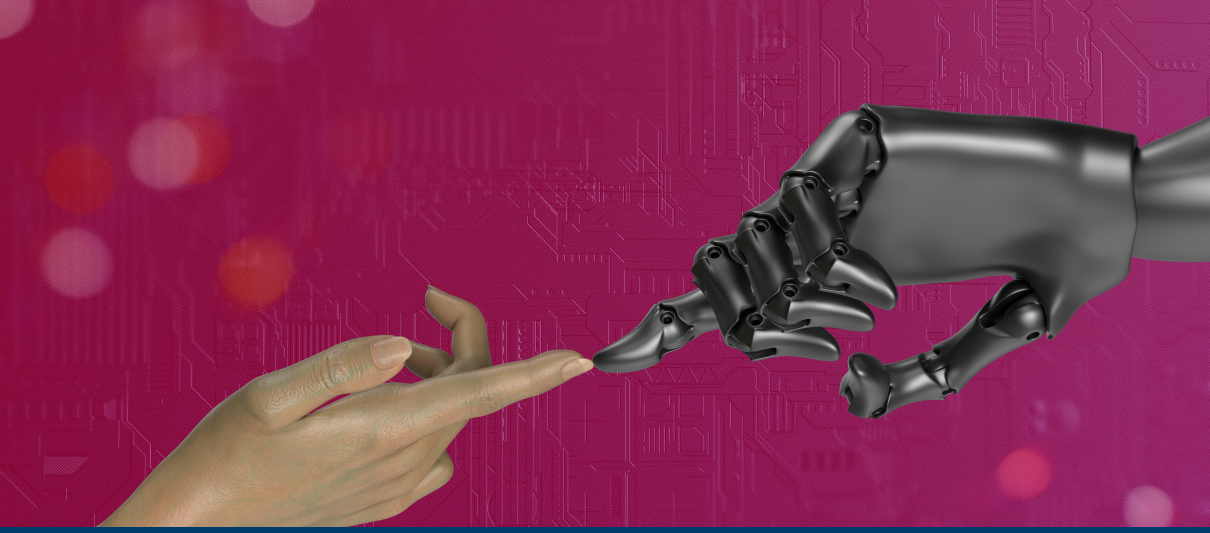


Hands-On Machine Learning

Dr. Derek Bridge | 2026/2027

UCC Coláiste na hOllscoile Corcaigh
University College Cork, Ireland

School of Computer Science and Information Technology
Scoil na Ríomheolaíochta agus na Teicneolaíochta Faisnéise

18. Training Neural Networks

In this lecture, we use MNIST again, but briefly. Then, the main part of the lecture uses a dataset of images of cats and dogs. It comes from Microsoft researchers, for a Kaggle competition: https://www.kaggle.com/c/dogs-vs-cats.. It contains 12,500 medium-resolution JPEGs depicting cats and 12,500 depicting dogs. We use a subset of the full dataset.

Gradient Descent for Neural Networks

- At the output layer, we can straightforwardly compute loss: we can get the network's output (its prediction) and we have the target value, because the training set is a labeled dataset.
- But we cannot straightforwardly compute loss at the hidden layers: we know what outputs their neurons produce but we do not know what they should produce (target values).
- The solution is to assume that each neuron in layer l is responsible for some part of the loss of the neurons it is connected to in layer $l + 1$.

The Backpropagation Algorithm

- The **Backpropagation algorithm** repeatedly makes two passes through the network:
  - a **forward pass** to make predictions: we feed training examples into the network and then work forwards through the network, computing activations layer by layer until we reach the output layer;
  - a **backward pass** to compute the gradients: we calculate error at the output layer and then work backwards computing shares of the error layer by layer until we reach the input layer.
- With these two steps completed, we have all the gradients, so we can update all the weights and biases.

Demos

- http://experiments.mostafa.io/ffbpann/
- https://mlu-explain.github.io/neural-networks/

The Vanishing Gradient Problem

- Each weight is updated by an amount proportional to the gradient of the loss function with respect to that weight.
- But if the gradient is very small, the weight doesn't change much.
- This may prevent the network from converging to a good approximation of the target function.
- This is worse for deeper networks. The shares of the error become ever smaller as they are backpropagated.

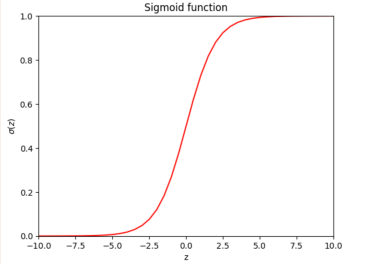
Non-saturating activation functions

Better initialization

Batch normalization

Saturating Activation Functions

- Certain activation functions, including sigmoid, are one cause of the vanishing gradient problem.
- When the input to this function becomes large (positive or negative), the function **saturates** (i.e. becomes very flat).
- When it saturates, its derivative is extremely close to 0 so there's not much gradient to propagate back to earlier layers.
- Even when the gradient is at its greatest (when input z is 0 and $\sigma(z) = 0.5$), it is only 0.25. So in the back propagation, gradients always diminish by a quarter or more.
- This is why we rarely use the sigmoid function as the activation function in the hidden layers of deep networks.

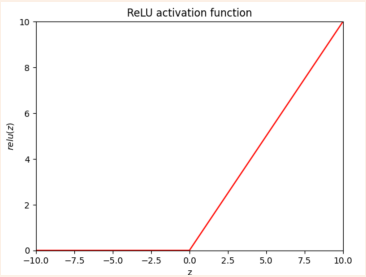


Non-Saturating Activation Functions

- Alternatives to sigmoid have been proposed, including the **rectified linear unit activation function**, ReLU

$$ReLU(z) = \max(0, z)$$

- Neurons that use ReLU have obvious problems too:
  - If their input (weighted sum) is negative, the output is zero and the gradient is zero — and if this is true for all examples in the training set then, in effect, the neuron dies.
  - The gradient changes abruptly at $z = 0$, which can make Gradient Descent bounce around.
- Alternatives to ReLU such as **Leaky ReLU**, **Exponential Linear Unit** (ELU) and **Scaled ELU** have been proposed, having at least some non-zero gradient for negative inputs but they are slower to compute and they introduce further hyperparameters.

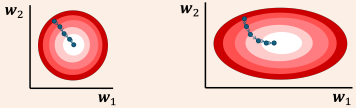


Better Random Initialization

- It turns out that vanishing gradients are more likely for certain ways of initializing the weights and biases.
- Perhaps the most typical method was to use a normal distribution with mean of 0 and standard deviation of, e.g., 0.05.
- Better methods have been proposed, e.g. **Glorot uniform initialization** (also called Xavier uniform initialization).

Batch Normalization

- We know feature scaling is useful for Gradient Descent.



- But, if this is a good idea for the inputs to the first hidden layer, why not use the same idea for the inputs to subsequent layers?
- In essence, in **Batch Normalization**, we normalize the activations (outputs) of layer l prior to them being used as inputs to layer $l + 1$.
- This will control the distribution of the values throughout the training process.

Other Benefits of Batch Normalization

- Batch Normalization reduces the vanishing gradients problem so much, we can even use saturating activation functions.
- Training becomes less sensitive to the method used for randomly initializing weights.
- Much larger learning rates work (faster convergence) with less risk of divergence.
- It acts like a regularizer.

Pretrained Convolutional Neural Networks

- A **pretrained network** is a saved network that was trained, usually on a large dataset.
- Remarkably, these are increasingly being made available for image classification, speech recognition and other tasks.
- Consider the ImageNet dataset (http://www.image-net.org/):
  - 1.4 million images, each manually labeled with one class per image;
  - thousands of classes, mostly animals and everyday objects;
  - annual competitions (ImageNet Large Scale Visual Recognition Challenges, ILSVRC), now hosted on Kaggle.
- There are several pretrained neural networks for ImageNet, half a dozen of which are made available directly in Keras, including
  - VGG16 and VGG19;
  - ResNet50 and ResNet101;
  - Inception V3;
  - Xception.
- These networks incorporate a number of innovations, which we don't have time to study in depth, including local response normalization, skip connections, inception modules, and depthwise separable convolutional layers.

Transfer Learning

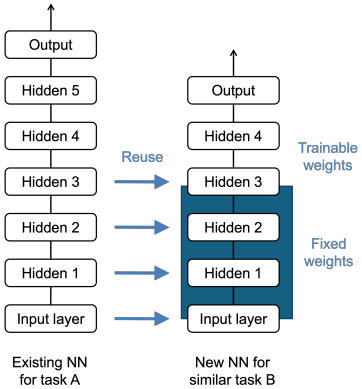
- In **Transfer Learning**, we take a model that was learned when solving one problem and re-use it for solving a different but related problem.
- Specifically,
  - take a pre-trained network;
  - re-use its lower layers, even freezing their weights.
- E.g. re-use the lower layers of Xception in a new network that is trained to classify images of just cats and dogs, or different types of vehicles, or for face recognition, or perhaps even facial expression recognition.
- Of course, this will only work well if the original and new tasks share similar low-level features.

Advantages of Transfer Learning

- It speeds-up training for the new problem.
- It means that less training data may be needed for the new problem (see later).

Re-using the Convolutional Base of a Pretrained CNNt

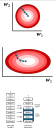
- Convolutional Neural Networks typically comprise two parts:
  - the **convolutional base**: the convolutional and pooling layers;
  - the densely-connected top layers for, e.g. classification.
- We want to reuse the convolutional base:
  - the features learned by these layers are likely to be more generic;
  - the features learned by the top layers will be more specific to our current task.



Transfer Learning Reduces Data Needs

- It's often said that we need lots of data for deep learning.
- But transfer learning helps us in cases where we have more limited data.
- E.g. we want to do face recognition but we have only a few pictures of each person, not enough to train a good classifier:
  - Collect loads of pictures of faces of random people from the web.
  - Train a network to detect whether or not two different pictures portray the same person.
  - This network is presumably a good feature detector for faces.
  - Use the lower layers of this network as the lower layers of your face recognition classifier.
- E.g. we want to do some natural language processing but we don't have a large dataset.
  - Collect loads of sentences from the web, label them 'good'.
  - Corrupt these sentences in various ways, and label them 'bad'.
  - Train a network to classify sentences and non-sentences ('good' vs. 'bad').
  - Use the lower layers of this network as the lower layers of your natural language processing system.

Image credits



I based these diagrams on ones to be found in A. Géron: Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow (2nd edn), O'Reilly, 2019