Lab Course SS2024:
Data Science in Astrophysics
Session 9: Neural Networks-II

Sudeshna Boro Saikia*

May 6, 2024

*sudeshna.boro.saikia@univie.ac.at

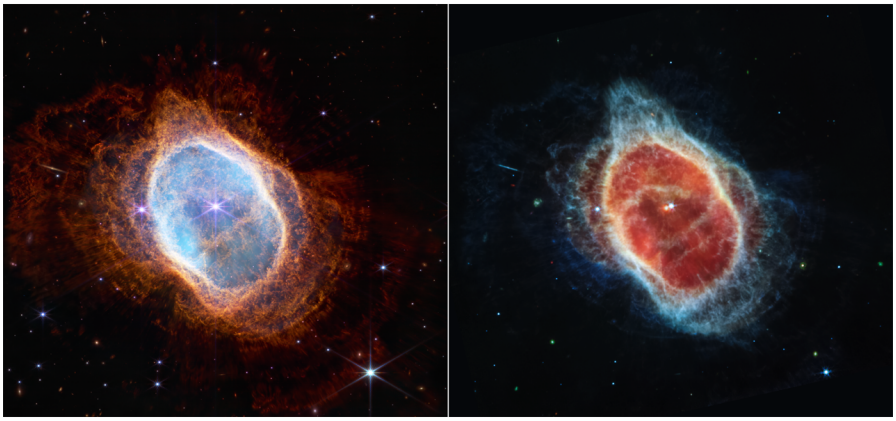

Figure 1: Southern ring nebula as observed by JWST, credit: NASA, ESA, CSA, STScI

1 Introduction

In this lab we will discuss one of the most widely used neural network architecture convolutional neural network (CNN), followed by some practical methodologies.

As the name suggests, CNNs are neural networks that employ the convolution operation. While the simple feed-forward neural network discussed in the previous lecture performs general matrix multiplication in its layers, a CNN applies the convolution operation in at least one of its layers. Convolutional neural networks or CNNs have been very successful in the field of computer vision and are the architecture of choice for most image classification problems. CNNs are commonly used for data sets that have a grid-like geometry and strong spatial dependencies in the neighbouring regions of a grid. One example is an image data, 2D-grid of pixels, where the spatial dependency could be the colour, as often neighbouring pixels have similar colour values. The colour values could be considered as a third dimension (depth) to the input data (width$\times$height$\times$depth), and this shows that the features of a CNN have a dependency amongst one another. Another example of grid based data would be a time series data, which can be considered as a 1D grid of samples at regular time intervals with various dependencies or relationships between nearby samples. In this chapter we will use discuss the inner workings of a CNN by referring to image data as examples.

2 The convolution operation

Convolution is a linear mathematical operation between two functions $f(t)$ and $g(t)$ that results in another function $s(t)$. It is often denoted by an asterisk and takes the following

form,

$$s(t) = \int f(\tau)g(t-\tau)d\tau \tag{1}$$

$$s(t) = (f * g)(t) \tag{2}$$

Equation 1 shows us that the resulting function $s(t)$ is a function of t. Convolution can be defined for any functions for which the integral in equation 1 is valid.

In neural network terminology the two functions (f and g) on the right hand side of equation 2 are known as **input** and **kernel** respectively, and the function s on the left hand side of equation 2 is refereed to as the **feature map**.

In reality the input consists of discrete values and the discrete convolution operation can be defined as,

$$s(t) = (f * g)(t) = \sum_{\tau=-\infty}^{\infty} f(\tau)g(t-\tau) \tag{3}$$

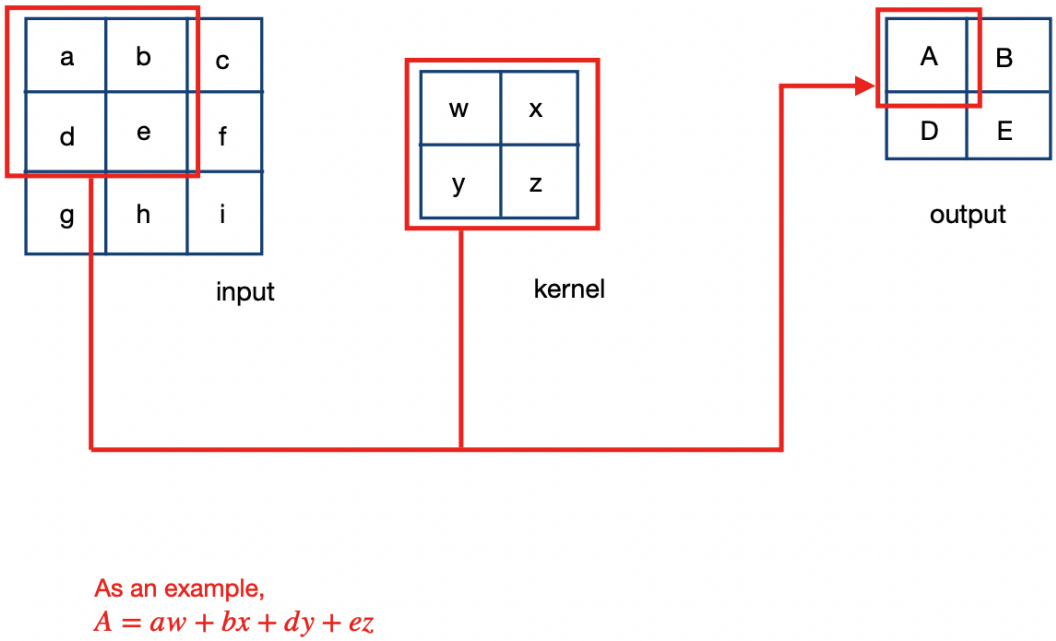

Figure 2: "Convolution" performed by sliding a kernel over an image and computing the summed product at each location. The output is limited to only positions where the kernel lies entirely within the image, 'valid' convolution.

Furthermore, in machine learning applications both the input and kernel are multi-dimensional arrays known as tensors. Each element of the input and the kernel are

stored separately and the functions are assumed to be zero everywhere except where the values are stored. Hence, the infinite summation can be applied over a finite summation of array elements. If we now look at a real world application of a CNN with a 2D-image as input then equation 3 can be rewritten as,

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{4}$$

where $I(i,j)$ is the 2D-image and $K(i,j)$ is the 2D-kernel.

Using convolution's commutative property equation 4 can be re-written as,

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{5}$$

Equation 5 is more straight forward to implement compared to equation 4. We have flipped the kernel relative to the input.

However, most neural network libraries apply another related function **cross-correlation**, which can be expressed as,

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{6}$$

Hence, in the context of CNNs a cross-correlation operation is usually performed instead of a convolution, but the term convolution is used to describe both. If a convolution operation is used then the kernel is flipped relative to the image, i.e., the convolution of the image I and the kernel K is the cross-correlation of I and the rotated version of K, rotated by an angle of 180 degrees.

In CNNs we are often using cross-correlation, hence in this context applying a "convolution" means sliding a kernel over an image and taking the sum of products at each location (which is essentially cross-correlation), as shown in Figure 2.

3 Convolutional neural networks

Convolutional neural network benefits from three key ideas: **sparse interactions**, **parameter sharing**, and **equivariant representations**.

Sparse connectivity: In the traditional feed-forward model, discussed in the previous lab, the network layers use matrix multiplication, where every neuron in one layer interacts with the neurons from the previous layer. However, convolutional neural networks have sparse interactions (also known as sparse weights or sparse connectivity), which is achieved by making the kernel smaller than the input. Figures 3 and 4 show sketches of sparse connectivity with a kernel width of 3.

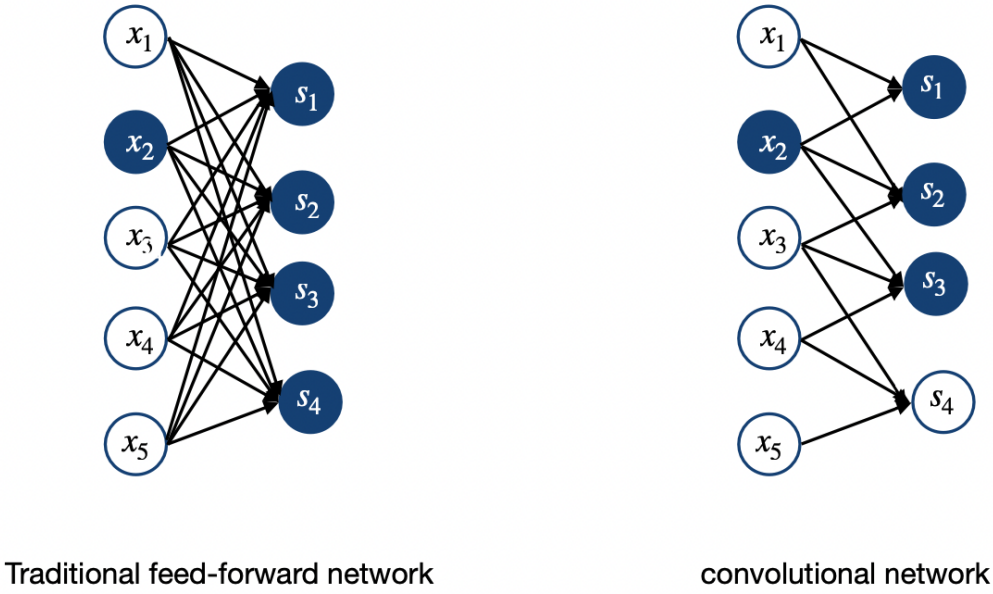

Figure 3: One input node (x_2) and the output nodes affected by this unit is shown for a traditional feed-forward network (left) and a convolutional neural network (right). In the CNN case a kernel of width 3 is used, as a result of which 3 nodes are affected in the output layer (sparse connectivity, viewed from below).

The advantage of sparse connectivity is that computing the output requires fewer operations and leads to a substantial increase in efficiency. This is very handy when an input image contains thousands or millions of pixels, and by applying a kernel (of tens or hundreds of pixels) we store fewer parameters, which reduce the memory requirements and computing the output requires fewer operations. If m and n are the inputs and outputs respectively then matrix multiplication requires $m \times n$ parameters but in the sparsely connected network the parameters get reduced to $k \times n$. Even when sparse connectivity is applied in a deep convolutional network the deeper layers may indirectly interact with a larger portion of the input, as shown in Figure 5.

Parameter sharing: In a traditional network the weights are used only once when computing the output of a layer and no weight is used a second time. In CNNs a kernel is applied to every position of the input (see Figure 2). Thus rather than learning a separate set of parameters for each position in the image matrix we learn only one set of parameters. This reduces the storage requirements of the model parameters (see Figure 9.6 in [1]).

equivariant representation: Equivariant representation means that if there is a change in the input the output also changes in the same way, i.e., function $f(t)$ is equiv-

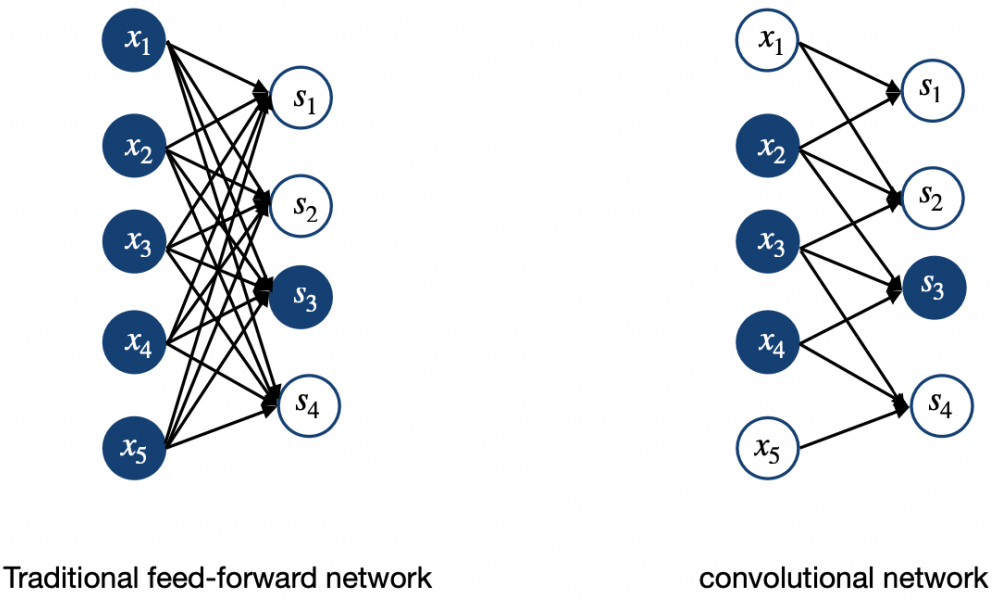

Figure 4: Sparse connectivity from above. One output node s_3 and the input nodes that affect s_3. The input nodes that affect s_3 are known as the receptive field of s_3. The left sketch represents a feed-forward network and the right sketch represents a CNN. Kernel size is the same as Figure 3.

ariant to function g if $f(g(t)) = g(f(t))$. For example, if I is a function that gives the image brightness and g is a function that maps I to another image function $I' = g(I)$, then the result obtained by applying this transformation to I and then applying the convolution will be the same as the result obtained by applying the convolution to I' and then applying the transformation g to the output. In CNNs the parameter sharing causes the layers to follow the equivariance property. Convolution creates a 2-D map of where certain features appear in the input and if these features are moved in the input their representation will move the same amount in the output. Convolution is not naturally equivariant to all transformations and other mechanisms are required to handle these transformations.

3.1 The basic structure of a CNN

A convolutional neural network operates in a very similar fashion as a feed-forward network, the key difference being that the operation in the layers of a CNN are spatially organized with sparse connection between the layers. The commonly used layers in a CNN are: input, **convolution**, **pooling**, fully-connected, and output. The fully connected layer is the same non-linear activation layer that we came across in the previous

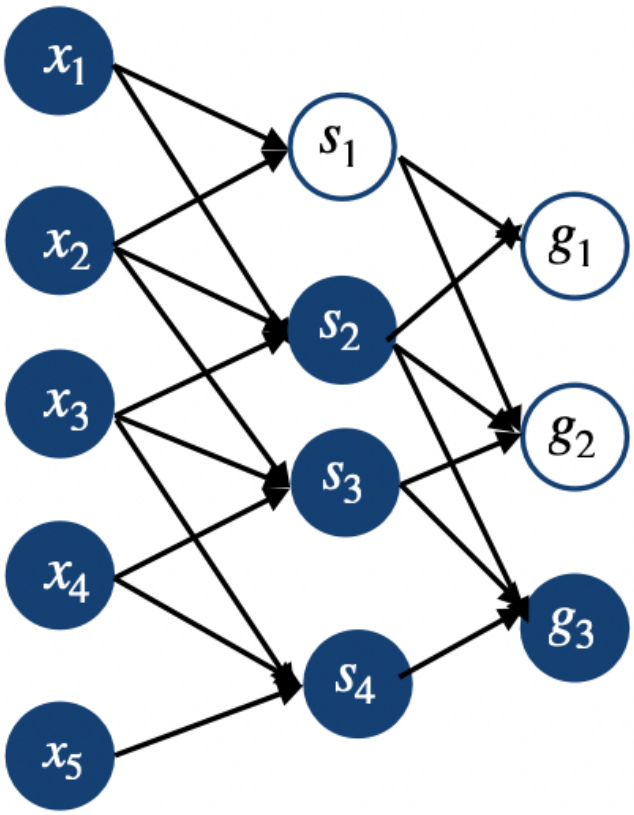

Figure 5: Sketch of the receptive fields of deeper layers in a CNN. The output layer g_3 is directly connected to only a few nodes (highlighted by filled circles), but they are indirectly connected to all or most of the input layer.

lab (sigmoid or ReLU, although ReLU is the most common activation function used). The convolution and the pooling layers are the ones that set a CNN apart from the traditional feed-forward model.

Input layer: For a standard CNN, that deals with 2D-image classification problem [1], the input layer has a 3D grid structure consisting of a **height**, a **width** and a **depth**. The height and the width represents the spatial dimensions of the image and the depth refers to the number of **channels**. This "depth" is different from the depth terminology we associate with a traditional feed forward model [2].

The input data is organized in a 2D-grid and each value of the individual grid points or

[1] data can be of any shape and form but we will assume a 2D-image as the input data when talking about CNNs in this lab.

[2] In a traditional feed-forward model depth refers to the number of layers, but in a CNN the term depth refers to the number of channels in a convolutional layer

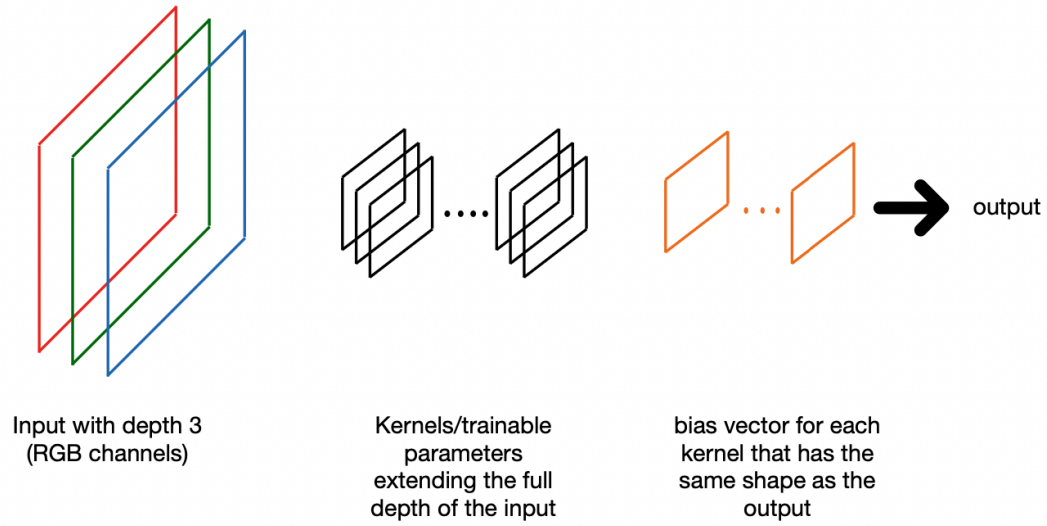

Figure 6: Sketch of a convolutional layer, its input and the associated parameters.

each pixel correspond to a spatial location within the image. Images also have colours and under the RGB scheme three primary colour can be used to define the colours in each gridpoint of the image. The depth corresponds to this three RGB colours or RGB channels. As an example an image of size 32×32 and a depth of size 3 (RGB channels) will have 32×32×3 pixels, where the R, the G, and the B channel each will have a width and a height (2D-grid), as shown in Figure 6.

Convolutional layers: The hidden layers in a CNN perform the convolution operation and are termed as convolutional layers. The parameters of the convolutional layer are the kernels/filters and the biases, where as the parameters associated with a fully-connected hidden layer are the weights and biases.

The kernel is typically square and much smaller than the input. The depth of the kernel is the same as that of the layer that it is applied. The convolution operation places the filter at each (possible) position in the image (Figures 2 and 6) and performs a dot product between the parameters in the filter and the matching input grid. As a result the output is a feature map which is now the input to the next layer (convolutional or traditional fully connected). The number of kernels used determines the depth of the output, or the input to the next layer. Figure 7 shows how this operation is carried out for a 3×3×3 input using 2 kernels (2×2) and 2 biases (2×2) to produce an output of size 2×2 and a depth of 2.

The mathematical operation applied to obtain the output in Figure 7 can be expressed

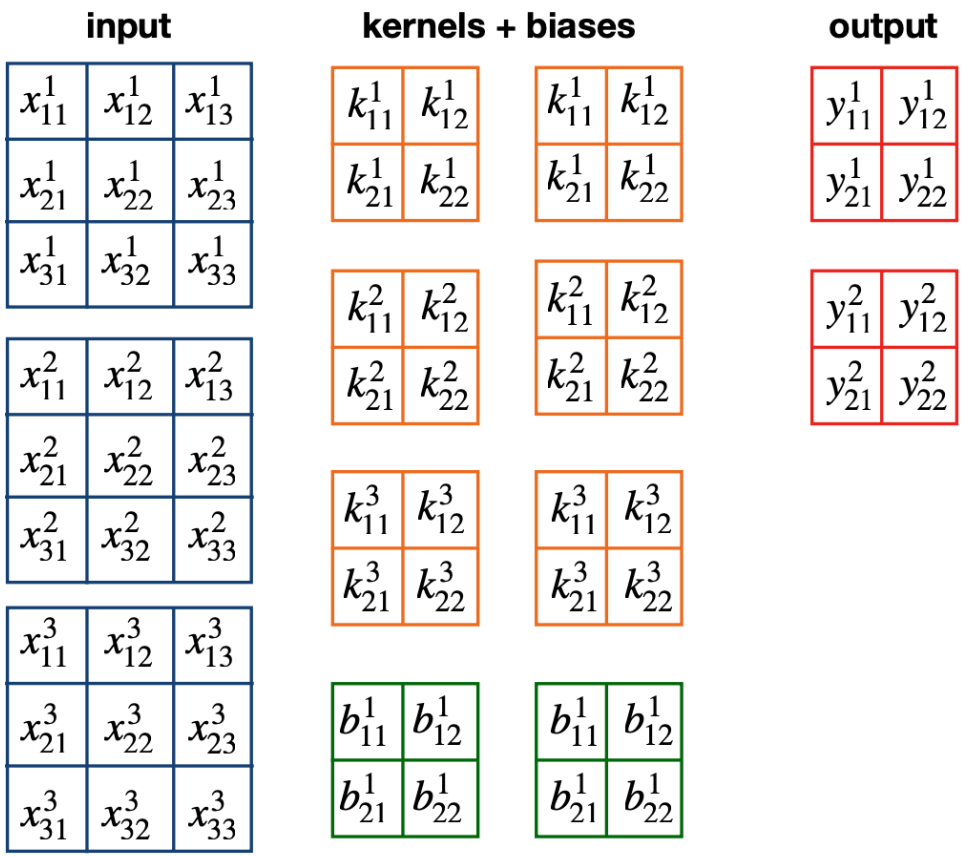

Figure 7: Same as Figure 6 but in matrix form.

as,

$$\mathbf{Y}_i = \mathbf{B}_i + \sum_{j=1}^{d} \mathbf{X}_j * \mathbf{K}_{ij}, i = 1...n \tag{7}$$

where the depth of input $j = 1, 2, .., d$ and the number of kernels $i = 1, 2, .., n$. As you can see the depth of the next layer or the depth of the output is determined by the number of kernels applied. This is how we perform the forward propagation for a convolutional layer. The output of the convolution layer is also run through a non-linear activation function such as ReLU. Equation 7 is the same as the forward calculation in a traditional network but instead of working with scalars each of these variables are matrices [3]. In the back-propagation stage, when performing gradient descent, the derivative of the cost function is applied w.r.t to the parameters, which are the kernels and the biases (matrices).

When a convolution operation is performed in a hidden layer the size of the output is often less than the input, and we might lose some information along the way, specially close to the border of an image or a feature map (see Figure 2). This problem is resolved by using **padding**, where pixels are added around the boundary of the image. The number of pixels added is given by,

$$P = \frac{F-1}{2} \tag{8}$$

where F is the kernel or filter size used in the layer. The value of each padded pixels are set to zero and the padded portion do not contribute to the final output of the layer. Padding allows the kernel to "stick out" of the image or feature map when applying the convolution. Padding is usually applied to all convolutional layers. If padding is not applied then the pixels at the border of the image are under-represented and this gets compounded when multiple convolutional layers are used. As an example, for the image of size 32×32×3 if we apply a kernel $5 \times 5 \times 3$ then equation 8 tells us that $\frac{5-1}{2} = 2$ zeros are padded on all sides of the image. As a result the 32×32 spatial footprint increases to 36×36 and reduces back to 32×32 after the convolution is applied. Figure 8 illustrates this example for a reduced image size of 3×3. This form of padding is known as **half-padding**.

A full-padding can be also applied, where the kernel and the image/feature map might overlap on a single pixel in the corners. In this case the input is padded by $F - 1$ zeros on each side. Each spatial dimension of the input increases to $2(F - 1)$. If the input dimension of the original image is L and B then the padded dimensions in the input volume become $L + 2(F - 1)$ and $B + 2(F - 1)$, and after performing the convolution the dimensions of the output or the feature map becomes $L + F - 1$ and $B + F - 1$. Thus instead of reducing the dimension full-padding increases it, and this increase in dimension is the same value $(F - 1)$ that no-padding decreases it.

[3] The operation used in this convolution layer is actually cross correlation as discussed earlier

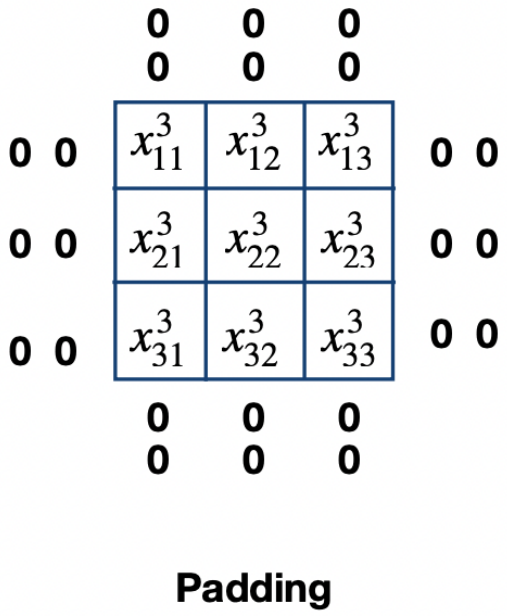

Figure 8: Half-padding for an input image.

Until now we discussed scenarios where convolution is applied at every position in the spatial location of the image or feature map (i.e., applied to every single pixel). It is however not necessary to apply convolution at every spatial position. To avoid applying convolution to every single location we can apply **strides**. If a stride S is used in a given layer then the convolution is performed at the locations $1, S+1, 2S+1, ...$ If $S = 1$ then the convolution is performed at every single pixel location. The spatial size of the output on performing a convolution with strides is $(L-F/S+1)$ and $(B-F)/S+1$.The use of stride thus reduces the spatial dimension of the layer, by a factor of approximately S. A larger stride might help with over-fitting but too large a value might reduce the accuracy.

Pooling layer: The layer that follows a convolutional layer is a pooling layer. Once the convolution layers provide the feature map, a pooling function is applied to further modify the output. A pooling function replaces the output of the network at a certain location with a summary statistic of the nearby outputs. The most widely used pooling function is the **max pooling** operation that reports the maximum output within a rectangular neighbourhood. Pooling makes the output invariant to small translations of the input. This means that the network is more interested in finding features than focusing on where that feature is located in an image.

The pooling operation works on small grid regions of size $P \times P$ in each layer and produces another layer with the same depth (unlike the filter). The pooling window is slid across the grid and the maximum of the neighbouring values are returned (in max-

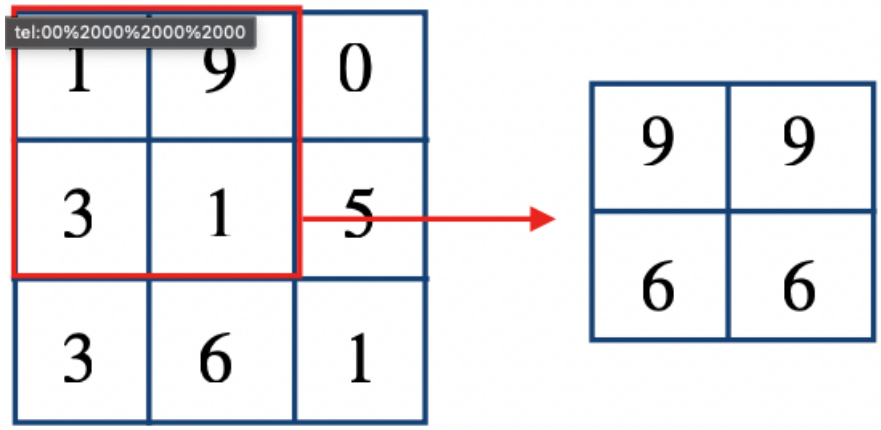

Figure 9: Pooling with a 2×2 kernel and a stride of 1.

pooling). Unlike the convolution operation, where the kernel is applied to the image grid or feature map to return a feature value, pooling operates on each feature map to return a new feature map, as a result of which the depth of the pooling output is the same as the depth of the input. Figure 9 demonstrates how pooling works for a stride of 1 and a kernel size of 2. The output of the pooling layer has a different dimension and in this case the dimension is 2×2.

In Figure 9 the higher values of the pixels are preserved and passed on to the next layer. Thus the sharpest features are extracted and a low-level representation of the image is obtained. It adds translational invariance to the network.

Pooling is also applied to increase the size of the receptive field while reducing the spatial footprint of the layer (when using strides greater than 1). Increased sizes of the receptive fields are needed to capture larger regions of the image within a complex feature in the next layers. Additionally, it is expected that max pooling could help with overfitting.

Fully connected layers: The output/feature map of the pooling layers are usually passed on to a fully-connected non-linear layer (same as the layers in a traditional feed-forward network). The number of parameters also increases in these last hidden layers as dense connections are used. Usually more than one fully-connected layer is included in the network architecture to increase the power of computations towards the end. However, there is a trade-off between the number of fully connected layers (increased number of parameters) and memory usage.

Output layer: The output layer of a CNN is the same as the output layer of a traditional feed-forward network, in the sense that the design of the output layer is dependent on the problem at hand. In the case of image classification the output layer is fully connected to the layer before.

Output size in a convolutional or pooling layer The output of a convolutional layer or the pooling layer doesn't have the same size as the output of the input image or feature space. The size can be calculated using the equation below,

$$output = \frac{input - kernel + 2 * padding}{stride} + 1 \tag{9}$$

As an example, for an input of $32 \times 32 \times 3$ and a kernel of 5×5, no padding and a stride of 1, the output size is 28. If the number of kernels applied is 2 then the output of a convolutional layer will have the dimension of $28 \times 28 \times 2$. If we now apply a pooling layer with a kernel 2×2 and a stride of 2 then the output of the pooling layer is 14.

4 Data augmentation

One way of improving the performance of a network is **data augmentation**. Data augmentation refers to the generation of new training data by applying some transformation to the training dataset. In image classification data augmentation is an effective way to increase the training size. As an example, by mirroring or reflecting an image we can create new sets of data and the model is exposed to different orientations of the image during the training phase.

This is particularly useful in problems where we might have a class imbalance, such as the Kepler exoplanet detection problem in lab 8. However, data augmentation is not well-suited for all applications and care should be taken when applying to astrophysical data, as not all transformations are physically meaningful.

To do: What kind of data augmentation could we apply to the Kepler exoplanet dataset?

5 Practical methodologies

Until now we discussed the inner workings of a fully-connected (lab 8) and a convolutional neural network. If we now go ahead and apply these models to a scientific problem, often times the model's performance is not very good at the first try. As an example, in the Kepler exoplanet problem in the last lecture our model was not very good at classifying exoplanets. Based on our experience we can already identify few different avenues where improvements could be made, such as, data, model architecture, hyper parameter tuning etc.

If we take the example of the training dataset then the first thought that comes to our mind is increasing the number of exoplanets in the sample, which can be done by including real data or by data augmentation. Both of these options come with their own

problems. Adding real data might not be possible due to the limitation of the Kepler dataset. We could add data taken by other instruments but they might come with additional uncertainties and severe data processing might be needed. We could apply data augmentation but the augmentation methods might not be realistic. We could be adding features that are not produced by stars and their exoplanets. Or should we add more features instead? Additional features could be information on the stellar type, stellar activity.

Additionally, in terms of the model architecture we can try a shallow fully-connected network, deep fully-connected, or a CNN. Which one will work better? A shallow fully-connected network will have less layers than a deep network, hence less parameters to tune. A CNN might be good at identifying different features and perform a better classification. The choice of the model can be obvious for certain kinds of datasets, but not so much for others. As an example, a CNN is perfectly suited to perform image classification. However, is a CNN better than a fully-connected network in classifying the Kepler light curves?

How do we make a decision on which direction to go so that we increase the model's performance? but before that, how do we evaluate the performance of a model?

5.1 Model evaluation

The first step of evaluating the performance of a model is **cross-validation**. In neural networks this is achieved by splitting the dataset into train-validation-test splits. During training phase the model learns from the training data, and is evaluated against another data set known as the validation data set. The validation data set is usually generated from the training data set. The model doesn't learn from the validation data set but has access to the data during the training phase. Based on the model performance on the validation data set the model hyper-parameters are adjusted. Once the model is fully trained the performance of the model is evaluated against the test data.

We can use some form of an performance metric to help us evaluate model performance. As an example, in the Kepler exoplanet dataset we used accuracy as the performance metric or error rate of the model. Our model had a high accuracy but it was not very successful at classifying exoplanets. This is because our dataset is very biased and dominated by examples of non-exoplanet host stars. The model is correctly identifying the non-exoplanet hosts with a high level of accuracy but failing at identifying the limited number of exoplanet host stars. As a result of which this simple metric is not very useful.

Two common metrics that are usually used instead are, **precision** and **recall**. We can define these terms using the confusion matrix in Figure 10. For a binary classification, our Kepler example, the confusion matrix is a 2×2 matrix with ground truths and predicted classifications on each axis. This matrix describes the performance of a model using the terms true positive (TP), true negative (TN), false positive (FP), and false negative

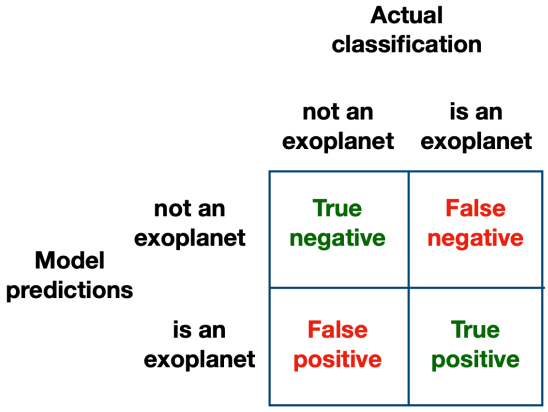

Figure 10: Confusion matrix for the Kepler exoplanet classification problem.

(FN). The matrix is pretty self explanatory but as an example, a true positive occurs when the model classifies a light curve as an exoplanet and the label also classifies as an exoplanet. Since the model is aimed at classifying exoplanets we consider the exoplanets to be positive and non-exoplanets to be negative.

Using this confusion metric we can define the precision, the recall, and the accuracy. We already came across accuracy and can define it as the ratio of the correct predictions and all predictions, which can be written as,

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{10}$$

The precision is defined as the ratio of the correct positive predictions to the total positive predictions,

$$precision = \frac{TP}{TP + FP} \tag{11}$$

Recall is defined as the ratio of the correct positive predictions to the true positive examples

$$recall = \frac{TP}{TP + FN} \tag{12}$$

Precision and recall provide a better indication of the model performance when the dataset has skewed classes. Our exoplanet example has skewed classes as there are very few examples with exoplanets (positive examples).

If we apply these two new metrics to the exoplanet example then what does precision and recall mean? Precision tells us: out of all the lightcurves where the model predicted the presence of exoplanets, how many are true exoplanets? If the model has a high precision it means that most of the lightcurves that the model is classifying as exoplanets are actual exoplanets. Recall tell us: out of all the lightcurves that actually have exoplanets

how many did the model correctly predict? If a model has high recall it means that the model is good at identifying lightcurves that have exoplanets (refer to [1] for an in depth discussion on the performance metrics).

Once we know how the model is performing we can take the next steps to improve the performance. This could be done by improving the algorithms [4]or by including more data or by tuning the different hyperparameters.

References

[1] Ian Goodfellow and Yoshua Bengio and Aaron Courville (2016) *Deep learning*, MIT Press.

[2] Charu C Aggarwal (2018) *Neural Networks and Deep learning*, Springer.

[4]If we want improve to improve a learning algorithm then knowing the bias and the variance of the model is very useful. Bias and variance are prediction errors of a model and gives an indication of whether the model is suffering from an underfitting or an overfitting problem. In simple term a model that has a high bias is underfitting the data and a model with a high variance is overfitting the data.