03_Tabular_Data_NN_Exercises_only

February 20, 2026

1 Hands-on 03: Tabular data and NNs: Classifying particle jets

1.1 Exercises

[15]:
```
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
```

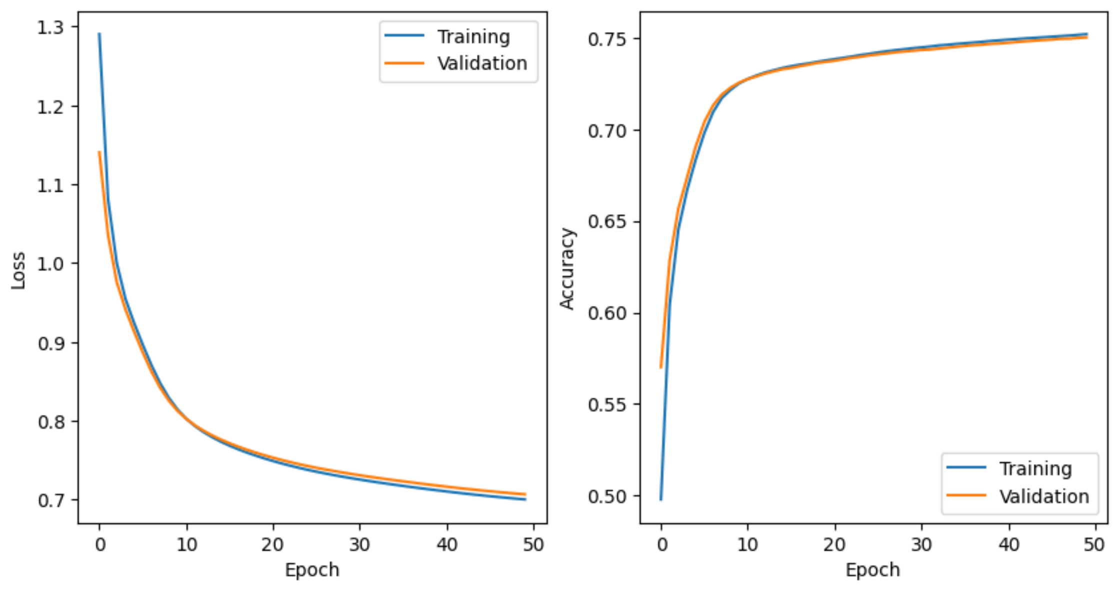
1. Apply a standard scaler to the inputs. How does the performance of the model change?

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

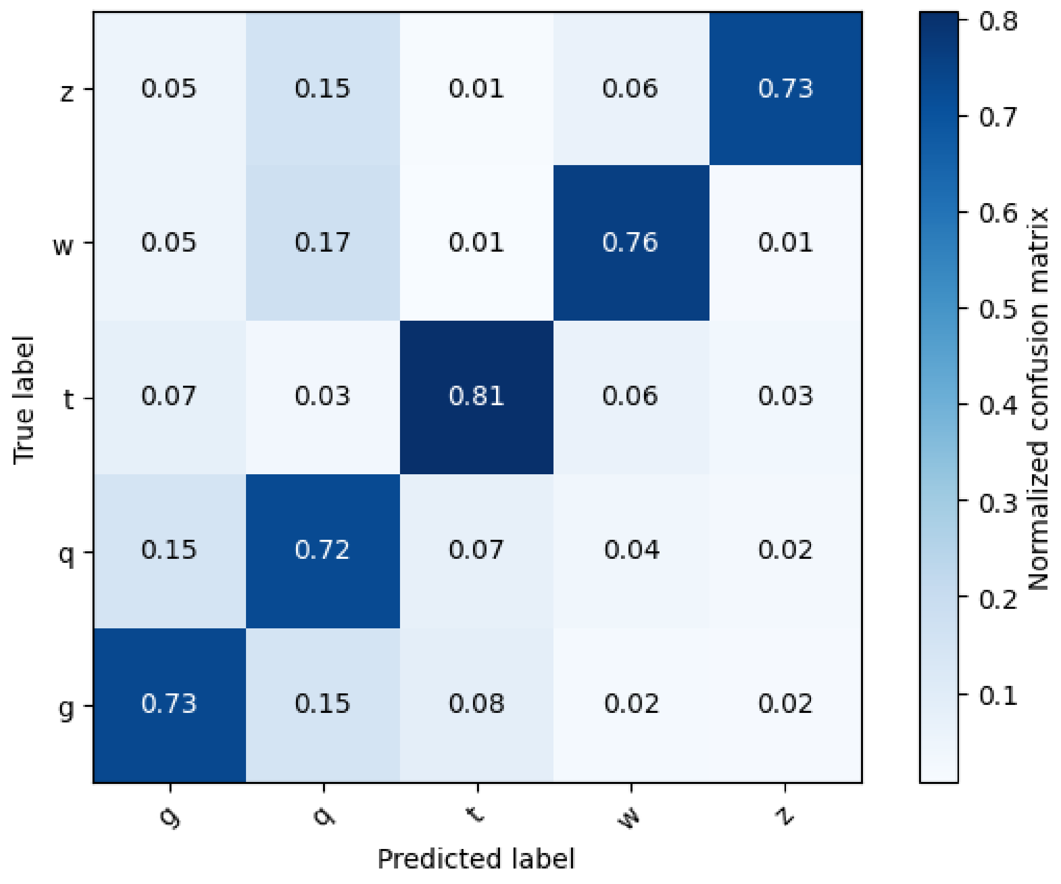
[16]:
```
run_model("scaled", X_train_val, y_train_val, X_test, y_test, scaled=True)
```

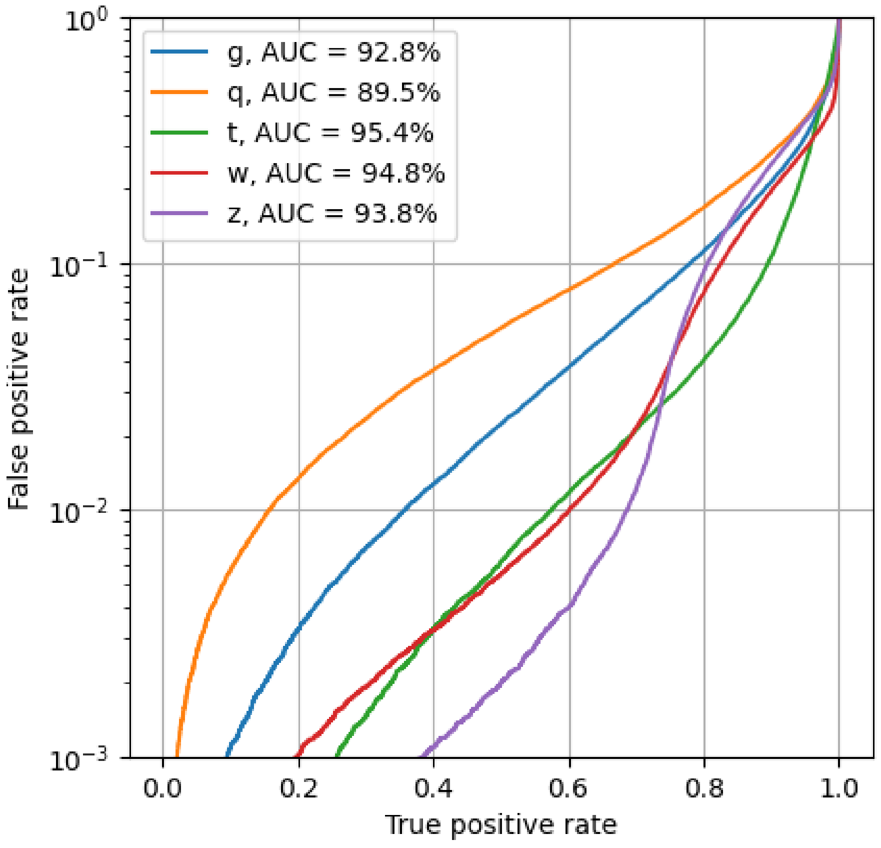
```
Accuracy: 0.7492048192771085
fc1 min:  -0.6972411 max:  1.4959888 mean:  -0.0047975243 std:  0.19467126
fc2 min:  -0.71080106 max:  0.47231993 mean:  0.0060272356 std:  0.16698979
fc3 min:  -0.5945778 max:  0.6200948 mean:  -0.0045326417 std:  0.21238191
fc4 min:  -1.0746753 max:  1.0037723 mean:  -0.014541777 std:  0.38414928
```

[16]:
```
<Sequential name=sequential1_scaled, 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
0.05 0.15 0.01 0.06 0.73
0.05 0.17 0.01 0.76 0.01
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



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

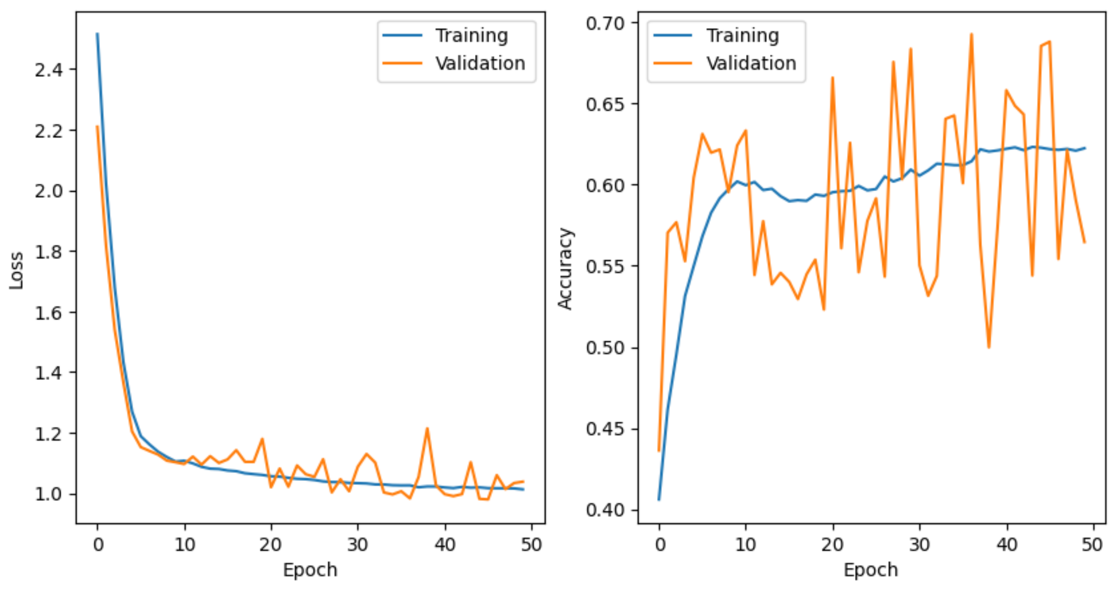
2. Apply L1 regularization. How does the performance of the model change? How do the distribution of the weight values change?

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

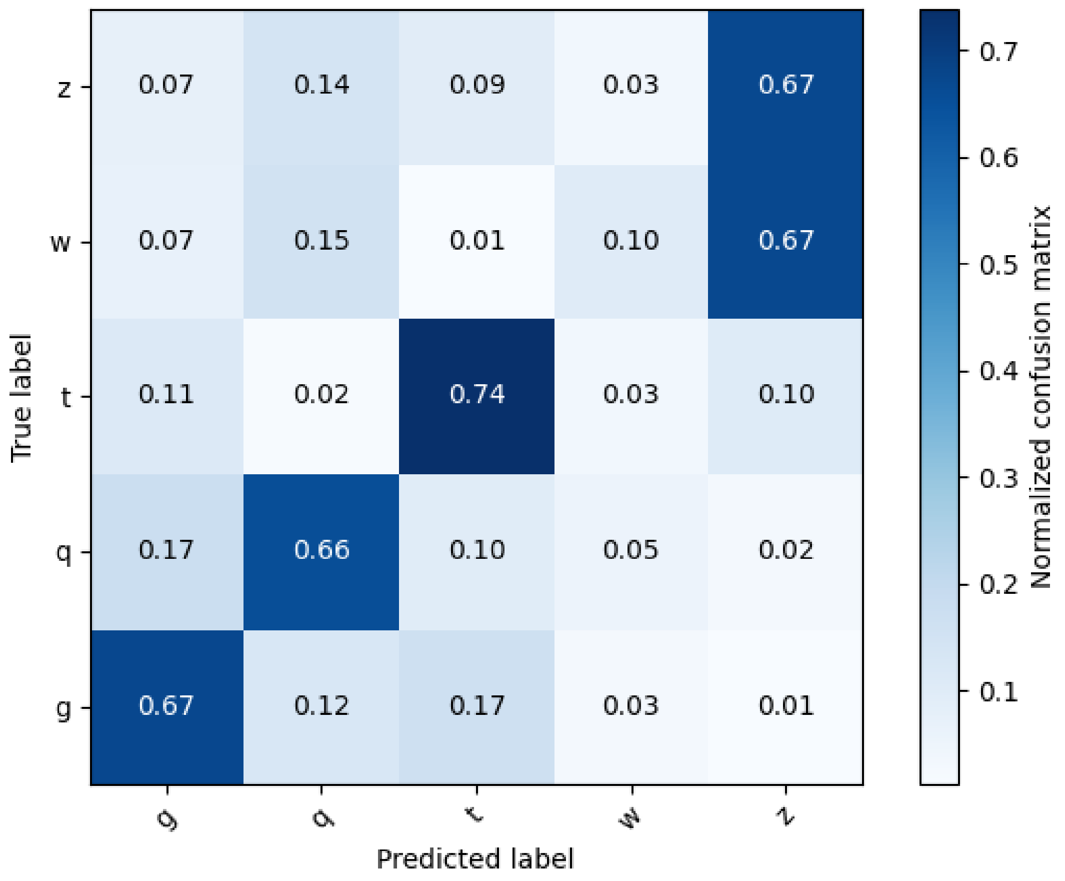
[ ]:
```
run_model("L1", X_train_val, y_train_val, X_test, y_test, regularization=True)
```

```
Accuracy: 0.5648795180722892
fc1 min:  -0.068717696 max:  0.23690452 mean:  0.0007591712 std:  0.012971567
fc2 min:  -0.37616462 max:  0.3731854 mean:  -0.0010794414 std:  0.1439574
fc3 min:  -0.5453241 max:  0.50437176 mean:  0.007048251 std:  0.18619554
fc4 min:  -1.1301262 max:  1.2366422 mean:  0.013635397 std:  0.36573339
```

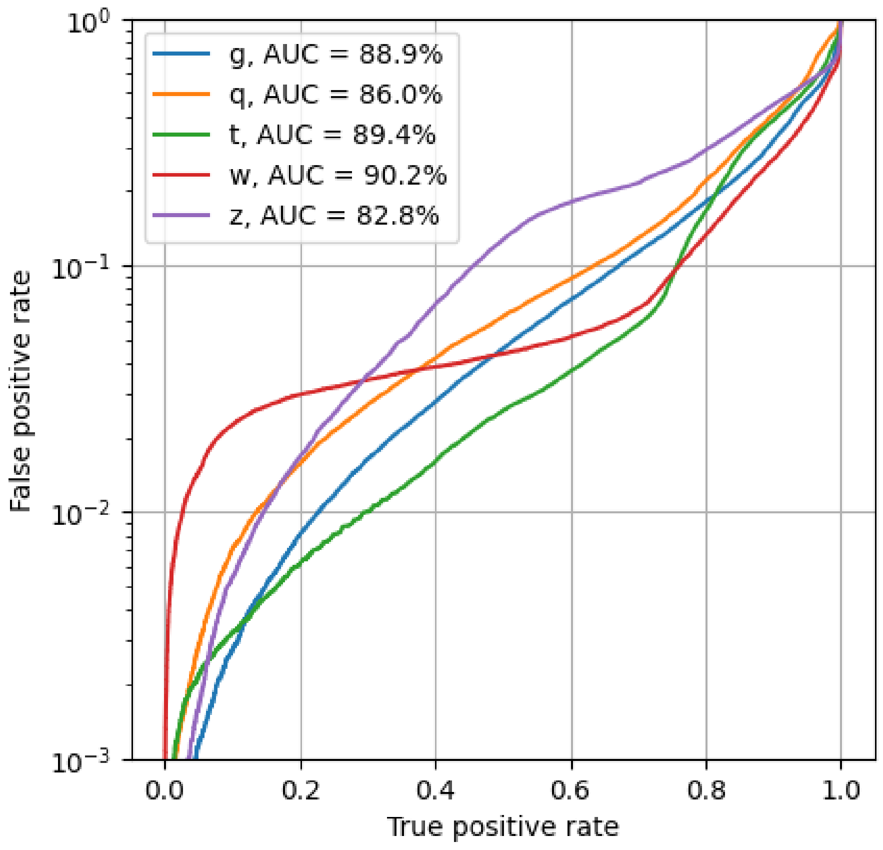
[ ]: `<Sequential name=sequential1_L1, built=True>`



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



	g	q	t	w	z
z	0.07	0.14	0.09	0.03	0.67
w	0.07	0.15	0.01	0.10	0.67
t	0.11	0.02	0.74	0.03	0.10
q	0.17	0.66	0.10	0.05	0.02
g	0.67	0.12	0.17	0.03	0.01

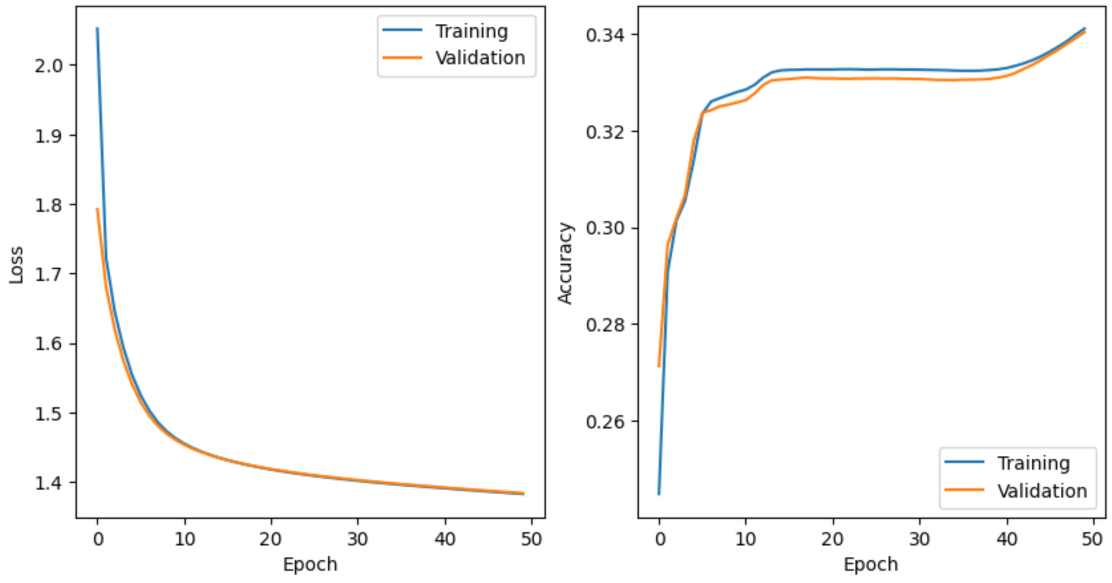


3. How do the loss curves change if we use a smaller learning rate (say `1e-5`) or a larger one (say `0.1`)?

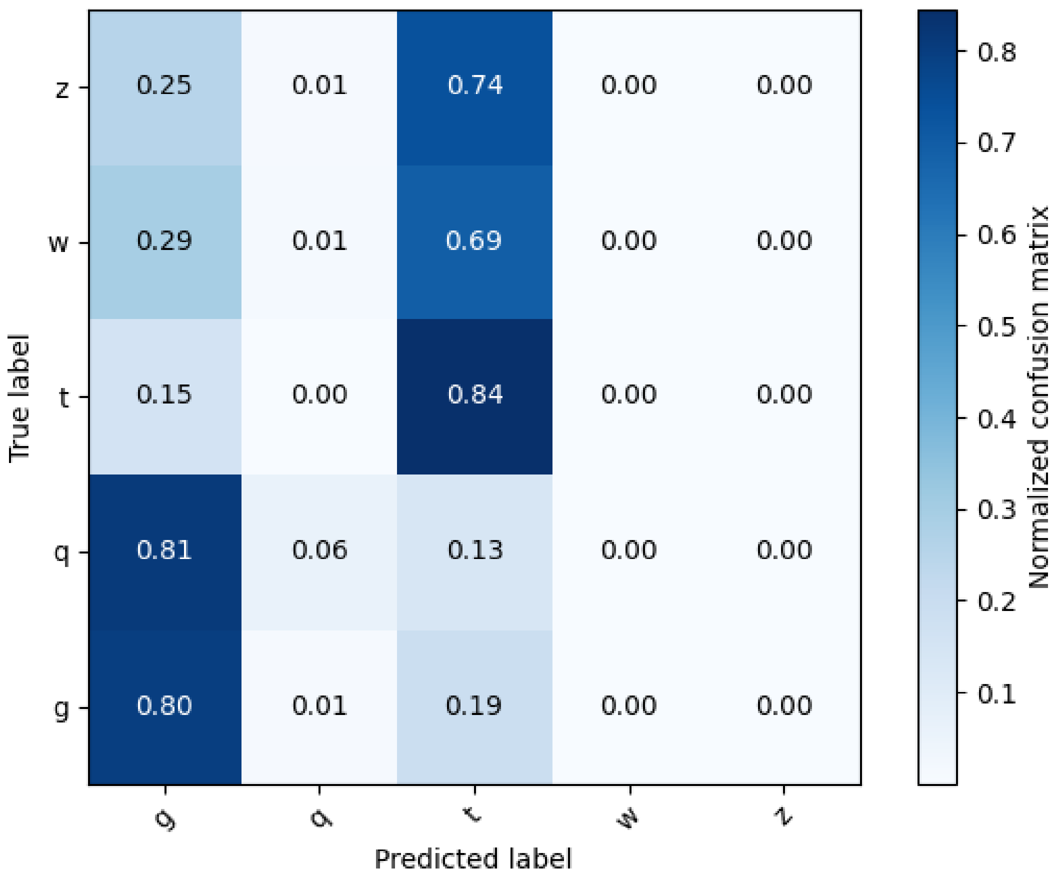
[ ]:
```
from tensorflow.keras.optimizers import SGD

learning_rates = [1e-5, 0.1]

for lr in learning_rates:
    model = run_model(f"lr_{lr}", X_train_val, y_train_val, X_test, y_test,␣
 ↪optimizer=SGD(learning_rate=lr))
```

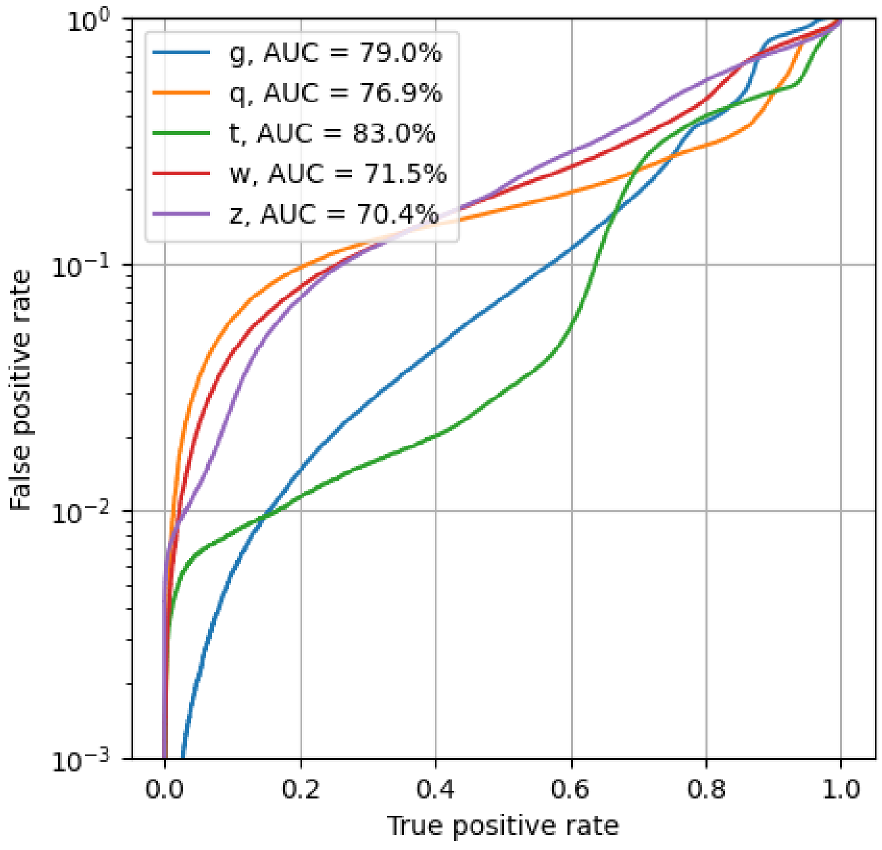
```
Accuracy: 0.3423975903614458
fc1 min:  -0.27380824 max:  0.2763283 mean:  0.0046199765 std:  0.15473199
fc2 min:  -0.25294265 max:  0.25226074 mean:  -0.00514176 std:  0.14175405
fc3 min:  -0.30614704 max:  0.3119812 mean:  -0.0036122734 std:  0.18058378
fc4 min:  -0.39139897 max:  0.39898956 mean:  -0.0038495972 std:  0.23330776
Accuracy: 0.2016566265060241
fc1 min:  -97525.484 max:  3356.983 mean:  -337.14032 std:  4304.066
fc2 min:  -3.7962168e+06 max:  216790.89 mean:  -3126.5369 std:  92918.516
```

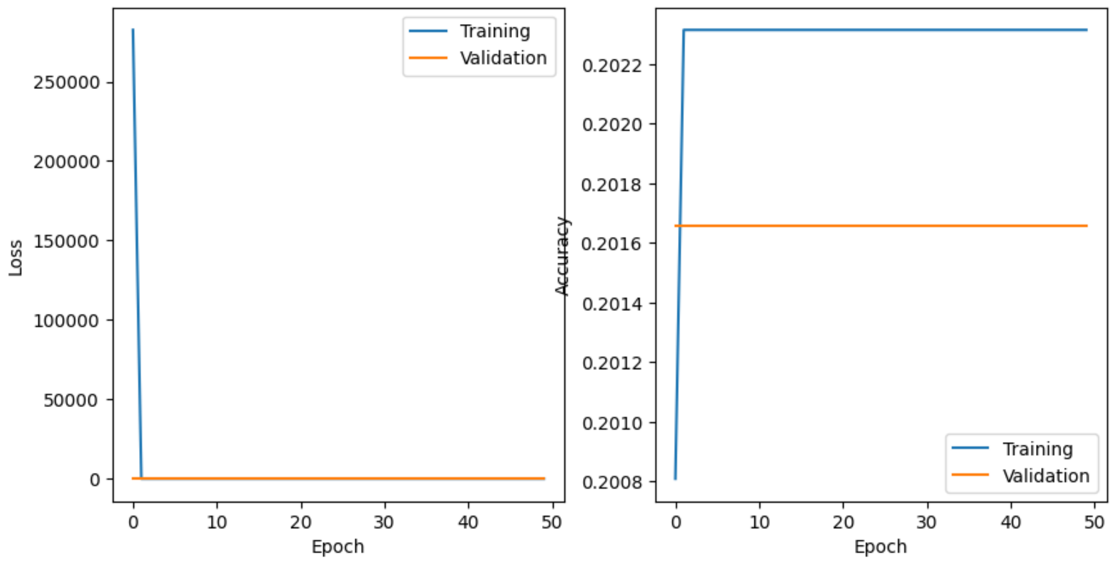
```
fc3 min:  -681894.4 max:  515.36865 mean:  -2621.2642 std:  28438.91
fc4 min:  -6748.86 max:  2229.5803 mean:  0.00068511965 std:  815.6741
```



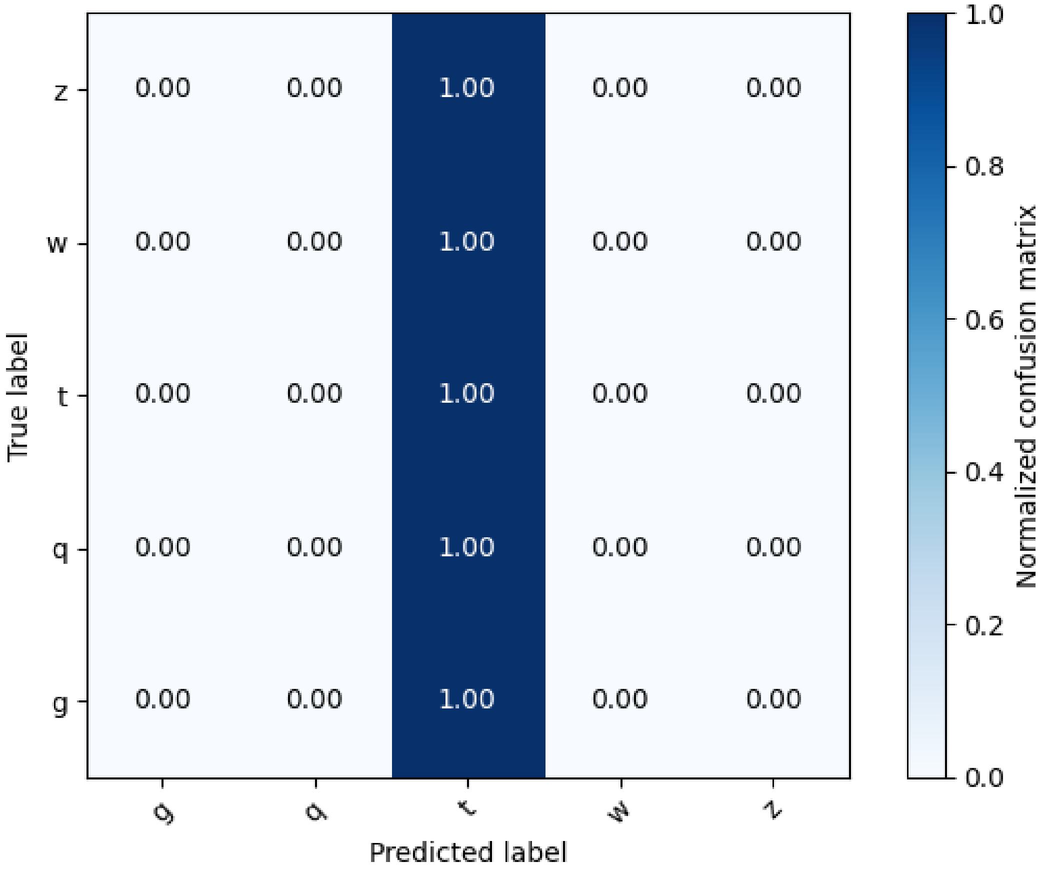
```
<Figure size 500x500 with 0 Axes>
```


True label
Predicted label
Normalized confusion matrix
z
w
t
q
g
0.25
0.01
0.74
0.00
0.00
0.29
0.01
0.69
0.00
0.00
0.15
0.00
0.84
0.00
0.00
0.81
0.06
0.13
0.00
0.00
0.80
0.01
0.19
0.00
0.00
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

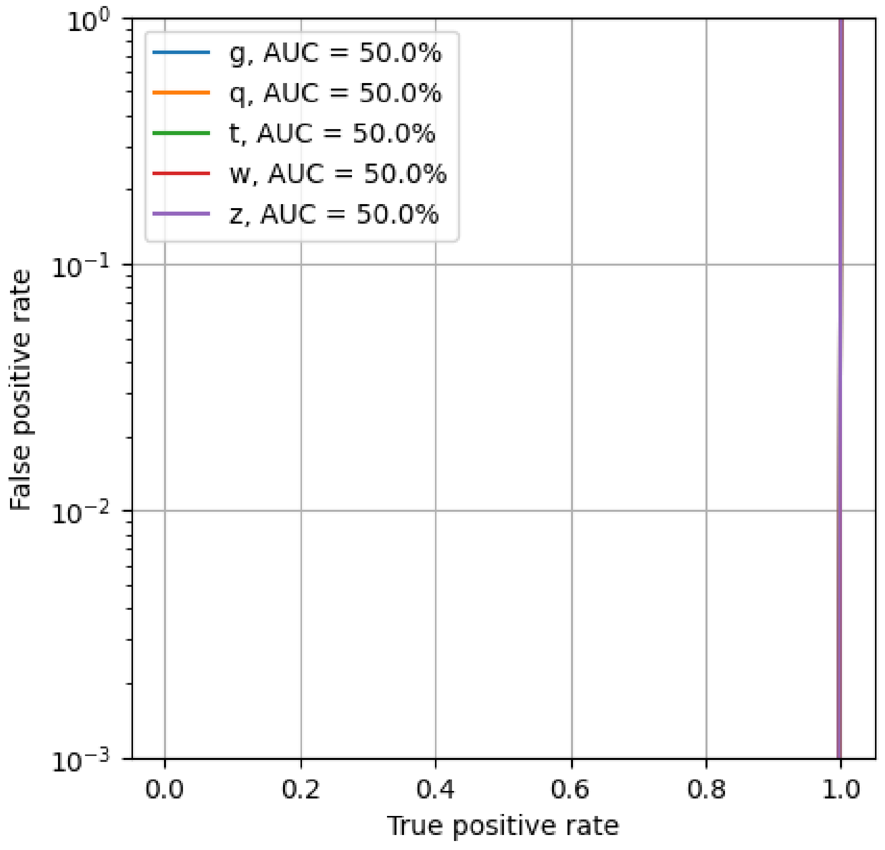

g, AUC = 79.0%
q, AUC = 76.9%
t, AUC = 83.0%
w, AUC = 71.5%
z, AUC = 70.4%
False positive rate
True positive rate
10^0
10^{-1}
10^{-2}
10^{-3}
0.0
0.2
0.4
0.6
0.8
1.0


Training
Validation
Loss
Epoch
Accuracy
Epoch
250000
200000
150000
100000
50000
0
0.2022
0.2020
0.2018
0.2016
0.2014
0.2012
0.2010
0.2008
0
10
20
30
40
50

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



True label \ Predicted label	g	q	t	w	z
z	0.00	0.00	1.00	0.00	0.00
w	0.00	0.00	1.00	0.00	0.00
t	0.00	0.00	1.00	0.00	0.00
q	0.00	0.00	1.00	0.00	0.00
g	0.00	0.00	1.00	0.00	0.00



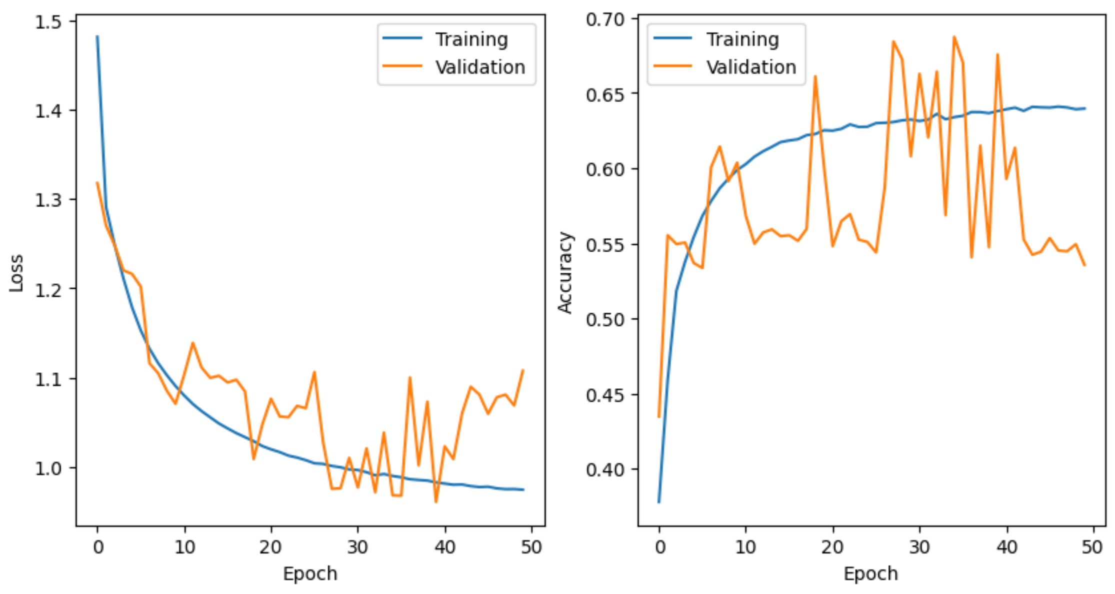
Both are quite shit. 1e-5 won't get anywhere, and with 0.1 the weights are exploding making t the answer to everything

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

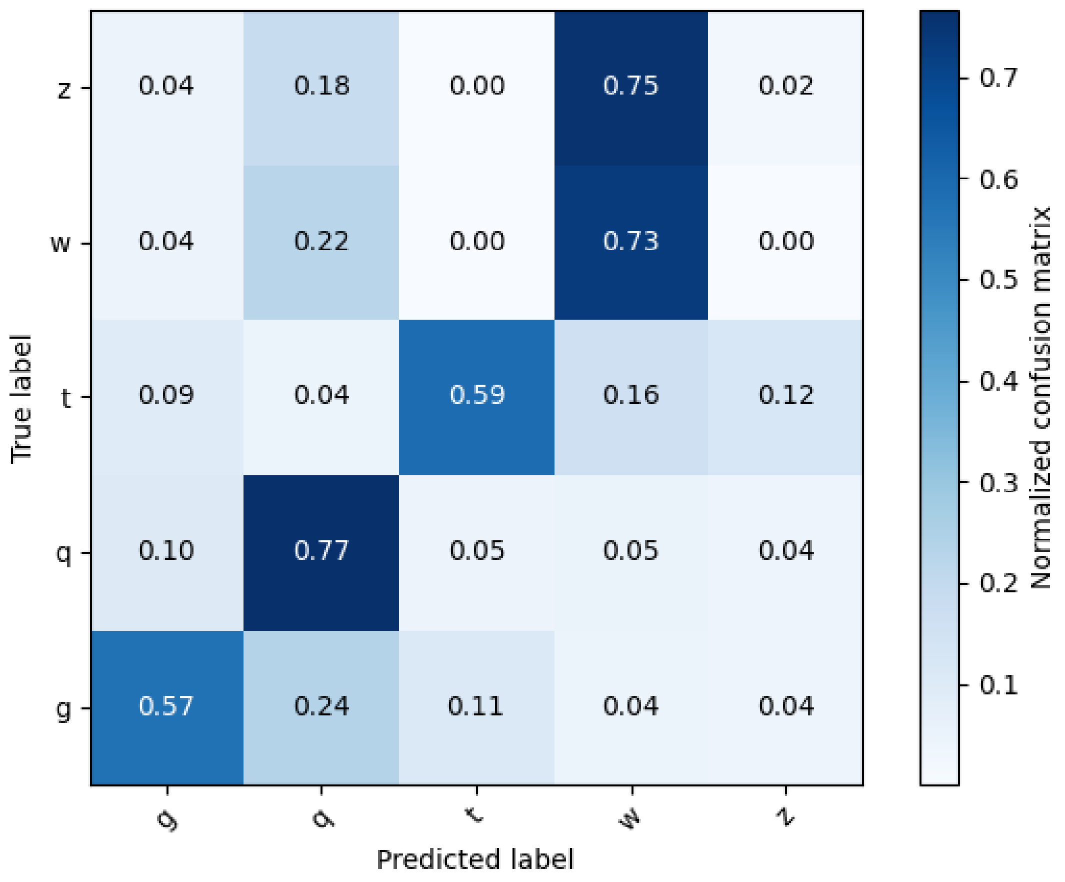
[ ]:
```
run_model(f"sgd", X_train_val, y_train_val, X_test, y_test)
```

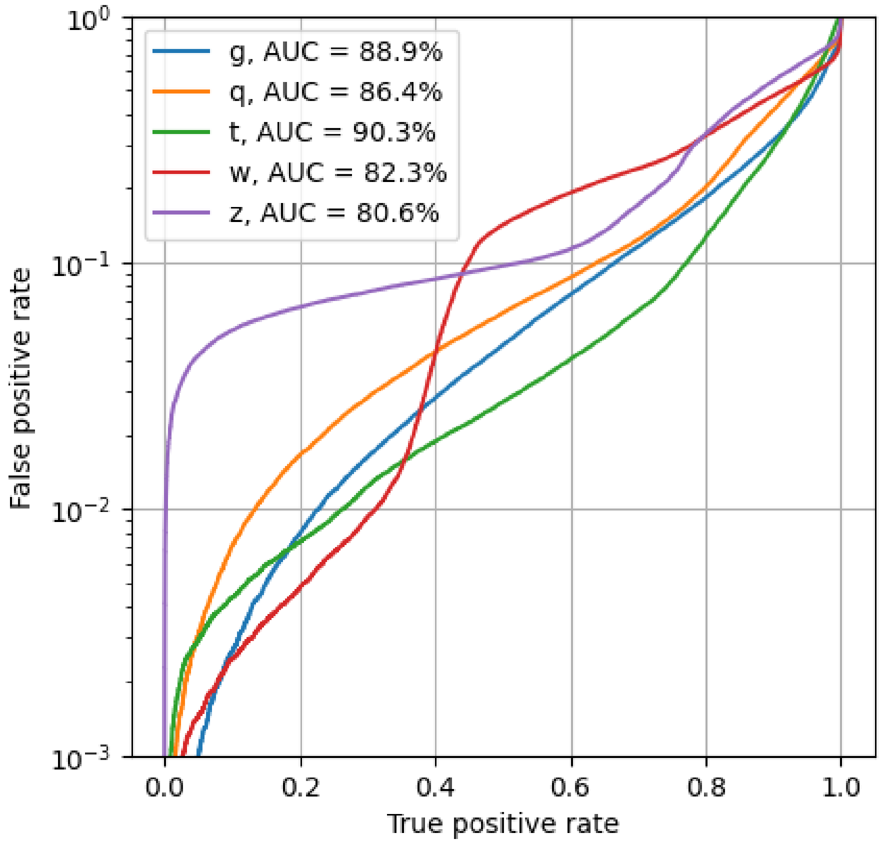
```
Accuracy: 0.5349939759036144
fc1 min:  -0.9018445 max:  1.1101032 mean:  -0.012129007 std:  0.18807948
fc2 min:  -0.35119617 max:  0.53683394 mean:  -0.006469223 std:  0.14495628
fc3 min:  -0.54729956 max:  0.47858587 mean:  -0.004440236 std:  0.18243611
fc4 min:  -1.7156276 max:  1.251062 mean:  -0.0043031434 std:  0.37132877
```

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


Loss
Epoch
Training
Validation
Accuracy
Epoch
Training
Validation

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

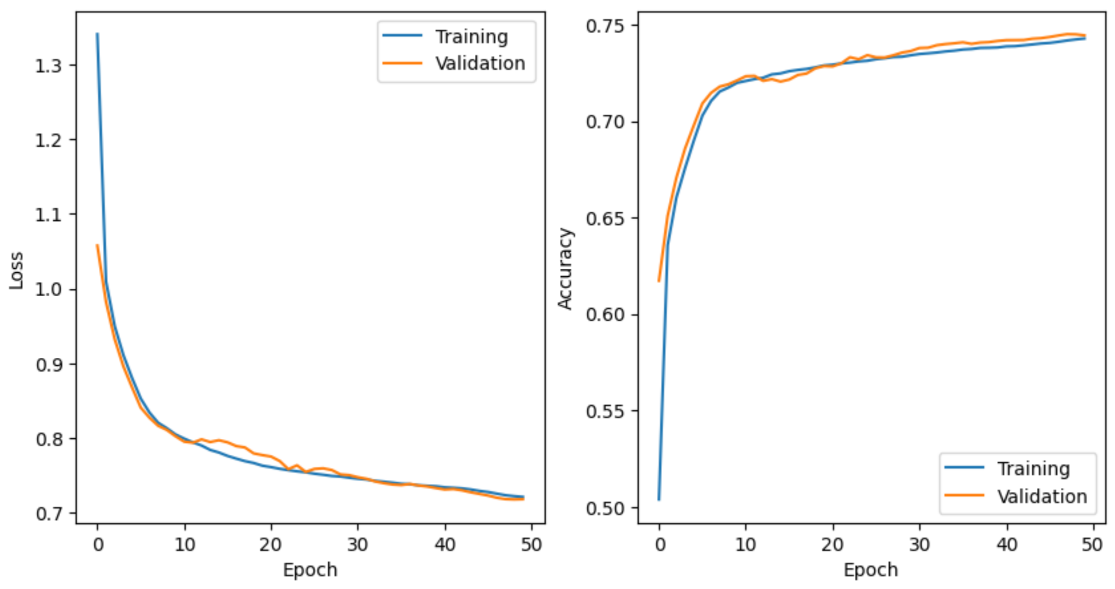

True label
Predicted label
Normalized confusion matrix
z
w
t
q
g
g
q
t
w
z
0.04 0.18 0.00 0.75 0.02
0.04 0.22 0.00 0.73 0.00
0.09 0.04 0.59 0.16 0.12
0.10 0.77 0.05 0.05 0.04
0.57 0.24 0.11 0.04 0.04



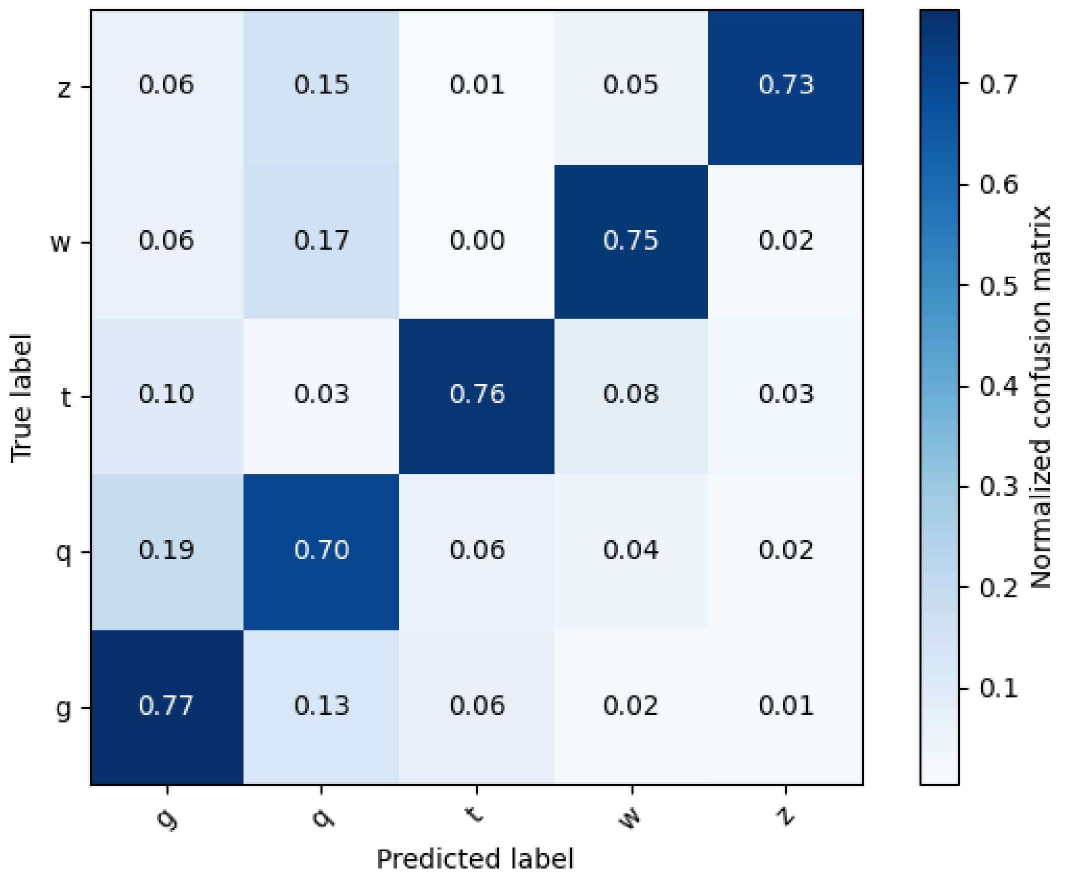
```
[ ]: run_model(f"adam", X_train_val, y_train_val, X_test, y_test, optimizer="adam")
```

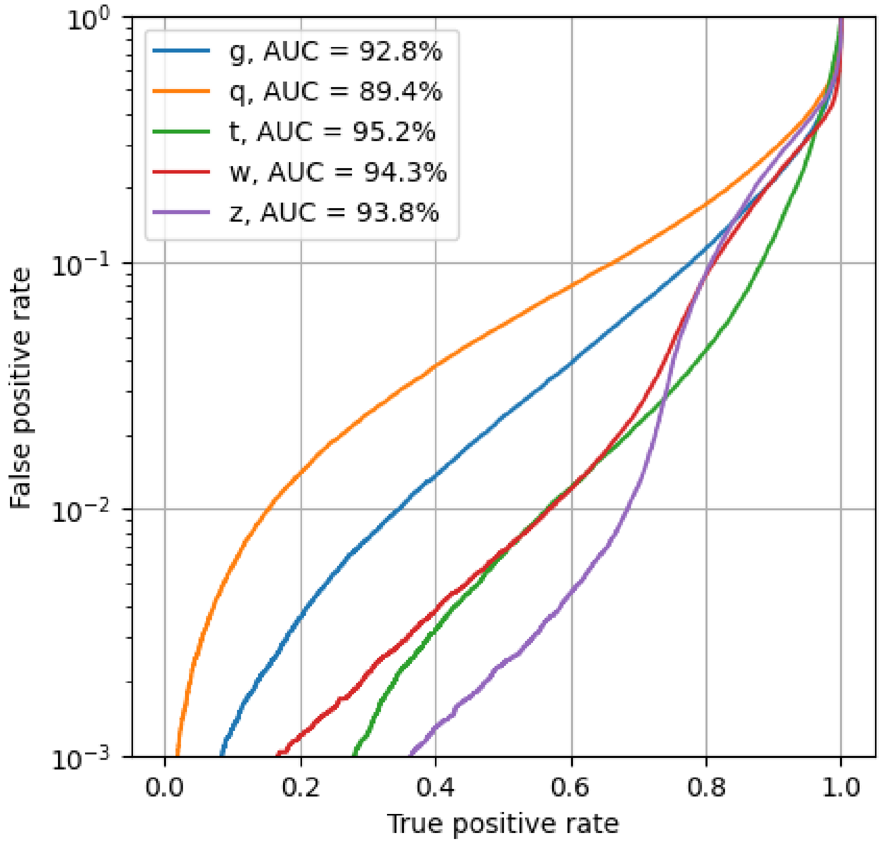
```
Accuracy: 0.7433493975903614
fc1 min:  -6.576317 max:  6.3866954 mean:  -0.049999982 std:  1.335469
fc2 min:  -1.647329 max:  1.1457243 mean:  -0.004924349 std:  0.19020441
fc3 min:  -2.8531551 max:  1.6692454 mean:  -0.006563271 std:  0.28431255
fc4 min:  -1.6900365 max:  1.8332944 mean:  0.0103285415 std:  0.49108788
```

```
[ ]: <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 0.05 0.73
0.06 0.17 0.00 0.75 0.02
0.10 0.03 0.76 0.08 0.03
0.19 0.70 0.06 0.04 0.02
0.77 0.13 0.06 0.02 0.01



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.