AI for Shrubland Identification and Mapping

Michael J Mahoney[1,*] Lucas K Johnson[1] Colin M Beier[2]

[1] Graduate Program in Environmental Science, State University of New York College of Environmental Science and Forestry, 1 Forestry Drive, Syracuse, NY, 13210
[2] Department of Sustainable Resources Management, State University of New York College of Environmental Science and Forestry, 1 Forestry Drive, Syracuse, NY, 13210

[*] Correspondence: Michael J Mahoney <mjmahone@esf.edu>

1 Introduction

This chapter walks through a procedure for predicting the prevalence of "shrubland" (defined here as low-statured vegetation between 1 and 5 meters in height) across a diverse region in New York State, patterned off the process used in Mahoney, Johnson, and Beier (2022a). Due to the impacts of climate change and human land use patterns, these shrublands are becoming an increasingly important land cover type in the region, often representing an entirely novel ecosystem type. As a result of this novelty, these shrublands and the roles they play in the larger landscape (for instance, as habitats and as components of biogeochemical cycles) are poorly understood. Even identifying shrublands using remote sensing data, a potential way to monitor their development over time, is difficult given the relative rarity of shrublands in this region and their similar appearance to forest lands and wetlands in satellite imagery.

The chapter introduces step-by-step how to fit a feedforward neural network using the Keras module in the popular TensorFlow package and a subset of the data from Mahoney, Johnson, and Beier (2022a). Due to the rarity of shrubland in this region of New York, the chapter focuses on the adjustments necessary when building models from data with imbalanced classes, and on how to interpret model performance metrics when fitting classification models for specific purposes. Chapter exercises prompt learners to investigate how different priorities for a model might result in notably different performance measures.

2 What You'll Learn

- Ways to think about "model performance" in the context of machine-learning shrubland classification models
- Creating a model for imbalanced classes
- Fitting a simple feedforward neural network with Keras and TensorFlow
- Rasterizing model predictions for visualization

3 Background

Human land use has fundamentally reshaped the structure and composition of the surrounding environment, leaving lasting legacies including the emergence of novel communities and ecosystem types (Foster, Motzkin, and Slater 1998; Cramer, Hobbs, and Standish 2008). Among the outcomes of these changes, the emergence of low-statured vegetation or 'shrublands' as a more common cover type in the US Northeast has been suggested by numerous field studies, but is poorly understood from a landscape perspective. Although long disregarded, these lands are rapidly gaining attention in today's urgent push to implement 'natural climate solutions' (Fargione et al. 2018) and identify 'marginal' or 'underutilized' lands for renewable energy generation.

However, current limitations to the classification and mapping of these cover types pose obstacles to advancing both science and stewardship opportunities (Hobbs, Higgs, and Harris 2009). Shrublands are a very challenging cover class to identify from imagery alone, given the breadth of community types included and the high variability in density and canopy cover that exists within and among those community types (King and Schlossberg 2014). In practical terms this means that, when relying solely on imagery, shrublands encompass a full gradient from resembling herbaceous or barren land to resembling closed-canopy conditions (Brown et al. 2020). As a result, satellite or aerial imagery-based approaches tend to classify shrubland categories with substantially lower accuracy than other land use and land cover (LULC) classes (Wickham et al. 2021; Brown et al. 2020).

A solution for this problem might be to incorporate additional, non-imagery sources of remote sensing data into LULC classification methodologies. LiDAR data collected through airborne laser scanning can provide essential information for identifying low-statured vegetation such as early-successional forests (Falkowski et al. 2009). In combination with imagery, LiDAR data can enable continuous, broad-scale estimation of canopy heights and other structural traits which greatly simplifies the task of distinguishing between low-statured and taller closed-canopy cover types (Ruiz et al. 2018). Unfortunately, the cost and logistical challenges of airborne LiDAR collection have constrained its availability to smaller extents and with much longer return intervals than provided by satellite imagery. Yet if canopy structural estimates from airborne LiDAR could be used to label a training dataset in order to fit models using satellite imagery, it should be possible to produce models capable of identifying shrubland

with greater accuracy than those trained on imagery alone, while being able to map/model a larger and more contiguous spatial extent than models relying on airborne LiDAR data as predictors.

As such, we undertook a project aiming to use an AI/ML based approach to identify probable 'shrubland' areas across New York State, USA, using predictors derived from optical imagery classified as 'shrubland' using available aerial LiDAR data. Full details on this project are available in Mahoney, Johnson, and Beier (2022a). This chapter uses a subset of the data used in Mahoney, Johnson, and Beier (2022a) to walk through our modeling approach and discuss the specific concerns associated with attempting to model a relatively rare land cover class across large regions.

4 Prerequisites

This chapter was created using Python version 3.8.13 (Python Core Team 2022), TensorFlow version 2.9.1 (Abadi et al. 2015; Chollet 2015), scikit-learn version 1.1.2 (Pedregosa et al. 2011), pandas version 1.4.3 (team 2020; McKinney 2010), numpy version 1.23.1 (Harris et al. 2020), pydot 1.4.2 (Carrera, Nowee, and Kalinowski 2021), matplotlib 3.5.3 (Hunter 2007), and rasterio 1.3.0 (Gillies et al. 2013). While the code in this chapter may work with other versions, it has not been tested with other configurations, and the code may produce different results.

All of the required libraries can be installed using the command:

```
pip install \
  tensorflow==2.9.1 \
  scikit-learn==1.1.2 \
  numpy==1.23.1 \
  pandas==1.4.3 \
  pydot==1.4.2 \
  matplotlib==3.5.3 \
  rasterio==1.3.0
```

This command will install the main libraries we'll be relying upon, alongside all of the other libraries these need in order to work properly. Because these "dependencies" are installed automatically, this command also installs all of the other libraries we'll be using throughout this chapter.

We'll be working with a subset of the data used in the original study, published on Zenodo (Mahoney, Johnson, and Beier 2022b). The following code can be used to download the data and unpack it in the current working directory. Note that the data is approximately 1.3 gigabytes, and as such can take a while to download over slow connections.

```
import urllib.request
from zipfile import ZipFile

urllib.request.urlretrieve(
    "https://zenodo.org/record/6824173/files/data.zip?download=1",
    "data.zip",
)

ZipFile("data.zip", "r").extractall(".")
```

This directory contains a file, `3_county_2014.csv`, which contains all the data we'll use to fit models in this chapter. Each row in this CSV represents a 30-meter square "pixel" of land in New York's lower Hudson River valley (Figure 1). This study area includes Duchess, Orange and Ulster counties, and has a wide variety of land cover types ranging from highly urbanized areas along the Hudson River to the highly forested, largely protected Catskill Mountains in the western part of the study area. The data only reflects areas that are classified as vegetation based on the US Geological Survey's Land Change Monitoring, Assessment, and Projection (LCMAP) data set's primary land cover classification (Brown et al. 2020). As a result, most bodies of water and urban areas are excluded from the data.

In order to focus primarily on the modeling process, we'll be skipping most of the work involved in collecting and processing data. Instead, we'll use a set of predictors pre-calculated from Landsat imagery collected between July 1st 2014 and September 1st 2014 (Table 1). These predictors were adjusted using the Landtrendr algorithm in order to fill in gaps from clouds and shadows and remove noise from each pixel (Kennedy, Yang, and Cohen 2010; Kennedy et al. 2018). More detail on the data retrieval and preprocessing procedures can be found in Mahoney, Johnson, and Beier (2022a).

Table 1: Definitions of predictors used for model fitting.

Raster Band Name	Definition
TCB, TCW, TCG	Tassled cap brightness (TCB), wetness (TCW), and greenness (TCG), with noise removed using Landtrendr
NBR	Normalized burn ratio (NBR) with noise removed using Landtrendr
MAG, YOD	Magnitude (MAG) and year of most recent disturbance (YOD), as identified using Landtrendr
PRECIP, TMAX, TMIN	30-year normals for precipitation (PRECIP), maximum temperature (TMAX), and minimum temperature (TMIN), derived from annual PRISM climate models

Raster Band Name	Definition
ASPECT, DEM, SLOPE, TWI	Aspect, elevation (DEM), slope, and topographic wetness index (TWI) derived from a 30-meter digital elevation model
LCSEC	LCMAP secondary land cover classification

These predictors are also included as a TIFF file, `3_county_2014.tiff`, projected using the PROJ string:

```
+proj=aea +lat_0=23 +lon_0=-96 +lat_1=29.5 \
+lat_2=45.5 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs
```

You can open this TIFF file in any GIS software in order to see the actual distribution of each predictor throughout the study area.

5 Model Building

5.1 Preprocessing

With our libraries installed and our data downloaded, we're ready to begin! First things first, let's load all the libraries we'll be using:

```
# We'll be working with our data primarily as pandas DataFrames
# and converting them to numpy arrays as necessary:
import numpy as np
import pandas as pd

# We'll set a random number seed
# to ensure reproducibility across notebook runs.
#
# First, set the environment variable 'PYTHONHASHSEED' to 0:
import os
os.environ['PYTHONHASHSEED']=str(0)

# Now, set the random seeds from the `random` and `numpy` packages:
import random
random.seed(123)
np.random.seed(123)
```

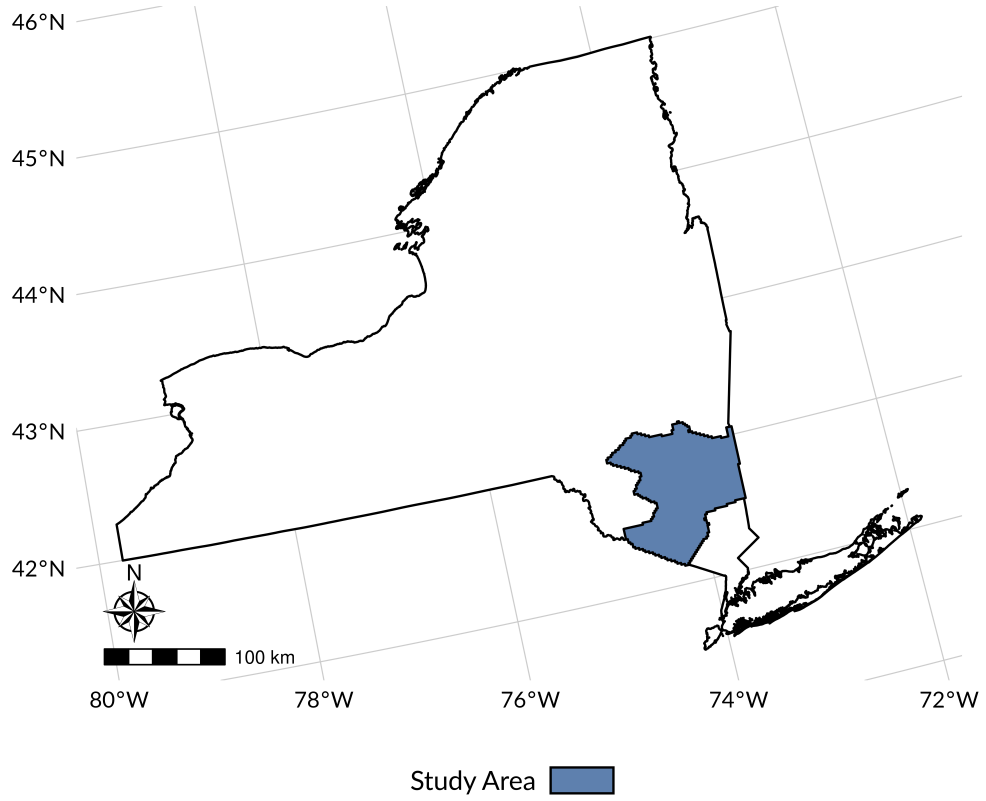

Figure 1: A map showing the location of the study area (filled blue polygon) within New York State.

```
# We'll use scikit-learn for normalizing our data,
# and for splitting our data into training-validation-testing sets:
import sklearn
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# We'll use Keras, installed as part of Tensorflow, for model fitting:
import tensorflow as tf
from tensorflow import keras

# Set one more random seed value, this time from Tensorflow itself:
tf.random.set_seed(123)

# Initialize our GPUs to use memory growth, in order to work better
# when multiple jobs are using the same GPU
gpu_devices = tf.config.experimental.list_physical_devices("GPU")
for device in gpu_devices:
    tf.config.experimental.set_memory_growth(device, True)
```

With our environment all ready, we can go ahead and start preprocessing our data into an AI-ready format. Our data has been filtered to only include areas that LCMAP (Brown et al. 2020) has identified as being vegetated. As a result, all of our data represents areas that were classified by LCMAP as being either agricultural, forestland, herbaceous/grassland, or wetland areas.

Let's load that data into our session now, and then pop the label column (named `shrub`) out as its own object:

```
full_data = pd.read_csv("data/3_county.csv")
full_labels = full_data.pop("shrub")
```

If all has gone as planned, `full_data` should now be a data frame with roughly 7 million observations of 17 separate features. These features are primarily variables derived from Landsat imagery, but also include the X and Y coordinate positions of each pixel of the raster, and the LCMAP secondary land cover classification (stored as `lcsec_2014`; Table 1):

```
print(full_data.iloc[:, 0:6].head(n=5))
```

```
         x        y  lcsec_2014  tcb_2014  tcw_2014  tcg_2014
0  1788750  2339280           4      2995      -638      2642
1  1788780  2339280           3      2759      -487      2437
2  1788480  2339250           1      2655      -634      2259
```

```
3   1788510   2339250            4       2908       -720       2253
4   1788720   2339250            4       3153       -698       2726
```

In the `3_county` file, that 'LCSEC' feature is stored as a single categorical variable, meaning the variable's value can only be chosen from a group of candidate values, with each different value indicating a different land cover class. However, ML models cannot directly digest categorical variables as ML models tend to output continuous values. In order for ML models to make use of this information, we need to encode this single categorical variable into a number of boolean indicator variables, transposing our single categorical variable into a set of booleans. We can do this using the `get_dummies()` function from pandas:

```
full_data = pd.concat(
    [
        full_data,
        pd.get_dummies(full_data["lcsec_2014"], prefix="lcsec", drop_first=True),
    ],
    axis=1,
)
print(full_data.iloc[:, 16:23].head(n=5))
```

```
   lcsec_2  lcsec_3  lcsec_4  lcsec_5  lcsec_6  lcsec_8
0        0        0        1        0        0        0
1        0        1        0        0        0        0
2        0        0        0        0        0        0
3        0        0        1        0        0        0
4        0        0        1        0        0        0
```

With the 'LCSEC' categorical variable encoded, the next step is to drop the variables we don't intend to use in our model. Specifically, we're going to drop the `lcsec_2014` column, as it's now encoded as a number of boolean variables. We'll also be dropping the `x` and `y` coordinate variables:

```
full_features = full_data.drop(["lcsec_2014", "x", "y"], axis=1)

print(full_features.iloc[:, 0:6].head(n=5))
```

```
   tcb_2014  tcw_2014  tcg_2014  nbr_2014  mag_2014  yod_2014
0      2995      -638      2642       737         0         0
1      2759      -487      2437       750         0         0
2      2655      -634      2259       708         0         0
```

```
3       2908       -720       2253       678          0          0
4       3153       -698       2726       742          0          0
```

Now that we have the full set of variables we intend to use to fit our ML models, it's time to split our data to create a "hold-out" test set that we'll use to assess our final model. Normally 20% of the data will be allocated to the test set, to make sure that there are enough observations to assess our final models while also leaving enough data behind to train our neural net. We'll use the `train_test_split()` function from scikit-learn to create these splits:

```
[train_features, test_features, train_labels, test_labels] = train_test_split(
    full_features, full_labels, test_size=0.2, random_state=123, stratify=full_labels
)
```

That test set will be used to calculate the our final performance metrics after our ML model training finished. We'll still want to assess all the intermediate models produced during model training! To do so, we'll need to split our data one more time to create a "validation set", which will be used to evaluate models and get out-of-sample performance estimates before we're ready to use our final test set. Just like before, we'll use `train_test_split()` to take 20% of the remaining training set to produce our validation set:

```
[
    train_features,
    validation_features,
    train_labels,
    validation_labels,
] = train_test_split(
    train_features, train_labels, test_size=0.2, random_state=123, stratify=train_labels
)
```

Now that we've split our data into training, validation, and testing sets, it's time for the last bit of preprocessing before we start fitting our models. First, we'll need to convert our data into numpy arrays, a data format that Keras will automatically understand and work with:

```
train_features = np.array(train_features)
validation_features = np.array(validation_features)
test_features = np.array(test_features)
```

Secondly, we'll need to standardize our data so that all of the input variables have zero mean and unit variance. To do this, we'll use the `StandardScaler()` function from scikit-learn. We'll initialize the rescaler on our training data, to calculate the mean and standard deviation of each feature using the training data alone:

```
scaler = StandardScaler()
train_features = scaler.fit_transform(train_features)
```

And then we'll use the same rescaler to transform our validation and test data. It's very important to make sure you're not including your evaluation data when fitting the rescaler, as this is a form of "data leakage" which makes your final model evaluation not truly independent from the model fitting process, meaning your reported accuracy might be too optimistic when compared to the model's real-world performance.

```
validation_features = scaler.transform(validation_features)
test_features = scaler.transform(test_features)
```

We're also going to go ahead and transform our numpy arrays into TensorFlow datasets, which will make fitting and evaluating our models more efficient. By batching our data and setting it to prefetch future batches, we'll speed up the modeling process:

```
train_ds = (
    tf.data.Dataset.from_tensor_slices((train_features, train_labels))
    .batch(64)
    .prefetch(2)
)

val_ds = (
    tf.data.Dataset.from_tensor_slices((validation_features, validation_labels))
    .batch(64)
    .prefetch(2)
)

test_ds = (
    tf.data.Dataset.from_tensor_slices((test_features, test_labels))
    .batch(64)
    .prefetch(2)
)
```

The final preparation step is going to be determining class weights for tuning our model. Our data is extremely imbalanced, because most of New York State is not shrubland – our "positive" shrubland class is much, much smaller than the "negative" not-shrubland class:

```
neg, pos = np.bincount(train_labels)
total = neg + pos
print(
    "Examples:\n    Total: {}\n    Positive: {} ({:.2f}% of total)\n".format(
```

```
        total, pos, 100 * pos / total
    )
)
```

```
Examples:
    Total: 4585730
    Positive: 70213 (1.53% of total)
```

Because shrubland only constitutes about 1.5% of all observations, our model could achieve 98.5% accuracy by never predicting shrubland! As such, we need to do something to make our model "care" more about our positive shrubland class, in order to make sure the model is attempting to predict both classes.

There are several different approaches we could take to balance our classes, including resampling or downsampling so that our training data had the same number of observations in each class. However, when possible it's often more efficient to set the weights of each class in your model, to make the model "care" more about getting the right answer on your less prevalent class. Here, we'll calculate how much we need to adjust the class weights so that the positive shrubland class is as important as the negative not-shrubland class in the ML model's decision making:

```
class_weight = {
    0: (1 / neg) * (total / 2.0),
    1: (1 / pos) * (total / 2.0)
}
class_weight
```

```
{0: 0.5077746357726036, 1: 32.65584720778204}
```

Notice that we calculated these class weights using only our training data labels. Just as with rescaling your data, including your evaluation data when calculating class weights can be a form of data leakage which makes your evaluation data non-independent and your reported accuracy metrics too optimistic.

5.2 Model Fitting

With the training data rescaled and class weights calculated, we're ready to get into the modeling process. Simple models can give stable decent performance at lower cost, while the fancy more cutting-edge models can do better in accuracy but have a toll on costs and complexity. We'll be fitting a relatively straightforward feedforward neural net, using the

Keras module of the TensorFlow library (Chollet 2015; Abadi et al. 2015; LeCun, Bengio, and Hinton 2015).

In order to assess how well our models perform, we'll need to calculate several different metrics. Classification models can be tricky to assess, however, because your priorities and targets for a model determine what makes a model "better". Even the metrics you use to assess your model may vary as a result of your modeling goals. For instance, models of rare but highly-important events – such as models for detecting credit card fraud, or screening for diseases – might prioritize catching as many "positive" cases as possible, even if that means increasing the number of "false positive" classifications from the model. Other models, however, might have the exact opposite preference; for instance, a model predicting what stocks might be good investment decisions might prefer to only predict "sure bets", and would be willing to produce more false negatives in order to avoid spending money on bad investments.

In our situation, where we're modeling a relatively rare land cover classification, we're willing to accept some false positives to make sure that we're capturing as much shrubland as possible. At the same time, we want to make sure that our shrubland predictions are as precise as possible, so that when our model classifies a pixel as "shrubland" we can be decently sure it is truly a shrubland pixel. As such, we'll calculate a number of different model metrics, but are going to focus in particular on precision and PRC, the area under the precision-recall curve, as the way we interpret and understand our results.

Let's go ahead and define all the metrics we want to calculate when evaluating our models:

```
metrics = [
    keras.metrics.TruePositives(name="True_Positives"),
    keras.metrics.FalsePositives(name="False_Positives"),
    keras.metrics.TrueNegatives(name="True_Negatives"),
    keras.metrics.FalseNegatives(name="False_Negatives"),
    keras.metrics.BinaryAccuracy(name="Binary_Accuracy"),
    keras.metrics.Precision(name="Precision"),
    keras.metrics.Recall(name="Recall"),
    keras.metrics.AUC(name="AUC"),
    keras.metrics.AUC(name="PRC", curve="PR"),
]
```

With our metrics defined, our next step is to create the actual model structure. For the purposes of this chapter, we'll use a straightforward feedforward neural network, with a total of six densely connected layers and a single dropout layer. We can define the model using Keras' sequential API like so:

```
def make_model(metrics=metrics):
    # Create a model object using the sequential API:
```

```
model = keras.Sequential(
    [
        # Add a dense layer, using:
        # + 256 neurons
        # + The rectified linear unit ("ReLU") activation function
        # + An input layer with the same number of neurons
        #   as predictors in our training data
        keras.layers.Dense(
            256, activation="relu", input_shape=(train_features.shape[-1],)
        ),
        # Add another dense layer, this time with 128 neurons:
        keras.layers.Dense(
            128, activation="relu"),
        # Another with 64:
        keras.layers.Dense(
            64, activation="relu"),
        # Another with 32:
        keras.layers.Dense(
            32, activation="relu"),
        # And another with 16:
        keras.layers.Dense(
            16, activation="relu"),
        # Add a dropout layer.
        # This will randomly set 20% of the inputs --
        # -- that is, the outputs from the last dense layer --
        # to 0, which helps protect against overfitting
        keras.layers.Dropout(0.2),
        # Finally, add a dense layer with a single neuron,
        # using the "sigmoid" activation function
        #
        # This will produce our final probability predictions
        keras.layers.Dense(1, activation="sigmoid"),
    ]
)
model.compile(
    # Use "Adaptive Moment Estimation" optimization to tune weights
    #
    # This algorithm will adjust the weights of each neuron in
    # the network every epoch, attempting to optimize the loss function
    # defined in the next argument
    #
```

```
        # While the details are somewhat complicated, for applied purposes
        # it's often practical to just use the Adam optimizer with a rather
        # small "learning rate" to achieve a decent model
        optimizer=keras.optimizers.Adam(learning_rate=1e-3),
        # Use the most standard loss for binary classification models,
        # binary cross-entropy, to judge how well our model is doing.
        #
        # Lower cross-entropy values are better.
        loss=keras.losses.BinaryCrossentropy(name="Binary_Cross_Entropy"),
        # In addition to calculating our cross-entropy loss at each step
        # and adjusting our model weights, we'll also ask Keras to calculate
        # the metrics we defined earlier for every epoch
        metrics=metrics,
    )
    return model
```

The bulk of this model is made up of "dense" layers, which are made up of some number of "neurons" (between 256 and 16). Each of those neurons takes the results from every neuron in the previous layer as input, and transforms them using a standard formula:

$$\text{input} \cdot \text{kernel} + \text{bias} \tag{1}$$

Where kernel is a weights matrix created by each layer, and bias is a bias vector created by each layer.

The output of this formula is then run through an "activation function" in order to get the final output from each neuron. In this case, almost all of our layers are using the "`relu`" activation function, which stands for "rectified linear unit". For a given value x, this function returns x if x is positive and 0 otherwise:

$$\max(0, x) \tag{2}$$

The outputs from this activation function are then provided as inputs to every neuron in the next layer of the neural net.

In addition to these densely connected layers, this neural net also has a dropout layer at the end of the net. This layer takes the inputs from all the neurons in the previous layer and randomly sets 20% of them to 0, reducing the model's ability to overfit on the training data.

Last but not least, the results from that dropout layer are passed as inputs to our final dense layer. Unlike the other dense layers, this layer – which we refer to as the "output" layer – is only going to generate a single result, which will be our predicted probability.

Because we want to predict probability, which ranges from 0 to 1, we want to make sure our output layer will only predict probabilities between 0 and 1. In order to do so, we use the "`sigmoid`" activation function, which transforms a given input x via the formula:

$$\frac{1}{1+e^{-x}} \tag{3}$$

This activation function will force our predictions to fall between 0 and 1 as desired. Because this layer only has a single neuron, it will generate a single output; this is how we'll generate predictions for each observation in our data and eventually for every pixel in our map.

All told, our model looks something like the schematic in Figure 2, with a single input layer, a number of densely connected "hidden" layers, a dropout layer, and finally the single neuron output layer which will generate our final predictions.

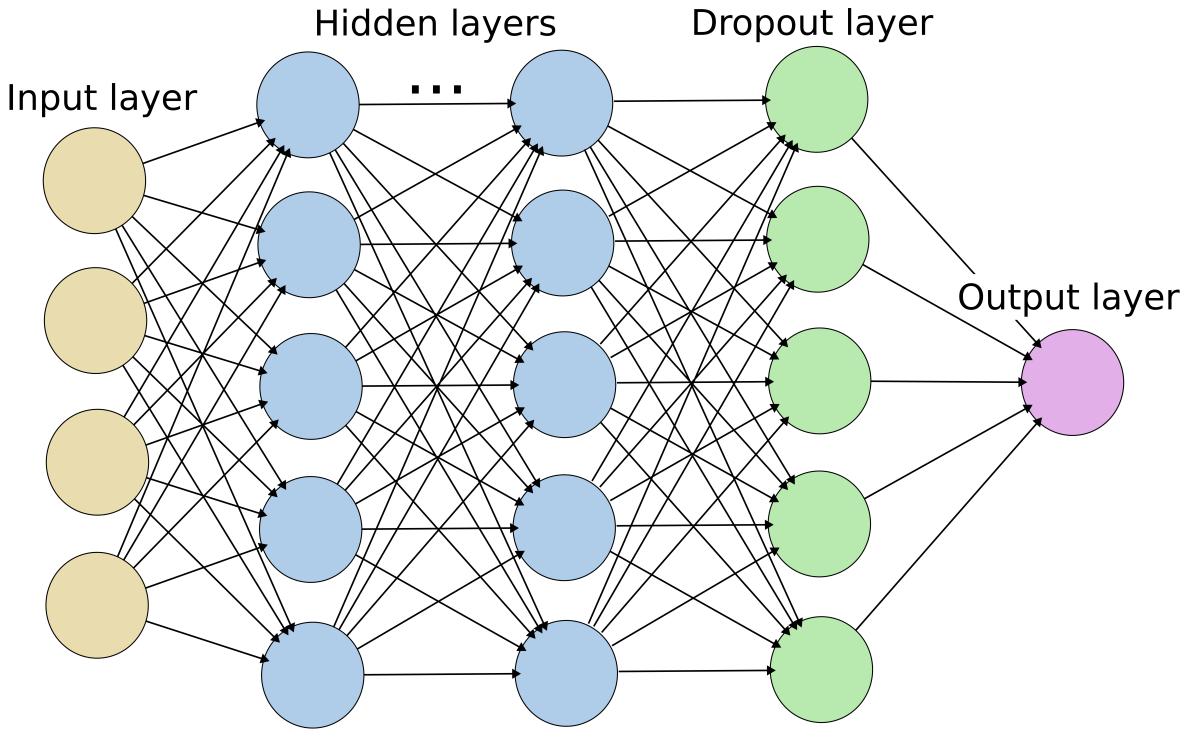

Figure 2: A diagram of a feedforward neural network using densely connected layers.

We can also visualize this specific model, using the `plot_model()` function from `keras.utils`. This function will give us the schematic in Figure 3, showing the type of layers we're using (either "InputLayer", "Dense", or "Dropout"), the activation function in use (either "relu"

or “sigmoid”), the number of inputs to each neuron in the layer, and the number of outputs generated by the layer.

```
shrubland_model = make_model()

keras.utils.plot_model(
  shrubland_model, show_shapes=True, show_layer_activations=True, dpi=1200
)
```

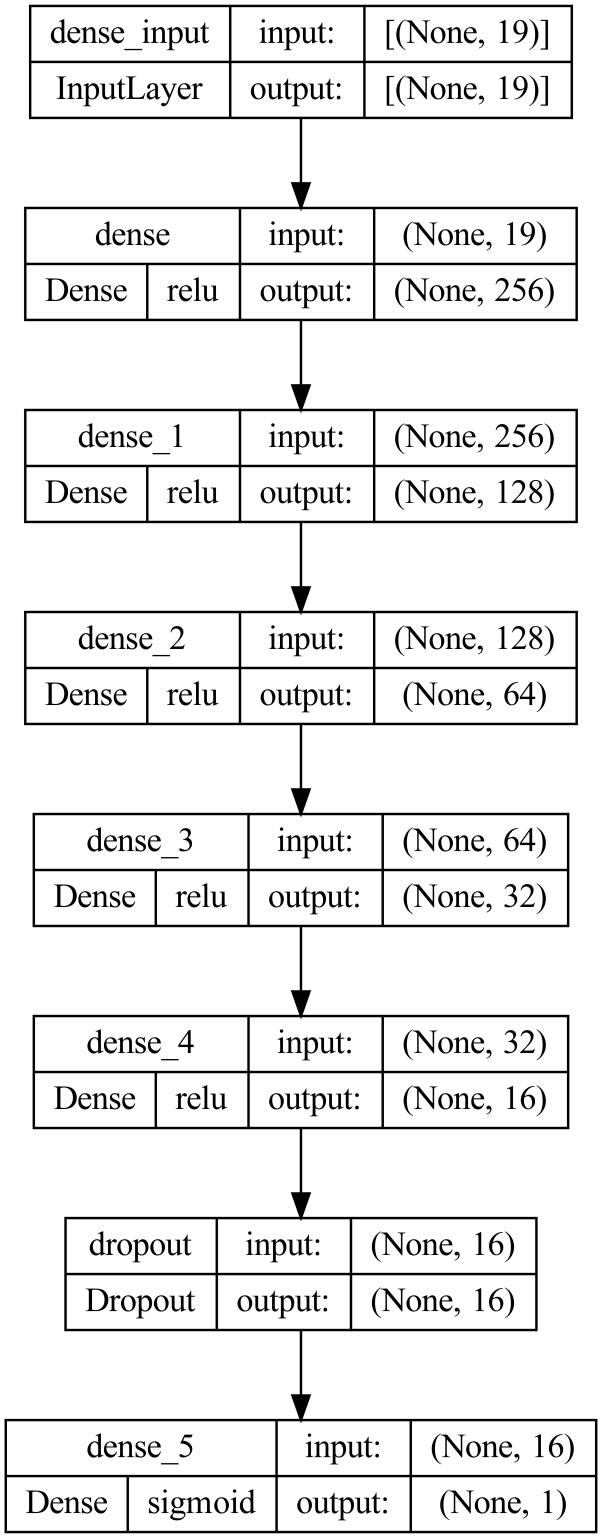

Figure 3: A schematic showing the structure of our neural network.

As we're fitting a rather deep neural network against rather simple structured data, we need to be careful to avoid overfitting while we train the model. As a result, we should define a way to stop our training process early once we stop seeing improved accuracy against the validation data set. We can use the `EarlyStopping()` function to enforce this behavior, so that we'll cut the training process short once we stop seeing improvements in PRC against the validation data:

```
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor="val_PRC", verbose=1, patience=10, mode="max", restore_best_weights=True
)
```

And now we're ready to fit the model! Because we're using early stopping, we can set the number of epochs to use extremely high, as we'll automatically use the most successful iteration for our final model. We'll also make sure to use the class weights we defined earlier:

```
resampled_history = shrubland_model.fit(
    train_ds,
    steps_per_epoch=20,
    epochs=1000,
    callbacks=[early_stopping],
    validation_data=(val_ds),
    class_weight=class_weight,
    verbose=0,
)
```

```
Restoring model weights from the end of the best epoch: 36.
```

```
Epoch 46: early stopping
```

It appears that our model's PRC score stops improving after 36 epochs, which due to our "patience" value of 10 causes our early stopping rules to kick in after epoch 46. We can visualize this process by plotting the PRC values from each epoch of model training, using the `resampled_history` object returned from the fitting process:

```
import matplotlib.pyplot as plt

plt.plot(resampled_history.history["PRC"], label="PRC (training data)")
plt.plot(resampled_history.history["val_PRC"], label="PRC (validation data)")
plt.ylabel("Metric value")
plt.xlabel("Epoch number")
plt.legend(loc="upper left")
```

```
plt.show()
```

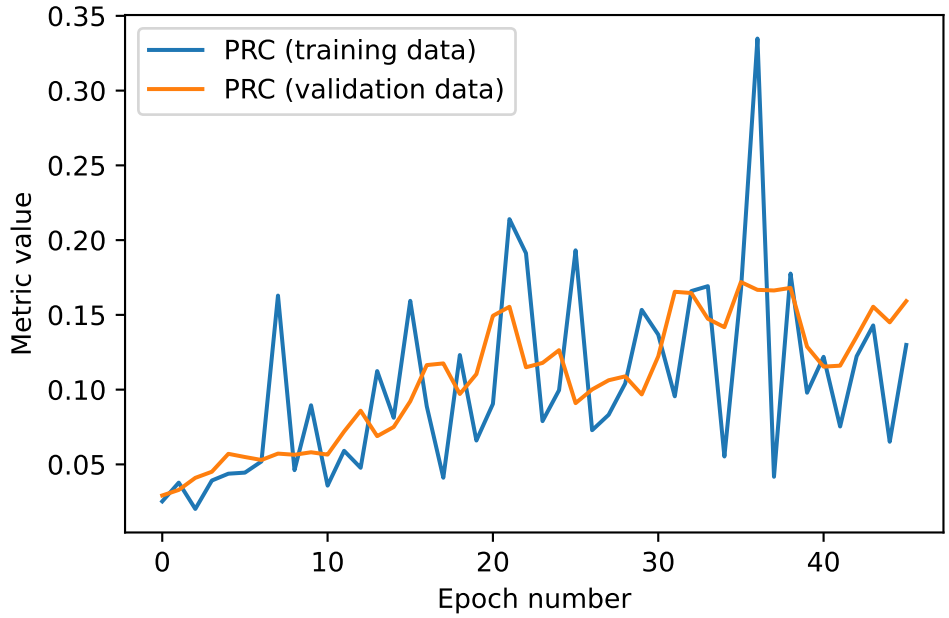

Figure 4: Precision-recall curve (PRC) at each epoch of model training. Higher PRC values indicate a better classifier.

This graph (Figure 4) provides more information about the model fitting process than simply knowing when our early stopping rules kicked in.

It appears that, even though our highest PRC score was achieved after 36 epochs, we might have achieved even higher PRC values had early stopping not kicked in.

For the purposes of this tutorial, we're going to continue using the model produced after 36 epochs. Later, as part of the Assignment section of the chapter, you might want to try other parameters in the early stopping function to see if you can improve the performance of the model.

5.3 Model Evaluation

And just like that, we have a neural net trained to identify shrubland! Now that our model is fully trained (after 36 epochs), the next step is to evaluate it against our hold-out test data frame. We can use the `evaluate()` method of our model object to do so:

```
results = shrubland_model.evaluate(test_ds, verbose=0)

for name, value in zip(shrubland_model.metrics_names, results):
    print(name, ": ", value)
print()
```

```
loss :  0.360729843378067
True_Positives :  17973.0
False_Positives :  289779.0
True_Negatives :  1121321.0
False_Negatives :  3968.0
Binary_Accuracy :  0.7950184345245361
Precision :  0.058400921523571014
Recall :  0.8191513419151306
AUC :  0.8857248425483704
PRC :  0.1751827746629715
```

We'll discuss these results in more detail in the Discussion (Section 6). For now, though, make note of how high our model's AUC (area under the ROC curve) is, compared to its PRC (area under the precision-recall curve) and precision.

Last but not least, it's time for us to visualize our predictions to get a sense of where our model believes we're most likely to find shrublands. In order to map our results, we need to first generate a prediction for each cell in our raster. To do that, we need to first pre-process our full data set in the same way as our training and test data by rescaling it and transforming it to a TensorFlow dataset:

```
full_array = np.array(full_features)
full_array = scaler.transform(full_array)

full_ds = tf.data.Dataset.from_tensor_slices((full_array, full_labels)).cache()
full_ds = full_ds.batch(64).prefetch(2)
```

We can then generate a prediction for each cell of our raster using our model's `predict()` method:

```
predictions = pd.DataFrame(shrubland_model.predict(full_ds, verbose=0))

full_data = pd.concat(
    [full_data, predictions],
    axis=1,
)
```

Now all that's left is to save our predictions out as a raster file, so that we can visualize them in our favorite GIS tool. In order to save space, we'll only save our X and Y coordinates and predictions in the output raster, producing a simple XYZ raster file:

```
location_predictions = full_data[["x", "y", 0]]
location_predictions.columns = ["x", "y", "z"]
```

We'll use `rasterio` in order to save this out as a raster file that GIS tools will understand. Let's import it (and its dependency, `affine`) now:

```
import rasterio
from affine import Affine
```

A raster file is effectively an array of values, with each cell's X and Y position in the array corresponding to its X and Y position in space. As such, in order to create a raster file we must first transpose our one-dimensional column of predictions into a two-dimensional array.

Our first step in this process is to find the corners of our data's bounding box:

```
xmin = location_predictions["x"].min()
xmax = location_predictions["x"].max()
ymin = location_predictions["y"].min()
ymax = location_predictions["y"].max()
```

We then need to identify the cell positions of each pixel in our data set:

```
# Resolution of our Landsat-derived predictors:
# Each observation represents a 30-meter square "pixel" of the map
pixel_size = 30

# Identify the X and Y values for each pixel in our output raster
xv = pd.Series(np.arange(xmin, xmax + pixel_size, pixel_size))
yv = pd.Series(np.arange(ymin, ymax + pixel_size, pixel_size)[::-1])

# Get the X and Y cell indices for each of these pixels
xi = pd.Series(xv.index.values, index=xv)
yi = pd.Series(yv.index.values, index=yv)
```

And we'll then use those positions to create an empty array, which we'll then fill in with our predicted values:

```
# Create an empty array of the proper size for our data:
nodata = -9999.0
zv = np.ones((len(yi), len(xi)), np.float32) * nodata

# Fill in the array with our predicted values, wherever they exist:
zv[
    yi[location_predictions["y"]].values, xi[location_predictions["x"]].values
] = location_predictions["z"]
```

And just like that, we've transformed our single-dimension prediction vector into a two-dimensional array. All that remains is to translate that array from a numpy array into a raster file. We'll first define a transformation, to give rasterio instructions on how much area each of our array cells should represent:

```
transform = Affine(pixel_size, 0, xmin, 0, -pixel_size, ymax) * Affine.translation(
    -0.5, -0.5
)
```

And then lastly we'll use rasterio and this transformation to actually write our values out to a GeoTIFF file:

```
# This is the PROJ string for the raster data used in this study
# It represents how to associate the X and Y coordinates with real world data
projection = "+proj=aea +lat_0=23 +lon_0=-96 +lat_1=29.5 +lat_2=45.5"
projection = projection + " +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs"

with rasterio.open(
    # Our output file name
    "predictions.tiff",
    # What mode to open the file in -- here, write mode
    "w",
    # What driver to use to write our file
    "GTiff",
    # Number of columns to write
    len(xi),
    # Number of rows to write
    len(yi),
    # How many "bands" to write
    1,
    projection,
    # The transformation created above
    transform,
```

```
    # What data type to save as
    rasterio.float32,
    # What value indicates a missing value
    nodata,
) as ds:
    ds.write(zv.astype(np.float32), 1)
```

Once we've saved this GeoTIFF file, we can visualize it in any GIS program to see where our model predicts shrubland is located (Figure 5).

6 Discussion

So we've fit our models, predicted our data, and made a map of the results. But what do our results actually mean, both for our ability to identify shrubland and for how we understand model performance?

A lot of researchers are immediately drawn to the best performance metrics of the model – in this case, likely our fantastic AUC statistic. However, remember that our data is severely imbalanced, with only 1.5% of the training data representing shrublands. AUC is a measure of how well your model performs at the "pairing test" – that is, it represents how well our model would do at classifying two observations if one was guaranteed to represent shrubland and one was guaranteed to not (Hand 2009).

In that situation, our model would give the right results 89% of the time, which makes it a highly effective classifier. However, given that only 1.5% of our region is shrubland, the scenario described by AUC isn't a great representation of how our model actually performs on the ground.

More interesting are our model's recall and precision scores. Our recall score – that is, the proportion of actual "true" shrublands which the model calls shrubland – is extremely high for this model. Our model produces very few false negative predictions. However, our precision – the proportion of shrubland predictions which actually reflect "true" shrubland – is much lower, as we have a very high number of false positives. Depending on our goals for this model, this may be desirable; if our aim is to identify the majority of shrubland across the state, we may accept these false positives as a necessary drawback of that goal. However, if our goal is to produce the most accurate map of shrubland locations possible, for instance to try and choose sites for fieldwork in shrubland regions, we might want a higher precision in exchange for a lower recall value.

Because our model predicts the probability that an observation represents shrubland, and not just the class, we can use different classification thresholds to balance recall and precision according to our tastes. For instance, we can see a large increase in model precision if we require a predicted probability of more than 90% before we classify an observation as shrubland:

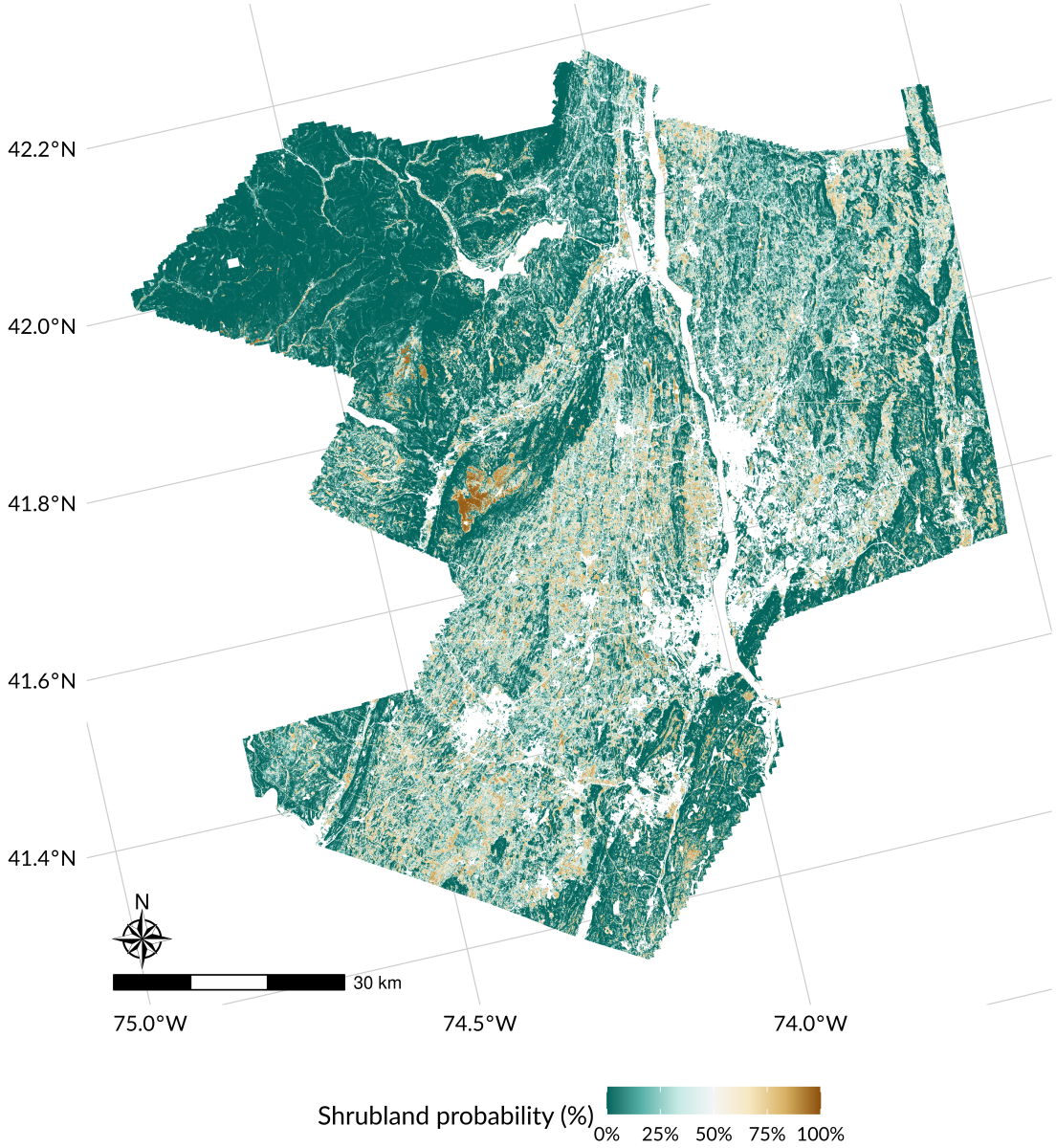

Figure 5: A map of predicted probability of shrubland occurance across New York's lower Hudson River valley, including Duchess, Orange and Ulster counties.

```
test_predictions = pd.DataFrame(shrubland_model.predict(test_ds, verbose=0))

import statistics

statistics.mean(test_labels.loc[np.array(test_predictions[0] > 0.9)])
```

```
0.5772755039706781
```

And an even bigger improvement if we require a probability of at least 95%:

```
statistics.mean(test_labels.loc[np.array(test_predictions[0] > 0.95)])
```

```
0.8323932312651088
```

Of course, this improved precision comes at the cost of a decrease in recall. This is a common trade-off with classification models: decreasing the number of false positives also decreases the number of true positives predicted by a given model. The appropriate threshold to use when making predictions for any classifier will be dependent upon the relative costs of false negative and false positive predictions for your use case.

Perhaps more interesting than the specific probability predictions is the spatial arrangement of predictions across our study area (Figure 5). Generally speaking, it appears like our model is expecting shrubland to be more dominant in areas along road networks and rivers (which are white in the map, as they were excluded from our input data set) – which makes a lot of sense, as these are the areas more likely to have been recently impacted by humans. In this way, mapping the results of a predictive model can help us to understand the patterns and processes happening across the landscape, even without the use of an inferential or causal framework. Being able to visualize what areas are more likely to be shrubland, in this scenario, can help us generate hypotheses for why shrubland occurs where it does and perhaps even suggest future areas for inferential investigation.

7 Summary

This chapter provided a step-by-step walk through of the process for producing models of a rare land-cover class, using a case study attempting to identify shrublands across a region in New York State. Due to the rarity of shrublands in this region, specific attention was paid to how to model imbalanced classes and how to measure model performance with specific objectives for the model. While our model was better at identifying shrubland than random chance alone (with a precision multiple times greater than the 1.5% "base rate" of all pixels

being shrubland), the rarity of this land cover class means that the model's precision is rather low in absolute terms. As higher predicted probabilities of shrubland are, as expected, more likely to represent actual shrubland areas, adjusting the classification threshold to require higher probabilities can help to improve model performance.

More generally, this chapter focused on the difficulties of modeling rare events, and approaches that can be used in this common situation. It is frequently true that rare events and abnormalities are more scientifically interesting than the baseline case, and as such it is important to be able to model and predict these situations. By being able to assign class weights and thinking carefully about model performance metrics, we're able to apply most modeling tools to this common type of problem.

8 Assignment

- Try altering the architecture of the neural network – remove layers, change the number of nodes, alter the early stopping callback, and generally play with the form of the model. Can you out-perform the model from the chapter?
- What happens if you use a different metric for early stopping? Can you optimize for a different performance metric?
- What happens if you change the class weights to more strongly emphasize shrublands? To de-emphasize them? What metrics are impacted the most?

9 Open Questions

There remain some clear future directions for this model:

- Could additional predictors (derived from Landsat imagery or other remote sensing data sources) improve predictive accuracy?
- Could this model be used to track the development of shrubland areas over time, in order to monitor the abundance and distribution of this land cover type?
- Will the reported performance statistics remain stable as the model is used to extrapolate into other regions of New York? Into other regions of the country?
- Could a similar approach be used to track other novel land cover classes, or a finer gradation of land cover types than is usually modeled in LULC studies?
- Could more complex models, such as convolutional neural networks, achieve higher accuracy against this data set?

References

Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2015. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems." https://www.tensorflow.org/.

Brown, Jesslyn F., Heather J. Tollerud, Christopher P. Barber, Qiang Zhou, John L. Dwyer, James E. Vogelmann, Thomas R. Loveland, et al. 2020. "Lessons Learned Implementing an Operational Continuous United States National Land Change Monitoring Capability: The Land Change Monitoring, Assessment, and Projection (LCMAP) Approach." *Remote Sensing of Environment* 238: 111356. https://doi.org/10.1016/j.rse.2019.111356.

Carrera, Ero, Peter Nowee, and Sebastian Kalinowski. 2021. *Pydot*. https://github.com/pydot/pydot.

Chollet, François. 2015. "Keras." https://keras.io.

Cramer, Viki A, Richard J Hobbs, and Rachel J Standish. 2008. "What's New about Old Fields? Land Abandonment and Ecosystem Assembly." *Trends in Ecology & Evolution* 23 (2): 104–12. https://doi.org/10.1016/j.tree.2007.10.005.

Falkowski, Michael J., Jeffrey S. Evans, Sebastian Martinuzzi, Paul E. Gessler, and Andrew T. Hudak. 2009. "Characterizing Forest Succession with Lidar Data: An Evaluation for the Inland Northwest, USA." *Remote Sensing of Environment* 113 (5): 946–56. https://doi.org/https://doi.org/10.1016/j.rse.2009.01.003.

Fargione, Joseph E, Steven Bassett, Timothy Boucher, Scott D Bridgham, Richard T Conant, Susan C Cook-Patton, Peter W Ellis, et al. 2018. "Natural Climate Solutions for the United States." *Science Advances* 4 (11): eaat1869. https://doi.org/10.1126/sciadv.aat1869.

Foster, David R, Glenn Motzkin, and Benjamin Slater. 1998. "Land-Use History as Long-Term Broad-Scale Disturbance: Regional Forest Dynamics in Central New England." *Ecosystems* 1 (1): 96–119. https://doi.org/10.1007/s100219900008.

Gillies, Sean et al. 2013. *Rasterio: Geospatial Raster i/o for Python Programmers*. Mapbox. https://github.com/rasterio/rasterio.

Hand, David J. 2009. "Measuring Classifier Performance: A Coherent Alternative to the Area Under the ROC Curve." *Machine Learning* 77: 103–23.

Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, et al. 2020. "Array Programming with NumPy." *Nature* 585 (7825): 357–62. https://doi.org/10.1038/s41586-020-2649-2.

Hobbs, Richard J, Eric Higgs, and James A Harris. 2009. "Novel Ecosystems: Implications for Conservation and Restoration." *Trends in Ecology & Evolution* 24 (11): 599–605. https://doi.org/10.1016/j.tree.2009.05.012.

Hunter, J. D. 2007. "Matplotlib: A 2d Graphics Environment." *Computing in Science & Engineering* 9 (3): 90–95. https://doi.org/10.1109/MCSE.2007.55.

Kennedy, Robert E, Zhiqiang Yang, and Warren B. Cohen. 2010. "Detecting Trends in Forest Disturbance and Recovery Using Yearly Landsat Time Series: 1. LandTrendr — Temporal Segmentation Algorithms." *Remote Sensing of Environment* 114 (12): 2897–2910. https://doi.org/10.1016/j.rse.2010.07.008.

Kennedy, Robert E, Zhiqiang Yang, Noel Gorelick, Justin Braaten, Lucas Cavalcante, Warren

B. Cohen, and Sean Healey. 2018. "Implementation of the LandTrendr Algorithm on Google Earth Engine." *Remote Sensing* 10 (5). https://doi.org/10.3390/rs10050691.

King, David I., and Scott Schlossberg. 2014. "Synthesis of the Conservation Value of the Early-Successional Stage in Forests of Eastern North America." *Forest Ecology and Management* 324: 186–95. https://doi.org/10.1016/j.foreco.2013.12.001.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015. "Deep Learning." *Nature* 521: 436–44. https://doi.org/10.1038/nature14539.

Mahoney, Michael J, Lucas K Johnson, and Colin M Beier. 2022a. "Classification and Mapping of Low-Statured 'Shrubland' Cover Types in Post-Agricultural Landscapes of the US Northeast." arXiv. https://doi.org/10.48550/ARXIV.2205.05047.

———. 2022b. "Data for: AI for Shrubland Identification and Mapping (in AI For Earth Science)." Zenodo. https://doi.org/10.5281/zenodo.6824173.

McKinney, Wes. 2010. "Data Structures for Statistical Computing in Python." In *Proceedings of the 9th Python in Science Conference*, edited by Stéfan van der Walt and Jarrod Millman, 56–61. https://doi.org/ 10.25080/Majora-92bf1922-00a .

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. "Scikit-Learn: Machine Learning in Python." *Journal of Machine Learning Research* 12: 2825–30.

Python Core Team. 2022. *Python: A dynamic, open source programming language*. Python Software Foundation. https://www.python.org/.

Ruiz, Luis Ángel, Jorge Abel Recio, Pablo Crespo-Peremarch, and Marta Sapena. 2018. "An Object-Based Approach for Mapping Forest Structural Types Based on Low-Density LiDAR and Multispectral Imagery." *Geocarto International* 33 (5): 443–57. https://doi.org/10.1080/10106049.2016.1265595.

team, The pandas development. 2020. *Pandas-Dev/Pandas: Pandas* (version latest). Zenodo. https://doi.org/10.5281/zenodo.3509134.

Wickham, James, Stephen V. Stehman, Daniel G. Sorenson, Leila Gass, and Jon A. Dewitz. 2021. "Thematic Accuracy Assessment of the NLCD 2016 Land Cover for the Conterminous United States." *Remote Sensing of Environment* 257: 112357. https://doi.org/https://doi.org/10.1016/j.rse.2021.112357.