EMORY UNIVERSITY
Department of Computer Science
CS 334 Section 2 — Machine Learning
Fall 2024

Homework 5: The Convoluted Case of Ken's Canine Companions
Issued: Fri. 11/08, Due: Fri. 11/22 at 11:59pm

Dog-gone! The neighborhood kennel owner, Ken, is having a "ruff" time: during his morning rounds to check on his canine companions, he found to his horror that they had all dug their way under the fence and escaped! Knowing how much these dogs like to play tricks on him, he thought of a trick of his own to help find his furry friends: placing cameras around town and using image recognition techniques to detect passing dogs. He's come to the CS-334-2 team in search of help implementing this system.

Ken's kennel houses dogs from 10 distinct breeds: Samoyeds, Miniature Poodles, Golden Retrievers, Great Danes, Dalmatians, Collies, Siberian Huskies, Yorkshire Terriers, Chihuahuas, and Saint Bernards. He has provided us with images of all of his dogs — a few of which are shown below.

Figure 1: Sample images from the the dataset.

Submission Instructions: The homework is due on Gradescope in **two** parts.

- **Upload PDF to HW5-Written**: Your submission may be typed or handwritten, and pages must be tagged with relevant parts on Gradescope. To help you navigate different questions of this homework, we've highlighted question parts that should be included in the write-up.
- **Submit `challenge.csv` and your code to HW5-Code&Challenge**: Upload your predictions for the held-out data in a single csv. You may submit multiple times; only the last submission will be considered. Code is not auto-graded for this assignment. Please include the following ***SIGNED*** *honor code statement*:

```
THIS CODE IS MY OWN WORK, IT WAS WRITTEN WITHOUT CONSULTING CODE
WRITTEN BY OTHER STUDENTS OR LARGE LANGUAGE MODELS SUCH AS CHATGPT.
/* Your_Name_Here */
I collaborated with the following classmates for this homework:
<names of classmates>
```

The main part of this HW requires you to write <30 lines of code. We've designed the HW to run in a reasonable time (<5min) on standard CPU machines; however, if you have access to a compatible GPU and wish to use it, please specify if you have used any CUDA/MPS GPU device. Challenge submissions that have used GPU hardware, additional data, or pretrained models will be graded separately.

Getting Started

Download `HW5_SkeletonCode.zip` from Canvas. This zip file contains the full project dataset, and may take several minutes to download. After unzipping, please ensure the all of the following files are present:

Data-related:
- `data/dogs.csv`
- `data/images/`
- `data.py`
- `visualize_data.py`

Main scripts (Q2):
- `model.py`
- `train_cnn.py`
- `visualize_cnn.py`

Challenge scripts (Q3):
- `challenge_model.py`
- `challenge_train.py`
- `challenge_predict.py`

Helper files:
- `requirements.txt`
- `config.json`
- `utils.py`

Dataset

This dataset contains 12,775 PNG image files of 10 different dog breeds. These images are in `data/images/` and named as `000001.png` through `012775.png`. Each image contains 3 color channels (red, green, blue) and is of size $64 \times 64 \times 3$. These images have already been divided into 3 class-balanced data partitions: a training set, a validation set, and a test set. The metadata containing the label and data partition for each image is documented in `data/dogs.csv`.

1 Data Preprocessing [20 pts]

Real datasets often contain numerous imperfections (e.g., wrong labels, noise, or irrelevant examples). To develop a dataset more conducive for training, we've already implemented the preprocessing steps for you in `data.py`. We first use `resize()` to downsample the images from $64 \times 64 \times 3$ to $32 \times 32 \times 3$, and then use `ImageStandardizer` to standardize the images using the mean and standard deviation of pixel values from training data. These will serve two purposes: speeding up computation time and highlighting the underlying structure of the image dataset.

While you don't need to write any code for this question, you are encouraged to study the implementation of the preprocessing functions as well as how they are used. Then, answer the following questions:

(a) [6pts] **Run** `data.py` and report the mean and standard deviation of each color channel (RGB), based on the entire training partition.

(b) [2pts] Why do we extract the per-channel image mean and standard deviation from the training set as opposed to the other data partitions?

(c) [12pts] **Run** `visualize_data.py` to see the effects of this preprocessing on some example images. Save and include at least 3 sets of images in your write-up (each set is 4 pairs). What are the visible effects of preprocessing on the image data?

2 Convolutional Neural Networks [60 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. For this question, we will focus to the **first 5 classes**: Samoyeds, Miniature Poodles, Golden Retrievers, Great Danes, and Dalmatians.

We have provided the training framework necessary to complete this problem in `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.

(a) [15 pts] Study the CNN architecture in Appendix B. How many learnable float-valued parameters does the model have? Show you calculations by completing the following table. Later, you can verify your answer against the program output after you implement the model.

Layer	Weights	Biases	Total
1	$5 \times 5 \times 3 \times 16$	16	$1,216$
2			
3			
4			
5			
6			
Grand Total			

(b) [10 pts] Speculate on these architecture choices by completing the table below (use your intuition). Fill in the current choice and one possible alternative. In the three rightmost columns, compare the alternative against the current choice by filling in either ↑ or ↓. For each comparison, provide a short justification that identifies the main differences.

	Current choice	**Alternative**	**How might the alternative affect...**		
			Training Speed	Approximation Error	Estimation Error
Initialization	random normal	zero	↓	↑	
Activation	ReLU	ELU			
Depth					
Regularization					

(c) [12 pts] In `pytorch`, we define a neural net by subclassing `torch.nn.Module`, which internally handles gradient computations. Complete the `CNN` class in `model.py`, by filling in the following three functions: `__init__()`, `init_weights()` and `forward(x)`. Attach a screenshot of your code.

- `__init__()` defines the architecture, i.e. what layers our network contains. At the end of this function we call `init_weights()` to initialize all layer weights/biases to desired distributions.
- `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. It performs back propagation and computes the partial derivatives with respect to each model parameter using chain rule.

(d) [3 pts] Fill in the definitions for `criterion` and `optimizer` in `train_cnn.py`, based on the model specification in Appendix B. Attach a screenshot of your code.

Now that you've implemented the architecture, let's train the neural net. **Review** the content of the trainer script `train_cnn.py`. In the main function, we have a training loop that loops through the dataset for a prespecified number of times as defined in the `config.json` file. After each complete pass of the training set (aka "epoch"), we evaluate the model on the validation set, and save model parameters as a checkpoint.

- `_train_epoch`: within one epoch, we pass batches of training examples through the network, use back propagation 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!

Run `train_cnn.py` to train the model. 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 two graphs that are updated every epoch to monitor the model's performance in terms of loss and accuracy on the train and validation set.

(e) Include the final learning curve plots [3 pts] in your write-up and answer the following questions.

i. [3 pts] You should observe that the training plot is fairly noisy and validation loss does not monotonically decrease. Describe at least two sources of noise that contribute to this behavior.

ii. [3 pts] If we were to continue training the model, what do you expect will happen to (1) training loss and (2) validation loss? Consider both the effects of additional training on these individual statistics as well as their relation to each other.

iii. [3 pts] Here, we stopped training after a fixed number of iterations. Based on your training plot, at which epoch should you stop training the model? Write down this value, and your reasoning for why you picked this value. Use this epoch number for the next problem.

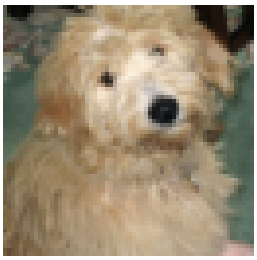
(f) **Run** `visualize_cnn.py` to generate visualizations for the activation maps of the first convolutional layer, when each of the following three images is passed through the network. These sample images are representative of their respective classes. Include the plots in your write-up [3 pts] .

In a convolutional layer, every filter has different weights and generates a distinct activation map given the same input. For example, the first convolutional layer has 16 filters, thus we have 16 different activation maps. These activation maps are single-channel images, and we can plot them using a continuum of colors, where darker color indicates small values and lighter color indicates large values. Filters in one layer are sorted by the mean of each filter's weights to improve consistency of results.

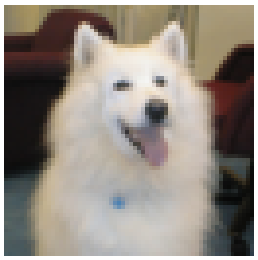
Answer the following questions:

i. [3 pts] **What** do you notice about the first-layer activations for input images of different classes? Compare and contrast the output from these three sample images based on their similarities and differences (in terms of color, shape, texture, etc.).

ii. [2 pts] **What** low-level features could each filter be looking for?



`000001.png`
$y = 1$
Miniature Poodle



`000122.png`
$y = 0$
Samoyed

`000265.png`
$y = 3$
Great Dane

3 Challenge [20 pts]

Equipped with your knowledge of neural nets and PyTorch, you're now ready to take on Ken'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 the first 5 classes; however, you will now be 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 exploration.

- **Implement** your model within `challenge_model.py`.
- **Fill in** the definitions for the loss function, optimizer, model, and the train˙epoch function. Feel free to alter the training framework in `challenge_train.py` as you see appropriate.
- If you wish to modify any portion of the data batching or preprocessing, please make a copy of `data.py` and call it `challenge_data.py` so it doesn't interfere with Q2.
- You may modify `config.json` as you feel necessary.

Once your model has been trained, ensure that its checkpoint exists under `checkpoints/challenge/` and **run** the script `challenge_predict.py`. This will load your model from your saved checkpoint and generate `predictions.csv` that contains your model's predictions on the test set.

Your grade for this question will be assessed by the following two components:

1. Effort [10 pts]: We will evaluate how much effort you have invested in this problem based on the your write-up and your code submission. You should describe the experiments you conducted and how they influenced your final model. We will look for at least some discussion of your design decisions regarding the the following.
   - Regularization (weight decay, dropout, etc.)
   - Feature selection
   - Model architecture
   - Hyperparameters
   - Model evaluation (i.e., the criteria you used to determine which model is best)
   - Utilization of the additional data
   - Whether your model was trained using GPU hardware, with additional data, or with pretrained models. (if you do not specify, we will assume you used GPU with additional data and pretrained models, so it's in your best interest to state this clearly)
2. Accuracy [10 pts]: We will evaluate the top-1 accuracy of your classifier's predictions on the ground truth test labels. In other words, given n test images, ground truth test labels $\{y^{(i)}\}_{i=1}^{n}$, and your predictions $\{\hat{y}^{(i)}\}_{i=1}^{n}$, your performance will be evaluated as follows.

$$\text{accuracy} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}[y^{(i)} == \hat{y}^{(i)}]$$

   Note that this means that the output test predictions follow a specified order. For that reason, please do not shuffle the test set data points during prediction.

Figure 2: Some of Ken's dogs back home safe and sound

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 consolutional 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 3, since the filter size is $k = 3$, the appropriate padding size is $p = \lfloor 3/2 \rfloor = 1$.

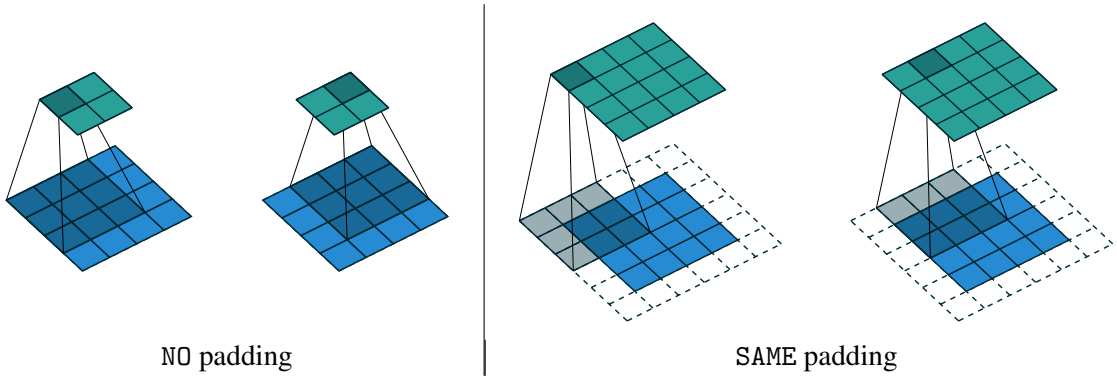


Figure 3: Comparison of padding schemes.

A.3 `torch.nn.CrossEntropyLoss`

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

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:

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

We then compare this probability distribution $\hat{y}$ with the ground truth distribution $\bar{y}$ (a one-hot vector), computing the cross entropy loss, $\mathcal{L}$:

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

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

B CNN Architecture

Training hyperparameters

- Criterion: `torch.nn.CrossEntropyLoss`
- Optimizer: `torch.optim.Adam`
- Learning rate: 10^{-4} (stored in `config("cnn.learning_rate")` which is `1e-4`)
- Number of epochs: 40
- Batch Size: 128

Architecture

Layer 0: Input image

- Output: $3 \times 32 \times 32$

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 $\mu = 0.0$, $\sigma^2 = \frac{1}{5 \times 5 \times 3}$
- Bias initialization: constant 0.0
- Output: $16 \times 16 \times 16$

Layer 2: Convolutional Layer 2

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

Layer 3: Convolutional Layer 3

- Number of filters: 32
- Filter size: 5×5
- Stride size: 2×2
- Padding: SAME
- Activation: ReLU
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{5 \times 5 \times 32}$
- Bias initialization: constant 0.0
- Output: $32 \times 4 \times 4$

Layer 4: Fully connected layer 1

- Input: 512
- Activation: ReLU
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{512}$
- Bias initialization: constant 0.0
- Output: 64

Layer 5: Fully connected layer 2

- Input: 64
- Activation: ReLU
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{64}$
- Bias initialization: constant 0.0
- Output: 32

Layer 6: Fully connected layer 3 (Output layer)

- Input: 32
- Activation: None
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma^2 = \frac{1}{32}$
- Bias initialization: constant 0.0
- Output: 5 (number of classes)