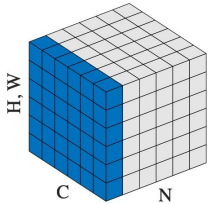
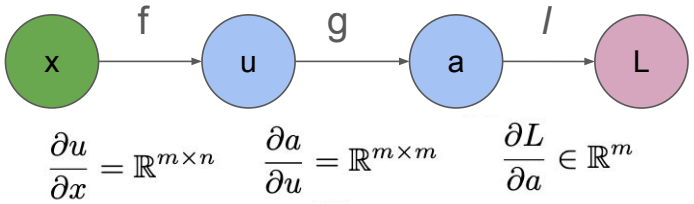
NYU CS-GY 6923 Machine Learning

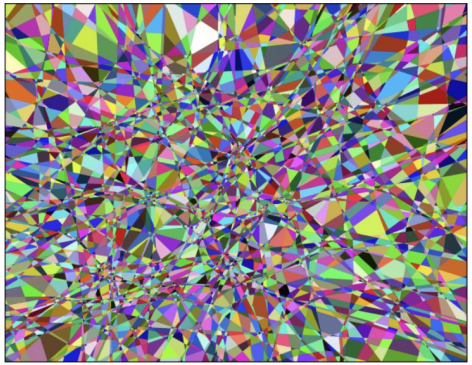
Prof. Pavel Izmailov

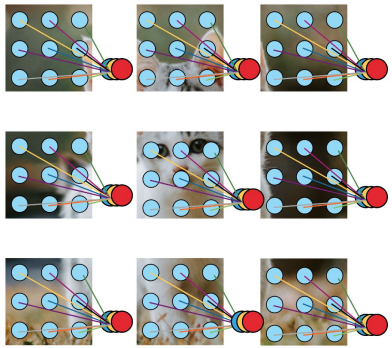
Today

- More Backprop
- Regularization in NNs
- Effect of Depth
- Initialization
- Activation Functions
- Residual Connections
- Normalization Layers
- Convolution



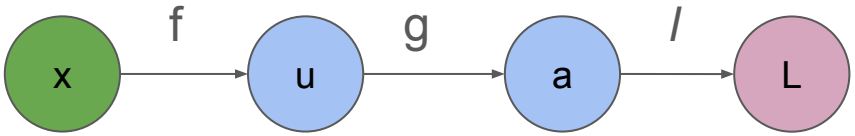






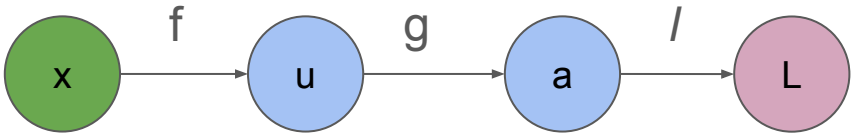

More Backprop
0.36
0.34

Backprop: Multiple Dimensions



$$x \in \mathbb{R}^n$$
$$u = f(x) \in \mathbb{R}^m$$
$$a = g(u) \in \mathbb{R}^m$$
$$L = \ell(a) \in \mathbb{R}$$

Backprop: Multiple Dimensions



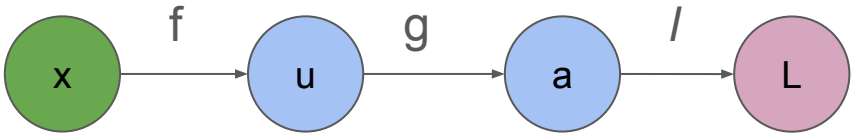
$$\frac{\partial u}{\partial x} = \mathbb{R}^{m \times n} \qquad \frac{\partial a}{\partial u} = \mathbb{R}^{m \times m} \qquad \frac{\partial L}{\partial a} \in \mathbb{R}^{m}$$

$$x \in \mathbb{R}^n$$
$$u = f(x) \in \mathbb{R}^m$$
$$a = g(u) \in \mathbb{R}^m$$
$$L = \ell(a) \in \mathbb{R}$$

$$\frac{\partial L}{\partial x} = \left(\frac{\partial u}{\partial x}\right)^T \left(\frac{\partial a}{\partial u}\right)^T \frac{\partial L}{\partial a}$$

$$(n \times m) \cdot (m \times m) \cdot (m \times 1) = (n \times 1)$$

Backprop: Multiple Dimensions



$$\frac{\partial u}{\partial x} = \mathbb{R}^{m \times n} \qquad \frac{\partial a}{\partial u} = \mathbb{R}^{m \times m} \qquad \frac{\partial L}{\partial a} \in \mathbb{R}^{m}$$

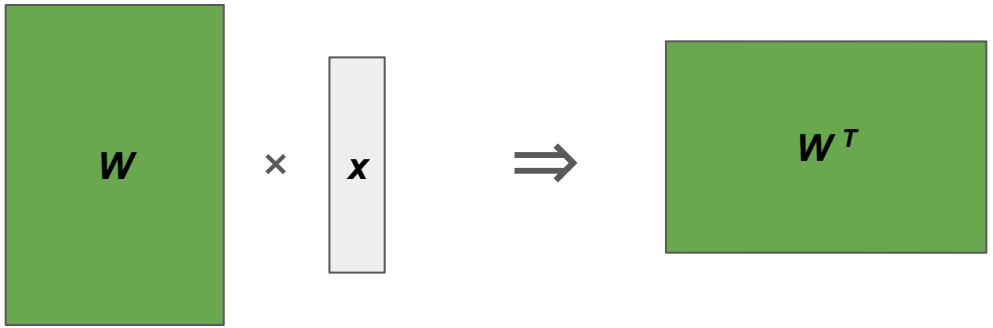
$$x \in \mathbb{R}^n$$
$$u = f(x) \in \mathbb{R}^m$$
$$a = g(u) \in \mathbb{R}^m$$
$$L = \ell(a) \in \mathbb{R}$$

$$\frac{\partial L}{\partial x} = \left(\frac{\partial u}{\partial x}\right)^T \left(\frac{\partial a}{\partial u}\right)^T \frac{\partial L}{\partial a}$$

$$(n \times m) \cdot (m \times m) \cdot (m \times 1) = (n \times 1)$$

- Order of operations is important! We will avoid matrix-matrix products
- We do not need to form the Jacobian, just jacobian-vector-product (JVP)!

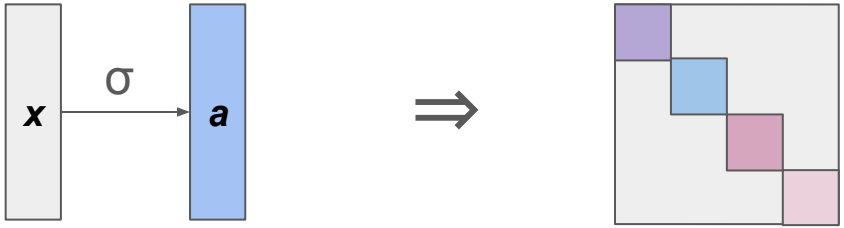
JVP: Linear Layers



$$\frac{\partial(Wx)}{\partial x} = W^T \qquad \frac{\partial(Wx)}{\partial x}u = W^T u$$

For the linear layers, the Jacobian is equal to the transposed weights

JVP: Sigmoid

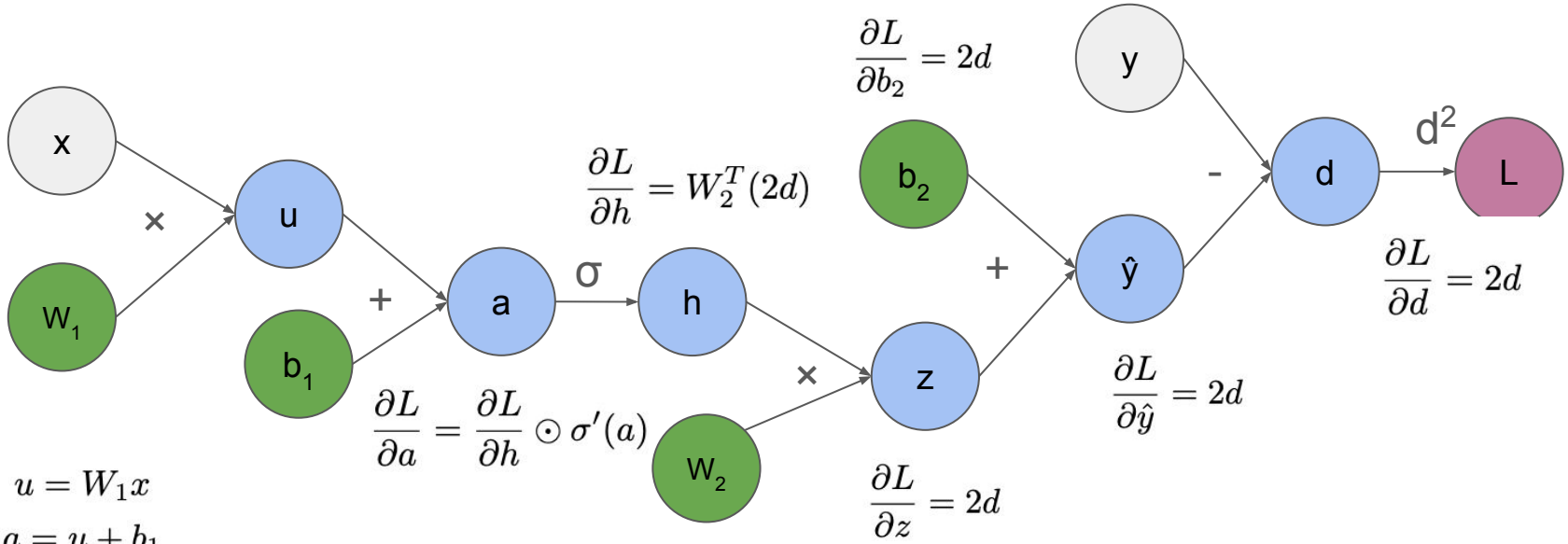


$$\frac{\partial \sigma(x)}{\partial x} = \begin{bmatrix} \sigma'(x_1) & 0 & & 0 \\ 0 & \sigma'(x_2) & & 0 \\ & & \ddots & \\ 0 & 0 & & \sigma'(x_n) \end{bmatrix}$$

$$\frac{\partial \sigma(x)}{\partial x} u = \sigma'(x) \odot u$$

For sigmoid, the Jacobian is diagonal and we do not need to construct it.

Backprop: MLP



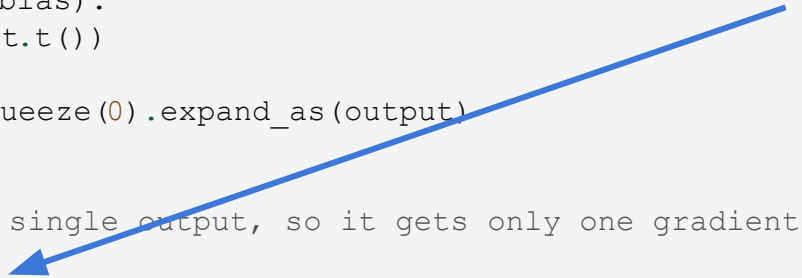
$$u = W_1 x$$
$$a = u + b_1$$
$$h = \sigma(a)$$
$$z = W_2 h$$
$$\hat{y} = z + b_2$$
$$d = \hat{y} - y$$
$$L = d^T d = \|d\|^2$$

https://docs.pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

JVP: PyTorch

```
class LinearFunction(Function):

    @staticmethod
    def forward(input, weight, bias):
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    # This function has only a single output, so it gets only one gradient
    @staticmethod
    def backward(ctx, grad output):
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None

        if ctx.needs input grad[0]:
            grad input = grad output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)

        return grad_input, grad_weight, grad_bias
```

Backward takes grad_output as input, and implements JVP!

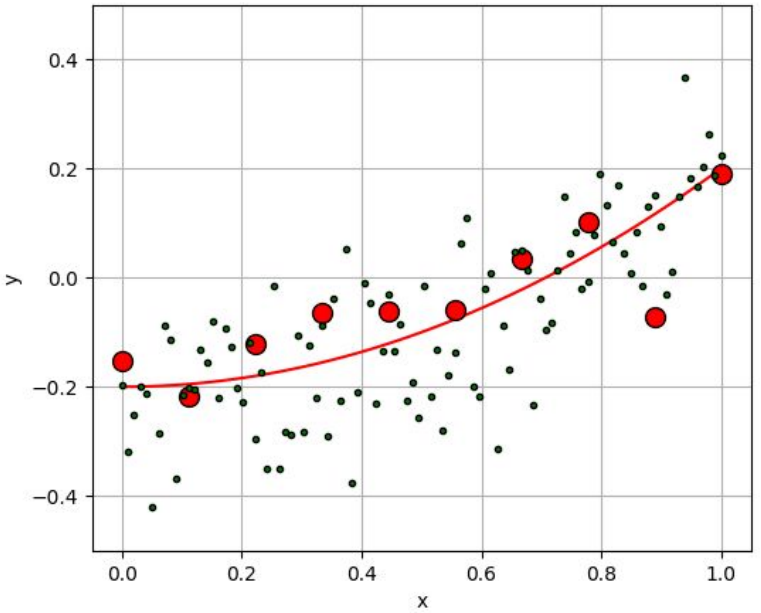


https://docs.pytorch.org/docs/main/notes/extending.html


NN Regularization
0.36
0.34

Overfitting Example

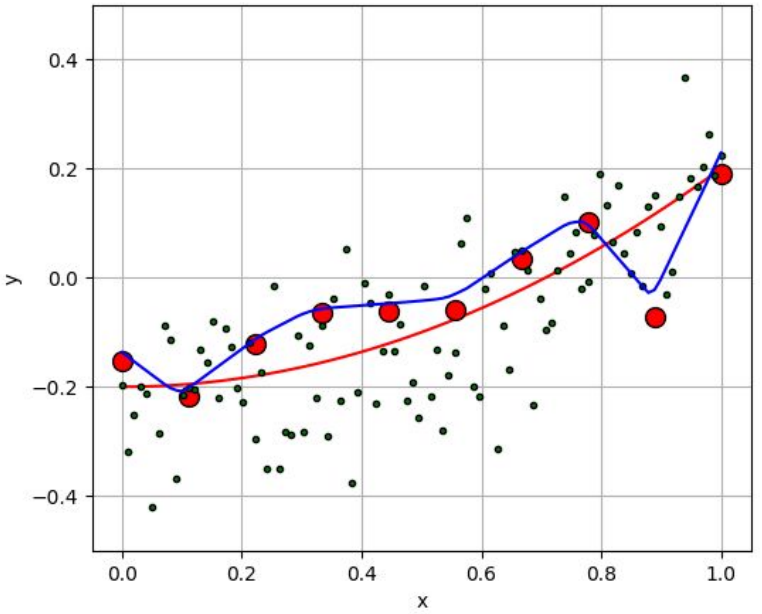
- Let's fit a neural network on a quadratic function with noise



$$y(x) = -0.2 + 0.4x^2 + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 0.1^2)$$

Overfitting Example

- Let's fit a neural network on a quadratic function with noise
- 1 hidden layer MLP with 256 neurons overfits



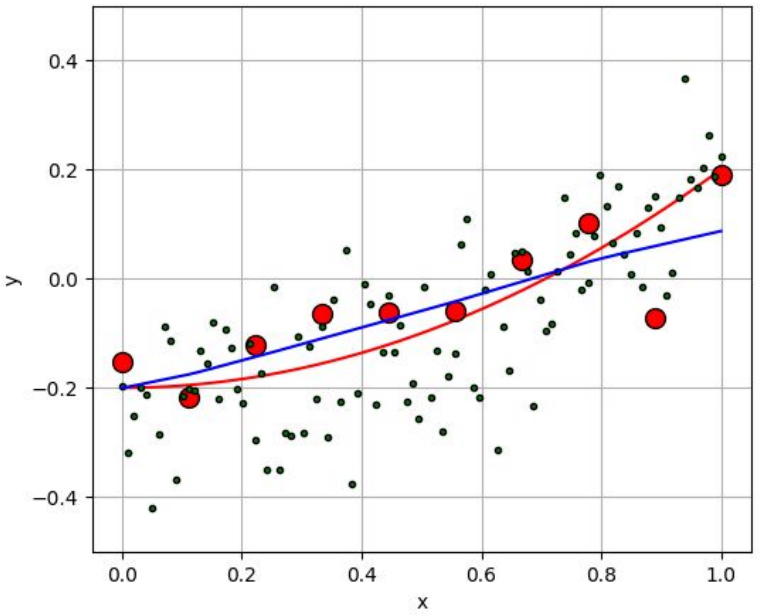
$$y(x) = -0.2 + 0.4x^2 + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 0.1^2)$$

```
tr_loss=.0006
te_loss=0.017
```

Weight Decay

$$L(w) = L_{\text{MSE}}(w) + \lambda \|w\|^2$$

- Weight decay: add L2-regularization to the loss
- It's equivalent to L2 for SGD but not Adam



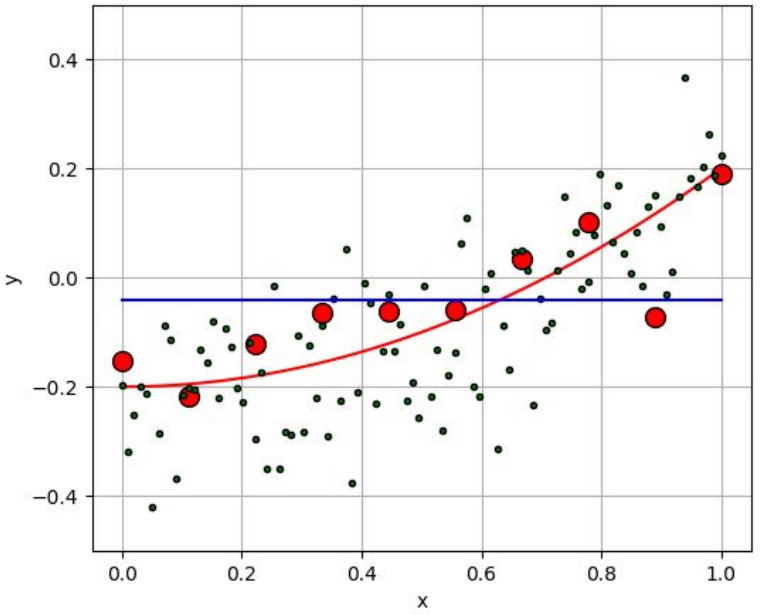

```
tr_loss=.004
te_loss=0.014
```

https://arxiv.org/abs/1711.05101

Weight Decay

$$L(w) = L_{\text{MSE}}(w) + \lambda \|w\|^2$$

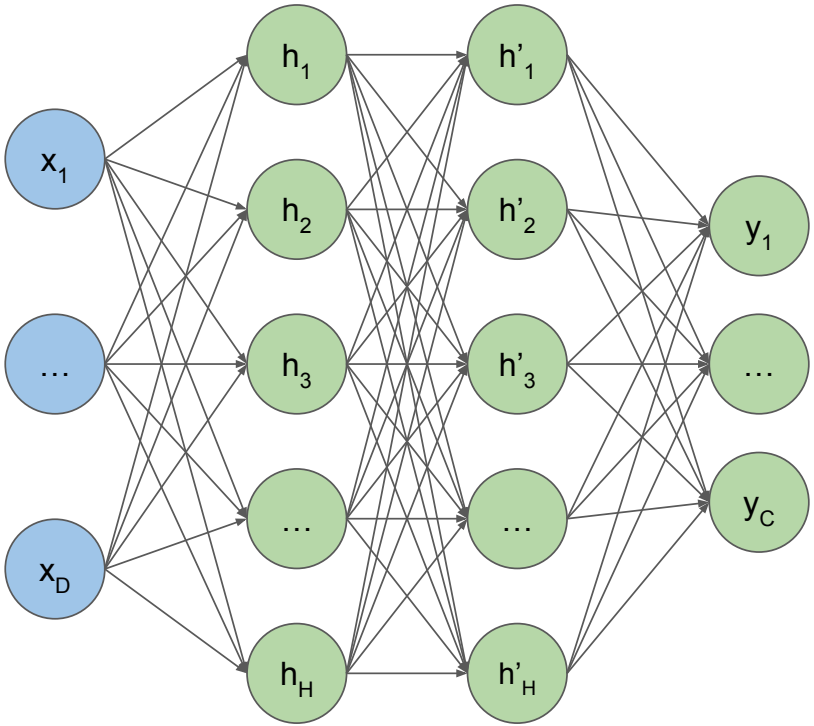
- Weight decay: add L2-regularization to the loss
- It's equivalent to L2 for SGD but not Adam
- Too much wd → underfitting




```
tr_loss=.013
te_loss=0.029
```

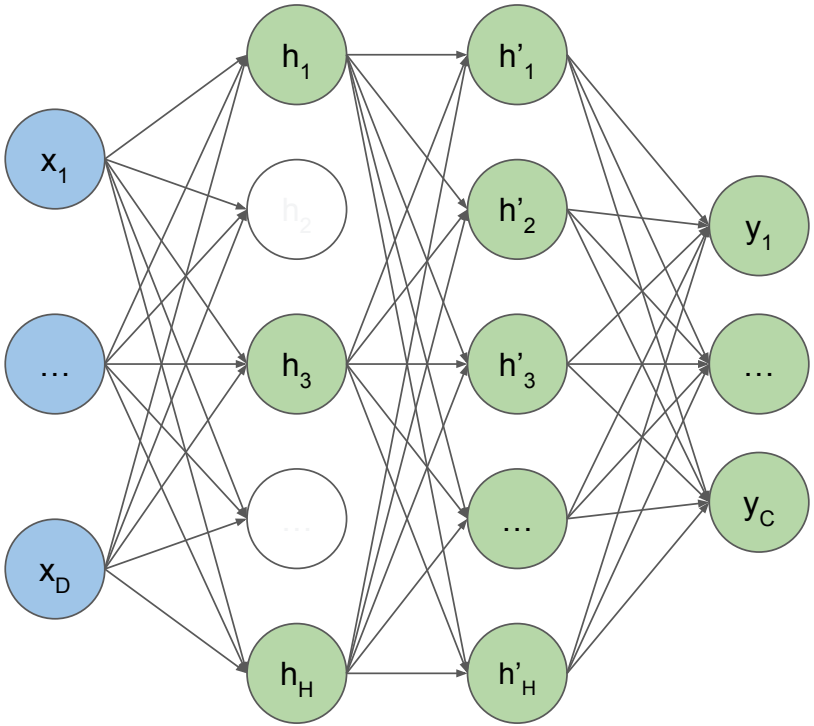
https://arxiv.org/abs/1711.05101

Dropout



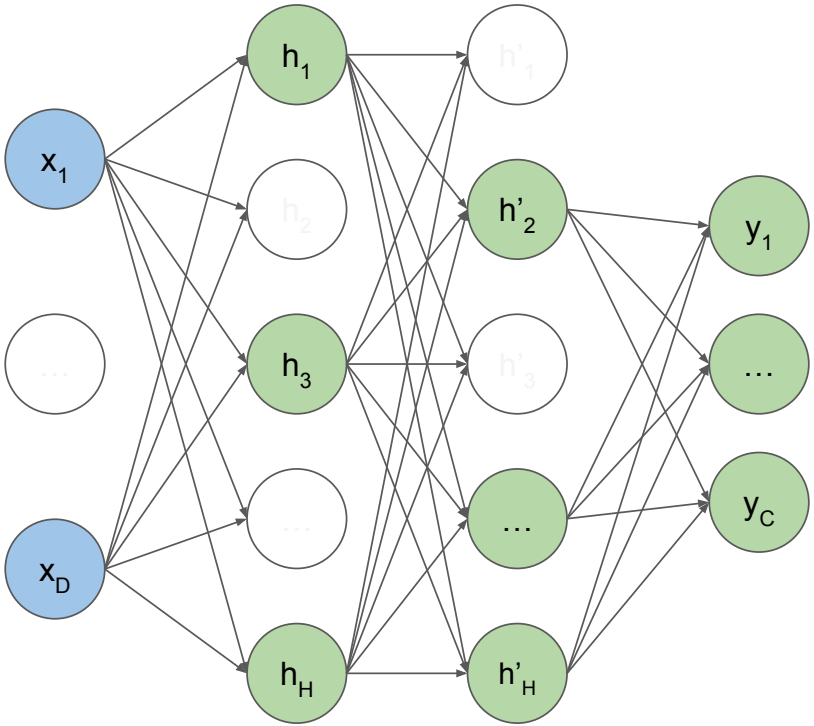
- We will randomly set *p* fraction of activations to 0

Dropout



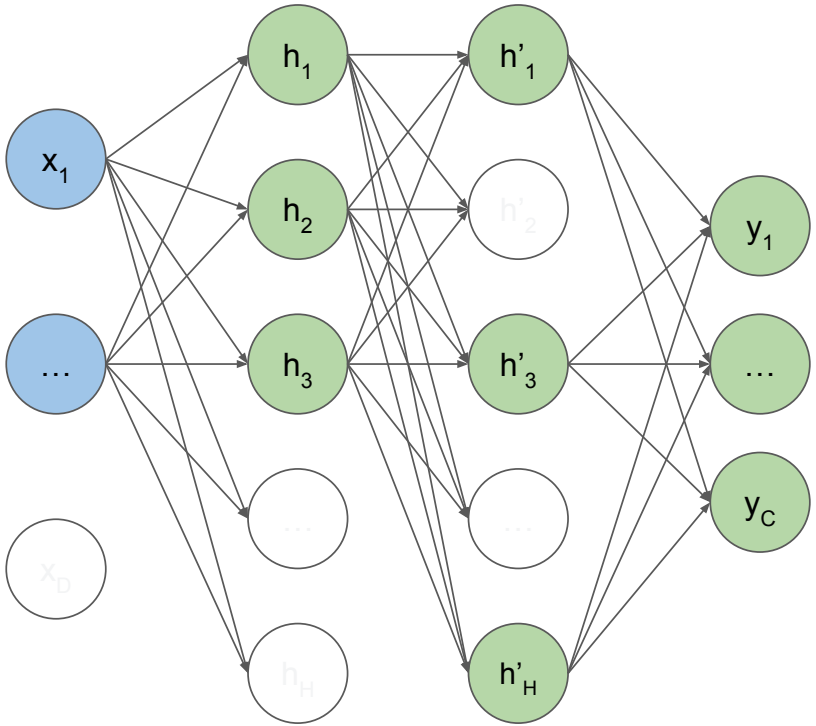
- We will randomly set *p* fraction of activations to 0

Dropout



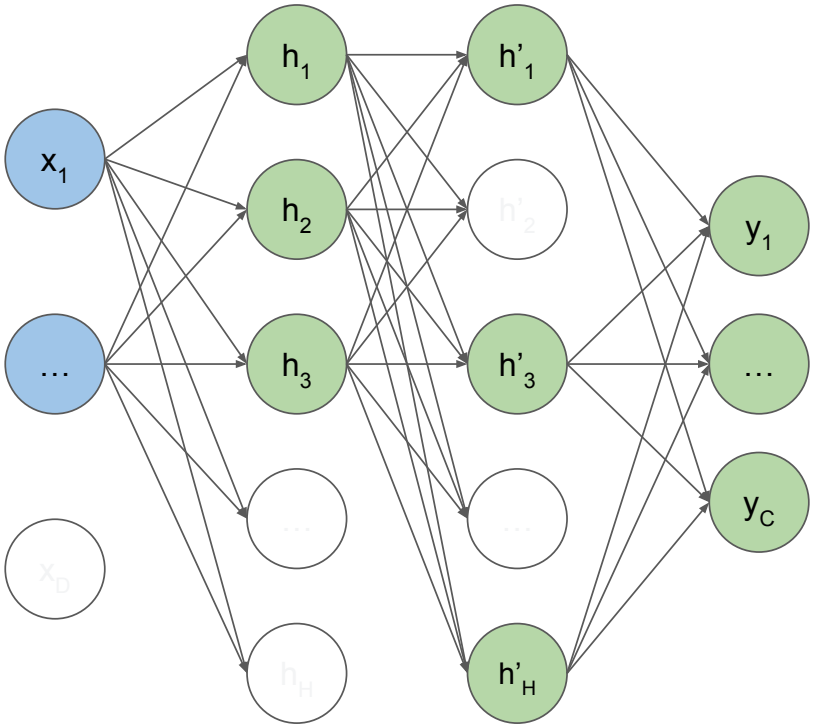
- We will randomly set *p* fraction of activations to 0
- Can apply in multiple layers, or even at inputs

Dropout



- We will randomly set *p* fraction of activations to 0
- Can apply in multiple layers, or even at inputs
- We mask out a different set of activations each time

Dropout

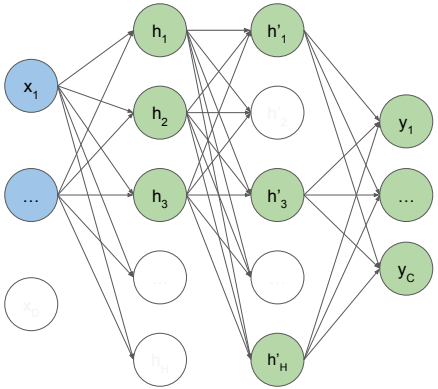


- We will randomly set *p* fraction of activations to 0
- Can apply in multiple layers, or even at inputs
- We mask out a different set of activations each time
- Dropout probability is the parameter to tune

Dropout: Test Time

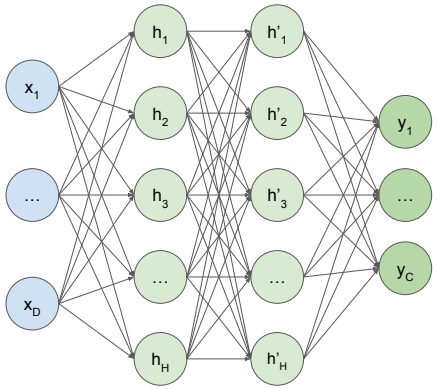
Train time

$$\tilde{\mathbf{x}} = \mathbf{x} \odot \mathbf{m}_0, \quad \mathbf{m}_0 \sim \text{Bernoulli}(1 - p_0)$$
$$\mathbf{h}_1 = \sigma(\mathbf{W}_1 \tilde{\mathbf{x}} + \mathbf{b}_1)$$
$$\tilde{\mathbf{h}}_1 = \mathbf{h}_1 \odot \mathbf{m}_1, \quad \mathbf{m}_1 \sim \text{Bernoulli}(1 - p_1)$$
$$\mathbf{h}_2 = \sigma(\mathbf{W}_2 \tilde{\mathbf{h}}_1 + \mathbf{b}_2)$$
$$\tilde{\mathbf{h}}_2 = \mathbf{h}_2 \odot \mathbf{m}_2, \quad \mathbf{m}_2 \sim \text{Bernoulli}(1 - p_2)$$
$$\mathbf{y} = \mathbf{W}_3 \tilde{\mathbf{h}}_2 + \mathbf{b}_3$$

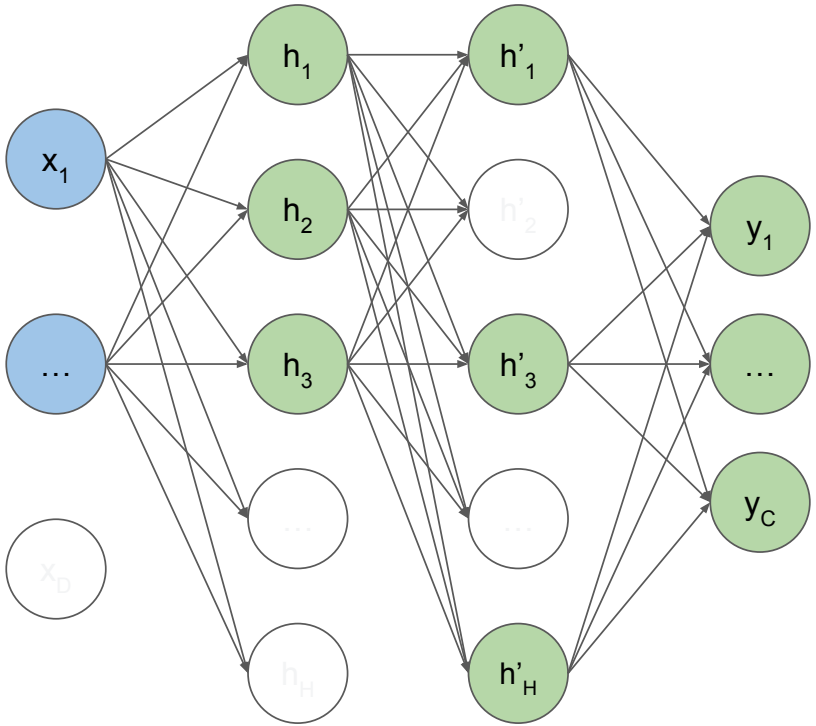


Test time

$$\tilde{\mathbf{x}} = (1 - p_0) \cdot \mathbf{x},$$
$$\mathbf{h}_1 = \sigma(\mathbf{W}_1 \tilde{\mathbf{x}} + \mathbf{b}_1)$$
$$\tilde{\mathbf{h}}_1 = (1 - p_1) \cdot \mathbf{h}_1,$$
$$\mathbf{h}_2 = \sigma(\mathbf{W}_2 \tilde{\mathbf{h}}_1 + \mathbf{b}_2)$$
$$\tilde{\mathbf{h}}_2 = (1 - p_2) \cdot \mathbf{h}_2$$
$$\mathbf{y} = \mathbf{W}_3 \tilde{\mathbf{h}}_2 + \mathbf{b}_3$$

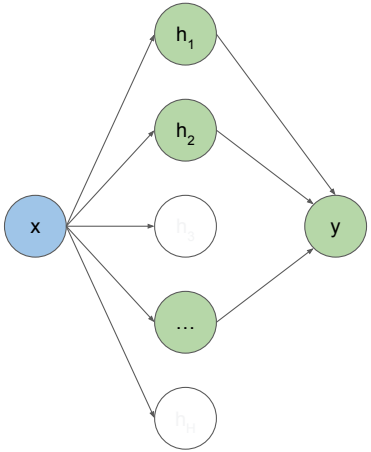


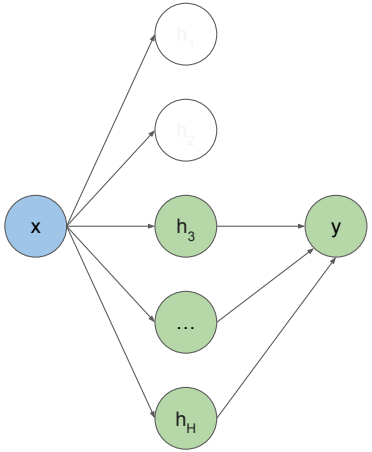
Dropout: Intuition

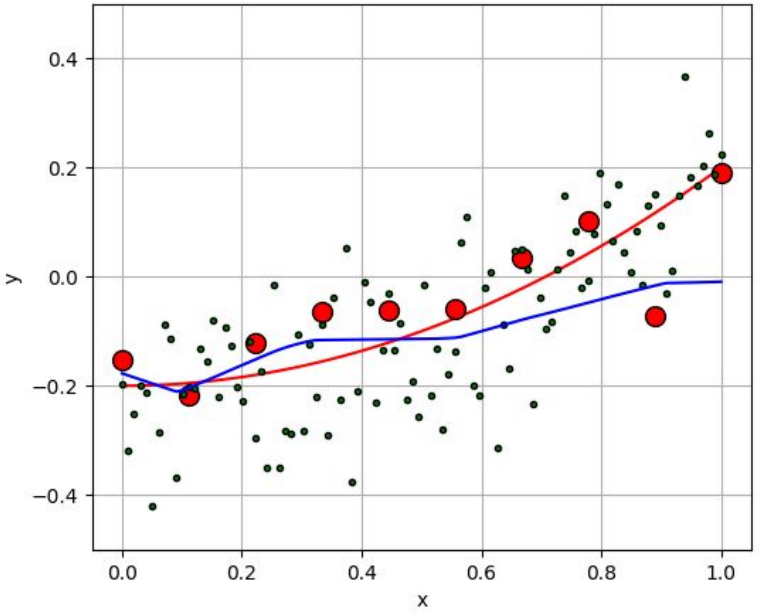


- Reduce co-reliance of neurons
- More robust solution, built-in redundancy
- Approximates ensemble of all subnetworks
- *Dropout is not very popular for large models anymore*

Dropout



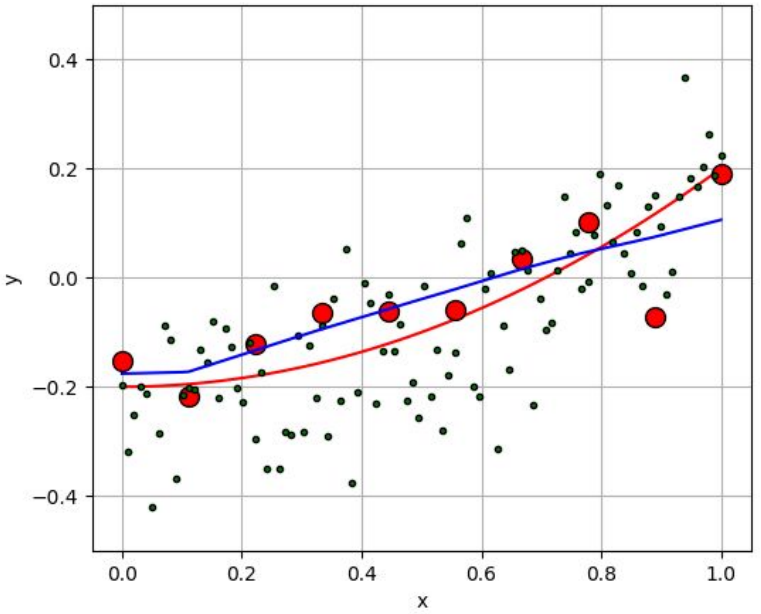





```
tr loss=.009
te_loss=0.016
```

Early Stopping

- Early stopping: just train for fewer steps
- Also serves as a regularization

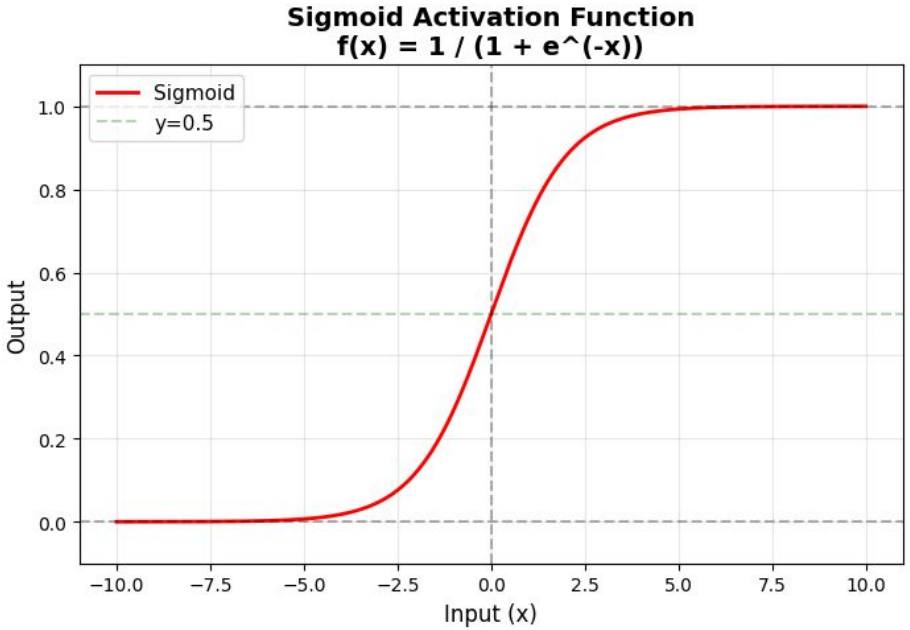



```
tr_loss=.0037
te_loss=0.015
```

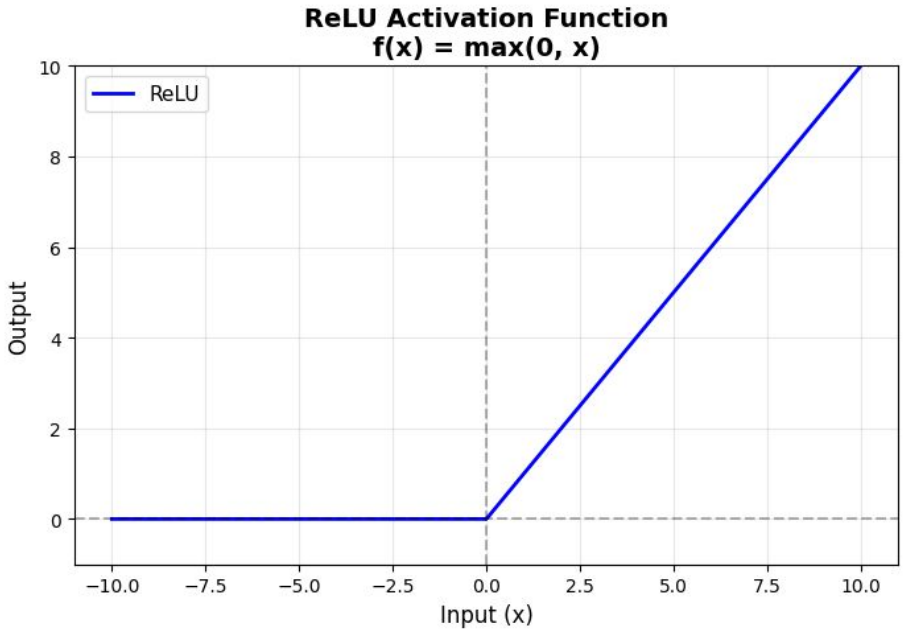
All of these regularization strategies are quite unreliable in this example, but they do work at larger scales.

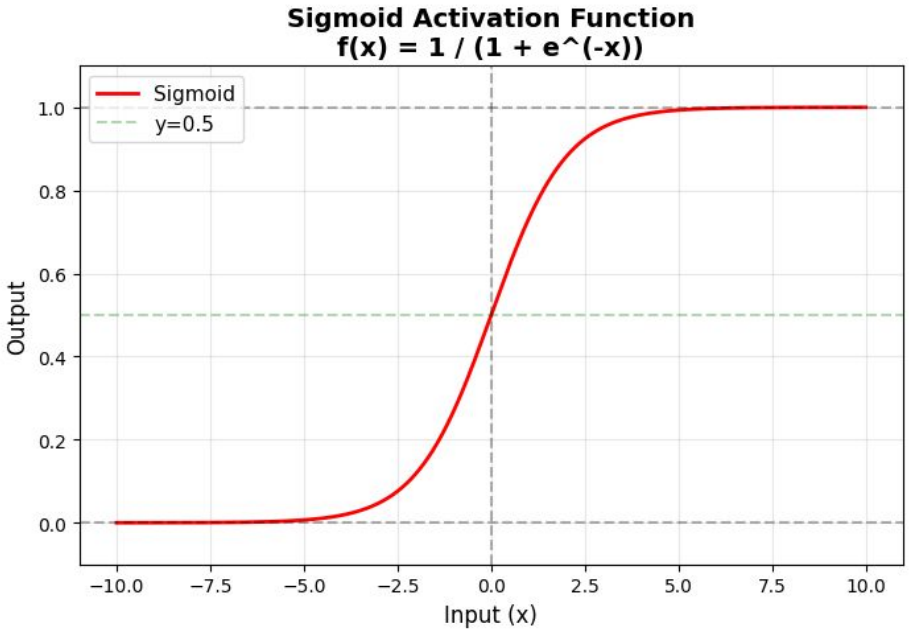

Going Deep
0.36
0.34

ReLU activation



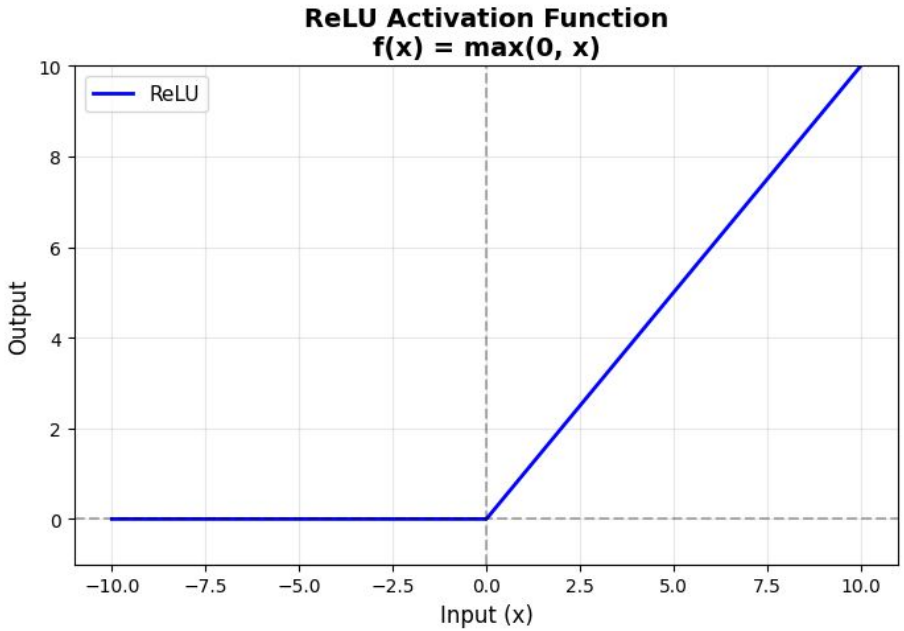
ReLU activation

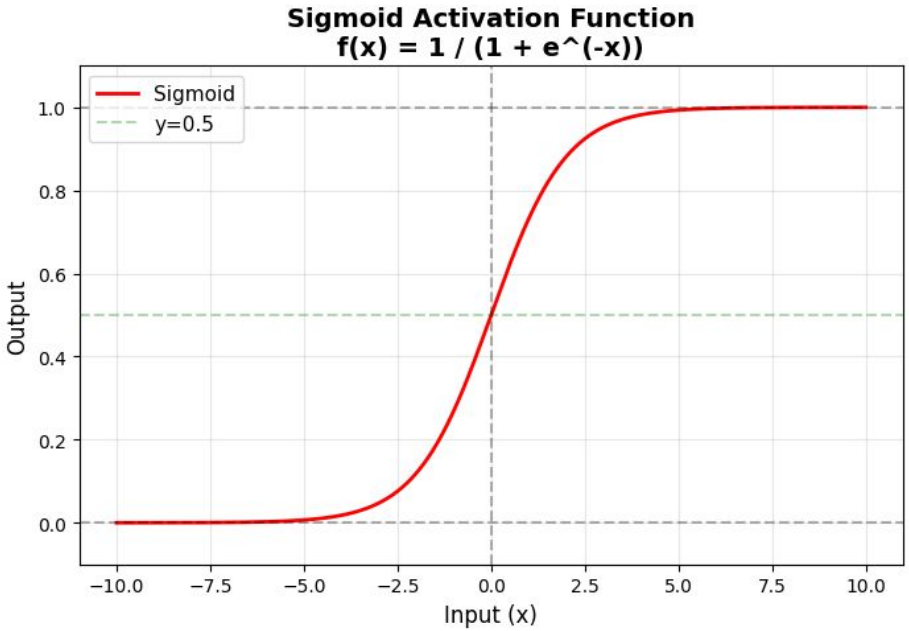




ReLU is not smooth at zero, what are the implications?

ReLU activation





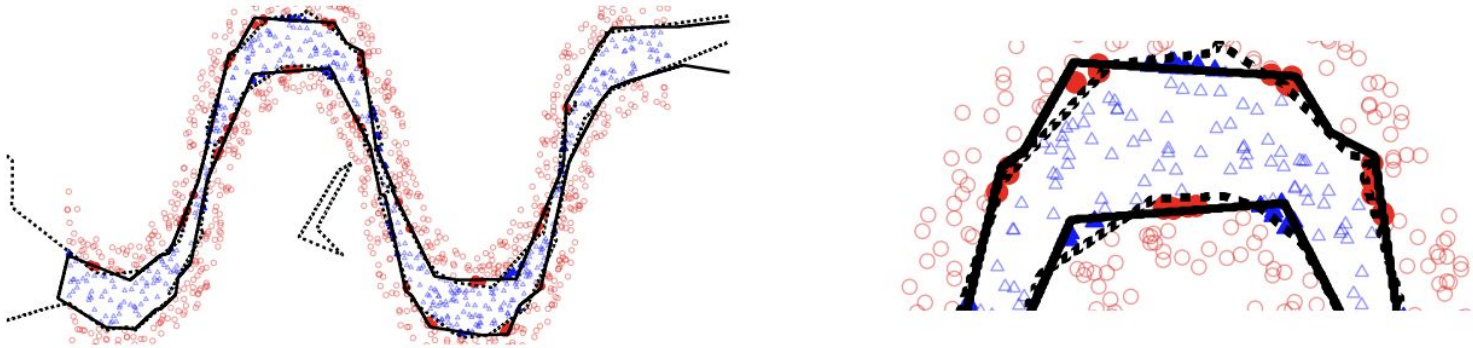
ReLU is not smooth at zero, what are the implications?

May affect GD convergence, but fine in practice. Non-convexity is the bigger issue.

Impact of Depth

1 hid layer,
20 neurons

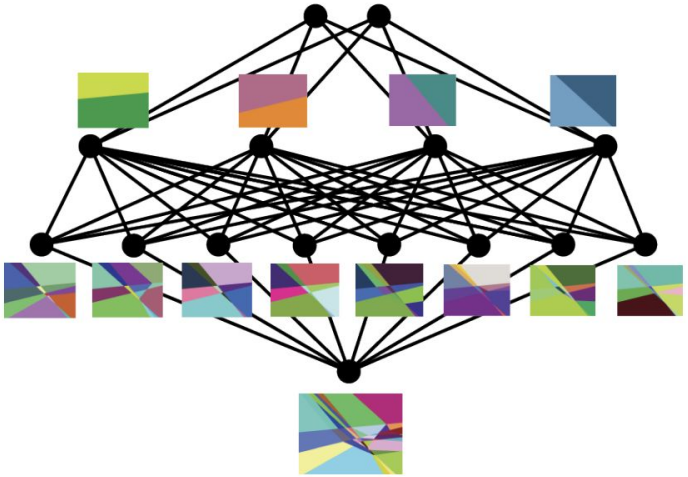
2 hid layer,
10 neurons



Deeper models split the input space into more linear regions.

https://arxiv.org/abs/1402.1869

Impact of Depth



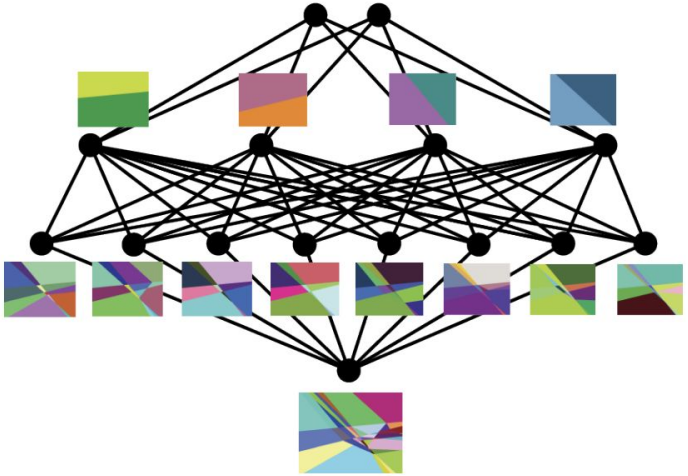
$$\mathrm{ReLU}(x) = \max(0, x)$$

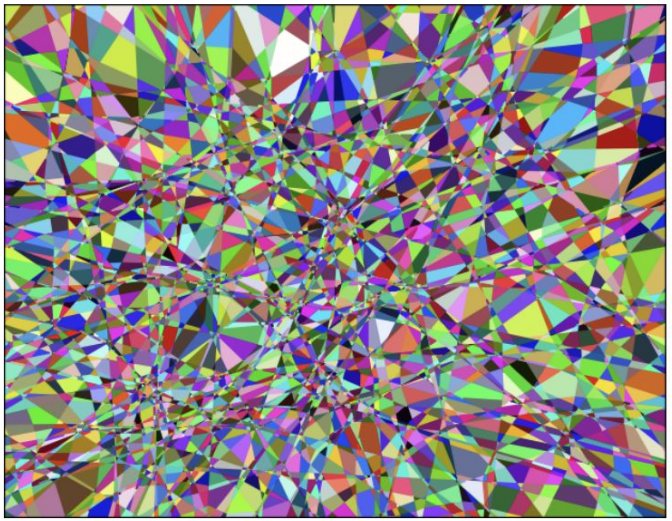
- Activations for neurons in the first layer: 2 linear regions
- *How many regions for second layer activations?*
- Simple upper bound: $2^{\#width+1}$
  - Actually, it's lower but still exponential
- Tight upper bound: $O(width^{input\_dim \cdot depth})$

https://arxiv.org/abs/1901.09021

https://arxiv.org/abs/1606.05336

Impact of Depth





MNIST, 2d slice of input space,

MLP with 3 hidden layers, width 64

https://arxiv.org/abs/1901.09021

Exploding and Vanishing Activations

$$\hat{y} = W_d \sigma(\dots W_1 \sigma(W_0 x))$$

- Let’s consider a deep linear model first

Exploding and Vanishing Activations

$$\hat{y} = W_d \sigma(\ldots W_1 \sigma(W_0 x))$$

$$\hat{y} = W_d \ldots W_1 W_0 x$$

- Let's consider a deep linear model first

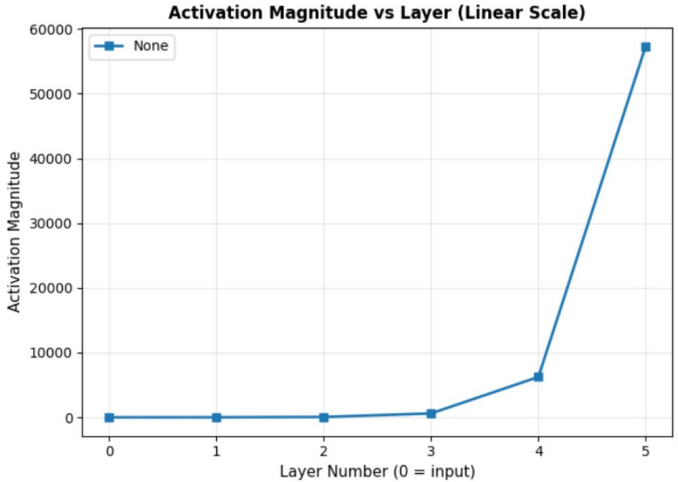
Exploding and Vanishing Activations

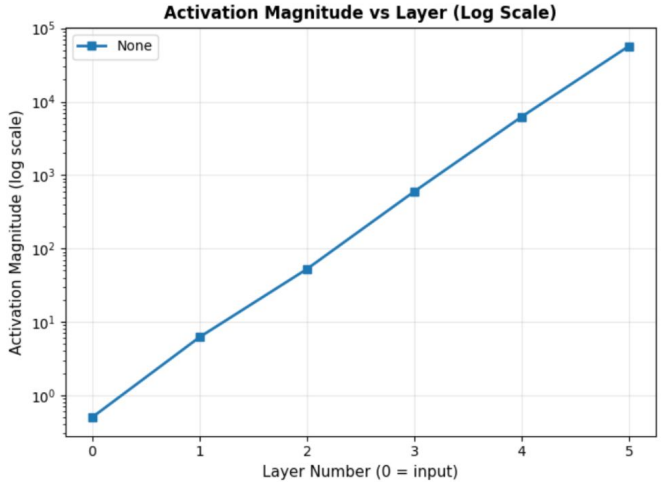
$$\hat{y} = W_d \sigma(\ldots W_1 \sigma(W_0 x))$$

$$\hat{y} = W_d \ldots W_1 W_0 x$$

Consider the case when all weight matrices are scaled identity!

- Let's consider a deep linear model first
- If the singular values of weight matrices are above 1, the activations will explode





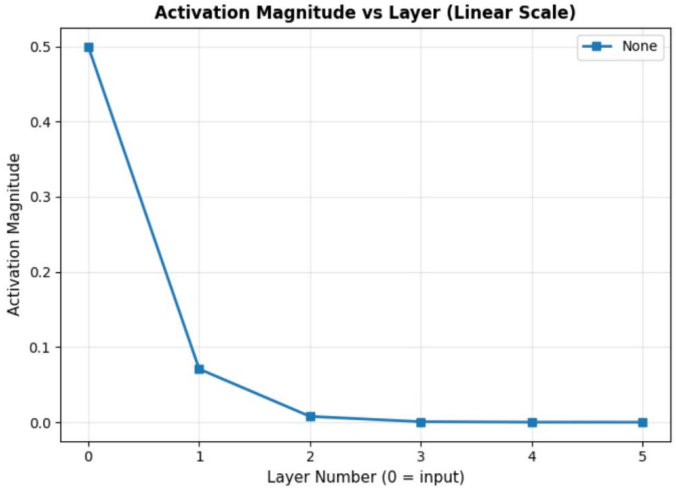
Exploding and Vanishing Activations

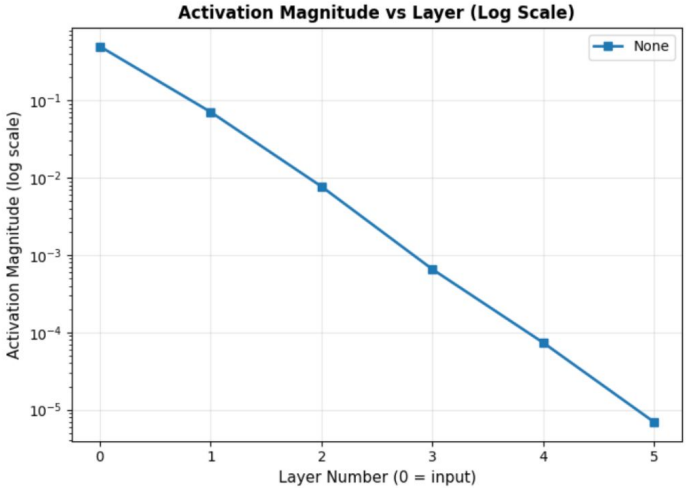
$$\hat{y} = W_d \sigma(\ldots W_1 \sigma(W_0 x))$$

$$\hat{y} = W_d \ldots W_1 W_0 x$$

Consider the case when all weight matrices are scaled identity!

- Let's consider a deep linear model first
- If the singular values of weight matrices are above 1, the activations will explode
- If they are below 1, activations vanish





Exploding and Vanishing Activations

$$\hat{y} = W_d \sigma(\dots W_1 \sigma(W_0 x))$$

$$\hat{y} = W_d \dots W_1 W_0 x$$

- Consider a random weight matrix

$$W \in \mathbb{R}^{n \times m}, \quad W_{ij} \sim \mathcal{N}(0, \sigma^2)$$

- The scales of its singular values are

$$s_i = \Theta(\sigma\sqrt{m})$$

```
torch.nn.init.kaiming_normal_(tensor, a=0, mode='fan_in',
nonlinearity='leaky_relu', generator=None)
```

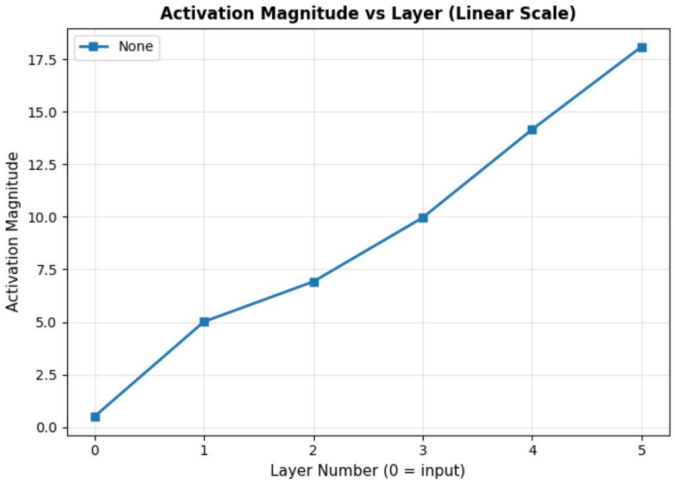
$$\sigma = \frac{1}{\sqrt{\text{fan\_in}}}$$

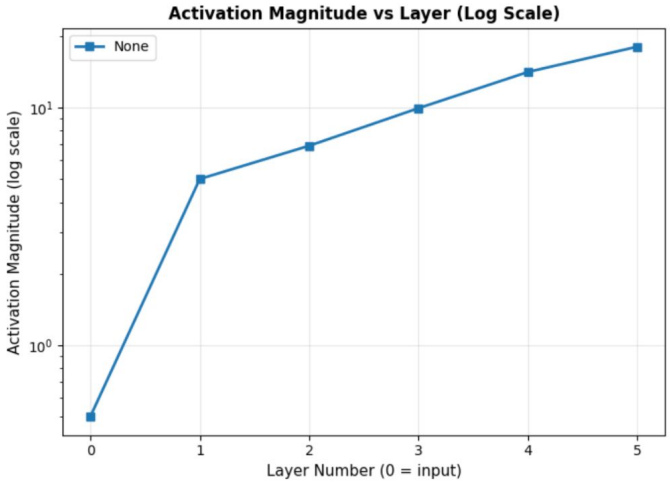
https://arxiv.org/abs/1502.01852

Exploding and Vanishing Activations

```
torch.nn.init.kaiming_normal_(tensor, a=0, mode='fan_in',
nonlinearity='leaky_relu', generator=None)
```

$$\sigma = \frac{1}{\sqrt{\text{fan\_in}}}$$





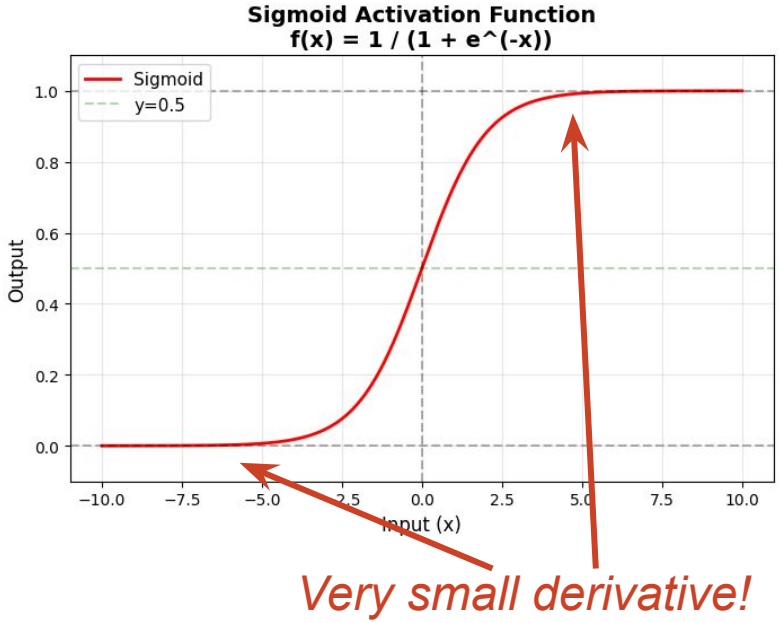
https://arxiv.org/abs/1502.01852

Stable activations!

Vanishing Gradients

$$\hat{y} = W_d \sigma(\ldots W_1 \sigma(W_0 x))$$

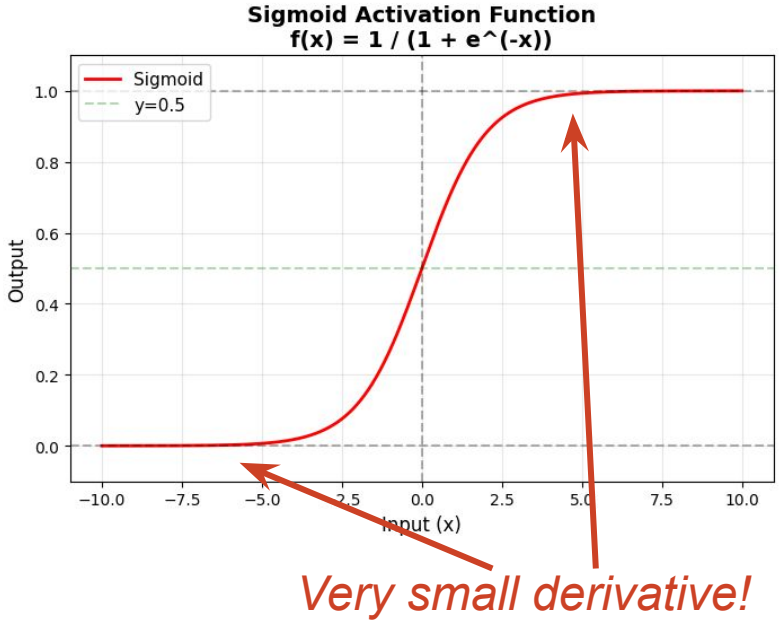
Let's add sigmoids back in.

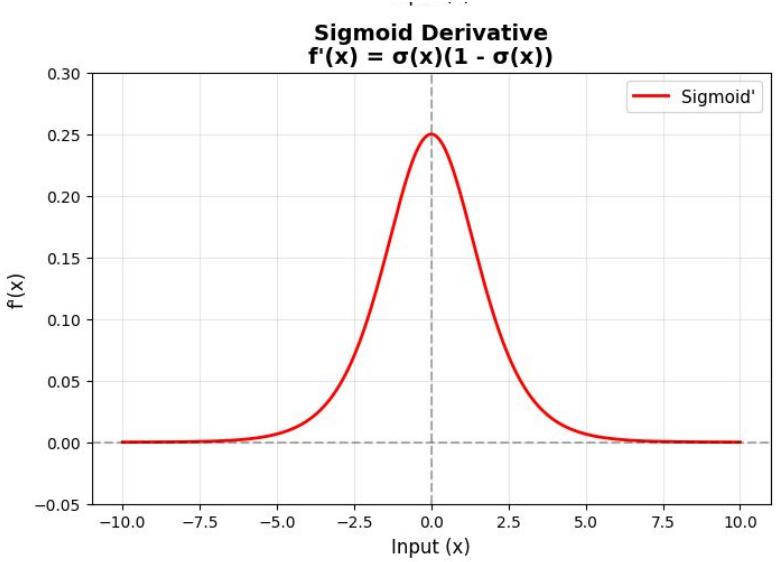


Vanishing Gradients

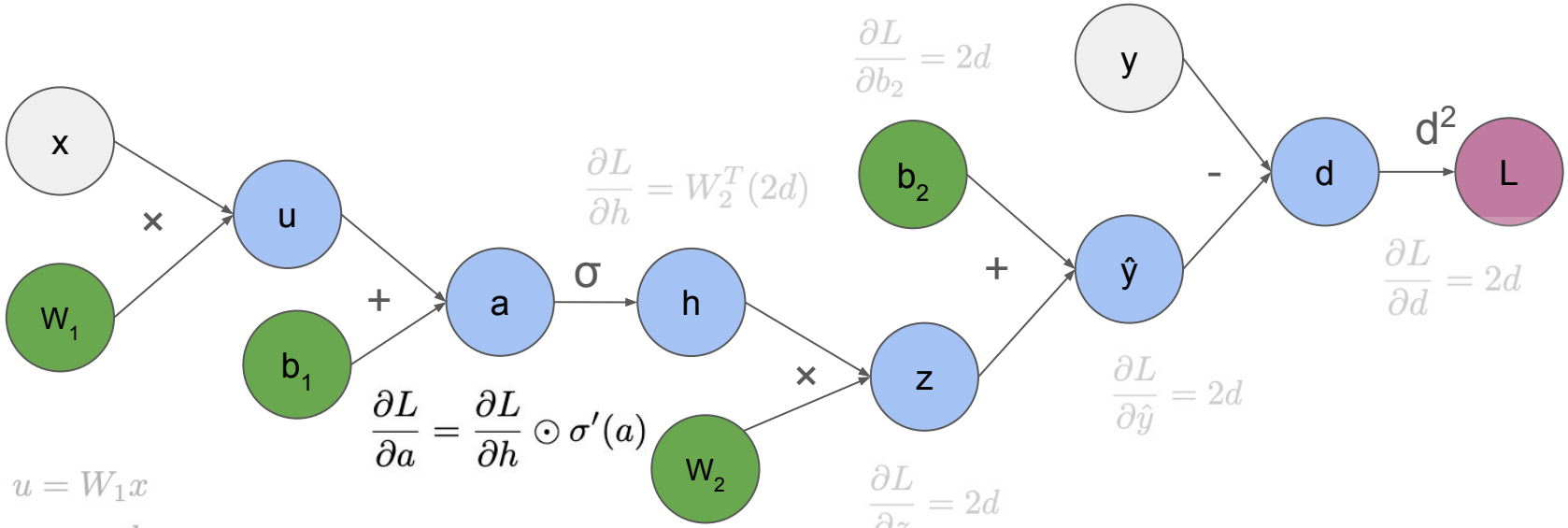
$$\hat{y} = W_d\sigma(\ldots W_1\sigma(W_0x))$$

Let's add sigmoids back in.



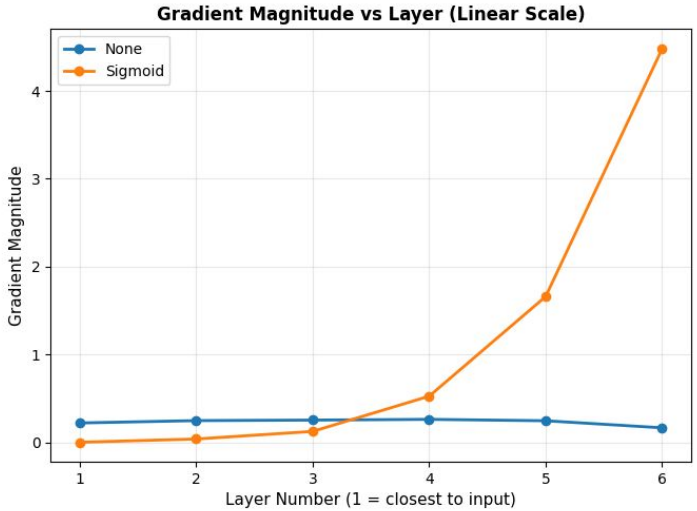


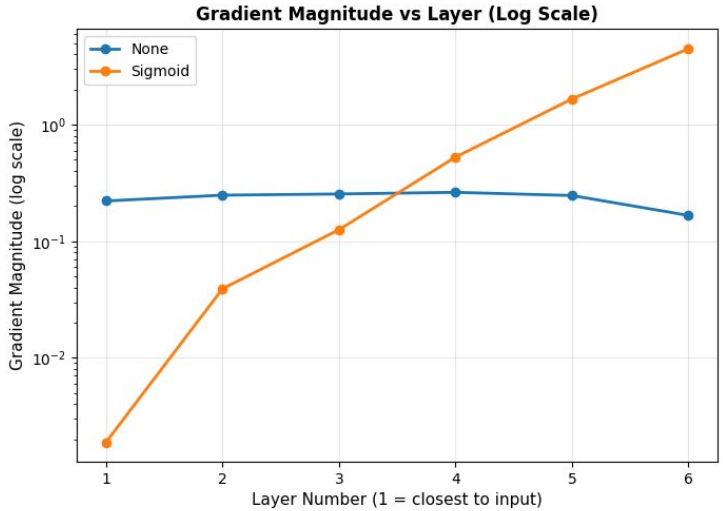
Backprop: MLP



The gradients are scaled by the sigmoid derivative.

Vanishing Gradients

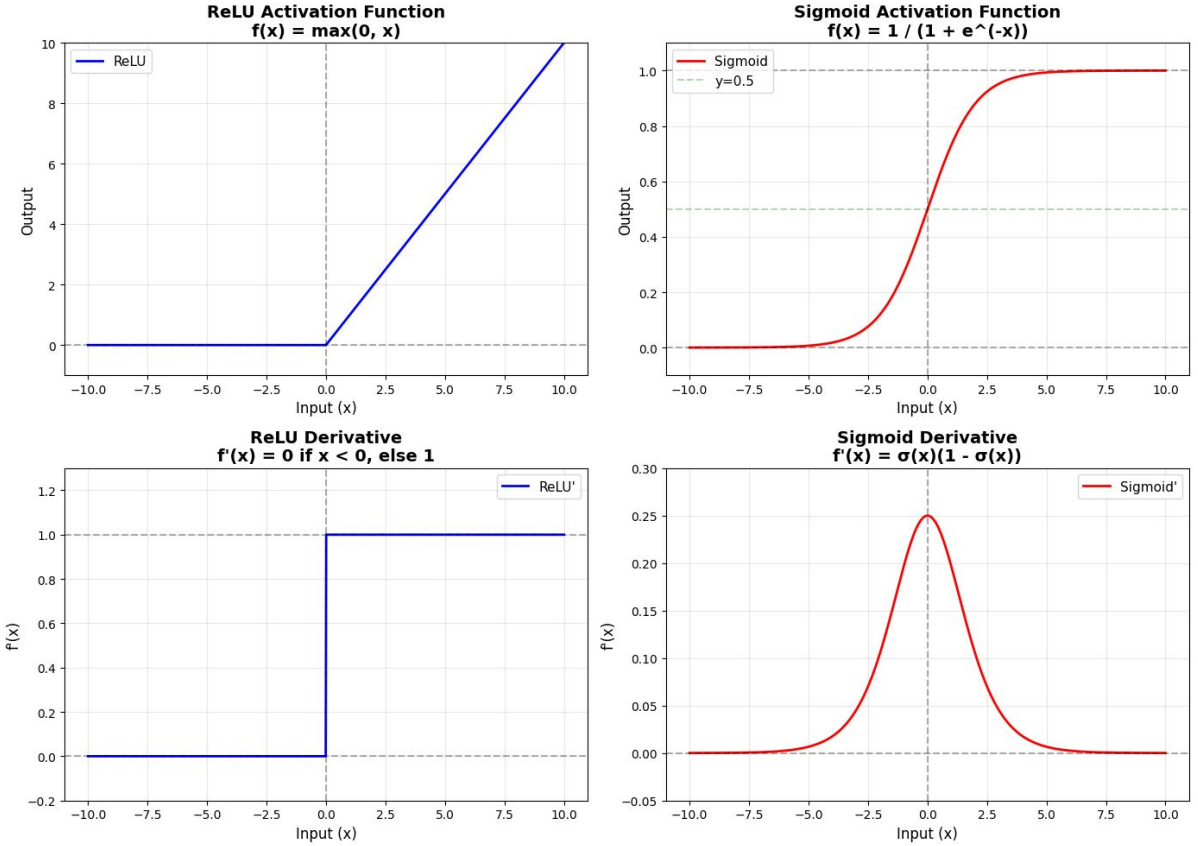




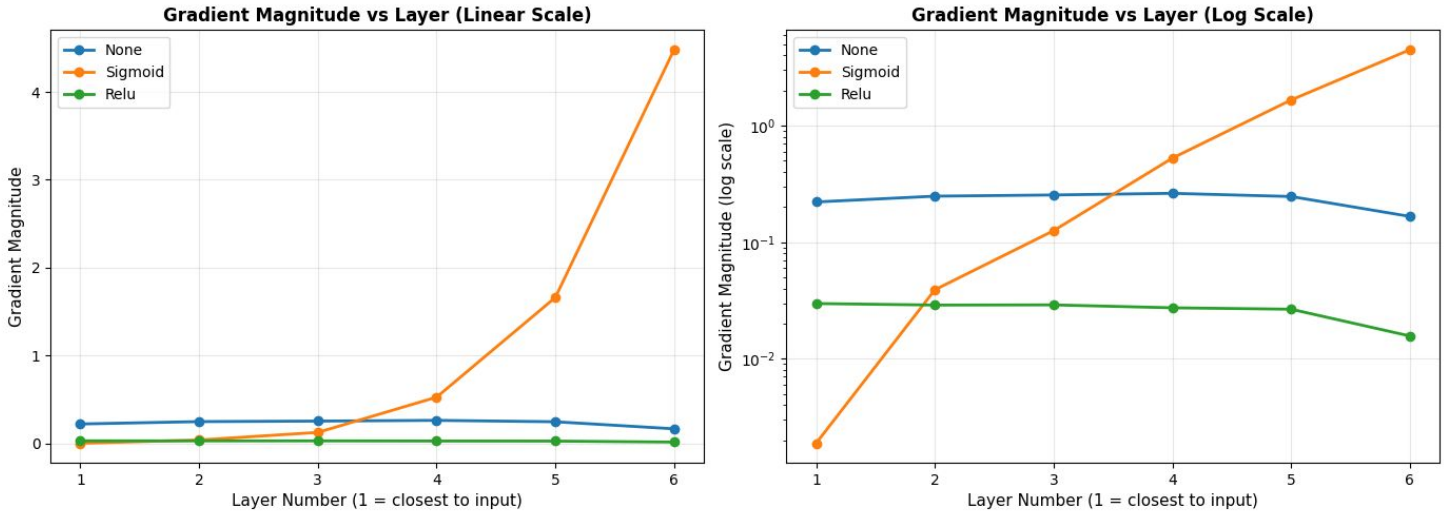
With the sigmoid activation, we get a *vanishing gradient* problem: gradients for early layers are tiny!

Vanishing Gradients

ReLU gradient is a step function.



Vanishing Gradients



ReLU fixes the vanishing gradients in this case!

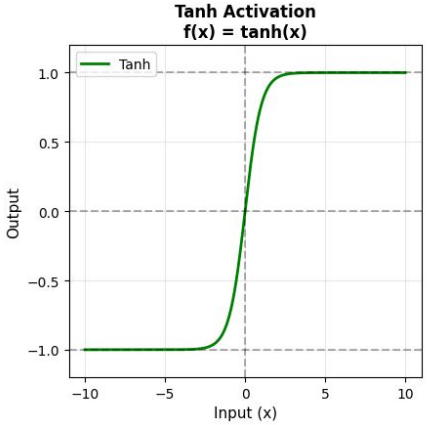
Activation Functions

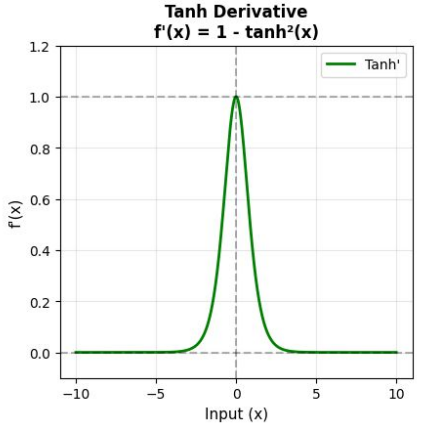
- Tanh is like sigmoid, but (-1, 1)
  - Still vanishing gradients

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

$$\tanh(x) = 2\sigma(2x) - 1$$





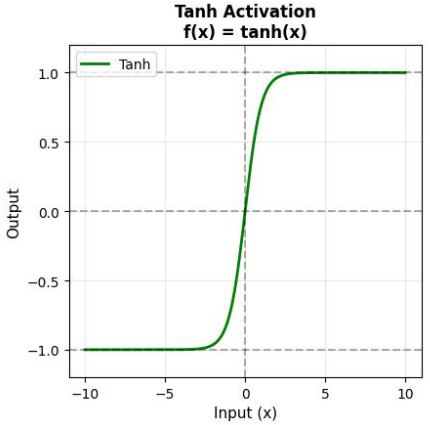
Activation Functions

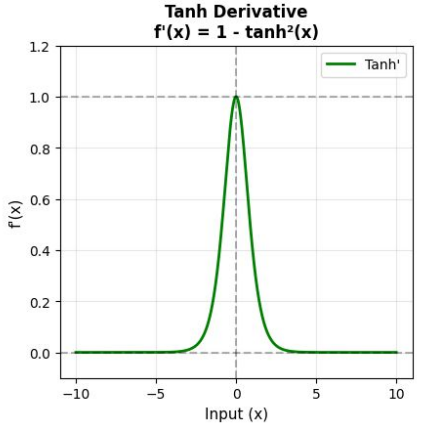
- Tanh is like sigmoid, but (-1, 1)
  - Still vanishing gradients

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

$$\tanh(x) = 2\sigma(2x) - 1$$





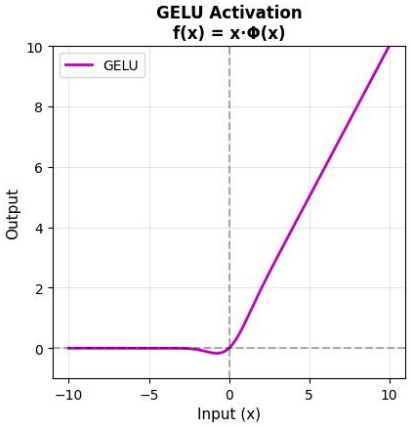
Activation Functions

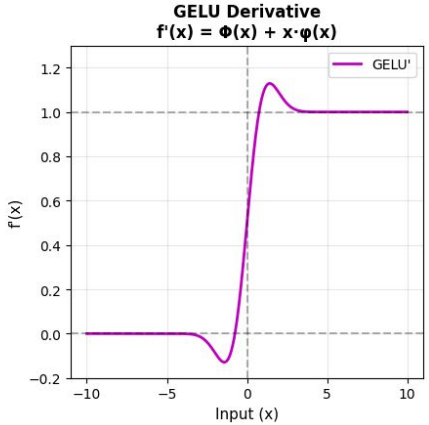
- Tanh is like sigmoid, but (-1, 1)
  - Still vanishing gradients
- GeLU is like ReLU but smooth
  - No vanishing gradients, used in transformers

$$\text{GELU}(x) = \frac{x}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

$$\text{GELU}(x) \approx x \cdot \sigma(1.702x)$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$





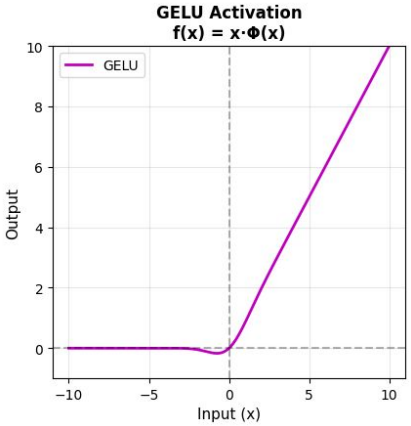
Skip Connections

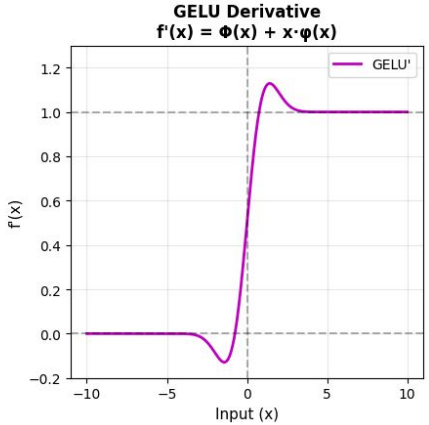
- Tanh is like sigmoid, but (-1, 1)
  - Still vanishing gradients
- GeLU is like ReLU but smooth
  - No vanishing gradients, used in transformers

$$\text{GELU}(x) = \frac{x}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

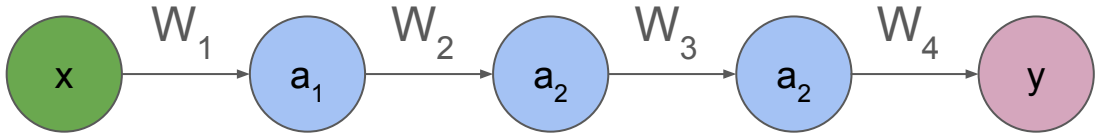
$$\text{GELU}(x) \approx x \cdot \sigma(1.702x)$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$





Skip Connections

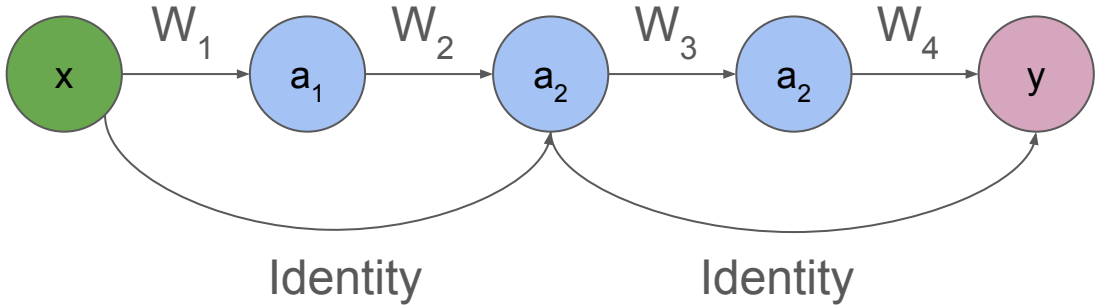


- The model learns an arbitrary linear mapping in each layer
- Easy to forget information in early layers, no way to recover!

$$a_{n+1} = \alpha(W_{n+1}\alpha(W_n a_{n-1} + b_n) + b_{n+1})$$

https://arxiv.org/abs/1512.03385

Skip Connections

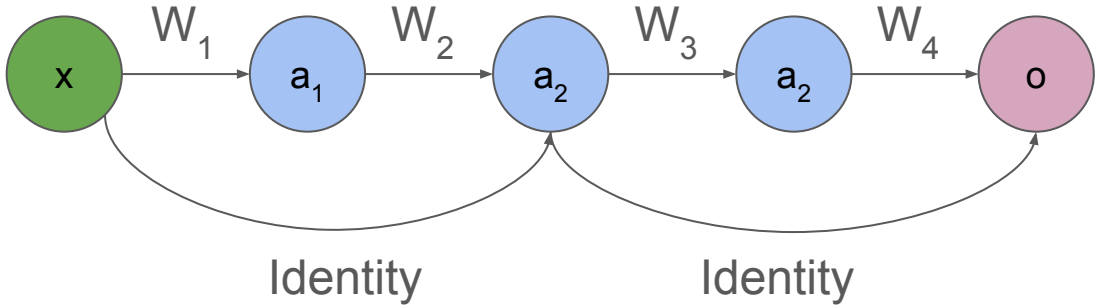


- The model learns an arbitrary linear mapping in each layer
- Easy to forget information in early layers, no way to recover!
- Idea: residual connections

$$a_{n+1} = \alpha(W_{n+1}\alpha(W_n a_{n-1} + b_n) + b_{n+1})$$
$$a_n = x + \alpha(W_{n+1}\alpha(W_n x + b_n) + b_{n+1})$$

https://arxiv.org/abs/1512.03385

Skip Connections



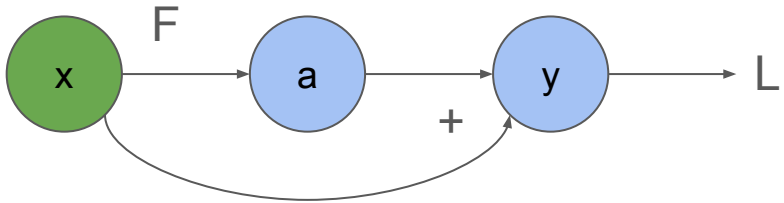
Makes it easy to learn identity transform!

- The model learns an arbitrary linear mapping in each layer
- Easy to forget information in early layers, no way to recover!
- Idea: residual connections

$$a_{n+1} = \alpha(W_{n+1}\alpha(W_n a_{n-1} + b_n) + b_{n+1})$$
$$a_n = x + \alpha(W_{n+1}\alpha(W_n x + b_n) + b_{n+1})$$

https://arxiv.org/abs/1512.03385

Skip Connections



$$y = F(x, \{W_i\}) + x$$

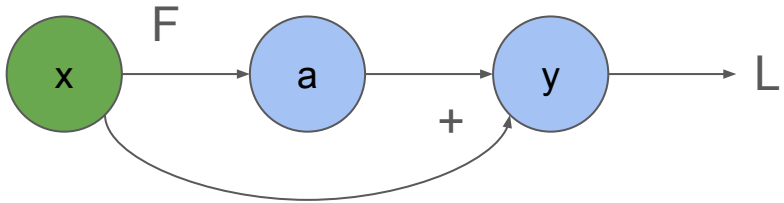
$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial F(x)}{\partial x} + \frac{\partial \mathcal{L}}{\partial y}$$

Gradient highway!

- Skip connections create *gradient highways*
- Avoid vanishing gradients
- Allowed to train 1000-layer models for the first time

https://arxiv.org/abs/1512.03385

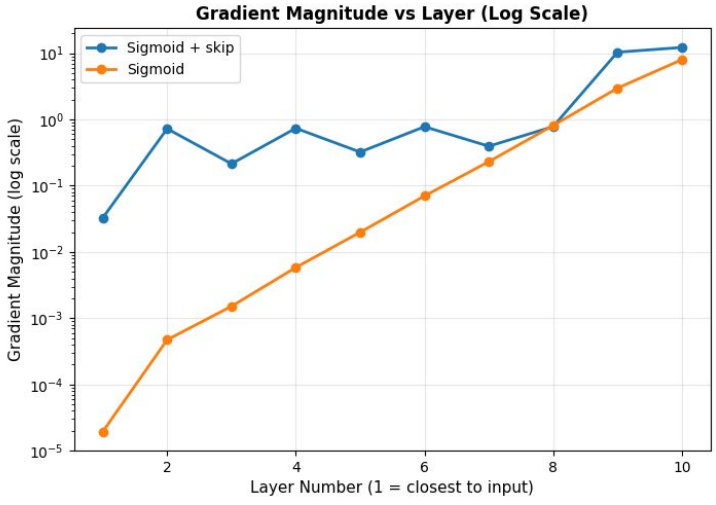
Skip Connections



$$y = F(x, \{W_i\}) + x$$

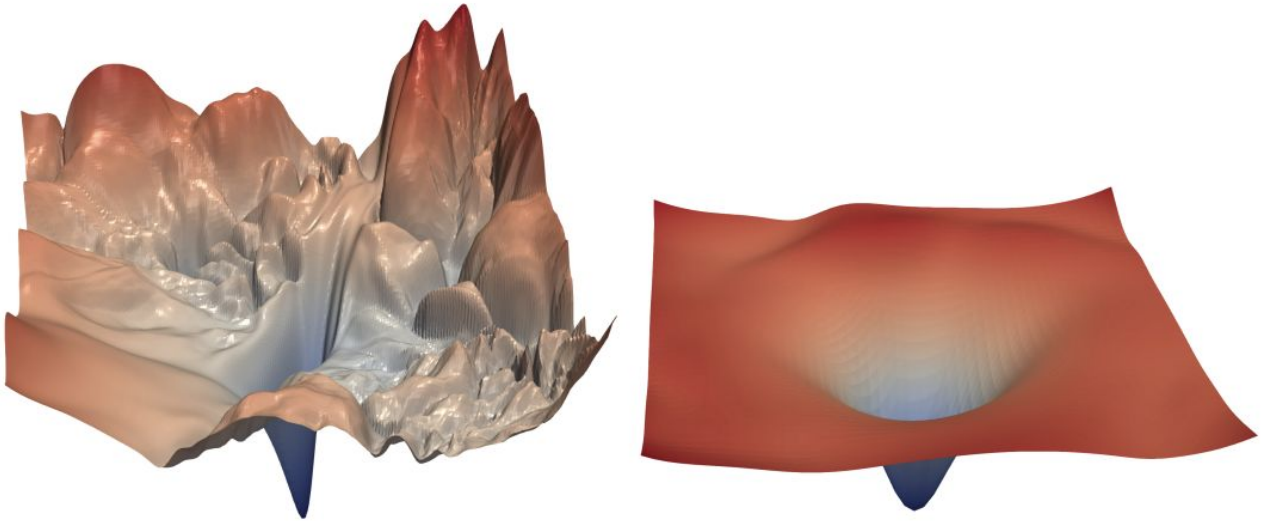
$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial F(x)}{\partial x} + \frac{\partial \mathcal{L}}{\partial y}$$

- Skip connections create *gradient highways*
- Avoid vanishing gradients
- Allowed to train 1000-layer models for the first time



https://arxiv.org/abs/1512.03385

Skip Connections



No skip connections

With skip connections

https://arxiv.org/abs/1712.09913

Normalization layers

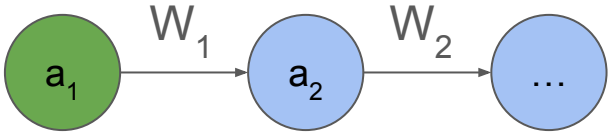
- Input to a layer is the output of previous layer

$$W_2\alpha(W_1a_1 + b_1) + b_2$$

- If previous layer changes the input to current layer can change dramatically

$$W_2\alpha(W_1 \cdot 10\ a_1 + b_1 \cdot 10) + b_2$$

- *Internal covariate shift:* input distribution to deeper layers keeps changing in training



Normalization layers

- Input to a layer is the output of previous layer

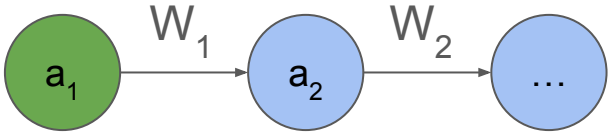
$$W_2\alpha(W_1a_1 + b_1) + b_2$$

- If previous layer changes the input to current layer can change dramatically

$$W_2\alpha(W_1 \cdot 10\ a_1 + b_1 \cdot 10) + b_2$$

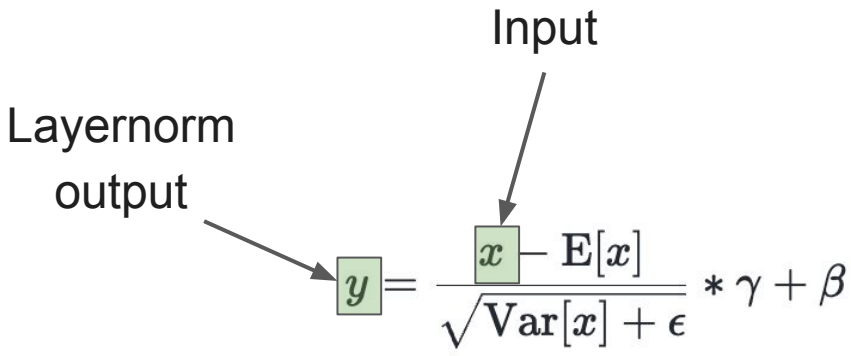
- *Internal covariate shift:* input distribution to deeper layers keeps changing in training

$$W_2\alpha(\text{normalize}(W_1 \cdot 10\ a_1 + b_1 \cdot 10)) + b_2$$



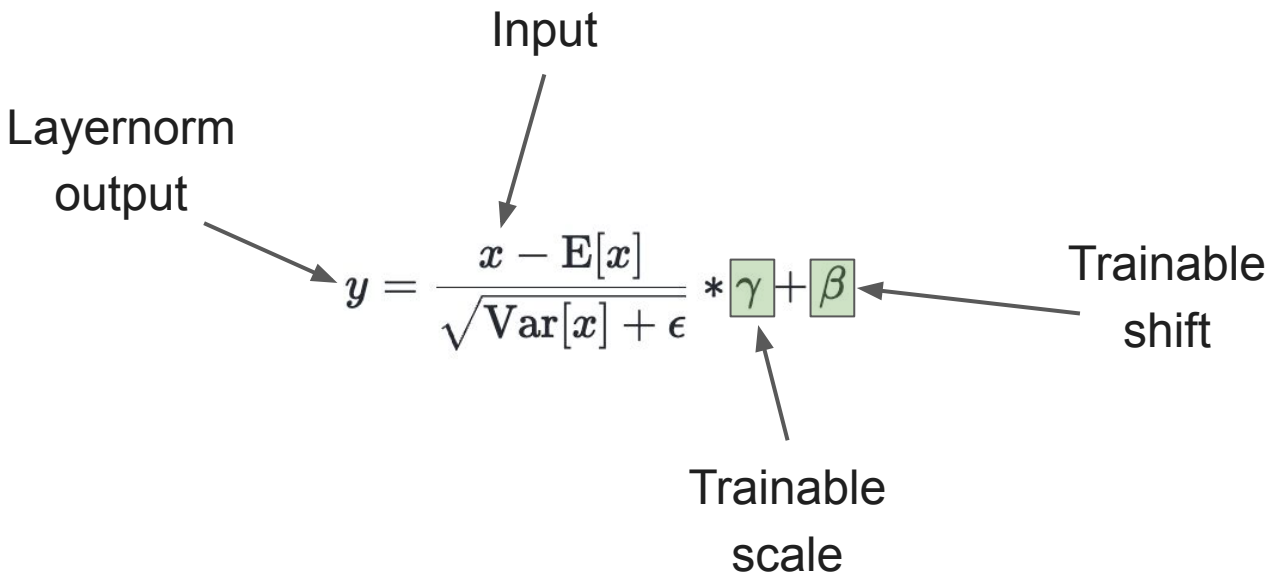
Idea: let's renormalize the input to the layer!

Normalization layers



https://arxiv.org/abs/1502.03167

Normalization layers



https://arxiv.org/abs/1502.03167

Normalization layers

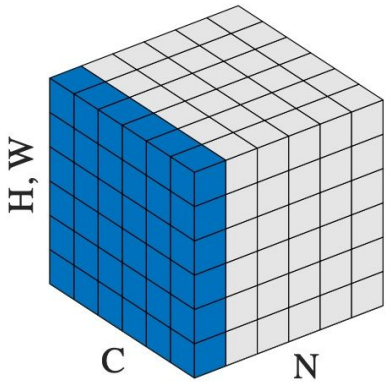
Input

Layernorm output

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Trainable shift

Trainable scale



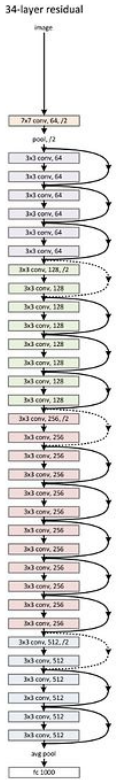
- Mean and variance are computed over all dimensions of the input

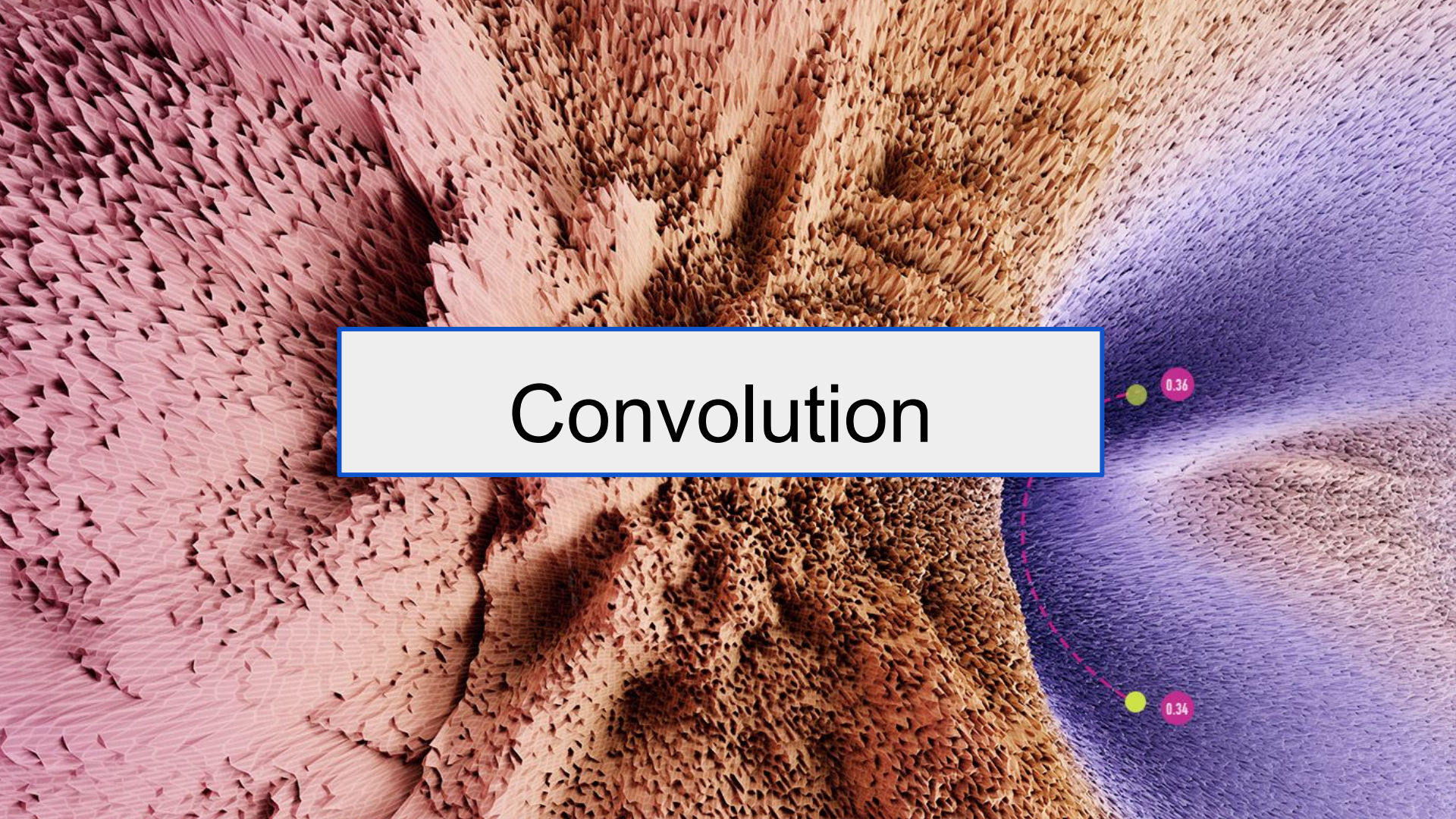
https://arxiv.org/abs/1502.03167

Summary

- Deeper models do better
- But they are harder to train
  - Exploding and vanishing activations
  - Vanishing gradients
  - Hard to learn identity
  - Internal covariate shift
- Ideas
  - Better initialization
  - ReLU / GeLU activations
  - Skip connections
  - Normalization

All of these ideas were needed before we could train very deep models!

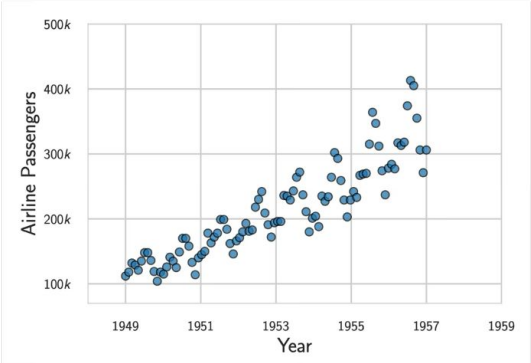



Convolution
0.36
0.34

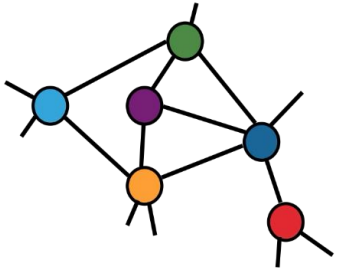
Convolutions

Images

Time-Series

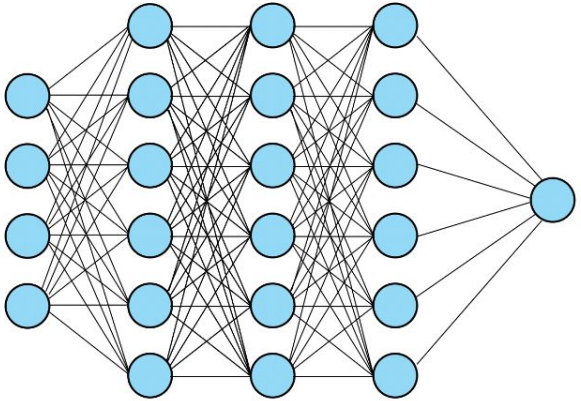


Graphs

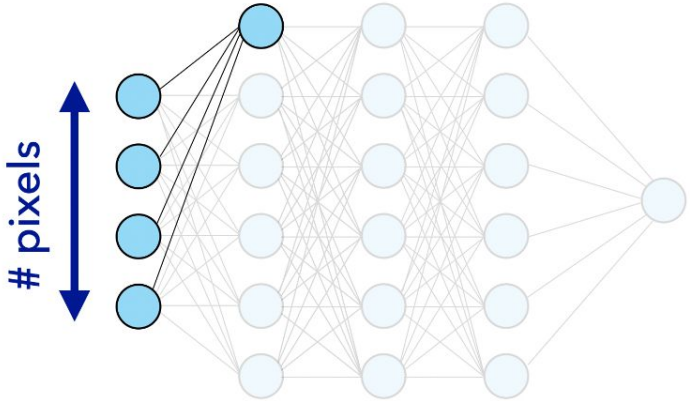


- Convolution is a fundamental idea applicable to many domains
- We will focus on images today

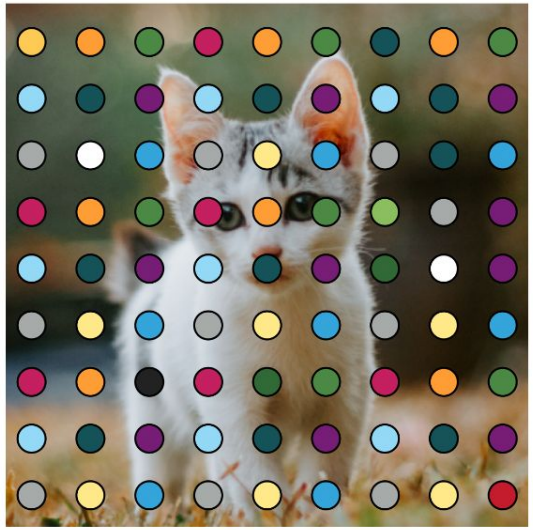
Convolutions

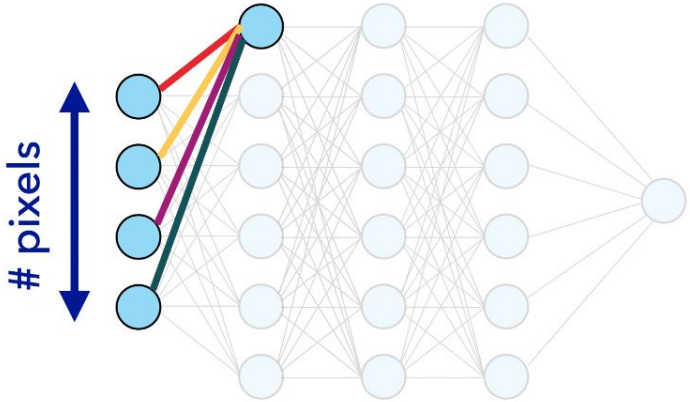


Convolutions

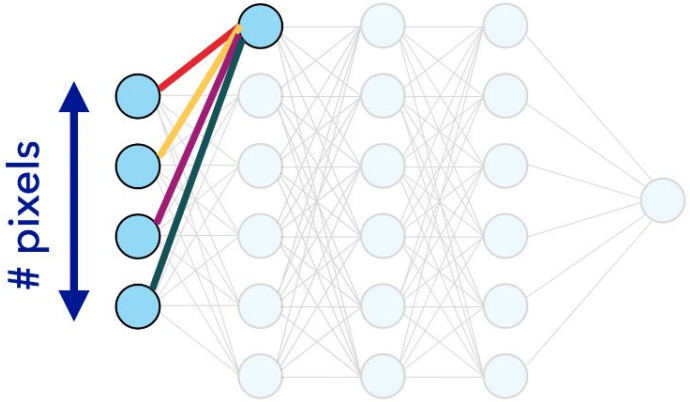


Convolutions

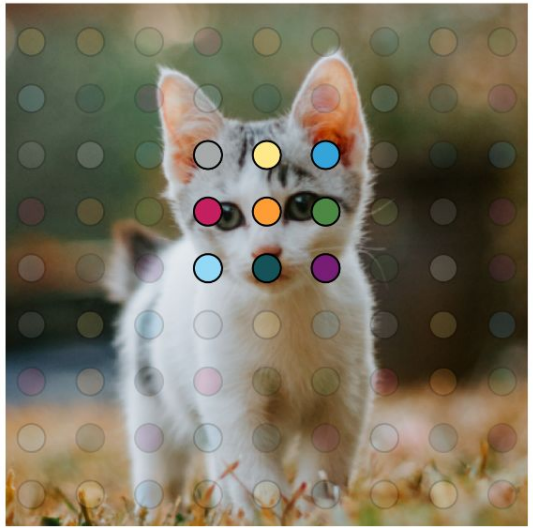




Convolutions

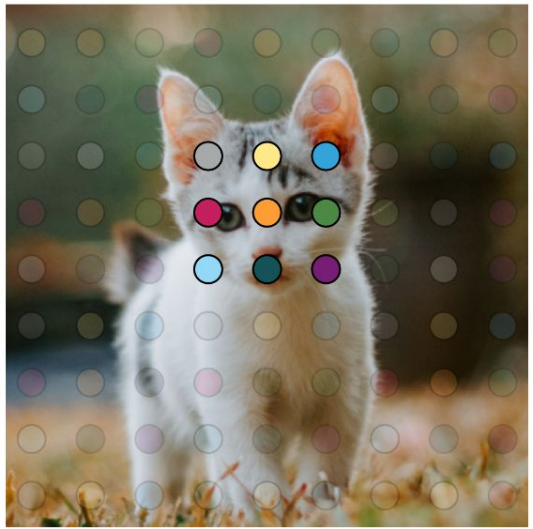


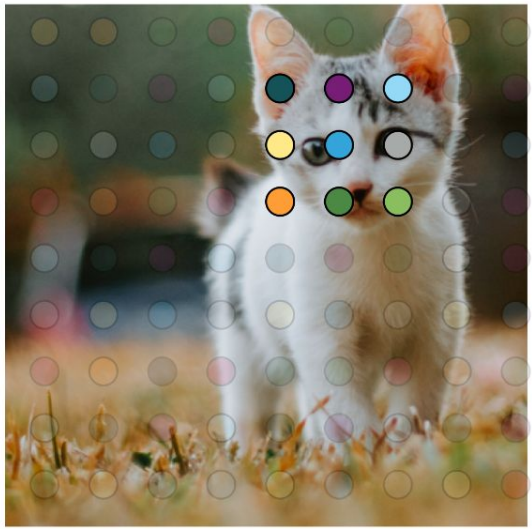
Convolutions



Convolutions

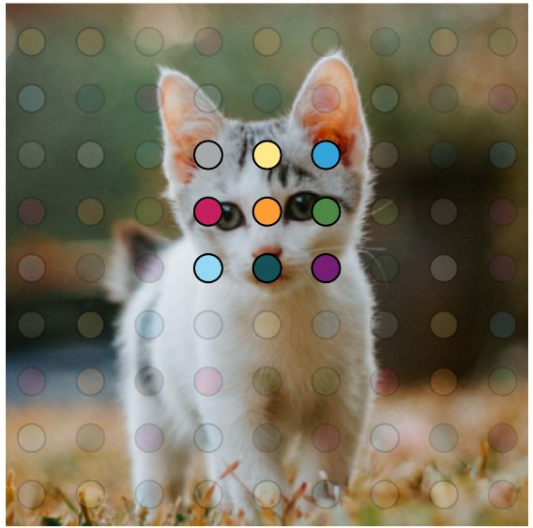
We have to learn the same  face detector in each location independently!





Convolutions

We have to learn the same face detector in each location independently!

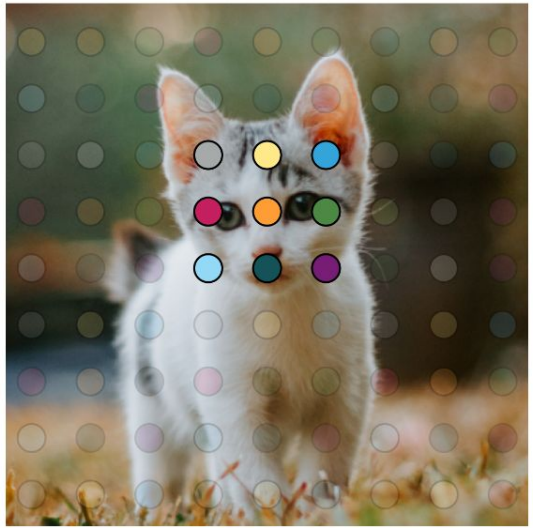


Data inefficient

Parameter inefficient

Convolutions

We have to learn the same face detector in each location independently!



Data inefficient

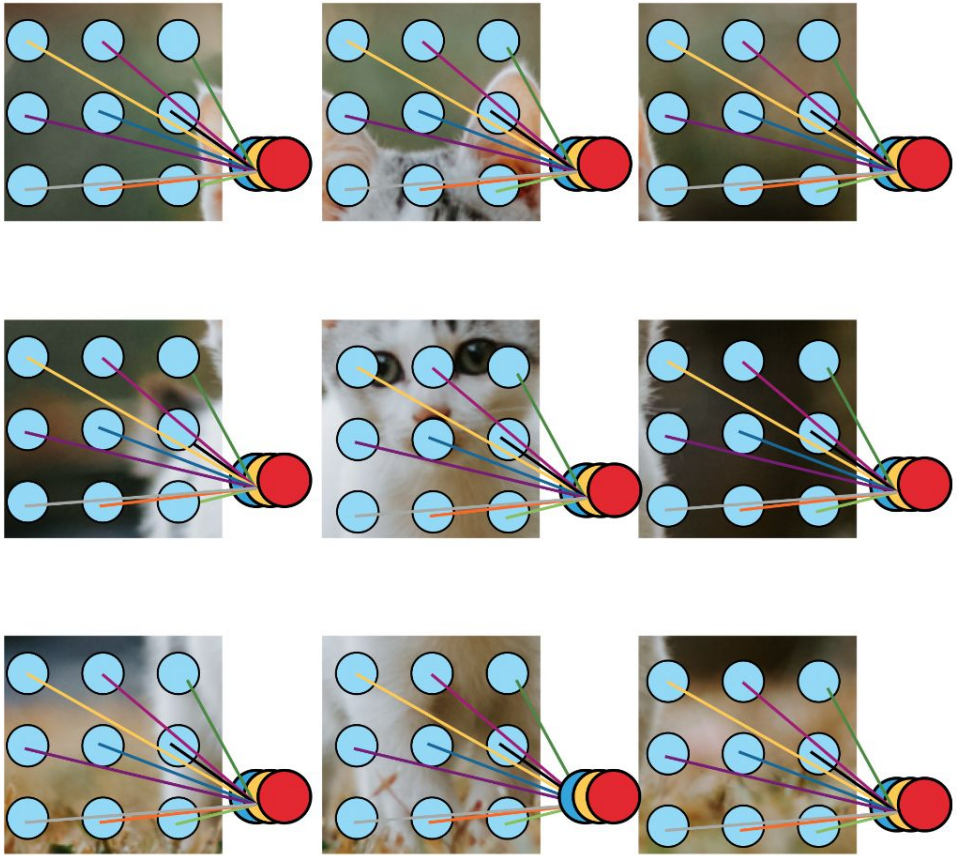
Parameter inefficient

Convolutions

Convolutions

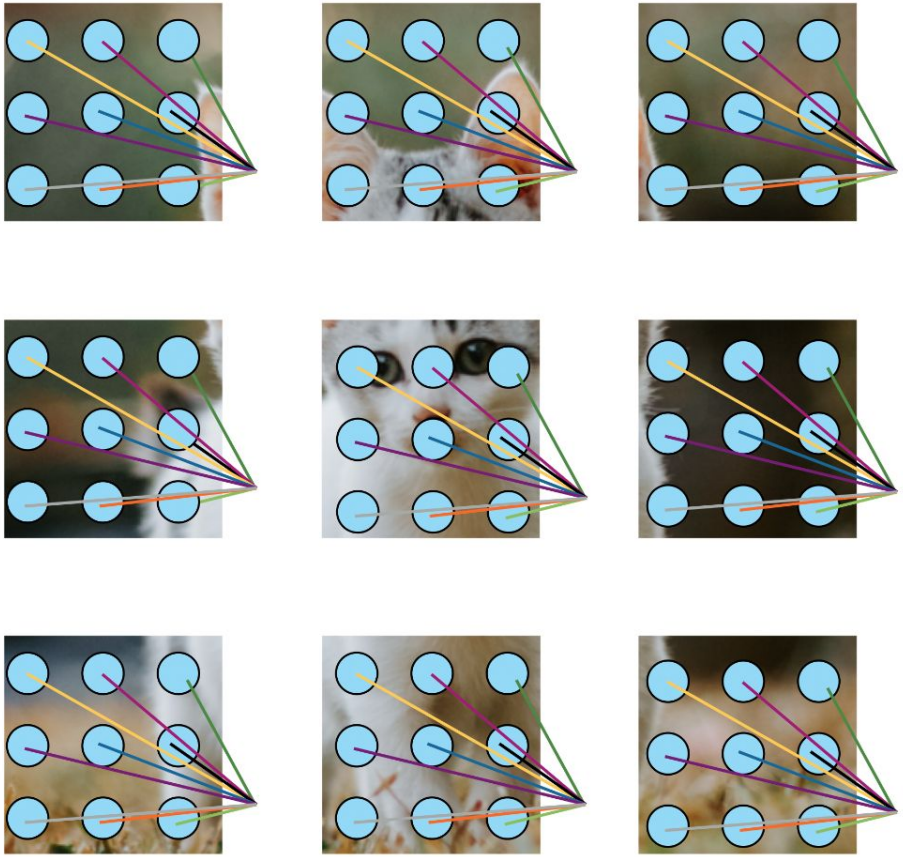
- Split the image in patches

Convolutions

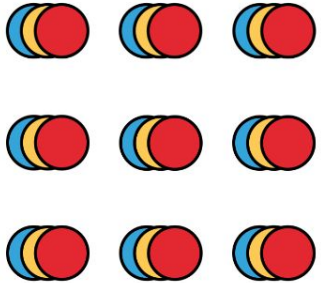


- Split the image in patches
- Apply the same fully-connected layer to each patch

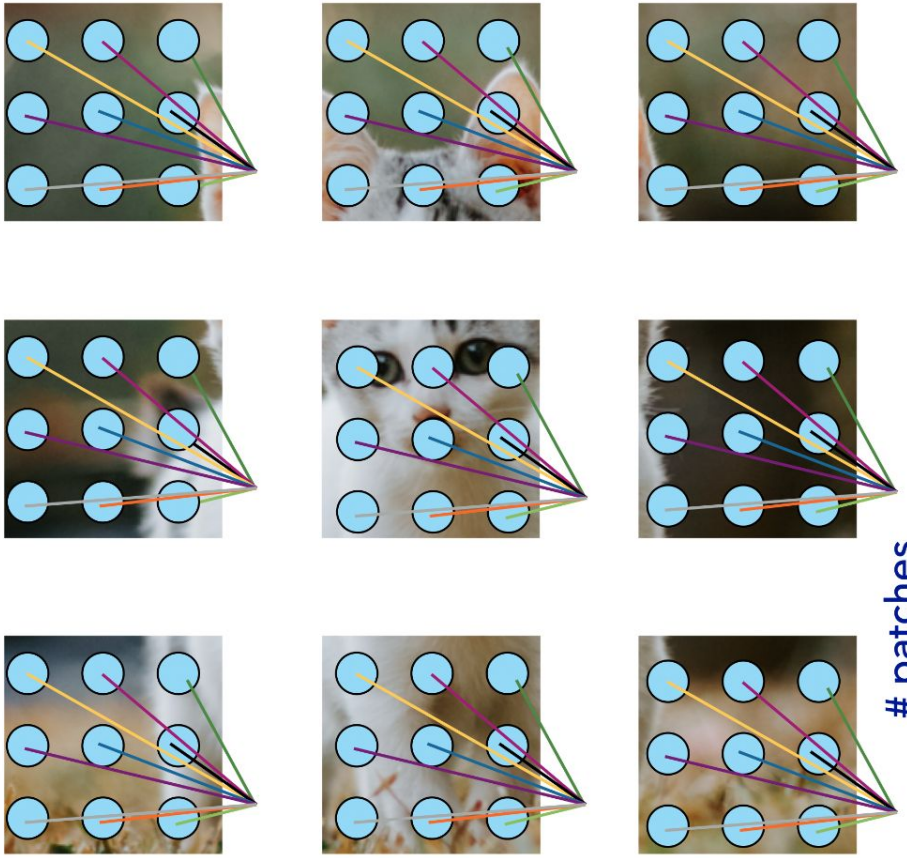
Convolutions



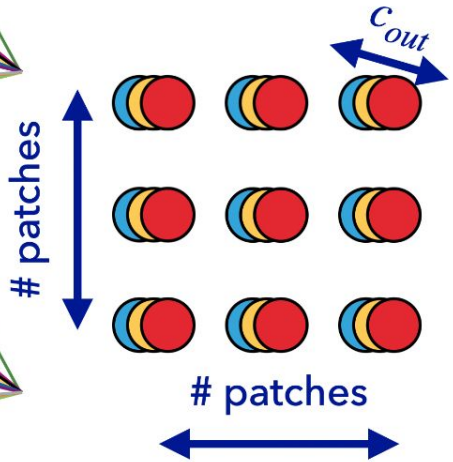
- Split the image in patches
- Apply the same fully-connected layer to each patch
- Collect the outputs



Convolutions



- Split the image in patches
- Apply the same fully-connected layer to each patch
- Collect the outputs

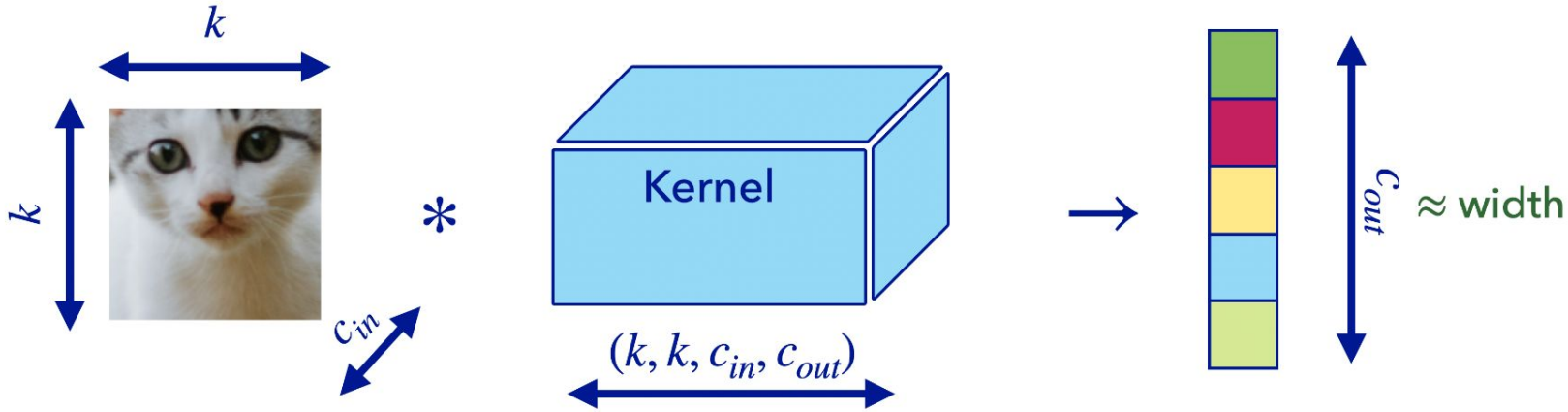


Output is an image

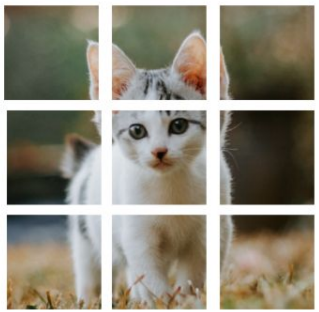
⇓

Can stack!

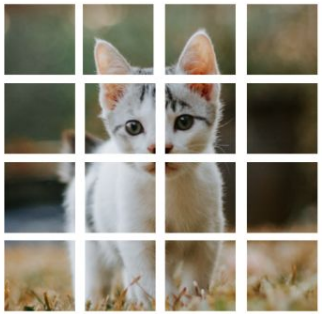
Convolutions



Larger k

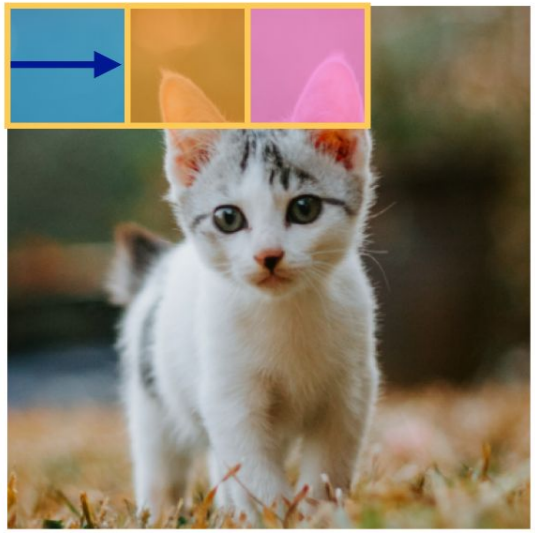


Smaller k



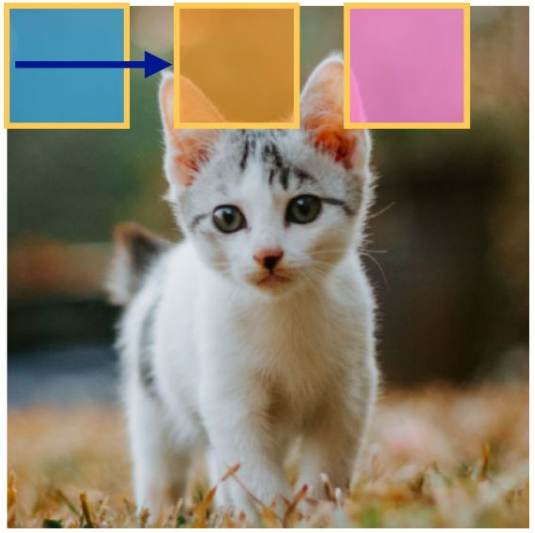
Convolutions: stride

Convolutions: stride



Smaller stride

Larger stride



Convolutions: stride

Fully-connected

```
fc_net = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28**2, 100),
    nn.ReLU(),
    nn.Linear(100, 100),
    nn.ReLU(),
    nn.Linear(100, 100),
    nn.ReLU(),
    nn.Linear(100, 10)
)
```

Params: 100k, Acc: 94.5%

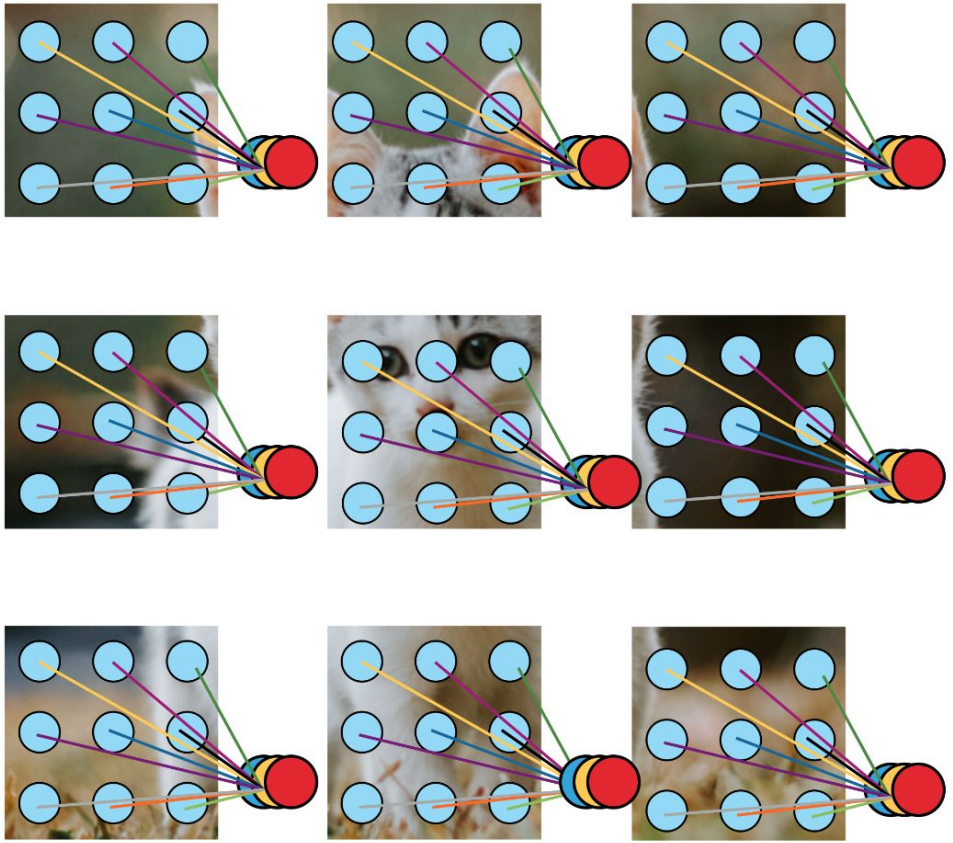
Convolutional

```
conv_net = nn.Sequential(
    nn.Conv2d(in_channels=1,
              out_channels=32,
              kernel_size=5,
              stride=2),
    nn.ReLU(),
    nn.Conv2d(in_channels=32,
              out_channels=32,
              kernel_size=5,
              stride=2),
    nn.ReLU()
    nn.Flatten(),
    nn.Linear(512, 10)
)
```

Params: 32k, Acc: 97.2%

Convolutions: stride

- Shared fully-connected layer applied to patches
- Better parameter efficiency and data efficiency
- Better performance



HW diagram