Homework2-problem2

February 19, 2026

[89]:
```
# disclaimer: all PLOTTING functions/code are not written by me (but by AI), I hope thats alright (if not, let me know and we can work something out!)
# everything else, unless said otherwise, is written by me (or copied from the hands-on tutorials / homework notebooks from Duarte)
```

1 Problem 2, Parts A-B: Boosted Decision Tree

In this Jupyter notebook, we will train a boosted decision tree on the MiniBooNE dataset.

Use this notebook to write your code for problem 1 parts A-B by filling in the sections marked `# TODO` and running all cells.

[ ]:
```
# copied from plotting.py but added ylims for this notebook
def plot_model_history(history):
    """Plot the training and validation history for a TensorFlow network"""

    # Extract loss and accuracy
    loss = history.history["loss"]
    val_loss = history.history["val_loss"]
    acc = history.history["accuracy"]
    val_acc = history.history["val_accuracy"]
    n_epochs = len(loss)

    fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))
    ax[0].plot(np.arange(n_epochs), loss, label="Training")
    ax[0].plot(np.arange(n_epochs), val_loss, label="Validation")
    ax[0].set_ylim(0, 0.5)
    ax[0].legend()
    ax[0].set_xlabel("Epoch")
    ax[0].set_ylabel("Loss")

    ax[1].plot(np.arange(n_epochs), acc, label="Training")
    ax[1].plot(np.arange(n_epochs), val_acc, label="Validation")
    ax[1].set_ylim(.7, 1)
    ax[1].legend()
    ax[1].set_xlabel("Epoch")
    ax[1].set_ylabel("Accuracy")
```

[ ]:
```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import numpy as np
import math
import matplotlib.pyplot as plt

data = fetch_openml("miniboone", parser="auto", version=1)
X, y = data["data"].values, (data["target"].values == "True").astype(float)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42, shuffle=True)

# print dimensions of the datasets
print("X_train shape: ", X_train.shape)
print("y_train shape: ", y_train.shape)
print("X_test shape: ", X_test.shape)
print("y_test shape: ", y_test.shape)

# filter out rows containing the -999 sentinel in ANY feature
train_mask = (X_train != -999).all(axis=1)
test_mask = (X_test != -999).all(axis=1)

X_train_filtered = X_train[train_mask]
X_test_filtered = X_test[test_mask]
y_train_filtered = y_train[train_mask]
y_test_filtered = y_test[test_mask]

# print the new dimensions of the filtered datasets
print("X_train_filtered shape: ", X_train_filtered.shape)
print("y_train_filtered shape: ", y_train_filtered.shape)
print("X_test_filtered shape: ", X_test_filtered.shape)
print("y_test_filtered shape: ", y_test_filtered.shape)
```

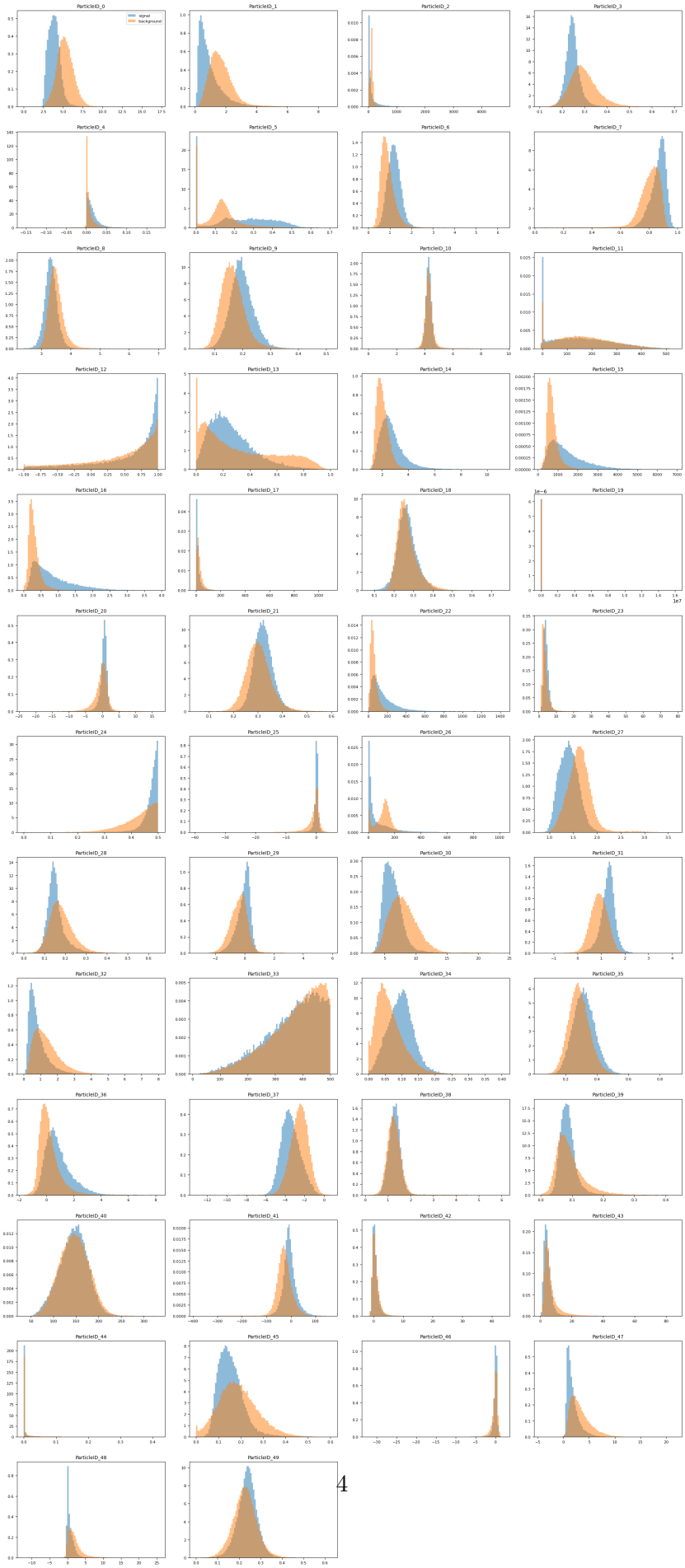
```
X_train shape:  (104051, 50)
y_train shape:  (104051,)
X_test shape:  (26013, 50)
y_test shape:  (26013,)
X_train_filtered shape:  (103675, 50)
y_train_filtered shape:  (103675,)
X_test_filtered shape:  (25921, 50)
y_test_filtered shape:  (25921,)
```

[ ]:
```
def plot_feature_distributions(X_data, y_labels, feature_names, ncols=4,␣
 ↪figsize_per_col=5, figsize_per_row=3.5, n_bins=100, title_fontsize=10,␣
 ↪tick_fontsize=8, legend_fontsize=8):

    y_arr = np.asarray(y_labels)
```

```
    signal_mask = y_arr == 1
    background_mask = y_arr == 0

    n_features = X_data.shape[1]
    nrows = math.ceil(n_features / ncols)

    fig, axs = plt.subplots(nrows, ncols, figsize=(figsize_per_col * ncols,␣
 ↪figsize_per_row * nrows))
    axs = np.array(axs).reshape(-1)

    for i in range(n_features):
        ax = axs[i]
        signal_vals = X_data[signal_mask, i]
        background_vals = X_data[background_mask, i]

        xmin = min(signal_vals.min(), background_vals.min())
        xmax = max(signal_vals.max(), background_vals.max())
        bins = np.linspace(xmin, xmax, n_bins)

        ax.hist(signal_vals, bins=bins, alpha=0.5, density=True, label="signal")
        ax.hist(background_vals, bins=bins, alpha=0.5, density=True,␣
 ↪label="background")
        ax.set_title(feature_names[i], fontsize=title_fontsize)
        ax.tick_params(axis="both", labelsize=tick_fontsize)

    # Hide any unused subplot axes
    for i in range(n_features, len(axs)):
        axs[i].axis("off")

    axs[0].legend(fontsize=legend_fontsize)
    plt.tight_layout()

    return fig
```

Let's analyze what we are working with

[54]:
```
feature_names = data.feature_names

# Plot feature distributions for unscaled data
fig = plot_feature_distributions(X_train_filtered, y_train_filtered,␣
 ↪feature_names)
plt.show()
```


ParticleID_0
signal
background
ParticleID_1
ParticleID_2
ParticleID_3
ParticleID_4
ParticleID_5
ParticleID_6
ParticleID_7
ParticleID_8
ParticleID_9
ParticleID_10
ParticleID_11
ParticleID_12
ParticleID_13
ParticleID_14
ParticleID_15
ParticleID_16
ParticleID_17
ParticleID_18
ParticleID_19
1e−6
1e7
ParticleID_20
ParticleID_21
ParticleID_22
ParticleID_23
ParticleID_24
ParticleID_25
ParticleID_26
ParticleID_27
ParticleID_28
ParticleID_29
ParticleID_30
ParticleID_31
ParticleID_32
ParticleID_33
ParticleID_34
ParticleID_35
ParticleID_36
ParticleID_37
ParticleID_38
ParticleID_39
ParticleID_40
ParticleID_41
ParticleID_42
ParticleID_43
ParticleID_44
ParticleID_45
ParticleID_46
ParticleID_47
ParticleID_48
ParticleID_49

We see we will have to manipulate these features, to get a good result in the end. Almost nothing is centered around 0 yet, and the standard deviation is quite big for some. This will be fixed when we scale the data. Also we have some features with extremely long tails or outliers. This will be fixed in the last part with further refinements.

Let's check that we did things correctly and analyze a bit further

[3]:
```
# sanity checks after filtering
print("Number of -999 values in X_train_filtered:", np.sum(X_train_filtered ==␣
 ↪-999))
print("Number of -999 values in X_test_filtered:", np.sum(X_test_filtered ==␣
 ↪-999))
print("Number of nan values in X_train_filtered:", np.sum(np.
 ↪isnan(X_train_filtered)))
print("Number of nan values in X_test_filtered:", np.sum(np.
 ↪isnan(X_test_filtered)))
print("Number of inf values in X_train_filtered:", np.sum(np.
 ↪isinf(X_train_filtered)))
print("Number of inf values in X_test_filtered:", np.sum(np.
 ↪isinf(X_test_filtered)))
print("All train entries finite:", np.isfinite(X_train_filtered).all())
print("All test entries finite:", np.isfinite(X_test_filtered).all())

# print all min and max values for each feature of X
for feature_idx in range(X_train_filtered.shape[1]):
        train_min = np.min(X_train_filtered[:, feature_idx])
        train_max = np.max(X_train_filtered[:, feature_idx])
        test_min = np.min(X_test_filtered[:, feature_idx])
        test_max = np.max(X_test_filtered[:, feature_idx])
        print(
                f"Feature {feature_idx:2d} | "
                f"train[min,max]=({train_min:.6g}, {train_max:.6g}) | "
                f"test[min,max]=({test_min:.6g}, {test_max:.6g})"
    )
```

```
Number of -999 values in X_train_filtered: 0
Number of -999 values in X_test_filtered: 0
Number of nan values in X_train_filtered: 0
Number of nan values in X_test_filtered: 0
Number of inf values in X_train_filtered: 0
Number of inf values in X_test_filtered: 0
All train entries finite: True
All test entries finite: True
Feature  0 | train[min,max]=(0, 17.0573) | test[min,max]=(0, 16.0604)
Feature  1 | train[min,max]=(0.0707482, 8.80282) | test[min,max]=(0.0596913,
7.66563)
```

```
Feature  2 | train[min,max]=(0.0123806, 4747.67) | test[min,max]=(0.0392624,
4266.45)
Feature  3 | train[min,max]=(0.104697, 0.704169) | test[min,max]=(0.109527,
0.736804)
Feature  4 | train[min,max]=(-0.156118, 0.179012) | test[min,max]=(-0.136905,
0.174658)
Feature  5 | train[min,max]=(0, 0.703859) | test[min,max]=(0, 0.604472)
Feature  6 | train[min,max]=(0, 6.24108) | test[min,max]=(0, 4.60401)
Feature  7 | train[min,max]=(0.041932, 0.982981) | test[min,max]=(0.0335104,
0.989713)
Feature  8 | train[min,max]=(2.39744, 6.99229) | test[min,max]=(2.37527,
7.17017)
Feature  9 | train[min,max]=(0.033432, 0.515115) | test[min,max]=(0.0453282,
0.524888)
Feature 10 | train[min,max]=(0, 9.55934) | test[min,max]=(2.88235, 8.74258)
Feature 11 | train[min,max]=(-6.74457, 537.262) | test[min,max]=(-6.96411,
529.575)
Feature 12 | train[min,max]=(-0.99999, 1) | test[min,max]=(-0.999999, 1)
Feature 13 | train[min,max]=(0, 1) | test[min,max]=(0, 1)
Feature 14 | train[min,max]=(0.974378, 11.1956) | test[min,max]=(0.959427,
11.2961)
Feature 15 | train[min,max]=(139.28, 6907.69) | test[min,max]=(135.279, 6376.98)
Feature 16 | train[min,max]=(0.00253297, 3.90153) | test[min,max]=(0.00378816,
3.74218)
Feature 17 | train[min,max]=(0.627859, 1097.13) | test[min,max]=(0.903767,
1015.59)
Feature 18 | train[min,max]=(0.0694006, 0.758865) | test[min,max]=(0.0574351,
0.739247)
Feature 19 | train[min,max]=(-17256.7, 1.60009e+07) | test[min,max]=(-1.90883,
6.21709)
Feature 20 | train[min,max]=(-23.5881, 16.7897) | test[min,max]=(-21.592,
22.4035)
Feature 21 | train[min,max]=(0.0486111, 0.596774) | test[min,max]=(0.0641822,
0.545064)
Feature 22 | train[min,max]=(0, 1428.59) | test[min,max]=(0, 1217.94)
Feature 23 | train[min,max]=(1.16433, 78.3633) | test[min,max]=(1.18354,
52.1195)
Feature 24 | train[min,max]=(0, 0.5) | test[min,max]=(0, 0.499992)
Feature 25 | train[min,max]=(-39.6302, 4.71662) | test[min,max]=(-37.1168,
4.56662)
Feature 26 | train[min,max]=(0, 1024.21) | test[min,max]=(0, 969.733)
Feature 27 | train[min,max]=(0.815627, 3.64689) | test[min,max]=(0.756133,
3.47754)
Feature 28 | train[min,max]=(0, 0.647185) | test[min,max]=(0.00490567, 0.652919)
Feature 29 | train[min,max]=(-3.32909, 5.86979) | test[min,max]=(-3.35161,
5.19292)
Feature 30 | train[min,max]=(2.39377, 23.9331) | test[min,max]=(2.63564,
22.8614)
```

```
Feature 31 | train[min,max]=(-1.55476, 4.12743) | test[min,max]=(-1.48048,
2.69916)
Feature 32 | train[min,max]=(0.00285458, 7.99446) | test[min,max]=(0.0917655,
6.82532)
Feature 33 | train[min,max]=(13.2357, 499.999) | test[min,max]=(6.33627,
499.988)
Feature 34 | train[min,max]=(0, 0.403012) | test[min,max]=(0, 0.327491)
Feature 35 | train[min,max]=(0.0429305, 0.897137) | test[min,max]=(0.0688675,
0.770593)
Feature 36 | train[min,max]=(-1.65624, 8.19201) | test[min,max]=(-1.58214,
6.69712)
Feature 37 | train[min,max]=(-13.1665, 0.669757) | test[min,max]=(-7.68483,
0.480739)
Feature 38 | train[min,max]=(0.144033, 6.05862) | test[min,max]=(0.199853,
5.19429)
Feature 39 | train[min,max]=(0, 0.430449) | test[min,max]=(0, 0.411972)
Feature 40 | train[min,max]=(33.8952, 331.925) | test[min,max]=(48.6812,
304.442)
Feature 41 | train[min,max]=(-387.617, 161.298) | test[min,max]=(-324.187,
150.445)
Feature 42 | train[min,max]=(-1.6865, 43.6512) | test[min,max]=(-1.55709,
34.9829)
Feature 43 | train[min,max]=(0.249186, 85.8231) | test[min,max]=(0.367699,
78.0003)
Feature 44 | train[min,max]=(0, 0.416873) | test[min,max]=(0, 0.446914)
Feature 45 | train[min,max]=(0, 0.600906) | test[min,max]=(0, 0.605667)
Feature 46 | train[min,max]=(-32.1098, 1.89319) | test[min,max]=(-27.5484,
1.93289)
Feature 47 | train[min,max]=(-4.42695, 21.7682) | test[min,max]=(-13.4922,
15.3895)
Feature 48 | train[min,max]=(-12.2505, 25.4233) | test[min,max]=(-15.9978,
22.063)
Feature 49 | train[min,max]=(0, 0.625484) | test[min,max]=(0, 0.580793)
```

[4]:
```
# ! pip install xgboost
```

[5]:
```
import xgboost as xgb
from xgboost import XGBClassifier
# help(XGBClassifier)
```

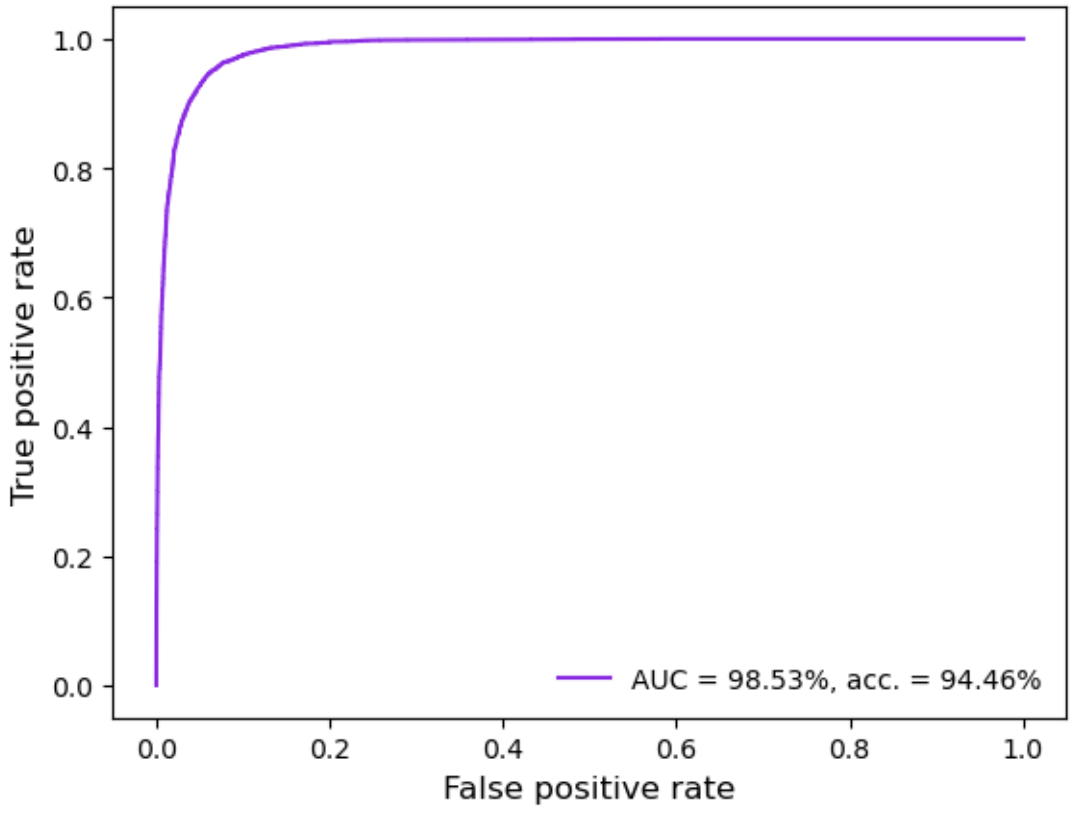
1.1 Problem 1A

Using the MiniBooNE dataset and XGBoost, train a boosted decision tree on the training dataet. Use the Scikit-learn API xgboost.XGBClassifier. For an initial choice of hyperparam- eters use 100 trees (n_estimators), maximum tree depth (max_depth) of 10, learning rate (learning_- rate) of 0.1, colsample_bytree of 0.8, and subsample of 0.8.

[6]:
```
bdt = XGBClassifier(n_estimators=100, max_depth=10, learning_rate=0.1,␣
↪colsample_bytree=0.8, subsample=0.8)

bdt.fit(X_train_filtered, y_train_filtered)

# retrieve predictions and take index [:, 1] corresponding to signal
preds_bdt = bdt.predict_proba(X_test_filtered)[:, 1]
```

Plot the receiver operating characteristic (ROC) curve using the testing dataset. What area under the curve (AUC) and accuracy do you achieve "out of the box"?

[7]:
```
from roc_helper import plot_roc
from sklearn.metrics import roc_auc_score

plot_roc(y_test_filtered, preds_bdt)

print("Out of the box AUC score: ", roc_auc_score(y_test_filtered, preds_bdt))
```



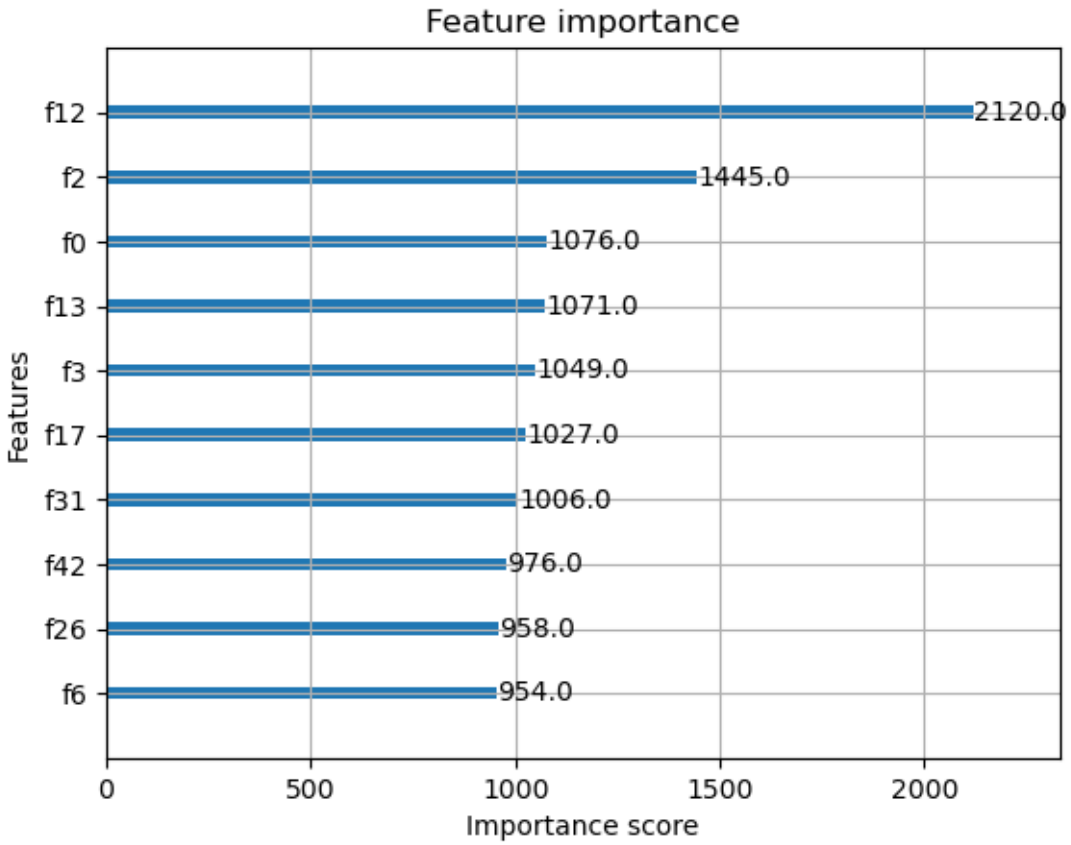

```
Out of the box AUC score:  0.9853333691641539
```

1.2 Problem B

Plot the F -score for all the 10 “most important” features using xgboost.plot_- importance. Which feature is the most important?

[8]:
```
xgb.plot_importance(bdt, max_num_features=10)
```

[8]:
```
<Axes: title={'center': 'Feature importance'}, xlabel='Importance score',
ylabel='Features'>
```



Plot this feature using the testing dataset in a 1D histogram separately for signal and background. For the histogram binning, use 100 bins from the minimum value of this feature to the maximum value of this feature in the testing dataset. What do you notice about this feature?

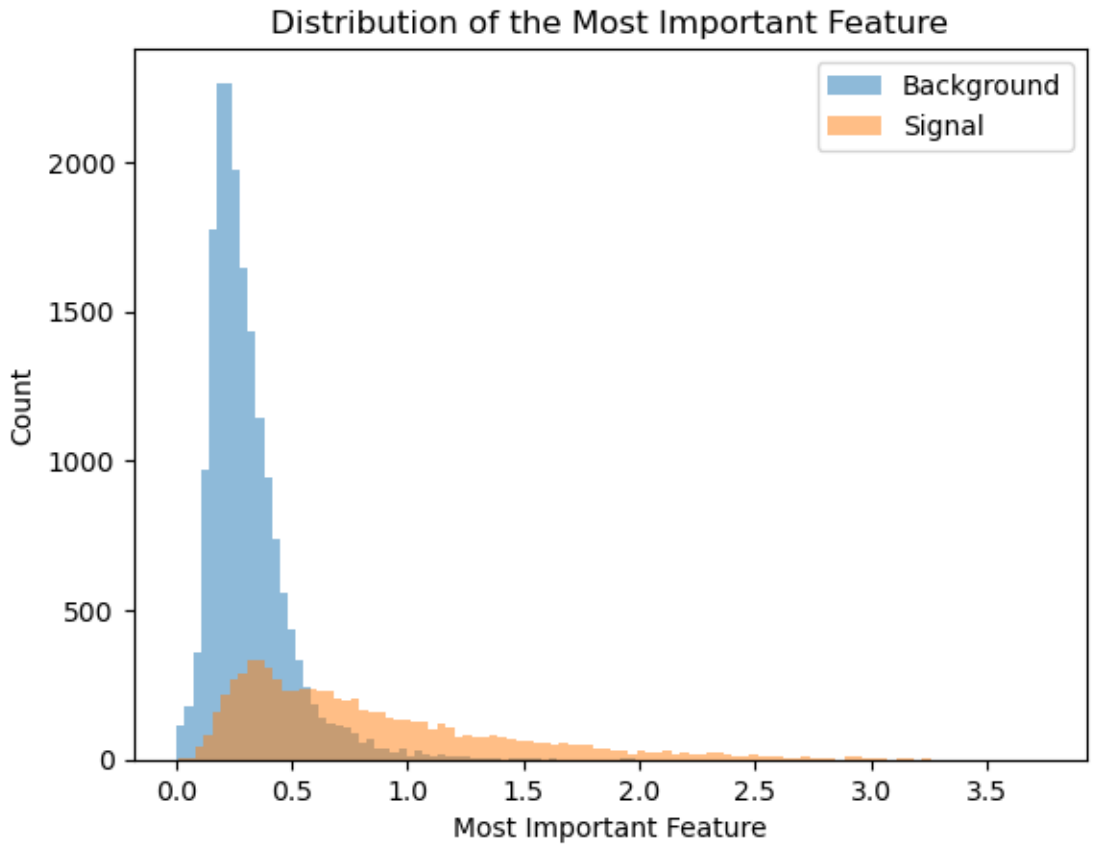
[9]:
```
print("feature importances:", bdt.feature_importances_)
mostImportantFeatureIndex = bdt.feature_importances_.argmax()
print("index of most important feature:", mostImportantFeatureIndex)
```

```
feature importances: [0.15744042 0.04005584 0.06105227 0.01814788 0.00976868
0.01615861
 0.01002657 0.00559988 0.01047468 0.01171287 0.00847025 0.01755949
 0.02532551 0.01314844 0.00589246 0.02176288 0.21118349 0.01247786
```

```
 0.00749157 0.00852642 0.01507393 0.00540459 0.01422502 0.01035366
 0.00716532 0.01012424 0.02838334 0.01274519 0.00867581 0.00912567
 0.00829527 0.05480712 0.00753342 0.00927151 0.00614356 0.00722038
 0.01005467 0.00703015 0.00856305 0.00719818 0.00736588 0.00731477
 0.00925392 0.00582307 0.01144512 0.0112385  0.00703905 0.00983341
 0.00538994 0.00562628]
index of most important feature: 16
```

[10]:
```python
import numpy as np
import matplotlib.pyplot as plt

feat_vals = X_test_filtered[:, mostImportantFeatureIndex]

xmin = np.nanmin(feat_vals)
xmax = np.nanmax(feat_vals)

signal_vals = X_test_filtered[y_test_filtered == 1, mostImportantFeatureIndex]
background_values = X_test_filtered[y_test_filtered == 0,␣
 ↪mostImportantFeatureIndex]

print("feature min, max:", xmin, xmax)
print("unique values count:", len(np.unique(feat_vals)))

# Create 100 bins between min and max (101 edges)
bins = 100

plt.hist(background_values, bins=bins, alpha=0.5, label="Background")
plt.hist(signal_vals, bins=bins, alpha=0.5, label="Signal")
plt.xlabel("Most Important Feature")
plt.ylabel("Count")
plt.title("Distribution of the Most Important Feature")
plt.legend()
plt.show()
```

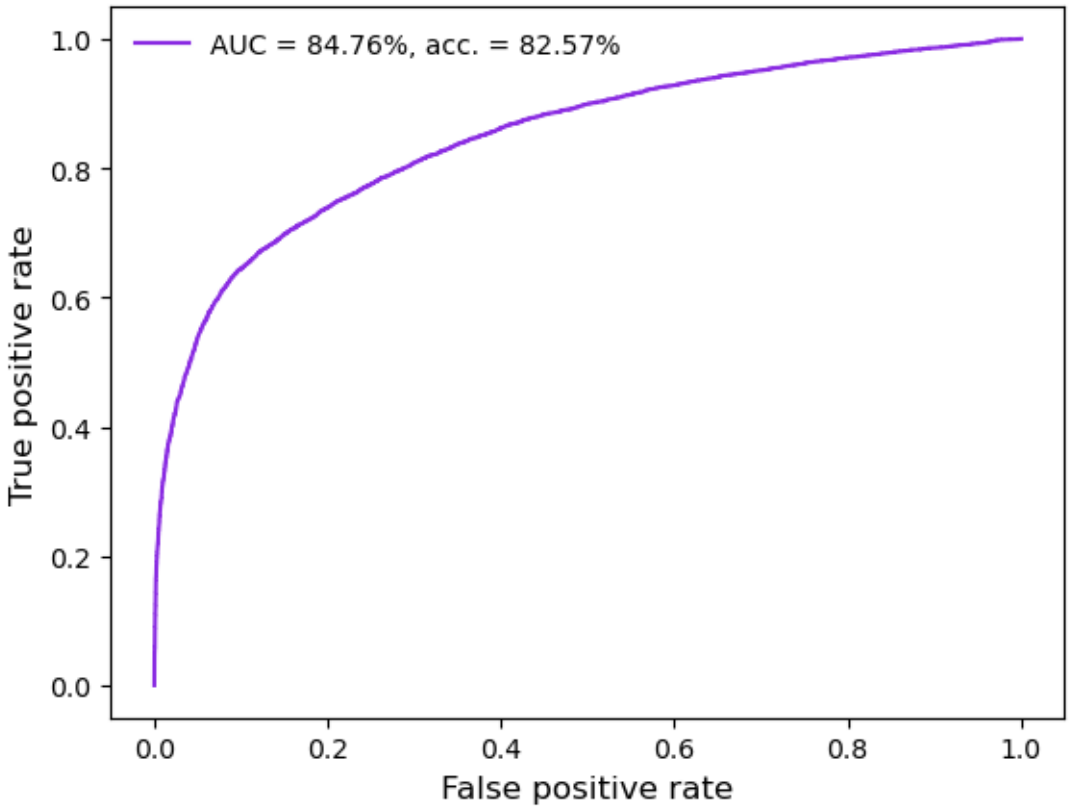
```
feature min, max: 0.00378816 3.74218
unique values count: 25396
```



The feature is a strong discriminator: the signal distribution is shifted to larger values than background and is much wider (longer tail). There is still overlap around the central region (medians are separated but not disjoint), so it's useful but not perfectly separable alone. The signal's heavy tail and positive skew suggest potential benefit from scaling or a transform (e.g. log or StandardScaler) before using in a neural net.

Lets see how well it performs with just this feature:

[11]:
```
from roc_helper import plot_roc

feat = X_test_filtered[:, mostImportantFeatureIndex]
auc_score = roc_auc_score(y_test_filtered, feat)
plot_roc(y_test_filtered, feat)
plt.close()
```



Already very good!

2 Problem 2, Parts C-E: Neural Network

In this Jupyter notebook, we will train a neural network on the MiniBooNE dataset.

Use this notebook to write your code for problem 1 parts C-E by filling in the sections marked `# TODO` and running all cells.

2.1 Problem C

Using the MiniBooNE dataset and the Keras Model API, train a neural network with 3 hidden layers each with 128 units and tanh activations. The final layer should have sigmoid activa- tion. Use the binary crossentropy loss function, the SGD optimizer with a learning rate of 0.01 (which is the default), and a batch size of 128. Train the model for 50 epochs.

[ ]:
```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Input

inputs = Input(shape=(X_train_filtered.shape[1],))
x = Dense(128, activation='tanh')(inputs)
x = Dense(128, activation='tanh')(x)
```

```
x = Dense(128, activation='tanh')(x)
outputs = Dense(1, activation='sigmoid')(x)
model_c = Model(inputs=inputs, outputs=outputs)

model_c.compile(optimizer='sgd', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

history_c = model_c.fit(X_train_filtered, y_train_filtered, epochs=50,␣
 ↪batch_size=128, validation_data=(X_test_filtered, y_test_filtered),verbose=0)

# retrieve predictions
preds_nn_c = model_c.predict(X_test_filtered)
```

```
Epoch 1/50
810/810             4s 4ms/step -
accuracy: 0.8375 - loss: 0.3703 - val_accuracy: 0.8464 - val_loss: 0.3540
Epoch 2/50
810/810             2s 3ms/step -
accuracy: 0.8435 - loss: 0.3559 - val_accuracy: 0.8559 - val_loss: 0.3351
Epoch 3/50
810/810             3s 3ms/step -
accuracy: 0.8443 - loss: 0.3604 - val_accuracy: 0.8459 - val_loss: 0.3713
Epoch 4/50
810/810             3s 3ms/step -
accuracy: 0.8127 - loss: 0.4260 - val_accuracy: 0.8245 - val_loss: 0.4095
Epoch 5/50
810/810             6s 4ms/step -
accuracy: 0.8268 - loss: 0.4013 - val_accuracy: 0.8525 - val_loss: 0.3568
Epoch 6/50
810/810             4s 3ms/step -
accuracy: 0.8268 - loss: 0.3936 - val_accuracy: 0.8244 - val_loss: 0.4475
Epoch 7/50
810/810             3s 4ms/step -
accuracy: 0.8277 - loss: 0.4025 - val_accuracy: 0.8073 - val_loss: 0.4125
Epoch 8/50
810/810             3s 3ms/step -
accuracy: 0.8268 - loss: 0.3953 - val_accuracy: 0.8514 - val_loss: 0.3477
Epoch 9/50
810/810             3s 3ms/step -
accuracy: 0.8343 - loss: 0.3910 - val_accuracy: 0.7376 - val_loss: 0.5389
Epoch 10/50
810/810             3s 4ms/step -
accuracy: 0.8375 - loss: 0.3750 - val_accuracy: 0.8429 - val_loss: 0.3473
Epoch 11/50
810/810             3s 3ms/step -
accuracy: 0.8450 - loss: 0.3554 - val_accuracy: 0.8643 - val_loss: 0.3250
Epoch 12/50
```

```
810/810            5s 4ms/step -
accuracy: 0.8482 - loss: 0.3520 - val_accuracy: 0.8680 - val_loss: 0.3508
Epoch 13/50
810/810            3s 4ms/step -
accuracy: 0.8442 - loss: 0.3549 - val_accuracy: 0.8333 - val_loss: 0.3577
Epoch 14/50
810/810            5s 4ms/step -
accuracy: 0.8504 - loss: 0.3473 - val_accuracy: 0.8598 - val_loss: 0.3291
Epoch 15/50
810/810            5s 4ms/step -
accuracy: 0.8445 - loss: 0.3489 - val_accuracy: 0.8265 - val_loss: 0.3394
Epoch 16/50
810/810            4s 5ms/step -
accuracy: 0.8448 - loss: 0.3532 - val_accuracy: 0.8688 - val_loss: 0.3632
Epoch 17/50
810/810            3s 4ms/step -
accuracy: 0.8444 - loss: 0.3465 - val_accuracy: 0.8534 - val_loss: 0.3245
Epoch 18/50
810/810            4s 5ms/step -
accuracy: 0.8483 - loss: 0.3398 - val_accuracy: 0.8649 - val_loss: 0.3284
Epoch 19/50
810/810            4s 5ms/step -
accuracy: 0.8494 - loss: 0.3419 - val_accuracy: 0.8672 - val_loss: 0.3447
Epoch 20/50
810/810            6s 6ms/step -
accuracy: 0.8456 - loss: 0.3425 - val_accuracy: 0.8433 - val_loss: 0.3387
Epoch 21/50
810/810            4s 4ms/step -
accuracy: 0.8538 - loss: 0.3512 - val_accuracy: 0.8321 - val_loss: 0.3460
Epoch 22/50
810/810            4s 3ms/step -
accuracy: 0.8530 - loss: 0.3412 - val_accuracy: 0.8151 - val_loss: 0.3747
Epoch 23/50
810/810            3s 4ms/step -
accuracy: 0.8526 - loss: 0.3358 - val_accuracy: 0.8739 - val_loss: 0.3309
Epoch 24/50
810/810            3s 4ms/step -
accuracy: 0.8625 - loss: 0.3335 - val_accuracy: 0.8561 - val_loss: 0.3364
Epoch 25/50
810/810            3s 3ms/step -
accuracy: 0.8536 - loss: 0.3360 - val_accuracy: 0.8277 - val_loss: 0.3654
Epoch 26/50
810/810            5s 3ms/step -
accuracy: 0.8548 - loss: 0.3295 - val_accuracy: 0.8615 - val_loss: 0.3346
Epoch 27/50
810/810            5s 3ms/step -
accuracy: 0.8544 - loss: 0.3333 - val_accuracy: 0.8615 - val_loss: 0.3223
Epoch 28/50
```

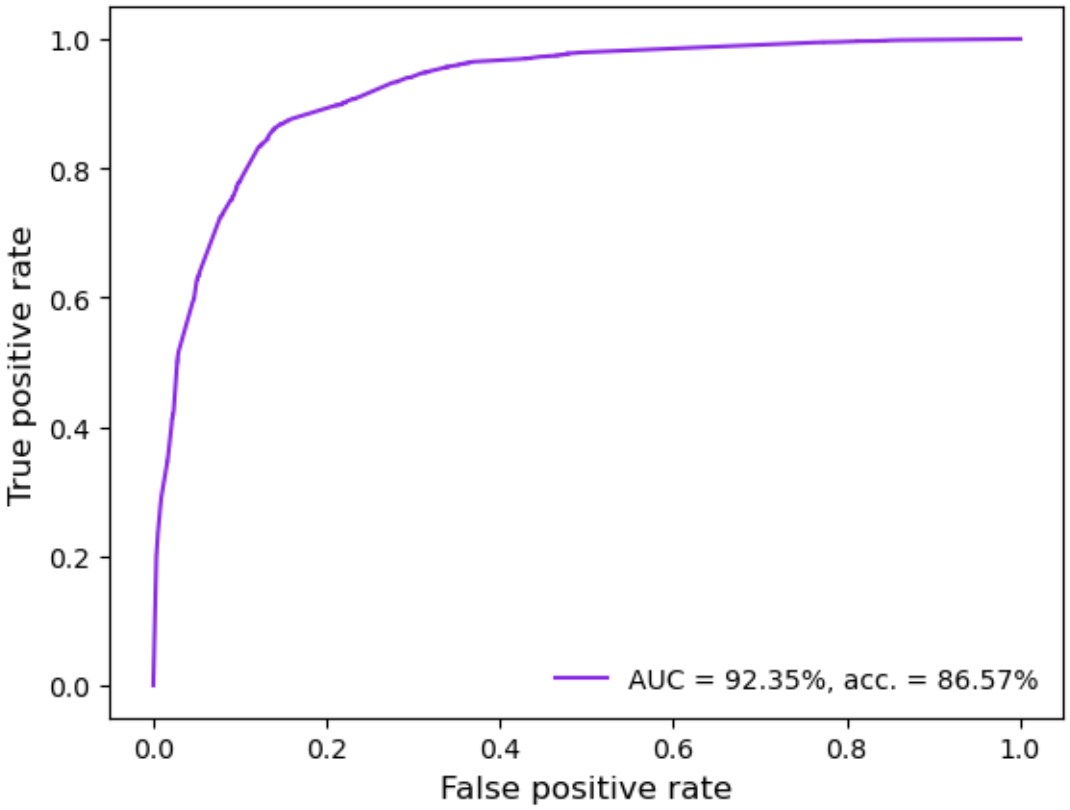
```
810/810            3s 3ms/step -
accuracy: 0.8579 - loss: 0.3275 - val_accuracy: 0.8500 - val_loss: 0.3250
Epoch 29/50
810/810            5s 3ms/step -
accuracy: 0.8562 - loss: 0.3329 - val_accuracy: 0.8277 - val_loss: 0.3562
Epoch 30/50
810/810            3s 3ms/step -
accuracy: 0.8540 - loss: 0.3393 - val_accuracy: 0.8662 - val_loss: 0.3348
Epoch 31/50
810/810            5s 3ms/step -
accuracy: 0.8563 - loss: 0.3339 - val_accuracy: 0.8406 - val_loss: 0.3460
Epoch 32/50
810/810            3s 3ms/step -
accuracy: 0.8520 - loss: 0.3335 - val_accuracy: 0.8438 - val_loss: 0.3416
Epoch 33/50
810/810            3s 4ms/step -
accuracy: 0.8611 - loss: 0.3242 - val_accuracy: 0.8691 - val_loss: 0.3125
Epoch 34/50
810/810            3s 3ms/step -
accuracy: 0.8618 - loss: 0.3231 - val_accuracy: 0.8709 - val_loss: 0.3123
Epoch 35/50
810/810            3s 4ms/step -
accuracy: 0.8582 - loss: 0.3227 - val_accuracy: 0.8623 - val_loss: 0.3162
Epoch 36/50
810/810            5s 3ms/step -
accuracy: 0.8589 - loss: 0.3305 - val_accuracy: 0.8685 - val_loss: 0.3191
Epoch 37/50
810/810            3s 3ms/step -
accuracy: 0.8565 - loss: 0.3294 - val_accuracy: 0.8720 - val_loss: 0.3285
Epoch 38/50
810/810            5s 3ms/step -
accuracy: 0.8654 - loss: 0.3225 - val_accuracy: 0.8718 - val_loss: 0.3124
Epoch 39/50
810/810            5s 3ms/step -
accuracy: 0.8620 - loss: 0.3204 - val_accuracy: 0.8647 - val_loss: 0.3211
Epoch 40/50
810/810            3s 4ms/step -
accuracy: 0.8596 - loss: 0.3275 - val_accuracy: 0.8701 - val_loss: 0.3356
Epoch 41/50
810/810            3s 4ms/step -
accuracy: 0.8600 - loss: 0.3262 - val_accuracy: 0.8426 - val_loss: 0.3536
Epoch 42/50
810/810            3s 3ms/step -
accuracy: 0.8576 - loss: 0.3280 - val_accuracy: 0.8569 - val_loss: 0.3217
Epoch 43/50
810/810            3s 3ms/step -
accuracy: 0.8643 - loss: 0.3213 - val_accuracy: 0.8701 - val_loss: 0.3136
Epoch 44/50
```

```
810/810             3s 3ms/step -
accuracy: 0.8638 - loss: 0.3202 - val_accuracy: 0.8588 - val_loss: 0.3228
Epoch 45/50
810/810             6s 4ms/step -
accuracy: 0.8619 - loss: 0.3264 - val_accuracy: 0.8769 - val_loss: 0.3207
Epoch 46/50
810/810             5s 3ms/step -
accuracy: 0.8627 - loss: 0.3370 - val_accuracy: 0.8580 - val_loss: 0.3378
Epoch 47/50
810/810             5s 3ms/step -
accuracy: 0.8651 - loss: 0.3238 - val_accuracy: 0.8612 - val_loss: 0.3248
Epoch 48/50
810/810             3s 3ms/step -
accuracy: 0.8653 - loss: 0.3189 - val_accuracy: 0.8645 - val_loss: 0.3205
Epoch 49/50
810/810             3s 3ms/step -
accuracy: 0.8676 - loss: 0.3191 - val_accuracy: 0.8680 - val_loss: 0.3144
Epoch 50/50
810/810             3s 3ms/step -
accuracy: 0.8670 - loss: 0.3167 - val_accuracy: 0.8657 - val_loss: 0.3130
811/811             1s 2ms/step
```

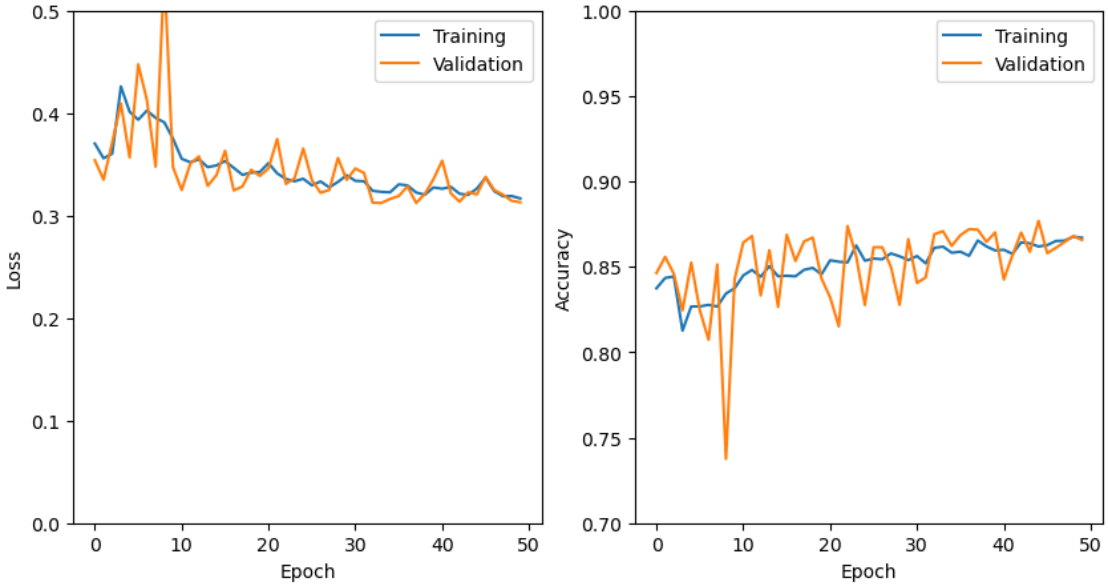
Plot the receiver operating characteristic (ROC) curve using the testing dataset. What AUC and accuracy do you achieve "out of the box"?

[13]:
```
print("Out of the box AUC score: ", roc_auc_score(y_test_filtered, preds_nn_c))
roc_c = plot_roc(y_test_filtered, preds_nn_c)
plt.close()
```

```
Out of the box AUC score:  0.9234507183830509
```



[14]:
```
plot_model_history(history_c)
```



This is not too bad but not good either. This is because tanh works best when the values are around 0 with a standard deviation of 1. Higher values » 1 or « 1 get mapped to ~1 so the activation cannot act on the values as well. This can be helped with scaling.

2.2 Problem D

Swap out the tanh activations for ReLU activations, while keeping everything else the same. Does the network train effectively? Why or why not?

[ ]:
```
inputs = Input(shape=(X_train_filtered.shape[1],))
x = Dense(128, activation='relu')(inputs)
x = Dense(128, activation='relu')(x)
x = Dense(128, activation='relu')(x)
outputs = Dense(1, activation='sigmoid')(x)
model_d = Model(inputs=inputs, outputs=outputs)

model_d.compile(optimizer='sgd', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

history_d = model_d.fit(X_train_filtered, y_train_filtered, epochs=50,␣
 ↪batch_size=128, validation_data=(X_test_filtered, y_test_filtered),verbose=0)

# retrieve predictions
preds_nn_d = model_d.predict(X_test_filtered)
```

```
Epoch 1/50
810/810             4s 3ms/step -
accuracy: 0.7173 - loss: 3.8723 - val_accuracy: 0.7163 - val_loss: 0.6006
Epoch 2/50
810/810             2s 3ms/step -
accuracy: 0.7190 - loss: 0.5933 - val_accuracy: 0.7163 - val_loss: 0.6003
Epoch 3/50
810/810             2s 3ms/step -
accuracy: 0.7190 - loss: 0.5933 - val_accuracy: 0.7163 - val_loss: 0.5990
Epoch 4/50
810/810             2s 3ms/step -
accuracy: 0.7190 - loss: 0.5931 - val_accuracy: 0.7163 - val_loss: 0.5994
Epoch 5/50
810/810             2s 2ms/step -
accuracy: 0.7190 - loss: 0.5933 - val_accuracy: 0.7163 - val_loss: 0.5993
Epoch 6/50
810/810             2s 3ms/step -
accuracy: 0.7190 - loss: 0.5930 - val_accuracy: 0.7163 - val_loss: 0.6000
Epoch 7/50
810/810             2s 2ms/step -
accuracy: 0.7190 - loss: 0.5927 - val_accuracy: 0.7163 - val_loss: 0.5998
Epoch 8/50
810/810             2s 2ms/step -
```

```
accuracy: 0.7190 - loss: 0.5926 - val_accuracy: 0.7163 - val_loss: 0.5998
Epoch 9/50
810/810             2s 2ms/step -
accuracy: 0.7190 - loss: 0.5922 - val_accuracy: 0.7163 - val_loss: 0.5985
Epoch 10/50
810/810             2s 2ms/step -
accuracy: 0.7190 - loss: 0.5927 - val_accuracy: 0.7163 - val_loss: 0.5981
Epoch 11/50
810/810             2s 2ms/step -
accuracy: 0.7190 - loss: 0.5915 - val_accuracy: 0.7163 - val_loss: 0.5973
Epoch 12/50
810/810             2s 2ms/step -
accuracy: 0.7190 - loss: 0.5928 - val_accuracy: 0.7163 - val_loss: 0.5967
Epoch 13/50
810/810             2s 3ms/step -
accuracy: 0.7190 - loss: 0.5929 - val_accuracy: 0.7163 - val_loss: 0.5972
Epoch 14/50
810/810             2s 2ms/step -
accuracy: 0.7190 - loss: 0.5924 - val_accuracy: 0.7163 - val_loss: 0.5972
Epoch 15/50
810/810             2s 2ms/step -
accuracy: 0.7185 - loss: 0.5927 - val_accuracy: 0.7163 - val_loss: 0.6005
Epoch 16/50
810/810             2s 3ms/step -
accuracy: 0.7189 - loss: 0.5746 - val_accuracy: 0.7163 - val_loss: 0.4475
Epoch 17/50
810/810             2s 2ms/step -
accuracy: 0.7374 - loss: 0.4828 - val_accuracy: 0.8418 - val_loss: 0.4085
Epoch 18/50
810/810             2s 2ms/step -
accuracy: 0.8238 - loss: 0.4149 - val_accuracy: 0.8460 - val_loss: 0.3848
Epoch 19/50
810/810             2s 2ms/step -
accuracy: 0.8425 - loss: 0.3841 - val_accuracy: 0.8453 - val_loss: 0.3723
Epoch 20/50
810/810             2s 3ms/step -
accuracy: 0.8430 - loss: 0.3732 - val_accuracy: 0.8475 - val_loss: 0.3657
Epoch 21/50
810/810             2s 2ms/step -
accuracy: 0.8442 - loss: 0.3673 - val_accuracy: 0.8495 - val_loss: 0.3610
Epoch 22/50
810/810             3s 2ms/step -
accuracy: 0.8458 - loss: 0.3631 - val_accuracy: 0.8366 - val_loss: 0.3792
Epoch 23/50
810/810             2s 2ms/step -
accuracy: 0.8458 - loss: 0.3603 - val_accuracy: 0.8474 - val_loss: 0.3686
Epoch 24/50
810/810             2s 2ms/step -
```

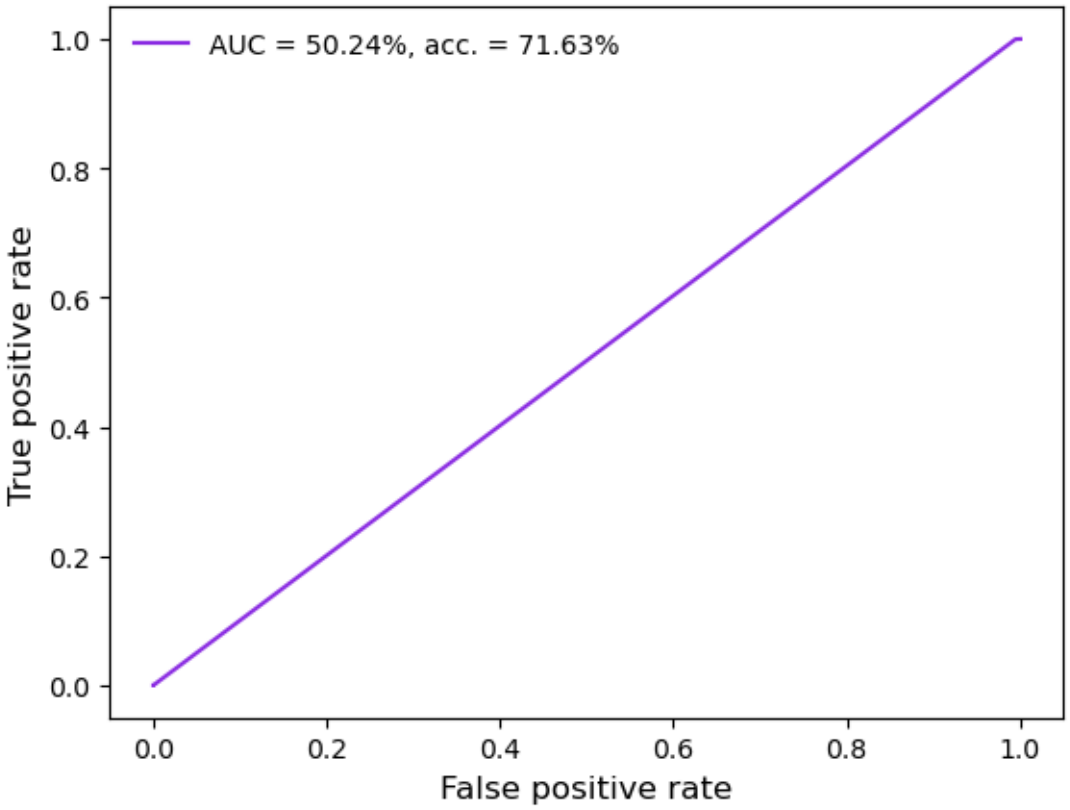
```
accuracy: 0.7325 - loss: 0.5661 - val_accuracy: 0.7161 - val_loss: 0.5970
Epoch 25/50
810/810             2s 2ms/step -
accuracy: 0.7177 - loss: 0.5984 - val_accuracy: 0.7136 - val_loss: 0.5982
Epoch 26/50
810/810             2s 2ms/step -
accuracy: 0.7189 - loss: 0.5936 - val_accuracy: 0.7162 - val_loss: 0.5967
Epoch 27/50
810/810             2s 2ms/step -
accuracy: 0.7189 - loss: 0.5935 - val_accuracy: 0.7161 - val_loss: 0.5970
Epoch 28/50
810/810             2s 3ms/step -
accuracy: 0.7189 - loss: 0.5935 - val_accuracy: 0.7162 - val_loss: 0.5973
Epoch 29/50
810/810             2s 2ms/step -
accuracy: 0.7188 - loss: 0.5936 - val_accuracy: 0.7163 - val_loss: 0.5977
Epoch 30/50
810/810             2s 2ms/step -
accuracy: 0.7188 - loss: 0.5936 - val_accuracy: 0.7158 - val_loss: 0.5988
Epoch 31/50
810/810             2s 3ms/step -
accuracy: 0.7187 - loss: 0.5935 - val_accuracy: 0.7163 - val_loss: 0.5973
Epoch 32/50
810/810             2s 2ms/step -
accuracy: 0.7189 - loss: 0.5934 - val_accuracy: 0.7163 - val_loss: 0.5973
Epoch 33/50
810/810             2s 3ms/step -
accuracy: 0.7189 - loss: 0.5934 - val_accuracy: 0.7162 - val_loss: 0.5972
Epoch 34/50
810/810             3s 3ms/step -
accuracy: 0.7189 - loss: 0.5934 - val_accuracy: 0.7162 - val_loss: 0.5977
Epoch 35/50
810/810             2s 3ms/step -
accuracy: 0.7189 - loss: 0.5933 - val_accuracy: 0.7163 - val_loss: 0.5977
Epoch 36/50
810/810             2s 2ms/step -
accuracy: 0.7189 - loss: 0.5935 - val_accuracy: 0.7163 - val_loss: 0.5977
Epoch 37/50
810/810             2s 2ms/step -
accuracy: 0.7189 - loss: 0.5933 - val_accuracy: 0.7163 - val_loss: 0.5981
Epoch 38/50
810/810             2s 3ms/step -
accuracy: 0.7190 - loss: 0.5934 - val_accuracy: 0.7163 - val_loss: 0.5982
Epoch 39/50
810/810             2s 2ms/step -
accuracy: 0.7189 - loss: 0.5935 - val_accuracy: 0.7163 - val_loss: 0.5983
Epoch 40/50
810/810             2s 2ms/step -
```

```
accuracy: 0.7189 - loss: 0.5932 - val_accuracy: 0.7163 - val_loss: 0.5984
Epoch 41/50
810/810             2s 3ms/step -
accuracy: 0.7189 - loss: 0.5930 - val_accuracy: 0.7163 - val_loss: 0.5986
Epoch 42/50
810/810             2s 3ms/step -
accuracy: 0.7190 - loss: 0.5931 - val_accuracy: 0.7163 - val_loss: 0.5986
Epoch 43/50
810/810             2s 3ms/step -
accuracy: 0.7190 - loss: 0.5929 - val_accuracy: 0.7163 - val_loss: 0.5999
Epoch 44/50
810/810             2s 3ms/step -
accuracy: 0.7190 - loss: 0.5929 - val_accuracy: 0.7163 - val_loss: 0.5991
Epoch 45/50
810/810             2s 3ms/step -
accuracy: 0.7190 - loss: 0.5934 - val_accuracy: 0.7162 - val_loss: 0.5990
Epoch 46/50
810/810             2s 2ms/step -
accuracy: 0.7189 - loss: 0.5927 - val_accuracy: 0.7163 - val_loss: 0.5990
Epoch 47/50
810/810             2s 2ms/step -
accuracy: 0.7190 - loss: 0.5919 - val_accuracy: 0.7163 - val_loss: 0.5983
Epoch 48/50
810/810             2s 2ms/step -
accuracy: 0.7190 - loss: 0.5927 - val_accuracy: 0.7163 - val_loss: 0.5987
Epoch 49/50
810/810             2s 2ms/step -
accuracy: 0.7186 - loss: 0.5921 - val_accuracy: 0.7161 - val_loss: 0.5988
Epoch 50/50
810/810             2s 2ms/step -
accuracy: 0.7189 - loss: 0.5925 - val_accuracy: 0.7163 - val_loss: 0.5987
811/811             1s 952us/step
```

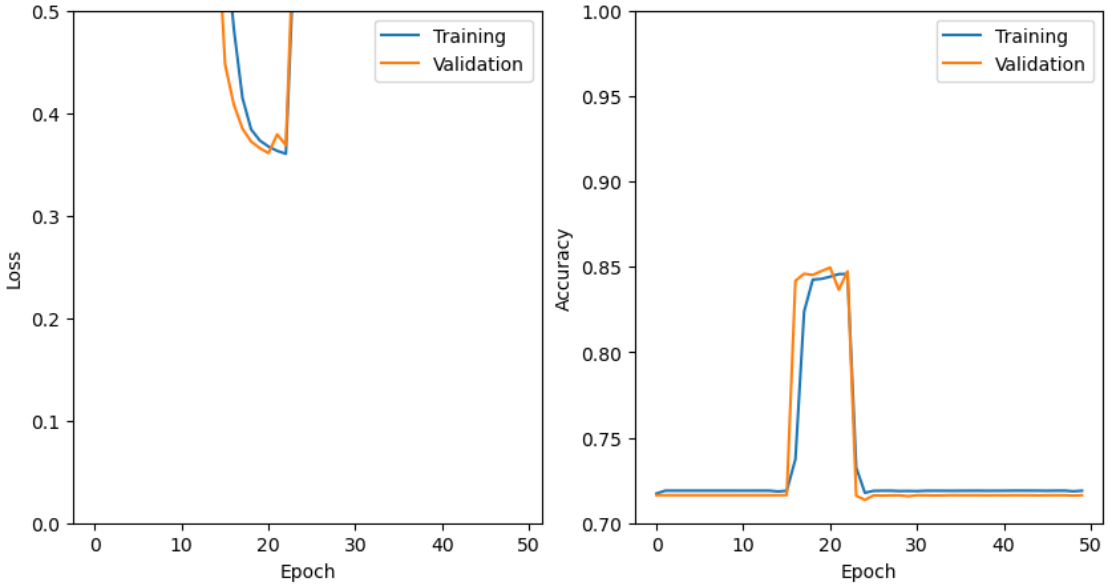
Numerical instability / overflow on the first weight update: unscaled large positive inputs + ReLU produce very large activations whihc gives huge gradients/weight updates -> Inf/NaN in logits or loss. Sometimes when running the cell above I get nan for loss and it breaks for this reason.

[16]:
```
print("ReLU without scaling score: ", roc_auc_score(y_test_filtered,␣
  ↪preds_nn_d))
roc_d = plot_roc(y_test_filtered, preds_nn_d)
plt.close()
```

```
ReLU without scaling score:  0.5023672255688736
```



[17]:
```
plot_model_history(history_d)
```



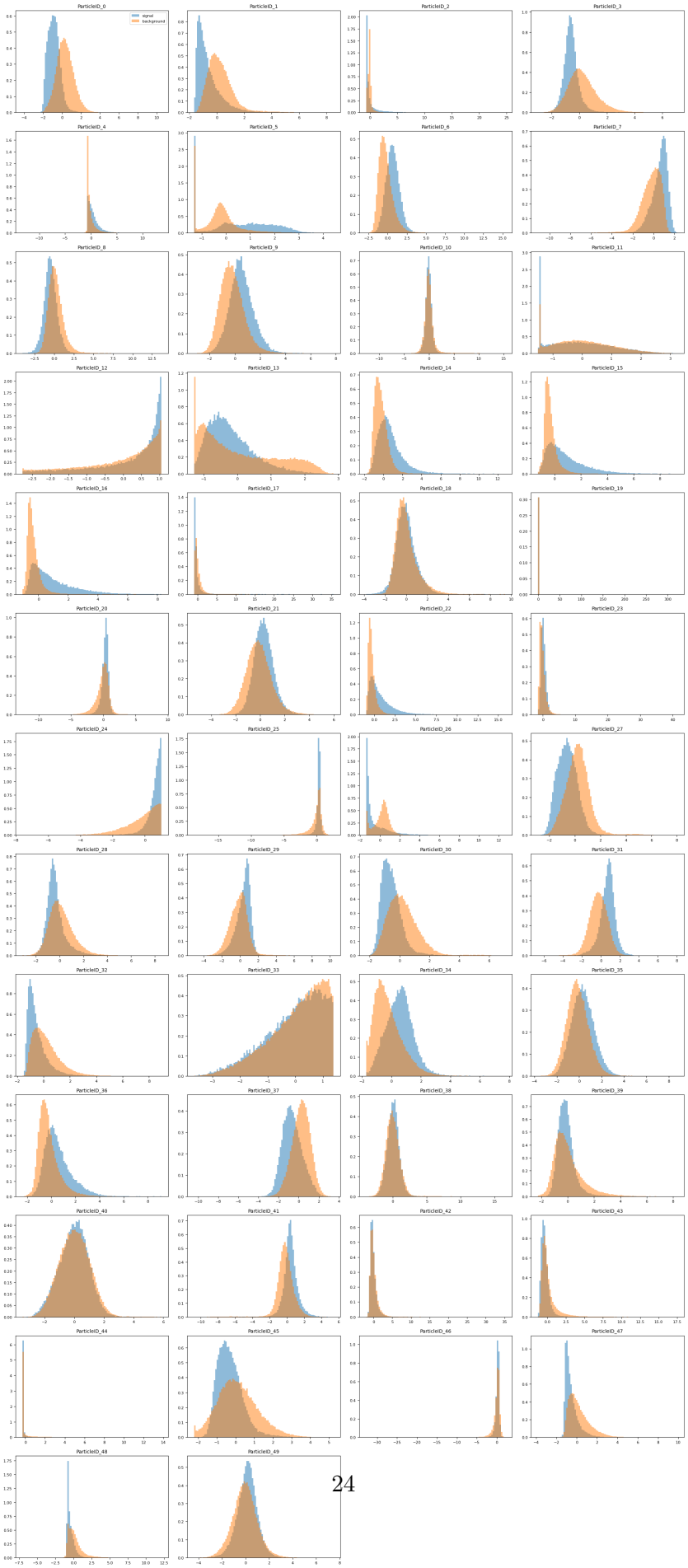
This is horrible, it's 50% - %50, so completely randomly guessing. For the features all values are already positive (x > 0) so ReLU (max(0, x)) acts like a linear model for most features. In problem E we will standardize the input features so it shouldnt be a problem anymore.

2.3 Problem E

Now, we will make two minor changes to the network with ReLU activations: preprocessing and the optimizer. For the feature preprocessing use sklearn.preprocessing.StandardScaler to standardize the input features. Note you should fit the standard scaler to the training data only and apply it to both the training and testing data. For the optimizer, use Adam with a learning rate of 0.001 (which is the default) instead of SGD. Train the model for 50 epochs

[18]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_filtered)
X_test_scaled = scaler.transform(X_test_filtered)
```

[55]:
```
fig = plot_feature_distributions(X_train_scaled, y_train_filtered,␣
↪feature_names)
plt.show()
```


ParticleID_0
signal
background
ParticleID_1
ParticleID_2
ParticleID_3
ParticleID_4
ParticleID_5
ParticleID_6
ParticleID_7
ParticleID_8
ParticleID_9
ParticleID_10
ParticleID_11
ParticleID_12
ParticleID_13
ParticleID_14
ParticleID_15
ParticleID_16
ParticleID_17
ParticleID_18
ParticleID_19
ParticleID_20
ParticleID_21
ParticleID_22
ParticleID_23
ParticleID_24
ParticleID_25
ParticleID_26
ParticleID_27
ParticleID_28
ParticleID_29
ParticleID_30
ParticleID_31
ParticleID_32
ParticleID_33
ParticleID_34
ParticleID_35
ParticleID_36
ParticleID_37
ParticleID_38
ParticleID_39
ParticleID_40
ParticleID_41
ParticleID_42
ParticleID_43
ParticleID_44
ParticleID_45
ParticleID_46
ParticleID_47
ParticleID_48
ParticleID_49

[19]:
```
# print all min and max values for each feature of X
for feature_idx in range(X_train_scaled.shape[1]):
        train_min = np.min(X_train_scaled[:, feature_idx])
        train_max = np.max(X_train_scaled[:, feature_idx])
        test_min = np.min(X_test_scaled[:, feature_idx])
        test_max = np.max(X_test_scaled[:, feature_idx])
        print(
                f"Feature {feature_idx:2d} | "
                f"train_scaled[min,max]=({train_min:.6g}, {train_max:.6g}) | "
                f"test_scaled[min,max]=({test_min:.6g}, {test_max:.6g})"
        )
```

```
Feature  0 | train_scaled[min,max]=(-4.18872, 10.4987) |
test_scaled[min,max]=(-4.18872, 9.64031)
Feature  1 | train_scaled[min,max]=(-1.65225, 8.49562) |
test_scaled[min,max]=(-1.6651, 7.17405)
Feature  2 | train_scaled[min,max]=(-0.687905, 24.7797) |
test_scaled[min,max]=(-0.687761, 22.1983)
Feature  3 | train_scaled[min,max]=(-2.99522, 7.06401) |
test_scaled[min,max]=(-2.91417, 7.61163)
Feature  4 | train_scaled[min,max]=(-13.2572, 13.6022) |
test_scaled[min,max]=(-11.7174, 13.2533)
Feature  5 | train_scaled[min,max]=(-1.30515, 4.41768) |
test_scaled[min,max]=(-1.30515, 3.6096)
Feature  6 | train_scaled[min,max]=(-2.82584, 15.3475) |
test_scaled[min,max]=(-2.82584, 10.5805)
Feature  7 | train_scaled[min,max]=(-11.1038, 2.25953) |
test_scaled[min,max]=(-11.2234, 2.35513)
Feature  8 | train_scaled[min,max]=(-4.07694, 13.6915) |
test_scaled[min,max]=(-4.16267, 14.3793)
Feature  9 | train_scaled[min,max]=(-3.21021, 7.801) |
test_scaled[min,max]=(-2.93826, 8.02441)
Feature 10 | train_scaled[min,max]=(-12.6434, 15.3338) |
test_scaled[min,max]=(-4.20765, 12.9434)
Feature 11 | train_scaled[min,max]=(-1.49893, 3.23759) |
test_scaled[min,max]=(-1.50084, 3.17066)
Feature 12 | train_scaled[min,max]=(-2.77265, 1.04952) |
test_scaled[min,max]=(-2.77267, 1.04952)
Feature 13 | train_scaled[min,max]=(-1.28165, 2.85509) |
test_scaled[min,max]=(-1.28165, 2.85509)
Feature 14 | train_scaled[min,max]=(-1.81975, 12.7653) |
test_scaled[min,max]=(-1.84108, 12.9087)
Feature 15 | train_scaled[min,max]=(-1.24611, 9.29861) |
test_scaled[min,max]=(-1.25234, 8.4718)
Feature 16 | train_scaled[min,max]=(-1.08601, 8.27712) |
```

```
test_scaled[min,max]=(-1.083, 7.89446)
Feature 17 | train_scaled[min,max]=(-0.876684, 35.4732) |
test_scaled[min,max]=(-0.867538, 32.7701)
Feature 18 | train_scaled[min,max]=(-3.82138, 9.42185) |
test_scaled[min,max]=(-4.05121, 9.04502)
Feature 19 | train_scaled[min,max]=(-0.350372, 321.984) |
test_scaled[min,max]=(-0.00315228, -0.00298876)
Feature 20 | train_scaled[min,max]=(-12.5223, 9.0015) |
test_scaled[min,max]=(-11.4583, 11.994)
Feature 21 | train_scaled[min,max]=(-5.39061, 5.97557) |
test_scaled[min,max]=(-5.06774, 4.90336)
Feature 22 | train_scaled[min,max]=(-0.994151, 15.8323) |
test_scaled[min,max]=(-0.994151, 13.3512)
Feature 23 | train_scaled[min,max]=(-1.48937, 41.3489) |
test_scaled[min,max]=(-1.47871, 26.786)
Feature 24 | train_scaled[min,max]=(-7.60548, 1.02563) |
test_scaled[min,max]=(-7.60548, 1.0255)
Feature 25 | train_scaled[min,max]=(-18.6674, 2.54226) |
test_scaled[min,max]=(-17.4653, 2.47052)
Feature 26 | train_scaled[min,max]=(-1.39061, 12.6484) |
test_scaled[min,max]=(-1.39061, 11.9017)
Feature 27 | train_scaled[min,max]=(-2.89641, 8.02042) |
test_scaled[min,max]=(-3.12581, 7.36744)
Feature 28 | train_scaled[min,max]=(-3.13531, 8.55054) |
test_scaled[min,max]=(-3.04673, 8.65407)
Feature 29 | train_scaled[min,max]=(-5.07012, 10.3574) |
test_scaled[min,max]=(-5.10789, 9.2222)
Feature 30 | train_scaled[min,max]=(-2.24337, 7.0815) |
test_scaled[min,max]=(-2.13866, 6.61754)
Feature 31 | train_scaled[min,max]=(-6.62612, 8.02517) |
test_scaled[min,max]=(-6.43459, 4.34244)
Feature 32 | train_scaled[min,max]=(-1.60246, 8.94079) |
test_scaled[min,max]=(-1.48516, 7.39835)
Feature 33 | train_scaled[min,max]=(-3.66108, 1.37617) |
test_scaled[min,max]=(-3.73248, 1.37606)
Feature 34 | train_scaled[min,max]=(-1.7143, 7.72325) |
test_scaled[min,max]=(-1.7143, 5.95474)
Feature 35 | train_scaled[min,max]=(-3.66981, 8.73032) |
test_scaled[min,max]=(-3.29329, 6.89334)
Feature 36 | train_scaled[min,max]=(-2.39728, 9.14473) |
test_scaled[min,max]=(-2.31044, 7.39274)
Feature 37 | train_scaled[min,max]=(-10.4091, 3.45568) |
test_scaled[min,max]=(-4.91611, 3.26627)
Feature 38 | train_scaled[min,max]=(-3.94163, 16.5709) |
test_scaled[min,max]=(-3.74804, 13.5733)
Feature 39 | train_scaled[min,max]=(-2.24024, 8.30364) |
test_scaled[min,max]=(-2.24024, 7.85105)
Feature 40 | train_scaled[min,max]=(-3.5069, 5.88) |
```

```
test_scaled[min,max]=(-3.04119, 5.01438)
Feature 41 | train_scaled[min,max]=(-10.769, 5.3868) |
test_scaled[min,max]=(-8.90209, 5.06737)
Feature 42 | train_scaled[min,max]=(-1.98426, 35.163) |
test_scaled[min,max]=(-1.87822, 28.0607)
Feature 43 | train_scaled[min,max]=(-1.26017, 17.4819) |
test_scaled[min,max]=(-1.23421, 15.7686)
Feature 44 | train_scaled[min,max]=(-0.305386, 13.803) |
test_scaled[min,max]=(-0.305386, 14.8196)
Feature 45 | train_scaled[min,max]=(-2.22026, 5.2416) |
test_scaled[min,max]=(-2.22026, 5.30072)
Feature 46 | train_scaled[min,max]=(-32.7509, 2.10256) |
test_scaled[min,max]=(-28.0754, 2.14326)
Feature 47 | train_scaled[min,max]=(-3.80891, 9.88104) |
test_scaled[min,max]=(-8.54654, 6.54744)
Feature 48 | train_scaled[min,max]=(-7.06687, 12.1341) |
test_scaled[min,max]=(-8.97674, 10.4215)
Feature 49 | train_scaled[min,max]=(-4.36067, 7.45888) |
test_scaled[min,max]=(-4.36067, 6.61437)
```

From the plots and the min, max values we see clearly now the features have data points in the negative range as well, so ReLU can act better on it in a non-linear way.

[ ]:
```
inputs = Input(shape=X_train_scaled.shape[1:])
outputs = Dense(128, activation='relu')(inputs)
outputs = Dense(128, activation='relu')(outputs)
outputs = Dense(128, activation='relu')(outputs)
outputs = Dense(1, activation='sigmoid')(outputs)
model_e = Model(inputs=inputs, outputs=outputs)

model_e.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
history_e = model_e.fit(X_train_scaled, y_train_filtered, epochs=50,␣
 ↪batch_size=128, validation_data=(X_test_scaled, y_test_filtered), verbose=0)

preds_nn_e = model_e.predict(X_test_scaled)
```

```
Epoch 1/50
810/810            6s 4ms/step -
accuracy: 0.9280 - loss: 0.1792 - val_accuracy: 0.9407 - val_loss: 0.1512
Epoch 2/50
810/810            3s 3ms/step -
accuracy: 0.9428 - loss: 0.1449 - val_accuracy: 0.9445 - val_loss: 0.1439
Epoch 3/50
810/810            3s 3ms/step -
accuracy: 0.9463 - loss: 0.1358 - val_accuracy: 0.9446 - val_loss: 0.1414
Epoch 4/50
810/810            2s 3ms/step -
```

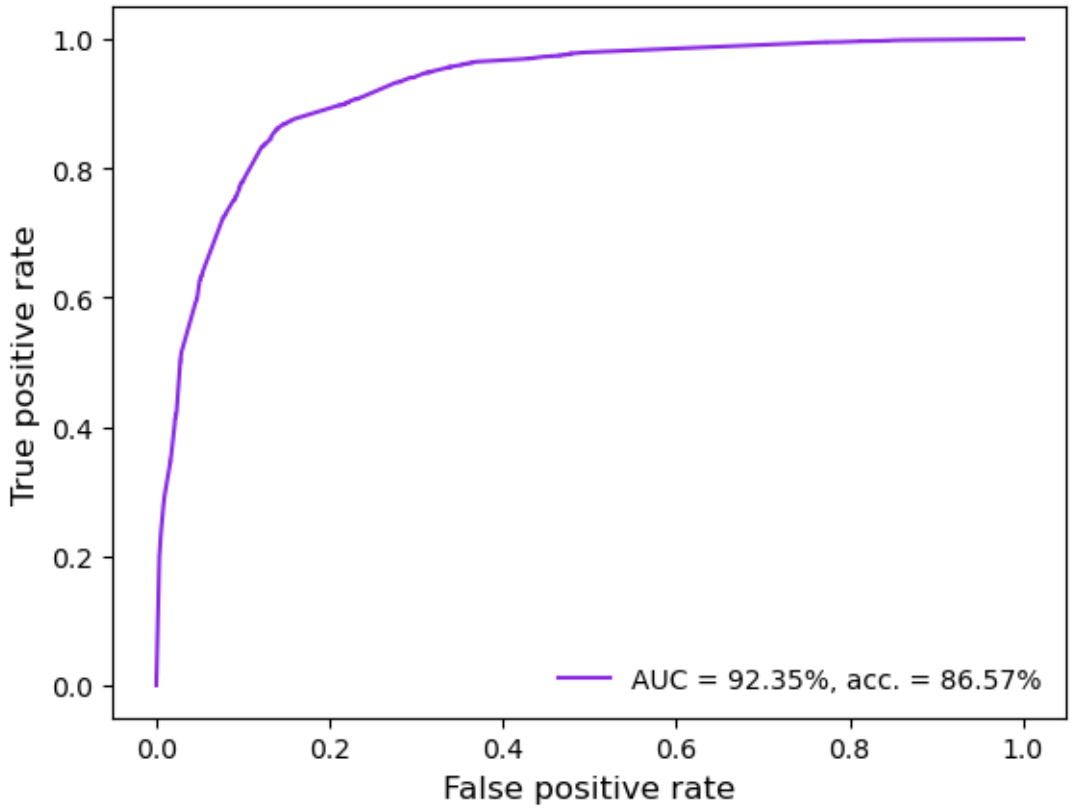
```
accuracy: 0.9483 - loss: 0.1305 - val_accuracy: 0.9415 - val_loss: 0.1470
Epoch 5/50
810/810             2s 3ms/step -
accuracy: 0.9510 - loss: 0.1250 - val_accuracy: 0.9479 - val_loss: 0.1330
Epoch 6/50
810/810             2s 3ms/step -
accuracy: 0.9522 - loss: 0.1210 - val_accuracy: 0.9451 - val_loss: 0.1378
Epoch 7/50
810/810             2s 3ms/step -
accuracy: 0.9540 - loss: 0.1175 - val_accuracy: 0.9476 - val_loss: 0.1364
Epoch 8/50
810/810             3s 3ms/step -
accuracy: 0.9550 - loss: 0.1139 - val_accuracy: 0.9496 - val_loss: 0.1294
Epoch 9/50
810/810             2s 2ms/step -
accuracy: 0.9566 - loss: 0.1110 - val_accuracy: 0.9487 - val_loss: 0.1325
Epoch 10/50
810/810             2s 2ms/step -
accuracy: 0.9580 - loss: 0.1078 - val_accuracy: 0.9488 - val_loss: 0.1321
Epoch 11/50
810/810             2s 3ms/step -
accuracy: 0.9589 - loss: 0.1049 - val_accuracy: 0.9488 - val_loss: 0.1339
Epoch 12/50
810/810             2s 3ms/step -
accuracy: 0.9608 - loss: 0.1015 - val_accuracy: 0.9496 - val_loss: 0.1375
Epoch 13/50
810/810             3s 3ms/step -
accuracy: 0.9617 - loss: 0.0986 - val_accuracy: 0.9487 - val_loss: 0.1399
Epoch 14/50
810/810             3s 3ms/step -
accuracy: 0.9624 - loss: 0.0956 - val_accuracy: 0.9486 - val_loss: 0.1397
Epoch 15/50
810/810             2s 3ms/step -
accuracy: 0.9641 - loss: 0.0923 - val_accuracy: 0.9472 - val_loss: 0.1421
Epoch 16/50
810/810             2s 3ms/step -
accuracy: 0.9657 - loss: 0.0891 - val_accuracy: 0.9465 - val_loss: 0.1439
Epoch 17/50
810/810             2s 3ms/step -
accuracy: 0.9667 - loss: 0.0860 - val_accuracy: 0.9488 - val_loss: 0.1570
Epoch 18/50
810/810             2s 2ms/step -
accuracy: 0.9677 - loss: 0.0837 - val_accuracy: 0.9487 - val_loss: 0.1536
Epoch 19/50
810/810             2s 3ms/step -
accuracy: 0.9692 - loss: 0.0803 - val_accuracy: 0.9471 - val_loss: 0.1606
Epoch 20/50
810/810             2s 2ms/step -
```

```
accuracy: 0.9703 - loss: 0.0780 - val_accuracy: 0.9473 - val_loss: 0.1566
Epoch 21/50
810/810             2s 2ms/step -
accuracy: 0.9718 - loss: 0.0745 - val_accuracy: 0.9471 - val_loss: 0.1708
Epoch 22/50
810/810             2s 3ms/step -
accuracy: 0.9723 - loss: 0.0721 - val_accuracy: 0.9469 - val_loss: 0.1741
Epoch 23/50
810/810             2s 3ms/step -
accuracy: 0.9732 - loss: 0.0692 - val_accuracy: 0.9443 - val_loss: 0.1741
Epoch 24/50
810/810             2s 3ms/step -
accuracy: 0.9744 - loss: 0.0666 - val_accuracy: 0.9451 - val_loss: 0.1858
Epoch 25/50
810/810             2s 3ms/step -
accuracy: 0.9756 - loss: 0.0642 - val_accuracy: 0.9425 - val_loss: 0.1980
Epoch 26/50
810/810             2s 3ms/step -
accuracy: 0.9766 - loss: 0.0610 - val_accuracy: 0.9441 - val_loss: 0.1865
Epoch 27/50
810/810             2s 3ms/step -
accuracy: 0.9776 - loss: 0.0588 - val_accuracy: 0.9424 - val_loss: 0.1990
Epoch 28/50
810/810             2s 2ms/step -
accuracy: 0.9789 - loss: 0.0560 - val_accuracy: 0.9452 - val_loss: 0.2113
Epoch 29/50
810/810             2s 3ms/step -
accuracy: 0.9793 - loss: 0.0543 - val_accuracy: 0.9412 - val_loss: 0.2111
Epoch 30/50
810/810             2s 2ms/step -
accuracy: 0.9792 - loss: 0.0531 - val_accuracy: 0.9404 - val_loss: 0.2205
Epoch 31/50
810/810             2s 3ms/step -
accuracy: 0.9807 - loss: 0.0502 - val_accuracy: 0.9419 - val_loss: 0.2232
Epoch 32/50
810/810             3s 3ms/step -
accuracy: 0.9818 - loss: 0.0481 - val_accuracy: 0.9428 - val_loss: 0.2368
Epoch 33/50
810/810             2s 3ms/step -
accuracy: 0.9825 - loss: 0.0457 - val_accuracy: 0.9419 - val_loss: 0.2495
Epoch 34/50
810/810             2s 3ms/step -
accuracy: 0.9834 - loss: 0.0437 - val_accuracy: 0.9417 - val_loss: 0.2605
Epoch 35/50
810/810             2s 3ms/step -
accuracy: 0.9837 - loss: 0.0423 - val_accuracy: 0.9405 - val_loss: 0.2610
Epoch 36/50
810/810             3s 3ms/step -
```

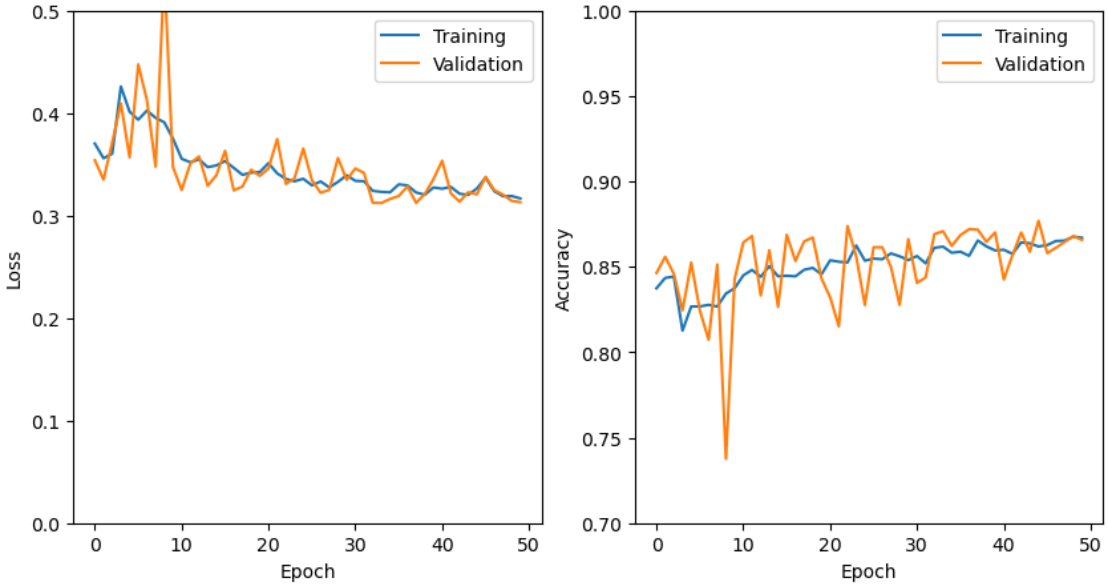
```
accuracy: 0.9844 - loss: 0.0417 - val_accuracy: 0.9385 - val_loss: 0.2818
Epoch 37/50
810/810             3s 3ms/step -
accuracy: 0.9845 - loss: 0.0408 - val_accuracy: 0.9407 - val_loss: 0.2642
Epoch 38/50
810/810             3s 3ms/step -
accuracy: 0.9859 - loss: 0.0374 - val_accuracy: 0.9413 - val_loss: 0.2914
Epoch 39/50
810/810             3s 3ms/step -
accuracy: 0.9863 - loss: 0.0368 - val_accuracy: 0.9404 - val_loss: 0.2975
Epoch 40/50
810/810             3s 4ms/step -
accuracy: 0.9863 - loss: 0.0377 - val_accuracy: 0.9416 - val_loss: 0.2976
Epoch 41/50
810/810             3s 3ms/step -
accuracy: 0.9871 - loss: 0.0346 - val_accuracy: 0.9412 - val_loss: 0.3068
Epoch 42/50
810/810             3s 4ms/step -
accuracy: 0.9877 - loss: 0.0339 - val_accuracy: 0.9400 - val_loss: 0.3037
Epoch 43/50
810/810             2s 2ms/step -
accuracy: 0.9883 - loss: 0.0312 - val_accuracy: 0.9409 - val_loss: 0.3188
Epoch 44/50
810/810             2s 3ms/step -
accuracy: 0.9872 - loss: 0.0333 - val_accuracy: 0.9411 - val_loss: 0.3253
Epoch 45/50
810/810             4s 5ms/step -
accuracy: 0.9888 - loss: 0.0303 - val_accuracy: 0.9385 - val_loss: 0.3339
Epoch 46/50
810/810             5s 4ms/step -
accuracy: 0.9900 - loss: 0.0276 - val_accuracy: 0.9394 - val_loss: 0.3382
Epoch 47/50
810/810             4s 5ms/step -
accuracy: 0.9891 - loss: 0.0292 - val_accuracy: 0.9396 - val_loss: 0.3431
Epoch 48/50
810/810             4s 5ms/step -
accuracy: 0.9898 - loss: 0.0276 - val_accuracy: 0.9385 - val_loss: 0.3675
Epoch 49/50
810/810             4s 5ms/step -
accuracy: 0.9902 - loss: 0.0267 - val_accuracy: 0.9394 - val_loss: 0.3787
Epoch 50/50
810/810             4s 5ms/step -
accuracy: 0.9899 - loss: 0.0285 - val_accuracy: 0.9389 - val_loss: 0.3769
811/811             1s 1ms/step
```

[21]:
```
roc_c
```

[21]:

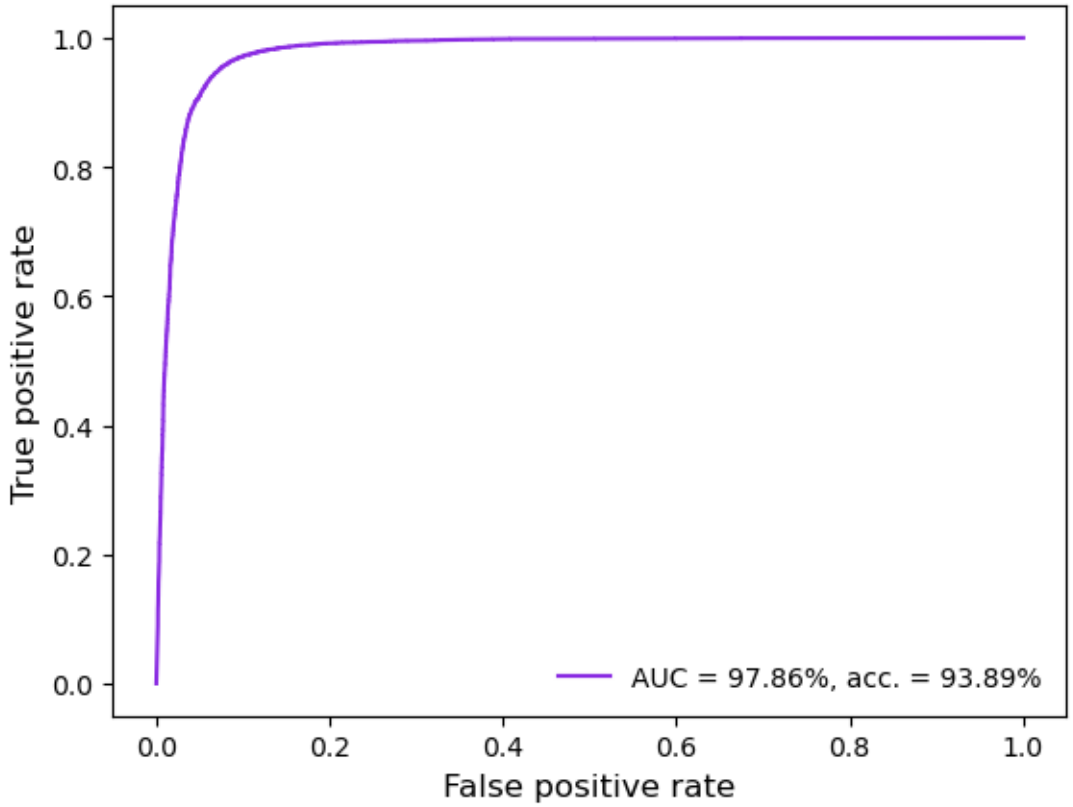


[22]:
```
plot_model_history(history_c)
```

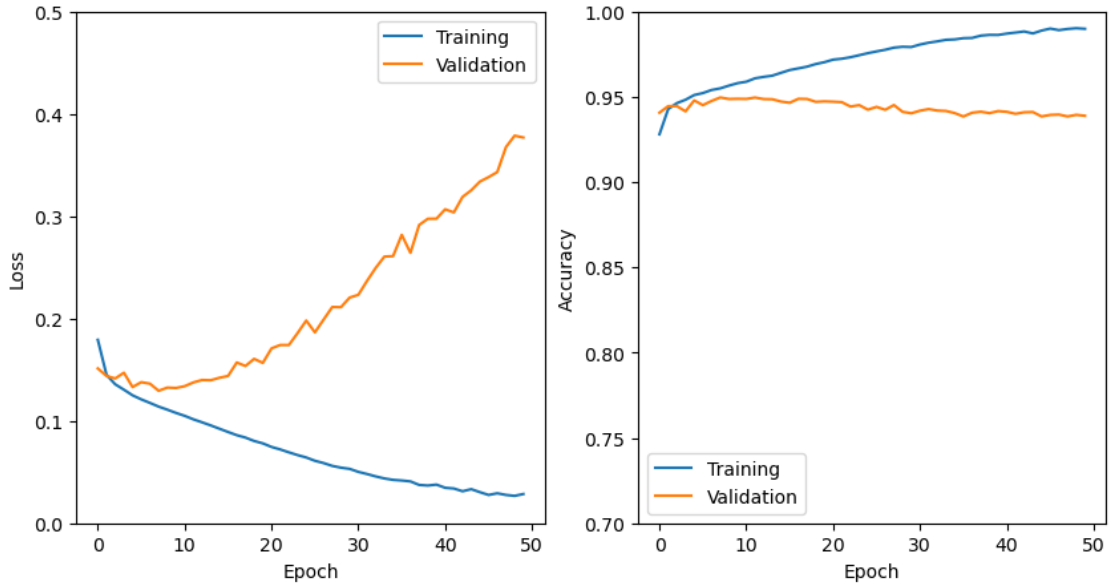


[23]:
```
print("Scaled AUC score: ", roc_auc_score(y_test_filtered, preds_nn_e))
roc_e = plot_roc(y_test_filtered, preds_nn_e)
plt.close()
```

```
Scaled AUC score:  0.9785773075297518
```



[24]:
```
plot_model_history(history_e)
```



We see it has a very nice roc curve now. It performs around 6-9% better than the out of the box NN version. The scaled NN AUC closesly resembles the BDT version, only slightly (0.5%) worse. There does appear to be some overfitting.

3 Optional refinements

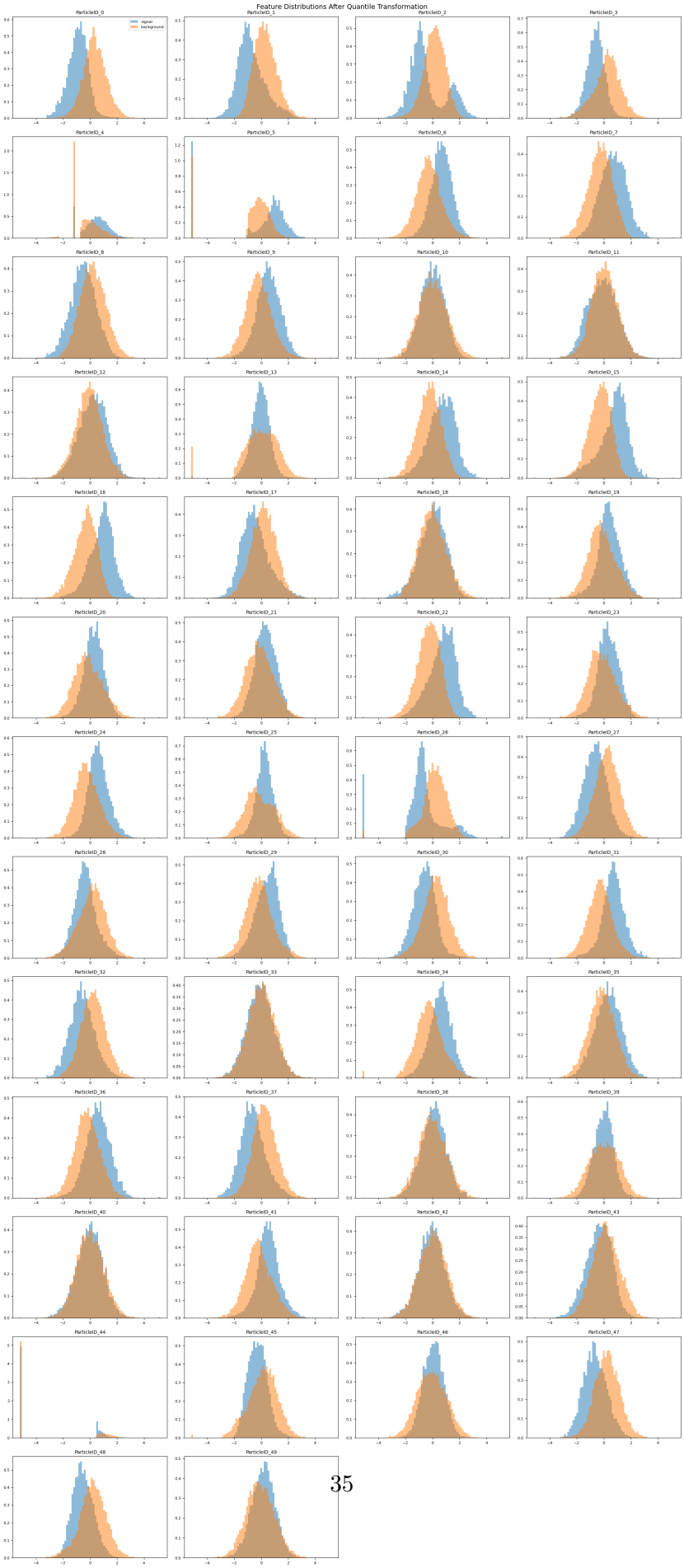
Add further refinements to the neural network in the last step of the previous exercise. Try and get the classification as good as you can. For example, consider batchnorm, layer initialization, dropout, wider/deeper nets, …. . If you can't find a significant improvement, just show the code and report on the result of a few experiments (I tried X because I thought/saw Y, result was Z).

Let's first manipulate our data so that ReLU can work with it even better!

[ ]:
```
# this was suggested to me by ChatGPT I did not come up with this myself
# my first try was removing outliers based on percentile but then 40% of the␣
 ↪training data was removed, ouch

from sklearn.preprocessing import QuantileTransformer

qt = QuantileTransformer(output_distribution='normal', n_quantiles=1000)
X_train_transformed = qt.fit_transform(X_train_filtered)
X_test_transformed = qt.transform(X_test_filtered)
```

[65]:
```
# Visualize the transformed features
fig = plot_feature_distributions(X_train_transformed, y_train_filtered,␣
 ↪feature_names)
plt.suptitle('Feature Distributions After Quantile Transformation',␣
 ↪fontsize=14, y=1.001)
```

```
plt.show()
```


Feature Distributions After Quantile Transformation
ParticleID_0
signal
background
ParticleID_1
ParticleID_2
ParticleID_3
ParticleID_4
ParticleID_5
ParticleID_6
ParticleID_7
ParticleID_8
ParticleID_9
ParticleID_10
ParticleID_11
ParticleID_12
ParticleID_13
ParticleID_14
ParticleID_15
ParticleID_16
ParticleID_17
ParticleID_18
ParticleID_19
ParticleID_20
ParticleID_21
ParticleID_22
ParticleID_23
ParticleID_24
ParticleID_25
ParticleID_26
ParticleID_27
ParticleID_28
ParticleID_29
ParticleID_30
ParticleID_31
ParticleID_32
ParticleID_33
ParticleID_34
ParticleID_35
ParticleID_36
ParticleID_37
ParticleID_38
ParticleID_39
ParticleID_40
ParticleID_41
ParticleID_42
ParticleID_43
ParticleID_44
ParticleID_45
ParticleID_46
ParticleID_47
ParticleID_48
ParticleID_49

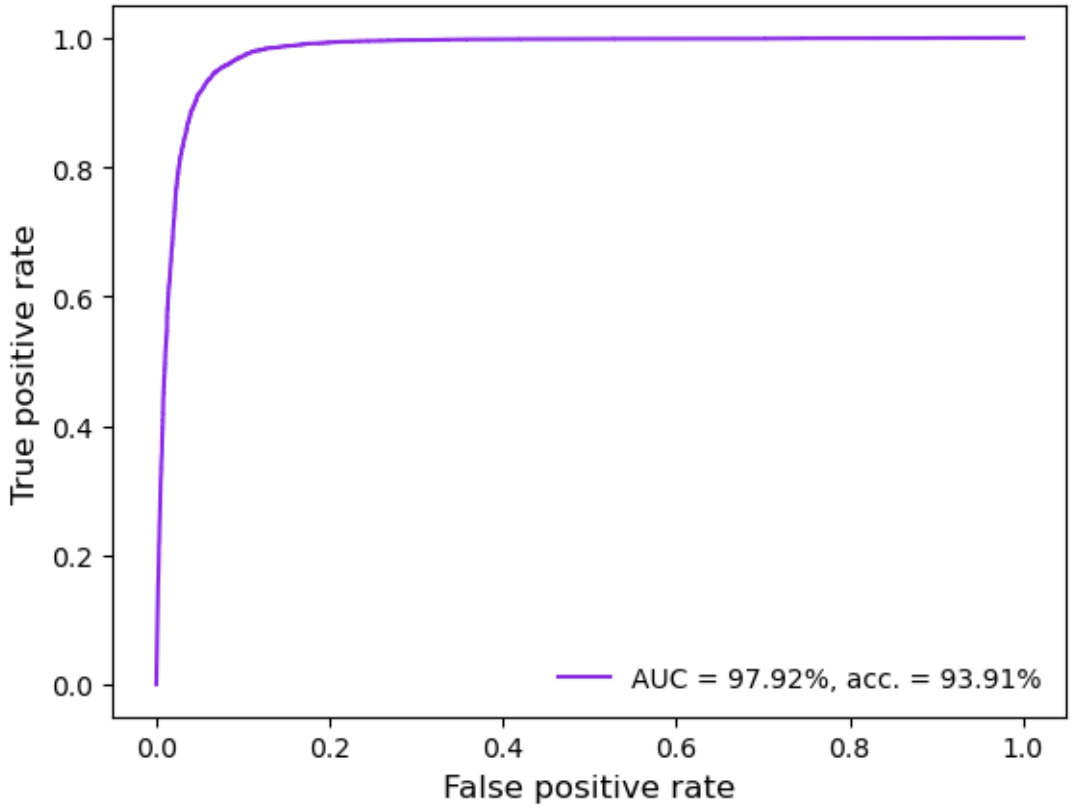
The scaling really worked well together with Adam, so we keep that. Now we can apply other things as well.

[ ]:
```
from tensorflow.keras.layers import BatchNormalization, Dropout
from tensorflow.keras.initializers import HeNormal, GlorotUniform
from tensorflow.keras.optimizers import Adam

def run_experiment(hidden_layers, use_batchnorm=False, dropout_rate=0.0,␣
↪kernel_initializer='glorot_uniform'):

    inputs = Input(shape=(X_train_transformed.shape[1],))

    x = inputs
    for units in hidden_layers:
        x = Dense(units, activation='relu',␣
↪kernel_initializer=kernel_initializer)(x)
        if use_batchnorm:
            x = BatchNormalization()(x)
        if dropout_rate > 0:
            x = Dropout(dropout_rate)(x)

    outputs = Dense(1, activation='sigmoid')(x)
    model = Model(inputs=inputs, outputs=outputs)

    model.compile(optimizer=Adam(learning_rate=1e-3),␣
↪loss='binary_crossentropy', metrics=['accuracy'])

    history = model.fit(X_train_transformed, y_train_filtered, epochs=50,␣
↪batch_size=128, validation_data=(X_test_transformed, y_test_filtered),␣
↪verbose=0)

    preds = model.predict(X_test_transformed)

    auc = roc_auc_score(y_test_filtered, preds)

    print(f"AUC: {auc:.6f}")

    roc_plot = plot_roc(y_test_filtered, preds)

    return model, history, auc, roc_plot
```

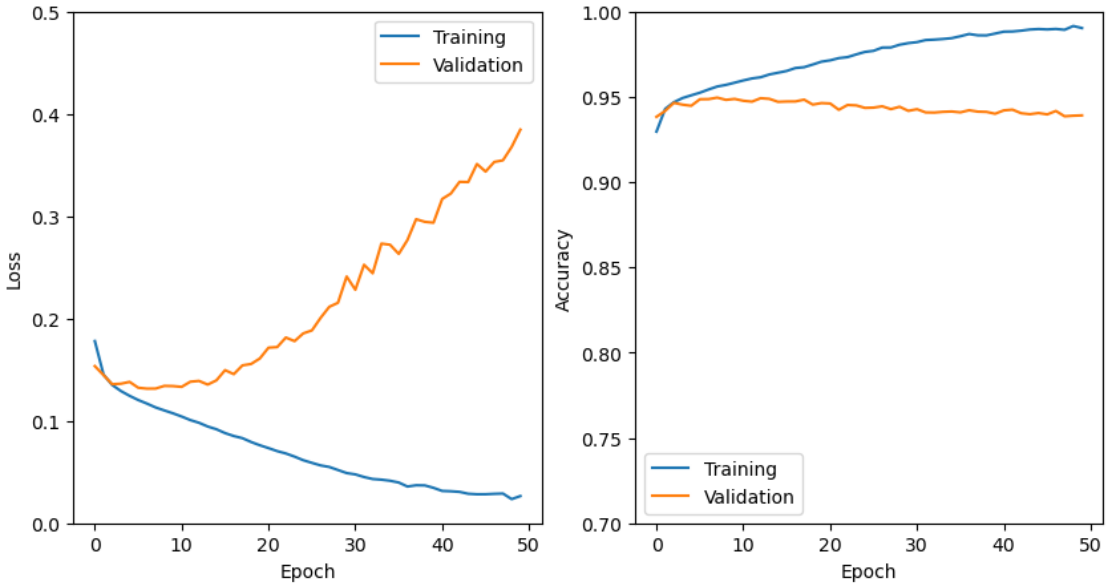
3.0.1 Experiment 0: Transformed with Quantile

[73]:
```
model_qu, history_qu, auc_qu, roc_plot_qu = run_experiment(
    hidden_layers=[128, 128, 128],
    use_batchnorm=False,
    dropout_rate=0.0,
    kernel_initializer=GlorotUniform()
)
```

```
811/811             1s 1ms/step
AUC: 0.979218
```



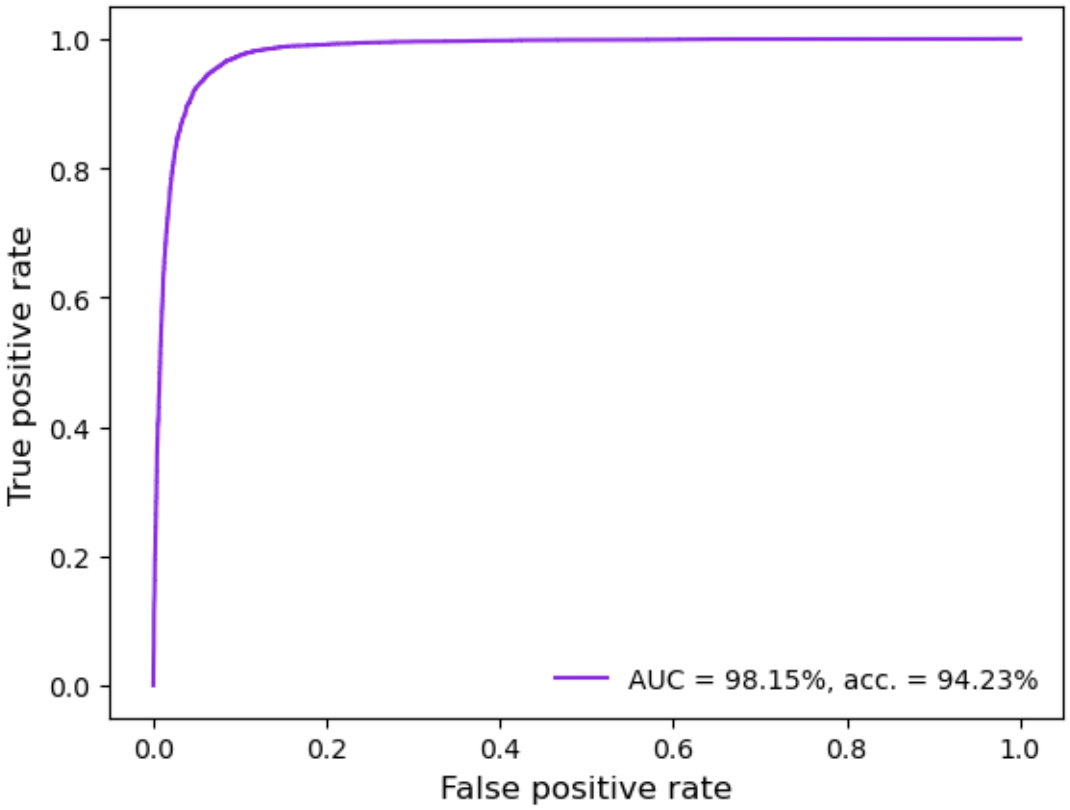
[74]:
```
plot_model_history(history_qu)
```



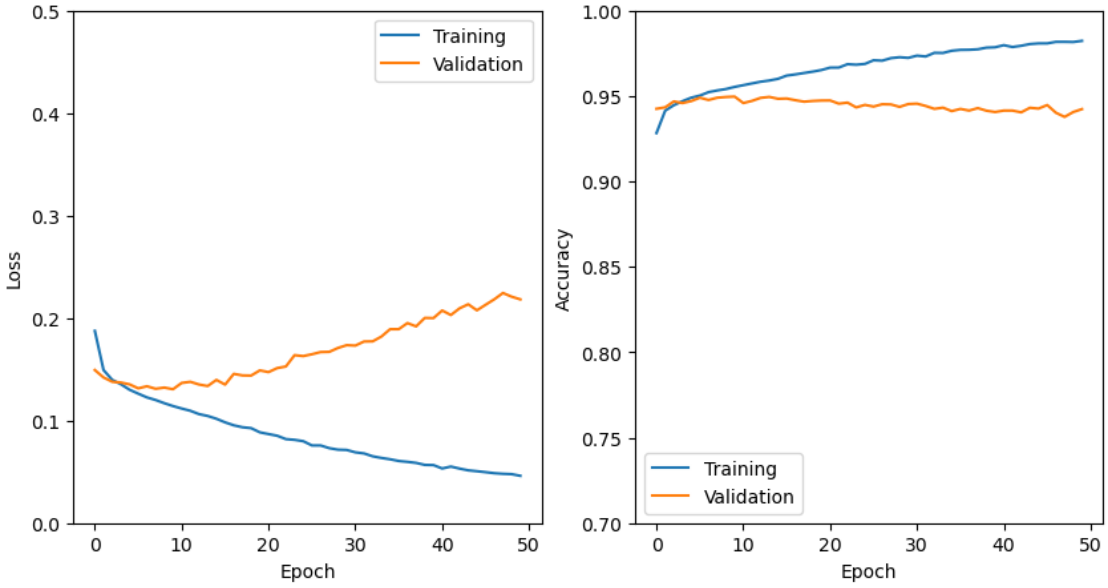
3.0.2 Experiment 1: Batch Normalization

[75]:
```
model_bn, history_bn, auc_bn, roc_plot_bn = run_experiment(
    hidden_layers=[128, 128, 128],
    use_batchnorm=True,
    dropout_rate=0.0,
    kernel_initializer=GlorotUniform()
)
```

```
811/811             2s 2ms/step
AUC: 0.981540
```



[76]:
```
plot_model_history(history_bn)
```



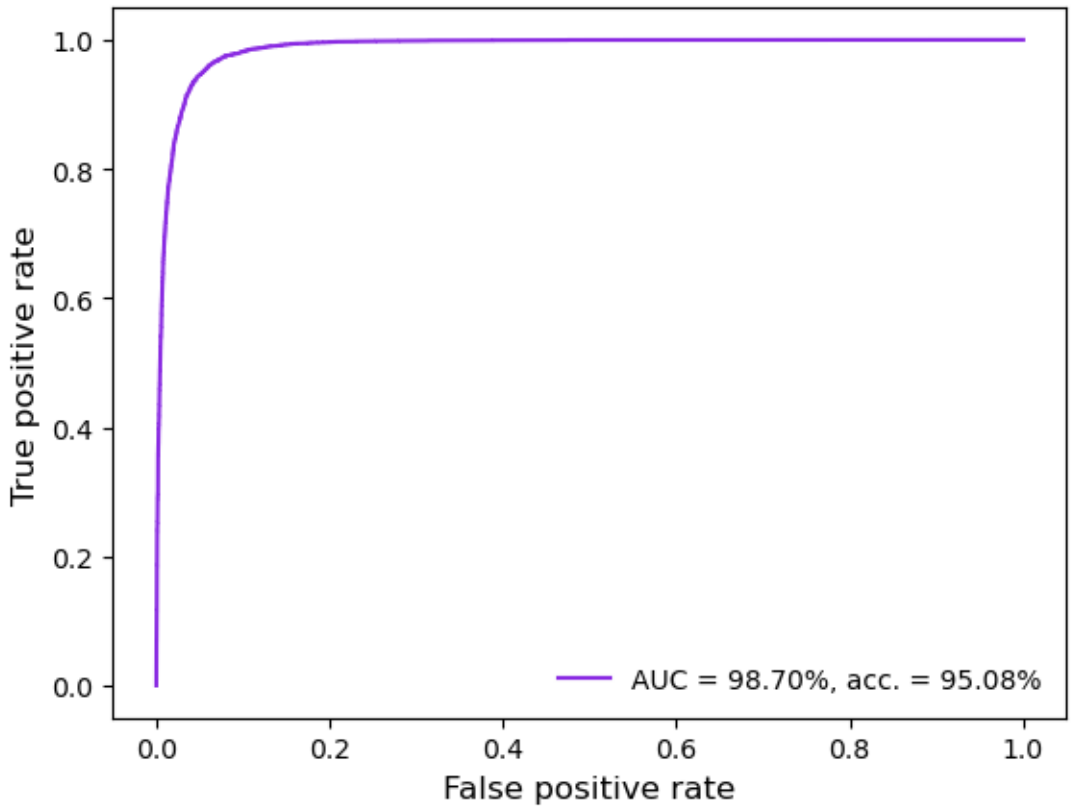
Is a bit better than the scaled version and pretty close to the BDT. But it looks like we have some overfitting.

3.0.3 Experiment 2: Dropout

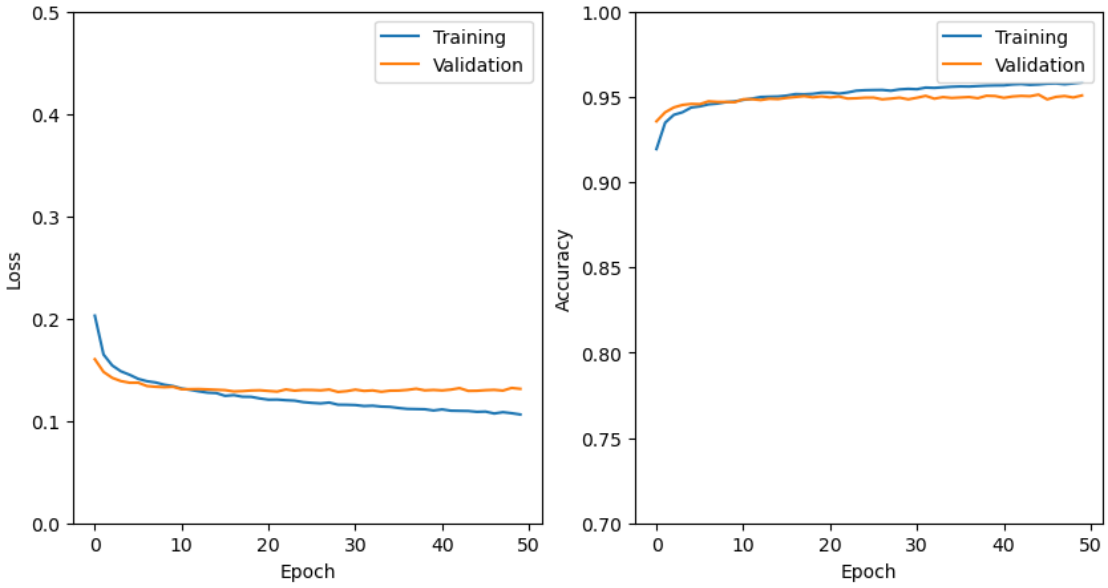
This would have the most promising results because people are doing this a lot. And if people are doing it, then it must be useful.

[78]:
```
model_do, history_do, auc_do, roc_plot_do = run_experiment(
    hidden_layers=[128, 128, 128],
    use_batchnorm=False,
    dropout_rate=0.2,
    kernel_initializer=GlorotUniform()
)
```

```
811/811             1s 1ms/step
AUC: 0.986951
```



[79]:
```
plot_model_history(history_do)
```



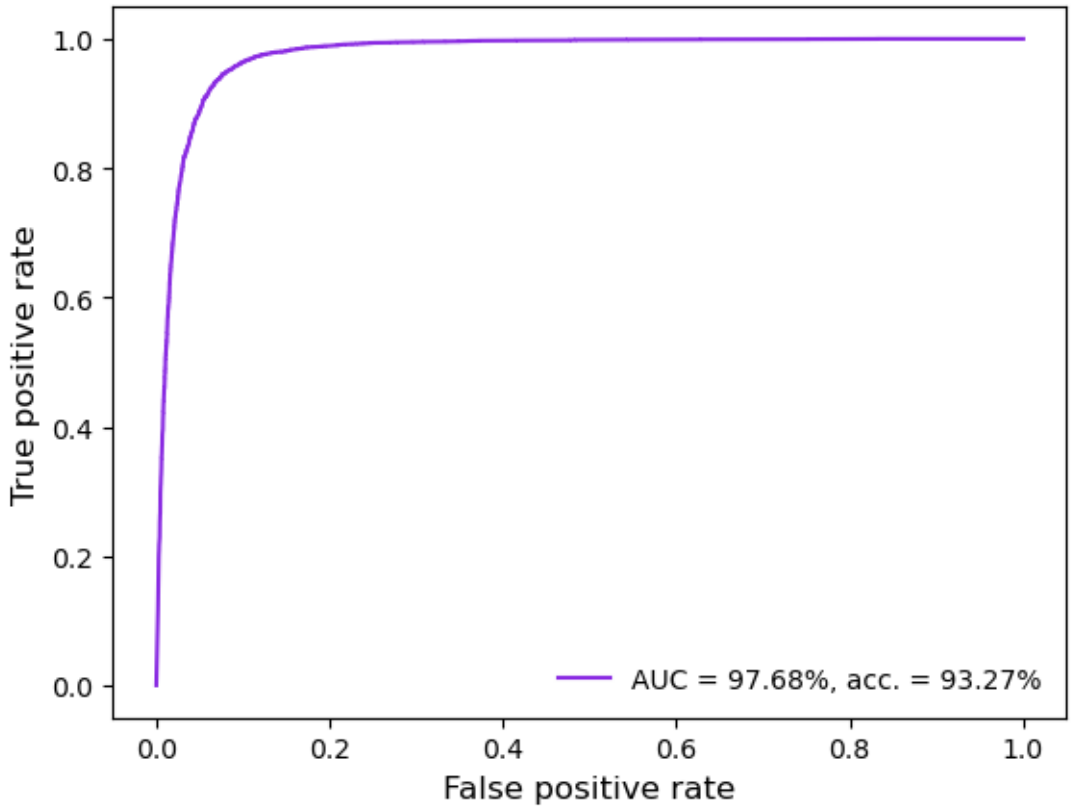
Outperforms the BDT! And there is no overfitting.

3.0.4 Experiment 3: Layer Initialization (He Normal)

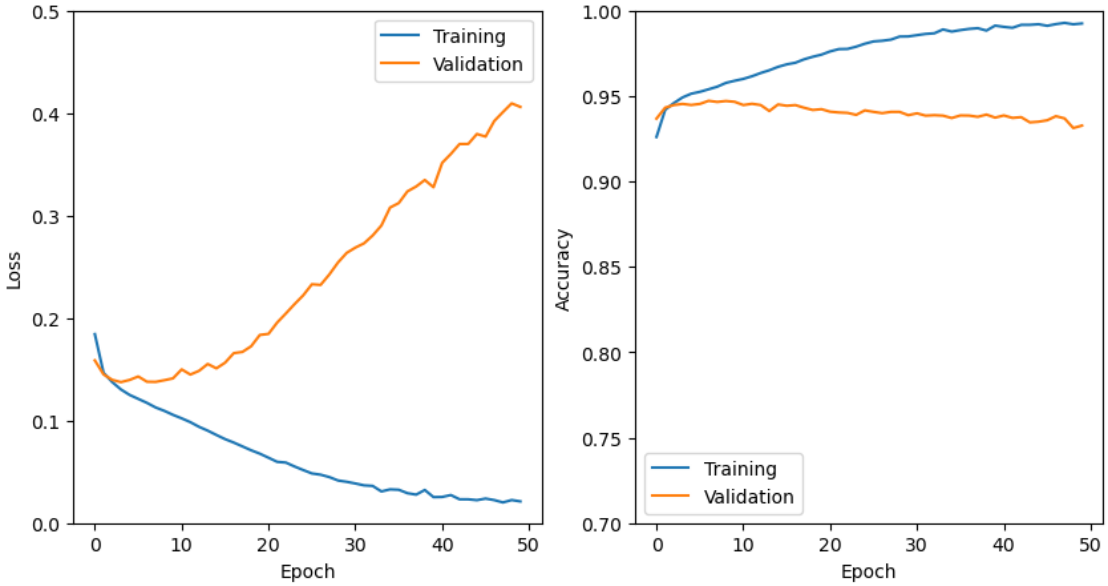
ReLU zeros about half the activations, He normal prevents signal shrinking.

[80]:
```
model_he, history_he, auc_he, roc_plot_he = run_experiment(
    hidden_layers=[128, 128, 128],
    use_batchnorm=False,
    dropout_rate=0.0,
    kernel_initializer=HeNormal()
)
```

```
811/811             1s 1ms/step
AUC: 0.976757
```



[81]:
```
plot_model_history(history_he)
```



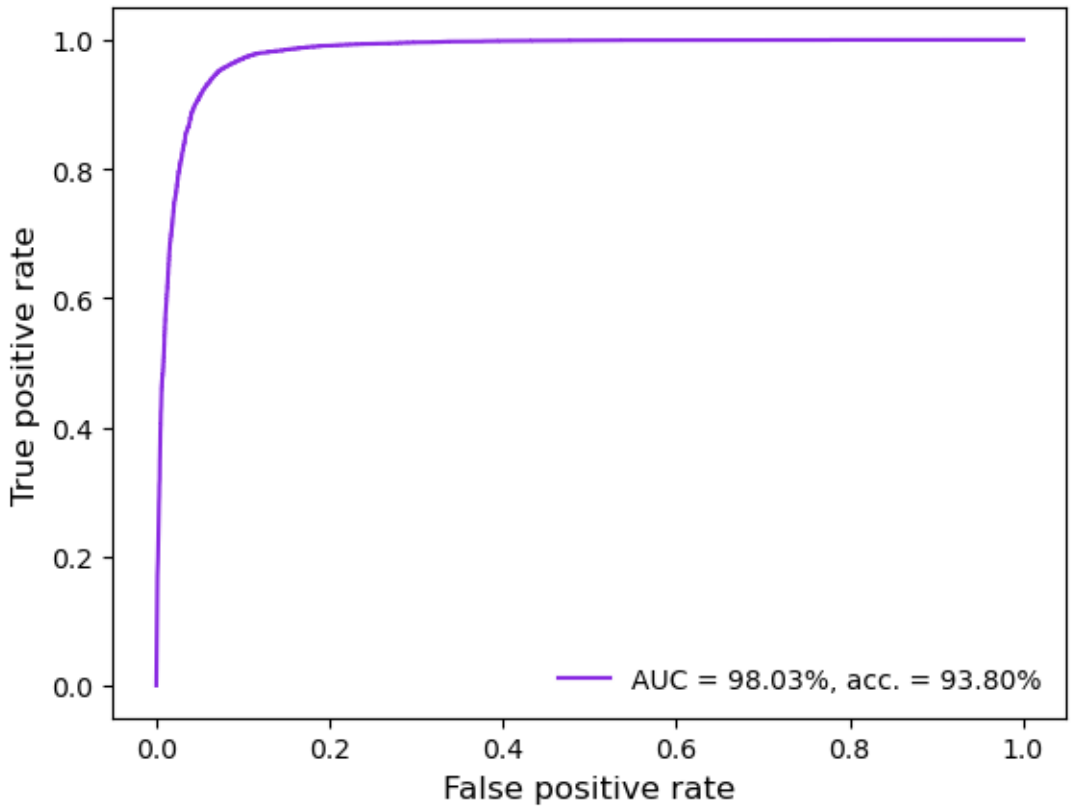
This gives a small improvement over the scaled model if we just look at AUC and acc. Although it does seem to be overfitting quite some if we look at the loss curve and the difference between training and validation accuracy.

3.0.5 Experiment 4: Wider and Deeper Networks

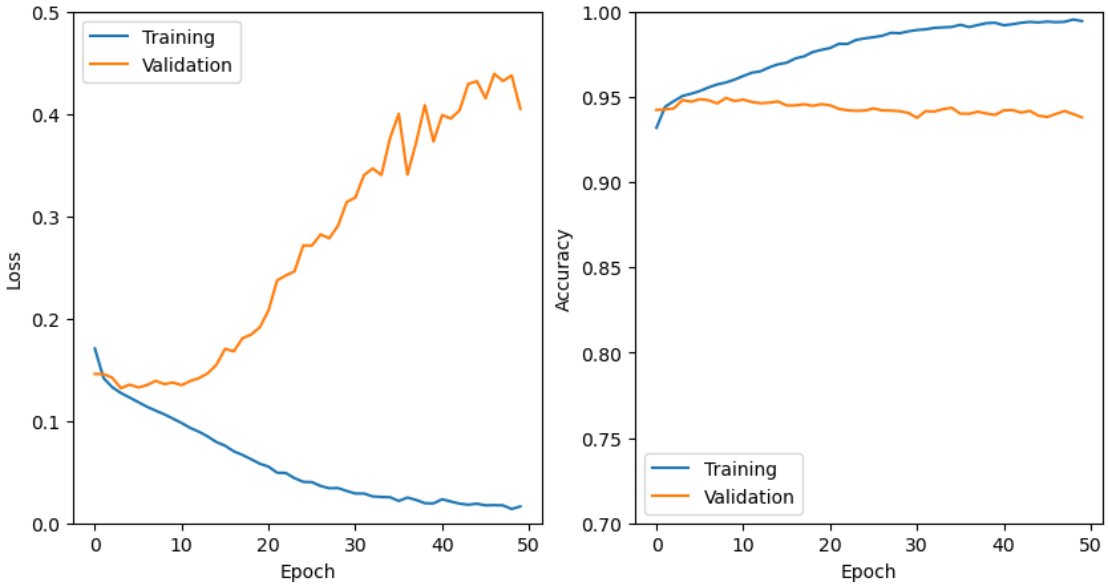
Wider

[82]:
```
model_wide, history_wide, auc_wide, roc_plot_wide = run_experiment(
    hidden_layers=[256, 256, 128],
    use_batchnorm=False,
    dropout_rate=0.0,
    kernel_initializer=GlorotUniform()
)
```

```
811/811             1s 1ms/step
AUC: 0.980305
```



[83]:
```
plot_model_history(history_wide)
```

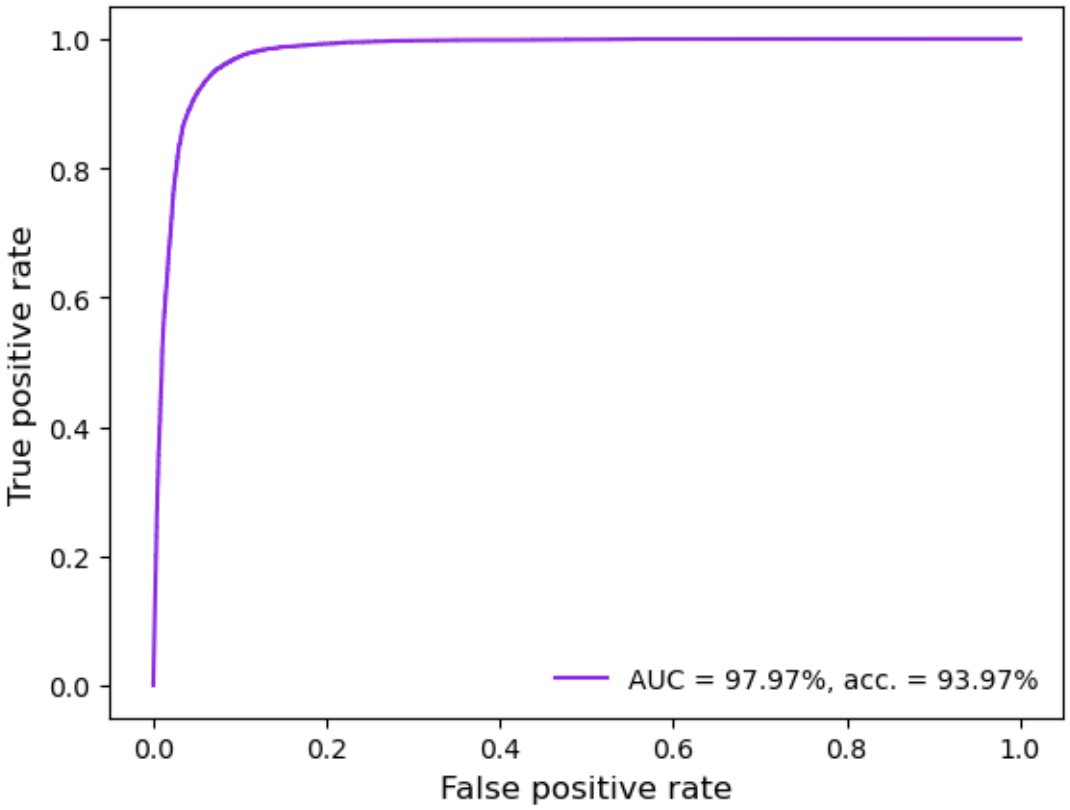


Enormous overfitting, if I may say so. But quite good accuracy

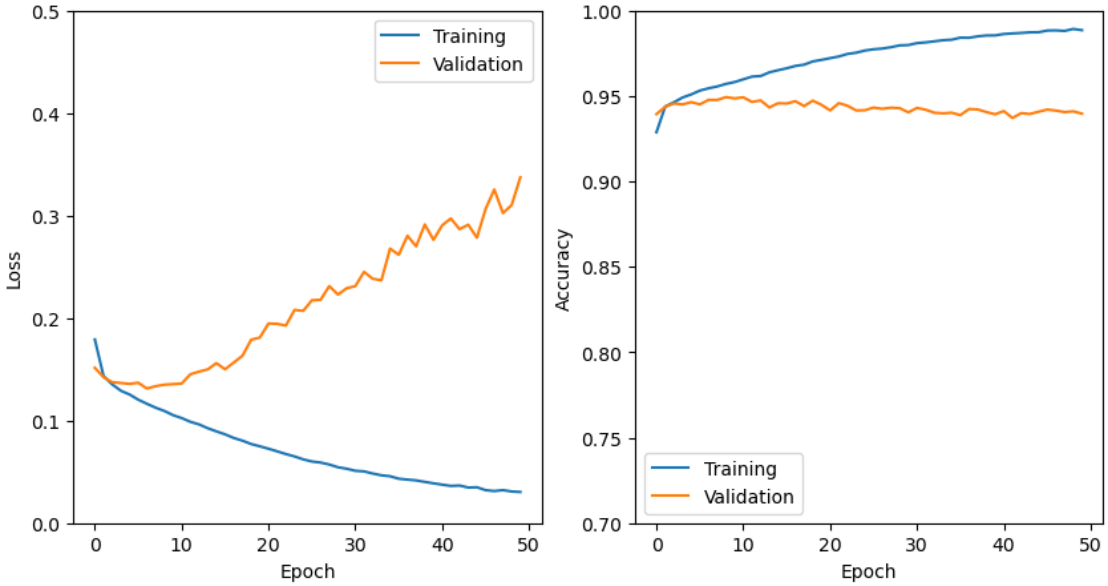
deeper

[84]:
```
model_deep, history_deep, auc_deep, roc_plot_deep = run_experiment(
    hidden_layers=[128, 128, 128, 64, 64],
    use_batchnorm=False,
    dropout_rate=0.0,
    kernel_initializer=GlorotUniform()
)
```

```
811/811             1s 1ms/step
AUC: 0.979680
```



[85]:
```
plot_model_history(history_deep)
```



3.0.6 Experiment 5: Replicating a paper

To get a classification as good as possible, why not just copy someone elses work!

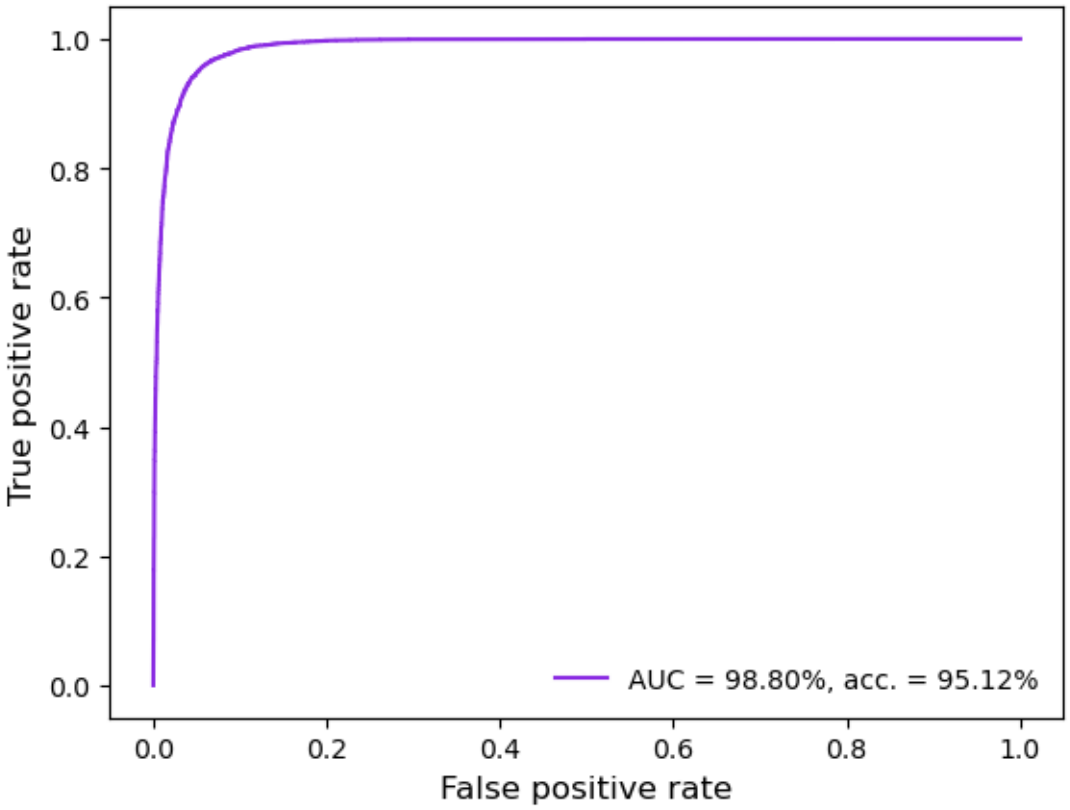
Although they use a different filter/scaler for processing the features, i will use the same architecture

https://arxiv.org/pdf/2104.14045

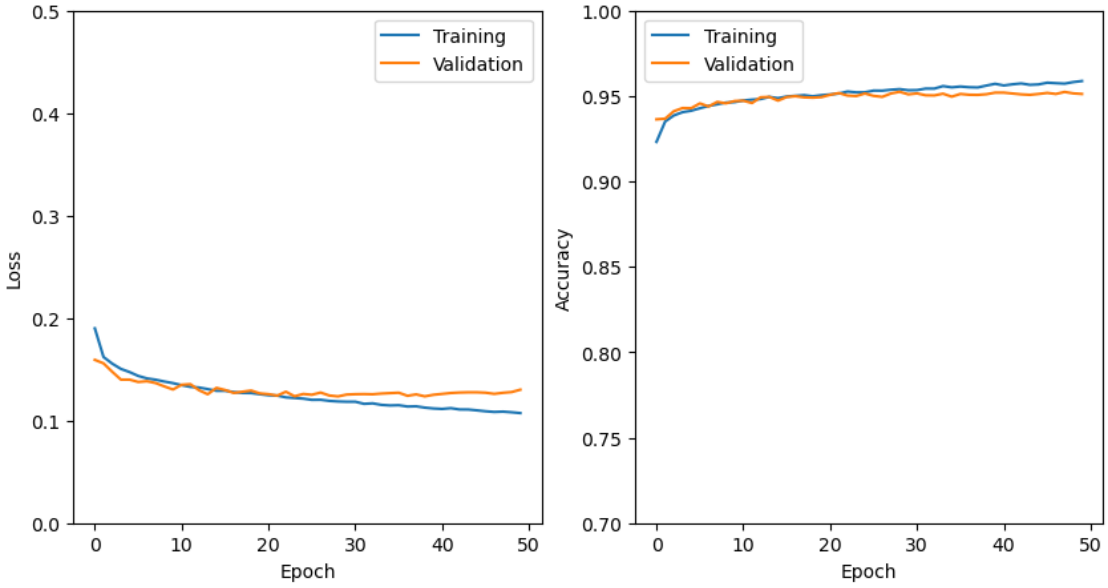
- Hidden layers: 256 -> 128 -? 128-> 64 nodes
- Activation: ELU
- Dropout: 0.2 on first 3 hidden layers
- Output: sigmoid
- Loss: Binary Crossentropy
- Optimizer: Adam

[86]:
```
inputs_paper = Input(shape=(X_train_transformed.shape[1],))

x_paper = Dense(256, activation='elu')(inputs_paper)
x_paper = Dropout(0.2)(x_paper)
x_paper = Dense(128, activation='elu')(x_paper)
x_paper = Dropout(0.2)(x_paper)
x_paper = Dense(128, activation='elu')(x_paper)
x_paper = Dropout(0.2)(x_paper)
x_paper = Dense(64, activation='elu')(x_paper)
outputs_paper = Dense(1, activation='sigmoid')(x_paper)

model_paper = Model(inputs=inputs_paper, outputs=outputs_paper)

# Compile with Binary Crossentropy and Adam
model_paper.compile(optimizer="adam", loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

# Train the model
history_paper = model_paper.fit(X_train_transformed, y_train_filtered,␣
 ↪epochs=50, batch_size=128, validation_data=(X_test_transformed,␣
 ↪y_test_filtered), verbose=0)

# Get predictions and compute AUC
preds_paper = model_paper.predict(X_test_transformed)
auc_paper = roc_auc_score(y_test_filtered, preds_paper)

print(f"Paper Architecture AUC: {auc_paper:.6f}")
roc_plot_paper = plot_roc(y_test_filtered, preds_paper)
plt.show()
```

```
811/811             1s 2ms/step
Paper Architecture AUC: 0.987984
```

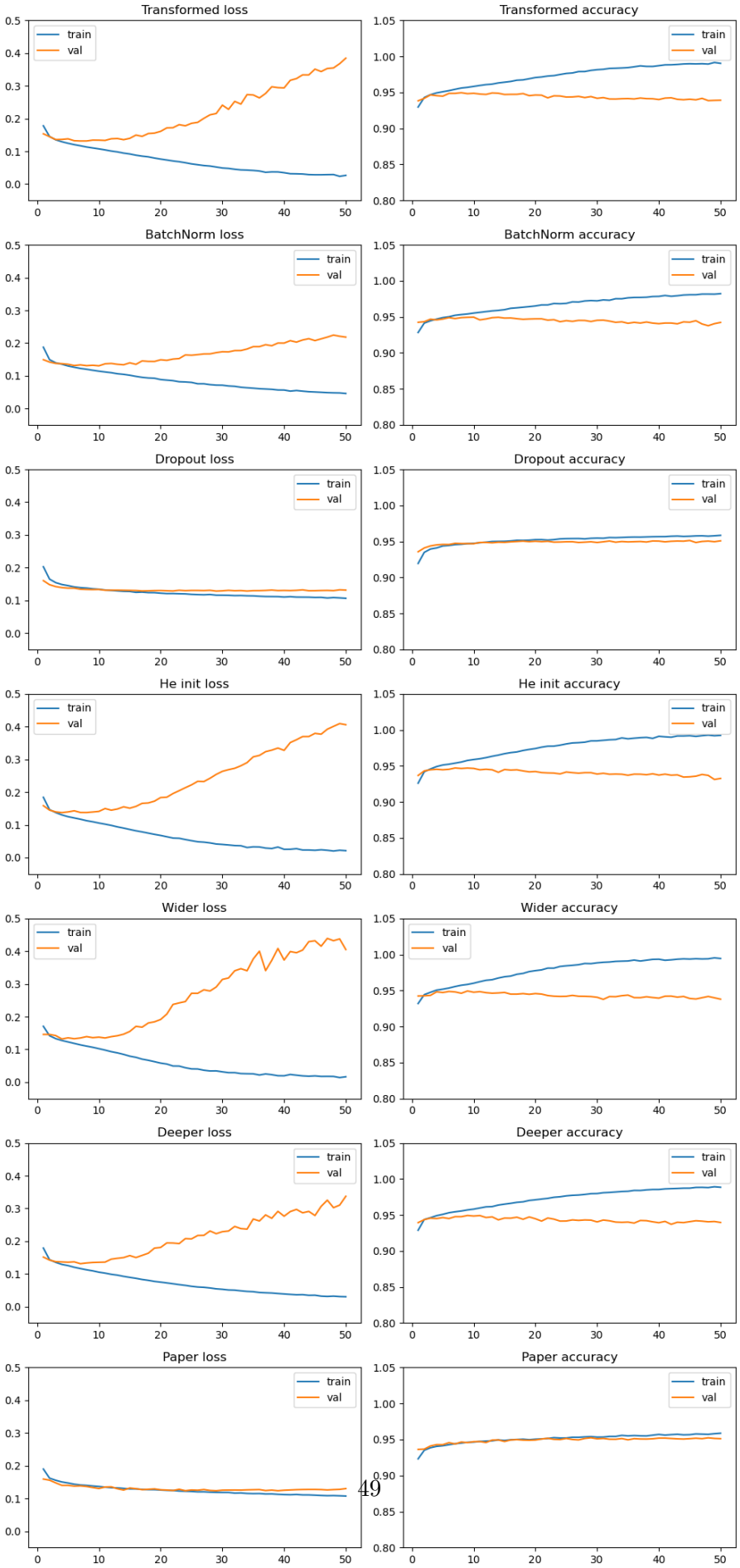


[87]:
```
plot_model_history(history_paper)
```



3.0.7 Compare Optional Experiments

[88]:
```
models = [
    ("Transformed",model_qu, history_qu),
    ("BatchNorm", model_bn, history_bn),
    ("Dropout", model_do, history_do),
    ("He init", model_he, history_he),
    ("Wider", model_wide, history_wide),
    ("Deeper", model_deep, history_deep),
    ("Paper", model_paper, history_paper)
]

# Plot train/val loss and accuracy for each experiment
fig, axes = plt.subplots(len(models), 2, figsize=(10, 3 * len(models)))
for i, (name, m, h) in enumerate(models):
    ep = range(1, len(h.history["loss"]) + 1)
    axes[i, 0].plot(ep, h.history["loss"], label="train")
    axes[i, 0].plot(ep, h.history["val_loss"], label="val")
    axes[i, 0].set_ylim(-0.05,0.5)
    axes[i, 0].set_title(f"{name} loss"); axes[i, 0].legend()

    axes[i, 1].plot(ep, h.history["accuracy"], label="train")
    axes[i, 1].plot(ep, h.history["val_accuracy"], label="val")
    axes[i, 1].set_ylim(0.8, 1.05)
    axes[i, 1].set_title(f"{name} accuracy"); axes[i, 1].legend()

plt.tight_layout()
plt.show()

print("Train AUC  |  Test AUC  |  Gap   | Model")
for name, m, h in models:
    preds_train = m.predict(X_train_transformed, verbose=0)
    preds_test  = m.predict(X_test_transformed, verbose=0)
    auc_train = roc_auc_score(y_train_filtered, preds_train)
    auc_test  = roc_auc_score(y_test_filtered,  preds_test)
    print(f"{auc_train:.4f}    {auc_test:.4f}    {auc_train-auc_test:.4f}  ␣
 ↪{name}")
```


Transformed loss
Transformed accuracy
BatchNorm loss
BatchNorm accuracy
Dropout loss
Dropout accuracy
He init loss
He init accuracy
Wider loss
Wider accuracy
Deeper loss
Deeper accuracy
Paper loss
Paper accuracy
train
val

```
Train AUC  |  Test AUC  |  Gap   | Model
0.9523    0.9453    0.0070   Transformed
0.9272    0.9176    0.0096   BatchNorm
0.9626    0.9598    0.0029   Dropout
0.9263    0.9193    0.0069   He init
0.9386    0.9303    0.0083   Wider
0.9607    0.9558    0.0049   Deeper
0.9940    0.9880    0.0060   Paper
```

Sorted by Test AUC (best to worst):

Paper

Dropout

Deeper

Transformed

He init

BatchNorm

Wider

Surprise surprise, the paper is the best, boring.

Interestingly, in the paper they get a 0.952 in accuracy and I have a 0.951, pretty close I might say.

From the paper: "We train with a learning rate of 0.001 for 45 epochs, starting at a batch size of 32 and doubling it every 5 epochs, ending at a batch size of 8192. We then train for an additional 50 epochs at batch size 8192 with decaying learning rate to fine tune the weights around the found minimum."

Which is something that I did not investigate as I did not set up my GPU environment yet.