Chapter 15: Processing Sequences Using RNNs and CNNs

- Introduction
  - Main difficulties
    - Unstable gradients: alleviated through dropout or layer normalization
    - Limited short-term memory: extended using LSTM or GRU
  - CNNs and regular NNs can also handle short sequences
- Recurrent Neurons and Layers
  - Intro
    - RNN connections forward and backward
    - Each time step each neuron receives both input vector x_t and output vector from previous time step, y_{t-1}
    - Each recurrent neuron has two sets of weights: inputs & t-1 output
    - RNN output: $y_{(t)} = \phi(w_x^T x_{(t)} + w_y^T y_{(t-s\_} + b)$
    - Weights often concatenated vertically, x and y horizontally
  - Memory Cells
    - Since output of recurrent neuron (rn) is function of all inputs from previous steps → forms a kind of memory
    - Part of NN that preserves state across time steps is memory cell
    - Cell's state at time step t → h_t, **h**idden, is $f(h_{(t-1)}, x_t)$ or function of inputs and state at time step t-1
    - Basic cells: output = state, more complex not the same
  - Input and Output Sequences
    - Sequence-to-sequence network: feed previous stock prices, predict next price
    - Sequence-to-vector network: feed word sequence output sentiment score
    - Encoder-Decoder: S-2-v (encoder) → v-2-S (decoder), good for translation, better than word-by-word
- Training RNNs
  - Backpropagation through time
  - Forward pass through unrolled network
  - Output evaluated using cost function
  - Gradients flow backward through all outputs used by cost function
- Forecasting a Time series
  - Intro
    - Terms: univariate (one value per time step), multivariate (multi-values), imputation (postdicting, filling in missing values)
    - Note: input features represented as 3D array [batch_size, time_steps, dimensionality]
  - Baseline Metrics
    - Ways to compare model forecast
    - Naïve: predicts the last value
    - Fully connected network (FCN): add a flatten layer,
  - Implement a Simple RNN

        - Single layer, single neuron, no need to specify length of input sequences, first input dimension = None
        - SimpleRNN uses hyperbolic tangent, initial state (h_{init}) = 0,
            - Computes wt'd sum of values x_0, applies tanh → y_0, which is new state h_0
            - New state, h_0, passed to same rn with next input value x_1
            - Repeat until last time step
        - Note: recurrent layers only return final output, to get them to return one output per time step → return_sequences=True
        - Parameters:
            - Linear: one per input and per time step + bias
            - Simple RNN: one per input and per hidden state dimension + bias
        - Trend & seasonality
            - Time series models usually need to remove trend and seasonality through differencing, then train, then add back
            - RNNs generally don't have to remove
    - Deep RNNS
        - Stack multiple layers of cells on one another
        - Note: set return_sequences=True for all recurrent layers but last
        - Why you don't want last layer to be SimpleRNN
            - Want prior hidden states to carry over into final output
            - SimpleRNN uses tanh so predicted values need to be -1 to 1
            - Solution: replace with Dense layer
    - Forecasting Several Time Steps Ahead
        - Predict values several steps ahead → change target to desired time step
        - Predict sequences of steps ahead use first prediction as input for next
            - Errors ↑ as number of time steps in sequence↑
        - Predict next sequence of steps at every time step → sequence-to-vector RNN becomes sequence-to-sequence RNN
            - More error gradients flowing through model
            - Flow through time, from output of each time step
            - Stabilize and speed up training [Why speed up?]
        - Causal model: only sees last time steps, but targets and values will overlap
        - To create sequence-to-sequence model → return_sequences=True for all layers
            - Must apply Dense layer at every time step → use TimeDistributed layer
            - Turns [batch_size, time_step, dim] into [batch_size * time_step, dim] and back
        - All outputs needed during training, only last output at last time step useful for predictions and evaluation
        - Note: error bars useful to include in predictions. Use MCDropout within each cell. After training forecast many times and compute mu and sigma of predictions

- Handling Long Sequences
  - Intro
    - Training RNN on long sequences means many time steps, means deep NN → unstable gradients problem
    - RNN will forget earlier inputs in long sequence
  - Fighting the Unstable Gradients Problem
    - Common solutions: good parameter initialization, faster optimizers, dropout
    - Non-saturating activation not as successful
      - Early wts used later → may increase or explode
    - Other methods: reduce learning rate, using saturating activation function, gradient clipping
    - Batch normalization not as efficient: can't use between time steps, only between layers, improves only slightly if applied initially
      - Reason: BN layer used at each time step with same parameters, regardless of actual sale and offset of inputs and hidden state
    - Layer Normalization works better: normalizes across feature dimensions
      - Can compute stats on fly at each time step independently of each instance
      - Behaves same during training and testing, does not need to use EMA to estimate feature stats across instances
      - LN learns a scale and offset parameter for each input, used after linear combination of inputs and hidden states
    - All Keras layers (except .RNN) have dropout and recurrent_dropout hyperparameters, apply to inputs or to hidden state
  - Tacking the Short-Term Memory Problem
    - Into
      - Information lost at each step when traversing an RNN
      - Solution: cells with long-term memory
    - LSTM cells
      - Apply with keras.layers.LSTM or with layers.RNN(keras.layers.LSTMCell())
      - Main idea: learn what to store in long term state, what to throw away, and what to read from it
      - Each time step: some memories dropped, some added, long-term state copied passed, activated, and filtered to output → short term state (h_t)
      - LSTM learns to recognize important input, store it in long-term sate, preserve for as long as needed, and then extract when needed
      - Peephole connections: look at long-term state, only in experimental version
    - GRU cells
      - Gated Recurrent Unit: simplified version of LSTM, performs just as well
      - State vectors merged into h, single gate controller, no output gate

            - New gate controller controls which part of previous state shown to main layer
    - Issues
        - LSTM and GRUs still have problem handling sequences longer than 100 steps
    - Other structures
        - Using 1D convolutional layers to process sequences
            - 1D convolutional layer slides several kernels across a sequence producing 1D feature map per kernel
        - Wavenet
            - Stacks 1D convolutional layers, doubling dilation rate at every layer
            - Dilation rate: how spread apart neuron’s inputs are
            - First conv layer sees 2 time steps, next 4, next 8, up to 512