UNIVERSITY OF MICHIGAN
Department of Electrical Engineering and Computer Science
EECS 445 — Introduction to Machine Learning
Winter 2022

Project 2: Karl's Convolutional Kennel
An exploration of deep learning techniques for classification and representation learning
Due: Wednesday, 3/23 at 10:00pm

Section	Points	Recommended Completion Date
1. Data Preprocessing	10	Saturday, 3/12
2. Convolutional Neural Networks	30	Wednesday, 3/16
3. Visualizing what the CNN has learned	10	Wednesday, 3/16
4. Transfer Learning & Data Augmentation	25	Saturday, 3/19
5. Challenge	15	Wednesday, 3/23
6. Code Appendix	10	Wednesday, 3/23

Include your entire project code (copy/paste or screenshot) as an appendix in your report submission. Please try to format lines of code so they are visible within the pages.

Upload your file `uniqname.csv` containing the label predictions for the held-out data to the canvas assignment named Project 2 Challenge Submissions.

Introduction

Dog-gone! The neighborhood kennel owner, Karl, is having a "ruff" time: during his morning rounds to check on his new foster puppies, he found to his horror that they had all dug their way under the fence and escaped! He's now enlisted the help of the entire neighborhood to help round up his lost pups. He's asked his neighbors to send in photos of dogs that they believe could have escaped from his kennel. Now Karl is receiving thousands of images a day - too many to go through himself. He needs your help to build a system to automatically distinguish his dogs from the others.

Karl's kennel houses Golden Retrievers, but another kennel that houses Collies has experienced an escape of its own! We need a classifier that distinguishes Golden Retrievers versus Collies. To help with that, Karl has provided us with a few images of his dogs along with images of Collies. He has also sent images of other breeds including Samoyeds, Miniature Poodles, Great Danes, Dalmatians, Siberian Huskies, Yorkshire Terriers, Chihuahuas, and Saint Bernards just in case those might be helpful. We've included some of the pictures he sent below.

Figure 1: Sample images from the the dataset.

In our mission to help Karl, we will explore supervised deep learning techniques for image data. In this setting, we will use a convolutional neural network (CNN) to classify Karl's dog images by breed, where the breed associated with each image serves as the classification label. We will then explore *supervised pretraining*, a type of *transfer learning* that will use a feature representation learned on a related task to augment our classification abilities. Finally, we will experiment with data augmentation to enrich our training data and improve the robustness of our models. After this, you will be well-equipped to take on Karl's challenge: implementing and training a deep neural network of your own design to classify dog images by breed!

Getting Started

Skeleton Code

Please download the Project 2 skeleton code `project2.zip` from Canvas. This zip file contains the full project dataset, and may take several minutes to download. After unzipping, please ensure that all of the following files are present:

- **Executables**
  - `augment_data.py`
  - `confusion_matrix.py`
  - `dataset.py`
  - `predict_challenge.py`
  - `test_cnn.py`
  - `train_challenge.py`
  - `train_cnn.py`
  - `train_source.py`

  - `train_target.py`
  - `visualize_cnn.py`
  - `visualize_data.py`

- **Models**
  - `model/challenge.py`
  - `model/source.py`
  - `model/target.py`

- **Checkpoints**
  - `checkpoints/`

- **Data**
  - `data/dogs.csv`
  - `data/images/`

- **Utilities**
  - `config.json`
  - `train_common.py`
  - `utils.py`
  - `requirements.txt`

Dataset

This dataset contains 12,775 PNG image files of 10 different dog breeds. These images are stored under `data/images/`. Each image contains 3 color channels (red, green, blue) and is of size $3 \times 64 \times 64$. These images have already been divided into 4 data partitions: a training set, a validation set, a test set and a held-out challenge set. Metadata containing the image label and information regarding to what data partition the image belongs is documented in `data/dogs.csv`. Note that the challenge labels have been removed from this file. As in Project 1, you will include your challenge model's predictions on this data partition as a part of your submission.

Python Packages

This project will introduce you to PyTorch , a deep learning framework written and open-sourced by Facebook's AI group. To install PyTorch, follow the instructions on https://pytorch.org/get-started/locally/. We recommend selecting **Pytorch Build - Stable** and **CUDA - None**. We have written this project to run in a reasonable time on standard laptop machines; however, if you have access to a compatible GPU and wish to use it, please select the appropriate CUDA version. To ensure fairness, challenge submissions that utilize GPU hardware will be graded separately from non-GPU submissions.

Here is a list of all the Python packages necessary to complete this project.

```
matplotlib (3.5.1)
numpy (1.22.2)
torch (1.10.2)
torchvision (0.11.3)
pandas (1.4.1)
imageio (2.16.0)
scikit-learn (1.0.2)
pillow (9.0.1)
scipy (1.8.0)
```

Note: if you are using WSL, some extra configuration may be necessary for matplotlib to display plots. As an alternative, you may want to use `plt.savefig()` to directly save the plot to a file.

1 Data Preprocessing [10 pts]

Real datasets often contain numerous imperfections (e.g., incorrect labels, feature noise, or irrelevant examples) and may not be immediately useful for training our model. To develop a dataset more conducive for training, we will explore normalizing our data. The preprocessing will be implemented in the file `dataset.py`.

(a) **Implement** the `fit()` and `transform()` functions in the `ImageStandardizer` class. The `fit()` function is first called with the training dataset as input, and it calculates and stores the per-channel mean and standard deviation from the training data. This results in a scalar value each for the mean and standard deviation of the R channel, G channel, and B channel (6 scalars total for the entire training data). Specifically, we assume that the underlying distributions of each image channel (red, green, and blue) are independent distributions with mean 0 and variance 1. To embed these assumptions, zero-center each image channel by subtracting the per-channel mean value, then scale each image channel by dividing by the per-channel standard deviation. These mean and standard deviation values computed from training data then are applied to the standardization of all data partitions (validation/testing) with the `transform()` function.

Note: When reading in images, make sure images are represented as `np.float64` rather than `ints`.

Answer the following questions regarding this normalization process.

i. (4 points) Run the script `dataset.py`. What is the mean and standard deviation of each color channel (RGB), learned from the entire training partition?

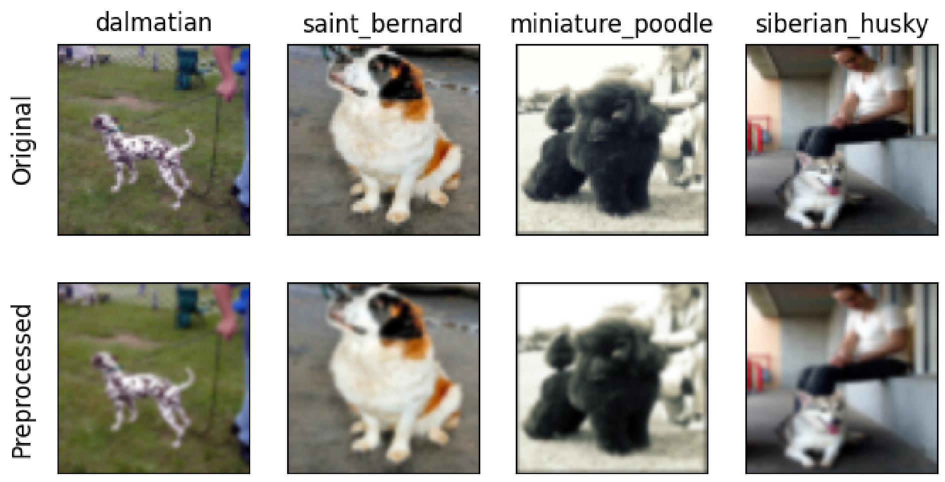
Solution:
(R, G, B)
Mean: [124.495 118.847 95.293]
Std: [62.754 59.597 62.425]

ii. (4 points) Why do we extract the mean and standard deviation from the training set as opposed to the other data partitions?

Solution: Rationale for standardizing with respect to only the training set:
We want our validation and test performance to be indicative of the ability of our model to generalize to new, unseen data. By taking the mean and std of only the training set, we ensure that we are not capturing any of the structure of the data partitions we are using for evaluation.

(b) (2 points) **Run** the script `visualize_data.py` to check that the preprocessing did not result in a loss of information. If you did it correctly, the images should look slightly blurred after running `visualize_data.py`. **Include** your output images from `visualize_data.py` below.

Solution:



Included images need not be the same pictures but should show a similar minor change in blurring between original and preprocessed versions.

2 Convolutional Neural Networks [30 pts]

With our data now in a suitable form for training, we will explore the implementation and effectiveness of a convolution neural network (CNN) on the dataset. In this section, we focus on our *target* task which involves **only 2 classes: Collies and Golden Retrievers.** Recall the basic structure of a CNN: a sequential combination of convolutional, activation, pooling and fully connected layers.

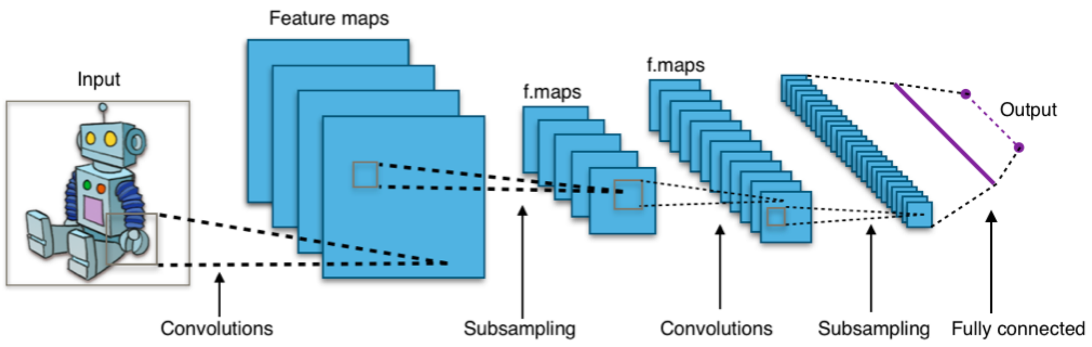


Figure 2: An example convolutional neural network. By Aphex34, licensed under CC BY-SA 4.0.

CNNs used for image classification can often be viewed in two parts, where the first part learns a representation of the image data — or "encoding" — and the second learns a mapping from this image encoding to the output space. Our CNN will follow this schema, and the output space will be the two class labels we are considering. The image encoder will be composed of convolutional layers followed by max pooling, which captures patterns across the image data by sliding filters with learned weights across the image. Once we have our image encoding, we will train a single fully connected layer to learn a linear combination of this representation, from which we will ultimately derive our classification.

We have provided the training framework necessary to complete this problem in the file `train_cnn.py`. All of the following problems that involve implementation will require you to use the PyTorch API. We've provided some tips in Appendix A, and you are encouraged to explore the documentation and online tutorials to learn more.

(a) (3 points) **Study** the Target architecture specified in Appendix B (on page 27). We call this CNN architecture the "Target" architecture because our "target" task is discerning between Collies and Golden Retrievers. **How many** learnable float-valued parameters does the model have? Show your work by manually calculating this number, and you can later verify against the program output after implementing the model.

Solution: The general formula for counting model paratemeters (layer weights) are:

- Convolutional layer: filter_width $\times$ filter_height $\times$ n_input_channels $\times$ n_output_channels
- FC layer: n_input $\times$ n_output

Layer	Weights	Biases	Total
1: Conv$_1$	$5 \times 5 \times 3 \times 16$	16	$1,216$
2: Pool$_1$	0	0	0
3: Conv$_2$	$5 \times 5 \times 16 \times 64$	64	$25,664$
4: Pool$_2$	0	0	0
5: Conv$_3$	$5 \times 5 \times 64 \times 8$	8	$12,808$
6: FC$_1$	32×2	2	66

The above total to $\boxed{39,754}$ float-valued parameters.

(b) **Implement** the architecture by completing the `Target` class in `model/target.py`. In pytorch, we define a neural net by subclassing `torch.nn.Module`, which handles all the gradient computations automatically. Your job is to fill in the appropriate lines in three functions: `__init__()`, `init_weights()` and `forward(x)`.

- Use `__init__()` to define the architecture, i.e. what layers our network contains. At the end of `__init__()` we call `init_weights()` to initialize all model parameters (weights and biases) in all layers to desired distributions.
- The `forward(x)` function defines the forward propagation for a batch of input examples, by successively passing output of the previous layer as the input into the next layer after applying activation functions, and returning the final output as a `torch.Tensor` object.

- The `torch.Tensor` class implements a `backward()` function. As its name suggests, it performs back propagation and computes the partial derivatives with respect to each model parameter using chain rule. While you do not have to implement `backward()`, you should review the lecture slides on backpropagation, so that you understand why it is important when training the network.

(c) After making a forward pass, we can use the output from the CNN to make a prediction. **Implement** the `predictions()` function in `train_common.py` to predict the numeric class label given the network output. This output is just the raw output prediction of the network (without normalization). *Hint: When interpreting the raw output from the model, think about what the* ***true*** *labels are and note that the network is trying to output this true label. Pay attention to the fact that the output dimension is 2 and think about how the indices of the output vector relate to the class labels.*

(d) **Review** the training script `train_cnn.py`. Based on the model specification in Appendix B, **fill in** the definitions for `criterion` and `optimizer`.

(e) In `train_common.py`, `train_epoch()` loops through the training data and updates the model parameters based on the criterion and optimizer. After a complete pass of the training set (called an "epoch"), we evaluate the model on the validation set, and save this solution as a checkpoint. We can then use `evaluate_epoch` to evaluate the learned parameters on both the training and validation data. **Review** the definitions of the following functions:

- `train_epoch`: within one epoch, we pass batches of training examples through the network, use backpropagation to compute gradients, and update model weights using the gradients.
- `evaluate_epoch`: we pass the entire validation set (in batches) through the network and get the model's predictions, and compare these with the true labels to get an evaluation metric.
- `save_checkpoint`: checkpointing — the periodic saving of model parameters — is an important technique for training large models in machine learning; if a hardware failure occurs due to a power outage or our code fails for whatever reason, we don't want to lose all of our progress!

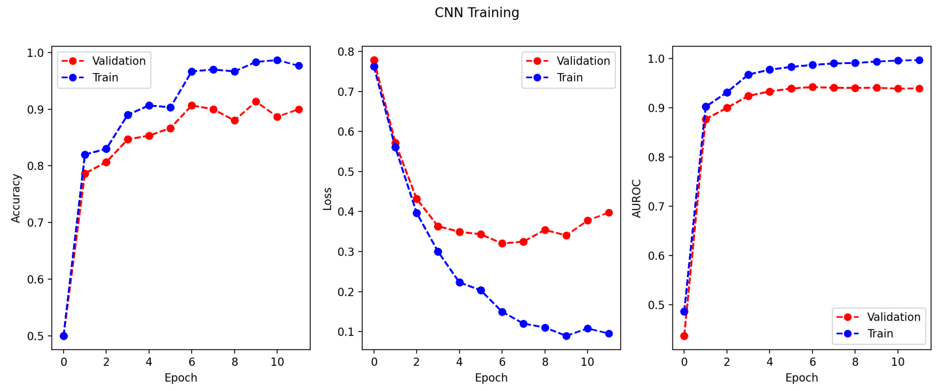
Given the size of our training data, we're unlikely to have a learned a good model after a single epoch. Thus, we will have to loop over the data multiple times i.e., train for multiple epochs. But how will we know when to stop? We will use 'early stopping' and leverage the performance on the validation data. More specifically, we will stop training after the validation loss does not strictly decrease for some number of epochs defined with the variable `patience`. In other words, we should keep track of how many contiguous epochs have passed in which the validation loss has either stayed the same or increased relative to the global minimum loss. See `log_training` in `utils.py` for how to extract this validation loss from the stats data structure. In `train_common.py` **implement** `early_stopping` and **implement** the training steps in `train_epoch`.

(f) **Execute** `train_cnn.py` to train the model with a patience of **5**.

Note that if you run the script more than once, you will be prompted to enter the epoch number at which the model parameters from the saved checkpoint will be restored and training will *continue* from that epoch. If you enter 0, then the model will be trained from scratch and the old checkpoints will be deleted.

This script will also create three graphs that are updated every epoch to monitor the model's performance in terms of loss, accuracy, and AUROC on the train and validation set. **Answer** the following questions.

i. (4 points) You may observe that the validation loss does not monotonically decrease in the training plot. It should look similar to the plot included below. **Describe** at least two reasons for this behavior.

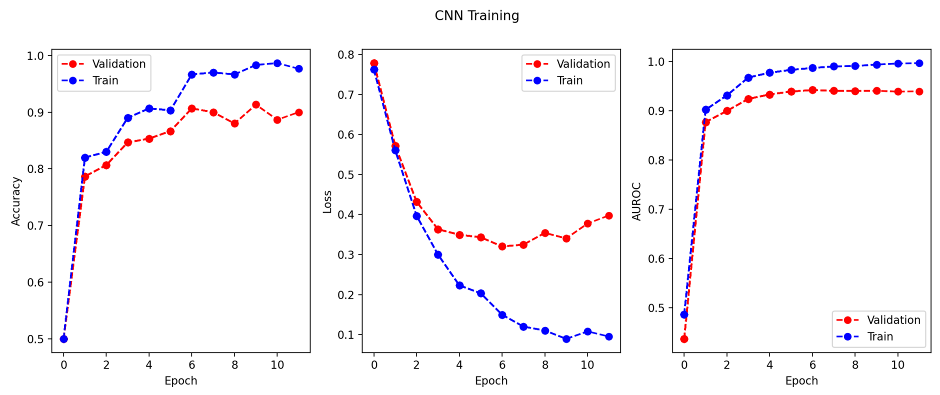


Solution: Sources of noise in validation loss:

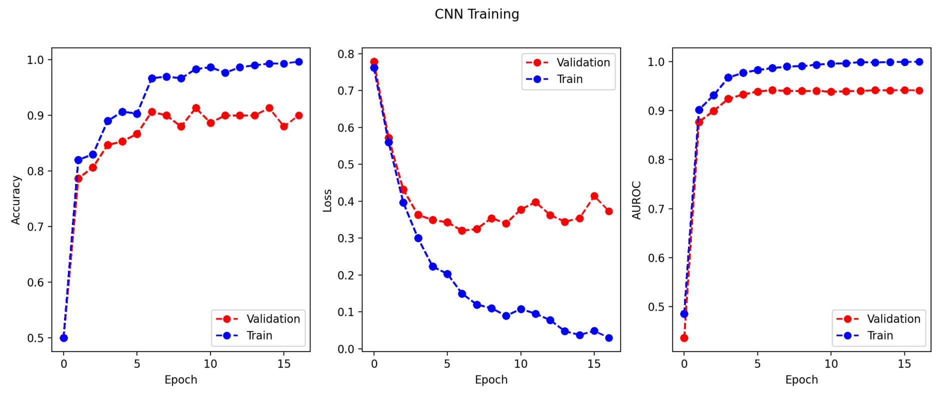
- From data: the data itself has multiple sources of noise: luminance (i.e., a mean different from the mean of the training data) or color balance.
- From the optimization procedure: ADAM, a variant of SGD is stochastic. Therefore, each update step will not necessarily decrease the training loss.
- Random shuffling training data at the end of each epoch.
- We directly optimize for training loss, but not validation loss. Therefore, even if we used true gradient descent on training data, validation loss has no guarantees of monotonic behavior.

ii. (6 points) Report the epoch that your model stopped training at with using a patience of **5**. Try increasing your patience to **10**. **Answer** the following: What epoch does your model stop training at now? Based on the training graphs, which patience value works better and why? In what scenario might increased patience be better? **Include** the training plot generated by using a patience of 5 and the one with a patience of 10 in your answer.

Solution: Example plot with patience = 5



Corresponding Example plot with patience = 10



Example Answers to analysis questions are based on first graph in each set above.

- Epoch when model stopped training for a patience of **5**: 11
- Epoch when model stopped training for a patience of **10**: 16
- When using patience, we know the best epoch (lowest validation loss) is the epoch number: final epoch number - patience. In the graphs above, this means both models select the same epoch as the best model. So, an increased patience did **not** improve our model and it was better to stick with a patience of 5.
- Answers will vary. For example, any time you see your objective function (such as minimizing loss) has not converged to a value such that it can still be optimized, you should increase your patience. This could perhaps happen when you have a task that is difficult to get exactly correct such that more epochs are required to minimize loss.

iii. (9 points) Let's explore how changing our model architecture affects performance. Change the number of output filters in the last convolutional layer from **8** to **64** and update the input dimension of the fully connected layer accordingly. What's the new size of the input to the fully connected layer? **Report** the new size and update the architecture in `model/target.py`. Using a patience of **5**, train the model again with the updated architecture starting from epoch 0. **Complete** the table below, reporting the area under the receiver operating characteristic (AUROC) curve on the validation and training set for the model (i.e., epoch) **associated with the lowest validation loss** (note: this is not necessarily the model associated with the last epoch). How does performance vary as we increase the number of filters from 8 to 64? What might account for this change? **Answer** these questions in a few sentences.

	Epoch	**Training AUROC**	**Validation AUROC**
8 filters			
64 filters			

Solution:

- New size of input to FC layer: **256**
- Some observations about the changes with 64 filter graphs
  - Slightly larger gap between train and validation performance in all graphs
  - Smoother curves
  - Trained for fewer epochs with same patience value
- Some reasoning for each of the above changes
  - Model has more parameters now so it can easily overfit to training data and may not generalize as well to the validation data, thus we observe a larger gap in performance.
  - We converge more quickly with more parameters.
  - Converging more quickly means early stopping will kick in sooner so we train for fewer epochs with more parameters.

	Epoch	**Training AUROC**	**Validation AUROC**
8 filters	6	0.9867	0.9419
64 filters	2	0.9777	0.923

Note: table answers will vary, but AUROC values should be ≥ 0.90

(g) **Execute** `test_cnn.py` to measure the performance of your trained CNN on the heldout test set. `test_cnn.py` measures the performance of your model using both accuracy and AUROC scores on the training data, validation data, and test data. The model being evaluated in this section will be our original trained CNN from part 2(f) with a patience of **5** and with **8 filters** in the last convolutional layer. You will need to train a new CNN (execute `train_cnn.py`) with this architecture if the most recent model you trained doesn't match these specifications. Make sure to load your trained model using the number of epochs you chose in the part 2(f)(iii).

i. (3 points) **Include** your CNN's scores in the table below.

	Training	**Validation**	**Testing**
Accuracy			
AUROC			

Solution:

	Training	**Validation**	**Testing**
Accuracy	0.9667	0.9067	0.61
AUROC	0.9867	0.9419	0.6548

ii. (2 points) We know that if our training performance far exceeds our validation performance it means that we may have overfit to the training data. **Compare** the training and validation performance. Do we see any evidence of overfitting here?

Solution: The training and validation performances are similar though validation performance is slightly worse suggesting little to no overfitting.

iii. (3 points) Often, we assume that our training and validation data are representative of our testing data, so that we can make correct decisions regarding our model. **Compare** the validation and test performance. Ideally, they would be similar. What trend do you see here? Hypothesize a possible explanation for such a trend.

Solution: The testing performance is far lower than the validation performance. This means our training/validation set was **not** a good representation of our testing set. Many possible hypothesis answers. For example, the validation set may include some bias that is not present in the test set, i.e. the validation images all have some feature that is not heavily present in the testing set - like green backgrounds!

3 Visualizing what the CNN has learned [10 pts]

While CNNs perform well in a variety of tasks, they lack interpretability. To ensure confidence in our models, it is important to understand *why they predict what they predict*. Fortunately, approaches for explaining the reasoning behind neural networks is an active area research. In this section, we will use Gradient-weighted Class Activation Mapping (Grad-CAM) [1] to visualize which regions of an image contribute most to the final classification.

In a convolutional layer, every filter has different weights and generates a distinct activation map given the same input. Activation maps represent the output of a convolutional layer. For example, the first convolutional layer has 16 filters, thus we have 16 different activation maps. Grad-CAM utilizes these activation maps to assign importance to individual neurons in determining a specific class-label. It computes the gradient of the class score (i.e., Collies) with respect to each activation map and assigns importance weights to each activation map. Grad-CAM then computes the sum of all activation maps, weighted by their importance, to obtain a heatmap of areas in the original image that strongly correspond to the class-label.

We can plot this heatmap on top of the original image (with some upsampling if necessary) to visualize the regions the CNN uses to classify images. This computation can be performed for any of the convolutional layers in order to determine which features are important at a particular depth. You can imagine that the earlier convolutional layers will be highlighting important low level structure while the later layers will highlight the important semantics in the image. Below are two examples from the original paper of Grad-CAM being used.

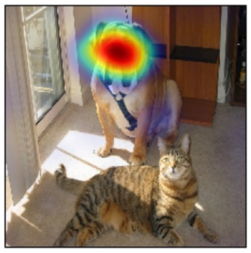


Figure 3: Grad-CAM for the ‘dog’ label

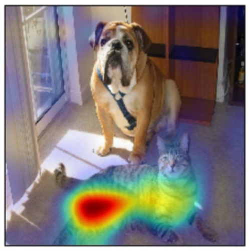


Figure 4: Grad-CAM for the ‘cat’ label

(a) (5 points) Let us suppose that we have a network where at a particular convolutional layer, we produce two activation maps, $A^{(1)}$ and $A^{(2)}$ which are both in $\mathbb{R}^{4\times 4}$. Additionally, we have calculated $\frac{\partial y^1}{\partial A_{ij}^{(k)}}$ for $k \in \{1, 2\}$ and $1 \leq i, j \leq 4$. What is L^1, the output from the Grad-CAM algorithm for label 1 and this particular convolutional layer? Please review the paper at the following link for the mathematical definition of L^1 to answer this question. You may use $Z = 16$.

$$A^{(1)} = \begin{bmatrix} 1 & 1 & 2 & 1 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 1 & -2 & -2 \end{bmatrix}, \frac{\partial y^1}{\partial A^{(1)}} = \begin{bmatrix} 0 & -1 & 0 & 1 \\ -2 & -1 & 0 & 0 \\ -1 & 0 & 0 & 1 \\ 1 & 1 & 2 & 2 \end{bmatrix}$$

[1]Selvaraju, R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2019). Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based LocalizationInternational Journal of Computer Vision, 128(2), 336–359.

$$A^{(2)} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 1 & 0 \\ -1 & -1 & -1 & 0 \end{bmatrix}, \frac{\partial y^1}{\partial A^{(2)}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 2 & 2 & 2 & 2 \\ 0 & 0 & 1 & 1 \\ -1 & -2 & -1 & 0 \end{bmatrix}$$

Solution: Calculating L^1:

$$\alpha_1^1 = \frac{1}{16} \sum_{i=1}^{4} \sum_{j=1}^{4} \frac{\partial y^1}{\partial A_{ij}^{(1)}} = \frac{3}{16}$$

$$\alpha_2^1 = \frac{1}{16} \sum_{i=1}^{4} \sum_{j=1}^{4} \frac{\partial y^1}{\partial A_{ij}^{(2)}} = \frac{7}{16}$$

$$L^1 = ReLU(\alpha_1^1 * A^{(1)} + \alpha_2^1 * A^{(2)})$$

$$L^1 = ReLU(\begin{bmatrix} 10 & 10 & 13 & 10 \\ 17 & 20 & 17 & 14 \\ 14 & 17 & 7 & -3 \\ -7 & -4 & -13 & -6 \end{bmatrix} * \frac{1}{16})$$

$$L^1 = \begin{bmatrix} 10 & 10 & 13 & 10 \\ 17 & 20 & 17 & 14 \\ 14 & 17 & 7 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} * \frac{1}{16}$$

(b) (3 points) **Execute** `visualize_cnn.py` to generate visualizations of the first convolutional layer using Grad-CAM (done on the training set). Please use the model used in 2g (with the same epoch from 2f(iii)). Recall that our CNN is trying to classify Collies vs Golden Retrievers. **What** features does the CNN appear to be using to identify the Collie class? We use the Viridis color scale (in contrast to the figures above), so lighter colors have higher values.

Solution: The model seems to be making classification decisions using background features, and often times fails to activate on the dog at all. Many of the images are segmented such that the collie classification is influenced almost entirely by the grass background.

(c) (2 points) Let's revisit our findings from 2g. Do the Grad-Cam visualizations confirm your hypothesis? Why or why not? Make sure to check that your architecture matches what is given in Appendix B.

Solution: Since the model is making decisions based on the background and not the actual dog features, we hypothesize that the background is highly correlated with the label, and as a result our model has learned to use this to classify the dogs. However, since our test set performance is so much lower, it seems likely that this shortcut does not generalize well to the test set, possibly because the backgrounds from the training/validation set are not representative of the test set. In other words, since our model did not actually learn to distinguish breeds and instead learned to classify based on background, it is not generalizable to other scenarios. This confirms our hypothesis that the training/validation set are not representative of the test set, since our learned shortcut does not transfer well

4 Transfer Learning & Data Augmentation [25 pts]

From our results in Sections 2 and 3, there appears to be something up with the data. Oh no! In this section we will explore two strategies that could help us improve our performance on the test set: transfer learning and data augmentation.

4.1 Transfer Learning

In settings with only limited training data, we may benefit from leveraging auxiliary data that isn't directly involved with our task. For example, suppose we want to classify cars versus bicycles, but have a large auxiliary dataset of trucks versus pedestrians. In such a scenario, we refer to the task of interest as the *target* task and the auxiliary task as the *source* task.

When the target and source tasks are related, we might seek to *transfer* knowledge about the source task to the target task. For example, the features useful for distinguishing trucks from people may also be useful in distinguishing cars from bicycles (e.g., number of wheels, placement on road). There are many different approaches for transferring knowledge from the source to the target task. But a common approach involves training a model on the source task and then transferring the learned parameters to the target task.

The parameters of the source task model may serve as a good initialization for the target task model, or we can "freeze" the learned parameters and only update the parameters associated with the output layer. When everything except the output layer is frozen you have essentially learned a "feature extractor" on the source task and are applying it to the target task data. Then the only learning you are doing with respect to the target task is learning how these features combine to make target predictions.

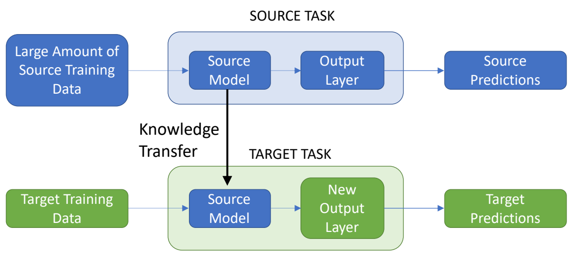


Figure 5: In settings with limited training data we may be able to transfer knowledge from a related but distinct source task with large amounts of training data.

Here, our target task is the binary classification task of distinguishing Collies from Golden Retrievers. However, we have labeled training data from 8 other breeds: Samoyed, Miniature Poodle, Saint Bernard, Great Dane, Dalmatian, Chihuahua, Siberian Husky, and Yorkshire Terrier. We hypothesize that the learned representations, or features, useful for solving this source task will transfer to our target task. In the steps below, we will explore a transfer learning approach to leveraging these source data.

(a) Study the architectures presented in Appendix B (page 27) and C (page 30). These architectures will correspond to the architecture of the target and source models.

Implement the Source model architecture by completing the `Source` class in `model/source.py`. Follow the same steps as implementation for the `Target` from Section 2. Be sure not to change the name of fully connected layers from the original code in `source.py` and `target.py`. The names of these layers are intentionally different to work with the way that PyTorch restores model parameters from checkpoints.

(b) **Review** the script `train_source.py`. Based on the model specification in Appendix C, **fill in** the definitions for `criterion` and `optimizer`.

(c) (2 points) **Run** the script `train_source.py` to train the network with a patience of 10 on the 8 classes of training data and evaluate on the validation set. On a standard laptop, this should take about 3 minutes. **Include** the final training plot. Select an epoch number corresponding to the model with the lowest validation loss. This is the model we will transfer to our target task. **Report** the epoch number and use this value for the rest of the transfer learning questions.

Solution: An example of a potential solution plot.

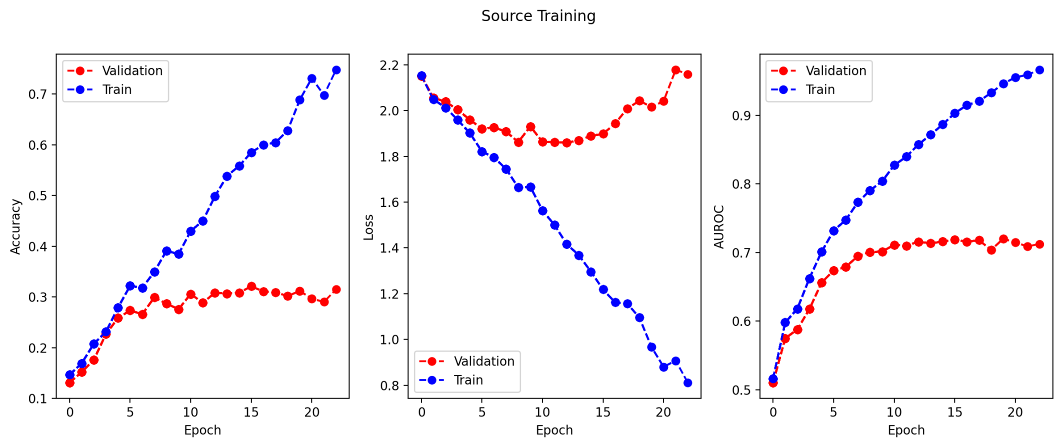
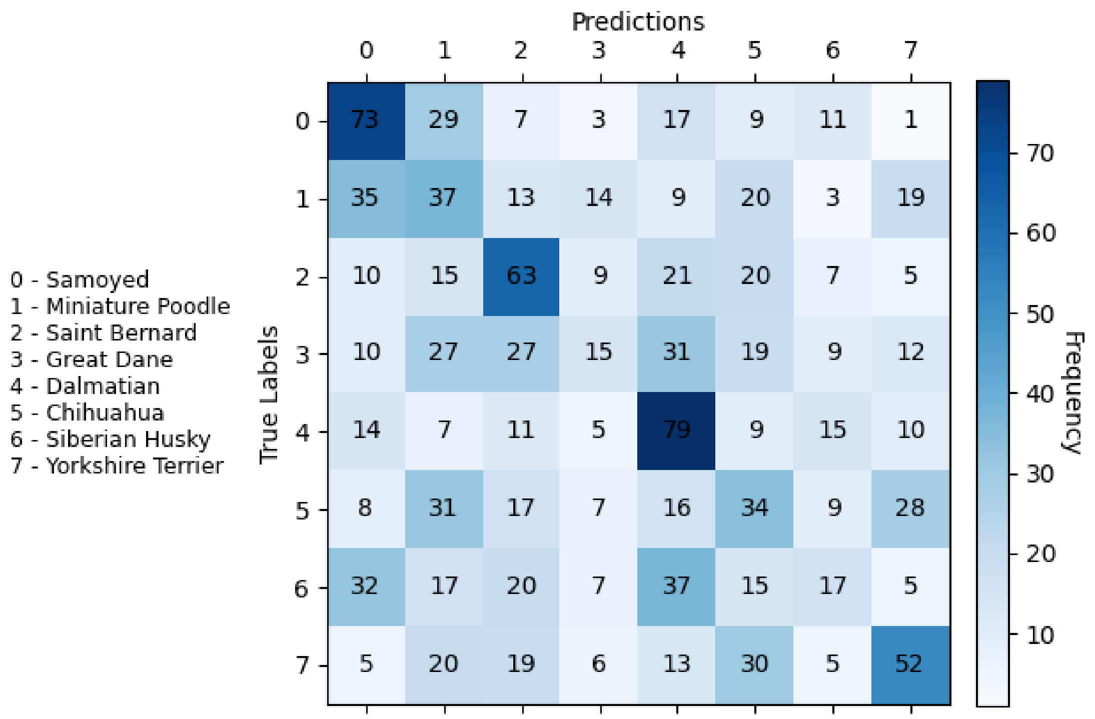


Figure 6: Source Training Plot

Lowest validation loss epoch: **12**

(d) (4 points) **Run** the script `confusion_matrix.py` to visualize the prediction accuracy of each class on the validation dataset. Choose the epoch you reported in part 4.1.c when prompted. **Include** the plot in your write up. **Examine** for which breed(s) the classifier is the most accurate. For which breed(s) is the classifier the least accurate? **Briefly explain** why this might be the case.

Solution:



	0	1	2	3	4	5	6	7
0	73	29	7	3	17	9	11	1
1	35	37	13	14	9	20	3	19
2	10	15	63	9	21	20	7	5
3	10	27	27	15	31	19	9	12
4	14	7	11	5	79	9	15	10
5	8	31	17	7	16	34	9	28
6	32	17	20	7	37	15	17	5
7	5	20	19	6	13	30	5	52

Figure 7: Confusion Matrix

Samoyed and Dalmatians were the most accurate, while Great Danes and Siberian Huskies were the most incorrectly classified. Potential reasons for why could include but is not limited to: the presence of distinctive features in some breeds (e.g. spotted coats on Dalmatians) making them easier to classify; the model favoring predicting only certain labels (e.g. everything is predicted to be a Samoyed or Dalmatian over other similar looking breeds); the coloring of certain breeds being similar (e.g. Dalmatians, black and white Huskies, and Samoyeds); or certain breeds looking very similar at low-resolution images (e.g. poodles and terriers).

(e) In Section 2, we randomly initialized the weights of the CNN. Here, we will leverage the model learned in part (c) to initialize the weights. We can then selectively freeze or fine-tune different parts of our network (e.g., freezing all convolutional layers, freezing only the first two convolutional layers, etc.).

Review the script `train_target.py`. Based on the model specification in Appendix B, **fill in** the definitions for `criterion`, `optimizer`, and `patience`. **Implement** the `freeze_layers` function to modify the model with the given number of layers frozen. Note that Python is pass-by-reference, so modifying the model parameter also modifies the original model. Additionally, loop through `model.named_parameters()` to determine which convolutional layers to freeze.

(f) (8 points) **Review** and **run** the script `train_target.py` with different number of layers frozen to train our classifier to fit the binary task. **Compare** this classifier to your previous CNN classifier from 2.g.i. in terms of the train, validation and test AUROC for the model associated with the lowest validation loss. Format your results as follows:

	AUROC		
	TRAIN	**VAL**	**TEST**
Freeze all CONV layers (Fine-tune FC layer)			
Freeze first two CONV layers (Fine-tune last CONV and FC layers)			
Freeze first CONV layer (Fine-tune last 2 conv. and fc layers)			
Freeze no layers (Fine-tune all layers)			
No Pretraining or Transfer Learning (Section 2 performance)			

Examine the results of using transfer learning. Compared to the CNN in Section 2 that did not leverage the source task data, does transfer learning help? Was the source task helpful? Why do you hypothesize it was (or wasn't)? Examine the training curves. How does freezing all convolutional layers compare to freezing just a subset, versus freezing none? Why do you think this might occur?

Solution:

	AUROC		
	TRAIN	**VAL**	**TEST**
Freeze all CONV layers	0.8731	0.8736	0.8368
Freeze first 2 CONV layers	0.9688	0.9047	0.8012
Freeze first CONV layer	0.9866	0.922	0.7696
Freeze no layers	0.9927	0.9337	0.762
No Pretraining or Transfer	0.9867	0.9419	0.6548

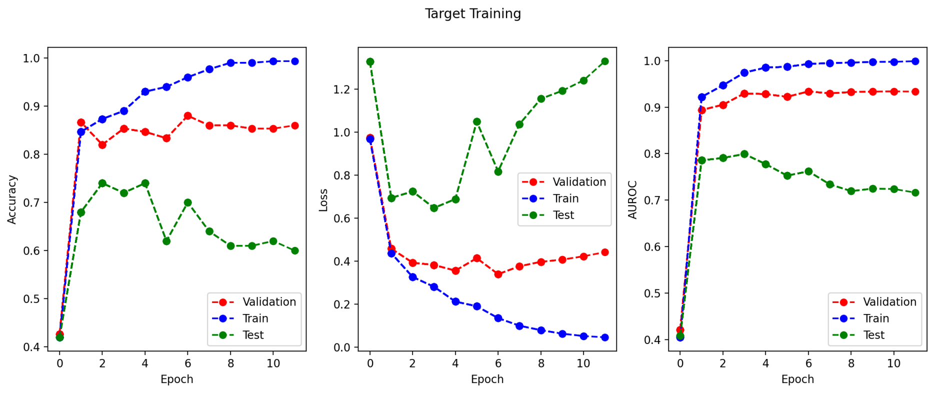


Figure 8: No Layers Frozen

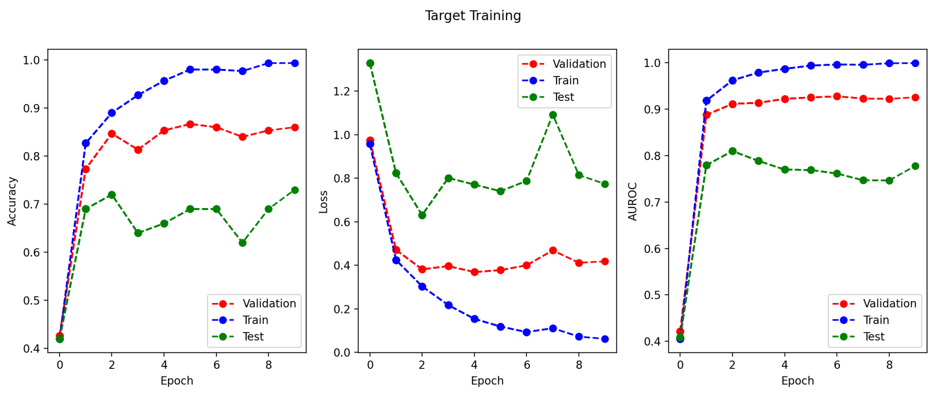


Figure 9: First CONV layer frozen

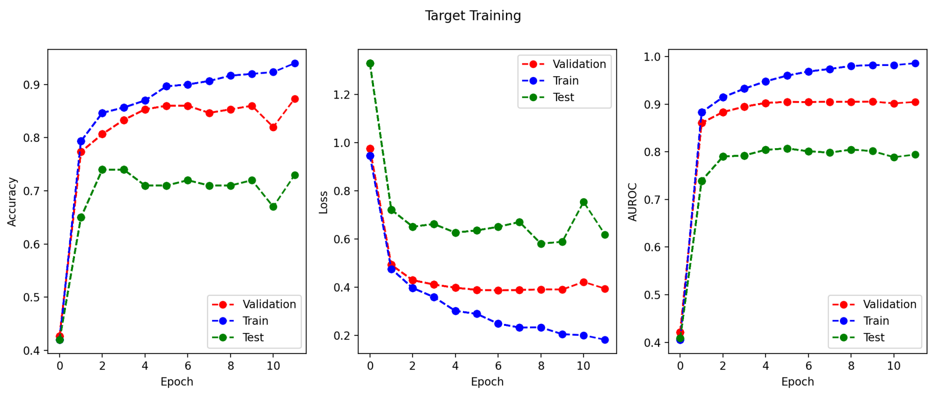


Figure 10: First two CONV layers frozen

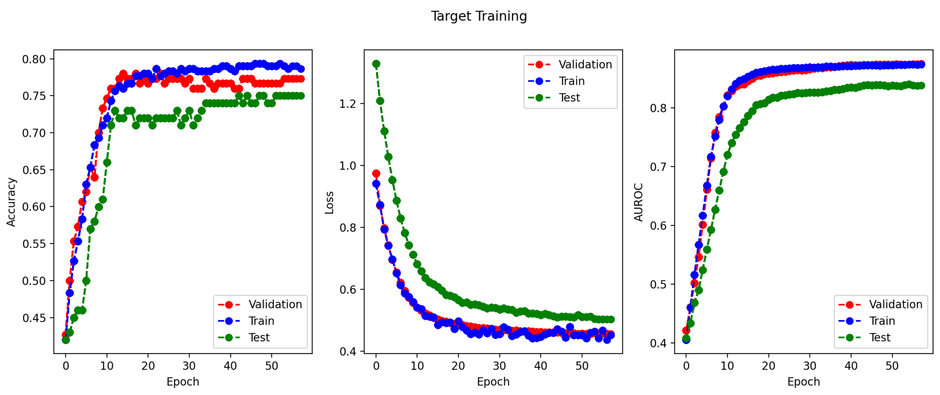


Figure 11: All CONV layers frozen

As we freeze more layers, the test AUROC generally increases. This is due to the model no longer

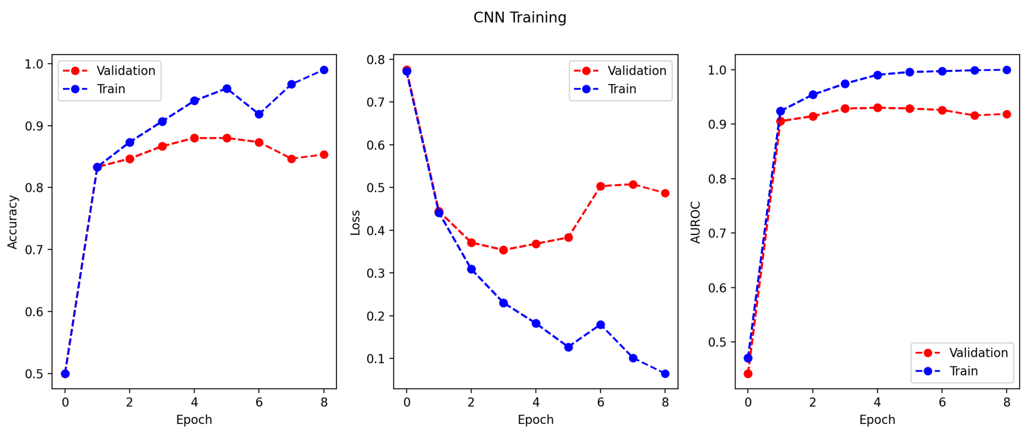


Figure 12: Training plot for grayscale augmented data (with originals)

being able to learn the bias of the background in classification. The train and validation AUROC tend to decrease, but still hover at very high values since the model is already trained on similar task of differentiating other dog breeds. The transfer learning produced better results, even with no layers frozen, than the section 2 performance in test AUROC, indicating that the initialization itself actually helped performance.

4.2 Data Augmentation

It can be difficult to accurately capture invariances in a task when learning with limited training data. An invariance corresponds to a transformation that when applied to the input data does not change the label - e.g., a Collie is a Collie no matter *where* it appears in the image. To more accurately capture such invariances we can transform our training data, synthesizing new data. This is called 'Data Augmentation'. As a result of Data Augmentation, we will have transformed data to train on, in addition to our original data. In this part, you will apply both `grayscale` and `rotation` data augmentation methods to the training dataset.

Once the data have been augmented, you will train the CNN from Section 2 on the augmented dataset, and report the performance relative to the CNN model trained without data augmentation.

Important note: Data augmentation is only used to augment the data in training set (make sure you understand why), so avoid applying it to test/validation sets.

(a) **Complete** `Rotate` and `Grayscale` in `augment_data.py`. When implementing `Rotate`, use a random degree value from the range `(-deg, deg)` to rotate the image.

(b) Now let's experiment training on augmented data. For each configuration specified in the table below in part (c):

   i. Generate the corresponding augmented dataset by **specifying** in `augment_data.py` which augmentations to apply and whether to keep the original image, then **running** the augmentation script:

```
python augment_data.py data/dogs.csv data
```

This will output `data/augmented_dogs.csv` and `data/augmented/`. You may view the images in `data/augmented/` to verify your results.

ii. **Run** the `train_cnn.py` script to train your CNN on the augmented data, again with 8 filters in the last convolutional layers and a patience of 5. Then, **run** `test_cnn.py`, selecting the epoch with lowest validation loss, to see how your trained model performs on all splits.

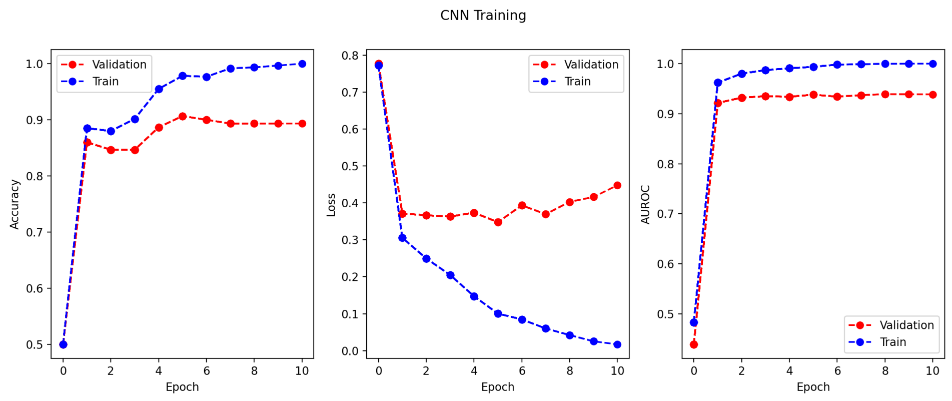
iii. (8 points) **Include** in your writeup training plots of the train and validation splits and **fill in** the AUROC for the train, validation and test splits. See Figure 12 for an example training plot.

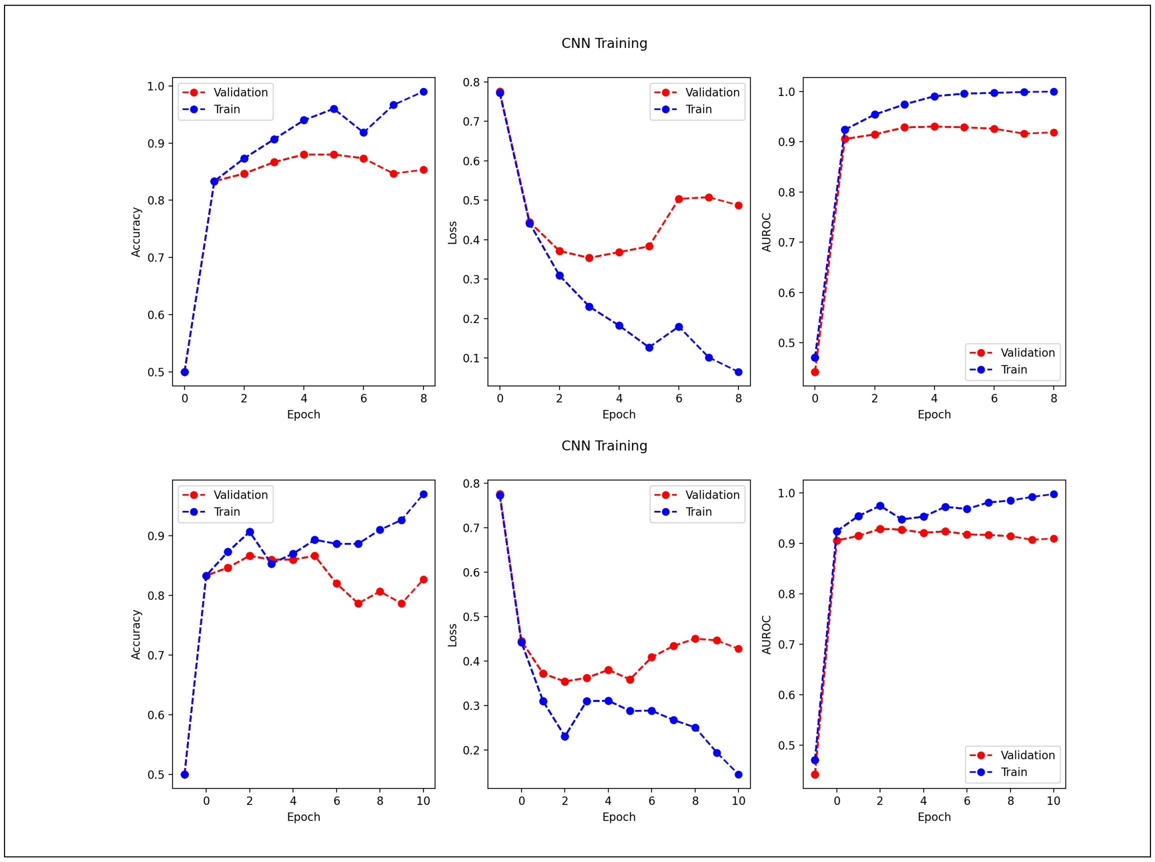
	AUROC		
	TRAIN	**VAL**	**TEST**
Rotation (keep original)			
Grayscale (keep original)			
Grayscale (discard original)			
No augmentation (Section 2 performance)			

Solution: Exact curves and values may vary due to random initializations.

	AUROC		
	TRAIN	**VAL**	**TEST**
Rotation (keep original)	0.9941	0.938	0.6672
Grayscale (keep original)	0.9914	0.9271	0.74
Grayscale (discard original)	0.9952	0.9214	0.756
No augmentation (Section 2 performance)	0.9867	0.9419	0.6548

From top to bottom: Rotation, Grayscale (keep), Grayscale (discard)





(c) (3 points) **Answer** the following questions: How do the validation and training plots change as you apply these augmentation techniques? What do you hypothesize is the reason for this change?

Solution: Both the Rotation and Grayscale (keep original) training plots should look similar to the original CNN training plots (i.e. the validation and training AUROC are both high) because they both still allow for learning the shortcut. The Grayscale (discard original) training plot should have significantly worse validation performance compared to training performance because the shortcut can no longer be learned, so the model must learn to actually discern between the dogs.

5 Challenge [15 pts]

Armed with your knowledge of neural networks and PyTorch, you are now ready to take on Karl's challenge: **designing**, **implementing**, and **training** a deep neural network of your own design to classify dog breeds! We will again restrict our final classifier to Collies and Golden Retrievers; however, you are allowed to use the training and validation sets for all 10 classes to train your model. These additional data may be useful if you wish to further explore transfer learning in your challenge solution. In addition, you will have *full control* over the network architecture, including weight initialization, filter sizes, activation function, network depth, regularization techniques, preprocessing methods, and any other changes you see fit. You will be evaluated on a heldout challenge set drawn from the same distribution as the test set used previously in the project. We ask you to adhere to the following guidelines when developing your challenge submission.

- You *must* use only the data provided for this challenge; *do not attempt to acquire any additional training data, or modify the ground truth test labels. This includes data in the dataset we give you that is not in the training split!* However, you can apply data augmentation. Additionally, while you may choose to replicate or modify existing architectures or use image preprocessing functions, you must train all models yourself. Do not use pretrained models. Note your final model must be a CNN.
- **Implement** your model within `model/challenge.py`.
- **Fill in** the definitions for the loss function, optimizer, and model.
- You may alter the training framework in `train_challenge.py` as you please.
- If you wish to modify any portion of the data batching or preprocessing, please make a copy of `dataset.py` and call it `dataset_challenge.py`. You may then modify this file however you choose.
- You may add additional variables to the `config.json` file and modify any of the existing challenge variables as you see fit.

Once your model has been trained, ensure that its checkpoint exists in the directory `checkpoints/challenge/` and **run** the script `predict_challenge.py` as follows.

```
python predict_challenge.py --uniqname=<your_uniqname>
```

This script will load your model from your saved checkpoint and produce the file `<your_uniqname>.csv` that contains your model's predictions on the test set.

Double check that your final output csv file has float-value predictions (not binary 0/1 labelling) for each of the 200 images in the challenge dataset before you submit to Canvas.

In your write-up, **describe** in detail the experiments you conducted and how they influenced your final model. When grading your challenge write-up, we will look for discussion of your decisions regarding the following. If you chose not to implement some of the approaches, add a short discussion on methods you potentially could have done.

- Regularization (weight decay, dropout, etc.)

- Feature selection
- Model architecture
- Hyperparameters
- Transfer learning
- Data augmentation
- Model evaluation (i.e., the criteria you used to determine which model is best)
- Whether your model was trained using GPU hardware

We will evaluate your submission based on two components:

1. (9 points) Effort: We will evaluate how much effort you have applied to this problem based on the experiments described in your write-up and your code submission.

   Solution: Answers will vary.

2. (6 points) AUROC: We will evaluate the AUROC of your classifier's predictions on the ground truth challenge labels. Note that this means that the output test predictions follow a specified order. For that reason, please **do not shuffle the challenge data points during prediction.**

   Solution: Challenge graded separately on Canvas.

REMEMBER to submit your project report by 10:00pm ET on March 23rd, 2022 to Gradescope.

Include your code as an appendix (copy and pasted) in your report. Please try to format lines of code so they are visible within the pages.

Upload your file `uniqname.csv` containing the label predictions for the held-out data to the canvas assignment named Project 2 Challenge Submissions.

Figure 13: Some of Karl's dogs back home safe and sound

Appendix A Implementation notes

A.1 `torch.nn.Conv2d`

This class implements the 2D convolutional layer we have learned in lecture. Create the layer as follows: `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)`. Use default settings for all other arguments. For example `dilation=1` means no dilation is used, and `bias=True` adds a learnable bias term to the neuron.

A.2 the `SAME` padding

With Pytorch's default setting `padding=0`, if we apply a 3×3 convolutional filter to a 4×4 input image, the output would be 2×2. As we keep applying convolutional layers in this way, the spatial dimensions will keep decreasing. However, in the early layers of a CNN, we want to preserve as much information about the original input volume so that we can extract those low level features. To do this, we can pad the borders of the input image with zeros in a way such that the output dimension is the `SAME` as the input (assuming unit strides). If the filter size is odd $k = 2m + 1$, then this amount of zero padding on each side is $p = \lfloor k/2 \rfloor = m$. For example, in Figure 14, since the filter size is $k = 3$, the appropriate padding size is $p = \lfloor 3/2 \rfloor = 1$.

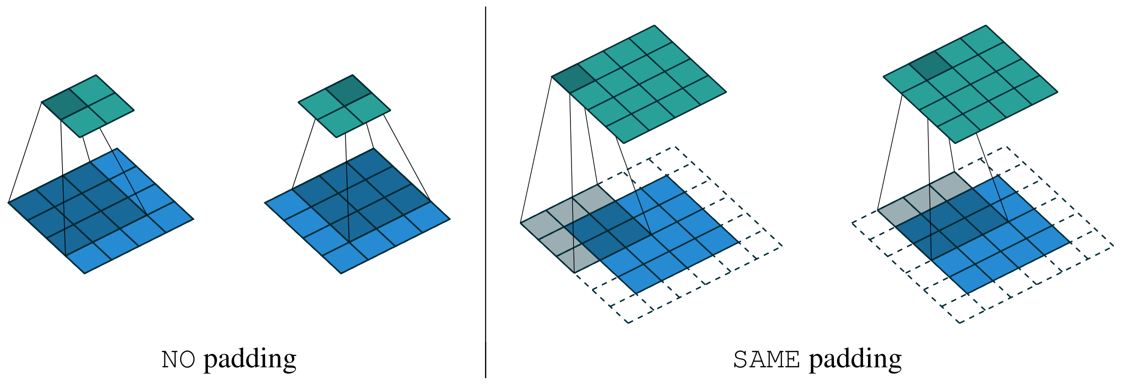


Figure 14: Comparison of padding schemes.
Adapted from: Vincent Dumoulin, Francesco Visin - A guide to convolution arithmetic for deep learning.
Source code: https://github.com/vdumoulin/conv_arithmetic

A.3 `torch.nn.CrossEntropyLoss`

Let $\bar{z}$ be the network output logits and $\hat{y}_c$ the probability that the network assigns to a single input example that belongs to class c. Let D be the number of output classes, and $\bar{y} \in \mathbb{R}^D$ be a vector with a one in the position of the true label and zero otherwise (i.e., if the true label is t, then $\bar{y}_c = 1$ if $t = c$ and $\bar{y}_c = 0$ if $t \neq c$). This is known as *one hot encoding*.

For a multiclass classification problem, we typically apply a soft max activation at the output layer to generate a probability distribution vector. Each entry of this vector can be computed as:

$$\hat{y}_c = \frac{\exp(z_c)}{\sum_j \exp(z_j)}$$

The soft max activation guarantees that each component is in the range between 0 and 1, and that entries sum to 1. We then compare this probability distribution $\hat{y}$ with the ground truth distribution $\bar{y}$ (a one-hot vector), computing the cross entropy loss, $\mathcal{L}$:

$$\mathcal{L}(\bar{y}, \hat{y}) = -\sum_c \bar{y}_c \log \hat{y}_c$$

These two steps can be done at once in PyTorch using the `CrossEntropyLoss` class, which combines the softmax activation with negative log likelihood loss. We don't need to add a separate soft max activation at the output layer. This improves computational efficiency and numerical stability.

Appendix B Target Architecture

Note that the dimensions given here match the expected input of Pytorch documentation and may be slightly different from the dimensions used in discussion.

Training Parameters

- Criterion: `torch.nn.CrossEntropyLoss`
- Optimizer: `torch.optim.Adam`
- Learning rate: 10^{-3}
- Patience: 5
- Batch Size: 32

Architecture

Layer 0: Input image

- Output: $3 \times 64 \times 64$

Layer 1: Convolutional Layer 1

- Number of filters: 16
- Filter size: 5×5
- Stride size: 2×2
- Padding: SAME
- Activation: ReLU
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{5 \times 5 \times num\_input\_channels}$
- Bias initialization: constant 0.0
- Output: $16 \times 32 \times 32$

Layer 2: Max Pooling Layer

- Kernel size: 2×2
- Stride: 2×2
- Padding: none

Layer 3: Convolutional Layer 2

- Number of filters: 64
- Filter size: 5×5
- Stride size: 2×2
- Padding: SAME
- Activation: ReLU
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{5 \times 5 \times num\_input\_channels}$
- Bias initialization: constant 0.0
- Output: $64 \times 8 \times 8$

Layer 4: Max Pooling Layer

- Kernel size: 2×2
- Stride: 2×2
- Padding: none

Layer 5: Convolutional Layer 3

- Number of filters: 8
- Filter size: 5×5
- Stride size: 2×2
- Padding: SAME
- Activation: ReLU
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{5 \times 5 \times num\_input\_channels}$
- Bias initialization: constant 0.0
- Output: $8 \times 2 \times 2$

Layer 6: Fully connected layer 1 (Output layer)

- Input: 32
- Activation: None
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{input\_size}$
- Bias initialization: constant 0.0
- Output: 2 (number of classes)

Appendix C Source Model Architecture

Note that the dimensions given here match the expected input of Pytorch documentation and may be slightly different from the dimensions used in discussion.

Training Parameters

- Criterion: `torch.nn.CrossEntropyLoss`
- Optimizer: `torch.optim.Adam` with 0.01 weight decay
- Learning rate: 10^{-3}
- Patience: 10
- Batch Size: 64

Architecture

Layer 0: Input image

- Output: $3 \times 64 \times 64$

Layer 1: Convolutional Layer 1

- Number of filters: 16
- Filter size: 5×5
- Stride size: 2×2
- Padding: SAME
- Activation: ReLU
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{5 \times 5 \times num\_input\_channels}$
- Bias initialization: constant 0.0
- Output: $16 \times 32 \times 32$

Layer 2: Max Pooling Layer

- Kernel size: 2×2
- Stride: 2×2
- Padding: none

Layer 3: Convolutional Layer 2

- Number of filters: 64
- Filter size: 5×5
- Stride size: 2×2

- Padding: SAME
- Activation: ReLU
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{5 \times 5 \times num\_input\_channels}$
- Bias initialization: constant 0.0
- Output: $64 \times 8 \times 8$

Layer 4: Max Pooling Layer

- Kernel size: 2×2
- Stride: 2×2
- Padding: none

Layer 5: Convolutional Layer 3

- Number of filters: 8
- Filter size: 5×5
- Stride size: 2×2
- Padding: SAME
- Activation: ReLU
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{5 \times 5 \times num\_input\_channels}$
- Bias initialization: constant 0.0
- Output: $8 \times 2 \times 2$

Layer 6: Fully connected layer 1

- Input: 32
- Activation: None
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{input\_size}$
- Bias initialization: constant 0.0
- Output: 8