First Presentation

ELVIS CABRERA
Main Topics

1. Personal Background
2. Assignment 1
3. Research Project
4. Preliminary study
5. Plan for this week
1. Personal Background

. Born in Cuba

. Emigrated to the United Stated in June 2023

. Currently pursuing an Associate’s degree in Mathematics

. Math enthusiast
2. Assignment 1

- Dataset: Cifar100
- Loss Function: Categorical Cross-entropy
- Optimizer: Adam
- Batch size: 128
- Number of Epochs: 100
2.1. Reduced VGG16 architecture

```python
model = keras.Sequential(

    # Block 1
    layers.Conv2D(32, (3,3), activation='relu', padding='same', input_shape=x_train.shape[1:]),
    layers.Conv2D(32, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D(pool_size=(2,2), strides=(2,2)),

    # Block 2
    layers.Conv2D(64, (3,3), activation='relu', padding='same'),
    layers.Conv2D(64, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D(pool_size=(2,2), strides=(2,2)),

    # Block 3
    layers.Conv2D(128, (3,3), activation='relu', padding='same'),
    layers.Conv2D(128, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D(pool_size=(2,2), strides=(2,2)),

    # Block 4
    layers.Conv2D(256, (3,3), activation='relu', padding='same'),
    layers.Conv2D(256, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D(pool_size=(2,2), strides=(2,2)),

    # Fully Connected Layers
    layers.Flatten(),
    layers.Dense(2048, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(num_classes, activation='softmax'),
)
```
2.2. Modified VGG16 implemented in Keras

```python
model = keras.Sequential(

    # Block 1
    layers.Conv2D(32, (3,3), activation='relu', padding='same', input_shape=x_train.shape[1:]),
    layers.BatchNormalization(),
    layers.Conv2D(32, (3,3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2,2)),
    layers.Dropout(0.4),

    # Block 2
    layers.Conv2D(64, (3,3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.Conv2D(64, (3,3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2,2)),
    layers.Dropout(0.4),

    # Block 3
    layers.Conv2D(128, (3,3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.Conv2D(128, (3,3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2,2)),
    layers.Dropout(0.5),
)
```
2.2. Modified VGG16 implemented in Keras

```python
# Block 4
layers.Conv2D(256, (3,3), activation='relu', padding='same'),
layers.BatchNormalization(),
layers.Conv2D(256, (3,3), activation='relu', padding='same', kernel_regularizer=regularizers.l1_l2(l1=0.01, l2=0.01)),
layers.BatchNormalization(),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Dropout(0.5),

# Fully connected layers
layers.Flatten(),
layers.Dense(2048, activation='relu'),
    layers.BatchNormalization(),
layers.Dense(512, activation='relu'),
layers.BatchNormalization(),
layers.Dropout(0.5),
layers.Dense(num_classes, activation='softmax'),
]
```

model.summary()
2.4. Results: VGG16 vs modified VGG16
2.3. Implementation in PyTorch

class VGG16_NET(nn.Module):
    def __init__(self):
        super(VGG16_NET, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)

        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)

        self.conv5 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.conv6 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)

        self.conv7 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
        self.conv8 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(256)

        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(50176, 4096)
        self.bn5 = nn.BatchNorm1d(4096)
        self.fc2 = nn.Linear(4096, 512)
        self.bn6 = nn.BatchNorm1d(512)
        self.fc3 = nn.Linear(512, 100)
2.3. Implementation in PyTorch

def forward(self, x):
    x = F.relu(self.bn1(self.conv1(x)))
    x = F.relu(self.bn1(self.conv2(x)))
    x = self.maxpool1(x)
    x = F.dropout(x, 0.4)
    x = F.relu(self.bn2(self.conv3(x)))
    x = F.relu(self.bn2(self.conv4(x)))
    x = self.maxpool1(x)
    x = F.dropout(x, 0.4)
    x = F.relu(self.bn3(self.conv5(x)))
    x = F.relu(self.bn3(self.conv6(x)))
    x = self.maxpool1(x)
    x = F.dropout(x, 0.5)
    x = F.relu(self.bn4(self.conv7(x)))
    x = F.relu(self.bn4(self.conv8(x)))
    x = self.maxpool1(x)
    x = F.dropout(x, 0.5)
    x = x.reshape(x.shape[0], -1)
    x = F.relu(self.bn5(self.fc1(x)))
    x = F.relu(self.bn6(self.fc2(x)))
    x = F.dropout(x, 0.5)
    x = self.fc3(x)
    return x
2.3. Some Results
3. Research Project

We are interested in fine-tuning CLIP for multi-label classification purposes on datasets with noisy and partial labels.

Concepts.
- Fine-Tuning
- Noisy and Partial labels
- Multi-label Classification
- CLIP
3.1. CLIP (Contrastive Language-Image Pretraining)
3.2. Why CLIP?

HIGH ZERO-SHOT ACCURACIES. OUTPERFORMS RESNET50 IN 16 OUT OF 27 DATABASES.
5. My Study

5.1. Transformers.

```python
class EncoderLayer(nn.Module):
    def __init__(self, hidden_size, filter_size, dropout_rate):
        super(EncoderLayer, self).__init__()
        self.self_attention_norm = nn.LayerNorm(hidden_size, eps=1e-6)  # Layer normalization
        self.self_attention = MultiHeadAttention(hidden_size, dropout_rate)  # Multi-head self-attention
        self.self_attention_dropout = nn.Dropout(dropout_rate)  # Dropout for self-attention

        self.ffn_norm = nn.LayerNorm(hidden_size, eps=1e-6)  # Layer normalization for FFN
        self.ffn = FeedForwardNetwork(hidden_size, filter_size, dropout_rate)  # Feed-forward network
        self.ffn_dropout = nn.Dropout(dropout_rate)  # Dropout for FFN

    def forward(self, x, mask):
        # pylintr: disable:arguments-differ
        y = self.self_attention_norm(x)  # Apply layer normalization
        y = self.self_attention(y, y, y, mask)  # Apply self-attention
        y = self.self_attention_dropout(y)  # Apply dropout
        x = x + y  # Add residual connection

        y = self.ffn_norm(x)  # Apply layer normalization
        y = self.ffn(y)  # Apply feed-forward network
        y = self.ffn_dropout(y)  # Apply dropout
        x = x + y  # Add residual connection

        return x  # Return the output
```
5.2. Common techniques to use CLIP for multi-label classification

5.3. Pseudocode for CLIP from original paper.

```python
# image_encoder - ResNet or Vision Transformer
# text_encoder - CBOW or Text Transformer
# I[n, h, w, c] - minibatch of aligned images
# T[n, l] - minibatch of aligned texts
# W_i[d_i, d_e] - learned proj of image to embed
# W_t[d_t, d_e] - learned proj of text to embed
# t - learned temperature parameter

# extract feature representations of each modality
I_f = image_encoder(I) # [n, d_i]
T_f = text_encoder(T) # [n, d_t]

# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1)

# scaled pairwise cosine similarities [n, n]
logits = np.dot(I_e, T_e.T) * np.exp(t)

# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss = (loss_i + loss_t)/2
```
6. Plan for this week

6.1. Understand Code for CLIP (meeting tomorrow)

6.2. Duplicate some results from CLIP’s original paper.

6.3. Read 3 papers about multilabel classification

https://proceedings.neurips.cc/paper_files/paper/2022/file/c5169260ef32d1bd3597c14d8c89b034-Paper-Conference.pdf