In [ ]:

```python
# core
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import shutil
import random
from time import time

# images
from PIL import Image
from IPython.display import display

# file download
from zipfile import ZipFile
from kaggle.api.kaggle_api_extended import KaggleApi

# torch
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader

# sklearn & xgb
from sklearn.decomposition import PCA
from sklearn.metrics import classification_report
from xgboost import XGBClassifier

# styles
plt.style.use('dark_background')

# authentication
api = KaggleApi()
api.authenticate()

# gpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [ ]:

```python
# Hyperparameters
NUM_EPOCHS = 100
BATCH_SIZE = 32
OPTIMIZER = optim.Adam
LEARNING_RATE = 0.001
DROPOUT = 0.3
```

Load Data

In [ ]:

```python
# clear out data directory
for folder in os.listdir('data'):
    shutil.rmtree(f'data/{folder}')
```

```
# download dataset with Kaggle API
api.dataset_download_files('puneet6060/intel-image-classification')

# designate downloaded file as zip, and unzip
zf = ZipFile('intel-image-classification.zip')
zf.extractall('data')
zf.close()

# delete downloaded zip and extracted csv - keep your directory clean!
os.remove('intel-image-classification.zip')

# clean up folder structure
for subset in ['train', 'test']:

    if not os.path.exists(f'data/{subset}'):
        os.makedirs(f'data/{subset}')

    base_path = f'data/seg_{subset}'
    source_path = f'data/seg_{subset}/seg_{subset}'
    target_path = f'data/{subset}'

    num_files = 0
    for folder in os.listdir(source_path):
        shutil.move(os.path.join(source_path, folder), os.path.join(target_path, fo
        num_files += len(os.listdir(os.path.join(target_path, folder)))
    print(f'Number of {subset} files: {num_files}')

    os.rmdir(source_path)
    os.rmdir(base_path)

num_pred_files = len(os.listdir('data/seg_pred/seg_pred'))
print(f'Number of pred files: {num_pred_files}')
for img in os.listdir('data/seg_pred/seg_pred'):
    os.remove(f'data/seg_pred/seg_pred/{img}')
os.rmdir('data/seg_pred/seg_pred')
os.rmdir('data/seg_pred')
```

```
Number of train files: 14034
Number of test files: 3000
Number of pred files: 7301
```

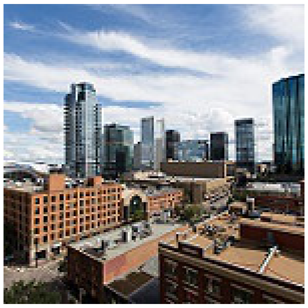
EDA

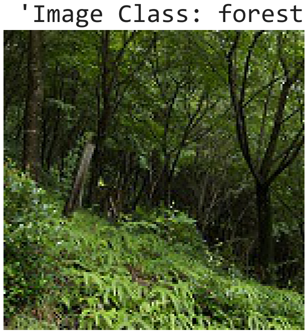
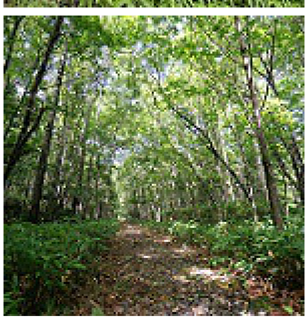
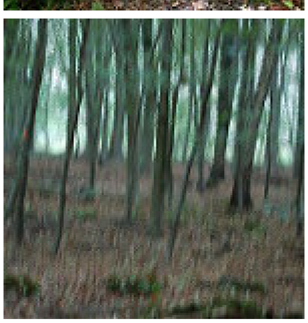
View Images

In [ ]:

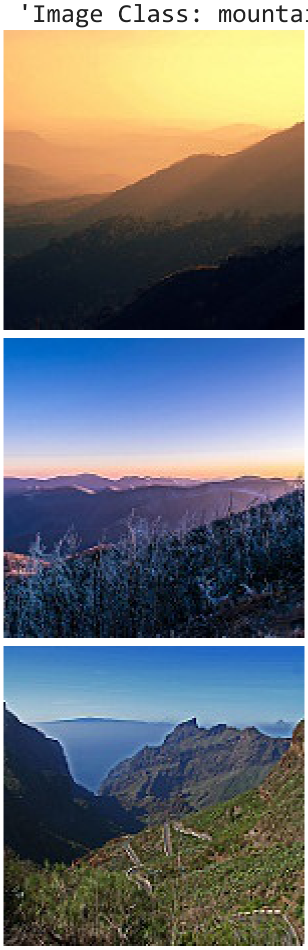
```
for label in os.listdir('data/train'):
    display(f'Image Class: {label}')
    for idx in range(3):
        img_filename = os.listdir(f'data/train/{label}')[idx]
        img = Image.open(f'data/train/{label}/{img_filename}')
        display(img)
```

```
'Image Class: buildings'
```

'Image Class: forest'



'Image Class: glacier'

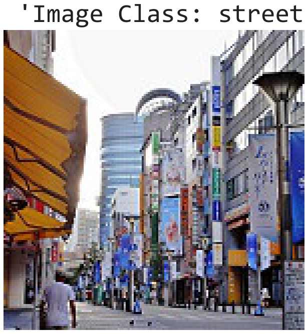
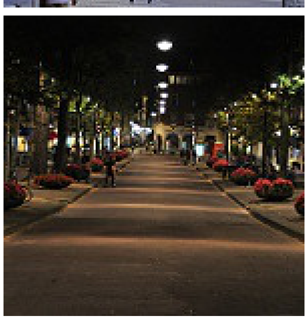
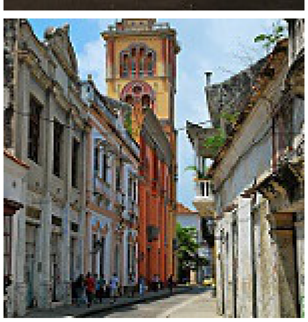
'Image Class: mountain'



'Image Class: sea'

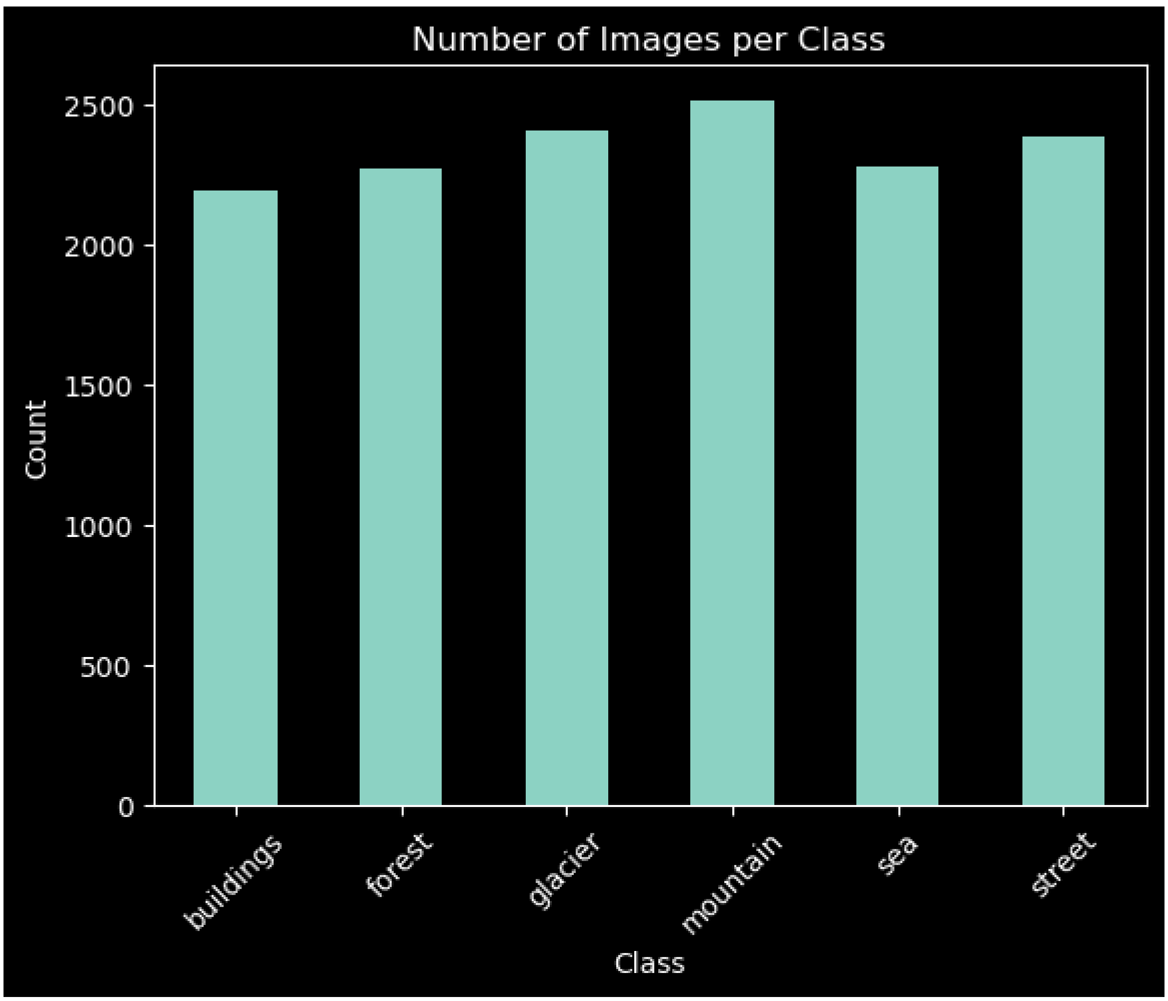


'Image Class: street'

Distribution Across Classes

In [ ]:
```
class_counts = {}
for folder in os.listdir('data/train'):
    class_path = f'data/train/{folder}'
    if os.path.isdir(class_path):
        class_counts[folder] = len(os.listdir(class_path))

pd.DataFrame(class_counts, index=['count']).transpose().plot(kind='bar')
plt.title('Number of Images per Class')
plt.ylabel('Count')
plt.xlabel('Class')
plt.xticks(rotation=45)
plt.legend().remove()
plt.show()
```



Distribution of Image Size and Quality

In [ ]:
```
sizes = []
for folder in os.listdir('data/train'):
    class_path = f'data/train/{folder}'
    for image_name in os.listdir(class_path):
        image_path = os.path.join(class_path, image_name)
        with Image.open(image_path) as img:
```

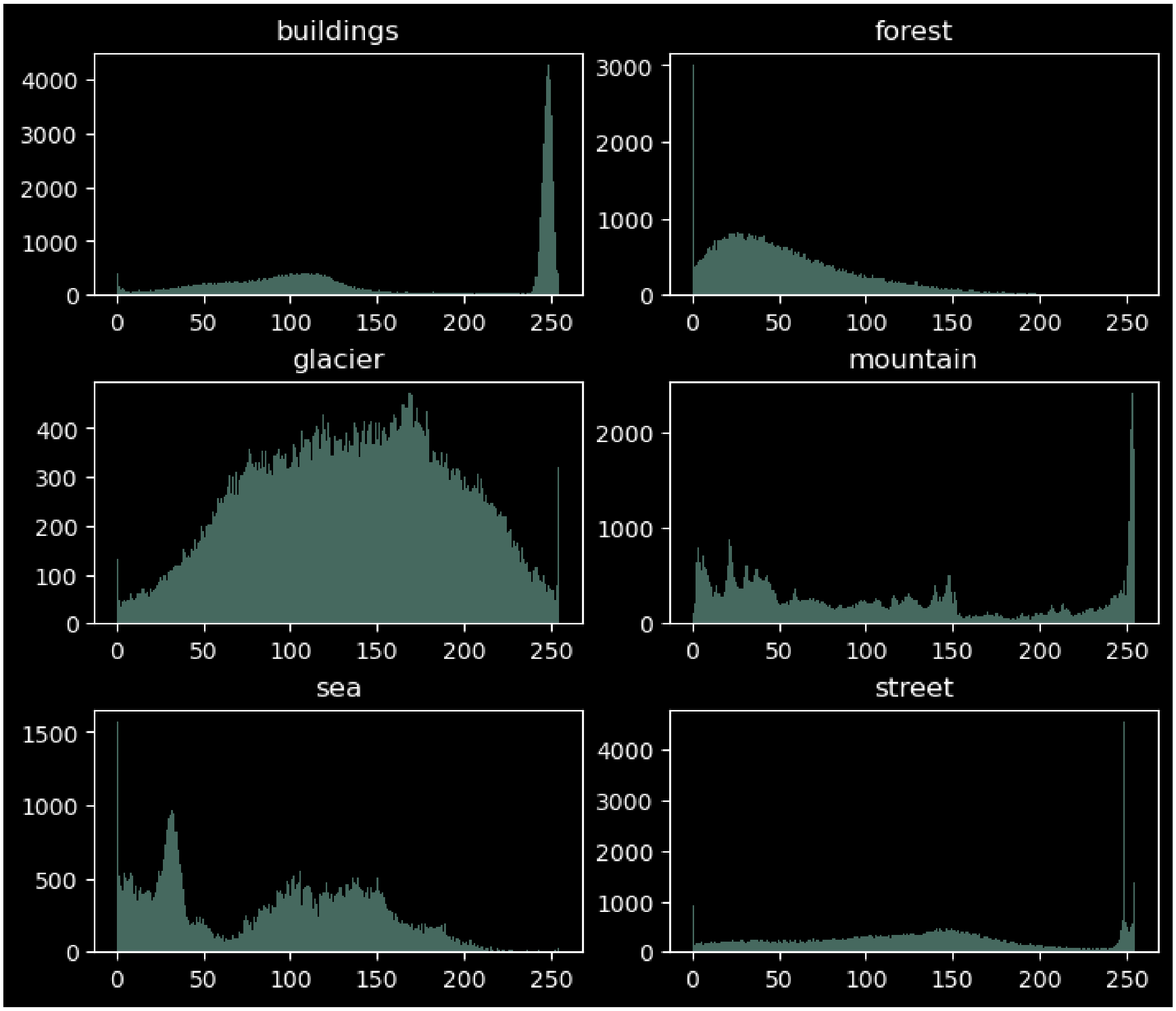
```
            sizes.append(img.size)

sizes = np.array(sizes)
width_vals, width_cts = np.unique(sizes[:, 0], return_counts=True)
height_vals, height_cts = np.unique(sizes[:, 1], return_counts=True)

print(
    f'Widths: {width_vals}'
    f'\nCounts: {width_cts}'
    f'\n\nHeights: {height_vals}'
    f'\nCounts: {height_cts}'
)
```

```
Widths: [150]
Counts: [14034]

Heights: [ 76  81  97 100 102 103 105 108 110 111 113 115 119 120 123 124 131 133
 134 135 136 140 141 142 143 144 145 146 147 149 150]
Counts: [    1     1     1     1     1     1     1     2     1     3     7     1
     1     1     2     1     1     1     2     3     2     1     1     2
     2     2     1     2     1     1 13986]
```

Pixel Intensity Distribution

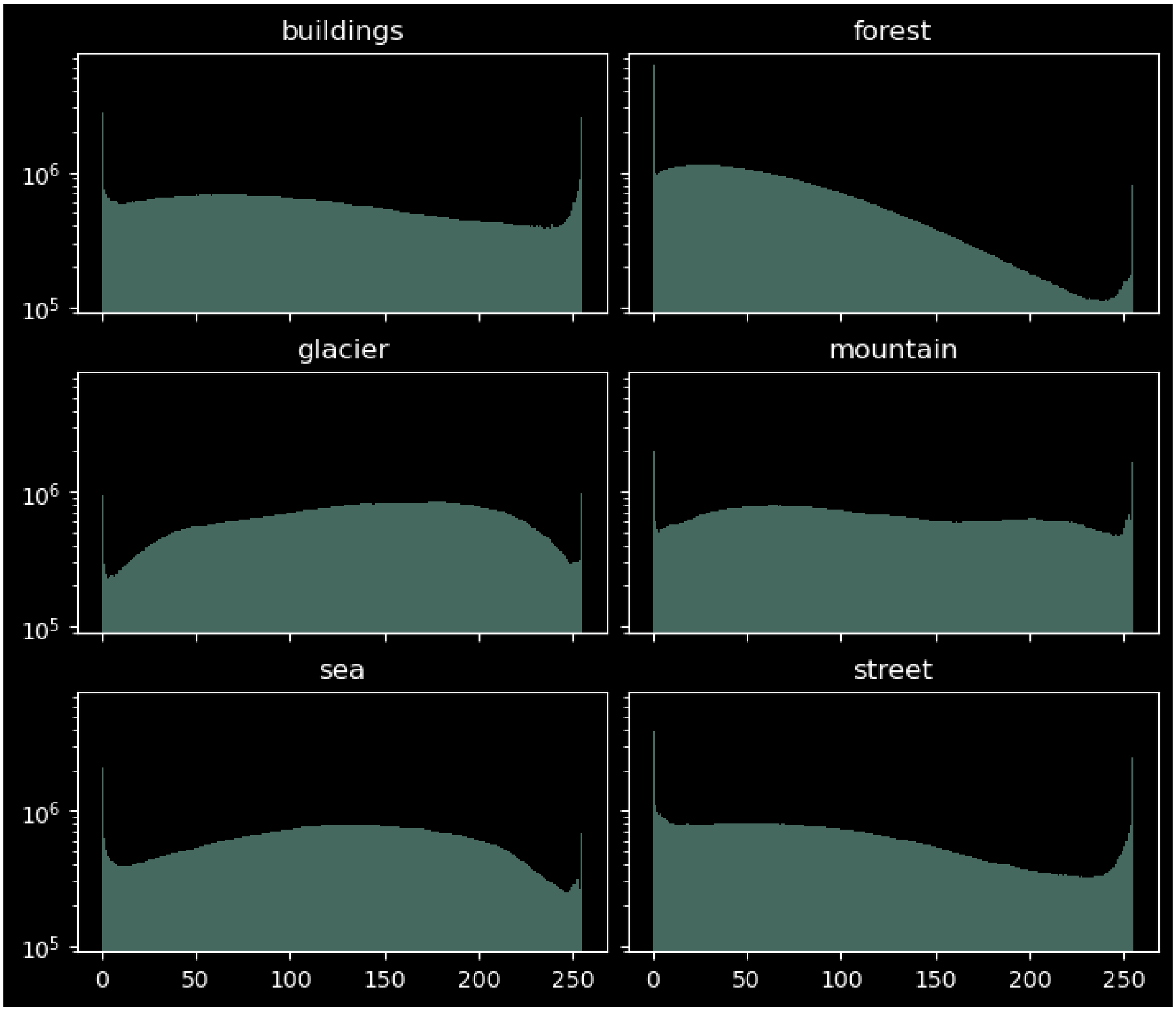
In [ ]:

```
# For a single, random image in each class
fig, axes = plt.subplots(3, 2, figsize=(7,6), layout='constrained')
for folder, ax in zip(os.listdir('data/train'), axes.ravel()):
    class_path = f'data/train/{folder}'
    for image_name in os.listdir(class_path):
        image_path = os.path.join(class_path, image_name)
        with Image.open(image_path) as img:
            img_array = np.array(img)
            ax.hist(img_array.ravel(), bins=256, alpha=0.5, label=f'{folder} (sampl
            break
    ax.set_title(folder)

plt.show()
```



In [ ]:

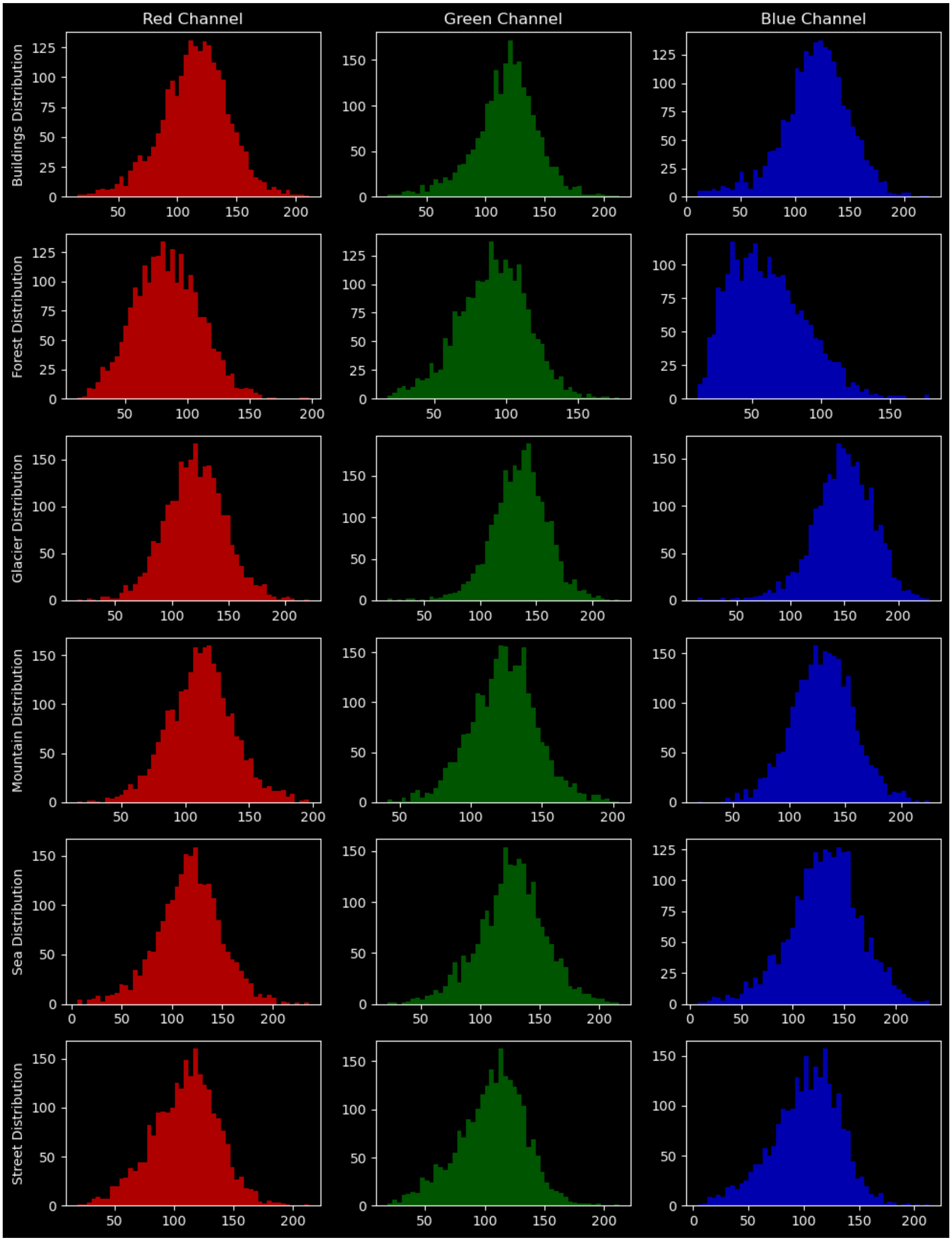
```
# Aggregated across all images
fig, axes = plt.subplots(3, 2, figsize=(7, 6), layout='constrained', sharex=True, s

# Iterate over each class folder
for folder, ax in zip(os.listdir('data/train'), axes.ravel()):
    class_path = f'data/train/{folder}'
    total_hist = np.zeros(256)  # Initialize a histogram to accumulate pixel values

    # Accumulate histogram data from all images in the class folder
    for image_name in os.listdir(class_path):
        image_path = os.path.join(class_path, image_name)
        with Image.open(image_path) as img:
            img_array = np.array(img)
            hist, _ = np.histogram(img_array.ravel(), bins=256, range=[0, 256])
            total_hist += hist  # Accumulate the histograms

    # Plot the accumulated histogram
    ax.hist(np.arange(256), weights=total_hist, bins=256, alpha=0.5, label=f'{folde
    ax.set_title(folder)
    ax.legend().remove()
    ax.set_yscale('log')

plt.show()
```



Distribution Across RGB Color Channels

In [ ]:
```
# Set up the plot
fig, axes = plt.subplots(6, 3, figsize=(10, 13), layout='tight')

# Set titles for columns
axes[0, 0].set_title('Red Channel')
axes[0, 1].set_title('Green Channel')
axes[0, 2].set_title('Blue Channel')

# Iterate over each class directory
for folder, ax_row in zip(os.listdir('data/train'), axes):
    class_path = f'data/train/{folder}'
    colors = []

    # Collect all colors in this class
    for image_name in os.listdir(class_path):
        image_path = os.path.join(class_path, image_name)
        with Image.open(image_path) as img:
            colors.append(np.mean(np.array(img), axis=(0, 1)))
    colors = np.array(colors)

    # Plot histograms for each color channel
```

```
    ax_row[0].hist(colors[:, 0], bins=50, color='red', alpha=0.7)
    ax_row[1].hist(colors[:, 1], bins=50, color='green', alpha=0.7)
    ax_row[2].hist(colors[:, 2], bins=50, color='blue', alpha=0.7)

    # Set class label as the row title
    ax_row[0].set_ylabel(f'{folder.title()} Distribution')

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



Check for Corrupt Images

In [ ]:
```
bad_files = []
for folder in os.listdir('data/train'):
    class_path = f'data/train/{folder}'
    for i, image_name in enumerate(os.listdir(class_path)):
        try:
```

```
            image_path = os.path.join(class_path, image_name)
            with Image.open(image_path) as img:
                img.verify()
        except (IOError, SyntaxError) as e:
            print('Bad file:', image_name)
            bad_files.append(f'{folder}/{image_name}')

print(f'Number of bad files: {len(bad_files)}')
```

```
Number of bad files: 0
```

Preprocessing

Carve Out Validation Subset

In [ ]:
```
# set up vars
train_dir = 'data/train'
val_dir = 'data/val'
num_val_samples = 3000
num_val_samples_per_class = num_val_samples // len(os.listdir(train_dir))

# create validation folder
if not os.path.exists(val_dir):
    os.makedirs(val_dir)

for label in os.listdir(train_dir):
    new_label_dir = os.path.join(val_dir, label)
    if not os.path.exists(new_label_dir):
        os.makedirs(new_label_dir)

print(f'Folders in {val_dir}:', os.listdir(val_dir))

# move random train images to validation
for label in os.listdir(train_dir):
    train_label_dir = os.path.join(train_dir, label)
    val_label_dir = os.path.join(val_dir, label)

    imgs = os.listdir(train_label_dir)
    random.shuffle(imgs)

    for img in imgs[:num_val_samples_per_class]:
        source_path = os.path.join(train_label_dir, img)
        target_path = os.path.join(val_label_dir, img)
        shutil.move(source_path, target_path)

for subset in ['train', 'val', 'test']:
    num_files = 0
    for folder in os.listdir(f'data/{subset}'):
        num_files += len(os.listdir(f'data/{subset}/{folder}'))
    print(f'Number of {subset} files: {num_files}')
```

```
Folders in data/val: ['buildings', 'forest', 'glacier', 'mountain', 'sea', 'street']
Number of train files: 11034
Number of val files: 3000
Number of test files: 3000
```

Preprocess for ResNet

In [ ]:
```
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

image_datasets = {
    x: datasets.ImageFolder(os.path.join('data', x), data_transforms[x])
    for x in ['train', 'val', 'test']
}

dataloaders = {
    x: DataLoader(image_datasets[x], batch_size=32, shuffle=True, num_workers=4)
    for x in ['train', 'val', 'test']
}

dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val', 'test']}

class_names = image_datasets['train'].classes

print(
    f'Class names: {class_names}'
    f'\nDataset sizes: {dataset_sizes}'
)
```

```
Class names: ['buildings', 'forest', 'glacier', 'mountain', 'sea', 'street']
Dataset sizes: {'train': 11034, 'val': 3000, 'test': 3000}
```

Create Flat, Tabular Data for XGB Benchmark

In [ ]:
```
tabular_datasets = {}
means = np.array([0.485, 0.456, 0.406])
stds = np.array([0.229, 0.224, 0.225])
```

```python
for subset in ['train', 'val', 'test']:
    features = []
    labels = []
    for inputs, classes in dataloaders[subset]:
        for i in range(inputs.shape[0]):
            # Convert tensor to numpy array
            normalized_image = inputs[i].numpy()
            # Denormalize
            denorm_image = normalized_image * stds[:, None, None] + means[:, None,
            # Flatten and convert to 1D
            flat_image = denorm_image.transpose(1, 2, 0).reshape(-1)
            features.append(flat_image)
            labels.append(classes[i].item())
    # Add to dict
    tabular_datasets.update({subset: (np.array(features), np.array(labels))})

print(
    f'Train features shape: {tabular_datasets["train"][0].shape}\n'
    f'Train labels shape: {tabular_datasets["train"][1].shape}\n'
    f'Val features shape: {tabular_datasets["val"][0].shape}\n'
    f'Val labels shape: {tabular_datasets["val"][1].shape}\n'
    f'Test features shape: {tabular_datasets["test"][0].shape}\n'
    f'Test labels shape: {tabular_datasets["test"][1].shape}'
)
```

```
Train features shape: (11034, 150528)
Train labels shape: (11034,)
Val features shape: (3000, 150528)
Val labels shape: (3000,)
Test features shape: (3000, 150528)
Test labels shape: (3000,)
```

Generate PCA features

In [ ]:

```python
pca_datasets = {}
pca = PCA(n_components=1000)

for subset in ['train', 'val', 'test']:
    pca.fit(tabular_datasets[subset][0])
    pca_datasets.update({subset: (
        pca.transform(tabular_datasets[subset][0]),
        tabular_datasets[subset][1]
    )})

print(
    f'Train PCA shape: {pca_datasets["train"][0].shape}\n'
    f'Val PCA shape: {pca_datasets["val"][0].shape}\n'
    f'Test PCA shape: {pca_datasets["test"][0].shape}'
)
```

```
Train PCA shape: (11034, 1000)
Val PCA shape: (3000, 1000)
Test PCA shape: (3000, 1000)
```

Benchmark XGBoost Training

In [ ]:

```
# Set up pca data
X_train_pca, y_train = pca_datasets['train']
X_val_pca, y_val = pca_datasets['val']

# Train the model
xgb_pca = XGBClassifier(tree_method='hist', device='cuda', objective='multi:softpro
xgb_pca.fit(X_train_pca, y_train)

# Evaluate the model on validation set
y_pred_pca = xgb_pca.predict(X_val_pca)
print(classification_report(y_val, y_pred_pca, target_names=class_names))
```

```
              precision    recall  f1-score   support

   buildings       0.19      0.27      0.22       500
      forest       0.04      0.03      0.03       500
     glacier       0.19      0.22      0.21       500
    mountain       0.41      0.39      0.40       500
         sea       0.32      0.15      0.21       500
      street       0.20      0.24      0.22       500

    accuracy                           0.22      3000
   macro avg       0.22      0.22      0.21      3000
weighted avg       0.22      0.22      0.21      3000
```

ResNet-50 Setup and Baseline Training

In [ ]:

```
model = models.resnet50(weights='IMAGENET1K_V1')
model
```

Out[ ]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=F
alse)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=F
alse)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), b
ias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), b
ias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), b
ias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
```

```
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (relu): ReLU(inplace=True)
    )
```

```
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running
```

```
_stats=True)
      (relu): ReLU(inplace=True)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
      (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
```

```
stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

In [ ]:

```
# Freeze all the parameters in the network
for param in model.parameters():
    param.requires_grad = False

# Replace the last fully connected layer with a new one with 6 output classes
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 6)

model.fc
```

Out[ ]:

```
Linear(in_features=2048, out_features=6, bias=True)
```

In [ ]:

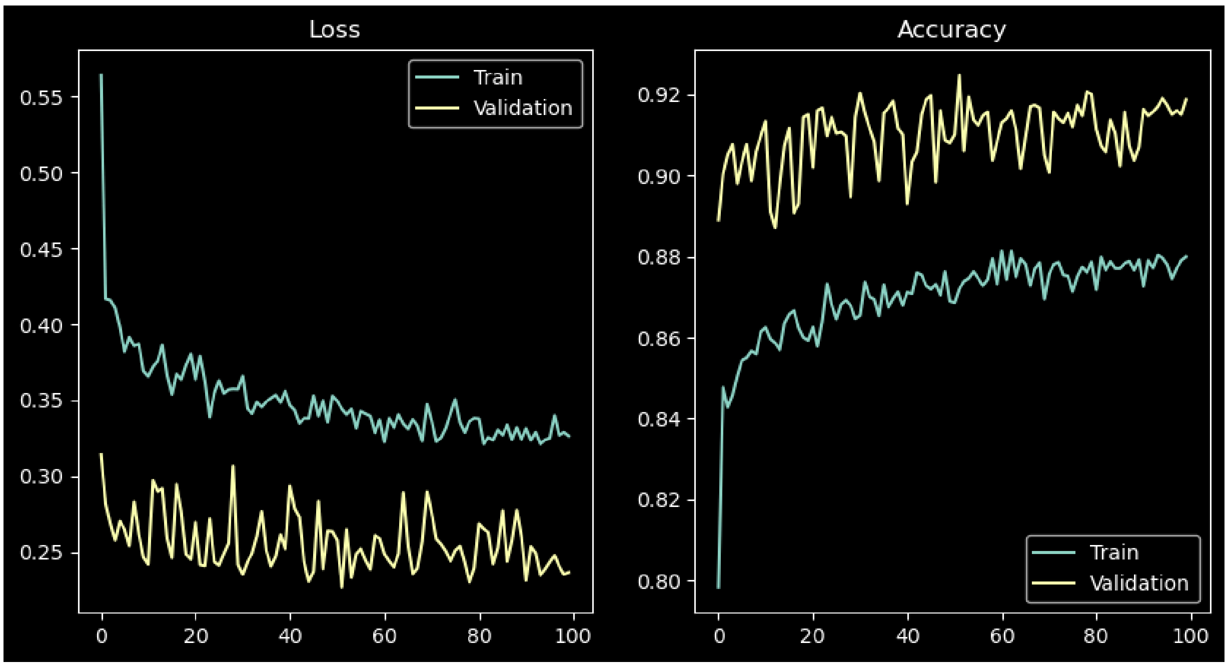
```
# Transfer the model to GPU if available
model = model.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = OPTIMIZER(model.fc.parameters(), lr=LEARNING_RATE)

# Set up variables
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []
start = time()

# Train model
for epoch in range(NUM_EPOCHS):

    if epoch % (NUM_EPOCHS // 10) == 0 or epoch == NUM_EPOCHS - 1:
        print(f'\nEpoch {epoch + 1}/{NUM_EPOCHS} - Time Elapsed: {(time() - start)
```

```
    # Repeat for training and validation
    for phase in ['train', 'val']:
        # Set model to training mode or evaluation mode
        if phase == 'train':
            model.train()
        else:
            model.eval()

        running_loss = 0.0
        running_corrects = 0

        # Iterate over data.
        for inputs, labels in dataloaders[phase]:
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward
            with torch.set_grad_enabled(phase == 'train'):
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)

                # Backward + optimize only if in training phase
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects.double() / dataset_sizes[phase]

        if phase == 'train':
            train_losses.append(epoch_loss)
            train_accuracies.append(epoch_acc)
        else:
            val_losses.append(epoch_loss)
            val_accuracies.append(epoch_acc)

        if epoch % (NUM_EPOCHS // 10) == 0 or epoch == NUM_EPOCHS - 1:
            print(f'{phase.title()} Loss: {epoch_loss:.4f} - Accuracy: {epoch_acc:.

torch.save(model, 'models/baseline_model.pt')
```

```
Epoch 1/100 - Time Elapsed: 0.00 minutes
----------
Train Loss: 0.5641 - Accuracy: 0.7984
Val Loss: 0.3141 - Accuracy: 0.8890

Epoch 11/100 - Time Elapsed: 4.79 minutes
----------
Train Loss: 0.3657 - Accuracy: 0.8624
Val Loss: 0.2416 - Accuracy: 0.9133

Epoch 21/100 - Time Elapsed: 9.40 minutes
----------
Train Loss: 0.3636 - Accuracy: 0.8625
Val Loss: 0.2694 - Accuracy: 0.9020

Epoch 31/100 - Time Elapsed: 14.01 minutes
----------
Train Loss: 0.3659 - Accuracy: 0.8654
Val Loss: 0.2350 - Accuracy: 0.9203

Epoch 41/100 - Time Elapsed: 18.56 minutes
----------
Train Loss: 0.3466 - Accuracy: 0.8711
Val Loss: 0.2933 - Accuracy: 0.8930

Epoch 51/100 - Time Elapsed: 23.09 minutes
----------
Train Loss: 0.3493 - Accuracy: 0.8686
Val Loss: 0.2574 - Accuracy: 0.9100

Epoch 61/100 - Time Elapsed: 27.62 minutes
----------
Train Loss: 0.3224 - Accuracy: 0.8813
Val Loss: 0.2486 - Accuracy: 0.9130

Epoch 71/100 - Time Elapsed: 32.16 minutes
----------
Train Loss: 0.3363 - Accuracy: 0.8756
Val Loss: 0.2755 - Accuracy: 0.9007

Epoch 81/100 - Time Elapsed: 36.70 minutes
----------
Train Loss: 0.3377 - Accuracy: 0.8718
Val Loss: 0.2684 - Accuracy: 0.9113

Epoch 91/100 - Time Elapsed: 41.24 minutes
----------
Train Loss: 0.3312 - Accuracy: 0.8726
Val Loss: 0.2311 - Accuracy: 0.9163

Epoch 100/100 - Time Elapsed: 45.32 minutes
----------
Train Loss: 0.3264 - Accuracy: 0.8799
Val Loss: 0.2364 - Accuracy: 0.9187
```

View Loss Over Epochs

In [ ]:
```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

axes[0].plot(train_losses, label='Train')
axes[0].plot(val_losses, label='Validation')
axes[0].set_title('Loss')
axes[0].legend()

axes[1].plot([x.detach().cpu().numpy() for x in train_accuracies], label='Train')
axes[1].plot([x.detach().cpu().numpy() for x in val_accuracies], label='Validation'
axes[1].set_title('Accuracy')
axes[1].legend()

plt.show()
```



Error Analysis

Classification Report - Training Data

In [ ]:
```
# Set up variables
true_labels = []
pred_labels = []

# Loop through validation set
for inputs, labels in dataloaders['train']:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # Generate predictions
    with torch.no_grad():
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        # Store true and predicted labels
        true_labels.extend(labels.cpu().numpy())
```

```
        pred_labels.extend(preds.cpu().numpy())

# Convert to numpy arrays
true_labels = np.array(true_labels)
pred_labels = np.array(pred_labels)

# Print classification report
print(classification_report(true_labels, pred_labels, target_names=class_names))
```

```
              precision    recall  f1-score   support

   buildings       0.90      0.91      0.91      1691
      forest       0.99      0.97      0.98      1771
     glacier       0.82      0.89      0.86      1904
    mountain       0.86      0.83      0.85      2012
         sea       0.95      0.90      0.93      1774
      street       0.91      0.92      0.91      1882

    accuracy                           0.90     11034
   macro avg       0.91      0.90      0.91     11034
weighted avg       0.90      0.90      0.90     11034
```

Classification Report - Validation Data

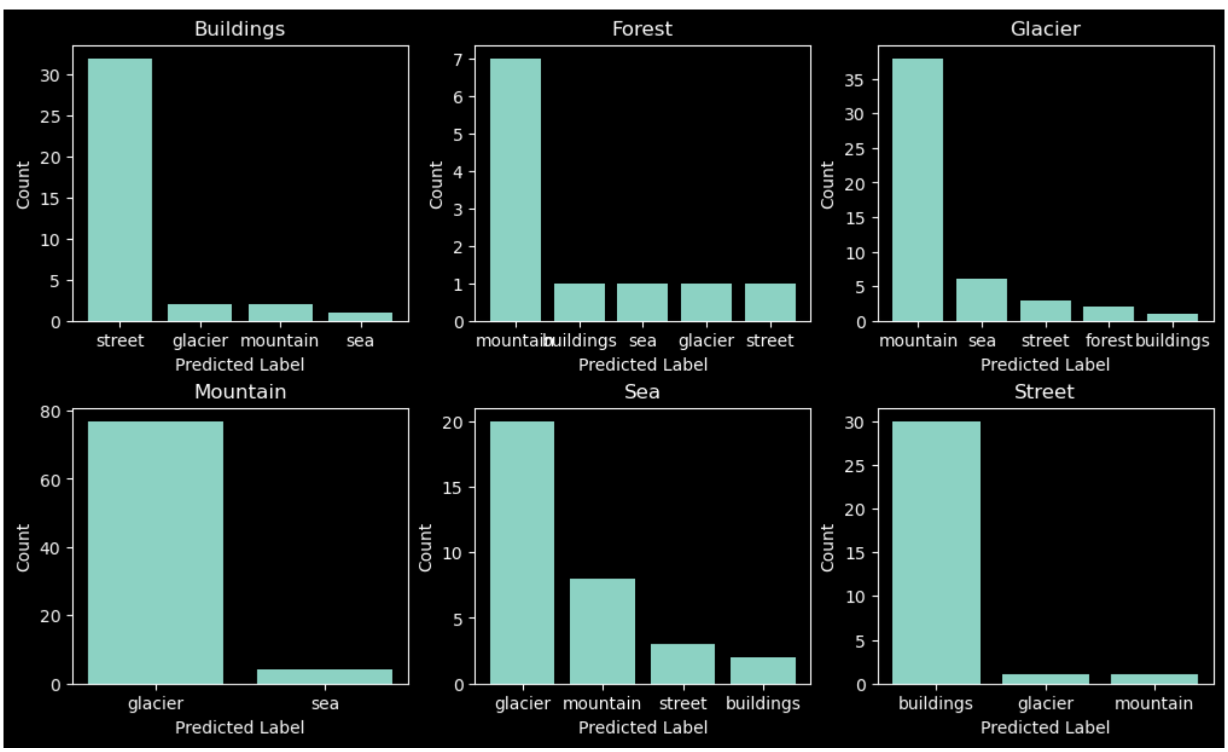
In [ ]:

```
# Set up variables
true_labels = []
pred_labels = []

# Loop through validation set
for inputs, labels in dataloaders['val']:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # Generate predictions
    with torch.no_grad():
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        # Store true and predicted labels
        true_labels.extend(labels.cpu().numpy())
        pred_labels.extend(preds.cpu().numpy())

# Convert to numpy arrays
true_labels = np.array(true_labels)
pred_labels = np.array(pred_labels)

# Log classification report
class_report = classification_report(true_labels, pred_labels, target_names=class_n
class_report = pd.DataFrame(class_report).transpose()
class_report.to_csv('logs/class_report_baseline.csv')

# Print string version
print(classification_report(true_labels, pred_labels, target_names=class_names))
```

```
              precision    recall  f1-score   support

   buildings       0.93      0.93      0.93       500
      forest       1.00      0.98      0.99       500
     glacier       0.82      0.90      0.86       500
    mountain       0.88      0.84      0.86       500
         sea       0.97      0.93      0.95       500
      street       0.92      0.94      0.93       500

    accuracy                           0.92      3000
   macro avg       0.92      0.92      0.92      3000
weighted avg       0.92      0.92      0.92      3000
```

In [ ]:

```python
error_df = pd.DataFrame({'true_label': true_labels, 'pred_label': pred_labels})
error_df = error_df[true_labels != pred_labels].reset_index(drop=True)
error_df.true_label = error_df.apply(lambda row: class_names[row['true_label']], ax
error_df.pred_label = error_df.apply(lambda row: class_names[row['pred_label']], ax

fig, axes = plt.subplots(2, 3, figsize=(10, 6), layout='constrained')
for label, ax in zip(class_names, axes.ravel()):
    label_df = error_df[error_df.true_label == label]
    ax.bar(label_df.pred_label.value_counts().index, label_df.pred_label.value_coun
    ax.set_title(label.title())
    ax.set_xlabel('Predicted Label')
    ax.set_ylabel('Count')
plt.show()
```



Add Dropout

In [ ]:

```python
# Replace the last fully connected layer with a new one, inclusive of dropout
model.fc = nn.Sequential(
```

```
    nn.Dropout(DROPOUT),
    nn.Linear(num_ftrs, 6)
)

model.fc
```

Out[ ]:

```
Sequential(
  (0): Dropout(p=0.3, inplace=False)
  (1): Linear(in_features=2048, out_features=6, bias=True)
)
```

Re-run Training

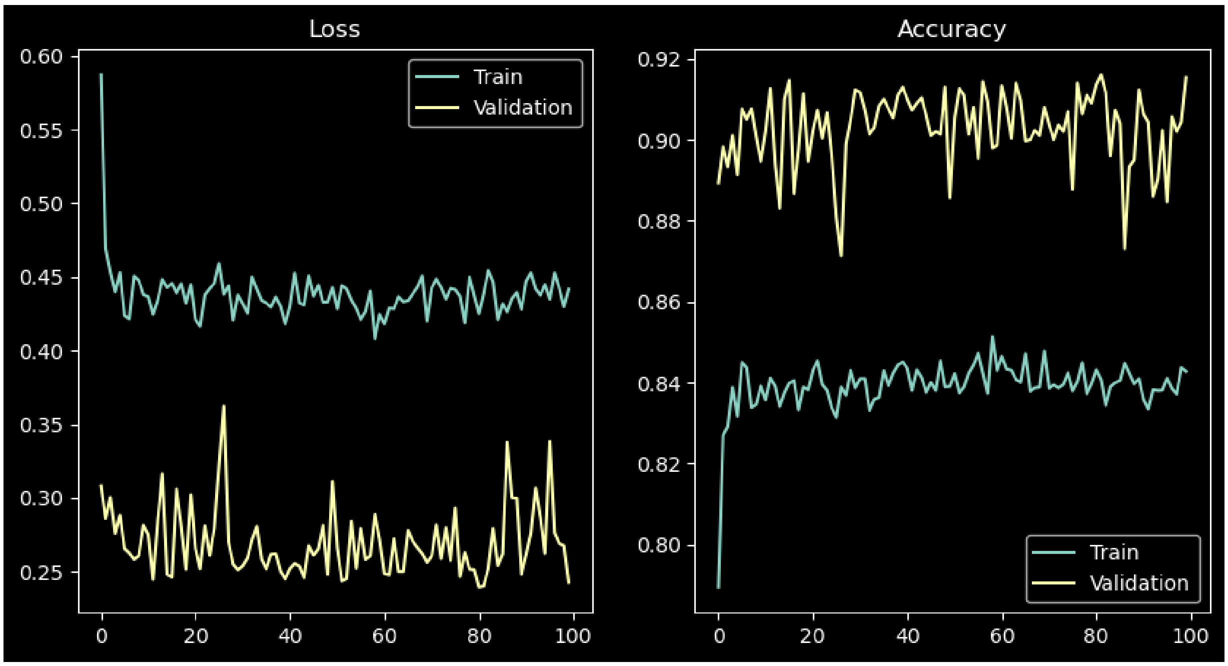
In [ ]:

```
# Transfer the model to GPU if available
model = model.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = OPTIMIZER(model.fc.parameters(), lr=LEARNING_RATE)

# Set up variables
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []
start = time()

# Train model
for epoch in range(NUM_EPOCHS):

    if epoch % (NUM_EPOCHS // 10) == 0 or epoch == NUM_EPOCHS - 1:
        print(f'\nEpoch {epoch + 1}/{NUM_EPOCHS} - Time Elapsed: {(time() - start)

    # Repeat for training and validation
    for phase in ['train', 'val']:
        # Set model to training mode or evaluation mode
        if phase == 'train':
            model.train()
        else:
            model.eval()

        running_loss = 0.0
        running_corrects = 0

        # Iterate over data.
        for inputs, labels in dataloaders[phase]:
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward
            with torch.set_grad_enabled(phase == 'train'):
                outputs = model(inputs)
```

```
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)

                # Backward + optimize only if in training phase
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects.double() / dataset_sizes[phase]

        if phase == 'train':
            train_losses.append(epoch_loss)
            train_accuracies.append(epoch_acc)
        else:
            val_losses.append(epoch_loss)
            val_accuracies.append(epoch_acc)

        if epoch % (NUM_EPOCHS // 10) == 0 or epoch == NUM_EPOCHS - 1:
            print(f'{phase.title()} Loss: {epoch_loss:.4f} - Accuracy: {epoch_acc:.

torch.save(model, 'models/dropout_model.pt')
```

```
Epoch 1/100 - Time Elapsed: 0.00 minutes
----------
Train Loss: 0.5871 - Accuracy: 0.7895
Val Loss: 0.3081 - Accuracy: 0.8893

Epoch 11/100 - Time Elapsed: 4.52 minutes
----------
Train Loss: 0.4366 - Accuracy: 0.8357
Val Loss: 0.2752 - Accuracy: 0.9020

Epoch 21/100 - Time Elapsed: 9.05 minutes
----------
Train Loss: 0.4212 - Accuracy: 0.8430
Val Loss: 0.2657 - Accuracy: 0.9023

Epoch 31/100 - Time Elapsed: 13.60 minutes
----------
Train Loss: 0.4317 - Accuracy: 0.8409
Val Loss: 0.2539 - Accuracy: 0.9117

Epoch 41/100 - Time Elapsed: 18.16 minutes
----------
Train Loss: 0.4297 - Accuracy: 0.8435
Val Loss: 0.2519 - Accuracy: 0.9100

Epoch 51/100 - Time Elapsed: 22.70 minutes
----------
Train Loss: 0.4283 - Accuracy: 0.8422
Val Loss: 0.2663 - Accuracy: 0.9053

Epoch 61/100 - Time Elapsed: 27.25 minutes
----------
Train Loss: 0.4180 - Accuracy: 0.8465
Val Loss: 0.2487 - Accuracy: 0.9133

Epoch 71/100 - Time Elapsed: 31.80 minutes
----------
Train Loss: 0.4425 - Accuracy: 0.8386
Val Loss: 0.2605 - Accuracy: 0.9037

Epoch 81/100 - Time Elapsed: 36.35 minutes
----------
Train Loss: 0.4251 - Accuracy: 0.8431
Val Loss: 0.2395 - Accuracy: 0.9137

Epoch 91/100 - Time Elapsed: 40.90 minutes
----------
Train Loss: 0.4470 - Accuracy: 0.8359
Val Loss: 0.2615 - Accuracy: 0.9063

Epoch 100/100 - Time Elapsed: 44.99 minutes
----------
Train Loss: 0.4416 - Accuracy: 0.8428
Val Loss: 0.2428 - Accuracy: 0.9153
```

In [ ]:
```python
# Plot losses and accuracies
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

axes[0].plot(train_losses, label='Train')
axes[0].plot(val_losses, label='Validation')
axes[0].set_title('Loss')
axes[0].legend()

axes[1].plot([x.detach().cpu().numpy() for x in train_accuracies], label='Train')
axes[1].plot([x.detach().cpu().numpy() for x in val_accuracies], label='Validation'
axes[1].set_title('Accuracy')
axes[1].legend()

plt.show()
```



In [ ]:
```python
# Set up variables
true_labels = []
pred_labels = []

# Loop through validation set
for inputs, labels in dataloaders['val']:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # Generate predictions
    with torch.no_grad():
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        # Store true and predicted labels
        true_labels.extend(labels.cpu().numpy())
        pred_labels.extend(preds.cpu().numpy())

# Convert to numpy arrays
true_labels = np.array(true_labels)
pred_labels = np.array(pred_labels)
```

```
# Log classification report
class_report = classification_report(true_labels, pred_labels, target_names=class_n
class_report = pd.DataFrame(class_report).transpose()
class_report.to_csv('logs/class_report_dropout.csv')

# Print string version
print(classification_report(true_labels, pred_labels, target_names=class_names))
```

```
              precision    recall  f1-score   support

   buildings       0.91      0.94      0.92       500
      forest       0.99      0.98      0.99       500
     glacier       0.87      0.82      0.85       500
    mountain       0.83      0.87      0.85       500
         sea       0.95      0.97      0.96       500
      street       0.93      0.91      0.92       500

    accuracy                           0.92      3000
   macro avg       0.92      0.92      0.92      3000
weighted avg       0.92      0.92      0.92      3000
```

Add More Data Augmentations

In [ ]:

```
data_transforms = {
    'train': transforms.Compose([
        # Original transforms
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        # Additional transforms
        transforms.RandomRotation(15),
        transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.
        # Conversion and normalization
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

image_datasets = {
    x: datasets.ImageFolder(os.path.join('data', x), data_transforms[x])
    for x in ['train', 'val', 'test']
```

```python
}

dataloaders = {
    x: DataLoader(image_datasets[x], batch_size=32, shuffle=True, num_workers=4)
    for x in ['train', 'val', 'test']
}

dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val', 'test']}

class_names = image_datasets['train'].classes

print(
    f'Class names: {class_names}'
    f'\nDataset sizes: {dataset_sizes}'
)
```

```
Class names: ['buildings', 'forest', 'glacier', 'mountain', 'sea', 'street']
Dataset sizes: {'train': 11034, 'val': 3000, 'test': 3000}
```

Re-run model training

In [ ]:

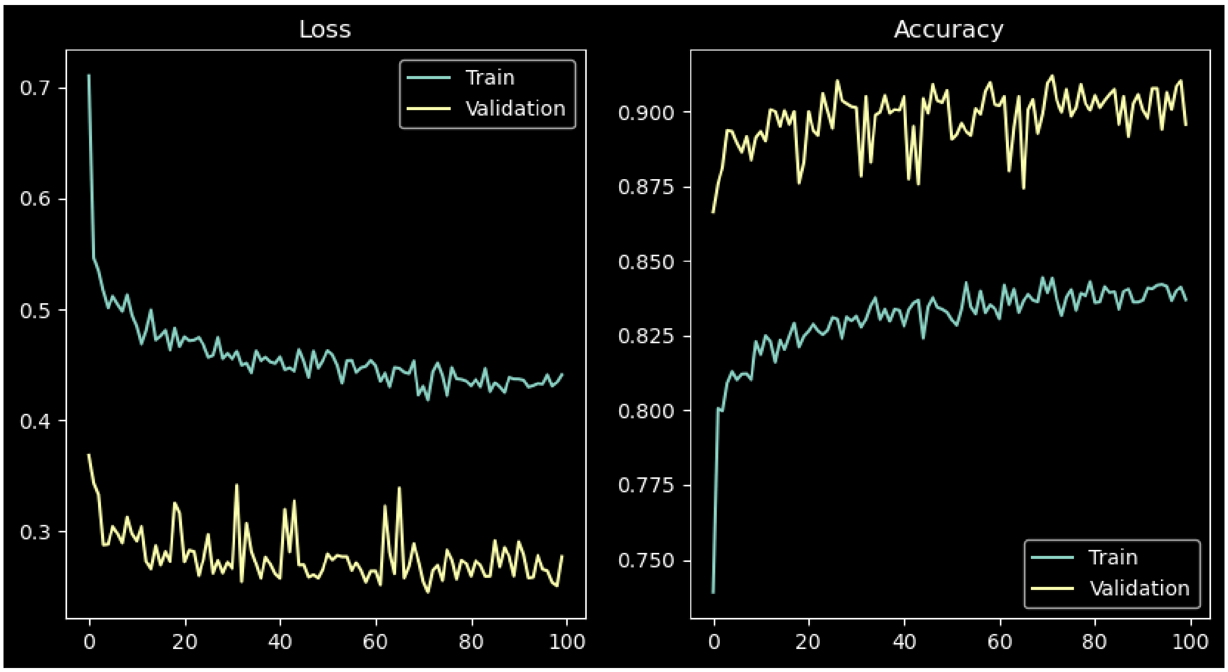
```python
# Re-instantiate model (without dropout)
model = models.resnet50(weights='IMAGENET1K_V1')

# Freeze all the parameters in the network
for param in model.parameters():
    param.requires_grad = False

# Replace the last fully connected layer with a new one with 6 output classes
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 6)

# Move to GPU
model = model.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = OPTIMIZER(model.fc.parameters(), lr=LEARNING_RATE)

# Set up variables
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []
start = time()

# Train model
for epoch in range(NUM_EPOCHS):

    if epoch % (NUM_EPOCHS // 10) == 0 or epoch == NUM_EPOCHS - 1:
        print(f'\nEpoch {epoch + 1}/{NUM_EPOCHS} - Time Elapsed: {(time() - start)

    # Repeat for training and validation
    for phase in ['train', 'val']:
        # Set model to training mode or evaluation mode
        if phase == 'train':
```

```
            model.train()
        else:
            model.eval()

        running_loss = 0.0
        running_corrects = 0

        # Iterate over data.
        for inputs, labels in dataloaders[phase]:
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward
            with torch.set_grad_enabled(phase == 'train'):
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)

                # Backward + optimize only if in training phase
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects.double() / dataset_sizes[phase]

        if phase == 'train':
            train_losses.append(epoch_loss)
            train_accuracies.append(epoch_acc)
        else:
            val_losses.append(epoch_loss)
            val_accuracies.append(epoch_acc)

        if epoch % (NUM_EPOCHS // 10) == 0 or epoch == NUM_EPOCHS - 1:
            print(f'{phase.title()} Loss: {epoch_loss:.4f} - Accuracy: {epoch_acc:.

torch.save(model, 'models/augments_model.pt')
```

```
Epoch 1/100 - Time Elapsed: 0.00 minutes
----------
Train Loss: 0.7106 - Accuracy: 0.7390
Val Loss: 0.3684 - Accuracy: 0.8663

Epoch 11/100 - Time Elapsed: 5.21 minutes
----------
Train Loss: 0.4842 - Accuracy: 0.8186
Val Loss: 0.2910 - Accuracy: 0.8933

Epoch 21/100 - Time Elapsed: 10.39 minutes
----------
Train Loss: 0.4752 - Accuracy: 0.8264
Val Loss: 0.2722 - Accuracy: 0.9000

Epoch 31/100 - Time Elapsed: 15.59 minutes
----------
Train Loss: 0.4551 - Accuracy: 0.8315
Val Loss: 0.2663 - Accuracy: 0.9013

Epoch 41/100 - Time Elapsed: 20.79 minutes
----------
Train Loss: 0.4570 - Accuracy: 0.8281
Val Loss: 0.2576 - Accuracy: 0.9050

Epoch 51/100 - Time Elapsed: 25.99 minutes
----------
Train Loss: 0.4629 - Accuracy: 0.8303
Val Loss: 0.2796 - Accuracy: 0.8907

Epoch 61/100 - Time Elapsed: 31.19 minutes
----------
Train Loss: 0.4491 - Accuracy: 0.8306
Val Loss: 0.2640 - Accuracy: 0.9020

Epoch 71/100 - Time Elapsed: 36.39 minutes
----------
Train Loss: 0.4305 - Accuracy: 0.8392
Val Loss: 0.2548 - Accuracy: 0.9093

Epoch 81/100 - Time Elapsed: 41.60 minutes
----------
Train Loss: 0.4309 - Accuracy: 0.8360
Val Loss: 0.2594 - Accuracy: 0.9053

Epoch 91/100 - Time Elapsed: 46.80 minutes
----------
Train Loss: 0.4373 - Accuracy: 0.8368
Val Loss: 0.2905 - Accuracy: 0.9003

Epoch 100/100 - Time Elapsed: 51.49 minutes
----------
Train Loss: 0.4411 - Accuracy: 0.8370
Val Loss: 0.2772 - Accuracy: 0.8957
```

In [ ]:

```
# Plot losses and accuracies
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

axes[0].plot(train_losses, label='Train')
axes[0].plot(val_losses, label='Validation')
axes[0].set_title('Loss')
axes[0].legend()

axes[1].plot([x.detach().cpu().numpy() for x in train_accuracies], label='Train')
axes[1].plot([x.detach().cpu().numpy() for x in val_accuracies], label='Validation'
axes[1].set_title('Accuracy')
axes[1].legend()

plt.show()
```



In [ ]:

```
# Set up variables
true_labels = []
pred_labels = []

# Loop through validation set
for inputs, labels in dataloaders['val']:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # Generate predictions
    with torch.no_grad():
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        # Store true and predicted labels
        true_labels.extend(labels.cpu().numpy())
        pred_labels.extend(preds.cpu().numpy())

# Convert to numpy arrays
true_labels = np.array(true_labels)
pred_labels = np.array(pred_labels)
```

```
# Log classification report
class_report = classification_report(true_labels, pred_labels, target_names=class_n
class_report = pd.DataFrame(class_report).transpose()
class_report.to_csv('logs/class_report_augments.csv')

# Print string version
print(classification_report(true_labels, pred_labels, target_names=class_names))
```

```
              precision    recall  f1-score   support

   buildings       0.91      0.90      0.90       500
      forest       1.00      0.97      0.98       500
     glacier       0.78      0.90      0.84       500
    mountain       0.89      0.72      0.80       500
         sea       0.92      0.97      0.94       500
      street       0.89      0.92      0.91       500

    accuracy                           0.90      3000
   macro avg       0.90      0.90      0.89      3000
weighted avg       0.90      0.90      0.89      3000
```

Unfreeze Additional Layers

In [ ]:

```
# View model layers
for name, _ in model.named_children():
    print(name)
```

```
conv1
bn1
relu
maxpool
layer1
layer2
layer3
layer4
avgpool
fc
```

In [ ]:

```
# Unfreeze layers 3 and 4
for name, child in model.named_children():
    if name in ['layer3', 'layer4', 'fc']:
        for param in child.parameters():
            param.requires_grad = True
    else:
        for param in child.parameters():
            param.requires_grad = False

# Check the requires_grad attribute for each parameter layer
for name, child in model.named_children():
    counter = 0
    for param in child.parameters():
        if param.requires_grad:
            counter += 1
```

```
    if counter == len(list(child.parameters())):
        print(f'{name}: unfrozen')
    elif counter == 0:
        print(f'{name}: frozen')
    else:
        print(f'{name}: partially frozen - {counter} of {len(list(child.parameters(
```

```
conv1: frozen
bn1: frozen
relu: unfrozen
maxpool: unfrozen
layer1: frozen
layer2: frozen
layer3: unfrozen
layer4: unfrozen
avgpool: unfrozen
fc: unfrozen
```

NOTE: the `relu`, `maxpool` and `avgpool` layers all have zero trainable parameters, so therefore appear as frozen.

Re-run model training

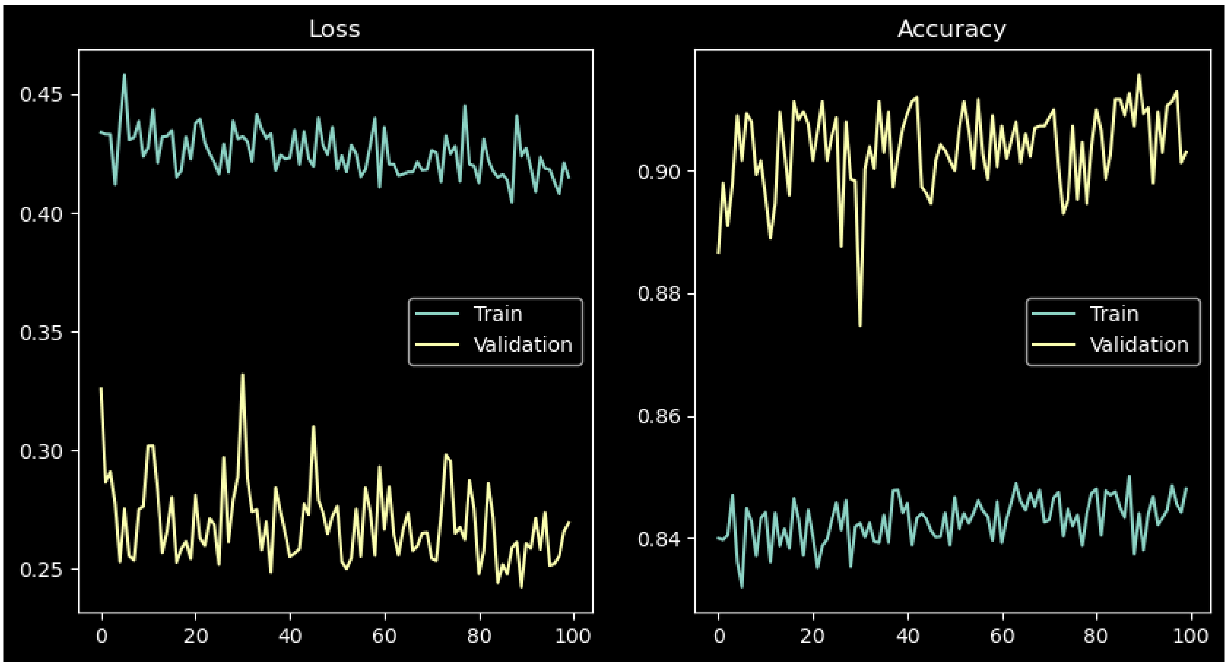
In [ ]:

```
# Transfer the model to GPU
model = model.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = OPTIMIZER(model.fc.parameters(), lr=LEARNING_RATE)

# Set up variables
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []
start = time()

# Train model
for epoch in range(NUM_EPOCHS):

    if epoch % (NUM_EPOCHS // 10) == 0 or epoch == NUM_EPOCHS - 1:
        print(f'\nEpoch {epoch + 1}/{NUM_EPOCHS} - Time Elapsed: {(time() - start)

    # Repeat for training and validation
    for phase in ['train', 'val']:
        # Set model to training mode or evaluation mode
        if phase == 'train':
            model.train()
        else:
            model.eval()

        running_loss = 0.0
        running_corrects = 0

        # Iterate over data.
```

```
        for inputs, labels in dataloaders[phase]:
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward
            with torch.set_grad_enabled(phase == 'train'):
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)

                # Backward + optimize only if in training phase
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects.double() / dataset_sizes[phase]

        if phase == 'train':
            train_losses.append(epoch_loss)
            train_accuracies.append(epoch_acc)
        else:
            val_losses.append(epoch_loss)
            val_accuracies.append(epoch_acc)

        if epoch % (NUM_EPOCHS // 10) == 0 or epoch == NUM_EPOCHS - 1:
            print(f'{phase.title()} Loss: {epoch_loss:.4f} - Accuracy: {epoch_acc:.

torch.save(model, 'models/unfrozen_model.pt')
```

```
Epoch 1/100 - Time Elapsed: 0.00 minutes
----------
Train Loss: 0.4338 - Accuracy: 0.8399
Val Loss: 0.3258 - Accuracy: 0.8867

Epoch 11/100 - Time Elapsed: 5.42 minutes
----------
Train Loss: 0.4275 - Accuracy: 0.8442
Val Loss: 0.3018 - Accuracy: 0.8957

Epoch 21/100 - Time Elapsed: 10.86 minutes
----------
Train Loss: 0.4378 - Accuracy: 0.8402
Val Loss: 0.2812 - Accuracy: 0.9017

Epoch 31/100 - Time Elapsed: 16.27 minutes
----------
Train Loss: 0.4320 - Accuracy: 0.8424
Val Loss: 0.3318 - Accuracy: 0.8747

Epoch 41/100 - Time Elapsed: 21.69 minutes
----------
Train Loss: 0.4231 - Accuracy: 0.8457
Val Loss: 0.2550 - Accuracy: 0.9093

Epoch 51/100 - Time Elapsed: 27.11 minutes
----------
Train Loss: 0.4182 - Accuracy: 0.8467
Val Loss: 0.2764 - Accuracy: 0.9000

Epoch 61/100 - Time Elapsed: 32.58 minutes
----------
Train Loss: 0.4360 - Accuracy: 0.8392
Val Loss: 0.2667 - Accuracy: 0.9073

Epoch 71/100 - Time Elapsed: 38.01 minutes
----------
Train Loss: 0.4263 - Accuracy: 0.8429
Val Loss: 0.2543 - Accuracy: 0.9087

Epoch 81/100 - Time Elapsed: 43.45 minutes
----------
Train Loss: 0.4126 - Accuracy: 0.8480
Val Loss: 0.2480 - Accuracy: 0.9100

Epoch 91/100 - Time Elapsed: 48.88 minutes
----------
Train Loss: 0.4273 - Accuracy: 0.8380
Val Loss: 0.2606 - Accuracy: 0.9093

Epoch 100/100 - Time Elapsed: 53.79 minutes
----------
Train Loss: 0.4150 - Accuracy: 0.8480
Val Loss: 0.2693 - Accuracy: 0.9030
```

In [ ]:
```
# Plot losses and accuracies
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

axes[0].plot(train_losses, label='Train')
axes[0].plot(val_losses, label='Validation')
axes[0].set_title('Loss')
axes[0].legend()

axes[1].plot([x.detach().cpu().numpy() for x in train_accuracies], label='Train')
axes[1].plot([x.detach().cpu().numpy() for x in val_accuracies], label='Validation'
axes[1].set_title('Accuracy')
axes[1].legend()

plt.show()
```



In [ ]:
```
# Set up variables
true_labels = []
pred_labels = []

# Loop through validation set
for inputs, labels in dataloaders['val']:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # Generate predictions
    with torch.no_grad():
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        # Store true and predicted labels
        true_labels.extend(labels.cpu().numpy())
        pred_labels.extend(preds.cpu().numpy())

# Convert to numpy arrays
true_labels = np.array(true_labels)
pred_labels = np.array(pred_labels)
```

```
# Log classification report
class_report = classification_report(true_labels, pred_labels, target_names=class_n
class_report = pd.DataFrame(class_report).transpose()
class_report.to_csv('logs/class_report_unfrozen.csv')

# Print string version
print(classification_report(true_labels, pred_labels, target_names=class_names))
```

```
              precision    recall  f1-score   support

   buildings       0.95      0.85      0.90       500
      forest       0.99      0.97      0.98       500
     glacier       0.90      0.78      0.84       500
    mountain       0.82      0.88      0.85       500
         sea       0.90      0.98      0.94       500
      street       0.88      0.95      0.91       500

    accuracy                           0.90      3000
   macro avg       0.91      0.90      0.90      3000
weighted avg       0.91      0.90      0.90      3000
```

Final Model Evaluation

In [ ]:
```
# Load Best Model - BASELINE
model = torch.load('models/baseline_model.pt')

# Save model
torch.save(model, 'models/final_model.pt')

# Set up variables
true_labels = []
pred_labels = []

# Loop through validation set
for inputs, labels in dataloaders['test']:
    inputs = inputs.to(device)
    labels = labels.to(device)

    # Generate predictions
    with torch.no_grad():
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        # Store true and predicted labels
        true_labels.extend(labels.cpu().numpy())
        pred_labels.extend(preds.cpu().numpy())

# Convert to numpy arrays
true_labels = np.array(true_labels)
pred_labels = np.array(pred_labels)

# Log classification report
class_report = classification_report(true_labels, pred_labels, target_names=class_n
```

```
class_report = pd.DataFrame(class_report).transpose()
class_report.to_csv('logs/class_report_final.csv')

# Print string version
print(classification_report(true_labels, pred_labels, target_names=class_names))
```

```
              precision    recall  f1-score   support

   buildings       0.94      0.92      0.93       437
      forest       1.00      0.98      0.99       474
     glacier       0.83      0.89      0.86       553
    mountain       0.89      0.83      0.86       525
         sea       0.96      0.95      0.96       510
      street       0.93      0.95      0.94       501

    accuracy                           0.92      3000
   macro avg       0.92      0.92      0.92      3000
weighted avg       0.92      0.92      0.92      3000
```

Compare All Model Results

In [ ]:

```
# Create results dataframe
results_df = pd.DataFrame()

# Loop through model results
for model_type in ['baseline', 'dropout', 'augments', 'unfrozen', 'final']:
    error_df = pd.read_csv(f'logs/class_report_{model_type}.csv', index_col=0)
    accuracy = error_df.loc['accuracy', 'precision']
    precision = error_df.loc['macro avg', 'precision']
    recall = error_df.loc['macro avg', 'recall']
    f1 = error_df.loc['macro avg', 'f1-score']
    # Add results to dataframe
    results_df = pd.concat([results_df, pd.DataFrame({
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1': f1
    }, index=[model_type])], axis=0)

# Print results
results_df
```

Out[ ]:

	accuracy	precision	recall	f1
baseline	0.918667	0.920724	0.918667	0.919169
dropout	0.915333	0.915749	0.915333	0.915258
augments	0.895667	0.898505	0.895667	0.894761
unfrozen	0.903000	0.905552	0.903000	0.902429
final	0.920000	0.924146	0.921752	0.922514