Rules of Thumb for Activation Functions in Neural Networks and Matrix Shape Notes

Rules of Thumb for Activation Functions in Neural Networks

Overview

Activation functions are essential in neural networks. They introduce non-linearity, help models learn complex patterns, and influence training stability. Below are the key rules of thumb.

General Guidelines

- Avoid Sigmoid in deep networks – can cause vanishing gradients and slow convergence; still useful in the output layer for binary classification.
- ReLU is usually the default choice – simple, efficient, reduces vanishing gradient, but may suffer from "dying ReLU" problem.
- Leaky ReLU or variants (PReLU, ELU) help when ReLU neurons die, since they allow small gradients for negative inputs.
- Use Tanh if negative outputs are useful – zero-centered but still prone to vanishing gradients in deeper nets.
- Softmax is for multi-class classification outputs – converts logits into probability distribution across classes.
- Linear activation for regression tasks – no activation in the final layer if predicting continuous values.

Practical Rules of Thumb

- Hidden layers: Start with ReLU; try Leaky ReLU/ELU if training is unstable.
- Binary classification output: Use sigmoid in the final layer.

- Multi-class classification output: Use softmax in the final layer.
- Regression output: Use linear activation (no activation) in the final layer.
- Small/shallow networks: Tanh may sometimes work better.
- RNNs: Typically use tanh or sigmoid inside cells; ReLU in newer architectures.
- Normalization: Combine batch normalization with ReLU for more stable training.

Quick Mnemonic

- ReLU = default for hidden layers
- Sigmoid = binary output
- Softmax = multi-class output
- Linear = regression output
- Tanh = centered data / small networks

General Guidelines (Detailed)

Activation Function Guidelines

- **Sigmoid:** Range: $(0, 1)$; used in binary output. Decision rule: if output $\geq 0.5 \rightarrow$ class 1, else class 0. Problems: vanishing gradients, outputs not zero-centered.
- **Tanh:** Range: $(-1, 1)$; zero-centered and generally better than sigmoid. Useful in shallow networks or RNNs but still prone to vanishing gradients in deep networks.
- **ReLU (Rectified Linear Unit):** Range: $[0, \infty)$; default activation for hidden layers. Fast and efficient; helps reduce vanishing gradients. Risk: "dying ReLU" problem — neurons stuck at zero output.
- **Leaky ReLU / PReLU / ELU:** Range: $(-\infty, \infty)$; allows a small non-zero gradient for negative inputs. Solves the dead neuron issue in standard ReLU. Recommended when ReLU performance stagnates.
- **Softmax:** Range: $(0, 1)$ for each output neuron, with all outputs summing to 1. Converts logits into a probability distribution. Used exclusively in the output layer for multi-class classification.
- **Linear (Identity Function):** Range: $(-\infty, \infty)$; used in regression outputs. No squashing effect — allows real-valued predictions without restriction.

Activation Function Comparison Table

gray!20 Activation	Value Range	Typical Use Case	Notes
Sigmoid (σ)	$(0, 1)$	Binary classification output	Threshold at 0.5 $\rightarrow$ label 1 else 0; not good for hidden layers (vanishing gradients).
Tanh	$(-1, 1)$	Shallow networks, RNNs	Zero-centered; better than sigmoid but still vanishes in deep networks.
ReLU	$[0, \infty)$	Default for hidden layers	Fast, efficient, but risk of "dying ReLU."
Leaky ReLU / PReLU / ELU	$(-\infty, \infty)$	Hidden layers when ReLU neurons die	Allows small gradient for negative inputs; mitigates dead neurons.
Softmax	$(0, 1)$, sum $=$ 1	Multi-class classification output	Converts logits into a normalized probability distribution.
Linear	$(-\infty, \infty)$	Regression outputs	No squashing; direct prediction of real values.

Matrix Shape, Dimensions, and Neural Network Equation

1. Shape of a Matrix

In NumPy (and mathematics generally): `.shape` $\rightarrow$ gives the size of each dimension as a tuple.

Listing 1: Matrix shape example

```
A = [[1,2,3],
     [4,5,6]]
A.shape # (2,3)    2 rows    3 columns
```

`shape[0]` $\rightarrow$ number of rows. `shape[1]` $\rightarrow$ number of columns.

2. Dimensions (Rank of Array)

- Scalar: no dimension → ()
- Vector: 1D → (n,)
- Matrix: 2D → (m,n)
- Tensor: higher-D → e.g., (batch, height, width, channels)

3. Matrix Operations

Matrix Operations Summary

Addition/Subtraction: Two matrices can be added/subtracted only if shapes match. Example: (m,n) + (m,n) → (m,n)

Multiplication:

1. Elementwise (Hadamard): (m,n)*(m,n) → (m,n)
2. Matrix multiplication: If A is (m,n) and B is (n,p), then C=A · B → (m,p). Rule: Inner dimensions must match.

4. Neural Network Equation

Forward pass: $z = W \cdot x + b$

$$\begin{aligned} x &\to (n,1) && \text{input vector (n features)} \\ W &\to (m,n) && \text{weight matrix (m neurons, n inputs)} \\ b &\to (m,1) && \text{bias vector} \\ z &\to (m,1) && \text{output} \end{aligned}$$

$$W(m,n) \times x(n,1) \to (m,1), \quad +b(m,1) \to (m,1)$$

Example

Input has 4 features → $x.shape = (4,1)$ Layer has 3 neurons → $z.shape = (3,1)$ Then:

$$W.shape = (3,4), \quad b.shape = (3,1), \quad z = W \cdot x + b \to (3,1)$$

Symbol	Shape	Meaning
x	(n,1)	Input features
W	(m,n)	Weights
b	(m,1)	Bias
z	(m,1)	Linear output

NumPy `.shape` Attribute – Notes with Examples

1. Overview

Every ndarray object in NumPy has a `.shape` attribute that returns a tuple showing size along each axis.

2. 1D Arrays (Vectors)

```
import numpy as np
a = np.array([10, 20, 30, 40, 50])
print(a.shape)    # (5,)
print(a.ndim)     # 1
print(a.shape[0]) # 5
```

Explanation: 1D array with 5 elements → `.shape = (5,)`

3. 2D Arrays (Matrices)

```
b = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9],
              [10, 11, 12]])
print(b.shape) # (4, 3)
```

Explanation: 4 rows, 3 columns → `(4,3)`

4. 3D Arrays (Tensors)

```
c = np.array([
 [[1, 2], [3, 4], [5, 6]],
 [[7, 8], [9, 10], [11, 12]]
])
```

$$c.shape = (2, 3, 2), \quad c.ndim = 3$$

5. 4D Arrays (e.g., Image Batches)

```
d = np.random.rand(10, 64, 64, 3)
print(d.shape) # (10, 64, 64, 3)
```

Explanation: 10 images, 64×64, 3 RGB channels.

6. Practical Uses

```python
# Iteration by rows
rows, cols = b.shape
for i in range(rows):
    print("Row:", b[i])

# Reshaping arrays
arr = np.arange(12)
reshaped = arr.reshape(3, 4)
print(reshaped.shape)   # (3, 4)
```

7. Quick Reference Table

Array Type	Example Shape	Meaning
1D vector	(5,)	5 elements
2D matrix	(4, 3)	4 rows, 3 columns
3D tensor	(2, 3, 2)	2 blocks, each 3×2
4D tensor	(10, 64, 64, 3)	10 images, 64×64 pixels, 3 channels