Facial Keypoint Detection

REVIEW | CODE REVIEW | HISTORY

Meets Specifications

Dear Student,
Congratulations ! You did a great job.

You reached all the goal of the lesson: Manipulating tensors, convnet, dropout, find the loss function and test optimizers.
if you want to play on the next level : try transfert learning from pretrained network like vgg, resnet ...etc.
https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html. The goal here is to used already working architecture and add yours for a specific task.

Some suggestions and reference to improve the network for future :

This is a very rough good guide on improving the network (though you have most of that in your net except for data augmentation) - https://machinelearningmastery.com/improve-deep-learning-performance/
Here's one reference paper from one of the Pioneers in deep learning - Yoshua Bengio
Practical Recommendations for Gradient-Based Training of Deep Architectures - https://arxiv.org/pdf/1206.5533v2.pdf
I know it's too theoretical and long but worth reading for in general good practices.

Also If this review helped you in some way, Please consider rating it positively.
I hope you enjoyed the nd so far and wish you the best. Great going !

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 required files are submitted 👍

`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.**

Good job defining an architecture. It's always tricky to calculate the correct dimension for the first linear layer and you did it correctly.
You used all the base tools for deep learning. It will to help to make the learning faster and better generalize.
If you want to dig deeper here are some guidelines: http://www.deeplearningbook.org/contents/guidelines.html

A suggestion for better practice is to reuse the pooling because these layers use the same attributes. Each of these layer is not a specific instance and thus doesn't requires to create a new object for every layer. So you could define one pooling layer in init once and call it multiple times as required in the forward pass. Not required for this case but Something to remember in future when designing a ConvNet.

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.**

Great job defining the data_transforms to turn an input image into a normalized, square, grayscale image in Tensor format !

A suggestion for future improvemnts is to consider augmenting the training data by randomly rotating and/or cropping the dataset. You can read the official documentation on torchvision.transforms to learn about the available transformations here - https://pytorch.org/docs/master/torchvision/transforms.html

✓ **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.**

✓ **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.**

Nice. A suggestion, though, consider visualizing the network's loss in a graph – this is extremely helpful for the non-technical among us, whom you'll likely have to present findings and progress to at some point in your career (and even the technical who are just unfamiliar with the domain(s) you're dealing with). 😃 Graphs are excellent, and concise, ways to convey information like loss of the network as people, even the non-mathematically-inclined, tend to be good at identifying trends.

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

Please find my comments below for different answers -

For Answer 1 :
Nice work choosing Adam, just to elaborate the use of Adam optimizer is that it combines the benefits of both Adagrad and RMSProp algorithms. In short, it works well with sparse gradients and also deals with the problem where training hits a local optimum (saddle point) but doesn't make any progress like in the case of Stochastic Gradient Descent;

You correctly used the loss function required for regression problem i.e. - MSE Loss. It's Great that you tried SmoothL1 Loss well. Refer to this answer for where choosing different losses can have different effect - https://stats.stackexchange.com/a/48268

For Answer 2 :
Sticking to a smaller architecture would make more sense here as the problem is not to complex. Exact NamishNet or VGG is an overkill for this problem and as expected would result in overfitting. Overall Nice job for all the work on this one.
In future before training the net for whole data do the sanity checks first and rectify the problems you might have. Refer to this excellent guide by Andrej Karapathy from Stanford for this - http://cs231n.github.io/neural-networks-3/#sanitycheck

For Answer 3 :
Great way to choose batch size to utilise the most potential of GPU. It has been observed in practice that when using a larger batch there is a degradation in the quality of the model, as measured by its ability to generalize. Could try a smaller batch_size for better accuracy. Refer here for more details - https://arxiv.org/abs/1609.04836
Also In practice using batch_size in powers of 2 helps in more effective loss computation.

✓ **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.**

Excellent job displaying the filter. 😃 Just bear in mind, you aren't limited to displaying a single kernel – it's almost always best to show more of the internals of the network than to show less. If anything, it may also contribute to building an intuition about what tends to happen in the beginnings of convolutional nets, among others.

✓ **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.**

I concede that it is far from obvious to say exactly what filters do since filters are always not pure.
I see the filter you choose seems to do what you say. And you did a good job answering that.
Here some materials to read if you want :
https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/
Another fun one :
What is convolution ?: https://mathoverflow.net/questions/5892/what-is-convolution-intuitively/5916#5916

Notebook 3: Facial Keypoint Detection

✓ **Use a Haar cascade face detector to detect faces in a given 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).**

Nice work on applying correct data transforms.
This step required multiple things and you did all well.

✓ **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 accuracy is quite good for the predicted points.

DOWNLOAD PROJECT

RETURN TO PATH