CMSC 422-0201 FINAL PROJECT REPORT

Motivation

A lot of countries still perform manual waste segregation. For humans, this can be:

1. Time-consuming
2. Hazardous to health
3. An unscalable solution for a world that is becoming increasingly consumerist.

Proposal

I propose a machine learning model that will look at images of waste and classify them into the required bins based on the visual features.

Dataset

The dataset is derived from https://www.kaggle.com/datasets/asdasdasasdas/garbage-classification. The following table shows the list of classes and the number of images in each class.

CLASSES	NO. OF EXAMPLES	% OF DATASET
Trash	137	5.02
Cardboard	403	15.94
Metal	410	16.22
Plastic	482	19.07
Glass	501	19.82
Paper	594	23.50

Class Imbalance

Looking at the number of examples in each class, it is obvious that the trash has significantly lesser examples than the rest of the classes. According to https://developers.google.com/machine-learning/data-prep/construct/sampling-splitting/imbalanced-data there are two ways to go from here.

1. Train on true distribution and see if model generalizes well.
2. If not, try downsampling and upweighting.

Preprocessing the images

The images in the dataset are of the size 512x384. Some transformations we do on the images before feeding it to the model are:

1. Resize the images to 224x224.
2. Convert the images to tensors.
3. Normalize the images.

```
transform = transforms.Compose(
    [
        torchvision.transforms.Resize(size=(224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
    ]
)
```

We perform normalization because:

Normalization helps get data within a range and reduces the skewness which helps learn faster and better. In the process of training our network, we're going to be multiplying (weights) and adding to (biases) these initial inputs to cause activations that we then backpropogate with the gradients to train the model. We'd like in this process for each feature to have a similar range so that our gradients don't go out of control.

We use the values highlighted in the image, because those are the values used to normalize the images in the ImageNet dataset - The images in our dataset are not color filtered, contrast adjusted, uncommon lighting and do not contain an "un-natural subject" (medical images, satellite imagery, hand drawings, etc.) just like in ImageNet.

Hyperparameter overview

- Epochs: 10
- Learning Rate: 0.001
- Batch size: 4
- Loss function: Cross Entropy Loss
- Optimizer: SGD
- Momentum: 0.9

Base model

As a starting point, we will use the Convolutional Neural Network defined here: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html. We will use this model to iteratively build upon and improve our accuracy.

```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)

        self.fc1 = nn.Linear(128 * 56 * 56, 1296)
        self.fc2 = nn.Linear(1296, 216)
        self.fc3 = nn.Linear(216, 6)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Accuracy: 60%

F1-Score: 0.60201

Improved Model (ImpNet)

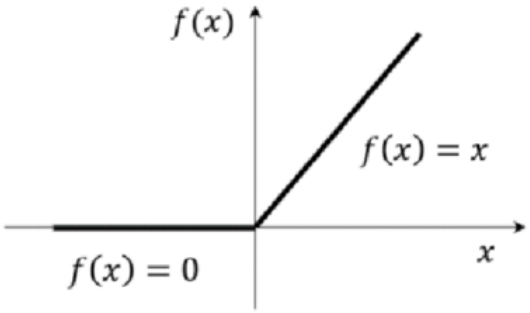
To this model, I used the LeakyReLU function instead of the ReLU function on the activation layers. In addition to this, I added the argument **inplace = True** to save on memory and training time. More information on this can be found here: https://github.com/pytorch/vision/issues/807

```python
class ImpNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(128 * 56 * 56, 1296)
        self.fc2 = nn.Linear(1296, 216)
        self.fc3 = nn.Linear(216, 6)

    def forward(self, x):

        x = self.pool(F.leaky_relu(self.conv1(x), inplace=True))
        x = self.pool(F.leaky_relu(self.conv2(x), inplace=True))

        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x), inplace=True)
        x = F.relu(self.fc2(x), inplace=True)
        x = self.fc3(x)
        return x
```

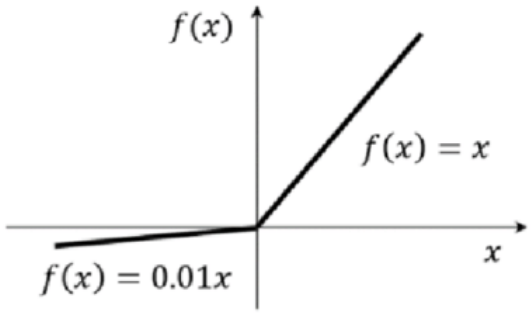
Accuracy : 67%

F1-Score: 0.67289

Conclusion:



ReLU activation function



LeakyReLU activation function

1. ReLU sets all negative values to zero, effectively eliminating any negative information from the input.
2. Leaky ReLU introduces a small negative slope for negative inputs, allowing the network to capture and propagate negative information.

Dropout on linear layers only (DropNet)

We add dropout regularization on the linear layers of this model. The value we use for the dropout is 0.5 which is based on the recommendation from this paper: [Improving neural networks by preventing co-adaptation of feature detectors](). .

```
class DropNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(128 * 56 * 56, 1296)
        self.dropout1 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(1296, 216)
        self.dropout2 = nn.Dropout(0.5)
        self.fc3 = nn.Linear(216, 6)

    def forward(self, x):
        x = self.pool(F.leaky_relu(self.conv1(x), inplace=True))
        x = self.pool(F.leaky_relu(self.conv2(x), inplace=True))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.dropout1(x)
        x = F.relu(self.fc2(x))
        x = self.dropout2(x)
        x = self.fc3(x)
        return x
```

Accuracy: 63%

F1-Score: 0.63625

Possible explanation:

We see an improvement from the accuracy of the base model but a degradation from the ImpNet model. A dropout layer operates by "dropping out" (i.e., deactivating) a certain percentage of neurons in the previous layer (in this case 50%). We might get the above results because:

1. The dropout value of 0.5 was too aggressive for the linear layers, and the model ended up learning less than it could have because it is already such a simple model.

CDropNet

We add dropout regularization to the convolution layers as well.

```
class CDropNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.dropout1 = nn.Dropout(0.1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.dropout2 = nn.Dropout(0.1)
        self.fc1 = nn.Linear(128 * 56 * 56, 1296)
        self.fc2 = nn.Linear(1296, 216)
        self.fc3 = nn.Linear(216, 6)

    def forward(self, x):
        x = self.pool(F.leaky_relu(self.conv1(x)))
        x = self.dropout1(x)
        x = self.pool(F.leaky_relu(self.conv2(x)))
        x = self.dropout2(x)
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x), inplace=True)
        x = self.dropout1(x)
        x = F.relu(self.fc2(x), inplace=True)
        x = self.dropout2(x)
        x = self.fc3(x)
        return x
```

Accuracy: 62%

F1-Score: 0.6049

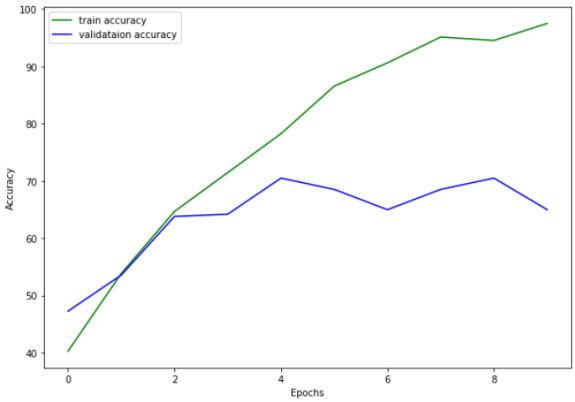
Possible explanations:

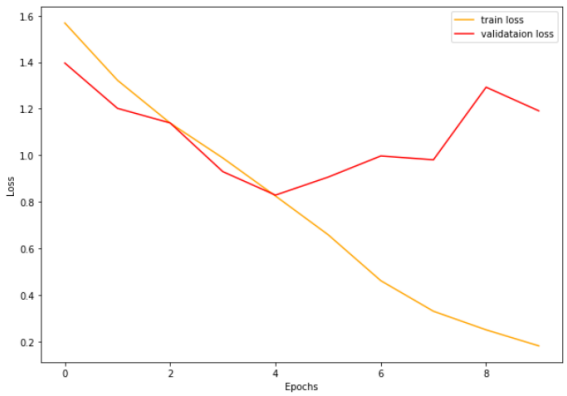
We see an improvement from the accuracy of the base model but a degradation from the ImpNet and DropNet models. We might get the above results because:

1. The dropout value of 0.5 was too aggressive for the linear layers, and the model ended up learning less than it could have because it is already such a simple model.

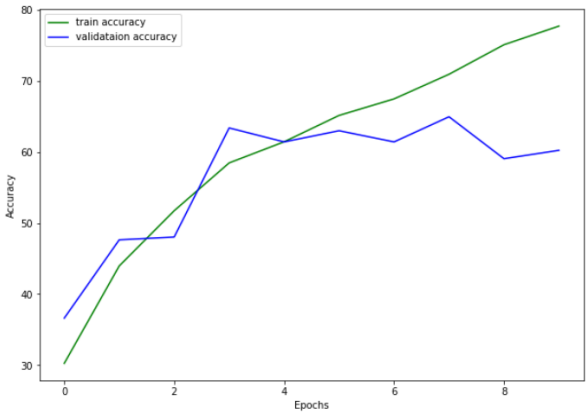
Observations on the above models:

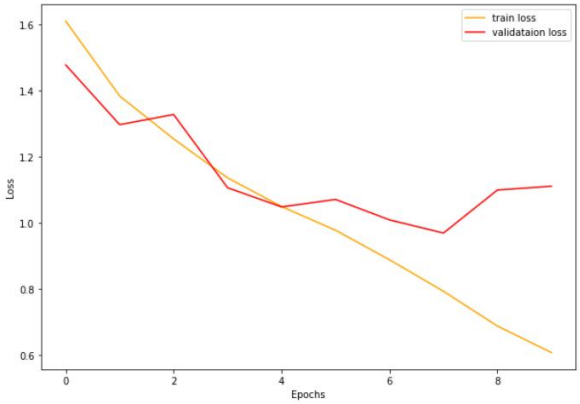
1. **Net**



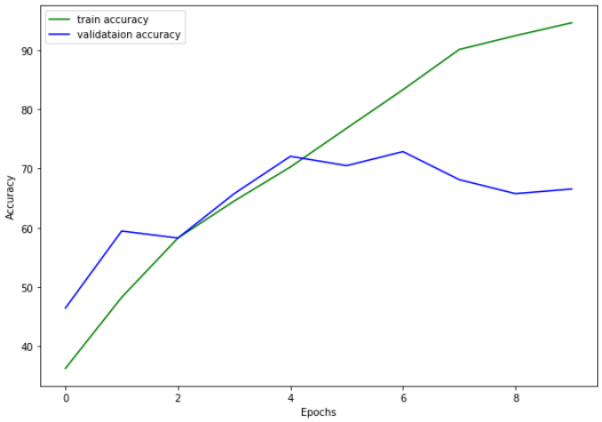


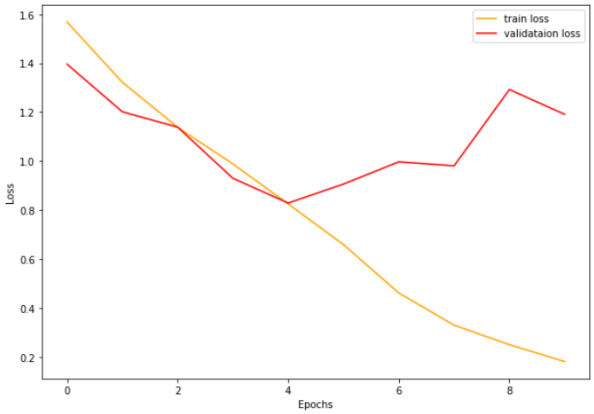
2. ImpNet



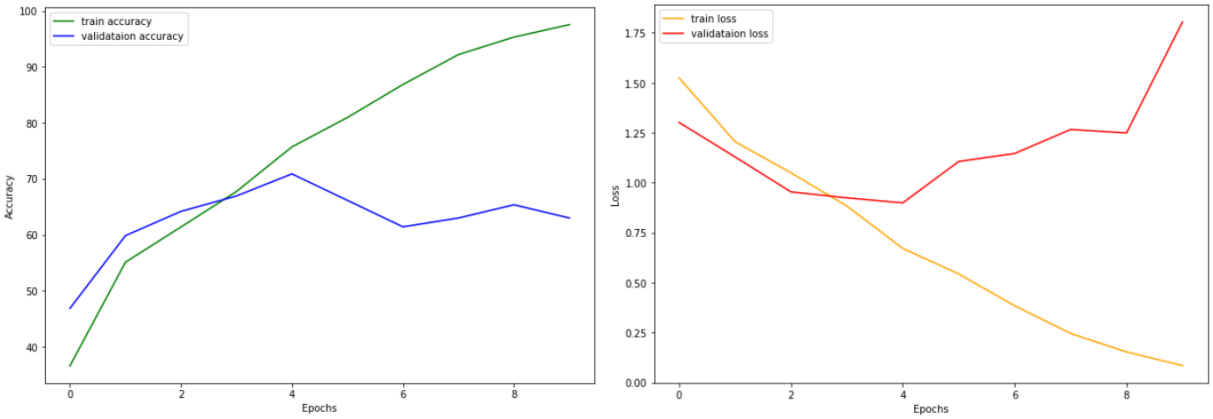


3. DropNet





4. **CDropNet**



Looking at the above curves for the models, it is obvious that the models are severely overfitting. Some of the things I did to mitigate overfitting are:

1. Early stopping – using different values for delta and tolerance.
2. Weight decay – using values ranging from 0.01,0.001, 0.0001.
3. Different Learning Rates
4. Slightly larger images

I began to suspect there was an architectural problem with the model, given that it is so simple. I decided to investigate this using two experimental models.

1. MLP
    a. Three fully connected layers
2. CNN
    a. 2 convolutional layers + 1 fully connected layers

The MLP gave me an accuracy of 41% whereas the CNN gave me a 53% accuracy. This means that the linear layers DO HELP in understanding the features, but it's just that the number of features that's being fed into the first fully connected layer is very large.

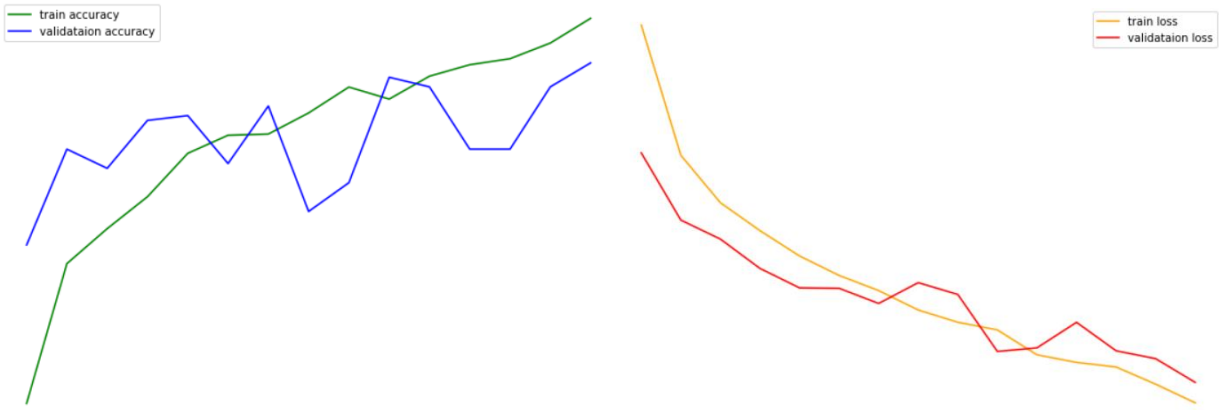
Some ways to mitigate this might have been to:

1. Use a smaller number of output channels on the convolution layers.
2. Use more convolutional layers to gain more information about images.
3. Use a larger number of fully connected layers and progressively reduce the output sizes so that the image features are not reduced to extremely compact representations.

VGG-11

After experimenting with and building upon the sample model from the PyTorch website, I decided to implement a variant of the VGG architecture mentioned in this paper: https://arxiv.org/pdf/1409.1556.pdf

On plotting the training and validation, accuracy and loss, I saw promise in this model because there was no longer any evidence of overfitting.



After this, I trained the model on 15 epochs to achieve an overall accuracy of 84%.

To conclude, I think the models were overfitting because the large number of features that were being fed into the linear layers resulted in a large number of redundant and inefficient weights with little idea of spatial structure. For context, the models before VGG-11 fed in 128*56*56 = 401408 features into the first linear layer and had an output dimension of 1296 unlike the VGG-11 whose first linear layer was fed 25088 features (16 times smaller) and had an output size of 4069./

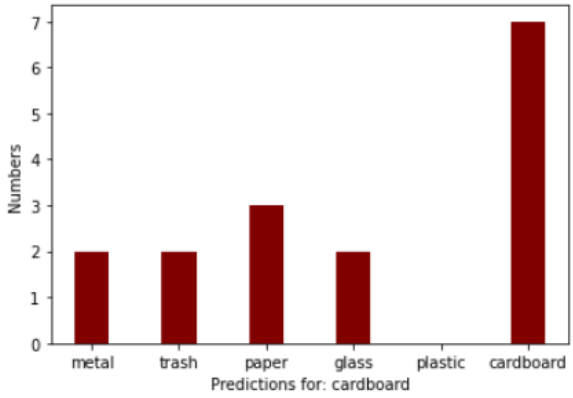
Since such a large number of features was forced into such a small output size, the model is forced to learn a very compact representation of these features, that can help with training accuracy but not validation accuracy.

Closing Thoughts

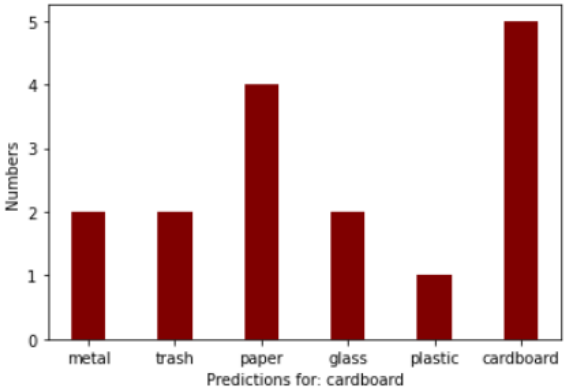
1. The accuracy of classifying cardboard images is significantly lower than the rest of the classes despite accounting for almost 16% of the dataset. I decided to inspect the cardboard images in the dataset for noise and collected some random images. Looking at the images below, I think I can attribute the low accuracy to the lack of discriminative features in the images.

To see how this low accuracy affects the actual garbage classification problem, I visualize the misclassifications of the carboard images (only for those models where cardboard accuracy was very low)

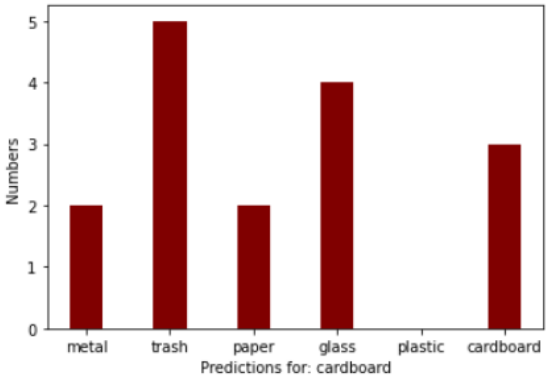
Impnet



DropNet



VGG-11



Most of the time cardboard was most misclassified cardboard as paper, which is not ideal, but is generally better than throwing in the trash bin.

2. The accuracy of classifying trash was as good as classifying the examples of at least 2-3 of the other classes in the dataset. This means that in the end, we did not have to try the down sampling and upweighting approach on the dataset.

Improvements

Model architecture

The models had a hard time classifying cardboard images suggesting that there was a lack of discriminative features in the cardboard images. We need more powerful models to extract these features through:

1. Increase the depth or width of the network by adding more convolutional layers or increasing the number of filters in existing layers.
2. Using advanced architectures like ResNet

Data

1. Rescaling images - two issues to address are that the rescaling was too high, and the aspect ratio was changed during rescaling. This caused some information loss. (This was measured separately using MSE to see the difference between original and rescaled image) - Try to use original images.
    a. Altering the aspect ratio can lead to a distortion of the objects or content within the image. This distortion can affect the spatial relationships and proportions of objects, which may make it more difficult for the model to accurately recognize and classify them. For example, stretching or compressing an image horizontally or vertically can change the shape of objects, making them appear wider or narrower than they should be.
2. Gather more data - use existing datasets.
3. Apply data augmentation techniques during training, such as random cropping, rotation, scaling, or flipping of the input images => increases diversity of the data

Finding optimal hyperparameters

1. Using Bayesian optimization to find the best set of hyperparameters.

Metrics

1. The f1-Score was quite similar to the accuracy in terms of magnitude. Even though we present the class-wise classification accuracy, we can add confusion matrices too.