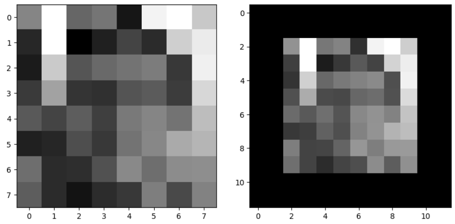
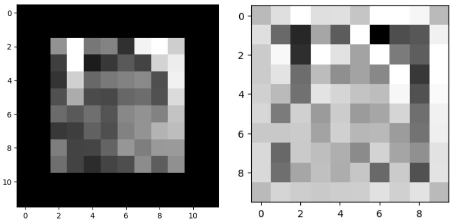
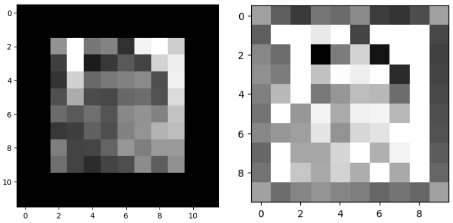
Padding: We want the kernel (filter) to move over the image matrix (the next step). However, since we want it to have an equal effect on every value, **we fill the edges of our matrix with '0'**.



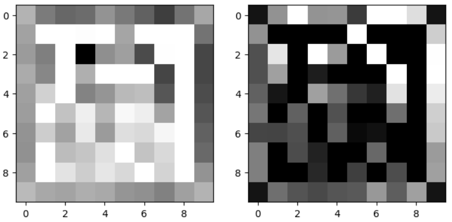
Full Correlation: The image matrix is traversed with a special kernel (filter). The values of the kernel and the corresponding image area are multiplied, then the matrix values are summed up and turned into a single float value, which is then written to the new area. **This is used to learn features in the image (edges, corners, patterns).**

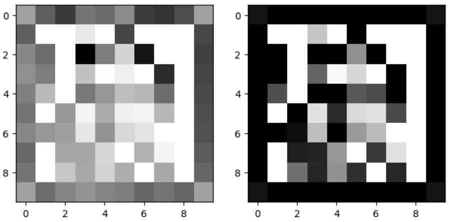




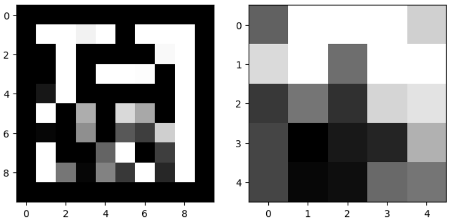
Activation (ReLU, Sigmoid):

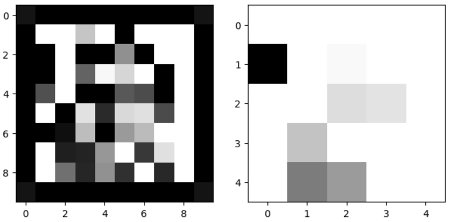
- **ReLU: It works by setting negative values to zero and keeping positive values unchanged.** This function helps gradients propagate more effectively, especially in deep networks.
- **Sigmoid**: It provides an S-curve activation with **outputs in the range [0, 1].** It is particularly used in binary classification problems.





Pooling (Max Pooling): We move a 3x3 matrix over the image. The maximum value in the area covered by this 3x3 matrix is taken and written to the new matrix. **This is used to reduce the size of the image.**





Initialization of Filters and Weights:

We have two filters, and each filter has a size of 5x5:

f1 = **initializeFilter**(size, scale = 1.0)
f2 = **initializeFilter**(size, scale = 1.0)

And We've got 2 hidden layers, each layer has 10 weight. They are size of 10x500 and 10x10:

w3 = **initializeWeight**(size)
w4 = **initializeWeight**(size)

implementation of **initializeFilter:**	implementation of **initializeWeight:**
$stddev = \frac{scale}{\sqrt{\prod_{k=0}^{n} size[k]}}$ $\mathcal{N}(0, stddev)$	$\mathcal{N}(0, 0.01)$

> *$\mathcal{N}(\mu, \sigma)$: denotes a normal distribution with mean μ and variance σ.*

Randomizing Images to Prevent Over-fitting:

1. **Scaling:**

   Randomly select a **scale factor between 0.9 and 1.1**, **resize the input image** with selected scale factor.

2. **Rotation:**

   Randomly **selects an angle between -45 and 45 degrees**, calculate center of the image then **apply rotation to the scaled image** using the rotation matrix.

3. **Translation:**

   Randomly select **translations in the x and y directions between -2 and 2 pixels**, construct a translation matrix based on the selected translations. **apply translation to the rotated image** using the translation matrix.

4. **Additive Noise:**

   Randomly select a **probability between 0 and 0.3**, generate a binary mask of the same shape as the translated image based on the selected probability.

   Create random noise values of the same shape as the translated image, scaled between -0.5 and 0.5, **combine the translated image with the noise values based on the binary mask.**

   Clip the resulting noisy image to ensure pixel values are within the range [0, 1].

Forward-Propagation:

1. **First Convolution Layer (Conv2D + ReLU):**

   $\mathit{conv1} = \mathbf{Conv2D}(\mathit{image}, \text{f1}, \text{b1}, \text{conv\_stride})$

   $\mathit{conv1} = \mathbf{ReLU}(\text{conv1}) = \mathbf{max}(0, \text{conv1})$

2. **Second Convolution Layer (Conv2D + ReLU):**

   $\mathit{conv2} = \mathbf{Conv2D}(\mathit{conv1}, \text{f2}, \text{b2}, \text{conv\_stride})$

   $\mathit{conv2} = \mathbf{ReLU}(\text{conv2}) = \mathbf{max}(0, \text{conv2})$

3. **Max Pooling:**

   $\mathit{pooled} = \mathbf{MaxPool2D}(\mathit{conv2}, \text{pool\_size}, \text{pool\_stride})$

4. **Flattening the Pooled Layer:**

   $\mathit{fc} = \mathbf{Flatten}(\mathit{pooled}) = \text{pooled}\mathbf{.reshape}((\text{shape of } \mathit{pooled} \text{ in 1 dimension}))$

5. **Fully Connected Layer:**

   $\mathit{z} = \text{W3} \cdot \mathit{fc} + \text{b3}$

   $\mathit{z} = \mathbf{ReLU}(\mathit{z}) = \mathbf{max}(0, \mathit{z})$

   $\mathit{out} = \text{W4} \cdot \mathit{z} + \text{b4}$

   $\mathit{probs} = \mathbf{Softmax}(\mathit{out})$

Calculating Cross-Entropy Loss:

> *probs*: The result from the final layer of the Fully Connected Layer (Softmax), which is our prediction. **(Ŷ)**

> *label:* The true label **(Y)**

> ϵ: A very small value added to avoid taking the log(10). **(for exp. 10^{-10})**

$\mathit{probs} = \mathbf{max}(\epsilon, \mathbf{min}(\mathit{probs}, 1-\epsilon))$

$\mathit{loss} = -\sum(\text{label} \cdot \log(\mathit{probs}))$

Backpropagation:

1. **Compute** *dout*:

   > *probs*: The result from the final layer of the Fully Connected Layer (Softmax), which represents our prediction. **(Ŷ)**
   >
   > *label:* The true label. **(Y)**

   $$dout = probs - \text{label}$$

2. **Compute** *dw4* (Partial Derivative of *Loss* with respect to W_4):

   $$dw4 = \frac{\partial Loss}{\partial W4} = \frac{\partial Loss}{\partial out} \cdot \frac{\partial out}{\partial W4} = dout \cdot z^{\mathrm{T}}$$

3. **Compute** *db4* (Partial Derivative of *Loss* with respect to b_4):

   $$db4 = \frac{\partial Loss}{\partial b4} = \frac{\partial Loss}{\partial out} \cdot \frac{\partial out}{\partial b4} = \sum \text{dout}$$

4. **Compute** *dz* and *ReLU* (Partial Derivative of *Loss* with respect to *z*):

   $$dz = \frac{\partial Loss}{\partial z} = \frac{\partial Loss}{\partial out} \cdot \frac{\partial out}{\partial z} = W_4^T \cdot dout$$

   $$dz = \text{ReLU}(z) \cdot dz$$

5. **Compute** *dw3* (Partial Derivative of *Loss* with respect to W_3):

   $$dW3 = \frac{\partial Loss}{\partial W3} = \frac{\partial Loss}{\partial z} \cdot \frac{\partial z}{\partial W1} = dz \cdot fc^{\mathrm{T}}$$

6. **Compute** *db3* (Partial Derivative of *Loss* with respect to b_3):

   $$db3 = \frac{\partial Loss}{\partial b3} = \frac{\partial Loss}{\partial z} \cdot \frac{\partial z}{\partial b3} = \sum dz$$

7. **Compute** *dFc* and *dPool,* Then flatten *dPool*:

   $$dFc = \frac{\partial Loss}{\partial Fc} = W_3^T \cdot dz$$

   dPool = dFc.**reshape**((shape of *pooled*))

8. **Compute** *dConv2* and Backpropagate Max Pooling Layer:

   dConv2 = **MaxPool2D_Backward**(*dPool*, *conv2*, pool_size, pool_stride)

   dConv2 = **ReLU(***conv2***)** · *dConv2*

9. **Compute** *dConv1, dF2, db2* and Backpropagate Convolution Layer:

   dConv1, df2, db2 = **Convolution_Backward**(dConv2, conv1, f2, conv_stride)

   dConv1 = **ReLU(**conv1**)** · dConv1

10. **Compute** *dImage, dF1, db1* and Backpropagate Convolution Layer:

    dImage, df1, db1 = **Convolution_Backward**(*dConv, conv2, f1,* conv_stride)

Adam (Adaptive Moment Estimation) Optimization:

It adapts the learning rate for each parameter by computing individual learning rates for each parameter from estimates of first and second moments of the gradients.

1. **Initialize parameters**

   θ, (Model Parameters)
   $m = 0$, (First Moment)
   $v = 0$, (Second Moment)

2. For each ***parameter* (θ)** in ***Parameters***:

   > ***g*** *is our gradient such as df1, db1, dw3*
   > ϵ *is a very small number such as* 10^{-10}*, that prevents us from getting dived by 0*

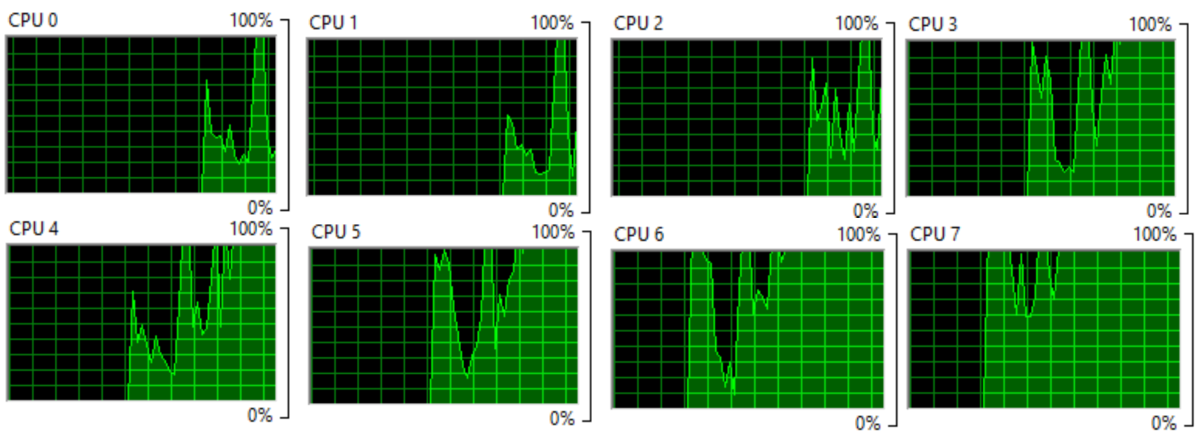
   $m = \beta_1 \cdot m + (1-\beta_1) \cdot g / batch\_size$
   $v = \beta_2 \cdot v + (1-\beta_2) \cdot (g / batch\_size)^2$
   $\theta = \theta - \frac{learningRate \cdot m}{\sqrt{v} + \epsilon}$

Using Multiple Threads For Faster Computing:

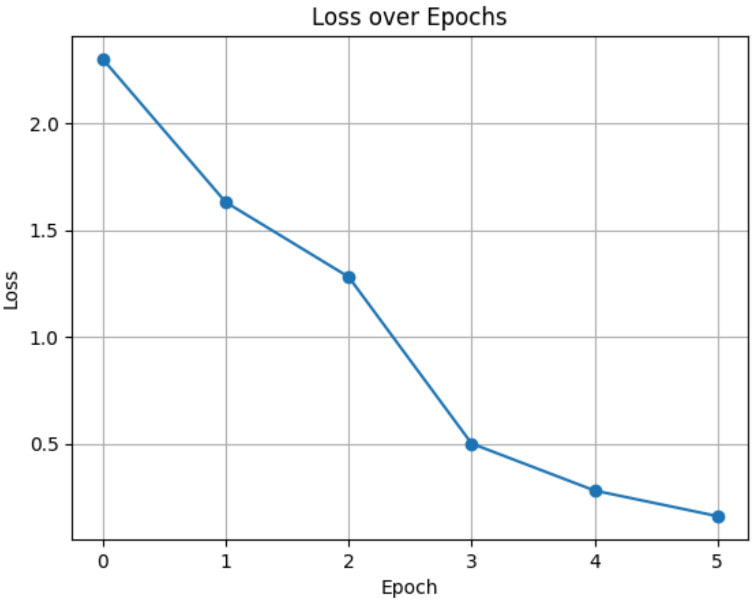
Due to the extensive size of our training dataset, it's beneficial to split our batches into smaller sections. This enables us to leverage asynchronous processing with multiple threads. By doing so, we can distribute the workload across various threads, allowing for simultaneous computation on smaller subsets of data.

Batch Size = 1024
Total Training Data = 81.000
Number of Epochs = $\frac{Total\ number\ of\ training\ data}{BatchSize}$
Number of Processes = 8

So we split our batches into 8 sections, then asynchronously processing data with multiple threads.

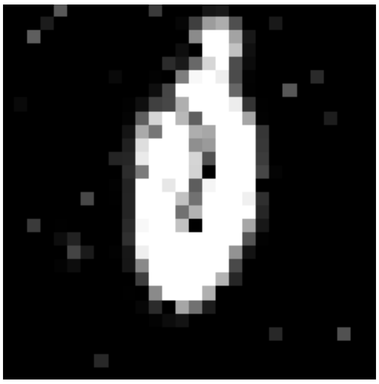
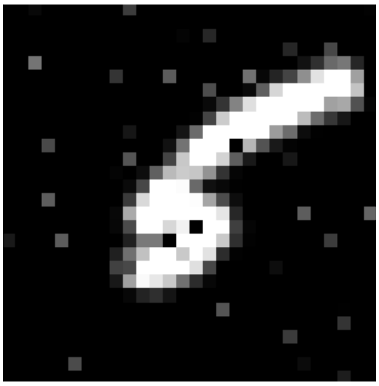
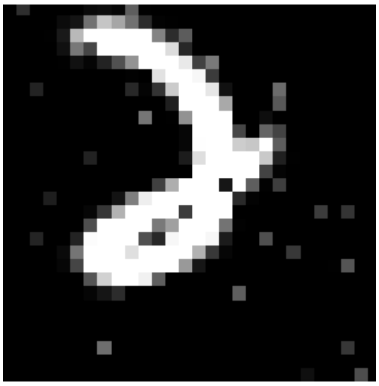
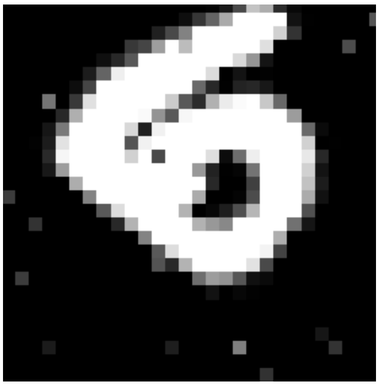
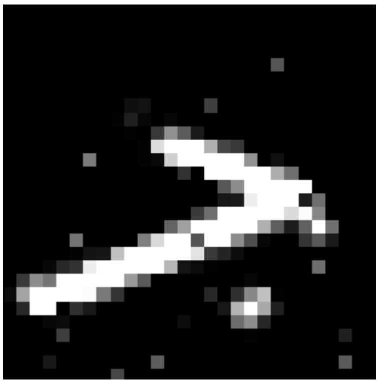


Loss Values Over Epochs:



with final accuracy of ***93.252%***

T.L = **T**rue Label, **P** = **P**rediction

 **T.L : 0, P : 0**	 **T.L : 5, P : 5**	 **T.L : 2, P : 7**
 T.L : 9, P : 9	 **T.L : 6, P : 0**	 **T.L : 7, P : 7**