Research

NNs
-fundamentals: epochs, tanh, ReLU, layers blabla schau code
-feed forward for frame prediction

-CNN for pooling and 1d Cnn examples

Linear [12] is one of the most important components in building a DNN. It creates a layer of neurons. It is mandatory to specify the parameters, input size and output size. The number of neurons that are initialized corresponds to the output size. The input size corresponds to the number of incoming connections of a neuron. This component performs a linear transformation of the incoming data [12]. This corresponds to the equation 6.1.
ReLU is a component that can be placed after a linear component. This provides the execution of the activation function ReLU, as discussed earlier, on the incoming data of a neuron.
Dropout is a component that deactivates neurons in a layer at a pre-determined probability [11, 52].

Deep learning is a machine learning concept based on artificial neural networks. For many applications, deep learning models outperform shallow machine learning models and traditional data analysis approaches
https://link.springer.com/article/10.1007/s12525-021-00475-2

In the last few years, the deep learning (DL) computing paradigm has been deemed the Gold Standard in the machine learning (ML) community. Moreover, it has gradually become the most widely used computational approach in the field of ML, thus achieving outstanding results on several complex cognitive tasks, matching or even beating those provided by human performance. One of the benefits of DL is the ability to learn massive amounts of data. The DL field has grown fast in the last few years and it has been extensively used to successfully address a wide range of traditional applications. More importantly, DL has outperformed well-known ML techniques in many domains, e.g., cybersecurity, natural language processing, bioinformatics, robotics and control, and medical information processing, among many others. Despite it has been contributed several works reviewing the State-of-the-Art on DL, all of them only tackled one aspect of the DL, which leads to an overall lack of knowledge about it. Therefore, in this contribution, we propose using a more holistic approach in order to provide a more suitable starting point from which to develop a full understanding of DL. Specifically, this review attempts to provide a more comprehensive survey of the most important aspects of DL and including those enhancements recently added to the field. In particular, this paper outlines the importance of DL, presents the types of DL techniques and networks.
https://link.springer.com/article/10.1186/s40537-021-00444-8?fromPaywallRec=false

Deep learning (DL), a branch of machine learning (ML) and artificial intelligence (AI) is nowadays considered as a core technology of today's Fourth Industrial Revolution (4IR or Industry 4.0). Due to its learning capabilities from data, DL technology originated from artificial neural network (ANN), has become a hot topic in the context of computing, and is widely applied in various application areas like healthcare, visual recognition, text analytics, cybersecurity, and many more.
https://link.springer.com/article/10.1007/s42979-021-00815-1#Sec2

DL:

Deep learning (DL) is a subset of machine learning (ML) that has become a widely used computational approach in the field of ML due to its impressive results. One of the benefits is the ability to learn massive amounts of data, which is why it is applied in various application areas, including emotion recognition.
https://link.springer.com/article/10.1007/s42979-021-00815-1#Sec2
https://link.springer.com/article/10.1186/s40537-021-00444-8?fromPaywallRec=false
https://www.ll.mit.edu/sites/default/files/publication/doc/2018-05/2016_Khorrami_ICIP_FP.pdf

DL is based on Artificial Neural Networks (ANN), which simulate a network of model neurons in a computer.
http://www.lmse.org/assets/learning/bioinformatics/Reading/Krogh2008NatureBiotech_ANN.pdf

Artificial neural networks are popular machine learning techniques that simulate the mechanism of learning in biological organisms. The neurons in the nervous system are connected to each other through synapses, using axons and dendrites. Living organisms learn by responding to external stimuli, which can result in changes to the strength of synaptic connections.

In ANNs, computational units which represent the neurons, are interconnected through weights, analogous to synaptic strengths in biological neurons. Each neuron receives inputs that are scaled by weights, influencing the function that the neuron computes. The network processes these inputs by transmitting the resulting values from the input neurons through to the output neurons, using the weights as modifying factors. Learning in these networks involves modifying these weights based on training data, which consists of input-output pairs. For example, in emotion recognition, features and their corresponding labels (arousal and valence) train the network. In this case, features are the input and the labels are the outputs. The network adjusts weights in response to prediction errors with the goal to modify the computed function to make the predictions more correct in future predictions. This process, called model *generalization*, allows the network to apply learned patterns to new, unseen data.
http://ndl.ethernet.edu.et/bitstream/123456789/88552/1/2018_Book_NeuralNetworksAndDeepLearning.pdf Seite 1-2

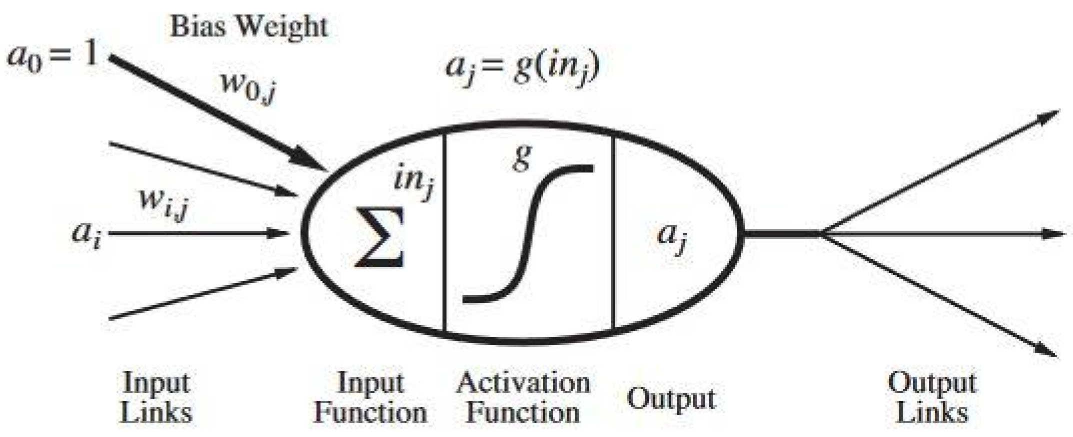


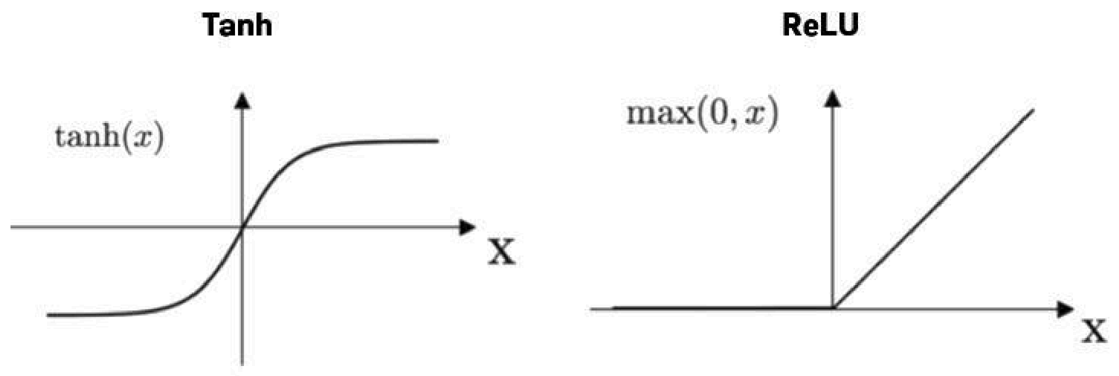
Figure: The structure of artificial neuron
(https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf)

The figure 0.0 demonstrates a simple mathematical model of the neuron.
Let us break down what each part of the neuron model means:

- Input Links: These are the connections from the previous layer's neurons to the current neuron. Each connection has an associated weight wi,j which determines the strength of the signal passed to the neuron. ai is the activation from a neuron in the previous layer. It is the output of neuron i that is being fed into neuron j. Bias Weight (a0 = 1 and w0,j) effectively has the role of shifting the activation function to enable accurate responses, independent of other inputs.
- Input function: The function sums all weighted inputs, including the bias weight. The formula is expressed like this:
  inj=∑i=0nwi,jai
  where inj is the input to the neuron before the activation function is applied.
- Activation Function: It takes the input inj and transforms it into the output activation aj.
- Output: This is the result after the activation function g is applied to the weighted sum of inputs inj. The output activation for this neuron is then used as input to neurons in the next layer or as part of the final output if this is the output layer. The output is calculated as:
  aj=g(∑i=0nwi,jai) (but take from marlene)
- **Output Links**: These are the connections that transmit the activation to the neurons in the next layer. Just like input links, each output link will have its own weight that will adjust the signal for the next layer's processing.

https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf

There are different activation functions that can be applied, like Rectified Linear Unit (ReLU), Tanh function, threshold function and logistic function. (BUCH) Since only ReLU and Tanh functions are used in the experiments in this work, we will mainly focus on them.



Visualization of Tanh and ReLU activation functions.
https://itsudit.medium.com/activation-functions-in-deep-learning-understanding-the-role-of-activation-functions-in-neural-423694f7f54e

The ReLU function is a widely used activation function in deep learning, because it has the advantage of not activating all the neurons at the same time. The neurons will be deactivated only if the output of linear transformation is less than zero. In Figure 0.0 you can see that any negative input values result in a zero and any positive value x returns the value back.
The function can be written as f(x) = max(0,x).

$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$$

https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/

Another commonly used activation function is tangent hyperbolic or simly tanh. It is a s-shaped curve that maps any input to a value between -1 and 1. In this work, the tanh function is used in the output layer to constrain the output to between -1 and 1, which makes sense in emotion recognition tasks where arousal and valence are predicted, which typically have normalized ranges.
https://www.researchgate.net/publication/351728572_Neuroergonomic_Stress_Assessment_with_Two_Different_Methodologies_in_a_Manual_Repetitive_Task-Product_Assembly

The function is defined as:
f(x) = 2 / (1 + e^(-2x)) — 1
https://itsudit.medium.com/activation-functions-in-deep-learning-understanding-the-role-of-activation-functions-in-neural-423694f7f54e

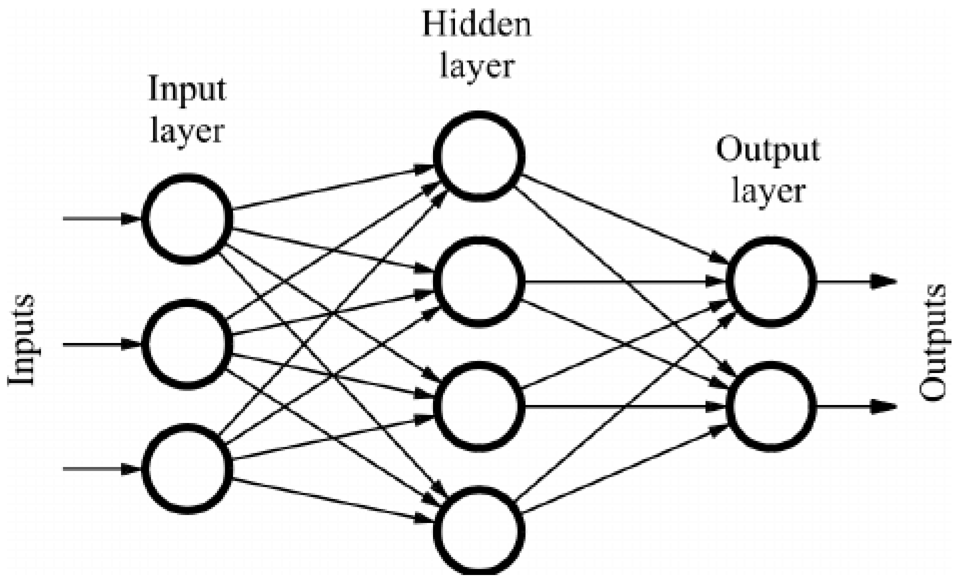
After deciding on the mathematical model for individual neurons, the next step is connecting them together, which forms a network.
https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf

There are several types of networks in Deep Learning like Multilayer Perceptron (MLP), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN) and Transformers that will be explained later on in this Chapter.

Feedforward:

Quelle: Goodfellow!

The most basic architecture is feedforward neural network (FFNN) or multilayer perceptron (MLP). The goal is to approximate the function $y = f$^∗ (x), which maps input x to output y. For example, in emotion recognition function f∗ maps input features x (such as facial expressions or physiological signals) to emotional states like arousal or valence (output y). FFNN defines a mapping $y = f(x; \theta)$ and learns the values of the parameters θ through training processes. The idea is to adjust θ so that the network's predicted mapping f(x;θ) comes as close as possible to the true function f^∗.
A feedforward neural network is called 'feedforward' because computations flow in a single direction - from the input x, through intermediate layers, to the output y. There are no feedback connections, which means that the output of the model is not fed back into itself at any point.



https://www.researchgate.net/figure/Feedforward-neural-network_fig1_329586439
Figure: FFNN architecture

Figure 0.1 demonstrates a simple FFNN architecture. Each circle represents a neuron, and the lines between neurons represent the connections across which the signals travel. The Figure 0.0 from before that demonstrated the structure of an artificial neuron is a “zoom in” on an individual neuron, which is one of the circles in Figure 0.1
It consists of three layers:

Input Layer: This layer accepts the data (typically in the form of a vector) and passes it to the next layer without performing any computation.
https://www.analyticsvidhya.com/blog/2021/03/basics-of-neural-network/

Hidden Layer– Each neuron in a hidden layer receives inputs from all the neurons in the previous layer, multiplies these inputs by its weights, and then passes the result through an activation function. The output of each neuron is then used as input to the next layer. https://deepai.org/machine-learning-glossary-and-terms/hidden-layer-machine-learning

Output layer– This layer is the last layer of the network, and its neurons represents the predictions of the model.https://www.analyticsvidhya.com/blog/2021/03/basics-of-neural-network/

To sum it up, when the feedforward network accepts an input x and produces an output y, information flows forward through the network. The process of inputs x providing the initial information that then propagates up to the hidden units at each layer to produce y, is called *forward propagation*. (Goodfellow)

The “Multilayer” part in the MLP comes from the fact that this type of network includes one or more hidden layers. For comparison, a single-layer neural network has no hidden layers, only an input and an output, and is called a perceptron. So as soon as one hidden layer is added, it can be considered a multilayer network.
https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf
It is important to note that even though the above neural network has one hidden layer, it is not automatically considered a deep neural network. Networks that have multiple (at least two) hidden layers are classified as deep neural networks. These networks are part of deep learning because they have the "depth" to represent more complex hierarchical features.
https://link.springer.com/article/10.1007/s42979-021-00815-1
Nevertheless, the above FFNN incorporates the principles of artificial neural networks, which are the foundation of modern deep learning, which is why it is included in this Chapter.

Quelle: https://link.springer.com/book/10.1007/978-3-658-30211-5
Training a neural network is an important phase in developing a deep learning model. The training process involves iterative forward and backward passes through the network to adjust the weights and minimize prediction error.

During forward propagation, an input x is passed through the network, producing an output. Let us say, there is a model that takes an input x and maps it through two functions f1 and f2 associated with weights w1 and w2, respectively. The output of the first function, h1, serves as the input to the second function, which produces the predicted output y^. It can be represented as h1=f1(w1 · x) and y^=f2(w2 · h1). The loss is computed by the function loss = L(f2(w2f1(w1x)), y)= L(y^,y), which measures the difference between the predicted output y^ and the true output y. The model’s goal is to adjust weights in order to reduce this loss, which represents the error in prediction.

Now begins the process of backpropagation, where the model learns. To update the weights w1 and w2, we calculate the gradients of the loss function with respect to each weight. Using the chain rule, the gradient with respect to w2 is represented as:

∂L/w2=∂L/∂y^ · ∂y^/∂w2,
where ∂y^/∂w2 can be further broken down using the chain rule through the function f2.

Similarly, the gradient with respect to w1 is:

∂L/∂w1=∂L/∂y^ · ∂y^/∂h1 · ∂h1/∂w1,
where ∂y^/∂h1 is taken from the output of function f2 and ∂h1/∂w1 from the output of function f1.

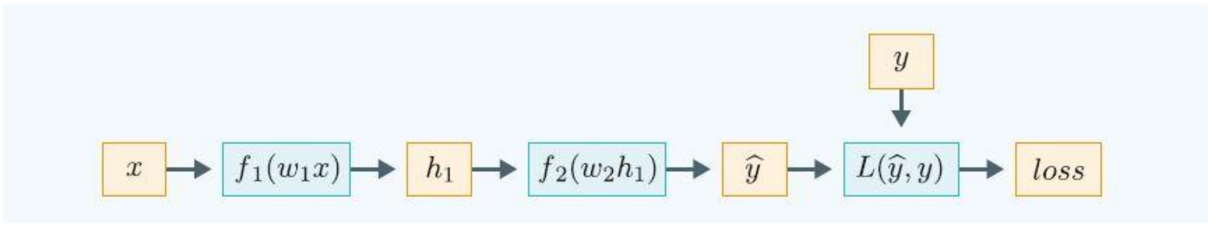
After obtaining the gradients, the weights are updated in the opposite direction of the gradients to minimize the loss, using a *learning rate* α:

w1=w1−α · ∂L/∂w1
w2=w2−α · ∂L/∂w2

The process is repeated across many iterations, known as *epochs*, with the forward and backward passes being recalculated each time, gradually improving the model's predictions.

The figure 0.2 illustrates the flow of information during the forward and backward propagation steps.

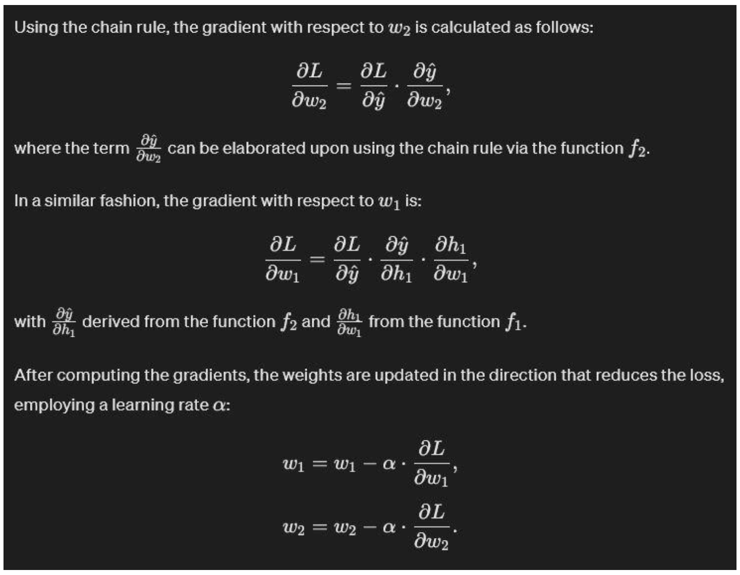
Figure 0.2 : simple neural network model that goes through the backpropagation process



$$\begin{tikzcd} & & & & & & y \arrow[d] & \\ x \arrow[r] & f_1(w_1x) \arrow[r] & h_1 \arrow[r] & f_2(w_2h_1) \arrow[r] & \hat{y} \arrow[r] & L(\hat{y}, y) \arrow[r] & loss \end{tikzcd}$$

https://link.springer.com/book/10.1007/978-3-658-30211-5

Chain rule footnote:
https://en.wikipedia.org/wiki/Chain_rule
The chain rule is a formula that computes the derivative of a composite function as the product of the derivatives of its constituent functions.



Using the chain rule, the gradient with respect to w_2 is calculated as follows:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2},$$

where the term $\frac{\partial \hat{y}}{\partial w_2}$ can be elaborated upon using the chain rule via the function f_2.

In a similar fashion, the gradient with respect to w_1 is:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1},$$

with $\frac{\partial \hat{y}}{\partial h_1}$ derived from the function f_2 and $\frac{\partial h_1}{\partial w_1}$ from the function f_1.

After computing the gradients, the weights are updated in the direction that reduces the loss, employing a learning rate α:

$$w_1 = w_1 - \alpha \cdot \frac{\partial L}{\partial w_1},$$

$$w_2 = w_2 - \alpha \cdot \frac{\partial L}{\partial w_2}.$$

When building a deep neural network, there are many components that can be used in the model. The ones used in the experiments will be presented here.

The “Linear” or ‘torch.nn.Linear’ in PyTorch is a fundamental component for constructing neural networks, also known as a fully connected or *dense* layer. It performs a linear transformation that maps a given set of input data to a new space of output data. This layer applies a weighted sum across input features to produce output features, as shown in equation NUMBER earlier. https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

The “Dropout” or “torch.nn.Dropout” in PyTorch is a regularization technique where nodes (from input and hidden layers) along with their connections are randomly deactivated during training. This process creates a new, thinned network at each training step, which helps to reduce overfitting by preventing complex co-adaptations on training data.
https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html
https://arxiv.org/abs/1207.0580
https://towardsdatascience.com/dropout-in-neural-networks-47a162d621d9

Footnote overfitting:Overfitting is a common issue in machine learning where a model learns to fit the training data too closely, preventing it from generalizing effectively to new, unseen data.
https://iopscience.iop.org/article/10.1088/1742-6596/1168/2/022022/pdf

Early stopping is another regularization technique in neural networks that can be used to stop the training process if there is no improvement after a given number of epochs. It also helps to reduce overfitting, because it does not allow the model to train too much in the training data.
https://link.springer.com/chapter/10.1007/978-3-642-35289-8_5#preview
https://pytorch.org/ignite/generated/ignite.handlers.early_stopping.EarlyStopping.html

In the following, the training pipeline is described step by step as it was done in the experiments.

1. **Loading and Preprocessing Data**: Data is loaded and preprocessed from CSV files, splitting it into features and labels, which are then converted into PyTorch tensors.

   https://pytorch.org/docs/stable/tensors.html

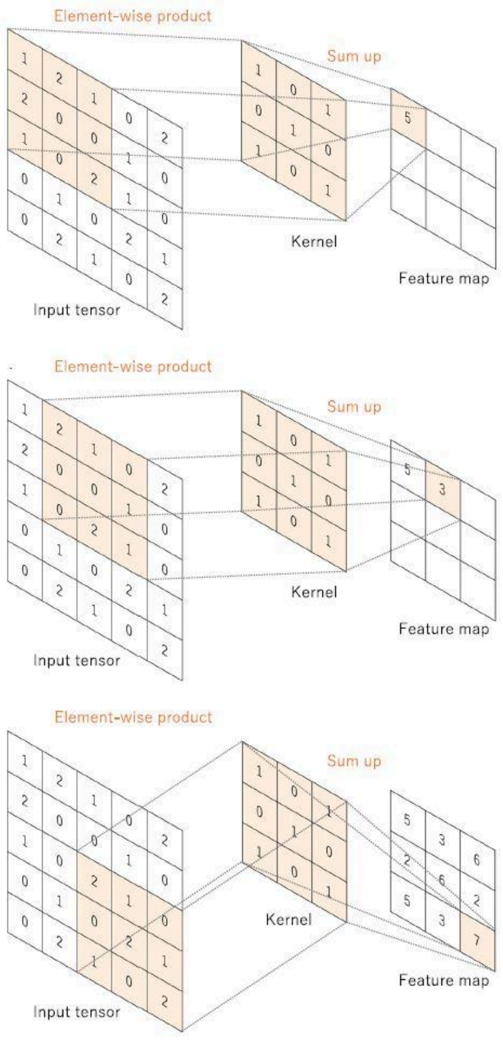
2. **Creating DataLoaders**: DataLoaders for training, development, and testing datasets are prepared to manage batch processing and shuffling of data. DataLoaders will be explained more in depth in Chapter 5.
3. **Defining the Neural Network** with specific layers, activation functions, and output dimensions.
4. **Initializing Hyperparameters** such as input dimensions, hidden layer size, output dimensions, learning rate, and number of epochs.
5. **Instantiate Loss Function and Optimizer**: Separate loss functions for different targets (arousal and valence) are defined, and an optimizer is initialized to update

network weights based on computed gradients.https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

6. **Training Loop Begins**: The model enters a training phase where each epoch consists of:
    - **Zeroing Gradients**: At the start of each batch, gradients are reset to zero to prevent mixing them up with those from previous batches.
    - **Forward Pass**: Data is fed forward through the network to compute predictions.
    - **Computing Loss**: Loss is calculated separately for arousal and valence by comparing predictions with actual labels.
    - **Backward Pass**: Backpropagation is performed to calculate gradients of the loss with respect to network weights.
    - **Update Weights**: Weights are updated using the optimizer.
7. **Validation Check**: At the end of each epoch, the model evaluates performance on the development set to compute metrics like RMSE and MAE for arousal and valence, which will be explained more in depth in Chapter 3.
8. **Early Stopping Criteria**: The process checks for improvements in validation metrics over a predefined number of epochs, known as *patience*. If there's no improvement within this period, the best state of the model is saved, and training can be stopped early to prevent overfitting.
9. **Saving Model**: If the results of the current epoch are better than previous ones, the best metrics are updated and the weights of the model are saved.
10. **Repeat or Stop**: The training continues to the next epoch or it stops if early stopping criteria are met.

CNN:
https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9
Convolutional Neural Network (CNN) is a class of artificial neural networks specifically designed for processing data with a grid pattern, such as images. CNNs are particularly effective in various computer vision tasks due to their ability to automatically and adaptively learn spatial hierarchies of features through a combination of different types of layers: convolutional layers, pooling layers, and fully connected layers.

Convolutional Layer: The convolutional layer is the fundamental part of the CNN that performs feature extraction, which usually consists of convolution operation and activation function. It applies a set of filters, also known as kernels, to the input image to create feature maps. These feature maps transform the input image into features that become more complex with more layers. The convolution operation, illustrated in Figure 3, involves sliding the kernel over the input image, calculating the element-wise product at each position, which extracts features from the image.



Pooling Layer: Following convolutional layers, pooling layers reduce the in-plane dimensions (width and height, not depth) of the input data for the next convolutional layer. It works by summarizing the features in patches of the feature map into a single value.
Max Pooling takes the maximum value from each patch of the feature map. Similarly, Average pooling takes the average value from each patch of the feature map. Both are shown in Figure NUMBER.

Fully Connected Layer: These layers are used where neurons are fully connected to all activations in the previous layer. They are typically placed after several convolutional and pooling layers and function as classifiers that use the features from earlier layers. The output can be used to classify the input image into various classes based on the training dataset.

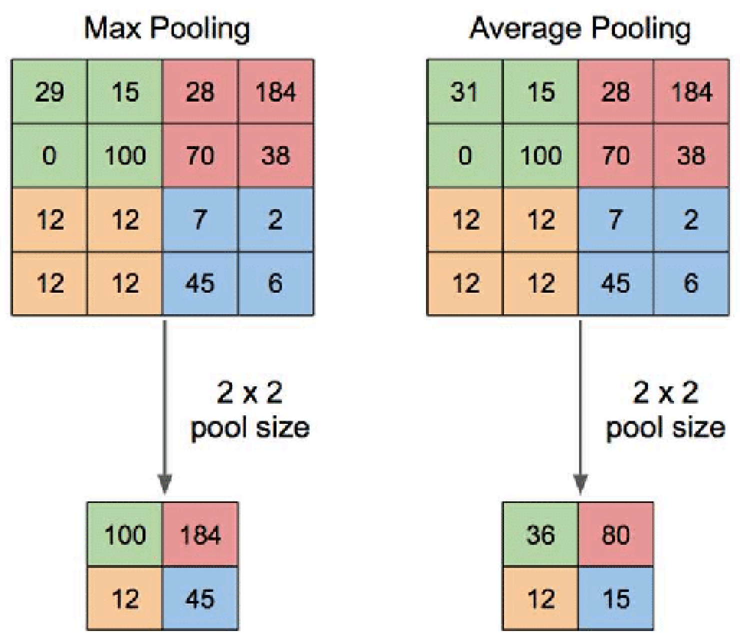


Figure: Example of max pooling operation and average pooling with a 2x2 pixel filter size from 4x4 pixel input.

https://www.researchgate.net/publication/333593451_Application_of_Transfer_Learning_Using_Convolutional_Neural_Network_Method_for_Early_Detection_of_Terry's_Nail

CONV1D MISSING

CNNs have several benefits, including being able to recognize patterns in images regardless of where they appear. This ability, known as shift invariance, means the network can identify the same patterns even if they move around in the image. Additionally, the option to select different kernel sizes allows the network to detect both small, detailed patterns with smaller kernels and larger objects with bigger kernels. In addition, pooling operations not only enhance pattern recognition, particularly for larger objects, but also significantly reduce computational demands and memory usage. This boosts the efficiency and speed of the network, making CNNs effective for complex image analysis tasks.
Quelle: buch deutsch und
https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9

POOLING:

This architecture is effectively an RNN with an additional pooling step that borrows from CNN methodologies. The average pooling here is used to condense the temporal information into a more manageable form before making a final prediction, potentially helping the model to abstract the most essential features from the sequence data.

Since in the conducted experiments in Chapter 4, not classification, but regression is predicted, the whole CNN architecture is not used, but rather the CNN methods, specifically pooling (MAX Pool and AVG Pool) and Conv1D. Temporal pooling compacts the information along

sequence data into a single vector representation, making it more manageable. Conv1D is used in the experiments to apply filters to the sequence data to capture patterns.

Since the experiments conducted in Chapter 4 focus on regression rather than classification, the full CNN architecture is not used. Instead, some CNN methods are used, specifically pooling (MAX Pool and AVG Pool) and Conv1D. Temporal pooling is used to compact the information along the sequence data into a single vector representation, therefore making it more manageable. https://arxiv.org/pdf/2105.04310 Conv1D is used in the experiments to apply filters on the sequence data, capturing patterns effectively.https://link.springer.com/article/10.1007/s11227-022-04431-5#Sec12

Conv1D

- **Function**: Conv1D (1-dimensional convolutional layer) is primarily used to extract features from the sequence data. It operates over the temporal dimension of the data, applying filters to the sequence to capture local dependencies or patterns.
- **Utility in RNN Context**: While GRUs are effective at capturing long-range dependencies within sequence data, Conv1D can complement this by extracting local features that may be indicative of shorter-term patterns within the sequence. This can be particularly useful for tasks where local context within the sequence significantly influences the output.

Max Pool

- **Function**: Max Pooling is a form of down-sampling that uses the maximum value from each segment of the input data in the pooling window. In a 1D context, it reduces the length of the output sequence from the Conv1D layer by selecting the maximum value from each pooling window, thus reducing the sequence length while preserving the most significant features.
- **Utility in RNN Context**: Just like AVG Pooling, Max Pooling in this setting helps to reduce the complexity of the sequence data, making it easier to manage by reducing its dimensionality. It emphasizes the most salient features, which might represent peaks or important markers within the sequence data. This is useful for ensuring that subsequent layers, such as the dense layer (fully connected layer), receive only the most impactful features, potentially enhancing model performance by focusing on the most relevant aspects of the input data.

1D Average Pooling: Post dropout, the model applies 1-dimensional average pooling. This is more commonly seen in CNN architectures where spatial pooling is used to reduce the spatial size of the representation and to increase the field of reception. In this RNN context, temporal pooling (1D pooling) is used to reduce the sequence length dimension and to summarize the features across the time dimension. This simplifies the data by averaging out the features over the sequence, potentially helping to focus on more general features rather than specifics of the input sequence.

RNN:
When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks \cite{norvig}.
https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf

Recurrent Neural Networks (RNNs) are a class of deep learning models designed to handle sequential data by capturing temporal dependencies. Unlike feed-forward neural networks, RNNs integrate an internal memory that allows them to process inputs considering contextual information.

Traditional neural networks don't have the ability to maintain information about previous inputs when processing new data. This limitation is challenging in tasks that involve sequences where the previous data influences the current processing. To address this, RNNs have been developed that allow maintaining a memory of previous computations and therefore enhancing the model's ability to handle sequential data.

Figure: Simple RNN internal operation.

As shown in Figure \ref{fig:rnn-structure}, a simple recurrent unit includes a hidden state (h_t) that is computed based on the current input (x_t) and the previous hidden state (h_{t-1}). This relationship can be captured by the equation:

h_t=g(Wx_t+Uh_t+b),

where h_t is the new hidden state, g is the activation function (typically a hyperbolic tangent), W and U are adjustable weight matrices for the hidden state, b is the bias term and x denotes the input vector.

Even though simple RNNs excel in sequential data processing, they face limitations in retaining information over long sequences due to their short-term memory. To overcome this, more advanced variants such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) have been developed. These architectures introduce mechanisms to selectively retain or forget information, thereby improving the network's ability to learn over longer sequences.

GRU:

The Gated Recurrent Unit (GRU) is a neural network architecture designed to address the limitations of traditional Recurrent Neural Networks (RNNs) ' short-term memory. By simplifying the structure, GRUs offer a more efficient alternative to Long-Short-Term Memory (LSTM) units while maintaining comparable performance.

A GRU unit consists of three main components that manage the flow and updating of information within the network: the update gate, reset gate, and current memory content. These components allow the GRU to selectively update and utilize information from previous steps, allowing it to capture long-term dependencies in sequences.

The structure of a GRU unit is demonstrated in Figure \ref{fig:gru-unit}. The operations are controlled through several gates and states:

- **Update Gate (z_t):** The update gate decides how much of the past information should be carried forward and is defined by:

  $z_t=\sigma(W_z[h_{t-1},x_t]+b_z)$

  It uses a sigmoid function σ to filter the extent to which previous states affect the current state.

- **Reset Gate (r_t):** The reset gate decides how much of the previous information should be forgotten:

  $r_t=\sigma(W_r[h_{t-1},x_t]+br)$

  It is computed similarly to the update gate.

- **Current Memory Content ($^\wedge{\sim}h_t$):** It is calculated based on the reset gate and concatenation of the previous hidden state and current input. The result is passed through an activation function.:

  $^\wedge{\sim}h_t=\tanh(W_h[r_t\cdot h_{t-1},x_t])$

- **Final Memory State (h_t):** The new states determined by a combination of the previous hidden state and the candidate activation:

  $h_t=(1-z_t)\cdot h_{t-1}+z_t\cdot{}^\wedge{\sim}h_t$.

  Additionally, an optional output gate o_t can be defined to control the information flow from the current memory content to the output:

  $o_t=\sigma_o(W_o\cdot h_t+bo)$

GRUs provide a simpler alternative to LSTMs. They reduce the number of tensor operations required for training, which can lead to faster training times. Since GRU offers a solution for sequence modeling with the ability to capture information across long sequences and has a faster training process, it makes it a good neural network architecture choice for the video model, described later in Chapter 4.

https://arxiv.org/pdf/2305.17473

Transformer:

In sequence-to-sequence problems, the initial approaches utilized RNNs. However, this architecture faced limitations while working with long sequences because it tended to lose information from initial elements when new ones were added. Vaswani et al. \cite{attention} introduced the attention mechanism and transformer model architecture to address this issue.

Figure: The Transformer architecture \cite{attention}.

The Transformer, shown in Figure \ref{fig:transformer}, contains an encoder model on the left side and a decoder on the right one. Both have a block of "an attention and a feed-forward network" repeated N times.

Let us now address the fundamentals of the transformer architecture.

Self-attention is a process where a sequence of vectors is transformed into another sequence of vectors. Let us say we have input vectors x1,x2,…,xt and aim to produce corresponding output vectors y1,y2,…,yt, each with a k dimension. To generate each output vector yi, self-attention computes a weighted average of all input vectors. One common method for determining the weights is to use the dot product.
https://peterbloem.nl/blog/transformers

The self-attention mechanism has three essential elements: queries, values, and keys.

Figure: Visualization of the self-attention with key, query, and value transformations. peter

In the self-attention process, each input vector xi is transformed into three roles using weight matrices Wq,Wk,and Wv, representing Query, Key, and Value, respectively. These transformations allow each input vector to interact with itself, making it possible to calculate attention weights and then generate the corresponding output vectors. The query matrix helps determine the output vector yi , and the key matrix calculates other outputs yj. The Value matrix finalizes each output after the weights are determined. The weight layers K, Q, and V are applied to the same encoded input. Since each of these matrices comes from the same input, we can apply the attention mechanism of the input vector to itself.
https://towardsdatascience.com/attention-is-all-you-need-discovering-the-transformer-paper-73e5ff5e0634 and peter

We use the Q, K, and V matrices to calculate attention scores. For example, in the context of neural machine translations, these scores determine 'how much focus to place on other places or words of the input sequence w.r.t a word at a certain position . That is, the dot product of the query vector with the key vector of the respective word we're scoring'

\cite{trans-science}. For example, we calculate the dot products q1 · k1,q1 · k2,q1 · k3, etc. for the first position.

To stabilize gradients, we scale these scores before applying the softmax function. After applying softmax, we multiply the results by the Value matrix.

The formula for this operations is:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Multi-head attention, shown in Figure \ref{multihead}, processes the input through multiple attention mechanisms in parallel, allowing it to focus on different parts of the sequence simultaneously(e.g., longer-term dependencies versus shorter-term dependencies). Each independent attention output is computed, concatenated, and linearly transformed into the expected dimension. The weights W are learnable parameters. Typically, this mechanism employs scaled dot-product attention, but it can be swapped out for other types of attention mechanism.

Quelle: attention

Figure: Multi-Head Attention consists of several attention layers running in parallel. FROM ATTENTION

Since the Transformer model does not have recurrence and convolution, it needs a way to recognize the sequence order. This is achieved by adding "positional encodings" to the input embeddings at the base of both the encoder and decoder stacks. This provides information about the position of tokens in the sequence.

So, the encoder consists of the following components:

- Positional encoding
- Two sublayers: a multi-head self-attention mechanism and a fully connected feed-forward network
- Residual connection around each sublayer for summing up the output
- Layer normalization

The decoder consists of the following components:

- Positional encoding
- Three sublayers: The first is Masked Multi-head attention to prevent positions from attending to subsequent positions. The second one is 'encoder-decoder attention' (also known as cross-attention), which performs multi-head attention over the output of the decoder. The Key and Value come from the output of the encoder, but the queries come from the previous decoder layer. Afterward, there is a fully connected network.
- Residual connection around each sublayer for summing up the output

- Layer normalization

The transformer model consists of two components: an encoder and a decoder.

Since transformer can handle long data sequences, outperforms RNN sequence-to-sequence models and provides good results, it is the used architecture in the fusion methods experiments.