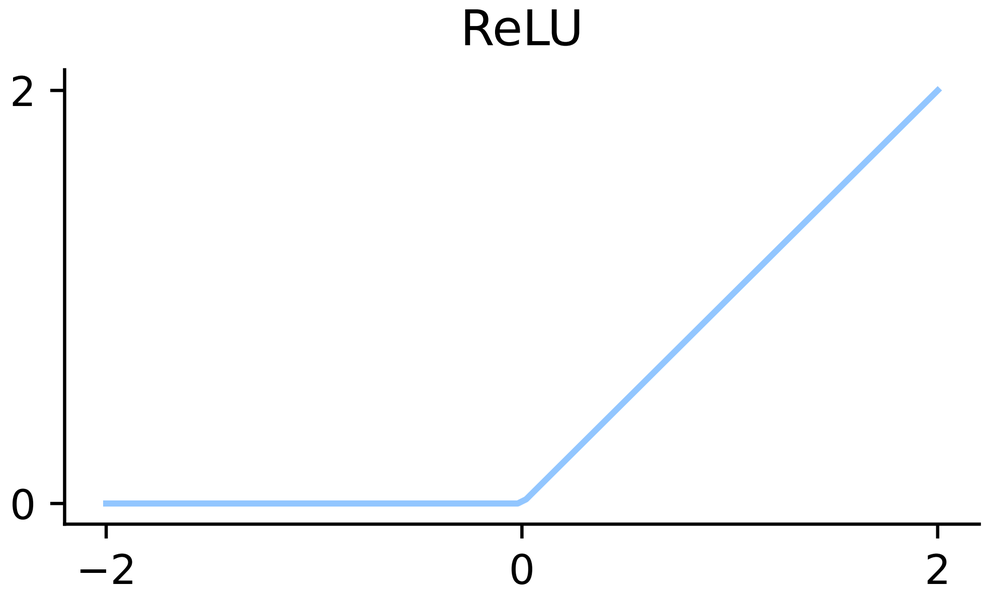
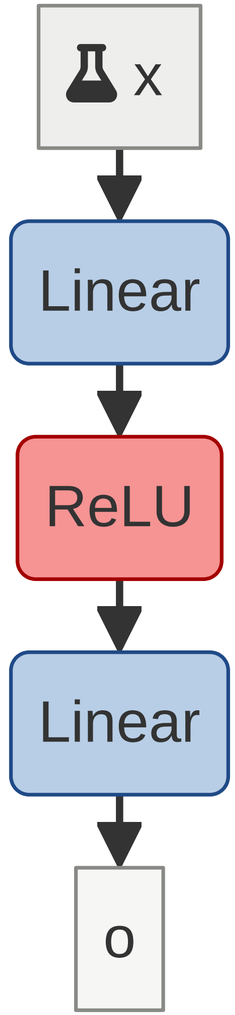
Deep Networks

Philipp Krähenbühl, UT Austin

Recap: Non-linearities

- Rectified Linear Unit (ReLU)
  $\text{ReLU}(x) = \max(x, 0)$
- Non-linear
- Differentiable almost anywhere

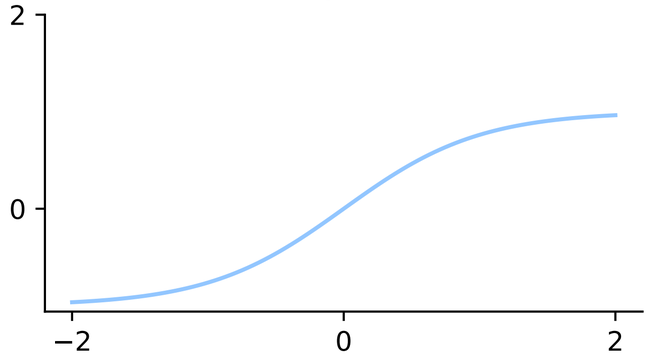




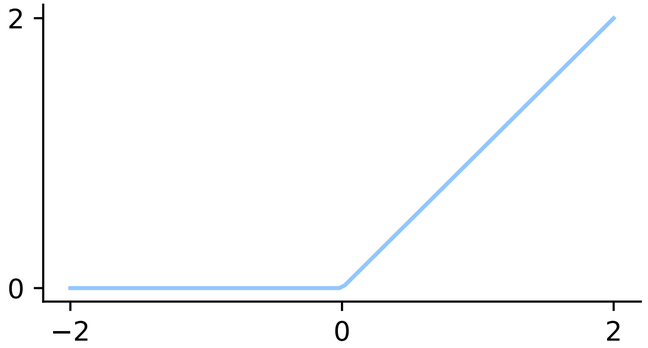
Activations

Zoo of Activation Functions

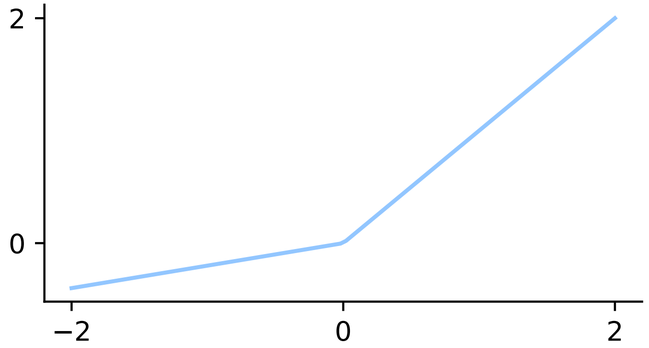
Tanh



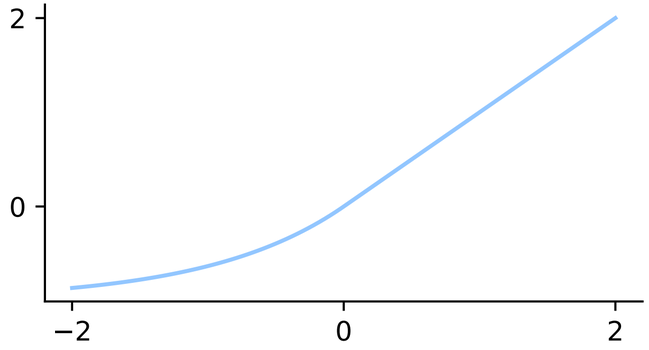
ReLU



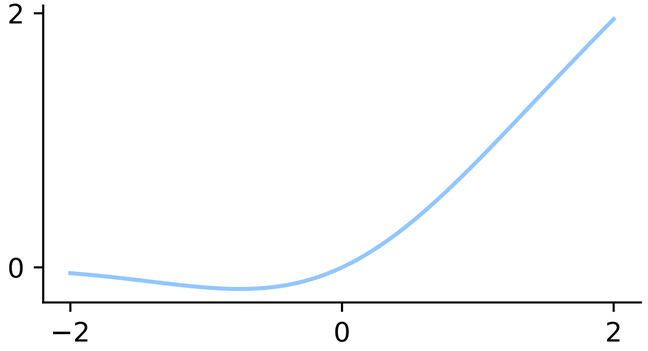
LeakyReLU



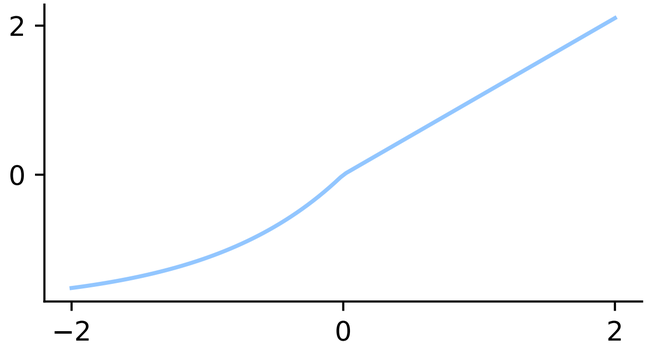
ELU



Swish

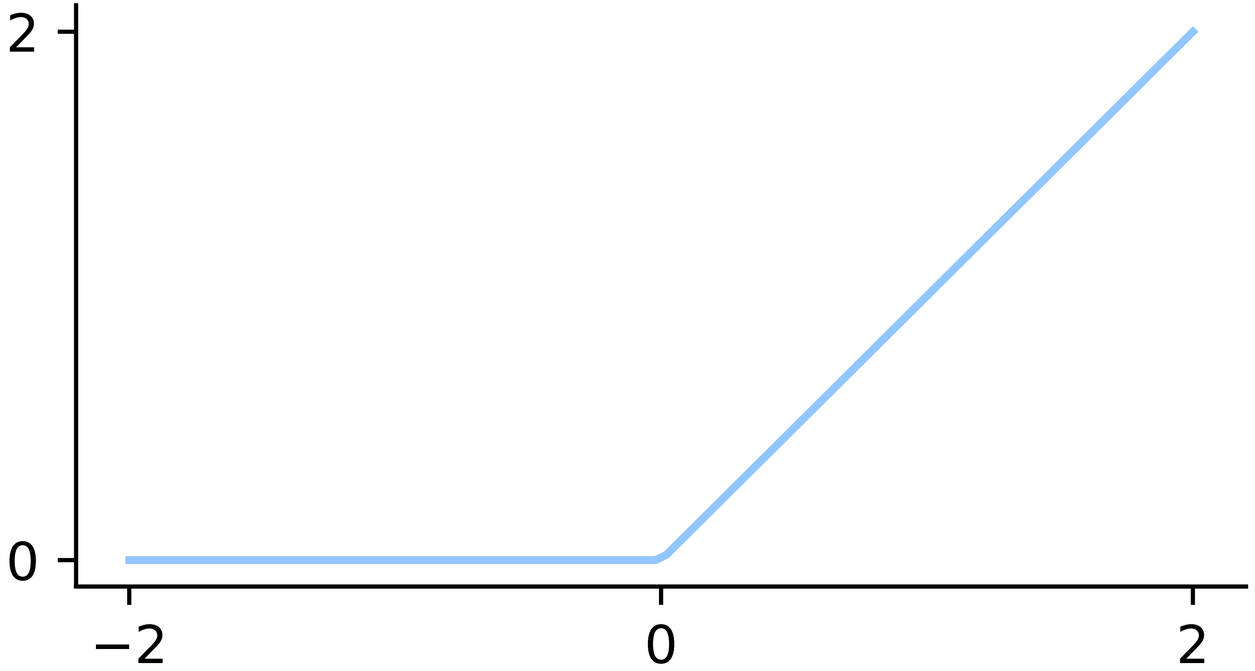


SeLU



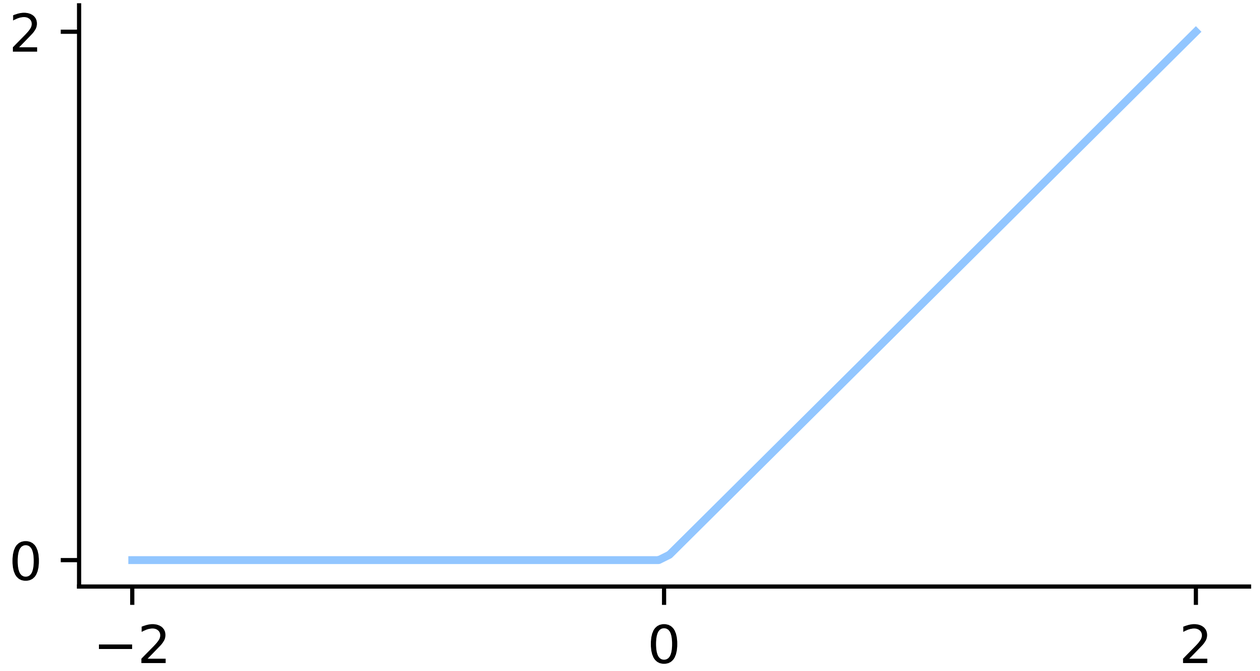
ReLU

- $\text{ReLU}(x) = \max(x, 0)$
- ✔ Simple
- ❌ ReLU units can be fragile during training and "die"



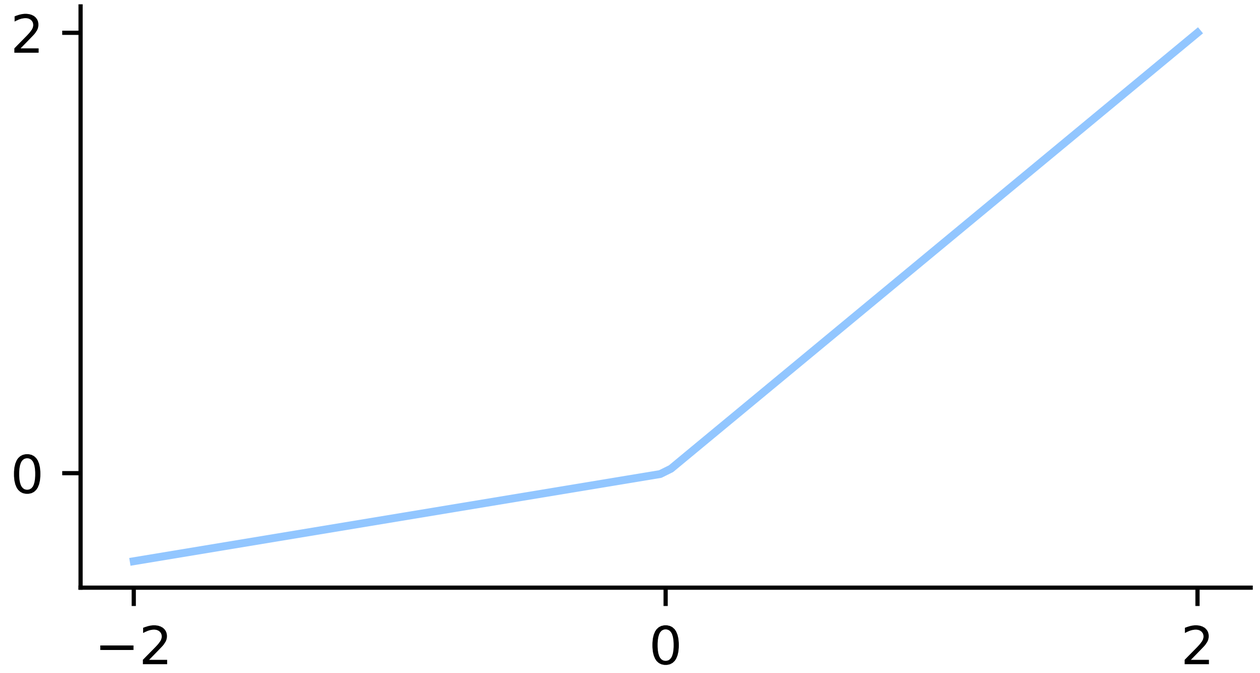
Dead ReLUs

- How can we prevent dead ReLUs?
  - Initialize network carefully
  - Decrease the learning rate



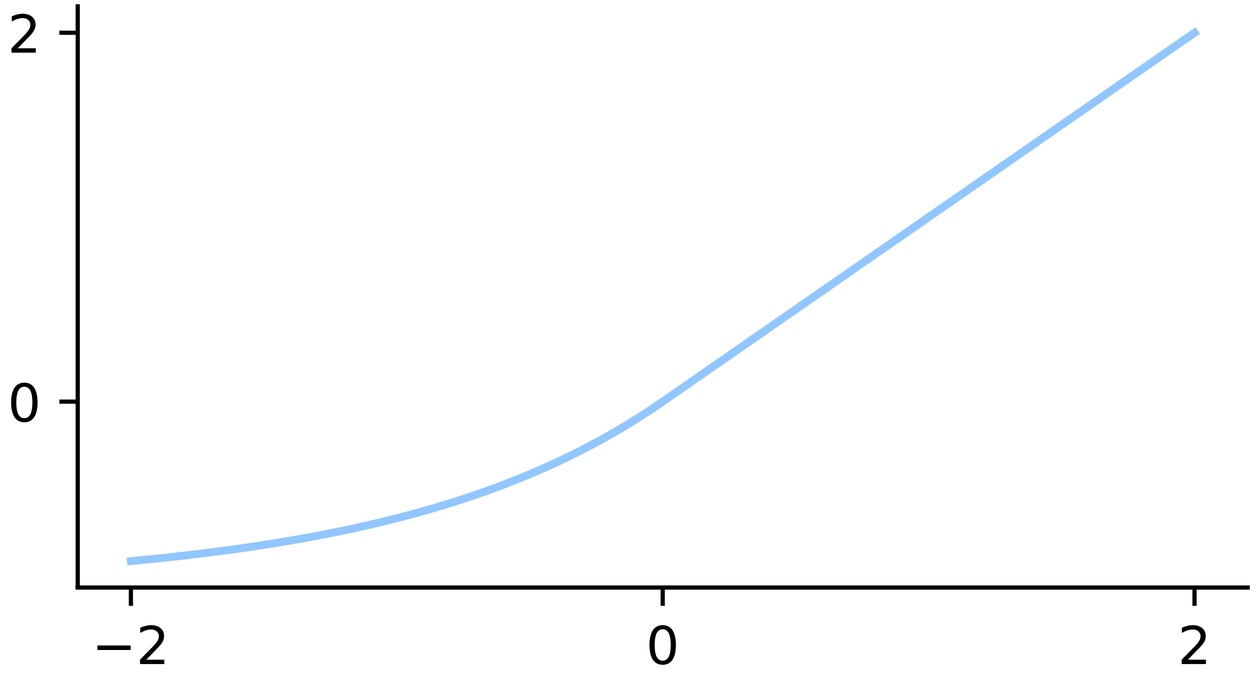
LeakyReLU

- LeakyReLU$(x) = \max(x, \alpha x)$
  - $0 < \alpha < 1$
  - Called PReLU if α is learned
- ✔ Non-zero gradient for negative inputs
- ❌ Slope α needs to be tuned
- ❌ Cannot wipe the negative signal out



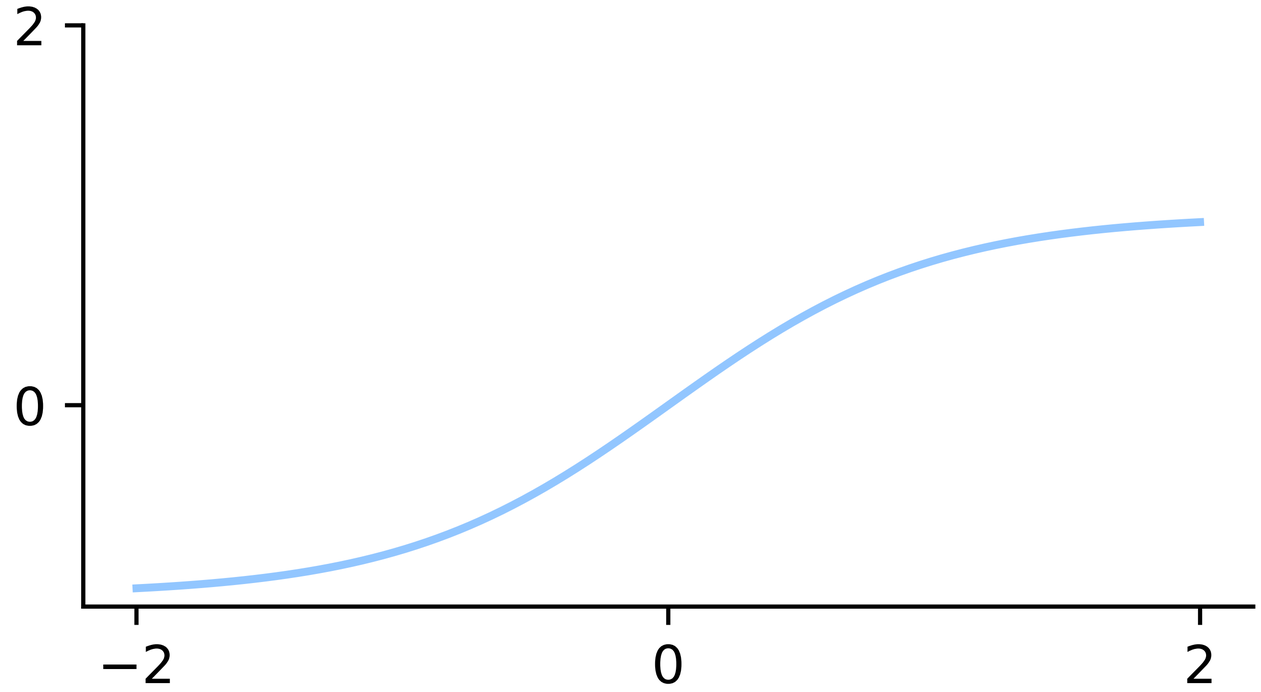
ELU

- $\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{otherwise} \end{cases}$
- ✔ Non-zero gradient for negative inputs
- ❌ Slope α needs to be tuned
- ❌ Exponential is computationally expensive



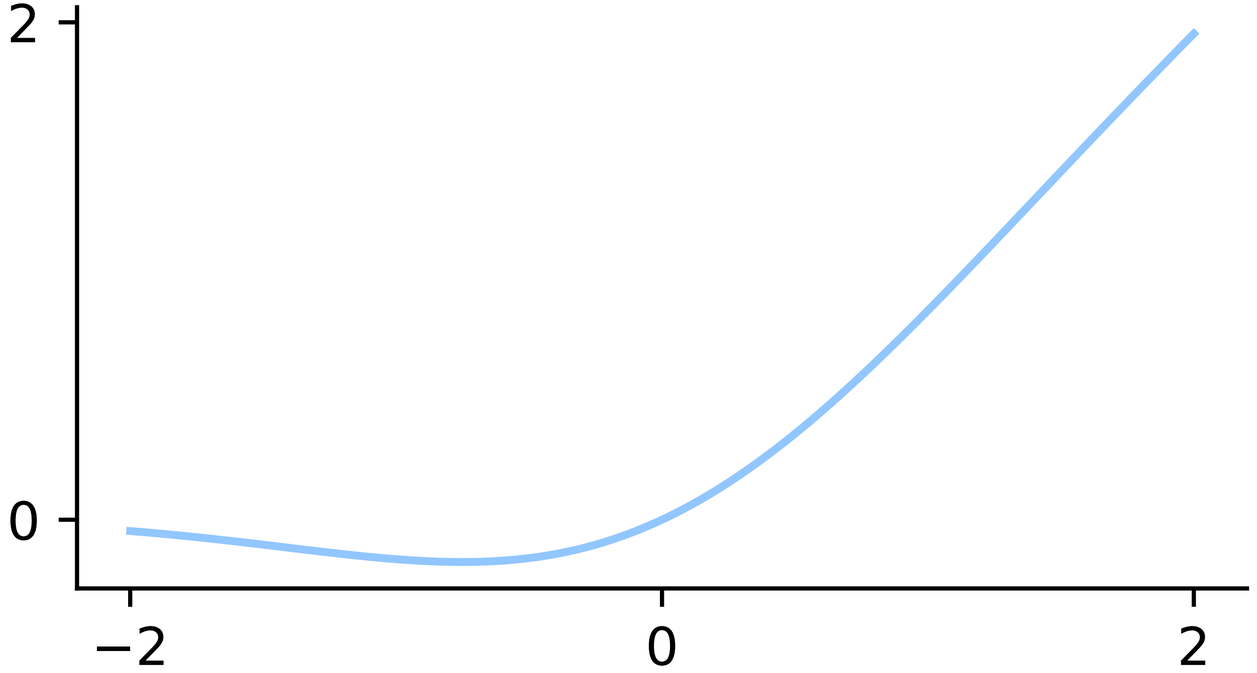
Sigmoid / Tanh

- $\sigma(x) = \frac{1}{1 + \exp(-x)}$
- $tanh(x) = 2\sigma(x) - 1$
- ❌ Saturates on both ends
- ❌ Do **not** use sigmoid/tanh



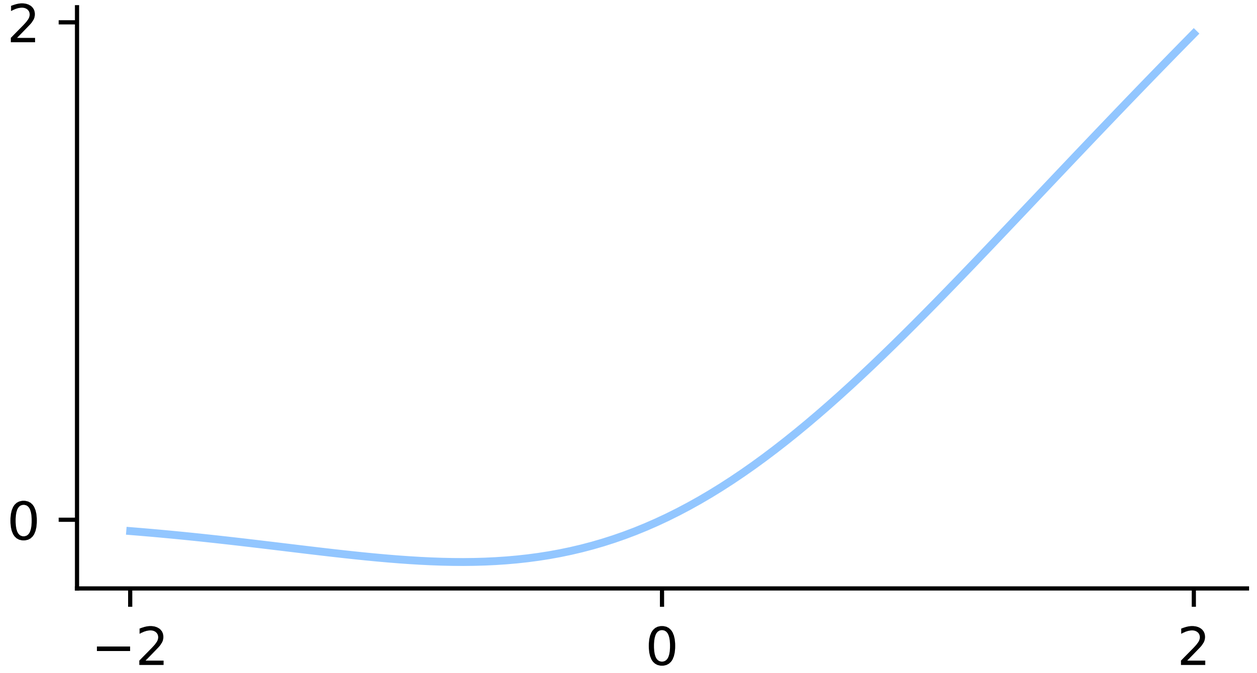
Swish

- Swish$(x) = x\sigma(\beta x)$
- ✔ Non-zero gradient for negative inputs
- ❌ Requires more computation



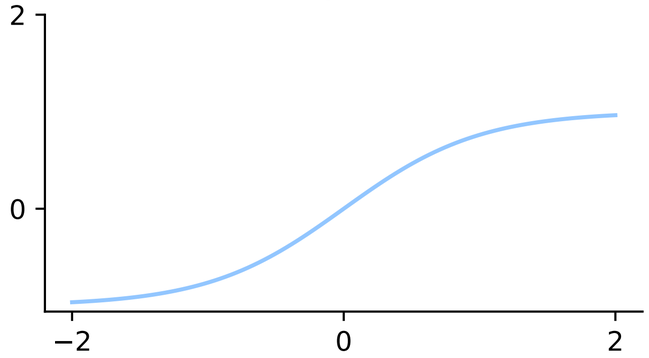
SwiGLU

- $\text{SwiGLU}(x) = \sigma(Wx + b) \cdot (Vx + c)$
  - Implemented as
    - Linear $z = Wx + b$
    - $\text{SwiGLU}(x) = \sigma(z_{1::2}) \cdot z_{0::2}$
- ✔ Learnable activation per parameter
- ✔ Used in most modern architectures
- ❌ Computationally expensive

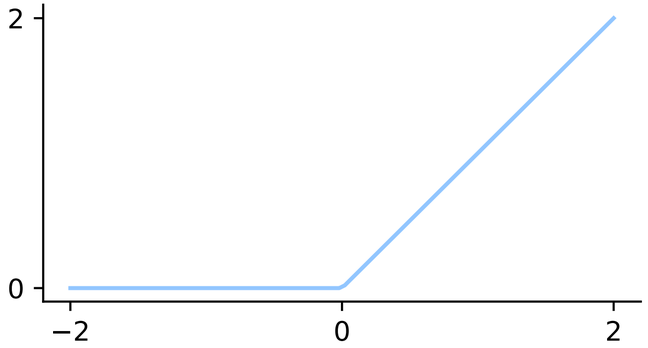


Zoo of Activation Functions

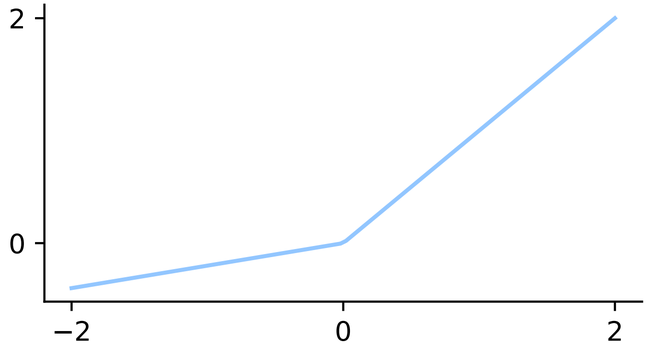
Tanh



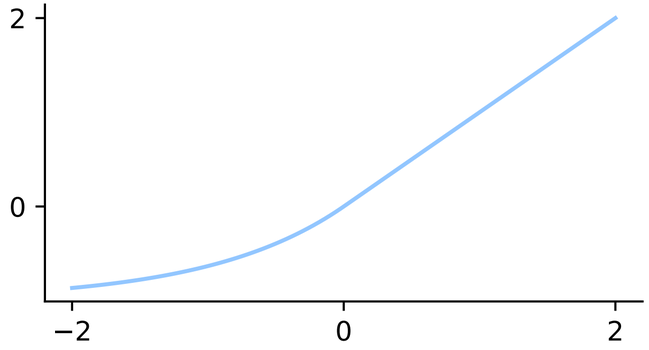
ReLU



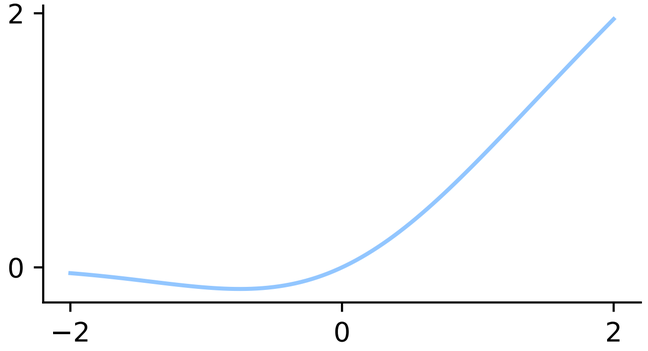
LeakyReLU



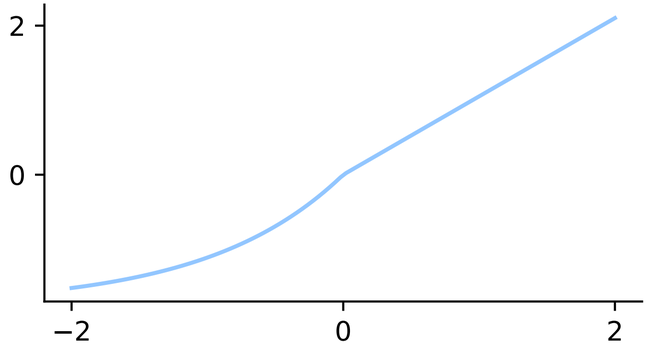
ELU



Swish



SeLU



Activation Functions - TL;DR

- Use ReLU with careful initialization and small learning rate
  - If ReLU fails, try Leaky ReLU or PReLU
- Avoid Sigmoid and Tanh
  - Use SwiGLU for sophisticated models

Deep Networks in PyTorch

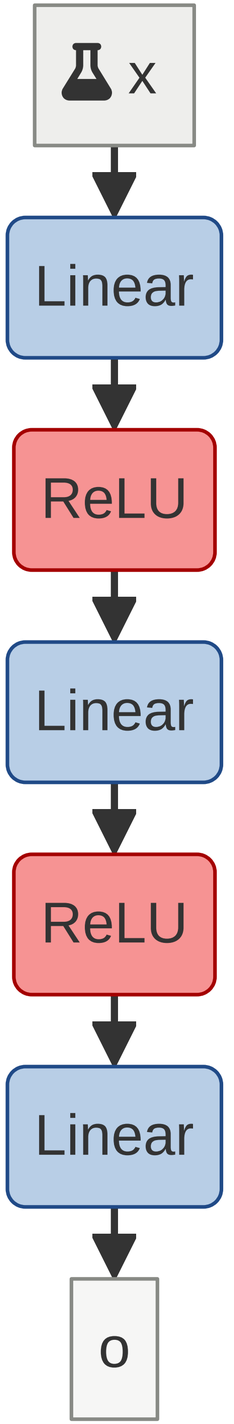
Output Representations

Recap: Universal Approximation Theorem

Universal Approximation Theorem

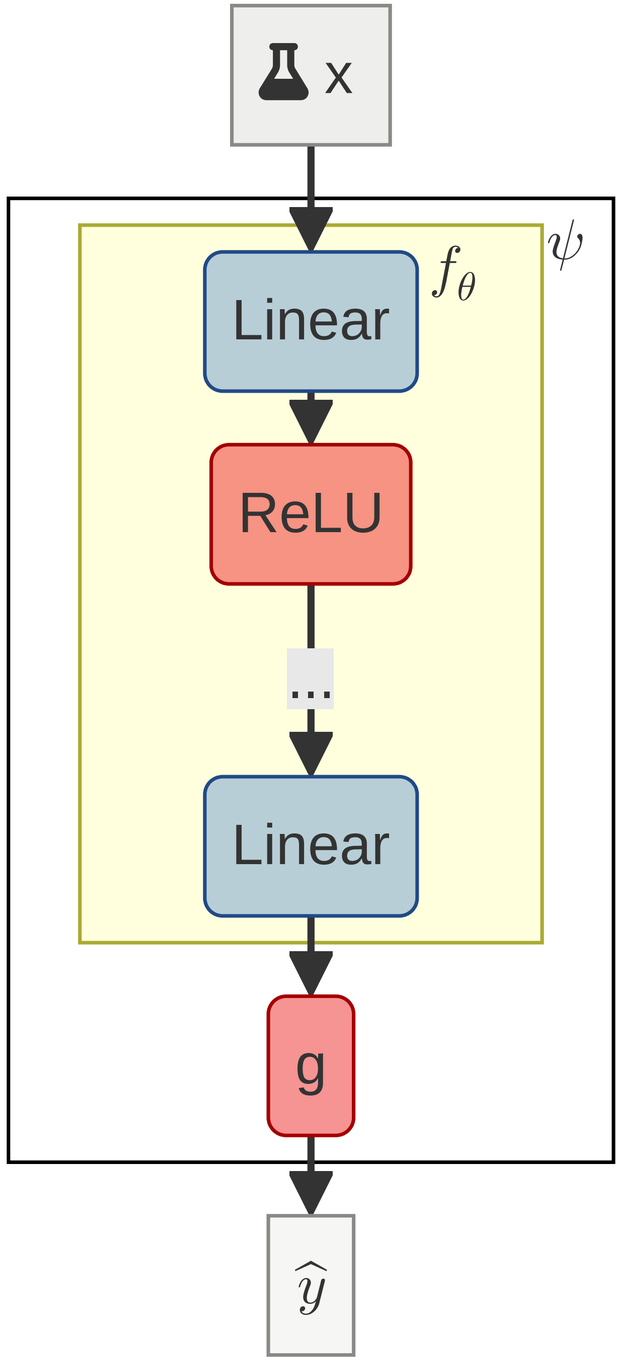
A two-layer deep network can approximate any **continuous** function.

- We might not always want continuous (real-valued) outputs
- How can we convert the real value to what we want?



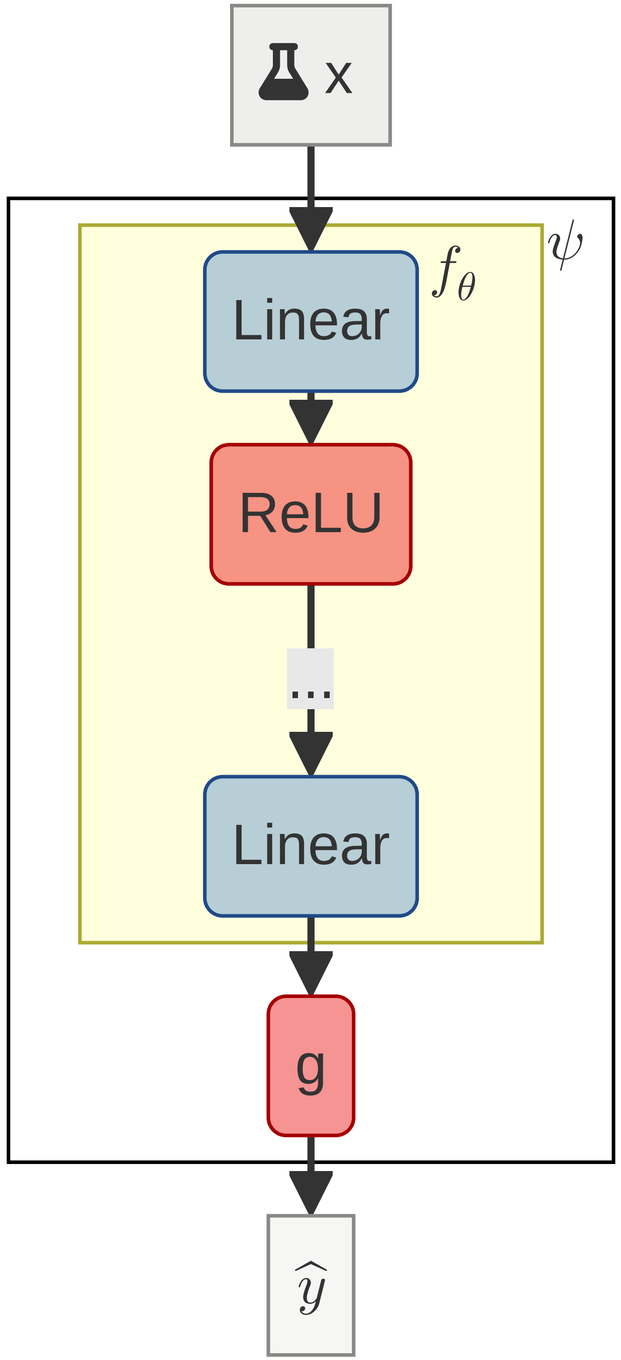
Inputs and Outputs of Networks

- Input: $x \in \mathbb{R}^n$
- Output: $o = f_\theta(x) \in \mathbb{R}^m$
  - Deep network f_θ
- Output transformations: g
  - $\psi = g \circ f_\theta$



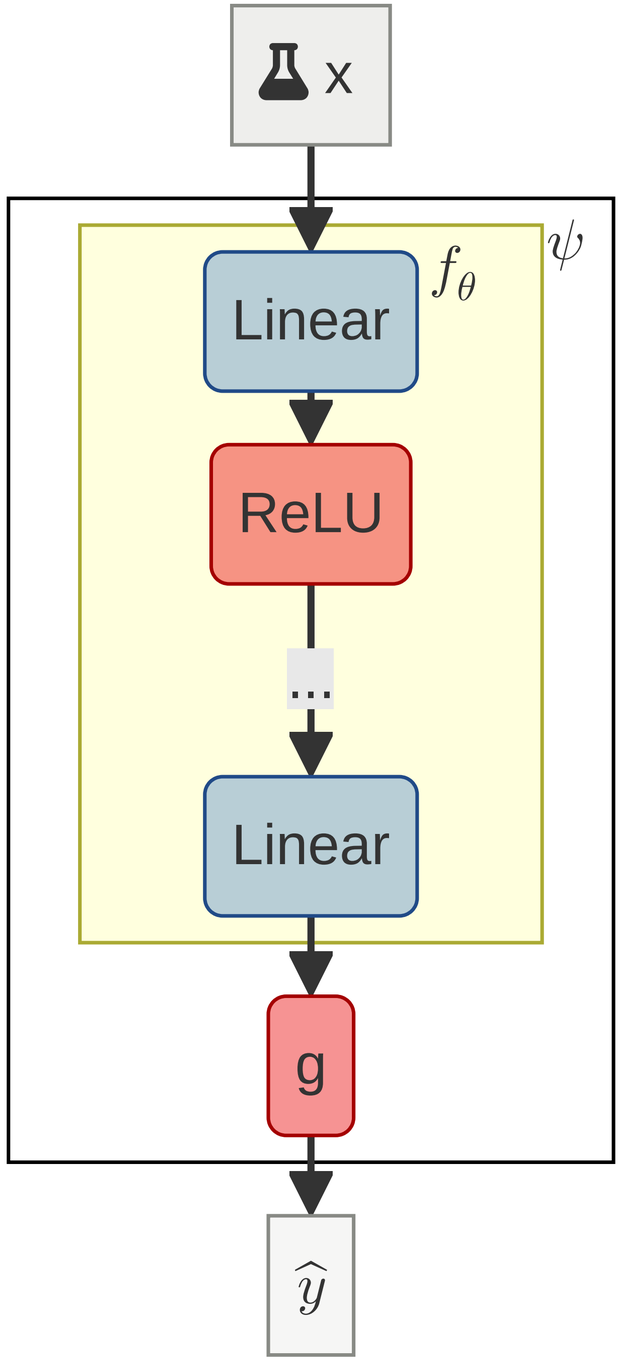
Regression

- $\psi : \mathbb{R}^n \to \mathbb{R}$
- Identity mapping: $g(o) = o$



Positive Regression

- $\psi : \mathbb{R}^n \to \mathbb{R}_+$
- Option 1: $g(o) = \text{ReLU}(o) = \max(o, 0)$
- Option 2: $g(o) = \log(1 + \exp(o))$

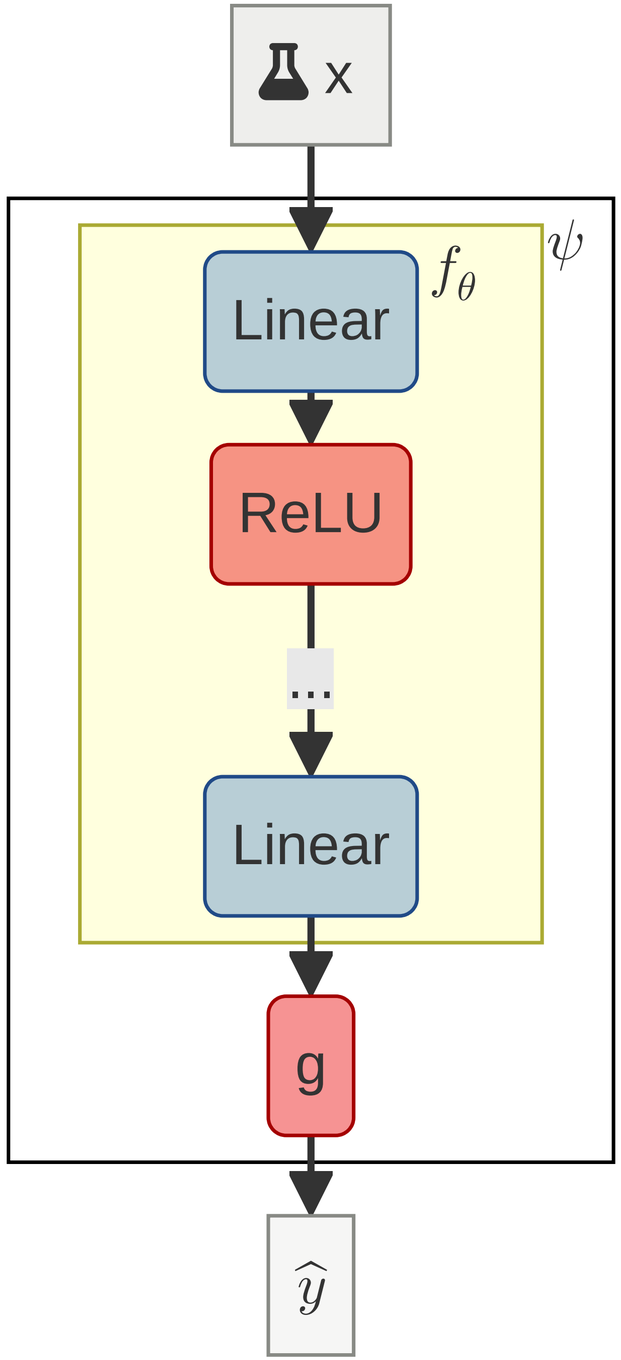


Binary Classification

- $\psi : \mathbb{R}^n \to \{0,1\}$
- Option 1: Thresholding

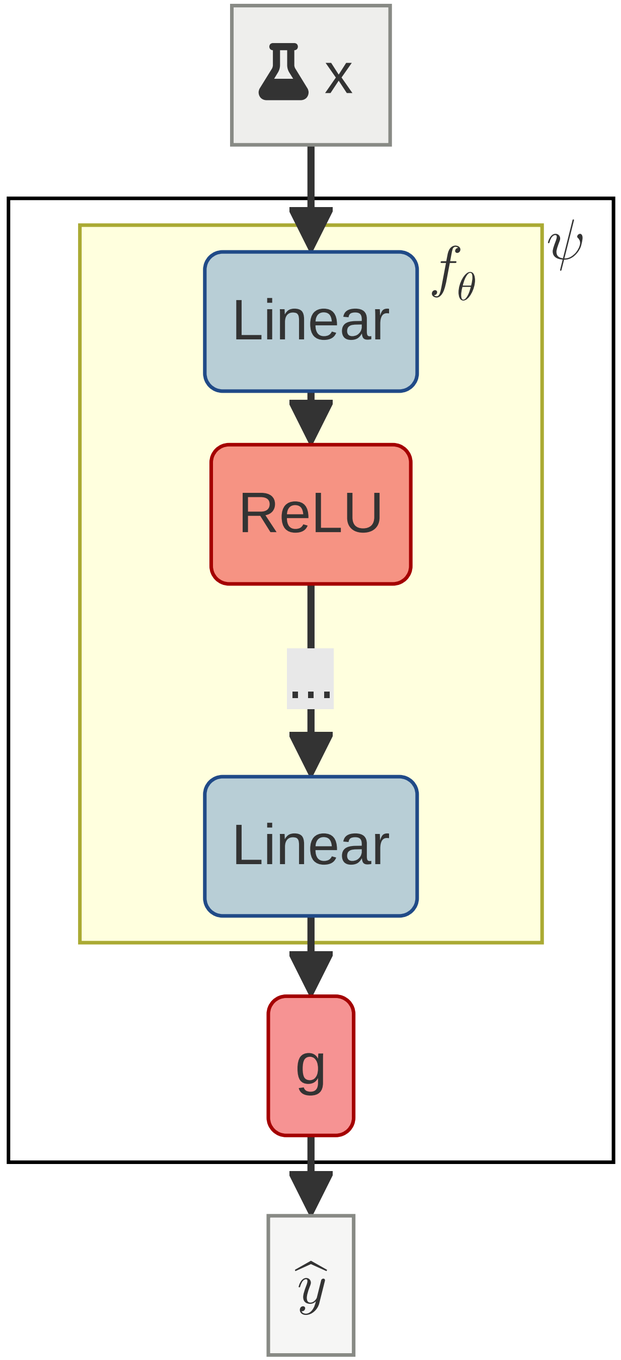
  $g(o) = [o > 0]$
- Option 2: Sigmoid

  $P(Y = 1) = g(o) = \sigma(o) = \dfrac{1}{1 + \exp(-x)}$



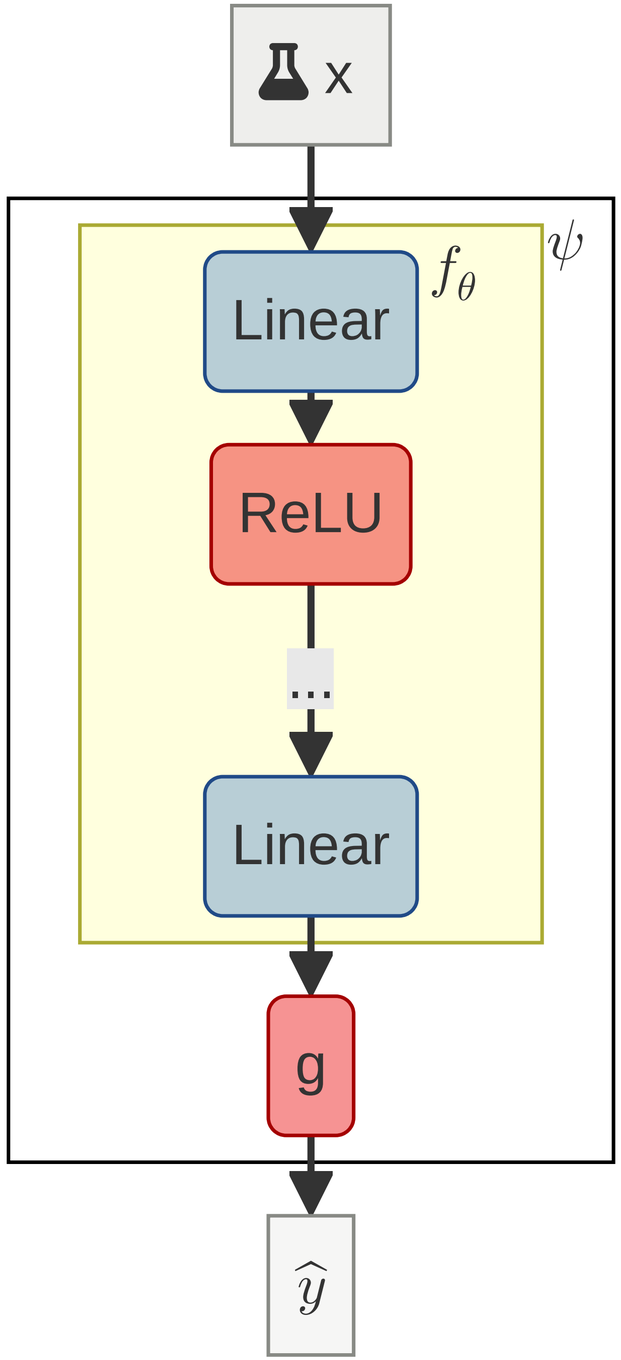
Multi-class Classification

- $\psi : \mathbb{R}^n \rightarrow \{1, 2, \ldots, C\}$
- Option 1: Argmax
  - $g(o) = \arg\max(o)$
- Option 2: Indicator
  - $g(o) = [[o_1 = \max(o)], [o_2 = \max(o)], \ldots]^\top$
- Option 3: Softmax
  - $P(y) = g(o) = \text{softmax}(o)$



Output Representations in Practice

- Do **not** add to model
  - Most output transformations are not differentiable (or hard to differentiate)
  - Model cannot train with them
- Always output raw values



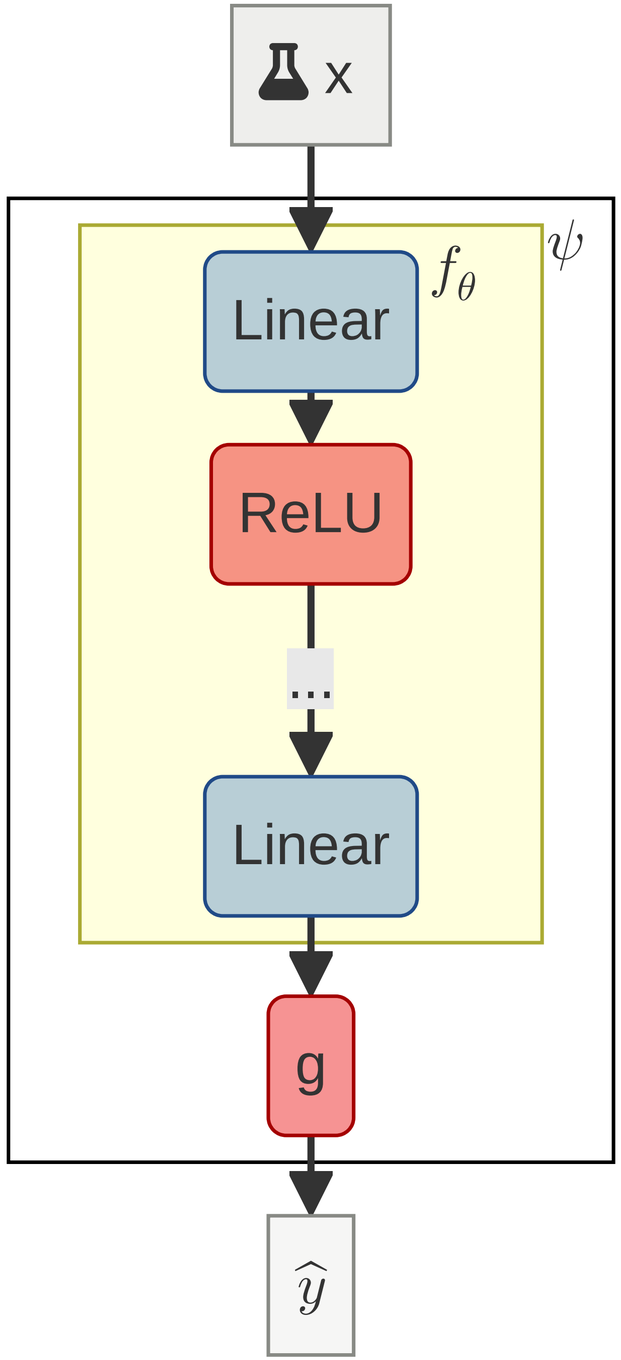
Output Representations - TL;DR

- Deep networks always output real values
- Output transformations convert them into what you want
- Train the network without output transformations!

Loss Functions

Recap: Output transformations

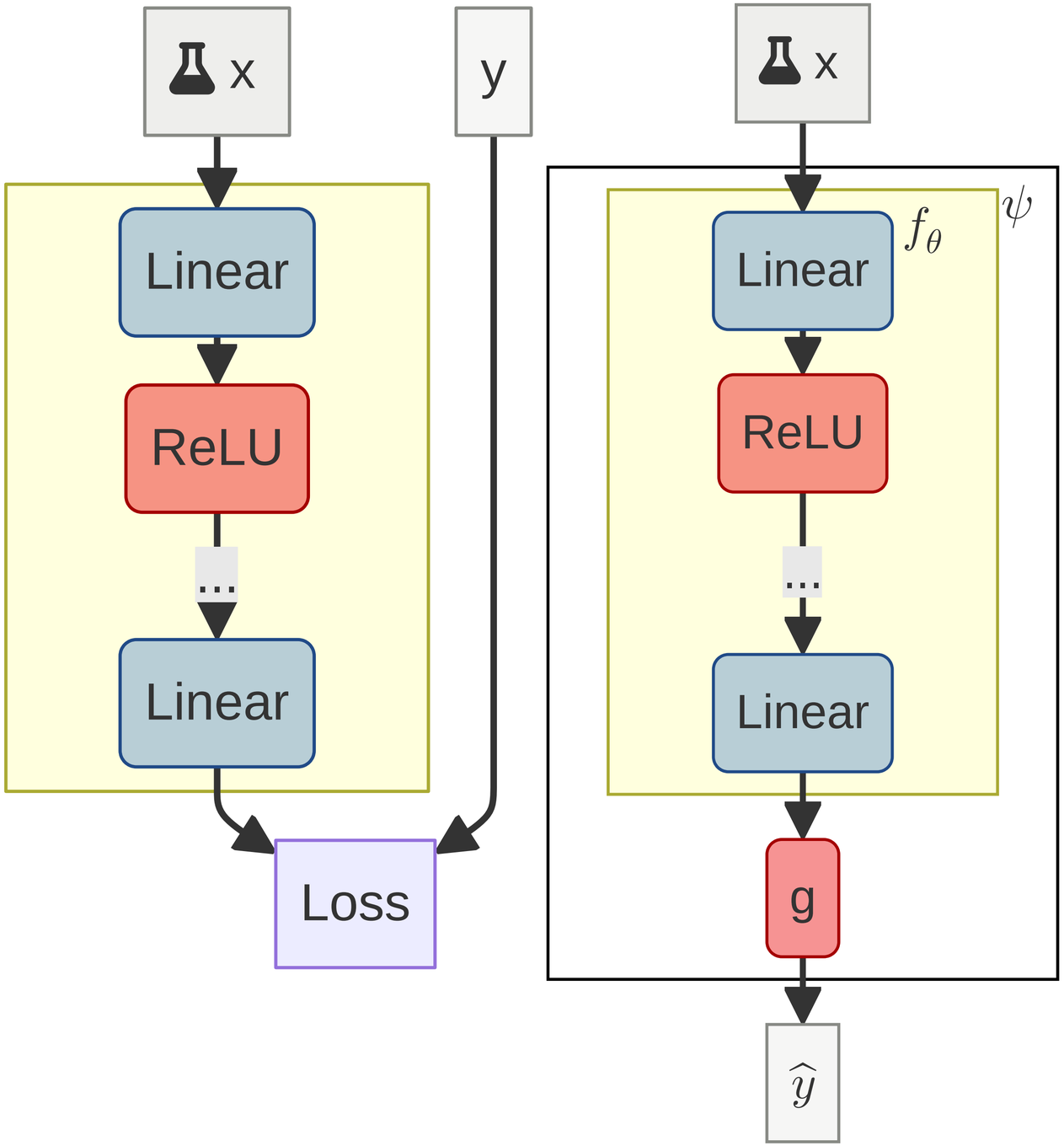
- Input: $x \in \mathbb{R}^n$
- Output: $o = f_\theta(x) \in \mathbb{R}^m$
  - Deep network f_θ
- Output transformations: g
  - $\psi = g \circ f_\theta$
- Do **not** add to model: does not train



Training deep networks

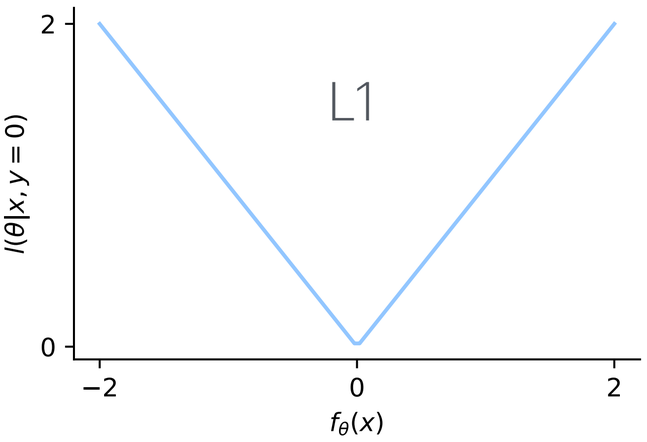
Loss functions

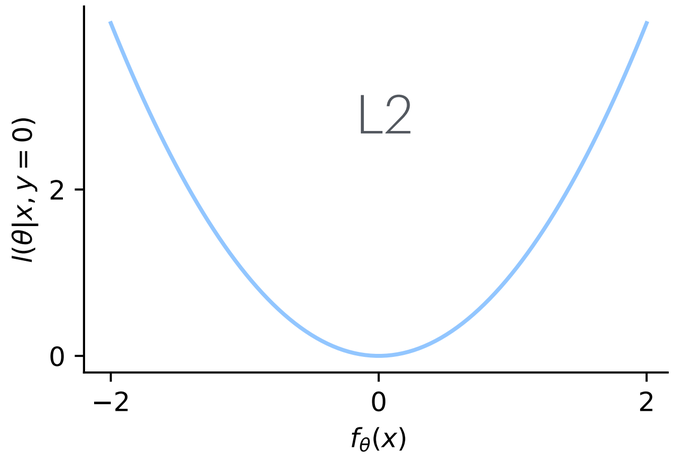
- Loss function: $\ell(\theta \mid x, y)$
- Expected Loss:
  $L(\theta \mid \mathcal{D}) = E_{x,y \sim \mathcal{D}} \left[\ell(\theta \mid x, y) \right]$

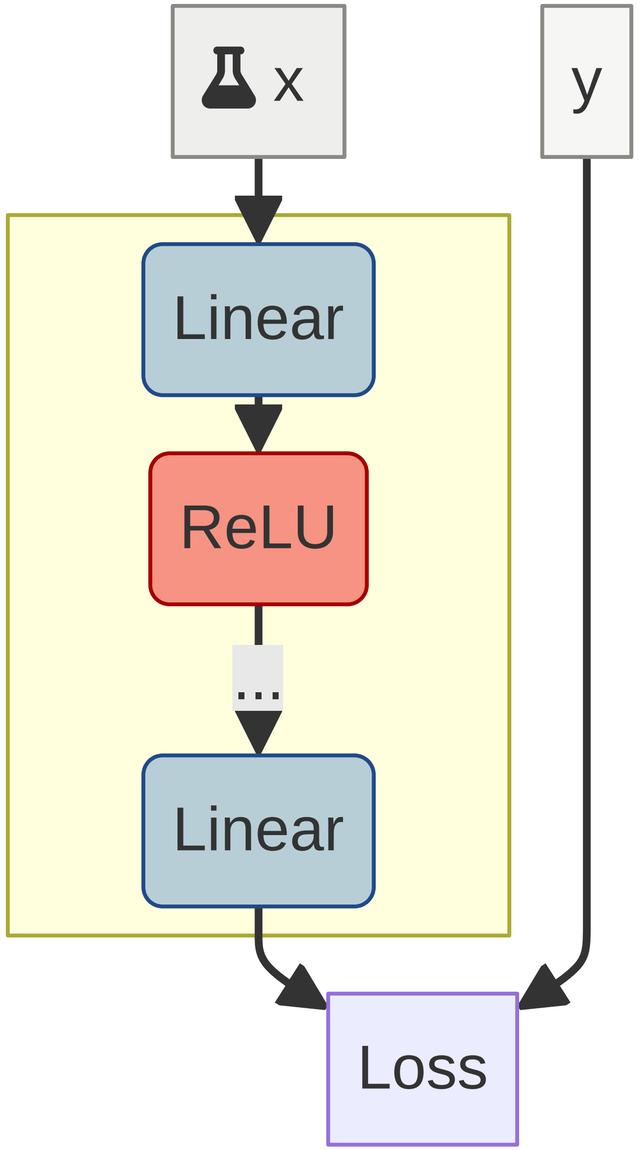


Regression

- $\psi : \mathbb{R}^n \to \mathbb{R}$
- Identity mapping: $g(o) = o$
- L1 loss function: $\ell(\theta \,|\, x, y) = |f(x) - y|$
  L2 loss function: $\ell(\theta \,|\, x, y) = |f(x) - y|^2$



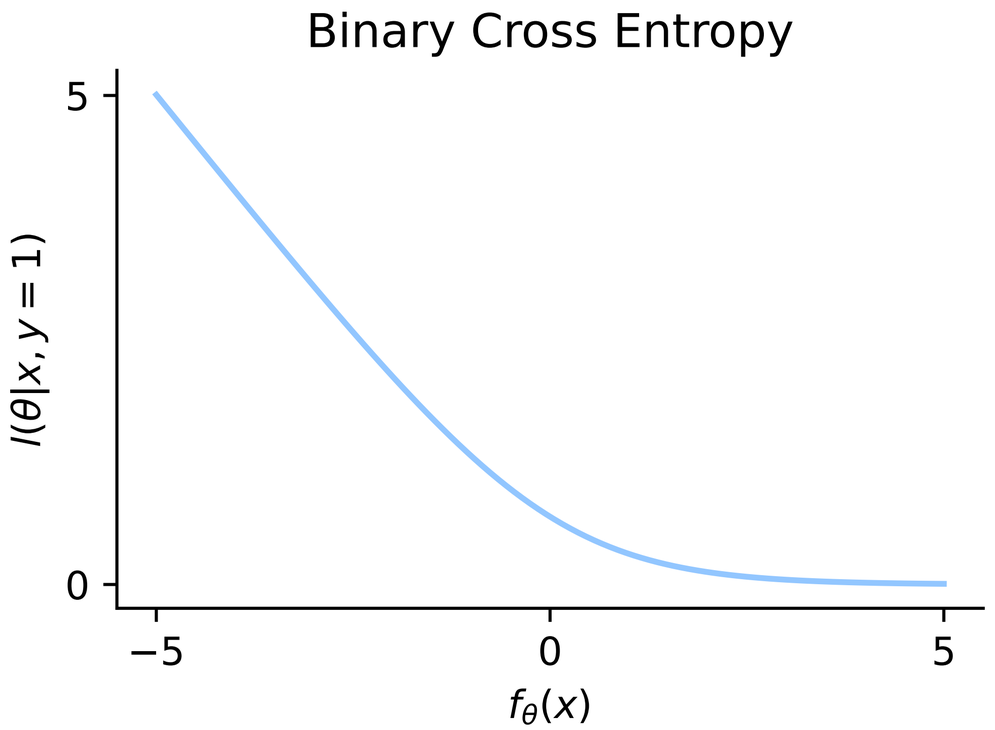




Binary Classification

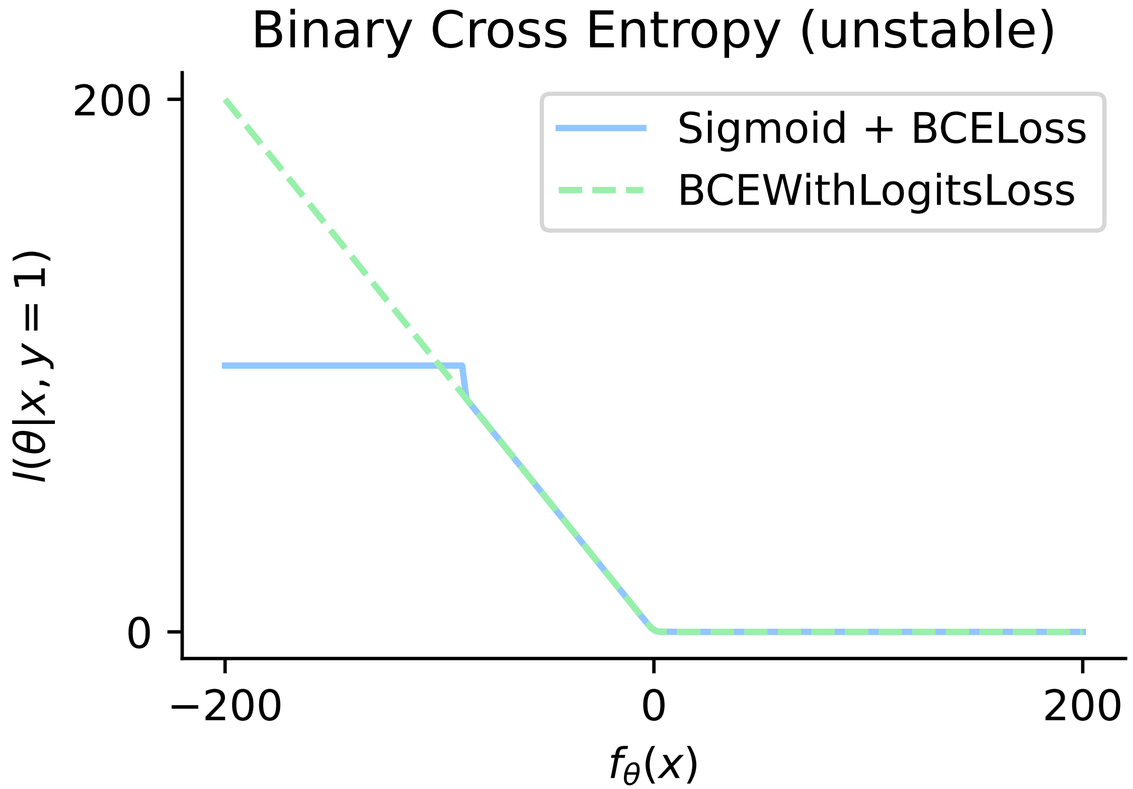
- $\psi : \mathbb{R}^n \to \{0,1\}$
- Output transformation: $g(o) = [o > 0]$
- Binary cross entropy loss function:

$$
\begin{aligned}
p(0) &= 1 - \sigma(f(x)) \\
p(1) &= \sigma(f(x)) \\
\ell(\theta \,|\, x, y) &= -\, y \log p(1) - (1 - y) \log p(0) \\
&= -\, y \log \sigma(f(x)) - (1 - y) \log(1 - \sigma(f(x))
\end{aligned}
$$



Binary Classification in practice

- Use `BCEWithLogitsLoss`



Multi-class Classification

- $\psi : \mathbb{R}^n \to \{1, 2, \ldots, C\}$
- Output transformation: $g(o) = \arg\max(o)$
- Cross-entropy loss:

$$P = \text{softmax}(f(x))$$
$$\ell(\theta \mid x, y) = -\log P(y)$$

- Use `CrossEntropyLoss`

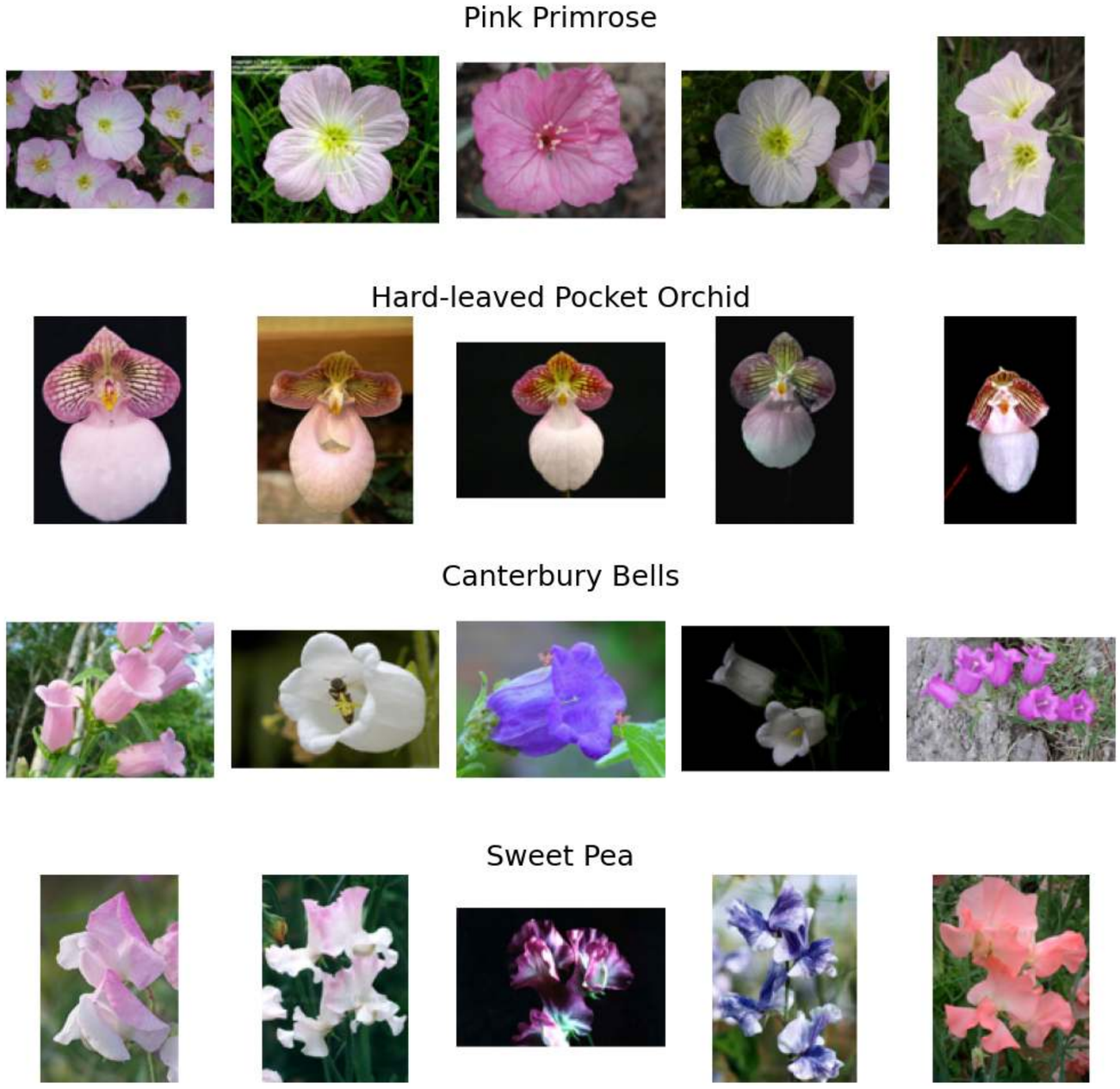
Loss Functions - TL;DR

- Regression: L1 loss `torch.nn.L1Loss`, L2 loss `torch.nn.MSELoss`
- Binary classification: binary cross-entropy loss `torch.nn.BCEWithLogitsLoss`
- Multi-class classification: cross-entropy loss `torch.nn.CrossEntropyLoss`
- Always use PyTorch loss for better numerical stability!

Stochastic Gradient Descent

Recap: Data

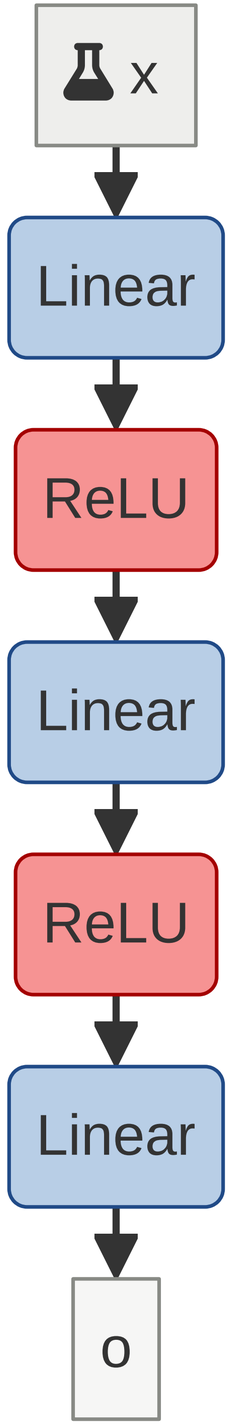
- Input: $\mathbf{x}_i$
- Label: $\mathbf{y}_i$
- Dataset: $\mathscr{D} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots\}$



Recap: Deep Networks

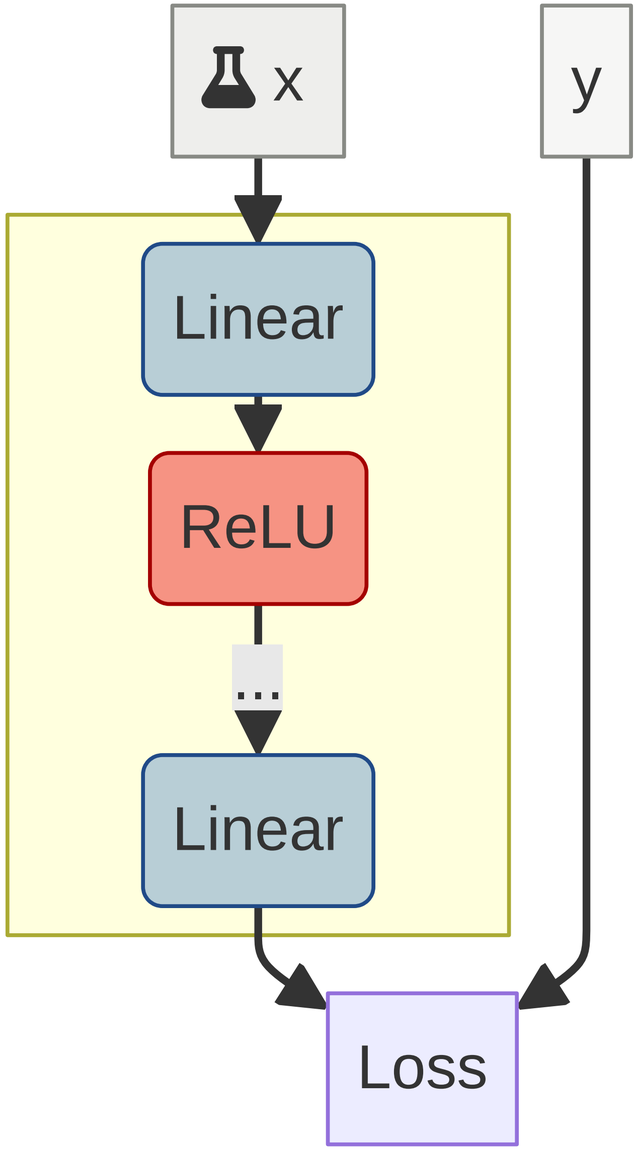
- Model: $f_\theta : \mathbb{R}^n \to \mathbb{R}^c$
- Parameters: $\theta = (\mathbf{W}_1, \mathbf{b}_1, \ldots, \mathbf{W}_N, \mathbf{b}_N)$
- Computation:

$$\mathbf{z}_1 = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{z}_2 = \text{ReLU}(\mathbf{W}_2\mathbf{z}_1 + \mathbf{b}_2)$$
$$\vdots$$
$$\mathbf{z}_{N-1} = \text{ReLU}(\mathbf{W}_{N-1}\mathbf{z}_{N-2} + \mathbf{b}_{N-1})$$
$$f_\theta(\mathbf{x}) = \mathbf{W}_N\mathbf{z}_{N-1} + \mathbf{b}_N$$



Recap: Loss

- Loss function: $\ell(\theta \mid x, y)$
- Expected Loss:
  $L(\theta \mid \mathcal{D}) = E_{x,y \sim \mathcal{D}} \left[\ell(\theta \mid x, y) \right]$



Recap: Gradient Descent

- Deep network $f_\theta : \mathbb{R}^N \to \mathbb{R}^M$
- Dataset: $\mathscr{D} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots\}$
- Expected Loss:
  $L(\theta \mid \mathscr{D}) = E_{x,y\sim\mathscr{D}}\left[\ell(\theta \mid x, y)\right]$

Gradient $\nabla L(\theta \mid D) = E_{x,y\sim\mathscr{D}}\left[\nabla \ell(\theta \mid x, y)\right]$

```
θ ~ Init
for iteration in range(n):
    J = ∇L(θ)
    θ = θ - ϵ * J.mT
```

Recap: Gradient Descent

Gradient $\nabla L(\theta \mid D) = E_{x,y\sim\mathcal{D}}\left[\nabla \ell(\theta \mid x, y)\right]$

- Deep network $f_\theta : \mathbb{R}^N \to \mathbb{R}^M$
- Dataset: $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots\}$
- Expected Loss:
  $L(\theta \mid \mathcal{D}) = E_{x,y\sim\mathcal{D}}\left[\ell(\theta \mid x, y)\right]$

```
θ ~ Init
for epoch in range(n):
  J = s0
  for (x, y) in dataset:
    J += ∇l(θ|x,y)
  θ = θ - ϵ * J.mT
```

How fast is gradient descent? What factors does the runtime depend on?

Gradient Descent

- Deep network $f_\theta : \mathbb{R}^N \to \mathbb{R}^M$
- Dataset: $\mathscr{D} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots\}$
- Expected Loss:
  $L(\theta \mid \mathscr{D}) = E_{x,y\sim\mathscr{D}}\left[\ell(\theta \mid x, y)\right]$

Gradient $\nabla L(\theta \mid D) = E_{x,y\sim\mathscr{D}}\left[\nabla \ell(\theta \mid x, y)\right]$

```
θ ~ Init
for epoch in range(n):
  J = s0
  for (x, y) in dataset:
    J += ∇l(θ|x,y)
  θ = θ - ϵ * J.mT
```

$$O(n_{\text{epoch}} \mid \mathscr{D} \mid \mid \theta \mid)$$

Stochastic Gradient Descent

Gradient $\nabla L(\theta \mid D) = E_{x,y \sim \mathcal{D}} \left[\nabla \ell(\theta \mid x, y) \right]$

- Update parameters after every gradient computation

```
θ ~ Init
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    θ = θ - ϵ * J.mT
```

Gradient Descent vs Stochastic Gradient Descent

Gradient Descent

```
θ ~ Init
for epoch in range(n):
  J = s0
  for (x, y) in dataset:
    J += ∇l(θ|x,y)
  θ = θ - ϵ * J.mT
```

Convergence guarantees

Smooth loss

Stochastic Gradient Descent

```
θ ~ Init
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    θ = θ - ϵ * J.mT
```

Faster convergence (empirically)

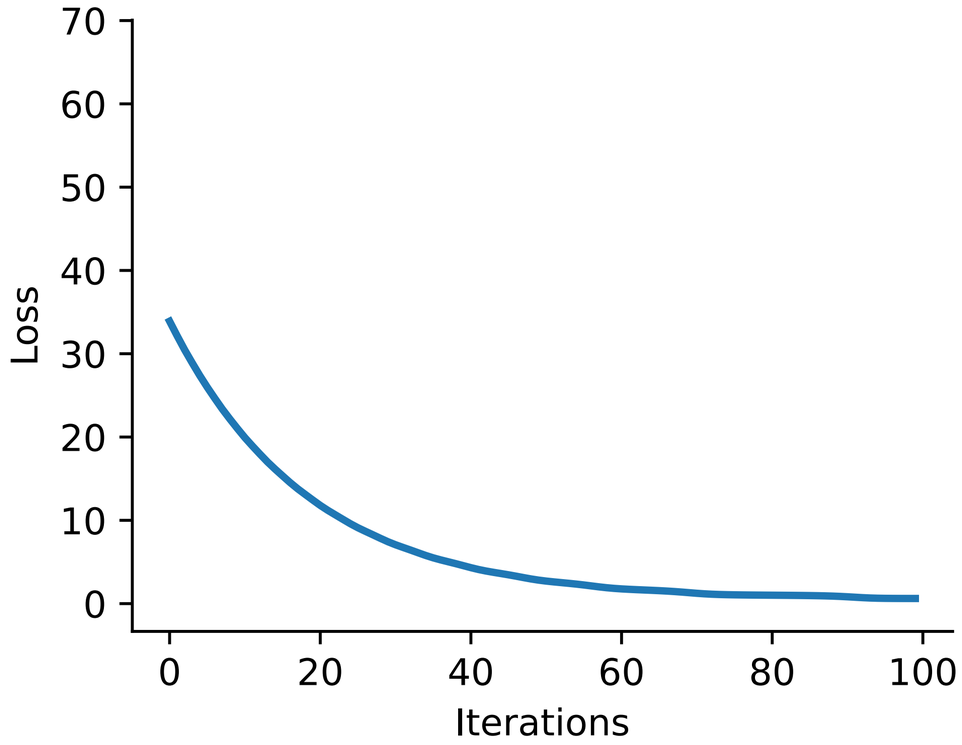
More chaotic convergence

Gradient Descent vs Stochastic Gradient Descent

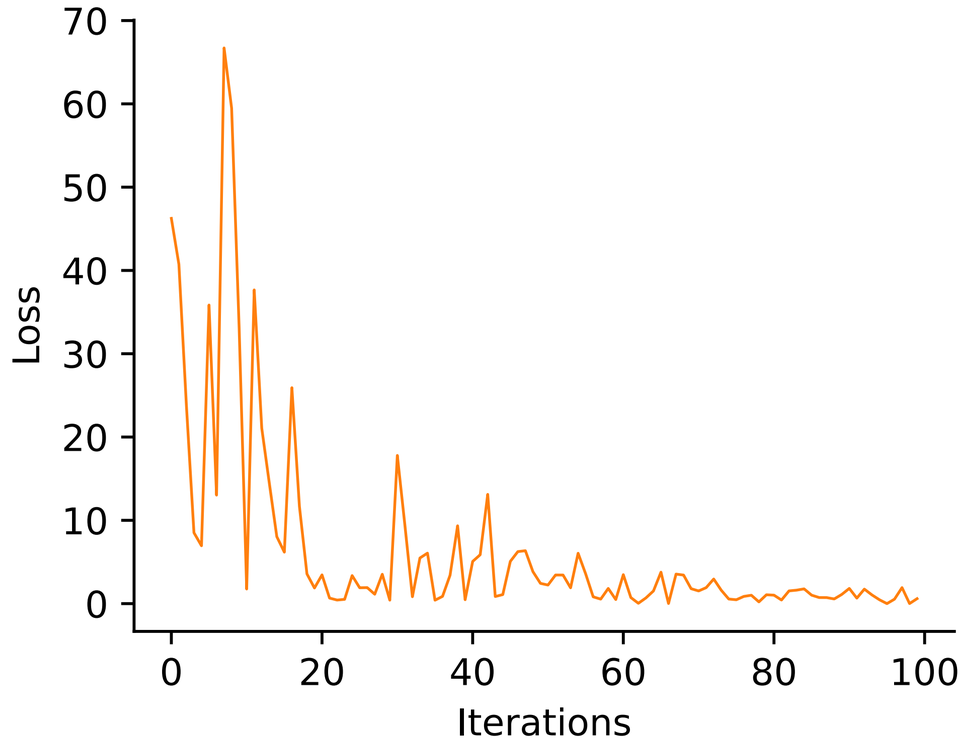
- Notebook example

Learning curves

Gradient Descent



Stochastic Gradient Descent

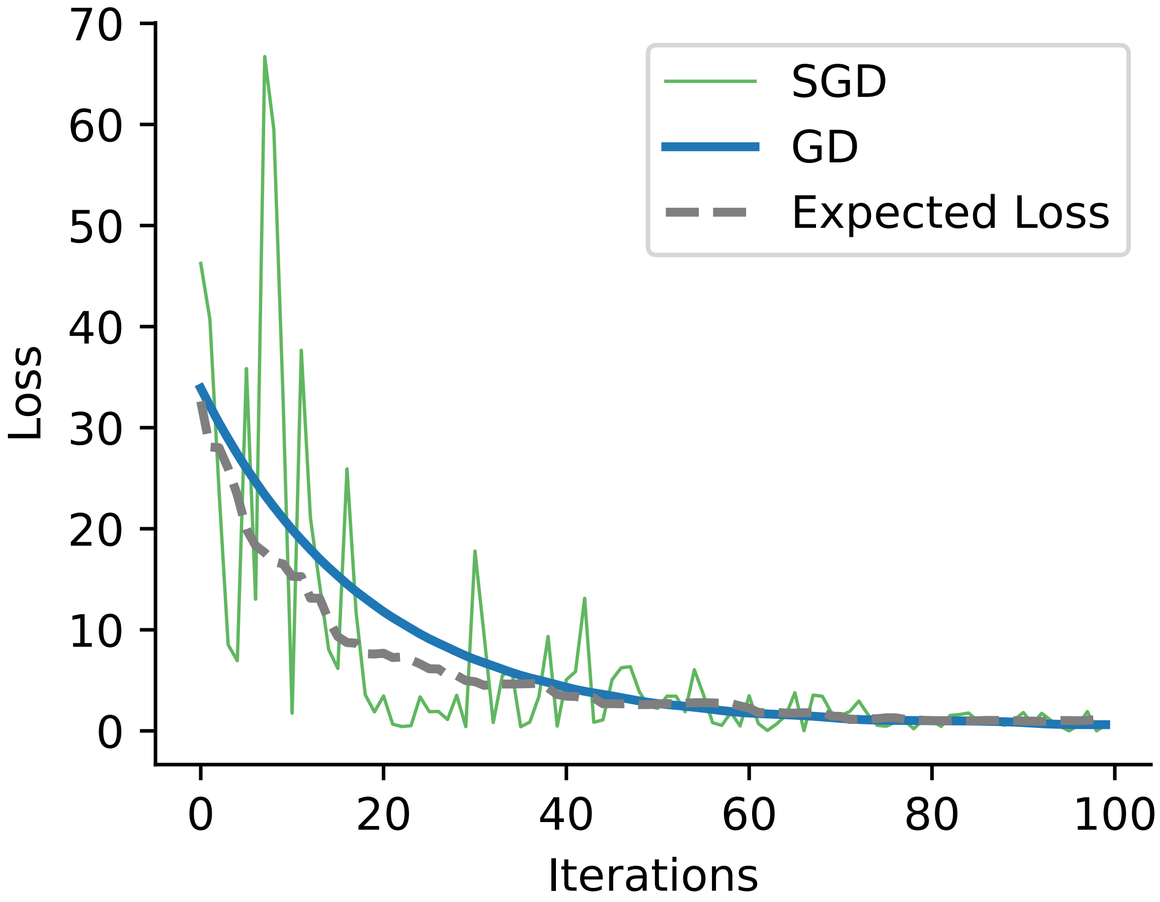


Learning curves

- Notebook example

Learning curves

- Why do learning curves fluctuate?
  - Loss evaluated on a single example
  - Gradient might be wrong



How fast does SGD converge?

Case 1: $\frac{\partial}{\partial\theta}l(\theta|x_i,y_i) \approx \frac{\partial}{\partial\theta}l(\theta|x_j,y_j)$

- SGD is equivalent to GD
- Faster

Case 2 (reality): $\frac{\partial}{\partial\theta}l(\theta|x_i,y_i) \neq \frac{\partial}{\partial\theta}l(\theta|x_j,y_j)$

- Convergences speed depends on variance of SGD[1]:

$$\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\mathcal{D}}\left[\left(\frac{\partial l(\theta|\mathbf{x},\mathbf{y})}{\partial\theta} - \frac{\partial L(\theta|\mathcal{D})}{\partial\theta}\right)^2\right]$$

$$= \mathbb{E}_{\mathbf{x},\mathbf{y}\sim\mathcal{D}}\left[\left(\frac{\partial l(\theta|\mathbf{x},\mathbf{y})}{\partial\theta}\right)^2\right] - \left(\frac{\partial L(\theta|\mathcal{D})}{\partial\theta}\right)^2$$

1. Léon Bottou, Stochastic Gradient Descent Tricks. in Neural Networks, Tricks of the Trade, Reloaded ↩

How can we make SGD converge faster?

- Mini-batches
  - Average gradient of subset of datapoints
- Momentum
  - Running average of gradients

How can we make SGD converge faster?

Stochastic Gradient Descent (with Mini-Batch)

```
for epoch in range(n):
  for i in range(len(dataset) // batch_size):
    J = 0
    batch = dataset[i * batch_size: (i + 1) * batch_size]
    for (x, y) in batch:
      J += ∇l(θ|x,y)
    θ = θ - ϵ * J.mT
```

Stochastic Gradient Descent (with Momentum)

```
b = 0
for epoch in range(n):
  for (x, y) in dataset:
    b = ∇l(θ|x,y) + momentum * b
    θ = θ - ϵ * b.mT
```

How can we make SGD converge faster?

- Mini-batches
  - How big should your mini-batch be?
- Momentum
  - What would your momentum be?

```
b = 0
for epoch in range(n):
  for i in range(len(dataset) // batch_size):
    batch = dataset[i * batch_size : (i + 1) * batch_size]
    for (x, y) in batch:
      b = ∇l(θ|x,y) + momentum * b
      θ = θ - ϵ * b.mT
```

How can we make SGD converge faster?

- Mini-batches
  - How big should your mini-batch be?
    - Most often: **As large as will fit your GPU**
- Momentum
  - What would your momentum be? **0.9**

```
b = 0
for epoch in range(n):
  for i in range(len(dataset) // batch_size):
    batch = dataset[i * batch_size : (i + 1) * batch_size]
    for (x, y) in batch:
      b = ∇l(θ|x,y) + momentum * b
      θ = θ - ϵ * b.mT
```

How can we make SGD converge faster?

- Notebook example

Stochastic Gradient Descent - TL;DR

- We use stochastic gradient descent (SGD) to optimize deep networks
- SGD runs more **efficiently** but has higher variance than GD

Stochastic Gradient Descent in PyTorch

Training a Deep Network in PyTorch

Hyperparameters

Stochastic Gradient Descent

- Model Parameters
  - Weights / Bias of network
- What are other parameters can we change?

```
θ ~ Init
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    θ = θ - ϵ * J.mT
```

Hyperparameters

- Hyperparameter = any parameter not learned by SGD
- Number of epochs
- Learning rate
- Model architecture / size
- Loss function

```
θ ~ Init
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    θ = θ - ϵ * J.mT
```

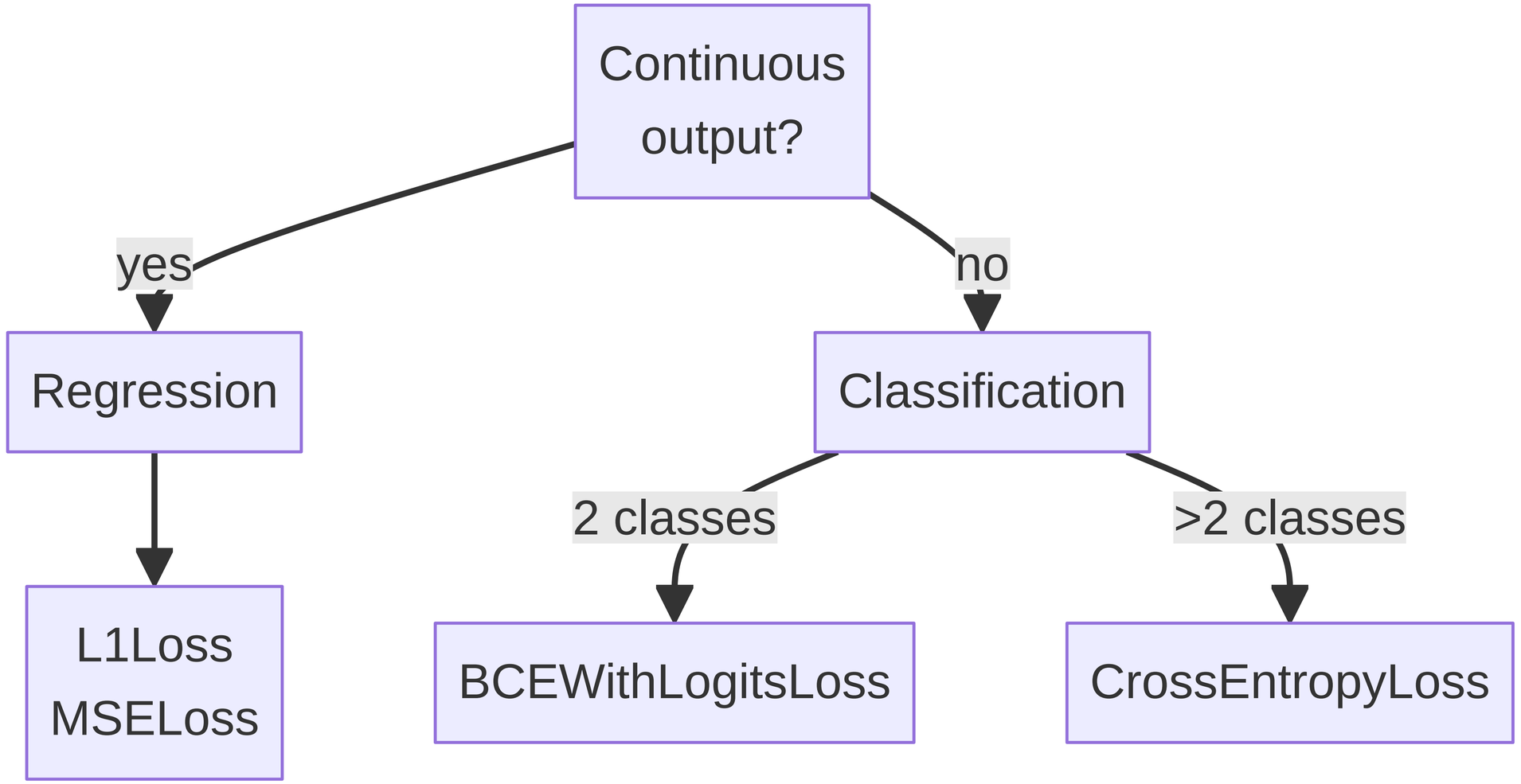
How do we set hyperparameters?

```
θ ~ Init
for epoch in range(n):
  for (x, y) in dataset:
    J = ∇l(θ|x,y)
    θ = θ - ϵ * J.mT
```

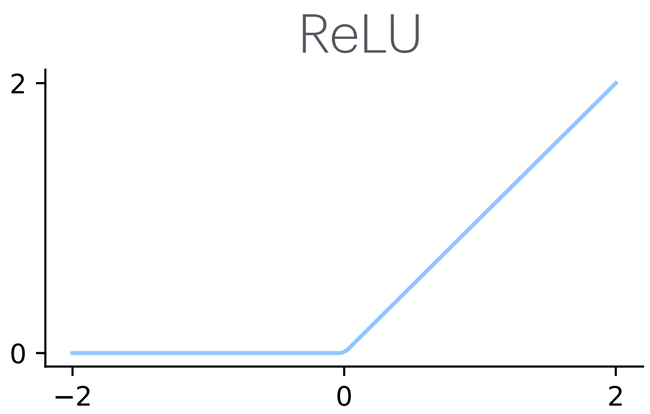
- By hand (tuning)
  - Using values that have worked for similar tasks
  - Through iterative experimentation
  - “Graduate Student Descent"

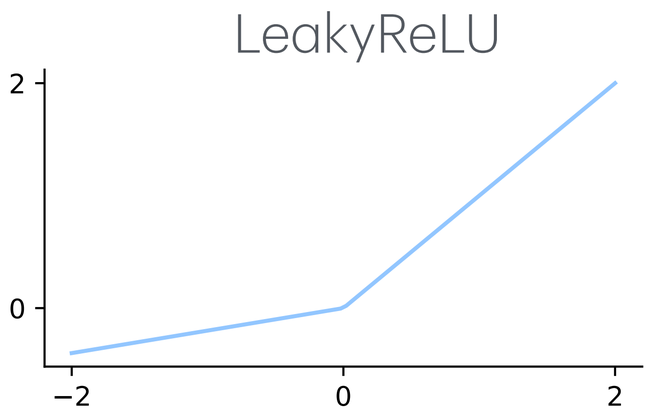
A Practical Guide to Deep Network Design

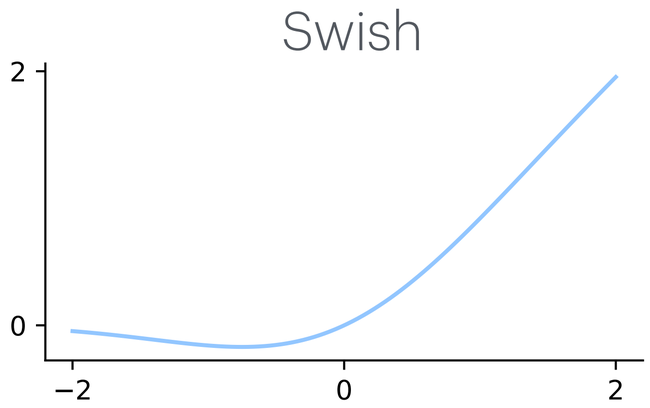
Loss Functions

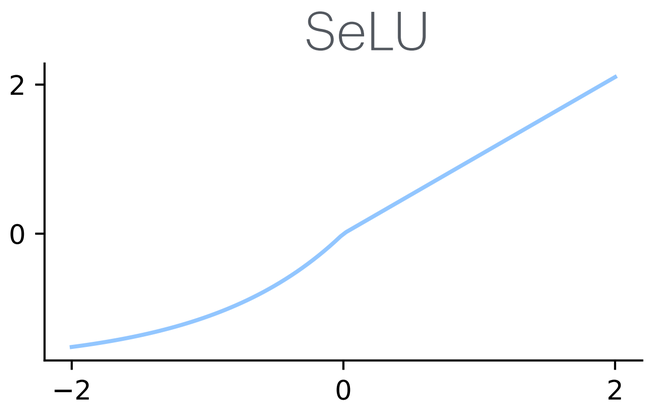


- Try ReLU
- If ReLU fails, try:
  - Leaky ReLU / PReLU
  - Swish for sophisticated models









SGD

- Select largest possible batch size
- momentum = 0.9
- Tune learning rate

```
b = 0
for epoch in range(n):
  for i in range(len(dataset) // batch_size):
    batch = dataset[i * batch_size : (i + 1) * batch_size]
    for (x, y) in batch:
      b = ∇l(θ|x,y) + momentum * b
      θ = θ - ϵ * b.mT
```