
# Deep Learning Essentials

## 6. Activation, Normalization and Regularization

Activation Functions, BatchNorm, Dropout, weight decay

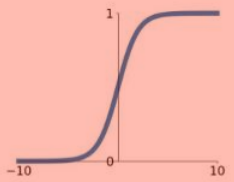
Lukáš Neumann

Adapted from B3B33UROB slides of Karel Zimmerman

# Activation Functions

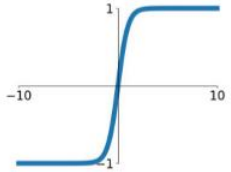
**Sigmoid**

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



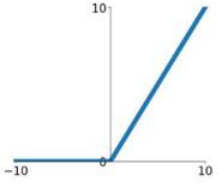

**tanh**

$$\tanh(x)$$



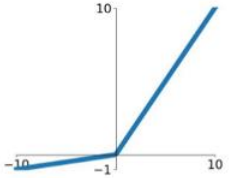

**ReLU**

$$\max(0, x)$$




**Leaky ReLU**

$$\max(0.1x, x)$$

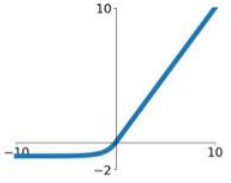



**Maxout**

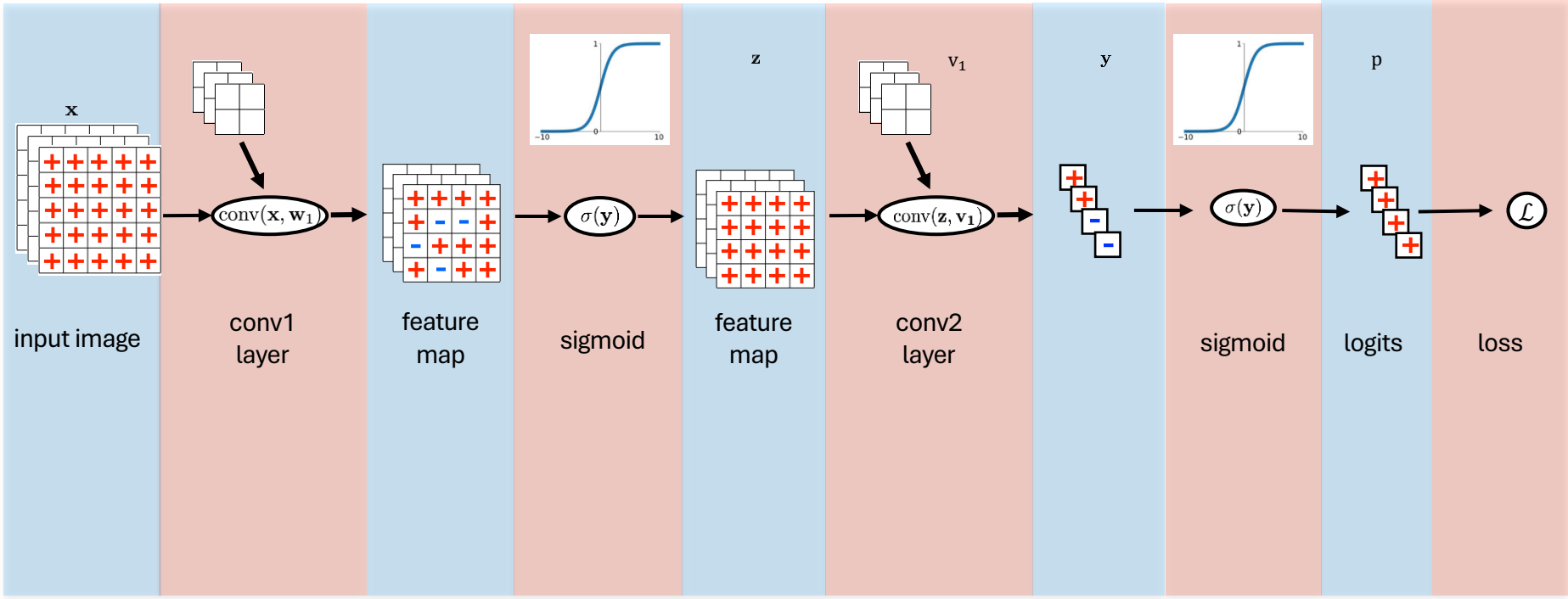
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

**ELU**

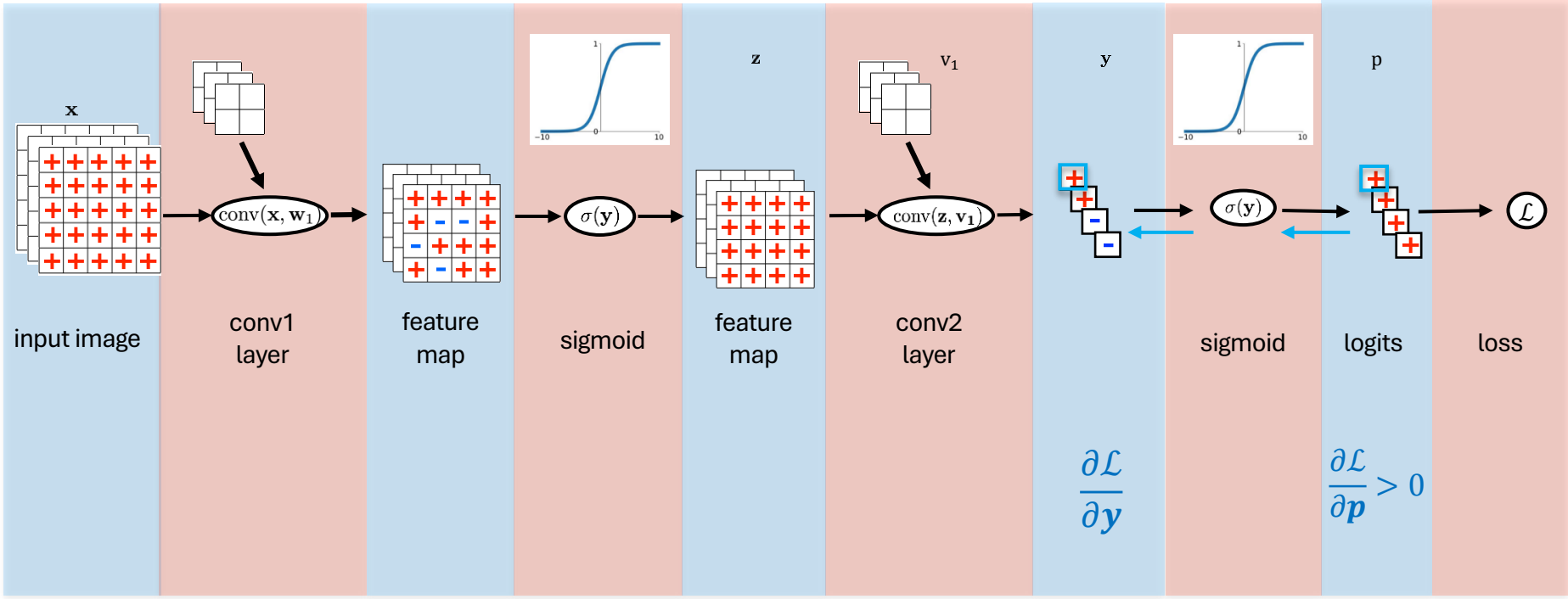
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



# Sigmoid


x
conv(x, w1)
conv1 layer
feature map
sigmoid
σ(y)
z
feature map
v1
conv(z, v1)
conv2 layer
y
σ(y)
sigmoid
p
logits
loss

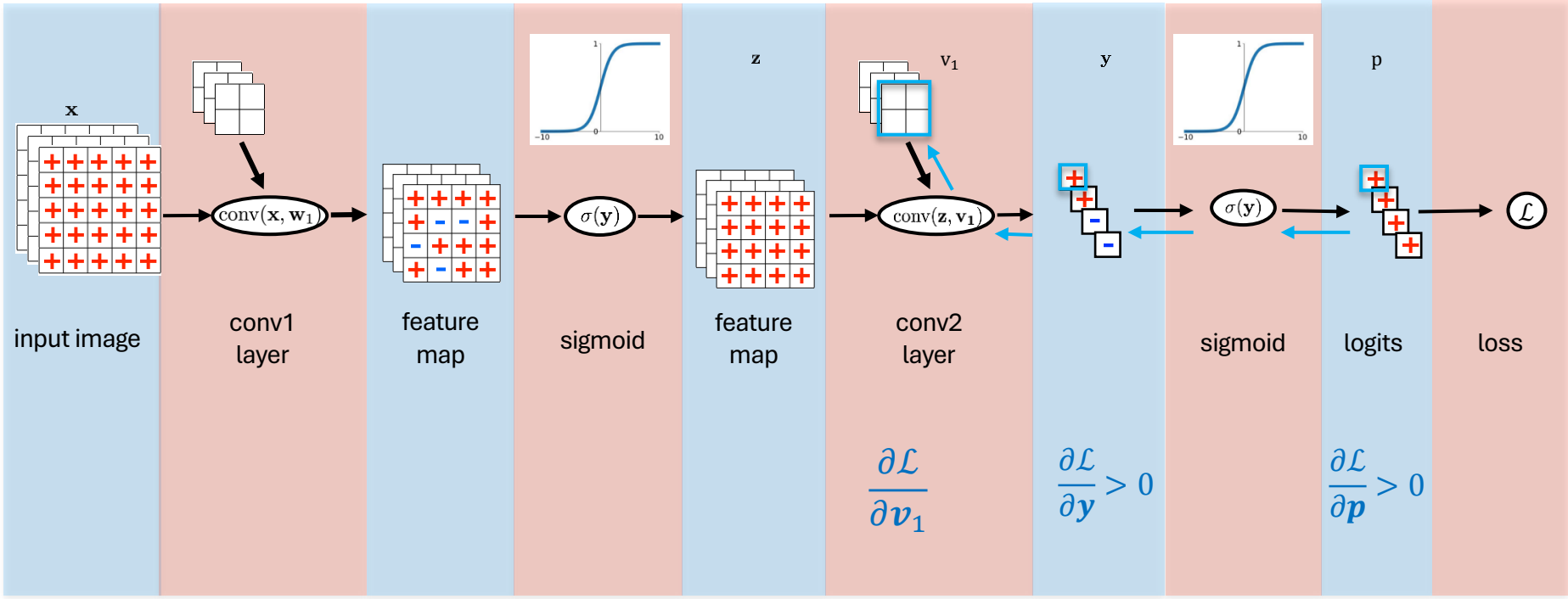
# Sigmoid




$$\boxed{\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}}} = \frac{\partial \boldsymbol{p}}{\partial \boldsymbol{y}} \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{p}} = \boxed{\frac{\partial \sigma(\mathbf{y})}{\partial \mathbf{y}}} \cdot \boxed{\frac{\partial \mathcal{L}}{\partial \boldsymbol{p}}}$$

$> 0$ $> 0$ $> 0$

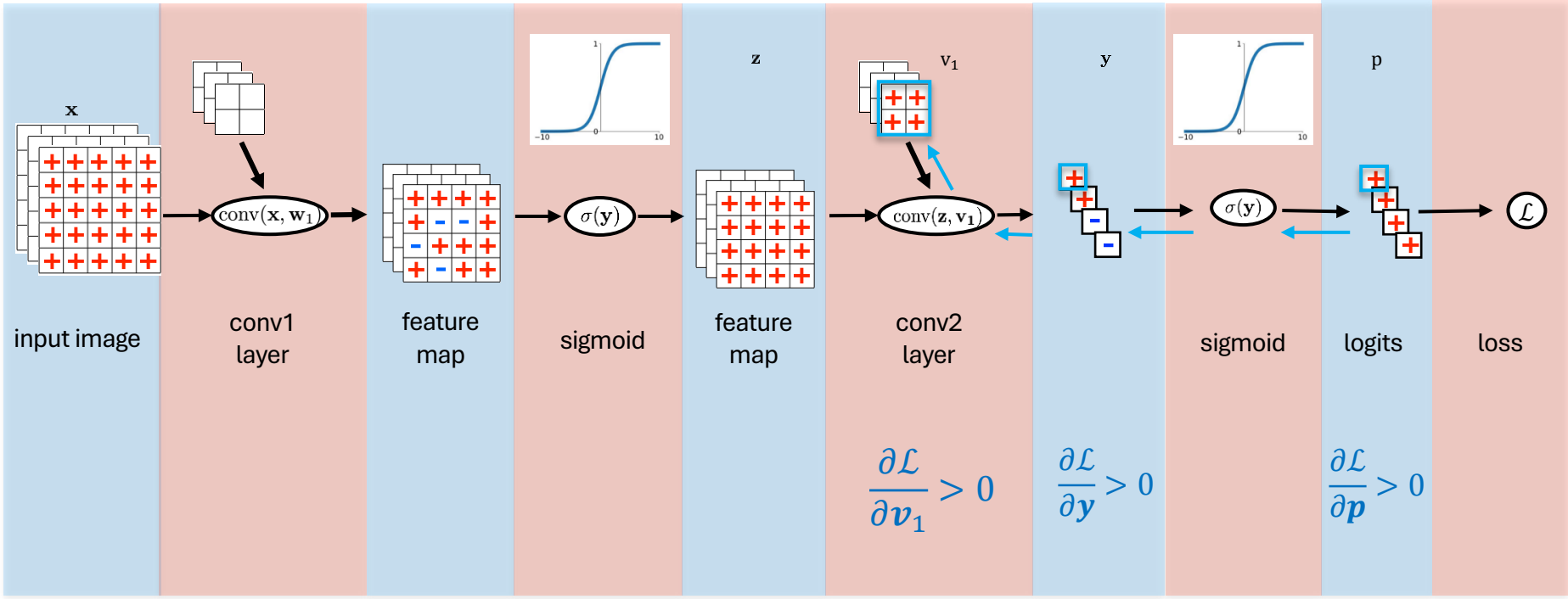
# Sigmoid




$$\frac{\partial \mathcal{L}}{\partial \mathbf{v}_1} \approx \text{conv}\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}}, \mathbf{z}\right)$$

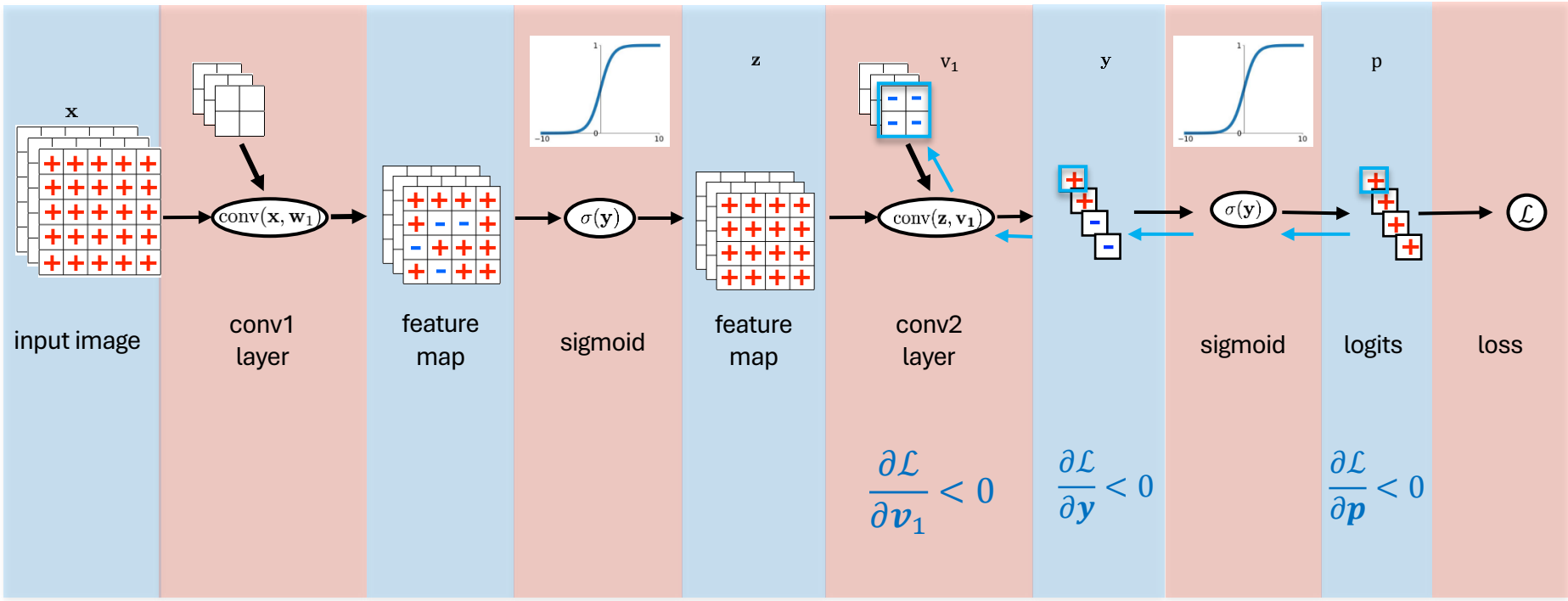
$> 0$ $> 0$ $> 0$

# Sigmoid

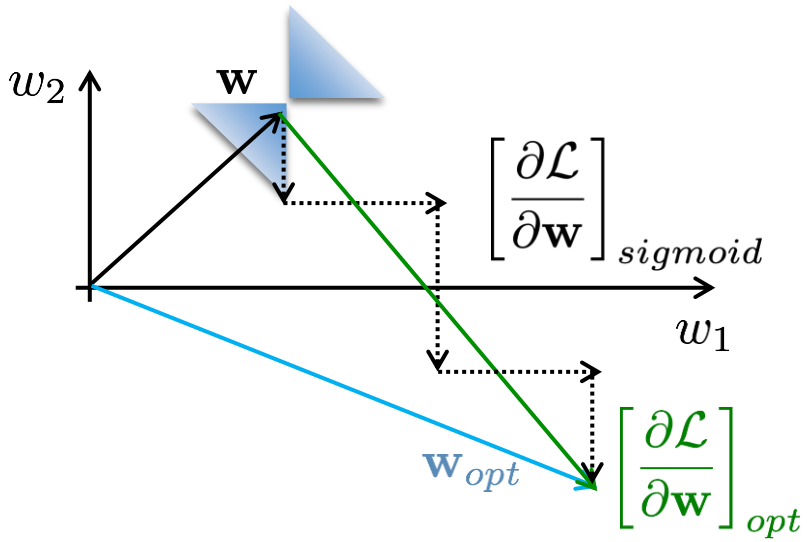

x
conv(x, w1)
z
v1
y
p
σ(y)
conv(z, v1)
σ(y)
ℒ
input image
conv1 layer
feature map
sigmoid
feature map
conv2 layer
sigmoid
logits
loss
∂ℒ/∂v1 > 0
∂ℒ/∂y > 0
∂ℒ/∂p > 0

# Sigmoid

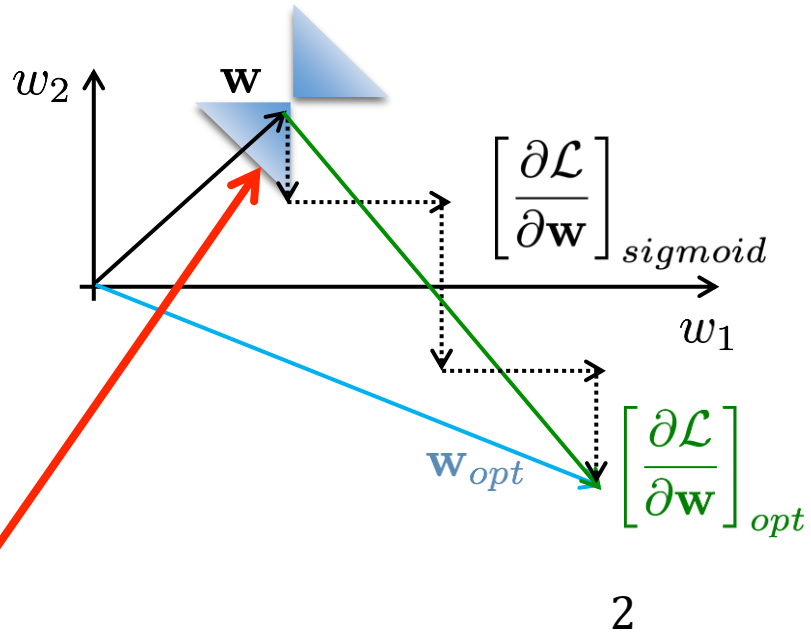
- Upstream gradient is positive → weights gradient is positive for **all weights**
- Upstream gradient is negative → weights gradient is negative for **all weights**



# Sigmoid


$w_2$
$\mathbf{w}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{sigmoid}$
$w_1$
$\mathbf{w}_{opt}$
$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]_{opt}$

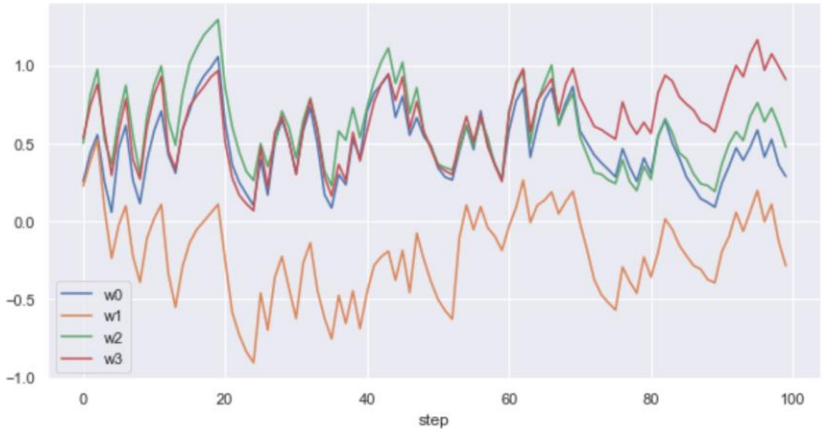
# Sigmoid



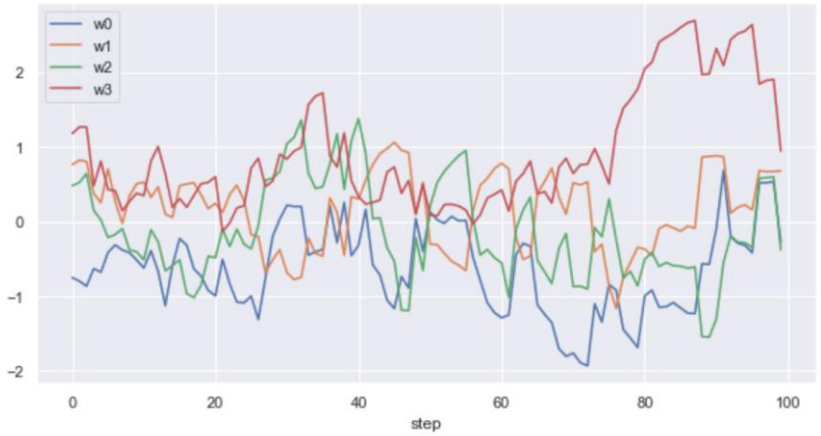

How big fraction is the blue region in a d-dimensional space? $\frac{2}{2^{\mathrm{d}}}$

# Sigmoid

## Sigmoid Activation



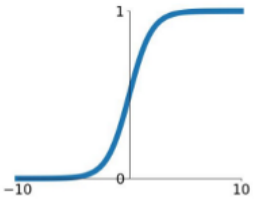

## Tanh Activation



# Sigmoid

- No gradient when saturated
- Not centered around zero (output always positive)
- Computationally expensive

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

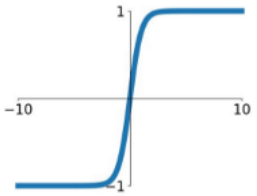



```
# Pytorch implementation
nn.Sigmoid()
```

# Tanh

- No gradient when saturated
- Centered around zero
- Computationally expensive

$$\tanh(x)$$



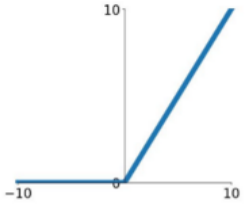

```
# Pytorch implementation
nn.Tanh()
```

# Rectified Linear Unit (ReLU)

- No gradient when saturated (for negative values)
- Not centered around zero (output always positive or zero)
- Computationally & memory cheap
- Back-propagation

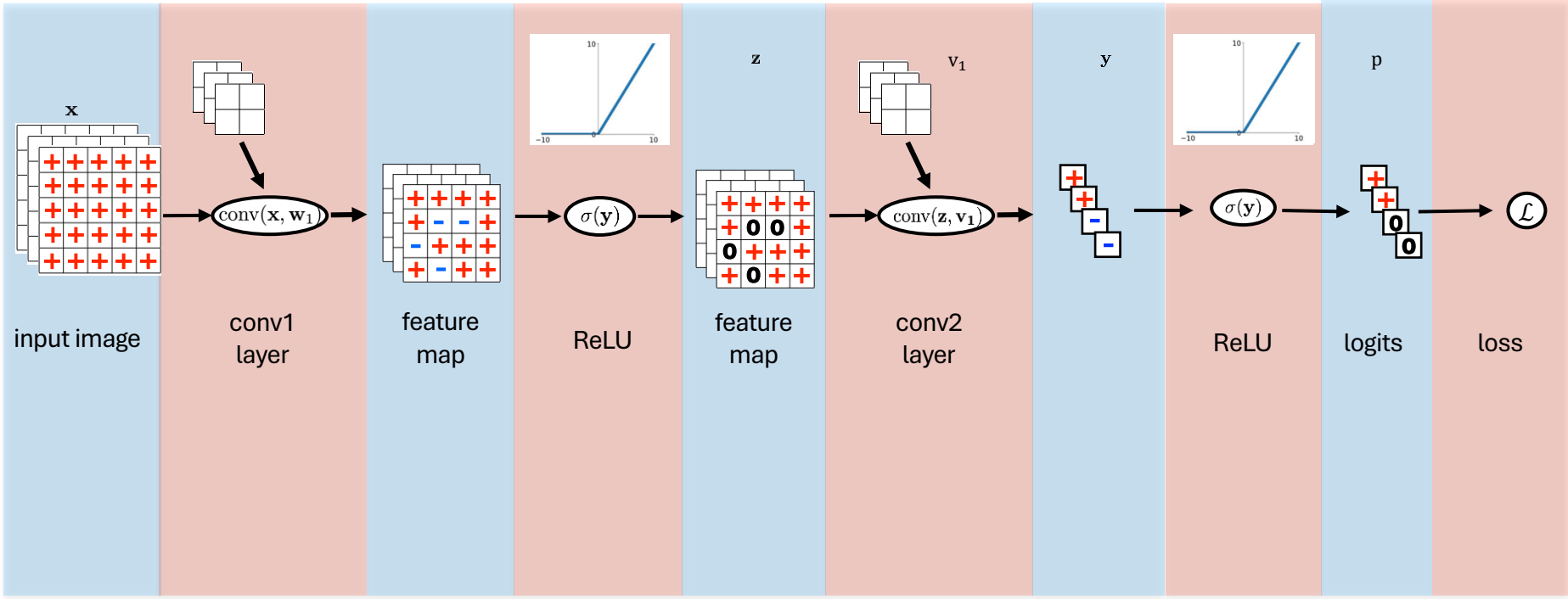
$$\frac{\partial \max(0, x)}{\partial x} = \begin{cases} 0 & x < 0 \\ 1 & \text{otherwise} \end{cases}$$

$$\max(0, x)$$

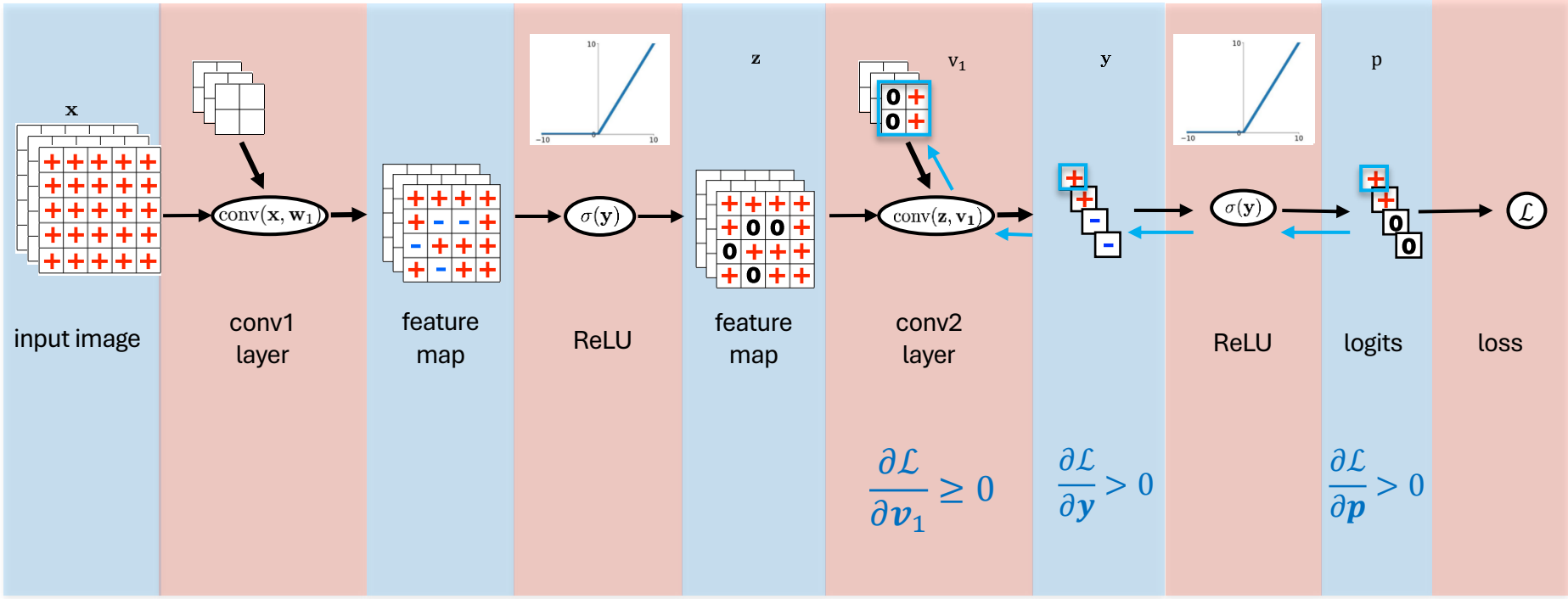



```
# Pytorch implementation
nn.ReLU(inplace=True)
```

# Rectified Linear Unit (ReLU)

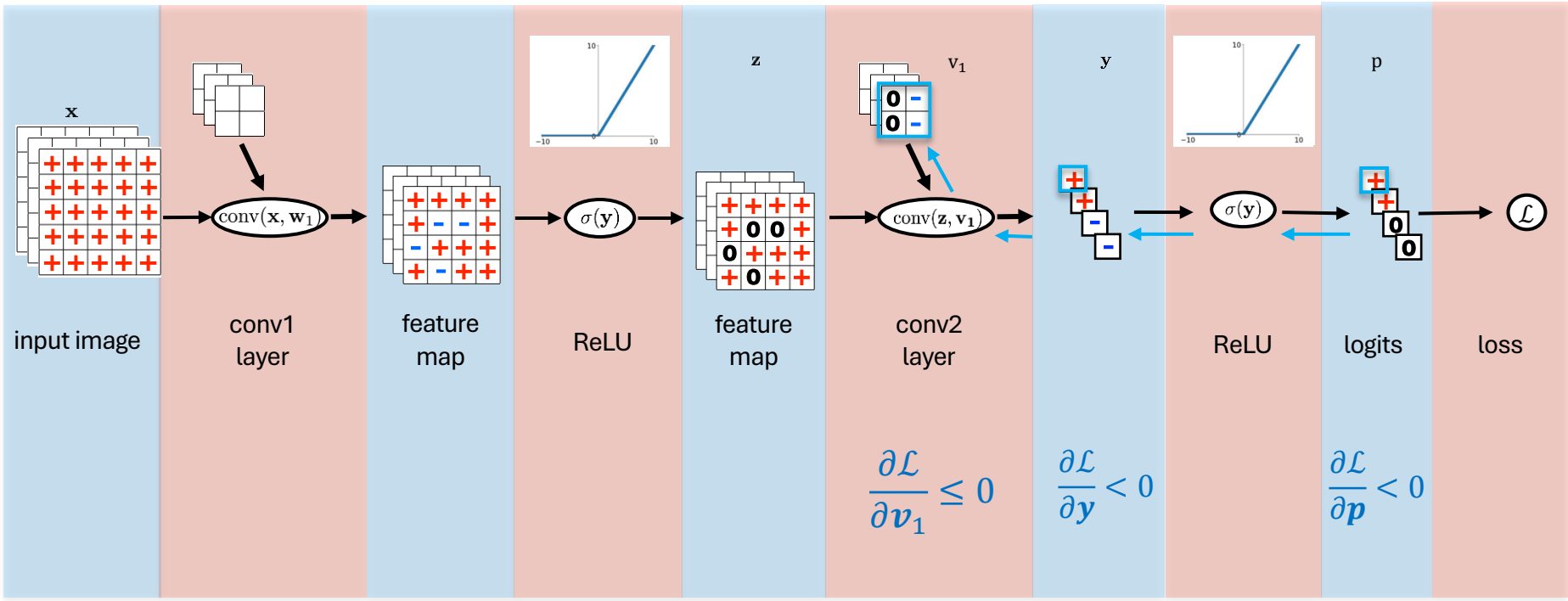


# Rectified Linear Unit (ReLU)



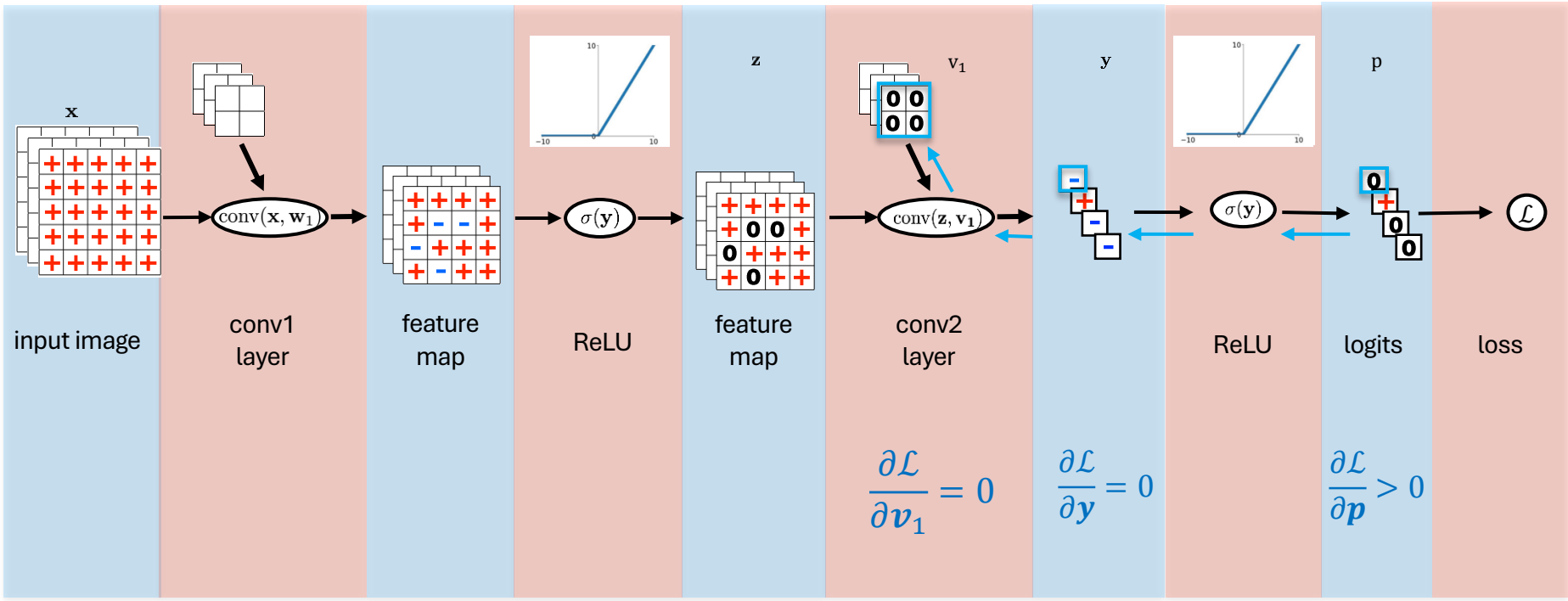
# Rectified Linear Unit (ReLU)

- Upstream gradient is positive → weights gradient is positive or zero (the given weight remains unchanged)
- Upstream gradient is negative → weights gradient is negative or zero



# Rectified Linear Unit (ReLU)

- When output of ReLU is zero, gradient propagation stops completely
- **“Dead ReLU problem”**

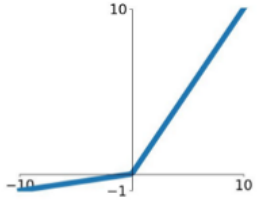



$$\frac{\partial \max(0, x)}{\partial x} = \begin{cases} 0 & x < 0 \\ 1 & \text{otherwise} \end{cases}$$

# ReLU alternatives

## Leaky ReLU

$$\max(0.1x, x)$$

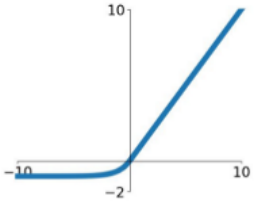



- Does not suffer from dead ReLUs
- Backpropagation:

$$\frac{\partial \max(0.1x, x)}{\partial x} = \begin{cases} 0.1 & x < 0 \\ 1 & \text{otherwise} \end{cases}$$

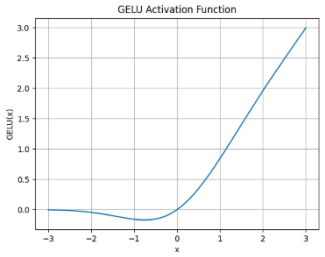
## ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$




- Does not suffer from dead ReLUs
- Removes discontinuity around zero

## GELU



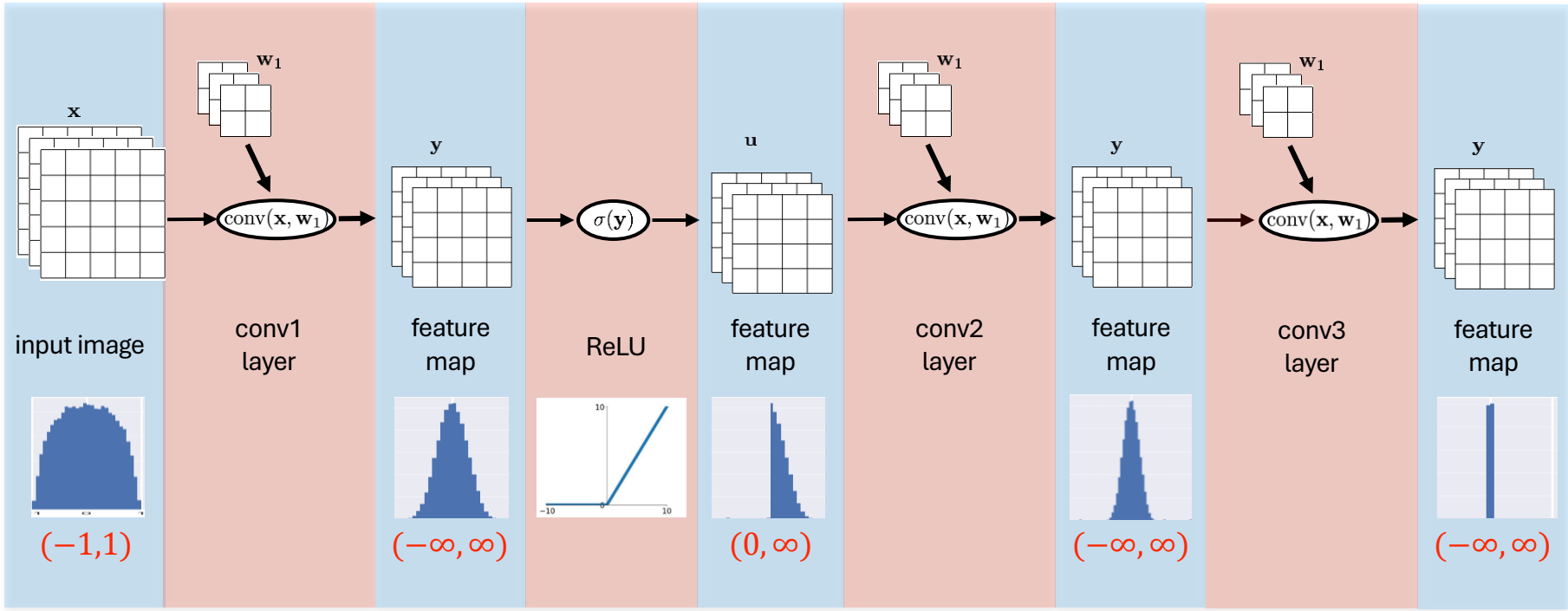

- Does not suffer from dead ReLUs
- Removes discontinuity around zero
- Zero activation for large negative values
- Used in Transformers


# Activation Functions

- Many reasonable choices (ReLU, LeakyReLU, ELU, GELU, …)
- Choosing the best one strongly depends on problem and the data

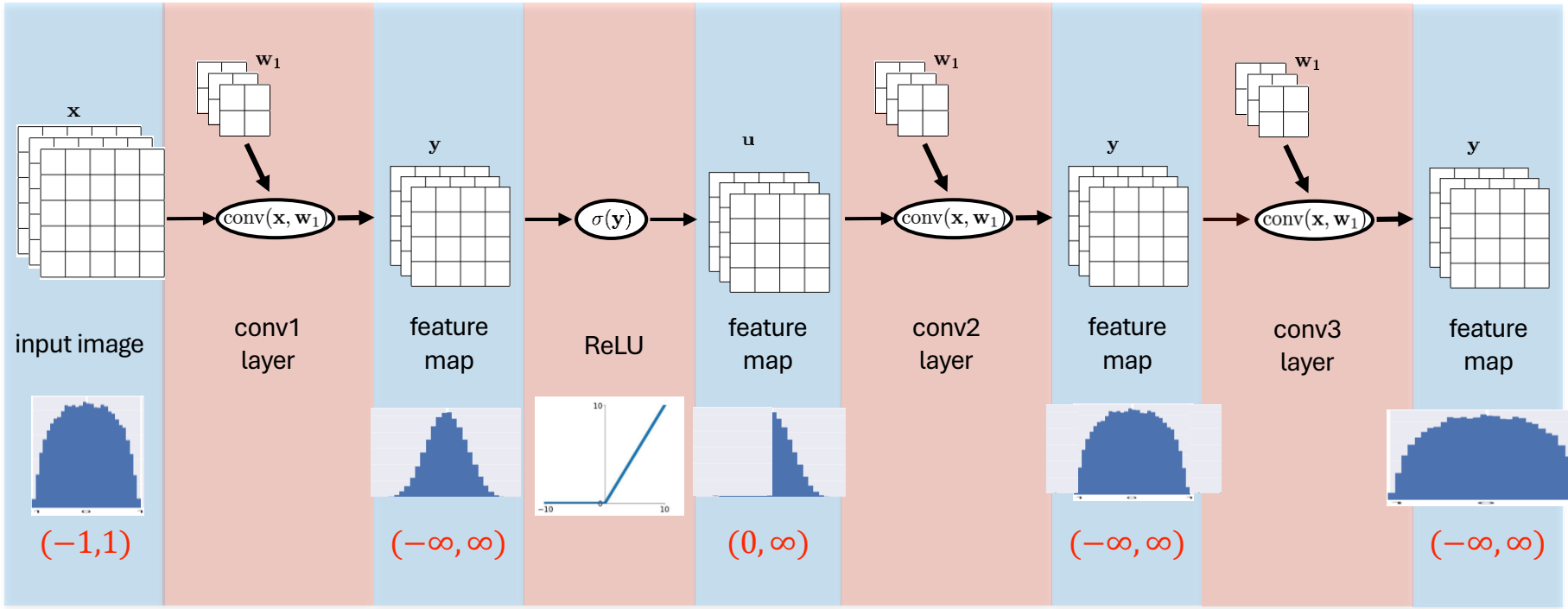
# Feature Values

- Small weights cause feature values converge towards zero



# Feature Values

- Small weights cause feature values converge towards zero
- Large weights cause feature values explode towards infinity
- Gradients are also a convolution → gradients tend to vanish or explode



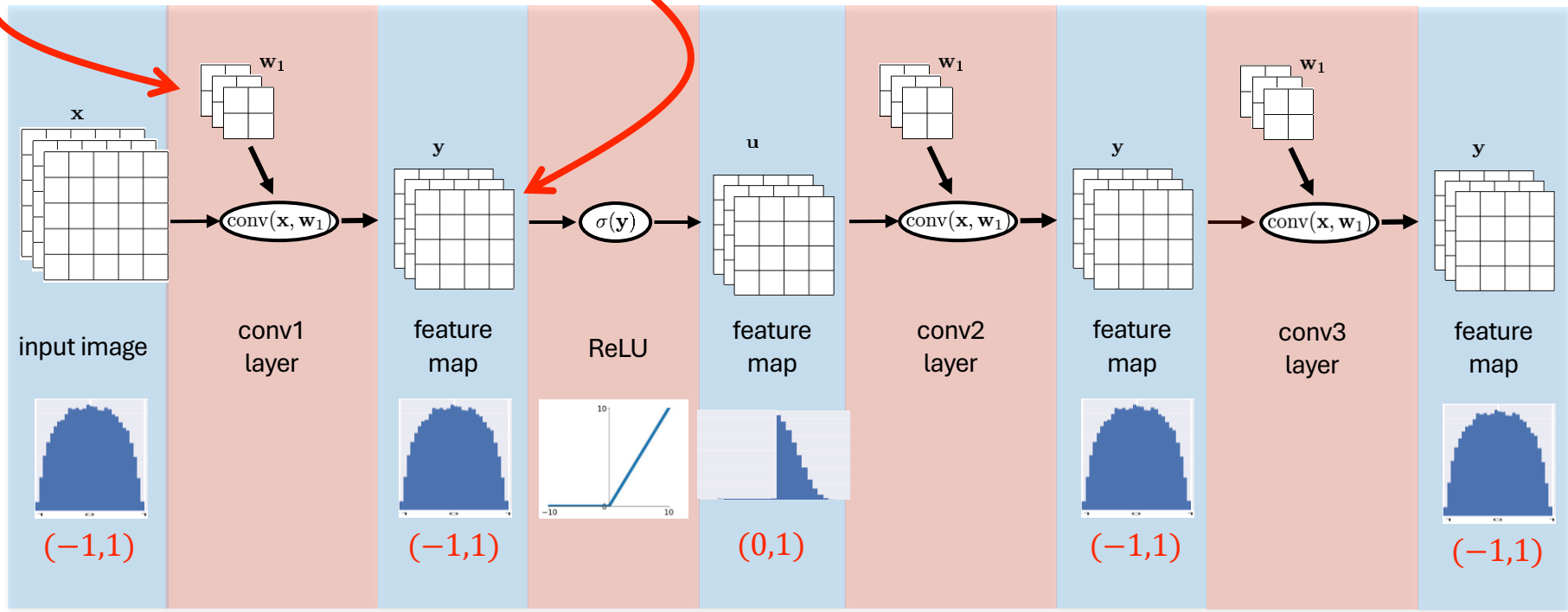
# Feature Values

- Values in feature maps can easily explode, diminish or get biased
- Gradients can easily explode or diminish or get biased
- We need to keep reasonable values - what are reasonable values?


$$\mu_x = 0 \quad \mathrm{var}(x) = 1$$

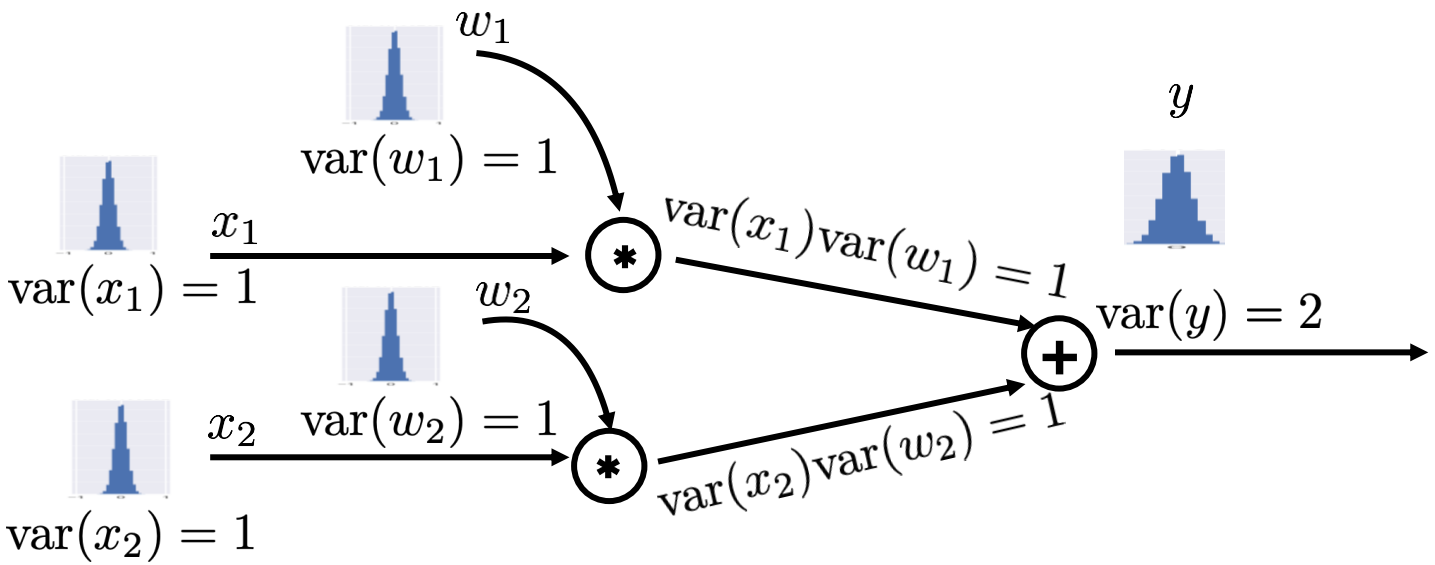
# Feature Values

- To keep feature values within “reasonable” range, we need to consider
  1. Weight Initialization
  2. Feature Normalization



# Weight Initialization

- How to preserve variance between layers (i.e $\mathrm{var}(y) = \mathrm{var}(x_i)$) ?



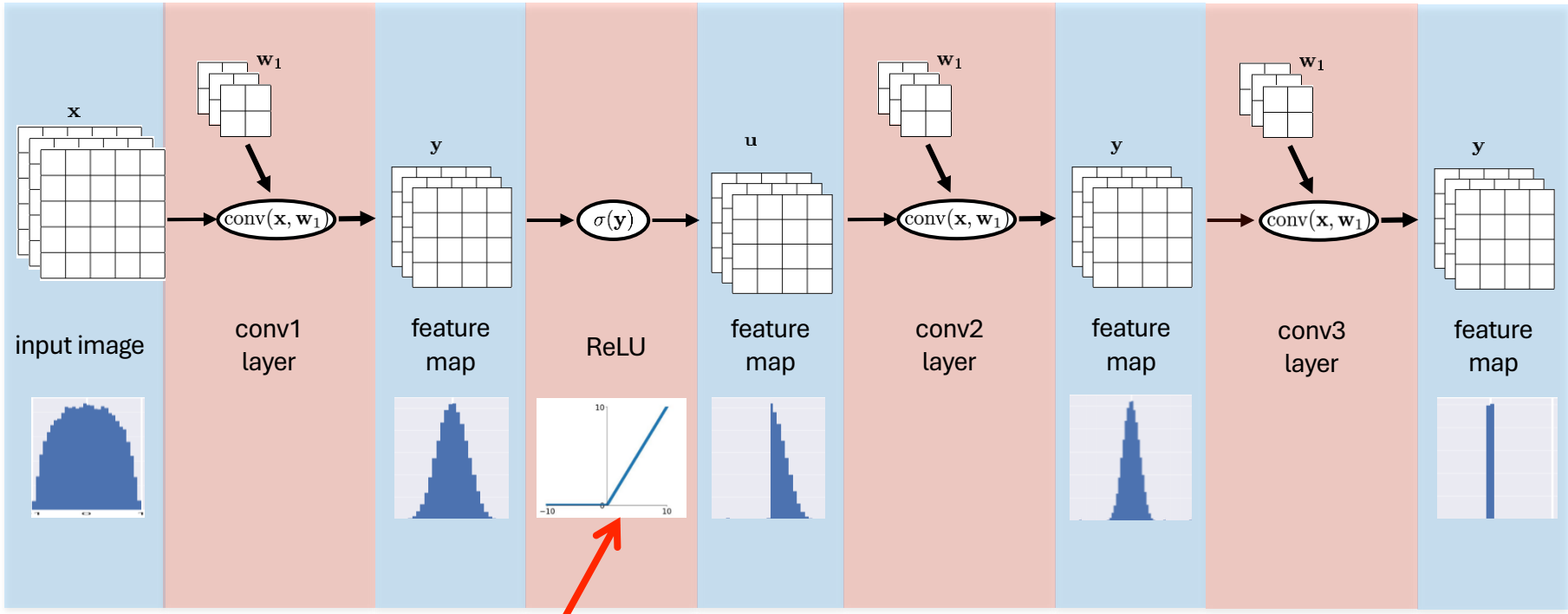

$$\mathrm{var}(x_1 w_1) = (\mathrm{var}(x_1) + \mu_{x_1}^2)(\mathrm{var}(w_1) + \mu_{w_1}^2) - \mu_{x_1}^2 \mu_{w_1}^2 = \mathrm{var}(x_1)\mathrm{var}(w_1) = 1$$

$$\mathrm{var}(y) = \mathrm{var}(x_1 w_1 + x_2 w_2) = \mathrm{var}(x_1 w_1) + \mathrm{var}(x_2 w_2) = 2 \Rightarrow \mathrm{var}(w_i) = \frac{1}{2}$$

$$\mathrm{var}(y) = \mathrm{var}(w_1 x_1 + w_2 x_2 + \cdots + w_N x_N) =$$

$$= \sum_{i=1}^{N} \mathrm{var}(w_i)\mathrm{var}(x_i) \approx N * \mathrm{var}(w_i)\mathrm{var}(x_i) \Rightarrow \mathrm{var}(w_i) = \frac{1}{N}$$

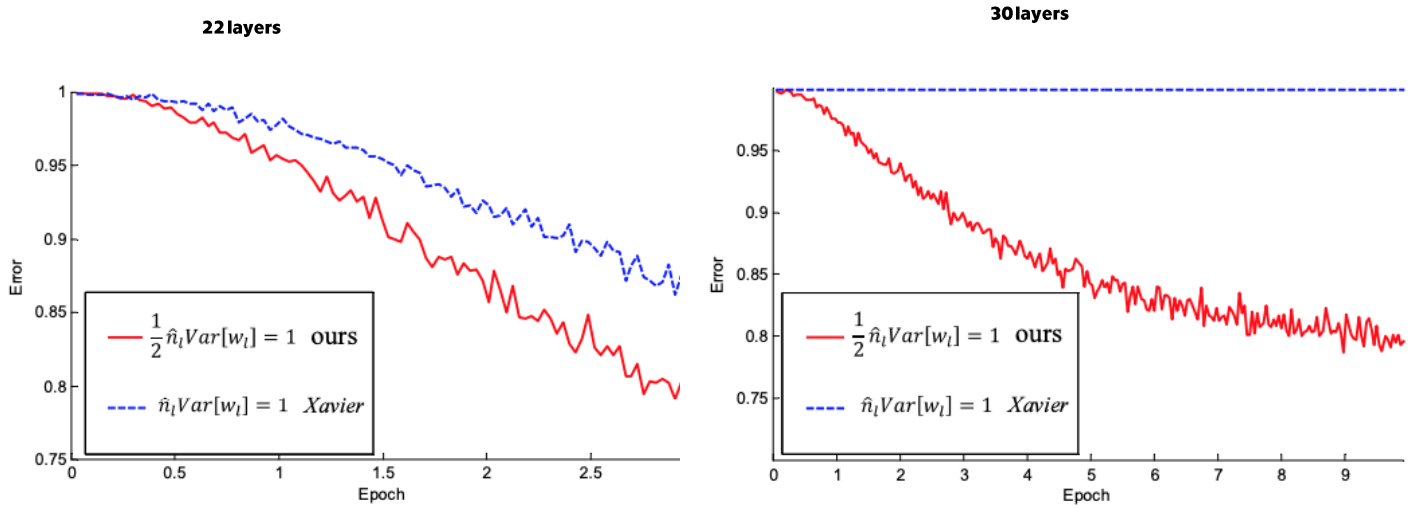
# Weight Initialization




What does ReLU do to the variance? It halves it.

# Kaiming Initialization

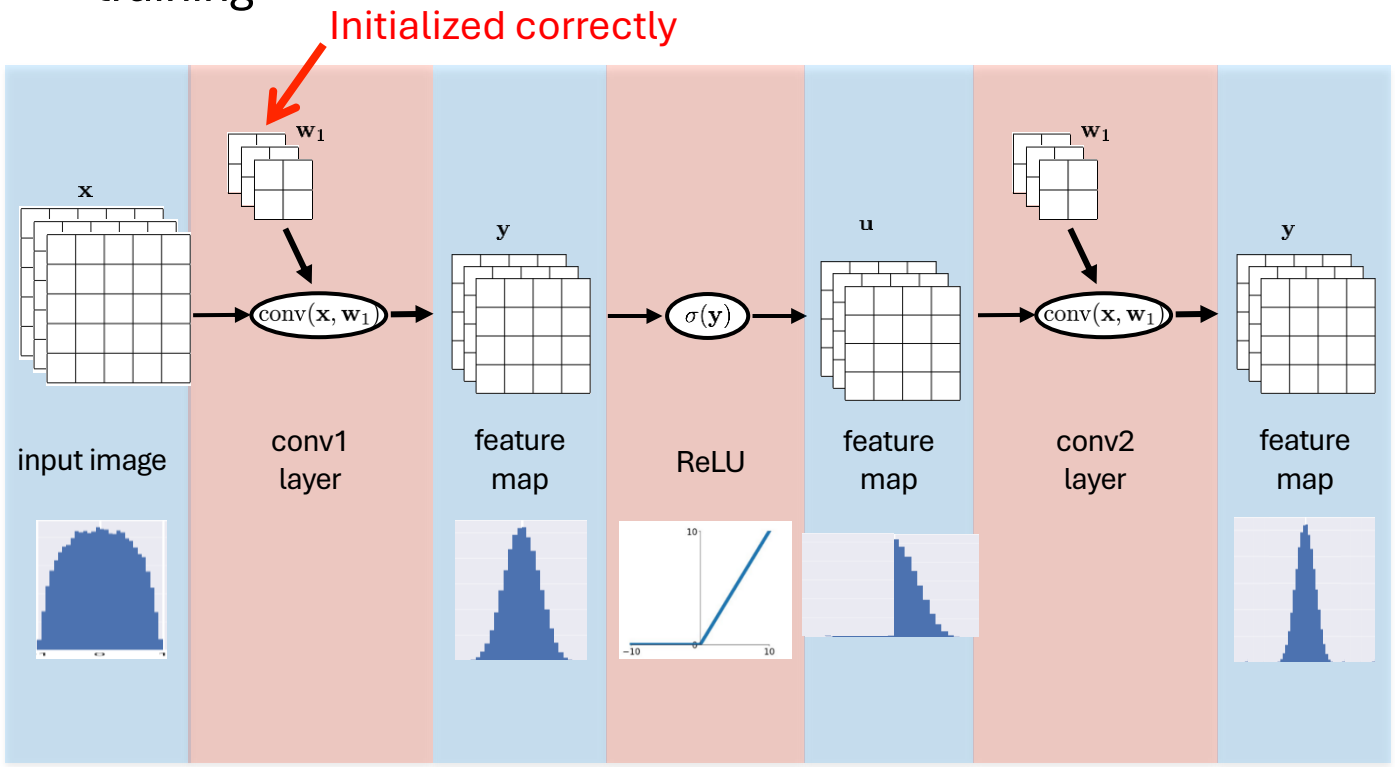
- ReLU reduces variance 2x by itself $\Rightarrow \mathrm{var}(w_i) = \dfrac{2}{N}$
- Default in PyTorch for Conv layers




He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." ICCV 2015

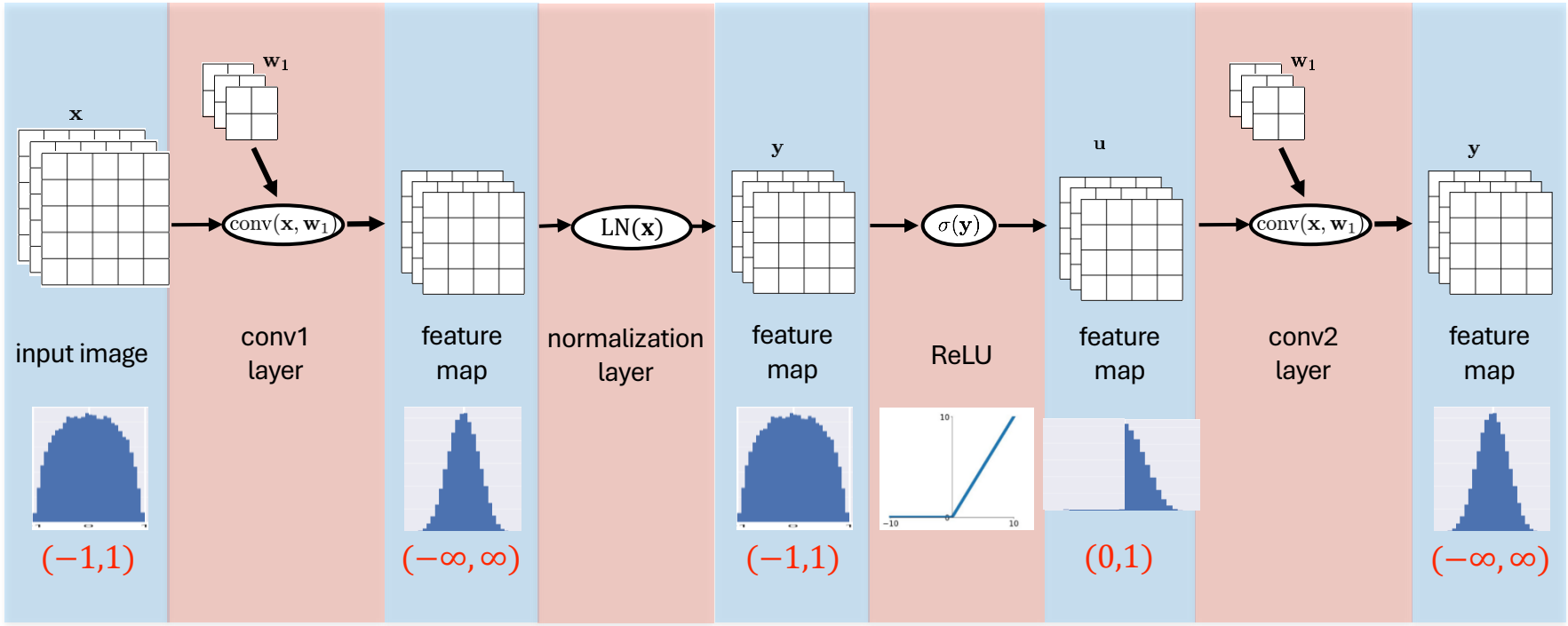
# Normalization

- What happens to weights after the first update?
- We need to add feature normalization that keeps everything in range during the training



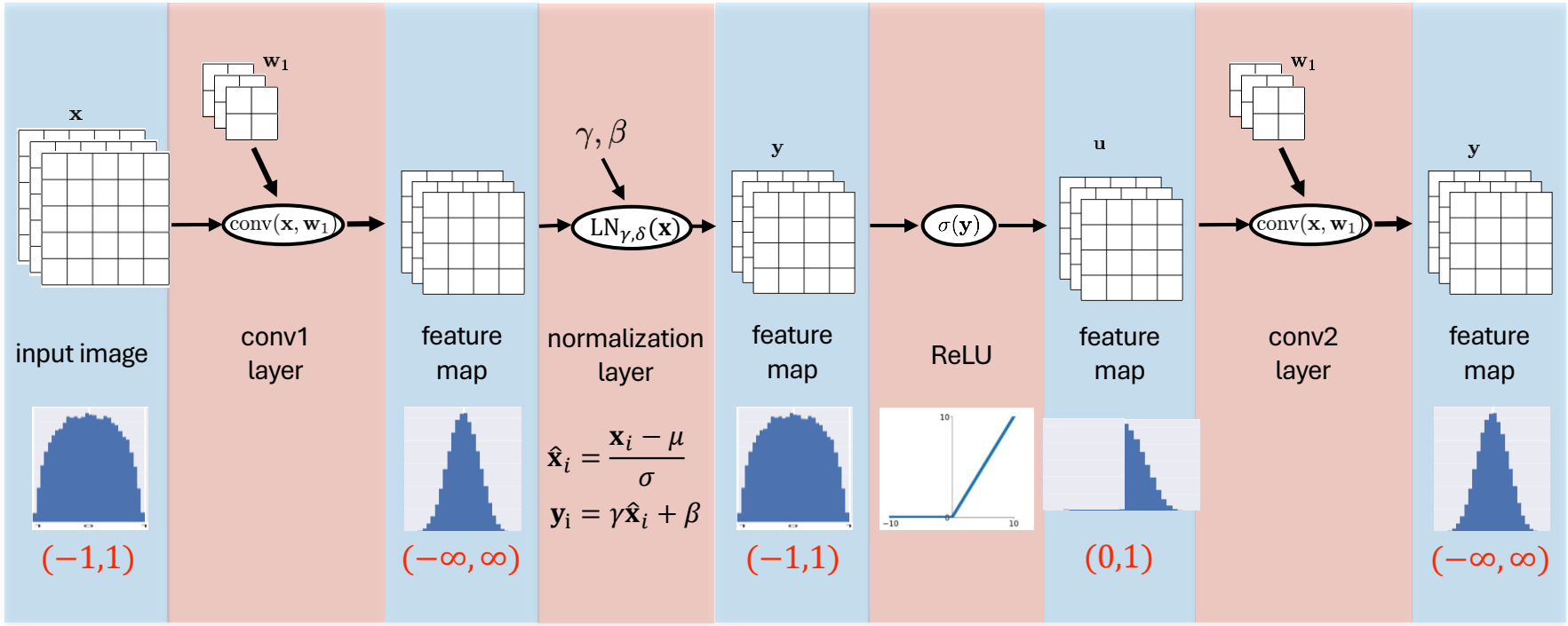
# Normalization

- What happens to weights after the first update?
- We need to add feature normalization that keeps everything in range during the training



# Normalization

- Subtract mean $\mu$ and divide by variance $\sigma$ to get $\mathcal{N}(0,1)$ distribution
- Learnable affine transformation ($\gamma$ and $\beta$) allows the network to shift the distribution if it's helpful to minimize the loss



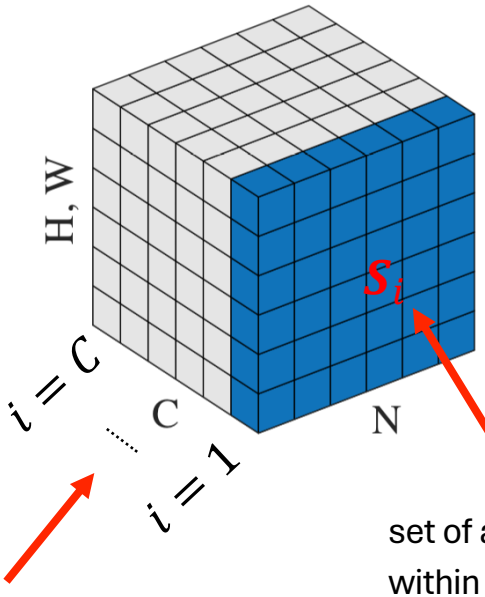
# Batch Normalization (BatchNorm)

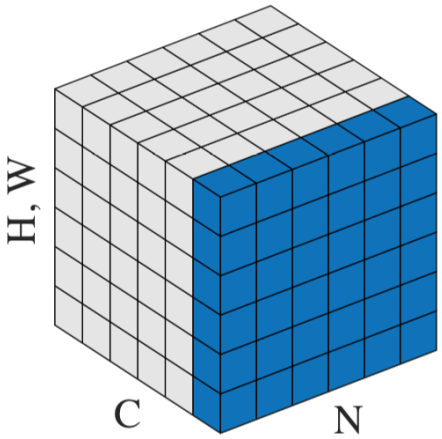
$$\mu_i = \frac{1}{m} \sum_{k \in S_i} \mathbf{x}_k$$

$$\sigma_i = \sqrt{\frac{1}{m} \sum_{k \in S_i} (\mathbf{x}_k - \mu_i)^2 + \epsilon}$$

$$\hat{\mathbf{x}}_{ki} = \frac{\mathbf{x}_{ki} - \mu_i}{\sigma_i}$$

$$\mathbf{y}_{ki} = \gamma_i \hat{\mathbf{x}}_{ki} + \beta_i$$



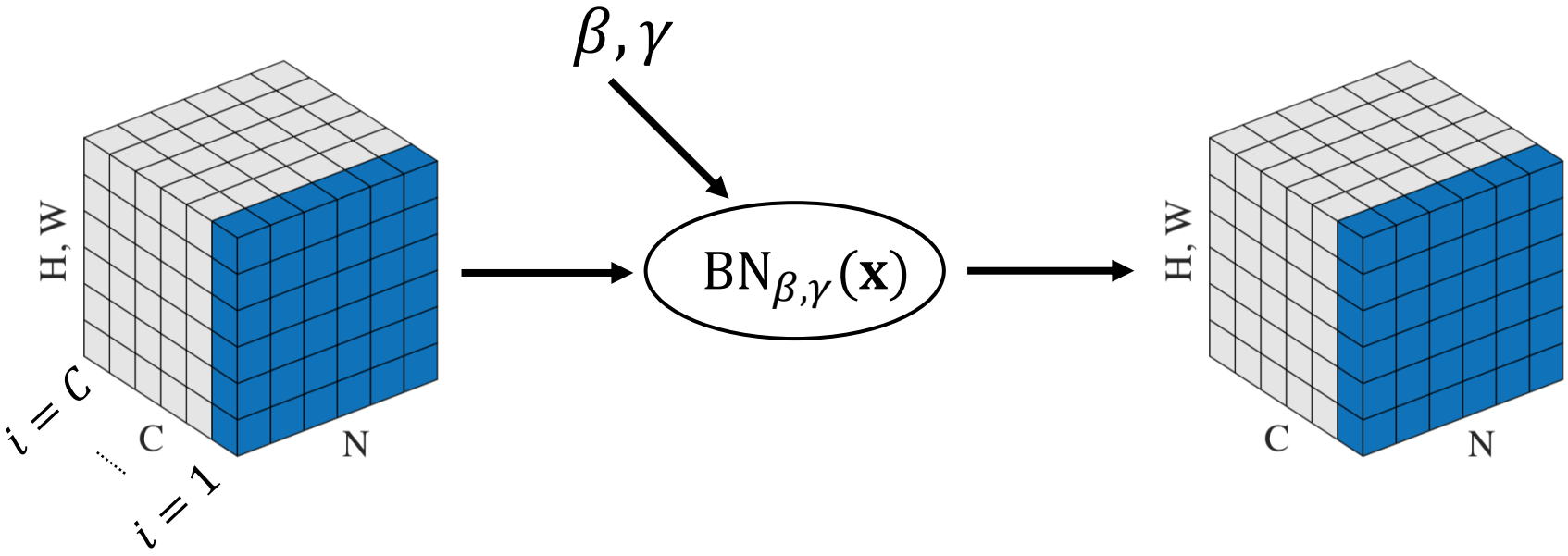




channel index

set of all element indexes within channel $i$

**Sergey Ioffe and Christian Szegedy. "Batch normalization: accelerating deep network training by reducing internal covariate shift.", ICML 2015**

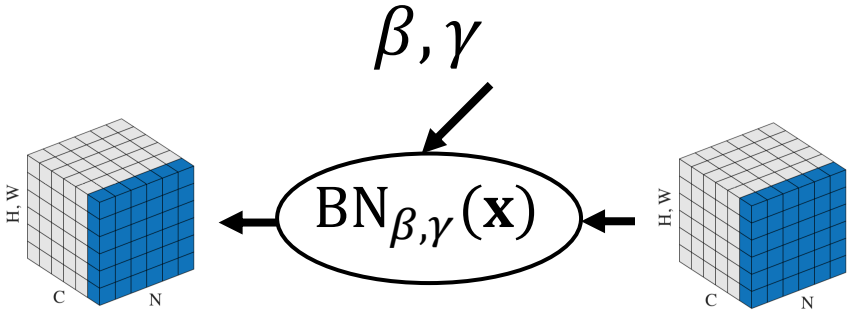
# Batch Normalization (BatchNorm)




Sergey Ioffe and Christian Szegedy. "Batch normalization: accelerating deep network training by reducing internal covariate shift.", ICML 2015

# Batch Normalization (BatchNorm)

```
# Batch x Channels x Width x Height
input = torch.randn(20, 100, 35, 45)

# Batch norm
bn = nn.BatchNorm2d(100)
output = bn(input)
```




- What is dimensionality of the output?
- The same (20x100x35x45)

- What is dimensionality of mean $\mu$ ?
- 100 dimensions

- What is dimensionality of parameter $\gamma$ ?
- 100 dimensions

- Which $\mu, \sigma$ are used during inference (testing)?
- Estimated in training set, exponential smoothing during inference, …

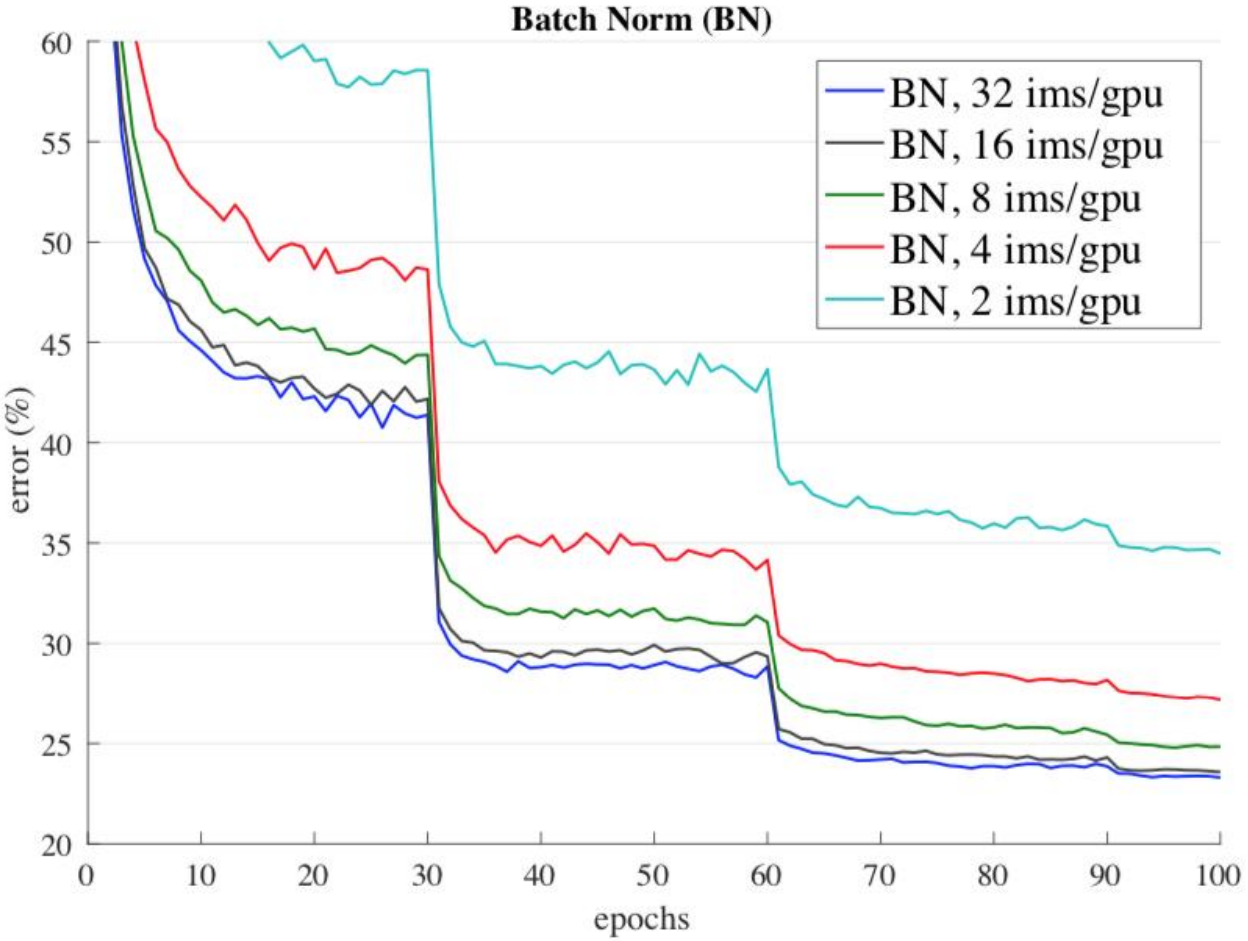
# Batch Normalization (BatchNorm)

- Why batch normalization helps?
  - It is still unclear, remains an active research topic

- Some claims / hypotheses:
  - Reduces internal covariate shift, i.e. changes in distribution of layer inputs
  - Improves gradient flow and smoother loss
  - Model regularizer: one training example always normalized differently
    $\rightarrow$ small feature map jittering (dataset augmentation)
    $\rightarrow$ better generalization

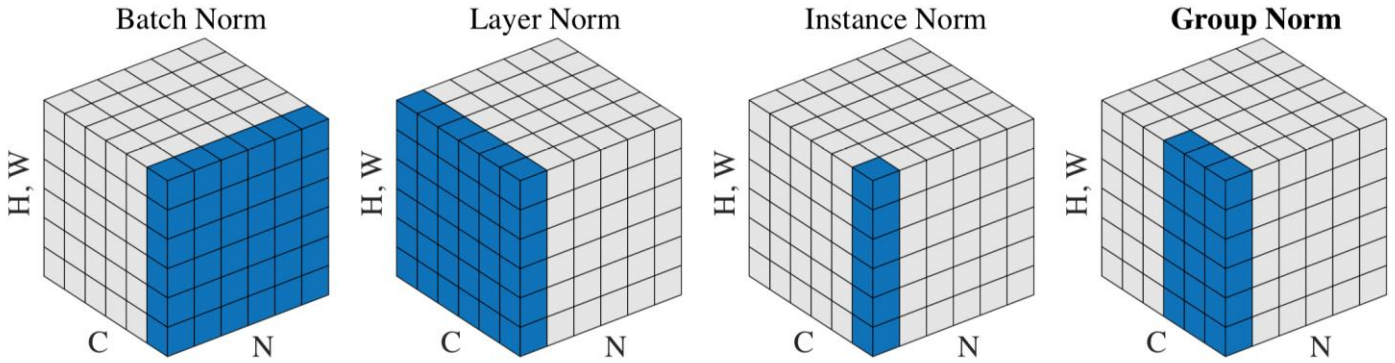
# Batch Normalization (BatchNorm)

- **Main drawback:** Does not work well for small batch sizes



# Normalization Layers

- Different normalizations are better-suited for different tasks & batch sizes

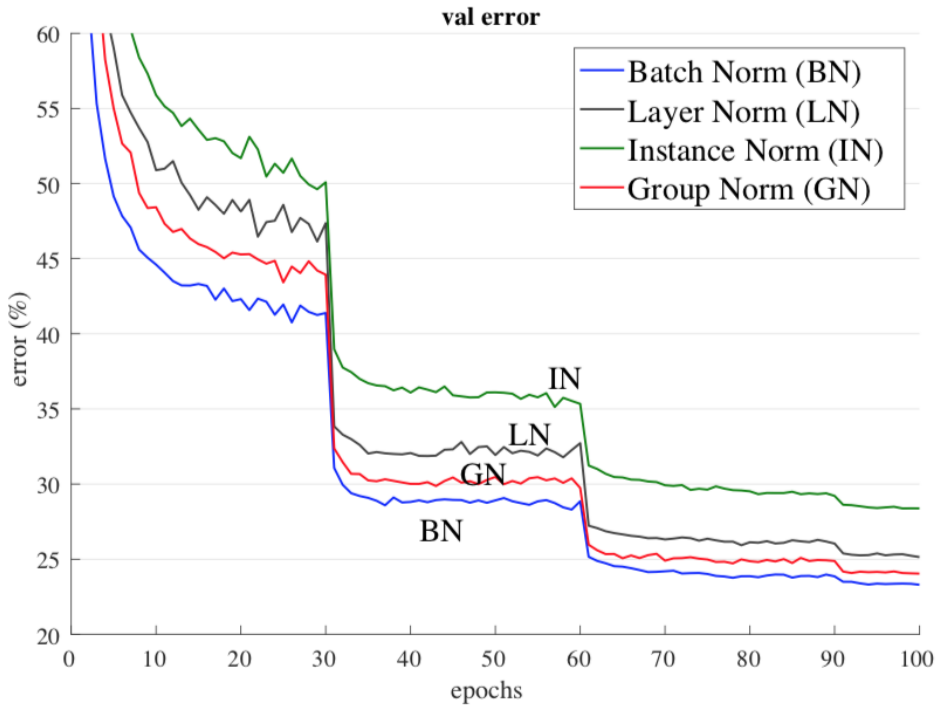



| | | | | |
|---|---|---|---|---|
| **Best for:** | Classification | RNN | Style transfer | Small batches |

# Normalization Layers

- Different normalizations are better-suited for different tasks & batch sizes

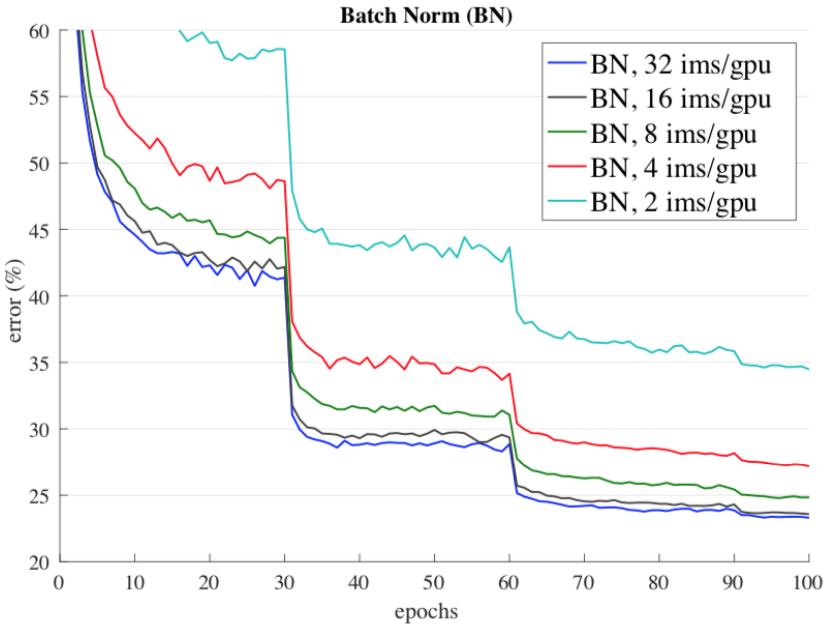
Image classification, batch size = 32

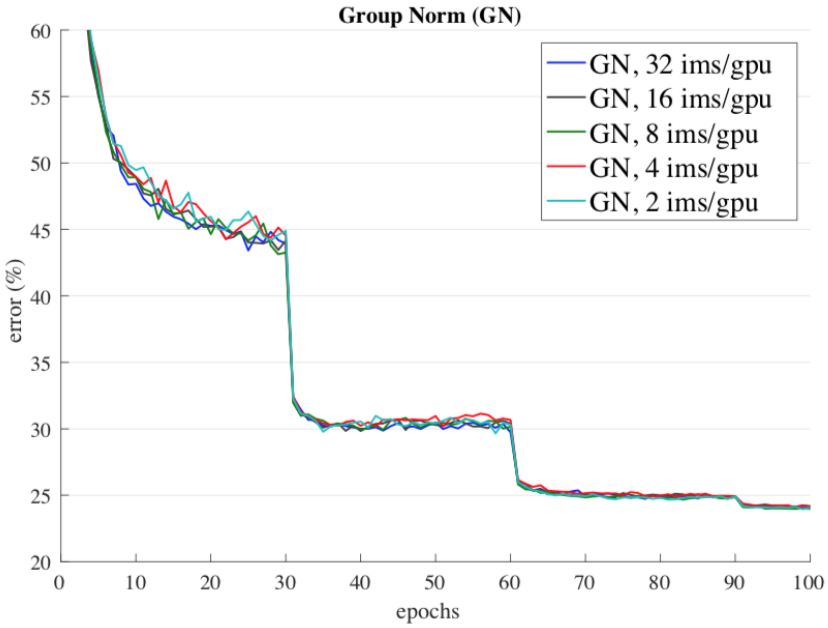



[Wu, He, CVPR, 2018] https://arxiv.org/pdf/1803.08494.pdf

# Normalization Layers

- Different normalizations are better-suited for different tasks & batch sizes




[Wu, He, CVPR, 2018] https://arxiv.org/pdf/1803.08494.pdf

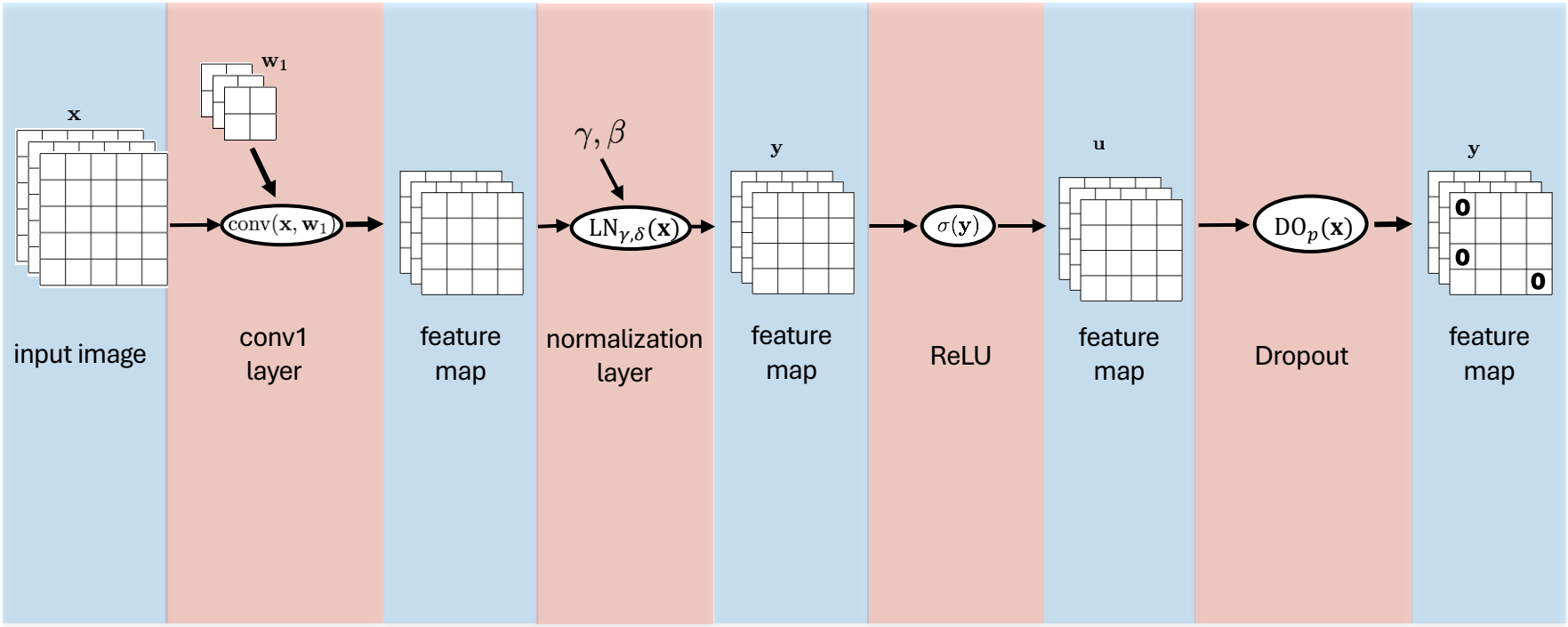

# Model Regularization

- Prevents overfitting by preferring simple models

- Dropout
- Weight decay

# Dropout

- During training, in each iteration randomly set certain fraction of cells to zero
- Reset cells independently, reset random channel (Dropout2d), ...



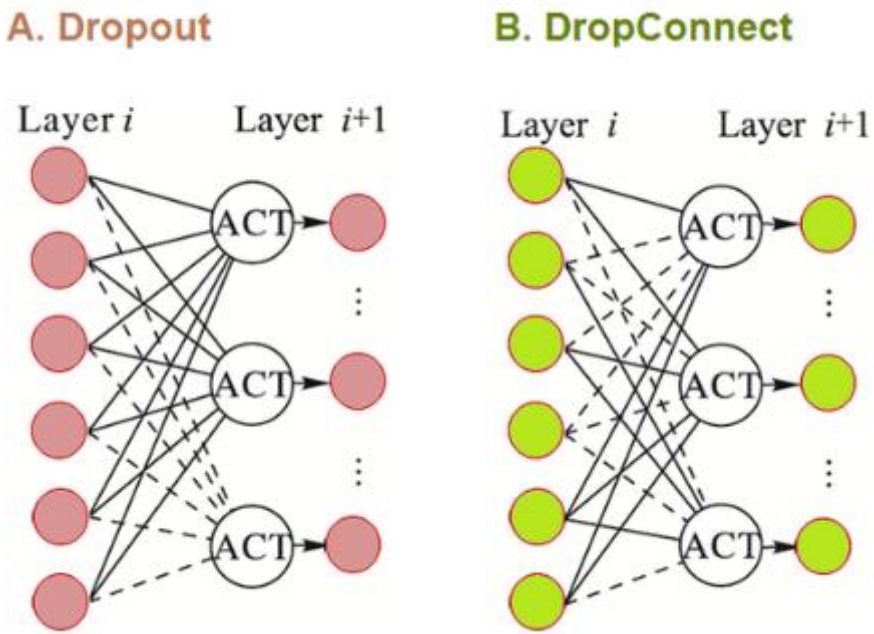
# Dropout

- In Pytorch, is important to switch model between training / eval modes because this controls Dropout layers

```
# Disable dropout -> use all cells
model.eval()

evaluate_model(model)

# Re-enable dropout -> randomly zero cells
model.train()

train_one_epoch(model)
```

# DropConnect

- Randomly drop connections rather than cells



# Weight Decay

- We add a L2 norm term to the loss to penalize large weights
- Original loss

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \left( \prod_i p(y_i|\mathbf{x}_i, \mathbf{w}) \right) = \arg\min_{\mathbf{w}} \mathcal{L}(f(\mathbf{x}, \mathbf{w}), y)$$

- New loss

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \left( \prod_i p(y_i|\mathbf{x}_i, \mathbf{w}) p(\mathbf{w}) \right) = \arg\min_{\mathbf{w}} \mathcal{L}(f(\mathbf{x}, \mathbf{w}), y) + \lambda \|\mathrm{w}\|^2$$

# Weight Decay

- Can be efficiently implemented as part of the update step

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \left( \prod_i p(y_i|\mathbf{x}_i, \mathbf{w}) p(\mathbf{w}) \right) = \arg\min_{\mathbf{w}} \mathcal{L}(f(\mathbf{x}, \mathbf{w}), y) + \lambda \|\mathrm{w}\|^2$$

$$\mathbf{w} := \mathbf{w} - \alpha \left[ \frac{\partial \mathcal{L}(f(\mathbf{x}, \mathbf{w}), y)}{\partial \mathbf{w}} + 2\lambda \mathbf{w} \right] = (1 - 2\lambda\alpha)\mathbf{w} - \alpha \frac{\partial \mathcal{L}(f(\mathbf{x}, \mathbf{w}), y)}{\partial \mathbf{w}}$$

```
# Set weight decay as part of the optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=0.0001)
```

# Conclusion

- **Initialization:** set weights to have “normal” values in all layers
- **Normalization:**
  - **BN is reparametrization** of the original NN which has the same expressive power.
  - **BN yields** less sensitivity to vanishing or exploding gradient
  - **BN is model regularizer:** one training example always normalized differently => small feature map jittering (dataset augmentation) => better generalization
  - **BN works well on classification** problems with **larger batches** (>4).
  - Alternatives **LN, GN, IN** are typically used for **smaller batches, recurrent, generative** and **transformers** networks
- **Regularization:**
  - **Drop-out** reduces overfitting by randomly deactivating neurons/channels
  - **Weight decay** reduces overfitting by reducing reliance on specific weights (kernels)

# Competencies gained for the test

- Activation functions (Sigmoid, ReLU, LeakyReLU)
- Weight initialization
- Normalization layers
- Weight decay, dropout