Modern Techniques

CISC 7026: Introduction to Deep Learning

University of Macau

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Agenda

1. **Review**
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Agenda

1. **Review**
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Agenda

1. Review
2. **Dirty secret of deep learning**
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Dirty Secret of Deep Learning

So far, I gave you the impression that deep learning is rigorous

Dirty Secret of Deep Learning

So far, I gave you the impression that deep learning is rigorous

Biological inspiration, theoretical bounds and mathematical guarantees

Dirty Secret of Deep Learning

So far, I gave you the impression that deep learning is rigorous

Biological inspiration, theoretical bounds and mathematical guarantees

For complex neural networks, deep learning is a **science** not **math**

Dirty Secret of Deep Learning

So far, I gave you the impression that deep learning is rigorous

Biological inspiration, theoretical bounds and mathematical guarantees

For complex neural networks, deep learning is a **science** not **math**

There is no widely-accepted theory for why deep neural networks are so effective

Dirty Secret of Deep Learning

So far, I gave you the impression that deep learning is rigorous

Biological inspiration, theoretical bounds and mathematical guarantees

For complex neural networks, deep learning is a **science** not **math**

There is no widely-accepted theory for why deep neural networks are so effective

In modern deep learning, we progress using trial and error

Dirty Secret of Deep Learning

So far, I gave you the impression that deep learning is rigorous

Biological inspiration, theoretical bounds and mathematical guarantees

For complex neural networks, deep learning is a **science** not **math**

There is no widely-accepted theory for why deep neural networks are so effective

In modern deep learning, we progress using trial and error

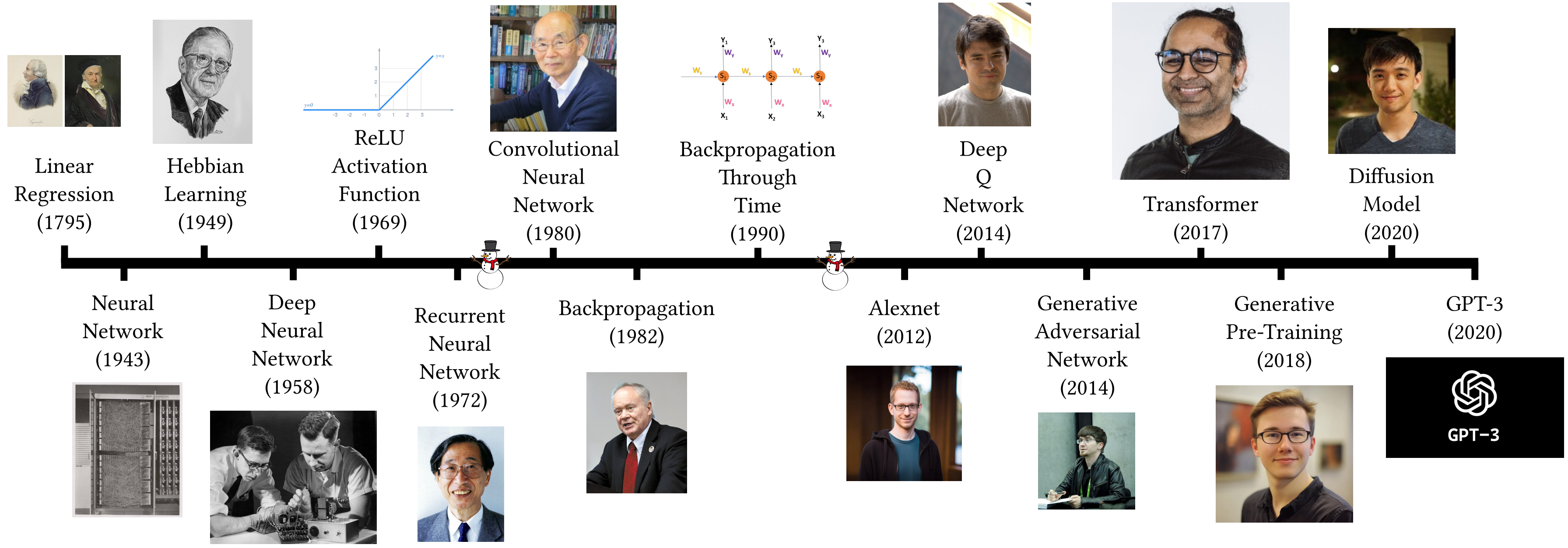
Today we experiment, and maybe tomorrow we discover the theory

Dirty Secret of Deep Learning

Similar to using neural networks for 40 years without knowing how to train them

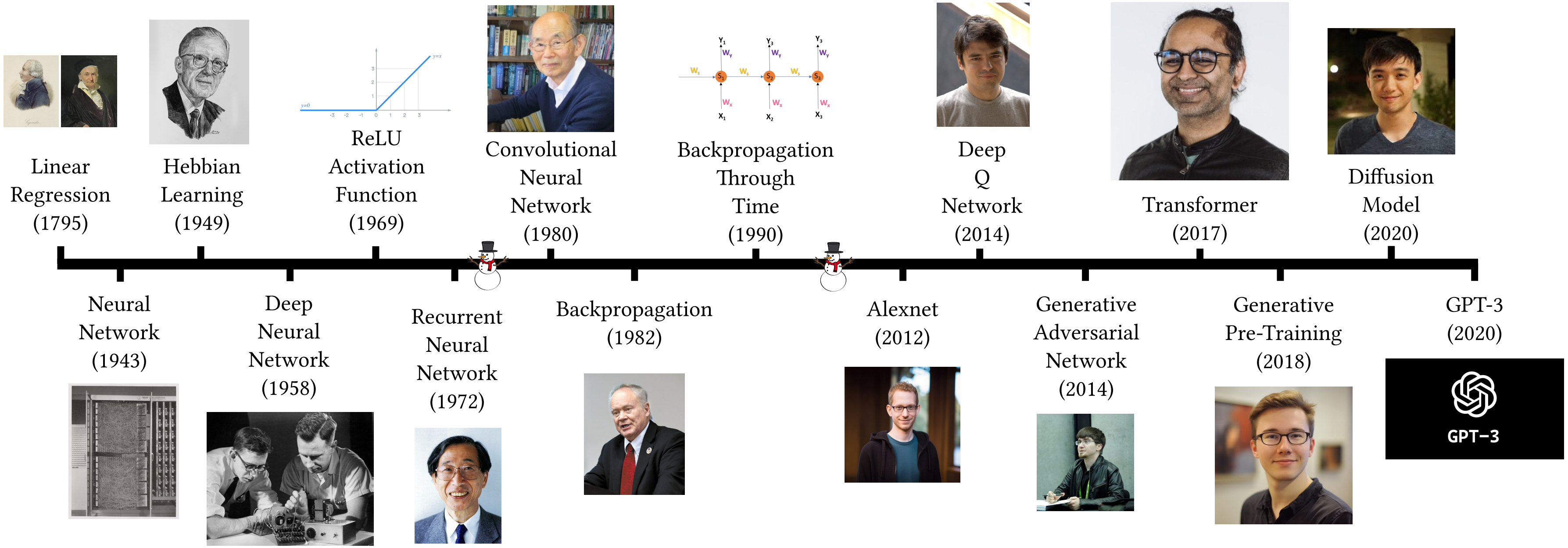
Dirty Secret of Deep Learning

Similar to using neural networks for 40 years without knowing how to train them



Dirty Secret of Deep Learning

Similar to using neural networks for 40 years without knowing how to train them



Are modern networks are too complex for humans to understand?

Dirty Secret of Deep Learning

Scientific method:

Dirty Secret of Deep Learning

Scientific method:

1. Collect observations

Dirty Secret of Deep Learning

Scientific method:

1. Collect observations
2. Form hypothesis

Dirty Secret of Deep Learning

Scientific method:

1. Collect observations
2. Form hypothesis
3. Run experiment

Dirty Secret of Deep Learning

Scientific method:

1. Collect observations
2. Form hypothesis
3. Run experiment
4. Publish theory

Dirty Secret of Deep Learning

Scientific method:

1. Collect observations
2. Form hypothesis
3. Run experiment
4. Publish theory

However, there is a second part:

Dirty Secret of Deep Learning

Scientific method:

1. Collect observations
2. Form hypothesis
3. Run experiment
4. Publish theory

However, there is a second part:

1. Find theory

Dirty Secret of Deep Learning

Scientific method:

1. Collect observations
2. Form hypothesis
3. Run experiment
4. Publish theory

However, there is a second part:

1. Find theory
2. Find counterexample

Dirty Secret of Deep Learning

Scientific method:

1. Collect observations
2. Form hypothesis
3. Run experiment
4. Publish theory

However, there is a second part:

1. Find theory
2. Find counterexample
3. Publish counterexample

Dirty Secret of Deep Learning

Scientific method:

1. Collect observations
2. Form hypothesis
3. Run experiment
4. Publish theory

However, there is a second part:

1. Find theory
2. Find counterexample
3. Publish counterexample
4. Falsify theory

Deep learning is new, so much of part 2 has not happened yet!

Dirty Secret of Deep Learning

For many concepts, the **observations** are stronger than the **theory**

Dirty Secret of Deep Learning

For many concepts, the **observations** are stronger than the **theory**

Observe that a concept improves many types of neural networks

Dirty Secret of Deep Learning

For many concepts, the **observations** are stronger than the **theory**

Observe that a concept improves many types of neural networks

Then, try to create a theory

Dirty Secret of Deep Learning

For many concepts, the **observations** are stronger than the **theory**

Observe that a concept improves many types of neural networks

Then, try to create a theory

Often, these theories are incomplete

Dirty Secret of Deep Learning

For many concepts, the **observations** are stronger than the **theory**

Observe that a concept improves many types of neural networks

Then, try to create a theory

Often, these theories are incomplete

If you do not believe the theory, prove it wrong and be famous!

Dirty Secret of Deep Learning

For many concepts, the **observations** are stronger than the **theory**

Observe that a concept improves many types of neural networks

Then, try to create a theory

Often, these theories are incomplete

If you do not believe the theory, prove it wrong and be famous!

Even if we do not agree on **why** a concept works, if we **observe** that it helps, we can still use it

Dirty Secret of Deep Learning

For many concepts, the **observations** are stronger than the **theory**

Observe that a concept improves many types of neural networks

Then, try to create a theory

Often, these theories are incomplete

If you do not believe the theory, prove it wrong and be famous!

Even if we do not agree on **why** a concept works, if we **observe** that it helps, we can still use it

This is how medicine works (e.g., Anesthetics)!

Agenda

1. Review
2. **Dirty secret of deep learning**
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Agenda

1. Review
2. Dirty secret of deep learning
3. **Optimization is hard**
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Optimization is Hard

A 2-layer neural network can represent **any** continuous function to arbitrary precision

Optimization is Hard

A 2-layer neural network can represent **any** continuous function to arbitrary precision

$$| \ f(\boldsymbol{x}, \boldsymbol{\theta}) - g(\boldsymbol{x}) \ | < \varepsilon$$

Optimization is Hard

A 2-layer neural network can represent **any** continuous function to arbitrary precision

$$| \; f(\boldsymbol{x}, \boldsymbol{\theta}) - g(\boldsymbol{x}) \; | < \varepsilon$$

$$\lim_{d_h \rightarrow \infty} \varepsilon = 0$$

Optimization is Hard

A 2-layer neural network can represent **any** continuous function to arbitrary precision

$$| \ f(\boldsymbol{x}, \boldsymbol{\theta}) - g(\boldsymbol{x}) \ | < \varepsilon$$

$$\lim_{d_h \rightarrow \infty} \varepsilon = 0$$

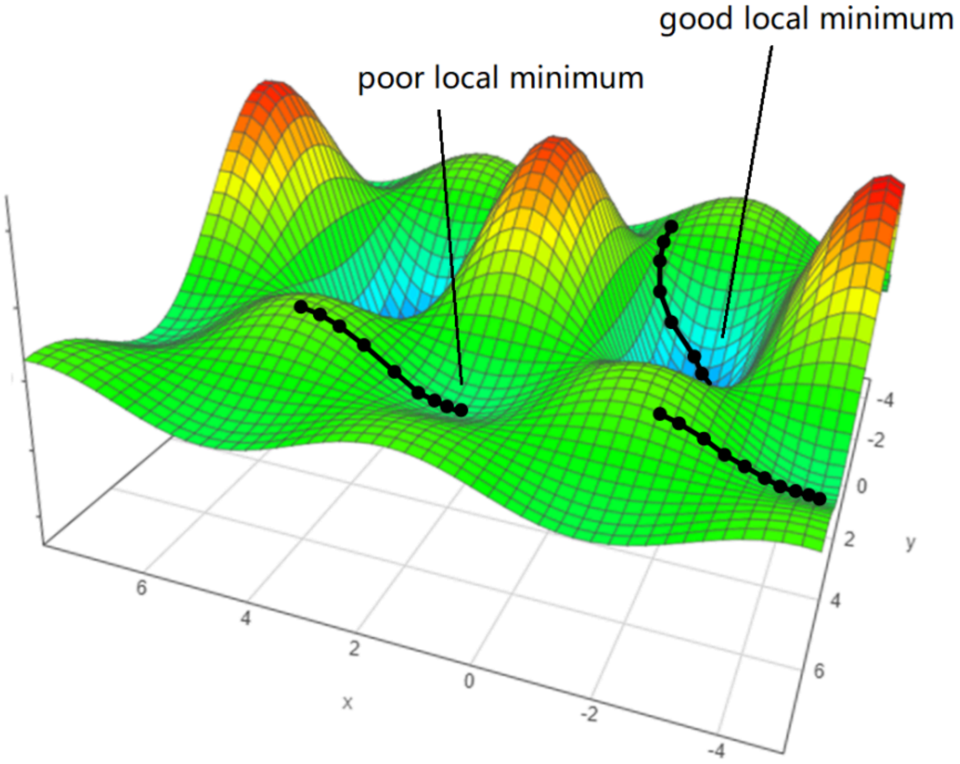
However, finding such $\boldsymbol{\theta}$ is a much harder problem

Optimization is Hard

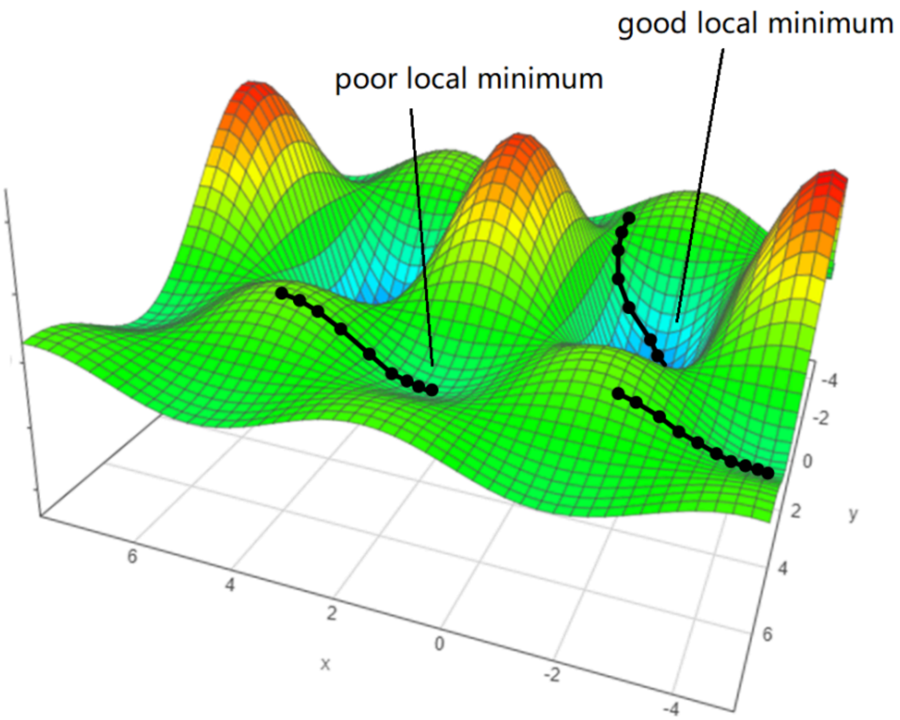
Gradient descent only guarantees convergence to a **local** optima

Optimization is Hard

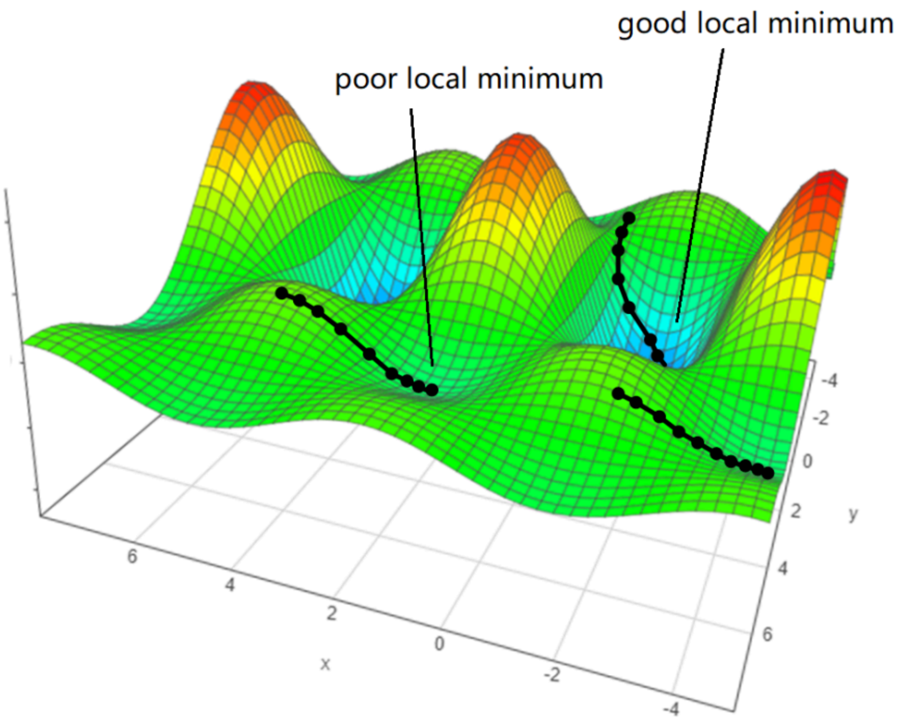
Gradient descent only guarantees convergence to a **local** optima



Optimization is Hard



Optimization is Hard



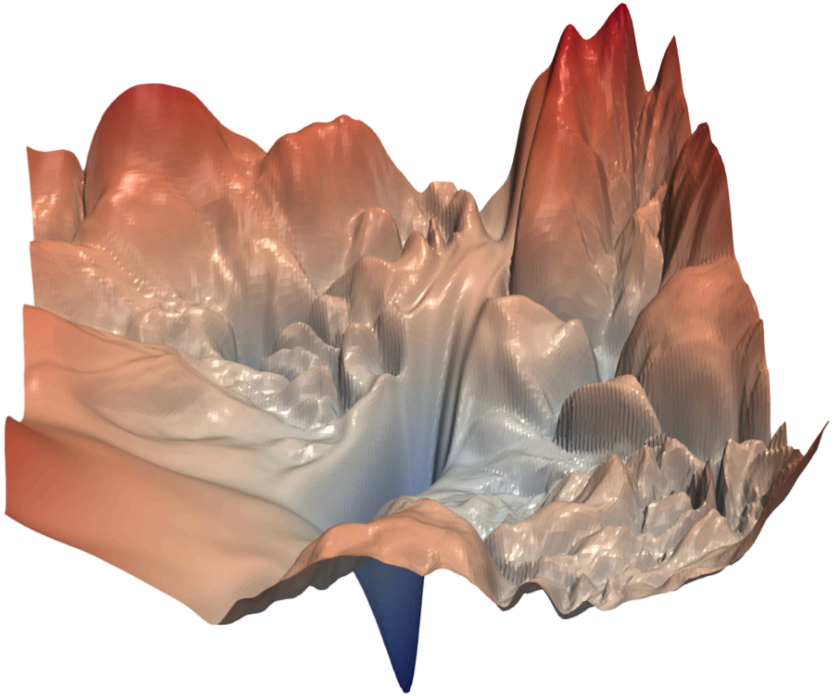
Harder tasks can have millions of local optima, and many of the local optima are not very good!

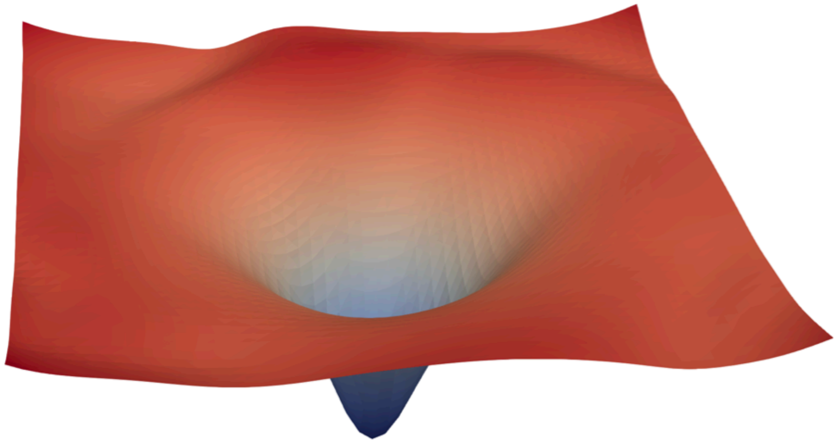
Optimization is Hard

Many of the concepts today create a **flat** loss landscape

Optimization is Hard

Many of the concepts today create a **flat** loss landscape





Gradient descent reaches a better optimum more quickly in these cases

Agenda

1. Review
2. Dirty secret of deep learning
3. **Optimization is hard**
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. **Deeper neural networks**
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Deeper Neural Networks

A two-layer neural network is sufficient to approximate any continuous function to arbitrary precision

Deeper Neural Networks

A two-layer neural network is sufficient to approximate any continuous function to arbitrary precision

But only with infinite width $d_h \rightarrow \infty$

Deeper Neural Networks

A two-layer neural network is sufficient to approximate any continuous function to arbitrary precision

But only with infinite width $d_h \to \infty$

For certain problems, adding one more layer is equivalent to **exponentially** increasing the width

- Eldan, Ronen, and Ohad Shamir. "The power of depth for feedforward neural networks." Conference on learning theory. PMLR, 2016.

$$2 \times 32 \times 32 \Rightarrow 2^{2 \times 32 \times 32} \approx 10^{616}; \quad \text{universe has } 10^{80} \text{ atoms}$$

Deeper Neural Networks

A two-layer neural network is sufficient to approximate any continuous function to arbitrary precision

But only with infinite width $d_h \to \infty$

For certain problems, adding one more layer is equivalent to **exponentially** increasing the width

- Eldan, Ronen, and Ohad Shamir. "The power of depth for feedforward neural networks." Conference on learning theory. PMLR, 2016.

$$2 \times 32 \times 32 \Rightarrow 2^{2 \times 32 \times 32} \approx 10^{616}; \quad \text{universe has } 10^{80} \text{ atoms}$$

We need more layers for harder problems

Deeper Neural Networks

In fact, we do not just need **deeper** networks, but also **wider** networks

Deeper Neural Networks

In fact, we do not just need **deeper** networks, but also **wider** networks

The number of neurons in a deep neural network affects the quality of local optima

Deeper Neural Networks

In fact, we do not just need **deeper** networks, but also **wider** networks

The number of neurons in a deep neural network affects the quality of local optima

From Choromanska, Anna, et al. "The loss surfaces of multilayer networks.":

Deeper Neural Networks

In fact, we do not just need **deeper** networks, but also **wider** networks

The number of neurons in a deep neural network affects the quality of local optima

From Choromanska, Anna, et al. "The loss surfaces of multilayer networks.":

- "For large-size networks, most local minima are equivalent and yield similar performance on a test set."

Deeper Neural Networks

In fact, we do not just need **deeper** networks, but also **wider** networks

The number of neurons in a deep neural network affects the quality of local optima

From Choromanska, Anna, et al. "The loss surfaces of multilayer networks.":

- "For large-size networks, most local minima are equivalent and yield similar performance on a test set."
- "The probability of finding a "bad" (high value) local minimum is non-zero for small-size networks and decreases quickly with network size"

Deeper Neural Networks

To summarize, deeper and wider neural networks tend to produce better results

Deeper Neural Networks

To summarize, deeper and wider neural networks tend to produce better results

Add more layers to your network

Deeper Neural Networks

To summarize, deeper and wider neural networks tend to produce better results

Add more layers to your network

Increase the width of each layer

Deeper Neural Networks

```python
# Deep neural network
from torch import nn

d_x, d_y, d_h = 1, 1, 16
# Linear(input, output)
l1 = nn.Linear(d_x, d_h)
l2 = nn.Linear(d_h, d_y)
```

Deeper Neural Networks

```python
# Deep neural network
from torch import nn

d_x, d_y, d_h = 1, 1, 16
# Linear(input, output)
l1 = nn.Linear(d_x, d_h)
l2 = nn.Linear(d_h, d_y)
```

```python
# Deeper and wider neural network
from torch import nn

d_x, d_y, d_h = 1, 1, 256
# Linear(input, output)
l1 = nn.Linear(d_x, d_h)
l2 = nn.Linear(d_h, d_h)
l3 = nn.Linear(d_h, d_h)
...
l6 = nn.Linear(d_h, d_y)
```

Deeper Neural Networks

```python
import torch
d_x, d_y, d_h = 1, 1, 256
net = torch.nn.Sequential([
    torch.nn.Linear(d_x, d_h),
    torch.nn.Sigmoid(),
    torch.nn.Linear(d_h, d_h),
    torch.nn.Sigmoid(),
    ...
    torch.nn.Linear(d_h, d_y),
])

x = torch.ones((d_x,))
y = net(x)
```

Deeper Neural Networks

```
import jax, equinox
d_x, d_y, d_h = 1, 1, 256
net = equinox.nn.Sequential([
    equinox.nn.Linear(d_x, d_h),
    equinox.nn.Lambda(jax.nn.sigmoid),
    equinox.nn.Linear(d_h, d_h),
    equinox.nn.Lambda(jax.nn.sigmoid),
    ...
    equinox.nn.Linear(d_h, d_y),
])

x = jax.numpy.ones((d_x,))
y = net(x)
```

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. **Deeper neural networks**
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Agenda

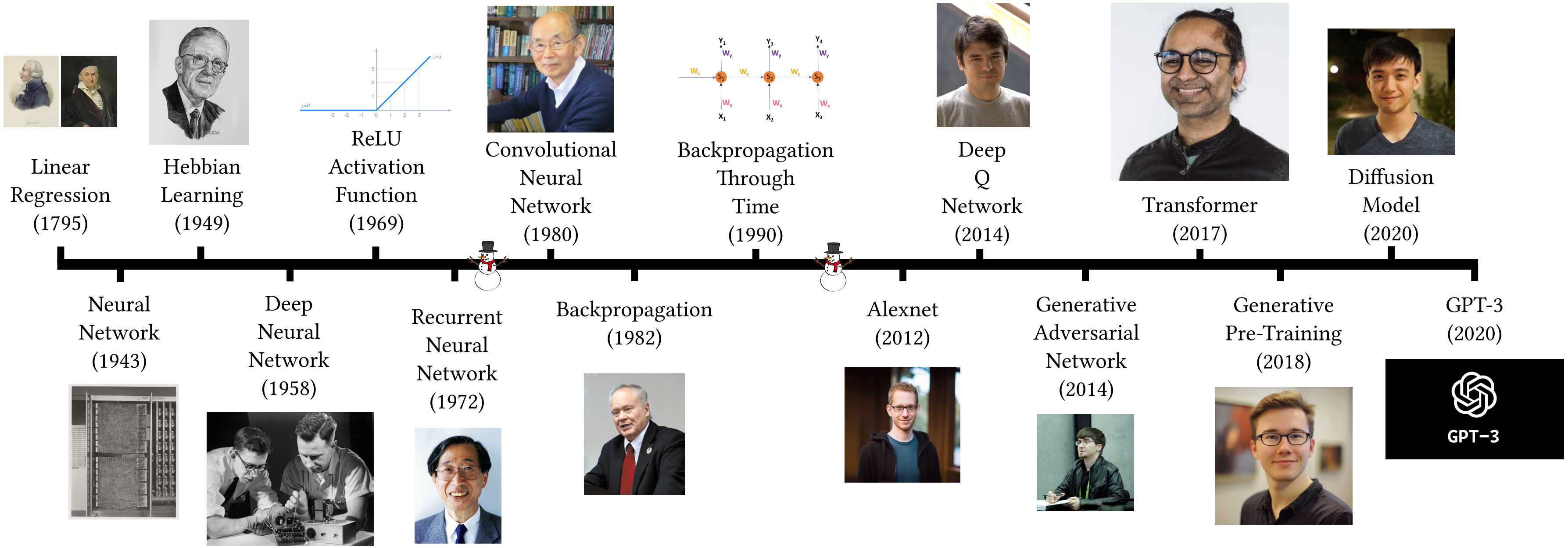
1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. **Activation functions**
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Activation Functions

The sigmoid function was the standard activation function until ~ 2012

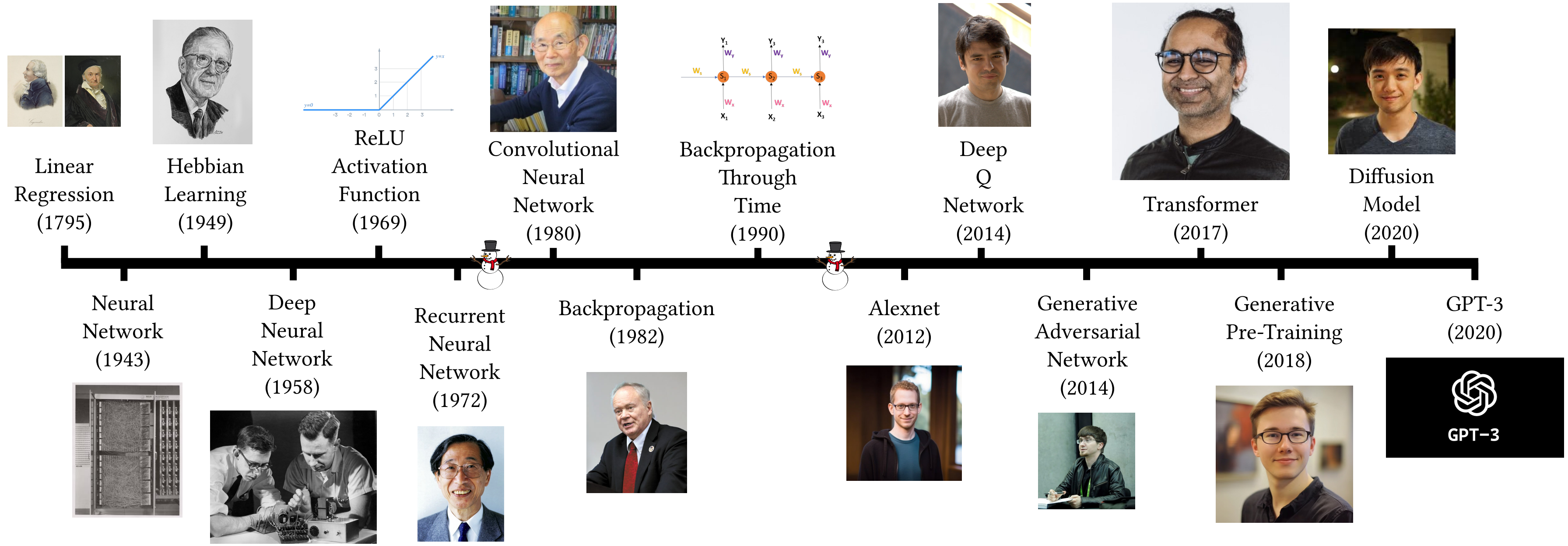
Activation Functions

The sigmoid function was the standard activation function until ~ 2012



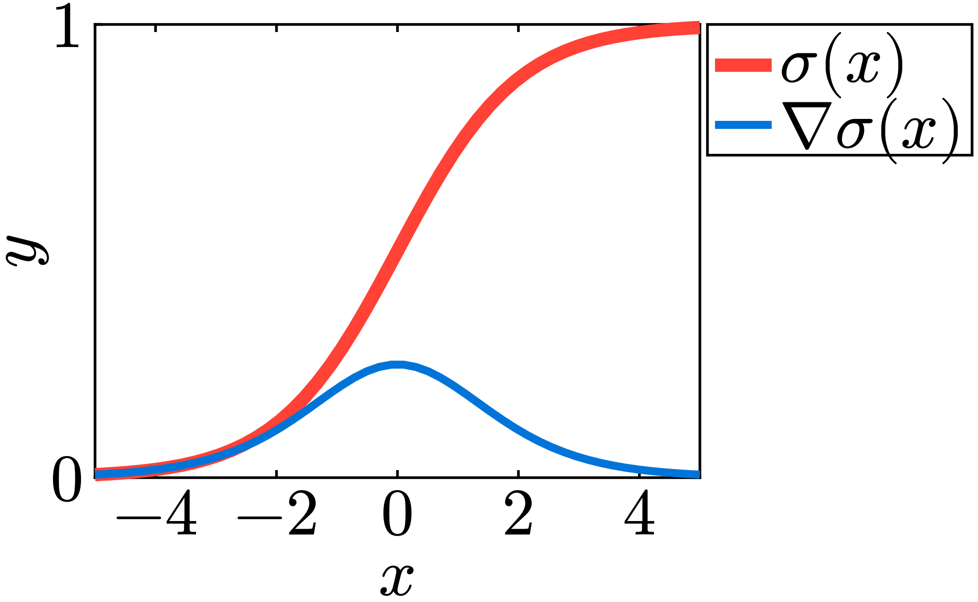
Activation Functions

The sigmoid function was the standard activation function until ~ 2012

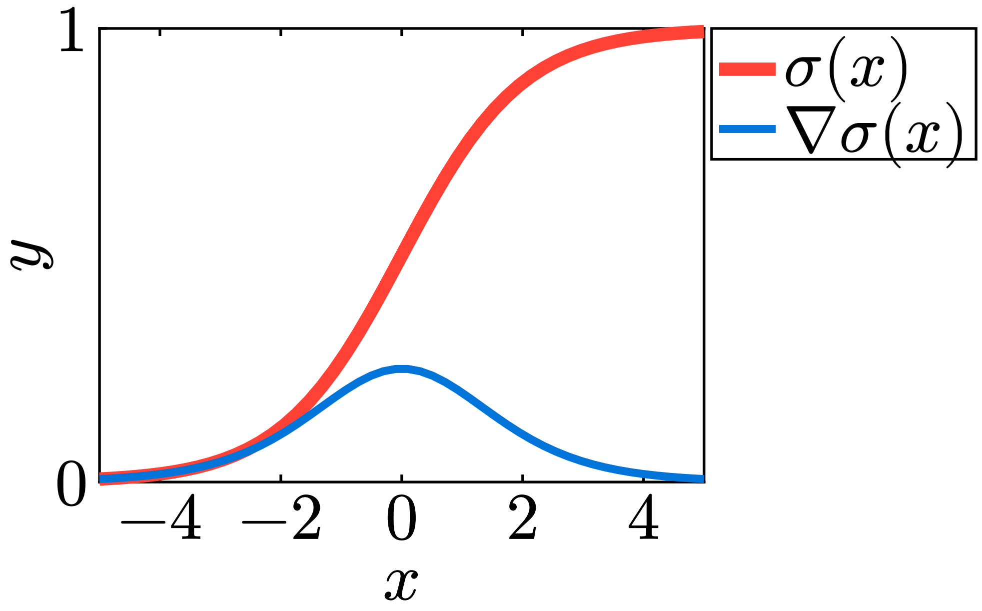


In 2012, people realized that ReLU activation performed much better

Activation Functions

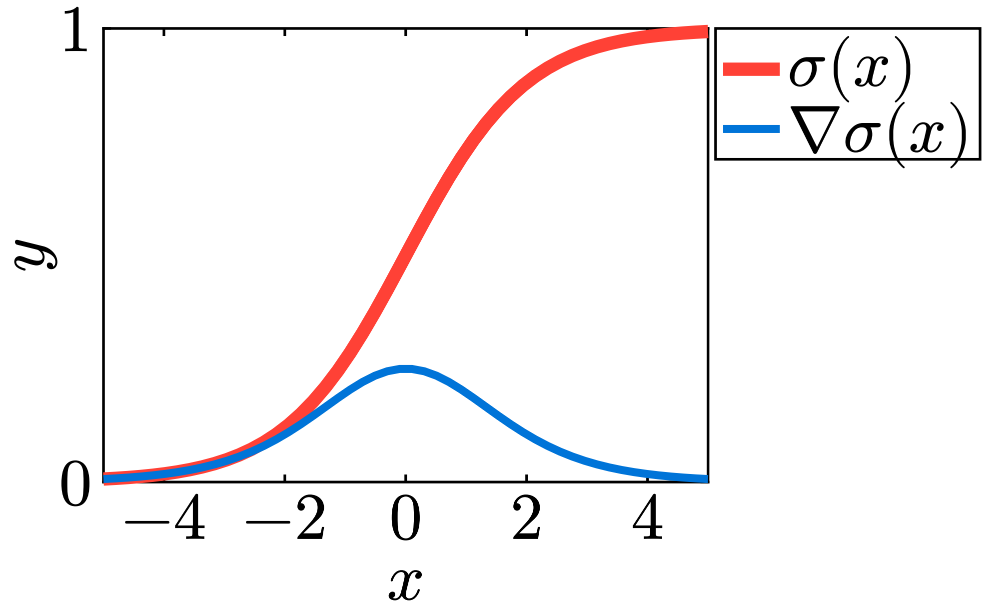


Activation Functions



The sigmoid function can result in a **vanishing gradient**

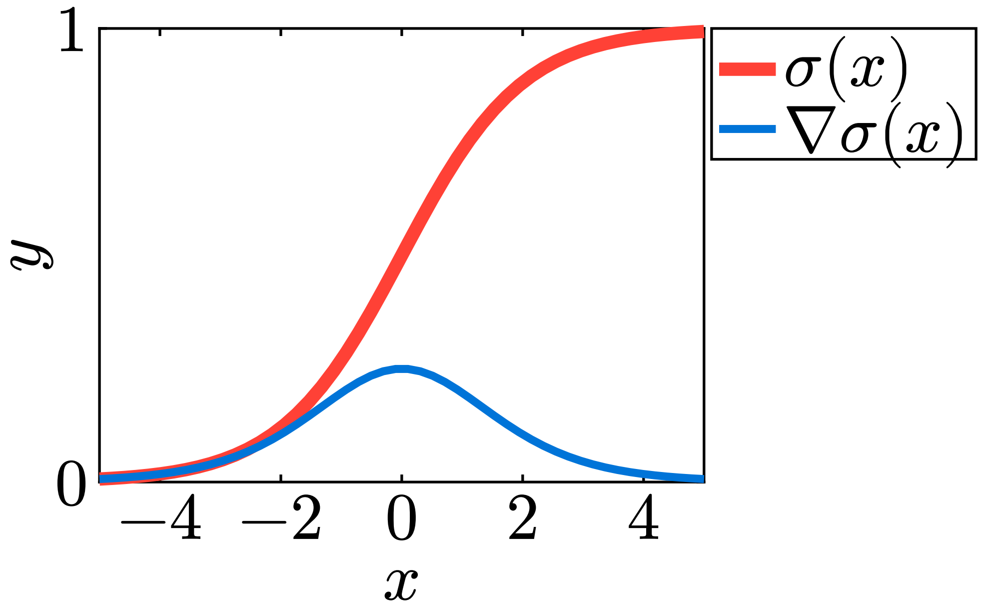
Activation Functions



The sigmoid function can result in a **vanishing gradient**

$$f(\boldsymbol{x}, \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}_3^\top \sigma(\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})))$$

Activation Functions

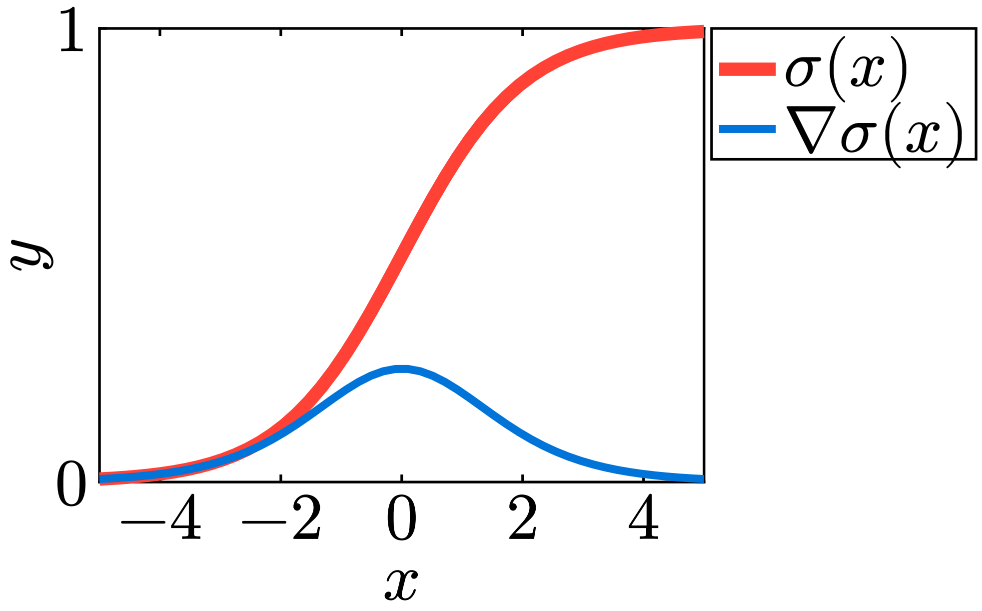


The sigmoid function can result in a **vanishing gradient**

$$f(\boldsymbol{x}, \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}_3^\top \sigma(\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})))$$

$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) = \nabla[\sigma](\boldsymbol{\theta}_3^\top \sigma(\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}}))) \cdot \nabla[\sigma](\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})) \cdot \nabla[\sigma](\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})$$

Activation Functions

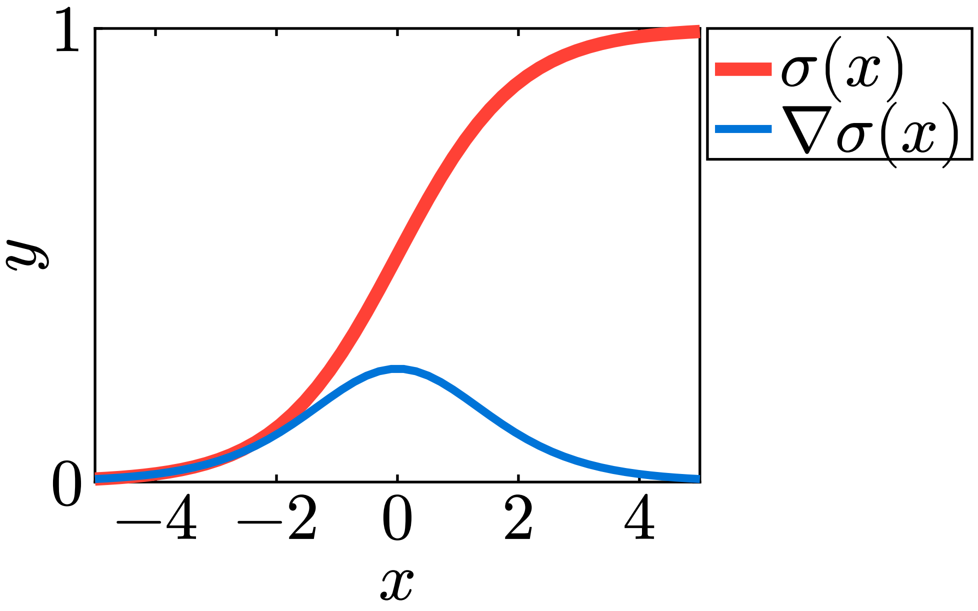


The sigmoid function can result in a **vanishing gradient**

$$f(\boldsymbol{x}, \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}_3^\top \sigma(\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})))$$

$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) = \underbrace{\nabla[\sigma](\boldsymbol{\theta}_3^\top \sigma(\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})))}_{<1} \cdot \underbrace{\nabla[\sigma](\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}}))}_{<1} \cdot \underbrace{\nabla[\sigma](\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})}_{<1}$$

Activation Functions



The sigmoid function can result in a **vanishing gradient**

$$f(\boldsymbol{x}, \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}_3^\top \sigma(\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})))$$

$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) = \underbrace{\nabla[\sigma](\boldsymbol{\theta}_3^\top \sigma(\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})))}_{<1} \cdot \underbrace{\nabla[\sigma](\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}}))}_{<1} \cdot \underbrace{\nabla[\sigma](\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})}_{<1}$$

$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) \approx 0$$

Activation Functions

To fix the vanishing gradient, researchers use the **rectified linear unit (ReLU)**

$$\sigma(x) = \max(0, x)$$

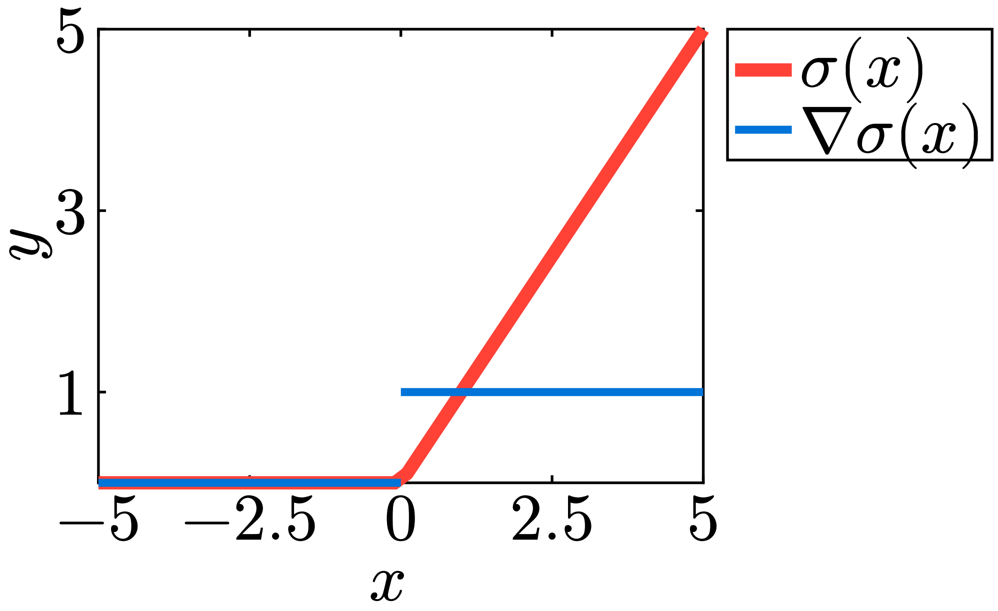
$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$

Activation Functions

To fix the vanishing gradient, researchers use the **rectified linear unit (ReLU)**

$$\sigma(x) = \max(0, x)$$

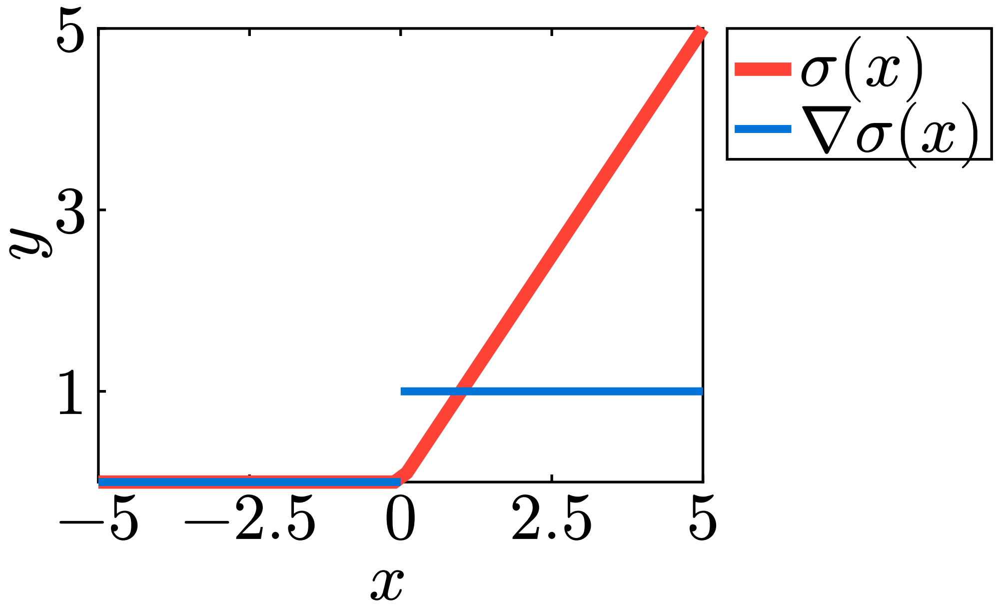
$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$

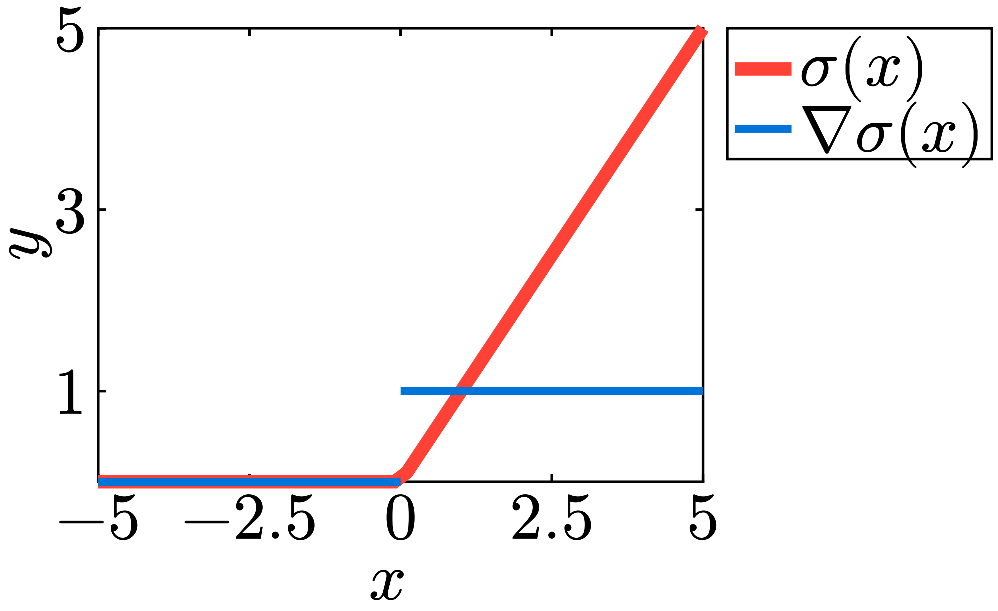


Looks nothing like a biological neuron

Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



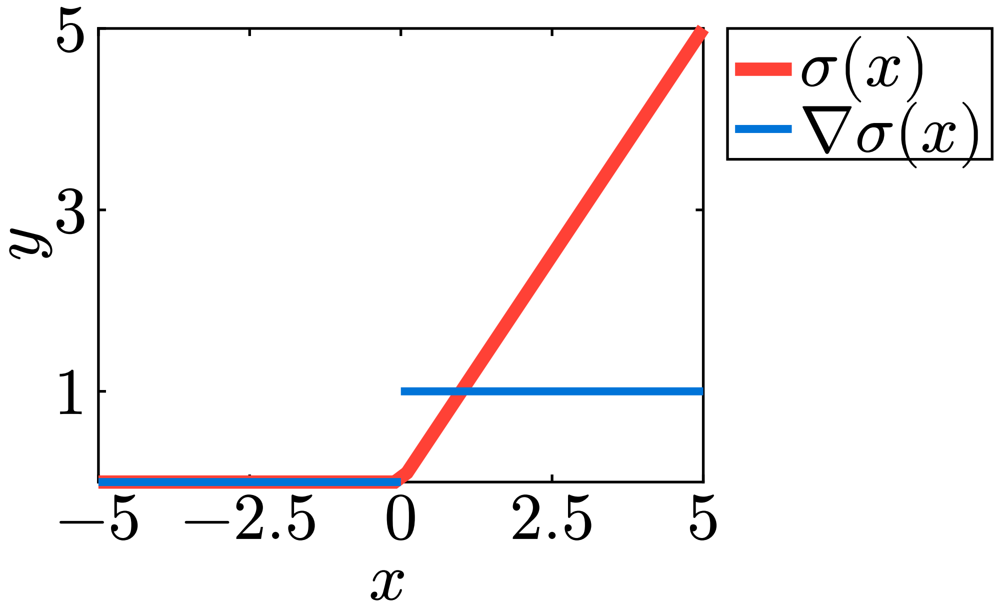
Looks nothing like a biological neuron

However, it works much better than sigmoid in practice

Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



Looks nothing like a biological neuron

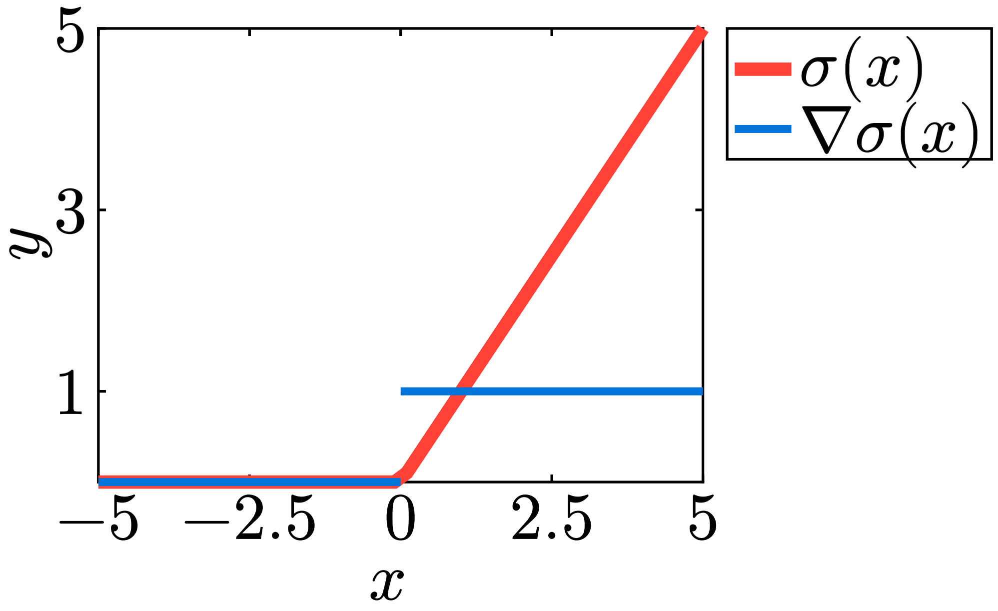
However, it works much better than sigmoid in practice

Via chain rule, gradient is $1 \cdot 1 \cdot 1 \ldots$ which does not vanish

Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



However, it works much better than sigmoid in practice

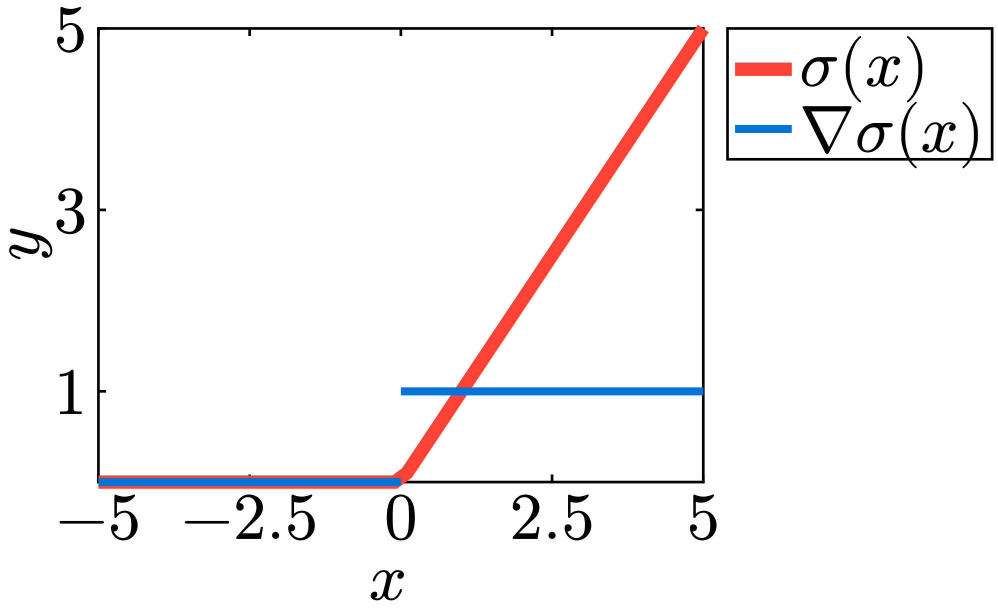
Via chain rule, gradient is $1 \cdot 1 \cdot 1 \ldots$ which does not vanish

The gradient is constant, resulting in easier optimization

Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



Via chain rule, gradient is $1 \cdot 1 \cdot 1 \ldots$ which does not vanish

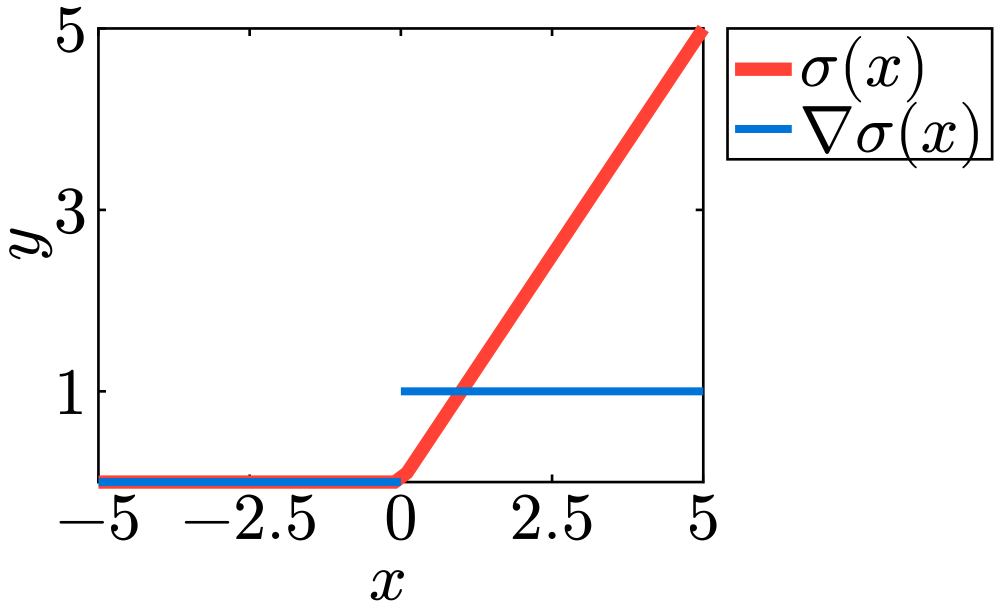
The gradient is constant, resulting in easier optimization

Question: Any problems?

Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



The gradient is constant, resulting in easier optimization

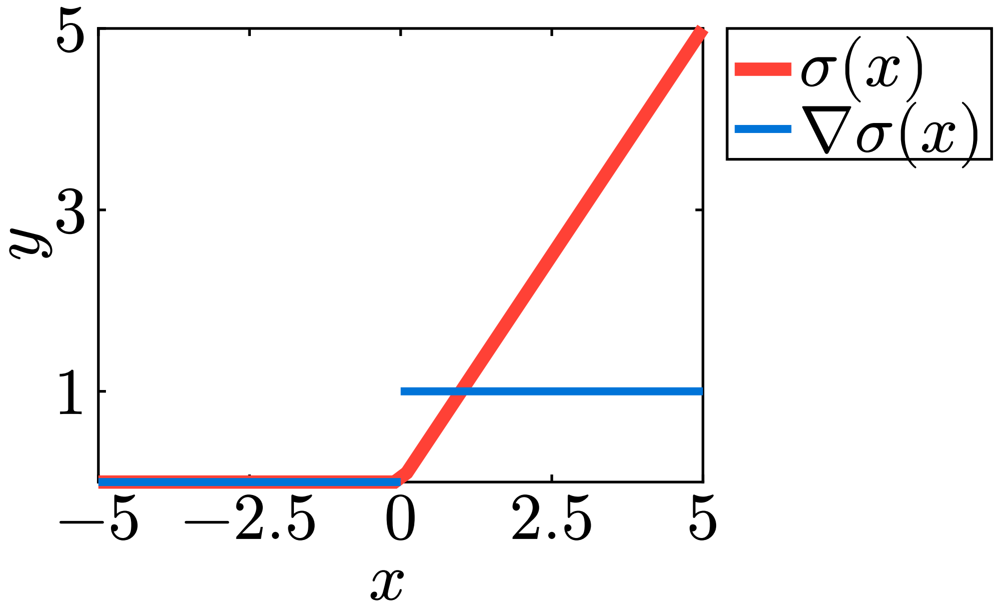
Question: Any problems?

Answer: Zero gradient region!

Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



Question: Any problems?

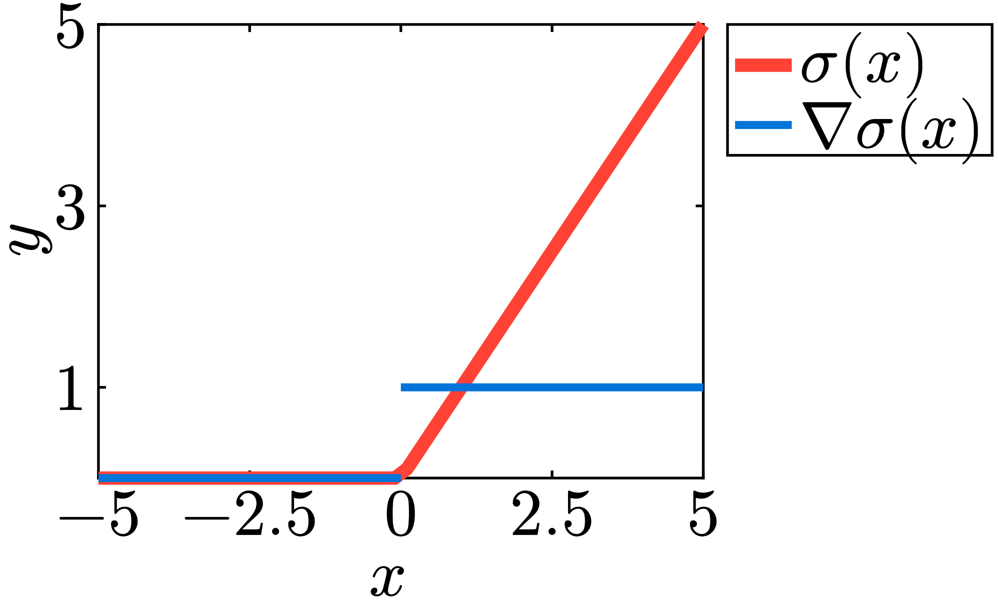
Answer: Zero gradient region!

Neurons can get "stuck", always output 0

Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



Answer: Zero gradient region!

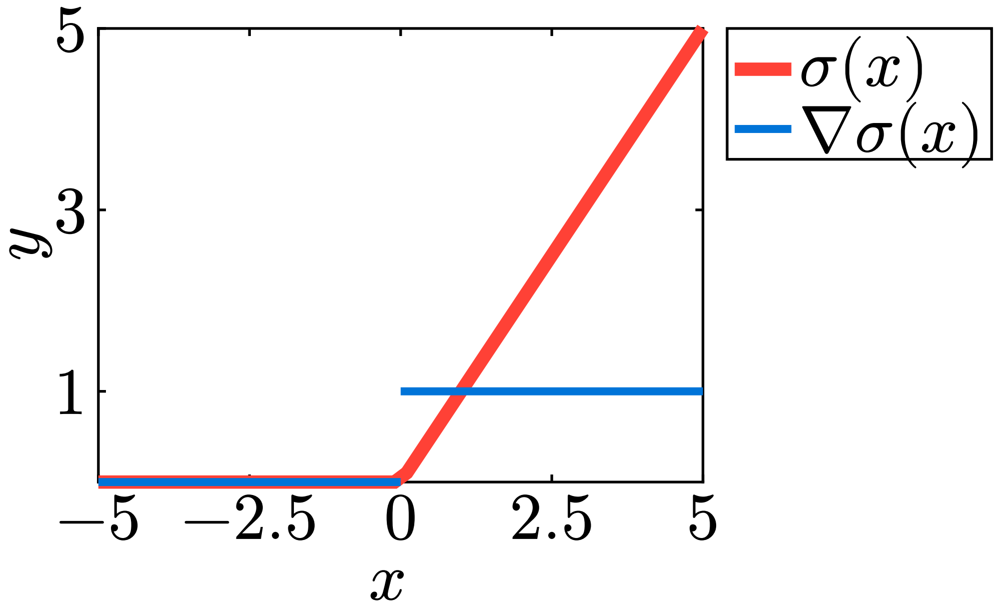
Neurons can get "stuck", always output 0

These neurons cannot recover, they are **dead neurons**

Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$

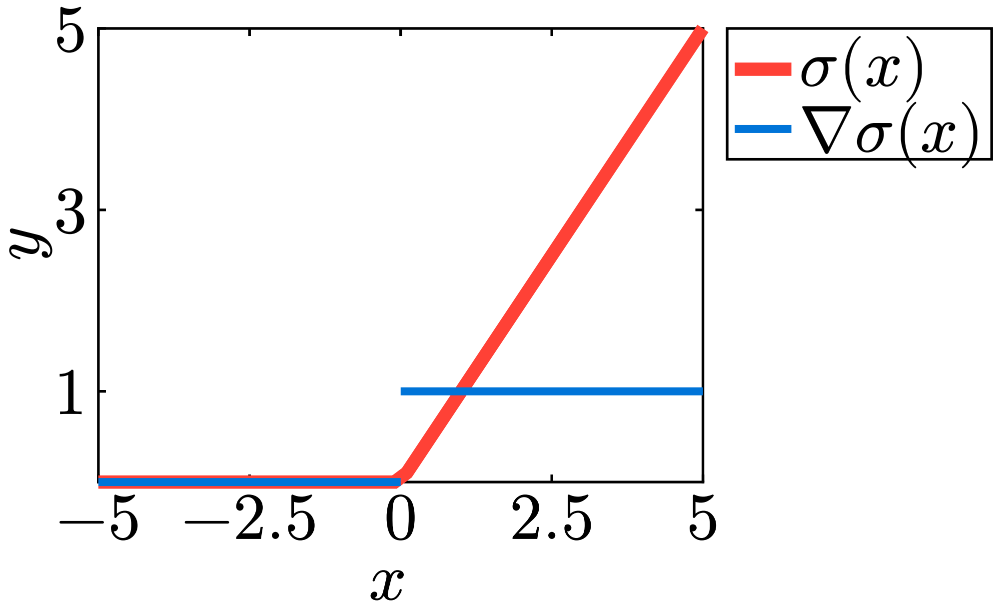


These neurons cannot recover, they are **dead neurons**

Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



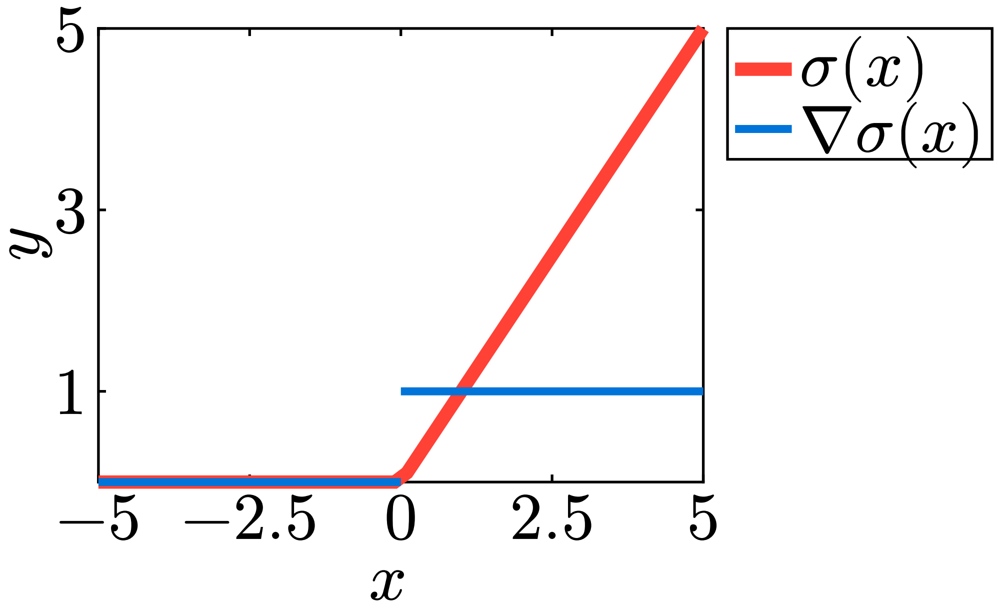
These neurons cannot recover, they are **dead neurons**

Training for longer results in more dead neurons

Activation Functions

$$\sigma(x) = \max(0, x)$$

$$\nabla \sigma(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



These neurons cannot recover, they are **dead neurons**

Training for longer results in more dead neurons

Dead neurons hurt your network!

Activation Functions

To fix dying neurons, use **leaky ReLU**

Activation Functions

To fix dying neurons, use **leaky ReLU**

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

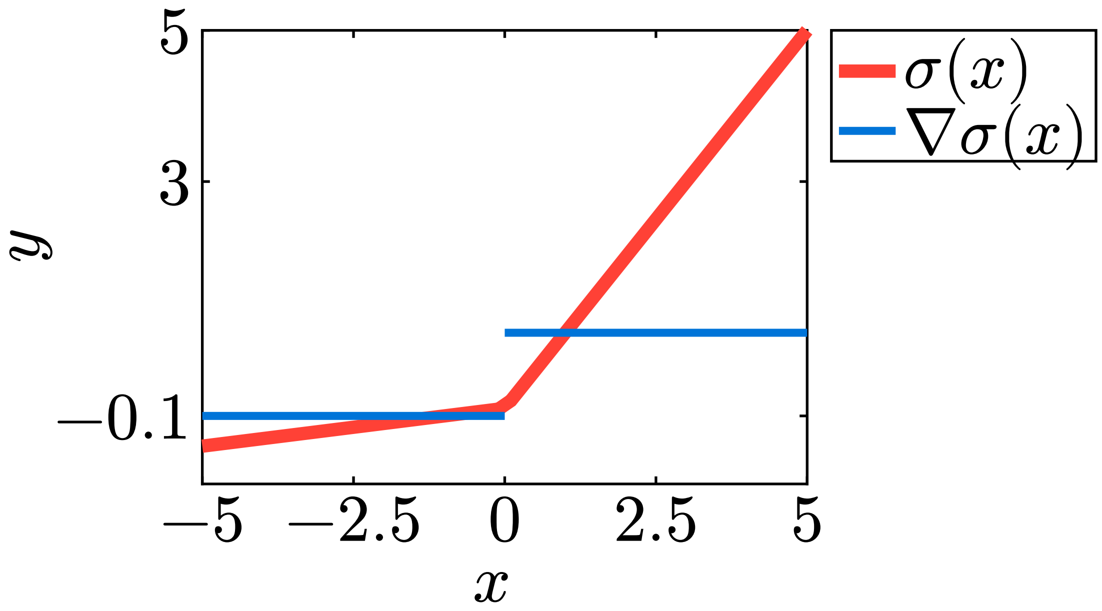
$$\nabla \sigma(x) = \begin{cases} 0.1 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$

Activation Functions

To fix dying neurons, use **leaky ReLU**

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

$$\nabla\sigma(x) = \begin{cases} 0.1 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$

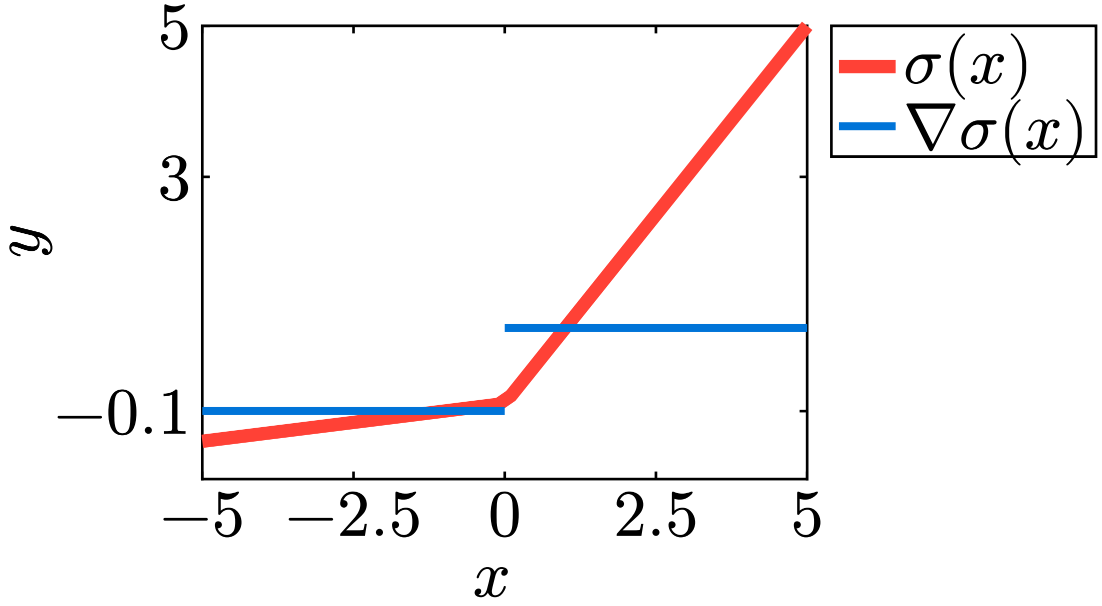


Activation Functions

To fix dying neurons, use **leaky ReLU**

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

$$\nabla \sigma(x) = \begin{cases} 0.1 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



Small negative slope allows dead neurons to recover

Activation Functions

There are other activation functions that are better than leaky ReLU

Activation Functions

There are other activation functions that are better than leaky ReLU

- Mish

Activation Functions

There are other activation functions that are better than leaky ReLU

- Mish
- Swish

Activation Functions

There are other activation functions that are better than leaky ReLU

- Mish
- Swish
- ELU

Activation Functions

There are other activation functions that are better than leaky ReLU

- Mish
- Swish
- ELU
- GeLU

Activation Functions

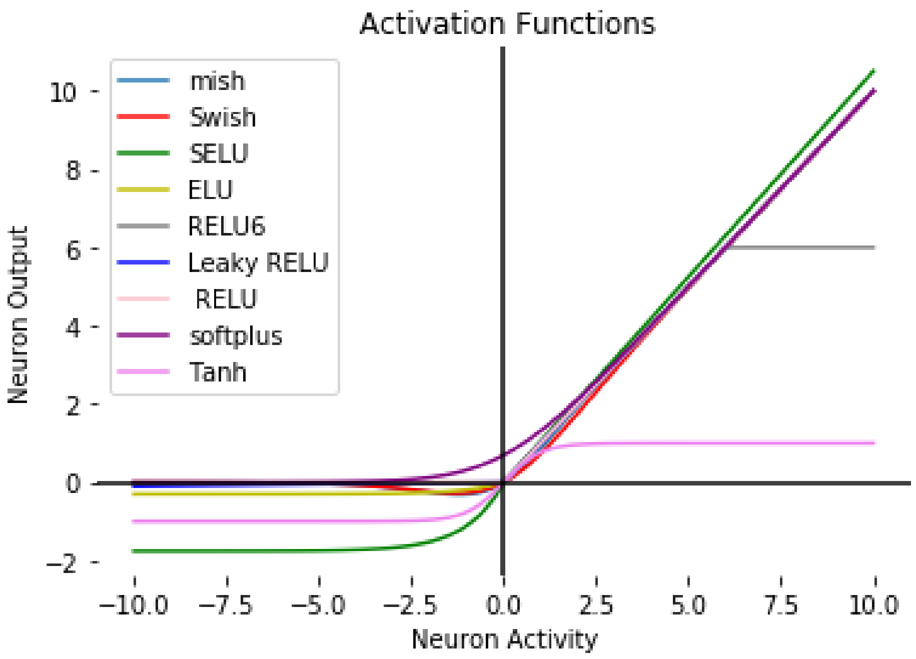
There are other activation functions that are better than leaky ReLU

- Mish
- Swish
- ELU
- GeLU
- SeLU

Activation Functions

There are other activation functions that are better than leaky ReLU

- Mish
- Swish
- ELU
- GeLU
- SeLU

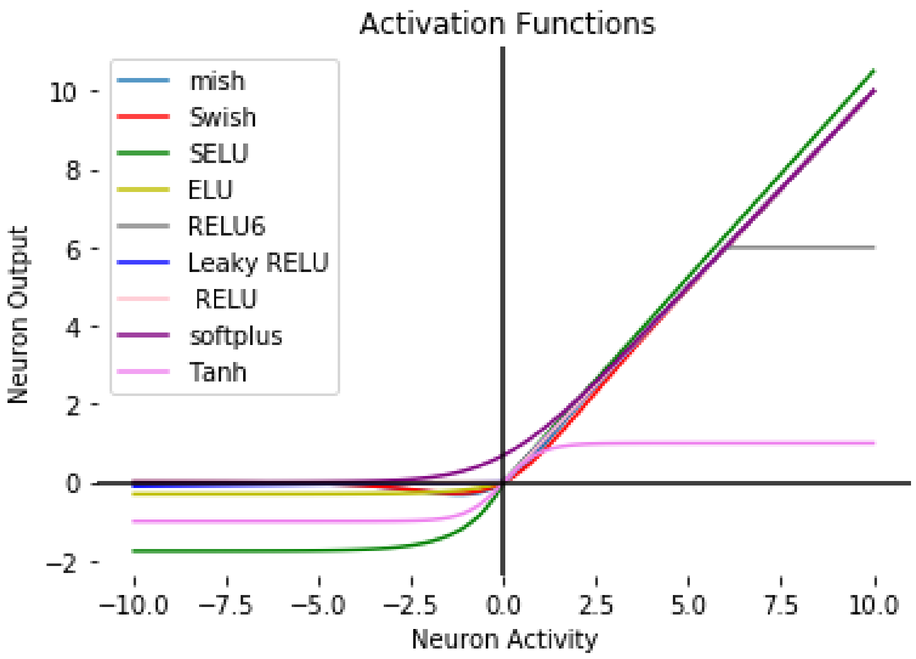


Activation Functions

There are other activation functions that are better than leaky ReLU

- Mish
- Swish
- ELU
- GeLU
- SeLU

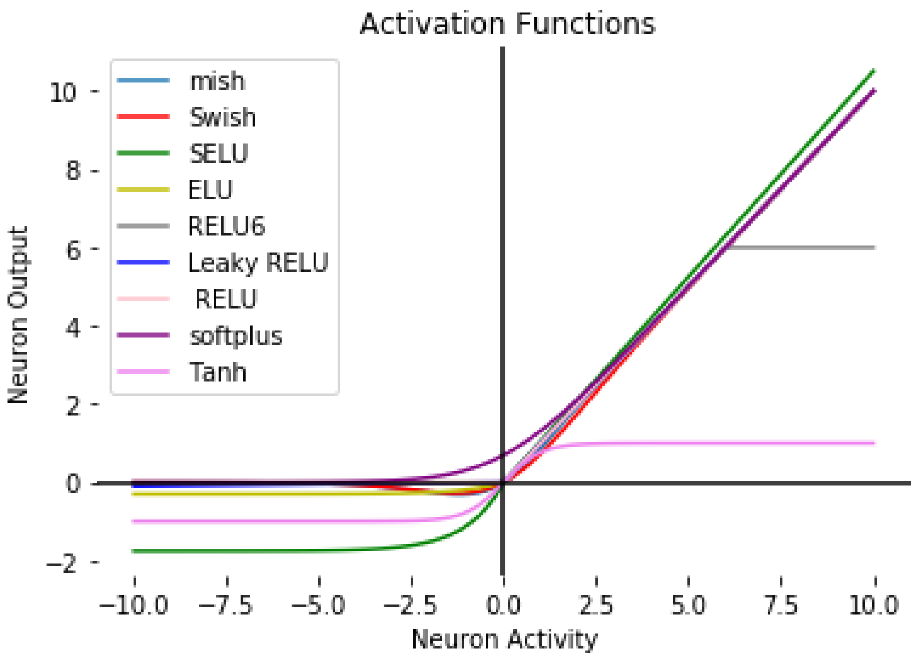
They are all very similar



Activation Functions

There are other activation functions that are better than leaky ReLU

- Mish
- Swish
- ELU
- GeLU
- SeLU



They are all very similar

I usually use leaky ReLU because it works well enough

Activation Functions

https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

https://jax.readthedocs.io/en/latest/jax.nn.html#activation-functions

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. **Activation functions**
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. **Parameter initialization**
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Parameter Initialization

Recall the gradient descent algorithm

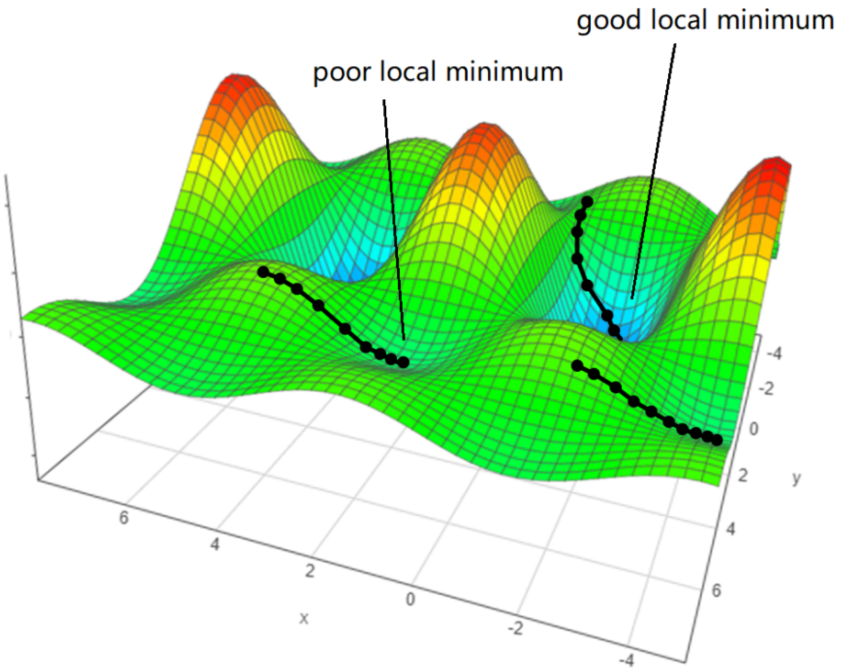
function Gradient Descent($\boldsymbol{X}, \boldsymbol{Y}, \mathcal{L}, t, \alpha$)
    ▷ Randomly initialize parameters
    $\boldsymbol{\theta} \leftarrow \mathcal{N}(0, 1)$
    for $i \in 1 \ldots t$ **do**
        ▷ Compute the gradient of the loss
        $\boldsymbol{J} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\theta})$
        ▷ Update the parameters using the negative gradient
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{J}$
    return $\boldsymbol{\theta}$

Parameter Initialization

Initial $\boldsymbol{\theta}$ is starting position for gradient descent

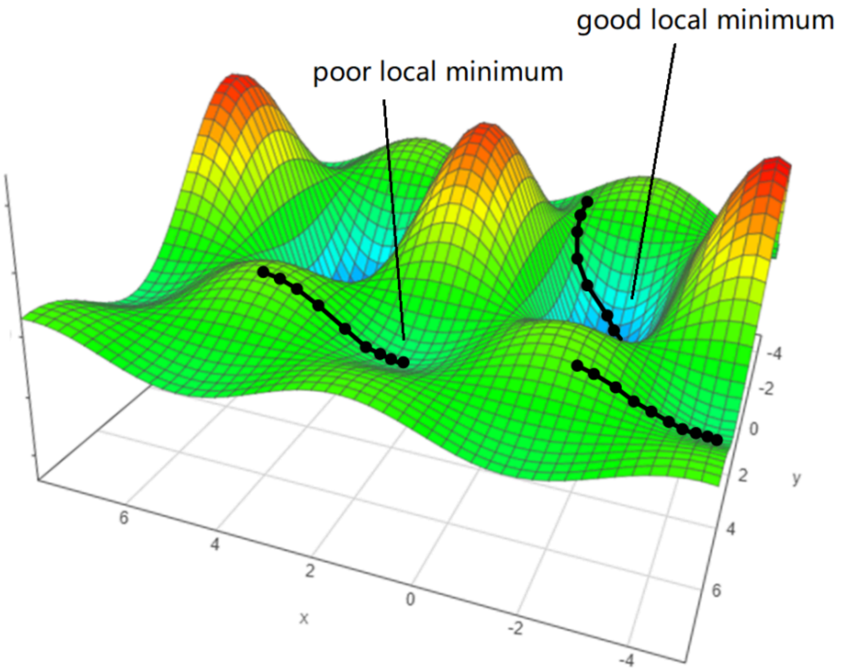
Parameter Initialization

Initial θ is starting position for gradient descent



Parameter Initialization

Initial $\boldsymbol{\theta}$ is starting position for gradient descent



Pick $\boldsymbol{\theta}$ that results in good local minima

Parameter Initialization

Start simple, initialize all parameters to 0

$$\boldsymbol{\theta} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}, \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}, \dots$$

Parameter Initialization

Start simple, initialize all parameters to 0

$$\boldsymbol{\theta} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}, \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}, \dots$$

Question: Any issues?

Parameter Initialization

Start simple, initialize all parameters to 0

$$\boldsymbol{\theta} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}, \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}, \dots$$

Question: Any issues?

Answer: The gradient will always be zero

$$\nabla_{\boldsymbol{\theta}_1} f = \sigma\left(\boldsymbol{\theta}_2^\top \sigma\left(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}}\right)\right) \sigma\left(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}}\right) \overline{\boldsymbol{x}}$$

Parameter Initialization

Start simple, initialize all parameters to 0

$$\boldsymbol{\theta} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}, \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix}, \dots$$

Question: Any issues?

Answer: The gradient will always be zero

$$\nabla_{\boldsymbol{\theta}_1} f = \sigma(\boldsymbol{\theta}_2^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}}))\ \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})\ \overline{\boldsymbol{x}}$$

$$\nabla_{\boldsymbol{\theta}_1} f = \sigma(\mathbf{0}^\top \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}}))\ \sigma(\boldsymbol{\theta}_1^\top \overline{\boldsymbol{x}})\ \overline{\boldsymbol{x}} = 0$$

Parameter Initialization

Ok, so initialize $\boldsymbol{\theta} = \mathbf{1}$

Parameter Initialization

Ok, so initialize $\boldsymbol{\theta} = \mathbf{1}$

$$\boldsymbol{\theta} = \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}, \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}, \dots$$

Parameter Initialization

Ok, so initialize $\boldsymbol{\theta} = \mathbf{1}$

$$\boldsymbol{\theta} = \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}, \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}, \dots$$

Question: Any issues?

Parameter Initialization

Ok, so initialize $\boldsymbol{\theta} = \mathbf{1}$

$$\boldsymbol{\theta} = \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}, \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}, \dots$$

Question: Any issues?

All neurons in a layer will have the same gradient, and so they will always be the same (useless)

$$z_i = \sigma\left(\sum_{j=1}^{d_x} \theta_j \cdot \overline{x}_j\right) = \sigma\left(\sum_{j=1}^{d_x} \overline{x}_j\right)$$

Parameter Initialization

$\boldsymbol{\theta}$ must be randomly initialized for neurons

$$\boldsymbol{\theta} = \begin{bmatrix} -0.5 & \dots & 2 \\ \vdots & \ddots & \vdots \\ 0.1 & \dots & 0.6 \end{bmatrix}, \begin{bmatrix} 1.3 & \dots & 1.2 \\ \vdots & \ddots & \vdots \\ -0.8 & \dots & -1.1 \end{bmatrix}, \dots$$

Parameter Initialization

$\boldsymbol{\theta}$ must be randomly initialized for neurons

$$\boldsymbol{\theta} = \begin{bmatrix} -0.5 & \dots & 2 \\ \vdots & \ddots & \vdots \\ 0.1 & \dots & 0.6 \end{bmatrix}, \begin{bmatrix} 1.3 & \dots & 1.2 \\ \vdots & \ddots & \vdots \\ -0.8 & \dots & -1.1 \end{bmatrix}, \dots$$

But what scale? If $\boldsymbol{\theta} \ll 0$ the gradients will vanish to zero, if $\boldsymbol{\theta} \gg 0$ the gradients explode to infinity

Parameter Initialization

$\boldsymbol{\theta}$ must be randomly initialized for neurons

$$\boldsymbol{\theta} = \begin{bmatrix} -0.5 & \dots & 2 \\ \vdots & \ddots & \vdots \\ 0.1 & \dots & 0.6 \end{bmatrix}, \begin{bmatrix} 1.3 & \dots & 1.2 \\ \vdots & \ddots & \vdots \\ -0.8 & \dots & -1.1 \end{bmatrix}, \dots$$

But what scale? If $\boldsymbol{\theta} \ll 0$ the gradients will vanish to zero, if $\boldsymbol{\theta} \gg 0$ the gradients explode to infinity

Almost everyone initializes following a single paper from 2010:

Parameter Initialization

$\boldsymbol{\theta}$ must be randomly initialized for neurons

$$\boldsymbol{\theta} = \begin{bmatrix} -0.5 & \dots & 2 \\ \vdots & \ddots & \vdots \\ 0.1 & \dots & 0.6 \end{bmatrix}, \begin{bmatrix} 1.3 & \dots & 1.2 \\ \vdots & \ddots & \vdots \\ -0.8 & \dots & -1.1 \end{bmatrix}, \dots$$

But what scale? If $\boldsymbol{\theta} \ll 0$ the gradients will vanish to zero, if $\boldsymbol{\theta} \gg 0$ the gradients explode to infinity

Almost everyone initializes following a single paper from 2010:
- Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks."

Parameter Initialization

$\boldsymbol{\theta}$ must be randomly initialized for neurons

$$\boldsymbol{\theta} = \begin{bmatrix} -0.5 & \dots & 2 \\ \vdots & \ddots & \vdots \\ 0.1 & \dots & 0.6 \end{bmatrix}, \begin{bmatrix} 1.3 & \dots & 1.2 \\ \vdots & \ddots & \vdots \\ -0.8 & \dots & -1.1 \end{bmatrix}, \dots$$

But what scale? If $\boldsymbol{\theta} \ll 0$ the gradients will vanish to zero, if $\boldsymbol{\theta} \gg 0$ the gradients explode to infinity

Almost everyone initializes following a single paper from 2010:

- Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks."
- Maybe there are better options?

Parameter Initialization

Here is the magic equation, given the input and output size of the layer is d_h

$$\boldsymbol{\theta} \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{2d_h}}, \frac{\sqrt{6}}{\sqrt{2d_h}}\right]$$

Parameter Initialization

Here is the magic equation, given the input and output size of the layer is d_h

$$\boldsymbol{\theta} \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{2 d_h}}, \frac{\sqrt{6}}{\sqrt{2 d_h}}\right]$$

If you have different input or output sizes, such as d_x, d_y, then the equation is

$$\boldsymbol{\theta} \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{d_x + d_y}}, \frac{\sqrt{6}}{\sqrt{d_x + d_y}}\right]$$

Parameter Initialization

These equations are designed for ReLU and similar activation functions

Parameter Initialization

These equations are designed for ReLU and similar activation functions

They prevent vanishing or exploding gradients

Parameter Initialization

Usually `torch` and `jax/equinox` will automatically use this initialization when you create `nn.Linear`

```
layer = nn.Linear(d_x, d_h) # Uses Glorot init
```

You can find many initialization functions at https://pytorch.org/docs/stable/nn.init.html

For JAX it is https://jax.readthedocs.io/en/latest/jax.nn.initializers.html

Parameter Initialization

```python
import torch
d_h = 10
# Manually
theta = torch.zeros((d_h + 1, d_h))
torch.nn.init.xavier_uniform_(theta)
theta = torch.nn.Parameter(theta)

# Using nn.Linear
layer = torch.nn.Linear(d_h, d_h)
# USe .data, to bypass autograd security
torch.nn.init.xavier_uniform_(layer.weight.data)
torch.nn.init.xavier_uniform_(layer.bias.data)
```

Parameter Initialization

```
import jax
d_h = 10

init = jax.nn.initializers.glorot_uniform()
theta = init(jax.random.key(0), (d_h + 1, d_h))
```

Parameter Initialization

```
import jax, equinox
d_h = 10

layer = equinox.nn.Linear(d_h, d_h, key=jax.random.key(0))
# Create new bias and weight
new_weight = init(jax.random.key(1), (d_h, d_h))
new_bias = init(jax.random.key(2), (d_h,))

# Use a lambda function to save space
# tree_at creates a new layer with the new weight
layer = equinox.tree_at(lambda l: l.weight, layer,
new_weight)
layer = equinox.tree_at(lambda l: l.bias, layer, new_weight)
```

Parameter Initialization

Remember, in `equinox` and `torch`, `nn.Linear` will already be initialized correctly!

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. **Parameter initialization**
7. Stochastic gradient descent
8. Modern optimization
9. Coding

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. **Stochastic gradient descent**
8. Modern optimization
9. Coding

Stochastic Gradient Descent

function GRADIENT DESCENT($\boldsymbol{X}$, $\boldsymbol{Y}$, $\mathcal{L}$, t, α)

$\quad \triangleright$ Randomly initialize parameters

$\quad \boldsymbol{\theta} \leftarrow \text{Glorot}()$

$\quad$ **for** $i \in 1...t$ **do**

$\quad\quad \triangleright$ Compute the gradient of the loss

$\quad\quad \boldsymbol{J} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\theta})$

$\quad\quad \triangleright$ Update the parameters using the negative gradient

$\quad\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{J}$

$\quad$ **return** $\boldsymbol{\theta}$

Stochastic Gradient Descent

function $\textsc{Gradient Descent}(\boldsymbol{X}, \boldsymbol{Y}, \mathcal{L}, t, \alpha)$

  $\triangleright$ Randomly initialize parameters

  $\boldsymbol{\theta} \leftarrow \text{Glorot}()$

  for $i \in 1...t$ **do**

    $\triangleright$ Compute the gradient of the loss

    $\boldsymbol{J} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\theta})$

    $\triangleright$ Update the parameters using the negative gradient

    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{J}$

  return $\boldsymbol{\theta}$

Gradient descent computes $\nabla \mathcal{L}$ over all $\boldsymbol{X}$

Stochastic Gradient Descent

This works for our small datasets, where $n = 1000$

Stochastic Gradient Descent

This works for our small datasets, where $n = 1000$

Question: How many GB are the LLM datasets?

Stochastic Gradient Descent

This works for our small datasets, where $n = 1000$

Question: How many GB are the LLM datasets?

Answer: About 774,000 GB according to *Datasets for Large Language Models: A Comprehensive Survey*

Stochastic Gradient Descent

This works for our small datasets, where $n = 1000$

Question: How many GB are the LLM datasets?

Answer: About 774,000 GB according to *Datasets for Large Language Models: A Comprehensive Survey*

This is just the dataset size, the gradient is orders of magnitude larger

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{x}_{[i]}, \boldsymbol{y}_{[i]}, \boldsymbol{\theta}\right) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_\ell}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_\ell}{\partial x_1} \end{bmatrix}_{[i]}$$

Stochastic Gradient Descent

Question: We do not have enough memory to compute the gradient

Stochastic Gradient Descent

Question: We do not have enough memory to compute the gradient

Answer: What can we do?

Stochastic Gradient Descent

Question: We do not have enough memory to compute the gradient

Answer: What can we do?

We approximate the gradient using a subset of the data

Stochastic Gradient Descent

Question: We do not have enough memory to compute the gradient

Answer: What can we do?

We approximate the gradient using a subset of the data

Stochastic Gradient Descent

First, we sample random datapoint indices

$$i, j, k, \ldots \sim \mathcal{U}[1, n]$$

Stochastic Gradient Descent

First, we sample random datapoint indices

$$i, j, k, \dots \sim \mathcal{U}[1, n]$$

Then construct a **batch** of training data

$$\begin{bmatrix} \boldsymbol{x}_{[i]} \\ \boldsymbol{x}_{[j]} \\ \boldsymbol{x}_{[k]} \\ \vdots \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{y}_{[i]} \\ \boldsymbol{y}_{[j]} \\ \boldsymbol{y}_{[k]} \\ \vdots \end{bmatrix}$$

Stochastic Gradient Descent

First, we sample random datapoint indices

$$i, j, k, ... \sim \mathcal{U}[1, n]$$

Then construct a **batch** of training data

$$\begin{bmatrix} \boldsymbol{x}_{[i]} \\ \boldsymbol{x}_{[j]} \\ \boldsymbol{x}_{[k]} \\ \vdots \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{y}_{[i]} \\ \boldsymbol{y}_{[j]} \\ \boldsymbol{y}_{[k]} \\ \vdots \end{bmatrix}$$

We call this **stochastic gradient descent**

Stochastic Gradient Descent

function $\textsc{Stochastic Gradient Descent}(\boldsymbol{X}, \boldsymbol{Y}, \mathcal{L}, t, \alpha)$

$\quad \boldsymbol{\theta} \leftarrow \mathrm{Glorot}()$

$\quad$ **for** $i \in 1...t$ **do**

$\quad\quad \boldsymbol{X}, \boldsymbol{Y} \leftarrow \mathrm{Shuffle}(\boldsymbol{X}), \mathrm{Shuffle}(\boldsymbol{Y})$

$\quad\quad$ **for** $j \in 0...\frac{n}{B} - 1$ **do**

$\quad\quad\quad \boldsymbol{X}_j \leftarrow \begin{bmatrix} \boldsymbol{x}_{[jB]} & \boldsymbol{x}_{[jB+1]} & \dots & \boldsymbol{x}_{[(j+1)B]} \end{bmatrix}$

$\quad\quad\quad \boldsymbol{Y}_j \leftarrow \begin{bmatrix} \boldsymbol{y}_{[jB]} & \boldsymbol{y}_{[jB+1]} & \dots & \boldsymbol{y}_{[(j+1)B]} \end{bmatrix}$

$\quad\quad\quad \boldsymbol{J} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{X}_j, \boldsymbol{Y}_j, \boldsymbol{\theta})$

$\quad\quad\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{J}$

$\quad$ **return** $\boldsymbol{\theta}$

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is useful for saving memory

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is useful for saving memory

But it can also improve performance

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is useful for saving memory

But it can also improve performance

Since the "dataset" changes every update, so does the loss manifold

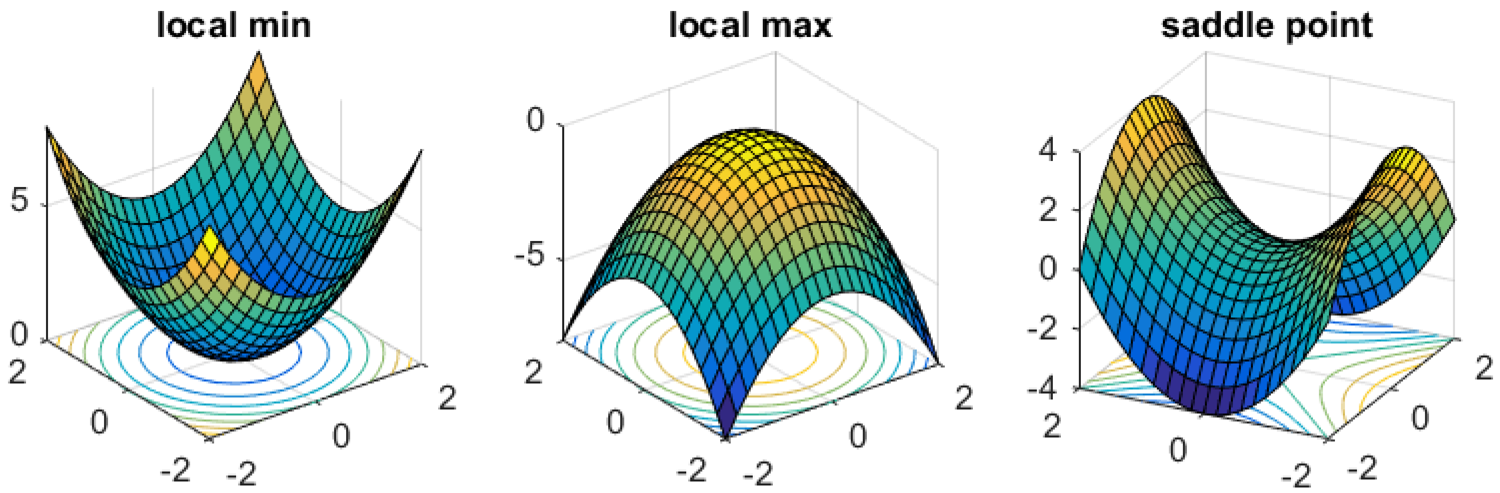
Stochastic Gradient Descent

Stochastic gradient descent (SGD) is useful for saving memory

But it can also improve performance

Since the “dataset” changes every update, so does the loss manifold

This makes it less likely we get stuck in bad optima



Stochastic Gradient Descent

There is `torch.utils.data.DataLoader` to help

Stochastic Gradient Descent

There is `torch.utils.data.DataLoader` to help

```
import torch
dataloader = torch.utils.data.DataLoader(
    training_data,
    batch_size=32, # How many datapoints to sample
    shuffle=True, # Randomly shuffle each epoch
)
for epoch in number_of_epochs:
    for batch in dataloader:
        X_j, Y_j = batch
        loss = L(X_j, Y_j, theta)
        ...
```

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. **Stochastic gradient descent**
8. Modern optimization
9. Coding

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. **Modern optimization**
9. Coding

Modern Optimization

Gradient descent is a powerful tool, but it has issues

Modern Optimization

Gradient descent is a powerful tool, but it has issues

1. It can be slow to converge

Modern Optimization

Gradient descent is a powerful tool, but it has issues

1. It can be slow to converge
2. It can get stuck in poor local optima

Modern Optimization

Gradient descent is a powerful tool, but it has issues

1. It can be slow to converge
2. It can get stuck in poor local optima

Many researchers work on improving gradient descent to converge more quickly, while also preventing premature convergence

Modern Optimization

Gradient descent is a powerful tool, but it has issues

1. It can be slow to converge
2. It can get stuck in poor local optima

Many researchers work on improving gradient descent to converge more quickly, while also preventing premature convergence

It is hard to teach adaptive optimization through math

Modern Optimization

Gradient descent is a powerful tool, but it has issues

1. It can be slow to converge
2. It can get stuck in poor local optima

Many researchers work on improving gradient descent to converge more quickly, while also preventing premature convergence

It is hard to teach adaptive optimization through math

So first, I want to show you a video to prepare you

https://www.youtube.com/watch?v=MD2fYip6QsQ&t=77s

Modern Optimization

The video simulations provide an intuitive understanding of adaptive optimizers

Modern Optimization

The video simulations provide an intuitive understanding of adaptive optimizers

The key behind modern optimizers is two concepts:

- Momentum

Modern Optimization

The video simulations provide an intuitive understanding of adaptive optimizers

The key behind modern optimizers is two concepts:

- Momentum
- Adaptive learning rate

Modern Optimization

The video simulations provide an intuitive understanding of adaptive optimizers

The key behind modern optimizers is two concepts:
- Momentum
- Adaptive learning rate

Let us discuss the algorithms more slowly

Modern Optimization

Review gradient descent again, because we will be making changes to it

Modern Optimization

Review gradient descent again, because we will be making changes to it

function $\textsc{Gradient Descent}(\boldsymbol{X}, \boldsymbol{Y}, \mathcal{L}, t, \alpha)$

  $\triangleright$ Randomly initialize parameters

  $\boldsymbol{\theta} \leftarrow \text{Glorot}()$

  for $i \in 1...t$ **do**

    $\triangleright$ Compute the gradient of the loss

    $\boldsymbol{J} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\theta})$

    $\triangleright$ Update the parameters using the negative gradient

    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{J}$

  return $\boldsymbol{\theta}$

Modern Optimization

Introduce **momentum** first

Modern Optimization

Introduce **momentum** first

function Momentum $\textsc{Gradient Descent}(\boldsymbol{X}, \boldsymbol{Y}, \mathcal{L}, t, \alpha, \beta)$

$\boldsymbol{\theta} \leftarrow \text{Glorot}()$

$\boldsymbol{M} \leftarrow \boldsymbol{0}$ # Init momentum

for $i \in 1...t$ **do**

$\boldsymbol{J} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\theta})$ # Represents acceleration

$\boldsymbol{M} \leftarrow \beta \cdot \boldsymbol{M} + (1 - \beta) \cdot \boldsymbol{J}$ # Momentum and acceleration

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{M}$

return $\boldsymbol{\theta}$

Modern Optimization

Now **adaptive learning rate**

Now **adaptive learning rate**

function RMSProp$(\boldsymbol{X}, \boldsymbol{Y}, \mathcal{L}, t, \alpha, \beta, \varepsilon)$

$\quad \boldsymbol{\theta} \leftarrow \mathrm{Glorot}()$

$\quad \boldsymbol{V} \leftarrow \boldsymbol{0}$ # Init variance

$\quad$ **for** $i \in 1...t$ **do**

$\quad\quad \boldsymbol{J} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\theta})$ # Represents acceleration

$\quad\quad \boldsymbol{V} \leftarrow \beta \cdot \boldsymbol{V} + (1 - \beta) \cdot \boldsymbol{J} \odot \boldsymbol{J}$ # Magnitude

$\quad\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{J} \oslash \sqrt[\circ]{\boldsymbol{V} + \varepsilon}$ # Rescale grad by prev updates

$\quad$ **return** $\boldsymbol{\theta}$

Adaptive Optimization

Combine **momentum** and **adaptive learning rate** to create **Adam**

Adaptive Optimization

Combine **momentum** and **adaptive learning rate** to create **Adam**

function Adaptive Moment Estimation($\boldsymbol{X}, \boldsymbol{Y}, \mathcal{L}, t, \alpha, \beta_1, \beta_2, \varepsilon$)

$\boldsymbol{\theta} \leftarrow \text{Glorot}()$

$\boldsymbol{M}, \boldsymbol{V} \leftarrow \boldsymbol{0}$

for $i \in 1...t$ **do**

$\boldsymbol{J} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\theta})$

$\boldsymbol{M} \leftarrow \beta_1 \boldsymbol{M} + (1 - \beta_1) \boldsymbol{J}$ # Compute momentum

$\boldsymbol{V} \leftarrow \beta_2 \cdot \boldsymbol{V} + (1 - \beta_2) \cdot \boldsymbol{J} \odot \boldsymbol{J}$ # Magnitude

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{M} \oslash \sqrt[\odot]{\boldsymbol{V} + \varepsilon}$ # Adaptive param update

return $\boldsymbol{\theta}$ # Note, we use biased $\boldsymbol{M}, \boldsymbol{V}$ for clarity

Adaptive Optimization

```python
import torch
betas = (0.9, 0.999)
net = ...
theta = net.parameters()

sgd = torch.optim.SGD(theta, lr=alpha)
momentum = torch.optim.SGD(
    theta, lr=alpha, momentum=betas[0])
rmsprop = torch.optim.RMSprop(
    theta, lr=alpha, momentum=betas[1])
adam = torch.optim.Adam(theta, lr=alpha, betas=betas)
...
sgd.step(), momentum.step(), rmsprop.step(), adam.step()
```

Adaptive Optimization

```python
import optax
betas = (0.9, 0.999)
theta = ...

sgd = optax.sgd(lr=alpha)
momentum = optax.sgd(lr=alpha, momentum=betas[0])
rmsprop = optax.rmsprop(lr=alpha, decay=betas[1])
adam = optax.adam(lr=alpha, b1=betas[0], b2=betas[1])

v = rmsprop.init(theta)
theta, v = rmsprop.update(J, v, theta)
mv = adam.init(theta) # contains M and V
theta, mv = mv.update(J, mv, theta)
```

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. **Modern optimization**
9. Coding

Agenda

1. Review
2. Dirty secret of deep learning
3. Optimization is hard
4. Deeper neural networks
5. Activation functions
6. Parameter initialization
7. Stochastic gradient descent
8. Modern optimization
9. **Coding**