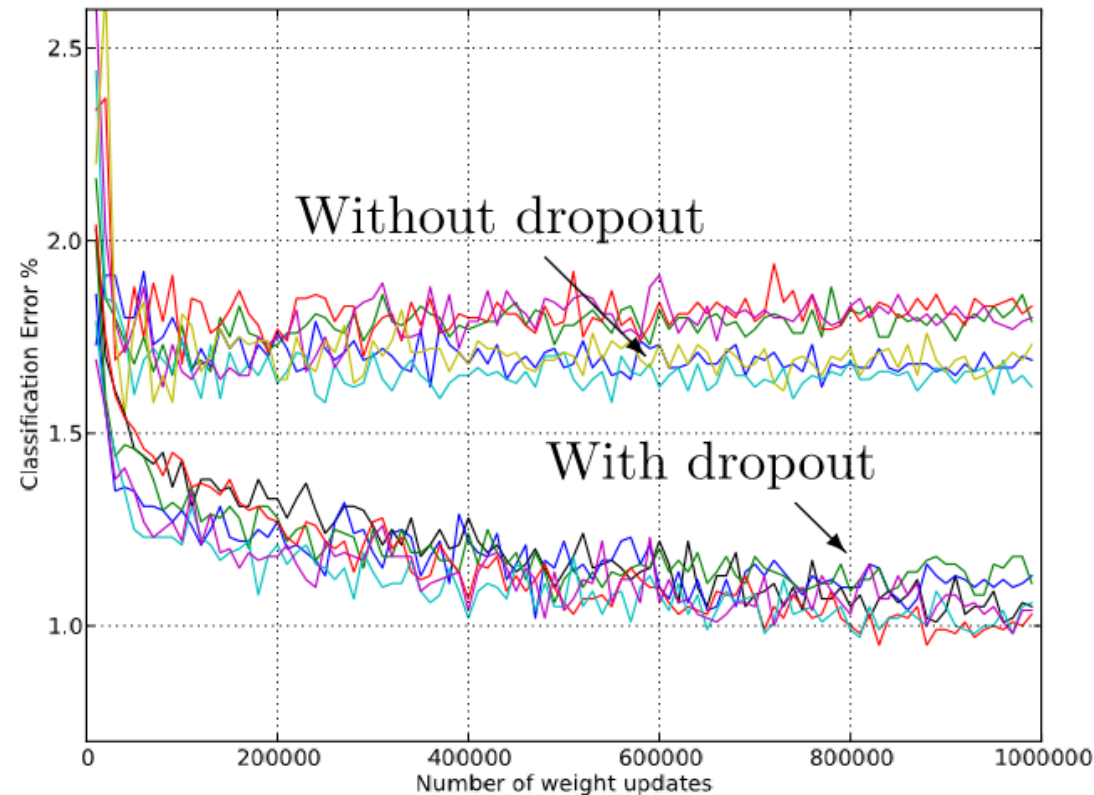
- Different keep-prob can be used for different layers
 - However, more hyper-parameters are required



Dropout issues

- If your NN is significantly overfitting, dropout usually help to prevent overfitting
- Training takes longer time but provides better generalization
 - Since in each iteration of weight update, only a subset of weights are updated.
- If your NN is not overfitting, bigger NN with dropout may help
- The cost function is no longer well defined.

Dropout: Typical results



- Test error for different architectures on MNIST with and without dropout
 - 2-4 hidden layers with 1024-2048 units

Data augmentation

- Getting more training data can be expensive
- But, sometimes we can generate more training examples from the datasets
- transform each input data example in such a way that the label stays the same

Additional heuristics: Data augmentation



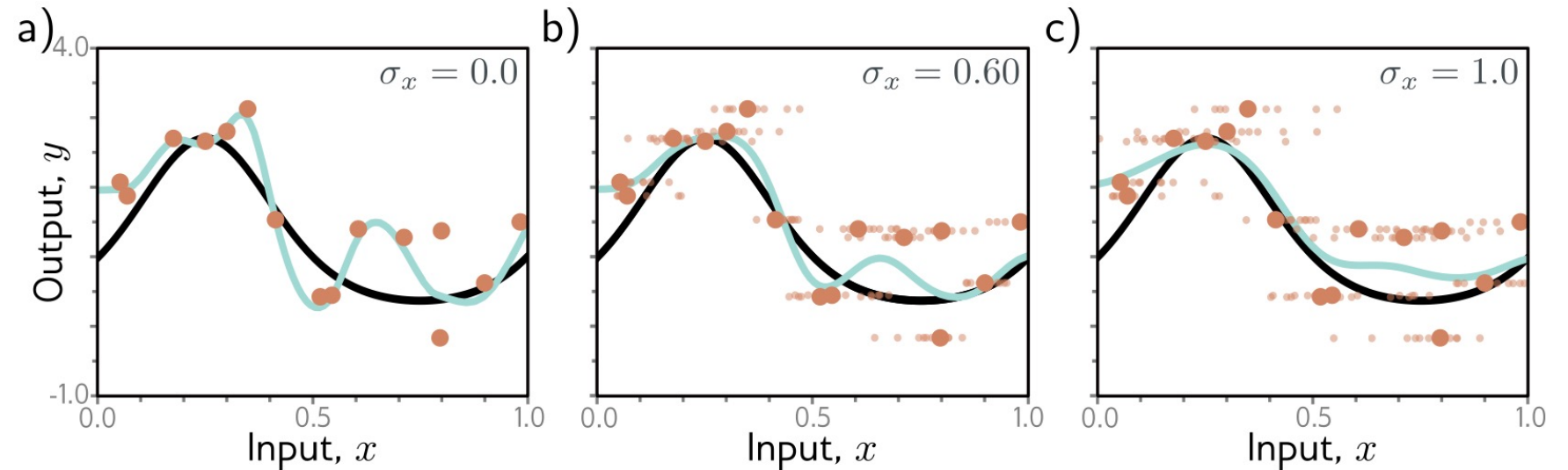
- Available training data will often be small
- “Extend” it by distorting examples in a variety of ways to generate synthetic labelled examples

Applying noise

- Input data
- Weights
- Labels

Applying noise

- Input data
- Weights
- Labels



Prince, Understanding Deep Learning, 2023

Applying noise

- Input data
- **Weights**
 - encourages sensible predictions even for small perturbations of the weights
- Labels

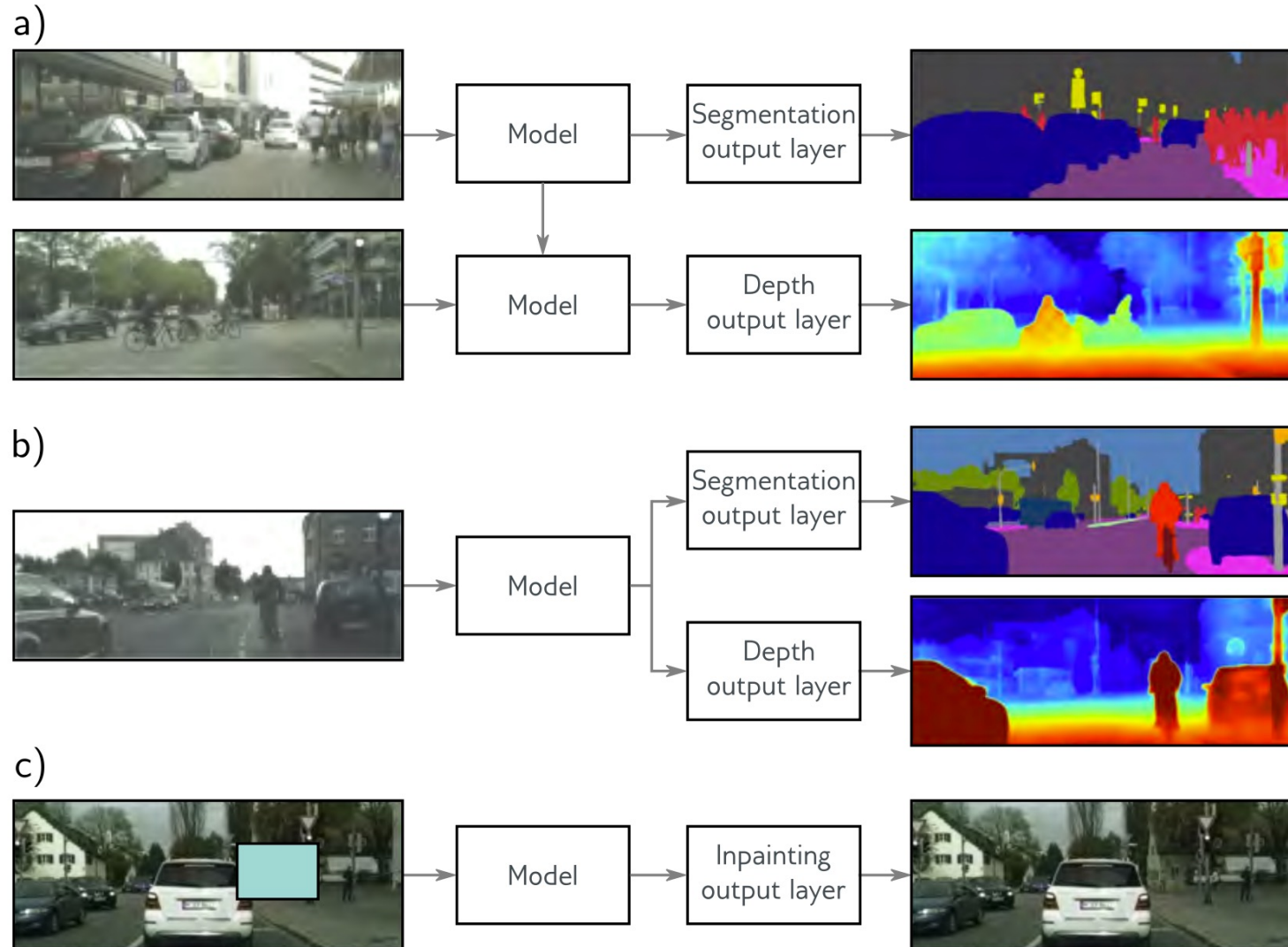
Applying noise

- Input data
- Weights
- **Labels**
 - MLE for multiclass classification is pushed to overconfident behavior
 - very large logit for the correct class and very small logit for the wrong classes
 - randomly changing a subset of the labels at each training iteration to avoid the above problem

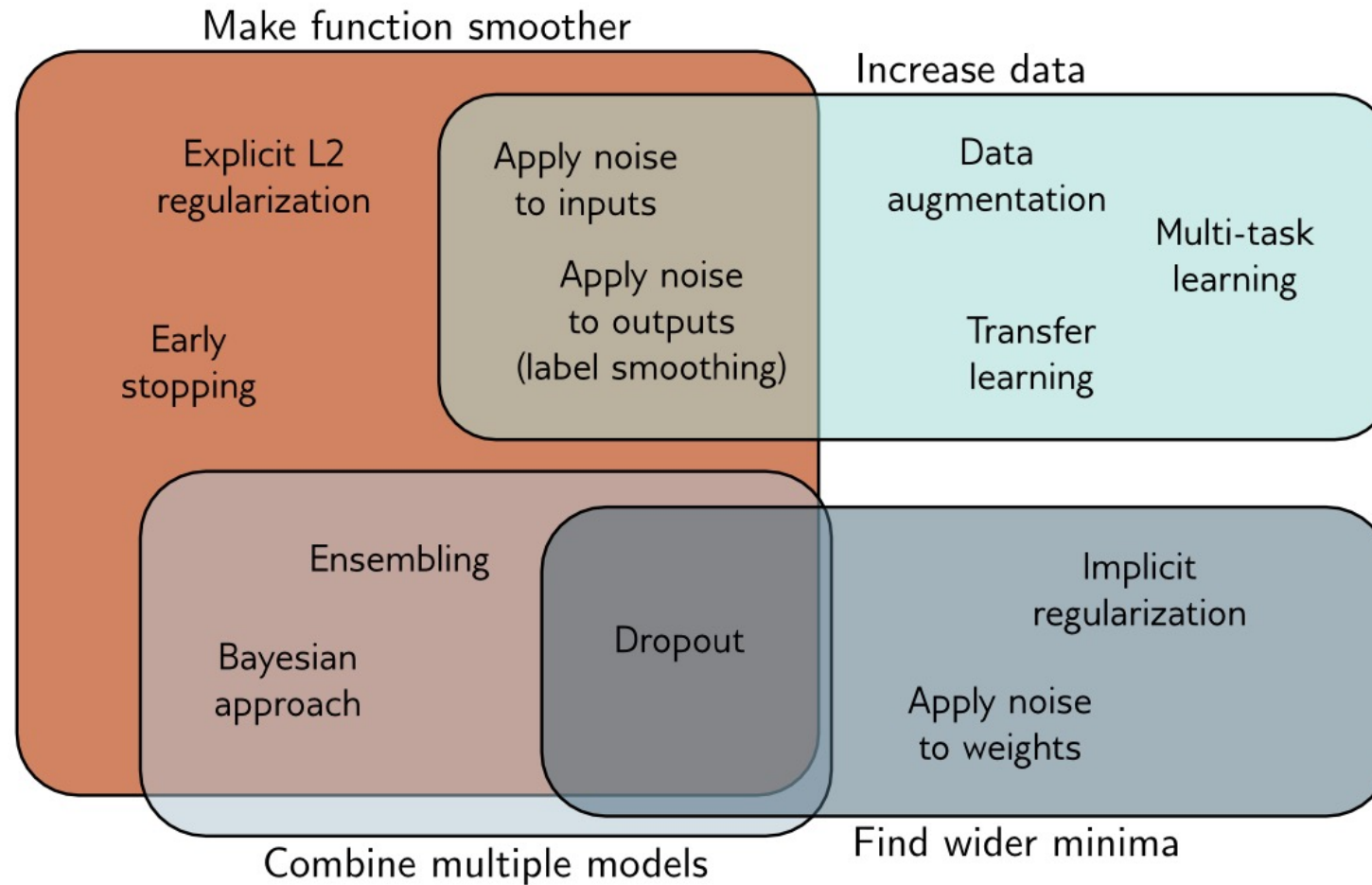
Inductive bias via architecture: Weight sharing

- Inductive bias injecting into a network to express some known invariances
- Parameter or weight sharing
 - forming weights into groups and requiring that all weights of each group share the same value

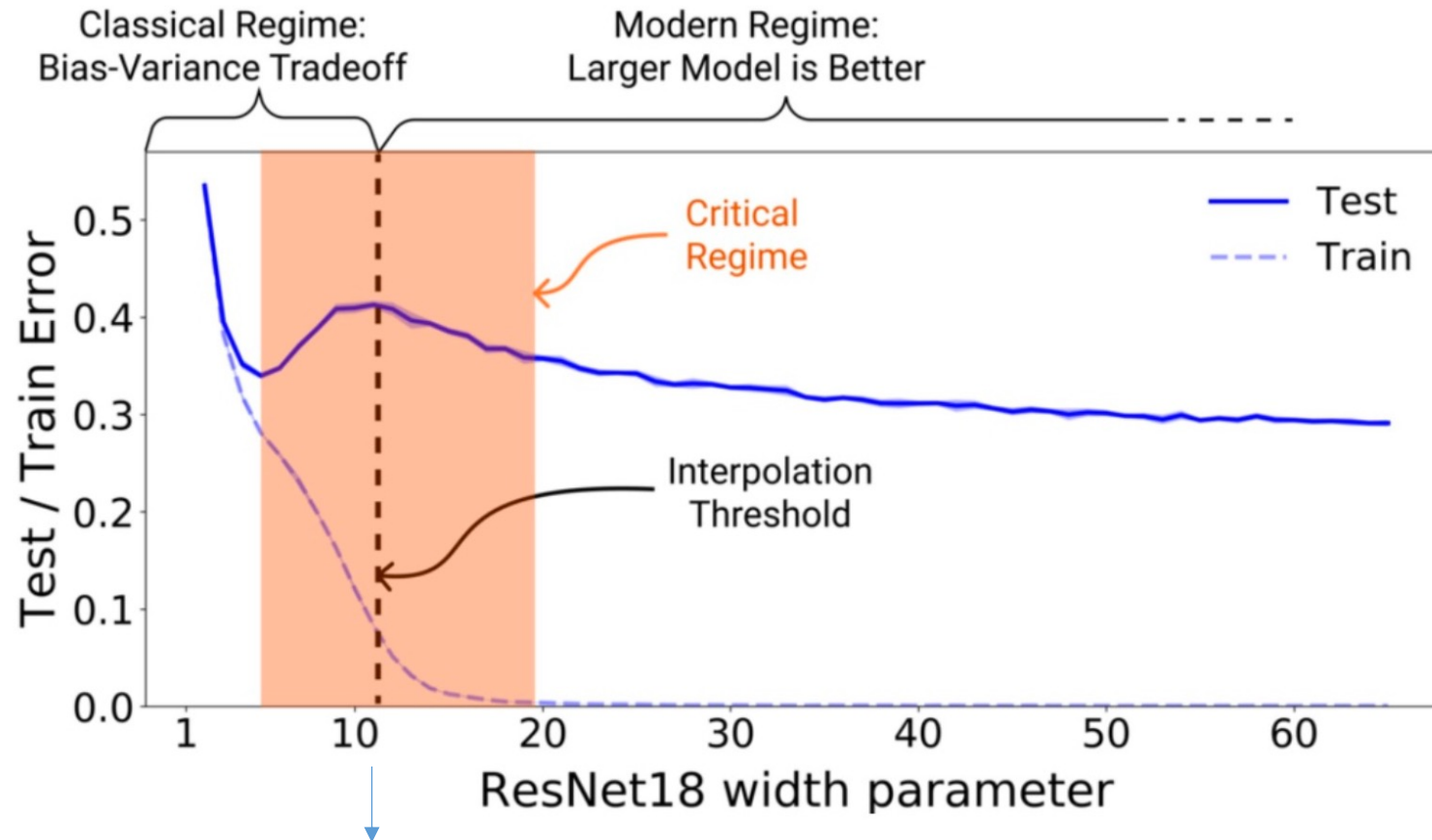
Generalization by learning approach



Overview of regularization techniques



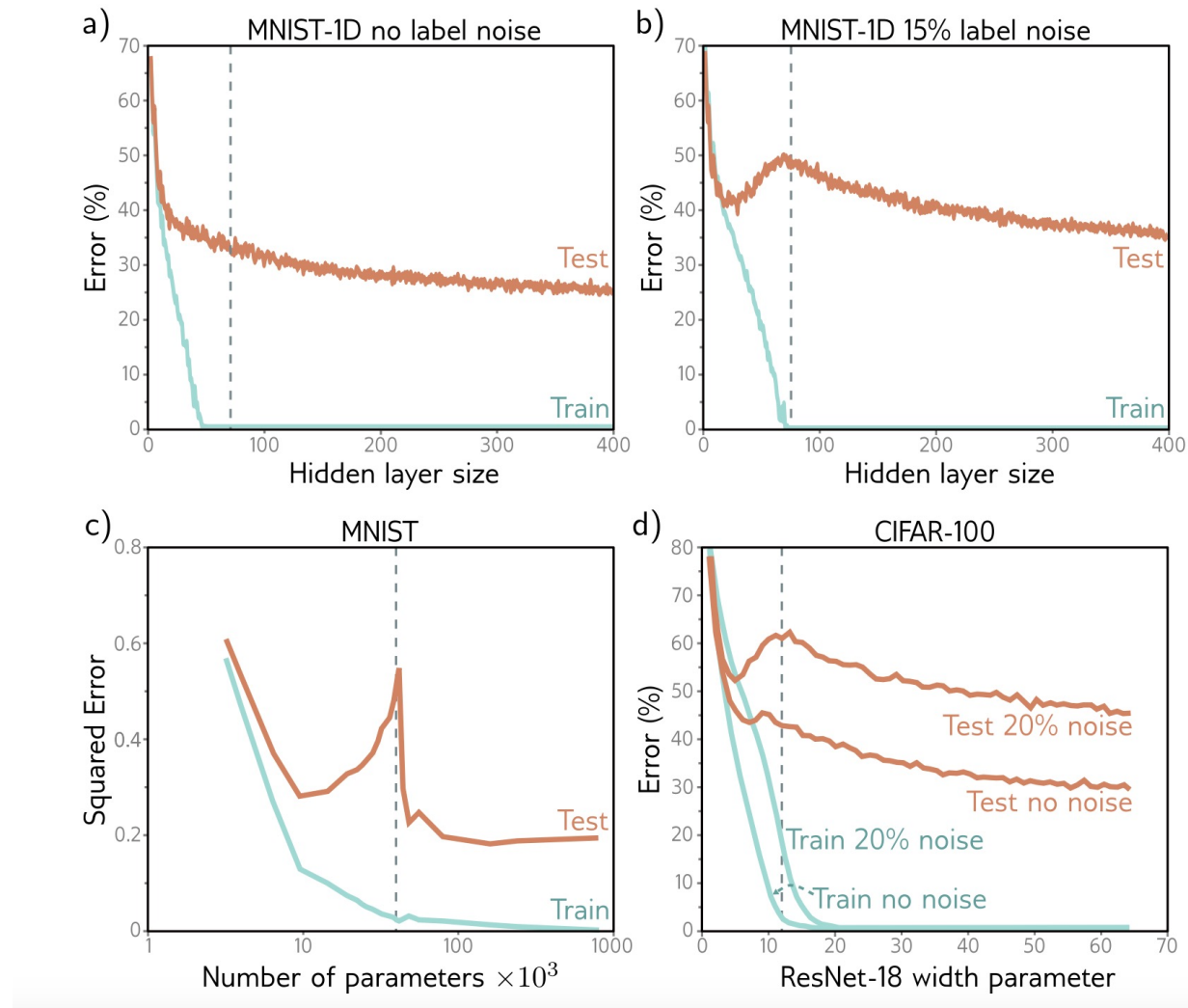
Double descent



Where the effective model complexity equals
the number of data points in the training set

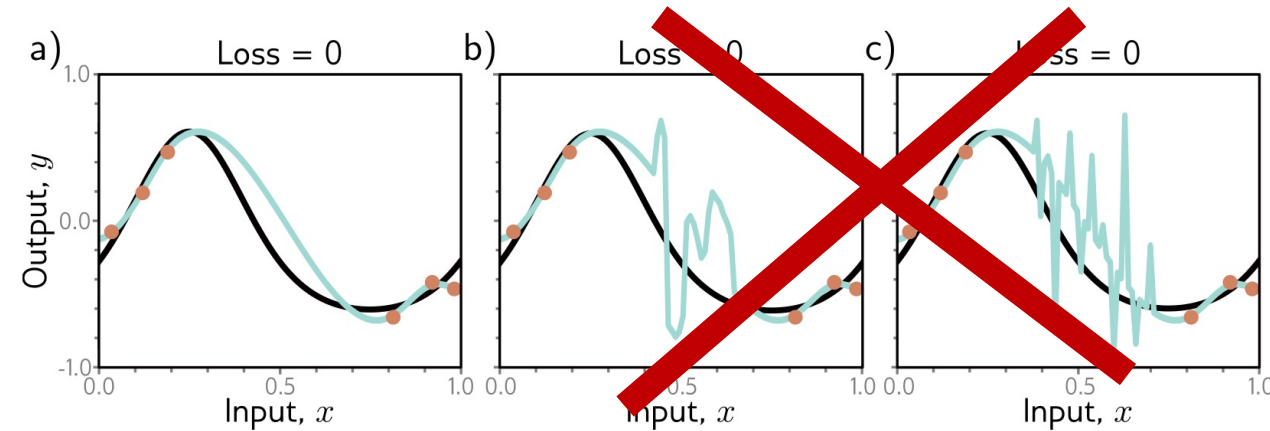
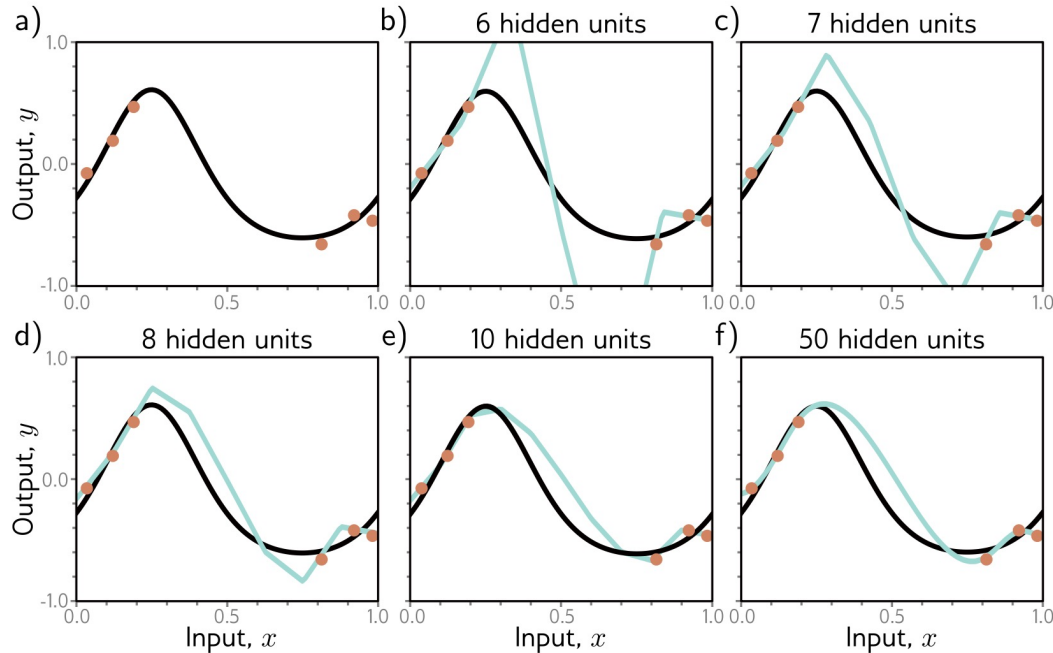
Nakkiran *et al.*, 2019

Double descent



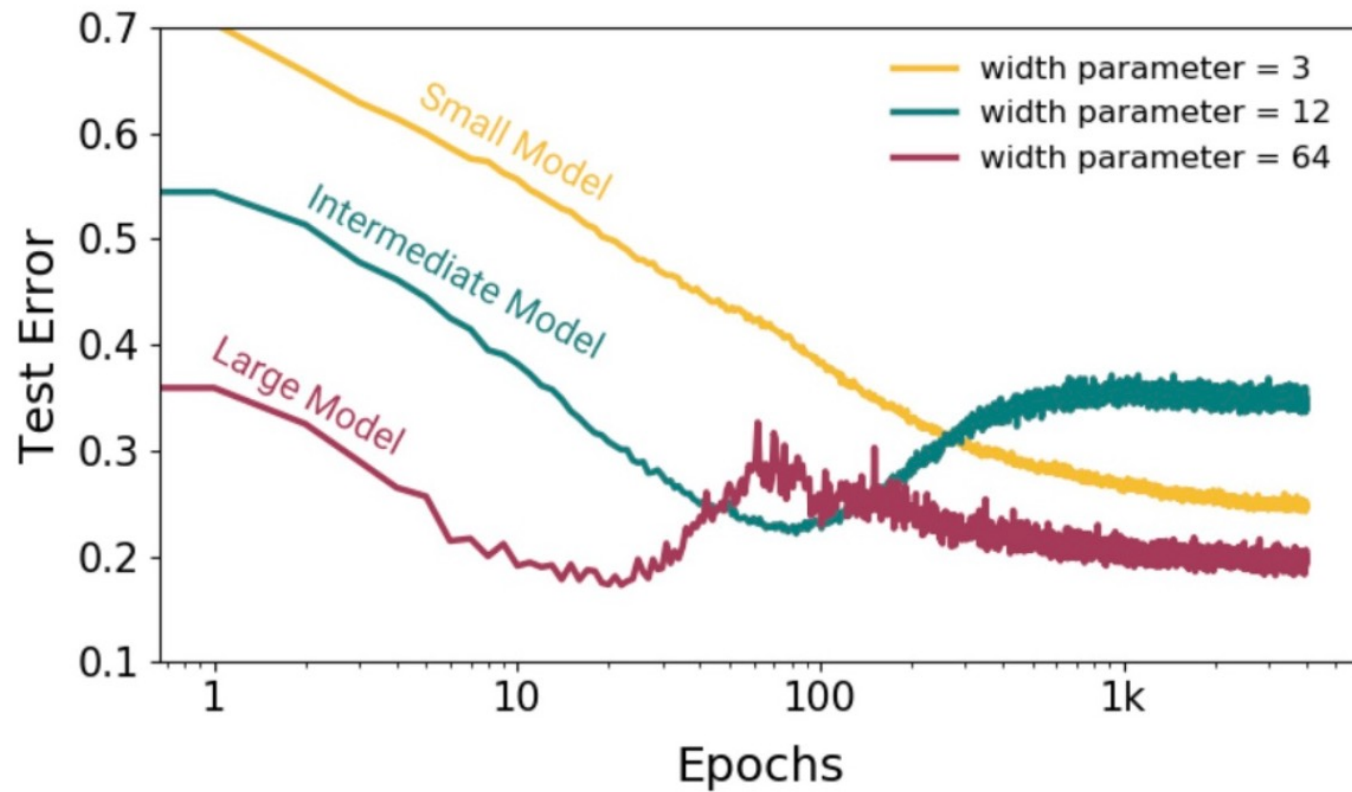
Double descent

- Adding more capacity to the model will have the capability to create smoother functions.



- Why over-parameterized models should produce smooth functions? Maybe:
 - the network initialization encourage smoothness
 - the training algorithm may somehow “prefer” to converge to smooth functions
 - the loss landscape

Epoch-wise double descent



Choosing hyperparameters

- To choose the capacity, the structure, and training algorithm hyperparameters, we fit multiple models and evaluate their performance using a separate validation set.
 - It is computationally expensive since we must train an entire model for each combination of hyperparameters.
- We cannot rely on GD to adjust most hyperparameters as we did for learning the model parameters

Resources

- Understanding Deep Learning, Chapter 8-9.
- Deep Learning Book, Chapter 8.
- Bishop Deep Learning, Chapter 9.