Fitting an Artificial Neural Network using Keras

In order to predict the incidence of diabetes given our 21 lifestyle factors, let's try fitting an artificial neural network to our data. We will start with a simple neural network with one hidden layer, measuring loss, accuracy (correct predictions/all predictions), AUC (true positive rate vs. false positive rate), precision (true positives/all predicted positives), and recall (true positives/all actual positives). We will gradually add complexity to our model as we go, graphing our metrics over the course of the 150-epoch training loop as a visual aid, and hopefully arrive at a model that can accurately predict incidence of diabetes.

Importing Data

First let's take a look at our data. We're using data from the CDC's 2015 Behavioral Risk Factor Surveillance System, a large-scale survey that collects data on Americans' lifestyle factors and incidence of disease in order to make predictions. The dataset was already cleaned, being sourced from Kaggle - a link to it can be found in our report. We chose to use the 50-50 split dataset as the original dataset had a heavy bias toward non-diabetics, potentially skewing our data - rather than risk inaccurate modeling, we chose to use a dataset that had undersampled non-diabetics to produce a perfect 50-50 ratio between diabetic and non-diabetic for binary classification.

In [1]:
```
import pandas as pd
diabetes = pd.read_csv("diabetes_binary_5050split_health_indicators_BRFSS2015.csv")
pd.set_option('display.max_columns', None)
```

In [2]:
```
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from sklearn.preprocessing import StandardScaler
from keras.callbacks import ReduceLROnPlateau, EarlyStopping
import warnings
warnings.filterwarnings('ignore')
```

In [3]:
```
diabetes
```

Out[3]:

	Diabetes_binary	HighBP	HighChol	CholCheck	BMI	Smoker	Stroke	HeartDiseaseorAttack	PhysActivity	Fruits	Veggies
0	0.0	1.0	0.0	1.0	26.0	0.0	0.0	0.0	1.0	0.0	1.0
1	0.0	1.0	1.0	1.0	26.0	1.0	1.0	0.0	0.0	1.0	0.0
2	0.0	0.0	0.0	1.0	26.0	0.0	0.0	0.0	1.0	1.0	1.0
3	0.0	1.0	1.0	1.0	28.0	1.0	0.0	0.0	1.0	1.0	1.0
4	0.0	0.0	0.0	1.0	29.0	1.0	0.0	0.0	1.0	1.0	1.0
...	...	...	...	...	...	...	...	...	...	...	...
70687	1.0	0.0	1.0	1.0	37.0	0.0	0.0	0.0	0.0	0.0	1.0
70688	1.0	0.0	1.0	1.0	29.0	1.0	0.0	1.0	0.0	1.0	1.0
70689	1.0	1.0	1.0	1.0	25.0	0.0	0.0	1.0	0.0	1.0	0.0
70690	1.0	1.0	1.0	1.0	18.0	0.0	0.0	0.0	0.0	0.0	0.0
70691	1.0	1.0	1.0	1.0	25.0	0.0	0.0	1.0	1.0	1.0	0.0

70692 rows × 22 columns

Our dataset has 70692 rows - more than enough data to ensure that our predictions won't be skewed by a small sample size. There are 21 predictors, used to predict diagnosis of diabetes:

- high blood pressure
- high cholesterol
- how long it has been since cholesterol was checked
- body mass index
- smoking status
- history of stroke
- history of heart disease or attack
- physical activity level
- consumption of fruits
- consumption of vegetables
- heavy alcohol consumption
- healthcare status
- lack of access to a doctor due to cost

- general health rating
- mental health rating
- physical health rating
- difficulty walking or climbing stairs
- sex
- age
- education level
- income bracket

First Model

Let's try fitting a basic Sequential neural network with one hidden Dense layer on this data. We'll graph accuracy, ROC-AUC, precision, and recall for the training and validation sets to get a good idea of the model's fit. Due to the fact that this is a binary classification problem, we're using the `sigmoid` activation function on the output and the `binary_crossentropy` function to measure loss.

In [4]:
```
X = diabetes.loc[:, diabetes.columns!='Diabetes_binary']
y = diabetes['Diabetes_binary']
model = Sequential()
model.add(Dense(21, input_shape=(21,), activation='relu'))
model.add(Dense(1, activation='sigmoid'))
adam = tf.keras.optimizers.legacy.Adam(learning_rate=0.01)
model.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy', 'AUC', 'Precision', 'Recall'])
```

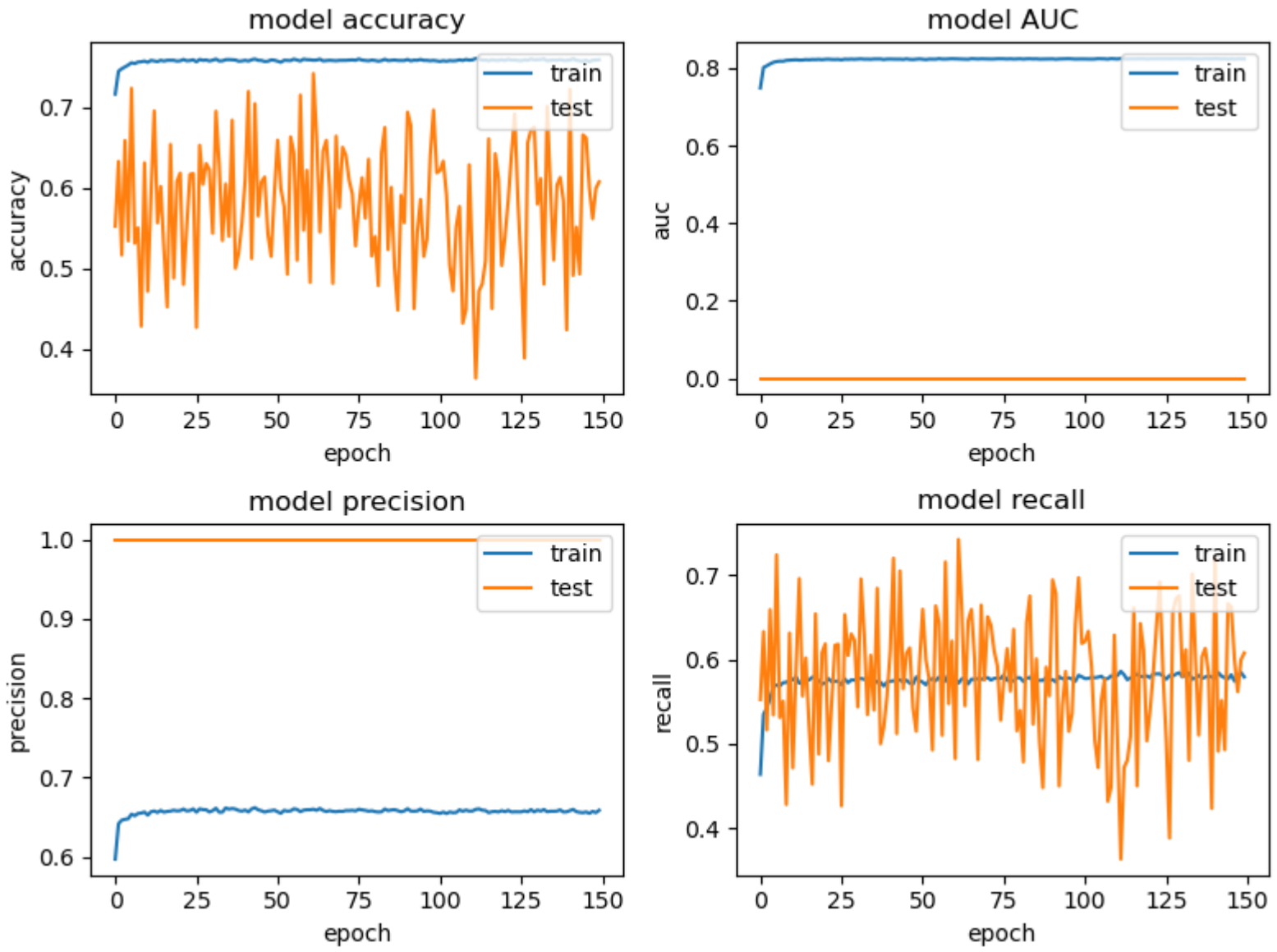
In [ ]:
```
history = model.fit(X, y, validation_split=0.25, epochs=150, batch_size=100, verbose=2)
```

In [6]:
```
loss, accuracy, auc, precision, recall = model.evaluate(X, y)
print('Loss: %.2f' % (loss*100))
print('Accuracy: %.2f' % (accuracy*100))
print('AUC: %.2f' % (auc*100))
print('Precision: %.2f' % (precision*100))
print('Recall: %.2f' % (recall*100))
```

```
2210/2210 [==============================] - 1s 397us/step - loss: 0.5419 - accuracy: 0.7228 - auc: 0.8286 - precisi
on: 0.7915 - recall: 0.6050
Loss: 54.19
Accuracy: 72.28
AUC: 82.86
Precision: 79.15
Recall: 60.50
```

These values for our metrics are neither terrible nor fantastic, but what really matters is how each of these values changed over the course of the 150 epochs while fitting the model. Let's graph those below to see what's going on.

In [7]:
```
fig, axs = plt.subplots(2, 2, figsize=(8, 6))

axs[0, 0].plot(history.history['accuracy'])
axs[0, 0].plot(history.history['val_accuracy'])
axs[0, 0].set_title('model accuracy')
axs[0, 0].set_ylabel('accuracy')
axs[0, 0].set_xlabel('epoch')
axs[0, 0].legend(['train', 'test'], loc='upper right')

axs[0, 1].plot(history.history['auc'])
axs[0, 1].plot(history.history['val_auc'])
axs[0, 1].set_title('model AUC')
axs[0, 1].set_ylabel('auc')
axs[0, 1].set_xlabel('epoch')
axs[0, 1].legend(['train', 'test'], loc='upper right')

axs[1, 0].plot(history.history['precision'])
axs[1, 0].plot(history.history['val_precision'])
axs[1, 0].set_title('model precision')
axs[1, 0].set_ylabel('precision')
axs[1, 0].set_xlabel('epoch')
axs[1, 0].legend(['train', 'test'], loc='upper right')

axs[1, 1].plot(history.history['recall'])
axs[1, 1].plot(history.history['val_recall'])
axs[1, 1].set_title('model recall')
axs[1, 1].set_ylabel('recall')
axs[1, 1].set_xlabel('epoch')
axs[1, 1].legend(['train', 'test'], loc='upper right')

plt.tight_layout()
plt.show()
```



There seem to be some errors with the fit of the model, given these strange metric graphs. Testing AUC is certainly not supposed to be 0, considering that that means the predictions are 100% wrong. Precision also shouldn't be 100%, since such a perfect value is impossible with real-world noisy data such as what we have. Our accuracy for testing also varies wildly from below 40% to almost 80%, meaning we have to adjust the fit of our model.

Second Model

Let's try fitting a model with more layers and a different number of nodes to see if this improves our graphs.

In [8]:
```
model2 = Sequential()
model2.add(Dense(64, input_shape=(21,), activation='relu'))
model2.add(Dense(32, activation='relu'))
model2.add(Dense(1, activation='sigmoid'))
model2.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy', 'AUC', 'Precision', 'Recall'])
```

In [ ]:
```
history_2 = model2.fit(X, y, validation_split=0.25, epochs=150, batch_size=100, verbose=2)
```

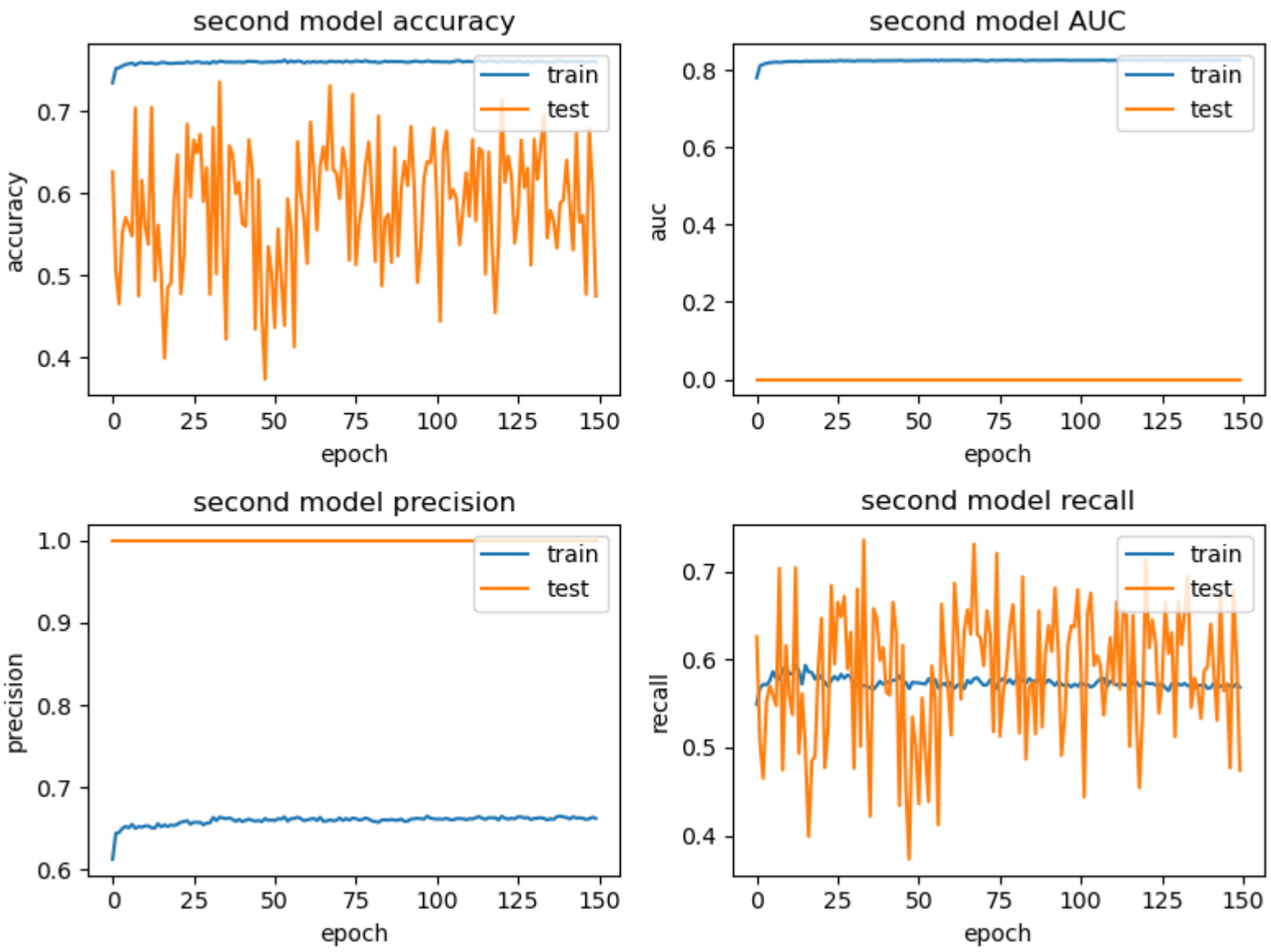
In [10]:
```
loss2, accuracy2, auc2, precision2, recall2 = model2.evaluate(X, y)
print('Loss: %.2f' % (loss2*100))
print('Accuracy: %.2f' % (accuracy2*100))
print('AUC: %.2f' % (auc2*100))
print('Precision: %.2f' % (precision2*100))
print('Recall: %.2f' % (recall2*100))
```

```
2210/2210 [==============================] - 1s 386us/step - loss: 0.5743 - accuracy: 0.6865 - auc: 0.8297 - precisi
on: 0.8244 - recall: 0.4738
Loss: 57.43
Accuracy: 68.65
AUC: 82.97
Precision: 82.44
Recall: 47.38
```

Right away, we can see here that loss, recall, and accuracy are higher, and AUC and precision basically haven't changed between these two models. These look like pretty mixed results, suggesting that adding more layers hasn't done much. Let's look at the graphs.

In [11]:
```
fig, axs = plt.subplots(2, 2, figsize=(8, 6))

axs[0, 0].plot(history_2.history['accuracy'])
axs[0, 0].plot(history_2.history['val_accuracy'])
axs[0, 0].set_title('second model accuracy')
axs[0, 0].set_ylabel('accuracy')
axs[0, 0].set_xlabel('epoch')
axs[0, 0].legend(['train', 'test'], loc='upper right')

axs[0, 1].plot(history_2.history['auc'])
axs[0, 1].plot(history_2.history['val_auc'])
axs[0, 1].set_title('second model AUC')
axs[0, 1].set_ylabel('auc')
```

```
axs[0, 1].set_xlabel('epoch')
axs[0, 1].legend(['train', 'test'], loc='upper right')

axs[1, 0].plot(history_2.history['precision'])
axs[1, 0].plot(history_2.history['val_precision'])
axs[1, 0].set_title('second model precision')
axs[1, 0].set_ylabel('precision')
axs[1, 0].set_xlabel('epoch')
axs[1, 0].legend(['train', 'test'], loc='upper right')

axs[1, 1].plot(history_2.history['recall'])
axs[1, 1].plot(history_2.history['val_recall'])
axs[1, 1].set_title('second model recall')
axs[1, 1].set_ylabel('recall')
axs[1, 1].set_xlabel('epoch')
axs[1, 1].legend(['train', 'test'], loc='upper right')

plt.tight_layout()
plt.show()
```



As suspected, adding more layers and nodes doesn't seem to have changed anything. We still see the same problems in our graphs as we did the first time.

Third Model

Let's try normalizing the predictors before we compile the model this time. I'll use scikit-learn's `StandardScaler` to normalize our `X` and measure how this impacts our various metrics and graphs.

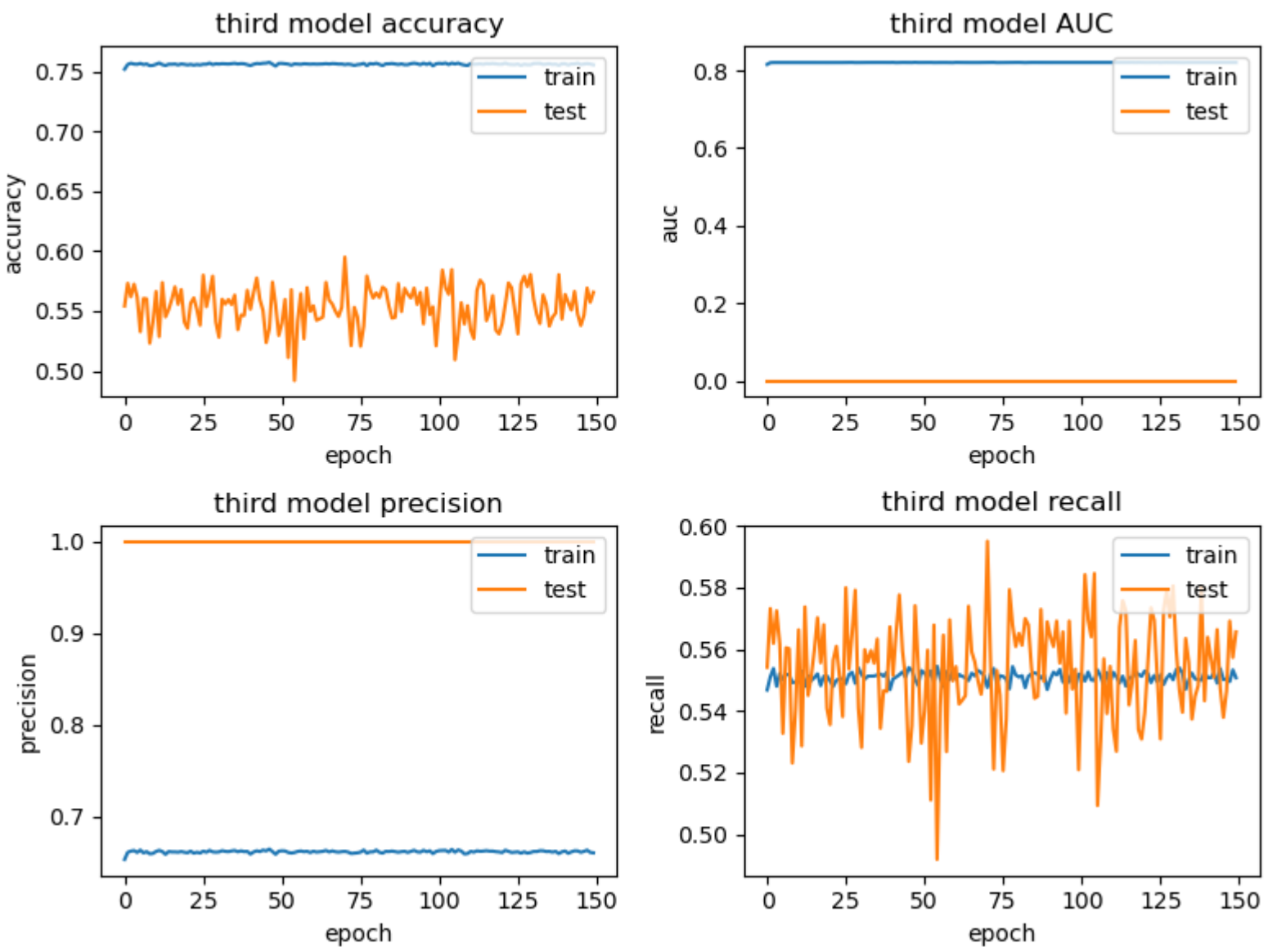
In [ ]:
```
scaler = StandardScaler()

X = scaler.fit_transform(X)

model3 = Sequential()
model2.add(Dense(64, input_shape=(21,), activation='relu'))
model2.add(Dense(32, activation='relu'))
model3.add(Dense(1, activation='sigmoid'))
model3.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy', 'AUC', 'Precision', 'Recall'])
history_3 = model3.fit(X, y, validation_split=0.25, epochs=150, batch_size=100, verbose=2)
```

In [13]:
```
loss3, accuracy3, auc3, precision3, recall3 = model3.evaluate(X, y)
print('Loss: %.2f' % (loss3*100))
print('Accuracy: %.2f' % (accuracy3*100))
print('AUC: %.2f' % (auc3*100))
print('Precision: %.2f' % (precision3*100))
print('Recall: %.2f' % (recall3*100))
```

```
2210/2210 [==============================] - 1s 354us/step - loss: 0.5537 - accuracy: 0.7101 - auc: 0.8230 - precisi
on: 0.7947 - recall: 0.5665
Loss: 55.37
Accuracy: 71.01
AUC: 82.30
Precision: 79.47
Recall: 56.65
```

Accuracy has decreased, while loss has increased slightly and AUC and precision have stayed the same. Recall has decreased significantly. Let's take a look at the graphs.

In [14]:
```
fig, axs = plt.subplots(2, 2, figsize=(8, 6))

axs[0, 0].plot(history_3.history['accuracy'])
axs[0, 0].plot(history_3.history['val_accuracy'])
axs[0, 0].set_title('third model accuracy')
axs[0, 0].set_ylabel('accuracy')
axs[0, 0].set_xlabel('epoch')
axs[0, 0].legend(['train', 'test'], loc='upper right')

axs[0, 1].plot(history_3.history['auc'])
axs[0, 1].plot(history_3.history['val_auc'])
axs[0, 1].set_title('third model AUC')
axs[0, 1].set_ylabel('auc')
axs[0, 1].set_xlabel('epoch')
axs[0, 1].legend(['train', 'test'], loc='upper right')

axs[1, 0].plot(history_3.history['precision'])
axs[1, 0].plot(history_3.history['val_precision'])
axs[1, 0].set_title('third model precision')
axs[1, 0].set_ylabel('precision')
axs[1, 0].set_xlabel('epoch')
axs[1, 0].legend(['train', 'test'], loc='upper right')

axs[1, 1].plot(history_3.history['recall'])
axs[1, 1].plot(history_3.history['val_recall'])
axs[1, 1].set_title('third model recall')
axs[1, 1].set_ylabel('recall')
axs[1, 1].set_xlabel('epoch')
axs[1, 1].legend(['train', 'test'], loc='upper right')

plt.tight_layout()
plt.show()
```



Now the test accuracy is improving a bit more — it no longer fluctuates as wildly, maintaining values between 0.5 and 0.6. However, the other metric graphs still look incorrectly fit, as our AUC is still 0 and our precision is still 1 for the validation set.

Fourth Model

Let's try splitting `X` and `y` into a training and testing set beforehand instead of using the `validation_split` parameter in `model.fit()` to see if this helps. For this we'll use scikit-learn's `train_test_split()` function. We'll still normalize inputs, since that did lead to some improvement last time. We'll also use the same model.

In [39]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, shuffle=True)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

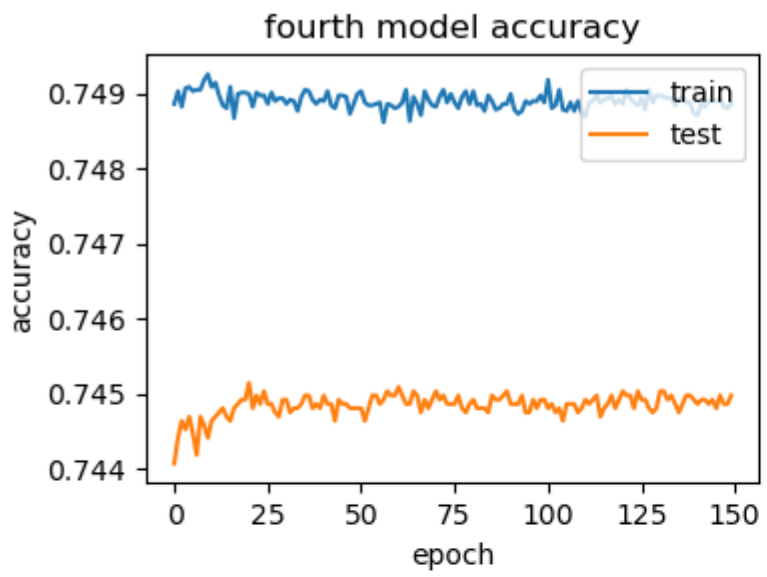
In [ ]:
```
history_4 = model3.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=150, batch_size=100, verbose=2)
```

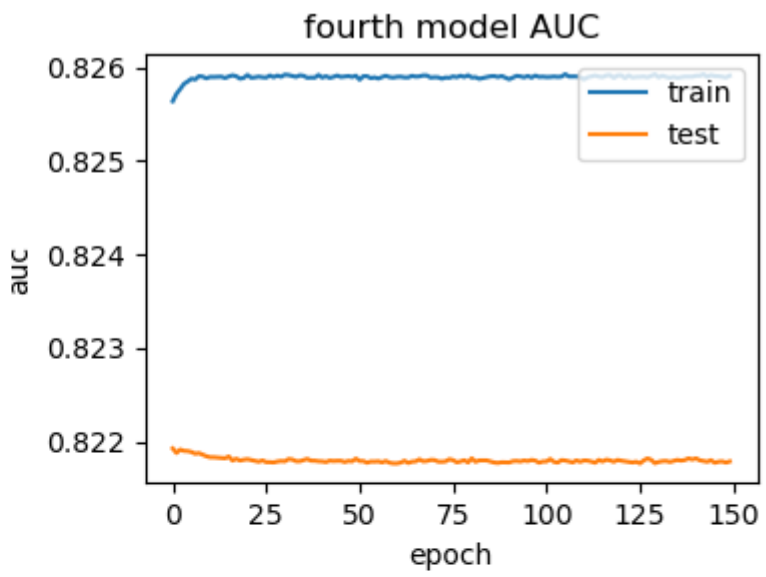
In [41]:
```
loss4, accuracy4, auc4, precision4, recall4 = model3.evaluate(X_test, y_test)

print("Loss: %.2f" % (loss4*100))
print("Accuracy: %.2f" % (accuracy4*100))
print("AUC: %.2f" % (auc4*100))
print("Precision: %.2f" % (precision4*100))
print("Recall: %.2f" % (recall4*100))
```

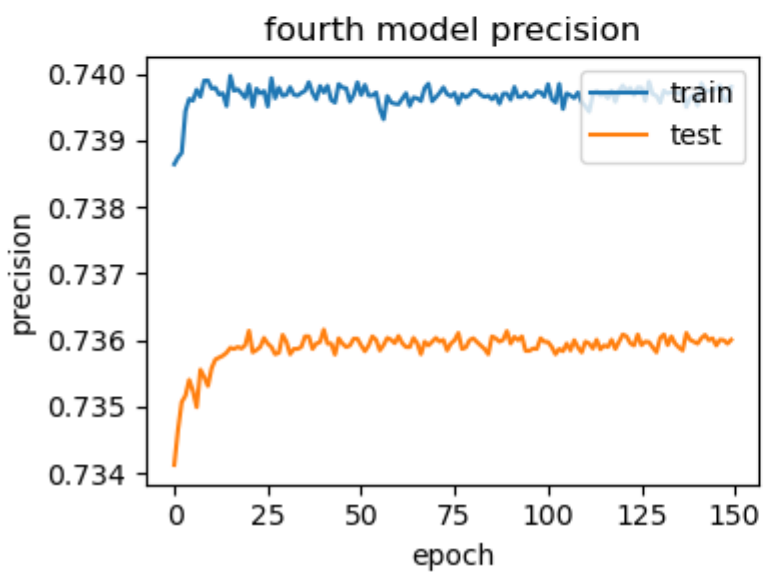
```
553/553 [==============================] - 0s 358us/step - loss: 0.5169 - accuracy: 0.7450 - auc: 0.8218 - precisio
n: 0.7360 - recall: 0.7666
Loss: 51.69
Accuracy: 74.50
AUC: 82.18
Precision: 73.60
Recall: 76.66
```

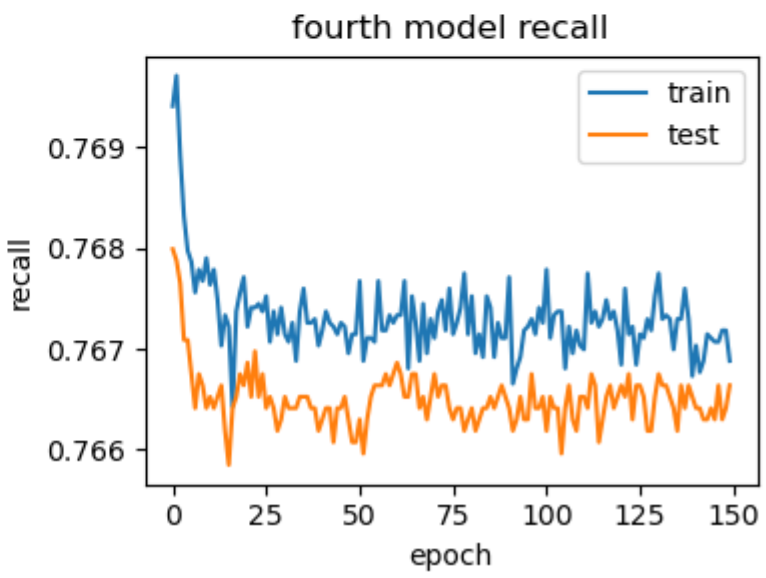
Accuracy and recall have substantially improved, which is a good sign. AUC is still hovering around 82%. Loss and precision have both decreased. Let's take a look at our graphs.

In [42]:
```
fig, axs = plt.subplots(2, 2, figsize=(8, 6))

axs[0, 0].plot(history_4.history['accuracy'])
axs[0, 0].plot(history_4.history['val_accuracy'])
axs[0, 0].set_title('fourth model accuracy')
axs[0, 0].set_ylabel('accuracy')
axs[0, 0].set_xlabel('epoch')
axs[0, 0].legend(['train', 'test'], loc='upper right')

axs[0, 1].plot(history_4.history['auc'])
axs[0, 1].plot(history_4.history['val_auc'])
axs[0, 1].set_title('fourth model AUC')
axs[0, 1].set_ylabel('auc')
axs[0, 1].set_xlabel('epoch')
axs[0, 1].legend(['train', 'test'], loc='upper right')

axs[1, 0].plot(history_4.history['precision'])
axs[1, 0].plot(history_4.history['val_precision'])
axs[1, 0].set_title('fourth model precision')
axs[1, 0].set_ylabel('precision')
axs[1, 0].set_xlabel('epoch')
axs[1, 0].legend(['train', 'test'], loc='upper right')

axs[1, 1].plot(history_4.history['recall'])
axs[1, 1].plot(history_4.history['val_recall'])
axs[1, 1].set_title('fourth model recall')
axs[1, 1].set_ylabel('recall')
axs[1, 1].set_xlabel('epoch')
axs[1, 1].legend(['train', 'test'], loc='upper right')

plt.tight_layout()
plt.show()
```









These look *much* better than our previous examples. While ROC-AUC is neither decreasing nor increasing over time, which is definitely not how the graph should look, it still hovers around 0.82 for the testing set, meaning that the model has good predictive power. The amplitudes of our other metrics have also decreased significantly. However, recall is decreasing slightly over time.

Fifth Model

Let's try to improve these graphs further. We'll reshuffle the training and testing data, add a learning rate scheduler, and implement early stopping to prevent overfitting. For both of these functions, we'll focus on minimizing the loss function for the testing set. I'm giving the training loop a head start of 30 epochs before the early stopping function starts just to ensure that there's enough data to plot a curve.

In [35]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, shuffle=True)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

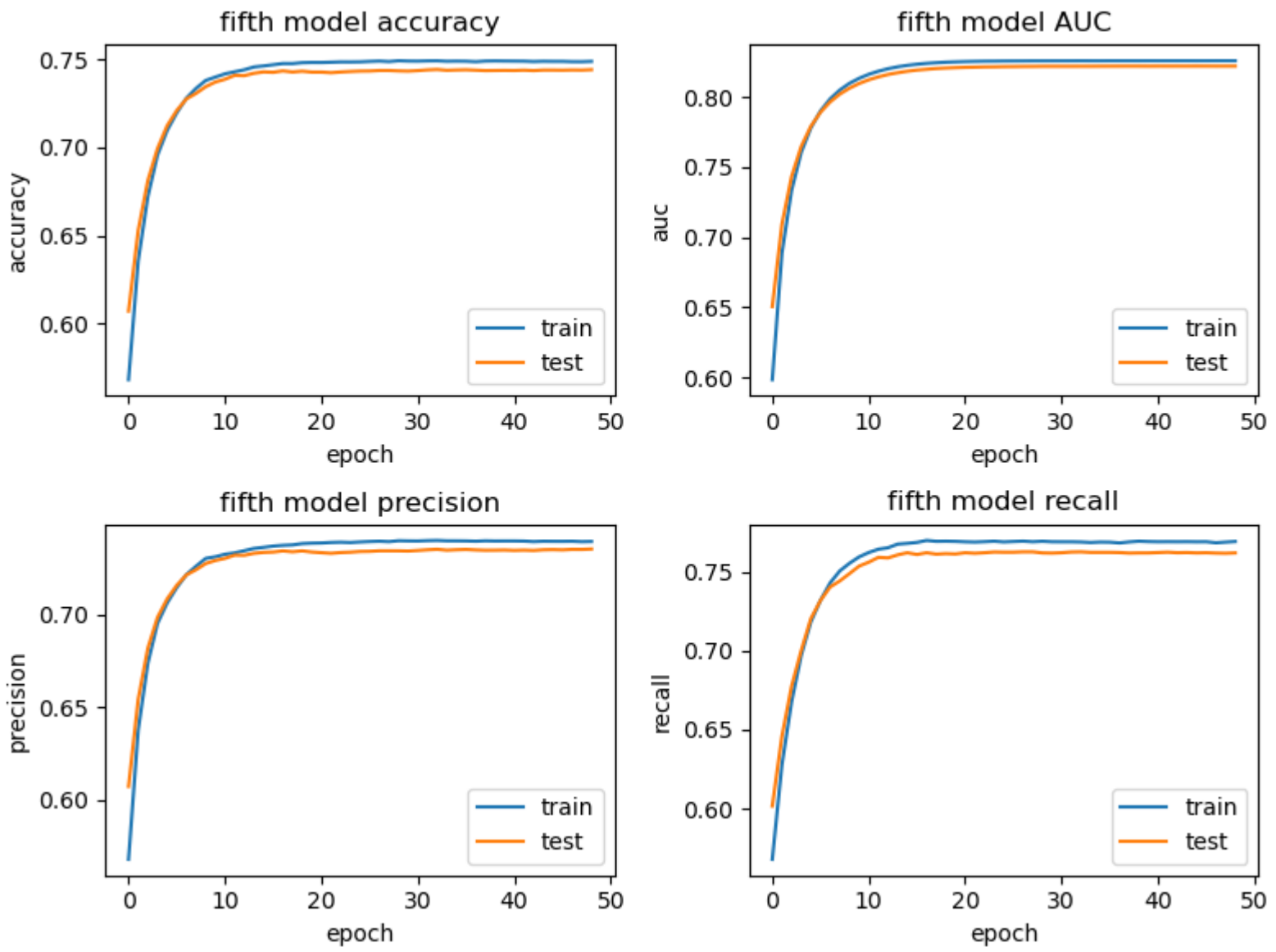
In [ ]:
```
learning_reducer = ReduceLROnPlateau(monitor='val_loss', factor=0.3, patience=10, mode='min', min_lr=0.0001)
early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.0001, patience=10, mode='min', restore_best_weights=

model5 = Sequential()
model2.add(Dense(64, input_shape=(21,), activation='relu'))
model2.add(Dense(32, activation='relu'))
model5.add(Dense(1, activation='sigmoid'))
model5.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy', 'AUC', 'Precision', 'Recall'])
history_5 = model5.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=150, batch_size=100, callbacks=[l
```

In [37]:
```
loss5, accuracy5, auc5, precision5, recall5 = model5.evaluate(X_test, y_test)

print("Loss: %.2f" % (loss5*100))
print("Accuracy: %.2f" % (accuracy5*100))
print("AUC: %.2f" % (auc5*100))
print("Precision: %.2f" % (precision5*100))
print("Recall: %.2f" % (recall5*100))
```

```
553/553 [==============================] - 0s 368us/step - loss: 0.5152 - accuracy: 0.7439 - auc: 0.8219 - precisio
n: 0.7348 - recall: 0.7623
Loss: 51.52
Accuracy: 74.39
AUC: 82.19
Precision: 73.48
Recall: 76.23
```

In [38]:
```
fig, axs = plt.subplots(2, 2, figsize=(8, 6))

axs[0, 0].plot(history_5.history['accuracy'])
axs[0, 0].plot(history_5.history['val_accuracy'])
axs[0, 0].set_title('fifth model accuracy')
axs[0, 0].set_ylabel('accuracy')
```

```
axs[0, 0].set_xlabel('epoch')
axs[0, 0].legend(['train', 'test'], loc='lower right')

axs[0, 1].plot(history_5.history['auc'])
axs[0, 1].plot(history_5.history['val_auc'])
axs[0, 1].set_title('fifth model AUC')
axs[0, 1].set_ylabel('auc')
axs[0, 1].set_xlabel('epoch')
axs[0, 1].legend(['train', 'test'], loc='lower right')

axs[1, 0].plot(history_5.history['precision'])
axs[1, 0].plot(history_5.history['val_precision'])
axs[1, 0].set_title('fifth model precision')
axs[1, 0].set_ylabel('precision')
axs[1, 0].set_xlabel('epoch')
axs[1, 0].legend(['train', 'test'], loc='lower right')

axs[1, 1].plot(history_5.history['recall'])
axs[1, 1].plot(history_5.history['val_recall'])
axs[1, 1].set_title('fifth model recall')
axs[1, 1].set_ylabel('recall')
axs[1, 1].set_xlabel('epoch')
axs[1, 1].legend(['train', 'test'], loc='lower right')

plt.tight_layout()
plt.show()
```



These graphs look great! The curves follow the patterns we would expect them to follow, especially AUC - while testing accuracy, AUC, precision and recall are all very slightly lower than training, this is to be expected as the testing data is novel. Our model now has a final loss of 51.52%, an accuracy of 74.39%, an AUC of 82.19%, precision 73.48%, and recall 76.23%. Our initial values for these metrics were 54.19%, 72.28%, 82.86%, 79.15%, and 60.50%. While AUC and precision have decreased, our recall or true positive rate has increased significantly.

In the context of diagnosing diabetes, it is far less costly to err on the side of caution and accept some false positives than to allow people who actually do have diabetes to slip through the cracks as false negatives. For this reason, it is more prudent to maximize recall than maximize precision. Overall, this model is quite successful at predicting incidence of diabetes, though of course there is always room for improvement. In future iterations, we could reserve a validation set separate from our testing set, add more layers, or tune hyperparameters to improve the fit of this neural network.