CSE 676 Code Demo and Paper Presentation
Model Visualization using TensorBoard and PyTorch
Team Members : Priya Patil (ppatil22) , Hariharan sriram(hsriram)

Introduction:

TensorBoard is used to make the deep learning models easier by allowing the visual representations of designs, metrics, graphs, images, and how model weights change over time.
It helps us in understanding how the data is prepared before it is integrated into the model.

It helps us to assess model performance and convergence by enabling the tracking of training metrics such as loss and accuracy over epochs and they are plotted in real-time or over time.
This is used for troubleshooting models, improving their performance and training progress as it happens.

When dealing with deep learning, the models can be complex with numerous layers and connections. Using these visualizations, it gives us a better understanding of the different connected and hidden layers that are present in the model.

While using convolutional neural network (CNN), RNN models, it displays the sequential arrangement of convolutional, pooling, and fully connected layers.

During training, validation and testing the model, the visualizations continuously monitor and display key metrics such as loss, recall, accuracy etc. It provides real-time plots and summaries that update as the training progresses. We get to see how the weights change when the model is applied.

During visualizations, it helps us bring transparency to deep learning models which makes us take better decisions at every stage of development.

Importance of Model Visualizations:

The major importance of model visualizations are:

1) Understanding of the Model Architecture
2) Viewing the Model Decisions
3) Debugging and Troubleshooting
4) Monitoring of the Training process
5) Optimizing the model performance by getting insights as plots and graphs in real time

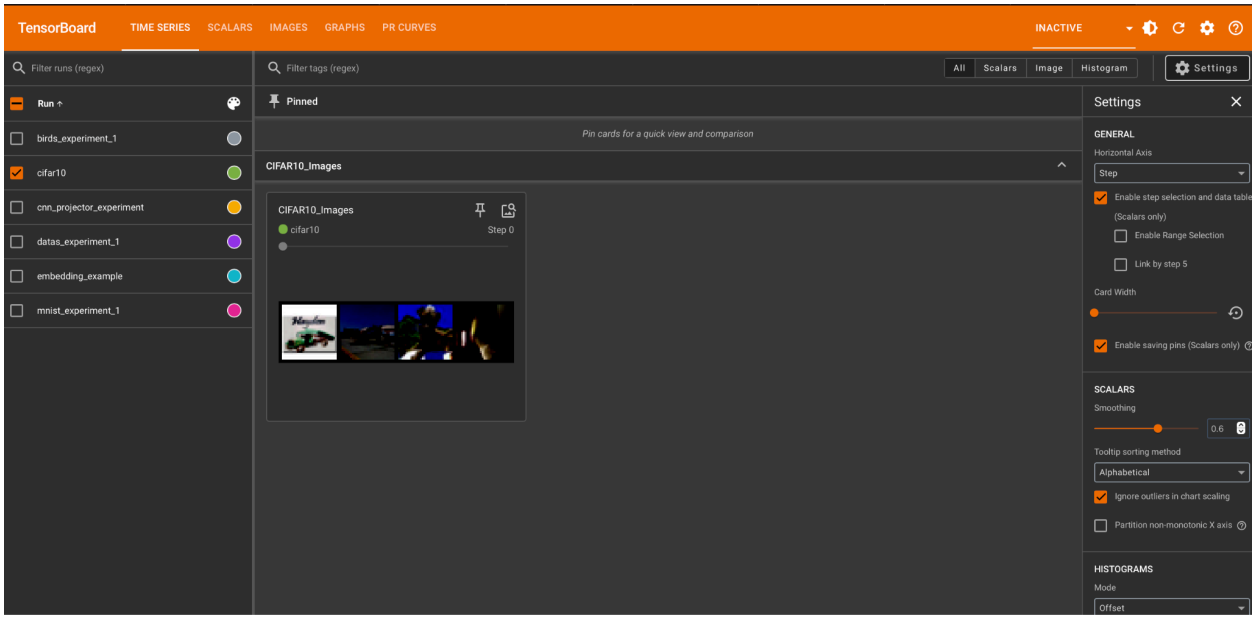
Overview of Tensor Board:

TensorBoard is like a magnifying glass for deep learning projects. It lets you visualize what's happening inside your model. While training complex models line CNN, RNN etc, TensorBoard shows graphs, charts, and images to help us debug, understand, and optimize its performance. It's like watching your model learn in real-time, catching issues and making sure it's on the right track.

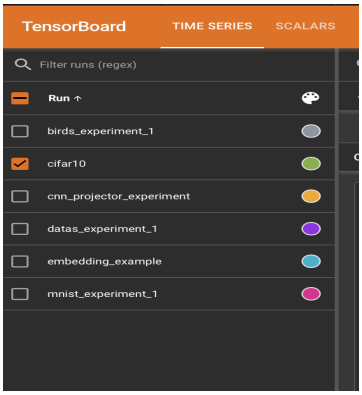
Key features:

1) **Visualization of Metrics:** To track and visualize training metrics like loss,accuracy, recall etc over time(epoch)
2) **Inspecting the DL Models:** To see the structure of the deep learning models with graphs that shows the layers and operations involved.
3) **Exploring the Data:** Viewing distributions of weights, biases, and other data using histograms and graphs.
4) **Analyzing the Performance :** To analyze how the model works and to see the results.

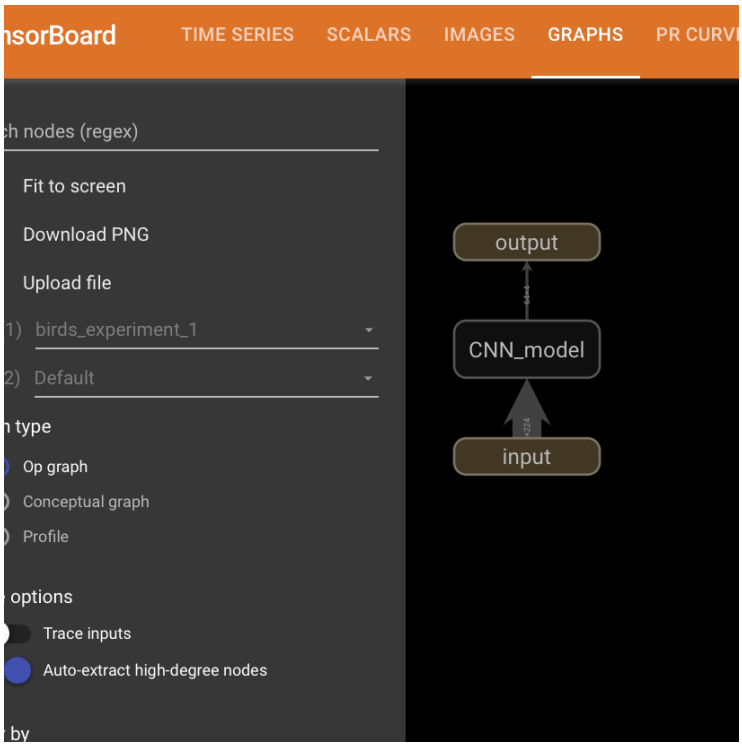
A quick review of features present in TensorFlow:



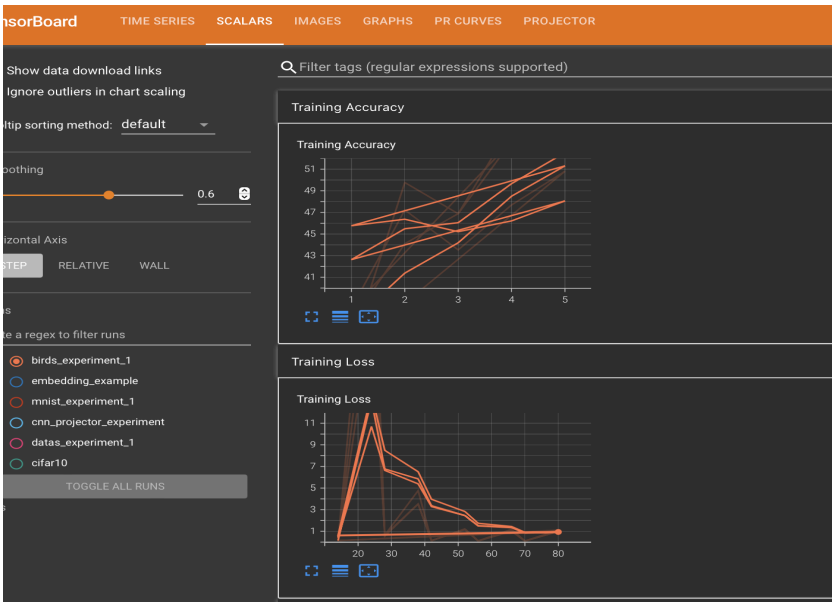
The TensorBoard consists of time series, scalars, images, graphs, PC curves.
We get to see and analyze the SummaryWriter of different writers for different projects and helps us to compare them and their performance.



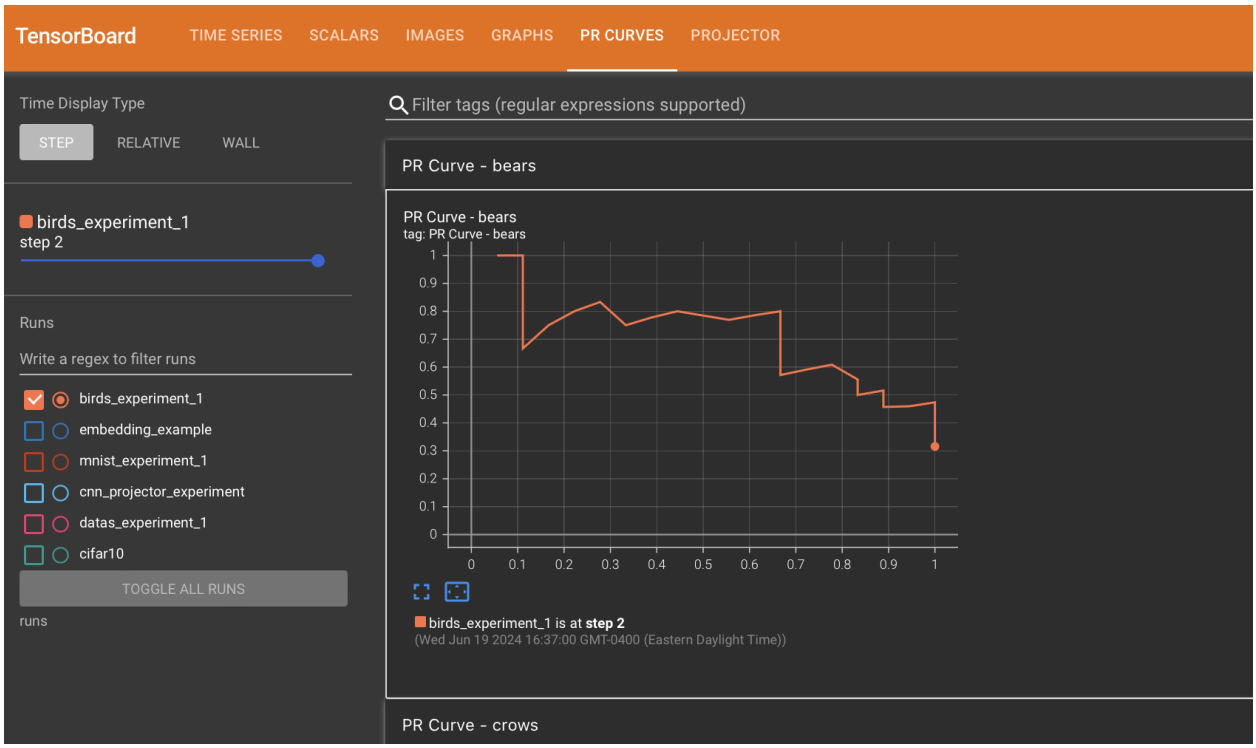
Under the graphs section, we can see the different types of graphs which helps in analysis of the model. For example,



Under Scalars, we get to analyze the metrics like loss, accuracy etc across real time, if there are any. For example,



Under PR Curves, we get to see and compare the different Precision-Recall curves for different models and datasets applied and compare them in real time for different classes present in the datasets. For example,



There are various other features that can be visualized in the TensorFlow board which helps us to analyze the working of the models applied and compare the outputs such as metrics like accuracy, loss, precision, recall etc. which are depicted as histograms or graphs. This helps us to improve and train the model's working more efficiently.

Why is Model Visualization different?

Generally, the traditional methods often rely on summary statistics, tables, or textual descriptions to analyze data and model outputs. But Model Visualization provides us graphical representations that helps us to analyze the model more effectively.

Model visualization by TensorFlow helps us to view the inner workings of models, such as activations, gradients, and feature importance etc which allows us for monitoring the model in real time so that we can debug during training itself.

General methods involve testing the model manually. But using TensorFlow Model Visualization, we can find the performance metrics, parametre tuning and other methods to make the model work efficiently.

About the Dataset:

The dataset used for this model is **CIFAR-10**.

The CIFAR-10 dataset consists of **60,000** color images with a size of 32x32 pixels.

It has over **10** classes: **'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'.**

Each class has **6000** images.

This dataset is widely used for image classification where various deep learning and machine learning models can be applied.

The dataset has images size being small and as it contains various images of objects that are different from each other making it. Due to these features, it is challenging and is one of the best datasets to evaluate the performance and analysis of the models being implemented.

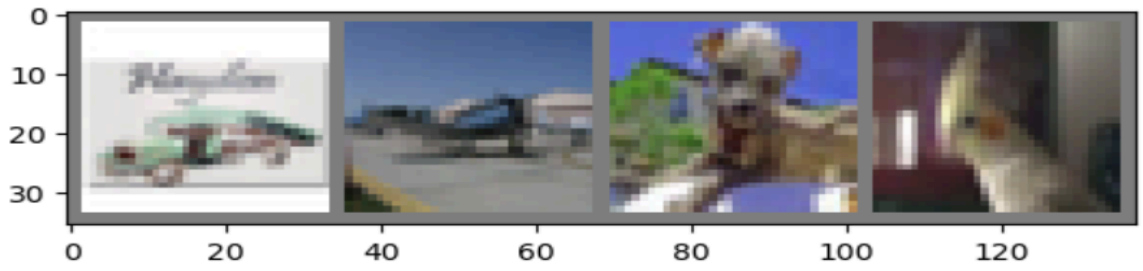


Fig. CIFAR-10 dataset

Steps to create a TensorBoard for your Deep Learning Model :

Step 1: Install the tensor board library

Install the tensorboard library using the pip command. TensorBoard is a library, integrated with PyTorch via the torch.utils.tensorboard module, allows for logging and visualizing training metrics, model architecture, and other data to monitor and debug the training process effectively.

```
Last login: Wed Jun 19 17:04:03 on ttys005
(base) priyapatil@Priyas-MBP-3 ~ % pip install tensorboard
Requirement already satisfied: tensorboard in ./anaconda3/lib/python3.11/site-packages (2.16.2)
Requirement already satisfied: absl-py>=0.4 in ./anaconda3/lib/python3.11/site-packages (from tensorboard) (2.1.0)
Requirement already satisfied: grpcio>=1.48.2 in ./anaconda3/lib/python3.11/site-packages (from tensorboard) (1.62.2)
Requirement already satisfied: markdown>=2.6.8 in ./anaconda3/lib/python3.11/site-packages (from tensorboard) (3.4.1)
Requirement already satisfied: numpy>=1.12.0 in ./anaconda3/lib/python3.11/site-packages (from tensorboard) (1.24.3)
Requirement already satisfied: protobuf!=4.24.0,>=3.19.6 in ./anaconda3/lib/python3.11/site-packages (from tensorboard) (4.25.3)
Requirement already satisfied: setuptools>=41.0.0 in ./anaconda3/lib/python3.11/site-packages (from tensorboard) (68.0.0)
Requirement already satisfied: six>1.9 in ./anaconda3/lib/python3.11/site-packages (from tensorboard) (1.16.0)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in ./anaconda3/lib/python3.11/site-packages (from tensorboard) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in ./anaconda3/lib/python3.11/site-packages (from tensorboard) (2.2.3)
Requirement already satisfied: MarkupSafe>=2.1.1 in ./anaconda3/lib/python3.11/site-packages (from werkzeug>=1.0.1->tensorboard) (2.1.1)
(base) priyapatil@Priyas-MBP-3 ~ %
```

Step 2 : Importing the required Libraries

Import all the required libraries for data loading, preprocessing, building the neural network, and other related tasks.

```
: import matplotlib.pyplot as plt
  import numpy as np
  import torch
  import torchvision
  import torchvision.transforms as transforms
  import torch.nn as nn
  import torch.nn.functional as F
  import torch.optim as optim
  from torch.utils.tensorboard import SummaryWriter
```

Step 3 : Data Pre-processing

Data Pre-processing

- Defining the transformer to tranform the input size of the image ,convert it into a tensor format and Normalize the array

```
In [ ]: transform = transforms.Compose(
            [transforms.ToTensor(),
             transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

- Downloading the test and train dataset and loading those dataset using dataloader

```
In [ ]: batch_size = 4

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

        classes = ('plane', 'car', 'bird', 'cat',
                   'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

The preprocessing pipeline involves converting images from PIL Image or numpy.ndarray format to PyTorch tensors using transforms.ToTensor() and scales the pixel values to the range [0.0, 1.0]. Then, transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) is applied to normalize each channel to the range [-1, 1]. Using DataLoader, the dataset is separated into batches of 4 samples. The test data is left un shuffled to preserve order during evaluation, while the training data is shuffled at each epoch to ensure unpredictability. This setup, including worker threads for parallel data loading, ensures efficient and standardized data preparation for training and testing the neural network.

Step 4: Neural Network setup

Define the Neural Network Setup for the dataset

```
In [ ]: class Net(nn.Module):
            def __init__(self):
                super().__init__()
                self.conv1 = nn.Conv2d(3, 6, 5)
                self.pool = nn.MaxPool2d(2, 2)
                self.conv2 = nn.Conv2d(6, 16, 5)
                self.fc1 = nn.Linear(16 * 5 * 5, 120)
                self.fc2 = nn.Linear(120, 84)
                self.fc3 = nn.Linear(84, 10)

            def forward(self, x):
                x = self.pool(F.relu(self.conv1(x)))
                x = self.pool(F.relu(self.conv2(x)))
                x = torch.flatten(x, 1) # flatten all dimensions except batch
                x = F.relu(self.fc1(x))
                x = F.relu(self.fc2(x))
                x = self.fc3(x)
                return x


        net = Net()
```

Since this is an image classification problem a typical CNN setup is used in order to leverage its hierarchical feature extraction capability .The network architecture consists of two convolutional layers, each followed by a max-pooling layer, and three fully connected layers. It uses ReLU activations after each convolution and fully connected layer.

Step 5: Setting up the Loss function and Optimizer for the Neural Network Setup

Define a loss function and a optimizer for the model

```
In [ ]: criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

After defining the neural network architecture, the loss function and optimizer are set up for training.The loss function, nn.CrossEntropyLoss(), is used to compute the loss between the predicted class probabilities and the true class labels, since its a classification and it would be optimum for the scenario. The optimizer, optim.SGD(net.parameters(), lr=0.001, momentum=0.9), employs Stochastic Gradient Descent with a learning rate of 0.001 and momentum of 0.9. By keeping the gradients in the same direction, it improves convergence speed and facilitates effective training.

Step 6: Define the SummaryWriter object

Use the SummaryWriter to write on the tensorboard

- Create an object of the SummaryWriter and 'runs/cifar10' specifies the directory where the SummaryWriter will save the log files for TensorBoard, which will store metrics, images, and other data related to the CIFAR-10 experiment.

In [ ]:
```
writer = SummaryWriter('runs/cifar10')
```

SummaryWriter from TensorBoard is a utility in PyTorch that logs training metrics, images, and other data to be visualized in TensorBoard, helping to monitor and analyze the model's performance and training process. So now create an object of this utility.

Step 7: Write the training images on the tensor board

Helper Functions

- This method displays a PyTorch tensor as an image using Matplotlib, optionally converting it to grayscale.

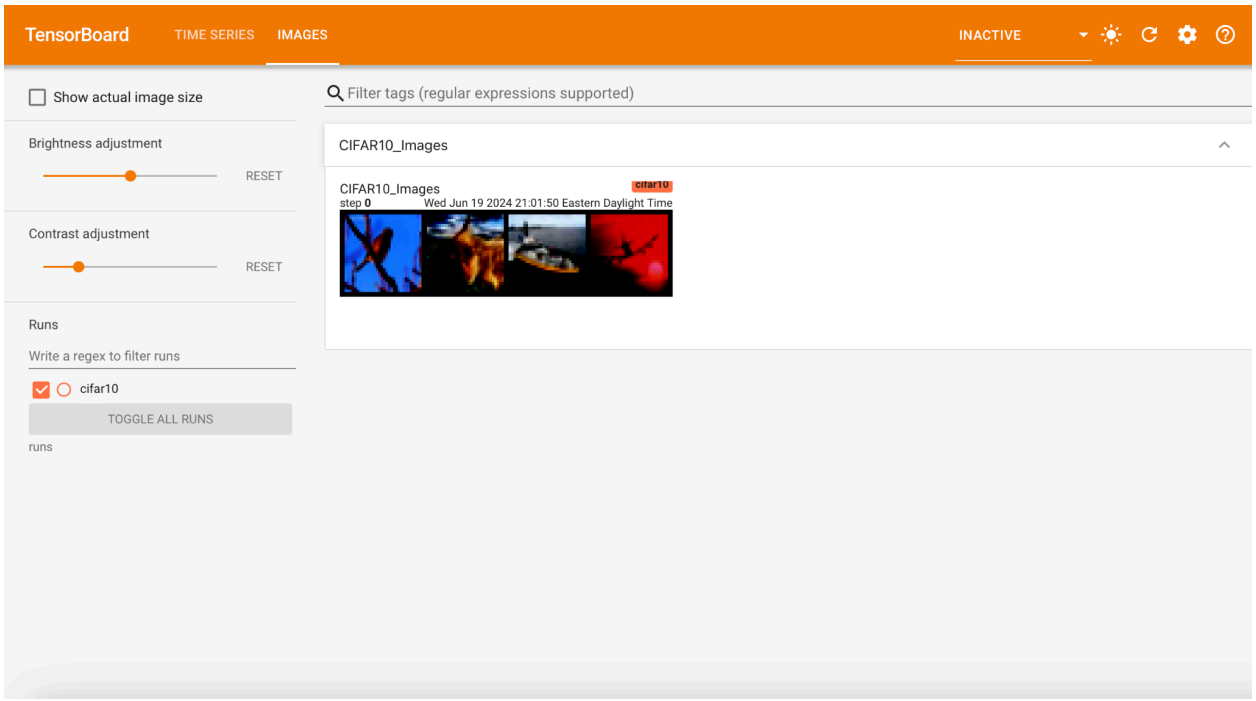
In [ ]:
```
def matplotlib_imshow(img, one_channel=False):
    if one_channel:
        img = img.mean(dim=0)
    img = img / 2 + 0.5
    npimg = img.numpy()
    if one_channel:
        plt.imshow(npimg, cmap="Greys")
    else:
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
```

Helper Function : The **matplotlib_imshow** function displays images using Matplotlib, handling single-channel images by averaging the color channels. It unnormalizes the image tensor from the range [-1, 1] to [0, 1], converts it to a NumPy array, and plots it using the "Greys' ' colormap for grayscale images or the default for RGB images.

Write an image onto the TensorBoard, in form of a grid - using make_grid.

In [ ]:
```
dataiter = iter(trainloader)
images, labels = next(dataiter)

img_grid = torchvision.utils.make_grid(images)
matplotlib_imshow(img_grid, one_channel=False)
writer.add_image('CIFAR10_Images', img_grid)
```

Using the above mentioned helper function and writer.add_img() to log a batch of images as a grid using Matplotlib and TensorBoard.



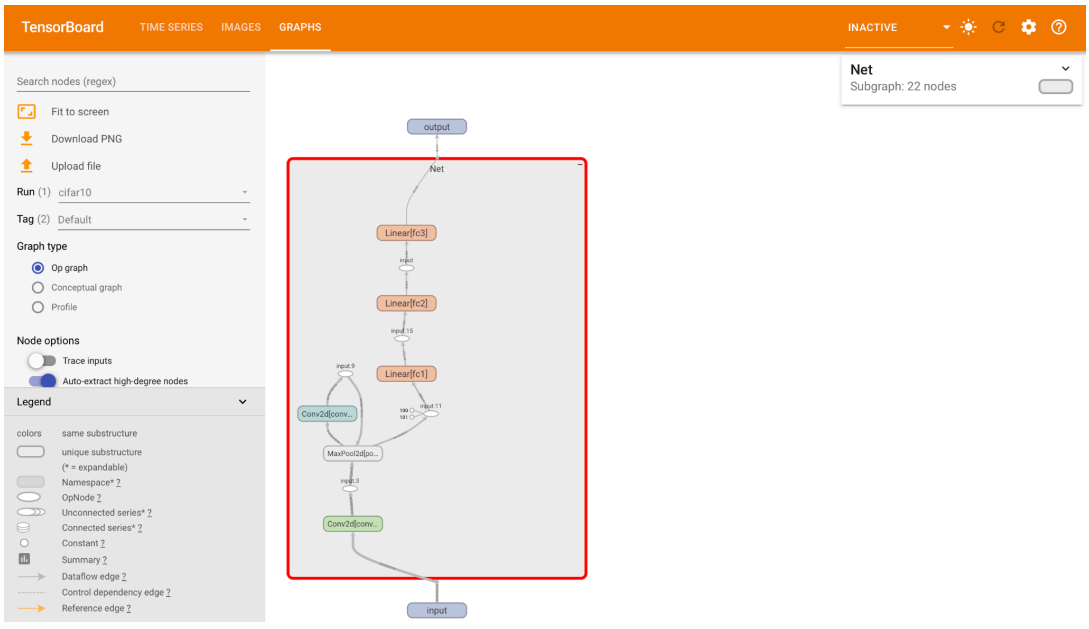
This is how the images are being displayed on the tensorboard.

Step 8: Visualize the model set up

Inspect the model : visualize the model built.

```
In [10]: writer.add_graph(net, images)
```

The writer.add_graph() function in TensorBoard is used to log the computational graph of a PyTorch model, allowing visualization of the model's architecture and its flow of data through various layers.



This is how the model is being displayed on the tensor board.

Step 9: Training the model and visualizing the training process

Helper Functions:

Helper Functions

- Produces predictions and their associated probabilities using the trained network and a list of images.
- Creates a Matplotlib Figure using a trained network, along with images and labels from a batch. This figure displays the network's top prediction and its probability, alongside the actual label, with the information color-coded to indicate whether the prediction was correct. Utilizes the "images_to_probs" function.

In [12]:
```
def images_to_probs(net, images):
    output = net(images)
    _, preds_tensor = torch.max(output, 1)
    preds = np.squeeze(preds_tensor.numpy())
    return preds, [F.softmax(el, dim=0)[i].item() for i, el in zip(preds, output)]


def plot_classes_preds(net, images, labels):
    preds, probs = images_to_probs(net, images)
    fig = plt.figure(figsize=(10,25))
    for idx in np.arange(4):
        ax = fig.add_subplot(1, 4, idx+1, xticks=[], yticks=[])
        matplotlib_imshow(images[idx], one_channel=False)
        ax.set_title("{0}, {1:.1f}%\n(label: {2})".format(
            classes[preds[idx]],
            probs[idx] * 100.0,
            classes[labels[idx]]),
                    color=("green" if preds[idx]==labels[idx].item() else "red"))
    return fig
```

i. **images_to_probs(net, images)** : function takes a neural network (net) and a batch of images (images), and returns the predicted class indices and their corresponding probabilities.

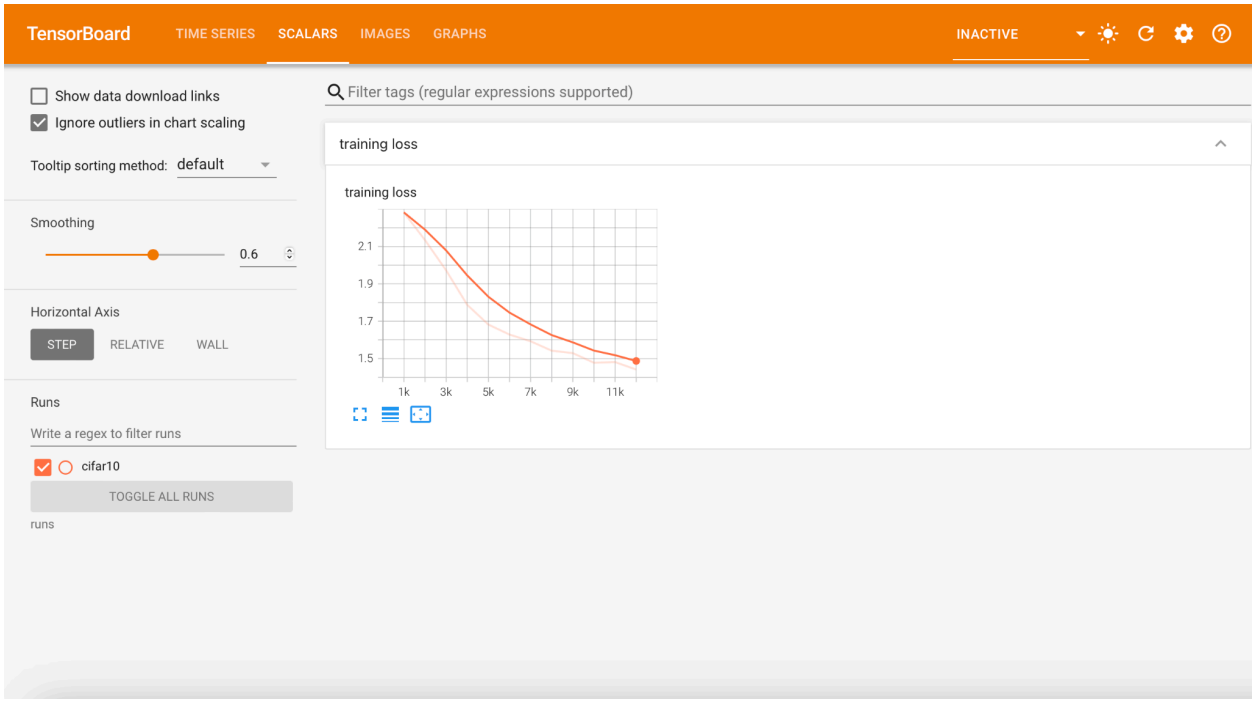
ii. **plot_classes_preds(net, images, labels)** : function visualizes the predictions of the neural network on a batch of images, showing the predicted class, the probability, and the true label.Each subplot displays the image, the predicted class with its probability, and the true label, coloring the title green if the prediction is correct and red if incorrect.

Train the neural network for one epoch, logging the training loss and a figure comparing predictions to actual labels every 1000 batches.

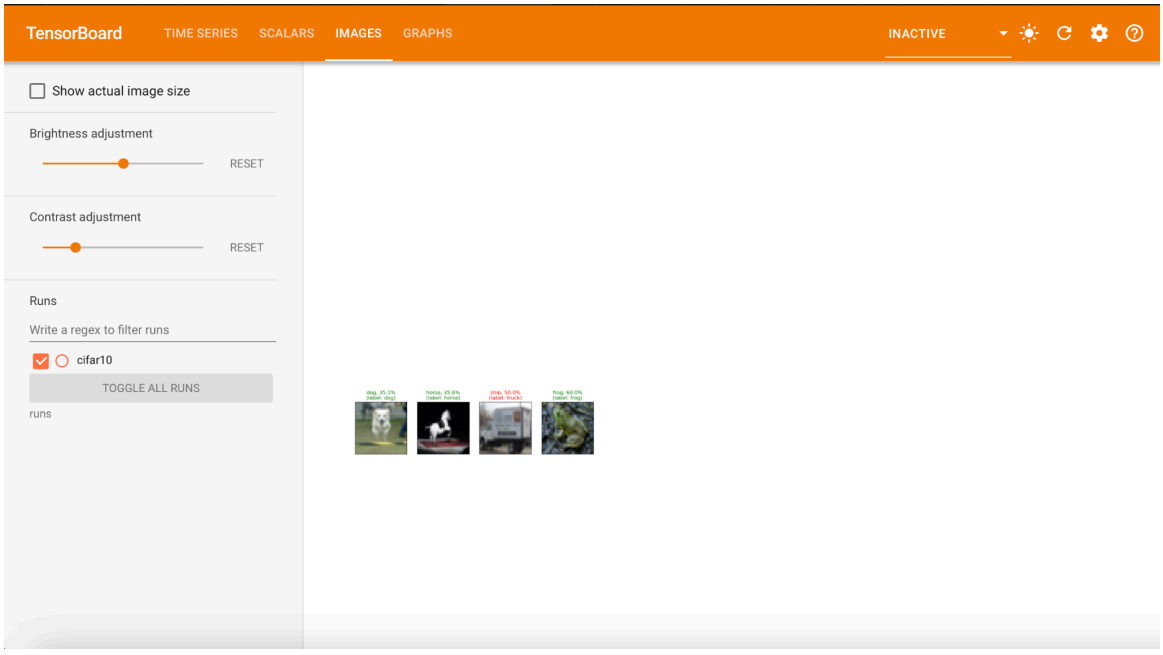
In [13]:
```
running_loss = 0.0
for epoch in range(1):

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 1000 == 999:
            writer.add_scalar('training loss',
                              running_loss / 1000,
                              epoch * len(trainloader) + i)
            writer.add_figure('predictions vs. actuals',
                              plot_classes_preds(net, inputs, labels),
                              global_step=epoch * len(trainloader) + i)
            running_loss = 0.0
print('Finished Training')
```

```
Finished Training
```

Using the above mentioned helper function the training of neural network is done.
The for loop iterates over the data loader for one epoch, processes each mini-batch by computing forward and backward passes, and updates the model parameters with the optimizer. It accumulates the loss for every 1000 mini-batches and logs the average loss and predictions versus actual labels using TensorBoard's writer.add_scalar() and writer.add_figure()respectively. After finishing the epoch, it resets the running loss and prints a completion message. This setup helps in monitoring training progress and visualizing the model's performance.



This is how the training loss graph is being displayed on the tensorboard.



This is how we can check the test predictions on the trained model ; what is the actual label and what the model is predicting.

Step 9: Generating the PR Curves

Precision-Recall (PR) curves in TensorBoard are graphical representations that show the trade-off between precision and recall for different threshold values of a classification model.

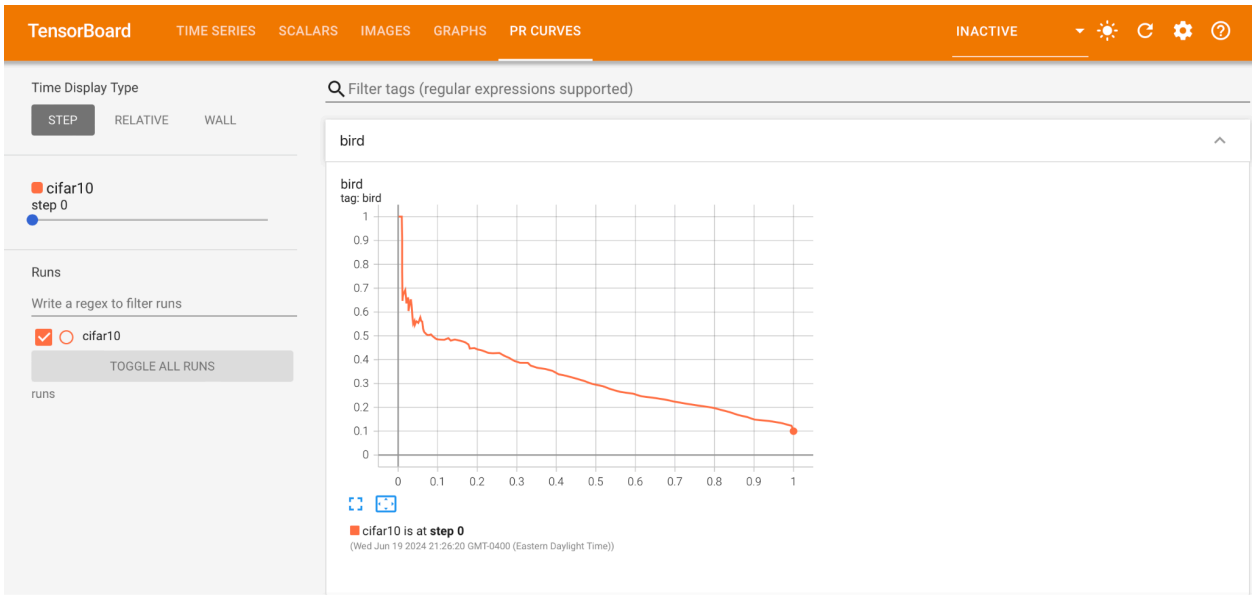
Creating the PR (Precision Recall) Curves for every category that is to be predicted using the helper function 'add_pr_curve_tensorboard'.

In [15]:
```
class_probs = []
class_label = []
with torch.no_grad():
    for data in testloader:
        images, labels = data
        output = net(images)
        class_probs_batch = [F.softmax(el, dim=0) for el in output]

        class_probs.append(class_probs_batch)
        class_label.append(labels)

test_probs = torch.cat([torch.stack(batch) for batch in class_probs])
test_label = torch.cat(class_label)

# helper function
def add_pr_curve_tensorboard(class_index, test_probs, test_label, global_step=0):
    tensorboard_truth = test_label == class_index
    tensorboard_probs = test_probs[:, class_index]

    writer.add_pr_curve(classes[class_index],
                        tensorboard_truth,
                        tensorboard_probs,
                        global_step=global_step)
    writer.close()

for i in range(len(classes)):
    add_pr_curve_tensorboard(i, test_probs, test_label)
```

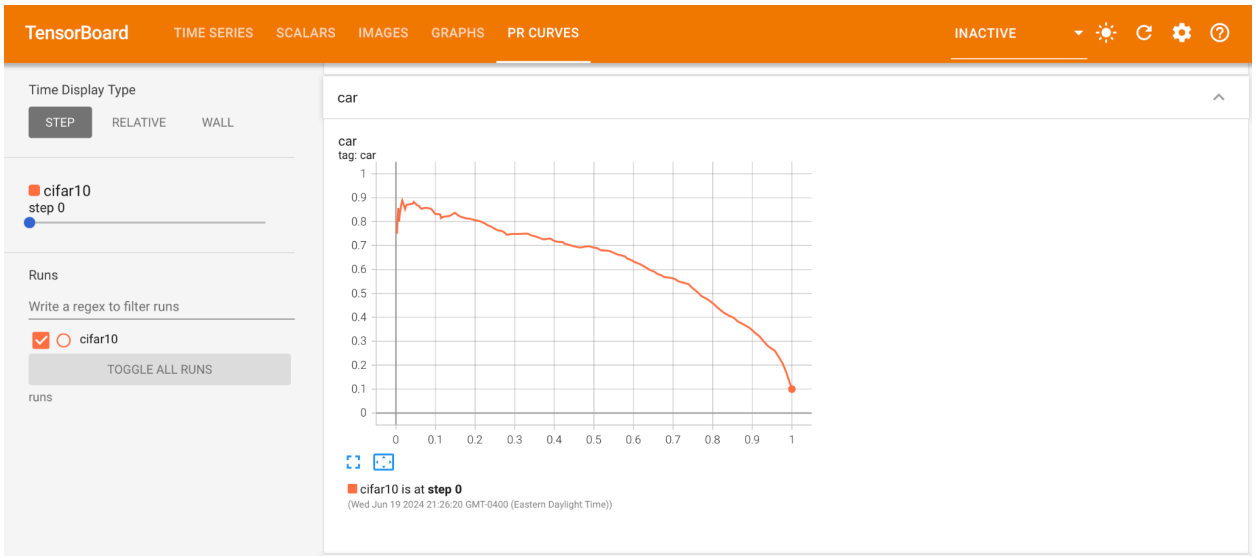
Helper Function :

i. add_pr_curve_tensorboard(class_index, test_probs, test_label, global_step=0) : function logs a Precision-Recall (PR) curve to TensorBoard for a specific class. It takes the class index, predicted probabilities (test_probs), true labels (test_label), and an optional global_step to indicate the training step. It calculates the truth values for the specified class and extracts the corresponding predicted probabilities, then uses writer.add_pr_curve() to visualize the PR curve for that class, helping to monitor and evaluate the model's performance on that particular class.

Then we test the model by using test dataset to obtain predicted class probabilities using the trained model . It iterates over the test data loader, computes the softmax probabilities for the network's output for each image batch, and stores them in class_probs. Simultaneously, the true labels are collected in class_label. After processing all batches, it concatenates the list of probability tensors and the list of label tensors into single tensors, test_probs and test_label, respectively, using torch.cat(). This prepares the data for further analysis, used for evaluating the model's performance with Precision-Recall curves.



This is how the PR curve looks for the class bird on the tensorboard.



This is how the PR curve looks for the class car on the tensorboard. And similarly the PR Curve for every class can be viewed on the tensorboard.

REFERENCES:

- Visualizing Models, Data, and Training with TensorBoard — PyTorch Tutorials 2.3.0+cu121 documentation
- Training a Classifier — PyTorch Tutorials 2.3.0+cu121 documentation
- PyTorch Tutorial 16 - How To Use The TensorBoar
- https://scikit-learn.org/stable/modules/neural_networks_supervised.html
- https://pytorch.org/tutorials/beginner/saving_loading_models.html