Deep Learning: Getting Started With Neural Network

Rensheng Wang,
https://sit.instructure.com/courses/26266
February 1, 2018

Keras, TensorFlow, Theano, and CNTK

- Keras is a model-level library, providing high-level building blocks for developing deep-learning models.
- It doesnt handle low-level operations such as tensor manipulation and differentiation.
- Instead, it relies on a specialized, well-optimized tensor library to do so, serving as the backend engine of Keras.
- Rather than choosing a single tensor library and tying the implementation of Keras to that library, Keras handles the problem in a modular way (see figure below); thus several different backend engines can be plugged seamlessly into Keras.

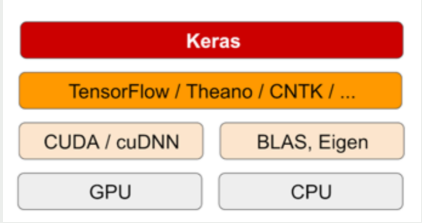


Figure: The deep-learning software and hardware stack.

Keras, TensorFlow, Theano, and CNTK

- Currently, the three existing backend implementations are the TensorFlow backend, the Theano backend, and the Microsoft Cognitive Toolkit (CNTK) backend. In the future, its likely that Keras will be extended to work with even more deep-learning execution engines.
- TensorFlow, CNTK, and Theano are some of the primary platforms for deep learning today.
- Any piece of code that you write with Keras can be run with any of these backends without having to change anything in the code: you can seamlessly switch between the two during development, which often proves usefulfor instance, if one of these backends proves to be faster for a specific task.
- Via TensorFlow (or Theano, or CNTK), Keras is able to run seamlessly on both CPUs and GPUs.

Developing with Keras: a quick overview

The typical Keras workflow:

1. Define your training data: input tensors and target tensors.

Developing with Keras: a quick overview

The typical Keras workflow:

1. Define your training data: input tensors and target tensors.
2. Define a network of layers (or model) that maps your inputs to your targets.

Developing with Keras: a quick overview

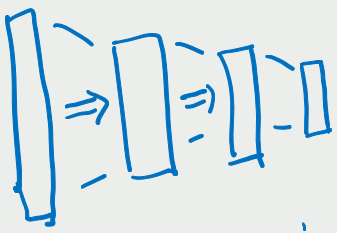
The typical Keras workflow:

1. Define your training data: input tensors and target tensors.
2. Define a network of layers (or model) that maps your inputs to your targets.
3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.

Developing with Keras: a quick overview

The typical Keras workflow:

1. Define your training data: input tensors and target tensors.
2. Define a network of layers (or model) that maps your inputs to your targets.
3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
4. Iterate on your training data by calling the `fit()` method of your model.



- high-D i/p ⇒ low-D o/p
- decide # of hidden layers
- set feedback → loss fn or & tuning by optimizer

Developing with Keras: a quick overview

The typical Keras workflow:

1. Define your training data: input tensors and target tensors.
2. Define a network of layers (or model) that maps your inputs to your targets.
3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
4. Iterate on your training data by calling the `fit()` method of your model.

There are two ways to define a model:

- Using the `Sequential` class (only for linear stacks of layers), which is the most common network architecture so far.
- Using the functional API (for directed acyclic graphics of layers), which lets you build completely arbitrary arichitectures.

`Sequential` class vs. Functional API

- Heres a two-layer model defined using the Sequential class (note that were passing the expected shape of the input data to the first layer):

Sequential class vs. Functional API

- Heres a two-layer model defined using the Sequential class (note that were passing the expected shape of the input data to the first layer):

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```


Sequential class vs. Functional API

- Heres a two-layer model defined using the Sequential class (note that were passing the expected shape of the input data to the first layer):

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

- Heres the same model defined using the functional API:

Sequential class vs. Functional API

- Heres a two-layer model defined using the Sequential class (note that were passing the expected shape of the input data to the first layer):

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

- Heres the same model defined using the functional API:

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = models.Model(inputs=input_tensor, outputs=output_tensor)
```

With the functional API, youre manipulating the data tensors that the model processes and applying layers to this tensor as if they were functions

Developing with Keras: a quick overview

- Once your model architecture is defined, it doesnt matter whether you used a Sequential model or the functional API. All of the following steps are the same.

Developing with Keras: a quick overview

- Once your model architecture is defined, it doesnt matter whether you used a Sequential model or the functional API. All of the following steps are the same.
- The learning process is configured in the compilation step, where you specify the optimizer and loss function(s) that the model should use, as well as the metrics you want to monitor during training.

Developing with Keras: a quick overview

- Once your model architecture is defined, it doesnt matter whether you used a Sequential model or the functional API. All of the following steps are the same.
- The learning process is configured in the compilation step, where you specify the optimizer and loss function(s) that the model should use, as well as the metrics you want to monitor during training.

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
loss='mse',
metrics=['accuracy'])
```


Developing with Keras: a quick overview

- Once your model architecture is defined, it doesnt matter whether you used a Sequential model or the functional API. All of the following steps are the same.
- The learning process is configured in the compilation step, where you specify the optimizer and loss function(s) that the model should use, as well as the metrics you want to monitor during training.

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
loss='mse',
metrics=['accuracy'])
```

- Finally, the learning process consists of passing arrays of input data to the model via the `fit()` method

Developing with Keras: a quick overview

- Once your model architecture is defined, it doesnt matter whether you used a Sequential model or the functional API. All of the following steps are the same.
- The learning process is configured in the compilation step, where you specify the optimizer and loss function(s) that the model should use, as well as the metrics you want to monitor during training.

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
loss='mse',
metrics=['accuracy'])
```

- Finally, the learning process consists of passing arrays of input data to the model via the `fit()` method

```
model.fit(input_tensor, target_tensor, batch_size=128,
epochs=10)
```

↳ you repeat and also reshuffle the terms

The IMDB DataSet

- IMDB dataset: a set of 50,000 highly polarized reviews from the Internet Movie Database.
- Theyre split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.
- Why use separate training and test sets? Because you should never test a machine-learning model on the same data that you used to train it!
- Just because a model per- forms well on its training data doesnt mean it will perform well on data it has never seen; and what you care about is your models performance on new data.
- Just like the MNIST dataset, the IMDB dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

Loading IMDB DataSet

- Loading DataSet

```
from keras.datasets import imdb (train_data, train_labels),
(test_data, test_labels) = imdb.load_data( num_words=10000)
```

- The argument `num_words=10000` means youll only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows you to work with vector data of manageable size.
- The variables `train_data` and `test_data` are lists of reviews; each review is a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists of 0s and 1s, where "0" stands for negative and "1" stands for positive:
- Peek the data:

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]
>>> train_labels[0]
1
```


Preparing IMDB DataSet

- Because youre restricting yourself to the top 10,000 most frequent words, no word index will exceed 10,000:

```
>>> max([max(sequence) for sequence in train_data])
9999
```

- You cant feed lists of integers into a neural network. You have to turn your lists into tensors. There are two ways to do that:

  1. Pad your lists so that they all have the same length, turn them into an integer tensor of shape `(samples, word_indices)`, and then use as the first layer in your network a layer capable of handling such integer tensors.
  2. One-hot encode your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence [3, 5] into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s. Then you could use as the first layer in your network a Dense layer, capable of handling floating-point vector data.

Encoding the Integer Sequences into a Binary Matrix

- Lets go with the latter solution to vectorize the data:

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

- Heres what the samples look like now:

```
>>> x_train[0]
array([ 0., 1., 1., ..., 0., 0., 0.])
```

- You should also vectorize your labels. Then the data is ready for a neural network.

```
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Building Your Network

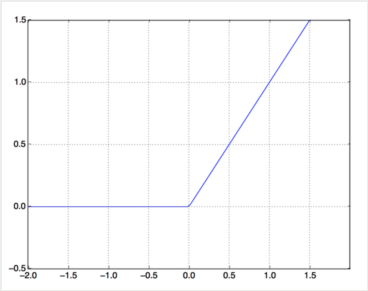
- The input data is vectors, and the labels are scalars (1s and 0s).
- A type of network that performs well on such a problem is a simple stack of fully connected (`Dense`) layers with `relu`activations: Dense(16, activation='relu').
- The argument being passed to each `Dense` layer (16) is the number of hidden units of the layer. A hidden unit is a dimension in the representation space of the layer. Each such Dense layer with a relu activation implements the following chain of tensor operations:

  ```
  output = relu(dot(W, input) + b)
  ```

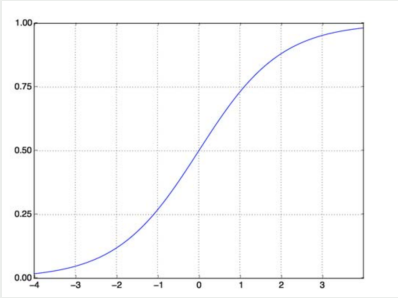
- There are two key architecture decisions to be made about such a stack of Dense layers:
  - How many layers to use
  - How many hidden units to choose for each layer

Building Your Network

- The intermediate layers will use relu as their activation function, and the final layer will use a sigmoid activation so as to output a probability (a score between 0 and 1, indicating how likely the sample is to have the target "1": how likely the review is to be positive)
- A relu (rectified linear unit) function vs. sigmoid function:



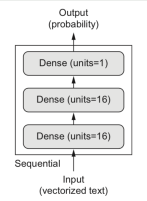
(a) ReLU



(b) sigmoid

The Three-layer Network

- A sigmoid squashes arbitrary values into the [0, 1] interval, outputting something that can be interpreted as a probability.




```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

- Why are activation functions necessary?
  - Without an activation function like `relu` (also called a non-linearity), the `Dense` layer would consist of two linear operationsa dot product and an addition.
  - So the layer could only learn linear transformations of the input data.
  - Such a hypothesis space is too restricted and wouldnt benefit from multiple layers of representations, because a deep stack of linear layers would still implement a linear operation: adding more layers wouldnt extend the hypothesis space.

Choose a Loss Function

- Because we are facing a binary classification problem and the output of the network is a probability (you end your network with a single-unit layer with a sigmoid activation), its best to use the `binary_crossentropy` loss.
- Cross entropy formula given two distributions over discrete variable x, where $q(x)$ is the estimate for true distribution $p(x)$ is given by

$$H(p, q) = -\sum_{\forall x} p(x) \log(q(x))$$

- In a neural network, the binary cross entropy will show as the quation written into a form where $\mathbf{y}$ is the ground truth vector and $\hat{\mathbf{y}}$ is the estimate.

$$H(\mathbf{y}, \hat{\mathbf{y}}) = -\left(\mathbf{y} \log(\hat{\mathbf{y}}) + (1 - \mathbf{y}) \log(1 - \hat{\mathbf{y}})\right)$$

- For instance, for binary output $\mathbf{y} = [1,\ 0,\ 1,\ \cdots]$ while $\hat{\mathbf{y}} = [0.4,\ 0.7,\ 0.9,\ \cdots]$

$$H(\mathbf{y}, \hat{\mathbf{y}}) = -\left[1 \times \log(0.4) + (1 - 0) \times \log(1 - 0.7) + 1 \times \log(0.9)\right]$$

Compilation Step

- Setup compilation steps:
  - A *loss function*: How the network will be able to measure its performance on the training data, and thus how it will be able to steer itself in the right direction.
  - An *optimizer*: The mechanism through which the network will update itself based on the data it sees and its loss function.
  - *Metrics to monitor during training and testing*: Here, well only care about accuracy (the fraction of the items that were correctly classified).
- Heres the step where you configure the model with the `rmsprop` optimizer and the `binary_crossentropy` loss function.
- Note that youll also monitor `accuracy` during training.

```
model.compile(optimizer='rmsprop',
    loss='binary_crossentropy',
    metrics=['accuracy'])
```

Validating the Approach

- In order to monitor during training the accuracy of the model on data it has never seen before, youll create a validation set by setting apart 10,000 samples from the original training data.

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
```

- Training the model:

```
model.compile(optimizer='rmsprop',
    loss='binary_crossentropy',
    metrics=['acc'])

history = model.fit(partial_x_train, partial_y_train, epochs=20,
batch_size=512, validation_data=(x_val, y_val))
```

Compare Training and Validation Loss

- Note that the call to `model.fit()` returns a History object. This object has a member history, which is a dictionary containing data about everything that happened during training.

```
>>> history_dict = history.history
>>> history_dict.keys()
    [u'acc', u'loss', u'val_acc', u'val_loss']
```

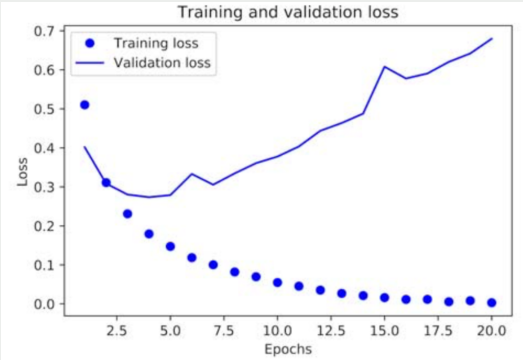
- Plotting the training and validation loss

```
import matplotlib.pyplot as plt
history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(acc) + 1)
```

Plotting the Training and Validation Loss

- Plotting the training and validation loss

```
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Generate Predictions on New Data

- After having trained a network, youll want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the `predict` method:

```
>>>   model.predict(x_test)

array([[ 0.98006207]
       [ 0.99758697]
       [ 0.99975556]
       ...,
       [ 0.82167041]
       [ 0.02885115]
       [ 0.65371346]], dtype=float32)
```

- The model performance is evaluated via the test data set instead of the training data set.

Summary

- Usually the preprocessing of the raw data is needed in order to be able to feed itas tensorsinto a neural network. Sequences of words can be encoded as binary vectors, but there are other encoding options, too.
- Stacks of `Dense` layers with `relu` activations can solve a wide range of problems due to non-linear transform freedom.
- In a binary classification problem (two output classes), your network should end with a `Dense` layer with one unit and a `sigmoid` activation.
- With such a scalar `sigmoid` output on a binary classification problem, the loss function you should use is `binary_crossentropy`.
- The rmsprop optimizer is generally a good enough choice, whatever your problem.
- As they get better on their training data, neural networks eventually start over-fitting and end up obtaining increasingly worse results on data theyve never seen before. Be sure to always monitor performance on data that is outside of the training set.