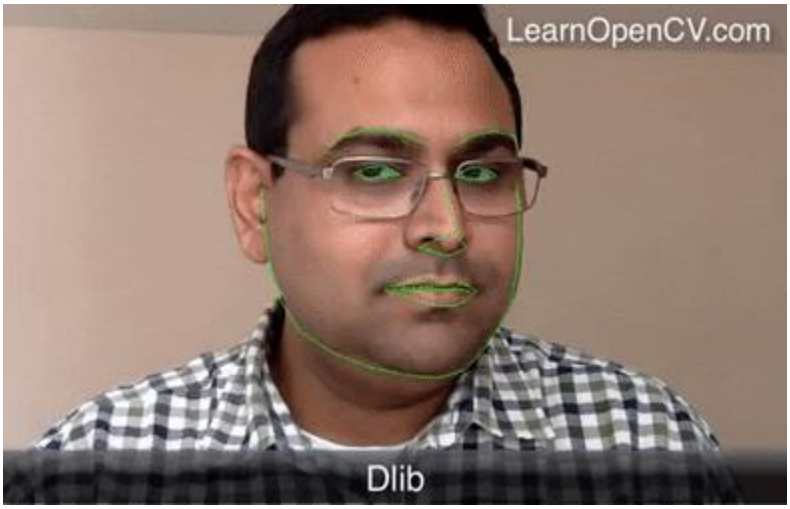
Meets Specifications

You've done a fantastic job completing the Facial Keypoints Detection project. The final predictions look great. I appreciate your effort in experimenting and making informed decisions with the help of online resources and research papers. This is a very important trait since you'd have to do the same while working on real-world or new projects.

In your next projects, I encourage you to be more verbose with your answers. You simply have to keep track and let us know what experiments you performed, what your thought process was and what made you doing certain things (usually a direct consequence of your thoughts and ideas). This is all very useful to us in providing appropriate feedback. Please remember that it's sometimes okay to not try several combinations of things as you might be getting decent results or you might be lacking proper computational resources but please mention that as well (whatever may be your case). Looking forward to more verbose answers in your future projects.

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, all visualization cells have been executed 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!

As a step forward from here, if you are interested in making this network even better and more generalizable, you might want to try out Transfer Learning to extract better features from a well-trained model and then tune the model on this dataset. What you'd basically do is use a network like `ResNet18` and `ResNet50` and initialize the model with ImageNet pretrained weights. And then you'd go on about training your model either updating only the fully connected layers (since the ImageNet pretrained weights are great feature extractors) or the entire model. This would save us a lot of training time since the model need not learn features from scratch. You can find the PyTorch tutorial on how to do so here.

You also added BatchNorm to every layer, something very few students care to do, great job! Often, the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization and makes it notoriously hard to train models with saturating nonlinearities. Come to the rescue, Batch Normalization

standardizes the inputs to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks. Now, post introduction of Batch Normalization, several works have proposed other variants such as the ones shown below, perhaps you should try them out later:

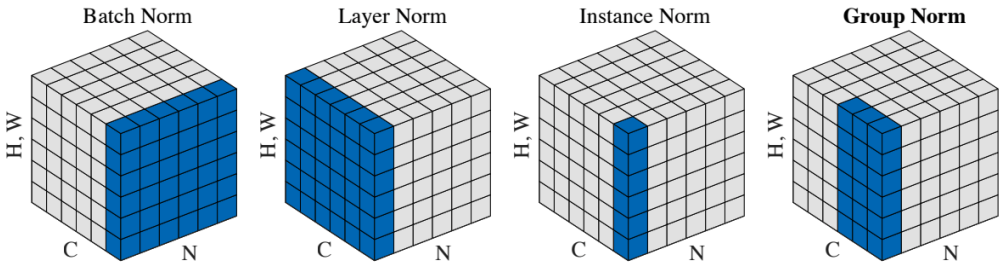


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

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.

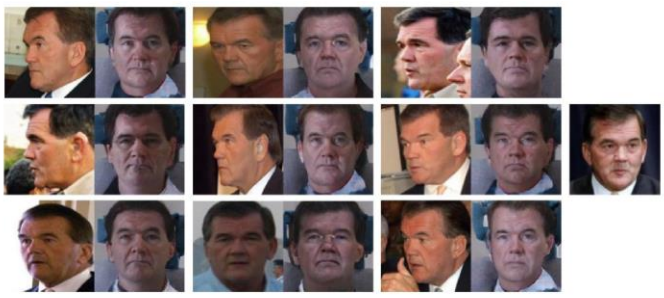
Depending on the complexity of the network you define, and other hyperparameters the model can take some time to train. We encourage you to start with a simple network with only 2 layers. You'll be graded based on the implementation of your models rather than accuracy.
Nice job using the provided transforms inside `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 and/or shearing the images in the dataset. You can read the official documentation on [torchvision.transforms] to learn about the available transforms.

Broadly speaking, there are two ways of augmentations possible with a given face dataset:

- One-to-many Augmentations
- Many-to-one Augmentations

One-to-many Augmentation



Many-to-one Normalization

While many-to-one augmentations are great for a network to look at and are conducive for generalized learning, it often is the case that datasets at hand do not particularly include multiple viewpoints/perspectives of the same face. Such datasets are expensive to obtain and maintain as well. This is why performing one-to-many augmentation is perhaps the best alternative as it contributes a lot in making reasonably better inferences in real-time.

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. `SmoothL1Loss` is a very reliable alternative to `MSELoss`, as it tends to be more robust to outliers in the data while training. 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). Good choice!

L2 - MSE, Mean Square Error

$$L_2(x) = x^2$$

$$f(y, \hat{y}) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Smooth L1

$$\text{smooth } \mathrm{L}_1(x) = \begin{cases} 0.5x^2 & if|x| < 1 \\ |x| - 0.5 & otherwise \end{cases}$$

$$f(y, \hat{y}) = \begin{cases} 0.5(y - \hat{y})^2 & \text{if } |y - \hat{y}| < 1 \\ |y - \hat{y}| - 0.5 & otherwise \end{cases}$$

If interested, you can go through these three brilliant articles to further improve your understanding of various types of loss functions available out there for us to use:

- A Detailed Guide to 7 Loss Functions with Python Code
- Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names
- Understanding Ranking Loss, Contrastive Loss, Margin Loss, Triplet Loss, Hinge Loss and all those confusing names
  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. To that end, you could use the `test_loader` in the notebook as a validation set during the training. 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()

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. It is 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, provided you have the computational resources at hand (GPUs). You should always be able to answer (to yourself at the very least) some questions like:

- Why 5 CNN layers (NaimishNet)? Why not more or why not less?
- Why not just 1 CNN layer? Remember that fewer CNN layers don't mean fewer parameters since the fully connected layers have most of the model parameters and CNNs typically reduce the dimensions of the input which are fed to the fully connected layers. Pause and ponder if you are not convinced with this.
- Can a single FC layer do the job? Once again, fully connected layers have the majority of model parameters so knowing this would help us.
- If yes, how far can we go with the layer pruning or layer adding?

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). To that end, there was a student who got near-perfect predictions with just 1 CNN layer and 1 FC layer and no pooling, but this is not recommended too since the FC layer will have a lot of parameters compared to a simple 3CNN (this reduces the parameters) + 2FC network. 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 shallow and deeper 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.

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 emphasizes edges (not specifically vertical ones if you ask me). But overall, good observation and analysis.

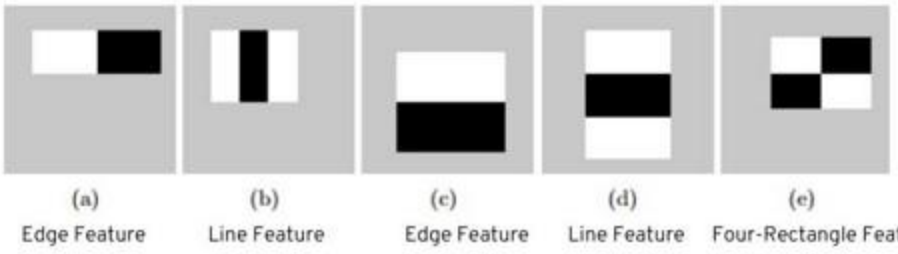

To be technically honest, we cannot ever be absolutely sure what a filter is exactly doing, especially about the ones a neural network learns. The motivation behind this rubric is to simply make you see and visualize one of the filters (a.k.a kernels) from the first layer of the model and interpret its 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. Anyways, there are more advanced ways of understanding what is going on inside NNs - check out these resources if you are interested.:

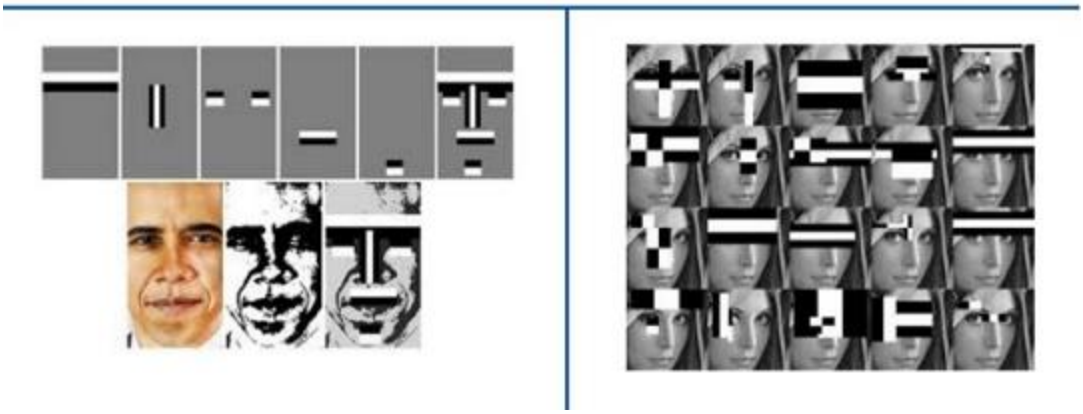
- The Building Blocks of Interpretability
- An Overview of Early Vision in InceptionV1
- Zoom In: An Introduction to Circuits
- Differentiable Image Parameterizations

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!

Object Detection is hard in general. In the early 2000s, Haar feature-based cascade classifiers were proposed by Paul Viola and Michael Jones and to this day they are the most widely used face detectors. Haar classifiers are fast, accurate, and simple and these features makes building applications that run in realtime possible. You could read this article from OpenCV or this great article to know more about how they use simple edge and line features to get the job done.





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 borderline acceptable. Very well done!

There is a very effective trick that could improve the alignment of the keypoints. 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 padding in a generic way, without having to use a constant padding value, with the following code:

```
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])]
```

I can see that you tried to manually set the SD and Mean values in the denormalization steps. Please be aware, a "good" model doesn't enforce us to manually tune these values. It is alright for now but in real-time, an application with this code would probably not yield intended/desired predictions. One way to mitigate this issue is to keep your model complex (which gives it the capability to learn complex facial feature patterns and face orientations, etc.) and train it for a large number of epochs.

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