BackwardCompatibilityML

Release 1.1.0

Oct 09, 2020

Contents

CHAPTER 1

Project Overview

Updates that may improve an AI system's accuracy can also introduce new and unanticipated errors that damage user trust. Updates that introduce new errors can also break trust between software components and machine learning models, as these errors are propagated and compounded throughout larger integrated AI systems. The Backward Compatibility ML library is an open-source project for evaluating AI system updates in a new way for increasing system reliability and human trust in AI predictions for actions.

In this project, we define an update to an AI component to be compatible when it does not disrupt an experienced user's insights and expectations—or mental model—of how the classifier works. An update is considered compatible *only if* the updated model recommends the same correct action as recommended by the previous version, which received the same input. A compatible update supports the user's mental model and maintains trust.

Compatibility is both a usability and engineering concern. This project's series of loss functions provides important metrics that extend beyond the single score of accuracy. These support ML practitioners in navigating performance and tradeoffs in system updates. The functions integrate easily into existing AI model-training workflows. Simple visualizations, such as Venn diagrams, further help practitioners compare models and explore performance and compatibility tradeoffs for informed choices.

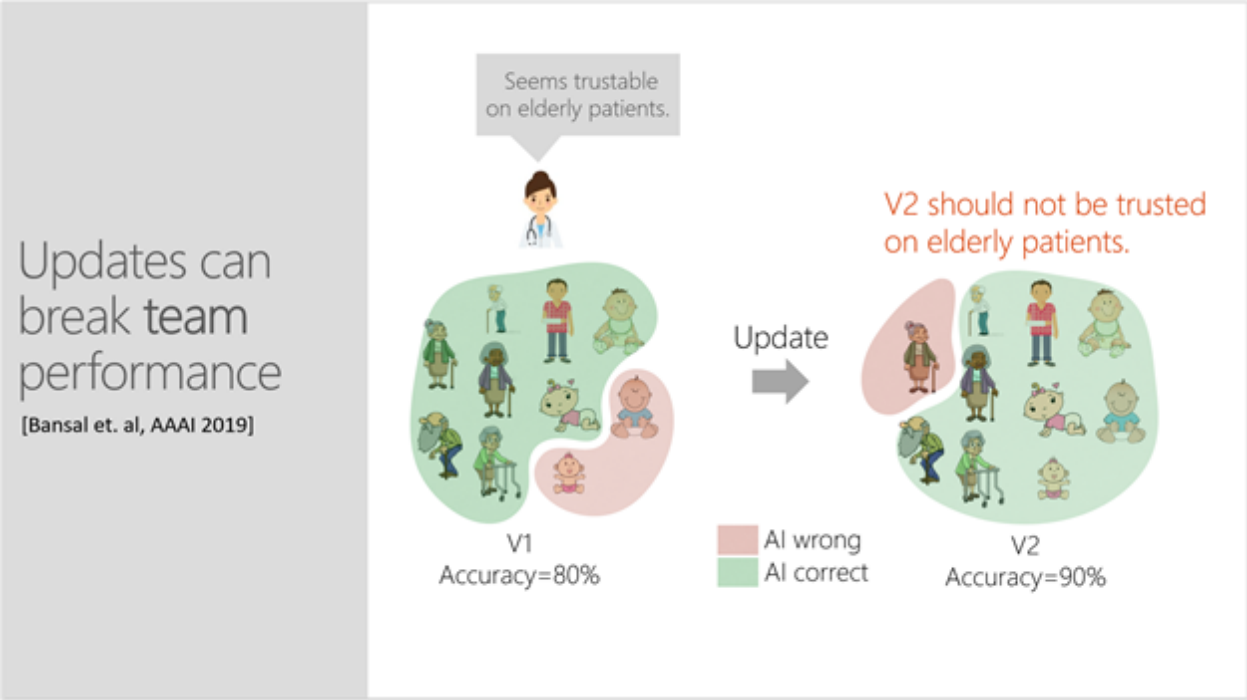
1.1 Building trust in human-AI teams

After repeated experience with an AI system, users develop insights and expectations, a mental model, of the system's competence. The success of human-AI partnerships is dependent on people knowing whether to trust the AI or override it. This is critically important as AI systems are used to augment human decision making in high-stakes domains such as, for example, healthcare, criminal justice, or transportation.

A problem arises when developers regularly update AI systems with improved training data or algorithms: Updates that may improve an AI's predictive performance can also introduce new and unexpected errors that breach the end-users' trust in the AI.

For example, a doctor uses a classifier to predict whether an elderly patient will be readmitted to the hospital shortly after being discharged. Based on the AI's prediction and her own experience, she must decide if the patient should be placed in an outpatient program to avoid readmission. The doctor has interacted with the model quite a few times and knows that it is 80% accurate. Having learned the error boundary, she has concluded that the model is trustworthy for

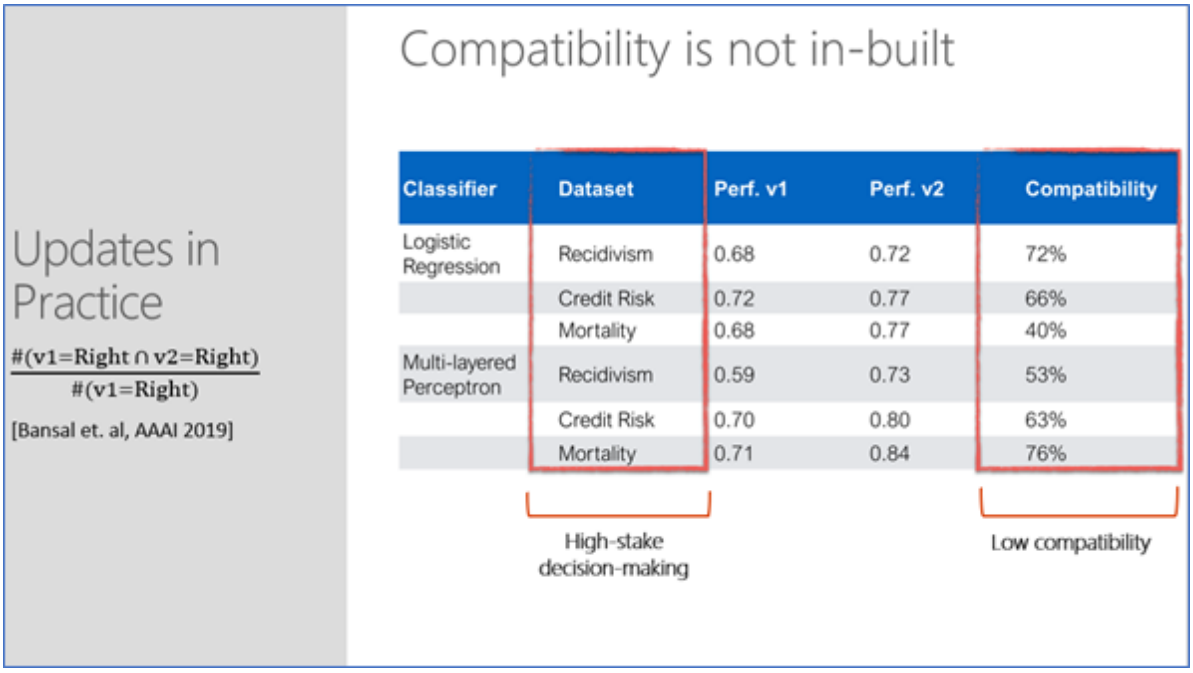
elderly patients. However, she is unaware that an update, which has made the model 90% accurate, now introduces errors for elderly patients and should not be trusted for this population. This puts the doctor—who is relying on an outdated mental model—at risk of making a wrong decision for her patient and will undermine her trust in the AI's future recommendations.



Updates that may improve an AI system's predictive performance can also introduce new and unexpected errors that breach end-users' trust and damage the effectiveness of human-AI teams. Here, a doctor is not yet aware that an update, which increased a model's accuracy, now introduces errors for elderly patients and should not be trusted when making decisions for this population.

1.2 Identifying unreliability problems in an update

It is helpful to understand that compatibility is not inbuilt, and that measuring backward compatibility can identify unreliability issues during an update. As shown in the table below, experimenting with three different datasets in high-stakes decision making (predicting recidivism, credit risk, and mortality) by *updating with a larger training set only,* there are cases where compatibility is as low as 40%. This means the model is now making a mistake in 60% of the cases it was getting right before the update.



1.3 Maintaining component-to-component trust

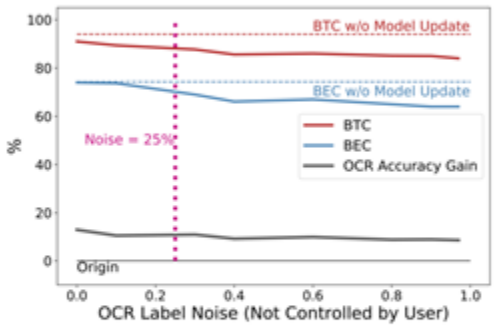
An incompatible update can also break trust with other software components and machine learning models that are not able to handle new errors. They instead propagate and compound these new errors throughout complex systems. Measuring backward compatibility can identify unreliability issues during an update and help ML practitioners control for backward compatibility to avoid downstream degradation.

For example, a financial services team uses an off-the-shelf OCR model to detect receipt fraud in expense reports. They have developed a heuristic blacklist component of spoofed company names (e.g., "Nlke" vs. "Nike" or "G00gle" vs. "Google"), which works well with the OCR model. Developers, with the aim of improving model performance for a wider variety of fonts, update the model with a noisy dataset of character images scraped from the internet, which people have labelled through CAPTCHA tasks. Common human annotation mistakes of confusing "l" for "i" or "0" for "o" now unexpectedly reduce the classifier's ability to discriminate between spoofed and legitimate business names, which can lead to costly system failures.

As shown in the image below, developers can use two separate measures of backward compatibility for evaluating and avoiding downstream failures: Backward Trust Compatibility (BTC), which describes the percentage of trust preserved after an update, and Backward Error Compatibility (BEC), which captures the probability that a mistake made by the newly trained model is not new. The 89% BTC and 71% BEC scores show a decrease in backward compatibility compared with the baseline.

Backward Compatibility Analysis

[Srivastava et al., KDD 2020]



BTC: Backward Trust Compatibility describes the percentage of trust preserved after an update.

BEC: Backward Error Compatibility captures the probability that a mistake made by the newly trained model is not new.

In this example, above, while the overall accuracy of word recognition might improve after the model update, the performance of the system on specific words in the blacklist heuristics may degrade significantly. Additionally, with backward compatibility analysis, seeing the distribution of incompatibility can be a useful guide for pinpointing where there are problems with the data.

Below illustrates how a holistic view of decreases in performance enable users to monitor incompatibility beyond examples that are explicitly impacted by noise. Here, the uppercase "Z" is often among incompatible points, even though it is not directly influenced by noise.

Backward Compatibility Analysis

[Srivastava et al., KDD 2020]

OCR pipeline component

Incompatible OCR examples (25% noise)

0	l	Z
True Label: Digit 0	True Label: lower L	True Label: upper Z
19% of incompatible points	16% of incompatible points	13% of incompatible points
Model 1 Accuracy: 89%	Model 1 Accuracy: 77%	Model 1 Accuracy: 79%
Model 2 Accuracy: 10%	Model 2 Accuracy: 17%	Model 2 Accuracy: 21%

Downstream failures in receipt fraud detection

Fraud Attack	Error Score (h_1)	Error Score (h_2)
"Nlke"	55%	86.4%
"G00gle"	85.3%	99.01%
"ZUP"	49%	84.6%

1.4 Components

The Backward Compatibility ML library has two components:

- **A series of loss functions** in which users can vary the weight assigned to the dissonance factor and explore performance/capability tradeoffs during machine learning optimization.
- **Visualization widgets** that help users examine metrics and error data in detail. They provide a view of error intersections between models and incompatibility distribution across classes.

1.5 References

*Updates in Human-AI Teams: Understanding and Addressing the Performance/Compatibility Trade-off.*Gagan Bansal, Besmira Nushi, Ece Kamar, Daniel S Weld, Walter S Lasecki, Eric Horvitz; AAAI 2019.pdf

An Empirical Analysis of Backward Compatibility in Machine Learning Systems. Megha Srivastava, Besmira Nushi, Ece Kamar, Shital Shah, Eric Horvitz; KDD 2020. pdf

*Towards Accountable AI: Hybrid Human-Machine Analyses for Characterizing System Failure.*Besmira Nushi, Ece Kamar, Eric Horvitz; HCOMP 2018.pdf

CHAPTER 2

Help Topics

2.1 Getting Started

2.1.1 Backward Compatibility ML library requirements

The requirements for installing and running the Backward Compatibility ML library are:

- Windows 10 / Linux OS (tested on Ubuntu 18.04 LTS)
- Python 3.6

2.1.2 Installing the Backward Compatibility ML library

Follow these steps to install the Backward Compatibility ML library on your computer. You may want to install Anaconda (or other virtual environment) on your system for convenience, then follow these steps:

1. **(optional) Prepare a conda virtual environment:**

```
conda create -n bcml python=3.6
conda activate bcml
```

2. **(optional) Ensure you have the latest pip**

```
python -m pip install --upgrade pip
```

3. **Install the Backward Compatibility ML library:**

   On Linux:

```
pip install backwardcompatibilityml
```

   On Windows:

```
pip install backwardcompatibilityml -f https://download.pytorch.
↪org/whl/torch_stable.html
```

4. Import the 'backwardcompatibilityml' package in your code. For example:

```
import backwardcompatibilityml.loss as bcloss
import backwardcompatibilityml.scores as scores
```

2.1.3 Running the Backward Compatibility ML library examples

Note: The Backward Compatibility ML library examples were developed as Jupyter Notebooks and require the Jupyter Software to be installed. The steps below assume that you have git installed on your system.

The Backward Compatibility ML library includes several examples so you can quickly get an idea of its benefits and learn how to integrate it into your existing ML training workflow.

To download and run the examples, follow these steps:

1. Clone the BackwardCompatibilityML repository:

```
git clone https://github.com/microsoft/BackwardCompatibilityML.git
```

2. Install the requirements for the examples:

```
cd BackwardCompatibilityML
```

On Linux:

```
pip install -r example-requirements.txt
```

On Windows:

```
pip install -r example-requirements.txt -f https://download.pytorch.org/
↪whl/torch_stable.html
```

3. Start your Jupyter Notebooks server and load an example notebook under the 'examples' folder:

```
cd examples
jupyter notebook
```

Backward Compatibility ML library examples included

Notebook name	Framework	Dataset	Network	Optimizer	Backward Compatibility Dissonance Function	Backward Compatibility Loss Function	Uses Compatibility-Analysis widget
bcnllloss	PyTorch	MNIST	Custom	SGD	New Error	NLLLoss	N
compatibility-analysis	PyTorch	MNIST	Custom	SGD	New Error & Strict Imitation	CrossEntropyLoss	Y
compatibility-analysis-adult	PyTorch	UCI Adult Data Set	LogisticRegression & MLPClassifier	SGD	New Error & Strict Imitation	CrossEntropyLoss	Y
compatibility-analysis-cifar10-resnet18	PyTorch	CIFAR10	Custom & RESNet 18	SGD	New Error & Strict Imitation	CrossEntropyLoss	Y
compatibility-analysis-cifar10-resnet18-pretrained	PyTorch	CIFAR10	Custom & RESNet 18 (pretrained)	SGD	New Error & Strict Imitation	CrossEntropyLoss	Y
compatibility-analysis-from-saved-data	PyTorch	MNIST	Custom	SGD	New Error & Strict Imitation	CrossEntropyLoss	Y
si_cross_entropy_loss	PyTorch	MNIST	Custom	SGD	Strict Imitation	CrossEntropyLoss	N
si_nllloss	PyTorch	MNIST	Custom	SGD	Strict Imitation	NLLLoss	N

2.1.4 Next steps

Do you want to learn how to integrate the Backward Compatibility ML Loss Function in your new or existing ML training workflows? *Follow this tutorial.*

If you want to ask us a question, suggest a feature or report a bug, please contact the team by filing an issue in our repository on GitHub. We look forward to hearing from you!

2.2 Integrating the Backward Compatibility ML Loss Functions

We have implemented the following compatibility loss functions:

1. `BCCrossEntropyLoss`
2. `BCNLLLoss`

And the following strict imitation loss functions:

1. `StrictImitationCrossEntropyLoss`
2. `StrictImitationNLLLoss`

Both these sets of loss functions are implemented along the lines of

```
compatibility_loss(x, y) = underlying_loss(h2(x), y) + lambda_c *
dissonance(h1, h2, x, y)
```

Where the dissonance is the backward compatibility dissonance for the compatibility loss functions, and the strict imitation dissonance in the case of the strict imitation loss functions.

2.2.1 Example Usage

Let us assume that we have a pre-trained model `h1` that we want to use as our reference model while training / updating a new model `h2`.

Let us load our pre-trained model:

```
h1 = MyModel()
h1.load_state_dict(torch.load("path/to/state/dict.state"))
```

Then let us instantiate `h2` and train / update it, using `h1` as a reference:

```
from backwardcompatibilityml.loss import BCCrossEntropyLoss

h2 = MyModel()
lambda_c = 0.7
bc_loss = BCCrossEntropyLoss(h1, h2, lambda_c)

for data, target in updated_training_set:
    h2.zero_grad()
    loss = bc_loss(data, target)
    loss.backward()
```

Calling `loss.backward()` at each step of the training iteration, updates the weights of the model `h2`.

You may also decide to use an optimizer as follows:

```
import torch.optim as optim
from backwardcompatibilityml.loss import BCCrossEntropyLoss

h2 = MyModel()
lambda_c = 0.7
learning_rate = 0.01
momentum = 0.5
bc_loss = BCCrossEntropyLoss(h1, h2, lambda_c)
optimizer = optim.SGD(h2.parameters(), lr=learning_rate, momentum=momentum)

for data, target in updated_training_set:
    loss = bc_loss(data, target)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

The usage for `BCNLLLoss`, `StrictImitationCrossEntropyLoss` and `StrictImitationNLLLoss` is exactly the same as above.

2.2.2 Assumptions on the implementation of h1 and h2

It is important*to emphasize that since the compatibility and strict imitation loss functions need to use `h1` and `h2` to calculate the loss, some assumptions had to be made on the output returned by `h1` and `h2`.

Specifically, we require that both the models `h1` and `h2` return an ordered triple containing:

1. The raw logits output from the final layer.
2. The function softmax applied to the raw logits.
3. The function log_softmax applied to the raw logits.

Here is an example Logistic Regression model satisfying these requirements:

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


class LogisticRegression(nn.Module):

    def __init__(self, input_dim, output_dim):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        out = self.linear(x)
        out_softmax = F.softmax(out, dim=-1)
        out_log_softmax = F.log_softmax(out, dim=-1)

        return out, out_softmax, out_log_softmax
```

Here is an example Convolutional Network model satisfying these requirements:

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

class ConvolutionalNetwork(nn.Module):
    def __init__(self):
        super(ConvolutionalNetwork, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x, F.softmax(x, dim=1), F.log_softmax(x, dim=1)
```

2.3 Using the Backward Compatibility ML Compatibility Analysis Widget

Note: This topic will be added in a future release. Will cover how to use the Compatibility Analysis widget and visualizations to interpret outputs from loss functions

CHAPTER 3

backwardcompatibilityml

3.1 backwardcompatibilityml package

3.1.1 Subpackages

backwardcompatibilityml.helpers package

Submodules

backwardcompatibilityml.helpers.training module

backwardcompatibilityml.helpers.training.**compatibility_scores**(*h1*, *h2*, *dataset*, *device='cpu'*)

Parameters

- **h1** – Reference Pytorch model.
- **h2** – The model being compared to h1.
- **dataset** – Data in the form of a list of batches of input/target pairs.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns A pair consisting of **btc_dataset** - the average trust compatibility score over all batches, and **bec_dataset** - the average error compatibility score over all batches.

backwardcompatibilityml.helpers.training.**compatibility_sweep**(*sweeps_folder_path*, *number_of_epochs*, *h1*, *h2*, *training_set*, *test_set*, *batch_size_train*, *batch_size_test*, *OptimizerClass*, *optimizer_kwargs*, *NewErrorLossClass*, *StrictImitationLossClass*, *performance_metric=None*, *lambda_c_stepsize=0.25*, *percent_complete_queue=None*, *new_error_loss_kwargs=None*, *strict_imitation_loss_kwargs=None*, *device='cpu'*)

This function trains a new model using the backward compatibility loss function BCNLLLoss with respect to an existing model. It does this for each value of lambda_c betweek 0 and 1 at the specified step sizes. It saves the newly trained models in the specified folder.

Parameters

- **sweeps_folder_path** – A string value representing the full path of the folder wehre the result of the compatibility sweep is to be stored.
- **number_of_epochs** – The number of training epochs to use on each sweep.
- **h1** – The reference model being used.
- **h2** – The new model being traind / updated.
- **training_set** – The list of training samples as (input, target) pairs.
- **test_set** – The list of testing samples as (input, target) pairs.
- **batch_size_train** – An integer representing batch size of the training set.
- **batch_size_test** – An integer representing the batch size of the test set.
- **OptimizerClass** – The class to instantiate an optimizer from for training.
- **optimizer_kwargs** – A dictionary of the keyword arguments to be used to instantiate the optimizer.
- **NewErrorLossClass** – The class of the New Error style loss function to be instantiated and used to perform compatibility constrained training of our model h2.
- **StrictImitationLossClass** – The class of the Strict Imitation style loss function to be instantiated and used to perform compatibility constrained training of our model h2.
- **performance_metric** – Optional performance metric to be used when evaluating the model. If not specified then accuracy is used.
- **lambda_c_stepsize** – The increments of lambda_c to use as we sweep the parameter space between 0.0 and 1.0.
- **percent_complete_queue** – Optional thread safe queue to use for logging the status of the sweep in terms of the percentage complete.

- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

backwardcompatibilityml.helpers.training.**evaluate_model_performance_and_compatibility**(*h1*, *h2*, *training_se test_se performance device='*

Calculate the error overlap of h1 and h2 on a batched dataset. Calculate the h2 model error fraction by class on a batched dataset.

Parameters

- **h1** – The reference model being used.
- **h2** – The model being traind / updated.
- **performance_metric** – Optional performance metric to be used when evaluating the model. If not specified then accuracy is used.
- **training_set** – The list of batched training samples as (input, target) pairs.
- **test_set** – The list of batched testing samples as (input, target) pairs.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns A dictionary containing the results of the model performance and evaluation performed on the training and the testing sets separately.

backwardcompatibilityml.helpers.training.**evaluate_model_performance_and_compatibility_on_da**

Parameters

- **h1** – The reference model being used.
- **h2** – The model being traind / updated.
- **performance_metric** – Optional performance metric to be used when evaluating the model. If not specified then accuracy is used.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns A dictionary containing the models error overlap between h1 and h2, the error fraction by class of the model h2, the trust compatibility score of h2 with respect to h1, and the error compatibility score of h2 with respect to h1.

backwardcompatibilityml.helpers.training.**get_error_fraction_by_class**(*model*, *batched_evaluation_data*, *batched_evaluation_target*, *device='cpu'*)

Return the fraction of errors of the model by class.

Parameters

- **model** – The model being evaluated.
- **batched_evaluation_data** – A single batch of input data to be passed to our model.
- **batched_evaluation_target** – A single batch of the corresponding output targets.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns A dictionary of key / value pairs, where the key is the output class and the value is the fraction of misclassification errors of the model within that class.

backwardcompatibilityml.helpers.training.**get_error_instance_indices**(*model*, *batched_evaluation_data*, *batched_evaluation_target*, *device='cpu'*)

Return the list of indices of instances from batched_evaluation_data on which the model prediction differs from the ground truth in batched_evaluation_target.

Parameters

- **model** – The model being evaluated.
- **batched_evaluation_data** – A single batch of input data to be passed to our model.
- **batched_evaluation_target** – A single batch of the corresponding output targets.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns A list of indices of the instances within the batched data, for which the model did not match the expected target.

backwardcompatibilityml.helpers.training.**get_normalized_model_error_overlap**(*h1*, *h2*, *batched_evaluation_d*, *batched_evaluation_t*, *device='cpu'*)

Return the fraction of errors of each model and the fraction of errors common to both models, all with respect to the total number of errors of each model.

Parameters

- **h1** – Reference Pytorch model.

- **h2** – The model being compared to h1.
- **batched_evaluation_data** – A single batch of input data to be passed to our model.
- **batched_evaluation_target** – A single batch of the corresponding output targets.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns

If there are any errors at all it returns a triple of the form proportion_of_error_due_to_h1, proportion_of_error_due_to_h2, proportion_of_error_due_to_h1_and_h2

If there are no errors at all it returns the triple 0, 0, 0

backwardcompatibilityml.helpers.training.**model_accuracy**(*model*, *dataset*, *device='cpu'*)

backwardcompatibilityml.helpers.training.**test**(*network*, *loss_function*, *test_set*, *batch_size_test*, *device='cpu'*)

Tests a model in a test set using the loss function provided.

(Please note that this is not to be used for testing with a compatibility loss function.)

Parameters

- **network** – The model which is undergoing testing.
- **loss_function** – An instance of the loss function to use for training.
- **test_set** – The list of testing samples as (input, target) pairs.
- **batch_size_test** – An integer representing the batch size of the test set.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns Returns a list of test loses.

backwardcompatibilityml.helpers.training.**test_compatibility**(*h2*, *loss_function*, *test_set*, *batch_size_test*, *device='cpu'*)

Tests a model in a test set using the backward compatibility loss function provided.

Parameters

- **h2** – The model which is undergoing training / updating.
- **loss_function** – An instance of a compatibility loss function.
- **test_set** – The list of testing samples as (input, target) pairs.
- **batch_size_test** – An integer representing the batch size of the test set.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on

the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns Returns a list of test loses.

backwardcompatibilityml.helpers.training.**train**(*number_of_epochs*, *network*, *optimizer*, *loss_function*, *training_set*, *test_set*, *batch_size_train*, *batch_size_test*, *device='cpu'*)

Trains a model with respect to a loss function, using an instance of an optimizer.

(Please note that this is not to be used for training with a compatibility loss function.)

Parameters

- **network** – The model which is undergoing training.
- **number_of_epochs** – Number of epochs of training.
- **optimizer** – The optimizer instance to use for training.
- **loss_function** – An instance of the loss function to use for training.
- **training_set** – The list of training samples as (input, target) pairs.
- **test_set** – The list of testing samples as (input, target) pairs.
- **batch_size_train** – An integer representing batch size of the training set.
- **batch_size_test** – An integer representing the batch size of the test set.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns

Returns four lists
train_counter - The index of a training samples at which training losses were logged.
test_counter - The index of testing samples at which testing losses were logged.
train_losses - The list of logged training losses.
test_losses - The list of logged testing losses.

backwardcompatibilityml.helpers.training.**train_compatibility**(*number_of_epochs*, *h2*, *optimizer*, *loss_function*, *training_set*, *test_set*, *batch_size_train*, *batch_size_test*, *device='cpu'*)

Trains a new model with respect to an existing model using the compatibility loss function provided. The compatibility loss function may be either a New Error or Strict Imitation type loss function.

Parameters

- **h2** – The model which is undergoing training / updating.
- **number_of_epochs** – Number of epochs of training.

- **loss_function** – An instance of a compatibility loss function.
- **optimizer** – The optimizer instance to use for training.
- **training_set** – The list of training samples as (input, target) pairs.
- **test_set** – The list of testing samples as (input, target) pairs.
- **batch_size_train** – An integer representing batch size of the training set.
- **batch_size_test** – An integer representing the batch size of the test set.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns

Returns four lists
train_counter - The index of a training samples at which training losses were logged.
test_counter - The index of testing samples at which testing losses were logged.
train_losses - The list of logged training losses.
test_losses - The list of logged testing losses.

backwardcompatibilityml.helpers.training.**train_compatibility_epoch**(*epoch*, *h2*, *optimizer*, *loss_function*, *training_set*, *batch_size_train*, *device='cpu'*)

Trains a new model using the instance compatibility loss function provided, over a single epoch. The compatibility loss function instnace may be either a New Error or Strict Imitation type loss function.

Parameters

- **epoch** – The integer index of the training epoch being run.
- **h2** – The model which is undergoing training / updating.
- **optimizer** – The optimizer instance to use for training.
- **loss_function** – An instance of a compatibility loss function.
- **training_set** – The list of training samples as (input, target) pairs.
- **batch_size_train** – An integer representing batch size of the training set.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns A list of pairs of the form (training_instance_index, training_loss) at regular intervals of 10 training samples.

backwardcompatibilityml.helpers.training.**train_epoch**(*epoch*, *network*, *optimizer*, *loss_function*, *training_set*, *batch_size_train*, *device='cpu'*)

Trains a model over a single training epoch, with respect to a loss function, using an instance of an optimizer.

(Please note that this is not to be used for training with a compatibility loss function.)

Parameters

- **network** – The model which is undergoing training.
- **optimizer** – The optimizer instance to use for training.
- **loss_function** – An instance of the loss function to use for training.
- **training_set** – The list of training samples as (input, target) pairs.
- **batch_size_train** – An integer representing batch size of the training set.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

Returns A list of pairs of the form (training_instance_index, training_loss) at regular intervals of 10 training samples.

backwardcompatibilityml.helpers.training.**train_new_error**(*h1*, *h2*, *number_of_epochs*, *training_set*, *test_set*, *batch_size_train*, *batch_size_test*, *OptimizerClass*, *optimizer_kwargs*, *NewErrorLossClass*, *lambda_c*, *new_error_loss_kwargs=None*, *device='cpu'*)

Parameters

- **h1** – Reference Pytorch model.
- **h2** – The model which is undergoing training / updating.
- **number_of_epochs** – Number of epochs of training.
- **training_set** – The list of training samples as (input, target) pairs.
- **test_set** – The list of testing samples as (input, target) pairs.
- **batch_size_train** – An integer representing batch size of the training set.
- **batch_size_test** – An integer representing the batch size of the test set.
- **OptimizerClass** – The class to instantiate an optimizer from for training.
- **optimizer_kwargs** – A dictionary of the keyword arguments to be used to instantiate the optimizer.
- **NewErrorLossClass** – The class of the New Error style loss function to be instantiated and used to perform compatibility constrained training of our model h2.
- **lambda_c** – The regularization parameter to be used when calibrating the degree of compatibility to enforce while training.

- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

backwardcompatibilityml.helpers.training.**train_strict_imitation**(*h1*, *h2*, *number_of_epochs*, *training_set*, *test_set*, *batch_size_train*, *batch_size_test*, *OptimizerClass*, *optimizer_kwargs*, *StrictImitationLossClass*, *lambda_c*, *strict_imitation_loss_kwargs=None*, *device='cpu'*)

Parameters

- **h1** – Reference Pytorch model.
- **h2** – The model which is undergoing training / updating.
- **number_of_epochs** – Number of epochs of training.
- **training_set** – The list of training samples as (input, target) pairs.
- **test_set** – The list of testing samples as (input, target) pairs.
- **batch_size_train** – An integer representing batch size of the training set.
- **batch_size_test** – An integer representing the batch size of the test set.
- **OptimizerClass** – The class to instantiate an optimizer from for training.
- **optimizer_kwargs** – A dictionary of the keyword arguments to be used to instantiate the optimizer.
- **StrictImitationLossClass** – The class of the Strict Imitation style loss function to be instantiated and used to perform compatibility constrained training of our model h2.
- **lambda_c** – The regularization parameter to be used when calibrating the degree of compatibility to enforce while training.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

backwardcompatibilityml.helpers.utils module

backwardcompatibilityml.helpers.utils.**add_memory_hooks**(*idx*, *mod*, *mem_log*, *exp*, *hr*)

backwardcompatibilityml.helpers.utils.**clean_from_gpu**(*tensors*)

Utility function to clean tensors from the GPU. This is only intended to be used when investigating why memory usage is high. An in production solution should instead rely on correctly structuring your code so that Python garbage collection automatically removes the unreferenced tensors as they move out of function scope. :param tensors: A list of tensor objects to clean from the GPU.

Returns None

backwardcompatibilityml.helpers.utils.**generate_mem_hook**(*handle_ref*, *mem*, *idx*, *hook_type*, *exp*)

backwardcompatibilityml.helpers.utils.**get_class_probabilities**(*batch_label_tensor*)

backwardcompatibilityml.helpers.utils.**get_gpu_mem**()

backwardcompatibilityml.helpers.utils.**labels_to_probabilities**(*batch_class_labels*, *num_classes=None*, *batch_size=None*)

backwardcompatibilityml.helpers.utils.**log_mem**(*model*, *mem_log=None*, *exp=None*)

Utility funtion for adding memory usage logging to a Pytorch model.

Example usage:

model = MyModel()

hook_handles, mem_log = log_mem(model, exp="memory-profiling-experiment")

... then do a training run ...

mem_log should now contain the results of the memory profiling experiment.

Parameters

- **model** – A pytorch model
- **mem_log** – Optional list object, which may contain data from previous profiling experiments.
- **exp** – String identifier for the profiling experiment name.

Returns A pair consisting of **mem_log** - either the same mem_log list object that was passed in, or a newly constructed one, that will contain the results of the logging, and **hook_handles** - a list of handles for our logging hooks that will need to be cleared when we are done logging.

backwardcompatibilityml.helpers.utils.**remove_memory_hooks**(*hook_handles*)

Clear the memory profiling hooks put in place by log_mem :param hook_handles: A list of hook hndles to clear

Returns None

backwardcompatibilityml.helpers.utils.**show_allocated_tensors**()

Attempts to print out the tensors in memory. :param None:

Returns None

backwardcompatibilityml.helpers.utils.**sigmoid_to_labels**(*batch_sigmoids*, *discriminant_pivot=0.5*)

Module contents

backwardcompatibilityml.loss package

Submodules

backwardcompatibilityml.loss.new_error module

class backwardcompatibilityml.loss.new_error.**BCBinaryCrossEntropyLoss**(*h1*, *h2*, *lambda_c*, *discriminan_pivot=0.5*, ***kwargs*)

Bases: torch.nn.modules.module.Module

Backward Compatibility Cross Entropy Loss

This class implements the backward compatibility loss function with the underlying loss function being the Cross Entropy loss.

Example usage: h1 = MyModel() ... train h1 ... h1.eval() (it is important that h1 be put in evaluation mode)

lambda_c = 0.5 (regularization parameter) h2 = MyNewModel() (this may be the same model type as MyModel) bcloss = BCBinaryCrossEntropyLoss(h1, h2, lambda_c)

for x, y in training_data: loss = bcloss(x, y) loss.backward()

Note that we pass in the input and the target directly to the bcloss function instance. It calculates the outputs of h1 and h2 internally.

Parameters

- **h1** – Our reference model which we would like to be compatible with.
- **h2** – Our new model which will be the updated model.
- **lambda_c** – A float between 0.0 and 1.0, which is a regularization parameter that determines how much we want to penalize model h2 for being incompatible with h1. Lower values panalize less and higher values penalize more.

dissonance(*h2_support_output_sigmoid*, *target_labels*)

forward(*x*, *y*, *reduction='mean'*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class backwardcompatibilityml.loss.new_error.**BCCrossEntropyLoss**(*h1*, *h2*, *lambda_c*, ***kwargs*)

Bases: torch.nn.modules.module.Module

Backward Compatibility Cross Entropy Loss

This class implements the backward compatibility loss function with the underlying loss function being the Cross Entropy loss.

Example usage: h1 = MyModel() ... train h1 ... h1.eval() (it is important that h1 be put in evaluation mode)

lambda_c = 0.5 (regularization parameter) h2 = MyNewModel() (this may be the same model type as MyModel) bcloss = BCCrossEntropyLoss(h1, h2, lambda_c)

for x, y in training_data: loss = bcloss(x, y) loss.backward()

Note that we pass in the input and the target directly to the bcloss function instance. It calculates the outputs of h1 and h2 internally.

Parameters

- **h1** – Our reference model which we would like to be compatible with.
- **h2** – Our new model which will be the updated model.
- **lambda_c** – A float between 0.0 and 1.0, which is a regularization parameter that determines how much we want to penalize model h2 for being incompatible with h1. Lower values panalize less and higher values penalize more.

dissonance(*h2_output_logit*, *target_labels*)

forward(*x*, *y*, *reduction='mean'*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class backwardcompatibilityml.loss.new_error.**BCKLDivergenceLoss**(*h1*, *h2*, *lambda_c*, *num_classes=None*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Backward Compatibility Kullback–Leibler Divergence Loss

This class implements the backward compatibility loss function with the underlying loss function being the Kullback–Leibler Divergence loss.

Example usage: h1 = MyModel() ... train h1 ... h1.eval() (it is important that h1 be put in evaluation mode)

lambda_c = 0.5 (regularization parameter) h2 = MyNewModel() (this may be the same model type as MyModel) bcloss = BCKLDivergenceLoss(

h1, h2, lambda_c, num_classes=num_classes)

for x, y in training_data: loss = bcloss(x, y) loss.backward()

Note that we pass in the input and the target directly to the bcloss function instance. It calculates the outputs of h1 and h2 internally.

Parameters

- **h1** – Our reference model which we would like to be compatible with.
- **h2** – Our new model which will be the updated model.
- **lambda_c** – A float between 0.0 and 1.0, which is a regularization parameter that determines how much we want to penalize model h2 for being incompatible with h1. Lower values panalize less and higher values penalize more.

- **num_classes** – An integer denoting the number of classes that we are attempting to classify the input into.

dissonance(*h2_output_log_softmax*, *target_labels*)

forward(*x*, *y*, *reduction='batchmean'*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class backwardcompatibilityml.loss.new_error.**BCNLLLoss**(*h1*, *h2*, *lambda_c*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Backward Compatibility Negative Log Likelihood Loss

This class implements the backward compatibility loss function with the underlying loss function being the Negative Log Likelihood loss.

Example usage: h1 = MyModel() . . . train h1 . . . h1.eval() (it is important that h1 be put in evaluation mode)

lambda_c = 0.5 (regularization parameter) h2 = MyNewModel() (this may be the same model type as MyModel) bcloss = BCNLLLoss(h1, h2, lambda_c)

for x, y in training_data: loss = bcloss(x, y) loss.backward()

Note that we pass in the input and the target directly to the bcloss function instance. It calculates the outputs of h1 and h2 internally.

Parameters

- **h1** – Our reference model which we would like to be compatible with.
- **h2** – Our new model which will be the updated model.
- **lambda_c** – A float between 0.0 and 1.0, which is a regularization parameter that determines how much we want to penalize model h2 for being incompatible with h1. Lower values panalize less and higher values penalize more.

forward(*x*, *y*, *reduction='mean'*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

backwardcompatibilityml.loss.strict_imitation module

class backwardcompatibilityml.loss.strict_imitation.**StrictImitationBinaryCrossEntropyLoss**(*h*
h
d
d
i
r

Bases: torch.nn.modules.module.Module

Strict Imitation Cross Entropy Loss

This class implements the strict imitation loss function with the underlying loss function being the Cross Entropy loss.

Example usage: h1 = MyModel() ... train h1 ... h1.eval() (it is important that h1 be put in evaluation mode)

lambda_c = 0.5 (regularization parameter) h2 = MyNewModel() (this may be the same model type as MyModel) siloss = StrictImitationBinaryCrossEntropyLoss(h1, h2, lambda_c)

for x, y in training_data: loss = siloss(x, y) loss.backward()

Note that we pass in the input and the target directly to the siloss function instance. It calculates the outputs of h1 and h2 internally.

Parameters

- **h1** – Our reference model which we would like to be compatible with.
- **h2** – Our new model which will be the updated model.
- **lambda_c** – A float between 0.0 and 1.0, which is a regularization parameter that determines how much we want to penalize model h2 for being incompatible with h1. Lower values panalize less and higher values penalize more.

dissonance(*h1_output_sigmoid*, *h2_output_sigmoid*)

forward(*x*, *y*, *reduction='mean'*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class backwardcompatibilityml.loss.strict_imitation.**StrictImitationCrossEntropyLoss**(*h1*,
h2,
lambda_c
***kwargs*

Bases: torch.nn.modules.module.Module

Strict Imitation Cross Entropy Loss

This class implements the strict imitation loss function with the underlying loss function being the Cross Entropy loss.

Example usage: h1 = MyModel() ... train h1 ... h1.eval() (it is important that h1 be put in evaluation mode)

lambda_c = 0.5 (regularization parameter) h2 = MyNewModel() (this may be the same model type as MyModel) siloss = StrictImitationCrossEntropyLoss(h1, h2, lambda_c)

for x, y in training_data: loss = siloss(x, y) loss.backward()

Note that we pass in the input and the target directly to the siloss function instance. It calculates the outputs of h1 and h2 internally.

Parameters

- **h1** – Our reference model which we would like to be compatible with.
- **h2** – Our new model which will be the updated model.
- **lambda_c** – A float between 0.0 and 1.0, which is a regularization parameter that determines how much we want to penalize model h2 for being incompatible with h1. Lower values panalize less and higher values penalize more.

dissonance(*h1_output_labels*, *h2_output_logit*)

forward(*x*, *y*, *reduction='mean'*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class backwardcompatibilityml.loss.strict_imitation.**StrictImitationKLDivergenceLoss**(*h1*, *h2*, *lambda_c*, *num_clas*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Strict Imitation Kullback–Leibler Divergence Loss

This class implements the strict imitation loss function with the underlying loss function being the Kullback–Leibler Divergence loss.

Example usage: h1 = MyModel() ... train h1 ... h1.eval() (it is important that h1 be put in evaluation mode)

lambda_c = 0.5 (regularization parameter) h2 = MyNewModel() (this may be the same model type as MyModel) siloss = StrictImitationKLDivergenceLoss(h1, h2, lambda_c, num_classes=num_classes)

for x, y in training_data: loss = siloss(x, y) loss.backward()

Note that we pass in the input and the target directly to the siloss function instance. It calculates the outputs of h1 and h2 internally.

Parameters

- **h1** – Our reference model which we would like to be compatible with.
- **h2** – Our new model which will be the updated model.
- **lambda_c** – A float between 0.0 and 1.0, which is a regularization parameter that determines how much we want to penalize model h2 for being incompatible with h1. Lower values panalize less and higher values penalize more.

- **num_classes** – An integer denoting the number of classes that we are attempting to classify the input into.

dissonance(*h1_output_logsoftmax*, *h2_output_logsoftmax*)

forward(*x*, *y*, *reduction='batchmean'*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class backwardcompatibilityml.loss.strict_imitation.**StrictImitationNLLLoss**(*h1*, *h2*, *lambda_c*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Strict Imitation Negative Log Likelihood Loss

This class implements the strict imitation loss function with the underlying loss function being the Negative Log Likelihood loss.

Example usage: h1 = MyModel() ... train h1 ... h1.eval() (it is important that h1 be put in evaluation mode)

lambda_c = 0.5 (regularization parameter) h2 = MyNewModel() (this may be the same model type as MyModel) siloss = StrictImitationNLLLoss(h1, h2, lambda_c)

for x, y in training_data: loss = siloss(x, y) loss.backward()

Note that we pass in the input and the target directly to the siloss function instance. It calculates the outputs of h1 and h2 internally.

Parameters

- **h1** – Our reference model which we would like to be compatible with.
- **h2** – Our new model which will be the updated model.
- **lambda_c** – A float between 0.0 and 1.0, which is a regularization parameter that determines how much we want to penalize model h2 for being incompatible with h1. Lower values panalize less and higher values penalize more.

dissonance(*h1_output_prob*, *h2_output_prob*)

forward(*x*, *y*, *reduction='mean'*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Module contents

backwardcompatibilityml.widget package

Subpackages

backwardcompatibilityml.widget.resources package

Module contents

Submodules

backwardcompatibilityml.widget.compatibility_analysis module

class backwardcompatibilityml.widget.compatibility_analysis.**CompatibilityAnalysis**(*folder_name*
number_of_epoch
h1,
h2,
training_set,
test_set,
batch_size_t
batch_size_t
lambda_c_st
Optimizer
Class=None
optimizer_kwarg
NewErrorLossClass=None
StrictImitationLossClass=None
port=None,
new_error_l
strict_imitat
device='cpu')

Bases: object

The CompatibilityAnalysis class is an interactive widget intended for use within a Jupyter Notebook. It provides an interactive UI for the user to interact with for:

1. Initiating a sweep of the lambda_c parameter space while performing compatibility training / updating of a model h2 with respect to a reference model h1.
2. Checking on the status of the sweep being performed.
3. Interacting with the data generated during the sweep, once the sweep is completed.

Note that this class may only be instantiated once within the same Notebook at this time.

This class works by instantiating a Flask server listening on a free port in the 5000 - 5099 range, or a port explicitly specified by the user.

It then registers a few REST api endpoints on this Flask server. The UI for the widget which is displayed within the Jupyter Notebook, interacts with these REST api endpoints over HTTP requests. It dynamically loads data and uses it to render visualizations within the widget UI.

Parameters

- **folder_name** – A string value representing the full path of the folder wehre the result of the compatibility sweep is to be stored.
- **number_of_epochs** – The number of training epochs to use on each sweep.
- **h1** – The reference model being used.
- **h2** – The new model being traind / updated.
- **training_set** – The list of training samples as (input, target) pairs.
- **test_set** – The list of testing samples as (input, target) pairs.
- **batch_size_train** – An integer representing batch size of the training set.
- **batch_size_test** – An integer representing the batch size of the test set.
- **lambda_c_stepsize** – The increments of lambda_c to use as we sweep the parameter space between 0.0 and 1.0.
- **OptimizerClass** – The class to instantiate an optimizer from for training.
- **optimizer_kwargs** – A dictionary of the keyword arguments to be used to instantiate the optimizer.
- **NewErrorLossClass** – The class of the New Error style loss function to be instantiated and used to perform compatibility constrained training of our model h2.
- **StrictImitationLossClass** – The class of the Strict Imitation style loss function to be instantiated and used to perform compatibility constrained training of our model h2.
- **port** – An integer value to indicate the port to which the Flask service should bind.
- **device** – A string with values either "cpu" or "cuda" to indicate the device that Pytorch is performing training on. By default this value is "cpu". But in case your models reside on the GPU, make sure to set this to "cuda". This makes sure that the input and target tensors are transferred to the GPU during training.

backwardcompatibilityml.widget.compatibility_analysis.**build_environment_params**(*flask_service_env*)

A small helper function to return a dictionary of the environment type and the base url of the Flask service for the environment type.

Parameters **flask_service_env** – An instance of an environment from rai_core_flask.environments.

Returns A dictionary of the environment type specified as a string, and the base url to be used when accessing the Flask service for this environment type.

Module contents

3.1.2 Submodules

3.1.3 backwardcompatibilityml.scores module

backwardcompatibilityml.scores.**error_compatibility_score**(*h1_output_labels*, *h2_output_labels*, *expected_labels*)

The fraction of instances labeled incorrectly by h1 and h2 out of the total number of instances labeled incorrectly by h1.

Parameters

- **h1_output_labels** – A list of the labels outputted by the model h1.
- **h2_output_labels** – A list of the labels output by the model h2.
- **expected_labels** – A list of the corresponding ground truth target labels.

Returns If h1 has any errors, then we return the error compatibility score of h2 with respect to h1. If h1 has no errors then we return 0.

backwardcompatibilityml.scores.**trust_compatibility_score**(*h1_output_labels*, *h2_output_labels*, *expected_labels*)

The fraction of instances labeled correctly by both h1 and h2 out of the total number of instances labeled correctly by h1.

Parameters

- **h1_output_labels** – A list of the labels outputted by the model h1.
- **h2_output_labels** – A list of the labels output by the model h2.
- **expected_labels** – A list of the corresponding ground truth target labels.

Returns If h1 has any errors, then we return the trust compatibility score of h2 with respect to h1. If h1 has no errors then we return 0.

3.1.4 backwardcompatibilityml.sweep_management module

class backwardcompatibilityml.sweep_management.**SweepManager**(*folder_name*, *number_of_epochs*, *h1*, *h2*, *training_set*, *test_set*, *batch_size_train*, *batch_size_test*, *OptimizerClass*, *optimizer_kwargs*, *NewErrorLossClass*, *StrictImitationLossClass*, *lambda_c_stepsize=0.25*, *new_error_loss_kwargs=None*, *strict_imitation_loss_kwargs=None*, *device='cpu'*)

Bases: object

The SweepManager class is used to manage an experiment that performs training / updating a model h2, with respect to a reference model h1 in a way that preserves compatibility between the models. The experiment performs a sweep of the parameter space of the regularization parameter lambda_c, by performing compatibility trainings for small increments in the value of lambda_c for some settable step size.

The sweep manager can run the sweep experiment either synchronously, or within a separate thread. In the latter case, it provides some helper functions that allow you to check on the percentage of the sweep that is complete.

Parameters

- **folder_name** – A string value representing the full path of the folder wehre the result of the compatibility sweep is to be stored.
- **number_of_epochs** – The number of training epochs to use on each sweep.
- **h1** – The reference model being used.
- **h2** – The new model being traind / updated.
- **training_set** – The list of training samples as (input, target) pairs.
- **test_set** – The list of testing samples as (input, target) pairs.
- **batch_size_train** – An integer representing batch size of the training set.
- **batch_size_test** – An integer representing the batch size of the test set.
- **OptimizerClass** – The class to instantiate an optimizer from for training.
- **optimizer_kwargs** – A dictionary of the keyword arguments to be used to instantiate the optimizer.
- **NewErrorLossClass** – The class of the New Error style loss function to be instantiated and used to perform compatibility constrained training of our model h2.
- **StrictImitationLossClass** – The class of the Strict Imitation style loss function to be instantiated and used to perform compatibility constrained training of our model h2.
- **performance_metric** – Optional performance metric to be used when evaluating the model. If not specified then accuracy is used.
- **lambda_c_stepsize** – The increments of lambda_c to use as we sweep the parameter space between 0.0 and 1.0.

- **device** – A string with values either “cpu” or “cuda” to indicate the device that Pytorch is performing training on. By default this value is “cpu”. But in case your models reside on the GPU, make sure to set this to “cuda”. This makes sure that the input and target tensors are transferred to the GPU during training.

get_evaluation(*evaluation_id*)

get_sweep_status()

get_sweep_summary()

start_sweep()

start_sweep_synchronous()

3.1.5 Module contents

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index