# A Guide to Deep Learning in PyTorch

March 10, 2018

Bayu Aldi Yansyah

bay@machinelearning.id

## Contents



## 1 Overview

Dokumen ini berisi panduan untuk mengimplementasikan model Deep Learning menggunakan PyTorch.

Panduan ini dibuat dengan asumsi bahwa:

1. Pembaca sudah terbiasa menggunakan bahasa pemograman Python
2. Pembaca sudah familiar dengan Deep Learning

Saya barusaha untuk membuat panduan ini sepraktikal mungkin, lebih banyak memberi contoh kodenya dan hanya menjelaskan teori yang terkait saja.

Panduan ini terbagi menjadi 2 bagian utama:

- **Bagian 1: Pengenalan PyTorch**. Pada bagian ini akan diperkenalkan PyTorch, konsep-konsep dasar beserta kodenya.
- **Bagian 2: Implementasi & Deployment model Deep Learning**. Pada bagian ini kita akan membuat model untuk *Sentiment Classification* . Dataset yang kita pakai adalah Large Movie Review Dataset v1.0. Nanti kita akan menggunakan teknik Unupervised Representation Learning dari paper Deconvolutional Paragraph Representation Learning oleh Zhang et al (2017) dan simple MLP untuk classifiernya.

Ini kebetulan nanti penerapan Deep Learning nya ke Natural Language Processing, tapi nanti konsep penerapan model paper ke dalam kode harusnya bisa diterapkan untuk hal yang lain seperti Image Recognition dan lain-lain.

Saya nanti akan mulai dari konsep-konsep dasar PyTorch seperti `torch.Tensor`, operasi-operasi yang ada di `torch.Tensor`, kemudian review beberapa teori deep learning, implementasi model deep learning hingga cara menjalankan model deep learning untuk keperluan production. Panduan ini saya buat untuk teman-teman yang ingin mengimplementasikan model-model deep learning untuk keperluan pekerjaan ataupun penelitian.

Sebagai catatan, panduan ini yang di bahas pada acara Machine Learnind ID Meetup #3 di Surabaya. Durasi acaranya sekitar 120 menit. Bagi peserta acara atau pembaca yang ingin mengikuti tutorial ini harus menginstall PyTorch dulu. Jika ada saran, koreksi atau kritik pembaca bisa menyampaikannya melalui twitter atau email..

## 2 An Overview of PyTorch

PyTorch adalah versi Python dari Torch, versi aslinya di tulis dengan bahasa pemograman Lua-JIT. Package PyTorch digunakan untuk komputasi numerik, ala-ala NumPy, tapi sudah support untuk melakukan operasi di GPU. PyTorch juga dilengkapi fungsi-fungsi bawaan yang mempermudah untuk menerapkan model-model deep learning. Package ini bisa digunakan untuk implementasi model-model deep learning atau conventional machine learning untuk keperluan kerjaan ataupun penelitian kalian.

Ada beberapa konsep dasar yang harus dipahami di Pytorch yaitu (1) `torch.Tensor` dan (2) operasi-operasinya.

```
In [1]: import torch
        import numpy as np
```

### 2.1 torch.Tensor

`torch.Tensor` adalah sebuah Python class yang merepresentasikan **n-dimensional array**. Setiap elemen di dalam `torch.Tensor` hanya memiliki satu tipe data yang sama. `torch.Tensor` mirip dengan NumPy n-dimensional array (`numpy.ndarray`), hanya saja `torch.Tensor` sudah mendukung penyimpanan data di memory GPU dan `torch.Tensor` mempunyai banyak fungsi-fungsi pembantu untuk mempermudah melakukan komputasi numerik di CPU maupun GPU.

Untuk penjelasan tentang **n-dimensional array** lebih jelas kalau melalui contoh seperti dibawah berikut:

Ini adalah array berdimmensi satu, disebut dengan **Vector**.

```
In [8]: np.array([1, 1, 3, 4])

        # TODO: shape & indexing

Out[8]: array([1, 1, 3, 4])
```

Kemudian array berdimensi 2 disebut dengan **Matrix**.

```
In [9]: np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8]])

        # TODO: shape & indexing

Out[9]: array([[1, 2, 3, 4],
               [5, 6, 7, 8]])
```

Untuk yang berdimensi 3 atau lebih seperti ini:

```
In [10]: np.array([
             [[1, 2, 3, 4], [5, 6, 7, 8]],
             [[1, 2, 3, 4], [5, 6, 7, 8]],
         ])
         # TODO: shape & indexing

Out[10]: array([[[1, 2, 3, 4],
                 [5, 6, 7, 8]],

                [[1, 2, 3, 4],
                 [5, 6, 7, 8]]])
```

Nah karena `torch.Tensor` adalah `n-dimensional` array, kita bisa merepresentasikan konsep-konsep matematik seperti vektor (**1-dimensional array**) dan matriks (**2-dimensional array**). Untuk penerapannya di model Deep Learning, `torch.Tensor` digunakan untuk merepresentasikan input datan, matrix bobot dan bias tiap lapisan yang akan di optimaliasi dan output datanya. Maka dari itu `torch.Tensor` adalah dasar pertama yang harus dipahami.

Semua elemen dari `torch.Tensor` mempunyai satu tipe data yang sama, untuk versi `0.3.1` tipe data yang didukung adalah:

- 32-bit floating point
- 64-bit floating point
- 16-bit floating point
- 8-bit integer (unsigned)
- 8-bit integer (signed)
- 16-bit integer (signed)
- 32-bit integer (signed)

- 64-bit integer (signed)

Tidak perlu cemas jika belum familiar dengan semua tipe datanya. Tipe data yang sering digunakan hanya dua yaitu `float` yang merepresentasikan bilangan rasional seperti `1.0`, `3.14`, `0.002` dan tipe data `integer` yang merepresentasikan bilangan bulat seperti `1`, `2` dan `3`. Untuk arsitekturnya `32-bit` dan `64-bit` nanti yang menentukan berapa maksimal dan minimal nilainya. Kemudian beda `signed` dan `unsigned` adalah, untuk `unsigned` semua nilainya positif sedangkan yang `signed` ada negatifnya. Untuk penggunaan nanti tergantung kebutuha, tapi yg lebih sering digunakan adalah `64-bit floating point` dan `64-bit integer (signed)`.

Selanjutnya kita akan membahas operasi-operasi yang ada pada `torch.Tensor`.

## 2.2 torch.Tensor Operations

Tensor operations di bagi menjadi 6 kelompok: Constructor, Indexing, Slicing, Joining dan Mutating ops. Kita akan bahas beberapa saja yang sering digunakan.

### 2.2.1 Contructor Ops

Constructor berguna untuk menginialisasi tensor baru. Beberapa constructor ops adalah `torch.eye`, `torch.from_numpy` dan `torch.zeros`.

Yang pertama constructor `torch.eye`, constructor ini berguna untuk membuat matrix identitas seperti berikut:

```
In [19]: # Inialisasi matrix 10x10
         torch.eye(10)
```

```
Out[19]:
          1     0     0     0     0     0     0     0     0     0
          0     1     0     0     0     0     0     0     0     0
          0     0     1     0     0     0     0     0     0     0
          0     0     0     1     0     0     0     0     0     0
          0     0     0     0     1     0     0     0     0     0
          0     0     0     0     0     1     0     0     0     0
          0     0     0     0     0     0     1     0     0     0
          0     0     0     0     0     0     0     1     0     0
          0     0     0     0     0     0     0     0     1     0
          0     0     0     0     0     0     0     0     0     1
         [torch.FloatTensor of size 10x10]
```

```
In [16]: # Inialisasi matrix 10x5
         torch.eye(10, 5)
```

```
Out[16]:
          1     0     0     0     0
          0     1     0     0     0
          0     0     1     0     0
          0     0     0     1     0
          0     0     0     0     1
          0     0     0     0     0
```

```
    0     0     0     0     0
    0     0     0     0     0
    0     0     0     0     0
    0     0     0     0     0
[torch.FloatTensor of size 10x5]
```

Constructor kedua adalah `torch.from_numpy`, constructor ini yang paling sering digunakan. Untuk convert data dari `np.array` ke `torch.Tensor`. Contoh penggunaannya seperti berikut:

```
In [18]: # Inialisasi vector 3x1
         a = np.array([12, 12, 0])
         b = torch.from_numpy(a).type(torch.IntTensor)
         b

Out[18]:
 12
 12
  0
[torch.IntTensor of size 3]
```

Constructor ops yang sering digunakan selanjutnya adalah `torch.zeros`. Constructor ini menginialisasi `torch.Tensor` dengan nilai `0` semua. Contoh penggunaanya seperti ini:

```
In [17]: # Inialisasi matrix 5x5 yang nilainya 0 semua:
         torch.zeros(5, 5)

Out[17]:
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
[torch.FloatTensor of size 5x5]
```

3 constructor diatas yang sering digunakan. Sebenernya constructor masih banyak lagi seperti `torch.linspace`, `torch.logspace`, `torch.ones` dan lain-lain. Kalian bisa menggunakannya sesuai kebutuhan.

### 2.2.2 Indexing, Slicing, Joining dan Mutating Ops

Dari ops indexing, slicing, joining dan mutating ops yang ada, yang paling sering digunakan adalah `squeeze()`, `unsqueeze()`.

`squeeze()` berguna untuk menghilangkan dimensi yang nilainya `1`. Misal kita punya n-dimensional array dengan ukuran `1 x 4 x 3`, untuk merubahnya ke ukuran `4 x 3` tinggal di squeeze aja.

```
In [37]: a = torch.rand(1, 4, 3)
         a
```

```
Out[37]:
(0 ,.,.) =
  0.6446  0.0436  0.5237
  0.6474  0.5786  0.7418
  0.2897  0.9541  0.5214
  0.2414  0.0673  0.7289
[torch.FloatTensor of size 1x4x3]

In [44]: a = a.squeeze()
         a

Out[44]:
 0.6446  0.0436  0.5237
 0.6474  0.5786  0.7418
 0.2897  0.9541  0.5214
 0.2414  0.0673  0.7289
[torch.FloatTensor of size 4x3]
```

Begitu juga sebaliknya kita bisa `unsqueeze()` untuk menambahkan dimensi:

```
In [46]: a.unsqueeze(0)

Out[46]:
(0 ,.,.) =
  0.6446  0.0436  0.5237
  0.6474  0.5786  0.7418
  0.2897  0.9541  0.5214
  0.2414  0.0673  0.7289
[torch.FloatTensor of size 1x4x3]
```

### 2.3 torch.nn

`torch.nn` adalah building block PyTorch yang kedua setelah `torch.Tensor`. Di module ini terdapat class-class arsitektur Neural Network generik yang berguna untuk menyusun arsitektur model neural network. Nanti contohnya ada di sesi tutorial.

### 2.4 torch.autograd

Kemudian komponen terakhir yang menjadi basis building block untuk membangun model neural network dengan PyTorch adalah `torch.autograd`, disini terdapat class `Variable` yang berguna sebagai autograd tracker. Autograd adalah mekanisme untuk menghitung gradient otomatis bawaan dari PyTorch. Nanti contohnya akan ada pada sesi tutorial bawah.

### 2.5 Utilities

Module PyTorch lain yang berfungsi sebagai utilitas yang mempermudah pemrosesan data. Misalnya untuk membaca data dan melakukan operasi batching dan shufeling terdapat module `torch.utils.data`. Kemudian untuk menyimpan dan meload snapshot terdapat tool `torch.save` dan `torch.load`.

# 3 Tutorial: Unsupervised Representation Learning with Deconvolutional Autoencoder

Dokumen ini berisi penjelasan tentang paper "Deconvolutional Paragraph Representation Learning" oleh Zhang et al (2017) beserta panduan untuk mengimplementasikan modelnya menggunakan PyTorch.

Mengekstrak fitur dari sebuah teks merupakan langkah awal dari task-task Natural Language Processing lainnya seperti Sentiment analysis, Machine translation, Dialogue systems, Text classification dan Text summarization. Paper yang akan kita bahas berisi tentang autoencoding framework yang bernama *Convolutional-Deconvolutional* (CNN-DCNN) yang bertujuan untuk "belajar" mengekstrak fitur atau merepresentasikan sebuah paragraph menjadi sebuah vektor dari sebuah raw data. Output dari CNN-DCNN inilah yang nantinya akan jadi input model-model lain untuk task-task NLP tersebut.

Kita akan mulai dari review sedikit tentang apa itu teknik *Feature/Representation Learning*, dilanjutkan dengan penjelasan bagaimana pendekatan teknik *Repesentation Learning* untuk teks, kemudian apa kelebihan CNN-DCNN dari metode yang sudah ada, implementasi CNN-DCNN dan terakhir penerapannya untuk task Sentiment Classification.

Ada beberapa konsep seperti operasi konvolusi, fungsi aktivasi ReLU dan Batch Normalization yang tidak saya jelaskan detail di dokumen ini karena perlu sesi sendiri untuk menjelaskan hal-hal tersebut dari awal.

## 3.1 Representation Learning

Di bagian ini akan kita review sedikit tentang apa itu teknik *Feature/Rerepesentation Learning*.

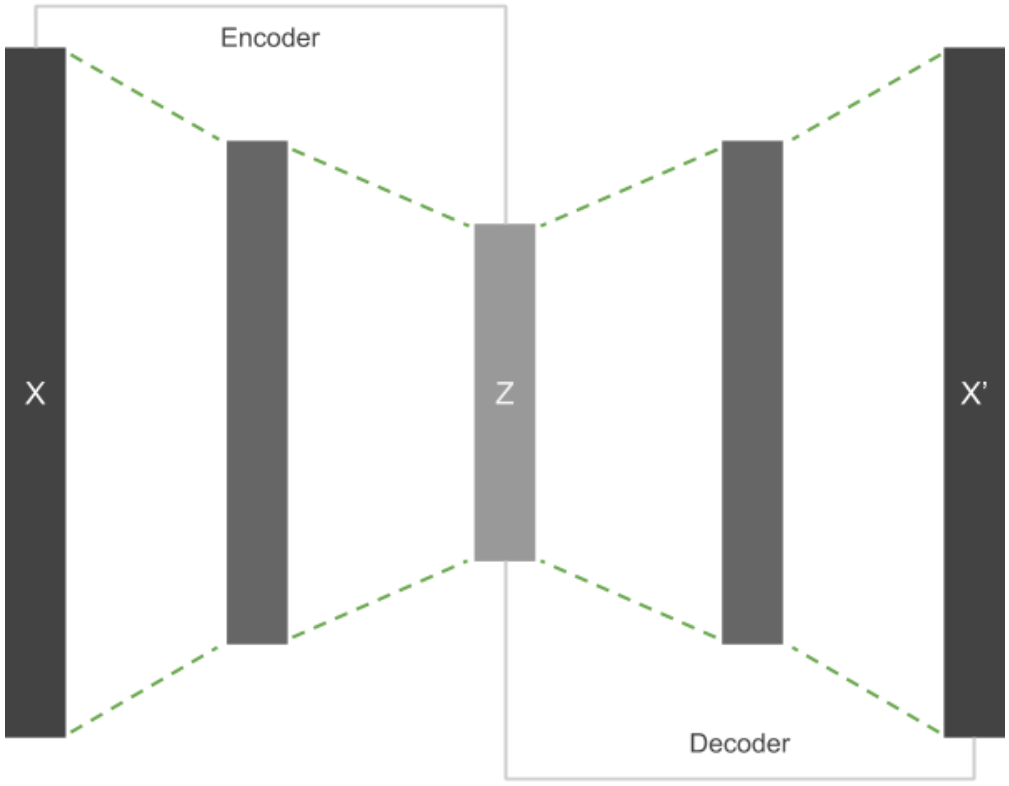
Dalam Machine Learning, Feature learning atau Representation Learning adalah teknik yang berguna untuk mencari sebuah fitur atau representasi sebuah raw data. Contohnya ketika ingin melatih model untuk klasifikasi teks, langkah pertama adalah mengekstrak fitur dari teks tersebut. Biasanya kita secara manual mengekstrak fitur dari teks menggunakan Bag-of-Words atau TF-IDF setelah itu menyeleksi fitur-fiturnya, kegiatan ini sering disebut *Feature Engineering*. Untuk kasus *Feature Learning* ini modelnya lah yang belajar untuk mengekstrak fitur teksnya dari data teks itu sendiri. Salah satu perbedaan dengan Bag-of-Words atau TF-IDF adalah representasi vektor yang dihasilkan oleh metode Feature Learning tidak bergantung pada jumlah vocabulary, melainkan fixed-size vector.

Sebelumnya sedikit catatan: Nanti mungkin saya akan menulis kata `{"fitur", "representasi vektor kalimat"}` dan `{"word vector", "word embedding", "vektor kata", "latent representation vector"}` secara bergantian, kata dalam satu set tersebut artinya sama.

Representation Learning sendiri ada 2 macam yaitu Supervised dan Unsupervised, hal ini yang akan kita bahas pada sesi dibawah.

### 3.1.1 Supervised & Unsupervised Representation Learning

Representaion Learning sendiri ada 2 macam: Supervised dan Unsupervised. Supervised representation learning belajar mengekstrak fitur dari data yang sudah diberi label. Sedangkan Unsupervised repsresentation learning belajar mengekstrak fitur dari data yang tidak ada labelnya. Disini kita akan fokus pada Unsupervised Representation Learning terutama metode Autoencoder.




Gambar 2.1 - Autoencoder

### 3.1.2 Autoencoder

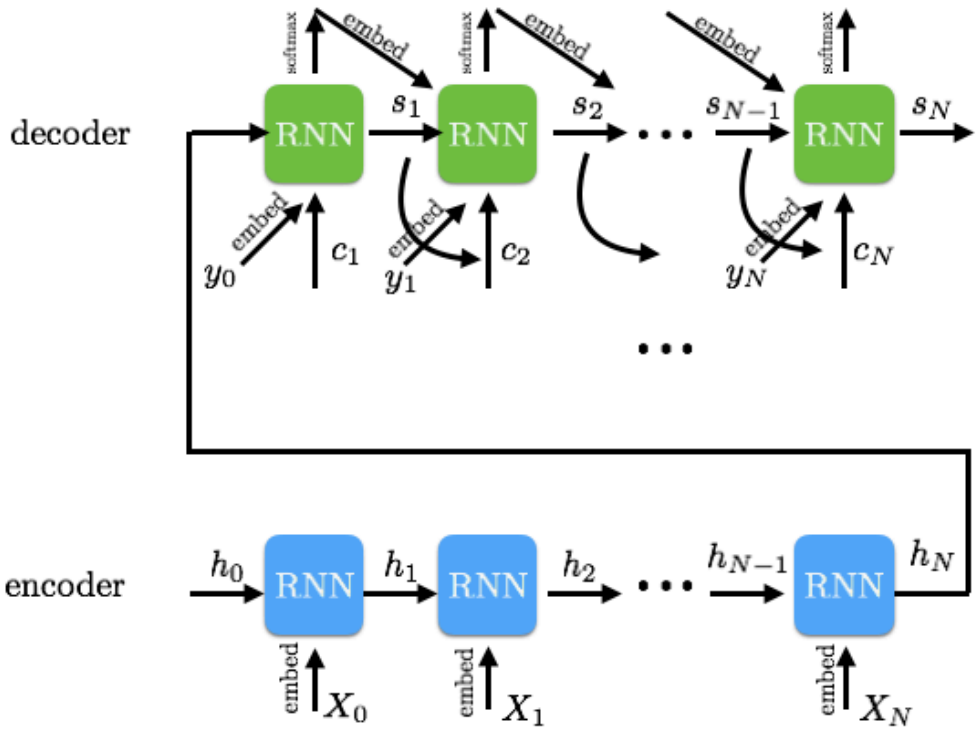
Salah satu metode Unsupervised representation learning adalah Autoencoder, dimana terdapat sebuah konsep *encoder* dan *decoder* pada arsitektur deep learningnya. Seperti pada Gambar 2.1, konsepnya adalah *encoder* menggunakan raw data (misal: teks atau gambar) sebagai input ($X$) dan menghasilkan sebuah fitur atau representasi sebagai output ($Z$). Lalu *decoder* menggunakan fitur yang berhasil di ekstrak oleh *encoder* sebagai input ($Z$) dan mencoba untuk merekonstruksi input raw data yang asli sebagai output ($X'$).

Untuk data kalimat, langkah-langkah autoencodingnya adalah pertama *encoder* menggunakan urutan vektor kata (*embedding*) dari kalimatnya dan menghasilkan representasi vektor dari kalimatnya, kemudian *decoder* menggunakan representasi vektor tersebut untuk mencoba menyusun kembali kalimat yang asli.

Untuk Autoencoder teks, arsitektur yang sering digunakan adalah Recurrent Neural Networks (RNNs) terutama varian Long Short-Term Memory (LSTM) yang terlihat seperti pada Gambar 2.2. Hanya saja untuk *decoder* yang RNN-based punya masalah kalau kalimatnya semakin panjang performa rekonstruksi kalimatnya semakin menurun.

Untuk itu Zhang et al propose CNN-DCNN yang tidak terpengaruh dengan panjangnya kalimat, lebih simpel, komputasi lebih efisien dan terbukti setelah di lakukan evaluasi secara kuantitatif untuk Text Classification dan Text Summarization hasilnya lebih bagus.

CNN-DCNN akan kita bahas lebih detail pada sesi dibawah.




Gambar 2.2 - Autoencoder RNN

| Model | BLEU | ROUGE-1 | ROUGE-2 |
|---|---|---|---|
| LSTM-LSTM [47] | 24.1 | 57.1 | 30.2 |
| Hier. LSTM-LSTM [47] | 26.7 | 59.0 | 33.0 |
| Hier. + att. LSTM-LSTM [47] | 28.5 | 62.4 | 35.5 |
| CNN-LSTM | 18.3 | 56.6 | 28.2 |
| CNN-DCNN | **94.2** | **97.0** | **94.2** |

Table 2: Reconstruction evaluation results on the Hotel Reviews Dataset.

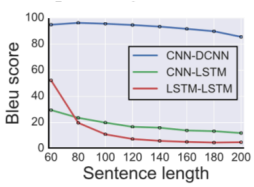
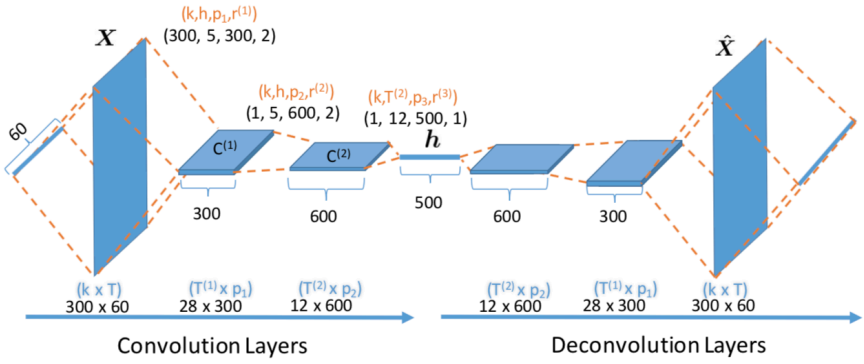



Figure 2: BLEU score *vs.* sentence length for Hotel Review data.

Gambar 2.3 - Hasil Evaluasi CNN-DCNN dari Paper




Gambar 3.1 - Arsitektur CNN-DCNN

## 3.2 CNN-DCNN Autoencoder

CNN-DCNN adalah sebuah arsitektur model deep learning yang di propose oleh Zhang et al (2017) untuk digunakan sebagai autoencoding framework. Seperti namanya *encoder* di CNN-DCNN menggunakan Convolutional sedangkan *decoder* nya menggunakan Deconvolutional.

Kita akan mulai dari membreakdown gimana arsitektur modelnya, kemudian alur data dari teks sampai output decodernya seperti apa, dan prosedur trainingnya gimana .

### 3.2.1 Model Architecture

Di bagian model architecture ini saya mencoba untuk tidak menjelaskan menggunakan rumus matematikanya melainkan saya lebih fokus untuk menjelaskan menggunakan visualisasi komponen apa aja yand ada di dalam tiap layer dengan sesimpel mungkin dan contoh kodenya langsung di PyTorch beserta proses forward dengan data asli biar tambah mudah untuk dimengerti.

**Input Layer** Kita akan mulai dari Input Layernya dulu, kalo di Gambar 3.1 disimbolkan dengan matrix `X` berukuran `kxT` dengan `k` adalah ukuran dimensi vektor tiap kata dan `T` adalah jumlah kata tiap review. Untuk simulasinya akan kita samakan dengan yang ada di paper, kita akan menggunakan `k=300` dan `T=60`. Catatanya pada paper column-wise kita akan row-wise, jadi matrix `X` di implementasi kita akan punya ukuran `Txk`. Matrix `X` diperoleh dengan proses berikut:

Langkah pertama, kita load dulu data trainingnya lalu kita ubah menjadi n-dimensional array dengan ukuran `[jumlah_review, T]`. Dilangkah ini, kita susun vocabulary dari training data dan ubah review text yang sebelumnya berisi list of words menjadi list of word indices dalam vocabulary. Untuk memperkecil jumlah vocab kita hanya menggunakan kata yang muncul lebih dari 2 kali aja.

Selanjutnya kita batasi panjang review textnya menjadi `T`, agar lebih jelas `max_review_text_len=60`. Jika ada review yang mempunyai panjang lebih dari `max_review_text_len` akan kita potong, otherwise kita padding sampe `max_review_text_len`.

```
In [11]: # Collections berguna untuk vocab utility
         from collections import Counter
         # Numpy berguna untuk input dan output
         import numpy as np
         # Torch berguna untuk implementasi model
```

```
import torch
from torch.autograd import Variable

# Dataset
reviews = None
with open("data/imdb_train_data.txt") as f:
    reviews = [line.split() for line in f]
    print("Total Reviews:", len(reviews))
```

```
Total Reviews: 25000
```

In [53]:
```
# Kita kumpulkan katanya
words = []
for review_words in reviews:
    words.extend(review_words)
word_counter = Counter(words)

# Buat vocabulary
word2index = {"<PAD>": 0, "<UNK>": 1}
index2word = {0: "<PAD>", 1: "<UNK>"}
word_count = 2

for wc in word_counter.most_common():
    word = wc[0]
    count = wc[1]
    if count <= 2:
        continue
    if word not in word2index:
        word2index[word] = word_count
        index2word[word_count] = word
        word_count += 1

# Ubah jadi list of word ids
review_word_ids = []
for review_words in reviews:
    indices = []
    for word in review_words:
        index = word2index["<UNK>"]
        if word in word2index:
            index = word2index[word]
        indices.append(index)
    review_word_ids.append(indices)

print(reviews[:2])
print(review_word_ids[:2])

word = reviews[0][0]
```

```
        print("ID", "'{}'".format(word), "is", word2index[word])

        print("word_count:", word_count)
```

```
[['for', 'a', 'movie', 'that', 'gets', 'no', 'respect', 'there', 'sure', 'are', 'a', 'lot', 'o
[[16, 4, 18, 12, 211, 58, 1134, 40, 249, 26, 4, 174, 5, 886, 4309, 3528, 16, 11, 1496, 820, 4,
ID 'for' is 16
word_count: 37472
```

```
In [54]: max_review_text_len = 60
         dataset = []
         pad_id = word2index["<PAD>"]
         for review_word_id in review_word_ids:
             if len(review_word_id) <= max_review_text_len:
                 # Kita pad review text yg kurang dari `max_review_text_len`
                 padded = np.pad(array=np.array(review_word_id, dtype=np.int32),
                                 pad_width=(0, max_review_text_len - len(review_word_id)),
                                 mode="constant",
                                 constant_values=pad_id)
                 dataset.append(padded)
             else:
                 # Kita potong yang lebih dari `review_word_ids`
                 truncated = np.array(review_word_id[:max_review_text_len])
                 dataset.append(truncated)
         dataset = np.array(dataset, dtype=np.int32)
         dataset
```

```
Out[54]: array([[   16,     4,    18, ...,     0,     0,     0],
                [ 1135,   187,    18, ...,   398,  8476, 24103],
                [    4,  1129,    46, ...,     0,     0,     0],
                ...,
                [  435,   147,  9058, ...,     2,   307,  5956],
                [ 2277,  1769,  6062, ...,  4557,    15,     4],
                [   24,    12,    10, ...,   119,   154,     5]], dtype=int32)
```

```
In [55]: print("Shape:", dataset.shape)
```

```
Shape: (25000, 60)
```

Langkah kedua adalah menghitung matrix `X` nya. Setelah kita dapet n-dimensional array dengan ukuran `[jumlah_review, T]`, ini nanti yang kita kalikan dengan matrix embeddingnya.

Di langkah ini kita harus menginialisasi matrix embeddingnya dulu, kalau dipapernya matrix embedding `We` mmepunyai ukuran `jumlah_kata_dalam_vocab x k`, lalu l2-normnya menyesuaikan di paper juga. l2-norm adalah batas nilai dari suatu vektor, dalam hal ini vektor tiap kata dalam matrix embedding. Untuk mengimplementasikan ini, di PyTorch ada `torch.nn.Embedding` yang bisa kita pakai.

```
In [56]: k = 300
         embed = torch.nn.Embedding(num_embeddings=word_count,
                                    embedding_dim=k, # Note: Nilai 'k'
                                    max_norm=1.0,
                                    norm_type=2.0)

         # Kita coba ambil dari batch dengan ukuran 1 dari dataset
         # terus kita ubah ke torch.Tensor dulu
         sample = torch.from_numpy(dataset[:1]).type(torch.LongTensor)
         sample = Variable(sample, requires_grad=False)

         # Kita hitung X
         X = embed(sample)
         print("X => {}".format(X.size()))
         # Kita ubah ke 4 dimensi inputnya
         X = X.view(X.size()[0], 1, X.size()[1], X.size()[2])
         print("X => {}".format(X.size()))
```

```
X => torch.Size([1, 60, 300])
X => torch.Size([1, 1, 60, 300])
```

Sampai sini kita sudah dapat matrix `X` dengan ukuran `60x300` sesuai `Txk` yang kita definisikan sebelumnya dengan `k=300` adalah besar vektor tiap kata dan `T=60` adalah jumlah review tiap kata. Matrix inilah yang nantinya akan diencode oleh *encoder* seperti dibawah ini.

**Convolutional Layer** *Encoder* pada CNN-DCNN menggunakan arsitektur *Convolutional Nerual Network* (CNN). CNN yang dipakai terdiri dari `L` layers, `layers = {1, 2, ... L}`. Dimana setiap layer adalah convolutional layer diakhiri oleh fully-connected layer di layer `L`. Tujuan dari layer ini adalah mensummarize input kalimatnya menjadi latent representation vector `h` yang mempunyai ukuran tetap (fixed-length vector, misal: 300). Untuk setiap convolutional layernya menggunakan ReLU sebagai fungsi aktivasi feature mapnya.

Dengan tau requirementsnya seperti apa, kita bisa langsung implementasikan ini. Untuk kasus ini kita pakai sesuai yang ada di paper, 3-layer convolutional encoder. Kita mulai dengan mendefinisikan layer convolutionalnya, kalau kita breakdown jadi gini:

1. Convolutional layer 1 `l=1`
   - Filter size `5`
   - `300` filters of the same size
   - dan stride size `2`
2. Convolutional layer 2 `l=2`
   - Filter size `5`
   - `600` filters of the same size
   - dan stride size `2`.
3. Convolutional layer 3 `l=1`
   - Filter size `5`

- 500 filters of the same size
- dan stride size 1.

Ekspektasi kita dalam proses encode ini adalah: dengan input matrix `X` berukuran `60x300` kita akan dapat output latent representation vector `h` berukuran `500x1`. Untuk implementasinya kita bisa pake torch.nn.Conv2d untuk Convolutional layernya. Berikut langkah-langkah encode matrix `X`:

```
In [57]: # Layer l=1
         conv1_num_filters = 300
         conv1_kernel_size = (5, k)
         conv1_stride_size = 2
         conv1 = torch.nn.Conv2d(in_channels=1,
                                 out_channels=conv1_num_filters,
                                 kernel_size=conv1_kernel_size,
                                 stride=conv1_stride_size)
         c1 = conv1(X)
         h1 = torch.nn.functional.relu(c1)
         print("h1 => {}".format(h1.size()))

h1 => torch.Size([1, 300, 28, 1])


In [58]: # Layer l=2
         conv2_num_filters = 600
         conv2_kernel_size = (5, 1)
         conv2_stride_size = 2
         conv2 = torch.nn.Conv2d(in_channels=conv1_num_filters,
                                 out_channels=conv2_num_filters,
                                 kernel_size=conv2_kernel_size,
                                 stride=conv2_stride_size)
         c2 = conv2(h1)
         h2 = torch.nn.functional.relu(c2)
         print("h2 => {}".format(h2.size()))

h2 => torch.Size([1, 600, 12, 1])


In [59]: # Layer l=3
         conv3_num_filters = 500
         conv3_filter_size = (12, 1)
         conv3_stride_size = 1
         conv3 = torch.nn.Conv2d(in_channels=conv2_num_filters,
                                 out_channels=conv3_num_filters,
                                 kernel_size=conv3_filter_size,
                                 stride=conv3_stride_size)
         c3 = conv3(h2)
         h = torch.nn.functional.relu(c3)
         print("h => {}".format(h.size()))
```

```
h => torch.Size([1, 500, 1, 1])
```

Kalo kita lihat sudah sesuai dengan di paper, pada fase encoding ini kita dapet latent representation vector `h` dengan ukuran `500x1`.

### 3.2.2 Deconvolutional Layer

*Decoder* pada CNN-DCNN prosesnya sama kaya di *Encoder* hanya saja menggunakan operasi transpose konvolusional dan di balik. Operasinya dimulai dari layer convolutional terakhir ke awal.

Ekspektasi kita dalam proses ini adalah: dengan input latent representation vektor `h` berukuran `500x1` kita akan dapat output matrix `X'` berukuran `60x300`. Kita definisikan dulu layer-layer decodernya lalu melakukan proses decode langkah demi langkah.

```
In [60]: # Deconvolutional layer l=1
         deconv1_num_filters = 600
         deconv1_filter_size = (12, 1)
         deconv1_stride_size = 2
         deconv1 = torch.nn.ConvTranspose2d(in_channels=conv3_num_filters,
                                            out_channels=deconv1_num_filters,
                                            kernel_size=deconv1_filter_size,
                                            stride=deconv1_stride_size)
         dc1 = deconv1(h)
         dh1 = torch.nn.functional.relu(dc1)
         print("dh1 => {}".format(dh1.size()))

dh1 => torch.Size([1, 600, 12, 1])


In [61]: # Deconvolutional layer l=2
         deconv2_num_filters = 300
         deconv2_filter_size = (6, 1)
         deconv2_stride_size = 2
         deconv2 = torch.nn.ConvTranspose2d(in_channels=deconv1_num_filters,
                                            out_channels=deconv2_num_filters,
                                            kernel_size=deconv2_filter_size,
                                            stride=deconv2_stride_size)
         dc2 = deconv2(dh1)
         dh2 = torch.nn.functional.relu(dc2)
         print("dh2 => {}".format(dh2.size()))

dh2 => torch.Size([1, 300, 28, 1])


In [64]: # Deconvolutional layer l=3
         deconv3_num_filters = 1
         deconv3_filter_size = (6, k)
         deconv3_stride_size = 2
```

```
        deconv3 = torch.nn.ConvTranspose2d(in_channels=deconv2_num_filters,
                                           out_channels=deconv3_num_filters,
                                           kernel_size=deconv3_filter_size,
                                           stride=deconv3_stride_size)
        dc3 = deconv3(dh2)
        X_hat = torch.nn.functional.relu(dc3)
        print(X_hat)
        print("X_hat => {}".format(X_hat.size()))
```

```
Variable containing:
( 0 , 0 ,.,.) =
1.00000e-04 *
  0.9578  1.4422  0.8380  ...   0.9923  0.9964  0.6853
  0.4944  2.3805  1.3768  ...   1.0383  0.0000  3.0957
  0.0000  1.5221  0.6717  ...   0.5288  0.4584  1.8805
           ...                          ...
  1.2827  1.8125  1.1380  ...   1.4202  1.8816  1.8018
  0.9722  0.0000  1.4844  ...   1.0620  0.6958  0.3163
  1.4584  0.8830  1.7582  ...   1.9010  1.5552  0.3742
[torch.FloatTensor of size 1x1x60x300]

X_hat => torch.Size([1, 1, 60, 300])
```

Sampai disini kita berhasil untuk mendecode latent representation vector `h` menjadi matrix `X_hat`. Langkah terakhir adalah output layernya.

**Output layer** Di output layer ini ada normalisasi `X_hat` dan menghitung nilai probabilitas `p(wt_hat = wt)` (rumus ada di paper) untuk tiap kata yang di rekonstruksi dalam kalimat `X_hat`. Jadi nanti outputnya adalah matrix dengan ukuran `[jumlah_kata_dalam_kalimat x banyak_vocab]` dimana setiap barisnya adalah 1-hot vektor dimana kata ke-`t` dalam vocab akan muncul. Implementasi di PyTorch kuran lebih kaya gini:

```
In [65]: # Kita ubah dulu dari 4-dimensi (output deconv)
         # jadi 3-dimensi: [jumlah_sample, jumlah_kata, besar_dimensi_vektor]
         X_hat = X_hat.squeeze(dim=1)

         # Kita lakukan L2-Norm dulu sesuai paper
         p_norm = torch.norm(X_hat, 2, dim=2, keepdim=True)
         X_hat = X_hat / p_norm

         # Define beberapa parameter yg kira perluin untuk rumus p(wt_hat = v)
         param_tau = 0.01
         matrix_embedding = Variable(embed.weight.data).t()
         # Kita expand size matrix_embedding biar sama kaya X_hat
         matrix_embedding = matrix_embedding.expand(X_hat.size()[0], *matrix_embedding.size())

         # Kita hitung sesuai rumus p(wt_hat = v)
```

```
        prob_logits = torch.bmm(X_hat, matrix_embedding) / param_tau
        log_prob = torch.nn.functional.log_softmax(prob_logits, dim=2)
        print(log_prob)
```

```
Variable containing:
(  0  ,.,.) =
 -400.8444 -451.6430 -405.4294  ...   -527.9830 -403.1998 -401.4234
 -361.0033 -395.0816 -366.3619  ...   -427.3355 -378.9833 -226.7664
 -388.9048 -434.0253 -396.0014  ...   -478.8749 -365.3870 -397.3977
              ...                          ...
 -518.9983 -518.2566 -508.9860  ...   -524.0754 -535.5562 -490.0146
 -430.1424 -533.0607 -422.4556  ...   -438.9644 -404.8709 -368.0703
 -483.9164 -550.8015 -480.8058  ...   -494.9898 -425.6831 -437.8596
[torch.FloatTensor of size 1x60x37472]
```

Diatas sudah menjelaskan bagaimana arsitektur model sekaligus alur tensornya menggunakan data asli. Selanjutnya akan dibahas mengenai penerapan beserta prosedur perlatihannya untuk salah satu task NLP yaitu Sentiment Classification.

### 3.3 Task: Sentiment Classification

Pada bagian ini kita akan coba reproduce Supervised model yang ada di papernya. Di papernya dijelaskan kalau supervised model terdiri convolutional encoder dengan latent size 500 diikuti dengan Multi-layer perceptron dengan hidden units 300. Mudah sekali di implemen dengan PyTorch.

#### 3.3.1 Dataset

Dataset yang akan kita gunakan adalah Large Movie Review Dataset v1.0. Di dataset tersebut terdapat ulasan film dari IMDB beserta labelnya yaitu positif atau negatif. Untuk mempermudah operasi waktu load data, seperti melakukan batching dan shuffle, kita bisa menggunakan interface/absctract class torch.utils.data.Dataset, kita tinggal merewrite kode sebelumnya menjado subclass dari `torch.utils.data.Dataset`.

Seperti dibawah ini:

```
In [2]: import torch
        import numpy as np

        from torch.utils import data
        from collections import Counter

        class Dataset(data.Dataset):
            def __init__(self,
                         train_data_path,
                         train_label_path,
                         max_review_text_len):
                """Inialisasi dataset dengan train_data_path"""
```

```
self.train_data_path = train_data_path
self.train_label_path = train_label_path
self.max_review_text_len = max_review_text_len

reviews = None
with open(self.train_data_path) as f:
    reviews = [line.split() for line in f]
self.total = len(reviews)

# Kita kumpulkan katanya
words = []
for review_words in reviews:
    words.extend(review_words)
word_counter = Counter(words)

# Buat vocabulary
self.word2index = {"<PAD>": 0, "<UNK>": 1}
self.index2word = {0: "<PAD>", 1: "<UNK>"}
self.vocab_size = 2

for wc in word_counter.most_common():
    word = wc[0]
    count = wc[1]
    if count <= 2:
        continue
    if word not in self.word2index:
        self.word2index[word] = self.vocab_size
        self.index2word[self.vocab_size] = word
        self.vocab_size += 1

# Ubah jadi list of word ids
review_word_ids = []
for review_words in reviews:
    indices = []
    for word in review_words:
        index = self.word2index["<UNK>"]
        if word in self.word2index:
            index = self.word2index[word]
        indices.append(index)
    review_word_ids.append(indices)


# Susun datasetnya
# Yang melebihi MAX_REVIEW_TEXT_LEN kita potong
# yg kurang kita padding.
review_texts = []
pad_id = self.word2index["<PAD>"]
for review_word_id in review_word_ids:
```

```
                if len(review_word_id) <= self.max_review_text_len:
                    # Kita pad review text yg kurang dari `max_review_text_len`
                    padded = np.pad(array=np.array(review_word_id, dtype=np.int32),
                                    pad_width=(0, self.max_review_text_len - len(review_wo
                                    mode="constant",
                                    constant_values=pad_id)
                    review_texts.append(padded)
                else:
                    # Kita potong yang lebih dari `review_word_ids`
                    truncated = np.array(review_word_id[:self.max_review_text_len])
                    review_texts.append(truncated)
            self.review_texts = np.array(review_texts, dtype=np.int32)

            # Label
            self.review_labels = []
            with open(self.train_label_path, "r") as f:
                self.review_labels = np.array([np.array([int(label)]) for label in f], dty

        def __getitem__(self, index):
            review_text = self.review_texts[index]
            review_label = self.review_labels[index]

            data = torch.from_numpy(review_text).type(torch.LongTensor)
            label = torch.from_numpy(review_label).type(torch.LongTensor)
            sample = {"review_text": data, "review_label": label}
            return sample

        def __len__(self):
            return len(self.review_texts)
```

Dengan ini kita bisa menggunakannya sebagai input torch.utils.data.DataLoader untuk melakukan operasi batching atau shuffling. Kaya gini:

```
In [3]: dataset = Dataset(train_data_path="data/imdb_train_data.txt",
                          train_label_path="data/imdb_train_label.txt",
                          max_review_text_len=250)

        loader = data.DataLoader(dataset=dataset,
                                 batch_size=5,
                                 shuffle=True)

        # Misal kita ambil 5 batch aja secara acak
        i = 1
        for batch in loader:
            print("Review text")
            print(batch["review_text"])
            print("Review label")
            print(batch["review_label"])
```

```
            print("batch {} => {}".format(i, batch["review_text"].size()))
            if i >= 5: break
            i += 1

Review text

    11     14      4  ...       0      0      0
  2522    157   2686  ...       0      0      0
    15    109   1937  ...    9115      5      2
    59    154    101  ...       0      0      0
   161     10    374  ...       0      0      0
[torch.LongTensor of size 5x250]

Review label

 1
 1
 1
 1
 0
[torch.LongTensor of size 5x1]

batch 1 => torch.Size([5, 250])
Review text

    10    641     52  ...    1236     62     10
    10    242     55  ...       0      0      0
    10     14     24  ...       0      0      0
    38    283     10  ...       0      0      0
   602     48      2  ...       0      0      0
[torch.LongTensor of size 5x250]

Review label

 0
 0
 0
 0
 0
[torch.LongTensor of size 5x1]

batch 2 => torch.Size([5, 250])
Review text

    24      2    814  ...       0      0      0
    46     22     39  ...       0      0      0
    46     22    155  ...      12     33     61
   257    391     34  ...       0      0      0
```

```
     9     11     20  ...       1   5011     15
[torch.LongTensor of size 5x250]

Review label

 0
 1
 1
 1
 1
[torch.LongTensor of size 5x1]

batch 3 => torch.Size([5, 250])
Review text

 20664     14      4  ...       0      0      0
    11     20      7  ...       0      0      0
     2    349    321  ...       0      0      0
    10    293     11  ...       0      0      0
    10     14   6145  ...      50    158      0
[torch.LongTensor of size 5x250]

Review label

 1
 0
 1
 0
 0
[torch.LongTensor of size 5x1]

batch 4 => torch.Size([5, 250])
Review text

     4   4006  12405  ...       0      0      0
    48      4  11966  ...       0      0      0
     2  14183   2011  ...    1550      6  23679
   444     23     35  ...       0      0      0
    61    238     29  ...      19     16    147
[torch.LongTensor of size 5x250]

Review label

 0
 0
 0
 1
 1
```

```
[torch.LongTensor of size 5x1]

batch 5 => torch.Size([5, 250])
```

### 3.3.2 Supervised Model

Langkah berikutnya adalah menyusun model classifiernya. Untuk hal ini kita akan menggunakan torch.nn.Module, abstract class tersebut membantu kita untuk menyusun networknya dengan mudah.

Kita akan memakai kode implementasi sebelumnya lalu menggabungkannya dalam class `Supervised`.

### 3.3.3 Training Procedure

Untuk melatih model `Supervised` kita harus menginialisasi optimizernya, melakukan forward untuk memperoleh `y_hat` dan backward untuk memperbarui parameternya.

Berikut implementasinya:

```
In [5]: """Based on Kumparan's model interface
           https://github.com/kumparan/kumparanian/tree/master/interface
        """

        import torch
        from torch import autograd
        from torch.nn.functional import relu
        from torch.nn.functional import log_softmax
        from torch.nn.functional import nll_loss

        class Supervised(torch.nn.Module):
            def __init__(self, vocab_size, sentence_len, learning_rate):
                super(Supervised, self).__init__()
                # Inialisasi model
                self.learning_rate = learning_rate
                # Matrix embedding
                embedding_dim = 300
                latent_dim = 500
                self.embedding = torch.nn.Embedding(num_embeddings=vocab_size,
                                                    embedding_dim=embedding_dim,
                                                    max_norm=1.0,
                                                    norm_type=2.0)

                # Convolutional Layers
                conv1_num_filters = 300
                kernel_size = 5
                conv1_kernel_size = (kernel_size, embedding_dim)
                conv1_stride_size = 2
                self.conv1 = torch.nn.Conv2d(in_channels=1,
```

```
                                    out_channels=conv1_num_filters,
                                    kernel_size=conv1_kernel_size,
                                    stride=conv1_stride_size)

        conv2_num_filters = 600
        conv2_kernel_size = (kernel_size, 1)
        conv2_stride_size = 2
        self.conv2 = torch.nn.Conv2d(in_channels=conv1_num_filters,
                                    out_channels=conv2_num_filters,
                                    kernel_size=conv2_kernel_size,
                                    stride=conv2_stride_size)

        # Seperti yang di paper
        # T1 = (T - kernel_size)/stride_size + 1
        T1 = int((sentence_len - kernel_size)/conv1_stride_size + 1)
        #  T2 = (T1 - kernel_size)/stride_size + 1
        T2 = int((T1 - kernel_size)/conv2_stride_size + 1)
        latent_size = 500
        conv3_num_filters = latent_size
        conv3_stride_size = 1
        conv3_filter_size = (T2, 1)
        self.conv3 = torch.nn.Conv2d(in_channels=conv2_num_filters,
                                    out_channels=conv3_num_filters,
                                    kernel_size=conv3_filter_size,
                                    stride=conv3_stride_size)
        # Fully-connected layer
        output_dim = 2 # untuk 2 class
        dropout_p = 0.5 # probability untuk drop out; sesuai paper
        mlp_hidden_units = 300
        self.fully_connected = torch.nn.Linear(conv3_num_filters, mlp_hidden_units)
        self.output = torch.nn.Linear(mlp_hidden_units, output_dim)
        self.dropout = torch.nn.Dropout(dropout_p)

        # Initialize optimizer
        self.optimizer = torch.optim.Adam(params=self.parameters(),
                                          lr=self.learning_rate)

    def forward(self, x):
        """Forward return y_hat, predicted label """
        # Kita clear dulu optimizer sebelumnya
        self.optimizer.zero_grad()

        # Kita hitung X
        X = self.embedding(x)
        # Kita ubah ke 4 dimensi inputnya
        X = X.view(X.size()[0], 1, X.size()[1], X.size()[2])

        # Matrix X kita encode jadi latent vector h3
```

```
        c1 = self.conv1(X)
        h1 = relu(c1)
        c2 = self.conv2(h1)
        h2 = relu(c2)
        c3 = self.conv3(h2)
        h3 = relu(c3)
        h3 = h3.squeeze()

        # Latent vector h3 nya kita forward ke fully-connected
        h4 = self.fully_connected(h3)
        h4 = self.dropout(h4)

        # Output layer
        out = self.output(h4)
        output = log_softmax(out, dim=1)
        return output

    def backward(self, y_hat, y):
        # Hitung nilai loss nya
        loss = nll_loss(y_hat, y)

        # Backward
        loss.backward()

        # Update parameter
        self.optimizer.step()
        return loss.data[0]

    def predict(self, x):
        y_hat = self.forward(x)
        _, predicted = torch.max(y_hat, dim=1)
        return predicted
```

prosedur trainingnya sebagai berikut, untuk satu epoch:

```
In [7]: # 1. Set parameter seperti epoch, learning rate dll
        max_epoch = 100  # untuk demo 2 epoch aja
        learning_rate = 0.005
        batch_size = 128

        # 2. Load datanya
        dataset = Dataset(train_data_path="data/imdb_train_data.txt",
                          train_label_path="data/imdb_train_label.txt",
                          max_review_text_len=250)

        loader = data.DataLoader(dataset=dataset,
                                 batch_size=batch_size,
                                 shuffle=True)
```

```
        model = Supervised(vocab_size=dataset.vocab_size,
                           sentence_len=dataset.max_review_text_len,
                           learning_rate=learning_rate)
        # Kita set dalam training mode
        model.train()

        for epoch in range(1, max_epoch + 1):
            for batch in loader:
                x = autograd.Variable(batch["review_text"])
                y = batch["review_label"]
                y = y.view(y.size()[0])
                y = autograd.Variable(y)

                # Forward & Backward
                y_hat = model.forward(x)
                loss = model.backward(y_hat, y)
                print("loss:", loss)
```

```
loss: 0.6947369575500488
loss: 7.018113136291504
loss: 1.079958200454712
loss: 0.6920189261436462
```

```
        ---------------------------------------------------------------------------

        KeyboardInterrupt                         Traceback (most recent call last)

        <ipython-input-7-d45e3c18d15b> in <module>()
         28         # Forward & Backward
         29         y_hat = model.forward(x)
    ---> 30         loss = model.backward(y_hat, y)
         31         print("loss:", loss)


        <ipython-input-5-12c8d5063709> in backward(self, y_hat, y)
         98
         99         # Backward
    --> 100         loss.backward()
        101
        102         # Update parameter


        ~/.virtualenvs/pytorch-guide/lib/python3.6/site-packages/torch/autograd/variable.py in
        165                 Variable.
        166         """
```

```
--> 167         torch.autograd.backward(self, gradient, retain_graph, create_graph, retain
    168
    169     def register_hook(self, hook):


~/.virtualenvs/pytorch-guide/lib/python3.6/site-packages/torch/autograd/__init__.py in
 97
 98     Variable._execution_engine.run_backward(
---> 99         variables, grad_variables, retain_graph)
100
101


KeyboardInterrupt:
```

#### 3.3.4 Snapshot

Kalau proses diatas aku stop karena lumayan lama. Kalau sudah selesai trainingnya nanti bisa save modelnya dengan cara:

```
In [8]: # 3. Simpan modelnya
        output_path = "model.pickle"
        torch.save(model.state_dict(), output_path)
```

Lalu load dengan cara:

```
model = Supervised(..)
model.load_state_dict(torch.load(PATH))
```

#### 3.3.5 Future Improvement

Mungkin hal-hal berikut yang bisa kita lakuin untuk improve modelnya:

1. Implementasi batch norm di tiap layer convolution nya
2. Implementasi API server untuk deployment. Model tinggal load lalu perform predict.