Review

Meets Specifications

You've done a fantastic job completing the Facial Keypoints Detection Project!

Facial Keypoints Detection is a well-known machine learning challenge. You might want to check the resources below to find ways you can further improve your model's performance:

- Berkeley team's attempt at beating the Facial Keypoints Detection Kaggle competition
- Facial Keypoints Detection using the Inception model
- Facial Key Points Detection using Deep Convolutional Neural Network - NaimishNet

Keep up the good work! 😊 👍

Files Submitted

The submission includes models.py and the following Jupyter notebooks, where all questions have been answered and training and visualization cells have been executed:

2. Define the Network Architecture.ipynb, and

3. Facial Keypoint Detection, Complete Pipeline.ipynb.

Other files may be included, but are not necessary for grading purposes. Note that all your files will be zipped and uploaded should you submit via the provided workspace.

> All the required files have been submitted and all questions have been answered. Good job!

`models.py`

Define a convolutional neural network with at least one convolutional layer, i.e. `self.conv1 = nn.Conv2d(1, 32, 5)`. The network should take in a grayscale, square image.

You have nicely configured a functional convolutional neural network along with the feedforward behavior. Good job adding the dropout layers to avoid overfitting and the pooling layers to detect complex features!

If you are interested in making this network even better, you might want to try out Transfer Learning to extract better features. You can find the PyTorch tutorial on how to do so here.

Notebook 2: Define the Network Architecture

Define a `data_transform` and apply it whenever you instantiate a DataLoader. The composed transform should include: rescaling/cropping, normalization, and turning input images into torch Tensors. The transform should turn any input image into a normalized, square, grayscale image and then a Tensor for your model to take it as input.

Nice job defining the `data_transform` to turn an input image into a normalized, square, grayscale image in Tensor format. You have also nicely added *RandomCrop* operation to perform Data Augmentation. Well done! 😊 👍

You might want to consider further augmenting the training data by randomly rotating and/or flipping the images in the dataset. You can read the official documentation on torchvision.transforms to learn about the available transforms.

Select a loss function and optimizer for training the model. The loss and optimization functions should be appropriate for keypoint detection, which is a regression problem.

Appropriate loss function and optimizer have been selected. Well done!

`Adam` is a great (also a safe) choice for an optimizer. `MSELoss` is a decent choice too. You may want to read about `SmoothL1Loss` as well, a very reliable alternative to `MSELoss`. Checkout its PyTorch implementation. Please go through these two brilliant blog posts - One and Two to further improve your understanding of various types of loss functions available out there for us to use.

Train your CNN after defining its loss and optimization functions. You are encouraged, but not required, to visualize the loss over time/epochs by printing it out occasionally and/or plotting the loss over time. Save your best trained model.

The model has trained well for over 175 epochs. The code is well written and nicely commented. Keep it up!

After training, all 3 questions about model architecture, choice of loss function, and choice of batch_size and epoch parameters are answered.

The questions have been answered and your reasoning is clear and sound.

Thank you for writing such descriptive answers to all the questions. This really helps us, reviewers, to understand your thought process behind the decisions you have made while working on the project, which is useful in providing appropriate feedback. It is clear to me that you took well-informed decisions in coming up with the current network architecture and also in selecting the hyperparameters, it is also clear that you are comfortable with the overall pipeline of the project.

Your CNN "learns" (updates the weights in its convolutional layers) to recognize features and this criteria requires that you extract at least one convolutional filter from your trained model, apply it to an image, and see what effect this filter has on an image.

The CNN does "learn" to recognize the features in the image and a convolutional filter was extracted from the trained model for analysis.

After visualizing a feature map, answer: what do you think it detects? This answer should be informed by how a filtered image (from the criteria above) looks.

You are absolutely right about the filter you picked. Most filters are very complex as they often do *mixed* jobs but you are on point with the explanation. The filter indeed detects edges. Good analysis. 😊

Notebook 3: Facial Keypoint Detection

Use a Haar cascade face detector to detect faces in a given image.

Great job using the Haar cascade face detector for detecting frontal faces in the image!

You should transform any face into a normalized, square, grayscale image and then a Tensor for your model to take in as input (similar to what the `data_transform` did in Notebook 2).

Well done transforming the face image into a normalized, grayscale image and passing it into the model as a Tensor!

After face detection with a Haar cascade and face pre-processing, apply your trained model to each detected face, and display the predicted keypoints for each face in the image.

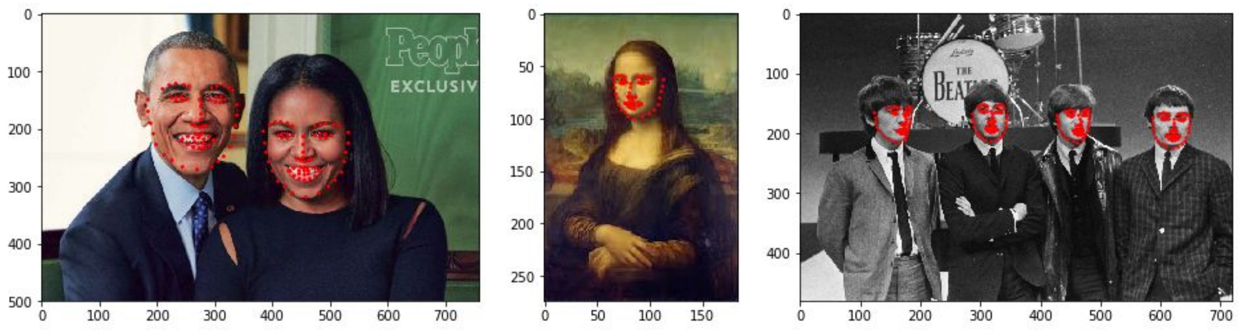
The model has been applied and the predicted key-points are being displayed on each face in the image. Your model predictions look acceptable. Very well done!

For a decent model like this one trained for over 175 epochs is expected to give better alignments of the keypoints. However, the misalignment here may not be due to the model itself. Looking at the examples in Notebook 2, I bet you would agree with me that the faces in the dataset are not as zoomed in as the ones Haar Cascade detects. Which is why you can grab more area around the detected faces to make sure the entire head (the curvature) is present in the input image, you can do that with the following changes:

```
margin = int(w*0.2)
roi = image_copy[y-margin:y+h+margin, x-margin:x+w+margin]
```

These changes would most likely give you better keypoints. 😊

Although NOT a rubric of the project, you can take up the task of mapping the points on to the original image (instead of plotting the points on separate faces) offline. It is a simple yet a mind-tingling programming exercise, give it a go.



Note: The model used for the predictions above was a 5-layer CNN network (with BatchNorm) followed by 3 FC layers, trained for 300 epochs. So it is alright if your model's predictions don't align as precisely.