Overfitting with Neural Networks

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

```python
In [ ]: %load_ext autoreload
%autoreload 2
%matplotlib inline

In [ ]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

In [ ]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.regularizers import l1
from tensorflow.keras.regularizers import l2
from tensorflow.keras.datasets import mnist

In [ ]: # MNIST dataset
    # Load MNIST into four Numpy arrays
    (mnist_x_train, mnist_y_train), (mnist_x_test, mnist_y_test) = mnist.load_data()
    mnist_x_train = mnist_x_train.reshape((60000, 28 * 28))
    mnist_x_train = mnist_x_train.astype("float32") / 255
    # Normalize
    mnist_x_test = mnist_x_test.reshape((10000, 28 * 28))
    mnist_x_test = mnist_x_test.astype("float32") / 255
```

Acknowledgement

- The analogy between dropout and a company whose employees are told to toss a coin to decide whether to go to work each morning comes from A. Géron: *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow (2nd edn)*, O'Reilly, 2019
Introduction

- One of the central problems of deep learning is overfitting
- Reminder: If your model overfits, your main options are:
  - Gather more training data (or use Data Augmentation)
  - Remove noise in the training examples
  - Change model: move to a less complex model
  - Simplify by reducing the number of features
  - Stick with your existing model but add constraints (if you can) to reduce its complexity

- Here we’ll look at:
  - Reducing the network’s size — an example of moving to a less complex model
  - Weight regularization — an example of adding constraints to reduce complexity
  - Dropout — also an example of adding constraints to reduce complexity

Reducing Network Size

- We can make the model (neural network) less complex by reducing the number of parameters
- Obviously enough, this is achieved by
  - Reducing the number of hidden layers, and/or
  - Reducing the number of neurons within the hidden layers

Example of reducing network size

```
In [ ]: def build_mnist_network(num_neurons):
    network = Sequential()
    network.add(Dense(num_neurons, activation="relu", input_shape=(28 * 28,)))
    network.add(Dense(10, activation="softmax"))
    network.compile(optimizer=RMSprop(lr=0.003), loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return network

In [ ]: network = build_mnist_network(512)
network.fit(mnist_x_train, mnist_y_train, epochs=5, batch_size=128, verbose=1, validation_split=0.25)

In [ ]: test_loss, test_acc = network.evaluate(mnist_x_test, mnist_y_test, verbose=0)
    test_acc

In [ ]: smaller_network = build_mnist_network(256)
smaller_network.fit(mnist_x_train, mnist_y_train, epochs=5, batch_size=128, verbose=1, validation_split=0.25)

In [ ]: test_loss, test_acc = smaller_network.evaluate(mnist_x_test, mnist_y_test, verbose=0)
    test_acc
```
Weight Regularization

- For linear regression, we used **regularization** to ensure that the coefficients $\beta$ took only small values by penalizing large values in the loss function
  - Lasso: we penalized by the $l_1$-norm (the sum of their absolute values)
  - Ridge: we penalized by the $l_2$-norm (the sum of their squares)
- A hyperparameter $\lambda$, called the ‘regularization parameter’ controlled the balance between fitting the data versus shrinking the parameters
- Weight Regularization in neural networks is the same idea, but applied to the weights in the layers of a network

```python
In [ ]: # Unregularized
def build_mnist_network():
    network = Sequential()
    network.add(Dense(256, activation="relu", input_shape=(28 * 28,)))
    network.add(Dense(10, activation="softmax"))
    network.compile(optimizer=RMSprop(lr=0.003), loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return network

In [ ]: network = build_mnist_network()
network.fit(mnist_x_train, mnist_y_train, epochs=5, batch_size=128, verbose=1, validation_split=0.25)

In [ ]: test_loss, test_acc = network.evaluate(mnist_x_test, mnist_y_test, verbose=0)
    test_acc

In [ ]: # Regularized
def build_regd_mnist_network():
    network = Sequential()
    network.add(Dense(256, activation="relu", kernel_regularizer=l2(0.001), input_shape=(28 * 28,)))
    network.add(Dense(10, activation="softmax", kernel_regularizer=l2(0.001)))
    network.compile(optimizer=RMSprop(lr=0.003), loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return network

In [ ]: regd_network = build_regd_mnist_network()
    regd_network.fit(mnist_x_train, mnist_y_train, epochs=5, batch_size=128, verbose=1, validation_split=0.25)

In [ ]: test_loss, test_acc = regd_network.evaluate(mnist_x_test, mnist_y_test, verbose=0)
    test_acc
```
Dropout

- Imagine we have a layer that uses dropout with dropout rate \( p \)
- Then, in a given step of the backprop algorithm, each neuron in the layer has probability \( p \) of being ignored — treated as if it were not there.

One way of doing dropout

- Suppose the dropout rate is \( p \), so the keep probability is \( (1 - p) \).
- Training: for any given mini-batch
  - In the forward propagation,
    - Decide which neurons will be dropped (chosen with probability \( p \)).
    - Set the activations of the dropped neurons to zero.
    - Multiply the activations of the kept neurons by \( (1 - p) \).
  - In the backpropagation, ignore the dropped out neurons.
- Note that different neurons will get dropped for each mini-batch.
- Testing: no change.
- But why did we multiply activations by \( (1 - p) \)?
  - In testing, for \( p = 0.5 \) a neuron in the next layer will receive input from on average twice as many neurons as it did in training.
  - The multiplication by \( (1 - p) \) compensates for this.

Why does dropout reduce overfitting?

- Consider a company whose employees were told to toss a coin every morning to decide whether to go to work or not.
  - The organization would need to become more resilient. It could not rely on any one employee to perform critical tasks: the expertise would need to be spread across many employees.
- Similarly, in dropout layers, neurons learn more robust features.
- Another way to think about it
  - Since a neuron can be present or absent, it’s like training on a different neural network at each step.
  - The final result is a bit like an ensemble of these many different virtual neural networks.
- However, it typically increases the number of epochs needed for convergence (roughly double when \( p = 0.5 \)).
Dropout in Keras

```python
In [ ]: # No dropout
def build_mnist_network():
    network = Sequential()
    network.add(Dense(256, activation="relu", input_shape=(28 * 28,)))
    network.add(Dense(10, activation="softmax"))
    network.compile(optimizer=RMSprop(lr=0.003), loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return network

In [ ]: network = build_mnist_network()
network.fit(mnist_x_train, mnist_y_train, epochs=5, batch_size=128, verbose=1, validation_split=0.25)

In [ ]: test_loss, test_acc = network.evaluate(mnist_x_test, mnist_y_test, verbose=0)
test_acc

In [ ]: # Dropout
def build_dropout_mnist_network():
    network = Sequential()
    network.add(Dropout(0.5, input_shape=(28 * 28,)))
    network.add(Dense(256, activation="relu"))
    network.add(Dropout(0.5))
    network.add(Dense(10, activation="softmax"))
    network.compile(optimizer=RMSprop(lr=0.003), loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return network

In [ ]: dropout_network = build_dropout_mnist_network()
dropout_network.fit(mnist_x_train, mnist_y_train, epochs=5, batch_size=128, verbose=1, validation_split=0.25)

In [ ]: test_loss, test_acc = dropout_network.evaluate(mnist_x_test, mnist_y_test, verbose=0)
test_acc
```

Conclusions

- Overfitting is a major problem but has many solutions
- There are lots of solutions in addition to the ones above:
  - Remember Batch Normalization has a regularizing effect
  - There are other techniques that we won’t cover (e.g. clipping)
  - There are the things we’ve mentioned in an earlier lecture, especially getting more data!