# MLP_Implementation_CIFAR100

## Summary

This is a simple Multi-Layer Perceptron (MLP) implementation using the CIFAR-100 dataset. The code is modified from the class lecture and PyTorch's documentation.

## Implementation Overview

## Imports

Import all the relevant PyTorch and matplotlib/numpy/sklearn libraries in addition to make_grid which will be used to visualize some of the data, and tqdm for tracking progress.

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
import torchvision.transforms as transforms
from torchvision.utils import make_grid
import tqdm
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import f1_score
```

## Data Preparation

The data is fetched using PyTorch's `datasets.CIFAR100` module, specifying the root directory for storage, whether it's for training or testing, and the transformation pipeline which includes converting the images to tensors and normalizing their pixel values. After downloading the dataset, it's divided and loaded into DataLoader objects for efficient batch-wise processing during training and evaluation.

Confirming the expected size of the dataset, which comprises 60,000 images in total, with 50,000 images allocated for training and 10,000 for testing, distributed across 100 classes, each containing 600 images.

```
# Download the data
transforms = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

training_data = datasets.CIFAR100(
    root = 'data', #directory to store the dataset in
    # True for training dataset, False for testing
    train = True,
    download = True,
    transform = transforms
)

testing_data = datasets.CIFAR100(
    root = 'data',
    train = False,
    download = True,
    transform = transforms
)
classes = training_data.classes

# DataLoader pytorch.org/docs/stable/data.html
batch_size = 128
training_dataloader = DataLoader(training_data, batch_size=batch_size)
testing_dataloader = DataLoader(testing_data, batch_size=batch_size)

# For CIFAR-100 we expect 60k total images
# 100 classes with 600 images each
# that's 50k training and 10k testing (500 training/100 testing per class)
print(f'training: {batch_size*len(training_dataloader)}, testing:
{batch_size*len(testing_dataloader)}')
```

**Output**

```
Files already downloaded and verified
Files already downloaded and verified
training: 50048, testing: 10112
```

## Visualize Data

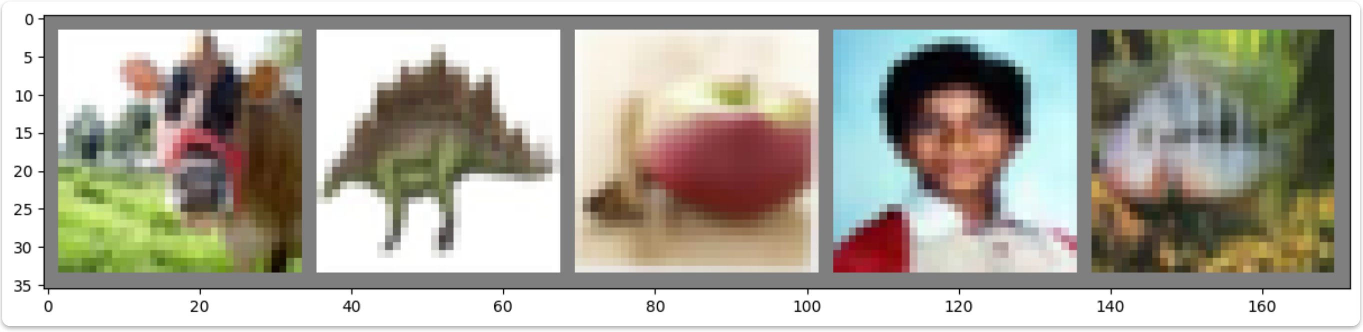
Peek at the data by displaying some images and labels. I chose 5 but you could display as many as needed. You can also manually verify the labels using this list from HuggingFace.

```
# Peek at the data
# manually verify labels are correct:
# https://huggingface.co/datasets/cifar100
def imshow(img):
  img = img / 2 + 0.5
  npimg = img.numpy()
  plt.figure(figsize=(15, 15))
  plt.imshow(np.transpose(npimg, (1, 2, 0)))
  plt.show()

dataiter = iter(training_dataloader)
images, labels = next(dataiter)
images = images[:5]
labels = labels[:5]

imshow(make_grid(images))
print('Ordered Labels:')
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(5)))
```

### Output



```
Ordered Labels:
cattle dinosaur apple boy   aquarium_fish
```

Another visualization using modified code from class:

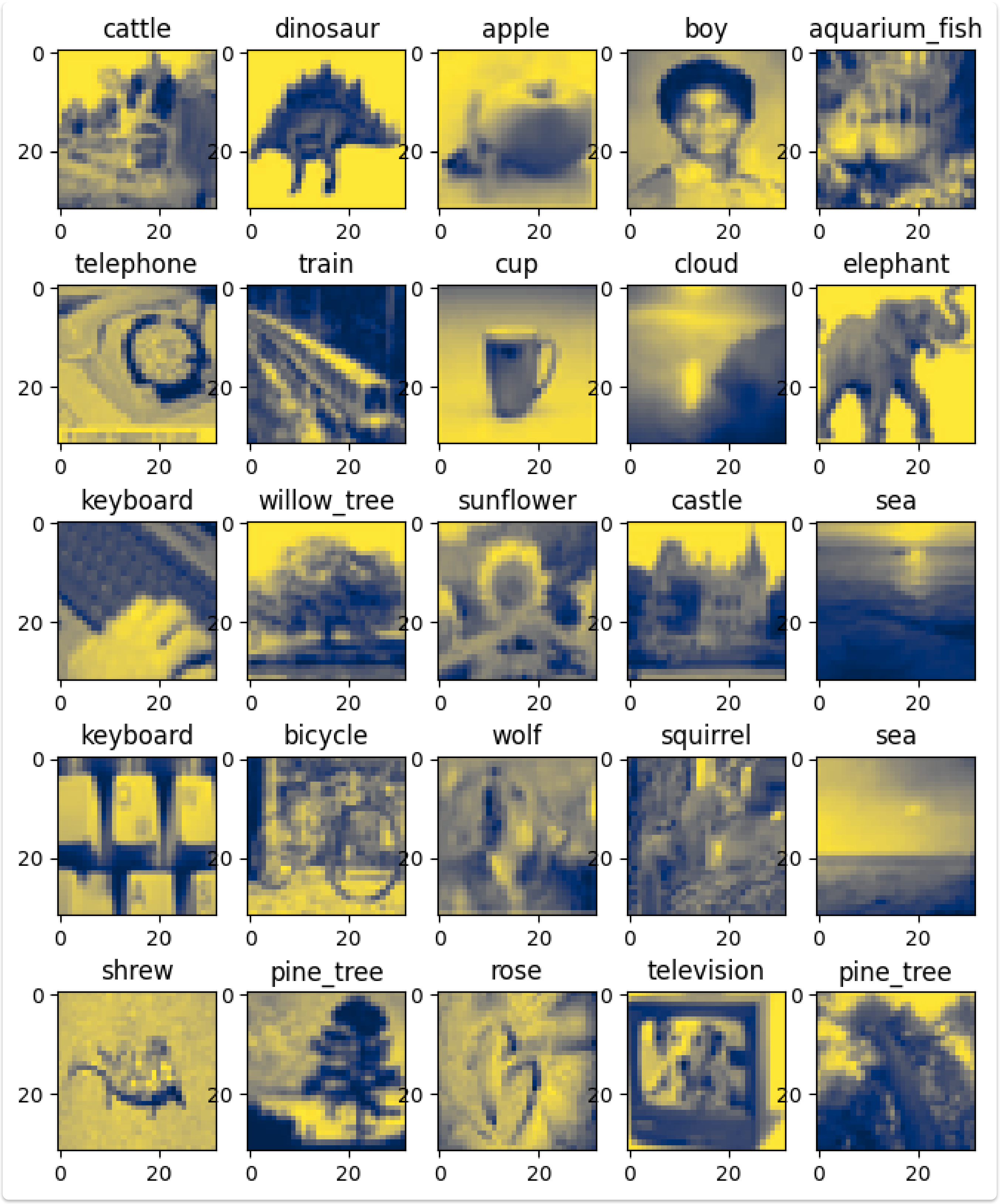
```
# each data is length 2 where data[0] is data and data[1] is fine label
for data in training_dataloader:
  break
```

```
# manually verify labels are correct:
# https://huggingface.co/datasets/cifar100
X = data[0]
Y = data[1]

colormap = 'cividis'
print(len(data[1]))
print('shape of X [batch, channel, height, width]:', X.shape)
print('shape of Y:', Y.shape)
plt.figure(figsize=(8,10))
for i in range(25):
  plt.subplot(5,5,i+1)
  # https://matplotlib.org/stable/users/explain/colors/colormaps.html#colormaps
  plt.imshow(X[i,0,:,:], cmap=colormap)
  plt.title(classes[Y[i].item()])
```

**Output**

```
128
shape of X [batch, channel, height, width]: torch.Size([128, 3, 32, 32])
shape of Y: torch.Size([128])
```



## Define MLP Class

This code defines a Multi-Layer Perceptron (MLP) neural network class using PyTorch. The MLP architecture consists of an input layer, three hidden layers with ReLU activation functions, and an output layer. The input layer accepts flattened images, where each image is represented as a column vector. The sizes of the input and output layers are determined by the dimensions of the CIFAR-100 dataset, with 32x32 pixels and 3 color channels (RGB) resulting in an input feature size of 32 *32* 3. The hidden layers are set to 512, 256, and 256 neurons respectively, while the output layer corresponds to 100 classes or labels in the CIFAR-100 dataset.

The `forward` method defines the flow of data through the network. It begins by flattening the input images using `nn.Flatten()` and then passes them through a sequence of linear transformations followed by ReLU activation functions using `nn.Linear()` and `nn.ReLU()` respectively. The output of the final linear transformation yields the logits, representing the raw scores for each class, which are then used for classification. This MLP architecture serves as a baseline model for image classification tasks on the CIFAR-100 dataset.

```
# Define MLP Class
in_features =  32 * 32 * 3 # 32x32 px and 3 channel (RGB)
hidden_layer_1 = 1024
hidden_layer_2 = 512
hidden_layer_3 = 256
out_features = 100 # 100 classes/labels

class MLP(nn.Module):
  # define structure
  def __init__(self):
    super(MLP, self).__init__()
    # flatten image to a column vector
    self.flatten = nn.Flatten()
    self.linear_relu_stack = nn.Sequential(
        # input layer
        nn.Linear(in_features, hidden_layer_1),
        # non-linear activation
        nn.ReLU(),

        # hidden layers
        nn.Linear(hidden_layer_1, hidden_layer_2),
        nn.ReLU(),
        nn.Linear(hidden_layer_2, hidden_layer_2),
        nn.ReLU(),
        nn.Linear(hidden_layer_2, hidden_layer_3),
        nn.ReLU(),

        # output layer
        nn.Linear(hidden_layer_3, out_features)
    )
  # data flow
  def forward(self, x):
    x = self.flatten(x)
    logits = self.linear_relu_stack(x)

    return logits
```

## Training and Testing

This code (some of it modified from class) defines the training and testing procedures for the MLP model on the CIFAR-100 dataset. The `train` function is responsible for training the model. It iterates through the provided data loader, which supplies batches of training data. For each batch, it performs the following steps: sends the input data and labels to the appropriate device (CPU or GPU), computes the model's predictions, calculates the loss using the specified loss function, performs backpropagation to compute gradients, and updates the model's parameters using the optimizer.

During training, the function periodically prints the step number and the corresponding loss to monitor training progress.

The `test` function evaluates the trained model's performance on a separate validation set. It operates similarly to the training function but without backpropagation. Instead, it accumulates the total loss and the number of correctly classified samples. After processing all batches, it computes the average loss and the accuracy of the model on the validation set. Finally, it prints the test accuracy, providing insight into the model's performance on unseen data. These training and testing procedures enable the evaluation and refinement of the MLP model for image classification tasks on the CIFAR-100 dataset.

*Note* some of the outputs below do not have the f1 score as it was added late in development after double checking the assignment instructions.

```
# Define Training

def train(dataloader, model, loss_fn, optimizer, device):
  # model has train and evaluation modes
  model.train()
  # step being a training step
  for step, (X, y) in enumerate(dataloader):
    # send to CPU/GPU
    X = X.to(device)
    y = y.to(device)
    # model's prediction
    pred = model(X)
    # get loss
    loss = loss_fn(pred, y)
    # backprop
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # progress check
    if step % 100 == 0:
      print(f'step: {step}, loss: {loss.item()}')

# validation
def test(dataloader, model, loss_fn, device):
  num_steps = len(dataloader)
  model.eval()
  test_loss = 0
  correct = 0
  y_true = []
  y_predicted = []

  with torch.no_grad():
    for X, y in dataloader:
      X = X.to(device)
      y = y.to(device)
      pred = model(X)
      loss = loss_fn(pred, y)
      test_loss += loss.item()
      y_hat = pred.argmax(1)
      correct_step = (y_hat == y).type(torch.float).sum().item()
      correct += correct_step

  test_loss /= num_steps # average loss
  correct = correct / (num_steps * batch_size)
  f1 = f1_score(y_true, y_predicted, average='macro')

  print(f'F1 Score: {f1}')
  print(f'Test Accuracy: {correct}')
```

## Running the Models

This code segment initializes the device for computation, creates an instance of the MLP model, defines the loss function and optimizer, and executes the training loop over a specified number of epochs.

First, it determines whether a GPU (cuda) is available and sets the device accordingly. Then, it instantiates the MLP model and moves it to the selected device.

Next, it defines the loss function as CrossEntropyLoss and the optimizer as Stochastic Gradient Descent (SGD) with a learning rate of 0.001.

The training loop iterates over the specified number of epochs, where each epoch consists of training the model on the training dataset and evaluating its performance on the testing dataset using the `train` and `test` functions defined earlier. tqdm.tqdm is used to visualize the progress of the epochs.

After training completes, it prints "Done!" indicating the completion of the training process. This code orchestrates the training of the MLP model on the CIFAR-100 dataset, enabling the model to learn and improve its performance over multiple epochs.

```
# get device
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f'Using {device}')
# create model
model = MLP().to(device)
print(model)
# optimize
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
# train model
epochs = 5
for t in tqdm.tqdm(range(epochs)):
  print(f'Epoch\n\n{t}')
  train(training_dataloader, model, loss_fn, optimizer, device)
  test(testing_dataloader, model, loss_fn, device)
print('Done!')
```

## Outputs

### Model # 1, optimizer = SGD

Initially, a single 512 node hidden layer was used.

```
Using cpu MLP(
        (flatten): Flatten(start_dim=1, end_dim=-1)
        (linear_relu_stack): Sequential(
                (0): Linear(in_features=3072, out_features=512, bias=True)
                (1): ReLU()
                (2): Linear(in_features=512, out_features=512, bias=True)
                (3): ReLU()
                (4): Linear(in_features=512, out_features=100, bias=True)
        )
)
```

`5/5 [02:29<00:00, 29.86s/it]Test Accuracy: 0.03045886075949367`

## Model # 2, optimizer = SGD

Given the low accuracy of ~3% more hidden layers were added:

```
Using cpu
MLP(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=3072, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=256, bias=True)
    (5): ReLU()
    (6): Linear(in_features=256, out_features=100, bias=True)
  )
)
```

`5/5 [02:23<00:00, 28.64s/it]Test Accuracy: 0.018591772151898733`

## Model # 3, optimizer = SGD

Surprisingly, the accuracy had gone down to <2%. After altering the number of nodes in the hidden layers the accuracy again went down to <1%.

```
Using cpu
MLP(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=3072, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=256, bias=True)
    (5): ReLU()
    (6): Linear(in_features=256, out_features=256, bias=True)
    (7): ReLU()
    (8): Linear(in_features=256, out_features=100, bias=True)
  )
)
```

```
5/5 [02:06<00:00, 25.23s/it]Test Accuracy: 0.009691455696202531
```

## Model # 4, optimizer = Adam

Given the poor performance, another optimizer was evaluated: Adam. After some reading it was thought the underfitting could be due to the low number of nodes in the hidden layers given 3-channel input images. To address this, the number of nodes in the hidden layers was increased several-fold and was structured in descending order. The idea was each layer would become more and more specific to the fine label. *Note* the significant increase in accuracy, from <1% to >19%.

```
Using cpu
MLP(
        (flatten): Flatten(start_dim=1, end_dim=-1)
        (linear_relu_stack): Sequential(
                (0): Linear(in_features=3072, out_features=2048, bias=True)
                (1): ReLU()
                (2): Linear(in_features=2048, out_features=1024, bias=True)
                (3): ReLU()
                (4): Linear(in_features=1024, out_features=1024, bias=True)
                (5): ReLU()
                (6): Linear(in_features=1024, out_features=1024, bias=True)
                (7): ReLU()
                (8): Linear(in_features=1024, out_features=100, bias=True) ) )
```

```
6/6 [12:37<00:00, 126.18s/it]Test Accuracy: 0.19363132911392406
```

## Model #5, optimizer = Adam

As a last attempt at model

```
Using cpu
MLP(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=3072, out_features=1024, bias=True)
    (1): ReLU()
    (2): Linear(in_features=1024, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=512, bias=True)
    (5): ReLU()
    (6): Linear(in_features=512, out_features=256, bias=True)
    (7): ReLU()
    (8): Linear(in_features=256, out_features=100, bias=True)
  )
)
```

```
5/5 [04:33<00:00, 54.69s/it]
F1 Score: 0.20568181309963277
Test Accuracy: 0.2164754746835443
```

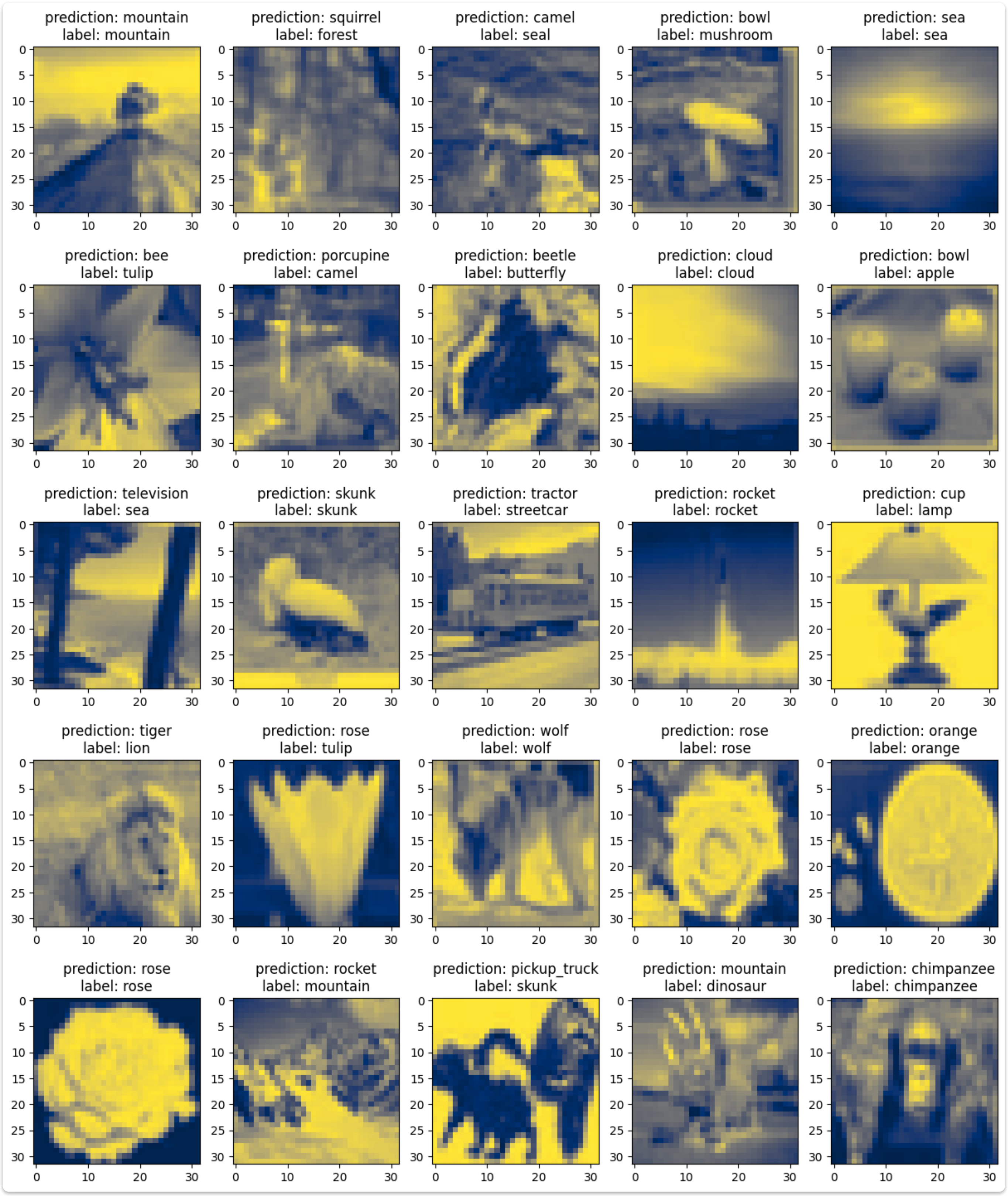
## Evaluation

Though the accuracy is ~22% (which is, of course, low), the model served its purpose as a learning tool.

Some additional transformations could be used to augment the training of the model, but after some reading on PyTorch I believe the only meaningful improvements to this network will come from changing the structure from a MLP to a CNN with much more advanced techniques.

The code snippet below performs inference on a batch of images from a testing dataloader using a trained model. It sets the model to evaluation mode, iterates over the testing dataloader to get a batch of images (`X`) and their corresponding labels (`y`), passes the images through the model to obtain predictions (`pred`), and computes the predicted labels by taking the argmax of the model's output. Then, it visualizes a grid of images along with their predicted labels and ground truth labels using matplotlib. Each subplot in the grid displays an image along with its corresponding prediction and ground truth label, allowing quick visual inspection of the model's performance on the test dataset.

``


prediction: mountain
label: mountain
prediction: squirrel
label: forest
prediction: camel
label: seal
prediction: bowl
label: mushroom
prediction: sea
label: sea
prediction: bee
label: tulip
prediction: porcupine
label: camel
prediction: beetle
label: butterfly
prediction: cloud
label: cloud
prediction: bowl
label: apple
prediction: television
label: sea
prediction: skunk
label: skunk
prediction: tractor
label: streetcar
prediction: rocket
label: rocket
prediction: cup
label: lamp
prediction: tiger
label: lion
prediction: rose
label: tulip
prediction: wolf
label: wolf
prediction: rose
label: rose
prediction: orange
label: orange
prediction: rose
label: rose
prediction: rocket
label: mountain
prediction: pickup_truck
label: skunk
prediction: mountain
label: dinosaur
prediction: chimpanzee
label: chimpanzee