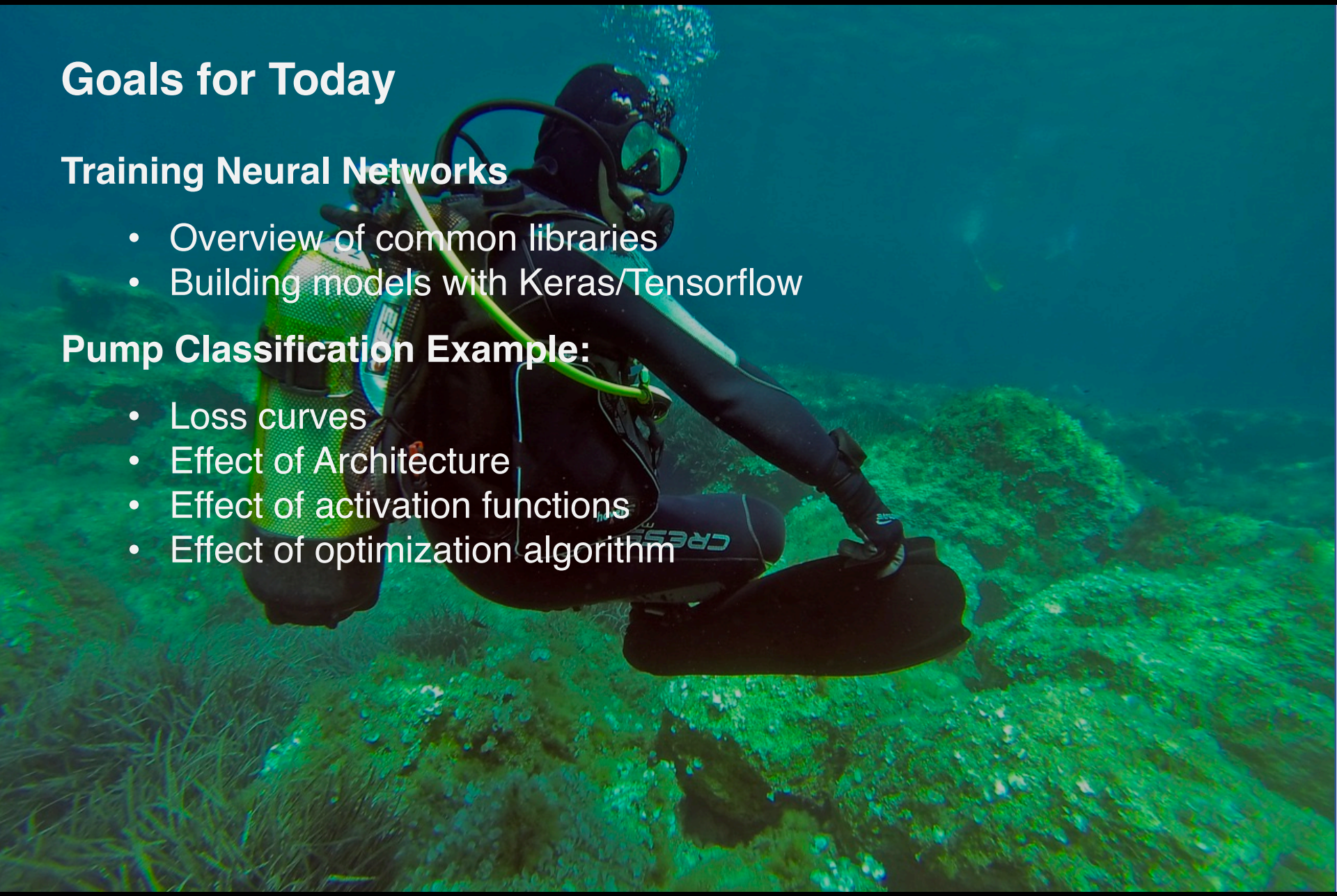
Lecture 20: Deep Learning Implementation



Goals for Today

Training Neural Networks

- Overview of common libraries
- Building models with Keras/Tensorflow

Pump Classification Example:

- Loss curves
- Effect of Architecture
- Effect of activation functions
- Effect of optimization algorithm

Deep Learning in a Python Environment

Low Level Packages:

Theano: Original library dedicated to working with tensors and automatic differentiation (provides GPU support).

Tensorflow: Google's open-source package for working with tensors and automatic differentiation (provides GPU support)

High Level Packages:

Keras: Uses Theano and Tensorflow as a backend for doing all of the computation, but provides simple high-level objects for intuitive model building

Mxnet: Has it's own "back-end" but is built with a philosophy similar to Keras which enables quick experimentation and model building.

Notable Mention:

Pytorch: Written completely in Python (i.e., doesn't call pre-compiled executables in the background) which offers some advantages in debugging. Similar to Mxnet, has it's own "back-end" and built for quick experimentation.

Deep Learning in a Python Environment

Low Level Packages:

Theano: Original library dedicated to working with tensors and automatic differentiation (provides GPU support).

Tensorflow: Google's open-source package for working with tensors and automatic differentiation (provides GPU support)

High Level Packages:

Keras: ~~Uses Theano and Tensorflow as a backend for doing all of the computation, but provides simple high-level objects for intuitive model building~~

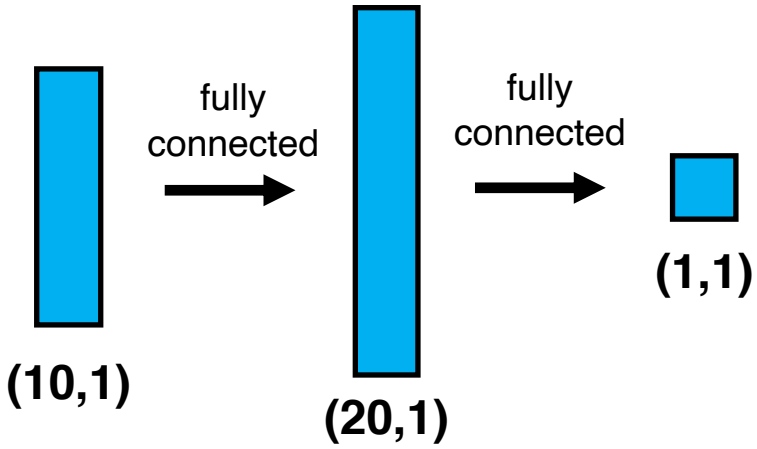
Keras is now merged with Tensorflow as a unified package. Make sure that you use the tf.keras package rather than stand-alone Keras

Notable Mention:

Pytorch: Written completely in Python (i.e., doesn't call pre-compiled executables in the background) which offers some advantages in debugging. Similar to Mxnet, has it's own "back-end" and built for quick experimentation.

Building Models in tf.Keras

One-hidden layer feed-forward network with one output and a vector of inputs



How we build this architecture in keras (via layer objects):

```
from tensorflow.keras.layers import Input,Dense
from tensorflow.keras.models import Model
hidden_layer = Dense(20,activation = 'relu')(input_layer)
input_layer = Input(shape=(10,))
output_layer = Dense(1)(hidden_layer)
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(loss='mean_squared_error',optimizer='sgd',metrics=['accuracy'])
```

1) We instantiate the individual layers and indicate their connectivity via the trailing call (). For example, hidden = Dense(…)(input_layer)

2) Our actual model is built using the Model() call which automatically combines all intermediate layers necessary for evaluating the outputs

Building Models in tf.Keras

A few general notes about the logic here:

```
from tensorflow.keras.layers import Input,Dense
from tensorflow.keras.models import Model
input_layer = Input(shape=(10,))
hidden_layer = Dense(20,activation = 'relu')(input_layer)
output_layer = Dense(1)(hidden_layer)
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(loss='mean_squared_error',optimizer='sgd',metrics=['accuracy'])
```

1) We instantiate the individual **layers** and indicate their connectivity via the trailing **()** argument. For example, **hidden = Dense(…)(input_layer)**

2) Our actual model is built using the **Model()** call which automatically combines all intermediate layers necessary for evaluating the **outputs**

3) Finally, we **compile** our model in preparation for training. This is where we tell Keras what we intend to do with this model with respect to what our **loss** function is, what **optimizer** we want to use, and which **metrics** that we want to keep track of during training.

4) You can build up arbitrarily complex network topologies by including with multiple input and output layers (specified as lists of layers) and combination layers.

Building Models in tf.Keras

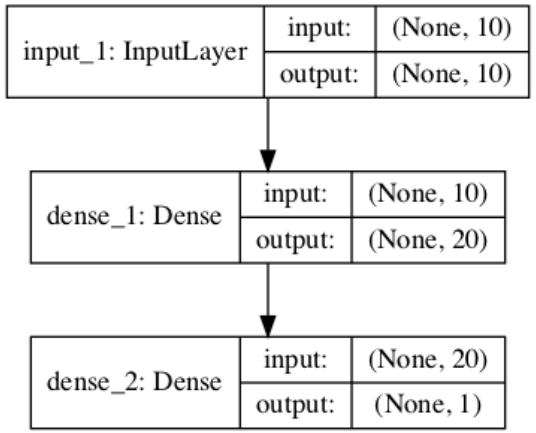
A few general notes about the logic here:

```
from tensorflow.keras.layers import Input,Dense
from tensorflow.keras.models import Model
input_layer = Input(shape=(10,))
hidden_layer = Dense(20,activation = 'relu')(input_layer)
output_layer = Dense(1)(hidden_layer)
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(loss='mean_squared_error',optimizer='sgd',metrics=['accuracy'])
```

5) You should check your work by using Keras' built in plot_model function:

```
import tensorflow as tf
tf.keras.utils.plot_model(model, show_shapes=True)
```

Result:
Note: shapes=True gives
input and
output dimensions



Building Models in tf.Keras

A few general notes about the logic here:

```
from tensorflow.keras.layers import Input,Dense
from tensorflow.keras.models import Model
input_layer = Input(shape=(10,))
hidden_layer = Dense(20,activation = 'relu')(input_layer)
output_layer = Dense(1)(hidden_layer)
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(loss='mean_squared_error',optimizer='sgd',metrics=['accuracy'])
```

6) You should also use the model object's summary() method:

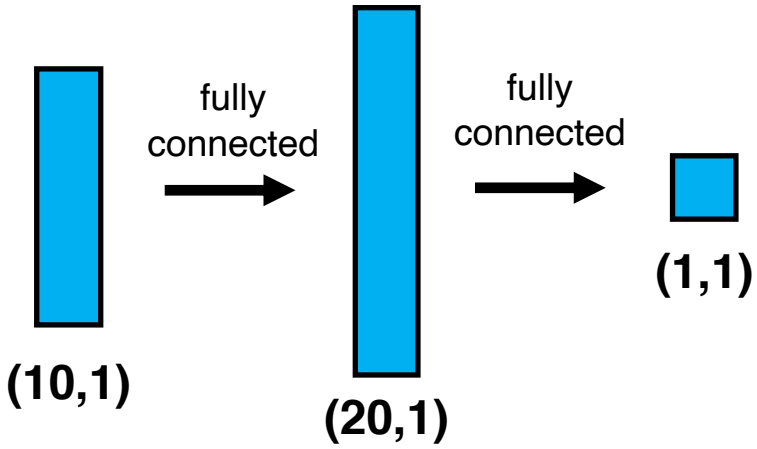
```
model.summary()
```

Result:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 10)                0
_________________________________________________________________
dense_1 (Dense)              (None, 20)                220
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 21
=================================================================
Total params: 241
Trainable params: 241
Non-trainable params: 0
```

Building Models in tf.Keras (Sequential)

One-hidden layer feed-forward network with one output and a vector of inputs



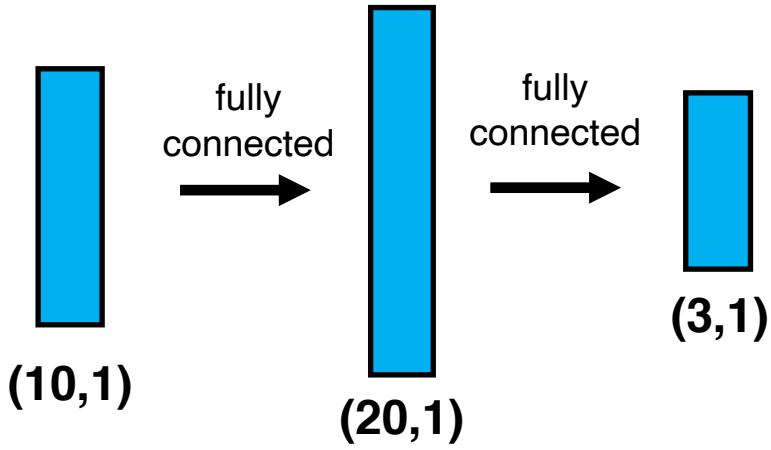
How we build this architecture in keras (via `Sequential`):

```
from tensorflow.keras.models import Sequential
model = Sequential()
model.add(Dense(20,activation ='relu',input_shape=(10,)))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='sgd',metrics=['accuracy'])
```

For simple architectures, the **Sequential** class is an alternative and equivalent way of specifying successive layers (connectivity implied)

Changing model complexity is easy in tf.Keras

One-hidden layer feed-forward network with vector outputs and a vector of inputs



How we build this architecture in keras (via layer objects):

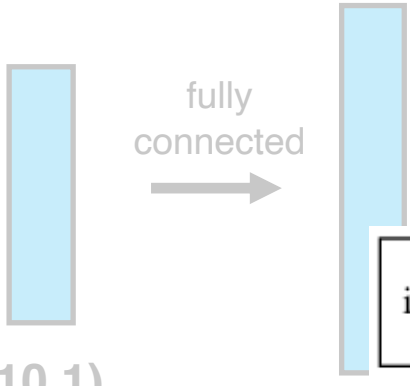
```
tf.keras.backend.clear_session() # resets model
input_layer = Input(shape=(10,))
hidden_layer = Dense(20,activation = 'sigmoid')(input_layer)
output_layer = Dense(3)(hidden_layer)
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(loss='mean_squared_error',optimizer='sgd',metrics=['accuracy'])
```

Note: This only required us to change a single argument, this is part of the ease of Keras.

Note: We also changed the activation function in the hidden layer (Keras supports all of the common activation functions with built-in arguments)

Changing model complexity is easy in tf.Keras

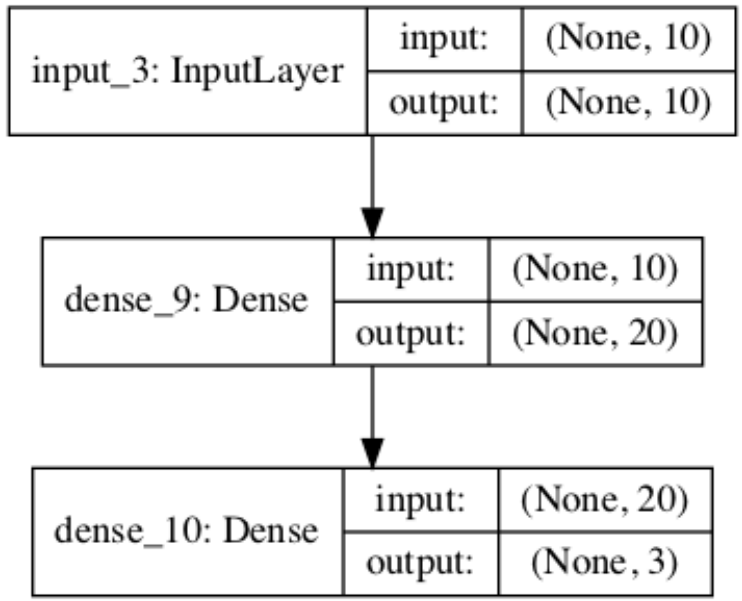
One-hidden layer feed-forward network with vector outputs and a vector of inputs



model.summary()

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         (None, 10)                0
_________________________________________________________________
dense_9 (Dense)              (None, 20)                220
_________________________________________________________________
dense_10 (Dense)             (None, 3)                 63
=================================================================
Total params: 283
Trainable params: 283
Non-trainable params: 0
```

plot_model()

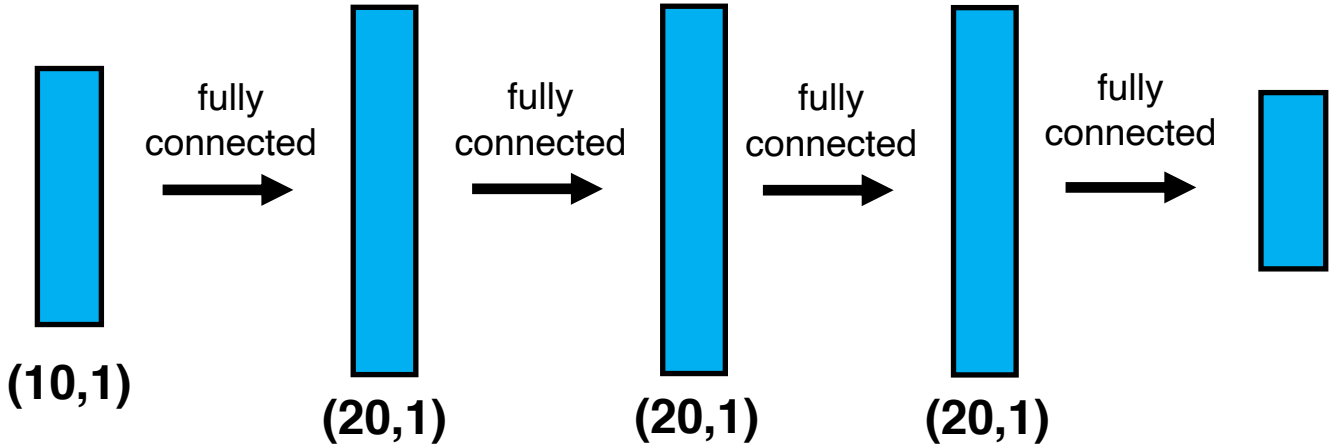


Note: This only required us to change a single argument, this is part of the ease of Keras.

Note: We also changed the activation function in the hidden layer (Keras supports all of the common activation functions with built-in arguments)

Changing model complexity is easy in tf.Keras

Three-hidden layer feed-forward network with vector outputs and a vector of inputs



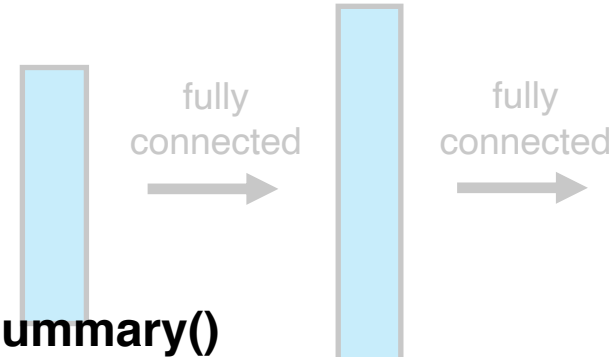
How we build this architecture in keras (via layer objects):

```
tf.keras.backend.clear_session() # resets model
input_layer = Input(shape=(10,))
hidden_1 = Dense(20,activation = 'sigmoid')(input_layer)
hidden_2 = Dense(20,activation = 'relu')(hidden_1)
hidden_3 = Dense(20,activation = 'softmax')(hidden_2)
output_layer = Dense(3)(hidden_3)
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(loss='mean_squared_error',optimizer='sgd',metrics=['accuracy'])
model.summary()
```

Note: we can use different activation functions in each layer.

Changing model complexity is easy in tf.Keras

Three-hidden layer feed-forward network with vector outputs and a



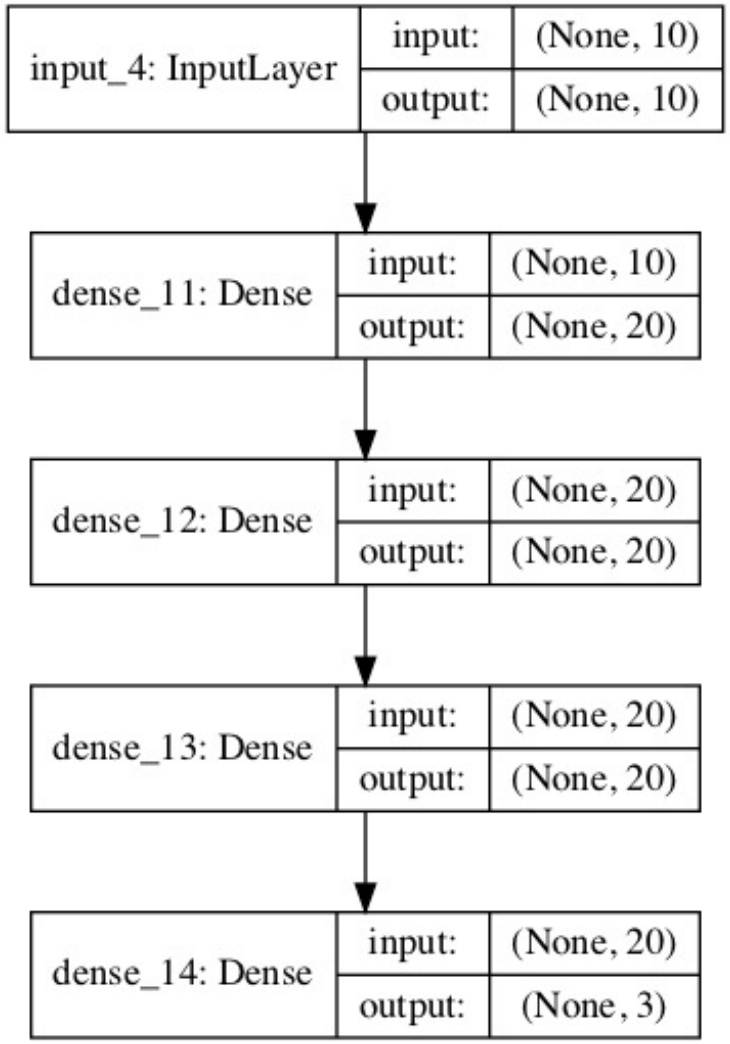
model.summary()

```
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         (None, 10)                0
_________________________________________________________________
dense_11 (Dense)             (None, 20)                220
_________________________________________________________________
dense_12 (Dense)             (None, 20)                420
_________________________________________________________________
dense_13 (Dense)             (None, 20)                420
_________________________________________________________________
dense_14 (Dense)             (None, 3)                 63
=================================================================
Total params: 1,123
Trainable params: 1,123
Non-trainable params: 0
_________________________________________________________________
```

```
model = Model(inputs=input_layer, outputs=outpu
model.compile(loss='mean_squared_error',optimiz
model.summary()
```

Note: we can use different activation functions in e

plot_model()



Pump Classification Example

We are assessing fluid pump failure for a costly project. After (costly) testing each pump is classified as "passed" "rework" (i.e., worth the cost of repair) or "fail" (i.e., irreparable).

We can easily measure four features of the pumps:

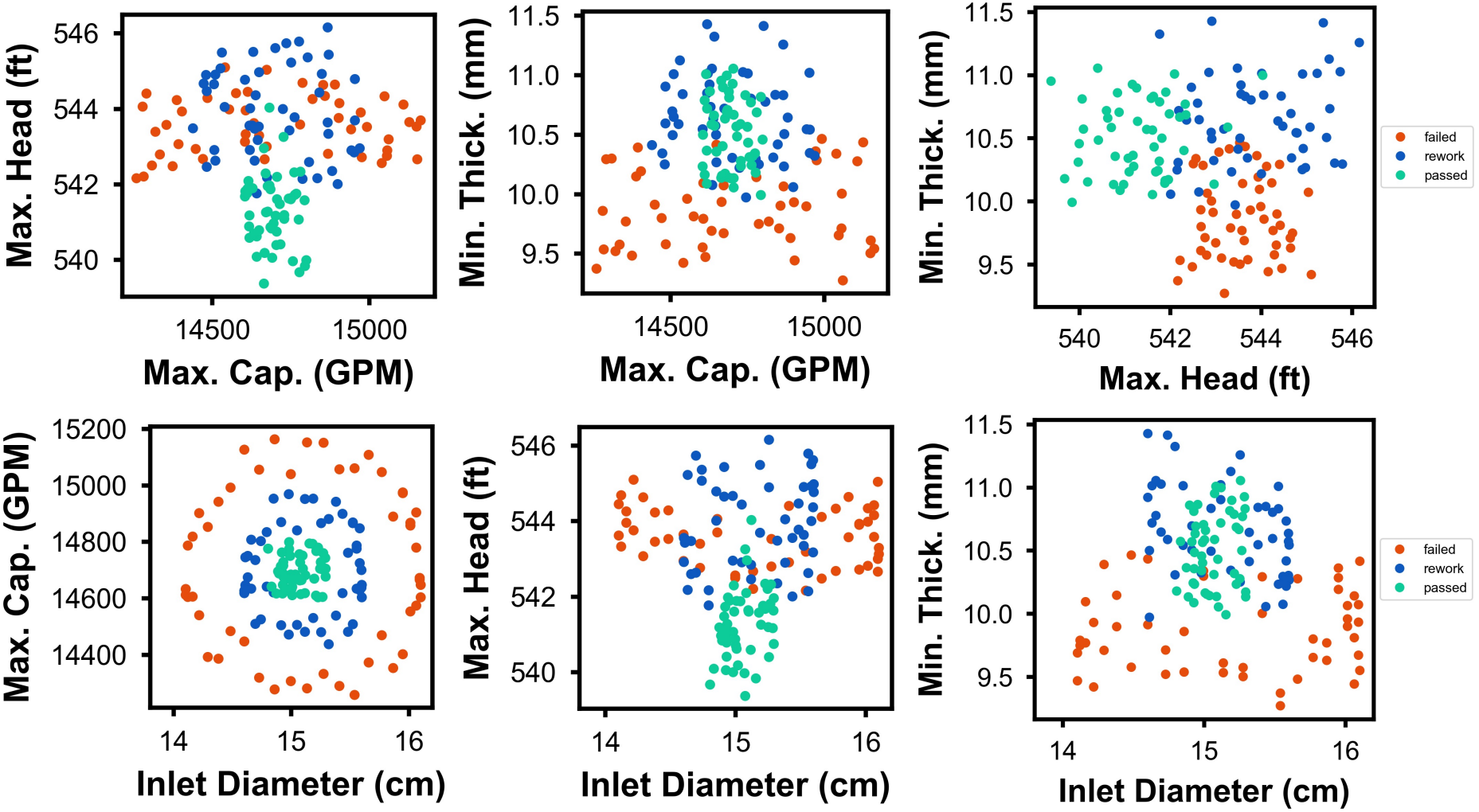
- **Inlet diameter** (can affect dynamic pressure and cavitation)
- **Maximum head** (pressure/elevation we can pump against)
- **Maximum capacity** (gallons per minute flow through the pump)
- **Minimum wall thickness** (surrogate for pressure related failure)

None of these directly reports on pump failure, but we would like to see if we can build a model to classify the pumps using these features in order to avoid more time-consuming stress testing.

Our division has given us the testing results for 150 different pumps as well as these four features in order to train and validate our model.

Pump Classification Example

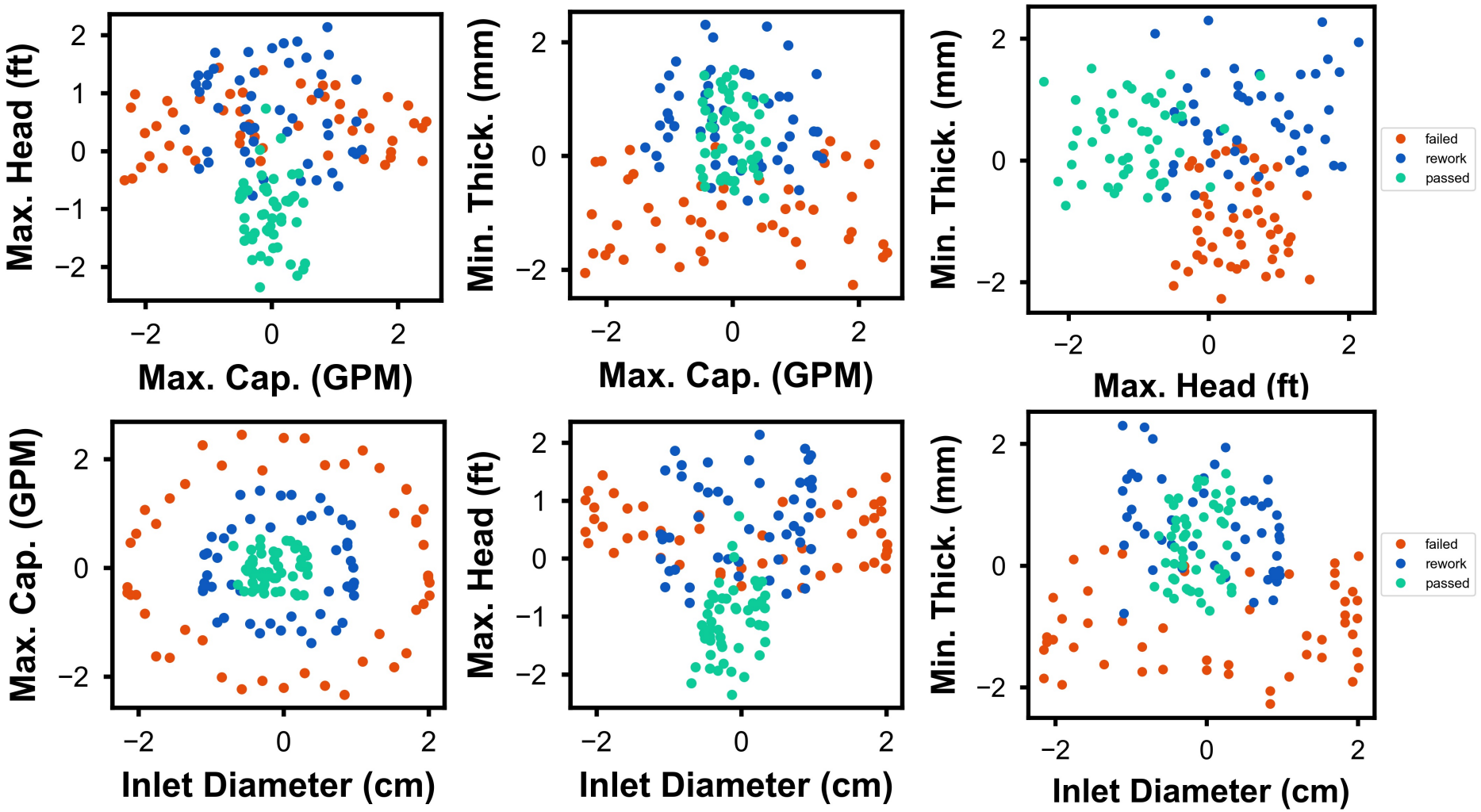
Let's look at the data to see what we are working with:



These are clearly useful features (esp. max cap and inlet diameter) but it isn't clear whether the labels are linearly separable.

Pump Classification Example

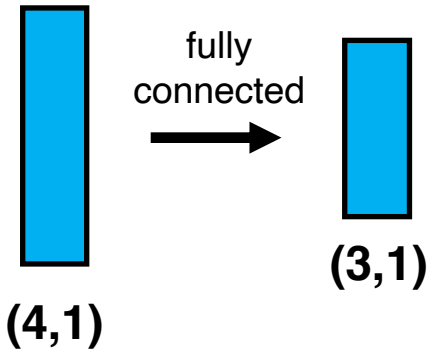
Let's standardize the features before training models:



Recall that standardization entails: $\bar{\mathbf{x}} = \frac{\mathbf{x} - \mu_{\mathrm{x}}}{\sigma_{\mathrm{x}}}$

Classification with 1 Layer

Our initial model won't be deep at all, it will just be a single fully connected layer

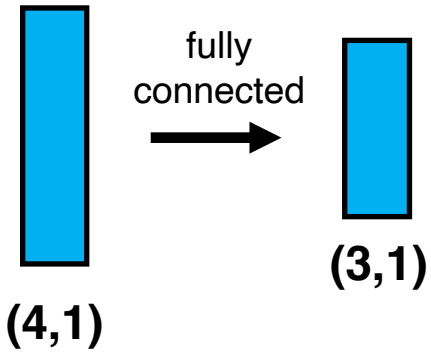


Prepare the data for training:

```
# Load the features
features = pd.read_csv('features.txt', delimiter=r"\s+")
labels = pd.read_csv('labels.txt',header=None)

# print out summary statistics for each feature
print(features.describe())

# Make the data arrays
X = features.to_numpy()
label_dict = { i:count_i for count_i,i in enumerate(set(labels[0])) }
Y = np.array([ label_dict[_] for _ in labels[0] ])

# Generate the training/testing split
x_train, x_test, y_train, y_test = train_test_split(X, Y,
test_size=0.2,random_state=135323)
```

Classification with 1 Layer

Our initial model won't be deep at all, it will just be a single fully connected layer



Build the model:

```
from tensorflow.keras.metrics import Precision, Recall, CategoricalAccuracy

# Initialize input layer (Note the shape, we have four input features)
input_layer = Input(shape=(4,))

# Connect output layer to input layer (Note the shape, we have three classes)
output_layer = Dense(3,activation = 'softmax')(input_layer)

# Create the model
model = Model(inputs=input_layer,outputs=output_layer)
model.compile(loss='categorical_crossentropy',optimizer='sgd',\
              metrics=[Precision(), Recall(), CategoricalAccuracy()])
```

Note: we are predicting a one-hot vector (one column for each potential class). The cross-entropy is the logical loss function. Zero for perfect prediction.

Classification with 1 Layer

Train the model:

```
from tensorflow.keras.utils import to_categorical

# Initialize scalar object for features
scaler = StandardScaler().fit(x_train)

# Train the model using sgd and 100 epochs
history=model.fit(x=scaler.transform(x_train),y =
to_categorical(y_train),epochs=100,verbose=1,batch_size=10,\
validation_data=(scaler.transform(x_test), to_categorical(y_test)))
```

Note: Keras/Tensorflow trains on numpy arrays, so we can use sklearn scaler.

Note: training categorization tasks on numbers can mislead the model, so we train on one-hots (i.e., matrices with a column for each category, a "1" in the category of the sample, and "0" elsewhere) using the `to_categorical()` function.

```
# Illustrate what to_categorically() is doing.
converted = to_categorical(y_train)
for count_i,i in enumerate(y_train):
    print("{} -> {}".format(i,converted[count_i,:]))
```

```
2 -> [0. 0. 1.]
2 -> [0. 0. 1.]
1 -> [0. 1. 0.]
0 -> [1. 0. 0.]
0 -> [1. 0. 0.]
1 -> [0. 1. 0.]
1 -> [0. 1. 0.]
…
```

Classification with 1 Layer

Train the model:

```
from tensorflow.keras.utils import to_categorical

# Initialize scalar object for features
scaler = StandardScaler().fit(x_train)

# Train the model using sgd and 100 epochs
history=model.fit(x=scaler.transform(x_train),y =
to_categorical(y_train),epochs=100,verbose=1,batch_size=10,\
validation_data=(scaler.transform(x_test), to_categorical(y_test)))
```

Note: Keras/Tensorflow trains on numpy arrays, so we can use sklearn scaler.

Note: training categorization tasks on numbers can mislead the model, so we train on one-hots (i.e., matrices with a column for each category, a "1" in the category of the sample, and "0" elsewhere) using the `to_categorical()` function.

Note: the returned object, which we have labeled history, stores the accuracy and loss function at each epoch.

```
print("history keys: {}".format(history.history.keys()))
```

```
history keys: dict_keys(['loss', 'precision', 'recall',
'categorical_accuracy', 'val_loss', 'val_precision', 'val_recall',
'val_categorical_accuracy'])
```

Classification with 1 Layer

Train the model:

```
from tensorflow.keras.utils import to_categorical

# Initialize scalar object for features
scaler = StandardScaler().fit(x_train)

# Train the model using sgd and 100 epochs
history=model.fit(x=scaler.transform(x_train),y =
to_categorical(y_train),epochs=100,verbose=1,batch_size=10,\
validation_data=(scaler.transform(x_test), to_categorical(y_test)))
```

Note: Keras/Tensorflow trains on numpy arrays, so we can use sklearn scaler.

Note: training categorization tasks on numbers can mislead the model, so we train on one-hots (i.e., matrices with a column for each category, a "1" in the category of the sample, and "0" elsewhere) using the `to_categorical()` function.

Note: the returned object, which we have labeled history, stores the accuracy and loss function at each epoch.

```
print("history keys: {}".format(history.history.keys()))
```

```
history keys: dict_keys(['loss', 'precision', 'recall',
'categorical_accuracy', 'val_loss', 'val_precision', 'val_recall',
'val_categorical_accuracy'])
```

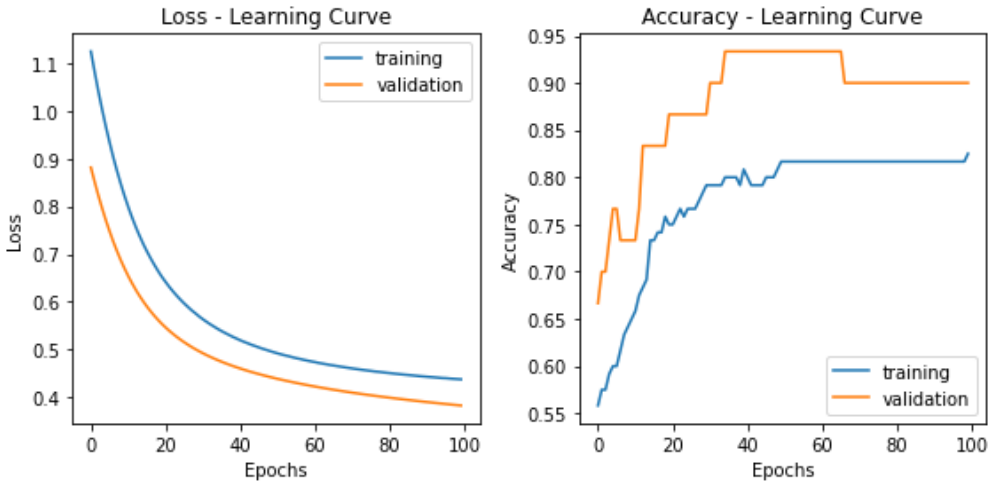
Classification with 1 Layer

Plotting the learning curves:

```
# Plot Loss Curve
plt.figure(figsize=(8, 4))
plt.subplot(121)
plt.plot(np.arange(len(history.history["loss"])),\
         history.history["loss"],label="training")
plt.plot(np.arange(len(history.history["loss"])),\
         history.history["val_loss"],label="validation")
plt.title('Loss - Learning Curve')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()

… similar for accuracy …
```

Note: our loss function is actually our relevant figure of merit. It is a smooth surrogate.

Note: the fact that the validation accuracy is higher than the training accuracy is an artifact of the small dataset



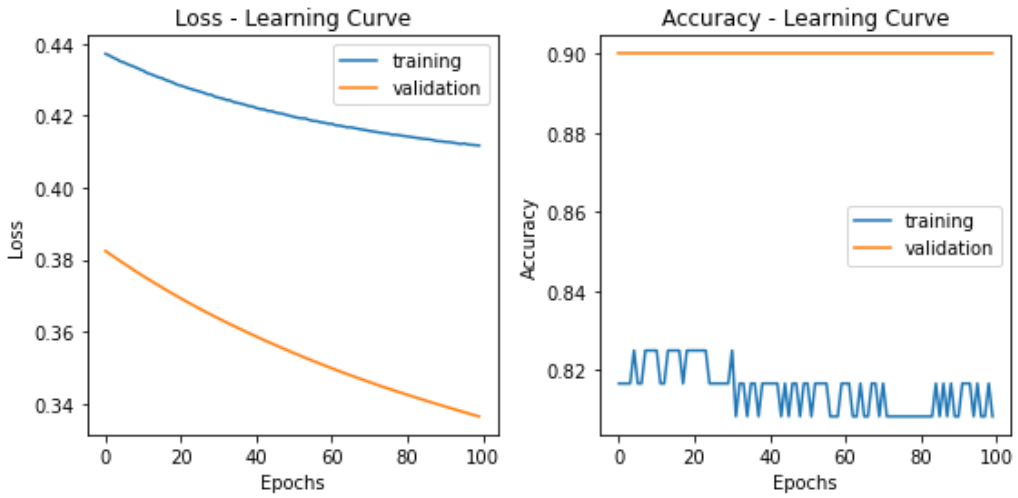
Classification with 1 Layer

Subsequent .fit() calls resume rather than restart training:

```
# Train the model using sgd and 100 epochs
history=model.fit(x=scaler.transform(x_train),\
                  y = to_categorical(y_train),\
                  epochs=100,verbose=0,batch_size=10,\
                  validation_data=(scaler.transform(x_test), \
                  to_categorical(y_test)))

…same plot calls…
```

Note: the loss starts where the previous training stopped, because the model is resuming rather than restarting.

Note: because the problem is so simple, we don't see an overtraining effect, just saturation of the performance.



Classification with 1 Layer

Making predictions with the trained model (`Model.predict()` method):

```
predictions = model.predict(scaler.transform(x_test))
for count_i,i in enumerate(y_test):
    print("{} : {}".format(i,predictions[count_i]))
```

```
0 : [0.6177614  0.34464324 0.03759535]
1 : [0.07616617 0.90408427 0.01974963]
0 : [0.8805375  0.0551772  0.06428528]
1 : [0.16098681 0.48489758 0.35411566]
2 : [0.03220062 0.4577406  0.5100588 ]
0 : [0.68620193 0.19326305 0.12053499]
1 : [0.01861818 0.47442153 0.5069603 ]
1 : [0.4630421 0.4231049 0.113853 ]
…
```

Note: our model predicts the relative probability of each class rather than the class label itself.

To get the class label, we can use the numpy `argmax()` function:

```
predictions = model.predict(scaler.transform(x_test))
predictions = np.argmax(predictions, axis=1)
```

Classification with 1 Layer

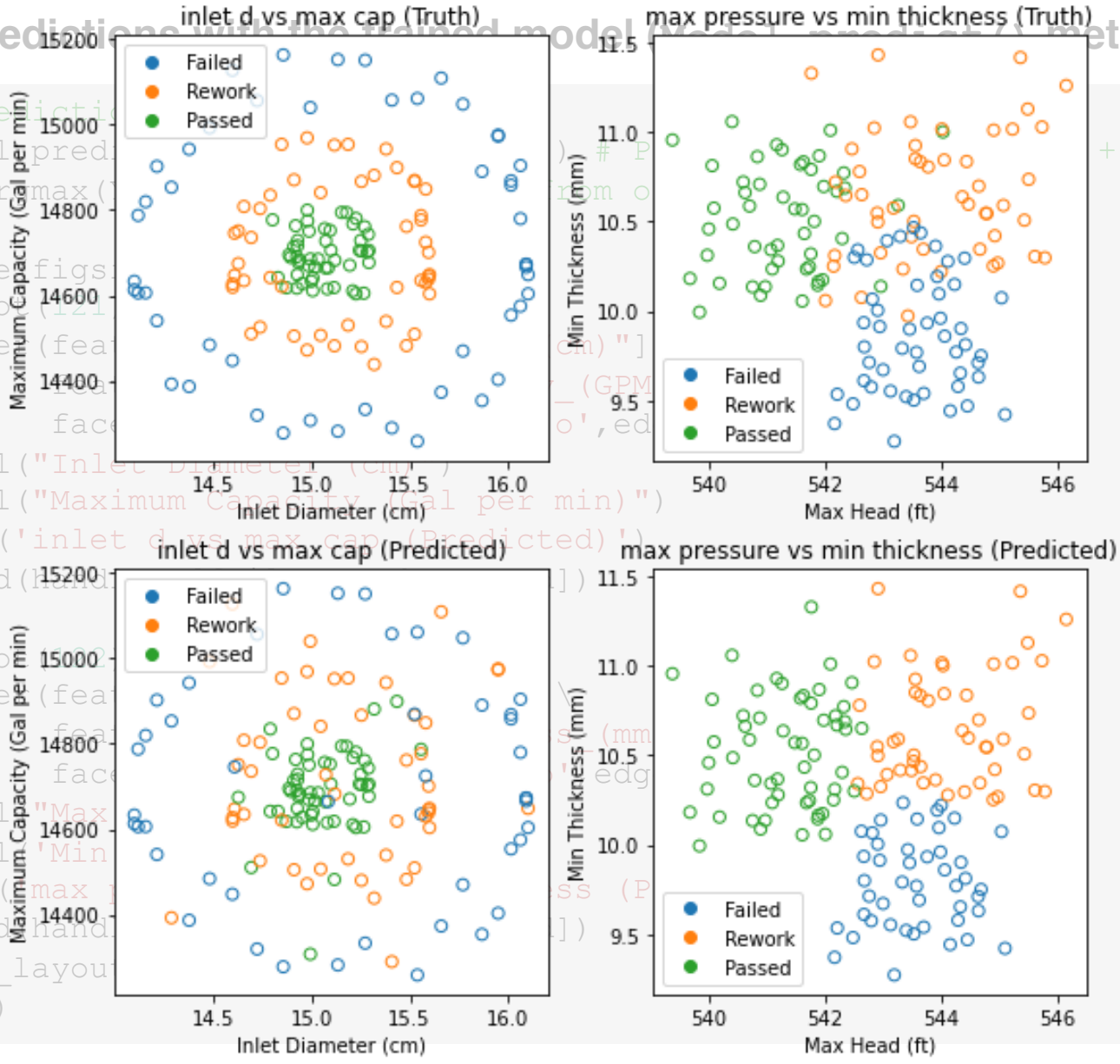
Making predictions with the trained model (`Model.predict()` method):

```
# Make prediction plots
YP = model.predict(scaler.transform(X)) # Predictions on training + testing
YP = np.argmax(YP, axis=1) # Convert from one-hots to labels

plt.figure(figsize=(8, 4))
plt.subplot(121)
plt.scatter(features["Inlet_Diameter_(cm)"],\
            features["Maximum_Capacity_(GPM)"],\
            facecolor='None', marker='o',edgecolors=cm.tab10(YP))
plt.xlabel("Inlet Diameter (cm)")
plt.ylabel("Maximum Capacity (Gal per min)")
plt.title('inlet d vs max cap (Predicted)')
plt.legend(handles=[fail,rework,passed])

plt.subplot(122)
plt.scatter(features["Max_Head_(ft)"],\
            features["Minimum_Thickness_(mm)"],\
            facecolor='None',marker='o',edgecolors=cm.tab10(YP))
plt.xlabel("Max Head (ft)")
plt.ylabel('Min Thickness (mm)')
plt.title('max pressure vs min thickness (Predicted)')
plt.legend(handles=[fail,rework,passed])
plt.tight_layout()
plt.show()
```

Classification with 1 Layer

Making predictions with the trained model (Model.predict() method):

```
# Make predictions on the entire dataset
YP = model.predict(X) # Predict on the entire dataset (training + testing
YP = np.argmax(YP, axis=1) # Convert from one-hot encoding to class labels

plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.scatter(features[:, 0], features[:, 1], c=YP, cmap='viridis', edgecolor='k')
plt.xlabel("Inlet Diameter (cm)")
plt.ylabel("Maximum Capacity (Gal per min)")
plt.title('inlet d vs max cap (Predicted)')
plt.legend(handles=scatter.legend_elements()[0], labels=['Failed', 'Rework', 'Passed'])

plt.subplot(122)
plt.scatter(features[:, 2], features[:, 3], c=YP, cmap='viridis', edgecolor='k')
plt.xlabel("Max Head (ft)")
plt.ylabel("Min Thickness (mm)")
plt.title('max pressure vs min thickness (Predicted)')
plt.legend(handles=scatter.legend_elements()[0], labels=['Failed', 'Rework', 'Passed'])
plt.tight_layout()
plt.show()
```



Classification with Deep Model (varying activation)

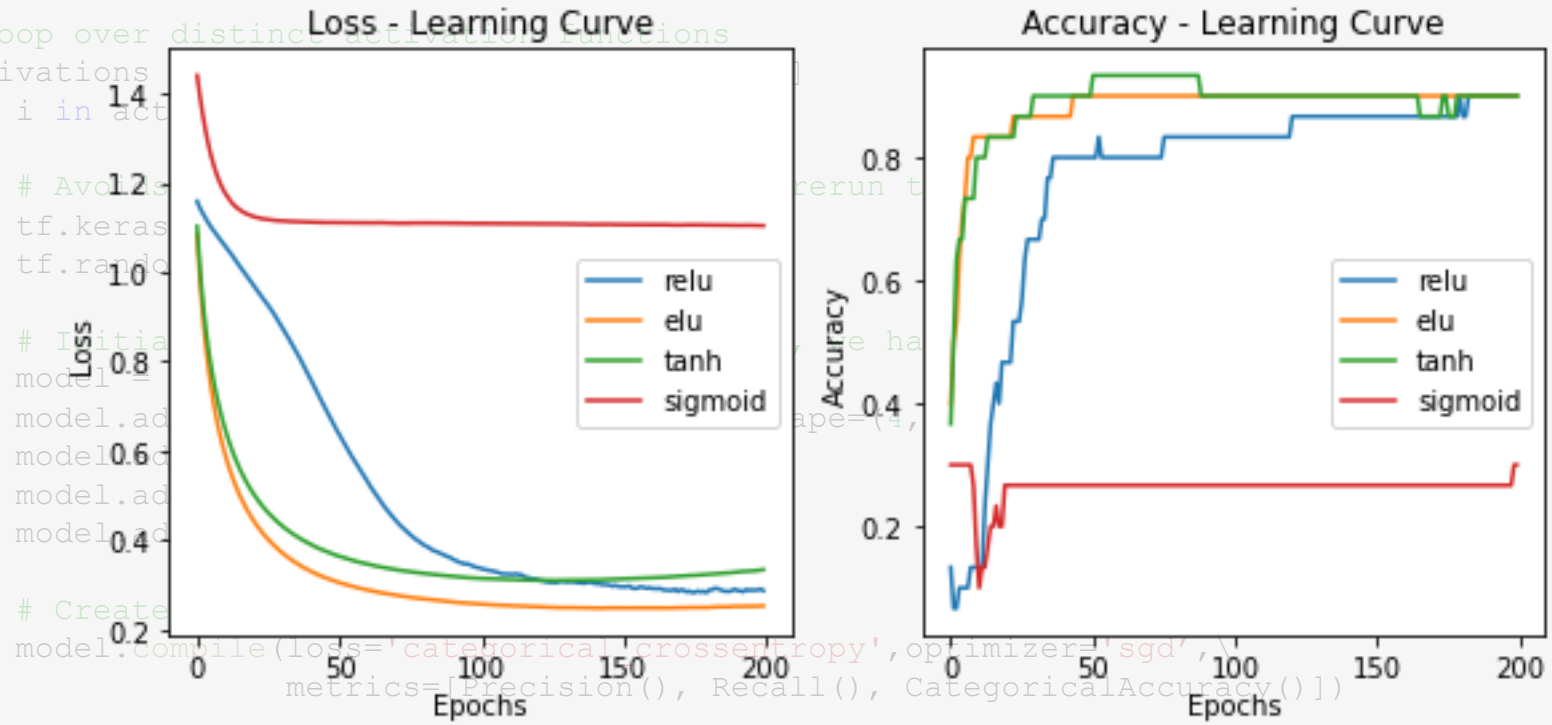
Training models with three hidden layers (10 nodes) and varying activation:

```
# Initialize dictionaries to hold the models
models = {}

# Loop over distinct activation functions
activations = ['relu','elu','tanh','sigmoid']
for i in activations:

    # Avoids making duplicate models if you rerun these cells
    tf.keras.backend.clear_session()
    tf.random.set_seed(54238789)

    # Initialize input layer (Note the shape, we have four input features)
    model = Sequential()
    model.add(Dense(10,activation=i,input_shape=(4,)))
    model.add(Dense(10,activation=i))
    model.add(Dense(10,activation=i))
    model.add(Dense(3,activation='softmax'))

    # Create the model
    model.compile(loss='categorical_crossentropy',optimizer='sgd',\
                  metrics=[Precision(), Recall(), CategoricalAccuracy()])

    # Train the model
    models[i]=model.fit(x=scaler.transform(x_train),\
                        y=to_categorical(y_train),epochs=200,verbose=0,\
                        batch_size=10, validation_data=\
                        (scaler.transform(x_test), to_categorical(y_test)))
…Plot functions…
```

Classification with Deep Model (varying activation)

Training models with three hidden layers (10 nodes) and varying activation:

```
# Initialize dictionaries to hold the models
models = {}

# Loop over distinct activation functions
activations
for i in act

    # Avo                                    rerun t
    tf.keras
    tf.rando

    # Initia                                 e ha
    model =
    model.ad                                   ape=(4,
    model.ad
    model.ad
    model.ad

    # Create
    model.compile(loss='categorical_crossentropy',optimizer='sgd',\
                  metrics=[Precision(), Recall(), CategoricalAccuracy()])

    # Train the model
    models[i]=model.fit(x=scaler.transform(x_train),\
                        y=to_categorical(y_train),epochs=200,verbose=0,\
                        batch_size=10, validation_data=\
                        (scaler.transform(x_test), to_categorical(y_test)))
…Plot functions…
```



Classification with Deep Model (varying optimizer)

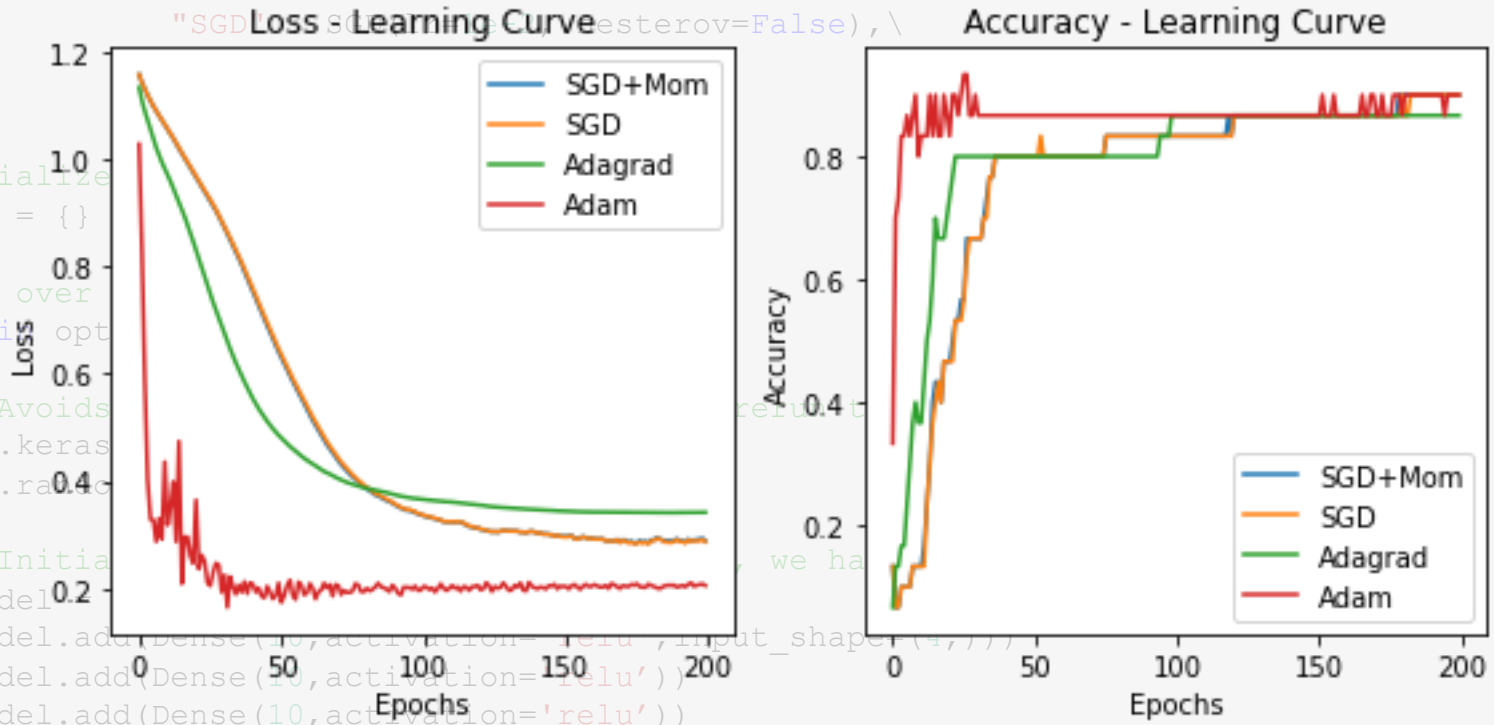
Training models with three hidden layers (10 nodes) and varying optimizer:

```
from tensorflow.keras.optimizers import SGD, Adagrad, Adam

optimizers = { "SGD+Mom": SGD(lr=1e-2, nesterov=True,momentum=0.01),\
               "SGD" : SGD(lr=1e-2, nesterov=False),\
               "Adagrad": Adagrad(lr=1e-2),\
               "Adam" : Adam(lr=1e-2) }

# Initialize dictionaries to hold the models
models = {}

# Loop over distinct optimizers
for i in optimizers.keys():

    # Avoids making duplicate models if you rerun these cells
    tf.keras.backend.clear_session()
    tf.random.set_seed(54238789)

    # Initialize input layer (Note the shape, we have four input features)
    model = Sequential()
    model.add(Dense(10,activation='relu',input_shape=(4,)))
    model.add(Dense(10,activation='relu'))
    model.add(Dense(10,activation='relu'))
    model.add(Dense(3,activation='softmax'))

    # Create the model
    model.compile(loss='categorical_crossentropy',optimizer=optimizers[i],\
    metrics=[Precision(), Recall(), CategoricalAccuracy()])

…Omitted Omitted .fit() call and plotting commands…
```

Classification with Deep Model (varying optimizer)

Training models with three hidden layers (10 nodes) and varying optimizer:

```
from tensorflow.keras.optimizers import SGD, Adagrad, Adam

optimizers = { "SGD+Mom": SGD(lr=1e-2, nesterov=True,momentum=0.01),\
               "SGD": SGD(lr=1e-2, nesterov=False),\

# Initialize
models = {}

# Loop over
for i in opt
    # Avoids
    tf.keras
    tf.ra

    # Initia
    model
    model.add(Dense(10,activation='relu',input_shape=(4,)))
    model.add(Dense(10,activation='relu'))
    model.add(Dense(10,activation='relu'))
    model.add(Dense(3,activation='softmax'))

    # Create the model
    model.compile(loss='categorical_crossentropy',optimizer=optimizers[i],\
    metrics=[Precision(), Recall(), CategoricalAccuracy()])

…Omitted Omitted .fit() call and plotting commands…
```



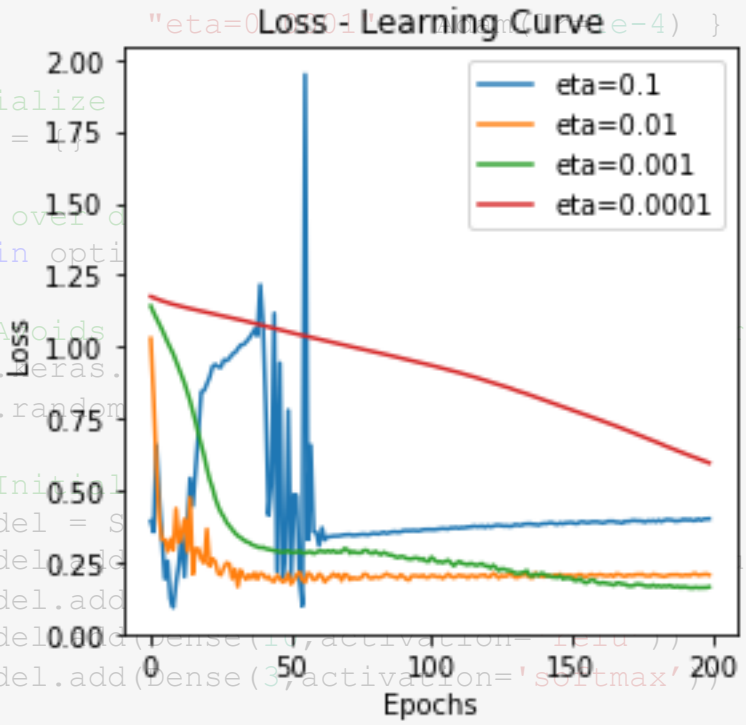
Classification with Deep Model (varying learning rate)

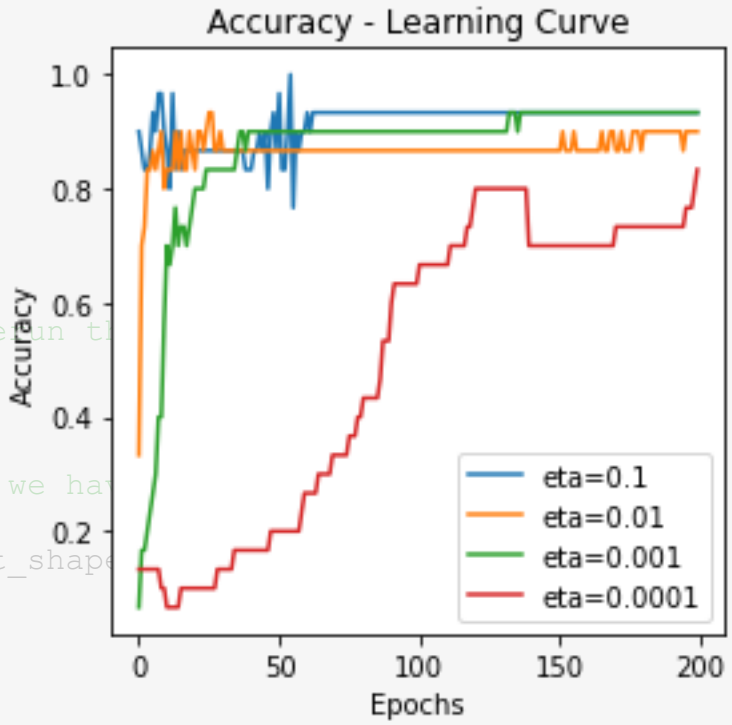
Training models with three hidden layers (10 nodes) and varying learning:

```
optimizers = { "eta=0.1" : Adam(lr=1e-1),\
               "eta=0.01" : Adam(lr=1e-2),\
               "eta=0.001" : Adam(lr=1e-3),\
              "eta=0.0001" : Adam(lr=1e-4) }

# Initialize dictionaries to hold the models
models = {}

# Loop over distinct optimizers
for i in optimizers.keys():

    # Avoids making duplicate models if you rerun these cells
    tf.keras.backend.clear_session()
    tf.random.set_seed(54238789)

    # Initialize input layer (Note the shape, we have four input features)
    model = Sequential()
    model.add(Dense(10,activation='relu',input_shape=(4,)))
    model.add(Dense(10,activation='relu'))
    model.add(Dense(10,activation='relu'))
    model.add(Dense(3,activation='softmax'))

    # Create the model
    model.compile(loss='categorical_crossentropy',optimizer=optimizers[i],\
    metrics=[Precision(), Recall(), CategoricalAccuracy()])

…Omitted Omitted .fit() call and plotting commands…
```

Classification with Deep Model (varying learning rate)

Training models with three hidden layers (10 nodes) and varying learning:






```
optimizers = { "eta=0.1" : Adam(lr=1e-1),\
               "eta=0.01" : Adam(lr=1e-2),\
               "eta=0.001" : Adam(lr=1e-3),\
               "eta=0.0001" : Adam(lr=1e-4) }

# Initialize
models = {}

# Loop over
for i in opti

    # Avoids
    tf.keras.
    tf.random

    # Init
    model = S
    model.
    model.add
    model.add(Dense(10,activation='relu'))
    model.add(Dense(3,activation='softmax'))

    # Create the model
    model.compile(loss='categorical_crossentropy',optimizer=optimizers[i],\
    metrics=[Precision(), Recall(), CategoricalAccuracy()])

…Omitted Omitted .fit() call and plotting commands…
```