MACHINE LEARNING NANODEGREE

FACIAL KEYPOINTS DETECTION

CAPSTONE PROJECT

1. DEFINITION

Project Overview

The Capstone Project that I have chosen is the Facial Keypoints Detection. I chose the medical field , since I am deeply interested in using Machine Learning in Medicine due to the vast amount of data, research available and so many results to be found. I had in my mind to devote my time and skills in medicine but was confused about the process of getting data. Hence I turned my mind to kaggle and I thought what better way to even get my results judged compared to others other than a past competition. So after searching for a bit this Competition got my attention. We know that there is technology that can judge our results like advanced machines , but we could use Machine Learning to speed up the process and compare our results to those advanced machine and try to make our model work better and better for such applications.

Problem Statement

The objective of this task is to predict keypoint positions on face images. This can be used as a building block in several applications, such as:

- tracking faces in images and video
- analysing facial expressions
- detecting dysmorphic facial signs for medical diagnosis
- biometrics / face recognition

Detecting facial keypoints is a very challenging problem. By keypoints I mean that for a person the position of important and common features such as eyes , nose , upper lip , lower lip etc. Facial features vary greatly from one individual to another, and even for a

single individual, there is a large amount of variation due to 3D pose, size, position, viewing angle, and illumination conditions. Computer vision research has come a long way in addressing these difficulties, but there remain many opportunities for improvement.

The intended solution is Data Preprocessing & then training the model on Neural Nets , since they have been the best way to achieve greater success in such problems.

Metrics

Root Mean Squared Error (RMSE) The evaluation metric I chose is the root mean squared error. RMSE is very common and is a suitable general-purpose error metric.

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2},$$

where y hat is the predicted value and y is the original value.

Compared to the Mean Absolute Error, RMSE punishes large errors Moreover submission made to the competition , i.e. the kaggle platform will also help in evaluation of the model compared to others. One thing that might help is training the initial model on the some simpler algorithms and then later on comparing it with our Neural Net to see whether our model is the better solution or not.

2. ANALYSIS

Data Exploration

The data set was easily obtained from the kaggle competition. In the following lines I will be citing the information as obtained from kaggle.

The data set for this competition was graciously provided by Dr. Yoshua Bengio of the University of Montreal.

Each predicted keypoint is specified by an (x,y) real-valued pair in the space of pixel indices. There are 15 key points, which represent the following elements of the face: left_eye_center, right_eye_center, left_eye_inner_corner, left_eye_outer_corner, right_eye_inner_corner, right_eye_outer_corner, left_eyebrow_inner_end, left_eyebrow_outer_end, right_eyebrow_inner_end, right_eyebrow_outer_end, nose_tip, mouth_left_corner, mouth_right_corner, mouth_center_top_lip, mouth_center_bottom_lip Left and right here refers to the point of view of the subject. In some examples, some of the target keypoint positions are missing (encoded as missing entries in the csv, i.e., with nothing between two commas). The input image is given in the last field of the data files, and consists of a list of pixels (ordered by row), as integers in (0,255). The images are 96x96 pixels.

Data files

● training.csv: list of training 7049 images. Each row contains the (x,y) coordinates for 15 keypoints, and image data as row-ordered list of pixels.

● test.csv: list of 1783 test images. Each row contains ImageId and image data as row-ordered list of pixels

● submissionFileFormat.csv: list of 27124 keypoints to predict. Each row contains a RowId, ImageId, FeatureName, Location. FeatureName are "left_eye_center_x," "right_eyebrow_outer_end_y," etc. Location is what you need to predict.

The following is one of the sample of the data.Each of the features represent the location of the keypoint and the Image feature helps in defining the whole Image.

```
   left_eye_center_x  left_eye_center_y  right_eye_center_x  \
0          66.033564          39.002274           30.227008

   right_eye_center_y  left_eye_inner_corner_x  left_eye_inner_corner_y  \
0           36.421678                59.582075                39.647423

   left_eye_outer_corner_x  left_eye_outer_corner_y  right_eye_inner_corner_x  \
0                73.130346                39.969997                 36.356571

   right_eye_inner_corner_y  \
0                 37.389402

                         ...                         nose_tip_y  \
0                        ...                          57.066803

   mouth_left_corner_x  mouth_left_corner_y  mouth_right_corner_x  \
0            61.195308            79.970165             28.614496

   mouth_right_corner_y  mouth_center_top_lip_x  mouth_center_top_lip_y  \
0             77.388992               43.312602               72.935459

   mouth_center_bottom_lip_x  mouth_center_bottom_lip_y  \
0                  43.130707                  84.485774

                                               Image
0  238 236 237 238 240 240 239 241 241 243 240 23...
```

Now let's take a slice of the Describe function called on the training data.

	left_eye_center_x	left_eye_center_y	right_eye_center_x	\
count	7039.000000	7039.000000	7036.000000	
mean	66.359021	37.651234	30.306102	
std	3.448233	3.152926	3.083230	
min	22.763345	1.616512	0.686592	
25%	NaN	NaN	NaN	
50%	NaN	NaN	NaN	
75%	NaN	NaN	NaN	
max	94.689280	80.502649	85.039381	

	right_eye_center_y	left_eye_inner_corner_x	left_eye_inner_corner_y	\
count	7036.000000	2271.000000	2271.000000	
mean	37.976943	59.159339	37.944752	
std	3.033621	2.690354	2.307332	
min	4.091264	19.064954	27.190098	
25%	NaN	NaN	NaN	
50%	NaN	NaN	NaN	
75%	NaN	NaN	NaN	
max	81.270911	84.440991	66.562559	

	left_eye_outer_corner_x	left_eye_outer_corner_y	\
count	2267.000000	2267.000000	
mean	73.330478	37.707008	
std	3.405852	2.881438	
min	27.571879	26.250023	
25%	NaN	NaN	
50%	NaN	NaN	
75%	NaN	NaN	
max	95.258090	64.618230	

	right_eye_inner_corner_x	right_eye_inner_corner_y	\
count	2268.000000	2268.000000	
mean	36.652607	37.989902	
std	2.350268	2.311907	
min	5.751046	26.250023	
25%	NaN	NaN	
50%	NaN	NaN	
75%	NaN	NaN	
max	70.714966	69.808803	

Here one thing to observer is that there are several features that have NULL values , hence we need to do some type of Data Preprocessing. One more thing to notice is that Image feature has values separated by space , which we will want to change later on.

Exploratory Visualization

Although the images have already been attached above , however we can start exploring a bit more. The following are some the visualizations about the data.

- Each facial keypoint has different number of available labels , hence we won't be having much data initially to train with.
- The Image feature has space separated features , it will be better to convert it to Numpy Arrays.
- The pixels should be better of normalized/standardized since then they will be better for a model to train on them.

The following image depicts the count of available labels for each keypoint.

```
left_eye_center_x            7039
left_eye_center_y            7039
right_eye_center_x           7036
right_eye_center_y           7036
left_eye_inner_corner_x      2271
left_eye_inner_corner_y      2271
left_eye_outer_corner_x      2267
left_eye_outer_corner_y      2267
right_eye_inner_corner_x     2268
right_eye_inner_corner_y     2268
right_eye_outer_corner_x     2268
right_eye_outer_corner_y     2268
left_eyebrow_inner_end_x     2270
left_eyebrow_inner_end_y     2270
left_eyebrow_outer_end_x     2225
left_eyebrow_outer_end_y     2225
right_eyebrow_inner_end_x    2270
right_eyebrow_inner_end_y    2270
right_eyebrow_outer_end_x    2236
right_eyebrow_outer_end_y    2236
nose_tip_x                   7049
nose_tip_y                   7049
mouth_left_corner_x          2269
mouth_left_corner_y          2269
mouth_right_corner_x         2270
mouth_right_corner_y         2270
mouth_center_top_lip_x       2275
mouth_center_top_lip_y       2275
mouth_center_bottom_lip_x    7016
mouth_center_bottom_lip_y    7016
Image                        7049
dtype: int64
```

Algorithms & Techniques

There are various approaches to the problem. I will discuss everyone of them in a little bit detail. The data processing algorithms in this problem that are applied are :-

- Gaussian blurring : Blurring of image by a Gaussian function
- Histogram Stretching : Adjusting the Image's contrast
- PCA : Identifying the Principal Components(In this Case didn't help much)
- Inversion of Images to generate more datasets to train the model on

Since we have already removed most of the data due to NaN values we need some more data to train the model better.The above algorithms generate a better data to help identify the optimal policy of the model. Inversion will help in generating more datasets since we know we are already short on the datasets due to NaN values. Except PCA which will help us identify if there are some other components on which if data is scaled it might be better to train on it.

Since it's a task requiring high precision and accuracy hence models like RF , Decision Trees , knn would be a very bad model for such a problem. It has been observed and seen that Neural Nets prove to be very good for such kind of problems. A simple Neural Net will be good but not the best. To improve more and more we can move to Theano & Lasagne or even Tensor Flow and train CNN with more layers. These libraries help in implementing better neural networks and makes it easier to add layers and define them. Since there are lot of parameters with which we can optimize the Neural Net these libraries provide easier way to optimize them. Even further Hyperparameter Optimization is easier.

The following are the parameters of the NN while using Lasagne & Theano:

- Input shape - defining the shape of input
- Hidden_num_units - number of units in hidden layer
- Output_num_units - number of units in hidden layer
- output/hidden_nonlinearity- we can define the hidden/output layers to use non_linear or linear functions for their activation

- Learning _ rate : this decides the precision and rate with which the NN converges to the optimal policy and update its weights
- The libraries further help in initializing weights using Glorot-style initialization , which it figures out on its own depending on the data and parameters , thus saving us time and making it easier
- Update : defines the function which is used to update weights
- Update_momentum

LeNet5-style convolutional neural nets are at the heart of deep learning's recent breakthrough in computer vision. Convolutional layers are different to fully connected layers; they use a few tricks to reduce the number of parameters that need to be learned, while retaining high expressiveness. These are:

1. *local connectivity*: neurons are connected only to a subset of neurons in the previous layer,
2. *weight sharing*: weights are shared between a subset of neurons in the convolutional layer (these neurons form what's called a *feature map*),
3. *pooling*: static subsampling of inputs.

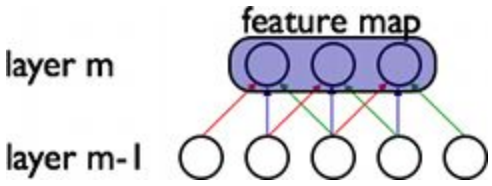


Illustration of local connectivity and weight sharing.
(Taken from the deeplearning.net tutorial.)

Units in a convolutional layer actually connect to a 2-d patch of neurons in the previous layer, a prior that lets them exploit the 2-d structure in the input.

The following is a short algorithm followed in the project:

- Data Preprocessing
- Training Data on a simple NN

- Convolutional Neural Network with data augmentation , learning rate decay and dropout
- A pipeline of specialist CNNs with early stopping and supervised pre-training
- Removing Overfitting

Benchmark

The model is best evaluated by using the RMSE and the kaggle leaderboard. Other models cannot be used to evaluate the current model/algorithm since NN is the only best way out and it can be further improved by tuning the parameters , adding layers etc.

Hence the final RMSE value for my model should be between 1.28236 and 3.65092.

3. METHODOLOGY

Data Preprocessing

The following methods have been used for data preprocessing since they provided better outcomes:

- Dropping of rows that have Null values in them , thereby reducing size of dataset that is to be used for training
- Making the feature 'Image' values into numpy arrays and standardizing/normalizing them by dividing each value by 255(because of pixels). Normalization is needed in such case so that the model can compare each feature and update it's weights accordingly. No single feature should shadow other features and not cause weights of model to change largely. It's more a kind of regularization.
- Shuffling and splitting the data each time to make sure the model fits and trains better.

- Using the batch method we divide data into variable size batches consisting of 96 x 96 pixels since the update function of a NN is called after every batch.

```
X.shape == (2140, 9216); X.min == 0.000; X.max == 1.000
y.shape == (2140, 30); y.min == -0.920; y.max == 0.996
```

- Modifying shape of data depending on the model to be used for , like for Lasagne no flat vectors can be used hence we use 3D shape in the form (a,9,1) where a is the number of colors in the image , 0 is x coordinate and 1 is the y coordinate. Hence in our case a = 1 since our images have only gray color channel only hence it will be (1,96,96)
- Inversion of Images with a 50% Probability to generate more data to prevent overfitting.

Implementation & Refinement

The model that has been used for solving the problem is Neural Net & later on moving to further complex NN's and using libraries like Lasagne & Theano to ease the process.

A short list of steps is provided :-

1. Data preprocessing
2. Simple NN
3. Data preprocessing for CNN
4. 2nd CNN
5. 3rd CNN with Data augmentation
6. 4th CNN without Data augmentation but with optimization of learning_rate & momentum
7. 5th CNN is same as 4th CNN but with Data Augmentation
8. Using Dropout to reduce overfitting with 5th CNN this is the 6th CNN
9. 7th CNN is just 6th CNN but with larger number of epochs
10. Now we just use the data ignored in step 1 to train specialists i.e features having more data now have individual CNN with structure same as 7th CNN. These are referred as specialists.

Initially we preprocess the data as stated above. Further we train the available data on a simple NN with a single hidden layer having learning_rate = 0.01 & momentum = 0.9. These two factors determine a lot about the time it takes a NN to train. This will make a lot of change since while training on large models , it can save us time and make hyper parameter optimization easier. In the simple NN they have been initialized but later on we can change their values after each batch/epoch and this will help in reducing training time and make the model approach the optimal policy faster. The first NN trained has only a single hidden layer for the sake of simplicity and to test the performance of NN's on the data. The update function will update the weights of our network after each batch. The chosen one is nesterov_monetum gradient descent optimization method to do the job, as it has been seen to work pretty well on most of the problems.

Due to the good performance of a simple NN there is a huge chance that the CNN will perform quite well.

LeNet5-style convolutional neural nets are at the heart of deep learning's recent breakthrough in computer vision. Convolutional layers are different to fully connected layers; they use a few tricks to reduce the number of parameters that need to be learned, while retaining high expressiveness. These are:

4. *local connectivity*: neurons are connected only to a subset of neurons in the previous layer,
5. *weight sharing*: weights are shared between a subset of neurons in the convolutional layer (these neurons form what's called a *feature map*),
6. *pooling*: static subsampling of inputs.

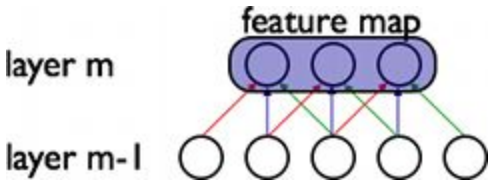


Illustration of local connectivity and weight sharing.
(Taken from the deeplearning.net tutorial.)

Units in a convolutional layer actually connect to a 2-d patch of neurons in the previous layer, a prior that lets them exploit the 2-d structure in the input.

To make the data fit for CNN to train on as discussed in Data Preprocessing , we need to make each point a kind of 3D vector.

The second convolutional neural net is built with three convolutional layers and two fully connected layers. Each CV layer is followed by a 2x2 max-pooling layer. The number of filters double every layer. The densely connected hidden layers both have 500 units.

```
InputLayer            (None, 1, 96, 96)       produces    9216 outputs
Conv2DCCLayer         (None, 32, 94, 94)      produces  282752 outputs
MaxPool2DCCLayer      (None, 32, 47, 47)      produces   70688 outputs
Conv2DCCLayer         (None, 64, 46, 46)      produces  135424 outputs
MaxPool2DCCLayer      (None, 64, 23, 23)      produces   33856 outputs
Conv2DCCLayer         (None, 128, 22, 22)     produces   61952 outputs
MaxPool2DCCLayer      (None, 128, 11, 11)     produces   15488 outputs
DenseLayer            (None, 500)             produces     500 outputs
DenseLayer            (None, 500)             produces     500 outputs
DenseLayer            (None, 30)              produces      30 outputs
```

Since NN's have been prone to overfitting hence this one is no exception. To remove overfitting best method is to provide more data. Data augmentation lets us artificially increase the number of training examples by applying transformations, adding noise etc.

I mentioned batch iterators already briefly. It is the batch iterator's job to take a matrix of samples, and split it up in batches, in this case of size 128. While it does the splitting, the batch iterator can also apply transformations to the data on the fly. So to produce those horizontal flips, no need to double the data in the input matrix. Rather it can just perform the horizontal flips with 50% chance while it's iterating over the data. This is convenient, and for some problems it allows to produce an infinite number of examples, without blowing up the memory usage. Also, transformations to the input images can be done while the GPU is busy processing a previous batch, so they come at virtually no cost.

Since images are being flipped, target values too have to be flipped. This is because the flipped left_eye_center_x no longer points to the left eye in the flipped image; now it corresponds to right_eye_center_x. Some points likenose_tip_y are not affected.

This was the 3rd CNN with Data Augmentation.

For the 4th CNN it's going to optimize the learning rate and the momentum of the update function. An intuition for the learning rate is that as it starts training, initially it's far away from optimum and it's better to take large steps and learn quickly. But the closer it gets

to the optimum, the lighter should it step for better accuracy.On the importance of initialization and momentum in deep learning is the title of a talk and a paper by Ilya Sutskever et al.

To get the basic idea that optimizing learning_rate & momentum helps , 4th CNN will be trained without Data Augmentation , and 5th CNN with Data Augmentation.

However our 5th CNN too does overfit because of which we now have to use Dropout.To use dropout with Lasagne, DropoutLayer layers are added between the existing layers and assign dropout probabilities to each one of them.

Name	Description	Epochs	Train loss	Valid loss
net1	single hidden	400	0.002244	0.003255
net2	convolutions	1000	0.001079	0.001566
net3	augmentation	3000	0.000678	0.001288
net4	mom + lr adj	1000	0.000496	0.001387
net5	net4 + augment	2000	0.000373	0.001184
net6	net5 + dropout	3000	0.001306	0.001121
net7	net6 + epochs	10000	0.000760	0.000787

The last and final thing is to now train specialist that each have the same final structure as net7 and each can identify a special facial keypoint so that the data which was thrown before can now be used. However this will make the training very slow for which we will use early stopping by during on_epoch_finished. Moreover to help with random initialization for each individual smaller net created it's better to use the weights from our larger net7. With this the implementation is complete.

Some of the complications that I encountered in the project were :

- How to utilize the 70% data that I had thrown initially
- Preventing Overfitting in Neural Nets

The way I got out of the first one was I went over Udacity's Neural Net's course again and there I got the idea to train specialists that is independent Neural Nets having structure similar to the final one but dealing with larger amount of data for a particular keypoint.

Then utilize these Neural Nets in the Final bigger NN in the end which gives the final answer/location of keypoint.

To prevent overfitting I remembered that best way is to provide more data , for which I tried Inversion of Images. However in my recent knowledge of *"Python Machine Learning"* book I realized that Regularization techniques seem to work remarkably well. So, a little bit digging in the internet brought to my attention the recent Dropout technique , which I thought to implement since I thought what better way to learn than to implement it.

Here are the **results of NN's after each Step** that I took as mentioned in the above list of steps:-

Name	Description	Epochs	Train loss	Valid loss
net1	single hidden	400	0.002244	0.003255
net2	convolutions	1000	0.001079	0.001566
net3	augmentation	3000	0.000678	0.001288
net4	mom + lr adj	1000	0.000496	0.001387
net5	net4 + augment	2000	0.000373	0.001184
net6	net5 + dropout	3000	0.001306	0.001121
net7	net6 + epochs	10000	0.000760	0.000787

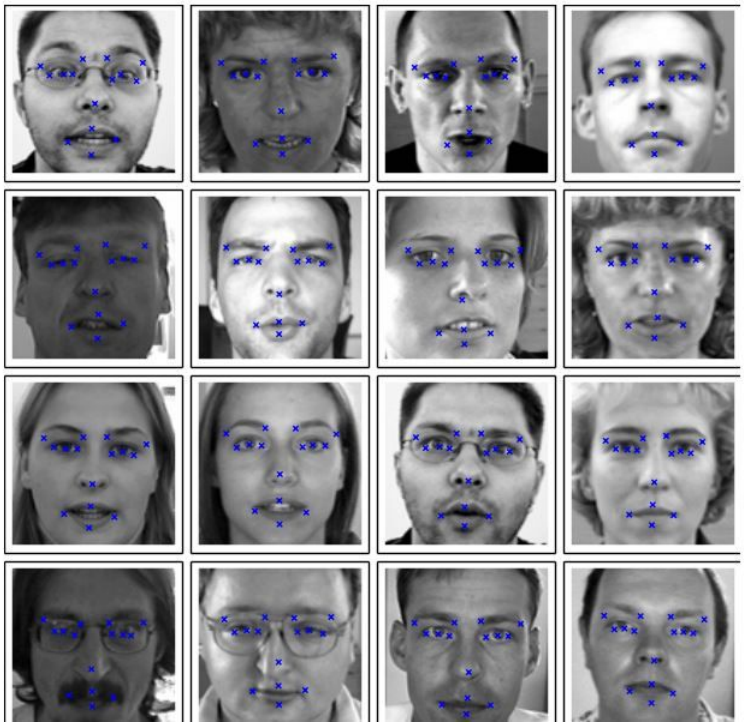
It's evident that each of our step is reducing the error a lot. The first major decrease was evident when we trained our first CNN compared to a normal NN. Hence we can say each step is important for the model and they have quite a lot of difference between them.

4. RESULTS

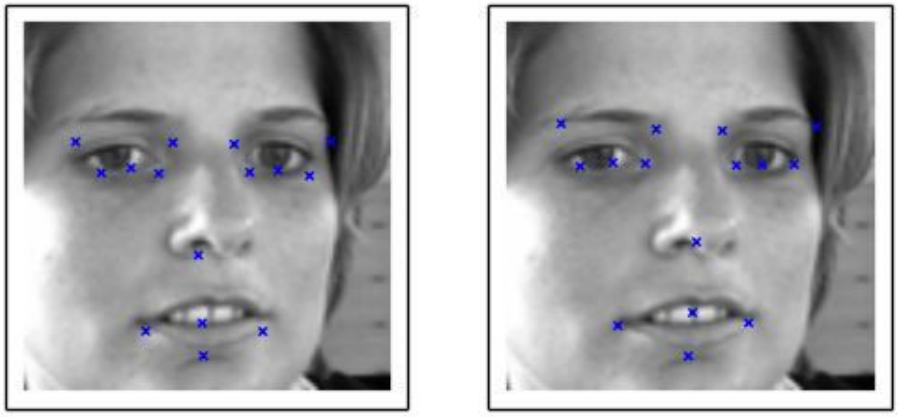
Model Evaluation , Results & Justification

I achieved 56th position on the kaggle Leaderboard with RMSE of 2.932.

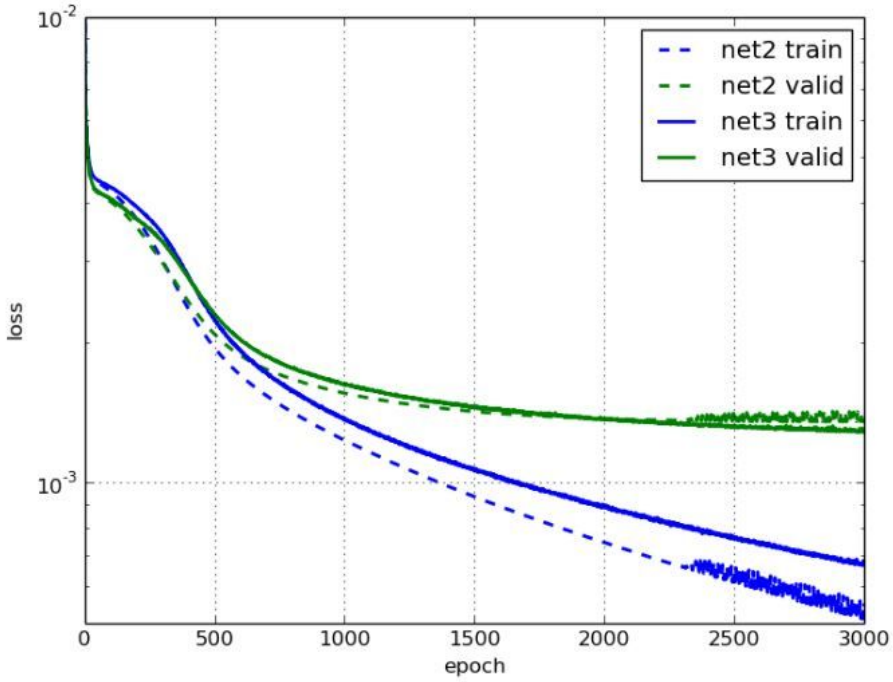
I will be adding a few plots and snaps since they speak a lot more.

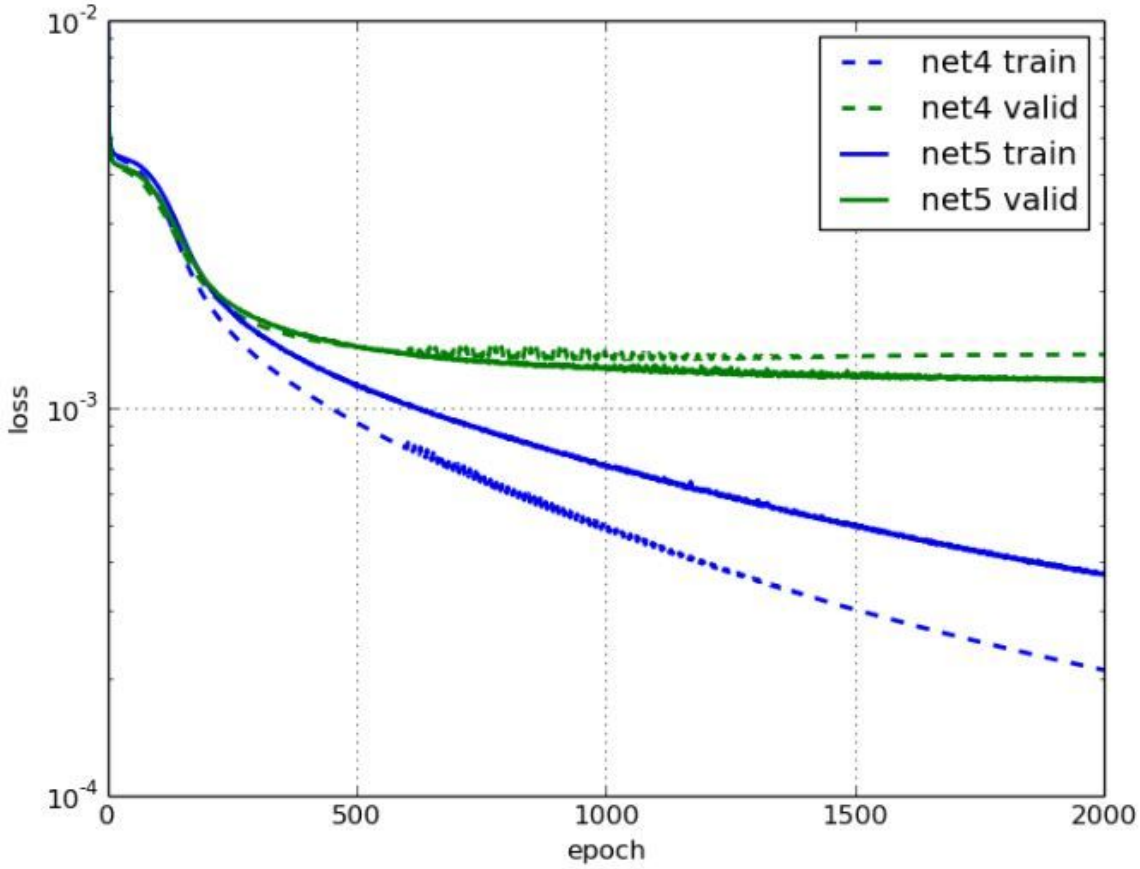


Results of Neural Net 1



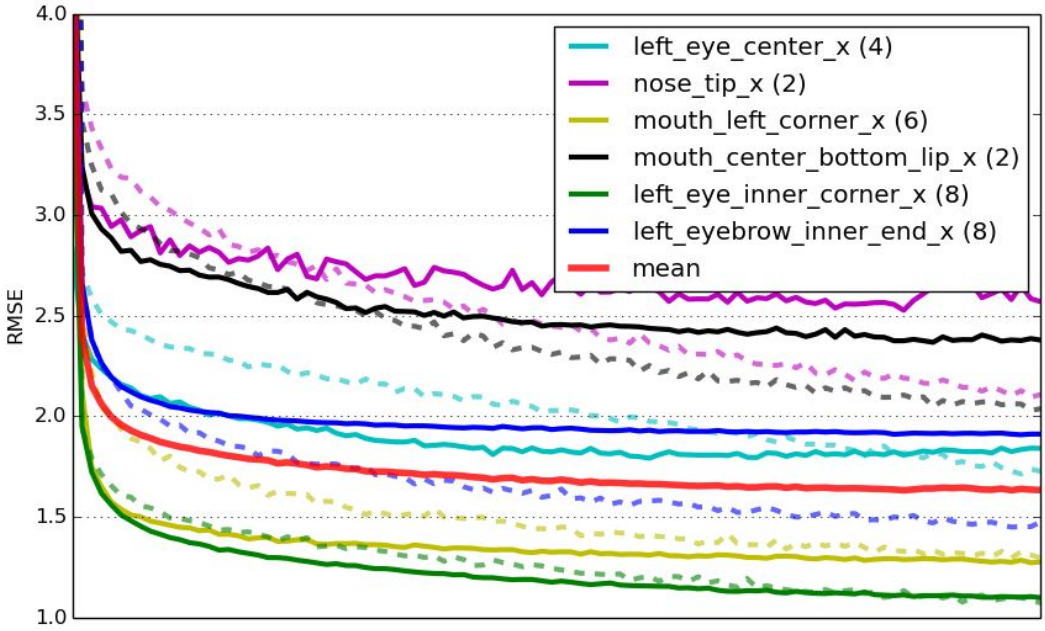
The predictions of net1 on the left compared to the predictions of net2.





Name	Description	Epochs	Train loss	Valid loss
net1	single hidden	400	0.002244	0.003255
net2	convolutions	1000	0.001079	0.001566
net3	augmentation	3000	0.000678	0.001288
net4	mom + lr adj	1000	0.000496	0.001387
net5	net4 + augment	2000	0.000373	0.001184
net6	net5 + dropout	3000	0.001306	0.001121
net7	net6 + epochs	10000	0.000760	0.000787

All the Neural Nets with their Validation errors and number of epochs



Learning curves for six specialist models that were pre-trained.

The final structure is as follows :

1. Input
2. Convolutional
3. Max Pooling
4. Dropout Layer
5. Convolutional
6. Max Pooling
7. Dropout Layer
8. Convolutional
9. Max Pooling
10. Dropout Layer
11. Hidden Layer
12. Dropout Layer

13. Hidden Layer
14. Output

A brief discussion on Model's Parameters :-

1. Learning _rate :- Linear decaying function , used it since as the model approaches to the optimum value it needs to make smaller steps to achieve more accuracy and not over shoot
2. Momentum - This helps in moving out of local minima , in such a complex problem there might be many cases of such hence we adopted the method of increasing momentum after every batch so as to make the model better.
3. max_epochs : this is the number of training samples the model has to be trained on. If it's low it can underfit if it's high it can overfit.
4. Batch_size : Since Batch Gradient Descent is faster and I couldn't go with other methods simply because BGD is good enough and the speed was very much needed due to my specifications.

The RMSE obtained is 2.932 which is above my expectations . It's ranked as the 56th model in the global leaderboard so I can definitely say my model solved the problem very well. Further parameter optimization can produce maybe better results but as for now the target has been achieved with a great result , and it will suffice for now.

5. CONCLUSION

Free-Form Visualization

Most of the plots/graphs/visualizations have been shown above. However in particular the following one sums up almost the whole project

Name	Description	Epochs	Train loss	Valid loss
net1	single hidden	400	0.002244	0.003255
net2	convolutions	1000	0.001079	0.001566
net3	augmentation	3000	0.000678	0.001288
net4	mom + lr adj	1000	0.000496	0.001387
net5	net4 + augment	2000	0.000373	0.001184
net6	net5 + dropout	3000	0.001306	0.001121
net7	net6 + epochs	10000	0.000760	0.000787

- This actually signifies that sometimes improvements might be seen as small however when coupled with other such small tuning can actually affect the model greatly. Let's say for the overfitting example , going from data augmentation to Dropout is a long way and has improved the model quite a lot.
- Moreover we know that If a CNN is getting better then it is bound to overfit for which NEED A REGULARIZATION , which here is Dropout
- The best way to speed up training times in such problems is EARLY STOPPING , which made the whole project possibly , given my specification of the computer
- Another interesting point to know is that even a little change in parameters such as learning_rate & momentum can affect the whole model a lot. This tuning

required lots of practice and a better GPU to speed up the training times , which in my case were close to 18 hours.

Reflection

The overall Project is successful. I have achieved a RMSE of 2.932 which fits into the kaggle leaderboard at 56th place and falls between my desired under 100 rank. The project's final structure was achieved through small steps and taking into account small goals. Firstly we achieved a bare good RMSE value. Then moving further on it was intended to help increasing data to prevent overfitting (Data Augmentation). After that to prevent the model from going to local minima and achieve better result , learning_rata & momentum were modified. However overfitting still persisted and for that , a regularization technique Dropout , recently introduced was implemented. Finally to use the data neglected till now specialist CNN's were trained for independent keypoint and the results were combined into a final CNN. Hence the final model was obtained.

The biggest problems were utilizing the data left out in the first step & the overfitting. This was very intriguing since both of them go hand in hand. Data Augmentation was still not enough and regularization technique had to be implemented . Here Udacity's Neural Net helped me with specialists technique/idea and a little bit research on Regularization Technique led me to DropOut. After it was simply a matter of tuning the parameters and training but the heart of the problem is the two things I mentioned above.

Improvement

There are various improvements that can be done :

- Hyperparameter optimization : tuning all other parameters more and exploring different optimization algorithms
- Usage of algorithms combined with a CNN of such complexity such as Histogram stretching , Mean Path Searching , Gaussian Blur , PCA etc
- Having a powerful GPU to train models faster so this would have allowed me to tune it better and explore more with parameters. This would enable me to add more layers and make my structure more detailed.