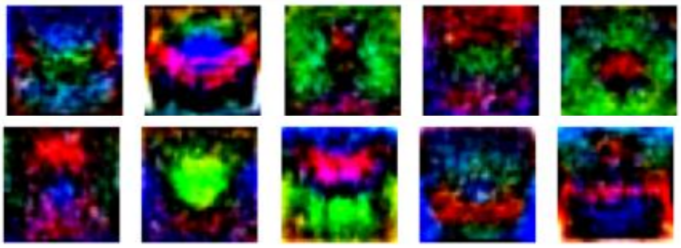
Assignment 4 **180427N**

1. https://github.com/Nirajkanth/Digital-image-processing-and-computer-vision

a) Gradient decent

```
iterations = 300
lr = 1.4e-2
lr_decay= 0.999
reg = 5e-5
loss_history = []
train_acc_history = []
val_acc_history = []
seed = 0
rng = np.random.default_rng(seed=seed)

for t in range(iterations):
    indices = np.arange(Ntr)
    rng.shuffle(indices)
    # Forward pass
    x = x_train[indices]
    y = y_train[indices]
    y_pred = x.dot(w1) + b1
    loss = 1./batch_size*np.square(y_pred - y).sum() + reg*np.sum(w1*w1)
    loss_history.append(loss)

    if t % 10 == 0:
        print("Loss after {} iteration {}".format(t, loss))
        train_acc = 1.0 - (1/Ntr)*(np.count_nonzero(
            np.abs(np.argmax(y_pred, axis=1) - np.argmax(y, axis=1))))
        train_acc_history.append(train_acc)
        print("Training accuracy : ",train_acc)
    # Backward pass
    dy_pred = (2.0/batch_size)*(y_pred - y)
    dw1 = x.T.dot(dy_pred)  # D x K
    db1 = dy_pred.sum(axis=0) # 1 x K coloumn wise summation

    w1 = w1 - lr*dw1
    b1 = b1 - lr*db1
    lr = lr*lr_decay
```

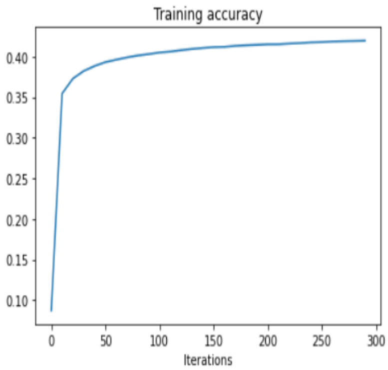
b)

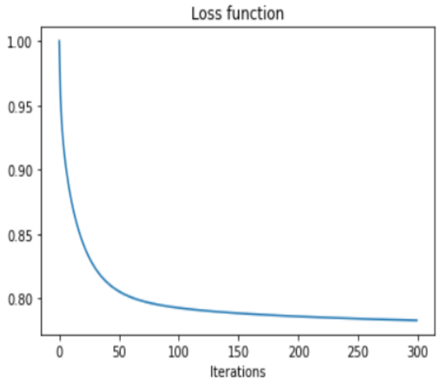


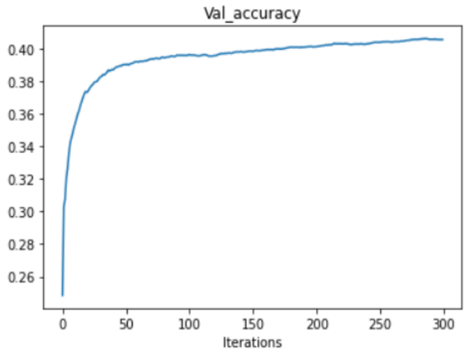
```
Training accuracy :  0.41941999999999999
Test accuracy :  0.40559999999999996
```

These are the 10 images that I obtained through weight matrix. We can see these images are nearly same as our training images (car, areophane, cat etc.). However, the learning of weight matrix is not enough since we iterated only 300 times. We can get more better images by increasing number of iterations.

c) I used initial learning rate as 1.4×10^{-2} and I obtained 0.419 training accuracy and 0.405 testing accuracy as shown in part b. Test accuracy is smaller than training accuracy as we expected.







Training accuracy Loss function Testing accuracy

2. a)

```
H = 200
std = 1e-5
w1 = std*np.random.randn(Din, H)
b1 = np.zeros(H) # raw vector
w2 = std*np.random.randn(H,K)
b2 = np.zeros(K)
print("w1:", w1.shape)
print("b1:", b1.shape)
print("w2:", w2.shape)
print("b2:", b2.shape)
batch_size = Ntr
iterations = 300
lr = 1.4e-2
lr_decay= 0.999
reg = 5e-6
loss_history = []
train_acc_history = []
val_acc_history = []
seed = 0
rng = np.random.default_rng(seed=seed)

for t in range(iterations):
    indices = np.arange(Ntr)
    rng.shuffle(indices)
    # Forward pass
    x = x_train[indices]
    y = y_train[indices]
    h = 1./(1.0 + np.exp(-x.dot(w1) - b1))
    y_pred = h.dot(w2) + b2
    loss = 1./batch_size*np.square(y_pred - y).sum() +
                        reg*(np.sum(w1*w1) + np.sum(w2*w2))
    loss_history.append(loss)
```

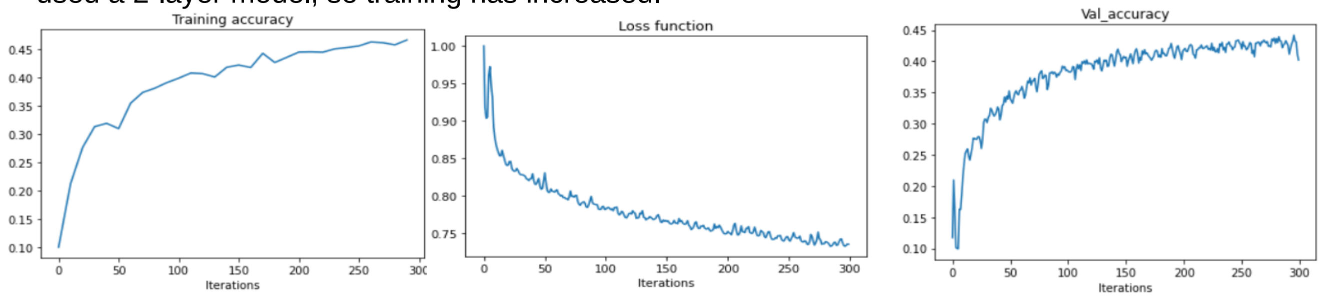
```
if t % 10 == 0:
    print("Loss after {} iteration {}".format(t, loss))
    train_acc = 1.0 - (1/Ntr)*(np.count_nonzero(
        np.argmax(y_pred, axis=1) - np.argmax(y, axis=1)))
    train_acc_history.append(train_acc)
    print("Training accuracy : ",train_acc)
# Backward pass
dy_pred = (2.0/batch_size)*(y_pred - y)
dw2 = h.T.dot(dy_pred) +  reg*w2 # H x K
db2 = dy_pred.sum(axis=0) # 1 x K, coloumn wise summation
dh = dy_pred.dot(w2.T) # Ntr x H
dw1 = x.T.dot(dh*h*(1-h)) + reg*w1
db1 = (dh*h*(1-h)).sum(axis = 0)

w1 = w1 - lr*dw1
b1 = b1 - lr*db1
w2 = w2 - lr*dw2
b2 = b2 - lr*db2
lr = lr*lr_decay
```

b)

Learning rate = 1.4×10^{-2}.

```
Training accuracy :  0.46824
Test accuracy :  0.40559999999999996
```

I used all hyperparameter as same as part 1. So, if we compare part 1 and part 2, we can observe training accuracy has increased but test accuracy remains the same. In part 2 we used a 2-layer model, so training has increased.



3.

a)

```
seed = 0
rng = np.random.default_rng(seed=seed)
for t in range(iterations):
    indices = np.arange(Ntr)
    rng.shuffle(indices)
    x1 = x_train[indices]
    y1 = y_train[indices]
    seed += 1
    loss_1 = []
    for st in range(0,Ntr+1, batch_size):
        # Forward pass
        end = st + batch_size
        x = x1[st:end:,]
        y = y1[st:end:,]
        z = x.dot(w1) + b1
```

```
        # avoiding overflow
        with np.errstate(over='ignore', invalid='ignore'):
              h = np.where(z >= 0,
                  1 / (1 + np.exp(-z)),
                  np.exp(z) / (1 + np.exp(z)))
        y_pred = h.dot(w2) + b2

        loss = 1./batch_size*np.square(y_pred - y).sum() +
                            reg*(np.sum(w1*w1) + np.sum(w2*w2))
        loss_1.append(loss)
        # Backward pass
        dy_pred = (2.0/batch_size)*(y_pred - y)
        dw2 = h.T.dot(dy_pred) +  reg*w2 # H x K
        db2 = dy_pred.sum(axis=0) # 1 x K, coloumn wise summation
        dh = dy_pred.dot(w2.T) # Ntr x H
        dw1 = x.T.dot(dh*h*(1-h)) + reg*w1
        db1 = (dh*h*(1-h)).sum(axis = 0)

        w1 = w1 - lr*dw1
        b1 = b1 - lr*db1
        w2 = w2 - lr*dw2
        b2 = b2 - lr*db2

    lr = lr*lr_decay
    loss_avg = np.average(loss_1)
    loss_history.append(loss_avg)
```

b)

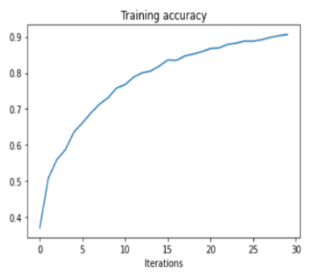
```
Training accuracy :  0.9076
Test accuracy :  0.40659999999999996
```

Learning rate and everything is same as in part 2 except stochastic gradient descent with batch size of 500. This is the result I obtained.

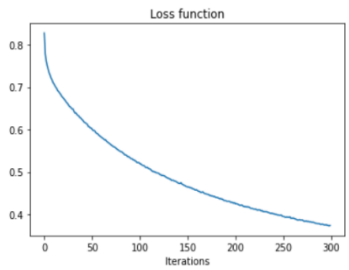
The training accuracy has drastically increased compared to part 2 (0.46 to 0.90). In part 3 the parameters get updated 100 times (50,000 training images and batch size 500) in an epoch but in part 2 parameters get updated only once in an epoch. That means the parameter leaning is very high (well trained) in part3. That is the reason for this increment in training accuracy.

If we observe the test accuracy that is nearly same for both cases (0.4). In part3 training accuracy is very high compared to testing accuracy. This is due to the overfitting of training images; however, we can improve the testing accuracy by tuning regularization parameter or by using optimization techniques. E.g., dropout regularization

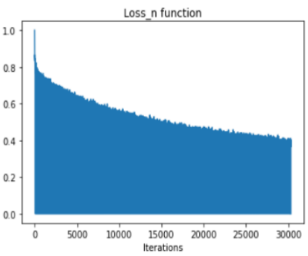
Conclusion: Stochastic gradient descent with a batch size is better for when the training data is large.



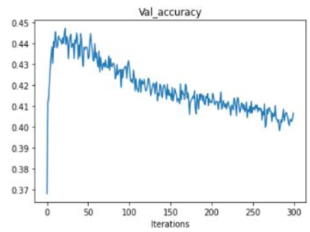
Training accuracy



Average loss



Actual loss(noisy)



Testing accuracy

4.

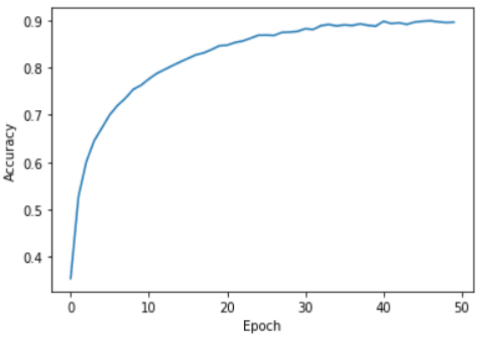
```
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
K = len(np.unique(y_train)) # Classes
Ntr = x_train.shape[0]

Nte = x_test.shape[0]
Din = 3072 # CIFAR10
y_train = tf.keras.utils.to_categorical(y_train, num_classes=K)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=K)
x_train = tf.dtypes.cast(x_train, tf.float32)
x_test = tf.dtypes.cast(x_test, tf.float32)
x_train, x_test = x_train/255., x_test/255.

model = keras.models.Sequential()
model.add(layers.Conv2D(32, (3,3), activation = "relu", input_shape = (32,32,3)))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64, (3,3), activation = "relu"))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64, (3,3), activation = "relu"))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation = "relu"))
model.add(layers.Dense(10))
```

```
model.compile(
    optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum = 0.9),
    loss = tf.keras.losses.CategoricalCrossentropy(from_logits = True),
    metrics= ["accuracy"]
)
print(model.summary())
history = model.fit(x_train, y_train, epochs=50,batch_size=50,
                     validation_data=(x_test, y_test))
test_loss, test_acc = model.evaluate(x_test, y_test, verbose = 2)
```

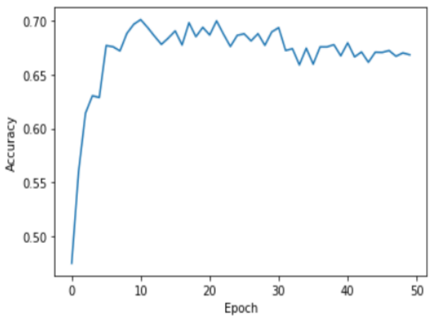
a) Learnable parameters = 73,418

b) Learning rate(initial) = 0.01, Momentum = 0.9

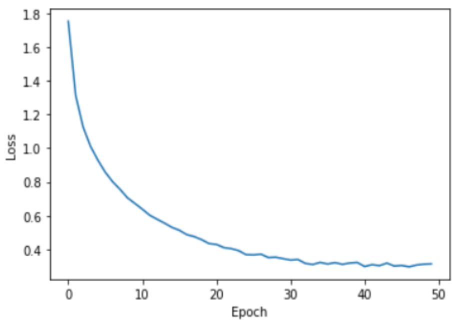
c) Training accuracy = 0.9057, Testing accuracy = 0.6686, Training loss = 0.2805, Testing loss = 1.6917.



Training accuracy



Validation accuracy



Training loss function