GROUP 2 REPORT

GitHub Link

Team Members:

Yaseen Mneimneih
Wael Khafagi
Sujit Bhatta
Ivan Radavskyi
MD Iftier Roshid

Contents

Dataset Selection & Description

Dataset selected

We have decided to go forward with the CIFAR-10 dataset.

Motivations for dataset

While simpler datasets like MNIST are available, they often fail to demonstrate a model's robustness in handling the nuances of real-world visual data.

1. Dimensionality and Colour Complexity Unlike simpler sets like MNIST, which primarily consist of grayscale images (often 28x28x1), CIFAR-10 consists of 60,000 colour images. Each 32x32 pixel image contains three colour channels (RGB), which inherently increases the input dimensionality and requires the neural network to learn to process and integrate colour information alongside spatial features.
2. Feature Diversity and Semantic Depth The complexity of CIFAR-10 lies in its 10 distinct and diverse classes, which are evenly distributed with 6,000 images per class. The dataset includes a wide variety of subjects, categorised into:
   - Animals: Bird, Cat, Deer, Dog, Frog, Horse.
   - Vehicles: Airplane, Automobile, Ship, Truck.

Classifying these requires the model to identify complex low-level, mid-level, and high-level features such as textures, distinct shapes, and varying backgrounds rather than just simple strokes or outlines. This variety forces the network to develop more generalisable feature representations.

Implement Sigmoid and ReLU Layers

Sigmoid Layer

The Sigmoid function, defined as $\sigma(x) = \frac{1}{1+e^{-x}}$, maps input values to a range between 0 and 1 (Appendix I).

The Sigmoid activation function was also implemented with forward and backward passes. In the forward pass, a numerically stable version of the Sigmoid function was used to avoid overflow issues when dealing with large positive or negative input values. The output of the activation function is cached for use in the backward pass.

During backpropagation, the cached output values are used to compute the gradients. This avoids recomputing the Sigmoid function and keeps the backward pass efficient and consistent with the forward computation.

Pros:

- Probabilistic Interpretation: Because its output is bounded between 0 and 1, it is ideal for models requiring a probability-based outcome, such as binary classification.
- Smooth Gradient: It provides a smooth, differentiable threshold that is useful for gradient-based optimization.

Cons:

- Vanishing Gradients: Saturated neurons "kill gradients". When inputs are very large or very small, the function becomes flat, causing the gradient to approach zero and halting the learning process.
- Not Zero-Centred: The outputs are not zero-centred, which can make the optimization process less efficient during gradient descent.
- Computational Expense: Calculating the function is computationally expensive compared to simpler alternatives.

ReLU Layer

ReLU is defined as $y = max(0, x)$, meaning it outputs the input directly if it is positive and zero otherwise (Appendix J).

The ReLU activation layer was implemented with both forward and backward passes. In the forward pass, the input values are passed through unchanged if they are positive, while negative values are set to zero. The input is stored during the forward pass so it can be reused later during backpropagation.

For the backward pass, gradients are only passed through neurons that were active in the forward pass. This is done by checking which input values were positive and masking the gradients accordingly. This approach avoids updating neurons that did not contribute to the output which makes it more efficient.

Pros:

- Computational Efficiency: ReLU is very efficient to calculate as it only involves a simple comparison at zero.
- Faster Convergence: Models using ReLU tend to converge much faster than those using Sigmoid or Tanh.
- Reduced Saturation: Unlike Sigmoid, ReLU does not saturate in the positive region, which helps maintain a steady gradient flow for positive inputs.

Cons:

- "Dying ReLU" Problem: If a neuron receives negative inputs, its gradient becomes zero. If this happens consistently, the neuron can "die" and remain permanently inactive, as it no longer contributes to the learning process.
- Not Zero-Centred: Like the Sigmoid function, its output is not zero-centred.

Implement SoftMax Layer

SoftMax Layer

The SoftMax function is defined as

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

where x_i represents the i-th element of the input vector. This function converts a vector of arbitrary real values (logits) into a probability distribution, where all output values are between 0 and 1 and sum to exactly 1 (Appendix K).

Implementation Approach

The softmax activation layer is implemented with both forward and backward passes. Additionally, a combined **SoftmaxCrossEntropy** class was created to merge the softmax activation with the cross-entropy loss function for improved numerical stability and computational efficiency.

Forward Pass

In the forward pass, a numerically stable softmax was implemented using the **Log-Sum-Exp (LSE) trick** (Joseph Rivera, 2020), (Appendix A).

Subtracting the maximum prevents overflow when computing exponentials while maintaining mathematical equivalence. The computed probabilities are cached for backpropagation.

Backwards Pass

When combined with Cross-Entropy loss, the gradient simplifies to

$$\frac{\partial L}{\partial x} = y_{\text{pred}} - y_{\text{true}}$$

This elegant simplification occurs because the Jacobian of the softmax function and the gradient of the cross-entropy loss combine algebraically. The gradient calculation is handled directly in the **NeuralNetwork** class's **backward_pass()** method for efficiency.

Numerical Issues and Solutions

1. **Overflow in Exponentials**:
   Computing $e^x$for large $x$causes overflow errors. The log-sum-exp trick ensures all exponentials use non-positive exponents, preventing overflow while maintaining mathematical correctness.
   Error: (Appendix B).
2. **Underflow during loss calculation**
   When computing cross-entropy loss, taking $\log(p)$where $p \approx 0$ can lead to numerical underflow or infinities. The **SoftmaxCrossEntropy** class computes loss directly from logits, which is more stable than computing probabilities first and then taking logarithms with an epsilon
3. **"/m error":** Without realising I was diving by m twice, once initialising and again when computing gradients. This made gradients m times smaller than it should be causing slower learning, slower loss decreases, and lower accuracy. Removing m during initialisation solved this issue, (Appendix C & D).
4. **Combined with Cross-Entropy**: Rather than computing softmax and cross-entropy separately, combining them into a single operation provides better numerical stability and more efficient gradient computation. The combined implementation avoids intermediate probability calculations that could have introduced more errors.

Implement Dropout

Dropout Regularization

Dropout is a regularization technique used to reduce overfitting in neural networks by randomly deactivating a proportion of neurons during training. By preventing the network from relying too heavily on specific neurons, dropout encourages the learning of more generalisable feature representations (Appendix L).

Inverted Dropout Implementation

During training, a binary mask is generated where each neuron is retained with a fixed probability. The activations corresponding to dropped neurons are set to zero, while the remaining activations are scaled appropriately. This scaling ensures that the overall magnitude of activations remains consistent between training and testing.

The use of inverted dropout allows the forward pass at test time to remain unchanged, as no additional scaling is required during inference. This simplifies the evaluation process and avoids discrepancies between training and testing behaviour.

Forward and Backward Passes

During training, dropout masks a subset of activations in the forward pass. During backpropagation, the same mask is reused so that gradients only flow through active neurons. During testing, dropout is disabled and all neurons remain active. This ensures that computations during optimisation remain consistent between the forward and backward passes.

Reasoning

Dropout helps improve generalisation, particularly in deeper networks, but excessive dropout can lead to underfitting. In this coursework, dropout is treated as a tuneable hyperparameter and evaluated alongside other regularization techniques. Its impact on performance is analysed through comparative experiments in later sections.

Implement a Fully Parametrizable Neural Network Class

Implement Optimiser

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent updates the model parameters by moving them in the direction opposite to the gradient of the loss function, scaled by a fixed learning rate. In this implementation, the optimizer iterates over all trainable parameters and applies updates separately to weights and biases. SGD is simple to implement and computationally efficient, but it may converge slowly and can be sensitive to the choice of learning rate.

SGD with Momentum

SGD with Momentum extends standard SGD by incorporating a velocity term that accumulates gradient information from previous updates. This helps accelerate learning in consistent directions and reduces oscillations during training, particularly in regions where gradients vary significantly. In the implementation, a momentum coefficient controls how much of the previous update is retained, allowing the optimiser to smooth parameter updates and improve convergence stability.

Reasoning

Both optimisers were implemented in a modular manner, allowing them to be easily swapped during training. Their performance was compared experimentally by analysing convergence behaviour and final classification accuracy. Results demonstrate that SGD with Momentum generally converges faster and provides more stable training compared to standard SGD.

Evaluate Different Neural Network Architectures/Parameters, Present and Discuss Your Results

A fully connected neural network was implemented from using NumPy. The model is designed to be flexible, allowing the number of hidden layers, the number of units per layer, and the activation functions to be configured when the network is initialised. This makes it easy to experiment with different architectures without changing the underlying implementation.

The network applies activation functions layer by layer during the forward pass and uses cached values during backpropagation. This approach keeps the forward and backward computations consistent and helps avoid unnecessary computation. The use of separate activation classes also makes the code more modular and easier to maintain.

Weight initialisation is selected based on the activation function used in each layer. The normal initialisation is used for layers with ReLU activation, while Xavier initialisation is used for Sigmoid and SoftMax layers. This helps keep activation values within a reasonable range during training and contributes to more stable learning.

Regularisation and dropout are both integrated into the network to help control overfitting. L1 and L2 regularisation are applied during backpropagation by modifying the weight gradients, while dropout is applied only during training and disabled during evaluation. This ensures consistent behaviour during inference.

The training process supports mini-batch learning, learning rate decay, and different optimisation methods. Optimisers are implemented separately and update parameters in place, allowing the training behaviour to be changed without modifying the network itself.

Trying different Parameters

Several hyperparameter configurations were explored to identify a model that achieved high accuracy while minimising overfitting. Four activation functions were evaluated: ReLU, Leaky ReLU, Tanh, and Sigmoid (see Appendix E-H). Among these, ReLU consistently produced the highest accuracy and exhibited the least overfitting. Increasing the number of training epochs beyond a certain point did not improve performance; instead, the model's accuracy stagnated due to overfitting.

Final Parameters

The final model configuration that achieved the best performance is summarised below. The network uses an input size of 3072 and three hidden layers with 512, 256, and 128 units respectively, followed by an output layer of size 10. ReLU activation is applied in all hidden layers, with Softmax used in the output layer. Dropout with a rate of 0.3 is applied to each hidden layer during training. The model is trained using mini-batch gradient descent with a batch size of 128, a learning rate of 0.01, and momentum (β = 0.9). Training was carried out for 25 epochs, resulting in a final classification accuracy of 55.64%.

Overall, the implementation is structured, flexible, and easy to extend. It provides a solid foundation for evaluating different architectures and hyperparameter choices in later experiments.

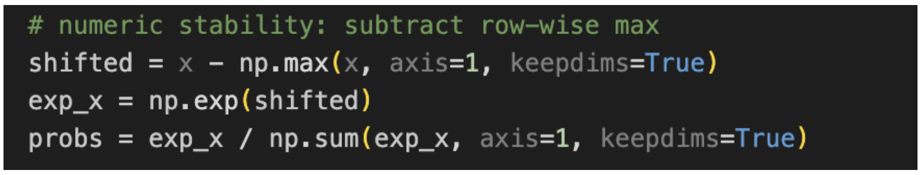
Conclusion

References

- Joseph Rivera, (2020) Stabalizing Overflow and Underflow [Video]. YouTube. Available at: https://www.youtube.com/watch?v=2P470Dg4en8 (Accessed: December 2024).

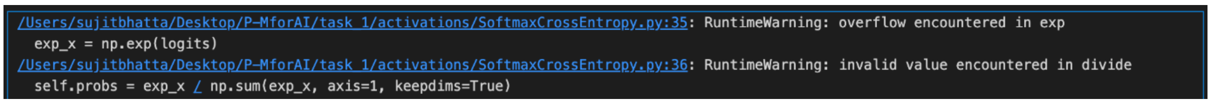
Appendix

A



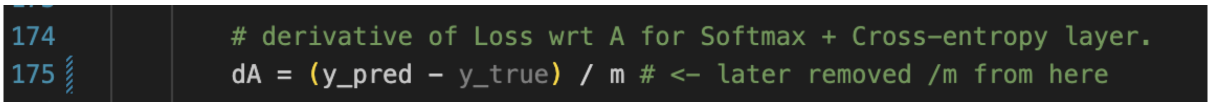
```
# numeric stability: subtract row-wise max
shifted = x - np.max(x, axis=1, keepdims=True)
exp_x = np.exp(shifted)
probs = exp_x / np.sum(exp_x, axis=1, keepdims=True)
```

B



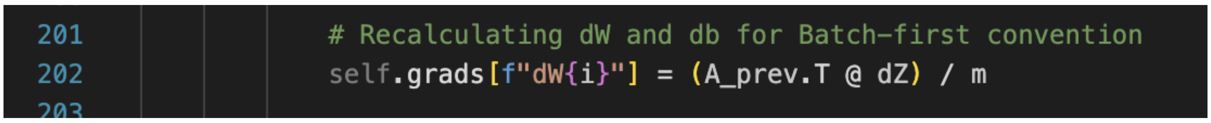
```
/Users/sujitbhatta/Desktop/P-MforAI/task 1/activations/SoftmaxCrossEntropy.py:35: RuntimeWarning: overflow encountered in exp
  exp_x = np.exp(logits)
/Users/sujitbhatta/Desktop/P-MforAI/task 1/activations/SoftmaxCrossEntropy.py:36: RuntimeWarning: invalid value encountered in divide
  self.probs = exp_x / np.sum(exp_x, axis=1, keepdims=True)
```

C



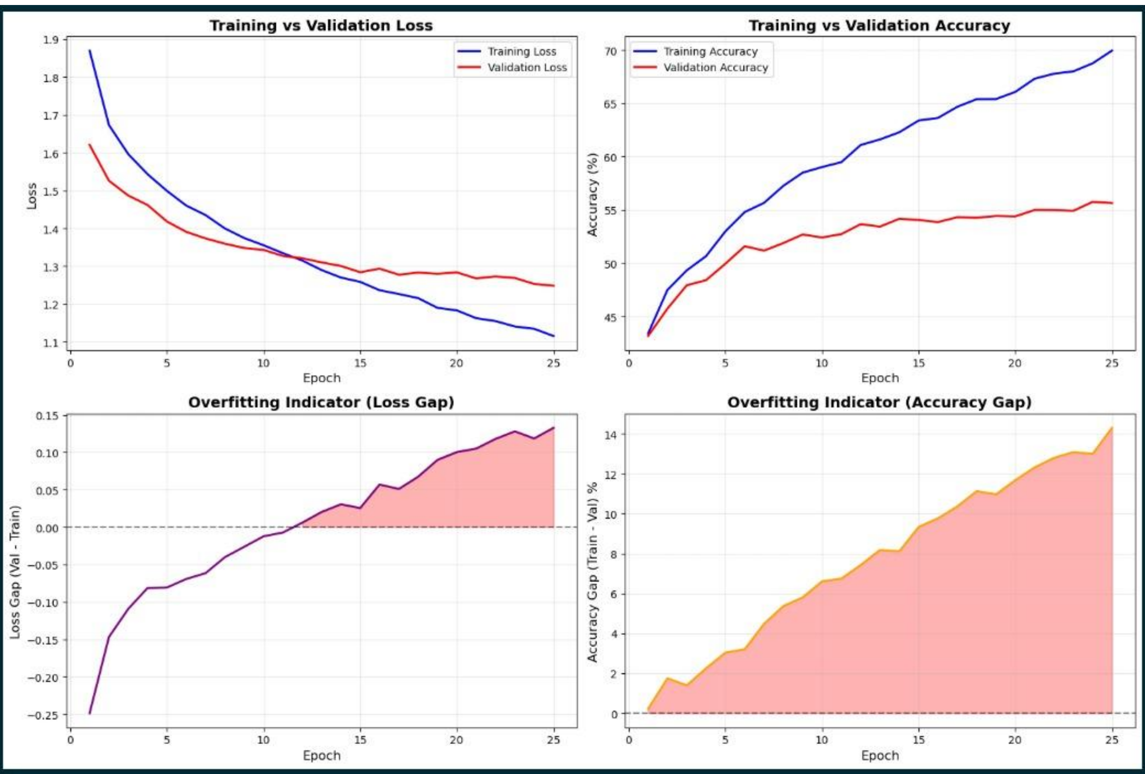
```
        # derivative of Loss wrt A for Softmax + Cross-entropy layer.
        dA = (y_pred - y_true) / m # <- later removed /m from here
```

D



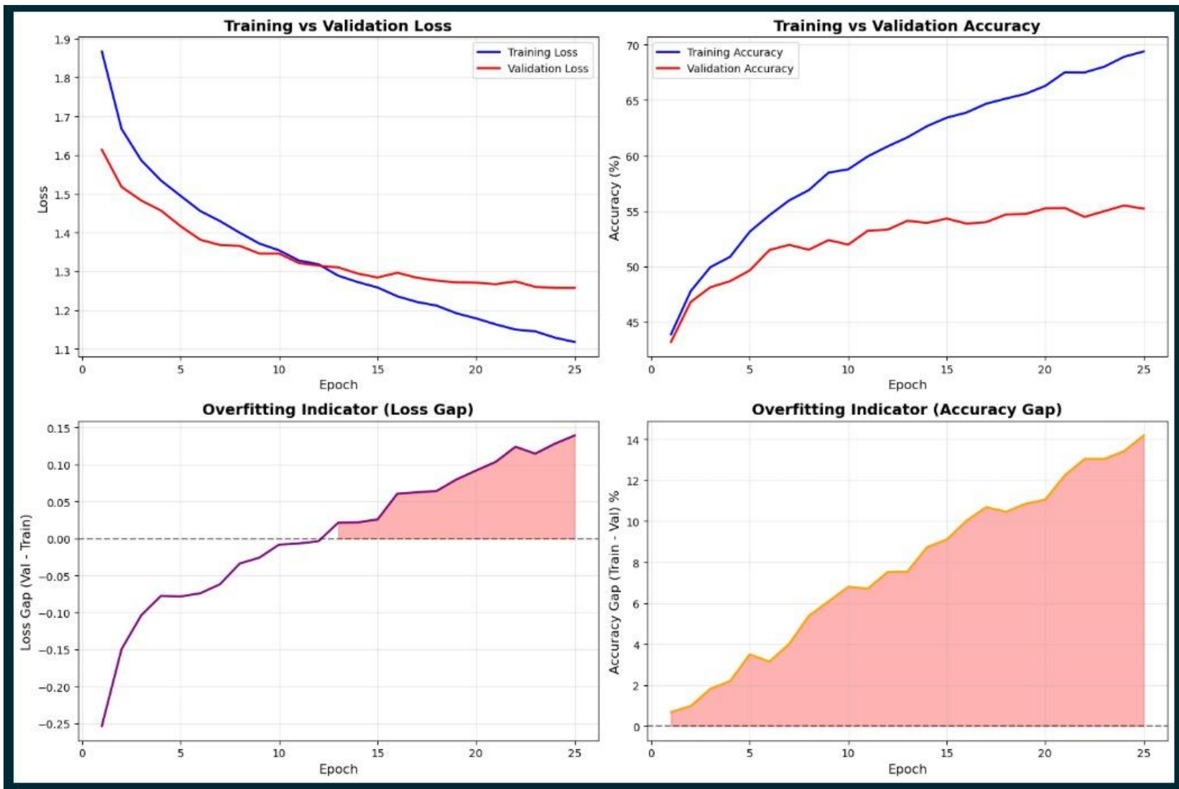
```
            # Recalculating dW and db for Batch-first convention
            self.grads[f"dW{i}"] = (A_prev.T @ dZ) / m
```

E (ReLU)



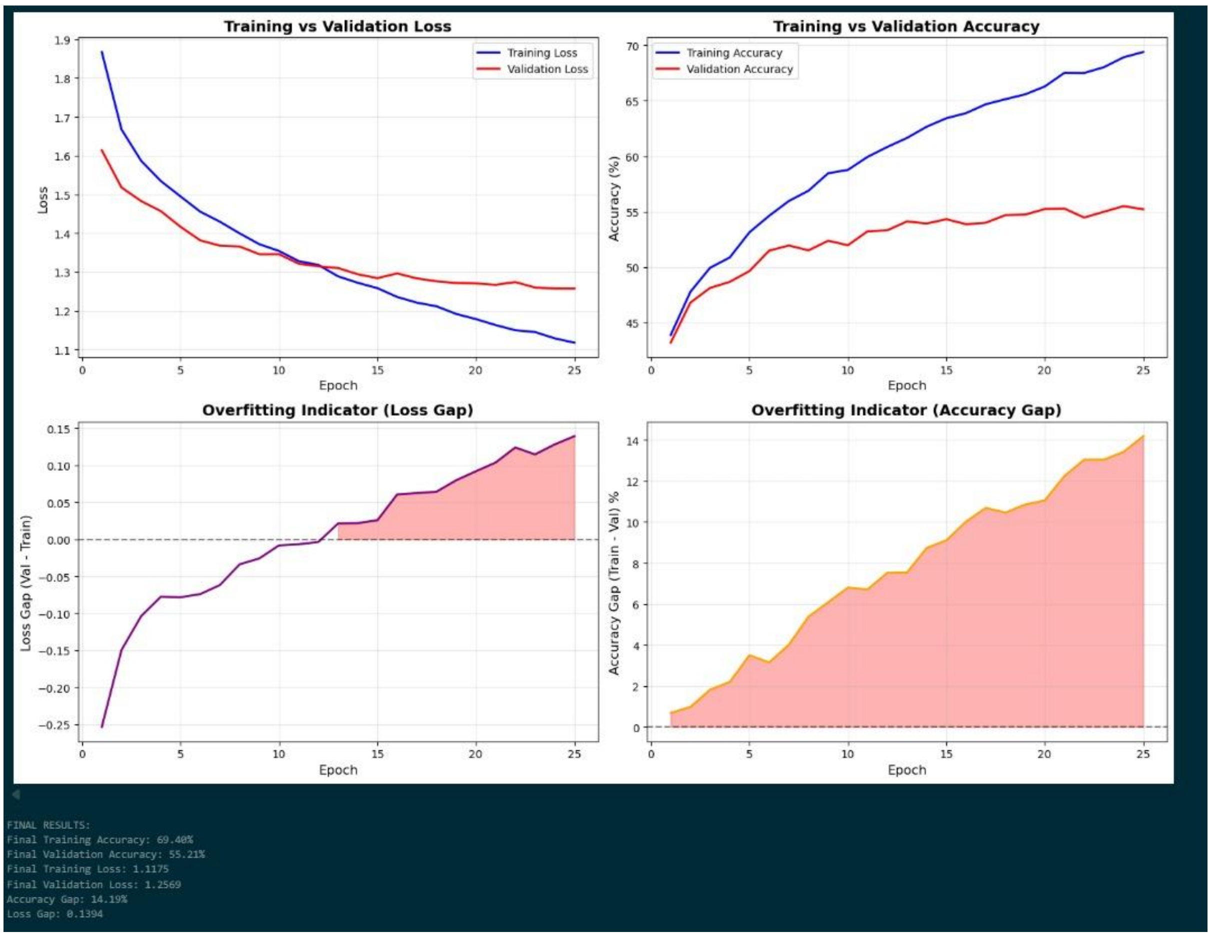
```
FINAL RESULTS:
Final Training Accuracy: 69.93%
Final Validation Accuracy: 55.64%
Final Training Loss: 1.1158
Final Validation Loss: 1.2485
Accuracy Gap: 14.29%
Loss Gap: 0.1327
```

F (Leaky ReLU)



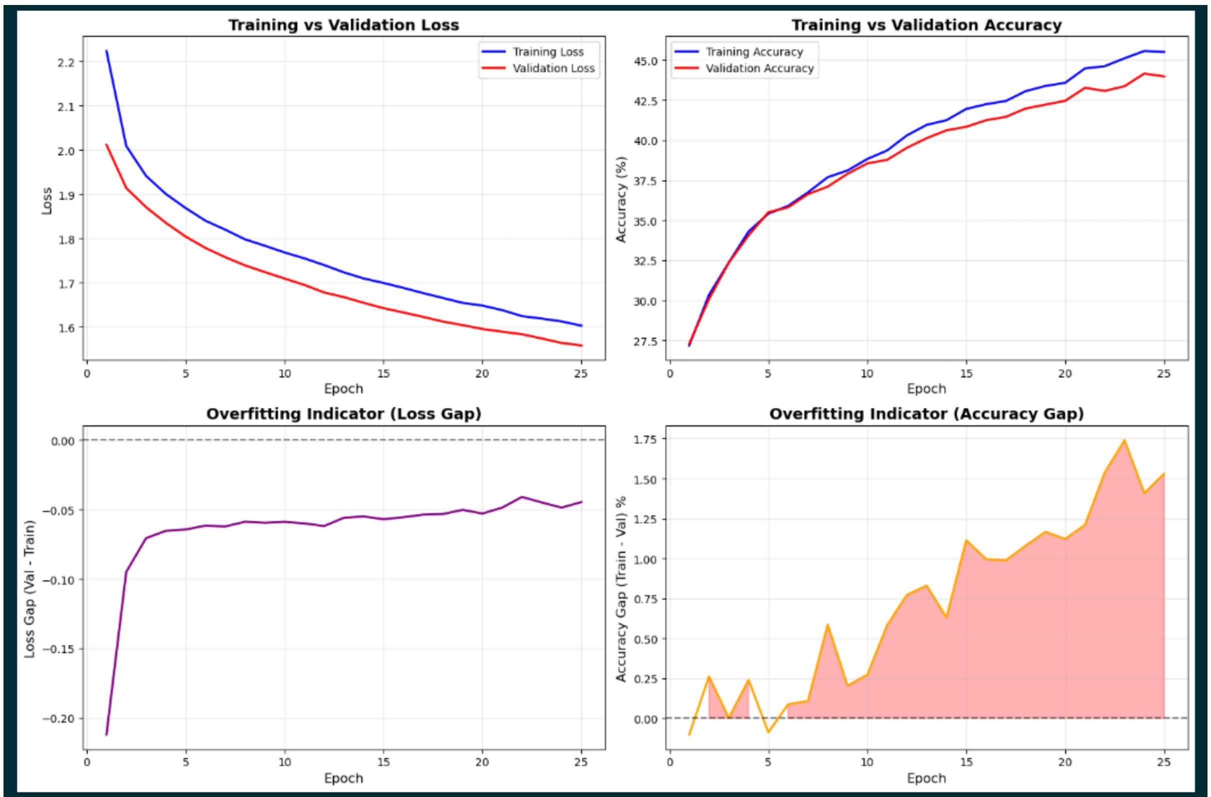
```
FINAL RESULTS:
Final Training Accuracy: 69.40%
Final Validation Accuracy: 55.21%
Final Training Loss: 1.1175
Final Validation Loss: 1.2569
Accuracy Gap: 14.19%
Loss Gap: 0.1394
```

G (Tanh)

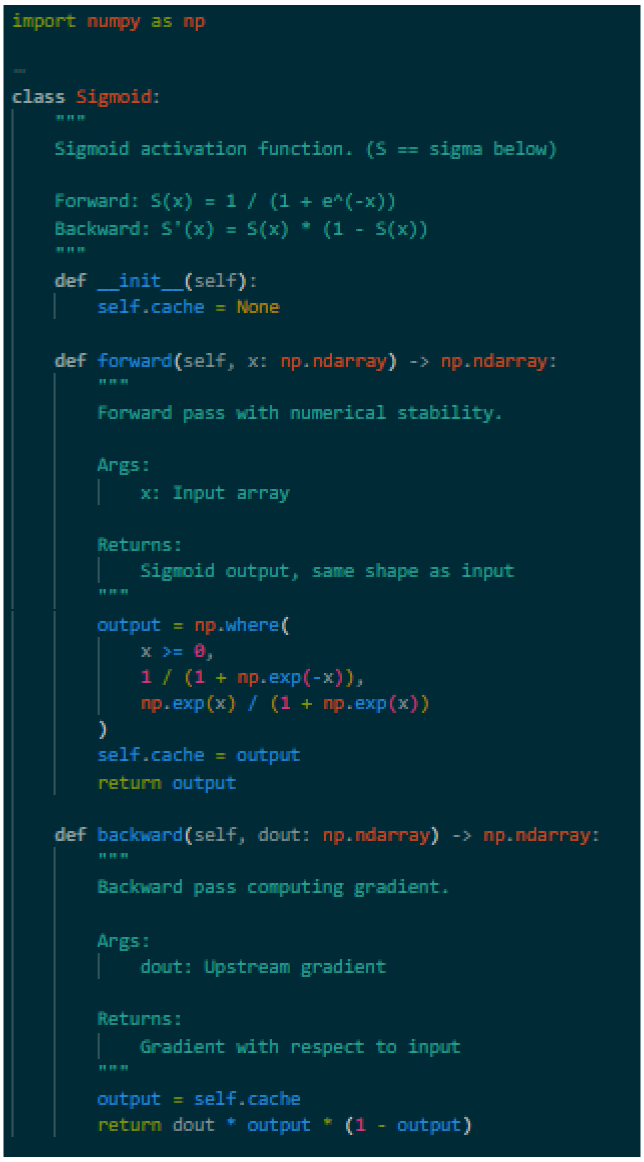


```
FINAL RESULTS:
Final Training Accuracy: 69.40%
Final Validation Accuracy: 55.21%
Final Training Loss: 1.1175
Final Validation Loss: 1.2569
Accuracy Gap: 14.19%
Loss Gap: 0.1394
```

H (Sigmoid)

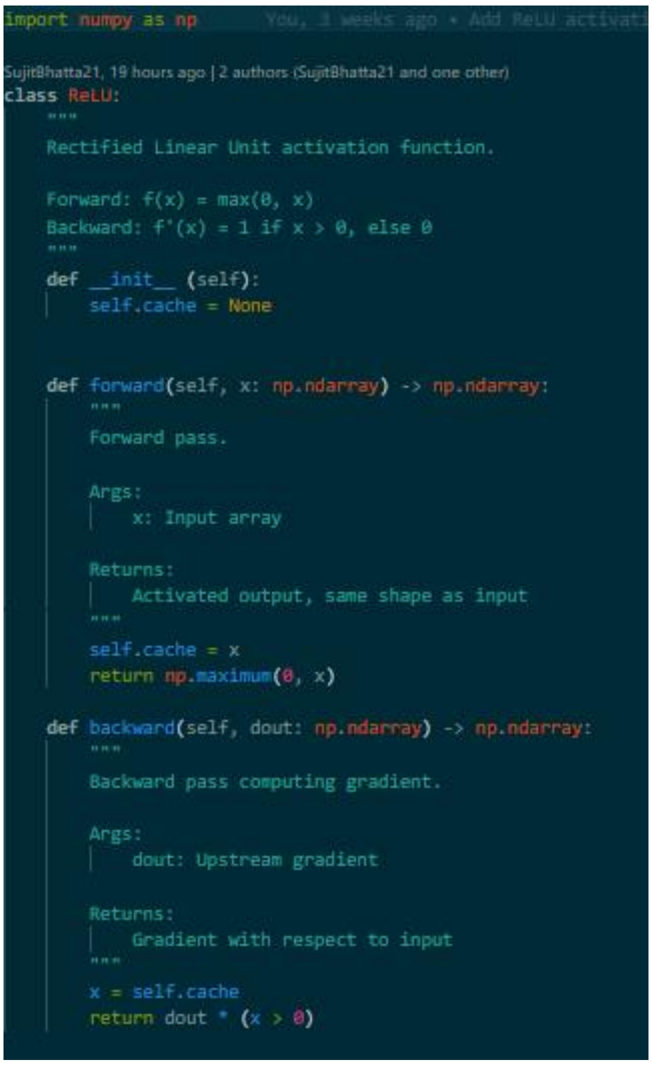


```
FINAL RESULTS:
Final Training Accuracy: 45.50%
Final Validation Accuracy: 43.97%
Final Training Loss: 1.6027
Final Validation Loss: 1.5580
Accuracy Gap: 1.53%
Loss Gap: -0.0446
```



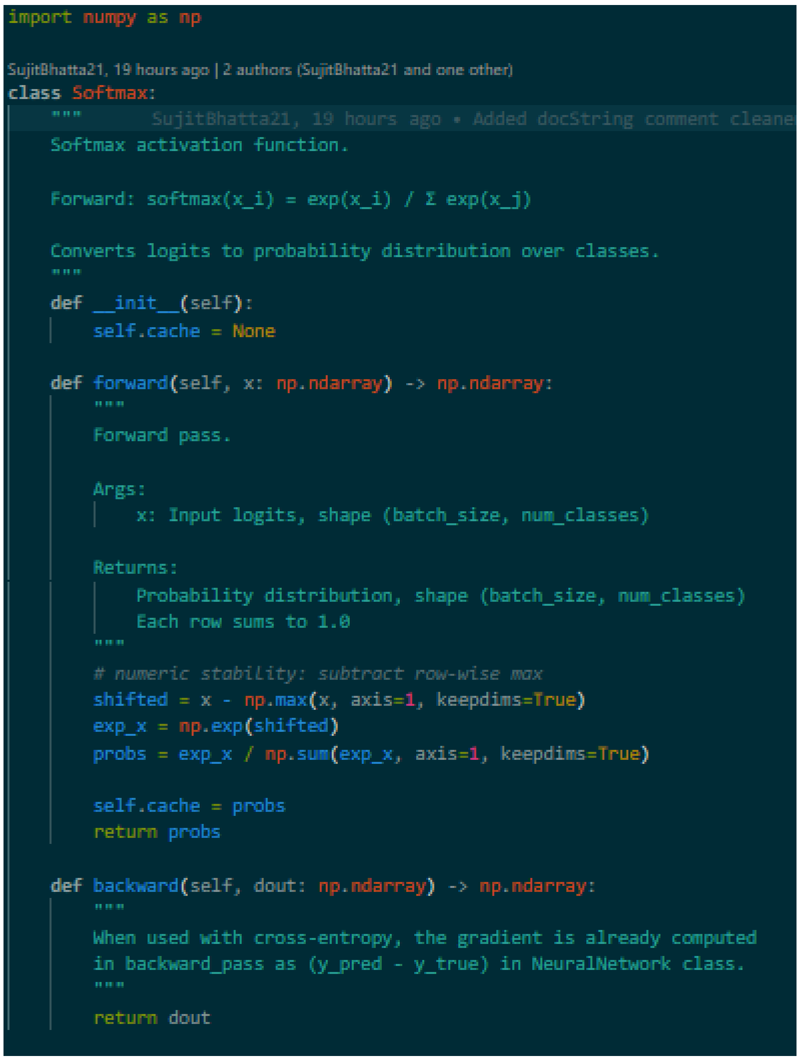
```python
import numpy as np

...

class Sigmoid:
    """
    Sigmoid activation function. (S == sigma below)

    Forward: S(x) = 1 / (1 + e^(-x))
    Backward: S'(x) = S(x) * (1 - S(x))
    """
    def __init__(self):
        self.cache = None

    def forward(self, x: np.ndarray) -> np.ndarray:
        """
        Forward pass with numerical stability.

        Args:
            x: Input array

        Returns:
            Sigmoid output, same shape as input
        """
        output = np.where(
            x >= 0,
            1 / (1 + np.exp(-x)),
            np.exp(x) / (1 + np.exp(x))
        )
        self.cache = output
        return output

    def backward(self, dout: np.ndarray) -> np.ndarray:
        """
        Backward pass computing gradient.

        Args:
            dout: Upstream gradient

        Returns:
            Gradient with respect to input
        """
        output = self.cache
        return dout * output * (1 - output)
```

J



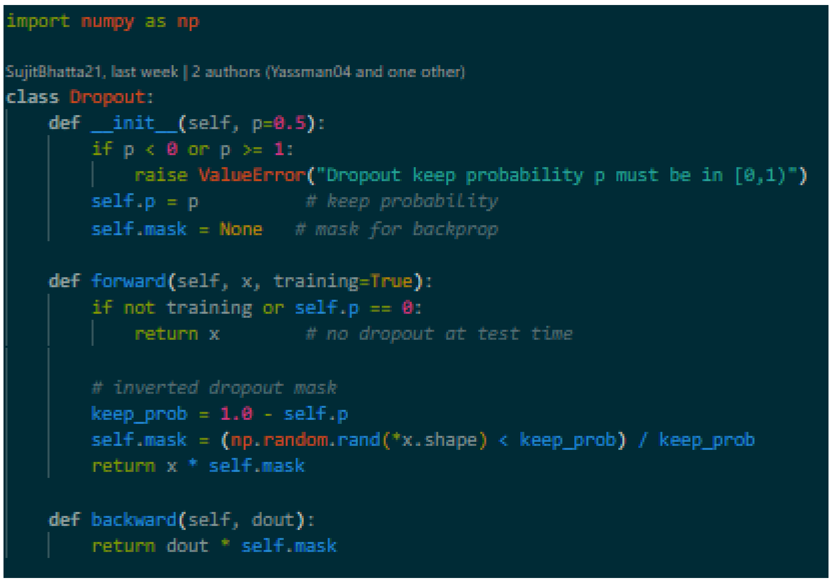
```
import numpy as np        You, 3 weeks ago • Add ReLU activati

SujitBhatta21, 19 hours ago | 2 authors (SujitBhatta21 and one other)
class ReLU:
    """
    Rectified Linear Unit activation function.

    Forward: f(x) = max(0, x)
    Backward: f'(x) = 1 if x > 0, else 0
    """
    def __init__ (self):
        self.cache = None


    def forward(self, x: np.ndarray) -> np.ndarray:
        """
        Forward pass.

        Args:
            x: Input array

        Returns:
            Activated output, same shape as input
        """
        self.cache = x
        return np.maximum(0, x)

    def backward(self, dout: np.ndarray) -> np.ndarray:
        """
        Backward pass computing gradient.

        Args:
            dout: Upstream gradient

        Returns:
            Gradient with respect to input
        """
        x = self.cache
        return dout * (x > 0)
```

K



```
import numpy as np

SujitBhatta21, 19 hours ago | 2 authors (SujitBhatta21 and one other)
class Softmax:
    """      SujitBhatta21, 19 hours ago • Added docString comment cleane
    Softmax activation function.

    Forward: softmax(x_i) = exp(x_i) / Σ exp(x_j)

    Converts logits to probability distribution over classes.
    """
    def __init__(self):
        self.cache = None

    def forward(self, x: np.ndarray) -> np.ndarray:
        """
        Forward pass.

        Args:
            x: Input logits, shape (batch_size, num_classes)

        Returns:
            Probability distribution, shape (batch_size, num_classes)
            Each row sums to 1.0
        """
        # numeric stability: subtract row-wise max
        shifted = x - np.max(x, axis=1, keepdims=True)
        exp_x = np.exp(shifted)
        probs = exp_x / np.sum(exp_x, axis=1, keepdims=True)

        self.cache = probs
        return probs

    def backward(self, dout: np.ndarray) -> np.ndarray:
        """
        When used with cross-entropy, the gradient is already computed
        in backward_pass as (y_pred - y_true) in NeuralNetwork class.
        """
        return dout
```

L



```
import numpy as np

SujitBhatta21, last week | 2 authors (Yassman04 and one other)
class Dropout:
    def __init__(self, p=0.5):
        if p < 0 or p >= 1:
            raise ValueError("Dropout keep probability p must be in [0,1)")
        self.p = p          # keep probability
        self.mask = None    # mask for backprop

    def forward(self, x, training=True):
        if not training or self.p == 0:
            return x        # no dropout at test time

        # inverted dropout mask
        keep_prob = 1.0 - self.p
        self.mask = (np.random.rand(*x.shape) < keep_prob) / keep_prob
        return x * self.mask

    def backward(self, dout):
        return dout * self.mask
```