Image Classification on the CIFAR-10 Dataset

Shivansh Rastogi
Zyro
Team Gyro

Abstract:

In this project I implemented two machine learning models Logistic Regression and a Deep Learning Model on CIFAR -10 data with CNN. To achieve the goal, I tried to get the best possible accuracy. To fully understand this dataset and the classification problem, I provided a detailed description of task and its issues. In the end, I analyzed the results of our training process. I started with data analysis, CIFAR-10 contains 60000 labeled for 10 classes images 32x32 in size for each RGB channel, train set has 50000 and test set 10000. The categories are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

I. INTRODUCTION

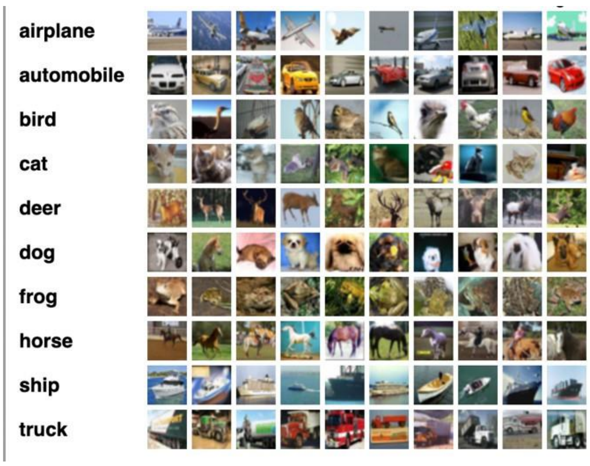
In this section we will introduce the Image Classification problem, which is the task of assigning an input image one label from a fixed set of categories. This is one of the core problems in Computer Vision that, despite its simplicity, has a large variety of practical applications.

Image Classification is a fundamental task that attempts to comprehend an entire image as a whole. The goal is to classify the image by assigning it to a specific label. Typically, Image Classification refers to images in which only one object appears and is analyzed. In contrast, object detection involves both classification and localization tasks, and is used to analyze more realistic cases in which multiple objects may exist in an image.

CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

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.

Train, test data sets are of shape (50000, 32x32) & (10000, 32x32).

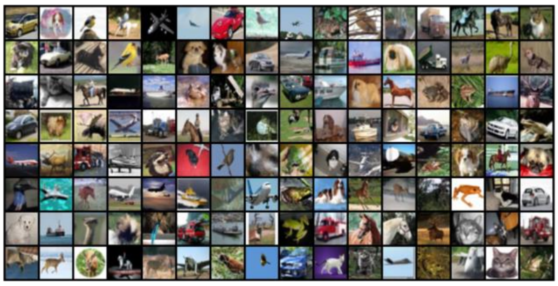


II. DATA ANALYSIS and MODELING

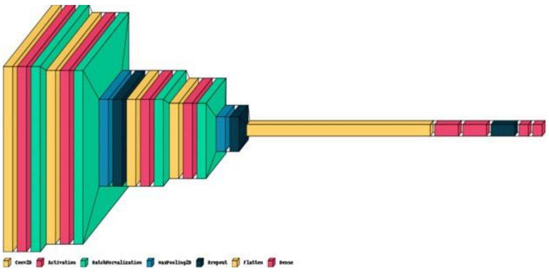
A. Data Analysis

The image shows 10 random images from the dataset .

Visualizing the data using make_grid helper function from torch vision.



Visualizing the Convolution Neural Network



Python Libraries used

- Pandas
- Numpy
- Matplotlib
- Tensorflow
- Torch, Torchvision

Pre-processing steps:

1. Normalization: To Normalize the pixel intensity values, each pixel intensity value was divided by 255 (which is the maximum possible intensity). After normalization, all pixel intensity values have a range between 0 &1.

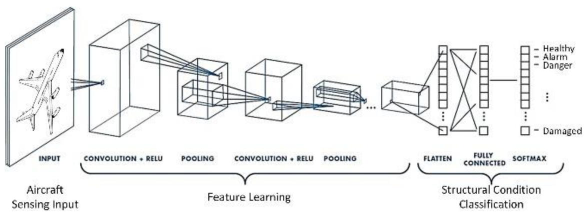
$$X_N = (X - X_{min})/(X_{max} - X_{min})$$

2. One hot encoding: All the categorical class variables in the form of alphabets were converted into a form that could be provided to ML algorithms to do a better job in prediction. One hot encoding converts the classes in the form of a one hot vector which contains all zero's except for the class index values which is given a value 1.

3. Mini-Batch: Since, the data set is very huge, we are using the data loader, with a batch size of 128.Total of 20 epochs were used to update weight parameters.

B. Model

1. CNN



I used CNN models for sign language classification. CNN models were first choice when it comes to image recognition because of various advantages such as sparse connections, parameter sharing. Etc.

Sparse connections: Single element in feature map connected to small patch of elements in the image, whereas each unit is connected to every other unit in the case feedforward networks.

Parameter sharing: Same weights were used for different patches of input images, whereas each connection has a different weight incase of feedforward networks. This allows to perform classification with less parameters.

2. Logistic Regression

Logistic regression, despite its name, is a classification model rather than regression model. Logistic regression is a simple and more efficient method for binary and linear classification problems. It is a classification model, which is very easy to realize and achieves very good performance with linearly separable classes.

Model 1-CNN:

Below is the summary of model 1. It has 4 convolution layers followed by max-pooling layers and dropout layers. The output from these layers is fed to 2 fully connected dense layers.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 64)        1792

 activation (Activation)     (None, 32, 32, 64)        0

 batch_normalization (BatchN  (None, 32, 32, 64)       256
 ormalization)

 conv2d_1 (Conv2D)           (None, 30, 30, 64)        36928

 activation_1 (Activation)   (None, 30, 30, 64)        0

 batch_normalization_1 (Batc  (None, 30, 30, 64)       256
 hNormalization)

 max_pooling2d (MaxPooling2D  (None, 15, 15, 64)       0
 )

 dropout (Dropout)           (None, 15, 15, 64)        0

 conv2d_2 (Conv2D)           (None, 15, 15, 128)       73856

 activation_2 (Activation)   (None, 15, 15, 128)       0

 batch_normalization_2 (Batc  (None, 15, 15, 128)      512
 hNormalization)

 conv2d_3 (Conv2D)           (None, 13, 13, 128)       147584

 activation_3 (Activation)   (None, 13, 13, 128)       0

 batch_normalization_3 (Batc  (None, 13, 13, 128)      512
 hNormalization)

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 128)        0
 2D)

 dropout_1 (Dropout)         (None, 6, 6, 128)         0

 flatten (Flatten)           (None, 4608)              0

 dense (Dense)               (None, 512)               2359808

 activation_4 (Activation)   (None, 512)               0

 dropout_2 (Dropout)         (None, 512)               0

 dense_1 (Dense)             (None, 10)                5130

 activation_5 (Activation)   (None, 10)                0

=================================================================
Total params: 2,626,634
Trainable params: 2,625,866
Non-trainable params: 768
_________________________________________________________________
None
```

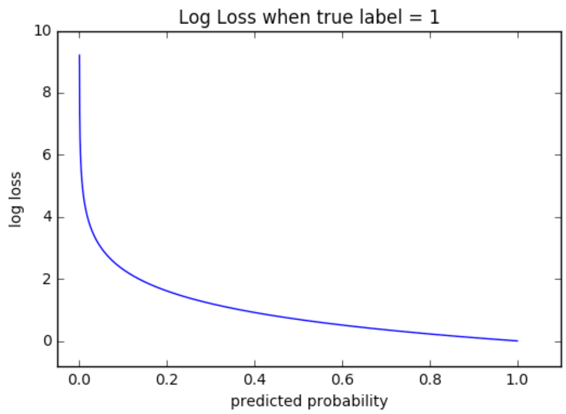
Activation function: Activation function mimics the firing of neurons in the brain, We used ReLu as the activation function after convolution operation for each convolution layer. ReLu is chosen due to it simplicity compared to other choices such as tanh, sigmoid which has exponential terms.

Loss function: Cross-Entropy loss functions is used to optimize the model during training. Cross-Entropy takes the output probabilities (P) and measure the distance from the truth values.

It is given by the formula shown below. T represents the target value, and S represents the predicted value.

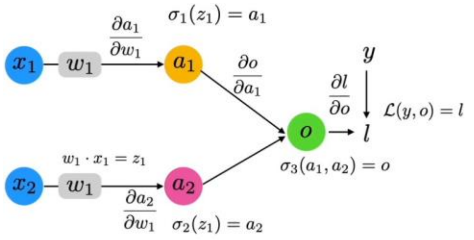
$$L_{CE} = -\sum_{i=1} T_i \log(S_i)$$

Image below shows the plot for a typical cross-entropy loss function. We can clearly observe the loss to be very high in case of prediction probability is less.



Back Propagation for CNN:

Backpropagation was shown below for a simple model. It is similar to feedforward network with minor changes such as same weights used for both inputs & sparse networks between layers.



Upper path

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_1}$$ (multivariable chain rule)

Lower path

Adam Optimizer:

Adam optimizer involves a combination of two gradient descent methodologies:

a. Momentum b. Root Mean Square Propagation (RMSP)

Momentum:
This algorithm is used to accelerate the gradient descent algorithm by taking into consideration the ‘exponentially weighted average’ of the gradients. Using averages makes the algorithm converge towards the minima in a faster pace.

Momentum:-

$$V_{dw} = \beta_1 V_{dw_{prev}} + (1-\beta_1) dw$$

$$V_{dB} = \beta_1 V_{dB_{prev}} + (1-\beta_1) dB$$

- dw = derivative of Loss wrst weight W

- dB = derivative of Loss wrst to Bias B

-V_{dw} = aggregate of gradients (initialise V as ‘0’ and update iteratively)

β1 = Moving average parameter (β1 = 0.9)

RootMean Square Propagation (RMSP):

The Root Mean Square Propagation RMS is a technique to dampen out the motion in the y-axis and speed up gradient descent**.** In the loss graph et us denote the Y-axis as the bias b and the X-axis as the weight W. The idea is to slow down the learning on the y-axis direction and speed up the learning on the x-axis direction.

RMSE propagation

$$S_{dw} = \beta_2 \cdot S_{dw_{prev}} + (1-\beta_2)(dw)^2$$

$$S_{dB} = \beta_2 \cdot S_{dB_{prev}} + (1-\beta_2)(dB)^2$$

- S_{dw} = sum of squares of past gradients

- β2 = Moving average parameters (β2 = 0.999)

Adam Optimizer uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum**.**

Adam:-

$$w = w - \alpha \cdot \frac{V_{dw}}{\sqrt{S_{dw} + \varepsilon}} \qquad B = B - \alpha \cdot \frac{V_{dB}}{\sqrt{S_{dB} + \varepsilon}}$$

Loss Metrics:

Categorical crossentropy (Log loss) is a loss function that is used in multi-class classification tasks. During model training, the model weights are iteratively adjusted accordingly with the aim of minimizing the Cross-Entropy loss. Entropy of a random variable X is the level of uncertainty inherent in the variables possible outcome. The greater the value of entropy,H(x) , the greater the uncertainty for probability distribution and the smaller the value the less the uncertainty.

$$L_{\mathrm{CE}} = -\sum_{i=1}^{n} t_i \log(p_i),$$

t_i = truth or actual label for ith class

p_i = predicted probability for ith class

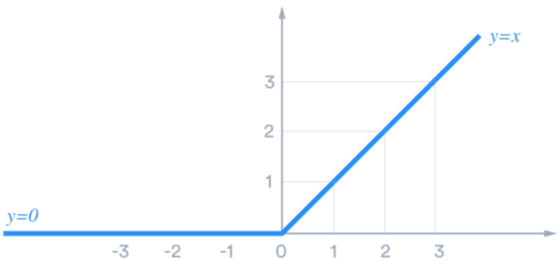
Model-2- Logistic Regression:

The training set is used to train our model, computing loss & adjust weights Validation set is used to evaluate the model with hyper parameters & pick the best model during training. I am using 10% of training data as validation set. Test data set is used to compare different models & report the final accuracy.
We define the model structure (such as number of input features) and Initialize the model's parameters and calculate the current loss (forward propagation
), current gradient (backward propagation), and update the parameters (gradient descent).

Created a class ImageClassificationBase which inherits from nn.module. This does not contain model architecture i.e __init & _forward_ methods.

Similar to linear regression, difference is that we have validation phase as well. input size is of 3x32x32 , output size is 10. We have used 2 hidden layers. The neural network architecture will look like 2048 x 1650 x 512 x 138 x 10 . Images are flattened into vectors and applying layers and activation function and finally getting predictions using output layer. Relu is used as activation function here.

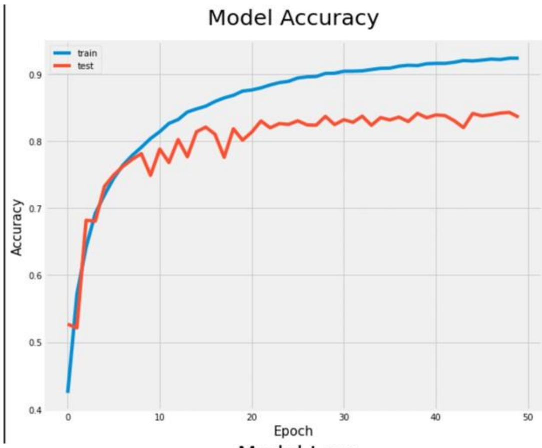
ReLu

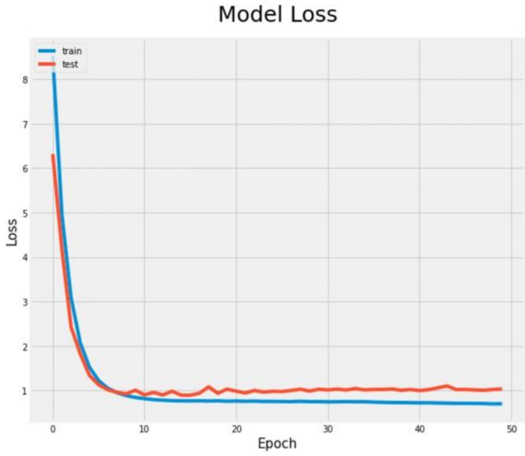


I Trained the model using fit function to reduce the loss and improve accuracy. Here I was trying out different learning rate and epochs. With learning rate of 0.001 and epochs 25 we get the best accuracy.

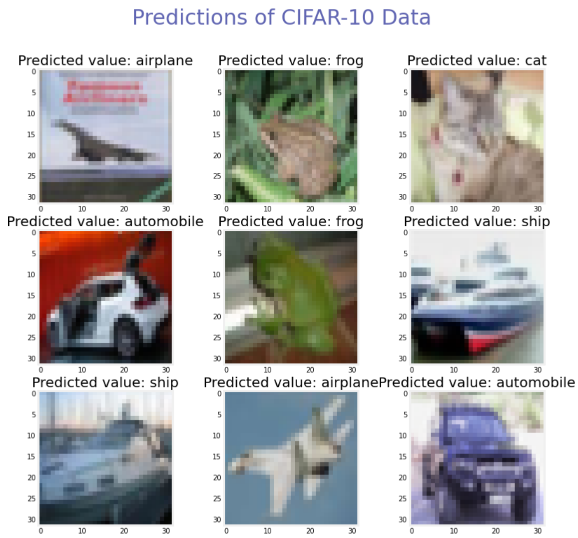
Results:

Model 1: Neural Network with CNN



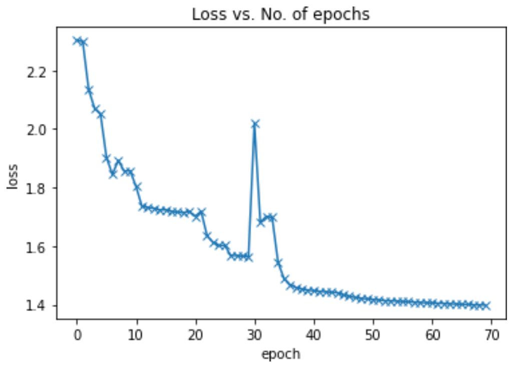


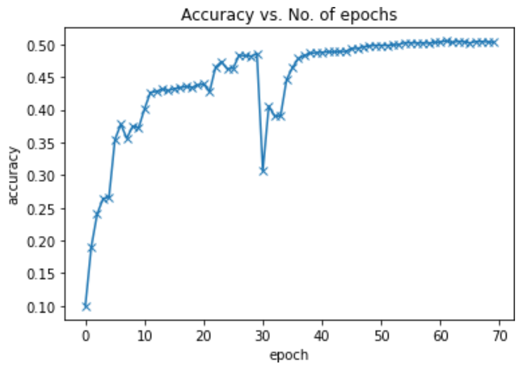
Train loss: 0.7044 , Train accuracy: 0.9236
val_loss: 1.0375 , val_accuracy: 0.8356



Checking the predictions on 9 random images. The model predicts airplane, frog cat, automobile and ship correctly.

Model 2: Logistic Regression





Val loss = 1.389 , val_acc = 0.5034

Test Loss: 1.353, Test acc= 0.5204

Conclusion:
In this project two models were implemented to classify cifar-10 data. I have used CNN based deep learning model for initial classification and achieved test accuracy of 83.5%. And then implemented the Logistic Regression model's accuracy was less for 25 epochs. The logistic regression model works better if the parameters are changed and it shows better accuracy for more number of epochs.

Future work (RESNET)
ResNet makes it possible to train up to hundreds or even thousands of layers and still achieves compelling performance. I want to understand the architecture& how the deep neural network handles the problem of vanishing gradients.

layer name	output size	18-layer	34-layer	50-layer
conv1	112×112			7×7, 64, stride 2
				3×3 max pool, stric
conv2_x	56×56	[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1			average pool, 1000-d fc,

Code:
https://github.com/aishwaryakurnutala/SML_PROJECT

References:
[1]
https://keystrokecountdown.com/articles/logistic/index.html
[2]
https://www.tensorflow.org/tutorials/images/cnn
[3]
https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/

Code:

Neural Network CNN:

```
# importing essential libraries and
packaged
from _future_ import print_function
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.regularizers import l2
import numpy as np
import os
import matplotlib.pyplot as plt
plt.rcParams['axes.unicode_minus'] = False
plt.style.use('fivethirtyeight')

# Defining the parameters
batch_size = 32
num_classes = 10
epochs=50

# Splitting the data between train and test
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

(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')

# plotting some random 10 images

class_names = ['airplane','automobile','bird','cat','deer',
        'dog','frog','horse','ship','truck']

fig = plt.figure(figsize=(10,5))
for i in range(num_classes):
    ax = fig.add_subplot(2, 5, 1 + i, xticks=[], yticks=[])
    idx = np.where(y_train[:]==i)[0]
    features_idx = x_train[idx,::]
    img_num = np.random.randint(features_idx.shape[0])
    im = (features_idx[img_num,::])
    ax.set_title(class_names[i])
    plt.imshow(im)
plt.show()
```

```
# Convert class vectors to binary class matrices.
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

# Building the CNN Model (Hidden Output)

model = Sequential()
model.add(Conv2D(64, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(128, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3)))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512,kernel_regularizer=l2(0.01)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))

!pip3 install visualkeras
# Visualizing our model
import tensorflow as tf
from tensorflow import keras
import visualkeras
visualkeras.layered_view(model, scale_xy=10, legend=True)

print(model.summary())

# compile (Hidden Output)
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

```
# Normalizing the input image
x_train /= 255
x_test /= 255

epochs=50

# Training the model
history = model.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=epochs,
            validation_data=(x_test, y_test),
            shuffle=True)
# Plotting the Model Accuracy & Model Loss vs Epochs
plt.figure(figsize=[20,8])

# summarize history for accuracy
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy', size=25, pad=20)
plt.ylabel('Accuracy', size=15)
plt.xlabel('Epoch', size=15)
plt.legend(['train', 'test'], loc='upper left')
# summarize history for loss

plt.subplot(1,2,2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss', size=25, pad=20)
plt.ylabel('Loss', size=15)
plt.xlabel('Epoch', size=15)
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# Checking the predictions! (Hidden Input)
predictions = model.predict(x_test)

plt.figure(figsize=[12,12])

class_names = ['airplane','automobile','bird','cat','deer','dog','frog','horse','ship','truck']

plt.subplot(3,3,1)
n = 3
plt.imshow(x_test[n].reshape(32, 32, -1), cmap=plt.cm.binary)
plt.title("Predicted value: " + str(class_names[np.argmax(predictions[n], axis=0)]), size=20)
plt.grid(False)

plt.subplot(3,3,2)
n = 4
plt.imshow(x_test[n].reshape(32, 32, -1), cmap=plt.cm.binary)
```

```
plt.title("Predicted value: " + str(class_names[np.argmax(predictions[n], axis=0)]), size=20)
plt.grid(False)

plt.subplot(3,3,3)
n = 8
plt.imshow(x_test[n].reshape(32, 32, -1), cmap=plt.cm.binary)
plt.title("Predicted value: " + str(class_names[np.argmax(predictions[n], axis=0)]), size=20)
plt.grid(False)

plt.subplot(3,3,4)
n = 6
plt.imshow(x_test[n].reshape(32, 32, -1), cmap=plt.cm.binary)
plt.title("Predicted value: " + str(class_names[np.argmax(predictions[n], axis=0)]), size=20)
plt.grid(False)

plt.subplot(3,3,5)
n = 7
plt.imshow(x_test[n].reshape(32, 32, -1), cmap=plt.cm.binary)
plt.title("Predicted value: " + str(class_names[np.argmax(predictions[n], axis=0)]), size=20)
plt.grid(False)

plt.subplot(3,3,6)
n = 1
plt.imshow(x_test[n].reshape(32, 32, -1), cmap=plt.cm.binary)
plt.title("Predicted value: " + str(class_names[np.argmax(predictions[n], axis=0)]), size=20)
plt.grid(False)

plt.subplot(3,3,7)
n = 2
plt.imshow(x_test[n].reshape(32, 32, -1), cmap=plt.cm.binary)
plt.title("Predicted value: " + str(class_names[np.argmax(predictions[n], axis=0)]), size=20)
plt.grid(False)

plt.subplot(3,3,8)
n = 10
plt.imshow(x_test[n].reshape(32, 32, -1), cmap=plt.cm.binary)
plt.title("Predicted value: " + str(class_names[np.argmax(predictions[n], axis=0)]), size=20)
plt.grid(False)

plt.subplot(3,3,9)
n = 9
plt.imshow(x_test[n].reshape(32, 32, -1), cmap=plt.cm.binary)
plt.title("Predicted value: " + str(class_names[np.argmax(predictions[n], axis=0)]), size=20)
plt.grid(False)

plt.suptitle("Predictions of CIFAR-10 Data", size=30, color="#6166B3")

plt.show()
```

Logistic Regression:

```
import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torchvision.datasets import CIFAR10
from torchvision.transforms import ToTensor
from torchvision.utils import make_grid
from torch.utils.data.dataloader import DataLoader
from torch.utils.data import random_split
dataset = CIFAR10(root='data/', download=True, transform=ToTensor())
test_dataset = CIFAR10(root='data/', train=False, transform=ToTensor())

#training image dataset
dataset_size = len(dataset)
dataset_size

#test dataset size
test_dataset_size = len(test_dataset)
test_dataset_size

classes = dataset.classes
classes

#shape of an image from
img, label = dataset[0]
img_shape = img.shape
img_shape

#RGB color image
img, label = dataset[0]
plt.imshow(img.permute((1, 2, 0)))
print('Label (numeric):', label)
print('Label (textual):', classes[label])

#Number of images belonging to each class
img_count_per_class = {label: 0 for label in classes}
for img, label in dataset:
    img_count_per_class[classes[label]] += 1

img_count_per_class

#Preparing data set for training
torch.manual_seed(43)
val_size = 5000
train_size = len(dataset) - val_size
```

```
#creating training & validation set using random_split
train_ds, val_ds = random_split(dataset, [train_size, val_size])
len(train_ds), len(val_ds)

#Creating data loader to load data in batches
batch_size=128

train_loader = DataLoader(train_ds, batch_size, shuffle=True, num_workers=4,
pin_memory=True)
val_loader = DataLoader(val_ds, batch_size*2, num_workers=4, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size*2, num_workers=4, pin_memory=True)

#visualize data using make_grid helper function from torch vision
for images, _ in train_loader:
    print('images.shape:', images.shape)
    plt.figure(figsize=(16,8))
    plt.axis('off')
    plt.imshow(make_grid(images, nrow=16).permute((1, 2, 0)))
    break

#Base model class & training on GPU
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

class ImageClassificationBase(nn.Module):
    def training_step(self, batch):
        images, labels = batch
        out = self(images)                  # Generate predictions
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss

    def validation_step(self, batch):
        images, labels = batch
        out = self(images)                    # Generate predictions
        loss = F.cross_entropy(out, labels)   # Calculate loss
        acc = accuracy(out, labels)           # Calculate accuracy
        return {'val_loss': loss.detach(), 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # Combine losses
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()      # Combine accuracies
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch, result['val_loss'],
result['val_acc']))
```

```
def evaluate(model, val_loader):
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)

def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        # Training Phase
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation phase
        result = evaluate(model, val_loader)
        model.epoch_end(epoch, result)
        history.append(result)
    return history

torch.cuda.is_available()

def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')

device = get_default_device()
device

def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)

class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def _init_(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def _len_(self):
```

```
        """Number of batches"""
        return len(self.dl)

def plot_losses(history):
    losses = [x['val_loss'] for x in history]
    plt.plot(losses, '-x')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.title('Loss vs. No. of epochs');

def plot_accuracies(history):
    accuracies = [x['val_acc'] for x in history]
    plt.plot(accuracies, '-x')
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.title('Accuracy vs. No. of epochs');

train_loader = DeviceDataLoader(train_loader, device)
val_loader = DeviceDataLoader(val_loader, device)
test_loader = DeviceDataLoader(test_loader, device)

#Training the model
input_size = 3*32*32
output_size = 10

class CIFAR10Model(ImageClassificationBase):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(input_size, 1650)
        # hidden layers
        self.linear2 = nn.Linear(1650, 512)
        self.linear3 = nn.Linear(512, 138)
        # output layer
        self.linear4 = nn.Linear(138, output_size)

    def forward(self, xb):
        # Flatten images into vectors
        out = xb.view(xb.size(0), -1)
        # Apply layers & activation functions
        out = self.linear1(out)
        # Apply activation function
        out = F.relu(out)
        # Get intermediate outputs using hidden layer 2
        out = self.linear2(out)
        # Apply activation function
        out = F.relu(out)
        # Get predictions using output layer
        out = self.linear3(out)
        # Apply activation function
        out = F.relu(out)
```

```
        # Get predictions using output layer
        out = self.linear4(out)
        # Apply activation function
        out = F.relu(out)
        return out

model = to_device(CIFAR10Model(), device)

history = [evaluate(model, val_loader)]
history

history += fit(10, 0.05, model, train_loader, val_loader)

history += fit(8, 0.005, model, train_loader, val_loader)

history += fit(7, 0.01, model, train_loader, val_loader)

history += fit(4, 0.001, model, train_loader, val_loader)

history += fit(5, 0.1, model, train_loader, val_loader)

history += fit(10, 0.0001, model, train_loader, val_loader)

#since 0.001 gives best accuracy, will go with that
history += fit(25, 0.001, model, train_loader, val_loader)

plot_losses(history)

plot_accuracies(history)

#Evaluating the test accuracy
evaluate(model, test_loader)
```