P6: Using Learning to Create a Novel Game Controller

Deep learning has the potential to profoundly impact game design, development, and play. It offers a general method of learning complex functions from data, it is easy to use without a great deal of specialized knowledge (thanks to a growing repertoire of tools and libraries), and, in principle, it is vastly cheaper than the alternative of hand coding game features and solution algorithms for specific tasks. Deep Learning is commonly applied to classification tasks, i.e., to assign inputs to one of a finite set of categories. A surprising range of problems can be expressed in this form, including situation interpretation, action selection, and user preference modeling as might be encountered in games.

This assignment employs deep learning to create a novel game controller – you will write a classifier to recognize three facial expressions from camera input and use its output to play tic-tac-toe (by selecting the row and column to make your mark). You will demonstrate the surprising generality of the learned solution by applying it to a seemingly unrelated problem largely of your choice (e.g., to dog vs cat, or sushi vs sandwich classification).

We ask you to employ Keras for this assignment. Keras is deep learning middleware with an associated library, implemented on top of TensorFlow, which is in turn implemented in python.

The following sections describe the assignment in terms of the work flow you should follow. As a result, the writeup is lengthy but each step is small.

1. Load and Install Keras

You can use the following command to download all the requirements for this assignment. Use of a `virtualenv` is highly recommended, but not required.

```
pip install -r requirements.txt
```

2. Gather the Data

The facial expression dataset was made available by Kaggle as part of a computer vision competition in 2013. You will need to create a Kaggle account if you don't already have one. The pictures are medium resolution JPEGs of different sizes, taken in natural, mostly indoor settings. You can download the images from https://www.kaggle.com/datasets/msambare/fer2013. For this assignment, we only need the **happy**, **neutral** and **surprise** categories.

We have provided code to extract the first 5000 elements of the facial dataset from the target categories. We have found that a 5000-image dataset strikes a good compromise between training time and accuracy of the resulting classifier.

Run the supplied code (`export_dataset.py`) to export the first 5000 elements of the target categories. For this to work, you have to put the data from Kaggle in a directory named `kaggle`.

To test the results, use `show_example.py` to show some examples for each category.

3. Preprocess the Data

All deep learning tasks require a certain amount of data preprocessing. Here, you will need to:

1. Read in the image files
2. Preprocess the original jpeg data into RGB grids of pixels
3. Convert those into floating-point tensors (multi-dimensional arrays)
4. Rescale the values to the 0-1 range.

Keras has a function in `keras.utils` called `image_dataset_from_directory` which lets you create `tf.data.Dataset` objects from image directories. It uses the name of directories as the categories for the images.

The Dataset object is useful in deep learning applications for multiple reasons. Here, they avoid the need to keep all training, validation, and test data in memory by streaming the data batch by batch whenever it's needed. In addition, they can preprocess the data to some extent (change the format to RGB, resize) and can be used in a pipeline for data augmentation (growing the dataset by applying meaning-preserving transformations to its elements). As mentioned above, you can use `image_dataset_from_directory` to read the data in this format:

```
train_dataset, validation_dataset = image_dataset_from_directory(
        train_directory,
        label_mode='categorical',
        color_mode='rgb',
        batch_size=batch_size,
        image_size=image_size,
        validation_split=validation_split,
        subset='both',
        seed=47
    )
```

Here, `image_dataset_from_directory` is used to create 3 Dataset objects (e.g, `train_dataset`, `validation_dataset` and `test_dataset`). The image target size is set to 150 x 150, the batch size to 128, and specify `label_mode='categorical'`, meaning your labels identify categories in a multi-class problem. This is already implemented in `preprocess.py`.

4. Build an Initial Network

Compose a neural net for the facial expression classification problem. It should include convolutional and maxpooling layers that reduce the image size, a flatten layer, at least one or more fully connected layers, leading to a 'softmax' activation function that outputs a probability distribution over the 3 facial expression classes. Use 'relu' activation in the convolutional layers. The model should have **no more than 150,000 parameters** for this task. (You can use `model.print_summary()` to print the number of parameters in your model.)

Define this initial model in `models/basic_model.py`. Your model should be set to `self.model` in the `_define_model` function of `BasicModel` class.

5. Train your Network

Training a model in Keras only requires a couple of function calls. First, you need to configure the model for training. Implement the `_compile_model` function to configure the model with the proper optimizer and loss function.

```
def _compile_model(self):
    self.model.compile(
        optimizer=RMSprop(learning_rate=0.001),
        loss='categorical_crossentropy',
        metrics=['accuracy'],
    )
```

The `loss` is the error measure that learning strives to reduce. Here, categorical_crossentropy is the measure appropriate to a multi-class classification problem. RMSprop is one of several available optimizers. The parameter `learning_rate` is the amount to adjust weights in the neural network in the direction that decreases the loss function. The parameter `metrics` is the function used to track progress during learning, through comparison of predicted vs actual values from the validation set. Here, `accuracy` refers to classification accuracy.

After configuring the model for training, you will need to fit the model using the fit operator. This is already implemented in the `train_model` function. The structure of the call is:

```
  history = self.model.fit(
    x=train_dataset,
    epochs=epochs,
    verbose="auto",
    validation_data=validation_dataset,
```

```
    )
```

Here, an epoch represents learning from 1 complete pass through the training data. In each step, Keras learns (performs one gradient descent pass through the model) on every supervised data pair in a batch.

You will need to alter the number of epochs to avoid overfitting as you train the various models in this assignment. Overfitting is the condition where *additional* training *decreases* accuracy (or increases loss) of the learned mapping on **the validation set**.

Experiment with the structure of this network, e.g., by altering its shape, the number of layers, and their composition. **Stop when you obtain a model with 60% accuracy on the test set.** Make sure to select the model associated with the epoch just before overfitting sets in. You can use the plotting software to identify that point.

It is always good practice to save the model after learning, via a call like the following:

```
model.save('model.keras')
```

This call saves both the architecture of the trained model (the layers and their connectivity) and the learned weights (collectively, the performance system). A saved model can be reloaded with an appropriate `load()` command. Both of these have already been implemented in the save_model function and load_model functions.

Run `python train.py` to train the model you define in `basic_model.py`. This code uses the model definition from `basic_model.py` and the training/evaluation code from `model.py`. It will print out your train, test, and validation accuracy and save the model for use in future sections.

Your target in this section is to build a model that obtains **60% accuracy** or greater on **the test data**. This is the output highlighted as *"* Evaluating basic_model"*.

Submit a pdf report containing the following:

- **a printout showing the shape of your network, as generated by `model.print_summary()` marked "Initial Network"**

- **a plot showing training and validation loss as function of epoch**
- **a plot showing accuracy against the training and validation sets as a function of epoch**
- **the accuracy and loss of your best learned model (obtained as the model in effect when overfitting begins) when measured against the held-back *test* set**
- **In addition to the pdf report, submit your best learned model as a .keras file**

We provide the code for generating these plots. (see the file `train.py`. You can simply call the function `plot_history` on the history object. The history object is the output of calling `train_model` in `train.py`) .

Note that the training process takes less than 15 minutes for all the cases in this assignment, even without using the GPU. In case you are having problems with the training time:

- Test your model with a smaller number of epochs (e.g. 3) before fully training your model. You can change the number of epochs in `train.py`.
- Use a smaller model. A model with ~150000 parameters should be able to get the accuracy you need, but smaller models may also work, and might take less training time.
- Use Google Colab for training your model. Google Colab is a free service you can use with your google account (or logging into your ucsc email).
    - Zip the whole project. You might want to do this after running the export_dataset.py, so that you only need to zip 2000 images instead of all of them. You need to include: models, test, train, config.py, preprocess.py, and train.py.
    - Open https://colab.research.google.com/. Create a new notebook. Make sure to connect the notebook to GPU by opening “Runtime/Change Runtime Type” from the top bar. Change “hardware accelerator” to “GPU” and save.
    - Click on Files on the left side bar, and upload the zip file.
    - Use the following command in Colab to unzip your project:
      ```
      !unzip -q preprocess.zip
      ```
    - Run the project by using: (you don’t need to install tensorflow, keras, etc. Colab already has all of them installed.)
      ```
      !python train.py
      ```
    - Apply any changes by double-clicking on the file name.
    - Training on the GPU on Colab should take about a minute or two for each model. Google allows you to use the GPU for 8 hours every day. If you run out of time, you can change the hardware accelerator back to None, in which case training should take around 10 minutes for each model.
    - Download the model from the Files section when the training is complete and you’re happy with the accuracy.

6. Employ Hyper-parameter optimization

Hyper-parameter optimization is the process of tuning your model by searching across model-defining parameters, looking for the configuration that yields the best accuracy at the overfitting point. For this problem, you should vary the number of convolutional layers, fully connected layers, and dropout layers (see below), and the learning rate. You can vary the number of epochs as well but remember to select the model with best performance on the validation set, normally at the epoch where overfitting begins, as the basis for measuring performance on the test set. The early stopping feature from Keras can help you in choosing the best number of epochs.

A *dropout* layer randomly sets the inputs of its neurons to zero with some frequency at each gradient descent path through the network. Their use typically increases network performance at the point when overfitting sets in. My favorite explanation is that dropout layers force networks to represent knowledge more diffusely, which inhibits their tendency to memorize training data. Memorizing training data causes overfitting because it implies poorer performance on held back data that demands generalization.

As part of the hyper-parameter search, consider inserting 0 – 2 dropout layers into your network. Experiment with the location in your network to insert them, and with the dropout rate.

Your target in this section is to build a model that obtains 70% accuracy on the test set.

Pick your best performing configuration, and **submit a pdf report containing the following**:

- **a printout showing the shape of your network, as generated by `model.print_summary()`.**
- **a plot showing training and validation loss as function of epoch**
- **a plot showing accuracy against the training and validation sets as a function of epoch**
- **the accuracy and loss of your best learned model (obtained as the model in effect when overfitting begins) when measured against the held-back *test* set**
- **A report on your hyperparameter optimization strategy including what hyperparameter you experimented with, how you changed them, and how this affected your accuracy.**
- **In addition to the pdf report, submit your best learned model as a .keras file**

7. Play tic-tac-toe using your face as a game controller

Now that you have a solution for facial expression recognition, you can use it as an input modality to play a game. We have provided a shell for tic-tac-toe that invokes your facial expression recognizer twice – once to choose a row, and once to choose a column where to place your mark. We have also provided a rather dumb (random) bot to act as your opponent.

You have to implement the `get_emotion` function in `player.py`. The input to this function is an image, and the output should be a value of 0, 1, or 2 based on which emotion the image is classified as (neutral, happy or surprise) by your model. Use `load_model` from `tensorflow.keras.models` to read your saved trained model.

For this section, play a game of tic-tac-toe using your face as an input device. Run the game by using `python run.py`. When running this code, the program uses the webcam to capture an image as a way to identify first the row and then the column of the next move. At any point if the detected emotion is wrong, the user can override the selected row or column by entering the keyword "text" followed by the index of the intended row or column, as specified in the terminal. Try to win. **Submit a pdf report containing the following:**

- **A trace of moves in that game.**
- **Answers to these questions:**
  - **How well did your interface work?**
  - **Did it recognize your facial expressions with the same accuracy as it achieved against the test set?**
  - **If not, why not?**
- **The lines of code you added in the _get_emotion function to use the model for emotion detection**

8. Transfer Learning

Deep learning models have a surprising (and fascinating) level of generality – a model trained on one task (called the *source* task) is often applicable to another (called the *target* task). It is called *far transfer* when the source and target tasks are conceptually distinct. This section asks you to perform *far transfer* from your best facial recognition model to a two-class classification problem of your choice, using a dataset available on Kaggle.

The process is straightforward, and illustrated for far transfer into sushi-vs-sandwich classification in Figure 1:

- load your trained facial recognition model,
- eliminate the final softmax layer (which specializes it to a 3-class problem),
- freeze all parameters in the remainder so that they cannot change through further learning,
- bolt on one or more fully connected layers to perform an arbitrary computation over the features inferred by your facial recognition model, and
- train *only* the new FCNN and softmax layers on data from the new task

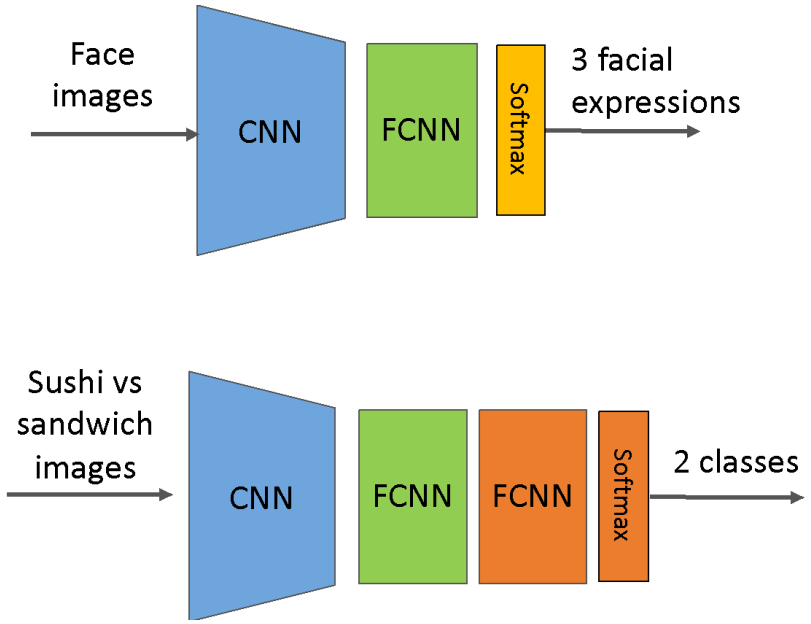

Figure 1. Architecture for far transfer from facial recognition to sushi-vs-sandwich classification.

We ask you to show a far transfer effect, not to obtain any fixed accuracy target on your chosen classification task. You show the effect by comparing two plots: validation loss on the target task with transfer, and validation loss without transfer (i.e., from training the modified architecture with all parameters randomly initialized, and available for learning). You can use the code provided in `train_transfer.py` as a starting point.

Show any of the following to receive credit:

- An increased intercept after epoch 1 with transfer
- An increased asymptote with transfer
- Achievement of the same accuracy level in fewer epochs with transfer
- Higher accuracy on the test set with transfer

You are free to employ any dataset you find on Kaggle. You should not need more than 2,000 images to show an effect, and you might need far less. You will likely need to change the input image shape to fit your existing architecture – the function *image_dataset_from_directory* is your friend here.

The following six datasets look promising:

- Dog v Cat
- Fresh vs rotten fruit
- Human portrait vs not human portrait
- Normal vs Leukemia blast cells
- Doom or animal crossing
- Ant v Bees

The last only has 120 images, but the goal is to show an effect not to obtain high accuracy.

The top 3 teams with the best example of far transfer will receive extra credit.

Submit a pdf containing the following:

- **A single plot showing validation accuracy as a function of epoch containing *with transfer* and *without transfer* curves, entitled "Far Transfer from Facial Recognition to <two class problem of your choice>"**
- **The final classification accuracy on the test set with transfer and without.**
- **An image of each facial recognition class**
- **An image of each class in the problem of your choice**
- **The summaries for both models (with and without transfer)**

Supplied code

├── kaggle
- Extract the kaggle data here

├── models
│ ├── __init__.py
│ ├── basic_model.py
│ ├── model.py
│ ├── random_model.py
│ └── transfered_model.py
- These are where you define your keras models. They are imported and run via `python train.py`

├── config.py
- Some configurations such as dataset size, image size, etc. You don't need to change anything here.

├── export_dataset.py
- For use in part 2 - extracts the 5000 images of target categories from the whole dataset. This creates two new directories called 'train' and 'test'.

├── game.py
- Logic of TicTacToe game for section 7. You don't need to change anything here.

├── gui.py
- Gui of TicTacToe game for section 7. You don't need to change anything here.

├── player.py
- Player logic and webcam access for section 7. You don't need to change anything here.

├── preprocess.py
- Scaffolding code for reading the images. You can use this as an example for importing the dataset for section 8.

├── requirements.txt
- You can install all project dependencies by running `pip install -r requirements.txt` (or pip3).

├── run.py
- Runs the TicTacToe game for section 7.

├── show_examples.py
- This file can be used to verify the dataset export step.

├── train.py

- This is where you will fit the model for section 5 and 6. As provided, it imports this model from `models/basic_model.py`.
- Included is a function to plot the model's training history with matplotlib.
- After training, it will save the model to a .keras file. These can be reloaded with a call to the load_model method.

└── train_transfer.py

- This is where you will fit the two models with and without transfer for section 8. As provided, it imports these models from `models/random_model.py` and `models/transfered_model.py`..
- Included is a function to plot the models' validation accuracies for comparison.