The Journal of Open Source Software

TorchSurv: A Lightweight Package for Deep Survival Analysis

Mélodie Monod [1], **Peter Krusche** [1], **Qian Cao** [2], **Berkman Sahiner** [2], **Nicholas Petrick** [2], **David Ohlssen** [3], **and Thibaud Coroller** [3¶]

1 Novartis Pharma AG, Switzerland **2** Center for Devices and Radiological Health, Food and Drug Administration, MD, USA **3** Novartis Pharmaceuticals Corporation, NJ, USA ¶ Corresponding author

DOI: 10.21105/joss.07341

Software

- Review
- Repository
- Archive

Editor: Kanishka B. Narayan

Reviewers:

- @WeakCha
- @XinyiEmilyZhang

Submitted: 24 July 2024
Published: 20 December 2024

Summary

TorchSurv is a Python package that serves as a companion tool to perform deep survival modeling within the PyTorch environment (Paszke et al., 2019). With its lightweight design, minimal input requirements, full PyTorch backend, and freedom from restrictive parameterizations, TorchSurv facilitates efficient deep survival model implementation and is particularly beneficial for high-dimensional and complex data analyses. At its core, TorchSurv features calculations of log-likelihoods for prominent survival models (Cox proportional hazards model (Cox, 1972), Weibull Accelerated Time Failure (AFT) model (Carroll, 2003)) and offers evaluation metrics, including the time-dependent Area Under the Receiver Operating Characteristic (ROC) curve (AUC), the Concordance index (C-index) and the Brier Score. TorchSurv has been rigorously tested using both open-source and synthetically generated survival data, against R and Python packages. The package is thoroughly documented and includes illustrative examples. The latest documentation for TorchSurv can be found on our website.

Statement of need

Survival analysis plays a crucial role in various domains, such as medicine, economics or engineering. Sophisticated survival analysis using deep learning, often referred to as "deep survival analysis," unlocks new opportunities to leverage new data types and uncover intricate relationships. However, performing comprehensive deep survival analysis remains challenging. Key issues include the lack of flexibility in existing tools to define survival model parameters with custom architectures and limitations in handling complex, high-dimensional datasets. Indeed, existing frameworks often lack the computational efficiency necessary to process large datasets efficiently, making them less suitable for real-world applications where time and resource constraints are paramount.

To address these gaps, we propose a library that allows users to define survival model parameters using custom PyTorch-based neural network architectures. By combining computational efficiency with ease of use, this toolbox opens new opportunities to advance deep survival analysis research and application. Figure 1 compares the functionalities of TorchSurv with those of auton-survival (Nagpal et al., 2022), pycox (Kvamme et al., 2019), torchlife (Abeywardana, 2021), scikit-survival (Pölsterl, 2020), lifelines (Davidson-Pilon, 2019), and deepsurv (Katzman et al., 2018). We notice that existing libraries constrain users to predefined functional forms for defining the parameters (e.g., linear function of covariates). Additionally, while there exist log-likelihood functions in the libraries, they cannot be leveraged. The limitations on the log-likelihood functions include protected functions (locked model architecture), specialized input requirements (format or class type), and reliance on external libraries like NumPy or Pandas. Dependence on external libraries hinders automatic gradient

calculation within PyTorch. Additionally, the implementation of likelihood functions instead of log-likelihood functions, as done by some packages, introduces numerical instability.

	TorchSurv	auton_survival[1]	pycox[2]	torchlife[3]	scikit-survival[4]	lifelines[5]	deepsurv[6]
PyTorch	✓	✓	✓	✓	✗	✗	✗
Standalone loss functions							
Weibull loss function	✓	✗	✗	✓	✗	✓	✗
Cox loss function	✓	✓	✓	✓	✓	✓	✗
Handle ties in event time	✓	✗	✗	✗	✓	✓	✗
Logarithm scale	✓	✓	✓	✗	✓	✓	✗
Standalone evaluation metrics							
Concordance index	✓	✗	✓	✗	✓	✓	✗
AUC	✓	✗	✗	✗	✓	✗	✗
Brier-Score	✓	✗	✓	✗	✓	✗	✗
Time-dependent risk score	✓	✗	✓	✗	✓	✗	✗
Subject-specific weights	✓	✗	✓	✗	✓	✗	✗
Confidence interval	✓	✗	✗	✗	✗	✗	✗
Compare two metrics	✓	✗	✗	✗	✗	✗	✗
Momentum	✓	✗	✗	✗	✗	✗	✗

Figure 1: Survival analysis libraries in Python. [1](Nagpal et al., 2022), [2](Kvamme et al., 2019), [3](Abeywardana, 2021), [4](Pölsterl, 2020), [5](Davidson-Pilon, 2019), [6](Katzman et al., 2018). A green tick indicates a fully supported feature, a red cross indicates an unsupported feature, a blue crossed tick indicates a partially supported feature. For computing the concordance index, pycox requires using the estimated survival function as the risk score and does not support other types of time-dependent risk scores. scikit-survival does not support time-dependent risk scores neither for the concordance index nor for the AUC computation. Additionally, both pycox and scikit-survival impose the use of inverse probability of censoring weighting (IPCW) for subject-specific weights. scikit-survival only offers the Breslow approximation of the Cox partial log-likelihood in case of ties in event time.

Functionality

Loss functions

Cox loss function. The Cox loss function is defined as the negative of the Cox proportional hazards model's partial log-likelihood (Cox, 1972). The function requires the subject-specific log relative hazards and the survival response (i.e., event indicator and time-to-event or time-to-censoring). The log relative hazards should be obtained from a PyTorch-based model pre-specified by the user. In case of ties in the event times, the user can choose between the Breslow method (Breslow, 1975) and the Efron method (Efron, 1977) to approximate the Cox partial log-likelihood.

```
from torchsurv.loss import cox

# PyTorch model outputs one log hazard per observation
my_cox_model = MyPyTorchCoxModel()

for data in dataloader:
    x, event, time = data  # covariate, event indicator, time
    log_hzs = my_cox_model(x)  # torch.Size([64, 1]), if batch size is 64
    loss = cox.neg_partial_log_likelihood(log_hzs, event, time)
    loss.backward()  # native torch backend
```

Weibull loss function. The Weibull loss function is defined as the negative of the Weibull AFT's log-likelihood (Carroll, 2003). The function requires the subject-specific log scale and log shape of the Weibull distribution and the survival response. The log parameters of the Weibull distribution should be obtained from a PyTorch-based model pre-specified by the user.

```
from torchsurv.loss import weibull
```

```
# PyTorch model outputs two Weibull parameters per observation
my_weibull_model = MyPyTorchWeibullModel()

for data in dataloader:
    x, event, time = data
    log_params = my_weibull_model(x)  # torch.Size([64, 2]), if batch size is 64
    loss = weibull.neg_log_likelihood(log_params, event, time)
    loss.backward()

# Log hazard can be obtained from Weibull parameters
log_hzs = weibull.log_hazard(log_params, time)
```

Momentum. Momentum helps train the model when the batch size is greatly limited by computational resources (i.e., large files). This impacts the stability of model optimization, especially when rank-based loss is used. Inspired by MoCO (He et al., 2020), we implemented a momentum loss that decouples batch size from survival loss, increasing the effective batch size and allowing robust training of a model, even when using a very limited batch size (e.g., ≤ 16).

```
from torchsurv.loss import Momentum

my_cox_model = MyPyTorchCoxModel()
my_cox_loss = cox.neg_partial_log_likelihood  # works with any TorchSurv loss
model_momentum = Momentum(backbone=my_cox_model, loss=my_cox_loss)

for data in dataloader:
    x, event, time = data
    loss = model_momentum(x, event, time)  # torch.Size([16, 1])
    loss.backward()

# Inference is computed with target network (k)
log_hzs = model_momentum.infer(x)  # torch.Size([16, 1])
```

Evaluation Metrics Functions

The `TorchSurv` package offers a comprehensive set of metrics to evaluate the predictive performance of survival models, including the AUC, C-index, and Brier score. The inputs of the evaluation metrics functions are the subject-specific risk score estimated on the test set and the survival response of the test set. The risk score measures the risk (or a proxy thereof) that a subject has an event. We provide definitions for each metric and demonstrate their use through illustrative code snippets.

AUC. The AUC measures the discriminatory capacity of the survival model at a given time t, specifically the ability to reliably rank times-to-event based on estimated subject-specific risk scores (Blanche et al., 2013; Heagerty & Zheng, 2005; Uno et al., 2007).

```
from torchsurv.metrics.auc import Auc
auc = Auc()
auc(log_hzs, event, time)  # AUC at each time
auc(log_hzs, event, time, new_time=torch.tensor(10.))  # AUC at time 10
```

C-index. The C-index is a generalization of the AUC that represents the assessment of the discriminatory capacity of the survival model across the entire time period (Harrell et al., 1996; Uno et al., 2011).

```
from torchsurv.metrics.cindex import ConcordanceIndex
cindex = ConcordanceIndex()
cindex(log_hzs, event, time)
```

Brier Score. The Brier score evaluates the accuracy of a model at a given time t (Graf et al., 1999). It represents the average squared distance between the observed survival status and the predicted survival probability. The Brier score cannot be obtained for the Cox proportional hazards model because the survival function is not estimated, but it can be obtained for the Weibull AFT model.

```
from torchsurv.metrics.brier_score import BrierScore
surv = weibull.survival_function(log_params, time)
brier = Brier()
brier(surv, event, time)  # Brier score at each time
brier.integral()  # Integrated Brier score over time
```

Additional features. In `TorchSurv`, the evaluation metrics can be obtained for risk scores that are time-dependent and time-independent (e.g., for proportional and non-proportional hazards). Additionally, subjects can optionally be weighted (e.g., by the inverse probability of censoring weighting (IPCW)). Lastly, functionalities including the confidence interval, a one-sample hypothesis test to determine whether the metric is better than that of a random predictor, and a two-sample hypothesis test to compare evaluation metrics obtained from two different models are implemented. The following code snippet exemplifies the aforementioned functionalities for the C-index.

```
cindex.confidence_interval()  # CI, default alpha=.05
cindex.p_value(alternative='greater')  # pvalue, H0:c=0.5, HA:c>0.5
cindex.compare(cindex_other)  # pvalue, H0:c1=c2, HA:c1>c2
```

Comprehensive Example: Fitting a Cox Proportional Hazards Model with TorchSurv

In this section, we provide a reproducible code example to demonstrate how to use `TorchSurv` for fitting a Cox proportional hazards model. We simulate data where each observations is associated with 10 features, a time-to-event that depends linearly on these features, and a time-to-censoring. The observable time is the minimum between the time-to-event and time-to-censoring, representing the first event that occurs. Subsequently, we fit a Cox proportional hazards model using maximum likelihood estimation and assess the model's predictive performance through the AUC and the C-index. To facilitate rapid execution, we use a simple linear backend model in PyTorch to define the log relative hazards. For more comprehensive examples using real data, we encourage readers to visit the `Torchsurv` website.

```
import torch
from torch.utils.data import DataLoader, Dataset
from torchsurv.loss import cox
from torchsurv.metrics.cindex import ConcordanceIndex
from torchsurv.metrics.auc import Auc

torch.manual_seed(42)

# 1. Simulate response
n_features = 10  # int, number of features per observation
time_end = torch.tensor(
    2000.0
)  # float, end of observational period after which all observations are censored
weights = (
    torch.randn(n_features) * 5
)  # float, weights associated with the features ~ normal(0, 5^2)

# Define the survival response generator function
```

```python
def tte_generator(batch_size: int):
    while True:
        x = torch.randn(batch_size, n_features)  # features
        mean_event_time, mean_censoring_time = 1000.0 + x @ weights, 1000.0

        event_time = (
            mean_event_time + torch.randn(batch_size) * 50
        )  # event time ~ normal(mean_event_time, 50^2)
        censoring_time = torch.distributions.Exponential(
            1 / mean_censoring_time
        ).sample(
            (batch_size,)
        )  # censoring time ~ Exponential(mean = mean_censoring_time)
        censoring_time = torch.minimum(
            censoring_time, time_end
        )  # truncate censoring time to time_end

        event = (event_time <= censoring_time).bool()  # event indicator
        time = torch.minimum(event_time, censoring_time)  # observed time

        yield x, event, time

# 2. Define the PyTorch dataset class
class TTE_dataset(Dataset):
    def __init__(self, generator: callable, batch_size: int):
        self.batch_size = batch_size
        self.generatated_data = generator(batch_size=batch_size)

    def __len__(self):
        return self.batch_size

    def __getitem__(self, index):
        return next(self.generatated_data)

# 3. Define the backbone model on the log hazards.
class MyPyTorchCoxModel(torch.nn.Module):
    def __init__(self):
        super(MyPyTorchCoxModel, self).__init__()
        self.fc = torch.nn.Linear(n_features, 1, bias=False)  # Simple linear model

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

# 4. Instantiate the model, optimizer, dataset and dataloader
cox_model = MyPyTorchCoxModel()
optimizer = torch.optim.Adam(cox_model.parameters(), lr=0.01)
batch_size = 64  # int, batch size
dataset = TTE_dataset(tte_generator, batch_size=batch_size)
dataloader = DataLoader(
    dataset, batch_size=1, shuffle=True
)  # Batch size of 1 because dataset yields batches

# 5. Training loop
for epoch in range(100):
    for i, batch in enumerate(dataloader):
```

```
        x, event, time = [t.squeeze() for t in batch]
        optimizer.zero_grad()
        log_hzs = cox_model(x)  # torch.Size([batch_size, 1])
        loss = cox.neg_partial_log_likelihood(log_hzs, event, time)
        loss.backward()
        optimizer.step()

# 6. Evaluate the model
n_samples_test = 1000  # int, number of observations in test set
data_test = next(tte_generator(batch_size=n_samples_test))
x, event, time = [t.squeeze() for t in data_test]  # test set
log_hzs = cox_model(x)  # log relative hazards evaluated on test set

# AUC at time point 1000
auc = Auc()
print(
    "AUC:", auc(log_hzs, event, time, new_time=torch.tensor(1000.0))
)  # tensor([0.5902])
print("AUC Confidence Interval:", auc.confidence_interval())  # tensor([0.5623, 0.6180])
print("AUC p-value:", auc.p_value(alternative="greater"))  # tensor([0.])

# C-index
cindex = ConcordanceIndex()
print("C-Index:", cindex(log_hzs, event, time))  # tensor(0.5774)
print(
    "C-Index Confidence Interval:", cindex.confidence_interval()
)  # tensor([0.5086, 0.6463])
print("C-Index p-value:", cindex.p_value(alternative="greater"))  # tensor(0.0138)
```

Conflicts of interest

MM, PK, DO and TC are employees and stockholders of Novartis, a global pharmaceutical company.

References

Abeywardana, S. (2021). *Torchlife: Survival analysis using pytorch*. https://doi.org/10.32614/CRAN.package.survival

Blanche, P., Dartigues, J., & Jacqmin-Gadda, H. (2013). Review and comparison of ROC curve estimators for a time-dependent outcome with marker-dependent censoring. *Biometrical Journal*, *55*(5), 687–704. https://doi.org/10.1002/bimj.201200045

Breslow, N. E. (1975). Analysis of survival data under the proportional hazards model. *International Statistical Review / Revue Internationale de Statistique*, *43*(1), 45. https://doi.org/10.2307/1402659

Carroll, K. J. (2003). On the use and utility of the weibull model in the analysis of survival data. *Controlled Clinical Trials*, *24*(6), 682–701. https://doi.org/10.1016/s0197-2456(03)00072-2

Cox, D. R. (1972). Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)*, *34*(2), 187–202. https://doi.org/10.1007/978-1-4612-4380-9_37

Davidson-Pilon, C. (2019). Lifelines: Survival analysis in python. *Journal of Open Source*

Software, *4*(40), 1317. https://doi.org/10.21105/joss.01317

Efron, B. (1977). The efficiency of cox's likelihood function for censored data. *Journal of the American Statistical Association*, *72*(359), 557–565. https://doi.org/10.1080/01621459.1977.10480613

Graf, E., Schmoor, C., Sauerbrei, W., & Schumacher, M. (1999). Assessment and comparison of prognostic classification schemes for survival data. *Statistics in Medicine*, *18*(17–18), 2529–2545. https://doi.org/10.1002/(sici)1097-0258(19990915/30)18:17/18%3C2529::aid-sim274%3E3.0.co;2-5

Harrell, F. E., Lee, K. L., & Mark, D. B. (1996). Multivariate prognostic models: Issues in developing models, evaluating assumptions and adequacy, and measuring and reducing errors. *Statistics in Medicine*, *15*(4), 361–387. https://doi.org/10.1002/(sici)1097-0258(19960229)15:4%3C361::aid-sim168%3E3.0.co;2-4

He, K., Fan, H., Wu, Y., Xie, S., & Girshick, R. (2020). Momentum contrast for unsupervised visual representation learning. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 9729–9738. https://doi.org/10.1109/cvpr42600.2020.00975

Heagerty, P. J., & Zheng, Y. (2005). Survival model predictive accuracy and ROC curves. *Biometrics*, *61*(1), 92–105. https://doi.org/10.1111/j.0006-341x.2005.030814.x

Katzman, J. L., Shaham, U., Cloninger, A., Bates, J., Jiang, T., & Kluger, Y. (2018). DeepSurv: Personalized treatment recommender system using a cox proportional hazards deep neural network. *BMC Medical Research Methodology*, *18*(1), 1–12. https://doi.org/10.1186/s12874-018-0482-1

Kvamme, H., Borgan, Ørnulf, & Scheel, I. (2019). Time-to-event prediction with neural networks and cox regression. *Journal of Machine Learning Research*, *20*(129), 1–30. http://jmlr.org/papers/v20/18-424.html

Nagpal, C., Potosnak, W., & Dubrawski, A. (2022). Auton-survival: An open-source package for regression, counterfactual estimation, evaluation and phenotyping with censored time-to-event data. *Machine Learning for Healthcare Conference*, 585–608. https://doi.org/10.48550/arXiv.2204.07276

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). *PyTorch: An imperative style, high-performance deep learning library*. https://doi.org/10.48550/arXiv.1912.01703

Pölsterl, S. (2020). Scikit-survival: A library for time-to-event analysis built on top of scikit-learn. *The Journal of Machine Learning Research*, *21*(1), 8747–8752. https://doi.org/10.5281/zenodo.3352342

Uno, H., Cai, T., Pencina, M. J., D'Agostino, R. B., & Wei, L. J. (2011). On the c-statistics for evaluating overall adequacy of risk prediction procedures with censored survival data. *Statistics in Medicine*, *30*(10), 1105–1117. https://doi.org/10.1002/sim.4154

Uno, H., Cai, T., Tian, L., & Wei, L. J. (2007). Evaluating prediction rules fort-year survivors with censored regression models. *Journal of the American Statistical Association*, *102*(478), 527–537. https://doi.org/10.1198/016214507000000149