

# Deep learning for survival in the browser

Jordan Anaya<sup>1</sup>

<sup>1</sup>The Fifth Axis LLC, San Jose, 95125 CA, USA

Corresponding author:

Jordan Anaya<sup>1</sup>

Email address: [support@fifthaxisllc.com](mailto:support@fifthaxisllc.com)

## ABSTRACT

Deep learning is a more flexible method for survival analysis than Cox regression, and is currently underutilized. Here, I introduce a fully client-side web application for building and training deep learning survival models. Users can upload their own data, generate simulated survival data, or query a database of TCGA mRNA expression and survival data. A model builder provides 27 TensorFlow.js layers to choose from, or users can provide their own model. Training follows best machine learning practices, utilizing Stratified K-Folds for model evaluation. Every computation runs on the client, keeping all data, models, and results on a user's machine. This resource is available at [AleaAxis.net](https://aleaaxis.net).

## BACKGROUND

I previously developed a resource for exploring survival correlations<sup>1</sup>. This resource, along with others<sup>2-5</sup>, utilizes Cox regression for survival analysis<sup>6</sup>. Cox regression was likely chosen for its explainability and widespread adoption within the field. However, Cox regression only models the linear contribution of each input feature to the log partial hazard (risk score, prognostic index), and in the absence of strong theory, there is no reason to assume linear contributions. As a result, the resources listed, and any other analyses involving Cox regressions, may have missed or misidentified nonlinear relationships.

Neural nets, when combined with an appropriate loss function, can identify these nonlinear relationships<sup>7-12</sup>. However, utilizing this technology requires installing specific software along with programming skills and knowledge of machine learning. To simplify its use and increase the adoption of using neural nets for survival analysis, I built a web application that provides user interfaces for the entire process.

## OVERVIEW

Alea Axis utilizes TensorFlow.js<sup>13</sup> to build and train deep learning models, and currently every computation is performed client-side. Having all the computations performed on the client allows for a responsive user-interface and precludes the need to send data to a server, providing complete user privacy and data security. However, this also means users will be limited by the hardware available to them. For survival training with a simple vector input and shallow graph, any modern computer with or without a dedicated GPU should be able to easily handle the training, but complex inputs such as large images and graphs with millions of parameters will require powerful GPUs and large amounts of RAM.

The application is designed to train deep learning models for evaluation, enabling model comparison or estimation of generalizability. The process of evaluating deep learning models requires data processing, data splitting, model building, hyperparameter selection, and finally, training and evaluation. Every one of these steps can be completed at Alea Axis without writing a single line of code, but users that want additional control have an upload option at certain steps to bypass the user interface and its limitations.

## STEPS

At [AleaAxis.net/train/survival](https://aleaaxis.net/train/survival), every step is given its own panel. Each step must be completed successfully to move on to future steps, and users can visit previous steps using the navigation above each panel. To the left of each panel is a

summary of the current state of the panel, along with upload and download options. The final submission for each panel is always the “Submit” button to the right. Helpful icons are provided, and informative error messages guide a user.

## Step 1: Data

At the Data Step, users must select either the “Provide Data”, “Generate Data”, or “Get Data” tabs. Alternatively, users can bypass the user interface by using the sidebar to upload data in JSON format (see [AleaAxis.net/docs](https://aleaaxis.net/docs)). Clicking on any of the tabs will clear any previously processed or uploaded data. The Provide Data tab is meant for either image inputs or tabular inputs. If a user has data that is incompatible with these formats, the JSON option will have to be used. The Generate Data tab will create tabular inputs (up to 10 features). The Get Data tab allows users to query a database containing TCGA data.

### Provide Data

Figure 1 illustrates how the panel will look if Provide Data is selected and “No” is selected for the image data prompt. A drop zone appears, which allows users to drag and drop, or click. The user’s file should be a plain text file with either comma or whitespace delimiters, and should contain a column for each feature, a column for times, and a column for events. Once an appropriate file is selected, the first 50 rows of the file and all its detected columns are shown to the user. This allows the user to inspect whether the columns were parsed correctly. The user can then select whether a header row is present and needs to be skipped, and specify which columns contain the inputs, the times, and the events. Any number of input feature columns is allowed, but they must be present consecutively so that the user can select all of them with the range slider.

The user must also identify whether their input features are numerical or categorical. Categorical inputs would be an input such as words. The selection will apply to all input columns, and the same vocabulary will be used for all the categorical inputs. For example, if each column contains a word, with multiple columns forming a complete document, the complete vocabulary will be the set of all the words in the rows and columns of inputs. Currently, it is not possible to have different input uploads. This would, for example, allow for both text and numerical inputs to be used together. When selecting categorical, the vocabulary size will be given to you, and the first layer of your model should be an embedding layer. Keep in mind that the embedding layer adds a dimension, which will need to be removed (for example, with a flatten or reshape layer).

Once the user has selected their input type, the option to “Parse File” will appear. If no errors are encountered, then the user will be given the option to select which value in the event column corresponds to an event. The event column should only contain two distinct values, for example, 0 and 1, or “Alive” and “Dead”, and must contain at least one value that corresponds to an event. Once a selection has been made, the user can finally “Process Data”. The exact inputs and labels that will be used as the dataset will be shown (first 50), with the labels being [time, event], and 0 indicating no event and 1 indicating an event. If everything looks correct, the user can click “Submit Data” to the right, and a summary of the dataset will appear to the left, along with an option to download it in JSON format (image data should not be downloaded in this format).

The user can now proceed to “Split Data” or take a detour to “Analysis”. Analysis is only available for single-valued inputs, which was the case here (mRNA expression values). This panel is described below.

### Generate Data

Generating survival data is a two-step process. First, features along with their relationship to a risk score must be generated. These risk scores will then be used to create time-to-event data with censoring. The start of Figure 2 shows the first step in the process. Users can generate between 100 and 10,000 samples, with up to 10 independent features. Each of these independent features can then have relationships with up to 10 targets, but our risk score must be a single value. This constraint requires the user to have 1 target per feature if the user sums over the target axis to create risk scores, and only 1 feature if the user sums over the feature axis to create risk scores.

In Figure 2, we have a single feature, a single target, and 500 samples. Once the “Target 1” tab is selected, we can choose our “Input Transformation”. We have  $x^2$  selected, which will create a non-monotonic relationship between our feature and

risk scores. Proceeding to “Label Creation”, we are able to see our risk scores from the perspective of each feature. We can also make some final adjustments to the risk scores prior to making our survival data. In this case, we adjusted the risk scale since the original values were larger than we are likely to encounter in actual data.

**Survival Training**

Home Learn Practice [Train](#) About

Data Split Data. Model Training Parameters. Train.

**Choose one way to select data.**

Provide Data Generate Data Get Data

Submit Data

Current Data  
None  
Task: Survival

Are you uploading images?  
No ☒ Yes ☐

Drag or select a single CSV/Text file  
The file should contain your inputs, times, and events.

↓

**Choose one way to select data.**

Provide Data Generate Data Get Data

Submit Data

Current Data  
None  
Task: Survival

Are you uploading images?  
No ☒ Yes ☐

Header row? ☒  
Time Column: 2 Event Column: 3

Input Columns

Raw Data (first 50 rows)

	1	2	3	4
1	id	time	status	expression
2	TCGA-B0-5702	2172	Alive	15.78
3	TCGA-AK-3447	1217	Alive	37.58
4	TCGA-B0-5117	1608	Alive	44.0

Parse File

Input type? ☒ Numerical ☐ Categorical

↓

**Choose one way to select data.**

Provide Data Generate Data Get Data

Submit Data

Current Data  
Samples: 522  
Input Shape: (1,)  
Task: Survival

Are you uploading images?  
No ☒ Yes ☐

Header row? ☒  
Time Column: 2 Event Column: 3

Input Columns

Raw Data Inputs Labels

1	[2172, 0]
2	[1217, 0]
3	[1608, 0]
4	[1314, 0]
5	[2090, 1]
6	[2531, 0]
7	[919, 0]

Parse File

Input type? ☒ Numerical ☐ Categorical

Event Label: Dead

**Your data is ready!**

522 Inputs with shape [1,]  
522 Labels of [time, event]  
174 Samples with an event

**Figure 1. Providing tabular data.**

↑ Data ?

Current Data

None

Task:

Survival

Choose one way to select data.

Submit Data

Provide Data      Generate Data      Get Data

Number of Samples: 500      Number of Features: 1      Number of Targets: 1      [To Label Creation](#)

☐ Switch ?

**Feature 1**

**Input Distribution ?**

Uniform      Normal      LogNormal

**Input Transformation ?**

None      x      **x<sup>2</sup>**      x<sup>3</sup>      sin      sigmoid      exp

☐ Reflect X      ☐ Reflect Y

**Input Scale ?**      **Input Noise ?**

.1X      5X      10X      0%      50%      100%

**Target 1**

3.98

target 1

-.000      -1.99      -.016      1.96

feature 1



<

↑ Data ?

Current Data

None

Task:

Survival

Choose one way to select data.

Submit Data

Provide Data      Generate Data      Get Data

[Back to Features](#)      **Mean Survival Time ?**      **Censor Scale ?**      **Censor Shape ?**

1000      1      1

**Risk Noise ?**      ☐ Log these Risks? ?

**Risk Scale ?**

0%      50%      100%

**Make Survival Data**

**Feature 1**

1.99

risk

-.000      -1.99      -.016      1.96

feature 1



<

↑ Data ?      ↓ Data ?

Current Data

Samples:

500

Input Shape:

(1,)

Task:

Survival

Choose one way to select data.

Submit Data

Split Data

Analysis

Provide Data      Generate Data      Get Data

[Back to Features](#)      **Mean Survival Time ?**      **Censor Scale ?**      **Censor Shape ?**

1000      1      1

**Data (first 50)**      **Event Counts**

Sample	Time	Event
1	199.92	1
2	141.65	1
3	1853.96	0
4	848.39	1
5	84.3	0
6	286.62	1
7	244.76	0

**Hide Kaplan**

Surviving

Time

**Back to Risk Options**

Figure 2. Generating survival data.

Survival times will be generated with an exponential distribution, and censoring will be performed with a Weibull distribution<sup>14</sup>. The “Mean Survival Time” should not have any impact on censoring, and its only potential impact is on rounding effects. For example, if a researcher plans on perfectly mimicking a real-world dataset, and in that dataset the times are rounded to the nearest month, then in a generated dataset with a mean time of 100 days, many samples would share the exact same time point (the month).

The values that will affect the level of censoring are the risk scores, the “Censor Scale”, and the “Censor Shape”. Once the survival data is made and a user is viewing the survival data, any change to the censoring values will immediately update the data. A tab for “Event Counts” allows researchers to quickly check if they’ve achieved their desired level of censoring. Given the compute required to generate the plot, an option to hide the plot is available and may speed up the process. When viewing the risk options, adjusting the survival parameters does not generate new survival data.

### **Get Data**

A database of TCGA data is made available to users (see methods). Users can retrieve data for one cancer at a time. Either Ensembl ID or name can be used to query the database, and suggestions will be presented for names. Researchers who have previously investigated a gene in the TCGA dataset with Cox regression may want to use this resource to confirm their previous results.

### **Optional Detour: Analysis**

Regardless of how a user submitted their data, if the input is single-valued, an optional detour to an analysis panel becomes available. Figure 3 shows the analysis panel for the synthetic data we just generated. The panel immediately notifies the user of the C-index for their data and the C-index for the risk scores (for synthetic data only). An option to find the optimal splitting (based on input value) is available, along with an option to split the data at a user-defined cutoff and visualize the results (fractional percents require typing into the “Cutoff” input).

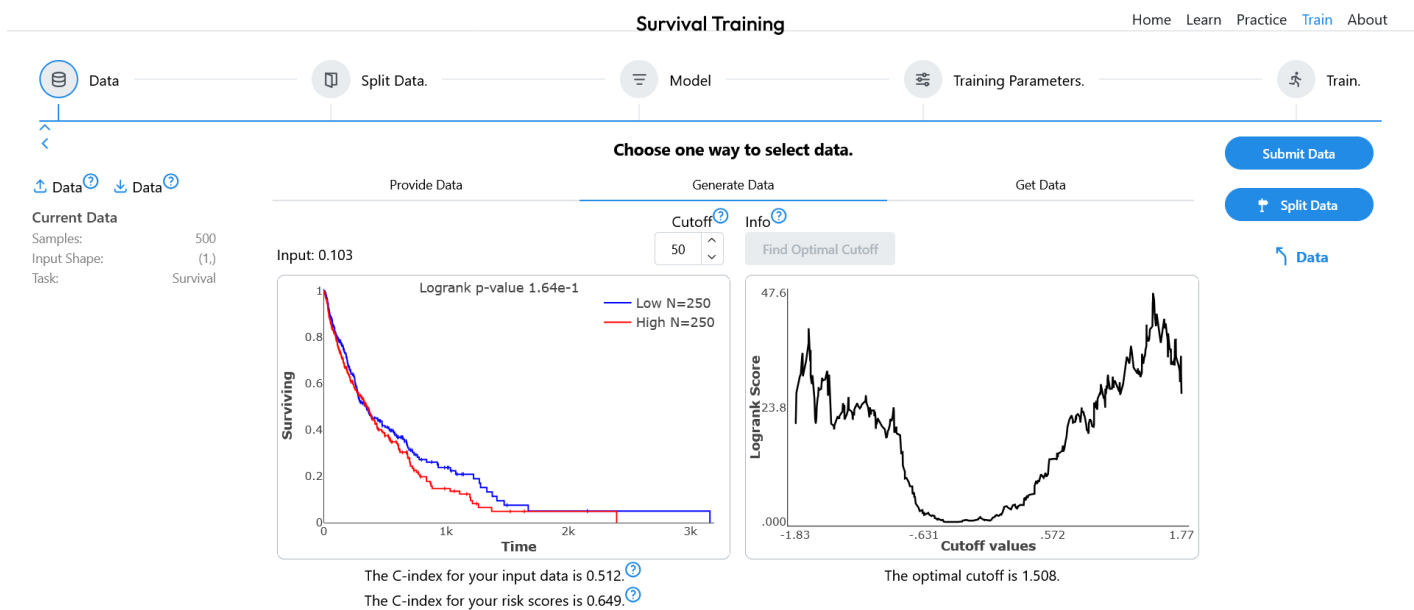
In our case, because our risk scores were non-monotonic (originally high, then low, then high again), splitting the data at the 50th percentile does not result in a difference in survival, and we also observe a near baseline C-index value of .512. The optimal cutoff plot shows us that splitting near the middle of input values is actually the worst way to split the data. The plot suggests splitting at either lower or higher percentiles (for this data, the optimal split actually requires 2 cutoff values). Once a user leaves the Data Step, returning will clear their previous dataset, so users should take note of the statistics in this panel if they hope to reference them during the Training Step.

### **Optimal Cutoff**

Depending on the data, the optimal cutoff can be unreliable<sup>15</sup>, and the p-value from the log-rank statistic needs to be corrected<sup>16</sup>. The cutoff will be most reliable for large sample sizes and a large step function. For smaller sample sizes, smaller differences in risk scores, and a smooth transition from low to high risk, the cutoff value can take on a range of values due to the noise inherent to survival data. Users may want to repeatedly generate survival data that attempts to mimic the properties of their actual or expected data, and observe the range of possible cutoff values.

Another issue with the optimal cutoff, which I haven’t seen addressed, is that it can be biased by censoring. As an example, imagine the case of strictly increasing risk scores. Here, samples with lower input values have lower risk scores, while samples with larger input values have higher risk scores. If each sample has the same constant rate of censoring, then the lower risk score samples will be more likely to be censored than the higher risk score samples. This creates a shift in the estimated optimal cutoff (in this case, towards the higher input values). Censoring is an artifact of survival studies, and in reality, every patient will eventually have an event (if the event is death).

Stated another way, if two studies observed the exact same group of patients, but one study lost fewer patients to follow-up or continued for a longer duration, the two studies could observe different optimal cutoffs in their data. I’m not sure how to mathematically estimate the size of this bias. However, if a user is confident that they can generate simulated data that closely matches their experimental data, then they can generate data with no censoring and find the optimal cutoff of this theoretical dataset of samples with perfect follow-up.



**Figure 3. Analysis panel.** Analysis of data generated in Figure 2.

## Step 2: Split Data

Stratified K-Folds will be used for model evaluation. This method generates folds such that each sample appears in a testing fold exactly once while keeping the folds as similar to each other as possible with respect to a label distribution. Users can choose to use events as the label, use no stratification, or upload their own stratification labels. Each training fold is what the model will use for weight updates, while the testing fold will never be shown to the model during the course of training and is used solely for post-training evaluation. Optionally, users can further split the training fold into training and validation, with the validation fold being used for early stopping.

## Step 3: Model Building

Alea Axis uses the Functional API of TensorFlow.js to build models, which allows for layers that take in multiple inputs, or branching models (specifying an input layer that is not the immediately preceding layer). The model must start with an input layer whose shape matches the input data, and the final layer must be a dense layer with 1 unit and no activation (our loss function requires predicted risk scores). The input layer is provided for the user, but the final dense layer is left for the user to provide. Between these two layers, the user can use as many layers or branches as they desire (but above a certain number of parameters, TensorFlow.js will be unable to build the model). For layers that don't require specifying the input layer, the immediately preceding layer is the input. 27 different TensorFlow.js layers are available to select from, with most of the Basic, Merge, Convolutional, Pooling, and Normalization layers included. For most layers, the user interface only exposes the essential parameters. Terminology follows the [TensorFlow.js API](#).

In this panel, the user has the option to upload or download the layers, or upload and download a model. Layers are an Alea Axis-specific JSON format that are used to populate the user interface. Users will likely want to save the layers since it will allow them to quickly edit the layers in the future. The model is a file created by TensorFlow.js, and can also include a separate weight file. If a user wants to use layers not available through the user interface, they could provide a TensorFlow.js model created elsewhere (as long as the input layer is compatible with their data and the model has the correct output layer). When using layers, the user must always click "Build Model" to create their TensorFlow.js model. If an error occurs during this process, the message from TensorFlow.js will be provided to the user.

Upon successfully building or uploading a model, the user will be able to view the model summary. Figure 4 shows the model summary for a simple model with 2 hidden dense layers with dropout. When a user downloads a model from

this panel, the weights are always the initial model weights, regardless of whether they return to this panel after training a model.

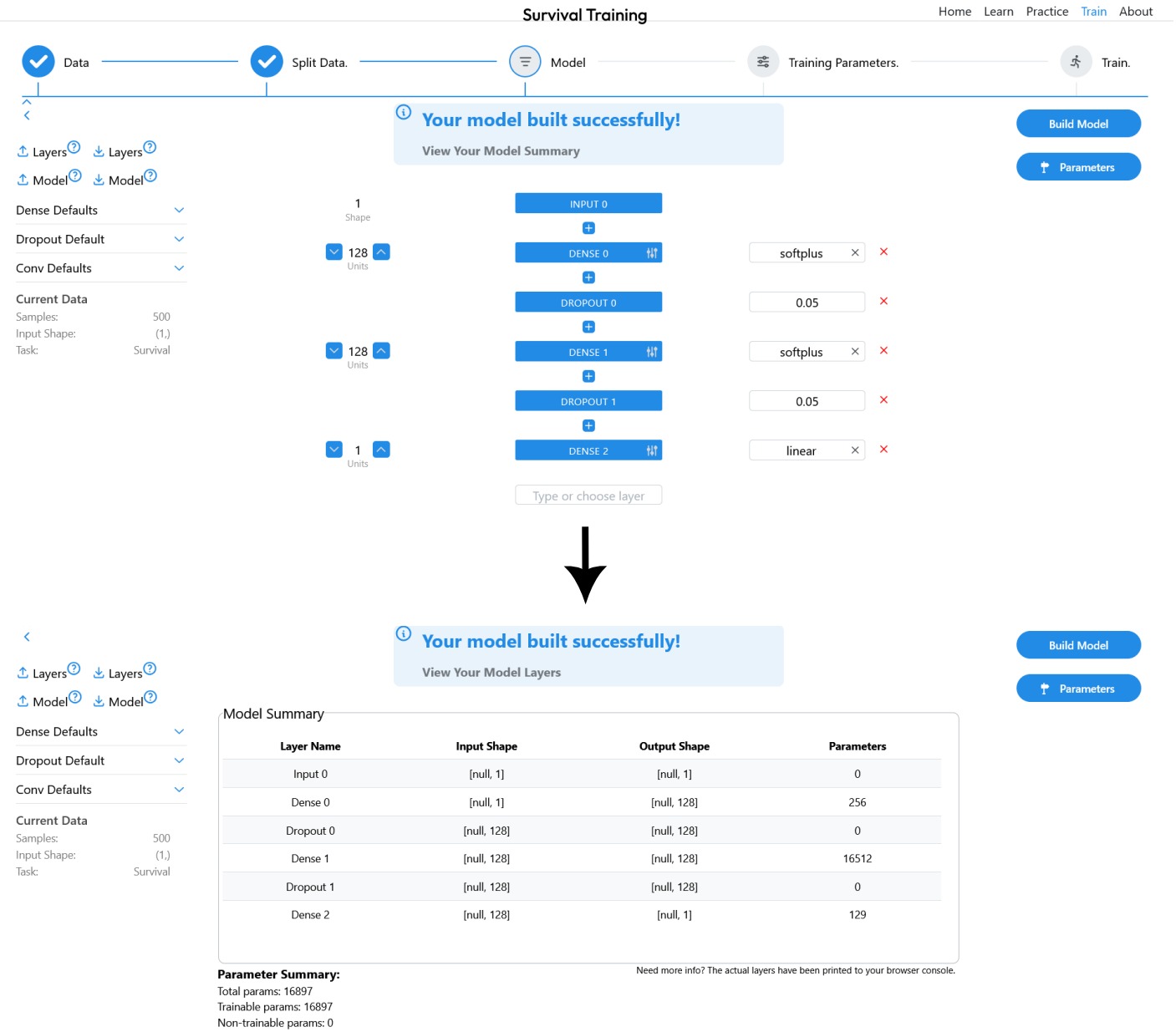


Figure 4. Building a model.

Step 4: Training Parameters

At the Training Parameters panel, users must first compile the model they just built (Figure 5). This requires selecting a loss, optimizer, learning rate, and optionally, metrics. Currently, only one loss is available, average negative partial log-likelihood (the loss used in Cox regression). All optimizers available in TensorFlow.js that only require a learning rate parameter are made available. For metrics, users can optionally select the Cox loss or C-index. Users should be aware that if they have thousands of samples, the C-index metric function can require large amounts of RAM.

Traditionally, an epoch is defined as one pass through the dataset, with the number of batches equal to the number of training samples divided by the batch size. This potentially results in a partial final batch if the number of training samples



is not perfectly divisible by the batch size. The Cox loss can only be calculated when a batch has an event, and is only informative when there are multiple samples. As a result, it is imperative that a partial batch is never created.

At Alea Axis, a batch will always be of the requested size, and an epoch can contain any number of batches (capped at training fold size for practical purposes). You can think of an epoch as how much data the model sees before it calculates and records metrics. If you want very fine-grained records, you would create smaller epochs. Model evaluation comes at a computational cost, and it can be more efficient to record metrics less often. The epoch definition also impacts the EarlyStopping callback. For example, small epochs with noisy metric updates will require larger patience values.


Best weight restoration also differs at Alea Axis. TensorFlow/Keras users will be accustomed to the best weights being restored when the EarlyStopping callback is triggered. However, the EarlyStopping callback in TensorFlow.js does not implement this functionality, and even if it did, the risk of hitting the epoch limit before an EarlyStopping trigger was always a flaw in this paradigm. Instead, Alea Axis maintains its own record of the best weights (as defined by the monitored metric) during training, and upon the completion of training, always restores them. Training for the current fold will be deemed complete if the EarlyStopping callback is triggered, the epoch limit is reached, or NaNs are encountered. At the completion of each fold, these best weights are then used by the model to make predictions for model evaluation.


If a user's batch does not contain any events, the loss cannot be calculated, and in this circumstance, I return a NaN. I could instead return 0; however, this would lower the overall loss for that epoch, and when the best weights are restored, the epoch that had missing events may be selected instead of the actual best epoch. To avoid encountering NaNs due to a lack of events, users simply have to maintain large batch sizes. Large batch sizes also help stabilize the Cox loss, since it compares samples to each other and is unstable for small numbers of comparisons. For this reason, the default batch size provided for the user is the training fold size. As long as memory permits, users will likely want to keep the batch size as large as possible.


Large input values can hinder model convergence and lead to longer training times, or even cause NaNs. Users will always be provided the option to subtract the mean and divide by SD (as calculated from each training fold), and if their input has a single dimension (i.e., is a vector), they will also have the option to take a natural logarithm. Turning off the shuffling of the TensorFlow dataset can slightly increase the speed of training, and users are provided the option to disable it (see methods). For the EarlyStopping callback, "Overall Loss" is equivalent to the Cox loss if a user has no regularization in their model, and there is no need to separately monitor the Cox loss.


Home Learn Practice **Train** About


---


 Data


 Split Data


 Model


 Training Parameters


 Train

 Data

 Split Data

 Model

 Training Parameters

 Train

**Epoch Definition**

Epochs: 1000

Train Batch Size: 400

Batches Per Epoch: 4

**Early Stopping**

Min Delta: 0

Patience: 40

Metric: loss

**Model Compile**

**Loss**

Value to be optimized

Cox

**Optimizer**

Model optimizer

adam

**Metric**

Optional values to monitor

Cox X

C-index X

**Learning rate**

Optimizer learning rate

0.0010

↑

↓

**Early Stopping**

**Metric/Loss**

Monitor for early stopping

Overall Loss

**Metric delta**

Model improvement delta

0.0000

↑

↓

**Patience**

Number of epochs to wait

40

↑

↓

**Optional Dataset Options**

**Transformation method?**

Strategy for feature scaling

None

Shuffle your dataset?

**Epoch Definition**

**Epochs**

Max training epochs

1000

↑

↓

**Batches per epoch**

Training batches per epoch

4

↑

↓

**Train batch size**

Samples per training batch

400

↑

↓

Submit

↑

 Training

Figure 5. Training parameters.



Step 5: Training and Evaluation

Once a user starts training, it will continue through all fold splits without any user intervention (browser tab must remain active), with the user interface updating at the end of each epoch. Clicking on each fold allows users to view plots for the loss and optional metrics for the training fold, and the validation fold if present. If the input data is a single value for each sample, a prediction plot will also be available. At the completion of each fold, an evaluation tab becomes available, which shows the training and testing metrics for the best weights of that fold split (and optionally validation metrics). These evaluations are from model predictions, and as a result, the training metrics may not match the plots if dropout is being used (dropout is turned off when predictions are made). For small fold sizes, both the Cox loss and C-index may show a large amount of variability between fold splits. Keep in mind that the Cox loss is dependent on fold size, with smaller folds having smaller losses.

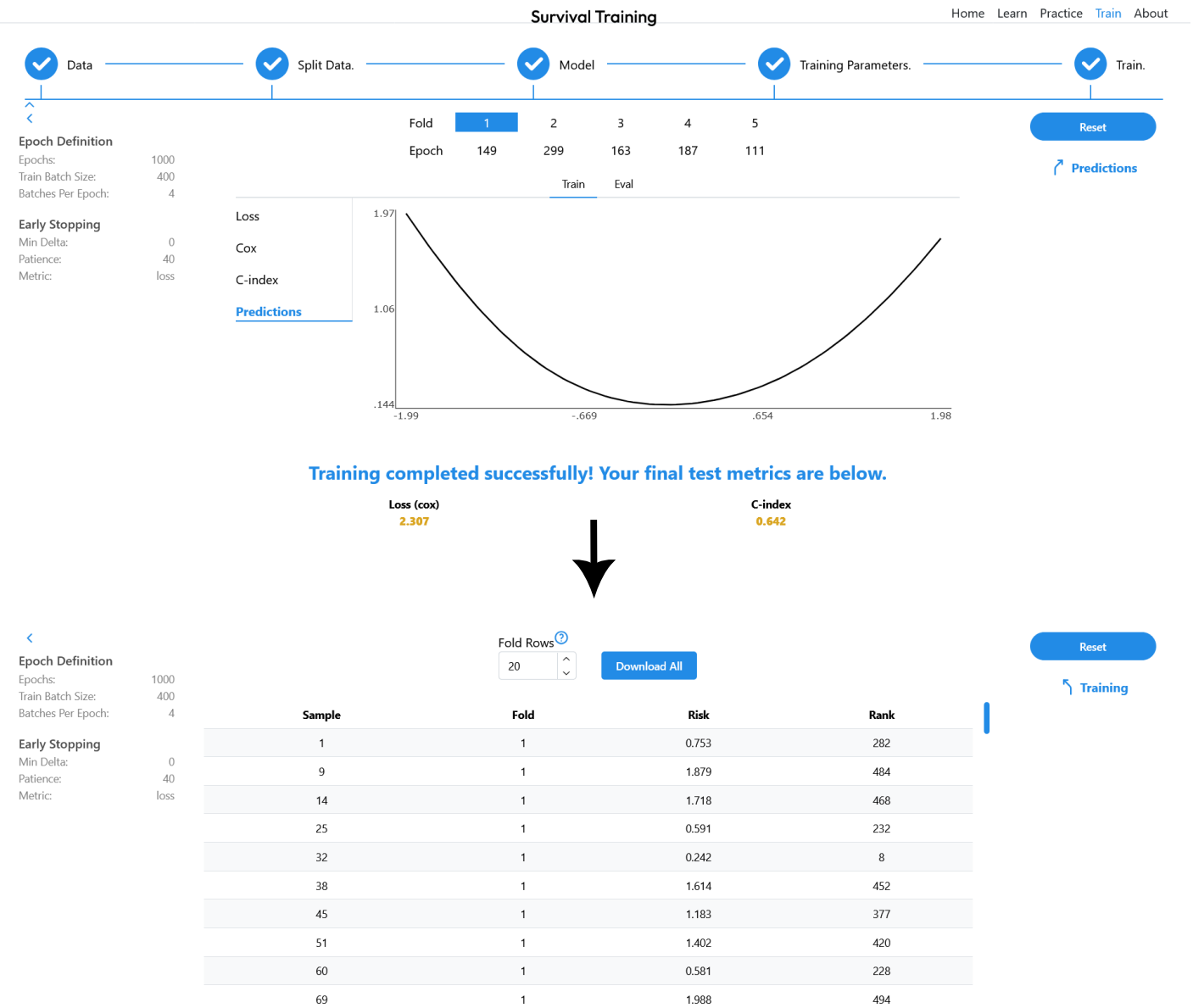


Figure 6. Training and evaluation. Training with the dataset from Figure 2, the model from Figure 4, and the parameters from Figure 5.

Figure 6 shows a completed training run from the data we generated in Figure 2, the model we built in Figure 4, and the training parameters from Figure 5. As shown, once training of all folds is complete, metrics that take into account

all testing folds are provided. The C-index calculation provided here is not affected by small sample sizes and is even compatible with leave-one-out cross-validation (see methods). Referring back to Figure 3, the true C-index for the hidden risk scores was .649. Although our model appears to have correctly modeled the risk scores (based on the prediction plots) for each fold, there is inherent randomness in going from risk scores to time, event data. In addition, with five folds the training data only contained 80% of the dataset, and the vertex of our predictions may not have been perfectly centered. With more folds our C-index would likely approach the theoretical value.

Once training is complete, an option to view all testing fold predictions also becomes available. A small table allows for quick viewing, with an option to download all the predictions to a CSV file. The scale of the risk scores is fold-dependent, and they should not be compared across folds. To compare across the different testing folds, the ranks should be used (see methods).

## RECOMMENDATIONS

The question on many users' minds will likely be whether the neural net they trained was a better fit than a Cox regression. The final metrics include the average negative partial log-likelihood averaged across the testing folds and the C-index for the testing ranks. If the neural net is a better fit, it should have a better partial log-likelihood. When comparing the partial log-likelihoods between models, it is essential to use the exact same fold indices.

For a single-valued input, the neural net will only have a better C-index if the true underlying relationship is non-monotonic. In the synthetic data we generated, performing a Cox regression would have resulted in a C-index of .512 while the neural net found a C-index of .642. But if we had a sigmoid-like relationship, there would have been no differences in the C-index.

For single-valued inputs, researchers should also look at the training predictions for each fold. A strong relationship should have the same shape for every fold. If the shape of the relationship changes each fold, that indicates the neural net is simply learning noise.

## METHODS

### Software

Alea Axis is built with TypeScript, React, Mantine components, Tailwind CSS, TanStack Router and Query, Plotly.js, and stdlib. The database of TCGA data is managed with a Django server.

### System Requirements

If a user's browser is compatible with WebGL and has it enabled, TensorFlow.js will utilize a GPU if one is available. All modern browsers should have WebGL enabled by default. Different browsers may have different performance. On my system, Windows 11 and AMD Radeon 780M, Google Chrome is much faster than Mozilla Firefox at training models. For a ballpark figure on how fast you should expect training to proceed, in the example provided in this paper, 100 epochs took around 7 seconds in Chrome.

### TensorFlow Datasets

Batching is performed with `.shuffle().repeat().batch()`. Without the shuffle, an array such as [1, 2, 3, 4] with batch 2 would create batches [1, 2], [3, 4] in the first iteration, and [1, 2], [3, 4] in the next iteration. With shuffling of the starting array each iteration, the batches each iteration will be random. Because the order of samples within a batch is irrelevant, when the batch size is the length of the array, shuffling has no impact on training. Users who want training to proceed as quickly as possible may also consider removing shuffling even when the batch size is smaller than the size of their training fold. Removing shuffling will also allow training to proceed deterministically (as long as you don't have dropout and use the same initial weights).

## Cox Loss

The loss calculated is the average negative partial log-likelihood:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N E_i \left[ \log \sum_{j \in \mathcal{R}(t_i)} \exp(f_{\theta}(\mathbf{x}_j)) - f_{\theta}(\mathbf{x}_i) \right] \quad (1)$$

where  $t_i$  and  $E_i$  denote the observed time and event indicator for sample  $i$ , respectively ( $E_i = 1$  if the event was observed,  $E_i = 0$  if censored).  $\mathbf{x}_i$  are the features of sample  $i$ ,  $f_{\theta}$  is the neural net parameterized by  $\theta$  (the weights of the neural net), and therefore  $f_{\theta}(\mathbf{x}_i)$  is the neural net output for sample  $\mathbf{x}_i$ .

$$\mathcal{R}(t_i) = \{ j : t_j \geq t_i \} \quad (2)$$

is the *risk set* at time  $t_i$ , comprising all samples still under observation at the moment of sample  $i$ 's event.

Because the predictions of the neural net can drift over time (or bad initial weights can lead to predictions far from 0), and because the partial log-likelihood has an exponential term, this loss can encounter numerical underflow or overflow. To make the loss function resistant to this, prior to calculating the loss I subtract the mean of the predictions from themselves. A standard method for providing numerical stability is to subtract the maximum. I did not want the loss to be numerically stable under all circumstances. If a user provided large input values (inputs are preferably near the range of -1 and 1), a max subtraction would cause vanishing gradients, and the model would not train. In this circumstance, I would prefer the loss result in a numerical overflow and generate NaNs to alert the user that there is a potential issue with their data rather than leave them guessing as to why their model is not training. As mentioned previously, I also have the loss return NaN if a batch contains no events.

## C-index calculation

The final C-index calculation follows the implementation in [12]. The C-index requires us to compare the risk score predictions to each other, but they are not comparable across folds. To make the risk scores comparable, I convert them to ranks in each fold split. One way to do this would be to create regularly spaced input values across the entire input feature space (a linspace) and observe the ranks of the testing fold samples among these simulated samples. While this would be trivial for a dataset that has a single-valued input, this is more complex and computationally expensive for an input such as images. We can instead utilize the input space already loaded in RAM, i.e., our entire dataset. Briefly, for each fold split, the ranks for every sample in the dataset are determined, and the ranks corresponding to the testing fold are recorded. All the testing fold ranks are then used to calculate a C-index.

## TCGA Processing

Clinical data was obtained from<sup>17</sup>. Expression files for each cancer were downloaded from <https://portal.gdc.cancer.gov/> (the rna\_seq.augmented\_star\_gene\_counts.tsv files, release v45.0). Only genes which had a "gene\_type" of "protein\_coding" were used. The "fpkm\_unstranded" value was recorded as the expression value. Both the GENCODE v36 Ensembl ID and gene name were saved as database fields, however the following names mapped to more than one Ensembl ID: ACTL10, MATR3, PDE11A, POLR2J3, SMIM40, TMSB15B. In these cases the first Ensembl ID and corresponding expression value were kept, with the following entries discarded. Only primary tumors were allowed, with the exception of SKCM, in which case metastatic tumors were also allowed. If a patient had more than one expression file for a tumor that matched these criteria, the expression values were averaged. To maximize performance, the clinical data is stored as binary files while the expression data for 17 cancers and 19938 genes is stored as individual BLOBs (Binary Large Objects) in an indexed relational database table.

## AI DISCLOSURE

Large language models were used to check the grammar of this article.

## AUTHORS' DISCLOSURES

Jordan Anaya is managing member of The Fifth Axis LLC, which owns and operates Alea Axis.

## ACKNOWLEDGMENTS

The results here are in whole or part based upon data generated by the TCGA Research Network.

## FUNDING

No funding was involved in the production of this work.

## REFERENCES

1. Anaya, J. OncoLnc: linking TCGA survival data to mRNAs, miRNAs, and lncRNAs. *PeerJ Computer Science* **2**, e67 (2016).
2. Györfy, B. *et al.* An online survival analysis tool to rapidly assess the effect of 22,277 genes on breast cancer prognosis using microarray data of 1,809 patients. *Breast cancer research and treatment* **123**, 725–731 (2010).
3. Gentles, A. J. *et al.* The prognostic landscape of genes and infiltrating immune cells across human cancers. *Nature medicine* **21**, 938–945 (2015).
4. Smith, J. C. & Sheltzer, J. M. Genome-wide identification and analysis of prognostic features in human cancers. *Cell reports* **38**, 110569 (2022).
5. Benard, B. A., Lalgudi, C. K., Ilertsen, I., Wang, R. H. & Gentles, A. J. PRECOG update: an augmented resource of clinical outcome associations with gene expression for adult, pediatric, and immunotherapy cohorts. *Nucleic Acids Research* **54**, D1579–D1589 (2026).
6. Cox, D. R. Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)* **34**, 187–202 (1972).
7. Faraggi, D. & Simon, R. A neural network model for survival data. *Statistics in medicine* **14**, 73–82 (1995).
8. Katzman, J. L. *et al.* DeepSurv: personalized treatment recommender system using a Cox proportional hazards deep neural network. *BMC medical research methodology* **18**, 1–12 (2018).
9. Ching, T., Zhu, X. & Garmire, L. X. Cox-nnet: an artificial neural network method for prognosis prediction of high-throughput omics data. *PLoS computational biology* **14**, e1006076 (2018).
10. Wang, D., Jing, Z., He, K. & Garmire, L. X. Cox-nnet v2. 0: improved neural-network-based survival prediction extended to large-scale EMR data. *Bioinformatics* **37**, 2772–2774 (2021).
11. Zhan, Z. *et al.* Two-stage Cox-nnet: biologically interpretable neural-network model for prognosis prediction and its application in liver cancer survival using histopathology and transcriptomic data. *NAR genomics and bioinformatics* **3** (2021).
12. Anaya, J., Kung, J. & Baras, A. S. Characterization of non-monotonic relationships between tumor mutational burden and clinical outcomes. *Cancer Research Communications* **4**, 1667–1676 (2024).
13. Smilkov, D. *et al.* Tensorflow.js: Machine learning for the web and beyond. *Proceedings of machine learning and systems* **1**, 309–321 (2019).
14. Bender, R., Augustin, T. & Blettner, M. Generating survival times to simulate Cox proportional hazards models. *Statistics in Medicine* **24**, 1713–1723 (May 2005).
15. Altman, D. G., Lausen, B., Sauerbrei, W. & Schumacher, M. Dangers of Using “Optimal” Cutpoints in the Evaluation of Prognostic Factors. *JNCI Journal of the National Cancer Institute* **86**, 829–835 (June 1994).
16. Gurjao, C., Tsukrov, D., Imakaev, M., Luquette, L. J. & Mirny, L. A. Is tumor mutational burden predictive of response to immunotherapy? *Elife* **12** (2024).
17. Liu, J. *et al.* An integrated TCGA pan-cancer clinical data resource to drive high-quality survival outcome analytics. *Cell* **173**, 400–416 (2018).