PATTERN RECOGNITION ASSIGNMENT 4: VISUAL QUESTION ANSWERING

ANAS EMAD 7048 MOHAB AYMAN 7127 MAZEN GHANEM 6896

read datasets into pandas dataframe

4]:
```
df_train = pd.read_json('/kaggle/input/vizwiz/Annotations/Annotations/train.json')
df_val = pd.read_json('/kaggle/input/vizwiz/Annotations/Annotations/val.json')
df_train.head()
```

4]:

	image	question	answers	answer_type	answerable
0	VizWiz_train_00000000.jpg	What's the name of this product?	[{'answer_confidence': 'yes', 'answer': 'basil...	other	1
1	VizWiz_train_00000001.jpg	Can you tell me what is in this can please?	[{'answer_confidence': 'yes', 'answer': 'soda'...	other	1
2	VizWiz_train_00000002.jpg	Is this enchilada sauce or is this tomatoes? ...	[{'answer_confidence': 'yes', 'answer': 'these...	other	1
3	VizWiz_train_00000003.jpg	What is the captcha on this screenshot?	[{'answer_confidence': 'yes', 'answer': 't36m'...	other	1
4	VizWiz_train_00000004.jpg	What is this item?	[{'answer_confidence': 'yes', 'answer': 'solar...	other	1

read JSON files into pandas DataFrames and display the first few rows of the training data.

split the train data to test and train

[5]:
```
train_df, test_df = train_test_split(df_train,test_size=0.05, random_state = 42 ,stratify=df_train['answer_type'])
```

train_test_split function from the scikit-learn library is used to split the df_train DataFrame into training and testing subsets were testing gets 0.05 of the data function to get answer for every question from the 10 answers

+ Code + Markdown

6]:

```
def vocab_load(ans_list):
    ans_txt=[]
    for ans in ans_list:
        ans_txt.append(ans['answer'])
        #         print(ans['answer'])
    word_count=Counter(ans_txt)
    max_count = max(word_count.values())
    max_occ = [element for element, count in word_count.items() if count == max_count]
    word_org=max_occ


    if len(max_occ)>1:
        word=''
        least=np.inf

        for k in max_occ:
            total_dst=0
            for words in ans_txt:
                total_dst=nltk.edit_distance(k,words) + total_dst
                if(total_dst<least):
                    least=total_dst
                    word=k
        word_org=[word]

    return word_org[0]
```

vocab_load function calculates the most representative answer text from a given list of answers. It first identifies the answer text(s) with the highest frequency and, if there are multiple such answers, selects the one that has the smallest total edit distance using **levenshtein** to all other answer texts. The function returns the most representative answer text as the output.

generate answer for every question and the dictionary of unique values

+ Code + Markdown

]:

```
df_train_ans = train_df['answers']

max_occ=[]
outp_feature=torch.empty((1,1024))

for i in (df_train_ans.index) :
    max_occ_val=vocab_load(df_train_ans[i])
    max_occ.append(max_occ_val)
    img_feature,question_feature=clip_encoder(0,i)
    con_feature=torch.cat((img_feature,question_feature), 1)
    outp_feature=torch.cat((outp_feature, con_feature), 0)
print(outp_feature.shape)
#     print(question_feature.shape)
print(len(max_occ))
max_occ=np.unique(max_occ)
print(len(max_occ))
outp_feature=outp_feature[1:]
print(outp_feature.shape)
```

this code iterates over the 'answers' column of the train_df DataFrame, calculates the most representative answer text for each entry using the vocab_load function, extracts image and question features using the clip_encoder function, concatenates these features, and stores them in the outp_feature tensor. It also keeps track of the most representative answer texts in the max_occ list. Finally, it prints the shape of the outp_feature tensor and the length of the max_occ list after some operations.

save the classes dictionary

[ ]:
```
torch.save(max_occ, 'class_occ.pt')
print(max_occ)
# Load the tensor
# loaded_tensor = torch.load('class_occ.pt')

# print(loaded_tensor)
```

save train features in tensor file

+ Code + Markdown

[ ]:
```
# Create a PyTorch tensor
my_tensor_train = outp_feature

# Save the tensor
torch.save(my_tensor_train, 'train_feat.pt')
print(my_tensor_train)
# Load the tensor
# loaded_tensor = torch.load('train_feat.pt')

print(loaded_tensor)
```

save the train indexes in tensor

[ ]:
```
train_idx = train_df.index
torch.save(train_idx, 'train_idx.pt')
print(train_idx)
# Load the tensor
# loaded_tensor = torch.load('train_idx.pt')
# print(loaded_tensor)
```

+ Code + Markdown

save test indexes in tensor

[ ]:
```
test_idx = test_df.index
torch.save(test_idx, 'test_indx.pt')
print(test_idx)
# Load the tensor
# loaded_tensor = torch.load('test_indx.pt')
# print(loaded_tensor)
```

save val indexes in torch tensor

]:
```
val_indx=df_val.index
torch.save(val_indx, 'val_indx.pt')
print(val_indx)
# Load the tensor
# loaded_tensor = torch.load('val_indx.pt')
# print(loaded_tensor)
```

+ Code + Markdown

We save the train ,test,validation features in torch serialized files to avoid having to extract features each time

load the saved indexes for test and train and val

+ Code + Markdown

]:

```python
train_idx = torch.load('/kaggle/input/saved-data/train_idx.pt')

test_indx = torch.load('/kaggle/input/saved-data/test_indx.pt')

val_indx = torch.load('/kaggle/input/saved-data/val_indx.pt')

print(train_idx)
print(test_indx)
print(val_indx)
```

```
Int64Index([14709,  9566, 11322,  5267,  9281, 18827,  5396, 11843,  2593,
             8153,
            ...
            13960,  2546,  2367, 14451,  1811, 15453, 19967, 17226, 16495,
             8401],
           dtype='int64', length=19496)
Int64Index([20148,  1384, 15850, 10122, 16891,  9551,  4248, 19739, 12851,
             9060,
            ...
            17283,  1797,  6696,  1489, 12106, 10922, 17572, 16464,  1537,
             5393],
           dtype='int64', length=1027)
RangeIndex(start=0, stop=4319, step=1)
```

load the saved values from the files

answer for every question in train

8]:

```python
df_train_ans = df_train['answers']
df_train_ans = df_train_ans.iloc[train_idx]###use the pre saved indices
ans_train=[]

for i in (df_train_ans.index) :
    ans_train.append(vocab_load(df_train_ans[i]))
print(len(ans_train))
```

```
19496
```

answer for every question in validation

9]:

```python
df_val_ans = df_val['answers']

df_val_ans = df_val_ans.iloc[val_indx]###use the pre saved indices

ans_val=[]
for i in (df_val_ans.index) :
    ans_val.append(vocab_load(df_val_ans[i]))
print(len(ans_val))
```

```
4319
```

We load the data from the rows of the saved indices

Features for validation data

+ Code + Markdown

[98]:
```
outp_feature_val=torch.empty((1,1024))

for i in (df_val.index):
    img_feature,question_feature=clip_encoder(1,i)

    con_feature_val=torch.cat((img_feature,question_feature), 1)

    outp_feature_val=torch.cat((outp_feature_val, con_feature_val), 0)

print(outp_feature_val.shape)

outp_feature_val=outp_feature_val[1:]

print(outp_feature_val.shape)


# Create a PyTorch tensor
my_tensor_val = outp_feature_val
# Save the tensor
torch.save(my_tensor_val, 'val_feat.pt')
print(my_tensor_val)

# Load the tensor
# loaded_tensor = torch.load('val_feat.pt')

# print(loaded_tensor)
```

his code extracts image and question features using the clip_encoder function for each index in df_val. It concatenates these features into a tensor called outp_feature_val, removes the initial empty row, saves the tensor as a serialized file, and prints the tensor and its shape.

Features for test

[ ]:
```
outp_feature_tst=torch.empty((1,1024))

for i in (test_df.index):
    img_feature,question_feature=clip_encoder(0,i)

    con_feature_tst=torch.cat((img_feature,question_feature), 1)

    outp_feature_tst=torch.cat((outp_feature_tst, con_feature_tst), 0)

print(outp_feature_tst.shape)

outp_feature_tst=outp_feature_tst[1:]

print(outp_feature_tst.shape)


# Create a PyTorch tensor
my_tensor_test = outp_feature_tst
# Save the tensor
torch.save(my_tensor_test, 'test_feat.pt')
print(my_tensor_test)

# Load the tensor
loaded_tensor = torch.load('test_feat.pt')

print(loaded_tensor)
```

```
import clip

clip.available_models()
```

```
['RN50',
 'RN101',
 'RN50x4',
 'RN50x16',
 'RN50x64',
 'ViT-B/32',
 'ViT-B/16',
 'ViT-L/14',
 'ViT-L/14@336px']
```

\+ Code \+ Markdown

```
# device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
model_clip, preprocess = clip.load("ViT-B/32")
input_resolution = model_clip.visual.input_resolution
context_length = model_clip.context_length
vocab_size = model_clip.vocab_size

print("Model parameters:", f"{np.sum([int(np.prod(p.shape)) for p in model_clip.parameters()]):,}")
print("Input resolution:", input_resolution)
print("Context length:", context_length)
print("Vocab size:", vocab_size)
```

```
Model parameters: 151,277,313
Input resolution: 224
Context length: 77
Vocab size: 49408
```

Choosing the model

```
[30]:
from PIL import Image

def clip_encoder(value,index):
#     print(index)
    if value==0:
        data_row=df_train.iloc[index]
        question=data_row["question"]
        img_path="/kaggle/input/vizwiz/train/train/"+str(data_row["image"])
    elif value==1:
        data_row=df_val.iloc[index]
        question=data_row["question"]
        img_path="/kaggle/input/vizwiz/val/val/"+str(data_row["image"])
    elif value==3:
        data_row=test_bonus.iloc[index]
        question=data_row["question"]
        img_path="/kaggle/input/vizwiz/test/test/"+str(data_row["image"])
#     print(question)
#     print(img_path)


    with open(img_path,"rb") as f:
            img = Image.open(f)
            img = img.resize((224, 224))
            img = np.array(img)
            img = torch.from_numpy(img)
            img = img.permute(2, 0, 1)
            img = img.unsqueeze(dim=0)

    question = clip.tokenize(question, truncate=True)
    question = question.squeeze()
    # print(f"Before encoding {img.shape}")
    with torch.no_grad():
        img_feature = model_clip.encode_image(img)
        question_feature = model_clip.encode_text(question.unsqueeze(dim=0))
    return img_feature,question_feature
```

This function takes an image and a corresponding question, preprocesses the image, tokenizes the question, and encodes both the image and question using the CLIP model to obtain their respective features.

load saved tensor files

+ Code + Markdown

```
train_ds = torch.load('/kaggle/input/saved-data/train_feat.pt')
print(train_ds)


test_ds = torch.load('/kaggle/input/saved-data/test_feat.pt')
print(test_ds)


val_ds = torch.load('/kaggle/input/saved-data/val_feat.pt')
print(val_ds)

max_occ = torch.load('/kaggle/input/class-occurence/class_occ.pt')
print(max_occ)

print(train_ds.shape)
print(test_ds.shape)
print(val_ds.shape)
print(max_occ.shape)
```

```
tensor([[ 1.6236e-02, -2.0383e-01, -8.5998e-02, ..., -2.0287e-01,
```

load saved tensor files

```
train_ds = torch.load('/kaggle/input/saved-data/train_feat.pt')
print(train_ds)


test_ds = torch.load('/kaggle/input/saved-data/test_feat.pt')
print(test_ds)


val_ds = torch.load('/kaggle/input/saved-data/val_feat.pt')
print(val_ds)

max_occ = torch.load('/kaggle/input/class-occurence/class_occ.pt')
print(max_occ)

print(train_ds.shape)
print(test_ds.shape)
print(val_ds.shape)
print(max_occ.shape)
```


label mapping

```
labels_map = {"other":0,
             "unanswerable":1,
              "yes/no":2,
              "number":3}
train_ans_type=torch.tensor([ labels_map[index] for index in train_df['answer_type'].tolist()])
val_ans_type=torch.tensor([ labels_map[index] for index in df_val['answer_type'].tolist()])
test_ans_type=torch.tensor([ labels_map[index] for index in test_df['answer_type'].tolist()])

print(train_ans_type)
print(val_ans_type)
print(test_ans_type)
```

onehot for dictionary

17]:

```
onehot_encoder=OneHotEncoder(sparse_output=False,handle_unknown="ignore")


labels_2d = [[label] for label in ans_train]
onehot_encoder.fit(labels_2d)
y_train_encoded= onehot_encoder.transform(labels_2d)
y_train_encoded=torch.tensor(y_train_encoded)

labels_2d = [[label] for label in ans_val]
y_val_encoded= onehot_encoder.transform(labels_2d)
y_val_encoded=torch.tensor(y_val_encoded)

labels_2d = [[label] for label in ans_test]
y_test_encoded=onehot_encoder.transform(labels_2d)
y_test_encoded=torch.tensor(y_test_encoded)



print(y_train_encoded.shape)
print(y_val_encoded.shape)
print(y_test_encoded.shape)
```

```
torch.Size([19496, 5458])
```

The code snippet performs one-hot encoding on the target labels using scikit-learn's OneHotEncoder class. It prepares the label data by creating 2D lists for the training, validation, and test sets. Then, it fits the OneHotEncoder on the training labels and transforms them into a one-hot encoded representation. The encoded labels are stored in tensors: y_train_encoded, y_val_encoded, and y_test_encoded. Finally, it prints the shapes of the encoded label tensors, providing information about the dimensions of the encoded label representations for each dataset.

convert answerability to list

+ Code + Markdown

]:

```
y_train_answerability=torch.tensor(train_df['answerable'].tolist())
y_test_answerability=torch.tensor(test_df['answerable'].tolist())
y_valid_answerability=torch.tensor(df_val['answerable'].tolist())

print(y_train_answerability.shape)
print(y_test_answerability.shape)
print(y_valid_answerability.shape)
```

```
torch.Size([19496])
torch.Size([1027])
torch.Size([4319])
```

]:

```
train_losses = []
train_accuracies = []
train_answerabilities = []

val_losses = []
val_accuracies = []
val_answerabilities = []
```

Training Model:

```
class LightningModel(pl.LightningModule):
    #
    def __init__(self,max_occ, weight_decay):
        super().__init__()
        self.vocab=max_occ
        self.lr = 0.0007585775750291836
        self.weight_decay = weight_decay

        self.ln1 = nn.LayerNorm(512*2)
        self.dp1 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(512*2, 512)

        # Answer branch
        self.ln2= nn.LayerNorm(512)
        self.dp2 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512,len(max_occ))

        # Answer type branch
        self.fc_aux = nn.Linear(512,4)
        self.fc_gate = nn.Linear(4,len(max_occ))
        self.act_gate = nn.Sigmoid()

        self.fc_answerable=torch.nn.Linear(1024,2)
        self.sigmoid= torch.nn.Sigmoid()
        self.softmax= torch.nn.Softmax(dim=1)

        self.train_acc_history =0.0
```

```python
def forward(self, xc):
    x=self.ln1(xc)
    x=self.dp1(x)
    x=self.fc1(x)

    aux=self.fc_aux(x)
    gate=self.fc_gate(aux)
    gate=self.act_gate(gate)

    x=self.ln2(x)
    x=self.dp2(x)
    vqa=self.fc2(x)

    output=vqa * gate

    answerable = self.fc_answerable(xc)
    answerable = self.sigmoid(answerable)

    answerable = self.softmax(answerable)
    return output,aux,answerable

def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=self.lr, weight_decay=self.weight_decay)
    return optimizer

def calculate_accuracy_answer_types(self,y_decoded,y_true):
    batch_size,vocab_size=y_decoded.size()
    num_correct=0
    for i in range(batch_size):
        x=torch.argmax(y_decoded[i])
        if x == y_true[i]:
            num_correct+=1
    accuracy=num_correct/batch_size
    return accuracy
```

```python
def calculate_accuracy_answers(self,y_decoded,y_true):
    batch_size,vocab_size=y_decoded.size()
    num_correct=0
    for i in range(batch_size):

        if torch.argmax(y_decoded[i])== torch.argmax(y_true[i]):
            num_correct+=1
    accuracy=num_correct/batch_size
    return accuracy
```

```python
def training_step(self, batch, batch_idx):
    # training_step defines the train loop. It is independent of forward
    x, y_t,y_a,y_ans = batch
    y_hat,y_aux,y_answerable=self(x)# prediction
    predicted_answerability = torch.argmax(y_answerable,dim=1)


    #loss calc
    loss=nn.CrossEntropyLoss()(y_hat,y_a)
    aux_loss=nn.CrossEntropyLoss()(y_aux,y_t)
    ans_loss=nn.CrossEntropyLoss()(y_answerable,y_ans)
    total_loss=loss + aux_loss + ans_loss

    #Training Accuracy
    predicted_answer=torch.argmax(y_hat,dim=1)
    actual_answer=torch.argmax(y_a,dim=1)
    training_accuracy=0.0
    for i in range(y_a.shape[0]):
        if actual_answer[i]== predicted_answer[i]:
            training_accuracy+=1
    training_accuracy /= y_a.shape[0]


    #Answerability
    actual_answerability= y_ans
    answerability=0.0
    for i in range(y_ans.shape[0]):
        if predicted_answerability[i]== actual_answerability[i]:
            answerability+=1
    answerability /= y_ans.shape[0]

    #Logs
    self.log("train_loss",total_loss,prog_bar=True,on_step=False,on_epoch=True)
    self.log("train_acc",training_accuracy,prog_bar=True,on_step=False,on_epoch=True)
    self.log("train_answerability",answerability,prog_bar=True,on_step=False,on_epoch=True)

    return total_loss
```

```python
def validation_step(self, batch, batch_idx):
    x, y_t,y_a,y_ans = batch
    y_hat,y_aux,y_answerable=self(x)# prediction
    predicted_answerability = torch.argmax(y_answerable,dim=1)

    #loss calc
    loss=nn.CrossEntropyLoss()(y_hat,y_a)
    aux_loss=nn.CrossEntropyLoss()(y_aux,y_t)
    ans_loss=nn.CrossEntropyLoss()(y_answerable,y_ans)
    total_loss=loss + aux_loss + ans_loss

    #Training Accuracy
    predicted_answer=torch.argmax(y_hat,dim=1)
    actual_answer=torch.argmax(y_a,dim=1)
    validation_accuracy=0.0
    for i in range(y_a.shape[0]):
        if actual_answer[i]== predicted_answer[i]:
            validation_accuracy+=1
    validation_accuracy /= y_a.shape[0]

    #Answerability
    actual_answerability= y_ans
    answerability=0.0
    for i in range(y_ans.shape[0]):
        if predicted_answerability[i]== actual_answerability[i]:
            answerability+=1
    answerability /= y_ans.shape[0]

    #Logs
    self.log("val_loss",total_loss,prog_bar=True,on_step=False,on_epoch=True)
    self.log("val_acc",validation_accuracy,prog_bar=True,on_step=False,on_epoch=True)
    self.log("val_answerability",answerability,prog_bar=True,on_step=False,on_epoch=True)
    return total_loss

def predict_step(self,batch,batch_idx):
    x,y_t,y_a,y_ans=batch
    y_hat,y_aux,y_answerable=self(x)
    return y_hat,y_aux,y_answerable
```

The model architecture consists of several layers and components, such as linear layers, normalization layers, dropout layers, and activation functions. It includes branches for predicting answers, answer types, and answerability.

The **forward method** defines the forward pass of the model, where input tensors are passed through the defined layers to obtain output predictions for answers, answer types, and answerability.

The **configure_optimizers method** configures the optimizer for training the model. In this case, it uses the Adam optimizer with a specified learning rate.

The **training_step method** defines the training loop for a single batch of data. It computes the loss, performs forward propagation, calculates accuracy metrics, and logs the training loss, accuracy, and answerability.

The **validation_step** method defines the validation loop for a single batch of data. It computes the loss, performs forward propagation, calculates accuracy metrics, and logs the validation loss, accuracy, and answerability.

The **predict_step** method performs inference on a single batch of data and returns the predicted outputs for answers, answer types, and answerability.

The on_train_epoch_end and on_validation_epoch_end methods are called at the end of each training and validation epoch, respectively. They calculate average losses and accuracies and save the model.

The MetricTracker class is a custom callback that tracks validation accuracy during training and provides flexibility for additional tracking or operations.

[17]:

```
lr_find_results = tuner.lr_find(
    plmodel,
    train_dataloaders=training_dataloader,
    min_lr=1e-7,
    max_lr=1e-3,
    early_stop_threshold=None
)
new_lr = lr_find_results.suggestion();
print("suggested learning_rate" + str(new_lr))
```

Expand

[33]:

```
trainer.fit(plmodel,training_dataloader,validation_dataloader)
```

the code utilizes a tuner object to perform a learning rate range test on the plmodel, determines the suggested learning rate based on the test results, and outputs the suggested learning rate for potential use in training the model with an optimal learning rate.

DATALOADER:

Dataloaders init

```
train_dataset=TensorDataset(train_ds,train_ans_type,y_train_encoded,y_train_answerability)
training_dataloader=DataLoader(train_dataset,batch_size=128)
print(train_ans_type.shape)
print(y_train_encoded.shape)


validation_dataset= TensorDataset(val_ds,val_ans_type,y_val_encoded,y_valid_answerability)
validation_dataloader= DataLoader(validation_dataset,batch_size=128)
print(val_ans_type.shape)
print(y_val_encoded.shape)
testing_dataset= TensorDataset(test_ds,test_ans_type,y_test_encoded,y_test_answerability)
testing_dataloader= DataLoader(testing_dataset,batch_size=128)


plmodel= LightningModel(max_occ,weight_decay=0.001)
trainer= pl.Trainer(max_epochs=100)
tuner=pl.tuner.tuning.Tuner(trainer)
```

```
torch.Size([19496])
torch.Size([19496, 5458])
torch.Size([4319])
torch.Size([4319, 5458])
```

```
trainer.fit(plmodel,training_dataloader,validation_dataloader)
```

```
/opt/conda/lib/python3.10/site-packages/pytorch_lightning/trainer/connectors/logger_connector/result.py:432: PossibleUserWarning: It is recommended to use `self.log('val_loss', ..., sync_dist=True)` when logging on epoch level in distribute
d setting to accumulate the metric across devices.
  warning_cache.warn(
/opt/conda/lib/python3.10/site-packages/pytorch_lightning/trainer/connectors/logger_connector/result.py:432: PossibleUserWarning: It is recommended to use `self.log('val_acc', ..., sync_dist=True)` when logging on epoch level in distributed
setting to accumulate the metric across devices.
  warning_cache.warn(
/opt/conda/lib/python3.10/site-packages/pytorch_lightning/trainer/connectors/logger_connector/result.py:432: PossibleUserWarning: It is recommended to use `self.log('val_answerability', ..., sync_dist=True)` when logging on epoch level in d
istributed setting to accumulate the metric across devices.
  warning_cache.warn(
```

Epoch 99: 100% 77/77 [00:03<00:00, 19.59it/s, v_num=15, val_loss=1.990, val_acc=0.531, val_answerability=0.720, train_loss=1.490, train_acc=0.788, train_answerability=0.775]

```
train_acc = trainer.callback_metrics["train_acc"]
val_acc = trainer.callback_metrics["val_acc"]
train_loss = trainer.callback_metrics["train_loss"]
val_loss = trainer.callback_metrics["val_loss"]

print("training accuracy",np.array(train_acc))
print("training loss",np.array(train_loss))

print("validation accuracy",np.array(val_acc))
print("validation loss",np.array(val_loss))
```

```
training accuracy 0.78770006
training loss 1.4934309706678455
validation accuracy 0.5314815
validation loss 1.986055365546404
```

```
plmodel.eval()  # Set the model to evaluation mode

predicted_outputs = []

for batch in testing_dataloader:
    output = plmodel.predict_step(batch, None)  # Call the predict_step() function
    predicted_outputs.append(output)

print(predicted_outputs)
```

```
[(tensor([[ 2.3051,  0.8514,  0.4826,  ...,  0.0341,  1.7699, -0.7573],
        [-0.9067,  0.9967, -0.3617,  ..., -0.9648, -0.5415, -0.5470],
        [-1.0393, -0.9361,  0.4568,  ..., -0.3786,  0.6009, -0.4293],
        ...,
        [-0.5858,  0.0971, -0.3826,  ..., -0.7637, -0.7449, -0.0368],
        [-2.2065, -3.1868,  0.6954,  ...,  1.1388,  0.8770,  2.0465],
        [ 0.5409, -1.5038,  2.4959,  ..., -2.2972,  0.6138,  1.4075]],
       grad_fn=<MulBackward0>), tensor([[  5.4616,   7.1146,   3.1829,  -9.9359],
        [ 10.9999,   8.8829,   6.1532, -15.8376],
```

```python
predicted_answers,predicted_answers_type,predicted_answerabilities=predicted_outputs[0]
for output in predicted_outputs[1:]:
    predicted_answer,predicted_answer_type,predicted_answerability=output
    predicted_answers=torch.cat((predicted_answers,predicted_answer),0)
    predicted_answers_type=torch.cat((predicted_answers_type,predicted_answer_type),0)
    predicted_answerabilities=torch.cat((predicted_answerabilities,predicted_answerability),0)

#testing accuracies
predicted_answers=torch.argmax(predicted_answers,dim=1)
true_answers= torch.argmax(y_test_encoded,dim=1)
testing_accuracy=0.0
for i in range(predicted_answers.shape[0]):
    if true_answers[i]==predicted_answers[i]:

        testing_accuracy+=1
testing_accuracy/=predicted_answers.shape[0]
print("testing accuracy : {0} ".format(testing_accuracy))
#calculate testing answerability
predicted_answerabilities=torch.argmax(predicted_answerabilities,dim=1)
actual_answerabilities=y_test_answerability
answerability=0.0
for i in range(predicted_answerabilities.shape[0]):
    if actual_answerabilities[i]==predicted_answerabilities[i]:
        answerability+=1
answerability/=predicted_answerabilities.shape[0]
print("testing answerability : {0}" .format(answerability))
```

```
testing accuracy : 0.5589094449853943
testing answerability : 0.7585199610516066
```

Bonus part take test data from the test in the ViWiz test set

load test json file to data frame

```
test_bonus = pd.read_json('/kaggle/input/vizwiz/Annotations/Annotations/test.json')
test_bonus = test_bonus[0:50]
print(test_bonus)
```

```
import matplotlib.pyplot as plt
from PIL import Image
import random
#choose random 5 images
random_integers = random.sample(range(51), 5)
for i in random_integers:
    ques=test_bonus.iloc[i]['question']
    print(ques)
    # Specify the image file path
    image_path = "/kaggle/input/vizwiz/test/test/"+str(test_bonus.iloc[i]['image'])

    # Open the image using PIL (Python Imaging Library)
    image = Image.open(image_path)

    # Display the image using Matplotlib
    plt.imshow(image)
    plt.axis('off')
    plt.show()
    print("Answer:",answers[i])
    print()
```

```
What is this? And what color is it?
```

TRAINING TRIAL 1: LOW NUMBER OF EPOCHS(30), NO HYPER PARAMETER TUNING , NO REGULARIZATION.

RESULTS:

```
training accuracy 0.5177472
training loss 2.9474460124268655
validation accuracy 0.27037036
validation loss 4.065017604850756
```

```
teting accuracy : 0.2843232716650438‌4
testing answerability : 0.7848101265822784
```

BONUS PART: TAKING RANDOM TEST DATA AND PREICTING CLASSIFICATION

```
what is this?
```

```
Answer: keyboard
```

What does the photo on the wall show?

Answer: unanswerable

What is this?

Answer: unsuitable

What is this game?

Answer: just dance 3

<u>NOTES: </u> AS WE CAN SEE THE CLSSIFCATION WASN'T EFFIECNT ENOUGH BECAUSE OF THE LOW MODEL COMPLEXITY AND NO TUNING FOR HYPER PARAMTERS AND LOW NUMBER OF EPOCHS , BUT STILL THERE WAS FEW CORRECT CLASSIFCATIONS.

TRAINING TRIAL 2: HIGHER NUMBER OF EPOCHS(100), HYPER PARAMETER TUNING , REGULARIZATION LIKE NORMALIZATION AND L2 REGULARIZATION.

```
training accuracy 0.78770006
training loss 1.4934309706678455
validation accuracy 0.5314815
validation loss 1.986055365546404
```

```
testing accuracy : 0.5589094449853943
testing answerability : 0.7585199610516066
```

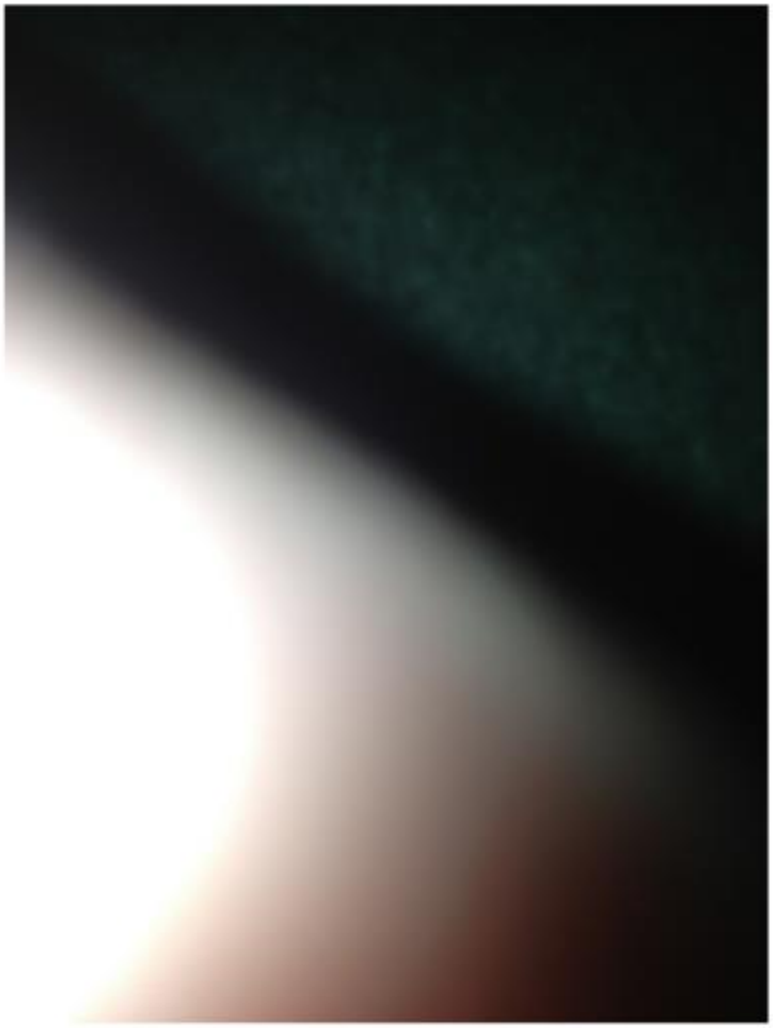
+ Code + Markdown

BOUNS PART : GENERATING RANDOM TEST DATA

What is this? And what color is it?

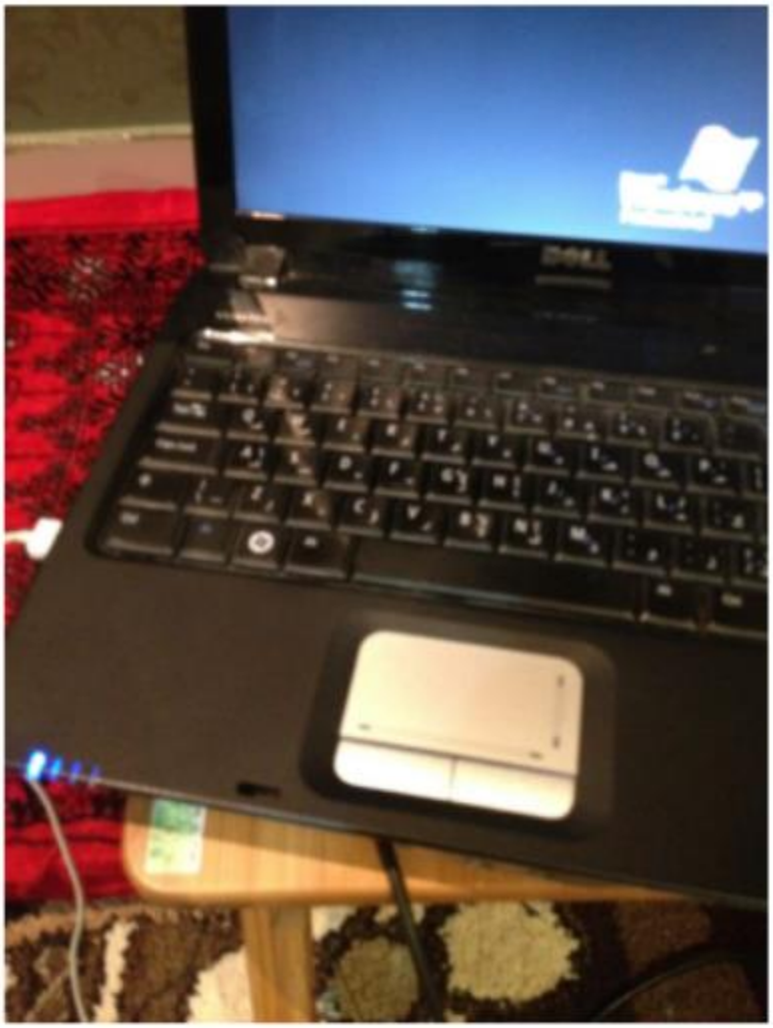
Answer: black

What is this?



Answer: unsuitable

What is this?



Answer: remote control

What is this?

Answer: bottle

NOTES: AS WE CAN SEE THE MODEL AFTER LEARNING AND TRAINING MORE EPOCHS AND TUNING HYPER PARAMTERS LEARNED EVEN BETTER AND CLASSIFIED THE IMAGES CORRECTLY BUT OFCOURSE THERE ARE STILL WRONG CLASSIFCATIONS DUE TO THE LOW NUMBER OF IMAGES IN DATASET