Assignment 2

Names:

- Alejandro Eisen Jofré - 202385817
- Noor Chaloner Yassein - 202375223
- Michele Romano - 202354540
- Tricia Govindasamy - 202375355
- Christy Bridgeman - 202353289

Overview

In this project, four main models (plus various configurations) were used in an attempt to find the best model for classifying emotions within images. The model that performed the best was the convolutional neural network (CNN), and was thus the final recommended model for this task.

Methodology

Exploring the data found a limited number of images, and particularly low numbers for certain emotions classes. Further, the data was given in one-channel pixels (i.e: grayscale), ranging from 0-255, and as such had to be transformed and scaled down to a 0-1 range to pass it into the neural network models. For this we simply divided the pixels by 255.

Given the low number of images, data augmentation was applied based off ImageDataGenerator from Keras, which was passed into the models. The script works by iterating over the original data and performing alterations of each of the images within the model as it ran, allowing the model to train on shifted, rotated, and zoomed versions of the original images. Data augmentation was passed only onto our training data, and as such the validation was performed with the “original” images.

There was a test and training set given, so the data did not need to be split for that purpose. However, the training and validation strategy was to split the data in the training set so that all the training data would be up to row 2900, and the data for validation would be all the rows after row 2900, which was about 10% of the training data. This was justified by a study on validation splits using a similarly limited dataset, which found 90/10 to be the split giving the highest accuracy (Muraina, 2022).

Learning rates were another decision that had to be made for each of the models. The main learning rates were the 1cycle policy, one which decreases exponentially throughout the model, and a plateau one. The 1cycle policy class allows the learning rate to start at a specific point, then increase up to its maximum, and decrease to a minimum lower than the original specific point. This increases accuracy, as a number of learning rates can be attempted before settling on a specific one (Smith, 2018). The exponentially decreasing learning rate is useful as it balances the benefits of both a large and small learning rate - essentially, the steps start out large, allowing efficient progress to be made, and gradually become smaller, which allows the model to converge (Géron, 2019, p. 183). The plateau one reduces the learning rate when in a plateau of loss for a defined number of epochs.

Basic Imports

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import os, datetime
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from google.colab import drive
from sklearn.utils import class_weight
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
from keras import backend as K
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Activation, Flatten, Input, Conv
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import LearningRateScheduler, ReduceLROnPlateau

keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)
```

In [ ]:
```
drive.mount('/content/drive')
```

In [ ]:
```
%cd /content/drive/MyDrive/CS985/cs985-987-emotion-recognition-project-20
```

Data Set-up

In [ ]:
```
df_train = pd.read_csv("my_emotion_train.csv")
df_test = pd.read_csv("my_emotion_test.csv")

data_train = df_train.copy() #copies of data for rollback just in case
data_test = df_test.copy()

data_train['pixels'] = data_train['pixels'].apply(lambda x: [int(pixel) f
data_test['pixels'] = data_test['pixels'].apply(lambda x: [int(pixel) for

#check for empty rows or the ones containing missing values
i = 0
while i in range(len(data_train)):
    if len(data_train.pixels[i])==0:
        print(f'empty here: {i}')
    elif np.nan in data_train.pixels[i]:
        print(f'empty here: {i}')
    i+=1
```

In [ ]:
```
def pixel_division(pixel_list):
  return[pixel / 255.0 for pixel in pixel_list] #scaling data into valid

data_train['pixels'] = data_train['pixels'].apply(pixel_division)
data_test['pixels'] = data_test['pixels'].apply(pixel_division)

X_train_full, y_train_full, id = data_train.pixels, data_train.emotion, d


X_valid, X_train = X_train_full[:2900], X_train_full[2900:] #separate dat
y_valid, y_train = y_train_full[:2900], y_train_full[2900:]

X_valid = np.array(X_valid.to_list()) #convert to array
X_train = np.array(X_train.to_list())
y_valid = np.array(y_valid.to_list())
y_train = np.array(y_train.to_list())

X_valid = X_valid.reshape(-1, 48, 48, 1) #reshape to pass into neural net
X_train = X_train.reshape(-1, 48, 48, 1)

X_test = data_test['pixels']
X_test = np.array(X_test.to_list())
X_test_id = data_test["id"]
X_test = X_test.reshape(-1, 48, 48, 1)  # Reshape to match model's expect
```

Class (im)balance

In [ ]:
```
emotion_names = {
    0: "Angry",
    1: "Disgust",
    2: "Fear",
    3: "Happy",
    4: "Sad",
    5: "Surprise",
    6: "Neutral"
}

emotion_counts = data_train["emotion"].value_counts()
emotion_counts = emotion_counts.rename(emotion_names)
emotion_counts.plot(kind="bar") #we can see the imbalanced distribution o

class_weights = class_weight.compute_class_weight(class_weight = 'balance
                                                  classes= np.unique(y_tr
                                                  y = y_train)
class_weight_dict = dict(enumerate(class_weights)) #we can use this later
```

Data Augmentation

In [ ]:
```
data_generation = ImageDataGenerator( #check hyperparameters to tune prop
    rotation_range=30,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1,
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False
)
```

Standard Machine Learning (ML) Model

For the standard ML model, a basic stochastic gradient descent (SGD) model was chosen. This decision was made as it is a basis for early neural network models, and as such, was a good place to start.

In [ ]:
```
X_train_full_array = np.array(X_train_full.to_list()) #since we dont need
X_train_full_array
sgd_X_train, sgd_X_test = train_test_split(X_train_full_array, test_size
sgd_y_train, sgd_y_test = train_test_split(y_train_full, test_size = 0.2)

#print(sgd_X_train.shape, sgd_X_test.shape, sgd_y_train.shape, sgd_y_test

sgd_clf = SGDClassifier()
sgd_clf.fit(sgd_X_train, sgd_y_train)

sgd_predictions = sgd_clf.predict(sgd_X_test)
sgd_conf_mat = confusion_matrix(sgd_y_test, sgd_predictions)
sgd_acc = accuracy_score(sgd_y_test, sgd_predictions)
print(sgd_conf_mat, sgd_acc)
```

In [ ]:
```
X_test.shape
```

In [ ]:
```
sgd_xtest = data_test['pixels']
sgd_xtest = np.array(sgd_xtest.to_list())
sgd_ytest_pred = sgd_clf.predict(sgd_xtest)

sgd_data = {'id': X_test_id, 'predicted class': sgd_ytest_pred}
sgd_data = pd.DataFrame(sgd_data)

# Save the DataFrame to a CSV file
sgd_data.to_csv("sgd_pred.csv", index=False)  # Set index=False to avoid
```

Deep Model Configurations

The first deep model configuration had five layers, decreasing neurons, and a regular decreasing learning rate. Given the significant time of convergence, the next configuration included a dropout layer for regularisation. This also increased the already-augmented data, as dropout layers drop a certain percentage of neurons in each iteration to vary the training set (Géron, 2019, p. 479). However, the accuracy was lower than the first model, thus l1 regularisation was implemented next instead of dropout. This iteration also performed worse, and so a batch normalisation layer was added in the next configuration, to normalise and standardise the training set, and help with accuracy. This performed better, but still not as high as the first, so l2 regularisation was attempted next - first with a dropout layer, as research suggested that this combination was useful (Srivastava et al, 2014). It did not prove useful, however, so the next configuration implemented batch normalisation instead. The accuracy scores improved significantly in this configuration. The combination of l1&l2 regularisation was next, but both together was not found to be an improvement, so l2 regularisation was settled on. For the final model, the learning rate was changed to the 1cycle class (created below), to see impacts of a different learning rate. The results did not improve, and thus the best configuration was the fifth, with l2 regularization & batch normalization.

In [ ]:
```
class DeepModel(keras.models.Model):
    def __init__(self, units=500, activation='relu', kernel_initializer='
                 hidden_layers=5, use_batch_norm = False, momentum = 0.99
                 dropout_rate = 0.0, kernel_regularizer = None, last_laye
        super().__init__(**kwargs)
        self.hidden_layers = []
        self.flatten = keras.layers.Flatten(input_shape=[48, 48])
        if last_layer_large:
          for i in range(hidden_layers-1):
            self.hidden_layers.append(keras.layers.Dense(units - i * 100,
                                                         activation = act
                                                         kernel_initializ
                                                         kernel_regulariz
          self.hidden_layers.append(keras.layers.Dense(units = 350,
                                       activation = activation,
                                       kernel_initializer=kernel_initializ
                                       kernel_regularizer = self.regulariz
        else:
          self.hidden_layers = [
              keras.layers.Dense(units - i * 100,
                                 activation=activation,
                                 kernel_initializer=kernel_initializer,
                                 kernel_regularizer=self.regularizer(kerne
              for i in range(hidden_layers)
            ]
        if use_batch_norm == True:
          self.batch_normalization_layers = [
```

```
            keras.layers.BatchNormalization(momentum = momentum) for _
          ]
        else:
          self.batch_normalization_layers = None

        if dropout_rate > 0.0:
          self.dropout_layers = [
              keras.layers.Dropout(rate = dropout_rate) for _ in range(hi
          ]
        else:
          self.dropout_layers = None
        self.output_layer = keras.layers.Dense(7, activation="softmax", k
        #we usually skip regularizing in output layer

    def regularizer(self, kernel_regularizer):
      if kernel_regularizer == None:
        return None
      elif kernel_regularizer == "l1":
        return keras.regularizers.L1()
      elif kernel_regularizer == "l2":
        return keras.regularizers.L2()
      elif kernel_regularizer == "l1l2":
        return keras.regularizers.L1L2()
      else:
        return ValueError("Not a valid regularizer. Use 'l1', 'l2' or 'l1

    def call(self, inputs, training = None):
        x = self.flatten(inputs)
        for i, layer in enumerate(self.hidden_layers):
          x = layer(x)
          if self.batch_normalization_layers is not None:
            x = self.batch_normalization_layers[i](x, training = training
          if self.dropout_layers is not None:
            x = self.dropout_layers[i](x, training = training)
        output = self.output_layer(x)
        return output

class OneCycleScheduler(keras.callbacks.Callback):
    def __init__(self, iterations, max_rate, start_rate=None,
                 last_iterations=None, last_rate=None):
        self.iterations = iterations
        self.max_rate = max_rate
        self.start_rate = start_rate or max_rate / 10
        self.last_iterations = last_iterations or iterations // 10 + 1
        self.half_iteration = (iterations - self.last_iterations) // 2
        self.last_rate = last_rate or self.start_rate / 1000
        self.iteration = 0
    def _interpolate(self, iter1, iter2, rate1, rate2):
        return ((rate2 - rate1) * (self.iteration - iter1)
                / (iter2 - iter1) + rate1)
    def on_batch_begin(self, batch, logs):
        if self.iteration < self.half_iteration:
            rate = self._interpolate(0, self.half_iteration, self.start_r
        elif self.iteration < 2 * self.half_iteration:
            rate = self._interpolate(self.half_iteration, 2 * self.half_i
                                     self.max_rate, self.start_rate)
        else:
```

```
            rate = self._interpolate(2 * self.half_iteration, self.iterat
                                     self.start_rate, self.last_rate)
        self.iteration += 1
        K.set_value(self.model.optimizer.lr, rate)


'''Checkpoints'''
annealer_exp = LearningRateScheduler(lambda x: 1e-4 * 0.95 ** x)

annealer_1cycle = OneCycleScheduler(iterations = 81600, #approximate of e
                                    max_rate = 0.01,
                                    start_rate = 0.001,
                                    last_iterations = 9063,
                                    last_rate = 1e-7)

annealer_plateau = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                                patience=5)

early_stopping_cb = keras.callbacks.EarlyStopping(patience=20, #patience
                                                  restore_best_weights=Tr
'''Optimizer'''
opt = keras.optimizers.Adam(amsgrad=True)
```

Configuration 1 - baseline deep model

In [ ]:
```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_1", save_best

dnn_1 = DeepModel(use_batch_norm= False)
dnn_1.compile(optimizer=opt, #perform backpropagation with adam
              loss = "sparse_categorical_crossentropy", # we use this los
              metrics = ["accuracy"])

history_1 = dnn_1.fit(data_generation.flow(X_train, y_train, batch_size=3
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 2: dropout

In [ ]:
```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_2", save_best

dnn_2 = DeepModel(activation="relu", use_batch_norm= False, dropout_rate=
dnn_2.compile(optimizer=opt,
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])

history_2 = dnn_2.fit(data_generation.flow(X_train, y_train, batch_size=3
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 3: l1 regularization

In [ ]:
```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_3", save_best

dnn_3 = DeepModel(activation="relu", use_batch_norm= False, kernel_regula
dnn_3.compile(optimizer=opt,
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])

history_3 = dnn_3.fit(data_generation.flow(X_train, y_train, batch_size=3
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 4: l1 regularization & batch norm

In [ ]:
```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_4", save_best

dnn_4 = DeepModel(activation="relu", use_batch_norm= True, momentum = 0.9
dnn_4.compile(optimizer=opt,
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])

history_4 = dnn_4.fit(data_generation.flow(X_train, y_train, batch_size=3
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 5: l2 regularization & batch norm

In [ ]:
```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_5", save_best

dnn_5 = DeepModel(activation="relu", use_batch_norm= True, momentum = 0.9
dnn_5.compile(optimizer=opt,
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])

history_5 = dnn_5.fit(data_generation.flow(X_train, y_train, batch_size=3
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 6: l1|2 regularization & batch norm

In [ ]:
```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_6", save_best

dnn_6 = DeepModel(activation="relu", use_batch_norm= True, momentum = 0.9
dnn_6.compile(optimizer=opt,
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])

history_6 = dnn_6.fit(data_generation.flow(X_train, y_train, batch_size=3
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 7: 1cycle & l2 regularization & batch norm

In [ ]:
```
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_7", save_best

dnn_7 = DeepModel(use_batch_norm= True, momentum = 0.9, kernel_regularize
dnn_7.compile(optimizer=opt,
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])

history_7 = dnn_7.fit(data_generation.flow(X_train, y_train, batch_size=3
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, annealer_1cycle
```

Configuration 1b - baseline deep model (different random seed)

In [ ]:
```
keras.backend.clear_session()
np.random.seed(43)
tf.random.set_seed(43)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_1b", save_bes

dnn_1b = DeepModel(use_batch_norm= False)
dnn_1b.compile(optimizer=opt, #perform backpropagation with adam
              loss = "sparse_categorical_crossentropy", # we use this los
              metrics = ["accuracy"])

history_1b = dnn_1b.fit(data_generation.flow(X_train, y_train, batch_size
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 2b: dropout (different random seed)

In [ ]:
```
keras.backend.clear_session()
np.random.seed(43)
tf.random.set_seed(43)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_2b", save_bes

dnn_2b = DeepModel(activation="relu", use_batch_norm= False, dropout_rate
dnn_2b.compile(optimizer=opt,
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])

history_2b = dnn_2b.fit(data_generation.flow(X_train, y_train, batch_size
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 3b: l1 regularization (different random seed)

In [ ]:
```
keras.backend.clear_session()
np.random.seed(43)
tf.random.set_seed(43)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_3b", save_bes

dnn_3b = DeepModel(activation="relu", use_batch_norm= False, kernel_regul
dnn_3b.compile(optimizer=opt,
               loss = "sparse_categorical_crossentropy",
               metrics = ["accuracy"])

history_3b = dnn_3b.fit(data_generation.flow(X_train, y_train, batch_size
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 4b: l1 regularization & batch norm (different random seed)

In [ ]:
```
keras.backend.clear_session()
np.random.seed(43)
tf.random.set_seed(43)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_4b", save_bes

dnn_4b = DeepModel(activation="relu", use_batch_norm= True, momentum = 0.
dnn_4b.compile(optimizer=opt,
               loss = "sparse_categorical_crossentropy",
               metrics = ["accuracy"])

history_4b = dnn_4b.fit(data_generation.flow(X_train, y_train, batch_size
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 5b: l2 regularization & batch norm (different random seed)

In [ ]:
```
keras.backend.clear_session()
np.random.seed(43)
tf.random.set_seed(43)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_5b", save_bes

dnn_5b = DeepModel(activation="relu", use_batch_norm= True, momentum = 0.
dnn_5b.compile(optimizer=opt,
               loss = "sparse_categorical_crossentropy",
               metrics = ["accuracy"])

history_5b = dnn_5b.fit(data_generation.flow(X_train, y_train, batch_size
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

In [ ]:
```
# save model

from keras.models import load_model

X_test = data_test['pixels']
X_test = np.array(X_test.to_list())
X_test_id = data_test["id"]
X_test = X_test.reshape(-1, 48, 48, 1)  # Reshape to match model's expect

dnn_best_model = load_model("dnn_config_5b")
dnn_predictions = dnn_best_model.predict(X_test)
# Get predicted class indices for each image
dnn_predicted_classes = np.argmax(dnn_predictions, axis=1)

dnn_data = {'id': X_test_id, 'predicted class': dnn_predicted_classes}
dnn_gru = pd.DataFrame(dnn_data)

# Save the DataFrame to a CSV file
dnn_gru.to_csv("dnn_pred.csv", index=False)  # Set index=False to avoid s
```

Configuration 6b: l1|2 regularization & batch norm (different random seed)

In [ ]:
```
keras.backend.clear_session()
np.random.seed(43)
tf.random.set_seed(43)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_6b", save_bes

dnn_6b = DeepModel(activation="relu", use_batch_norm= True, momentum = 0.
dnn_6b.compile(optimizer=opt,
               loss = "sparse_categorical_crossentropy",
               metrics = ["accuracy"])

history_6b = dnn_6b.fit(data_generation.flow(X_train, y_train, batch_size
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, early_stopping_
```

Configuration 7b: 1cycle & l2 regularization & batch norm (different random seed)

In [ ]:
```
keras.backend.clear_session()
np.random.seed(43)
tf.random.set_seed(43)

opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("dnn_config_7b", save_bes

dnn_7b = DeepModel(use_batch_norm= True, momentum = 0.9, kernel_regulariz
dnn_7b.compile(optimizer=opt,
               loss = "sparse_categorical_crossentropy",
               metrics = ["accuracy"])

history_7b = dnn_7b.fit(data_generation.flow(X_train, y_train, batch_size
                                epochs = 100,
                                steps_per_epoch = X_train.shape[0]//32,
                                validation_data=(X_valid, y_valid),
                                callbacks = [checkpoint_cb, annealer_1cycle
```

GRU model

For the second complex model, a Gated Recurrent Unit (GRU) model was chosen. A paper on the subject suggested the use of a GRU model combined with an attention mechanism to specify which sections of an image the model should be focusing on (Li et al, 2021). The idea was that, since classic CNNs do not assign discriminative weights to the informative local areas, this lack of focus may lead to misclassifications (Li et al, 2021). As such, a GRU was created, along with a custom multi-attention function which was added to the model as an individual layer.

In [ ]:
```
#create a class for Attention layer

class Attention_function(tf.keras.layers.Layer):
    def __init__(self, units):
        super(Attention_function, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)

def call(self, features, hidden):
  hidden_with_time_axis = tf.expand_dims(hidden, 1)
  score = tf.nn.tanh(self.W1(features) + self.W2(hidden_with_time_axis))
  attention_weights = tf.nn.softmax(self.V(score), axis=1)
  context_vector = attention_weights * features
  context_vector = tf.reduce_sum(context_vector, axis=1)
  return context_vector, attention_weights
```

In [ ]:
```
#create the model

gru_model = keras.models.Sequential()
gru_model.add(keras.layers.Reshape((48, 48), input_shape=(48,48,1)))
gru_model.add(keras.layers.GRU(32, activation='selu', kernel_initializer=
gru_model.add(keras.layers.GRU(128, activation='selu', kernel_initializer
gru_model.add(keras.layers.BatchNormalization())
gru_model.add(Attention_function(160))
gru_model.add(keras.layers.Dense(128))
gru_model.add(keras.layers.BatchNormalization())
gru_model.add(keras.layers.Dense(128))
gru_model.add(keras.layers.Dense(160))
gru_model.add(keras.layers.BatchNormalization())
gru_model.add(keras.layers.Dense(7, activation = "softmax", kernel_initia
```

In [ ]:
```
import math

initial_lr = 1e-4
def step_decay(epoch):
    initial_lr = 1e-4
    drop = 0.5
    epochs_drop = 10
    lr = initial_lr * math.pow(drop, math.floor((1+epoch)/epochs_drop))
    return lr

annealer = LearningRateScheduler(step_decay)
```

In [ ]:
```
opt = keras.optimizers.Adam(amsgrad=True)

checkpoint_cb = keras.callbacks.ModelCheckpoint("gru_model", save_best_on
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=Tr


#compile the model
gru_model.compile(optimizer=opt,
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])


#fit the model
gru_model = gru_model.fit(data_generation.flow(X_train, y_train, batch_si
                            epochs = 100,
                            steps_per_epoch = X_train.shape[0]//32,
                            validation_data = (X_valid, y_valid),
                            callbacks = [checkpoint_cb, early_stopping_cb
```

In [ ]:
```
pd.DataFrame(gru_model.history).plot()
```

In [ ]:
```
gru_best_model = load_model("gru_model")
gru_predictions = gru_best_model.predict(X_test)
# Get predicted class indices for each image
gru_predicted_classes = np.argmax(gru_predictions, axis=1)

gru_data = {'id': X_test_id, 'predicted class': gru_predicted_classes}
df_gru = pd.DataFrame(gru_data)

# Save the DataFrame to a CSV file
df_gru.to_csv("gru_pred.csv", index=False)  # Set index=False to avoid sa
```

Convolutional Neural Network

For the second (and final) complex model we chose a Convolutional Neural Network (CNN), this because CNNs are particularly good at image recognition tasks (Chauhan, Ghanshala, & Joshi, 2018). We tried a series of different configurations for our CNN architecture but all followed the same basic pattern:

1. Input
2. Convolution
    - Conv2D (x1 or x2):
        - Filters = 64, 128 or 256. Increasing depending on which convolution.
        - Size = (7,7), (5,5) or (3,3). Decreasing in size depending on convolution
        - Padding = Same
        - Activation = Softplus
        - Initializer = he_normal
3. Pooling
    - Strides = (2,2)
    - Pool Size = (2,2)
4. Batch Normalization
5. Dropout
6. Fully Connected
7. Output

We tried adding extra convolutions, changing the normalization momentum, the dropout rate, as well as batch sizes, and even the ImageDataGenerator parameters, some of which were trial and error decisions and others were due to previously known well performing configurations of CNNs. For example, both the AlexNet and LeNet-5 architectures show the use of fully connected layers right before the output. AlexNet model shows the use of more than one convolutional layer on top of one another without pooling in the middle (Géron, 2019). We used Softplus which is part of the ReLU family and seemed to perform better on our model.

In [ ]:
```
keras.backend.clear_session()
np.random.seed(43)
tf.random.set_seed(43)

cnn_model = Sequential([
    Input(shape=(48,48,1)),
    Conv2D(64, kernel_size=(7,7), activation='softplus', kernel_initializ
    Conv2D(64, kernel_size=(3,3), activation='softplus', kernel_initializ
    MaxPooling2D(pool_size=(2,2), strides=(2,2)),
    BatchNormalization(),
    Dropout(0.3),
    Conv2D(128, kernel_size=(3,3), activation='softplus', kernel_initiali
    Conv2D(128, kernel_size=(3,3), activation='softplus', kernel_initiali
    AveragePooling2D(pool_size=(2,2), strides=(2,2)),
    BatchNormalization(),
    Dropout(0.3),
    Conv2D(256, kernel_size=(3,3), activation='softplus',kernel_initializ
    Conv2D(256, kernel_size=(3,3), activation='softplus',kernel_initializ
    MaxPooling2D(pool_size=(2,2), strides=(2,2)),
    BatchNormalization(),
    Dropout(0.3),
    Conv2D(256, kernel_size=(3,3), activation='softplus',kernel_initializ
    Conv2D(256, kernel_size=(3,3), activation='softplus',kernel_initializ
    Flatten(),
    Dense(500, activation='relu',kernel_initializer='he_normal'),
    Dense(400, activation='relu',kernel_initializer='he_normal'),
    Dense(300, activation='relu',kernel_initializer='he_normal'),
    Dense(7, activation = "softmax", kernel_initializer = "glorot_normal"
])

#keras.utils.plot_model(cnn_model)
opt = keras.optimizers.Adam(amsgrad=True)
checkpoint_cb = keras.callbacks.ModelCheckpoint("cnn_model.keras", save_b

total_samples = X_train.shape[0]
batch_size = 64
steps_per_epoch = total_samples // batch_size

if total_samples % batch_size != 0:
    steps_per_epoch += 1

cnn_model.compile(optimizer=opt,
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])

cnn_model = cnn_model.fit(data_generation.flow(X_train, y_train, batch_si
                                    epochs=200,
                                    steps_per_epoch=steps_per_epoch,
                                    class_weight=class_weight_dict,
                                    validation_data=(X_valid, y_valid),
                                    callbacks=[checkpoint_cb, annealer_pla
```

In [ ]:
```
best_cnn = load_model("cnn_model.keras")  # Load the best saved model
cnn_predictions = best_cnn.predict(X_test)
cnn_predicted_classes = np.argmax(cnn_predictions, axis=1)

cnn_data = {'id': X_test_id, 'predicted class': cnn_predicted_classes}
df_cnn = pd.DataFrame(cnn_data)


df_cnn.to_csv("cnn_pred.csv", index=False)  # index=False due to own inde
```

Results

The results table below outlines the training and testing scores for each model attempted. The first model was the base-level SGD model. Its performance wasn't extremely high, with a mean of 0.2079 accuracy on the training data, however this served as a basis for the models that followed. The next model attempted was the deep neural network model, along with several configurations. Its performance was improved, as can be seen by the accuracy score in the table. Then a different configuration of the model was attempted – a single layer, which in theory is the optimal number for hidden layers (Heaton, 2008). However, the performance did not improve as expected. One potential reason for this could be that the shape of the dataset was not as expected – one hidden layer is best for continuous mapping, but two hidden layers are useful for functions with any shape (Heaton, 2008). The multiattention gated recurrent unit (GRU) model was also attempted. As explained above, using the multiattention function in combination with the GRU was meant to increase focus on the important aspects of the images, so as to increase accuracy and minimise the likelihood of misclassification. However, despite various iterations of the model (including changing the optimiser and number of layers, as well as varying the neurons in each layer), it did not perform as well as expected. One possible reason for this could be that the images were not high quality enough for the attention mechanism to be useful, as it was suggested in the paper that the highest impacts will likely be seen on high-quality images (Li et al, 2021). Finally, a convolutional neural network (CNN) was tried, with much improved performance. The Kaggle score of 0.59 (and mean accuracy score of 0.58) was the highest out of all the models attempted, and therefore was the chosen final recommendation for this task.

Generally, increasing the number of layers tended to improve performance. Increasing the number of epochs also helped, as it allowed the model more time to learn. Data augmentation improved the performance for similar reasons. Additionally, in most models, batch normalization was useful – and generally should not be used next to dropout, so that was something to ensure throughout the process (Kim, 2021). However, we did find that adding both in our CNN gave us the best performance.

Model name	Iteration	Accuracy Score - Training	Accuracy Score - Validation	Mean	Standard Deviation
Standard Baseline	01	0.1934	-	-	
	02	0.2181	-	-	
-	03	0.2112	-	-	
-	-	-	-	0.2079	0.009
	-				
Deep Model	01	0.3949	0.4179	-	
	02	0.3991	0.4186	-	
	-	-	-	0.4076	0.0107
	-				
GRU model	01	0.3401	0.3579	-	
	02	0.3201	0.3314	-	
	-	-	-	0.3374	0.0138
	-				
CNN model	01	0.5943	0.5807	-	
	02	0.5480	0.5817	-	
	-	-	-	0.5812	0.0005

Model name	Kaggle Score
Standard Baseline	0.21
Deep Model	0.42
GRU Model	0.36
CNN Model	0.59

Summary

Overall, the model recommended for this task is the CNN model. This is because it had the best performance, without overfitting - and this allows the results to be generalisable. When entered into Kaggle, the CNN model performed well, with an accuracy of 0.59. This is higher than the other three models: the SGD model had a score of 0.21, the deep model had a score of 0.42, and the multi-attention GRU model had a score of 0.36.

One recommendation for the future would be to ensure investment in consistent GPU. Google colab was used for this project, which has a limited amount of GPU for each user. As such, once that allotted amount ran out, all models took significantly longer to run. Another method to improve the model would be to add not only more images to the dataset, but also increase the number of emotions used as categories, to further improve both accuracy and range (and therefore generalisability of the model).

References

Smith, L.N. (2018) A disciplined approach to neural network hyperparameters: Part 1 - - learning rate, batch size, momentum, and weight decay. US Naval Research Laboratory. Available at: https://arxiv.org/abs/1803.09820v2.

Li, B., Guo, Y., Yang, J., Wang, L., Wang, Y. and An, W. (2021), 'Gated Recurrent Multiattention Network for Remote Sensing Image Classification', IEEE Transactions on Geoscience and Remote Sensing, pp.1-13. https://www.researchgate.net/figure/Illustration-of-the-proposed-gated-recurrent-multiattention-network-a-Multiscale_fig2_353467605.

Heaton, J. (2008) Introduction to Neural Networks for Java: 2nd edition. Available at: https://dl.acm.org/doi/10.5555/1502373. (Accessed: 06 April 2024).

Kim, S. (2021) 'Demystifying Batch Normalization vs Drop out', Medium, 11 October. Available at: https://skirene.medium.com/demystifying-batch-normalization-vs-drop-out-1c8310d9b516#:~:text=Dropout%2C%20on%20the%20other%20hand,creates%20disha

Géron, A. (2019) Hands-On Machine Learning with Scikit-learn, Keras, and TensorFlow. 2nd edn. California: O'Reilly.

Muraina, I. (2022) 'Ideal dataset splitting ratios in machine learning algorithms: general concerns for data scientists and data analysts'. Available at: https://www.researchgate.net/profile/Ismail-Muraina/publication/358284895_IDEAL_DATASET_SPLITTING_RATIOS_IN_MACHINE_LE DATASET-SPLITTING-RATIOS-IN-MACHINE-LEARNING-ALGORITHMS-GENERAL-CONCERNS-FOR-DATA-SCIENTISTS-AND-DATA-ANALYSTS.pdf.

Chauhan, R., Ghanshala, K., Joshi, R.C. (2018) 'Convolutional Neural Network (CNN) for Image Detection and Recognition'. 2018 First International Conference on Secure Cyber COmputing and Communication. https://doi.org/10.1109/ICSCCC.2018.8703316.