Vision Transformer

Requirements/Setup

In [2]:
```
!pip install opencv-python
```

```
Requirement already satisfied: opencv-python in c:\users\anders\miniconda3\li
b\site-packages (4.6.0.66)
Requirement already satisfied: numpy>=1.14.5 in c:\users\anders\miniconda3\li
b\site-packages (from opencv-python) (1.21.5)
```

In [3]:
```
!pip install einops
```

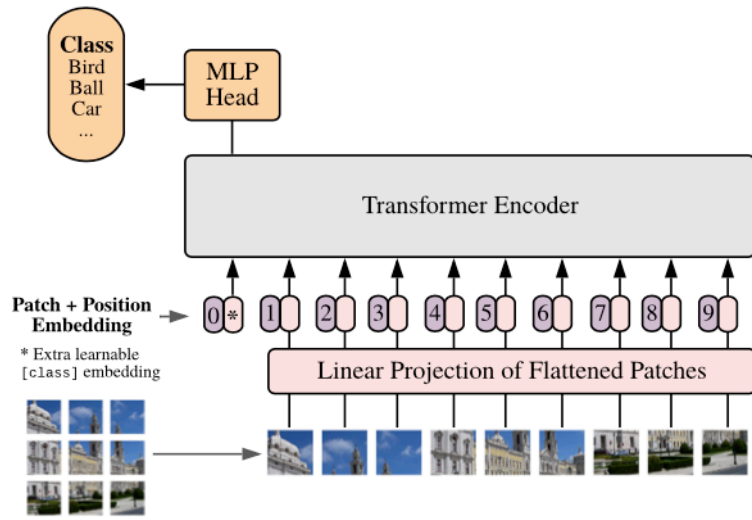
```
Requirement already satisfied: einops in c:\users\anders\miniconda3\lib\site-
packages (0.6.0)
```

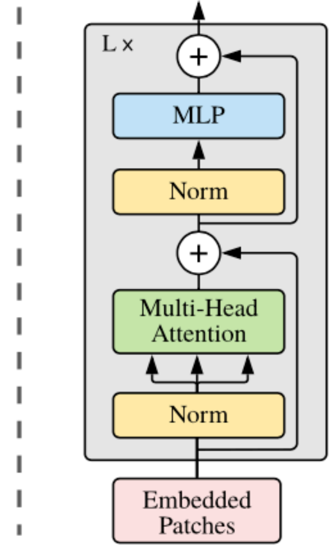
In [45]:
```
import test
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import random
import copy
import einops
import torchvision
import torchvision.transforms as transforms
from matplotlib import pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
from tqdm import trange, tqdm
import cv2
from torchvision.models.feature_extraction import create_feature_extractor, get
```

In [5]:
```
# This cell autoreloads the notebook when you change you python file code.
# If you think the notebook did not reload, rerun this cell.
%load_ext autoreload
%autoreload 2
```

In [6]:
```
def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
torch_device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Here is the basic vision transformer described in the paper. However, we will implement a version where we pass images into a CNN before splitting the image into patches.


Vision Transformer (ViT)
Class
Bird
Ball
Car
...
MLP
Head
Transformer Encoder
Patch + Position
Embedding
* Extra learnable
[class] embedding
0 *
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches


Transformer Encoder
L ×
+
MLP
Norm
+
Multi-Head
Attention
Norm
Embedded
Patches

In [25]:
```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(3, 32, 7, padding=3),
            nn.Conv2d(32, 32, 3, padding=1),
            nn.Conv2d(32, 3, 1, padding=0),
        )
        self.initialize_weights()

    def initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.xavier_uniform_(m.weight)
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)

    def forward(self, x):
        return self.cnn(x)

class MLPBlock(nn.Module):
    "Feed forward block for the Encoder layer"
    def __init__(self, input_dim, feed_forward_dim, dropout_rate=0.0):
        super().__init__()
        self.input_dim = input_dim
        self.feed_forward_dim = feed_forward_dim
        self.dropout_rate = dropout_rate
        self.lin1 = nn.Linear(input_dim, feed_forward_dim)
        self.activation = nn.GELU()
        self.dropout = nn.Dropout(p=dropout_rate)
        self.lin2 = nn.Linear(feed_forward_dim, input_dim)

    def forward(self, input):
        input = self.lin1(input)
        input = self.activation(input)
        input = self.dropout(input)
        input = self.lin2(input)
        input = self.dropout(input)
        return input
```

(b) Encoder

The transformer encoder is main architecture we will implement. Implement an encoder block and run the check.

In [15]:
```
class EncoderBlock(nn.Module):
    def __init__(self, input_dim, feed_forward_dim, num_heads, dropout_rate=0.1
        super().__init__()
        self.input_dim = input_dim
        self.feed_forward_dim = feed_forward_dim
        self.dropout_rate = dropout_rate
        self.mlpblock = MLPBlock(input_dim, feed_forward_dim, dropout_rate=drop
        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.dropout1 = nn.Dropout(dropout_rate)
        self.self_attention = nn.MultiheadAttention(input_dim, num_heads, dropo

    def forward(self, input):
        x = input
        y = self.self_attention(x, x, x, need_weights=False)[0]
        y = self.dropout1(y)
        x = self.norm1(x + y)
        y = self.mlpblock(x)
        x = self.norm2(x + y)
        return x
    def forward_scores(self, input):
        x = input
        y, scores = self.self_attention(x, x, x, need_weights=True, average_att
        y = self.dropout1(y)
        x = self.norm1(x + y)
        y = self.mlpblock(x)
        x = self.norm2(x + y)
        return x, scores
```

In [16]:
```
def check_encoder_block():
    batch_size = 5
    input_dim = 12
    feed_forward_dim = 10
    num_heads = 4
    dropout = 0

    torch_device = 'cuda' if torch.cuda.is_available() else 'cpu'
    set_seed(182)

    encoder_block = EncoderBlock(input_dim=input_dim, feed_forward_dim=feed_fo
    encoder_block = encoder_block.to(torch_device)
    #torch.save(encoder_block.state_dict(),"checks/encoder_block")
    encoder_block.load_state_dict(torch.load("checks/encoder_block"))

    #inputs = torch.randn((batch_size, input_dim))
    #torch.save(inputs, "checks/encoder_block_input")
    inputs = torch.load("checks/encoder_block_input").to(torch_device)

    encoder_block.to(torch_device)
    actual_output = encoder_block(inputs)
    #torch.save(actual_output, "checks/encoder_block_output")

    expected_output = torch.load("checks/encoder_block_output").to(torch_device
    print("Error is:", torch.sum(torch.abs(expected_output-actual_output)).ite
check_encoder_block()
```

```
Error is: 3.4347176551818848e-06 (error should be around 0)
```

In [39]:
```
class Encoder(nn.Module):
    def __init__(self, input_dim, feed_forward_dim, num_heads, n_layers, dropou
        super().__init__()
        self.input_dim = input_dim
        self.feed_forward_dim = feed_forward_dim
        self.dropout_rate = dropout_rate
        temp = EncoderBlock(input_dim, feed_forward_dim, num_heads, dropout_ra
        self.encoder_list = nn.ModuleList([copy.deepcopy(temp) for _ in range(

    def forward(self, input):
        x = input
        for enc in self.encoder_list:
            x = enc(x)
        return x

    def forward_w_attn(self, input):
        x = input
        atten_scores = []
        for enc in self.encoder_list:
            x, scores = enc.forward_scores(x)
            atten_scores.append(scores)
        return x, torch.stack(atten_scores)
```

In [13]:
```
def check_encoder():
    batch_size = 5
    input_dim = 12
    feed_forward_dim = 10
    num_heads = 4
    n_layers = 3

    torch_device = 'cuda' if torch.cuda.is_available() else 'cpu'
    set_seed(182)

    encoder = Encoder(input_dim=input_dim, feed_forward_dim=feed_forward_dim, r
    encoder = encoder.to(torch_device)
    #torch.save(encoder.state_dict(),"checks/encoder")
    encoder.load_state_dict(torch.load("checks/encoder"))

    #inputs = torch.randn((batch_size, input_dim))
    #torch.save(inputs, "checks/encoder_input")
    inputs = torch.load("checks/encoder_input").to(torch_device)

    encoder.to(torch_device)
    actual_output = encoder(inputs)
    #torch.save(actual_output, "checks/encoder_output")

    expected_output = torch.load("checks/encoder_output").to(torch_device)
    print("Error is:", torch.sum(torch.abs(expected_output-actual_output)).iter
check_encoder()
```

```
Error is: 3.6512501537799835e-06 (error should be around 0)
```

(c) Patches

Transformers take a sequence of data. To send images, we would want to split images into a sequence of patches.

In [18]:

```
def patchify(images, patch_size=4):
    """Splitting images into patches.
    Args:
        images: Input tensor with size (batch, channels, height, width)
            We can assume that image is square where height == width.
    Returns:
        A batch of image patches with size (
          batch, (height / patch_size) * (width / patch_size),
        channels * patch_size * patch_size)
    Hint: use einops.rearrange. The "space-to-depth operation" example at https
    is not exactly what you need, but it gives a good idea of how to use rearra
    """
    return einops.rearrange(
        images,
        'b c (h p1) (w p2) -> b (h w) (c p1 p2)',
        p1=patch_size,
        p2=patch_size
    )


def unpatchify(patches, patch_size=4):
    """Combining patches into images.
    Args:
        patches: Input tensor with size (
        batch, (height / patch_size) * (width / patch_size),
        channels * patch_size * patch_size)
    Returns:
        A batch of images with size (batch, channels, height, width)

    Hint: einops.rearrange can be used here as well.
    """
    return einops.rearrange(
        patches,
        'b (h w) (c p1 p2) -> b c (h p1) (w p2)',
        p1=patch_size,
        p2=patch_size,
        h=int(patches.shape[1] ** 0.5),
        w=int(patches.shape[1] ** 0.5),
    )
```

(f) CNN_ViT

Here we implement the vision transformer. Follow the steps in the forward function for implementation.

In [37]:

```
class CNN_ViT(nn.Module):
    def __init__(self, n_classes, embedding_dim=256, patch_size=4, num_patches=
        super().__init__()
        self.patch_size = patch_size
        self.num_patches = num_patches
        self.embedding_dim = embedding_dim

        self.transformer = Encoder(embedding_dim, 1024, 4, 4)
        #self.transformer = Transformer(embedding_dim)
        self.cls_token = nn.Parameter(torch.randn(1, 1, embedding_dim) * 0.02)
        self.position_encoding = nn.Parameter(
            torch.randn(1, num_patches * num_patches + 1, embedding_dim) * 0.02
        )
        self.patch_projection = nn.Linear(patch_size * patch_size * 3, embeddin

        # A Layernorm and a Linear layer are applied on ViT encoder embeddings
        self.output_head = nn.Sequential(
            nn.LayerNorm(embedding_dim), nn.Linear(embedding_dim, n_classes)
        )

        self.cnn = CNN()

    def forward(self, images):
        """
        (1) Splitting images into fixed-size patches;
        (2) Linearly embed each image patch, prepend CLS token;
        (3) Add position embeddings;
        (4) Feed the resulting sequence of vectors to Transformer encoder.
        (5) Extract the embeddings corresponding to the CLS token.
        (6) Apply output head to the embeddings to obtain the logits
        """
        batch_size = images.shape[0]

        cnn_out = self.cnn(images)
        patch_out = patchify(cnn_out, patch_size=self.patch_size)  # shape = b,
        linear_out = self.patch_projection(patch_out)  # shape = b, h * w / p**
        cls_out = torch.cat((self.cls_token.expand(batch_size, -1, -1), linear_
                            dim=1)  # shape = b, h * w / p**2 + 1, hidden
        position_out = self.position_encoding + cls_out  # shape = b, h * w / p
        transformer_out = self.transformer(position_out)  # shape = b, h * w /
        head_out = self.output_head(transformer_out)  # shape = b, h * w / p**2
        head_cls_output = head_out[:, 0, :]  # shape = b, 1, n_classes

        return head_cls_output

    def foward_full(self, images):
        batch_size = images.shape[0]

        cnn_out = self.cnn(images)
        patch_out = patchify(cnn_out, patch_size=self.patch_size)  # shape = b,
        linear_out = self.patch_projection(patch_out)  # shape = b, h * w / p**
        cls_out = torch.cat((self.cls_token.expand(batch_size, -1, -1), linear_
                            dim=1)  # shape = b, h * w / p**2 + 1, hidden
        position_out = self.position_encoding + cls_out  # shape = b, h * w / p
        transformer_out, attn = self.transformer.forward_w_attn(position_out)
        head_out = transformer_out  # shape = b, h * w / p**2 + 1, hidden
        head_cls_output = head_out[:, 0, :]  # shape = b, 1, n_classes
```

```
        return head_cls_output, attn
```

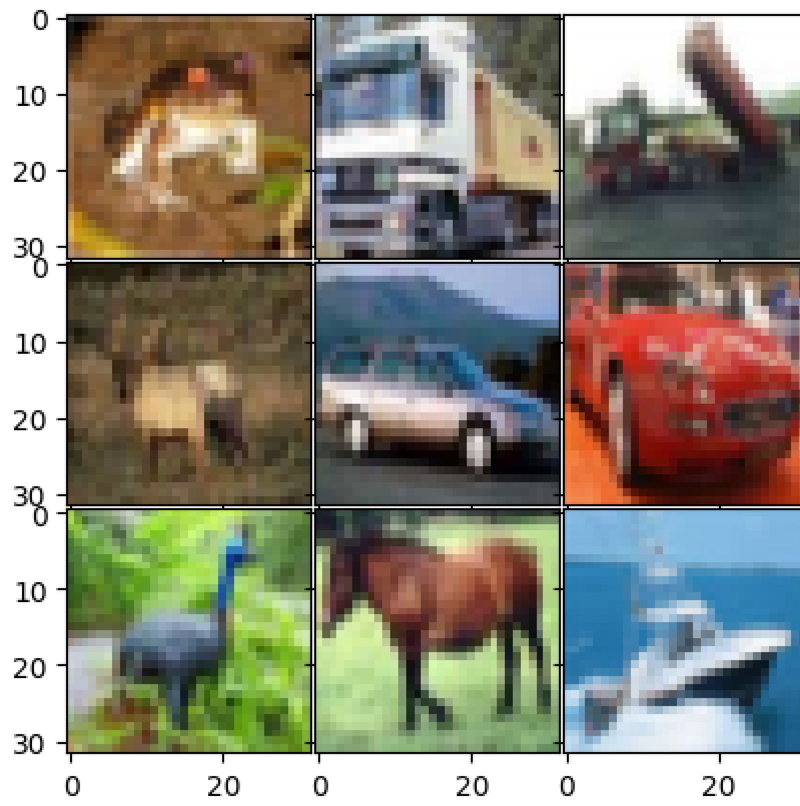
In [28]:
```
def check_cnn_vit():
    batch_size = 10
    test_image_size = (batch_size, 3, 32, 32)
    n_classes = 10
    patch_size = 4

    torch_device = 'cuda' if torch.cuda.is_available() else 'cpu'
    set_seed(182)

    ViT = CNN_ViT(n_classes=n_classes)
    ViT = ViT.to(torch_device)
    #torch.save(ViT.state_dict(),"checks/CNN_ViT")
    ViT.load_state_dict(torch.load("checks/CNN_ViT"))

    #inputs = torch.randn((test_image_size))
    #torch.save(inputs, "checks/CNN_ViT_input")
    inputs = torch.load("checks/CNN_ViT_input").to(torch_device)

    ViT.to(torch_device)
    actual_output = ViT(inputs)
    #torch.save(actual_output, "checks/CNN_ViT_output")

    expected_output = torch.load("checks/CNN_ViT_output").to(torch_device)
    print("Error is:", torch.sum(torch.abs(expected_output-actual_output)).iter
check_cnn_vit()
```

```
Error is: 1.9781291484832764e-05 (error should be around 0)
```

Train CNN_ViT

In [30]:
```
torch_device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(torch_device, torch.cuda.is_available())
seed = 182
set_seed(seed)

# ============== Visualize Dataset ===================

cifar10 = torchvision.datasets.CIFAR10(root='./data', train=True, download=True

fig = plt.figure()
grid = ImageGrid(fig, 111, nrows_ncols=(3, 3))

for ax, im in zip(grid, [cifar10[i][0] for i in range(9)]):
    ax.imshow(im)

# ======================== prepare training dataset ======================
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.Resize(32),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.Resize(32),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

batch_size = 128

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform_tra
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                       shuffle=False, num_workers=2)
```

```
cuda True
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz (https://
www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz) to ./data\cifar-10-python.ta
r.gz

HBox(children=(FloatProgress(value=0.0, max=170498071.0), HTML(value='')))


Extracting ./data\cifar-10-python.tar.gz to ./data
Files already downloaded and verified
Files already downloaded and verified
```


0
10
20
30
0
10
20
30
0
10
20
30
0
20
0
20
0
20

In [33]:

```
# ==================== Train ViT==================
# Initilize model (ClassificationViT)
model = CNN_ViT(10, embedding_dim=48)
# Move model to GPU
model.to(torch_device)
# Create optimizer for the model

# You may want to tune these hyperparameters to get better performance
optimizer = optim.AdamW(model.parameters(), lr=1e-3, betas=(0.9, 0.95), weight_

total_steps = 0
num_epochs = 10
train_logfreq = 100
losses = []
train_acc = []
all_val_acc = []
best_val_acc = 0

epoch_iterator = trange(num_epochs)
for epoch in epoch_iterator:
    # Train
    data_iterator = tqdm(trainloader)
    for x, y in data_iterator:
        total_steps += 1
        x, y = x.to(torch_device), y.to(torch_device)
        logits = model(x)
        loss = torch.mean(F.cross_entropy(logits, y))
        accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        data_iterator.set_postfix(loss=loss.item(), train_acc=accuracy.item())

        if total_steps % train_logfreq == 0:
            losses.append(loss.item())
            train_acc.append(accuracy.item())

    # Validation
    val_acc = []
    model.eval()
    for x, y in testloader:
        x, y = x.to(torch_device), y.to(torch_device)
        with torch.no_grad():
            logits = model(x)
        accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
        val_acc.append(accuracy.item())
    model.train()

    all_val_acc.append(np.mean(val_acc))
    # Save best model
    if np.mean(val_acc) > best_val_acc:
        best_val_acc = np.mean(val_acc)

    epoch_iterator.set_postfix(val_acc=np.mean(val_acc), best_val_acc=best_val_

plt.plot(losses)
```

```
plt.title('Train Loss')
plt.figure()
plt.plot(train_acc)
plt.title('Train Accuracy')
plt.figure()
plt.plot(all_val_acc)
plt.title('Val Accuracy')

torch.save(model.state_dict(), './model.pt')
```

```
10:47,  1.72s/it, loss=2.19, train_acc=0.203]
  4%|█                                                   | 14/391 [00:07<
10:47,  1.72s/it, loss=2.07, train_acc=0.258]
  5%|█                                                   | 18/391 [00:07<
07:31,  1.21s/it, loss=2.07, train_acc=0.258]
  5%|█                                                   | 18/391 [00:07<
07:31,  1.21s/it, loss=2.07, train_acc=0.219]
  5%|█                                                   | 18/391 [00:07<
07:31,  1.21s/it, loss=2.17, train_acc=0.164]
  5%|█                                                   | 18/391 [00:07<
07:31,  1.21s/it, loss=2.01, train_acc=0.305]
  5%|█                                                   | 18/391 [00:07<
07:31,  1.21s/it, loss=2.12, train_acc=0.141]
  6%|██                                                  | 22/391 [00:07<
05:15,  1.17it/s, loss=2.12, train_acc=0.141]
  6%|██                                                  | 22/391 [00:07<
05:15,  1.17it/s, loss=2.05, train_acc=0.234]
  6%|██                                                  | 22/391 [00:07<
05:15,  1.17it/s, loss=1.98, train_acc=0.266]
```

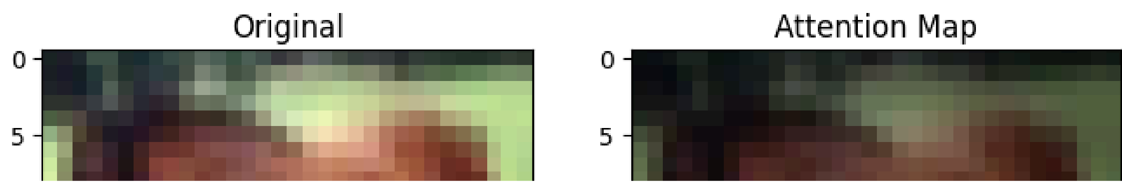
Best validation accuracy should be atleast 60%.

In [ ]:
```
def check_val(val):
    if val >= 0.6:
        print("Validation is above 60%")
    else:
        print(f"Validation is lower than expected. {val}")

check_val(best_val_acc)
```

(g) Visualizing Attention and Positional Embeddings

In [40]:
```
torch_device = 'cuda' if torch.cuda.is_available() else 'cpu'

model = CNN_ViT(10, embedding_dim=48)
model.load_state_dict(torch.load('./model.pt'))
model.eval()
model.to(torch_device)

transform_test = transforms.Compose([
    transforms.Resize(32),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
batch_size = 128
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

# ====================== Validation =======================
val_acc = []
for x, y in testloader:
    x, y = x.to(torch_device), y.to(torch_device)
    with torch.no_grad():
        logits = model(x)
    accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
    val_acc.append(accuracy.item())
print(f"test acc = {sum(val_acc) / len(val_acc)}")

# ================== Visualize CNN output ===================
cifar10 = torchvision.datasets.CIFAR10(root='./data', train=True, download=True
to_tensor = transforms.ToTensor()
to_img = transforms.ToPILImage()

imgs = [cifar10[i][0] for i in range(9)]
imgs_tensor = [to_tensor(img) for img in imgs]
imgs_cnn_out_tensor = [model.cnn(img.to(torch_device)) for img in imgs_tensor]
imgs_cnn_out = [to_img(img) for img in imgs_cnn_out_tensor]

fig = plt.figure()
grid = ImageGrid(fig, 111, nrows_ncols=(3, 3))

for ax, im in zip(grid, imgs_cnn_out):
    ax.imshow(im)

# ==================== visualize positional encoding =====================
patch_size = 4
pe = model.position_encoding[0, 1:, ]  # ignore batch_size and [CLS]
print(pe.shape)  # [64, 256] -> [patch_num**2, hidden_size]
pe = pe.reshape(patch_size, patch_size, -1)


def sim(pix_i, pix_j, patch_size):
    cos = torch.nn.CosineSimilarity(dim=0)
    similarity = []
    for i in range(patch_size):
        for j in range(patch_size):
            similarity.append(cos(pe[pix_i][pix_j], pe[i][j]).detach().cpu().nu
```

```
    return np.array(similarity).reshape(patch_size, patch_size)


fig = plt.figure()
grid = ImageGrid(fig, 111, nrows_ncols=(patch_size, patch_size))
for ax, im in zip(grid, [sim(i, j, patch_size) for i in range(patch_size) for j
    ax.imshow(im)

# ==================== visualize attention =====================
#Online discussion on Attention Map: https://www.kaggle.com/code/piantic/vision
imgs = [cifar10[i][0] for i in range(9)]
for img in imgs:
    att_mat = model.foward_full(to_tensor(img).unsqueeze(0).to(torch_device))[1
    att_mat = torch.mean(att_mat, dim=1)
    residual_att = torch.eye(att_mat.size(1)).to(torch_device)
    aug_att_mat = att_mat + residual_att
    aug_att_mat = aug_att_mat / aug_att_mat.sum(dim=-1).unsqueeze(-1)
    joint_attentions = torch.zeros(aug_att_mat.size()).to(torch_device)
    joint_attentions[0] = aug_att_mat[0]
    for n in range(1, aug_att_mat.size(0)):
        joint_attentions[n] = torch.matmul(aug_att_mat[n], joint_attentions[n -
    v = joint_attentions[-1]
    grid_size = int(np.sqrt(aug_att_mat.size(-1)))
    mask = v[0, 1:].reshape(grid_size, grid_size).detach().to("cpu").numpy()
    mask = cv2.resize(mask / mask.max(), imgs[0].size)[..., np.newaxis]
    result = (mask * img).astype("uint8")
    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 8))
    ax1.set_title('Original')
    ax2.set_title('Attention Map')
    _ = ax1.imshow(img)
    _ = ax2.imshow(result)
    for i in range(att_mat.size()[0]):
        v = aug_att_mat[i]
        grid_size = int(np.sqrt(aug_att_mat.size(-1)))
        mask = v[0, 1:].reshape(grid_size, grid_size).detach().to("cpu").numpy(
        mask = cv2.resize(mask / mask.max(), imgs[0].size)[..., np.newaxis]
        result = (mask * img).astype("uint8")
        fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 8))
        ax1.set_title('Original')
        ax2.set_title('Attention Map_%d Layer' % (i+1))
        _ = ax1.imshow(img)
        _ = ax2.imshow(result)
```



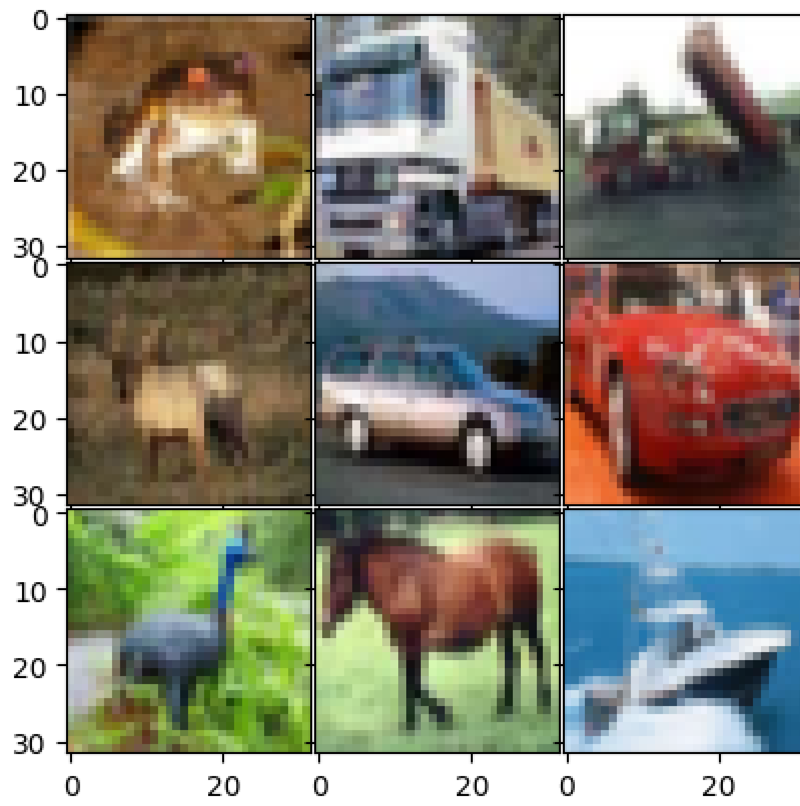
We have created a separate CNN that classifies images comparable to our current implementation of a ViT. Run the following cell to see how a CNN might learn and compare that to the visualization of attention from earlier.

In [41]:
```
class VisualizeCNN(nn.Module):
    """CNN used to visualize filters to compare to ViT
    Args:
    Returns:
        Logits of classfication
    """
    def __init__(self):
        super().__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(3, 32, 7, padding=3),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(64, 128, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Flatten(),
            nn.Dropout(p=0.5),
            nn.Linear(2048, 512),
            nn.ReLU(),
            nn.Linear(512, 128),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(128, 10),
        )

    def forward(self, x):
        return self.cnn(x)
```

In [42]:

```
seed = 182
set_seed(seed)

# ============== Visualize Dataset ===================

cifar10 = torchvision.datasets.CIFAR10(root='./data', train=True, download=True

fig = plt.figure()
grid = ImageGrid(fig, 111, nrows_ncols=(3, 3))

for ax, im in zip(grid, [cifar10[i][0] for i in range(9)]):
    ax.imshow(im)

# ======================== prepare training dataset ======================
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.Resize(32),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.Resize(32),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

batch_size = 128

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                    download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                      shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                   download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                     shuffle=False, num_workers=2)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```


0
10
20
30
0
10
20
30
0
10
20
30
0
20
0
20
0
20

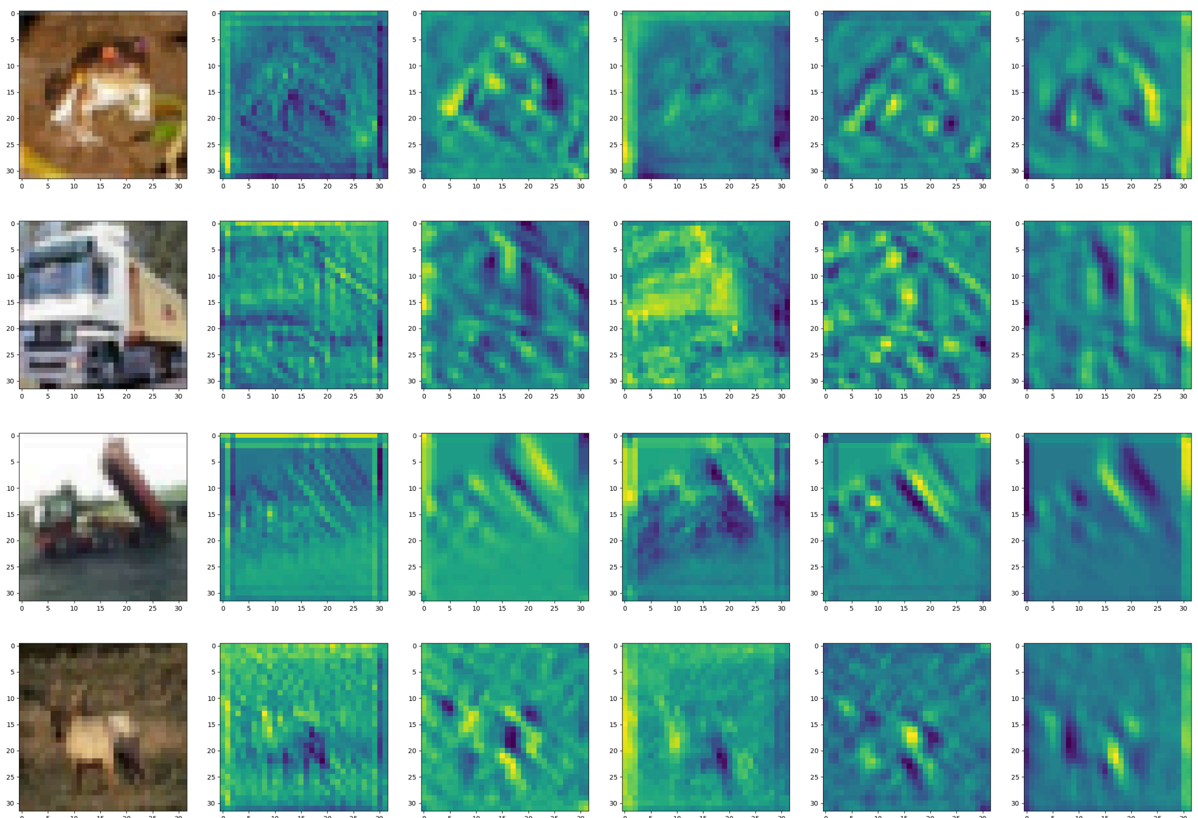
In [43]:
```
# ===================== Train CNN===================
# Initilize model (VisualizeCNN)
model = VisualizeCNN()
# Move model to GPU
model.to(torch_device)
# Create optimizer for the model

# You may want to tune these hyperparameters to get better performance
optimizer = optim.AdamW(model.parameters(), lr=1e-3, betas=(0.9, 0.95), weight_

total_steps = 0
num_epochs = 10
train_logfreq = 100
losses = []
train_acc = []
all_val_acc = []
best_val_acc = 0

epoch_iterator = trange(num_epochs)
for epoch in epoch_iterator:
    # Train
    data_iterator = tqdm(trainloader)
    for x, y in data_iterator:
        total_steps += 1
        x, y = x.to(torch_device), y.to(torch_device)
        logits = model(x)
        loss = torch.mean(F.cross_entropy(logits, y))
        accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        data_iterator.set_postfix(loss=loss.item(), train_acc=accuracy.item())

        if total_steps % train_logfreq == 0:
            losses.append(loss.item())
            train_acc.append(accuracy.item())

    # Validation
    val_acc = []
    model.eval()
    for x, y in testloader:
        x, y = x.to(torch_device), y.to(torch_device)
        with torch.no_grad():
            logits = model(x)
        accuracy = torch.mean((torch.argmax(logits, dim=-1) == y).float())
        val_acc.append(accuracy.item())
    model.train()

    all_val_acc.append(np.mean(val_acc))
    # Save best model
    if np.mean(val_acc) > best_val_acc:
        best_val_acc = np.mean(val_acc)

    epoch_iterator.set_postfix(val_acc=np.mean(val_acc), best_val_acc=best_val_

plt.plot(losses)
```

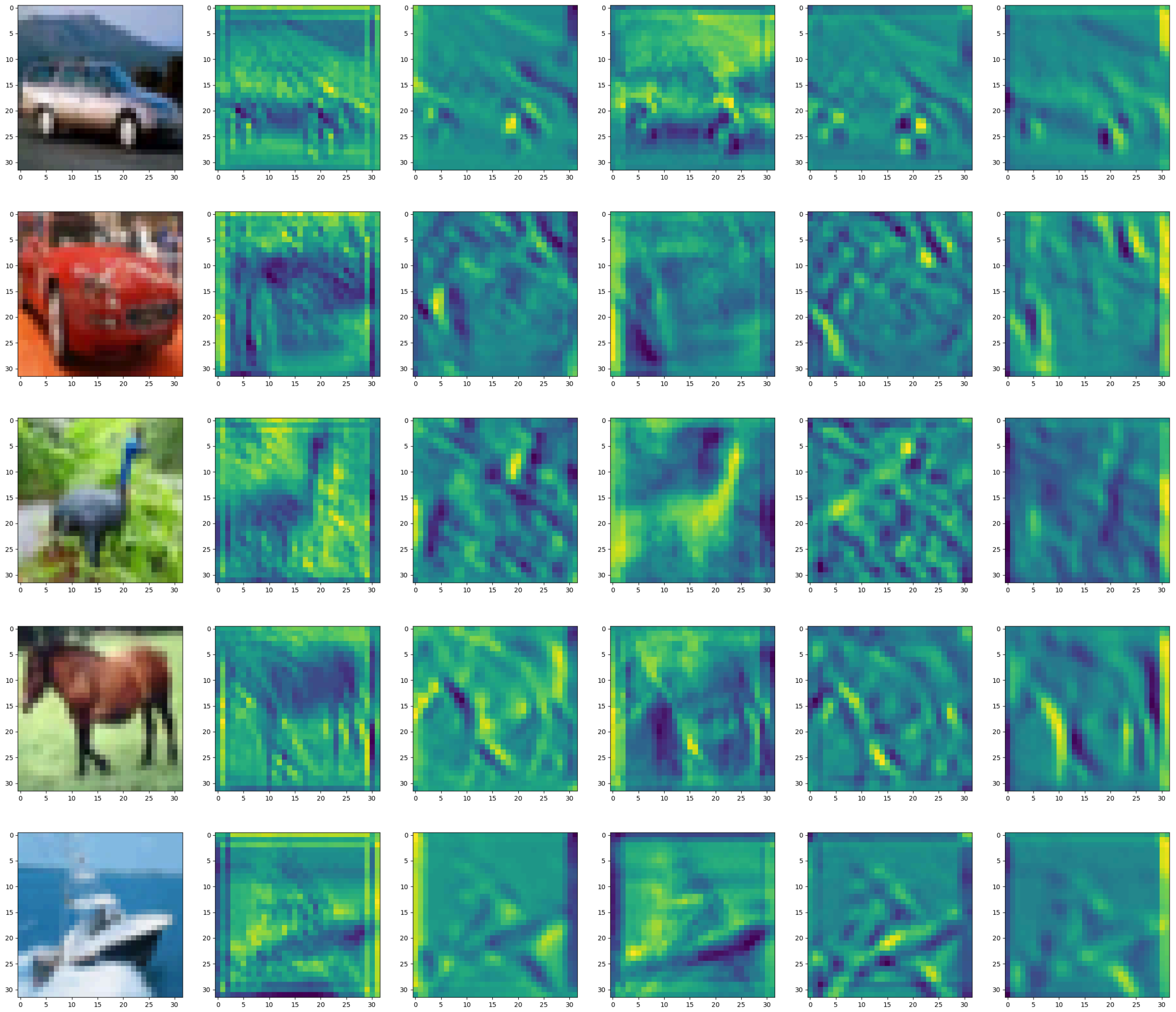
```
plt.title('Train Loss')
plt.figure()
plt.plot(train_acc)
plt.title('Train Accuracy')
plt.figure()
plt.plot(all_val_acc)
plt.title('Val Accuracy')

torch.save(model.state_dict(), './model_cnn.pt')
```

```
10:41,  1.71s/it, loss=1.15, train_acc=0.602]
  4%|█▍                                         | 17/391 [00:07<
10:41,  1.71s/it, loss=1.08, train_acc=0.578]
  6%|██                                         | 22/391 [00:07<
07:25,  1.21s/it, loss=1.08, train_acc=0.578]
  6%|██                                        | 22/391 [00:07<0
7:25,  1.21s/it, loss=0.991, train_acc=0.641]
  6%|█▊                                          | 22/391 [00:07
<07:25,  1.21s/it, loss=1.2, train_acc=0.57]
  6%|██                                         | 22/391 [00:07<
07:25,  1.21s/it, loss=1.01, train_acc=0.617]
  6%|██                                        | 22/391 [00:07<0
7:25,  1.21s/it, loss=0.905, train_acc=0.719]
  7%|██                                        | 26/391 [00:07<0
5:11,  1.17it/s, loss=0.905, train_acc=0.719]
  7%|██                                        | 26/391 [00:07<0
5:11,  1.17it/s, loss=1.19, train_acc=0.562]
  7%|██                                        | 26/391 [00:07<0
5:11,  1.17it/s, loss=1.06, train_acc=0.664]
  7%|██▍                                          | 26/391 [00:
```

In [46]:
```
cifar10 = torchvision.datasets.CIFAR10(root='./data', train=True, download=True
to_tensor = transforms.ToTensor()
to_img = transforms.ToPILImage()

imgs = [cifar10[i][0] for i in range(9)]
imgs_tensor = [to_tensor(img) for img in imgs]


cnn_model = VisualizeCNN()
cnn_model.load_state_dict(torch.load('./model_cnn.pt'))
cnn_model.eval()
cnn_model.to(torch_device)
return_nodes = {
    "cnn.0": "cnn0",
    "cnn.3": "cnn1",
    "cnn.6": "cnn2"
}
train_nodes, eval_nodes = get_graph_node_names(cnn_model)
model2 = create_feature_extractor(cnn_model, return_nodes=return_nodes)
intermediate_outputs = [model2(img.unsqueeze(0).to(torch_device)) for img in ir

for k, into in enumerate(intermediate_outputs):
    j = into["cnn0"]
    fig, ax = plt.subplots(ncols=6, figsize=(32, 32))
    ax[0].imshow(to_img(imgs_tensor[k]))
    for i, axn in enumerate(ax):
        if i == 0:
            continue
        _ = axn.imshow(j.squeeze()[i - 1].detach().to('cpu').numpy())
```

```
Files already downloaded and verified
```





In [ ]: