

# 1 Deep Learning Theory

The following section covers the most important background theory for the experiments in Section ???. This includes an introduction to various types of neural networks, as well as an introduction to the optimization of such networks.

## 1.1 General Machine Learning Terminology

The following section covers the general terminology and some algorithms, that will be used through this paper.

**Loss Function:** Training a machine learning model is generally done by minimizing a pre-defined **loss function**, which is used for measuring the error of the model. One common loss function for regression is the **Mean Squared error (MSE)**. Let  $f^*(x)$  be a vector of ground truth observations and  $f(x; \theta)$  be an estimation of  $f^*(x)$ . Then, MSE is defined as

$$MSE = \frac{1}{n} \sum_{i=1}^n \left( f^* \left( x^{(i)} \right) - f \left( x^{(i)}; \theta \right) \right)^2. \quad (1)$$

Thus, MSE measures the average squared difference between the ground truth observations and the estimated observations [3].

**Optimizer:** The minimization of the loss function is usually done by an **optimizer**, which is a type of algorithm, that is typically some type of variation of **gradient descent**. Gradient descent is a process that iteratively updates the parameters of a function by

$$\theta \leftarrow \theta - \eta \nabla_{\theta} L \left( f \left( x^{(i)}; \theta \right), f^* \left( x^{(i)} \right) \right) \quad (2)$$

where  $\eta$  is called the **learning-rate**, which controls the stepsize of each update, and  $L$  is a loss function such as MSE.

One of the most used optimizers is **ADAM**, which is an optimizer that uses **momentum** and **adaptive learning-rate** to accelerate its learning. Momentum works by accumulating an exponentially decaying moving average of past gradients and continuing to move in their direction, which especially accelerates learning in the face of high curvature, small but consistent gradients, or noisy gradients. Adaptive learning rate is a mechanism for adapting individual learning rates for the model parameters. The idea is as follows; if the partial derivative of the loss, with respect to a given model parameter, remains the same sign, then the learning rate should increase. If that partial derivate changes sign, then the learning rate should decrease [2].

**Batch:**

**Epoch:** bla bla bla

**Layer Normalization:** bla bla bla

**Activation function:** bla bla bla

**Residual Connection:** bla bla bla

**Evaluation Metric:** bla bla bla

---

**Algorithm 1** Adam [2]

---

**Require:** Learning rate  $\eta$

**Require:** Exponential decay rates for moment estimates,  $\rho_1$  and  $\rho_2$  in  $[0, 1)$

**Require:** Small constant  $\delta$  used for numerical stabilization

**Require:** Initial parameters  $\theta$

Initialize 1st and 2nd moment variables  $\mathbf{s} = \mathbf{0}, \mathbf{r} = \mathbf{0}$

**while** stopping criterion not met **do**

    Sample a minibatch of  $m$  random observations from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$

    Compute gradient:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

$t \leftarrow t + 1$

    Update biased first moment estimate:  $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$

    Update biased second moment estimate:  $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$

    Correct bias in first moment:  $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$

    Correct bias in second moment:  $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$

    Compute update:  $\Delta \theta = -\eta \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}} + \delta}}$

    Apply update:  $\theta = \theta + \Delta \theta$

---

**Dropout:** bla bla bla

### 1.1.1 Optimizer

### 1.1.2 Loss Function

### 1.1.3 Layer normalization

## 1.2 Feedforward Neural Networks

**Feedforward neural networks** are the most basic type of neural networks. The aim of a feed-forward neural network is to approximate some function  $f^*$ , by defining a mapping  $\mathbf{y} = f(\mathbf{x}; \theta)$  and learning the parameters  $\theta$ , that results in the best approximation of  $f^*$ . These models are called **feedforward** because there are no **feedback** connections in which the outputs of the model are fed back into itself. Instead, information flows through the function being evaluated from  $\mathbf{x}$ , through the intermediate computations used to define  $f$ , and finally to the output  $\mathbf{y}$ . Feedforward neural networks generally consists of multiple **layers**, arranged in a chain structure, with each layer being a function of the layer that preceded it [2].

### 1.2.1 Fully-connected Layers

The most simple type of layer found in a feedforward neural network is the **fully-connected layer**. The fully-connected layer usually consists of some learnable parameter matrix  $\mathbf{W}$  and learnable parameter vector  $\mathbf{b}$ , as well as a non-linear **activation function**  $g$  (which will be covered further in Section 1.1). In this case, the  $i$ 'th layer is defined as

$$\mathbf{h}^{(i)} = \begin{cases} g^{(i)}(\mathbf{W}^{(i)\top} \mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}) & \text{if } i > 1 \\ g^{(1)}(\mathbf{W}^{(1)\top} \mathbf{x} + \mathbf{b}^{(1)}) & \text{if } i = 1 \end{cases} \quad (3)$$

Thus, for a neural network with  $n$  layers, we have the mapping [2]

$$\mathbf{y} = f(\mathbf{x}; \theta) = \mathbf{h}^{(n)}. \quad (4)$$

### 1.2.2 Convolutional Layer

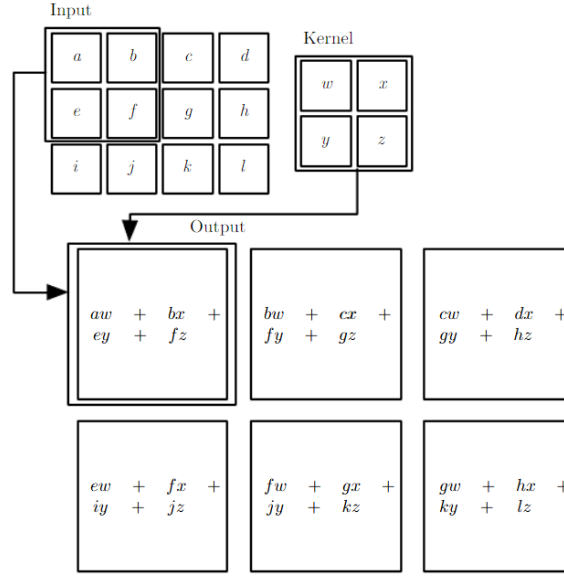


Figure 1: An example of applying a 2d kernel on an input [2].

A **convolutional layer** is a specialized kind of feedforward layer, usually used in analysis of time-series or image data. If a network has at least one convolutional layer, it is called a **Convolutional neural network (CNN)** [2].

The convolutional layer consists of a set of **kernels**, each to be applied to the entire input vector, where each kernel is a learnable parameter matrix  $k \times k$  [4]. Each kernel is applied on the input to produce a **feature map**. The kernels are applied to the input by "sliding" over the input (where the step size is called **stride**). Each  $k \times k$  grid of the input is then used to compute the dot-product between the grid and each kernel, which is then placed in the corresponding feature map of each kernel, as visualized in Figure 1. [1]. To control the dimensions of the output, one might **pad** the sides with a constant value. Commonly, zero is used as the padding-value.

As seen in Figure 1, each kernel produces a linear combination of all pixel values in a neighbourhood defined by the size of the kernel. Thus, unlike a fully-connected layer, a convolutional layer captures the high correlation between a pixel and its neighbours. Further, by limiting the size of the kernel, the network will use much fewer parameters, than if it was replaced by a fully-connected layer [2].

### 1.3 Recurrent Neural Networks

**Recurrent neural networks (RNNs)** are a family of neural networks for processing sequential data. Figure 2 illustrates the general setup of such a network, which maps an input sequence of  $x$  values to a corresponding sequence of output  $o$  values. Generally, a RNN consists of three parts: (1) the input ( $x^{(i)}$ ), (2) the hidden state ( $h^{(i)}$ ), and (3) the output ( $o^{(i)}$ ). During inference, the model maps each input value to an output value in a sequential matter, where it first maps the first input value, then the second, then the third, and so forth. The network maps each input value to an output value by making use of the hidden state from the preceding step, where the first hidden state has to be initialized [2].

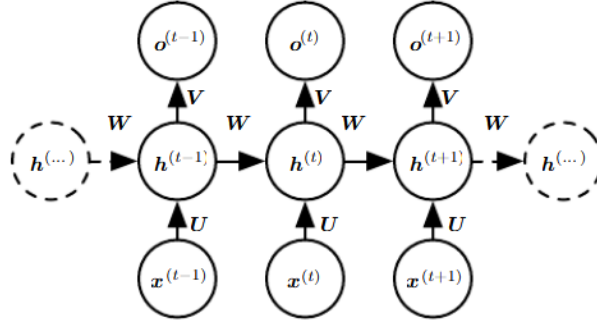


Figure 2: An illustration of an RNN [2].

### 1.3.1 Convolutional Long Short-Term Memory

One common recurrent neural network unit is the **convolutional long short-term memory (ConvLSTM)**, which is an adaptation of the standard **long short-term memory (LSTM)** for sequences of images. Both LSTMs work by potentially stacking multiple **cells** together, such that each LSTM cell is taking in outputs of the preceding LSTM cell as its input.

The idea of both LSTM cells is to create paths through time that have derivatives that neither vanish nor explode. This is done by introducing a memory cell  $C_t$ , which accumulates state information over a long duration. This cell is accessed, written and cleared by several controlling gates. The model learns during training, when to access, write and clear this memory cell. By using the memory cell and gates to control the flow of information, the gradient will be trapped in the cell and thus be prevented from vanishing too quickly [2, 5].

The ConvLSTM cell consists of three gates. The first gate is the input gate  $I_t$ , which controls whether or not to accumulate the information of a new input to the cell. This gate is defined as

$$I_t = \sigma(W_{xi} * X_t + W_{hi} * H_{t-1} + W_{ci} \circ C_{t-1} + B_i) \quad (5)$$

where  $X_t$  is the current input image,  $H_{t-1}$  is the hidden state of the previous time step,  $C_{t-1}$  is the cell output of the previous time step, and  $W_{xi}$ ,  $W_{hi}$ ,  $W_{ci}$  and  $B_i$  are learnable parameters

The second gate is the **forget gate** unit  $F_t$  (at time step  $t$ ), which controls whether or not to "forget" the status of the cell output of the previous time step. The forget gate is defined as

$$F_t = \sigma(W_{xf} * X_t + W_{hf} * H_{t-1} + W_{cf} \circ C_{t-1} + B_f) \quad (6)$$

where  $W_{xf}$ ,  $W_{hf}$ ,  $W_{cf}$  and  $B_f$  are learnable parameters.

The last gate is the **output gate** unit  $O_t$ , which controls whether or not the latest cell output will be propagated to the final state  $H_t$ . This cell is defined as

$$O_t = \sigma(W_{xo} * X_t + W_{ho} * H_{t-1} + W_{co} \circ C_t + B_o) \quad (7)$$

where  $W_{xo}$ ,  $W_{ho}$ ,  $W_{co}$  and  $B_o$  are learnable parameters.

By combining the three gates, we get the following definition of the ConvLSTM Cell. Let

$X_1, X_2, \dots, X_t$  be a sequence of input images. Then, at each time step  $t$ , we compute

$$I_t = \sigma(W_{xi} * X_t + W_{hi} * H_{t-1} + W_{ci} \circ C_{t-1} + B_i) \quad (8)$$

$$F_t = \sigma(W_{xf} * X_t + W_{hf} * H_{t-1} + W_{cf} \circ C_{t-1} + B_f) \quad (9)$$

$$C_t = F_t \circ C_{t-1} + I_t \circ \tanh(W_{xc} * X_t + W_{hc} * H_{t-1} + B_c) \quad (10)$$

$$O_t = \sigma(W_{xo} * X_t + W_{ho} * H_{t-1} + W_{co} \circ C_t + B_o) \quad (11)$$

$$H_t = O_t \circ \tanh(C_t) \quad (12)$$

For  $t = 1$ , both  $H_{t-1}$  and  $C_{t-1}$  have to be initialized [5].

### 1.3.2 Gated Recurrent Unit

## 1.4 Transformer

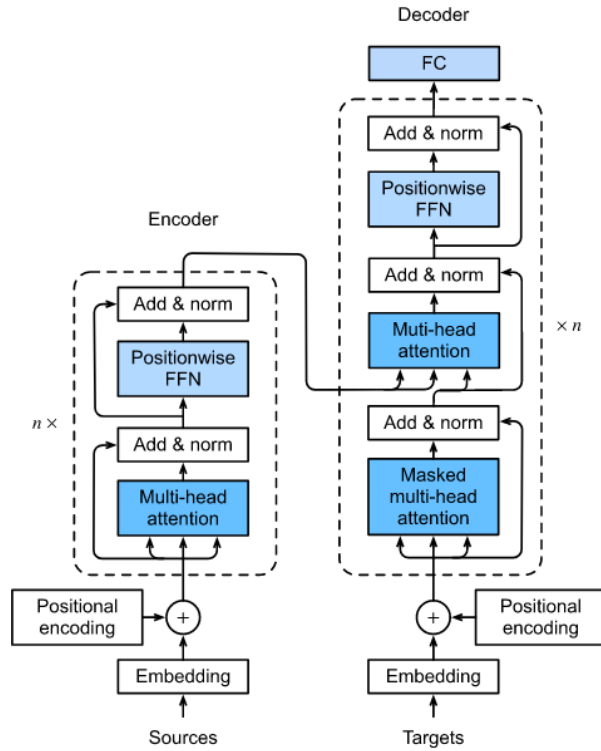


Figure 3: An illustration of the Transformer architecture [7].

The sequential nature of RNNs precludes parallelization with training examples, heavily slowing down the training of these models. Another type of model for sequential data is the **Transformer**, which eschews recurrence and instead relies on an **attention** mechanism to draw dependencies between input and output. The Transformer allows for more parallelization and can reach state of the art results [6].

### 1.4.1 Attention

An attention function is a function that maps a given query and a given set of key-value pairs to an output. In this function the query, keys, values, and output are all vectors, and the output is a weighted sum of the values, where the weight of each value is computed by a compatibility function of the query with the corresponding key [6]. Figure 4 illustrates the two types of attention functions used in the Transformer.

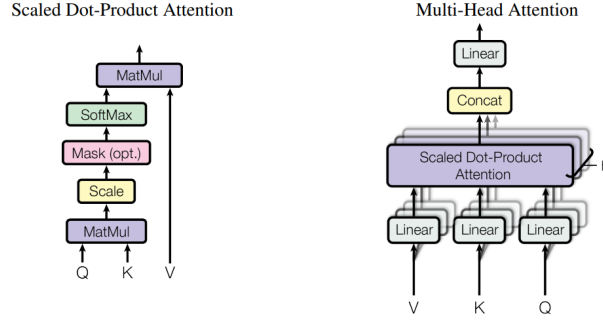


Figure 4: Illustration of (lef) the Scaled Dot-Product Attention, and (right) the Multi-Head Attention, which consists of several attention layers running in parallel [6].

**Scaled Dot-Product Attention:** The scaled dot-product attention is the core attention function of the transform. The function takes queries and keys of dimensions  $d_k$  as input, and values of dimension  $d_v$ . In practice, the attention function is computed on a set of queries, keys and values simultaneously, by packing them into matrices  $\mathbf{Q}$ ,  $\mathbf{K}$ , and  $\mathbf{V}$ , respectively. Thus, the scaled dot-product attention is computed as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \mathbf{V} \quad (13)$$

where the scaling of  $\mathbf{Q}\mathbf{K}^T$  is done to counteract the cases with a large  $d_k$ , which would result in the softmax-function having an extremely small gradient [6].

**Multi-Head Attention:** The **multi-head attention** is the used attention function by the Transformer and is an extention to the scaled dot-product attention. This attention function linearly projects the queries, keys and values  $h$  times with different, learned linear projections to  $d_k$ ,  $d_k$ , and  $d_v$  dimensions, respectively. For each of the projected queries, keys and values, the scaled dot-product attention is applied, yielding  $d_v$ -dimensional output values. These output values are finally concatenated and further projected, resulting in the final values. By using multi-head attention, the model is allowed to jointly attend to information from different representation subspaces at different positions [6].

#### 1.4.2 The Architecture

The Transformer follow an encoder-decoder structure, where the encoder maps an input sequence  $\mathbf{x}$  to a sequence of continuous representations  $\mathbf{z}$ . The decoder then uses  $\mathbf{z}$  to generate an output sequence  $\mathbf{y}$ , one element at a time. At each step the model consumes the previously generated output element as additional input when generating the next output [6]. Figure 3 illustrates the overall architecture of the Transformer.

**Encoder:** The encoder consists of  $N$  identical layers, where each layer consists of two sub-layers. The first sub-layer is a multi-head **self-attention** layer, and the second sub-layer is a position-wise fully-connected feedforward network. Around each sub-layer is a **residual connection**, which is followed by a round of **layer normalization**. In this self-attention layer the keys, values and queries come from the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder [6].

**Decoder:** The decoder also consists of  $N$  identical layers. In addition to the two sub-layers in each encoder layer, the decoder also consists of a third sub-layer, which performs multi-head attention over the output of the encoder stack. Also here, is residual connections used

around each of the sub-layers, followed by layer normalization. To ensure, that the predictions for position  $i$  only depends on the known outputs at positions less than  $i$ , the self-attention sub-layer in the decoder stack is modified. This self-attention sub-layer, allows each position in the decoder to attend to all positions in the decoder up to and including that position. The self-attention sub-layer is modified by masking out all values in the input of the softmax which correspond to illegal connections [6].

**Feedforward Networks:** Each of the layers in the encoder and decoder contains a fully-connected feedforward layer, consisting of two linear transformations with the ReLU activation-function applied in between, as described below [6]

$$FFN(\mathbf{x}) = \max(0, \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1) \mathbf{W}_2 + \mathbf{b}_2. \quad (14)$$

**Positional Encoding:** As the model does not contain any recurrences nor any convolutions, it has to carry some other type of information to know about the relative or absolute position of the elements in the input sequence. This is done by adding **positional encodings** to the input of the encoder and decoder stacks. The positional encoding is a vector of size  $d_{model}$  and is defined as the following

$$PE_{pos,2i} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad (15)$$

$$PE_{pos,2i+1} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad (16)$$

$$(17)$$

where  $pos$  is the position and  $i$  is the dimension [6].