main

August 30, 2025

1 Titanic Kaggle Competition

The Titanic competition on Kaggle presents the challenge of identifying the factors that contribute to surviving the sinking of the ship.

[1]:
```
import collections
import typing

import numpy as np
import pandas as pd

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import Tensor

from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from matplotlib.figure import Figure
from matplotlib.axes import Axes
PltAxes: typing.TypeAlias = typing.Union[typing.Sequence[Axes], typing.
 ↪Sequence[typing.Sequence[Axes]], np.ndarray, Axes]

import tqdm
```

We have the following notes about the dataset: - `survival` Survival (0 = No, 1 = Yes) - `pclass` Ticket class (1 = 1st, 2 = 2nd, 3 = 3rd) - `sex` Sex - `Age` Age in years - `sibsp` # of siblings / spouses aboard the Titanic - `parch` # of parents / children aboard the Titanic - `ticket` Ticket number - `fare` Passenger fare - `cabin` Cabin number - `embarked` Port of Embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

[2]:
```
df = pd.read_csv('train.csv')
df.describe()
```

[2]:
```
       PassengerId    Survived      Pclass         Age       SibSp  \
count   891.000000  891.000000  891.000000  714.000000  891.000000
```

```
mean    446.000000    0.383838    2.308642   29.699118    0.523008
std     257.353842    0.486592    0.836071   14.526497    1.102743
min       1.000000    0.000000    1.000000    0.420000    0.000000
25%     223.500000    0.000000    2.000000   20.125000    0.000000
50%     446.000000    0.000000    3.000000   28.000000    0.000000
75%     668.500000    1.000000    3.000000   38.000000    1.000000
max     891.000000    1.000000    3.000000   80.000000    8.000000

            Parch        Fare
count  891.000000  891.000000
mean     0.381594   32.204208
std      0.806057   49.693429
min      0.000000    0.000000
25%      0.000000    7.910400
50%      0.000000   14.454200
75%      0.000000   31.000000
max      6.000000  512.329200
```

[3]:
```
df.head()
```

[3]:
```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3

                                                Name     Sex   Age  SibSp  \
0                            Braund, Mr. Owen Harris    male  22.0      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th…  female  38.0      1
2                             Heikkinen, Miss. Laina  female  26.0      0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4                           Allen, Mr. William Henry    male  35.0      0

   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
3      0            113803  53.1000  C123        S
4      0            373450   8.0500   NaN        S
```

It's also important to see how much data is missing so we can figure out the best way to handle it.

[4]:
```
df.isna().sum()
```

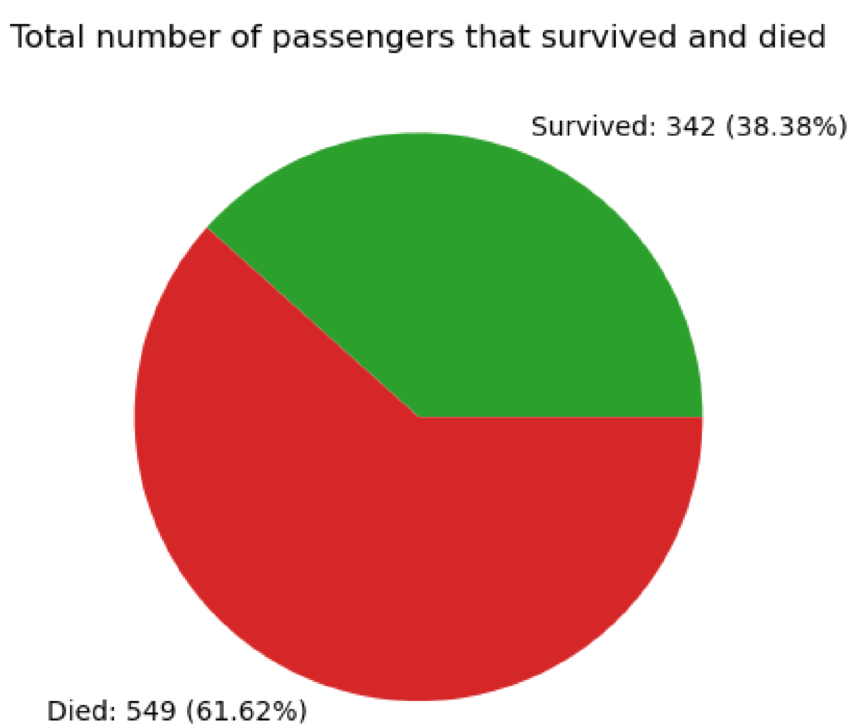
[4]:
```
PassengerId      0
Survived         0
Pclass           0
```

```
Name           0
Sex            0
Age          177
SibSp          0
Parch          0
Ticket         0
Fare           0
Cabin        687
Embarked       2
dtype: int64
```

1.1 Exploration

Here we wil visualize different aspects of the data to find promising features that can help us predict survival.

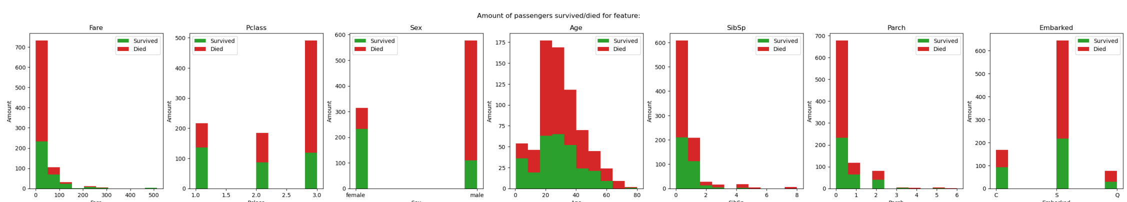
[5]:
```
count_survived = len(df[df['Survived'] == 1])
count_died = len(df[df['Survived'] == 0])
count_all = len(df)
plt.figure()
plt.pie([count_survived, count_died], colors=['tab:green', 'tab:red'],␣
  ↪labels=[f'Survived: {count_survived} ({100*count_survived/count_all:.2f}%)',␣
  ↪f'Died: {count_died} ({100*count_died/count_all:.2f}%)'])
plt.title('Total number of passengers that survived and died')
plt.show()
```



```python
[6]: def histogram(df: pd.DataFrame, feature: str, axes: Axes, title: typing.Optional[str] = None, xlabel: typing.Optional[str] = None, ylabel: typing.Optional[str] = None, dropna: bool = False) -> None:
    '''Plot stacked histogram of amount of passengers that survived/died.'''
    survived = df[df['Survived'] == 1][feature]
    died = df[df['Survived'] == 0][feature]
    if dropna:
        survived = survived.dropna()
        died = died.dropna()
    axes.hist([survived, died], stacked=True, color=['tab:green', 'tab:red'], label=['Survived', 'Died'])
    axes.legend()
    axes.set_xlabel(xlabel if xlabel else feature)
    axes.set_ylabel(ylabel if ylabel else 'Amount')
    axes.set_title(title if title else feature)
```

Some obvious (and easily analyzable) features to check are fare, passenger class, sex, age, number of siblings, number of parents, and embarked location.

```python
[7]: fig: Figure
axes: PltAxes
features = ['Fare', 'Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Embarked']
fig, axes = plt.subplots(1, len(features), figsize=(35, 5))
for i, feature in enumerate(features):
    histogram(df, feature, axes[i], dropna=True)
fig.suptitle('Amount of passengers survived/died for feature:')
plt.show()
```



There are a few easily observable heuristics that seem to be generally true. For example, male, 3rd class, and low-fare passengers were less likely to survive. However, there are not any glaringly obvious survival indicators we can notice by analyzing any single attribute.

This indicates that if there is a way to predict survival, it will be a mix of these features.

Before conducting a more intense analysis, we are going to explore the features we left out: name, ticket, and cabin.

Let's start with cabin. Upon first glance, we notice that most of the values for cabin are missing.

[8]:

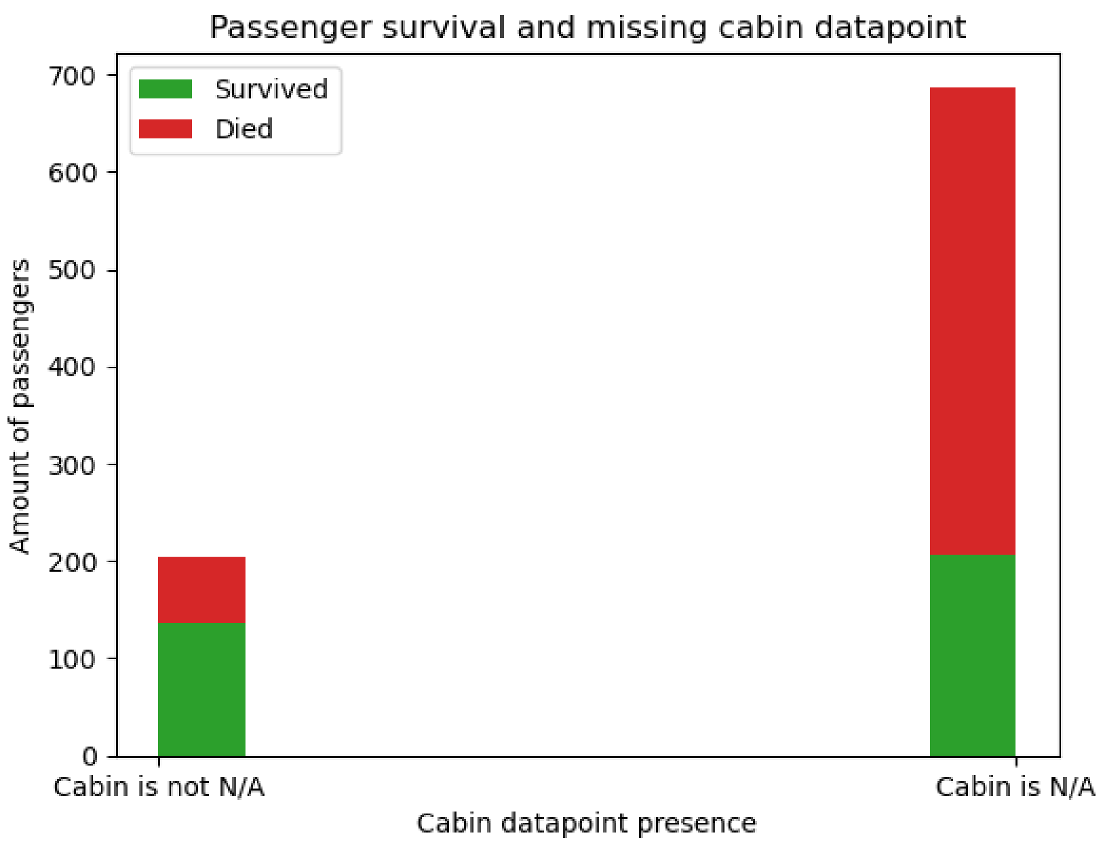
```
len(df['Cabin'].isna())
```

[8]: 891

Let's check if the absence of a cabin attribute affects the survival of a passenger.

[9]:

```
def cabin_na(x: float | str):
    if pd.isna(x):
        return 'Cabin is N/A'
    return 'Cabin is not N/A'
df['CabinNa'] = df['Cabin'].apply(cabin_na)
plt.figure()
histogram(df, 'CabinNa', plt.gca(), 'Passenger survival and missing cabin␣
↪datapoint', 'Cabin datapoint presence', 'Amount of passengers')
plt.show()
```



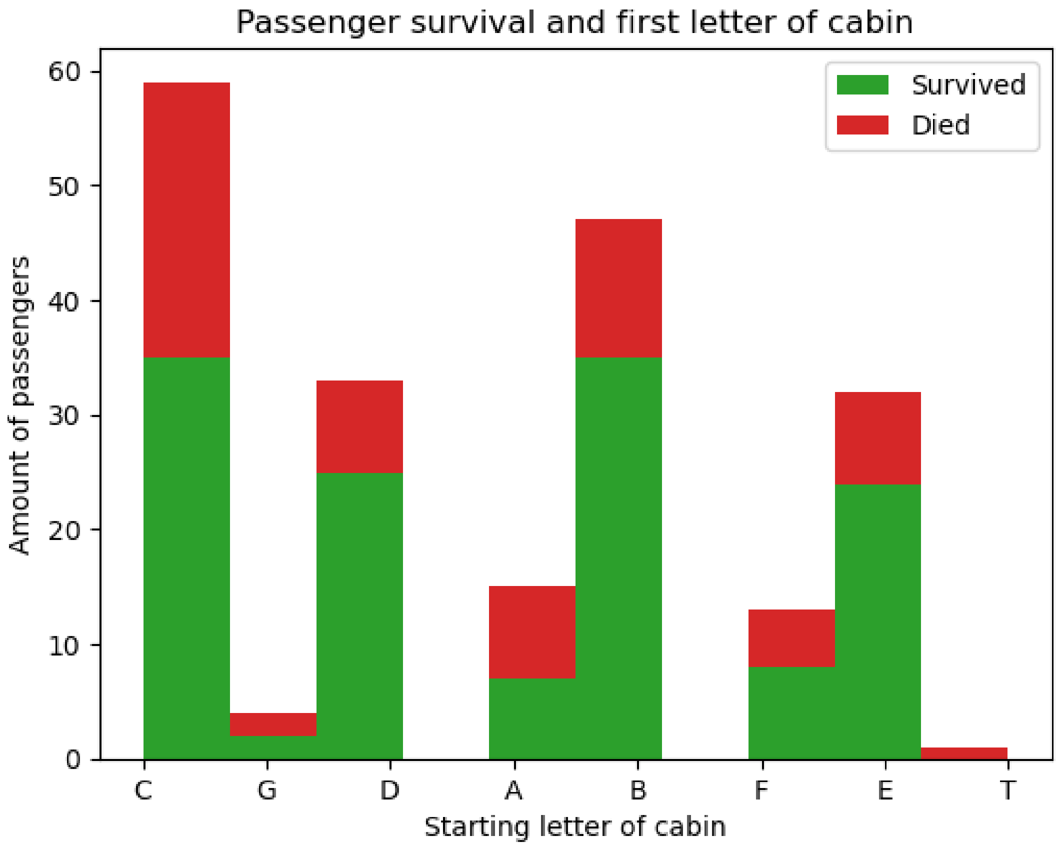
Finding the unique values of cabin, we see that they are alphanumeric strings. The letter at the beginning seems to correspond to a deck of the ship (https://www.encyclopedia-titanica.org/titanic-deckplans/), which could influence survival probability (rooms closer to the iceberg would be more affected, rooms deeper in the ship would have a farther distance to the life boats). Given that the crash occurred late in the night (https://www.thoughtco.com/

titanic-timeline-1779210), it is likely that many people would be in their rooms when the Titanic hit the iceberg.

[10]:
```python
df['Cabin'].unique()
```

[10]:
```
array([nan, 'C85', 'C123', 'E46', 'G6', 'C103', 'D56', 'A6',
       'C23 C25 C27', 'B78', 'D33', 'B30', 'C52', 'B28', 'C83', 'F33',
       'F G73', 'E31', 'A5', 'D10 D12', 'D26', 'C110', 'B58 B60', 'E101',
       'F E69', 'D47', 'B86', 'F2', 'C2', 'E33', 'B19', 'A7', 'C49', 'F4',
       'A32', 'B4', 'B80', 'A31', 'D36', 'D15', 'C93', 'C78', 'D35',
       'C87', 'B77', 'E67', 'B94', 'C125', 'C99', 'C118', 'D7', 'A19',
       'B49', 'D', 'C22 C26', 'C106', 'C65', 'E36', 'C54',
       'B57 B59 B63 B66', 'C7', 'E34', 'C32', 'B18', 'C124', 'C91', 'E40',
       'T', 'C128', 'D37', 'B35', 'E50', 'C82', 'B96 B98', 'E10', 'E44',
       'A34', 'C104', 'C111', 'C92', 'E38', 'D21', 'E12', 'E63', 'A14',
       'B37', 'C30', 'D20', 'B79', 'E25', 'D46', 'B73', 'C95', 'B38',
       'B39', 'B22', 'C86', 'C70', 'A16', 'C101', 'C68', 'A10', 'E68',
       'B41', 'A20', 'D19', 'D50', 'D9', 'A23', 'B50', 'A26', 'D48',
       'E58', 'C126', 'B71', 'B51 B53 B55', 'D49', 'B5', 'B20', 'F G63',
       'C62 C64', 'E24', 'C90', 'C45', 'E8', 'B101', 'D45', 'C46', 'D30',
       'E121', 'D11', 'E77', 'F38', 'B3', 'D6', 'B82 B84', 'D17', 'A36',
       'B102', 'B69', 'E49', 'C47', 'D28', 'E17', 'A24', 'C50', 'B42',
       'C148'], dtype=object)
```

[11]:
```python
def cabin_start_char(x: float | str):
    if pd.isna(x):
        return x
    return x[0]
df['CabinStartChar'] = df['Cabin'].apply(cabin_start_char)
plt.figure()
histogram(df, 'CabinStartChar', plt.gca(), 'Passenger survival and first letter␣
 ↪of cabin', 'Starting letter of cabin', 'Amount of passengers', dropna=True)
plt.show()
```



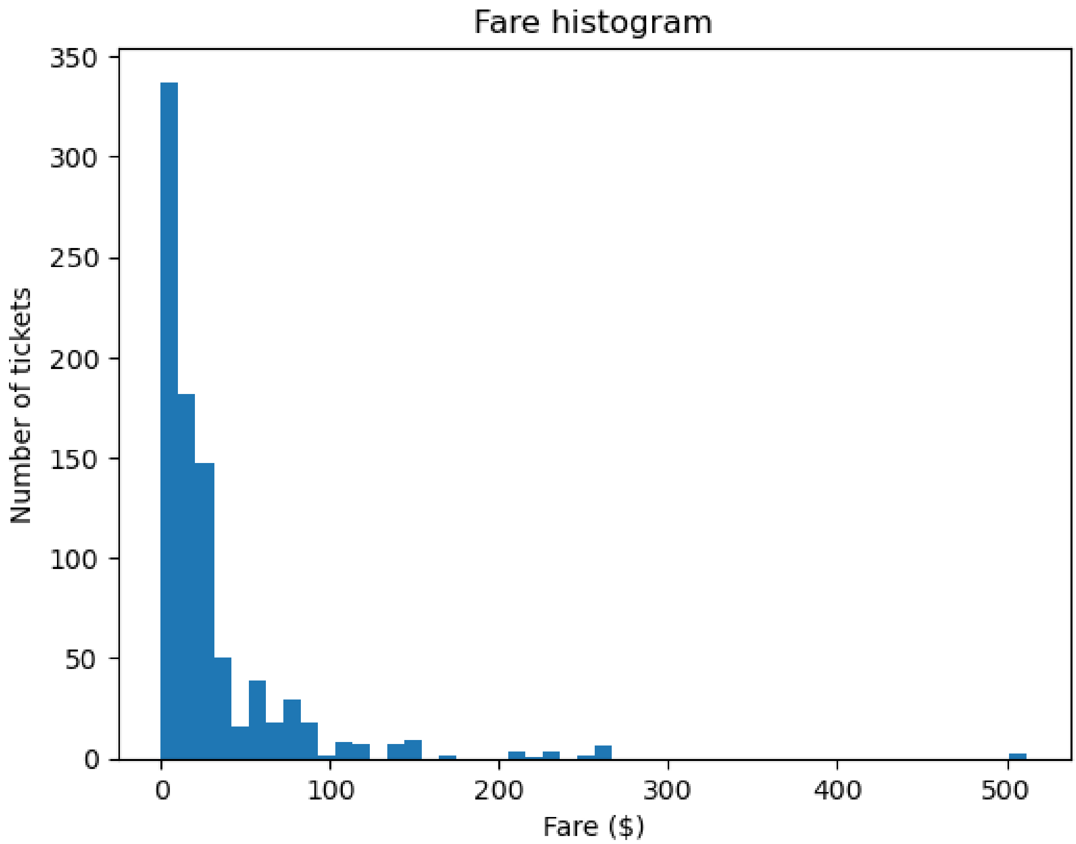
For now we will skip the ticket and name features. The ticket likely either random or corresponds to the passenger class, fare, and location of purchase. The name would require complex processing and NLP, so for simplicity we ignore it for now.

1.2 Preparing Data

Now that we know what features we will be working with, we can clean up the data to be processed more easily. We will likely be using a neural network, so we want numerical columns to be normalized and categorical columns one-hot encoded.

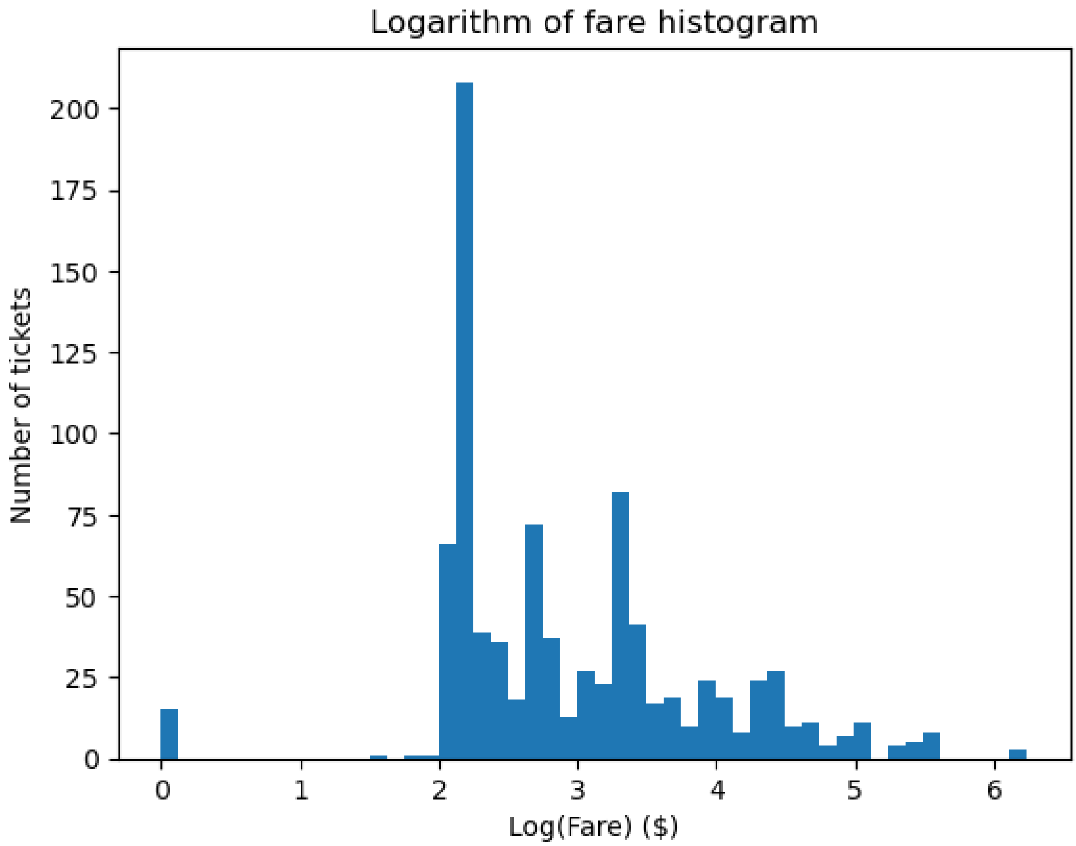
First, note that fare has a high variance and is very long-tailed (very few tickets are very expensive, most are much cheaper).

[12]:
```
plt.figure()
plt.hist(df['Fare'], bins=50)
plt.title('Fare histogram')
plt.xlabel('Fare ($)')
plt.ylabel('Number of tickets')
plt.show()
```



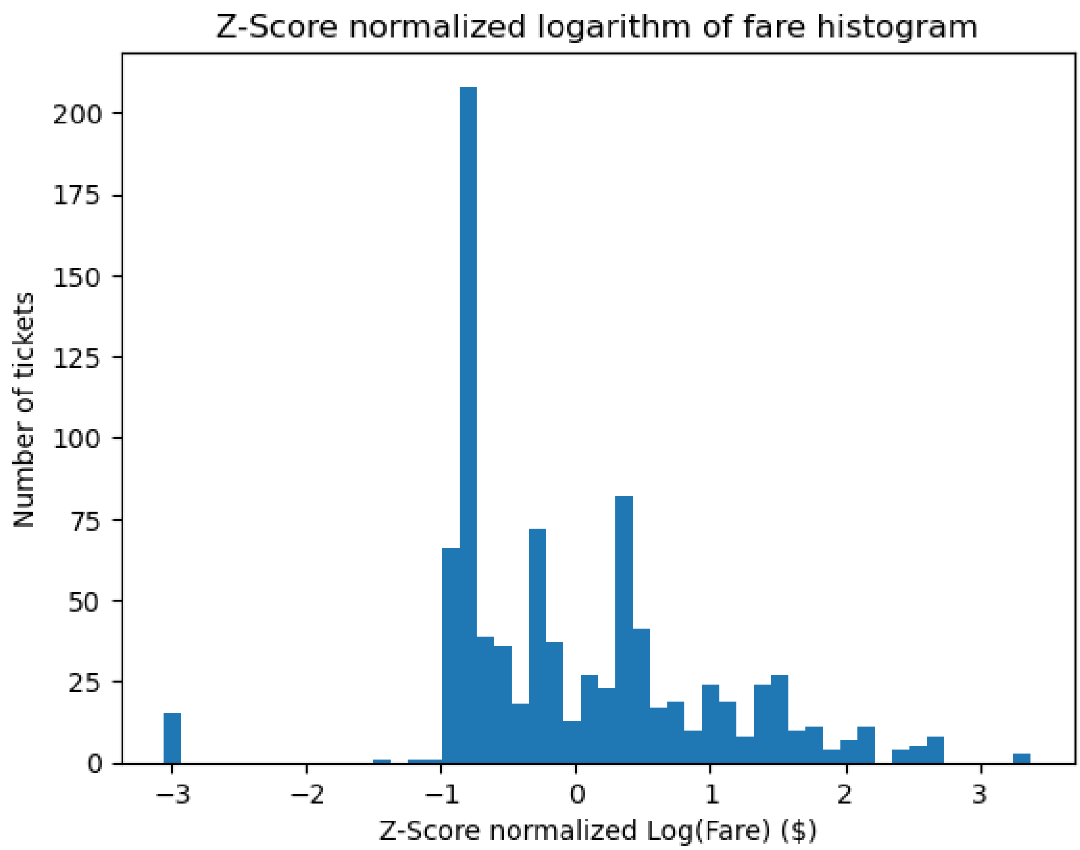
A good way to normalize this column would be to take the log.

[13]:
```
df['FareLog'] = np.log1p(df['Fare'])
plt.figure()
plt.hist(df['FareLog'], bins=50)
plt.title('Logarithm of fare histogram')
plt.xlabel('Log(Fare) ($)')
plt.ylabel('Number of tickets')
plt.show()
```



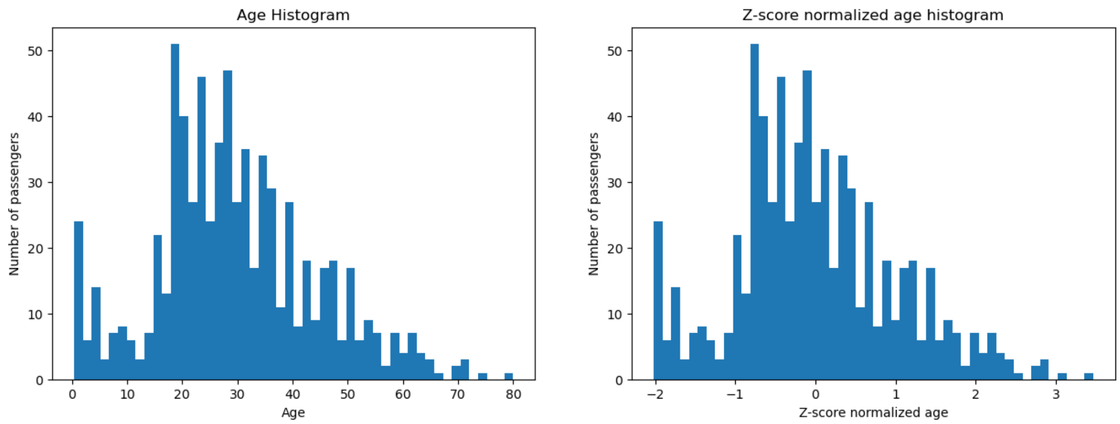
We are also going to Z-score normalize the log-normalized data because neural networks like when data has mean 0 and std 1.

[14]:
```
df['FareLogZ'] = (df['FareLog'] - df['FareLog'].mean()) / df['FareLog'].std()
plt.figure()
plt.hist(df['FareLogZ'], bins=50)
plt.title('Z-Score normalized logarithm of fare histogram')
plt.xlabel('Z-Score normalized Log(Fare) ($)')
plt.ylabel('Number of tickets')
plt.show()
```



The other numerical category is age, which already looks approximately normally distributed. I think we can get away with just Z-score normalization here.

[15]:
```
df['AgeZ'] = (df['Age'] - df['Age'].mean()) / df['Age'].std()
fig: Figure
axes: PltAxes
fig, axes = plt.subplots(1, 2, figsize=(15, 5))
axes[0].hist(df['Age'], bins=50)
axes[0].set_title('Age Histogram')
axes[0].set_xlabel('Age')
axes[0].set_ylabel('Number of passengers')
axes[1].hist(df['AgeZ'], bins=50)
axes[1].set_title('Z-score normalized age histogram')
axes[1].set_xlabel('Z-score normalized age')
axes[1].set_ylabel('Number of passengers')
plt.show()
```



The rest of the data is categorical. To turn this into neural-network friendly inputs we will use one-hot encoding.

[16]:
```
df = pd.get_dummies(df, columns=['Pclass', 'Sex', 'SibSp', 'Parch', 'Embarked',␣
↪'CabinStartChar'])
```

[17]:
```
df.head()
```

[17]:
```
   PassengerId  Survived                                               Name  \
0            1         0                            Braund, Mr. Owen Harris
1            2         1  Cumings, Mrs. John Bradley (Florence Briggs Th…
2            3         1                             Heikkinen, Miss. Laina
3            4         1       Futrelle, Mrs. Jacques Heath (Lily May Peel)
4            5         0                           Allen, Mr. William Henry

    Age            Ticket     Fare Cabin          CabinNa  FareLog  \
0  22.0         A/5 21171   7.2500   NaN      Cabin is N/A  2.110213
1  38.0          PC 17599  71.2833   C85  Cabin is not N/A  4.280593
2  26.0  STON/O2. 3101282   7.9250   NaN      Cabin is N/A  2.188856
3  35.0            113803  53.1000  C123  Cabin is not N/A  3.990834
4  35.0            373450   8.0500   NaN      Cabin is N/A  2.202765

   FareLogZ  …  Embarked_Q  Embarked_S  CabinStartChar_A  CabinStartChar_B  \
0 -0.879247  …       False        True             False             False
1  1.360456  …       False       False             False             False
2 -0.798092  …       False        True             False             False
3  1.061442  …       False        True             False             False
4 -0.783739  …       False        True             False             False

   CabinStartChar_C  CabinStartChar_D  CabinStartChar_E  CabinStartChar_F  \
0             False             False             False             False
1              True             False             False             False
```

```
2              False              False              False              False
3               True              False              False              False
4              False              False              False              False

   CabinStartChar_G  CabinStartChar_T
0              False             False
1              False             False
2              False             False
3              False             False
4              False             False

[5 rows x 41 columns]
```

[18]:

```
df.columns
```

[18]:

```
Index(['PassengerId', 'Survived', 'Name', 'Age', 'Ticket', 'Fare', 'Cabin',
       'CabinNa', 'FareLog', 'FareLogZ', 'AgeZ', 'Pclass_1', 'Pclass_2',
       'Pclass_3', 'Sex_female', 'Sex_male', 'SibSp_0', 'SibSp_1', 'SibSp_2',
       'SibSp_3', 'SibSp_4', 'SibSp_5', 'SibSp_8', 'Parch_0', 'Parch_1',
       'Parch_2', 'Parch_3', 'Parch_4', 'Parch_5', 'Parch_6', 'Embarked_C',
       'Embarked_Q', 'Embarked_S', 'CabinStartChar_A', 'CabinStartChar_B',
       'CabinStartChar_C', 'CabinStartChar_D', 'CabinStartChar_E',
       'CabinStartChar_F', 'CabinStartChar_G', 'CabinStartChar_T'],
      dtype='object')
```

We now have a 32-dimensional input vector and a 1-dimension output. We have 891 data points in the training set.

[19]:

```
df_in = df.drop(columns=['PassengerId', 'Name', 'Survived', 'Age', 'Ticket',␣
 ↪'Cabin', 'CabinNa', 'Fare', 'FareLog', 'Age'])
df_out = df[['Survived']]
print(f'{df_in.shape=}, {df_out.shape=}')
```

```
df_in.shape=(891, 32), df_out.shape=(891, 1)
```

To make this data cleaning process repeatable, we will collect the steps into a function so that we can perform it on the validation set.

[20]:

```
def clean_data(df: pd.DataFrame, drop_columns: bool = True, keep_id: bool =␣
 ↪False) -> pd.DataFrame:
    df_clean = df.copy()

    df_clean['CabinStartChar'] = df_clean['Cabin'].apply(cabin_start_char)

    df_clean['FareLog'] = np.log1p(df_clean['Fare'])
    df_clean['FareLogZ'] = (df_clean['FareLog'] - df_clean['FareLog'].mean()) /␣
 ↪df_clean['FareLog'].std()

    df_clean['Age'] = df_clean['Age'].fillna(df_clean['Age'].mean())
```

```
    df_clean['AgeZ'] = (df_clean['Age'] - df_clean['Age'].mean()) /␣
↪df_clean['Age'].std()

    non_dummy_columns = df_clean.columns

    df_clean = pd.get_dummies(df_clean, columns=['Pclass', 'Sex', 'SibSp',␣
↪'Parch', 'Embarked', 'CabinStartChar'])

    dummy_columns = df_clean.columns.difference(non_dummy_columns)
    feature_columns = ['FareLogZ', 'AgeZ'] + list(dummy_columns)

    if 'Survived' in df_clean.columns:
        feature_columns += ['Survived']

    df_clean[feature_columns] = df_clean[feature_columns].astype(float)
    if drop_columns:
        if keep_id:
            df_clean = df_clean[['PassengerId'] + feature_columns]
        else:
            df_clean = df_clean[feature_columns]
    else:
        if not keep_id:
            df_clean.drop(columns=['PassengerId'])
        else:
            pass

    return df_clean
```

We also define a dataset class to work well with PyTorch's data loaders.

[21]:
```
class PandasDataset(torch.utils.data.Dataset):

    def __init__(self, df: pd.DataFrame, features: typing.Sequence[str], label:␣
↪str, dtype: typing.Optional[torch.dtype] = torch.float32, device: typing.
↪Optional[torch.device] = None):
        self.X = torch.tensor(df[features].values, dtype=dtype, device=device)
        self.y = torch.tensor(df[label].values, dtype=dtype, device=device)

    def __len__(self):
        return len(self.y)

    def __getitem__(self, index):
        return self.X[index], self.y[index]
```

Now we load the data again, clean it, and split it into training and validation sets.

[22]:
```
df = pd.read_csv('train.csv')
df_clean = clean_data(df)
```

```
df_train, df_val = train_test_split(df_clean, test_size=0.2, random_state=42,␣
↪stratify=df['Survived'])
```

These are the features we have:

[23]:
```
label = 'Survived'
features = [feature for feature in df_clean.columns.tolist() if feature !=␣
↪label]
print(features)
```

```
['FareLogZ', 'AgeZ', 'CabinStartChar_A', 'CabinStartChar_B', 'CabinStartChar_C',
'CabinStartChar_D', 'CabinStartChar_E', 'CabinStartChar_F', 'CabinStartChar_G',
'CabinStartChar_T', 'Embarked_C', 'Embarked_Q', 'Embarked_S', 'Parch_0',
'Parch_1', 'Parch_2', 'Parch_3', 'Parch_4', 'Parch_5', 'Parch_6', 'Pclass_1',
'Pclass_2', 'Pclass_3', 'Sex_female', 'Sex_male', 'SibSp_0', 'SibSp_1',
'SibSp_2', 'SibSp_3', 'SibSp_4', 'SibSp_5', 'SibSp_8']
```

1.3 Defining the Neural Network

We have a 32-dimensional input vector, so the neural network will start with 33 inputs. We will treat this as a binary classification problem where we predict 1 (survived) or 0 (died).

We start with a small neural network with three blocks consisting of linear -> batch norm -> GELU -> dropout, and one final linear layer to project back to 1-dimension.

[24]:
```
model = nn.Sequential(
    collections.OrderedDict([
        ('lin1', nn.Linear(32, 48)),
        ('norm1', nn.BatchNorm1d(48)),
        ('gelu1', nn.GELU()),
        ('drop1', nn.Dropout(0.5)),

        ('lin2', nn.Linear(48, 64)),
        ('norm2', nn.BatchNorm1d(64)),
        ('gelu2', nn.GELU()),
        ('drop2', nn.Dropout(0.5)),

        ('lin3', nn.Linear(64, 96)),
        ('norm3', nn.BatchNorm1d(96)),
        ('gelu3', nn.GELU()),
        ('drop3', nn.Dropout(0.5)),

        ('lin4', nn.Linear(96, 1))
    ])
)
```

Hopefully we have a GPU so that training is faster.

[25]:
```python
if torch.cuda.is_available():
    device = 'cuda'
else:
    device = 'cpu'
print(f'Using device: {device}')
```

```
Using device: cuda
```

We turn our training and validation datasets into data loaders.

[26]:
```python
train_dataset = PandasDataset(df_train, features, label, device=device)
val_dataset = PandasDataset(df_val, features, label, device=device)

BATCH_SIZE = 16
train_loader = torch.utils.data.DataLoader(train_dataset,␣
 ↪batch_size=BATCH_SIZE, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=BATCH_SIZE,␣
 ↪shuffle=True)
```

We prepare for the training loop by moving the model to the device we're using (hopefully a GPU), define the optimizer used to update our weights and biases (AdamW), and specify that our loss function is binary cross entropy loss.

[27]:
```python
model.to(device)
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.9)
```

We also include a function to calculate the loss and accuracy over an entire dataset.

[28]:
```python
@torch.no_grad()
def loss_and_accuracy(split: str):
    X: Tensor
    y: Tensor
    loss: Tensor
    logits: Tensor
    total_loss = 0
    total_correct = 0
    total = 0
    if split == 'train':
        data_loader = train_loader
    elif split == 'val':
        data_loader = val_loader
    else:
        raise Exception(f'Invalid split {split}')
    for X, y in data_loader:
        logits = model(X)
        loss = criterion(logits.squeeze(), y)
        predictions = (torch.sigmoid(logits.squeeze()) >= 0.5).long()
```

```
        total_correct += (predictions == y).sum().item()
        total += y.shape[0]
        total_loss += loss.item() * y.shape[0]
    avg_loss = total_loss / total
    accuracy = total_correct / total
    return avg_loss, accuracy
```

1.4 Training Loop

This training loop trains the model with back-propagation and returns details about the training process (like loss and accuracy throughout the training run).

[29]:
```
def train(model: nn.Module, epochs: int):
    train_val_details = []
    losses = []
    learning_rates = []
    for epoch in tqdm.tqdm(range(epochs)):
        model.eval()
        train_loss, train_accuracy = loss_and_accuracy('train')
        val_loss, val_accuracy = loss_and_accuracy('val')
        train_val_details.append((train_loss, train_accuracy, val_loss,␣
↪val_accuracy))
        learning_rates.append(scheduler.get_last_lr()[0])
        model.train()
        X: Tensor
        y: Tensor
        loss: Tensor
        for X, y in train_loader:
            optimizer.zero_grad()
            logits = model(X)
            loss = criterion(logits.squeeze(), y)
            loss.backward()
            optimizer.step()
            losses.append(loss.item())
        scheduler.step()
        epoch += 1
        if epoch >= epochs:
            break
    df_train_val = pd.DataFrame(train_val_details, columns=['train loss',␣
↪'train accuracy', 'val loss', 'val accuracy'])
    losses = np.array(losses)
    learning_rates = np.array(learning_rates)
    return df_train_val, losses, learning_rates
```

We also define this plotting function to plot the details of the training process.

[30]:
```
def training_plot(df_training_details: pd.DataFrame, losses: np.ndarray,␣
↪learning_rates: np.ndarray, fig: Figure):
```

```
    axes = []
    gs = gridspec.GridSpec(3, 2, figure=fig)
    axes.append(fig.add_subplot(gs[0, :]))
    axes.append(fig.add_subplot(gs[1, 0]))
    axes.append(fig.add_subplot(gs[1, 1]))
    axes.append(fig.add_subplot(gs[2, 0]))
    axes.append(fig.add_subplot(gs[2, 1]))

    axes[0].scatter(np.arange(len(losses)), losses, alpha=0.5)
    axes[0].set_title('Loss over training')
    axes[0].set_xlabel('Iteration')
    axes[0].set_ylabel('Loss')
    axes[0].grid(True)

    axes[1].scatter(np.arange(df_training_details['train loss'].shape[0]), df_training_details['train loss'], label='training loss', alpha=0.5)
    axes[1].scatter(np.arange(df_training_details['val loss'].shape[0]), df_training_details['val loss'], label='validation loss', alpha=0.5)
    axes[1].set_title('Training and validation loss over training')
    axes[1].set_xlabel('Iteration')
    axes[1].set_ylabel('Loss')
    axes[1].grid(True)
    axes[1].legend()

    axes[2].scatter(np.arange(len(learning_rates)), learning_rates, alpha=0.5)
    axes[2].set_title('Learning rate over training')
    axes[2].set_xlabel('Epoch')
    axes[2].set_ylabel('Learning rate')
    axes[2].grid(True)

    axes[3].scatter(np.arange(df_training_details['train accuracy'].shape[0]), df_training_details['train accuracy'], label='training accuracy', alpha=0.5)
    axes[3].scatter(np.arange(df_training_details['val accuracy'].shape[0]), df_training_details['val accuracy'], label='validation accuracy', alpha=0.5)
    axes[3].set_title('Training and validation accuracy over training')
    axes[3].set_xlabel('Iteration')
    axes[3].set_ylabel('Accuracy')
    axes[3].grid(True)
    axes[3].legend()

    axes[4].scatter(np.arange(df_training_details['train accuracy'].shape[0]), df_training_details['train accuracy'], label='training accuracy', alpha=0.5)
    axes[4].scatter(np.arange(df_training_details['val accuracy'].shape[0]), df_training_details['val accuracy'], label='validation accuracy', alpha=0.5)
    axes[4].set_title('Training and validation accuracy over training with y-axis from 0 to 1')
```

```
    axes[4].set_xlabel('Iteration'); axes[3].set_ylabel('Accuracy')
    axes[4].grid(True)
    axes[4].legend()
    axes[4].set_ylim([0, 1])
```

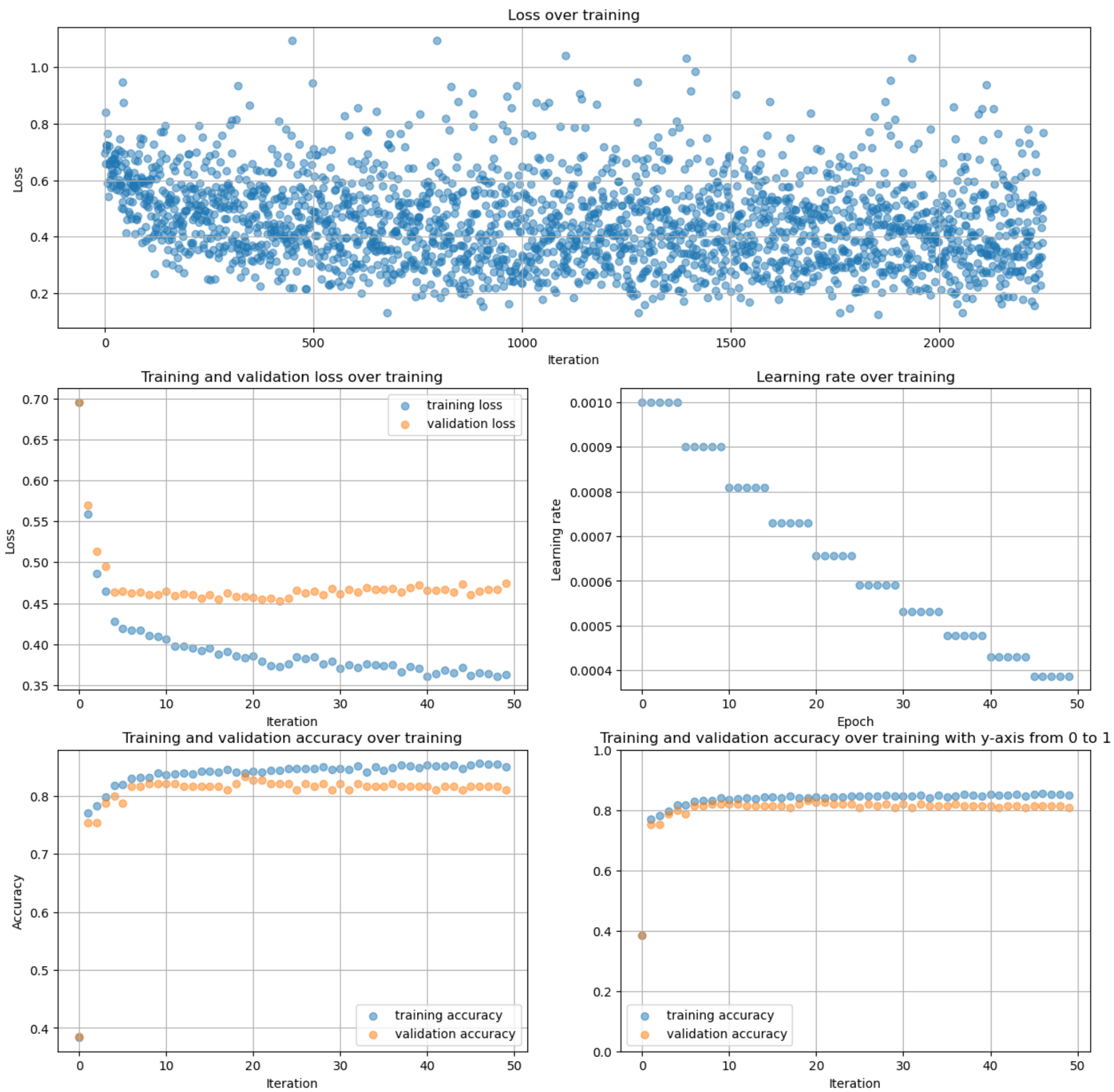
Here we train the model for 15 epochs.

[31]:
```
df_training_details, losses, learning_rates = train(model, epochs=50)
```

```
 98%|     | 49/50 [00:02<00:00, 17.21it/s]
```

The graphs below show the loss and accuracy throughout the 15 epochs, as well as the learning rate as it decays. Note that the model seems to peak around 80% accuracy on the validation set.

[32]:
```
fig: Figure
axes: PltAxes
fig = plt.figure(figsize=(15, 15))
training_plot(df_training_details, losses, learning_rates, fig)
plt.show()
```



1.4.1 Using a Larger Neural Network

The larger model has seven blocks of linear -> batch norm -> GELU -> dropout, then a final linear layer. The largest layer in this model has 512 neurons, up from 96 neurons in the largest layer in the smaller model.

[33]:
```
big_model = nn.Sequential(
    collections.OrderedDict([
        ('lin1', nn.Linear(32, 48)),
        ('norm1', nn.BatchNorm1d(48)),
        ('gelu1', nn.GELU()),
        ('drop1', nn.Dropout(0.5)),

        ('lin2', nn.Linear(48, 64)),
```

```
        ('norm2', nn.BatchNorm1d(64)),
        ('gelu2', nn.GELU()),
        ('drop2', nn.Dropout(0.5)),

        ('lin3', nn.Linear(64, 96)),
        ('norm3', nn.BatchNorm1d(96)),
        ('gelu3', nn.GELU()),
        ('drop3', nn.Dropout(0.5)),

        ('lin4', nn.Linear(96, 128)),
        ('norm4', nn.BatchNorm1d(128)),
        ('gelu4', nn.GELU()),
        ('drop4', nn.Dropout(0.5)),

        ('lin5', nn.Linear(128, 256)),
        ('norm5', nn.BatchNorm1d(256)),
        ('gelu5', nn.GELU()),
        ('drop5', nn.Dropout(0.5)),

        ('lin6', nn.Linear(256, 512)),
        ('norm6', nn.BatchNorm1d(512)),
        ('gelu6', nn.GELU()),
        ('drop6', nn.Dropout(0.5)),

        ('lin7', nn.Linear(512, 128)),
        ('norm7', nn.BatchNorm1d(128)),
        ('gelu7', nn.GELU()),
        ('drop7', nn.Dropout(0.5)),

        ('lin8', nn.Linear(128, 1))
    ])
)
```

To better accommodate the larger network, we set the learning rate scheduler decay rate to 0.99 so that we can train for more epochs.

[34]:
```
big_model.to(device)
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.AdamW(big_model.parameters(), lr=1e-3)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.99)
```

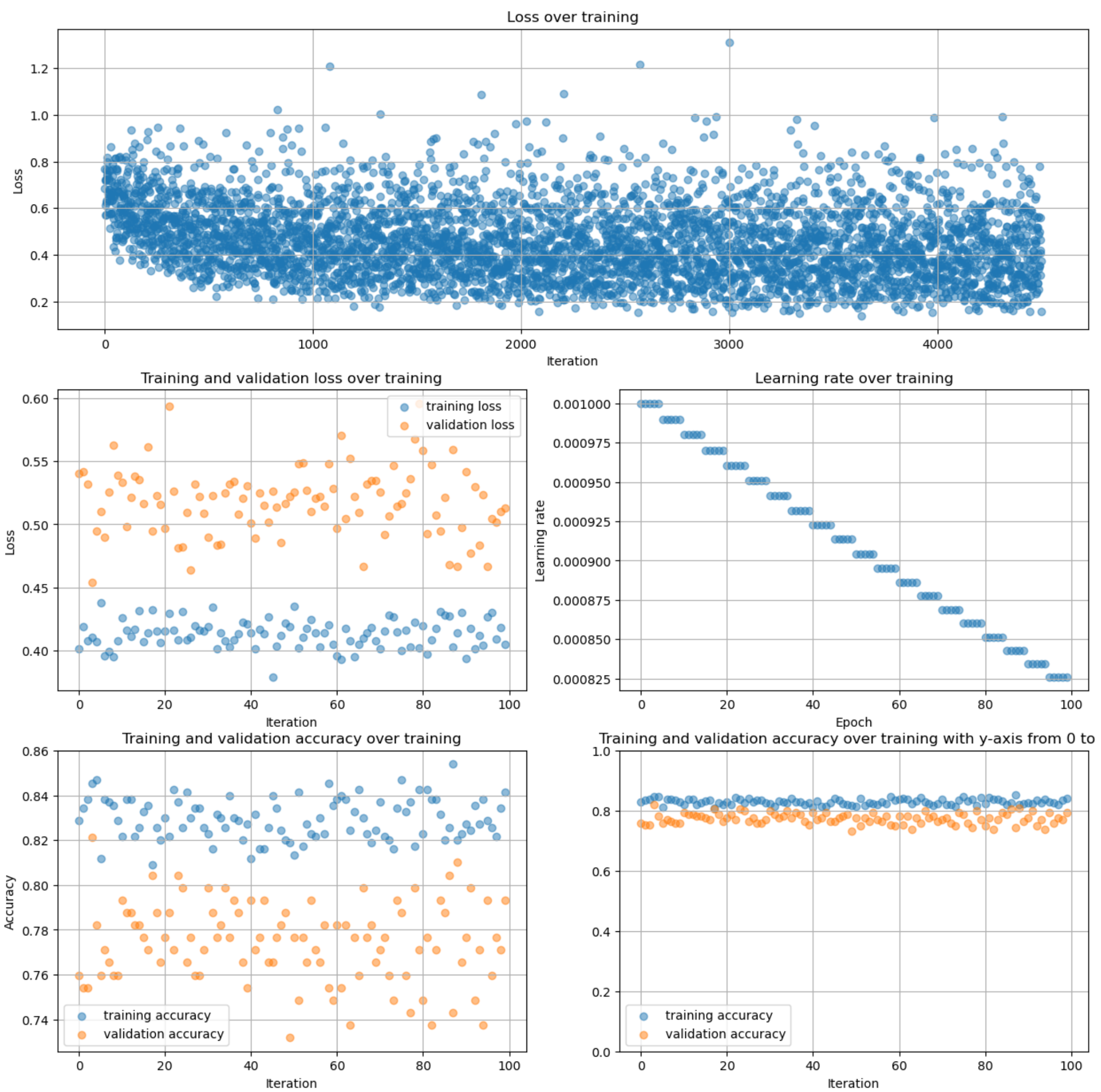
Here we train the larger network.

[35]:
```
df_training_details, losses, learning_rates = train(big_model, epochs=100)
```

```
 99%|      | 99/100 [00:08<00:00, 12.34it/s]
```

Note that the larger model performs almost identically to the small model, indicating that to achieve better performance we would need to engineer better features than what we have.

[36]:
```
fig: Figure
axes: PltAxes
fig = plt.figure(figsize=(15, 15))
training_plot(df_training_details, losses, learning_rates, fig)
plt.show()
```



1.5 Saving the Model and Generating the Competition Submission

We will save the smaller model (and use it for prediction) since it's performance is on par with the large model.

[37]:
```
torch.save(model.state_dict(), './model/model.pth')
print('Model saved')
```

```
Model saved
```

We load the competition dataset and clean it.

[38]:
```
df_submission = pd.read_csv('test.csv')
df_submission = clean_data(df_submission, keep_id=True)
df_submission.head()
```

[38]:
```
   PassengerId  FareLogZ      AgeZ  CabinStartChar_A  CabinStartChar_B  \
0          892 -0.865727  0.334592               0.0               0.0
1          893 -0.967611  1.323944               0.0               0.0
2          894 -0.668402  2.511166               0.0               0.0
3          895 -0.772558 -0.259019               0.0               0.0
4          896 -0.443455 -0.654760               0.0               0.0

   CabinStartChar_C  CabinStartChar_D  CabinStartChar_E  CabinStartChar_F  \
0               0.0               0.0               0.0               0.0
1               0.0               0.0               0.0               0.0
2               0.0               0.0               0.0               0.0
3               0.0               0.0               0.0               0.0
4               0.0               0.0               0.0               0.0

   CabinStartChar_G  …  Pclass_3  Sex_female  Sex_male  SibSp_0  SibSp_1  \
0               0.0  …       1.0         0.0       1.0      1.0      0.0
1               0.0  …       1.0         1.0       0.0      0.0      1.0
2               0.0  …       0.0         0.0       1.0      1.0      0.0
3               0.0  …       1.0         0.0       1.0      1.0      0.0
4               0.0  …       1.0         1.0       0.0      0.0      1.0

   SibSp_2  SibSp_3  SibSp_4  SibSp_5  SibSp_8
0      0.0      0.0      0.0      0.0      0.0
1      0.0      0.0      0.0      0.0      0.0
2      0.0      0.0      0.0      0.0      0.0
3      0.0      0.0      0.0      0.0      0.0
4      0.0      0.0      0.0      0.0      0.0

[5 rows x 33 columns]
```

Initially, we predict all passengers to have died, then we change it to survived on an individual basis according to the model's output.

[39]:
```
submission_features = list(df_submission.columns)
submission_features.remove('PassengerId')
df_submission['Survived'] = 0
```

Here is where we run inference and find passengers that are likely to survive (according to the model).

[40]:

```
model.eval()
with torch.no_grad():
    for idx, row in df_submission.iterrows():
        X = torch.tensor(row[submission_features].values, dtype=torch.float32,␣
↪device=device)
        logits = model(X.unsqueeze(0))
        predictions = (torch.sigmoid(logits.squeeze()) >= 0.5).long()
        if predictions.item() == 1:
            df_submission.loc[idx, 'Survived'] = 1
```

Export only the ID and survived columns to CSV.

[42]:

```
df_submission = df_submission[['PassengerId', 'Survived']]
df_submission.to_csv('./submission.csv', index=False)
```