Backprop

Release 0.1.1

Backprop

May 03, 2021

INTRODUCTION

Backprop makes it simple to use, finetune, and deploy state-of-the-art ML models.

CHAPTER

ONE

QUICKSTART

1.1 Installation

Install Backprop via PyPi

```
pip install backprop
```

1.2 Supported Tasks

Out of the box tasks you can solve with Backprop:

- Conversational question answering in English (for FAQ chatbots, text analysis, etc.)
- Text Classification in 100+ languages (for email sorting, intent detection, etc.)
- Image Classification (for object recognition, OCR, etc.)
- Text Vectorisation in 50+ languages (semantic search for ecommerce, documentation, etc.)
- Summarisation in English (TLDRs for long documents)
- Emotion detection in English (for customer satisfaction, text analysis, etc.)
- Text Generation (for idea, story generation and broad task solving)

For more specific use cases, you can adapt a task with little data and a couple of lines of code using finetuning. We are adding finetuning support for all tasks soon.

1.3 Basic Task Solving

```
import backprop

context = "Take a look at the examples folder to see use cases!"

qa = backprop.QA()

# Start building!
answer = qa("Where can I see what to build?", context)

print(answer)
# Prints
"the examples folder"
```

See examples for all available tasks in *Tasks*.

1.4 Basic Finetuning and Uploading

```
from backprop.models import T5
from backprop import TextGeneration

tg = TextGeneration(T5, local=True)

# Any text works as training data
inp = ["I really liked the service I received!", "Meh, it was not impressive."]
out = ["positive", "negative"]

# Finetune with a single line of code
tg.finetune({"input_text": inp, "output_text": out})

# Use your trained model
prediction = tg("I enjoyed it!")

print(prediction)
# Prints
"positive"

# Upload to Backprop for production ready inference
# Describe your model
name = "t5-sentiment"
description = "Predicts positive and negative sentiment"

tg.upload(name=name, description=description, api_key="abc")
```

Learn more about finetuning in *Finetuning*.

1.5 Why Backprop?

1. No experience needed
    - Entrance to practical AI should be simple
    - Get state-of-the-art performance in your task without being an expert
2. Data is a bottleneck
    - Use AI without needing access to "big data"
    - With transfer learning, no data is required, but even a small amount can adapt a task to your niche.
3. There is an overwhelming amount of models
    - We implement the best ones for various tasks
    - A few general models can accomplish more with less optimisation
4. Deploying models cost effectively is hard work
    - If our models suit your use case, no deployment is needed
    - Adapt and deploy your own model with a couple of lines of code

- Our API scales, is always available, and you only pay for usage

CHAPTER

TWO

TASKS

Tasks are classes that act as a "middleman" between you and a model. They know how to communicate with models running locally and in our API.

2.1 Supported Tasks

Tasks can be imported directly from `backprop`.

See the full reference in *backprop.tasks*.

To see which models are supported for each task, check out the Backprop Model Hub.

2.2 Q&A

Q&A answers a question based on a paragraph of text. It also supports previous questions and answers for a conversational setting.

2.2.1 Inference

```
from backprop import QA

qa = QA()

qa("Where does Sally live?", "Sally lives in London.")
"London"
```

2.2.2 Finetuning

The params dictionary for Q&A training consists of contexts, questions, answers, and optionally a list of previous Q/A pairs to be used in context.

The `prev_qa` parameter is used to make a Q&A system conversational – that is to say, it can infer the meaning of words in a new question, based on previous questions.

Finetuning Q&A also accepts keyword arguments `max_input_length` and `max_output_length`, which specify the maximum token length for inputs and outputs.

```
import backprop

# Initialise task
qa = backprop.QA()

"""
Set up training data for QA. Note that repeated contexts are needed,
along with empty prev_qas to match.
Input must be completely 1:1 each question has an associated
answer, context, and prev_qa (if prev_qa is to be used).
"""

questions = ["What's Backprop?",
             "What language is it in?",
             "When was the Moog synthesizer invented?"]

answers = ["A library that trains models", "Python", "1964"]

contexts = ["Backprop is a Python library that makes training and using models easier.
↪",
            "Backprop is a Python library that makes training and using models easier.
↪",
            "Bob Moog was a physicist. He invented the Moog synthesizer in 1964."]

prev_qas = [[],
            [("What's Backprop?", "A library that trains models")],
            []]

params = {"questions": questions,
        "answers": answers,
        "contexts": contexts,
        "prev_qas": prev_qas}

# Finetune
qa.finetune(params=params)
```

Consider the second example in the training data above. The question "What language is it in?" does not mean much on its own. But because the model can access the matching `prev_qa` example, it gets additional information: as the first question reads, "What's Backprop?", it can infer that the "it" in the next question refers to Backprop as well.

If you use `prev_qa`, you must ensure it is 1:1 with the rest of your input data: even if a row does not have any previous question/answer pairs, an empty list is still required at that index in `prev_qa`.

See the Q&A Notebook examples with code.

See the *Q&A Task Reference*.

2.3 Text Classification

Text Classification looks at input and assigns probabilities to a set of labels.

It is supported in 100+ languages: Afrikaans, Albanian, Amharic, Arabic, Armenian, Assamese, Azerbaijani, Basque, Belarusian, Bengali, Bengali Romanized, Bosnian, Breton, Bulgarian, Burmese, Burmese, Catalan, Chinese (Simplified), Chinese (Traditional), Croatian, Czech, Danish, Dutch, English, Esperanto, Estonian, Filipino, Finnish, French, Galician, Georgian, German, Greek, Gujarati, Hausa, Hebrew, Hindi, Hindi Romanized, Hungarian, Icelandic, Indonesian, Irish, Italian, Japanese, Javanese, Kannada, Kazakh, Khmer, Korean, Kurdish (Kurmanji), Kyrgyz, Lao, Latin, Latvian, Lithuanian, Macedonian, Malagasy, Malay, Malayalam, Marathi, Mongolian, Nepali, Norwegian, Oriya,

Oromo, Pashto, Persian, Polish, Portuguese, Punjabi, Romanian, Russian, Sanskri, Scottish, Gaelic, Serbian, Sindhi, Sinhala, Slovak, Slovenian, Somali, Spanish, Sundanese, Swahili, Swedish, Tamil, Tamil Romanized, Telugu, Telugu Romanized, Thai, Turkish, Ukrainian, Urdu, Urdu Romanized, Uyghur, Uzbek, Vietnamese, Welsh, Western, Frisian, Xhosa, Yiddish.

2.3.1 Inference

```
import backprop

tc = backprop.TextClassification()

tc("I am mad because my product broke.", ["product issue", "nature"])
{"product issue": 0.98, "nature": 0.05}
```

2.3.2 Finetuning

Supplying parameters for text classification is straightforward: the params dict contains the keys "texts" and "labels". The values of these keys are lists of input texts and the labels to which they are assigned. When you finetune, Backprop will automatically set up a model with the correct number of outputs (based on the unique labels passed in).

Finetuning text classification also accepts the keyword argument `max_length`, which specifoes the maximum token length for inputs.

```
import backprop

tc = backprop.TextClassification()

"""
Set up input data. Labels will automatically be used to set up
model with number of classes for classification.
"""

inp = ["This is a political news article",
       "This is a computer science research paper",
       "This is a movie review"]

out = ["Politics", "Science", "Entertainment"]

params = {"texts": inp, "labels": out}

# Finetune
tc.finetune(params)
```

Check the example Text Classification Notebook with code.

See the *Text Classification Task Reference*.

2.4 Sentiment/Emotion Detection

This is exactly what it says on the tin: analyzes emotional sentiment of some provided text input.

2.4.1 Inference

Use is simple: just pass in a string of text, and get back an emotion or list of emotions.

```
import backprop

emotion = backprop.Emotion()

emotion("I really like what you did there")
"approval"
```

2.4.2 Finetuning

Sentiment detection finetuning is currently a generative task. This will likely be converted to a wrapper around Text Classification in the future.

The schema will remain the same, however: the emotion task params dict contains the keys "input_text" and "output_text". The inputs are the strings to be analysed, and the outputs are the emotions corresponding to those inputs.

Finetuning this task also accepts keyword arguments `max_input_length` and `max_output_length`, which specify the maximum token length for inputs and outputs.

```
import backprop

emote = backprop.Emotion()

# Provide sentiment data for training
inp = ["I really liked the service I received!",
       "Meh, it was not impressive."]

out = ["positive", "negative"]

params = {"input_text": inp, "output_text": out}

# Finetune
emote.finetune(params)
```

See Sentiment Detection Notebook with code.

See the *Emotion Task Reference*.

2.5 Text Summarisation

Also self-explanatory: takes a chunk of input text, and gives a summary of key information.

2.5.1 Inference

```
import backprop

summarisation = backprop.Summarisation()

summarisation("This is a long document that contains plenty of words")
"short summary of document"
```

2.5.2 Finetuning

The summarisation input schema is a params dict with "input_text" and "output_text" keys. Inputs would be longer pieces of text, and the corresponding outputs are summarised versions of the same text.

Finetuning sumamrisation also accepts keyword arguments `max_input_length` and `max_output_length`, which specify the maximum token length for inputs and outputs.

```
import backprop

summary = backprop.Summarisation()

# Provide training data for task
inp = ["This is a long news article about recent political happenings.",
       "This is an article about some recent scientific research."]

out = ["Short political summary.", "Short scientific summary."]

params = {"input_text": inp, "output_text": out}

# Finetune
summary.finetune(params)
```

See the example for Text Summarisation Notebook with code.

See the *Text Summarisation Task Reference*.

2.6 Image Classification

Image classification functions exactly like text classification but for images. It takes an image and a set of labels to calculate the probabilities for each label.

2.6.1 Inference

```
import backprop

ic = backprop.ImageClassification()

ic("/home/Documents/dog.png", ["cat", "dog"])
{"cat": 0.01, "dog": 0.99}
```

2.6.2 Finetuning

The params dict for image classification consists of "images" (input images) and "labels" (image labels). This task also includes variants for single-label and multi-label classification.

```
import backprop

ic = backprop.ImageClassification()

"""
Prep training images/labels. Labels are automatically used to set up
model with number of classes for classification.
"""

images = ["images/beagle/photo.jpg", "images/dachsund/photo.jpg", "images/malamute/
↪photo.jpg"]
labels = ["beagle", "dachsund", "malamute"]
params = {"images": images, "labels": labels}

# Finetune
ic.finetune(params, variant="single_label")
```

Check the example Image Classification Notebook with code.

See the *Image Classification Task Reference*.

2.7 Image Vectorisation

Image Vectorisation takes an image and turns it into a vector.

This makes it possible to compare different images numerically.

2.7.1 Inference

```
import backprop

iv = backprop.ImageVectorisation()

iv("/home/Documents/dog.png")
[0.92949192, 0.23123010, ...]
```

2.7.2 Finetuning

When finetuning image vectorisation, the task input determines on the loss variant you plan to use. This comes in two flavors: triplet, or cosine similarity.

The default is triplet. This schema requires keys "images" (input images), and "groups" (group in which each image falls). This variant uses a distinct sampling strategy, based on group numbers. A given "anchor" image is compared to a positive match (same group number) and a negative match (different group number). The goal is to minimise the distance between the anchor vector and the positive match vector, while also maximising the distance between the anchor vector and negative match vector.

For cosine similarity, the schema is different. It requires keys "imgs1", "imgs2", and "similarity_scores". When training on row *x*, this variant vectorises `imgs1[x]` and `imgs2[x]`, with the target cosine similarity being the value at `similarity_scores[x]`.

```
import backprop

iv = backprop.ImageVectorisation()

# Set up training data & finetune (triplet variant)

images = ["images/beagle/photo.jpg",  "images/shiba_inu/photo.jpg",
          "images/beagle/photo1.jpg", "images/malamute/photo.jpg"]

groups = [0, 1, 0, 2]

params = {"images": images, "groups": groups}

iv.finetune(params, variant="triplet")

# Set up training data & finetune (cosine_similarity variant)

imgs1 = ["images/beagle/photo.jpg", "images/shiba_inu/photo.jpg"]
imgs2 = ["images/beagle/photo1.jpg", "images/malamute/photo.jpg"]

similarity_scores = [1.0, 0.0]

params = {"imgs1": imgs1, "imgs2": imgs2, "similarity_scores": similarity_scores}

iv.finetune(params, variant="cosine_similarity")
```

Check the example Image Vectorisation Notebook with code.

See the *Image Vectorisation Task Reference*.

2.8 Text Generation

Text Generation takes some text as input and generates more text based on it.

This is useful for story/idea generation or solving a broad range of tasks.

2.8.1 Inference

```
import backprop

tg = backprop.TextGeneration()

tg("I like to go to")
" the beach because I love the sun."
```

2.8.2 Finetuning

Text generation requires a params dict with keys "input_text" and "output_text". The values here are simply lists of strings.

When trained, the model will learn expected outputs for a given context – this is how tasks such as generative sentiment detection or text summary can be trained.

Finetuning text generation also accepts keyword arguments `max_input_length` and `max_output_length`, which specify the maximum token length for inputs and outputs.

```
import backprop

tg = backprop.TextGeneration()

# Any text works as training data
inp = ["I really liked the service I received!",
       "Meh, it was not impressive."]

out = ["positive", "negative"]

params = {"input_text": inp, "output_text": out}

# Finetune
tg.finetune(params)
```

Check the example Text Generation Notebook with code.

See the *Text Generation Task Reference*.

2.9 Text Vectorisation

Text Vectorisation takes some text and turns it into a vector.

This makes it possible to compare different texts numerically. You could see how similar the vectors of two different paragraphs are, to group text automatically or build a semantic search engine.

2.9.1 Inference

```
import backprop

tv = backprop.TextVectorisation()

tv("iPhone 12 128GB")
[0.92949192, 0.23123010, ...]
```

2.9.2 Finetuning

When finetuning text vectorisation, the task input determines on the loss variant you plan to use. Like with image vectorisation, this can be either "triplet" or "cosine_similarity".

The default is cosine_similarity. It requires keys "texts1", "texts2", and "similarity_scores". When training on row *x*, this variant vectorises `texts1[x]` and `texts2[x]`, with the target cosine similarity being the value at `similarity_scores[x]`.

Triplet is different. This schema requires keys "texts" (input texts), and "groups" (group in which each piece of text falls). This variant uses a distinct sampling strategy, based on group numbers. A given "anchor" text is compared to a positive match (same group number) and a negative match (different group number). The goal is to minimise the distance between the anchor vector and the positive match vector, while also maximising the distance between the anchor vector and negative match vector.

Finetuning text vectorisation also accepts the keyword argument `max_length` which specifies the maximum token length for encoded text.

```
import backprop

tv = backprop.TextVectorisation()

# Set up training data & finetune (cosine_similarity variant)
texts1 = ["I went to the store and bought some bread",
          "I am getting a cat soon"]

texts2 = ["I bought bread from the store",
          "I took my dog for a walk"]

similarity_scores = [1.0, 0.0]

params = {"texts1": texts1, "texts2": texts2, "similarity_scores": similarity_scores}

tv.finetune(params, variant="cosine_similarity")

# Set up training data & finetune (triplet variant)
texts = ["I went to the store and bought some bread",
         "I bought bread from the store",
         "I'm going to go walk my dog"]

groups = [0, 0, 1]

params = {"texts": texts, "groups": groups}

tv.finetune(params, variant="triplet")
```

Check the example Text Vectorisation Notebook with code.

See the *Text Vectorisation Task Reference*.

2.10 Image-Text Vectorisation

Image-Text Vectorisation takes an associated text/image pair, and returns a normalized vector output.

This task could be used for making a robust image search system, that takes into account both input text and similar images.

2.10.1 Inference

```
import backprop

itv = backprop.ImageTextVectorisation()

image = "images/iphone/iphone-12-128GB.jpg"
text = "iPhone 12 128GB"

tv(image=image, text=text)
[0.82514237, 0.35281924, ...]
```

2.10.2 Finetuning

Similar to the other vectorisation tasks (text & image separately), this task has both triplet and cosine similarity loss variants. The variant determines the input data schema.

The default is triplet. This params dict requires keys "images" (input images), "texts" (input texts) and "groups" (group in which each image/text pair falls). This variant uses a distinct sampling strategy, based on group numbers. A given "anchor" image/text pair is compared to a positive match (same group number) and a negative match (different group number). The goal is to minimise the distance between the anchor vector and the positive match vector, while also maximising the distance between the anchor vector and negative match vector.

For cosine similarity, a few things are needed. It requires keys "imgs1", "imgs2", "texts1", "texts2", and "similarity_scores". When training on row *x*, this variant gets a normalized vector for `imgs1[x]` and `texts[x]`, as well as one for and `imgs2[x]` and `texts2[x]`. The target cosine similarity between both normalized vectors is the value at `similarity_scores[x]`.

```
import backprop

itv = backprop.ImageTextVectorisation()

# Prep training data & finetune (triplet variant)
images = ["product_images/crowbars/photo.jpg",
          "product_images/crowbars/photo1.jpg",
          "product_images/mugs/photo.jpg"]

texts = ["Steel crowbar with angled beak, 300mm",
         "Crowbar tempered steel 300m angled",
         "Sturdy ceramic mug, microwave-safe"]

groups = [0, 0, 1]

params = {"images": images, "texts": texts, "groups": groups}
```

(continues on next page)

(continued from previous page)

```
itv.finetune(params, variant="triplet")

# Prep training data & finetune (cosine_similarity variant)
imgs1 = ["product_images/crowbars/photo.jpg", "product_images/mugs/photo.jpg"]
texts1 = ["Steel crowbar with angled beak, 300mm", "Sturdy ceramic mug, microwave-safe
↪"]

imgs2 = ["product_images/crowbars/photo1.jpg", "product_images/hats/photo.jpg]
texts2 = ["Crowbar tempered steel 300m angled", "Dad hat with funny ghost picture on␣
↪the front"]

similarity_scores = [1.0, 0.0]
params = {"imgs1": imgs1,
          "imgs2": imgs2,
          "texts1": texts1,
          "texts2": texts2,
          "similarity_scores": similarity_scores}

itv.finetune(params, variant="cosine_similarity")
```

CHAPTER

THREE

MODELS

Models are classes that power tasks. 99% of users should not have to use models directly. Instead, you should use the appropriate *Tasks* where you specify the model.

See the available models and tasks they support in the Backprop Model Hub.

The only valid reason for using models directly is if you are implementing your own.

You can import models from `backprop.models`.

See the full model reference in *backprop.models*.

3.1 Model Requirements

A valid model must:

- implement one or more tasks according to the task's input schema
- have a valid list of tasks (strings) as a tasks field
- have a name field that must be between 3 and 100 lowercase a-z characters, with numbers, dashes (-) and underscores (_) allowed.
- produce JSON serializable output
- not have any unspecified external dependencies
- be pickleable with `dill`

If these criteria are fulfilled, then it is very easy to save, load and upload the model.

A model can:

- offer finetuning support by implementing the `process_batch`, `training_step` and optionally `pre_finetuning` methods
- support single and batched task requests

Every model included in our library fulfils the necessary criteria. Our models also support both single and batched requests. For example, you can vectorise text with both `"some text"` and `["first text", "second text"]` as input.

3.2 Auto Model

`AutoModel` is a special class that can load any supported model by its string identifier. You can also use it to see what models are available.

Example usage:

```
from backprop.models import AutoModel

AutoModel.list_models(display=True, limit=10)

model = AutoModel.from_pretrained("distilgpt2")
```

3.3 Generic Models

Generic models are used to implement more specific models. Generic models don't support any tasks out of the box. When implementing a model, it is useful to inherit from a generic model to ensure it fits in with other Backprop modules.

Example usage:

```
from backprop.models import PathModel

from transformers import AutoModelForPreTraining, AutoTokenizer

model = PathModel("t5-base", init_model=AutoModelForPreTraining, init_
↪tokenizer=AutoTokenizer)

# Use model
model([0, 1, 2])
```

See an example how to implement a generic model for a task.

See the generic models reference in *backprop.models*.

3.4 Supporting Task inference with Models

In order for a model to support inference for a task, it must follow the task's input schema.

For each task, the input consists of an argument and a keyword argument.

This is passed by calling the model object directly.

```
from backprop.models import BaseModel

class MyModel(BaseModel):
    def __call__(self, task_input, task="emotion"):
        if task == "emotion":
            text = task_input.get("text")
            # Do some AI magic with text, assume result is "admiration"
            return "admiration"
        else:
            raise ValueError("Unsupported task!")
```

(continues on next page)

(continued from previous page)

```
model = MyModel()

# Use model
model({"text": "This is pretty cool!"}, task="emotion")
"admiration"
```

The input argument is a dictionary, while the keyword argument `task` is a string.

3.4.1 Q&A

Task string is `"qa"`.

Dictionary argument specification:

key	type	description
question	`str` or `List[str]`	question or list of questions
context	`str` or `List[str]`	context or list of contexts
prev_q	`List[str]` or `List[List[str]]`	List of previous questions or list of previous question lists
prev_a	`List[str]` or `List[List[str]]`	List of previous answers or list of previous answer lists

3.4.2 Text Classification

Task string is `"text-classification"`.

Dictionary argument specification:

key	type	description
text	`str` or `List[str]`	text or list of texts to classify
labels	`List[str]` or `List[List[str]]`	optional (zero-shot) labels or list of labels to assign probabilities to
top_k	`int`	optional number of highest probability labels to return

3.4.3 Sentiment Detection (Emotion)

Task string is `"emotion"`.

key	type	description
text	`str` or `List[str]`	text or list of texts to detect emotion from

3.4.4 Text Summarisation

Task string is `"summarisation"`.

key	type	description
text	`str` or `List[str]`	text or list of texts to summarise

3.4.5 Image Classification

Task string is `"image-classification"`.

key	type	description
image	`str` or `List[str]` or `PIL.Image` or `List[PIL.Image]`	PIL or base64 encoded image or list of them
labels	`List[str]` or `List[List[str]]`	optional (zero-shot) labels or list of labels to assign probabilities to
top_k	`int`	optional number of highest probability labels to return

3.4.6 Image Vectorisation

Task string is `"image-vectorisation"`.

key	type	description
image	`str` or `List[str]` or `PIL.Image` or `List[PIL.Image]`	PIL or base64 encoded image or list of them

3.4.7 Image-Text Vectorisation

Task string is `"image-text-vectorisation"`.

key	type	description
image	`str` or `List[str]` or `PIL.Image` or `List[PIL.Image]`	PIL or base64 encoded image or list of them
text	`str` or `List[str]`	text or list of texts to vectorise

3.4.8 Text Generation

Task string is `"text-generation"`.

key	type	description
text	`str` or `List[str]`	text or list of texts to generate from
min_length	`int`	minimum number of tokens to generate
max_length	`int`	maximum number of tokens to generate
temperature	`float`	value that alters softmax probabilities
top_k	`float`	sampling strategy in which probabilities are redistributed among top k most-likely words
top_p	`float`	sampling strategy in which probabilities are distributed among set of words with combined probability greater than p
repetition_penalty	`float`	penalty to be applied to words present in the text and words already generated in the sequence
length_penalty	`float`	penalty applied to overall sequence length. >1 for longer sequences, or <1 for shorter ones
num_beams	`int`	number of beams to be used in beam search
num_generations	`int`	number of times to generate
do_sample	`bool`	whether to sample or do greedy search

3.4.9 Text Vectorisation

Task string is `"text-vectorisation"`.

key	type	description
text	`str` or `List[str]`	text or list of texts to vectorise

3.5 Supporting Task finetuning with Models

In order for a model to support finetuning for a task, it must follow the task's finetuning schema.

This involves implementing three methods:

1. `process_batch` - receive task specific data and process it
2. `training_step` - receive data processed by the `process_batch` method and produce output
3. `pre_finetuning` - optionally receive task specific parameters and adjust the model before finetuning

The inputs and outputs for each of these methods vary depending on the task.

3.5.1 Q&A

`process_batch` takes dictionary argument `params` and keyword argument `task="qa"`.

`params` has the following keys and values:

key	type	description
question	`str`	Question
context	`str`	Context that contains answer
prev_qa	`List[Tuple[str, str]]`	List of previous question-answer pairs
output	`str`	Answer
max_input_length	`int`	Max number of tokens in input
max_output_length	`int`	Max number of tokens in output

`training_step` must return loss.

`pre_finetuning` is not used.

3.5.2 Text Classification

Currently, only the single label variant is supported.

`process_batch` takes dictionary argument `params` and keyword argument `task="text-classification"`.

`params` has the following keys and values:

key	type	description
inputs	`str`	Text
class_to_idx	`str`	Maps labels to integers
labels	`str`	Correct label
max_length	`str`	Max number of tokens in inputs

`training_step` must return loss.

`pre_finetuning` takes labels argument which is a dictionary that maps integers (from 0 to n) to labels.

3.5.3 Sentiment Detection (Emotion)

`process_batch` takes dictionary argument `params` and keyword argument `task="emotion"`.

`params` has the following keys and values:

key	type	description
input	`str`	Text to detect emotion from
output	`str`	Emotion text
max_input_length	`int`	Max number of tokens in input
max_output_length	`int`	Max number of tokens in output

`training_step` must return loss.

`pre_finetuning` is not used.

3.5.4 Text Summarisation

`process_batch` takes dictionary argument `params` and keyword argument `task="summarisation"`.

`params` has the following keys and values:

key	type	description
input	`str`	Text to summarise
output	`str`	Summary
max_input_length	`int`	Max number of tokens in input
max_output_length	`int`	Max number of tokens in output

`training_step` must return loss.

`pre_finetuning` is not used.

3.5.5 Image Classification

`process_batch` takes dictionary argument `params` and keyword argument `task="image-classification"`.

`params` has the following keys and values:

key	type	description
image	`str`	Path to image

`training_step` must return logits for each class (label).

`pre_finetuning` takes:

- `labels` keyword argument which is a dictionary that maps integers (from 0 to n) to labels.
- `num_classes` keyword argument which is an integer for the number of unique labels.

3.5.6 Image Vectorisation

`process_batch` takes dictionary argument `params` and keyword argument `task="image-vectorisation"`.

`params` has the following keys and values:

key	type	description
image	`str`	Path to image

`training_step` must return vector tensor.

`pre_finetuning` takes no arguments.

3.5.7 Text Generation

`process_batch` takes dictionary argument `params` and keyword argument `task="text-generation"`.

`params` has the following keys and values:

key	type	description
input	`str`	Generation prompt
output	`str`	Generation outpu
max_input_length	`int`	Max number of tokens in input
max_output_length	`int`	Max number of tokens in output

`training_step` must return loss.

`pre_finetuning` is not used.

3.5.8 Text Vectorisation

`process_batch` takes dictionary argument `params` and keyword argument `task="text-vectorisation"`.

`params` has the following keys and values:

key	type	description
text	`str`	Text to vectorise

`training_step` must return vector tensor.

`pre_finetuning` takes no arguments.

3.5.9 Image-Text Vectorisation

`process_batch` takes dictionary argument `params` and keyword argument `task="image-text-vectorisation"`.

`params` has the following keys and values:

key	type	description
image	`str`	Path to image
text	`str`	Text to vectorise

`training_step` must return vector tensor.

`pre_finetuning` takes no arguments.

CHAPTER

FOUR

FINETUNING

Finetuning lets you take a model that has been trained on a very broad task and adapt it to your specific niche.

4.1 Finetuning Parameters

There are a variety of parameters that can optionally be supplied when finetuning a task, to allow for more flexibility.

For references on each task's data input schema, find your task here.

- `validation_split` : Float value that determines the percentage of data that will be used for validation.
- `epochs` : Integer determining how many training iterations will be run while finetuning.
- `batch_size` : Integer specifying the batch size for training. Leavings this out lets Backprop determine it automatically for you.
- `optimal_batch_size` : Integer indicating the optimal batch size for the model to be used. This is model-specific, so in most cases will not need to be supplied.
- `early_stopping_epochs` : Integer value. When training, early stopping is a mechanism that determines a model has finished training based on lack of improvements to validation loss. This parameter indicates how many epochs will continue to run without seeing an improvement to validation loss. Default value is *1*.
- `train_dataloader` : DataLoader that will be used to pull batches from a dataset. We default this to be a DataLoader with the maximum number of workers (determined automatically by CPU).
- `val_dataloader` : The same as `train_dataloader`, for validation data.

Along with these parameters, finetuning has two keyword arguments that are functions, used for further customization.

- `step` : This function determines how a batch will be supplied to your chosen model, and returns loss. All of our included models/tasks have a default `step`, but for custom models, you can define exactly how to pass training data and calculate loss.
- `configure_optimizers` : Sets up an optimizer for use in training. As with `step`, we include optimizers suited for each particular task. However, if you wish to experiment with other options, you can simply define a function that returns your chosen optimzer setup.

4.2 Basic Example

Here is a simple example of finetuning for text generation with T5.

```
from backprop.models import T5
from backprop import TextGeneration

tg = TextGeneration(T5)

# Any text works as training data
inp = ["I really liked the service I received!", "Meh, it was not impressive."]
out = ["positive", "negative"]
params = {"input_text": inp, "output_text": out}

# Finetune with a single line of code
tg.finetune(params)

# Use your trained model
prediction = tg("I enjoyed it!")
```

4.3 In-depth Example

See the in-depth Getting Started with Finetuning notebook with code.

4.4 Supported tasks

To see all models that a task supports for finetuning, load that task and call `.list_models()`

4.4.1 Text Generation

References:

- *text-generation task*
- text-generation finetuning notebook

4.4.2 Image Classification

References:

- *image-classification task*
- image-classification finetuning notebook

4.4.3 Text Classification

References:

- *text-classification task*
- text-classification finetuning notebook

4.4.4 Text Vectorisation

References:

- *text-vectorisation task*
- text-vectorisation finetuning notebook

Image-Text Vectorisation

References:

- *image-text-vectorisation task*

CHAPTER

FIVE

UTILS

Functions from `backprop.utils` are used for model inference, finetuning, saving, loading and uploading.

This page covers saving, loading and uploading. See the full reference in *backprop.utils*.

5.1 Save

It is recommended to save a model via a task instead (i.e `task.save()`)

save(*model*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *path=None*)

Saves the provided model to the backprop cache folder using:

1. provided name
2. model.name
3. provided path

The resulting folder has three files:

- model.bin (dill pickled model instance)
- config.json (description and task keys)
- requirements.txt (exact python runtime requirements)

Parameters

- **model** – Model object
- **name** – string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.
- **description** – String description of the model.
- **tasks** – List of supported task strings
- **details** – Valid json dictionary of additional details about the model
- **path** – Optional path to save model

Example:

```
import backprop

backprop.save(model_object, "my_model")
model = backprop.load("my_model")
```

5.2 Load

load(*path*)
Loads a saved model and returns it.

Parameters **path** – Name of the model or full path to model.

Example:

```
import backprop

backprop.save(model_object, "my_model")
model = backprop.load("my_model")
```

5.3 Upload

It is recommended to upload a model via a task instead (i.e `task.upload()`)

For a successful upload, ensure that the model is valid by following the check list in *Models*.

upload(*model*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *path=None*, *api_key: Optional[str] = None*)
Saves and deploys a model to Backprop.

Parameters

- **model** – Model object
- **api_key** – Backprop API key
- **name** – string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.
- **description** – String description of the model.
- **tasks** – List of supported task strings
- **details** – Valid json dictionary of additional details about the model
- **path** – Optional path to save model

Example:

```
import backprop

tg = backprop.TextGeneration("t5_small")

# Any text works as training data
inp = ["I really liked the service I received!", "Meh, it was not impressive."]
out = ["positive", "negative"]

# Finetune with a single line of code
tg.finetune({"input_text": inp, "output_text": out})

# Use your trained model
prediction = tg("I enjoyed it!")

print(prediction)
# Prints
```

(continues on next page)

(continued from previous page)

```
"positive"

# Upload to Backprop for production ready inference

model = tg.model
# Describe your model
name = "t5-sentiment"
description = "Predicts positive and negative sentiment"

backprop.upload(model, name=name, description=description, api_key="abc")
```

CHAPTER

SIX

BACKPROP.MODELS

6.1 backprop.models.clip

6.1.1 backprop.models.clip.clip

available_models()

load(*name: str*, *device: Union[str, torch.device] = 'cpu'*, *jit=False*)

tokenize(*tokenizer*, *texts: Union[str, List[str]]*, *context_length: int = 77*, *truncation=True*)

6.1.2 backprop.models.clip.model

class AttentionPool2d(*spacial_dim: int*, *embed_dim: int*, *num_heads: int*, *output_dim: Optional[int] = None*)

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

forward(*x*)

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.

training: bool

class Bottleneck(*inplanes*, *planes*, *stride=1*)

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

expansion = 4

forward(*x: torch.Tensor*)

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.

training: bool

class CLIP(*embed_dim: int*, *image_resolution: int*, *vision_layers: Union[Tuple[int, int, int, int], int]*, *vision_width: int*, *vision_patch_size: int*, *context_length: int*, *vocab_size: int*, *transformer_width: int*, *transformer_heads: int*, *transformer_layers: int*)

Bases: torch.nn.modules.module.Module

build_attention_mask()

property dtype

encode_image(*image*)

encode_text(*text*)

forward(*image*, *text*)

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.

---

initialize_parameters()

training: bool

class LayerNorm(*normalized_shape: Union[int, List[int], torch.Size]*, *eps: float = 1e-05*, *elementwise_affine: bool = True*)

Bases: torch.nn.modules.normalization.LayerNorm

Subclass torch's LayerNorm to handle fp16.

elementwise_affine: bool

eps: float

forward(*x: torch.Tensor*)

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.

---

normalized_shape: Union[int, List[int], torch.Size]

class ModifiedResNet(*layers*, *output_dim*, *heads*, *input_resolution=224*, *width=64*)

Bases: torch.nn.modules.module.Module

A ResNet class that is similar to torchvision's but contains the following changes: - There are now 3 "stem" convolutions as opposed to 1, with an average pool instead of a max pool. - Performs anti-aliasing strided convolutions, where an avgpool is prepended to convolutions with stride > 1 - The final pooling layer is a QKV attention instead of an average pool

forward(*x*)

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.

training: bool

class QuickGELU

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

forward(*x: torch.Tensor*)

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.

training: bool

class ResidualAttentionBlock(*d_model: int*, *n_head: int*, *attn_mask: Optional[torch.Tensor] = None*)

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

attention(*x: torch.Tensor*)

forward(*x: torch.Tensor*)

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.

training: bool

class Transformer(*width: int*, *layers: int*, *heads: int*, *attn_mask: Optional[torch.Tensor] = None*)

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

forward(*x: torch.Tensor*)

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.

training: bool

class VisualTransformer(*input_resolution: int*, *patch_size: int*, *width: int*, *layers: int*, *heads: int*, *output_dim: int*)

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

forward (*x: torch.Tensor*)
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.

---

training: bool

build_model (*state_dict: dict*)

convert_weights (*model: torch.nn.modules.module.Module*)
Convert applicable model parameters to fp16

6.1.3 backprop.models.clip.models_list

6.1.4 backprop.models.clip.module

class CLIP (*model_path='ViT-B/32', init_model=<function load>, init_tokenizer=<class 'backprop.models.clip.simple_tokenizer.SimpleTokenizer'>, name: Optional[str] = None, description: Optional[str] = None, tasks: Optional[List[str]] = None, details: Optional[Dict] = None, device=None*)
Bases: *backprop.models.generic_models.BaseModel*

CLIP is a recent model by OpenAI.

model_path
ViT-B/32, RN50, RN101, RN50x4

init_model
initialise model from model_path

init_tokenizer
initializes tokenizer

name
string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

description
String description of the model.

tasks
List of supported task strings

details
Dictionary of additional details about the model

device
Device for model. Defaults to "cuda" if available.

__call__ (*task_input, task='image-classification', return_tensor=False*)
Do inference with the model.

Parameters

- **task_input** – input dictionary according to task
- **task** – one of supported tasks

- **return_tensor** – return a tensor instead of list for vectorisation output

image_classification(*image: torch._C.TensorType*, *text: torch._C.TensorType*, *labels*, *top_k=10000*)

image_text_vectorisation(*image: torch._C.TensorType*, *text: torch._C.TensorType*)

image_vectorisation(*image: torch._C.TensorType*)

static list_models()

process_batch(*params*, *task*)

text_vectorisation(*text: torch._C.TensorType*)

training: bool

training_step(*params*, *task*)

6.1.5 backprop.models.clip.simple_tokenizer

class SimpleTokenizer

Bases: `object`

bpe(*token*)

decode(*tokens*)

encode(*text*)

basic_clean(*text*)

bytes_to_unicode()

Returns list of utf-8 byte and a corresponding list of unicode strings. The reversible bpe codes work on unicode strings. This means you need a large # of unicode characters in your vocab if you want to avoid UNKs. When you're at something like a 10B token dataset you end up needing around 5K for decent coverage. This is a signficant percentage of your normal, say, 32K bpe vocab. To avoid that, we want lookup tables between utf-8 bytes and unicode strings. And avoids mapping to whitespace/control characters the bpe code barfs on.

default_bpe()

get_pairs(*word*)

Return set of symbol pairs in a word. Word is represented as tuple of symbols (symbols being variable-length strings).

whitespace_clean(*text*)

6.2 backprop.models.efficientnet

6.2.1 backprop.models.efficientnet.model

class EfficientNet(*model_path: str = 'efficientnet-b0'*, *init_model=None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *device=None*)

Bases: *backprop.models.generic_models.PathModel*

EfficientNet is a very efficient image-classification model. Trained on ImageNet.

model_path

Any efficientnet model (smaller to bigger) from efficientnet-b0 to efficientnet-b7

init_model
Callable that initialises the model from the model_path

name
string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

description
String description of the model.

tasks
List of supported task strings

details
Dictionary of additional details about the model

device
Device for model. Defaults to "cuda" if available.

__call__(*task_input*, *task='image-classification'*)
Uses the model for the image-classification task

Parameters

- **task_input** – input dictionary according to the `image-classification` task specification
- **task** – image-classification

configure_optimizers()

image_classification(*image*, *top_k=10000*)

static list_models()

pre_finetuning(*labels=None*, *num_classes=None*)

process_batch(*params*, *task='image-classification'*)

training: bool

training_step(*batch*, *task='image-classification'*)

6.2.2 backprop.models.efficientnet.models_list

6.3 backprop.models.hf_causallm_tg_model

6.3.1 backprop.models.hf_causallm_tg_model.model

class HFCausalLMTGModel(*model_path=None*, *tokenizer_path=None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *details: Optional[Dict] = None*, *tasks: Optional[List[str]] = None*, *model_class=<class 'transformers.models.auto.modeling_auto.AutoModelForCausalLM'>*, *tokenizer_class=<class 'transformers.models.auto.tokenization_auto.AutoTokenizer'>*, *device=None*)
Bases: *backprop.models.generic_models.HFTextGenerationModel*

Class for Hugging Face causal LM models

model_path
path to HF model

tokenizer_path
path to HF tokenizer

name
string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

description
String description of the model.

tasks
List of supported task strings

details
Dictionary of additional details about the model

model_class
Class used to initialise model

tokenizer_class
Class used to initialise tokenizer

device
Device for model. Defaults to "cuda" if available.

__call__(*task_input*, *task='text-generation'*)
Uses the model for the text-generation task

Parameters

- **task_input** – input dictionary according to the `text-generation` task specification.
- **task** – text-generation

static list_models()

training: bool

6.3.2 backprop.models.hf_causallm_tg_model.models_list

6.4 backprop.models.hf_nli_model

6.4.1 backprop.models.hf_nli_model.model

class HFNLIModel(*model_path=None*, *tokenizer_path=None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *model_class=<class 'transformers.models.auto.modeling_auto.AutoModelForSequenceClassification'>*, *tokenizer_class=<class 'transformers.models.auto.tokenization_auto.AutoTokenizer'>*, *device=None*)
Bases: *backprop.models.generic_models.HFModel*

Class for Hugging Face sequence classification models trained on a NLI dataset

model_path
path to HF model

tokenizer_path
path to HF tokenizer

name
string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

description
String description of the model.

tasks
List of supported task strings

details
Dictionary of additional details about the model

model_class
Class used to initialise model

tokenizer_class
Class used to initialise tokenizer

device
Device for model. Defaults to "cuda" if available.

__call__(*task_input*, *task='text-classification'*)
Uses the model for the text-classification task

Parameters

- **task_input** – input dictionary according to the `text-classification` task specification. Needs labels (for zero-shot).
- **task** – text-classification

calculate_probability(*text*, *labels*)

classify(*text*, *labels*, *top_k*)
Classifies text, given a set of labels.

static list_models()

training: bool

6.4.2 backprop.models.hf_nli_model.models_list

6.5 backprop.models.hf_seq2seq_tg_model package

6.5.1 backprop.models.hf_seq2seq_tg_model.model

class HFSeq2SeqTGModel(*model_path=None*, *tokenizer_path=None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *details: Optional[Dict] = None*, *tasks: Optional[List[str]] = None*, *model_class=<class 'transformers.models.auto.modeling_auto.AutoModelForSeq2SeqLM'>*, *tokenizer_class=<class 'transformers.models.auto.tokenization_auto.AutoTokenizer'>*, *device=None*)
Bases: *backprop.models.generic_models.HFTextGenerationModel*

Class for Hugging Face causal Seq2Seq generation models.

model_path
path to HF model

tokenizer_path
path to HF tokenizer

name
string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

description
String description of the model.

tasks
List of supported task strings

details
Dictionary of additional details about the model

model_class
Class used to initialise model

tokenizer_class
Class used to initialise tokenizer

device
Device for model. Defaults to "cuda" if available.

__call__(*task_input*, *task='text-generation'*)
Uses the model for the text-generation task

Parameters

- **task_input** – input dictionary according to the `text-generation` task specification
- **task** – text-generation

encode_input(*text*, *max_length=128*)

encode_output(*text*, *max_length=32*)

static list_models()

process_batch(*params*, *task*)

training: bool

training_step(*task_input*)

6.5.2 backprop.models.hf_seq2seq_tg_model.models_list

6.6 backprop.models.hf_seq_tc_model

6.6.1 backprop.models.hf_seq_tc_model.model

class HFSeqTCModel(*model_path=None*, *tokenizer_path=None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *model_class=<class 'transformers.models.auto.modeling_auto.AutoModelForSequenceClassification'>*, *tokenizer_class=<class 'transformers.models.auto.tokenization_auto.AutoTokenizer'>*, *device=None*)
Bases: *backprop.models.generic_models.HFModel*

Class for Hugging Face sequence classification models

model_path
path to HF model

tokenizer_path
path to HF tokenizer

name
string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

description
String description of the model.

tasks
List of supported task strings

details
Dictionary of additional details about the model

model_class
Class used to initialise model

tokenizer_class
Class used to initialise tokenizer

device
Device for model. Defaults to "cuda" if available.

__call__(*task_input*, *task='text-classification'*, *train=False*)
Uses the model for text classification. At this point, the model needs to already have been finetuned. This is what sets up the final layer for classification.

Parameters

- **task_input** – input dictionary according to the `text-classification` task specification
- **task** – text-classification

encode(*text*, *target*, *max_input_length=128*)

get_label_probabilities(*outputs*, *top_k*)

init_pre_finetune(*labels*)

static list_models()

process_batch(*params*, *task='text-classification'*)

training: bool

training_step(*batch*, *task='text-classification'*)

6.6.2 backprop.models.hf_seq_tc_model.models_list

6.7 backprop.models.st_model

6.7.1 backprop.models.st_model.model

class STModel(*model_path*, *init_model=<class 'sentence_transformers.SentenceTransformer.SentenceTransformer'>*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *max_length=512*, *device=None*)
Bases: *backprop.models.generic_models.PathModel*

Class for models which are initialised from Sentence Transformers

model_path
path to ST model

name
string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

max_length
Max supported token length for vectorisation

description
String description of the model.

tasks
List of supported task strings

details
Dictionary of additional details about the model

init_model
Class used to initialise model

device
Device for model. Defaults to "cuda" if available.

__call__(*task_input*, *task='text-vectorisation'*, *return_tensor=False*)
Uses the model for the text-vectorisation task

Parameters

- **task_input** – input dictionary according to the `text-vectorisation` task specification
- **task** – text-vectorisation

configure_optimizers()

static list_models()

process_batch(*params*, *task='text-vectorisation'*)

training: bool

training_step(*params*, *task='text-vectorisation'*)

vectorise(*features*)

6.7.2 backprop.models.st_model.models_list

6.8 backprop.models.t5_qa_summary_emotion

6.8.1 backprop.models.t5_qa_summary_emotion.model

class T5QASummaryEmotion(*model_path=None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *details: Optional[Dict] = None*, *tasks: Optional[List[str]] = None*, *device=None*)
Bases: *backprop.models.hf_seq2seq_tg_model.model.HFSeq2SeqTGModel*

Initialises a T5 model that has been finetuned on qa, summarisation and emotion detection.

model_path
path to an appropriate T5 model on huggingface (kiri-ai/t5-base-qa-summary-emotion)

name
string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

description
String description of the model.

tasks
List of supported task strings

details
Dictionary of additional details about the model

device
Device for model. Defaults to "cuda" if available.

__call__(*task_input*, *task='text-generation'*)
Uses the model for the chosen task

Parameters

- **task_input** – input dictionary according to the chosen task's specification
- **task** – one of text-generation, emotion, summarisation, qa

emote_or_summary(*text*, *task_prefix*, ***gen_kwargs*)

encode_input(*inp*, *max_length*)

encode_output(*out*, *max_length*)

static list_models()

process_batch(*params*, *task*)

process_qa(*question*, *context*, *prev_qa*)

qa(*question*, *context*, *prev_qa: List[Tuple[str, str]] = []*)

training: bool

training_step(*task_input*)

6.8.2 backprop.models.t5_qa_summary_emotion.models_list

6.9 Submodules

6.10 backprop.models.auto_model

class AutoModel
Bases: object

static from_pretrained(*model_name: str*, *aliases: Optional[Dict] = None*, *device: Optional[str] = None*)
Loads a model by name

Parameters

- **model_name** – unique name of the model
- **aliases** – dictionary that maps aliases to model_name
- **device** – device to use model on. Defaults to "cuda" if available

Returns Initialised model object

Example:

```
import backprop

model = backprop.models.AutoModel.from_pretrained("t5_small")
```

static list_models(*task=None*, *return_dict=False*, *display=False*, *limit=None*, *aliases: Optional[Dict] = None*)

Lists available models

Parameters

- **task** – filter by task identifier
- **return_dict** – whether to return dictionary instead of a list
- **display** – print instead of returning
- **limit** – maximum number of models to include
- **aliases** – dict that maps aliases to model_name

Example:

```
import backprop
backprop.models.AutoModel.list_models(task="text-vectorisation", display=True)

> Name         clip
  Description  Model by OpenAI
  ...
```

6.11 backprop.models.generic_models

class BaseModel(*model*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*)

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

The base class for a model.

model
Your model that takes some args, kwargs and returns an output. Must be callable.

name
string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

description
String description of the model.

tasks
List of supported task strings

details
Dictionary of additional details about the model

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

Returns self

Return type Module

finetune(**args*, ***kwargs*)

num_parameters()

to(*device*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
```

(continues on next page)

(continued from previous page)

```
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train (*mode: bool = True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

class HFModel (*model_path*, *tokenizer_path=None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *model_class=<class 'transformers.models.auto.modeling_auto.AutoModelForPreTraining'>*, *tokenizer_class=<class 'transformers.models.auto.tokenization_auto.AutoTokenizer'>*, *device=None*)

Bases: *backprop.models.generic_models.PathModel*

Class for huggingface models

model_path

Local or huggingface.co path to the model

name

string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

description

String description of the model.

tasks

List of supported task strings

details

Dictionary of additional details about the model

init_model

Callable to initialise model from path Defaults to AutoModelForPreTraining from huggingface

tokenizer_path
Path to the tokenizer

Type optional

init_tokenizer
Callable to initialise tokenizer from path Defaults to AutoTokenizer from huggingface.

Type optional

device
Device for inference. Defaults to "cuda" if available.

Type optional

training: bool

class HFTextGenerationModel(*model_path*, *tokenizer_path=None*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *model_class=<class 'transformers.models.auto.modeling_auto.AutoModelForPreTraining'>*, *tokenizer_class=<class 'transformers.models.auto.tokenization_auto.AutoTokenizer'>*, *device=None*)
Bases: *backprop.models.generic_models.HFModel*

Class for huggingface models that implement the .generate method.

***args and **kwargs are passed to HFModel's __init__**

generate(*text*, *variant='seq2seq'*, ***kwargs*)
Generate according to the model's generate method.

training: bool

class PathModel(*model_path*, *init_model*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *tokenizer_path=None*, *init_tokenizer=None*, *device=None*)
Bases: *backprop.models.generic_models.BaseModel*

Class for models which are initialised from a path.

model_path
Path to the model

name
string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.

description
String description of the model.

tasks
List of supported task strings

details
Dictionary of additional details about the model

init_model
Callable to initialise model from path

tokenizer_path
Path to the tokenizer

Type optional

init_tokenizer
Callable to initialise tokenizer from path

Type optional

device
Device for inference. Defaults to “cuda” if available.

Type optional

training: bool

6.12 Module contents

CHAPTER

SEVEN

BACKPROP.TASKS

7.1 backprop.tasks.base

class Task(*model*, *local=False*, *api_key=None*, *task: Optional[str] = None*, *device: Optional[str] = None*, *models: Optional[Dict] = None*, *default_local_model: Optional[str] = None*, *local_aliases: Optional[Dict] = None*)

Bases: `pytorch_lightning.core.lightning.LightningModule`

Base Task superclass used to implement new tasks.

model
Model name string for the task in use.

local
Run locally. Defaults to False.

api_key
Backprop API key for non-local inference.

device
Device to run inference on. Defaults to "cuda" if available.

models
All supported models for a given task (pulls from config).

default_local_model
Which model the task will default to if initialized with none provided: Defined per-task.

configure_optimizers()
Sets up optimizers for model. Must be defined in task: no base default.

finetune(*dataset=None*, *validation_split: Union[float, Tuple[List[int], List[int]]] = 0.15*, *epochs: int = 20*, *batch_size: Optional[int] = None*, *optimal_batch_size: Optional[int] = None*, *early_stopping_epochs: int = 1*, *train_dataloader=None*, *val_dataloader=None*, *dataset_train: Optional[torch.utils.data.dataset.Dataset] = None*, *dataset_valid: Optional[torch.utils.data.dataset.Dataset] = None*, *step=None*, *configure_optimizers=None*)

save(*name: str*, *description: Optional[str] = None*, *details: Optional[Dict] = None*)
Saves the model used by task to `~/.cache/backprop/name`

Parameters

- **name** – string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.
- **description** – String description of the model.
- **details** – Valid json dictionary of additional details about the model

step(*batch*, *batch_idx*)
Implemented per-task, passes batch into model and returns loss.

Parameters

- **batch** – Batch output from dataloader.
- **batch_idx** – Batch index.

train_dataloader()
Returns a default dataloader of training data.

training: bool

training_step(*batch*, *batch_idx*)
Performs the step function with training data and gets training loss.

Parameters

- **batch** – Batch output from dataloader.
- **batch_idx** – Batch index.

upload(*name: str*, *description: Optional[str] = None*, *details: Optional[Dict] = None*, *api_key: Optional[str] = None*)
Saves the model used by task to `~/.cache/backprop/name` and deploys to backprop

Parameters

- **name** – string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.
- **description** – String description of the model.
- **details** – Valid json dictionary of additional details about the model
- **api_key** – Backprop API key

val_dataloader()
Returns a default dataloader of validation data.

validation_step(*batch*, *batch_idx*)
Performs the step function with validation data and gets validation loss.

Parameters

- **batch** – Batch output from dataloader.
- **batch_idx** – Batch index.

7.2 backprop.tasks.emotion

class Emotion(*model: Optional[Union[str, backprop.models.generic_models.BaseModel]] = None*, *local: bool = False*, *api_key: Optional[str] = None*, *device: Optional[str] = None*)
Bases: *backprop.tasks.base.Task*

Task for emotion detection.

model

1. Model name
2. Model name on Backprop's emotion endpoint
3. Model object that implements the emotion task

local
Run locally. Defaults to False

Type optional

api_key
Backprop API key for non-local inference

Type optional

device
Device to run inference on. Defaults to "cuda" if available.

Type optional

__call__(*text: Union[str, List[str]]*)
Perform emotion detection on input text.

Parameters **text** – string or list of strings to detect emotion from keep this under a few sentences for best performance.

Returns Emotion string or list of emotion strings.

configure_optimizers()
Returns default optimizer for text generation (AdaFactor, learning rate 1e-3)

finetune(*params*, *validation_split: Union[float, Tuple[List[int], List[int]]] = 0.15*, *max_input_length: int = 256*, *max_output_length: int = 32*, *epochs: int = 20*, *batch_size: Optional[int] = None*, *optimal_batch_size: Optional[int] = None*, *early_stopping_epochs: int = 1*, *train_dataloader=None*, *val_dataloader=None*, *step=None*, *configure_optimizers=None*)
Finetunes a generative model for sentiment detection.

Note: input_text and output_text in params must have matching ordering (item 1 of input must match item 1 of output)

Parameters

- **params** – Dictionary of model inputs. Contains 'input_text' and 'output_text' keys, with values as lists of input/output data.
- **max_input_length** – Maximum number of tokens (1 token ~ 1 word) in input. Anything higher will be truncated. Max 512.
- **max_output_length** – Maximum number of tokens (1 token ~ 1 word) in output. Anything higher will be truncated. Max 512.
- **validation_split** – Float between 0 and 1 that determines what percentage of the data to use for validation.
- **epochs** – Integer specifying how many training iterations to run.
- **batch_size** – Batch size when training. Leave as None to automatically determine batch size.
- **optimal_batch_size** – Optimal batch size for the model being trained – defaults to model settings.
- **early_stopping_epochs** – Integer determining how many epochs will run before stopping without an improvement in validation loss.
- **train_dataloader** – Dataloader for providing training data when finetuning. Defaults to inbuilt dataloder.

- **val_dataloader** – Dataloader for providing validation data when finetuning. Defaults to inbuilt dataloader.
- **step** – Function determining how to call model for a training step. Defaults to step defined in this task class.
- **configure_optimizers** – Function that sets up the optimizer for training. Defaults to optimizer defined in this task class.

Examples:

```
import backprop

emote = backprop.Emotion()

# Provide sentiment data for training
inp = ["I really liked the service I received!", "Meh, it was not impressive.
↪"]
out = ["positive", "negative"]
params = {"input_text": inp, "output_text": out}

# Finetune
emote.finetune(params)
```

static list_models(*return_dict=False*, *display=False*, *limit=None*)

Returns the list of models that can be used and finetuned with this task.

Parameters

- **return_dict** – Default False. True if you want to return in dict form. Otherwise returns list form.
- **display** – Default False. True if you want output printed directly (overrides return_dict, and returns nothing).
- **limit** – Default None. Maximum number of models to return – leave None to get all models.

step(*batch*, *batch_idx*)

Performs a training step and returns loss.

Parameters

- **batch** – Batch output from the dataloader
- **batch_idx** – Batch index.

training: bool

7.3 backprop.tasks.image_classification

class ImageClassification(*model: Optional[Union[str, backprop.models.generic_models.BaseModel]] = None*, *local: bool = False*, *api_key: Optional[str] = None*, *device: Optional[str] = None*)

Bases: *backprop.tasks.base.Task*

Task for image classification.

model

1. Model name

2. Model name on Backprop's image-classification endpoint
3. Model object that implements the image-classification task

local
Run locally. Defaults to False

Type optional

api_key
Backprop API key for non-local inference

Type optional

device
Device to run inference on. Defaults to "cuda" if available.

Type optional

__call__(*image: Union[str, List[str]]*, *labels: Optional[Union[List[str], List[List[str]]]] = None*, *top_k: int = 0*)
Classify image according to given labels.

Parameters

- **image** – image or list of images to vectorise. Can be both PIL Image objects or paths to images.
- **labels** – list of strings or list of labels (for zero shot classification)
- **top_k** – return probabilities only for top_k predictions. Use 0 to get all.

Returns dict where each key is a label and value is probability between 0 and 1 or list of dicts

configure_optimizers()
Returns default optimizer for image classification (SGD, learning rate 1e-1, weight decay 1e-4)

finetune(*params*, *validation_split: Union[float, Tuple[List[int], List[int]]] = 0.15*, *variant: str = 'single_label'*, *epochs: int = 20*, *batch_size: Optional[int] = None*, *optimal_batch_size: Optional[int] = None*, *early_stopping_epochs: int = 1*, *train_dataloader=None*, *val_dataloader=None*, *step=None*, *configure_optimizers=None*)
Finetunes a model for image classification.

Parameters

- **params** – Dictionary of model inputs. Contains 'images' and 'labels' keys, with values as lists of images/labels.
- **validation_split** – Float between 0 and 1 that determines what percentage of the data to use for validation.
- **variant** – Determines whether to do single or multi-label classification: "single_label" (default) or "multi_label"
- **epochs** – Integer specifying how many training iterations to run.
- **batch_size** – Batch size when training. Leave as None to automatically determine batch size.
- **optimal_batch_size** – Optimal batch size for the model being trained – defaults to model settings.
- **early_stopping_epochs** – Integer determining how many epochs will run before stopping without an improvement in validation loss.

- **train_dataloader** – Dataloader for providing training data when finetuning. Defaults to inbuilt dataloder.
- **val_dataloader** – Dataloader for providing validation data when finetuning. Defaults to inbuilt dataloader.
- **step** – Function determining how to call model for a training step. Defaults to step defined in this task class.
- **configure_optimizers** – Function that sets up the optimizer for training. Defaults to optimizer defined in this task class.

Examples:

```
import backprop

ic = backprop.ImageClassification()

# Prep training images/labels. Labels are automatically used to set up model␣
↪with number of classes for classification.
images = ["images/beagle/photo.jpg", "images/dachsund/photo.jpg", "images/
↪malamute/photo.jpg"]
labels = ["beagle", "dachsund", "malamute"]
params = {"images": images, "labels": labels}

# Finetune
ic.finetune(params, variant="single_label")
```

static list_models(*return_dict=False*, *display=False*, *limit=None*)

Returns the list of models that can be used and finetuned with this task.

Parameters

- **return_dict** – Default False. True if you want to return in dict form. Otherwise returns list form.
- **display** – Default False. True if you want output printed directly (overrides return_dict, and returns nothing).
- **limit** – Default None. Maximum number of models to return – leave None to get all models.

step_multi_label(*batch*, *batch_idx*)

Performs a training step for multi-label classification and returns loss.

Parameters

- **batch** – Batch output from the dataloader
- **batch_idx** – Batch index.

step_single_label(*batch*, *batch_idx*)

Performs a training step for single-label classification and returns loss.

Parameters

- **batch** – Batch output from the dataloader
- **batch_idx** – Batch index.

training: bool

7.4 backprop.tasks.image_text_vectorisation

class ImageTextVectorisation(*model: Optional[Union[str, backprop.models.generic_models.BaseModel]] = None*, *local: bool = False*, *api_key: Optional[str] = None*, *device: Optional[str] = None*)

Bases: *backprop.tasks.base.Task*

Task for combined image-text vectorisation.

model

1. Model name
2. Model name on Backprop's image-text-vectorisation endpoint
3. Model object that implements the image-text-vectorisation task

local
Run locally. Defaults to False

Type optional

api_key
Backprop API key for non-local inference

Type optional

device
Device to run inference on. Defaults to "cuda" if available.

Type optional

__call__(*image: Union[str, List[str]]*, *text: Union[str, List[str]]*, *return_tensor=False*)
Vectorise input image and text pairs.

Parameters

- **image** – image or list of images to vectorise. Can be both PIL Image objects or paths to images.
- **text** – text or list of text to vectorise. Must match image ordering.

Returns Vector or list of vectors

configure_optimizers()
Returns default optimizer for image-text vectorisation (AdamW, learning rate 1e-5)

finetune(*params*, *validation_split: Union[float, Tuple[List[int], List[int]]] = 0.15*, *variant: str = 'triplet'*, *epochs: int = 20*, *batch_size: Optional[int] = None*, *optimal_batch_size: Optional[int] = None*, *early_stopping_epochs: int = 1*, *train_dataloader=None*, *val_dataloader=None*, *step=None*, *configure_optimizers=None*)
Finetunes a model for combined image & text vectorisation. Includes different variants for calculating loss.

Parameters

- **params** – Dictionary of model inputs. If using triplet variant, contains keys "texts", "images", and "groups". If using cosine_similarity variant, contains keys "texts1", "texts2", "imgs1", "imgs2", and "similarity_scores".
- **validation_split** – Float between 0 and 1 that determines percentage of data to use for validation.
- **variant** – How loss will be calculated: "triplet" (default) or "cosine_similarity".

- **epochs** – Integer specifying how many training iterations to run.
- **batch_size** – Batch size when training. Leave as None to automatically determine batch size.
- **optimal_batch_size** – Optimal batch size for the model being trained – defaults to model settings.
- **early_stopping_epochs** – Integer determining how many epochs will run before stopping without an improvement in validation loss.
- **train_dataloader** – Dataloader for providing training data when finetuning. Defaults to inbuilt dataloder.
- **val_dataloader** – Dataloader for providing validation data when finetuning. Defaults to inbuilt dataloader.
- **step** – Function determining how to call model for a training step. Defaults to step defined in this task class.
- **configure_optimizers** – Function that sets up the optimizer for training. Defaults to optimizer defined in this task class.

Examples:

```
import backprop

itv = backprop.ImageTextVectorisation()

# Prep training data & finetune (triplet variant)
images = ["product_images/crowbars/photo.jpg", "product_images/crowbars/
↪photo1.jpg", "product_images/mugs/photo.jpg"]
texts = ["Steel crowbar with angled beak, 300mm", "Crowbar tempered steel␣
↪300m angled", "Sturdy ceramic mug, microwave-safe"]
groups = [0, 0, 1]
params = {"images": images, "texts": texts, "groups": groups}

itv.finetune(params, variant="triplet")

# Prep training data & finetune (cosine_similarity variant)
imgs1 = ["product_images/crowbars/photo.jpg", "product_images/mugs/photo.jpg"]
texts1 = ["Steel crowbar with angled beak, 300mm", "Sturdy ceramic mug,␣
↪microwave-safe"]
imgs2 = ["product_images/crowbars/photo1.jpg", "product_images/hats/photo.jpg]
texts2 = ["Crowbar tempered steel 300m angled", "Dad hat with funny ghost␣
↪picture on the front"]
similarity_scores = [1.0, 0.0]
params = {"imgs1": imgs1, "imgs2": imgs2, "texts1": texts1, "texts2": texts2,
↪"similarity_scores": similarity_scores}

itv.finetune(params, variant="cosine_similarity")
```

static list_models(*return_dict=False*, *display=False*, *limit=None*)

Returns the list of models that can be used and finetuned with this task.

Parameters

- **return_dict** – Default False. True if you want to return in dict form. Otherwise returns list form.
- **display** – Default False. True if you want output printed directly (overrides return_dict, and returns nothing).

- **limit** – Default None. Maximum number of models to return – leave None to get all models.

step_cosine(*batch*, *batch_idx*)
Performs a training step and calculates cosine similarity loss.

Parameters

- **batch** – Batch output from dataloader.
- **batch_idx** – Batch index.

step_triplet(*batch*, *batch_idx*)
Performs a training step and calculates triplet loss.

Parameters

- **batch** – Batch output from dataloader.
- **batch_idx** – Batch index.

train_dataloader_triplet()
Returns training dataloader with triplet loss sampling strategy.

training: bool

val_dataloader_triplet()
Returns validation dataloader with triplet loss sampling strategy.

7.5 backprop.tasks.image_vectorisation

class ImageVectorisation(*model: Optional[Union[str,* backprop.models.generic_models.BaseModel*]] = None*, *local: bool = False*, *api_key: Optional[str] = None*, *device: Optional[str] = None*)
Bases: *backprop.tasks.base.Task*

Task for image vectorisation.

model

1. Model name
2. Model name on Backprop's image-vectorisation endpoint
3. Model object that implements the image-vectorisation task

local
Run locally. Defaults to False

Type optional

api_key
Backprop API key for non-local inference

Type optional

device
Device to run inference on. Defaults to "cuda" if available.

Type optional

__call__(*image: Union[str, PIL.Image.Image, List[str], List[PIL.Image.Image]]*, *return_tensor=False*)
Vectorise input image.

Parameters **image** – image or list of images to vectorise. Can be both PIL Image objects or paths to images.

Returns Vector or list of vectors

configure_optimizers()
Returns default optimizer for image vectorisation (AdamW, learning rate 1e-5)

finetune(*params*, *validation_split: Union[float, Tuple[List[int], List[int]]] = 0.15*, *variant: str = 'triplet'*, *epochs: int = 20*, *batch_size: Optional[int] = None*, *optimal_batch_size: Optional[int] = None*, *early_stopping_epochs: int = 1*, *train_dataloader=None*, *val_dataloader=None*, *step=None*, *configure_optimizers=None*)
Finetunes a model for image vectorisation. Includes different variants for calculating loss.

Parameters

- **params** – Dictionary of model inputs. If using triplet variant, contains keys "images" and "groups". If using cosine_similarity variant, contains keys "imgs1", "imgs2", and "similarity_scores".
- **validation_split** – Float between 0 and 1 that determines percentage of data to use for validation.
- **variant** – How loss will be calculated: "triplet" (default) or "cosine_similarity".
- **epochs** – Integer specifying how many training iterations to run.
- **batch_size** – Batch size when training. Leave as None to automatically determine batch size.
- **optimal_batch_size** – Optimal batch size for the model being trained – defaults to model settings.
- **early_stopping_epochs** – Integer determining how many epochs will run before stopping without an improvement in validation loss.
- **train_dataloader** – Dataloader for providing training data when finetuning. Defaults to inbuilt dataloder.
- **val_dataloader** – Dataloader for providing validation data when finetuning. Defaults to inbuilt dataloader.
- **step** – Function determining how to call model for a training step. Defaults to step defined in this task class.
- **configure_optimizers** – Function that sets up the optimizer for training. Defaults to optimizer defined in this task class.

Examples:

```
import backprop

iv = backprop.ImageVectorisation()

# Set up training data & finetune (triplet variant)
images = ["images/beagle/photo.jpg",  "images/shiba_inu/photo.jpg", "images/
↪beagle/photo1.jpg", "images/malamute/photo.jpg"]
groups = [0, 1, 0, 2]
params = {"images": images, "groups": groups}

iv.finetune(params, variant="triplet")

# Set up training data & finetune (cosine_similarity variant)
```

(continues on next page)

(continued from previous page)

```
imgs1 = ["images/beagle/photo.jpg", "images/shiba_inu/photo.jpg"]
imgs2 = ["images/beagle/photo1.jpg", "images/malamute/photo.jpg"]
similarity_scores = [1.0, 0.0]
params = {"imgs1": imgs1, "imgs2": imgs2, "similarity_scores": similarity_
↪scores}

iv.finetune(params, variant="cosine_similarity")
```

static list_models(*return_dict=False*, *display=False*, *limit=None*)
Returns the list of models that can be used and finetuned with this task.

Parameters

- **return_dict** – Default False. True if you want to return in dict form. Otherwise returns list form.
- **display** – Default False. True if you want output printed directly (overrides return_dict, and returns nothing).
- **limit** – Default None. Maximum number of models to return – leave None to get all models.

step_cosine(*batch*, *batch_idx*)
Performs a training step and calculates cosine similarity loss.

Parameters

- **batch** – Batch output from dataloader.
- **batch_idx** – Batch index.

step_triplet(*batch*, *batch_idx*)
Performs a training step and calculates triplet loss.

Parameters

- **batch** – Batch output from dataloader.
- **batch_idx** – Batch index.

train_dataloader_triplet()
Returns training dataloader with triplet loss sampling strategy.

training: bool

val_dataloader_triplet()
Returns validation dataloader with triplet loss sampling strategy.

7.6 backprop.tasks.qa

class QA(*model: Optional[Union[str,* backprop.models.generic_models.BaseModel*]] = None*, *local: bool = False*, *api_key: Optional[str] = None*, *device: Optional[str] = None*)
Bases: *backprop.tasks.base.Task*

Task for Question Answering.

model

1. Model name
2. Model name on Backprop's qa endpoint

3. Model object that implements the qa task

local
Run locally. Defaults to False

Type optional

api_key
Backprop API key for non-local inference

Type optional

device
Device to run inference on. Defaults to "cuda" if available.

Type optional

__call__(*question: Union[str, List[str]]*, *context: Union[str, List[str]]*, *prev_qa: Union[List[Tuple[str, str]], List[List[Tuple[str, str]]]] = []*)
Perform QA, either on docstore or on provided context.

Parameters

- **question** – Question (string or list of strings) for qa model.
- **context** – Context (string or list of strings) to ask question from.
- **prev_qa** (*optional*) – List of previous question, answer tuples or list of prev_qa.

Returns Answer string or list of answer strings

configure_optimizers()
Returns default optimizer for Q&A (AdaFactor, learning rate 1e-3)

finetune(*params*, *validation_split: Union[float, Tuple[List[int], List[int]]] = 0.15*, *max_input_length: int = 256*, *max_output_length: int = 32*, *epochs: int = 20*, *batch_size: Optional[int] = None*, *optimal_batch_size: Optional[int] = None*, *early_stopping_epochs: int = 1*, *train_dataloader=None*, *val_dataloader=None*, *step=None*, *configure_optimizers=None*)
Finetunes a model for Q&A tasks.

Parameters

- **params** – dictionary of lists: 'questions', 'answers', 'contexts'. Optionally includes 'prev_qas': list of lists containing (q, a) tuples to prepend to context.
- **max_input_length** – Maximum number of tokens (1 token ~ 1 word) in input. Anything higher will be truncated. Max 512.
- **max_output_length** – Maximum number of tokens (1 token ~ 1 word) in output. Anything higher will be truncated. Max 512.
- **validation_split** – Float between 0 and 1 that determines what percentage of the data to use for validation.
- **epochs** – Integer specifying how many training iterations to run
- **batch_size** – Batch size when training. Leave as None to automatically determine batch size.
- **optimal_batch_size** – Optimal batch size for the model being trained – defaults to model settings.
- **early_stopping_epochs** – Integer determining how many epochs will run before stopping without an improvement in validation loss.

- **train_dataloader** – Dataloader for providing training data when finetuning. Defaults to inbuilt dataloder.
- **val_dataloader** – Dataloader for providing validation data when finetuning. Defaults to inbuilt dataloader.
- **step** – Function determining how to call model for a training step. Defaults to step defined in this task class.
- **configure_optimizers** – Function that sets up the optimizer for training. Defaults to optimizer defined in this task class.

Examples:

```
import backprop

# Initialise task
qa = backprop.QA()

# Set up training data for QA. Note that repeated contexts are needed, along␣
↪with empty prev_qas to match.
# Input must be completely 1:1, each question has an associated answer,␣
↪context, and prev_qa (if prev_qa is to be used).
questions = ["What's Backprop?", "What language is it in?", "When was the␣
↪Moog synthesizer invented?"]
answers = ["A library that trains models", "Python", "1964"]
contexts = ["Backprop is a Python library that makes training and using␣
↪models easier.",
            "Backprop is a Python library that makes training and using␣
↪models easier.",
            "Bob Moog was a physicist. He invented the Moog synthesizer in␣
↪1964."]

prev_qas = [[],
            [("What's Backprop?", "A library that trains models")],
            []]

params = {"questions": questions,
          "answers": answers,
          "contexts": contexts,
          "prev_qas": prev_qas}

# Finetune
qa.finetune(params=params)
```

static list_models (*return_dict=False*, *display=False*, *limit=None*)

Returns the list of models that can be used and finetuned with this task.

Parameters

- **return_dict** – Default False. True if you want to return in dict form. Otherwise returns list form.
- **display** – Default False. True if you want output printed directly (overrides return_dict, and returns nothing).
- **limit** – Default None. Maximum number of models to return – leave None to get all models.

step (*batch*, *batch_idx*)

Performs a training step and returns loss.

Parameters

- **batch** – Batch output from the dataloader
- **batch_idx** – Batch index.

training: bool

7.7 backprop.tasks.summarisation

class Summarisation(*model: Optional[Union[str, backprop.models.generic_models.BaseModel]] = None*, *local: bool = False*, *api_key: Optional[str] = None*, *device: Optional[str] = None*)

Bases: *backprop.tasks.base.Task*

Task for summarisation.

model

1. Model name
2. Model name on Backprop's summarisation endpoint
3. Model object that implements the summarisation task

local
Run locally. Defaults to False

Type optional

api_key
Backprop API key for non-local inference

Type optional

device
Device to run inference on. Defaults to "cuda" if available.

Type optional

__call__(*text: Union[str, List[str]]*)
Perform summarisation on input text.

Parameters **text** – string or list of strings to be summarised - keep each string below 500 words.

Returns Summary string or list of summary strings.

configure_optimizers()
Returns default optimizer for summarisation (AdaFactor, learning rate 1e-3)

finetune(*params*, *validation_split: Union[float, Tuple[List[int], List[int]]] = 0.15*, *max_input_length: int = 512*, *max_output_length: int = 128*, *epochs: int = 20*, *batch_size: Optional[int] = None*, *optimal_batch_size: Optional[int] = None*, *early_stopping_epochs: int = 1*, *train_dataloader=None*, *val_dataloader=None*, *step=None*, *configure_optimizers=None*)
Finetunes a generative model for summarisation.

Note: input_text and output_text in params must have matching ordering (item 1 of input must match item 1 of output)

Parameters

- **params** – Dictionary of model inputs. Contains 'input_text' and 'output_text' keys, with values as lists of input/output data.
- **max_input_length** – Maximum number of tokens (1 token ~ 1 word) in input. Anything higher will be truncated. Max 512.
- **max_output_length** – Maximum number of tokens (1 token ~ 1 word) in output. Anything higher will be truncated. Max 512.
- **validation_split** – Float between 0 and 1 that determines what percentage of the data to use for validation
- **epochs** – Integer specifying how many training iterations to run
- **batch_size** – Batch size when training. Leave as None to automatically determine batch size.
- **optimal_batch_size** – Optimal batch size for the model being trained – defaults to model settings.
- **early_stopping_epochs** – Integer determining how many epochs will run before stopping without an improvement in validation loss
- **train_dataloader** – Dataloader for providing training data when finetuning. Defaults to inbuilt dataloder.
- **val_dataloader** – Dataloader for providing validation data when finetuning. Defaults to inbuilt dataloader.
- **step** – Function determining how to call model for a training step. Defaults to step defined in this task class.
- **configure_optimizers** – Function that sets up the optimizer for training. Defaults to optimizer defined in this task class.

Examples:

```
import backprop

summary = backprop.Summarisation()

# Provide training data for task
inp = ["This is a long news article about recent political happenings.",
↪"This is an article about some recent scientific research."]
out = ["Short political summary.", "Short scientific summary."]
params = {"input_text": inp, "output_text": out}

# Finetune
summary.finetune(params)
```

static list_models(*return_dict=False*, *display=False*, *limit=None*)

Returns the list of models that can be used and finetuned with this task.

Parameters

- **return_dict** – Default False. True if you want to return in dict form. Otherwise returns list form.
- **display** – Default False. True if you want output printed directly (overrides return_dict, and returns nothing).
- **limit** – Default None. Maximum number of models to return – leave None to get all models.

step (*batch*, *batch_idx*)
Performs a training step and returns loss.

Parameters

- **batch** – Batch output from the dataloader
- **batch_idx** – Batch index.

training: bool

7.8 backprop.tasks.text_classification

class TextClassification (*model: Optional[Union[str, backprop.models.generic_models.BaseModel]] = None*, *local: bool = False*, *api_key: Optional[str] = None*, *device: Optional[str] = None*)
Bases: *backprop.tasks.base.Task*

Task for classification.

model

1. Model name
2. Model name on Backprop's text-classification endpoint
3. Model object that implements the text-classification task

local
Run locally. Defaults to False

Type optional

api_key
Backprop API key for non-local inference

Type optional

device
Device to run inference on. Defaults to "cuda" if available.

Type optional

__call__ (*text: Union[str, List[str]]*, *labels: Optional[Union[List[str], List[List[str]]]] = None*, *top_k: int = 0*)
Classify input text based on previous training (user-tuned models) or according to given list of labels (zero-shot)

Parameters

- **text** – string or list of strings to be classified
- **labels** – list of labels for zero-shot classification (on our out-of-the-box models). If using a user-trained model (e.g. XLNet), this is not used.
- **top_k** – return probabilities only for top_k predictions. Use 0 to get all.

Returns dict where each key is a label and value is probability between 0 and 1, or list of dicts.

configure_optimizers ()
Returns default optimizer for text classification (AdamW, learning rate 2e-5)

finetune(*params*, *validation_split: Union[float, Tuple[List[int], List[int]]] = 0.15*, *max_length: int = 128*, *epochs: int = 20*, *batch_size: Optional[int] = None*, *optimal_batch_size: Optional[int] = None*, *early_stopping_epochs: int = 1*, *train_dataloader=None*, *val_dataloader=None*, *step=None*, *configure_optimizers=None*)

Finetunes a text classification model on provided data.

Parameters

- **params** – Dict containing keys "texts" and "labels", with values being input/output data lists.
- **validation_split** – Float between 0 and 1 that determines percentage of data to use for validation.
- **max_length** – Int determining the maximum token length of input strings.
- **epochs** – Integer specifying how many training iterations to run.
- **batch_size** – Batch size when training. Leave as None to automatically determine batch size.
- **optimal_batch_size** – Optimal batch size for the model being trained – defaults to model settings.
- **early_stopping_epochs** – Integer determining how many epochs will run before stopping without an improvement in validation loss.
- **train_dataloader** – Dataloader for providing training data when finetuning. Defaults to inbuilt dataloder.
- **val_dataloader** – Dataloader for providing validation data when finetuning. Defaults to inbuilt dataloader.
- **step** – Function determining how to call model for a training step. Defaults to step defined in this task class.
- **configure_optimizers** – Function that sets up the optimizer for training. Defaults to optimizer defined in this task class.

Examples:

```
import backprop

tc = backprop.TextCLassification()

# Set up input data. Labels will automatically be used to set up model with␣
↪number of classes for classification.
inp = ["This is a political news article", "This is a computer science␣
↪research paper", "This is a movie review"]
out = ["Politics", "Science", "Entertainment"]
params = {"texts": inp, "labels": out}

# Finetune
tc.finetune(params)
```

static list_models(*return_dict=False*, *display=False*, *limit=None*)

Returns the list of models that can be used and finetuned with this task.

Parameters

- **return_dict** – Default False. True if you want to return in dict form. Otherwise returns list form.

- **display** – Default False. True if you want output printed directly (overrides return_dict, and returns nothing).
- **limit** – Default None. Maximum number of models to return – leave None to get all models.

step(*batch*, *batch_idx*)
Performs a training step and returns loss.

Parameters

- **batch** – Batch output from the dataloader
- **batch_idx** – Batch index.

training: bool

7.9 backprop.tasks.text_generation

class TextGeneration(*model: Optional[Union[str, backprop.models.generic_models.BaseModel]] = None*, *local: bool = False*, *api_key: Optional[str] = None*, *device: Optional[str] = None*)
Bases: *backprop.tasks.base.Task*

Task for text generation.

model

1. Model name
2. Model name on Backprop's text-generation endpoint
3. Model object that implements the text-generation task

local
Run locally. Defaults to False

Type optional

api_key
Backprop API key for non-local inference

Type optional

device
Device to run inference on. Defaults to "cuda" if available.

Type optional

__call__(*text: Union[str, List[str]]*, *min_length: Optional[int] = None*, *max_length: Optional[int] = None*, *temperature: Optional[float] = None*, *top_k: Optional[int] = None*, *top_p: Optional[float] = None*, *repetition_penalty: Optional[float] = None*, *length_penalty: Optional[float] = None*, *num_beams: Optional[int] = None*, *num_generations: Optional[int] = None*, *do_sample: Optional[bool] = None*)
Generates text to continue from the given input.

Parameters

- **input_text** (*string*) – Text from which the model will begin generating.
- **min_length** (*int*) – Minimum number of tokens to generate (1 token ~ 1 word).
- **max_length** (*int*) – Maximum number of tokens to generate (1 token ~ 1 word).

- **temperature** (*float*) – Value that alters the randomness of generation (0.0 is no randomness, higher values introduce randomness. 0.5 - 0.7 is a good starting point).
- **top_k** (*int*) – Only choose from the top_k tokens when generating (0 is no limit).
- **top_p** (*float*) – Only choose from the top tokens with combined probability greater than top_p.
- **repetition_penalty** (*float*) – Penalty to be applied to tokens present in the input_text and tokens already generated in the sequence (>1 discourages repetition while <1 encourages).
- **length_penalty** (*float*) – Penalty applied to overall sequence length. Set >1 for longer sequences, or <1 for shorter ones.
- **num_beams** (*int*) – Number of beams to be used in beam search. Does a number of generations to pick the best one. (1: no beam search)
- **num_generations** (*int*) – How many times to run generation. Results are returned as a list.
- **do_sample** (*bool*) – Whether or not sampling strategies (temperature, top_k, top_p) should be used.

Example:

```
import backprop

tg = backprop.TextGeneration()
tg("Geralt knew the sings, the monster was a", min_length=20, max_length=50,␣
↪temperature=0.7)
> " real danger, and he was the only one in the village who knew how to␣
↪defend himself."
```

configure_optimizers()
Returns default optimizer for text generation (AdaFactor, learning rate 1e-3)

finetune(*params*, *validation_split: Union[float, Tuple[List[int], List[int]]] = 0.15*, *max_input_length: int = 128*, *max_output_length: int = 32*, *epochs: int = 20*, *batch_size: Optional[int] = None*, *optimal_batch_size: Optional[int] = None*, *early_stopping_epochs: int = 1*, *train_dataloader=None*, *val_dataloader=None*, *step=None*, *configure_optimizers=None*)
Finetunes a model for a text generation task.

Note: input_text and output_text in params must have matching ordering (item 1 of input must match item 1 of output)

Parameters

- **params** – Dictionary of model inputs. Contains 'input_text' and 'output_text' keys, with values as lists of input/output data.
- **max_input_length** – Maximum number of tokens (1 token ~ 1 word) in input. Anything higher will be truncated. Max 512.
- **max_output_length** – Maximum number of tokens (1 token ~ 1 word) in output. Anything higher will be truncated. Max 512.
- **validation_split** – Float between 0 and 1 that determines what percentage of the data to use for validation.

- **epochs** – Integer specifying how many training iterations to run.
- **batch_size** – Batch size when training. Leave as None to automatically determine batch size.
- **optimal_batch_size** – Optimal batch size for the model being trained – defaults to model settings.
- **early_stopping_epochs** – Integer determining how many epochs will run before stopping without an improvement in validation loss.
- **train_dataloader** – Dataloader for providing training data when finetuning. Defaults to inbuilt dataloder.
- **val_dataloader** – Dataloader for providing validation data when finetuning. Defaults to inbuilt dataloader.
- **step** – Function determining how to call model for a training step. Defaults to step defined in this task class.
- **configure_optimizers** – Function that sets up the optimizer for training. Defaults to optimizer defined in this task class.

Examples:

```
import backprop

tg = backprop.TextGeneration()

# Any text works as training data
inp = ["I really liked the service I received!", "Meh, it was not impressive.
↪"]
out = ["positive", "negative"]
params = {"input_text": inp, "output_text": out}

# Finetune
tg.finetune(params)
```

static list_models(*return_dict=False*, *display=False*, *limit=None*)

Returns the list of models that can be used and finetuned with this task.

Parameters

- **return_dict** – Default False. True if you want to return in dict form. Otherwise returns list form.
- **display** – Default False. True if you want output printed directly (overrides return_dict, and returns nothing).
- **limit** – Default None. Maximum number of models to return – leave None to get all models.

step(*batch*, *batch_idx*)

Performs a training step and returns loss.

Parameters

- **batch** – Batch output from the dataloader
- **batch_idx** – Batch index.

training: bool

7.10 backprop.tasks.text_vectorisation

class TextVectorisation(*model: Optional[Union[str, backprop.models.generic_models.BaseModel]] = None*, *local: bool = False*, *api_key: Optional[str] = None*, *device: Optional[str] = None*)

Bases: *backprop.tasks.base.Task*

Task for text vectorisation.

model

1. Model name
2. Model name on Backprop's text-vectorisation endpoint
3. Model object that implements the text-vectorisation task

local
Run locally. Defaults to False

Type optional

api_key
Backprop API key for non-local inference

Type optional

device
Device to run inference on. Defaults to "cuda" if available.

Type optional

__call__(*text: Union[str, List[str]]*, *return_tensor=False*)
Vectorise input text.

Parameters **text** – string or list of strings to vectorise. Can be both PIL Image objects or paths to images.

Returns Vector or list of vectors

configure_optimizers()
Returns default optimizer for text vectorisation (AdamW, learning rate 1e-5)

finetune(*params*, *validation_split: Union[float, Tuple[List[int], List[int]]] = 0.15*, *max_length: Optional[int] = None*, *variant: str = 'cosine_similarity'*, *epochs: int = 20*, *batch_size: Optional[int] = None*, *optimal_batch_size: Optional[int] = None*, *early_stopping_epochs: int = 1*, *train_dataloader=None*, *val_dataloader=None*, *step=None*, *configure_optimizers=None*)
Finetunes a model for text vectorisation. Includes different variants for calculating loss.

Parameters

- **params** – Dictionary of model inputs. If using triplet variant, contains keys "texts" and "groups". If using cosine_similarity variant, contains keys "texts1", "texts2", and "similarity_scores".
- **validation_split** – Float between 0 and 1 that determines percentage of data to use for validation.
- **max_length** – Int determining the maximum token length of input strings.
- **variant** – How loss will be calculated: "cosine_similarity" (default) or "triplet".
- **epochs** – Integer specifying how many training iterations to run.

- **batch_size** – Batch size when training. Leave as None to automatically determine batch size.
- **optimal_batch_size** – Optimal batch size for the model being trained – defaults to model settings.
- **early_stopping_epochs** – Integer determining how many epochs will run before stopping without an improvement in validation loss.
- **train_dataloader** – Dataloader for providing training data when finetuning. Defaults to inbuilt dataloder.
- **val_dataloader** – Dataloader for providing validation data when finetuning. Defaults to inbuilt dataloader.
- **step** – Function determining how to call model for a training step. Defaults to step defined in this task class.
- **configure_optimizers** – Function that sets up the optimizer for training. Defaults to optimizer defined in this task class.

Examples:

```
import backprop

tv = backprop.TextVectorisation()

# Set up training data & finetune (cosine_similarity variant)
texts1 = ["I went to the store and bought some bread", "I am getting a cat␣
↪soon"]
texts2 = ["I bought bread from the store", "I took my dog for a walk"]
similarity_scores = [1.0, 0.0]
params = {"texts1": texts1, "texts2": texts2, "similarity_scores": similarity_
↪scores}

tv.finetune(params, variant="cosine_similarity")

# Set up training data & finetune (triplet variant)
texts = ["I went to the store and bought some bread", "I bought bread from␣
↪the store", "I'm going to go walk my dog"]
groups = [0, 0, 1]
params = {"texts": texts, "groups": groups}

tv.finetune(params, variant="triplet")
```

static list_models(*return_dict=False*, *display=False*, *limit=None*)

Returns the list of models that can be used and finetuned with this task.

Parameters

- **return_dict** – Default False. True if you want to return in dict form. Otherwise returns list form.
- **display** – Default False. True if you want output printed directly (overrides return_dict, and returns nothing).
- **limit** – Default None. Maximum number of models to return – leave None to get all models.

step_cosine(*batch*, *batch_idx*)

Performs a training step and calculates cosine similarity loss.

Parameters

- **batch** – Batch output from dataloader.
- **batch_idx** – Batch index.

step_triplet(*batch*, *batch_idx*)
Performs a training step and calculates triplet loss.

Parameters

- **batch** – Batch output from dataloader.
- **batch_idx** – Batch index.

train_dataloader_triplet()
Returns training dataloader with triplet loss sampling strategy.

training: bool

val_dataloader_triplet()
Returns validation dataloader with triplet loss sampling strategy.

CHAPTER

EIGHT

BACKPROP.UTILS

8.1 Subpackages

8.1.1 backprop.utils.losses

backprop.utils.losses.triplet_loss

class TripletLoss(*device*)

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

forward(*input*, *target*, ***kwargs*)

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.

training: bool

TripletSemiHardLoss(*y_true*, *y_pred*, *device*, *margin=1.0*)

Computes the triplet loss_functions with semi-hard negative mining. The loss_functions encourages the positive distances (between a pair of embeddings with the same labels) to be smaller than the minimum negative distance among which are at least greater than the positive distance plus the margin constant (called semi-hard negative) in the mini-batch. If no such negative exists, uses the largest negative distance instead. See: https://arxiv.org/abs/1503.03832. We expect labels *y_true* to be provided as 1-D integer *Tensor* with shape [batch_size] of multi-class integer labels. And embeddings *y_pred* must be 2-D float *Tensor* of l2 normalized embedding vectors. :param margin: Float, margin term in the loss_functions definition. Default value is 1.0. :param name: Optional name for the op.

pairwise_distance_torch(*embeddings*, *device*)

Computes the pairwise distance matrix with numerical stability. output[i, j] = || feature[i, :] - feature[j, :] ||_2 :param embeddings: 2-D Tensor of size [number of data, feature dimension].

Returns 2-D Tensor of size [number of data, number of data].

Return type pairwise_distances

8.2 backprop.utils.datasets

class ImageGroupDataset(*images*, *groups*, *process_batch*)
Bases: Generic[torch.utils.data.dataset.T_co]

class ImagePairDataset(*imgs1*, *imgs2*, *similarity_scores*, *process_batch*)
Bases: Generic[torch.utils.data.dataset.T_co]

class ImageTextGroupDataset(*images*, *texts*, *groups*, *process_batch*)
Bases: Generic[torch.utils.data.dataset.T_co]

class ImageTextPairDataset(*img_text_pairs1*, *img_text_pairs2*, *similarity_scores*, *process_batch*)
Bases: Generic[torch.utils.data.dataset.T_co]

class MultiLabelImageClassificationDataset(*images*, *labels*, *process_batch*)
Bases: Generic[torch.utils.data.dataset.T_co]

class SingleLabelImageClassificationDataset(*images*, *labels*, *process_batch*)
Bases: Generic[torch.utils.data.dataset.T_co]

class SingleLabelTextClassificationDataset(*params*, *process_batch*, *length*)
Bases: Generic[torch.utils.data.dataset.T_co]

class TextGroupDataset(*texts*, *groups*, *process_batch*, *max_length=None*)
Bases: Generic[torch.utils.data.dataset.T_co]

class TextPairDataset(*texts1*, *texts2*, *similarity_scores*, *process_batch*, *max_length=None*)
Bases: Generic[torch.utils.data.dataset.T_co]

class TextToTextDataset(*params*, *task*, *process_batch*, *length*)
Bases: Generic[torch.utils.data.dataset.T_co]

8.3 backprop.utils.download

download(*url: str*, *folder: str*, *root: str = '/home/docs/.cache/backprop'*, *force: bool = False*)
Downloads file from url to folder

8.4 backprop.utils.functions

cosine_similarity(*vec1: Union[List[float], torch.Tensor]*, *vec2: Union[List[float], torch.Tensor, List[List[float]], List[torch.Tensor]]*)
Calculates cosine similarity between two vectors.

Parameters

- **vec1** – list of floats or corresponding tensor
- **vec2** – list of floats / list of list of floats or corresponding tensor

Example:

```
import backprop

backprop.cosine_similarity(vec1, vec2)
0.8982
```

(continues on next page)

(continued from previous page)

```
backprop.cosine_similarity(vec1, [vec2, vec3])
[0.8982, 0.3421]
```

8.5 backprop.utils.helpers

base64_to_img (*image: Union[str, List[str]]*)
Returns PIL Image objects of base64 encoded images

img_to_base64 (*image: Union[PIL.Image.Image, List[PIL.Image.Image]]*)
Returns base64 encoded strings of PIL Image objects

path_to_img (*image: Union[str, List[str]]*)
Returns PIL Image objects of paths to images

8.6 backprop.utils.load

load (*path*)
Loads a saved model and returns it.

Parameters **path** – Name of the model or full path to model.

Example:

```
import backprop

backprop.save(model_object, "my_model")
model = backprop.load("my_model")
```

8.7 backprop.utils.samplers

class SameGroupSampler (*dataset*)
Bases: Generic[torch.utils.data.sampler.T_co]

8.8 backprop.utils.save

save (*model*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *path=None*)

Saves the provided model to the backprop cache folder using:

1. provided name
2. model.name
3. provided path

The resulting folder has three files:

- model.bin (dill pickled model instance)
- config.json (description and task keys)

- requirements.txt (exact python runtime requirements)

Parameters

- **model** – Model object
- **name** – string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.
- **description** – String description of the model.
- **tasks** – List of supported task strings
- **details** – Valid json dictionary of additional details about the model
- **path** – Optional path to save model

Example:

```
import backprop

backprop.save(model_object, "my_model")
model = backprop.load("my_model")
```

8.9 backprop.utils.upload

upload(*model*, *name: Optional[str] = None*, *description: Optional[str] = None*, *tasks: Optional[List[str]] = None*, *details: Optional[Dict] = None*, *path=None*, *api_key: Optional[str] = None*)
Saves and deploys a model to Backprop.

Parameters

- **model** – Model object
- **api_key** – Backprop API key
- **name** – string identifier for the model. Lowercase letters and numbers. No spaces/special characters except dashes.
- **description** – String description of the model.
- **tasks** – List of supported task strings
- **details** – Valid json dictionary of additional details about the model
- **path** – Optional path to save model

Example:

```
import backprop

tg = backprop.TextGeneration("t5_small")

# Any text works as training data
inp = ["I really liked the service I received!", "Meh, it was not impressive."]
out = ["positive", "negative"]

# Finetune with a single line of code
tg.finetune({"input_text": inp, "output_text": out})
```

(continues on next page)

(continued from previous page)

```
# Use your trained model
prediction = tg("I enjoyed it!")

print(prediction)
# Prints
"positive"

# Upload to Backprop for production ready inference

model = tg.model
# Describe your model
name = "t5-sentiment"
description = "Predicts positive and negative sentiment"

backprop.upload(model, name=name, description=description, api_key="abc")
```

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W