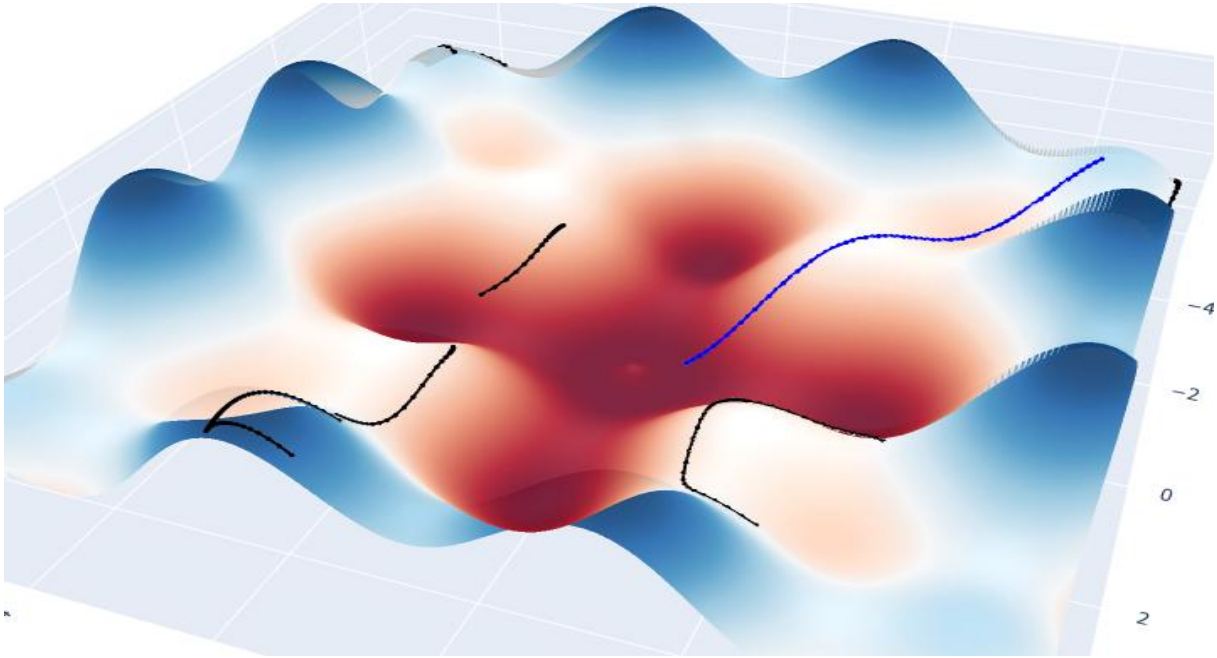
MSc Software Design with AI
Advanced Machine Learning (AL_KSAIM_9_1)
Assignment 2– Emotion Recognition



various gradient descent trajectories with blue being optimal and black not escaping local minima source

Facial Emotion Recognition with Data Augmentation

Andrew Nolan

A00325359@student.tus.ie

15th May 2025

Contents

Introduction

I started this project locally, with a structured folder. The code seen in the submitted Jupyter notebook is code pulled from multiple notebooks. I didn't realise while starting this that collab doesn't allow you to work with multiple files and a persistent folder structure. Hence, you will see code that I've written to automate making the folder structure and pulling data from Kaggle.

At the time of porting to collab, I didn't have much time to rewrite more Collab friendly code. However, you can see all commits and the evolution of this project on my github repository here.

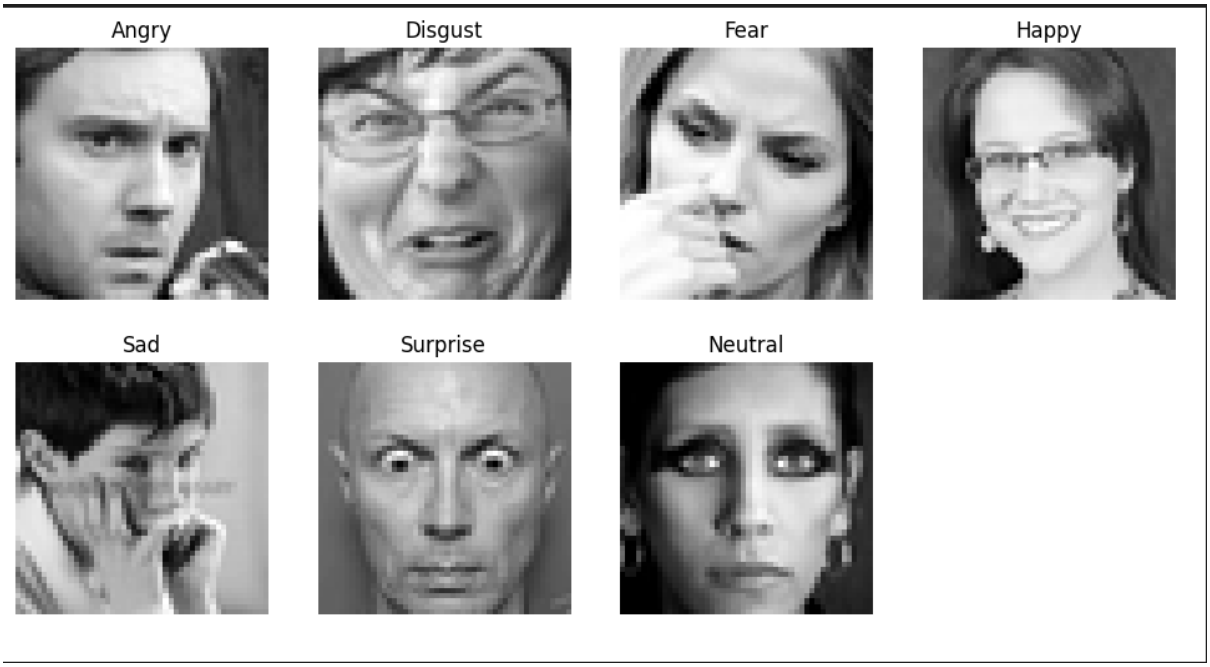
Data Exploration

First thing I did was load in the dataset. It comes as a DataFrame with two main columns: emotion (which is an integer from 0 to 6) and pixels, which is a long string of 48x48 greyscale values separated by spaces.

To actually work with the image data, I split the pixel strings into numbers and converted them into numpy arrays using:

```
x = df.pixels.str.split().explode().astype(int).values.reshape(-1,48,48)
```

To get a feel for what the data looked like, I grabbed one image per emotion and plotted them in a grid. This helped just to check visually if they seemed sensible and if the labels looked like they matched the expression.

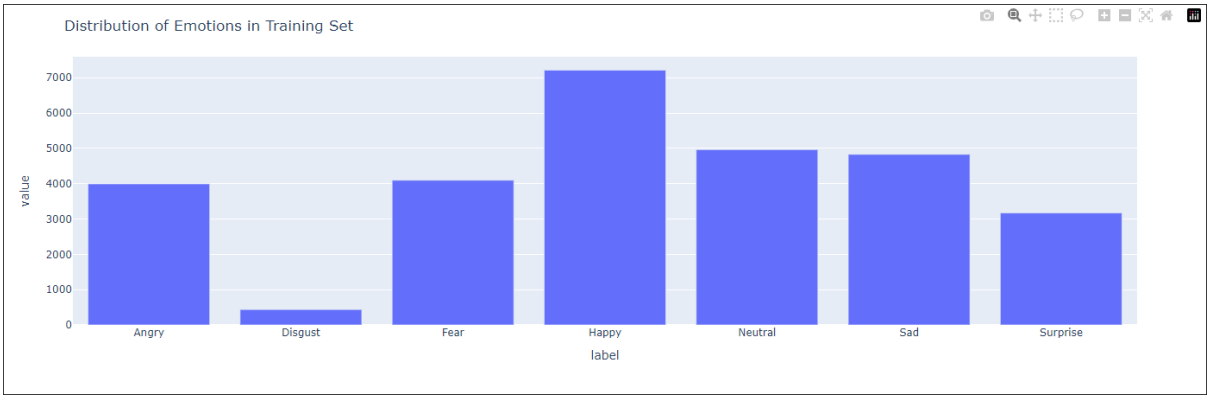


After that, I looked into the distribution of emotions across the dataset. I mapped the integer labels to their actual meanings and counted how many samples belonged to each class. Here's what I found:

- Happy: ~25%
- Neutral: ~17%
- Sad: ~17%
- Fear: ~14%
- Angry: ~14%
- Surprise: ~11%
- Disgust: ~1.5%

So the dataset is definitely imbalanced. Disgust in particular has a very low count compared to the rest, which might make it harder for the model to learn that class properly.

I also plotted the distribution as a bar chart to make it a bit clearer:



This confirmed the class imbalance visually and highlighted that some emotions (like Happy and Neutral) dominate the dataset, while others like Disgust are barely present. Something to keep in mind when training.

Data Preprocessing

I started by splitting the dataset into training and test sets. The test set was left untouched for now - I'll only be using it right at the end for final evaluation.

Once I had the training data separated, I noticed the class imbalance from earlier would likely be an issue. To deal with that, I balanced the training set so that each emotion had the same number of samples. I did this by duplicating samples from the underrepresented classes using sklearn's resample method. Here's the function I used:

```
from sklearn.utils import resample

def balance_labels(df, label_col):
    max_count = df[label_col].value_counts().max()
    balanced = pd.concat([
        resample(group, replace=True, n_samples=max_count, random_state=42)
        for _, group in df.groupby(label_col)
    ])
    return balanced.sample(frac=1, random_state=42).reset_index(drop=True)

df_balanced = balance_labels(df_train, 'emotion')
```

This gave me a nicely balanced training set, where each emotion is equally represented. I saved the processed data at this stage so I could load it later when training.

For now, I didn't touch the scaling part - I handle that further down the pipeline using a custom class during training. I'll explain that in the training section. I then saved down the training data.

Data Augmentation

The goal here was to apply a set of data augmentation strategies on the balanced training set to both improve generalisation and help deal with overfitting. I loaded in the balanced training data from the previous step.

First, I defined a few augmentation functions:

- horizontal_flip() – mirrors the image
- random_rotation() – rotates the image at 90° intervals
- random_brightness() and random_contrast() – tweak pixel intensity
- random_noise() – adds Gaussian noise
- random_translation() – shifts the image along x and y

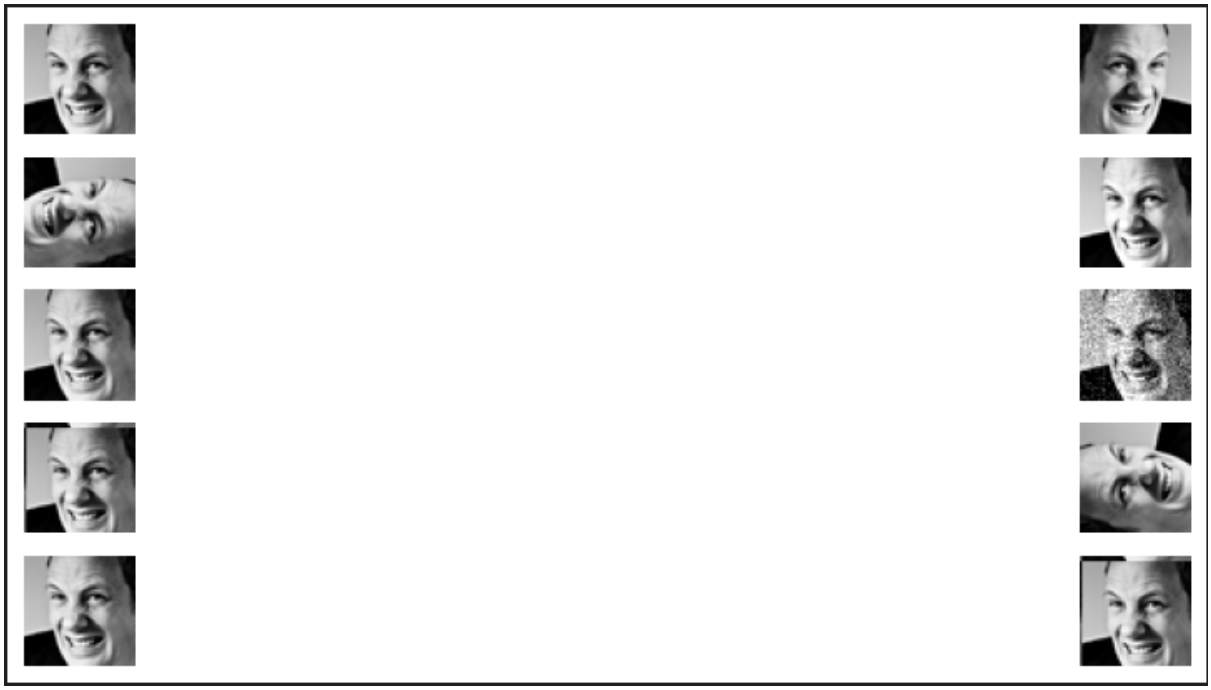
I tested each function individually on a test image for a sanity check.

Then I wrote a wrapper function, augment_image(image), that applies all six augmentations to a single image, and then applies three more at random using combinations of the previous ones. The result is a 10x increase in data per image - 1 original + 9 augmented. So if I start with 1 image, I end with 10.

```
def augment_image(image:np.ndarray)-> np.ndarray:
    '''
    one 48x48 image is passed
    returns 10 augmented images

    shape of return will be (10, 48, 48). Goal is then to have overall shape as
(len(df),10, 48, 48)
    Then it can be exploded and then flattend down to the string again
    '''
    augmented_images = []

    augmented_images.append(image)
    operations = [
        horizontal_flip,
        random_rotation,
        random_brightness,
        random_contrast,
        random_noise,
        random_translation,
    ]
    for operation in operations:
        augmented_images.append(operation(image))

    # Randomly select 3 images from the augmented images
    selection = np.random.choice(list(range(7)), 3, replace=False)
    for idx in selection:
        # randomly select a function from the list
        fn = np.random.choice(operations)
        augmented_images.append(fn(augmented_images[idx]))
    return np.array(augmented_images)
```

I tested this approach on a small slice of 3 images, just to confirm it works. The results showed the correct shape and variety.

```
res = augment_image(testImage)


fig, axes = plt.subplots(5, 2, figsize=(15, 5))
for ax, img in zip(axes.flatten(), res):
    ax.imshow(img, cmap="gray")
    ax.axis("off")
```



I'm not sure why the spacing for the plots is so strange here, but it wasn't that important to me. I just needed to do a sense check.

Since the FER2013 format stores each image as a string of space-separated values, I converted each augmented image back to this format using .flatten() and join(). I verified that everything looked fine and that the new images retained the original emotion labels.

```
testDF = pd.DataFrame(data = dict(emotion = df.emotion[:3].values.tolist(), pixels =
df.pixels[:3].values.tolist()))
augmentations = []
for augmentedBatch in testShape:
    res = ''
    for img in augmentedBatch:
        res += ' '.join(map(str, img.flatten())) + '\n'
    res = res.strip()
    augmentations.append(res)

testDF["augmentated_pixels"] = augmentations

(
    testDF
    .assign(augmentated_pixels = lambda s: s.augmentated_pixels.str.split("\n"))
    .explode("augmentated_pixels")

)
```

	emotion	pixels	augmentated_pixels
0	1	168 168 167 166 166 166 165 165 165 169 152 56...	168 168 167 166 166 166 165 165 165 169 152 56...
0	1	168 168 167 166 166 166 165 165 165 169 152 56...	1 3 2 2 32 19 16 29 43 55 53 52 55 56 55 57 70...
0	1	168 168 167 166 166 166 165 165 165 169 152 56...	1 2 4 3 3 5 10 13 9 3 4 9 12 7 2 5 13 10 6 8 1...
0	1	168 168 167 166 166 166 165 165 165 169 152 56...	87 87 86 86 86 86 85 85 85 87 79 29 74 91 92 9...
0	1	168 168 167 166 166 166 165 165 165 169 152 56...	154 154 154 153 153 153 152 152 152 155 144 79...
0	1	168 168 167 166 166 166 165 165 165 169 152 56...	151 167 151 169 140 149 152 166 139 159 157 45...
0	1	168 168 167 166 166 166 165 165 165 169 152 56...	4 7 8 9 8 5 3 3 5 7 6 3 5 0 116 253 234 230 21...
0	1	168 168 167 166 166 166 165 165 165 169 152 56...	2 4 5 5 5 3 1 1 3 4 3 1 3 0 76 166 153 150 139...
0	1	168 168 167 166 166 166 165 165 165 169 152 56...	168 168 167 166 166 166 165 165 165 169 152 56...
0	1	168 168 167 166 166 166 165 165 165 169 152 56...	178 158 106 156 163 164 147 164 101 159 162 52...
1	1	165 170 163 140 101 68 54 62 58 76 128 148 154...	165 170 163 140 101 68 54 62 58 76 128 148 154...
1	1	165 170 163 140 101 68 54 62 58 76 128 148 154...	34 56 34 38 73 46 44 59 53 58 67 47 51 50 55 5...
1	1	165 170 163 140 101 68 54 62 58 76 128 148 154...	34 40 44 50 52 46 34 34 34 33 41 44 40 40 41 3...
1	1	165 170 163 140 101 68 54 62 58 76 128 148 154...	234 241 232 199 143 96 76 88 82 108 182 210 21...
1	1	165 170 163 140 101 68 54 62 58 76 128 148 154...	175 181 172 146 100 62 45 55 50 71 132 155 162...
1	1	165 170 163 140 101 68 54 62 58 76 128 148 154...	166 178 144 138 109 77 45 65 69 85 111 154 159...
1	1	165 170 163 140 101 68 54 62 58 76 128 148 154...	52 47 45 41 171 165 153 113 99 103 94 63 43 23...
1	1	165 170 163 140 101 68 54 62 58 76 128 148 154...	16 44 16 21 65 31 28 47 40 46 57 32 37 36 42 3...
1	1	165 170 163 140 101 68 54 62 58 76 128 148 154...	102 49 25 49 146 158 210 148 113 117 78 54 79 ...
1	1	165 170 163 140 101 68 54 62 58 76 128 148 154...	157 158 144 84 87 41 110 16 119 137 148 161 13...
2	6	242 242 241 242 242 243 243 241 239 238 237 24...	242 242 241 242 242 243 243 241 239 238 237 24...
2	6	242 242 241 242 242 243 243 241 239 238 237 24...	244 245 245 245 246 245 244 245 247 247 245 24...
2	6	242 242 241 242 242 243 243 241 239 238 237 24...	140 152 159 159 150 141 134 137 143 83 83 101 ...
2	6	242 242 241 242 242 243 243 241 239 238 237 24...	156 156 155 156 156 157 157 155 154 153 153 15...
2	6	242 242 241 242 242 243 243 241 239 238 237 24...	255 255 255 255 255 255 255 255 255 255 255 25...
2	6	242 242 241 242 242 243 243 241 239 238 237 24...	255 219 233 221 224 227 255 222 219 235 255 24...
2	6	242 242 241 242 242 243 243 241 239 238 237 24...	241 242 240 240 240 239 242 237 203 192 169 16...
2	6	242 242 241 242 242 243 243 241 239 238 237 24...	174 174 172 174 174 175 175 172 171 170 170 17...
2	6	242 242 241 242 242 243 243 241 239 238 237 24...	242 241 240 240 240 242 240 240 239 239 238 23...
2	6	242 242 241 242 242 243 243 241 239 238 237 24...	244 242 242 244 244 241 237 236 232 233 242 24...

Then I plotted them again, just to make sure

```
test_ims = (
    testDF
    .assign(augmentated_pixels = lambda s: s.augmentated_pixels.str.split("\n"))
    .explode("augmentated_pixels")
    .augmentated_pixels.str.split().explode().astype(int).values.reshape(-1,48,48)
)

fig, axes = plt.subplots(5, 6, figsize=(16,16))
for ax, img in zip(axes.flatten(), test_ims):
    ax.imshow(img, cmap="gray")
```

```
    ax.axis("off")
```

yielding the following

Once I confirmed that everything worked, I ran the augmentation over the full balanced dataset. This scaled the dataset by 10× - took a while, but it finished fine. The final augmented dataset has the same structure as before but ten times the number of rows.

```
augmentedBatches = np.array([augment_image(img) for img in images])
print(augmentedBatches.shape) # shape should be (len(df),10,48,48)
augmentedImageStrings = []
for augmentedBatch in augmentedBatches:
    res = ''
    for img in augmentedBatch:
        res += ' '.join(map(str, img.flatten())) + '\n'
    res = res.strip()
    augmentedImageStrings.append(res)
    print("{}/{}".format(len(augmentedImageStrings), len(augmentedBatches)), end = '\r')
```

```
print(len(augmentedImageStrings))

df["augmentated_pixels"] = augmentedImageStrings

AugmentedDataFrame = (
    df
    .assign(augmentated_pixels = lambda s: s.augmentated_pixels.str.split("\n"))
    .explode("augmentated_pixels")
    # .reset_index(drop=True)
)

print(AugmentedDataFrame.shape)
print(df.shape)

AugmentedDataFrame = (
    AugmentedDataFrame
    .assign(pixels = AugmentedDataFrame.augmentated_pixels)
    .drop(columns = 'augmentated_pixels')
)

AugmentedDataFrame.to_pickle("../data/02 Processed/train.pkl")
```

So before I started this, the balanced data set had 40404 samples, but the augmented data frame had 404040 samples. And only about 100k of them are duplicates due to the randomness. This means that after dropping duplicates, cardinality of each class was more or less conserved. The AugmentedDataFrame was saved and used in training.

Model Development

Custom Scaler

Designing the CNN was a bit of a back-and-forth. I tried a few different setups - swapping in different activation functions, tweaking loss functions, filter sizes, depths, etc. Some converged faster, some generalised better. The one I eventually landed on gave me the best tradeoff - solid convergence speed without overfitting. It held up well on the test set, so I stuck with it.

Before feeding anything into the model though, I had to scale the data properly. sklearn's built-in scalers weren't quite cutting it for this - I needed it so that:

Suppose X is an array, containing input images, M_i, and anch M_i consists of (m_ijk), i in (1,n) and j,k in (1,48). I want mu_X to be a 48x48 matrix such that mu_X = (mu_jk) where mu_jk = mean(m_ijk) over i. Similarly with sigma_X. sklearn's StandardScaler wouldn't let me do this, so I made my own. The scaler then subtracts mu_X from X, element wise and divides by sigma_X, element wise.

I also experimented with a min max scaler but it was slower to converge than the standard scaler. I also made the scaler mistake-proof, ie it throws an error if you try to fit it more than once. Only one scaler should exist in this entire notebook.

```
class CustomStandardScaler(OneToOneFeatureMixin, TransformerMixin, BaseEstimator):
    def __init__(self, mu: float, sigma: float):
        self.mu = mu
        self.sigma = sigma
        self.__mean = None
        self.__std = None
        self.__fitted = False

    @property
    def _fitted(self)->bool:
        return self.__fitted

    @_fitted.setter
    def _fitted(self, value: bool):
        if self.__fitted:
            raise AttributeError("The 'fit' method has already been called.")
        if not isinstance(value, bool):
            raise TypeError("The 'fitted' attribute must be a boolean.")
        self.__fitted = value

    @property
    def _mean(self)->np.ndarray:
        '''
        mean of the passed data
        '''
        if self.__mean is None:
            raise AttributeError("The 'fit' method must be called before accessing the
mean.")
        return self.__mean

    @property
    def _std(self)->np.ndarray:
        '''
        std of the passed data
        '''
        if self.__std is None:
            raise AttributeError("The 'fit' method must be called before accessing the
std.")
        return self.__std

    def fit(self, X: np.ndarray)-> 'CustomStandardScaler':
        self._fitted = True
        self.__mean = X.mean(axis=0)
        self.__std = X.std(axis=0)
```

```
        return self

    def transform(self, X: np.ndarray)-> np.ndarray:
        newX = X - self._mean
        newX = (newX / self._std) * self.sigma
        return newX+self.mu

    def fit_transform(self, X:np.ndarray) -> np.ndarray:
        self.fit(X)
        return self.transform(X)

    def inverse_transform(self, X:np.ndarray) -> np.ndarray:
        newX = X - self.mu
        newX = (newX / self._std) * self.sigma
        return newX + self._mean
```

This gave me exactly what I wanted.

Custom Dataset

To keep everything tidy, I also made a custom Dataset class. This feeds into a PyTorch DataLoader later, but I wanted a bit more control than usual. So I passed in the scaler directly - or if none is passed, it just fits its own. Probably overengineered it a bit, but I was being paranoid about mutability and side effects.

I locked down the internals - everything that shouldn't be touched is private. I didn't want any weird surprises from accidental mutations halfway through training. Also made the transform optional, with a dummy identity fallback if nothing's passed in.

The class takes care of flattening the pixel strings into 48x48 images, scales them properly, and converts labels to one-hot vectors. Nothing too fancy, just clean and predictable.

The full setup made it easy to experiment later - I could just plug in different scalers or transforms as needed without touching anything else.

```
class Data(Dataset):
    _identityFunc = lambda x: x
    def __init__(self,
        data:pd.DataFrame,
        transform:transforms = None,
        scaler:BaseEstimator = None):

        if not scaler:
            scaler = CustomStandardScaler(0, 0.25)
        # making them private so can't be altered
        self.__transform = transform if transform else Data._identityFunc
        self.__Y = data['emotion'].astype(int).values.tolist()
```

```
        self.__X = data.pixels.str.split().explode().astype(np.uint8).values.reshape(-
1,48,48)
        self.__len = data.shape[0]

        try:
            self.__X = scaler.fit_transform(self.__X).astype(np.float32)
        except AttributeError:
            print("Already fit")
            self.__X = scaler.transform(self.__X).astype(np.float32)

    def __len__(self)->int:
        return self.__len

    def __getitem__(self, index)->tuple[np.ndarray,np.ndarray]:
        img:np.ndarray = self.__X[index]
        emotion:int = self.__Y[index]
        label = np.zeros((7,), dtype=np.float32)
        label[emotion] = 1
        return self.__transform(img), label
```

I opted to one-hot encode the variables. This was because it felt more natural. I am now aware the CrossEntropyLoss expects label indices, however I didn't realise this at the time. Which led to a lot of frustrating. This leads me onto the next custom class I made.

Custom Loss Function

As previously noted, I made this because I needed a loss function that expects a one-hot prediction and a one hot label. I also made it because CrossEntropyLoss takes the softmax itself – I found this to be overarching and inflexible, incase I wanted to use some analogue for softmax later down the line. Here's the code, see further description in the docstring.

The 1e-10 is a softening buffer, to prevent singularities with low (near zero) probabilities.

```
class CustomLoss(nn.Module):
    '''
    nn.CrossEntropyLosss is annoying me the way it expects labels
    it also does the softmax part itself (apparently), which also annoys me.

    this class expects two R^n vectors or two arrays of R^n vectors
    it then penalises the difference using negative log liklihood
    '''
    def __init__(self):
        super(CustomLoss, self).__init__()
    def forward(self, probabilites:torch.Tensor, target:torch.Tensor)->torch.Tensor:
        '''
        probabilites: R^n
        target: R^n
        '''
```

```
        return -torch.sum(target * torch.log(probabilites + 1e-10), dim=1).mean()
```

CNN Architecture

I started off with the base template given to us, just two conv layers and a couple of fully connected ones. Pretty shallow. Felt like a good place to begin but obviously had to grow it if I wanted anything decent out of it.

```
class EmotionRecognitionCNN(nn.Module):
    def __init__(self, num_classes):
        super(EmotionRecognitionCNN, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.fc1 = nn.Sequential(
            nn.Linear(32 * 12 * 12, 256),
            nn.ReLU(),
        )
        self.fc2 = nn.Sequential(
            nn.Linear(256, num_classes),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

So from there, I made a bunch of incremental changes, just building things up and trying to figure out what helped. First thing I did was add an extra fully connected layer before the softmax - with a sigmoid activation. My thinking was maybe it'd give the network space to learn something useful in terms of combining probabilities before making the final decision. I kept softmax in the final layer the whole way through.

Tried out different activation functions too - I gave Tanh a shot in the first layer early on in development but I didn't really like how it trained, it just felt slow. Swapped it back to ReLU. I experimented with sigmoids in the intermediate layers, my thinking was on/off

gates. However, it didn't really work and I read online somewhere that ReLU is the best of CNN, so I switched back to that.

I started noticing vanishing gradient problems, so I switched the ReLU activation function in the first fully connected layer to LeakyReLU. I didn't use LeakyReLU in any of the convolutional layers as it doesn't make sense to have negative pixel values. That actually made a noticeable difference. The network trained smoother and a bit quicker ie less plateaus.

Then I started adding BatchNormalisation after every layer, except the final two. That really helped a lot with training speed and convergence, it was hitting 100% on the training set after about 15 epochs. But the downside was, it started overfitting fast. Like, training accuracy would shoot up to 100% and test accuracy would just stall, while the Loss would begin to increase.

So to fix that, I brought in Dropout. I added it everywhere just to try slow the model down a bit. I also had more than enough neurons at this point, so I set the probabilities for the BN quite high to force it to make sub-connections. It slowed down training and training accuracy came down a bit, but test accuracy actually improved. This might be because it started to make multiple internal networks that would ensemble in the final layers.

After that, I threw in a residual connection. Honestly not sure if it made a massive difference. My reasoning was that it'd help the model keep some original signal as the conv blocks got deeper. Makes sense in theory anyway, so I left it in.

Then I started pushing the architecture a bit - bumping up the number of channels in the early conv layers and seeing how far I could go before the crashing my RAM. This gave me a nice boost in test performance too. Bigger feature maps early on just seemed to help the model pick up more useful patterns.

Final model looks like this:

```
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=5, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
```

```
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25)
        )
        self.residual = nn.Sequential(
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32)
        )
        self.fc1 = nn.Sequential(
            nn.Linear(32 * 12 * 12, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(),
            nn.Dropout(0.5)
        )
        self.fc2 = nn.Sequential(
            nn.Linear(256, 100),
            nn.Sigmoid(),
            nn.Dropout(0.5)
        )
        self.fc3 = nn.Sequential(
            nn.Linear(100, num_classes),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)

        residual = x
        x = self.residual(x)
        x += residual
        x = nn.ReLU()(x)

        x = x.flatten(start_dim=1)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x
```

It ended up being a pretty solid architecture. The tweaks weren't random - everything I added had a reason behind it, and the training/test behaviour backed that up. In the end I was able to achieve 60% accuracy on the test set. Which seems low, however, after looking at some of the images – they can be quite ambiguous, I'm not sure If a human brain would be able to get 60% accuracy.

Model Training and Evaluation

I loaded in the augmented training data (train.pkl), the test set (testData.pkl), and the original unaltered dataset straight from CSV. One small issue though - the original data still had the test set in it. So I removed those rows using an outer join and filtered out anything that showed up in both.

After that, I ran into a bit of a RAM wall. The augmented training data was huge. I kept hitting memory limits, so I had to shrink it. First thing I did was drop duplicates, just to cut down the size. Then I took a random 60% slice of the dataset and kept only that. The rest got deleted to free up RAM.

But I didn't stop there. With the data size now manageable, I needed to make sure it was balanced. Otherwise, accuracy would mean nothing. I checked which class had the fewest samples, and then just trimmed all the others down to match that number. That way, every class was represented equally.

After doing this, each emotion class had exactly 23,311 samples. Which is more than enough. RAM stayed happy, the data was clean, and I was good to go.

Here's the code (not as it appears in the notebook, but lumped together)

```
TrainingData = pd.read_pickle('../data/02 Processed/train.pkl')
FinalTestData = pd.read_pickle('../data/02 Processed/testData.pkl')
unalteredDataset = pd.read_csv('../data/01 Raw/train.csv')

# remove the predestined test set
unalteredDataset = (
    unalteredDataset.merge(FinalTestData, how = 'outer', indicator = True)
    .query("_merge == 'left_only'")
    .drop(columns = '_merge')
)

# Reduce size of TrainingData
TrainingData = TrainingData.drop_duplicates().reset_index(drop = True)
big,small = train_test_split(TrainingData, test_size=0.6, stratify=TrainingData['emotion'],
random_state=42)
TrainingData = small
del big # save memory


# Then I made sure that the dataset was completely balanced with
# make sure each class has equal representation so accuracy isn't a misleading metric
emotionMin,Min = (
    TrainingData.emotion
    .value_counts()
    .pipe(lambda s:(s.idxmin(),s.min()))
```

```
)

idx = TrainingData.index.to_list()
np.random.shuffle(idx)
TrainingData = TrainingData.loc[idx]
TrainingData.reset_index(drop = True)

tmp = pd.DataFrame()
for emotion in TrainingData.emotion.unique():
  tmp = pd.concat([tmp,TrainingData.query(f"emotion == {emotion}").iloc[:Min]])

TrainingData = tmp.copy()
del tmp
```

Before training, I wrote training loop functions that prints loss and accuracy for both training and test data every epoch and also computes the backward pass. The results are then plotted. The loop is also wrapped in basic error handling, so the graph still plots if there's an error.

```
def train(model, dataloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    for images, labels in dataloader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

        # get accuracy
        predicted = outputs.argmax(axis=1)
        target = labels.argmax(axis=1)
        correct = (predicted == target).sum().item()
        accuracy = correct / labels.size(0)
    return running_loss / len(dataloader), accuracy

def evaluate(model, dataloader, criterion, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
      for images, labels in dataloader:
          images, labels = images.to(device), labels.to(device)
          outputs = model(images)
```

```
            loss = criterion(outputs, labels)
            running_loss += loss.item()
            predicted = outputs.argmax(axis=1)
            total += labels.size(0)
            correct += (predicted == labels.argmax(axis = 1)).sum().item()

    return running_loss / len(dataloader), correct / total
```

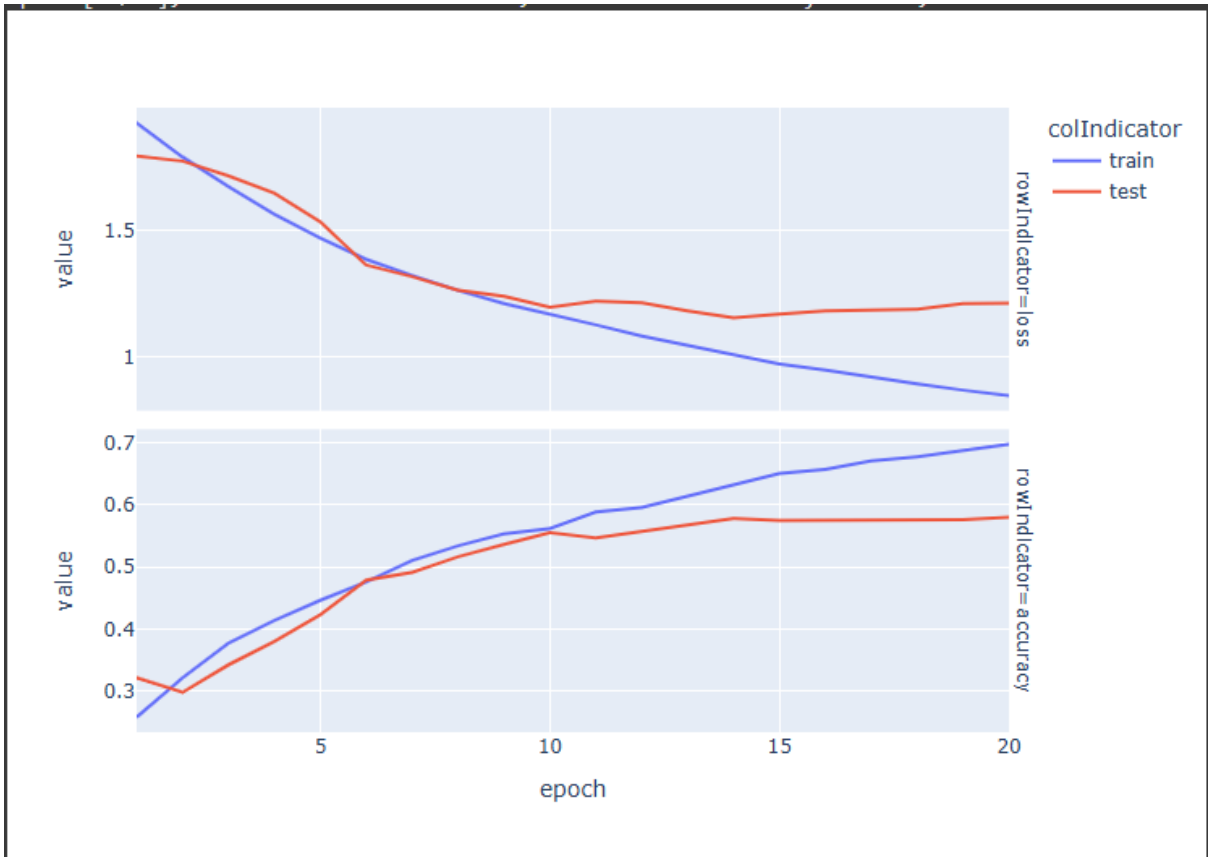
the code above is more or less the same as what was given, just tweaked slightly to work with my custom loss function and also return test accuracy.

This is the training loop function. It trains it for given data and number of epochs and plots the validation curves.

```
def doTrainingLoop(num_epochs:int,
                   _model,
                   _train_loader,
                   _test_loader,
                   _criterion,
                   _optimizer,
                   _device):
  trainingResults = list()

  try:
    for epoch in range(num_epochs):
        train_loss, train_accuracy = train(_model, _train_loader, _criterion, _optimizer, _device)
        test_loss, accuracy = evaluate(_model, _test_loader, _criterion, _device)

        results = dict(zip(['epoch','train_loss', 'train_accuracy', 'test_loss', 'test_accuracy'],
                           [epoch+1,train_loss, train_accuracy, test_loss, accuracy]))
        trainingResults.append(results)
        print(f"Epoch [{epoch+1}/{num_epochs}], \
        Train Loss: {train_loss:.4f}, \
        Train Accuracy: {train_accuracy:.4f}, \
        Test Loss: {test_loss:.4f}, \
        Test Accuracy: {accuracy:.4f}")
  except KeyboardInterrupt:
    pass
  except Exception as e:
    print('an exception occured :{}\ntb:\n{}'.format(e,
                                                    ''.join(tb.format_exception(type(e), e, e.__traceback__)))
    )
    pass


  return (
      pd.DataFrame(trainingResults)
      .melt(id_vars = 'epoch')
      .assign(colIndicator = lambda s: s.variable.apply(lambda x: 'train' if 'train' in x else 'test'))
      .assign(rowIndicator = lambda s: s.variable.apply(lambda x: 'loss' if 'loss' in x else 'accuracy'))
```

```
    .plot(x = 'epoch',
          facet_row = 'rowIndicator',
          y = 'value',
          color = 'colIndicator',
          backend = 'plotly')
    .update_yaxes(matches = None)
)
```

Here's an example



Then I initialised my data objects, dataloaders and scaler along with the CNN.

With the test data sorted and the model ready, I moved on to training using the enriched dataset. This was the augmented and class-balanced version of the training data.

I went with a batch size of 15,000 here, which was mostly dictated by my ram constraints – I couldn't train on the full set or I'd run out of ram and I couldn't train on smaller batches or I'd run out of GPU ram. It was actually quite quick per epoch. Only taking about 5 minutes for 20 epochs on ~160k samples.

For the optimiser, I used Adam with a learning rate of 0.01 which learned quickly.

The training loop ran for 20 epochs. Loss and accuracy for both train and test sets were logged each time and plotted with Plotly at the end. The results showed exactly what I was hoping for - consistent improvement on both sets through the first half, with test accuracy peaking just shy of 59% by the end. After around epoch 14, test loss started increasing, which suggests the model was starting to overfit.

Once the model had learned those general patterns, I switched over to the unaltered (realistic) dataset. This step was all about fine-tuning. I dropped the learning rate to 0.001 to avoid wrecking what it had already learned and to slowly nudge the model toward the real data distribution.

Straight away, the model had decent accuracy on the unaltered data - ~75% training accuracy in the first epoch - and even better test accuracy than the previous phase. Over the next 40 epochs, it nudged test accuracy up toward ~60.6% while pushing training accuracy to nearly 80%. The key takeaway here is that the model didn't forget what it learned from the augmented data - it just specialized to the real thing.

This whole two-step approach let me balance generalization and realism.

Finally, I took a picture of me smiling, made it 48x48 and passed it to the model and it predicted happy with 99.83% confidence. You will also see that in the notebook.