Backpropogation algortihm:

Backpropagation is basically the network's way of teaching itself what it did wrong and how to fix it. When you give a neural network some input, it pushes that data forward through all its layers until it produces an output. Then it compares that output to the correct answer and measures how bad the mistake was. Once it knows the size of the mistake, it works backward through the network and figures out which neurons and which weights contributed to that mistake. The key idea is that every weight gets a "blame score" that tells it how much it influenced the final error. The network then adjusts each weight slightly in the direction that would have reduced the error. Doing this repeatedly over many examples gradually shapes the network into something that makes better predictions.

The implementation is straightforward in concept: you run a forward pass to compute outputs, compute the loss, then run a backward pass where each layer receives an error signal from the layer above it, computes how much it contributed, and passes the remaining error backward. Frameworks like PyTorch or TensorFlow automate this, but under the hood it's just systematic bookkeeping of how each weight affects the final output.

The softmax function is used when you want the network to output probabilities over multiple classes. It takes a list of raw scores from the final layer and converts them into values that sum to 1 and behave like probabilities. The highest score becomes the highest probability, but everything stays nicely normalized. It's especially useful in classification tasks because it forces the model to commit to a distribution rather than arbitrary numbers.

Common nonlinear output functions depend on the task. For binary classification, you typically use a sigmoid function, which squashes the output into a value between 0 and 1. For multi-class classification, you use softmax. For regression tasks, you often don't use a nonlinear output at all, you just let the final layer output a raw number. The point of these output functions is to shape the network's final prediction into a form that matches the type of problem you're solving.

First error: output layer error computation.

At this point, the network has already completed a forward pass. Each input image has gone through all layers and the network has produced its final prediction for each class. These predictions are probabilities that indicate how confident the network is that the image belongs to each digit class.

What happens next is that the network compares its prediction to the correct label. The correct label is represented in a way that makes it directly comparable to the predicted probabilities. By subtracting the correct label information from the prediction, the network computes how wrong it is for each class. This difference represents the error signal at the output layer.

This error signal tells the network two things at the same time: how large the mistake is and in which direction the prediction needs to change. This is the starting point of backpropagation. From here on, the network does not look at the input anymore, only at how the error should be sent backwards through the layers.

Second error: backward propagation through hidden layers.

This stage propagates the error from the output layer back through the hidden layers, one layer at a time. The loop goes backwards because learning needs to assign responsibility for the final mistake to earlier layers.

For each hidden layer, the network takes the error coming from the layer after it and translates that error into something meaningful for the current layer. This is done by considering how strongly each neuron in the current layer influenced the neurons in the next layer. Bias connections are ignored in this step because they do not represent real neuron activations.

After the error has been transferred backward, it is adjusted based on how active each neuron in the current layer was. Neurons that were not active should not receive much blame, while neurons that strongly influenced the output should receive more. The result is an error signal for the current layer that correctly reflects its contribution to the final prediction error.

This process repeats until the first hidden layer is reached. At the end of this step, every layer in the network knows how much it contributed to the final error.

Third error: weight update and learning rate.

Once every layer has its error signal, the network is ready to learn. Learning happens by slightly adjusting the weights so that similar mistakes become less likely in the future.

For each layer, the network looks at two things: the error signal from the layer and the activations that were sent into that layer during the forward pass. Together, these describe how each weight contributed to the mistake. The weights are then updated in the direction that reduces the error.

The learning rate controls how large these updates are. If the learning rate is small, the network learns slowly but safely. If it is large, the network learns faster but risks becoming unstable and failing to converge.

After this step, one training iteration is complete. Repeating this process over many epochs gradually improves the network’s performance.

Changing eta from 0.05 to 0.005

With a learning rate (eta) of 0.05, the network converges very quickly during the initial epochs. This is visible in the sharp decrease of both training and test error within the first few iterations. However, as training progresses, the training error continues to decrease until it reaches zero, while the test error stops improving significantly and shows small fluctuations. By the final epoch, the test error stabilizes at approximately 0.0299, corresponding to a test accuracy of about 97.0%. This behavior indicates that the relatively large learning rate enables fast learning but limits fine-grained adjustments of the weights, which can slightly reduce generalization performance and lead to overfitting of the training data.

When the learning rate is reduced to 0.005, the learning process becomes noticeably slower at the beginning, as reflected by higher training and test errors in the early epochs. Nevertheless, the decrease in error is more gradual and stable over time. The network continues to improve steadily and eventually reaches a lower test error of approximately 0.0239, corresponding to a test accuracy of about 97.6%. The smaller learning rate allows the model to make more controlled updates to the weights, which improves stability and leads to better generalization.

Overall, a higher learning rate accelerates early convergence but may hinder precise optimization in later stages, while a lower learning rate slows down training but can result in improved final performance and better generalization.

eta 0.5

With learning rate (eta) equal to 0.5, the training process does not converge. The training and test errors remain very high, hovering around approximately 0.89 to 0.91 across almost all epochs, with only small random-looking fluctuations. This corresponds to very low accuracy, close to chance-level performance for a 10-class problem, meaning the network is not successfully learning meaningful decision boundaries.

The underlying reason is that eta = 0.5 produces weight updates that are too large. Instead of gradually moving the parameters toward a region that reduces the loss, each update step tends to overshoot good solutions. As a result, the optimisation process becomes unstable: any partial improvement achieved in one mini-batch is typically undone by the next update, preventing consistent reduction of the error. In practical terms, the model repeatedly “jumps” through parameter space rather than settling into a configuration that fits the data.

This explains the qualitative difference across the three runs: eta = 0.005 yields slow but stable improvement and the lowest final test error; eta = 0.05 yields fast convergence but slightly worse final generalisation; eta = 0.5 is too large and prevents learning altogether.

Relu

When the sigmoid activation function in the hidden layers was replaced with the ReLU activation function, the training dynamics of the multilayer perceptron changed noticeably. Initially, the network exhibited high and nearly constant training and test error values, indicating that learning was slow at the beginning. This behavior is consistent with ReLU-based networks, where a significant portion of neurons may initially produce zero output, resulting in limited gradient flow during early training.

Despite this slow start, the network eventually began to learn meaningful representations. As training progressed, the error decreased sharply, and by the end of the training process the training error reached zero while the test error decreased to approximately 0.023. This corresponds to a classification accuracy comparable to, and slightly better than, the sigmoid-based configuration with the same number of epochs.

The results demonstrate that ReLU can be successfully used in the hidden layers of the network, but its performance is sensitive to the learning rate and implementation details. Once these conditions are satisfied, ReLU enables effective training and good generalization performance. This experiment confirms that different activation functions can lead to distinct learning behaviors while achieving similar final classification accuracy when properly configured.