STAT 992: Foundation Models for Biomedical Data

Ben Lengerich

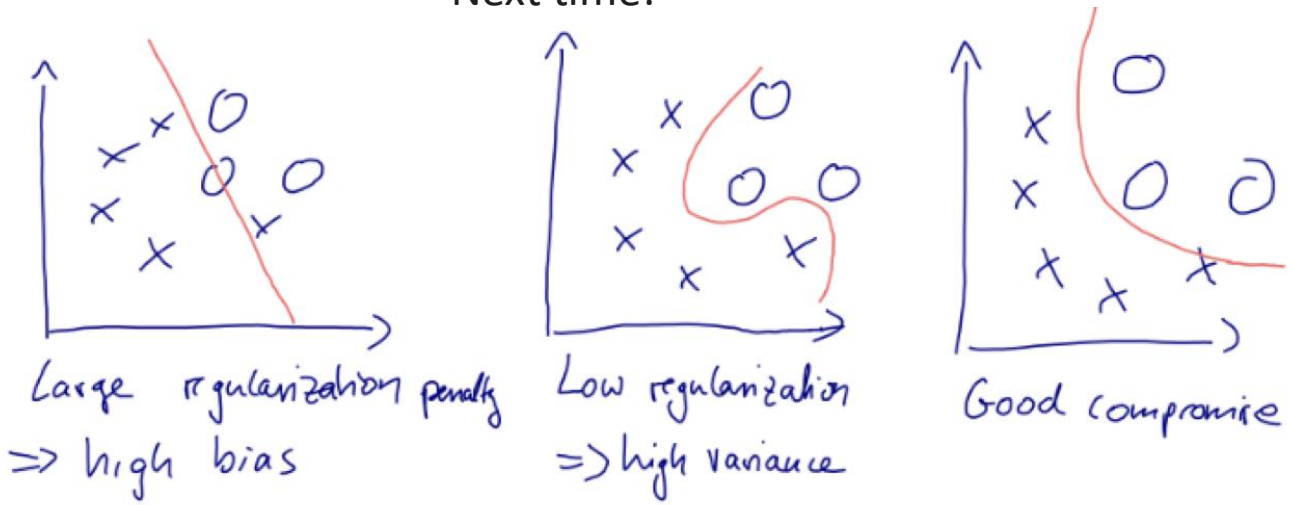
Lecture 03: Regularization, Optimization, and Initialization in Deep Learning

Jan 28, 2026

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:



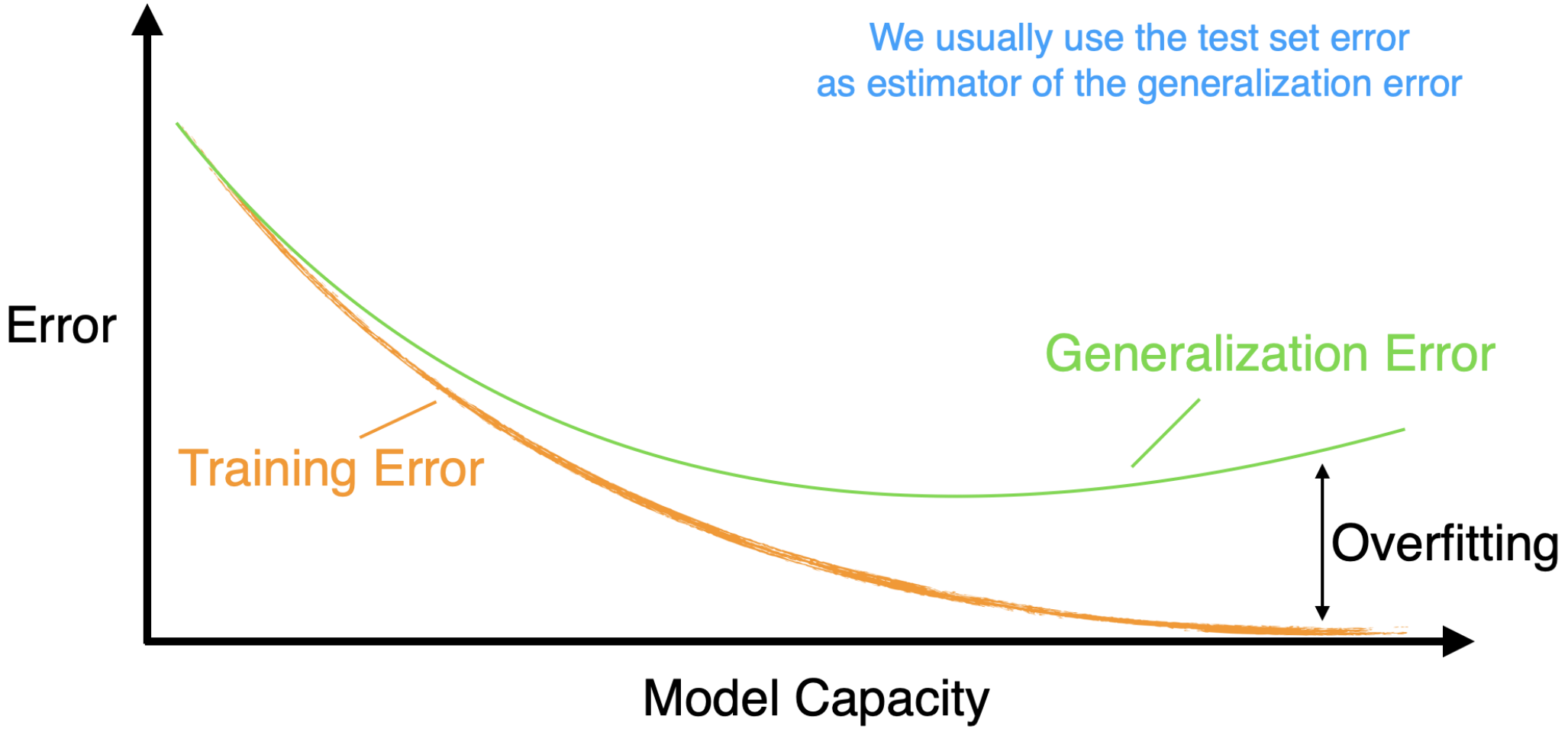
Regularization

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



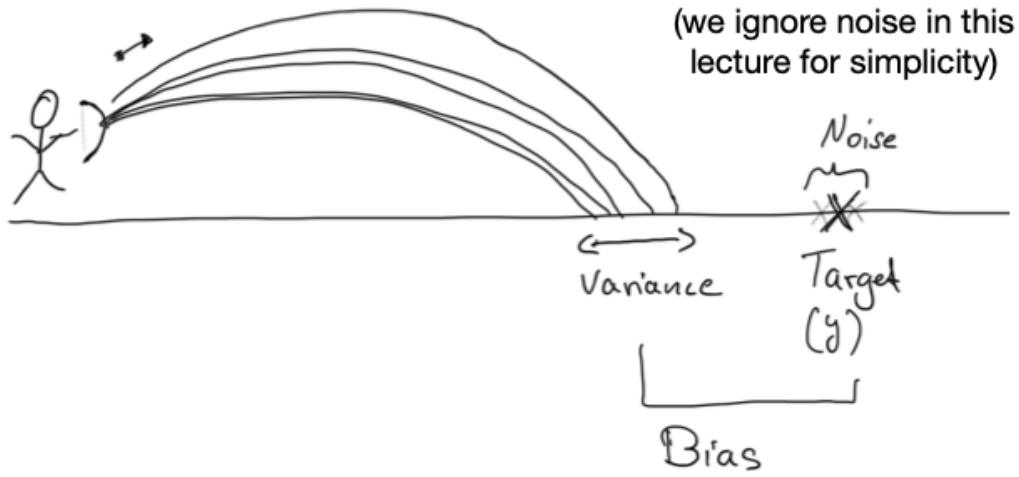
Bias-Variance Decomposition

General Definition:

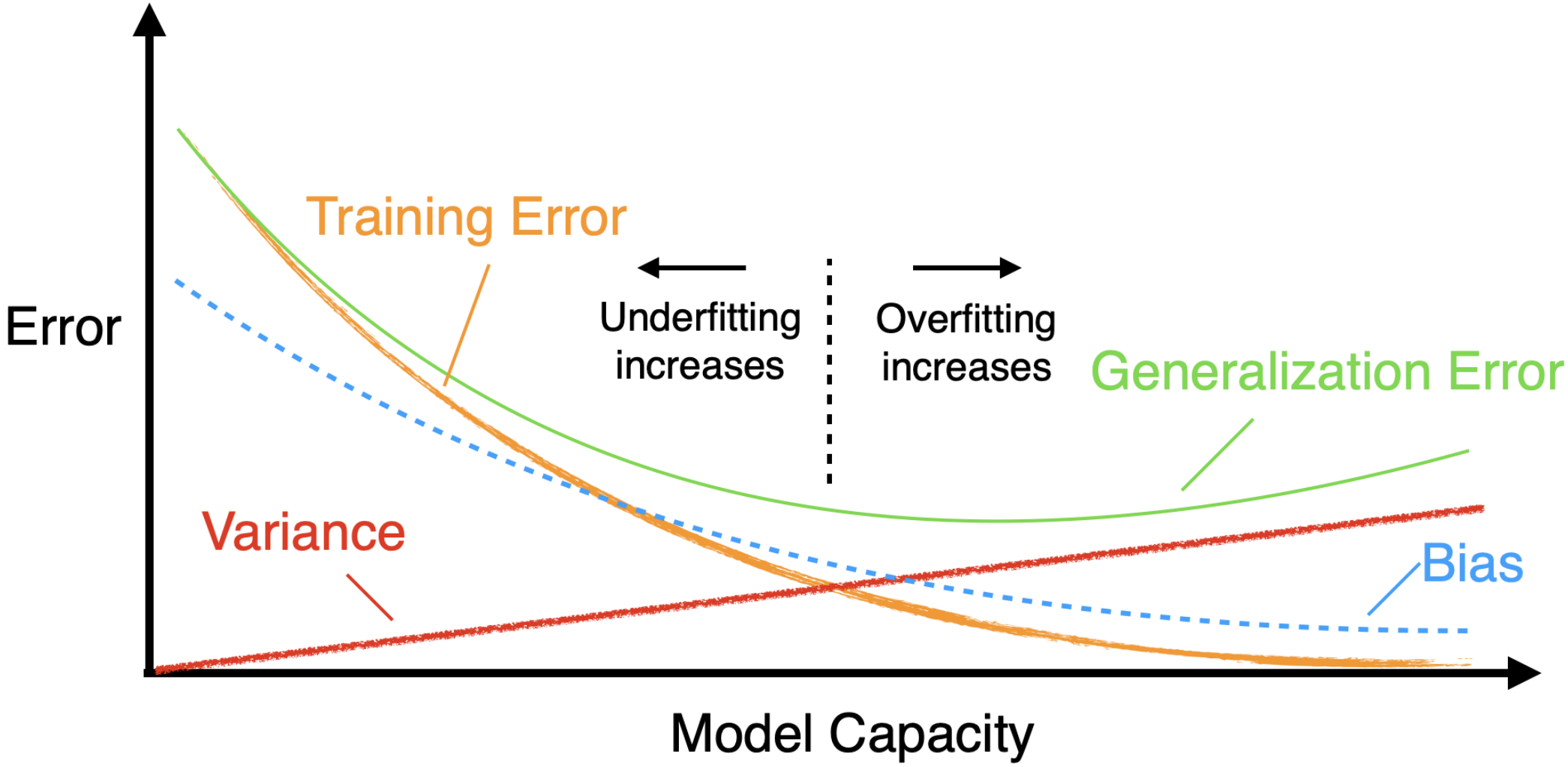
$$\text{Bias}_\theta[\hat{\theta}] = E[\hat{\theta}] - \theta$$

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

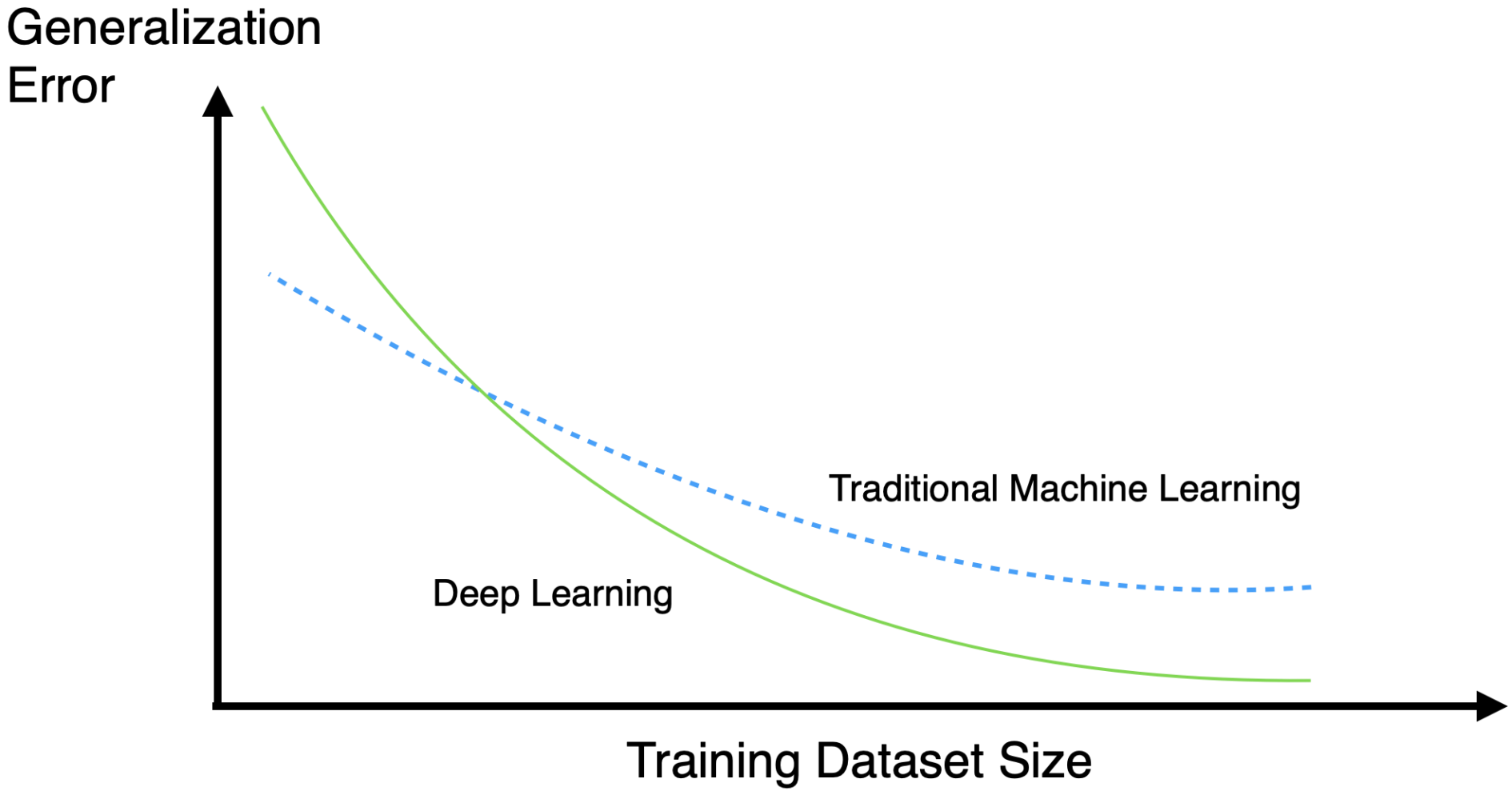
Intuition:



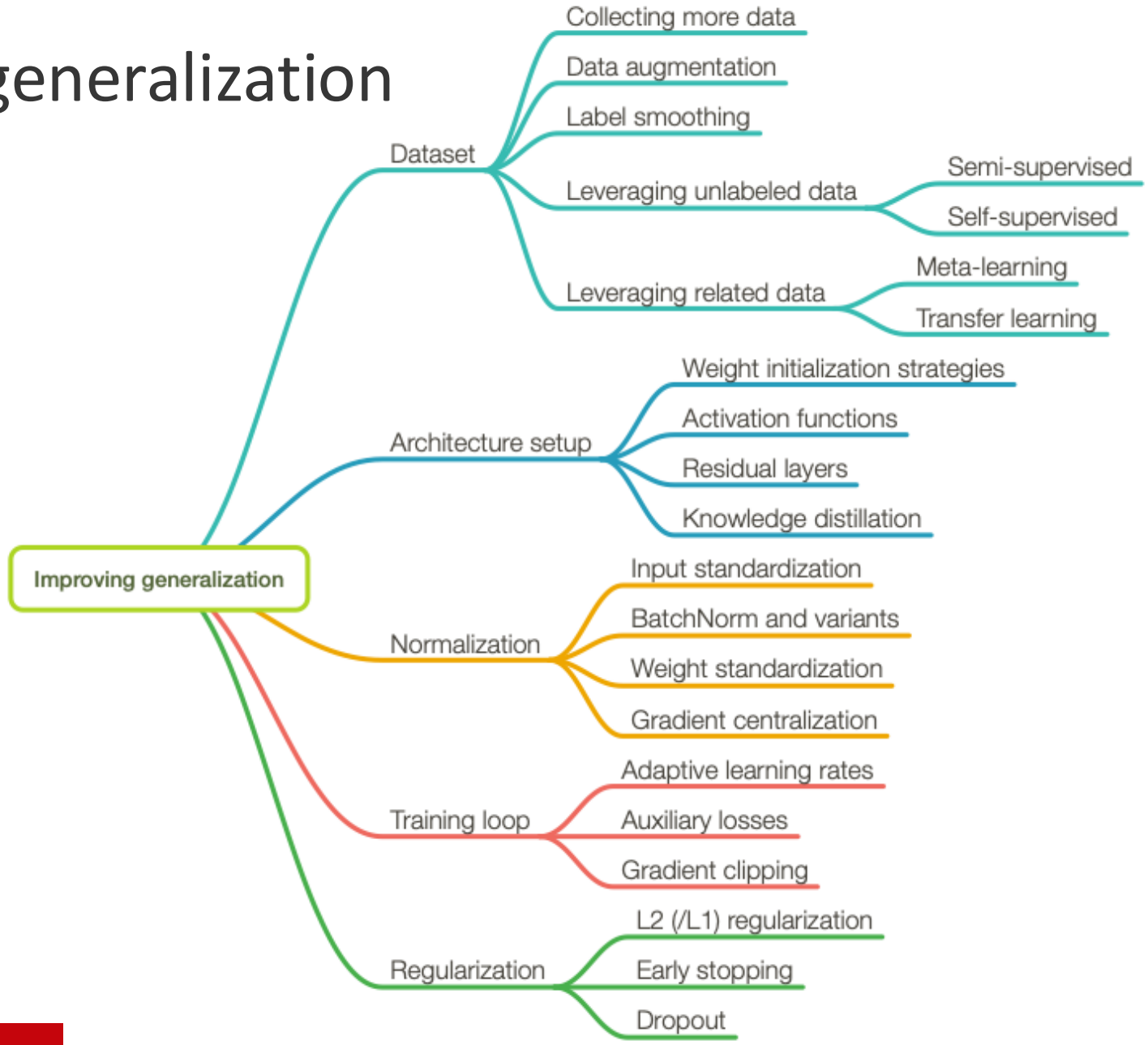
Bias-Variance & Overfitting-Underfitting



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

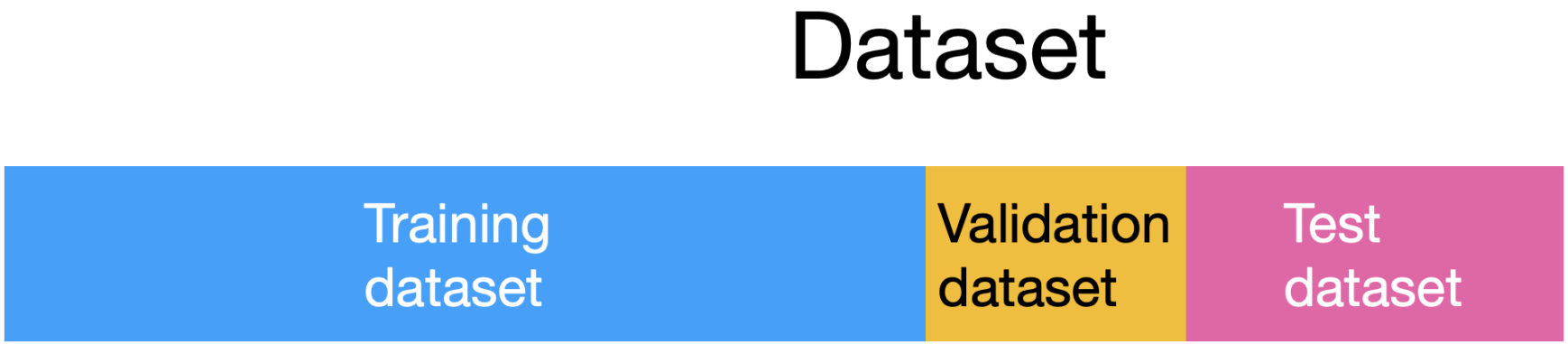
- **Key Idea:** If we know the label shouldn't depend on a transformation h(x), then we can generate new training data $h(x^i), y^i$
- But we must already know something that our outcome doesn't depend on
- Example: image classification
  - rotation, zooming, sepia filter, etc.

Reduce Network 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)

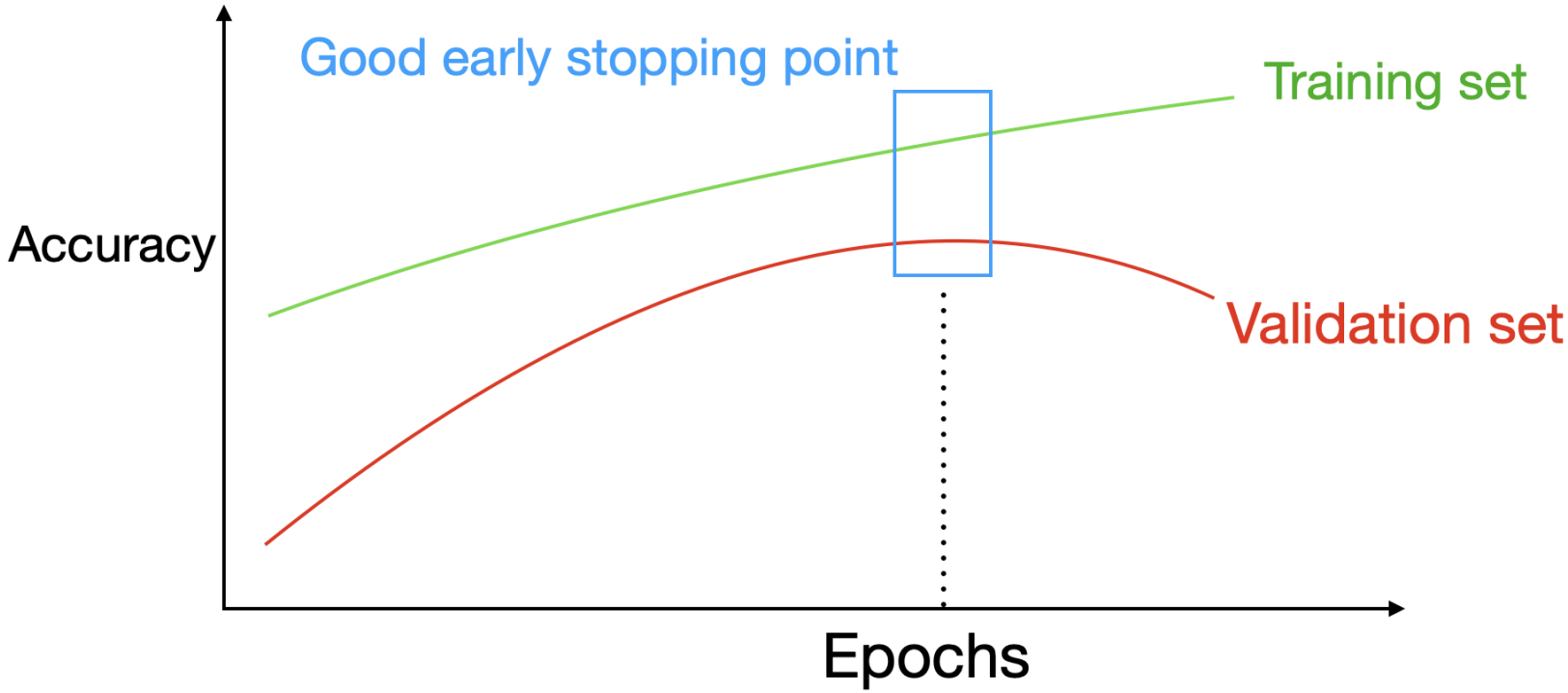
Early Stopping

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



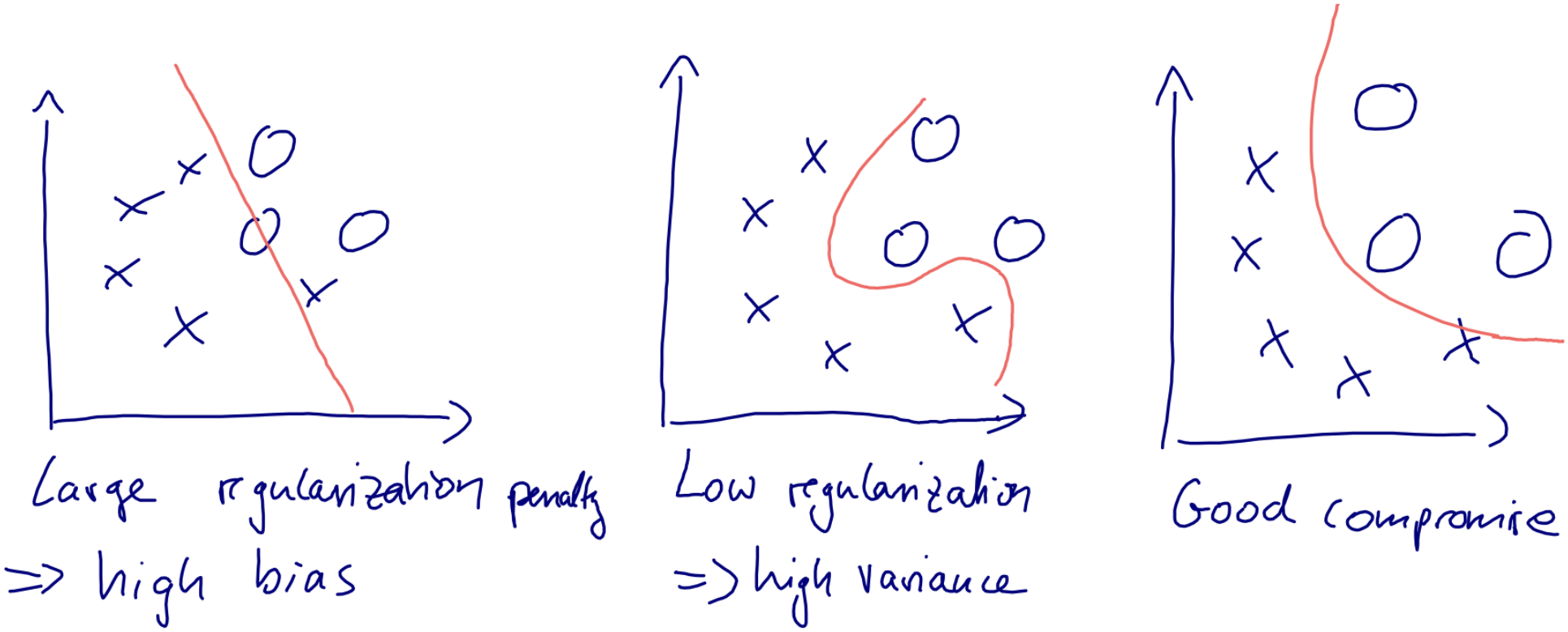
Early Stopping

- **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

Assume a nonlinear model



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:

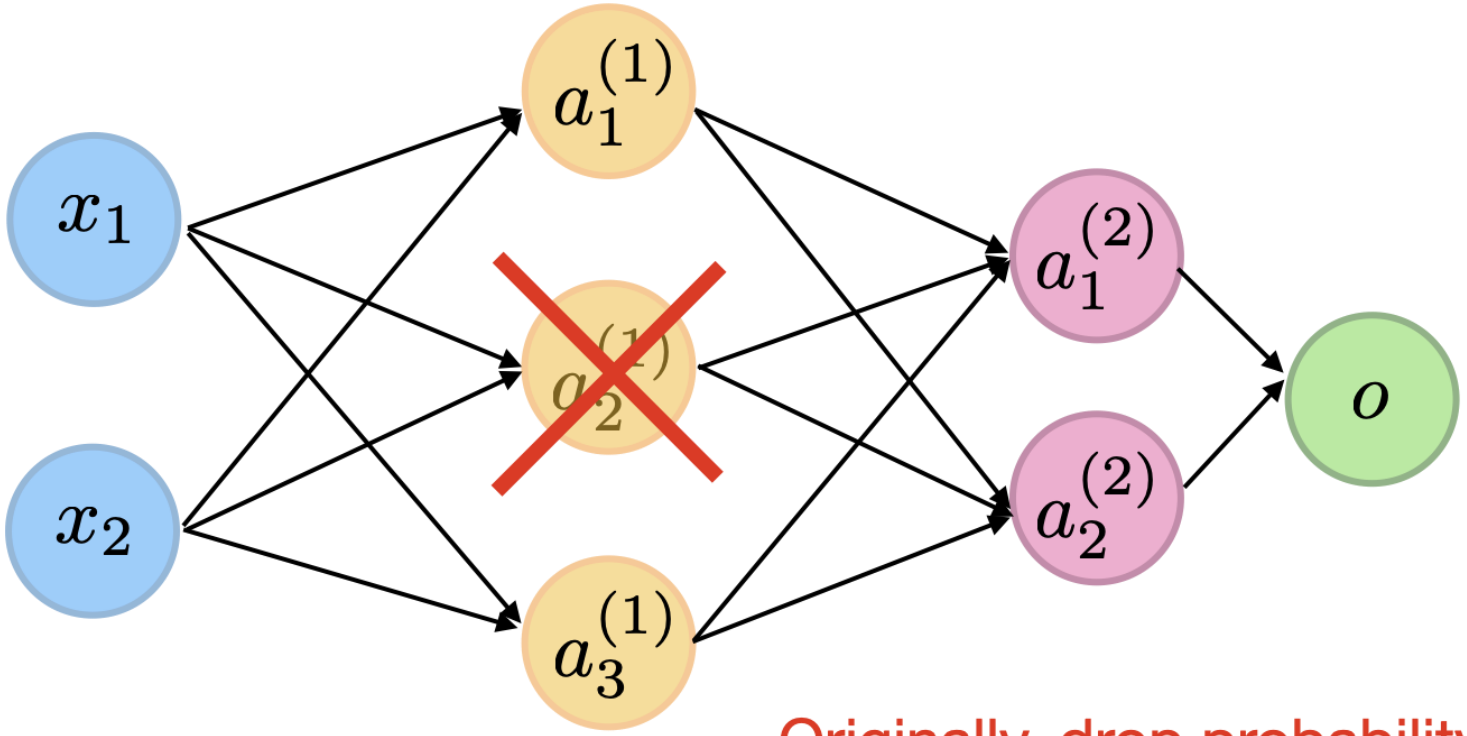
```
# regularize loss
L2 = 0.
for name, p in model.named_parameters():
    if 'weight' in name:
        L2 = L2 + (p**2).sum()

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

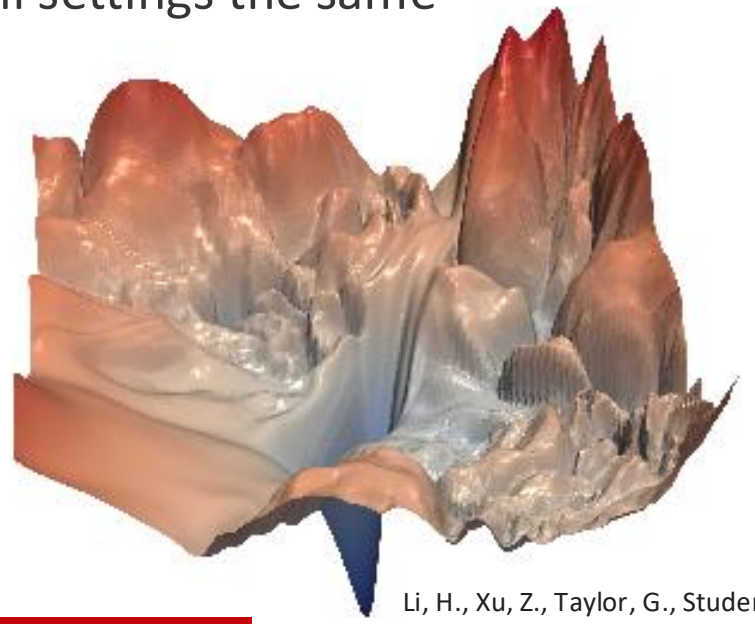
Dropout in PvTorch

```
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
```

Improvements to optimization

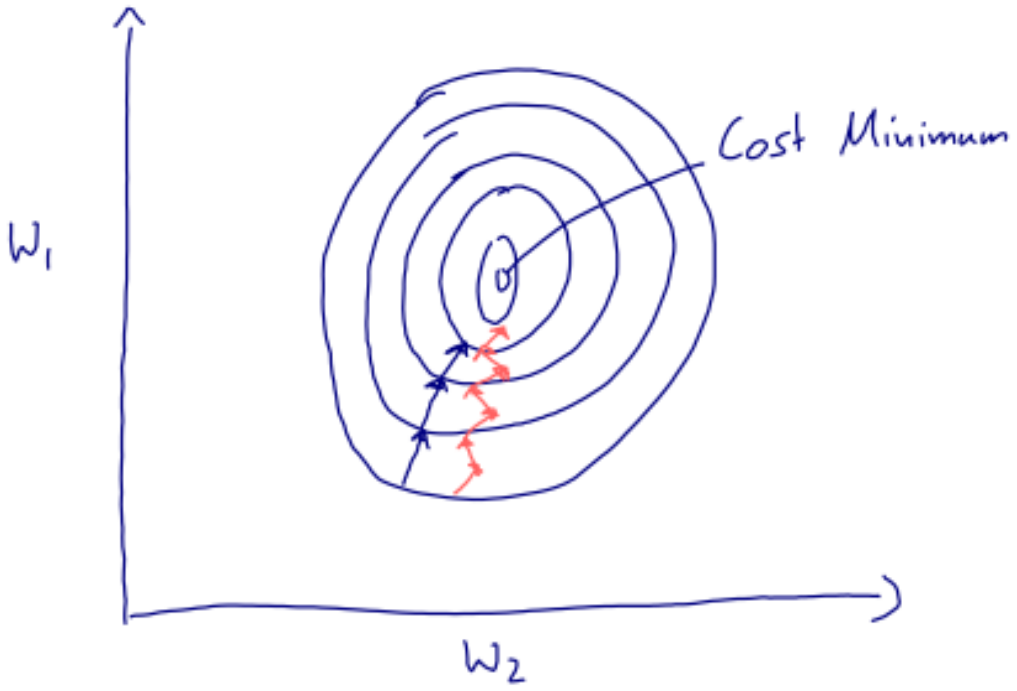
Note that our Loss is Not Convex Anymore

- 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



- 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**

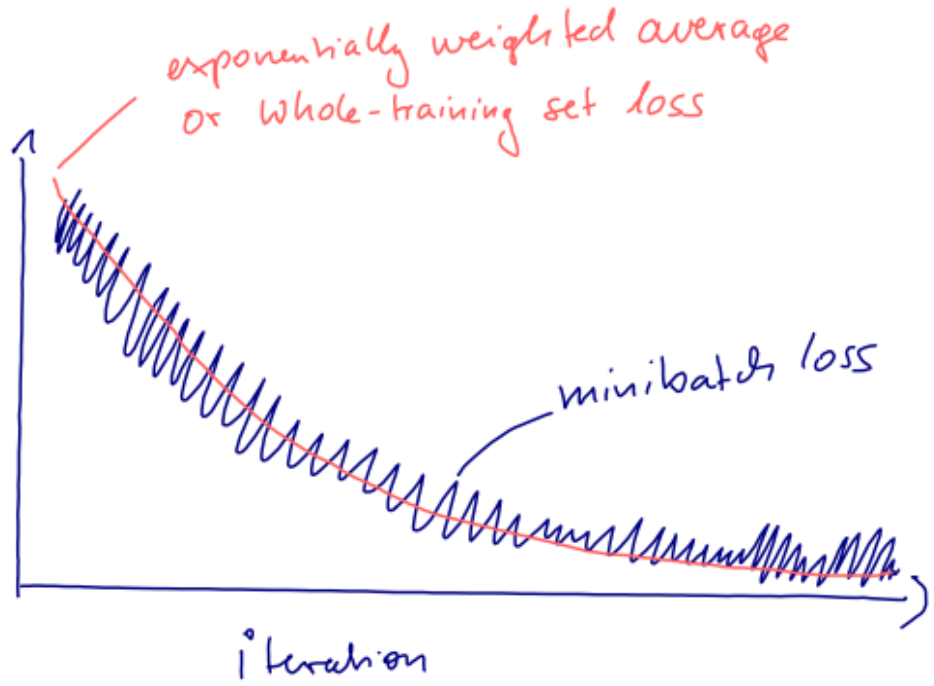
A **noisy** gradient can be:

- **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)



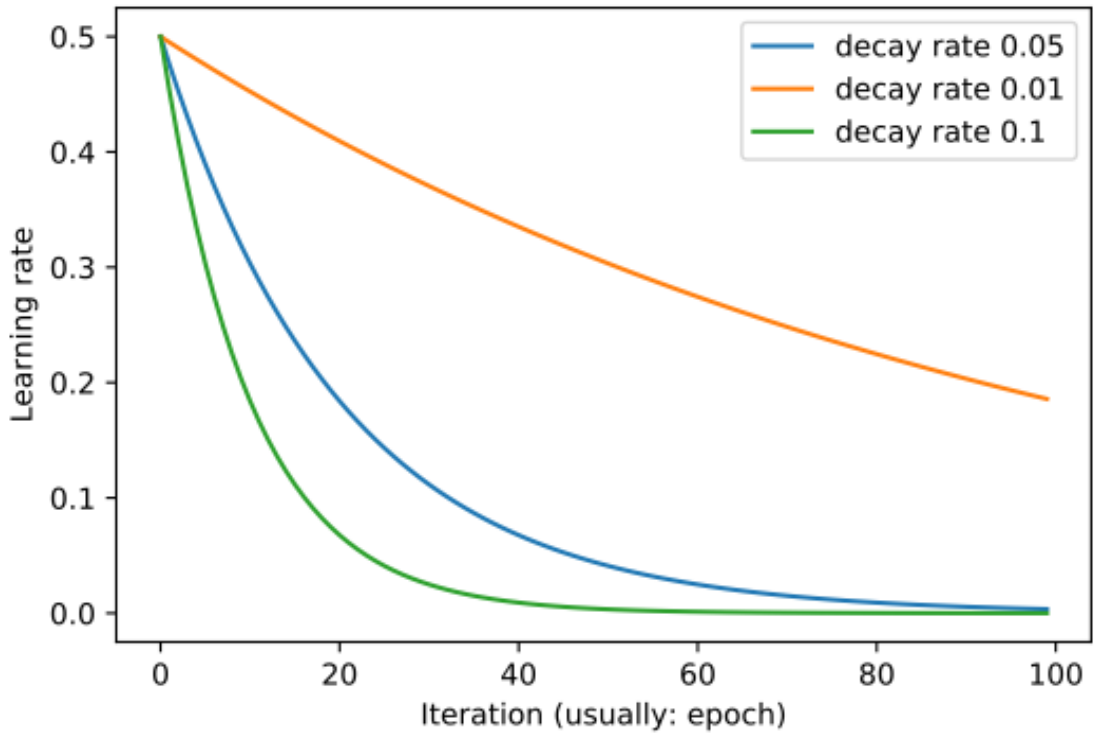
Learning Rate Decay

Most common variants for lr decay:

1. Exponential Decay:

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

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:

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

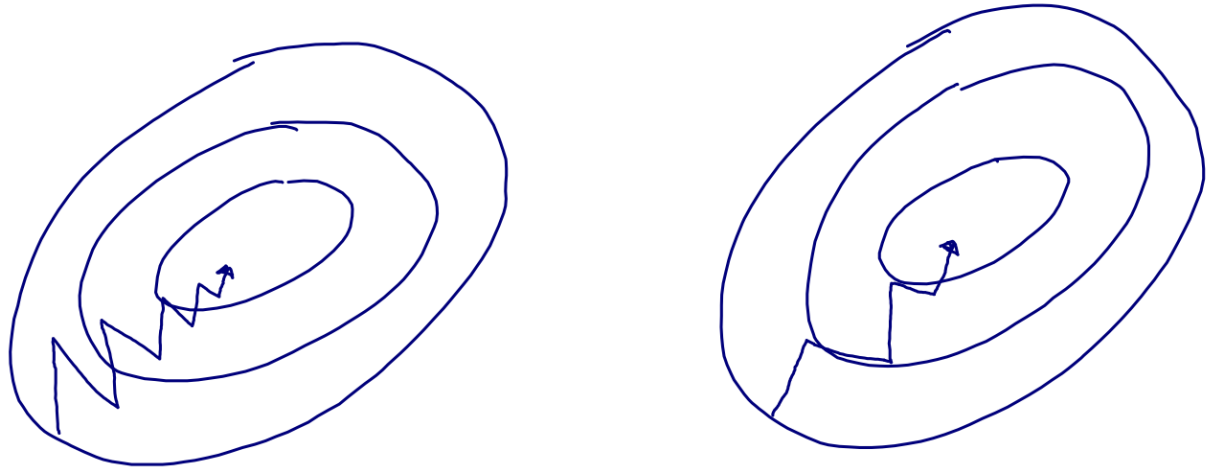
Training with “Momentum”

- 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)



Without momentum

With momentum

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:

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

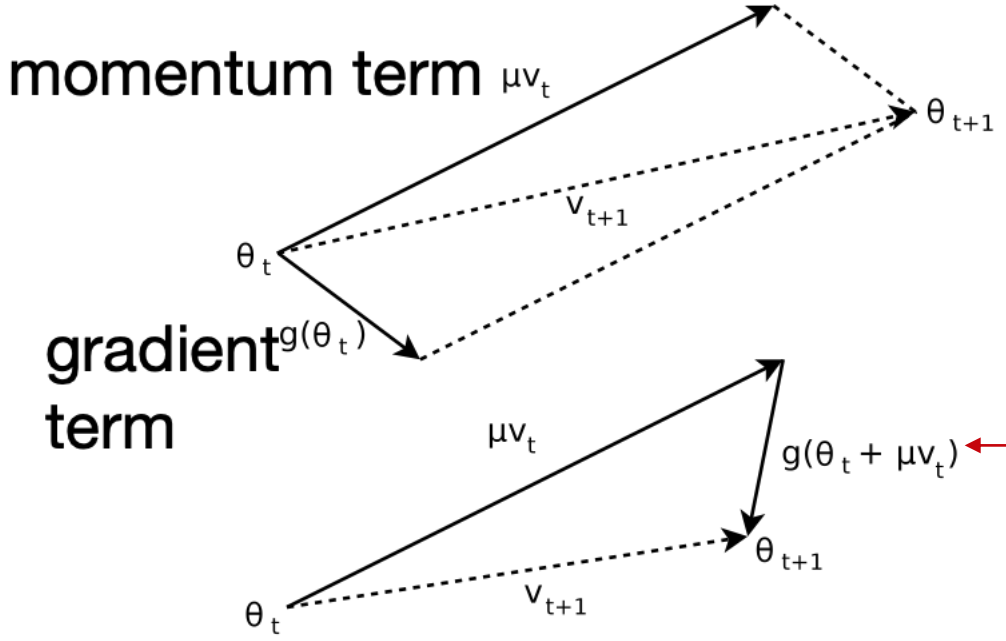
Nesterov:

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

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:

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

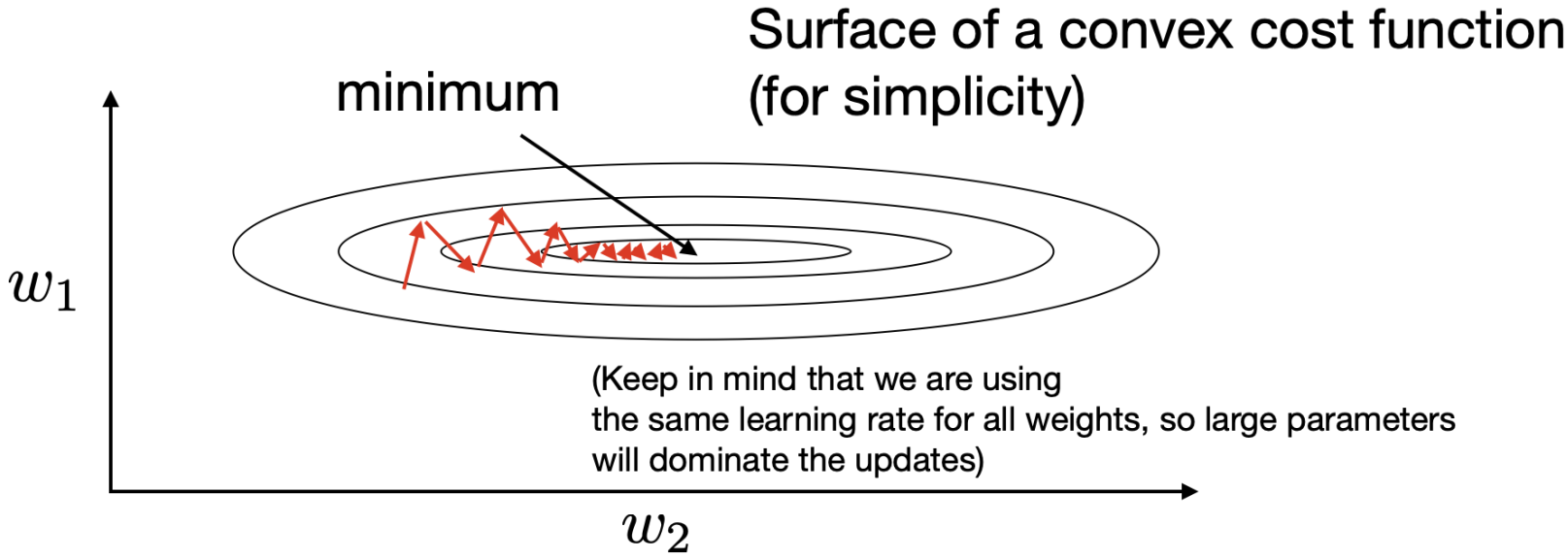
ADAM update:

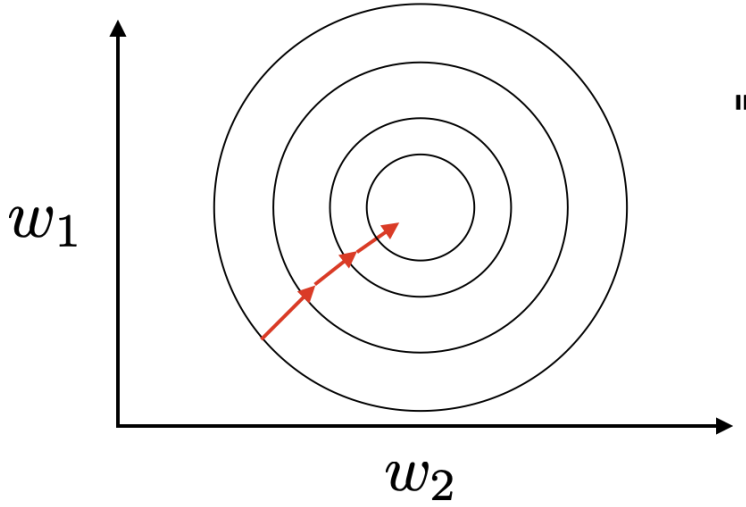
$$w_{i,j} := w_{i,j} - \eta \frac{m_t}{\sqrt{r} + \epsilon}$$

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

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")

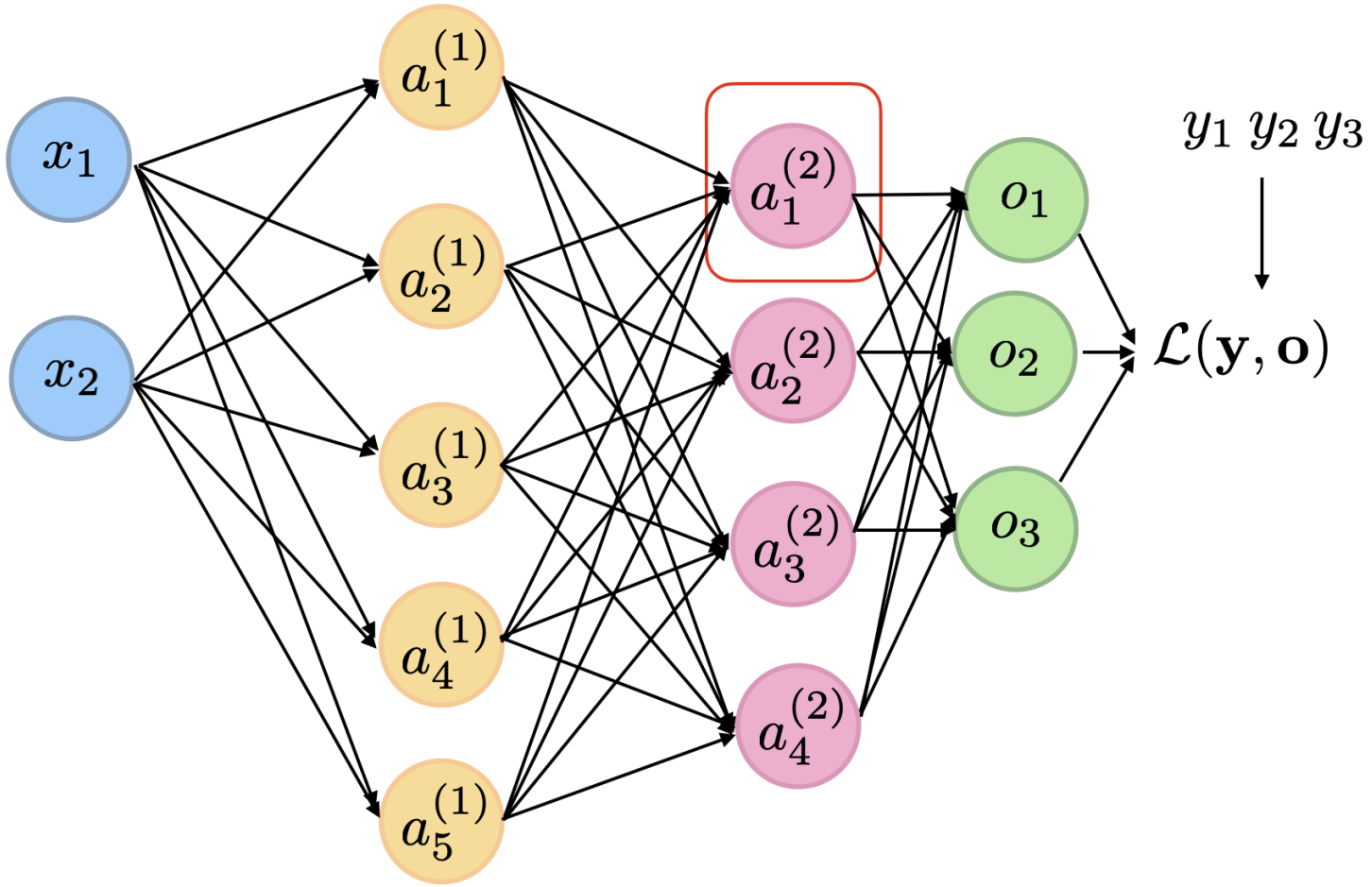
Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448-456).

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")

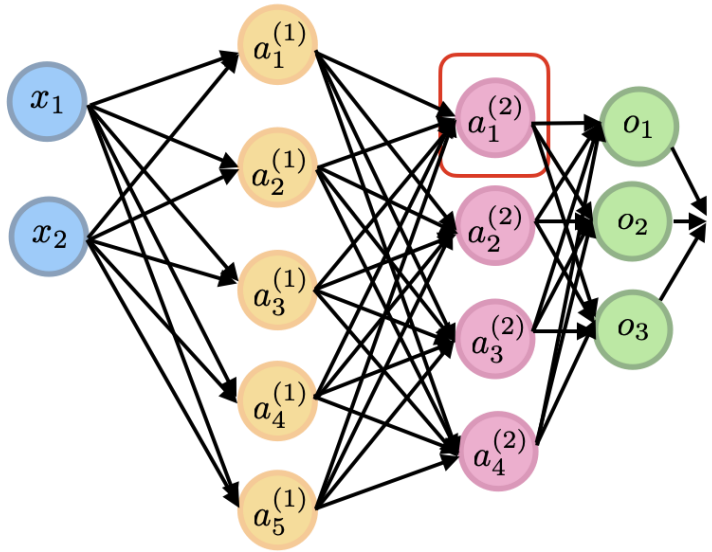
Suppose, we have net input $z_1^{(2)}$
associated with an activation in the 2nd hidden layer



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

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

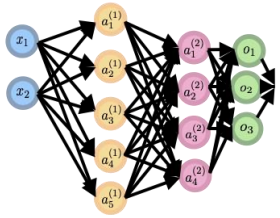
$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

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

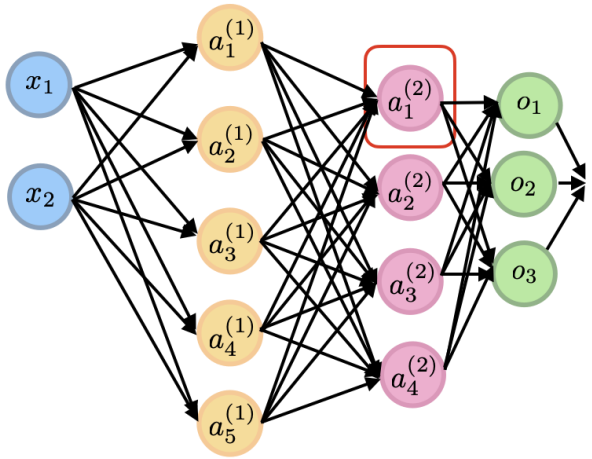
$z^{(2)}_1 \rightarrow z'^{(2)}_1 \rightarrow a'^{(2)}_1 \rightarrow a^{(2)}_1$ second hidden layer

...

BatchNorm Steps 1+2 Together

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

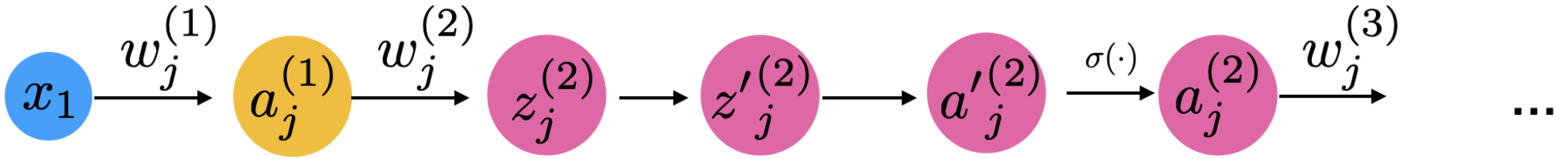
This parameter makes the bias units redundant



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

BatchNorm and Backprop

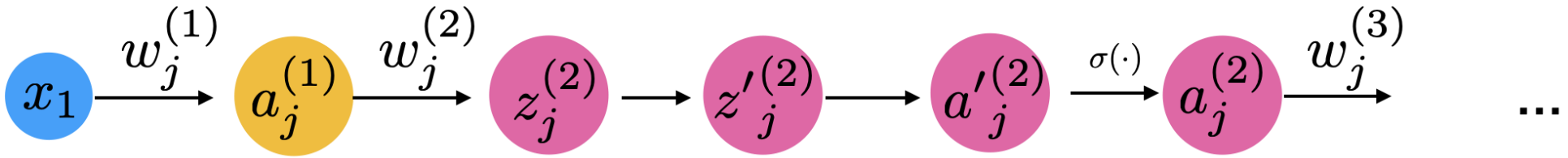
$$z'^{(2)}_j = \frac{z^{(2)}_j - \mu_j}{\sigma_j} \qquad a'^{(2)}_j = \gamma_j \cdot z'^{(2)}_j + \beta_j$$



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

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

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

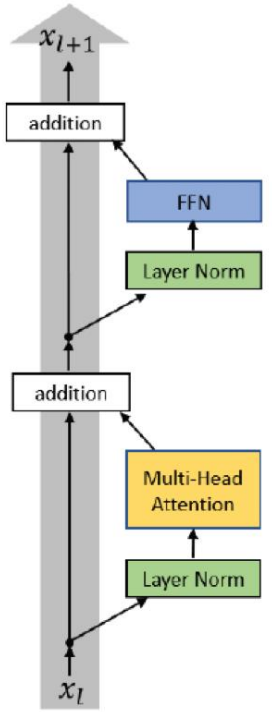
running_mean = momentum * running_mean + (1 - momentum) * sample_mean

(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?

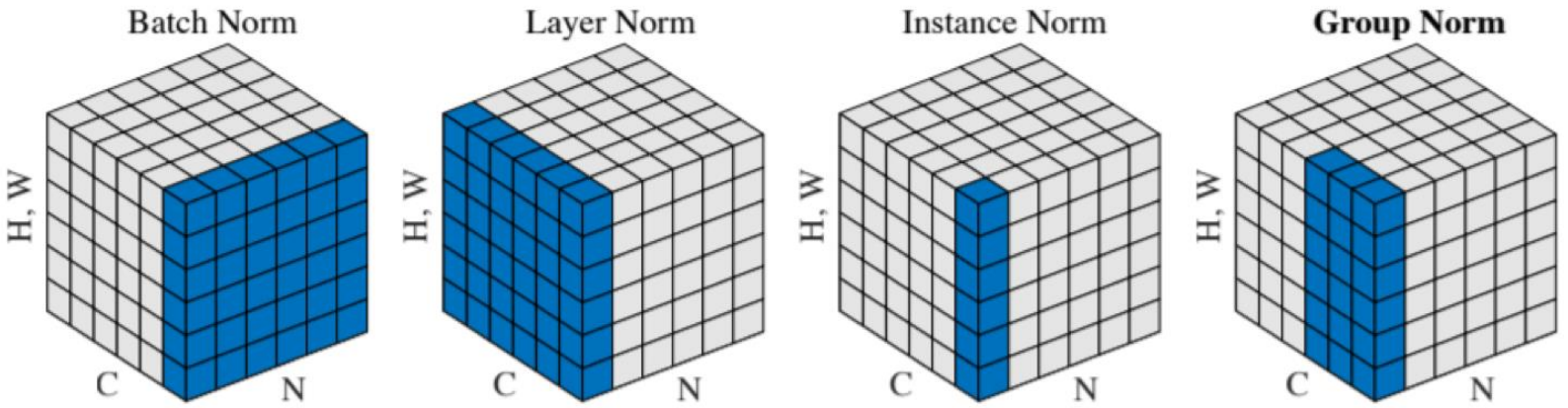


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

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.

Questions?