Tutorial 9 - Neural Networks

Gidon Rosalki

2025-12-17

1 Learning

The traditional goal of learning is to learn a mapping $f(x)$ from a domain set $\mathcal{X}$ to a label set $\mathcal{Y}$. There are two common settings:

- Classification: Here $\mathcal{Y}$ is discrete, where each $y \in \mathcal{Y}$ represents a class
- Regression: $\mathcal{Y}$ is continuous

$\mathcal{X}$	$\mathcal{Y}$	$f(x)$
Images of an animal	{Cat, dog, deer, ...}	Recognise the animal in the image
Sounds of speech	Words in English	Converts speech to text
Images of faces	$\mathbb{R}^2$	Returns location of the mouth in the image
Corrupted images	Natural images	Restores a corrupted image to its original version

Table 1: Examples

To learn *parametric functions*, we have a training set which is a collection of labelled samples:

$$S = \{(x_i \in \mathcal{X}, y_i \in \mathcal{Y})\}_{i=1}^{n}$$

The *hypothesis space* is a set of parametric functions:

$$\mathcal{H} = \left\{ f_\theta(x) \,|\theta \in \mathbb{R}^S \right\}$$

This could be:

- Linear functions where $f(x) = \sum_{i=1}^{S} x_i \cdot w_i$, parametrised by a vector $w \in \mathbb{R}^S$
- Linear transformations $f(x) = Ax$, parametrised by a matrix $A \in \mathbb{R}^{d \times S}$
- Neural networks

Our **goal** is to find parameters θ such that $f_\theta \in \mathcal{H}$ is the "best fit" for the training set S. Not all parameters are learned, there are also hyper-parameters that we set beforehand.

1.1 Best fit

We need to consider what it means to be a "best fit". We need to define how good a fit is, so to do this we define a loss function $L(f, S)$, which measures the "error" of f with respect to S. A lower loss will mean a better fit. This has transformed our **goal** to finding f^* that minimises the loss:

$$f^* = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \{L(f, S)\}$$

For example:

- **0 - 1 Loss**: the percentage of examples on which f is wrong:

$$L_{0-1}(f, S) = \frac{\text{number of times } f(x_i) \neq y_i}{|S|}$$

This is not particularly used, since it is not great.

- **Mean squared error**: Average distance of $f(x_i)$ from y_i

$$L_{MSE}(f, S) = \frac{1}{|S|} \sum_i ||f(x_i) - y_i||^2$$

- **Cross Entropy Loss**: Assume that $f(x)$ outputs the probabilities of x belonging to each possible class. Use the probability of choosing the correct class on all examples.

$$\text{Loss} = -\frac{1}{n} \sum_{j=1}^{n} \log\left(f\left(x_{j,\text{correct}}\right)\right)$$

  Where $f(x_{j,\text{correct}})$ is the probability assigned by the model to the correct class for x_i

So as we can see, we want to minimise the loss function. If $L(f, S)$ is a differentiable function, then we can use **Gradient Descent**. The key principle here is that at each iteration i of GD, we take a step in the direction of the steepest descent:

$$\theta_i = \theta_{i-1} - \eta \triangle f_{\theta_{i-1}}$$

The step size is the learning rate, called η, is a hyper-parameter. You need to pick a value that is not too large (will miss the minimum), and not too small (will take forever to finish).

If $|S|$ is large, then every GD iteration is expensive. Instead, we approximate it using a small, randomly chosen batch $B \subset S$. This is called **stochastic gradient descent**. This has the loop:

1. Sample a random batch $B \subset S$
2. Compute the gradient of $L(f, B)$ with respect to the parameters of f
3. Update parameters of f using the gradient

We perform enough steps until we have covered the entire dataset, called an **epoch**. We perform multiple epochs during training. Original SGD could be difficult to use for non experts, so instead ADAM is a variant of SGD which is easier to use in practice.

1.1.1 Generalisation and overfitting

Let us assume that we have a model that fits the training set. We do not know if it is any good, it could **overfit** the data, and be terrible for unseen data as a result. In order to avoid this, we must evaluate the model on unseen data. To do this, before training we split our available data into training, validation, and test sets. Then, when training, we choose hyper-parameters using the validation set.

1.2 Summary

We want to fit a parametric function $f_\theta(x)$ to a training set S. We measure the fitness with a loss function $L(f_\theta, S)$:

- For classification we generally use cross entropy loss
- For regression we typically use mean squared error

We find the best f_θ by minimising the loss. For minimisation we use SGD, or one of its variants. Models must be evaluated on unseen data in order to avoid overfitting.

2 Neural Networks

In broad terms, a neural network is a directed acyclic graph of differentiable operations. Each node is called a *layer*, defined by its type and parameters. All the parameters together are the *weights* of the network. The *depth* of the network is the longest path from the input to the output. When learning, we fix the structure, or *architecture*, and only learn the weights. A network is called *feed-forward* if every node has at most one input, and one output connected.

Intermediate results are *Tensors*. Tensors are a fancy term fora multi dimensional array $A \in \mathbb{R}^{M_1 \times \cdots \times M_n}$. A is an nD tensor with the shape being the tuple $(M_1, \ldots, M_n)$. Images are 3D tensors of the shape $(\text{channels}, \text{height}, \text{width})$. For RGB, then channels = 3, and for greyscale channels = 1. The convention in PyTorch is NCHW (batch size, channels, height, width), but some other frameworks use NHWC (Keras). The input and output of each layer are expressed as tensors. The shape and dimension of the output depends on the type of the layer.

2.1 Common layers

2.1.1 Linear / Fully connected

The **operation** is $f(x) = Ax + b$. The **input** is some assumed $x \in \mathbb{R}^N$, and the **output** is $y \in \mathbb{R}^M$. The **hyper-parameter** is M, the output dimension. It has the **learned parameters** of the weights matrix $A \in \mathbb{R}^{N \times M}$, and the bias vector $b \in \mathbb{R}^M$

2.1.2 Activation

The **operation** is to apply a point wise non-linear function $\sigma : \mathbb{R} \to \mathbb{R}$, called the *activation function*, on every entry (neuron) of the input tensor:

- ReLU: $\sigma(z) = \max\{z, 0\}$
- Hyperbolic tangent: $\sigma(z) = \tanh(z)$
- Sigmoid: $\sigma(z) = \dfrac{1}{1 + \exp(-z)}$

This takes the **input** of any tensor of some shape, and returns an **output** that is the same shape as the input. It typically takes no **hyper-parameters**, but there is an exception of LeakyReLU: $\sigma(z) = \max\{z, 0.01z\}$. There are also typically no **learned parameters**, with the exception of PReLU: $\sigma(z) = \max\{z, \alpha z\}$ where α is learned

2.1.3 Convolution

The **operation** is to convolve the input with a set of kernels. We will run the convolution on each of the layers independently, then combine them together. Each different kernel will give us a different output channel. It takes the **input** of a 3D tensor of the shame $(\text{in\_channels}, \text{height}, \text{width})$, and **outputs** a 3D tensor of the shape $(\text{out\_channels}, \text{height}, \text{width})$. The **hyper-parameters** are

1. A spatial shape of kernel $k \times k$
2. Number of kernels M (number of output channels)

It has the following **learned parameters**: For $1 \leq j \leq M$ the kernel $w_j \in \mathbb{R}^{C \times K \times K}$, where C is the number of input channels.

2.1.4 Pooling

The **operation** is to sub-sample each input channel. There are two main types:

- Average pooling: Replace every window with its average value
- Max pooling: REplace eevery window iwth its maximum value

The **input** is a 3D tensor of shape $(\text{in\_channels}, \text{height}, \text{width})$, and returns an **output** which is a 3D tensor of the shape $\left(\text{in\_channels}, \dfrac{\text{height}}{\text{window height}}, \dfrac{\text{width}}{\text{window width}}\right)$. It takes the **hyper-parameter** of the size of the pooling window (typically 2×2), and has no **learned parameters**.

3 PyTorch

There are many deep learning frameworks, like keras, tensorflow, PyTorch, and so on. We will be using PyTorch in this course. The API is very similar to numpy. The data, and models are loaded by default to the CPU. NN computations are very computationally expensive, and slow on the CPU, but well aimed for GPUs, so we will prefer running on the GPU, where computations are much faster. In PyTorch, we have to explicitly move tensors and the model to the GPU (and back to CPU). Datasets and Neural networks are written as classes.

4 Classification

To classify, we need datasets from which to learn. For example, there is MNIST, a very large dataset of handwritten digits, each is a 28×28 greyscale image. There are 60,000 training images, and 10,000 test images. There is also CIFAR10, which has 10 categories aeroplane, vehicle, bird, cat, deer, dog, frog, horse, ship, and truck. Each image is a 32×32 colour image, and there are 50,000 training images, and 10,000 test images.
There is also IMAGENET, which has 1,000 classes, for 256×256 colour images, and around 1.2M of them.

Data augmentation is a technique to artificially enlarge the training set. For example, we can take images, and add flips, rotations, crops, colour jitter, noise, and so on. In short, we apply simple transformations, while keeping the label the same. This helps reduce overfitting, and improve generalisation.

4.1 CNNs

This is a convolutional neural network. CNN is a general term for neural networks that apply convolutions. They typically use convolution layers in the early stages, and linear (fully connected) layers in the final stages.
In these networks, the *receptive field* is the region of the input that influences a specific neuron's activation. So if we convolve with a 3×3 window each layer, then in layer 2 the receptive field is 3×3, but in layer 3 it will be 5×5:

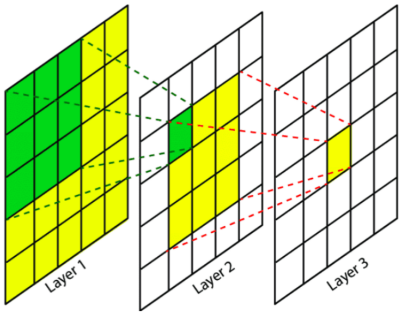

Figure 1: Receptive fields

We also use something called *dropout*. This is another technique to avoid overfitting. Here, we randomly drop units (e.g. neurons) during training, which prevents co adaptations. Units are trained according to a loss that is dependent on all units (i.e. what all the units are doing). Units may change to "fix" the mistakes of other units. This builds complex adaptations that do not generalise to unseen data. Model combination improves performance, and dropout approximates combining many different models.
In general, this is preventing each individual unit from memorising some answer.

4.1.1 ZFNet

We will not talk about this network itself, since it is not especially interesting, but rather interesting things learned from it. It was established that different kernels turned on and off by differing amounts for different textures / colours.

4.1.2 VGG

This was an interesting paper on Very Deep Convolutional Networks for Large Scale Image Recognition. Here they theorised instead of using large kernels between layers, we can use lots of smaller kernels, which will together have the same receptive field as a larger kernel. For example, 3 3×3 kernels have the same receptive field as a 7×7 kernel. This is good since we have more non linearities (deeper), and fewer parameters:

$$3 \cdot \left(3^2 C^2\right) = 27C^2 < 7^2 C^2 = 49C^2$$

Where C is the number of input and output channels. This also means that there is no local response normalisation, adn is a good basis for transfer learning.

4.1.3 GoogLeNet

Then Google (surprise surprise) created GoogLeNet, where they basically went "let's go deeper". It includes deeper networks, with 22 layers, greater computational efficiency. They are not fully connected, and have fewer parameters, with 5M parameters, in comparison to AlexNet which has 60M, and VGG16 which has 138M. It also uses something called the *inception module*.

The *inception model* may be naïvely seen as:

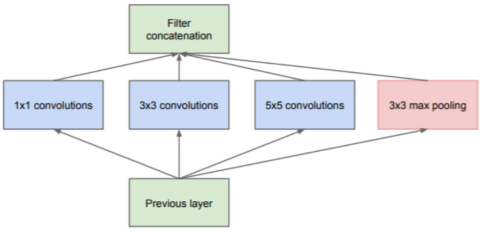

Figure 2: Inception model, naïve version

A more complex model adds a dimension reductoin, but adding more convolutions to each of the 3 rightmost convolutions.

4.1.4 ResNet

Let $F(x)$ be a sequence of regular layers. In a *residual block*, we merge $F(x)$ with the input x, and return

$$H(x) = F(x) + x$$

The addition of x to the output of the computation is called *skip connection*. A residual block can contain any number of layers and any type.

It was noticed that for a 56 layer model, there was a greater loss than for a 20 layer model, on CIFAR-10 ResNet was created to resolve this, it is composed of asequence of residual blocks, in addition to standard layers. It allows training very deep networks with

- Vanishing gradient
- Shallow to deep architecture

For example, with 152 layers, it uses batch normalization after each convolution layer.

4.2 Other tasks

There are other computer vision tasks, such as classifying and detecting objects in an image (like fruit, dogs, people, ties, number plates, and so on). We can also use these models to do semantic segmentation, where we identify different parts of an image (i.e. the person riding a horse, the horse, and other horses in the field).