Udacity Machine Learning Nanodegree

Capstone Report

Classifying Dog Breeds

Marc Seitz

March 2020

1. Definition

1.1 Project Overview

The visual sense is one of five core senses humans have. It has always been a challenge to computer programmers to teach computers the way humans perceive the world through their senses. Image recognition is a relatively easy task for humans and yet a very difficult task for computers to perform.

In order to create a better algorithm for image recognition, in 2010, ImageNet Large Scale Visual Recognition Challenge proposed an annual competition for research teams to evaluate their algorithms on a large dataset with 1000 categories and over 1 million images [1]. In the 2010s, the error rates of misclassifying images fell drastically, in large part due to the introduction of deep neural network optimized for spatial variations, called convolutional neural networks [2]. The ImageNet competition is still held every year.

By now, convolutional neural networks (CNN) are widely accepted for image recognition and image classification tasks. Image recognition is also a thriving research field with applications limited by our imagination: everything humans see, an algorithm could potentially be trained to see.

There is a plethora of real-world application for this research, such as:

- Car lane recognition for enabling self-driving cars
- Handwriting recognition for archiving documents faster
- Action recognition for supporting industrial line workers

My personal motivation for working on image classification is my background as software engineer in the manufacturing industry. I have seen endless opportunities on factory floors to introduce cameras and image recognition to increase production efficiency. Even beyond manufacturing, I am keen to apply my machine learning knowledge to other domains.

1.2 Problem Statement

The objective of this project will be to use deep learning techniques to classify dog breeds.

When given an image, we want to be able to determine if it contains a dog and furthermore, return the canine's breed.

To achieve this, we plan on using different neural network architectures such as a basic shallow CNN and a pretrained deep CNN.

1.3 Metrics

The evaluation metric for this problem will be the classification accuracy, which is defined as the percentage of correction prediction.

$$\boldsymbol{Accuracy} = \frac{\boldsymbol{\#\ of\ correct\ classifications}}{\boldsymbol{\#\ of\ total\ classifications}}$$

Classification accuracy was an optimal metric because it is presumed that the dataset will be relatively symmetrical with this being a multi-classifier whereby the target data classes will be generally uniform in size.

Other metrics such as Precision, Recall, or F1 score were ruled out as they are more applicable to classification challenges that contain a relatively small target class in an unbalanced dataset.

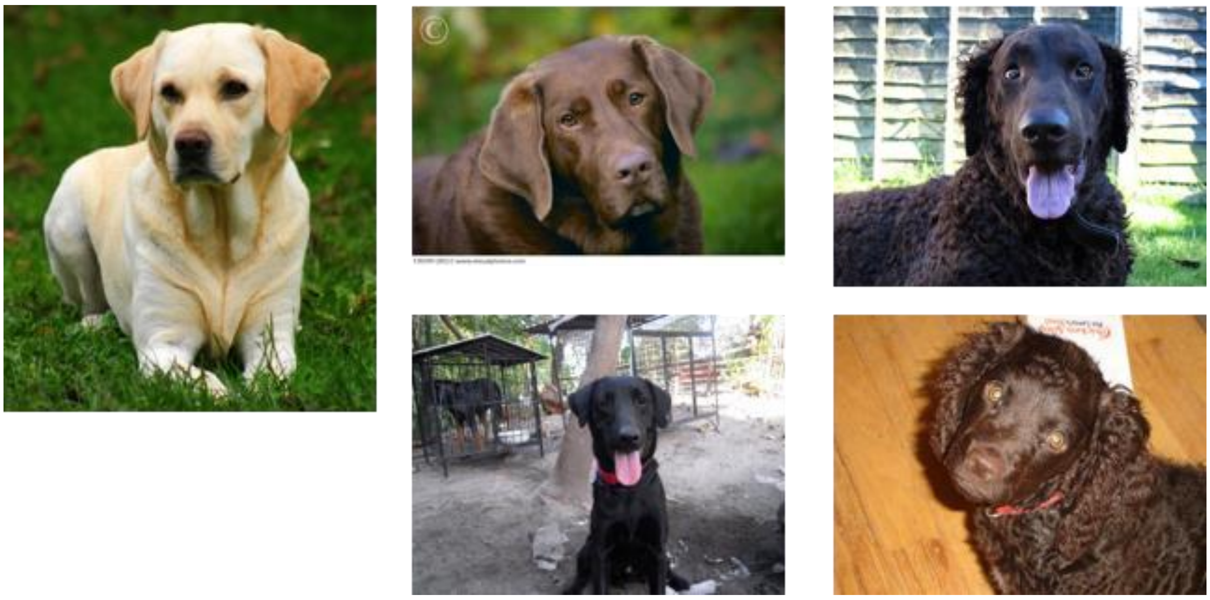
2. Analysis

2.1 Data Exploration and Visualization

For this project we will use two datasets provided by Udacity and by the University of Massachusetts, respectively. The dataset provided by Udacity contains 8351 images of dogs from 133 classes (dog dataset). The dataset from the University of Massachusetts contains 13233 images of humans (human dataset).

The full datasets can be downloaded here (dogs) and here (humans).

A set of example images from the dog dataset:



A set of example image from the human dataset:

2.2 Algorithms and Techniques

The proposed solution to this problem is to apply deep learning techniques that have proved to be highly successful in the field of image classification.

First, we will preprocess the images (more on that in a later section) to be able to feed them into the neural network models.

Second, we will train a convolutional neural network with these datasets and make predictions. We will begin with a CNN with a simple, shallow architecture before using a more sophisticated, more complex CNN that has been pretrained on the ImageNet dataset.

A convolutional neural network is a class of deep, feed-forward layers that are space invariant. The layers in a CNN are organized into three dimensions (width, height, depth) and neurons in one layer are not necessarily connected to all neurons in subsequent layers.

CNNs extract features from images by sliding a filter over the input, row by row, which activates when there is a visual feature and finally stores everything in a feature map. Each convolutional layer usually followed by a pooling layer, which reduces the dimensionality of the feature map but retains meaningful data. The classification layer is the final layer of a CNN, which is flattens the 3-dimensional data into a 1-dim vector that returns the network output.

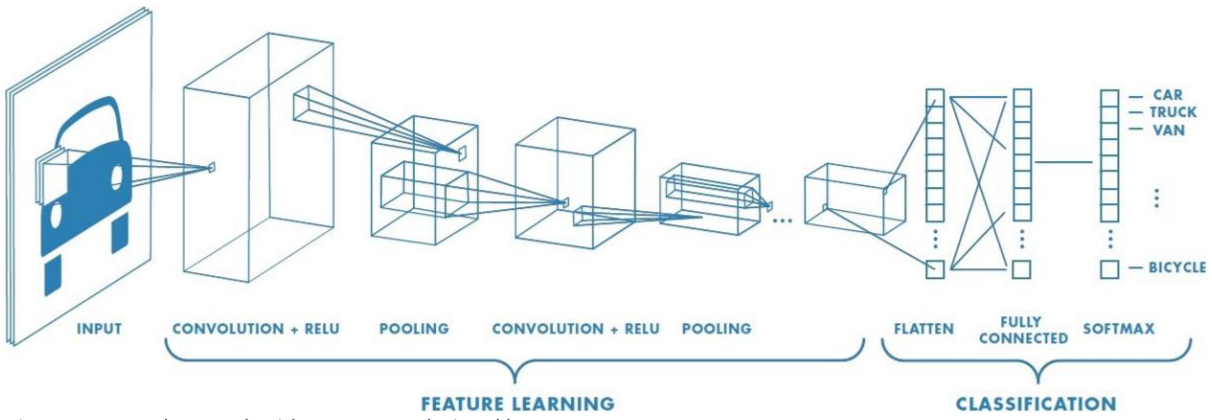


Figure 1 - Neural network with many convolutional layers

We will use the evaluation metrics described in earlier section to compare the performance of these solutions against the benchmark models in the next section.

2.3 Benchmark

For the benchmark model, we will use models that have performed well on a similar dataset of dog breeds from Stanford [3].

Table 1: Summary of Benchmarks of Stanford Dogs [4]

Method	**Top - 1 Accuracy**
SIFT + Gaussian Kernel	22%
Unsupervised Learning Template	38%
Gnostic Fields	47%
Selective Pooling Vectors	52%

3. Methodology

3.1 Data Preprocessing

3.1.2 Data Splitting

Splitting a dataset into training and testing set is best practice for evaluating a model's true accuracy. In this project we go a step further and split the dog dataset into a train, validation and test set. The train set with 6680 images will be used to train the model, whereas the validation set (835 images) is used for evaluating the model throughout the process of training. The test set (836 images) will be used to get the classification accuracy of the model.

3.1.2 Data Preprocessing

In order to feed our images in our neural network, we have to preprocess them in the appropriate input format. The images are resized by performing a random crop to 224x224 pixels, which is the minimum height and width for pretrained models from PyTorch [5]. In addition, the images are normalized with a mean and standard deviation as followed:

$$\begin{aligned} \boldsymbol{mean} &= [\mathbf{0.485, 0.456, 0.406}] \\ \boldsymbol{std} &= \mathbf{0.229, 0.224, 0.225}] \end{aligned}$$

3.1.3 Data Augmentation

We have also applied several transformations to augment the data in the train set for a more robust algorithm. We have used:

- A random vertical flip with a 30% likelihood;
- A random horizontal flip with a 30% likelihood; and
- A random rotation by 30 degrees.

3.2 Implementation

3.2.1 Model Architecture

We will start with designing a basic convolutional neural network from scratch using PyTorch.

Starting with a *Sequential* model, so we can build the model layer by layer.

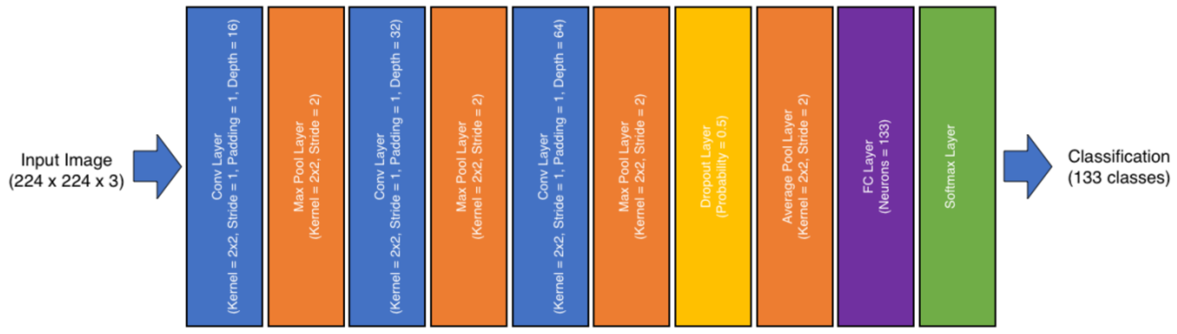
We will begin with a simple model architecture, consisting of three *Convolutional* layers. All three convolutional layers have a *Rectified Linear Unit* activation function as well as a *Max Pooling* layer.

Then, there is a layer with a *Dropout* value of 50%, which randomly excludes neurons from each update cycle in order to make the network more generalizable and prevent overfitting to the training data.

Next, we add another pooling layer with an *Averaging Pooling* function before we flatten then 3D data into a 1D vector.

This 1D vector is passed into a fully-connected *Linear* layer with 133 output neurons. These neurons are then passed through a *Log Softmax* activation function. Softmax makes the output sum up to 1 so the output can be interpreted as probabilities.

The model will make its prediction based on the class with the highest probability.


Input Image
(224 x 224 x 3)
Conv Layer
(Kernel = 2x2, Stride = 1, Padding = 1, Depth = 16)
Max Pool Layer
(Kernel = 2x2, Stride = 2)
Conv Layer
(Kernel = 2x2, Stride = 1, Padding = 1, Depth = 32)
Max Pool Layer
(Kernel = 2x2, Stride = 2)
Conv Layer
(Kernel = 2x2, Stride = 1, Padding = 1, Depth = 64)
Max Pool Layer
(Kernel = 2x2, Stride = 2)
Dropout Layer
(Probability = 0.5)
Average Pool Layer
(Kernel = 2x2, Stride = 2)
FC Layer
(Neurons = 133)
Softmax Layer
Classification
(133 classes)

Furthermore, we have chosen the loss function *Cross Entropy Loss*, which is most common for classification problems, and *Adam* optimizer, which is generally a good optimizer for many use cases.

3.2.2 Training and Validation

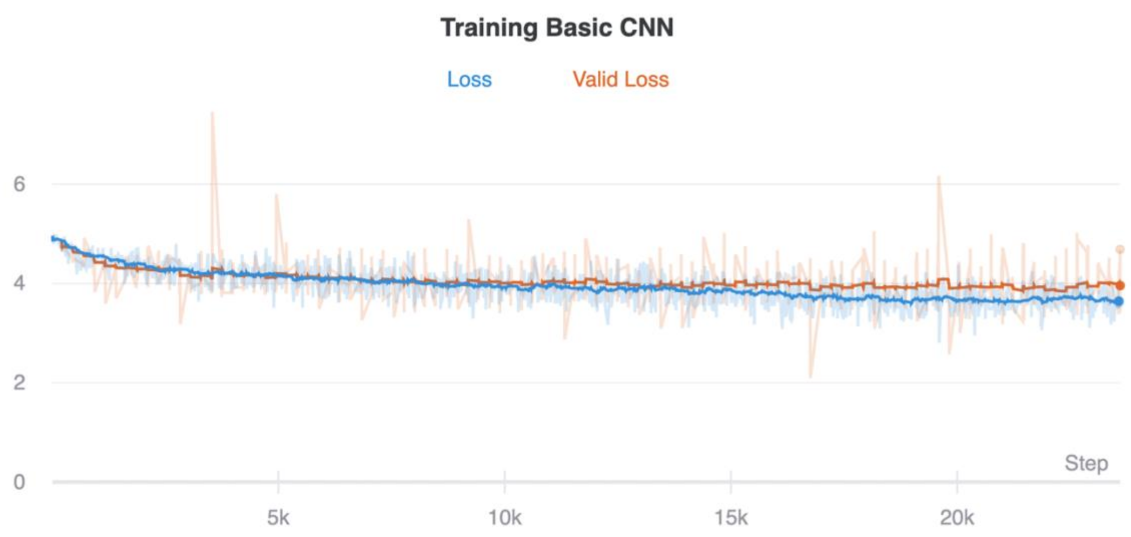
Here we will train and evaluate the model. We will start with 100 epochs, which is the number of times the model will cycle through the data. Each cycle the model will improve until it reaches a plateau.

We chose a small batch size of 32 and a learning rate for the optimizer of 0.0001.

```python
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation loss
        train_loss = 0.0
        valid_loss = 0.0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss

            optimizer.zero_grad()
            outputs = model(data)
            loss = criterion(outputs, target)

            loss.backward()
            optimizer.step()

            wandb.log({"Loss": loss})

            train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss

            val_outputs = model(data)
            val_loss = criterion(val_outputs, target)

            wandb.log({"Valid Loss": val_loss})

            valid_loss += ((1 / (batch_idx + 1)) * (val_loss.data - valid_loss))


        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

        # save the model if validation loss has decreased

        if valid_loss < valid_loss_min:
            valid_loss_min = valid_loss
            torch.save(model.state_dict(), save_path)
            torch.save(model.state_dict(), os.path.join(wandb.run.dir, save_path))
            print("Saving model with current validation loss: {}".format(valid_loss
))

    # return trained model
    return model
```

The best model has been on Epoch 87 and hasn't improved since then:

```
Epoch: 1         Training Loss: 4.858908   Validation Loss: 4.754373
Saving model with current validation loss: 4.754372596740723
Epoch: 2         Training Loss: 4.681979   Validation Loss: 4.567915
Saving model with current validation loss: 4.567914962768555
Epoch: 3         Training Loss: 4.571255   Validation Loss: 4.478428
Saving model with current validation loss: 4.478428363800049
…
Epoch: 87        Training Loss: 3.668716   Validation Loss: 3.905561
Saving model with current validation loss: 3.9055614471435547
…
Epoch: 97        Training Loss: 3.637293   Validation Loss: 3.928775
Epoch: 98        Training Loss: 3.623734   Validation Loss: 3.963676
Epoch: 99        Training Loss: 3.619128   Validation Loss: 3.961084
Epoch: 100       Training Loss: 3.615571   Validation Loss: 3.974969
```



3.2.3 Testing

Here we will review the accuracy of the model on test set.

```
def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.
    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().nump
y())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
        100. * correct / total, correct, total))
```

```
Test Loss: 3.840610
Test Accuracy: 13% (117/836)
```

The accuracy on the test set of 13% is very low. It's to be expected that the accuracy will not be high. The CNN is only three layers deep and not very feature rich either with maximum depth of 64 neurons on the third convolutional layer.

3.3 Refinement

The basic CNN, which we design from scratch did not perform well on our classification task. Hence, we have decided to use a pretrained model and use it for transfer learning.

3.3.1 Model Architecture

From all possible pretrained models, we have chosen the ResNet50 model. ResNet50 is a 50-layer deep convolutional neural network that has been trained on the ImageNet dataset with millions of images in 1000 classes. Among the 1000 classes are also 120 classes of dogs. Hence, the images the model has been trained on is similar to our image dataset and we can take advantage of the model's feature extractions from its convolutional layers.

In order to successfully transfer the learning of this model to our classification task, we freeze the weights of all layers and replace the final fully-connected *Linear* layer with our own *Linear* layer that connects to 133 output neurons and is activated by a *Log Softmax* function.

```
# Instantiate a pretrained ResNet50 model
model_transfer = models.resnet50(pretrained=True)

# Freeze parameters
for param in model_transfer.parameters():
       param.requires_grad = False

# Overwrite the last fully-connected layer
model_transfer.fc = nn.Sequential(
                           nn.Linear(2048, 133),
                              nn.LogSoftmax(dim=1))

if use_cuda:
      model_transfer = model_transfer.cuda()
```

Optimizer and loss function remain the same as in the previous model.

3.3.2 Training and Validation

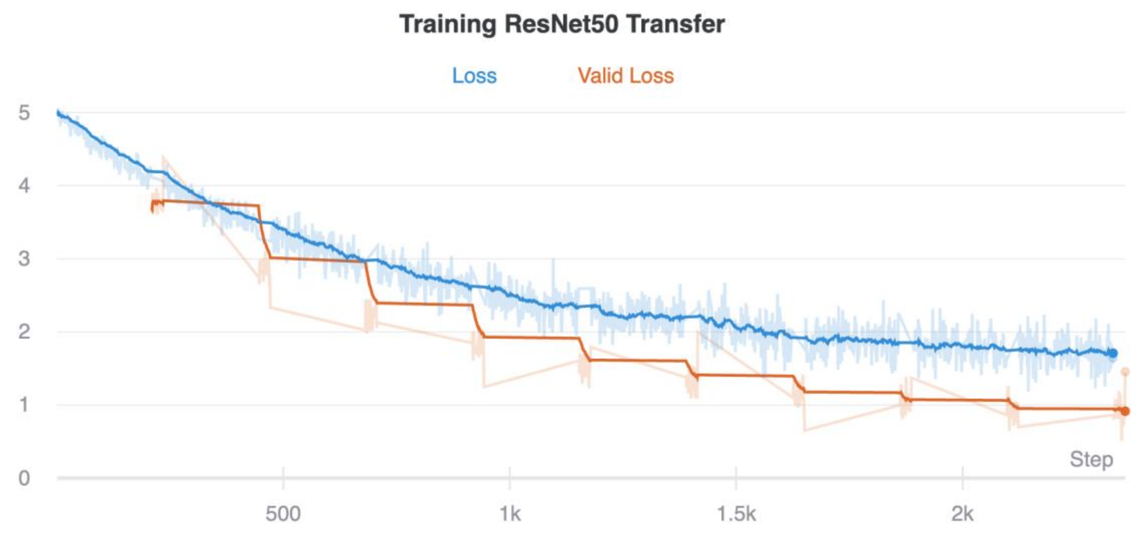
Here we will train and evaluate the transfer model. We will start with 10 epochs. We chose a small batch size of 32 and a learning rate for the optimizer of 0.0001.

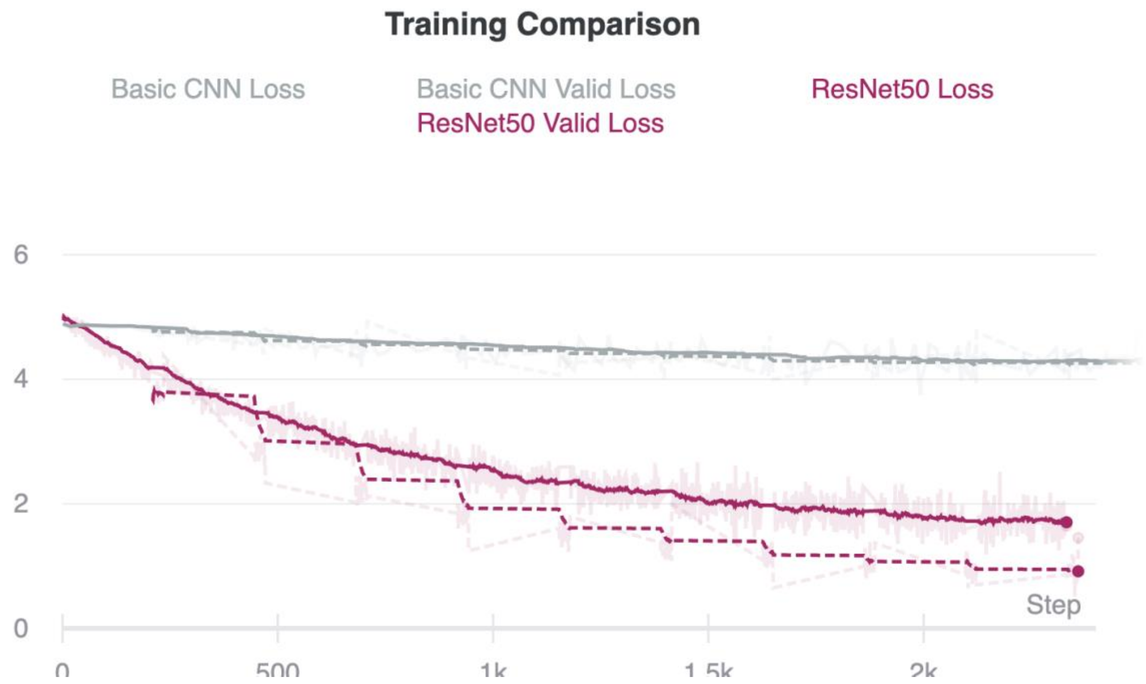
```
# train the model
model_transfer = train(EPOCHS_TRANSFER, loaders_transfer, model_transfer, optimizer
_transfer, criterion_transfer, use_cuda, MODEL_PATH_TRANSFER)
```

With only 10 epochs to train, we can see the improvements in the validation loss immediately:

```
Epoch: 1         Training Loss: 4.511115   Validation Loss: 3.784088
Saving model with current validation loss: 3.784088373184204
Epoch: 2         Training Loss: 3.712216   Validation Loss: 2.846710
Saving model with current validation loss: 2.846710443496704
Epoch: 3         Training Loss: 3.133514   Validation Loss: 2.207248
Saving model with current validation loss: 2.2072482109069824
Epoch: 4         Training Loss: 2.732800   Validation Loss: 1.794451
Saving model with current validation loss: 1.7944506406784058
Epoch: 5         Training Loss: 2.418190   Validation Loss: 1.519997
Saving model with current validation loss: 1.5199968814849854
Epoch: 6         Training Loss: 2.215704   Validation Loss: 1.331293
Saving model with current validation loss: 1.3312932252883911
Epoch: 7         Training Loss: 2.038000   Validation Loss: 1.123548
Saving model with current validation loss: 1.1235483884811401
Epoch: 8         Training Loss: 1.890112   Validation Loss: 1.032005
Saving model with current validation loss: 1.0320050716400146
Epoch: 9         Training Loss: 1.784554   Validation Loss: 0.930292
Saving model with current validation loss: 0.9302915930747986
Epoch: 10        Training Loss: 1.735156   Validation Loss: 0.904379
Saving model with current validation loss: 0.9043793678283691
```

The final trained model has a significantly lower validation loss than we could have ever achieved with the basic CNN.





3.3.3 Testing

Here we will review the accuracy of the model on test set.

```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
Test Loss: 0.871511
Test Accuracy: 84% (706/836)
```

The accuracy on the test set of 84% is very high and clearly much better than the basic CNN. After training the transfer model for only 10 epochs, this is a huge success and astonishing accuracy of the algorithm.

4. Results

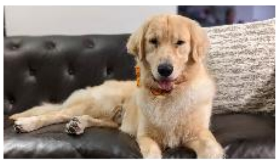
4.1 Model Evaluation and Validation

During the model development phase, the validation data was used to evaluate the model. The final model architecture, a pretrained ResNet50 with a custom fully-connected layer, and the hyperparameters were chosen because they performed well overall.

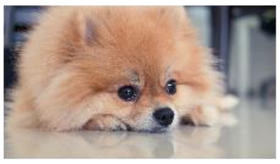
After training both the basic CNN and transferred ResNet50 model on the dog dataset, it's time to test if the classifier also performs well with new data. We run the classifier on new images and perform one of the following three tasks:

- If the image contains a dog, print the classified dog breed;
- If the image contains a human, print the lookalike dog breed; or
- If the image contains neither, print the lookalike dog breed.

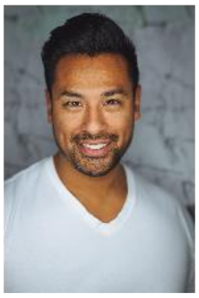
Hello, you dog!
You are a Golden retriever.



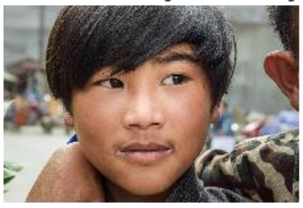
Hello, you dog!
You are a Pomeranian.



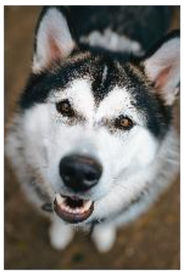
What's up, human!
You look like a Bullmastiff.



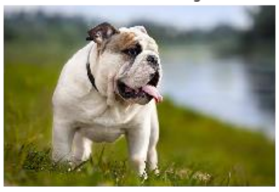
What's up, human!
You look like a Portuguese water dog.



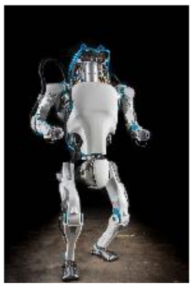
Hello, you dog!
You are a Alaskan malamute.



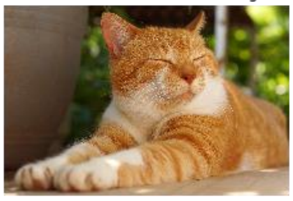
Hello, you dog!
You are a Bulldog.



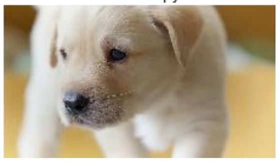
Hi, whatever you are!
You look like a Bull terrier.



Hi, whatever you are!
You look like a Nova scotia duck tolling retriever.



Hello, you dog!
You are a Great pyrenees.



The model performs well on the new set of images of dogs, humans and others. The dog breed labels are accurate. The labels of humans are interesting but accurate and going through the training images of these classes one can see where the features come from. For the image of the robot it's completely arbitrary in my opinion. I don't see the features to expect a bull terrier, expect perhaps the slim torso and thick head.

4.2 Justification

The final model achieved a classification accuracy of 84% on the testing data which exceeded the benchmark from section 2.3.

Given the classification accuracy on the brand-new images in section 4.1, we can safely say the final model is an adequate solution to solve the problem of dog breed classification.

5. Conclusion

In this project, we have built an algorithm to detect and classify dogs in images according to their breed. First, we tried a basic CNN architecture, which performed poorly with 13% classification accuracy. Second, we tried a more complex CNN architecture based on a pretrained ResNet50 model with a custom final fully-connected layer. This model performed extremely well with a classification accuracy of 84% on our test set of 836 dog images.

The result of the transfer learning is much better than I had expected and probably much better than a human (without a trained eye) could classify dog breeds.

Personally, I have not even heard of some of the dog breeds before, so having an algorithm tell me the type of dog is magical.

5.1 Reflection

The process used for this project can be summarized with the following steps:

1. The initial problem was defined, and an appropriate dataset was obtained.
2. The data was explored and analyzed.
3. The data was preprocessed, and features were extracted.
4. A basic model was trained and evaluated.
5. A more advanced model was trained and evaluated.

Without a doubt, the transfer learning model beats the basic CNN in any way in this classification problem. One take away for future problems would be to take a look at existing, pretrained models where the underlying dataset shares similarities with one's dataset and then use transfer learning instead of building a model from scratch.

5.2 Improvements

If we were to continue with this project, there are a number of additional areas that could be explored to improve the algorithm:

- Increase the dataset's size – currently, we are training on only 6680 images of dogs;
- Include more dog breed classes as well as mixed dog breeds;
- Tune hyperparameters further and more carefully; and
- Experiment with other transfer model architectures, such as Inception V3 or Xception.

6. References

[1] O. Russakovsky, J. Deng, S. Hao, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. Berg, L. Fei-Fei (2014), “ImageNet Large Scale Visual Recognition Challenge”, arXiv:1409.0575 [cs.CV]

[2] A. Krizhevsky, I. Sutskever, G. Hinton (2012), “ImageNet Classification with Deep Convolutional Neural Networks”, https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[3] A. Khosla, N. Jayadevaprakash, B. Yao, L. Fei-Fei (2011), “Stanford Dogs Dataset”, http://vision.stanford.edu/aditya86/ImageNetDogs/main.html

[4] Hsu, David (2015), "Using Convolutional Neural Networks to Classify Dog Breeds", http://cs231n.stanford.edu/reports/2015/pdfs/fcdh_FinalReport.pdf

[5] https://pytorch.org/docs/stable/torchvision/models.html