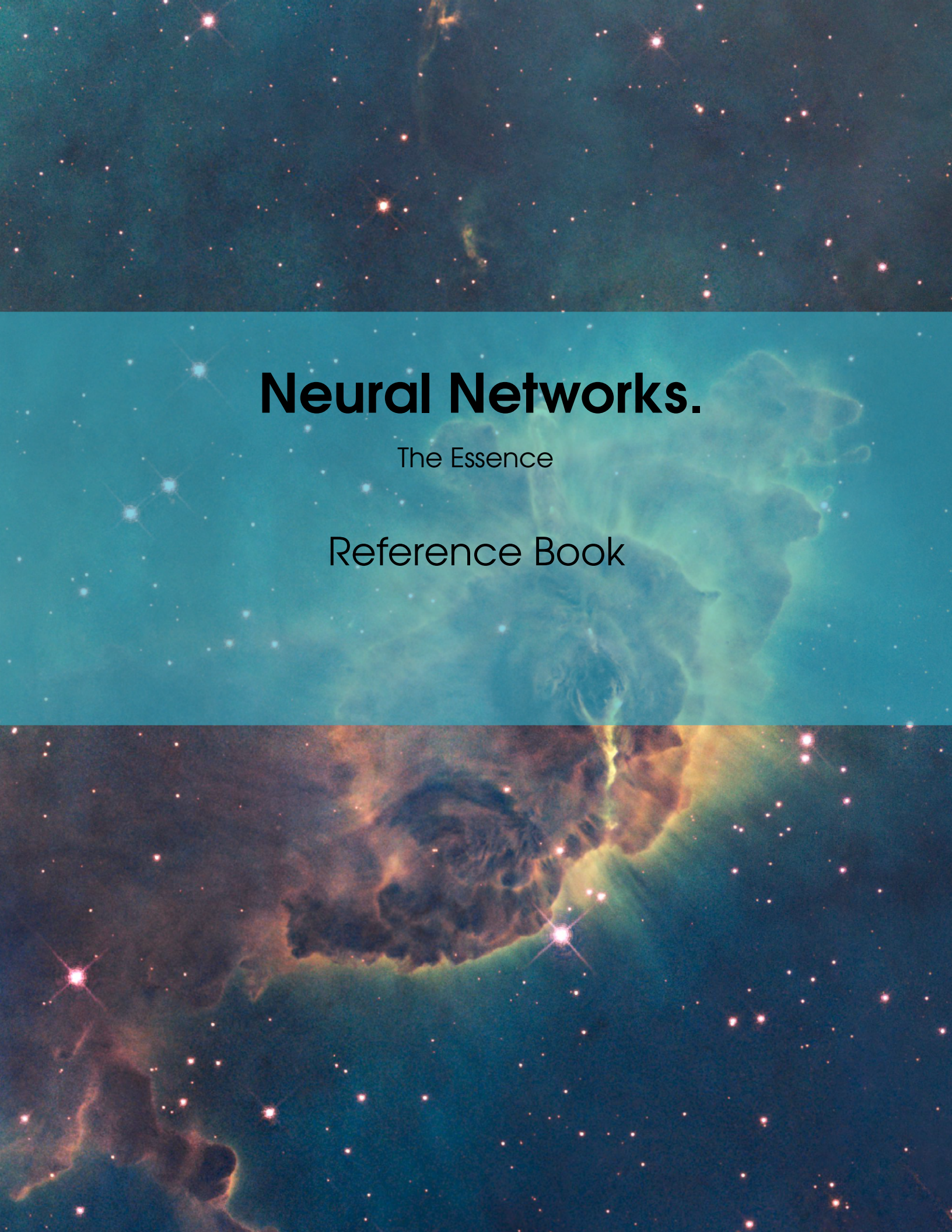


Neural Networks.

The Essence

Reference Book

HANDS ON MACHINE LEARNING AND NEURAL NETWORKS

GITHUB.COM/YASHSOLANKI2007/

Notes from above said book and more

First release, November 6

Contents

1 Common Problems with Neural Networks 5

1.1 Vanishing Gradient Problem 5

1.2 Exploding Gradient Problem 5

1.3 Glorot and He Initialization (solution to the above 2 problems) 5

1.4 Gradient Clipping (Solution to Exploding Gradients) 5

1.5 Dying ReLU Problem 6

1.6 Short Term Memory Problem 6

2 Optimization Tecniques 7

2.1 Gradient Descent 7

2.2 Momentum 8

2.3 Nesterov's Accelerated Gradient 8

2.4 AdaGrad 9

2.5 RMSProp 9

2.6 Adam 10

2.7 Nadam 10

3 General Layers 11

3.1 Dense 11

3.2 Dropout 11

3.3 Batch Normalization 11

4 Layers for CNNs 13

4.1 Convolutional Layer 13

4.2 Pooling Layers 13

4.2.1 MaxPooling 14

4.2.2 AveragePooling 14

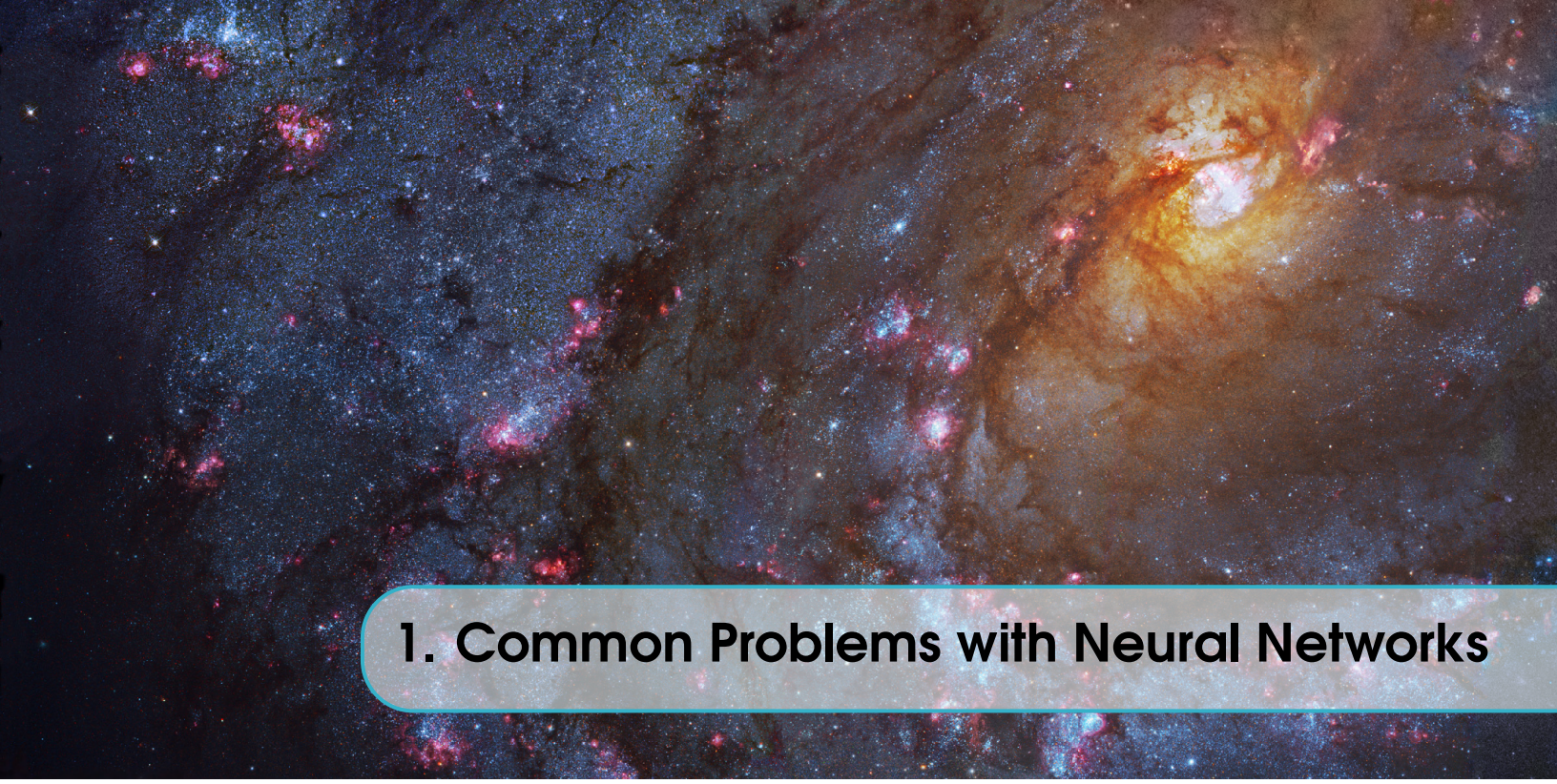
5 Layers for RNNs 15

5.1 Simple RNNs 15

5.2 LSTM 16

6 Attention Mechanisms 17

6.1 Bahdanau Attention 17



1. Common Problems with Neural Networks

1.1 Vanishing Gradient Problem

When we backpropogate through a neural network. We differentiate the cost function (loss function) with respect to all the model params to reach a minima i.e lower the loss. However, often while training DNNs (Deep Neural Networks) the gradients tend to get smaller and smaller as the algorithm tends to go to the lower layers. Since these gradients are crucial for the optimization step the lower layer connection weights remain basically unchanged and we are unable to converge and reach the minima. This is the vanishing gradient problem. The vanishing gradient problem can be seen in the logistic sigmoid activation function since for higher inputs the function saturates to 0 or 1. Thus when the gradients to these functions are computed they are virtually nothing and the existing small gradient keeps getting diluted as we move backwards. Pg 332 of primary source.

1.2 Exploding Gradient Problem

Opposite of the vanishing gradient problem. In this case the gradients grow bigger and bigger and diverge. This mainly occurs in RNNs (Recurrent Neural Networks)

1.3 Glorot and He Initialization (solution to the above 2 problems)

In summary, in an ideal condition the variance of outputs of each layer should be equal to the variance of the inputs. The same should also apply to the gradients before and after flowing through a layer in the reverse direction

While specifying an activation function tensorflow keras uses glorot initialization by default, however, it can be changed by using the kernal_initializer parameter.

1.4 Gradient Clipping (Solution to Exploding Gradients)

This clips the gradients so that they do not get greater than a certain threshold. This can be done to eliminate the exploding gradient problem particularly in RNNs where it is difficult to use batch

normalization. This can be done by 2 argument clip_value wherein we have to explicitly state the value at which the gradient should be clipped. This poses some problems wherein the orientation of the gradient vector may change. To counter this we make use of the clipnorm which clips them in such a way that the gradient vector direction does not change. More info on Pg 345 of primary source.

1.5 Dying ReLU Problem

When the ReLU activation function is used some neurons effectively die meaning that they output only 0s. A neuron is dead when the weighted sum of its inputs outputs a negative for all instances in a training set. When this happens ReLU keeps outputting 0s. And because of this backpropogation does not affect it. A solution to this problem was to introduce activation functions that were based on ReLUs principles but introduced leaks such as leaky relu and elu.

1.6 Short Term Memory Problem

This happens in RNNs where the deeper RNNs have virtually no trace of the first inputs. For more information check Pg 514 of primary resource.

2. Optimization Tecniques

2.1 Gradient Descent

Gradient Descent is one of the most popular Neural Network Optmization Tecniques. It involves computing the negative gradient of the model weights and then modifying the weights in that respect so as to converge to a local or better a global minimum.
The equation to represent gradient descent is given below. However, a more mathematical definition would be that gradient descent measures the local gradient of the error function with regard to the parameter vector θ, and then goes in the direction of the descending gradient.

$$\theta^{(nextstep)} = \theta - \eta \nabla_\theta J(\theta) \tag{2.1}$$

In the above equation η is the learning rate. $J(\theta)$ is your cost(loss) function.
A typical gradient of the Mean Squared Error cost function will look something like this.

$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \frac{\partial}{\partial \theta_2} MSE(\theta) \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{pmatrix} = \frac{2}{m} X^T (X\theta - y) \tag{2.2}$$

This is called batch gradient descent wherein the batch size is the complete dataset. Thus naturally this gets very slow as the dataset gets larger.

2.2 Momentum

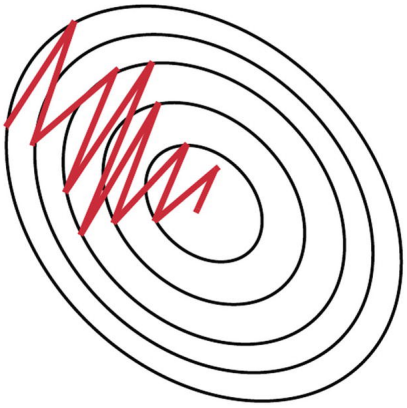
The momentum optimization technique is used to speed up gradient descent in neural networks. Normally gradient descent works by calculating the gradient of the cost function and then subtracting it from the parameter vector. However, in momentum we add an extra term beta to the parameter vector(model weights) which acts as the momentum. Momentum also cares a lot about the previous gradients. Momentum values range from [0, 1] -> typically the value is 0.9.

The equation for momentum is given below.

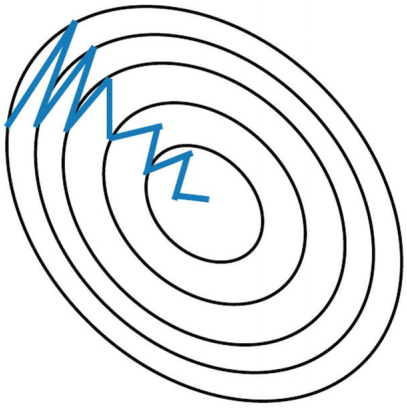
$$m \leftarrow \beta m - \eta \nabla_\theta J(\theta) \tag{2.3}$$

Then we update θ with the following equation

$$\theta \leftarrow \theta + m \tag{2.4}$$



Stochastic Gradient Descent **without** Momentum



Stochastic Gradient Descent **with** Momentum

As it is seen momentum can be used to dampen the movement i.e prevent sensitive movement.

2.3 Nesterov's Accelerated Gradient

This is a variant of momentum in this case we add a small tweak to the momentum vector to obtain a slightly lower loss. What we really do is take a small jump in the direction of the previous gradient as we know that it is in that direction that a minimum lies. This when scaled up can greatly save processing time and accuracy.

$$m \leftarrow \beta m - \eta \nabla_\theta J(\theta + \beta m) \tag{2.5a}$$

$$\theta \leftarrow \theta + m \tag{2.5b}$$

We can see this is action in the above equation(2.5a) where we have the extra βm term in our $J(\theta)$.

2.4 AdaGrad

AdaGrad is an optimization technique that is used to better converge towards the global minimum rather than using something like gradient descent that simply converges to the local minimum. AdaGard achieves this by scaling down the gradient vector down the steepest dimensions.

$$s \leftarrow s + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta) \tag{2.6}$$

In the above equation $\otimes$ represents element-wise multiplication.
This step is responsible to accumulate the square of the gradients into the vector s. Thus if the cost function gets steeper along the kth dimension then s will also get steeper along the kth dimension.

Now we update the parameter θ using the following equation.

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{s+\varepsilon} \tag{2.7}$$

The above update equation is a bit different than the previous update equation that we were using.
The main difference is the presence of $\oslash\sqrt{s+\varepsilon}$. This means that the gradient vector is being scaled down by a factor of $\sqrt{s+\varepsilon}$ where ε is a soothing term to avoid division by zero error.

Thus in summary AdaGrad or Adaptive Gradient Descent is used to decay the learning rate adaptively i.e more during larger slopes and lesser during smaller slopes.

2.5 RMSProp

AdaGrad has a risk of slowing down a too fast and not converging to a global minimum. RMSProp tries to get rid of this issue by accumulating gradients from the most recent iterations rather than all of the gradients from the start of training. This is done by introducing exponential decay in the first step as seen below by the hyperparameter β generally set to 0.9.

The equation below is the RMSProp algorithm.

$$s \leftarrow \beta s + (1-\beta) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta) \tag{2.8a}$$

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{s+\varepsilon} \tag{2.8b}$$

2.6 Adam

The adam optimizer is one of the most commonly used optimizers in deep learning. It stands for adaptive moment estimation. It combines the idea of momentum and RMSProp.
The equation below is the adam algorithm.

$$m \leftarrow \beta_1 s + (1-\beta_1)\nabla_\theta J(\theta)) \tag{2.9a}$$
$$s \leftarrow \beta_2 s + (1-\beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta) \tag{2.9b}$$
$$\hat{m} \leftarrow \frac{m}{1-\beta_1^T} \tag{2.9c}$$
$$\hat{s} \leftarrow \frac{s}{1-\beta_2^T} \tag{2.9d}$$
$$\theta \leftarrow \theta + \eta \hat{m} \oslash \sqrt{\hat{s}+\varepsilon} \tag{2.9e}$$

In the above equation m and s are initially initialized to 0. The momentum decay β_1 and scale decay β_2 are initialized to 0.9 and 0.999 respectively in a normal case.

2.7 Nadam

Nadam is very similar to the Adam optimizer and is used in certain cases where adam fails. Nadam really is just Adam + Nesterov's Accelerated Gradient. It generally thus outperforms adam, however, it does depend upon the dataset.

3. General Layers

3.1 Dense

The dense layer is one of the fundamental layers of deep learning. In a dense layer the all the outputs of the previous layer are connected to the inputs of the next layer. Thus they are also called fully connected layers.
The output of a fully connected layer can be computed using the equation below.

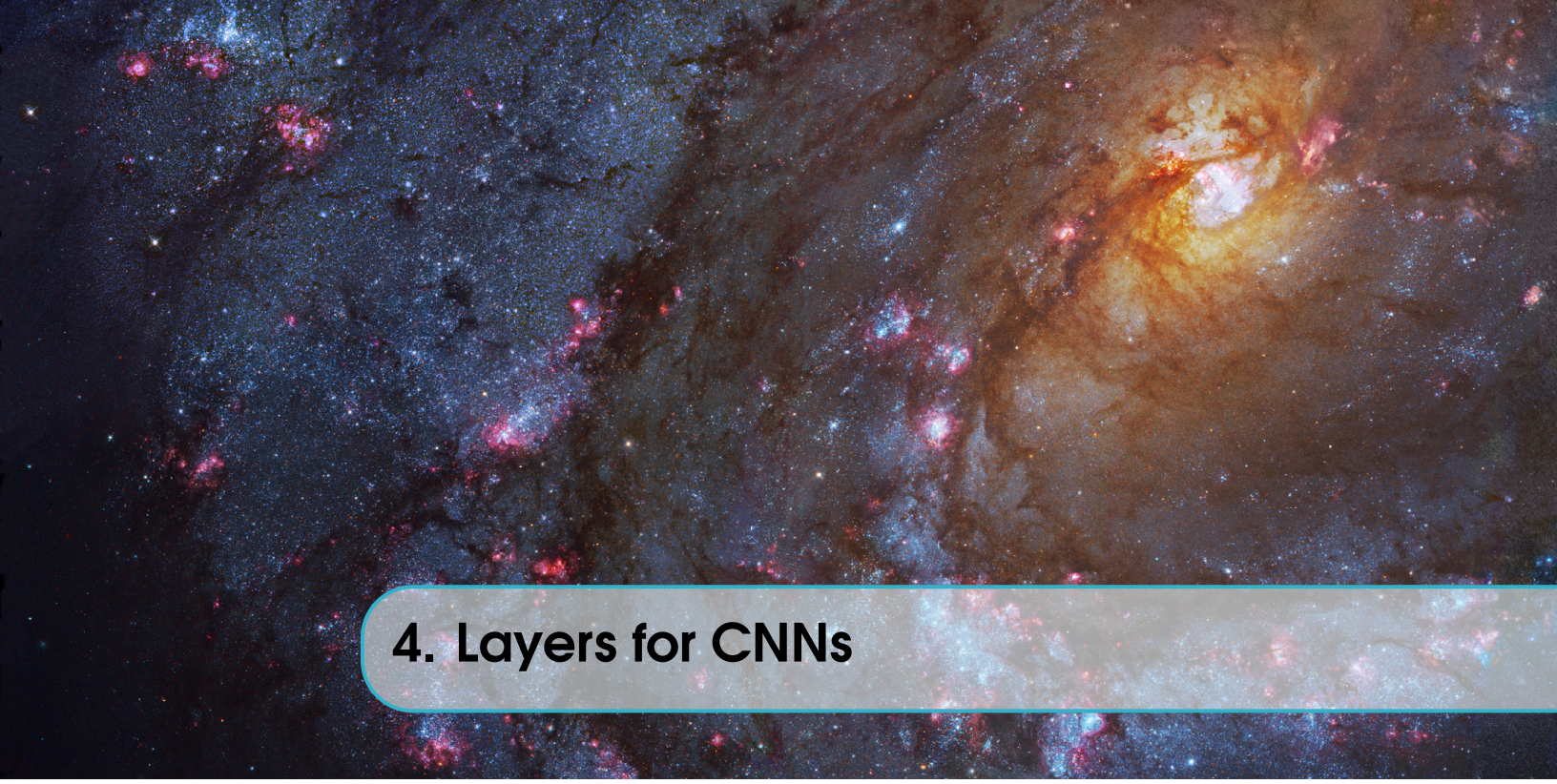
$$z = \sum_{i=0}^{j} x_i \cdot w_i + b \tag{3.1}$$

3.2 Dropout

It is one of the most popular regularization techniques. In this each neuron including the input neurons but excluding the output neurons has a probability p of being ignored during the training step. After training neurons do not get dropped out any more. To learn more check Pg 365 of primary resource.

3.3 Batch Normalization

Batch normalization is a technique for training very deep neural networks that normalizes the contributions to a layer for every mini-batch. This has the impact of settling the learning process and drastically decreasing the number of training epochs required to train deep neural networks. In more technical terms Batch Normalization aims to bring the mean μ of the data to 0 and standard deviation σ to 1. Thus Batch normalization basically allows the network to automatically learn the scale and mean of each of the layer inputs. It standardizes the inputs, then rescales and offsets them. For more concise information check Pg 339 of primary resource and Towards Data Science article about batch normalization.



4. Layers for CNNs

4.1 Convolutional Layer

The neurons in the first convolutional layer arent connected to every single pixel of the input image but only to pixels in their receptive field this continues as we move on to the second Convolutional layer. A neurons weights can be represented as a small image the size of the receptive fields. Check Pg 450 of primary resource for more visual detail. This is called a filter. Thus when the filter is applied to the image the features represented in the filter are shown appearing in the image while the other details get blurred out. Thus a full layer of neurons using the same filter outputs a feature map.

We can compute the output of a neuron of a Convolutional layer using the equation given below.

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f'_n-1} x_{i',j',k'} \cdot w_{u,v,k',k} \tag{4.1}$$

In this case

$$i' = i \times s_h + u \tag{4.2a}$$
$$j' = j \times s_w + v \tag{4.2b}$$
$$\tag{4.2c}$$

The meaning of the symbols are on Pg 453 of primary resource.

4.2 Pooling Layers

Pooling layers are used in combination with Convolutional layers. Their goal is to subsample the image in order to reduce computational load, memory usage, etc.
In pooling layers also each neuron is connected to a limited number of neurons of the previous layer.

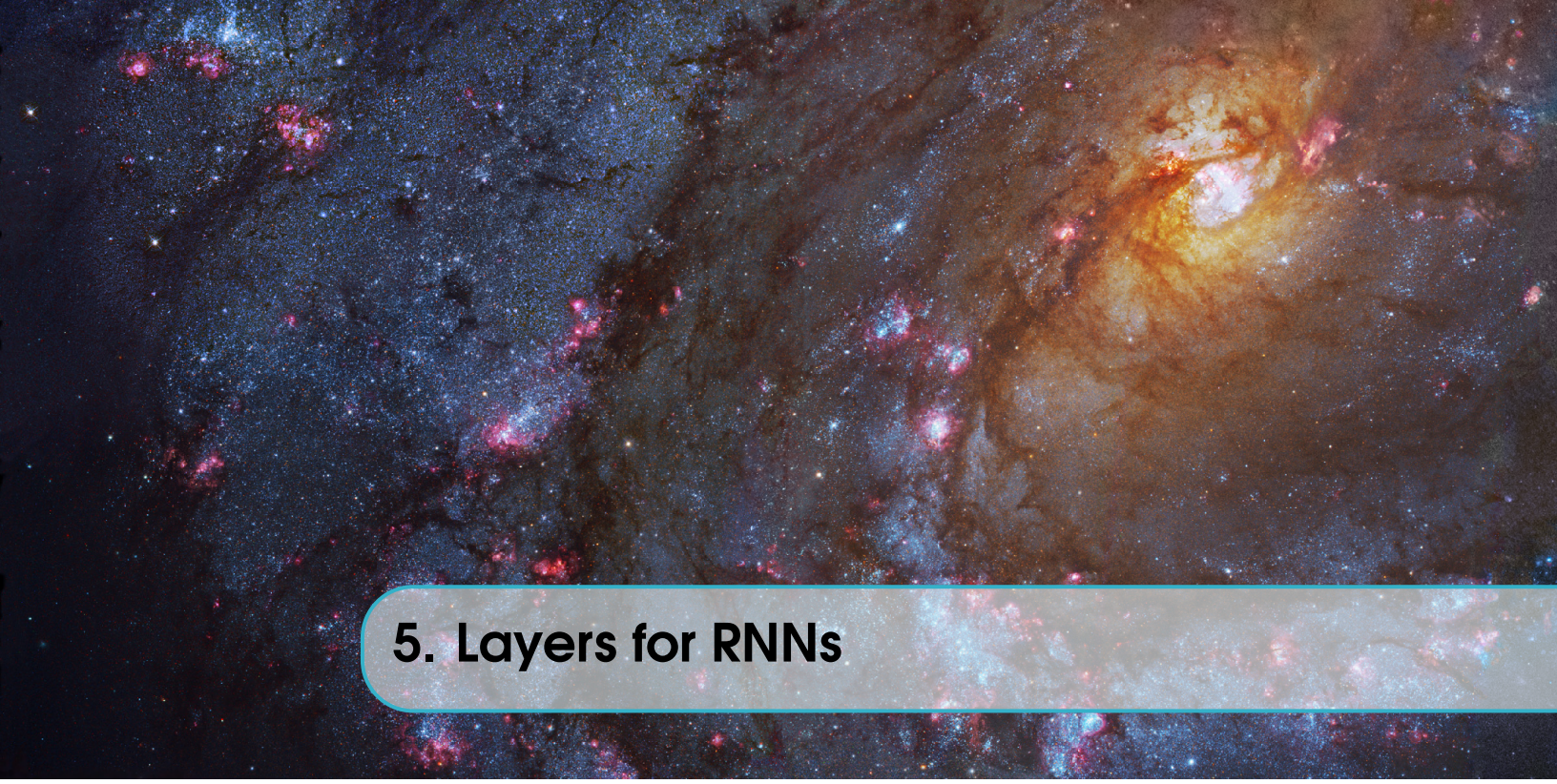
Pooling layers have no weights. Their job is to aggregate inputs using an aggregation function. To know more check Pg 457 of primary resource.

4.2.1 MaxPooling

MaxPooling layers simply just grab the maximum value of each receptive field. This introduces invariance which causes the model to generalise better in many situations thus performing better. Check Pg 457 of primary resource for more information.

4.2.2 AveragePooling

In some cases max pooling actually hinders the performance of the model as there is a loss of a lot of information and a lot of invariance is not very good in all situations. Thus average pooling introduces equivariance. In this case it computes the mean instead of the max of each receptive field.



5. Layers for RNNs

5.1 Simple RNNs

In a RNN a single neuron receives two connections. The first weight is its own weight and the second is the output of the previous layers. Thus we can compute the output of a recurrent layer of a single instance using the equation below.

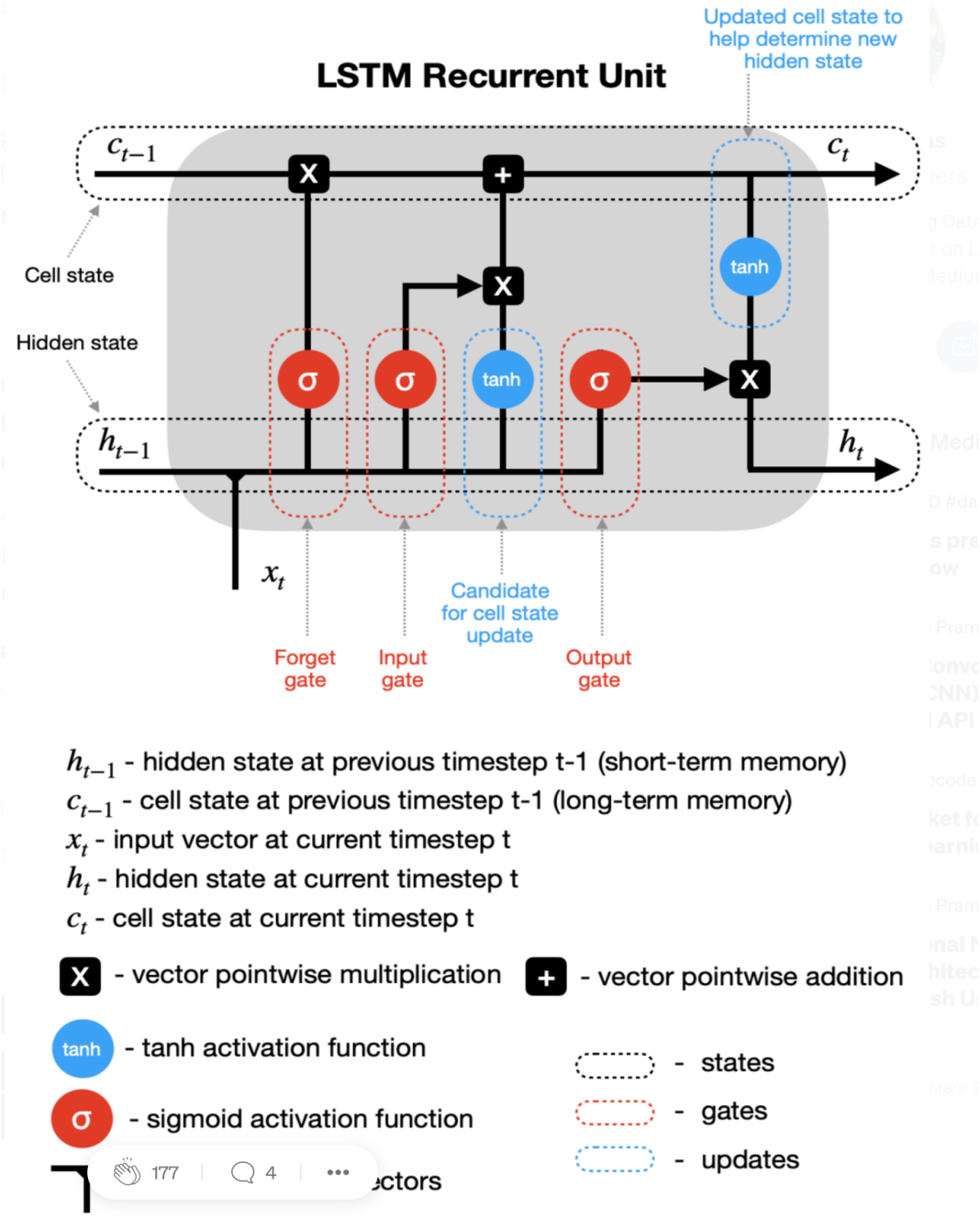
However, just like feedforward neural networks we can compute a recurrent layer's output in one shot for the whole batch by placing all the inputs at t in an input matrix as shown below.

$$y_{(t)} = \phi(W_x^T x_{(t)} + W_y^T y_{(t-1)} + b) \tag{5.1a}$$
$$= \phi([X_{(t)} Y + (t-1)]W + b) \tag{5.1b}$$

In this case $W = \begin{bmatrix} W_X \\ W_y \end{bmatrix}$

Note :- In a sequence to vector RNN pass the paramter $return\_sequences = True$ in all Recurrent layers except the last one. For more details check Pg 505 and 507 of primary resource.

5.2 LSTM



LSTMs and GRUs are a bit to complex to explain without the equations and jargon at the moment. So check out Pg 515 - Pg 521 of primary resource.

6. Attention Mechanisms

Attention Mechanisms are pretty self explanatory. What this means is that when performing some task such as language translation we only pay attention to the important core words and ignore the unimportant words.
The way this works is that once the encoder produces the final state, instead of directly passing it through the decoder, we perform a weighted sum of all the encoder outputs. For example $\alpha_{(t,i)}$ is the i^{th} encoder output at t^{th} decoder time step. So if one of these values is much larger then more attention will be paid to it.

6.1 Bahdanau Attention

In this attention mechanism we use an intermediate model to get the values that were discussed above. All this model is composed of is a TimeDistributed Dense layer with a softmax activation function (to output a probability score). Thus higher the score more the attention. More info on Pg 550 of primary source