Implementing Deep Neural Networks for Classification and Regression Tasks

Sepinood Ghiami
Department of Computer Science
Shahid Beheshti University

***Abstract*—In this Report, we discuss the theory behind deep learning and then we have a discussion on Multi-layer Perceptron (MLP) and Convolutional Neural Network (CNN), after that we will develop these networks in Python and train and test the networks via Fashion_mnist and Cifar_10 datasets.**

***Keywords*—Deep Learning, Convolutional Neural Networks, CNN, Multi-layer Perceptron, MLP**

I. INTRODUCTION (*HEADING 1*)

Deep learning has truly become a mainstream in the past few years. Deep learning uses neural nets with a lot of hidden layers (dozens in today's state of the art) and requires large amounts of training data. These models have been particularly effective in gaining insight and approaching human-level accuracy in perceptual tasks like vision, speech, language processing. The theory and mathematical foundations were laid several decades ago. Primarily, two phenomena have contributed to the rise of machine learning: a) Availability of huge datasets/training examples in multiple domains and b) Advances in raw compute power and the rise of efficient parallel hardware.

Deep Learning Algorithms use something called a neural network to find associations between a set of inputs and outputs. The basic structure is seen in fig.1:

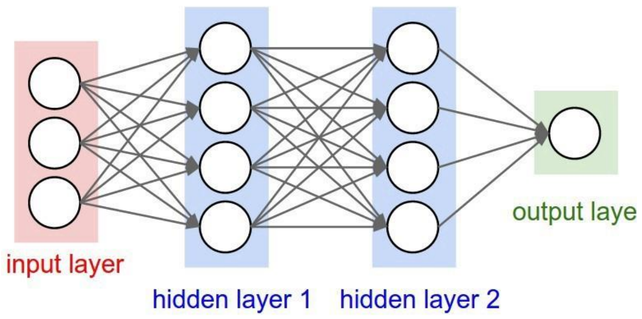


Fig.1: A Neural Network representation

A neural network consists of input, hidden, and output layers — all of which are composed of "nodes". Input layers take in a numerical representation of data (e.g. images with pixel specs), output layers output predictions, while hidden layers are correlated with most of the computation. information is passed between network layers through the function shown in fig.2. The major points to keep note of here are the tunable weight and bias parameters — represented by w and b respectively in the function above. These are essential to the actual "learning" process of a deep learning algorithm.

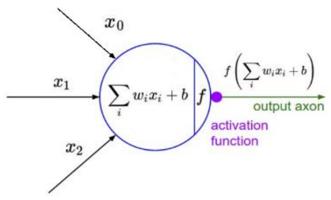


Fig.2: Node with Activation function representation

After the neural network passes its inputs all the way into its outputs, the network evaluates how good its prediction was (relative to the expected output) through something called a loss function. As an example, the "Mean Squared Error" loss function is shown below.

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \quad (1)$$

The goal of our network is to ultimately minimize this function by adjusting the weights and biases of the network. In using something called "back propagation" through gradient descent, the network backtracks through all its layers to update the weights and biases of every node in the opposite direction of the loss function — in other words, every iteration of back propagation should result in a smaller loss function than before.

II. MULTI LAYER PERCEPTRON (MLP)

in perceptron we just multiply weights and add Bias, but we do this only in one layer. We update the weight when we found an error in classification or miss-classified data. Weight update equation is shown below:

weight = weight + learning_rate * (expected - predicted) * x

In the Multilayer perceptron, there can be more than one linear layer (combinations of neurons). In a three-level network, the first layer will be the input layer and the last will be output layer and the middle layer is called hidden layer. We feed our input data into the input layer and take the output from the output layer. We can increase the number of the hidden layers as much as we want, to make the model more complex according to our task(fig.4).

Feed Forward Network, is the most typical neural network model. Its goal is to approximate some function f (). Given, for example, a classifier **y = f ∗ (x)** that maps an input x to an output class y, the MLP find the best approximation to that classifier by defining a mapping, y = f(x; θ) and learning the best parameters θ for it. The MLP networks are composed of

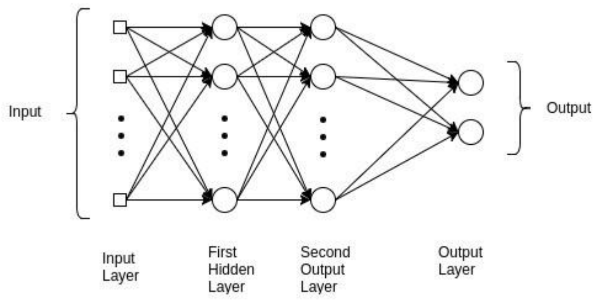


fig.3 MLP representation of NN

many functions that are chained together. A network with three functions or layers would form f(x) = f (3)(f (2)(f (1)(x))). Each of these layers is composed of units that perform an affine transformation of a linear sum of inputs. Each layer is represented as y = f(WxT + b), Where f is the activation function, W is the set of parameters or weights, in the layer, x is the input vector, which can also be the output of the previous layer, and b is the bias vector. The layers of an MLP consists of several fully connected layers, because each unit in a layer is connected to all the units in the previous layer. In a fully connected layer, the parameters of each unit are independent of the rest of the units in the layer, that means each unit possess a unique set of weights.

III. CONVOLUTIONAL NEURAL NETWORK (CNN)

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

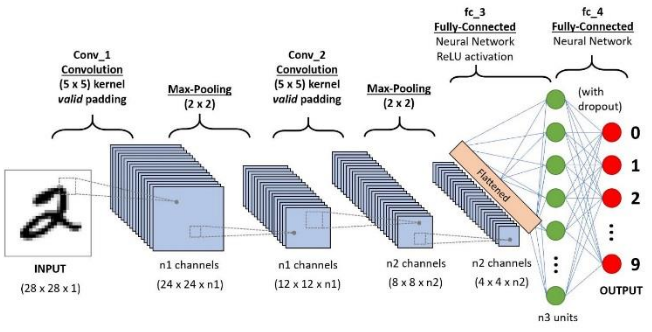

Conv_1 Convolution (5 x 5) kernel valid padding
Max-Pooling (2 x 2)
Conv_2 Convolution (5 x 5) kernel valid padding
Max-Pooling (2 x 2)
fc_3 Fully-Connected Neural Network ReLU activation
fc_4 Fully-Connected Neural Network
(with dropout)
Flattened
INPUT (28 x 28 x 1)
n1 channels (24 x 24 x n1)
n1 channels (12 x 12 x n1)
n2 channels (8 x 8 x n2)
n2 channels (4 x 4 x n2)
n3 units
0
1
2
9
OUTPUT

Fig.4 CNN representation of an NN

In the fig.5, we have an RGB image which has been separated by its three colored planes — Red, Green, and Blue. There are a number of such color spaces in which images exist — Grayscale, RGB, HSV, CMYK, etc.

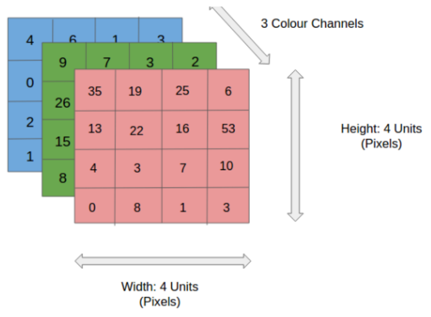


Fig.5: A matrix representation of an image

Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in three dimensions: width, height, depth. (Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network). For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively).

As we described above, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet architecture.

A simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more details:

INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.

- CONV layer will compute the output of neurons that are connected to the local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the max(0,x), thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a down sampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, ConvNets transform the original image, layer by layer, from the original pixel values to the final class scores.

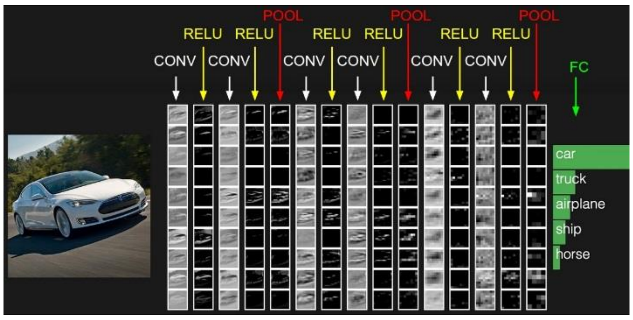


Fig.6: Steps of a simple CNN

IV. Implementation:

In this section we explain the implementation details of our networks. We implemented our models in tensorflow framework. All variables are randomly initialized under the default setting of tensorflow. Inputs are simply normalized in range[0,1]. To achieve network analysis, we divide our work in two parts. First, with a default sets of parameters, we investigate architectural design such as the number of layers and units. Then we choose a model to investigate on hyper-parameters selection such as learning rate, drop rate and batch size.

For the first part, default parameters are:

- learning rate: 0.001
- batch size: 64
- epochs: 20
- optimizer: Adam
- activation: ReLu
- kernel size: (3,3)
- drop rate: 0.0 (no dropout)
- loss function: soft max cross entropy
- regularization: None

also, to compare our models, we use accuracy defined as true positives divided by all samples. To evaluate our work, we use two common datasets, fashion mnist and cifar10.

A. *Fully Connected Network (MLP) Implementation:*

By fixing activation functions and hyper parameters, we have two other choices to design a MLP: number of layers and number of units per layer. For the rest of this report, we refer to number of layers as depth and number of units as width of the network. The width and depth of network depend on each other and we can not just fix one of them and change the other to achieve a good model. But for the sake of discussion, first we use a simple one-layer network with width W and then we scale W to see the effect of width in MLP networks. to investigate on depth, we fix network width to 16 for fashion mnist and 64 for cifar10 and then change the number of layers.

A.1. Experiment on Fashion mnist

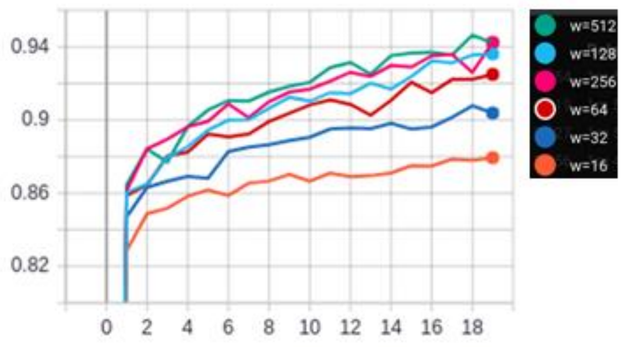


Fig.7: Scaling model with different network width, fashion mnist(train accuracy)

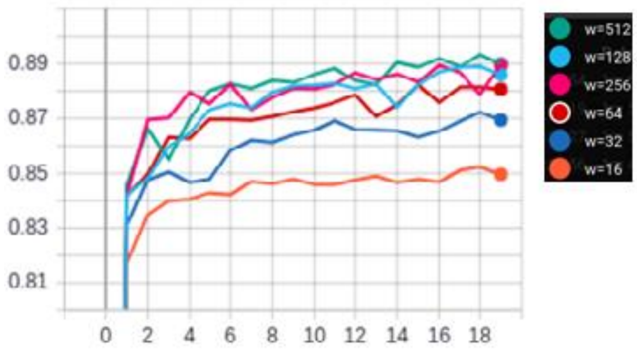


Fig.8: Scaling model with different network width, fashion mnist(test accuracy)

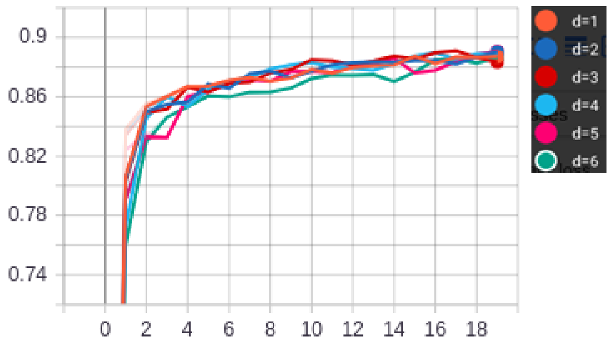


Fig.9: Scaling model with different network depth, fashion mnist (train accuracy)

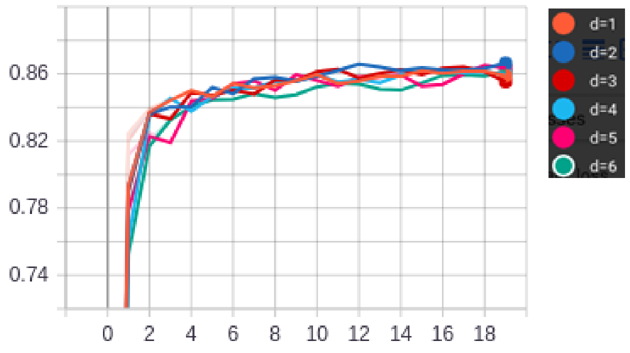


Fig.10: Scaling model with different network depth, fashion mnist (test accuracy)

A.2. Experiment on cifar10:

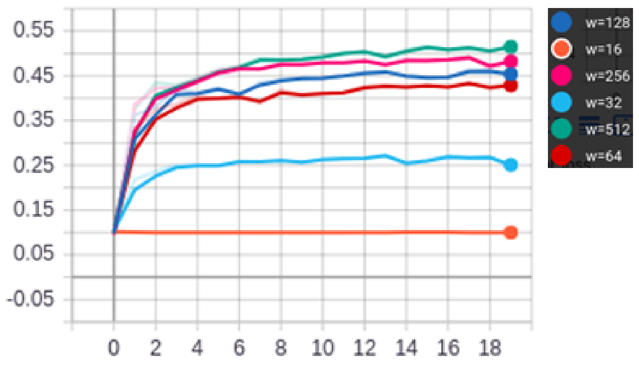


Fig.11: Scaling model with different network width, Cifar_10(train accuracy)

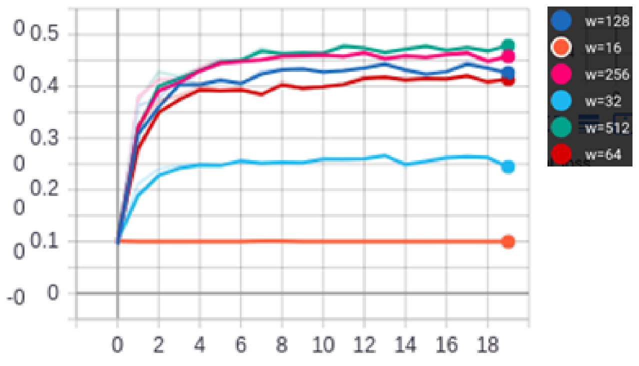


Fig.12: Scaling model with different network width, Cifar_10(test accuracy)

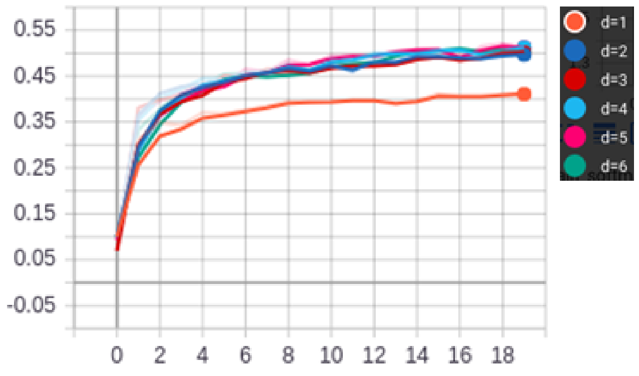


Fig.13: Scaling model with different network depth, Cifar_10 (train accuracy)

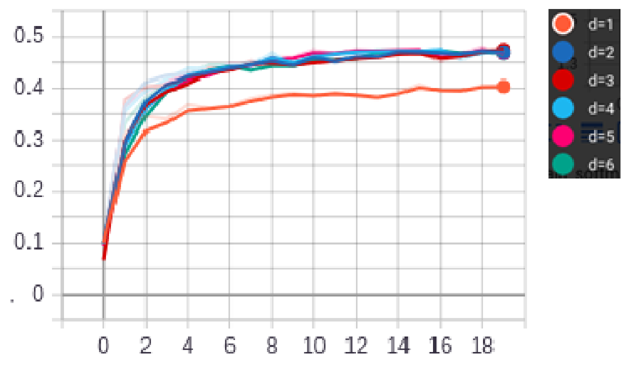


Fig.14: Scaling model with different network depth, Cifar_10 (train accuracy)

B. *Convolutional Neural Networks (CNN):*

With the same setting as MLP, we fix kernel size to 3x3 for all our experiments on CNNs. First, to investigate width effect, we use two conv layers with the same number of filters followed by a global average pooling; then we scale the number of filters. To investigate depth effect, we use a block of two conv layers with 32 filters, each followed by a max pooling layer; then we stack this block to build a deeper network.

B.1. Experiment on Fashion mnist:

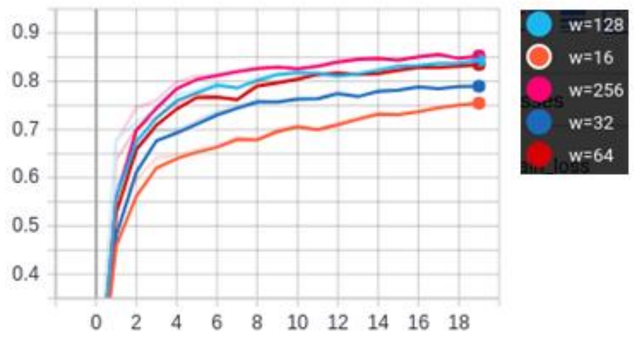


Fig.15: Scaling model with different network width, fashion mnist (train accuracy)

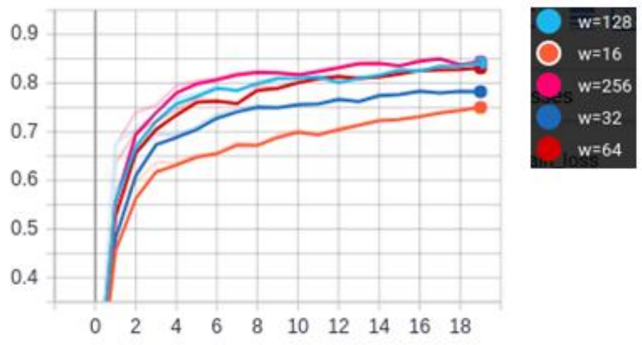


Fig.16: Scaling model with different network depth, fashion mnist (train accuracy)

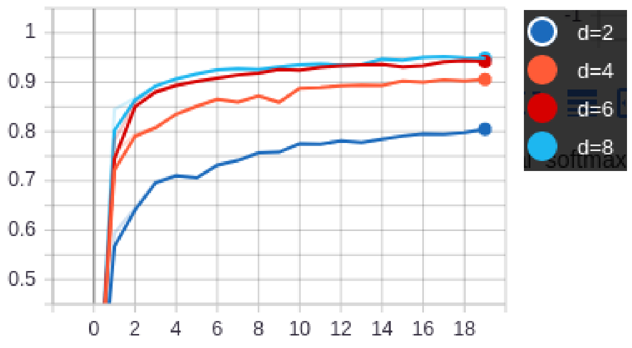


Fig.17: Scaling model with different network depth, fashion mnist (train accuracy)

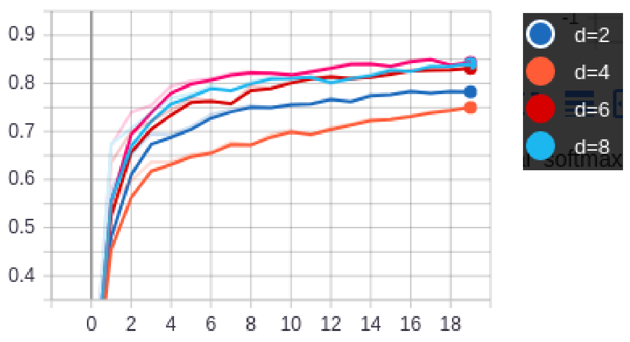


Fig.18: Scaling model with different network depth, fashion mnist (train accuracy)

B.2. Experiment on Cifar_10:

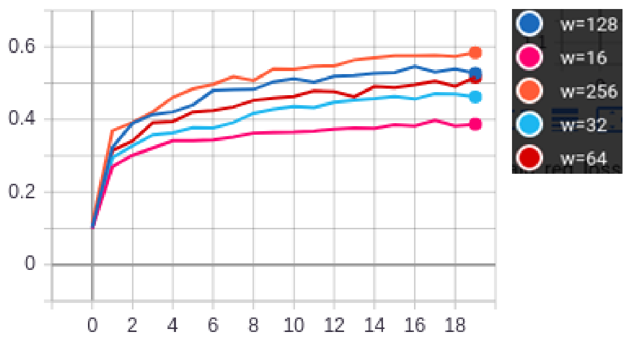


Fig.19: Scaling model with different network width, Cifar_10(train accuracy)

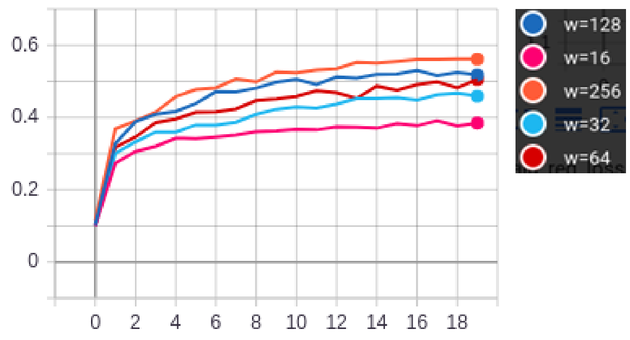


Fig.20: Scaling model with different network width, Cifar_10(test accuracy)

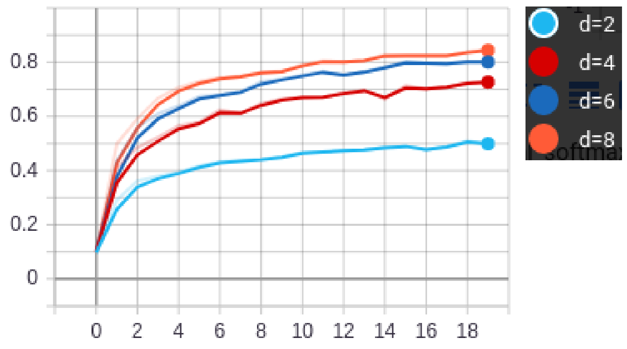


Fig.21: Scaling model with different network depth, Cifar_10 (train accuracy)

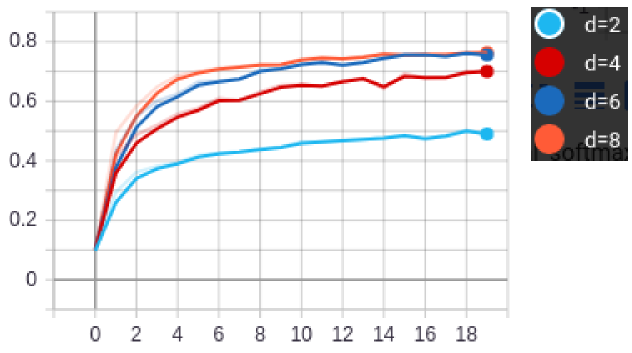
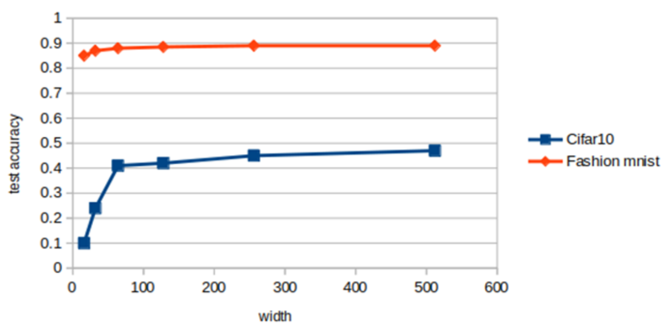


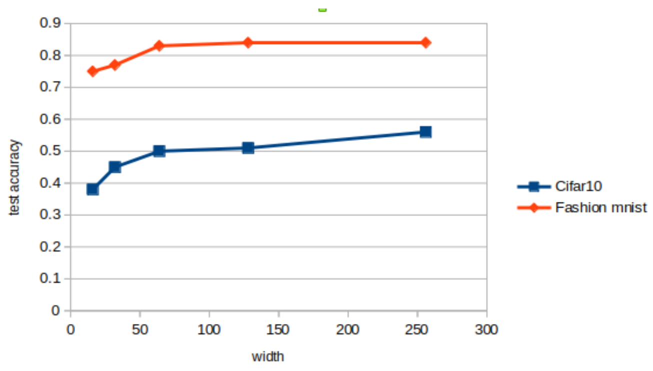
Fig.22: Scaling model with different network depth, Cifar_10 (train accuracy)

C. *Observations:*

Observation 1: Wider networks seem to be able to capture more fine-grained features and are easier to train. However, extremely wide but shallow networks tend to have difficulties in capturing higher level features. Our empirical results on both MLP and CNN in figure 9, Show that accuracy quickly saturates when networks become much wider.



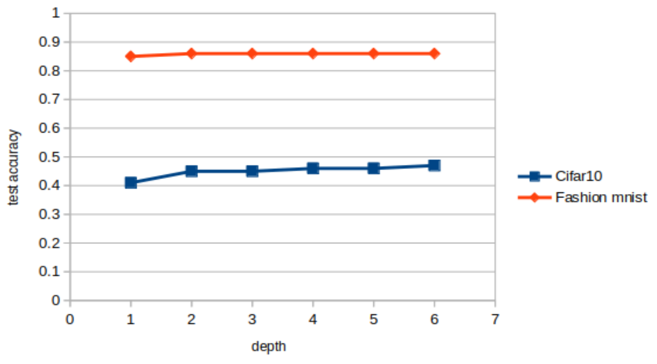
(a)



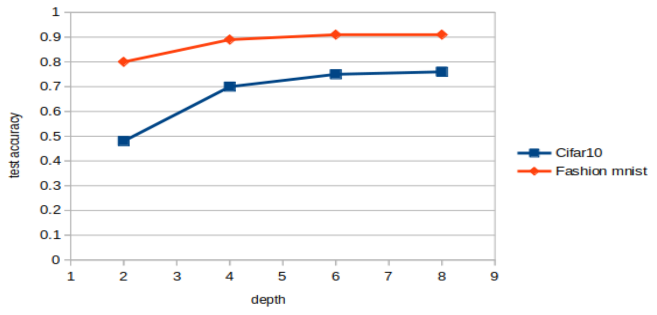
(B)

Fig.23: Scaling network width, (a) MLP test accuracy, (b) CNN test accuracy

Observation 2: Scaling depth is the most common way used in many CNNs. The intuition is that deeper networks can capture richer and more complex features and generalize well on new tasks. However, deeper networks are also more difficult to train due to vanishing gradients. Also, in fully connected structures, deep networks saturate much faster. Figure 10 Shows our empirical results on both MLP and CNN.



(a)



(B)

Fig.24: Scaling network depth, (a) MLP test accuracy, (b) CNN test accuracy

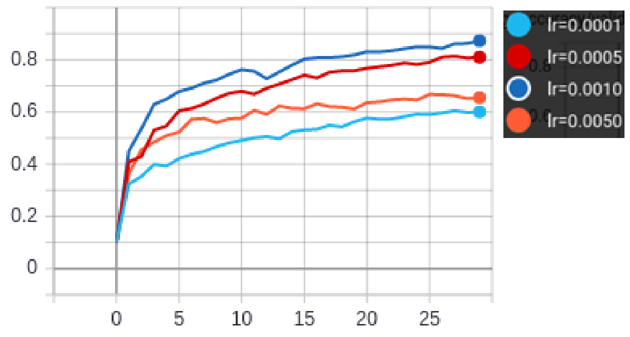
D. *Hyper Parameters:*

In this section, we study different hyper parameters. To do that, we choose a common CNN architecture and we experiment it on only cifar10 dataset. The reason of our choices is that CNNs have less parameters to train and cifar10 is complicated enough to support our results. Also, we study each of hyper parameters in isolation; it means that we fix other parameters to the default values. To conduct our experiments, we use a six-layer CNN as follow:

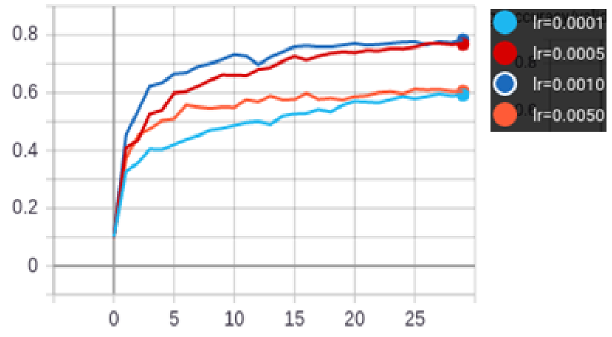
Conv2D(filters=16,kernel_size=3, strides=1,padding='same', activation='relu')
Conv2D(filters=16,kernel_size=3, strides=1,padding='same', activation='relu')
MaxPooling2D(pool_size=3, strides=2)
Conv2D(filters=32,kernel_size=3, strides=1,padding='same', activation='relu')
Conv2D(filters=32,kernel_size=3, strides=1,padding='same', activation='relu')
MaxPooling2D(pool_size=3, strides=2)
Conv2D(filters=64,kernel_size=3, strides=1,padding='same', activation='relu')
Conv2D(filters=64,kernel_size=3, strides=1,padding='same', activation='relu')
GlobalAveragePooling2D()
Dense(units=num_classes)

D.1. learning rate:

To investigate on learning rate effects, we use batch size 64 and Adam optimizer for all our experiments. In low learning rates, the model converges to optimal point slowly and we can avoid over-fitting. By increasing learning rate, the model converges faster but also over-fits faster and there is a possibility that the model passes the optimal point or oscillates around it. Figure 11 shows learning rate effects.



(a)

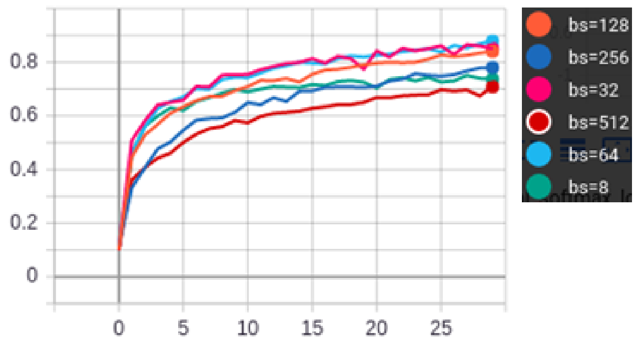


(b)

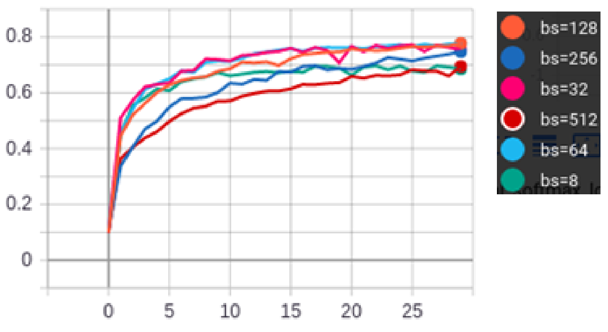
Fig.25: Training with different learning rates, (a) train accuracy, (b) test accuracy

D.2. Batch Size:

Due to the stochastic optimization algorithms that we use in training process, batch size selection is very important. If batch size is small, the network converges faster but never reaches the optimal point and oscillates around it. For larger batch size, the model converges much slower but never misses the optimal point. Another effect of batch size is that due to batch operations in cpu or gpu, bigger batch sizes run much faster. Figure 12 shows batch size effect.



(a)

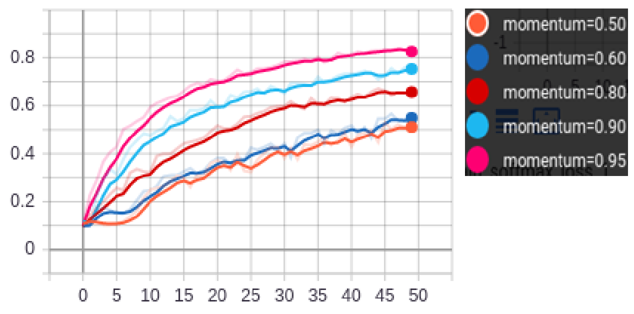


(b)

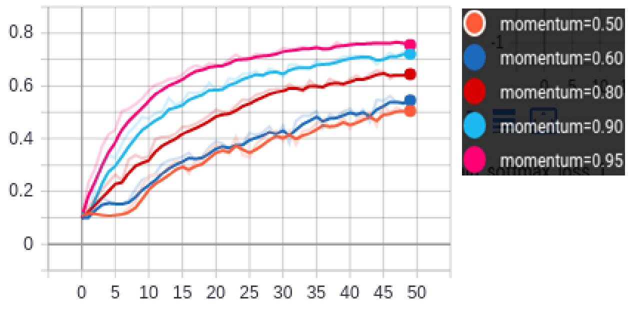
Fig.26: Training with different batch size, (a) train accuracy, (b) test accuracy

D.3. Momentum:

In addition to learning rate and batch size, momentum is another important factor in stochastic algorithms. Momentum keeps the direction of gradients from sudden changes. Low momentums have more stochastic behavior. To investigate on momentum, we use momentum optimizer. Figure 13 shows the momentum effect.



(a)

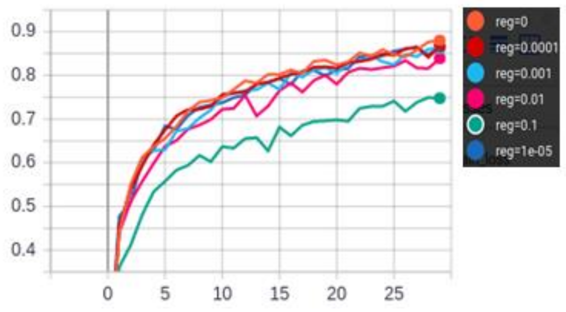


(b)

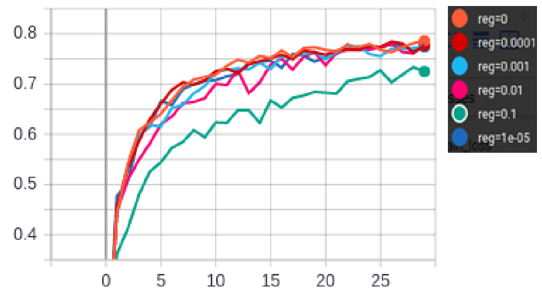
Fig.27: Training with different momentum, (a) train accuracy, (b) test accuracy

D.4. Regularization:

To avoid over-fitting, a common strategy is to use regularization. Common regularizations are L2 norm and L1 norm. In this experiment, we use only L2 norm regularization, also known as ridge regression. The intuition of L2 norm is that it ignores noise features and only concentrates on important features. This leads to small weights. Figure 14 Shows different regularization coefficient effects.



(a)

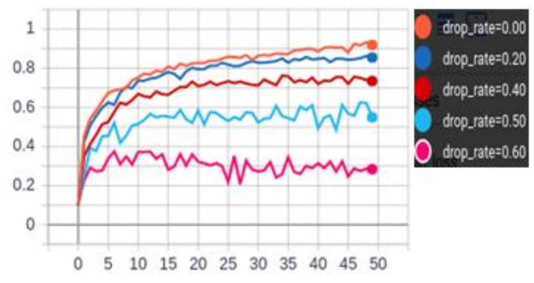


(b)

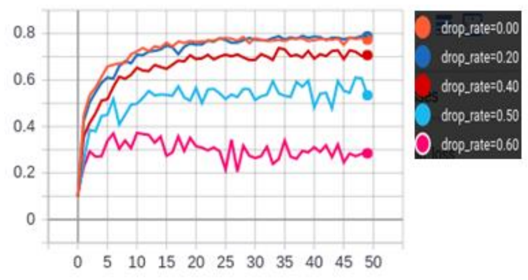
Fig.28: Training with different regularization coefficient, (a) train accuracy, (b) test accuracy

D.4. Dropout:

Another common strategy to avoid over-fitting is to use dropout between network layers. dropout can be thought as a method of making bagging practical for ensembles of very many large neural networks or can be thought as feature occlusion. We use dropout after each conv layer. Dropout after conv layers causes occlusion in channels. and if used after dense layers, it causes occlusion in units. When we increase drop rate, train accuracy decreases but at some point, test accuracy increases. It's worth mentioning that if we use dropout, we need to make the network much wider to achieve a good performance. Figure 15 Shows different drop rate effect in our model.



(a)



(b)

Fig.29: 15. Training with different drop rate, (a) train accuracy, (b) test accuracy