# Assignment Details

Build out a modular convolutional neural network (CNN) that performs image classification on the CIFAR10 dataset. It should also process batches efficiently using `einops`.

We will follow the steps outlined below to complete the task.

1. Create a transforms to resize the image and normalize the tensors.
2. Create the train and test Dataset and DataLoaders.
3. Build out the model, keeping size in mind.
4. Initializing the optimizers, loss functions, etc.
5. Writing out the training loop.
6. Execution.

## Step 0: Importing the necessary libraries.

Our primary library for building out the model will be done in `torch`. For tensor manipulation, we will be using `einops` instead of `torch`'s own tensor manipulation methods.

We will be using `tqdm` as the progress bar.

```
# Requirements to install einops
!pip install monai
!pip install einops

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms, datasets
from einops import rearrange
from tqdm import tqdm
```

```
Requirement already satisfied: monai in
/opt/conda/lib/python3.10/site-packages (1.3.2)
Requirement already satisfied: torch>=1.9 in
/opt/conda/lib/python3.10/site-packages (from monai) (2.4.0)
Requirement already satisfied: numpy>=1.20 in
/opt/conda/lib/python3.10/site-packages (from monai) (1.26.4)
Requirement already satisfied: filelock in
/opt/conda/lib/python3.10/site-packages (from torch>=1.9->monai)
(3.15.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/opt/conda/lib/python3.10/site-packages (from torch>=1.9->monai)
(4.12.2)
Requirement already satisfied: sympy in
/opt/conda/lib/python3.10/site-packages (from torch>=1.9->monai)
(1.13.2)
```

```
Requirement already satisfied: networkx in
/opt/conda/lib/python3.10/site-packages (from torch>=1.9->monai) (3.3)
Requirement already satisfied: jinja2 in
/opt/conda/lib/python3.10/site-packages (from torch>=1.9->monai)
(3.1.4)
Requirement already satisfied: fsspec in
/opt/conda/lib/python3.10/site-packages (from torch>=1.9->monai)
(2024.6.1)
Requirement already satisfied: MarkupSafe>=2.0 in
/opt/conda/lib/python3.10/site-packages (from jinja2->torch>=1.9-
>monai) (2.1.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/opt/conda/lib/python3.10/site-packages (from sympy->torch>=1.9-
>monai) (1.3.0)
Requirement already satisfied: einops in
/opt/conda/lib/python3.10/site-packages (0.8.0)
```

## Step 1: Create a transforms to resize the image and normalize the tensors.

We will not be augmenting the data in any way. To keep it simple, I've decided to resize all the images in the dataset to be 32x32, and normalize the tensors as common practice.

```
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])

# Initializing the device as well as we will be using the GPU.
device = 'cuda' if torch.cuda.is_available() else 'cpu'
device

'cuda'
```

## Step 2: Create the train and test Dataset and DataLoaders

We will be creating batches of 64 images where the train dataset will be shuffed but the test data won't.

```
train_dataset = datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
test_dataset = datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=64, shuffle=True, num_workers=2)
```

```
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64,
shuffle=False, num_workers=2)

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

## Step 3: Building out the model, keeping size in mind.

If you look at `models.py`, you will notice that there's actually two CNNs that I have implemented. For this example, I will take `VerySmallCNN` as the model that I will use.

The model contains a sequential layer of three convolution layers that upscales the output from 3 to 64. And after each convolution layer, we are adding a dropout layer, with a probability of 0.4 in order to reduce the overfitting problem I faced initially when I built this network.

After the layers, we will perform `MaxPooling` to downsample the features and reduce the noise. And then, we will be doing something handy using `einops`. Let us look at the line of code and discuss.

`out = rearrange(out, 'b c h w -> b (c h w)')`

`out` defines the output tensor, and `b, c, h, w` are the batch size, the channels, the height and width respectively. To put the code in English, the output of the maxpooling will be a 4D tensor of shape: (batch_size, no_channels, height, width) and we are squashing the last three dimensions, into a single value, hence the `(c h w)`. We are converting the 4D tensor to a 2D tensor in order to push to our linear layers and finally apply ReLU.

To this in `torch`, we can do something similar like so: `out = out.view(out.size(0), -1)`. However, using a library `einops` makes the manipulations easier.

```
class VerySmallCNN(nn.Module):
    def __init__(self, n_classes: int):
        """
        Initializes the VerySmallCNN model.

        Args:
            n_classes (int): The number of classes in our output
layer.
            Because of the dataset, our number of classes will be 10.
        """
        super(VerySmallCNN, self).__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.Dropout(p=0.4, inplace=True),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.Dropout(p=0.4, inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
        )
        self.dropout=nn.Dropout(p=0.4, inplace=True)
        self.fc1 = nn.Linear(64 * 16 * 16, 512)
        self.fc2 = nn.Linear(512, n_classes)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        out = self.layers(x)
        out = self.dropout(out)
        out = self.pool(out)
        out = rearrange(out, 'b c h w -> b (c h w)')
        out = self.relu(self.fc2(self.fc1(out)))
        return out
```

## Step 4: Initializing the optimizers, loss functions, etc.

Now that our model bulding is done, we will now initialize the appropriate optimizers and loss functions for this use case. As this is an image classification scenario, we will use Categorical Cross Entropy as the loss function. And for our optimizer, we can use either Adam / Stochastic Gradient Descent (SGD). I found better performance in SGD, so I will be using that optimizer.

```
model = VerySmallCNN(n_classes = 10).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

## Step 5: Writing out the training loop

The training loop is a standard piece of code that's present in every PyTorch model building use-case, where only the loss and accuracy calculations differ based on the use-case. I've added the code for evaluations as well in the `main()` function.

```
def main(train_loader, test_loader, num_epochs):
    for epoch in range(num_epochs):
        train_loss, train_acc = 0, 0
        for inputs, labels in tqdm(train_loader):
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()
            output = model(inputs)
            loss = criterion(output, labels)
            _, pred = torch.max(output, 1)
            loss.backward()
            optimizer.step()

            train_loss += loss.item() * inputs.size(0)
            train_acc += torch.sum(pred == labels.data)

        epoch_loss = train_loss / len(train_dataset)
        epoch_acc = (train_acc.double() / len(train_dataset)) * 100
        print(f"Epoch {epoch+1}/{num_epochs} Loss: {epoch_loss:.3f}
Acc: {epoch_acc:.2f}%")
```

```python
    with torch.no_grad():
        run_loss, run_acc = 0, 0
        for inputs, labels in tqdm(test_loader):
            inputs = inputs.to(device)
            labels = labels.to(device)
            output = model(inputs)
            loss = criterion(output, labels)
            _, pred = torch.max(output, 1)
            run_loss += loss.item() * inputs.size(0)
            run_acc += torch.sum(pred == labels.data)
    test_loss = run_loss / len(test_dataset)
    test_acc = (run_acc.double() / len(test_dataset)) * 100
    print(f"Test Loss: {test_loss:.3f}, Test Acc: {test_acc:.2f}%")
```

## Step 6: Execution!

We will now execute the code and see our model become better in action.

```python
n_epochs: int = 50
main(train_loader, test_loader, n_epochs)
```

```
100%|██████████| 782/782 [00:12<00:00, 62.20it/s]

Epoch 1/50 Loss: 1.926 Acc: 32.90%

100%|██████████| 782/782 [00:12<00:00, 60.51it/s]

Epoch 2/50 Loss: 1.545 Acc: 45.89%

100%|██████████| 782/782 [00:12<00:00, 61.06it/s]

Epoch 3/50 Loss: 1.397 Acc: 50.62%

100%|██████████| 782/782 [00:12<00:00, 61.46it/s]

Epoch 4/50 Loss: 1.291 Acc: 54.56%

100%|██████████| 782/782 [00:12<00:00, 61.07it/s]

Epoch 5/50 Loss: 1.217 Acc: 57.50%

100%|██████████| 782/782 [00:12<00:00, 60.71it/s]

Epoch 6/50 Loss: 1.153 Acc: 59.56%

100%|██████████| 782/782 [00:13<00:00, 59.98it/s]

Epoch 7/50 Loss: 1.104 Acc: 61.48%

100%|██████████| 782/782 [00:12<00:00, 62.84it/s]
```

```
Epoch 8/50 Loss: 1.052 Acc: 63.55%
100%|██████████| 782/782 [00:12<00:00, 61.72it/s]
Epoch 9/50 Loss: 1.009 Acc: 65.03%
100%|██████████| 782/782 [00:12<00:00, 61.31it/s]
Epoch 10/50 Loss: 0.976 Acc: 66.15%
100%|██████████| 782/782 [00:12<00:00, 61.74it/s]
Epoch 11/50 Loss: 0.947 Acc: 67.38%
100%|██████████| 782/782 [00:12<00:00, 61.70it/s]
Epoch 12/50 Loss: 0.925 Acc: 68.00%
100%|██████████| 782/782 [00:12<00:00, 62.12it/s]
Epoch 13/50 Loss: 0.908 Acc: 68.69%
100%|██████████| 782/782 [00:12<00:00, 60.45it/s]
Epoch 14/50 Loss: 0.885 Acc: 69.31%
100%|██████████| 782/782 [00:12<00:00, 60.54it/s]
Epoch 15/50 Loss: 0.871 Acc: 69.89%
100%|██████████| 782/782 [00:12<00:00, 62.81it/s]
Epoch 16/50 Loss: 0.856 Acc: 70.63%
100%|██████████| 782/782 [00:12<00:00, 61.74it/s]
Epoch 17/50 Loss: 0.845 Acc: 70.69%
100%|██████████| 782/782 [00:12<00:00, 63.71it/s]
Epoch 18/50 Loss: 0.832 Acc: 71.38%
100%|██████████| 782/782 [00:12<00:00, 61.22it/s]
Epoch 19/50 Loss: 0.825 Acc: 71.35%
100%|██████████| 782/782 [00:12<00:00, 62.27it/s]
Epoch 20/50 Loss: 0.811 Acc: 72.08%
100%|██████████| 782/782 [00:12<00:00, 61.78it/s]
Epoch 21/50 Loss: 0.800 Acc: 72.40%
100%|██████████| 782/782 [00:12<00:00, 61.60it/s]
```

```
Epoch 22/50 Loss: 0.793 Acc: 72.57%
100%|██████████| 782/782 [00:12<00:00, 62.64it/s]
Epoch 23/50 Loss: 0.787 Acc: 72.69%
100%|██████████| 782/782 [00:12<00:00, 60.74it/s]
Epoch 24/50 Loss: 0.780 Acc: 73.03%
100%|██████████| 782/782 [00:12<00:00, 61.89it/s]
Epoch 25/50 Loss: 0.771 Acc: 73.20%
100%|██████████| 782/782 [00:12<00:00, 61.79it/s]
Epoch 26/50 Loss: 0.766 Acc: 73.43%
100%|██████████| 782/782 [00:12<00:00, 62.28it/s]
Epoch 27/50 Loss: 0.758 Acc: 73.94%
100%|██████████| 782/782 [00:12<00:00, 61.95it/s]
Epoch 28/50 Loss: 0.748 Acc: 74.11%
100%|██████████| 782/782 [00:12<00:00, 63.94it/s]
Epoch 29/50 Loss: 0.742 Acc: 74.15%
100%|██████████| 782/782 [00:12<00:00, 60.69it/s]
Epoch 30/50 Loss: 0.733 Acc: 74.42%
100%|██████████| 782/782 [00:12<00:00, 61.90it/s]
Epoch 31/50 Loss: 0.726 Acc: 74.63%
100%|██████████| 782/782 [00:12<00:00, 61.90it/s]
Epoch 32/50 Loss: 0.718 Acc: 74.96%
100%|██████████| 782/782 [00:12<00:00, 63.40it/s]
Epoch 33/50 Loss: 0.712 Acc: 75.32%
100%|██████████| 782/782 [00:12<00:00, 62.93it/s]
Epoch 34/50 Loss: 0.706 Acc: 75.25%
100%|██████████| 782/782 [00:13<00:00, 60.05it/s]
Epoch 35/50 Loss: 0.707 Acc: 75.44%
100%|██████████| 782/782 [00:12<00:00, 60.66it/s]
```

```
Epoch 36/50 Loss: 0.701 Acc: 75.48%
100%|██████████| 782/782 [00:12<00:00, 61.97it/s]
Epoch 37/50 Loss: 0.692 Acc: 75.90%
100%|██████████| 782/782 [00:12<00:00, 62.18it/s]
Epoch 38/50 Loss: 0.689 Acc: 75.75%
100%|██████████| 782/782 [00:12<00:00, 62.45it/s]
Epoch 39/50 Loss: 0.683 Acc: 76.13%
100%|██████████| 782/782 [00:12<00:00, 61.76it/s]
Epoch 40/50 Loss: 0.678 Acc: 76.50%
100%|██████████| 782/782 [00:12<00:00, 63.35it/s]
Epoch 41/50 Loss: 0.670 Acc: 76.59%
100%|██████████| 782/782 [00:12<00:00, 63.02it/s]
Epoch 42/50 Loss: 0.674 Acc: 76.18%
100%|██████████| 782/782 [00:12<00:00, 60.21it/s]
Epoch 43/50 Loss: 0.668 Acc: 76.60%
100%|██████████| 782/782 [00:12<00:00, 62.57it/s]
Epoch 44/50 Loss: 0.656 Acc: 76.94%
100%|██████████| 782/782 [00:12<00:00, 62.47it/s]
Epoch 45/50 Loss: 0.657 Acc: 77.01%
100%|██████████| 782/782 [00:12<00:00, 63.86it/s]
Epoch 46/50 Loss: 0.657 Acc: 76.80%
100%|██████████| 782/782 [00:12<00:00, 62.56it/s]
Epoch 47/50 Loss: 0.653 Acc: 77.10%
100%|██████████| 782/782 [00:12<00:00, 62.28it/s]
Epoch 48/50 Loss: 0.644 Acc: 77.29%
100%|██████████| 782/782 [00:11<00:00, 65.30it/s]
Epoch 49/50 Loss: 0.641 Acc: 77.24%
100%|██████████| 782/782 [00:12<00:00, 63.22it/s]
```

```
Epoch 50/50 Loss: 0.639 Acc: 77.44%

100%|██████████| 157/157 [00:02<00:00, 68.01it/s]

Test Loss: 1.061, Test Acc: 66.67%
```

```python
path: str = "very_smallcnn.pth"
torch.save(model.state_dict(), path)
```

## Visualizing our results

We will now check how well our predictions are working using the test dataset of CIFAR10. We will first have the list of classes and unnormalize the images, plot the images with the labels and run the same images to the model and compare the labels predicted vs the actual labels.

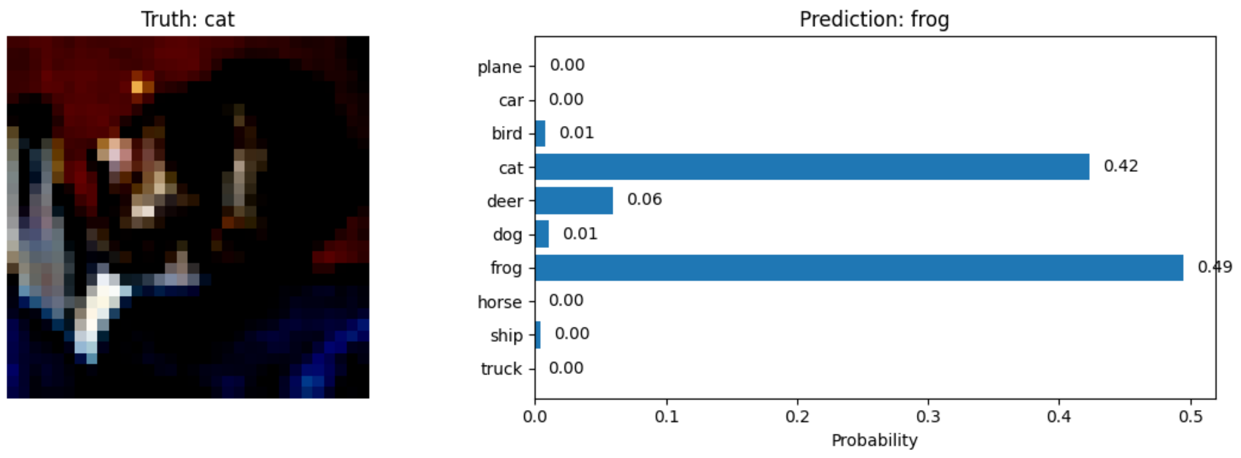
```python
# Loading our saved model weights to test the model.

test_model = VerySmallCNN(n_classes = 10)
test_model.load_state_dict(torch.load("very_smallcnn.pth"))
```

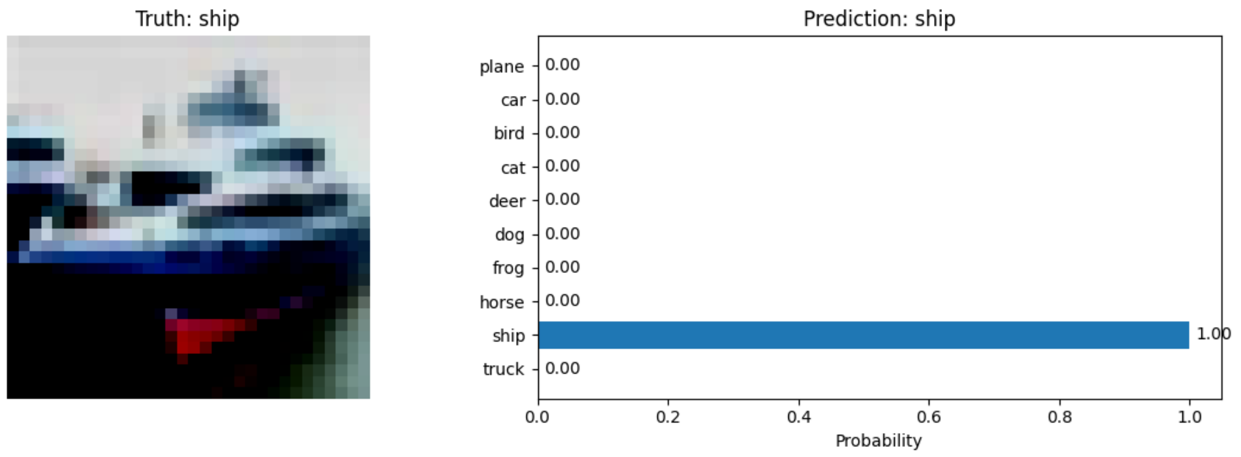
```
/tmp/ipykernel_36/3762676004.py:4: FutureWarning: You are using
`torch.load` with `weights_only=False` (the current default value),
which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code
during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  test_model.load_state_dict(torch.load("very_smallcnn.pth"))

<All keys matched successfully>
```

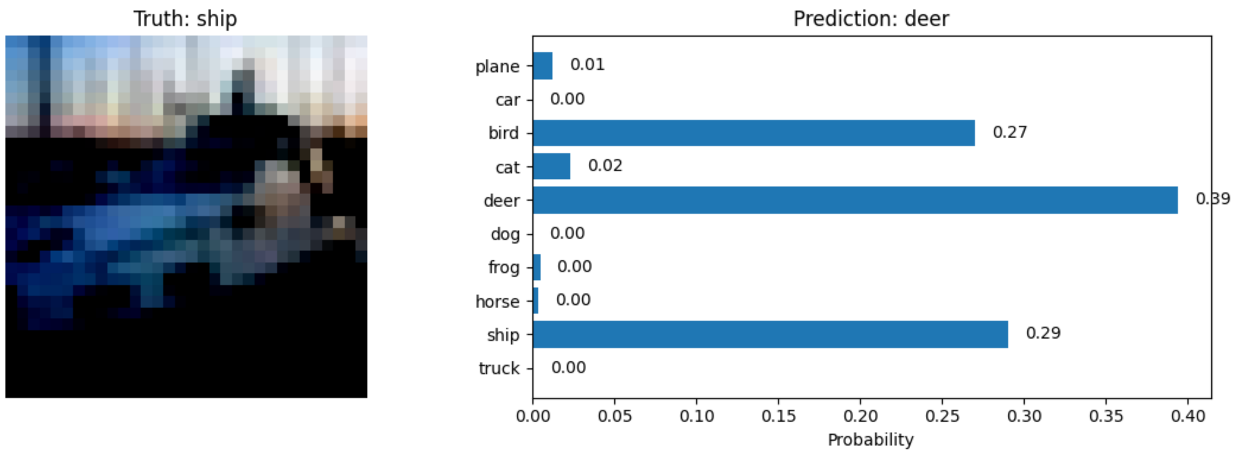
```python
# Define the classes and our helper functions to visualize the
results.
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck')
test_model.to(device)

# Running our model in evaluation mode
test_model.eval()
```

```
VerySmallCNN(
  (layers): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (1): Dropout(p=0.4, inplace=True)
    (2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (3): Dropout(p=0.4, inplace=True)
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  )
  (dropout): Dropout(p=0.4, inplace=True)
  (fc1): Linear(in_features=16384, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=10, bias=True)
  (relu): ReLU()
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
)
```

```python
import numpy as np
import matplotlib.pyplot as plt
import torch.nn.functional as F

def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis('off')

def plot_image_with_labels(img, truth, prediction, probs):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

    # Plot the image
    ax1.imshow(np.transpose(img.numpy(), (1, 2, 0)))
    ax1.axis('off')
    ax1.set_title(f'Truth: {truth}')

    # Plot the probabilities
    y_pos = np.arange(len(classes))
    ax2.barh(y_pos, probs, align='center')
    ax2.set_yticks(y_pos)
    ax2.set_yticklabels(classes)
    ax2.invert_yaxis()  # labels read top-to-bottom
    ax2.set_xlabel('Probability')
    ax2.set_title(f'Prediction: {prediction}')

    # Add text labels to the bars
    for i, v in enumerate(probs):
        ax2.text(v + 0.01, i, f'{v:.2f}', va='center')
    plt.tight_layout()
    plt.show()

# Get some test images and visualize the results
import torchvision

dataiter = iter(test_loader)
img, lab = next(dataiter)

img = img.to(device)
out = test_model(img)
_, pred = torch.max(out, 1)
prob = F.softmax(out, dim=1)

for i in range(3):
    image = img[i]
    truth = classes[lab[i]]
    prediction = classes[pred[i]]
    plot_image_with_labels(image.cpu(), truth, prediction,
prob[i].cpu().detach().numpy())
```







## Caption

The figure on the left is the image that we are taking from the test dataset. Ship, cat and deer are some of the classes in the CIFAR-10 dataset and the plot on the right shows the potential probabilities the figure might be. For example. the first picture where the actual label is a cat, our model predicts a variety of probabilities such as bird, ship, cat, frog, etc.

```
correct, total = 0, 0
with torch.no_grad():
    for data in test_loader:
        img, lab = data
        img, lab = img.to(device), lab.to(device)
        outputs = test_model(img)
        _, pred = torch.max(outputs.data, 1)
        total += lab.size(0)
        correct += (pred == lab).sum().item()

print(f'Accuracy: {100 * correct / total:.2f}%')
```

```
Accuracy: 26.19%
```

## Why did our model perform so low?

It is important to note that we are trading the model's accuracy for the size of the model. We are using a very small convolutional neural network on a large dataset, making the model unable to capture most of the features of the model. With larger networks, ex: VGG models, one can expect far better accuracy, in the ranges of 70-80%. But one can start out with a small network like the one defined above and slowly increase the complexity for better control of the model's performance.

One aspect we can also control is the dataset itself. Augmenting the data creates modifies versions of existing data, allows the model to learn even more patterns and perform better. In this scenario however, there's only so much that the model could learn, but these are potential solutions that can be adopted in the real world.