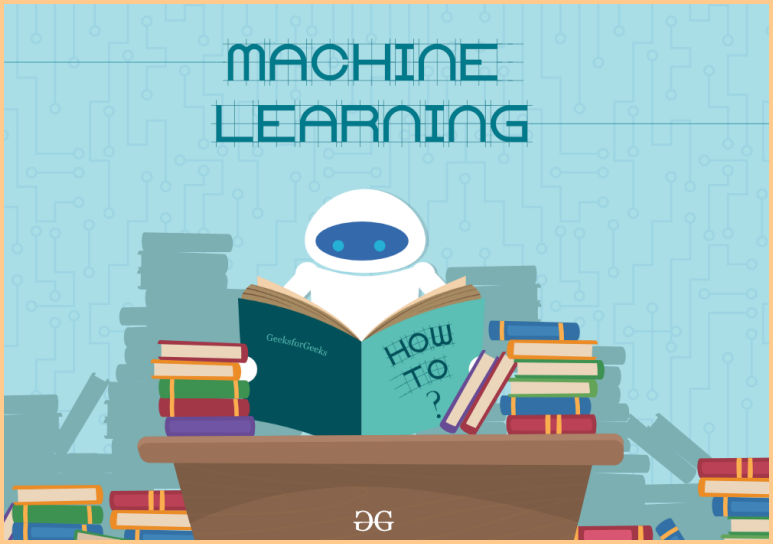
Machine Learning for beginners (Scikit-Learn and Keras)



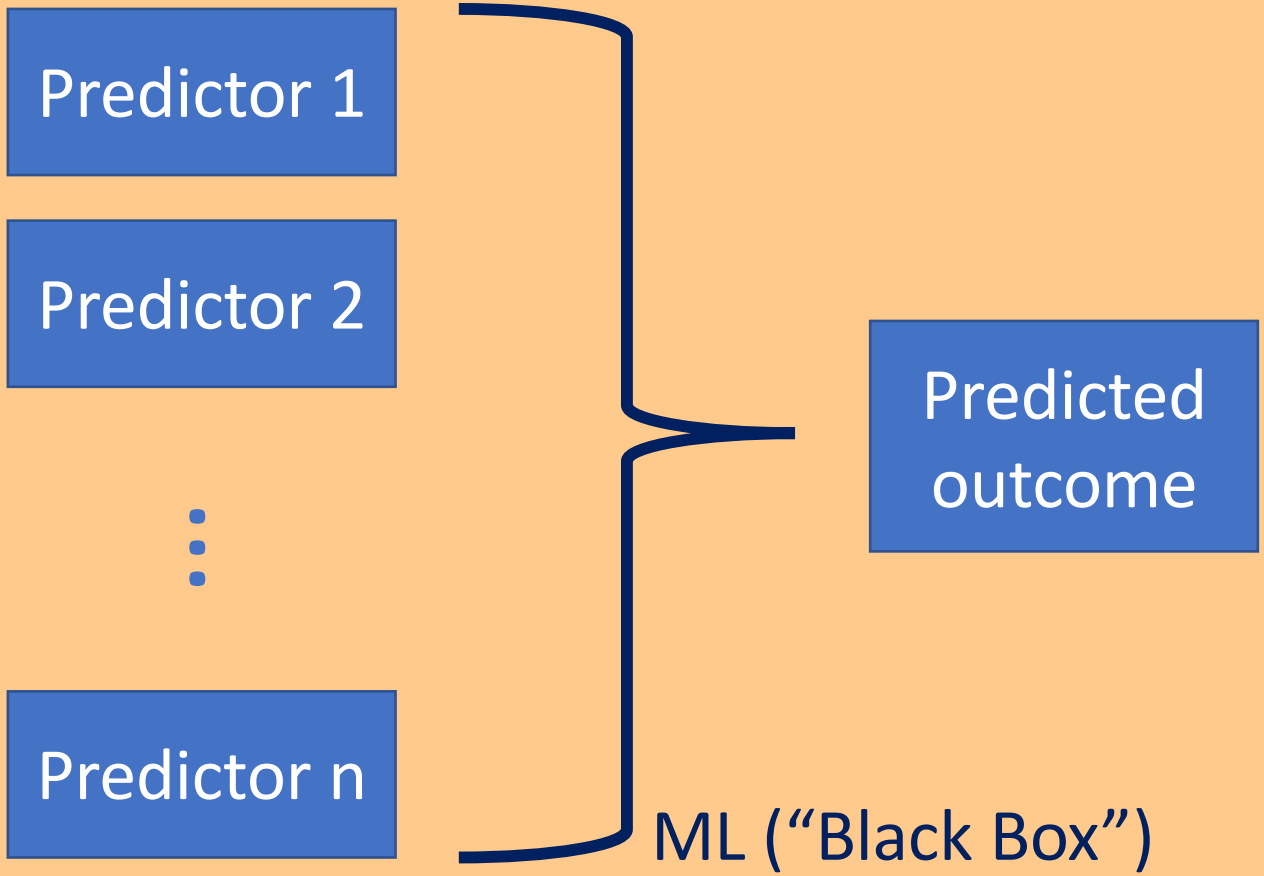
Eric Loken (DTC visitor)

Host: Jamie Wolff

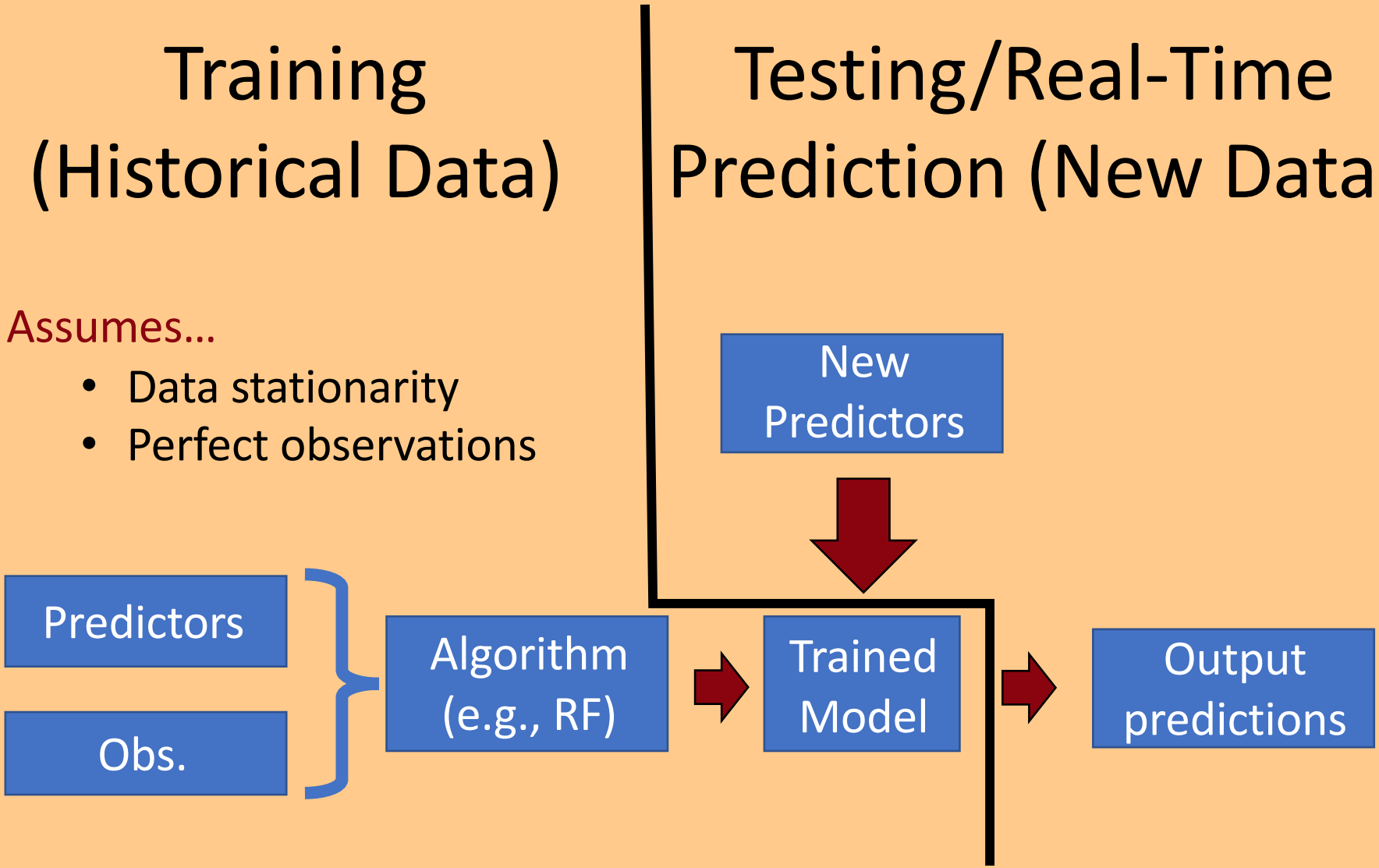
Ph.D. Advisor: Adam Clark

What is ML doing, conceptually?

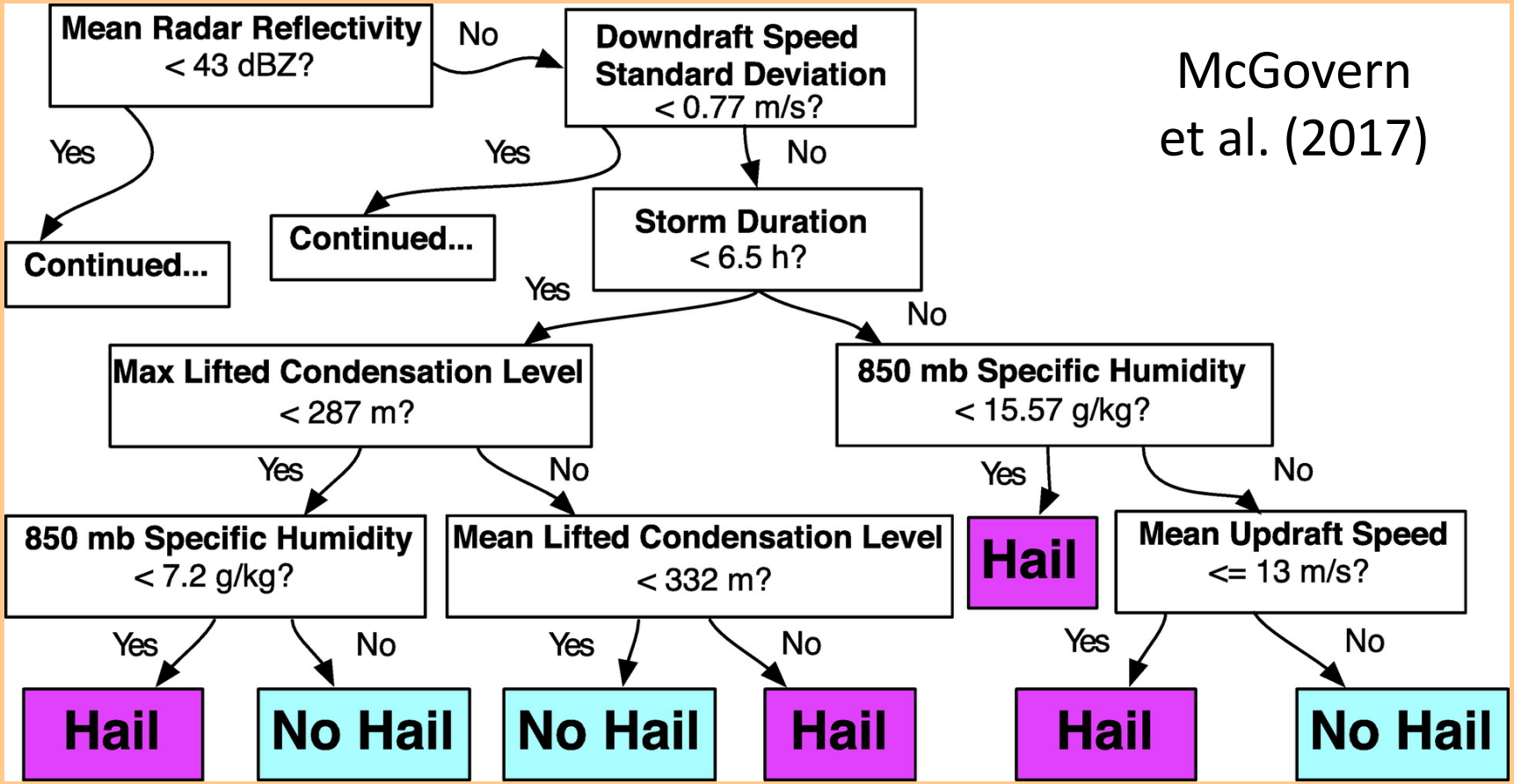
Maps predictors to outputs based on relationships learned from training



2 Stages of ML: Training and Testing



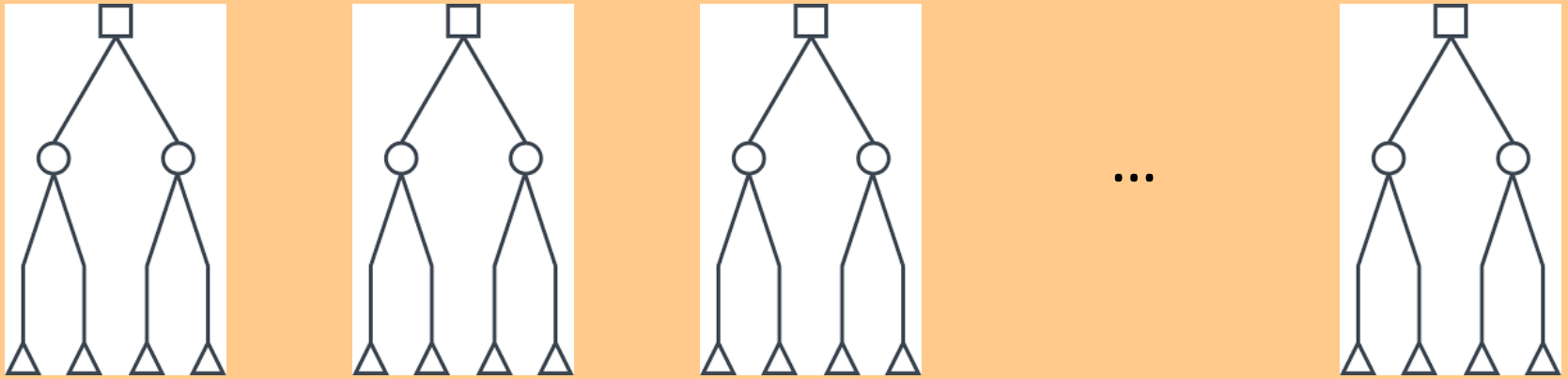
Decision tree: chooses the best splits to minimize a cost function



McGovern et al. (2017)

A Random Forest (RF; Breiman 2001) is an ensemble of decision trees

Tree 1 Tree 2 Tree 3 ... Tree n



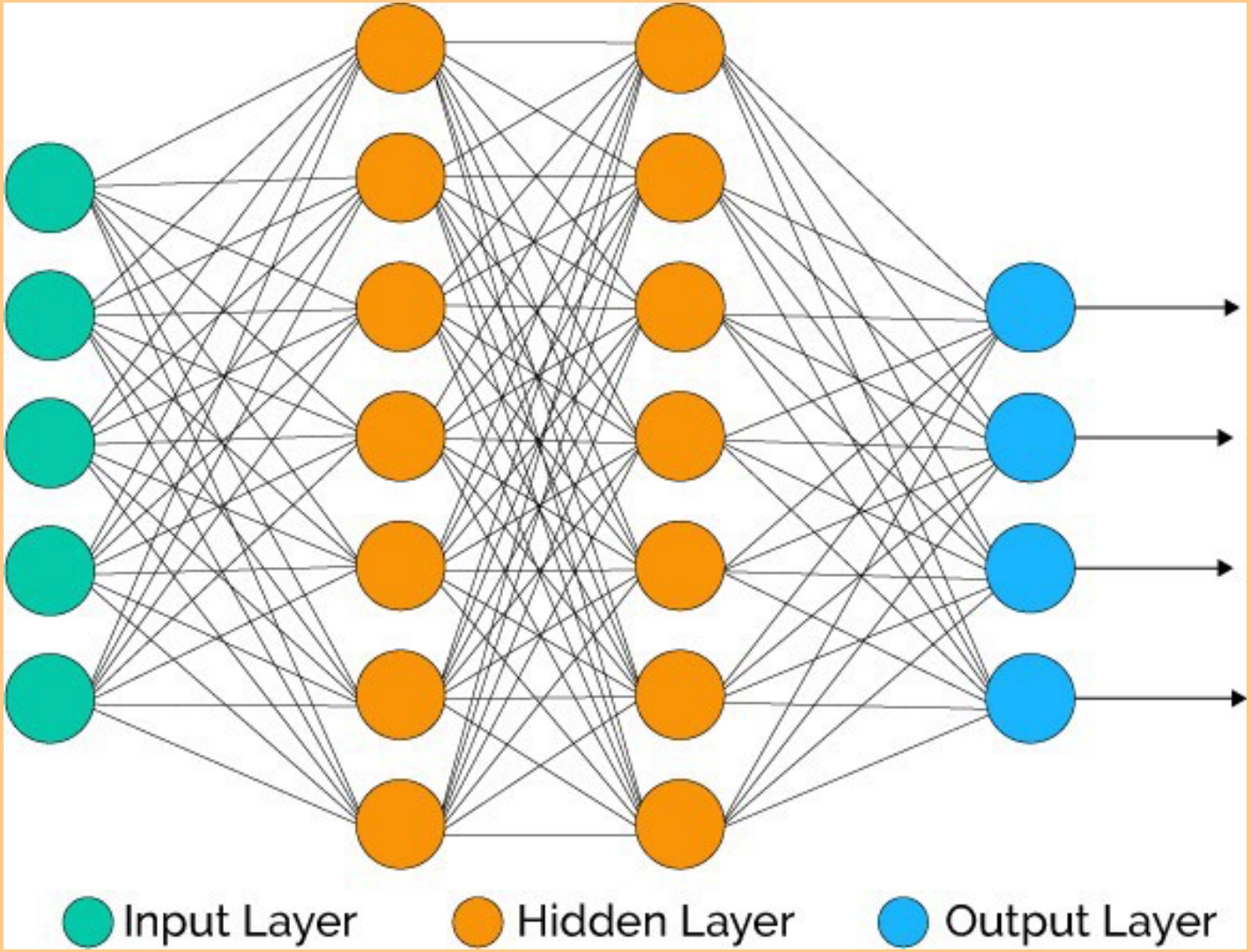
Each tree uses a random subset of training samples (bootstrapping)

Each node considers a subset of predictors on which to split the training samples.

Probs obtained at each leaf node of each tree

RF probs are the mean probs over the entire forest

Artificial Neural Networks (ANNs) learn weights between neurons

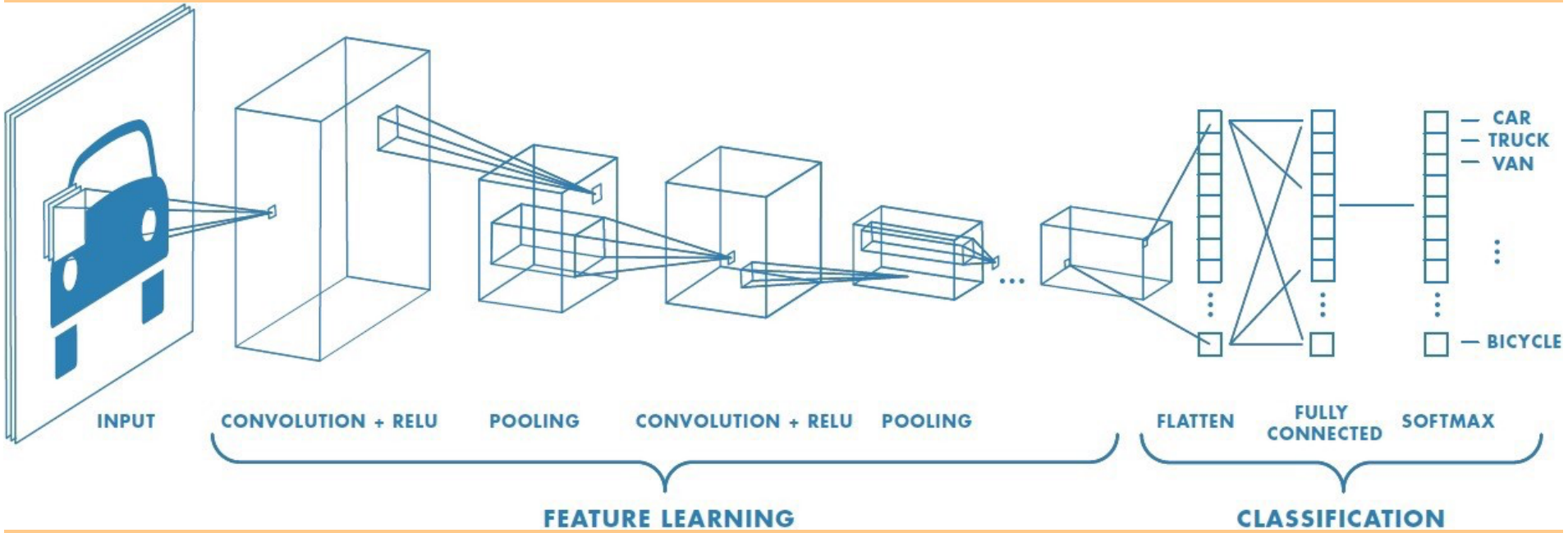


How do ANNs work?

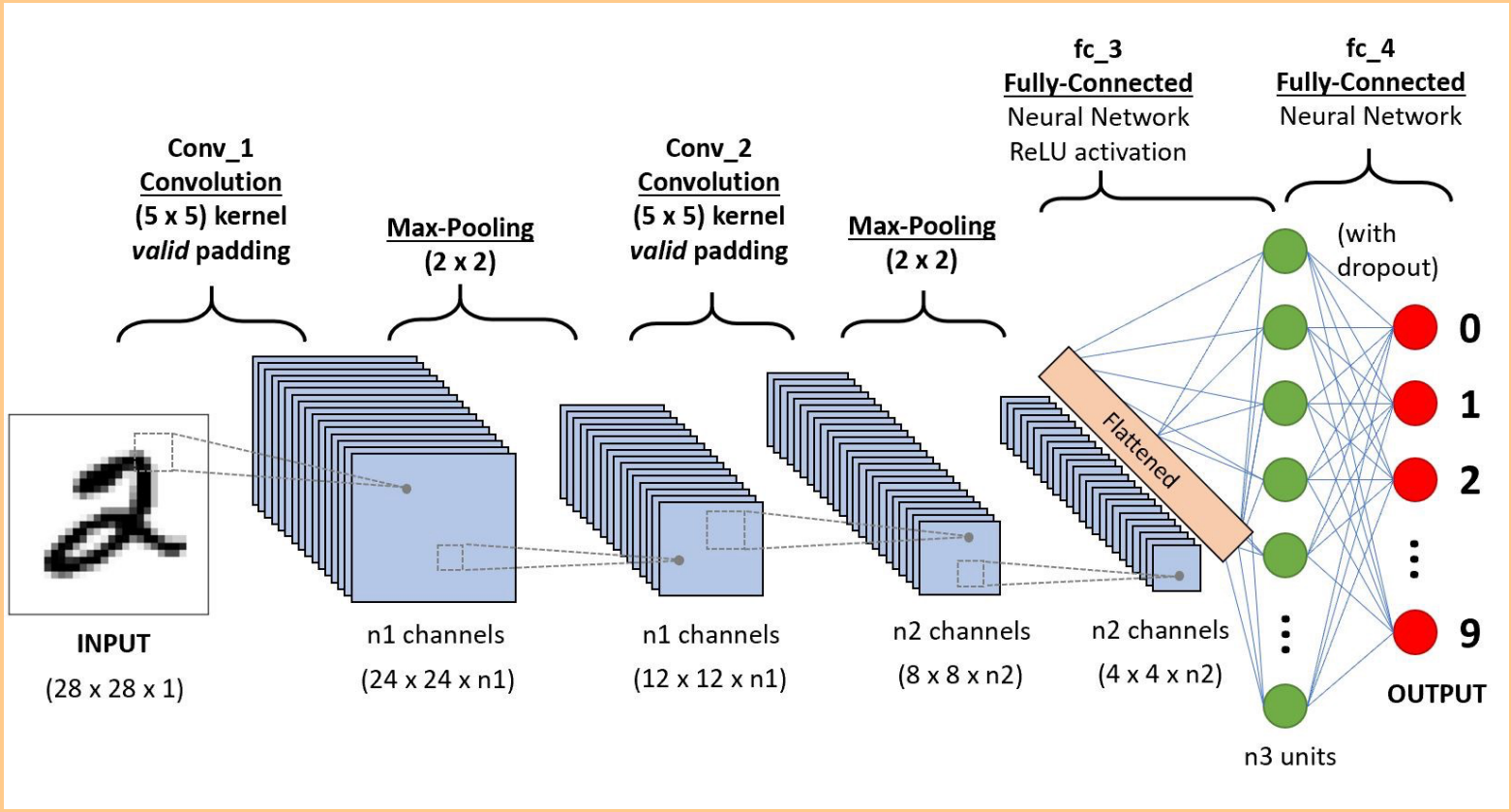
From *Deep Learning With Python* (Francois Chollet):

1. Draw a batch of training samples x and corresponding targets y.
2. Run the network on x (forward pass) to obtain predictions y_pred.
3. Compute the loss of the network on the batch (a measure of the mismatch between y_pred and y).
4. Update all weights of the network in a way that slightly reduces the loss on this batch.
    1. Compute the gradient of the loss with regard to the network's parameters (backward pass).
    2. Move the parameters in the opposite direction of the gradient (e.g., *w -= step*gradient*).

Convolutional neural networks (CNNs) learn spatial patterns



Convolutional neural networks (CNNs) learn spatial patterns



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

How convolution works

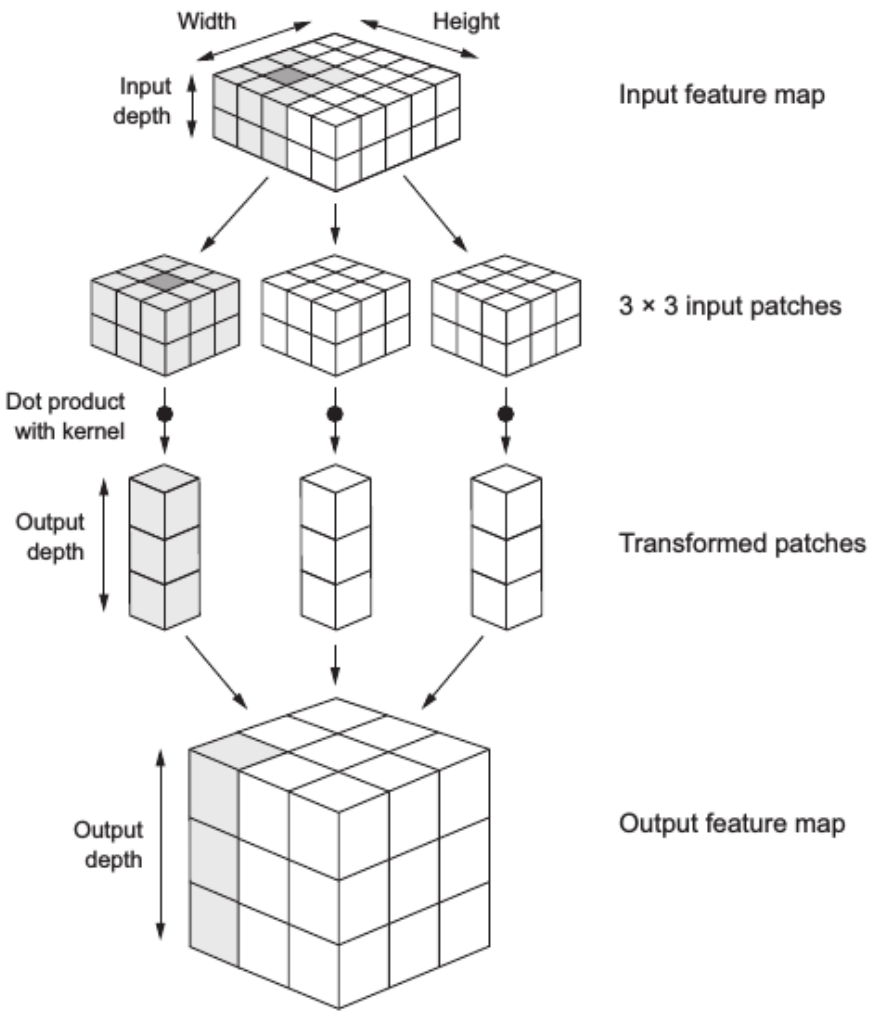


Figure 5.4 How convolution works

Convolution "slides," e.g., 3x3 windows over input image.

Each 3D patch transformed into 1D vector, then patches reassembled into 3D output map

Fig 5.4 in *Deep Learning With Python* by Francois Chollet.

How convolution works

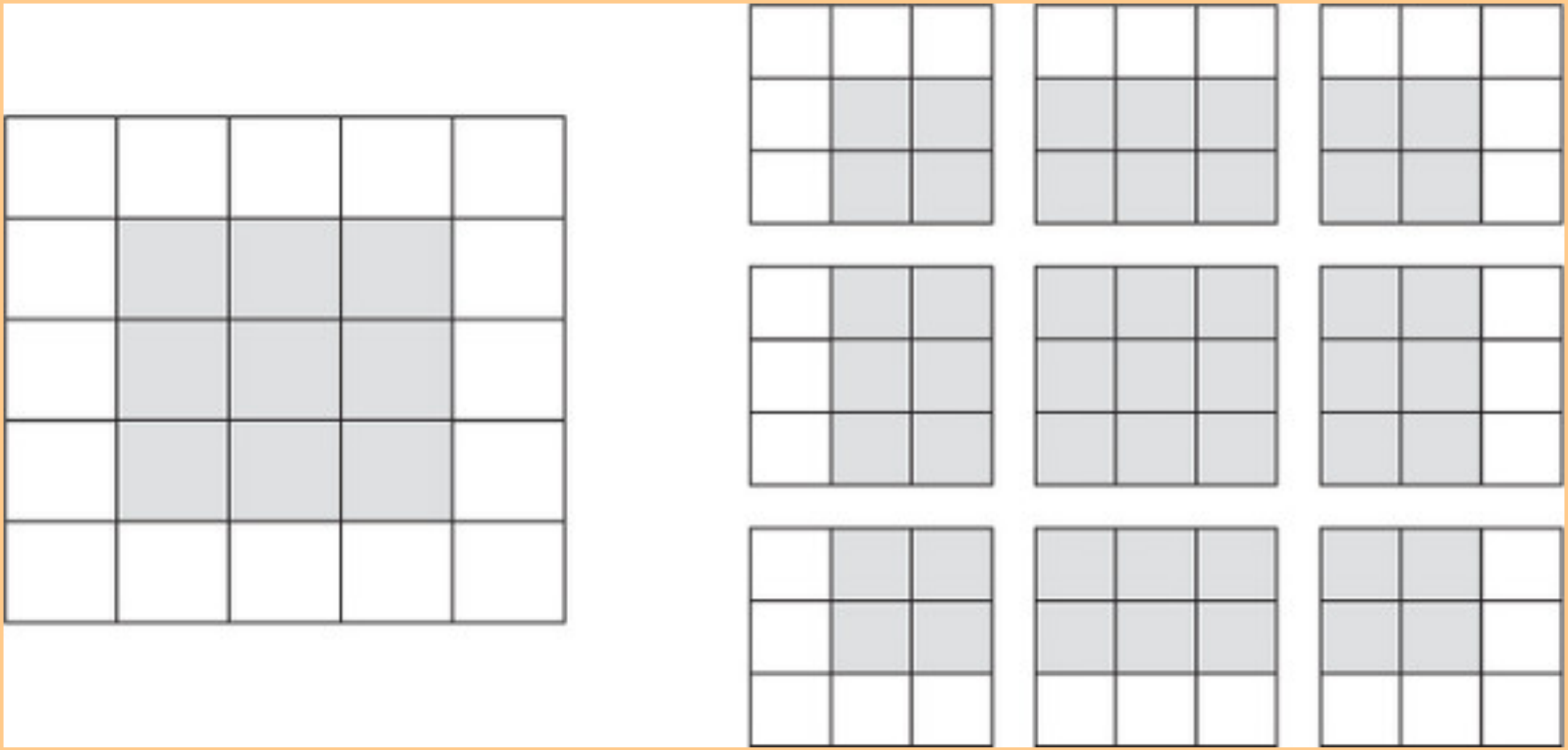
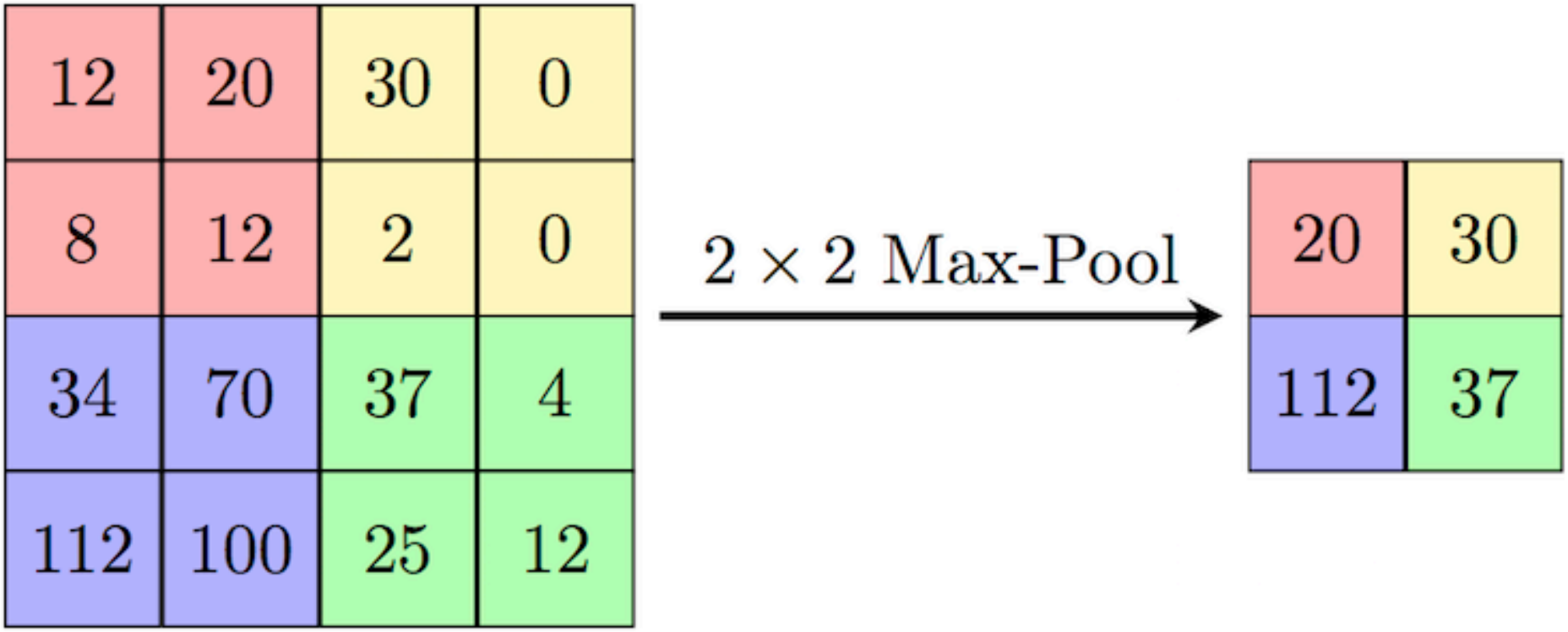


Figure 5.5 from Chollet

How Max Pooling Works



https://computersciencewiki.org/index.php/Max-pooling_/_Pooling

Depthwise Separable Convolution generally makes model lighter, better

1. Performs spatial convolution on each channel of input separately.
2. Mixes output channels using a pointwise (i.e., 1x1) convolution.

Separately learns spatial features and channel features—

Can be especially good if spatial locations in the input are highly correlated but different channels are fairly independent

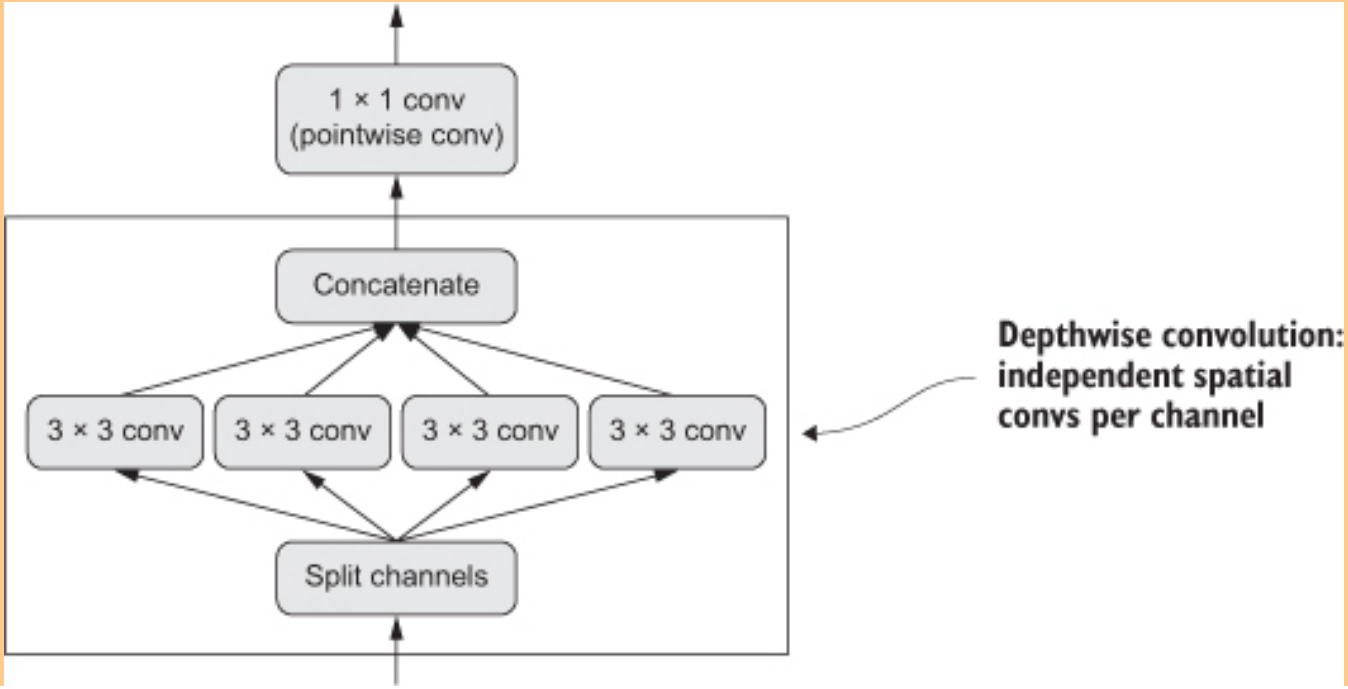


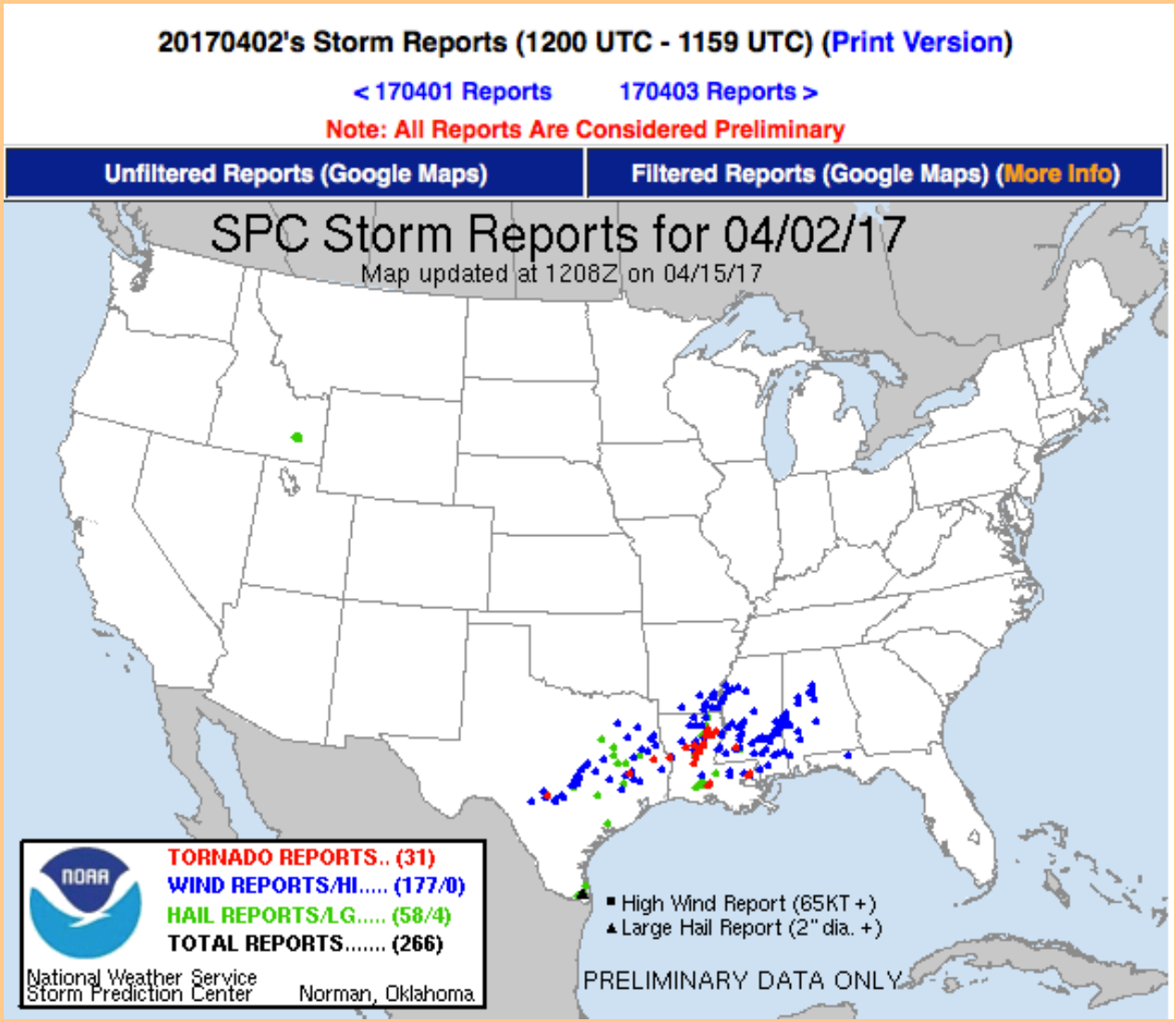
Fig. 7.16 from Chollet

Task: Train a RF, neural network, and deep neural network to predict severe hail based on SSEO forecast variables and observed storm reports.

Predictors: SSEO ensemble mean forecast variables (24-h max/mean on 80 km grid)

- 2-m Relative humidity (**RELH**)
- 0-6 km shear (**SHEAR**)
- 2-m temperature (K; **TMPK**)
- 2-m dew point temperature (K; **DWPK**)
- Updraft speed (**UPDRHM**)
- 2-5 km UH (**UPHLHM**)
- CAPE (**CAPE**)
- CAPE*SHEAR product (**CAPESHEAR**)
- Simulated 1-h accumulated precip. (**P01M**)
- Simulated 1-km AGL Reflectivity (**REFDHM**)

Predictands/Targets:
Observed SPC severe hail reports



How do I use Scikit-Learn (Sklearn)?

Step 1: Obtain archive of historical predictors/predictands (e.g., SSEO forecast variables and observed storm reports) for training/testing.

Step 2: Put training data in a format that Sklearn can use.

Step 3: Divide dataset into separate training/testing/validation sets.

Step 4: Compile the model. Set hyper-parameters (e.g., number of trees, stopping criteria, splits based on entropy or gini, etc.).

Step 5: Train the model.

Step 6: Use the trained model to make predictions on unseen data.

Step 7: Verification

Step 0a: Set up a virtual environment on Cheyenne for ML

Instructions: https://www2.cisl.ucar.edu/resources/computational-systems/cheyenne/software/python

ncar_pylib -c 20190627 /glade/work/$USER/venv_ml_tutorial

ncar_pylib venv_ml_tutorial

To activate:
source /glade/work/$USER/venv_ml_tutorial/bin/activate.csh

To deactivate:
deactivate

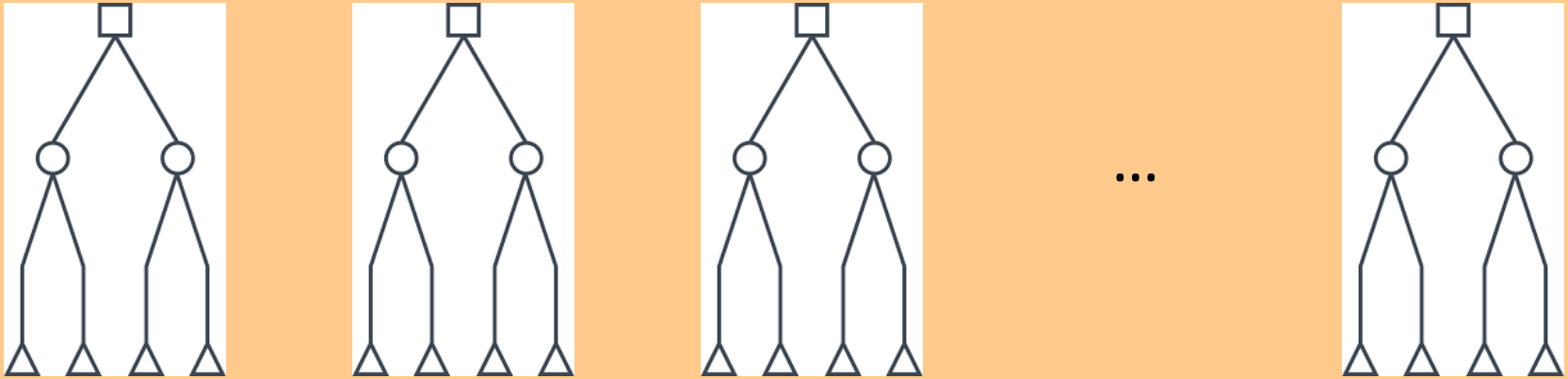
Step 0a: Set up a virtual environment on Cheyenne for ML

If you don't want to create your own virtual environment, you can source mine:

source /glade/work/eloken/venv_20190627/bin/activate.csh

Task 1: Use a RF to predict severe hail from SSEO data using sklearn

Tree 1 Tree 2 Tree 3 ... Tree n



Step 1: Obtain archive of historical predictors/predictands

1. Make a new directory in your /glade/work directory and copy over the data into that directory.

   cd /glade/work/$username
   mkdir ML_tut ; cd ML_tut
   cp /glade/work/eloken/ML_tutorial_final/files/data/.tar.gz .*

2. While we're at it, let's create new directories for us to work in throughout the tutorial.

   mkdir rf nn dl

Step 1: Obtain archive of historical predictors/predictands

3. Make new directories for predictor (all_x) and predictand (all_y) data.

mkdir all_x all_y

4. Move predictor/predictand data into respective directories:

mv all_x.tar.gz all_x ;

mv all_y.tar.gz all_y

5. Change into each directory and Untar the data:

cd all_x ; tar -xzvf all_x.tar.gz;

cd ../all_y; tar -xzvf all_y.tar.gz;

Step 1: Obtain archive of historical predictors/predictands

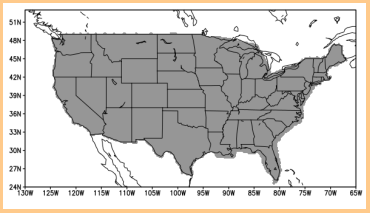
6. Change to the rf directory and copy over the relevant RF files.

cd ../rf
cp /glade/work/eloken/ML_tutorial_final/files/rf/rf_files.tar.gz .

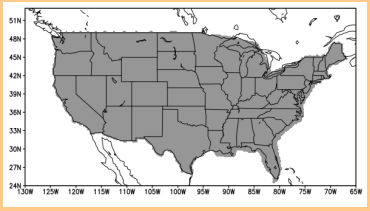
7. Untar the files

tar -xzvf rf_files.tar.gz

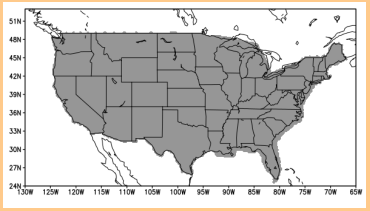
Step 2:Need to put data in proper format



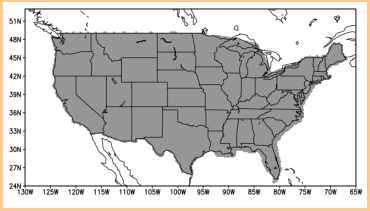
Day 1 CAPE



Day 1 SHEAR

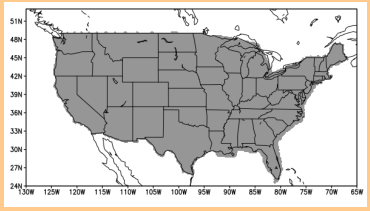


Day 1 2-m RH

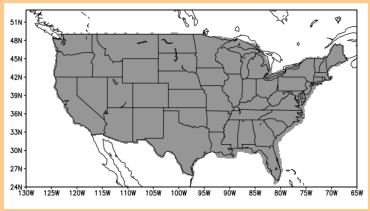


Day 1 UH

...



Day d 2-m RH



Day d UH

	Predictor 1 (e.g., CAPE)	...	Predictor m (e.g., 0-6 km Shear)
Sample 1 (e.g., Day 1, point 1)			
...			
Sample n (e.g., Day d, point p)			

Step 2: Data Preprocessing. Put data into “table” format

	Predictor 1 (e.g., CAPE)	Predictor 2 (e.g., 2-m Temp)	Predictor 3 (e.g., UH)	…	Predictor m (e.g., 0-6 km Shear)
Sample 1 (e.g., Day 1, point 1)					
Sample 2 (e.g., Day 1, point 2)					
…					
Sample n (e.g., Day d, point p)					

Step 2: Data Preprocessing. Put data into “table” format

	Obs.
Sample 1 (e.g., Day 1, point 1)	
Sample 2 (e.g., Day 1, point 2)	
…	
Sample n (e.g., Day d, point p)	

Step 2: Preprocess the data

1. We'll employ a trick to more efficiently reformat the data. Start by making a fcst and obs directory.

mkdir fcst obs

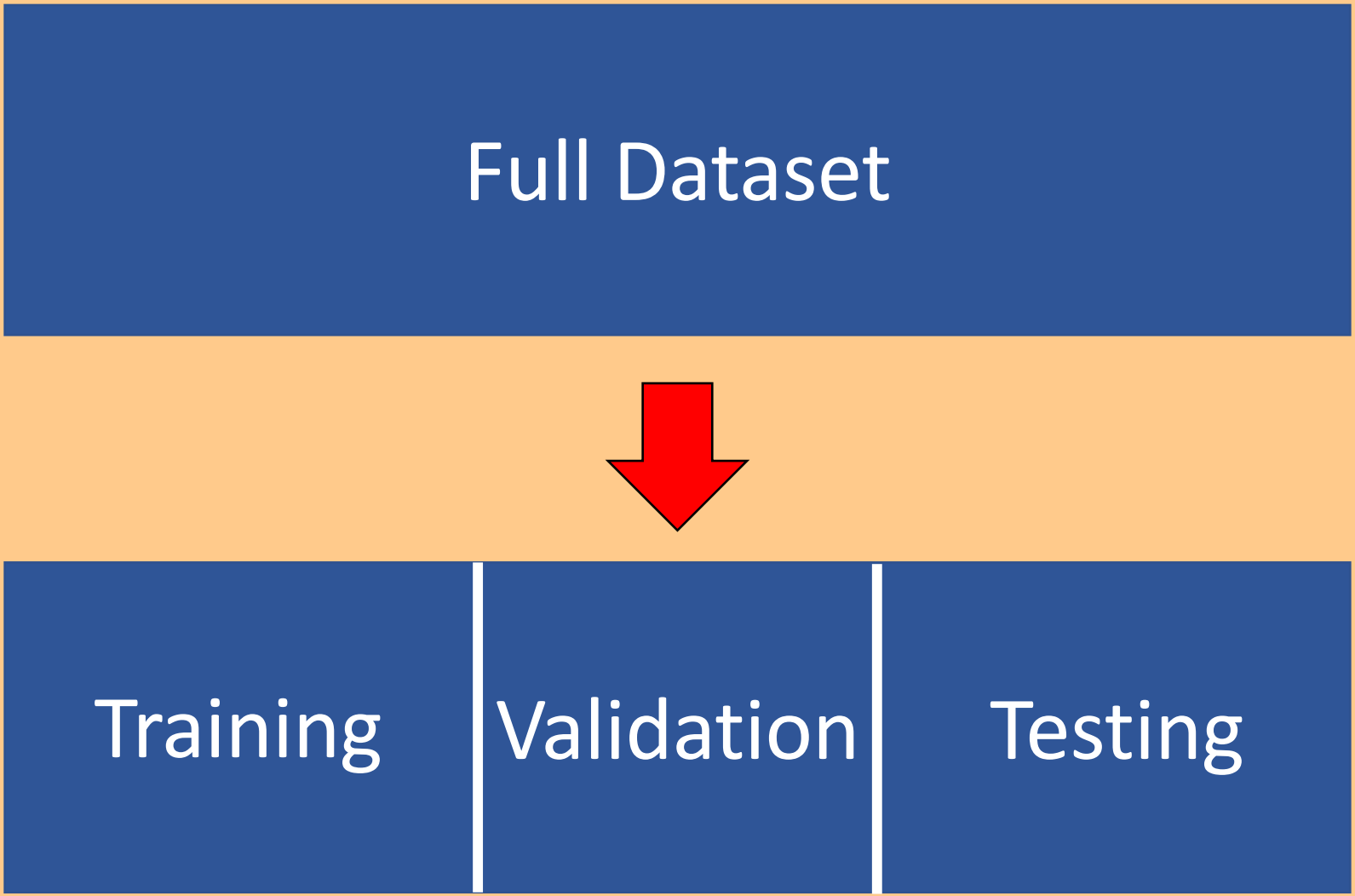
2. The trick is to do the preprocessing using a separate script. We'll point to the "raw" data in *preprocess_rf.py* and then output nicely formatted data in the *fcst* and *obs* directories we just created.

In *preprocess_rf.py*, make sure *indir_forecast = "../all_x"* and *indir_observations = "../all_y"*

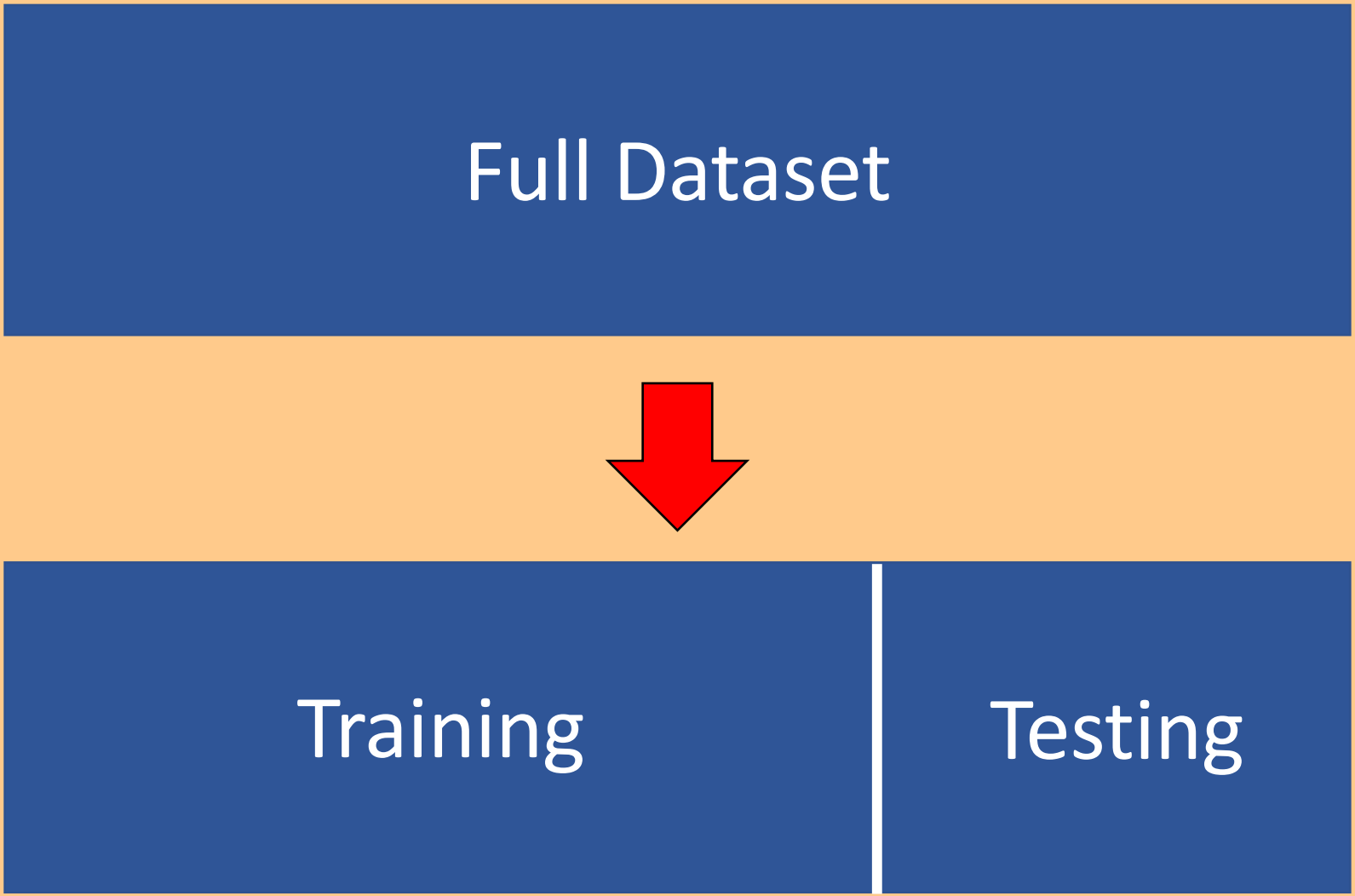
3. Run the preprocessing script. This will output preprocessed unformatted binary files to the *fcst/* and *obs/* directories we just created.

python3 preprocess_rf.py

Step 3: Divide dataset into separate training/testing/validation sets



NOTE: For RF, we will just divide into training/testing sets



NOTE: For RF, we will just divide into training/testing sets

Full Dataset
221 days:
20150421 - 20170624

Training	Testing
20150421- 20160630	20170402- 20170624

NOTE: Dataset is implicitly divided by passing in lists of training and testing dates

Training 20150421-20160630	Testing 20170402-20170624

Open *run_rf.py* (e.g., *vi run_rf.py*).

Note *test_date_file = test_dates.txt* and *train_date_file = train_dates.txt*

Step 4: Compile the Random Forest (RF)

1. Open *run_rf.py* and search for “Train the RF classifier.” The following command compiles the RF with the hyperparameters that we assign.

```
clf = RandomForestClassifier(n_jobs=n_proc, \
        n_estimators = n_trees, criterion = 'entropy',\
        max_depth = best_max_depth, min_samples_leaf =
        best_min_samples_leaf, max_features="sqrt")
```

What does the above line mean? Let’s break it down…

Step 4: Compile the Random Forest (RF)

```
clf = RandomForestClassifier(n_jobs=n_proc, \
                n_estimators = n_trees, criterion = 'entropy',\
                max_depth = best_max_depth, min_samples_leaf =
                best_min_samples_leaf, max_features="sqrt")
```

n_jobs: The number of parallel processors to use when training the RF. Here, we have it set to 36 for Cheyenne.

criterion: The loss function used to determine the “best” split.

Max_depth: The maximum depth a tree can reach before splitting is terminated. Here, we have it set to 15.

Min_samples_leaf: Another stopping criterion. The minimum number of examples required at a leaf node. Here, 20.

Max_features: The number of features over which to consider splits at each node. Here, the square root of the total number of features.

Step 5: Train the Random Forest (RF)

1. This step is easy—a 1-liner!

   clf.fit(training_x, training_y)

   (Where *clf* is the RF object we created in the previous step.)

Step 6 (Optional): Save trained RF with pickle

Pickle can save the trained RF object to a file, which can then be used to make predictions in other scripts.

1. Make sure pickle is imported:
   import pickle at the top of the script.
2. Execute the following lines of code:

```
clf_pkl = open(clf_filename, 'wb')
pickle.dump(clf,clf_pkl)
clf_pkl.close()
```

Step 7: Use the trained RF to make predictions on the test data.

1. Probabilistic predictions can be made by calling the *predict_proba* function on the RF object.

   clf_probs = clf.predict_proba(testing_x)[:,1]

2. Output probabilities will be in a 1-d array, so we have to do some accounting to get individual-day probs. Here, that's done with the *np.array_split* function and the *save_probs* method. As a result, individual-day hail probabilities are saved in .txt files to be read by another script for evaluation.

To execute steps 3-7:

1. First, create a directory to hold the resulting prediction text files:

   mkdir outfiles

2. Run the *run_rf.py* script by submitting a batch job.

   qsub train_rf_qsub

Step 8: Evaluate the skill of the RF probabilities using the test_y data

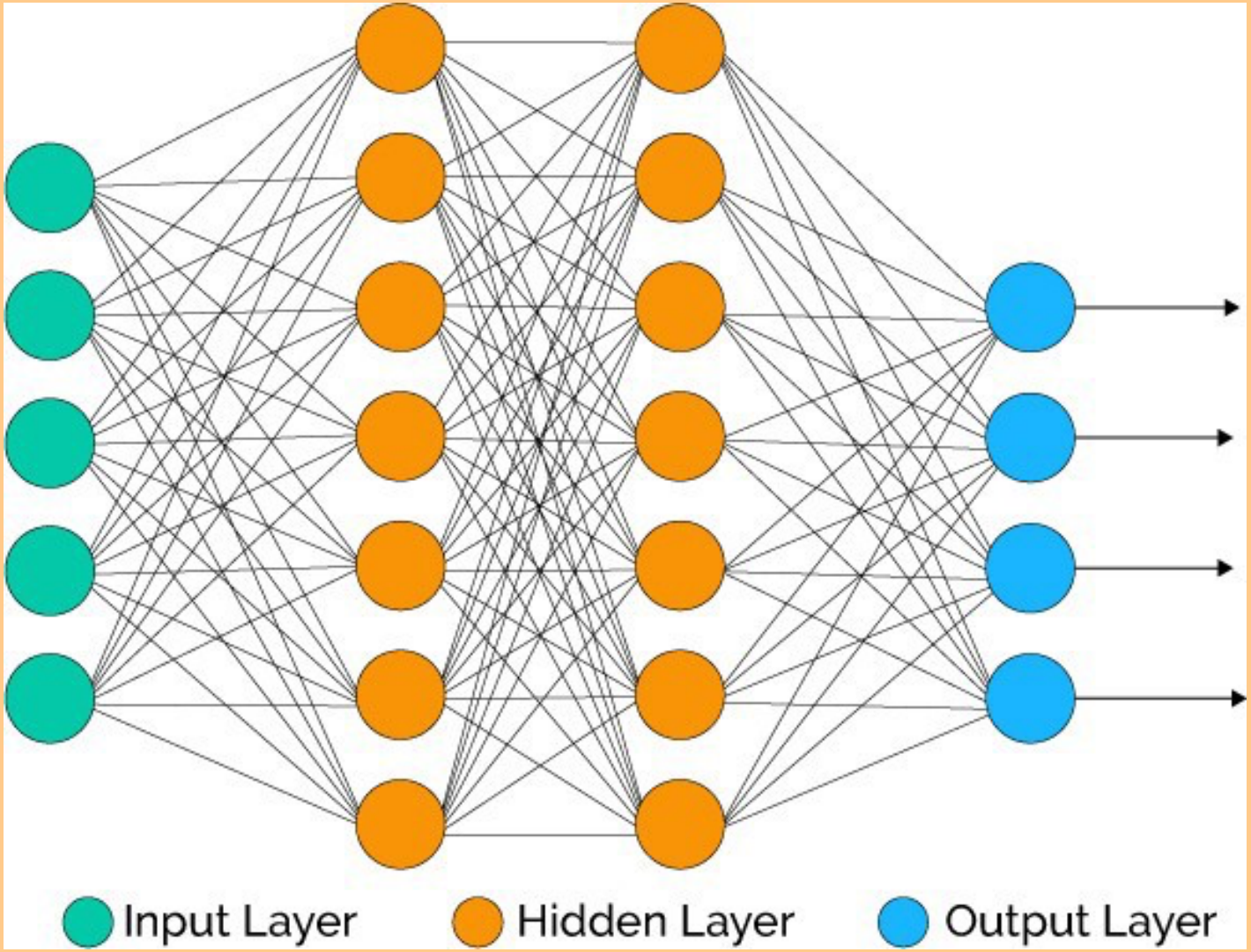
1. To obtain overall skill metrics, attributes, performance, and ROC diagrams, execute:

   python3 skill_scores_evaluate_rf.py

2. To visualize what these forecasts look like on individual days, execute:

   python3 evaluate_rf.py

Task 2: Use a NN instead of a RF to make predictions with sklearn



Step 1: Download relevant files

1. Change into your nn directory

   cd ../nn

2. Download nn files

   cp /glade/work/eloken/ML_tutorial_final/files/nn/nn_files.tar.gz .

3. Untar the files.

   tar -xzvf nn_files.tar.gz

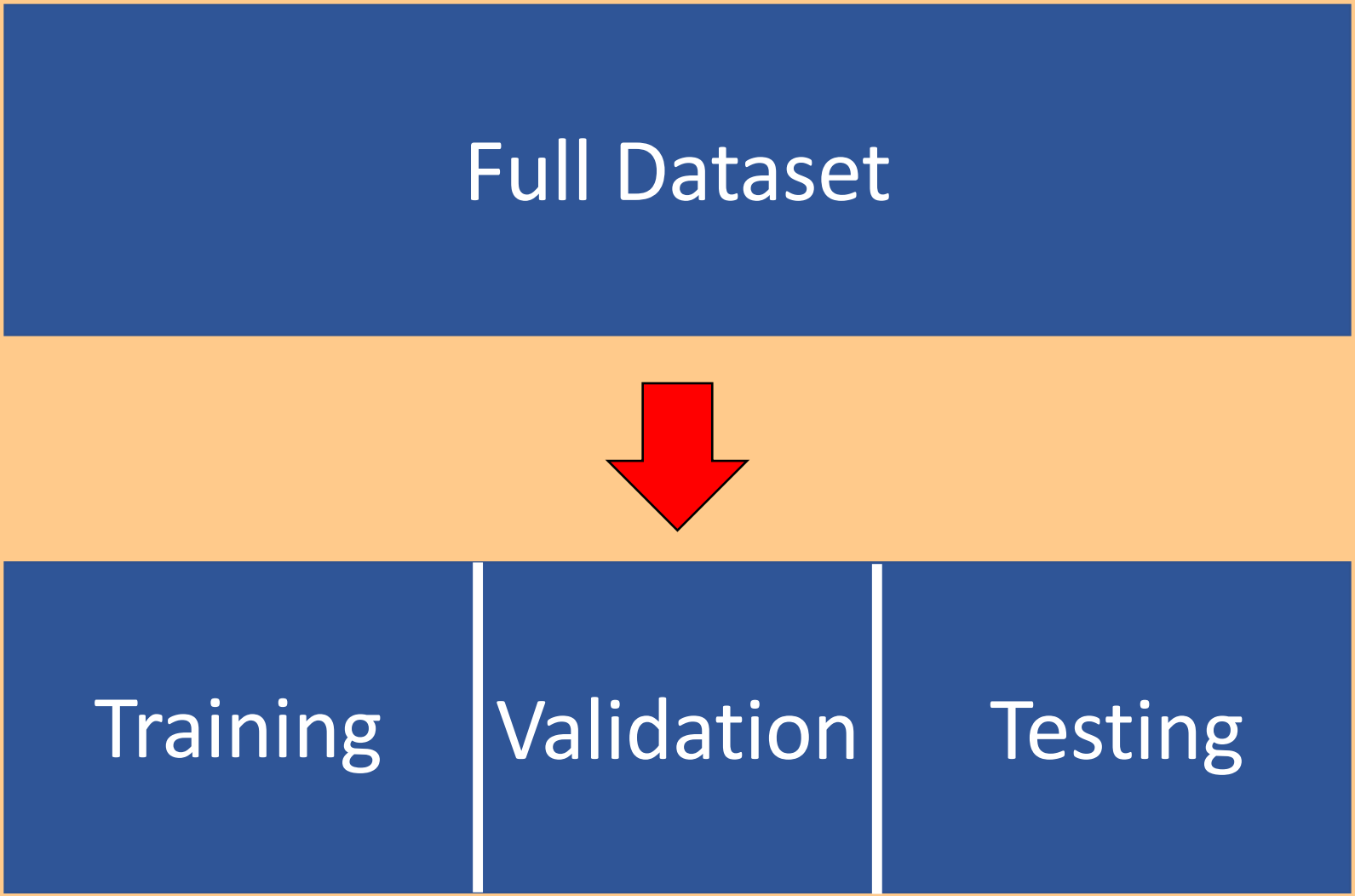
4. While we're here, let's also make the directory to hold future output files:

   mkdir outfiles

Step 2: Preprocess the data

1. Since we already did this for the RF, we can just use the results from before. We'll just have to point to those files (in ../rf/fcst and ../rf/obs) in the *run_nn* script.

Step 3: Divide dataset into separate training/testing/validation sets



Again, for simplicity, we won't tune hyperparameters

Full Dataset
221 days:
20150421 - 20170624

Training	Testing
20150421- 20160630	20170402- 20170624

Step 4: Normalize the inputs (important step!)

1. While the RF is capable of working directly with the predictor variables, the NN is not! The weights of the NN are best applied to normalized predictor variables. How to normalize? Subtract by the training sample mean and divide by the training sample standard deviation. This is done verbosely in *run_nn.py* in the following lines of code:

```
train_mean = training_x.mean(axis=0)
train_sd = training_x.std(axis=0)
delta = 0.000001 #prevent divide by zero error
train_sd += delta
np.save("train_mean.npy", train_mean) #save for future
np.save("train_std.npy", train_sd)
training_x = (training_x - train_mean)/train_sd
testing_x = (testing_x - train_mean)/train_sd
```

Step 4: Compile the NN

1. Open *run_nn.py* and search for “Train the NN”. The following command compiles the RF with the hyperparameters that we assign.

```
nn = MLPClassifier(hidden_layer_sizes=(64, 12, ),
        activation='relu', solver='adam', alpha=0.00001, \
        batch_size='auto', learning_rate='adaptive', \
        learning_rate_init=0.0001, max_iter=100, \
        shuffle=True, random_state=None,\
        tol=0.0001, verbose=True, warm_start=True, \
        momentum=0.8, early_stopping=False,\
        validation_fraction=0.2, beta_1=0.9, beta_2=0.999, \
        epsilon=1e-08, n_iter_no_change=30)
```

The NN has many more hyperparameters to tune than the RF. Let’s break down what this all means.

Step 4: Compile the NN

```
nn = MLPClassifier(hidden_layer_sizes=(64, 12, ),
        activation='relu', solver='adam', alpha=0.00001, \
        batch_size='auto', learning_rate='adaptive', \
        learning_rate_init=0.0001, max_iter=100, \
                shuffle=True, random_state=None,\
                tol=0.0001, verbose=True, warm_start=True, \
                momentum=0.8, early_stopping=False,\
                validation_fraction=0.2, beta_1=0.9, beta_2=0.999, \
                epsilon=1e-08, n_iter_no_change=30)
```

hidden_layer_sizes: The number of neurons in each hidden layer. Here, there are two hidden layers. The first has 64 neurons; the second has 12.

activation: The nonlinear activation function the NN uses. 'Relu' stands for rectified linear unit (y = max(0,x)).

Step 4: Compile the NN

```
nn = MLPClassifier(hidden_layer_sizes=(64, 12, ),
        activation='relu', solver='adam', alpha=0.00001, \
        batch_size='auto', learning_rate='adaptive', \
        learning_rate_init=0.0001, max_iter=100, \
                shuffle=True, random_state=None,\
                tol=0.0001, verbose=True, warm_start=True, \
                momentum=0.8, early_stopping=False,\
                validation_fraction=0.2, beta_1=0.9, beta_2=0.999, \
                epsilon=1e-08, n_iter_no_change=30)
```

solver: The solver for weight optimization. “Adam” is stochastic gradient-based optimizer by Kingma, Diederik, and Ba. Another option is ’sgd,’ which stands for stochastic gradient descent.

alpha: L2 penalty – i.e., penalty to apply to the weights to prevent them from getting too large.

Step 4: Compile the NN

```
nn = MLPClassifier(hidden_layer_sizes=(64, 12, ),
        activation='relu', solver='adam', alpha=0.00001, \
        batch_size='auto', learning_rate='adaptive', \
        learning_rate_init=0.0001, max_iter=100, \
                shuffle=True, random_state=None,\
                tol=0.0001, verbose=True, warm_start=True, \
                momentum=0.8, early_stopping=False,\
                validation_fraction=0.2, beta_1=0.9, beta_2=0.999, \
                epsilon=1e-08, n_iter_no_change=30)
```

batch_size: Size of batch (in # of samples) after which the weights are updated. ‘auto’ is min(200, n_samples).

learning_rate: Controls the step size for updating the weights. “Adaptive” means keep learning rate constant as long as training loss keeps improving. If 2 epochs don’t decrease loss by tol, then current learning rate divided by 5.

Step 4: Compile the NN

```
nn = MLPClassifier(hidden_layer_sizes=(64, 12, ),
        activation='relu', solver='adam', alpha=0.00001, \
        batch_size='auto', learning_rate='adaptive', \
        learning_rate_init=0.0001, max_iter=100, \
                shuffle=True, random_state=None,\
                tol=0.0001, verbose=True, warm_start=True, \
                momentum=0.8, early_stopping=False,\
                validation_fraction=0.2, beta_1=0.9, beta_2=0.999, \
                epsilon=1e-08, n_iter_no_change=30)
```

Learning_rate_init: Initial learning rate.

max_iter: Maximum number of passes through the dataset

shuffle: Whether to shuffle samples in each iteration

random_state: Seed used to initialize the weights. If None, uses output from np.random.

Step 4: Compile the NN

```
nn = MLPClassifier(hidden_layer_sizes=(64, 12, ),
        activation='relu', solver='adam', alpha=0.00001, \
        batch_size='auto', learning_rate='adaptive', \
        learning_rate_init=0.0001, max_iter=100, \
                shuffle=True, random_state=None,\
                tol=0.0001, verbose=True, warm_start=True, \
                momentum=0.8, early_stopping=False,\
                validation_fraction=0.2, beta_1=0.9, beta_2=0.999, \
                epsilon=1e-08, n_iter_no_change=30)
```

tol: Tolerance for the optimization. When the loss does not at least improve by tol for n_iter_no_change, training stops (convergence assumed).

verbose: Whether to print out messages during training.

warm_start: If true, use the solution from previous call to fit as initialization

Step 4: Compile the NN

```
nn = MLPClassifier(hidden_layer_sizes=(64, 12, ),
        activation='relu', solver='adam', alpha=0.00001, \
        batch_size='auto', learning_rate='adaptive', \
        learning_rate_init=0.0001, max_iter=100, \
                shuffle=True, random_state=None,\
                tol=0.0001, verbose=True, warm_start=True, \
                momentum=0.8, early_stopping=False,\
                validation_fraction=0.2, beta_1=0.9, beta_2=0.999, \
                epsilon=1e-08, n_iter_no_change=30)
```

momentum: Momentum for gradient descent update. Between 0 and 1. Only used when solver = ‘sgd’.

early_stopping: Whether to stop early if validation score is not improving. If true, withholds validation_fraction of the training samples for the validation set. Terminates when validation score doesn’t improve by tol for n_iter_no_change.

Step 4: Compile the NN

```
nn = MLPClassifier(hidden_layer_sizes=(64, 12, ),
        activation='relu', solver='adam', alpha=0.00001, \
        batch_size='auto', learning_rate='adaptive', \
        learning_rate_init=0.0001, max_iter=100, \
                shuffle=True, random_state=None,\
                tol=0.0001, verbose=True, warm_start=True, \
                momentum=0.8, early_stopping=False,\
                validation_fraction=0.2, beta_1=0.9, beta_2=0.999, \
                epsilon=1e-08, n_iter_no_change=30)
```

beta_2: Exponential decay rate for estimates of second moment vector in adam. Only used when solver='adam'.

epsilon: Value for numerical stability in adam. Only when solver='adam'

n_iter_no_change: Max number of epochs to meet tol improvement.

Step 4: Compile the NN

```
nn = MLPClassifier(hidden_layer_sizes=(64, 12, ),
        activation='relu', solver='adam', alpha=0.00001, \
        batch_size='auto', learning_rate='adaptive', \
        learning_rate_init=0.0001, max_iter=100, \
                shuffle=True, random_state=None,\
                tol=0.0001, verbose=True, warm_start=True, \
                momentum=0.8, early_stopping=False,\
                validation_fraction=0.2, beta_1=0.9, beta_2=0.999, \
                epsilon=1e-08, n_iter_no_change=30)
```

validation_fraction: Fraction of the training set to use as the validation set. Only used if early_stopping = True.

Beta_1: Exponential decay rate for estimates of first moment vector in adam. Only used when solver='adam'.

Step 5: Train the NN

1. Call the fit function on the nn object:
   nn.fit(training_x, training_y)

Step 6 (Optional): Save trained NN with pickle

Pickle can save the trained NN object to a file, which can then be used to make predictions in other scripts.

1. Make sure pickle is imported:
   import pickle at the top of the script.
2. Execute the following lines of code:

```
nn_pkl = open(clf_filename, 'wb')
pickle.dump(nn,nn_pkl)
nn_pkl.close()
```

Step 7: Use the trained NN to make predictions on the test data.

1. Probabilistic predictions can be made by calling the *predict_proba* function on the RF object.

   clf_probs = nn.predict_proba(testing_x)[:,1]

2. Output probabilities will be in a 1-d array, so we have to do some accounting to get individual-day probs. Here, that's done with the *np.array_split* function and the *save_probs* method. As a result, individual-day hail probabilities are saved in .txt files to be read by another script for evaluation.

To execute steps 3-7:

1. First, create a directory to hold the resulting prediction text files:

   mkdir outfiles – NOTE: We should have already done this in step 1.

2. Run the *run_rf.py* script by submitting a batch job.

   qsub train_nn_qsub

Step 8: Evaluate the skill of the RF probabilities using the test_y data

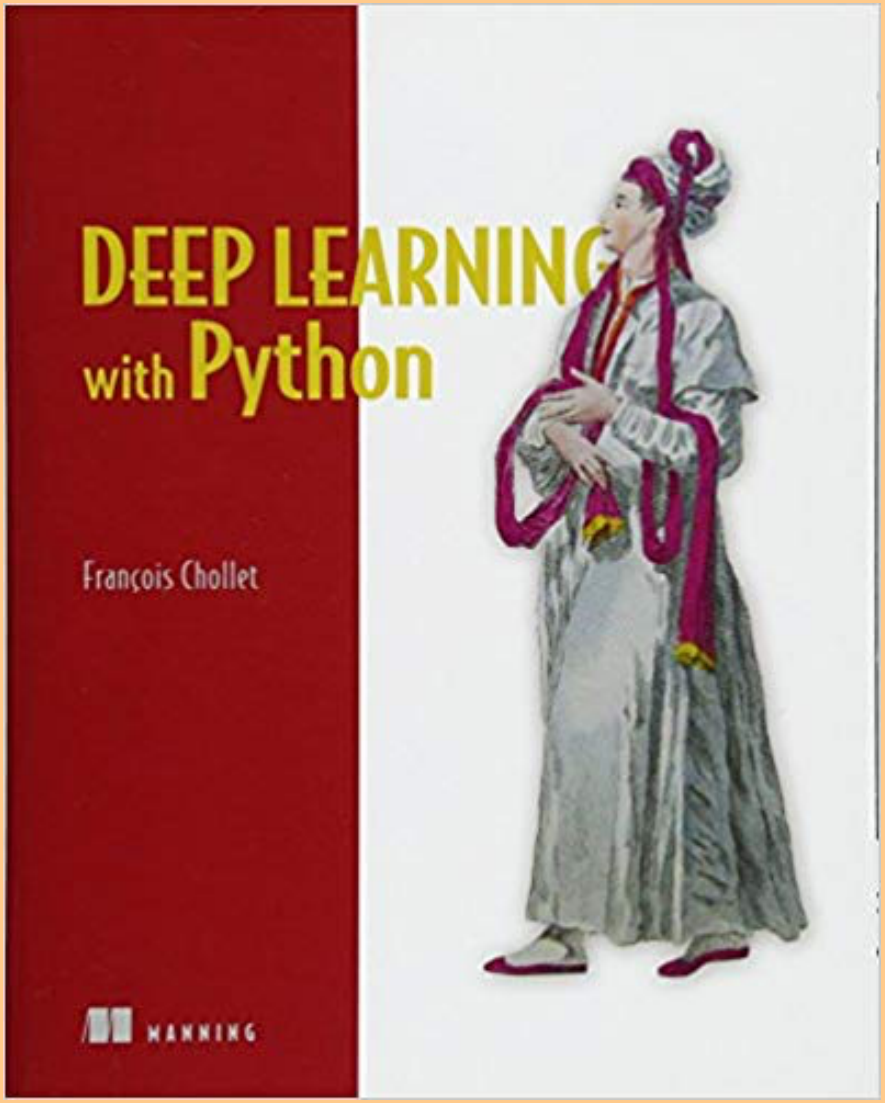
1. To obtain overall skill metrics, attributes, performance, and ROC diagrams, execute:

   python3 skill_scores_evaluate_nn.py

2. To visualize what these forecasts look like on individual days, execute:

   python3 evaluate_nn.py

Task 3: Use a CNN to make predictions using Keras



Highly recommended text for deep learning.

Buy online for ~$15-$35

Step 1: Download relevant files

1. Change into your nn directory

   cd ../dl

2. Download dl files

   cp /glade/work/eloken/ML_tutorial_final/files/keras/dl_files.tar.gz .

3. Untar the files.

   tar -xzvf dl_files.tar.gz

4. While we're here, let's also make the directory to hold future output files:

   mkdir outfiles

Step 2: Preprocess the data

1. Preprocessing for a CNN with Keras is a little bit different than before. The files have to be in the following format: (samples, height, width, channels). Where height and width are the height/width of the patch, and number of channels corresponds to the number of training variable "fields" (e.g., CAPE, CIN, etc.). Here, we will be looking at 5x5 image patches (i.e., height = width = 5) and 12 variables (i.e., channels = 12). There will be 1434 samples per day (the number of points in the domain).

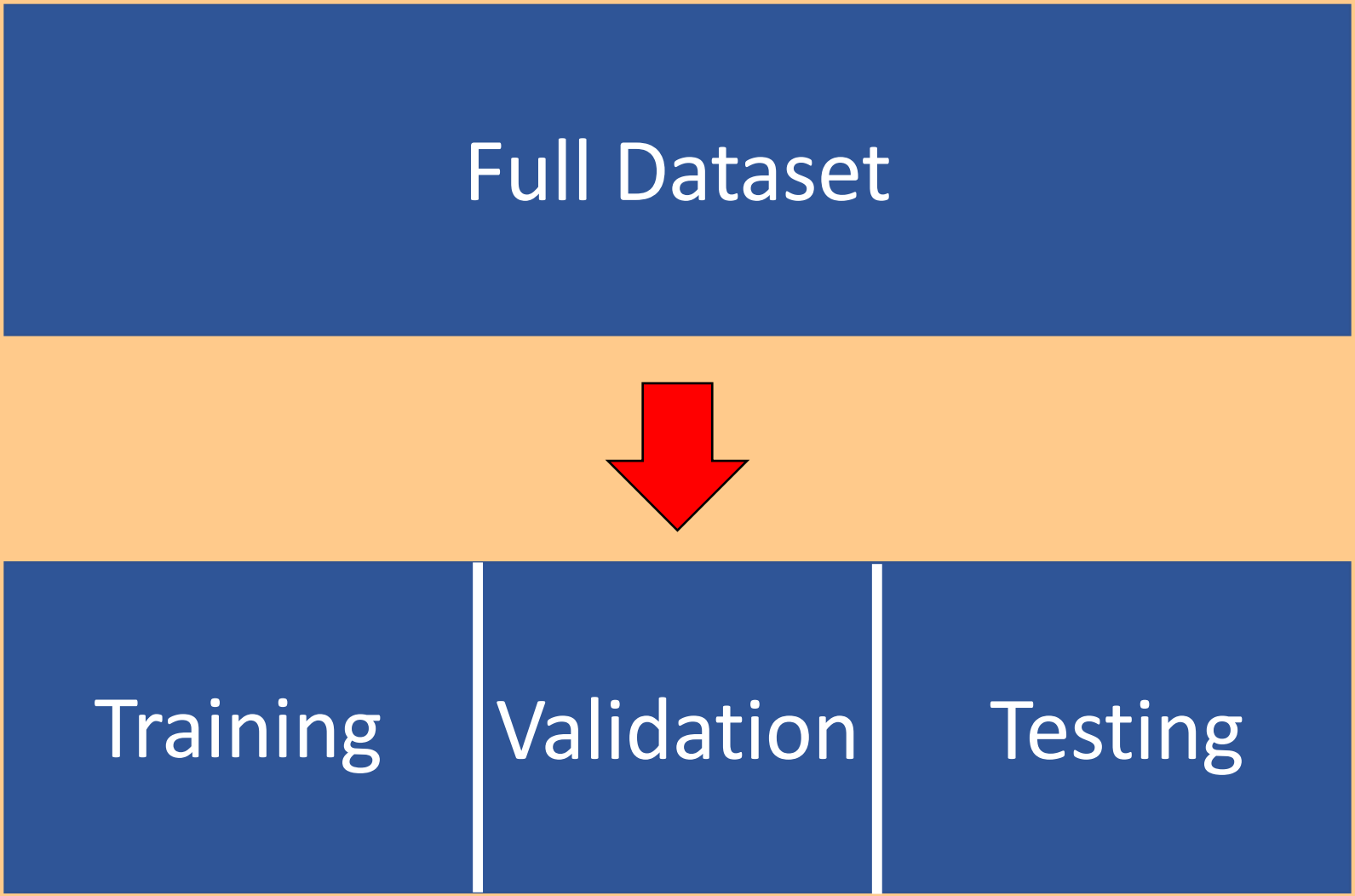
2. The *preprocess_keras.py* script will handle the preprocessing. First, we need to make directories where the preprocessed files will be output.

    mkdir fcst obs

3. Now, run the preprocessing script.

    python3 preprocess_keras.py

Step 3: Divide dataset into separate training/testing/validation sets



Step 3: Divide dataset into separate training/testing/validation sets

1. This is accomplished by *run_dl.py* reading in separate dates associated with the training, testing, and validation sets.
   - train_dates.txt: 20150421 – 20160428
   - val_dates.txt: 20160429 – 20160630
   - test_dates.txt: 20170402 – 20170624

Step 4: Normalize the data

1. Again, we want to subtract the training mean and divide by the training standard deviation (for all variables/channels):

```
train_mean = training_x.mean(axis=0)
train_sd = training_x.std(axis=0)
delta = 0.000001
train_sd += delta
np.save("train_mean.npy", train_mean)
np.save("train_std.npy", train_sd)
training_x = (training_x - train_mean)/train_sd
testing_x = (testing_x - train_mean)/train_sd
val_x = (val_x - train_mean)/train_sd
```

Step 5: Write the DL model

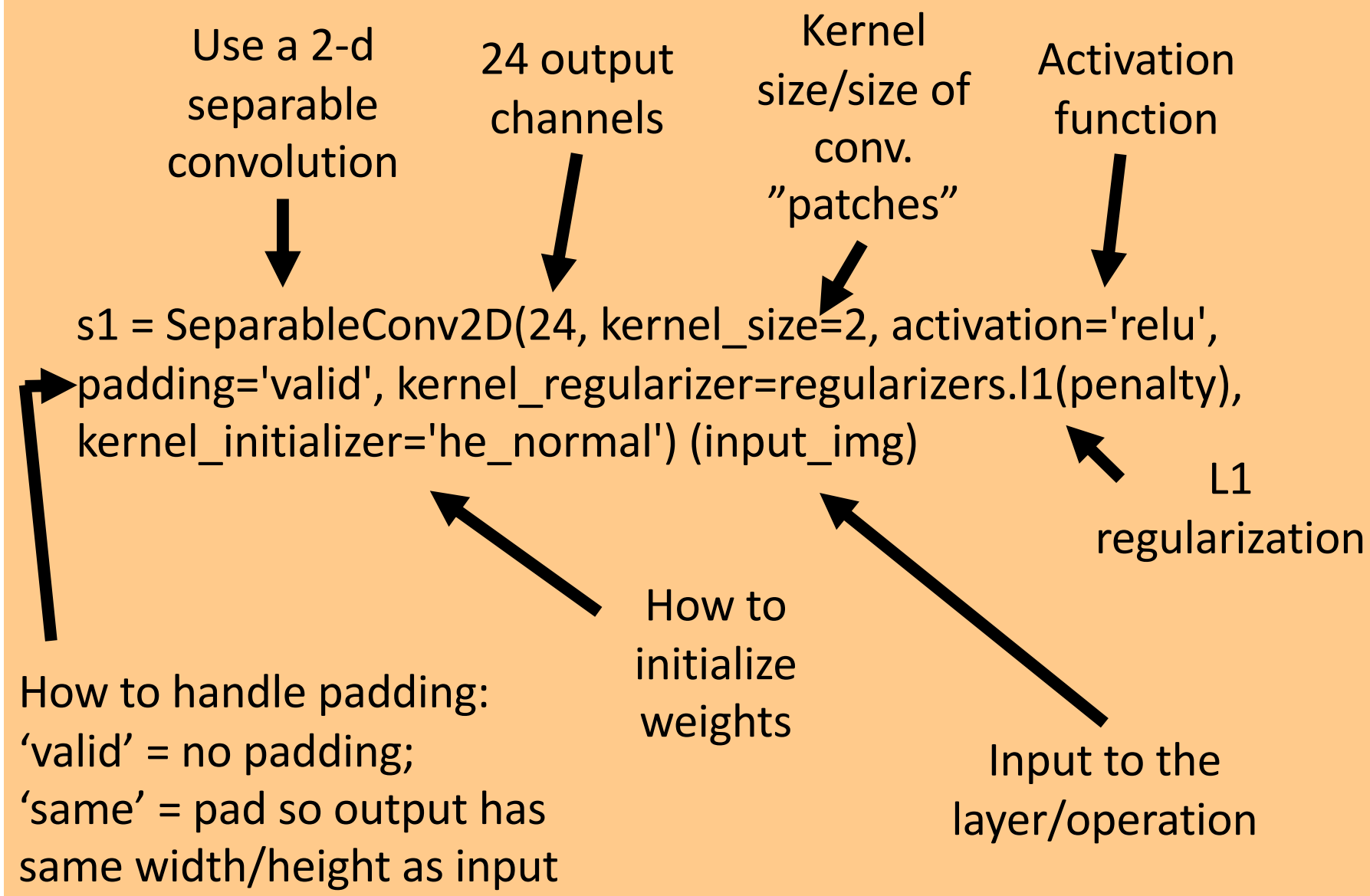
1. We will make a relatively simple residual NN to illustrate some key principles of DL. This is done in *separable_conv_severe_5x5.py*. We'll break down what's going on in the script.

Step 5: separable_conv_severe_5x5.py

def **conv_net**(input_img, drop_rate, penalty):

Input parameters: the input image, the dropout rate fraction, and the L1 regularization penalty applied to each weight (goal is to prevent weights from getting too large).

Step 5: separable_conv_severe_5x5.py



Step 5: separable_conv_severe_5x5.py

s1 = BatchNormalization() (s1) #Batch normalization performed on s1
s1 = MaxPooling2D((2,2)) (s1) #Max pooling performed on s1
s1 = Dropout(drop_rate) (s1) #Dropout performed on s1

Step 5: separable_conv_severe_5x5.py

```
s2 = SeparableConv2D(48, kernel_size=2, activation='relu',
padding='valid', kernel_regularizer=regularizers.l1(penalty),
kernel_initializer='he_normal') (s1)

s2 = BatchNormalization() (s2)

s2 = Dropout(drop_rate) (s2)
```

Step 5: separable_conv_severe_5x5.py

```
#First residual layer

#Note that this Separable Conv2D layer uses input from the input
image.
r1 = SeparableConv2D(48, kernel_size=3, activation='relu',
padding='valid', strides=1, kernel_regularizer=regularizers.l1(penalty),
kernel_initializer='he_normal') (input_img)

r1 = BatchNormalization() (r1)

r1 = AveragePooling2D((2,2)) (r1)

r1 = Dropout(drop_rate) (r1)
```

Step 5: separable_conv_severe_5x5.py

```
#Add the residual layer and main branch
p1 = layers.add([s2, r1])
```

Step 5: separable_conv_severe_5x5.py

```
#Add the residual layer and main branch
p1 = layers.add([s2, r1])

#Now, flatten and use to make predictions
 f1 = Flatten() (p1)
 f2 = Dense(36, activation='relu') (f1)
 f2 = BatchNormalization() (f2)
 f2 = Dropout(drop_rate) (f2)

#Second fully connected layer
f3 = Dense(12, activation='relu') (f2)
f3 = BatchNormalization() (f3)
f3 = Dropout(drop_rate) (f3)
```

Step 5: separable_conv_severe_5x5.py

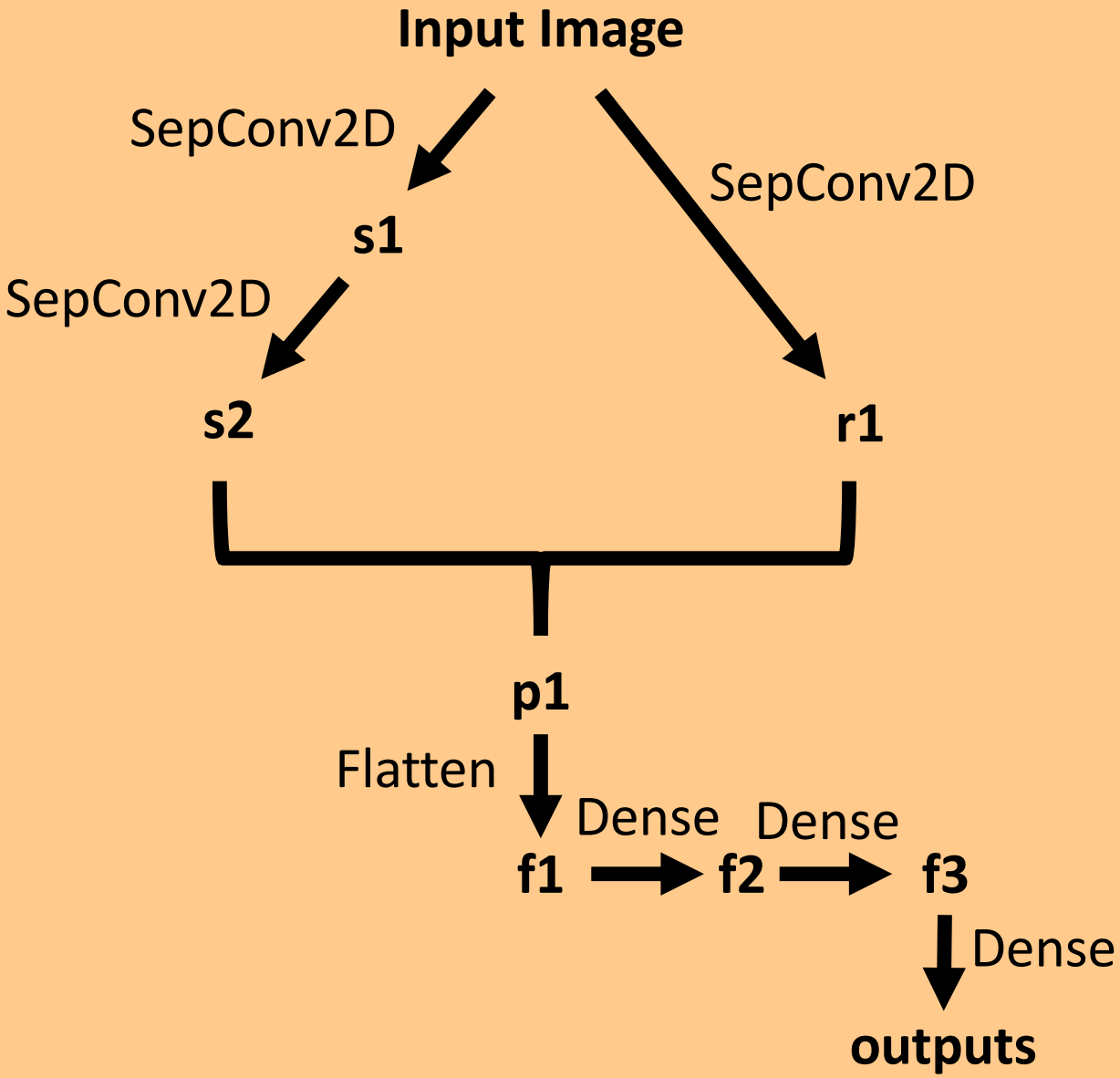
```
#Now get output layer
outputs = Dense(1, activation='sigmoid') (f3)

model = Model(inputs=[input_img], outputs=[outputs])

print (model.summary())

return model
```

Step 5: separable_conv_severe_5x5.py



Step 6: Compile the model

```
input_image = Input((patch_size, patch_size, n_channels))

conv_model = conv_net(input_image, dropout_rate, reg_penalty)

loss='binary_crossentropy', metrics=['accuracy'])

conv_model.compile(optimizer=Adam(), loss='binary_crossentropy',
metrics=['accuracy'])

history = AccuracyHistory()

metrics = Metrics()

callbacks_list = [ EarlyStopping(monitor='val_loss', patience=2,),
ModelCheckpoint(filepath=model_outname, monitor='val_loss',
save_best_only=True,), history, metrics]
```

Step 7: Train the model

```
conv_model.fit(training_x, training_y, batch_size=batch_size,
        epochs=n_epochs, verbose=1, validation_data=(val_x,
        val_y), callbacks=callbacks_list)


score = conv_model.evaluate(testing_x, testing_y, verbose=0)
```

To execute steps 3-7:

1. Run the *run_dl.py* script by submitting a batch job.
   sbatch casper_sbatch

2. Note that a trained model is produced:
separable_conv_tutorial_5x5.h5
We will test this model in a subsequent step.

Step 8: Use the trained model to make predictions on the test data.

1. We will do this step in a separate script using the trained model. Run *get_output_dl.py*. Works by reading in the trained model, appropriately normalizing the test data (based on training normalization) and using the trained model to make predictions on the test data.

   python3 get_output_dl.py

Step 8: Evaluate the skill of the DL probabilities using the test_y data

1. To obtain overall skill metrics, attributes, performance, and ROC diagrams, execute:

   python3 skill_scores_evaluate_dl.py

2. To visualize what these forecasts look like on individual days, execute:

   python3 evaluate_dl.py