STAT 453: Introduction to Deep Learning and Generative Models

Ben Lengerich

Lecture 14: Review

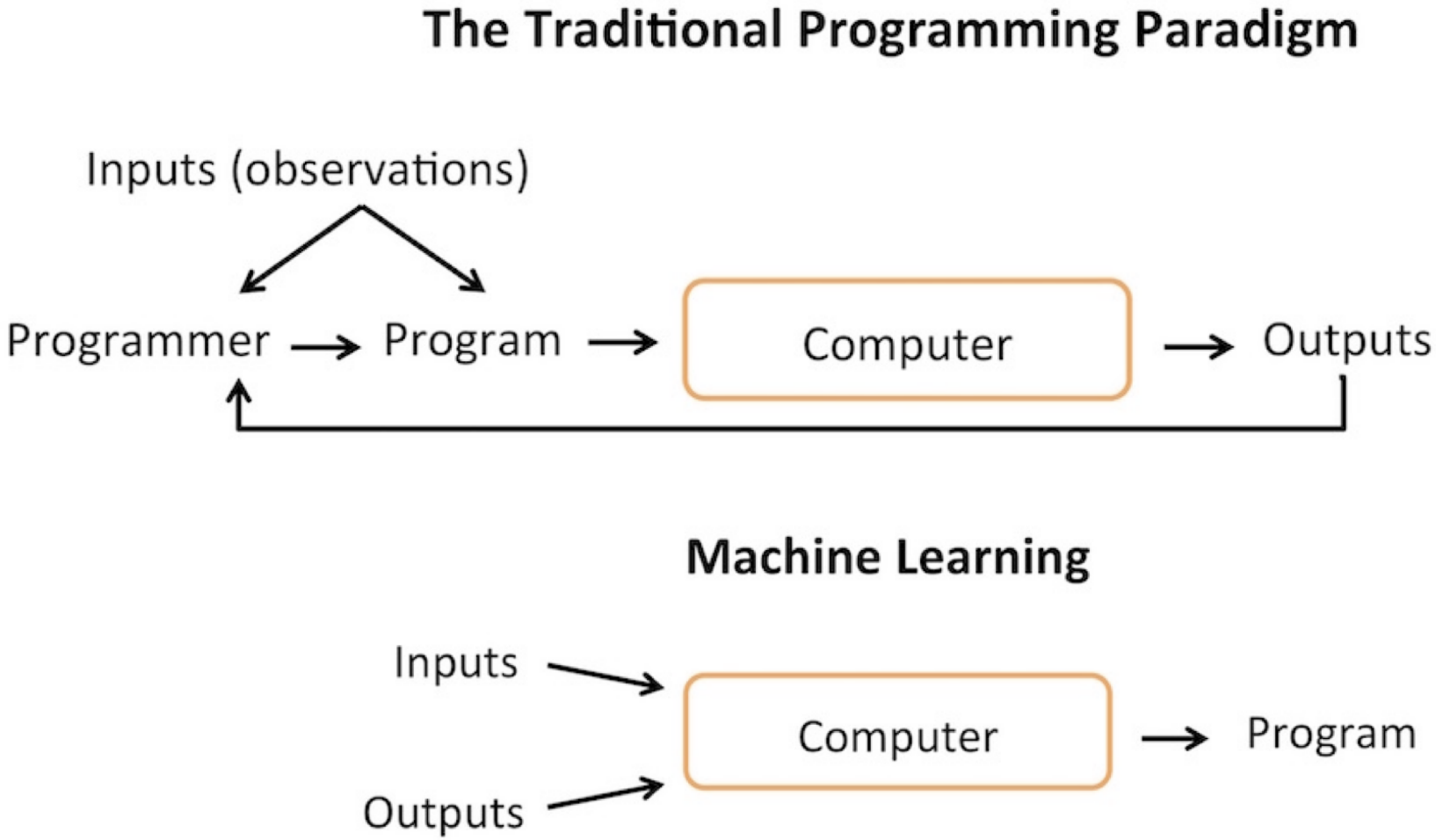
October 20, 2025

Course Schedule / Calendar

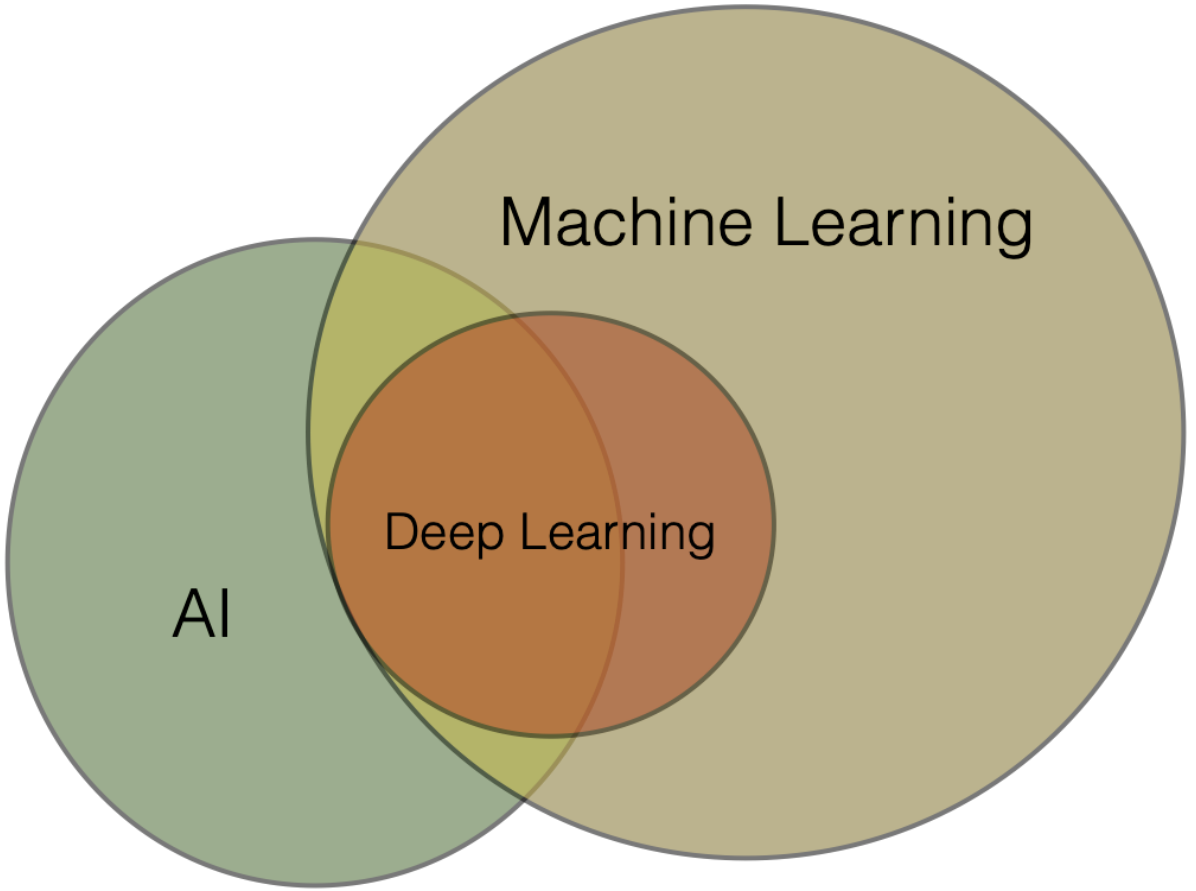
Week	Lecture Dates	Topic	Assignments
Module 1: Introduction and Foundations			
1	9/3	Course Introduction	
2	9/8, 9/10	A Brief History of DL, Statistics / linear algebra / calculus review	HW1
3	9/15, 9/17	Single-layer networks Parameter Optimization and Gradient Descent	
4	9/22, 9/24	Automatic differentiation with PyTorch, Cluster and cloud computing resources	HW 2
Module 2: Neural Networks			
5	9/29, 10/1	Multinomial logistic regression, Multi-layer perceptrons and backpropagation	
6	10/6, 10/8	Regularization Normalization / Initialization	HW 3
7	10/13, 10/15	Optimization, Learning Rates CNNs	Project Proposal
8	10/20, 10/22	Review, **Midterm Exam**	In-class Exam

Week	Lecture Dates	Topic	Assignments
Module 3: Intro to Generative Models			
9	10/27, 10/29	A Linear Intro to Generative Models, Factor Analysis, Autoencoders, VAEs	
10	11/3, 11/5	Generative Adversarial Networks, Diffusion Models	Project Midway Report
Module 4: Large Language Models			
11	11/10, 11/12	Sequence Learning with RNNs Attention, Transformers	HW4
12	11/17, 11/19	GPT Architectures, Unsupervised Training of LLMs	
13	11/24, 11/26	Supervised Fine-tuning of LLMs, Prompts and In-context learning	HW5
14	12/1, 12/3	Foundation models, alignment, explainability Open directions in LLM research	
15	12/8, 12/10	**Project Presentations**	Project Final Report
16	12/17	**Final Exam**	Final Exam

What is Machine Learning?



The Connection Between Fields



What is Machine Learning?

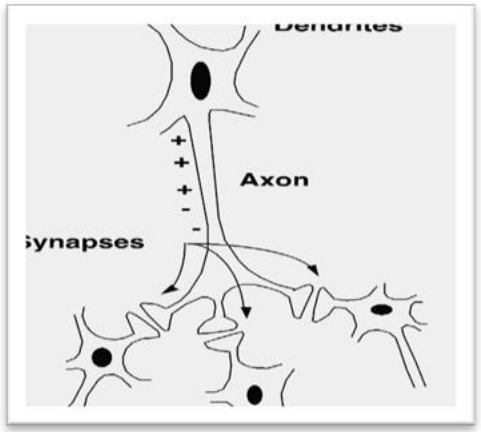
Formally, a computer program is said to **learn** from experience $\mathcal{E}$ with respect to some task $\mathcal{T}$ and performance measure $\mathcal{P}$ if its **performance at $\mathcal{T}$ as measured by $\mathcal{P}$ improves with $\mathcal{E}$.**

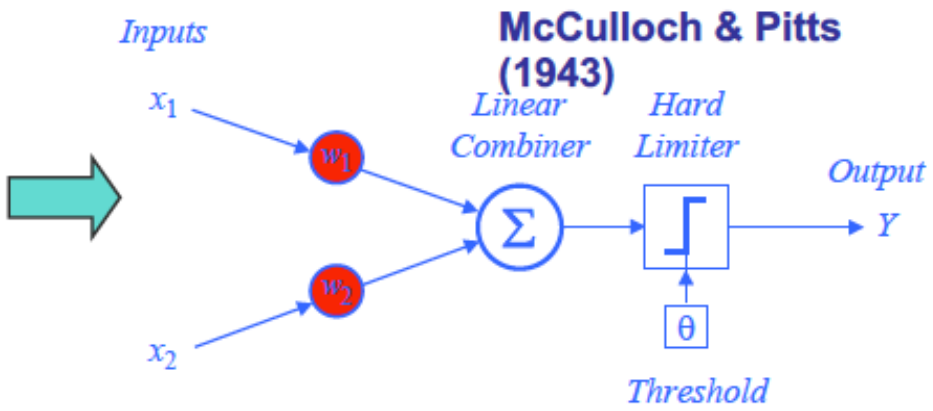
Supervised Learning	› Labeled data › Direct feedback › Predict outcome/future	• Task $\mathcal{T}$: Learn a function $h: \mathcal{X} \rightarrow \mathcal{Y}$ • Experience $\mathcal{E}$: Labeled samples $\{(x_i, y_i)\}_{i=1}^{n}$ • Performance $\mathcal{P}$: A measure of how good h is
Unsupervised Learning	› No labels/targets › No feedback › Find hidden structure in data	• Task $\mathcal{T}$: Discover structure in data • Experience $\mathcal{E}$: Unlabeled samples $\{x_i\}_{i=1}^{n}$ • Performance $\mathcal{P}$: Measure of fit or utility
Reinforcement Learning	› Decision process › Reward system › Learn series of actions	• Task $\mathcal{T}$: Learn a policy $\pi: S \rightarrow A$ • Experience $\mathcal{E}$: Interaction with environment • Performance $\mathcal{P}$: Expected reward

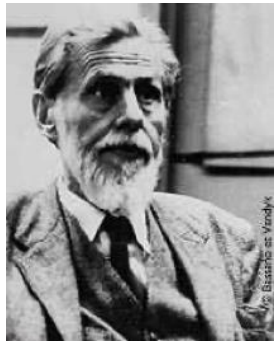
Source: Raschka and Mirjalily (2019). *Python Machine Learning, 3rd Edition*

The building blocks of Deep Learning

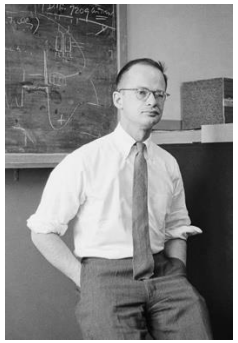
McCulloch & Pitt's neuron model (1943)







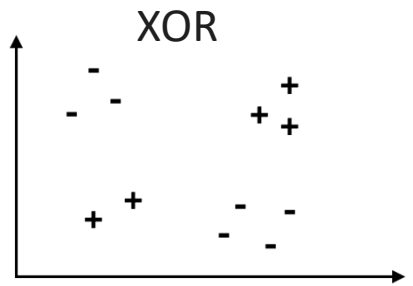
Warren McCulloch

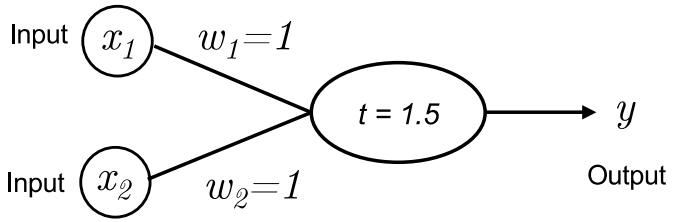


Walter Pitts

From biological neuron to artificial neuron

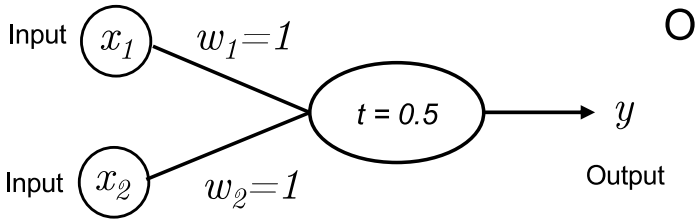
- McCulloch & Pitts neuron: Threshold and (+1, -1) weights
- Can represent “AND”, “OR”, “NOT”
- But not “XOR”





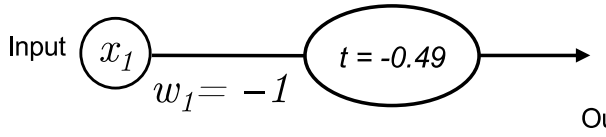
AND

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



OR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



NOT

x_1	y
0	1
1	0

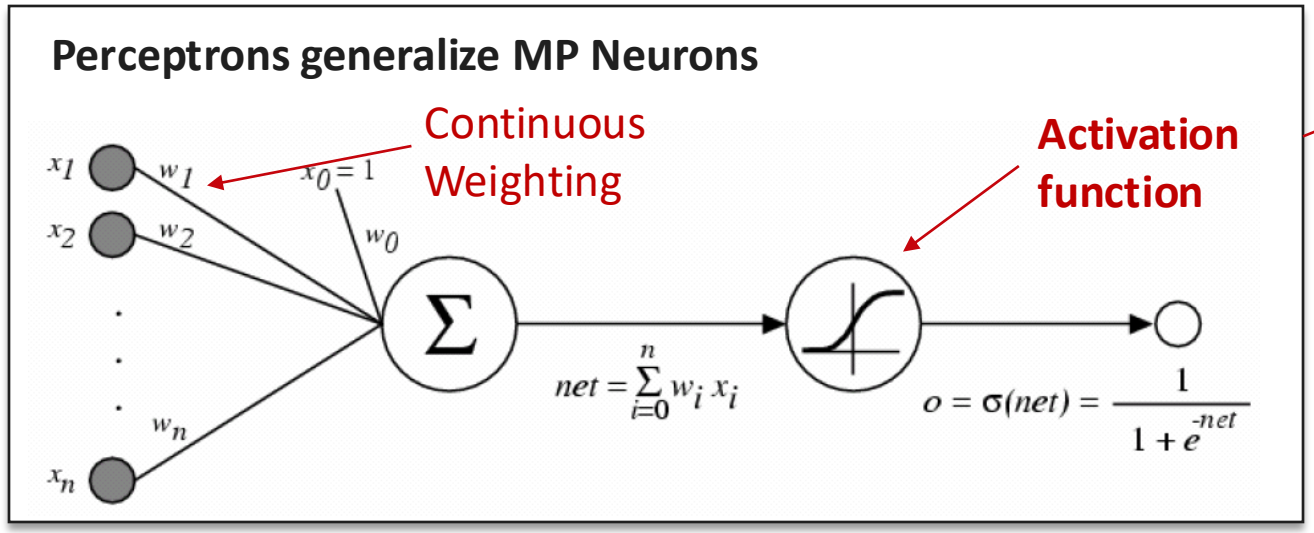
Rosenblatt's Perceptron

A learning rule for the computational/mathematical neuron model

Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton. Project Para*. Cornell Aeronautical Laboratory.

Source: http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/Members/wilex4/Rosen-2.jpg

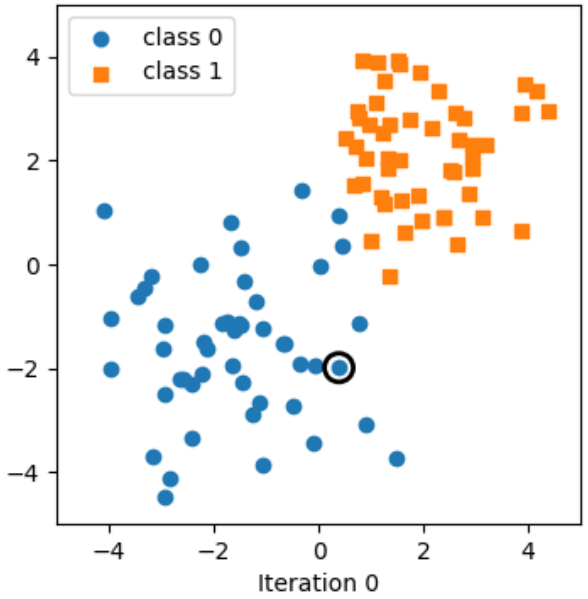
Rosenblatt's Perceptron



threshold function → Classic Rosenblatt Perceptron

Perceptron Learning Algorithm

- Assume binary classification task
- Perceptron finds decision boundary is classes are separable



[animated GIF]

Code at https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L03-perceptron/code/perceptron-animation.ipynb

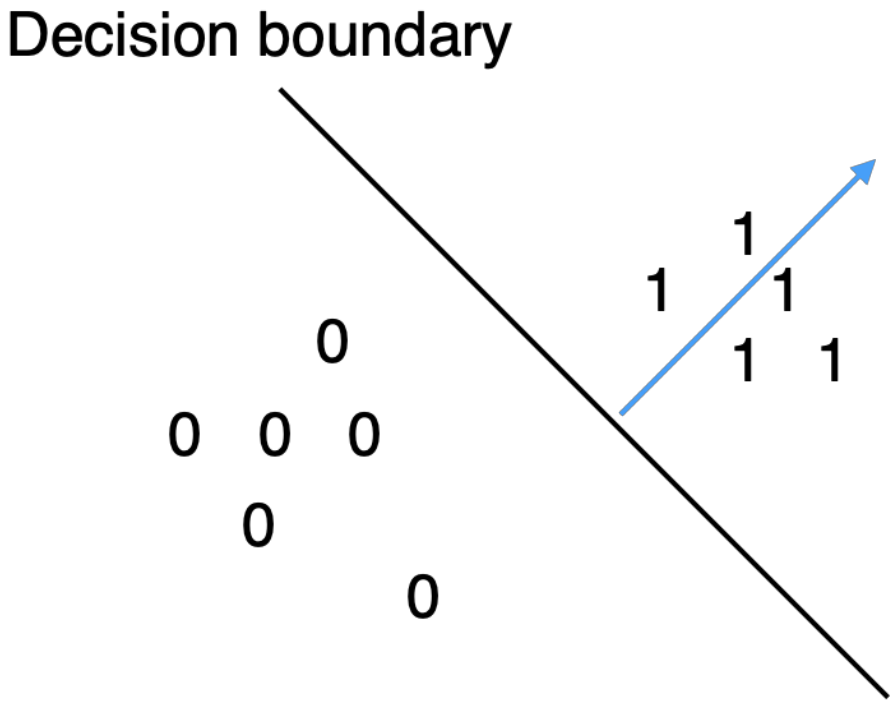
Perceptron Learning Algorithm (pseudocode)

Let

$$\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, \ldots, \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$$

1. Initialize $\boldsymbol{w} \coloneqq 0^m$ (assume weight incl. bias)
2. For every training epoch:
    1. For every $\langle \boldsymbol{x}^{[i]}, y^{[i]} \rangle \in D$:
        1. $\hat{y}^{[i]} \coloneqq \sigma(\boldsymbol{x}^{[i]T} \boldsymbol{w})$ ← Only -0 or 1
        2. $err \coloneqq (y^{[i]} - \hat{y}^{[i]})$ ← Only -1, 0, or 1
        3. $\boldsymbol{w} \coloneqq \boldsymbol{w} + err \times \boldsymbol{x}^{[i]}$

Perceptron Geometric Intuition



Weight vector is perpendicular to the boundary. Why?

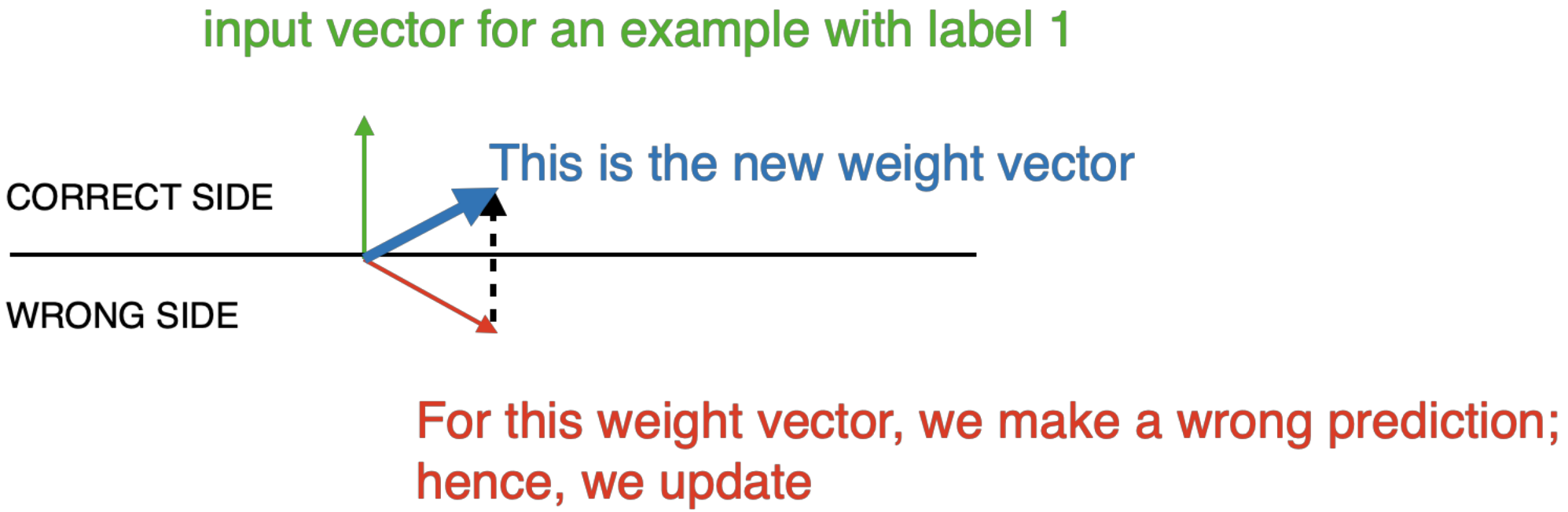
Remember,

$$\hat{y} = \begin{cases} 0, & \mathbf{w}^T\mathbf{x} \leq 0 \\ 1, & \mathbf{w}^T\mathbf{x} > 0 \end{cases}$$

$$\mathbf{w}^T\mathbf{x} = ||\mathbf{w}|| \cdot ||\mathbf{x}|| \cdot \underbrace{\cos(\theta)}$$

So this needs to be 0 at the boundary, and it is zero at 90°

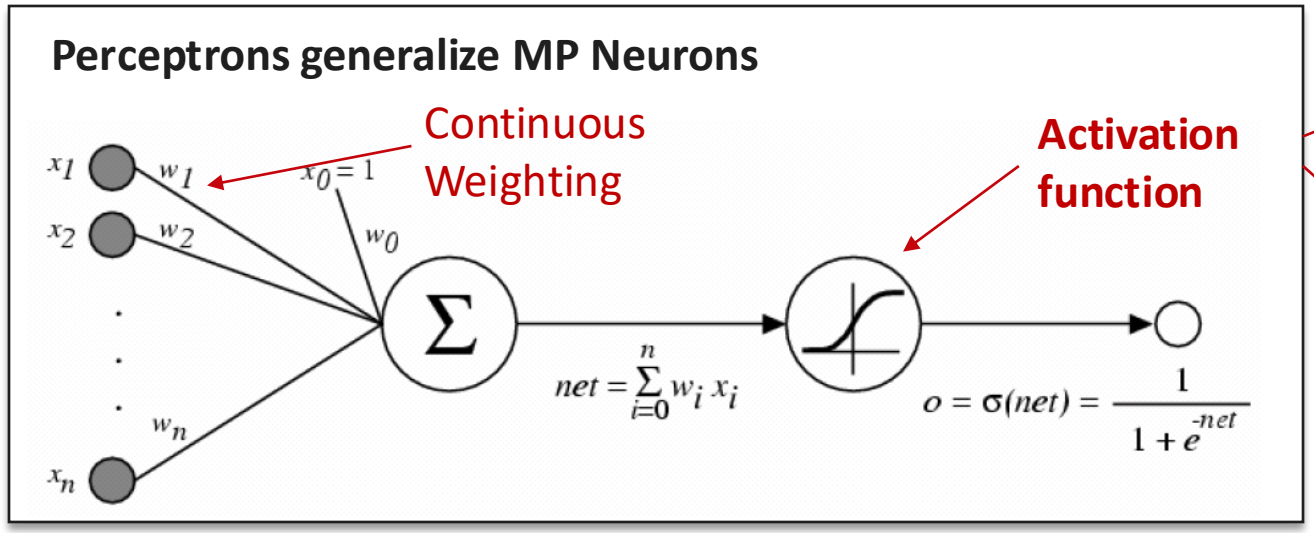
Perceptron Geometric Intuition: Learning



Perceptron Limitations

- Rosenblatt's Perceptron has many problems
  - Linear classifier, no non-linear boundaries
  - Binary classifier, cannot solve XOR problems
  - Does not converge if classes are not linearly separable
  - Many "optimal" solutions in terms of 0/1 loss on the training data
    - Most will not be optimal in terms of generalization performance

Beyond Rosenblatt's Perceptron



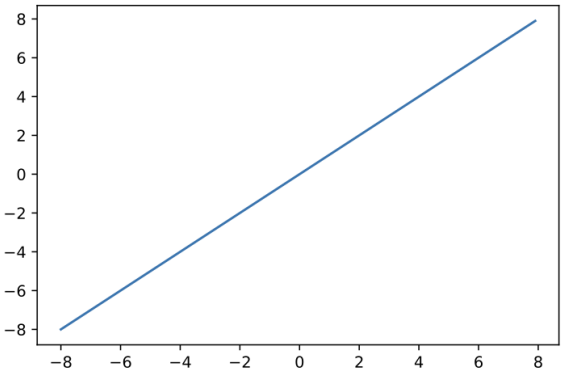
threshold function → Classic Rosenblatt Perceptron

sigmoid → DL "Perceptron" / sigmoid unit

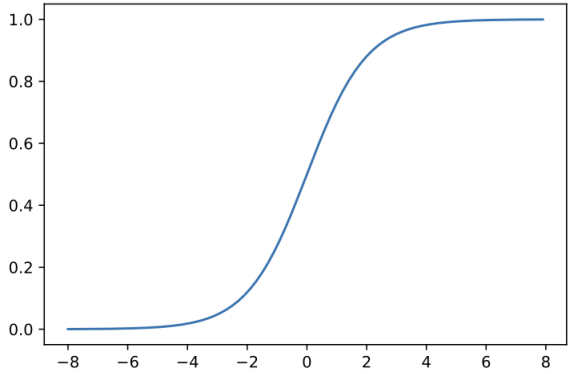
- Many activation functions:
  - Threshold function (perceptron, 1950+)
  - Sigmoid function (before 2000)
  - ReLU function (popular since CNNs)
  - Many variants of ReLU, e.g. leaky ReLU, GeLU

A Selection of Common Activation Functions

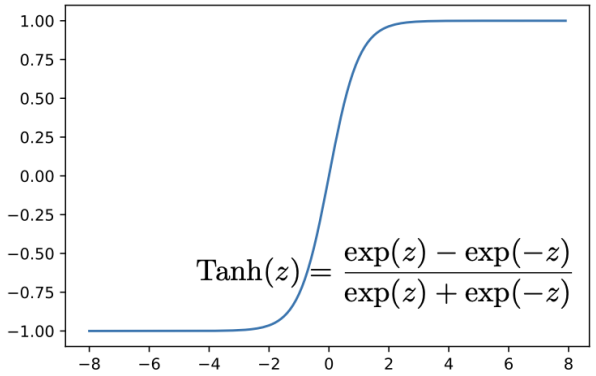
Identity



(Logistic) Sigmoid

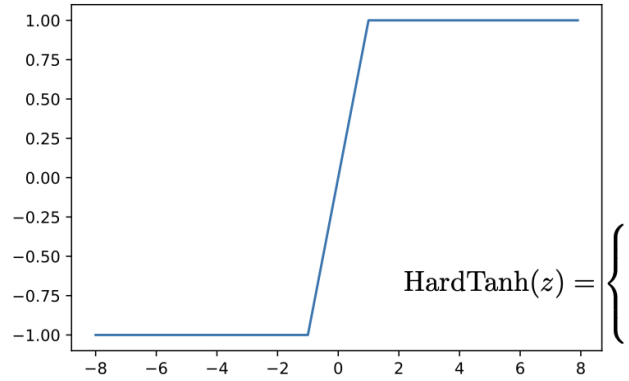


Tanh ("tanH")



$$\text{Tanh}(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

Hard Tanh

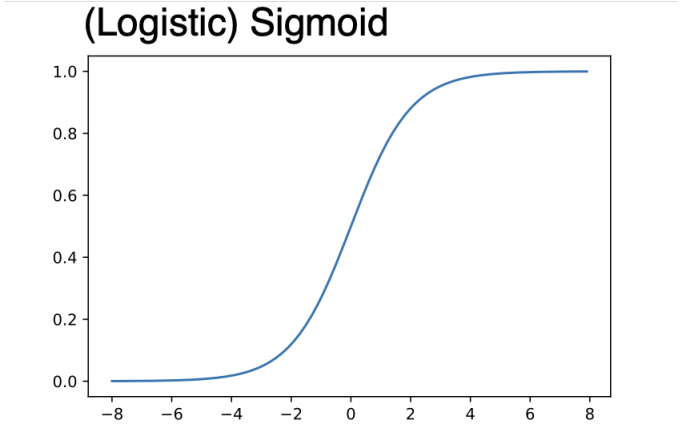


$$\text{HardTanh}(z) = \begin{cases} 1 & \text{if } z > 1 \\ -1 & \text{if } z < -1 \\ z & \text{otherwise} \end{cases}$$

A Selection of Common Activation Functions

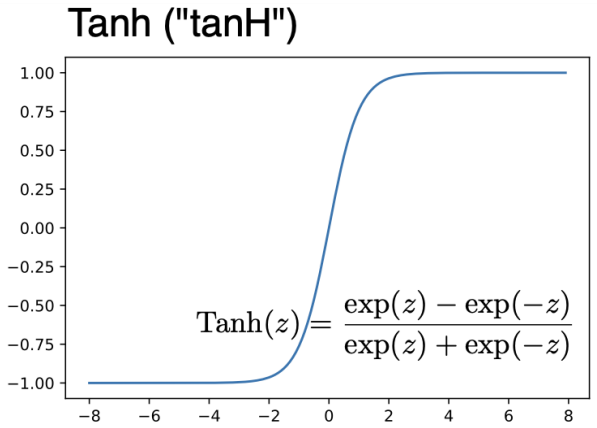
Advantages of Tanh

- Mean centering
- Positive and negative values
- Larger gradients



Also simple derivative:

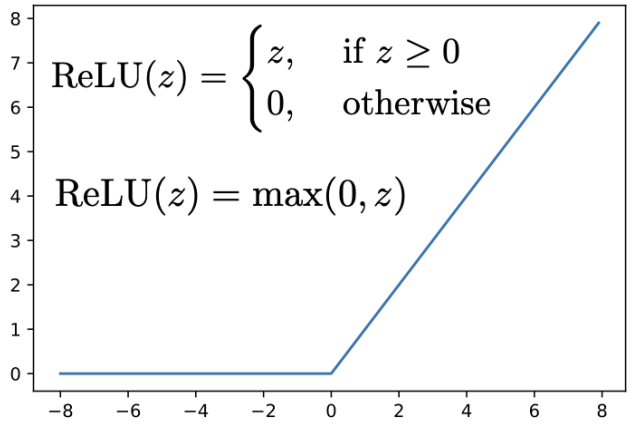
$$\frac{d}{dz}Tanh(z) = 1 - Tanh(z)^2$$



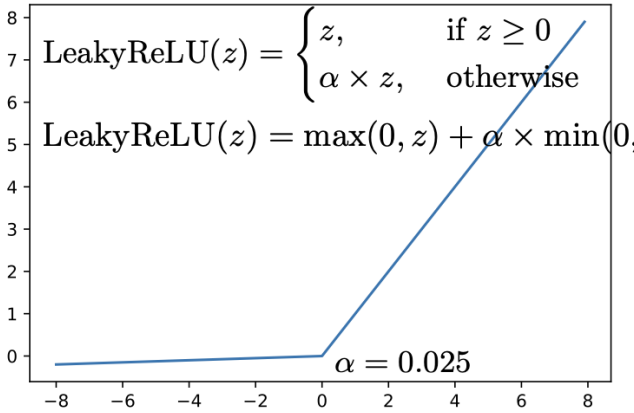
Important to normalize inputs to mean zero and use random weight initialization with **avg. weight centered at zero**

A Selection of Common Activation Functions (cont.)

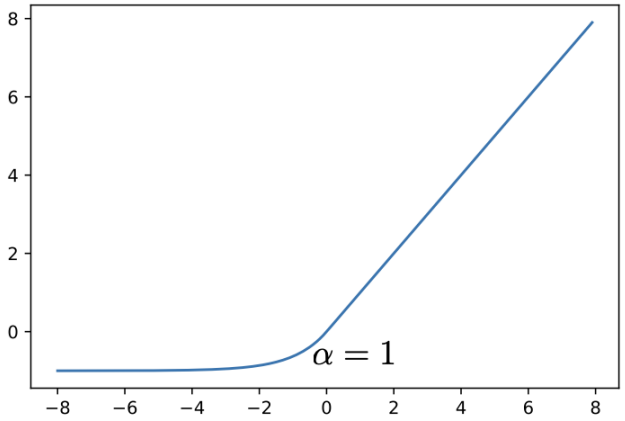
ReLU (Rectified Linear Unit)



Leaky ReLU



ELU (Exponential Linear Unit)



PReLU (Parameterized Rectified Linear Unit)

here, alpha is a trainable parameter

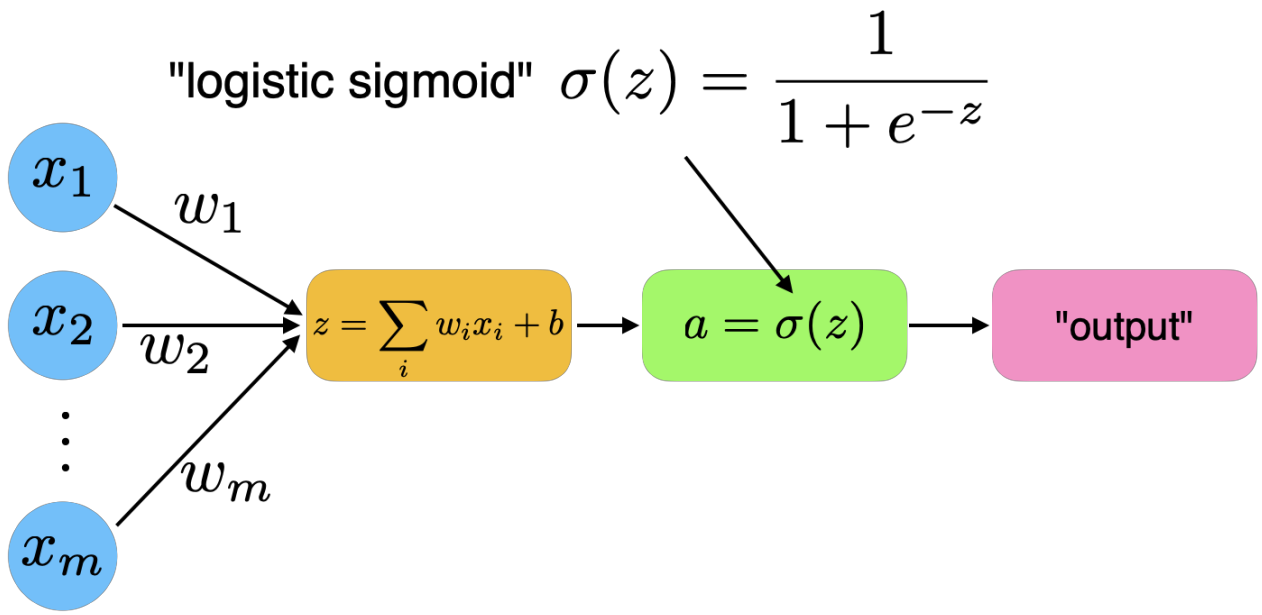
$$\mathrm{PReLU}(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha z, & \text{otherwise} \end{cases}$$

$$\mathrm{PReLU}(z) = \max(0, z) + \alpha \times \min(0, z)$$

Logistic Regression: A Bridge from Perceptron to Probabilistic Model

Logistic Regression Neuron

- For binary classes $y \in \{0, 1\}$



Logistic Regression

- Given the output:

$$h(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

- We compute the probability as

$$P(y|\mathbf{x}) = \begin{cases} h(\mathbf{x}) & \text{if } y = 1 \\ 1 - h(\mathbf{x}) & \text{if } y = 0 \end{cases}$$

$$P(y|\mathbf{x}) = a^y (1-a)^{(1-y)}$$

Recall Bernoulli distribution…

Logistic Regression: Estimation

- Given the probability:

$$P(y|\mathbf{x}) = a^y(1-a)^{(1-y)}$$

- Under MLE estimation, we would like to maximize the multi-sample likelihood:

$$P\left(y^{[i]},\ldots,y^{[n]}|\mathbf{x}^{[1]},\ldots,\mathbf{x}^{[n]}\right) = \prod_{i=1}^{n} P\left(y^{[i]}|\mathbf{x}^{[i]}\right)$$

$$= \underbrace{\prod_{i=1}^{n}\left(\sigma\left(z^{(i)}\right)\right)^{y^{(i)}}\left(1-\sigma\left(z^{(i)}\right)\right)^{1-y^{(i)}}}_{\text{Likelihood}}$$

Logistic Regression: Estimation

$$P\left(y^{[i]}, \ldots, y^{[n]} | \mathbf{x}^{[1]}, \ldots, \mathbf{x}^{[n]}\right) = \underbrace{\prod_{i=1}^{n} \left(\sigma\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \sigma\left(z^{(i)}\right)\right)^{1-y^{(i)}}}_{\text{Likelihood}}$$

- We are going to optimize via gradient descent, so let's apply the logarithm to separate components:

$$l(\mathbf{w}) = \log L(\mathbf{w})$$

$$= \underbrace{\sum_{i=1}^{n} \left[y^{(i)} \log\left(\sigma\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(z^{(i)}\right)\right)\right]}_{\text{Log-Likelihood}}$$

Logistic Regression: Gradient Descent Learning Rule

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial \mathcal{L}}{\partial a}\frac{da}{dz}\frac{\partial z}{\partial w_j}$$

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{a-y}{a-a^2}$$

$$\frac{da}{dz} = \frac{e^{-z}}{(1+e^{-z})^2} = a \cdot (1-a)$$

$$\frac{\partial z}{w_j} = x_j$$

$$\frac{\partial \mathcal{L}}{\partial z} = a - y \longrightarrow \frac{\partial \mathcal{L}}{\partial w_j} = (a-y)x_j$$

Logistic Regression: Learning Rule

Stochastic gradient descent:

1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$, $\mathbf{b} := 0$
2. For every training epoch:
   A. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$
      (a) $\hat{y}^{[i]} := \sigma\left(\mathbf{x}^{[i]T}\mathbf{w} + b\right)$
      (b) $\nabla_{\mathbf{w}}\mathcal{L} = -\left(y^{[i]} - \hat{y}^{[i]}\right)\mathbf{x}^{[i]}$
      $\nabla_b\mathcal{L} = -\left(y^{[i]} - \hat{y}^{[i]}\right)$
      (c) $\mathbf{w} := \mathbf{w} + \eta \times \left(-\nabla_{\mathbf{w}}\mathcal{L}\right)$
      $b := b + \eta \times \underbrace{\left(-\nabla_b\mathcal{L}\right)}$

learning rate

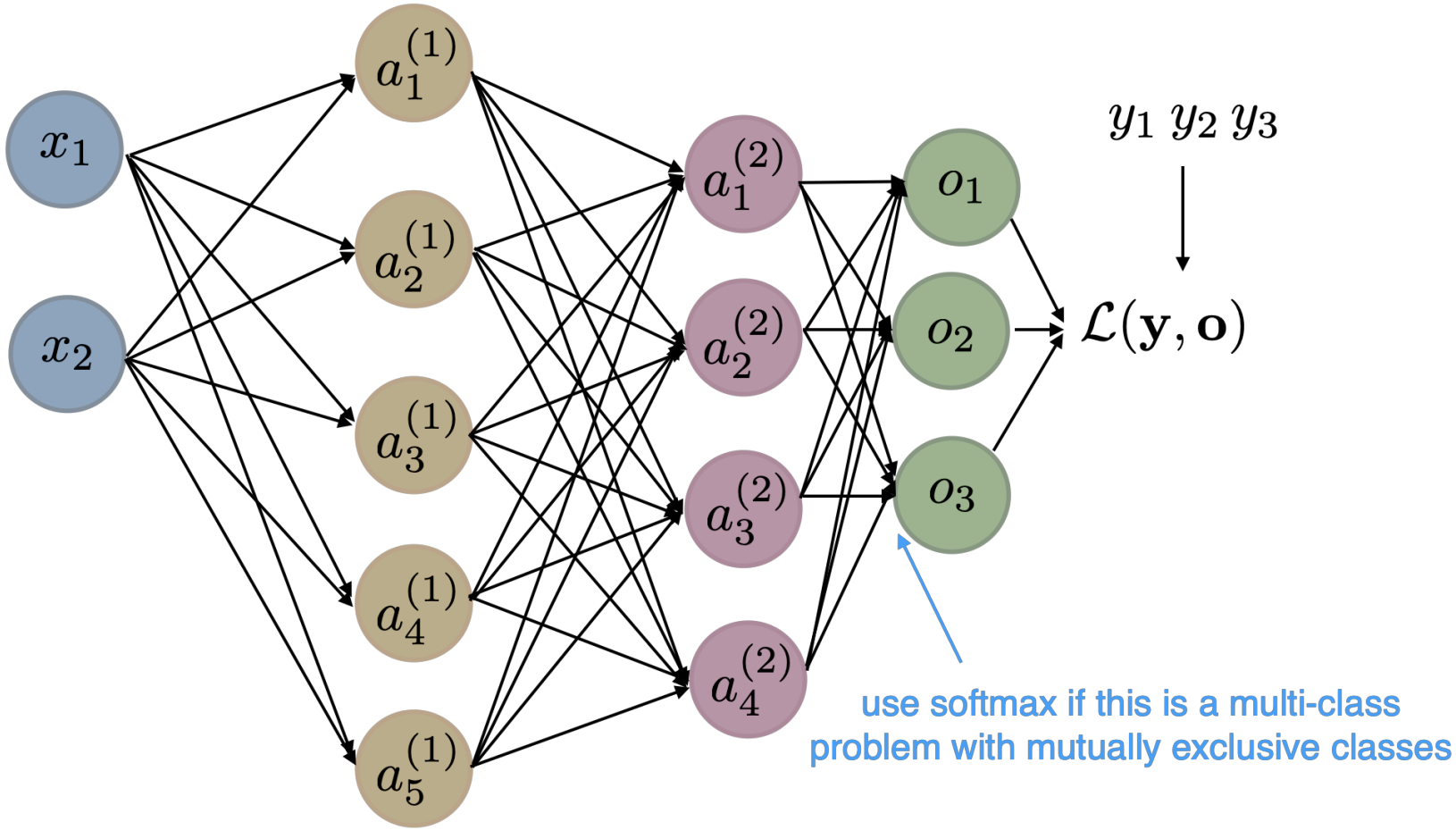
negative gradient

Note

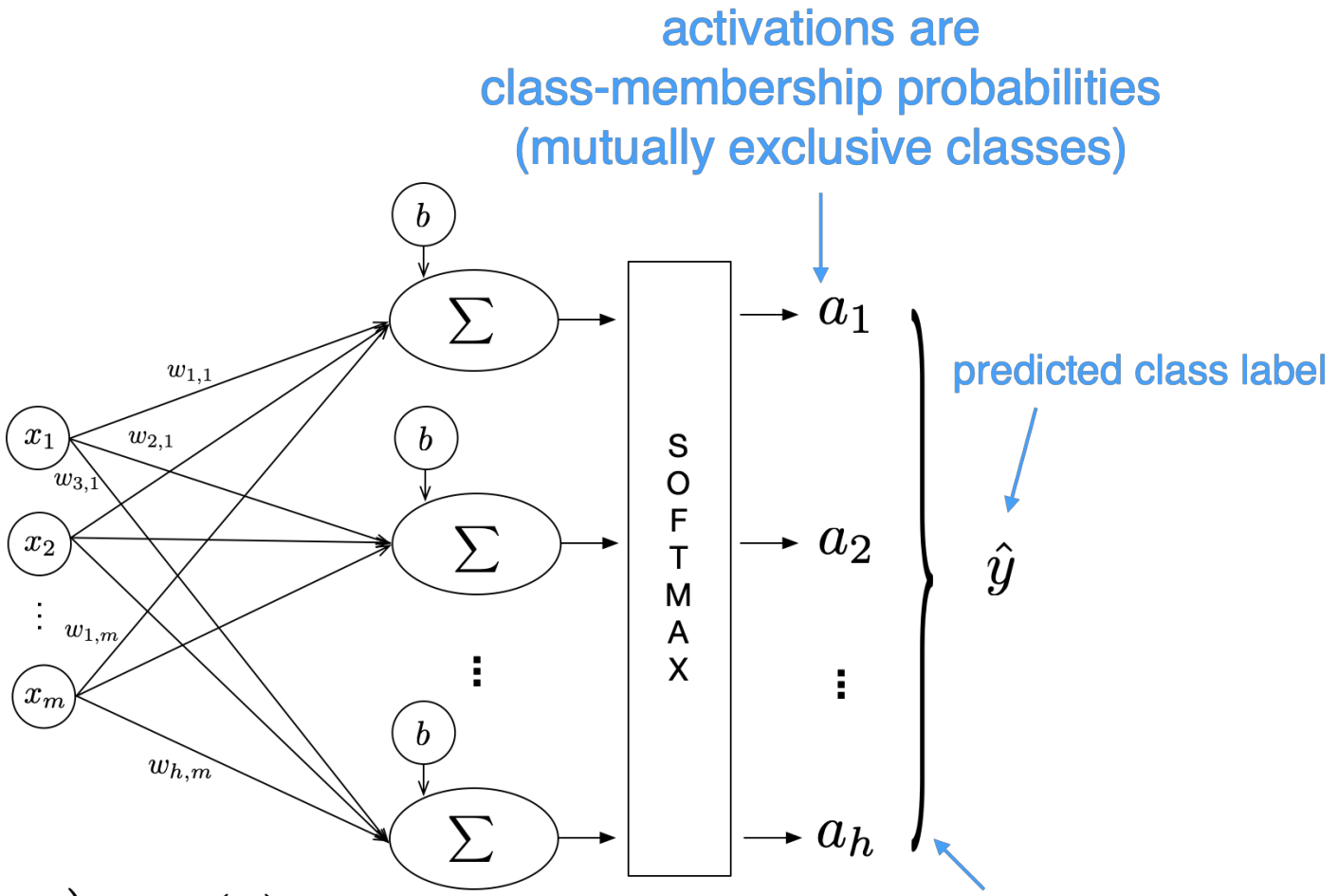
$$a - y \Leftrightarrow -\left(y^{[i]} - \hat{y}^{[i]}\right)$$

Multilayer Perceptron

- Computation Graph with Multiple Fully-Connected Layers



Multinomial ("Softmax") Logistic Regression



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

“Softmax”

$$P(y = t \mid z_t^{[i]}) = \sigma_{\text{softmax}}(z_t^{[i]}) = \frac{e^{z_t^{[i]}}}{\sum_{j=1}^{h} e^{z_j^{[i]}}}$$

$$t \in \{j...h\}$$

h is the number of class labels

A “soft” (differentiable) version of ”max”

Requires one-hot encoding

class labels
0
1
3
2

class_0	class_1	class_2	class_3
1	0	0	0
0	1	0	0
0	0	0	1
0	0	1	0

Loss Function (assuming one-hot encoding)

(Multi-category) Cross Entropy

for *h* different class labels

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{j=1}^{h} -y_j^{[i]} \log\left(a_j^{[i]}\right)$$

Loss Function (assuming one-hot encoding)

$$\mathcal{L}_{\text{binary}} = -\sum_{i=1}^{n} \left(y^{[i]} \log(a^{[i]}) + (1 - y^{[i]}) \log(1 - a^{[i]}) \right)$$

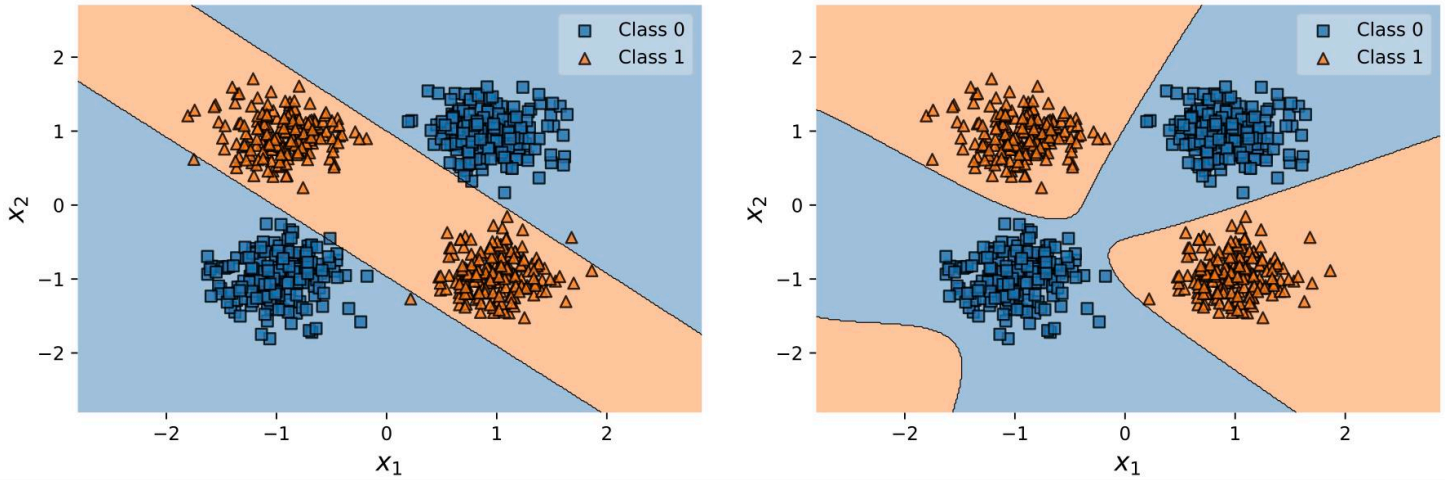
This assumes one-hot encoded labels!

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{j=1}^{h} -y_j^{[i]} \log \left(a_j^{[i]} \right)$$

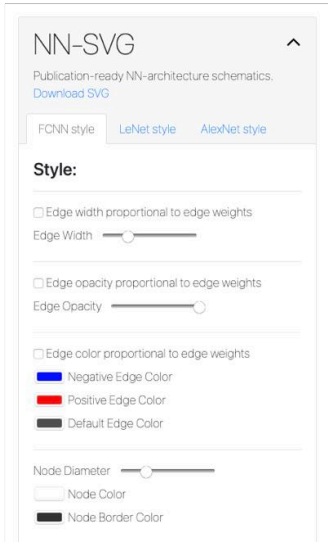
for *h* different class labels

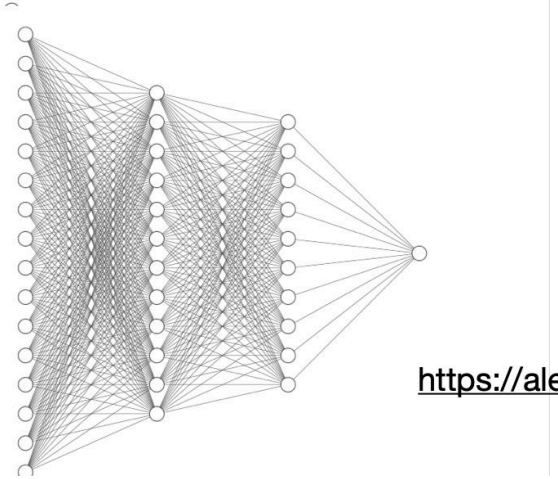
(Multi-category) Cross Entropy

Multilayer Perceptrons Can Solve XOR



Decision boundaries of two different multilayer perceptrons on simulated data solving the XOR problem





https://alexlenail.me/NN-SVG/index.html

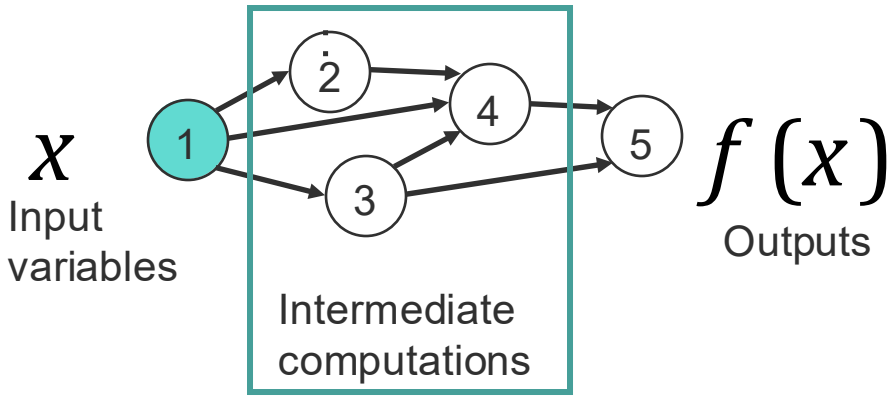
A new problem: Training

- How can we train a multilayer model?
  - No targets / ground truth for the hidden nodes
- Solution: Backpropagation

An algorithm to train models with hidden variables

Backpropagation

- Neural networks are function compositions that can be represented as computation graphs:



$$\frac{\partial f_n}{\partial x} =$$

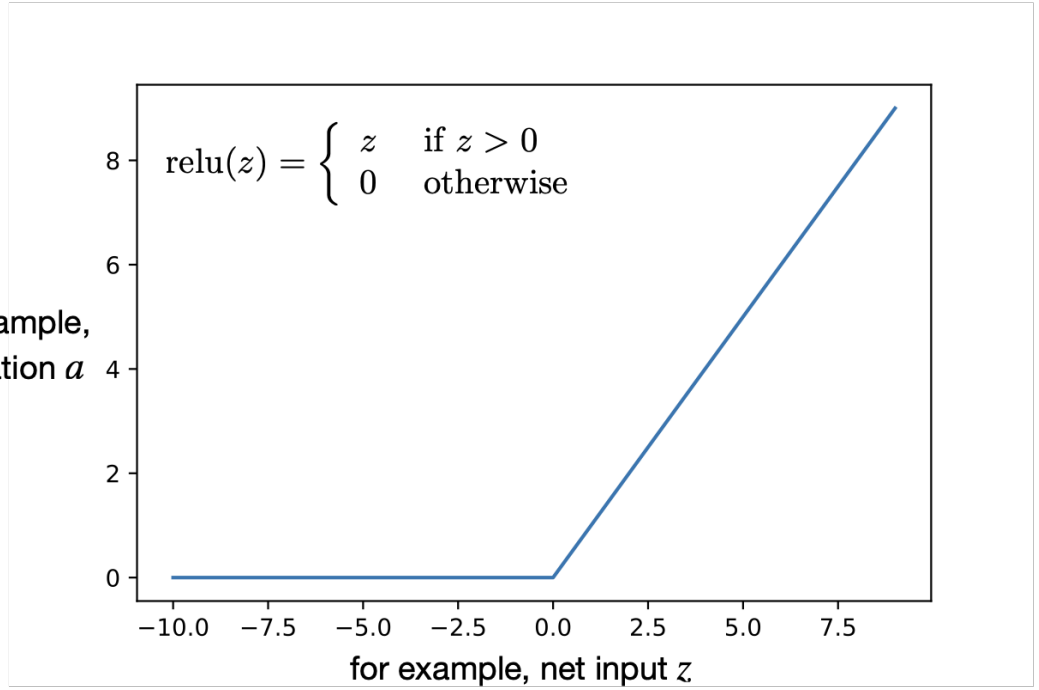
- By applying the chain rule, and working in reverse order, we get:

$$\frac{\partial f_n}{\partial x} = \sum_{i_1 \in \pi(n)} \frac{\partial f_n}{\partial f_{i_1}} \frac{\partial f_{i_1}}{\partial x} = \sum_{i_1 \in \pi(n)} \frac{\partial f_n}{\partial f_{i_1}} \sum_{i_2 \in \pi(i_1)} \frac{\partial f_{i_1}}{\partial f_{i_2}} \frac{\partial f_{i_1}}{\partial x} = \dots$$

Computation graphs: ReLU

Suppose we have the following activation function:

$$a(x, w, b) = relu(w \cdot x + b)$$

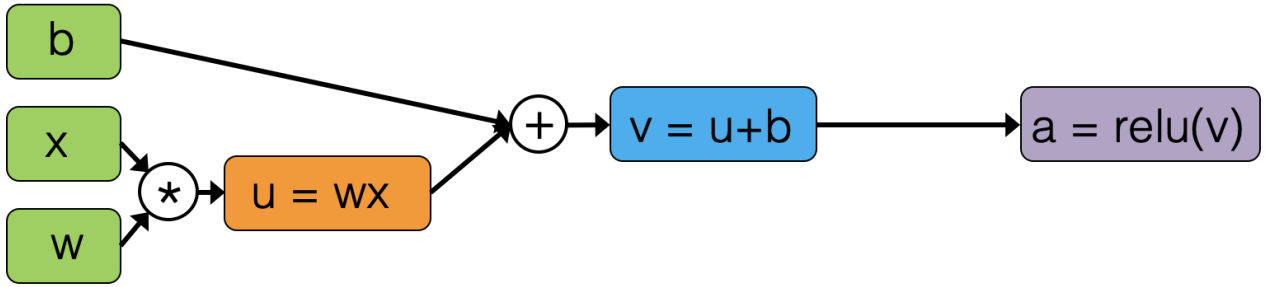


ReLU = Rectified Linear Unit

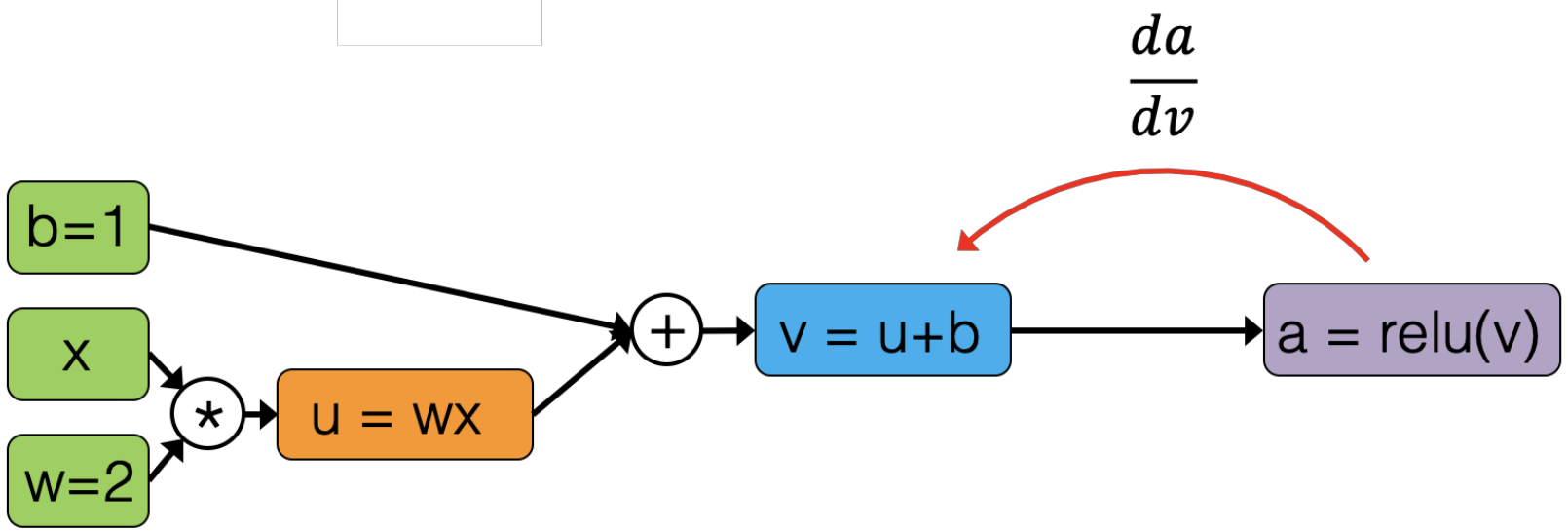
(prob. the most commonly used activation function in DL)

Computation graphs: ReLU

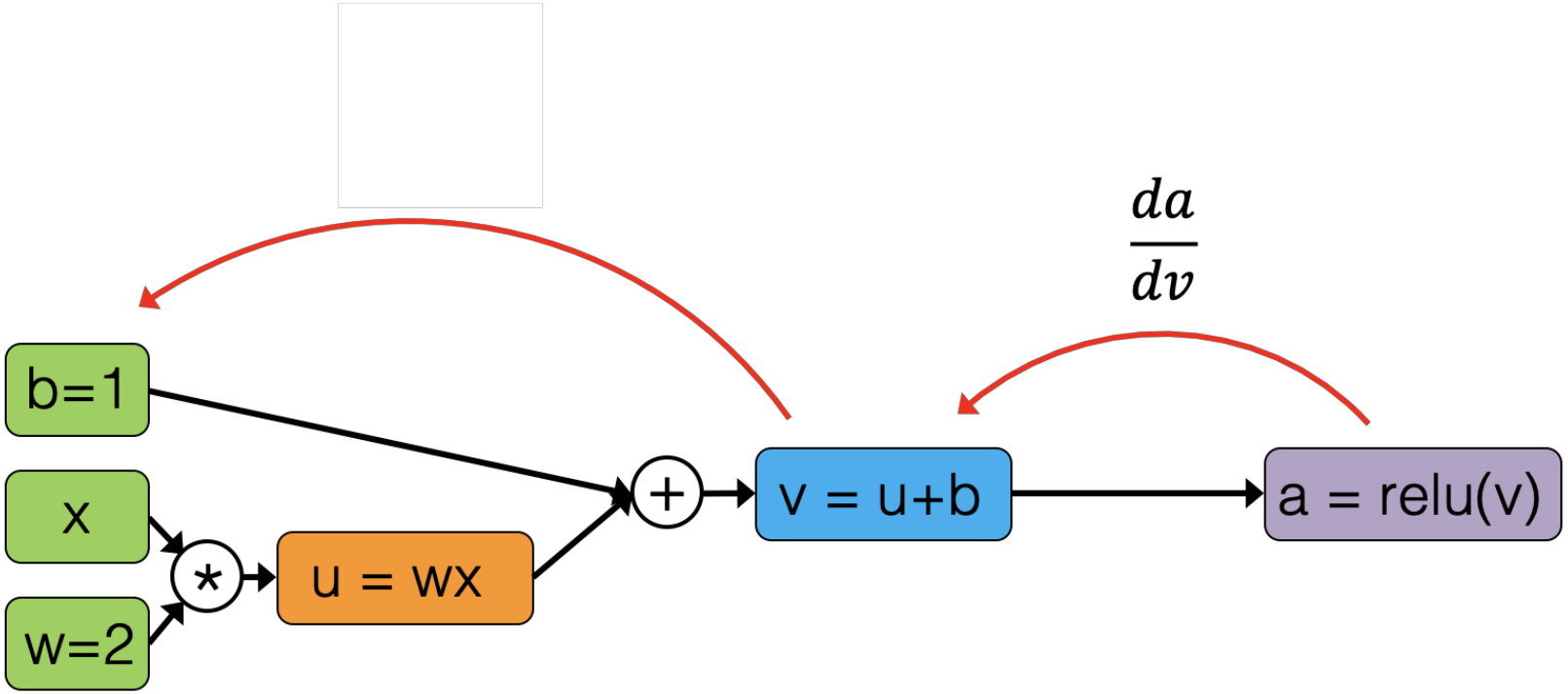
$$a(x, w, b) = relu(\underbrace{\underbrace{w \cdot x}_{u} + b}_{v})$$



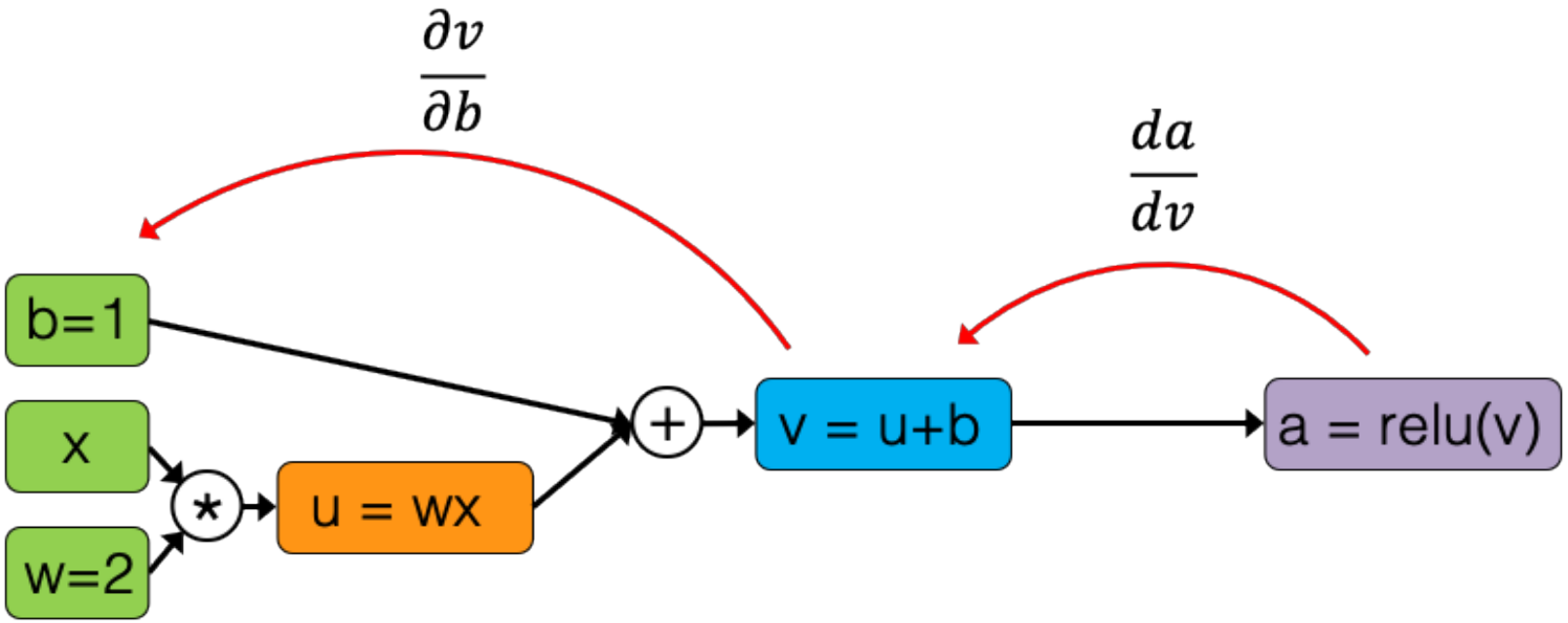
Computation graphs: ReLU



Computation graphs: ReLU

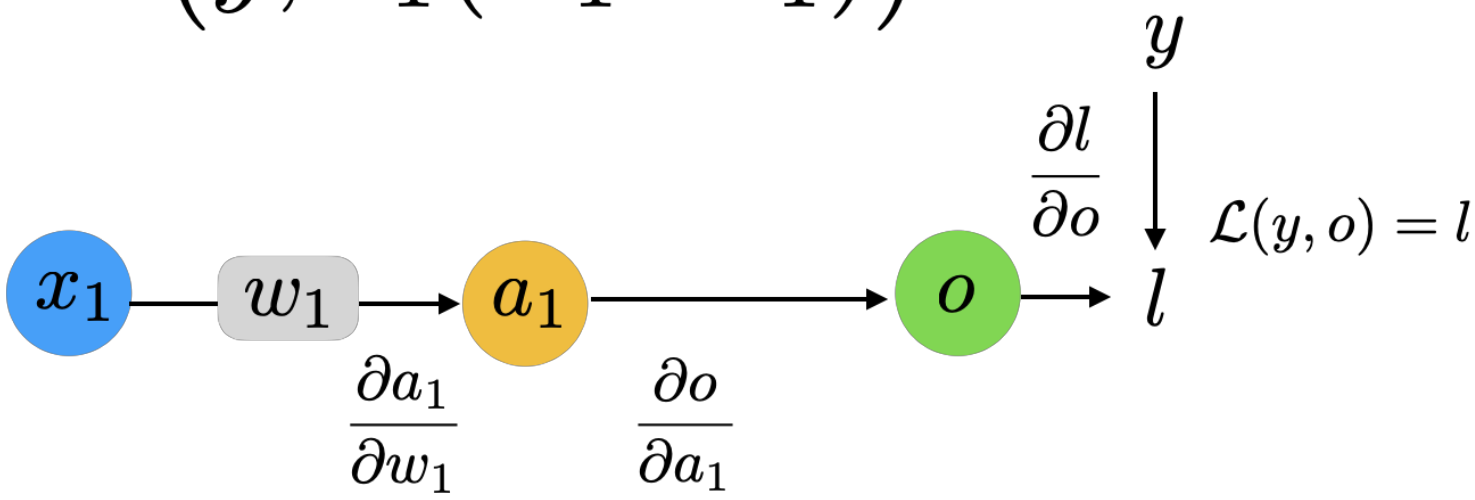


Computation graphs: ReLU



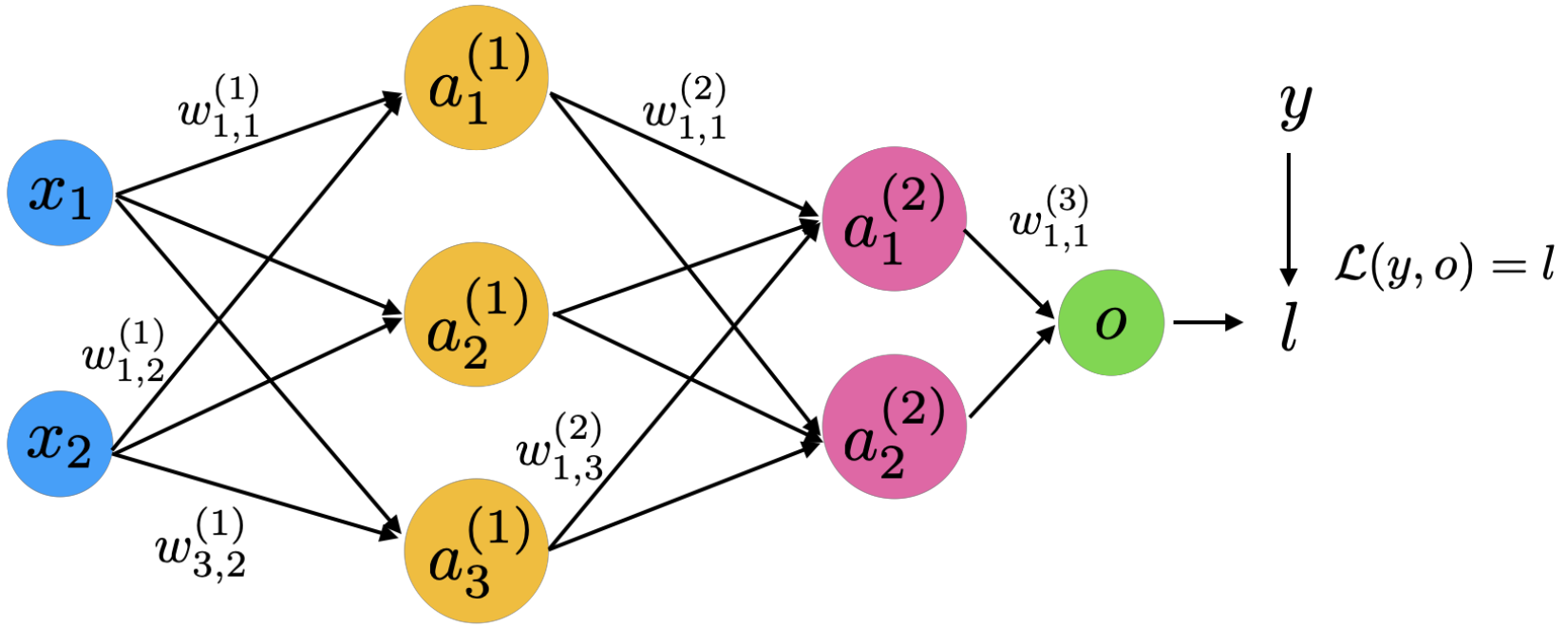
Computation graphs: Single-path

$$\mathcal{L}\big(y, \sigma_1(w_1 \cdot x_1)\big)$$



$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}$$ (univariate chain rule)

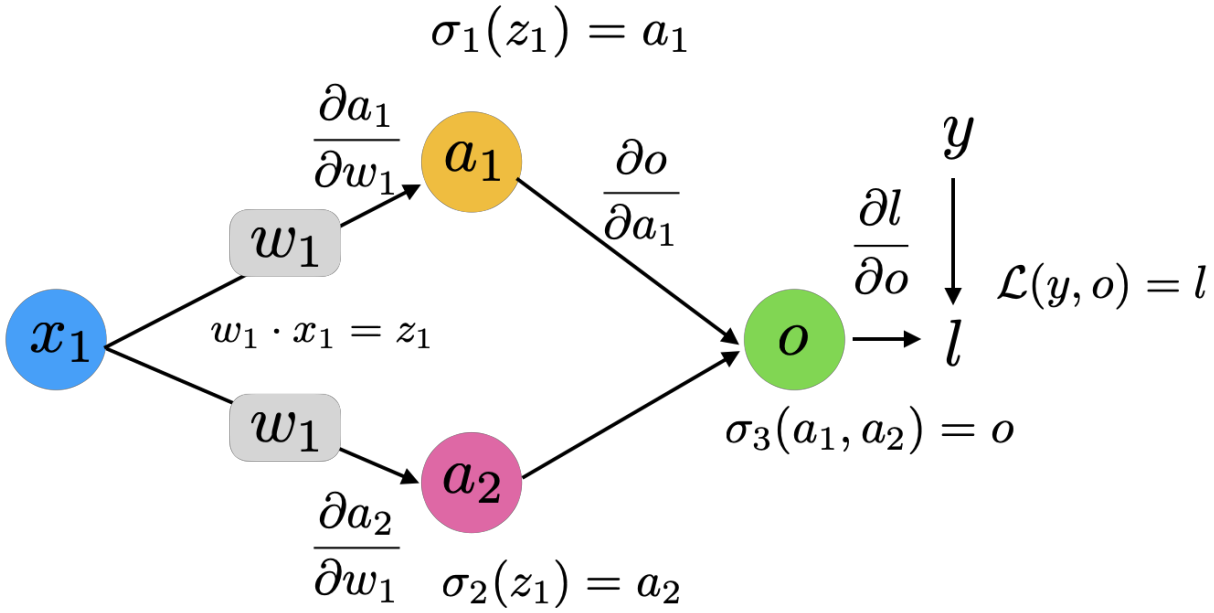
Computation graphs: Fully-Connected Layer



$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$
$$+ \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

Computation graphs: Weight-Sharing

$$\mathcal{L}\big(y, \sigma_3\big[\sigma_1(w_1 \cdot x_1), \sigma_2(w_1 \cdot x_1)\big]\big)$$



Upper path

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_1}$$ (multivariable chain rule)

Lower path

PyTorch: Automated Differentiation

PyTorch Usage: Step 1 (Definition)

```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_feat, num_h1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_h1, num_h2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_h2, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
```

Backward will be inferred automatically if we use the nn.Module class!

Define model parameters that will be instantiated when created an object of this class

Define how and it what order the model parameters should be used in the forward pass

PyTorch Usage: Step 2 (Creation)

```
torch.manual_seed(random_seed)
model = MultilayerPerceptron(num_features=num_features,
                             num_classes=num_classes)

model = model.to(device)

optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
```

Instantiate model
(creates the model parameters)

Define an optimization method

PyTorch Usage: Step 3 (Training)

Run for a specified number of epochs

Iterate over minibatches in epoch

If your model is on the GPU, data should also be on the GPU

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        cost = F.cross_entropy(probas, targets)
        optimizer.zero_grad()

        cost.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy
```

y = model(x) calls .__call__ and then .forward(), where some extra stuff is done in __call__;

don't run y = model.forward(x) directly

Gradients at each leaf node are accumulated under the .grad attribute, not just stored. This is why we have to zero them before each backward pass

PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy
```

This will run the forward() method

Define a loss function to optimize

Set the gradient to zero
(could be non-zero from a previous forward pass)

Compute the gradients, the backward is automatically constructed by "autograd" based on the forward() method and the loss function

Use the gradients to update the weights according to the optimization method (defined on the previous slide)
E.g., for SGD, $w := w +$ learning_rate $\times$ gradient

PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy
```

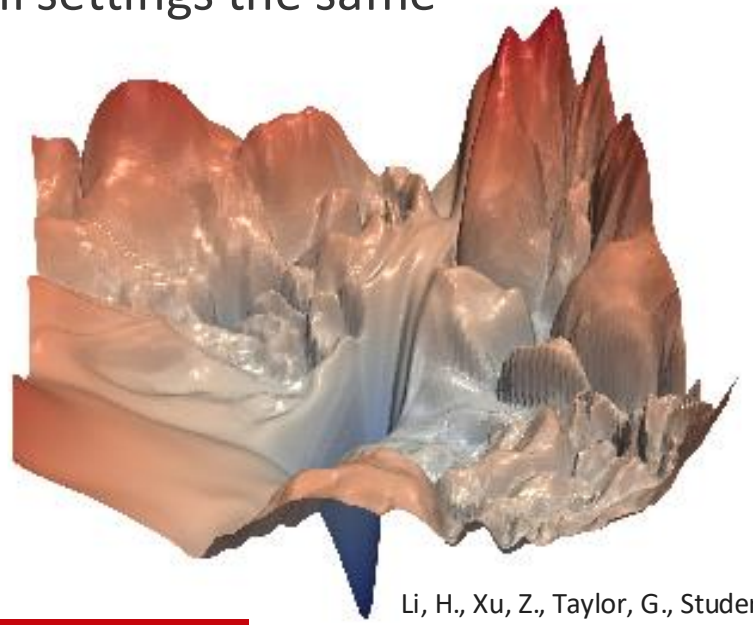
For evaluation, set the model to eval mode (will be relevant later when we use DropOut or BatchNorm)

This prevents the computation graph for backpropagation from automatically being build in the background to save memory

Improvements to optimization

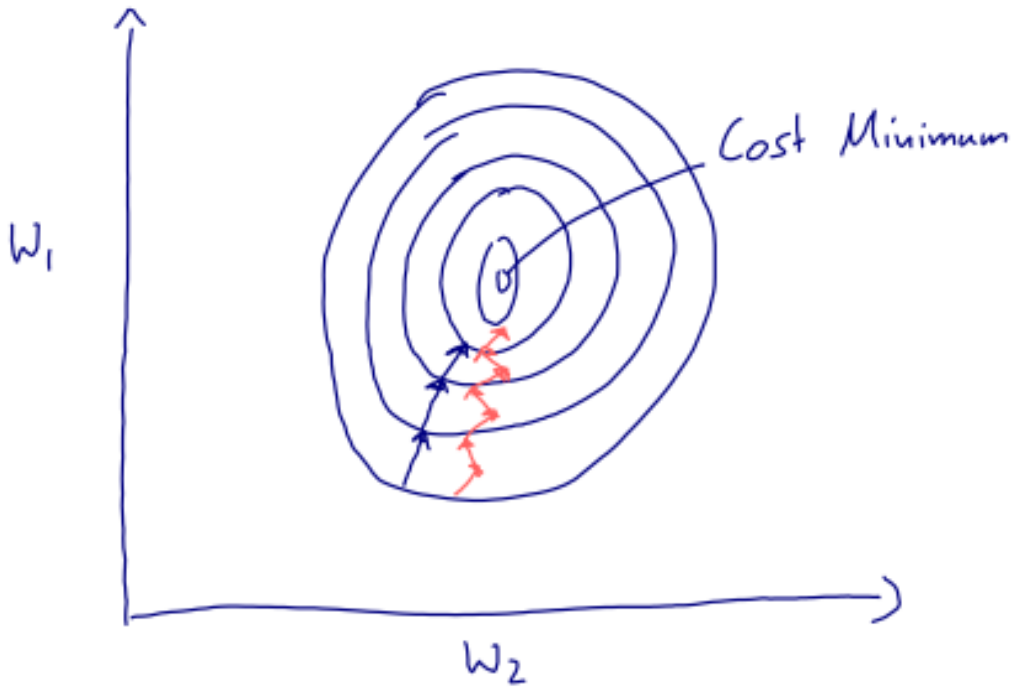
Note that our Loss is Not Convex Anymore

- Linear regression, Adaline, Logistic Regression, and Softmax Regression have convex loss functions
- But our deep loss is no longer convex (most of the time)
  - In practice, we usually end up at different local minima if we repeat the training (e.g. by changing the random seed for weight initialization or shuffling the dataset while leaving all settings the same



Li, H., Xu, Z., Taylor, G., Studer, C. and Goldstein, T., 2018. Visualizing the loss landscape of neural nets. In Advances in Neural Information Processing Systems (pp. 6391-6401).

Minibatch Training Recap



- Minibatch learning is a form of stochastic gradient descent
- Each minibatch can be considered a sample drawn from the training set (where the training set is in turn a sample drawn from the population)
- Hence, the gradient is **noisier**

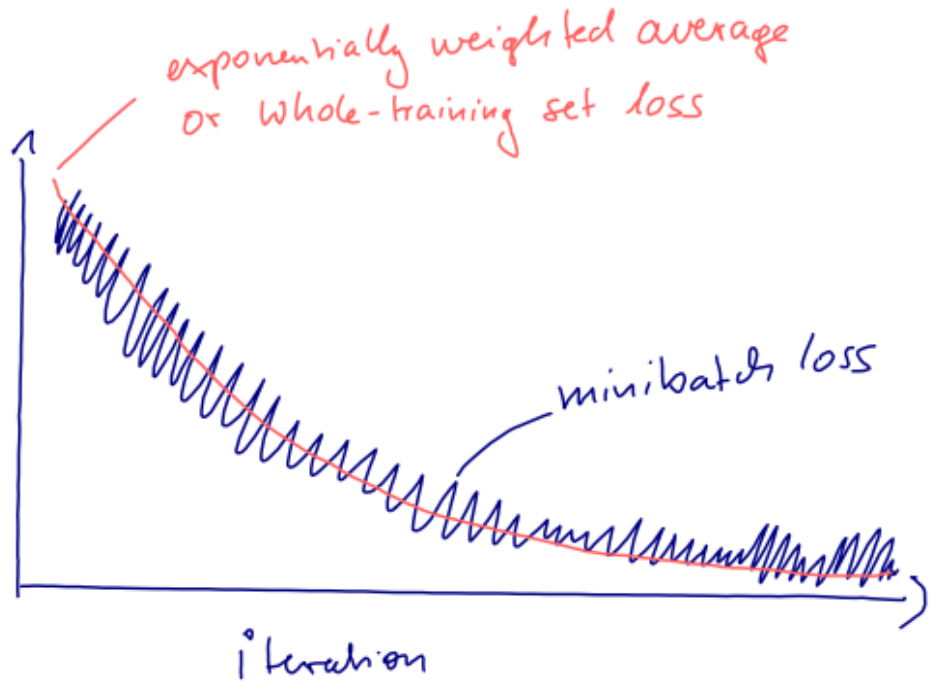
A **noisy** gradient can be:

- **good**: chance to escape local minima
- **bad**: can lead to extensive oscillation

Learning Rate Decay

- Batch effects -- minibatches are samples of the training set, hence minibatch loss and gradients are approximations
- Hence, we usually get oscillations
- To dampen oscillations towards the end of the training, we can **decay the learning rate**

  - Danger of learning rate is to decrease the learning rate too early
  - Practical tip: try to **train the model without learning rate decay first**, then add it later
  - You can also use the validation performance (e.g., accuracy) to judge whether lr decay is useful (as opposed to using the training loss)



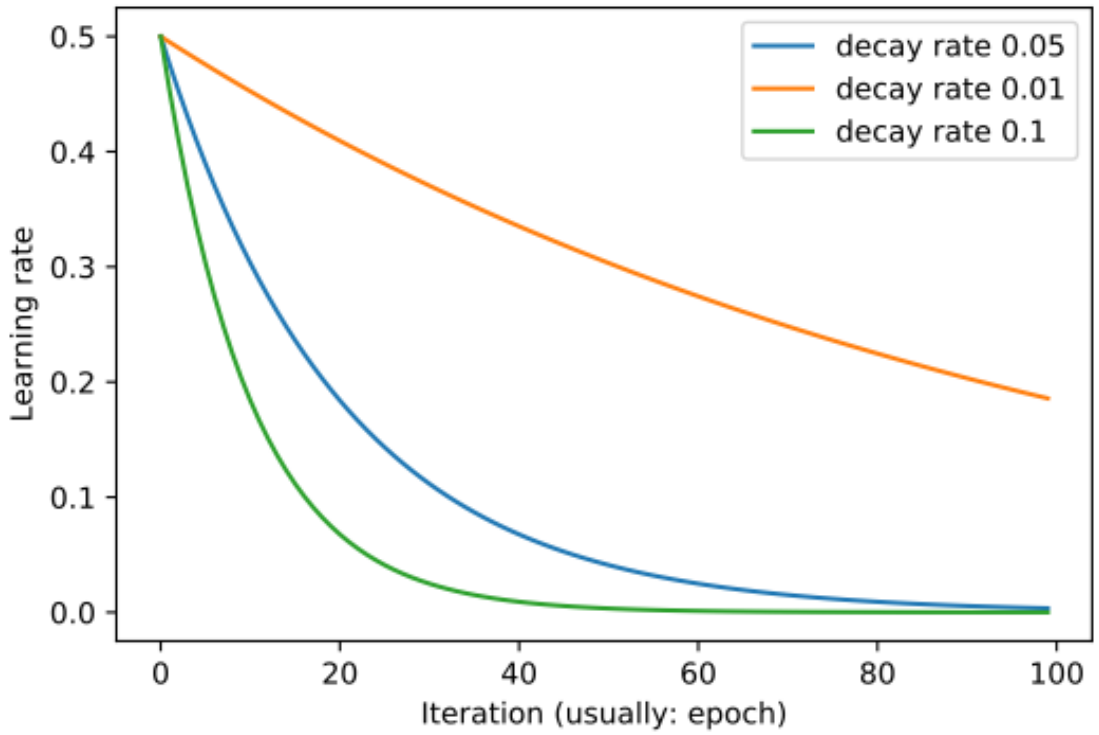
Learning Rate Decay

Most common variants for lr decay:

1. Exponential Decay:

$$\eta_t := \eta_0 \mathrm{e}^{-\mathrm{k}\cdot t}$$

where k is the decay rate



Learning Rate Decay

Most common variants for lr decay:

1. Exponential Decay:

$$\eta_t := \eta_0 \mathrm{e}^{-\mathrm{k} \cdot t}$$

where k is the decay rate

2. Halving the learning rate:

$$\eta_t := \eta_{t-1}/2$$

where t is a multiple of T_0 (e.g. $T_0 = 100$)

3. Inverse decay:

$$\eta_t := \frac{\eta_0}{1 + k \cdot t}$$

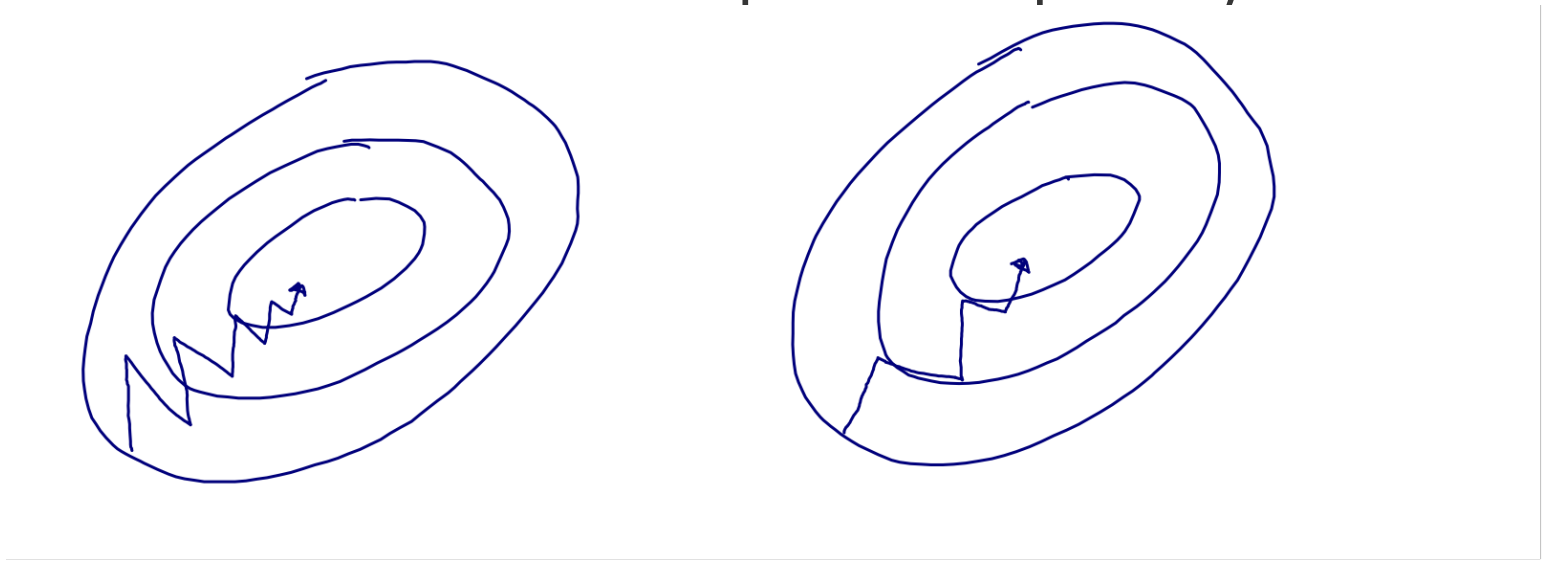
Training with “Momentum”

- Main idea: Let’s dampen oscillations by using “velocity” (the speed of the “movement” from previous updates)

https://www.asherworldturns.com/zorbing-new-zealand/

Training with “Momentum”

- Main idea: Let’s dampen oscillations by using “velocity” (the speed of the “movement” from previous updates)



Key take-away: Not only move in the (opposite) direction of the gradient, but also move in the “**weighted averaged**" direction of the last few updates

Training with “Momentum”

Often referred to as "velocity" ***V***

"velocity" from the previous iteration

$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t-1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

Usually, we choose a momentum rate between 0.9 and 0.999; you can think of it as a "friction" or "dampening" parameter

Regular partial derivative/ gradient multiplied by learning rate at current time step *t*

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural Networks : The Official Journal of the International Neural Network Society, 12(1), 145–151. http://doi.org/10.1016/S0893-6080(98)00116-6

Nesterov: A Better Momentum

We already know where the momentum part will push us in this step. Let's calculate the **new gradient** with that update in mind:

Before:

$$\Delta \mathbf{w}_t := \alpha \cdot \Delta \mathbf{w}_{t-1} + \eta \cdot \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}_t)$$
$$\mathbf{w}_{t+1} := \mathbf{w}_t - \Delta \mathbf{w}_t$$

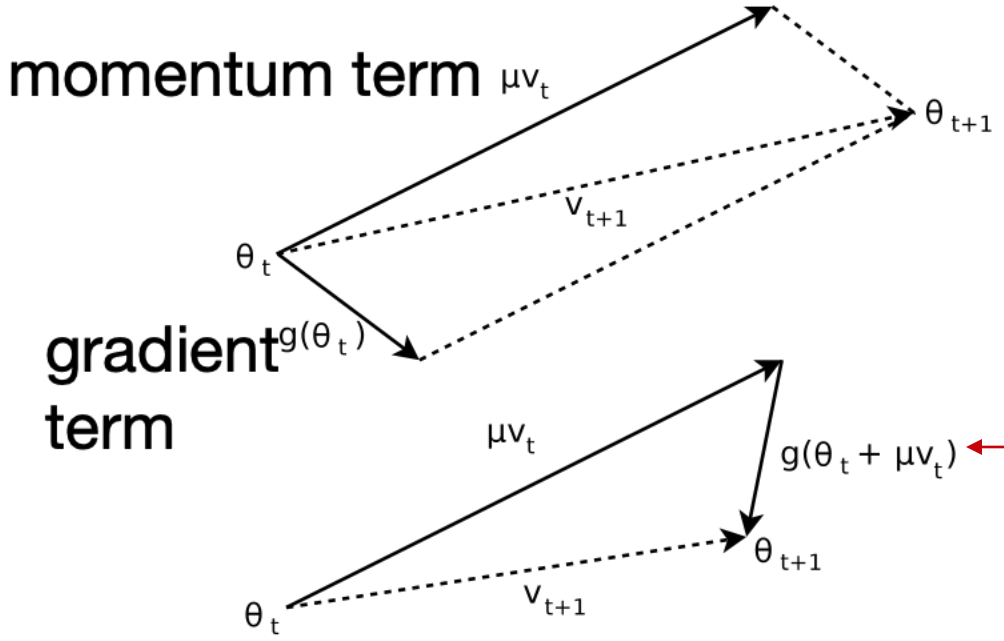
Nesterov:

$$\Delta \mathbf{w}_t := \alpha \cdot \Delta \mathbf{w}_{t-1} + \eta \cdot \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}_t - \alpha \cdot \Delta \mathbf{w}_{t-1})$$
$$\mathbf{w}_{t+1} := \mathbf{w}_t - \Delta \mathbf{w}_t$$

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

Sutskever, I., Martens, J., Dahl, G. E., & Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. ICML (3), 28(1139-1147), 5.

Nesterov: A Better Momentum



correction term (gradient of the point where you would have ended up via the standard momentum method)

Figure 1. **(Top)** Classical Momentum **(Bottom)** Nesterov Accelerated Gradient

Sutskever, I., Martens, J., Dahl, G. E., & Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. ICML (3), 28(1139-1147), 5.

Adaptive Learning Rates

Many different flavors of adapting the learning rate

Rule of thumb:

1. decrease learning if the gradient changes its direction
2. increase learning if the gradient stays consistent

RMSProp

- Unpublished (but very popular) algorithm by Geoff Hinton
- Based on Rprop [1]
- Very similar to another concept called AdaDelta
- **Main idea:** divide learning rate by an exponentially decreasing moving average of the squared gradients
  - **RMS = "Root Mean Squared"**
  - Takes into account that gradients can vary widely in magnitude
  - Damps oscillations like momentum (in practice, works better)

[1] Igel, Christian, and Michael Hüsken. "Improving the Rprop learning algorithm." Proceedings of the Second International ICSC Symposium on Neural Computation (NC 2000). Vol. 2000. ICSC Academic Press, 2000.

ADAM (Adaptive Moment Estimation)

- Probably the most widely used optimization algorithm in DL
- Combination of momentum + RMSProp

Momentum-like term:

$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t-1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

m_t m_{t-1}

$$m_t := \alpha \cdot m_{t-1} + (1-\alpha) \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

RMSProp term:

$$r := \beta \cdot MeanSquare(w_{i,j}, t-1) + (1-\beta)\left(\frac{\partial \mathcal{L}}{\partial w_{i,j}(t)}\right)^2$$

ADAM update:

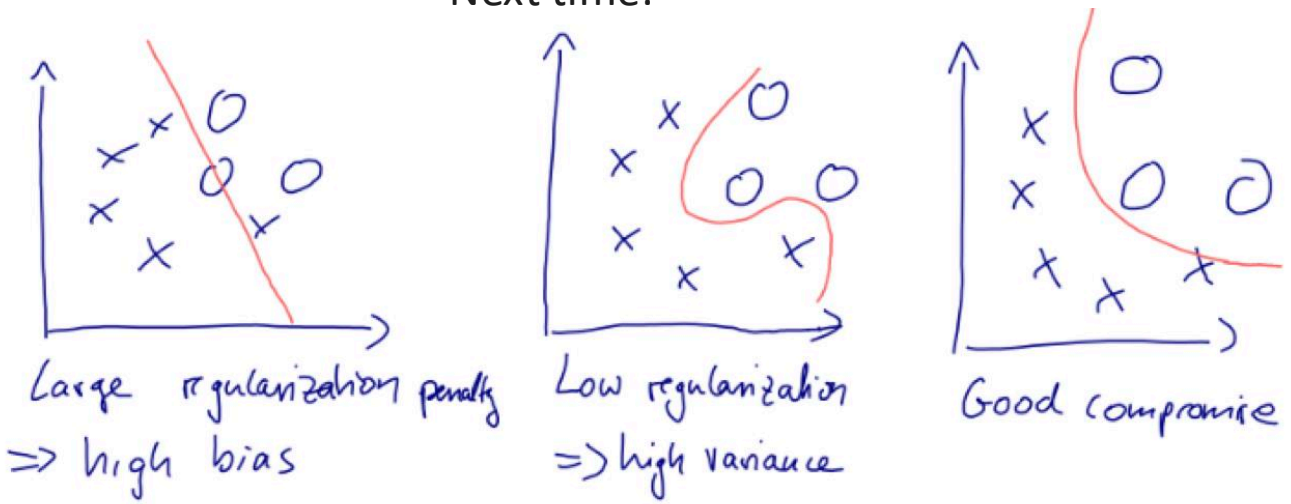
$$w_{i,j} := w_{i,j} - \eta \frac{m_t}{\sqrt{r} + \epsilon}$$

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Where we are…

- Good news: We can solve non-linear problems!
- Bad news: Our multilayer neural networks have lots of parameters and it's easy to overfit the data…

Next time:



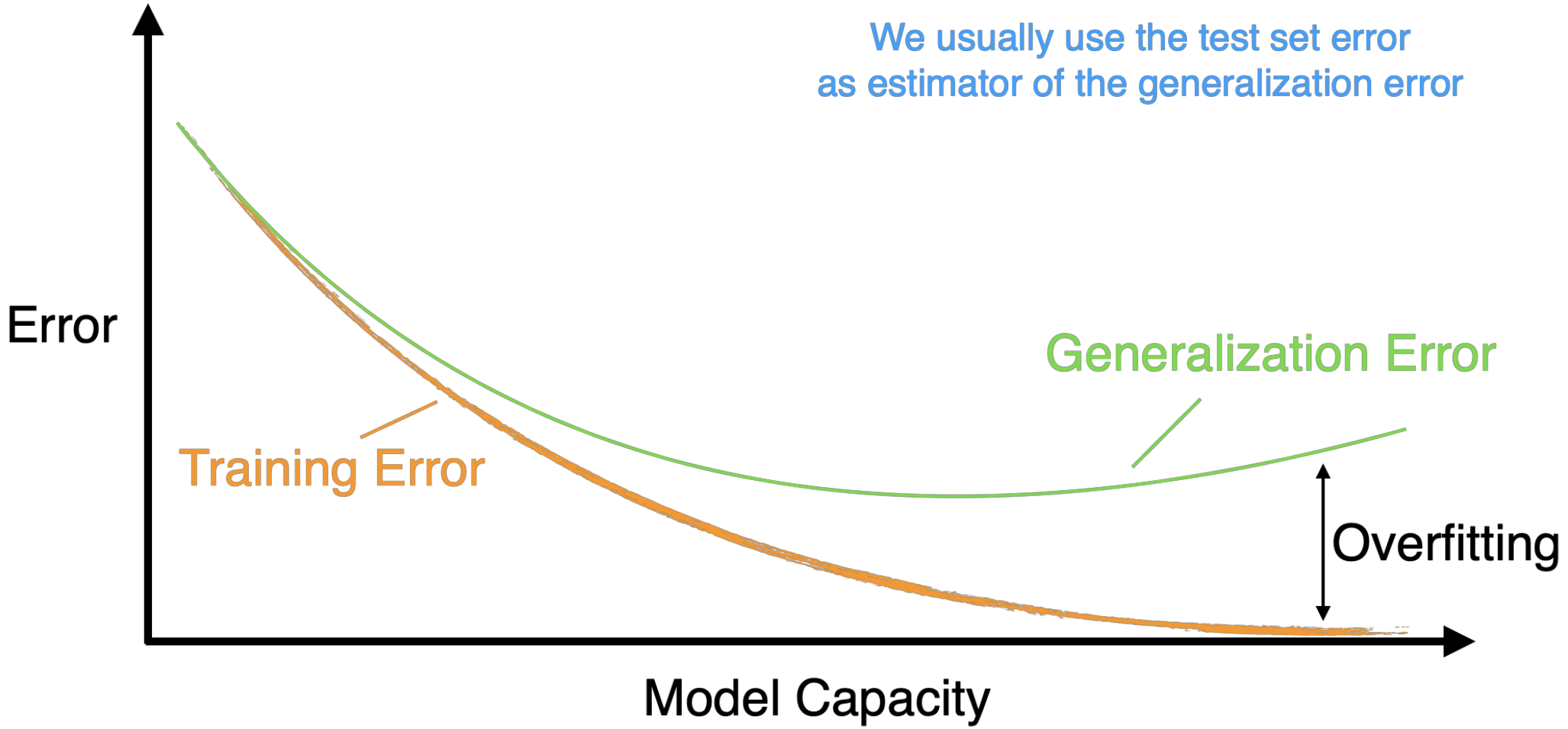
Regularization

Parameters vs Hyperparameters

weights (weight parameters)
biases (bias units)

minibatch size
data normalization schemes
number of epochs
number of hidden layers
number of hidden units
learning rates
(random seed, why?)
loss function
various weights (weighting terms)
activation function types
regularization schemes (more later)
weight initialization schemes (more later)
optimization algorithm type (more later)
…

Overfitting and Underfitting



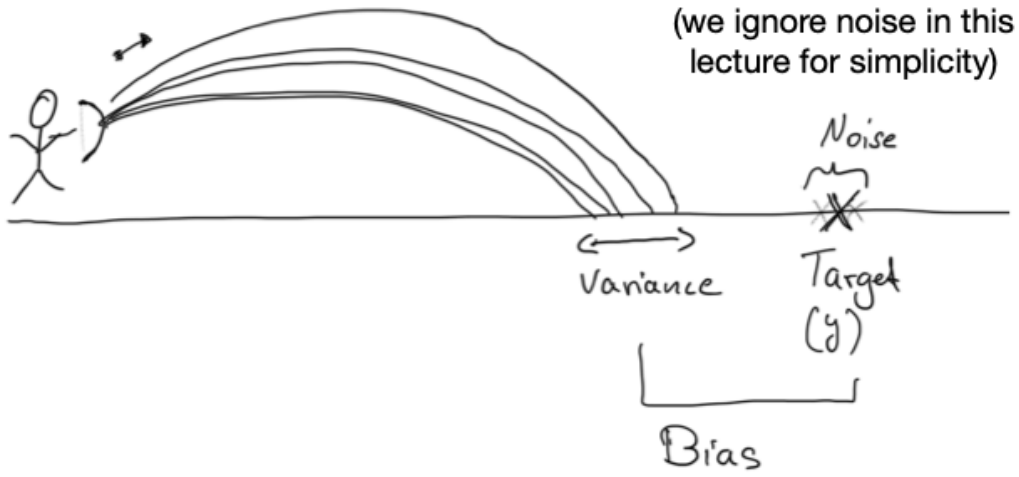
Bias-Variance Decomposition

General Definition:

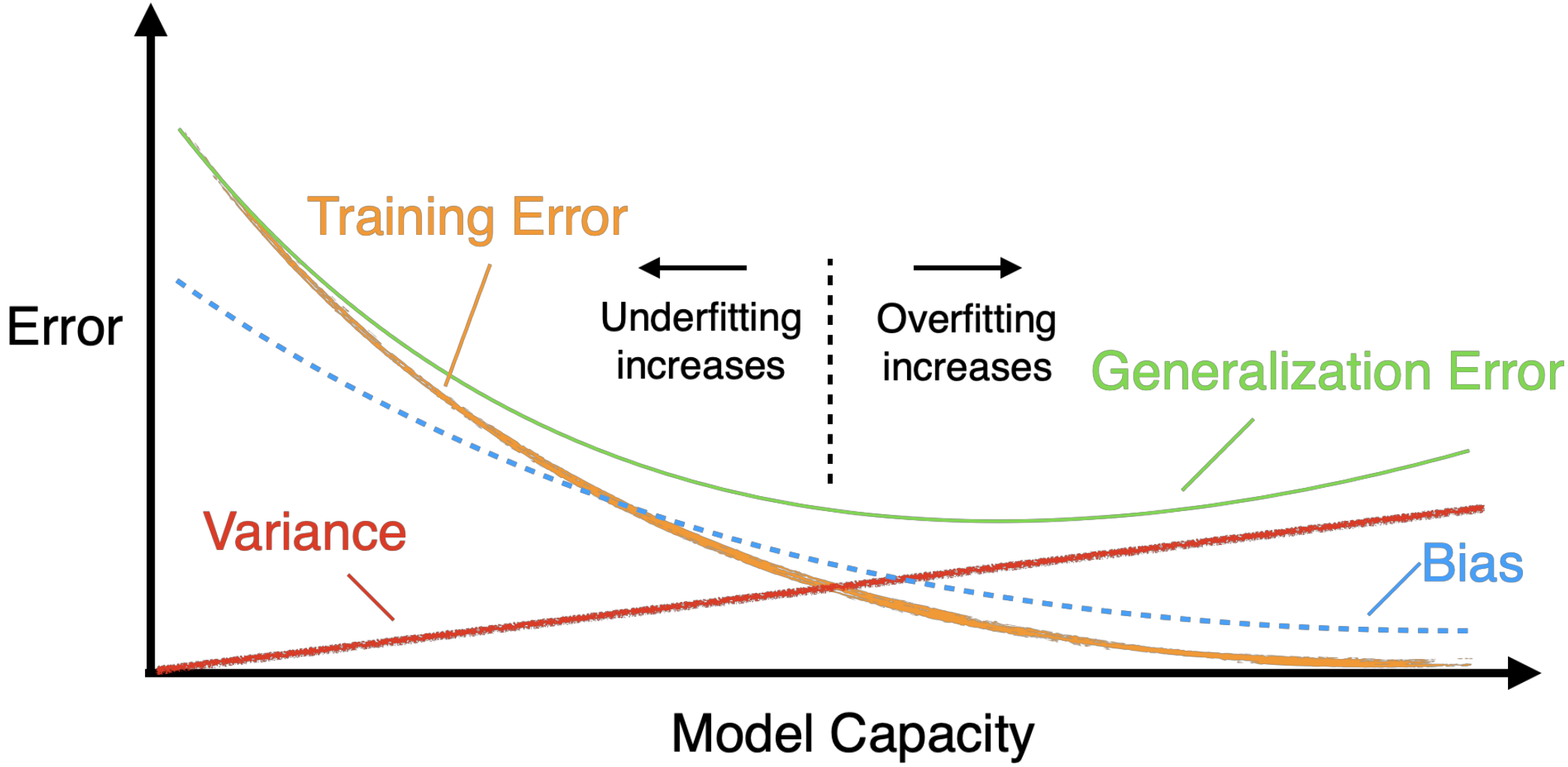
$$\text{Bias}_\theta[\hat{\theta}] = E[\hat{\theta}] - \theta$$

$$\text{Var}_\theta[\hat{\theta}] = E\left[\hat{\theta}^2\right] - (E[\hat{\theta}])^2$$

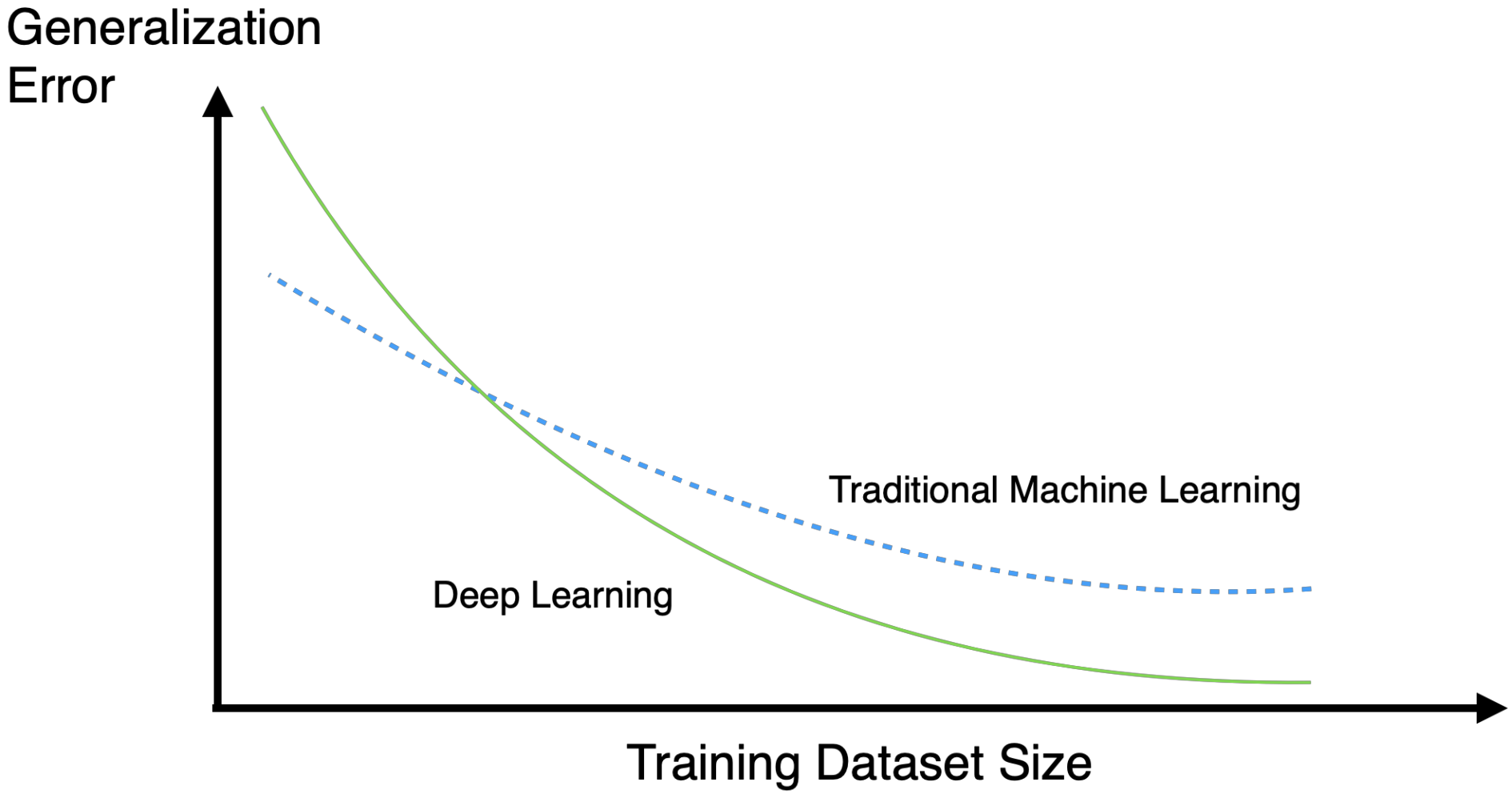
Intuition:



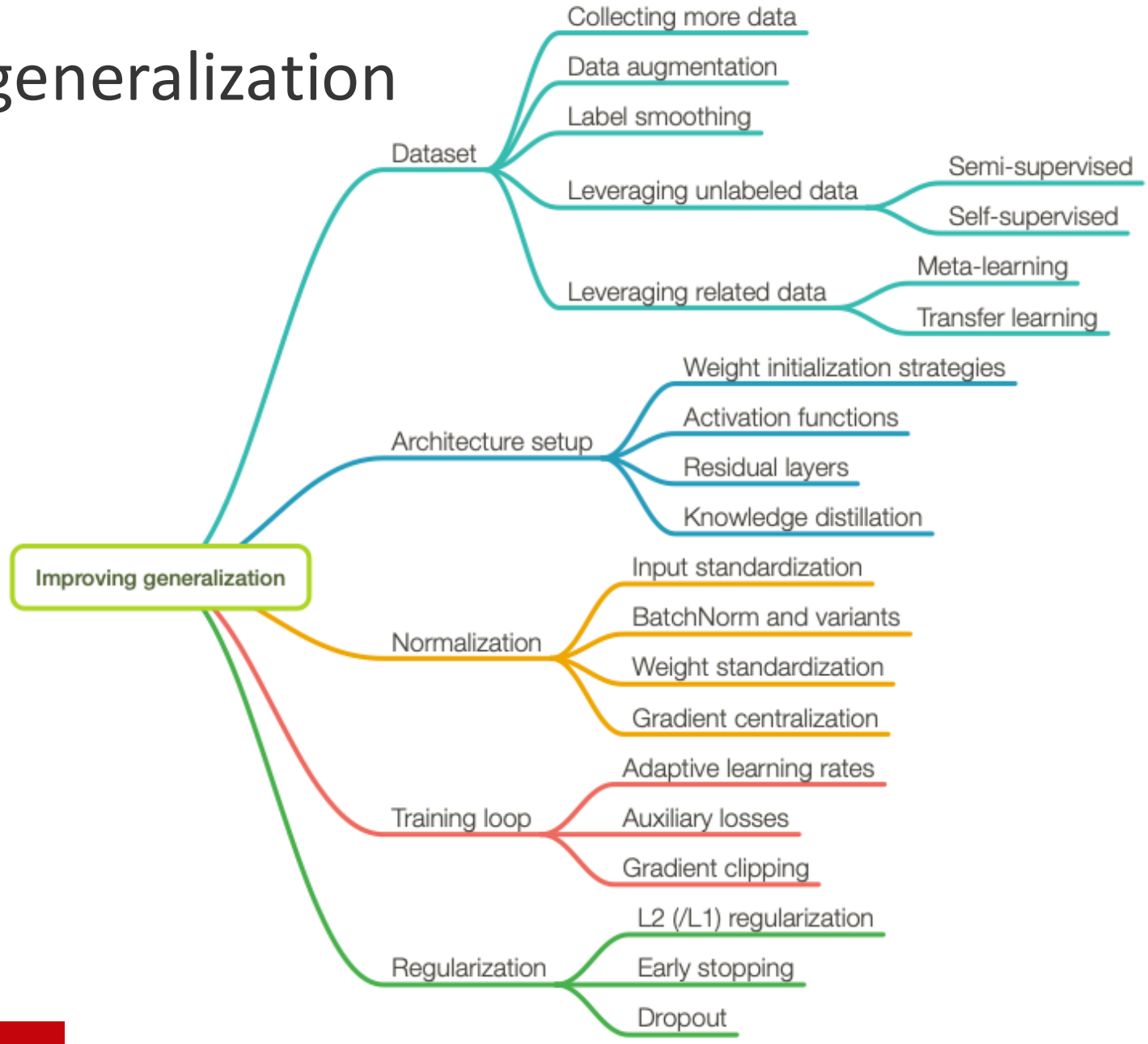
Bias-Variance & Overfitting-Underfitting



Deep Learning works best with large datasets



Many ways to improve generalization



General Strategies to Avoid Overfitting

- Collecting more data, especially high-quality data, is best & always recommended
  - Alternatively: semi-supervised learning, transfer learning, and self-supervised learning
- Data augmentation is helpful
  - Usually requires prior knowledge about data or tasks
- Reducing model capacity can help

Data Augmentation

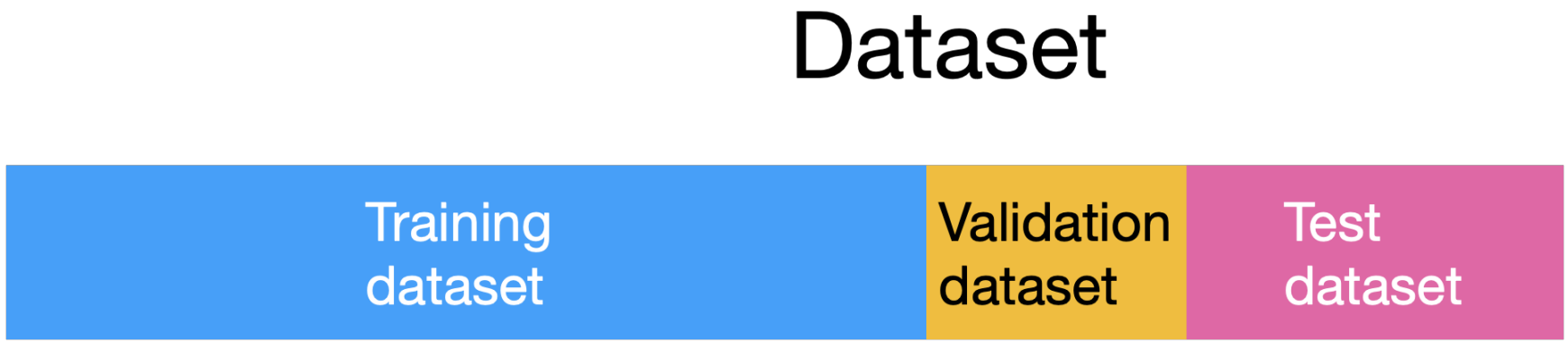
- **Key Idea:** If we know the label shouldn't depend on a transformation h(x), then we can generate new training data $h(x^i), y^i$
- But we must already know something that our outcome doesn't depend on
- Example: image classification
  - rotation, zooming, sepia filter, etc.

Reduce Network's Capacity

- **Key Idea:** The simplest model that matches the outputs should generalize the best
- Choose a smaller architecture: fewer hidden layers & units, add dropout, use ReLU + L1 penalty to prune dead activations,e tc.
- Enforce smaller weights: Early stopping, L2 norm penalty
- Add noise: Dropout
- Note: With recent LLMs and foundation models, it's possible to use a large pretrained model and perform efficient **fine-tuning** (updating small number of parameters of a large model)

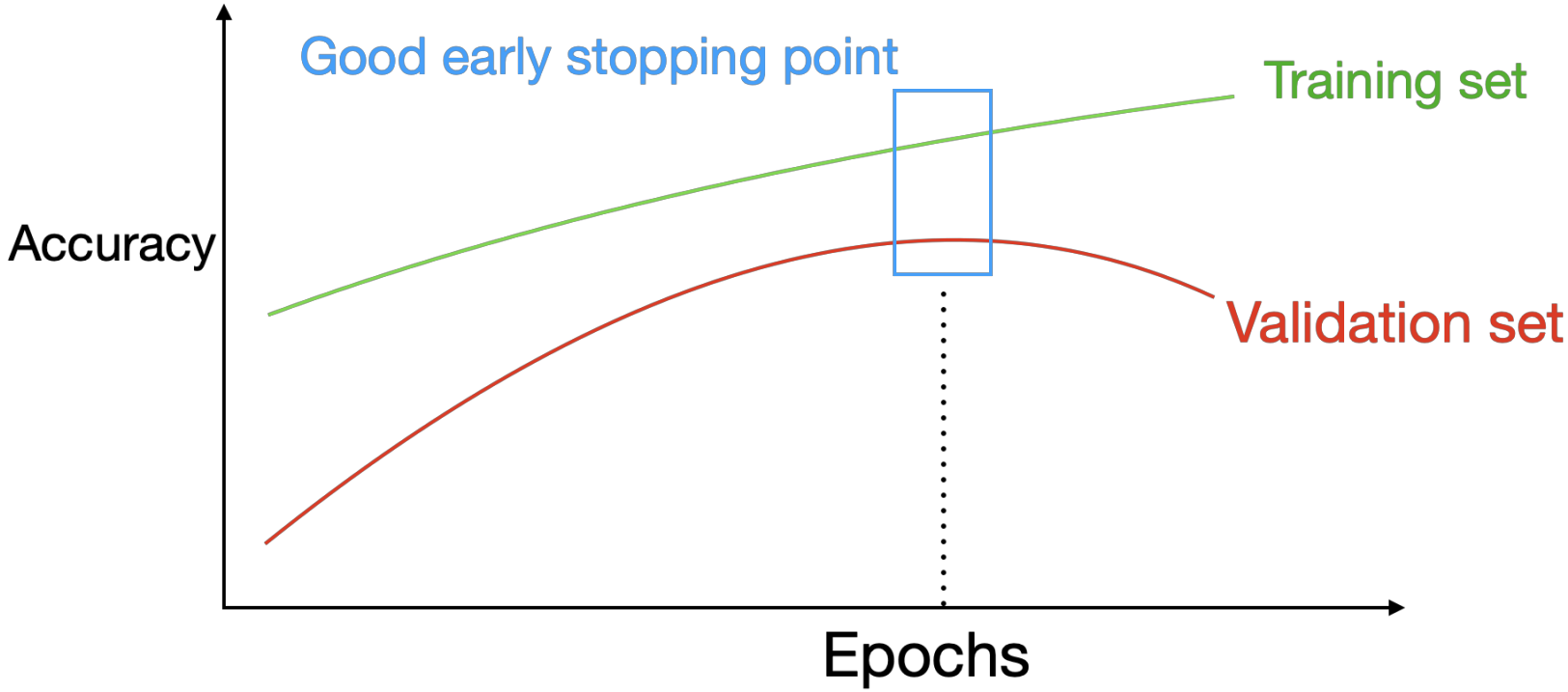
Early Stopping

- **Step 1:** Split your dataset into 3 parts (as always)
  - Use test set only once at the end
  - Use validation accuracy for tuning

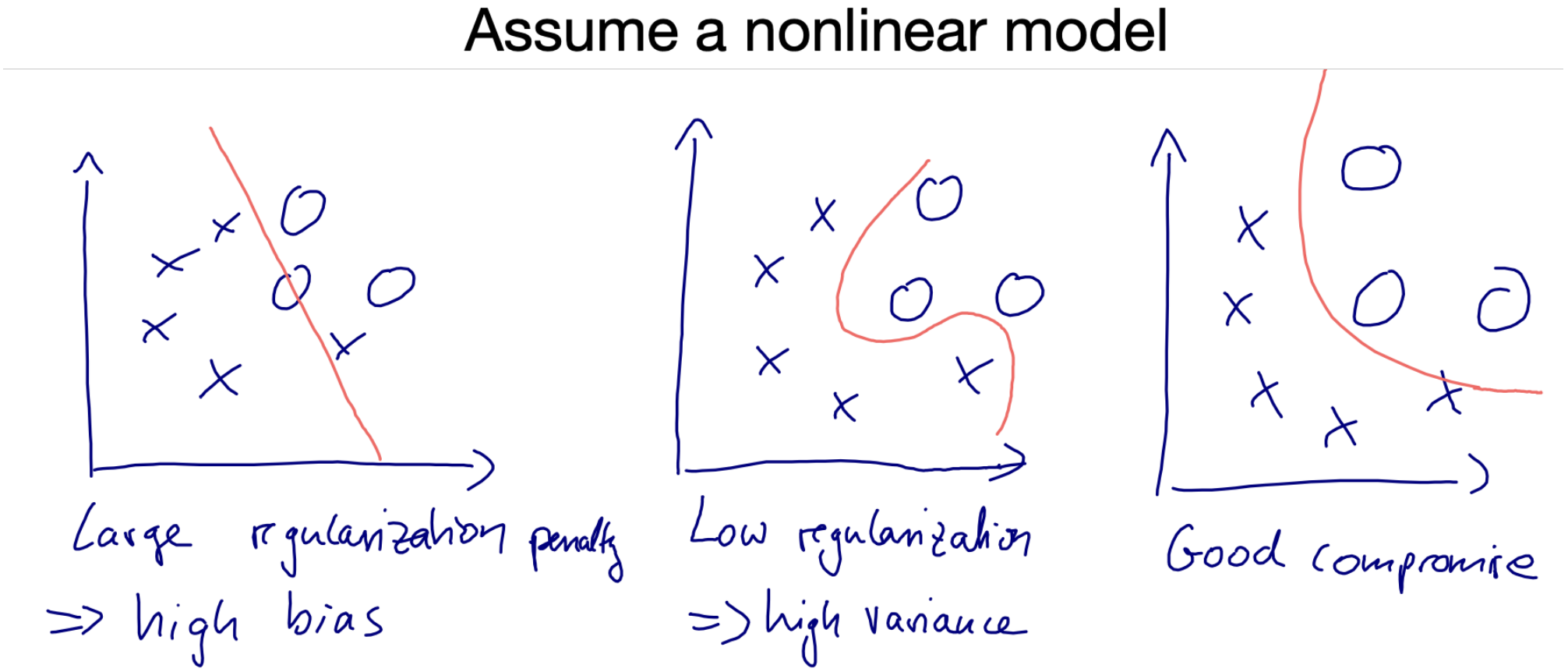


Early Stopping

- **Step 2:** Stop training early
  - Reduce overfitting by observing the training/validation accuracy gap during training and then stop at the “right” point



Effect of Regularization on Decision Boundary



L2 regularization for Multilayer Neural Networks

$$\text{L2-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{l=1}^{L}||\mathbf{w}^{(l)}||_F^2$$

sum over layers

where $||\mathbf{w}^{(l)}||_F^2$ is the Frobenius norm (squared):

$$||\mathbf{w}^{(l)}||_F^2 = \sum_i \sum_j (w_{i,j}^{(l)})^2$$

L2 regularization for Multilayer Neural Networks

Regular gradient descent update:

$$w_{i,j} := w_{i,j} - \eta \frac{\partial \mathcal{L}}{\partial w_{i,j}}$$

Gradient descent update with L2 regularization:

$$w_{i,j} := w_{i,j} - \eta \left(\frac{\partial \mathcal{L}}{\partial w_{i,j}} + \frac{2\lambda}{n} w_{i,j} \right)$$

L2 regularization for Neural Networks in PyTorch

Manually:

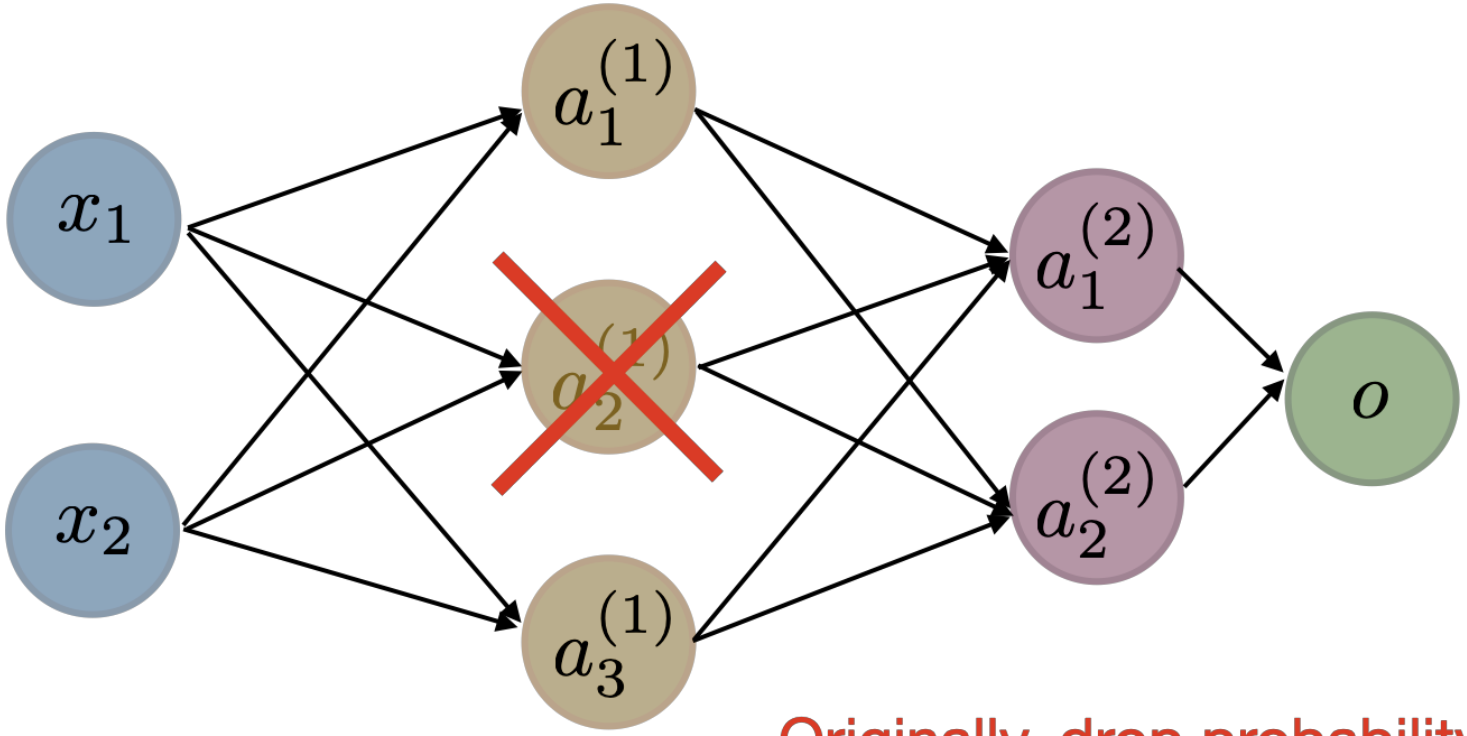
```
        # regularize loss
        L2 = 0.
        for name, p in model.named_parameters():
            if 'weight' in name:
                L2 = L2 + (p**2).sum()

        cost = cost + 2./targets.size(0) * LAMBDA * L2

        optimizer.zero_grad()
        cost.backward()
```

L2 regularization for Neural Networks in PyTorch

Automatically:

```
#########################################################
## Apply L2 regularization
optimizer = torch.optim.SGD(model.parameters(),
                            lr=0.1,
                            weight_decay=LAMBDA)
#--------------------------------------------------------


for epoch in range(num_epochs):

    #### Compute outputs ####
    out = model(X_train_tensor)

    #### Compute gradients ####
    cost = F.binary_cross_entropy(out, y_train_tensor)
    optimizer.zero_grad()
    cost.backward()
```

Dropout



Originally, drop probability 0.5

(but 0.2-0.8 also common now)

Dropout

- How do we drop node activations practically / efficiently?

Bernoulli Sampling (during training):

- p := drop probability
- v := random sample from uniform distribution in range [0, 1]
- $\forall i \in \mathbf{v} : v_i := 0 \text{ if } v_i < p \text{ else } 1$
- $\mathbf{a} := \mathbf{a} \odot \mathbf{v}$ *($p \times 100\%$ of the activations a will be zeroed)*

Then, after training when making predictions (during "inference")

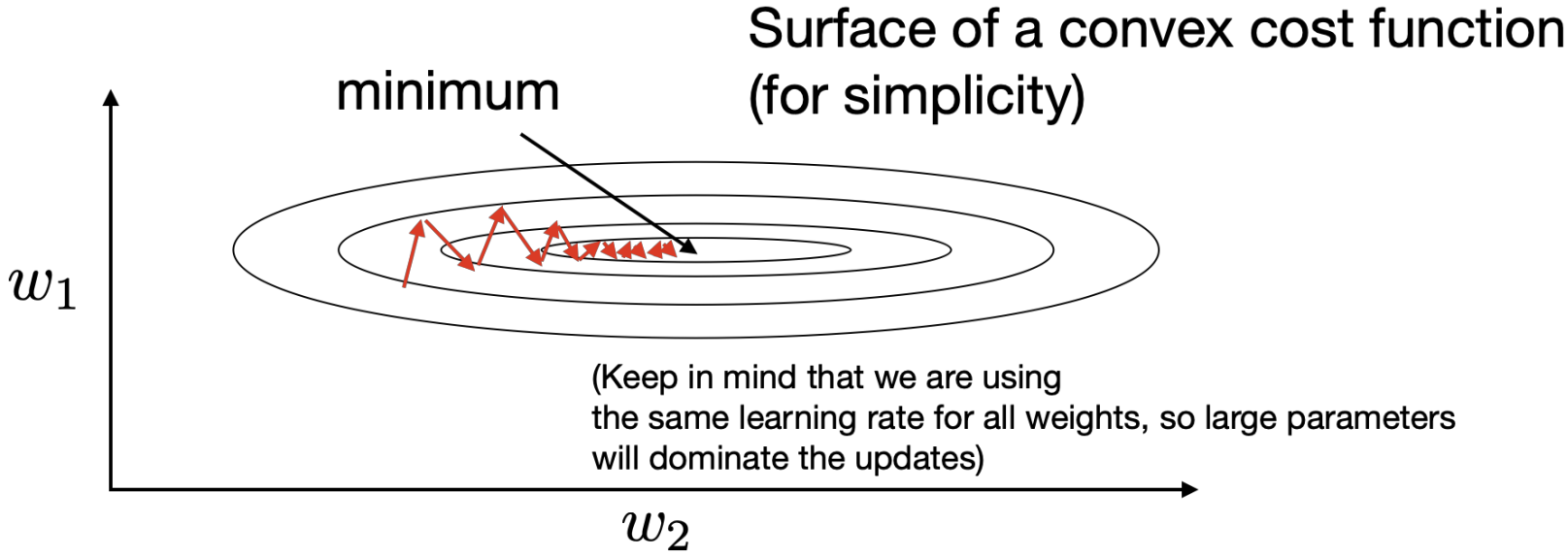
scale activations via $\mathbf{a} := \mathbf{a} \odot (1 - p)$

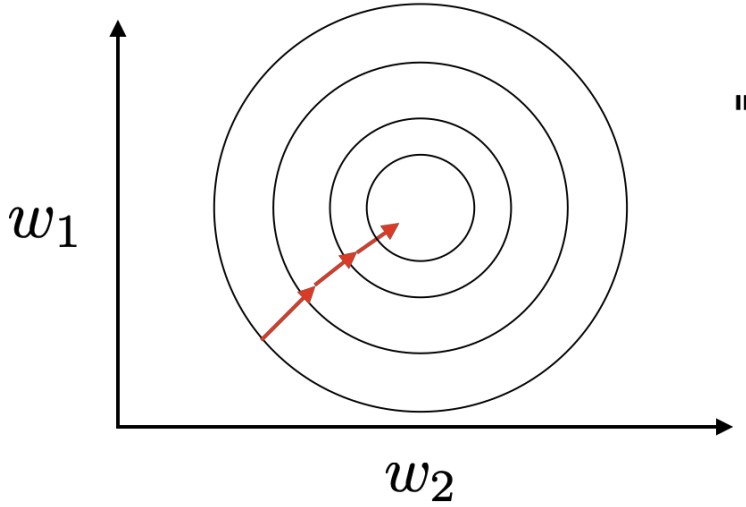
Dropout in PvTorch

```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super().__init__()

        self.my_network = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Flatten(),
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Dropout(drop_proba),
            # 2nd hidden layer
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Dropout(drop_proba),
            # output layer
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        return logits
```

Normalization

Normalization and gradient descent





"Standardization" of input features

$$x_j'^{[i]} = \frac{x_j^{[i]} - \mu_j}{\sigma_j}$$

(scaled feature will have zero mean, unit variance)

In deep models…

Normalizing the **inputs** only affects the first hidden layer…what about the rest?

Batch Normalization ("BatchNorm")

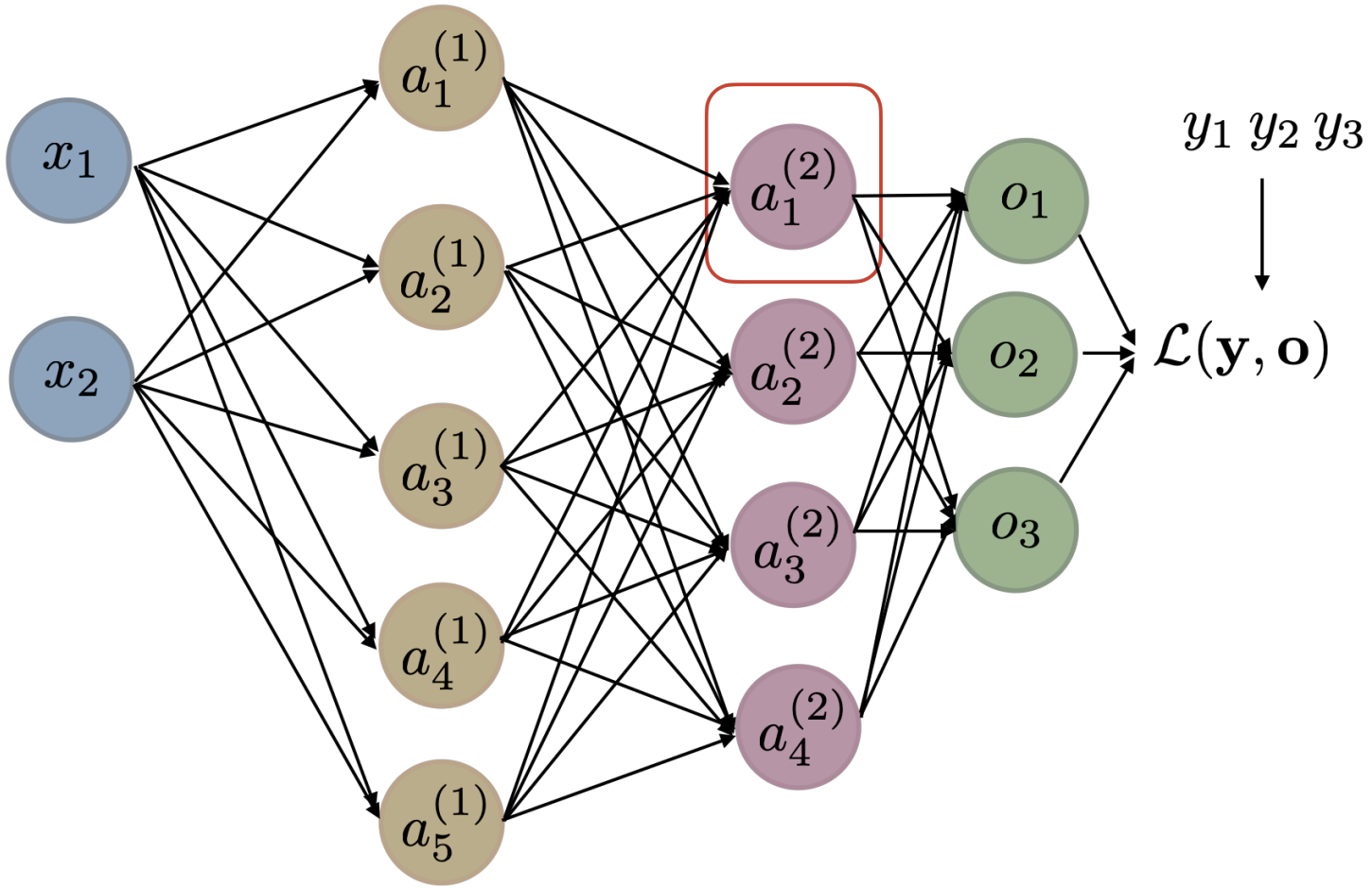
Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448-456).

http://proceedings.mlr.press/v37/ioffe15.html

- Normalizes hidden layer inputs
- Helps with exploding/vanishing gradient problems
- Can increase training stability and convergence rate
- Can be understood as additional (normalization) layers (with additional parameters)

Batch Normalization ("BatchNorm")

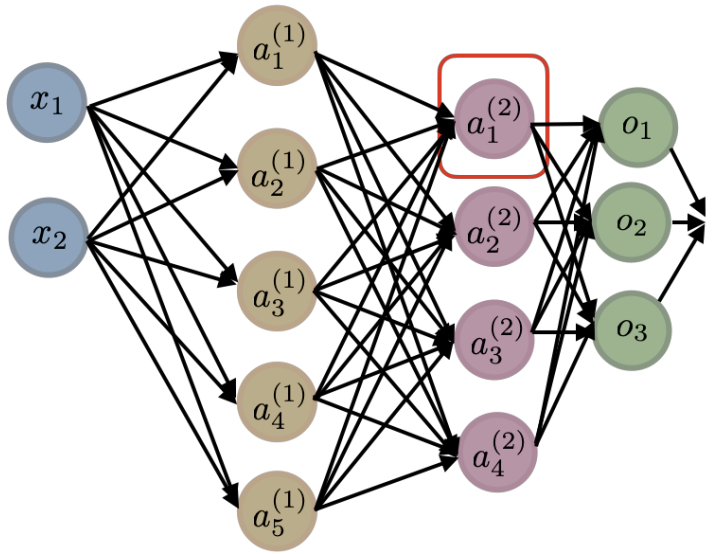
Suppose, we have net input $z_1^{(2)}$
associated with an activation in the 2nd hidden layer



Batch Normalization ("BatchNorm")

Now, consider all examples in a minibatch such that the net input of a given training example at layer 2 is written as $z_1^{(2)[i]}$

where $i \in \{1, ..., n\}$



In the next slides, let's omit the layer index, as it may be distracting...

BatchNorm Step 1: Normalize Net Inputs

$$\mu_j = \frac{1}{n}\sum_i z_j^{[i]}$$

$$\sigma_j^2 = \frac{1}{n}\sum_i (z_j^{[i]} - \mu_j)^2$$

$$z'^{[i]}_j = \frac{z_j^{[i]} - \mu_j}{\sigma_j}$$

In practice:

$$z'^{[i]}_j = \frac{z_j^{[i]} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

For numerical stability, where epsilon is a small number like 1E-5

BatchNorm Step 2: Pre-Activation Scaling

$$z'^{[i]}_j = \frac{z^{[i]}_j - \mu_j}{\sigma_j}$$

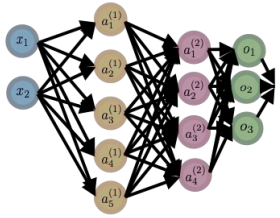
$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

Controls the mean

Controls the spread or scale

Technically, a BatchNorm layer could learn to perform "standardization" with zero mean and unit variance

BatchNorm Steps 1+2 Together



$$z'^{[i]}_j = \frac{z^{[i]}_j - \mu_j}{\sigma_j}$$

$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

$x_1 \rightarrow z^{(1)}_1 \rightarrow z'^{(1)}_1 \rightarrow a'^{(1)}_1 \rightarrow a^{(1)}_1$ first hidden layer

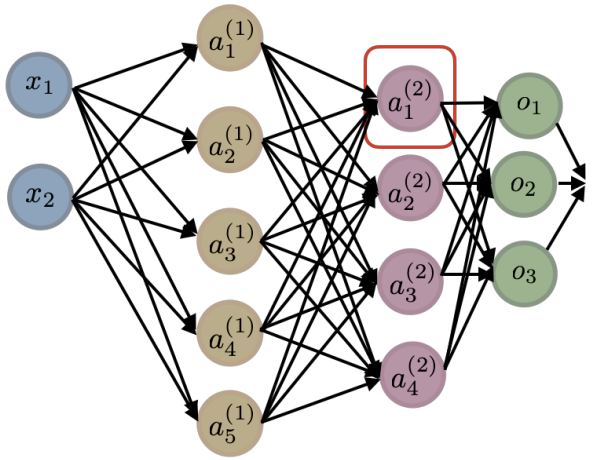
$z^{(2)}_1 \rightarrow z'^{(2)}_1 \rightarrow a'^{(2)}_1 \rightarrow a^{(2)}_1$ second hidden layer

...

BatchNorm Steps 1+2 Together

$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

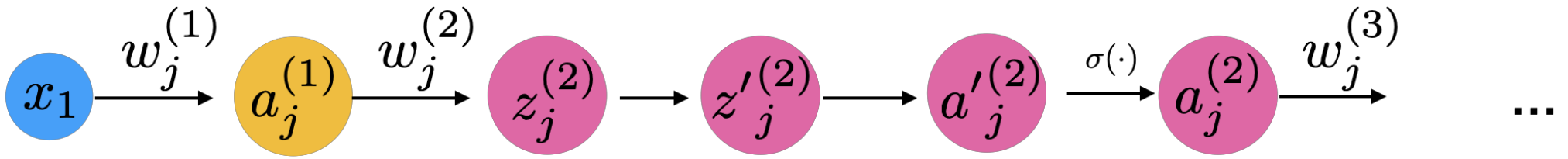
This parameter makes the bias units redundant



Also, note that the batchnorm parameters are vectors with the same number of elements as the bias vector

BatchNorm and Backprop

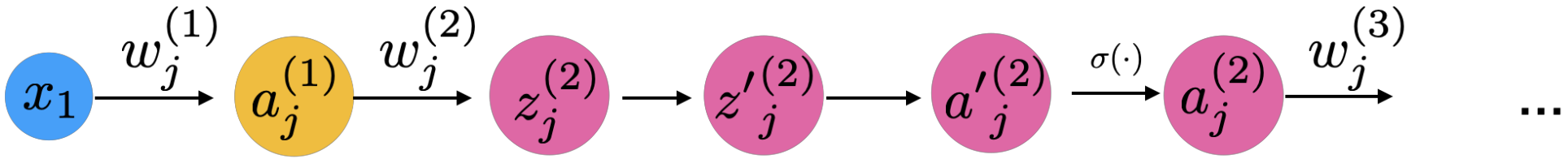
$$z'^{(2)}_j = \frac{z^{(2)}_j - \mu_j}{\sigma_j} \qquad a'^{(2)}_j = \gamma_j \cdot z'^{(2)}_j + \beta_j$$



$$\frac{\partial l}{\partial \beta_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j} \cdot \frac{\partial a'^{(2)[i]}_j}{\partial \beta_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j}$$

$$\frac{\partial l}{\partial \gamma_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j} \cdot \frac{\partial a'^{(2)[i]}_j}{\partial \gamma_j} = \sum_{i=1}^{n} \frac{\partial l}{\partial a'^{(2)[i]}_j} \cdot z'^{(2)[i]}_j$$

BatchNorm and Backprop



Since the minibatch mean and variance act as parameters, we can/have to apply the multivariable chain rule

$$\frac{\partial l}{\partial z_j^{(2)[i]}} = \frac{\partial l}{\partial z'^{(2)[i]}_j} \cdot \frac{\partial z'^{(2)[i]}_j}{\partial z_j^{(2)[i]}} + \frac{\partial l}{\partial \mu_j} \cdot \frac{\partial \mu_j}{\partial z_j^{(2)[i]}} + \frac{\partial l}{\partial \sigma_j^2} \cdot \frac{\partial \sigma_j^2}{\partial z_j^{(2)[i]}}$$

$$= \frac{\partial l}{\partial z'^{(2)[i]}_j} \cdot \frac{1}{\sigma_j} + \frac{\partial l}{\partial \mu_j} \cdot \frac{1}{n} + \frac{\partial l}{\partial \sigma_j^2} \cdot \frac{2(z_j^{(2)} - \mu_j)}{n}$$

BatchNorm and Backprop

$$\frac{\partial l}{\partial z_j^{(2)[i]}} = \frac{\partial l}{\partial z'^{(2)[i]}_j} \cdot \frac{\partial z'^{(2)[i]}_j}{\partial z_j^{(2)[i]}} + \frac{\partial l}{\partial \mu_j} \cdot \frac{\partial \mu_j}{\partial z_j^{(2)[i]}} + \frac{\partial l}{\partial \sigma_j^2} \cdot \frac{\partial \sigma_j^2}{\partial z_j^{(2)[i]}}$$

$$= \frac{\partial l}{\partial z'^{(2)[i]}_j} \cdot \frac{1}{\sigma_j} + \frac{\partial l}{\partial \mu_j} \cdot \frac{1}{n} + \frac{\partial l}{\partial \sigma_j^2} \cdot \frac{2(z_j^{(2)} - \mu_j)}{n}$$

If you like math & engineering, you can solve the remaining terms as an ungraded HW exercise ;)

BatchNorm in PyTorch

```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super().__init__()

        self.my_network = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Flatten(),
            torch.nn.Linear(num_features, num_hidden_1, bias=False)
            torch.nn.BatchNorm1d(num_hidden_1),
            torch.nn.ReLU(),
            # 2nd hidden layer
            torch.nn.Linear(num_hidden_1, num_hidden_2, bias=False)
            torch.nn.BatchNorm1d(num_hidden_2),
            torch.nn.ReLU(),
            # output layer
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        return logits
```

https://github.com/rasbt/stat453-deep-learningss21/blob/main/L11/code/batchnorm.ipynb

BatchNorm in PyTorch

```
def train_model(model, num_epochs, train_loader,
                valid_loader, test_loader, optimizer, device):

    start_time = time.time()
    minibatch_loss_list, train_acc_list, valid_acc_list = [], [], []
    for epoch in range(num_epochs):

        model.train()
        for batch_idx, (features, targets) in enumerate(train_loader):

            features = features.to(device)
            targets = targets.to(device)

            # ## FORWARD AND BACK PROP
            logits = model(features)
            loss = torch.nn.functional.cross_entropy(logits, targets)
            optimizer.zero_grad()

            loss.backward()

            # ## UPDATE MODEL PARAMETERS
            optimizer.step()

            # ## LOGGING
            minibatch_loss_list.append(loss.item())
            if not batch_idx % 50:
                print(f'Epoch: {epoch+1:03d}/{num_epochs:03d} '
                      f'| Batch {batch_idx:04d}/{len(train_loader):04d} '
                      f'| Loss: {loss:.4f}')

        model.eval()
        with torch.no_grad():  # save memory during inference
            train_acc = compute_accuracy(model, train_loader, device=device)
```

don't forget model.train()
and model.eval()
in training and test loops

BatchNorm at Test-Time

- Use exponentially weighted average (moving average) of mean and variance

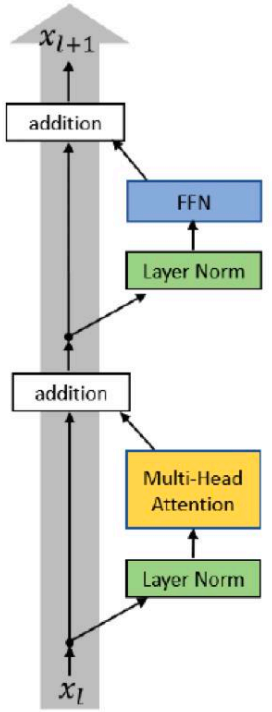
running_mean = momentum * running_mean + (1 - momentum) * sample_mean

(where momentum is typically ~0.1; and same for variance)

- Alternatively, can also use global training set mean and variance

Related: LayerNorm

- Layer normalization (LN)
- BN calculates mean/std based on a mini batch, whereas LN calculates mean/std based on feature/embedding vectors
- In the stats language, BN zero mean unit variance, whereas LN projects feature vector to **unit sphere**
- LN in Transformers



Pre-LN Transformer

$$x_{l,i}^{pre,1} = \text{LayerNorm}(x_{l,i}^{pre})$$
$$x_{l,i}^{pre,2} = \text{MultiHeadAtt}(x_{l,i}^{pre,1}, [x_{l,1}^{pre,1}, \cdots, x_{l,n}^{pre,1}])$$
$$x_{l,i}^{pre,3} = x_{l,i}^{pre} + x_{l,i}^{pre,2}$$
$$x_{l,i}^{pre,4} = \text{LayerNorm}(x_{l,i}^{pre,3})$$
$$x_{l,i}^{pre,5} = \text{ReLU}(x_{l,i}^{pre,4} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$$
$$x_{l+1,i}^{pre} = x_{l,i}^{pre,5} + x_{l,i}^{pre,3}$$

Final LayerNorm: $x_{Final,i}^{pre} \leftarrow \text{LayerNorm}(x_{L+1,i}^{pre})$

Normalize everything?

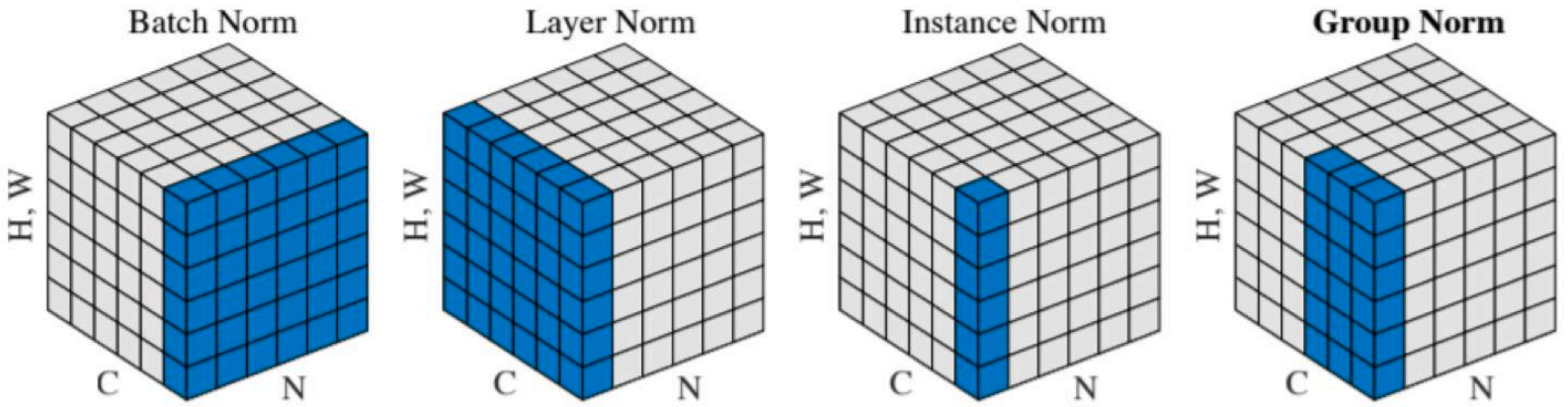


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Wu, Y., & He, K. (2018). Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 3-19).

Initialization

Weight initialization

- Recall: Can't initialize all weights to 0 (symmetry problem)
- But we want weights to be relatively small.
  - Traditionally, we can initialize weights by sampling from a random uniform distribution in range [0, 1], or better, [-0.5, 0.5]
  - Or, we could sample from a Gaussian distribution with mean 0 and small variance (e.g., 0.1 or 0.01)

Xavier Initialization

Method:

- **Step 1:** Initialize weights from Gaussian or uniform distribution
- **Step 2:** Scale the weights proportional to the number of inputs to the layer
  - For the first hidden layer, that is the number of features in the dataset; for the second hidden layer, that is the number of units in the 1st hidden layer, etc.

Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

Xavier Initialization

Rationale behind this scaling:

Variance of the sample (between data points, not variance of the mean) linearly increases as the sample size increases (variance of the sum of independent variables is the sum of the variances); square root for standard deviation

$$\mathrm{Var}\left(z_j^{(l)}\right) = \mathrm{Var}\left(\sum_{j=1}^{m_{l-1}} W_{jk}^{(l)} a_k^{(l-1)}\right)$$

$$= \sum_{j=1}^{m^{(l-1)}} \mathrm{Var}\left[W_{jk}^{(l)} a_k^{(l-1)}\right] = \sum_{i=1}^{m^{(l-1)}} \mathrm{Var}\left[W_{jk}^{(l)}\right] \mathrm{Var}\left[a_k^{(l-1)}\right]$$

$$= \sum_{j=1}^{m^{(l-1)}} \mathrm{Var}\left[W^{(l)}\right] \mathrm{Var}\left[a^{(l-1)}\right] = m^{(l-1)} \mathrm{Var}\left[W^{(l)}\right] \mathrm{Var}\left[a^{(l-1)}\right]$$

He Initialization

- Assuming activations with mean 0, which is reasonable, Xavier Initialization assumes a derivative of 1 for the activation function (which is reasonable for tanH)
- For ReLU, the **activations are not centered at zero**
- He initialization takes this into account
- The result is that we add a scaling factor of $\sqrt{2}$

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} \cdot \sqrt{\frac{2}{m^{(l-1)}}}$$

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In *Proceedings of the IEEE international conference on computer vision*, pp. 1026-1034. 2015.

Convolutional Neural Networks

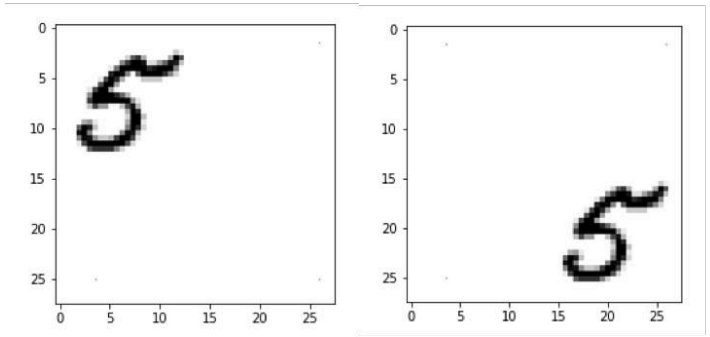
Why images are hard

Different lighting, contrast, viewpoints, etc.

Image Source:
twitter.com%2Fcats&psig=AOvVaw30_o-PCM-K21DiMAJQimQ4&ust=1553887775741551

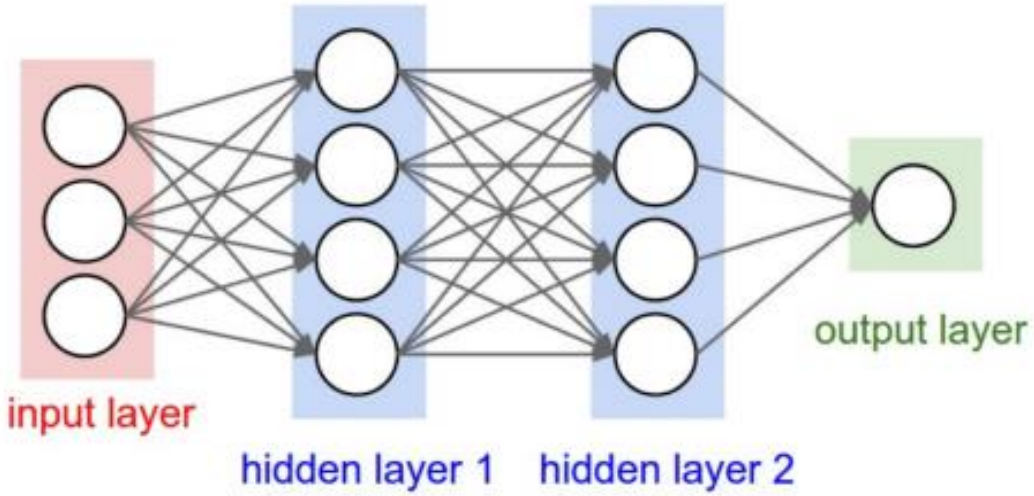
Image Source: https://www.123rf.com/photo_76714328_side-view-of-tabby-cat-face-over-white.html

Or even simple translation



Do deep fully-connected nets solve this?

Full connectivity is a problem for large inputs

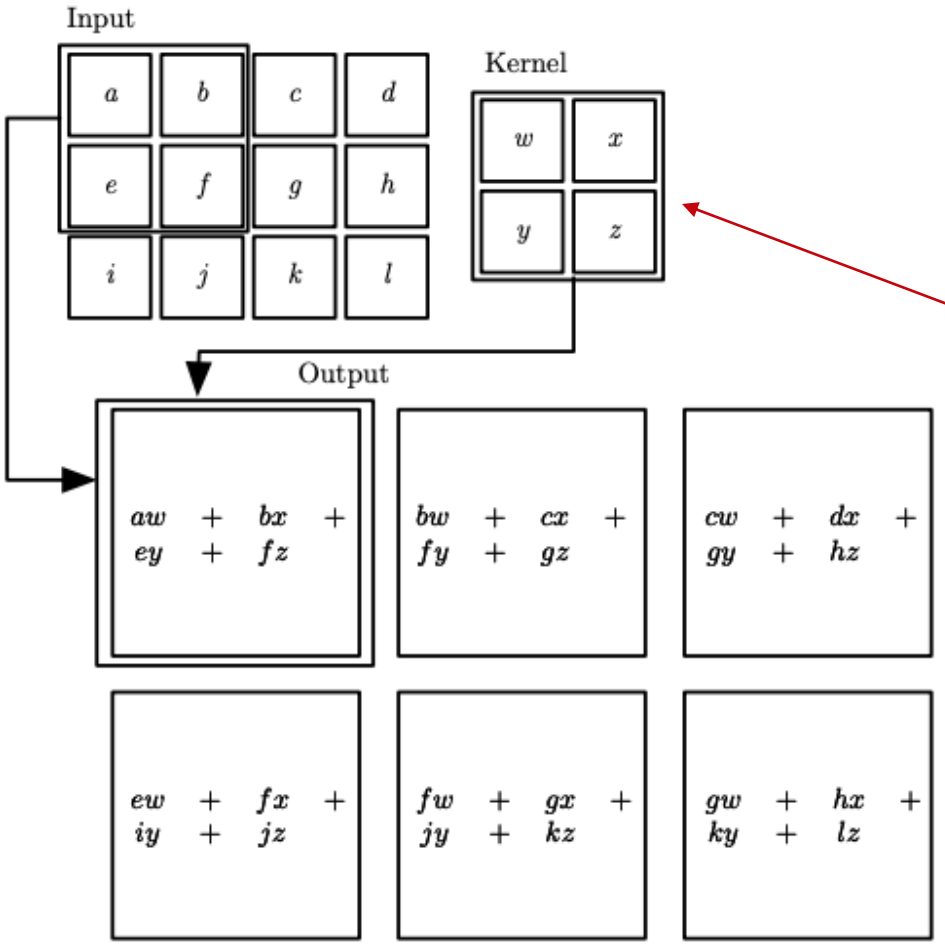


- 3x200x200 images imply **120,000** weights per neuron in first hidden layer

Convolutional Neural Networks [LeCun 1989]

- Let's share parameters.
- Instead of learning position-specific weights, learn weights defined for **relative positions**
  - Learn "filters" that are reused across the image
  - Generalize across spatial translation of input
- Key idea:
  - Replace matrix multiplication in neural networks with a convolution
- Later, we will see that this can work for any graph-structured data, not just images.

Weight sharing in kernels



Sliding filters (kernels)

Reused weights (small)!

Fig. Goodfellow et al. 2016

Convolutional Neural Networks [LeCun 1989]

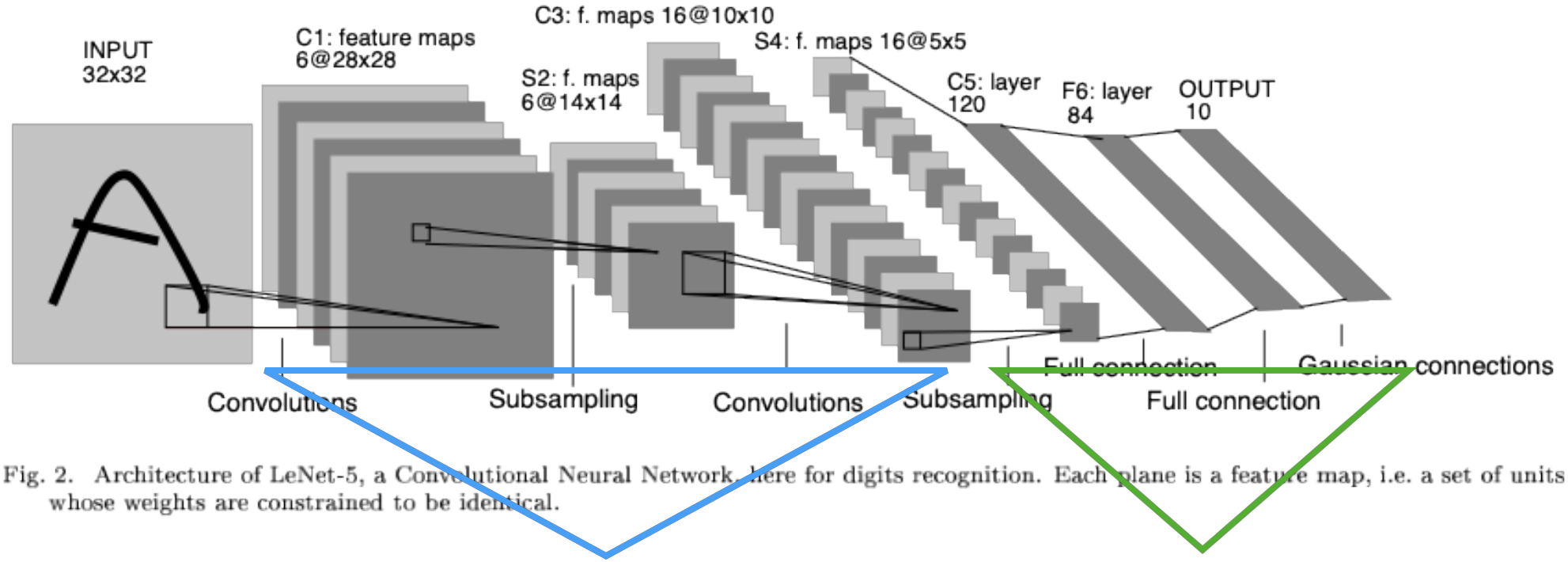


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

"Automatic feature extractor" "Regular classifier"

Yann LeCun, Léon Bottou, Yoshua Bengio and Patrick Haffner: Gradient Based Learning Applied to Document Recognition, Proceedings of IEEE, 86(11):2278–2324, 1998.

Convolutional Neural Networks [LeCun 1989]

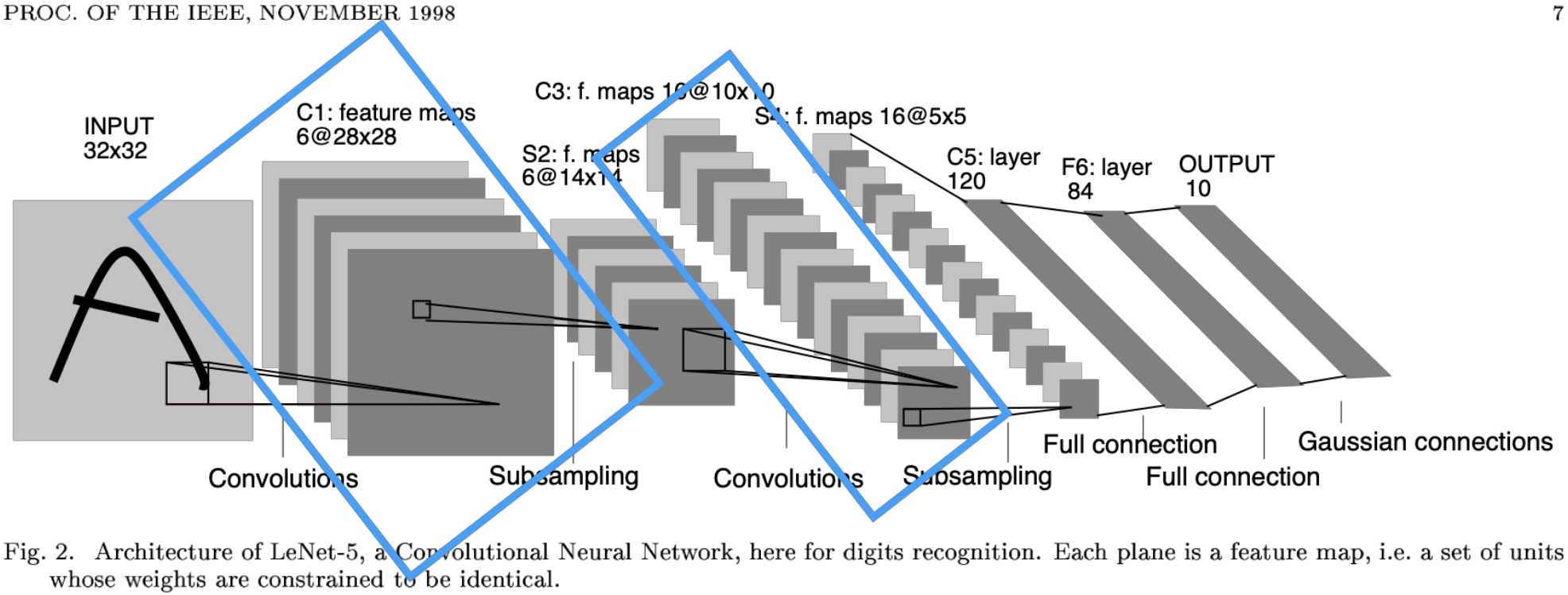


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Each "bunch" of feature maps represents one hidden layer in the neural network.

Counting the FC layers, this network has **5** layers

Convolutional Neural Networks [LeCun 1989]

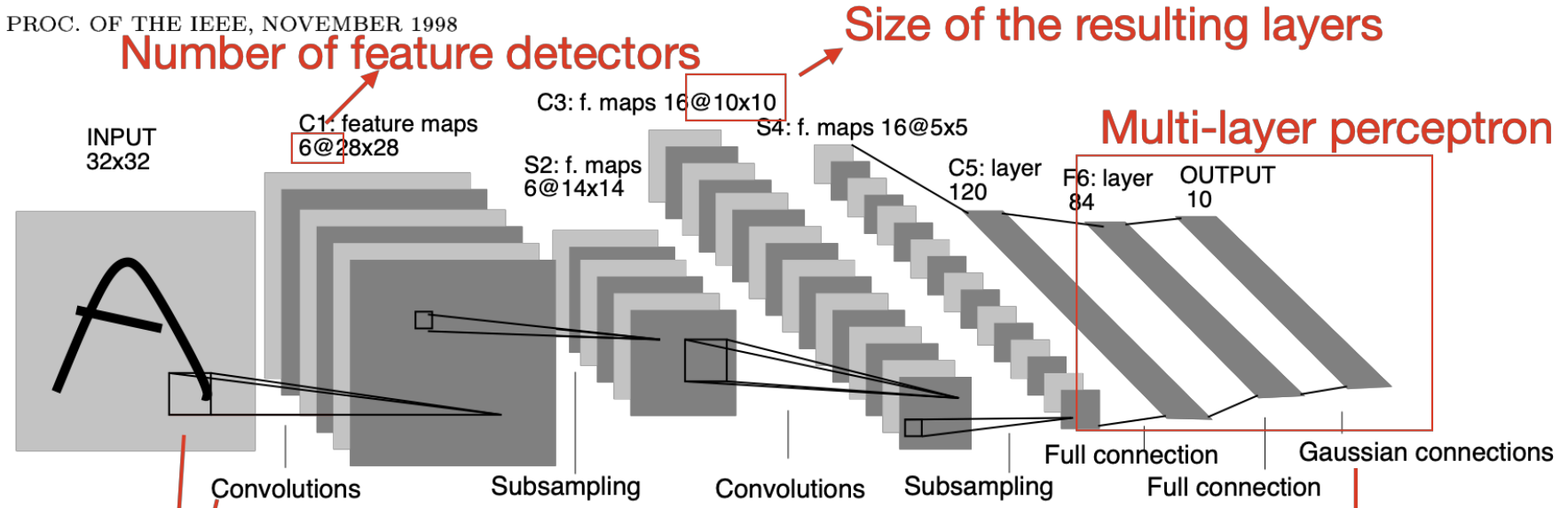


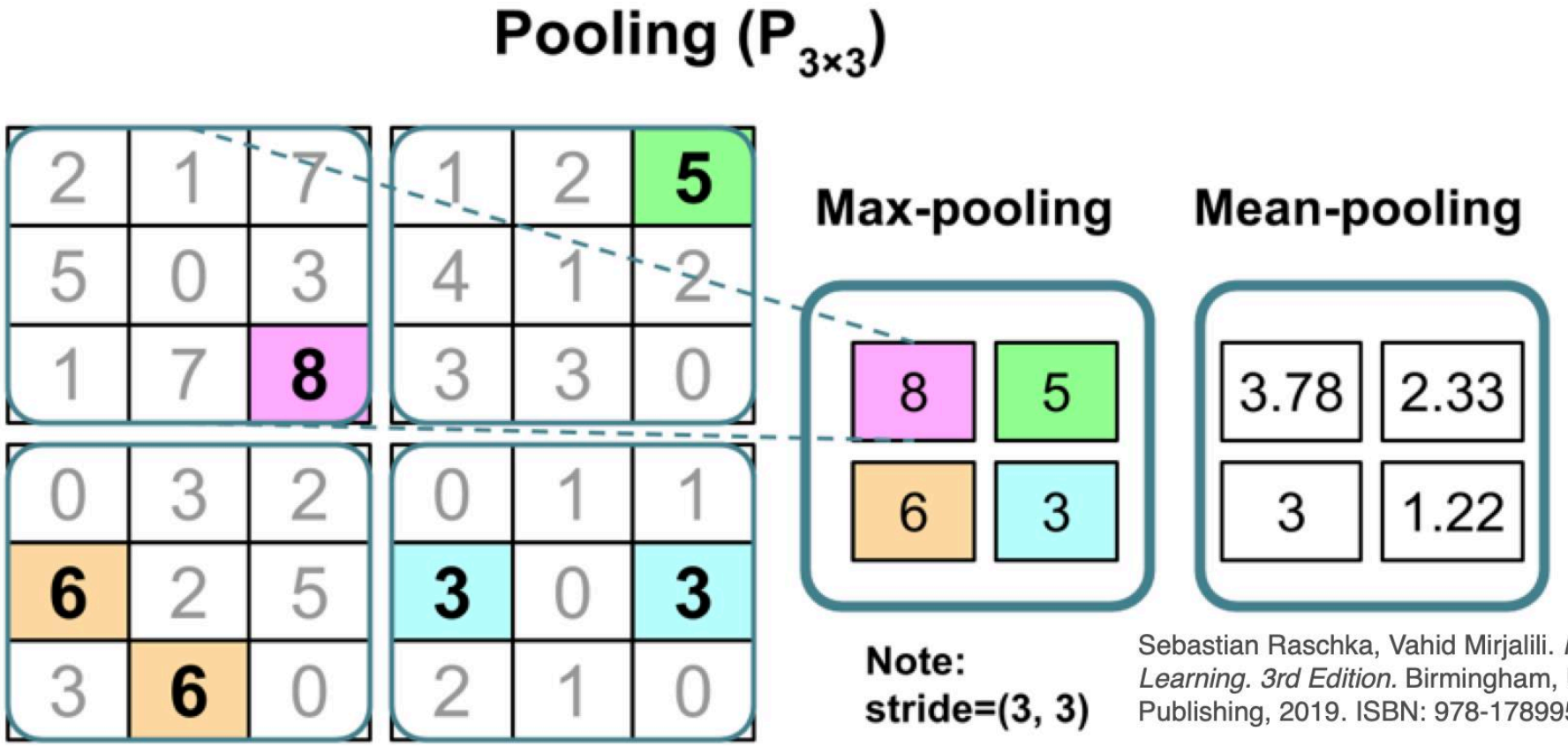
Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

"Feature detectors" (weight matrices)
that are being reused ("weight sharing")
=> also called "kernel" or "filter"

nowadays called "pooling"

basically a fully-connected
layer + MSE loss
(nowadays common to use
fc-layer + softmax
+ cross entropy)

“Pooling”: lossy compression



Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Birmingham, UK: Packt Publishing, 2019. ISBN: 978-1789955750

Main ideas of CNNs

- **Sparse-connectivity:** A single element in the feature map is connected to only a small patch of pixels. (This is very different from connecting to the whole input image, in the case of multi-layer perceptrons.)
- **Parameter-sharing:** The same weights are used for different patches of the input image.
- **Many layers:** Combining extracted local patterns to global patterns

CNNs give sparse connectivity

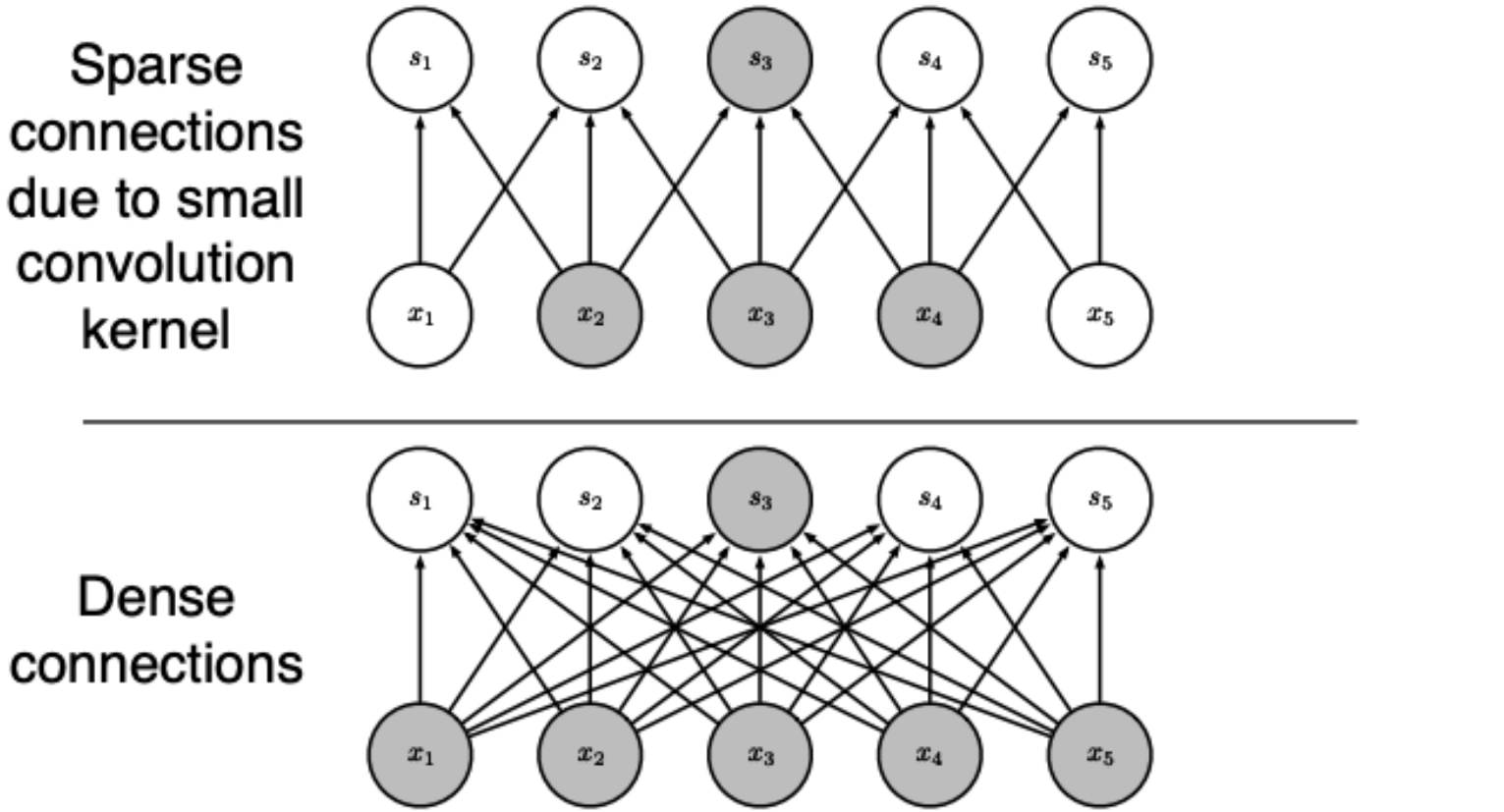


Figure 9.3

(Goodfellow 2016)

Receptive fields grow over depth

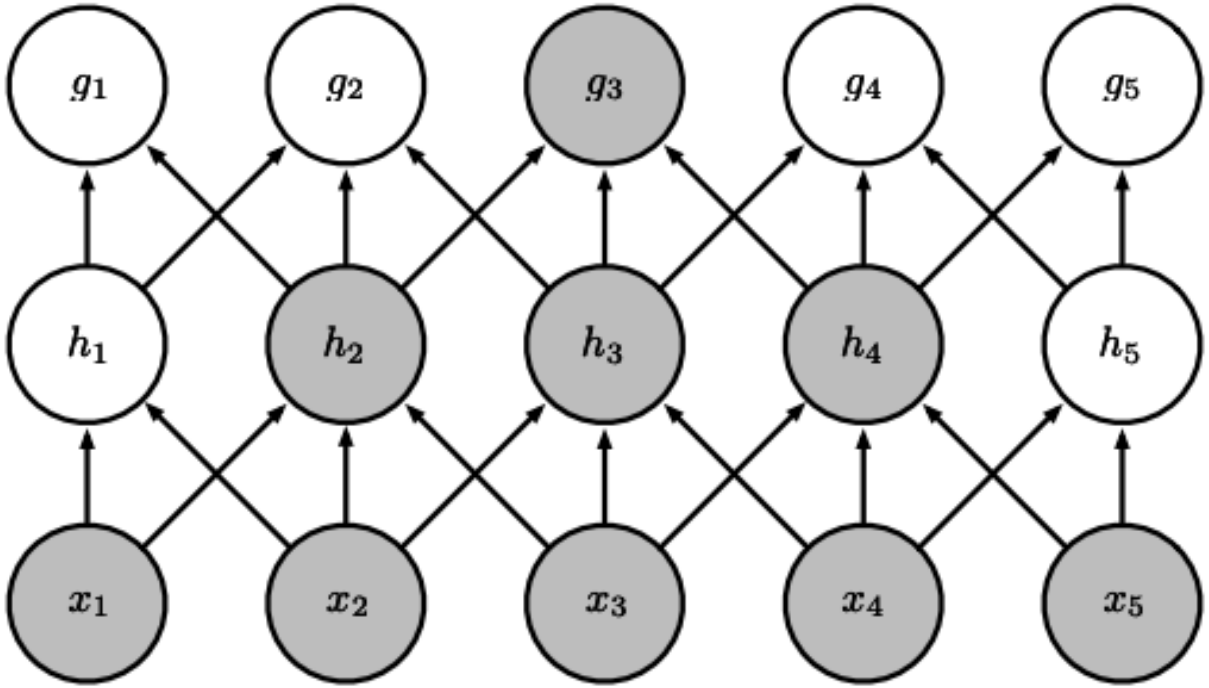


Figure 9.4

(Goodfellow 2016)

Parameter sharing

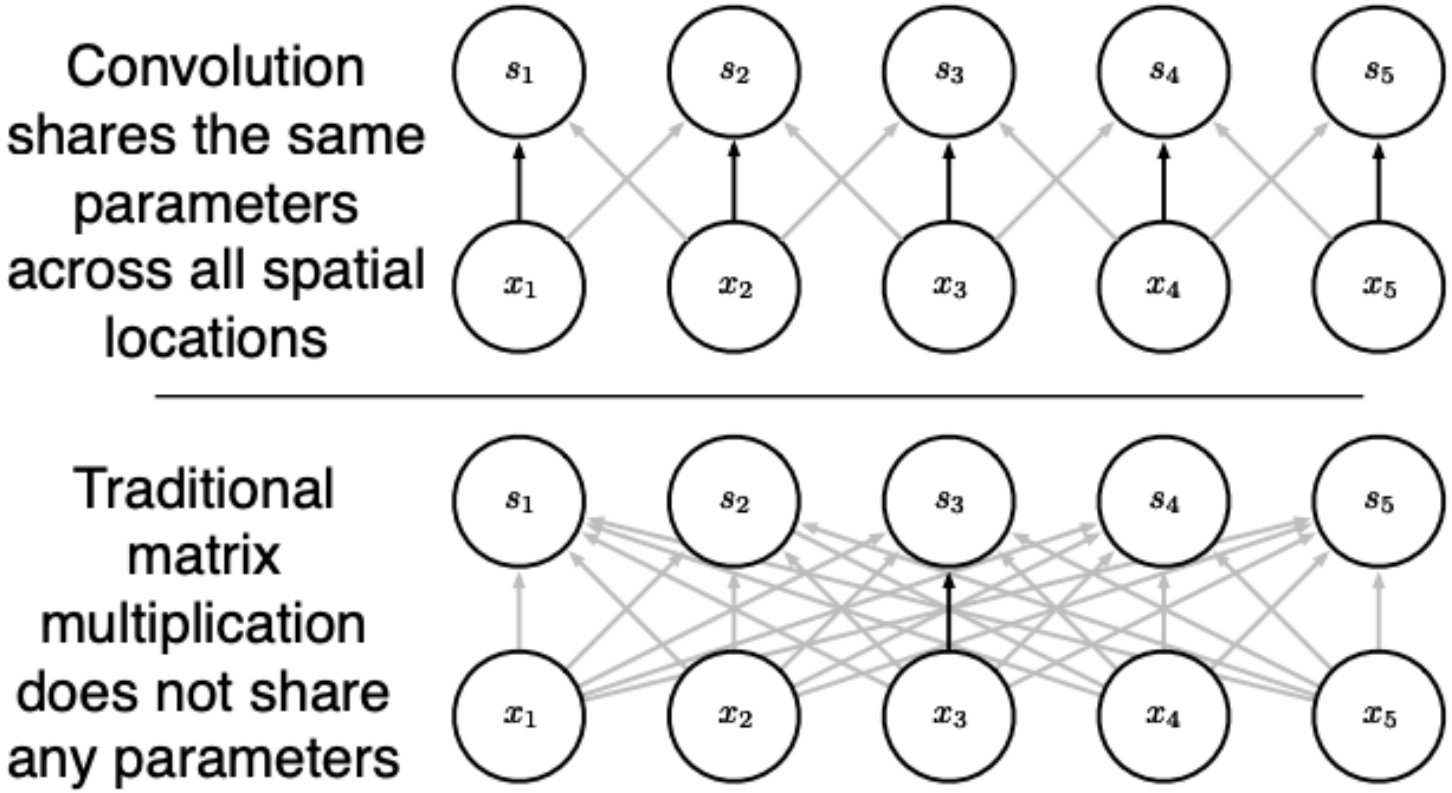
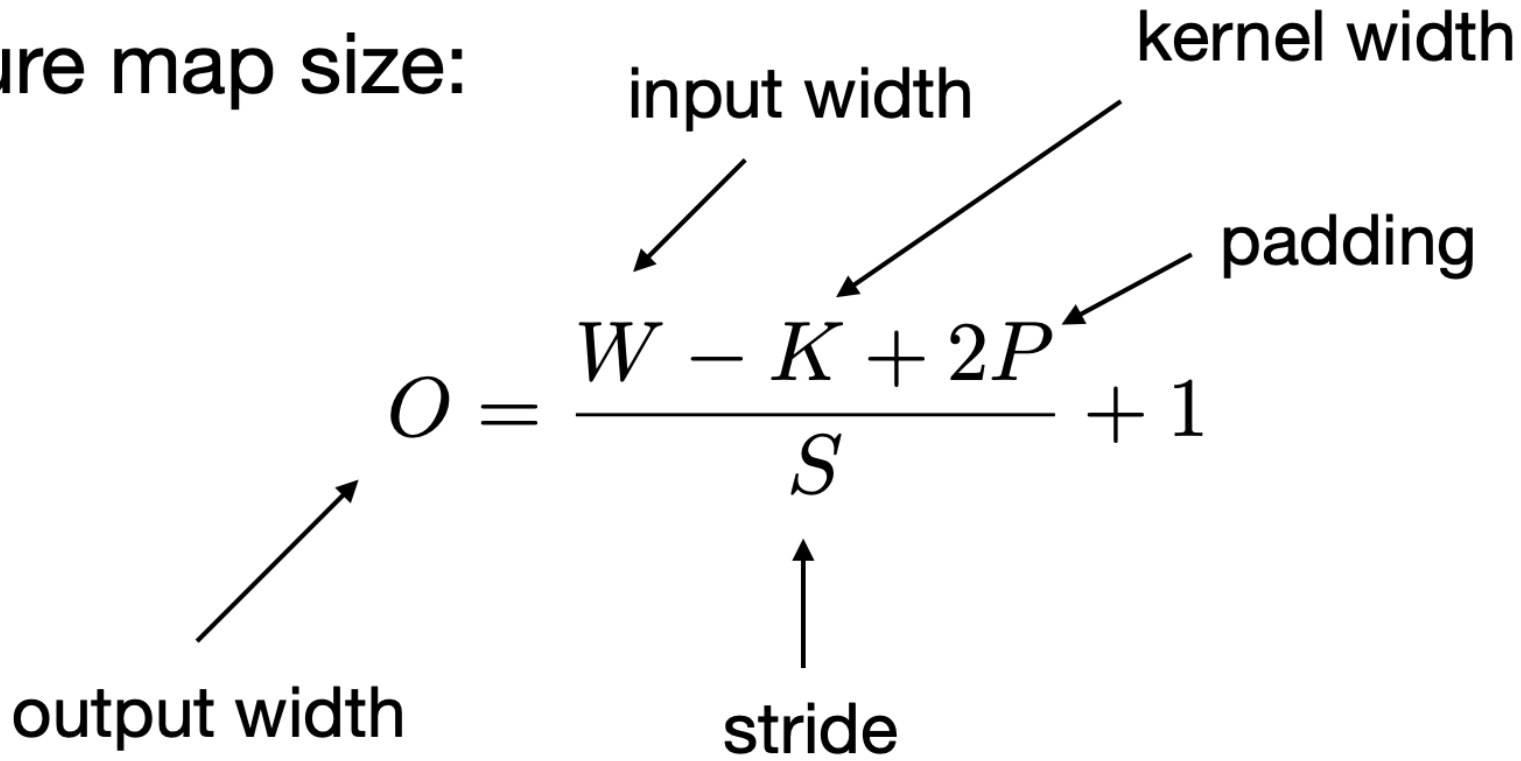


Figure 9.5

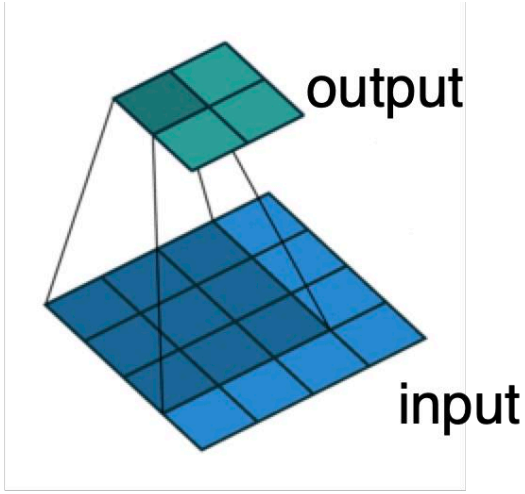
(Goodfellow 2016)

Impact of convolutions on size

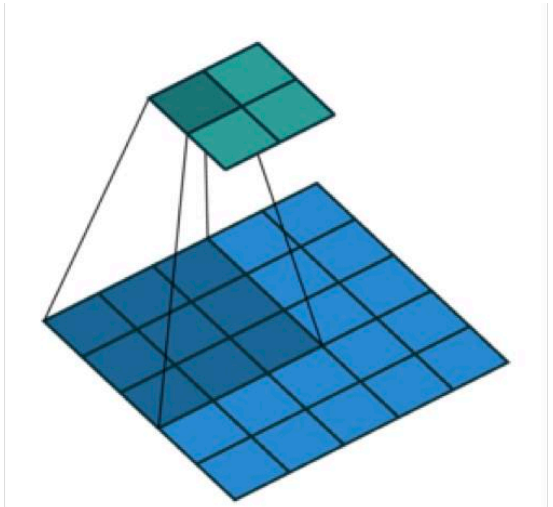
Feature map size:



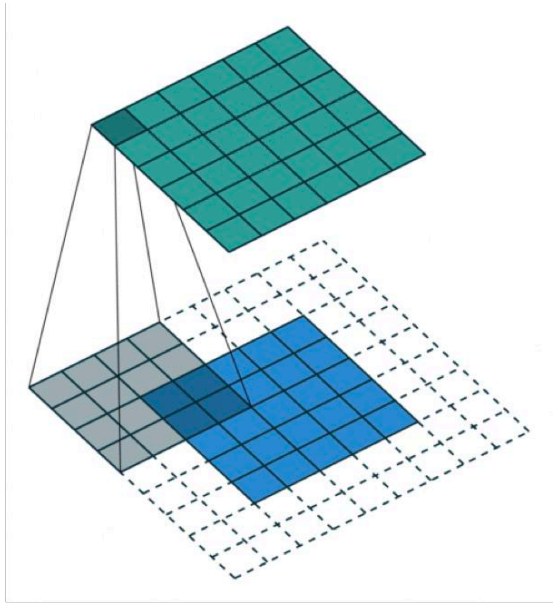
Padding



No padding, stride=1



No padding, stride=2



padding=2, stride=1

Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

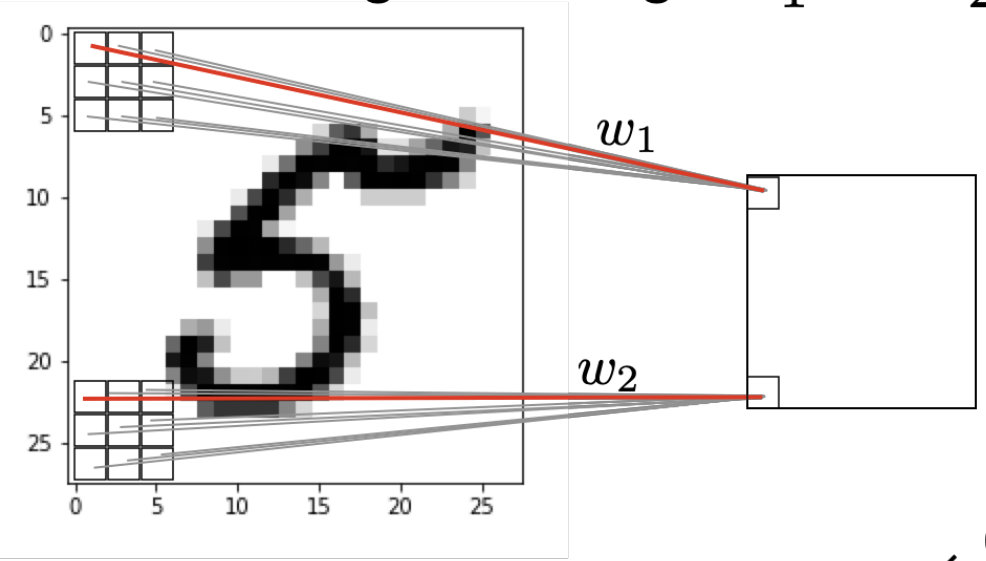
Backpropagation in CNNs

- Same concept as before: Multivariable chain rule, and now with an additional weight-sharing constraint

Backpropagation in CNNs

- Same concept as before: Multivariable chain rule, and now with an additional weight-sharing constraint

Due to weight sharing: $w_1 = w_2$

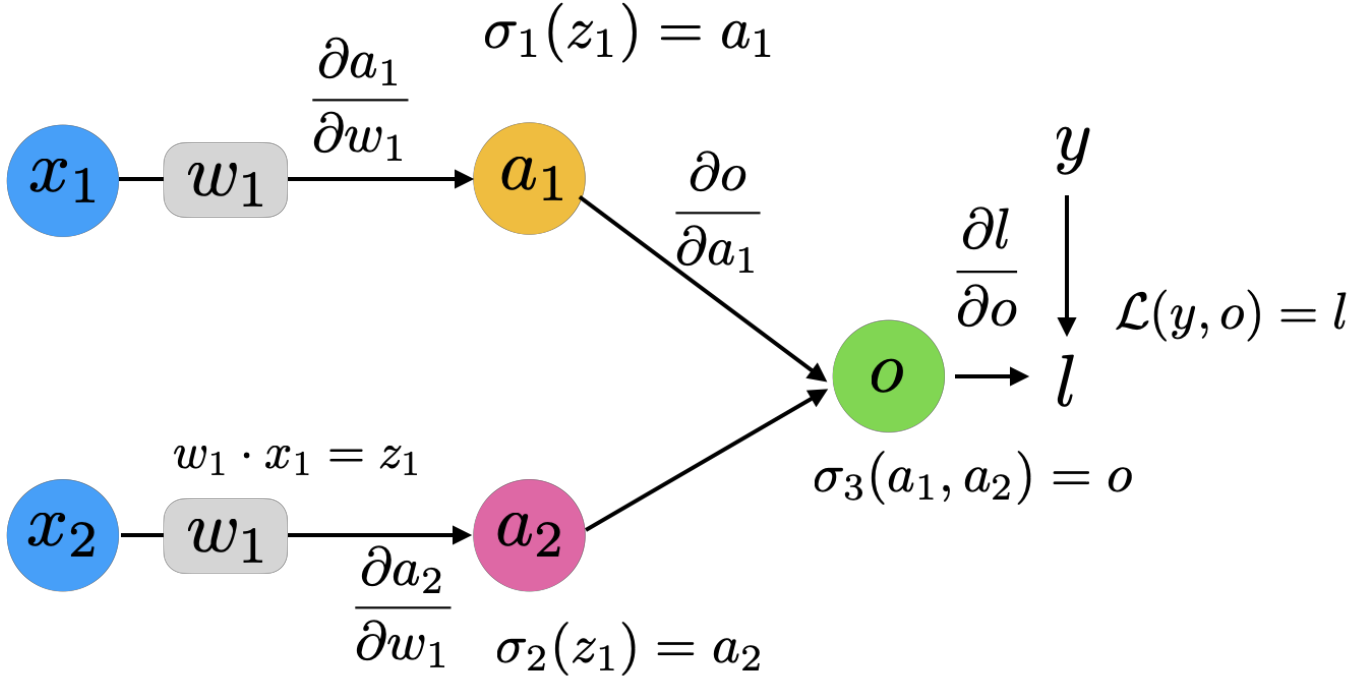


Optional averaging

weight update:

$$w_1 := w_2 := w_1 - \eta \cdot \frac{1}{2}\left(\frac{\partial \mathcal{L}}{\partial w_1} + \frac{\partial \mathcal{L}}{\partial w_2}\right)$$

Recall: Weight sharing in computation graphs



Upper path

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_1}$$ (multivariable chain rule)

Lower path

Questions?