12. Intro to Neural Networks

Elliott Ash

Text Data Course, Bocconi 2018

“Neural Networks”

- “Neural”:
  - nothing like brains
- “Networks”:
  - nothing to do with “networks” as normally understood – in particular, nothing to do with network theory in social science.

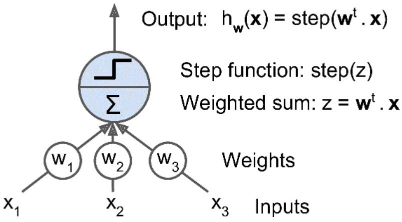
Recent History

- NNs frequently outperform other ML techniques on very large and complex problems.
- Increase in computing power makes them computationally tractable, graphical processing units (GPUs, designed for video games) give you over 100x performance gain over CPUs.
- Training algorithms have improved – small tweaks have made a huge impact.
- Some theoretical limitations of ANNs have turned out to be benign in practice – for example, they work well on non-convex functions.

Will it last?

- Three key principles of deep learning will persist:
  - **Simplicity**
    - feature engineering is obsolete
    - complex, brittle, engineering-heavy pipelines replaced with simple, end-to-end trainable models, composed of 5-6 tensor operations.
  - **Scalability**
    - amenable to parallelization on GPUs or TPUs (tensor processing units)
    - trained on batches of data, so can be scaled to datasets of arbitrary size.
  - **Versatility and reusability**
    - can be trained on additional data without restarting from scratch, therefore amenable for continuous online learning.
    - deep-learning models are repurposable and thus reusable

Perceptron LTU



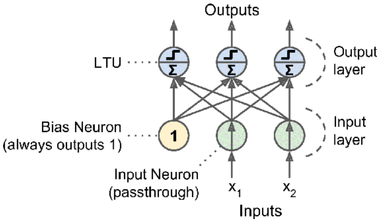
- In a perceptron, an individual neuron (called an LTU, or linear threshold unit) is defined by

$$h(\boldsymbol{x}) = \text{step}(\boldsymbol{\omega}'\boldsymbol{x})$$

  where step$(\cdot)$ is the step function.
  - The neuron computes a linear combination of the inputs; if result exceeds threshold, output positive class, otherwise negative class.

Perceptron

- A perceptron is an array of LTUs in parallel:



- This basic perceptron is similar to a logistic regression model.

In Notation

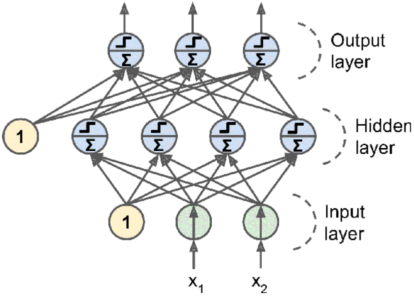
- The simplest perceptron is a linear combination of the inputs, the same as a linear regression:

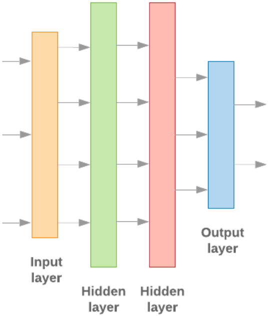
  $$y = \alpha + x'\omega$$

  where x is a vector of inputs and ω is the vector of weights (or a matrix for a multi-class outcome).

Hidden Layers $\leftrightarrow$ "Deep Learning"

- The predictive performance of perceptrons improved substantially by stacking them into multiple layers:





- Input variables are connected to multiple neurons in the hidden layer(s), which in turn are connected to the output layer.
  - This is called a multi-layer perceptron or a feed forward neural network; with enough neurons, it can approximate any continuous function.

DNN Notation

- An DNN with a single hidden layer can be written as

$$y = \alpha_2 + g(\alpha_1 + x'\omega_1)'\omega_2$$

  - α_1 and ω_1, the intercept and coefficients in the input layer
    - ω_1 is a $d_0 \times d_1$ matrix, where d_0 is the dimension of the input and d_1 is the number of neurons in the hidden layer.
  - $g(\cdot)$, the non-linear activation function.
    - without this, the DNN could only represent linear transformations of the input.
  - α_2 and ω_2, the intercept and coefficients in the hidden layer.
    - ω_2 is a $d_1 \times d_2$ matrix, where d_2 is the dimensionality of the output.

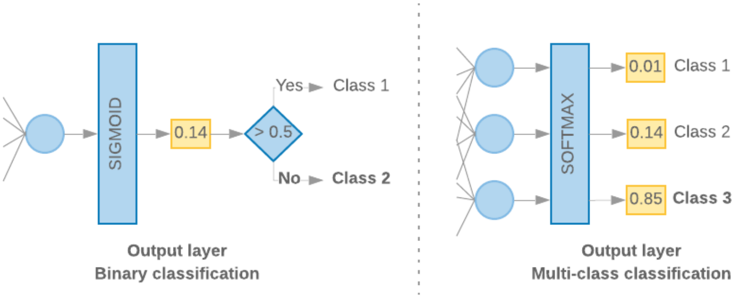
DNN Notation: two hidden layers

- Similarly, with two hidden layers we have

$$y = \alpha_3 + g_2(\alpha_2 + g_1(\alpha_1 + x'\omega_1)'\omega_2)'\omega_3$$

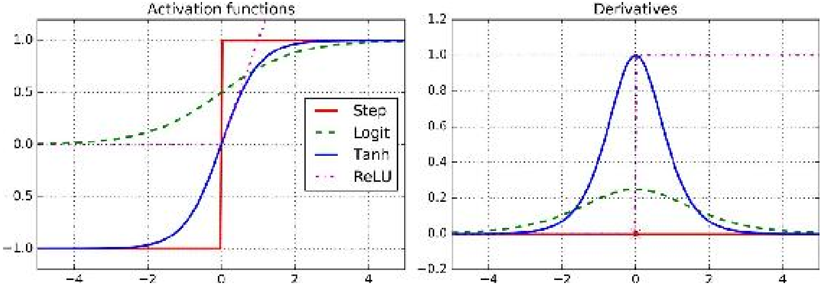
  - $g_1(\cdot)$ and $g_2(\cdot)$, activation functions for the first and second layers.
  - α_3 and ω_3, intercepts and coefficients for the second hidden layer.

Constructing the Last Layer



- MLPs will output a probability distribution across output classes.
  - can also output a real number, which would make a regression model.

Modern MLPs: New activation functions



- logistic function: $\sigma(z) = \frac{1}{1+\exp(-z)}$
- hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$
  - ranges between -1 and 1 (rather than between 0 and 1, as the case with the logistic)
  - centered on zero, can speed up convergence
- ReLU (rectified linear unit) function: $\max\{0, z\}$,
  - deceptively simple, fast to compute, and very effective in practice
  - gradient does not saturate to zero for large values (but is flat below zero)

Google Developers Advice: MLP baseline for Text Classification

1. Calculate the number of samples/number of words per sample ratio.
2. If this ratio is less than 1500, tokenize the text as n-grams and use a simple multi-layer perceptron (MLP) model to classify them.
   - In the case of N-grams models, Google testers found that MLPs tended to out-perform logistic regression and gradient boosting machines.

Setting up a model

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential() # create a sequential model
model.add(Dense(50, # neurons in first layer
          input_dim=X.shape[1], # number of predictors
          activation='relu',)) # activation function
model.add(Dense(50, activation='relu')) # hidden layer
model.add(Dense(1, activation='sigmoid')) # output layer
model.summary()
```

- “Dense” layer is the DNN baseline – means that all neurons are connected.
- Output layer:
  - for binary classification, use `activation='sigmoid'`
  - for regression, do not use an activation function
  - for multi-class classification, use `activation=softmax'`

Visualize a model

```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
dot = model_to_dot(create_reg_model(),
                   show_shapes=True,
                   show_layer_names=False)
SVG(dot.create(prog='dot', format='svg'))
```

Compile the model

```
model.compile(loss='binary_crossentropy', # cost function
              optimizer='adam', # use adam as the optimizer
              metrics=['accuracy']) # compute accuracy
```

- Loss function:
  - for binary classification, use `binary_crossentropy`
  - for regression, use `mean_squared_error`
  - for multi-class classification, use `sparse_categorical_crossentropy`
- Optimizer:
  - use `adam`
- Metrics:
  - for classification, use accuracy
  - for regression, you have to define a custom metric (see accompanying code)

Fit a model

```
model.fit(X, Y,
          epochs=5,
          validation_split=.2)

model.get_weights()

# Plot performance by epoch
plt.plot(model_info.epoch,model_info.history['acc'])
plt.plot(model_info.epoch,model_info.history['val_acc'])
plt.legend(['train', 'val'], loc='best')

# make predictions
ypred = model.predict(X)
```

Saving and Loading Models

```python
# Save a model
model.save('keras-clf.pkl')

# load model
from keras.models import load_model
model = load_model('keras-clf.pkl')
```