Outline

1 Neural Networks

Notes Artificial Intelligence. Sounds fancy, but how does it work?

2 Computer Vision

Notes Convolutional Neural Networks. The Basics.

Module Building our own binary classifier: Fine-Tuning a trained CNN.

Module Building our own multiclass classifier.

[Notes]
Artificial Intelligence. Sounds fancy, but how does it work?

Neural Networks
Why?

Artificial Neural Networks and Deep Learning have drastically improved machine-learning performance.

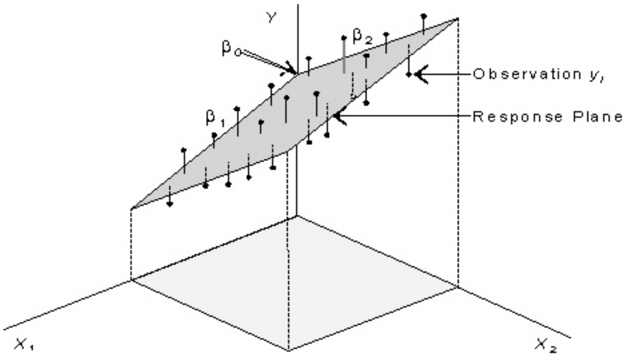
- Speech-recognition (e.g. Siri, Echo, Alexa)
- Translation (e.g. Google translator)
- Image recognition (e.g. Facebook's facial recognition photo tagging)

Neural Networks

In "conventional" machine learning, we only use a single parameter matrix: 1 variable = 1 coefficient.

Linear Model: regression formula

$$y_i = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon_i$$



Neural Networks

In “conventional” machine learning, we only use a single parameter matrix: 1 variable = 1 coefficient.

Linear Model: compact matrix form

$$\mathbf{Y} = \mathbf{X}\beta$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ 1 & x_{31} & x_{22} \\ 1 & x_{41} & x_{22} \\ \vdots & \vdots & \\ 1 & x_{n1} & x_{n2} \end{bmatrix} * \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}$$

Neural Networks

In "conventional" machine learning, we only use a single parameter matrix: 1 variable = 1 coefficient.

- Interested in finding the parameter matrix β that minimizes predictive error
- This is easy when using a Least Squares regression because there is an analytic solution

$$\beta = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'y$$

- We can use MLE to find this parameter matrix for more complex general linear models

$$\mathbf{Y} = logit(\mathbf{X}\beta)$$

Neural Networks

Conventional machine learning only take us so far... what about extending the learning process?

What if we use the output of a first model as input of a second model...

$$\hat{\mathbf{Y}}_2 = \mathbf{X}\beta_1$$

$$\hat{\mathbf{Y}} = \mathbf{X}_2\beta_2$$

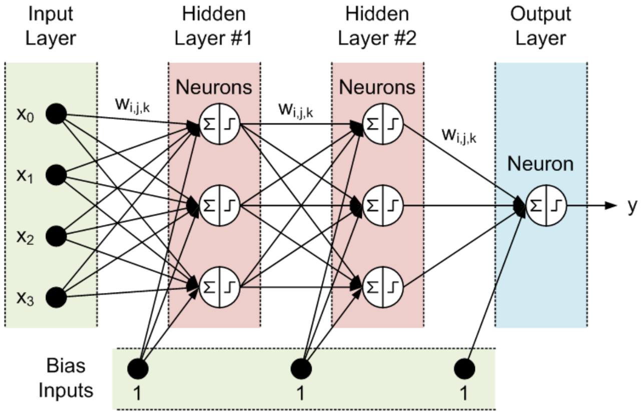
where

$$\hat{\mathbf{Y}}_2 = \mathbf{X}_2$$

and we try to minimize $\mathbf{Y} - \hat{\mathbf{Y}}$ instead of $\mathbf{Y} - \hat{\mathbf{Y}}_2$

This is what we call a Neural Network or Artificial Neural Network!

Neural Networks
Neural nets are often represented this way



To be technical, the term “deep learning” should be used only when the neural networks have a several hidden layers.

Neural Networks

Matrix multiplication is the key to understand neural nets!

Remembers these 2 key principles of matrix multiplication:

1 the number of columns in the first matrix has to be the same as the number of rows in the second matrix

2 the number of rows of the resulting matrix will equal the number of rows of the first matrix, and the number of columns will equal the number of columns of the second matrix

$$A[n, k] * B[k, z] = C[n, z]$$

Neural Networks
Matrix multiplication is the key to understand neural nets!

Instead of a simple linear or general linear model we can have a model that looks like this...

$$\text{Sigmoid}(\mathbf{X}[1000, 4]\ \beta_1[4, 250])\ \beta_2[250, 1] = \mathbf{Y}[1000, 1]$$

...

(1) $\mathbf{X}[1000, 4]\ \beta_1[4, 250] \rightarrow \mathbf{X_2}[1000, 250]$

(2) $\text{Sigmoid}(\mathbf{X_2}[1000, 250]) \rightarrow \mathbf{X_{2b}}[1000, 250]$

(3) $\mathbf{X_{2b}}[1000, 250]\ \beta_2[250, 1] \rightarrow \hat{\mathbf{Y}}[1000, 1]$

We calculate the parameters in the matrices β_1 and β_2 using e.g. Stochastic Gradient Descent $\rightarrow$ iterating until convergence

Neural Networks

Some basic terminology... different words for some familiar concepts

- **input layer**: the original data matrix ($\mathbf{X}$)
- **weight/s**: a single parameter (β_{ij}) / parameter matrix (β)
- **bias**: the intercept parameter matrix (α or β_0)
- **ReLu, Sigmoid, Tanh**: non-linear transformation we apply to $\mathbf{X}$ matrices. Also known as **activation functions**
- **hidden layer**: $\mathbf{X_2}, \mathbf{X_3}, \ldots$ a new intermediate representation of the input
- **loss function**: the function we want to minimize (e.g. $\hat{\mathbf{Y}} - \mathbf{Y}$)
- **regularization**: transformations we apply to the loss function (e.g. $|\hat{\mathbf{Y}} - \mathbf{Y}| \rightarrow$ L1 and $(\hat{\mathbf{Y}} - \mathbf{Y})^2 \rightarrow$ L2) or to the variables/columns of the input matrix
- **dropout**: setting some β_{ij} from a β matrix to 0 at random
- **forward propagation**: performing all the matrix multiplications
- **backpropagation**: calculating Stochastic Gradient Descent

Neural Networks

Some more terminology and... hyperparemeters, the dark mysteries of neural nets

- **graph**: a model
- **train-validation-test split**: 80-10-10? 50-25-25?
- **batch size**: the number of training observations we use for training in a given iteration
- **epochs**: number of times we run through every training iteration
- **learning rate**: by how much we update the weights at each training iteration
- Other terms: momentum, gamma, step size, etc....

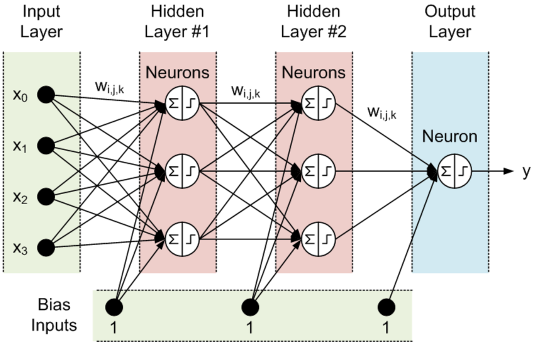
There are some conventions people follow, such as starting with a larger learning rate and then decreasing it as the iterations/epochs progress. Since we are preforming supervised training, we always look for the hyperparameters that achieve the highest out-of-sample accuracy.

Neural Networks

Fine tuning or transfer learning

Slightly tweaking an already trained neural net to predict a different outcome. Options are:

- Retraining the whole neural net with new data
- Retraining part of the neural net with new data
- Adding or changing layers

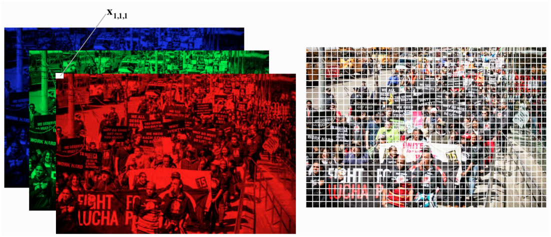


[Notes]
Convolutional Neural Networks. The Basics.

Convolutional Neural Nets for Computer Vision

Two main differences

(1) Images as inputs: 3-dimensional matrices (width x height x depth)



$$\mathbf{X} =$$

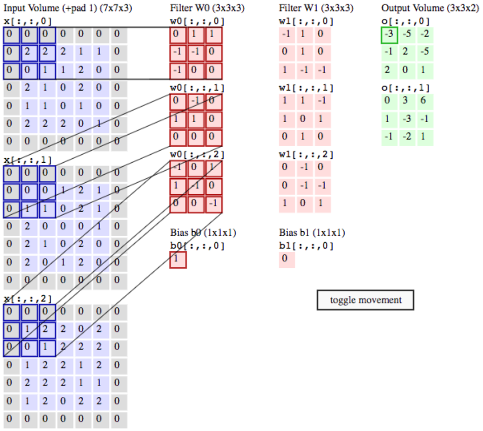
$$\begin{bmatrix} x_{111} & x_{112} & \dots & x_{11n} \\ x_{121} & x_{122} & \dots & x_{12n} \\ x_{131} & x_{132} & \dots & x_{13n} \\ x_{141} & x_{142} & \dots & x_{14n} \\ \vdots & \vdots & & \\ x_{1n1} & x_{1n2} & \dots & x_{1nn} \end{bmatrix}, \begin{bmatrix} x_{211} & x_{212} & \dots & x_{21n} \\ x_{221} & x_{222} & \dots & x_{22n} \\ x_{231} & x_{232} & \dots & x_{23n} \\ x_{241} & x_{242} & \dots & x_{24n} \\ \vdots & \vdots & & \\ x_{2n1} & x_{2n2} & \dots & x_{2nn} \end{bmatrix}, \begin{bmatrix} x_{311} & x_{312} & \dots & x_{31n} \\ x_{321} & x_{322} & \dots & x_{32n} \\ x_{331} & x_{332} & \dots & x_{33n} \\ x_{341} & x_{342} & \dots & x_{34n} \\ \vdots & \vdots & & \\ x_{3n1} & x_{3n2} & \dots & x_{3nn} \end{bmatrix}$$

Convolutional Neural Nets for Computer Vision

Two main differences

(2) Convolutional layers: weights (**filters**) are not connected to the whole **input volume**: convolution.

Click here for a full visualization by the Stanford cs231 folks.



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	2	2	2	1	1	0
0	0	1	1	2	0	0
0	2	1	0	2	0	0
0	1	1	0	1	0	0
0	2	2	2	0	1	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	0	0	1	2	1	0
0	1	1	0	2	1	0
0	2	0	0	0	1	0
0	2	1	0	2	0	0
0	2	0	1	2	0	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	1	2	2	0	2	0
0	0	1	2	2	2	0
0	1	2	2	1	2	0
0	2	2	2	1	1	0
0	1	2	0	2	2	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

0	1	1
-1	-1	0
-1	0	0

w0[:,:,1]

0	-1	0
1	1	0
0	0	0

w0[:,:,2]

-1	0	1
1	-1	0
0	0	-1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

-1	1	0
0	1	0
1	-1	-1

w1[:,:,1]

1	1	-1
1	0	1
0	1	0

w1[:,:,2]

0	-1	0
0	-1	-1
1	0	1

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

-3	-5	-2
-1	2	-5
2	0	1

o[:,:,1]

0	3	6
1	-3	-1
-1	-2	1

Convolutional Neural Nets for Computer Vision
Some new terminology... and more hyperparameters

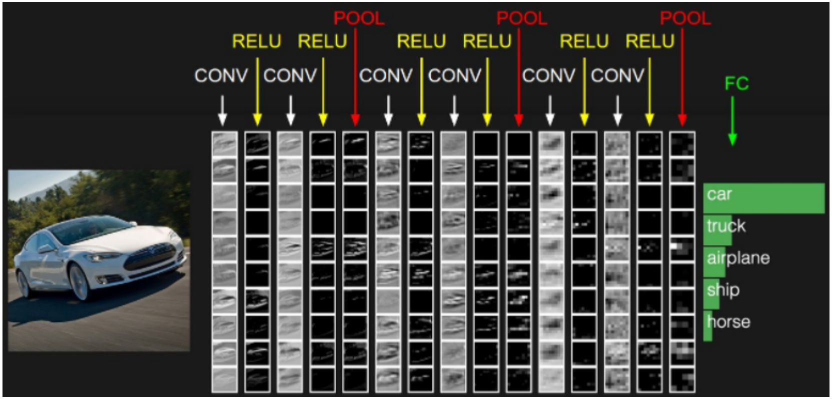
- **input volume**: a 3-dimensional input (or might be grayscale only)
- **convolutional layer**: a 4-dimensional parameter layer where convolutional filters are applied to the input volume; of size FxFxNxK where F is the width and height of the filter, N is the number of filter dimensions, and K is the number of filters $\rightarrow$ 3x3x3x2 in the previous example
- **stride**: the number of pixels we move the filter at a time. This is 2 in the previous example
- **zero-padding**: adding zeros around the input border (often done to avoid deforming input images or missing things at the edges)
- **pooling layer**: a layer where we reduce the size the output of a convolutional layer. From 224x224x3x64 to 112x112x3x64 for example.

Convolutional Neural Nets for Computer Vision

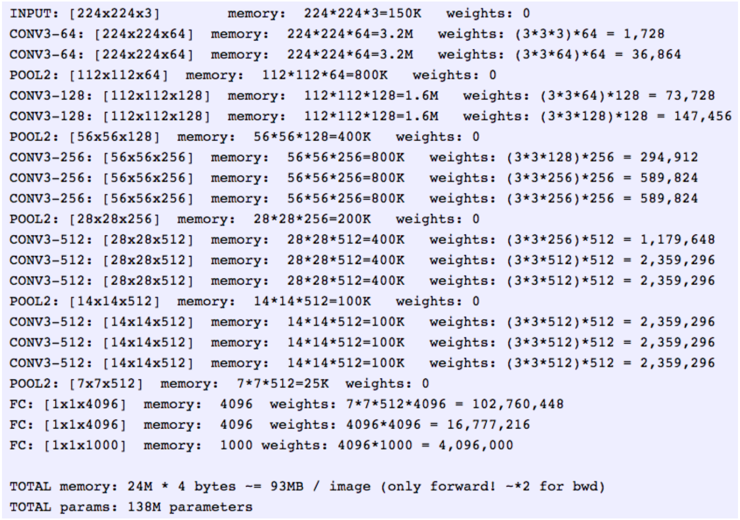
Some new terminology... and more hyperparameters

- **fully connected layer**: a layer of weights that is connected to the whole input volume. These are usually at the end of a network.
- **softmax**: a multi-class classifier. This is basically a multinomial logit model that uses the output of the last fully-connected layer to predict the final classes of interest

Convolutional Neural Nets for Computer Vision
This is what a ConvNet looks like



Convolutional Neural Nets for Computer Vision
VGG16's architecture



```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K   weights: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]  memory:  14*14*512=100K   weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
```

Hands-on Demo Outline

- What are we doing in the demo? What steps would you follow to replicate the demo with your own data?
- What is an “accurate” classifier?
- Example data for demo
- Fine-tuning a CNN on Code Ocean
- Try it out: hands-on demo

What we're doing at the highest level

- Select a pre-trained CNN [Resnet18, using the PyTorch framework]
- Adjusting the last layer of the CNN
- Provide the CNN with true positive and true negative images split into *train* and *validation* sets for each classifier [80-20 split]
- Run the CNN over multiple iterations until it learns to accurately classify the images
- Use the re-trained CNN to classify additional images

Project pipeline

- Collect images
- Deduplicate images
- Manually label images for classification of interest
- Organize images into required directory structure with your chosen train/test split
- Move data and code onto cloud computer [or run on your own machine]
- Adjust Python code as needed (last layer of CNN, paths, hyper-parameters), then train models
- Save the most accurate model to use in the future for classifying large sets of images for your feature of interest

Evaluating the most accurate model

There are four main measures we use to evaluate the "goodness" of a machine learning model:

- Accuracy: Percent of predictions by the classifier that are correct
- Precision: Percent of predicted positives that were actually positives
- Recall: Percent of actual positives that were predicted positives
- F1 score: Harmonic mean of precision and recall

For our demo: Example data

For the binary classifiers:

- Images collected from Twitter for previous paper supplemented with images found online
- Number of true positive images for each classifier:
  - **Protest** [n = 100]
  - **John Legend** [n = 100]
- You can see the images and the file structure in the Code Ocean '`data`' directory

Example data

For the multi-class classifier:

- Pictures of world leaders found online and from Microsoft celebrity faces database
- 6 country leaders [50 images per individual]
- You can see the images and the file structure in the Code Ocean '`data`' directory

Python Options for CNNs

- PyTorch
- Keras, TensorFlow, Caffe2, Gluon

Demo

Hands-on demo on Code Ocean

- Link to the Code Ocean capsule on `https://github.com/norawebbwilliams/images_as_data`.
- Duplicate the capsule to your account and start a cloud work session with Jupyter Notebooks
- If you finish playing with the binary classifier, try the multi-class version. How does this script and data structure differ?
- You can also check out the script for manipulating images to learn more about preprocessing
- Or try running it on your own images, if you've got them and can quickly set up the file structure!

Convolutional Neural Nets for Computer Vision
Let's practice!

[Modules]
Let's practice! Switching over to Code Ocean