1 Module 22: Neural Networks

Contents

Image: $n_{\text{width}} \times n_{\text{height}} \times n_{\text{color channels}}$ = number of features

With so many features, you risk overfitting with traditional models (DTree, SVM, etc.) $\rightarrow$ preprocessing

1.1 Approach

Define features $\phi_0, ..., \phi_{m-1}$ as nonlinear transformations of X such that $\phi_m(x_1, ..., x_D)$ in the vein of high-order-linear/logistic regression.

Aside: Recall, in the case of simple linear reg, we're just tuning slope a and intercept b to minimize the loss function. Extrapolating to m features, then the model

$$\begin{aligned} h(x) &= b + a_1\phi_1(x) + ... + a_{m-1}\phi_{m-1}(x) \\ &= \begin{bmatrix} a_1 & a_2 & ... & a_{m-1} \end{bmatrix} \begin{bmatrix} \phi_1(x) \\ \phi_2(x) \\ ... \\ \phi_{m-1}(x) \end{bmatrix} + b \\ &= A^T\phi(x) + b \end{aligned} \tag{1}$$

The linear regression is then just minimization of the square loss function:

$$\begin{aligned} L(a, b) &= \sum_{i-1}^{N} (h(x_i) - y_i)^2 \\ \text{plug in linreg } \rightarrow &= \sum_{i-1}^{N} (A^T\phi(x_i) + b - y_i)^2 \end{aligned} \tag{2}$$

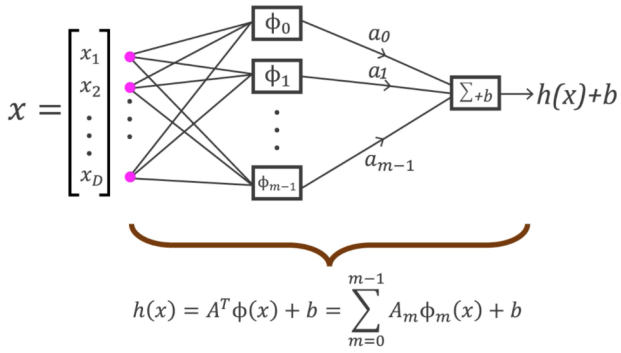

Figure 1: The image is a graph of multifeature linreg. Each line transmits a single number which is modified by weights $a_0, ..., a_{m-1}$.

Returning from aside:
A few generalizations...

1) Neural networks behave similarly, but use specific feature functions called *activation functions*. The most common of which are the

- Sigmoid (on/off switch)
$$\phi(x) = \frac{1}{1+e^{-x}}$$
- Tanh (saturation limits: truncates large $\pm$ values, allows small through $\sim$ unchanged)
$$\phi(x) = \frac{e^{2x}-1}{e^{2x}+1}$$
- ReLu (inhibit negative values)
$$\phi(x) = \max(0, x)$$

2) NNs add coefficients to all lines on the graph (not just the output b), i.e. LinReg $\phi(x) \to$ NN $\phi(Ax + b)$

3) Allow for many layers to be used, sequentially. Each feature vector ϕ is fed by the matrix $A\times$output of previous layer.

4) Add an additional activation function to the outer layer. With a sigmoid, you automatically create a binary/multiclass classification model.

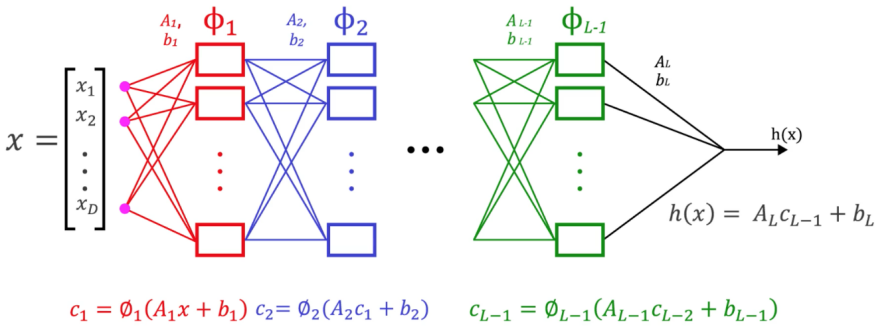

Figure 2: L: number of layers, Last layer $h(x)$ is output layer, $L-1$ hidden layers (outputs have no specific meaning), c: outputs from each layer

The training problem is to find the set of A, b coefficients to minimize loss over L nested functions. *Each layer is computing a set of features to be used by the other layers.*

1.2 Binary classification, Keras example

Google CoLab: Runtime → Change runtime type → GPU

```
from tensorflow import keras
from tensorflow.keras import layers

# 3 hidden layers: 3, 4, 2 layers respectively
model = keras.Sequential(
    layers.Dense(3, activation='relu'), # ReLu is fast to optimize
    # Dense := connections btw all neurons from prev to next
    layers.Dense(4, activation='relu'),
    layers.Dense(2, activation='relu'),
    layers.Dense(1, activation='sigmoid') # Probability £\in£ [0,1]
)

model.compile(
    optimizer = 'rmsprop', # SGD flavor
    loss = 'binary_crossentropy', # for binary classif.
    metrics = ['accuracy']
)

history = model.fit(
```

```
    x = X, y = y,
    epochs = 5, # rounds of SGD
    batch_size = 8 # batch per SGD
)
```

1.3 Multiclass classification, Keras example

Recall in logistic regression the approaches:

- One vs. rest
- One vs. one
- Multinomial logistic regression

The first two are generic and can be used for NNs, but require that we train a *set* of models & compare as an ensemble (expensive for NNs). Instead, let's adapt a NN to produce class probabilities.

Replace single-node output layer with a multi-node layer (1 node per possible label) with **softmax** algo:

- 1) Sum $x_1, ..., x_m$ for each m class.
- 2) Normalize values
  - Preserve orders s.t. if $x_i < x_j$ then $y_i < y_j$
  - The y's $\in$ [0,1]
  - $\sum y = 1$

Softmax mainly allows us to express probabilistically the *certainty* for the output node $\in$ [0,1] (though it may not reflect the true distribution).

```
# Encode labels
from sklearn.preprocessing import LabelBinarizer
lb = LabelBinarizer
y_ohe = lb.fit_transform(y) # Encode

lb.inverse_transform(y_ohe) # Retrieve labels

model = Sequential([
    Dense(5, input_dim=2, activation = 'relu'),
    Dense(3, activation = 'softmax')
])

model.compile(loss = 'categorical_crossentropy', metrics = ['accuracy'])

history = model.fit(X, y_ohe, epochs = 500)
```

Encoding	Integer	One-hot
Label	[0, 1, 2]	[[1,0,0], [0,1,0], [0,0,1]]
Loss	'sparse_categorical_crossentropy'	'categorical_crossentropy'

Table 1:

1.3.1 Integer encoding

This encoding assigns each label a numerical value, rather than creating binary multiclass arrays for each. See table.

1.4 Best practices

- **Batch normalization**: technique for standardizing inputs into a NN, applied either to activations of previous layer or direct inputs
- A **residual connection** is a skip-connection where residual functions are learned wrt layer inputs rather than learning unreferenced functions
- **Hyperparameter optimization** algo.s find the hyperparameter tuple that minimizes a loss fcn. based on independent data
- **Ensemble learning** reduced variance of predictions and generalization error by combining multiple NNs

1.4.1 Hyperparameter tunings

- Depth: number of layers
- Width: number of neurons/layer
- Activation fcn. for each neuron

Generally, deeper → fewer epochs to train. Then, wider → generally similar training rate, but better "fitting". Use cross-validation to contrast models/ prevent overfitting, i.e. `KerasTuner`.

```
history = model.fit(...,
    validation_data = (X_test, y_test)
)
#OR
history = mode.fit(...,
    validation_split = .2)
```

Other hyperparameter tunings

- **Dropout rate**: ignore a randomly selected neuron during training to see the impact of their downstream ativation. `tensorflow.keras.layers.Dropout( 0.2 )`

- **Regularization** (L1): penalize absolute magnitude of coeffs. → produces sparse (few coeff) models. `Dense(100, activation='relu', kernel_regularizer = 'l1')`
- (L2): penalizing square of magnitude, all coeffs are shrunk by same factor (not eliminated)
- **Early stopping**: stop training when oerfitting begins `model.fit(..., callbacks = EarlyStopping(patience = 4))`

1.4.2 Batch gradients

Computing gradient descent in (randomly ordered) "minibatches" – Keras default is 32 (samples per GD step). Epoch contains *all* datasets; num. minibatches = total dataset size / batch size.

Smaller batch sizes → noisier SGD descent & much slower descent/batch (though only ∼/epoch). Individual GD approximations are correspondingly weaker. That said, computations/batch are *much* faster.

1.5 Conclusions

Each transformation in a layer == *neuron* ∼ logistic regressor: compute weighted sum & apply an activation fcn.

Earlier layers compute features used by later layers. As such, combined layers can produce arbitrarily complex nonlinear featurizations of data (which do not need to be found later on).