DLiP - Assignment 2: Decision trees and Neural nets

Bradley Spronk - s2504758

February 19, 2026

1 GitHub experience

I am already familiar with Github. I have experience with creating repositories, branching, merging, and using push/pull requests. Link to a GitHub repository, or my GitHub profile.

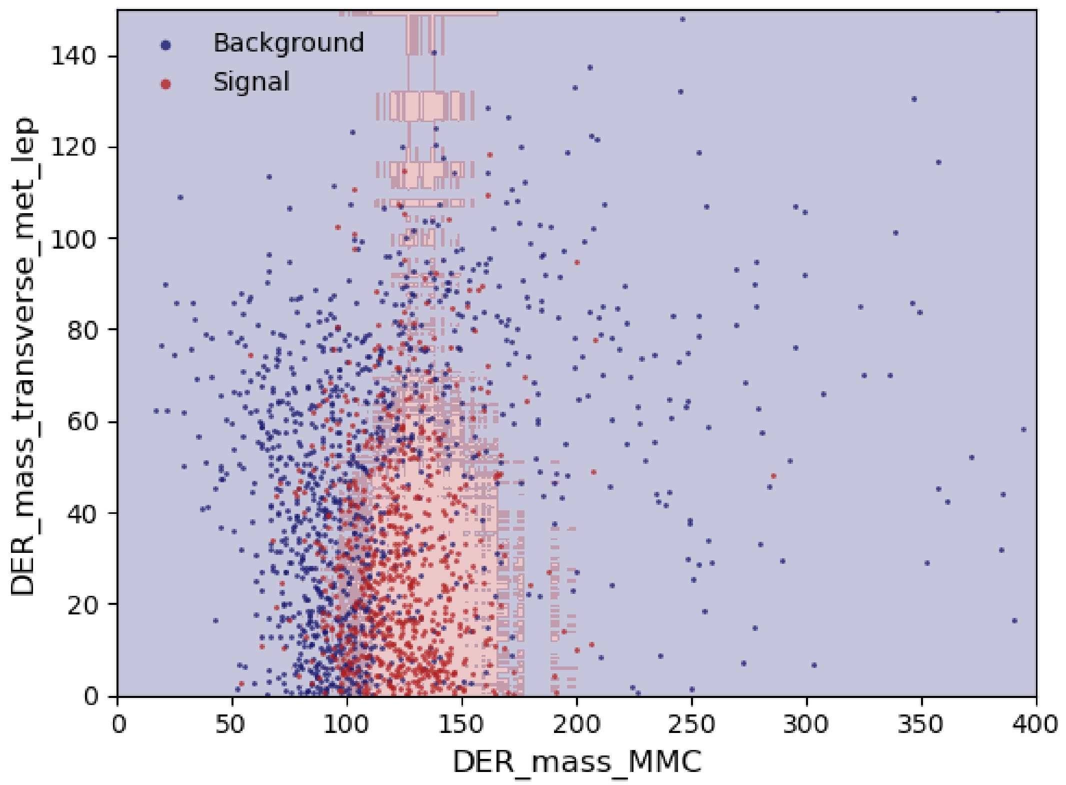
1.2 Exercise: Retrain BDT with 2 features and plot decision boundary

Repeat the steps above but manually set the features to just two of the most "important" ones: `feature_names = ["DER_mass_MMC", "DER_mass_transverse_met_lep"]`

Then use the code below to plot the decision boundary in 2D. 1) What do you notice about the shape of the decision boundary? 2) Do you see any evidence of overfitting? How can you prove it? (Hint: consider the training data)

[20]:
```
feature_names = ["DER_mass_MMC", "DER_mass_transverse_met_lep"]

train = xgb.DMatrix(
    data=data_train[feature_names], label=data_train.Label.cat.codes,␣
 ↪missing=-999.0, feature_names=feature_names
)

test = xgb.DMatrix(
    data=data_test[feature_names], label=data_test.Label.cat.codes,␣
 ↪missing=-999.0, feature_names=feature_names
)

booster = xgb.train(param, train, num_boost_round=num_trees)
```

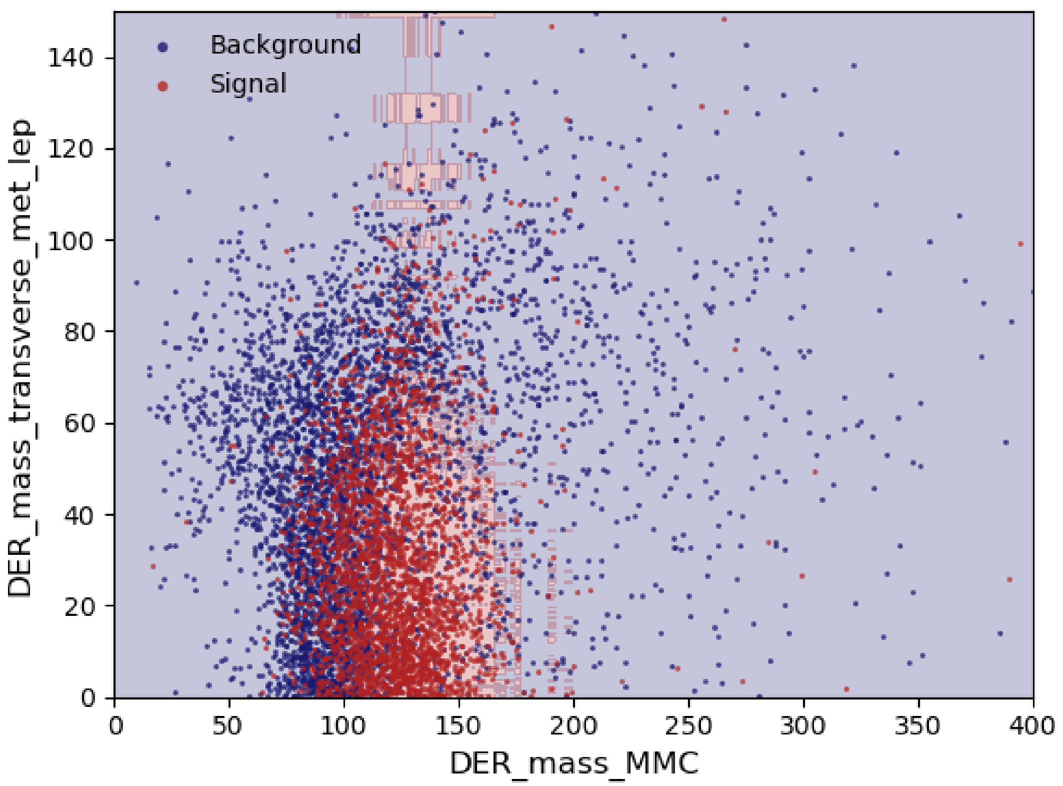
[21]:
```
from matplotlib.colors import ListedColormap

# first get a mesh grid
x_grid, y_grid = np.meshgrid(np.linspace(0, 400, 1000), np.linspace(0, 150,␣
 ↪1000))
# convert grid into DMatrix
matrix_grid = xgb.DMatrix(
    data=np.c_[x_grid.ravel(), y_grid.ravel()], missing=-999.0,␣
 ↪feature_names=feature_names
)
# run prediction for every value in grid
z_grid = booster.predict(matrix_grid)
# reshape
z_grid = z_grid.reshape(x_grid.shape)


plt.figure()
# plot decision boundary
ax = plt.subplot(111)
cm = ListedColormap(["midnightblue", "firebrick"])
plt.contourf(x_grid, y_grid, z_grid, levels=[0, 0.5, 1], cmap=cm, alpha=0.25)
# overlaid with test data points
plt.plot(
    DER_mass_MMC[mask_b],
    DER_mass_transverse_met_lep[mask_b],
    "o",
    markersize=2,
    color="midnightblue",
    markeredgewidth=0,
    alpha=0.8,
    label="Background",
)
plt.plot(
    DER_mass_MMC[mask_s],
    DER_mass_transverse_met_lep[mask_s],
    "o",
    markersize=2,
    color="firebrick",
    markeredgewidth=0,
    alpha=0.8,
    label="Signal",
)
ax.set_ylim(0,150)
ax.set_xlim(0,400)
plt.xlabel("DER_mass_MMC", fontsize=12)
plt.ylabel("DER_mass_transverse_met_lep", fontsize=12)
plt.legend(frameon=False, numpoints=1, markerscale=2)
plt.show()
```



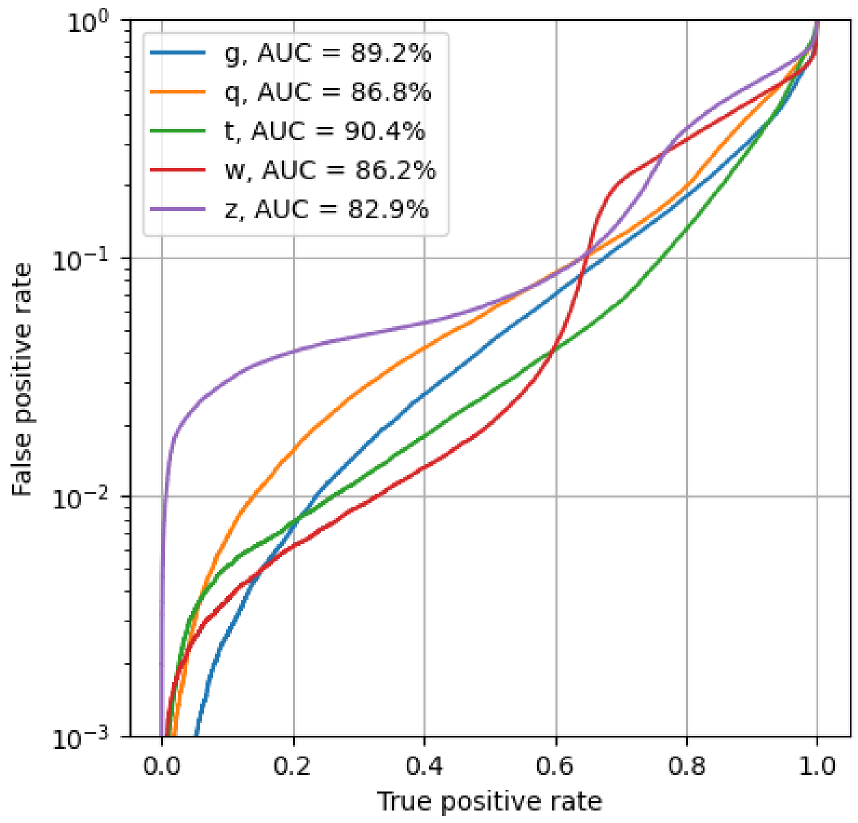
What I notice about the decision boundary is that it's made up of rectangles, not smooth at all. There is a concentrated mass around (x 100-150), (y 0-60). Which corresponds nicely with the 125 Gev mass of the Higgs Boson, fun fact.

I don't see real evidence of overfitting. Let's investigate a little:

[23]:
```python
mask_b = np.array(data_train.Label == "b")
mask_s = np.array(data_train.Label == "s")

DER_mass_MMC = np.array(data_train.DER_mass_MMC)
DER_mass_transverse_met_lep = np.array(data_train.DER_mass_transverse_met_lep)

plt.figure()
# plot decision boundary
ax = plt.subplot(111)
cm = ListedColormap(["midnightblue", "firebrick"])
plt.contourf(x_grid, y_grid, z_grid, levels=[0, 0.5, 1], cmap=cm, alpha=0.25)
# overlaid with test data points
plt.plot(
    DER_mass_MMC[mask_b],
    DER_mass_transverse_met_lep[mask_b],
    "o",
    markersize=2,
    color="midnightblue",
    markeredgewidth=0,
    alpha=0.8,
    label="Background",
)
plt.plot(
    DER_mass_MMC[mask_s],
    DER_mass_transverse_met_lep[mask_s],
    "o",
    markersize=2,
    color="firebrick",
    markeredgewidth=0,
    alpha=0.8,
    label="Signal",
)
ax.set_ylim(0,150)
ax.set_xlim(0,400)
plt.xlabel("DER_mass_MMC", fontsize=12)
plt.ylabel("DER_mass_transverse_met_lep", fontsize=12)
plt.legend(frameon=False, numpoints=1, markerscale=2)
plt.show()
```



If there was widespread overfitting, I would expect to see a red dot in every rectangle. Looking at the plot with the training data instead of the test data, I can say that that is not the case. Therefore I proclaim: no overfitting!



1.5 Exercises

[40]:
```python
def run_model(name,
              X_training_data,
              y_training_data,
              X_test_data,
              y_test_data,
              optimizer="sgd",
              scaled=False,
              regularization=False,
              modelSummary=False):

    if scaled:
        scaler = StandardScaler()
        X_training_data = scaler.fit_transform(X_training_data)
        X_test_data = scaler.transform(X_test_data)

    model = Sequential(name=f"sequential1_{name}")
    model.add(Input(shape=(16,)))
    if regularization:
        model.add(Dense(64, name="fc1", kernel_regularizer=l1(0.01)))
    else:
        model.add(Dense(64, name="fc1"))
    model.add(Activation(activation="relu", name="relu1"))
    model.add(Dense(32, name="fc2"))
    model.add(Activation(activation="relu", name="relu2"))
    model.add(Dense(32, name="fc3"))
    model.add(Activation(activation="relu", name="relu3"))
    model.add(Dense(5, name="fc4"))
    model.add(Activation(activation="softmax", name="softmax"))

    if modelSummary:
        model.summary()

    model.compile(optimizer=optimizer, loss=["categorical_crossentropy"],␣
↪metrics=["accuracy"])
    history = model.fit(X_training_data, y_training_data, batch_size=1024,␣
↪epochs=50, validation_split=0.25, shuffle=True, verbose=0)

    plot_model_history(history)

    y_keras = model.predict(X_test_data, batch_size=1024, verbose=0)
    print(f"Accuracy: {accuracy_score(np.argmax(y_test_data, axis=1), np.
↪argmax(y_keras, axis=1))}")

    distribution_weights(model)

    plt.figure(figsize=(5, 5))
    plot_confusion_matrix(y_test_data, y_keras, classes=le.classes_,␣
↪normalize=True)

    plt.figure(figsize=(5, 5))
    make_roc(y_test_data, y_keras, le.classes_)

    return model
```

1. Apply a standard scaler to the inputs. How does the performance of the model change?

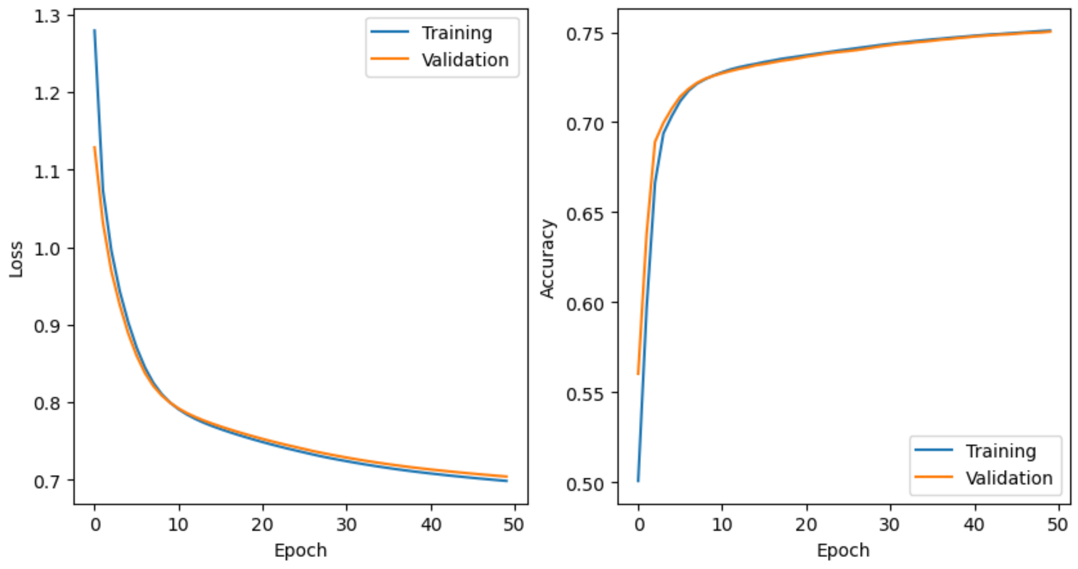
```
scaler = StandardScaler()
X_train_val = scaler.fit_transform(X_train_val)
X_test = scaler.transform(X_test)
```

[41]:
```
run_model("scaled", X_train_val, y_train_val, X_test, y_test, scaled=True)
```

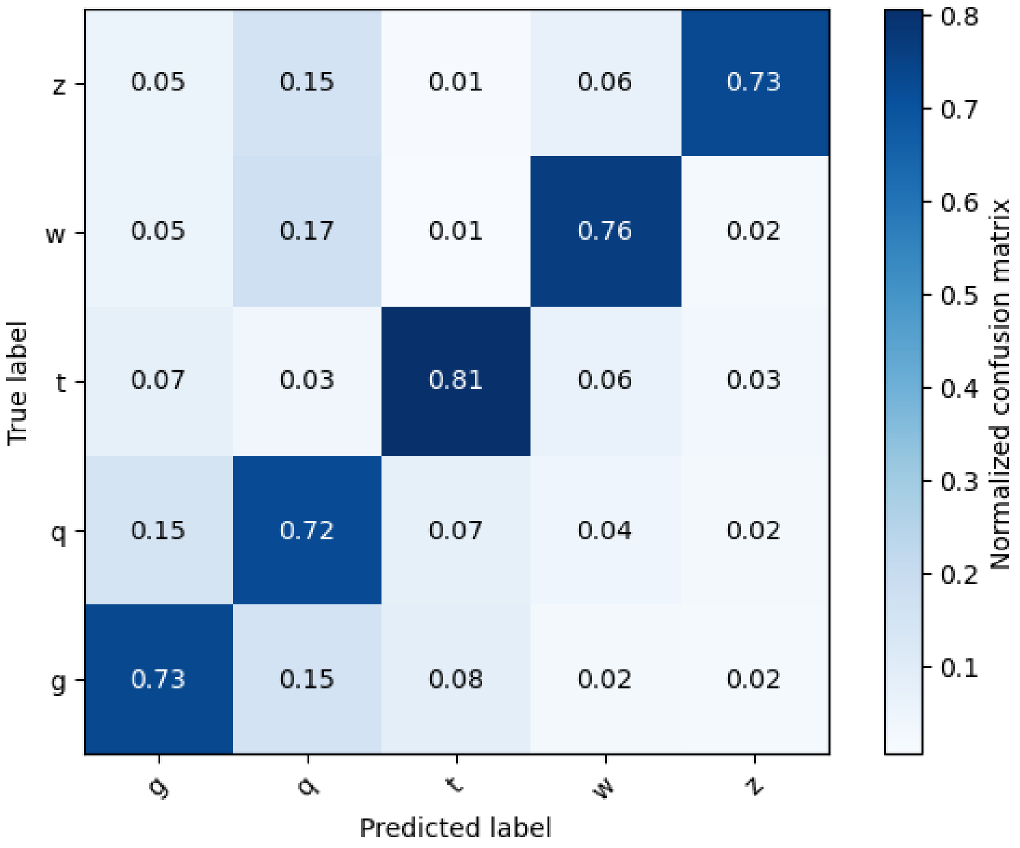
```
Accuracy: 0.7490060240963855
fc1 -0.9241199 0.81083953 -0.0036893745 0.19325085
fc2 -0.55273646 0.6438942 0.015476689 0.16474485
```

```
fc3 -0.71507347 0.71998227 0.010041023 0.21056752
fc4 -0.9602999 0.86939406 0.0074727335 0.38576978
```

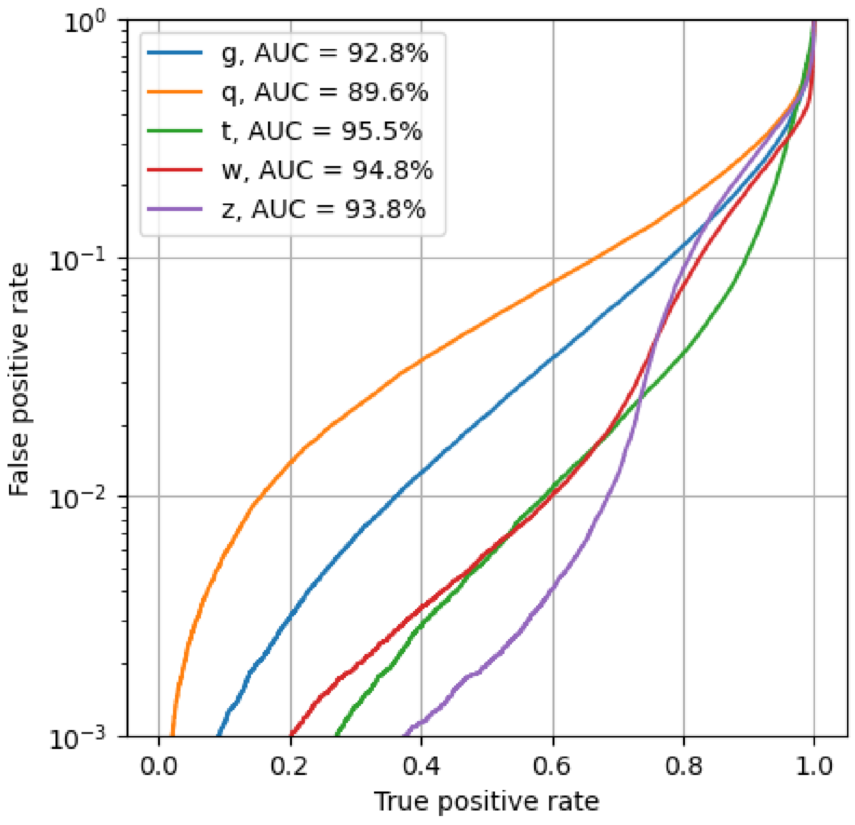
```
[41]: <Sequential name=sequential1_scaled, built=True>
```



```
<Figure size 500x500 with 0 Axes>
```



True label
z
w
t
q
g
0.05
0.15
0.01
0.06
0.73
0.05
0.17
0.01
0.76
0.02
0.07
0.03
0.81
0.06
0.03
0.15
0.72
0.07
0.04
0.02
0.73
0.15
0.08
0.02
0.02
g
q
t
w
z
Predicted label
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
Normalized confusion matrix



With the scaling, the loss and accuracy curves are very smooooth. Also the accuracy went way up, overall very good effect.

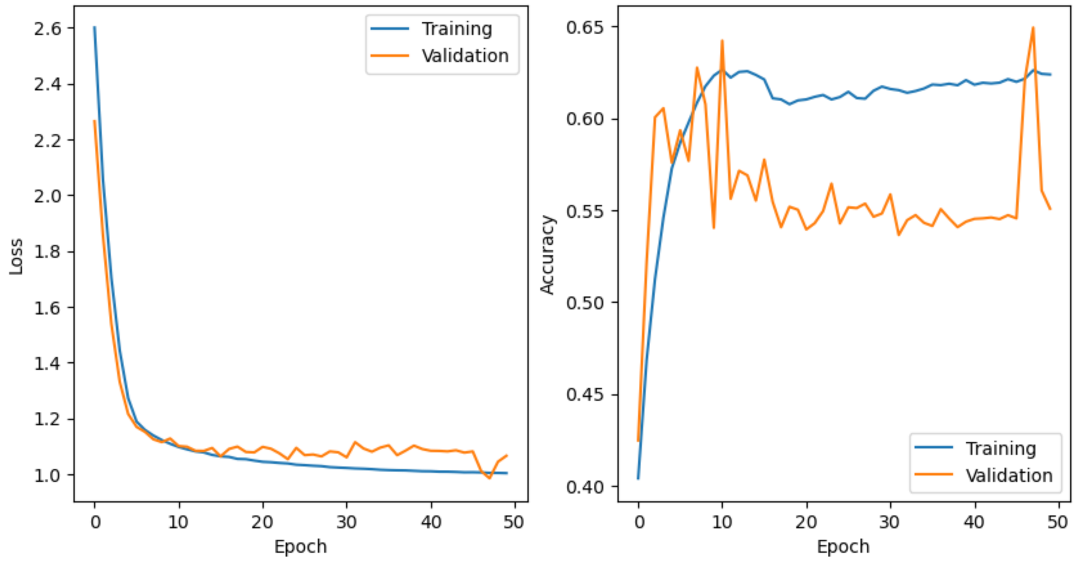
2. Apply L1 regularization. How does the performance of the model change? How do the distribution of the weight values change?

```
model.add(Dense(64, input_shape=(16,), name="fc1", kernel_regularizer=l1(0.01)))
```

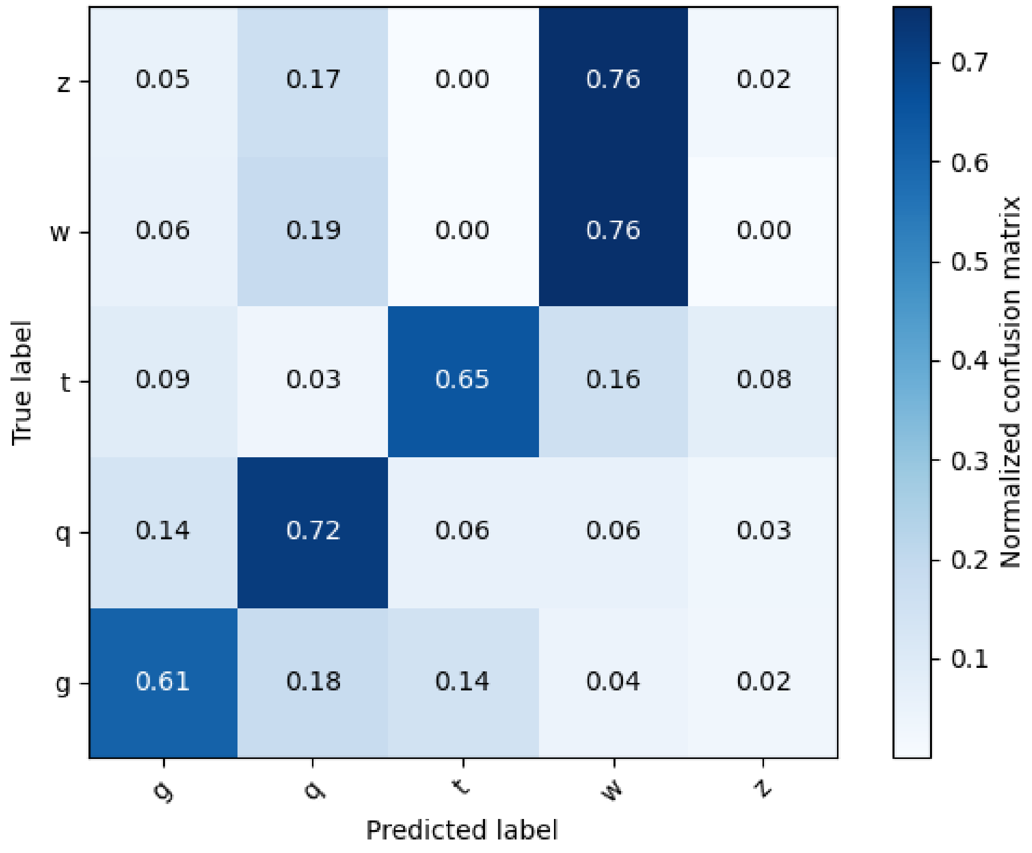
[42]:
```
run_model("L1", X_train_val, y_train_val, X_test, y_test, regularization=True)
```

```
Accuracy: 0.5503795180722891
fc1 -0.037136562 0.2775213 0.0007211752 0.013455774
fc2 -0.40801904 0.32105696 -0.002530575 0.14293677
fc3 -0.40351707 0.4430326 -0.0004476437 0.17798892
fc4 -1.9003155 0.9236936 0.015424743 0.38523453
```

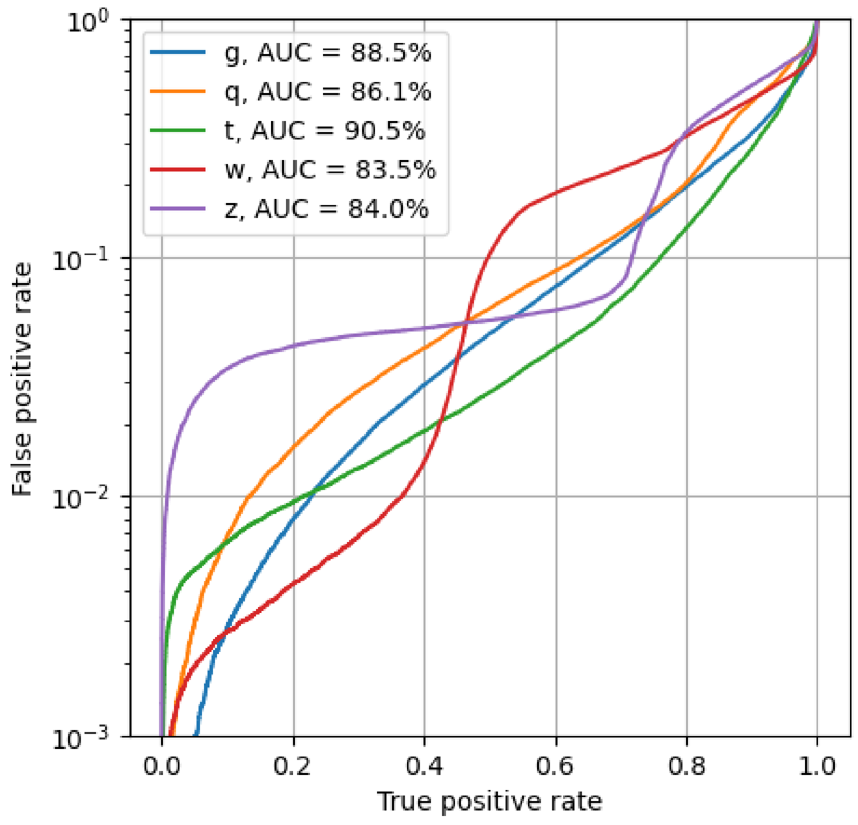
[42]:
```
<Sequential name=sequential1_L1, built=True>
```



```
<Figure size 500x500 with 0 Axes>
```



	g	q	t	w	z
z	0.05	0.17	0.00	0.76	0.02
w	0.06	0.19	0.00	0.76	0.00
t	0.09	0.03	0.65	0.16	0.08
q	0.14	0.72	0.06	0.06	0.03
g	0.61	0.18	0.14	0.04	0.02

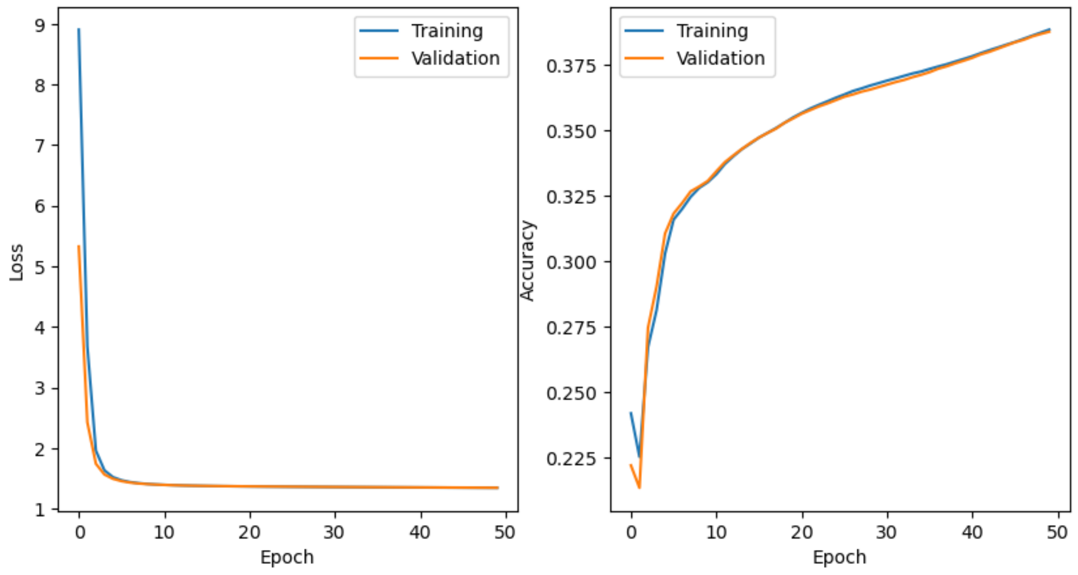


3. How do the loss curves change if we use a smaller learning rate (say `1e-5`) or a larger one (say `0.1`)?

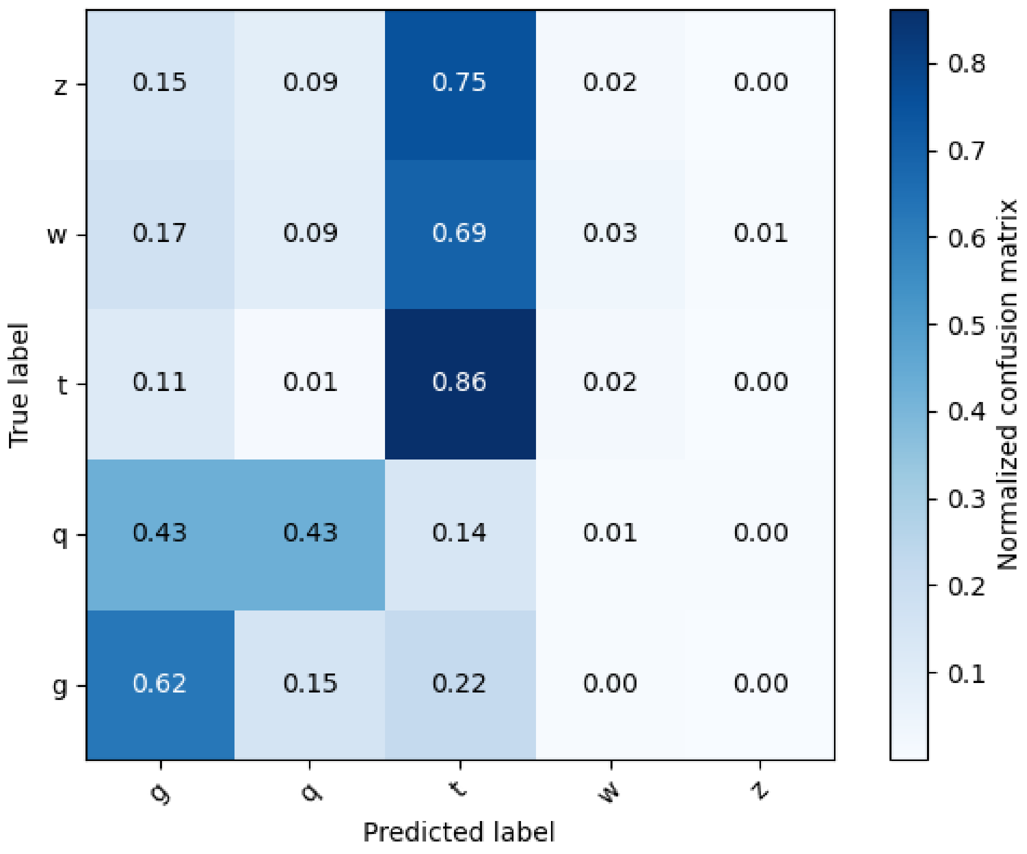
[43]:
```
from tensorflow.keras.optimizers import SGD

learning_rates = [1e-5, 0.1]

for lr in learning_rates:
    print(f"Training with learning rate: {lr}")
    model = run_model(f"lr_{lr}", X_train_val, y_train_val, X_test, y_test,␣
 ↪optimizer=SGD(learning_rate=lr))
    for layer in model.layers:
        w = layer.get_weights()
        if w:
            arr = w[0]
            print(layer.name, np.max(arr), np.mean(arr), np.min(arr))
```

```
Training with learning rate: 1e-05
Accuracy: 0.38946385542168677
```

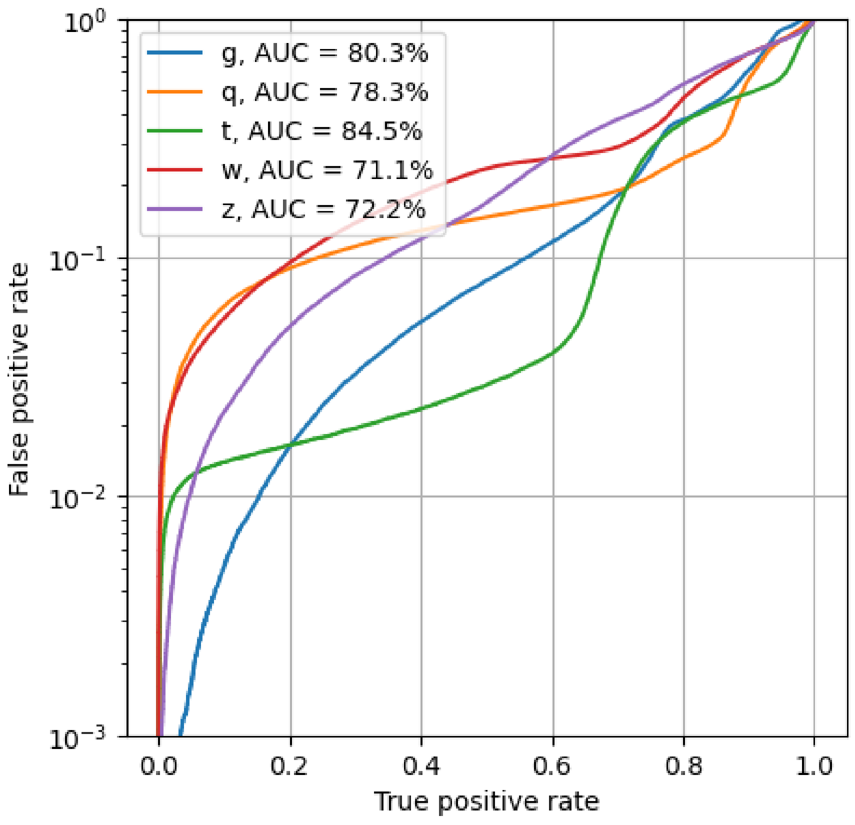
```
fc1 -0.27354676 0.27967808 0.0021745854 0.15698375
fc2 -0.26609194 0.26074687 0.0022924799 0.14444862
fc3 -0.30595315 0.32179144 0.0022396243 0.17677936
fc4 -0.41876158 0.40217695 0.0111032175 0.2319876
fc1 0.27967808 0.0021745854 -0.27354676
fc2 0.26074687 0.0022924799 -0.26609194
fc3 0.32179144 0.0022396243 -0.30595315
fc4 0.40217695 0.0111032175 -0.41876158
Training with learning rate: 0.1
Accuracy: 0.42093373493975905
fc1 -108.28281 2.8743262 -0.42084625 4.4976287
fc2 -182.4237 21.965376 -0.506625 6.6133
fc3 -97.48913 3.8975592 -1.481139 7.310942
fc4 -47.401962 13.3342705 0.007679033 7.1337285
fc1 2.8743262 -0.42084625 -108.28281
fc2 21.965376 -0.506625 -182.4237
fc3 3.8975592 -1.481139 -97.48913
fc4 13.3342705 0.007679033 -47.401962
```

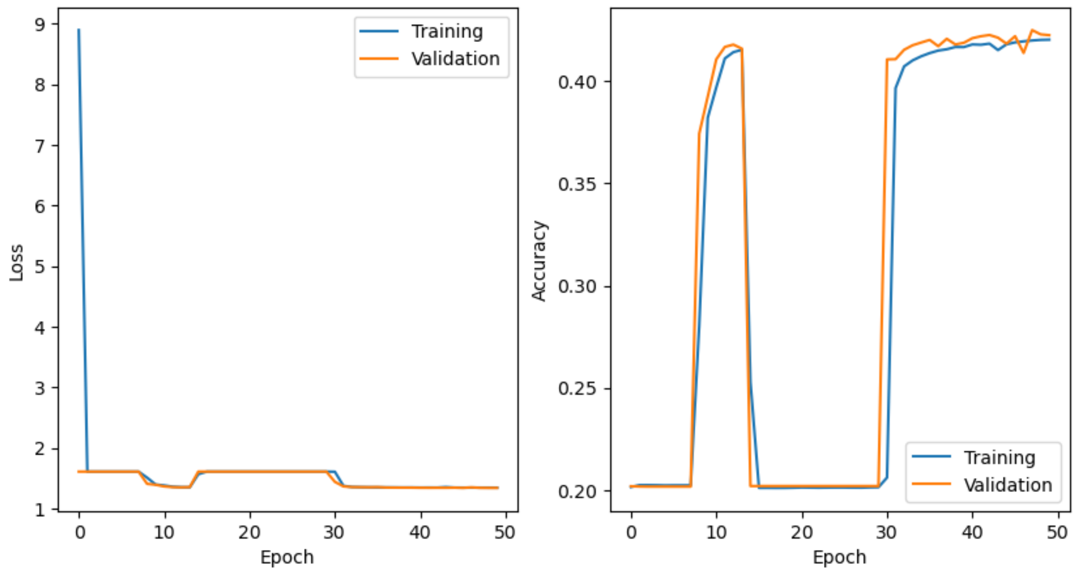


```
<Figure size 500x500 with 0 Axes>
```

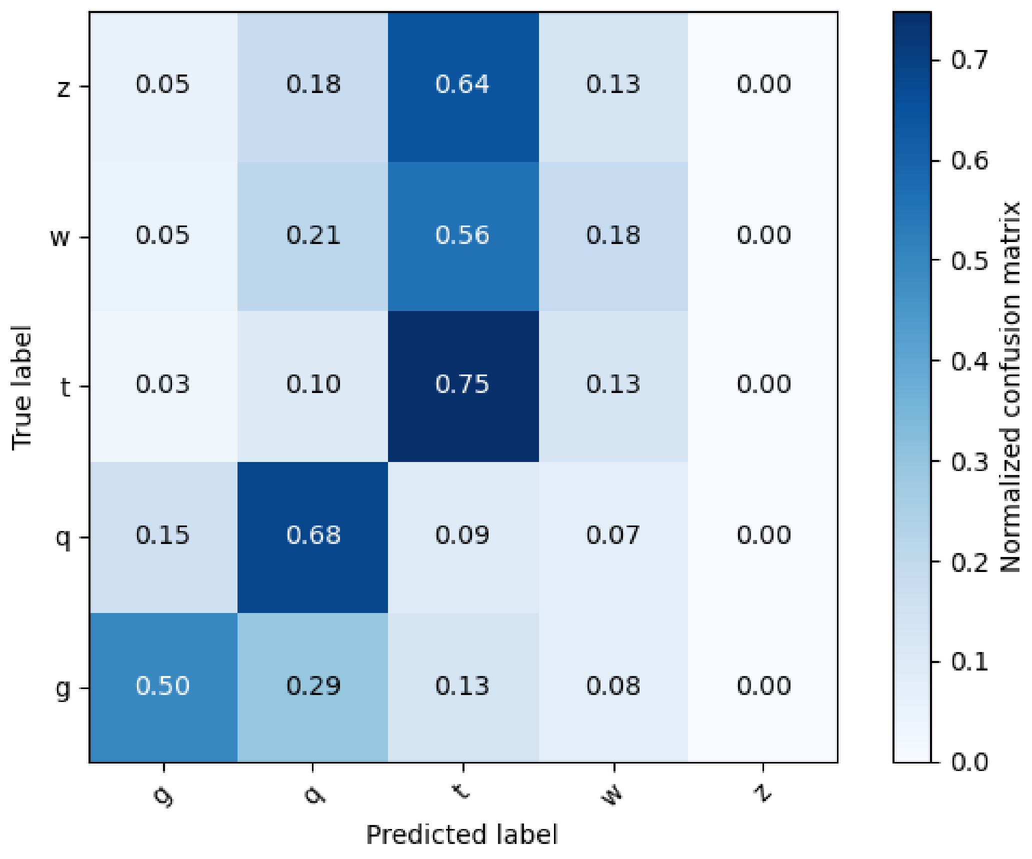


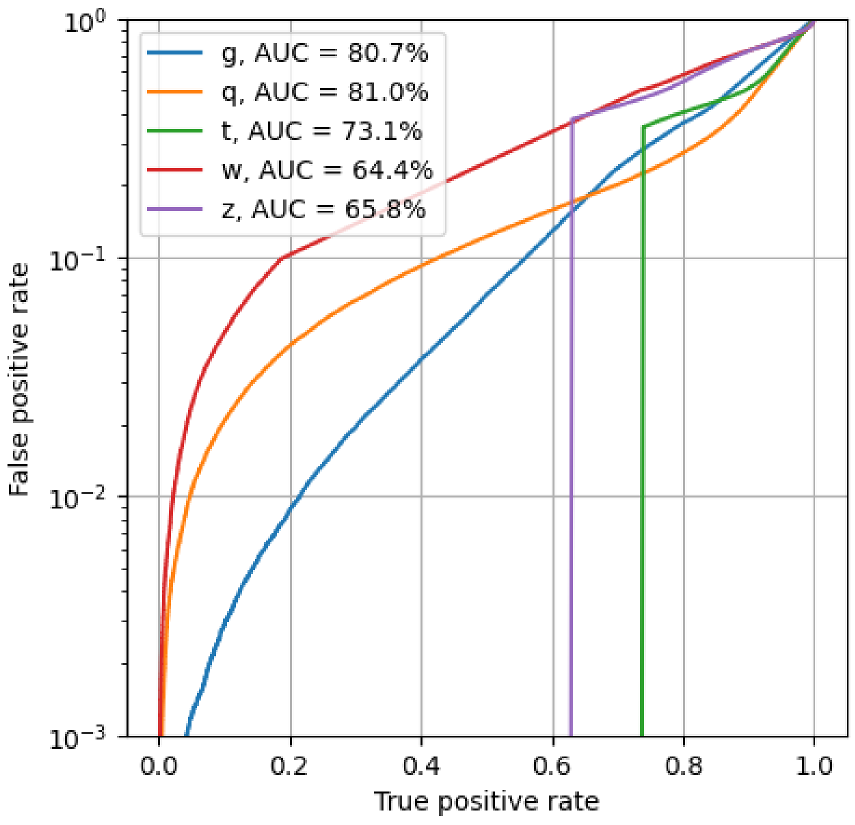
True label
Predicted label
Normalized confusion matrix
z
w
t
q
g
0.15 0.09 0.75 0.02 0.00
0.17 0.09 0.69 0.03 0.01
0.11 0.01 0.86 0.02 0.00
0.43 0.43 0.14 0.01 0.00
0.62 0.15 0.22 0.00 0.00
g
q
t
w
z
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1


g, AUC = 80.3%
q, AUC = 78.3%
t, AUC = 84.5%
w, AUC = 71.1%
z, AUC = 72.2%
False positive rate
True positive rate
10^0
10^{-1}
10^{-2}
10^{-3}
0.0
0.2
0.4
0.6
0.8
1.0


Training
Validation
Loss
Epoch
Accuracy
Epoch
Training
Validation

```
<Figure size 500x500 with 0 Axes>
```


True label
z
w
t
q
g
Predicted label
g
q
t
w
z
Normalized confusion matrix
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.05
0.18
0.64
0.13
0.00
0.05
0.21
0.56
0.18
0.00
0.03
0.10
0.75
0.13
0.00
0.15
0.68
0.09
0.07
0.00
0.50
0.29
0.13
0.08
0.00



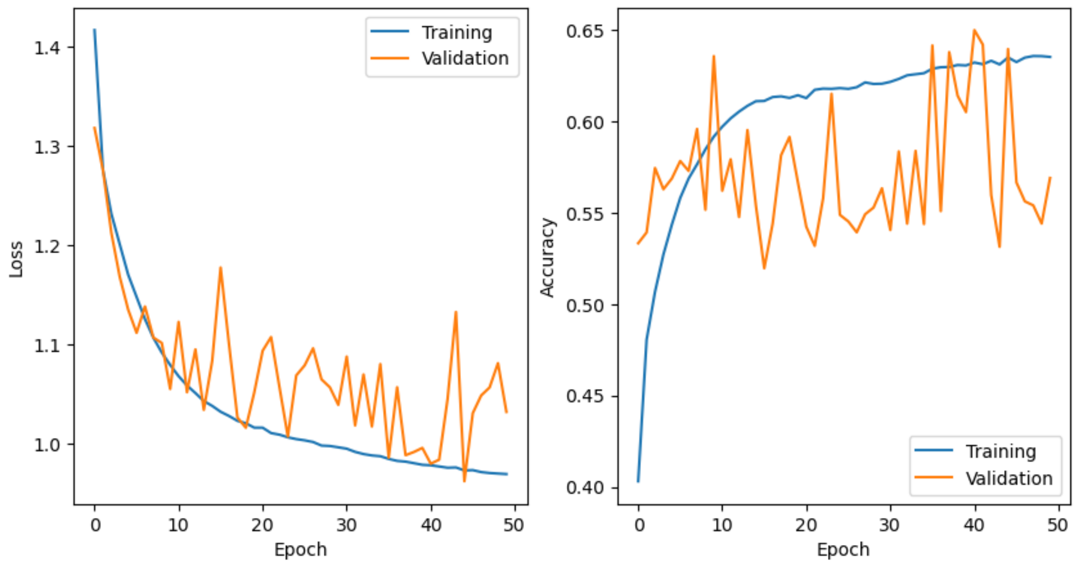
Both are quite shit. 1e-5 won't get anywhere, and with 0.1 the weights are exploding making w the answer to everything

4. How does the loss curve change and the performance of the model change if we use Adam as the optimizer instead of SGD?

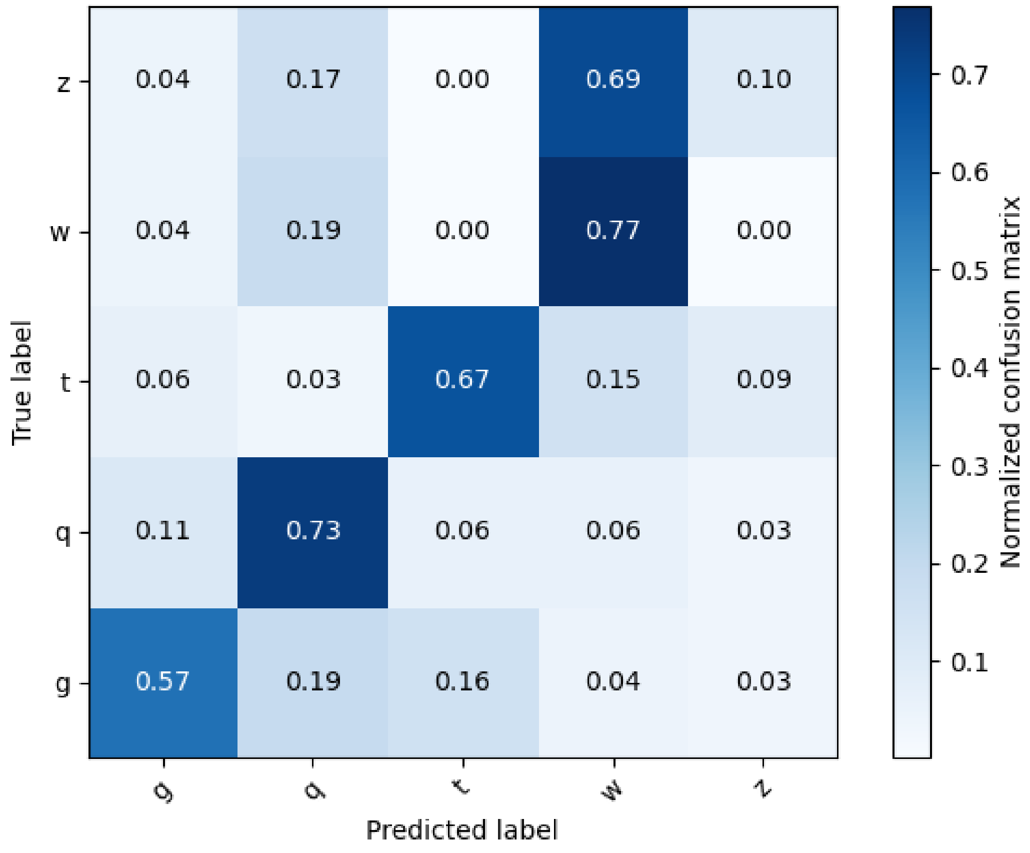
[44]:
```
run_model(f"sgd", X_train_val, y_train_val, X_test, y_test)
```

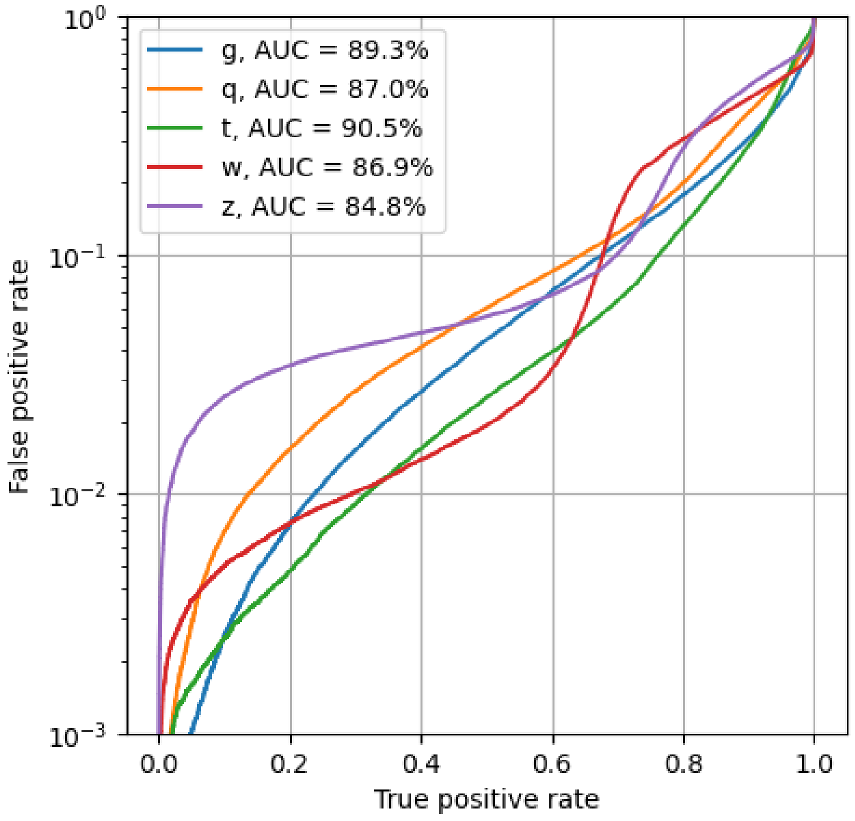
```
Accuracy: 0.5682349397590362
fc1 -1.1520971 1.4392871 -0.0061056023 0.19438897
fc2 -0.33207324 0.37656862 -0.0015244302 0.14497511
fc3 -0.56719625 0.5158807 -0.016675873 0.18554924
fc4 -1.537774 0.8938649 -0.0076303976 0.34872583
```

[44]:
```
<Sequential name=sequential1_sgd, built=True>
```


Loss
Accuracy
Epoch
Epoch
Training
Validation
Training
Validation

```
<Figure size 500x500 with 0 Axes>
```

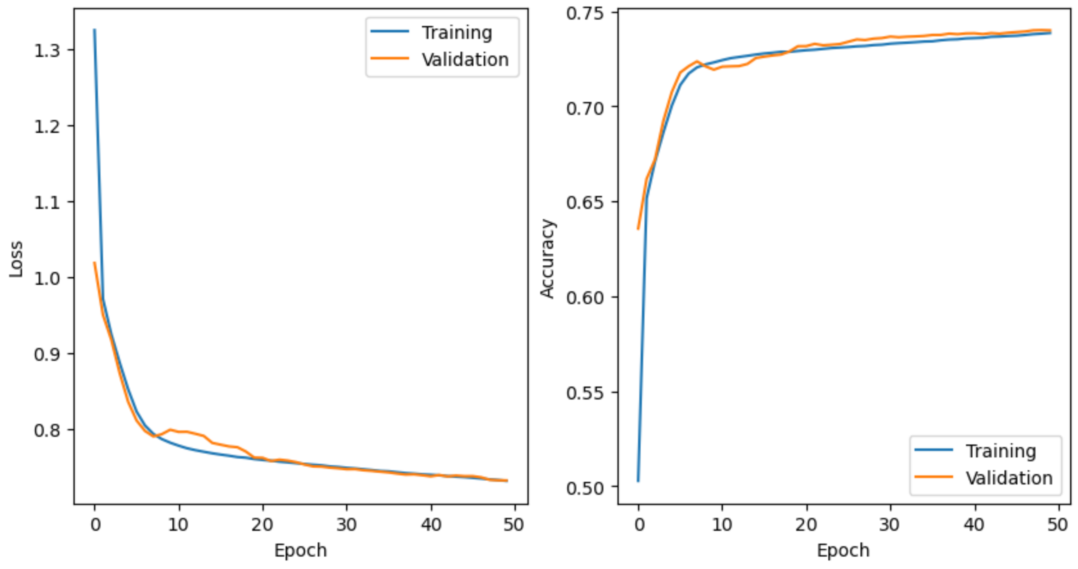

True label
Predicted label
Normalized confusion matrix
z
w
t
q
g
0.04 0.17 0.00 0.69 0.10
0.04 0.19 0.00 0.77 0.00
0.06 0.03 0.67 0.15 0.09
0.11 0.73 0.06 0.06 0.03
0.57 0.19 0.16 0.04 0.03
g
q
t
w
z



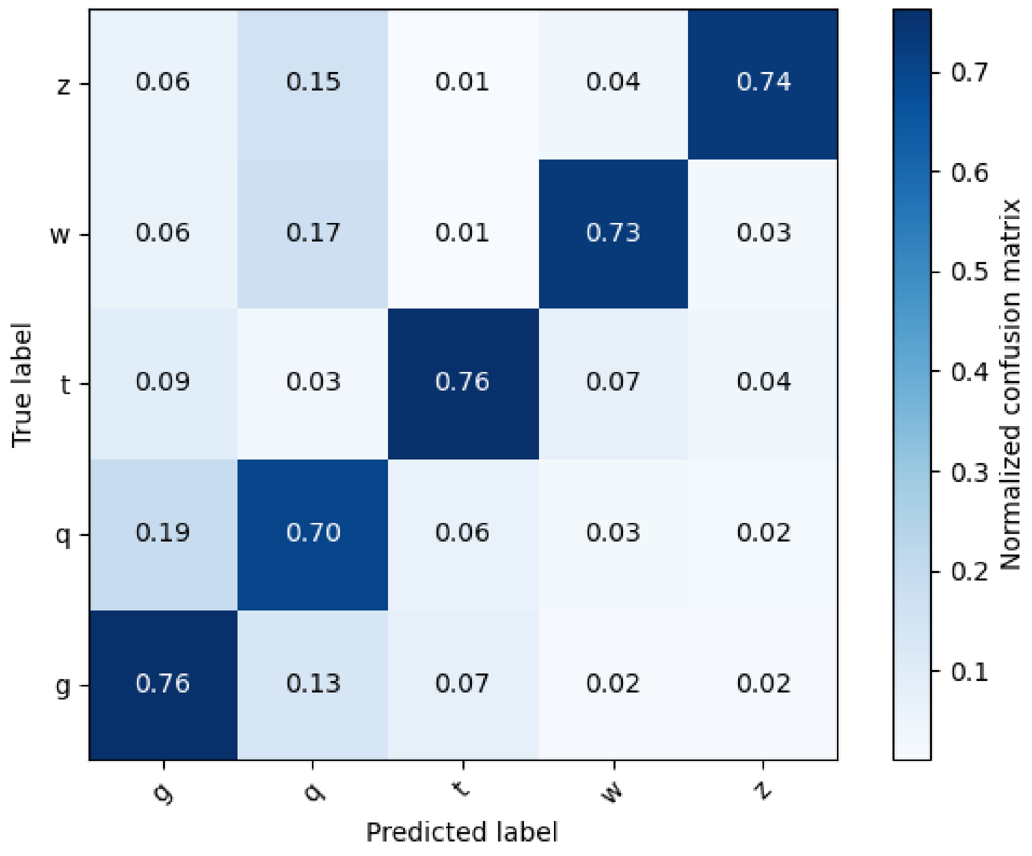
[45]:
```
run_model(f"adam", X_train_val, y_train_val, X_test, y_test, optimizer="adam")
```

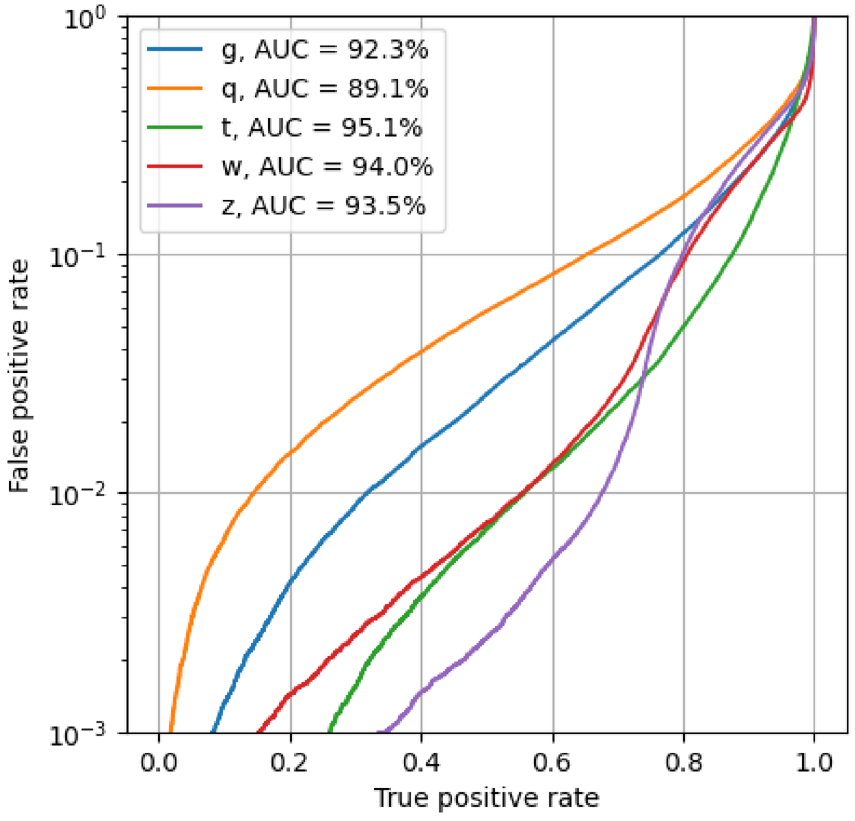
```
Accuracy: 0.7385
fc1 -9.025112 6.7606854 -0.03254625 1.3984452
fc2 -1.4220619 1.1046972 -0.010155535 0.18870592
fc3 -2.4307265 1.0565083 -0.014015345 0.27929738
fc4 -1.8018765 1.2599952 0.014330661 0.40919074
```

[45]:
```
<Sequential name=sequential1_adam, built=True>
```


Training
Validation
Loss
Epoch
Accuracy
Epoch
Training
Validation

```
<Figure size 500x500 with 0 Axes>
```


True label
Predicted label
Normalized confusion matrix
z
w
t
q
g
0.06 0.15 0.01 0.04 0.74
0.06 0.17 0.01 0.73 0.03
0.09 0.03 0.76 0.07 0.04
0.19 0.70 0.06 0.03 0.02
0.76 0.13 0.07 0.02 0.02



Very smooth, this is because Adam can change the learning rate based on the gradient. So we dont get the jaggedy curve like before and it reaches a pretty nice accuracy.

2 Neural nets vs BDTs on the MiniBOONE dataset.

3 Additional improvements