Tuning Hyper-Parameters, Early Stopping

Benjamin Roth

CIS LMU München

January 10, 2018

Defining a simple model: Multilayer Perceptron

```
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
model.add(Dense(32, input_shape=(500,)))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='sgd',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)
```

- `Sequential` means that layers are stacked on top of each other in a sequential manner, not that input/output are sequences.
- `Dense`: Fully connected layer (operating on one batch).
  - Input and output size: batch size $\times$ representation size
  - Input size only needs to be specified for first layer (otherwise it equals output size of previous layer).
  - Elementwise or per-instance activations can be specified.
- `model.compile`: $\sim$ to `theano.function.compile(...)`, but specifies additional choices, such as learning algorithm.
- `model.fit`: run training

Learning algorithms

```
from keras.optimizers import SGD
my_opt=SGD(lr=0.01, momentum=0.9, decay=1e-6,
    nesterov=True)
model.compile(optimizer=my_opt,
    loss='categorical_crossentropy',
    metrics=['accuracy'])
```

- Learning algorithms can be passed as objects or as their string identifiers.
  - SGD: what we have used so far
  - Adagrad and variants: reduce update strength on frequently updated features
  - RMSProp: similar to Adagrad, but use running average of feature updates
  - ... we will have a separate lecture on those and others
- Typical parameters:
  - Learning rate decay in relation to number of updates.
  - Momentum ($\sim$ *"inertia"*): previous updates influence current update
  - Nesterov acceleration: lookahead in direction of old gradient when computing new gradient

Loss functions, metrics

```
from keras.optimizers import SGD
my_opt=SGD(lr=0.01, momentum=0.9, decay=1e-6,
    nesterov=True)
model.compile(optimizer=my_opt,
    loss='categorical_crossentropy',
    metrics=['accuracy'])
```

- Loss
  - All typical differentiable loss functions (cross-entropy, mean squared error, ...)
  - Cross entropy requires multiclass label to be *one-hot-encoded*. This will be handy in those cases:

    ```
    from keras.utils.np_utils import to_categorical
    one_hot_matrix = to_categorical(ids_vector)
    ```
- `metrics` are not necessarily differentiable, and are computed during training on train-data and optionally on dev-data (next slide)

Training, Callbacks

- `model.fit()`
  - Applies the learning algorithm to fit model to training data accoring to loss.
  - Computes metrics.
- Callbacks can be passed and are executed at certain points during training, e.g. in order to collect metrics in a list:

```
class ValAccHistory(Callback):
  def on_train_begin(self, logs={}):
    self.val_acc = []
  def on_epoch_end(self, batch, logs={}):
    self.val_acc.append(logs.get('val_acc'))

model.fit(train_x, train_y, batch_size=10, nb_epoch=20,
  validation_data=(valid_x, valid_y), callbacks=[history])

print(history.val_acc)
```

Typical Example of Model Training

```
early_stopper = EarlyStopping(monitor='val_f1_i',
    patience=1, mode="max")
checkpoint = ModelCheckpoint(modelpath,
    save_best_only=True, monitor='val_acc', mode="max")

model.fit([train_x], [train_labels],
    batch_size=self.batch_size,
    sample_weight=[train_sample_weights],
    callbacks=[early_stopper, checkpoint],
    nb_epoch=self.epochs,
    validation_data=(
    [valid_x],[self.valid_labels],
    [self.valid_sample_weights]),
    verbose=1)

m = model.load_weights(modelpath)
```

Hyperparameter Optimization

- *"My model trains in under 1 hour!"*
- (But I had to search over 1000 hyperparameter configurations, and on a new data set this needs to be done again.)
- Finding the right hyper-parameters is part of model training!

Hyperparameter Optimization

- Many parameters
  - `embedding_size` $\in \{10, \ldots, 1000\}$
  - `hidden_size` $\in \{10, \ldots, 1000\}$
  - `l1_regularizer` $\in \{0, 0.00001, \ldots, 1.0\}$
  - `l2_regularizer` $\in \{0, 0.00001, \ldots, 1.0\}$
  - `dropout` $\in \{0, \ldots, 1.0\}$
  - `optimizer` $\in \{$rmsprop,adagrad,sgd,...$\}$
  - `Momentum, Learning rate decay, ...`
- Approach 1, Grid search, nested for-loop:
  Try all possible combinations of values, select best combination according to validation data set.
- Problem: If we try only 4 values for each of them, search space over all combinations is $4^8 = 65536$

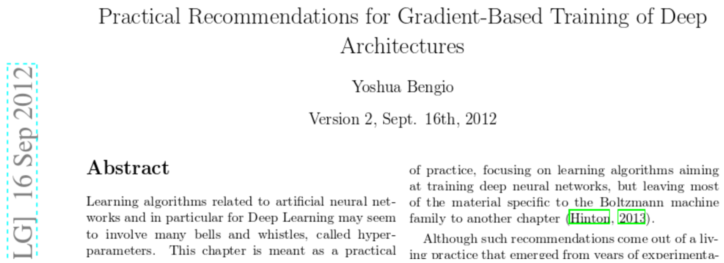
Hyperparameter Optimization 2

- Approach 2, Random sampling:
  For each of the different parameters sample values independently, and run as many configurations as you can afford (e.g. 100). Select according to performance on validation set.
- Intuition:
  - Some parameters values are good independent of other parameters. It is more important to hit those than to try all combinations.
  - Some parameters change the result very little over their whole range. Don't waste time exploring them, instead explore other parameters at the same time.
- Practical tips:
  - Discretize open-ended continous parameters on a (approximate) log scale and sample from that.
    E.g.: $0.01, 0.1, 1, 10, 100$
    or: $0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100$
    or: $0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100$
  - Closed-ended parameters can be discretized on a uniform scale.

Hyperparameter Optimization 3

- Approach 2a: Sample only over a small *"reasonable"* range of parameter values.
  - What if some values lie on a boundary?
  - Extend range on that boundaries manually (edit config file/code) and run optimization/search again ...
  - ... a huge waste of time!
- Approach 2b: Automate process of Approach 2a (`hyperopt.py`)
  - give the program *"unreasonably"* large range of parameter values
  - start sampling using *"reasonable"* subrange
  - extend subrange if best value after n iterations lies on a boundary.

More Ideas:



Practical Recommendations for Gradient-Based Training of Deep Architectures

Yoshua Bengio

Version 2, Sept. 16th, 2012

LG] 16 Sep 2012

Abstract

Learning algorithms related to artificial neural networks and in particular for Deep Learning may seem to involve many bells and whistles, called hyper-parameters. This chapter is meant as a practical of practice, focusing on learning algorithms aiming at training deep neural networks, but leaving most of the material specific to the Boltzmann machine family to another chapter (Hinton, 2013).

Although such recommendations come out of a living practice that emerged from years of experimenta-