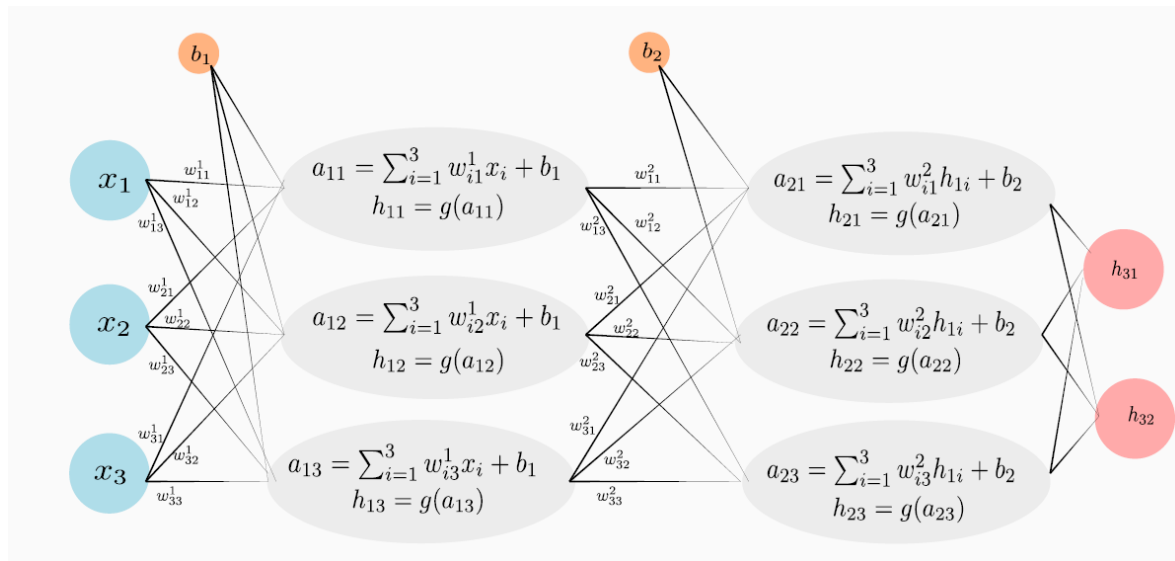


Computer Vision Assignment 3

Image Classification using CNN

By- Apoorva Srivastava (2019702014)

Neural Network:



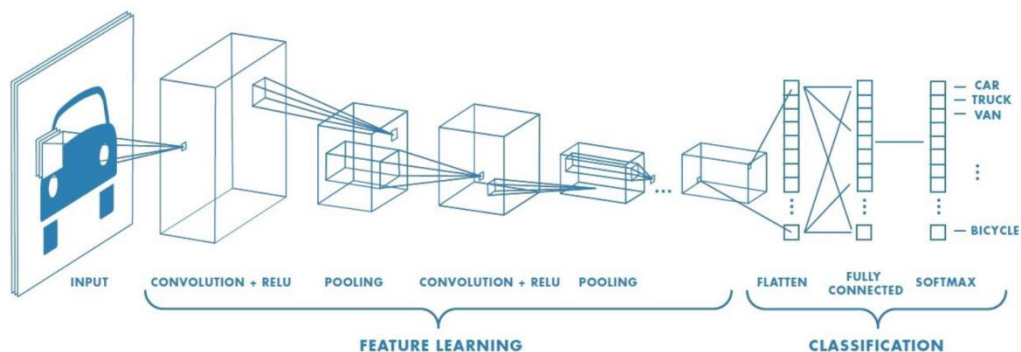
A neural network is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes. Thus a neural network is either a biological neural network, made up of real biological neurons, or an artificial neural network, for solving artificial intelligence (AI) problems. The connections of the biological neuron are modeled as weights. A positive weight reflects an excitatory connection, while negative values mean inhibitory connections. All inputs are modified by a weight and summed. This activity is referred to as a linear combination. Finally, an activation function controls the amplitude of the output.

These artificial networks may be used for predictive modeling, adaptive control and applications where they can be trained via a dataset. Self-learning resulting from experience can occur within networks, which can derive conclusions from a complex and seemingly unrelated set of information.

Convolutional Neural Network:

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.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.



Problem Statement: Train a CNN classifier for single class classification using CIFAR10 dataset.

Cifar10 Dataset: The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Here are the classes in the dataset, as well as 10 random images from each class:

'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'

Implementation Details:

Keras from TensorFlow is used to implement the CNN in python language. The evaluation measure is percentage accuracy on validation set. Multiple models have been trained by varying hyper-parameters to obtain maximum accuracy.

Code:

Code has been divided in 2 parts training and testing. Testing is done on images downloaded from Google Images. In the code first the dataset is loaded and is split into Testing and Training set. All the hyper-parameters were initialized. Then a graph model was created with the help of keras. Then by choosing proper optimizer and the loss functions the model was trained and accuracy for each epoch was plotted for both training and validation set. Loss function value decrease was also plotted with Epoch.

Testing code is written in a separate file which loads the trained models and predicts the output.

Classification Code:

```
import keras                                     #importing keras library
from keras.datasets import cifar10               #importing dataset cifar10 from keras dataset
from keras.models import Sequential             #importing model for sequential neural network structure
                                                #which allows addition of layers in the network
from keras.layers import Dense, Dropout, Activation, Flatten #keras has multiple definitions of
from keras.layers import Conv2D, MaxPooling2D    #standard layers to be used while
                                                #constructing a network they are called as a function

import os
import matplotlib.pyplot as plt

batch_size = 64                                #generally chosen in terms of 32,64,128.
                                                #Represents the number of samples after which weights are updated
num_classes = 10                               #the categories in which the data has to be classified
epochs = 5                                     #Epochs is the number of times whole training data is passed to the deep net
save_dir = os.path.join(os.getcwd(), 'saved_models') #saving the model in specified folder
model_name = 'model_lr0.001.h5'                #final name of the model

# The data, split between train and test sets:
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
# print('x_train shape:', x_train.shape)
# print(x_train.shape[0], 'train samples')
# print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
# Doing this only one bit will be high out of 10 bits in the output of 10x1 vector
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()                            #defining the model to be sequential
#model structure can be seen in model.summary() in the cell below
```

```

model = Sequential()      #defining the model to be sequential
#model structure can be seen in model.summary() in the cell below
#relu is the activation function
#same padding ensures same size
#Dropout layers prevent overfitting
#Conv2D layer provides 2D convolution
#Dense layer deploys fully connected layers
model.add(Conv2D(64, (3, 3), padding='same',
                input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.35))

model.add(Conv2D(32, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.35))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))

checkpoint_path = "train_ckpt/cp.ckpt"
# Create a callback that saves the model's weights every 10 epochs as checkpoints

```

```

cp_callback = keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_path,
    verbose=0,
    save_weights_only=False,
    period=10)

# initiate Adam optimizer learning rate 0.001
opt=keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=False)

# finalizing the model by defining loss, optimization and the evaluation metric
history = model.compile(loss='categorical_crossentropy',
                        optimizer=opt,
                        metrics=['accuracy'])

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

#training the model
model.fit(x_train, y_train,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=(x_test, y_test),
        shuffle=True)

# Save model and weights
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
model_path = os.path.join(save_dir, model_name)

```

```

# Save model and weights
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
model_path = os.path.join(save_dir, model_name)
model.save(model_path)
print('Saved trained model at %s ' % model_path)

# Score trained model
scores = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])

# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

```

Code for Prediction:

```

import numpy as np
import tensorflow as tf
import keras
from keras.preprocessing import image
from keras.models import load_model
model = load_model('./saved_models/copy1_model.h5')
# Give the link of the image here to test
test_image1 = image.load_img('./Images/32.jpg', target_size = (32,32))
test_image = image.img_to_array(test_image1)
test_image = np.expand_dims(test_image, axis = 0)
result = model.predict(test_image)
print(result)
if result[0][0]==1:
    print("Aeroplane")
elif result[0][1]==1:
    print('Automobile')
elif result[0][2]==1:
    print('Bird')
elif result[0][3]==1:
    print('Cat')
elif result[0][4]==1:
    print('Deer')
elif result[0][5]==1:
    print('Dog')
elif result[0][6]==1:
    print('Frog')
elif result[0][7]==1:
    print('Horse')
elif result[0][8]==1:
    print('Ship')
elif result[0][9]==1:
    print('Truck')
else:
    print('Error')

```

Models trained:

Note: All the models which were trained for different hyper-parameter optimization are shared at the following link-

https://iitaphyd-my.sharepoint.com/:f:/g/personal/apoorva_srivastava_research_iit_ac_in/EhGNGBKabyDq7SttAiAo6kBIPDhpuTxCNN54F_qcqOUUw?e=5YccCc

Model: "model1.h5"

Epoch=100, Batch Size=32, Optimizer=RMSProp, Loss Function=Cross Entropy, Filter Size=3x3

Test loss: 0.6459939745903015, Test accuracy: 79.14000153541565%, Drop Out Ratio=0.25, Initialization=Xavier's Initialization

Learning rate=0.0001, Decay=1e-6, Activation function=Relu and Softmax, Padding=Zero Padding

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 32, 32, 32)	896
activation_13 (Activation)	(None, 32, 32, 32)	0
conv2d_10 (Conv2D)	(None, 30, 30, 32)	9248
activation_14 (Activation)	(None, 30, 30, 32)	0
max_pooling2d_5 (MaxPooling2D)	(None, 15, 15, 32)	0
dropout_7 (Dropout)	(None, 15, 15, 32)	0
conv2d_11 (Conv2D)	(None, 15, 15, 64)	18496
activation_15 (Activation)	(None, 15, 15, 64)	0
conv2d_12 (Conv2D)	(None, 13, 13, 64)	36928
activation_16 (Activation)	(None, 13, 13, 64)	0
max_pooling2d_6 (MaxPooling2D)	(None, 6, 6, 64)	0
dropout_8 (Dropout)	(None, 6, 6, 64)	0
flatten_3 (Flatten)	(None, 2304)	0
dense_5 (Dense)	(None, 512)	1180160
activation_17 (Activation)	(None, 512)	0
dropout_9 (Dropout)	(None, 512)	0
dense_6 (Dense)	(None, 10)	5130
activation_18 (Activation)	(None, 10)	0
Total params: 1,250,858		
Trainable params: 1,250,858		
Non-trainable params: 0		

Now we will try to improve the above network in terms of size while trying to achieve same or higher accuracy.

Model: "model2.h5"

Epoch=100, Batch Size=64,Optimizer=Adam, Loss Function=Cross Entropy, Filter Size=3x3

**Test loss: 0.612385363303,Test accuracy: 79.5608865783541565%,Drop Out Ratio=0.35,
Initialization=Xavier's Initialization**

Learning rate=0.001,Decay=1e-6,Activation function=Relu and Softmax, Padding=Zero Padding

Layer (type)	Output Shape	Param #
conv2d_13 (Conv2D)	(None, 32, 32, 64)	1792
activation_19 (Activation)	(None, 32, 32, 64)	0
conv2d_14 (Conv2D)	(None, 30, 30, 64)	36928
activation_20 (Activation)	(None, 30, 30, 64)	0
max_pooling2d_7 (MaxPooling2D)	(None, 15, 15, 64)	0
dropout_10 (Dropout)	(None, 15, 15, 64)	0
conv2d_15 (Conv2D)	(None, 15, 15, 32)	18464
activation_21 (Activation)	(None, 15, 15, 32)	0
conv2d_16 (Conv2D)	(None, 13, 13, 32)	9248
activation_22 (Activation)	(None, 13, 13, 32)	0
max_pooling2d_8 (MaxPooling2D)	(None, 6, 6, 32)	0
dropout_11 (Dropout)	(None, 6, 6, 32)	0
flatten_4 (Flatten)	(None, 1152)	0
dense_7 (Dense)	(None, 512)	590336
activation_23 (Activation)	(None, 512)	0
dropout_12 (Dropout)	(None, 512)	0
dense_8 (Dense)	(None, 10)	5130
activation_24 (Activation)	(None, 10)	0
Total params: 661,898		
Trainable params: 661,898		
Non-trainable params: 0		

Hence almost 50% of the model size was reduced by making the model structure better and using better Optimizers, Dropout Probability, Batch Size and lessening the number of Epochs by 50%.

Loss Curve & Accuracy Curve while Hyperparameter Optimization:

Model2 will be used to compare the hyperparameter variation and their effect on Loss and accuracy. These Curves are drawn to compare the hyperparameters for 5 epochs, batch size 512

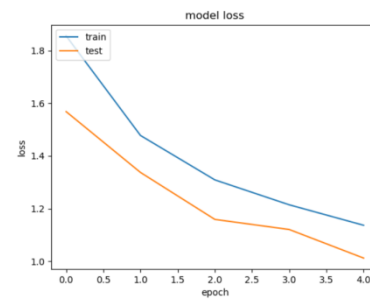
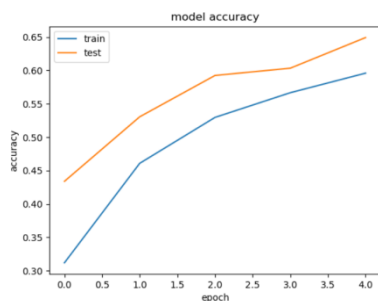
Learning Rate: Learning rate is a hyper-parameter which governs that how much portion of the gradient or the update quantity should be added while making update in the parameters after each batch. If learning rate is kept very high then we may overshoot the minima and if learning rate is kept much low then training process becomes very time-taking.

Various Learning rates from 0.001 to .0001 were tried with Optimizer = Adam, Loss Function= Cross Entropy Model Structure= Model2, Decay=1e-6, Activation function=Relu and Softmax, Zero Padding

Below are the Loss Curves for each learning rate:

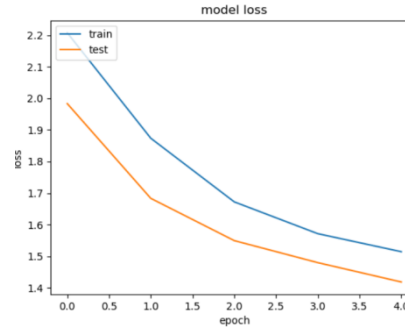
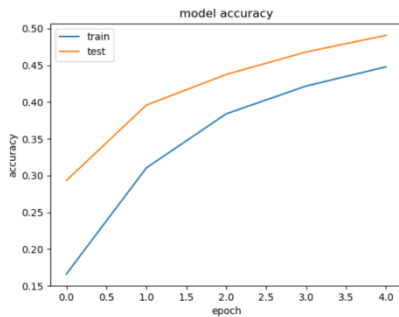
For Learning rate=0.001, loss=1.41 and accuracy=64%

Note: Accuracy is low due to smaller number of epochs due to lesser computation power and limited time.



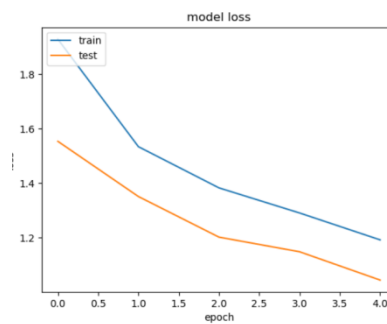
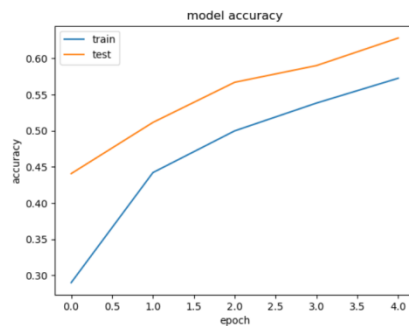
For Learning rate=0.0001 model accuracy 49%, Loss=1.41

Note: Accuracy is low due to smaller number of epochs due to lesser computation power and limited time.



For Learning Rate=0.005, Accuracy= 62% Loss=1.02

Note: Accuracy is low due to smaller number of epochs due to lesser computation power and limited time.



INFERENCE about Learning Rate: As mentioned above it can be seen that if learning rate is reduced then learning process decreases so if number of epochs is lesser than the required then lower learning rate model will achieve lesser accuracy. But however if we increase the learning rate beyond optimal value then in the same number epoch accuracy will decrease as due to higher learning rate the gradients overshoot. Hence learning rate=0.001 is the best value so far for the given other parameters and number of epochs.

Initialization: Two different initializations were used Xavier Normal Initialization and Variance Scaling. They were tried with Optimizer = Adam, Loss Function=Cross Entropy Model Structure= Model2, Decay=1e-6, Activation function=Relu and Softmax, Zero Padding, Learning rate=0.001

Glorot normal initializer

It is also called Xavier normal initializer. It draws samples from a truncated normal distribution centered on 0 with stddev = $\sqrt{2 / (\text{fan_in} + \text{fan_out})}$ where fan_in is the number of input units in the weight tensor and fan_out is the number of output units in the weight tensor.

Variance Scaling:

Initializer capable of adapting its scale to the shape of weights.

With distribution="normal", samples are drawn from a truncated normal distribution centered on zero, with stddev = $\sqrt{\text{scale} / n}$ where n is:

- number of input units in the weight tensor, if mode = "fan_in"
- number of output units, if mode = "fan_out"
- average of the numbers of input and output units, if mode = "fan_avg"

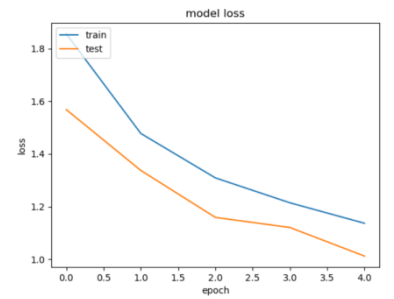
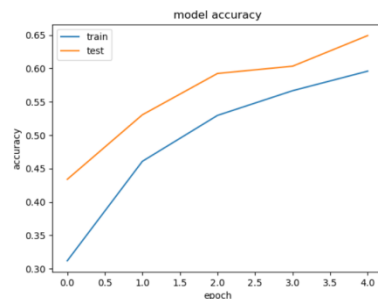
With distribution="uniform", samples are drawn from a uniform distribution within [-limit, limit], with limit = $\sqrt{3 * \text{scale} / n}$.

Following is the Loss curve for different Initializations of weights:

Xavier Initialization:

Loss=1.41 and accuracy=64.03%

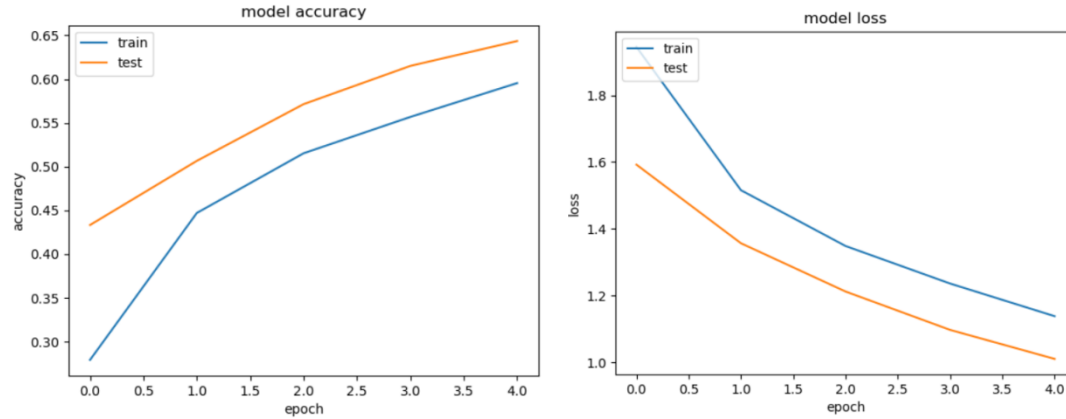
Note: Accuracy is low due to smaller number of epochs due to lesser computation power & limited time.



Variance Scaling Initializer:

Loss=1.01 and Accuracy= 64.01

Note: Accuracy is low due to smaller number of epochs due to lesser computation power & limited time.



Inference about Initializers:

Initialization of the weights play an important role as in the absence of proper initialization we face the problem of vanishing gradients and saturation of the gradients. However, the two initializations used here are well researched initializations and for the given data they give equivalent results. So we go by the default Xavier Initializer in keras.

Optimizer: RMS-Prop and Adam are compared with Loss Function=Cross Entropy Model Structure=Model2, Decay=1e-6, Activation function=Relu and Softmax, Zero Padding, Learnin Rate=0.001

RMS-Prop: RMS prop is a variation in Gradient Descent algorithm. RMS Prop tries to dampen the oscillations, but in a different way than momentum. RMS prop also takes away the need to adjust learning rate, and does it automatically. More so, RMS Prop choses a different learning rate for each parameter. In RMS prop, each update is done according to the equations described below. This update is done separately for each parameter.

For each Parameter w^j
(j subscript dropped for clarity)

$$\nu_t = \rho \nu_{t-1} + (1 - \rho) * g_t^2$$

$$\Delta \omega_t = - \frac{\eta}{\sqrt{\nu_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta \omega_t$$

η : Initial Learning rate
 ν_t : Exponential Average of squares of gradients
 g_t : Gradient at time t along ω^j

Adam Optimizer:

RMSProp and Momentum take contrasting approaches. While momentum accelerates our search in direction of minima, RMSProp impedes our search in direction of oscillations. **Adam** or **Adaptive Moment Optimization** algorithms combines the heuristics of both Momentum and RMSProp.

For each Parameter w^j
(j subscript dropped for clarity)

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

η : Initial Learning rate

g_t : Gradient at time t along ω^j

ν_t : Exponential Average of gradients along ω_j

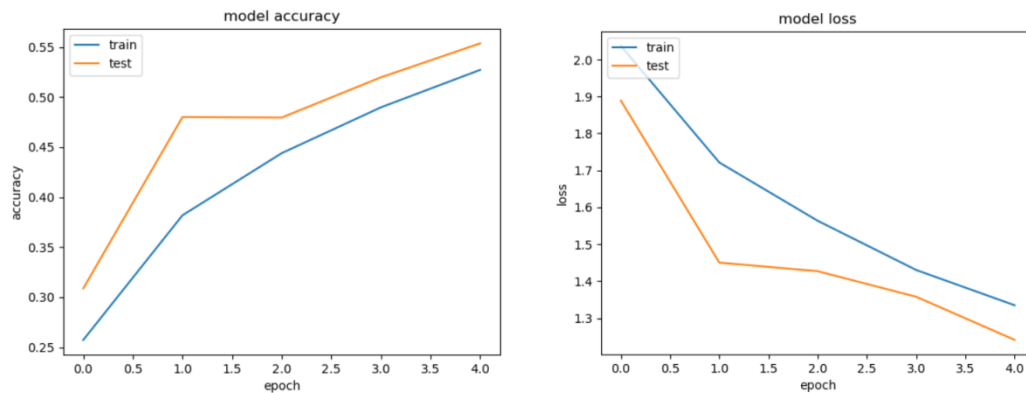
s_t : Exponential Average of squares of gradients along ω_j

β_1, β_2 : Hyperparameters

Curve for RMS Prop:

Loss=1.44 and accuracy=55.3%

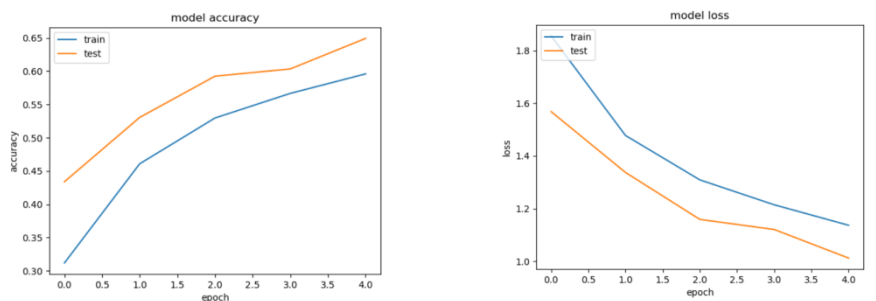
Note: Accuracy is low due to smaller number of epochs due to lesser computation power and limited time.



Curve for Adam Optimizer:

Loss=1.41 and accuracy=64%

Note: Accuracy is low due to smaller number of epochs due to lesser computation power and limited time.



Inference about Optimizer:

As Adam is known to be better optimizer than RMS Prop due to utilization of both kind of momentum hence that is visible in the curves above as Adam gives better accuracy and Loss Values than RMS Prop.

Dropout probability:

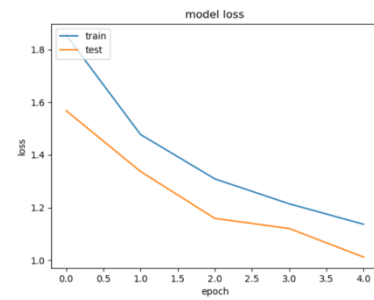
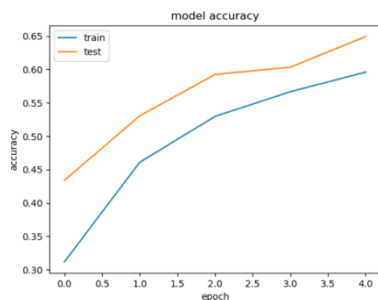
Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel. During training, some number of layer outputs are randomly ignored or “*dropped out.*” This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different “*view*” of the configured layer.

Two different Dropout probabilities .35 and 0.1 were used. They were tried with Optimizer = Adam, Loss Function=Cross Entropy Model Structure= Model2, Decay=1e-6, Activation function=Relu and Softmax, Zero Padding, Learning rate=0.001

Curve for Dropout=0.35:

Loss=1.41 and accuracy=64%

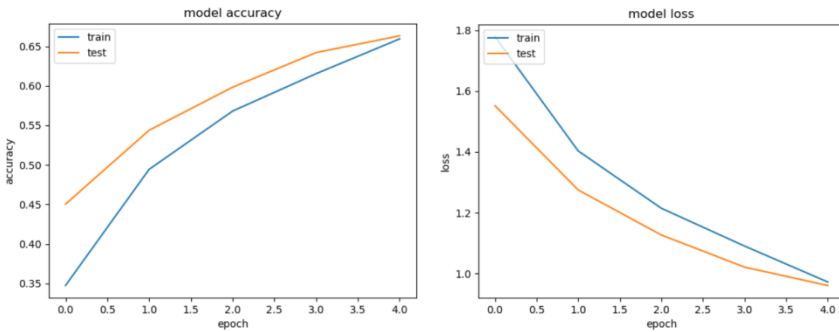
Note: Accuracy is low due to smaller number of epochs due to lesser computation power and limited time.



Curve for Dropout=0.1:

Loss=0.96 and accuracy= 66%

Note: Accuracy is low due to smaller number of epochs due to lesser computation power and limited time.



Inference about Dropout Probabilities:

Dropout Probability helps in removing overfitting but if the dropout rate is very high then learning for the network becomes tough as the network capacity reduces while for a low drop out overfitting occurs. From the above experiments ideal value for dropout probability comes to be 0.1.

Loss Functions:

Loss function helps in optimizing the parameters of the neural networks. Our objective is to minimize the loss for a neural network by optimizing its parameters(weights). The loss is calculated using loss function by matching the target(actual) value and predicted value by a neural network. Then we use the gradient descent method to update the weights of the neural network such that the loss is minimized. This is how we train a neural network.

Two different Loss Functions Cross Entropy and KL Divergence were used. They were tried with Optimizer = Adam, Loss Function=Cross Entropy Model Structure= Model2, Decay=1e-6, Activation function=Relu and Softmax, Zero Padding, Learning rate=0.001

Cross Entropy Loss Function:

Cross-entropy is the default loss function to use for multi-class classification problems. In this case, it is intended for use with multi-class classification where the target values are in the set $\{0, 1, 3, \dots, n\}$, where each class is assigned a unique integer value. Mathematically, it is the preferred loss function under the inference framework of maximum likelihood. It is the loss function to be evaluated first and only changed if you have a good reason. Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for all classes in the problem. The score is minimized and a perfect cross-entropy value is 0.

KL Divergence Loss Function:

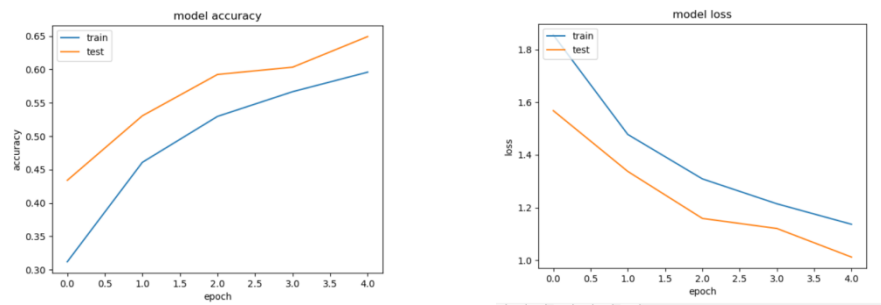
Kullback Leibler Divergence, or KL Divergence for short, is a measure of how one probability distribution differs from a baseline distribution. A KL divergence loss of 0 suggests the distributions are identical. In practice, the behavior of KL Divergence is very similar to cross-entropy. It calculates how much

information is lost (in terms of bits) if the predicted probability distribution is used to approximate the desired target probability distribution.

Curves for Cross Entropy Loss Function:

Loss=1.41 and accuracy=64.03%

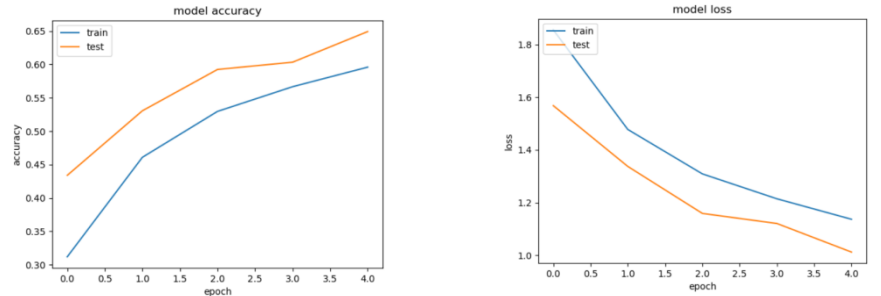
Note: Accuracy is low due to smaller number of epochs due to lesser computation power & limited time.



Curves for KL Divergence Loss Function:

Loss=1.39 and accuracy=63.97%

Note: Accuracy is low due to smaller number of epochs due to lesser computation power & limited time.



Inference about Loss Function:

As it can be seen the values are similar for both the loss functions for multi-class classification. As such, the KL divergence loss function is more commonly used when using models that learn to approximate a more complex function than simply multi-class classification, such as in the case of an autoencoder used for learning a dense feature representation under a model that must reconstruct the original input. In this case, KL divergence loss would be preferred. Nevertheless, it can be used for multi-class classification, in which case it is functionally equivalent to multi-class cross-entropy.

Epochs & Batch Size

Epochs is the number of times the whole training data is passed through the network and batch size represent the number of samples after which the parameters of the network are updated. While training the classifier for different models following was interpreted about the Number of Epochs and Batch Size-

Multiple numbers of Epochs ensure that the network does not learn any particular order about the presentation of the samples in multiple batches. This method also helps to deal the dearth of the Data and explore the Loss Function with multiple vies to find the global minima. However,if the accuracy stops increasing after certain number of epochs then that means the net is overfitting and either regularization parameters should be increased or number of epochs should be reduced.

Batch Size determines how much the value of each sample in the data is. If the batch size is kept small then variation of weights with respect to individual sample increases but it may cause overfitting and increased computational time. However, if the batch size is taken to be large then net becomes less sensitive to individual data and may loose some information. Batch size chosen is generally 32,64,128,512 etc.

After observation, for the given data and model with all other parameters optimal value of the Number of Epochs Is 75 and Batch Size is 64.





Note: All the models which were trained for different hyper-parameter optimization are shared at the following link-





https://iiitaphyd-my.sharepoint.com/:f:/g/personal/apoorva_srivastava_research_iiit_ac_in/EhGNGBKabypDq7SttAiAo6kBIPDhpuTxCNN54F_qcqOUUw?e=5YccCc





Results of prediction on unknown data by the Model2:

All the Test Images are shared at the following folder: https://iiitaphyd-my.sharepoint.com/:f:/g/personal/apoorva_srivastava_research_iiit_ac_in/EiCSJz99WylHuZPdCGwSt80Be45HuT4RTFym2DV1RccGWA?e=pNG95s

S.No.	Image	Prediction
1		Automobile

2		Aeroplane
3		Horse
4		Aeroplane
5		Bird

6		Dog
7		Horse
8		Horse
9		Cat

10		Dog
11		Horse
12		Ship
13		Frog

14		Truck
15		Sheep