Backpropagation: The Learning Engine of Neural Networks

NAME : Alpesh Tribhuvan Yadav

Student No: 24073394

Github link: https://github.com/Alpesh-hash/Backpropagation_tutorial.git

1. Introduction

Although neural networks are employed extensively in contemporary machine learning, many practitioners and students initially view them as "black boxes." They receive inputs, generate outputs, and are able to enhance their performance in some way through training. Backpropagation is the main process that makes this learning possible. The method known as backpropagation determines how each weight in the network should alter in order to improve the accuracy of its predictions in subsequent iterations. It gained popularity in the 1980s after Rumelhart, Hinton, and Williams demonstrated in their Nature publication "Learning representations by back-propagating errors" (Rumelhart et al., 1986) that gradient-based optimisation could be used to train multi-layer neural networks. Since then, a variety of neural architectures have made backpropagation their primary training technique (Goodfellow et al., 2016).

I concentrate on backpropagation in particular in this tutorial, looking at its mathematical and philosophical underpinnings and illustrating its behaviour using a fictitious credit-risk dataset. A borrower with real-valued characteristics, including Age, Annual income, Debt-to-income ratio, Number of late payments, Credit history length , Employment

years, and Home ownership status, is represented by each example. Predicting whether the borrower will miss loan payments is the aim. This issue is typical of many real-world applications in risk modelling and finance, where ethical issues are important and precise prediction is essential (Bishop, 2006).

This tutorial's structure highlights backpropagation as the main idea. I simply briefly describe forward propagation as the mechanism that generates predictions and supplies the intermediate values required for differentiation. After that, I describe the logic and mathematics of backpropagation, demonstrate how it is implemented in code, and provide experiments that highlight and illustrate backpropagation's role in learning. Standard deep learning textbooks (Goodfellow et al., 2016) and approachable introductions like Nielsen (2015) served as inspiration for the exposition.

2. Dataset and Neural Network Architecture

A Python program that generates realistic borrower attributes is used to create the dataset. Despite being artificial, the data exhibits characteristics that are similar to those of actual credit datasets: some patterns have a strong correlation with the likelihood of default, while others add noise and fluctuation that make prediction difficult.

Each input vector $x \in \mathbb{R}^7$ contains the following seven features:

1. Age

2. Annual income

3. Debt-to-income ratio

4. Credit history length

5. Number of late payments

6. Home ownership status (0 or 1)

7. Employment years

For every borrower, a weighted sum of normalised characteristics plus random noise is used to calculate a concealed "risk score." The final binary label $y \in \{0, 1\}$ is sampled after this score is run through a sigmoid function to produce a probability of default.

To learn this mapping, I use a small feedforward neural network:

• Input layer: 7 features

• Hidden layer: 8 neurons with ReLU activation

• Output layer: 1 neuron with sigmoid activation

This model is both expressive enough to discover the underlying relationships in the dataset and straightforward enough to comprehend in great detail.

3. Forward Propagation: Producing Predictions

The stage of a training iteration known as forward propagation is when the network calculates its outputs for a certain batch of inputs X and parameters (weights and biases). The pre-activation of the hidden layer for our network is

$$z^{(1)} = W^{(1)}x + b^{(1)}$$

We then apply the ReLU activation function

$$h = ReLU(z^{(1)}) = max(0, z^{(1)})$$

The output neuron computes

$$z^{(2)} = W^{(2)}h + b^{(2)}$$

and the sigmoid activation turns this into a probability of default

$$\hat{y} = \sigma\left(z^{(2)}\right) = \frac{1}{1 + \exp(-z^{(2)})}$$

To train the network, we use the binary cross-entropy loss

$$L(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})),$$

averaged over the batch.

The following Python function shows the core of the forward pass in my implementation, corresponding directly to these equations:

```
def forward(X, params):
    W1, b1 = params["W1"], params["b1"]
    W2, b2 = params["W2"], params["b2"]
    z1 = X @ W1 + b1
    h = np.maximum(0, z1)           # ReLU
    z2 = h @ W2 + b2
```

```
    y_hat = 1 / (1 + np.exp(-z2))  # sigmoid
    cache = {"z1": z1, "h": h, "z2": z2}
    return y_hat, cache
```

4. Backpropagation: Computing Gradients Efficiently

The algorithm that calculates the gradients of the loss with respect to each network parameter is called backpropagation. Gradient descent then uses these gradients to update the weights in the direction that minimises the loss (Rumelhart et al., 1986; Goodfellow et al., 2016). The chain rule is a crucial mathematical tool. We compute derivatives layer by layer, starting from the output, because each weight indirectly influences the output across several layers.

The derivative of the loss with respect to the output pre-activation z(2) simplifies to

$$\delta^{(2)} = \frac{\partial L}{\partial z^{(2)}} = \hat{y} - y$$

Using δ(2), the output-layer gradients are

$$\frac{\partial L}{\partial W^{(2)}} = h^{\top}\delta^{(2)}, \frac{\partial L}{\partial b^{(2)}} = mean\left(\delta^{(2)}\right)$$

The error then propagates into the hidden layer:

$$\frac{\partial L}{\partial h} = \delta^{(2)} {W^{(2)}}^{\top}$$

and, because the hidden layer uses ReLU,

$$\delta^{(1)} = \frac{\partial L}{\partial z^{(1)}} = (\delta^{(2)} W^{(2)\top}) \odot \mathrm{ReLU}'(z^{(1)})$$

where ReLU′(z) is 1 when z > 0 and 0 otherwise. Finally,

$$\frac{\partial L}{\partial W^{(1)}} = X^{\top}\delta^{(1)}, \quad \frac{\partial L}{\partial b^{(1)}} = mean\left(\delta^{(1)}\right)$$

The corresponding Python function for the backward pass is:

```
def backward(X, y, y_hat, cache, params):
    W2 = params["W2"]
    z1, h, z2 = cache["z1"], cache["h"],
cache["z2"]
    N = X.shape[0]

    # Output layer error
    dz2 = y_hat - y                    # δ(2)
    dW2 = h.T @ dz2 / N
    db2 = dz2.mean(axis=0, keepdims=True)

    # Backprop into hidden layer
    dh = dz2 @ W2.T
    dz1 = dh * (z1 > 0).astype(float)  #
ReLU'(z(1))
    dW1 = X.T @ dz1 / N
    db1 = dz1.mean(axis=0, keepdims=True)

    grads = {"dW1": dW1, "db1": db1,
```

```
            "dW2": dW2, "db2": db2}
    return grads
```

5. Training Loop: Putting Forward and Backward Together

The training loop combines forward propagation, backpropagation, and parameter updates. In each epoch we:

1. Run forward() to compute $\hat{y}$ and the loss.
2. Run backward() to compute gradients.
3. Update all weights and biases with a fixed learning rate η.

The core of the training loop in my notebook is:

```
def train_model(X_train, y_train, params, lr=0.01,
epochs=200):
    history = {"loss": []}

    for epoch in range(epochs):
        # 1. Forward pass
        y_hat, cache = forward(X_train, params)
        loss = binary_cross_entropy(y_train, y_hat)
        history["loss"].append(loss)

        # 2. Backpropagation
        grads = backward(X_train, y_train, y_hat,
cache, params)
```

```
        # 3. Gradient descent update
        params["W1"] -= lr * grads["dW1"]
        params["b1"] -= lr * grads["db1"]
        params["W2"] -= lr * grads["dW2"]
        params["b2"] -= lr * grads["db2"]

    return params, history
```

6. Experiments and Plots

6.1 Forward-only (no learning) vs Forward + Backpropagation

To show that backpropagation is essential for learning, I ran two experiments with the same initial weights:

- Forward-only: in each epoch I performed a forward pass and computed the loss, but I did not call backward() and did not update the weights.
- Forward + backpropagation: in each epoch I ran forward(), backward() and a gradient-descent update.

Because the weights never vary, the forward-only training loss in the resulting loss curves is basically flat, staying around 0.69. The model is learning a better mapping from borrower attributes to default probability, as evidenced by the curve for the backpropagation-trained model decreasing with time.

The related accuracy curves show that the model trained with backprop gradually improves to about 0.68, while the forward-only accuracy

remains close to its starting baseline at about 0.53. When taken as a whole, these plots demonstrate that learning requires both backpropagation and weight updates.

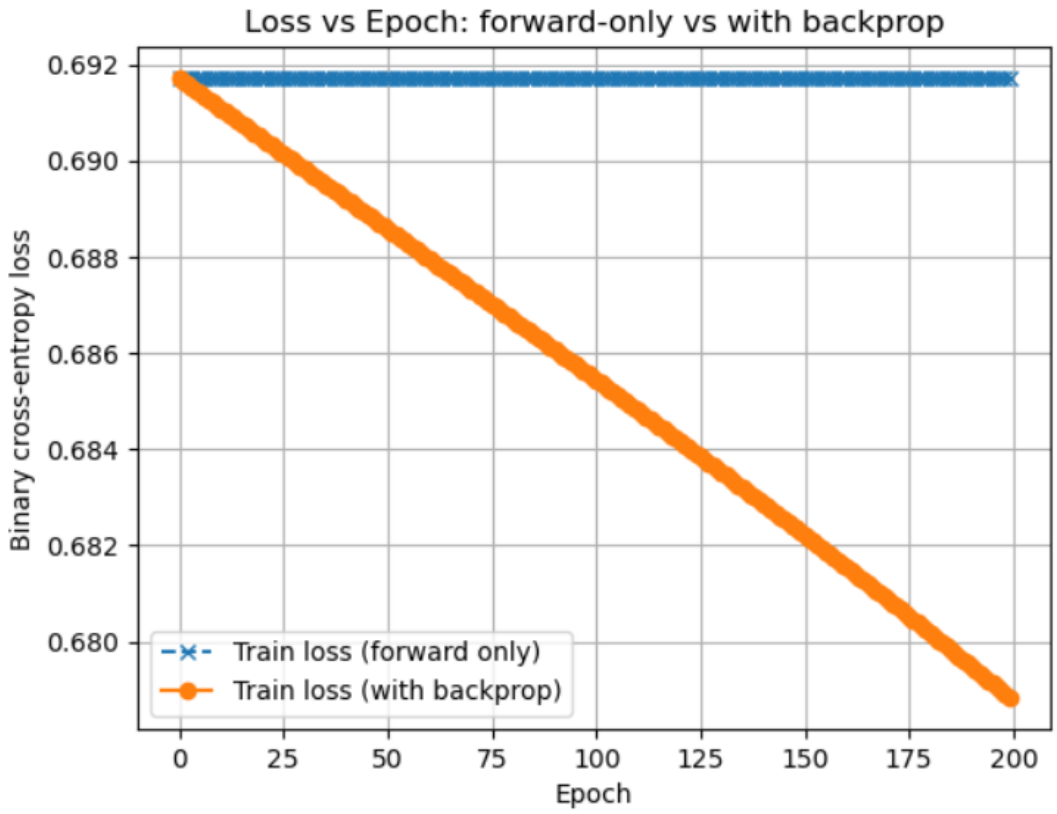

Figure 1: Training loss vs epoch for forward-only (no learning) and forward + backpropagation.

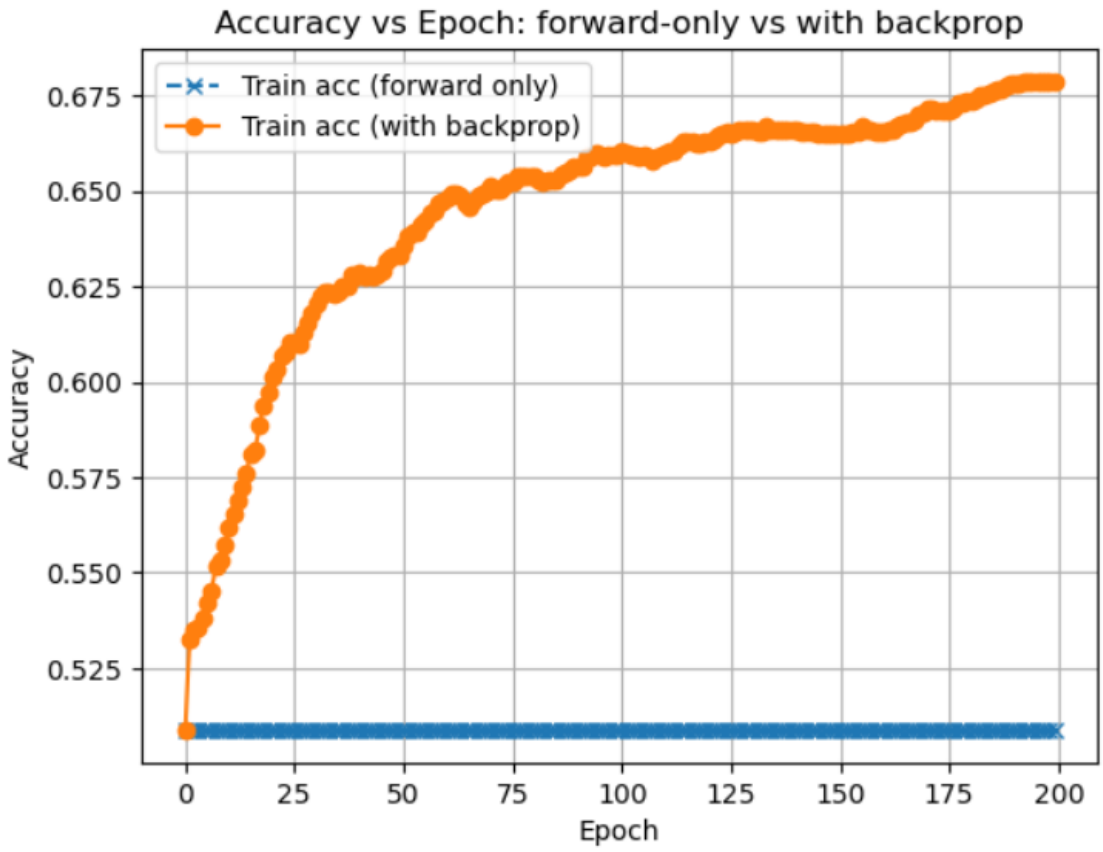

Figure 2: Training accuracy vs epoch for forward-only (no learning) and forward + backpropagation.

6.2 Effect of Different Learning Rates

Additionally, I trained the same network using three distinct learning rates: η = 0.001, 0.01, and 0.1. The loss diminishes extremely slowly when η = 0.001. The loss drops more clearly but conservatively when η = 0.01. For this straightforward case, the loss decreases significantly more quickly when η = 0.1. This demonstrates the basic idea that while a higher learning rate can speed up training until it becomes unstable, a very low learning rate results in slow learning (Goodfellow et al., 2016; Machine Learning Mastery, 2020). Figure 3 displays the corresponding plot.

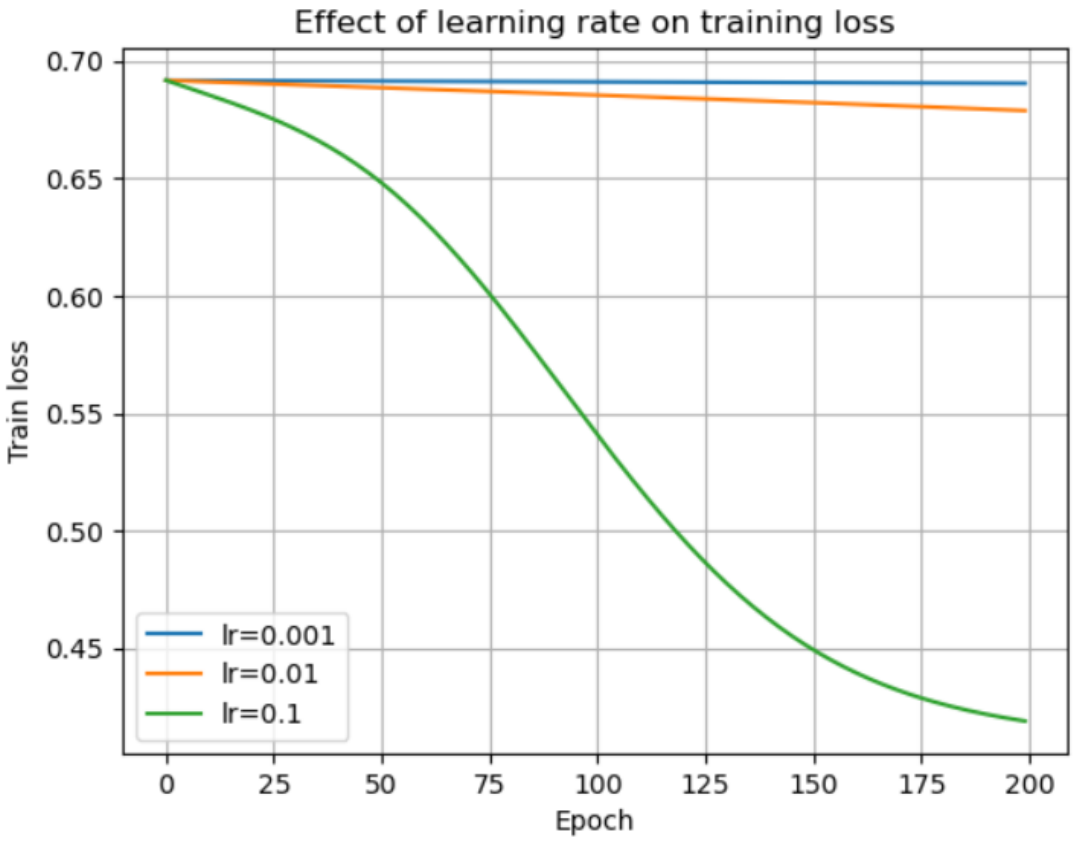

Figure 3: Training loss vs epoch for three different learning rates (η = 0.001, 0.01, 0.1).

7. Gradient Checking and Validation

I utilised gradient testing on a single weight to compare the analytic gradient from backward() with a numerical finite-difference approximation because backpropagation implementations are prone to subtle mistakes. This is demonstrated for a single weight W1[0, 0] in the following code:

```
epsilon = 1e-5
i, j = 0, 0

# Analytic gradient
```

```
y_hat, cache = forward(X_batch, params)
grads = backward(X_batch, y_batch, y_hat, cache,
params)
analytic = grads["dW1"][i, j]

# Numerical gradient
original = params["W1"][i, j]
params["W1"][i, j] = original + epsilon
loss_plus = binary_cross_entropy(y_batch,
forward(X_batch, params)[0])
params["W1"][i, j] = original - epsilon
loss_minus = binary_cross_entropy(y_batch,
forward(X_batch, params)[0])
params["W1"][i, j] = original

numeric = (loss_plus - loss_minus) / (2 * epsilon)
```

If the analytic and numerical gradients agree closely, this provides strong evidence that the backpropagation implementation is correct (Nielsen, 2015).

8. Ethical Considerations

Credit-risk prediction is a field with important ethical issues, despite the fact that this course uses artificial data. Any statistical patterns in the data, including potentially dangerous biases, will be faithfully learnt using backpropagation. Careful assessment is necessary for practical implementation to guarantee equity, openness, and adherence to moral and legal requirements (Bishop, 2006). Practitioners should think about how models behave in various demographic groups, what features are suitable to add, and how judgements are conveyed and audited.

9. Conclusion

This tutorial has demonstrated how learning in a neural network is made possible by the combination of forward propagation and backpropagation. While backpropagation uses the chain rule to track each weight's impact throughout the network, forward propagation calculates predictions and losses. Gradually, the model gets better at predicting the credit-risk task by using gradient descent to update weights. Visual proof of these effects may be found in the learning-rate experiment and the charts comparing forward-only and backpropagation training.

10. References

Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

Machine Learning Mastery. (2020). Understand the Dynamics of Learning Rate on Deep Learning Neural Networks. Available at: https://www.machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

Nielsen, M. (2015). Neural Networks and Deep Learning. Determination Press. Available online at http://neuralnetworksanddeeplearning.com

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323, 533–536. Retrieved from https://www.nature.com/articles/323533a0