Advanced neural networks: hyperparameter optimization

Erik Spence

SciNet HPC Consortium

2 October 2017

Today's slides

You can get the slides for today's class at the SciNet Education web page.

`https://support.scinet.utoronto.ca/education`

Click on the link for the class, and look under "File Storage", "hp_optimization.pdf".

About this class

The purpose of this class is to introduce you to a variety of advanced neural network architectures and techniques. Some notes about the class:

- The material is not introductory. You should already be familiar with neural networks and how they work.
- We will more-or-less begin where the "Introduction to Neural Networks" (DAT111) workshop left off.
- We'll be using Python 2.7.X.
- We'll be using the Keras neural-network programming framework, with a *Theano* back end (though it would likely also work using *TensorFlow* too).
- You will need the usual machine learning packages: numpy, matplotlib, scikit-learn.
- You'll obviously also need Theano (or Tensorflow) and Keras installed.

Ask questions!

Format of the class

How are we going to do this?

- We'll be using a class format, rather than a workshop format, since training the neural networks we'll be studying takes way too long.
- One class per week (or should we switch it to every other week?).
- No homework.
- Three weeks? Maybe more depending on interest.
- This material is all new. Please give me feedback on what you like and don't like.
- If there's a specific topic you'd like to see covered, let me know.

This is intended to be informal. Come if you can; no pressure.

Plan for the class

What are we going to cover?

- Hyperparameter optimization.
- Recurrent neural networks.
- Generative-adversarial networks.
- Reinforcement learning?
- Boltzmann networks? Autoencoders?

More classes may be added, if there is interest, and I find the time.

Hyperparameter optimization

Hyperparameters refer to the parameters and settings which define the neural network model. Examples include:

- Number of layers. Number of nodes within layers.
- Parameters within layers (for feature maps: filter size, stride).
- Choice of activation function.
- Choice of cost function.
- Optimization algorithm.
- Optimization algorithm hyperparameters. (stochastic gradient descent parameters: learning rate, mini-batch size, regularization parameters, decay rate).

There are zillions of possible combinations, all non-linearly coupled. How do you find the best combination of hyperparameters for your problem?

Hyperparameter optimization, continued

The first time I taught the "Introduction to neural network programming" workshop I claimed that trial-and-error was the most common way of determining hyperparameters. Are there other techniques available?

- Grid search: set out your grid of parameters and hit them all. Inefficient, though parallelizable.
- Random search: Can be more efficient than grid. Also parallelizable.
- Sequential Model-Based Optimization (SMBO) algorithms. These techniques model the performance of past hyperparameter combinations, and use that to predict the next best combination to evaluate:
  - Gaussian Processes.
  - Tree-structured Parzen Estimators.
- Others: particle swarm optimization (PSO), Simplex, Sobol sequences.

We'll go over Gaussian Processes.

Hyperparameter optimization, continued more

So what are we trying to do?

- We've got a neural network idea, and it works pretty well. But we think (hope?) we can do better, if we tweak the hyperparameters.
- Goal: find the best possible hyperparameter combination to solve our given problem.
- This obviously requires defining some definition of "best". Probably the best possible score for our test data.
- We do NOT want to perform a grid or random search.
- Instead we will use an algorithm which will look at past evaluations of the NN, and use those to make a prediction of which hyperparameter combination to evaluate next.

How do those algorithms work?

Sequential Model-Based Optimization

Sequential model-based optimization (SMBO) techniques are among the most efficient, in terms of number of function evaluations (training of the NN) required.

- The approach is inherently Bayesian.
- Given a model (neural network with a given set of hyperparameters), f, and a set of observations (evaluations of the neural network) $\mathcal{D}$, Bayes' Theorem states

$$P(f|\mathcal{D}) \propto P(\mathcal{D}|f)\,P(f)$$

- The "prior" probability of the model, $P(f)$, represents our belief about the space of possible models.
- The "posterior" probability of the model, given the observations, can be used to make predictions about what parameter combination should be calculated next.

Sequential Model-Based Optimization, continued

So what's the algorithm?

- Pick which hyperparameters we're going to explore, and choose their possible values or ranges.
- Randomly pick some hyperparameter combinations. For each combination, $\mathbf{x}$:
  - Put $\mathbf{x}$ in the set $\mathbf{X}$.
  - Calculate $f(\mathbf{x})$ and add it and $\mathbf{x}$ to $\mathcal{D} = \{\mathbf{x}, f(\mathbf{x})\}$ (the set of calculated values).
- Now repeat, for a pre-set number of iterations:
  - Using $\mathcal{D}$ and $\mathbf{X}$, determine the next combination of parameters to evaluate, $\mathbf{x}$.
  - Add $\mathbf{x}$ to $\mathbf{X}$.
  - Calculate $f(\mathbf{x})$ and add it and $\mathbf{x}$ to $\mathcal{D}$.

After the pre-set number of iterations have passed, you will have likely found the best hyperparameter combination.

Sequential Model-Based Optimization, continued more

The efficiency of this approach comes from the ability to incorporate new observations into the model, and use them to guide future parameter combination choices.

But how do we do that one step:

"Using $\mathcal{D}$ and $\mathbf{X}$, determine the next combination of parameters to evaluate."

How you approach this step determines which particular algorthm you're using. We'll look at Gaussian Processes. Tree-based Parsen Estimators are also used.

Note that this generic algorithm is not restricted to neural networks. Any machine learning algorithm which depends upon hyperparameter tuning can use this optimization approach.

Optimization using Gaussian Processes

The technique we're going to look at uses Gaussian Processes (GP).

- This technique directly estimates the posterior probability of the model, $\boldsymbol{P}(\boldsymbol{f}\,|\boldsymbol{\mathcal{D}})$.
- It does this by modelling the prior distribution as a Gaussian Process (GP).
- This means that $\boldsymbol{P}(\boldsymbol{f}\,|\boldsymbol{\mathcal{D}})$ is also a GP.
- Consequently, we can get an estimate for the mean values and standard deviations of all parameter combinations we care to test.
- These are then used to determine the next-best value of $\boldsymbol{x}$ to evaluate.

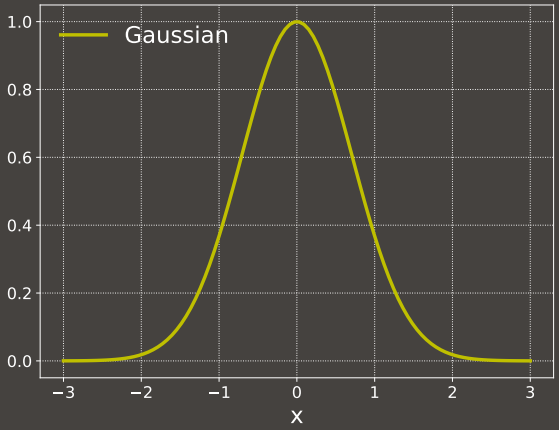
Gaussian distributions

Not familiar with Gaussian Processes? Let's start at the beginning.

Our old friend, the 1D Gaussian, has a probability distribution function given by

$$f(x, \mu, \sigma) = Ne^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where N is a normalization, μ is the mean and σ is the standard deviation.



Multivariate Gaussian distributions

The 1-D Gaussian can be generalized to higher dimensions.

$$\mathcal{N}(\mu, \Sigma) = Ne^{-\frac{(\mathrm{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathrm{x}-\mu)}{2}}$$

where $\boldsymbol{x}$ is now a $\boldsymbol{k}$-dimensional vector, $\boldsymbol{\mu}$ is the vector of mean values in each dimension, and $\boldsymbol{\Sigma}$ is the $\boldsymbol{k}$ x $\boldsymbol{k}$ covariance matrix

$$\Sigma = \mathrm{E}\left[(\mathrm{x} - \mu)(\mathrm{x} - \mu)^{\mathrm{T}}\right]$$

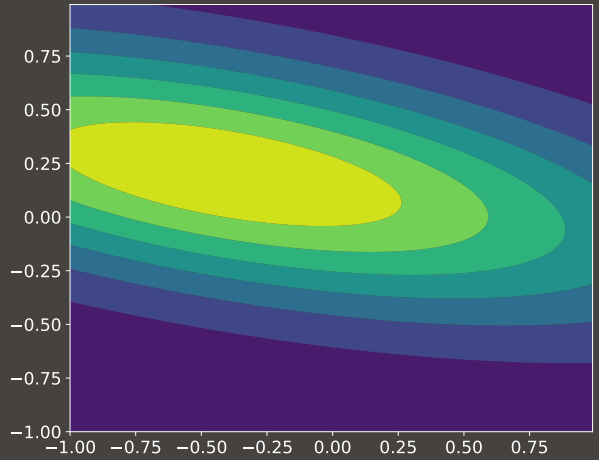
where E is the expectation value.

Multivariate Gaussian distributions, continued

If $\mu = [-0.4, 0.2]$ and

$$\Sigma = \begin{bmatrix} 1.5 & -0.3 \\ -0.3 & 0.2 \end{bmatrix}$$

then the probability distribution function looks like the figure to the right.



Gaussian Processes, continued

But what is a Gaussian Process (GP)?

- A GP is an extension of the multivariate Gaussian distribution to infinite dimensions.
- Any finite combination of the dimensions of the GP is a joint (multivariate) Gaussian distribution.
- As you know, a Gaussian distribution is a distribution of a random variable, specified by its mean and covariance.
- A GP is a distribution over functions, specified by a mean function, $m(\mathbf{x})$, and a covariance function, $k(\mathbf{x}, \mathbf{x}')$.
- When you sample from a GP, you get a mean and variance of a Gaussian distribution, for each dimension.
- For convenience, we can set our mean function to zero, $m(\mathbf{x}) = \mathbf{0}$.

Gaussian Processes, continued more

We have the function for the mean of our prior ($m(\mathbf{x}) = 0$), but we still need the covariance function to complete the GP. There are several popular ones out there.

- Linear: $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}'$
- Rational quadratic: $k(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{(\mathbf{x} - \mathbf{x}')^2}{2\alpha\ell^2}\right)^{-\alpha}$
- Squared exponential: $k(\mathbf{x}, \mathbf{x}') = e^{-\left(\frac{\mathbf{x}-\mathbf{x}'}{2\ell}\right)^2}$
- Matérn class: $k(\mathbf{x}, \mathbf{x}') = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}\,|\mathbf{x} - \mathbf{x}'|}{\ell}\right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu}\,|\mathbf{x} - \mathbf{x}'|}{\ell}\right)$

with ℓ a length scale, α and ν positive parameters, and K_ν the modified Bessel function.

We will use Matérn, with the default scikit-learn values.

Gaussian Processes, continued even more

So how does this work with Gaussian Processes?

- $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_p\}$ is the set of hyperparameter combinations that we've already tested, and $\mathbf{x}_{p+1}$ is a hyperparameter combination not yet tested.
- Let $\mathbf{f} = \{f(\mathbf{x}_1), f(\mathbf{x}_2), ..., f(\mathbf{x}_p)\}$ be the set of values generated by the test. I've been using the accuracy of the network on the validation data.
- The goal is the determine which value $\mathbf{x}_{p+1}$ to calculate next.
- The joint distribution of $\mathbf{f}$ and $f_{p+1} = f(\mathbf{x}_{p+1})$ is given by

$$\begin{bmatrix} \mathbf{f} \\ f_{p+1} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} & \mathbf{k}^T \\ \mathbf{k} & k(\mathbf{x}_{p+1}, \mathbf{x}_{p+1}) \end{bmatrix}\right)$$

where $K(\mathbf{X}, \mathbf{X})_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, $\mathcal{N}$ is the multivariate normal distribution, and $\mathbf{k} = [k(\mathbf{x}_{p+1}, \mathbf{x}_1)\ k(\mathbf{x}_{p+1}, \mathbf{x}_2)\ \ldots\ k(\mathbf{x}_{p+1}, \mathbf{x}_p)]$.

Bayesian hyperparameter optimization, continued more

The joint distribution is given by

$$\begin{bmatrix} \mathbf{f} \\ f_{p+1} \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} \mathbf{K} & \mathbf{k}^T \\ \mathbf{k} & k(\mathbf{x}_{p+1}, \mathbf{x}_{p+1}) \end{bmatrix}\right)$$

Then the posterior distribution is given by

$$P\left(f_{p+1} \,|\mathcal{D}, \mathbf{x}_{p+1}\right) = \mathcal{N}\left(\mu, \Sigma\right)$$

with

$$\mu = \mathbf{k}\mathbf{K}^{-1}\mathbf{f}$$
$$\Sigma = k(\mathbf{x}_{p+1}, \mathbf{x}_{p+1}) - \mathbf{k}\mathbf{K}^{-1}\mathbf{k}^T$$

the derivation of which is beyond the scope of this class.

Acquisition function

Now that we have a way of calculating the posterior, we need to decide which point we should evaluate next. This requires a search, which is performed using the "acquisition function". There are several options:

- Just take the value with the highest predicted value (highest mean). But this ignores the uncertainty.
- Something more sophisticated, like define the predicted "Improvement": $I(\mathbf{x}) = f(\mathbf{x}) - f_{\mathbf{best}}$. Thus, the "Expected Improvement" is

$$EI(\mathbf{x}) = E[\max\{0, I(\mathbf{x})\}] = \int_0^\infty Ip(I)dI = \Sigma^{\frac{1}{2}}(\mathbf{x})\left[u\Phi(u) + \phi(u)\right]$$

where $u = \dfrac{f(\mathbf{x}) - f_{\mathbf{best}}}{\Sigma(\mathbf{x})^{\frac{1}{2}}}$, and Φ and ϕ are the normal cumulative distribution and normal density function.

Acquisition function, continued

Let us choose the Expected Improvement as our acquisition function. Now what?

- We desire the point, $\mathbf{x}$, which maximizes $\boldsymbol{EI}(\mathbf{x})$.
- We will need to search the space of all possible values of $\mathbf{x}$ to find the $\mathbf{x}$ we desire.
- However, calculating $\boldsymbol{EI}(\mathbf{x})$ is cheap, unlike training our neural network.

Fortunately, all of this is simplified by using build-in sklearn functionality.

Gaussian processes, the code

```
# Learn_Parameters.py, continued

def get_next_param(model, f_train, bounds, n_restarts = 25):

  best_x = None;      best_value = 1;      n_params = bounds.shape[0]

  # Repeat a few times, starting at random locations.
  for point in npr.uniform(bounds[:,0], bounds[:,1], size = (n_restarts, n_params)):

    # Find the minima to get the next hyperparameter combination.
    res = sco.minimize(fun = expected_improvement, x0 = point.reshape(1, -1), bounds = bounds,
                       args = (model, f_train, n_params))

    # If it's the best yet, keep it.
    if res.fun < best_value:  best_value = res.fun;     best_x = res.x

  return [int(round(i)) for i in best_x]
```

Gaussian processes, the code, continued

```
# Learn_Parameters.py
import numpy as np, numpy.random as npr, sklearn.gaussian_process as gp, scipy.optimize as sco
from scipy.stats import norm as ssn

def expected_improvement(x, model, f_train, n_params = 1):

  # Initialize some parameters.
  new_x = x.reshape(-1, n_params);    best_f = np.max(f_train)

  # Get the mean and sd of the Gaussian for this x value.
  mu, sigma = model.predict(new_x, return_std = True)

  # Calculate the expected improvement.
  z = (mu - best_f) / sigma
  result = (mu - best_f) * ssn.cdf(z) + sigma * ssn.pdf(z)

  # Return the negative, since we're minimizing.
  return -result
```

Gaussian processes, the code, continued more

```
# Learn_Parameters.py, continued more
def call_nn(next_x, nn_func, best_f, best_x, X_train, f_train):

  # Train the NN with these parameters.
  next_f = nn_func(next_x)

  # Add them to the collection.
  X_train.append(next_x);   f_train.append(next_f)

  # Check to see if we have improvement.
  if (next_f > best_f):  best_f = next_f;  best_x = x

  # Return the current best.
  return best_f, best_x
```

Gaussian processes, the code, continued even more

```
# Learn_Parameters.py, continued more
def gp_search(num_iters, nn_func, bounds, num_presamples = 5):

  X_train = [];   f_train = [];   n_params = bounds.shape[0];   best_f = 0.0

  for params in npr.uniform(bounds[:, 0], bounds[:, 1], (num_presamples, n_params)):

    next_x = [int(round(i)) for i in params]
    best_f, best_x = call_nn(next_x, nn_func, best_f, best_x, X_train, f_train)

  model = gp.GaussianProcessRegressor(normalize_y = True, kernel = gp.kernels.Matern())

  for i in range(num_iters):
    model.fit(np.array(X_train), np.array(f_train))
    next_x = get_next_param(model, np.array(f_train), bounds, n_restarts = 100)
    best_f, best_x = call_nn(next_x, nn_func, best_f, best_x, X_train, f_train)

  return best_x, best_f
```

Gaussian processes, running the code

```
In [1]: import numpy as np, find_drumlins as fd, import Learn_Parameters as lp
In [2]:
In [2]: bounds = np.array([[5, 30], [20, 1500], [3, 10], [2, 4], [1, 4]])
In [3]:
In [3]: x_best, f_best = lp.gp_search(20, fd.solve_nn, bounds)
Determining the starting points.
Parameters [11, 203, 3, 2, 2] Score is [0.56549425492663896, 0.74847870243006742]
Parameters [29, 962, 8, 4, 3] Score is [0.5349482595316053, 0.75456389512783617]
⋮
Parameters [17, 322, 8, 3, 2] Score is [0.56482879856778934, 0.75253549756191318]
Starting optimization.
Parameters [12, 315, 6, 4, 2] Score is [0.55728807338111053, 0.75659229257285721]
⋮
Parameters [7, 1378, 5, 3, 3] Score is [0.48332941133158686, 0.78093306348483416]
Parameters [5, 1394, 7, 4, 1] Score is [0.56821845378159996, 0.75253549744101134]
In [4]:
```

Gaussian processes, the code, continued even more

Some notes about the code and example.

- In our case we had a range of values for each hyperparameter, and consequently had to search through the continuous space of possible hyperparameters to find the next best choice.
- Alternatively, you could have a finite set of possible values for each hyperparameter, and just search through that space for the next-best $\mathbf{x}$.
- If you persue the above approach (preselected possible hyperparameter values), which is a good one to start narrowing down your bounds, randomly put about 1% of the possible hyperparameter combinations in your starting set.

Other packages out there

There are several Python packages out there that help with hyperparameter optimization.

- sklearn.gaussian_process: as demonstrated here.
- Spearmint (`https://github.com/HIPS/Spearmint`): a package which implements a Bayesian optimization using Gaussian Processes. Appears to be set up to run on clusters.
- Optunity (`http://git.optunity.net`): a package which implements a Bayesian optimization using Tree-structured Parsen Estimators.
- auto-sklearn (`https://automl.github.io/auto-sklearn/stable`): claims to do everything under the sun. I haven't tested it.

There are likely others.

Linky goodness

Hyperparameter optimization:

- https://arxiv.org/abs/1609.08703
- https://arxiv.org/abs/1012.2599
- http://www.gaussianprocess.org/gpml/chapters/RW5.pdf
- http://mlg.eng.cam.ac.uk/pub/pdf/Ras04.pdf
- https://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf
- https://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf