3rd Baltic-Nordic Summer School on Neuroinformatics (BNNI 2015)

Practice on Artificial Neural Networks

Tambet Matiisen, Raul Vicente, Zurab Bzhalava

June 14, 2015

The 2014 Nobel prize in Physiology was awarded to Dr. John M. O'Keefe, Dr. May-Britt Moser and Dr. Edvard I. Moser for discovering particular cells in the brain that provide the sense of place and navigation. In this practice session, we are going to use computational approach to study the animal "GPS" system. In particular, we are going to use artificial neural networks to predict a rat's position based just on its hippocampal neural activity.

At first we divide the experimental arena into 16 blocks and approach this as a classification problem. Next we make the task harder by trying to predict rat's position (X and Y coordinates) directly, i.e. address this as a regression problem. Finally we automate the tedious task of hyperparameter optimization using Whetlab.

The data we use is multi-neuron electrophysiological recordings from the Buzsaki lab in New York. The data has been preprocessed as number of spikes in 200ms time window.

1 Technicalities

You can use either Matlab or Octave to do the practice exercises. Matlab is faster and more user-friendly, but Octave is free and open-source, and has been catching up with Matlab lately. Be sure to install at least version 3.8 of Octave, which includes experimental GUI.

- **Ubuntu:** `sudo apt-get install octave`. To start the GUI run this from command line: `octave --force-gui`. Then you can lock the application to launch bar.
- **Windows:** the most current version can be downloaded from here: `https://ftp.gnu.org/gnu/octave/windows/`.

For Octave you will also need to install packages `statistics` and `nan`, which include some basic machine learning tools we use for calculating baseline performance.

- **Ubuntu:** `sudo apt-get install octave-statistics octave-nan`
- **Windows:** run this from Octave command line: `pkg install -auto -forge statistics`. Unfortunately the latest `nan` package does not compile under Windows, so you have to download version 2.7.1 manually from here: `http://sourceforge.net/projects/octave/files/Octave%20Forge%20Packages/Individual%20Package%20Releases/`, and install it from Octave command line with `pkg install -auto nan-2.7.1.tar.gz`.

If you are new to Matlab/Octave, then these are excellent tutorials to start with:

- `https://en.wikibooks.org/wiki/Octave_Programming_Tutorial`
- `https://www.youtube.com/playlist?list=PLZ-E1VZLTeHMtUopmGx99KS775aHYP3e4`

Finally, for artificial neural networks we are going to use DeepLearnToolbox by Rasmus Berg Palm, which can be found from here: `https://github.com/rasmusbergpalm/DeepLearnToolbox`. It is also included with the tasks, so don't worry downloading it.

2 Artificial Neural Networks

Tuning artificial neural networks can be frustrating experience. Many people have tried and given up before they start to get any useful results. In this practice session we hope to give you step-by-step guidelines, how to tune the learning parameters of neural network and squeeze out maximum performance from them. But before that we need to have the basic understanding how artificial neural networks work.

Artificial neural network consists of interconnected nodes, often called "neurons". Some of those nodes are input nodes – they receive information from the outside world. Some of them are output nodes – they provide the results of the calculation. Finally there are hidden nodes, which are just intermediate steps of computation.

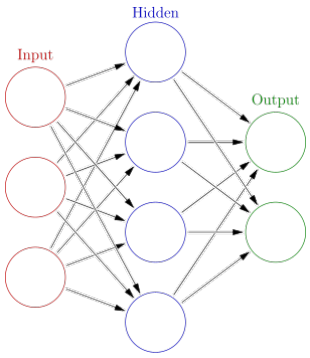


Figure 1: Artificial neural network with one hidden layer.

Nodes of artificial neural network are organized into layers. There are no connections between nodes in the same layer, only between layers – input layer nodes are connected to first hidden layer nodes, first hidden layer nodes are connected to the second hidden layer nodes and so on until last hidden layer nodes are connected to the output layer. In simplest feed-forward neural networks, the connections are in one direction only – from input towards output.

While artificial neurons are inspired by their biological counterparts, they are just cartoonish simplifications of actual neurons. Typical artificial neuron just multiplies its inputs with respective weights, sums the results and applies activation function.

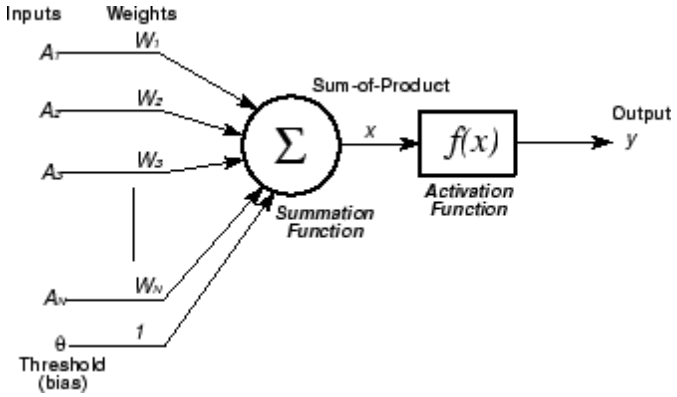


Figure 2: Artificial neuron.

Common activation functions used are $sigmoid(x) = \frac{1}{1-e^{-x}}$ (squashes any real value between 0 and 1), $tanh(x) = \frac{sinh(x)}{cosh(x)}$ (squashes any real value between -1 and 1) and $ReLU$ (rectified linear unit, $f(x) = max(x, 0)$). The purpose of activation function is to add non-linearity into calculations.

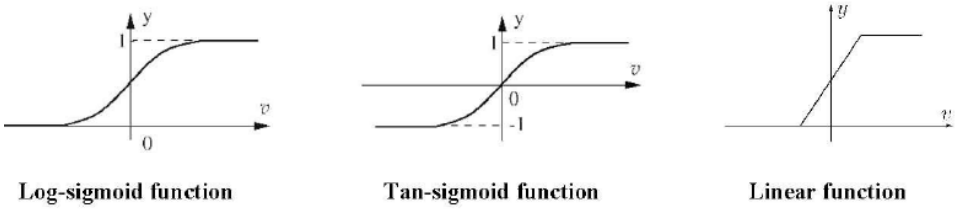


Figure 3: Common activation functions.

The beauty of artificial neural networks is, that they can learn to map inputs to outputs, given enough examples. The learning is accomplished by changing weights between neurons. The universal approximation theorem states, that any function can be approximated to given precision using artificial neural network with one hidden layer. In practice this result is irrelevant, because the claim that function can be represented by network does not say anything if the function can be learned by the network. Still the thing to remember from this is, that artificial neural networks are nothing more than universal function approximators.

How the learning works in artificial neural networks? It all starts with loss function, which measures how good networks' prediction is, how well its outputs match the expected values. Small loss is good, big loss is bad. The most common loss function is mean squared error (MSE) – you just square all differences between predicted and expected values, sum over all output nodes and take mean over training batch. It is easy to see, that better predictions result in lower loss values with MSE. If ouputs of your network are probabilities, then you would use different loss function (cross-entropy loss), but the general idea stays the same.

To learn you would need to change the weights in a way, that loss function decreases. How do you do that? The most common method is gradient descent – you take partial derivative of the loss function with respect to each weight and this gives you estimate, how much you need to change this particular weight to make loss function go down. Gradient is nothing more, than vector of loss function partial derivatives with respect to each weight. To calculate those derivatives efficiently in artificial neural network there is algorithm called backpropagation. Basically it applies chain rule to produce derivatives of bottom layers from derivatives of upper layers. We are not going to dwelve into details here, just assume that somehow you know the gradient, how much you should add to or subtract from each weight to make the loss decrease.

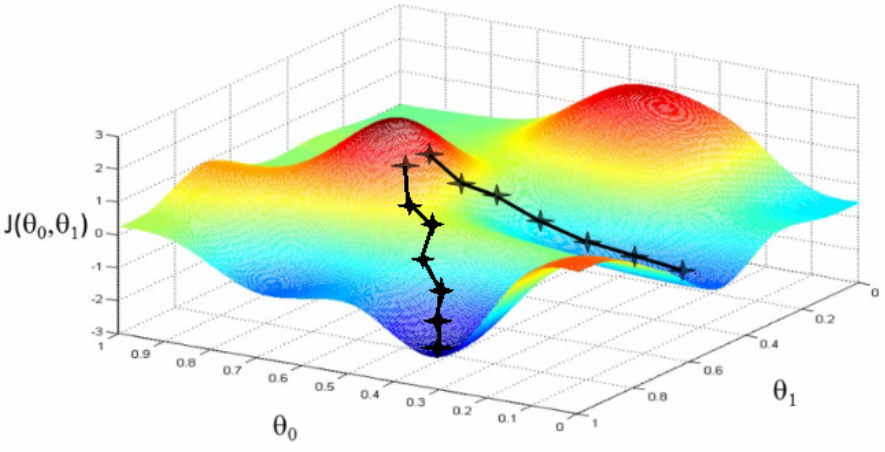


Figure 4: Gradient descent.

Complex nature of artificial neural network means, that there is no one set of weight values, for which loss function has minimal value, i.e. it is non-convex optimization problem. Often

the metaphore "loss landscape" is used – if you would plot loss value over many weight value combinations, then this would form landscape of many hills and valleys. If you just follow the gradient (steepest descent), then it is easy to get stuck in local minima (shallow valley). People have come up many techniques to work around this issue, but in principle it remains unresolved. Good news is, that many local minimas are pretty good and there are plenty of them.

3 Classification

In this exercise we divide the arena into 16 regions and try to predict the region where the rat is from activity of 61 neurons in hippocampal area of the brain.

Start by running the code in `classification.m` and observing the results:

1. First you should see baseline performance of linear classifier. This is there to just remind you, that artificial networks are heavy artillery and should be used only if simpler methods do not give satisfactory results.
2. Then you should see training progress of artificial neural network. You can ignore the command-line output and concentrate on the two figures shown during training:
   - The first one shows loss function value for both training and validation set. In this case we are using cross-entropy loss function. Its value is actually meaningless, we are more interested in the dynamics – does it decrease, how fast, is it stable?
   - The second figure shows misclassification rate – the fraction of training samples, for which the predicted class was wrong. You can get accuracy by subtracting this number from 1. Again, both training and validation set error rates are shown.

   Usually those two graphs are correlated, so it suffices to only pay attention to the latter. Also you care more about the validation set performance, than training set performance.
3. After that you should see accuracy of neural network printed out. At first it might not be much better than linear classifier.
4. Finally confusion matrix is shown. Confusion matrix shows common errors – which classes were mistaken for other classes.

Your job is to tune this neural network to have accuracy at least 70%.

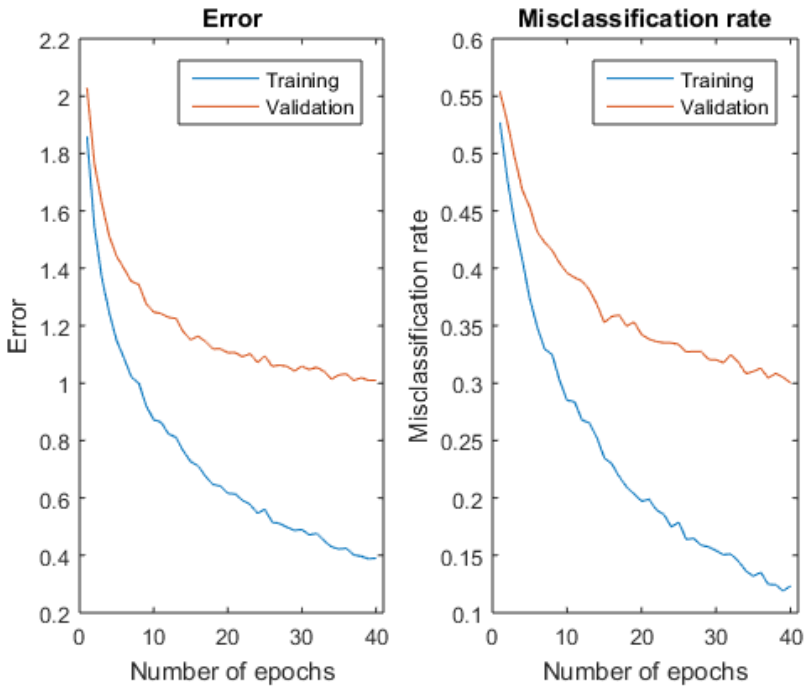


Figure 5: Example of network achieving 70% accuracy.

1. First you should start by finding optimal **learning rate**. Learning rate is the one most important learning parameter, it determines the step size on loss landscape.

   - Too big learning rate – you step over valleys and never into them.
   - Too small learning rate – you end up in closest valley, that might not be the deepest.

   Usually you try learning rate in decreasing powers of 10, i.e. 1, 0.1, 0.01, 0.001 and so on. Repeat until graphs are stable and validation accuracy does not improve any more. Basically you want to use the highest learning rate, that does not blow up your loss.

2. Then you might want to increase the **number of hidden nodes** to see if validation accuracy improves. More hidden nodes means the more complicated functions the network can express. Usually you do it in powers of 2, i.e. use twice as much hidden nodes as previously.

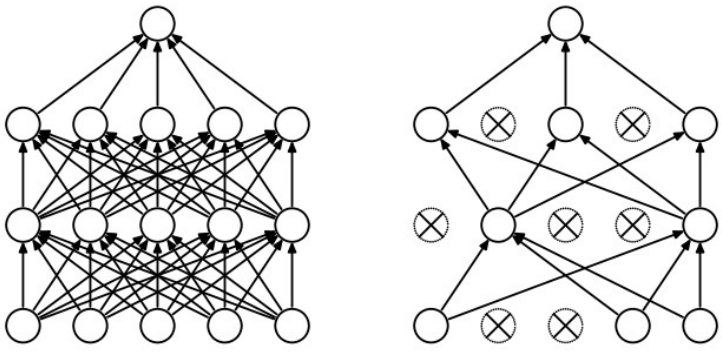
3. After that you should use **momentum** to speed up learning. Momentum adds inertia to weight updates – each update also includes some fraction of previous weight update, forcing faster movement on long descends in one direction. Sometimes momentum may overshoot resulting in zig-zags on graphs, but usually it helps to reach to minima faster. Usual values to try are 0.5, 0.9 and 0.95.

4. By now you should observe massive overfitting – training set error is much smaller than validation set error. Basically network memorizes the training set and does not generalize to the validation set. There are several ways how to deal with overfitting:

   Get more training data. Data is the most effective regularizer, so if possible you should always use that option. The reality is, that often it is not easy to get additional data, and then the following methods come to play.

   Use weight decay. This is well-known technique, that forces weights to have small values. It is implemented by adding regularization term to loss function, which results in weights decreased by some fraction of previous weight value during each training iteration (hence "weight decay"). You should start with low values of 10^{-4} and go up in powers on 10 until validation set accuracy does not improve any more.

   Use dropout. Dropout disables some subset of hidden nodes during training. This forces nodes to be more individualistic, because they are not able to rely on others. During testing hidden node activations (values) are multiplied by the same fraction to produce on average the same amount of input to next layer nodes. This functions also as a crude approximation to ensemble method – basically we are averaging results of many networks with shared weights. Common dropout fraction to use is 0.5, but in some cases lower values might work as well.



(a) Standard Neural Net (b) After applying dropout.

Figure 6: Dropout.

Finally, some amount of overfitting is inevitable and healthy with neural networks. So as long as validation set accuracy improves, do not worry too much about it.

5. Once you think your learning parameters are good enough, you should try to learn longer – increase the **number of epochs**. One epoch is full training session over entire training set. Usually loss function plateaus at some point and there is no point training any further. You should figure out where this point is by trying say 100 epochs. Then for further finetuning you can use the number of epochs before plateau is reached.

6. As gradient descent explores the loss landscape, there might be gradually more fine-grained valleys to descend to. With initial learning rate you may just step over those valleys or step out of them. For that reason it is usually good idea to try smaller learning rate just before loss function has plateaued. Usually you do it in powers of 10, i.e. multiply learning rate by 0.1. But DeepLearnToolbox applies scaling rate after each epoch, so you need to calculate approximate per-epoch scaling rate, that results in 0.1 multiplier after n epochs. This can be calculated by

$$scaling\_learningRate = \sqrt[n]{0.1}$$

Mind that there is delicate interplay between all the parameters – if you increase number of nodes, you might need to lower learning rate, if you increase weight decay, you might need to also increase learning rate, dropout usually needs bigger number of nodes and so on. Previous rules of thumb do not guarantee you the best results, they just help you to find quickly ranges of parameters that work, but from that point on you are on your own.

Things you should not invest too much effort into:

- Batch size. In stochastic gradient descent (SGD) you divide training set into batches of random samples and train one batch at a time. You use average gradient of all samples in batch to perform the weight update. If batch size is too small, your gradient may not point towards minima for entire training set and gradient descent will walk around randomly. If batch size is too big, you always step to right direction, but each training iteration is just very time consuming. Use smallest batch size, that is sufficiently representative of training set distribution. Often this is more restricted by the amount of memory your GPU has or how many parallel computations it can handle.

- Activation function. For small shallow networks use *tanh*. Modern deep networks use *ReLU* to combat the vanishing gradient problem and maximize performance. But this is not so much issue with small networks.

- Depth of the network. While hierarchical representation of features is the main selling point of artificial neural networks, the benefits of hierarchy show up only with more complex networks architectures like convolutional networks. These networks make use of the locality property of information. For small fully-connected networks usually one to three layers is enough.

4 Regression

In this exercise we try to predict rat's location directly. This time we are going to use activity of all 70 neurons as input. Outputs of the network are the X and Y coordinates of the rat.

Start again by running the code in `regression.m` and observing the results:

1. First you should see baseline performance of linear regression. The result is given as an average Euclidian distance between predicted and actual rat location.

2. Then you should see training progress of artificial neural network. This time there is only one figure – the loss. For regression we are using the classical mean square error loss (MSE). Again you should pay more attention to validation set loss than training set loss.

3. After that you should see the performance of neural network printed out. Again, at first it might not be much better than linear regression.

4. Finally 20 examples of predicted and actual locations are shown. You can tune the number of examples shown or duration of pause between them.

Your job is to achieve as good average distance as you can!

Use all the tricks you learned from previous exercise. As dataset is bigger this time, you might initially want to choose smaller subset of samples to work with. This way training completes in minutes and you can iterate fast. Once you have successfully overfitted the smaller dataset, try full dataset and see if performance on validation set improves.

5 Whetlab

As you probably saw with previous exercises, choosing the right hyperparameters can be tedious task. One of the methods that can help in this situation is Bayesian optimization. This is a specialized technique for finding the maximum values of functions that are very challenging to evaluate, due to their being very expensive or noisy. Hyperparameter optimization – where you might need to train a machine learning system for hours or days – is exactly such a difficult problem. But applications are not limited to machine learning – you can use it also to suggest new settings for lab experiments or for tuning marketing tactics. Whetlab is a startup by top machine learning experts, that makes using Bayesian optimization especially easy. Unfortunately it works only in Matlab.

Start by going over the code in `whetlab_regression.m`:

1. First half should be familiar already from previous exercise.

2. Then the hyperparameters for Whetlab optimization are defined. **Your job is to fill in the ranges of those parameters based on the experience from previous exercises!** Notice how we use log scale for learning rate and hidden nodes.

3. After that new experiment is created using `whetlab()` function. **Make sure you include your name in the experiment name!** For first run, the resume option must $false$, but for next runs it must be $true$.

4. Next 10 experiments are run using hyperparameter values suggested by Whetlab. You can first run the commands in that loop manually and observe the results.

5. Once you have made sure the code works as expected, you can run it for 10, 20 or 40 loops. While computer is crunching the numbers, you might want to read short introduction to Bayesian optimization from here: `https://www.whetlab.com/technology/`.

6. Finally the code plots out two graphs of the results. Whetlab homepage includes many more useful visualizations, but for that you have to sign up for closed beta yourself: `https://www.whetlab.com/beta-signup/`.

That's all, hope you enjoyed it!