Uncertainty Wizard

Release 0.3.0

Michael Weiss

May 03, 2022

DOCUMENTATION

Uncertainty wizard is a plugin on top of `tensorflow.keras`, allowing to easily and efficiently create uncertainty-aware deep neural networks:

- **Plain Keras Syntax:** Use the layers and APIs you know and love.
- **Conversion from keras:** Convert existing keras models into uncertainty aware models.
- **Smart Randomness:** Use the same model for point predictions and sampling based inference.
- **Fast ensembles:** Train and evaluate deep ensembles lazily loaded and using parallel processing.
- **Super easy setup:** Pip installable. Only tensorflow as dependency.

CHAPTER

ONE

INSTALLATION

The installation is as simple as

```
pip install uncertainty-wizard
```

Then, in any file where you want to use uncertainty wizard, add the following import statement:

```
import uncertainty_wizard as uwiz
```

1.1 Dependencies

We acknowledge that a slim dependency tree is critical to many practical projects. Thus, the only dependency of uncertainty wizard is `tensorflow>=2.3.0`.

Note however that if you are still using python 3.6, you have to install the backport *dataclasses*.

Note: Uncertainty Wizard is tested with tensorflow 2.3.0 and tensorflow is evolving quickly. Please do not hesitate to report an issue if you find broken functionality in more recent tensorflow versions.

CHAPTER

TWO

USER GUIDE: MODELS

Uncertainty wizard supports three types of models capable to calculate uncertainties and confidences:

- *Stochastic Models (e.g. MC-Dropout)*
  `Stochastic models` use randomness during predictions (typically using dropout layers). Then, for a given input, multiple predictions are made (-> sampling). The final predictions und uncertainties are a function of the distribution of neural network outputs distributions over all the observed samples.
- *Ensemble Models*
  `Ensemble models` are collections of models trained to answer the same problem, but with slightly different weights due to different kernel initialization or slightly different training data. Then, for a given input, a prediction is made on each model in the collection (-> sampling). The final predictions und uncertainties are a function of the distribution of neural network outputs over all the models in the ensemble.
- *Point Predictor Models* [for classification problems only]
  We call models which base their prediction and uncertainty quantification based on a single inference on a single, traditional (i.e., non-bayesian) neural network a `Point-Predictor models`. Point predictor based uncertainty quantification can typically only be applied to classification problems, where the uncertainty quantification is based on the distribution of values in the softmax output layer.

See our papers and the references therein for a more detailed for a detailed information about the here described techniques.

2.1 Stochastic Models (e.g. MC-Dropout)

Stochastic models are models in which some randomness is added to the network during training. While this is typically done for network regularization, models trained in such a way can be used for uncertainty quantification. Simply speaking:

Randomness (which is typically disabled during inference) can be enforced during inference, leading to predictions which are impacted by the random noise. By sampling multiple network outputs for the same input, we can infer the robustness of the network to the random noise. We assume that the higher the robustness, the higher the networks confidence.

Uwiz stochastic models wrap a keras model, and inject a mechanism to automatically control the randomization during inference.

TL;DR? Get started with two short snippets

Listing 1: Stochastic API: The simplest way to stochastic models

```
model = uwiz.models.StochasticSequential()
```

(continues on next page)

(continued from previous page)

```
# The following lines are equivalent to a keras Sequential model
model.add(tf.keras.layers.Dense(100)
# Dropout and noise layers will be used to randomize predictions
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Softmax(10))
model.compile(..)
model.fit(..)

# Make predictions, and calculate the variation ratio as uncertainty metric
# (where x_test are the inputs for which you want to predict...)
pred, unc = model.predict_quantified(x_test, quantifier='var_ratio', num_samples=32)
```

Listing 2: Functional API: Full control over randomness

```
model = uwiz.models.StochasticFunctional()

# We create an object that will serve as a flag during inference
# indicating on whether randomization should be enabled
stochastic_mode = uwiz.models.stochastic.StochasticMode()

# We construct input and output as for classical tensorflow models
# Note that layers for prediction randomization have to be specified explicitly
input_l = tf.keras.layers.Input(100)
x = tf.keras.layers.Dense(100)(input_l)
x = uwiz.models.stochastic.layers.UwizBernoulliDropout(0.3, stochastic_mode=stochastic_
↪mode)(x)
output_l = tf.keras.layers.Softmax(10)(x)
model = uwiz.models.FunctionalStochastic(input_l, output_l, stochastic_mode=stochastic_
↪mode)
model.compile(..)
model.fit(..)

# Make predictions and calculate uncertainty (as shown in example above)
pred, unc = model.predict_quantified(x_test, quantifier='var_ratio', num_samples=32)
```

2.2 Ensemble Models

LazyEnsembles are uncertainty wizards implementation of Deep Ensembles, where multiple atomic models are trained for the same problem; the output distribution (and thus uncertainty) is then inferred from predicting on all atomic models.

Multi-Processing

This ensemble implementation is lazy as it does not keep the atomic models in memory (or even worse, in the tf graph). Instead, atomic models are persisted on the file system and only loaded when needed - and cleared from memory immediately afterwards. To further increase performance, in particular on high performance GPU powered hardware setups, where a single model instance training does not use the full GPU resources, LazyEnsemble allows to create multiple concurrent tensorflow sessions, each running a dedicated model in parallel. The number of processes to be used can be specified on essentially all of LazyEnsembles methods.

Models are loaded into a context, e.g. a gpu configuration which was configured before the model was loaded. The default context, if multiple processes are used, sets the GPU usage to dynamic memory growth. We recommend to set

the number of processes conservatively, observe the system load and increase the number of processes if possible.

If you use tensorflow in your main process, chances are the main thread allocates all available GPU resources. In such case you may for example want to enabling dynamic growth on the main thread, which can be done by calling the following utility method right after first importing tensorflow: `uwiz.models.ensemble_utils.DynamicGpuGrowthContextManager.enable_dynamic_gpu_growth()`

Warning: By using too many processes you will quickly exhaust your systems resources. Similarly, if you do not have a GPU: Your CPU will not be able to handle the high workload of training multiple models in parallel.

Multi-Processing can be disabled by setting the number of processes to 0. Then, predictions will be made in the main process on the main tensorflow session. *Attention*: In this case, the tensorflow session will be cleared after every model execution!

The LazyEnsemble Interface & Workflow

LazyEnsemble exposes five central functions: `create` `modify` `consume` `quantify_predictions` `run_model_free` create, modify, consume, quantify_predictions or run_model_free. In general, every of these functions expects a picklable function as input which either creates, modifies or consumes a plain keras model, or uses it to make predictions. Please refer to the specific methods documentation and examples for details.

Furthermore LazyEnsemble exposes utility methods wrapping the above listed methods, e.g. fit and predict_quantified, which expect numpy array inputs and automatically serialize and deserialize them to be used in parallel processes.

Note: The less often you call methods on your ensemble, the less often we have to deserialize and persist your models (which is some overhead). Thus, try reducing these calls for even faster processing: For example, you may want to fit your model as part of the `ensemble.create` call.

Stability of Lazy Ensembles

To optimize GPU use, LazyEnsemble relies on some of tensorflows features which are (as of August 2020) still experimental. Thus, by extension, our ensembles are also to be considered experimental.

TL;DR? Get started with one short snippet

Listing 3: Stochastic API: The simplest way to ensemble models

```
# Define how models should be trained. This function must be picklable.
def model_creator(model_id: int):
   import tensorflow as tf
   model = tf.keras.models.Sequential()
   model.add(tf.keras.layers.Dense(100)
   model.add(tf.keras.layers.Dropout(0.3))
   model.add(tf.keras.layers.Softmax(10))
   model.compile(..)
   fit_history = model.fit(..)
   return model, fit_history.history

# Define properties of the ensemble to be created
uwiz.models.LazyEnsemble(num_models=2,
                         model_save_path="/tmp/demo_ensemble",
                         default_num_processes=5)

# Create and train the inner models in your ensemble according to your process defined␣
↪above
```

(continues on next page)

(continued from previous page)

```
ensemble.create(create_function=model_creator)

# Now we are ready to make predictions
pred, unc = model.predict_quantified(x_test,
                                     quantifier='var_ratio',
                                     # For the sake of this example, lets assume we want␣
↪to
                                     # predict with a higher batch size and lower␣
↪process number
                                     # than our default settings.
                                     batch_size=128,
                                     num_processes=2)
```

2.3 Point Predictor Models

We call models which base their prediction and uncertainty quantification based on a single inference on a single, traditional (i.e., non-bayesian) neural network a Point-Predictor model. In *uncertainty wizard*, we can use the stochastic model classes `StochasticSequential` and `StochasticFunctional` for such predictions as well. To do so, create or re-use a stochastic model as explained above. Of course, if we only want to do point predictions, the stochastic model does not have to contain any stochastic layers (i.e., it can be deterministic). Stochastic layers (e.g. Dropout) which are included in the network are automatically disabled when doing point predictions.

The following snippet provides three examples on how to do point predictions on a stochastic model instance *model*:

Listing 4: Using the stochastic model classes for (non-sampled) point predictions

```
# Example 1: Plain Keras Prediction
# If we just want to use the keras model output (as it there were no uncertainty_wizard)
#   we can predict on the stochastic model as if it was a regular `tf.keras.Model`
nn_outputs = model.predict(x_test)


# Example 2: Point Prediction Confidence and Uncertainty Metrics
# We can also get confidences and uncertainties using predict_quantified.
#   For point-predictor quantifiers which don't rely on random sampling,
#   such as the prediction confidence score (PCS), randomness is automatically disabled
#   and the returned values are based on a one-shot prediction.
pred, unc = model.predict_quantified(x_test, quantifier='pcs')


# Example 2b: Doing Point-Prediction and Sampling Based Interence in one Call
# We can even combine point-prediction based and sampling based quantifiers
#   Randomization will only be used for the sampling based quantifiers
res = model.predict_quantified(x_test, quantifier=['pcs', 'var_ratio'])
#   If `quantifier` is a list, the returned res is also a list,
#   containing a (prediction, uncertainty_or_confidence_score) tuple
#   for every passed quantifier
point_predictor_predictions = res[0][0]
point_predictor_confidence_scores = res[0][1]
```

(continues on next page)

(continued from previous page)

```
sampling_based_predictions = res[1][0]
sampling_based_var_ratio = res[1][1]
```

CHAPTER

THREE

USER GUIDE: QUANTIFIERS

Quantifiers are dependencies, injectable into prediction calls, which calculate predictions and uncertainties or confidences from DNN outputs:

Listing 1: Use of quantifiers on uwiz models

```
# Let's use a quantifier that calculates the entropy on a regression variable as␣
↪uncertainty
predictions, entropy = model.predict_quantified(x_test, quantifier='predictive_entropy')

# Equivalently, we can pass the quantifier as object
quantifier = uwiz.quantifiers.PredictiveEntropy()
predictions, entropy = model.predict_quantified(x_test, quantifier=quantifer)

# We can also pass multiple quantifiers.
# In that case, `predict_quantified` returns a (prediction, confidence_or_uncertainty)␣
↪tuple
# for every passed quantifier.
results = model.predict_quantified(x_test, quantifier=['predictive_entropy', 'mean_
↪softmax'])
# results[0] is a tuple of predictions and entropies
# results[1] is a tuple of predictions and mean softmax values
```

Besides the prediction, quantifiers quantify either the networks confidence or its uncertainty. The difference between that two is as follows (assuming that the quantifier actually correctly captures the chance of misprediction):

- In *uncertainty quantification*, the higher the value, the higher the chance of misprediction.
- In *confidence quantification* the lower the value, the higher the chance of misprediction.

For most applications where you use multiple quantifiers, you probably want to quantify either uncertainties or confidences to allow to use the quantifiers outputs interchangeable. Setting the param `model.predict_quantified(..., as_confidence=True)` convert uncertainties into confidences. `as_confidence=False` converts confidences into uncertainties. The default is 'None', in which case no conversions are made.

Note: Independent on how many quantifiers you pass to the *predict_quantified* method, the outputs of the neural networks inference are re-used wherever possible for a more efficient execution. Thus, it is better to call *predict_quantified* with two quantifiers than to call *predict_quantified* twice, with one quantifier each.

3.1 Quantifiers implemented in Uncertainty Wizard

This Section provides an overview of the quantifiers provided in uncertainty wizard: For a precise discussion of the quantifiers listed here, please consult our paper and the docstrings of the quantifiers.

3.1.1 Point Prediction Quantifiers

Class (uwiz.quantifiers.<...>)	Problem Type	Aliases (besides class name)
MaxSoftmax	Classification	SM, softmax, max_softmax,
PredictionConfidenceScore	Classification	PCS, prediction_confidence_score
SoftmaxEntropy	Classification	SE, softmax_entropy

3.1.2 Monte Carlo Sampling Quantifiers

Class (uwiz.quantifiers.<...>)	Problem Type	Aliases (besides class name)
VariationRatio	Classification	VR, var_ratio, variation_ratio
PredictiveEntropy	Classification	PE, pred_entropy, predictive_entropy
MutualInformation	Classification	MI, mutu_info, mutual_information
MeanSoftmax	Classification	MS, mean_softmax, ensembling
StandardDeviation	Regression	STD, stddev, std_dev, standard_deviation

3.2 Custom Quantifers

You can of course also use custom quantifiers with uncertainty wizard. It's as easy as extending `uwiz.quantifiers.Quantifier` and implement all abstract methods according to the description in the superclass method docstrings.

Let's for example assume you want to create an **identity function** quantifier for a sampling based DNN (i.e., a stochastic DNN or a deep ensemble) for a classification problem, which does not actually calculate a prediction and uncertainty, but just returns the observed DNN outputs. This can be achieved using the following snippet:

Listing 2: Custom quantifier definition: Identity Quantifier

```
class IdentityQuantifer(uwiz.quantifiers.Quantifier):
    @classmethod
    def aliases(cls) -> List[str]:
        return ["custom::identity"]

    @classmethod
    def takes_samples(cls) -> bool:
        return True
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def is_confidence(cls) -> bool:
        # Does not matter for the identity function
        return False

    @classmethod
    def calculate(cls, nn_outputs: np.ndarray):
        # Return None as prediction and all DNN outputs as 'quantification'
        return None, nn_outputs

    @classmethod
    def problem_type(cls) -> uwiz.ProblemType:
        return uwiz.ProblemType.CLASSIFICATION
```

If you want to call your custom quantifier by its alias, you need to add it to the registry. To prevent name clashes in future uncertainty wizard versions, where more quantifiers might be registered by default, we recommend you to preprend "custom::" to any of your quantifiers aliases.

Listing 3: Register a quantifier in the quantifier registry

```
custom_instance = IdentityQuantifier()
uwiz.quantifiers.QuantifierRegistry().register(custom_instance)

model = # (...) uwiz model creation, compiling and fitting
x_test = # (...) get the data for your predictions

# Now this call, where we calculate the variation ratio,
# and also return the observed DNN outputs...
results = model.predict_quantified(x_test, num_samples=20,
                                   quantifier=["var_ratio", "custom::identity"])
# ... is equivalent to this call...
results = model.predict_quantified(x_test, num_samples=20,
                                   quantifier=["var_ratio", IdentityQuantifier()])
```

Warning: Quantifiers added to the registry should be stateless and all their functions should be pure functions. Otherwise, reproduction of results might not be possible.

CHAPTER

FOUR

EXAMPLES

Besides the examples provided in the user guides for the usage of *models* and *quantifiers*, the following Jupyter notebooks explain specific tasks:

- **Creating a Stochastic Model using the Sequential API**This shows the simplest, and recommended, way to create an uncertainty aware DNN which is capable of calculating uncertainties and confidences based on point prediction approaches as well as on stochastic samples based approaches (e.g. MC-Dropout)

- **Convert a traditional keras Model into an uncertainty-aware model**This shows how you can use any keras model you may have, which was not created through uncertainty wizard, into an uncertainty-aware DNN.

- **Create a lazily loaded and highly parallelizable Deep Ensemble**This show the fastest way to create an even faster implementation of the powerful Deep Ensembles - in a way which respects the fact that your PC and your GPU are powerful and your time is costly.

- **Multi-Device Ensemble**This shows an example on how to use uncertainty wizard lazy ensembles using multiple gpus in parallel.

  Note: As this example is only applicable to machines with two gpus, we do not provide a colab link or jupyter notebook, but instead a classical python script to be run locally.

More examples will be added when we get feedback from our first users about the steps they found non-obvious. In the meantime, you may want to check out the *Complete API Documentation*.

CHAPTER

FIVE

ALL UNCERTAINTY WIZARD METHODS

Uncertainty Wizard aims for (and has, as far as we know) a 100% DocString coverage on public classes and methods. We recommend consulting the DocStrings in your IDE while using uncertainty wizard. Alternatively, the API can also be explored here:

- genindex
- modindex

Note that these indexes are auto-generated by Sphinx and are not always very nice to look at. Apparently a standard in many python packages, some people seem to like them, though ;-)

CHAPTER

SIX

PAPER

Uncertainty wizard was developed by Michael Weiss and Paolo Tonella at USI (Lugano, Switzerland). If you use it for your research, please cite these papers:

```
@inproceedings{Weiss2021FailSafe,
  title={Fail-safe execution of deep learning based systems through uncertainty␣
↪monitoring},
  author={Weiss, Michael and Tonella, Paolo},
  booktitle={2021 14th IEEE Conference on Software Testing, Verification and Validation␣
↪(ICST)},
  pages={24--35},
  year={2021},
  organization={IEEE}
}

@inproceedings{Weiss2021UncertaintyWizard,
  title={Uncertainty-wizard: Fast and user-friendly neural network uncertainty␣
↪quantification},
  author={Weiss, Michael and Tonella, Paolo},
  booktitle={2021 14th IEEE Conference on Software Testing, Verification and Validation␣
↪(ICST)},
  pages={436--441},
  year={2021},
  organization={IEEE}
}
```

The first paper (preprint) provides an empricial study comparing the approaches implemented in uncertainty wizard, and a list of lessons learned useful for reasearchers working with uncertainty wizard. The second paper (preprint) is a technical tool paper, providing a more detailed discussion of uncertainty wizards api and implementation.

References to the original work introducing the techniques implemented in uncertainty wizard are provided in the papers listed above.

Note that our documentation assumes basic knowledge of the tensorflow.keras API. If you do not know tensorflow.keras yet, check out the TensorflowGuide.