CS 540 Introduction to Artificial Intelligence

Neural Networks III

Stability, Generalization, and Regularization

University of Wisconsin–Madison

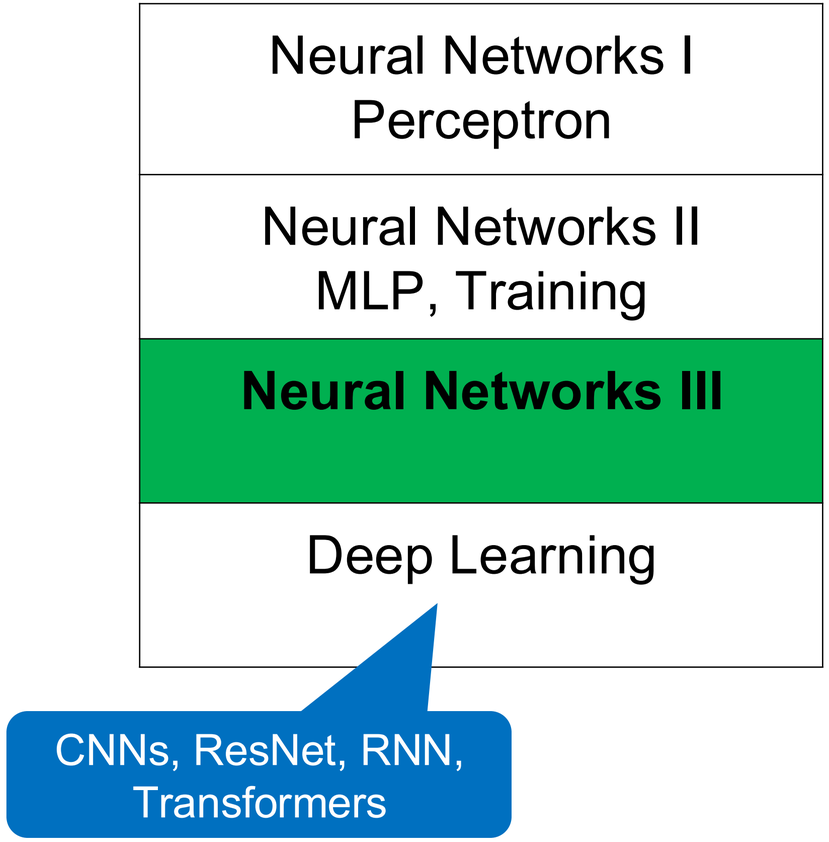
Fall 2025, Section 3

October 15, 2025

Announcements

- HW5 due Friday 10/17 at 11:59 pm
- Please complete midterm course evaluations before Friday
- Midterm exam:

Thursday 10/23
7:30 pm to 9:00 pm
Humanities Building, Room 3650



Midterm Information

- **Time:** Thursday October 23rd 7:30-9 PM
- **Location:**
  - Section 001 : 6210 Social Sciences Building
  - Section 002 : B10 Ingrahm Hall
  - **Section 003: 3650 Humanities Building**

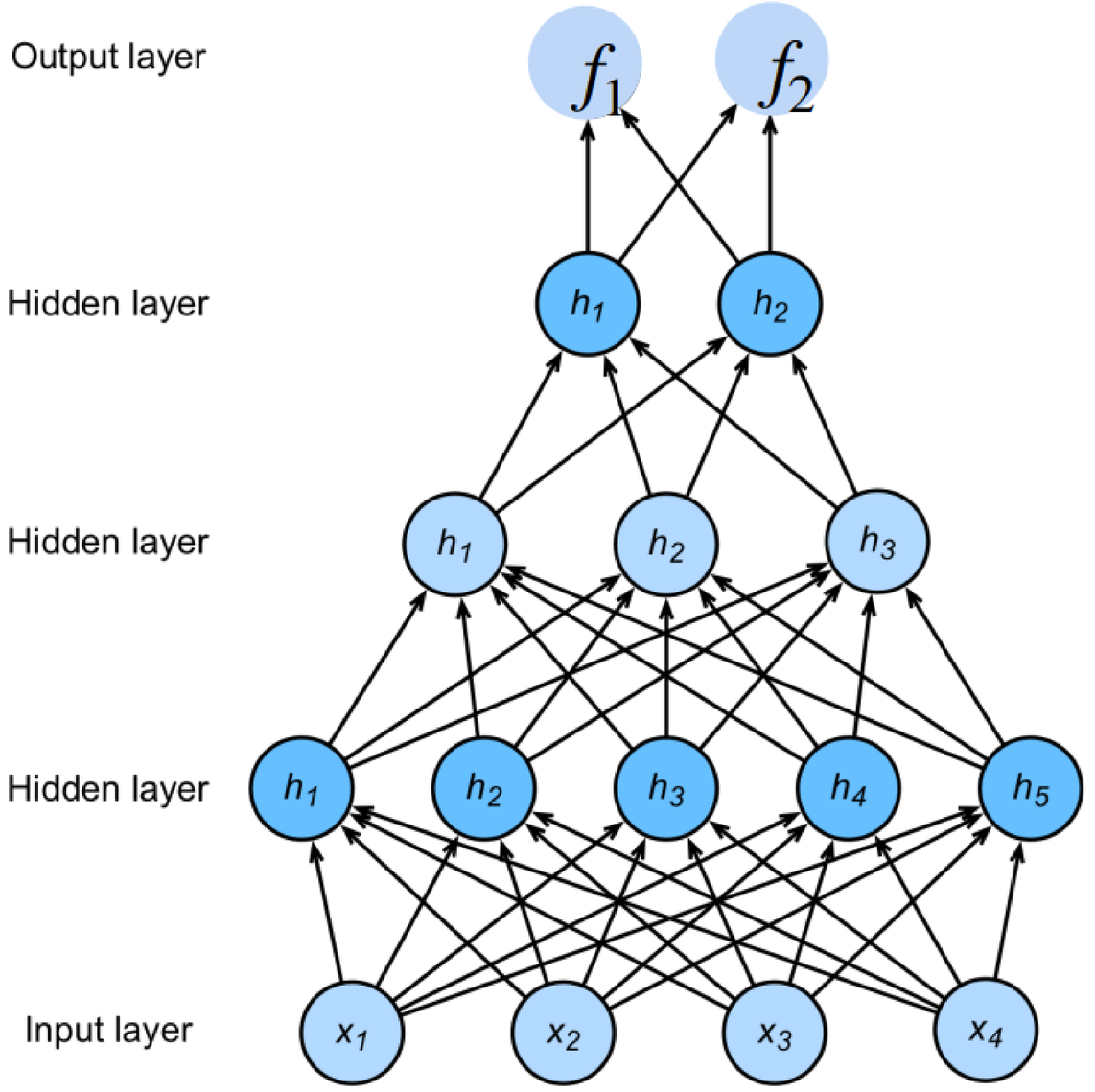
- McBurney students and students requesting alternate: reach out to your instructor if you have not received any email!
- Format: multiple choice
- Cheat sheet: single handwritten piece of paper, front and back
- Calculator: fine if it doesn't have an Internet connection
- Detailed topic list + practice on Piazza and Canvas

Today's goals

- Understand deep neural networks as **computational graphs**.
  - Forward propagation of inputs to outputs.
  - Backward propagation of loss gradients to weights and biases.
- Understand **numerical stability** issues in training neural networks.
  - Vanishing or exploding gradients.
- Review **generalization** and understand how to use **regularization** for better generalization.
  - Overfitting, underfitting
  - Weight decay and dropout
  - Data augmentation

Part I: Neural Networks as a Computational Graph

Deep neural networks (DNNs)



$$\mathbf{h}_1 = \sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

$$\mathbf{h}_2 = \sigma(\mathbf{W}^{(2)}\mathbf{h}_1 + \mathbf{b}^{(2)})$$

$$\mathbf{h}_3 = \sigma(\mathbf{W}^{(3)}\mathbf{h}_2 + \mathbf{b}^{(3)})$$

$$\mathbf{f} = \mathbf{W}^{(4)}\mathbf{h}_3 + \mathbf{b}^{(4)}$$

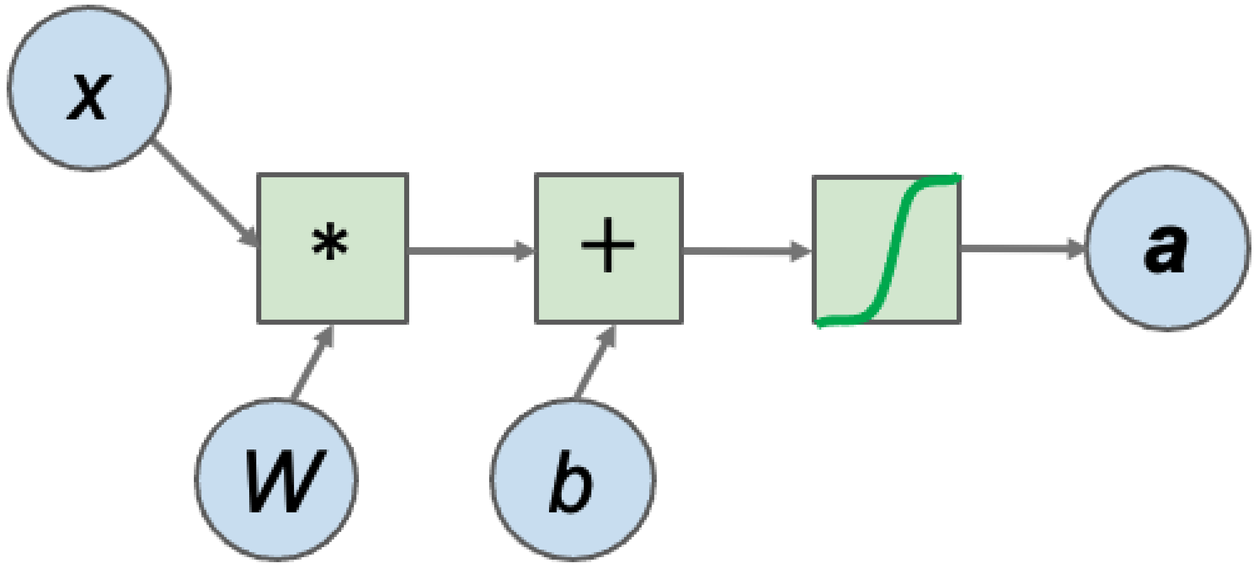
$$\mathbf{p} = \text{softmax}(\mathbf{f})$$

NNs are composition of nonlinear functions

Neural networks as variables + operations

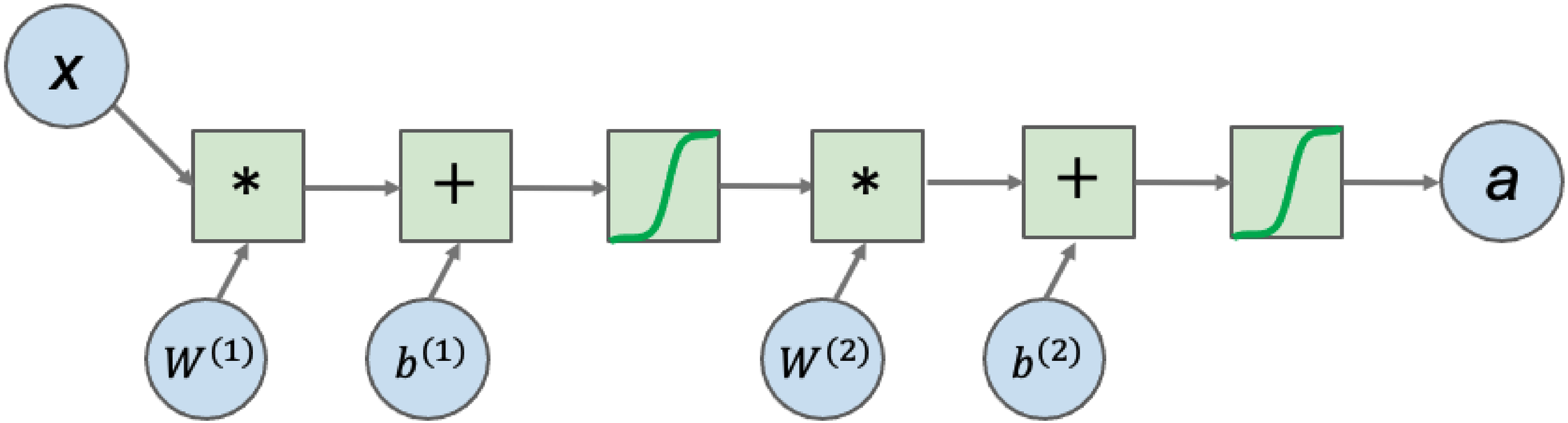
$$\mathbf{a} = sigmoid(\mathbf{W}\mathbf{x} + \mathbf{b})$$

- Can describe with a **computational graph**
- Decompose functions into atomic operations
- Separate data (variables) and computing (operations)



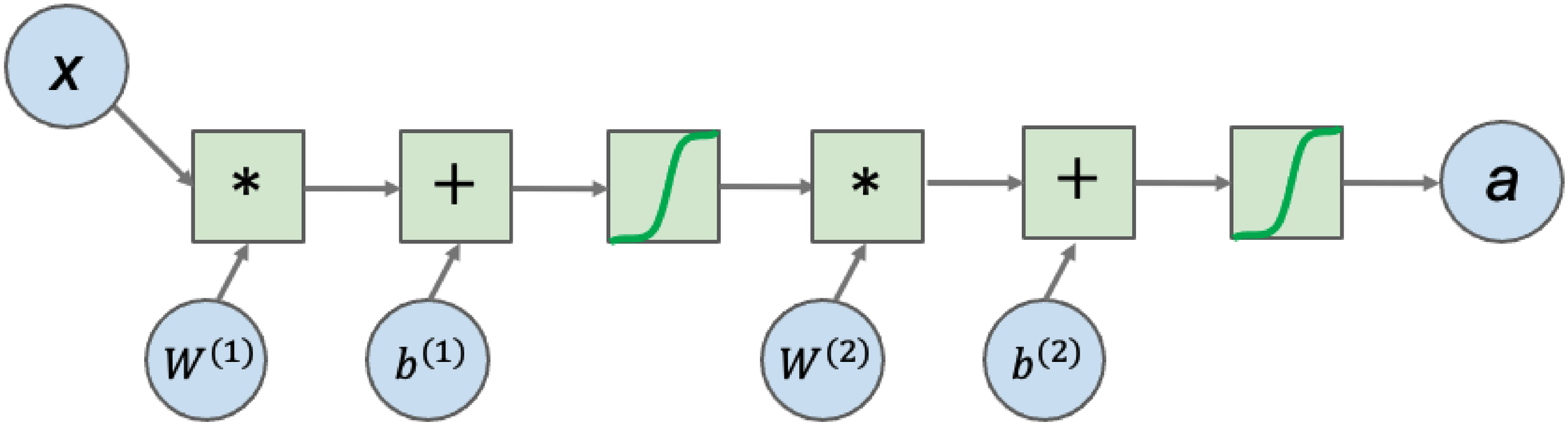
Neural networks as a computational graph

- A two-layer neural network



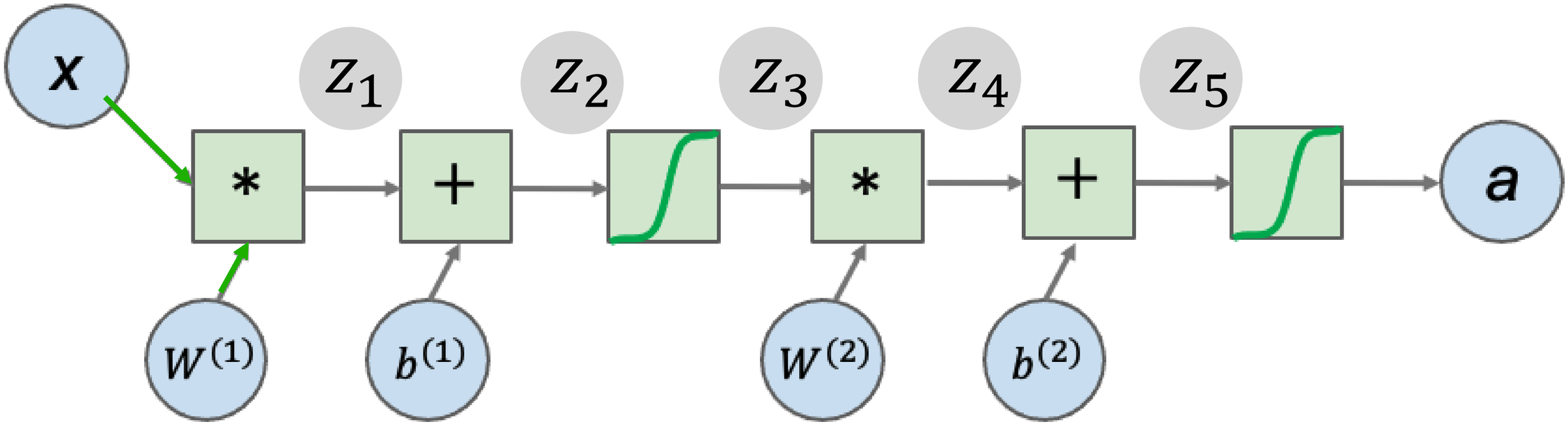
Neural networks as a computational graph

- A two-layer neural network
- Forward propagation vs. backward propagation



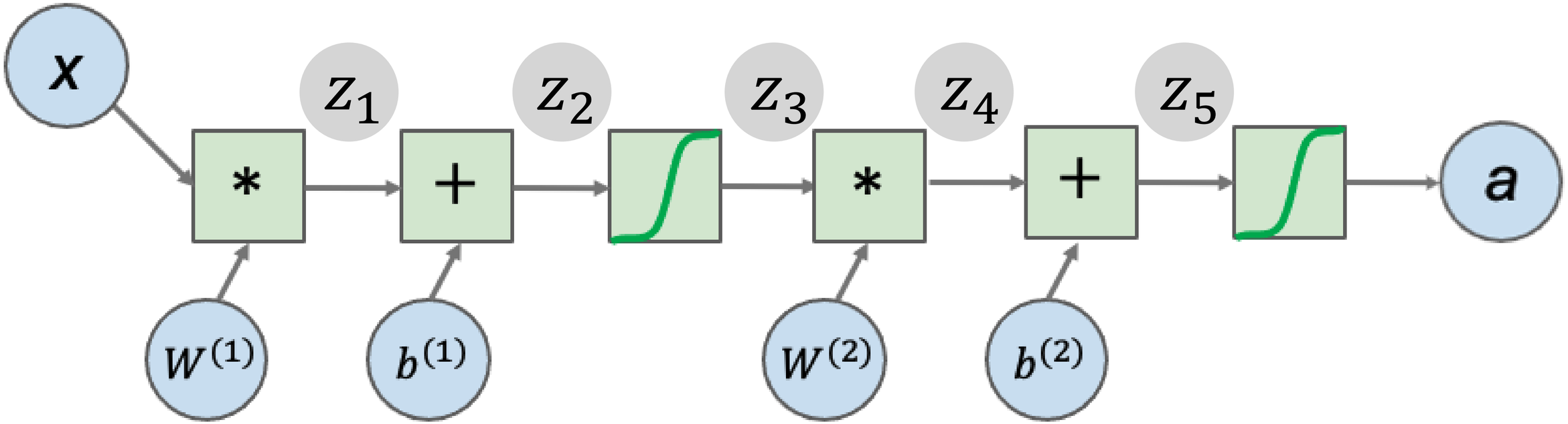
Neural networks: forward propagation

- A two-layer neural network
- Intermediate variables Z



Neural networks: backward propagation

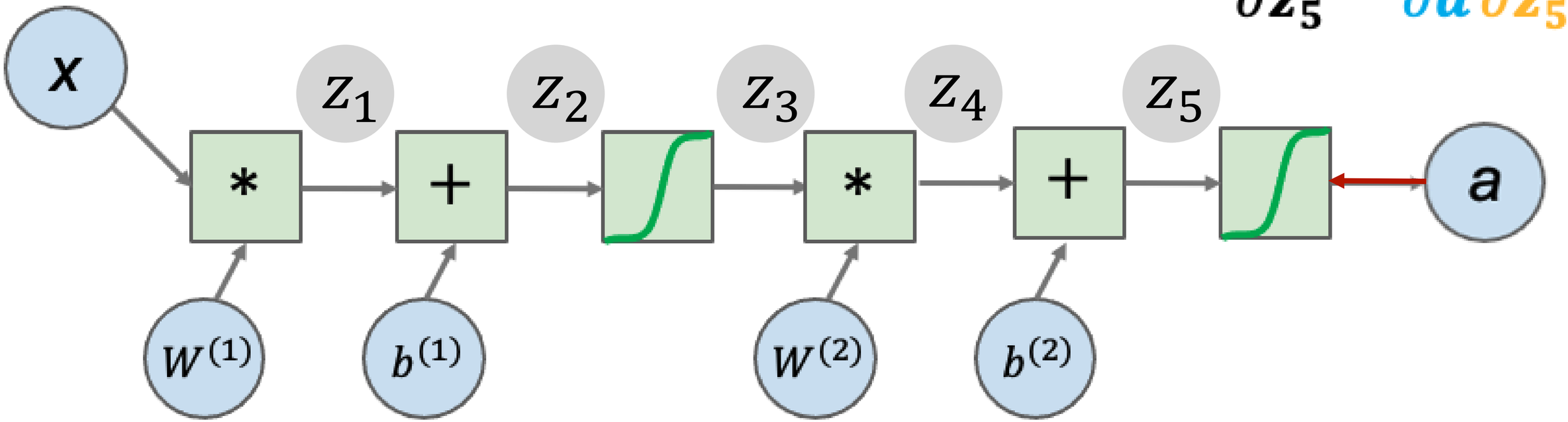
- A two-layer neural network
- Assuming forward propagation is done
- Minimize a **loss function** L



Neural networks: backward propagation

- A two-layer neural network
- Assuming forward propagation is done
- Minimize a **loss function** L

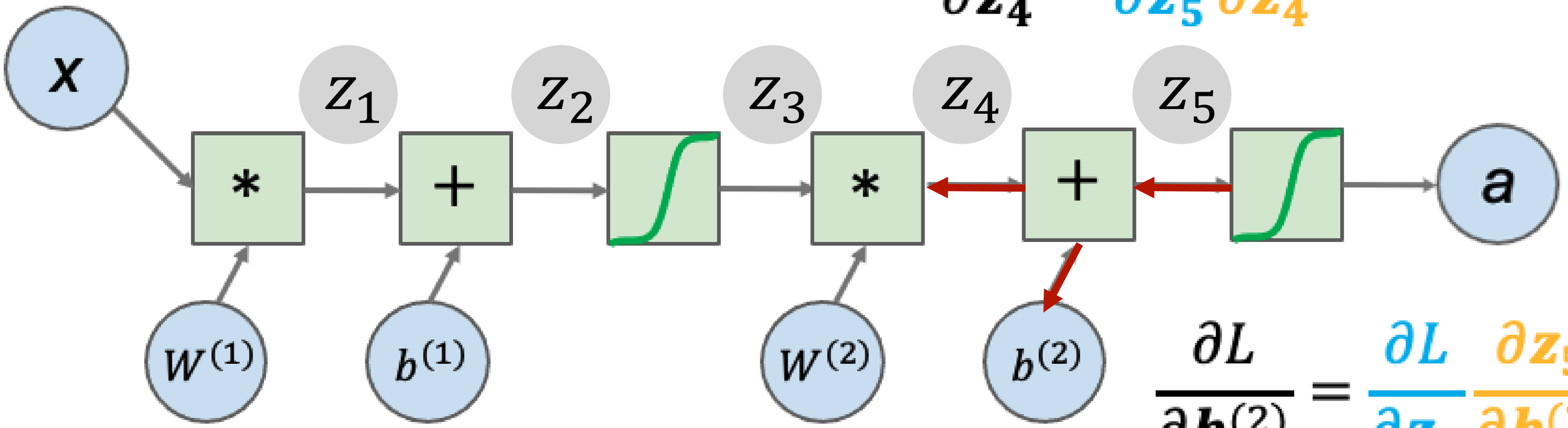
$$\frac{\partial L}{\partial \boldsymbol{z_5}} = \frac{\partial L}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial z_5}$$



Neural networks: backward propagation

- A two-layer neural network
- Assuming forward propagation is done
- Minimize a **loss function** L

$$\frac{\partial L}{\partial \boldsymbol{z_4}} = \frac{\partial L}{\partial \boldsymbol{z_5}} \frac{\partial z_5}{\partial z_4}$$

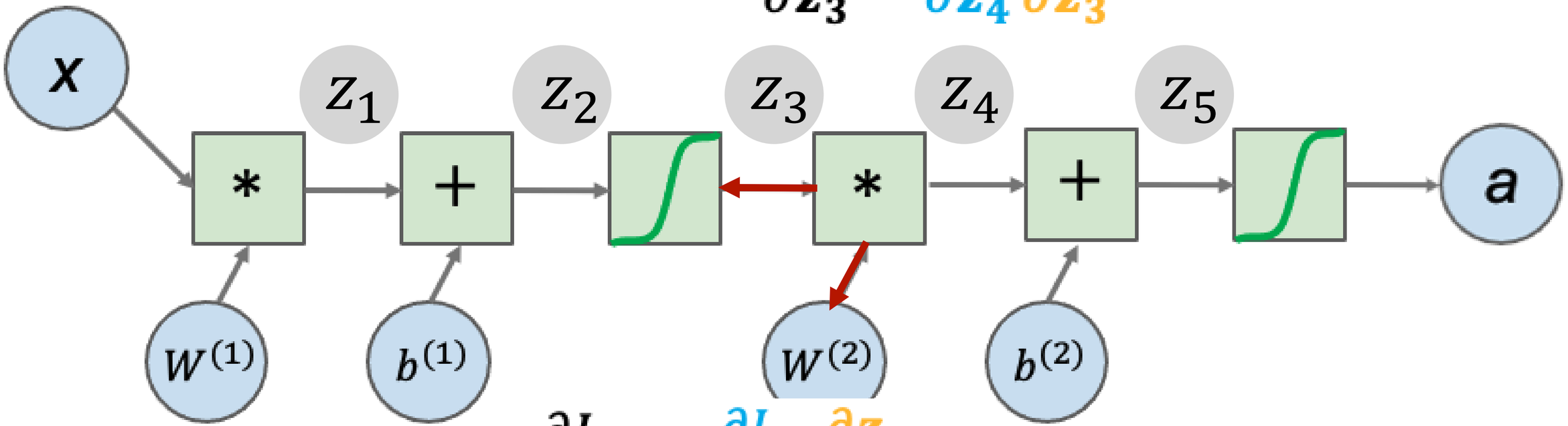


$$\frac{\partial L}{\partial \boldsymbol{b}^{(2)}} = \frac{\partial L}{\partial \boldsymbol{z_5}} \frac{\partial z_5}{\partial \boldsymbol{b}^{(2)}}$$

Neural networks: backward propagation

- A two-layer neural network
- Assuming forward propagation is done

$$\frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial z_4}\frac{\partial z_4}{\partial z_3}$$



$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial z_4}\frac{\partial z_4}{\partial W^{(2)}}$$

Backward propagation: A modern treatment

- First, define a neural network as a computational graph
  - Nodes are variables and operations.
- Must be a directed graph
- All operations must be differentiable.
- Backpropagation computes partial derivatives starting from the loss and then working backwards through the graph.

Backward propagation: PyTorch

```
for t in range(2000):

    # Forward pass: compute predicted y by passing x to the
    # override the __call__ operator so you can call them l
    # doing so you pass a Tensor of input data to the Modul
    # a Tensor of output data.
    y_pred = model(xx)

    # Compute and print loss. We pass Tensors containing th
    # values of y, and the loss function returns a Tensor c
    # loss.
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradient of the loss with resp
    # parameters of the model. Internally, the parameters c
    # in Tensors with requires_grad=True, so this call will
    # all learnable parameters in the model.
    loss.backward()

    # Update the weights using gradient descent. Each param
    # we can access its gradients like we did before.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

Forward propagation

Backward propagation

Gradient Descent

Part II: Numerical Stability

Gradients for Neural Networks

- 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 *many* matrices

Wikipedia

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

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

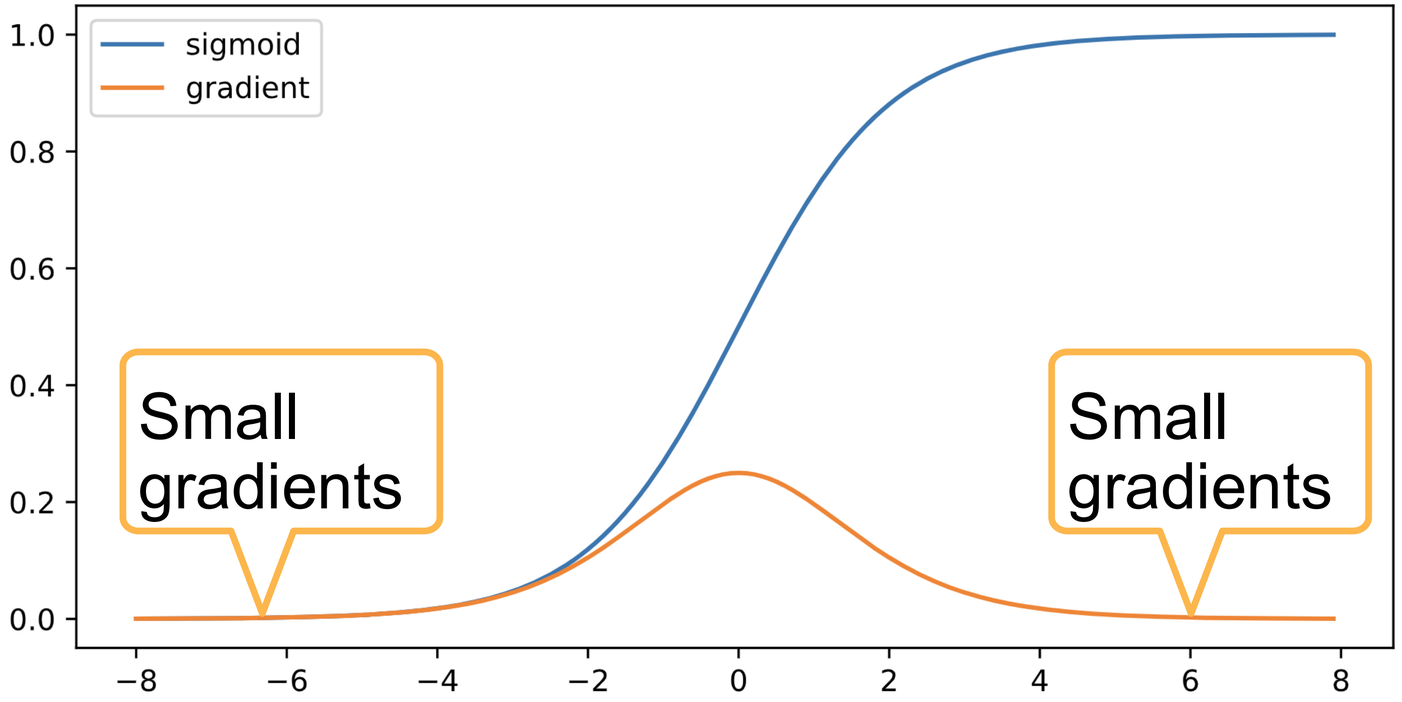
Issues with Gradient Exploding

- Value out of range: infinity value (NaN)
- Sensitive to learning rate (LR)
  - Not small enough LR → larger gradients
  - Too small LR → No progress
  - May need to change LR dramatically during training

Gradient Vanishing

- Use sigmoid as the activation function

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



Issues with Gradient Vanishing

- Gradients with value 0
- No progress in training
  - No matter how to choose learning rate
- Severe with bottom layers (those near the input)
  - Only top layers (near output) are well trained
  - No benefit to make networks deeper

How to stabilize training?

Stabilize Training: Practical Considerations

- Goal: make sure gradient values are in a proper range
  - E.g. in [1e-6, 1e3]
- Multiplication → plus
  - Architecture change (e.g., ResNet)
- Normalize
  - Batch Normalization, Gradient clipping
- Proper activation functions

Part III: Generalization & Regularization

How good are the models?

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)

Underfitting
Overfitting

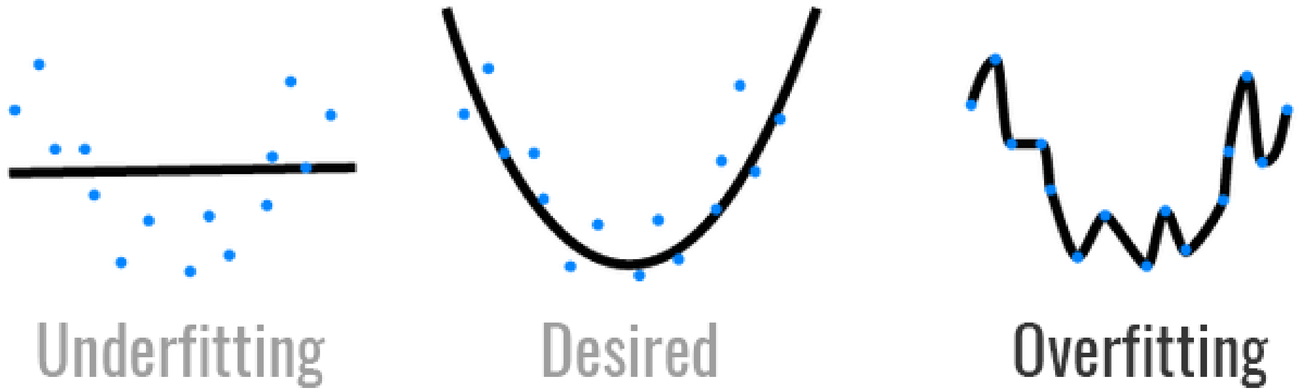
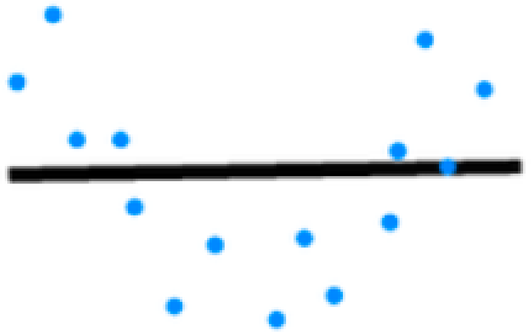
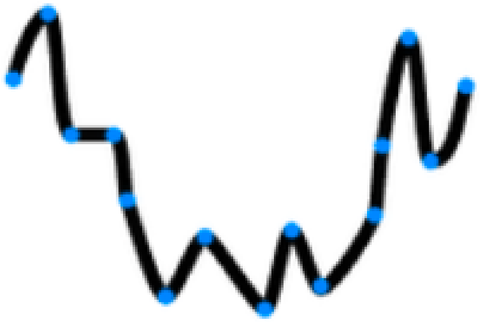


Image credit: hackernoon.com

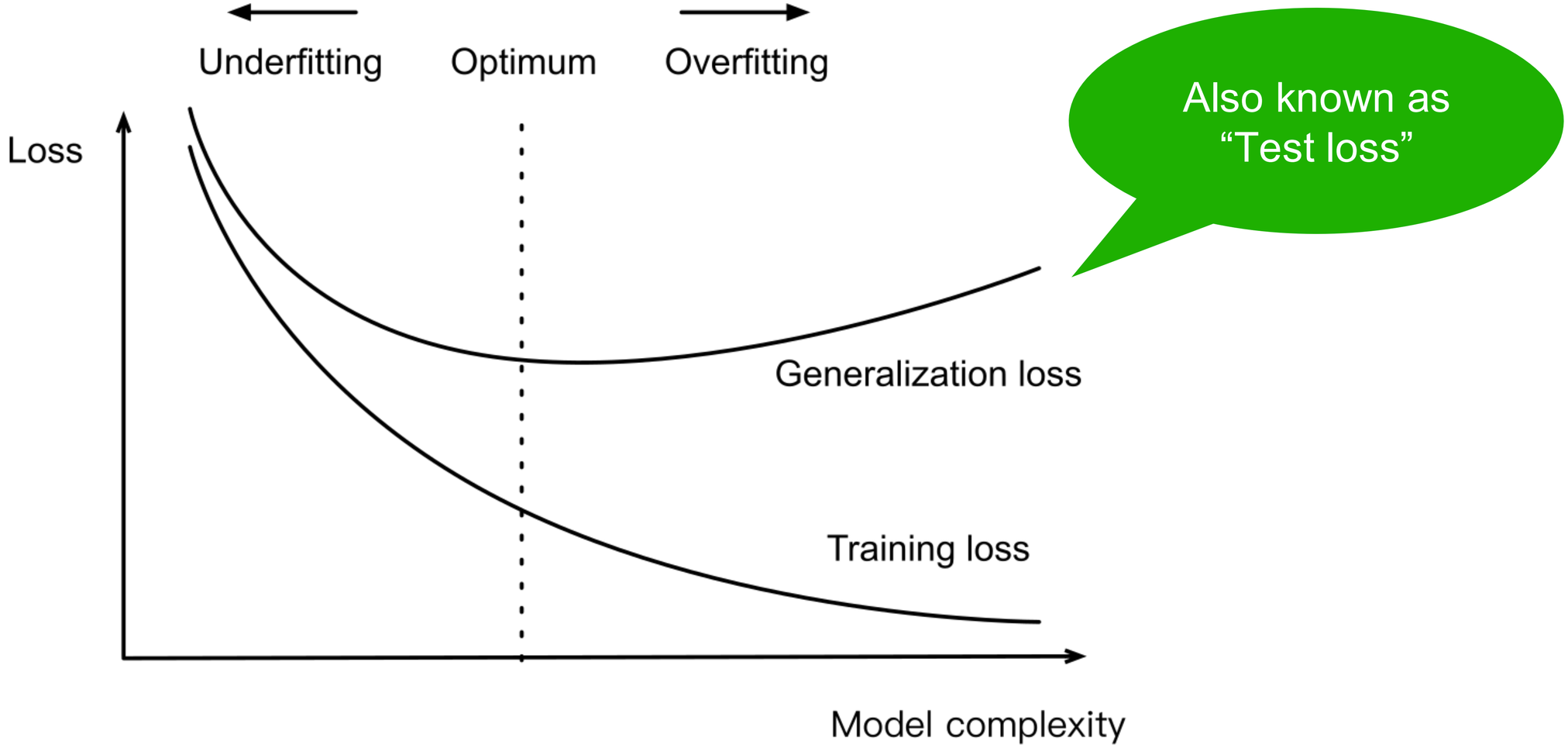
Model Capacity

- 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



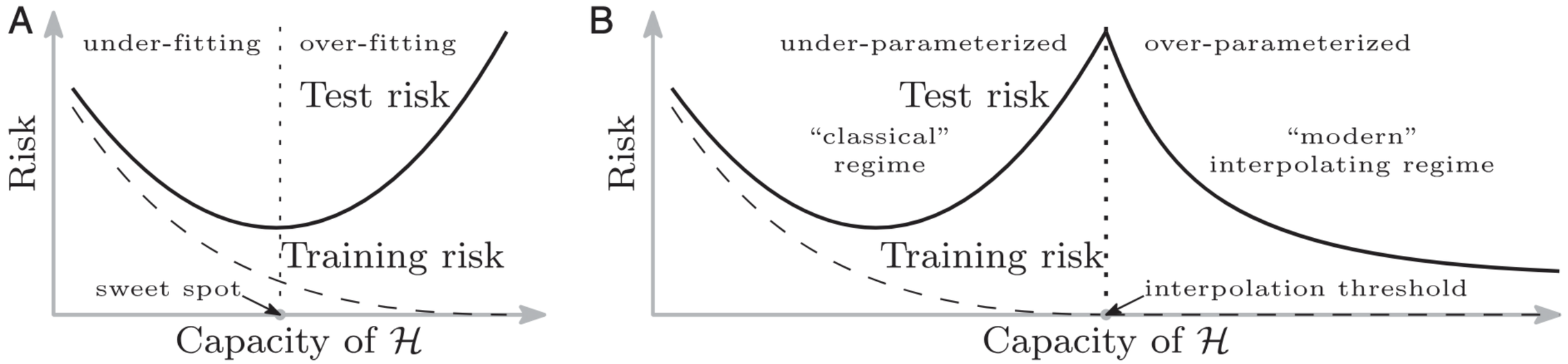


Classical View of Model Complexity



* Recent research has challenged this view for some types of models.

A Modern View: Double Descent

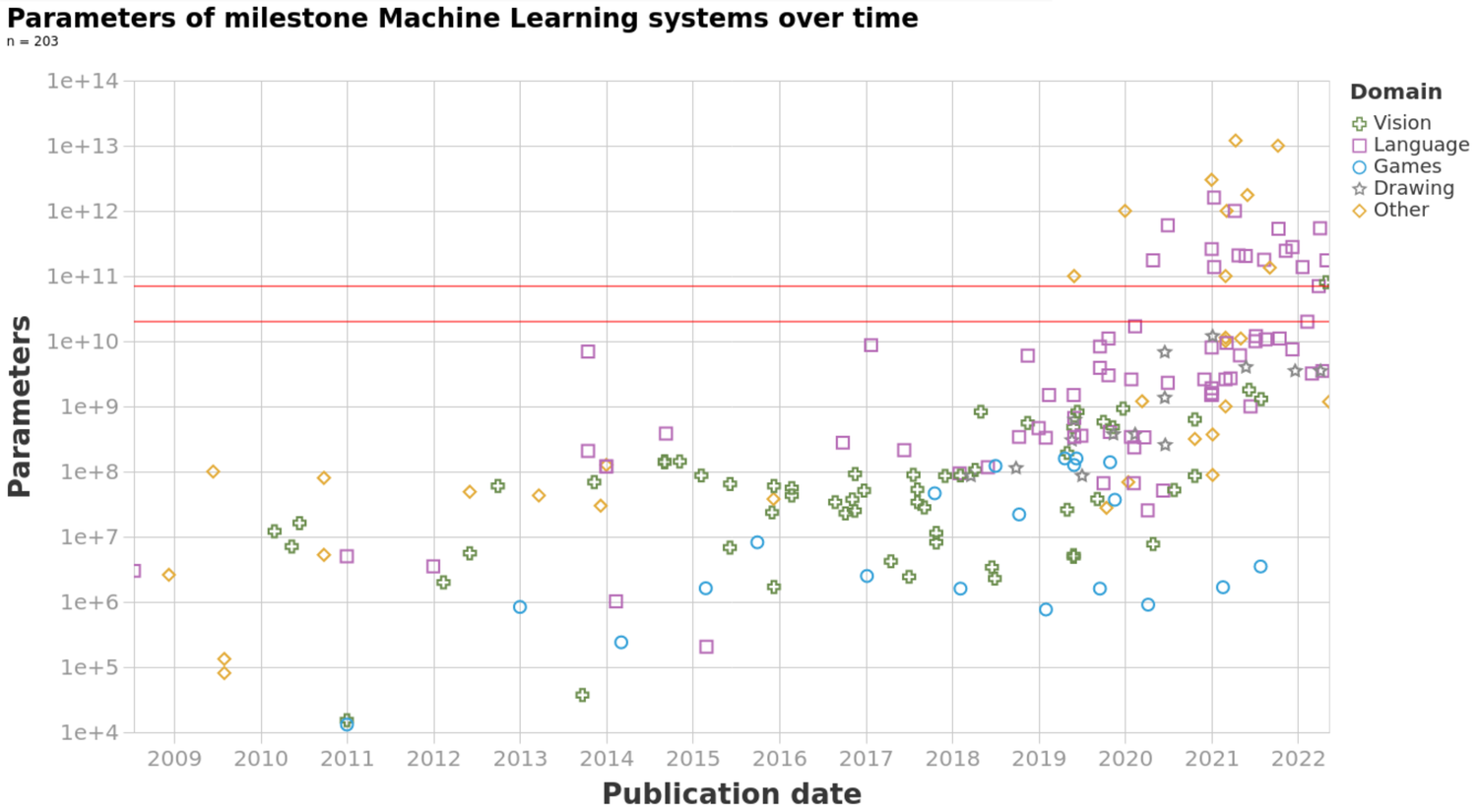


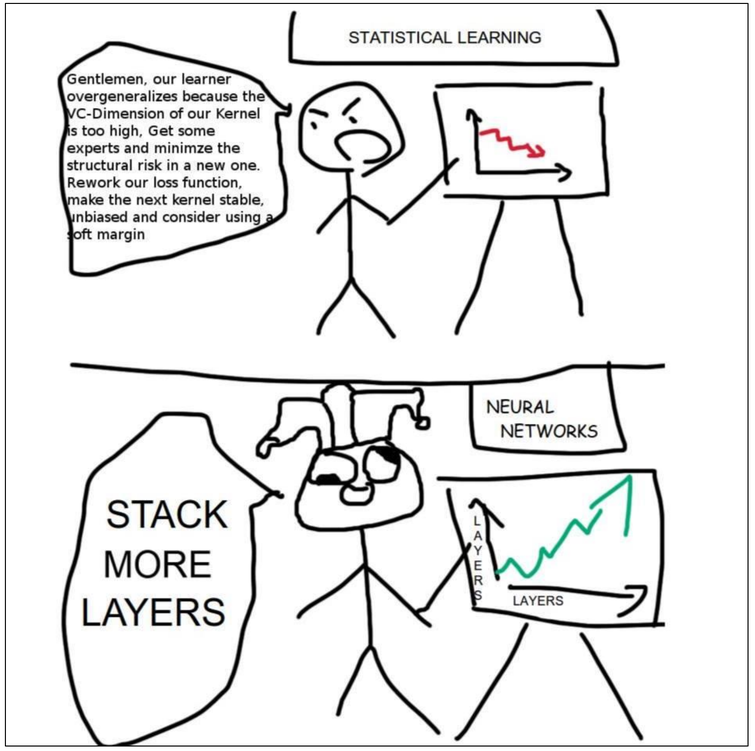
How many layers to use?
How wide should the layers be?
How big should my network be?

The ultimate measure is how well the network performs on new data

Belkin, Hsu, Ma, Mandal 2019

At the frontier, bigger is better



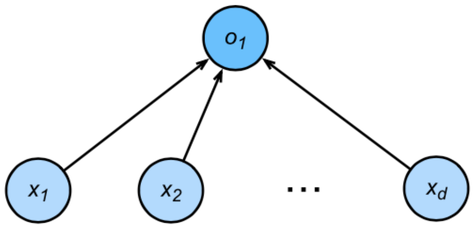


https://epoch.ai/blog/machine-learning-model-sizes-and-the-parameter-gap

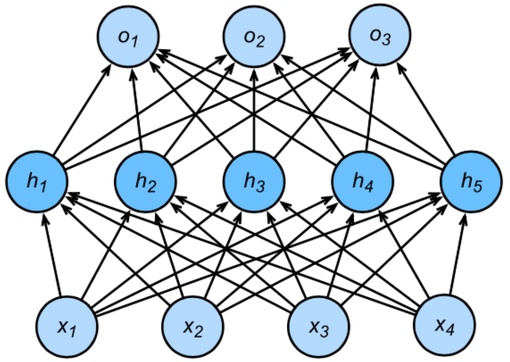
Estimate Neural Network Capacity

- It's hard to compare complexity between different families of models.
  - e.g. K-NN vs neural networks
- Given a model family, two main factors matter:
  - The number of parameters
  - The values taken by each parameter

$d + 1$

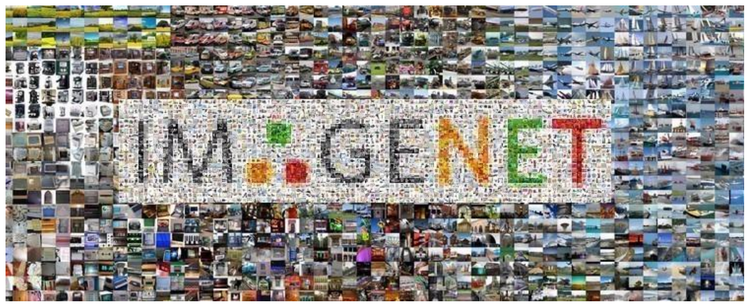


$(d + 1)m + (m + 1)k$



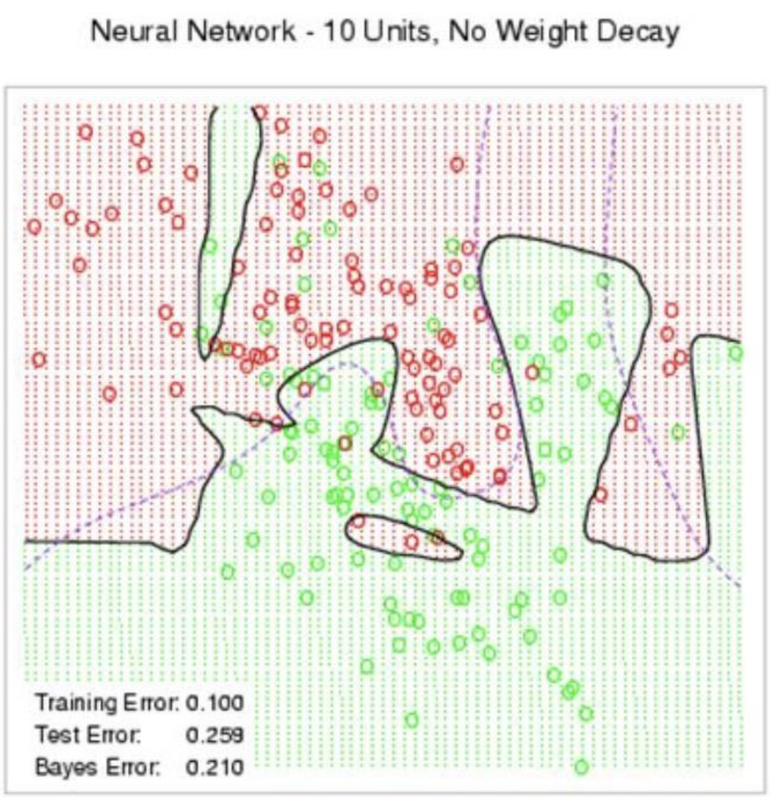
Data Complexity

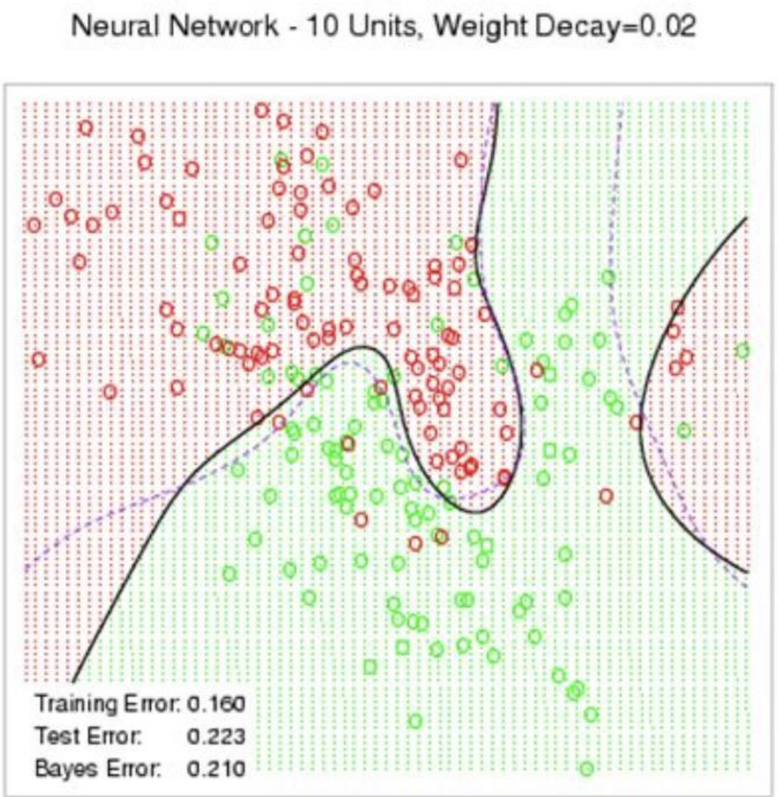
- Multiple factors matters
  - # of examples
  - # of features in each example
  - time/space structure
  - # of labels



How to regularize the model for better generalization?

Weight Decay



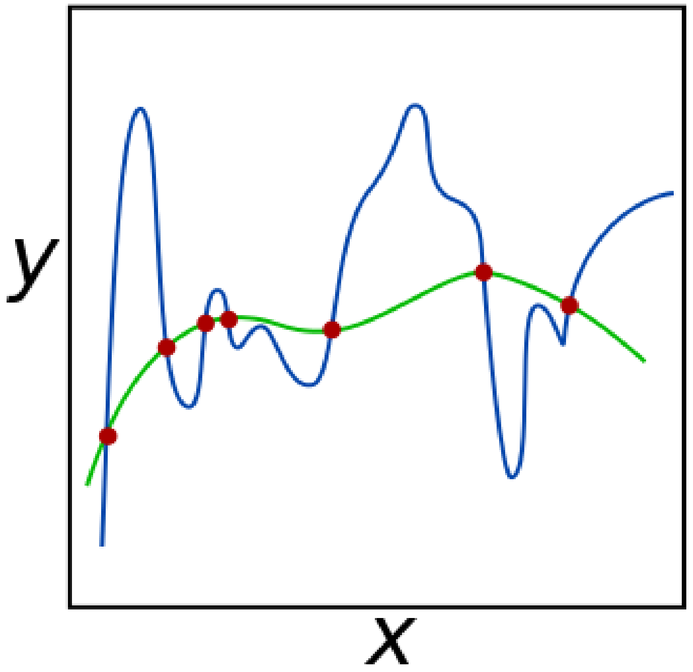


Squared Norm Regularization as Hard Constraint

- Reduce model complexity by limiting value range

$$minL(\mathbf{w}, b) subject\ to \parallel \mathbf{w} \parallel^2 \leq B$$

  - Often do not regularize bias b
    - Doing or not doing has little difference in practice
  - A small B means more regularization



Squared Norm Regularization as Soft Constraint

- We can rewrite the hard constraint version as

$$minL(\mathbf{w}, b) + \frac{\lambda}{2} \| \mathbf{w} \|^2$$

Squared Norm Regularization as Soft Constraint

- We can rewrite the hard constraint version as

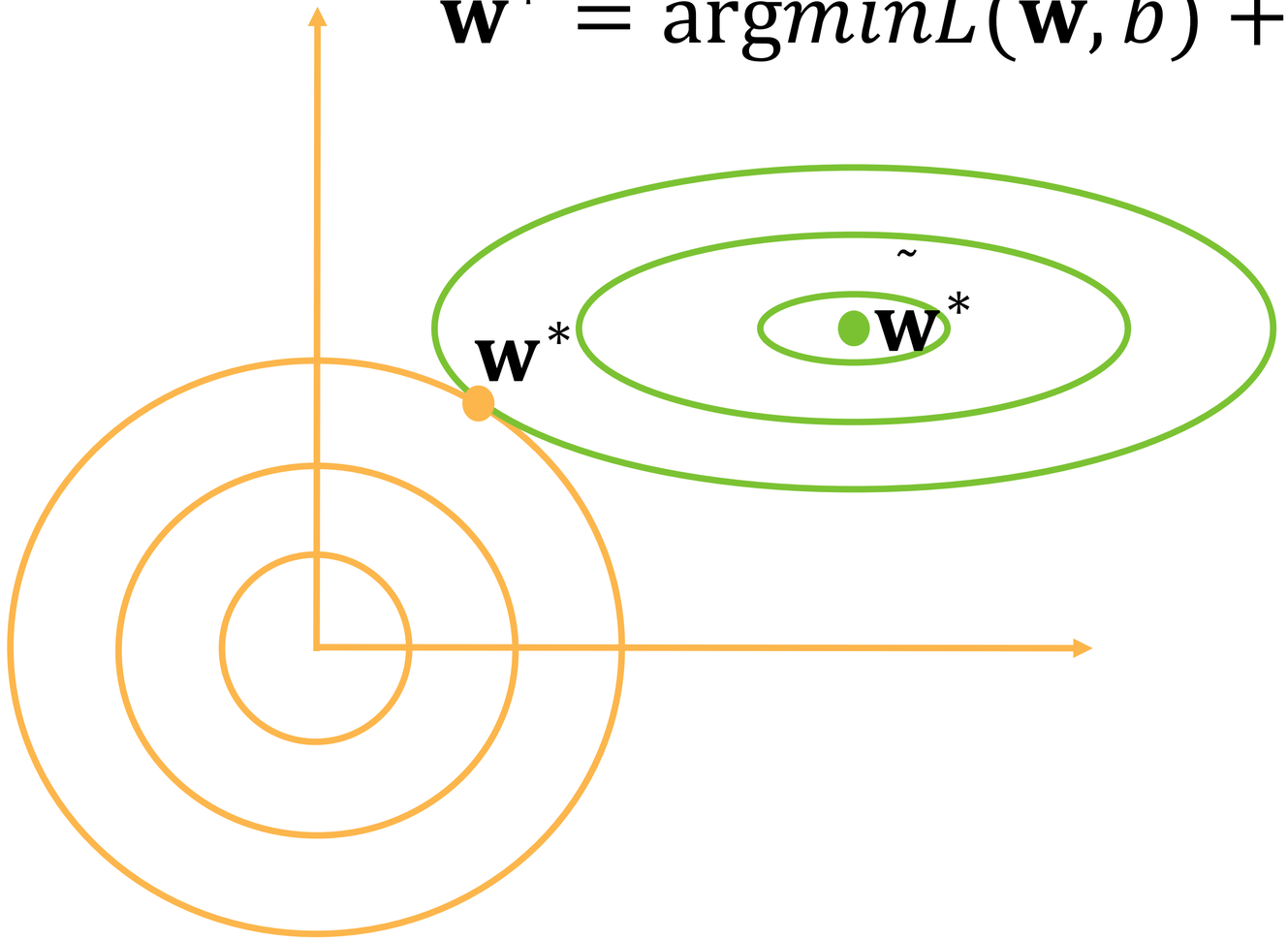
$$minL(\mathbf{w}, b) + \frac{\lambda}{2} \| \mathbf{w} \|^2$$

  - Hyper-parameter λ controls regularization importance
  - $\lambda = 0$: no effect

    $\lambda \to \infty, \mathbf{w}^* \to \mathbf{0}$

Illustrate the Effect on Optimal Solutions

$$\mathbf{w}^* = \arg min L(\mathbf{w}, b) + \frac{\lambda}{2} \| \mathbf{w} \|^2$$



$$\tilde{\mathbf{w}}^* = \arg min L(\mathbf{w}, b)$$

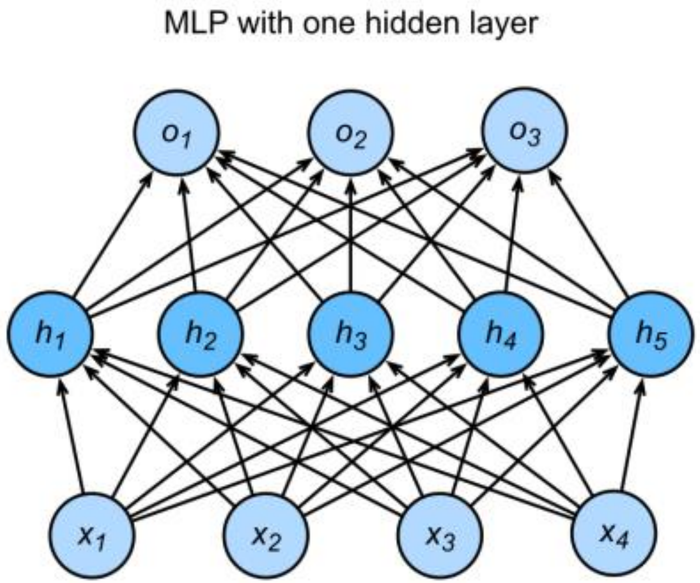
Dropout

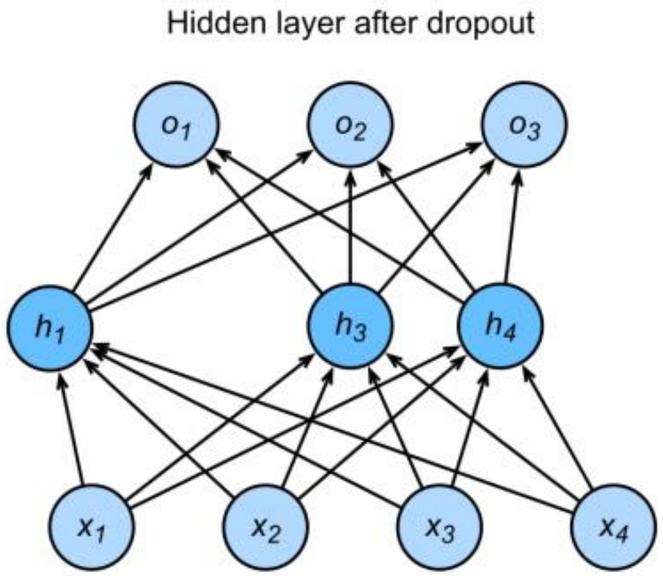
Hinton et al.

Apply Dropout

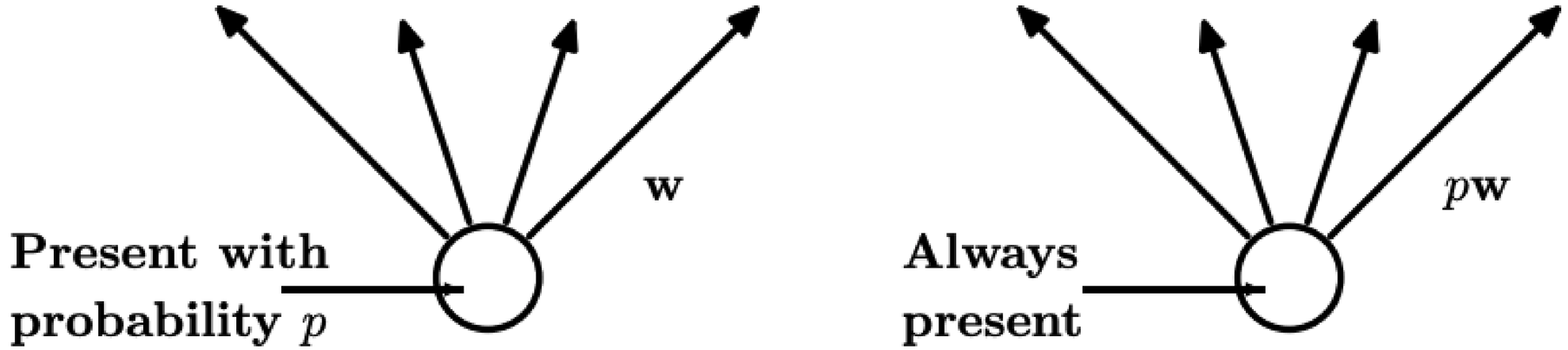
- Often apply dropout on the output of hidden fully-connected layers

$$\mathbf{h} = \sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$
$$\mathbf{h}' = \text{dropout}(\mathbf{h})$$
$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}' + \mathbf{b}^{(2)}$$
$$\mathbf{p} = \text{softmax}(\mathbf{o})$$





Dropout



(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

Dropout

Hinton et al.

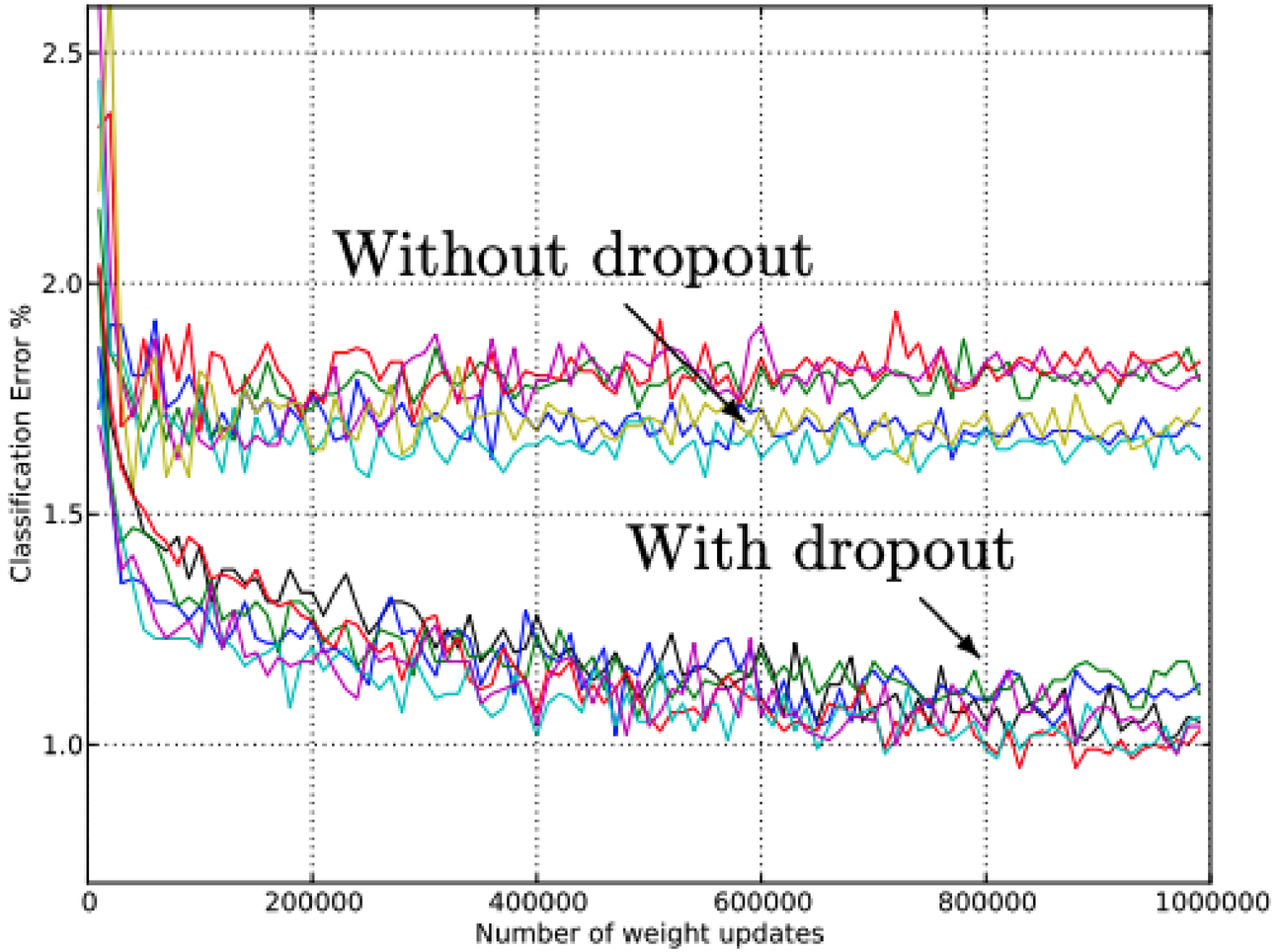
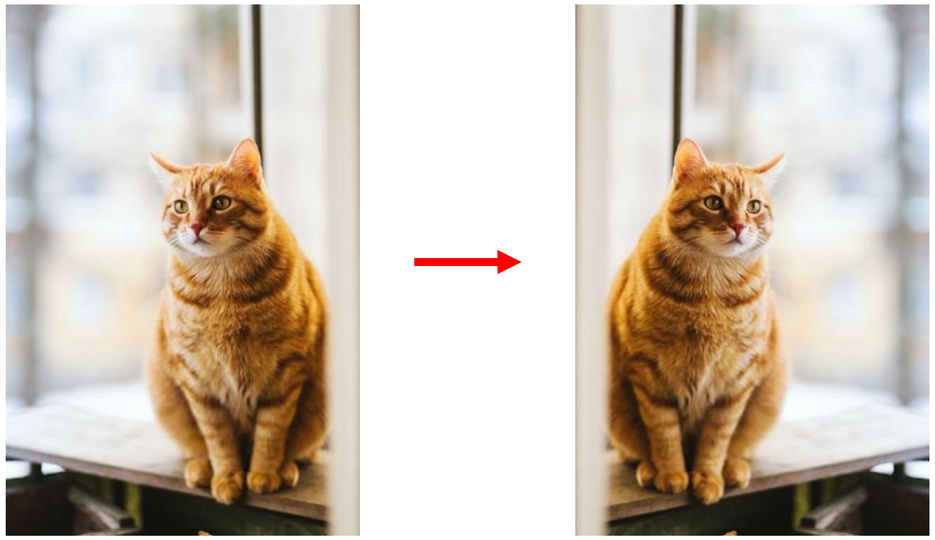


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Data Concerns

What if we don’t have a lot of data?

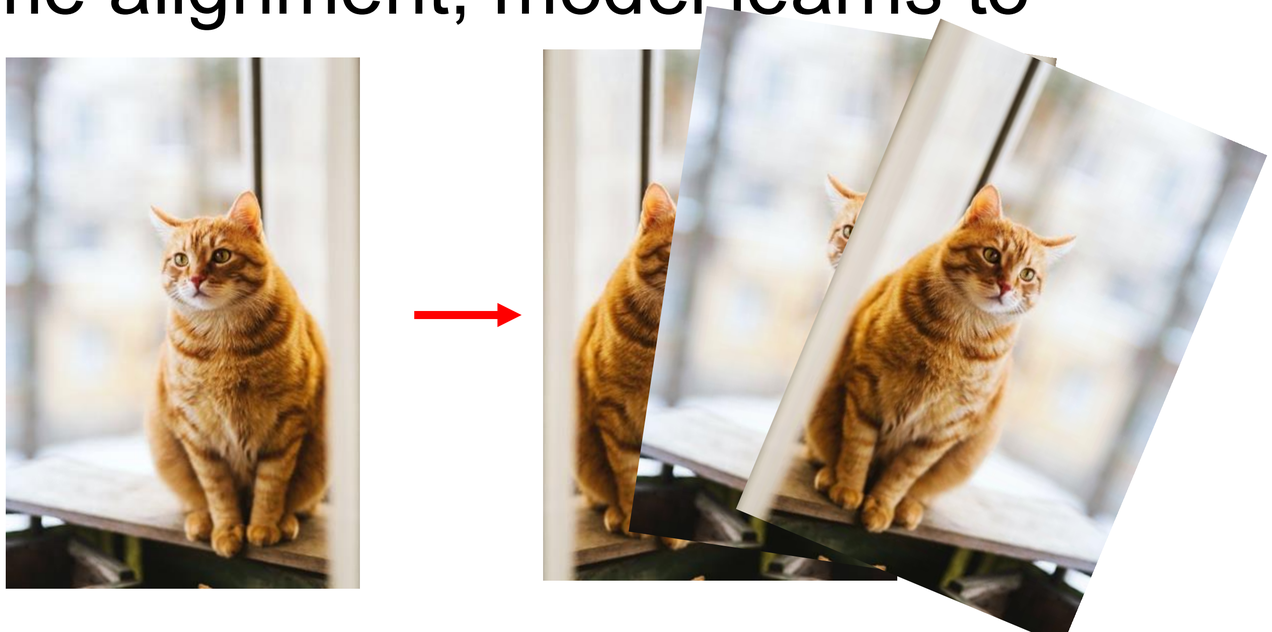
- We risk overfitting
- Avoiding overfitting: **regularization** methods
- Data augmentation: a classic way to regularize



Data Augmentation

Augmentation: transform + add new samples to dataset

- Transformations: based on domain
- Idea: build **invariances** into the model
  - **Ex**: if all images have same alignment, model learns to use it
- Keep the label the same!

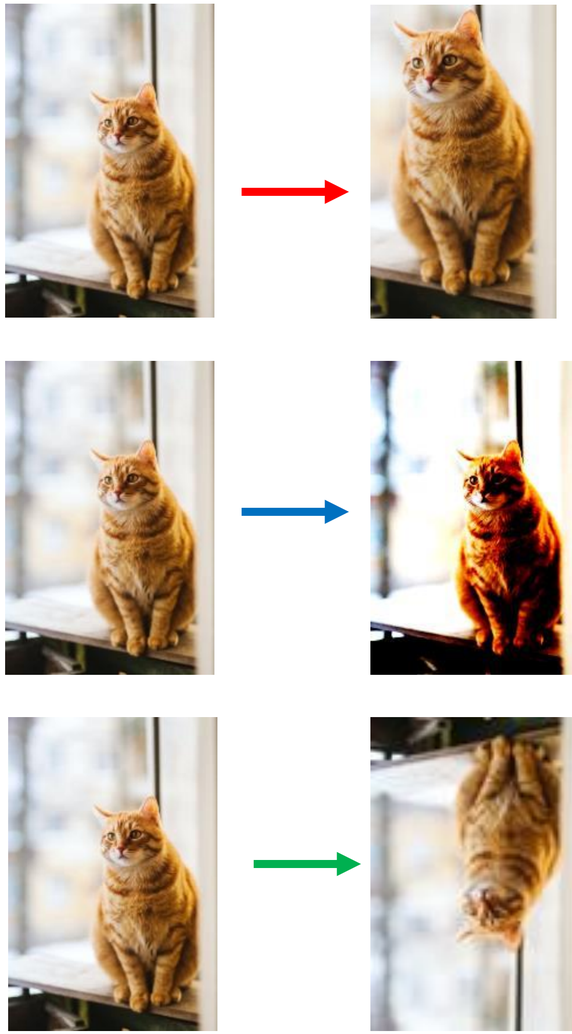


Transformations

Examples of transformations for images

- **Crop** (and zoom)
- **Color** (change contrast/brightness)
- **Rotations+** (translate, stretch, shear, etc)

Many more possibilities. Combine as well!



Q: how to deal with this at **test time**?

- A: transform, test, average

Other Domains

Not just for image data. For example, on text:

- Substitution
  - E.g., “It is a **great** day” ➔ “It is a **wonderful** day”
  - Use a thesaurus for particular words
  - Or use a model. Pre-trained word embeddings, language models
- Back-translation
  - “Given the low budget and production limitations, this movie is very good.” ➔ “There are few budget items and production limitations to make this film a really good one”

What we've learned today…

- Deep neural networks
  - Computational graph (forward and backward propagation)
- Numerical stability in training
  - Gradient vanishing/exploding
- Generalization and regularization
  - Overfitting, underfitting
  - Weight decay and dropout
  - Data augmentation