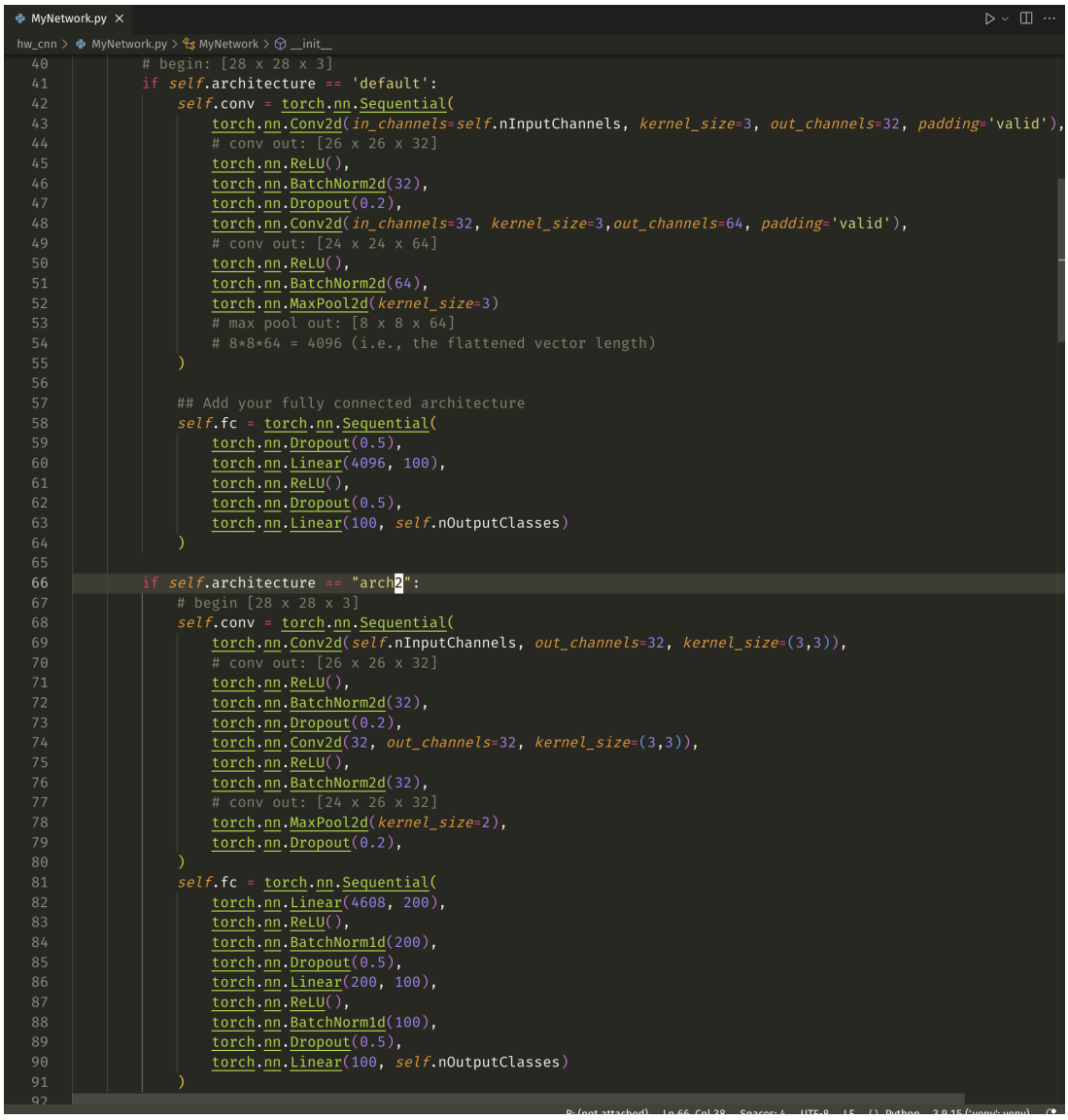
1. Design your own architecture and justify its components (width, depth, convolutions, activations and any other operations)



```python
# begin: [28 x 28 x 3]
if self.architecture == 'default':
    self.conv = torch.nn.Sequential(
        torch.nn.Conv2d(in_channels=self.nInputChannels, kernel_size=3, out_channels=32, padding='valid'),
        # conv out: [26 x 26 x 32]
        torch.nn.ReLU(),
        torch.nn.BatchNorm2d(32),
        torch.nn.Dropout(0.2),
        torch.nn.Conv2d(in_channels=32, kernel_size=3,out_channels=64, padding='valid'),
        # conv out: [24 x 24 x 64]
        torch.nn.ReLU(),
        torch.nn.BatchNorm2d(64),
        torch.nn.MaxPool2d(kernel_size=3)
        # max pool out: [8 x 8 x 64]
        # 8*8*64 = 4096 (i.e., the flattened vector length)
    )

    ## Add your fully connected architecture
    self.fc = torch.nn.Sequential(
        torch.nn.Dropout(0.5),
        torch.nn.Linear(4096, 100),
        torch.nn.ReLU(),
        torch.nn.Dropout(0.5),
        torch.nn.Linear(100, self.nOutputClasses)
    )

if self.architecture == "arch2":
    # begin [28 x 28 x 3]
    self.conv = torch.nn.Sequential(
        torch.nn.Conv2d(self.nInputChannels, out_channels=32, kernel_size=(3,3)),
        # conv out: [26 x 26 x 32]
        torch.nn.ReLU(),
        torch.nn.BatchNorm2d(32),
        torch.nn.Dropout(0.2),
        torch.nn.Conv2d(32, out_channels=32, kernel_size=(3,3)),
        torch.nn.ReLU(),
        torch.nn.BatchNorm2d(32),
        # conv out: [24 x 26 x 32]
        torch.nn.MaxPool2d(kernel_size=2),
        torch.nn.Dropout(0.2),
    )
    self.fc = torch.nn.Sequential(
        torch.nn.Linear(4608, 200),
        torch.nn.ReLU(),
        torch.nn.BatchNorm1d(200),
        torch.nn.Dropout(0.5),
        torch.nn.Linear(200, 100),
        torch.nn.ReLU(),
        torch.nn.BatchNorm1d(100),
        torch.nn.Dropout(0.5),
        torch.nn.Linear(100, self.nOutputClasses)
    )
```

Fig 1: Model architectures. Architecture 1 (**Arch1**) is the top-half of code and architecture 2 (**Arch2**) is the bottom-half of code.

The **two architectures share** common themes of using **batch normalization** to control the scale of input data to a neuron across batches, and using **dropouts** to prevent overfitting. The **main differences** between the architectures is the **dimensions of the feature maps** and the **pooling kernel shape** used for dimensionality reduction at the end of the convolutional layer, and **no batch normalization in the fully-connected layer of Arch1**.

The choice of depth, channels, convolutions, and activations was chosen by following similar architectures found from similar examples in literature (e.g., https://link.springer.com/chapter/10.1007/978-3-030-76773-0_15#Tab4). However, model architectures can be tweaked extensively making it hard to conclude what the best approach is without any additional knowledge. Given more time, it would be worth training on randomized values of output features and tweaking the model architecture to further minimize loss on the validation set during training.

2. Justify your choices of loss function and optimizer
3. Provide the curves (preferably from Tensorboard) with the evolution of the training and validation losses and accuracies during training

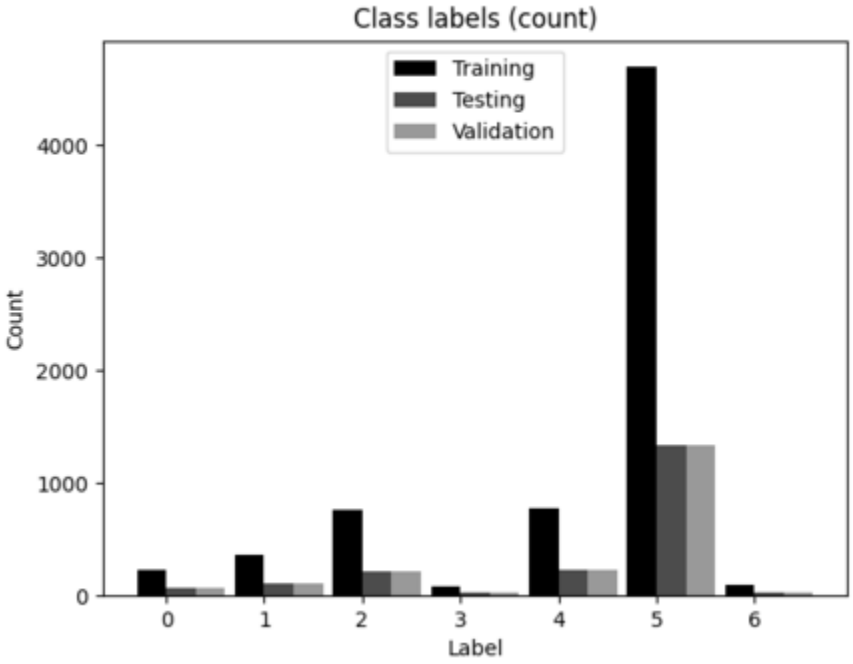


Fig 3: Class balance for MedMNIST dataset

A **categorical cross entropy loss function** was used since this function can penalize multiple classes for a single label. **Class weights** were assigned as $w_k = (1 - N_k/N)$ where N_k is the number of samples for a given kth class. Ultimately, this will weigh classes by the complement of their frequency in the dataset such that a highly represented class (e.g., class 5 [**Fig 3**]) will have a lesser effect on the loss function to prevent optimization of the highly represented class.

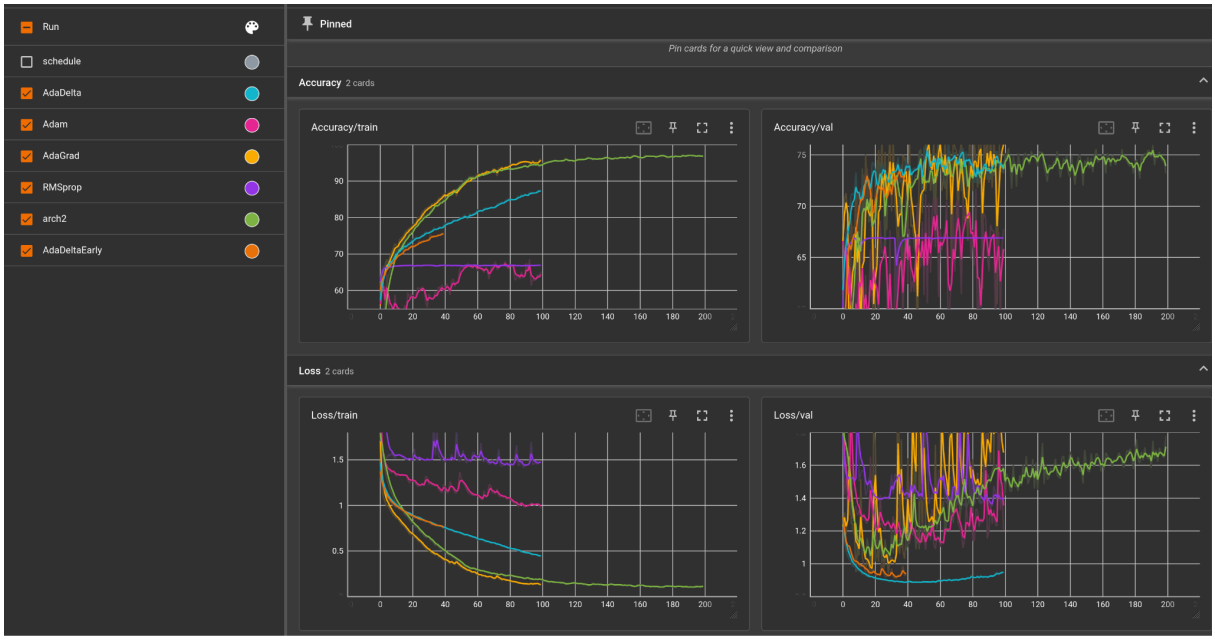
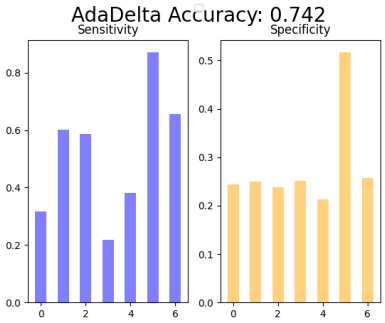
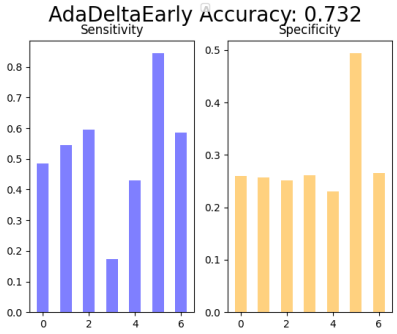


Fig 4: Loss/accuracy for various optimization functions and architectures. AdaDelta, Adam, AdaGrad, and RMSprop optimizers used during training with **Arch1**. The **Arch2** model used the AdaDelta optimizer.

The **AdaDelta** optimizer was chosen since it had the lowest loss in the validation set compared to other optimization functions.

4. Evaluate the performance on the test dataset by providing the overall accuracy of the model, and the individual sensitivity and specificity (and any other metrics that you may consider relevant) on every class. Interpret your results.





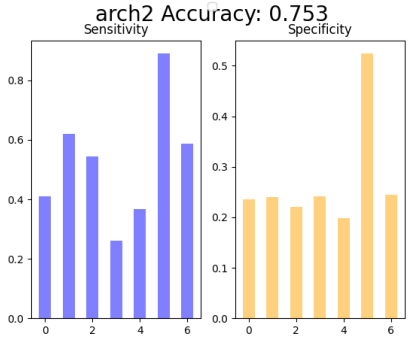


Fig 5 model performance on test. The **accuracy, sensitivity, and specificity** is reported for 3 models AdaDelta, AdaDelta with an early stop at 40 epochs, and Arch2 (from Fig1).

The most performant model in terms of accuracy was architecture 2, and the possible reasoning for the slight performance boost of Arch2 could be one of the following aspects that make it different from the other models: **batch normalization on the fully connected layer**, **different feature map size**, and/or **the increased number of epochs** (Fig 4).