ECE194N HW 2

CNN Report

February 22, 2019

Student:	Erik Rosten
Perm Number:	7143571
Email:	erosten@ucsb.edu

Department of Electrical and Computer Engineering, UCSB

Part a: Visualizing the Dataset

The code for this section is in view_examples.py and repeated below

```
import cifar_loader
import matplotlib.pyplot as plt
import numpy as np


def view_examples(train_imgs, train_cls, classes, class_names):
   print(class_names)
   plt.figure(1)
   for i in range(classes.shape[0]):
      label_class_name = class_names[i]
      rand_integer = np.random.randint(0,4999)
      # grab a random example index
      index = np.where(train_cls == classes[i])[0][rand_integer]
      img_example = train_imgs[index]

      ax = plt.subplot(2,classes.shape[0] / 2,i + 1)
      ax.axis('off')
      imgplot = plt.imshow(img_example)
      ax.set_title(label_class_name)
   plt.show()


def main():
   # load cifar data
   train_imgs, train_cls, train_names = cifar_loader.load_training_data()
   class_names = np.array(cifar_loader.load_class_names())
   classes = np.arange(0,10)
   # view examples
   view_examples(train_imgs, train_cls, classes, class_names)




if __name__ == '__main__':
    main()
```

Running the main function of this code gives the figure below

Figure 1: One example from each class in the cifar-10 dataset

Designing the CNN and Results

The code for this section is in train_cnn.py and repeated below

```
from __future__ import print_function
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten, Dropout
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import LeakyReLU
from keras.layers.normalization import BatchNormalization
import os
import numpy as np

# define some variables
batch_size = 32
num_classes = 10
epochs = 25
data_augmentation = True
```

```
num_predictions = 20
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'keras_cifar10_modified_vgg_dropout.h5'

# load cifar data
(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')

# Convert class vectors to one hot
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# create modified VGG
model = Sequential()
#block 1
model.add(Conv2D(64, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(LeakyReLU(alpha = 0.1))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(LeakyReLU(alpha = 0.1))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

#block 2
model.add(Conv2D(128, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(LeakyReLU(alpha = 0.2))
model.add(Conv2D(128, (3, 3), padding='same'))
model.add(LeakyReLU(alpha = 0.2))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))


#block 3
model.add(Conv2D(256, (3, 3), padding='same'))
model.add(LeakyReLU(alpha = 0.2))
model.add(BatchNormalization())
model.add(Conv2D(256, (3, 3), padding='same'))
model.add(LeakyReLU(alpha = 0.2))
model.add(BatchNormalization())
model.add(Conv2D(256, (3, 3), padding='same'))
model.add(LeakyReLU(alpha = 0.2))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))


model.add(Flatten())
model.add(Dense(4096))
model.add(LeakyReLU(alpha = 0.2))
model.add(Dropout(0.5))
model.add(Dense(4096))
```

```
model.add(LeakyReLU(alpha = 0.2))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(LeakyReLU(alpha = 0.2))
model.add(Activation('softmax'))

# initiate Adam optimizer
opt = keras.optimizers.Adam(epsilon = 1e-03)

model.compile(loss='categorical_crossentropy',
             optimizer=opt,
             metrics=['accuracy'])

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255


# This will do preprocessing and realtime data augmentation:
datagen = ImageDataGenerator(
    featurewise_center=False, # set input mean to 0 over the dataset
    samplewise_center=False, # set each sample mean to 0
    featurewise_std_normalization=False, # divide inputs by std of the dataset
    samplewise_std_normalization=False, # divide each input by its std
    zca_whitening=False, # apply ZCA whitening
    zca_epsilon=1e-06, # epsilon for ZCA whitening
    rotation_range=0, # randomly rotate images in the range (degrees, 0 to 180)
    # randomly shift images horizontally (fraction of total width)
    width_shift_range=0.1,
    # randomly shift images vertically (fraction of total height)
    height_shift_range=0.1,
    shear_range=0., # set range for random shear
    zoom_range=0., # set range for random zoom
    channel_shift_range=0., # set range for random channel shifts
    # set mode for filling points outside the input boundaries
    fill_mode='nearest',
    cval=0., # value used for fill_mode = "constant"
    horizontal_flip=True, # randomly flip images
    vertical_flip=False, # randomly flip images
    # set rescaling factor (applied before any other transformation)
    rescale=None,
    # set function that will be applied on each input
    preprocessing_function=None,
    # image data format, either "channels_first" or "channels_last"
    data_format=None,
    # fraction of images reserved for validation (strictly between 0 and 1)
    validation_split=0.0)

# Compute quantities required for feature-wise normalization
# (std, mean, and principal components if ZCA whitening is applied).
datagen.fit(x_train)

# Fit the model on the batches generated by datagen.flow().
```

```
history = model.fit_generator(datagen.flow(x_train, y_train,
                                 batch_size=batch_size),
              steps_per_epoch = x_train.shape[0] / batch_size,
                   epochs=epochs,
                   validation_data=(x_test, y_test),
                   workers=4, verbose = 1)
train_loss = history.history['loss']
test_loss  = history.history['val_loss']
train_acc = history.history['acc']
test_acc   = history.history['val_acc']
np.savetxt("train_loss.txt", train_loss, delimiter=",")
np.savetxt("train_acc.txt", train_acc, delimiter=",")
np.savetxt("test_loss.txt", test_loss, delimiter=",")
np.savetxt("test_acc.txt", test_acc, delimiter=",")


# Save model and weights
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
model_path = os.path.join(save_dir, model_name)
model.save(model_path)
print('Saved trained model at %s ' % model_path)

# Score trained model.
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

For this section, I chose a modified VGG16. I've taken the VGG16, removed the last 6 conv/pooling layers, and added dropout to the second two fully connected layers. It is therefore 10 layers, consisting of 7 convolutional layers, and 3 fully connected layers. The loss function is cross entropy, which is used to find the loss of two different probability distributions. This assumes that the data has an underlying probability distribution that the CNN is trying to match. The code above saves the train and test losses in text files after 25 epochs of training , which have been included in this zip file. The plots for training and test accuracy and loss is below, and calculated using the code below in plot_results.py, repeated here

```
import numpy as np
import matplotlib.pyplot as plt


test_loss = np.loadtxt('test_loss.txt')
test_acc  = np.loadtxt('test_acc.txt')
train_loss = np.loadtxt('train_loss.txt')
train_acc = np.loadtxt('train_acc.txt')


epochs = np.arange(test_loss.shape[0])

fig = plt.figure(1)
# plot losses
ax = plt.subplot(1,2,1)
plt.plot(epochs, train_loss)
```

```
plt.plot(epochs, test_loss)
ax.legend(['Training', 'Test'])
ax.set_title('Training and Test Loss vs Epoch')
ax.set_xlabel('Epoch')
# plot accuracies
ax = plt.subplot(1,2,2)
plt.plot(epochs, train_acc)
plt.plot(epochs, test_acc)
ax.legend(['Training', 'Test'])
ax.set_title('Training and Test Accuracy vs Epoch')
ax.set_xlabel('Epoch')


fig.tight_layout()
plt.show()
```

which gives the plot below

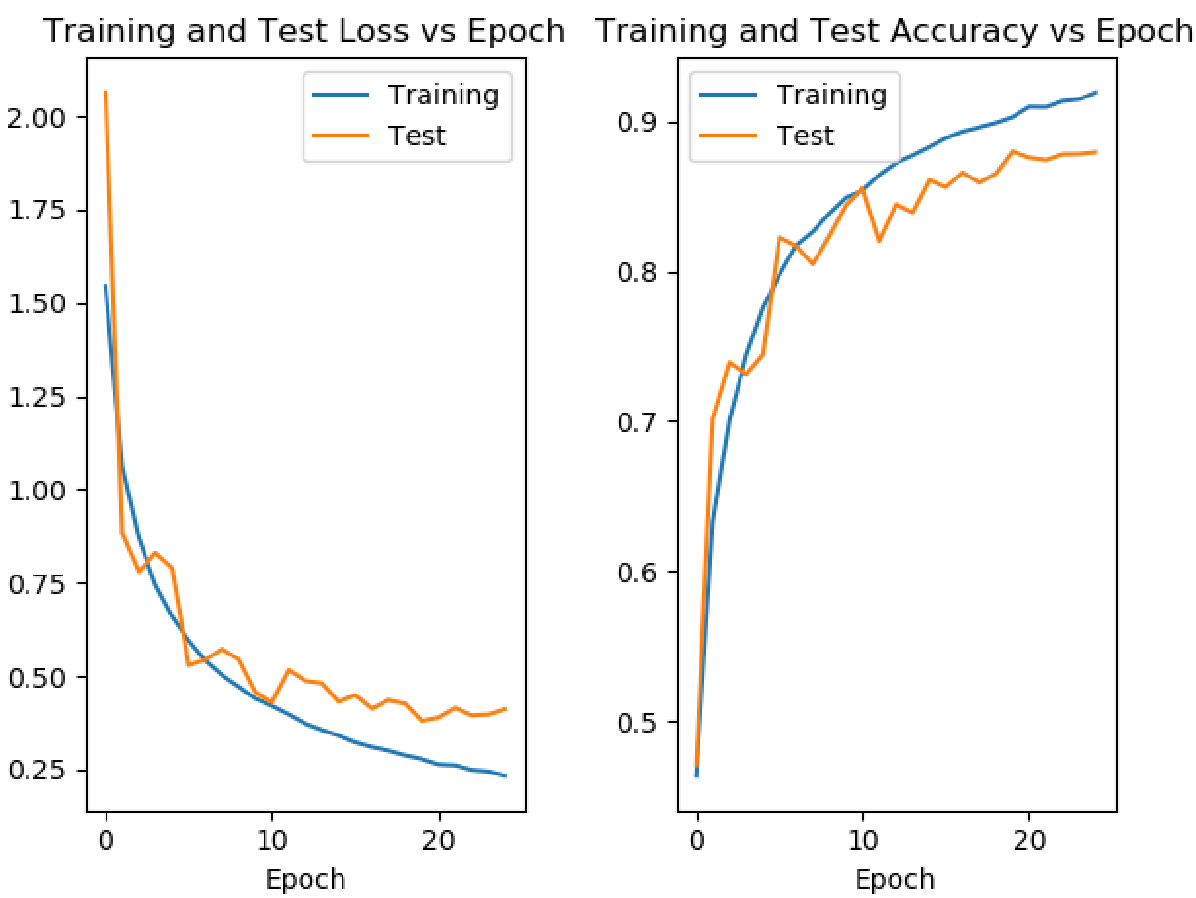

Figure 2: Results of the Modified VGG

Note that there is a good amount of overfitting after around epoch 10, which is what drove my choice to include dropout in the network. This is another reason for using real time data augmentation. The accuracy could be improved by perhaps adding more techniques that combat overfitting.

Part c: Model Parameters

One advantage of using keras is it's model summary function. Using the code below from print_num_parameters.py

```
from __future__ import print_function
import keras
from keras.models import load_model
from keras.datasets import cifar10
import numpy as np
import matplotlib.pyplot as plt


model = load_model('saved_models/keras_cifar10_modified_vgg_dropout.h5')
model.summary()
```

gives

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 64)        1792
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 32, 32, 64)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 64)        256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 64)        36928
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 32, 32, 64)        0
_________________________________________________________________
batch_normalization_2 (Batch (None, 32, 32, 64)        256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 128)       73856
_________________________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 128)       147584
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 16, 16, 128)       0
_________________________________________________________________
batch_normalization_4 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 256)         295168
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 8, 8, 256)         0
_________________________________________________________________
batch_normalization_5 (Batch (None, 8, 8, 256)         1024
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 256)         590080
```

```
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)   (None, 8, 8, 256)      0
_________________________________________________________________
batch_normalization_6 (Batch (None, 8, 8, 256)    1024
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 8, 8, 256)      590080
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)   (None, 8, 8, 256)      0
_________________________________________________________________
batch_normalization_7 (Batch (None, 8, 8, 256)    1024
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 256)    0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4096)           0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)           16781312
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)   (None, 4096)           0
_________________________________________________________________
dropout_1 (Dropout)          (None, 4096)           0
_________________________________________________________________
dense_2 (Dense)              (None, 4096)           16781312
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)   (None, 4096)           0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4096)           0
_________________________________________________________________
dense_3 (Dense)              (None, 10)             40970
_________________________________________________________________
leaky_re_lu_10 (LeakyReLU)  (None, 10)             0
_________________________________________________________________
activation_1 (Activation)   (None, 10)             0
=================================================================
Total params: 35,343,690
Trainable params: 35,341,386
Non-trainable params: 2,304
_________________________________
```

Part d: Viewing Wrongly Classified Images

This section utilized the code from find_incorrect_instances.py.

```python
from __future__ import print_function
import keras
from keras.models import load_model
from keras.datasets import cifar10
import numpy as np
import matplotlib.pyplot as plt


#load model
model = load_model('saved_models/keras_cifar10_modified_vgg_dropout.h5')
print('Model Loaded')
```

```
# load cifar data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('Dataset Loaded')


# preprocess
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255


# find incorrectly classified images
print('Finding Incorrectly Classified Images')
predictions = model.predict_classes(x_test).reshape((-1,1))
incorrects = np.nonzero(predictions!= y_test)[0]
np.random.shuffle(incorrects)
_, incorrect_indices = np.unique(y_test[incorrects], return_index = True)
incorrect_indices = incorrects[incorrect_indices]
for i in range(incorrect_indices.shape[0]):
   index = incorrect_indices[i]
   imgplt = plt.imshow(x_test[index])
   plt.title('Predicted Class: {} Actual Class: {}'.format(predictions[index], y_test[index]))
   plt.show()
```

A reminder of the classes in CIFAR-10 from 0 to 10 is below

$$[airplane,\ automobile,\ bird,\ cat,\ deer,\ dog,\ frog,\ horse,\ ship,\ truck]$$

Running the code above gives the incorrect images below

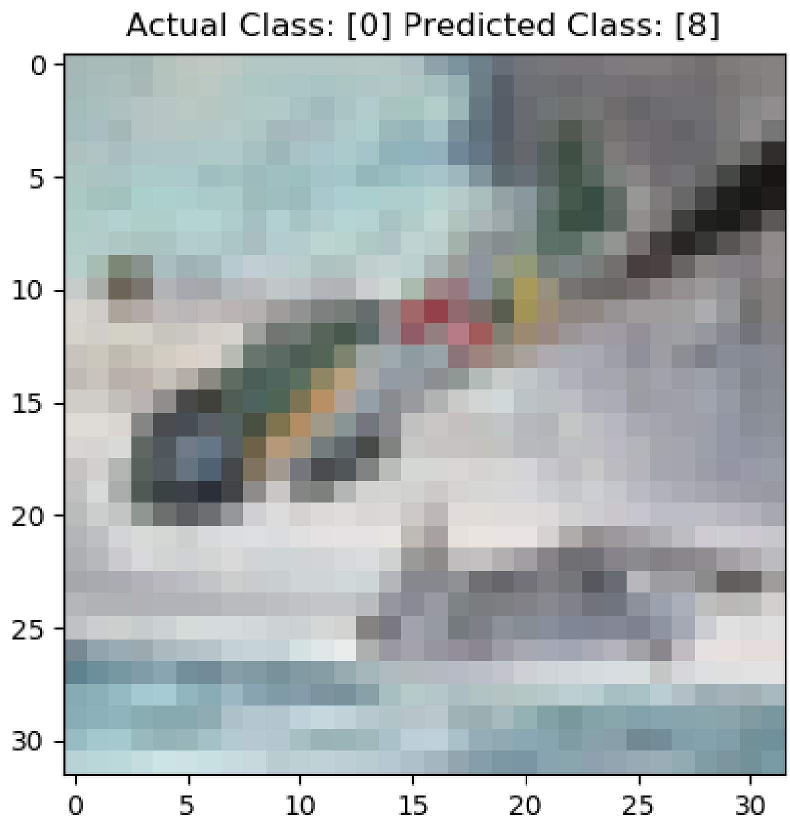

Figure 3: Real: airplane, Predicted: ship

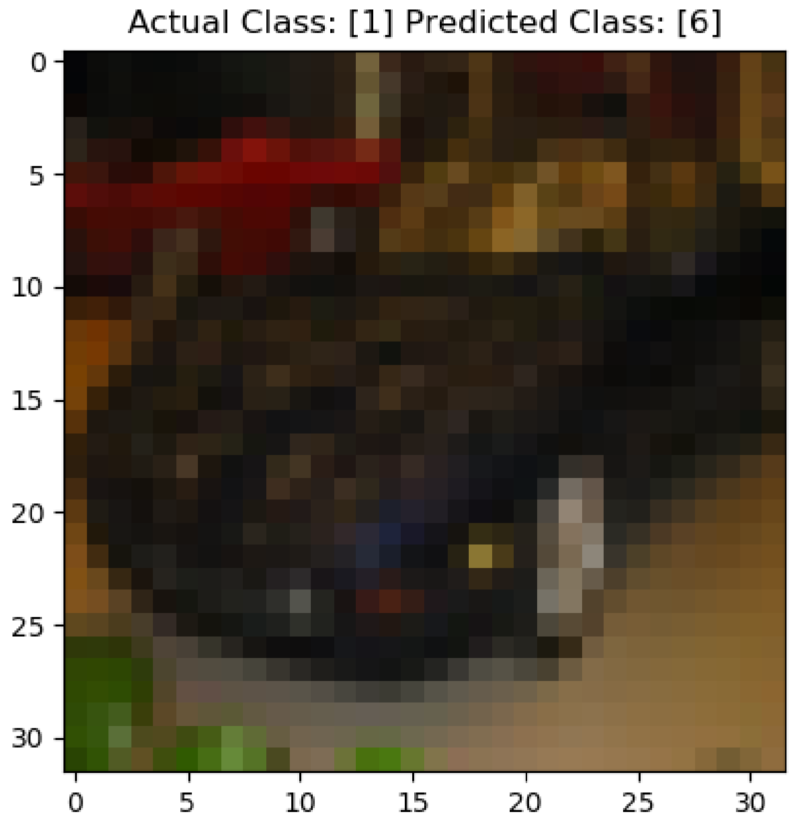

Figure 4: Real: automobile, Predicted: frog

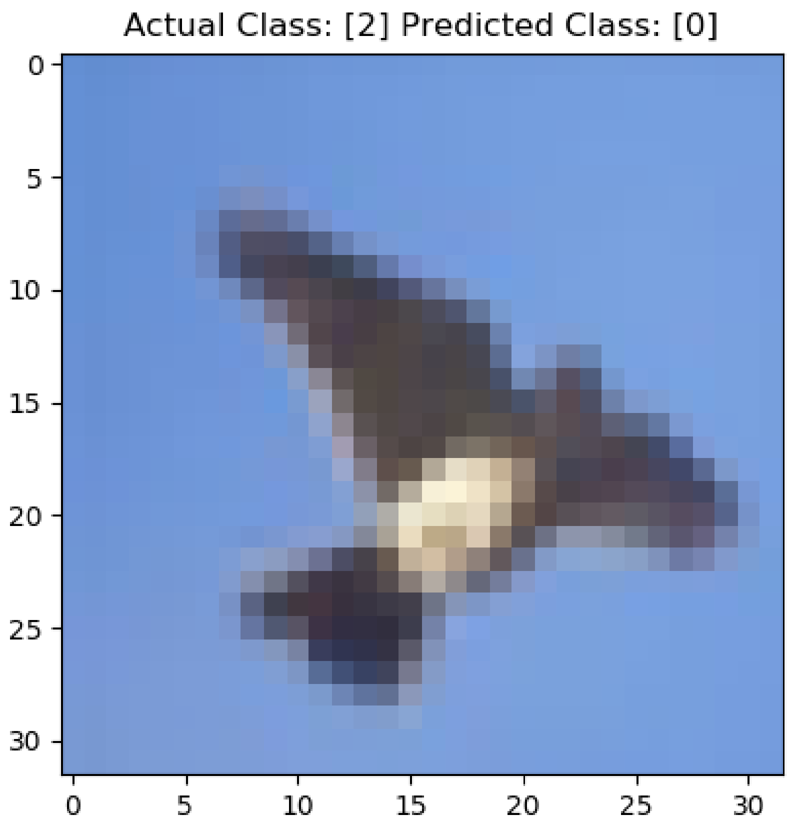

Figure 5: Real: bird, Predicted: airplane

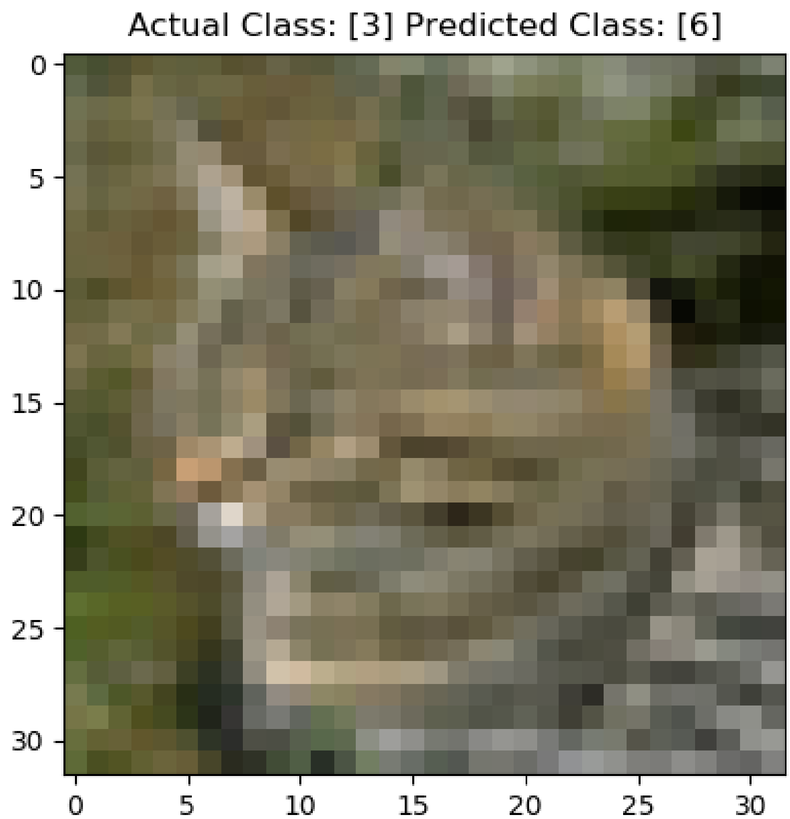

Figure 6: Real: cat, Predicted: frog

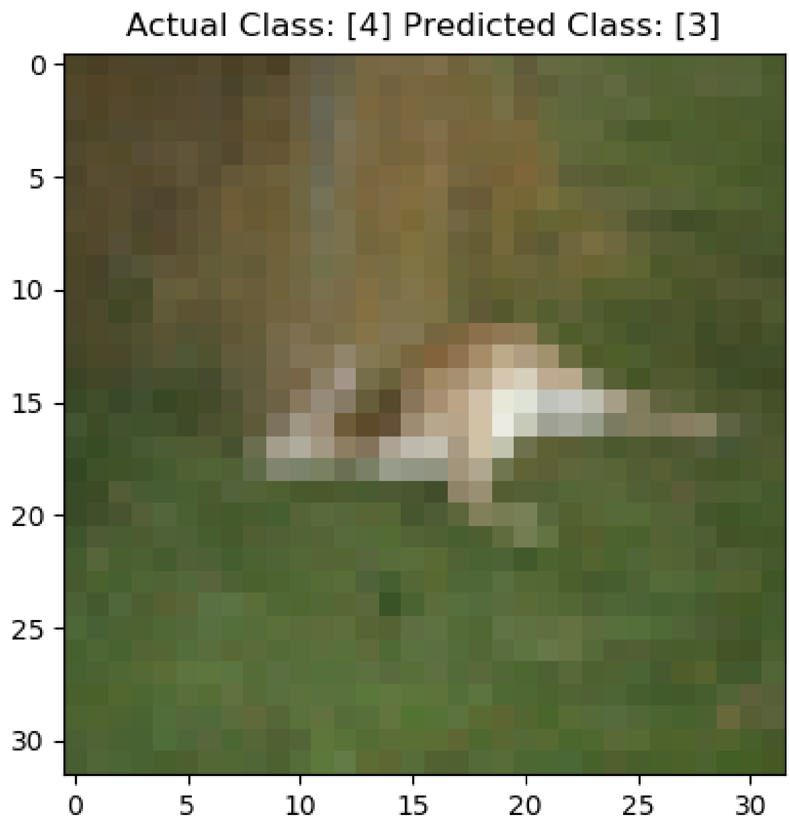

Figure 7: Real: deer, Predicted: cat

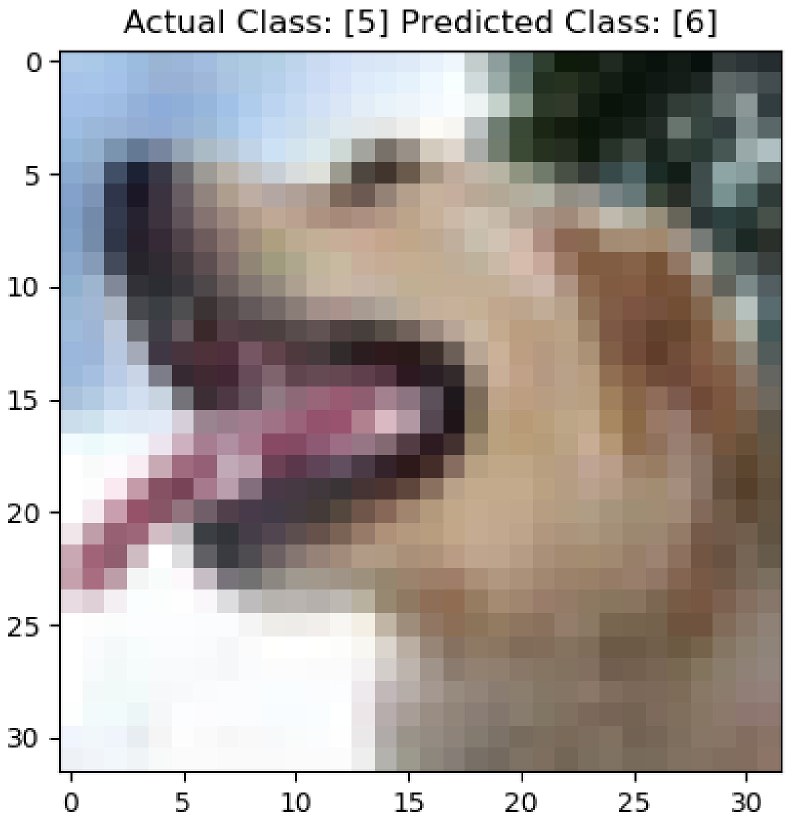

Figure 8: Real: dog, Predicted: frog

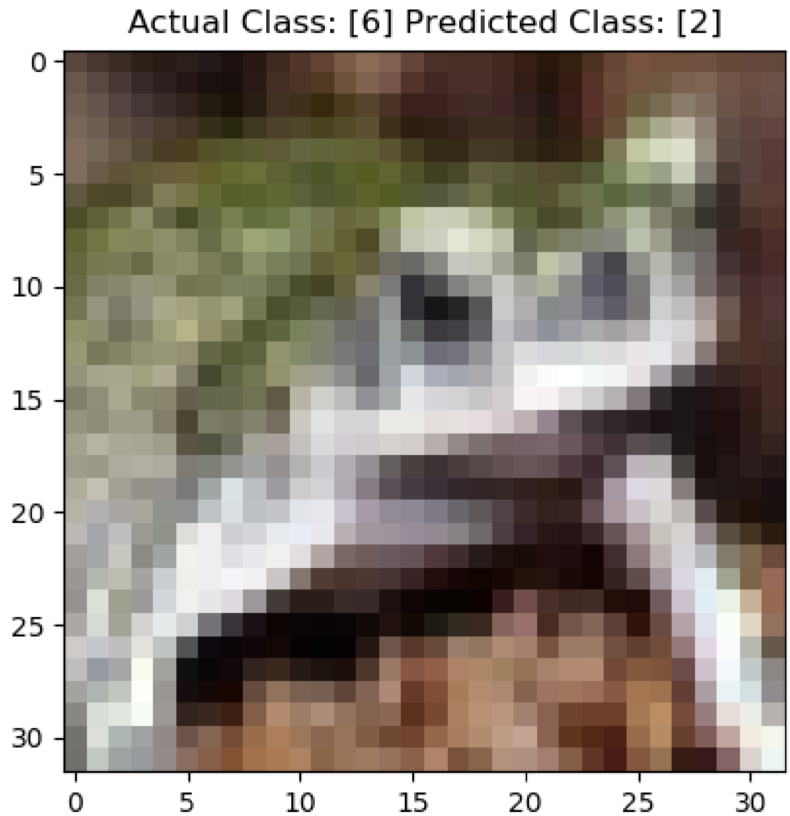

Figure 9: Real: frog, Predicted: automobile

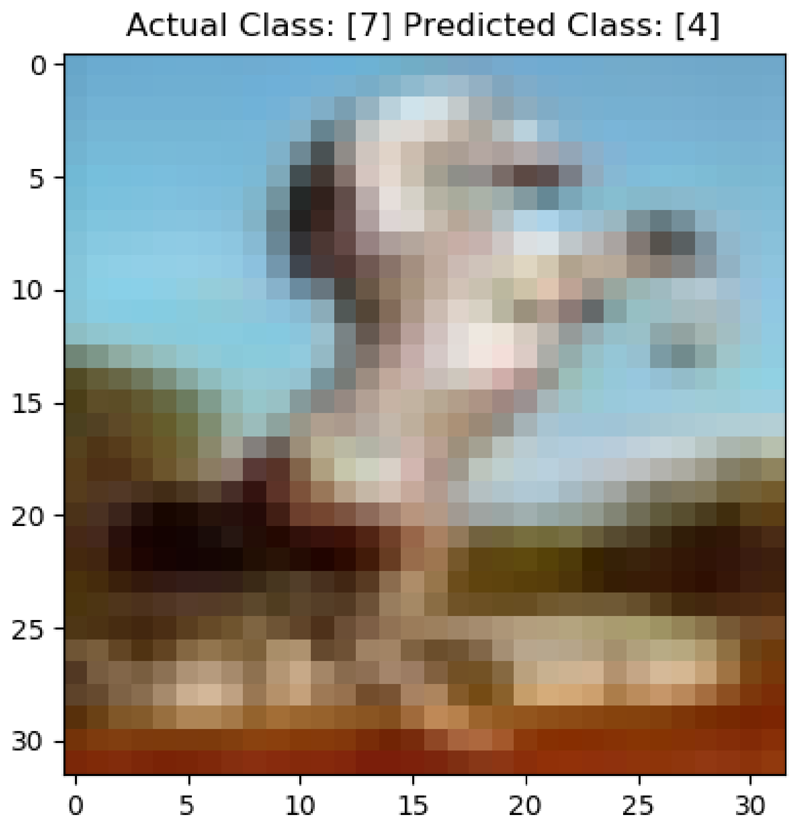

Figure 10: Real: horse, Predicted: deer

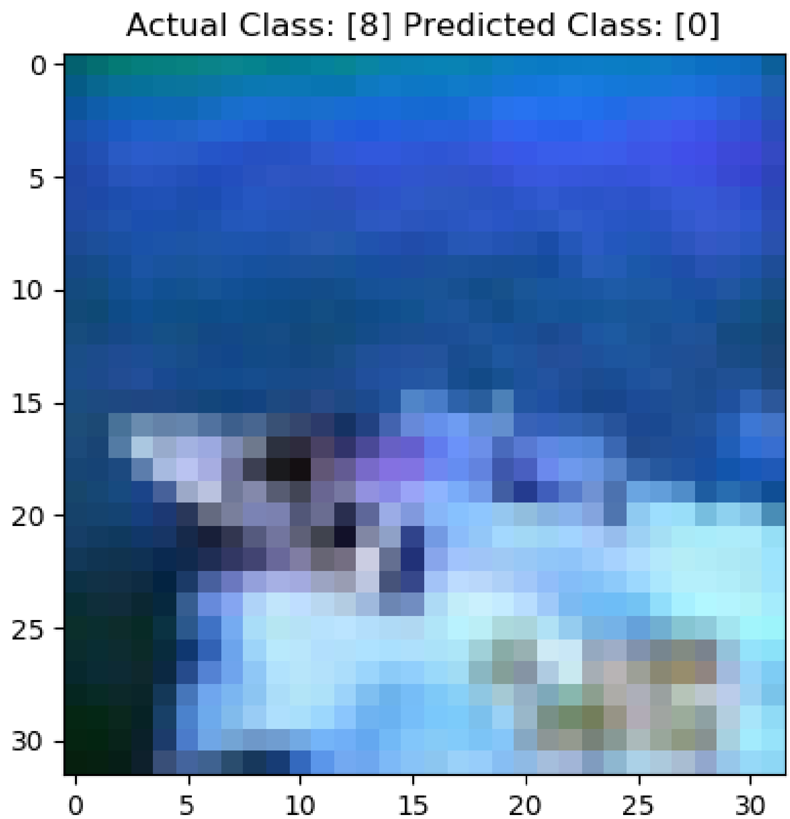

Figure 11: Real: ship, Predicted: airplane

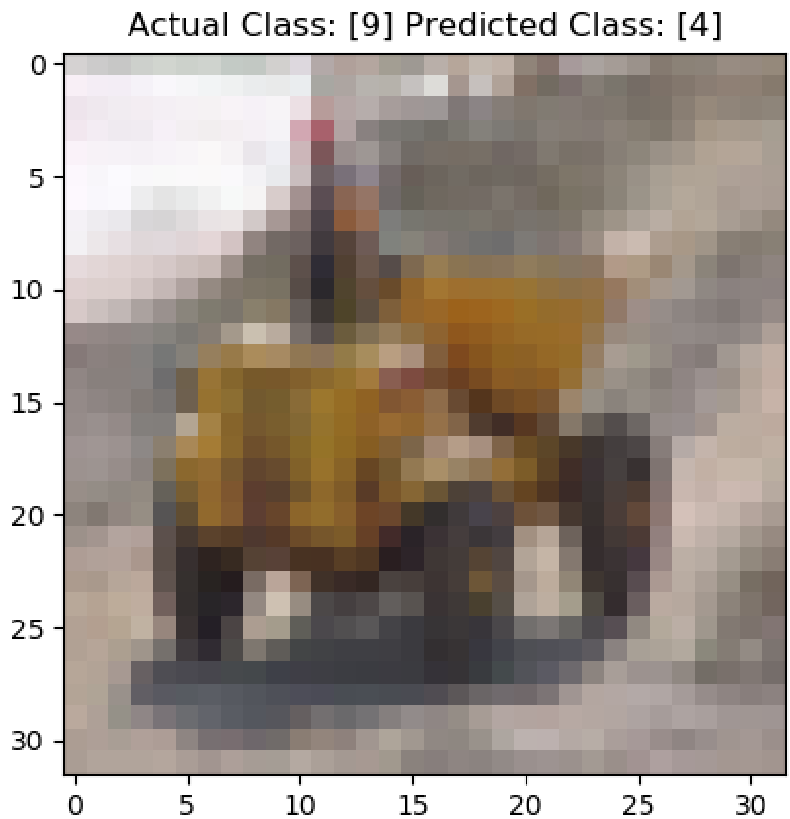

Figure 12: Real: truck, Predicted: deer