Counting Cells in Microscopy Images (2024)

Hadley Dixon

May 16, 2024

Github Repository

Project Overview

This project uses machine learning to develop an algorithm which is capable of predicting how many cells appear in a given microscopy image.

I implement a U-Net in PyTorch, trained using the data provided by Kaggle, and use my trained U-Net to compete in the Kaggle competition. The objective of this competition is to score as low of a final Kaggle score as possible, with a minimum threshold below 3.

Read in-depth about this Kaggle competition here. Data for this competition was collected by the University of San Francisco.

Evaluation Metrics

The evaluation metric for this competition is mean absolute error.

The final submission should be a .csv file with two columns: “index” and “count”. Each index is an integer between 0 and 1999 which specifies a particular test image. The “count” column contains your predictions for the number of cells which appear in each test image.

Background

The architecture of this U-Net model was developed based on the paper, “U-Net: Convolutional Networks for Biomedical Image Segmentation” by Ronneberger, Fischer, and Brox[1].

[1] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation.” arXiv.org, May 18, 2015. https://arxiv.org/abs/1505.04597.

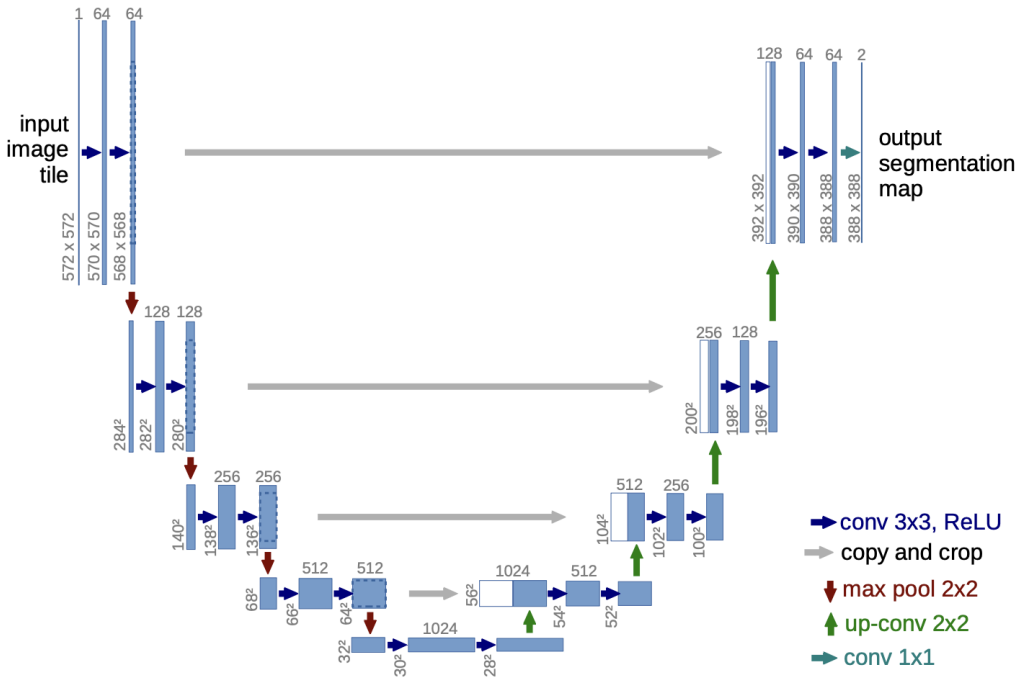


Fig. 1. U-Net Architecture

U-Net models are a convolutional neural network that was developed for biomedical image segmentation. U-Net architecture is modified and extended to work with fewer training images and to yield more precise segmentation. This is done by replacing pooling operations with upsampling operators, which increase the resolution of the output.

The contracting path of the model is a typical convolutional network that consists of repeated application of convolutions, each followed by a rectified linear unit (*ReLU*) and a max pooling operation. The expansive pathway combines the feature and spacial information through a sequence of up-convolutions and concatenations with high-resolution features from the contracting path.

Now, use of U-Net models extends past the medical field and is commonly employed in diffusion models for iterative image denoising. Notably, this technology underlies many modern image generation models, such as DALL-E, Midjourney, and Stable Diffusion.[2]

[2] Wikipedia contributors, "U-Net," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=U-Net&oldid=1223184075 (accessed May 15, 2024).

Data

The training data consists of 2000 microscopy images of cells, stored in a numpy array *X*, and corresponding segmentation masks stored in a numpy array y. Each image (and each corresponding mask) has shape *(128, 128)*. The segmentation mask *y[i]* for image *X[i]* contains "soft labels" (that is, numbers between 0 and 1) which tell you which pixels in the image *X[i]* belong to a cell nucleus.

The test data consists of 2000 more images of cells. No masks are provided for the test images. Your goal is to count how many cells appear in each test image.

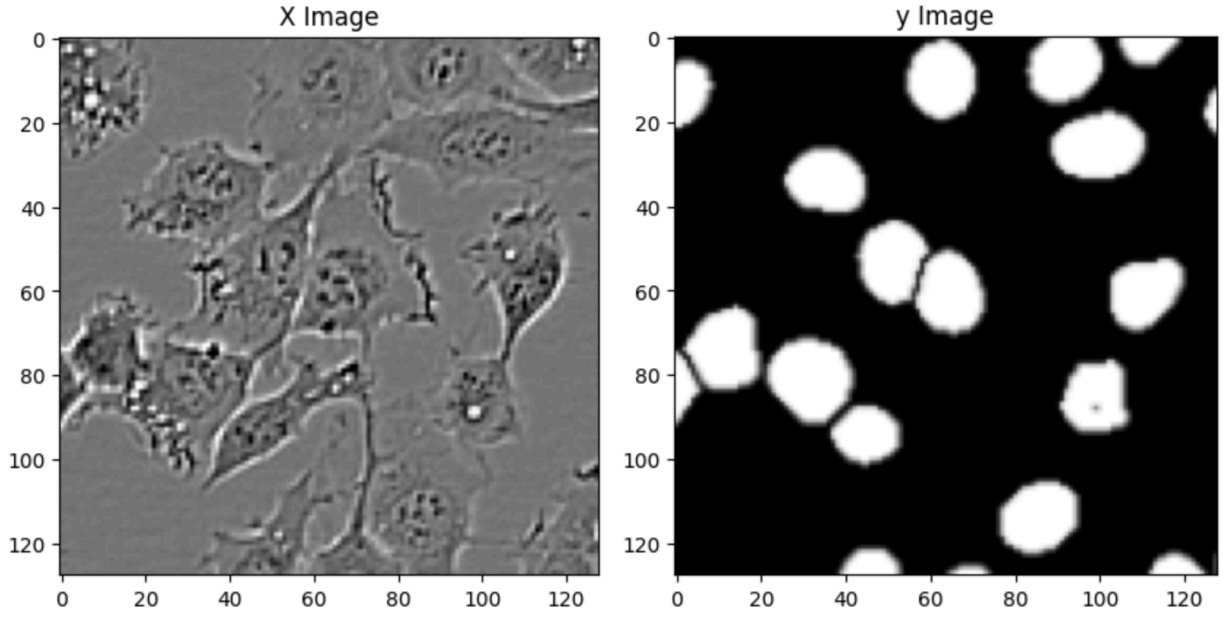


Fig. 2. Sample Image With Superimposed Labels

The training data and test images are stored in numpy's *.npz* file format.

Analysis

Data Preparation

After inputting the needed packages, I mounted my Google Drive and loaded the data as a numpy *.npz* file. To prepare the data, the images are randomly divided into training and validation sets, under an 80/20 split. Each image x_i has a corresponding mask y_i, shown in *Fig. 2*. above. The images are converted to grayscale.

PyTorch works with tensors, so preprocessing is required to handle the data correctly. Two dataset classes are defined to build dataset objects from the training, validation, and testing sets. Within these classes, three methods exist: *__init__, __len__, and __getitem__*. These initialize the dataset object and allow the user to call both the size of a dataset and grab an individual image. Data normalization and type conversion occurs within the *__getitem__* method, outputting a 32-bit floating point tensor.

Dataset objects and data loaders are created for training, validation, and testing sets Instances of the *'CellDataset'* and *'CellDataset_Test'* classes encapsulate the features X and labels y for the respective datasets, while data loaders handle batching and shuffling.

Architecture

The final network architecture modeled the diagram outlined in *Fig. 1.* I created a *'UNet'* class which contained two methods: *__init__* and *forward*. In the initializer, all 23 convolutional layers were defined, corresponding with the number of kernels from Ronneberger, Fischer, and Brox's paper, with a kernel size of 3x3.

During the encoding process, each convolutional layer progressively reduces the spatial dimensions of the input, while increasing the number of feature maps. Specifically, the first two convolutional layers take an input image with one channel and produce feature maps with 64 channels. The next two convolutional layers increase the number of channels to 128. This pattern repeats, doubling the number of channels after every two convolutional layers, reaching 512 channels.

After each convolutional layer, a ReLU activation function is applied to introduce non-linearity.

Max pooling layers with a kernel size of 2x2 and a stride of 2 are used to downsample the feature maps after every 2 convolutional layers. Before each max poling operation, the output is copied, which will be used later in the decoding path.

At the base layer, two convolutional layers with 1024 channels each process the compressed feature maps. After this bottleneck, the feature maps are upsampled by a factor of 2.

During the decoding process, the encoder structure is reflected. The feature maps are upsampled, and then concatenated with the corresponding copied feature maps from the encoder via skip connections. The concatenated feature maps undergo a series of

convolutional operations to reduce the number of channels back to 64. Specifically, after upsampling and concatenation, the combined feature maps are processed with two convolutional layers. This pattern repeats, reducing the number of channels until the output has the same number of channels as the input image.

The final convolutional layer produces an output with the same dimensions as the input image and one channel.

Training

In training my U-Net model, several steps ensue: GPU configuration, model initialization, an Adam optimizer with varying learning rates, a loss function (Binary Cross-Entropy with Logits Loss), and a training loop over varying epochs.

The training process spanned 15 Kaggle submissions, as well as two initial failed submissions.

The training phase first loops through the training data in batches of size 16. For each batch, input data and target labels are moved to the GPU. PyTorch performs a forward pass to get the model's predictions before a loss is computed between the model's predictions and the true labels. The gradient is wiped to 0, and PyTorch performs a backward pass to calculate the gradients, before updating the model's parameters using the optimizer.

The evaluation phase begins by disabling the gradient calculation. The model's performance is evaluated on both the training and validation data by computing the average cross-entropy loss, storing these values for later analysis.

For each epoch, the average cross-entropy loss for both the training and validation datasets are monitored through print statements, and aid in identifying if the model is overfitting or underfitting.

Results

The hyperparameters which yielded the most accurate results were a learning rate of 0.0002, over 45 epochs. This yielded a Kaggle score of 2.60500, and marked my 15th and final submission.

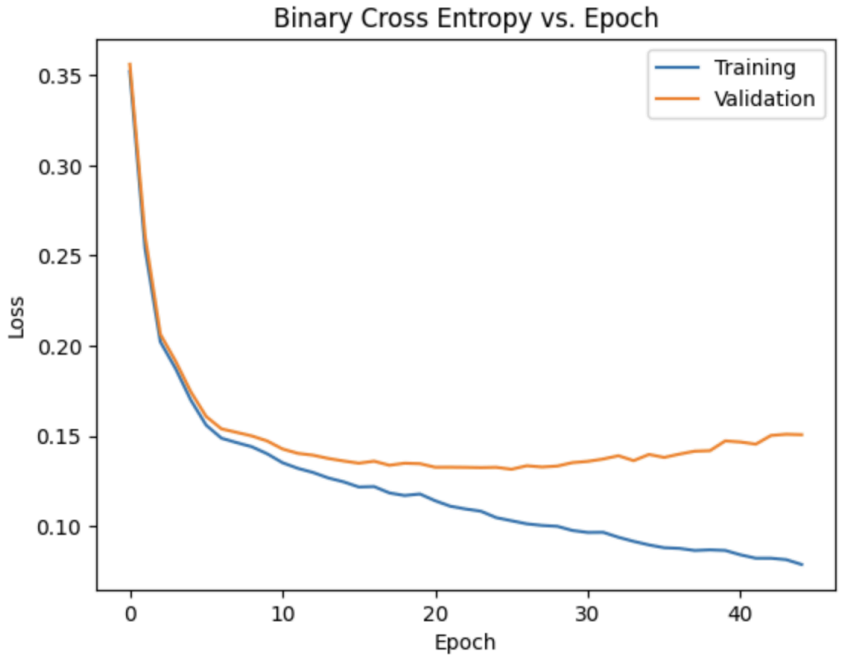


Fig. 3. Final Model Performance on Testing Data, 0.0002 learning rate

The final testing process counts the distinct connected components in each predicted image, using the testing data. To do this, the outputs generated by the model must be converted to probabilities using the sigmoid function, then rounded to obtain binary predictions. Each image is processed manually, moved back first to the CPU, detached from the computational graph, converted to a numpy array, and finally into an *unit8* data type for compatibility with the '*connectedComponents'* function. This function finds the number of connected components in the binary image.

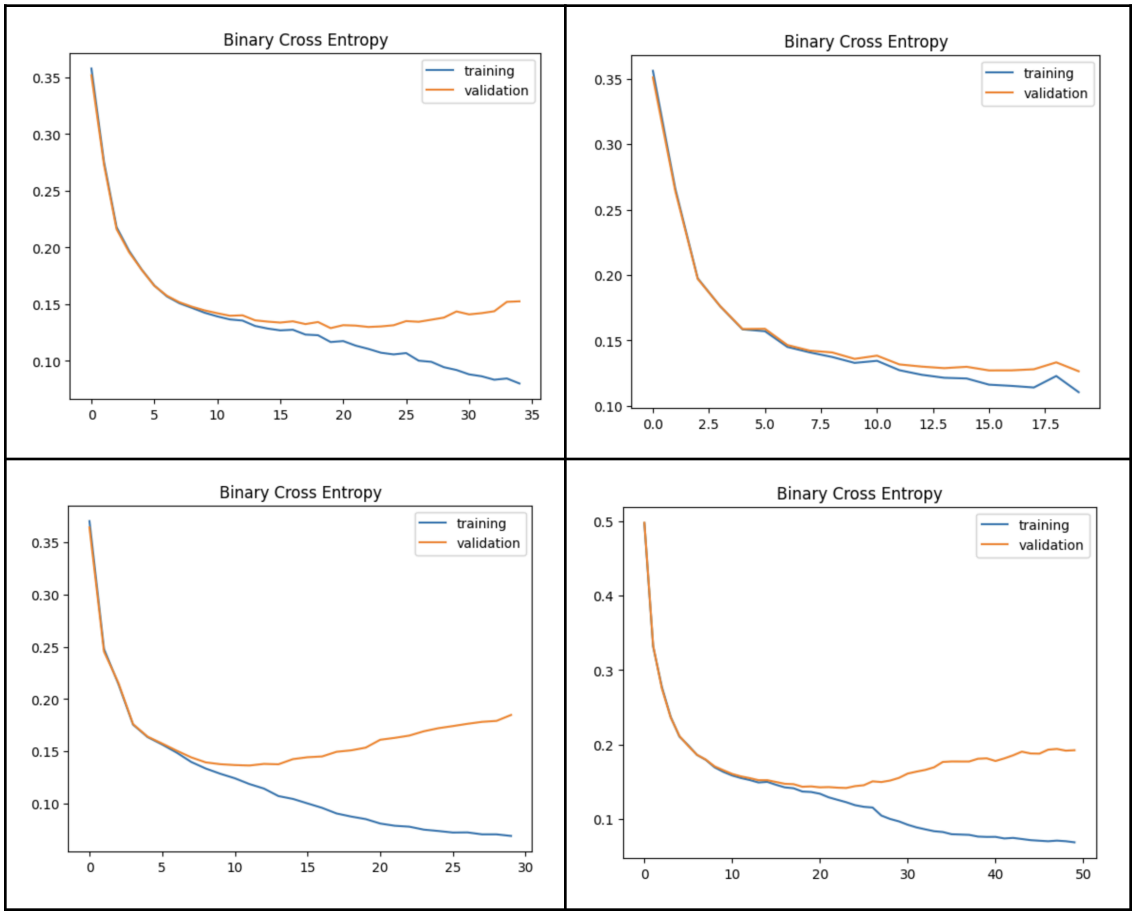
An important choice here is that after advisement from my professor, I refrained from subtracting 1 off of the connected components count. Traditionally, this function includes the background in its count of connected components, and therefore it is customary to subtract 1, so that strictly the number of cells are counted. However, upon refining my model, I found that by including the background in my count, my predictions were more accurate and my Kaggle score lowered.

Discussion

My first 2 failed submissions were due to improper submission format. Instead of a *'count'* column of the number of cells in an image, each row in the *'count'* column instead had an array of all the counts.

After I corrected this mistake, my first scored submission used the standard U-Net architecture, over 10 epochs with a learning rate of 0.0001. This earned a Kaggle score of 12.24000. I was able to improve my score significantly by simply increasing the number of epochs from 10 to 30, hovering around a score of 9. However, I encountered the main challenge of this project: I could not break 9. In the process of fine-tuning my model, it seemed that no matter what I changed, my model's predictive power did not budge. I implemented several changes:

i. Learning rate: 0.0001, 0.00015, 0.0002, 0.0005
ii. Epochs: 10, 15, 30, 40, 45, 50, 60, 100
iii. Altered U-Net architecture
iv. Different methodology for Up-Convolution



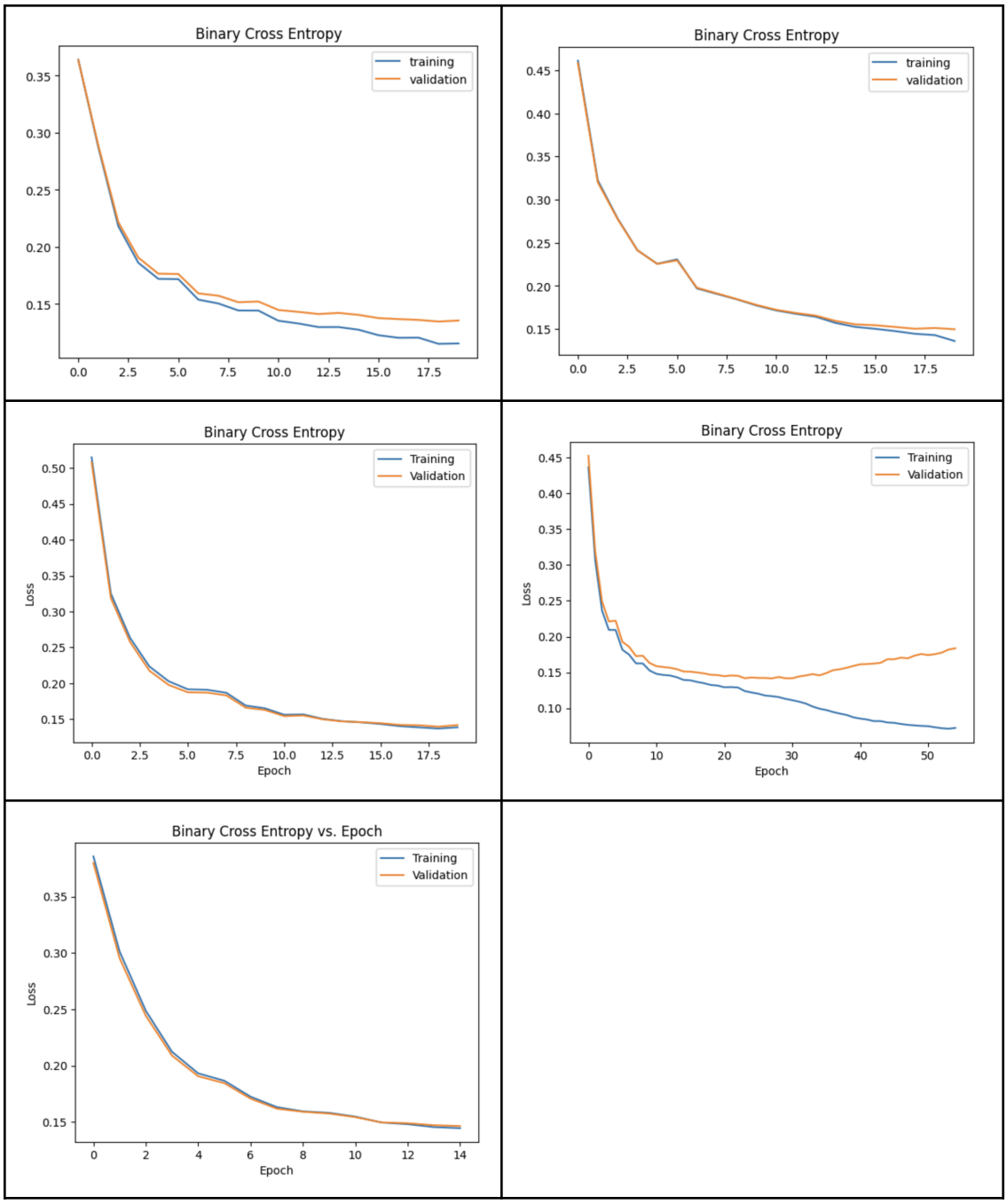


Fig. 4. Convergence Plots, Submissions 2-10

Despite my attempts, submissions 2 through 10 did not improve. Not only that, but there seemed to be little to no change in Kaggle scoring, meaning that my changes were not affecting the Mean Absolute Error of my model. In this process, I had exhausted the GPU limits of all three of my Google accounts, and sought advice from my professor.

Per my professor's suggestion, after looking at my code and also failing to identify the error, I inserted each section of my code (load data, dataset classes, architecture, training loop, test evaluation) into my classmate's properly functioning code, and ran theirs repeatedly until it stopped predicting as accurately. This allowed me to identify which component of my project was causing affecting my performance. Doing so, I established that the way I was using the 'connectedComponents' function improperly. I realized this topic was from the lecture I had been absent from, and in lieu of this explanation, was only interpreting its use from a picture on the whiteboard.

After correction, my Kaggle score immediately dropped to 5.43500. From here on, the changes I attempted to make earlier finally began improving the performance of my model. I continued to experiment, trying different learning rates and epochs from the ranges above, finding that the optimal number of epochs for my model was 45 before my model began overfitting (accompanied by a learning rate of 0.0002). Additionally, I found that removing a downward block from my architecture, from 4 to 3, did not improve the performance of my model. As I approached my final submission, the two changes that influenced my score most significantly were as follows:

1. Refraining from subtracting 1 from the count generated by the '*connectedComponents'* function, effectively leaving the background as a "cell count"
2. Manually performing up-convolution instead of calling *'torch.nn.ConvTranspose2d()'*. Instead, I implemented '*torch.nn.Upsamle()*', followed by *'torch.nn.Conv2d()'* and 'torch.nn.ReLU()'