Tensorflow in Practice Specialization

Computer Vision

Representation sparsity in Deep Learning: is a key feature of deep learning. These representations carry increasingly less information about the original pixels of the image, but increasingly refined information about the class of the image. You can think of a convolutional network (or a deep network in general) as an information distillation pipeline.

Overfitting happens when a model exposed to too few examples learns patterns that do not generalize to new data, i.e. when the model starts using irrelevant features for making predictions.

On **transfer learning** we usually can unlock also the last convolutional layers in case the network was trained on some other concepts. So, we will reuse only the basic low-level features from the first convolutional layers. However, if the model was trained on face images and we want to fine tune it on new faces we might need to unlock only the last dense layers. **When the concept changes drastically we might need to retrain some convolutional layers also.**

Techniques to use in computer vision for improving performance: dropout, augmentation, batch normalization, regularization, transfer learning, more data.

NLP

Text encoding in NLP: We build a word vocabulary out of a large text corpus and associate each word with a unique number. Depending on the task we might need to remove whitespace, punctuation, lowercase words, remove URLs/emails/numbers, etc. We usually do not use all the possible words to transform the train/test data but only the most N common words. So, we have also an <UNKNOWN> word token for unknown train/test words. We use a text tokenizer to encode a list of text sentences and a padding/truncating mechanism so that all vector encodings can have the same size (by pre/post truncating and/or padding sentences with zeros).

Why we need a rich corpus of text? We need a large corpus of text to create a rich word vocabulary. Otherwise, test data might be encoded as vectors with no meaning. eg: "My neighbor loves my dog" could be encoded as "my my dog" if only "my" and "dog" exist in the vocabulary.

Learn word semantics to improve classification performance: The encoded and padded text vectors can be used with embedding learning algorithms (Word2Vec, GloVe) to learn semantics and then word embeddings can be used in machine learning algorithms (e.g. classification) for better performance.

3D projection of normalized word embeddings learned for IMDB review binary classification: So, we'll see something that looks like a globe and one of the poles will have clustered words for the positive reviews. Also, the opposite pole of the glove will have clustered words for the negative reviews. It really helps us to start to see the semantics behind these words.

The labels drive the learning algorithm to discover similar encodings to similar words.

Example:

The movie was boring	0
The movie was dull	0

boring-dull will have similar encodings

The movie was interesting	1
The movie was amazing	1

interesting-amazing will have similar encodings

GlobalAveragePooling1D() vs Flatten(): Instead of Flatten() after Embedding() a GlobalAveragePooling1D() layer can be used (it calculates the mean vector of all words in the sentence – the mean word embedding), it is simpler and faster than Flatten() because it creates a smaller output shape.

Subwords difficulties and the need of sequence models: It is hard to pull semantics and meaning from characters or subwords when using embeddings. Subwords haven't very good sensical meanings and only when we put them in sequence meaning arises, that is why we need to use sequence models.

State-of-the-art NLP: Great results in NLP arise when we use sequence models (the position of a word matters same as its existence) and word embeddings (the semantics of words matters a lot).

The vanishing gradients problem in classic RNN leads to inability to capture short-long term dependencies and the solution LSTM/GRU models provide: Context in sequence/time-series data can be preserved from timestep to timestep but can also be lost in longer sequences (due to vanishing gradients problem) with simple RNN models. However, this problem can be solved using LSTM (which maintain a cell state appropriately that it works as a conveyor belt from which we can add, remove, update information from past to present and future timesteps). LSTM's cell state is an extra layer/pipeline of context information that can be passed along the sequence from past timesteps to future timesteps. The cell state can be bidirectional.

In NLP, when overfitting, the loss might increases gradually: When the validation loss increases there is an overfit and we need to tune the hyperparameters to flatten it out.

Smoothier results with stacked LSTMs: Stacked LSTMs produce smoothier increasing of accuracy and smoothier decreasing of loss.

1D convolutions with text data: Learn features from word embeddings of N sequential words.

Combination of techniques in sequence models: For example in NLP, we might use Embedding, Conv1D, RNN (GRU/LSTM), Dense layers with Bidirectional/Stacked architecture.

Time-series

Sequence data

- text
- time-series

Time-series

- Ordered sequence of values usually equally spaced over time (per hour/minute/second/day/month/year)
- Everything that has a time factor can been seen as time-series data! (e.g. video, music, speech audio)

Univariate and Multivariate signals

- Univariate: A single value over time
- Multivariate: A vector of values over time (can be seen as multiple univariate time-series)

  Multivariate time-series can be handled as separate univariate time-series. However, the real value comes when we handle them as a multivariate time-series signal.

Examples of time-series data

- Stock price (univariate)
- Weather temperature (univariate)
- Moore's law (univariate)
- EEG signal (multivariate since multiple electrodes are used)
- ECG signal (multivariate since multiple electrodes are used)
- Deaths-births in japan (multivariate)
- Global temperature-co2 concentration (multivariate)
- Weather forecast (multivariate)
- Sunspot activity (univariate)

Types of things ML can do with time-series

- Predict future value(s): forecasting
- Predict past value(s) or fill missing values in time-series:
  - Predict how past data would have been before collecting the data we already have
  - Find timesteps with no values and fill them appropriately (imputation)
- Anomaly detection (detect spikes in the signal, e.g. large #requests from a DoS attack)
- Detect patterns in time-series (e.g. heart pulse, wave patterns of spoken words)

Common patterns/properties of time-series

Seasonality means that there are repetitive data patterns in predictable periods of time

- Weather information (e.g. for temperature: usually June is hot and December is cold)
- ECG (PQRST pattern is repeated)
- High visit activity in Stack Overflow on Mon-Fri (work days) and low visit activity on Sat-Sun (weekend days)
- Shopping sites with peek on weekends
- Sport sites with peek at specific times throughout the year (championships, etc)

Trend means a specific direction (linear, exponential, n^{th} order polynomial) of the signal facing upward/downward

- Moore's law
- BTC

- Weather temperature (small increasing due to climate change)

Noise is a random factor that is hard to predict in a time-series signal

- Totally random time-series are not predictable at all since each value has equal probability to appear at each timestep (e.g. white noise)
- Small random noise in data should not be a problem for a ML model to detect patterns in time-series (e.g. a PQRST pattern in ECG might not always be exactly the same and there might be minor noise in the signal)
- Random big catastrophic events cannot be predicted easily, especially random factors that change the statistics of the signal

Autocorrelation

- Correlation of a signal with a delayed copy of itself (signals may not have seasonality or trend but that doesn't mean that they are totally random). Various parts of the signals might be linearly related with other parts of the signal.
- A correlation coefficient can be calculated for various delays/lags and we can plot them on graph (ACF, auto correlation function) to see how much related are past values to future values
- Can be used to see if the signal is generated randomly or not
- ACF(0)=1 (a time-series with no delay is 100% correlated with itself)
- When the ACF is close to zero for all lags it means that the time-series is random

Stationarity

- Non-stationary
  - The statistics of the signal change over time. Usually the behavior of non-stationary signals change drastically over time (e.g. seasonality/trend might stop/change after a big unpredictable event). Example: in stock price prediction a big changing event (financial crisis, big scandal, etc) can change the properties of the signal.
- Stationary:
  - The statistics of the signal do not change over time

Baseline

The signal has a DC constant added to it

Easy to predict in time-series: ML systems when trained on time-series with seasonality, trend, stationarity, autocorrelation and small noise can learn to spot patterns in data and predict them appropriately in the future (forecast the learned pattern).

Difficult to predict in time-series: Random events that change the properties of the signal (seasonality, trend, stationarity) may exist and they are difficult to predict.

Forecasting approaches

All methods cannot predict big events that change the signal drastically.

- Statistics / Numerical-analysis naive methods
  - **Naive Forecasting** copies the last value as the future value. The method does not try to handle noise in data. When the signal has seasonality/trend it cannot predict it correctly. Usually, when the signal changes slowly and we want just to predict the next moment this method can be an option.
  - **Moving Average** uses a moving window of fixed size and calculates the average of the window data as the next future value. This method removes noise but cannot predict correctly trend/seasonality. One solution is **differencing technique** which **1)** removes seasonality/trend from the signal using the **seasonality period** and f(t) – f(t-

365), **2)** performs the moving average on the difference signal which removes some of the noise and **3)** adds back the seasonality/trend by adding value f(t-365). For example, if we want to predict the value of tomorrow we use the corresponding day of previous year. However, when we apply step 3 the resulting signal despite having trend/seasonality has also some noise from the past values. One way to avoid this in step 3 is to add back the **smoothed version of the past values using also moving average**. Past values can be smoothed using also a **centered window which leads to more accurate** results! However, for smoothing present values we need to use trailing window since we do not know the future values.

- DNN (FC FF layers) trained on shuffled (to break sequence bias) windowed data. In this setup the predicted value is impacted mostly from the previous N values from the window. It is very hard the predicted value to impacted also from values from the very past in the time-series. That is why we use RNN/sequence models.
- RNN models (state-of-the-art approach). In this approach we can overcome the case of DNN approach where the predicted value is impacted mostly from the previous N values from the window. With RNN models we can carry information and context from the past windows of data. Examples of windowed data passed to an RNN does not need to be sequentially in time (can also be shuffled to break bias of sequence).
- Combinations of RNN/DNN/Conv1D

How to evaluate (measure performance) a time-series model?

First step: Data splitting

Fixed partitioning

1. We partition the time-series to three SEQUENTIAL partitions: training/validation/test (because we need to train the model on past data and evaluate/test it in future data). In case the data has seasonality each partition needs to contain whole seasonality periods.
2. Train model on train data and evaluate it to validation data
3. Tune the model hyperparameters to reduce the validation error
4. With best hyperparameters found train the model on train&validation data and check the model's performance on test data.

If the model generalizes well on test data (more or less we have similar validation-test errors) we can train the model on train&validation&test data with the best hyperparameters. **This is unusual in non time-series prediction models. So why we do it?** Test data is the closest data to the current point in time where we need to forecast a future value (test data are the strongest signal in determining future values and the model needs to be trained with to be optimal). **For this reason we can ignore the test set from the very beginning. We can use only a training and a validation set. After hyperparams tuning we can train the model on both training&validation partitions.**

Roll Forward Cross Validation

Same with Normal Cross Validation but we need to assure that the folds at each iteration are sequential in time and that the test fold is always after the train folds. Otherwise, it has no meaning (e.g: train in future data to predict the past?).

Second step: Metrics

We can use the following metrics to compute the performance on validation and test data

- MeanSquaredError (MSE) : Penalizes strongly the large errors
- RootMeanSquaredError (RMSE) : By taking the squared root of the MSE the total error will have the same scale of the original errors
- MeanAbsoluteError (MAE) : Does not penalizes strongly the large errors

Depending on the task we may prefer (R)MSE vs MAE. For example: If large errors are potentially dangerous and cost more than small errors we should use MSE.

ML and time-series

First we need to create a supervised dataset of windowed data. We can create a dataset of examples $\{x^i, y^i\}$ where x^i will be some N continuous values from the time-series and y^i the label. The label can be either a class for classification problems or the next value N+1 in the time-series to predict for regression problems. By sliding a window with specific size we can create various examples. We can use the $\{x^i, y^i\}$ examples to train/validate/test a ML model. The window size can be seen as a hyperparameter to be tuned.

Learning rate tuning in ML models

We can tune the learning rate as any other hyperparameter by searching for the appropriate value that leads to a better generalization on test data. However, a **LearningRateScheduler** can be used which it tries a different learning rate value at each epoch. It starts with an initial value and gradually reduces it to a minimum using a formula. We can plot the loss of each epoch with the corresponding learning rate and we can identify the learning rate that gives small loss. Last, we can retrain the whole model with that best learning rate for more epochs.

So, for a small number of epochs we train with a learning rate scheduler and check to see for which learning rate value we have good stable loss value. Then we retrain the model with the best learning rate for more epochs. This can saves us some time of tuning the learning rate exhaustively.

Vanilla DNN / RNN for time-series

Input/Output shapes:

1. The input/output of a vanilla DNN network is 2D (input: [batch size, #dimensions], output: [batch size, #dimensions]).

2. The input/output of an RNN network is 3D (input: [batch size, #time steps, #data dimensionality], output: [batch size, #time steps, #nodes of internal state]).

3. The input/output to an RNN layer for a given time step is 2D (batch size, #nodes of internal state).

Batch size: #of examples to use (e.g. #windows in a time-series, #sentences in text)

Time steps: #of time steps (e.g. #of words in a sentence, #of values in a window of data)

Data dimensionality (e.g. #of dimensions in the time-series, #embedding size for a word)

We have said that the input in RNNs is 3D. However, we specify the input shape of the first RNN layer as: input_shape = [None, 1]. This is because the batch size is never specified. So, the None is referred to the time steps (the RNN can handle sequences with various lengths) and 1 because at each time step we have only one value (e.g. univariate time-series signal).

Lambda Layers

Lambda layers can be used in a NN for pre-processing or post-processing of data. For example, can be used for scaling the output of a regression model (it helps training in regression problems) or reshaping input tensors (e.g. convert 2D matrix [batch size, time steps] to 3D tensor [batch size, time steps, 1] when using univariate time-series and RNN - same as in grayscale images where [width, height] is transformed to [width, height, 1] because CNN expects 3D input). Instead of adding code for performing these processing steps we can add code in the NN as lambda layers to perform it.

Sequence2Sequence (Tx = Ty), Sequence2Vector

An RNN layer can generate an output at each time step (sequence2sequence) or only in the last time step (sequence2vector). The default behavior in Tensorflow's Keras is to return the output value from the last time step. To use all the outputs we can use return_sequences=True. This is necessary when using stacked RNNs and we need to pass the outputs of all time steps to the next layer. Also when we use a Dense layer as the last layer of the NN and return_sequences=True then the same dense layer is used for each time step in the output.

Huber loss is more robust to outliers than MSE and usually is used in regression problems with data that are noisy.

Convolutions in time-series

Convolutions can be used before LSTM/RNN/GRU to extract useful features from the time-series signal. In a univariate/multivariate time-series [batch size, window size, dimensions of time-series] we can use a Conv1D layer with k filters and stride 1 and create actually a new tensor of size [batch size, width of convolved window, #filters]. So at each time step of the convolved window the RNN/GRU/LSTM gets as input a tensor of features extracted from the various filters of size [batch size, #filters].

Batch size: In mini-batch gradient descent small batch sizes introduce random noise in the direction of finding the minimum and thus loss does not decreasing steadily. Like stochastic gradient descent there might be many oscillations in the loss. In RNN networks we also need to tune the batch size.

Keywords: Bidirectional, Stacked, Conv1D, LSTM/GRU, LearningRateScheduler, Huber loss, Window size/Batch size hyperparameters, Shuffle windowed examples (Sequence bias breaking), 3D input of RNN, scale output with lambda layer.

Sunspot Activity: Monthly mean total sunspots from 1749 to 2018. The time-series has a seasonality (~11-year period). This time-series has seasonality, the peeks are not always the same, there is not trend, and there is some noise in the data.