UDACITY

‹ Return to "Computer Vision Nanodegree" in the classroom

DISCUSS ON STUDENT HUB

Facial Keypoint Detection

REVIEW

CODE REVIEW 1

HISTORY

Meets Specifications

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

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`, `Rotation` 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, 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.

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. 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`. It combines the advantages of both L1-loss (steady gradients for large values of x) and L2-loss (less oscillations during updates when x is small). Checkout its PyTorch implementation.

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! 😊

It is never a bad idea to see and compare training and validation losses. You can visualize them with this piece of code:

```
loss_data = np.asarray(loss_data)
plt.plot(loss_data[:,0], label='Training Loss')
plt.plot(loss_data[:,1], label='Validation Loss')
ax = plt.gca()
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.spines['left'].set_alpha(0.7)
ax.spines['bottom'].set_alpha(0.7)
plt.title("Training and Validation Loss")
plt.legend()
plt.show()
```

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. However, you answer regarding the loss function is inconsistent with the code. I am assuming you tried different things and forgot to update. Overall, 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.

Most students often blindly implement the NaimishNet architecture 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. You should always be able to answer (to yourself at the very least) some questions like "Why 5 CNN layers? Why not just 1 CNN layer?", "Can this problem be solved with less than 5 CNN layers?", "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 by starting your experiments with smaller network architectures. 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.

You have not executed the cells related to this rubric. 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 (and first layer only and not fourth layer, explained why below) of the model and interpret its role in the model based on our understanding of convolution filters. It appears to me there was a typo in the code and the output was lost during notebook kernel restart. It is alright for now, since you did great in all other rubrics this could be overlooked. But please make sure these kinds of inconsistencies do not exist in your future submissions.

To be completely honest, we cannot ever be absolutely sure what a filter is exactly doing, especially about the ones a neural network learns. Comparing convoluted images with their associated input images has long been a method of understanding kernels but this type of interpretation is naïve but also 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. The input to 4th layer is not the original input image but a complex feature map output of 3rd layer which we cannot possibly understand and relate back to the original image so the comparison you opted for is not ideal. 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 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! 😊

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

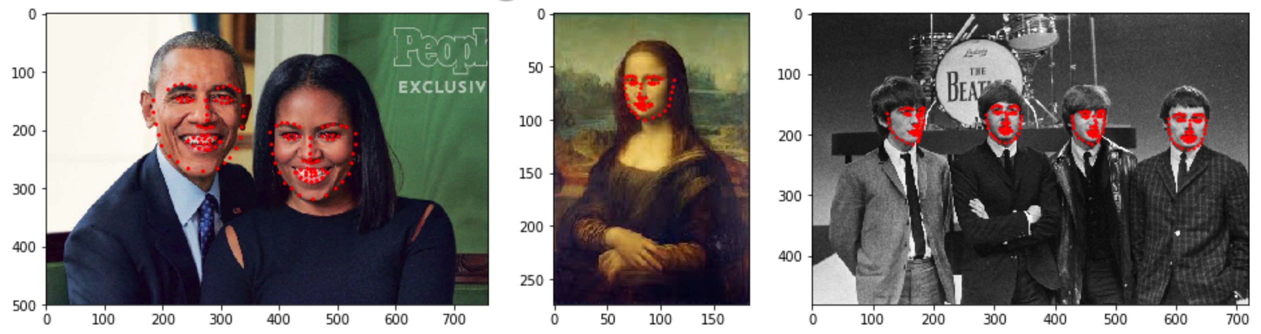
Cascade detects. This is why you MUST grab more area around the detected faces to make sure the entire head (the curvature) is present in the input image. You can do the same in a more elegant way with the following changes:

```
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

1 CODE REVIEW COMMENTS ›

RETURN TO PATH