Augmentation is all you need: training ResNet-18 for adversarial digit classification

Arjun Sharma
asharma3@caltech.edu

Summary

This report details my attempt to train a performant convolutional neural network classifier (CNN) on the provide 'MNIST-in-the-wild' (MNIST-W) dataset. I achieve **94.997**% validation accuracy using a model based on the Resnet-18 architecture containing 11,181,642 parameters. The model is trained on 85% of the provided MNIST-W samples, which are passed through an aggressive data augmentation pipeline; as well as on a 'synthetic' dataset consisting of images from the original MNIST dataset projected onto backgrounds in the MNIST-W dataset.

Experiments show that data augmentation is crucial for preventing overfitting in the model, and is responsible for a large part of the performance gains with a deep network. The synthetic dataset provides some additional, albeit limited, gains. An interactive dashboard[1] visualizing the data pre-processing and model performance is available at arjuns-148project2.netlify.app.

I include all components required by the project instructions, but swap the order of the first two parts: in section 1, I describe the data processing steps, particularly augmentations; in section 2, I provide a high-level overview of the basic training setup.

My final notebook is: https://drive.google.com/file/d/1KzDG4alqDO7AiCok8-jheFtcmccldoUo/view?usp=sharing.

1 Data and preprocessing

Data. The provided dataset consists of 10,294 images of digits, each of which has one of 10 labels. There are 873 ± 16 images in each class, i.e., the difference in the number of samples is at most 3.67% between any two classes. I therefore chose not to balance this dataset. All images are transformed to 128×128 later, as described in the dataloader implementation section.

Split. I chose an 85:15 train:validation split, which results in 8755 training samples and 1,539 validation samples. This split was stratified, i.e., I did not randomly selected 15% of the entire dataset directly to generate the validation dataset; rather, for each class, I selected 15% of the dataset to assemble the validation dataset. This ensured that the distribution of classes in the training and validation sets was as close to identical as possible.

Synthetic data. To improve the performance of the model, I wanted to expand the size of the dataset by incorporating more digit images, like from the original MNIST dataset. However, by the construction of this project, the original MNIST is approximately drawn from another distribution than the provided MNIST-W dataset. Therefore, I attempt to perturb MNIST images to make them resemble something like the MNIST-W dataset. Specifically, for each of the N_{synth} images in MNIST (where N_{synth} is a hyperparameter), I do the following, using helper methods from the `PIL` library.

1. Background sampling: A random image is selected from the MNIST-W set, and a random 128×128 square patch is cropped from it (resizing first using bilinear resampling if the image is too small). This provides an in-distribution background.

2. Digit rendering: A 128×128 canvas is created with p pixels of padding on each side, where $p \sim \mathcal{U}\{10, 30\}$. A random MNIST digit is resized to fit within this canvas. Each pixel's opacity is set to its original grayscale intensity. One random RGB color is selected and used for all these pixels. This produces a semi-transparent colored digit on a transparent canvas. An affine transformation is then applied, with the digit layer being rotated by a random angle sampled from $\mathcal{U}\{-10°, 10°\}$.

3. Final digit creation: Every pixel with a large enough opacity, i.e., alpha-channel value greater than 20 (in the $[0, 255]$ range) is rendered onto the background crop area. This gives a final 128×128 RGB image whose label inherits from the source MNIST digit.

We concatenate our original MNIST-W dataset with our synthetic dataset to produce a 'Combined training dataset' of size $8755 + N_{\text{synth}}$.

Augmentation. The following augmentations are applied to every item in the training dataset, using the `transforms` module in the `TorchVision` package.

[1]While this project was done by me, this specific website was almost entirely generated with the use of a large language model coding assistant to help me examine the quality of my augmented and synthetic data during development. I deployed it to a live site for this submission as a potentially useful but non-essential supplement to my project

Transform	Parameters
Random resized crop	128×128, scale $\in [0.7, 1.0]$, ratio $\in [0.75, 1.33]$
Random rotation	$\theta \sim \mathcal{U}(-20°, 20°)$
Random perspective	distortion scale $= 0.2$, $p = 0.5$
Random affine	translate $\in [0, 0.1]^2$, scale $\in [0.9, 1.1]$
Colour jitter	brightness, contrast, saturation $\in [0.6, 1.4]$; hue $\in [-0.1, 0.1]$
Gaussian blur	kernel 5×5, $\sigma \sim \mathcal{U}(0.1, 2.0)$
Gaussian noise	$\epsilon \sim \mathcal{N}(0, 0.05^2)$ added per pixel
Random erasing	$p = 0.25$, scale $\in [0.02, 0.15]$
Normalisation	$\mu = (0.485, 0.456, 0.406)$, $\sigma = (0.229, 0.224, 0.225)$

Table 1: Training augmentation pipeline. Note that I needed an additional 'transform' convert the image to a PyTorch tensor before the Gaussian noise augmentation, since, up to that point, PyTorch expects a PIL image, and the last two transformations expect a tensor. Images in the validation set are passed through a different transformation pipeline, only being resized to 128×128 and to be normalized to with the same parameters.

The values for μ and σ for the normalization steps are the values in the ImageNet dataset[2].

For every training batch, with probability p_{mix}, I additionally applied the **mixup** augmentation to every image, following [2]. This augmentation produces physically implausible images but implicitly acts as a regularization technique, since linear interpolations of images should correspond to linear interpolations of predicted labels. It replaces the batch with linear interpolations of random image pairs and their labels:

$$x = \lambda x_1 + (1 - \lambda)x_2, \quad y = \lambda y_1 + (1 - \lambda)y_2$$

with $\lambda \sim \beta(0.2)$.

Examples of the data augmentation pipeline, mixup, and synthetic data are presented in Figure 1, Figure 2, and Figure 3.

Dataloader implementation. I subclass the PyTorch 'Dataset' type to create a `ProvidedImageDataset` (MNIST-W), `SyntheticImageDataset` and `CombinedImageDataset`. The first one applies the transformations in Table 1 to an image loaded from a provided path (randomly sampled from the result of the stratified split function) and returns the image and its label. The synthetic dataset samples a random image from the original MNIST (with replacement) and applies the described transformation to produce a new image. The same transformations are applied. The validation dataset is instantiated using the `ProvidedImageDataset` class, but with a different set of paths provided.

A PyTorch `DataLoader` is wrapped around the combined dataset for training and the validation set. A batch size of 64 is set for both. Shuffling is set to true for training and false for validation.

9000 synthetic images were used during the final run and mixup probability was set to 0.2.

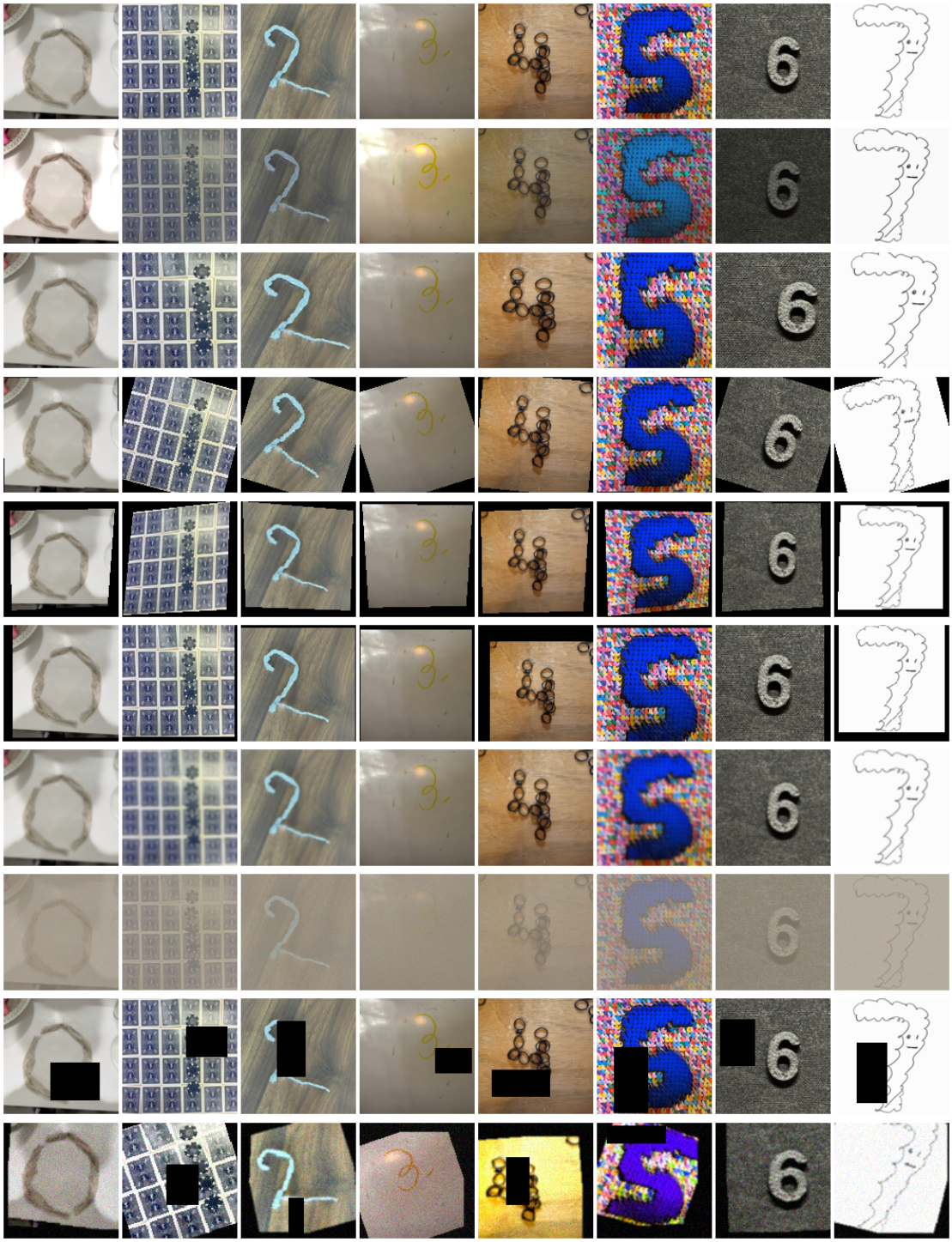
Augmentation analysis. Augmentation wencourages invariance to transformations that do not affect the digit identity, and are particularly likely to be present in MNIST-W test samples. Given that these images were collected by different people in vastly different settings with different cameras, transformations like crops, roations, perspectives, and affine translations help account for differences in setups, and the colour jitter, gaussian blur, and gaussian noise help account for differences in environments. To make the model more robust to adversarial settings, random erasing hopefully helps. Notably, we do not use more typical augmentations like flipping, and use a very small range for rotation, to preserve digit identity (eg. to not inadvertently change a 6 to a 9).

2 Model setup

Model architecture. A ResNet model with **18 layers** is used, exactly following the ResNet-18 setup from [1]. I re-describe the implementation here.

We define a single block of this architecture as being defined by the number of input channels c_{in} and number of output channels c_{out}. The block contains a 2-D convolutional layer, batch normalization, another convolutional layer, batch normalization, and a ReLu activation. The first convolutional layer has c_{in} input channels and c_{out} output channels with stride s, padding 1, and a 3×3 kernel. The second convolutional layer preserves c_{out} channels with

[2]Following advice I found online, it is acceptable to use these values for real-world images, and avoid the cost and instability of recalculating them for every training dataset – which was helpful when I was testing different amounts of synthetic data and augmentations.



(a) Original

(b) Colour Jitter

(c) Random Resized Crop

(d) Rotation

(e) Random Perspective

(f) Random Affine

(g) Gaussian Blur

(h) Gaussian Noise

(i) Random Erasing

(j) Full pipeline

Figure 1: Augmentation transforms applied to example images

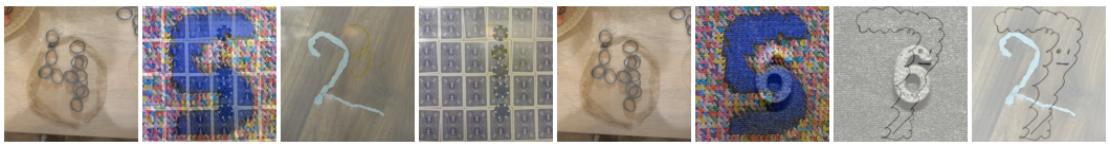


Figure 2: MixUp augmentation.

Figure 3: Synthetic data. Most, but not all, of these are decent samples.

stride 1 and padding 1, also with a 3×3 kernel. This is concatenated with the output of a shortcut function for that layer, which takes in the input to that layers and returns it exactly if $c_{\text{in}} = c_{\text{out}}$, and otherwise uses a convolutional block from c_{in} to c_{out} with stride 1.

ResNet-18 contains a stem, four 'stages', and a final output stage. The stages progressively increase the width from $w = 64$ to 128, 256, and 512. There are 18 layers not counting pooling and batch normalization. I provide a full list of layers below, italicizing non-weighted (i.e. pooling and batch normalization) layers. A padding of 1 is used everywhere unless specified otherwise. 'S<n>' denotes the n-th stage, 'B<i>' denotes the i-th block in a stage.

1. Stem convolution: 64 kernels (7×7, stride 2, padding 3)
2. *Stem batch normalization*
3. *Stem max pooling*: 3×3 with stride 2
4. S1 B1 Conv1: 64 kernels (3×3)
5. *S1 B1 batch normalization*
6. S1 B1 Conv2: 64 kernels (3×3)
7. *S1 B1 batch normalization*
8. S1 B2 Conv1: 64 kernels (3×3)
9. *S1 B2 batch normalization*
10. S1 B2 Conv2: 64 kernels (3×3)
11. *S1 B2 batch normalization*
12. S2 B1 Conv1: 128 kernels (3×3, stride 2)
13. *S2 B1 batch normalization*
14. S2 B1 Conv2: 128 kernels (3×3)
15. *S2 B1 batch normalization*
16. S2 B1 shortcut: 128 kernels (1×1, stride 2, padding 0)
17. *S2 B1 shortcut batch normalization*
18. S2 B2 Conv1: 128 kernels (3×3)
19. *S2 B2 batch normalization*
20. S2 B2 Conv2: 128 kernels (3×3)
21. *S2 B2 batch normalization*
22. S3 B1 Conv1: 256 kernels (3×3, stride 2)
23. *S3 B1 batch normalization*
24. S3 B1 Conv2: 256 kernels (3×3)
25. *S3 B1 batch normalization*
26. S3 B1 shortcut: 256 kernels (1×1, stride 2, padding 0)
27. *S3 B1 shortcut batch normalization*
28. S3 B2 Conv1: 256 kernels (3×3)
29. *S3 B2 batch normalization*
30. S3 B2 Conv2: 256 kernels (3×3)
31. *S3 B2 batch normalization*
32. S4 B1 Conv1: 512 kernels (3×3, stride 2)
33. *S4 B1 batch normalization*
34. S4 B1 Conv2: 512 kernels (3×3)
35. *S4 B1 batch normalization*
36. S4 B1 shortcut: 512 kernels (1×1, stride 2, padding 0)
37. *S4 B1 shortcut batch normalization*
38. S4 B2 Conv1: 512 kernels (3×3)
39. *S4 B2 batch normalization*
40. S4 B2 Conv2: 512 kernels (3×3)
41. *S4 B2 batch normalization*
42. *Global average pooling*
43. Fully connected: $512 \to C$ outputs, where C is the number of classes (10)

Further analyis about the model is in section 3.

Training. The model was trained with cross-entropy loss

$$\sum_i y^{(i)} \log(p_\theta^{(i)}) \tag{1}$$

where $p_\theta(\cdot)$ is the output of the model. Note that when mixup is applied, $y^{(i)}$ is not a one-hot vector, so we need to calculate cross-entropy manually as opposed to using the PyTorch `cross_entropy` utility. When mixup was not applied, the PyTorch cross entropy function was used with label smoothing set to 0.05, which softens the ground-truth target labels to introduce regularization.

During the final training run, the model was trained using the AdamW optimizer with an initial learning rate of 10^{-3} and weight decay set to 10^{-4}. A batch size of 64 was used[3]. The model was trained for 200 epochs with early stopping after 50 epochs of no improvement to validation accuracy. Typically, model performance climbed significantly until the 50th epoch, after which there were small but continuous gains in performance up to the 150th epoch. 200 epochs was therefore set as a reasonable number to try to saturate model performance.

[3]This was reasonable to fit into local memory of my MacBook's MPS device

A cosine learning rate scheduler was used, with 5 warm-up epochs where the learning rate was linearly increased from 0 to $\mathrm{lr}_{\mathrm{base}} = 10^{-3}$, after which it decayed to $\mathrm{lr}_{\mathrm{min}} = 10^{-6} \approx 0$ following the function

$$\mathrm{lr} = \mathrm{lr}_{\mathrm{min}} + (\mathrm{lr}_{\mathrm{base}} - \mathrm{lr}_{\mathrm{min}}) \cdot 0.5(1 + \cos(p_T)) \tag{2}$$

where p_T represents the fraction of completed epochs after warmup.

The training and validation loss and accuracy are presented side-by-side with the learning rate schedule in Figure 4.

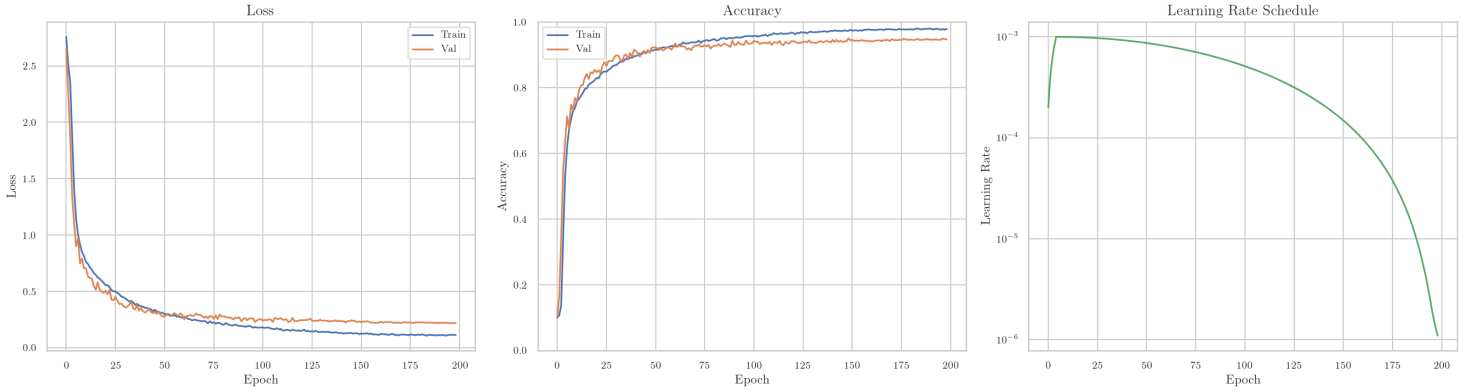


Figure 4: Model performance over training. Note that performance increases rapidly for the first 50 epochs, which is pronounced by the learning rate being relatively high in that period. Very little overfitting is observed, with validation accuracy being slightly better than training accuracy for some time, likely due to the aggressive perturbations applied to training images. The model at epoch 148 had the highest validation accuracy (94.997%) and its weights were restored for all later calculations.

The choice of these parameters was made using a grid search for 50 epochs, the results of which are reported in Table 2. The best choice is a high learning rate, with batch size 32 and 64 performing almost equally well. I chose 64 to speed up training.

	1e-4	3e-4	1e-3
32	0.866	0.909	*0.927*
64	0.8291	0.8772	**0.922**
128	0.7960	0.8661	0.909

Table 2: Grid search for optimal learning rate $\mathrm{lr}_{\mathrm{base}}$ (columns) and batch size (rows), training the model for 50 epochs with each configuration. Reported values are validation accuracy to three significant figures. 1000 synthetic images were used and the same augmentations as in the final pipeline were applied to the training images.

A key result of this study is that the augmentation and synthetic data generation pipeline is crucial for preventing overfitting and boosting performance. A study on the impacts of these components gives Table 3. Augmentations are crucial to prevent overfitting – the run without any augmentations and without any synthetic data achieved 99.94% training accuracy but only 85.12% validation accuracy, and the run without augmentations and with synthetic data achieved 98.15% training accuracy and only 83.43% valuidation accuracy. Note that, having observed the performance boost from adding synethic data, the number of synthetic examples was increased to 9,000 to the final run, which yielded the final reported 95.00% validation accuracy.

	Augmentations	No augmentations
Synthetic data	0.9396	0.8343
No synthetic data	0.9220	0.8512

Table 3: Validation accuracy when ablating over the addition of 6,000 synthetic training examples and including the augmentation pipeline described in section 1.

3 Model architecture

3.1 Interpretation

All model layers are listed in section 2. I include the output of `print(model)` in Appendix A, since it is extremely long and repetitive, and explain what each unique block in the model does and identify key components here.

1. Stem: the convolutional layer expands the image from the 3 RGB channels to 64 channels and uses stride 2 to downsample the image to reduce computational cost, and padding 3 to preserve spatial dimensions with a 7×7 kernel. Batch normalization normalizes activations across the batch for stability and regularization. The ReLU activation is a standard nonlinearity. Max pooling takes important activations for the next steps

2. For each of the next four blocks: the first convolutional layer expands the channel width from w_{old} to w_{current}, batch normalization normalizes features, the second convolutional layer helps the block apply a complex transformation, and batch normalization stabilizes activations again before the final ReLU. The key ResNet feature is the skip connection identity map, which, through the method described in the previous section, helps transmit the input to this block onto the next block. This block is repeated four times to help the model learn progressively more complex features in higher dimensions ($64 \to 128 \to 256 \to 512$).

3. Dropout is applied with probability 0.2 as a regularization technique. Average pooling at the end provides spatial invariance to the location of feature activations and reduces parameter count before the final linear layer, which maps from 512 to the final 10 logits required for classification.

Low-level features like edges and textures are captured in the stem and stage 1, which take in spatial maps with 64 channels. High-level features are captured in stages 3 and 4, which operate in 256 and 512 dimensions on the provided small spatial maps.

3.2 Model capacity analysis

The final model contains 11,181,642 parameters. Since our architecture is split up into stages which grow the width from 64 in stage 1 to 512 in stage 4, stage 4 will be the most expensive and stage 1 the least expensive.

CNN module	Formula	Parameter count
Stem convolution	$3 \times 64 \times 7^2$	$9,408$
Stage 1 Block 1	$2 \cdot (64 \times 64 \times 3^2)$	$73,728$
Stage 1 Block 2	$2 \cdot (64 \times 64 \times 3^2)$	$73,728$
Stage 2 Block 1	$(64 \times 128 \times 3^2) + (128 \times 128 \times 3^2) + (64 \times 128)$	$229,376$
Stage 2 Block 2	$2 \cdot (128 \times 128 \times 3^2)$	$294,912$
Stage 3 Block 1	$(128 \times 256 \times 3^2) + (256 \times 256 \times 3^2) + (128 \times 256)$	$917,504$
Stage 3 Block 2	$2 \cdot (256 \times 256 \times 3^2)$	$1,179,648$
Stage 4 Block 1	$(256 \times 512 \times 3^2) + (512 \times 512 \times 3^2) + (256 \times 512)$	$3,670,016$
Stage 4 Block 2	$2 \cdot (512 \times 512 \times 3^2)$	$4,718,592$
Final FC layer	512×10	$5,120$

Table 4: Parameter count contribution for each module in the Resnet-18 architecture used. Note that we can calculate parameter counts for a convolutional layer as $C_{\text{in}} \times C_{\text{out}} \times k^2$. Stages 2, 3, and 4 have an additional component due to the shortcut function not being the identity. Adding the above numbers up gives 11,172,032; the difference from the total count of 11,181,642 is explained by the batch normalization layers.

It would have been interesting to see what ommitting stage 4 – the most expensive stage – would have done. However, considering that overfitting was not a serious problem with sufficient data augmentations (noticing that there was not much of a train-validation accuracy gap in Figure 4), and progress tended to stall around 94–95%, it is likely that these parameters did not give the model full expressive power, so the model could have reasonably been made larger. Had I had more time and compute power, I would have likely tried extending the model even further, adding more stages – for example, using Resnet-34 as originally proposed by [1].

However, taking into account that the misclassified examples required significant high-level reasoning for me to label, I am not sure if more parameters would help, rather, an architectural change might have been required to achieve closer to perfect performance on the adversarial MNIST dataset.

A Output of `print(model)`

```
ResNet18(
  (stem): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
  (block1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (shortcut): Identity()
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (shortcut): Identity()
    )
  )
  (block2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (shortcut): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (shortcut): Identity()
    )
  )
  (block3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (relu): ReLU(inplace=True)
      (shortcut): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (shortcut): Identity()
    )
  )
  (block4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (shortcut): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (shortcut): Identity()
    )
  )
  (dropout): Identity()
  (avgpool): AdaptiveAvgPool2d(output_size=1)
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

B AI Usage

I used Claude code to assist in the implementation of this project, particularly setting up the full pipeline on top of the basic model and data setup, adding visualization and adding additional studies once the basic pipeline was in place. I started off by providing it a markdown file describing my plan, which I had prepared using Claude (web version). After this point, I used it to iteratively refine and change my codebase. I list my prompts below.

Prompts for Claude web:

1. The data we have is 10k images, but a lot of these are terrible - extremely noisy and, for example, people making shapes with their bodies. I eventually want to make a model that generalizes well to these. The model eventually has to be a hugging face pipeline for inference, but for training, I can do whatever.

   I know I'm allowed to discard the training data however I like, including with eg. fine tuning an off the shelf model. I can also use any architecture I like for my classifier, I just can't use a model with pretrained weights for

final inference. I can additionally add to the dataset, such as using MNIST. I have a 50 Google Colab GPU budget and a Macbook pro M5.

I'm thinking I'll: 1. Fine-tune some kind of pretrained model to help me measure the quality of an image / set of images (they're grouped by setting) (somehow, not sure how - maybe look at raw accuracy?) and discard the ones that are terrible, and maybe assign something like a difficulty score / discrete category to each image/dataset 2. Collect 'nice' data (eg. standard MNIST) and assign difficulties to that. Then adopt a curriculum approach for the model What do you think of this approach? How else would you deal with the fact that I eventually have to train on some of the provided (bad) data and do inference on that data? What specific ways / models / architectures would be suited to th is>?

2. I want to switch to using claude code - how would you recommend transferring the context of this conversation? If there's no easy way, generate a summary with everything it needs to know. I'll give it more specific details on what I want it to set up but it should understand the challenges and ideas we've discussed.

3. Labels are all verified by labellers - I can assume this.

4. Wouldn't data augmentation significantly increase the size of our set? Or is it assumed in these papers already (already taken into account for when considering the limitation of dataset size)? Do you think it'd be a good idea to think about more aggressive perturbations? Something like manually throwing in background parts of random images into the given images to add even more noise (the same idea as cutmix but for classification)? Suggest more augmentation ideas and evaluate the ones you sugested previously. Would it be a good idea to try to introduce something like standard MNIST or would that corrupt the distribution? I'd say about 40% of images are like the ones I'm attaching, which I think an MNIST model should be able to do

5. Yes, update the doc with this strategy. Additionally suggest a prompt to me to tell Claude code what to start doing.

I then heavily edited the provided markdown file - I culled a lot of bad or unnecessary ideas and edited many of the suggested lists of augmentations and hyperparameters. I switched it from using a suggested ConvNext architecture to ResNet architecture, and implemented that in code.

Prompts for Claude code:

1. I'm going to give you context on this project. [Pasted text #1 +61 lines]. Generate a very short paragraph demonstrating your understanding.

2. Write a script to run as much of the pipeline as you can. I'll leave my computer overnight. If there are some parts of the code you're unsure about, write small tests for them and run them right now.

3. List all changes you've made to things like the architecture, pipeline, augmentation, hyperparameters, etc. relative to the original instructions.

4. Let's remove the cutmix augmentation. Let's add a lot of metrics and visualisation. Add confusion matrices and other appropriate metrics. Add logging and plotting for loss if you havne't already. Save loss by class if you can. Also save loss by dataset type, if you can - I understand this might require a major refactor. I want loss histories for these.

5. *The next few prompts are the result of Claude creating a strange integer overflow bug and panicking – I had to intervene and fix it eventually* Why are you monkeypatching? Why not just run it on mps?

6. Finish writing your previous attempt to generate the validation confusion matrix correctly. Make sure you're shufflign the data during the split. Then, write and run the command for running the complete training pipeline, hopefully to get good performance, over the next 7 hours. Consider that we're getting mediocore performance (10% accuracy) on two epochs right now - about worse than chance.

7. use uv and activate .venv if you havent already

8. If the issue is just with the confusion matrix, you can delete the validartion confusion mtraix. If the issue is causing issues with how we're generating the validation data, consider just rewriting that code from scratch after deleting your current code to load the validation data. It's okay to be a little less efficient so long as the behaviour is very well defined.

9. Can you fix my visualize augmentations script? Remember that I changed the augmentations being used a while back.

10. Give me a command that I can run in another terminal session so that I can monitor the results of training.

11. Wait, there's clearly a bug here. Look at the model logs.json file - [Pasted text #1 +320 lines] even after 9 epochs it's just predicting 1 on the validation set / we're miscounting its predictions. This might explain why it loks like accuracy is stlling? In general thought it seems to have a bias towards 1 on all classes

12. Yeah, do that - kill and restart with less regularization

13. After you do this, 1. re-train resnet for more epochs - maybe 200? seemed like performance was still improving at 100 epochs. 2. consider and explain the rationale for re-trzining with our synthetic data or not. Also, add visualizations of misclassified examples.

14. or future runs, can you edit the script such that it always saves examples of misclassified examples, and the top misclassified examples, like you did last time?

15. When we're injecting synthetic data, I still want the entire validation set to only be data from the original dataset - are we doing this? This is to mitigate the risk of the synthetic data not being representative of the original data

16. For later - add visualisations of the original data in 2-D (eg. using PCA) with classes labelled in s catterplot. Then add a visualization with augmented data and synthetic data, with each data source labelled. Additionally, for each class, generate a scatterplot showing synthetic data, augmented data, and original data.

17. Update all plotting to be done using Seaborn with latex font with the deep seaborn palette.

18. Can you make a small web app / dashboard that shows the PCA of the features of these images on a 2-D scatterplot, but hovering over an item on the plot shows the original image? Just make it simple - each plot corrdesponds to one page, where the plots are the exact same as the visualizations you generated earlier - one plot with the original data colored by class, one plot with all three data sources, and one plot for each class colored by source. Try to keep it simple.

19. If the error rate is less than 8% on the validation set (ie val acc is $> 92\%$) can you save all misclassified examples to a folder? Just put a title on them - true class, predicted class, confidence. Organize it into folders by ground truth.

20. Also create a dashboard with all validation examples, with separate tabs for each, which shows the 2-D PCA of features. One color for correct, one color for incorrect. Also generate static PNG files for this. There should be 10 tabs and 10 png files - one for each digit.

21. Make a folder with dashboard.html and val_dashboard.html. Add a small homepage that links to these two with 1 sentence descriptions of each link. Use IBM Plex Mono as a font. I'll then deploy it to Netlify.

22. Re-run the code for visualizing classifications for run3. Make sure it updates everything in the site/ html as well.

23. Oops, sorry, let's actually refactor the site html so that there's a different one for each run. I just want it for run 3 and run 4, but modify the training pipeline so that it's automatically generated next time. Also, use IBM Plex mono for every page in the wbsite, not just the index

24. No, I want there to be a site/folder inside each checkpoint folder, just for that checkpoint. I deleted the root siter directory. Can you generate site/ folders for run3 and run4?

25. Write a script to run the model for 200 epochs with patience 50 without /any/ training data augmentations (so just train on the raw image data), and then with the exact same hyperparameters with training data augmentations.

26. First, refactor the entire codebase to get rid of the progressive resizing idea. It doesn't work and is adding bloat. I think this would allow you to get rid of the construct of phases as well.

27. Update run_augment_comparison with an additional set of experiments - no augmentation + synthetic MNIST, augmentation + synthetic MNIST + real data. This should be a comprehensive ablation. Then give me a command to run this script.

28. add gaussian blur and gaussian noise back to get_train_transform. kill the process for the current run - 13034 is its ID. delete any folders associated with it. re-run these runs with these transformations now included for augmentation.

29. Seems like there's degraded performance in run5_real_noaug relative to runs 2 and 4. I think the only change was the augmentation? They didn't use gaussian blur and noise and did fine. Were there anhy other differences?

30. Oops, sorry, I meant run6_real_aug. That's also worse than runs 2 to 4.

31. Yeah, let's dial it back. Also, run visualize_augmentations with target being the current checkpoint folder in every training run. Will help us track what augmentations we are applying.

32. Let's edit run_augment_comparison.py to use the exact same hyperparameters as run4, ie train.py –data-dir data/dataset –save-dir checkpoints/run4 –model resnet18 –epochs 200 –batch-size 64 –lr 1e-3 –weight-decay 1e-4 –warmup-epochs 5 –label-smoothing 0.05 –drop-path-rate 0.0 –drop-rate 0.2 –mixup-alpha 0.2 –mix-prob 0.3 –no-progressive –img-size 128 –synthetic-n 3000 –patience 40. Kill the current training run for augmentat_comparison and delete all the associated run information. Set patience to 20 epochs, not 40.

33. Looking at what's going on - we're at epoch 134 and have this unusual pattern: [Pasted text #1 +5 lines]. This means that the augmentations are really aggressive / the training data is too far out of distribution of the validation data. I think it's more the augmentations than the synthetic data. Let's scale them down. We currently have [Pasted text #2 +19 lines]. Let's change this to: 1. less aggresive gaussian blur - let's use kernel size 3 instead of 5. let's also delete mixup.

34. Let's add some testing. Grid search for 50 epochs with 1000 synthetic examples over 3 learning rates (currently, we have 1e-3, let's try 3e-4 and 1e-4) as well as over 3 batch size (32, 64, 128).

35. Wait, look at grid_search.log - it didn't do anything. It expects these checkpoints to already exist! I want to run these experiemnts.

36. Yeah, I refactored it - all code is now in src/, checkpoints are at the same level as src, data is at the same level as src.

37. Check if colab training will work

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[2] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.