CV HW5: Image Classifier

tags: `computer` `vision`

資料工碩　309551178　秦紫頤
電資學士班　610021　鄭伯俞
清華　H092505　翁紹育

Introduction

In this homework, we are going to implement three different ways of image classification. There are total 15 categories of scenes. OUr task is to predict the right class of scenes.

Implementation Procedure

1. Tiny images representation + nearest neighbor classifier

Resize

We simply resize each image into a small resolution, in this case we choose 16x16. Thus the high frequency part is discarded.

Standardize

To reach better performance, we apply standardization by substract each value with mean then divided by standard deviation.

KNN

For nearest neighbor classifier, we calculate the distance between testing data and training data, sort them in accending order and vote with k nearest neighbor to get the candidate. And it does need to pretrain to get the result.

2. Bag of SIFT representation + nearest neighbor classifier

Find SIFT Features

First, we read each image in **gray scale** because we don't need color information in all following steps.
Then, We use `cyvfleat.sift.dsift` and to find feature descriptors from each input and test image. **dsift** is the dense version of SIFT, where we extract feature points with certain sampling density controled by `step` parameter.

```
def detect_interest_p(img):
    _, desc = vlfeat.sift.dsift(img,step=10,fast=True)
    return desc

train_features = [detect_interest_p(x) for x in train_imgs]
test_features = [detect_interest_p(x) for x in test_imgs]
```

Run K-means algorithm on SIFT discriptors

In order to find the vocabularies in **Bag-of-words** model, we run K-means clustering algorithm on the feature descriptors. The parameter of K-means is set as follows: `k=150, initialization= "k-means++"`.

The algorithm is implemented by our own, and the code is provided here:

```
class k_means:
    def __init__(self, k, data):
        self.iter = 0
        self.converge = False
        self.k = k
        self.N = data.shape[0]
        self.d = np.expand_dims(data,axis=1)
        self.data = data
        self.centers = None
        self.labels = None

    def initialize_centers(self):
        #initialize centers with "k-means++" method
        centers = []
        #self.centers = np.empty(shape=(0,self.data.shape[1]))
        c1 = self.data[ np.random.choice(self.N)]
        centers.append(c1)
        for cnt in range(self.k - 1):
            print(f'initializing {cnt}th center')
            c = np.array(centers)
            c = np.expand_dims(c,axis=0)
            p = np.min(np.sum( (self.d - c)**2, axis=2), axis=1)
            p = p/ np.sum(p)
            new_center = self.data[np.random.choice(self.N, p=p),:]
            centers.append(new_center)
        self.centers = np.stack(centers,axis=0)
        self.find_labels()

    def find_labels(self):
        c = np.expand_dims(self.centers,axis=0)
        self.labels = np.argmin(np.sum((self.d - c) ** 2, axis=2),axis=1)
        return
    def find_centers(self):
        for i in range(self.k):
            self.centers[i] = np.mean(self.data[self.labels==i],axis=0)

    def one_step(self):
        l = self.labels.copy()
        self.find_centers()
        self.find_labels()
        if np.alltrue(l == self.labels):
            print('k-means converge!!')
            return True
        else :
            self.iter += 1
            return False

    def run_all(self,maxIter):
        for i in range(maxIter):
            print(f'starting iteration: {self.iter}')
            if self.one_step():
                break

    def classify(self,data):
        c = np.expand_dims(self.centers,axis=0)
        d = np.expand_dims(data,axis=1)
        labels = np.argmin(np.sum((d - c) ** 2, axis=2),axis=1)
        return labels
```

K-means converges after around iterations.

In practice, our implementation of k-means runs very slow, so we use `scipy.cluster.vq.kmeans2` to compute the centroids, then feed the resulting centroids into the k_means class.

```
k= 150
km = k_means(k,fs)
# My implementation runs very slow, use scipy.cluster is faster

#km.initialize_centers(method='random')
#km.run_all(1000)

from scipy.cluster.vq import *
codebook, distortion = kmeans2(fs,k, iter=1)
km.centers = codebook
```

Bag Of SIFT features

The resulting centers of k-means are used to classify the SIFT descriptors of testing images. We transform each image into a 150-element BOW vector.

```

train_bags = []
for f in train_features:
    words = km.classify(f)
    hist= [Counter(words)[i] for i in range(k)]
    train_bags.append(hist)
```

```
test_bags = []
for f in test_features:
    words = km.classify(f)
    hist= [Counter(words)[i] for i in range(k)]
    test_bags.append(hist)

train_bags = np.array(train_bags,dtype='float')
test_bags = np.array(test_bags,dtype='float')
```

KNN classifier

The implementation of KNN is modified from this tutorial

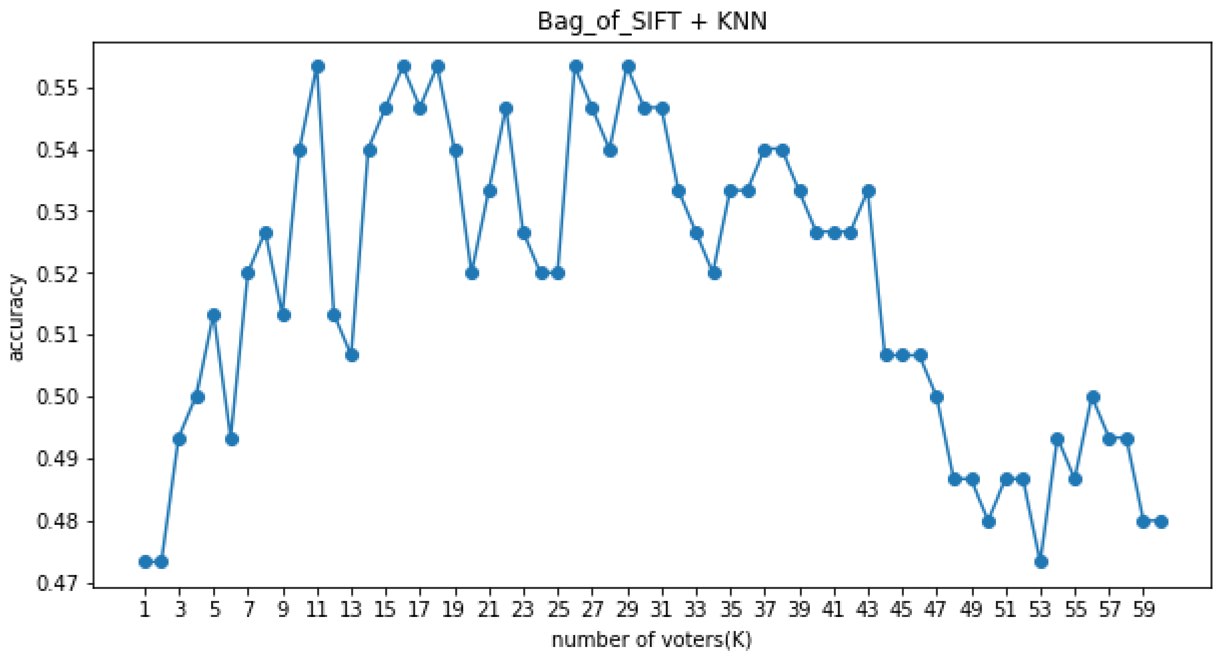
For each input bag-of-SIFT feature of testing image, we find its nearest k neighbour in euclidean norm. Then use the training labels of these neighbours to vote for the predicted label.

Another approch is to first normalize the bag-of-SIFT, then use inner product as similarity measure, we found out that these two approch gives similar result in KNN.

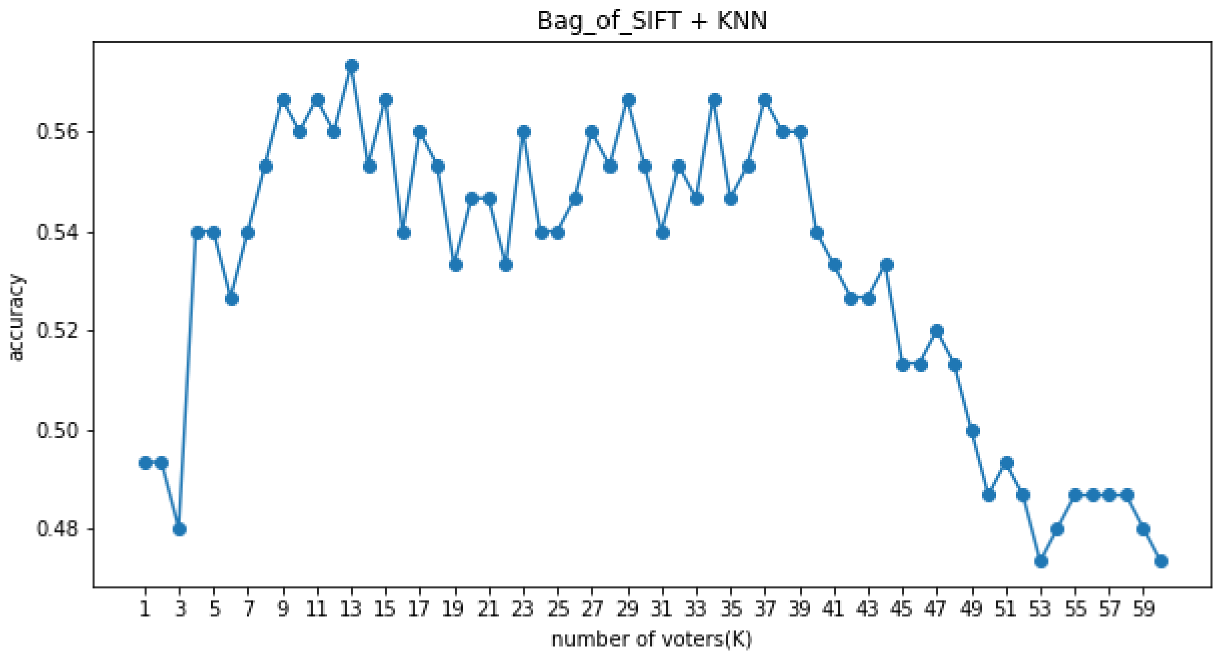
```
from scipy.spatial import distance
class KNN:
    def __init__(self, k):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X_test):
        final_output = []
        for i in range(len(X_test)):
            d = []
            votes = []
            for j in range(len(self.X_train)):

                dist = distance.euclidean(self.X_train[j] , X_test[i])
                #alternative approch
                #dist = -self.X_train[j] @ X_test[i]
                d.append([dist, j])
            d.sort()
            d = d[0:self.k]
            for d, j in d:
                votes.append(self.y_train[j])
            ans = Counter(votes).most_common(1)[0][0]
            final_output.append(ans)

        return final_output

    def score(self, X_test, y_test):
        predictions = self.predict(X_test)
        cnt = 0
        for i in range(len(y_test)):
            if predictions[i] == y_test[i]:
                cnt += 1
        return cnt/ len(y_test)
```

The result of different k is plotted.

Euclidean Norm
best Acc = 55.3%
best k = 10



Inner Product Similarity (first normalize the BOS features)
best Acc = 57.3%
best k = 12



---

3. Bag of SIFT representation + linear SVM classifier

We use `libsvm` to classify Bag-of-SIFT descriptor. We first try different kernels with default parameter, then do a grid search on γ and C of RBF kernel.

```
import libsvm.svmutil as svm
```

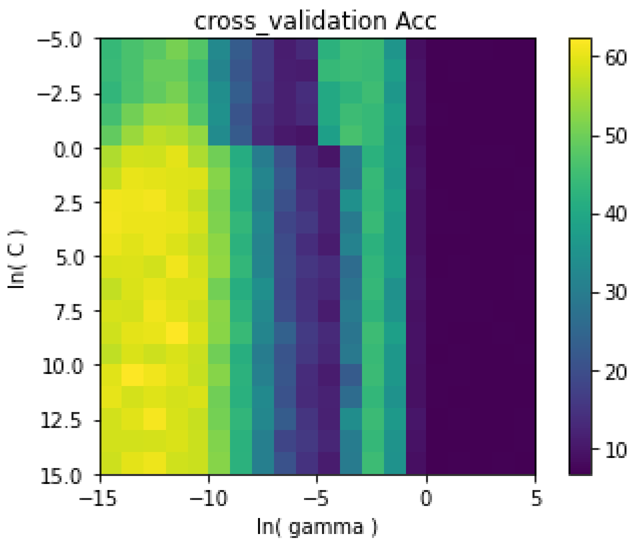
```
# Linear kernel
m_Linear = svm.svm_train(train_labs,train_bags,'-s 0 -t 0')
_,acc_Linear ,_  = svm.svm_predict(test_labs,test_bags,m_Linear)

#polynomial kernel, with coef0 = 1, degree = 3
m_Poly = svm.svm_train(train_labs,train_bags,'-s 0 -t 1 -r 1')
_,acc_Poly ,_  = svm.svm_predict(test_labs,test_bags,m_Poly)

# RBF kernel with C = 1 , gamma = 1/k
m_RBF = svm.svm_train(train_labs,train_bags,'-s 0 -t 2')
_,acc_RBF ,_  = svm.svm_predict(test_labs,test_bags,m_RBF)

#Sigmoid kernel coef0 = 0
m_Sigmoid = svm.svm_train(train_labs,train_bags,'-s 0 -t 3')
_,acc_Sigmoid ,_  = svm.svm_predict(test_labs,test_bags,m_Sigmoid)
```

Testing Accuracy for each kernel

(With default parameters)

Linear	Polynomial	RBF	Sigmoid
63.3%	67.3%	15.3%	11.3%

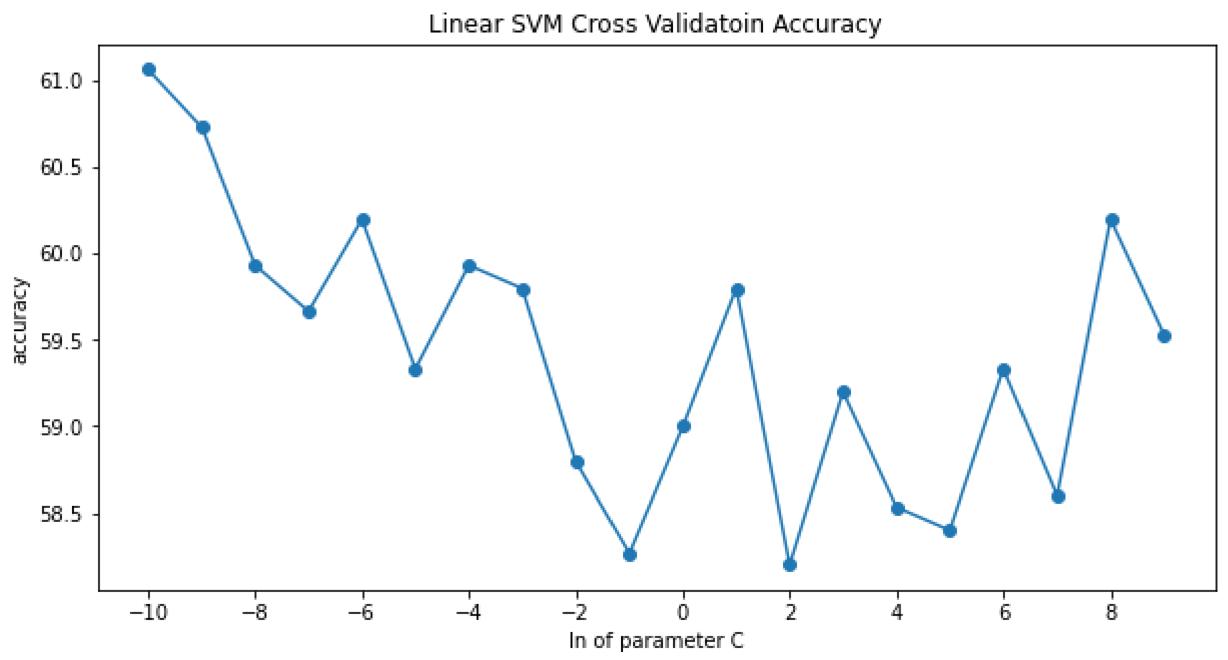
Do grid search on parameters γ and C of RBF kernel



We choose `gamma = 2**8` and `C = 2**-12` as final parameter. The Accuracy on testing set is **68.6667%**.

Search Parameter C for Linear SVM

We search for parameter C in linear SVM and plot cross-validation accuracy against $ln(C)$ in range `2**-10` ~ `2**10`



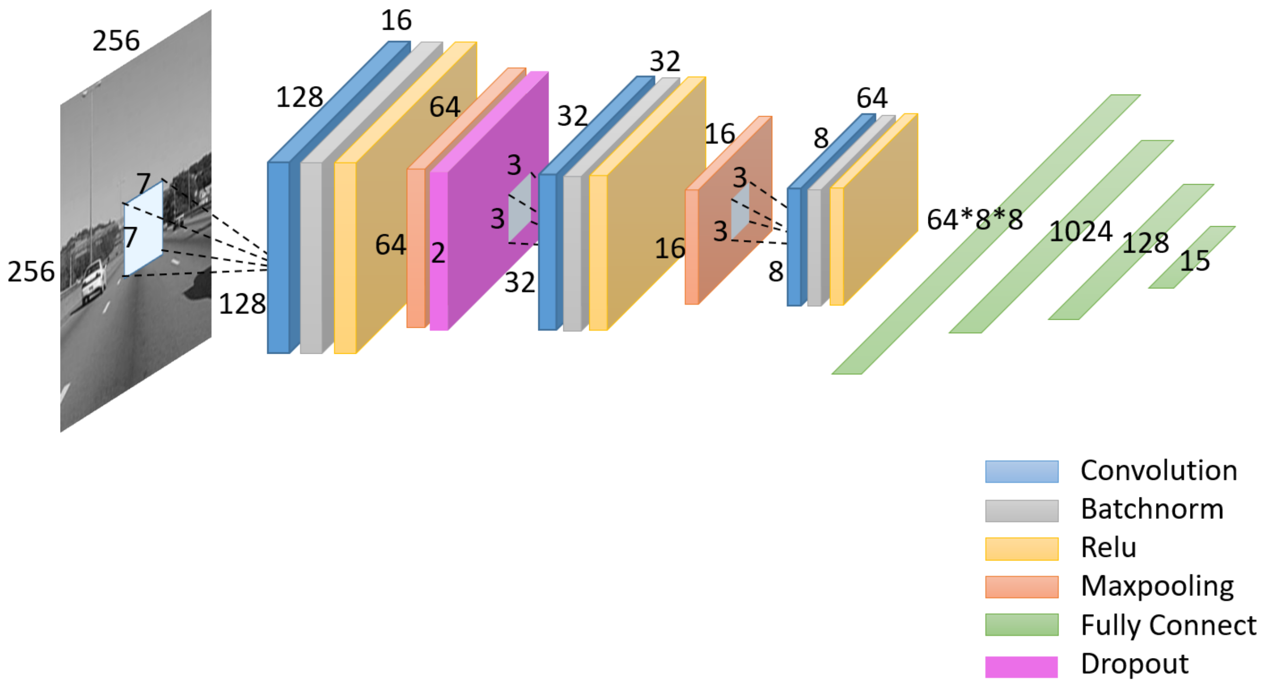
The testing accuracy with `C = 2 ** -10` is **68%**.

---

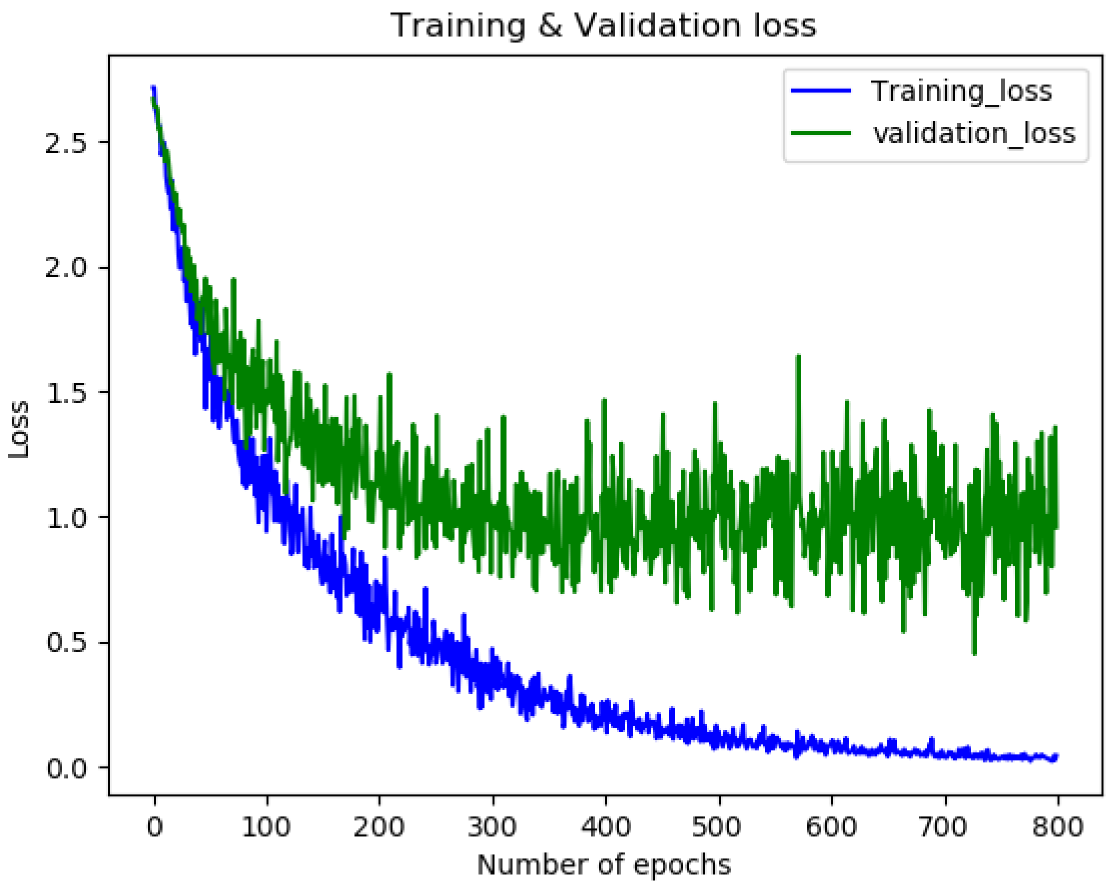
4. Bonus: Simple CNN

We build a simple CNN classifier shown as below. Input image size is 256x256. And use SGD as optimizer, loss function is crossentropy, batch size is 64 and learning rate is 0.001. Total training epochs is 800.
For data, we split 20% of training dataset as validation set and test it each epoch.

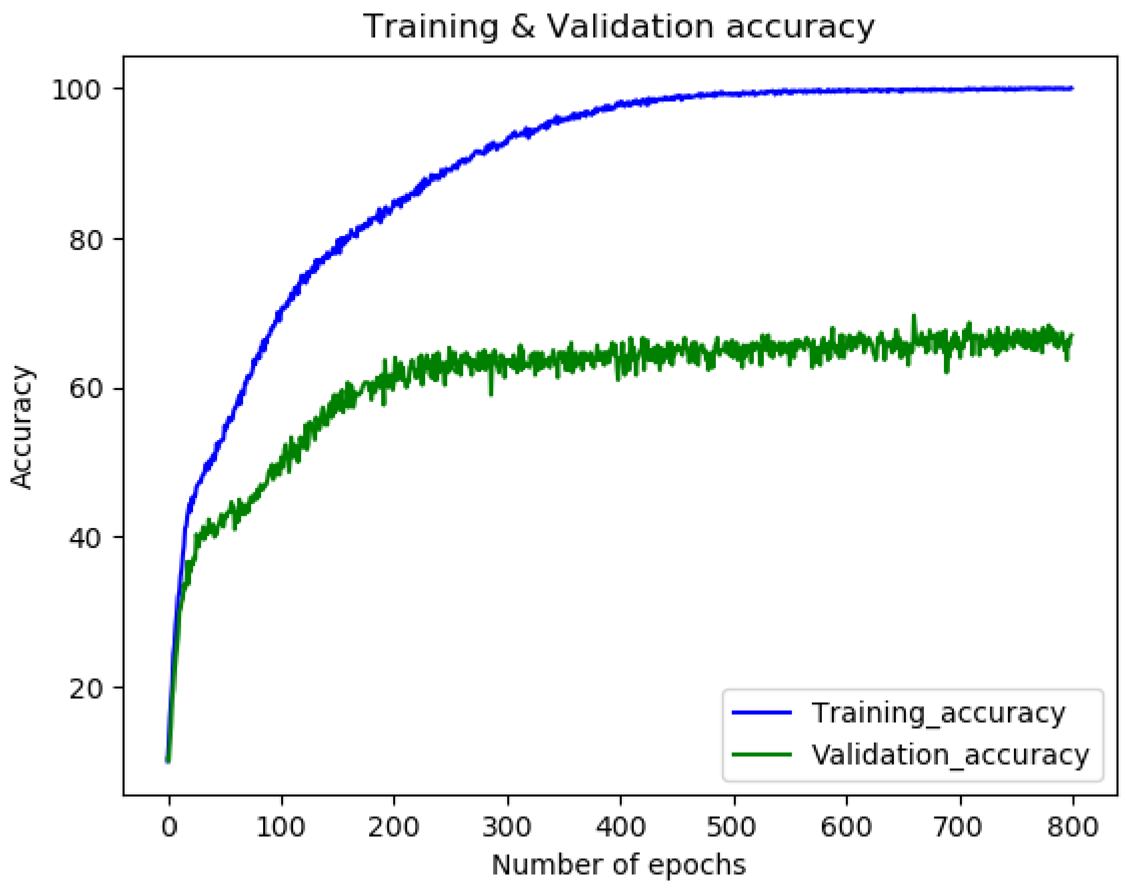
Model structure



Training & Validation loss curve



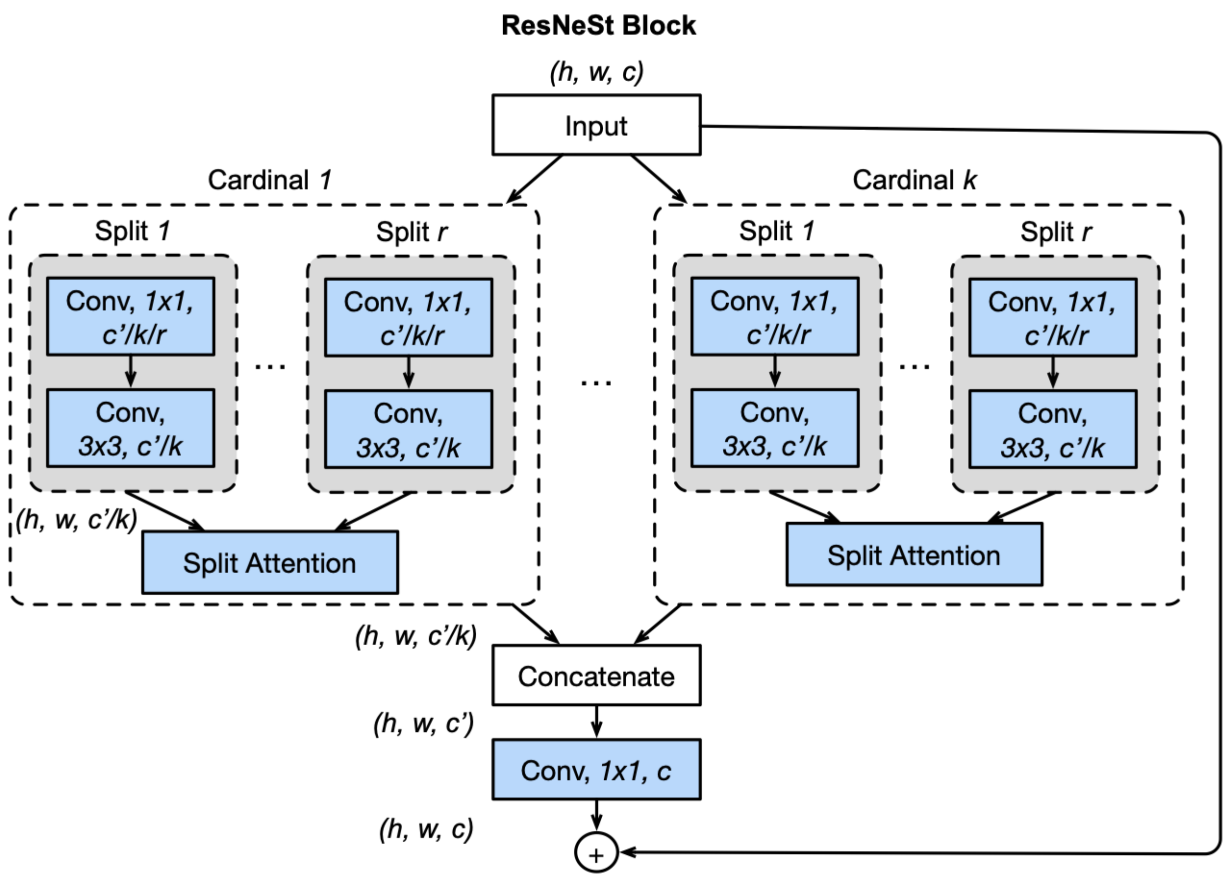
Training & Validation accuracy curve



5. Bonus: ResNest

ResNest is a variant on a ResNet, which instead stacks Split-Attention blocks. The cardinal group representations are then concatenated along the channel dimension: $V = Concat\{V^1, V^2 \dots V^K\}$. As in standard residual blocks, the final output Y of other Split-Attention block is produced using a shortcut connection: $Y = V + X$, if the input and output feature-map share the same shape. For blocks with a stride, an appropriate transformation $\mathcal{T}$ is applied to the shortcut connection to align the output shapes: $Y = V + \mathcal{T}(X)$. For example, $\mathcal{T}$ can be strided convolution or combined convolution-with-pooling.

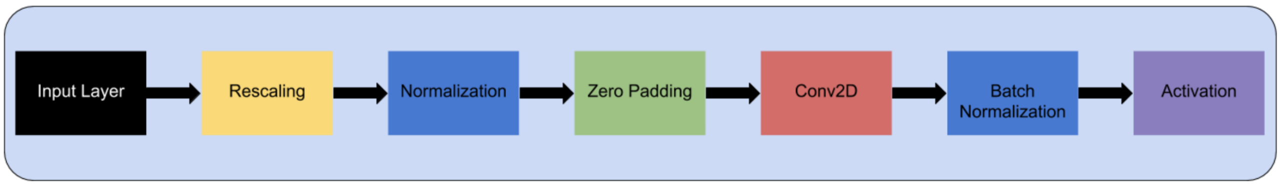
Model Structure



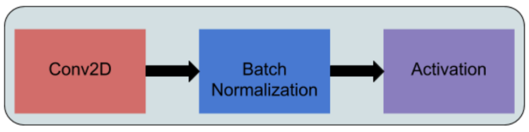
6. Bonus: EfficientNet

EfficientNet is a convolutional neural network architecture and scaling method that uniformly scales all dimensions of depth/width/resolution using a compound coefficient. Unlike conventional practice that arbitrary scales these factors, the EfficientNet scaling method uniformly scales network width, depth, and resolution with a set of fixed scaling coefficients. For example, if we want to use 2^N times more computational resources, then we can simply increase the network depth by α^N, width by β^N, and image size by γ^N, where α, β, γ are constant coefficients determined by a small grid search on the original small model. EfficientNet uses a compound coefficient ϕ to uniformly scales network width, depth, and resolution in a principled way. The compound scaling method is justified by the intuition that if the input image is bigger, then the network needs more layers to increase the receptive field and more channels to capture more fine-grained patterns on the bigger image.

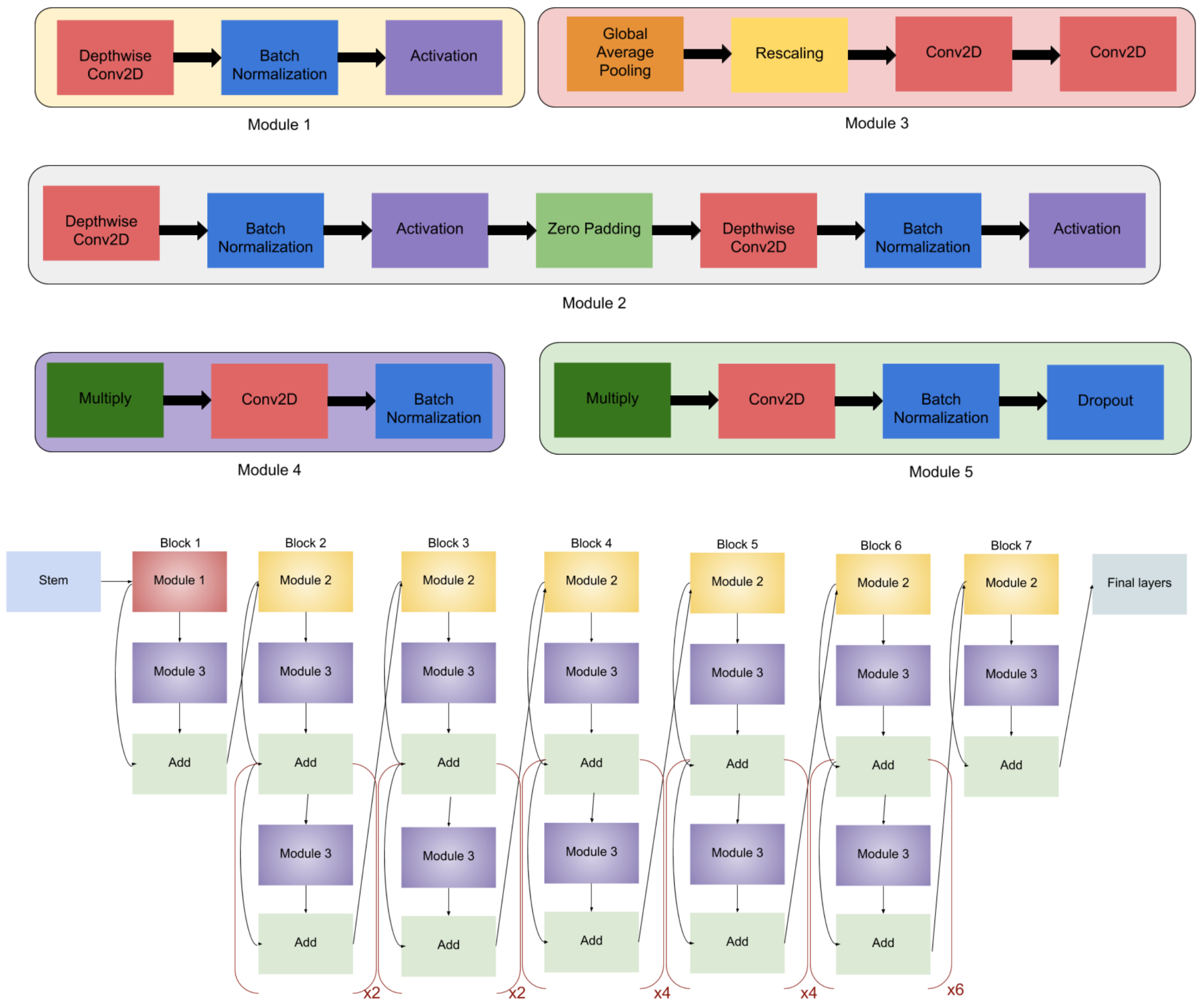
Model Structure



Stem

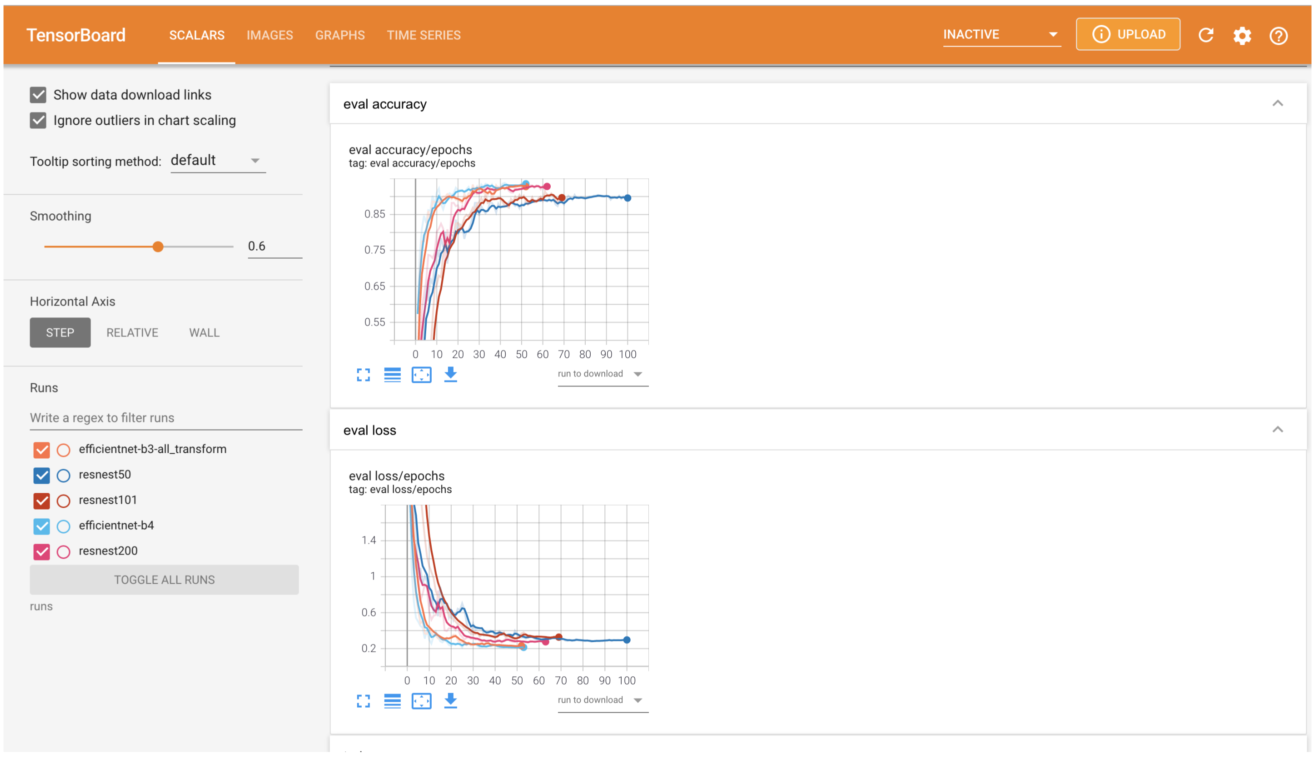


Final Layers



Architecture for EfficientNet-B4

Accuray and Loss Curve of ResNest and EfficientNet


TensorBoard SCALARS IMAGES GRAPHS TIME SERIES INACTIVE UPLOAD

- [✓] Show data download links
- [✓] Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis [✓ STEP] RELATIVE WALL

Runs

Write a regex to filter runs

- [✓] efficientnet-b3-all_transform
- [✓] resnest50
- [✓] resnest101
- [✓] efficientnet-b4
- [✓] resnest200

TOGGLE ALL RUNS

runs

eval accuracy

eval accuracy/epochs
tag: eval accuracy/epochs

eval loss

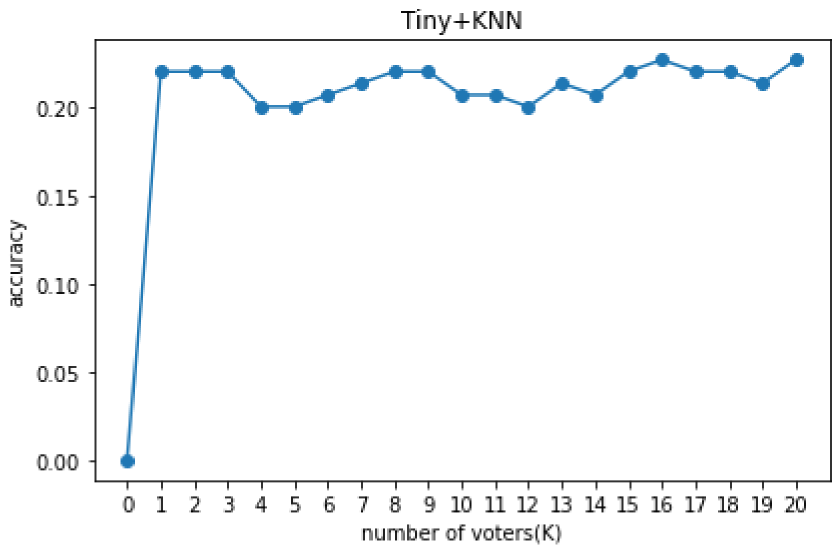
eval loss/epochs
tag: eval loss/epochs

Experiment result

1. Tiny images representation + nearest neighbor classifier

K	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Acc	22	22	22	20	20	20.67	21.33	22	22	20.67

K	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Acc	20.67	20	21.33	20.67	22	22.67	22	22	21.33	22.67

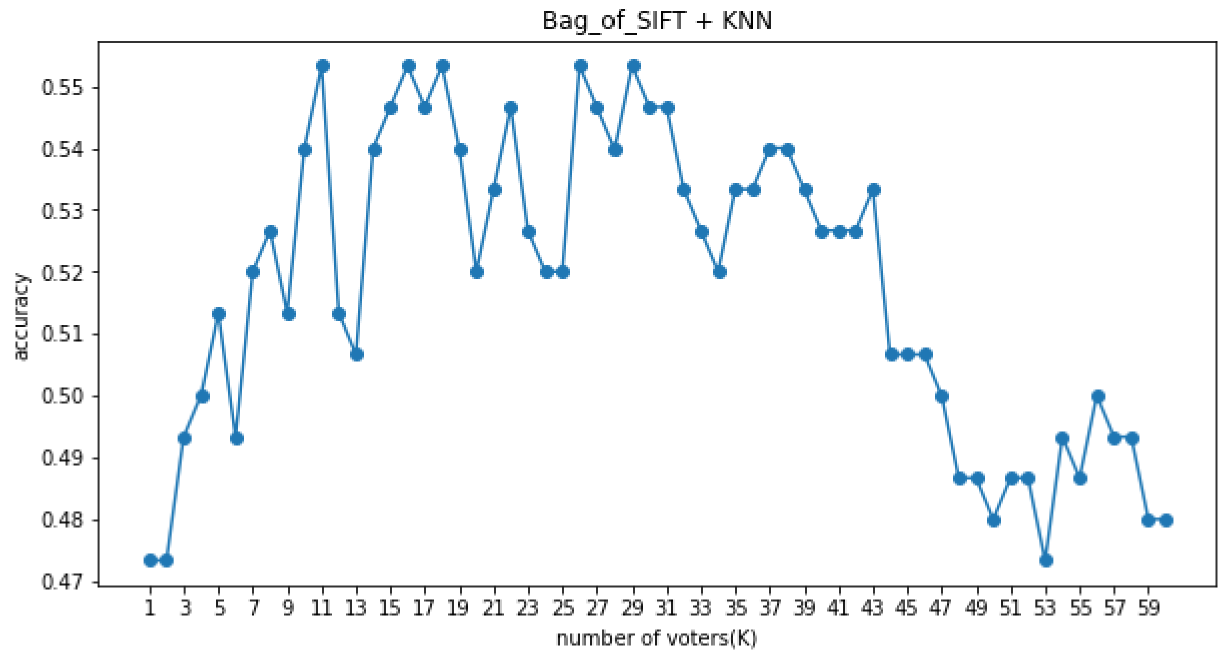


The best accuracy = 22.67% happens when k = 16 and 20.

2. Bag of SIFT representation + nearest neighbor classifier

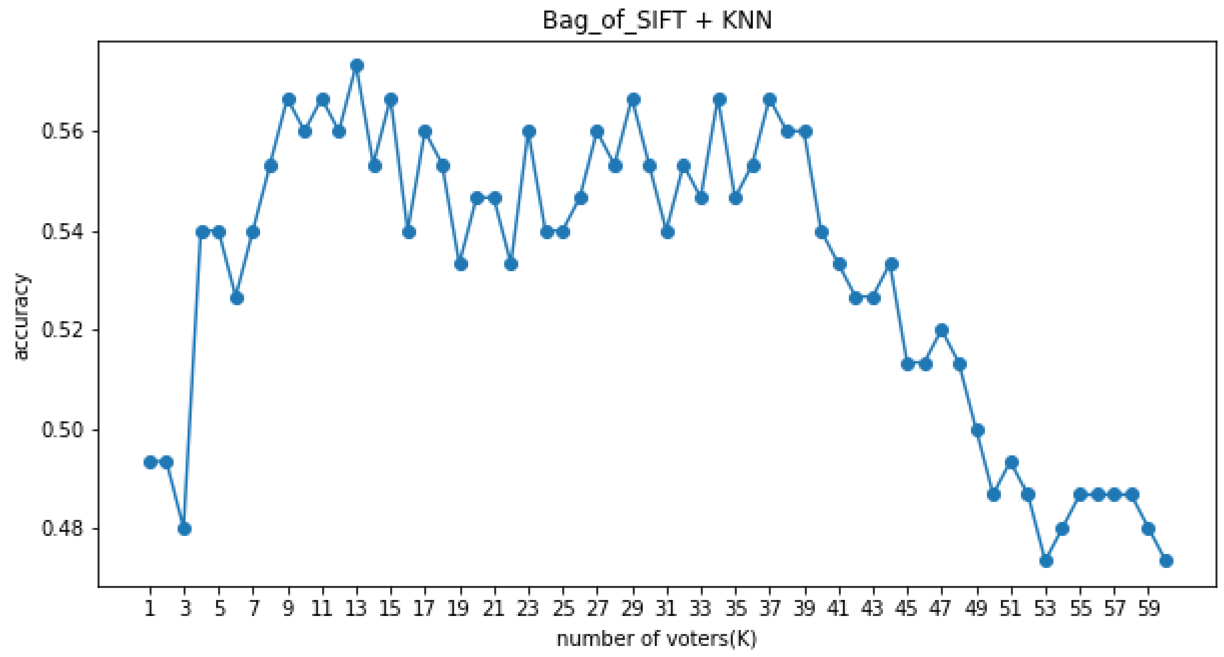
Euclidean Norm

best Acc = 55.3%
best k = 10



Inner Product Similarity (first normalize the BOS features)

best Acc = 57.3%
best k = 12



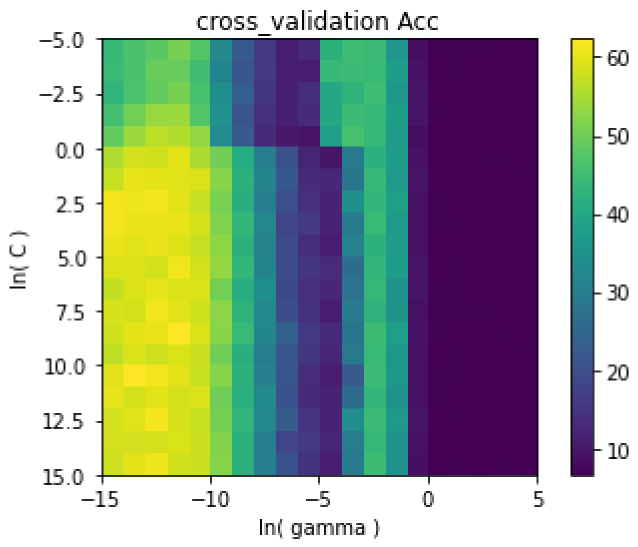
3. Bag of SIFT representation + linear SVM classifier

Testing Accuracy for each kernel

(With default parameters)

Linear	Polynomial	RBF	Sigmoid
63.3%	67.3%	15.3%	11.3%

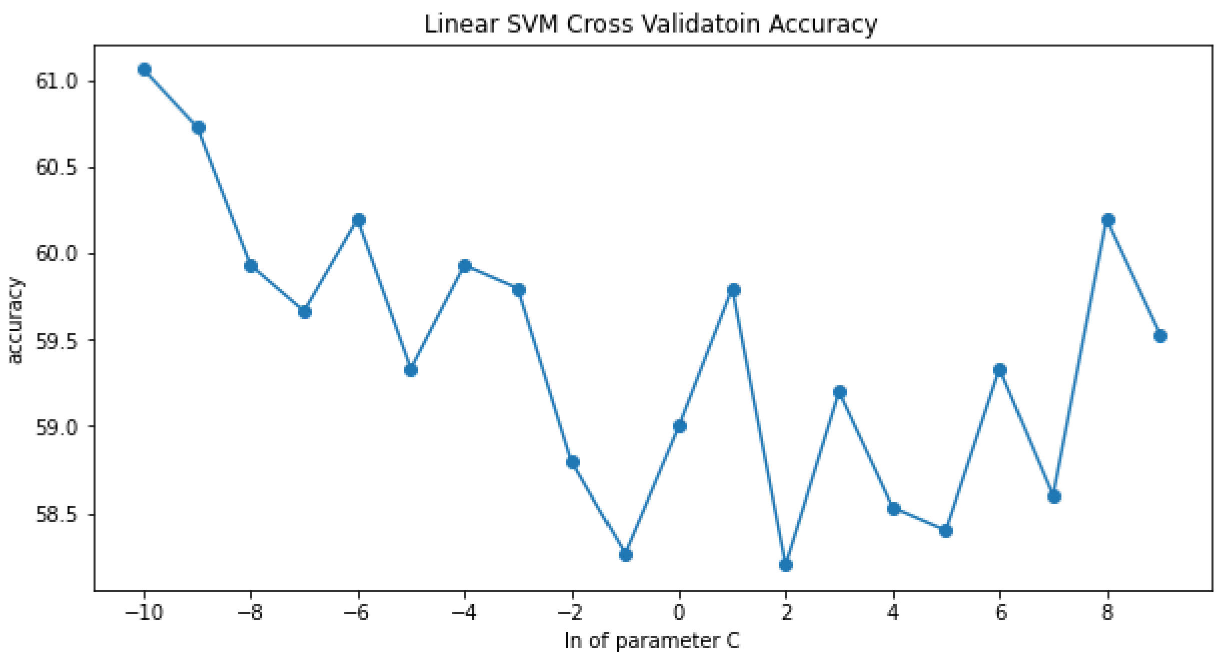
Do grid search on parameters γ and C of RBF kernel



We choose `gamma = 2**8` and `C = 2**-12` as final parameter. The Accuracy on testing set is **68.6667%**.

Search Parameter C for Linear SVM

We search for parameter C in linear SVM and plot cross-validation accuracy against $ln(C)$ in range `2**-10` ~ `2**10`



The testing accuracy with `C = 2 ** -10` is **68%**.

4. Bonus: CNN

Test accuracy = 74%

5. Bonus: ResNest

- ResNest50 accuracy: 88.72%
- ResNest101 accuracy: 90.67%
- ResNest200 accuracy: 93.11%

6. Bonus: EfficientNet

- EfficientNet-b3 accuracy: 93.56%
- EfficientNet-b4 accuracy: 93.78%

Discussion

In the Bag-of-SIFT part, we have the following discussions

- Normal SIFT setting is not enough, dense SIFT is needed. Dense SIFT tries to extract feature from every location, while SIFT only extract features from those locations determined by Lowe's algorithm. So dense SIFT provides more features from each image. If we use normal SIFT features, after all the optimization, best acc for KNN is only 44%, best acc for SVM is only 48%.

- Normalizing the BOS vector doesn't improve the result. So we only do it if we use inner product as similarity measure in KNN. The effect of normalize usually improves the result of classifier, however, in this case of BOS, the length of the histogram may provide additional information about the input picture (ie, the number of points that passes the SIFT threshold).
- Our original implementaion of k-means algorithm utilize the **broadcasting** behaviour of `numpy.arrays` to calculate distances between each data points and each kernels. For example, when finding new labels for each data points based on their distances to each center, we write:

  ```
  d = np.expand_dims(self.data,axis=1)
  c = np.expand_dims(self.centers,axis=0)
  self.labels = np.argmin(np.sum((d - c) ** 2, axis=2),axis=1)
  ```

  This method runs about 1.5~2 times faster than naive for-loop method because numpy speeds up array operations. However, it requires storing intermediate result in a large array with `shape=(N, k, d)`, where `N` is the number of input points, `k` is the number of clusters, `d` is the dimension of each data points. In our case, N=965994, k=150, d=128, the array takes about 8 * 128 * 150 * 965994 bytes = 138 Gb. Which is too large for my laptop.

  For-loop method runs pretty well, but is very slow (I estimate that it would take 4~6hr to finish k-means task in above mentioned settings). I decided to use external libraries for this homework, and finally find a nice implementation in `scipy.cluster`.
- Since the dimension and number of clusters is very large in our case, the result quality of k-means is unstable (sometimes we may encounter empty class), when the quality is bad, the acc of both KNN and SVM drops about 5~10%. In order to ensure the quality of the result, we run multiple trials of k-means and select the result that has the least energy. This is done by setting the `iter` variable of `scipy.cluter.kmeans` function. Another approch to use better initialization method such as 'k-means++', this can be done with setting the `minit` parameter of `scipy.cluster.kmeans2` function.
- The performance of SVM is sensitive to parameters, so we use grid search and cross-validation to tune some of the parameters and get better accuracy on testing data.

In the bonus part, we used three models: simple CNN, ResNest, and EfficientNet to do classification. EfficientNet is the state-of-the-art classification model; although it trains slow, the results are the best. Furthermore, ResNest trains faster and got high accuracy, but still cannot beat the robustness of EfficientNet. We think the accuracy of ResNest and EfficientNet is higher than other methods is because:

1. Both EfficientNet and ResNest are state of the art classification deep learning model
2. We have done many augmentations (affine, grayscale, horizontal flip, vertical flip, color jitter, shift, color transform rotation). This is important since our data is not a lot. If we do not want our model to overfitting easily, augmentations are a must.

Conclusion

In this homework, we learn the technique of Bag-of-Word method in the problem of image recognition. We also learn the basics of three different machine learning methods for clusttering and classification: **k-means** , **KNN** ,and **SVM** . In the bonus part, we learned 3 different network models and compare their performance: **CNN**, **ResNest**, and **EfficientNet** . In general, this is homework is a good practice for us, and we have improved our problem solving skills during the process.