Multilayer Perceptron Exercises

Input/output mapping

Single neuron models

Load each of the `data/mlp.*_neuron.naft` models and investigate visually the effect of changing the weight and bias parameters:

- What happens when the weight is increased?
- What happens when the bias is increased?
- Create a schematic drawing for each transfer function.

Linear hidden layer model

Load the `data/mlp.linear_hidden_layer.naft` model and set its parameters manually, or randomly:

- Is there any advantage of using *multi-layered* linear models over a single linear neuron?
- Is there any advantage of using *multi-layered* non-linear models over a single non-linear neuron?

Tanh hidden layer model

Load the `data/mlp.tanh_hidden_layer.naft` model.

Activate one neuron in the hidden layer by setting the weight from `input` to `hidden tanh neuron` to 1, and from `hidden tanh neuron` to `output linear neuron` to 1:

- What is the effect of changing the bias from `input` to `hidden tanh neuron`?
- What is the effect of changing the bias from `hidden tanh neuron` to `output linear neuron`?

Play around with its parameters to produce a Gaussian-like function. You can do this with only 2 neurons from the hidden layer.

Deep models

Draw on paper an *deep neural network* with:

- 1 input

- 4 hidden layers, 3 neurons per layer

- 1 output neuron
- How many parameters does such a model have?
- Validate your computation by loading the `data/mlp.tanh_deep_network.naft` model. The `ai.app` displays the number of weights the model contains.

Data cost and model complexity

Parameter norm

Load `data/mlp.tanh_deep_network.naft` and initialize its parameters randomly, making the output is visible. Next, use the weight scaler to increase/decrease all the weight and bias parameters together:

- What is the effect of increasing weights?

Data and weight cost

Load the `data/data.noisy_linear.naft` dataset and `data/mlp.linear_neuron.naft` model. Try to change the parameters to create the best fit for the data:

- What is the minimal cost you can achieve? Note it down.
- What happens with the `data cost` and `cost` curve when you do this?
- What happens with the `weight cost` when you increase parameters?
- What happens with the `data cost` when you change the `Output stddev`?
- What happens with the `weight cost` when you change the `Weights stddev`?

Repeat the same exercise for `data/data.noisy_sine.naft` and `data/mlp.tanh_hidden_layer.naft`:

- What is the minimal cost you can achieve? Note it down.

Exhaustive learning

Exhaustive learning will change selected model parameters independently, effectively walking a grid. To use the exhaustive learning mode, you have to:

- Load a model
- Load a dataset
- Select `Exhaustive` learning mode. When learning is not yet activated, selecting `Exhaustive` learning mode will add checkboxes next to the model parameters. These checkboxes allow the user to select the parameters that should be learned.
- Set the desired number of points to scan.
- Enable the `Learn` checkbox. Once exhaustive learning search the grid for the optimal parameters, it will switch into `NoLearn` learning mode.

Linear regression

Load the `data/data.noisy_linear.naft` dataset and `data/mlp.linear_neuron.naft` model. Use exhaustive learning to search the weight and bias space for this simple model.

- Can it do better than your manual learning above? Why?

Visualizations

Load the `data/data.noisy_linear.naft` dataset and `data/mlp.linear_neuron.naft` model. Use exhaustive learning one both parameters using 50 steps. Before starting to learn, specify the `Output filename` as `data.ssv`; this file will contain all the model parameters, the data cost, weight cost and total cost during the learning process. The last line will contain the optimal parameters found by the exhaustive search learner.

Install `gnuplot` and use it to visualize:

- The grid search exhaustive learning performs

  ```
  plot "data.ssv" using 1:2 with lines, "data.ssv" using 1:2
  OR
  plot "data.ssv" using 1:2 with linespoints
  ```

- The shape of the weight cost

  ```
  set dgrid3d 50,50
  splot "data.ssv" using 1:2:4 with pm3d
  ```

  What is the shape of the weight cost?

- The shape of the data cost

  ```
  set dgrid3d 50,50
  splot "data.ssv" using 1:2:3 with pm3d
  ```

Line search

Load the `data/data.noisy_sine.naft` dataset and `data/mlp.tanh_hidden_layer.naft` model. Find a reasonable random starting point and perform a line search for all parameters:

- Select exhaustive learning with 100 steps
- Learn the weight of the first hidden neuron
- Learn the bias of the first hidden neuron
- Continue for all hidden neurons and the output neuron
- Note the cost

- Do a 2-dimensional exhaustive learning on the weights of the first and second hidden neuron:
    - Visualize the total cost wrt these 2 parameters
    - Measure how long it takes to learn these 2 dimensions
- Based on your timing from above, how long would it take to learn *all* parameters together using exhaustive learning?

Metropolis

The Metropolis-Hastings algorithm creates a sequence of model parameters that will converge to the *a-posteriori* distribution $p(weights|dataset, outputstddev, weightsstddev)$. This distribution over the model parameters favours those parameters that provide a good fit for the data while keeping the magnitude of the weights reasonable. The terms *good fit* and *reasonable magnitude* are controlled via `Ouput stddev` and `Weights stddev`.

The Metropolis-Hastings algorithm works as follows:

1. Suppose we are located in `weights`
2. Choose a random direction and step into that direction: `candidate = weights + direction`
3. Evaluate the a-posteriori distribution in `candidate`
4. If $p(candidate) >= p(weights)$, we accept `candidate` as the new `weights` always
5. Else, we accept `candidate` as the new `weights` only with probability $p(candidate)/p(weight)$
6. Goto step 2

Linear model

Load the `data/data.noisy_linear.naft` dataset and `data/mlp.linear_neuron.naft` model. Start the Metropolis-Hasting learning:

- What happens with the cost?
- Does the learning converge?
- What is the effect of the `motion stddev` parameter? Does the algorithm stay stable?

Hidden layer model

Repeat the above for the `data/data.noisy_sine.naft` dataset and `data/mlp.tanh_deep_network.naft` model.

- What is the effect of the `motion stddev` parameter? Does the algorithm stay stable?

Steepest descent

The steepest descent algorithm creates a sequence of model parameters based on *gradient* information. The gradient vector for a given `weights` position points into the direction where the cost decreases the most. The magnitude of the gradient vector indicates how much the cost decreases. For a multilayered neural network, the gradient can be computed efficiently using the *back-propagation* algorithm. This is the reason that most of the literature call the steepenst descent method also back-propagation learning.

The steepest descent algorithm works as follows:

1. Suppose we are located in `weights`
2. Compute the `gradient` of the cost, computed in `weights`
3. Reduce the `gradient` L_2-norm to `max norm`, if needed: $gradient = gradient(maxnorm/L_2norm)$
4. Step into the gradient direction, with a given step-size: `candidate = weights + gradient*stepsize`
5. Accept the `candidate` always
6. Goto step 2

Hidden layer model

Load the `data/data.noisy_sine.naft` data and `data/mlp.tanh_hidden_layer.naft` model. Learn it via steepest descent:

- What happens? Why?
- How can this start-up problem be fixed?
- Plot the path followed by the weights of the weights of the first 2 hidden neurons
- Plot the data cost per iteration
- Plot the weights cost per iteration
- Note the minimal cost found by this algorithm

Repeat the above, but set the steepest descent step size to maximum.

- What happens?
- Plot the path followed by the weights of the weights of the first 2 hidden neurons: what do you see?
- What is happening?
- Reduce the step size until behaviour is normal again

Deep network

Load the `data/data.noisy_sine.naft` data and `data/mlp.tanh_deep_network.naft` model. Learn it via steepest descent:

- If the algorithm starts to oscillate, reduce the `step size`
- If the algorithm jumps from time to time, causing a drastic cost increase, reduce the `max norm`
- Can you find a setting that always works

Scaled conjugate gradient

The scaled conjugate gradient learning algorithm takes, similar to the steepest descent method, steps downhill the cost landscape. Instead of taking steps as steep downhill as possible, conjugate gradient methods assume the cost landscape is quadratic. This allows for a more optimal strategy which takes the curvature into account as well. This allows conjugate gradient methods to find the minimum of a quadratic surface on n steps, where is n is the dimensionality of the parameter space. Especially in long, narrow valleys will this improve the learning performance.

As opposed to Newton and Quasi-Newton optimization methods, conjugate gradient methods do not require the computation or explicit storage of the Hessian matrix, the matrix with second-order derivative information. Especially for optimization problems with many parameters, this is a huge advantage that allows the memory complexity to remain $O(n)$ in stead of $O(n^2)$.

Finaly, the *scaled* conjugate gradient method will automatically switch between steepest descent and conjugate gradient: when the Hessian is positive definite, conjugate gradient is used, else, steepest descent is applied. A positive definite Hessian matrix indicates that the cost landscape has a *bowl-like* shape, as opposed to a saddle point. If the Hessian is *not* positive definite, the cost landscape basically has no well-defined minimum, hence the transition to steepest descent behaviour.

Hidden layer model

Load the `data/data.noisy_sine.naft` data and `data/mlp.tanh_hidden_layer.naft` model. Learn it via scaled conjugate gradient:

- What happens? Why?
- How can this start-up problem be fixed?
- Plot the path followed by the weights of the weights of the first 2 hidden neurons
- Plot the data cost per iteration
- Plot the weights cost per iteration
- Note the minimal cost found by this algorithm

Deep network

Load the `data/data.noisy_sine.naft` data and `data/mlp.tanh_deep_network.naft` model. Learn it via scaled conjugate gradient:

- Change the `Output stddev` during learning. What happens to the `data cost` and the input/output mapping?
- Change the `Weight stddev` during learning. What happens to the `data cost` and the input/output mapping?
- Set the `Output stddev` to minimum, and `Weight stddev` to maximum. What happens to the generalization power of the network.

Load the `data/data.noisy_sine.naft` data and `data/mlp.relu_deep_network.naft` model. Learn it via scaled conjugate gradient:

- Note the minimal cost found by this algorithm

ADAM

The ADAM algorithm performs steepest-descent-like steps, but automatically controls the step size for each parameter individually. The `alpha` parameter controls all step sizes together, while the `decay` parameter controls how fast the individual step sizes can change (exponential decay parameter).

Deep network

Load the `data/data.noisy_sine.naft` data and `data/mlp.tanh_deep_network.naft` model. Learn it via ADAM:

- Change the `ADAM step size` during learning. What happens?
- Change the `ADAM decay` during learning. What happens?

Load the `data/data.noisy_sine.naft` data and `data/mlp.relu_deep_network.naft` model. Learn it via ADAM:

- Note the minimal cost found by this algorithm

2D-2D models

When switching from 1D-1D models to 2D-2D models, restart the `ai.app`. 2D data is indicated via green arrows: the position is the input, the direction of the arrow is the output. The network input/output mapping is represented via red arrows. Note that these red arrows are represented a bit smaller than the green arrows to keep the visualization clear enough.

Deep network

Load the `data/data.circle.naft` dataset and any of the `data/mlp.25552.naft` model. Try to learn it using a variety of learning algorihtms.

Load the `data/data.two_circle.naft` dataset and any of the `data/mlp.25552.naft` model. Try to learn it using a variety of learning algorihtms.

Auto-encoder

This multi-layer perceptron model consists of several hidden layers, where one layer has a smaller number of neurons than the input/output layer. When such a model is fed with the same output values as the input values, the model is forced to convert the input representation into a different, lower dimensional representation. This is often a more robust representation that has a higher abstraction, which can be useful.

Load the `data/data.circle.naft` dataset and any of the `data/mlp.2551552*.naft` models. Try to learn them using a variety of learning algorihtms.

- What representation could the network use?