Hand Recognition Exploration Using Deep Learning Methods for Computer Vision

Fansheng Shi & Fanmin Shi

Abstract—The goal of this paper is to conduct a deep exploration in various popular deep learning architecture and methods and fine tuning on model hyperparameters to improve model accuracy and runtime on hand recognition tasks. There are three methods explored in this paper: the first one is the baseline, basic 5 layer CNN architecture micking traditional LeNet-5 architecture; the second one is the using transfer learning method on the modern model such as Xception, ResNet50, MobileNET. The metric of evaluation is accuracy and runtime. Once we down select a model, we would like to finetune the other optimization areas such as nonlinearity functions, different optimizer methods, and lossfunction used.

1 Introduction

The ambition of enhancing user interaction with the real world has been the topic of the last few decades. The first virtual reality HMD, named The Sword of Damocles [1] by Sutherland and his student. Later, more advanced headsets were developed in the 80s and 90s using higher resolution displays and more powerful computing platforms such as CyberMaxx by SEGA in 1994 [1]. With the rapid advance in both hardware and software such as heterogeneous system of chip with low power arm processors with various accelerators such as graphic processing unit (GPU) and neuro processing unit (NPU) mobile computing frameworks such Android and IOS and readily available machine learning libraries and accelerators. The revival of the AR and VR has commenced in the last decade, and the race to deliver the first AR headset among the tech giants, such as Google, Apple, and Meta has begun.

The motivation of this paper is to tackle the motion detection and recognition using hand gestures for AR/VR control. The reason is that we would like to achieve a frictionless experience from the users in the AR/VR space and eliminate the need of a joystick type of physical controller, because we believe frictionless experience allows the adoption of AR/VR devices to a wider audience. Hence, we believe motion control using hand gestures and recognition using computer vision methods are critical next steps for human to machine interface applications such as AR and VR space.

The proposed experiment is a four step process:1) data preparation, 2) model exploration, 3) model performance evaluation, and 4) model hyperparameter fine tuning. In the first data preparation step, we will use a 20000 image set with 10 hand gesture classes and split it with training and test sets with a split ratio of 8 to 2 gathered from Infrared HandRecognition Dataset from Kaggle.

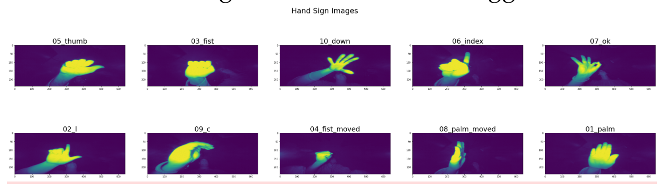


Fig. 1: Hand Recognition Datasets [13].

The dataset used is collected from 10 different hand-gestures (shown above) that were performed by 10 different subjects (5 men and 5 women) [3]. In the second model exploration step, we will build four models: a custom built CNN model like the LeNET 5 from the original CNN paper by LeCun and off the shelf models Xception, RestNET, and MoileNET with transfer learning.

Next we evaluate the model performance in terms of accuracy, memory size, latency and number of parameters used. And Lastly we fine tune the model with different learning rate, nonlinearity such as relu','sigmoid', and 'softmax', and optimizer methods during training phase such as RMSprop' ,'Adam', and 'Adadelta'.

2 Background

Convolution neural network (CNN) is a type of deep learning architecture commonly found in image classification. The classic CNN, LeNET 5 model originated from Yann LeCun's 1998 paper [4] consists of 7 layers, Convolution layer, pooling layer, Convolution layer, polling layer, convolution layer and lastly 2 fully connected dense layers are output 10 classes as shown below in Fig 2:

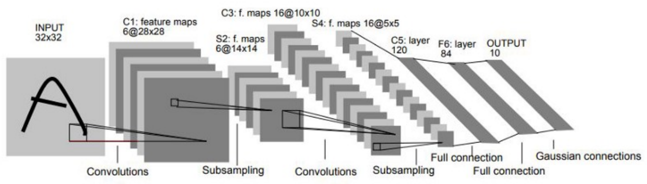


Fig. 2: LeNET 5 Architecture [4]

One interesting feature of LeNET 5 is that it does not include the modern nonlinearity layer such as Relu after each polling layer, instead the pooling layer itself is a nonlinear operation. Moving forward to modern day, in 2010, there was an imageNet challenge where different teams can submit their image classifier to the datasheets and the best model with the lowest error wins. Before

2012, all classifiers were handcrafted models using traditional computer vision techniques, However, in 2012 AlexNet used deep CNN architecture and won that year of the challenge, with an error rate of 16%, down from 25% from 2011. This sparked the deep learning era for computer vision. See below for a comparison between LeNET 5 and AlexNet

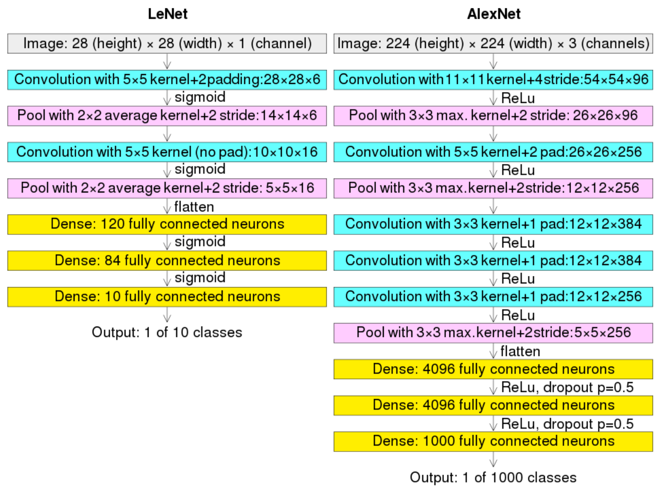

LeNet
AlexNet
Image: 28 (height) × 28 (width) × 1 (channel)
Convolution with 5×5 kernel+2padding:28×28×6
sigmoid
Pool with 2×2 average kernel+2 stride:14×14×6
Convolution with 5×5 kernel (no pad):10×10×16
sigmoid
Pool with 2×2 average kernel+2 stride: 5×5×16
flatten
Dense: 120 fully connected neurons
sigmoid
Dense: 84 fully connected neurons
sigmoid
Dense: 10 fully connected neurons
Output: 1 of 10 classes
Image: 224 (height) × 224 (width) × 3 (channels)
Convolution with 11×11 kernel+4stride:54×54×96
ReLu
Pool with 3×3 max. kernel+2 stride: 26×26×96
Convolution with 5×5 kernel+2 pad:26×26×256
ReLu
Pool with 3×3 max.kernel+2stride:12×12×256
Convolution with 3×3 kernel+1 pad:12×12×384
ReLu
Convolution with 3×3 kernel+1 pad:12×12×384
ReLu
Convolution with 3×3 kernel+1 pad:12×12×256
ReLu
Pool with 3×3 max.kernel+2stride:5×5×256
flatten
Dense: 4096 fully connected neurons
ReLu, dropout p=0.5
Dense: 4096 fully connected neurons
ReLu, dropout p=0.5
Dense: 1000 fully connected neurons
Output: 1 of 1000 classes

Fig. 3: Comparison of the LeNet and AlexNet convolution, pooling and dense layers

The main difference between Alexnet and LeNET is that AlexNet added Relu nonlinearity functions and extended to more layers and output has 1000 classes instead of 10 classes.

In 2015 and 2016, two major works in deep learning research helped define the current CNN architecture, batch normalization by Sergey Ioffe and Christian Szegedy [5] and residual network by He et al, in which now the CNN architecture can be extend to 50 layers such as in ResNET50. This residual network which essentially allows skipped connections to reduce vanishing gradient and easier to optimize reached 3.57% error on the ImageNet test set [6].

The generic CNN architecture feature consists of following building blocks, every convolution layer consists of convolution operation, pooling, and nonlinear functions followed by a fully connected layer or traditionally called multilayer perceptron(MLP) neural network at the output. Note, a MLP layer just means it connects every node in this layer to every node in the next year. The learned weights and bias vectors are the filter banks that use convolution operation. The filters contain the features of the image such as an edge. The iterative training updates the weights and bias vectors (filter banks) and they are also the CNN architecture parameters.

With the understanding of the history of CNN architectures and modern modification such as Residual network and batch normalization, we can train a deep neural network with high accuracy and stabilization for image classification application. In addition, there is a mobile network which is developed for low memory low compute mobile computing cases as well. And this is why we choose a deep learning CNN model for hand recognition tasks. Note, we are not building a model from scratch but rather we are going to evaluate different models using transfer learning based on model performance (i.e accuray) and number for parameters (i.e memory usage) for realtime AR/VR use cases.

3 Method and Experiment Setup

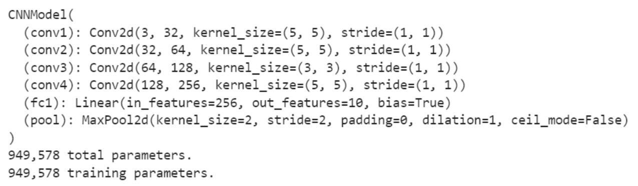
The goal of this design of the experiment is model exploration in terms of architecture definition such as simple 6 layer CNN models with and without self attention layer, and transfer learning model using off the shelf models such as Xception, ResNET50, and MobileNET.

The dataset used in a 20000 images from kaggle with 10 classes: '01_palm', '02_l', '03_fist', '04_fist_moved', '05_thumb', '06_index', '07_ok', '08_palm_moved', '09_c', '10_down'. Then we split the dataset into the 8 to 2 ratio for training and validation datasets respectively.
The loss function that we used in cross entropy because this is a classification task.

After setuping the training steps during the training phase we plan to evaluate model performance. The metrics for evaluation are: training loss per epoch ,validation accuracy per epoch, total # of trainable parameters model memory size, and execution latency.
We also plan to evaluate different nonlinearity functions in CNN architecture definition phase and optimizers in the training phase to see its impacts on model inference accuracy.

3.1 Model Exploration on 5 Layer Basic CNN Model

The basic model is consists of 5 layers, 4 convolution layer and 1 fully connected layer as shown below



```
CNNModel(
  (conv1): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (conv4): Conv2d(128, 256, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=256, out_features=10, bias=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
949,578 total parameters.
949,578 training parameters.
```

Fig. 4: Basic 5 layer CNN model

Each convolution layer is followed by a Relu layer.

Table 1: Hyperparameters

Hyperparameters	Configuration
Depth	5
Parameters	949,578
Learning Rate	0.001

Number of Epochs	10
Batch Size	32

We trained using 10 epoches and evaluated the model validation accuracy and training loss each epoch iteration. This allows us to see how the performance compares against other models.

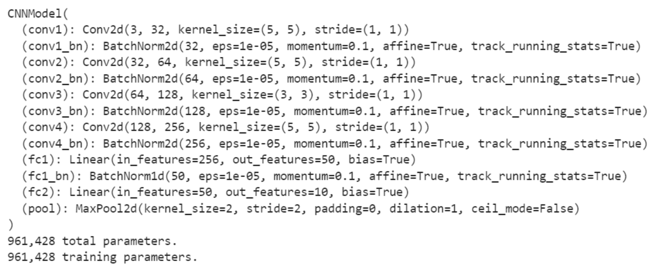
3.2 Basic Model Exploration with Batch Normalization and High Learning Rate

We would like to decrease the training time by possibly lowering the total number of epochs. One technique is using a higher learning rate but with a deeper neural net, it's harder to initialize weights and achieve fast convergence. One solution is to use batch normalization where we normalize the layer output value before sending it to the successive layer. The normalization is simply centered data over unit norm as shown below in eq1.

eq1:
$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

Batching Normalization Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity. After batch normalization, networks become more robust to initialization and it also acts as regularization during training [11]

For the same exact model, we added batch normalization after each convolution layer as well as the fully connected dense layer before the activation function and pooling layer. Moreover, for the fully connected layer, we break the layer into two layers with input size. The full model is shown below:



```
CNNModel(
  (conv1): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1))
  (conv1_bn): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (conv2_bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (conv3_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(128, 256, kernel_size=(5, 5), stride=(1, 1))
  (conv4_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1): Linear(in_features=256, out_features=50, bias=True)
  (fc1_bn): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
961,428 total parameters.
961,428 training parameters.
```

Fig. 5: Basic 6 layer CNN model with Batch Normalization

Model Architecture and training hyperparameters are summarized below.

Table 2: Hyperparameters

Hyperparameters	**Configuration**
Depth	6
Parameters	961,428
Learning Rate	0.005/0.01/0.05
Number of Epochs	10
Batch Size	32

3.3 Basic Model Exploration with Self Attention layer with Normalization and High Learning Rate

There is another optimization that one can do to improve the accuracy of the classification by adding an extra layer in the convolution architecture self attention layer. This is achieved by using a mechanism called "attention" that allows the network to weigh the importance of different parts of the input when making predictions or classifications. As a result, prediction improves. An example of self attention architecture is shown below:

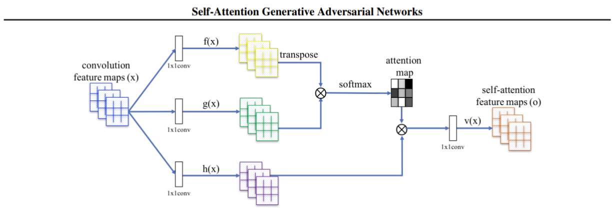
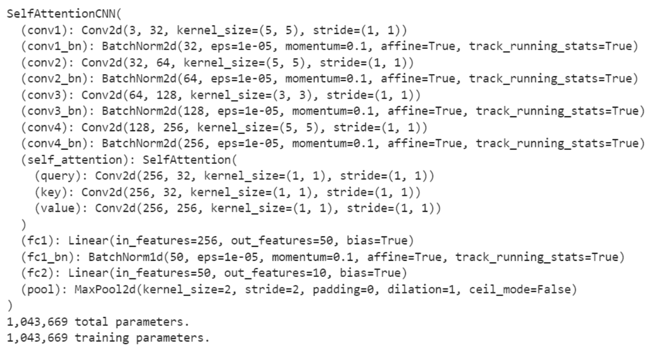

Self-Attention Generative Adversarial Networks
convolution feature maps (x)
1x1conv
f(x)
transpose
1x1conv
g(x)
softmax
attention map
1x1conv
h(x)
self-attention feature maps (o)
v(x)
1x1conv

Fig. 6: Self Attention Layer and Operation [12].

The image features from the previous hidden layer are first transformed into two feature spaces f, g to calculate the attention map o. [12]

The full model architecture is summarized below:



```
SelfAttentionCNN(
  (conv1): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1))
  (conv1_bn): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (conv2_bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (conv3_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(128, 256, kernel_size=(5, 5), stride=(1, 1))
  (conv4_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (self_attention): SelfAttention(
    (query): Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
    (key): Conv2d(256, 32, kernel_size=(1, 1), stride=(1, 1))
    (value): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
  )
  (fc1): Linear(in_features=256, out_features=50, bias=True)
  (fc1_bn): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
1,043,669 total parameters.
1,043,669 training parameters.
```

Fig. 7: Self Attention CNN model with Batch Normalization

Model Architecture and training hyperparameters are summarized below.

Table 3: Hyperparameters

Hyperparameters	**Configuration**
Depth	7
Parameters	1043699

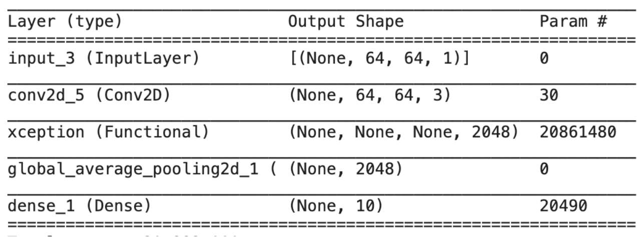
Learning Rate	0.005
Number of Epochs	10
Batch Size	32

3.4 Model Exploration with Transfer Learning on Xception, ResNET50, and MobileNET Networks

Transfer learning is a technique that takes features learned from one problem and then leverages them on a similar problem. For example, knowledge gained while learning to recognize cars can be used to recognize trucks. Transfer learning has a few advantages over traditional machine learning; It is computationally more efficient and achieves good results using a small data set; it achieves optimal performance faster than the traditional machine learning model because it leverages knowledge (features, weights, etc.) from previously trained models already understanding the features.

When applying transfer learning in the context of deep learning, the general idea is to add new layers on top of a trained model, and then train the new layers. The detailed workflow is 1) Taking layers from the pre-trained model. 2) Freeze the layers to avoid destroying the weights from future training. 3) Add new trainable layers on top of the frozen layer. 4) Train the new model with the dataset.

Our hand gestures dataset contains grayscale images of dim (W, H, 1). This won't work as the input to the most imagenet pre-trained models dataset simply due the the input takes in RGB image of the dim (W, H, 3). In order to use the pre-trained models, we add an additional layer to connect our input (W, H, 1) to the input (W, H, 3) of the pre-trained model. Furthermore, we also need to convert the output of the pre-trained model to the desired number of classes. Hence, we add a global_average_poolin_2d layer to convert the pre-trained model 4D output to 2D and then connect it to a dense layer that outputs the desired number of classes. See figure 4 to see the detailed layers.



```
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 64, 64, 1)]       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 64, 64, 3)         30
_________________________________________________________________
xception (Functional)        (None, None, None, 2048)  20861480
_________________________________________________________________
global_average_pooling2d_1 ( (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                20490
=================================================================
```

Fig 8. Transfer Learning Architecture

We will explore 3 pre-trained models based on imagenet dataset to explore their model size, classification accuracy, and time (ms) per inference.

The first model we will use is the Xception [8] which is a Deep Learning model with Depthwise Separable Convolutions. The paper claims to significantly outperform Inception V3 on a larger image classification dataset [8] as well as the ResNet.

The second model we will use is ResNet50 [9]. It is a popular deep learning model that won many 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation in 2015. We expect the model to have a good accuracy but not as well as the Xception model.

The third model we will try is MobileNet which is designed for mobile and embedded vision applications [10]. It significantly reduces the number of parameters when compared to the network with regular convolutions with the same depth in the nets. This results in lightweight deep neural networks. This model should produce the lowest inference latency but expect to have a lower accuracy due to a simpler network.

3.5 Model Architecture and Training Fine Tuning

In this experiment, we plan to change the first two layers of a basic 6 layer convolutional network with customization nonlinearities. Then in the training phase, we fixed the nonlinearity functions for all layers but varied optimizers. The goal is to see which nonlinearity function and optimizer methods achieve the highest accuracy score. Following lists a set of optimizers and activation functions used in the experiment.

Optimisers:
['RMSprop','Adam','Adadelta','Adagrad']

Activationfunction:
['relu','sigmoid','softmax','tanh','softsign','selu','elu']

4 Experiment Result

Model accuracy, training loss, memory size and runtime are shown in each section of the experiment.

4.1 Basic Model Exploration on 5 Layer Basic CNN Model Result

The result for just the 5 layer network reached a 99% accuracy for the validation set. The accuracy and the training loss plots are shown below.

Learning Rate	Accuracy	Train Time (s)
0.001	0.9925	2644

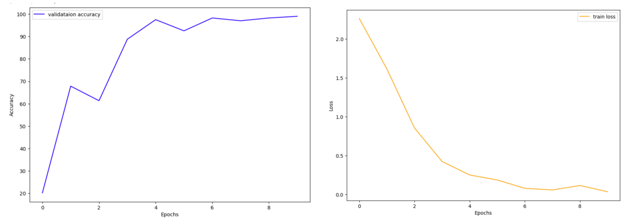


Fig. 9: Validation Accuracy (Left) and Training Loss (Right) Plots vs

From the above figure, the accuracy converges quite well in the end.

4.2 Basic Model Exploration with Batch Normalization and High Learning Rate result

The result table shown below

Learning Rate	Accuracy	Train Time (s)
0.005	0.9925	2644
0.01	0.9925	2698
0.05	0.9601	2661

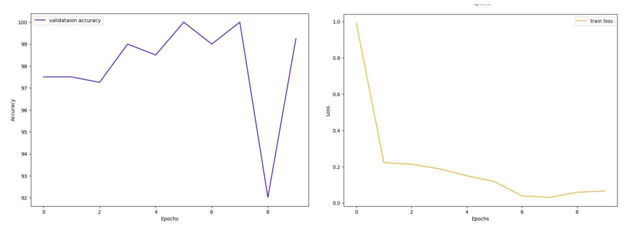


Fig. 10: Validation Accuracy (Left) and Training Loss (Right) Plots vs Epochs with lr = 0.005

One interest fact is that with 0.005 training rate, the model after epoch #1 reached a sore of 97%, and reached 100 at epoch # 7. This means that the model can reach very high accuracy with fewer epochs compared to that of the simple 5 layer Basic CNN model without batch normalization.

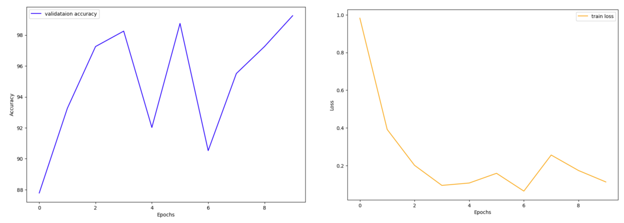


Fig. 11: Validation Accuracy (Left) and Training Loss (Right) Plots vs Epochs with lr = 0.01

From the above figure with learning rate of 0.01, the accuracy reached to 88% with just 1 epoch and 98% at the 4th epoch, however the accuracy plot fluctuated between 92 to 98% before it converged to 99% at 10th epoch. One can see that there is somewhat of an instability here with a high learning rate.

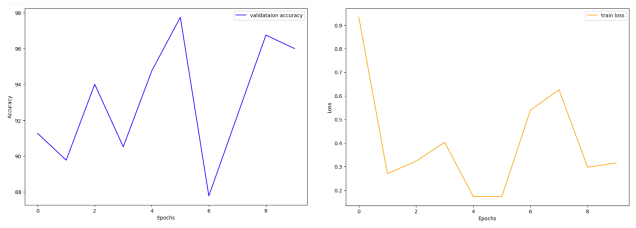


Fig. 12: Validation Accuracy (Left) and Training Loss (Right) Plots vs Epochs with lr = 0.05

From the above plot with a learning rate of 0.05, we can see the accuracy fluctuate throughout all the training phases, this means that the model has a hard time converging after reaching above 90% after the first step. This makes sense since we are using a quite large training step. For the optimal learning rate, we decide that 0.005 is a good one for this model. We will use 0.005 for the next experiment.

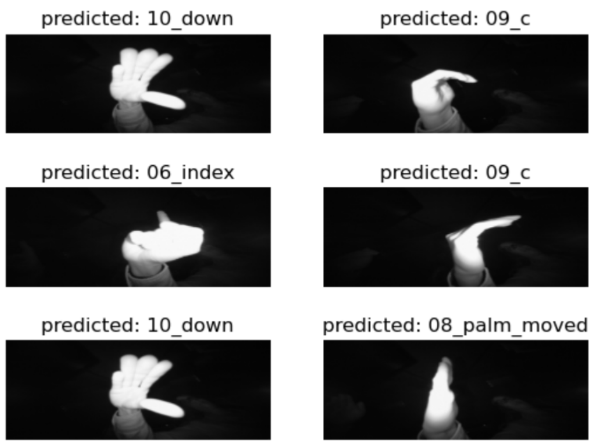


Fig. 13: Predicted Images using the CNN Model

In the above figure, we provide visualization to the predicted class on a few sets of images in the validation set. As one can see, all the classes correctly match the hand gestures.

4.3 Basic Model Exploration with Self Attention layer with Normalization and High Learning Rate

With self attention added to the 6 layered CNN with batch normalization at 0.005 learning rate, we see that the model converges to 100% at the epoch iteration 10. This is an improvement from the previous result which does not have an extra self attention layer. The result is summarized below.

Learning Rate	Size (MB)	Accurac y	Train Time [S]	Latency (ms)
0.005	3.985MB	1.00	3309	0.064

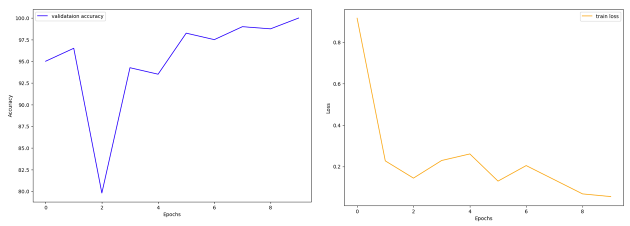


Fig. 14: Validation Accuracy (Left) and Training Loss (Right) Plots vs Epochs with Self Attention @ lr = 0.005

As we can see that self attention improves accuracy and also seems to converge more stably at higher epoch compared to its counterpart with the same learning rate as depicted in figure x. The model size is fairly small, 3.985 MB, since we constructed the model from scratch. The size is calculated by following:

> size_all_mb = (param_size + buffer_size) / 1024**2, where param_size is the total parameter size calculated by multiplying the total parameters of the model by 4 byte (i.e each parameter is 4 byte or 32 bits) and the same goes for the buffer_size, where each buffer is another tensor tracking non trainable parameters such as mean and std in batchnorm layers and each buffer is also 4 byte.

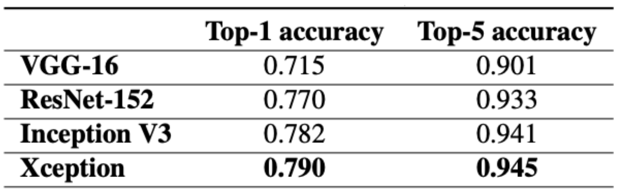
4.4 Transfer Learning Models Explorations

goo

Model	Size (MB)	Accuracy	Training time (s)	Latency (ms)
Xception	88	.994	160	.36
ResNet50	98	.999	160	.33
MobileNet	16	.934	130	.11

Table 3: Transfer learning models comparison

We are a bit surprised that the Xception model didn't perform as well as the ResNet50 in terms of the accuray. Given that Xception model is newer than ResNet. In addition, the Xception paper itself mentioned that its accuracy is higher than the ResNet [8] as shown below.



	Top-1 accuracy	**Top-5 accuracy**
VGG-16	0.715	0.901
ResNet-152	0.770	0.933
Inception V3	0.782	0.941
Xception	**0.790**	**0.945**

Fig 15. Classification performance comparison on ImageNet

However, we are not surprised that the MobileNet achieves the lowest latency (3 times faster than other two) because the model size is around 5 - 6 time smaller.

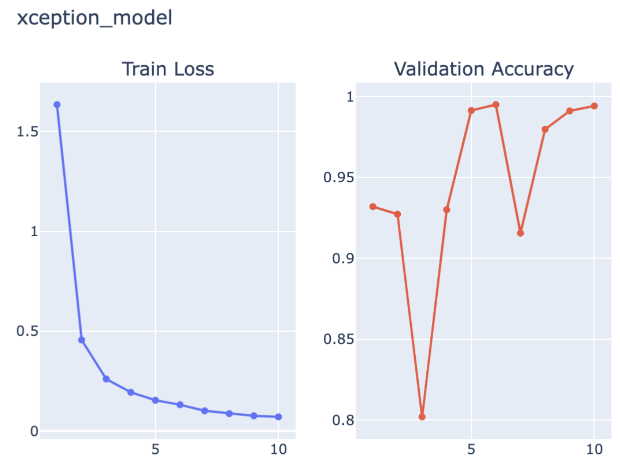


Fig 16: Training Loss and Validation Accuracy over epochs on Xception pre-trained model.

The Xception model is able to reach its top accuracy(.994) around Epoch 5. However, the accuracy drops down at Epoch 6 and then slowly reaches to the top around Epoch 10. It appears that the Xception model training improvement is a bit unpredictable..

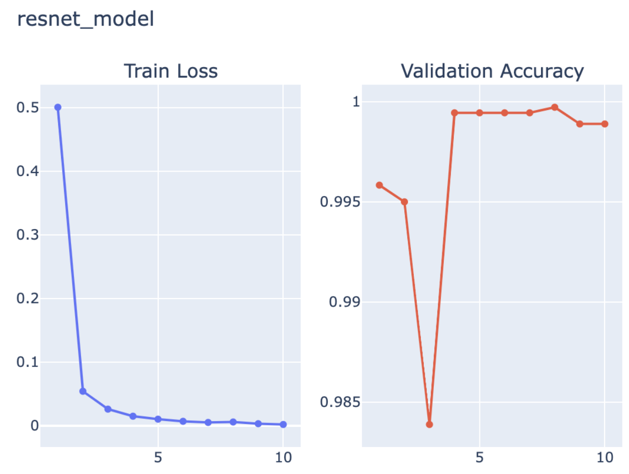


Fig 17: Training Loss and Validation Accuracy over epochs on ResNet50 pre-trained model.

The ResNet is able to reach the peak accuracy(.999) at the Epoch 4 and stays around that accuray for the rest of the Epoch. Hence the ResNet model has a consistent and gradual training improvement.

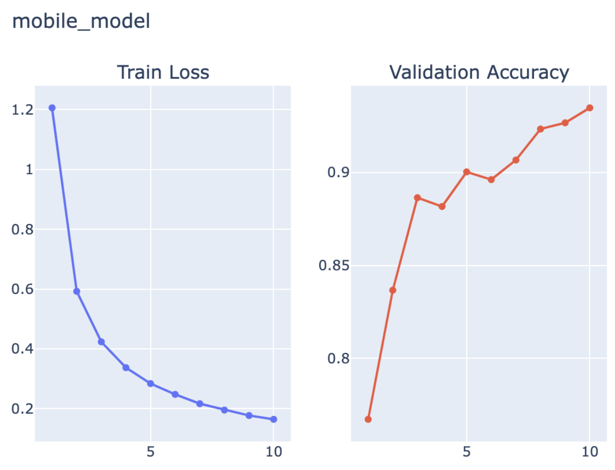


Fig 18: Training Loss and Validation Accuracy over epochs on MobileNet pre-trained model.

The MobileNet gradually reaches a high accuracy (.934) at Epoch 10. It doesn't converge as quickly as the other two models and has a much lower accuracy as well.

By comparing the rate of convergence and accuracy, the ResNet is the best out of the three because it achieves a .999 accuracy quite quickly. However, If we take into account the latency, the MobileNet is 3 times faster than ResNet but at a reduced accuracy .934 vs .999 (ResNet).

In conclusion, the pre-trained models work well on our grayscale dataset. The ResNet does extremely well with fast convergence and .999 accuracy, and the MobileNet achieves .934 accuracy but at a much lower inference latency than ResNet, .11 ms (MobileNet) vs .33 ms (ResNet). The difference between .33 ms and .11 ms is negligible with model mobile devices and probably won't be noticed by the users. Hence we recommend ResNet to because it has highest accuracy and quite low inference latency.

4.5 Model Architecture and Training Fine Tuning.

The results showed that Relu and RMSprop and Adam provide the highest training accuracy. Following two comparison tables summarize the results.

Table 4: Accuracy vs Nonlinearity Functions

Epochs	Optimizer	Activation_Function	Train_Loss	Train_Accuracy
10	RMSprop	tanh	0.06326176226	0.9835714102
10	RMSprop	elu	0.0626912117	0.9855555296
10	RMSprop	softmax	0.0601394549	0.9842857122
10	RMSprop	softsign	0.05525505543	0.9864285588
10	RMSprop	selu	0.05331727862	0.9859523773
10	RMSprop	relu	0.04389603436	0.9903174639
10	RMSprop	sigmoid	0.06428583711	0.9834920764

As shown in the above table, RELU activation function provides the highest training accuracy of around 99%.

Table 5: Accuracy vs Optimizer

Epochs	Optimizer	Activation_Function	Train_Loss	Train_Accuracy
10	Adadelta	relu	3.101783752	0.123968251
10	RMSprop	relu	0.05625912547	0.986587286
10	Adagrad	relu	1.453060031	0.4994444549
10	Adam	relu	0.05803629011	0.9851587415

As seen from the above image, RMSPROP and ADAM provide the highest training accuracy of around 98.5%.

5 Discussion

We started the experiment by constructing a simple network using the various foundation blocks (i.e structure hyperparameters) of CNN architecture such as convolution layers, batch normalization layers, dense layers, self attention layers, and nonlinearities. We learned that a simple 6 layer CNN with batch normalization and self attention layer trained on 2000 images can reach 99% accuracy with only 4MB of memory. In addition, we experimented in the training phase with different learning rate, nonlinearity, and optimizer functions and found empirically that combination of 0.005 learning rate, Relu, and RMSPROP and ADAM optimizer result in highest model inference accuracy. From building our own CNN architecture, we learned that with enough data, we can reach a very high accuracy, ~98%, with minimal model memory size, 4MB. This result is essential for an low profile inference engine on a AR/VR mobile platform which has limited compute and memory.

The customized model is not scalable and flexible when there is new data and has to be restrained constantly to keep the accuracy score high, which is not a good user experience for the AR/VR device that is targeted for the mass market. Hence, in the next experience, we tried to build a more generalized inference engine on a much massive datasheet, (hundred of million parameters) modern classification network and adapted it for hand recognition using transfer learnings. Three neural nets are tested, ResNet, MobileNet and Xception.

We found that ResNET has the highest accuracy of 0.999 percent but has the highest model size of 99 MB. On the other hand, the MobileNet has the lowest model size of 16 MB and also latency ⅓ of ResNET and MobileNET but at a lowest accuracy score of 0.934. With the customized designed 6 Layer CNN model with self attention, 0.064 ms and 4 MB of data at 99% accuracy.

With a resource constrained system such as a standalone next generation of AR/VR device, we recommend the development team to start with transfer learning using ResNET and MobileNET as a starter point for hand recognition. The reason is that the training time is much less on a GPU compared to a model built from scratch, a 15 to 20x time improvement. For flashship devices with larger memory and multicore systems (i.e typical dual quad cores and 6GB mobile) with a decent size of battery, ResNET should be sufficient. For low power design, which means running inference on a dedicated accelerator such as the image streaming processing unit on the handset, a MobileNET based model can be used.

However when the development has reached a mature level, the team can explore using a customized model which has much lower latency and a very good accuracy which can further optimize the performance. And we recommend using Relu the nonlinearity function and Adam for optimizer.

There's always system design trade offs from systems with constraints, and it is through user study that we can validate who model suits for what use case and having these options available early on help us plan the road ahead in AR/VR development.

6 Statement of Individual Contribution

Fansheng Shi worked on the Basic model explorations on CNN architecture definition including batch normalization, self learning, and nonlinear functions, learning rate and optimizer optimization. Also Fansheng worked on data preparation and model performance evaluation of the Basic CNN model including training time, accuracy, memory, and latency.

Fanmin Shi worked on transfer learning model explorations where he coded up the three models in tensorflow and made it work on grayscale images. He gathered the experimental results and wrote his findings on model size, training time, and latency and provided his recommendation on which pre-train model to use.

References

[1] D. Barnard, "History of VR - timeline of events and Tech Development," *VirtualSpeech*, 06-Oct-2022. [Online].Available: https://virtualspeech.com/blog/history-of-vr#:~:text=1968,simple%20virtual%20wire%2Dframe%20shapes. [Accessed: 12-Dec-2022].

[2] https://keras.io/guides/transfer_learning/

[3] T. Mantecón, C.R. del Blanco, F. Jaureguizar, N. García, "Hand Gesture Recognition using Infrared Imagery Provided by Leap Motion Controller", Int. Conf. on Advanced Concepts for Intelligent Vision Systems, ACIVS 2016, Lecce, Italy, pp. 47-57, 24-27 Oct. 2016. (doi: 10.1007/978-3-319-48680-2_5)

[4] LeCun, Y.; Bottou, L.; Bengio, Y. & Haffner, P. (1998).

[5] Gradient-based learning applied to document recognition.Proceedings of the IEEE. 86(11): 2278 - 2324.

[6] "Batch normalization," *Wikipedia*, 03-Oct-2022. [Online]. Available: https://en.wikipedia.org/wiki/Batch_normalization. [Accessed: 12-Dec-2022].

[7] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.

[8] Chollet, François. "Xception: Deep learning with depthwise separable convolutions." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017.

[9] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[10] Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).

[11] "Lecture 7: Convolutional Networks - Electrical Engineering and computer ..." [Online]. Available: https://web.eecs.umich.edu/~justincj/slides/eecs498/WI2022/598_WI2022_lecture07.pdf. [Accessed: 13-Dec-2022].

[12] Han Zhang, Ian Goodfellow, Dimitris Metaxas, Augustus Odena Proceedings of the 36th International Conference on Machine Learning, PMLR 97:7354-7363, 2019

[13] Hang Recognition Dataset, https://www.kaggle.com/datasets/gti-upm/leapgestrecog