CMSC 25040: Homework 3 Write-Up

Kameel Khabaz

February 17, 2023

1 Image Classification

For the image segmentation tasks on the MINST and CFAR-10 datasets, I built and train a modifiable convolutional neural network (CNN) model in Pytorch. I first prepared the data using `torchvision`, and prepared a `DataLoader` for the training and testing sets. I used a batch size of 128, which allowed me to parallelize operations on my machine's GPU, as well as using a train/test split of 80%/20%.

I created a single neural net class that consists of a series of unit convolution-based blocks (the model for the MINST dataset had 1 block, and the model for the CFAR-10 dataset had 4 blocks), followed by a flattening and a series of fully connected (FC) layers. Applying ideas from ResNet, I designed each unit block to consist of a smaller convolution block, a residual block, a second convolution block, and a pooling layer:

1. The first convolution block applies a 2D convolution over the input while also tripling the number of channels, and this is followed by batch normalization and a ReLU activation function.

2. The residual block applies two 2D convolutions, each followed by batch normalization and ReLU activation, while keeping the number of channels constant and then adding the result of these convolutions to the input of the residual layer. The addition of the input to the output of several convolutions (the shortcut connection) makes residual networks easier to train and optimize, and I took advantage of this design in my model.

3. Following this residual block, I added a second convolutional block consisting of a 2D convolution (keeping the number of channels constant), followed by batch normalization and ReLU activation.

4. The final convolution block is followed by max pooling (with kernel size of 2 and stride of 2 to decrease the image size in half).

Each 2D convolution had a kernel size of 3, a stride of 1 pixel, and a padding of 1 pixel to keep the size of the output image the same as the input size. Thus, each unit block effectively triples the number of channels in the image while halving the image size. After all of these blocks, I apply a flattening layer to flatten out the image to a linear array.

This is then followed by a series of FC layers, each of which consists of a linear transformation, batch normalization, and ReLU activation. Each layer halves the size of the input, and I add enough layers until I get to the smallest array size that is greater than the number of classes in the model. My last layer is a final linear layer that maps to the output classification (number of output channels equal to number of classes).

When training the model, I used a cross-entropy loss function because I have a multi-class classification problem. I also use an Adam optimizer, an extended version of stochastic gradient descent (SGD) that solves some of SGD's issues with local minima and saddle points. An Adam optimizer includes first/second momentum terms for the gradient, a bias correction term, and a AdaGrad/RMSProp (an extension of gradient descent that scales the step size based on the historical sum of squares of the gradient in each dimension). I used a learning rate of 0.002, which is double the default value, and set $\beta_1, \beta_2 = (0.9, 0.999)$, the default value.

I calculated the model loss and accuracy for each epoch in the training phase. To prevent unnecessary training cycles and potential overfitting (in which training loss decreases but testing accuracy also decreases), I implemented an early stopping criterion stating that if the validation loss at a step is larger than 1.1 times the minimum validation loss for five time steps in a row, then model training is stopped. This stops training if the validation loss significantly increases, while still allowing a grace period of 5 epochs to allow for the possibility of a model temporarily deteriorating only to later improve.

My results are shown in Figure 1 for the MINST classification model and in Figure 2 for the CFAR-10 classification model. For the MINST model, I had a final testing accuracy of 98.80%, and for the CFAR-10 model, I had a final testing accuracy of 79.82%. The results show that, as expected, training loss generally monotonically decreases (with training accuracy monotonically increasing). However, testing loss initially decreases and then stabilizes (eventually increasing a bit), indicating that model has stabilized and is beginning to overfit. Similarly, testing accuracy increases and then stabilizes. As the MINST dataset is extremely simple, my model almost immediately achieves an extremely high testing accuracy of 98%, explaining why I only needed one unit block in my neural network. On the other hand, the CFAR-10 dataset is more challenging, which is why I needed a larger number of layers (4 blocks instead of 1) in my network and more epochs in my training process to achieve a satisfactory testing accuracy of almost 80%.

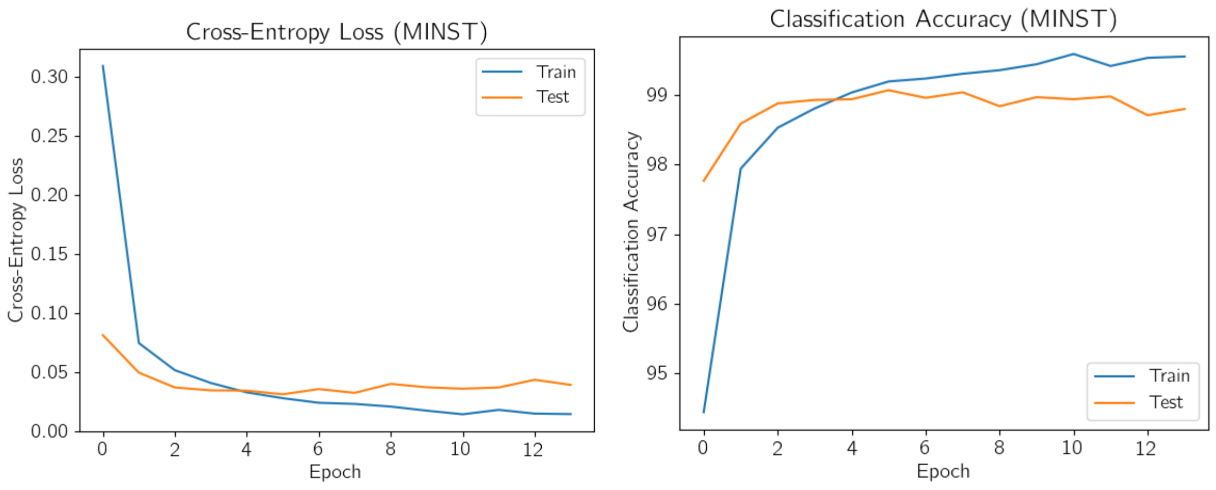

Figure 1: MINST Image Classification Model Training Results.

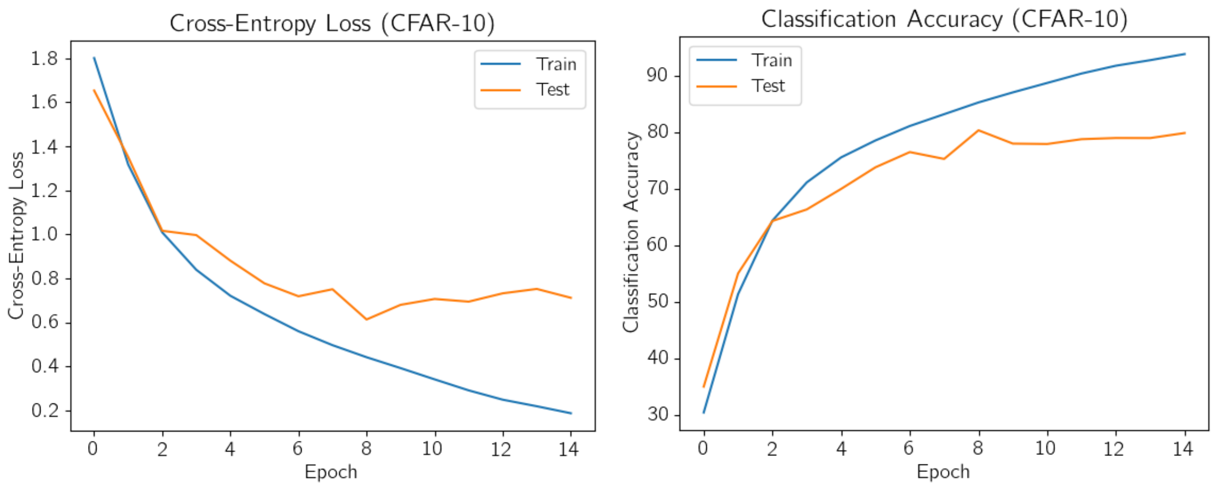

Figure 2: CFAR-10 Image Classification Model Training Results.

2 Image Segmentation

I created and trained an image segmentation neural network to perform pixel-wise classification on a synthetic dataset of images. Each image contains a MINST sample placed on top of a CIFAR-10 sample. The classification model predicts the class of each pixel, which corresponds to the MINST class if the pixel is within the MINST sample or the CIFAR-10 class if the pixel is within the CIFAR-10 sample. While each output pixel can have twenty possible values, in reality, the overall image has only two distinct numbers (all of the image pixels are either a MINST class or a CFAR-10 class).

I designed my network to take advantage of the inherent structure of the problem (in that there are two different types of pixels, CIFAR-10 and MINST, and ten possible classes within each type). The model is inspired from the U-Net architecture in that it consists of an encoder block that transforms each input image into a lower-dimensional feature map and a decoder block that transforms the feature map into an output classification equal in size to the image. Like U-Net, I also have residual layers (skip connections) that take outputs from convolutions at several steps in the encoder and add the result as input to the corresponding steps in the decoder. Unlike U-Net, I change the decoder portion of the network to take advantage of my data structure.

As with image classification, I used a batch size of 128 for my data loader and used the given training/testing loaders. As stated, my model consists of an encoder section and a decoder section. The encoder section contains a series of encoder blocks that each increase the number of channels in the model while halving the image size. Specifically, I have 4 encoder blocks with the following channel mappings: 3 to 32 channels, 32 to 64 channels, 64 to 128 channels, and 128 to 256 channels. Each encoder block consists of the following layers:

- 2D convolution that increases the number of channels.
- 2D batch normalization
- ReLU activation
- 2D convolution that keeps the number of channels constant.
- 2D batch normalization
- ReLU activation
- 2D max pooling with a kernel size of 2 and a stride of 2 that decreases the image size by a factor of 2.

As in the segmentation model, each 2D convolution has a kernel size of 3, a stride of 1, and a padding of 1 to keep the image size constant.

After each encoder block, the output is stored in an array to be later reintroduced into the model in the decoder section.

The decoder section of the model consists of two separate decoder paths, one of which decreases the number of channels to 2 (to determine the dataset that the pixel belongs two) and one of which decreases the number of channels to 10 (to determine the pixel classification within a specific dataset). These two outputs are then combined to produce a final segmentation. After the decoder section, the output image size equals the input image size of the entire network, as expected. I used two separate decoder paths because having the flexibility to separate the network for the two pixel-wise sub-classification tasks gave me around a $3-4\%$ test accuracy improvement over using a single decoder section and only separating the two output layers into the 2-channel and 10-channel predictions at the very end.

Each decoder section consists of a series of decoder blocks that decrease the number of channels while increasing the image size (the reverse of the encoder section). The first decoder path has the following channel mappings: 256 to 128 channels, 128 to 64 channels, 64 to 32 channels, and 32 to 2 channels. The second decoder path has the following channel mappings: 256 to 128 channels, 128 to 64 channels, 64 to 32 channels, 32 to 10 channels. Notice how these channel mappings are the reverse of the mappings in the encoder section, such that outputs from the encoder section can become skip connections that form inputs into corresponding layers in the decoder section. Each decoder block consists of the following layers:

- 2D convolution that increases the number of output channels. As before, it uses a kernel size of 3, a stride of 1, and a padding of 1.
- 2D batch normalization
- ReLU activation
- 2D transposed convolution (`ConvTranspose2d`) that doubles the image size while keeping the number of channels constant. This uses a kernel of 2 and a stride of 2.
- 2D batch normalization
- ReLU activation
- 2D convolution that keeps the number of output channels constant. As before, it uses a kernel size of 3, a stride of 1, and a padding of 1.
- 2D batch normalization
- ReLU activation

The direct outputs of the encoder section become the inputs for the first decoder block in the two decoder paths. For subsequent decoder blocks, the same-sized output from the encoder section is added to the output from the previous decoder block to become the input for the next decoder block (this is done for each of the two decoder paths). After the end of the decoder sections, the two outputs (`x_type` and `x_class`) are combined into a single matrix with pixel-wise 20-class classification (the segmentation). To do this, I perform element-wise multiplication of the first channel in `x_type` with all of the channels in `x_class`, and I perform element-wise multiplication of the second channel in `x_type` with all of the channels in `x_class`. I then concatenate the two results to get the final output. As before, I used an Adam optimizer with a learning rate of 0.002 (double the default value) and $\beta_1, \beta_2 = (0.9, 0.999)$ (the default value).

With this model, I obtained a final testing accuracy of 74.24% after 100 epochs of training. The accuracy is calculated pixel-wise, meaning that it is the fraction of pixels in each image for which my model predicted the correct class (out of a total of 20 classes from the MINST and CIFAR-10 datasets). As previously described, each image as a whole contains only two unique classes (one from each dataset), which is why I structured my network in this way to take advantage of the data's underlying structure.

Figure 3 shows the loss and accuracy results. The model exhibits a large initial increase in the testing accuracy to around 60%; however, further increases in accuracy follow the principle of diminishing returns. Subsequent epochs result in smaller and smaller increases in testing accuracy until the final value of 74.24%. Furthermore, the graph of the cross-entropy loss function shows some sign of overfitting as the testing loss increases after around 10 epochs even though the training loss continues to decrease. However, the testing accuracy does follow an increasing trend for the duration of training, so this may not be as significant of an issue.

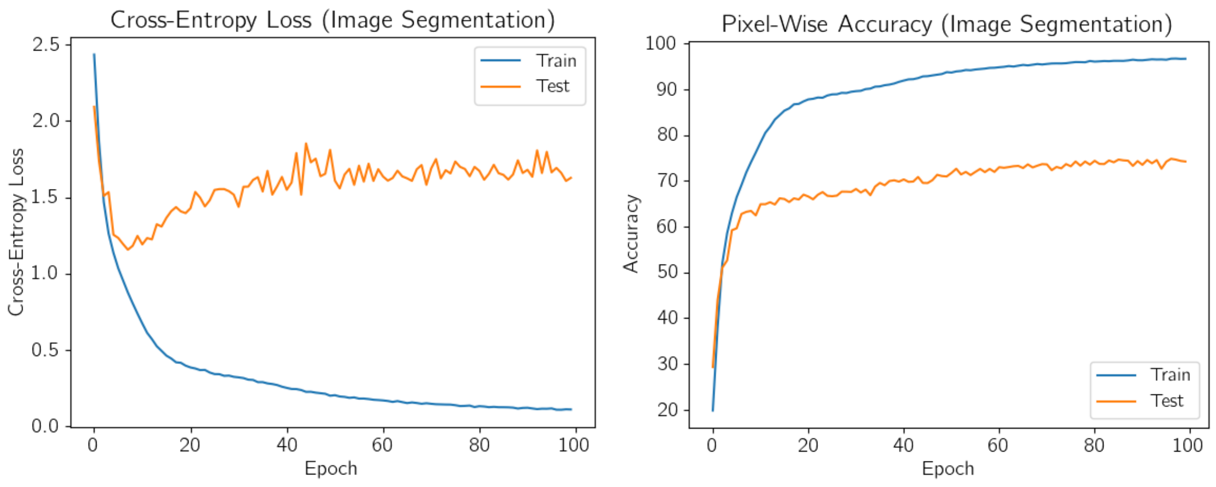

Figure 3: Image Segmentation Model Training Results.