P6: Facial Gaming

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). We add the wrinkle that the data is in short supply. This is a common problem that can be addressed via several technical tricks, like data augmentation and transfer learning as utilized here.

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.

# 1. Load and Install Keras

You can download Keras from <https://keras.io/getting_started/>. You can use the following command to download all the requirements for this assignment. Use of a [virtualenv](https://virtualenv.pypa.io/en/latest/) 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 2000 elements of the facial dataset from the target categories. **It is important for this assignment that you use *only* the first 2000 elements of this dataset.**

Run the supplied code (export\_dataset.py) to export the first 2000 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 (you will utilize this capability later, in the data augmentation part of this assignment). 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 2 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.

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. 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.

Experiment with the structure of this network, e.g., by altering its shape, the number of layers, and their composition. Look for an architecture that delivers higher classification accuracy at the overfitting point (see the next section).

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.

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.

We ask you to **submit the following:**

* **a printout showing the shape of your network, as generated by model.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**

We provide the code for generating these plots (see the file train.py. You can simply call the function plot\_history on the history. The history is the output of calling train\_model in train.py) .

Your model is expected to reach the accuracy of 60% or greater on the **test data**. This is the output highlighted as *“\* Evaluating basic\_model”*.

The training process should take less than 10 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 100000 to 200000 parameters should be able to get the accuracy you need. (You can use model.print\_summary() to print the number of parameters in your model.)
* 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 Dropout Layers

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.

For this portion of the assignment, insert at least two dropout layers into your network from (5) above. Experiment with the location in your network to insert them, and with the dropout rate. Pick your best performing configuration, and **submit the following**:

* **a printout showing the shape of your network, as generated by model.summary() marked “With Dropout”**
* **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**

# 7. Employ Data Augmentation

Data augmentation is a means of compensating for sparse training data. The idea is to modify each piece of training data in multiple ways that preserve its label but present new information to the learning system. In the case of image data, it is common to rotate, stretch, zoom, and flip each image (among other manipulations).

Keras provides a simple method of configuring a data generator to augment each training image. Read the Data Augmentation tutorial from tensorflow (<https://www.tensorflow.org/tutorials/images/data_augmentation>) and configure it to perform data augmentations of your choice. Specifically, look at “Option 2: Apply the preprocessing layers to your dataset”, since we don’t want data augmentation to be part of our model here. (Remember not to augment the validation data - that test needs to be pure.) Overall, data augmentation is about remixing information present in the training data and is not a substitute for employing additional training data if it is available.

For this portion of the assignment, experiment with the type and quantity of data augmentation to improve the performance of your learning system from (6) above. Then, pick your best performing configuration, and **submit the following**:

* **a printout showing the shape of your network, as generated by model.summary()marked “With data augmentation”.**
* **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**
* **Your best learned model as an .keras file**

You will need to update the \_augment\_dataset and get\_datasets functions in preprocess.py to add data augmentation.

If all goes well, you should see a final classification accuracy above 60%.

# 8. Play tic-tac-toe with your face

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. Try to win. Submit 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?

# 9. Extra Credit: Use Feature Extraction (3 points)

One way to bolster classification accuracy is to provide the classifier with inputs that encode features of the data useful for classification. Here, you will apply a pretrained image classifier to your data, extract a vector of features from it, and use those features to improve facial expression recognition. This is an exercise in *knowledge transfer*; using a learned solution for a related task where training data was prevalent to improve learning (and performance) in a target task.

Keras provides a number of pre-trained image classifiers (e.g., Xception, Inception V3, ResNet50, VGG16, VGG19, Mobile Net). You will use VGG16 for this section, as it has a similar structure to the network you have already created. You can import VGG16 from the *keras.applications* module:

from tensorflow.keras.applications.vgg16 import VGG16

base\_model = VGG16(weights="imagenet", include\_top=False, input\_shape=input\_shape)

Here:

* weights=’imagenet’ tells the system to use the weights obtained from training VGG16 on the ~1.3M images in the imagenet training set)
* include\_top = False keeps discards VGG16’s final fully connected layers that generate 1000 classes (and keeps the rest)
* input\_shape is an optional parameter that specifies the shape of the image tensors you will feed to base\_model.

If you want to look at the structure of base\_model, use the command:

model.summary()

Implement and compare two approaches:

1. Merge the VGG16 features into your previous solution (which you can access by loading the saved model) by making them additional inputs of your dense layers. Train only those dense layers and freeze the CNN layers so that their learned parameters do not change. Implement this in models/merged\_model.py.
2. Define a new network that inputs the VGG16 features alone. Include additional layers as required to complete the facial expression recognition classification task. (This network makes no use of your previous solution.) Implement this in models/vgg\_model.py.

Use model.plot\_model\_shape() to make sure the layers are shaped in the way you want.

**Submit the following**:

* **a printout showing the shape of your network for (1), as generated by model.summary()marked “CNN with VGG16 Feature Extraction”**
* **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**
* **Your best learned model as an .keras file**
* **a printout showing the shape of your network for (2), as generated by model.summary()marked “With VGG16 Feature Extraction Alone”**
* **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**
* **Your best learned model as an .keras file**
* **Answer these questions**
  + **“Which of these two approaches did you expect to perform better?**
  + **Which performed better, and why?”**

# Supplied code

├── kaggle

* Extract the kaggle data here

├── models

│ ├── \_\_init\_\_.py

│ ├── basic\_model.py

│ ├── dropout\_model.py

│ ├── merged\_model.py

│ ├── model.py

│ └── vgg\_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 2000 images of target categories from the whole dataset. This creates two new directories called ‘train’ and ‘test’.

├── game.py

* Logic of TicTacToe game for part 8. You don’t need to change anything here.

├── gui.py

* Gui of TicTacToe game for part 8. You don’t need to change anything here.

├── player.py

* Player logic and webcam access for part 8. You don’t need to change anything here.

├── preprocess.py

* Scaffolding code for reading the images. You have to add some code here for part 7 to augment the training dataset.

├── requirements.txt

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

├── run.py

* Runs the TicTacToe game for part 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. As provided, it imports from models/basic\_model.py. You can modify the models variable to train another model. You have to include both a name (for the save file) and a reference to the model class.
* Included is a function to plot the model’s training history with matplotlib.
* After training, it will save the model to an .keras file. These can be reloaded with a call to the corresponding .load() method.