Unit 7 – Deep Learning

So far we have studied how classical neural networks do their work. We have looked at individual neurons, looked at their computational process: taking a weighted sum of inputs, applying a non-linear activation function, and producing an output; and we have built and trained single layer, and two layer networks that are able to classify images.

We spent time looking at the error back-propagation method for training the network, and have explored how network performance changes as we increase the number of units available in hidden layers. Finally, we gained insight into the processing taking place across hidden layers of the network, and understood it as a form of ***feature extraction***. This process allows ***meaningful patterns*** to be learned progressively – from one layer to the next, providing a final output layer with a much richer, more informative, and easier to classify set of inputs.

This last observation motivates the idea that ***adding more and more hidden layers*** should greatly increase the ability of the neural network to carry out a particular task. In this section we will explore ***why*** this is the case in terms of the ***ability of the network to approximate an arbitrary function***, and we will explore the ***computational issues*** that arise once we start adding layers that are further and further away from the error signal used to train the network's weights.

Why Deep Learning?

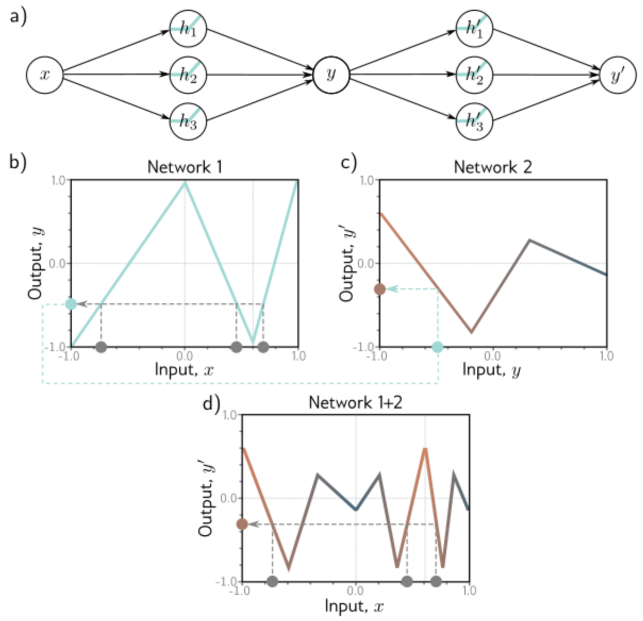
The important question is ***why does having a network with many layers produce better results***? To make the question more specific, consider this: If we have a fixed ***budget*** in terms of the number of units we can use to build our network, ***why does organizing these units into multiple hidden layers work better than having the same number of units in a single hidden layer?***

The answer lies in the power of ***composition*** of non-linear functions that occurs when the output of one hidden layer is used as input to another hidden layer. An excellent visualization of this from the book 'Understanding Deep Learning' by Simon Prince (https://udlbook.github.io/udlbook/) is shown below. The figure illustrates a simple, 2-hidden-layer network with one input ***x*** and one output ***y'*** (so this is a network intended for ***regression*** in 1-D).

The hidden units have a ReLU activation function, which is very common in deep networks. As a result, ***each unit learns to make a fold at a particular value for x, with a particular slope*** which is determined during training to whatever values help the network fit the input training data best. The key observation is that in ***1D each unit produces one linear fold*** working on ***whatever input it is given***.

Focus now on the ***3-unit layer immediately next to the input*** – because it contains 3 units, each producing a linear fold, the ***combined model for y*** (the intermediate value produced by the first layer) ***has 3 folds***. The second hidden layer, also with 3 units, takes as input the value ***y*** produced by the first layer and applies ***3 further folds*** to it. In 1D, this results in a final model for ***y'*** that has ***9 total folds***. This is more folds than are possible by a ***single-hidden-layer network with 6 total units***.

This is a small example, so the compounding effect looks small – but for a larger network with multiple inputs, the compounding effect results in a final model that quickly grows in complexity and power to closely fit very complex input datasets.



(Figure from ‘Understanding Deep Learning’ by Simon J.D. Prince, The MIT Press, 2023. – CC-BY-NC-ND license)

An estimate of the number of regions that can be generated by a deep network with ***K*** layers, ***P*** units per layer, working on an input with ***D*** dimensions is given by:

$$\prod_{i=1}^{K-1} \left\lfloor \frac{P}{D} \right\rfloor^{D} \cdot \sum_{j=0}^{D} \binom{P}{j} \qquad \text{[Montufar et al., 2014]}$$

The last part of the above formula, the sum, corresponds to the number of planar regions that can be generated by a one-hidden-layer network with ***P*** units. We can compare the maximum number of planar regions possible for networks with different topologies, and a quick check shows how quickly the number of regions the network can produce grows as we distribute the available units across more layers:

```
The maximum number of folds for a single-hidden-layer net with 40 units is 821
The max. number of folds that can be produced by a network with 4 layers and 10
units per layer is: 875000
```

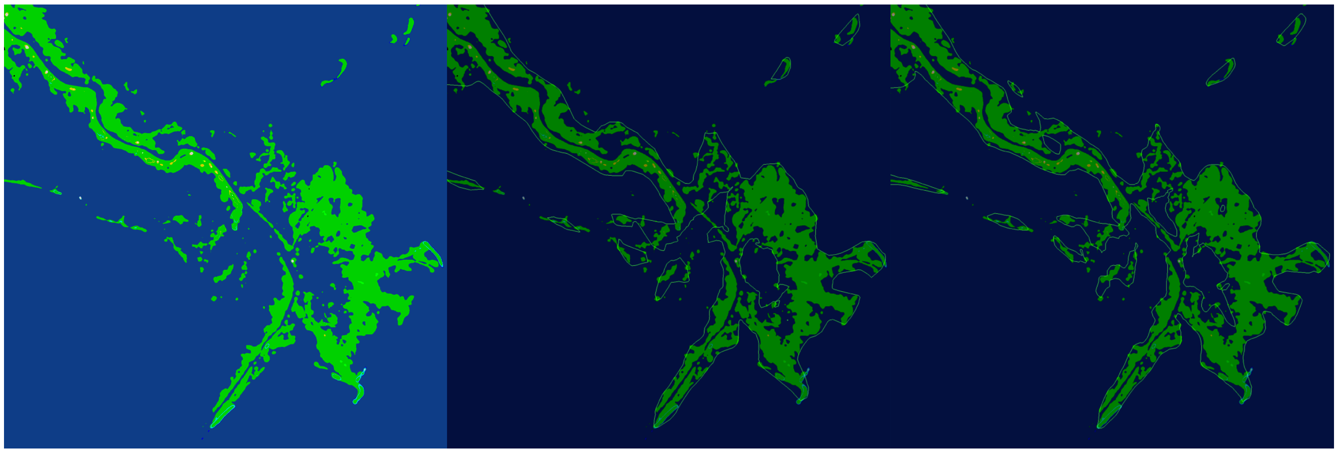
That’s a very large difference in the expressive power of the models these two networks can produce. Let’s have a look at what this means in practice.

Consider the problem of analyzing satellite imagery and using machine learning to determine which parts of a satellite photo correspond to land and which parts correspond to water. This may seem like a simple task, but consider what happens at river deltas where large rivers meet the sea

Mississippi River Delta – image credit: NASA, Public Domain

We want to train a neural net with a ***fixed number of units*** to distinguish between land (green areas above) and water (anything blue or blue-ish). The input training set will be a set of point locations [x,y] with a corresponding label for water or land. The network must then learn to classify ***any location*** in the map as either water or land.

Results below show a ***reference image***, and the ***classification boundaries*** produced by networks with one and two hidden layers using a fixed budget of 180 units (the two-hidden-layer network in this case will have layers with 90 units each).


Reference (expected classification) 1-hidden-layer decision boundary 2-hidden-layer decision boundary

The problem is difficult given how irregular and non-contiguous the land regions are. The network with a single hidden layer does a pretty decent job, with a classification error of 9.34% - the network with two hidden layers reduces the classification error to 8.32% (a 10% improvement), but more importantly, notice how much more complex the decision boundary is for the 2-hidden-layer network. We should expect a network with even more hidden layers, and a sufficiently large number of units, to work quite well even for such a complex classification problem.

Given that deep networks can learn more complex models for a given computational budget (in terms of the number of units in the network), why would we not always use a deep network rather than an equally large but shallow one?

As it turns out, adding layers to the network comes with potential complications in terms of the numerical stability and time required to train the network. Common issues that arise in training large, deep networks include:

Challenges with Gradient Propagation

Recall that the training process – error back-propagation, is a form of gradient descent. Gradient information is obtained at the output of the network (where concrete error information is available), and then propagated layer-by-layer backwards through the network, adjusting weights by some small amount (the learning rate) in the direction indicated by the gradient.

Even for shallow networks, this has to be done carefully – we discussed using randomized subsets of training samples (batches, and in Deep Learning, these are usually called ***minibatches***), and aggregating gradient information for the entire batch before performing weight updates (as opposed to updating the weights after each individual sample is processed). This is the standard training process known as ***stochastic gradient descent (SGD)***.

For very large networks, gradient propagation is tricky, and specialized techniques have to be applied in order to successfully train such models:

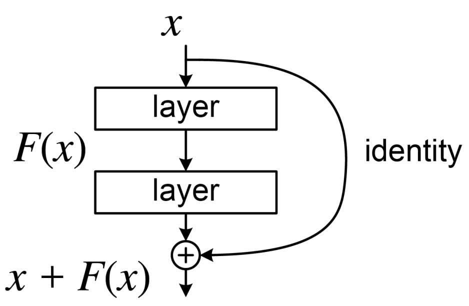
Gradient momentum – the more common optimizers for deep networks incorporate the idea of ***momentum*** – that is, they keep track of the direction of previous weight updates, and use previous updates (with careful weighting) in determining the direction of the current update for each weight. ***ADAM*** (Kingma, Ba, 2015), one of the most common optimizers for deep networks, incorporates momentum information, and uses an adaptive learning rate in order to improve network convergence and reduce problems related to vanishing gradients.

It should also be noted that the use of ***ReLU*** units is at least in part motivated by the fact that they provide a constant gradient across their linear operating region. Other activation functions such as the ***logistic*** or ***tanh*** sigmoids suffer from ***saturation*** at both ends of their operating region, which results in gradients near zero and slow or stalled learning.

Vanishing Gradient – as the gradient information is propagated backward, the gradients can become very small, the smaller the gradients, the slower the weight adjustments. The network learns very slowly or stops learning at all. The more layers a network has, the more this becomes a problem.

Exploding Gradient: The opposite of a vanishing gradient, this manifests itself in the form of weight updates with very large and unstable magnitude. While the vanishing gradient problem occurs because of successive multiplications with small numbers, the exploding gradient occurs due to successive multiplication of large numbers. If this happens, the network will not converge.

Introducing Skip Connections – both vanishing gradient and exploding gradient problems are related to the number of steps of back-propagation that are required to reach a particular network connection whose weight is to be updated. One way in which we can help the network learn better is by introducing ***skip connections*** as shown below:



Example of a skip connection in a ***residual network***
(Image: LunarLullaby – Wikimedia Commons, CC BY-SA 4.0)

The skip connection (most often called a ***residual connection***) can jump across any number of layers. A block in a deep network that show a structure like the one above is called a ***residual block*** and networks built as chains of residual blocks are called ***residual networks*** (He et al. 2015). For very deep networks, the residual connection is required to successfully train the model.

Residual connections are helpful in training deep learning models because they provide ***a shortcut*** for the propagation of gradient information. This greatly reduces the issues caused by gradient magnitudes becoming too small or too large. Networks with dozens or hundreds of layers (typical of modern A.I. models including LLMs) can not be trained without some form of skip connection.

Additional optimizations often used in practice:

Batch normalization is often applied and helps the network learning process. Batch normalization works by processing the ***output*** of a particular network layer, and transforming these output activations so that their variance and mean are normalized (zero mean, unit variance). Batch normalization is implemented as a ***layer*** inserted between two regular hidden-layers – it performs the following computation:

Assume the output of a hidden layer within the network is $\vec{a}_i = [a_{i,1} \;\; a_{i,2} \;\; \dots \;\; a_{i,k}]$, the entries in this vector are the activations produced by each of the units in the hidden layer. Compute the ***mean*** and ***standard deviation*** across ***each entry*** in the activation vector as

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} a_{i,j} \;\; \text{mean for activation } j$$

$$\sigma_j = \sqrt{\frac{1}{N}\sum_{i=1}^{N} a_{i,j} - \mu_j} \;\; \text{standard deviation for activation } j$$

Given these values, ***normalize*** the activations in the minibatch so as to obtain a normalized activation vector $\vec{a'}_i = [a'_{i,1} \;\; a'_{i,2} \;\; \dots \;\; a'_{i,k}]$ where

$$a'_{i,j} = \gamma \cdot \frac{a_{i,j} - \mu_j}{\sigma_j} + \beta$$

The next layer in the network is then provided with the normalized activation vectors.

Let's take a moment to process what this update does. First – it normalizes the activations produced by one layer of the network so that they have zero mean and unit variance. Then it ***scales*** and ***shifts*** these activations using ***two learned parameters*** γ and β. The scaling and shifting operation allows the ***batch normalization layer*** to learn the appropriate scale and shift for each layer from training data (the updates to these parameters are determined by taking the gradient of the network's loss function w.r.t. these parameters and using a gradient-descent-based procedure to update them in the direction that reduces the network's loss).

Batch normalization has been found to significantly improve network convergence.

Regularization can be used to help reduce issues caused by large gradients. Regularization works by adding a term to the ***network's loss function*** that is proportional to the ***magnitude of the network's weights*** and penalizes large weights. This can help the training process and can yield a more stable model.

Gradient clipping – this is used to help reduce the gradient explosion problem. Gradients that are too large get clipped or scaled to ensure the weight updates remain stable.

More general considerations

A general problem in the training of large neural networks, and one that is made significantly more relevant for very deep networks is the requirement for ***very large datasets*** in order to learn deep models. Quite simply, the number of weights that have to be learned for very deep models can become exceedingly large – in order to train such models, huge amounts of training data are required. Without sufficient training data, it is difficult to train very large deep networks.

Therefore, techniques to ***augment*** the available training data are often used – these involve applying transformations to the available training data in order to create ***synthetic training samples*** which depict inputs the trained network might conceivably find in regular operation. The type of transformations and the amount of synthetic data generated will depend on the purpose of the network and the nature of the input. But regardless of that, ***dataset augmentation*** can significantly improve the trained model's ability to generalize to data not found in the original training dataset.

Network Architecture

Possibly the most exciting aspect of deep learning is that it allows for a very large range of variations in network architecture – fully connected layers are very often replaced with specialized architectures that fit particular types of problems especially well.

Studying specialized architectures and understanding how the network topology achieves specific goals is essential to gain a working understanding of how and when to use each of the specialized models, and provides insight into relevant properties of large classes of problems (e.g. image understanding, natural language processing, generative A.I.) that motivate the use of specific architectures.