Shadow

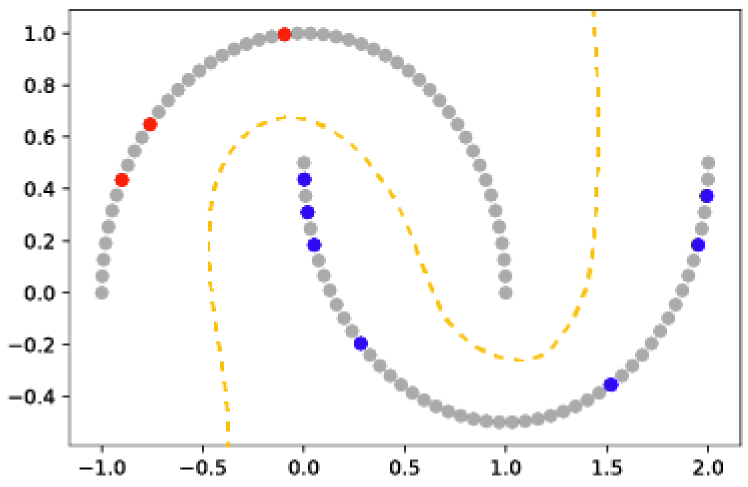
National Technology & Engineering Solutions of Sandia, LLC (NT

Oct 07, 2021

CONTENTS

SHADOW

Shadow is a PyTorch based library for semi-supervised machine learning. The `shadow` python 3 package includes implementations of Virtual Adversarial Training, Mean Teacher, and Exponential Averaging Adversarial Training. Semi-supervised learning enables training a model (gold dashed line) from both labeled (red and blue) and unlabeled (grey) data, and is typically used in contexts in which labels are expensive to obtain but unlabeled examples are plentiful.



CHAPTER

ONE

GITHUB DEVELOPMENT PAGE:

https://github.com/sandialabs/shadow

CHAPTER TWO

INSTALLATION

Shadow can be installed directly from pypi as:

```
pip install shadow-ssml
```

HELLO WORLD

Incorporating consistency regularizers into an existing supervised workflow for semi-supervised learning is straightforward. First, Shadow provides techniques that wrap an existing PyTorch model:

```
model = ...  # PyTorch torch.nn.Module
eaat = shadow.eaat.Eaat(model)  # Wrapped model
```

The wrapped model is used during training and inference. The model wrapper provides a *get_technique_cost* method for computed the consistency cost based on unlabeled data. This loss can be added to an existing loss computation to enable semi-supervised learning:

```
for x, y in trainloader:
    # zero the parameter gradients
    optimizer.zero_grad()

    # forward pass
    outputs = eaat(x)

    # get semi-supervised loss, using supervised criterion and unsupervised criterion
    # provided by the model wrapper
    loss = criterion(x, y) + eaat.get_technique_cost(x)
    loss.backward()
    optimizer.step()
```

For a full working example, see the *MNIST Example*.

CHAPTER FOUR

CITING SHADOW

To cite shadow, use the following reference:

- Linville, L., Anderson, D., Michalenko, J., Galasso, J., & Draelos, T. (2021). Semisupervised Learning for Seismic Monitoring Applications. Seismological Society of America, 92(1), 388-395. doi: https://doi.org/10.1785/0220200195

CHAPTER

FIVE

CONTENTS

5.1 Overview

The `shadow` python package implements multiple techniques for semi-supervised learning. Semi-supervised learning (SSL) enables training a model from both labeled and unlabeled data, and is typically used in contexts in which labels are expensive to obtain but unlabeled examples are plentiful. This is illustrated in Fig. 5.1. This package is built upon the PyTorch machine learning framework.

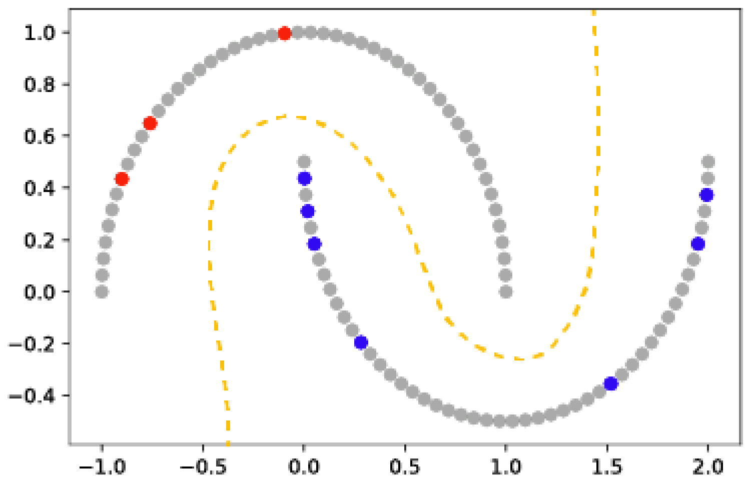


Fig. 5.1: Semi-supervised learning uses both labeled (red and blue) and unlabeled (grey) data to learn a better model (gold dashed line).

5.1.1 Package Design

Semi-supervised techniques require two steps for integration into an existing supervised workflow. First, the existing PyTorch model is wrapped using the desired technique, in this case Exponential Average Adversarial Training (EAAT):

```
model = ...  # PyTorch torch.nn.Module
eaat = shadow.eaat.Eaat(model)  # Wrapped model
```

Next, the loss function computation during training is modified to include a call to the wrapper-provided *get_technique_cost*:

```
loss = criterion(x, y) + eaat.get_technique_cost(x)
```

Note that as posed in this code example, both labeled and unlabeled data are passed to the supervised part of the loss function, *criterion*. The PyTorch CrossEntropyLoss provides an *ignore_index* parameter. For samples without labels, *y* is set to this *ignore_index*. Other loss functions can be modified to mask the mini-batch accordingly.

5.1.2 Consistency Regularization

Semi-supervised learning requires assumptions about the underlying data in order to make use of the unlabeled data. The majority of techniques implemented in the `shadow` package are consistency regularizers: they assume that the target variable changes smoothly over the data space. Said another way: data points close to each other likely share the same label. To enforce this smoothness, a consistency penalty is added to the overall loss function as:

$$\mathcal{L}(f_\theta(x_l), y_l) + \alpha g(f_\theta, x)$$

where $\mathcal{L}$ is the original loss function, (x_l, y_l) represents labeled data, $f_\theta(x_l)$ represents the predictions from model f_θ, α represents a relative penalty weighting (hyperparameter), g represents a consistency enforcing function, and x represents all data (labeled and unlabeled). Critically, the consistency function depends upon input data but not the labels, which enables semi-supervised learning. The specific form of g and the mechanism by which it enforces consistency varies between techniques.

Virtual Adversarial Training

Virtual Adversarial Training [Miyato18] (VAT) enforces model consistency over small perturbations to the input. Accordingly, the consistency term penalizes the difference between the model output of the data and the model output of perturbed data:

$$g(f_\theta, x) = d(f_\theta(x), f_\theta(x + r_{adv}))$$

where d represents some divergence function (typically mean squared error or Kullback-Liebler divergence) and r_{adv} represents a small data perturbation. Instead of sampling perturbations r_{adv} at random, VAT generates perturbations in the virtual adversarial direction which corresponds to the direction of greatest change in model output, enabling more effective regularization. This is illustrated in Fig. 5.2.

Mean Teacher

Mean Teacher [Tarvainen17] (MT) enforces consistency over model weights. MT builds on the idea that averaging over training epochs (temporal ensembling) produces more stable predictions. While temporal ensembling originally averaged the output values, [Tarvainen17] demonstrated advantages and performance increases by using the exponential moving average (EMA) of the model weights (denoted as the “teacher”) during training. Additionally, MT further assumes that additive noise or realistic data augmentation should minimally affect current model predictions. The consistency loss term for Mean Teacher is therefore:

$$g(f_\theta, x) = d(f_\theta(x + n), f'_\theta(x + n)$$

Where f'_θ (teacher) represents the weight-wise exponential moving average of the model f_θ (student), and n represents additive noise. This is illustrated in Fig. 5.3.

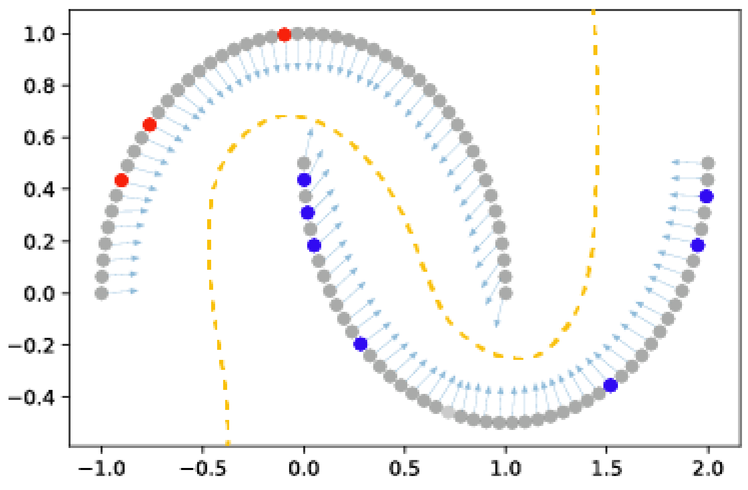


Fig. 5.2: Virtual Adverarial Training [Miyato18] enforces consistency over perturbations to model input (shown as arrows). Pertubations are in the virtual adversarial direction, which represents the maximum change in model output.

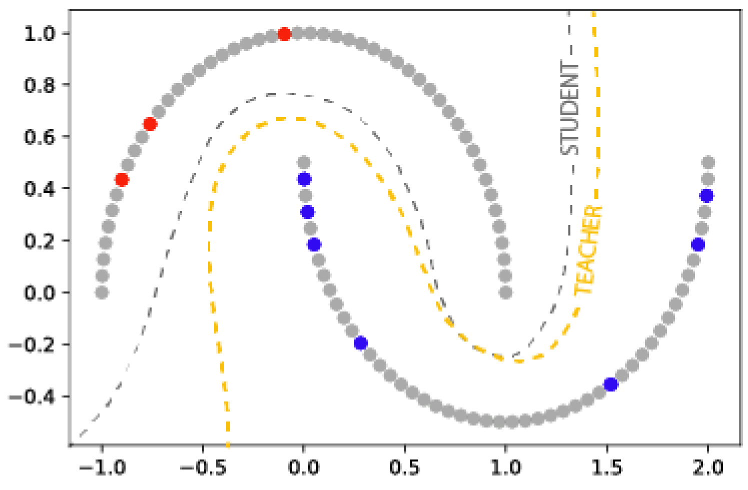


Fig. 5.3: Mean Teacher [Tarvainen17] enforces consistency over model weights between a student (grey dashed line) and a teacher (gold dashed line) model. The teacher model is the weight-wise exponential moving average during training of the student.

Exponential Averaging Adversarial Training

A natural extension of MT and VAT is to leverage the MT teacher-student framework but utilize virtual adversarial perturbations to regularize the student. We denote this joint implementation as Exponential Average Adversarial Training [Linville21] (EAAT). The consistency function is given as:

$$g(f_\theta, x) = d(f_\theta(x + r_{adv}), f'_\theta(x))$$

This is illustrated in Fig. 5.4.

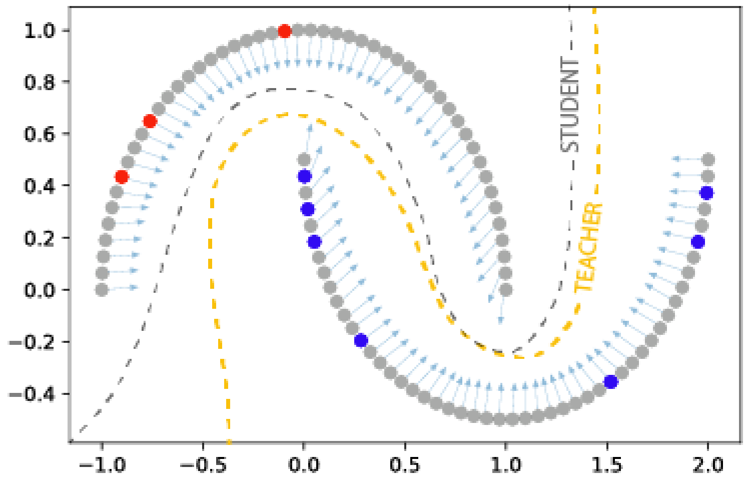


Fig. 5.4: Exponential Averaging Adversarial Training [Linville21] combines both VAT and MT by enforcing consistency between a teacher and a student in which the student is given virtual adversarial perturbed data.

5.1.3 Practical Recommendations

The techniques in `shadow` were developed to test the performance of various approaches to semi-supervised learning in a new application domain: seismic waveform data. Although we primarily focus on classification, the generalized framework provided here supports both classification and regression tasks. Although all new datasets and techniques require significant investment in tuning and optimization, for many of these SSL techniques we have observed significant sensitivity to small changes in hyperparameter settings and experiment set-up. Below, we offer some lessons learned for training and experiment setup for these techniques.

Although VAT and EAAT seem to tolerate significant imbalance between unlabeled and labeled data fractions per class, MT often learns best with a 50/50 labeled/unlabeled data fraction within each mini-batch. In the case of small label budgets this implies significant oversampling of labeled data. In many of our experiments, we found a consistency cost weighted at 2-4 times that of the class loss enabled meaningful learning beyond simply fitting the label set.

For example, to weight the consistency cost:

```
loss = criterion(x, y) + weight * mt.get_technique_cost(x)
```

VAT, on the other hand, often trains longer under large label/unlabeled fractions per batch, can tolerate a range of loss weights, but can be sensitive to the perturbation amplitude (*xi*). We suggest *xi* be tuned in advance of model training for new datasets to ensure the perturbation amplitudes are not unreasonably large or converge to zero over one power iteration. The *xi_check* parameter can be turned on to guide initial order of magnitude studies in this regard.

Likewise, the eigenvector estimation used to find virtual adversarial directions yields a sign ambiguity: perturbations are often estimated in the negative of the direction that provides the maximum change in model output. We provide a method to resolve the direction ambiguity as a technique parameter: *flip_correction* (defaults to *True*). If set to *False*, computation is faster but convergence may be slower as competing directions may more closely resemble random perturbations.

EAAT has more complexity than MT or VAT alone; it requires consistency between input and adversarially perturbed input on the exponential average model. Added loss complexity often requires more extensive hyperparameter exploration. In our experiments, this included the considerations mentioned above for both MT and VAT and model depth, which appeared to limit SSL performance more than fully-supervised learning in the same data regime using EAAT.

In [Linville21], we examine the performance between MT, VAT, and EAAT against several baselines in a label-limited regime (where unlabeled data significantly outweighs the labeled data quantity). In these experiments, SSL outperforms baselines significantly. However, we also highlight that there is a limit to SSL performance as the number of available labels increases. When larger label fractions are available, SSL for our data can typically match but not increase performance compared to fully-supervised models, but at the expense of significantly more time spent on parameter optimization. One exception is that adding even minimal quantities of unlabeled data from out-of-domain (OOD) examples, in this case geographically, can positively impact prediction accuracy on new OOD examples, even when the number of unlabeled OOD examples is small compared to the number of labeled examples.

We hope the consistency-based SSL techniques provided here enable exploration on a wide variety of problems and datasets. To get you started, we provide a simple use example on MNIST available in *MNIST Example*.

5.2 Half Moons Example

This notebook demonstrates the use of Shadow for the canonical semi-supervised learning example: half moons.

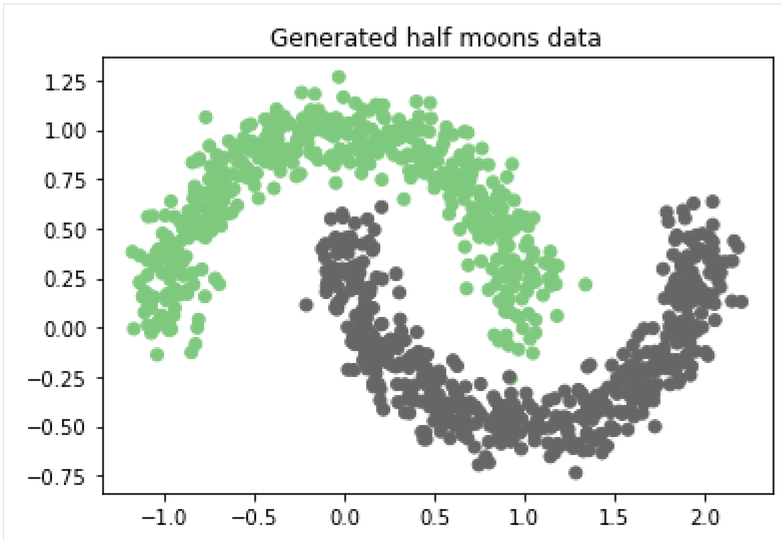
A strictly supervised (only labeled data) baseline model is trained alongside the same model architecture trained using Exponential Averaging Adversarial Training (EAAT).

[1]:
```
import shadow.utils
shadow.utils.set_seed(0, cudnn_deterministic=True)  # set seeds for reproducibility
```

5.2.1 Create Data

Half moons data is generated using the scikit-learn toy datasets interface.

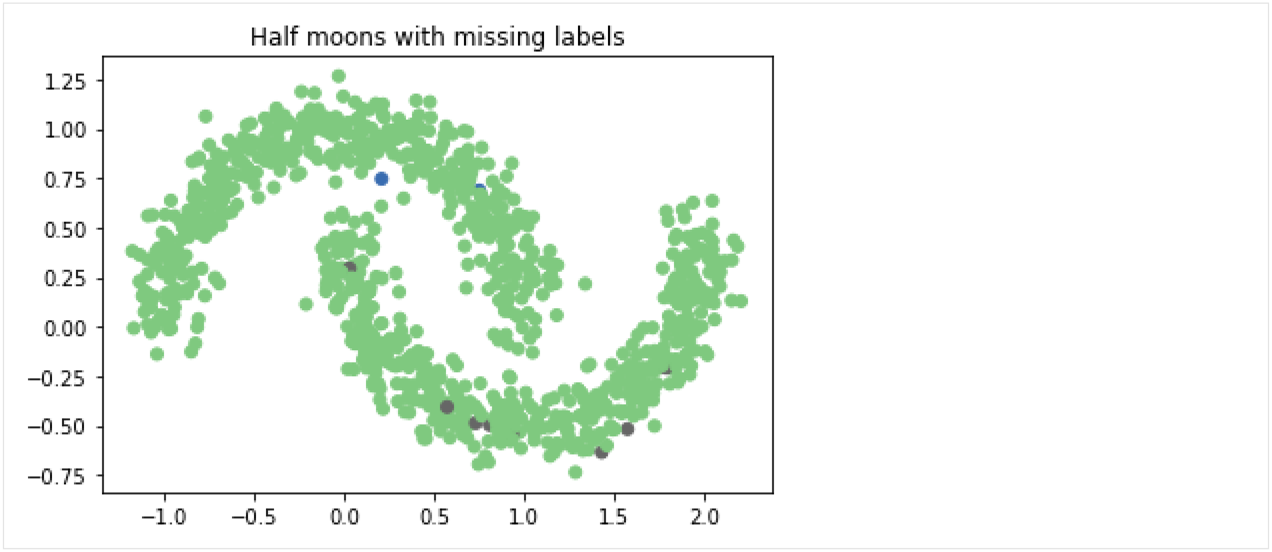
[2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn import datasets

n_samples = 1000  # number of samples to generate
noise = 0.1  # noise to add to sample locations
x, y = datasets.make_moons(n_samples=n_samples, noise=noise)

plt.scatter(*x.T, c=y, cmap=plt.cm.Accent)
plt.title("Generated half moons data");
```



5.2.2 Drop Labels

Since the half moons dataset is synthetic, all labels are available. To test out the semi-supervised learning infrastructure, we artificially remove 99% of labels. To demarcate "unlabeled" samples, we set their corresponding label to `-1`. This is convenient method to handle missing classification labels in torch, as we will see later in the definition of the cross entropy loss.

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

unlabeled_frac = 0.99  # fraction of data to drop labels

y_ssml = y.copy()
unlabeled = np.random.choice(range(n_samples), size=int(unlabeled_frac * n_samples),␣
↪replace=False)
y_ssml[unlabeled] = -1  # set missing labels to -1

plt.scatter(*x.T, c=y_ssml, cmap=plt.cm.Accent)
plt.title("Half moons with missing labels");
```



Train a classifier

Next, we will instantiate the various training objects used in torch: models, optimizers, and criterion.

5.2.3 Model Architecture

We define our simple model architecture for use on this toy problem. We wrap this model into a factory, so that we can produce it twice: once for the baseline and once for the semi-supervised test.

[4]:
```
import torch

def model_factory():
    return torch.nn.Sequential(
        torch.nn.Linear(2, 10),
        torch.nn.ReLU(),
        torch.nn.Linear(10, 10),
        torch.nn.ReLU(),
        torch.nn.Linear(10, 2)
    )

device = torch.device('cpu')  # run on cpu, since model and data are very small
```

5.2.4 Baseline Model

The baseline model and optimizer are defined normally.

[5]:
```
bl = model_factory().to(device)
bl_opt = torch.optim.SGD(bl.parameters(), lr=0.1, momentum=0.9)
```

5.2.5 Semi-supervised Model

The semi-supervised model is instantiated with one critical difference: the model is wrapped by the shadow-provided ssml technique (in this case Exponential Averaging Adversarial Training). The wrapper defines additional technique-specific hyperparameters, but should be instead of using a plain model directly.

```
[6]: import shadow.eaat
eaat = shadow.eaat.EAAT(model=model_factory(), alpha=0.8, xi=1e-4, eps=0.3).to(device)
eaat_opt = torch.optim.SGD(eaat.parameters(), lr=0.1, momentum=0.9)
```

5.2.6 Loss function

As was alluded to earlier, using `-1` to indicate missing labels is convenient in torch. That is because our standard CrossEntropyLoss criterion already has a parameter to ignore specific label values. We simple instatiate the loss function as normal, but set that parameter (`ignore_index`) to our unlabeled value.

Other loss functions may provide such a functionality or may need to be manually modified so as to mask data with missing labels.

```
[7]: xEnt = torch.nn.CrossEntropyLoss(ignore_index=-1).to(device)
```

5.2.7 Training Loop

We train both the baseline and semi-supervised models at the same time using standard `torch` conventions. Since the half-moons dataset is a small toy example, we use the whole dataset at once instead of batching.

The only modification to the training loop needed for the semi-supervised model is that the loss function is composed of both the supervised loss (`xEnt`) as well as a technique provided unsupervised loss (`eaat.get_technique_cost`). The details of this unsupervised loss vary between techniques, but it provides the mechanism for semi-supervised learning.

```
[8]: n_epochs = 500
xt, yt = torch.Tensor(x).to(device), torch.LongTensor(y_ssml).to(device)
for epoch in range(n_epochs):
    # Standard forward/backward pass for training baseline
    out = bl(xt)
    loss = xEnt(out, yt)  # ignores the unlabeled data (-1)
    bl_opt.zero_grad()
    loss.backward()
    bl_opt.step()

    # Forward/backward pass for training semi-supervised model
    out = eaat(xt)
    loss = xEnt(out, yt) + eaat.get_technique_cost(xt)  # supervised + unsupervised␣
↪loss
    eaat_opt.zero_grad()
    loss.backward()
    eaat_opt.step()
```

Evaluate performance

Finally, we evaluate and compare performance between our strictly supervised baseline model, and the same model trained using semi-supervised learning.

It is critical to set the EAAT wrapped model into `eval` mode after training, so as to disable certain augmentations performed for semi-supervised learning.

```
[9]: bl.eval()
     eaat.eval();
```

The accuracy provided by SSML greatly outperforms that of the baseline, as we would expect.

```
[10]: import shadow.losses

      bl_pred = torch.max(bl(xt), 1)[-1]
      eaat_pred = torch.max(eaat(xt), 1)[-1]

      print("bl accuracy:", shadow.losses.accuracy(bl_pred, torch.LongTensor(y)).data)
      print("eaat accuracy", shadow.losses.accuracy(eaat_pred, torch.LongTensor(y)).data)
```

```
bl accuracy: tensor(75.9000, dtype=torch.float64)
eaat accuracy tensor(98.7000, dtype=torch.float64)
```

5.2.8 Visualize Decision Boundary

In order to get a better sense of how "good" our EAAT trained model is, we can leverage the fact that half-moons is a toy 2D dataset to visualize the model decision boundary. We define a 2D meshgrid over the data space and evaluate the models over the grid. We then plot the contour corresponding to a score of `0.5`, which represents the decision boundary.

As illustrated in the plot, the baseline learns exactly what it can from the labeled data. However the semi-supervised trained model learns both labeled and unlabeled data and captures the overall data trends despite having labels for only 1% of the training data.

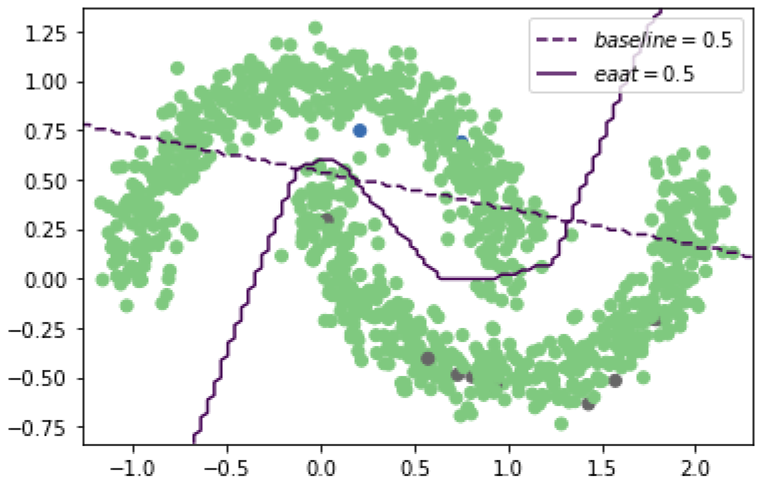
```
[11]: # Determine grid range
      x0_min, x0_max = x[:, 0].min() - 0.1, x[:, 0].max() + 0.1
      x1_min, x1_max = x[:, 1].min() - 0.1, x[:, 1].max() + 0.1

      # Create grid
      x0, x1 = np.meshgrid(np.linspace(x0_min, x0_max, 100),
                           np.linspace(x1_min, x1_max, 100))
      grid = torch.FloatTensor(np.hstack((x0.reshape(-1, 1), x1.reshape(-1, 1))))

      # Plot the data
      plt.scatter(*x.T, c=y_ssml, cmap=plt.cm.Accent)

      # Evaluate baseline on the grid and plot decision boundary
      bl_pred = torch.max(bl(grid), 1)[-1].numpy().reshape(x0.shape)
      cs = plt.contour(x0, x1, bl_pred, levels=[0.5], linestyles='--')
      h1, l1 = cs.legend_elements("baseline")

      # Evaluate EAAT on the grid and plot decision boundary
      eaat_pred = torch.max(eaat(grid), 1)[-1].numpy().reshape(x0.shape)
      cs = plt.contour(x0, x1, eaat_pred, levels=[0.5])
      h2, l2 = cs.legend_elements("eaat")
```

(continues on next page)

(continued from previous page)

```
plt.legend(h1 + h2, l1 + l2)
```

[11]:
```
<matplotlib.legend.Legend at 0x7f09007b5e20>
```



[ ]:

5.3 MNIST Example

For this demo, we load the MNIST (handwritten digits) dataset using torchvision, define a simple convolutional architecture, and train a prediction model using the exponential average adversarial training technique (EAAT) with 10% of the MNIST labels. This example is meant as a quick-start guide and to reinforce what is provided in the documentation.

[1]:
```python
# torch imports
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

# shadow-ssml imports
import shadow.eaat
from shadow.utils import set_seed

# helpers
import numpy as np
import random
```

Torchvision makes it easy to load and perform standard preprocessing operations on a variety of data transforms. Instead of using the MNIST class for the fully-labeled training datasets, we define our own MNIST class to return partially labeled (labeled and unlabeled) training data. Then we define our dataset for training as the MNIST training data with 90% of the labels reassigned to a value to -1 using a consistent sampling seed. Lastly, we use the standard torchvision MNIST class test partition, keeping all labels, for evaluation of SSL classification performance.

[2]:
```
datadir = 'data'
set_seed(0)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


class UnlabeledMNIST(torchvision.datasets.MNIST):
    def __init__(self, root, train=True,
                 transform=torchvision.transforms.ToTensor(),
                 download=False, unlabeled_frac=0.9):
        super(UnlabeledMNIST, self).__init__(root,
                 train=train, transform=transform,
                 download=download)
        labels_to_drop = np.random.choice(len(self),
                 size=int(len(self) * unlabeled_frac),
                 replace=False)
        self.targets[labels_to_drop] = -1


dataset = UnlabeledMNIST(datadir, train=True, download=True,
                         transform=torchvision.transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(dataset, batch_size=100)

test_loader = torch.utils.data.DataLoader(torchvision.datasets.MNIST(
    datadir, train=False, download=True,
    transform=torchvision.transforms.ToTensor()),
    batch_size=100, shuffle=True)
```

[3]:
```
print(dataset)
```

```
Dataset UnlabeledMNIST
    Number of datapoints: 60000
    Root location: data
    Split: Train
    StandardTransform
Transform: ToTensor()
```

Next we define our parameter dictionary for non-default parameters used by the EAAT technique. For example, we rarely require more than one power iteration to compute the adversarial direction. Likewise, we maintain defaults for student and teacher noise. As a reminder, EAAT is a combination of exponential averaging, which uses random gaussian perturbations, and adversarial training, which uses data-specific adversarial perturbations. If your dataset may benefit from additive noise AND adversarial perturbations, the EAAT parameters {student_noise, teacher_noise} would be included in the model and in hyperparameter searches.

[4]:
```
eaatparams = {
        "xi": 1e-8,
        "eps": 2.3,
        }
```

Here we define a simple convolutional architecture with Relu and Dropout. Forward, in this case, does not return Softmax on the final layer. Typically the loss for each technique implements Softmax scaling. We then instantiate the model and the optimizer.

[5]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
```

(continues on next page)

(continued from previous page)

```
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout = nn.Dropout2d(0.1)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x


model = Net()
eaat = shadow.eaat.EAAT(model=model, **eaatparams)
optimizer = optim.SGD(eaat.parameters(), lr=0.01)
```

At this point, we have partially-labeled training data available through our train_loader and fully-labeled testing data from test_loader. We have initialized a model, specified that we plan to use EAAT, and passed the EAAT parameters to the model. The last step is to train the model. The loss function for the SSL techniques implemented here is a combination of the loss on labeled data, where we typically use cross-entropy, and the technique-specific consistency cost. We specify the labeled data cost (xEnt), ignoring labels of -1, which we used as the unlabeled target values. During training, we give the labeled loss and the consistency loss equal weight by simply adding them together.

```
[6]: xEnt = torch.nn.CrossEntropyLoss(ignore_index=-1)

eaat.to(device)
losscurve = []
for epoch in range(10):
    eaat.train()
    lossavg = []
    for i, (data, targets) in enumerate(train_loader):
        x = data.to(device)
        y = targets.to(device)
        optimizer.zero_grad()
        out = eaat(x)
        loss = xEnt(out, y) + eaat.get_technique_cost(x)
        loss.backward()
        optimizer.step()
        lossavg.append(loss.item())
    losscurve.append(np.median(lossavg))
    print('epoch {} loss: {}'.format(epoch, losscurve[-1]))
```

```
epoch 0 loss: 1.6615383625030518
epoch 1 loss: 1.2582014799118042
epoch 2 loss: 1.0733909010887146
epoch 3 loss: 0.9297202229499817
epoch 4 loss: 0.8314944803714752
epoch 5 loss: 0.7584533393383026
epoch 6 loss: 0.6920907497406006
epoch 7 loss: 0.6233154237270355
```

(continues on next page)

(continued from previous page)

```
epoch 8 loss: 0.5829548835754395
epoch 9 loss: 0.5472914576530457
```

After training, we evaluate the performance over our test set.

[7]:
```
eaat.eval()
y_pred, y_true = [], []
for i, (data, targets) in enumerate(test_loader):
    x = data.to(device)
    y = targets.to(device)
    out = eaat(x)
    y_true.extend(y.detach().cpu().tolist())
    y_pred.extend(torch.argmax(out, 1).detach().cpu().tolist())
test_acc = (np.array(y_true) == np.array(y_pred)).mean() * 100
print('test accuracy: {}'.format(test_acc))
```

```
test accuracy: 96.33
```

[ ]:
```
```

5.4 API Documentation

This page provides the full API documentation for the Shadow package.

5.4.1 shadow.eaat module

class shadow.eaat.**EAAT**(*model*, *alpha=0.999*, *student_noise=0.1*, *teacher_noise=0.1*, *xi=1.0*, *eps=1.0*, *power_iter=1*, *consistency_type='kl'*, *flip_correction=True*)

Bases: *shadow.mt.MT*

Exponential Averaging Adversarial Training (EAAT, [Linville21]) model wrapper for consistency regularization.

Computes consistency using the teacher-student paradigm followed by Mean-Teacher ([Tarvainen17]) with virtual adversarial perturbations ([Miyato18]) applied to student model inputs.

Parameters

- **model** (*torch.nn.Module*) – The student model.
- **alpha** (*float, optional*) – The teacher exponential moving average smoothing coefficient. Defaults to 0.999.
- **student_noise** (*float, optional*) – If > 0.0, the standard deviation of gaussian noise to apply to the student input (in addition to virtual adversarial noise). Specifically, generates random numbers from a normal distribution with mean 0 and variance 1, and then scales them by this factor and add to the input data. Defaults to 0.1.
- **teacher_noise** (*float, optional*) – If > 0.0, the standard deviation of gaussian noise to apply to the teacher input. Specifically, generates random numbers from a normal distribution with mean 0 and variance 1, and then scales them by this factor and add to the input data. Defaults to 0.1.
- **xi** (*float, optional*) – Scaling value for the random direction vector. Defaults to 1.0

- **eps** (*float, optional*) – The magnitude of applied adversarial perturbation. Greater *eps* implies more smoothing. Defaults to 1.0
- **power_iter** (*int, optional*) – Number of power iterations used to estimate virtual adversarial direction. Per [Miyato18], defaults to 1.
- **consistency_type** (*{'kl', 'mse', 'mse_regress'}, optional*) – Cost function used to measure consistency. Defaults to *'kl'* (KL-divergence).
- **flip_correction** (*bool, optional*) – Correct flipped virtual adversarial perturbations induced by power iteration estimation. These iterations sometimes converge to a "flipped" perturbation (away from maximum change in consistency). This correction detects this behavior and corrects flipped perturbations at the cost of slightly increased compute. This behavior is not included in the original VAT implementation, which exhibits perturbation flipping without any corrections. Defaults to *True*.

calc_student_logits (*x*)
Student model logits, with perturbations added to the input data.

Parameters **x** (*torch.Tensor*) – Input data.

Returns The student logits.

Return type torch.Tensor

5.4.2 shadow.losses module

shadow.losses.**accuracy** (*y_pred, y*)
Classification accuracy.

Parameters

- **y_pred** (*array_like*) – Predicted labels.
- **y** (*array_like*) – True labels.

Returns Classification accuracy percentage.

Return type float

shadow.losses.**mse_regress_loss** (*y_pred, y_true, reduction='sum'*)
Measures the element-wise mean squared error (squared L2 norm) between two model outputs.

Directly passes *y_pred*, *y_true*, and *reduction* to torch.nn.function.mse_loss. *mse_regress_loss* differs from *softmax_mse_loss* in that it does not compute the softmax and therefore makes it applicable to regression tasks.

Parameters

- **y_pred** (*torch.Tensor*) – The predicted labels.
- **y_true** (*torch.Tensor*) – The target labels.
- **reduction** (*string, optional*) – The reduction parameter passed to torch.nn.functional.mse_loss. Defaults to 'sum'.

Returns Mean squared error.

Return type torch.Tensor

shadow.losses.**softmax_kl_loss** (*input_logits, target_logits, reduction='sum'*)
Apply softmax and compute KL divergence between two model outputs.

Parameters

- **input_logits** (*torch.Tensor*) – The input unnormalized log probabilities.

- **target_logits** (*torch.Tensor*) – The target unnormalized log probabilities.
- **reduction** (*string, optional*) – The reduction parameter passed to torch.nn.functional.kl_div. Defaults to 'sum'.

Returns KL divergence.

Return type torch.Tensor

`shadow.losses.`**`softmax_mse_loss`**(*input_logits*, *target_logits*, *reduction='sum'*)

Apply softmax and compute mean square error between two model outputs.

Parameters

- **input_logits** (*torch.Tensor*) – The input unnormalized log probabilities.
- **target_logits** (*torch.Tensor*) – The target unnormalized log probabilities.
- **reduction** (*string, optional*) – The reduction parameter passed to torch.nn.functional.mse_loss. Defaults to 'sum'.

Returns Softmax mean squared error.

Return type torch.Tensor

5.4.3 shadow.module_wrapper module

class `shadow.module_wrapper.`**`ModuleWrapper`**(*model*)

Bases: `torch.nn.Module`

Base module wrapper for SSML technique implementations.

Parameters model (*torch.nn.Module*) – The model to train with semi-supervised learning.

forward(*x*)

Passes data to the wrapped model.

Parameters x (*torch.Tensor*) – Input data.

Returns Model output.

Return type torch.Tensor

get_technique_cost(*x*)

Compute the SSML related cost for the implemented technique.

Parameters x (*torch.Tensor*) – Input data.

Returns Technique specific cost.

Return type torch.Tensor

Raises NotImplementedError – If not implemented in the specific technique.

Note: This must be implemented for each specific technique that inherits from this base.

5.4.4 shadow.mt module

class shadow.mt.**MT**(*model*, *alpha=0.999*, *noise=0.1*, *consistency_type='mse'*)

Bases: *shadow.module_wrapper.ModuleWrapper*

Mean Teacher [Tarvainen17] model wrapper for consistency regularization.

Mean Teacher model wrapper the provides both student and teacher model implementation. The teacher model is a running average of the student weights, and is updated during training. When switched to eval mode, the teacher model is used for predictions instead of the student. As the wrapper handles the hand off between student and teacher models, the wrapper should be used instead of the student model directly.

Parameters

- **model** (*torch.nn.Module*) – The student model.
- **alpha** (*float, optional*) – The teacher exponential moving average smoothing coefficient. Defaults to 0.999.
- **noise** (*float, optional*) – If > 0.0, the standard deviation of gaussian noise to apply to the input. Specifically, generates random numbers from a normal distribution with mean 0 and variance 1, and then scales them by this factor and adds to the input data. Defaults to 0.1.
- **consistency_type** (*{'kl', 'mse', 'mse_regress'}, optional*) – Cost function used to measure consistency. Defaults to *'mse'* (mean squared error).

calc_student_logits(*x*)

Student model logits, with noise added to the input data.

Parameters **x** (*torch.Tensor*) – Input data.

Returns The student logits.

Return type torch.Tensor

calc_teacher_logits(*x*)

Teacher model logits.

The teacher model logits, with noise added to the input data. Does not propagate gradients in the teacher forward pass.

Parameters **x** (*torch.Tensor*) – Input data.

Returns The teacher logits.

Return type torch.Tensor

forward(*x*)

Model forward pass.

During model training, adds noise to the input data and passes through the student model. During model evaluation, does not add noise and passes through the teacher model.

Parameters **x** (*torch.Tensor*) – Input data.

Returns Model output.

Return type torch.Tensor

get_evaluation_model()

The teacher model, which should be used for prediction during evaluation.

Returns The teacher model.

Return type torch.nn.Module

get_technique_cost (*x*)

Consistency cost between student and teacher models.

Consistency cost between the student and teacher, updates teacher weights via exponential moving average of the student weights. Noise is sampled and applied to student and teacher separately.

Parameters **x** (*torch.Tensor*) – Input data.

Returns Consistency cost between the student and teacher model outputs.

Return type torch.Tensor

update_ema_model ()

Exponential moving average update of the teacher model.

shadow.mt.**ema_update_model** (*student_model*, *ema_model*, *alpha*, *global_step*)

Exponential moving average update of a model.

Update *ema_model* to be the moving average of consecutive *student_model* updates via an exponential weighting (as defined in [Tarvainen17]). Update is performed in-place.

Parameters

- **student_model** (*torch.nn.Module*) – The student model.
- **ema_model** (*torch.nn.Module*) – The model to update (teacher model). Update is performed in-place.
- **alpha** (*float*) – Exponential moving average smoothing coefficient, between [0, 1].
- **global_step** (*int*) – A running count of exponential update steps (typically mini-batch updates).

5.4.5 shadow.pseudo module

class shadow.pseudo.**PL** (*model*, *weight_function*, *ssml_mode=True*, *missing_label=-1*)

Bases: *shadow.module_wrapper.ModuleWrapper*

Pseudo Label model wrapper.

The pseudo labeling wrapper weight samples according to model score. This is a form of entropy regularization. For example, a binary random variable with distribution $P(X = 1) = .5$ and $P(X = 0) = .5$ has a much higher entropy than $P(X = 1) = .9$ and $P(X = 0) = .1$.

Parameters

- **weight_function** (*callable*) – assigns weighting based on raw model outputs.
- **ssml_mode** (*bool, optional*) – semi-supevised learning mode, toggles whether loss is computed for all inputs or just those data with missing labels. Defaults to True.
- **missing_label** (*int, optional*) – integer value used to represent missing labels. Defaults to -1.

get_technique_cost (*x*, *targets*)

Compute loss from pseudo labeling.

Parameters

- **x** (*torch.Tensor*) – Tensor of the data
- **targets** (*torch.Tensor*) – 1D Corresponding labels. Unlabeled data is specified according to *self.missing_label*.

Returns Pseudo label loss.

Return type torch.Tensor

class shadow.pseudo.**Threshold**(*thresholds*)
Bases: torch.nn.Module

Per-class thresholding operator.

Parameters **threshold** (*torch.Tensor*) – 1D *float* array of thresholds with length equal to the number of classes. Each element should be between $[0, 1]$ and represents a per-class threshold. Thresholds are with respect to normalized scores (e.g. they sum to 1).

Example

```
>>> myThresholder = Threshold([.8, .9])
>>> myThresholder([[10, 90], [95, 95.4], [0.3, 0.4]])
[1, 0, 0]
```

forward(*predictions*)
Threshold multi-class scores.

Parameters **predictions** (*torch.Tensor*) – 2D model outputs of shape *(n_samples, n_classes)*. Does not need to be normalized in advance.

Returns binary thresholding for each sample.

Return type torch.Tensor

5.4.6 shadow.utils module

class shadow.utils.**ConstantCW**(*last_weight=1*)
Bases: shadow.utils._CWScheduler

Constant valued consistency weight scheduler.

Scheduler function to control a weight, often used to weigh a consistency cost relative to a supervised learning cost (e.g. Cross Entropy). This is intended to be stepped after each epoch during training to increase or decrease the weight accordingly. This provides a constant weighting function that does not change.

Parameters **last_weight** (*float, optional*) – Final consistency weight. Defaults to 1.

Example

```
>>> alpha = ConstantCW(last_weight)
>>> for epoch in epochs:
>>>     train(...)
>>>     loss = criterion + alpha() * consistency
>>>     validate(...)
>>>     alpha.step()
```

class shadow.utils.**IgnoreUnlabeledWrapper**(*criterion, ignore_index=numpy.NINF*)
Bases: torch.nn.Module

Wraps a loss function to filter out mising values for a Semi-Supervised learning task.

Parameters

- **criterion** (*callable*) – Used to compute the supervised loss.

- **ignore_index** (*bool, int, float, complex, optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. Defaults to negative infinity.

Example

```
>>> ssml_loss = IgnoreUnlabeledWrapper(criterion=torch.nn.MSELoss())
>>> y_true = torch.rand(3, 1)
>>> y_hat = y_true.clone()
>>> y_hat
tensor([[0.1543],
        [0.1572],
        [0.0404]])
>>> ssml_loss(y_hat, y_true)
tensor(0.)
>>> y_true[1] = np.NINF
>>> y_true
tensor([[0.1543],
        [  -inf],
        [0.0404]])
>>> ssml_loss(y_hat, y_true)
tensor(0.)
```

Example

```
>>> ssml_loss = IgnoreUnlabeledWrapper(criterion=torch.nn.BCELoss())
>>> y_hat = torch.Tensor([[0], [1], [1], [0]])
>>> y_true = torch.Tensor([[ignore_index], [1], [ignore_index], [1]])
>>> ssml_loss(y_hat, y_true)
tensor(50.)
```

forward (*y_hat, y_true*)

class shadow.utils.**QuadraticCW** (*last_epoch, last_weight=1, first_weight=0, epochs_before=0*)

Bases: `shadow.utils._CWScheduler`

Quadratic consistency weight scheduler.

Scheduler function to control a weight, often used to weigh a consistency cost relative to a supervised learning cost (e.g. Cross Entropy). This is intended to be stepped after each epoch during training to increase or decrease the weight accordingly. This provides a quadratic weighting function.

Parameters

- **last_epoch** (*int*) – Number of epochs until scheduler reaches *last_weight*.
- **last_weight** (*float, optional*) – Final consistency weight. Defaults to 1.
- **first_weight** (*float, optional*) – Consistency weight at beginning of ramp. Defaults to 0.
- **epochs_before** (*int, optional*) – Number of epochs to hold weight at *first_weight* before beginning ramp. Defaults to 0.

Example

```
>>> alpha = QuadraticCW(last_epoch, last_weight, first_weight, epochs_before)
>>> for epoch in epochs:
>>>     train(...)
>>>     loss = criterion + alpha() * consistency
>>>     validate(...)
>>>     alpha.step()
```

class `shadow.utils.`**`RampCW`**(*last_epoch*, *last_weight=1*, *first_weight=0*, *epochs_before=0*)
Bases: `shadow.utils._CWScheduler`

Linear ramp consistency weight scheduler.

Scheduler function to control a weight, often used to weigh a consistency cost relative to a supervised learning cost (e.g. Cross Entropy). This is intended to be stepped after each epoch during training to increase or decrease the weight accordingly. This provides a linear ramp weighting function.

Parameters

- **last_epoch** (*int*) – Number of epochs until scheduler reaches *last_weight*.
- **last_weight** (*float, optional*) – Final consistency weight. Defaults to 1.
- **first_weight** (*float, optional*) – Consistency weight at beginning of ramp. Defaults to 0.
- **epochs_before** (*int, optional*) – Number of epochs to hold weight at *first_weight* before beginning ramp. Defaults to 0.

Example

```
>>> alpha = RampCW(last_epoch, last_weight, first_weight, epochs_before)
>>> for epoch in epochs:
>>>     train(...)
>>>     loss = criterion + alpha() * consistency
>>>     validate(...)
>>>     alpha.step()
```

class `shadow.utils.`**`SigmoidCW`**(*last_epoch*, *last_weight=1*, *first_weight=0*, *epochs_before=0*)
Bases: `shadow.utils._CWScheduler`

Sigmoidal consistency weight scheduler.

Scheduler function to control a weight, often used to weigh a consistency cost relative to a supervised learning cost (e.g. Cross Entropy). This is intended to be stepped after each epoch during training to increase or decrease the weight accordingly. This provides a sigmoidal weighting function.

Parameters

- **last_epoch** (*int*) – Number of epochs until scheduler reaches *last_weight*.
- **last_weight** (*float, optional*) – Final consistency weight. Defaults to 1.
- **first_weight** (*float, optional*) – Consistency weight at beginning of ramp. Defaults to 0.
- **epochs_before** (*int, optional*) – Number of epochs to hold weight at *first_weight* before beginning ramp. Defaults to 0.

Example

```
>>> alpha = SigmoidCW(last_epoch, last_weight, first_weight, epochs_before)
>>> for epoch in epochs:
>>>     train(...)
>>>     loss = criterion + alpha() * consistency
>>>     validate(...)
>>>     alpha.step()
```

class `shadow.utils.`**`SkewedSigmoidCW`**(*last_epoch*, *last_weight=1*, *first_weight=0*, *epochs_before=0*, *beta=1*, *zeta=1*)
Bases: `shadow.utils._CWScheduler`

Skewed sigmoidal consistency weight scheduler with variable ramp up speed.

Scheduler function to control a weight, often used to weigh a consistency cost relative to a supervised learning cost (e.g. Cross Entropy). This is intended to be stepped after each epoch during training to increase or decrease the weight accordingly. This provides a skewed sigmoid weighting function with variable ramp up timing speed.

Parameters

- **last_epoch** (*int*) – Number of epochs until scheduler reaches *last_weight*.
- **last_weight** (*float, optional*) – Final consistency weight. Defaults to 1.
- **first_weight** (*float, optional*) – Consistency weight at beginning of ramp. Defaults to 0.
- **epochs_before** (*int, optional*) – Number of epochs to hold weight at *first_weight* before beginning ramp. Defaults to 0.
- **beta** (*float, optional*) – Controls how sharp the rise from *first_weight* to *last_weight* is. *beta* = 1 corresponds to a standard sigmoid. Increasing *beta* increases sharpness. Negative values can actually invert the sigmoid for a decreasing ramp. Defaults to 1.
- **zeta** (*float, optional*) – Skews when the rise from *first_weight* to *last_weight* occurs. *zeta* = 1 corresponds to a rise centered about the middle epoch. *zeta* = 0 corresponds to a flat weight at *last_weight*. *zeta* < 1 shifts rise to earlier epochs. *zeta* > 1 shifts to later epochs. Defaults to 1.

Example

```
>>> alpha = SkewedSigmoidCW(last_epoch, last_weight, first_weight, epochs_before,␣
↪beta, zeta)
>>> for epoch in epochs:
>>>     train(...)
>>>     loss = criterion + alpha() * consistency
>>>     validate(...)
>>>     alpha.step()
```

class `shadow.utils.`**`StepCW`**(*last_epoch*, *last_weight=1*, *first_weight=0*)
Bases: `shadow.utils._CWScheduler`

Step function consistency weight scheduler.

Scheduler function to control a weight, often used to weigh a consistency cost relative to a supervised learning cost (e.g. Cross Entropy). This is intended to be stepped after each epoch during training to increase or decrease the weight accordingly. This provides a step weighting function.

Parameters

- **last_epoch** (*int*) – Number of epochs until scheduler reaches *last_weight*.
- **last_weight** (*float, optional*) – Final consistency weight. Defaults to 1.
- **first_weight** (*float, optional*) – Consistency weight at beginning of ramp. Defaults to 0.

Example

```
>>> alpha = StepCW(last_epoch, last_weight, first_weight)
>>> for epoch in epochs:
>>>     train(...)
>>>     loss = criterion + alpha() * consistency
>>>     validate(...)
>>>     alpha.step()
```

shadow.utils.**flatten_to_two_dim**(*input_tensor*)

Flatten tensor along the first axis ([2, 3, 4] -> [2, 12])

Parameters **input_tensor** (*torch.Tensor*) – input tensor

Returns *input_tensor* flattened along first axis

Return type torch.Tensor

shadow.utils.**init_model_weights**(*model, value*)

Set all weights in model to a given value.

Parameters

- **model** (*torch.nn.Module*) – The model to update. Weight update is performed in place.
- **value** (*float*) – The weight value.

shadow.utils.**set_seed**(*seed, cudnn_deterministic=False*)

Sets the seeds for max reproducibility.

Sets seeds for random, numpy, and torch to *seed*, and can also enable deterministic mode for the CuDNN backend. This does not guarantee full reproducibility as some underlying options (e.g. *atomicAdd*) still have sources of non-determinism that cannot be disabled.

Parameters

- **seed** (*int*) – Seed used for *random*, *numpy*, and *torch*.
- **cudnn_deterministic** (*bool, optional*) – Sets the CuDNN backend into deterministic mode. This can negatively impact performance. Defaults to False.

Note: PyTorch provides only minimal guarantees on reproducibility. See <https://pytorch.org/docs/stable/notes/randomness.html> for more information.

5.4.7 shadow.vat module

class `shadow.vat.`**`RPT`**(*eps*, *model*, *consistency_type='mse'*)
Bases: `shadow.module_wrapper.ModuleWrapper`

Random Perturbation Training for consistency regularization.

Random Perturbation Training (RPT) is a special case of Virtual Adversarial Training (VAT, [Miyato18]) for which the number of power iterations is 0. This means that added perturbations are isotropically random (not in the adversarial direction).

Parameters

- **eps** (*float*) – The magnitude of applied perturbation. Greater *eps* implies more smoothing.
- **model** (*torch.nn.Module*) – The model to train and regularize.
- **consistency_type** (*{'kl', 'mse', 'mse_regress'}, optional*) – Cost function used to measure consistency. Defaults to *'kl'* (KL-divergence).

get_technique_cost(*x*)
Consistency cost (local distributional smoothness).

Parameters **x** (*torch.Tensor*) – Tensor of the data

Returns Consistency cost between the data and randomly perturbed data.

Return type torch.Tensor

class `shadow.vat.`**`VAT`**(*model*, *xi=1.0*, *eps=1.0*, *power_iter=1*, *consistency_type='kl'*, *flip_correction=True*, *xi_check=False*)
Bases: `shadow.module_wrapper.ModuleWrapper`

Virtual Adversarial Training (VAT, [Miyato18]) model wrapper for consistency regularization.

Parameters

- **model** (*torch.nn.Module*) – The model to train and regularize.
- **xi** (*float, optional*) – Scaling value for the random direction vector. Defaults to 1.0.
- **eps** (*float, optional*) – The magnitude of applied adversarial perturbation. Greater *eps* implies more smoothing. Defaults to 1.0.
- **power_iter** (*int, optional*) – Number of power iterations used to estimate virtual adversarial direction. Per [Miyato18], defaults to 1.
- **consistency_type** (*{'kl', 'mse', 'mse_regress'}, optional*) – Cost function used to measure consistency. Defaults to *'kl'* (KL-divergence).
- **flip_correction** (*bool, optional*) – Correct flipped virtual adversarial perturbations induced by power iteration estimation. These iterations sometimes converge to a "flipped" perturbation (away from maximum change in consistency). This correction detects this behavior and corrects flipped perturbations at the cost of slightly increased compute. This behavior is not included in the original VAT implementation, which exhibits perturbation flipping without any corrections. Defaults to *True*.
- **xi_check** (*bool, optional*) – Raise warnings for small perturbations lengths. It should be selected so as to be small (for algorithm assumptions to be correct), but not so small as to collapse the perturbation into a length 0 vector. This parameter controls optional warnings to detect a value of *xi* that causes perturbations to collapse to length 0. Defaults to *False*.

get_technique_cost (*x*)
VAT consistency cost (local distributional smoothness).

Parameters **x** (*torch.Tensor*) – Tensor of the data

Returns Consistency cost between the data and virtual adversarially perturbed data.

Return type torch.Tensor

shadow.vat.**adv_perturbation** (*x*, *y*, *model*, *criterion*, *optimizer*)
Find adversarial perturbation following [Goodfellow14].

Parameters

- **x** (*torch.Tensor*) – Input data.
- **y** (*torch.Tensor*) – Input labels.
- **model** (*torch.nn.Module*) – The model.
- **criterion** (*callable*) – The loss criterion used to measure performance.
- **optimizer** (*torch.optim.Optimizer*) – Optimizer used to compute gradients.

Returns Adversarial perturbations.

Return type torch.Tensor

shadow.vat.**l2_normalize** (*r*)
L2 normalize tensor, flattening over all but batch dim (0).

Parameters **r** (*torch.Tensor*) – Tensor to normalize.

Returns Normalized tensor (over all but dim 0).

Return type torch.Tensor

shadow.vat.**rand_unit_sphere** (*x*)
Draw samples from the uniform distribution over the unit sphere.

Parameters **x** (*torch.Tensor*) – Tensor used to define shape and dtype for the generated tensor.

Returns Random unit sphere samples.

Return type torch.Tensor

Reference: https://stats.stackexchange.com/questions/7977/how-to-generate-uniformly-distributed-points-on-the-surface-of-the

shadow.vat.**vadv_perturbation** (*x*, *model*, *xi*, *eps*, *power_iter*, *consistency_criterion*, *flip_correction=True*, *xi_check=False*)
Find virtual adversarial perturbation following [Miyato18].

Parameters

- **x** (*torch.Tensor*) – Input data.
- **model** (*torch.nn.Module*) – The model.
- **xi** (*float*) – Scaling value for the random direction vector.
- **eps** (*float*) – The magnitude of applied adversarial perturbation. Greater *eps* implies more smoothing.
- **power_iter** (*int*) – Number of power iterations used in estimation.
- **consistency_criterion** (*callable*) – Cost function used to measure consistency.

- **flip_correction** (*bool, optional*) – Correct flipped virtual adversarial perturbations induced by power iteration estimation. These iterations sometimes converge to a "flipped" perturbation (away from maximum change in consistency). This correction detects this behavior and corrects flipped perturbations at the cost of slightly increased compute. This behavior is not included in the original VAT implementation, which exhibits perturbation flipping without any corrections. Defaults to *True*.
- **xi_check** (*bool, optional*) – Raise warnings for small perturbations lengths. The parameter *xi* should be selected so as to be small (for algorithm assumptions to be correct), but not so small as to collapse the perturbation into a length 0 vector. This parameter controls optional warnings to detect a value of *xi* that causes perturbations to collapse to length 0. Defaults to *False*.

Returns Virtual adversarial perturbations.

Return type torch.Tensor

5.5 Copyright and License

5.5.1 Copyright

5.5.2 New BSD License

5.6 Contributing

We welcome all contributions including bug fixes, feature enhancements, and documentation improvments. Shadow manages source code contributions via GitHub pull requests (PRs).

5.6.1 Coding Standards

- 4 space indentation (no tabs) and PEP8 conformance
- No use of __author__
- Documentation must be in Sphinx-compliant format.
- Submitted code should follow standard practices for documentation and testing.
- Automated testing (using `pytest` and TravisCI) must pass prior to merging.

5.7 References

CHAPTER

SIX

CONTRIBUTORS

- Dylan Anderson
- Lisa Linville
- Joshua Michalenko
- Jennifer Galasso
- Brian Evans
- Henry Qiu
- Christopher Murzyn
- Brodderick Rodriguez

CHAPTER SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

BIBLIOGRAPHY

[Linville21] Linville, L., Anderson, D., Michalenko, J., Galasso, J., & Draelos, T. (2021). Semisupervised Learning for Seismic Monitoring Applications. Seismological Society of America, 92(1), 388-395. doi:https://doi.org/10.1785/0220200195

[Miyato18] Miyato, Takeru, et al. "Virtual adversarial training: a regularization method for supervised and semi-supervised learning." IEEE transactions on pattern analysis and machine intelligence 41.8 (2018): 1979-1993.

[Tarvainen17] Tarvainen, Antti, and Harri Valpola. "Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results." Advances in neural information processing systems. 2017.

[Oliver18] Oliver, Avital, et al. "Realistic evaluation of semi-supervised learning algorithms." (2018).

[Rasmus15] Rasmus, Antti, et al. "Semi-supervised learning with ladder networks." Advances in neural information processing systems. 2015.

[Goodfellow14] Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples." arXiv preprint arXiv:1412.6572 (2014).

PYTHON MODULE INDEX

INDEX