DSCI 565: MULTILAYER PERCEPTRONS

Ke-Thia Yao

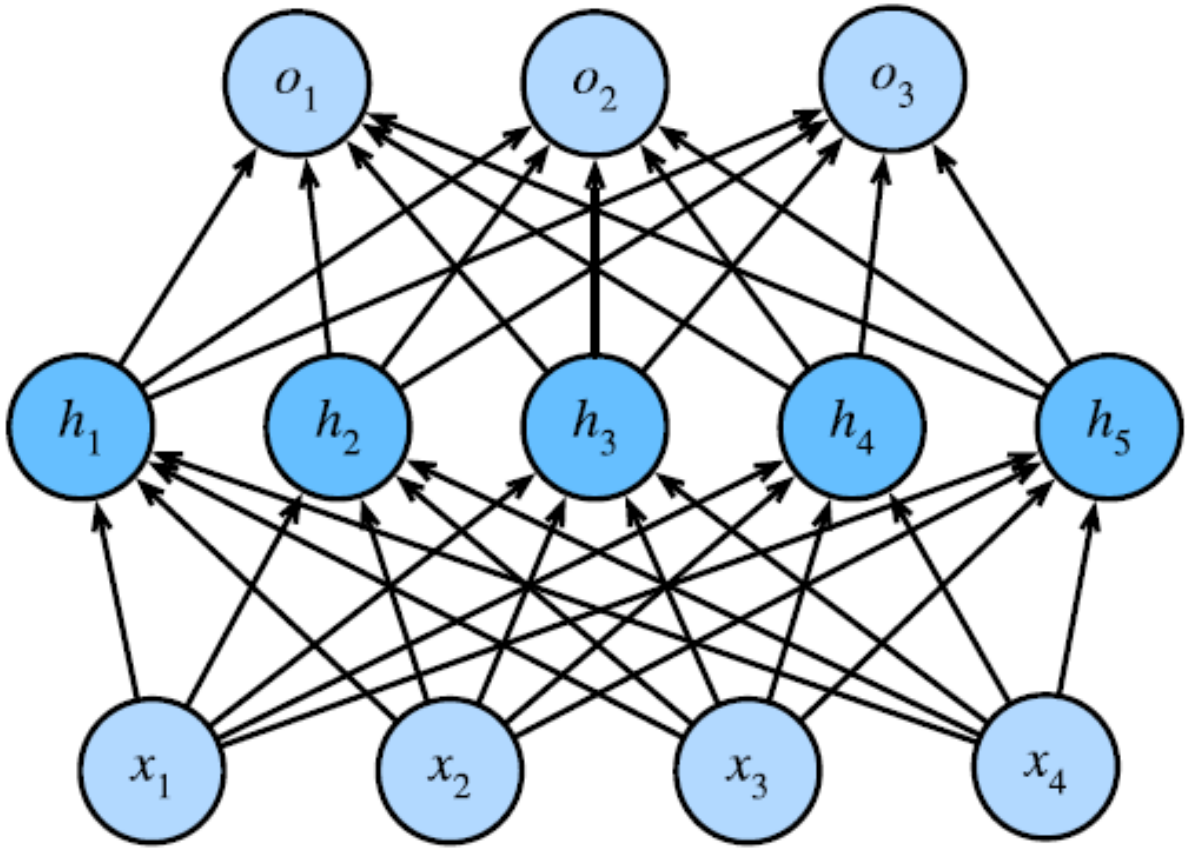
Lecture 6: 2025-09-25

Limitations of Linear Models

- While the linearity assumption is applicable for some problems, it is not appropriate for many problems
- For example, in loan risk assessment the higher the income of an individual should imply lower risks
  - However, does risk change linearly when income changes from $0 to $50,000 compared to when income changes from $1 million to $1.05 million?
  - Perhaps weaker assumption of monotonicity is more appropriate
  - Potentially this could be handled using postprocessing
- For health prediction, reducing fever temperature should imply better health outcome
  - But, only up to a certain point. Low body temperature is also bad
  - Potentially this could be handled using data preprocessing
- We want to do this using automatically without feature engineering

Hidden Layers

MLP: Multilayer Perceptions



From Linear to Nonlinear

- Simply adding hidden layers is not enough
- Because affine transformation of an affine transformation is still just an affine transformation
- Affine transformation is a linear transformation plus a constant

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)},$$

$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}$$

Nonlinear Activation Function

- Need to apply nonlinear activation function σ to each hidden layer
- For example, we could use ReLU (rectified Linear Unit) or sigmoid

$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}),$$

$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

- Here we define σ to apply to each row of $\mathbf{XW}^{(1)}$

⇒ Nonlinear Model

Universal Approximator

- A MLP with a single hidden layer (with sufficiently large number of hidden units) can approximate any function
- However, we typically user deeper networks (versus wide networks)
- Deep networks can represent functions more compactly

Rectified Linear Unit (ReLU)

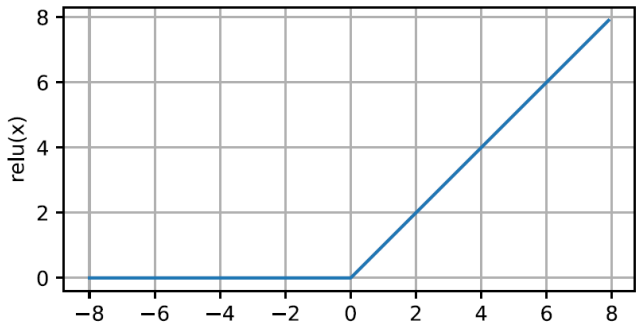
- Given x ReLU returns max of x and 0

$$\mathrm{ReLU}(x) = \max(x, 0)$$

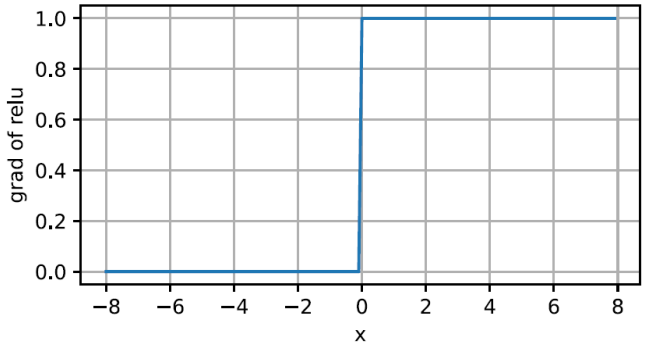
- Retains only positive elements, and discards negative elements
- Derivative of ReLU is a step function.
  - Well behaved, no vanishing gradients

$$\frac{d\mathrm{ReLU}(x)}{dx} = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```




```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



Sigmoid Function

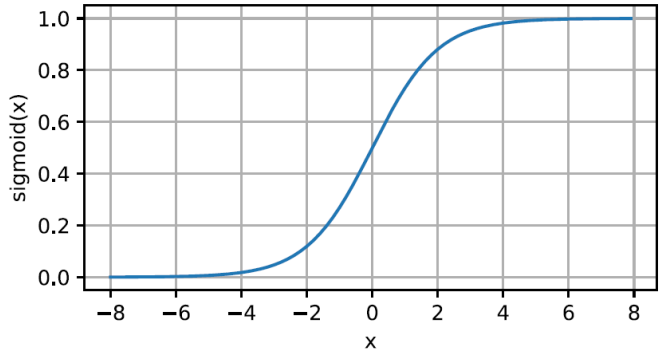
- Sigmoid function

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

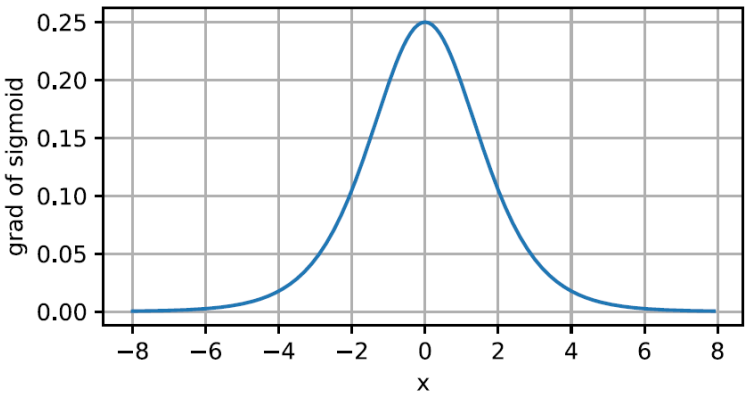
- Derivative

$$\frac{d\sigma(x)}{dx} = \sigma(x)\left(1 - \sigma(x)\right)$$

```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```




```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



Notebooks

- chapter_multilayer-perceptrons/mlp.ipynb
- chapter_multilayer-perceptrons/mlp-implementation.ipynb

10 Automatic Differentiation with Tensors

Forward Propagation

- Forward Propagation (or forward pass) of a neural network is the calculation from the input layer to the output layer

- The intermediate variables during the calculation are stored

Forward Propagation with One Hidden Layer

- Forward pass of a one-hidden layer network

$$\boldsymbol{z} = \boldsymbol{W}^{(1)}\boldsymbol{x}$$
$$\boldsymbol{h} = \phi(\boldsymbol{z})$$
$$\boldsymbol{o} = \boldsymbol{W}^{(2)}\boldsymbol{h}$$
$$L = l(\boldsymbol{o}, y)$$
$$s = \frac{\lambda}{2}\left(\left\|\boldsymbol{W}^{(1)}\right\|_F^2 + \left\|\boldsymbol{W}^{(2)}\right\|_F^2\right)$$
$$J = L + s$$

weight decay

- Where

$$\boldsymbol{x} \in \mathbb{R}^d$$
$$\boldsymbol{W}^{(1)} \in \mathbb{R}^{h \times d}$$
$$\boldsymbol{z}, \boldsymbol{h} \in \mathbb{R}^h$$
$$\boldsymbol{W}^{(2)} \in \mathbb{R}^{q \times h}$$
$$\boldsymbol{o} \in \mathbb{R}^q \quad , y \in \mathbb{R}^q$$
$$L, s, J \in \mathbb{R}$$

- And

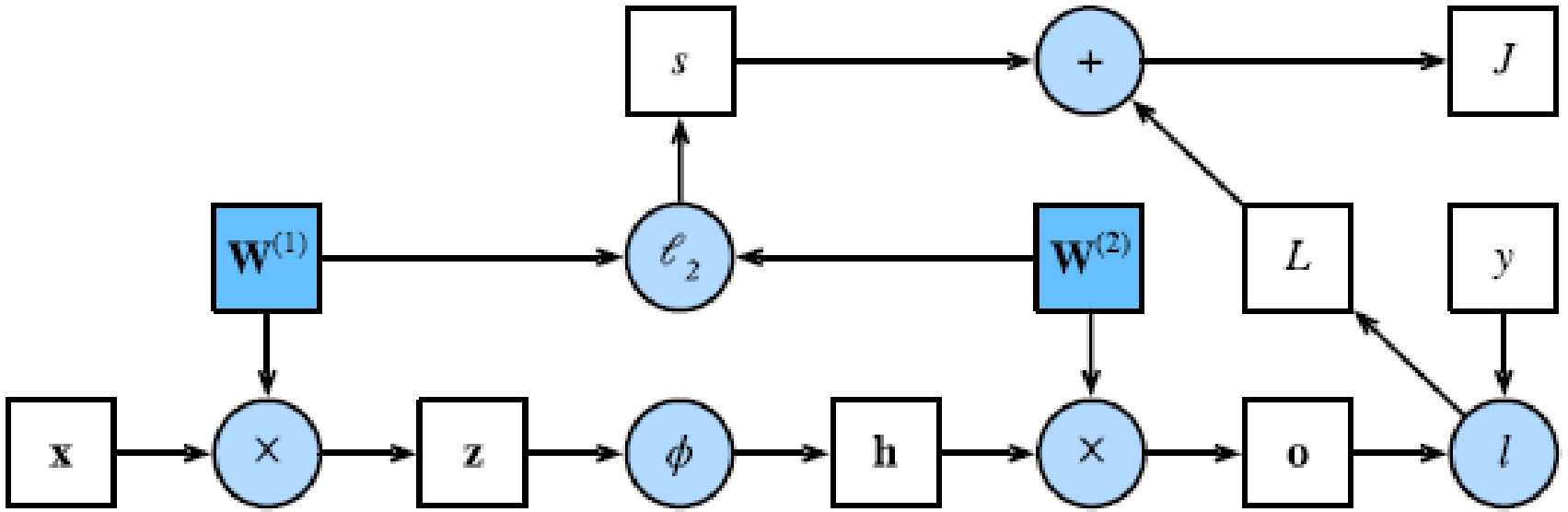
ϕ is activation function

l is lost function

Computational Graph

Captures the dependencies of operators and variables

Arrows point in the forward direction



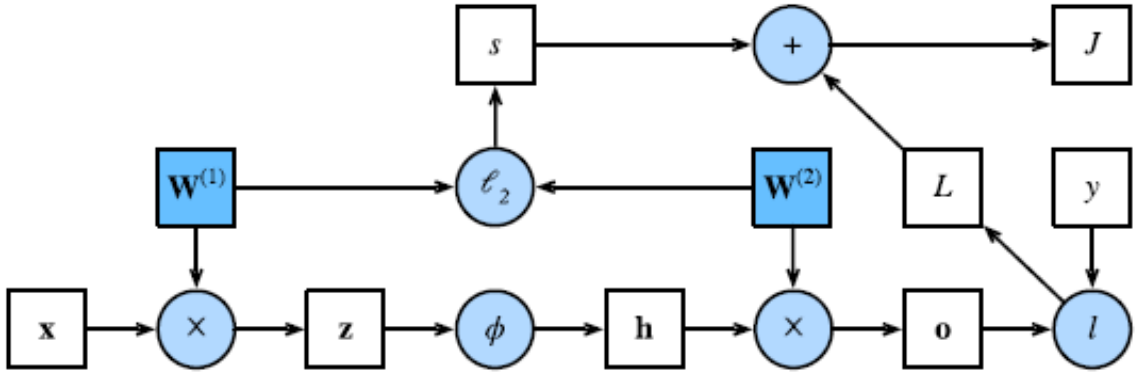
Backpropagation

- Backpropagation refers to the method of calculating the gradient of neural network parameters
- Traverse the computation graph backward from output to the input layer, according to the chain rule for derivatives
- Intermediate partial derivatives are stored
- Chain rule example, $Z = g(Y)$ and $Y = f(X)$

$$\frac{\partial Z}{\partial X} = \text{prod}\left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X}\right)$$

  - Where X, Y, Z are arbitrary tensors, and prod is multiplication operator

Backpropagation



$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1$$

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q$$

$$\frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}} = \mathbf{h}$$

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}}\mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$$

$q \times 1 \quad 1 \times h \qquad q \times h$

$$\frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$$

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z})$$

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}}\mathbf{x}^\top + \lambda \mathbf{W}^{(1)}$$

$h \times 1 \quad 1 \times d \qquad h \times d$

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)}$$

Training Neural Networks

- The training is performed in minibatches
- All the input, intermediate and output variables shown previously has an extra dimension corresponding to the minibatch
- During the forward pass, the current model parameters ($\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}$) are used compute the intermediate and output variables
- During the backward pass, these intermediate variables are used to compute the gradient
- Training is more memory intensive, because of the need to store these variables
- Training process alternates between forward and backward passes

Numerical Stability and Initialization

- Early deep networks suffered from numerical instability
- Sometimes the gradients would vanish leading to very slow learning
- Sometimes the gradients would explode causing the weights to diverge
- This is especially problematic for Recursive Neural Networks (RNN)
- Glorot and Bengio (2010) paper pointed to the causes
  - Use of sigmoid activation function, and
  - Using normal distribution for initialization of the weights

< activation function
initialization of the weights

Vanishing and Exploding Gradients

- Suppose we have a deep network of L layers with layer l defined by transformation f_l with weights $W^{(l)}$

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \cdots \circ f_1(\mathbf{x})$$

- Using the chain rule

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdots \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}$$

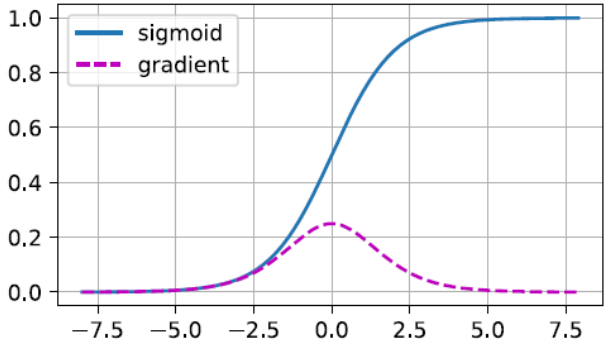
- With inappropriate initial values and/or activation functions
  - Vanishing gradients: multiply matrices with very small values
  - Exploding gradients: multiply matrices with very large values

Vanishing Gradients

- Historically sigmoid activation function was popular
  - Inspiration from biology, neurons are either on or off
  - Smooth gradient
- Saturation problem
  - Only a small region where the gradient is not near zero
  - Summing over many input nodes leads to saturation
  - Mean of input nodes (mean of the sigmoid) is 0.5

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.sigmoid(x)
y.backward(torch.ones_like(x))

d2l.plot(x.detach().numpy(), [y.detach().numpy(), x.grad.numpy()],
         legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



Exploding Gradients


```
M = torch.normal(0, 1, size=(4, 4))
print('a single matrix \n',M)
for i in range(100):
    M = M @ torch.normal(0, 1, size=(4, 4))
print('after multiplying 100 matrices\n', M)
```

divegence

```
a single matrix
 tensor([[-0.8755, -1.2171,  1.3316,  0.1357],
        [ 0.4399,  1.4073, -1.9131, -0.4608],
        [-2.1420,  0.3643, -0.5267,  1.0277],
        [-0.1734, -0.7549,  2.3024,  1.3085]])
after multiplying 100 matrices
 tensor([[-2.9185e+23,  1.3915e+25, -1.1865e+25,  1.4354e+24],
        [ 4.9142e+23, -2.3430e+25,  1.9979e+25, -2.4169e+24],
        [ 2.6578e+23, -1.2672e+25,  1.0805e+25, -1.3072e+24],
        [-5.2223e+23,  2.4899e+25, -2.1231e+25,  2.5684e+24]])
```

Problem with Initialization with Fixed Distributions

- Suppose use a fixed distribution with zero mean and variance σ^2 for

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j$$

- And further suppose inputs x_j have zero mean and variance γ^2
- Mean and variance of o_i

$$E[o_i] = \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j]$$

$$= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j]$$

$$= 0,$$

$$\text{Var}[o_i] = E[o_i^2] - (E[o_i])^2$$

$$= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0$$

$$= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2]$$

$$= n_{\text{in}} \sigma^2 \gamma^2.$$

Variance grows with number of inputs

Xavier (Glorot) Initialization for Sigmoids

- From the previous slide, we can select σ, such that $n_{\text{in}}\sigma^2 = 1$
- But this only works for forward propagation
- Need to make sure backward propagation does not blow up, $n_{\text{out}}\sigma^2 = 1$
- Xavier (Glorot) Initialization:

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}.$$

ReLU ⇒ He initialization (Kaiming)

Generalization in Deep Learning

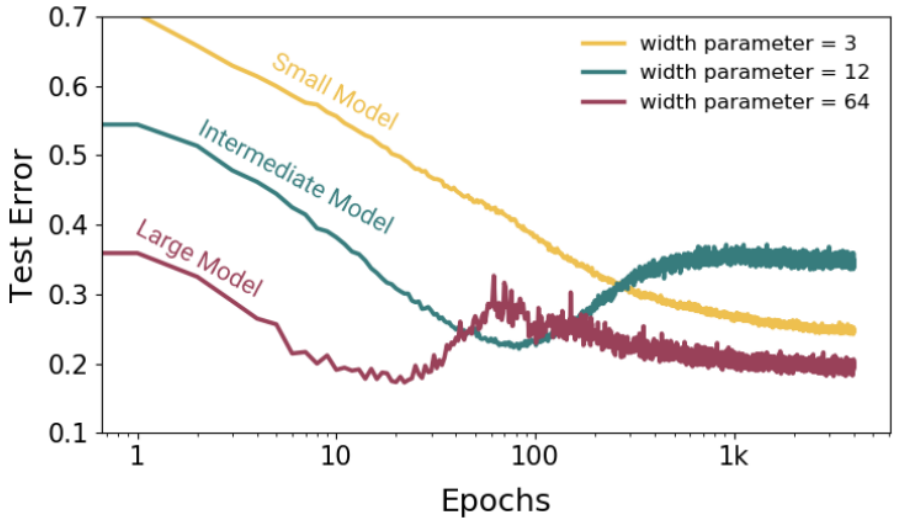
- Why does deep learning models generalize well in practices?
  - Models have very high VC dimensions.
    For sigmoid activation function, VC dimension is at least $\Omega(\|E\|^2)$
  - Models are flexible enough to fit image data with random labels
- Traditional answer is deep learning requires regularization, like
  - Early stopping
  - Weight decay
  - Dropout
  - Adding noise to inputs

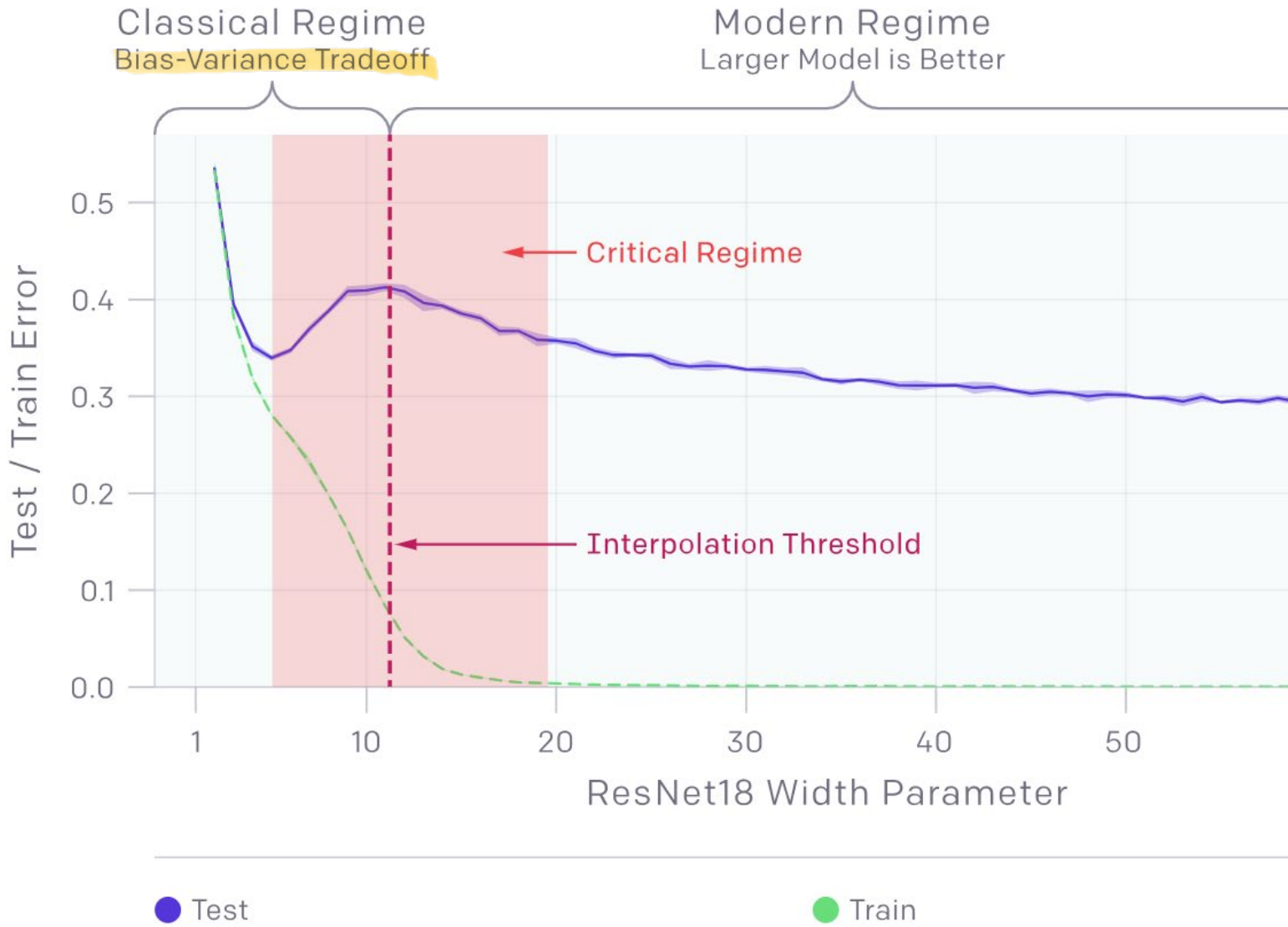
Generalization in Deep Learning

- However, recently research paints a fuzzier picture
- "Understanding Deep Learning (Still) Requires Rethinking Generalization," Zhang et al (2021)
  - "Despite their massive size, successful deep artificial neural networks can exhibit a remarkably small gap between training and test performance."
    - Why? Properties of the model architecture? Regularization methods?
  - "Deep neural networks easily fit random labels."
    - Deep neural networks shatter training data
  - "Explicit regularization may improve generalization performance, but is neither necessary nor by itself sufficient for controlling generalization error."
    - SGD acts like regularizer

Deep Double Descent

Nakkiran et al (2019)



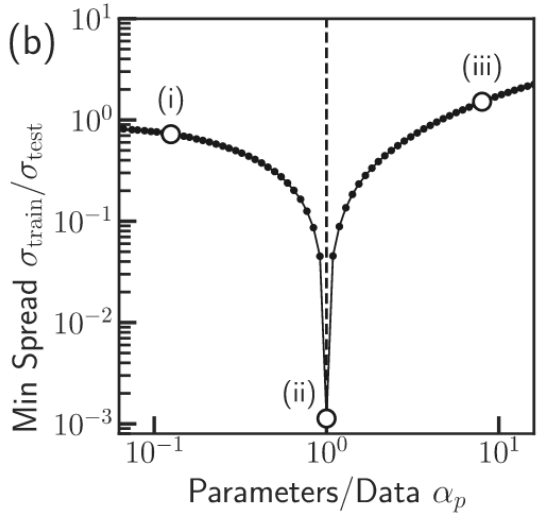


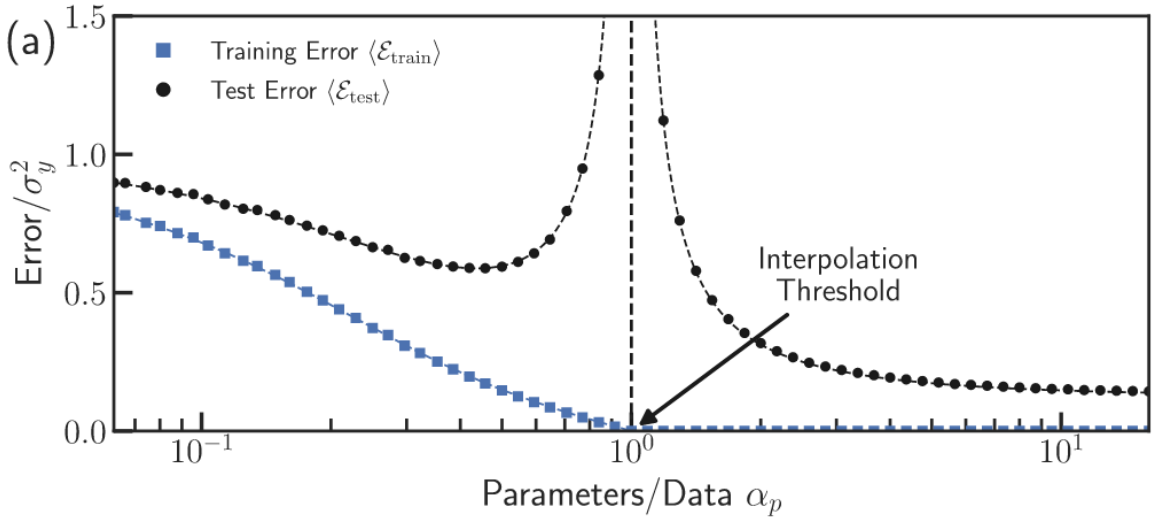
Explaining Deep Double Descent

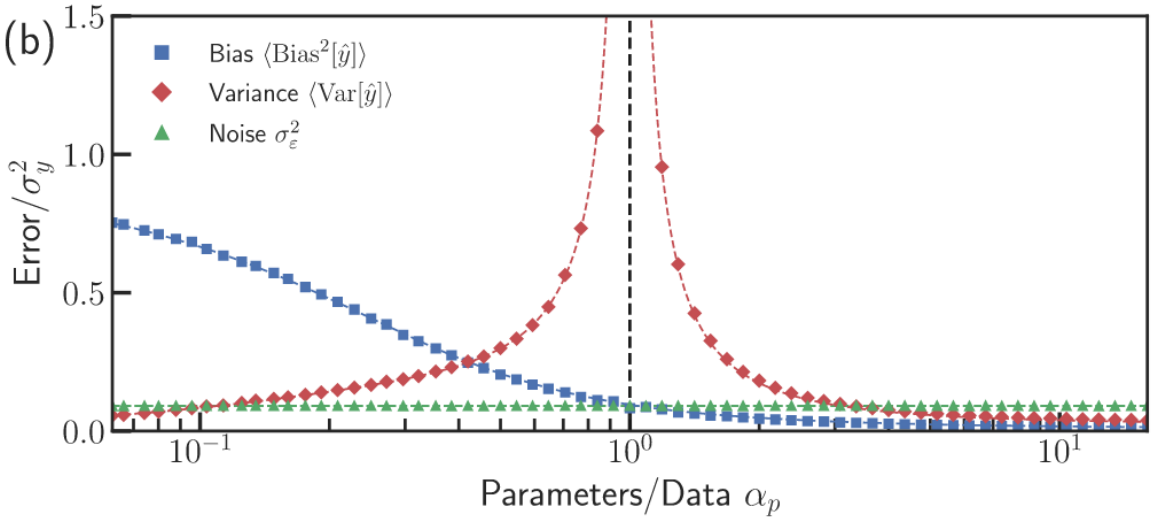
- Still an area of research
- Approaches
  - Extend Bias-Variance formulation. The classic formulation assume training set selection as the only random process, but in deep learning weights are random (random parameter initiation, minibatches). Examine role of weight regularization schemes.
  - Study problem with simplifying assumptions, e.g., assume linear regression or assume in "lazy training" regime that only trains the last layers of deep model.
- Sample papers
  - Adlam, Ben, and Jeffrey Pennington. 2020. "Understanding Double Descent Requires A Fine-Grained Bias-Variance Decomposition." *Advances in Neural Information Processing Systems* 33: 11022–32. https://proceedings.neurips.cc/paper_files/paper/2020/hash/7d420e2b2939762031eed0447a9be19f-Abstract.html.
  - Rocks, Jason W., and Pankaj Mehta. 2022. "Memorizing without Overfitting: Bias, Variance, and Interpolation in Overparameterized Models." *Physical Review Research* 4 (1): 013201. https://doi.org/10.1103/PhysRevResearch.4.013201.
  - Schaeffer, Rylan, Mikail Khona, Zachary Robertson, et al. 2023. "Double Descent Demystified: Identifying, Interpreting & Ablating the Sources of a Deep Learning Puzzle." arXiv:2303.14151. Preprint, arXiv, March 24. https://doi.org/10.48550/arXiv.2303.14151.

Rocks and Mehta, 2022

- Extends Bias-Variance formulation, and "lazy training" regime (two-layer NN with weight decay loss function)
- Bias error: imperfect models and incomplete exploration of features
- Variance error: overfitting from poorly sampled direction in feature space of training set $X_{n\times p}$
- Why does variance explode near interpolation threshold?
  - Error dominated by the smallest non-zero eigenvalue σ of $X^T X$, empirical covariance matrix
  - For $n \gg p$, good estimation of covariance. For $n < p$, $n - p$ zero eigenvalues.







Dropout

- Classical generalization theory suggests we should aim for a simpler model to close the generalization gap
- One notion of simpler model is smoothness with respect to the input
- Output should not be sensitive to small changes in input
- Adding random noise to input images can improve classification
- Bishop (1995) showed that adding noise is equivalent to Tikohonov regularization (aka ridge regression, aka l_2 normalization)
- Dropout extends this idea to nodes in the neural network

Dropout

- Original rationale given is based on analogy from biology
- Just as selective pressure can cause co-adaptation of genes, the training process can cause neurons to depend unnecessarily on the neurons from the previous layer
- Dropout proposes a random activation function returns 0 with probability p

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

$E(h') = \frac{h}{1-p} \times (1-p) = h$

- Where h is original activation function. By design $E[h'] = h$
- Dropout is often credited for AlexNet winning the ImageNet competition

See Notebook

- chapter_multilayer-perceptrons/dropout.ipynb