Introduction to Python, machine learning and data handling for the physical sciences

Mattias Ermakov Thing

November 30, 2023

1 Week 50

1.1 Intro

This week the topic is neural network and how to get started using the Python package `keras` [2], which is a simply interface extending the underlying package `tensorflow` [1].to get started you can always look at the official documentation:

- `https://keras.io/getting_started/`
- `https://www.tensorflow.org/learn`

Neural networks belongs to a separate tool of machine learning and there are many ways to use neural networks and how to train them. The most classical approach to using and training neural networks is via supervised learning. In this case you have some training dataset $\{X,Y\}$ of inputs and outputs. Since an arbitrarily large deep neural network can learn any function f [**?**], we train a neural network to imitate any arbitrary function:

$$X \to f(X) = Y. \tag{1.1}$$

This method is straight forward if you have a dataset. The problem is just that you might not have enough data, or no data at all. In this case things get more complicated and there are other methods to try such as genetic learning methods. In some cases we can also do the training unsupervised, but these advanced training methods are beyond this course.

1.2 Neural network

As the name suggest neural networks are composed of neurons similar to our brain. Any neuron network uses the same basic building block, the single neuron. In some applications such as in robots less than 100 may be sufficient [3], whereas image classification networks may contain thousands of neurons [4]. Thus, to understand neural networks we should take our time to understand how they work. Neurons are fundamentally doing linear regression, and later we discuss how nonlinear behavior is obtained as this is key to the power of neural networks. As a simple example, one can consider

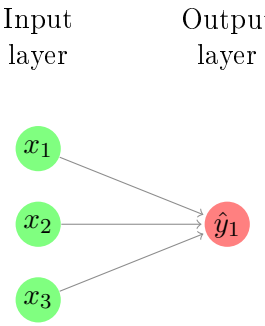

Figure 1: *A simple neural network with an input layer, and a single neuron in the output layer.*

a single neuron "network" shown in Figure 1. This "network" takes in three inputs, x_i, where $i = 1,2,4$, and computes a predicted value $\hat{y}$. The word network is in quotation as one would need multiple interconnected neurons to properly talk of a network.

In the case of this single neuron with three inputs shown in Figure 1 the equation becomes,

$$z(x_i) = w_1x_1 + w_2x_2 + w_3x_3 + b, \tag{1.2}$$

where each input is associated with the weight w_i which modulates the influence of the input, and then there is a bias term, b. This can be written in a compact matrix form as,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b \end{bmatrix} = w_{ij}x_i + b_j \cdot \vec{1}, \tag{1.3}$$

where an additional j index is added to indicate the number of neurons, but since there is only one neuron it is redundant. The use of this index becomes apparent when considering a network like in Figure 2, but with two fully connected output nodes, $j = 2$. In this case, the equation would be,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = w_{ij}x_i + b_j \cdot \vec{1}. \tag{1.4}$$

This highlights the benefit of the notation with the weight matrix as w_{ij}, the input vector as x_i, and the bias vector as b_j. A technical note would be that, in practice, it is more efficient to skip the process of adding the bias terms explicitly. One can exploit the definition of matrix multiplication and write it as,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} & b_1 \\ w_{12} & w_{22} & w_{32} & b_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix} = w_{ij}x_i + b_j \cdot \vec{1}. \tag{1.5}$$

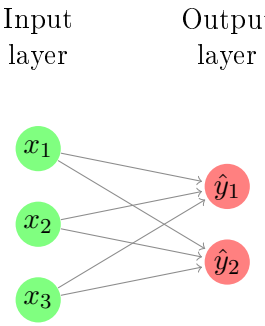

Figure 2: *A simple neural network with an input layer, and two neurons in the output layer.*

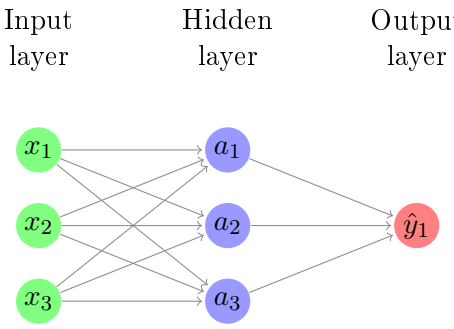

Figure 3: *A simple neural network with an input layer, a hidden layer, and a single neuron in the output layer.*

From this we can then also consider more complex networks with hidden layers as in Figure 3. In this case we would first compute the first the values at the hidden layers $z_1(x_i) = a_i$ and then we can use that result to compute that result $z_2(a_i) = \hat{y}$.

A linear equation is not sufficient to produce complex nonlinear decision boundaries, therefore it is key to modify the result $z_j(x)$ such nonlinearity is achieved. This is very simply to prove, consider our simple network with one hidden layers. The math becomes,

$$\hat{y} = z_2(z_1(x_i)) = w_{i2}\left(w_{i1}x_i + b_1 \cdot \vec{1}\right) + b_2 \cdot \vec{1}. \tag{1.6}$$

The problem here is that while we have different weights and biases, due to the lack of nonlinearity, this hidden actually achieve nothing because we can redefine this equation to be just like a network without a hidden layer,

$$\hat{y} = z_2(z_1(x_i)) = \underbrace{w_{i2}w_{i1}x_i}_{w_{i1}x_i} + \underbrace{w_{i2}b_1 \cdot \vec{1} + b_2 \cdot \vec{1}b_1 \cdot \vec{1}}_{b_1 \cdot \vec{1}}. \tag{1.7}$$

This is perfectly valid because the latter terms is nothing but a sum of trainable variable, which is nothing but a number, and then first part contains the input times two different weight, but again we can just redefine that as a single weight. Therefore, without any nonlinearity we can add an arbitrary number of hidden layers and achieve the equivalent of having no hidden layers.

This is fixed using nonlinear activation functions, $a(z)$, such as the sigmoid function, $\sigma(x)$, and the rectifier linear (ReLu) function, ReLu(x). There are many more options, and it is up to the designer of the network to determine the best activation function. The activation function, f, can be applied on a per-neuron basis, but it is commonly applied across the layers,

$$a(z_j(x_i)) = f(w_{ij}x_i + b_j \cdot \vec{1}). \tag{1.8}$$

Each layer of the neural network is generally computed in this way. One can then stack the layers, $a(z_j)^n$, where n is the number of layers. Deep neural networks then have three layers or more, $n \geq 3$. The output of the neural network is then computed by iteratively computing the layers from input to output, with the previous layer being the input to the next layer.

1.3 Supervised learning

The first step in training a network is to determine the error of the network. The goal is generally to minimize the error that the network is making. For this, one considers the loss function, $\mathcal{L}_{loss}$, which is a function of the network prediction, $\hat{Y}$, which is the result of the last layer of the neural network. In supervised learning, you have the input X and output Y, which you have labeled to be something specific. Since the goal is to mimic this result that you have determined to be correct, we are thus interested in comparing the two. There are many options comparing which can be chosen as the loss function. Popular choices include the root mean squared error (RMSE), mean squared error (MSE), and mean absolute error (MAE). As an example, one can write the loss function of the MSE as,

$$\mathcal{L}_{loss}\left(\hat{Y}, Y\right) = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{Y}_i - Y_i\right)^2, \tag{1.9}$$

where n is the number of entries in $\hat{Y}$ and Y. On top of the loss function, one can consider additional regularization or other custom terms, which can be included in the cost function that is used for training. An example of a regularizer is the L2 regularizer which is the sum of squares of the weight, which naturally penalize large weight and this can help against over-fitting. In this case, the total cost function with the MSE as the loss function is,

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}_{loss}\left(\hat{Y}_i, Y_i\right) + \frac{\lambda}{2k}\left\|\tilde{W}\right\|_2^2, \tag{1.10}$$

where θ represents the trainable parameters, m is the batch number, k is the number of regularized weights, and λ is the strength of the regularization term. Note that here $\tilde{W}$ is the vector of regularized weights.

Since the cost function is a function of the result of the network, and by extension its trainable parameters, it is possible to do gradient descent to train the network. In literature, the cost function is often called a loss function, and this term is often used interchangeably. Sometimes the loss function is considered as a single term that measures a loss, thus, one can

have multiple loss functions and regularizer terms in the cost function, where the total function which the network uses for training is the cost function. For a simple training rule, where one only uses a single loss function, the loss and cost function will be the same.

While the general concept of training involves some form of gradient descent, there are many optimizers available such as RMSProp, Adam, AdaGrad etc. which are modified gradient descent algorithms that might include effects such as momentum which is where the effect of the previous step is also part of the update. The network is then trained across a certain number of epochs, which is the number of times the network is training on the full data set. One may use batch sizes to evaluate a subset of input and output values before making the next gradient and decent update. This can improve training performance.

Another key training hyperparameter is the learning rate, α. Note that hyperparameters are model parameters that are not trained, but it is up to the designer to choose them such as the learning rate, batch size, epochs, number of layers, etc. The learning rate decides how large steps the learning should take. A too-large learning rate might prevent the learning algorithm to find the desired minimum, and a too-low learning rate may cause the network to train for an unnecessarily long amount of time, but could also cause the network to get stuck in small local minima.

1.4 Iris classification

To show a simple use-case of a simple neural network with just one hidden layers, we can try to classify iris flowers as we have a nice and simple dataset for that. In this dataset there are four input features and three different categories. There are many things we need to consider:

- Loading the data
- Covert the category to a number
- Consider normalization of the input
- Splitting the data into training, validation and test datasets
- Constructing the neural network
- Setting hyperparameters
- Training the network
- Evaluation of network performance

For this purpose you can find the example code for the iris network at: `https://github.com/CP3-Origins/FY555/blob/main/python/main.py`

When loading the data you should first understand the format of your data. Consider the file extension and find out how you load it in python. Then you can inspect it using commands like `print()`, `type()`, and `.shape` to learn about the structure of the data. You want to end up with a `numpy` array or a `tensorflow` tensor as your input and output.

Consider if any of you input and outputs are not numbers. Since neural networks are mathematical equations we need everything to be numbers. If you have categories that are not number, you need to encode them to a number. In the case of the iris dataset, we have three different types described by a name. So we need to encode these three categories to numbers.

Normalization is another thing to consider. Generally, this is not necessary as networks are sort of scale invariant. But since weights are generally initialized around one, we would also like to have our input to be around one, so it is often a good idea to normalize your input unless especially if your input values are far from $\mathcal{O}(1)$. Now there might be reasons not to normalize, but in many cases it might help the network perform better.

When training a network in a supervised manner we need to split of out dataset. We need data for the network to train on. This should be the bulk of the dataset. Now, we need some data that the network isn't trained on, because we want the network to learn an arbitrary underlying function and not just to fit to our training data. That's why we take away some of the data as validation data. This data is **not** used for training, but to check how our model is performing on unseen data. For good measure we can have another dataset called the test data. This is another small dataset which is unseen and can be used in the end to evaluate the trained model.

You might ask why do we need both a validation and test dataset? The validation data is used to fine tune hyperparameters and thus our model is biased based on both the training and validation data. The latter is key for us to ensure the selection of hyperparameters such that we are not over-fitting, but getting reliable result of unseen data. Therefore, in a true experiment you would want to save a test dataset for the end when you might have several different trained models, and then based on this final evaluation using the test data, we can choose the best model. How you split your dataset is up to you and it depends on how much data you have. In this example of the iris model start by taking away 20% for testing. Then we have 80% of which another 20% (16% of the total) are set aside for validation, with the remaining (64% of the total) being used for training.

Now you have your data and a strategy of how to use the dataset, you can construct your network. Here you need to consider the shape of the input and output. If your network doesn't match, then you will get an error. If you have four inputs, then you need your first layer to take four inputs, and likewise if your output has three categories, you need 3 nodes in your final layer. Now what you decide is how many nodes there are in the hidden layers, the number of hidden layers, the activation functions and other parameters beyond this course. These things are called hyperparameters

Hyperparameters is where practice, knowledge, and experience comes into play because you have to figure out these parameters. There is no go-to method of finding the right parameters. One strategy is to change one and see how that affects the result. Another strategy is to set hyperparamters at random, within some limits, and then simply log what random parameters worked best and use them in the end. Or one could use a combination of the two, or just try some other strategy. An experienced person might have an idea what could be a reasonable starting point, but there is no guarantee.

There are also other hyperparameters to consider such as learning rate, batch size, epoch, loss function, metrics etc.

Having build your network and set some initial values for the hyperparameters, it is time to train the network using training and validation data. This is how you test network performance, and you would want to train it every time you changed any hyperparamter to check how well this affects the performance. For classification we can often use accuracy as a metric to measure how often we classified correctly. This is the benchmark

One thing to consider is the issue of over-fitting, which is when your network begins learning your training data and not the underlying trend. You can detect such behavior if the accuracy or loss using the training dataset improves while it gets worse for the validation dataset. To help this you could train less or make the network small. There are other ways to deal with this like regularization which is beyond this course. In the end you can evaluate on your test data to determine how good your network performs.

References

[1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[3] Jan Nagler, Anna Levina, and Marc Timme. Impact of single links in competitive percolation. *Nature Physics*, 7(3):265–270, January 2011.

[4] Òscar Lorente, Ian Riera, and Aditya Rana. Image classification with classic and deep learning techniques, 2021.