MLP and Backpropagation Algorithm with Activation Functions

A Beginner's Guide to Neural Network Fundamentals

Introduction to Multi-Layer Perceptrons (MLPs)

Imagine you're trying to teach a computer to recognize handwritten digits, like the numbers 0 through 9. A single neuron (like a basic perceptron) can only draw straight lines to separate data – it's like trying to separate complex shapes using only a ruler. But what if you need to recognize curved letters or complex patterns? This is where Multi-Layer Perceptrons (MLPs) come to the rescue.

A Multi-Layer Perceptron is essentially a neural network with multiple layers of interconnected neurons. Think of it as a team of decision-makers, where each layer processes information and passes it to the next layer, gradually building up more sophisticated understanding of the input data [1] [2] [3].

Key Components of an MLP:

- **Input Layer**: Receives the raw data (like pixel values from an image)
- **Hidden Layer(s)**: Process and transform the data through weighted connections
- **Output Layer**: Produces the final prediction or classification

Unlike a simple perceptron that can only solve linearly separable problems (think of separating red and blue dots with a straight line), MLPs can solve complex, non-linear problems by stacking multiple layers together [4].

Why MLPs Matter

The magic happens because of two crucial elements:

1. **Multiple layers** that allow the network to learn hierarchical features
2. **Non-linear activation functions** that enable the network to model complex relationships

Without activation functions, even a multi-layer network would behave like a single linear model – no matter how many layers you stack, it would still only draw straight lines [5].

The Backpropagation Algorithm

Understanding the Learning Process

Training an MLP involves a two-step dance called **forward propagation** and **backward propagation** (backpropagation). Think of it like learning to play darts:

1. **Forward Pass**: You throw the dart (input data flows through the network)
2. **Check Result**: You see where it landed compared to the bullseye (calculate the error)
3. **Backward Pass**: You adjust your technique based on the error (update weights using backpropagation)

Forward Propagation

In forward propagation, data flows from input to output layer. At each neuron, the following calculation occurs:

For a neuron in any layer:
$z = \sum_i w_i x_i + b$

Where:

- z is the weighted sum
- w_i are the weights
- x_i are the inputs
- b is the bias term

Then an activation function is applied:
$a = f(z)$

This output a becomes the input for the next layer [1] [6].

Backward Propagation (Backpropagation)

Here's where the real learning happens. Backpropagation uses the **chain rule** from calculus to efficiently compute how much each weight contributed to the final error, working backwards from the output layer to the input layer [1] [6].

Why Backpropagation is Revolutionary:

- **Memory Efficient**: Uses less memory compared to other optimization methods
- **Fast**: Especially for small to medium networks
- **Generic**: Works with different network architectures
- **No Extra Parameters**: The algorithm itself has no parameters to tune [6]

The algorithm calculates gradients (slopes) that tell each weight how to adjust to reduce the error. It's like having a GPS for weight updates – it points each weight in the direction that will improve the network's performance.

Mathematical Foundation

The error at the output layer is typically calculated using Mean Squared Error:
$\text{MSE} = (\text{Predicted Output} - \text{Actual Output})^2$

The backpropagation algorithm then uses the chain rule to compute gradients:
$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$

Where $\frac{\partial a}{\partial z}$ is the derivative of the activation function – this is why activation functions need to be differentiable [1] [6].

Activation Functions: The Decision Makers

Activation functions are the "decision makers" in each neuron. They determine whether a neuron should be "activated" (fired) based on its input. Think of them as filters that transform the neuron's input into an output suitable for the next layer [5] [7].

Why Activation Functions Are Essential:

- **Non-linearity**: They enable the network to model complex, non-linear relationships
- **Feature Transformation**: They help map inputs to desired output ranges
- **Gradient Flow**: They affect how well gradients flow during backpropagation

Sigmoid Activation Function

The Concept

The sigmoid function is like a smooth switch that gradually transitions from "off" (0) to "on" (1). Imagine a dimmer switch for lights – instead of an abrupt on/off, it provides a smooth transition.

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

The sigmoid function maps any real number to a value between 0 and 1, making it naturally interpretable as a probability [5] [8].

Example

Let's see how sigmoid works with different inputs:

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Test various inputs
inputs = [-5, -1, 0, 1, 5]
outputs = [sigmoid(x) for x in inputs]

print("Input -&gt; Sigmoid Output")
for i, o in zip(inputs, outputs):
    print(f"{i:2d} -&gt; {o:.4f}")
```

Output:

```
Input -&gt; Sigmoid Output
-5 -&gt; 0.0067
-1 -&gt; 0.2689
 0 -&gt; 0.5000
 1 -&gt; 0.7311
 5 -&gt; 0.9933
```

What This Shows

Notice how sigmoid "squashes" all inputs into the range (0,1). Extreme negative values approach 0, extreme positive values approach 1, and 0 maps to exactly 0.5. This S-shaped curve provides smooth transitions, making it excellent for binary classification problems [8].

Pros and Cons

✅ Pros:

- **Smooth gradient**: Enables stable learning
- **Clear predictions**: Perfect for binary classification (0 or 1)
- **Bounded output**: Values stay between 0 and 1
- **Interpretable**: Output represents probability [9]

❌ Cons:

- **Vanishing gradient problem**: Gradients become very small for extreme values, slowing learning
- **Not zero-centered**: All outputs are positive, which can slow convergence
- **Computationally expensive**: Involves exponential calculations [8] [10]

Best Use Cases:

- Output layer for binary classification
- When you need probability-like outputs
- Shallow networks (not recommended for deep networks due to vanishing gradients)

Worst Use Cases:

- Hidden layers in deep networks
- When you need zero-centered outputs
- Real-time applications requiring fast computation [8]

Tanh (Hyperbolic Tangent) Activation Function

The Concept

Tanh is like sigmoid's upgraded cousin. While sigmoid outputs values between 0 and 1, tanh outputs values between -1 and 1. It's like having a balance scale that can tip both ways, rather than just measuring weight.

Mathematical Definition:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1+e^{-2x}} - 1$$

Example

```
import numpy as np

def tanh(x):
    return np.tanh(x)

# Test various inputs
inputs = [-3, -1, 0, 1, 3]
outputs = [tanh(x) for x in inputs]
```

```
print("Input -&gt; Tanh Output")
for i, o in zip(inputs, outputs):
    print(f"{i:2d} -&gt; {o:.4f}")
```

Output:

```
Input -&gt; Tanh Output
-3 -&gt; -0.9951
-1 -&gt; -0.7616
 0 -&gt;  0.0000
 1 -&gt;  0.7616
 3 -&gt;  0.9951
```

What This Shows

Tanh produces a symmetric S-curve centered at zero. This zero-centered property is crucial – it means the function can produce both positive and negative outputs, leading to better gradient flow during training [10].

Pros and Cons

✅ **Pros:**

- **Zero-centered**: Faster convergence compared to sigmoid
- **Stronger gradients**: Four times stronger than sigmoid gradients
- **Symmetric**: Balanced positive and negative outputs
- **Smooth**: Continuous and differentiable everywhere [8] [10]

❌ **Cons:**

- **Vanishing gradient problem**: Still suffers from this issue for extreme values
- **Computational cost**: More expensive than ReLU
- **Saturation**: Gradients approach zero for large inputs [8] [10]

Best Use Cases:

- Hidden layers in shallow to medium networks
- When you need zero-centered outputs
- Classification problems where you want symmetric activation [11]

Worst Use Cases:

- Very deep networks (due to vanishing gradients)
- When computational speed is critical
- Output layers requiring probability interpretation [8]

ReLU (Rectified Linear Unit) Activation Function

The Concept

ReLU is the superstar of modern deep learning. Think of it as a simple "on/off" switch with a twist – it lets positive values pass through unchanged but blocks all negative values (setting them to zero). It's like having a one-way valve that only allows positive flow.

Mathematical Definition:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & amp; \text{if } x \quad gt; 0 \\ 0 & amp; \text{if } x \leq 0 \end{cases}$$

Example

```
import numpy as np

def relu(x):
    return np.maximum(0, x)

# Test various inputs
inputs = [-3, -1, 0, 1, 3, 10]
outputs = [relu(x) for x in inputs]

print("Input -&gt; ReLU Output")
for i, o in zip(inputs, outputs):
    print(f"{i:2d} -&gt; {o:.4f}")
```

Output:

```
Input -&gt; ReLU Output
-3 -&gt;  0.0000
-1 -&gt;  0.0000
 0 -&gt;  0.0000
 1 -&gt;  1.0000
 3 -&gt;  3.0000
10 -&gt; 10.0000
```

What This Shows

ReLU's behavior is beautifully simple: negative inputs become zero, positive inputs remain unchanged. This simplicity is its strength – it's fast to compute and helps create sparse representations (many zeros) in the network [12] [13].

Pros and Cons

✅ **Pros:**

- **Computationally efficient**: Extremely fast – just a comparison and selection
- **No vanishing gradient**: For positive inputs, gradient is always 1
- **Sparse activation**: Creates sparse networks, improving efficiency

- **Unbounded output**: Can represent any positive value
- **Less sensitive to initialization**: More robust to random weight initialization
- **Default choice**: Works well as a general-purpose activation function [12] [13]

✖ Cons:

- **Dying ReLU problem**: Neurons can "die" (always output 0) if they consistently receive negative inputs
- **Not zero-centered**: All outputs are non-negative
- **Gradient discontinuity**: Gradient is undefined at x=0 [14] [13]

Best Use Cases:

- Hidden layers in deep neural networks
- Convolutional neural networks
- When training speed is important
- General-purpose activation when unsure what to choose [13] [15]

Worst Use Cases:

- Output layers for regression (unless you want only positive outputs)
- When you need symmetric outputs
- Networks prone to many negative inputs (leading to dead neurons) [13]

Leaky ReLU Activation Function

The Concept

Leaky ReLU is ReLU's "fixed" version. While ReLU completely blocks negative values, Leaky ReLU allows a small "leak" – it lets a tiny fraction of negative values pass through. Think of it as a valve that's almost closed but still allows a small trickle for negative values.

Mathematical Definition:

$$\text{Leaky ReLU}(x) = \begin{cases} x & amp; \text{if } x \quad gt; 0 \\ \alpha x & amp; \text{if } x \leq 0 \end{cases}$$

Where α is a small positive constant, typically 0.01.

Example

```
import numpy as np

def leaky_relu(x, alpha=0.01):
    return np.where(x &gt; 0, x, alpha * x)

# Test various inputs
inputs = [-3, -1, 0, 1, 3, 10]
outputs = [leaky_relu(x) for x in inputs]
```

```
print("Input -&gt; Leaky ReLU Output (α=0.01)")
for i, o in zip(inputs, outputs):
    print(f"{i:2d} -&gt; {o:.4f}")
```

Output:

```
Input -&gt; Leaky ReLU Output (α=0.01)
-3 -&gt; -0.0300
-1 -&gt; -0.0100
 0 -&gt;  0.0000
 1 -&gt;  1.0000
 3 -&gt;  3.0000
10 -&gt; 10.0000
```

What This Shows

Leaky ReLU preserves some information from negative inputs (multiplied by α), while keeping positive inputs unchanged. This small "leak" prevents neurons from completely dying [14] [16].

Pros and Cons

✅ **Pros:**

- **Prevents dying neurons**: The small gradient for negative inputs keeps neurons "alive"
- **Computationally efficient**: Still very fast to compute
- **Better gradient flow**: Gradients can flow even for negative inputs
- **Reduced dead neuron problem**: Addresses ReLU's main weakness [14] [16]

❌ **Cons:**

- **Extra hyperparameter**: Need to choose α (though 0.01 usually works well)
- **Slightly more computation**: Marginally more expensive than ReLU
- **Still not zero-centered**: Doesn't solve the zero-centering issue [14]

Best Use Cases:

- Deep networks where dying ReLU is a problem
- When you want ReLU's benefits but need to avoid dead neurons
- Hidden layers in networks with aggressive regularization
- As a safer alternative to ReLU [16]

Worst Use Cases:

- When simplicity is paramount (ReLU might be better)
- Output layers for probability estimation
- When the network already works well with ReLU [14]

Comparative Analysis: Choosing the Right Activation Function

Summary Comparison

Function	Range	Zero-Centered	Vanishing Gradient	Computational Cost	Main Strength	Main Weakness
Sigmoid	(0, 1)	✗ No	✓ Yes	High	Probability output	Vanishing gradients
Tanh	(-1, 1)	✓ Yes	✓ Yes	Medium	Zero-centered	Vanishing gradients
ReLU	[0, ∞)	✗ No	✗ No	Low	Fast, no vanishing	Dying neurons
Leaky ReLU	(-∞, ∞)	✗ No	✗ No	Low	Prevents dying	Extra parameter

Decision Framework

For Output Layers:

- **Binary Classification**: Sigmoid (probability interpretation)
- **Multi-class Classification**: Softmax (not covered here)
- **Regression**: Linear activation or ReLU (if only positive outputs expected)

For Hidden Layers:

- **Default Choice**: ReLU (fastest, works well generally)
- **Deep Networks**: Leaky ReLU (prevents dying neurons)
- **When Zero-Centering Matters**: Tanh
- **Legacy/Shallow Networks**: Sigmoid or Tanh [15]

Real-World Applications

Computer Vision (CNNs): ReLU dominates due to its speed and effectiveness with image data, where sparsity (many zeros) is beneficial [12].

Natural Language Processing: Tanh was historically popular, but ReLU variants are increasingly used in modern architectures.

Traditional Feedforward Networks: The choice depends on depth – ReLU for deep networks, Tanh for shallow to medium networks [11].

Key Takeaways and Common Pitfalls

Essential Insights

1. **Activation functions are crucial**: They provide the non-linearity that makes neural networks powerful
2. **Different layers, different functions**: Output layers need different activation functions than hidden layers
3. **Modern default**: ReLU has become the go-to choice for hidden layers in deep networks
4. **Context matters**: The best activation function depends on your specific problem and architecture

Common Pitfalls to Avoid

✖ **Using sigmoid in deep networks**: Leads to vanishing gradients and slow training
✖ **Forgetting to match output activation to problem type**: Using ReLU for binary classification output
✖ **Not considering computational cost**: Sigmoid/Tanh are slower than ReLU
✖ **Ignoring dying ReLU**: In very deep networks, consider Leaky ReLU
✖ **One-size-fits-all mentality**: Different problems may benefit from different activation functions

Best Practices

✔ **Start with ReLU for hidden layers**: It's a safe, fast default choice
✔ **Match output activation to your problem**: Sigmoid for binary, softmax for multi-class
✔ **Monitor for dead neurons**: If many ReLU neurons die, switch to Leaky ReLU
✔ **Experiment when performance matters**: Try different activations if default choices don't work well
✔ **Consider the full pipeline**: Activation choice affects initialization, learning rate, and other hyperparameters

Conclusion

Multi-Layer Perceptrons with backpropagation represent one of the foundational breakthroughs in artificial intelligence. The combination of multiple layers and non-linear activation functions enables these networks to learn complex patterns that simpler models cannot capture.

The Big Picture:

- **MLPs** provide the architecture for complex learning
- **Backpropagation** provides the efficient learning algorithm
- **Activation functions** provide the non-linear power

Understanding these fundamentals is crucial for anyone working with neural networks. While modern deep learning has introduced many sophisticated architectures, the principles of MLPs and backpropagation remain at the heart of most neural network training.

Remember: the best activation function for your problem might not be the most popular one. Understanding the trade-offs allows you to make informed decisions and debug issues when they arise. Start with the defaults (ReLU for hidden layers), but don't be afraid to experiment when the situation calls for it.

The journey into neural networks begins with these fundamentals, but the applications are limitless – from image recognition to natural language processing, from game playing to scientific discovery. Master these basics, and you'll have a solid foundation for exploring the exciting world of deep learning.

This document provides a comprehensive introduction to MLPs and activation functions. For practical implementation, combine this theoretical knowledge with hands-on coding experience using frameworks like TensorFlow, PyTorch, or Keras.
[17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37]

⁂

1. https://www.geeksforgeeks.org/machine-learning/backpropagation-in-neural-network/
2. https://milvus.io/ai-quick-reference/what-is-a-multilayer-perceptron-mlp
3. https://www.geeksforgeeks.org/deep-learning/multi-layer-perceptron-learning-in-tensorflow/
4. https://towardsai.net/p/artificial-intelligence/design-a-multi-layer-perceptron-mlp-neural-network-for-classification
5. https://www.v7labs.com/blog/neural-networks-activation-functions
6. https://neptune.ai/blog/backpropagation-algorithm-in-neural-networks-guide
7. https://www.superannotate.com/blog/activation-functions-in-neural-networks
8. https://www.geeksforgeeks.org/deep-learning/tanh-vs-sigmoid-vs-relu/
9. https://www.linkedin.com/pulse/top-10-activation-functions-advantages-disadvantages-dash
10. https://www.baeldung.com/cs/sigmoid-vs-tanh-functions
11. https://pureportal.strath.ac.uk/files/118946797/Nwankpa_etal_ICCST_2021_Activation_functions_comparison_of_trends_in_practice.pdf
12. https://arxiv.org/pdf/2010.09458.pdf
13. https://1cademy.com/node/pros-and-cons-of-relu/y3jLboZbwdmadKadNv8t
14. https://stackoverflow.com/questions/56287870/what-are-the-disadvantages-of-leaky-relu
15. https://www.machinelearningmastery.com/choose-an-activation-function-for-deep-learning/
16. https://www.meegle.com/en_us/topics/neural-networks/activation-functions-in-neural-networks
17. https://opencourse.inf.ed.ac.uk/sites/default/files/https/opencourse.inf.ed.ac.uk/inf1-cg/2025/inf1cgl06mlpbackprop_1.pdf
18. https://aiml.com/what-is-an-activation-function-what-are-the-different-types-of-activation-functions-discuss-their-pros-and-cons/
19. https://www.machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/
20. https://www.nature.com/articles/s41467-024-53827-9
21. https://www.reddit.com/r/learnmachinelearning/comments/ua6n6s/why_is_relu_considered_superior_compared_to_tanh/
22. https://www.mathworks.com/matlabcentral/fileexchange/63106-multilayer-neural-network-using-backpropagation-algorithm?s_tid=FX_rc3_behav
23. https://pabloinsente.github.io/the-multilayer-perceptron
24. https://apxml.com/courses/getting-started-with-pytorch/chapter-4-building-models-torch-nn/activation-functions
25. https://amanxai.com/2025/01/02/a-guide-to-activation-functions-in-neural-networks/
26. https://en.wikipedia.org/wiki/Multilayer_perceptron
27. https://www.tredence.com/blog/neural-networks-guide-basic-to-advance
28. https://moldstud.com/articles/p-key-considerations-for-integrating-activation-functions-in-neural-networks-a-guide-for-developers

29. https://www.sciencedirect.com/topics/computer-science/multilayer-perceptron
30. https://www.linkedin.com/pulse/best-practices-activation-functions-dropout-or-shulrufer-g7gnf
31. https://d2l.ai/chapter_multilayer-perceptrons/mlp.html
32. https://semiengineering.com/implementing-ai-activation-functions/
33. https://scikit-learn.org/stable/modules/neural_networks_supervised.html
34. https://www.reddit.com/r/MachineLearning/comments/1arovn8/r_three_decades_of_activations_a_comprehensive/
35. https://python.plainenglish.io/activation-functions-in-neural-networks-the-gatekeepers-of-learning-ac7728d1ff75
36. https://www.mldawn.com/backpropagation-algorithm-part1-mlp-sigmoid/
37. https://www.aitude.com/comparison-of-sigmoid-tanh-and-relu-activation-functions/