Assignment 2: CIFAR2: Convolutional Neural Networks

Through the eons, humans have interrogated the universe's many fundamental questions. How did we come to be? What comprises a "good" life? From whence can a sentient being derive meaning?

Of these many questions, one stands alone in its ability to evade consesus and vex the brightest thinkers.
Of course, it is....**Which are better: Cats or Dogs?**

We will not be able to answer such an intractable question in this short assignment. However, we can work towards our goal in earnest by tackling a simpler subproblem. **We will learn to tell them apart. (alternately you may ask the more bogan question… is it a bird or an airplane?)**

In this assignment, you will be building a Convolutional Neural Network (CNN) to classify an object in an image using the CIFAR dataset. ***Please read this handout in its entirety before beginning the assignment.***

Getting the stencil

You can find the files located on learn under the Assignments page. The files are compressed into a ZIP file, and to unzip the ZIP file, you can just double click on the ZIP file. There should be the files: assignment.py, preprocess.py, README.txt, and CIFAR_data folder. Place these in your google drive folder for assignment2 to get started.

Logistics

Work on this assignment off of the stencil code provided, but **do not change the stencil except where specified.** You shouldn't change any method signatures or add any trainable parameters to **init** that we don't give you (other instance variables are fine).

This assignment should take longer to run than the previous assignment. If completed correctly, the model should train and test within 10 minutes on Colab. While you will mainly be using TensorFlow functions, the second part of the assignment requires you to write your own convolution function, which is very computationally expensive.

CIFAR2, not CIFAR10

Your task is a binary classification problem. While the CIFAR10 dataset has 10 possible classes (airplane, automobile, bird, cat, deer, frog, horse, ship, and truck), you will build a CNN to take in an image and correctly predict its class to either be a cat or dog, hence CIFAR2. We limit this assignment to a binary classification problem so that you can train the model in a reasonable amount of time.

The assignment has **4 parts**.
Our stencil provides a model class with several methods and hyperparameters you need to use for your network. You will also fill out a function that performs the convolution operator. You will also answer a questions related to the assignment and class material as part of this assignment.

Part 1: The Model (a multi-layer network)

You will notice that the structure of the Model class is very similar to the Model class defined in your first assignment. **We strongly suggest that you complete the first assignment and review the Intro to TensorFlow Introduction Exercise (linked on Learn in Lecture Notes) before starting this assignment.** The exercise includes many explanations about the way a Model class is structured, what variables are, and how things work in TensorFlow including GradientTape.

Below is a brief outline of some things you should do. We expect you to fill in some of the missing gaps (review lecture slides to understand the pipeline) as this is your second assignment.

Step 1. Preprocess the data

- We have provided you with a function unpickle(file) in the preprocess file stencil, which unpickles an object and returns a dictionary. Do not edit it. We have already extracted the inputs and the labels from the pickled file into a dictionary for you, as you can see within get_data. The get_data function calls pre_process_data which you should edit.
- You will want to limit the inputs and labels returned by get_data to those representing the first and second classes of your choice. For every image and its corresponding label, if the label is not of the first or second class, then remove the image and label from your inputs and labels arrays.
- At this point, your inputs are still two dimensional. You will want to reshape your inputs into (num_examples, 3, 32, 32) using np.reshape(inputs, (-1, 3, 32 ,32)) and then transpose them so that the final inputs you return have shape (num_examples, 32, 32, 3).
- Recall that the label of your first class might be something like 5, representing a dog in the CIFAR dataset, but you will want to turn that to a 0 since this is binary classification problem. Likewise, for all images of the second class, say a cat, you will want to turn those labels to a 1.

- After doing that, you will want to turn your 0s and 1s to one hot vectors, where the index with a 1 represents the class of the correct image. You can do this with the function tf.one_hot(labels, depth=2).
- This is be a bit confusing so we'll just make it clear: **your labels should be of size (num_images, num_classes).** So for the first example, the corresponding label might be [0, 1] where a 1 in the second index means that it's a cat/dog/hamster/sushi.

Step 2. Train and test the provided single-layer ModelPart0

- In the main function, you will want to get your train and test data, initialize your model, and train it for many epochs. We suggest training for 25 epochs. We have provided for you a train and test method to fill out. The train method will take in the model and do the forward and backward pass for a SINGLE epoch. Yes, this means that, unlike the first assignment, your main function will have a for loop that goes through the number of epochs, calling train each time.
- Even though this is technically part of preprocessing, you should shuffle your inputs and labels when TRAINING so they split into different batches each epoch. Keep in mind that they have to be shuffled in the same order. We suggest creating a range of indices of length num_examples, then using tf.random.shuffle(indices). Finally you can use tf.gather(train_inputs, indices) to shuffle your inputs. You can do the same with your labels to ensure they are shuffled the same way.
- **You should also reshape the inputs into (batch_size, height, width, in_channels) before calling model.call().**
- Implement the cross-entropy loss function. Calculate the **average** softmax cross-entropy loss on the logits compared to the labels. We suggest using tf.nn.softmax_cross_entropy_with_logits to help.
- Call the model's forward pass and calculate the loss within the scope of tf.GradientTape. Then use the model's optimizer to apply the gradients to your model's trainable variables outside of the GradientTape. If you're unsure about this part, please refer to the TensorFlow Introduction Exercise. This is synonymous with doing the gradient_descent function in the first assignment, except that TensorFlow handles all of that for you!
- If you'd like, you can calculate the train accuracy and loss and print them each epoch.
- The test method will take in the same model, now with trained parameters, and return the accuracy given the test inputs and test labels. For ModelPart0 you should get around 58-60% accuracy on the test inputs.
- At the very end, we have written a method for you to visualize your results. The visualizer will not be graded but you can use it to check out your doggos and kittens.
- For fun, instead of passing in the indexes for dog and cats for your training and testing data, you can pass in other inputs and see how the model does when trying to classify something like bird vs. cat!

Step 3. Create your first multi-layer model

- Duplicate the ModelPart0 class and call your new class ModelPart1

- We recommend using an Adam Optimizer [tf.keras.optimizers.Adam] with a learning rate of 1e-3, but feel free to experiment with whatever produces the best results.
- Weight variables should be initialized from a normal distribution (tf.random.truncated_normal) with a standard deviation of 0.1.
- Add another linear layer by replicating the W1 and B1 to W2 and B2. We'd like to densely connect the two layers, so the size of the output for layer 1 should be 256 and the size of the input for layer 2 should be 256 (and its output size 2).
- Edit the code for the forward pass in call so that it calls the ReLU activation function after the first layer and then calls the linear_unit function for the second layer. Remember that the output from one layer should be the input to the next layer.
- Test the multi-layer model. It should improve the accuracy to around 60-63%

Part 2: Conv2d

In order to improve our classification performance, we want to introduce convolutions to our network. The convolutions help our network learn some invariance to image transformations such as translation as well as quickly and compactly learn filters that extract features from the images. Without convolutions we would need many more dense connections and would have real difficulty training such a large number of interconnections with gradient descent. Convolutions take advantage of known spatial locality in image features.

In this part you will implement a basic 2d convolution function. For the sake of simple math calculations (less is more, no?), we'll require that our conv2d function **only works with a stride of 1** (for both width and height). This is because the calculation for padding size changes as a result of the stride, which would be way more complex and unreasonable for a second assignment.

Do **NOT** change the parameters of the function we have provided. Even though the conv2d function takes in a strides argument, you should **ALWAYS** pass in [1, 1, 1, 1]. Leaving in strides as an argument was a conscious design choice - if you wanted to eventually make the function work for other kinds of strides in your own time, this would allow you to easily change it.

Roadmap

- Your inputs will have 4 dimensions. If we are to use this conv2d function for the first layer, the inputs would be [batch_size, in_height, in_width, input_channels].
- You should ensure that the input's number of "in channels" is equivalent to the filters' number of "in channels". Make sure to add an assert statement or throw an error if the number of input in channels are not the same as the filters in channels. You will lose points if you do not do this.

- When calculating how much padding to use for SAME padding, padding is just (filter_size - 1)/2 if you are using strides of 1. The calculation of padding differs if you increase your strides and is much more complex, so we won't be dealing with that. If you are interested, you may read about it here.
- You can use this hefty NumPy function np.pad to padd your input! Note that for SAME padding, the way you pad may result in different output shapes for inputs with odd dimensions depending on the way you pad. This is ok. **EDIT: To be fully transparent, we will only test that your convolution function matches TensorFlow's using inputs and filters whose output dimensions do not change if you floor your padding. This means that if your calculation for padding is not an integer, you should floor it, and if that results in output dimensions that are slightly smaller, that is ok.** Alternatively, if you have decided to handle padding fractional values separately by doing a sort of shifting, that's ok too.
- After padding (if needed), you will want to go through the entire batch of images and perform the convolution operator on each image. There are two ways of going about this - you can continuously append to multi dimensional NumPy arrays to an output array or you can create a NumPy array with the correct output dimensions, and just update each element in the output as you perform the convolution operator. We suggest doing the latter - it's conceptually easier to keep track of things this way.
- Your output dimension height is equal to (in_height + 2*padY - filter_height) / strideY + 1 and your output dimension width is equal to (in_width + 2*padX - filter_width) / strideX + 1. Again, strideX and strideY will always be 1 for this assignment. Refer to the slides if you'd like to understand this derivation.
- You will want to iterate the entire height and width including padding, stopping when you cannot fit a filter over the rest of the padding input. For convolution with many input channels, you will want to perform the convolution per input channel and sum those dot products together. Do not iterate the pixel values or sum/dot products.

- **NOTE: performance can be very poor as long as your solution is correct.**

Testing out your own conv2d:

- We have provided for you a few tests that compare the result of your very own conv2d and TensorFlow's conv2d. If you've implemented it correctly, the results should be very similar.

Part 3: A Convolutional Neural Network (CNN)

You will not receive credit if you use the tf.keras, tf.layers, and tf.slim libraries for this part. You can use tf.keras for your optimizer but do NOT use Keras layers!

Now that you have completed an implementation of a conv2d function, we are ready to build a Convolutional Neural Network. For performance reasons you can call the tf.nn.conv2d function instead of the one you wrote in Part 2 (this also lets you complete Part 3 if you don't have a fully working conv2d function in Part 2).

Step 1: Create a model with a Convolutional layer

- Again, copy and paste the previous model (ModelPart1) and create a ModelPart3.
- Add a single convolutional layer (and ReLU activation) at the beginning of the network (before the two densely connected linear layers).
    - You'll need to add a TensorFlow variable to your initialization to store the convolutional filter bank.
    - You also need to add a call to the conv2d function in the model.call function (and another call to ReLU after the conv2d)
    - See the Convolution in Tensorflow notebook on Learn (section Learning a Kernel) for an example.
- You will need to choose a size for your filter, a number of output channels (the number of input channels is fixed at 3 which are the RGB image color channels), and the stride.

Step 2: Train and test

- If you choose well you should be able to achieve about 70% accuracy and keep the training time for 25 epochs under 10 minutes.

Step 3: (optional) for the A to A+ achievement

- Change hyperparameters, add layers, and use 2D max pooling (see https://www.tensorflow.org/api_docs/python/tf/nn/max_pool2d) to your network architecture to achieve above 70% accuracy (75% is possible without additional techniques). You need to keep the 25 epoch, 10 minute time limit and may only use the max pooling, conv2d, ReLU, and linear units. You may not use additional data, augmented data (flipping, rotating, etc the training data), or hand crafted features/preprocessing.

Addendum: Data Set

The CIFAR files are pickled objects. We have provided you with a function unpickle(filename). You should not edit it. *Note:* You should normalize the pixel values so that they range from 0 to 1 (This can easily be done by dividing each pixel value by 255) to avoid any numerical overflow issues.

Data format

The testing and training data files to be read in are in the following format:

train: A pickled object of 50,000 train images and labels. This includes images and labels of all 10 classes. After unpickling the file, the dictionary will have the following elements:

- data -- a 50000x3072 numpy array of uint8s. Each row of the array stores a 32x32 colour image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image.
- labels -- a list of 50000 numbers in the range 0-9. The number at index i indicates the label of the ith image in the array data.
- Note that if you download the dataset from online, the training data is actually divided into batches. We have done the job of repickling all of the batches into one single train file for your ease.

test: A pickled object of 10,000 test images and labels. This includes images and labels of all 10 classes. Unpickling the file gives a dictionary with the same key values as above.

Visualizing Results

- We've provided the visualize_results(image_data, probabilities, image_labels, first_label, second_label) method for you to visualize your predictions against the true labels using matplotlib, a useful Python library for plotting graphs. This method is currently written with the image_labels having a shape of (num_images, num_classes). ***DO NOT EDIT THIS FUNCTION.*** You should call this function after training and testing, passing into into visualize_results an input of 10 images, 10 probabilities, 10 labels, the first label name, and second label name.
- Unlike the first assignment, you will need to pass in the strings of the first and second classes. A visualize_results method call might look like: visualize_results(image_inputs, probabilities, image_labels, "cat", "dog").
- This should result in a visual of 10 images with your predictions and the actual label written above so you can compare your results! You should do this after you are sure you have met the benchmark for test accuracy.

Addendum: Cats vs. Dogs? from the creators of this assignment

"Every dog has its day, but for all of the others I am a cat" - James Atlas
"I am currently neither, and I have been both in the past" - Daniel Ritchie
"Definitely a dog person. In fact I'm allergic to cats" - David Oyeka
"I'm a dog. But I am a cat person" - Zach Horvitz
"I'm a big time doggo" - Brian Oppenheim
"I feel like I'm neither dog or cat. I'm a bull" - Amy Pu (I'm actually a huge dog person)

- Assignment used with permission and significantly modified for use in COSC440 from CS1470 TA Staff | Computer Science Department | Brown University