

Exercise for Introduction to Machine Learning,
Advanced materials processing with intelligent systems

Explain intuitively why a thinner but deeper network (i.e., many layers, each with few parameters) should perform better than a shallower and thicker one (i.e., with few layers, but many parameters per layer) — given that they have the same number of parameters.

Intuitively, a thinner but deeper network should perform better than a shallower and thicker one because it is able to more easily learn hierarchical representations of the input data. In other words, the deeper network is able to learn more complex patterns in the data by building upon the simpler patterns learned by the earlier layers.

On the other hand, a shallower and thicker network with the same number of parameters may struggle to learn these hierarchical representations because it does not have the depth to build upon the learned patterns in a sequential manner. Instead, it may rely on the larger number of parameters in each layer to try to capture the patterns directly, which may not be as effective as learning them hierarchically.

It's worth noting that this is just an intuitive explanation and there are many factors that can influence the performance of a neural network, such as the type and quality of the data, the type of task being performed, and the optimization algorithm used to train the network. In some cases, a shallower and thicker network may even outperform a deeper and thinner one, but in general, a deeper and thinner network is likely to be more powerful and effective.

Normalization is the process of changing the signal's range to a predefined one (i.e., $[0,1]$, or $[-1,1]$). Mark the correct answer(s): 1. It should be avoided as it "destroys" the information about the amplitude 2. It helps gradient descent to converge faster 3. Allows better handling of multi-modal signals 4. It is not necessary if we use an optimization technique that adapts the learning rate per parameter.

It should be avoided as it "destroys" the information about the amplitude: False

Normalization does not destroy the information about the amplitude of the signal. It simply scales the signal so that it is within a specific range, which can often be beneficial for various reasons.

It helps gradient descent to converge faster: True

Normalization can often help gradient descent to converge faster because it scales the input data to a more consistent range. This can help prevent the learning process from getting stuck in local minima or saddle points, which can occur when the data is highly skewed or has large variations in scale.

Allows better handling of multi-modal signals: True

Normalization can also be helpful for handling multi-modal signals, as it can help reduce the impact of large variations in scale between different modes. This can make it easier for the model to learn patterns in the data and improve performance.

It is not necessary if we use an optimization technique that adapts the learning rate per parameter:
False

While it is true that some optimization techniques, such as Adam, adapt the learning rate per parameter, normalization is still generally considered to be a useful preprocessing step that can improve the performance of the model. Even with an optimization technique that adapts the learning rate per parameter, normalizing the input data can still be beneficial in many cases.

What downsides can normalization bring? Conversely, what are the advantages the standardization brings over normalization?

While normalization is generally a useful preprocessing step that can improve the performance of a model, there are some potential disadvantages to consider. For example, in the case of multi-channel signals, normalizing each channel independently can remove the information about the relative amplitudes of the different channels. This can be a problem if the relative amplitudes are meaningful and should be preserved, such as in the case of acoustic detection using different microphones at different positions. Removing this information can affect the performance of the model and may not be desirable in all cases.

It's also worth noting that normalization can be sensitive to the choice of the normalization range. For example, if the normalization range is too narrow, it may not capture the full range of the data and could potentially cause information loss. On the other hand, if the normalization range is too wide, it may not provide enough resolution to capture important patterns in the data.

In general, it is important to carefully consider the characteristics of the data and the requirements of the model when deciding whether to use normalization or any other preprocessing step. It may be necessary to try out different approaches and evaluate their performance to determine the best approach for a particular problem.

Normalization is sensitive to outliers. How can you make it more robust?

There are a few ways to make normalization more robust to outliers:

Trimming: One way to make normalization more robust to outliers is to use a trimmed mean for the normalization instead of the standard mean. To do this, you can remove a certain percentage of the data from the tails of the distribution before calculating the mean. This can help reduce the impact of extreme values on the normalization.

Winsorization: Another option is to use a technique called winsorization, which involves replacing the extreme values with less extreme values. For example, you could replace the highest and lowest 5% of the values with the next highest and lowest values, respectively. This can help reduce the impact of extreme values on the normalization without completely removing them from the data.

Robust scaling: Another option is to use a technique called robust scaling, which scales the data using the median and interquartile range instead of the mean and standard deviation. This can be more robust to outliers because the median and interquartile range are less sensitive to extreme values than the mean and standard deviation.

It's worth noting that these methods can also be combined, depending on the specific requirements of the problem. It may also be necessary to try out different approaches and evaluate their performance to determine the best approach for a particular problem.

The learning rate is probably the most critical hyper-parameters for most ML algorithms. Intuitively, a larger learning rate should be preferred to a smaller one for faster convergence. Nonetheless, large learning rates can make the optimization unstable and even leading to divergence. How can you combine the advantages of a large learning rate (fast convergence) with the ones of a smaller learning rate (stability)?

There are a few ways to combine the advantages of a large learning rate (fast convergence) with the stability of a smaller learning rate:

Adaptive learning rates: One option is to use an optimization algorithm that adjusts the learning rate adaptively during training. For example, algorithms such as Adam and RMSprop use a technique called "momentum" to adapt the learning rate based on the past gradients of the parameters. This can help the optimization algorithm converge faster and avoid getting stuck in local minima or saddle points.

Learning rate schedules: Another option is to use a learning rate schedule, which adjusts the learning rate over the course of training. For example, you could start with a larger learning rate and gradually decrease it as the optimization progresses. This can help the optimization algorithm converge faster at the beginning of training, while still maintaining stability as it approaches the end of training.

Batch normalization: Another technique that can help stabilize the optimization process is batch normalization, which normalizes the activations of the network across each batch of training data. This can help reduce the internal covariate shift and make the optimization more stable and less sensitive to the learning rate.

It's worth noting that these techniques can often be combined for even better performance. It may also be necessary to try out different approaches and evaluate their performance to determine the best approach for a particular problem.

A very popular optimization algorithm doesn't consider the magnitude of the gradient but only its sign. Think about how you can develop an optimization algorithm that doesn't consider the magnitude of the gradient but only its sign. What are the advantages?

An optimization algorithm that only considers the sign of the gradient, rather than the magnitude, would be a type of sign-based optimization algorithm. One way to develop such an algorithm would be to update the model parameters in the direction of the gradient, but only if the gradient is non-zero. If the gradient is zero, the model parameters would not be updated.

One advantage of this approach is that it can be computationally efficient, as it does not require calculating the magnitude of the gradient. It can also be less sensitive to the learning rate, as the update step is based solely on the direction of the gradient.

However, it is worth noting that sign-based optimization algorithms may not always converge as quickly as other optimization algorithms, as they do not take into account the magnitude of the gradient. In addition, they may be more sensitive to the initialization of the model parameters and may require careful tuning to achieve good performance.

As seen in class, the primary outcome of Deep Learning is to "automatically" learn to extract unconventional features from the raw data, which allows developing an end-to-end training approach (from raw data to outcomes). When can this technique be helpful, and when does it make sense to manually engineer some features to feed the model instead of using the raw data as input?

Deep learning is a type of machine learning that involves training neural networks on large datasets in order to automatically learn and extract features from the raw data. This allows for an end-to-end training approach, where the model can be trained directly from the raw data to make predictions or decisions.

This technique can be helpful in a number of situations, such as when the raw data contains complex patterns or relationships that may be difficult to manually identify and extract as features. Additionally, deep learning can be useful when there is a large amount of raw data available and it is not practical to manually engineer all of the features needed to train a model.

However, there are also situations where it may make sense to manually engineer some features to feed the model instead of using the raw data as input. For example, if the raw data is noisy or contains a lot of irrelevant information, manually engineering features may help to improve the performance of the model. Additionally, if there is domain knowledge that can be used to create more relevant features for the model, this can also be a helpful approach.

It is also worth noting that a careful feature extraction stage can help the model to generalize to data from different sensors. When a model is trained on data from a specific sensor or set of sensors, it may be less effective at making predictions or decisions when applied to data from different sensors. By carefully extracting features that are relevant and generalizable across different sensors, it is possible to improve the model's ability to generalize to data from different sensors. This can be especially important in applications such as robotics or Internet of Things (IoT) systems, where data may be collected from a variety of sensors and it is important for the model to be able to make predictions or decisions based on data from any of these sensors.

In summary, the decision of whether to manually engineer features or to use the raw data as input will depend on the specific problem being addressed and the available data. Carefully extracting relevant and generalizable features can be an important step in improving the performance and flexibility of a deep learning model, particularly when the model will be applied to data from different sensors.