UDACITY

‹ Return to Classroom

DISCUSS ON STUDENT HUB

Facial Keypoint Detection

REVIEW

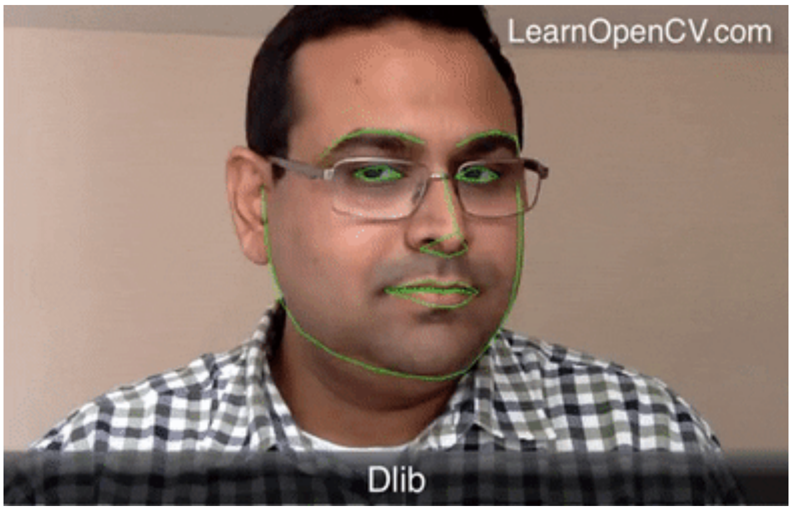
CODE REVIEW 4

HISTORY

Meets Specifications

You've done a fantastic job completing the Facial Keypoints Detection Project. The final predictions look great. I appreciate your sincere effort in experimenting and making informed decisions (instead of simply following an online resource or a research paper) regarding different loss functions, optimizers, architectures, etc. This is a very desirable trait since you'd have to do the same while working on real-world or new projects. You have carefully saved the output of some experiments to mention them in your answers, something I have seen other students rarely do, thank you very much for that. Lastly, I cannot emphasize enough on how brilliant you were with the data augmentation functions. Overall, a fantastic submission. 😊

Facial Keypoints Detection is a well-known machine learning challenge. If you want to improve this very model to allow it to work well in extreme conditions like bad lighting, bad head orientation (add appropriate PyTorch transformations like `ColorJitter`, `HorizontalFlip`, and more - see this post for more details), etc., the best thing you can do is to simply follow NaimishNet implementation details with some tweaks (optimizer, learning rate, batch size, etc) as per the latest improvements and your machine requirements. But for production-level performance, you can always use pre-trained models for better performance, say Dlib library provides real-time facial landmarks seamlessly. You can find a tutorial here. Here is an example:



Note: Predicted keypoints are joined with lines here.

Here are some advanced research works involving facial landmarks:

- Style Aggregated Network for Facial Landmark Detection (Code)
- Supervision-by-Registration: An Unsupervised Approach to Improve the Precision of Facial Landmark Detectors (Code)
- Teacher Supervises Students How to Learn From Partially Labeled Images for Facial Landmark Detection (Code)
- A Fast Keypoint Based Hybrid Method for Copy Move Forgery Detection
- Disguised Face Identification (DFI) with Facial KeyPoints using Spatial Fusion Convolutional Network
- Berkeley team's attempt at beating the Facial Keypoints Detection Kaggle competition
- Facial Keypoints Detection using the Inception model

Facial keypoint prediction pipeline can be extended to human poses, hand poses and more to help intelliget systems like robots, automatic anomaly detectors understand the orientation of subjects in CCTV footage, etc. Here are some works based on keypoint prediction:

- OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields (Code)
- PoseFix: Model-agnostic General Human Pose Refinement Network (Code)
- SRN: Stacked Regression Network for Real-time 3D Hand Pose Estimation (Code)
- Hand Pose Estimation: A Survey

Keep up the good work and good luck with the rest of the nanodegree! 😊👍

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. You also added BatchNorm to every layer, something very few students care to do, great job! 😊

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.

Depending on the complexity of the network you define, and other hyperparameters the model can take some time to train. We encourage you to start with a simple network with only 2 layers. You'll be graded based on the implementation of your models rather than accuracy.

Nice job defining the `data_transform` to turn an input image into a normalized, square, grayscale image in Tensor format. I am amazed by your efforts to write extra augmentation functions from scratch, a very original thing I have seen a student do in a long time for this project. It is one thing to use PyTorch's torchvision.transforms and a whole another thing to write it from scratch, it must have taken so much a decent amount of time in testing the code. I am actually saving your `data_load.py` to encourage future students to do the same. Well done! 😉👍

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. I am very glad to see that you tried `SGD` before settling for `Adam` and `L1Loss` before settling for `MSELoss`, an informed decision takes precedence over random guess. You may want to read about `SmoothL1Loss` as well, a very reliable alternative to `MSELoss`. It combines the advantages of both L1-loss (steady gradients for large values of x) and L2-loss (fewer oscillations during updates when x is small). Checkout its PyTorch implementation. Thank you for saving the training outputs of those experiments for me to see. 😊

If interested, you can go through these three brilliant articles - One, Two and Three 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. The code is well written with appropriate comments. Keep it up! 😊

Very nice to see you visualize the loss trajectory of all your experiments, something only a handful of students care to do. And its always nice to see your loss value go down, graphically. 😉

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, understand your thought process behind the decisions you have made while working on the project, which is useful in providing appropriate feedback. It is so obviously clear to me that you made 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.

Additional comments for real-time usage of these trained models:
It is nice to see that you followed the suggested paper - NaimishNet is an excellent place to start. Most of the students often do the same and it is probably the right thing to do when you are dealing with a new problem provided you are not sure where to begin. But what is also important is to understand the capabilities/limits of smaller networks (Please know that these comments are only relevant to the usage of these models on an edge device, there is absolutely nothing wrong with starting your experiments with a very deep network for this project) You should always be able to answer (to yourself at the very least) some questions like:

- Why 5 CNN layers? (NaimishNet)
- Why not just 1 CNN layer?
- Can this problem be solved with less than 3 CNN layers?
- Can a single FC layer do the job? (Fully connected layers have the majority of model parameters)
- If yes, how far can we go with the layer pruning?.

Experimenting with different networks can be extremely boring but the insights can really help us when we're dealing with real-time memory/computational constraints (often faced when deploying trained models on edge devices). There was a student who got near-perfect predictions with just 1 CNN layer and 1 FC layer and of course no pooling. Anyways, you did the right thing no doubt but try and make sure you can answer these questions to yourself whenever you are on to new problems i.e., try other architectures, optimizers, learning rates to test the strength of smaller networks. Good job! 😊

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 model 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.

Most filters like this one are very complex as they often do *mixed* jobs and are not straightforward to interpret at all. The filters are indeed detecting edges. Good observation and analysis. 😊

To be completely honest, we cannot ever be absolutely sure what a filter is exactly doing, especially about the ones a neural network learns. The motivation behind this rubric is to simply make you see and visualize one of the filters (a.k.a kernels) from the first layer of the model and interpret its role in the model based on our understanding of convolution filters. This type of interpretation is naïve but a good starting point towards understanding what neural nets are learning. I use the word "naïve" since you cannot possibly explain filters on the second layer because the input to the second layer is no more the image we know. However, there are more advanced ways of understanding what is going on inside NNs - check out these resources - one, two, three, four, five, if you are interested.

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 through 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! 😊

I appreciate your idea to add padding to the faces. It is a very effective trick. Looking at the training images 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. This is why you MUST grab more area around the detected faces to make sure the entire head (the curvature) is present in the input image. You can do the same in a slightly more generic way, without having to use a constant padding value with the following changes:

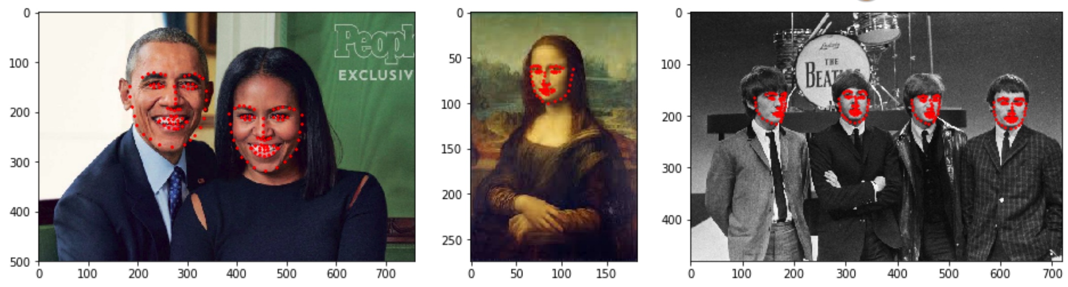
```
margin = int(w*0.3)
roi = image_copy[y-margin:y+h+margin, x-margin:x+w+margin]
```

or

```
margin = int(w*0.3)
```

```
roi = image_copy[max(y-margin,0):min(y+h+margin,image.shape[0]),
                 max(x-margin,0):min(x+w+margin,image.shape[1])]
```

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) after you are done with this project. 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.

DOWNLOAD PROJECT

4 CODE REVIEW COMMENTS ❯

RETURN TO PATH

Rate this review

START