# Cognitive Robotics and Computer Vision Report

## 1 Introduction

The aim of this report is to present a comparative study between using traditional computer vision approaches and using deep learning methods for Image Classification. The classification dataset used here is the well-known CIFAR-10 dataset. In this report, I have explored the Bag-of-Words (BoW) model, which incorporates feature extraction methods (such as Dense SIFT), clustering, and ensemble classifiers. Under the deep learning method, the report uses a custom-designed Convolutional Neural Network (CNN) trained from scratch and optimised via hyperparameter tuning. While there is larger explainability in traditional methods, the findings show that CNNs significantly outperform them in both accuracy and scalability. This report discusses the trade-offs among both the methods, with regards to performance, complexity, and interpretability for Visual Recognition Tasks.

## 2 Dataset

The CIFAR-10 [1] is a popular benchmarking classification dataset in computer vision. It consists of 60,000 Coloured Images of size 32×32 pixels. The images are equally distributed across 10 classes as shown in the figure 1: *airplane, automobile, bird, cat, deer, dog, frog, horse, ship,* and *truck*. The dataset is available in a split of 50,000 images for training and validation purposes and the rest 10,000 images for testing purposes. During the experimentation phase, the 50,000 training data was split 80%-20% into training and validation sets.

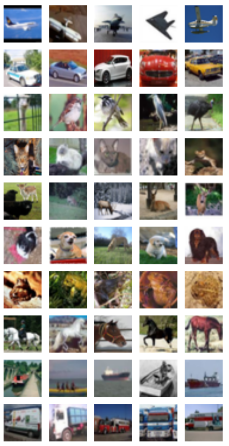


Figure 1: Sample Images CIFAR10

## 3 Methodology and Experimentation

Since there is negligible class imbalance, accuracy was chosen to be the evaluation metric for different experiments across both methods.

### 3.1 Bag of Words Method:

#### 3.1.1 Introduction:

Bag of Words (BoW) [2][3] method is inspired by the tokenization and dictionary-building concepts from the natural language processing methods. It involves developing a collection of key descriptive visual words (equivalent of tokens) across images and clustering them to form a visual vocabulary. The histogram of different such clusters (equivalent to word counts in text) in an image, acts as the representative of the image, which is then fed to classifiers for performance.

#### 3.1.2 Scaling:

The Image size for CIFAR-10 is quite small for traditional computer vision methods as the feature extractors depend on richer textures. Hence, for comparative study, upscaling (2×, 3×, 4×) was performed before testing on BoW method. The Figure 2 below shows how the edges get more smoother on upscaling.


Figure 2: Sample Image Upscaled to different scales

#### 3.1.3 Experimental Setup

Performing a brute force exploration across all parameters to be tested is both inefficient and time-consuming. Hence, a hierarchical setup is considered where we narrow down the search space after every step. The search setup consists of 3 layers, which is to get good Feature Descriptors, Vocabulary Size, and Classifer. The order is chosen based on a typical BoW method flow.

#### 3.1.4 Experimentation: Layer 1

- **Parameters to be explored :** Keypoints, Descriptors, Upscaling
- **Constant Parameters :** 2200 Vocabulary Size via Mini Batch KMeans, Linear SVM Classifier

##### 3.1.4.1 Keypoints:

SIFT (Scale-Invariant Feature Transform) [4] uses DoG (Difference of Gaussians) for automatic keypoints, but repeated scaling and downsampling of an already low-resolution image may not capture many keypoints. Hence, this report makes uses of keypoints from Dense SIFT with a small enhancement. Dense SIFT involves sampling keypoints at regular interval across the image grid. In order to balance keypoint density and computational overload, 64 equally distributed keypoints (8×8 grid) are chosen per image irrespective of scale.

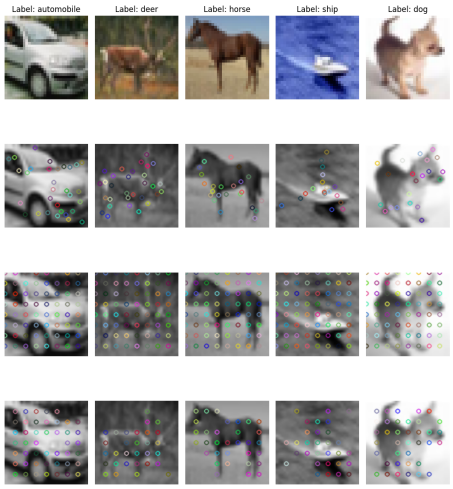



Figure 3: Comparison of Keypoints from SIFT vs Dense SIFT vs Enhanced Dense SIFT for 4x Scaled images

##### 3.1.4.2 Enhancement:

The keypoints where the image patch around them had low variance in their intensity values (low-texture regions) were removed. This enables greater focus on informative areas while also reducing computational complexity for further steps. The fig 3 shows the difference in how keypoints in plain backgrounds were occluded after performing this enhancement.

##### 3.1.4.3 Descriptors:

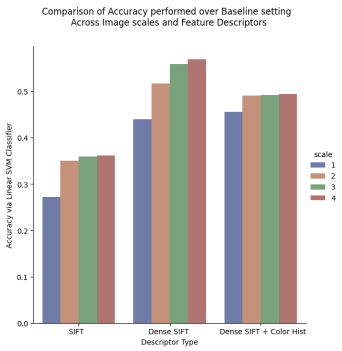



Figure 4: Validation Set Accuracy vs Scale and Descriptors

In this report, SIFT descriptors were used to capture local image features. SIFT encodes gradient orientation patterns around keypoints using a 128-dimensional vector and is robust to scale, rotation, and translation of image patches, making it effective for image representation. Another exploration involved adding the Color Histograms [5] information. Histograms were computed over image patches to capture HSV (Hue Saturation Value) distribution. HSV was chosen over RGB as it is more responsive to lighting variations. The histogram binning was performed to ensure safer computational overloads. Every descriptor had a 128-dimensional SIFT vector + 32-dimensional Color Histogram vector.

However, despite this addition, ColorHist features struggled to outperform enhanced Dense SIFT on CIFAR-10. This is likely due to CIFAR-10's high intra-class variability (in terms of colour and lightings), where spatial features captured by Dense SIFT had richer information than color-based features. The fig 4 shows the comparison of final classification accuracy metrics on validation set when the descriptors and scales were varied. The results showed good accuracies for scales 3× and 4× when Dense SIFT was applied over them.

#### 3.1.5 Experimentation: Layer 2

- **Parameters to be explored:** Vocabulary Size
- **Constant parameters:** Enhanced Dense SIFT on 3×-4× Upscaled images, tested on Linear SVC Classifier

##### 3.1.5.1 Clustering:

To optimise the computational overhead, Mini-Batch KMeans ( Batchwise processing version of KMeans) was employed in this report. After experimenting with different cluster sizes, from the figure 5 we can see that 2500 was the best performing vocabulary size, especially for scale 4×. The entire dataset is now represented by these 2500 Visual words.

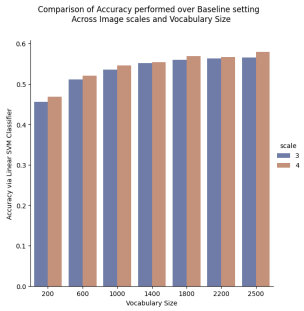



Figure 5: Validation Set Accuracy vs Scale and Vocabulary Size

##### 3.1.5.2 Image Representation via Descriptor Histogram:

Each image is now represented by a histogram of visual words: descriptors are assigned to their nearest cluster (visual word), and a frequency count is computed. These histograms are normalized to emphasize relative importance across clusters.

### 3.1.6 Experimentation: Layer 3

- **Parameters to be explored:** Classifier
- **Constant parameters:** Enhanced Dense SIFT with 4× Upscaling, and 1800-2500 Vocabulary size

#### 3.1.6.1 Classifier

Multiple classifiers such as Gaussian NaiveBayes (gnb), Random Forest (rf), k Nearest Neighbours (kNN), Light Gradient Boosting Method (LGBM), Support Vector Machine with Linear (lsvm) and rbf kernels were experimented. The best performing classifier was SVM (generalizes well to large datasets) but due to the volume of data, it was time consuming to experiment.

In order to match the performance of SVM, an ensemble of the remaining 5 classifiers were used with Voting Classifier. From the fig 6, we can conclude that on most occasions, the ensemble outplayed the individual classifiers by small margin. LGBM and LSVM have performed closest to the Ensemble individually.

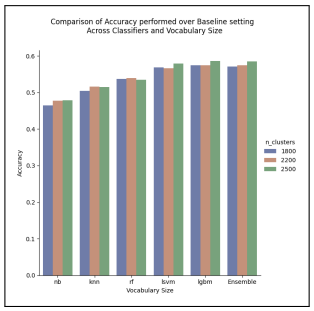



Figure 6: Validation Set Accuracy vs Classifier and Vocabulary Size

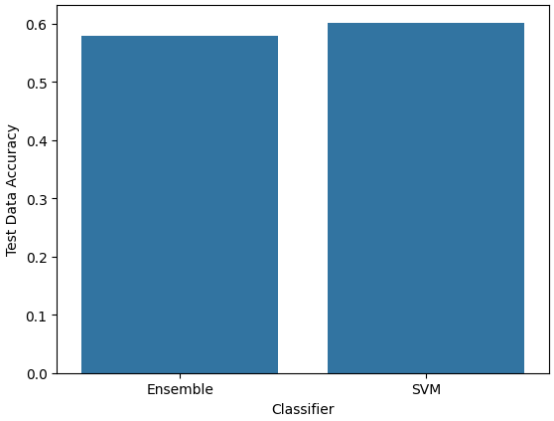
### 3.1.7 Final Model Selection

The final model parameters chosen are:

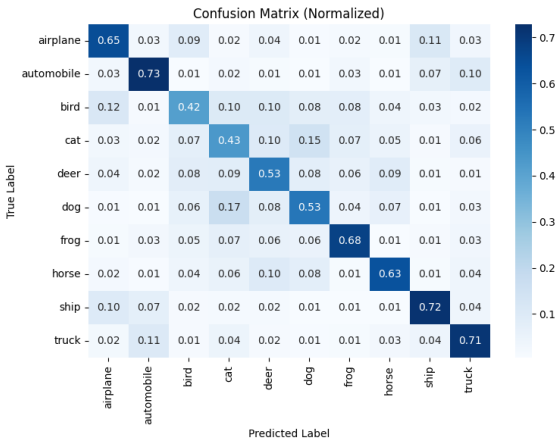
- Descriptors obtained **Enhanced Dense SIFT** applied on **4**× upscaled images
- Clustered with **Mini Batch KMeans 2500 Vocabulary size**
- Validated on an **Ensemble** of Gaussian Naive Bayes, Random Forest, kNN, Light GBM, Linear SVM Classifiers

### 3.1.8 Testing on Unseen Data:

From results 7, we can see that, the final test accuracy was around 60% when SVM was used which was a slight improvement over the Ensemble (however its worthful to consider time complexity). The confusion matrix highlights more misclassifications among the animal groups (deer, cat, dog, etc) while there were good distinctions among structurally rigid class groups such as truck, ship, automobile, and airplane. Sample predictions are shown in fig 8.




(a) Accuracy on test set compared with Ensemble and SVM Classifiers




(b) Confusion Matrix

Figure 7: Results on Unseen Test Set

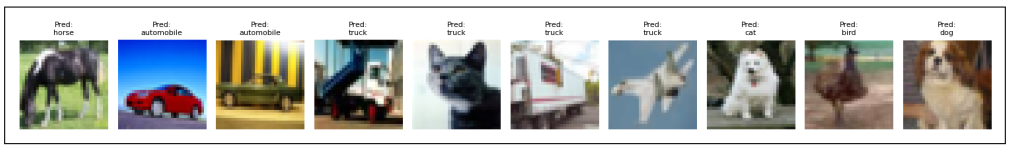



Figure 8: Sample Predictions from the BoW Method

## 3.2 Convolutional Neural Networks (CNNs)

### 3.2.1 Introduction

Convolution Neural Networks (CNNs) [6] come under the class of Deep Neural Networks designed to learn features from image data. Unlike fully connected neural networks, they make use of convolutions which capture spatial patterns, hence making them the top choice for vision-based problems.

### 3.2.2 Normalization

Normalizing an image helps in maintaining numerical stability for gradients. During the CNN experimentation, the images are normalized to have pixel values centered about 0 for each of the RGB channels. Per channel values from 0-255 are normalized to lie within -1 to +1 range. This method helps in accelerating convergence while training, especially when using non-linear activation function such as ReLU.

### 3.2.3 Experimental Setup

Performing a brute force exploration across all parameters is both inefficient and time consuming. Hence, a hierarchical setup is considered where we narrow down search space after every step. The search setup consists of 3 layers. The first layer is to experiment with different architecture sizes, optimizers, and learning rates. The second layer aims to tweak the batch-size to find a better solution. The third layer dives deeper into solving the issues that came in the previous layers by tweaking optimizer parameters. An early stopping criteria in included while training, which stops the epoch run whenever the difference between train and validation accuracy exceeds 15% (indicative of model overfitting). Across all experimentation CrossEntropyLoss function is chosen.

### 3.2.4 Experimental Setup: Layer 1

- **Parameters under focus:** CNN Architecture, Optimizer and Learning Rate (lr)
- **Constant Parameters:** Batch Size = 64

#### 3.2.4.1 Architecture

The data was tested against 3 types of custom-made architectures, namely small, medium, and large architectures based on number of trainable parameters. The small architecture had ∼14k trainable params, the medium one had ∼60k trainable parameters, and the large one had ∼100k trainable parameters. The model summaries are shown in the Appendix. While the smaller and medium models were just Convolutional layers stacked and converted to dense, the larger model was designed with a block-like approach. A block is considered to be a set of convolutional layers and such blocks are repeated until the image seems fit to be flattened.The architecture of the larger model in shown in fig 9.

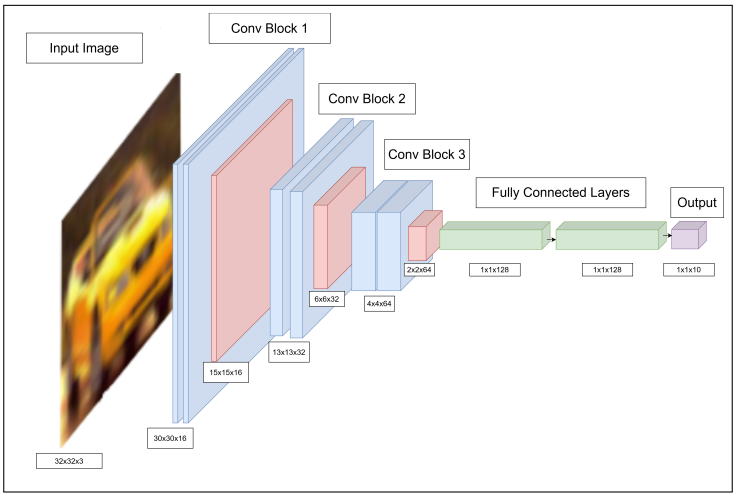



Figure 9: Large Model Architecture

#### 3.2.4.2 Optimizers

In this stage, 2 optimizers were tested. Adam Optimizer and SGD Optimizer. Since other parameters are kept constant, the current SGD optimizer has no momentum.

#### 3.2.4.3 Learning Rates

Two learning rates were considered: 0.01 and 0.001 for testing at this stage.

#### 3.2.4.4 Results

The following table 1 shows the top 5 models sorted based on the difference in train and validation accuracies. At this stage the Larger model fit with an SGD optimizer (lr: 0.01) proved to be marginally better than other experiments. Although Adam optimizer provided a slightly better validation accuracy, it has tendency to overfit while SGD has momentum parameters which can be used for controllability.

| Model Size | Optimizer | Learning Rate | Train Accuracy | Validation Accuracy |
|---|---|---|---|---|
| Large | SGD | 0.01 | 76.99% | 70.18% |
| Large | Adam | 0.001 | 79.22% | 71.49% |
| Small | SGD | 0.01 | 74.64% | 65.08% |
| Small | SGD | 0.001 | 73.96% | 63.66% |
| Medium | Adam | 0.001 | 78.31% | 67.36% |

Table 1: Experiment Layer 1: Top 5 results

### 3.2.5 Experimental Setup: Layer 2

- **Parameters to be explored:** Batch Size
- **Parameters kept constant:** Optimizers are kept at default and no dropout layers in Architecture.

#### 3.2.5.1 Batch Size Results

In this setup we vary Batch sizes among 16, 32, 64, 128 to compare with the model performance. We can conclude from the results 10 that while validation accuracy was almost similar across the batch sizes, there was less overfitting for batch size=64, hence this was selected.

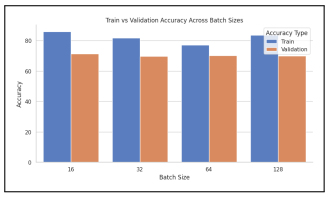



Figure 10: Train and Validation set accuracy at different batch sizes

### 3.2.6 Experimental Setup: Layer 3

- **Parameters to be explored:** Momentum, Dropout & weight decay

#### 3.2.6.1 Momentum :

Momentum plays a key role in accelerating gradients to speed up convergence. Here I considered to experiment with 0.8,0.9,0.99 momentum. While the model has not been largely overfitting, adding momentum may fit the training data faster. Hence 2 types of regularizations are tested.

#### 3.2.6.2 Dropout :

Dropout is a type of regularization that is associated with the CNN architecture. Here the dropout is experimented between the final fully connected layers with fractions of 0.3 and 0.5.

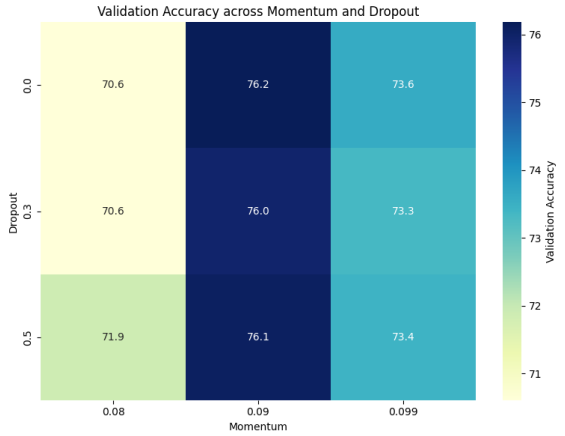
#### 3.2.6.3 Weight Decay :

Weight decay is a regularization method more closely related to the optimizer than the architecture. Here weight decays of $10^{-4}$, $10^{-3}$, $2 \times 10^{-3}$ and $10^{-2}$ were considered.

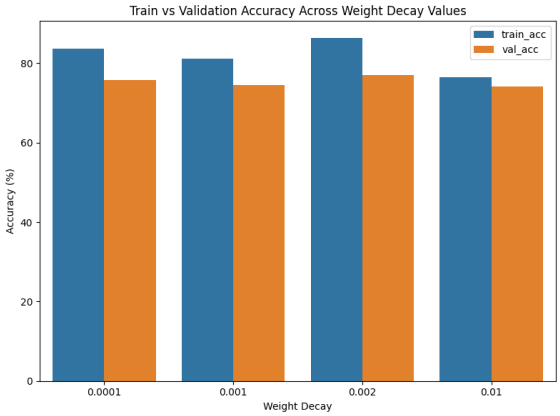
#### 3.2.6.4 Results

We see a sharp 6% increase 11a in validation accuracies with a momentum of 0.09. The negligible evidence across dropouts tells us that the overfitting is not associated with the architecture. The fig 11b suggests a slight 1% increase in accuracy when weight decay is considered.

### 3.2.7 Final Model Selection and Results




(a) Accuracy on validation set compared across dropouts and momentum




(b) Accuracy on Train vs Val sets for different Weight Decays

Figure 11: Results after tweaking Momentum, Dropout, and Weight Decay

The final model parameters chosen are:

- Large Model (100k params) + No dropouts + 64 Batch size
- SGD Optimizer (momentum = 0.9, lr = 0.01, weight decay = $2 \times 10^{-3}$)
- Train Set Accuracy: ~85 %
- Validation Set Accuracy: ~77.1 %

### 3.2.8 Testing on Unseen Data:

From results 13, the final test accuracy was ~ 76.1 %. The confusion matrix shown highlights more misclassifications among the animal groups (deer, cat, dog, etc) while there were good distinctions among structurally rigid class groups such as truck, ship, automobile, and airplane.

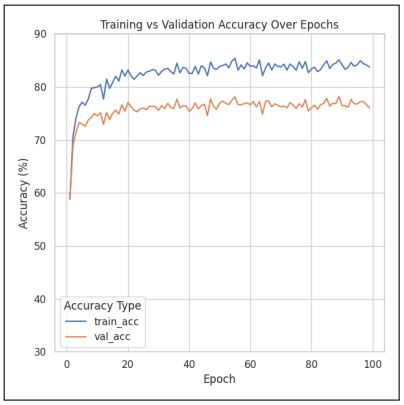
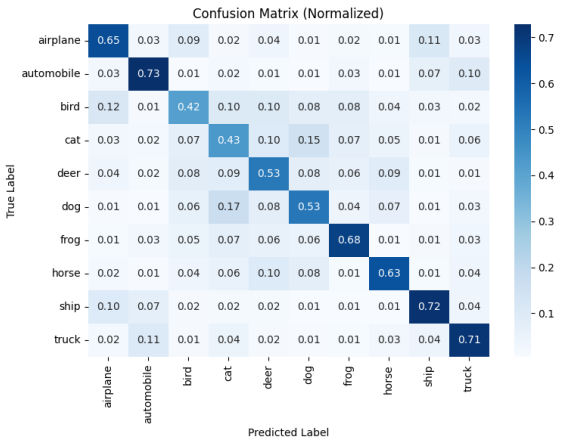



Figure 12: Train and Validation set accuracy across epochs for the final model




(a) Confusion Matrix


(b) Sample Test Set Predictions

Figure 13: Final Test Dataset Results

# 4 Result Comparison: Traditional CV vs Deep Learning

For the traditional CV, the Enhanced Dense SIFT features combined with Linear SVM Classifier is chosen for comparison. The reason to not use the non-linear SVM here is that despite its 2% extra accuracy, it is a slow and long duration procedure, while the Ensembles usually tend to overfit. Hence, the closest classifier in terms of accuracy was chosen to be the Linear SVM.

The result comparison highlights the fact that Deep Neural Networks outperform the traditional CV model presented in this report. Although the traditional CV model is almost matching with the smaller CNNs, it fails considerably when compared to the larger Neural Networks. However, it is also worth noticing the overfitting problem, which can sometimes occur more in the case of Neural Networks, especially when working on small-size datasets.

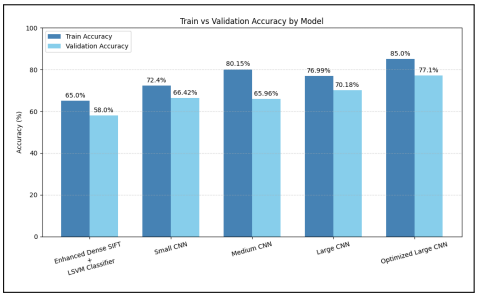



Figure 14: Train and Validation set accuracy across different models

# 5 State of the Art

Computer Vision plays an incredibly pivotal role across applications in Robotics. While traditionally the robotic systems heavily relied on sensors like LiDAR, these days, with the availability and the cheapness, camera is integrated on every robotic system. To enhance the power of a camera, computer vision in of utmost necessity. In this section, I shall discuss about two major places where computer vision is used in robotic systems.

## 5.1 Role of Computer Vision in Mobile Robots

Mapping an environment forms a key part of any mobile system. While LiDAR aids in mapping the boundaries, cameras provide information about colours and object systems around a robot. For a larger system such as Autonomous cars, CNNs are very widely used [7] as road image data is available in abundance (such as KITTI [8] ). Also the abundance of pre-trained models, aids people to apply transfer learnings [9] into their systems to quickly adapt to their requirements. A larger phyical system also enables space for large computational units which the CNNs demand. However, for systems on short scale, it is difficult to embed the powerful computing units required for deep neural networks. In such scenarios traditional computer vision algorithms are very helpful as they are easily deployable. Also deploying robots in a slightly unknown environments, traditional CV algorithms can work better as they are not in need for large corpses of data like Deep Learning. Scene construction [10] is also a widely explored area in autonomous cars where traditional CVs have been used for image stitching and 3D scene reconstructions.

## 5.2 Role of Computer Vision on Robotic Arms

Another major class of robots involves fixing cameras on top of a production line or around a robotic arm. In a more pick-and-place use cases [11], such as in a factory production line, traditional CVs are useful as they can be easily trained upon known test scenarios and automate it. The smaller model size is also advantageous in deploying them on embedded systems. However, for more complex tasks such as a surgical arm [12] or a tool tracking systems, CNNs are preferred due to their automated feature selection process unlike hand-crafted ones in traditional systems making them robust.

# References


[1] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., University of Toronto, 2009.

[2] G. Csurka, C. R. Dance, L. Fan, J. Willamowski, and C. Bray, "Visual categorization with bags of keypoints," in *Workshop on statistical learning in computer vision, ECCV*, vol. 1, pp. 1–2, 2004.

[3] L. Fei-Fei, R. Fergus, and P. Perona, "A bayesian hierarchical model for learning natural scene categories," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2, pp. 524–531, IEEE, 2005.

[4] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[5] M. J. Swain and D. H. Ballard, "Color indexing," *International journal of computer vision*, vol. 7, no. 1, pp. 11–32, 1991.

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[7] J. Song and J. Lee, "Cnn-based object detection and distance prediction for autonomous driving using stereo images," *International Journal of Automotive Technology*, vol. 24, pp. 773–786, 2023.

[8] J. Fritsch, T. Kuehnl, and A. Geiger, "A new performance measure and evaluation benchmark for road detection algorithms," in *2013 IEEE International Conference on Intelligent Transportation Systems*, pp. 1693–1700, IEEE, 2013.

[9] M. Ballesta, L. Payá, S. Cebollada, O. Reinoso, and F. Murcia, "A cnn regression approach to mobile robot localization using omnidirectional images," *Applied Sciences*, vol. 11, p. 7521, 2021.

[10] W. Zhang, X. Wang, and Q. Li, "3d reconstruction based on sift and harris feature points," in *2010 International Conference on Computer Application and System Modeling*, vol. 11, pp. V11–595–V11–598, IEEE, 2010.

[11] Y. Li, X. Wang, and W. Zhang, "Sift-based segmentation of multiple instances of low-textured objects for pick-and-place applications," *International Journal of Computer Theory and Engineering*, vol. 4, no. 3, pp. 486–490, 2012.

[12] A. A. Shvets, A. Rakhlin, A. A. Kalinin, and V. I. Iglovikov, "Automatic instrument segmentation in robot-assisted surgery using deep learning," *arXiv preprint arXiv:1803.01207*, 2018.

# Appendix

# CIFAR_SIFT_final

May 9, 2025

[3]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import cv2
import gc
import os
import lightgbm as lgb
import random
import pickle

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import MiniBatchKMeans
from sklearn.metrics import accuracy_score, confusion_matrix,␣
↪classification_report, silhouette_score
from sklearn.svm import SVC, LinearSVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB



from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

## 0.1 DATA LOADING

[ ]:
```
class DataLoader():
  '''
  Loading Data for CIFAR10 from locally downloaded files taken from CIFAR10␣
↪website.
  '''
  def __init__(self, path, train_frac = 0.8, sample_size = None):
```

```
    self.path = path
    self.train_frac = train_frac
    self.sample_size = sample_size

    self.train_list, self.val_list, self.test_list = self.get_img_lists()


  def unpickle(self,file):
      with open(file, 'rb') as fo:
          dict = pickle.load(fo, encoding='bytes')
      return dict

  def load_data(self,file_name):
    data = self.unpickle(file_name)
    del data[b'batch_label']
    img_list = []
    for i in range(len(data[b'filenames'])):
      img_list.append([data[b'filenames'][i], (data[b'data'][i].
↪reshape(3,32,32).transpose(1,2,0)), data[b'labels'][i]])
    return img_list

  def get_img_lists(self):
    '''
    Returns 3 lists : each entry in a list is a sample : (image_name,␣
↪image_data, label)
    '''
    image_list = []
    for i in range(5):
      image_list = image_list + self.load_data(self.path + f'data_batch_{i+1}')
    test_list = self.load_data(self.path + 'test_batch')

    random.shuffle(image_list)
    train_list, val_list = image_list[:int(self.train_frac*len(image_list))],␣
↪image_list[int(self.train_frac*len(image_list)):]
    if self.sample_size is not None:
      train_list = random.sample(train_list, self.sample_size)
    print(len(train_list),len(val_list),  len(test_list))
    return train_list, val_list, test_list
```

# 1 FEATURE EXTRACTION

[ ]:
```
class FeatureDescriptors():
  '''
  Class to extract feature descriptors from images based on config.
  '''
  def __init__(self, data, config):
```

```
  self.config = config

  self.data = data
  self.ft_type = config['type']
  self.scale = config['scale']
  self.grayscale = config['grayscale']
  self.step_size = config['step_size']
  self.color_hist = config['color_hist']


  self.ft_data = self.get_ft_df(self.data, ft_type = self.ft_type, scale =␣
↪self.scale, grayscale = self.grayscale, step_size = self.step_size,␣
↪color_hist = self.color_hist)


def get_feature_obj(self,ft_type):
  if ft_type == 'sift' or ft_type == 'dsift':
    desc = cv2.SIFT_create()
  return desc

def is_valid_kp(self,img_gray, kp, patch_size=16, threshold=100):
    '''
    Function that checks if a keypoint is valid
    If the variance in a patch around keypoint is less than threshold, it is␣
↪not a valid keypoint.
    '''
    x, y = int(kp.pt[0]), int(kp.pt[1])
    h, w = img_gray.shape
    half = patch_size // 2

    if x - half < 0 or x + half >= w or y - half < 0 or y + half >= h:
        return False

    patch = img_gray[y-half:y+half, x-half:x+half]
    return np.var(patch) >= threshold


def get_descriptors(self,img, ft_type = 'sift',scale = 1, grayscale = True,␣
↪step_size = 4):
  '''
  Returns keypoints and descriptors for a given image.
  ft_type : 'sift' and 'dsift' (dense sift) are allowed
  step_size : is used only for dsift
  scale : is used for resizing the image
  grayscale : is used to convert the image to grayscale (always set to true␣
↪for SIFT)
```

```
    '''
    desc = self.get_feature_obj(ft_type)

    img  = cv2.resize(img, (int(32*scale), int(32*scale)), interpolation=cv2.
↪INTER_CUBIC)

    if grayscale:
      img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
      self.img_gray = img

    if ft_type == 'dsift':
      keypoints = []
      for i in range(0, img.shape[0], step_size):
        for j in range(0, img.shape[1], step_size):
          kp = cv2.KeyPoint(i,j,step_size)
          if self.is_valid_kp(img, kp, step_size):
            keypoints.append(kp)
      _, descriptors = desc.compute(img, keypoints)
    else:
      keypoints, descriptors = desc.detectAndCompute(img, None)

    if descriptors is not None:
      if len(descriptors) > 0:
        norms = np.linalg.norm(np.array([x for x in descriptors if x is not␣
↪None]), axis=1, keepdims=True)
        desc_norm = descriptors / (norms + 1e-10)
      else:
        desc_norm = desc
    else:
      desc_norm = descriptors
    return keypoints,desc_norm


  def get_color_hist(self,img,kps, scale = 1, step_size =4):
    '''
    Returns color histogram for a given image and keypoints.

    '''
    kpx = [int(kp.pt[0]) for kp in kps]
    kpy = [int(kp.pt[1]) for kp in kps]
    img  = cv2.resize(img, (int(32*scale), int(32*scale)), interpolation=cv2.
↪INTER_CUBIC)
    hist_list = []
    hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV) # Converts rgb to hsv space
    for i,j in zip(kpx,kpy):
        hsv_sub = hsv[i:i+step_size, j:j+step_size]
```

```
      # Bins HSV values in 2-4-4 bins making a total of 32 extra points of␣
↪data for each descriptor
      hist = cv2.calcHist([hsv_sub], [0, 1, 2], None, [2, 4, 4],
                [0, 180, 0, 256, 0, 256])
      hist = cv2.normalize(hist, hist).flatten()
      hist_list.append(hist)

  hist = np.array(hist_list)
  return hist


def get_ft_df(self,data, ft_type = 'sift',scale = 1, grayscale = True,␣
↪step_size = 4, color_hist = False):
  ft = []
  kp_list = []
  name = []
  label = []
  for idx,img in enumerate(data):
    name.append(img[0])
    label.append(img[2])
    kp,desc = self.get_descriptors(img[1], ft_type = ft_type, scale = scale,␣
↪grayscale=grayscale, step_size = step_size) # Get descriptors

    if color_hist:
      col_hist = self.get_color_hist(img[1],kps = kp, scale = scale,␣
↪step_size = step_size) # If color histogram needed, get and append it to␣
↪original images
      final_desc = np.hstack((col_hist, desc))
    else:
      final_desc = desc

    if final_desc is not None:
      if len(final_desc) > 0:
        try:
          final_desc = final_desc / (np.linalg.norm(final_desc,axis=1,␣
↪keepdims=True) + 1e-10)
        except Exception as e:
          print(e, final_desc)
          final_desc = None
      else:
        final_desc = None
    else:
      final_desc = None

    ft.append(final_desc)
    kp_list.append(kp)
  ft_data = (list(zip(name,label,ft,kp_list)))
```

```
    return ft_data
```

## 2 CLUSTERING

[ ]:
```
class Clustering():
  '''
  Class for enable clustering
  '''
  def __init__(self, data, config, fit = False, clus_obj = None):
    self.config = config
    self.data = data
    self.fit = fit
    self.clus_obj = clus_obj
    self.unzipped = list(zip(*data))

    self.name = self.unzipped[0]
    self.label = self.unzipped[1]
    self.descs = self.unzipped[2]

    self.all_descriptors = []
    for img_desc in self.descs:
      if img_desc is not None:
        self.all_descriptors.extend(img_desc)

    self.cluster_alg = config['cluster_alg']
    self.n_clusters = config['n_clusters']

    self.data_new = self.get_final_train_data()

  def get_hist(self,descriptors, kmeans):
    clust_hist = np.zeros(kmeans.n_clusters)
    if descriptors is None:
      return clust_hist

    clus = kmeans.predict(descriptors)

    for c in clus:
        clust_hist[c] += 1
    clust_hist = clust_hist / np.sum(clust_hist)
    return clust_hist

  def generate_histograms(self,data, clus_obj):
    data_hist = [self.get_hist(x, clus_obj) for x in self.descs]

    return data_hist
```

```
  def get_cluster_obj(self,alg, n_clusters):
    if alg == 'kmeans':
      clus_obj = KMeans(n_clusters=n_clusters)
    if alg == 'mkmeans':
      clus_obj = MiniBatchKMeans(n_clusters=n_clusters, batch_size = 1000)
    return clus_obj

  def fit_cluster(self):
    self.cl_obj = self.get_cluster_obj(self.cluster_alg, self.n_clusters)
    self.cl_obj.fit(np.array(self.all_descriptors))
    return self.cl_obj

  def get_final_train_data(self):
    '''
    Returns the data histograms after converting descriptors to embedded␣
↪cluster space
    '''
    if self.fit:
      self.clus_obj = self.fit_cluster()

    self.data_hist = self.generate_histograms(self.data, self.clus_obj)
    return self.data_hist, self.label
```

[ ]:
```
class Classifier():
  '''
  Class for enable classification
  '''
  def __init__(self, X_val, y_val, config, fit=False, X_train=None,␣
↪y_train=None, clf = None):
    self.config = config
    self.X_train = X_train
    self.y_train = y_train
    self.X_val = X_val
    self.y_val = y_val
    self.fit = fit
    self.clf = clf

    self.acc_results = self.train_val_test_results()

  def get_classifier_obj(self,alg):
    if alg == 'linearsvm':
      clf = LinearSVC()
    if alg == 'svm':
      clf = SVC(kernel='rbf')
    if alg == 'knn':
      clf = KNeighborsClassifier()
```

```
    if alg == 'randomforestclassifier':
      clf = RandomForestClassifier()
    if alg == 'lgbm':
      clf = lgb.LGBMClassifier(verbose = -1)
    if alg == 'gnb':
      clf = GaussianNB()
    return clf

  def fit_predict(self,clf,X_train, y_train,X_val, y_val):
    if self.fit:
      clf = clf.fit(X_train, y_train)
    else:
      clf = self.clf
    y_pred = clf.predict(X_val)
    val_acc = accuracy_score(y_val, y_pred)
    return clf,val_acc

  def train_val_test_results(self):
    acc_results = dict()
    clf1 = self.get_classifier_obj('randomforestclassifier')
    clf2 = self.get_classifier_obj('linearsvm')
    clf3 = self.get_classifier_obj('knn')
    clf4 = self.get_classifier_obj('gnb')
    clf5 = self.get_classifier_obj('lgbm')

    self.eclf = VotingClassifier(estimators=[
            ('rf', clf1), ('lsvm',␣
 ↪clf2),('knn',clf3),('gnb',clf4),('lgbm',clf5)], voting='hard')
    self.eclf, en_val_acc = self.fit_predict(self.eclf, X_train, y_train,X_val,␣
 ↪y_val)
    acc_results['Ensemble'] = en_val_acc
    for name, cfr in self.eclf.named_estimators_.items():
      _,val_acc = self.fit_predict(cfr, X_train, y_train,X_val, y_val)
      acc_results[name] = val_acc

    return acc_results
```

## 3 RUNNING THE BoW

[ ]:
```
'''
Config File Inputs:
type: 'dsift' or 'sift' as per need of descriptors
scale: How much to upscale the image by
grayscale : Bool, True if you want to grayscale image before getting␣
 ↪descriptors [essential for SIFT]
stepsize : int , Feature used only for dense sift to give step size of keypoints
```

```
color_hist : Bool, True if you need to append Color histograms to each␣
 ↪desdcriptors
cluster_alg: 'kmeans' or 'mkmeans@ (Mini Batch K means) for clustering
n_clusters: int, Number of clusters for clustering
'''

config = {
    'type': 'dsift',
    'scale': 2,
    'grayscale': True,
    'step_size':8 ,
    'color_hist' : True,
    'cluster_alg' : 'mkmeans',
    'n_clusters' : 1000
}

dataloader = DataLoader(path = '/content/drive/MyDrive/cifar-10-python/
 ↪cifar-10-batches-py/', sample_size = None)
train_list = dataloader.train_list
val_list = dataloader.val_list
test_list = dataloader.test_list


data_df = FeatureDescriptors(train_list,config).ft_data
val_df = FeatureDescriptors(val_list,config).ft_data
test_df = FeatureDescriptors(test_list,config).ft_data

final_results = dict()
data_clus_obj = Clustering(data_df, config, fit = True)
X_train, y_train = data_clus_obj.data_new
clus_obj = data_clus_obj.clus_obj

X_val, y_val = Clustering(val_df, config, fit = False, clus_obj = clus_obj).
 ↪data_new
X_test, y_test = Clustering(test_df, config, fit = False, clus_obj = clus_obj).
 ↪data_new

classifier_obj = Classifier(X_val, y_val, config, fit = True, X_train=X_train,␣
 ↪y_train=y_train)
acc_results = classifier_obj.acc_results
classifier = classifier_obj.eclf

test_results = classifier.predict(X_test)
test_acc = accuracy_score(y_test, test_results)

final_results['config'] = config
final_results['acc_results'] = acc_results
```

```
final_results['test_acc'] = test_acc

print(final_results)
```

# CIFAR_CNN_Final

May 9, 2025

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import cv2
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchsummary import summary
import tarfile
import os
from torchvision import transforms

from google.colab import drive
drive.mount('/content/drive')

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
Mounted at /content/drive
```

## 1 Loading Data

[2]:
```
def get_cifar_data(batch_size, tr_split = 0.8):
  '''
  Function to load the CIFAR-10 dataset and split it into training, validation,␣
 ↪and test sets.
  Returns the dataloaders for the training, validation, and test sets.
  '''

  transform = transforms.Compose(
      [transforms.ToTensor(),
      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
      ])

  trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
```

```
                                          download=True, transform=transform)
  dataset_len = len(trainset)
  trainset_len = int(dataset_len*tr_split)
  valset_len = dataset_len - trainset_len

  testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                         download=True, transform=transform)
  trainset, valset = torch.utils.data.random_split(trainset, [trainset_len␣
↪,valset_len])

  trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                            shuffle=True, num_workers=2)
  valloader = torch.utils.data.DataLoader(valset, batch_size=batch_size,
                                            shuffle=True, num_workers=2)
  testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                          shuffle=False, num_workers=2)


  classes = ('plane', 'car', 'bird', 'cat',
             'deer', 'dog', 'frog', 'horse', 'ship', 'truck')


  return trainloader, valloader, testloader, classes
```

## 2 Training and Prediction Function

[3]:
```
global val_results
def predict(model, loader):
  '''
  Function to predict the labels of the images in the validation set.
  Returns the predictions and the true labels along with Accuracy.
  '''
  with torch.no_grad():
    correct = 0
    total = 0
    preds = []
    true_labels = []
    for images, labels in loader:
        images, labels = images.type(torch.float32).to(device), labels.
↪to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        preds.append(predicted)
        true_labels.append(labels)
```

```
    preds = torch.cat(preds)
    true_labels = torch.cat(true_labels)
    acc = 100 * correct / total
  return preds, true_labels, acc


def train(model, optimizer,criterion,trainloader, valloader, num_epochs,steps =␣
 ↪[10,25,50,75,100], scheduler = None):
  '''
  Function to train the model.
  Returns the validation results.
  steps: list of epochs to save the validation results at.
  '''
  global val_results
  val_results = dict()
  for epoch in range(num_epochs):
      running_loss = 0.0
      for images, labels in trainloader:
          images, labels = images.type(torch.float32).to(device), labels.
 ↪to(device)

          optimizer.zero_grad()
          outputs = model(images)
          loss = criterion(outputs, labels)
          loss.backward()
          optimizer.step()


          running_loss += loss.item()

      print(f"Epoch {epoch + 1}, Loss: {running_loss / len(trainloader):.4f}")
      val_preds, val_true_labels, val_acc = predict(model, valloader)
      print(f"Validation Accuracy: {val_acc:.2f}%")

      if scheduler is not None:
          scheduler.step(val_acc)
      if (epoch+1) in steps:
        val_results[epoch+1] = dict()
        train_preds, train_true_labels, train_acc = predict(model, trainloader)
        print(f"Training Accuracy: {train_acc:.2f}%")
        val_results[epoch+1]['train_acc'] = train_acc
        val_results[epoch+1]['val_acc'] = val_acc

        if train_acc - val_acc > 15:   # Early stopping criteria
          print(f'Early Stopping at epoch {epoch+1}')
          val_results[epoch+1]['train_preds'] = train_preds
          val_results[epoch+1]['val_preds'] = val_preds
```

```
            val_results[epoch+1]['train_true_labels'] = train_true_labels
            val_results[epoch+1]['val_true_labels'] = val_true_labels
            break

    return val_results
```

## 3 Network Architectures Used

[5]:
```
class SmallNet(nn.Module):
    '''
    Small Network Architecture
    Input: 32x32x3
    Output: 10
    Total Params: ~15000
    '''
    def __init__(self):
        super(SmallNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, kernel_size=3)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3)
        self.conv3 = nn.Conv2d(16,32, kernel_size=3)
        self.fc1 = nn.Linear(128, 64)
        self.fc2 = nn.Linear(64, 10)
        self.max_pool = nn.MaxPool2d(2,2)

    def forward(self, x):
        x = self.max_pool(F.relu(self.conv1(x)))
        x = self.max_pool(F.relu(self.conv2(x)))
        x = self.max_pool(F.relu(self.conv3(x)))
        x = x.view(-1,128)
        x = self.fc2(F.relu(self.fc1(x)))

        return x

model = SmallNet().to(device)
summary(model, (3,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 8, 30, 30]             224
         MaxPool2d-2            [-1, 8, 15, 15]               0
            Conv2d-3           [-1, 16, 13, 13]           1,168
         MaxPool2d-4             [-1, 16, 6, 6]               0
            Conv2d-5             [-1, 32, 4, 4]           4,640
         MaxPool2d-6             [-1, 32, 2, 2]               0
            Linear-7                   [-1, 64]           8,256
```

```
            Linear-8                   [-1, 10]             650
================================================================
Total params: 14,938
Trainable params: 14,938
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.10
Params size (MB): 0.06
Estimated Total Size (MB): 0.17
----------------------------------------------------------------
```

[6]:
```python
class MediumNet(nn.Module):
    '''
    Medium Network Architecture
    Input: 32x32x3
    Output: 10
    Total Params: ~60000
    '''
    def __init__(self):
        super(MediumNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, kernel_size=4, padding = 'valid')
        self.conv2 = nn.Conv2d(8, 16, kernel_size=4,padding='valid')
        self.conv3 = nn.Conv2d(16,32, kernel_size=4,padding='valid')
        self.conv4 = nn.Conv2d(32,64, kernel_size=4,padding='valid')
        self.fc1 = nn.Linear(576, 32)
        self.fc2 = nn.Linear(32, 10)
        self.max_pool = nn.MaxPool2d(2,2)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.max_pool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = self.max_pool(F.relu(self.conv4(x)))

        x = x.view(-1,1152//2)
        x = (F.relu(self.fc1(x)))
        x = self.fc2(x)
        return x

model = MediumNet().to(device)
summary(model, (3,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 8, 29, 29]             392
            Conv2d-2           [-1, 16, 26, 26]           2,064
```

```
         MaxPool2d-3          [-1, 16, 13, 13]               0
            Conv2d-4          [-1, 32, 10, 10]           8,224
            Conv2d-5            [-1, 64, 7, 7]          32,832
         MaxPool2d-6            [-1, 64, 3, 3]               0
            Linear-7                  [-1, 32]          18,464
            Linear-8                  [-1, 10]             330
================================================================
Total params: 62,306
Trainable params: 62,306
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.21
Params size (MB): 0.24
Estimated Total Size (MB): 0.46
----------------------------------------------------------------
```

[7]:
```python
class LargeNet(nn.Module):
    '''
    Large Network Architecture
    Input: 32x32x3
    Output: 10
    Total Params: ~100k
    '''
    def __init__(self):
        super(LargeNet, self).__init__()
        self.fc1 = nn.Linear(256, 128)
        self.fc2 = nn.Linear(128, 10)
        self.max_pool = nn.MaxPool2d(2,2)

        self.block1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding = 'valid'),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.Conv2d(16, 16, kernel_size=3,padding='same'),
            nn.BatchNorm2d(16),
            nn.ReLU(),
        )
        self.block2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3,padding='valid'),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32,32, kernel_size=3,padding='same'),
            nn.BatchNorm2d(32),
            nn.ReLU(),
        )
        self.block3 = nn.Sequential(
```

```
            nn.Conv2d(32,64, kernel_size=3,padding='valid'),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64,64, kernel_size=3,padding='same'),
            nn.BatchNorm2d(64),
            nn.ReLU(),
        )
        self.block4 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3,padding='valid'),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128,128, kernel_size=3,padding='same'),
            nn.BatchNorm2d(128),
            nn.ReLU(),
        )
        self.softmax = nn.Softmax()

    def forward(self, x):
        x = self.max_pool(self.block1(x))
        x = self.max_pool(self.block2(x))
        x = self.max_pool(self.block3(x))
        x = x.view(-1,256)
        x = (F.relu(self.fc1(x)))
        x = self.fc2(x)
        return x

model = LargeNet().to(device)
summary(model, (3,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 30, 30]             448
       BatchNorm2d-2           [-1, 16, 30, 30]              32
              ReLU-3           [-1, 16, 30, 30]               0
            Conv2d-4           [-1, 16, 30, 30]           2,320
       BatchNorm2d-5           [-1, 16, 30, 30]              32
              ReLU-6           [-1, 16, 30, 30]               0
         MaxPool2d-7           [-1, 16, 15, 15]               0
            Conv2d-8           [-1, 32, 13, 13]           4,640
       BatchNorm2d-9           [-1, 32, 13, 13]              64
             ReLU-10           [-1, 32, 13, 13]               0
           Conv2d-11           [-1, 32, 13, 13]           9,248
      BatchNorm2d-12           [-1, 32, 13, 13]              64
             ReLU-13           [-1, 32, 13, 13]               0
        MaxPool2d-14             [-1, 32, 6, 6]               0
           Conv2d-15             [-1, 64, 4, 4]          18,496
```

```
      BatchNorm2d-16             [-1, 64, 4, 4]             128
             ReLU-17             [-1, 64, 4, 4]               0
           Conv2d-18             [-1, 64, 4, 4]          36,928
      BatchNorm2d-19             [-1, 64, 4, 4]             128
             ReLU-20             [-1, 64, 4, 4]               0
        MaxPool2d-21             [-1, 64, 2, 2]               0
           Linear-22                  [-1, 128]          32,896
           Linear-23                   [-1, 10]           1,290
================================================================
Total params: 106,714
Trainable params: 106,714
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.99
Params size (MB): 0.41
Estimated Total Size (MB): 1.41
----------------------------------------------------------------
```

## 4 Utility Functions to help Experimentation

[8]:
```
def model_chooser(size):
  '''
  Function to choose the model architecture.
  '''
  if size == 'small':
    return SmallNet().to(device)
  elif size == 'medium':
    return MediumNet().to(device)
  elif size == 'large':
    return LargeNet().to(device)

def opt_chooser(model, opt_name, lr, momentum):
  '''
  Function to choose the optimizer.
  Supports Adam and SGD.
  '''
  if opt_name == 'adam':
    return optim.Adam(model.parameters(), lr=lr)
  elif opt_name == 'sgd':
    return optim.SGD(model.parameters(), lr=lr, momentum =␣
 ↪momentum,weight_decay = 2e-3)

def loss_chooser(loss_name):
  '''
  Function to choose the loss function.
  Supports Cross Entropy.
```

```
  '''
  if loss_name == 'cross_entropy':
    return nn.CrossEntropyLoss()

def get_model_full_info(size, opt_name, lr, loss_name, momentum):
  '''
  Function to choose the model, optimizer, and loss function.
  Returns the model, optimizer, and loss function.
  '''
  model = model_chooser(size)
  print(summary(model, (3,32,32)))
  opt = opt_chooser(model, opt_name, lr,momentum)
  loss = loss_chooser(loss_name)

  return model, opt, loss
```

## 5 Running the entire dala loading,training pipeline

[ ]:
```
steps = [int(1*x) for x in range(1,100)] # Steps at which validation accuracies␣
 ↪are needed
trainloader, valloader, testloader, classes = get_cifar_data(batch_size = 32) #␣
 ↪Gets the data
model, opt, loss = get_model_full_info('large', 'sgd', 0.01,␣
 ↪'cross_entropy',momentum = 0.9) #Gets the model, optimizer and loss function␣
 ↪based on inputs
val_summary = train(model, opt, loss, trainloader, valloader,100, steps =␣
 ↪steps, scheduler=None) # Returns the validation summary based on training
```