Optal_Presentation_by_Dristanta_Das

September 30, 2021

1 Name: Dristanta Das

1.1 The CIFAR 100 Dataset

This dataset is just like the CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs). Here is the list of classes in the CIFAR-100:

Superclass	Classes
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

[3]:
```
import torch
import numpy as np

# check if CUDA is available
```

```
train_on_gpu = torch.cuda.is_available()

if not train_on_gpu:
    print('CUDA is not available.  Training on CPU ...')
else:
    print('CUDA is available!  Training on GPU ...')
```

```
CUDA is available!  Training on GPU …
```

1.2 Data Download

Data is downloaded and the data is split into train-test-valid.

[4]:
```
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20
# percentage of training set to use as validation
valid_size = 0.2

# convert data to a normalized torch.FloatTensor
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

# choose the training and test datasets
train_data = datasets.CIFAR100('data', train=True,
                              download=True, transform=transform)
test_data = datasets.CIFAR100('data', train=False,
                             download=True, transform=transform)

# obtain training indices that will be used for validation
num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders (combine dataset and sampler)
```

```
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    sampler=train_sampler, num_workers=num_workers)
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    sampler=valid_sampler, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
    num_workers=num_workers)
```

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

[5]:
```
classes=train_data.classes
```

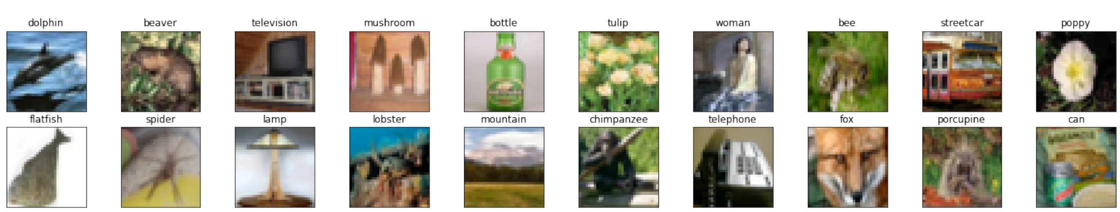
[ ]:
```

```

[6]:
```
import matplotlib.pyplot as plt
%matplotlib inline

# helper function to un-normalize and display an image
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    plt.imshow(np.transpose(img, (1, 2, 0)))  # convert from Tensor image
```

1.2.1 Some Visualisation of data

[7]:
```
# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy() # convert images to numpy for display

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
# display 20 images
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    imshow(images[idx])
    ax.set_title(classes[labels[idx]])
```

```
<ipython-input-7-2181f8df30a5>:10: MatplotlibDeprecationWarning: Passing non-
integers as three-element position specification is deprecated since 3.3 and
will be removed two minor releases later.
  ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
```



[ ]:

```

```

1.2.2 Defining CIFAR ResNet impementations from https://arxiv.org/abs/1512.03385

[8]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as init

from torch.autograd import Variable

def _weights_init(m):
    classname = m.__class__.__name__
    #print(classname)
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        init.kaiming_normal_(m.weight)

class LambdaLayer(nn.Module):
    def __init__(self, lambd):
        super(LambdaLayer, self).__init__()
        self.lambd = lambd

    def forward(self, x):
        return self.lambd(x)


class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1, option='A'):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride,␣
 ↪padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1,␣
 ↪padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != planes:
            if option == 'A':
                """
                For CIFAR10 ResNet paper uses option A.
```

```
            """
            self.shortcut = LambdaLayer(lambda x:
                                        F.pad(x[:, :, ::2, ::2], (0, 0, 0,␣
↪0, planes//4, planes//4), "constant", 0))
        elif option == 'B':
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion * planes,␣
↪kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion * planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=100):
        super(ResNet, self).__init__()
        self.in_planes = 16

        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1,␣
↪bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.layer1 = self._make_layer(block, 16, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 32, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 64, num_blocks[2], stride=2)
        self.linear = nn.Linear(64, num_classes)

        self.apply(_weights_init)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion

        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
```

```
        out = self.layer3(out)
        out = F.avg_pool2d(out, out.size()[3])
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out


def resnet20(num_classes=100):
    return ResNet(BasicBlock, [3, 3, 3], num_classes=num_classes)
```

[1]:
```
import torch
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

device cuda:0

[ ]:
```

```

[32]:
```
learning_rate = 0.1
weight_decay = 1e-4

model = resnet20(num_classes=100)
```

[33]:
```
import torch.optim as optim
```

[34]:
```
# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer
optimizer_sgd = optim.SGD(model.parameters(), lr=0.01)
optimizer_adam = optim.Adam(model.parameters(), lr=0.01)
optimizer_rmsprop = optim.RMSprop(model.parameters(), lr=0.01)
```

[35]:
```
model=model.cuda()
```

Stochastic Gradient Descent

Stochastic Gradient Descent is an iterative optimization technique that uses minibatches of data to form an expectation of the gradient, rather than the full gradient using all available data. That is for weights w and a loss function L we have:

$$w_{t+1} = w_t - \eta \hat{\nabla}_w L(w_t)$$

Where η is a learning rate. SGD reduces redundancy compared to batch gradient descent - which recomputes gradients for similar examples before each parameter update - so it is usually much faster.

[15]:
```
# number of epochs to train the model
n_epochs = 30

valid_loss_min = np.Inf # track change in validation loss

train_loss_list = []
valid_loss_list = []

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train()
    for data, target in train_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer_sgd.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model␣
↪parameters
```

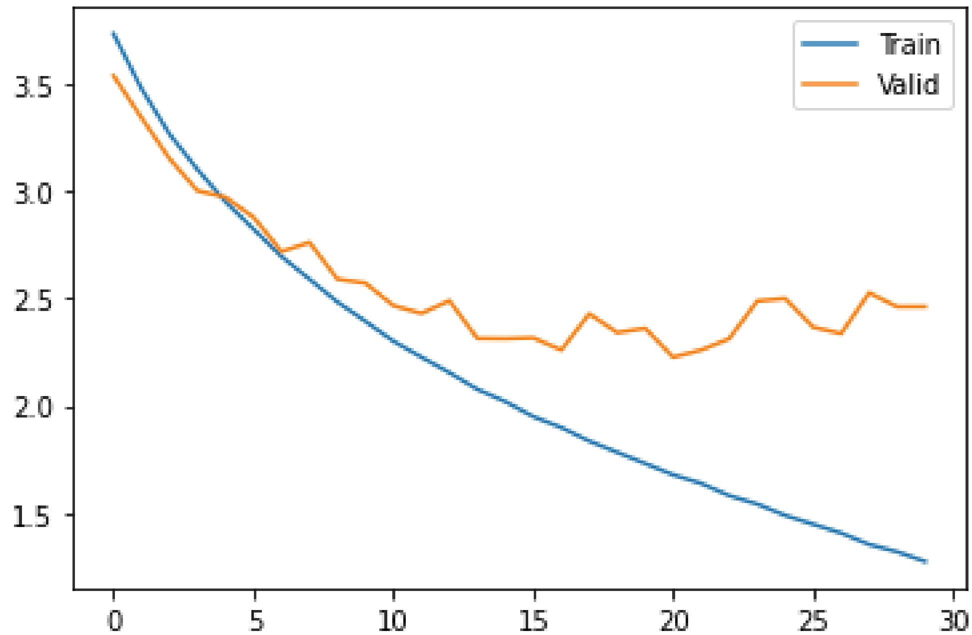
```
        loss.backward()
        #print("conv1 grads",torch.linalg.norm(model.conv1.weight.grad))
        #print("conv2 grads",torch.linalg.norm(model.conv2.bias.grad))
        # perform a single optimization step (parameter update)
        optimizer_sgd.step()
        # update training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    model.eval()
    for data, target in valid_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += loss.item()*data.size(0)

    # calculate average losses
    train_loss = train_loss/len(train_loader.sampler)
    valid_loss = valid_loss/len(valid_loader.sampler)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))
    train_loss_list.append(train_loss)
    valid_loss_list.append(valid_loss)

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
↪'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model_cifar.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 3.730502         Validation Loss: 3.536027
Validation loss decreased (inf --> 3.536027).  Saving model …
Epoch: 2        Training Loss: 3.475741         Validation Loss: 3.340418
Validation loss decreased (3.536027 --> 3.340418).  Saving model …
Epoch: 3        Training Loss: 3.264459         Validation Loss: 3.149833
```

```
Validation loss decreased (3.340418 --> 3.149833).  Saving model …
Epoch: 4        Training Loss: 3.099424         Validation Loss: 3.000438
Validation loss decreased (3.149833 --> 3.000438).  Saving model …
Epoch: 5        Training Loss: 2.949579         Validation Loss: 2.970573
Validation loss decreased (3.000438 --> 2.970573).  Saving model …
Epoch: 6        Training Loss: 2.821437         Validation Loss: 2.876893
Validation loss decreased (2.970573 --> 2.876893).  Saving model …
Epoch: 7        Training Loss: 2.694921         Validation Loss: 2.717911
Validation loss decreased (2.876893 --> 2.717911).  Saving model …
Epoch: 8        Training Loss: 2.589296         Validation Loss: 2.760667
Epoch: 9        Training Loss: 2.482899         Validation Loss: 2.589457
Validation loss decreased (2.717911 --> 2.589457).  Saving model …
Epoch: 10       Training Loss: 2.393062         Validation Loss: 2.570993
Validation loss decreased (2.589457 --> 2.570993).  Saving model …
Epoch: 11       Training Loss: 2.300458         Validation Loss: 2.466298
Validation loss decreased (2.570993 --> 2.466298).  Saving model …
Epoch: 12       Training Loss: 2.226197         Validation Loss: 2.429883
Validation loss decreased (2.466298 --> 2.429883).  Saving model …
Epoch: 13       Training Loss: 2.154225         Validation Loss: 2.488985
Epoch: 14       Training Loss: 2.076333         Validation Loss: 2.313256
Validation loss decreased (2.429883 --> 2.313256).  Saving model …
Epoch: 15       Training Loss: 2.019355         Validation Loss: 2.312177
Validation loss decreased (2.313256 --> 2.312177).  Saving model …
Epoch: 16       Training Loss: 1.950035         Validation Loss: 2.316027
Epoch: 17       Training Loss: 1.899267         Validation Loss: 2.259547
Validation loss decreased (2.312177 --> 2.259547).  Saving model …
Epoch: 18       Training Loss: 1.836076         Validation Loss: 2.427947
Epoch: 19       Training Loss: 1.783857         Validation Loss: 2.340590
Epoch: 20       Training Loss: 1.731655         Validation Loss: 2.360837
Epoch: 21       Training Loss: 1.678798         Validation Loss: 2.225923
Validation loss decreased (2.259547 --> 2.225923).  Saving model …
Epoch: 22       Training Loss: 1.639128         Validation Loss: 2.259225
Epoch: 23       Training Loss: 1.582162         Validation Loss: 2.311805
Epoch: 24       Training Loss: 1.542752         Validation Loss: 2.487813
Epoch: 25       Training Loss: 1.489751         Validation Loss: 2.499257
Epoch: 26       Training Loss: 1.448778         Validation Loss: 2.365676
Epoch: 27       Training Loss: 1.407711         Validation Loss: 2.336084
Epoch: 28       Training Loss: 1.354528         Validation Loss: 2.525634
Epoch: 29       Training Loss: 1.320424         Validation Loss: 2.460857
Epoch: 30       Training Loss: 1.275501         Validation Loss: 2.461306
```

[18]:
```
plt.plot(train_loss_list)
plt.plot(valid_loss_list)
plt.legend(["Train","Valid"])
```

[18]:
```
<matplotlib.legend.Legend at 0x7f7270091a30>
```


Train
Valid
3.5
3.0
2.5
2.0
1.5
0
5
10
15
20
25
30

[ ]:

1.3 ADAM

Adam is an adaptive learning rate optimization algorithm that utilises both momentum and scaling, combining the benefits of RMSProp and SGD w/th Momentum. The optimizer is designed to be appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients.

The weight updates are performed as:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

with

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

η is the step size/learning rate, around 1e-3 in the original paper. ϵ is a small number, typically 1e-8 or 1e-10, to prevent dividing by zero. β_1 and β_2 are forgetting parameters, with typical values 0.9 and 0.999, respectively.

[30]:
```
# number of epochs to train the model
n_epochs = 30

valid_loss_min = np.Inf # track change in validation loss

train_loss_list1 = []
valid_loss_list1 = []

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train()
    for data, target in train_loader:
```

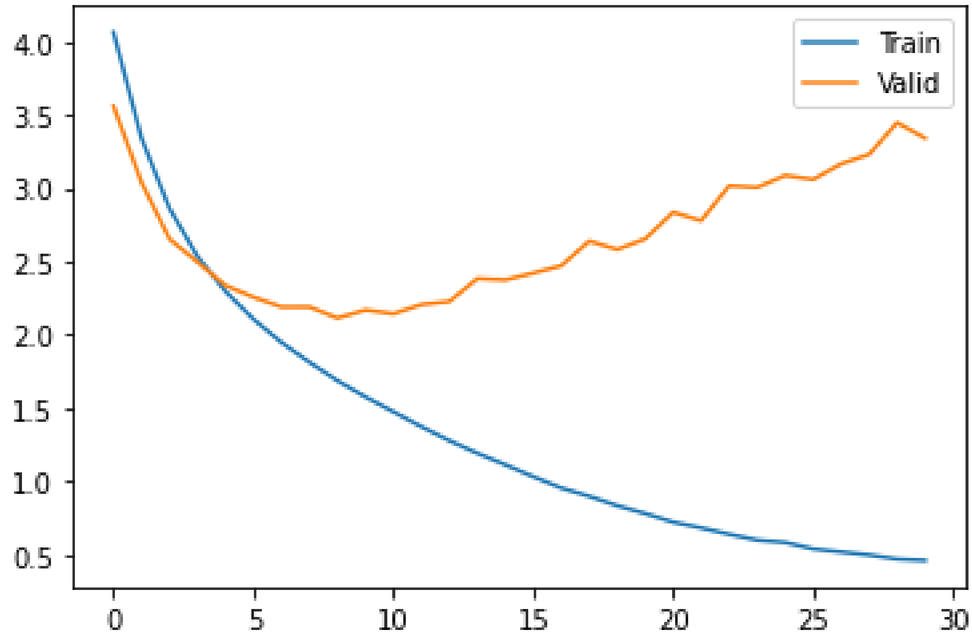
```
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer_adam.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model␣
↪parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer_adam.step()
        # update training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    model.eval()
    for data, target in valid_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += loss.item()*data.size(0)

    # calculate average losses
    train_loss = train_loss/len(train_loader.sampler)
    valid_loss = valid_loss/len(valid_loader.sampler)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))
    train_loss_list1.append(train_loss)
    valid_loss_list1.append(valid_loss)

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
↪'.format(
        valid_loss_min,
```

```
        valid_loss))
        torch.save(model.state_dict(), 'model_cifar_adam.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 4.062683         Validation Loss: 3.560229
Validation loss decreased (inf --> 3.560229).  Saving model …
Epoch: 2        Training Loss: 3.340821         Validation Loss: 3.039103
Validation loss decreased (3.560229 --> 3.039103).  Saving model …
Epoch: 3        Training Loss: 2.861335         Validation Loss: 2.653627
Validation loss decreased (3.039103 --> 2.653627).  Saving model …
Epoch: 4        Training Loss: 2.535089         Validation Loss: 2.497058
Validation loss decreased (2.653627 --> 2.497058).  Saving model …
Epoch: 5        Training Loss: 2.297021         Validation Loss: 2.336595
Validation loss decreased (2.497058 --> 2.336595).  Saving model …
Epoch: 6        Training Loss: 2.105815         Validation Loss: 2.256637
Validation loss decreased (2.336595 --> 2.256637).  Saving model …
Epoch: 7        Training Loss: 1.947464         Validation Loss: 2.190325
Validation loss decreased (2.256637 --> 2.190325).  Saving model …
Epoch: 8        Training Loss: 1.812029         Validation Loss: 2.190153
Validation loss decreased (2.190325 --> 2.190153).  Saving model …
Epoch: 9        Training Loss: 1.686480         Validation Loss: 2.114414
Validation loss decreased (2.190153 --> 2.114414).  Saving model …
Epoch: 10       Training Loss: 1.575623         Validation Loss: 2.169182
Epoch: 11       Training Loss: 1.475130         Validation Loss: 2.143641
Epoch: 12       Training Loss: 1.373631         Validation Loss: 2.206606
Epoch: 13       Training Loss: 1.278473         Validation Loss: 2.227270
Epoch: 14       Training Loss: 1.192002         Validation Loss: 2.383638
Epoch: 15       Training Loss: 1.113892         Validation Loss: 2.374428
Epoch: 16       Training Loss: 1.031813         Validation Loss: 2.421315
Epoch: 17       Training Loss: 0.954969         Validation Loss: 2.471868
Epoch: 18       Training Loss: 0.898275         Validation Loss: 2.639445
Epoch: 19       Training Loss: 0.834255         Validation Loss: 2.582749
Epoch: 20       Training Loss: 0.780663         Validation Loss: 2.653668
Epoch: 21       Training Loss: 0.721548         Validation Loss: 2.835197
Epoch: 22       Training Loss: 0.682648         Validation Loss: 2.778057
Epoch: 23       Training Loss: 0.638971         Validation Loss: 3.015733
Epoch: 24       Training Loss: 0.598994         Validation Loss: 3.005172
Epoch: 25       Training Loss: 0.583138         Validation Loss: 3.086495
Epoch: 26       Training Loss: 0.538935         Validation Loss: 3.059574
Epoch: 27       Training Loss: 0.518013         Validation Loss: 3.165812
Epoch: 28       Training Loss: 0.497538         Validation Loss: 3.230744
Epoch: 29       Training Loss: 0.469450         Validation Loss: 3.446984
Epoch: 30       Training Loss: 0.460621         Validation Loss: 3.341339
```

[31]:
```
plt.plot(train_loss_list1)
plt.plot(valid_loss_list1)
plt.legend(["Train","Valid"])
```

[31]: <matplotlib.legend.Legend at 0x7f725c383a30>



[ ]:

```

```

RMSProp

RMSProp is an unpublished adaptive learning rate optimizer proposed by Geoff Hinton. The motivation is that the magnitude of gradients can differ for different weights, and can change during learning, making it hard to choose a single global learning rate. RMSProp tackles this by keeping a moving average of the squared gradient and adjusting the weight updates by this magnitude. The gradient updates are performed as:

$$E\left[g^2\right]_t = \gamma E\left[g^2\right]_{t-1} + (1-\gamma)g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Hinton suggests $\gamma = 0.9$, with a good default for η as 0.001.

[36]:

```
# number of epochs to train the model
n_epochs = 30
```

```
valid_loss_min = np.Inf # track change in validation loss

train_loss_list2 = []
valid_loss_list2 = []

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train()
    for data, target in train_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer_rmsprop.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model␣
↪parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer_rmsprop.step()
        # update training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    model.eval()
    for data, target in valid_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
```

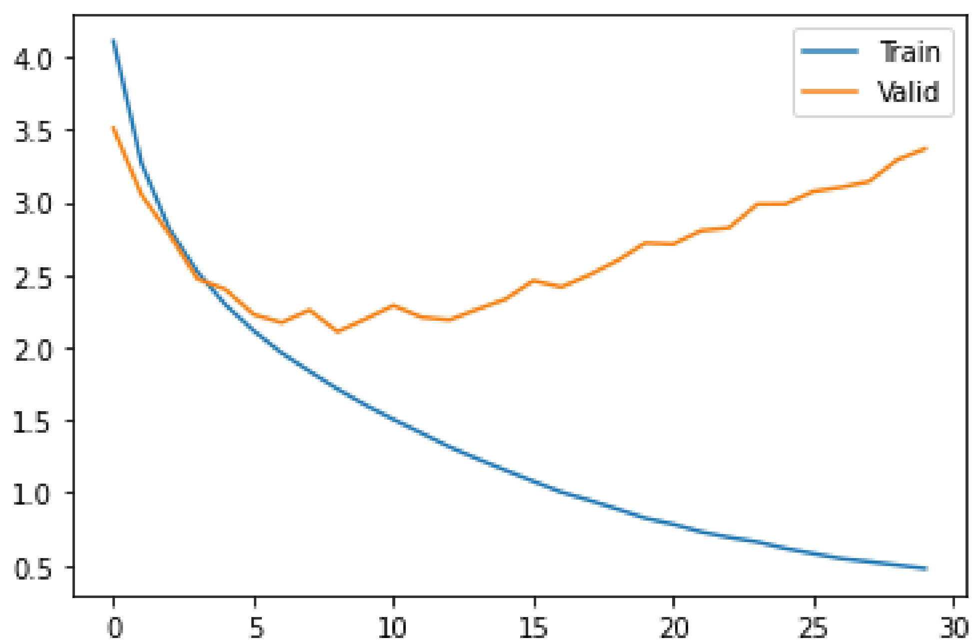
```
        valid_loss += loss.item()*data.size(0)

    # calculate average losses
    train_loss = train_loss/len(train_loader.sampler)
    valid_loss = valid_loss/len(valid_loader.sampler)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))
    train_loss_list2.append(train_loss)
    valid_loss_list2.append(valid_loss)

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
↪'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model_cifar_rmsprop.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 4.106680         Validation Loss: 3.507062
Validation loss decreased (inf --> 3.507062).  Saving model …
Epoch: 2        Training Loss: 3.266501         Validation Loss: 3.049953
Validation loss decreased (3.507062 --> 3.049953).  Saving model …
Epoch: 3        Training Loss: 2.812668         Validation Loss: 2.776028
Validation loss decreased (3.049953 --> 2.776028).  Saving model …
Epoch: 4        Training Loss: 2.519789         Validation Loss: 2.471569
Validation loss decreased (2.776028 --> 2.471569).  Saving model …
Epoch: 5        Training Loss: 2.294034         Validation Loss: 2.399516
Validation loss decreased (2.471569 --> 2.399516).  Saving model …
Epoch: 6        Training Loss: 2.114641         Validation Loss: 2.226972
Validation loss decreased (2.399516 --> 2.226972).  Saving model …
Epoch: 7        Training Loss: 1.961950         Validation Loss: 2.170414
Validation loss decreased (2.226972 --> 2.170414).  Saving model …
Epoch: 8        Training Loss: 1.835052         Validation Loss: 2.257077
Epoch: 9        Training Loss: 1.712741         Validation Loss: 2.107403
Validation loss decreased (2.170414 --> 2.107403).  Saving model …
Epoch: 10       Training Loss: 1.602155         Validation Loss: 2.195206
Epoch: 11       Training Loss: 1.504186         Validation Loss: 2.288133
Epoch: 12       Training Loss: 1.410837         Validation Loss: 2.207827
Epoch: 13       Training Loss: 1.315182         Validation Loss: 2.188577
Epoch: 14       Training Loss: 1.232923         Validation Loss: 2.262066
Epoch: 15       Training Loss: 1.154228         Validation Loss: 2.331305
Epoch: 16       Training Loss: 1.078158         Validation Loss: 2.459516
Epoch: 17       Training Loss: 1.003492         Validation Loss: 2.415298
Epoch: 18       Training Loss: 0.947661         Validation Loss: 2.498109
```

```
Epoch: 19       Training Loss: 0.887071        Validation Loss: 2.597153
Epoch: 20       Training Loss: 0.824049        Validation Loss: 2.717873
Epoch: 21       Training Loss: 0.782002        Validation Loss: 2.710188
Epoch: 22       Training Loss: 0.729545        Validation Loss: 2.802959
Epoch: 23       Training Loss: 0.691687        Validation Loss: 2.822530
Epoch: 24       Training Loss: 0.660510        Validation Loss: 2.985001
Epoch: 25       Training Loss: 0.615522        Validation Loss: 2.986121
Epoch: 26       Training Loss: 0.579927        Validation Loss: 3.072915
Epoch: 27       Training Loss: 0.545401        Validation Loss: 3.102075
Epoch: 28       Training Loss: 0.523845        Validation Loss: 3.141438
Epoch: 29       Training Loss: 0.500624        Validation Loss: 3.290888
Epoch: 30       Training Loss: 0.477116        Validation Loss: 3.367173
```

[37]:
```
plt.plot(train_loss_list2)
plt.plot(valid_loss_list2)
plt.legend(["Train","Valid"])
```

[37]: <matplotlib.legend.Legend at 0x7f725c076cd0>

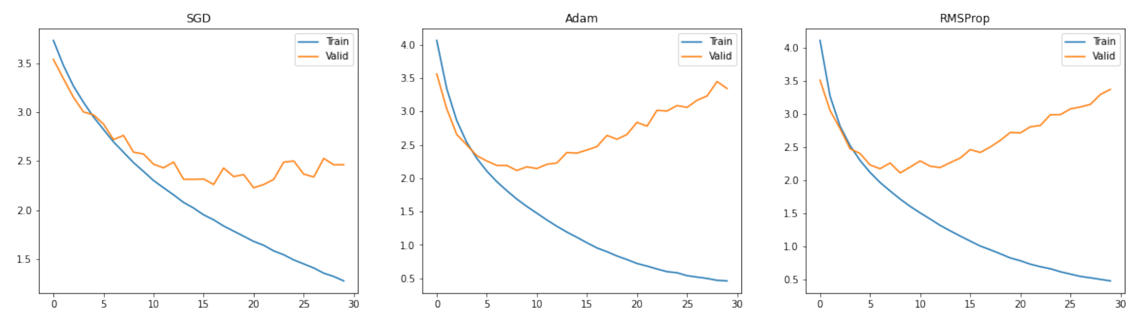


[ ]:

[38]:
```
plt.figure(figsize=(20,5))

plt.subplot(131)
plt.plot(train_loss_list)
plt.plot(valid_loss_list)
```

```
plt.legend(["Train","Valid"])
plt.title('SGD')
plt.subplot(132)
plt.plot(train_loss_list1)
plt.plot(valid_loss_list1)
plt.legend(["Train","Valid"])
plt.title('Adam')
plt.subplot(133)
plt.plot(train_loss_list2)
plt.plot(valid_loss_list2)
plt.legend(["Train","Valid"])
plt.title('RMSProp')
```

[38]: Text(0.5, 1.0, 'RMSProp')



[ ]:

[ ]:

1.4 Testing the Trained Network

1.5 SGD

[54]:
```
## SGD

model.load_state_dict(torch.load('model_cifar.pt'))


# track test loss
test_loss = 0.0
class_correct = list(0. for i in range(100))
class_total = list(0. for i in range(100))

model.eval()
```

```python
# iterate over test data
for data, target in test_loader:
    # move tensors to GPU if CUDA is available
    if train_on_gpu:
        data, target = data.cuda(), target.cuda()
    # forward pass: compute predicted outputs by passing inputs to the model
    output = model(data)
    # calculate the batch loss
    loss = criterion(output, target)
    # update test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
    # compare predictions to true label
    correct_tensor = pred.eq(target.data.view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu else np.
 ↪squeeze(correct_tensor.cpu().numpy())
    # calculate test accuracy for each object class
    for i in range(20):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# average test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))

for i in range(100):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            classes[i], 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))


#print(np.sum(class_correct))
print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

```
Test Loss: 2.195332

Test Accuracy of apple: 74% (74/100)
Test Accuracy of aquarium_fish: 60% (60/100)
Test Accuracy of  baby: 32% (32/100)
Test Accuracy of  bear: 15% (15/100)
```

```
Test Accuracy of beaver: 25% (25/100)
Test Accuracy of   bed: 45% (45/100)
Test Accuracy of   bee: 29% (29/100)
Test Accuracy of beetle: 26% (26/100)
Test Accuracy of bicycle: 46% (46/100)
Test Accuracy of bottle: 46% (46/100)
Test Accuracy of  bowl: 30% (30/100)
Test Accuracy of   boy: 41% (41/100)
Test Accuracy of bridge: 48% (48/100)
Test Accuracy of   bus: 59% (59/100)
Test Accuracy of butterfly: 14% (14/100)
Test Accuracy of camel: 29% (29/100)
Test Accuracy of   can: 57% (57/100)
Test Accuracy of castle: 74% (74/100)
Test Accuracy of caterpillar: 25% (25/100)
Test Accuracy of cattle: 29% (29/100)
Test Accuracy of chair: 56% (56/100)
Test Accuracy of chimpanzee: 75% (75/100)
Test Accuracy of clock: 40% (40/100)
Test Accuracy of cloud: 68% (68/100)
Test Accuracy of cockroach: 55% (55/100)
Test Accuracy of couch: 40% (40/100)
Test Accuracy of  crab: 37% (37/100)
Test Accuracy of crocodile: 36% (36/100)
Test Accuracy of   cup: 64% (64/100)
Test Accuracy of dinosaur: 41% (41/100)
Test Accuracy of dolphin: 40% (40/100)
Test Accuracy of elephant: 43% (43/100)
Test Accuracy of flatfish: 28% (28/100)
Test Accuracy of forest: 44% (44/100)
Test Accuracy of   fox: 31% (31/100)
Test Accuracy of  girl:  7% ( 7/100)
Test Accuracy of hamster: 49% (49/100)
Test Accuracy of house: 31% (31/100)
Test Accuracy of kangaroo: 34% (34/100)
Test Accuracy of keyboard: 60% (60/100)
Test Accuracy of  lamp: 26% (26/100)
Test Accuracy of lawn_mower: 68% (68/100)
Test Accuracy of leopard: 35% (35/100)
Test Accuracy of  lion: 38% (38/100)
Test Accuracy of lizard: 10% (10/100)
Test Accuracy of lobster:  8% ( 8/100)
Test Accuracy of   man: 22% (22/100)
Test Accuracy of maple_tree: 71% (71/100)
Test Accuracy of motorcycle: 65% (65/100)
Test Accuracy of mountain: 60% (60/100)
Test Accuracy of mouse: 22% (22/100)
Test Accuracy of mushroom: 49% (49/100)
```

```
Test Accuracy of oak_tree: 52% (52/100)
Test Accuracy of orange: 74% (74/100)
Test Accuracy of orchid: 63% (63/100)
Test Accuracy of otter:  1% ( 1/100)
Test Accuracy of palm_tree: 57% (57/100)
Test Accuracy of  pear: 32% (32/100)
Test Accuracy of pickup_truck: 43% (43/100)
Test Accuracy of pine_tree: 24% (24/100)
Test Accuracy of plain: 84% (84/100)
Test Accuracy of plate: 38% (38/100)
Test Accuracy of poppy: 56% (56/100)
Test Accuracy of porcupine: 38% (38/100)
Test Accuracy of possum: 30% (30/100)
Test Accuracy of rabbit: 16% (16/100)
Test Accuracy of raccoon: 30% (30/100)
Test Accuracy of   ray: 26% (26/100)
Test Accuracy of  road: 78% (78/100)
Test Accuracy of rocket: 58% (58/100)
Test Accuracy of  rose: 57% (57/100)
Test Accuracy of   sea: 61% (61/100)
Test Accuracy of  seal: 13% (13/100)
Test Accuracy of shark: 50% (50/100)
Test Accuracy of shrew: 26% (26/100)
Test Accuracy of skunk: 75% (75/100)
Test Accuracy of skyscraper: 73% (73/100)
Test Accuracy of snail: 23% (23/100)
Test Accuracy of snake: 27% (27/100)
Test Accuracy of spider: 24% (24/100)
Test Accuracy of squirrel:  6% ( 6/100)
Test Accuracy of streetcar: 40% (40/100)
Test Accuracy of sunflower: 61% (61/100)
Test Accuracy of sweet_pepper: 19% (19/100)
Test Accuracy of table: 31% (31/100)
Test Accuracy of  tank: 43% (43/100)
Test Accuracy of telephone: 48% (48/100)
Test Accuracy of television: 52% (52/100)
Test Accuracy of tiger: 43% (43/100)
Test Accuracy of tractor: 48% (48/100)
Test Accuracy of train: 45% (45/100)
Test Accuracy of trout: 50% (50/100)
Test Accuracy of tulip: 34% (34/100)
Test Accuracy of turtle: 19% (19/100)
Test Accuracy of wardrobe: 91% (91/100)
Test Accuracy of whale: 50% (50/100)
Test Accuracy of willow_tree: 37% (37/100)
Test Accuracy of  wolf: 33% (33/100)
Test Accuracy of woman: 26% (26/100)
Test Accuracy of  worm: 40% (40/100)
```

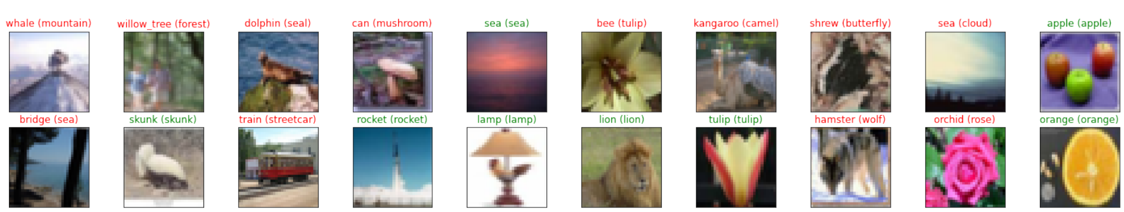
```
Test Accuracy (Overall): 42% (4202/10000)
```

1.5.1 Visualize Sample Test Results

[55]:

```python
# obtain one batch of test images
dataiter = iter(test_loader)
images, labels = dataiter.next()
images.numpy()

# move model inputs to cuda, if GPU available
if train_on_gpu:
    images = images.cuda()

# get sample outputs
output = model(images)
# convert output probabilities to predicted class
_, preds_tensor = torch.max(output, 1)
preds = np.squeeze(preds_tensor.numpy()) if not train_on_gpu else np.
 ↪squeeze(preds_tensor.cpu().numpy())

# plot the images in the batch, along with predicted and true labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    imshow(images[idx] if not train_on_gpu else images[idx].cpu())
    ax.set_title("{} ({})".format(classes[preds[idx]], classes[labels[idx]]),
                 color=("green" if preds[idx]==labels[idx].item() else "red"))
```

```
<ipython-input-55-a0724321e9b1>:19: MatplotlibDeprecationWarning: Passing non-
integers as three-element position specification is deprecated since 3.3 and
will be removed two minor releases later.
  ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
```



[ ]:

1.6 Adam

[56]:
```
## Adam

model.load_state_dict(torch.load('model_cifar_adam.pt'))


# track test loss
test_loss = 0.0
class_correct = list(0. for i in range(100))
class_total = list(0. for i in range(100))

model.eval()
# iterate over test data
for data, target in test_loader:
    # move tensors to GPU if CUDA is available
    if train_on_gpu:
        data, target = data.cuda(), target.cuda()
    # forward pass: compute predicted outputs by passing inputs to the model
    output = model(data)
    # calculate the batch loss
    loss = criterion(output, target)
    # update test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
    # compare predictions to true label
    correct_tensor = pred.eq(target.data.view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu else np.
 ↪squeeze(correct_tensor.cpu().numpy())
    # calculate test accuracy for each object class
    for i in range(20):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# average test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))

for i in range(100):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            classes[i], 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))
```

```python
#print(np.sum(class_correct))
print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

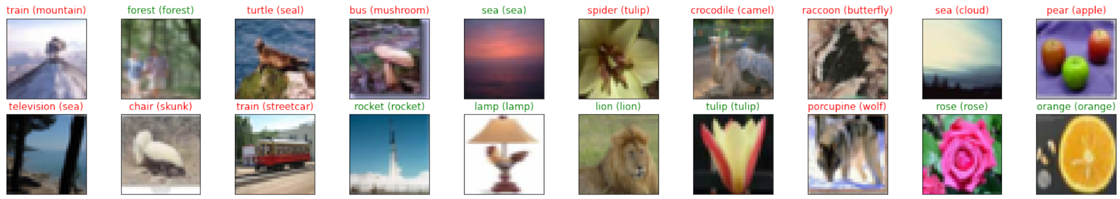
```
Test Loss: 2.066987

Test Accuracy of apple: 69% (69/100)
Test Accuracy of aquarium_fish: 64% (64/100)
Test Accuracy of  baby: 43% (43/100)
Test Accuracy of  bear: 23% (23/100)
Test Accuracy of beaver: 18% (18/100)
Test Accuracy of   bed: 44% (44/100)
Test Accuracy of   bee: 35% (35/100)
Test Accuracy of beetle: 62% (62/100)
Test Accuracy of bicycle: 59% (59/100)
Test Accuracy of bottle: 63% (63/100)
Test Accuracy of  bowl: 39% (39/100)
Test Accuracy of   boy:  7% ( 7/100)
Test Accuracy of bridge: 61% (61/100)
Test Accuracy of   bus: 40% (40/100)
Test Accuracy of butterfly: 30% (30/100)
Test Accuracy of camel: 31% (31/100)
Test Accuracy of   can: 58% (58/100)
Test Accuracy of castle: 59% (59/100)
Test Accuracy of caterpillar: 36% (36/100)
Test Accuracy of cattle: 31% (31/100)
Test Accuracy of chair: 80% (80/100)
Test Accuracy of chimpanzee: 68% (68/100)
Test Accuracy of clock: 18% (18/100)
Test Accuracy of cloud: 54% (54/100)
Test Accuracy of cockroach: 59% (59/100)
Test Accuracy of couch: 26% (26/100)
Test Accuracy of  crab: 24% (24/100)
Test Accuracy of crocodile: 18% (18/100)
Test Accuracy of   cup: 73% (73/100)
Test Accuracy of dinosaur: 40% (40/100)
Test Accuracy of dolphin: 40% (40/100)
Test Accuracy of elephant: 48% (48/100)
Test Accuracy of flatfish: 41% (41/100)
Test Accuracy of forest: 54% (54/100)
Test Accuracy of   fox: 43% (43/100)
Test Accuracy of  girl: 20% (20/100)
Test Accuracy of hamster: 48% (48/100)
Test Accuracy of house: 26% (26/100)
```

```
Test Accuracy of kangaroo: 36% (36/100)
Test Accuracy of keyboard: 64% (64/100)
Test Accuracy of  lamp: 41% (41/100)
Test Accuracy of lawn_mower: 70% (70/100)
Test Accuracy of leopard: 28% (28/100)
Test Accuracy of  lion: 31% (31/100)
Test Accuracy of lizard: 24% (24/100)
Test Accuracy of lobster: 26% (26/100)
Test Accuracy of   man: 55% (55/100)
Test Accuracy of maple_tree: 57% (57/100)
Test Accuracy of motorcycle: 76% (76/100)
Test Accuracy of mountain: 75% (75/100)
Test Accuracy of mouse: 20% (20/100)
Test Accuracy of mushroom: 34% (34/100)
Test Accuracy of oak_tree: 55% (55/100)
Test Accuracy of orange: 80% (80/100)
Test Accuracy of orchid: 56% (56/100)
Test Accuracy of otter:  6% ( 6/100)
Test Accuracy of palm_tree: 69% (69/100)
Test Accuracy of  pear: 56% (56/100)
Test Accuracy of pickup_truck: 58% (58/100)
Test Accuracy of pine_tree: 46% (46/100)
Test Accuracy of plain: 84% (84/100)
Test Accuracy of plate: 65% (65/100)
Test Accuracy of poppy: 60% (60/100)
Test Accuracy of porcupine: 49% (49/100)
Test Accuracy of possum: 14% (14/100)
Test Accuracy of rabbit: 18% (18/100)
Test Accuracy of raccoon: 46% (46/100)
Test Accuracy of   ray: 40% (40/100)
Test Accuracy of  road: 85% (85/100)
Test Accuracy of rocket: 66% (66/100)
Test Accuracy of  rose: 24% (24/100)
Test Accuracy of   sea: 67% (67/100)
Test Accuracy of  seal:  8% ( 8/100)
Test Accuracy of shark: 18% (18/100)
Test Accuracy of shrew: 28% (28/100)
Test Accuracy of skunk: 66% (66/100)
Test Accuracy of skyscraper: 61% (61/100)
Test Accuracy of snail: 28% (28/100)
Test Accuracy of snake: 44% (44/100)
Test Accuracy of spider: 40% (40/100)
Test Accuracy of squirrel: 26% (26/100)
Test Accuracy of streetcar: 36% (36/100)
Test Accuracy of sunflower: 75% (75/100)
Test Accuracy of sweet_pepper: 30% (30/100)
Test Accuracy of table: 29% (29/100)
Test Accuracy of  tank: 61% (61/100)
```

```
Test Accuracy of telephone: 49% (49/100)
Test Accuracy of television: 62% (62/100)
Test Accuracy of tiger: 34% (34/100)
Test Accuracy of tractor: 53% (53/100)
Test Accuracy of train: 65% (65/100)
Test Accuracy of trout: 53% (53/100)
Test Accuracy of tulip: 62% (62/100)
Test Accuracy of turtle: 17% (17/100)
Test Accuracy of wardrobe: 81% (81/100)
Test Accuracy of whale: 66% (66/100)
Test Accuracy of willow_tree: 30% (30/100)
Test Accuracy of  wolf: 47% (47/100)
Test Accuracy of woman:  6% ( 6/100)
Test Accuracy of  worm: 37% (37/100)

Test Accuracy (Overall): 45% (4545/10000)
```

[57]:
```
# obtain one batch of test images
dataiter = iter(test_loader)
images, labels = dataiter.next()
images.numpy()

# move model inputs to cuda, if GPU available
if train_on_gpu:
    images = images.cuda()

# get sample outputs
output = model(images)
# convert output probabilities to predicted class
_, preds_tensor = torch.max(output, 1)
preds = np.squeeze(preds_tensor.numpy()) if not train_on_gpu else np.
 ↪squeeze(preds_tensor.cpu().numpy())

# plot the images in the batch, along with predicted and true labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    imshow(images[idx] if not train_on_gpu else images[idx].cpu())
    ax.set_title("{} ({})".format(classes[preds[idx]], classes[labels[idx]]),
                 color=("green" if preds[idx]==labels[idx].item() else "red"))
```

```
<ipython-input-57-a0724321e9b1>:19: MatplotlibDeprecationWarning: Passing non-
integers as three-element position specification is deprecated since 3.3 and
will be removed two minor releases later.
  ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
```



[58]:
```
## RMSProp

model.load_state_dict(torch.load('model_cifar_rmsprop.pt'))


# track test loss
test_loss = 0.0
class_correct = list(0. for i in range(100))
class_total = list(0. for i in range(100))

model.eval()
# iterate over test data
for data, target in test_loader:
    # move tensors to GPU if CUDA is available
    if train_on_gpu:
        data, target = data.cuda(), target.cuda()
    # forward pass: compute predicted outputs by passing inputs to the model
    output = model(data)
    # calculate the batch loss
    loss = criterion(output, target)
    # update test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
    # compare predictions to true label
    correct_tensor = pred.eq(target.data.view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu else np.
 ↪squeeze(correct_tensor.cpu().numpy())
    # calculate test accuracy for each object class
    for i in range(20):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# average test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))
```

```
for i in range(100):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            classes[i], 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))


#print(np.sum(class_correct))
print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

```
Test Loss: 2.078130

Test Accuracy of apple: 78% (78/100)
Test Accuracy of aquarium_fish: 58% (58/100)
Test Accuracy of  baby: 55% (55/100)
Test Accuracy of  bear:  7% ( 7/100)
Test Accuracy of beaver: 16% (16/100)
Test Accuracy of   bed: 36% (36/100)
Test Accuracy of   bee: 51% (51/100)
Test Accuracy of beetle: 34% (34/100)
Test Accuracy of bicycle: 62% (62/100)
Test Accuracy of bottle: 59% (59/100)
Test Accuracy of  bowl: 32% (32/100)
Test Accuracy of   boy: 15% (15/100)
Test Accuracy of bridge: 56% (56/100)
Test Accuracy of   bus: 25% (25/100)
Test Accuracy of butterfly: 21% (21/100)
Test Accuracy of camel: 37% (37/100)
Test Accuracy of   can: 57% (57/100)
Test Accuracy of castle: 74% (74/100)
Test Accuracy of caterpillar: 42% (42/100)
Test Accuracy of cattle: 42% (42/100)
Test Accuracy of chair: 73% (73/100)
Test Accuracy of chimpanzee: 80% (80/100)
Test Accuracy of clock: 27% (27/100)
Test Accuracy of cloud: 68% (68/100)
Test Accuracy of cockroach: 69% (69/100)
Test Accuracy of couch: 28% (28/100)
Test Accuracy of  crab: 31% (31/100)
Test Accuracy of crocodile: 31% (31/100)
Test Accuracy of   cup: 65% (65/100)
Test Accuracy of dinosaur: 55% (55/100)
Test Accuracy of dolphin: 37% (37/100)
```

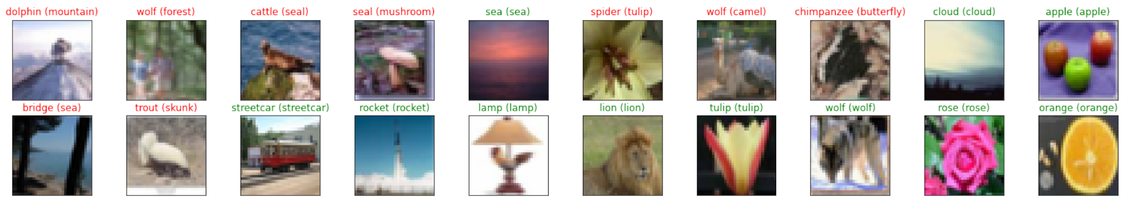
```
Test Accuracy of elephant: 49% (49/100)
Test Accuracy of flatfish: 27% (27/100)
Test Accuracy of forest: 50% (50/100)
Test Accuracy of   fox: 23% (23/100)
Test Accuracy of  girl:  2% ( 2/100)
Test Accuracy of hamster: 39% (39/100)
Test Accuracy of house: 42% (42/100)
Test Accuracy of kangaroo: 42% (42/100)
Test Accuracy of keyboard: 57% (57/100)
Test Accuracy of  lamp: 52% (52/100)
Test Accuracy of lawn_mower: 69% (69/100)
Test Accuracy of leopard: 38% (38/100)
Test Accuracy of  lion: 43% (43/100)
Test Accuracy of lizard: 25% (25/100)
Test Accuracy of lobster: 16% (16/100)
Test Accuracy of   man: 46% (46/100)
Test Accuracy of maple_tree: 40% (40/100)
Test Accuracy of motorcycle: 80% (80/100)
Test Accuracy of mountain: 50% (50/100)
Test Accuracy of mouse: 22% (22/100)
Test Accuracy of mushroom: 40% (40/100)
Test Accuracy of oak_tree: 67% (67/100)
Test Accuracy of orange: 63% (63/100)
Test Accuracy of orchid: 67% (67/100)
Test Accuracy of otter:  1% ( 1/100)
Test Accuracy of palm_tree: 68% (68/100)
Test Accuracy of  pear: 37% (37/100)
Test Accuracy of pickup_truck: 51% (51/100)
Test Accuracy of pine_tree: 41% (41/100)
Test Accuracy of plain: 84% (84/100)
Test Accuracy of plate: 57% (57/100)
Test Accuracy of poppy: 59% (59/100)
Test Accuracy of porcupine: 32% (32/100)
Test Accuracy of possum: 21% (21/100)
Test Accuracy of rabbit: 20% (20/100)
Test Accuracy of raccoon: 48% (48/100)
Test Accuracy of   ray: 30% (30/100)
Test Accuracy of  road: 76% (76/100)
Test Accuracy of rocket: 70% (70/100)
Test Accuracy of  rose: 39% (39/100)
Test Accuracy of   sea: 42% (42/100)
Test Accuracy of  seal: 19% (19/100)
Test Accuracy of shark: 25% (25/100)
Test Accuracy of shrew: 23% (23/100)
Test Accuracy of skunk: 72% (72/100)
Test Accuracy of skyscraper: 71% (71/100)
Test Accuracy of snail: 35% (35/100)
Test Accuracy of snake: 29% (29/100)
```

```
Test Accuracy of spider: 46% (46/100)
Test Accuracy of squirrel:  7% ( 7/100)
Test Accuracy of streetcar: 68% (68/100)
Test Accuracy of sunflower: 87% (87/100)
Test Accuracy of sweet_pepper: 23% (23/100)
Test Accuracy of table: 40% (40/100)
Test Accuracy of  tank: 65% (65/100)
Test Accuracy of telephone: 50% (50/100)
Test Accuracy of television: 45% (45/100)
Test Accuracy of tiger: 55% (55/100)
Test Accuracy of tractor: 68% (68/100)
Test Accuracy of train: 53% (53/100)
Test Accuracy of trout: 66% (66/100)
Test Accuracy of tulip: 11% (11/100)
Test Accuracy of turtle: 24% (24/100)
Test Accuracy of wardrobe: 65% (65/100)
Test Accuracy of whale: 48% (48/100)
Test Accuracy of willow_tree: 37% (37/100)
Test Accuracy of  wolf: 63% (63/100)
Test Accuracy of woman: 18% (18/100)
Test Accuracy of  worm: 70% (70/100)

Test Accuracy (Overall): 45% (4529/10000)
```

[59]:

```
# obtain one batch of test images
dataiter = iter(test_loader)
images, labels = dataiter.next()
images.numpy()

# move model inputs to cuda, if GPU available
if train_on_gpu:
    images = images.cuda()

# get sample outputs
output = model(images)
# convert output probabilities to predicted class
_, preds_tensor = torch.max(output, 1)
preds = np.squeeze(preds_tensor.numpy()) if not train_on_gpu else np.
 ↪squeeze(preds_tensor.cpu().numpy())

# plot the images in the batch, along with predicted and true labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    imshow(images[idx] if not train_on_gpu else images[idx].cpu())
    ax.set_title("{} ({})".format(classes[preds[idx]], classes[labels[idx]]),
                 color=("green" if preds[idx]==labels[idx].item() else "red"))
```

```
<ipython-input-59-a0724321e9b1>:19: MatplotlibDeprecationWarning: Passing non-integers as three-element position specification is deprecated since 3.3 and will be removed two minor releases later.
  ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
```



[ ]:

```

```

1.7 Let's see some custom optimiser

1.8 DemonRanger

[67]:

```python
from optimizers import *
```

[ ]:

```

```

[89]:

```python
learning_rate = 0.1
weight_decay = 1e-4

model = resnet20(num_classes=100)
model=model.cuda()
```

[90]:

```python
criterion = nn.CrossEntropyLoss()
```

AMSGrad is a stochastic optimization method that seeks to fix a convergence issue with Adam based optimizers. AMSGrad uses the maximum of past squared gradients v_t rather than the exponential average to update the parameters:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t$$

AMSGRAD

1.9 Algorithm

Input: $x_1 \in \mathcal{F}$, step size $\{\alpha_t\}_{t=1}^{T}$, $\{\beta_{1t}\}_{t=1}^{T}$, β_2
Set $m_0 = 0$, $v_0 = 0$ and $\hat{v}_0 = 0$
for $t = 1$ **to** T **do**
  $g_t = \nabla f_t(x_t)$
  $m_t = \beta_{1t} m_{t-1} + (1 - \beta_{1t}) g_t$
  $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$ and $\hat{V}_t = \mathrm{diag}(\hat{v}_t)$
  $x_{t+1} = \Pi_{\mathcal{F}, \sqrt{\hat{V}_t}}(x_t - \alpha_t m_t / \sqrt{\hat{v}_t})$
end for

[70]:
```
optimizer_AMSGRAD = DemonRanger(params=model.parameters(),
                                lr=0.01,
                                betas=(0.9,0.999,0.999), # restore default AdamW betas
                                nus=(1.0,1.0), # disables QHMomentum
                                k=0,  # disables lookahead
                                alpha=1.0,
                                #weight_decay=config.wd,
                                IA=False, # disables Iterate Averaging
                                rectify=False, # disables RAdam Recitification
                                AdaMod=False, #disables AdaMod
                                use_demon=False, #disables Decaying Momentum (DEMON)
                                use_gc=False, #disables gradient centralization
                                amsgrad=True # disables amsgrad
                                )
```

The **Quasi-Hyperbolic Momentum Algorithm (QHM)** is a simple alteration of momentum SGD, averaging a plain SGD step with a momentum step. **QHAdam** is a QH augmented version of Adam, where we replace both of Adam's moment estimators with quasi-hyperbolic terms. QHAdam decouples the momentum term from the current gradient when updating the weights, and decouples the mean squared gradients term from the current squared gradient when updating the weights.

In essence, it is a weighted average of the momentum and plain SGD, weighting the current gradient with an immediate discount factor v_1 divided by a weighted average of the mean squared gradients and the current squared gradient, weighting the current squared gradient with an immediate discount factor v_2.

$$\theta_{t+1,i} = \theta_{t,i} - \eta \left[\frac{(1 - v_1) \cdot g_t + v_1 \cdot \hat{m}_t}{\sqrt{(1 - v_2) g_t^2 + v_2 \cdot \hat{v}_t + \epsilon}} \right], \forall t$$

It is recommended to set $v_2 = 1$ and β_2 same as in Adam.

QHAdam

1.9.1 QHM update rule

QHM, parameterized by R, R, and R, uses the update rule:

$$g_{t+1} \leftarrow \cdot g_t + (1-) \cdot \ L_t(_t)$$

$$_{t+1} \leftarrow _t - [(1-) \cdot \ L_t(_t) + \ \cdot g_{t+1}]$$

[73]:
```
optimizer_QHAdam = DemonRanger(params=model.parameters(),
                        lr=0.01,
                        k=0,  # disables lookahead
                        alpha=1.0,
                        IA=False, # disables Iterate Averaging
                        rectify=False, # disables RAdam Recitification
                        AdaMod=False, #disables AdaMod
                        use_demon=False, #disables Decaying Momentum (DEMON)
                        use_gc=False, #disables gradient centralization
                        amsgrad=False # disables amsgrad
                        )
```

Demon Adam is a stochastic optimizer where the Demon momentum rule is applied to the Adam optimizer.

$$\beta_t = \beta_{init} \cdot \frac{\left(1 - \frac{t}{T}\right)}{(1 - \beta_{init}) + \beta_{init}\left(1 - \frac{t}{T}\right)}$$

$$m_{t,i} = g_{t,i} + \beta_t m_{t-1,i}$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_t^2$$

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Demon Adam

[91]:
```
optimizer_Demon = DemonRanger(params=model.parameters(),
                        lr=0.01,
                        #weight_decay=config.wd,
                        #epochs = config.epochs,
                        #step_per_epoch = step_per_epoch,
                        betas=(0.9,0.999,0.999), # restore default AdamW betas
                        nus=(1.0,1.0), # disables QHMomentum
                        k=0,  # disables lookahead
                        alpha=1.0,
                        IA=False, # enables Iterate Averaging
                        rectify=False, # disables RAdam Recitification
```

```
                        AdaMod=False, #disables AdaMod
                        AdaMod_bias_correct=False, #disables AdaMod bias␣
↪correction (not used originally)
                        use_demon=True, #enables Decaying Momentum (DEMON)
                        use_gc=False, #disables gradient centralization
                        amsgrad=False # disables amsgrad
                        )
```

[ ]:

1.9.2 Training

[81]:
```
## AMSGrad



# number of epochs to train the model
n_epochs = 30

valid_loss_min = np.Inf # track change in validation loss

train_loss_list3 = []
valid_loss_list3 = []

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train()
    for data, target in train_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer_AMSGRAD.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model␣
↪parameters
        loss.backward()
```

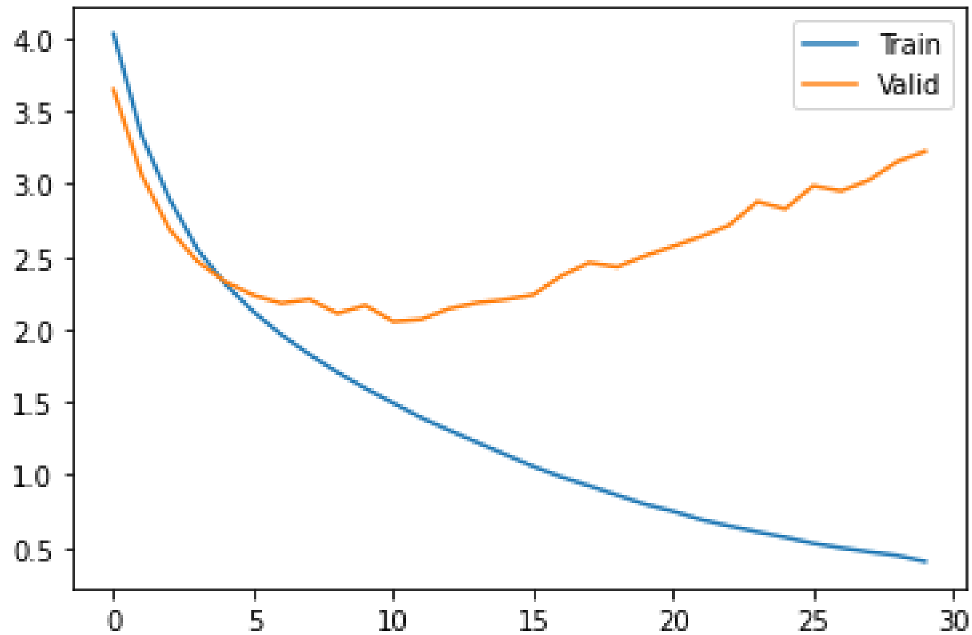
```
        #print("conv1 grads",torch.linalg.norm(model.conv1.weight.grad))
        #print("conv2 grads",torch.linalg.norm(model.conv2.bias.grad))
        # perform a single optimization step (parameter update)
        optimizer_AMSGRAD.step()
        # update training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    model.eval()
    for data, target in valid_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += loss.item()*data.size(0)

    # calculate average losses
    train_loss = train_loss/len(train_loader.sampler)
    valid_loss = valid_loss/len(valid_loader.sampler)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))
    train_loss_list3.append(train_loss)
    valid_loss_list3.append(valid_loss)

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
↪'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model_cifar_.pt')
        valid_loss_min = valid_loss
```

```
/home/sysadm/Documents/Dristanta_ML_Project/optimizers.py:398: UserWarning: This
overload of addcmul_ is deprecated:
        addcmul_(Number value, Tensor tensor1, Tensor tensor2)
Consider using one of the following signatures instead:
        addcmul_(Tensor tensor1, Tensor tensor2, *, Number value) (Triggered
internally at  /pytorch/torch/csrc/utils/python_arg_parser.cpp:1025.)
```

```
  exp_avg_sq.mul_(beta2).addcmul_(1 - beta2, grad, grad)
Epoch: 1        Training Loss: 4.030989         Validation Loss: 3.647615
Validation loss decreased (inf --> 3.647615).  Saving model …
Epoch: 2        Training Loss: 3.335276         Validation Loss: 3.061632
Validation loss decreased (3.647615 --> 3.061632).  Saving model …
Epoch: 3        Training Loss: 2.894942         Validation Loss: 2.686971
Validation loss decreased (3.061632 --> 2.686971).  Saving model …
Epoch: 4        Training Loss: 2.550194         Validation Loss: 2.466828
Validation loss decreased (2.686971 --> 2.466828).  Saving model …
Epoch: 5        Training Loss: 2.310163         Validation Loss: 2.326766
Validation loss decreased (2.466828 --> 2.326766).  Saving model …
Epoch: 6        Training Loss: 2.120512         Validation Loss: 2.234374
Validation loss decreased (2.326766 --> 2.234374).  Saving model …
Epoch: 7        Training Loss: 1.962311         Validation Loss: 2.179182
Validation loss decreased (2.234374 --> 2.179182).  Saving model …
Epoch: 8        Training Loss: 1.826674         Validation Loss: 2.207230
Epoch: 9        Training Loss: 1.704483         Validation Loss: 2.109395
Validation loss decreased (2.179182 --> 2.109395).  Saving model …
Epoch: 10       Training Loss: 1.593407         Validation Loss: 2.165431
Epoch: 11       Training Loss: 1.492046         Validation Loss: 2.052980
Validation loss decreased (2.109395 --> 2.052980).  Saving model …
Epoch: 12       Training Loss: 1.392124         Validation Loss: 2.068454
Epoch: 13       Training Loss: 1.305866         Validation Loss: 2.142862
Epoch: 14       Training Loss: 1.222614         Validation Loss: 2.182699
Epoch: 15       Training Loss: 1.139681         Validation Loss: 2.205436
Epoch: 16       Training Loss: 1.057265         Validation Loss: 2.237428
Epoch: 17       Training Loss: 0.985529         Validation Loss: 2.367383
Epoch: 18       Training Loss: 0.924271         Validation Loss: 2.458560
Epoch: 19       Training Loss: 0.859855         Validation Loss: 2.429482
Epoch: 20       Training Loss: 0.797164         Validation Loss: 2.505888
Epoch: 21       Training Loss: 0.749100         Validation Loss: 2.570983
Epoch: 22       Training Loss: 0.691530         Validation Loss: 2.637795
Epoch: 23       Training Loss: 0.647840         Validation Loss: 2.714430
Epoch: 24       Training Loss: 0.607191         Validation Loss: 2.877277
Epoch: 25       Training Loss: 0.569606         Validation Loss: 2.827529
Epoch: 26       Training Loss: 0.528353         Validation Loss: 2.986218
Epoch: 27       Training Loss: 0.497122         Validation Loss: 2.950652
Epoch: 28       Training Loss: 0.469734         Validation Loss: 3.025858
Epoch: 29       Training Loss: 0.444439         Validation Loss: 3.152833
Epoch: 30       Training Loss: 0.403728         Validation Loss: 3.223494
```

[82]:
```
plt.plot(train_loss_list3)
plt.plot(valid_loss_list3)
plt.legend(["Train","Valid"])
```

[82]: <matplotlib.legend.Legend at 0x7f727008a220>



[87]:
```
## QHAdam


# number of epochs to train the model
n_epochs = 30

valid_loss_min = np.Inf # track change in validation loss

train_loss_list4 = []
valid_loss_list4 = []

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train()
    for data, target in train_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
```

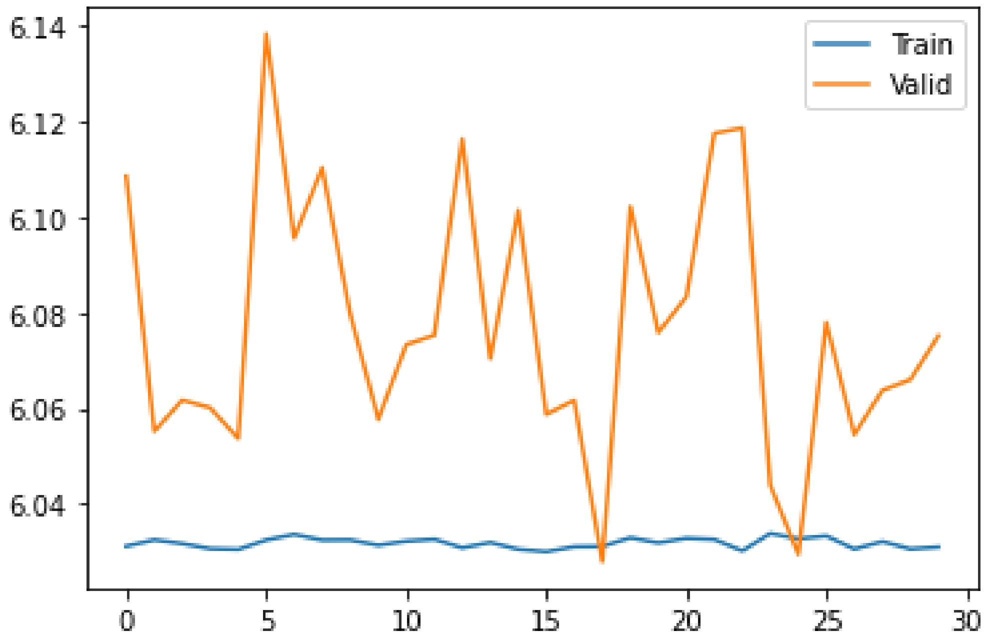
```
        optimizer_QHAdam.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model␣
↪parameters
        loss.backward()
        #print("conv1 grads",torch.linalg.norm(model.conv1.weight.grad))
        #print("conv2 grads",torch.linalg.norm(model.conv2.bias.grad))
        # perform a single optimization step (parameter update)
        optimizer_QHAdam.step()
        # update training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    model.eval()
    for data, target in valid_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += loss.item()*data.size(0)

    # calculate average losses
    train_loss = train_loss/len(train_loader.sampler)
    valid_loss = valid_loss/len(valid_loader.sampler)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))
    train_loss_list4.append(train_loss)
    valid_loss_list4.append(valid_loss)

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
↪'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model_cifar_QHAdam.pt')
```

```
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 6.031128         Validation Loss: 6.108565
Validation loss decreased (inf --> 6.108565).  Saving model …
Epoch: 2        Training Loss: 6.032455         Validation Loss: 6.055208
Validation loss decreased (6.108565 --> 6.055208).  Saving model …
Epoch: 3        Training Loss: 6.031669         Validation Loss: 6.061704
Epoch: 4        Training Loss: 6.030608         Validation Loss: 6.060126
Epoch: 5        Training Loss: 6.030455         Validation Loss: 6.053710
Validation loss decreased (6.055208 --> 6.053710).  Saving model …
Epoch: 6        Training Loss: 6.032426         Validation Loss: 6.138397
Epoch: 7        Training Loss: 6.033608         Validation Loss: 6.095640
Epoch: 8        Training Loss: 6.032398         Validation Loss: 6.110361
Epoch: 9        Training Loss: 6.032434         Validation Loss: 6.079818
Epoch: 10       Training Loss: 6.031327         Validation Loss: 6.057736
Epoch: 11       Training Loss: 6.032192         Validation Loss: 6.073348
Epoch: 12       Training Loss: 6.032579         Validation Loss: 6.075305
Epoch: 13       Training Loss: 6.030794         Validation Loss: 6.116459
Epoch: 14       Training Loss: 6.031901         Validation Loss: 6.070436
Epoch: 15       Training Loss: 6.030481         Validation Loss: 6.101495
Epoch: 16       Training Loss: 6.029984         Validation Loss: 6.058753
Epoch: 17       Training Loss: 6.031021         Validation Loss: 6.061681
Epoch: 18       Training Loss: 6.031075         Validation Loss: 6.027929
Validation loss decreased (6.053710 --> 6.027929).  Saving model …
Epoch: 19       Training Loss: 6.032909         Validation Loss: 6.102229
Epoch: 20       Training Loss: 6.031836         Validation Loss: 6.075868
Epoch: 21       Training Loss: 6.032803         Validation Loss: 6.083347
Epoch: 22       Training Loss: 6.032547         Validation Loss: 6.117578
Epoch: 23       Training Loss: 6.030155         Validation Loss: 6.118704
Epoch: 24       Training Loss: 6.033804         Validation Loss: 6.043897
Epoch: 25       Training Loss: 6.032688         Validation Loss: 6.029434
Epoch: 26       Training Loss: 6.033289         Validation Loss: 6.077964
Epoch: 27       Training Loss: 6.030533         Validation Loss: 6.054522
Epoch: 28       Training Loss: 6.032103         Validation Loss: 6.063757
Epoch: 29       Training Loss: 6.030549         Validation Loss: 6.066008
Epoch: 30       Training Loss: 6.030949         Validation Loss: 6.075199
```

[88]:
```
plt.plot(train_loss_list4)
plt.plot(valid_loss_list4)
plt.legend(["Train","Valid"])
```

[88]: <matplotlib.legend.Legend at 0x7f71ef401790>



[ ]:

```

```

[ ]:

```
## Demon


# number of epochs to train the model
n_epochs = 30

valid_loss_min = np.Inf # track change in validation loss

train_loss_list5 = []
valid_loss_list5 = []

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train()
    for data, target in train_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
```

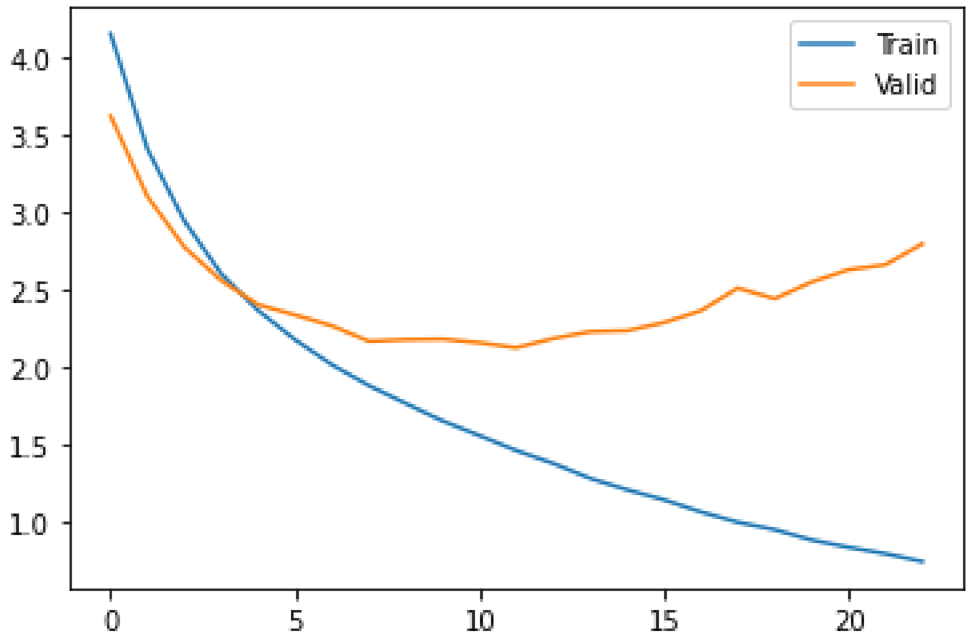
```
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer_Demon.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model␣
↪parameters
        loss.backward()
        #print("conv1 grads",torch.linalg.norm(model.conv1.weight.grad))
        #print("conv2 grads",torch.linalg.norm(model.conv2.bias.grad))
        # perform a single optimization step (parameter update)
        optimizer_Demon.step()
        # update training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    model.eval()
    for data, target in valid_loader:
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += loss.item()*data.size(0)

    # calculate average losses
    train_loss = train_loss/len(train_loader.sampler)
    valid_loss = valid_loss/len(valid_loader.sampler)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))
    train_loss_list5.append(train_loss)
    valid_loss_list5.append(valid_loss)

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
↪'.format(
        valid_loss_min,
```

```
        valid_loss))
        torch.save(model.state_dict(), 'model_cifar_Demon.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 4.155434         Validation Loss: 3.624016
Validation loss decreased (inf --> 3.624016).  Saving model …
Epoch: 2        Training Loss: 3.408505         Validation Loss: 3.101152
Validation loss decreased (3.624016 --> 3.101152).  Saving model …
Epoch: 3        Training Loss: 2.943246         Validation Loss: 2.774654
Validation loss decreased (3.101152 --> 2.774654).  Saving model …
Epoch: 4        Training Loss: 2.602039         Validation Loss: 2.558676
Validation loss decreased (2.774654 --> 2.558676).  Saving model …
Epoch: 5        Training Loss: 2.364206         Validation Loss: 2.401721
Validation loss decreased (2.558676 --> 2.401721).  Saving model …
Epoch: 6        Training Loss: 2.174921         Validation Loss: 2.335534
Validation loss decreased (2.401721 --> 2.335534).  Saving model …
Epoch: 7        Training Loss: 2.012397         Validation Loss: 2.264961
Validation loss decreased (2.335534 --> 2.264961).  Saving model …
Epoch: 8        Training Loss: 1.877797         Validation Loss: 2.165033
Validation loss decreased (2.264961 --> 2.165033).  Saving model …
Epoch: 9        Training Loss: 1.762087         Validation Loss: 2.175348
Epoch: 10       Training Loss: 1.649427         Validation Loss: 2.178752
Epoch: 11       Training Loss: 1.554248         Validation Loss: 2.154702
Validation loss decreased (2.165033 --> 2.154702).  Saving model …
Epoch: 12       Training Loss: 1.457562         Validation Loss: 2.123246
Validation loss decreased (2.154702 --> 2.123246).  Saving model …
Epoch: 13       Training Loss: 1.373948         Validation Loss: 2.182807
Epoch: 14       Training Loss: 1.277112         Validation Loss: 2.227169
Epoch: 15       Training Loss: 1.203368         Validation Loss: 2.233433
Epoch: 16       Training Loss: 1.138331         Validation Loss: 2.286236
Epoch: 17       Training Loss: 1.059569         Validation Loss: 2.362591
Epoch: 18       Training Loss: 0.992976         Validation Loss: 2.509676
Epoch: 19       Training Loss: 0.944879         Validation Loss: 2.440402
Epoch: 20       Training Loss: 0.878072         Validation Loss: 2.548921
Epoch: 21       Training Loss: 0.830218         Validation Loss: 2.628512
Epoch: 22       Training Loss: 0.788947         Validation Loss: 2.660701
Epoch: 23       Training Loss: 0.738655         Validation Loss: 2.795871
```

[4]:
```
plt.plot(train_loss_list5)
plt.plot(valid_loss_list5)
plt.legend(["Train","Valid"])
```

[4]: <matplotlib.legend.Legend at 0x7f9bb1e948e0>


Train
Valid
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0
5
10
15
20

```
[ ]:
```