

Assignment 2: Convolutional Neural Networks
Lucas Butler

1. Build CNN

For this assignment, the objective is to train a convolutional neural network (CNN) on the CIFAR-10 dataset. For my model architecture, I drew sizing inspiration from [heysachin's github repository](#) [1], who tackled the same problem and achieved an accuracy of 70%.

The model utilizes three convolutional layers of sizes 16, 32, and 64. All of them utilize a kernel size of 3x3, a stride of 1, and 1 pixel of padding on the outside of the image. These layers feed into the rectified linear unit activation function (ReLU), which was chosen for its computational efficiency and historic performance.

After each convolution and activation, a max pooling operation of size 2x2 reduces the resolution of the image by a factor of 2. The output of the last convolutional layer is flattened, creating a layer with $64 * 4 * 4 = 1024$ nodes. 25% of these 1024 nodes are randomly set to 0 through the dropout operation before being passed into a 500 node fully connected (FC) layer.

The 500 node FC layer passes through a ReLU function, then another 0.25% dropout operation before being connected to 10 output nodes for classification. Finally, the categorical cross entropy function is then used to determine the loss, before the optimizer kicks in.

Model Architecture
Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)
relu()
MaxPool2d(2, 2)
Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
relu()
MaxPool2d(2, 2)
Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
relu()
MaxPool2d(2, 2)
FLATTEN
Dropout(0.25)
Linear(64 * 4 * 4, 500)
relu()

Dropout(0.25)
Linear(500, 10)

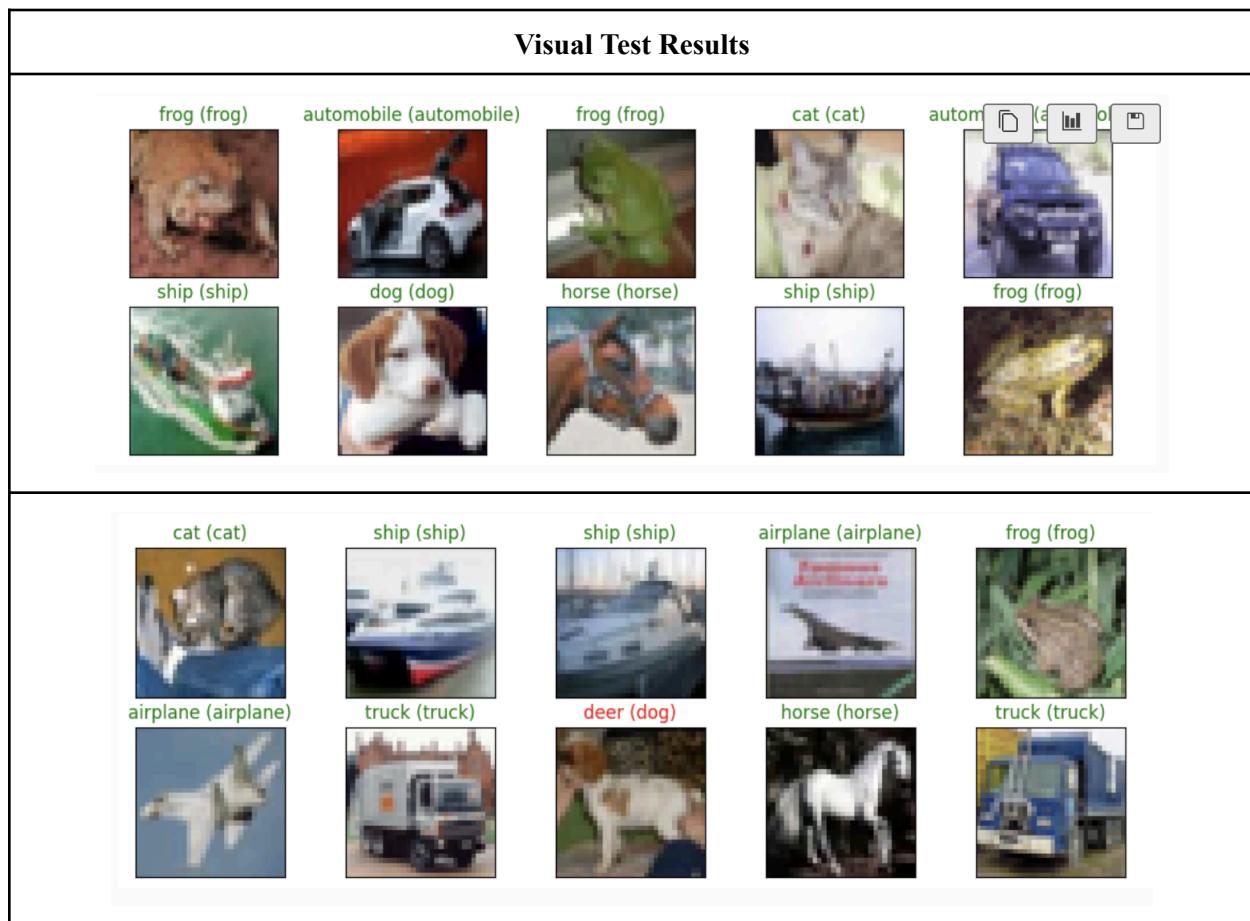
My hyperparameter selection slightly varies from the original setup in the assigned file. In my case, I increased the epochs from 5 to 100 and added an early stopping mechanism that stops the training after 10 epochs of no validation loss improvement. The learning rate was left at 0.01, however, both stochastic gradient descent and Adam optimizers were tested. Batch sizes of 16, 20, and 32 were experimented with, but it was found that 20 worked best for this model.

Hyper Parameter	Value
Max Epochs	100
Early Stopping Patience	10
Batch Size	20
Learning Rate	0.01
Optimizer	SGD, Adam

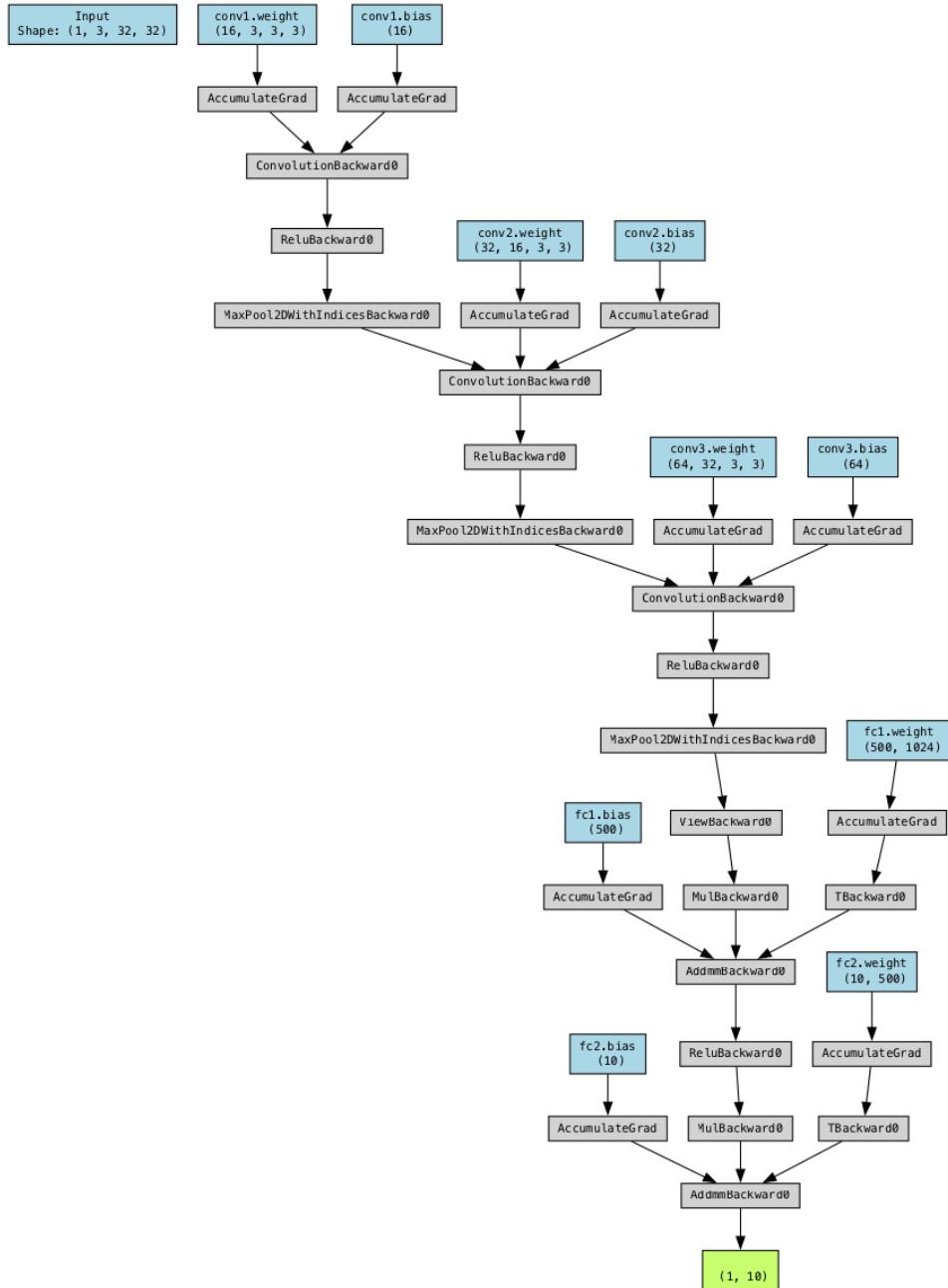
After trying all of the combinations, it was found the SGD optimizer was far better than the Adam optimizer. The best run ran for 34 epochs before early stopping was triggered, and achieved a validation loss of 0.69.

Adam Optimizer Training Run	SGD Training Run (Best Run)
<pre> Epoch: 4 Training Loss: 1.871627 Validation Loss: 1.800180 Epoch: 5 Training Loss: 1.865913 Validation Loss: 1.795681 Epoch: 6 Training Loss: 1.876183 Validation Loss: 1.891529 Epoch: 7 Training Loss: 1.869876 Validation Loss: 1.899572 Epoch: 8 Training Loss: 1.867193 Validation Loss: 2.009897 Epoch: 9 Training Loss: 1.887755 Validation Loss: 1.819636 Epoch: 10 Training Loss: 1.861460 Validation Loss: 1.785274 Validation loss decreased (1.788048 --> 1.785274). Saving model ... Epoch: 11 Training Loss: 1.865620 Validation Loss: 1.814370 Epoch: 12 Training Loss: 1.879742 Validation Loss: 1.830232 Epoch: 13 Training Loss: 1.874567 Validation Loss: 1.785262 Validation loss decreased (1.785274 --> 1.785262). Saving model ... Epoch: 14 Training Loss: 1.875062 Validation Loss: 1.780718 Validation loss decreased (1.785262 --> 1.780718). Saving model ... Epoch: 15 Training Loss: 1.873958 Validation Loss: 1.764497 Validation loss decreased (1.780718 --> 1.764497). Saving model ... Epoch: 16 Training Loss: 1.884847 Validation Loss: 1.812043 Epoch: 17 Training Loss: 1.893591 Validation Loss: 1.811983 Epoch: 18 Training Loss: 2.092671 Validation Loss: 2.038455 Epoch: 19 Training Loss: 2.046187 Validation Loss: 2.028712 Epoch: 20 Training Loss: 2.066684 Validation Loss: 2.018159 Epoch: 21 Training Loss: 2.036461 Validation Loss: 1.984657 Epoch: 22 Training Loss: 2.024290 Validation Loss: 1.958010 Epoch: 23 Training Loss: 2.057869 Validation Loss: 2.086961 Epoch: 24 Training Loss: 2.035258 Validation Loss: 1.958435 Epoch: 25 Training Loss: 2.015310 Validation Loss: 1.989358 </pre>	<pre> Validation loss decreased (0.799887 --> 0.781168). Saving model ... Epoch: 14 Training Loss: 0.751940 Validation Loss: 0.769140 Validation loss decreased (0.781168 --> 0.769140). Saving model ... Epoch: 15 Training Loss: 0.714594 Validation Loss: 0.751552 Validation loss decreased (0.769140 --> 0.751552). Saving model ... Epoch: 16 Training Loss: 0.686935 Validation Loss: 0.744286 Validation loss decreased (0.751552 --> 0.744286). Saving model ... Epoch: 17 Training Loss: 0.656549 Validation Loss: 0.742664 Validation loss decreased (0.744286 --> 0.742664). Saving model ... Epoch: 18 Training Loss: 0.633287 Validation Loss: 0.714291 Validation loss decreased (0.742664 --> 0.714291). Saving model ... Epoch: 19 Training Loss: 0.606360 Validation Loss: 0.710618 Validation loss decreased (0.714291 --> 0.710618). Saving model ... Epoch: 20 Training Loss: 0.581759 Validation Loss: 0.694608 Validation loss decreased (0.710618 --> 0.694608). Saving model ... Epoch: 21 Training Loss: 0.560011 Validation Loss: 0.700198 Epoch: 22 Training Loss: 0.530882 Validation Loss: 0.699327 Epoch: 23 Training Loss: 0.510691 Validation Loss: 0.728201 Epoch: 24 Training Loss: 0.493436 Validation Loss: 0.691854 Validation loss decreased (0.694608 --> 0.691854). Saving model ... Epoch: 25 Training Loss: 0.464579 Validation Loss: 0.709685 Epoch: 26 Training Loss: 0.455230 Validation Loss: 0.704332 Epoch: 27 Training Loss: 0.428798 Validation Loss: 0.701736 Epoch: 28 Training Loss: 0.414719 Validation Loss: 0.696596 Epoch: 29 Training Loss: 0.396459 Validation Loss: 0.714299 Epoch: 30 Training Loss: 0.383098 Validation Loss: 0.720496 Epoch: 31 Training Loss: 0.364282 Validation Loss: 0.702077 Epoch: 32 Training Loss: 0.350195 Validation Loss: 0.730433 Epoch: 33 Training Loss: 0.329403 Validation Loss: 0.727997 Epoch: 34 Training Loss: 0.320108 Validation Loss: 0.747220 </pre>

Adam Optimizer Test	SGD Optimizer Test (Best Results)
<p>Test Loss: 1.745755</p> <p>Test Accuracy of airplane: 20% (205/1000) Test Accuracy of automobile: 50% (504/1000) Test Accuracy of bird: 10% (101/1000) Test Accuracy of cat: 7% (79/1000) Test Accuracy of deer: 24% (241/1000) Test Accuracy of dog: 38% (384/1000) Test Accuracy of frog: 57% (579/1000) Test Accuracy of horse: 31% (319/1000) Test Accuracy of ship: 57% (571/1000) Test Accuracy of truck: 46% (469/1000)</p> <p>Test Accuracy (Overall): 34% (3452/10000)</p>	<p>Test Loss: 0.706488</p> <p>Test Accuracy of airplane: 79% (792/1000) Test Accuracy of automobile: 89% (896/1000) Test Accuracy of bird: 65% (657/1000) Test Accuracy of cat: 52% (527/1000) Test Accuracy of deer: 66% (666/1000) Test Accuracy of dog: 70% (706/1000) Test Accuracy of frog: 83% (839/1000) Test Accuracy of horse: 82% (821/1000) Test Accuracy of ship: 85% (857/1000) Test Accuracy of truck: 79% (799/1000)</p> <p>Test Accuracy (Overall): 75% (7560/10000)</p>



Model Architecture Compute Graph



Model Architecture Code

```
# define the CNN architecture
class Net(nn.Module):
    def __init__(self):
```

```

super(Net, self).__init__()

# TOTO: Build multiple convolutional layers (sees 32x32x3 image tensor in the
first hidden layer)
self.image_width = 32
self.image_height = 32

# for example, conv1, conv2 and conv3
self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3,
stride=1, padding=1)
self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,
stride=1, padding=1)
self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
stride=1, padding=1)

# max pooling layer
self.pool = nn.MaxPool2d(2, 2)

# TODO: Build some linear layers (fully connected)
# for example, fc1 and fc2
self.fc1 = nn.Linear(64 * 4 * 4, 500)
self.fc2 = nn.Linear(500, 10)

# TODO: dropout layer (p=0.25, you can adjust)
# example self.dropout = nn.Dropout(0.25)
self.dropout = nn.Dropout(0.25)

def forward(self, x):
    # add sequence of convolutional and max pooling layers
    # assume we have 2 convolutional layers defined above
    # and we do a maxpooling after each conv layer
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = self.pool(F.relu(self.conv3(x)))

    # TODO: flatten x at this point to get it ready to feed into the fully
    # connected layer(s)
    # Can use this but need to figure out the actual value for a, b and c
    x = x.view(-1, 4 * 4 * 64)

    # optional add dropout layer

```

```

x = self.dropout(x)

# add 1st hidden layer, with relu activation function
x = F.relu(self.fc1(x))

# optional add dropout layer
x = self.dropout(x)

# add 2nd hidden layer, with relu activation function
x = self.fc2(x)

return x

```

Modified Training Code

```

import torch.optim as optim

# create a complete CNN
model = Net()
print(model)

# move tensors to GPU if CUDA is available
if train_on_gpu:
    model.cuda()

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01)

# TODO, compare with optimizer ADAM
optimizer = optim.Adam(model.parameters(), lr=0.01)

# number of epochs to train the model, you decide the number
n_epochs = 60
patience = 10
patience_count = 0

```

```

valid_loss_min = np.inf # track change in validation loss

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    #####
    # train the model #
    #####
    model.train()

    for batch_idx, (data, target) in enumerate(train_loader):
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()

        # clear the gradients of all optimized variables
        optimizer.zero_grad()

        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)

        # calculate the batch loss
        loss = criterion(output, target)

        # backward pass: compute gradient of the loss with respect to model
        # parameters
        loss.backward()

        # perform a single optimization step (parameter update)
        optimizer.step()

        # update training loss
        train_loss += loss.item()*data.size(0)

    #####
    # validate the model #
    #####
    model.eval()

    for batch_idx, (data, target) in enumerate(valid_loader):
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()

        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)

        # calculate the batch loss

```

```
loss = criterion(output, target)
# update average validation loss
valid_loss += loss.item()*data.size(0)

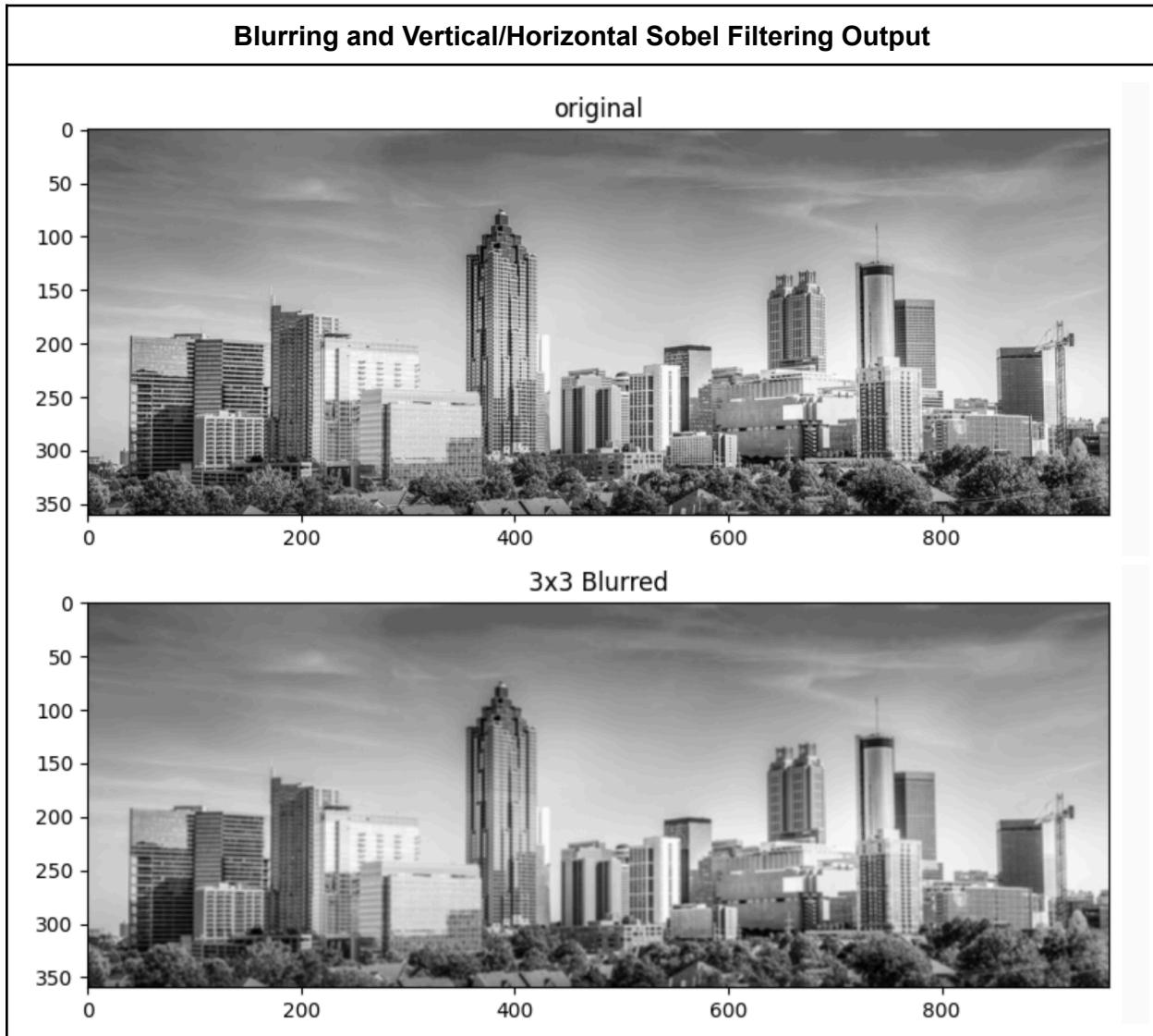
# calculate average losses
train_loss = train_loss/len(train_loader.sampler)
valid_loss = valid_loss/len(valid_loader.sampler)

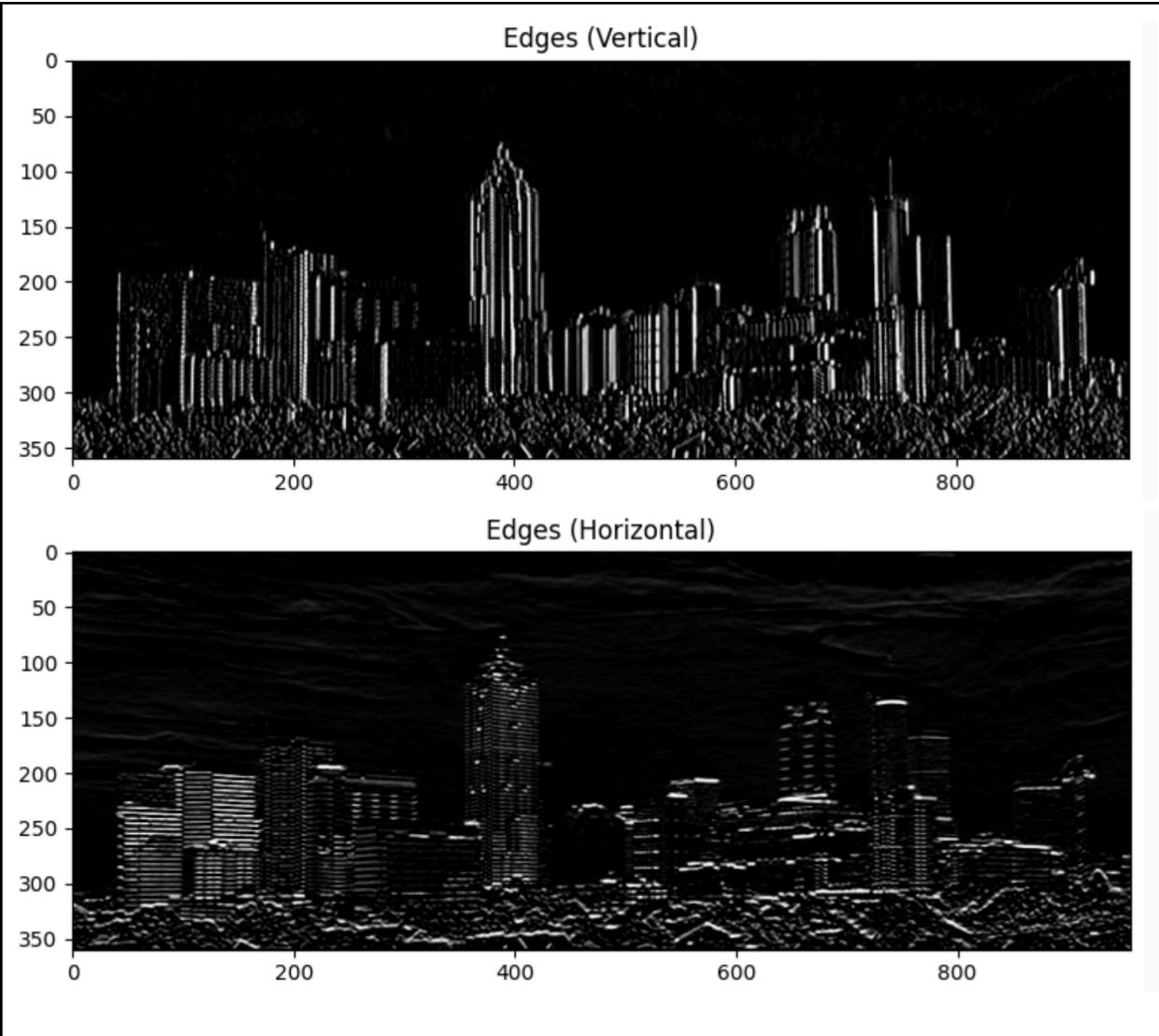
# print training/validation statistics
print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
    epoch, train_loss, valid_loss))

# save model if validation loss has decreased
if valid_loss <= valid_loss_min:
    print('Validation loss decreased ({:.6f} --> {:.6f}). Saving model
...'.format(
        valid_loss_min,
        valid_loss))
    torch.save(model.state_dict(), 'model_trained.pt')
    valid_loss_min = valid_loss
    patience_count = 0
else:
    patience_count += 1
    if patience_count >= patience:
        break
```

2. Image Filtering

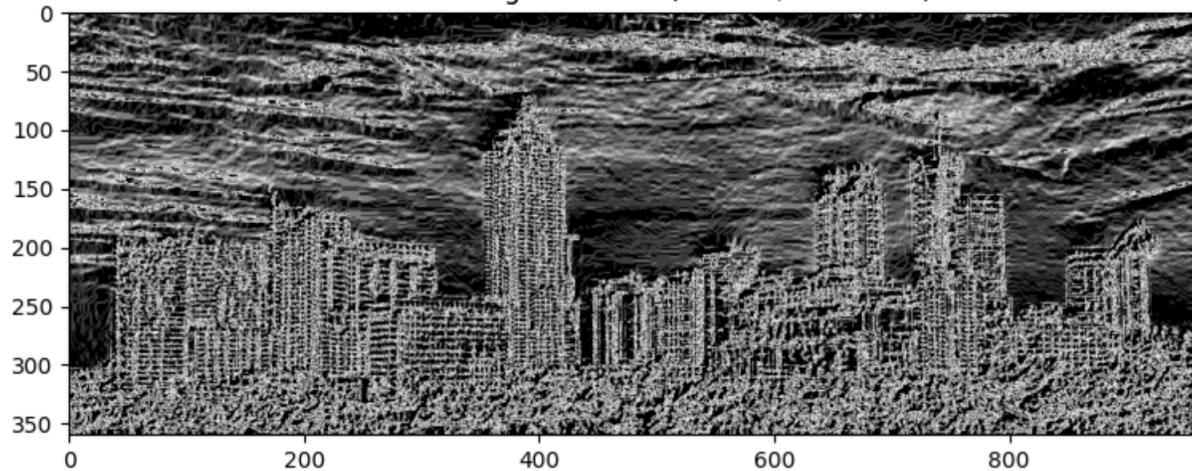
For this assignment, we are to combine blurring with the sobel operator to perform basic edge detection on the provided images. The blurring operation used is a 3x3 matrix of 1s. Larger matrices of 5x5 and 7x7 were experimented with, but washed out too many details from the image, so were discarded. The sobel operation was implemented using the OpenCV filter2d function along with the standard vertical and horizontal matrices. Finally, the magnitude of both horizontal and vertical images is taken at each picture to create the complete edge gradient of the image. This operation was conducted using numpy.square and numpy.sqrt functions on the vertical and horizontal sobel filtered images.





Resulting Edge Gradient from Combining Vertical and Horizontal Filters

Combined Edge Gradient (Vertical/Horizontal)



Blurring and Vertical/Horizontal Sobel Filtering Code

```
plt.clf()

# Define kernels
S3x3 = np.ones(shape=(3,3), dtype=np.float32)

# Sobel kernels for edge detection
Kx = np.array([[[-1, 0, 1],
                [-2, 0, 2],
                [-1, 0, 1]]])

Ky = np.array([[[-1, -2, -1],
                [ 0,  0,  0],
                [ 1,  2,  1]]])

fig = plt.figure(figsize=(48, 20))
fig.add_subplot(5,1,1)
plt.imshow(gray, cmap='gray')
plt.title('original')

# Blur image using a 3x3 average
blurred_image = cv2.filter2D(gray, -1, S3x3/9.0)
fig.add_subplot(5,1,2)
```

```

plt.imshow(blurred_image, cmap='gray')
plt.title('3x3 Blurred')

# Detect edges using Sobel operator (vertical edges)
edges_x = cv2.filter2D(blurred_image, -1, Kx)
fig.add_subplot(5,1,3)
plt.imshow(edges_x, cmap='gray')
plt.title('Edges (Vertical)')

# Detect edges using Sobel operator (horizontal edges)
edges_y = cv2.filter2D(blurred_image, -1, Ky)
fig.add_subplot(5,1,4)
plt.imshow(edges_y, cmap='gray')
plt.title('Edges (Horizontal)')

combined = np.sqrt(np.square(edges_x) + np.square(edges_y))
fig.add_subplot(5,1,5)
plt.imshow(combined, cmap='gray')
plt.title('Combined Edge Gradient (Vertical/Horizontal)')

plt.show()

```

References

- [1] *heysachine*, Convolutional-Neural-Network-CIFAR-10-PyTorch, Github, 2018,
https://github.com/heysachin/Convolutional-Neural-Network-CIFAR-10-PyTorch/blob/master/cifar10_cn.ipynb.