Data 188: Introduction to Deep Learning

Midterm Review

Speaker: Eric Kim
Lecture 14 (Week 07)
2026-03-05, Spring 2026. UC Berkeley.

Acknowledgement: slides are based on CMU's Deep Learning Systems course (10-414/714, Fall 2025). Thanks to: Prof. Zico Kolter, Prof. Tianqi Chen, Prof. Tim Dettmers.

Announcements

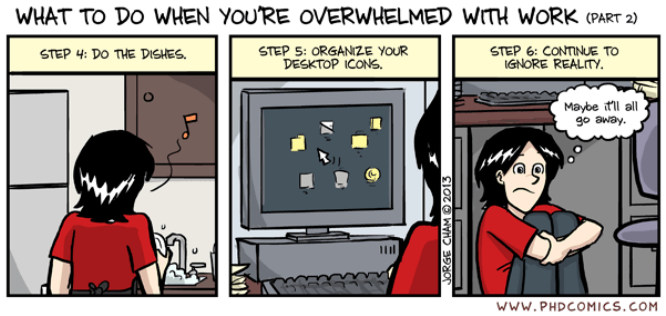
- HW2 is out! Due in a few days.
- Midterm next week!
  - "Midterm Study Guide" released (see Edstem post)
  - Midterm assignments will be emailed out soon

Outline

Overview of class concepts

Practice Problems

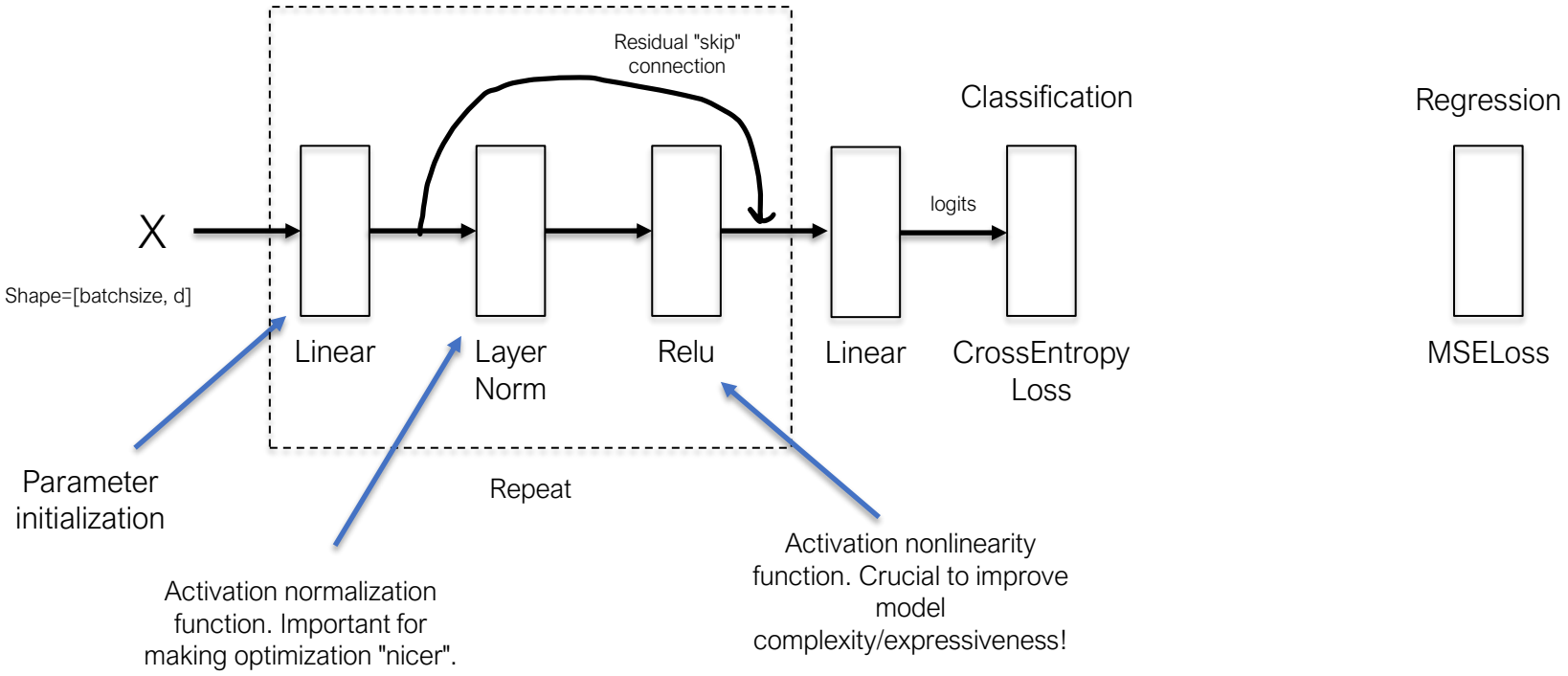
What have we done in this class?



https://phdcomics.com/comics/archive.php?comicid=820
https://phdcomics.com/comics/archive.php?comicid=1655
https://phdcomics.com/comics/archive.php?comicid=1656

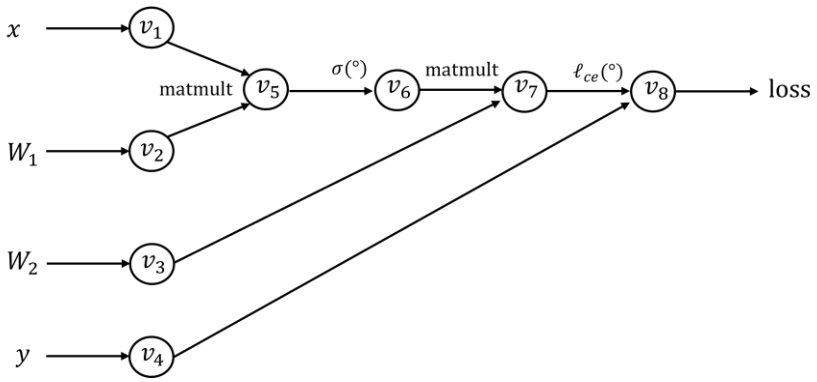
Model architecture design

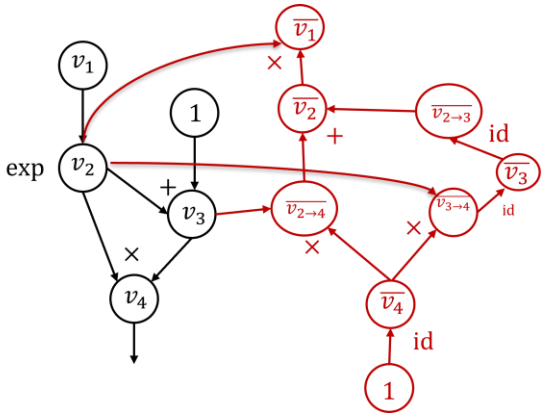
Learned (nearly) everything required to design model architectures for: classification, regression



Model Training

Learned how model training works, via the backprop algorithm (reverse mode autodiff)






```
class MatMul(Op):
    def forward(self, lhs, rhs):
        return lhs @ rhs
    def gradient(self, out_grad, node):
        lhs, rhs = node.inputs
        # calculate dloss_dout @ dout/dlhs
        dloss_dlhs = out_grad @ rhs.T
        # calculate dloss_dout @ dout/drhs
        dloss_drhs = lhs.T @ out_grad
        return dloss_dlhs, dloss_drhs
```

```
def gradient(out):
  node_to_grad = {out:  [1]}
  for i in reverse_topo_order(out):
    v̄_i = ∑_j v̄_{i→j} = sum(node_to_grad[i])
    for k ∈ inputs(i):
        compute v̄_{k→i} = v̄_i ∂v_i/∂v_k
        append v̄_{k→i} to node_to_grad[k]
    return adjoint of input v̄_input
```

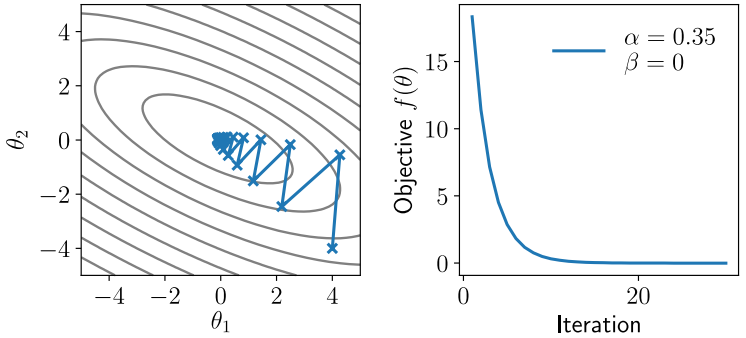
Recall:

$$\overline{v_i} = \frac{\partial y}{\partial v_i}$$

$$\overline{v_{i\to j}} = \overline{v_j}\,\frac{\partial v_j}{\partial v_i}$$

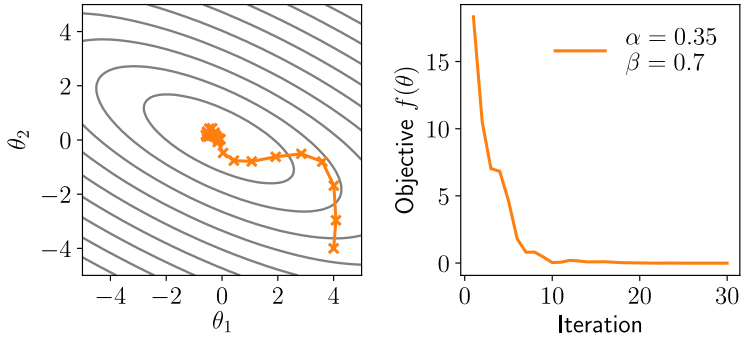
Optimization

Learned how we update model parameters, given adjoints (gradients).



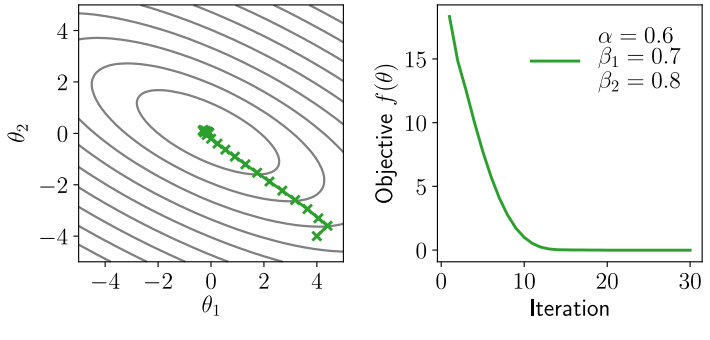
"Vanilla" SGD

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta f(\theta_t)$$



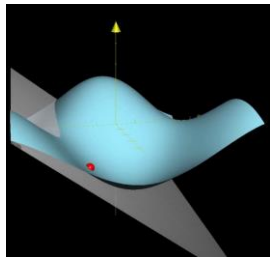
SGD+Momentum

$$u_{t+1} = \beta u_t + (1-\beta)\nabla_\theta f(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha u_{t+1}$$

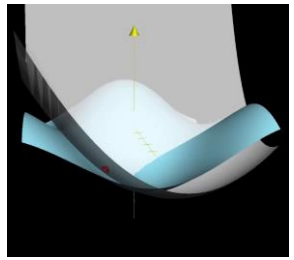


Adam

$$u_{t+1} = \beta_1 u_t + (1-\beta_1)\nabla_\theta f(\theta_t)$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\left(\nabla_\theta f(\theta_t)\right)^2$$
$$\theta_{t+1} = \theta_t - \alpha \frac{u_{t+1}}{\left(\sqrt{v_{t+1}} + \epsilon\right)}$$



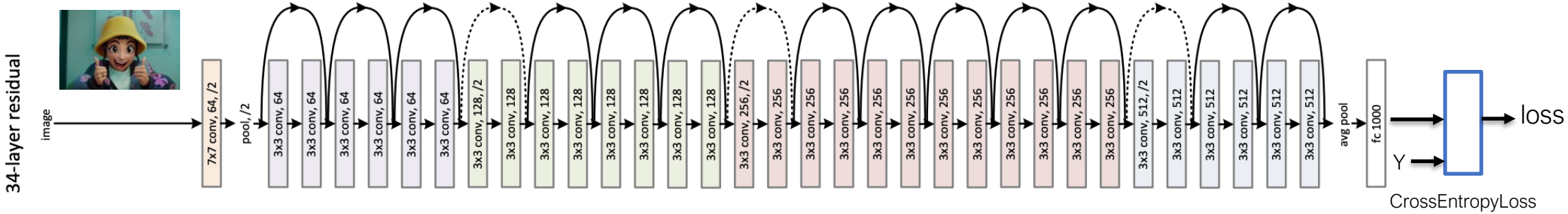
Linear approximation (first order methods)

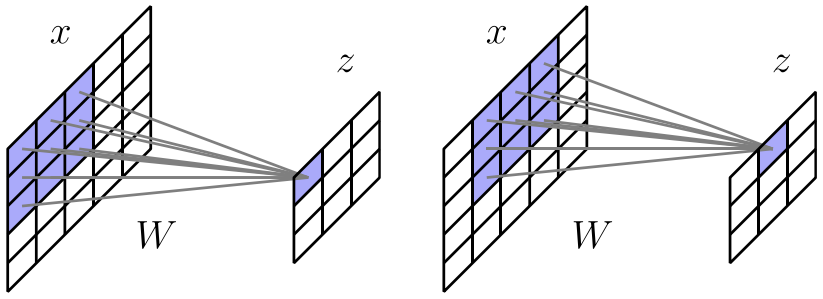


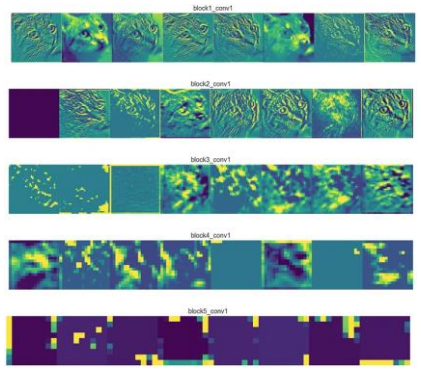
Quadratic approximation (second order methods)

ConvNets

Learned how we can better handle image data, via convolutional neural networks ("Convnets").







Computer Vision: object detection

Studied the "YOLO" object detection model, which frames object detection as box regression+classification on top of a ConvNet.

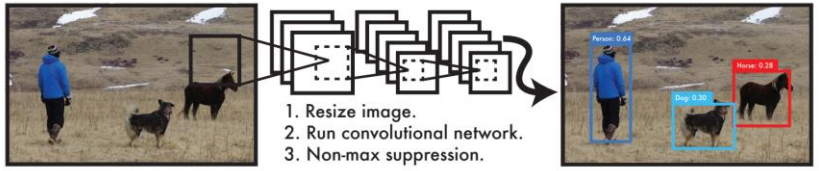


Figure 1: The YOLO Detection System. Processing images with YOLO is simple and straightforward. Our system (1) resizes the input image to 448 × 448, (2) runs a single convolutional network on the image, and (3) thresholds the resulting detections by the model's confidence.

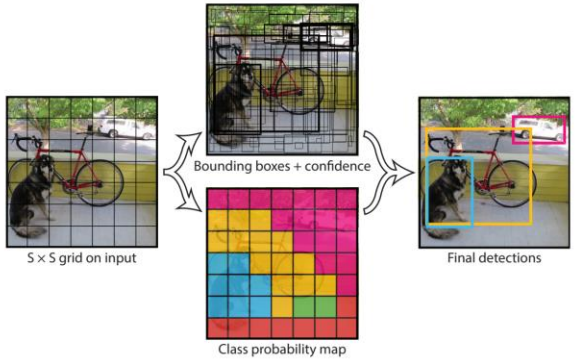
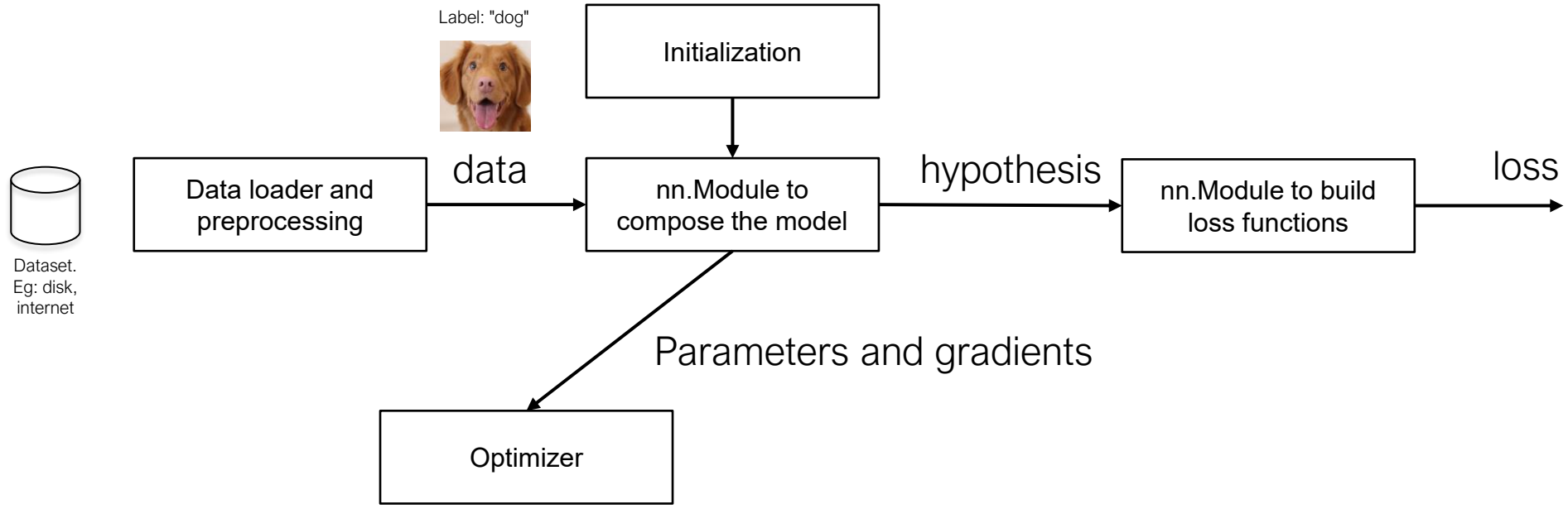


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

Deep learning library abstractions

Learned (and implemented!) the typical abstractions/interfaces used in modern popular deep learning libraries (ex: pytorch, tensorflow, needle)



Where are we now, and what is next?

At this point, you should be able to read and understand the deep learning code of state of the art, powerful models.

Further, you understand the inner workings of every step of the deep learning pipeline (model architecture design, backprop, optimizers).

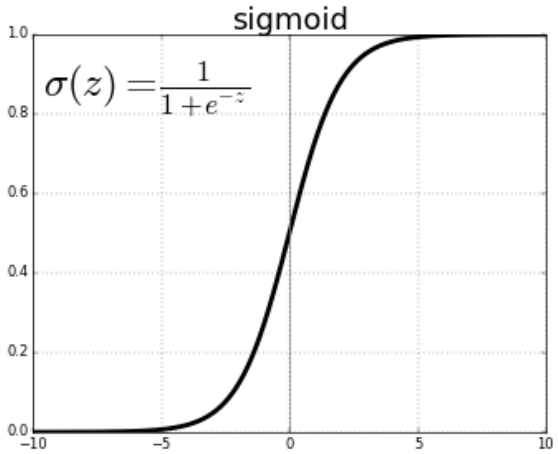
Next, we will move on from needle and start learning the pytorch framework.

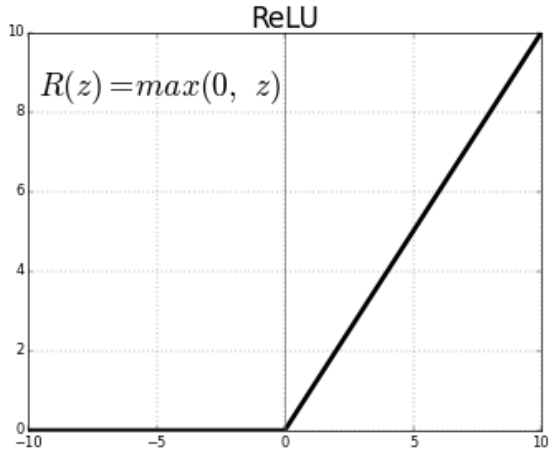
Fortunately, pytorch is extremely similar to needle, so this transition should be easy!

Sigmoid vs Relu

Question: suppose we used the Sigmoid function instead of Relu as our nonlinear activation function. What might go wrong?

Hint: consider the gradient of Sigmoid and Relu for large/small values.





Answer: Sigmoid can suffer from vanishing gradient problem if input activations have large magnitude, eg <-5 or >+5.
Relu does not suffer from this problem, but does have a "dying Relu" problem (see Disc03 Q2)

Layer Stacking

Question: Suppose I have this model architecture: Linear->Linear->Linear. Why is this architecture a red flag?

Answer: composition of multiple linear operators is another linear operator. In other words: model complexity of this model is equivalent to a single Linear layer.

Question: What about: Conv2d -> Conv2d -> Conv2d?

Answer: same issue as Linear layers, Conv2d is a linear operator, so this is equivalent to a single Conv2d layer.

Question: What about: Linear -> LayerNorm -> Linear -> LayerNorm -> Linear?

Answer: LayerNorm is a nonlinear operator (due to division, and calculating var is a nonlinear operation). Thus, technically we can use LayerNorm as a nonlinear activation function. But, not recommended, use a dedicated activation function like Relu in practice!

(optional, for fun) A blog post exploring using LayerNorm as the activation function: "Re-Examining LayerNorm"

Linear Layers

4. Linear Layers (12 pts)

(a) (8 pts)

Consider the following model architecture consisting of two Linear layers (Figure 1).

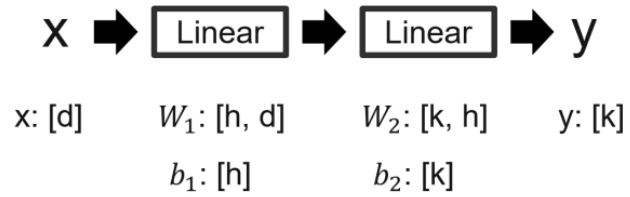


Figure 1: Model architecture with two Linear layers. Also pictured: the input/output shapes, and the shapes of the Linear weight/bias parameters. Notation: "$W_1 : [h, d]$" means that the weight matrix W_1 has shape $W_1 \in \mathcal{R}^{h \times d}$ (h rows, d columns), and "$b_1 : [h]$ means that b_1 is a column vector with h elements, aka $b_1 \in \mathcal{R}^h$.

I claim that these two Linear layers can be expressed as an equivalent single Linear layer with appropriately defined $\mathbf{W}^\star$ and $\mathbf{b}^\star$ (Figure 2).

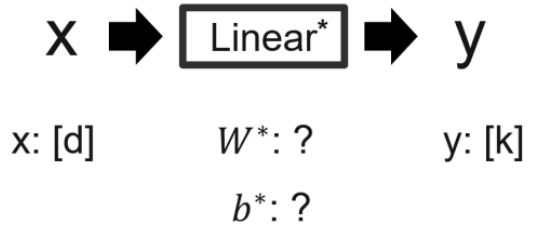


Figure 2: Desired model architecture of a single Linear layer with parameters $W^\star, b^\star$.

Show that this can be done, by expressing the new $\mathbf{W}^\star$, $b^\star$ in terms of W_1, b_1, W_2, and b_2. In other words, your new $Linear^\star$ (Figure 2) should produce the exact same outputs as the two Linear layers in Figure 1.

Note: for this problem, assume that a Linear layer performs the following operation: $Linear(x, W, b) = Wx + b$ where $x \in \mathcal{R}^d$, $W \in \mathcal{R}^{d_{out} \times d}$, $b \in \mathcal{R}^{d_{out}}$.

$W^\star$ = _____

$b^\star$ = _____

Solution:

$$W^\star = W_2 W_1$$

$$b^\star = W_2 b_1 + b_2$$

Backprop

Perform backprop (reverse-mode autodiff) on this computation graph

Linear has no bias. Residual ("skip") connection is implemented via elementwise-sum

First do it with scalar x (shape=[1]). Then, try it with vector x.shape=[d]