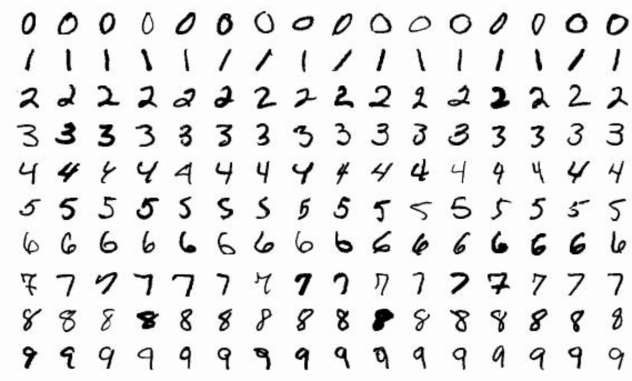
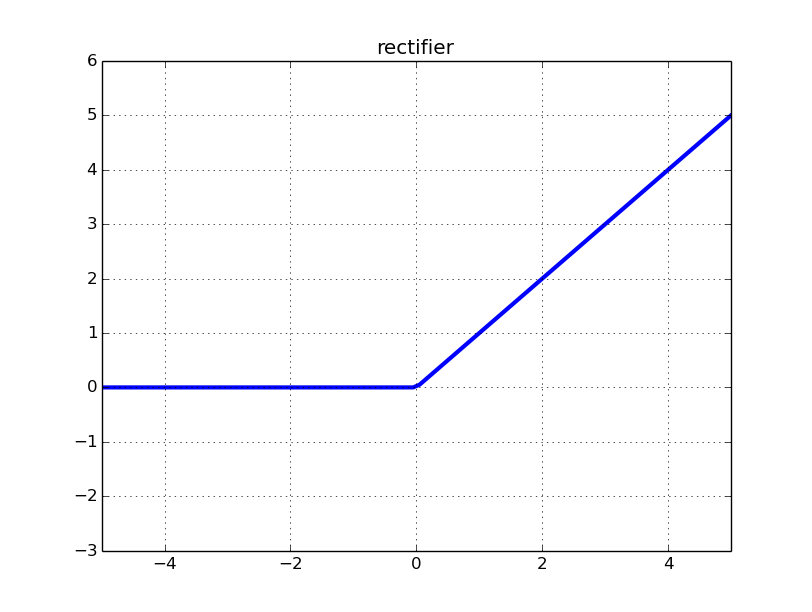
# HandwrittenDigitClassifier

This is a handwritten digit classifier using a **feedforward neural network** built from scratch using only NumPy and mathematics. The dataset consists of 28x28 images, each pixel being a number between 0 (white) and 255 (black). Thus, each image can be represented as a matrix with 784 numbers to represent each pixel value. We will have one large matrix, with *m­*—the number of images in the dataset—columns, each column representing one image in the dataset. We will call this matrix A. Within each column, we get another matrix with 784 rows to represent each pixel in the image it represents.



We will have 10 classes of digits, 0-9, that the neural network will try to classify the image data into. The first layer of the neural network—the input layer—will consist of 784 nodes for each pixel of the given image. The second layer is a hidden layer with 10 nodes, and, finally, the output layer will consist of 10 nodes representing the digits. **Forward propagation**—a recognition-inference architecture—in the first layer will be done by taking the dot product of a new 10x784 **weight** (W) matrix by the A matrix to make a 10xm Z matrix. The values within the W matrix will represent the 7840 connections between the input layer nodes and the hidden layer. The Z matrix will also have a BIAS (B) matrix added to the dot product of A and W to indicate patterns beyond those that pass through the origin (think y = mx+b!).

Next, we need an **activation function** to take the weighted sum of inputs and apply a transformation to produce an output. We will use a **ReLU: rectified linear unit**.  ReLU outputs the input directly if it's positive, or 0 otherwise. ReLU is advantageous because it avoids saturation, alleviating the **vanishing gradient problem**—when the gradient of the error/loss function becomes so small that it cannot be used to adjust the weights and bias.  ReLU applied to the Z will comprise the hidden layer. The output layer will be equal to a second W matrix, comprised of values representing the connection between the hidden layer, and the output layer's nodes, dot multiplied by the hidden layer matrix—ReLU(Z)— added to another bias vector.



[This Photo](https://sefiks.com/2017/08/21/relu-as-neural-networks-activation-function/) by Unknown Author is licensed under [CC BY](https://creativecommons.org/licenses/by/3.0/)

The final output layer matrix will have the **SoftMax** function—which takes a tuple of K real numbers and turns them into a probability distribution of K possible outcomes—as its activation function to take the output matrix and produce probabilities from it, describing how likely each final value is to be a class of digit.

We will use **backpropagation** and **gradient descent** to find the error and see how the previous weights and biases can be adjusted to minimize the error. This will allow the model to "learn" which images correspond to which digits. Once it adjusts, it will be passed back into the forward propagation, then back into backpropagation to minimize the error further. Repeat this cycle until it is fully accurate! The first equation of the algorithm looks like this.

This equation represents the **gradient** of the **loss function** (aka the direction and magnitude of the steepest increase of the differences between our model’s prediction and the target value at a specific point) with respect to the inputs of our output layer, before activation. Y represents the target labels, using a **one-hot** encoded vector. One-hot essentially represents categorical data, like our digit labels, as binary vectors for our model to conduct calculations with. Here is how that looks like mathematically.

Taking dZ^2, we plug it into the next equation. This equation will compute the gradient of the loss with respect to the weights W^2 of the output layer in our neural network. Essentially, this calculates how much each weight in W^2 contributed to the loss. Stronger activations of A^1’s neurons will lead to larger weight updates, prioritizing learning from active and more important nodes. dZ^2 indicates the direction of the loss; if it is negative, the weights are increased, if positive they are decreased. This contribution is averaged with the number of data points to keep it proportional to the batch size.

Next, we want to calculate the gradient of the loss with respect to the bias b^2 of the output layer. Like the last equation, we will take all the model’s errors and divide it over the number of data points. This shows us how much to adjust the bias term by in our output layer.

Now we want to go *back* to our hidden layer and compute the gradient of the loss with respect to the Z^1 before activation (aka how sensitive the loss function is to changes in the raw inputs of the hidden layer). We can do this by taking W^2^T to distribute the output error dZ^2 backward to each neuron in our hidden layer before activation, then finally “filter” those neurons using ReLU’s derivate with respect to Z^1. This operation determines how much each neuron in the hidden layer contributed to the error, then deactivates negative inputs that have ReLU “off”, and no updates occur to their weights and their gradients are zeroed out. Positive neurons will stay active and have their weights updated.

Now we want to calculate the gradient of the loss with respect to the weights W^1 in our hidden layer (aka how much each weight in W^1 contributed to the error). We can do this by taking the error signal dZ^1 to determine the direction of the update (if dZ^1 is positive, the weights are reduced; if negative, they are increased) and scaling it with the inputs A^0 (e.g. stronger input features lead to larger weight updates). Then we will take the average of this like the previous equations.

The next step of this algorithm will be to see how much each bias b^1 in the hidden layer contributed to the error. This is essentially the same as the db^2 equation, where we sum the how much each node in the hidden layer contributed to the loss and average it out over the number of data points.

Finally, we will update our weights and biases in the hidden and output layer. α is the **learning rate**, which is a small positive number that controls how big the update steps are when adjusting the weights and biases. Our gradients (dW^1, db^1, etc.) tell us which direction to move to decrease the loss, but not how big of a step to take. In our model, we set our learning rate to 0.05.

# Improvements

Upon first testing this model, I realized that it takes quite a while (200+ iterations) to reach reasonable accuracy rates (60%+), and it would start at a 10% accuracy rate. I decided to improve the initial design.

Number of Neurons

First, I increased the number of neurons in the hidden layer from 10 to 64. Before we had 7840 connections between the input and hidden layers, now we have 50,176 connections. This allows the model to make more connections between seemingly unrelated ideas, enabling more complex outputs. More neurons are also advantageous because the model now has more learnable parameters, since each neuron adds weights and biases. This allows the model to model more complex problems. For example, a single neuron can only represent linear lines, whereas multiple neurons can represent nonlinear, discontinuous, fractals, etc. and make connections between those ideas to create much more complex outputs.

Mini-Batch Processing

Second, I implemented mini-batch processing which selects a group of 64 images from the dataset to train on at a time, instead of taking in all 40,000 images at once, increasing efficiency. This will allow the computer to do less math as implementing these equations with 64 datapoints instead of 40K will be much less CPU-intensive.

Xavier Initialization

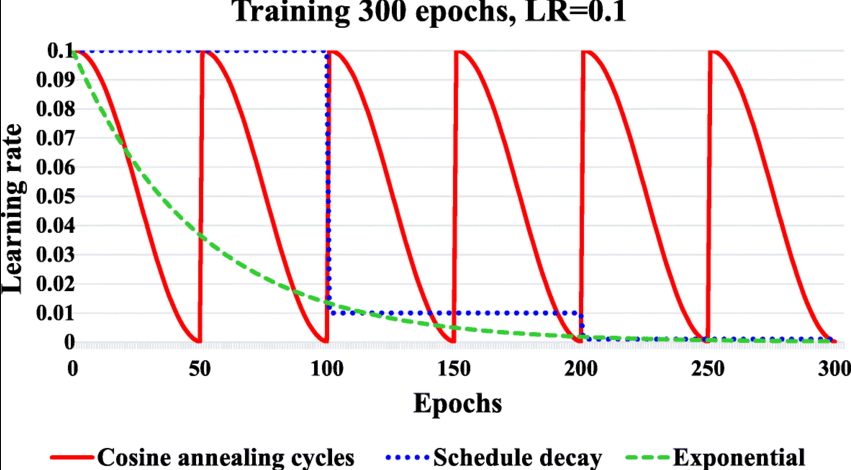
Third, I implemented **Xavier initialization** when initiating our weights. We used the Normal Distribution version of Xavier. Initially, I initiated the weights to random numbers between -0.5 and 0.5, but now we will multiply that why the following formula. Xavier is advantageous because it keeps our activations and gradients within a healthy range, preventing the vanishing gradient problem or **exploding gradients**---when gradients get huge, unstable, and the loss function is unable to process them. With Xavier, our data variance is preserved across layers, creating proper gradient updates. This will create faster convergence and significant training time reductions.

L2 Regularization

Fourth, I implemented **L2 regularization** during our calculations of dW^1 and dW^2. We will have a value named lambda that will be our regularization strength, essentially controlling the penalty size in the loss function when the model makes an error. This helps control **overfitting**---when the model is trained too well for a specific dataset causing it to learn the specific natural variance of the dataset, which makes it inaccurate and fail on other datasets. Below is the generalized loss function in our model before and after regularization, as well as the updated dW^1 and dW^2 formulas.

Learning-Rate Decay

Fifth, I implemented **learning-rate decay**.There are many ways to implement learning-rate decay, however we will use **Cosine Annealing**. It starts with a high learning rate and gradually decreases it to a minimum value, then restarts this cycle. This is advantageous because it allows our model to take larger initial steps to explore the solution space and then increasingly finer adjustments as it converges to the ideal outcomes. Below is the mathematical and graphical representation of Cosine Annealing.



Early Stop

Sixth, I implemented a mechanism to stop early if accuracy rates decline or plateau. Initially when I trained the model, when it reached the 400th iteration, the accuracy rate would begin to decline from 85.5% down to ~82% before I would stop the program. Thus, I implemented some Python code that will check the learning rate and stop if it declines a certain amount. This will avoid extra work that the CPU doesn’t need and shouldn’t do.

# Results

These reforms have enabled it to start training with a 90% accuracy rate and reach up to 98% accuracy within 150 iterations. (I’ll add tables and graphs comparing metrics later)