Facial Feature Detection

Introduction

In this project we aim to identify keypoints in facial photos. The goal is to look at facial images and locate 15 key facial features such as the eyes, nose, eyebrows, etc. This is a crucial step in facial recognition and authentication applications, emotion recognition, and other applications that require facial features to operate. To acieve this goal we aim to use a convolutional neural network (CNN) that can generalize and accurately predict each of the 15 key facial features on unseen facial images. The dataset for this project can be found on kaggles website using this link https://www.kaggle.com/c/facial-keypoints-detection.

Goal

The goal is to look at facial images and accurately locate 15 key facial features such as the eyes, nose, eyebrows, etc.

Data Overview

There are two different data sets: train: 7,049 face images. Includes labels of the 15 key facial features on each images. Each label as the x and y corrdinates for each image test: 1783 face images without any labels

Key Features:

- left_eye_center
- right_eye_center
- left_eye_inner_corner
- left_eye_outer_corner
- right_eye_inner_corner
- right_eye_outer_corner
- left_eyebrow_inner_end
- left_eyebrow_outer_end
- right_eyebrow_inner_end
- right_eyebrow_outer_end
- nose_tip
- mouth_left_corner
- mouth_right_corner
- mouth_center_top_lip
- mouth_center_bottom_lip

Image info:

- Each image is 96x96 pixels
- greyscale

Exploratory Data Analysis (EDA)

Okay before we start creating our model we need to explore the data set a bit to understand what we are working with. In our case we should be working with labeled grayscale facial images with 15 different labels on the face with coordinate info for these features. We should look at the raw images, see if we are missing an values, and just overall look at what we are working with.

First lets import the packages we are working with

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow.keras import layers, models
from tensorflow.keras.losses import MeanSquaredError
from sklearn.model_selection import train_test_split
import tensorflow as tf
```

Okay so lets load our data and take a quick look at how many samples we are working with

In [2]:
```
train_df = pd.read_csv("training.csv")
test_df = pd.read_csv("test.csv")
print("Training samples: ", train_df.shape[0])
print("Testing samples: ", test_df.shape[0])
```

```
Training samples:  7049
Testing samples:  1783
```

Visualization

Lets see what the photos look like

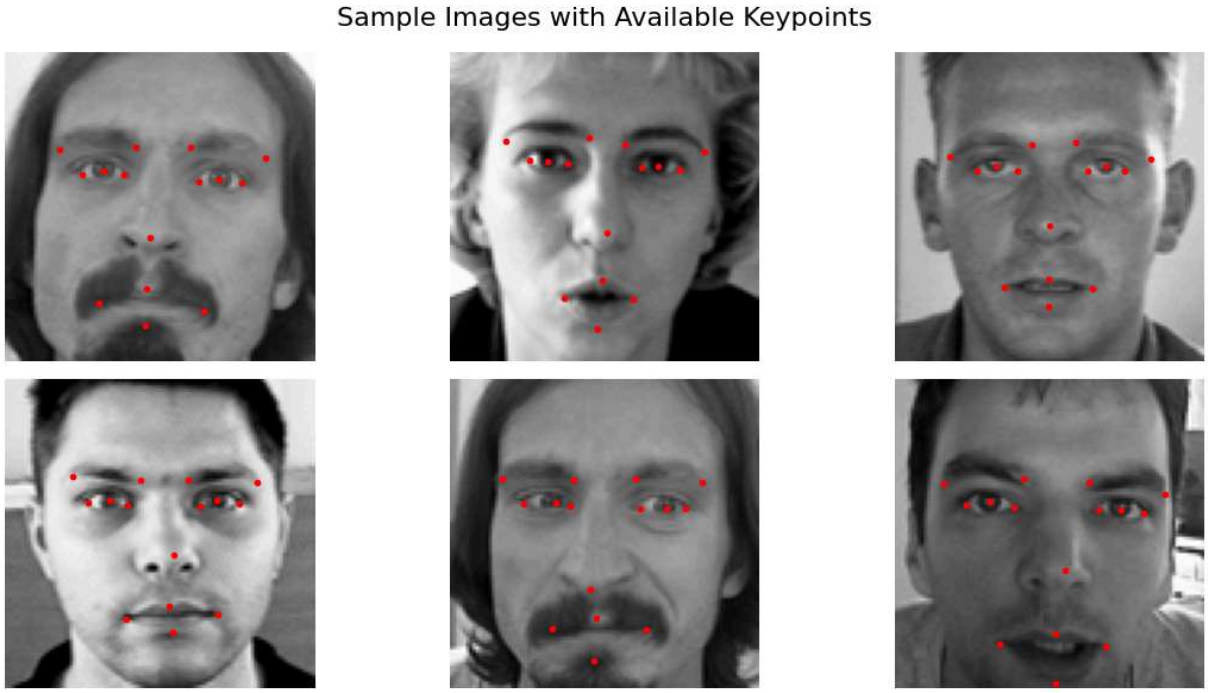
In [3]:
```
def plot_sample(index):
    row = train_df.iloc[index]
    img = np.fromstring(row['Image'], sep=' ').reshape(96, 96)
    plt.imshow(img, cmap='gray')

    for i in range(0, len(row)-1, 2):
        x = row[i]
        y = row[i+1]
        if pd.notna(x) and pd.notna(y):
            plt.plot(x, y, 'ro', markersize=3)
```

```
    plt.axis('off')


plt.figure(figsize=(12, 6))
for i in range(6):
    ax = plt.subplot(2, 3, i+1)
    plot_sample(i)
plt.suptitle("Sample Images with Available Keypoints", fontsize=16)
plt.tight_layout()
plt.show()
```

```
C:\Users\Grant\AppData\Local\Temp\ipykernel_11996\2899854254.py:8: FutureWarning: Se
ries.__getitem__ treating keys as positions is deprecated. In a future version, inte
ger keys will always be treated as labels (consistent with DataFrame behavior). To a
ccess a value by position, use `ser.iloc[pos]`
  x = row[i]
C:\Users\Grant\AppData\Local\Temp\ipykernel_11996\2899854254.py:9: FutureWarning: Se
ries.__getitem__ treating keys as positions is deprecated. In a future version, inte
ger keys will always be treated as labels (consistent with DataFrame behavior). To a
ccess a value by position, use `ser.iloc[pos]`
  y = row[i+1]
```



As you can see the faces have red dots indicating the spots of different features that the model can train on.

Data Understasnding

Now lets see how many of each features we have

In [4]:
```
print("Number of key features:")
keyfeature_counts = train_df.notnull().sum().drop('Image')
print(keyfeature_counts.sort_values(ascending=False))
```

```
Number of key features:
nose_tip_y                    7049
nose_tip_x                    7049
left_eye_center_x             7039
left_eye_center_y             7039
right_eye_center_x            7036
right_eye_center_y            7036
mouth_center_bottom_lip_x     7016
mouth_center_bottom_lip_y     7016
mouth_center_top_lip_x        2275
mouth_center_top_lip_y        2275
left_eye_inner_corner_x       2271
left_eye_inner_corner_y       2271
left_eyebrow_inner_end_x      2270
left_eyebrow_inner_end_y      2270
mouth_right_corner_x          2270
mouth_right_corner_y          2270
right_eyebrow_inner_end_y     2270
right_eyebrow_inner_end_x     2270
mouth_left_corner_x           2269
mouth_left_corner_y           2269
right_eye_inner_corner_x      2268
right_eye_inner_corner_y      2268
right_eye_outer_corner_x      2268
right_eye_outer_corner_y      2268
left_eye_outer_corner_y       2267
left_eye_outer_corner_x       2267
right_eyebrow_outer_end_x     2236
right_eyebrow_outer_end_y     2236
left_eyebrow_outer_end_y      2225
left_eyebrow_outer_end_x      2225
dtype: int64
```

Data Cleanup

Lets see how many feature values are missing in each feature

In [5]:
```
print("How many values are missing in each facial feature:")
print(train_df.isnull().sum())
```

```
How many values are missing in each facial feature:
left_eye_center_x              10
left_eye_center_y              10
right_eye_center_x             13
right_eye_center_y             13
left_eye_inner_corner_x      4778
left_eye_inner_corner_y      4778
left_eye_outer_corner_x      4782
left_eye_outer_corner_y      4782
right_eye_inner_corner_x     4781
right_eye_inner_corner_y     4781
right_eye_outer_corner_x     4781
right_eye_outer_corner_y     4781
left_eyebrow_inner_end_x     4779
left_eyebrow_inner_end_y     4779
left_eyebrow_outer_end_x     4824
left_eyebrow_outer_end_y     4824
right_eyebrow_inner_end_x    4779
right_eyebrow_inner_end_y    4779
right_eyebrow_outer_end_x    4813
right_eyebrow_outer_end_y    4813
nose_tip_x                      0
nose_tip_y                      0
mouth_left_corner_x          4780
mouth_left_corner_y          4780
mouth_right_corner_x         4779
mouth_right_corner_y         4779
mouth_center_top_lip_x       4774
mouth_center_top_lip_y       4774
mouth_center_bottom_lip_x      33
mouth_center_bottom_lip_y      33
Image                           0
dtype: int64
```

Okay so we have a lot of missing data and we need to figure out a way to deal with them. In most cases we would just drop the few missing rows of data, but in this case we would be dropping more than half of the training data which isnt ideal in the slightest. As you can see above we have some values that have way less missing values than others. These are left_eye_center_x, left_eye_center_y, right_eye_center_x, right_eye_center_y, nose_tip_x, nose_tip_y, mouth_center_bottom_lip_x, mouth_center_bottom_lip_y.

So we need to be able to work with what we have for each image because if we start dropping features we will lose many samples of data. To do this our model will run its model on features that are currently present in the photo. So features that appear in the most in the samples will have more training than features that have many missing values. To do this we are going to use masking.

Below in the block of code we will keep only the key feature information in each image and then we put these values into an array. Then we build a mask which essentially tells the model to only pay attention to the values that are present and dont pay attention to the values that are missing in each photo. This is exactly what we want to do when we have

many values that are missing in the data. Missing values will be replaced with the value 0, but it still lets the model train on the values that are present in each image.

In [6]:
```
y_raw = train_df.drop(columns=['Image']).values
y = (y_raw - 48) / 48
mask = ~np.isnan(y)
y[np.isnan(y)] = 0.0

def masked_mse_loss(y_true, y_pred):
    mask = tf.cast(tf.not_equal(y_true, 0.0), tf.float32)
    loss = tf.square(y_true - y_pred) * mask
    return tf.reduce_sum(loss) / tf.reduce_sum(mask)
```

Preprocessing

Now we need to get the data ready for our model. The code below is essentially turning the csv image file information and turning it into image tensors that our CNN model can process, learn, and then make predictions with.

In [7]:
```
image_list = []

for img_str in train_df['Image']:
    if isinstance(img_str, str):
        pixels = np.fromstring(img_str, sep=' ')
        img_array = pixels.reshape(96, 96)
        image_list.append(img_array)

X = np.array(image_list, dtype=np.float32) / 255.0
X = np.expand_dims(X, axis=-1)
```

Lets also break up the training set into training and testing

In [8]:
```
X_train, X_val, y_train, y_val, mask_train, mask_val = train_test_split(X, y, mask,
```

Model Architecture

We will be using a Convolutional Neural Network (CNN) for this project to predict facial features. We are using this method because CNNs are great for image data because they detect patterns between images such as nose, chins, and ears in our case. The modle will consists of multiple convolutional and pooling layers, then we apply a desnse layer that outsputs the 15 key facial features (30 points in total because its x, y coordinates). The model uses Mean Squre Error (MSE) loss, which is essentially the training method that tells the model how much it is off from the actual value. This is an easy baseline model that can be improved on.

In [9]:
```
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(96, 96, 1)),
    layers.MaxPooling2D(2),
```

```
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D(2),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(30)
])

model.compile(optimizer='adam', loss=masked_mse_loss)
```

```
C:\Users\Grant\AppData\Local\Programs\Python\Python312\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

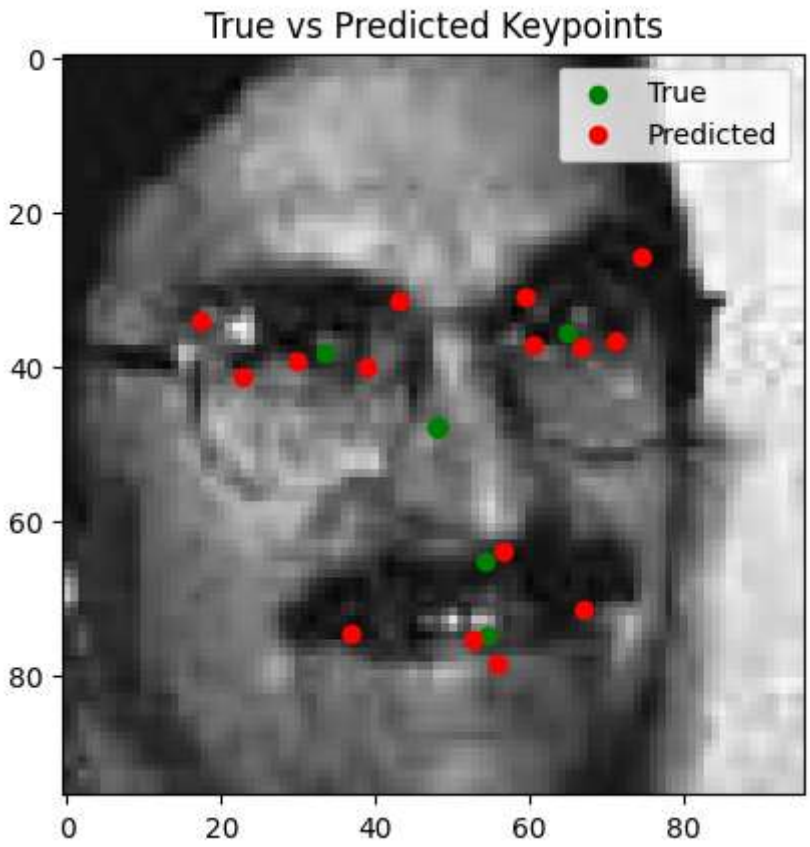
In [10]:
```
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=10,
    shuffle=True
)
```

```
Epoch 1/10
199/199 ━━━━━━━━━━━━━━━━━━━━ 45s 221ms/step - loss: 0.3391 - val_loss: 0.0095
Epoch 2/10
199/199 ━━━━━━━━━━━━━━━━━━━━ 44s 222ms/step - loss: 0.0053 - val_loss: 0.0048
Epoch 3/10
199/199 ━━━━━━━━━━━━━━━━━━━━ 40s 203ms/step - loss: 0.0039 - val_loss: 0.0048
Epoch 4/10
199/199 ━━━━━━━━━━━━━━━━━━━━ 38s 192ms/step - loss: 0.0036 - val_loss: 0.0038
Epoch 5/10
199/199 ━━━━━━━━━━━━━━━━━━━━ 43s 217ms/step - loss: 0.0028 - val_loss: 0.0037
Epoch 6/10
199/199 ━━━━━━━━━━━━━━━━━━━━ 45s 225ms/step - loss: 0.0026 - val_loss: 0.0034
Epoch 7/10
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 197ms/step - loss: 0.0021 - val_loss: 0.0036
Epoch 8/10
199/199 ━━━━━━━━━━━━━━━━━━━━ 42s 210ms/step - loss: 0.0020 - val_loss: 0.0033
Epoch 9/10
199/199 ━━━━━━━━━━━━━━━━━━━━ 44s 219ms/step - loss: 0.0017 - val_loss: 0.0034
Epoch 10/10
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 198ms/step - loss: 0.0017 - val_loss: 0.0034
```

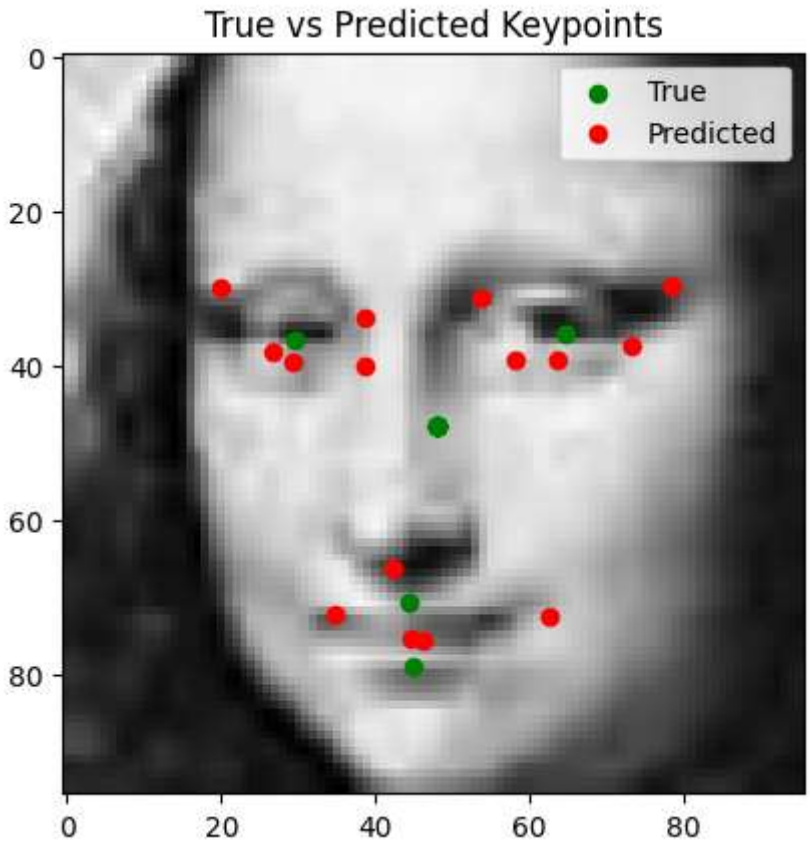
In [11]:
```
for i in range(3):
    image = X_val[i].squeeze()
    true_keypoints = y_val[i].reshape(-1, 2) * 48 + 48
    prediction = model.predict(X_val[i:i+1])
    predicted_keypoints = prediction[0].reshape(-1, 2) * 48 + 48

    plt.imshow(image, cmap='gray')
    plt.scatter(true_keypoints[:, 0], true_keypoints[:, 1], color='green', label='T
    plt.scatter(predicted_keypoints[:, 0], predicted_keypoints[:, 1], color='red',
    plt.legend()
    plt.title("True vs Predicted Keypoints")
    plt.show()
```

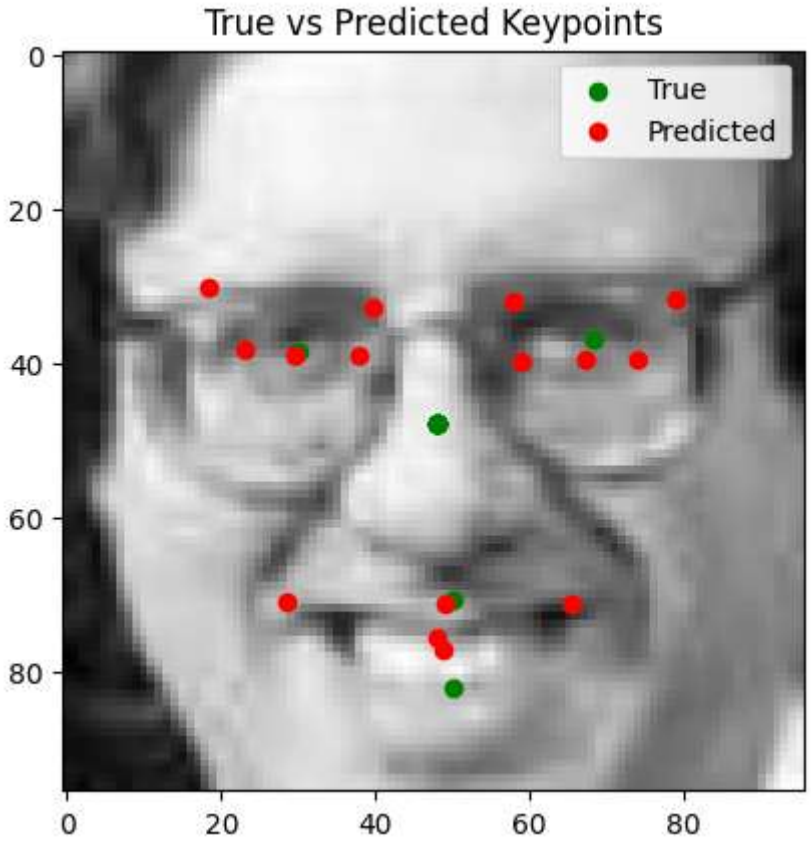
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
```

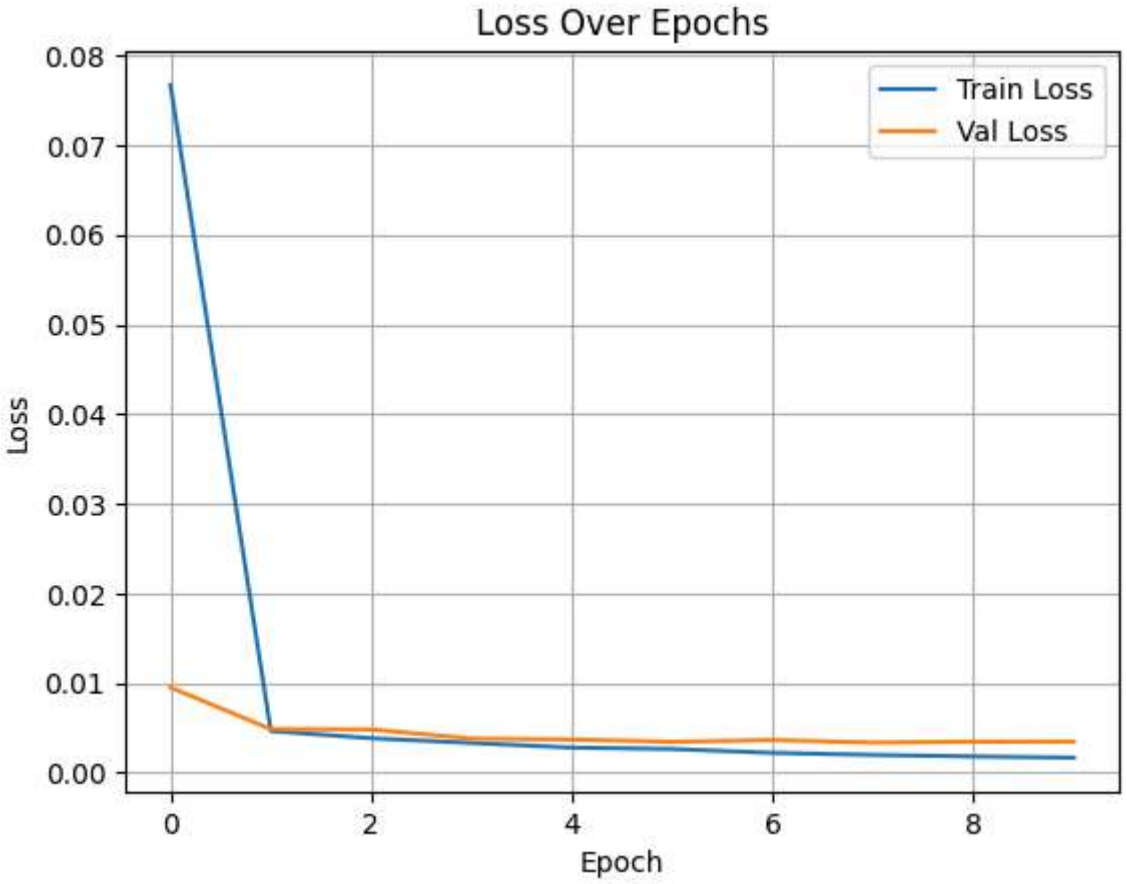



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
```



As you can see the guesses are good but it could still be better. Its also good with our model you can see on some of the photos that there arent true labels on some of the photos and our predicted values are guessing the key features and they seem valid! Lets see how are training loss is doing.

In [12]:
```
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



The graph clearly shows that the training and validation loss both go down as the number of epochs increase. This is overall good performance but lets see if we can further optimize the model and get better results.

Hyper Tuning

These results are good but as you can tell we are still slight off. So we need to increase the performance of the model by tuning it. To do this we will include a dropout layer that helps with the model overfitting the data. We will also make the dense layer deeper which lets the model learn more complex patterns. Also going to increase the number of epochs from 10 to 20 which should help the model greatly.

In [13]:
```
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(96, 96, 1)),
    layers.MaxPooling2D(2),
    layers.Dropout(0.05),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D(2),
    layers.Dropout(0.1),

    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(30)
```

```
])


model.compile(optimizer='adam', loss=masked_mse_loss)

history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=20,
    shuffle=True
)
```

```
Epoch 1/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 30s 145ms/step - loss: 0.3429 - val_loss: 0.0254
Epoch 2/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 29s 146ms/step - loss: 0.0093 - val_loss: 0.0235
Epoch 3/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 30s 149ms/step - loss: 0.0080 - val_loss: 0.0147
Epoch 4/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 31s 157ms/step - loss: 0.0071 - val_loss: 0.0187
Epoch 5/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 29s 146ms/step - loss: 0.0066 - val_loss: 0.0090
Epoch 6/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 29s 145ms/step - loss: 0.0059 - val_loss: 0.0100
Epoch 7/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 30s 150ms/step - loss: 0.0056 - val_loss: 0.0078
Epoch 8/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 29s 145ms/step - loss: 0.0051 - val_loss: 0.0078
Epoch 9/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 29s 144ms/step - loss: 0.0046 - val_loss: 0.0067
Epoch 10/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 140ms/step - loss: 0.0046 - val_loss: 0.0059
Epoch 11/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 140ms/step - loss: 0.0044 - val_loss: 0.0064
Epoch 12/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 138ms/step - loss: 0.0040 - val_loss: 0.0057
Epoch 13/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 141ms/step - loss: 0.0038 - val_loss: 0.0055
Epoch 14/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 138ms/step - loss: 0.0036 - val_loss: 0.0043
Epoch 15/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 140ms/step - loss: 0.0033 - val_loss: 0.0042
Epoch 16/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 140ms/step - loss: 0.0030 - val_loss: 0.0043
Epoch 17/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 139ms/step - loss: 0.0031 - val_loss: 0.0042
Epoch 18/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 138ms/step - loss: 0.0029 - val_loss: 0.0042
Epoch 19/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 140ms/step - loss: 0.0026 - val_loss: 0.0040
Epoch 20/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 28s 141ms/step - loss: 0.0025 - val_loss: 0.0039
```

Okay so the results are worse. We get a worse training loss and validation loss. This is fine though I think after trying to improve the model several times that this 'optimized model'

may have over complicated the model and learning process. So to fix optimize the model more we will just use the first model again and increaes the number of epochs to 20.

In [14]:
```
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(96, 96, 1)),
    layers.MaxPooling2D(2),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D(2),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(30)
])

model.compile(optimizer='adam', loss=masked_mse_loss)
```

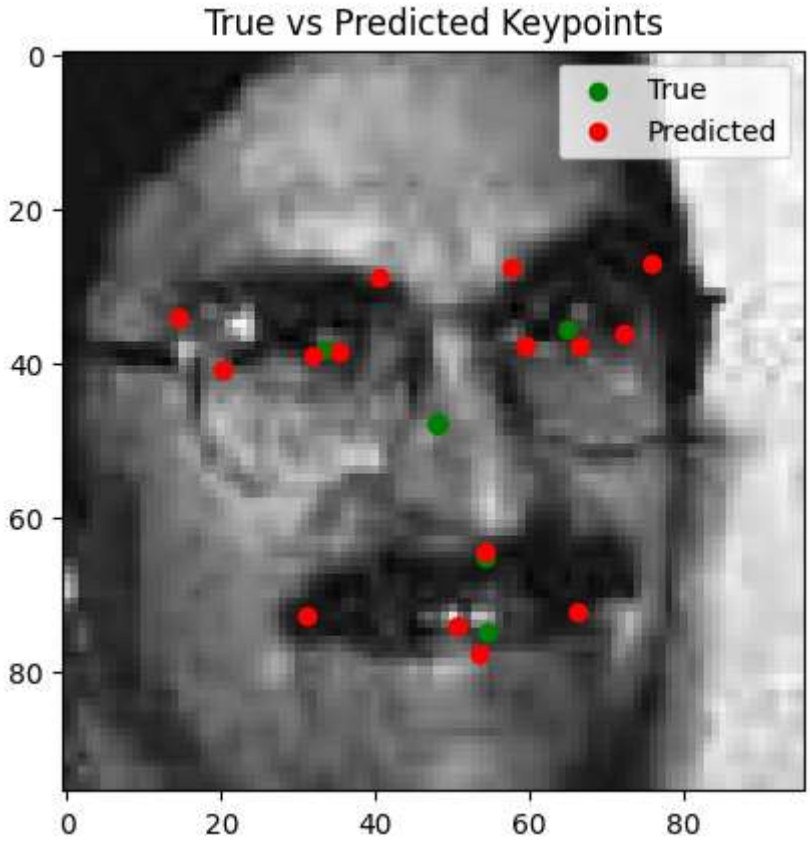
In [15]:
```
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=20,
    shuffle=True
)
```

```
Epoch 1/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 40s 198ms/step - loss: 0.0936 - val_loss: 0.0052
Epoch 2/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 195ms/step - loss: 0.0041 - val_loss: 0.0040
Epoch 3/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 198ms/step - loss: 0.0032 - val_loss: 0.0037
Epoch 4/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 197ms/step - loss: 0.0029 - val_loss: 0.0035
Epoch 5/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 196ms/step - loss: 0.0024 - val_loss: 0.0031
Epoch 6/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 197ms/step - loss: 0.0022 - val_loss: 0.0034
Epoch 7/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 198ms/step - loss: 0.0018 - val_loss: 0.0031
Epoch 8/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 197ms/step - loss: 0.0016 - val_loss: 0.0032
Epoch 9/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 40s 199ms/step - loss: 0.0015 - val_loss: 0.0031
Epoch 10/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 198ms/step - loss: 0.0013 - val_loss: 0.0032
Epoch 11/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 39s 198ms/step - loss: 0.0012 - val_loss: 0.0032
Epoch 12/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 43s 216ms/step - loss: 9.5441e-04 - val_loss: 0.0033
Epoch 13/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 44s 220ms/step - loss: 8.9983e-04 - val_loss: 0.0034
Epoch 14/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 43s 214ms/step - loss: 8.1724e-04 - val_loss: 0.0034
Epoch 15/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 46s 230ms/step - loss: 7.3821e-04 - val_loss: 0.0033
Epoch 16/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 48s 240ms/step - loss: 6.6693e-04 - val_loss: 0.0032
Epoch 17/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 44s 223ms/step - loss: 5.5908e-04 - val_loss: 0.0035
Epoch 18/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 44s 222ms/step - loss: 5.6873e-04 - val_loss: 0.0034
Epoch 19/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 44s 218ms/step - loss: 5.2341e-04 - val_loss: 0.0033
Epoch 20/20
199/199 ━━━━━━━━━━━━━━━━━━━━ 44s 222ms/step - loss: 4.3730e-04 - val_loss: 0.0033
```

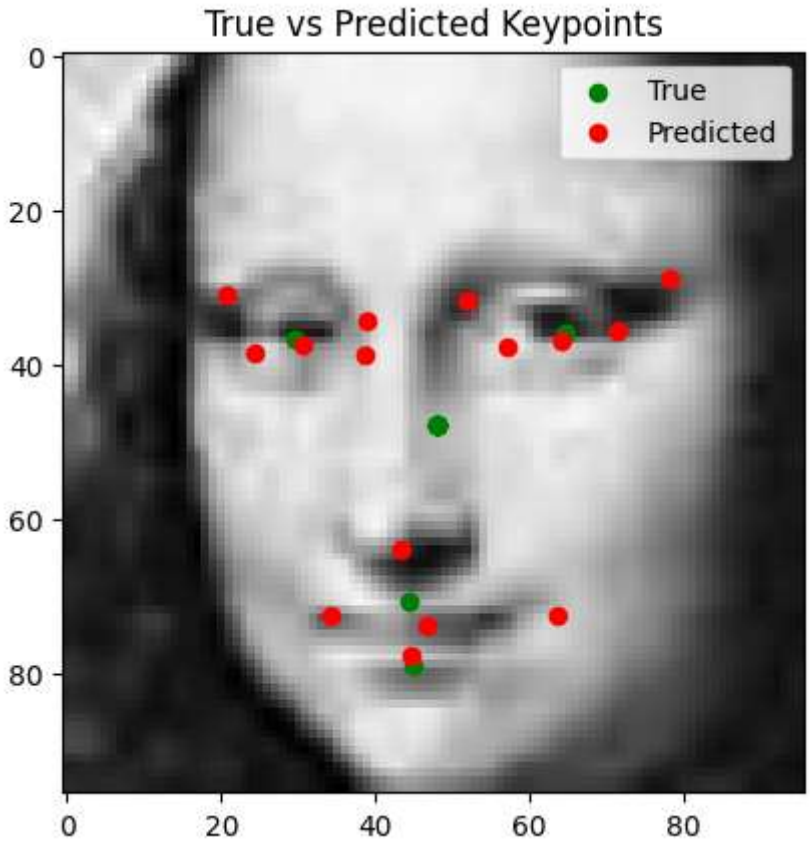
In [16]:
```python
for i in range(3):
    image = X_val[i].squeeze()
    true_keypoints = y_val[i].reshape(-1, 2) * 48 + 48
    prediction = model.predict(X_val[i:i+1])
    predicted_keypoints = prediction[0].reshape(-1, 2) * 48 + 48

    plt.imshow(image, cmap='gray')
    plt.scatter(true_keypoints[:, 0], true_keypoints[:, 1], color='green', label='T
    plt.scatter(predicted_keypoints[:, 0], predicted_keypoints[:, 1], color='red',
    plt.legend()
    plt.title("True vs Predicted Keypoints")
    plt.show()
```

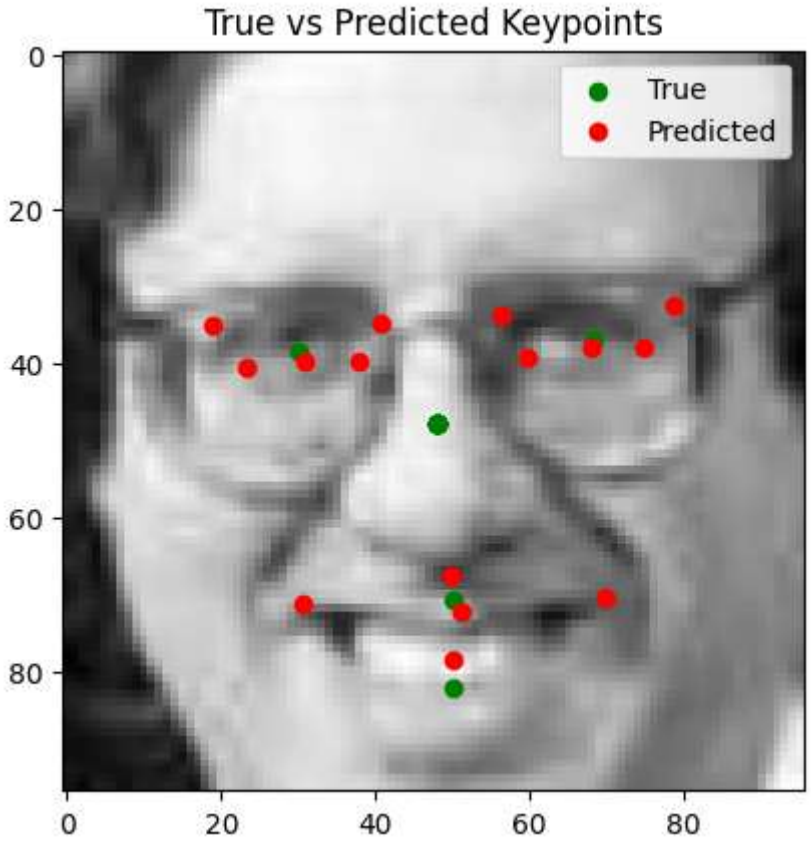
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
```

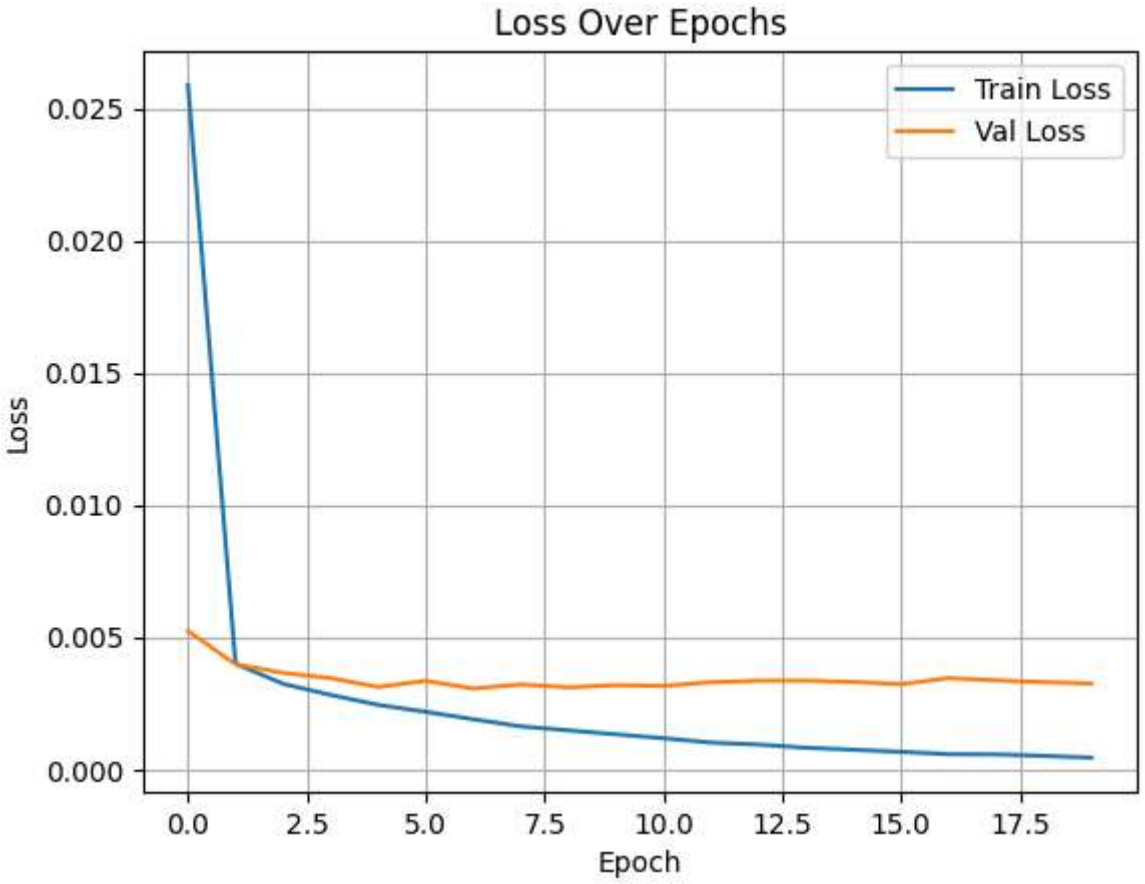



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
```



As you can see we are fitting the data extremely well. It is clearly outlining the major features even on images with horrible quality. Slight differences but overall really good results.

In [17]:
```
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



As you can see the validation loss decreases with the number of epochs and is overall better than the first test with 10 epochs. Another note is the training loss greatly decreases as you increase the number of epochs getting to a value of 5.9090e-04 at 20 epochs.

Results

We compared 2 CNN models in this project. The first was a CNN model with two convolution layers and a dense layer output which was orignially trained with 10 epochs giving us a validation loss around 0.0027. This showed we had good results but we were still slightly off, so we tried to further optimize the model. We then tried to make the model with an additional convolution layer, increased filters, and a heavier dropout to try and improve the generalization but the results were worse with a validation loss of about 0.0030. I think this was because it made the model too complicated and ruined results. So, to improve on the first model we increased the number of epochs to 20 and this decreased our validation loss to 0.0026 but it also greatly decreases the training loss from the 10 epoch test. This means that the model is learning and fitting the data better overtime. Overall, the first model performed better than the more complicated version and we increased the amount of training to make the results better

Conclusion

In this project we designed a CNN model to detect key facial features on images. We compared two CNN architectures: one simple one with two convolution layers, and a more complex model with additional layers and dropouts. For both of these models we used a masked loss to train the model even on incomplete data by ignoring missing keypoints in each photo that was key facial features. This helped the model focus on learning the available data without trying to train on data that wasnt there. It was discovered that the simplier model performed better and so we trained the simplier model further.

After increasing the training time on the simplier model from 10 to 20 epochs we achieved a great performance with a validation loss of 0.0026. The images of the predicted values compared to the actual values were extremely similar and the results were extremely promising.

In the future to increase the model peroformance it would be great to have more data with all the facial features included. It would also be nice to increase the number of epoch from 20 to something in the 100s. It would also be good to include some flipping or rotating the data so the model can learn more general info better.

Overall the results from this project were great and I think we accomplished what we set out to do.