Final Project

April 26, 2024

Nick Cantalupa and Sean Duffy

In this project we explore 4 models for image classification (on the CIFAR100 dataset). We trained a logistic regression model, a linear SVM, kernel SVM, and CNN. The code and accuracies for each of the models can be seen below.

[ ]:
```
from sklearnex import patch_sklearn
patch_sklearn()
import torch
from torchvision.datasets import CIFAR100
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

[2]:
```
ROOT_PATH='data'

BATCH_SIZE=1000

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

train_dataset = CIFAR100(root=ROOT_PATH, download=True, train=True,␣
↪transform=transform)
test_dataset = CIFAR100(root=ROOT_PATH, train=False, transform=transform)

train_data_loader = DataLoader(dataset=train_dataset, num_workers=4,␣
↪batch_size=BATCH_SIZE, shuffle=True)
test_data_loader = DataLoader(dataset=test_dataset, num_workers=4,␣
↪batch_size=BATCH_SIZE, shuffle=False)
```

```
Files already downloaded and verified
```

1 Logistic Regression

We do not expect Logistic Regression to be a great application of ML in this scenario but it will be good to check the accuracy to get a baseline.

We first are going to convert the Tensor of data into 2 dimensions so we can run Logistic regression on it for training the model.

[ ]:
```
from sklearn.linear_model import LogisticRegression
```

[ ]:
```
for train_data, train_labels in train_data_loader:
    print(train_data.shape)
    print(train_labels.shape)
    break

for eval_data, eval_labels in test_data_loader:
    print(eval_data.shape)
    print(eval_labels.shape)
    break
```

[1]:
```
def flatten(img_tensor):
    flattened_data = []
    for i in range(len(img_tensor)):
        if (i%1000 == 0):
            print(round((i/len(img_tensor))*100, 4),"%", end= "\r")
        flat = img_tensor[i].flatten()
        flat = flat.numpy()
        flattened_data.append(pd.Series(flat))

    return pd.DataFrame(flattened_data)
```

[ ]:
```
X_train = flatten(train_data)
y_train = train_labels

X_eval = flatten(eval_data)
y_eval = eval_labels

lr_model = LogisticRegression(multi_class='multinomial', solver='lbfgs',␣
↪max_iter=300)
lr_model.fit(X_train, y_train)
```

[ ]:
```
y_pred = lr_model.predict(X_eval)
eval_accuracy = accuracy_score(y_eval, y_pred)
print("Accuracy:", eval_accuracy)
```

This is a decent accuracy given that Logisitic Regression is not meant to handle this many classes effectively. Since this model hit the max iterations, we will see if performing PCA on it to reduce dimensinoality can improve the performance. We can do this since we saw above that the explained variance stays very high after reducing dimensions significantly (see appendix). This should help speed up our algorithm and make it easier to run.

```
[ ]: from sklearn.decomposition import PCA
     from sklearn.preprocessing import StandardScaler
     pca = PCA(random_state=41)
     scaler = StandardScaler()
     X_train_reduced = pca.fit_transform(scaler.fit_transform(X_train))
     X_train_reduced = X_train_reduced[:,:250]
     X_eval_reduced = pca.transform(X_eval)
     X_eval_reduced = X_eval_reduced[:,:250]
```

```
[ ]: lr_reduced_model = LogisticRegression(multi_class='multinomial', solver='lbfgs')
     lr_reduced_model.fit(X_train_reduced, y_train)
```

```
[ ]: y_reduced_pred = lr_reduced_model.predict(X_eval_reduced)
     eval_accuracy = accuracy_score(y_eval, y_reduced_pred)
     print("Accuracy:", eval_accuracy)
```

This PCA reduction slightly helped out the model and increases the accuracy to 15%. This is still not great but this is about what you can expect for a regression that focuses on binary classification. With CIFAR 100 this is still a good improvement from the baseline but we can hopefully increase our accuracy with other algorithms.

Final Logistic Regression Accuracy: 15.23%

2 Linear SVM

The data needs to be flattened from the tensors in order to be used in SVM. The function below loads the data and does this. The training set is loaded in as a train set and validation set, the test set is also loaded in.

```
[3]: def flatten(data_loader):
         images = []
         labels = []

         for img_chunk, label_chunk in data_loader:
             for img in img_chunk:
                 img = np.array(img)
                 img_flat = img.flatten()
                 images.append(img_flat)
             for label in label_chunk:
                 labels.append(label)

         images = np.array(images)
         labels = np.array(labels)
         return images, labels
```

[4]:
```
X, y = flatten(train_data_loader)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=41, stratify=y)
X_test, y_test = flatten(test_data_loader)
```

[7]:
```
from sklearn.svm import SVC
```

[10]:
```
linear_svm = SVC(C=1.0, kernel='linear', probability=False, random_state=41,
                 decision_function_shape='ovr', tol=0.001, max_iter= -1)
```

[11]:
```
linear_svm.fit(X_train, y_train)
y_pred_train = linear_svm.predict(X_train)
y_pred_val = linear_svm.predict(X_val)
train_accuracy = accuracy_score(y_train, y_pred_train)
val_accuracy = accuracy_score(y_val, y_pred_val)
print('Train acccuracy: ', train_accuracy)
print('Validation accuracy: ', val_accuracy)
```

```
Train acccuracy:  0.9996
Validation accuracy:  0.1556
```

Model is significantly overfitting. We'll now adjust the regularization parameter to address this. Due to the increased time and processing required for cross validations (5x), we'll assess performance via the single validation set.

[5]:
```
def fit_svm(X_train, X_val, y_train, y_val, C=1.0, max_iter=-1,␣
↪decision_function_shape='ovr', tol=0.001, kernel='linear'):
    linear_svm = SVC(C=C, kernel=kernel, probability=False, random_state=41,
                     decision_function_shape=decision_function_shape, tol=tol,␣
↪max_iter=max_iter)
    linear_svm.fit(X_train, y_train)
    y_pred_train = linear_svm.predict(X_train)
    y_pred_val = linear_svm.predict(X_val)
    train_accuracy = accuracy_score(y_train, y_pred_train)
    val_accuracy = accuracy_score(y_val, y_pred_val)
    print(f'C: {C}, tol: {tol}, max_iter: {max_iter},␣
↪{decision_function_shape}')
    print('Train acccuracy: ', train_accuracy)
    print('Validation accuracy: ', val_accuracy, '\n')
```

[14]:
```
for c in [0.1, 0.01, 0.001, 0.0001]:
    fit_svm(X_train, X_val, y_train, y_val, C=c, kernel='linear')
```

```
C: 0.1, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.930575
Validation accuracy:  0.1645

C: 0.01, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.485675
```

```
Validation accuracy:  0.1969

C: 0.001, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.262225
Validation accuracy:  0.1947

C: 0.0001, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.166925
Validation accuracy:  0.1598
```

With C = 0.01 as the regularization parameter, we get the best validation accuracy. Below a model with C=0.01 is fit and the accuracy assessed on the test set.

[8]:
```
fit_svm(X_train, X_test, y_train, y_test, C=0.01, kernel='linear')
```

```
C: 0.01, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.48135
Validation accuracy:  0.1936
```

2.0.1 Final Linear SVM Test Accuracy: 19.36%

3 Kernel SVM

In order to perform better than the linear SVM model above, we attempt kernel SVM. Below we test a few different values of C and both radial and polynomial kernels.

[17]:
```
fit_svm(X_train, X_val, y_train, y_val, C=1, kernel='rbf')
```

```
C: 1, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.470325
Validation accuracy:  0.2424
```

[18]:
```
fit_svm(X_train, X_val, y_train, y_val, C=0.1, kernel='rbf')
```

```
C: 0.1, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.189225
Validation accuracy:  0.1688
```

[20]:
```
fit_svm(X_train, X_val, y_train, y_val, C=10, kernel='rbf')
```

```
C: 10, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.89625
Validation accuracy:  0.2643
```

[8]:
```
fit_svm(X_train, X_val, y_train, y_val, C=100, kernel='rbf')
```

```
C: 100, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.999825
Validation accuracy:  0.2786
```

[19]:
```python
fit_svm(X_train, X_val, y_train, y_val, C=1, kernel='poly')
```

```
C: 1, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.513025
Validation accuracy:  0.1855
```

[9]:
```python
fit_svm(X_train, X_val, y_train, y_val, C=10, kernel='poly')
```

```
C: 10, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.9159
Validation accuracy:  0.2183
```

Based on the validation accuracies, the best model is one with a radial kernel and C = 100. Below is this models test accuracy.

[12]:
```python
best_kernel_svm = SVC(C=100, kernel='rbf', random_state=41,␣
 ↪decision_function_shape='ovr')
best_kernel_svm.fit(X, y)
y_pred_test = best_kernel_svm.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred_test)
print('Test acccuracy: ', test_accuracy)
```

```
Test acccuracy:  0.2906
```

3.0.1 Final Kernel SVM Accuracy: 29.06%

4 CNN

We will now try some different methods of using Convolutional Neural Networks to try to increase the accuracy of the models.

[1]:
```python
import torch
from torchvision.datasets import CIFAR100
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pickle
```

The cell below implements augmentation for the image data. This prevents overfitting and allows the model to better generalize. The inclusion of data augmentation allowed us to break 50%

accuracy with our CNN.

[10]:
```
ROOT_PATH = 'data'

BATCH_SIZE = 500

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

transform_augmentation = transforms.Compose([
    transforms.RandomHorizontalFlip(),  # Randomly flip images horizontally
    transforms.RandomRotation(10),       # Randomly rotate images by 10 degrees
    transforms.RandomCrop(32, padding=4),  # Randomly crop images with padding
    transforms.ToTensor(),               # Convert images to tensor format
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize images
])

train_dataset = CIFAR100(root=ROOT_PATH, download=True, train=True,␣
↪transform=transform_augmentation)
eval_dataset = CIFAR100(root=ROOT_PATH, train=False, transform=transform)

train_data_loader = DataLoader(dataset=train_dataset, num_workers=4,␣
↪batch_size=BATCH_SIZE, shuffle=True)
eval_data_loader = DataLoader(dataset=eval_dataset, num_workers=4,␣
↪batch_size=BATCH_SIZE, shuffle=False)
```

```
Files already downloaded and verified
```

[11]:
```
class ConvNN(torch.nn.Module):
    def __init__(self):
        super(ConvNN, self).__init__()
        self.conv1 = torch.nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.bn1 = torch.nn.BatchNorm2d(64)
        self.conv2 = torch.nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn2 = torch.nn.BatchNorm2d(128)
        self.pool1 = torch.nn.MaxPool2d(2, 2)
        self.dropout1 = torch.nn.Dropout(0.25)
        self.conv3 = torch.nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn3 = torch.nn.BatchNorm2d(256)
        self.pool2 = torch.nn.MaxPool2d(2, 2)
        self.dropout2 = torch.nn.Dropout(0.25)
        self.fc1 = torch.nn.Linear(8*8*256, 1024)
        self.bn4 = torch.nn.BatchNorm1d(1024)
        self.dropout3 = torch.nn.Dropout(0.5)
        self.fc2 = torch.nn.Linear(1024, 100)

    def forward(self, x):
```

```python
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.pool1(x)
        x = self.dropout1(x)
        x = F.relu(self.bn3(self.conv3(x)))
        x = self.pool2(x)
        x = self.dropout2(x)
        x = x.view(-1, 8*8*256)
        x = F.relu(self.bn4(self.fc1(x)))
        x = self.dropout3(x)
        x = self.fc2(x)
        return x
```

[25]:
```python
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

num_epochs = 20
learning_rate = 0.001
```

We found that loss plateaued at aroung 20 epochs

[26]:
```python
CNN_model = ConvNN().to(device)
```

[27]:
```python
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(CNN_model.parameters(), lr= learning_rate)
n_steps = len(train_data_loader)
```

[28]:
```python
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
optimizer = torch.optim.Adam(CNN_model.parameters(), lr=learning_rate)

for epoch in range(num_epochs):
    total_loss = 0
    for i, (images, labels) in enumerate(train_data_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = CNN_model(images)
        loss = loss_fn(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        if (i + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{n_steps}],␣
 ↪Loss: {loss.item():.4f}')
    avg_loss = total_loss / n_steps
    print(f'End of Epoch {epoch+1}, Average Loss: {avg_loss:.4f}')
    scheduler.step()
```

```
Epoch [1/20], Step [100/100], Loss: 3.2833
```

```
End of Epoch 1, Average Loss: 3.6223
Epoch [2/20], Step [100/100], Loss: 2.8808
End of Epoch 2, Average Loss: 3.0146
Epoch [3/20], Step [100/100], Loss: 2.5716
End of Epoch 3, Average Loss: 2.7093
Epoch [4/20], Step [100/100], Loss: 2.4733
End of Epoch 4, Average Loss: 2.5372
Epoch [5/20], Step [100/100], Loss: 2.3421
End of Epoch 5, Average Loss: 2.3929
Epoch [6/20], Step [100/100], Loss: 2.1844
End of Epoch 6, Average Loss: 2.2884
Epoch [7/20], Step [100/100], Loss: 2.3236
End of Epoch 7, Average Loss: 2.2037
Epoch [8/20], Step [100/100], Loss: 2.2170
End of Epoch 8, Average Loss: 2.1348
Epoch [9/20], Step [100/100], Loss: 1.9838
End of Epoch 9, Average Loss: 2.0714
Epoch [10/20], Step [100/100], Loss: 1.9912
End of Epoch 10, Average Loss: 2.0088
Epoch [11/20], Step [100/100], Loss: 1.9953
End of Epoch 11, Average Loss: 1.9536
Epoch [12/20], Step [100/100], Loss: 1.8299
End of Epoch 12, Average Loss: 1.9119
Epoch [13/20], Step [100/100], Loss: 1.8398
End of Epoch 13, Average Loss: 1.8570
Epoch [14/20], Step [100/100], Loss: 1.7821
End of Epoch 14, Average Loss: 1.8275
Epoch [15/20], Step [100/100], Loss: 1.8329
End of Epoch 15, Average Loss: 1.7782
Epoch [16/20], Step [100/100], Loss: 1.7297
End of Epoch 16, Average Loss: 1.7454
Epoch [17/20], Step [100/100], Loss: 1.8674
End of Epoch 17, Average Loss: 1.7065
Epoch [18/20], Step [100/100], Loss: 1.7427
End of Epoch 18, Average Loss: 1.6888
Epoch [19/20], Step [100/100], Loss: 1.8176
End of Epoch 19, Average Loss: 1.6396
Epoch [20/20], Step [100/100], Loss: 1.5950
End of Epoch 20, Average Loss: 1.6102
```

[29]:
```
def evaluate_model(model, data_loader):
    model.eval()
    total = 0
    correct = 0
    with torch.no_grad():
        for images, labels in data_loader:
            images = images.to(device)
```

```
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return 100 * correct / total
```

[30]:
```
validation_accuracy = evaluate_model(CNN_model, eval_data_loader)
print(f'Validation Accuracy after epoch {epoch+1}: {validation_accuracy:.2f}%')
```

```
Validation Accuracy after epoch 20: 55.98%
```

[31]:
```
torch.save(CNN_model, 'CIFAR100_CNN.pth')
```

4.0.1 Final CNN Test Accuracy: 55.98%

5 Final Conclusions

We trained 4 models - Logistic Regression: 15.23% - Linear SVM: 19.36% - Kernel SVM: 29.06% - CNN: 55.98%

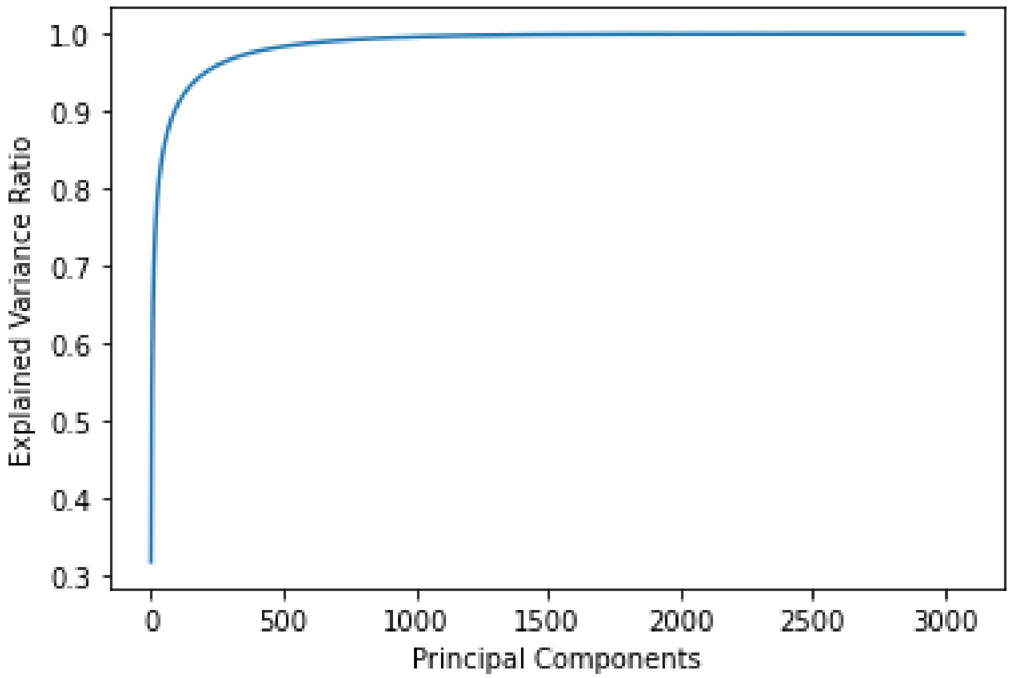
Each of these models performs significantly better than a random clasffifier (1% accuracy) on the CIFAR100 dataset. The accuracies also reflect what might be expected of these models. Logistic regression and linear SVM perform similarly but not very well for a complex task like image classification. As would be expected, introducing a non-linear kernel in SVM improved the accuracy as the model was able to better fit what is likely a non-linear decision boundary. For a task like image classification though a CNN would be the best model, and our projects reaches the same conclusion. A CNN offers the complexity necessary to capture the non-linear relationships while being significantly less demanding to train than a fully connected DNN.

6 SVM Appendix

Steps for potential further improvement of SVM accuracy

[13]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
pca = PCA(random_state=41)
scaler = StandardScaler()
X_train_reduced = pca.fit_transform(scaler.fit_transform(X_train))
exp_variance = pca.explained_variance_ratio_
```

[14]:
```
sns.lineplot(x=[x for x in range(0, 3072)], y=np.cumsum(exp_variance))
plt.xlabel('Principal Components')
plt.ylabel('Explained Variance Ratio')
plt.show()
```



[16]:
```
np.cumsum(exp_variance)[250]
```

[16]: 0.9593125

[17]:
```
X_train_reduced = X_train_reduced[:,:250]
X_val_reduced = pca.transform(X_val)
X_val_reduced = X_val_reduced[:,:250]
fit_svm(X_train_reduced, X_val_reduced, y_train, y_val, C=0.01, kernel='linear')
```

```
C: 0.01, tol: 0.001, max_iter: -1, ovr
Train acccuracy:  0.453225
Validation accuracy:  0.1345
```

Shown above is a the validation accuracy on the principle components using the best linear model established on the full data. It is similar 0.1345 vs 0.1969. This shows that comparable performance can be achieved on data compressed to 8.14% the original size (95.9% variance preserved). Below code where multiple different models are addressed via cross validation on the decomposed data. If the computational resources allowed, this code would produce the best model based on the data. It performs 5-fold cross validation on linear, radial, 3rd degree polynomial, and 4th degree polynomial kernel models, testing 0.001, 0.01, 0.1, 1, 10, and 100 as values of C. This results in 24 different models, each fit 5 times for a total of 120 fits.

[17]:
```
pca = PCA(n_components=250, random_state=41)
scaler = StandardScaler()
X_reduced = pca.fit_transform(scaler.fit_transform(X))
```

```
X_test_reduced = pca.transform(X_test)
```

[ ]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report

svm = SVC()
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'kernel': ['linear', 'rbf', 'poly'],
    'degree': [3, 4]
}

grid_search = GridSearchCV(estimator=svm, param_grid=param_grid, cv=5,␣
 ↪verbose=2, scoring='accuracy')
grid_search.fit(X_reduced, y)

best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_reduced)
y_pred_test = best_model.predict(X_test_reduced)
```

[ ]:
```
grid_results_df = pd.DataFrame(grid_search.cv_results_)
grid_restuls_df.to_csv('grid_results.csv')
grid_results_df
```

[ ]:
```
train_accuracy = accuracy_score(y, y_pred)
test_accuracy = accuracy_score(y_test, y_pred_test)
print('Train acccuracy: ', train_accuracy)
print('Test accuracy: ', val_accuracy)
```