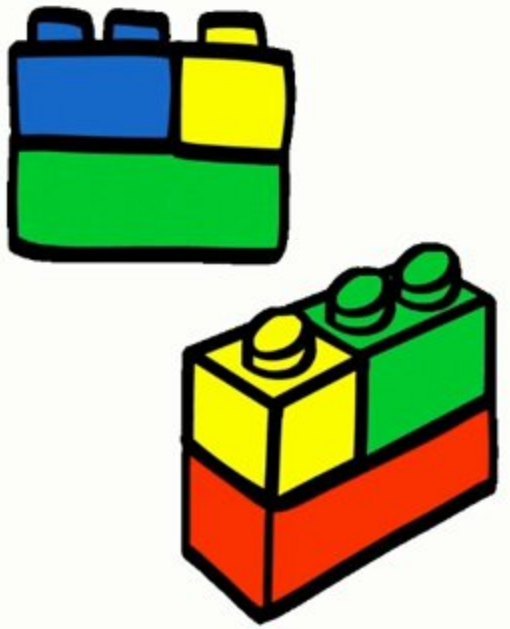
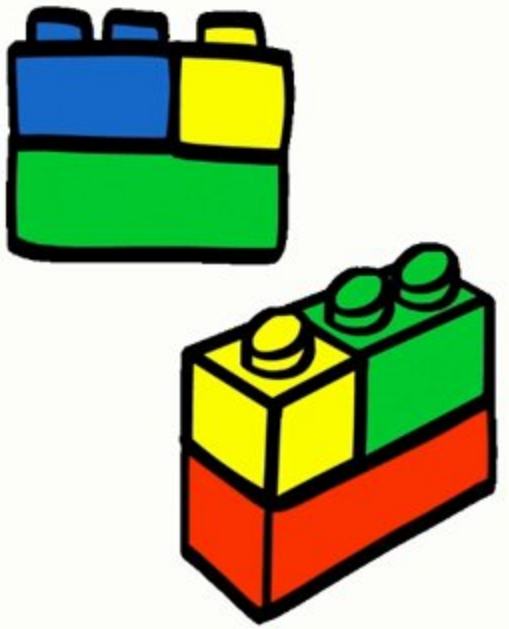
IMPORTANT ELEMENTS OF NEURAL NETWORK

- Loss Function
- Optimizer
- Activation Function

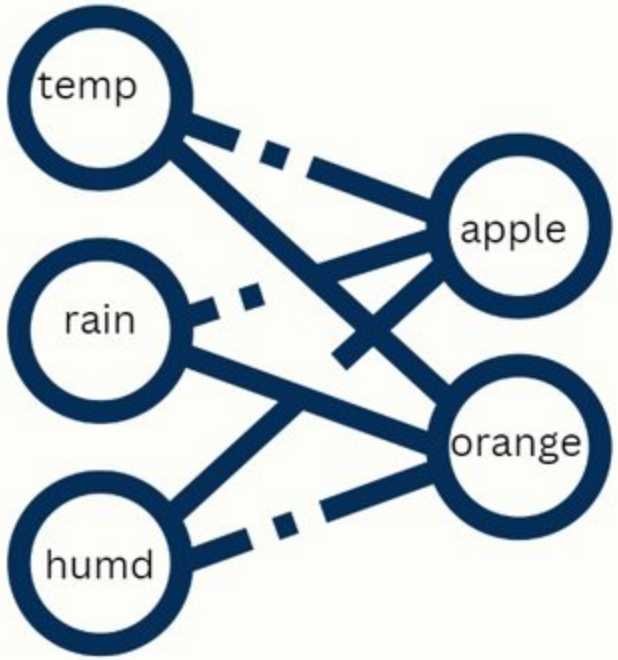


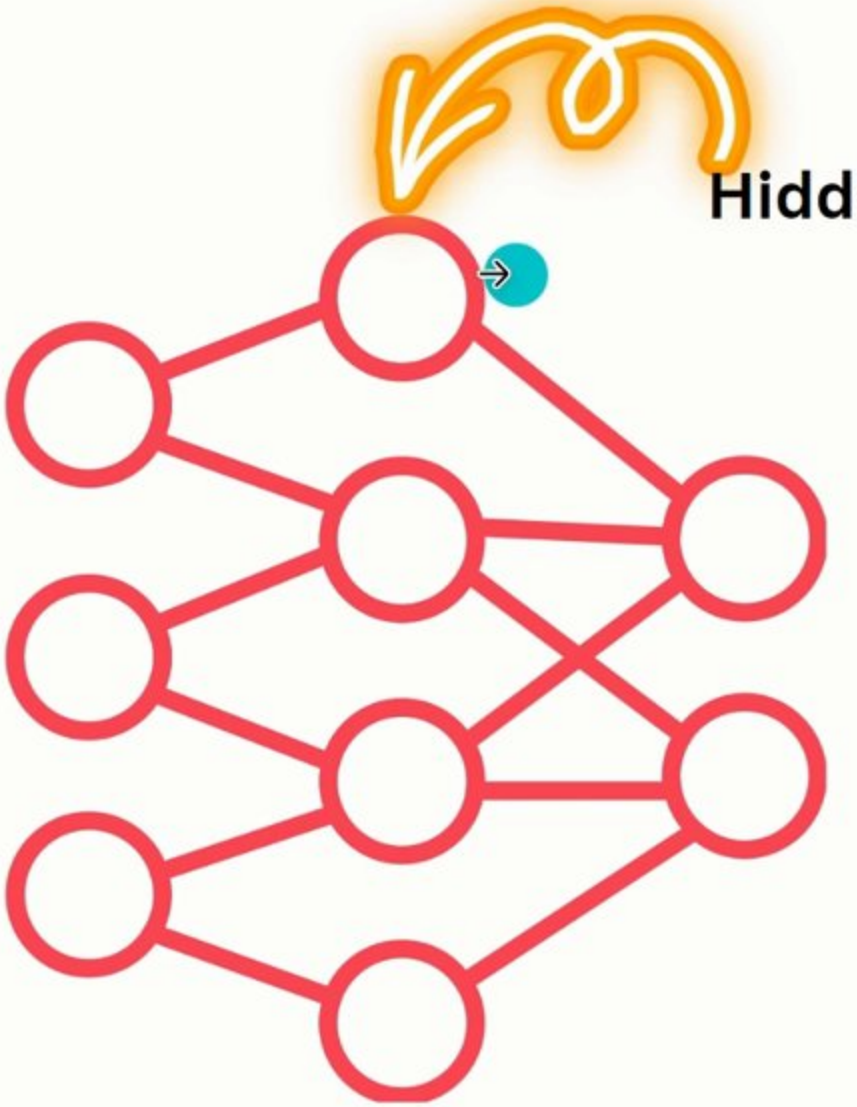
IMPORTANT ELEMENTS OF NEURAL NETWORK

- Loss Function
- Optimizer
- Activation Function



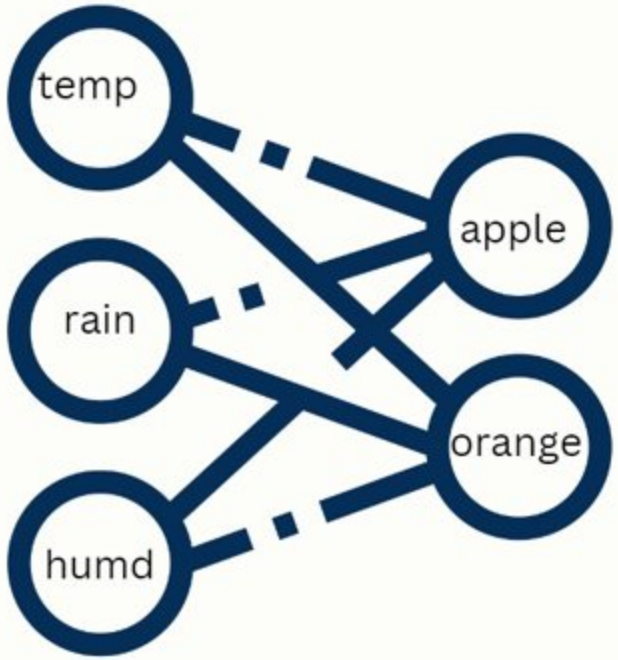
~~SIMPLE~~ NEURAL NETWORK

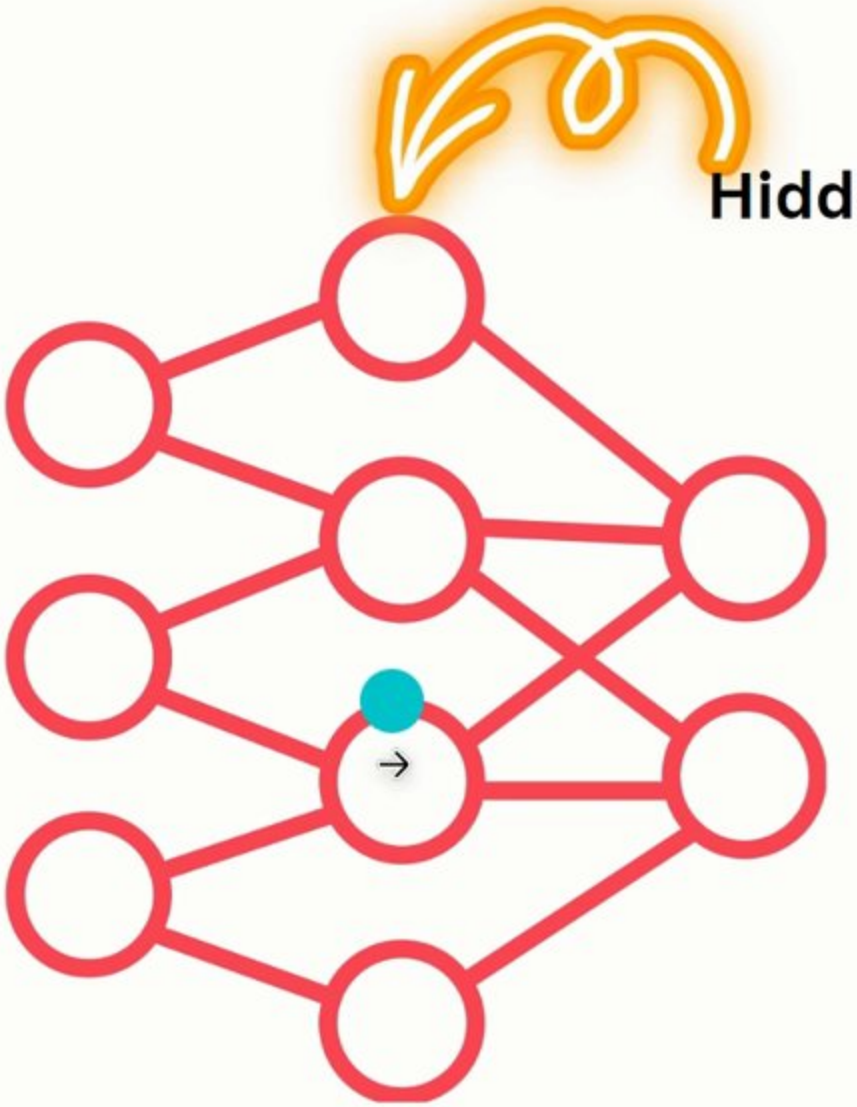




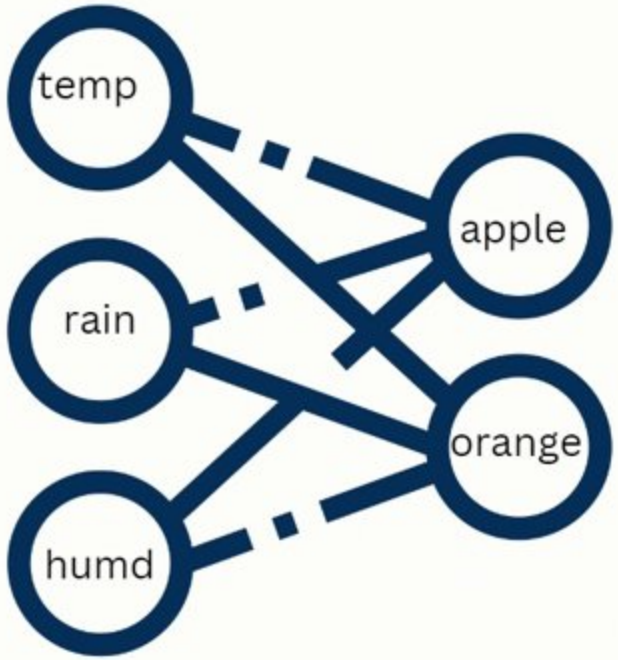
AISCIENCES

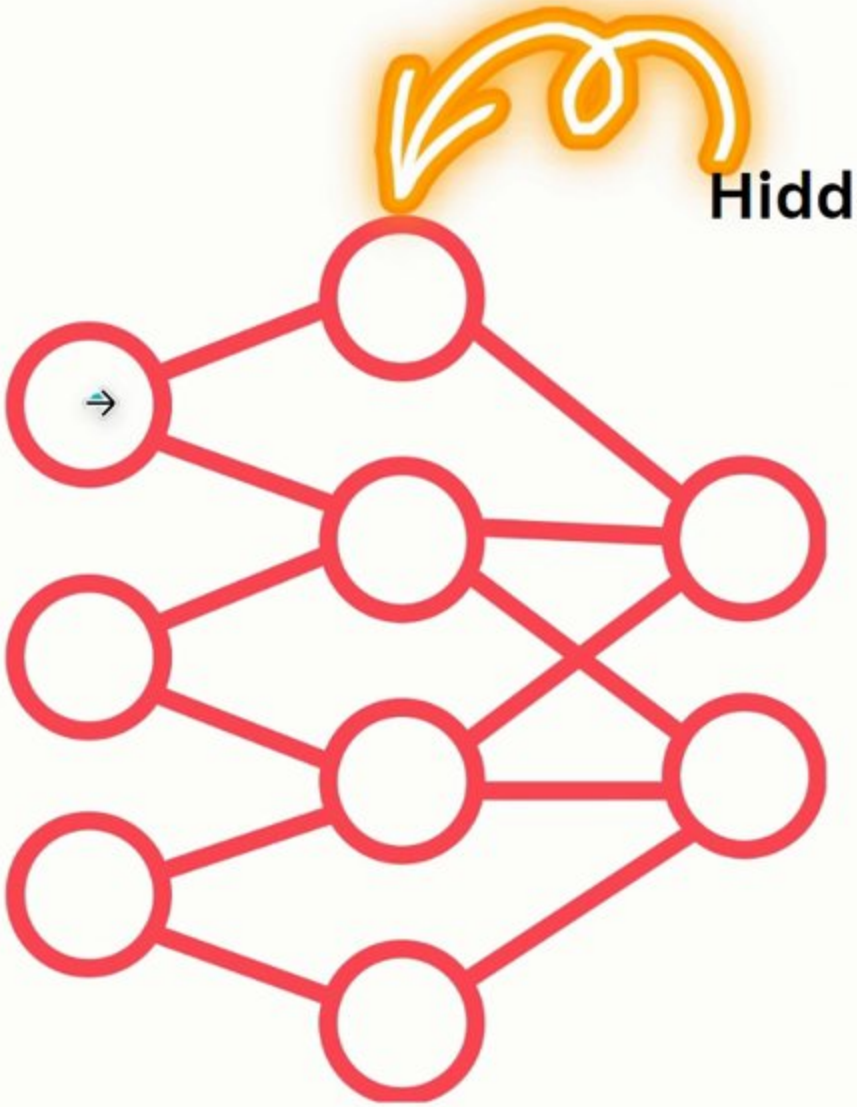
~~SIMPLE~~ NEURAL NETWORK





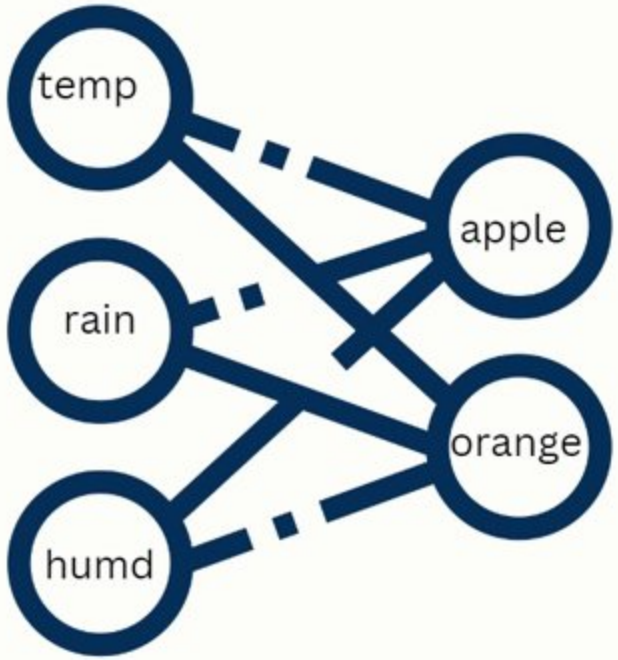
~~SIMPLE~~ NEURAL NETWORK

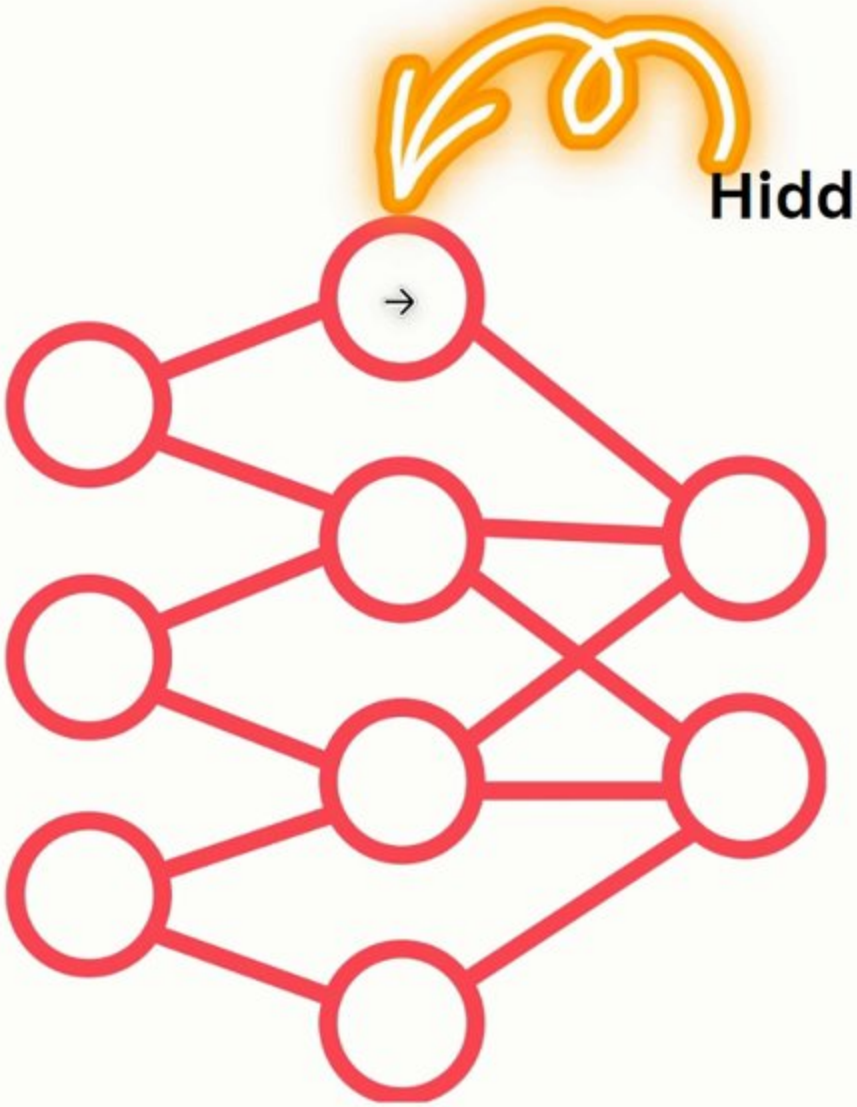




AISCIENCES

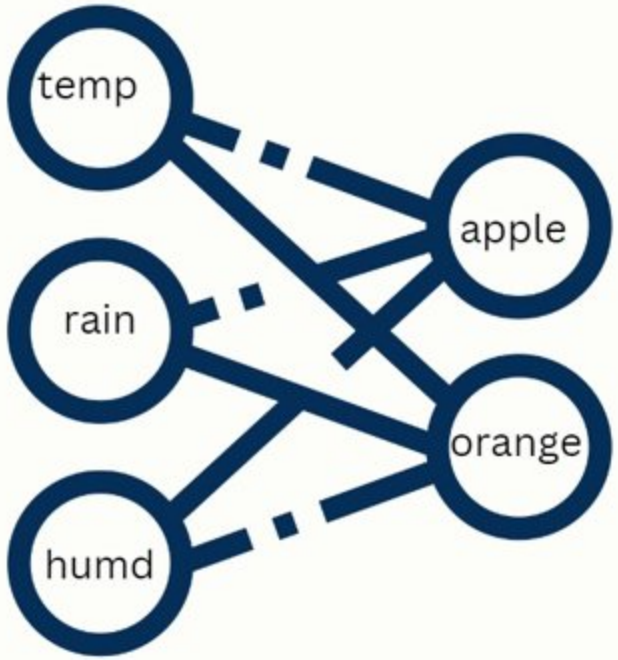
~~SIMPLE~~ NEURAL NETWORK

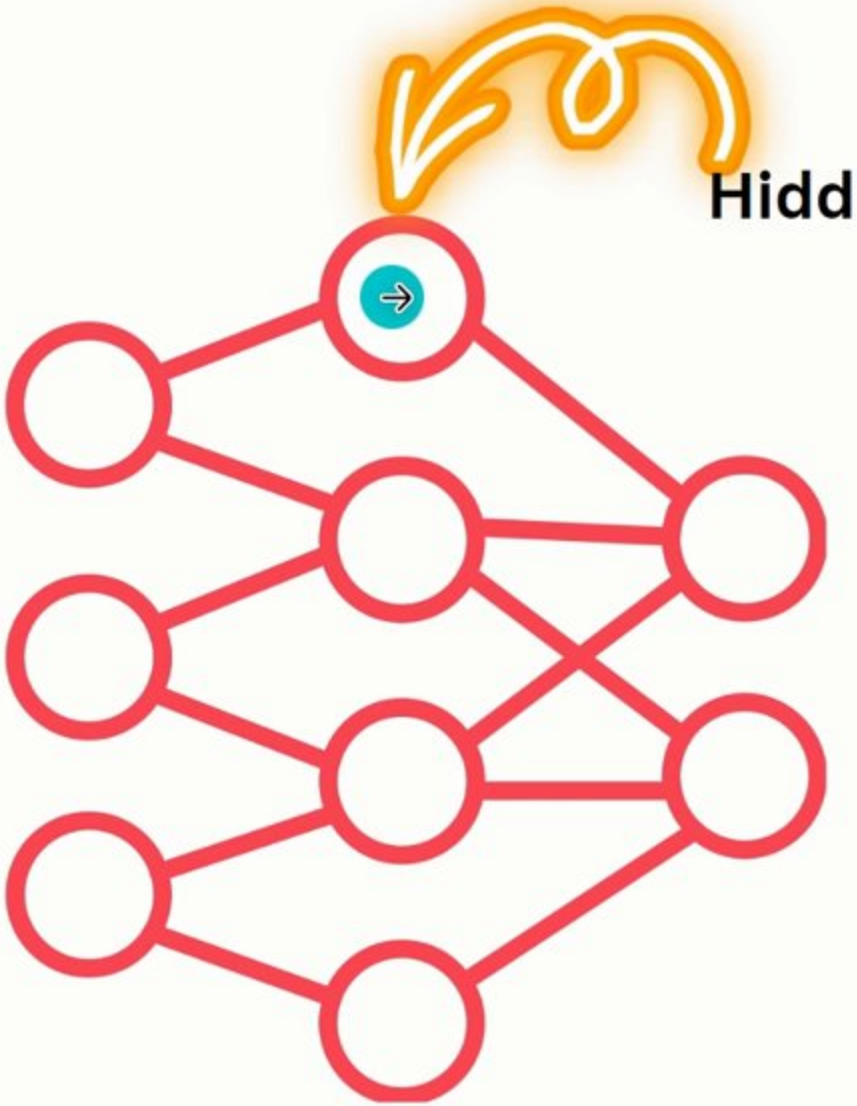




AISCIENCES

~~SIMPLE~~ NEURAL NETWORK





Softmax	Output layers for multiclass classification	Converts logits to probabilities; good for classification	computationally intensive for many classes
Softplus	Hidden layers, smooth activation	Smooth approximation of ReLU; always positive	Computationally more complex than ReLU
Softsign	Hidden layers, alternative to Tanh	Bounded outputs; reduces vanishing gradient problem	Slower convergence compared to ReLU and Tanh

Descriptions

1. **Sigmoid: Maps the input to a value between 0 and 1. Often used in the output layer of binary classification problems.**
   - **PyTorch Syntax: torch.nn.Sigmoid()**
2. **Tanh**: Maps the input to a value between -1 and 1. Often used in hidden layers and can be seen as a scaled version of the sigmoid function.
   - PyTorch Syntax: torch.nn.Tanh()
3. **ReLU (Rectified Linear Unit)**: Applies the rectifier function, which outputs the input directly if it is positive, otherwise, it outputs zero. Commonly used in hidden layers of neural networks.
   - PyTorch Syntax: torch.nn.ReLU()
4. **Leaky ReLU**: A variation of ReLU that allows a small, non-zero gradient when the input is negative, which helps to avoid dying ReLUs.
   - PyTorch Syntax: torch.nn.LeakyReLU()
5. **ELU (Exponential Linear Unit)**: An activation function that tends to converge faster and more reliably than ReLU by smoothing the transition from negative to positive inputs.
   - PyTorch Syntax: torch.nn.ELU()
6. **SELU (Scaled Exponential Linear Unit)**: A self-normalizing activation

Activation Functions

Activation Function	Use Case	Pros	Cons
Sigmoid	Binary classification, output layers	Outputs between 0 and 1; good for probabilities	Vanishing gradient problem
Tanh	Hidden layers, regression tasks	Outputs between -1 and 1; zero-centered	Vanishing gradient problem
ReLU (Rectified Linear Unit)	Hidden layers, convolutional neural networks (CNNs)	Computationally efficient; reduces vanishing gradient	Can cause dying ReLUs (neurons stop learning)
Leaky ReLU	Hidden layers, avoiding dying ReLUs	Prevents dying ReLUs by allowing small gradients	Introduces a small slope for negative inputs
ELU (Exponential Linear Unit)	Hidden layers, deep neural networks	Smooths the transition, faster and reliable convergence	More computationally expensive
SELU (Scaled Exponential Linear Unit)	Self-normalizing networks, hidden layers	Self-normalizes outputs; robust training	Requires specific initialization and dropout scaling
Softmax	Output layers for multiclass classification	Converts logits to probabilities; good for classification	Can be computationally intensive for many classes

Activation Functions

Activation Function	Use Case	Pros	Cons
Sigmoid	Binary classification, output layers	Outputs between 0 and 1; good for probabilities	Vanishing gradient problem
Tanh	Hidden layers, regression tasks	Outputs between -1 and 1; zero-centered	Vanishing gradient problem
ReLU (Rectified Linear Unit)	Hidden layers, convolutional neural networks (CNNs)	Computationally efficient; reduces vanishing gradient	Can cause dying ReLUs (neurons stop learning)
Leaky ReLU	Hidden layers, avoiding dying ReLUs	Prevents dying ReLUs by allowing small gradients	Introduces a small slope for negative inputs
ELU (Exponential Linear Unit)	Hidden layers, deep neural networks	Smooths the transition, faster and reliable convergence	More computationally expensive
SELU (Scaled Exponential Linear Unit)	Self-normalizing networks, hidden layers	Self-normalizes outputs; robust training	Requires specific initialization and dropout scaling
Softmax	Output layers for multiclass classification	Converts logits to probabilities; good for classification	Can be computationally intensive for many classes

Function			
Sigmoid	Binary classification, output layers	Outputs between 0 and 1; good for probabilities	Vanishing gradient problem
Tanh	Hidden layers, regression tasks	Outputs between -1 and 1; zero-centered	Vanishing gradient problem
ReLU (Rectified Linear Unit)	Hidden layers, convolutional neural networks (CNNs)	Computationally efficient; reduces vanishing gradient	Can cause dying ReLUs (neurons stop learning)
Leaky ReLU	Hidden layers, avoiding dying ReLUs	Prevents dying ReLUs by allowing small gradients	Introduces a small slope for negative inputs
ELU (Exponential Linear Unit)	Hidden layers, deep neural networks	Smooths the transition, faster and reliable convergence	More computationally expensive
SELU (Scaled Exponential Linear Unit)	Self-normalizing networks, hidden layers	Self-normalizes outputs; robust training	Requires specific initialization and dropout scaling
Softmax	Output layers for multiclass classification	Converts logits to probabilities; good for classification	Can be computationally intensive for many classes
Softplus	Hidden layers, smooth activation	Smooth approximation of ReLU; always positive	Computationally more complex than ReLU

Function			
Sigmoid	Binary classification, output layers	Outputs between 0 and 1; good for probabilities	Vanishing gradient problem
Tanh	Hidden layers, regression tasks	Outputs between -1 and 1; zero-centered	Vanishing gradient problem
ReLU (Rectified Linear Unit)	Hidden layers, convolutional neural networks (CNNs)	Computationally efficient; reduces vanishing gradient	Can cause dying ReLUs (neurons stop learning)
Leaky ReLU	Hidden layers, avoiding dying ReLUs	Prevents dying ReLUs by allowing small gradients	Introduces a small slope for negative inputs
ELU (Exponential Linear Unit)	Hidden layers, deep neural networks	Smooths the transition, faster and reliable convergence	More computationally expensive
SELU (Scaled Exponential Linear Unit)	Self-normalizing networks, hidden layers	Self-normalizes outputs; robust training	Requires specific initialization and dropout scaling
Softmax	Output layers for multiclass classification	Converts logits to probabilities; good for classification	Can be computationally intensive for many classes
Softplus	Hidden layers, smooth activation	Smooth approximation of ReLU; always positive	Computationally more complex than ReLU

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

google.com/search?sca_esv=4aa4cadb6d3a9fe9&rlz=1C1CHBF_enPK916PK916&sxsrf=ADLYWILnqnNSTxzpcVzJCfzJ577ZAlOVLQ:1...

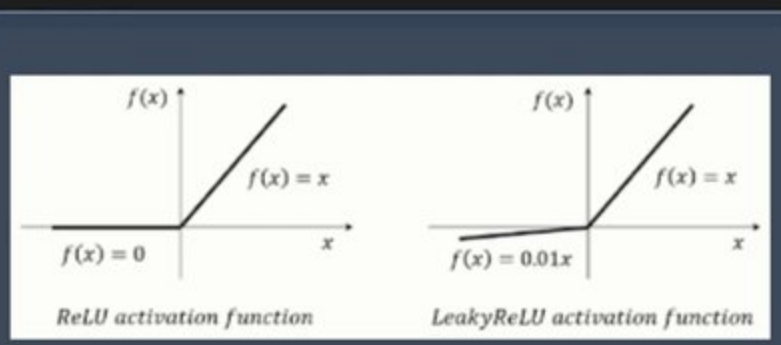
Paused

Google

relu vs leaky relu

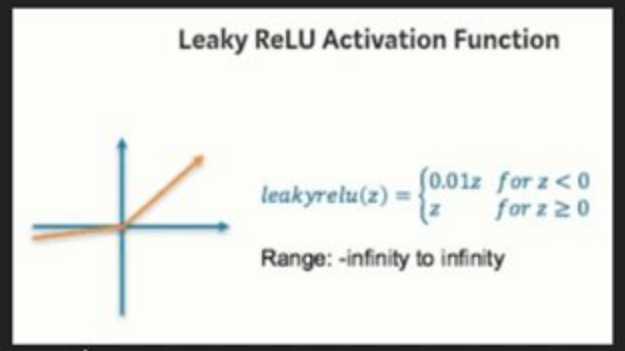
All | Images | Videos | News | Shopping | Books | Web | More | Tools | Saved

Sigmoid | Softmax | Prelu activation | Neural networks | Function | Linear | Swish activation | Paran



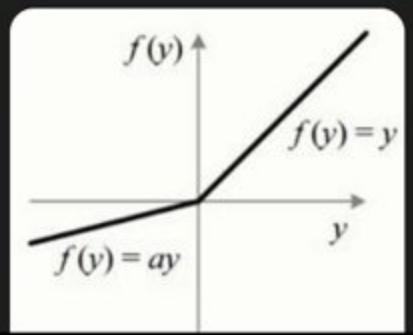
ResearchGate

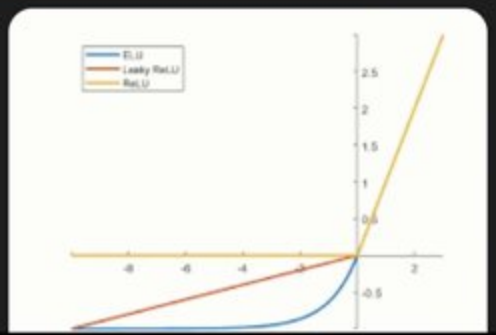
ReLU activation function vs. LeakyReLU ...

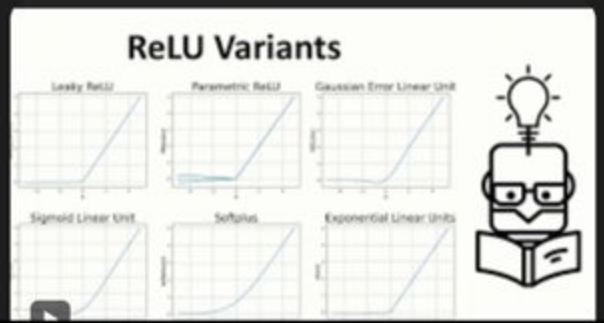


Medium

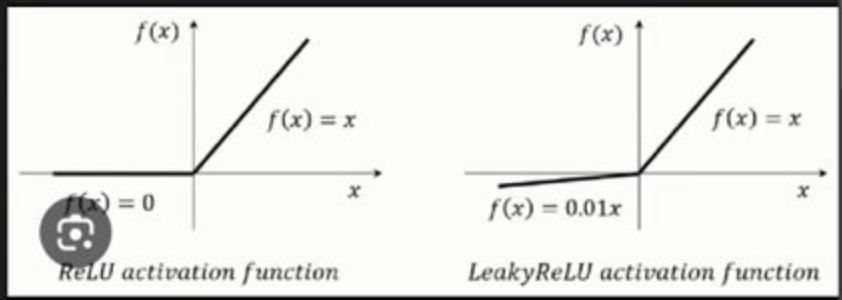
Activation functions: ReLU vs. Leaky ...







ResearchGate



ReLU activation function vs. LeakyReLU activation function. |...

Visit

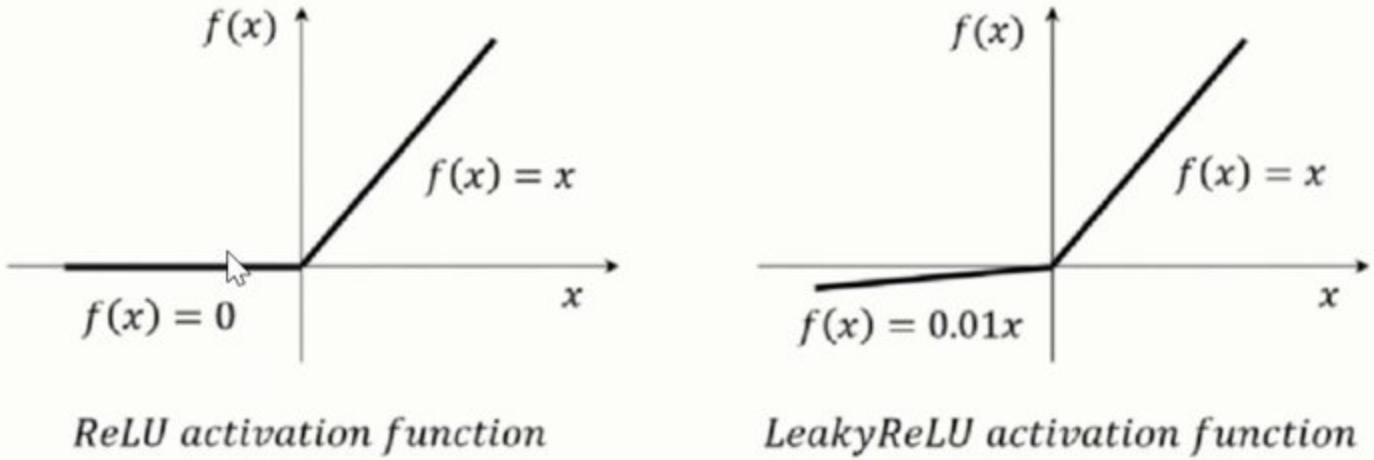
Images may be subject to copyright. Learn More

Share | Save

AISCIENCES

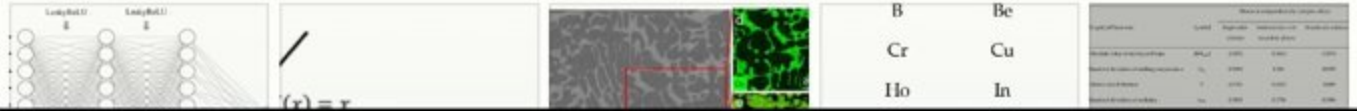
udemy

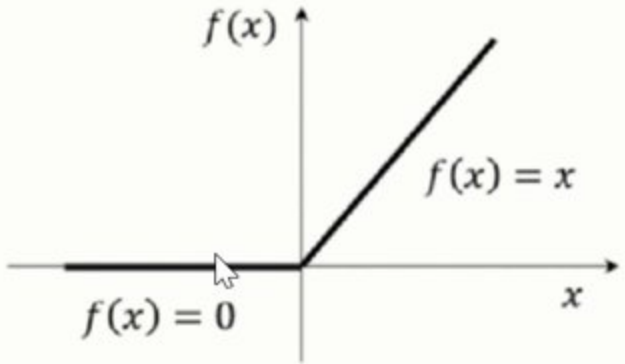
Download View publication



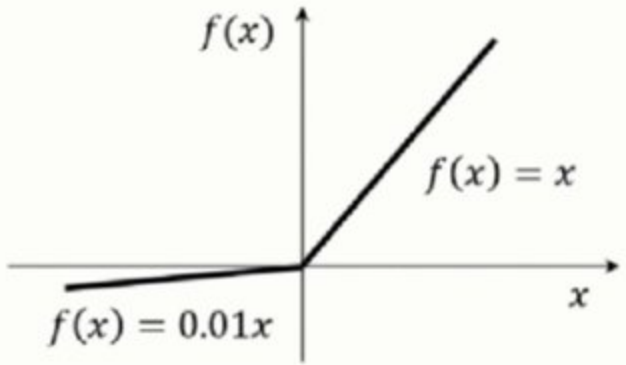
ReLU activation function vs. LeakyReLU activation function.

Source publication



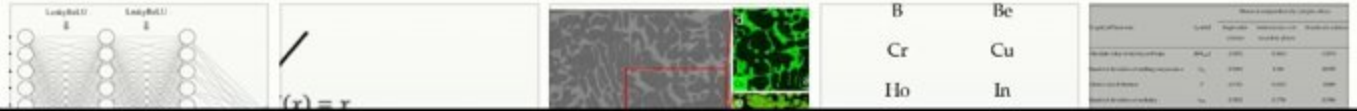


ReLU activation function



LeakyReLU activation function

ReLU activation function vs. LeakyReLU activation function.



```
[ 76.,  85.],
[ 65.,  75.]])
```

```python
#Replacing Model with Built-in Model
class SimpleNN(nn)

#Replacing MSE with built-in function
# mse = torch.nn.MSELoss()
mae = torch.nn.L1Loss()

# Training step
for i in range(1000):
  preds = model(inputs)
  loss = mae(preds, targets)
  loss.backward()
  if i%100 == 99:
    print(loss.item())

  with torch.no_grad():
    w -= w.grad * 0.00001
    b -= b.grad * 0.00001
    w.grad.zero_()
    b.grad.zero_()

preds = model(inputs)
loss = mae(preds, targets)
print(loss)
```

```
126.0269546508789
```

```
        [ 76.,  85.],
        [ 65.,  75.]])
```

```
#Replacing Model with Built-in Model
cal

#Replacing MSE with built-in function
# mse = torch.nn.MSELoss()
mae = torch.nn.L1Loss()

# Training step
for i in range(1000):
  preds = model(inputs)
  loss = mae(preds, targets)
  loss.backward()
  if i%100 == 99:
    print(loss.item())

  with torch.no_grad():
    w -= w.grad * 0.00001
    b -= b.grad * 0.00001
    w.grad.zero_()
    b.grad.zero_()

preds = model(inputs)
loss = mae(preds, targets)
print(loss)
```

```
126.0269546508789
```

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

colab.research.google.com/drive/1imQs4frEn2sYP0VW7SjxgJwC5ZgNZdpv#scrollTo=BZYFUl5XqzSc

Paused

+ Code + Text All changes saved

T4 RAM Disk

[11] 0s

```
[103., 119.],
[ 98., 110.],
[ 88.,  95.],
[115., 140.],
[ 76.,  85.],
[ 65.,  75.]])
```

```
w = torch.randn(3,2, requires_grad=True)
b = torch.randn(2, requires_grad=True)

# Defining Model
def model(x):
  # return (x @ w) + b
  return torch.matmul(x,w)+b

#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()



#Replacing MSE with built-in function
# mse = torch.nn.MSELoss()
mae = torch.nn.L1Loss()

# Training step
for i in range(1000):
```

AISCIENCES

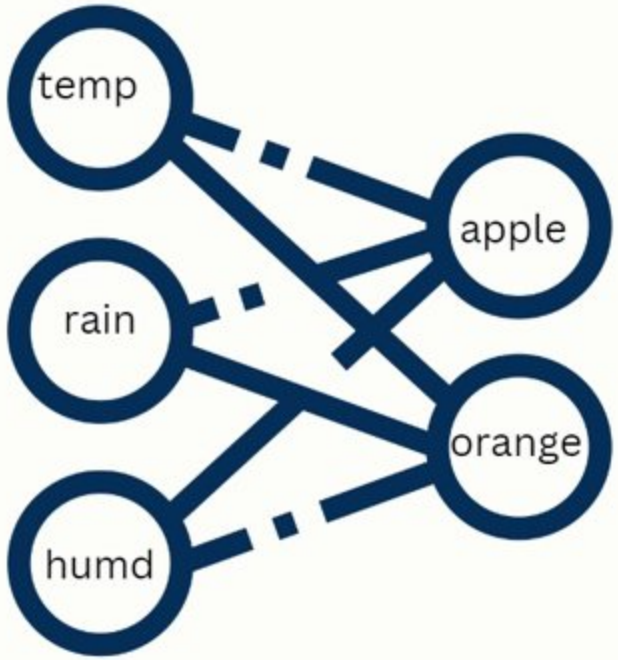
0s completed at 3:11 PM

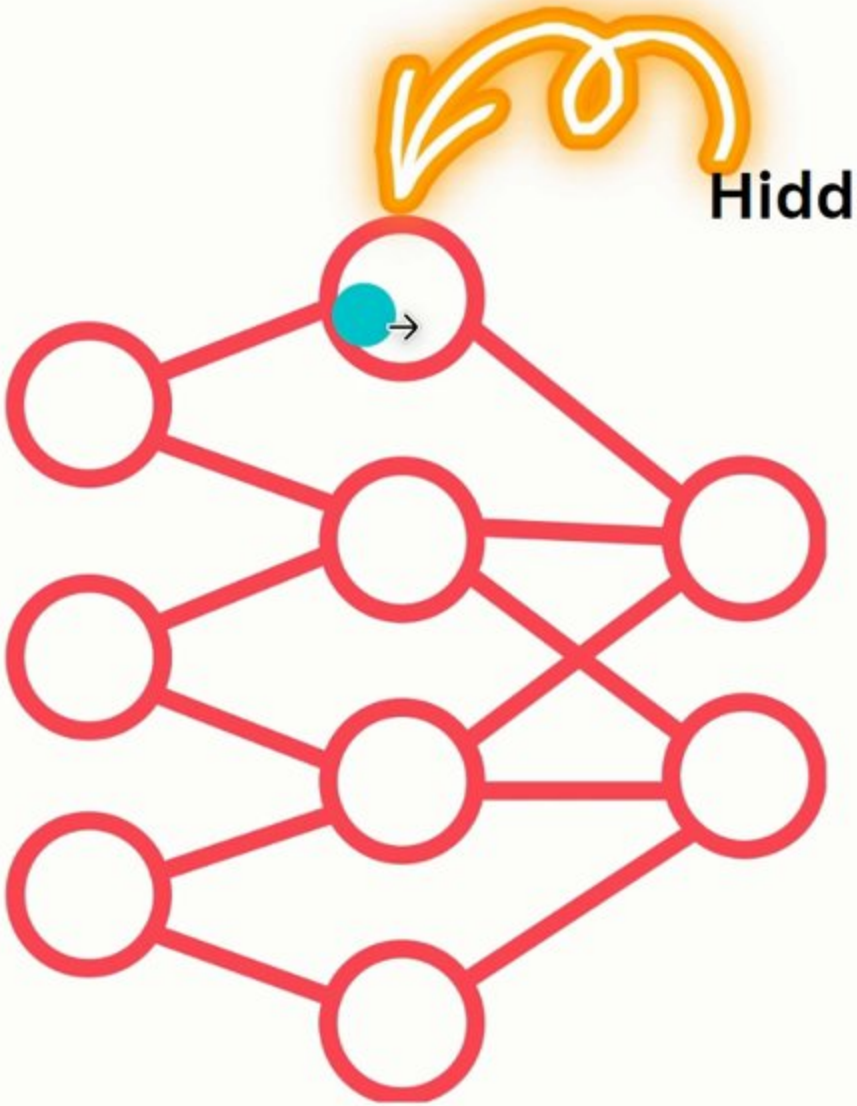
[11]

```
[103., 119.],
[ 98., 110.],
[ 88.,  95.],
[115., 140.],
[ 76.,  85.],
[ 65.,  75.]])
```

```
w = torch.randn(3,2, requires_grad=True)
b = torch.randn(2, requires_grad=True)

# Defining Model
def model(x):
  # return (x @ w) + b
  return torch.matmul(x,w)+b

#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.linear()



#Replacing MSE with built-in function
# mse = torch.nn.MSELoss()
mae = torch.nn.L1Loss()

# Training step
for i in range(1000):
```

~~SIMPLE~~ NEURAL NETWORK





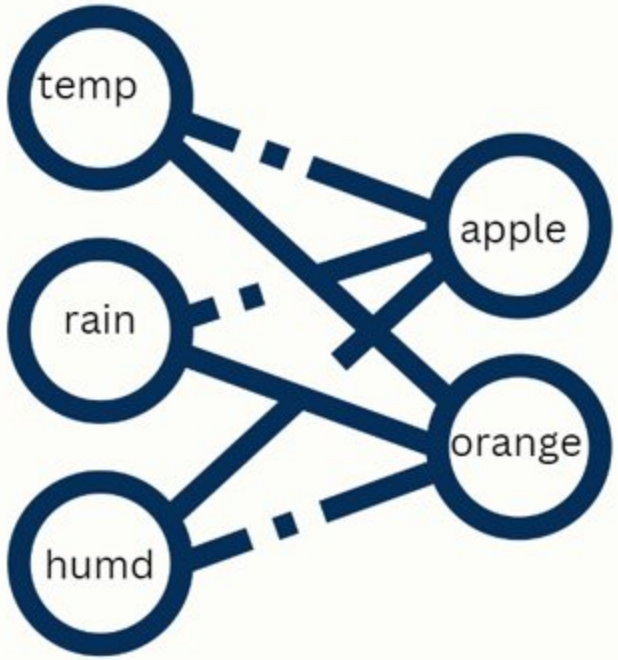
```
[103., 119.],
[ 98., 110.],
[ 88.,  95.],
[115., 140.],
[ 76.,  85.],
[ 65.,  75.]])
```

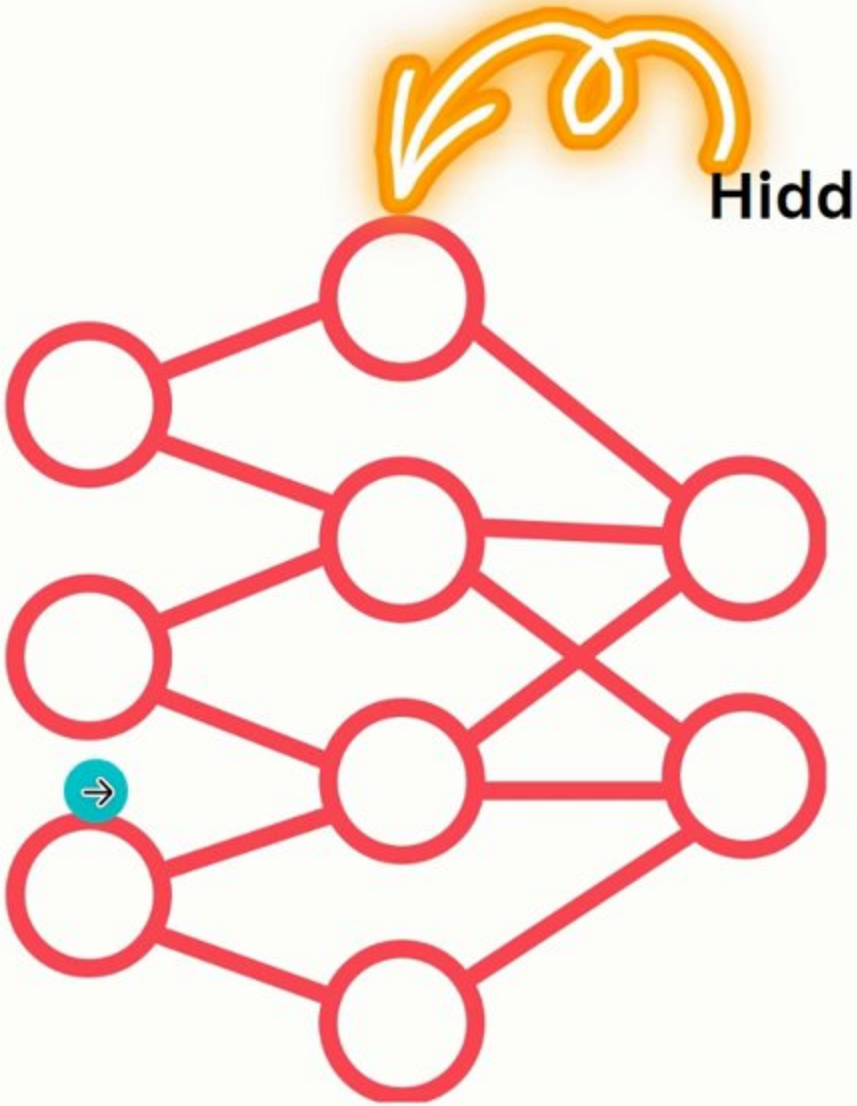
```
w = torch.randn(3,2, requires_grad=True)
b = torch.randn(2, requires_grad=True)

# Defining Model
def model(x):
  # return (x @ w) + b
  return torch.matmul(x,w)+b

#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1
    self.linear2 = nn.Linear(4, 2)



#Replacing MSE with built-in function
# mse = torch.nn.MSELoss()
mae = torch.nn.L1Loss()
```

```
[103., 119.],
[ 98., 110.],
[ 88.,  95.],
[115., 140.],
[ 76.,  85.],
[ 65.,  75.]])
```

```
w = torch.randn(3,2, requires_grad=True)
b = torch.randn(2, requires_grad=True)

# Defining Model
def model(x):
  # return (x @ w) + b
  return torch.matmul(x,w)+b

#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1 = nn.ReL
    self.linear2 = nn.Linear(4, 2)



#Replacing MSE with built-in function
# mse = torch.nn.MSELoss()
mae = torch.nn.L1Loss()
```

~~SIMPLE~~ NEURAL NETWORK





```
        [ 65.,  75.]])
```

```python
w = torch.randn(3,2, requires_grad=True)
b = torch.randn(2, requires_grad=True)

# Defining Model
def model(x):
  # return (x @ w) + b
  return torch.matmul(x,w)+b

#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1 = nn.ReLU()
    self.linear2 = nn.Linear(4, 2)

  def forward(self, x):
    x = self.linear1(x)
    x = self.act1(x)
    x = self.linear2(x)
    return x



#Replacing MSE with built-in function
# mse = torch.nn.MSELoss()
mae = torch.nn.L1Loss()
```

```
        [ 65.,  75.]])
```

```python
w = torch.randn(3,2, requires_grad=True)
b = torch.randn(2, requires_grad=True)

# Defining Model
def model(x):
  # return (x @ w) + b
  return torch.matmul(x,w)+b

#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1 = nn.ReLU()
    self.linear2 = nn.Linear(4, 2)

  def forward(self, x):
    x = self.linear1(x)
    x = self.act1(x)
    x = self.linear2(x)
    return x



#Replacing MSE with built-in function
```

```
        [ 65.,  75.]])
```

```
w = torch.randn(3,2, requires_grad=True)
b = torch.randn(2, requires_grad=True)

# Defining Model
def model(x):
  # return (x @ w) + b
  return torch.matmul(x,w)+b

#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1 = nn.ReLU()
    self.linear2 = nn.Linear(4, 2)

  def forward(self, x):
    x = self.linear1(x)
    x = self.act1(x)
    x = self.linear2(x)
    return x



#Replacing MSE with built-in function
```

```
        [ 65.,  75.]])
```

```python
#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1 = nn.ReLU()
    self.linear2 = nn.Linear(4, 2)

  def forward(self, x):
    x = self.linear1(x)
    x = self.act1(x)
    x = self.linear2(x)
    return x



#Replacing MSE with built-in function
# mse = torch.nn.MSELoss()
mae = torch.nn.L1Loss()

# Training step
for i in range(1000):
  preds = model(inputs)
```

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

colab.research.google.com/drive/1imQs4frEn2sYP0VW7SjxgJwC5ZgNZdpv#scrollTo=BZYFUl5XqzSc

Paused

+ Code + Text

RAM Disk

```
# # Training step
# for i in range(1000):
#     preds = model(inputs)
#     loss = mae(preds, targets)
#     loss.backward()
#     if i%100 == 99:
#         print(loss.item())

#     with torch.no_grad():
#         w -= w.grad * 0.00001
#         b -= b.grad * 0.00001
#         w.grad.zero_()
#         b.grad.zero_()

# preds = model(inputs)
# loss = mae(preds, targets)
# print(loss)
```

```
126.0269546508789
116.77412414550781
107.52127838134766
98.26844787597656
89.01560974121094
80.09945678710938
72.49869537353516
66.56382751464844
61.56400680541992
57.83696365356445
tensor(57.8033, grad_fn=<MeanBackward0>)
```

AISCIENCES

[ ]

✓ 7s completed at 3:43 PM

udemy

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

colab.research.google.com/drive/1imQs4frEn2sYP0VW7SjxgJwC5ZgNZdpv#scrollTo=BZYFUl5XqzSc

Paused

+ Code + Text

RAM Disk

```
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1 = nn.ReLU()
    self.linear2 = nn.Linear(4, 2)

  def forward(self, x):
    x = self.linear1(x)
    x = self.act1(x)
    x = self.linear2(x)
    return x

model = SimpleNN()


# #Replacing MSE with built-in function
# # mse = torch.nn.MSELoss()
# mae = torch.nn.L1Loss()

# # Training step
# for i in range(1000):
#   preds = model(inputs)
#   loss = mae(preds, targets)
#   loss.backward()
#   if i%100 == 99:
#     print(loss.item())

#   with torch.no_grad():
```

AISCIENCES

7s completed at 3:43 PM

udemy

```python
#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1 = nn.ReLU()
    self.linear2 = nn.Linear(4, 2)

  def forward(self, x):
    x = self.linear1(x)
    x = self.act1(x)
    x = self.linear2(x)
    return x

model = SimpleNN()
for name, params in model.named_parameters():
  print("Name:", name)
  print("Param",)



# #Replacing MSE with built-in function
# # mse = torch.nn.MSELoss()
# mae = torch.nn.L1Loss()

# # Training step
# for i in range(1000):
#   preds = model(inputs)
#   loss = mae(preds, targets)
```

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

colab.research.google.com/drive/1imQs4frEn2sYP0VW7SjxgJwC5ZgNZdpv#scrollTo=B6Ymnu54qzSb

Paused

+ Code + Text All changes saved

RAM Disk

Basic Neural Network with Built-in functions

Using Pytorch

7s

```
import torch
import numpy as np
import torch.nn.functional as F

# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70],
                   [85, 100, 60],
                   [95, 80, 55],
                   [105, 120, 75],
                   [78, 90, 50],
                   [82, 70, 45]], dtype='float32')

# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119],
```

AISCIENCES

0s completed at 3:44 PM

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

colab.research.google.com/drive/1imQs4frEn2sYP0VW7SjxgJwC5ZgNZdpv#scrollTo=B6Ymnu54qzSb

Paused

+ Code + Text

RAM Disk

Basic Neural Network with Built-in functions

Using Pytorch

```
import torch
import numpy as np
import torch.nn as nn

# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70],
                   [85, 100, 60],
                   [95, 80, 55],
                   [105, 120, 75],
                   [78, 90, 50],
                   [82, 70, 45]], dtype='float32')

# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119],
```


0s completed at 3:44 PM

```
#   if i%100 == 99:
#     print(loss.item())

#   with torch.no_grad():
#     w -= w.grad * 0.00001
#     b -= b.grad * 0.00001
#     w.grad.zero_()
#     b.grad.zero_()

# preds = model(inputs)
# loss = mae(preds, targets)
# print(loss)
```

```
Name: linear1.weight
Param Parameter containing:
tensor([[-0.1478, -0.5037, -0.5325],
        [ 0.3457,  0.3467, -0.5086],
        [-0.1229, -0.5641, -0.5510],
        [-0.3043,  0.1261, -0.0497]], requires_grad=True)
Name: linear1.bias
Param Parameter containing:
tensor([0.4177, 0.4991, 0.3199, 0.0991], requires_grad=True)
Name: linear2.weight
Param Parameter containing:
tensor([[ 0.3285,  0.4916, -0.0986,  0.4325],
        [ 0.4481,  0.4910, -0.3399, -0.2481]], requires_grad=True)
Name: linear2.bias
Param Parameter containing:
tensor([0.4516, 0.2715], requires_grad=True)
```

```
[ ]
```

```
#   if i%100 == 99:
#     print(loss.item())

#   with torch.no_grad():
#     w -= w.grad * 0.00001
#     b -= b.grad * 0.00001
#     w.grad.zero_()
#     b.grad.zero_()

# preds = model(inputs)
# loss = mae(preds, targets)
# print(loss)
```

```
Name: linear1.weight
Param Parameter containing:
tensor([[-0.1478, -0.5037, -0.5325],
        [ 0.3457,  0.3467, -0.5086],
        [-0.1229, -0.5641, -0.5510],
        [-0.3043,  0.1261, -0.0497]], requires_grad=True)
Name: linear1.bias
Param Parameter containing:
tensor([0.4177, 0.4991, 0.3199, 0.0991], requires_grad=True)
Name: linear2.weight
Param Parameter containing:
tensor([[ 0.3285,  0.4916, -0.0986,  0.4325],
        [ 0.4481,  0.4910, -0.3399, -0.2481]], requires_grad=True)
Name: linear2.bias
Param Parameter containing:
tensor([0.4516, 0.2715], requires_grad=True)
```



```
[ ]
```

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

colab.research.google.com/drive/1imQs4frEn2sYP0VW7SjxgJwC5ZgNZdpv#scrollTo=BZYFUl5XqzSc

Paused

+ Code + Text All changes saved

RAM Disk

```
#     b.grad.zero_()

# preds = model(inputs)
# loss = mae(preds, targets)
# print(loss)
```

```
Name: linear1.weight
Param Parameter containing:
tensor([[-0.1478, -0.5037, -0.5325],
        [ 0.3457,  0.3467, -0.5086],
        [-0.1229, -0.5641, -0.5510],
        [-0.3043,  0.1261, -0.0497]], requires_grad=True)
Name: linear1.bias
Param Parameter containing:
tensor([0.4177, 0.4991, 0.3199, 0.0991], requires_grad=True)
Name: linear2.weight
Param Parameter containing:
tensor([[ 0.3285,  0.4916, -0.0986,  0.4325],
        [ 0.4481,  0.4910, -0.3399, -0.2481]], requires_grad=True)
Name: linear2.bias
Param Parameter containing:
tensor([0.4516, 0.2715], requires_grad=True)
```

[ ]

[ ]

[ ]

AISCIENCES

0s completed at 3:45PM

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

colab.research.google.com/drive/1imQs4frEn2sYP0VW7SjxgJwC5ZgNZdpv#scrollTo=BZYFUl5XqzSc

Paused

+ Code + Text All changes saved

RAM Disk

```
#     b.grad.zero_()

# preds = model(inputs)
# loss = mae(preds, targets)
# print(loss)
```

```
Name: linear1.weight
Param Parameter containing:
tensor([[-0.1478, -0.5037, -0.5325],
        [ 0.3457,  0.3467, -0.5086],
        [-0.1229, -0.5641, -0.5510],
        [-0.3043,  0.1261, -0.0497]], requires_grad=True)
Name: linear1.bias
Param Parameter containing:
tensor([0.4177, 0.4991, 0.3199, 0.0991], requires_grad=True)
Name: linear2.weight
Param Parameter containing:
tensor([[ 0.3285,  0.4916, -0.0986,  0.4325],
        [ 0.4481,  0.4910, -0.3399, -0.2481]], requires_grad=True)
Name: linear2.bias
Param Parameter containing:
tensor([0.4516, 0.2715], requires_grad=True)
```

[ ]

[ ]

[ ]

AISCIENCES

0s completed at 3:45PM

udemy

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

colab.research.google.com/drive/1imQs4frEn2sYP0VW7SjxgJwC5ZgNZdpv#scrollTo=BZYFUl5XqzSc

Paused

+ Code + Text All changes saved

RAM Disk

```
#     b.grad.zero_()

# preds = model(inputs)
# loss = mae(preds, targets)
# print(loss)
```

```
Name: linear1.weight
Param Parameter containing:
tensor([[-0.1478, -0.5037, -0.5325],
        [ 0.3457,  0.3467, -0.5086],
        [-0.1229, -0.5641, -0.5510],
        [-0.3043,  0.1261, -0.0497]], requires_grad=True)
Name: linear1.bias
Param Parameter containing:
tensor([0.4177, 0.4991, 0.3199, 0.0991], requires_grad=True)
Name: linear2.weight
Param Parameter containing:
tensor([[ 0.3285,  0.4916, -0.0986,  0.4325],
        [ 0.4481,  0.4910, -0.3399, -0.2481]], requires_grad=True)
Name: linear2.bias
Param Parameter containing:
tensor([0.4516, 0.2715], requires_grad=True)
```

[ ]

[ ]

[ ]

AISCIENCES

0s completed at 3:45PM

udemy

```
[3]         [ 98., 110.],
            [ 88.,  95.],
            [115., 140.],
            [ 76.,  85.],
            [ 65.,  75.]])
```

```
#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1 = nn.ReLU()
    self.linear2 = nn.Linear(4, 2)

  def forward(self, x):
    x = self.linear1(x)
    x = self.act1(x)
    x = self.linear2(x)
    return x

model = SimpleNN()
for name, params in model.named_parameters():
  print("Name:", name)
  print("Param", params)



# #Replacing MSE with built-in function
# # mse = torch.nn.MSELoss()
```

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

colab.research.google.com/drive/1imQs4frEn2sYP0VW7SjxgJwC5ZgNZdpv#scrollTo=BZYFUl5XqzSc

Paused

+ Code + Text All changes saved

RAM Disk

```
[3]          [115., 140.],
             [ 76.,  85.],
             [ 65.,  75.]])
```

```python
#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1 = nn.ReLU()
    self.linear2 = nn.Linear(4, 2)

  def forward(self, x):
    x = self.linear1(x)
    x = self.act1(x)
    x = self.linear2(x)
    return x

model = SimpleNN()
for name, params in model.named_parameters():
  print("Name:", name)
  print("Param", params)



# #Replacing MSE with built-in function
# # mse = torch.nn.MSELoss()
# mae = torch.nn.L1Loss()
```


0s completed at 3:45 PM

Pytorch.ipynb - Colab | Activation Function.docx.pdf | relu vs leaky relu - Google Sear | ReLU activation function vs. Lea

colab.research.google.com/drive/1imQs4frEn2sYP0VW7SjxgJwC5ZgNZdpv#scrollTo=BZYFUl5XqzSc

Paused

+ Code + Text All changes saved

RAM Disk

```
[3]          [115., 140.],
             [ 76.,  85.],
             [ 65.,  75.]])
```

```python
#Replacing Model with built-in function
class SimpleNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear1 = nn.Linear(3, 4)
    self.act1 = nn.ReLU()
    self.linear2 = nn.Linear(4, 2)

  def forward(self, x):
    x = self.linear1(x)
    x = self.act1(x)
    x = self.linear2(x)
    return x

model = SimpleNN()
for name, params in model.named_parameters():
  print("Name:", name)
  print("Param", params)



# #Replacing MSE with built-in function
# # mse = torch.nn.MSELoss()
# mae = torch.nn.L1Loss()
```


0s completed at 3:45 PM