ANN Summary

Lecture 1 - Intro to Neural Networks

Underlying Idea for ANNs

- Brain as a function → **maps received inputs to outputs**
    - Inputs and outputs could be: speech, text, images, videos, labels, entities, words, audio, features

Abstracting from a Biological Model

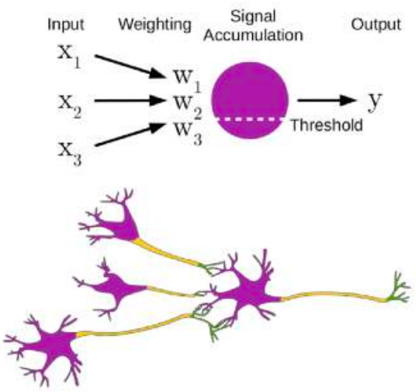


Figure 8: Key elements to model in a neuron [1]

Similarities between Biological and Artificial Neural Networks (NNs)

Biological	Artificial
Stimulus	Input
Receptors	Input Layer
Neural Net	Processing Layers
Threshold (Action Potential)	Activation Function
Response	Output

The Perceptron

- One-layer neural network
- Can compute binary classification → [0,1]
- Threshold function denotes the output
- Output → activation of the sum of all inputs times their weight

$$y = \sigma(\sum x_i w_i)$$

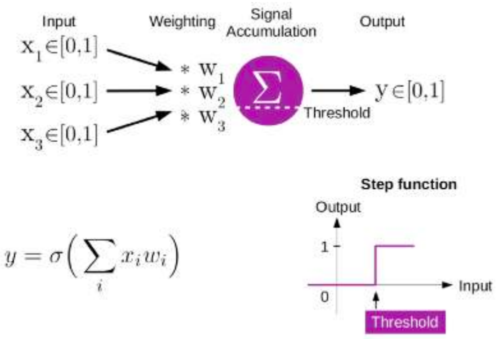


Figure 9: Perceptron [1]

Data Representation

- **Binary** → [0], [1]
- **One-hot encoding** → [1,0,0], …
- **Numerical Data Vector** → [0.42, 0.3, 0.87]

Multi-Layer Perceptron

- consists of multiple layers - each containing at least one perceptron/neuron
- Three different **types of layers**:
    - Input layers
    - Hidden layers
    - Output layer
- **Feed-Forward Network** → every neuron is only connected to neurons in the incoming or upcoming layer (no loops!) → only forward information flow!
- **Fully connected network** → each neuron is connected to each neuron in the layer before

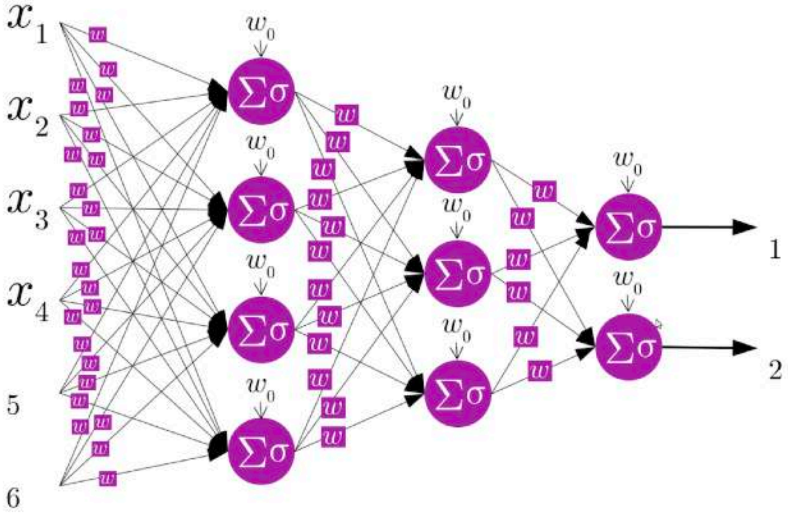
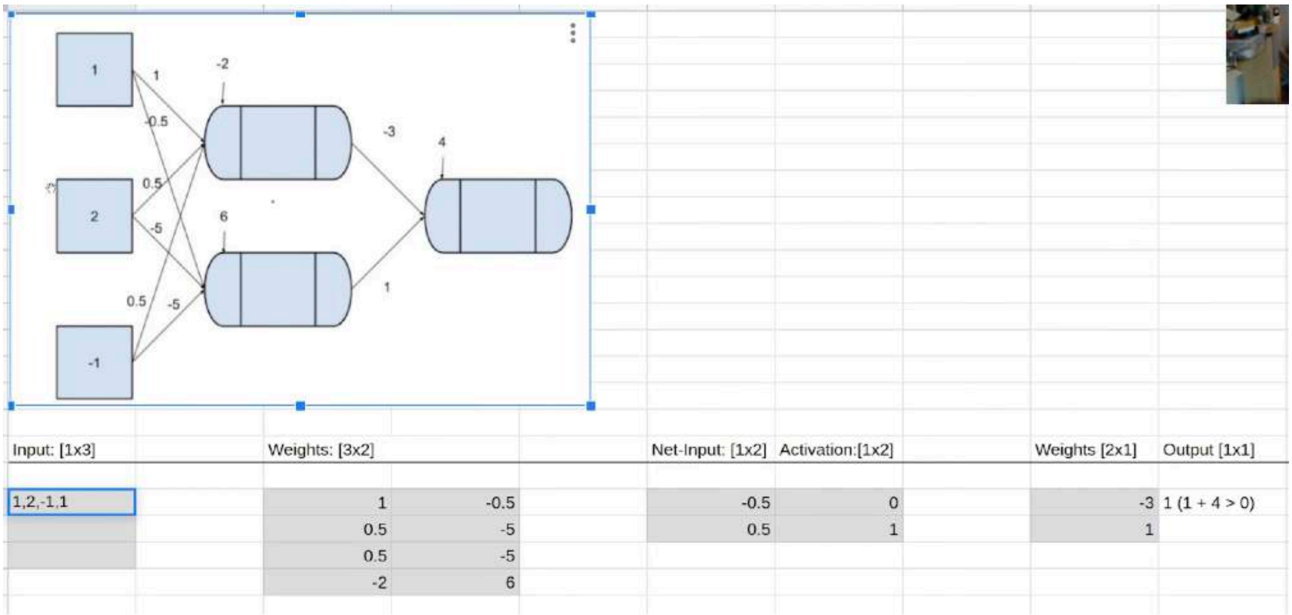


Figure 10: A multi layer perceptron (MLP) [1]

MLP Implementation as Matrix Multiplication

- **Input vector *x*** of size [1 x m] → where m is the size of the input
- **Weight matrix *W*** of size [m x n] → where n is the number of neurons in our MLP-Layer
- **Calculation of net-input** to all perceptron in the layer (of size [1 x n])→ $x \times W$
- **Biases** are incorporated as the base row of W + adding 1 to the end of x
- **Calculation of single MLP-Layer** → $\sigma(x \times W)$, where σ applies the threshold point wise to each vector element



Input: [1x3]	Weights: [3x2]		Net-Input: [1x2]	Activation:[1x2]	Weights [2x1]	Output [1x1]
1,2,-1,1	1	-0.5	-0.5	0	-3	1 (1 + 4 > 0)
	0.5	-5	0.5	1	1	
	0.5	-5				
	-2	6				

Lecture 2 - Training NNs via Backpropagation

Activation Functions

Perceptron calculation → $y = \sigma(\sum_i x_i w_i)$

Sigmoid → $\sigma(z) = \dfrac{1}{1+e^{-z}}$ with $\sigma' = \sigma(z)(1-\sigma(z))$

ReLu (Rectified Linear Unit) → $R(z) = \max(0,z)$ with $R'(z) = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$

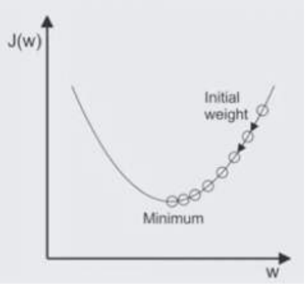
Derivatives - Chain Rule with Leibniz Notation

If a function is defined by a composition $y = f(g(x))$, it can be decomposed into $y = f(u), u = g(x)$.

The derivative w.r.t. x is then computed using the chain rule as $\dfrac{dy}{dx} = \dfrac{dy}{du} \cdot \dfrac{du}{dx}$.

Loss and Gradient Descent

- **Aim** → **Minimize the loss** by adjusting the values of the weights
- NN as a function → $y = \sigma(\sigma(xW_1)W_2)$ (for NN with one hidden layer)
- Loss → $L = \dfrac{1}{2}(t - \hat{y})^2$ (computed as Mean-Squared-Error MSE; t = target, $\hat{y}$ = prediction)
- Finding the min loss via **Gradient Descent:**
  - Let $\vec{w}$ be the vector of all weights in all layers
  - Let $NN(x, w) = y = \sigma(\sigma(xW_1)W_2)$
  - Let Loss $(x, t, w) = L = \sum_{x,t} \dfrac{1}{2}(t - \text{NN}(x, w))^2$
  - Get **Gradient** $\dfrac{dL}{d\vec{w}}$ → notice: $\dfrac{dL}{d\vec{w}} = [\dfrac{dL}{d\vec{w}_1}, \dfrac{dL}{d\vec{w}_2}, \ldots, \dfrac{dL}{d\vec{w}_n}]$
  - **Adjust weights**: $\vec{w} \leftarrow \vec{w} - \alpha \dfrac{dL}{d\vec{w}}$ with α = learning rate



- **Gradient Descent Algorithm** → takes a step in the direction of the negative gradient in order to reduce loss as quickly as possible
  - gradient always points in the direction of steepest increase in the loss function
- **Problem** → Gradient Descent might get stuck in **local optimum/saddle point**

Backpropagation

- Training algorithm based on Gradient Descent
- Computes the error and propagates this error backwards through the network determining which paths have the greatest influence on the output
- $\dfrac{dL}{dw_{ij}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$, where $\delta_i^{(l)}$ is the error signal, *l* is the respective layer and *a* is the activation
- $\delta_i^{(l)} = \begin{cases} \frac{2}{m}(-t_i + \hat{y}_i)\, \sigma'(d_i^{(N)}) \text{ if } l = N \\ (\sum_k^m \delta_k^{(l+1)} w_{ki}^{(l+1,l)})\, \sigma'(d_i^{(l)}) \text{ else} \end{cases}$ → where *d* = drive/net-input, $\sigma(x)$ = activation func.
- **Weight update:**

  $w_{ij\ new}^{(l)} = w_{ij}^{(l)} - \alpha \nabla w_{ij}^{(l)}$ with $\nabla w_{ij}^{(l)} = \delta_i^{(l)} \cdot a_j^{(l-1)}$
- **Bias update:**

  $b_{i\ new}^{(l)} = b_i^{(l)} - \alpha \nabla b_i^{(l)}$ with $\nabla b_i^{(l)} = \delta_i^{(l)}$

Lecture 3 - TensorFlow

Links to TF Notebooks:

- Leon’s ipynb: https://colab.research.google.com/drive/1ksP0PPWCQ47XvmJYNJihV6HNpmAQe8Ul?usp=sharing
- Mathis ipynb: https://github.com/Spinkk/TeachingTensorflow2022/tree/main/Tensorflow Basics
- Our Homework submissions: https://github.com/LauraKinderknecht/IANNwTF-homework

Creating Tensors:

- tf.constant
- tf.Variable
  - assign method, trainable flag
- tf.ones, tf.zeros, tf.range, tf.random.normal, tf.random.uniform ...

Operations on Tensors:

- Nothing happens in-place, except for a variable's assign method
- tf.reshape
- tf.expand_dims
- tf.squeeze
- tf.transpose
  - perm argument
- tf.concat
- tf.stack
- tf.reduce operations
  - sum
  - prod
  - mean
  - std (in tf.math)
- tf.matmul (@)
- element-wise operations (*, /, **)
- tf.convert_to_tensor
- tf.cast

Lecture 4 - Loss, Batching and Optimizers

Activation Functions

- Sigmoid → $\sigma(x) = \dfrac{1}{1 + e^{-x}}$
- tanh → $\tanh(x) = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$
- ReLu → $\text{ReLu} = \max(0,x)$
- Softmax → $\text{softmax}(x_i) = \dfrac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$,where x_i is an element of the vector x
    - Normalizes elements of vector x → each element will be in the range of 0 to 1
    - Elements will add up to 1 - after applying the softmax function

Loss Functions

- **Regression → MSE**

  $$\textbf{MSE} = \frac{1}{N} \sum_{i=1}^{N} (t_i - y_i)^2$$

  tf.keras.losses.MeanSquaredError

- **Classification → Binary Cross Entropy (two classes)**

  $$\textbf{BCE} = -\frac{1}{N} \sum_{i=0}^{N} (t_i \ \log(y_i) + (1 - t_i) \ \log(1 - y_i))$$

  tf.keras.losses.BinaryCrossentropy

- **Classification → Categorical Cross Entropy (more than two classes)**

  $$\text{CCE} = -\frac{1}{N} \sum_{i=0}^{N} \sum_{j=0}^{J} (t_j \log(y_j))$$

  tf.keras.losses.CategoricalCrossentropy

Batching and Minibatches

Batching Options

- **Single Sample** → often subsumed in **SGD** (Single Gradient Descent)
    - **Fast + lightweight** → regarding memory usage
    - Stochastically still moves in the right direction → but, some single samples might be divergent from shared error surface (very noisy gradients)
    - **Randomness** → helps **escape local minima**
- **Batch Learning** → computes gradient on the **whole dataset**
    - Computationally **very expensive → slow + too heavyweight**
    - Guaranteed to go in the right direction for every step
    - **Trouble with local minima**
- **Minibatch Learning → compromise** between single sample and batch learning
    - Take **random subset** of the data → create minibatch
    - Better average direction than Single Sample
    - Much faster to compute than the whole batch (Batch Learning)
    - **Still some randomness** → helps **escape local minima**
    - In practice, almost always the Minibatch approach is chosen

Minibatch Size

- Is a **hyperparameter**
- Strong influence on the time needed for training the model
- Some impact on how the final model performs
- **Key elements for choosing minibatch size:**
    - **Hardware constraint** → often, choose the largest minibatch size that still fits into GPU memory
    - How **chaotic/non-convex** the **error surface** is → use larger batch size, if gradients of single samples are expected to not align well with each other
    - **Speed vs. Performance trade-off:** smaller minibatch size → faster training + worse convergence
    - **Prior Knowledge** → often, just copy hyperparameter from a paper that already optimized it or one found the perfect parameter via hyperparameter search oneself
- **Minibatch Size Effect** → Full Batch calculates gradient on “true“ error surface
    - **Larger minibatches increase the approximation** of the “true“ gradient for minibatches

Optimization

Learning Rate

- Small learning rate → small steps (in loss surface)
    - Slower learning processes
- Large learning rate → big steps (in loss surface)
    - No guaranteed convergence (not small enough steps)
    - Noisy updates (since gradient has only local information)
- Early in training, we typically want a larger learning rate and with further training, a smaller learning rate

SGD & Momentum

→ tf.keras.optimizers.experimental.SGD(learning_rate=0.01, momentum= 0.0, nesterov=False, …)

- **Parameter update:**

  $\theta_t = \theta_{t-1} - \eta g$ with $g = \nabla_\theta \mathscr{L}$

    - θ_t are all parameters (so weights and biases), η = learning rate, g = gradients, $\mathscr{L}$ = loss func.
    - **Problem:** noisy gradients → Momentum helps with that!
- **Parameter update with Momentum:**

  $\theta_t = \theta_{t-1} - \eta m_t$ with $m_t = \gamma m_{t-1} + g_t$

    - One takes an exponentially weighted average over the gradients, which is weighted towards out most recent gradient
    - **New hyperparameter** γ → controls how the current gradient is weighted against the past momentum
    - The larger the γ is, the stronger this momentum is kept (analogy: reduced friction)
    - **Cases where momentum in extremely beneficial to training:**
        - When weights end up on a very flat error surface that is only slightly skewed → Momentum allows to gain speed and update weights quicker
        - In ravines, one can get a smooth directed path to the minimum in stead of jumping from one side of the ravine to the other and back again, …
        - When encountering local minimum, momentum might allow us to just skip over it
- **Nesterov | Nesterov accelerated gradients:**
    - Compute gradient after applying momentum for the current step

AdaGrad

- tracks the sum of updates performed in each direction → independently for every single weight!
- Choose a **different learning rate for every parameter** θ_i - dependent on the aggregation of all its past gradients
- **Learning rate becomes smaller** for parameters that were already updated a lot
- **Learning rate becomes larger** for parameters which so far haven't been updated a lot
- **Parameter Update:**

$$v_{t,i} = \sum_{0}^{t} \nabla_\theta \mathcal{L}_{t,i}^2 \ \rightarrow \ v_{t,i} = \text{sum of squared gradients only with respect to the parameter } \theta_i$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{v_{t-1,i}} + \epsilon} \cdot \nabla_\theta \mathcal{L}_{t-1,i}$$

  - ϵ = smoothing term + avoid division by zero
  - $\nabla_\theta \mathcal{L}_{t-1,i}$ = single element of the gradient

- Learning rate is decreasing continuously as the denominator is technically unbounded → we don't have to really care about choosing the initial learning rate too precisely, since it's annealed over time anyway!
- Achieves an **effective learning rate for every individual weight**
- Strong on convex surfaces
- Bad for non-convex surfaces and long training

RMS Prop → Root Mean Squared

- Similar to AdaGrad, but instead of using the accumulated gradients, a decaying average of the gradients is used

$$E[v_t] = \rho E[v_{t-1}] + (1-\rho)(\nabla_\theta \mathcal{L}_t)^2$$

  - $E[v_t]$ = running average of the summed and squared past gradients as it depends on the previous average and the current gradient
  - ρ = decay factor, which interpolates between the newest update and the past average; it is often chosen between zero and one (often close to one)

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{E[v_{t-1}] + \epsilon}} \cdot \nabla_\theta \mathcal{L}_{t-1} = \theta_{t-1} - \frac{\eta}{\text{RMS}[v_{t-1}]} \cdot \nabla_\theta \mathcal{L}_{t-1}$$

  - RMS = root mean squared error (we take an exponentially weighted mean though!)

Adam → **Ada**ptive **M**oment Estimation

- tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2 = 0.999, epsilon=1e-07, …)
- Combines both Momentum and AdaGrad/Adadelta
- Rewrite the momentum term as the following (based on the momentum decay factor β_1)

  $$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta \mathscr{L}_{t-1}$$

  - β_1 is used to regulate the trade-off between the momentum we already got and how much our new gradient is supposed to change this
- Exponentially weighted average over the past squared gradients

  $$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla_\theta \mathscr{L}_t^2$$

  - This introduces β_2 as the second hyperparameter → it controls the decay rate for the exponential average over the squared gradients
- **Simplest form of Adam Optimizer:**

  $$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{v_t} + \epsilon} \cdot m_t$$

  - **Problem** → v_o and m_0 start out at zero, so it will take quite some time to get them anywhere near the actual data → in other words: **the estimates are based towards 0!**
- **Adam Optimizer with Bias Correction:**

  $$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t \text{ with } \hat{m}_t = \frac{m_t}{1-\beta_1^t} \text{ and } \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

  - β^t converges to 0 with increasing time steps + the denominator converges to 1 and thereby the bias correct vanishes with the ever-decreasing bias
  - Applies element-wise weighting to each weight and updates via our momentum term

Metrics, Tensorboard and a Complete Training Example

https://colab.research.google.com/drive/1WqXdh42MIEBEGDCm8zCbpBKpR83yjuHu?usp=sharing → Keras Metrics

https://colab.research.google.com/drive/1EWCWbGAzkUOuhgCoI6tS6IlJpXQLFH0r?usp=sharing → Tensorboard for logging

https://colab.research.google.com/drive/1XVlzxY5ZSkNDyjpnlR5tzHsl6jZGLwIG?usp=sharing → full training loop

Lecture 5 - Convolutional Neural Networks (CNNs)

Image Representation

- Represent images as matrix to make them processable for computers → each pixel is encoded in one entry
- Gray scale images → $\vec{x}_{\text{gray}} = [0{,}255]^{h \times w}$
- Colored images have three channels (rgb) → each color has its own matrix with values ranging from 0 to 255 → each pixel is encoded with 3 values: $\vec{x}_{\text{colored}} = [0{,}255]^{h \times w \times 3}$

Input Normalization

- Center the data around zero
- **Normalization**:
  - Range (0,1):
    $$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$
  - Range (-1, 1):
    $$z_i = 2 * \frac{x_i - \min(x)}{\max(x) - \min(x)} - 1$$
- **z-Transformation/Standardization:**
  - To get values with $\mu = 0$ and $\sigma = 1$:
    $$z_i = \frac{x_i - \mu_x}{\sigma_x}$$

Convolutions for edge detection and pattern matching

- Convolution of a filter with an image → linear filtering operation done to the image
- **Mathematically** → 2D discrete convolution of a filter kernel **h** with an image **f**:
  - The output of the convolution operation for the coordinate pair [r (ow), c (olumn)] in the input image is defined by
    $$(h \cdot f)[r, c] = \sum_{m_r = m_{r_{\min}}}^{m_{r_{\max}}} \sum_{m_c = m_{c_{\min}}}^{m_{c_{\max}}} h[m_r, m_c] f[r - m_r, c - m_c]$$
  - Depending on the size of the filter kernel, which is given by $h_{\text{columns}} = -m_{c_{\min}} + m_{c_{\max}}$ and $h_{\text{rows}} = -m_{r_{\min}} + m_{r_{\max}}$, the computation of each output value at coordinate [r, c] can depend on a smaller or larger rectangular region in the image

Convolutions for edge detection and pattern matching

- **In practice** → take the values surrounding position [r, c] in the image and weight them with the corresponding value in the filter kernel matrix at the same position → then sum all weighted values
- Output for each input discrete convolutions is a **dot product** between a patch in the image of the size of the filter kernel, with the filter kernel
- Resulting **feature map** has **high values** where image content and filter content are more similar
  - Can be used for edge detection, template matching or in the case of multi-layer non-linear ANNs for learnable hierarchical pattern recognition

Hyperparameters

- **Number of kernels** (each resulting in an individual feature map)
- **Kernel size** → often 3x3
- **Padding** → if "same", setting zeros around image to scan all pixels; "valid" by default since the important things tends to be in the center of the image

$$\text{padding} = \frac{\text{kernel size} - 1}{2}$$

- **Stride size** → how many pixels to shift

Conv2D Layer

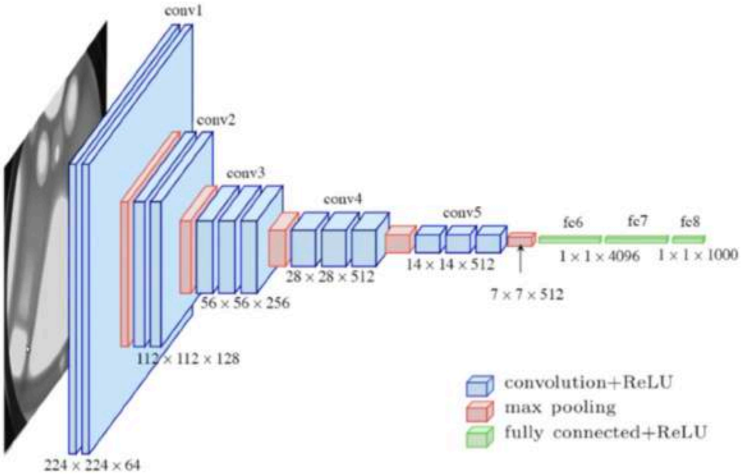
Tf.keras.layers.Conv2D(filters, kernel_size, strides, padding, …)

- filters → int; number of filters in the convolution, dimension of the output space
- kernel_size → int or tuple/list of 2 ints; specifies the size of the convolution window/filter size
- strides → int or tuple/list of 2 ints; specifies the stride length of the convolution
- padding → string ("same" or "valid"); "valid" = no padding, "same" = results in padding evenly to the left/right or up/down of the input → when padding="same" and strides=1, the output has the same size as the input

Building CNNs → Architecture and Tools (Pooling and Global Pooling)

VGG Net

- CNNs are often built from blocks of different resolutions
- Each block can have multiple layers in sequence (same layer size)
- A block typically repeats the same structure multiple times, like the basic CNN layer



Computational cost depends on:

- Image size
- Convolution depth (number of kernels/filters)

Architecture

- **Feature extractor:**
  - input → convolution layer → pooling layer
  - Reduce input to its basic shape then separate again into more detailed shapes (features) for the most prominent feature for identifying
- **Classifiers:**
  - Flatten the feature maps or use average pooling
  - Classifies into categories using the extracted features

Pooling Layers (red layers above)

- Downscale the image (typically by factor 2) | decrease image size
- **MaxPooling** → take the maximum of pooling region
  - tf.keras.layers.MaxPool2D(pool_size=(2,2), strides=None, padding="valid", …)
  - Generally preferred over AveragePooling
- **AveragePooling** → take the average of the pooling region
  - tf.keras.layers.AveragePool2D(pool_size=(2,2), strides=None, padding="valid", …)

Getting from a block to a (target) vector | Classifiers

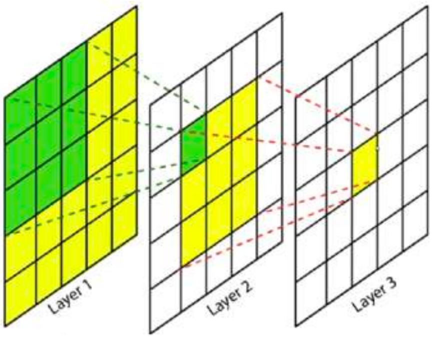
- Use tf.keras.layers.flatten to flatten the input → not the preferred way
- Better: tf.keras.layers.Global(Average/Max)Pooling2D → input = whole image

CNN Notebook

https://colab.research.google.com/drive/1kzkQdPEJilV-sMWKzOM2Cb8w1n8c4qAS?usp=sharing

Receptive Field of CNN

- **Receptive field** → area of pixels which could possibly influence the output that we have
- 3x3 convolution expands the receptive field by 1 in each direction



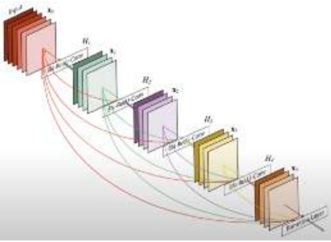
Kernel Size

- Stick to 3x3 convolutions → computationally more efficient
- Calculation of number of weights in a Conv Layer:
  - **For each kernel** → $x \cdot y \cdot \text{input\_channels} + 1$ **weights**, where x and y are the dimension of the kernel and input_channels is the depth
  - Typically → x = y = kernel_size so we get $\text{kernel\_size}^2 \cdot \text{input\_channels} + 1$ **weights**
  - **For each layer with num_filters many filters:**

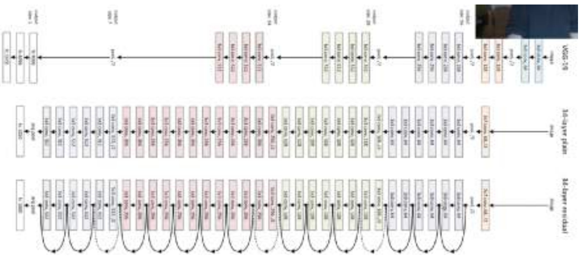
$$\text{num\_filters} \cdot (\text{kernel\_size}^2 \cdot \text{input\_channels} + 1)$$

Deeper Networks → ResNet and DenseNet

- **Problem with ever deeper networks** → become hard to train (information gets lost from layer to layer → information gets noisier)
- **DenseNet | Densely Connected Network:**



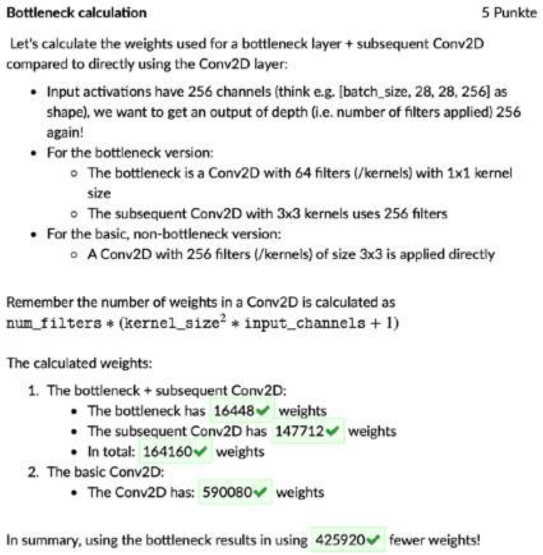
  - Concatenate the input of a sequence of layers to its output
  - $a = [F(x), x]$
  - Layers are connected to the ones before
  - Problem: input size increases linearly
- **ResNet | Residual Network**



  - **Helps against vanishing/exploding gradient problem**
  - Add the input of a sequence of layers to its output
  - $a = F(x) + x$
- **Notebook for DenseNet, ResNet and BottleneckLayers** → https://colab.research.google.com/drive/1kzkQdPEJilV-sMWKzOM2Cb8w1n8c4qAS?usp=sharing

Bottleneck Layers

- Convolutional layer with 1x1 sized filter
- Compresses the channels into filters
  - Input is compressed with the factor $\theta = \frac{\text{filters}}{\text{channels}}$
- [(batchsize x) height x width x channels] → 1 x 1 kernel = [(batchsize x) height x width x filters]



Bottleneck calculation 5 Punkte

Let's calculate the weights used for a bottleneck layer + subsequent Conv2D compared to directly using the Conv2D layer:

- Input activations have 256 channels (think e.g. [batch_size, 28, 28, 256] as shape), we want to get an output of depth (i.e. number of filters applied) 256 again!
- For the bottleneck version:
  - The bottleneck is a Conv2D with 64 filters (/kernels) with 1x1 kernel size
  - The subsequent Conv2D with 3x3 kernels uses 256 filters
- For the basic, non-bottleneck version:
  - A Conv2D with 256 filters (/kernels) of size 3x3 is applied directly

Remember the number of weights in a Conv2D is calculated as
`num_filters` $*$ (`kernel_size`^{2} $*$ `input_channels` $+ 1$)

The calculated weights:

1. The bottleneck + subsequent Conv2D:
   - The bottleneck has 16448✓ weights
   - The subsequent Conv2D has 147712✓ weights
   - In total: 164160✓ weights
2. The basic Conv2D:
   - The Conv2D has: 590080✓ weights

In summary, using the bottleneck results in using 425920✓ fewer weights!

Lecture 6 - Learnability, Expressivity, Generalization

Underfitting and Overfitting

- Aim:
  - model fits the training data well → minimize training loss
  - Model should generalize well to unseen data

Underfitting

- If model **doesn't fit the training data well**
- **Possible reasons for underfitting:**
  - Limited **learnability** (vanishing gradients, bad loss geometry)
  - Limited **expressivity** (no set of parameters can express the function)

Ways to improve learnability

- Change the model (introduce skip connections)
- Change the initial conditions (parameter initialization)
- Introduce stochasticity
- Use normalization layers
- Normalize the data
- Change the objective function to not just optimize for the task but also for its learnability

Ways to improve expressivity

- Add more parameters to the model
- Alternatively re-use existing parameters in smart ways (e.g. convolutions)
- Use a model with a more appropriate inductive bias

How to know if learnability or expressivity is the problem:

- generally, one can't tell if underfitting is caused by learnability or expressivity issues
- Solution → run **diagnostics** on the model to test for learning/optimization issues
  - Analyze gradients and activations
  - Inspect stability of training

Best practices to unsure learnability

- Sensible weight initialization
- Sensible learning rate and batch size
- (Adaptive) gradient clipping
- Normalization layers
- Skip connections

Overfitting

- Model fits the training data well, but **performs badly on unseen data** → high error for test data
- Two kinds of unseen data:
  - Data from the training distribution (**interpolation**)
  - Data that is not likely to come from the training distribution (**extrapolation**)
- When overfitting → model learns to predict based on the noise and spurious correlations seen in the training data

When does overfitting happen?

- Very expressive models that have the capacity to memorize the training data
  - Leads to a loss of zero on training data set
- Loss landscape in which we can easily end up is such optima
- **Solution** → simpler models (fewer parameters or stronger inductive bias) or change the loss landscape together with the optimizers
- Large parameter values indicate overfitting → **idea**: total Frobenius norm is a proxy for effective model size

Regularization

- Adding **penalties** to the loss function to minimize parameter magnitudes

  L1 loss: $L = \sum_i |w_i| \rightarrow$ tf.keras.regularizers.L1(0.001)

  L2 loss: $L = \sum_i w_i^2 \rightarrow$ tf.keras.regularizers.L2(0.001)

- **Get more diverse training data** whenever possible
- **Data augmentation** → adding random transformations under which the model predictions should be invariant
- **Dropout** → randomly dropping units or feature maps during training
  - e.g. tf.keras.layers.Dropout(0.5)
- **Other forms of adding stochasticity to the training** → Smaller batch size (e.g. Batch-Normalization layers)
- **Label smoothing** → do not learn categorization with hard labels but with soft probability distributions
- **Early stopping** → do not continue learning when the validation loss is consistently increasing
- **Overparameterization as regularization**
  - Very strong overparameterization can have a regularizing effect
  - This phenomenon is called **double descent**
    - First → generalization performance increases as the model complexity increases
    - Then → sudden decrease in generalization performance
    - Then → again increase in generalization performance (due to extremely large model)
  - **Reason** → smoother geometry in high-dimensional loss landscapes
- **Choice of optimizers**
  - Similar to using massive overparameterization
  - Introduce an inductive bias to the optimization to favor smooth, broad local optima over sharp optima

Notebook (different loss/frobenius more plots)

https://raw.githubusercontent.com/Spinkk/TeachingTensorflow2022/main/Tensorflow%20Basics/Regularization_Frobenius_Norm.ipynb

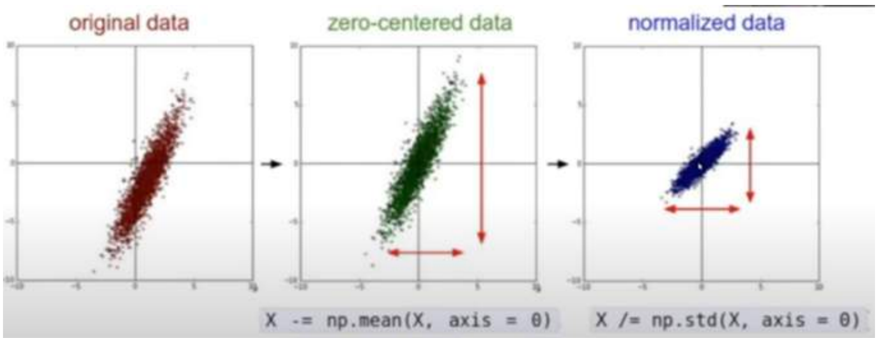
Normalization

What is normalized?

- Normalize every entry (i.e. index/pixel wise) or treat the whole input as one distribution?
- Depending on the data at hand → generally it’s easiest to just normalize everything at once, however this only works if we can assume the data comes from the same distribution:
    - If input elements come in different orders of magnitude → normalize them separately!
    - If there is reasonable suspicion that the data is not distributed the same among different parts of the data → normalize separately!
    - If performance really matters → normalize separately!
    - Otherwise at least normalize, or normalize approximately

Calculation of normalized values

- **Option 1** → calculate μ and σ based on whole dataset, normalize based on these! This doesn't work if dataset is very large and there is only iterative access to the data
- **Option 2** → calculate approximate μ and σ based on running averages for elements (e.g. use tf.data.Dataset.scan)
- **Option 3** → try to get close to normalization based on known range of values (e.g. pixel values between 0-255)
- **Note** → This generally assumes that data follows at least approximately a normal distribution. If data follows a different distribution, other normalization approaches might work better (e.g. normalizing based on another distribution).



```
X -= np.mean(X, axis = 0)
X /= np.std(X, axis = 0)
```

Initialization of parameters

- **Biases** → usually initialized to be zero → do not present a shit in the distribution of layer outputs
- **Weights and Filters** → not initialized to be zero → initialize randomly from normal distribution or the uniform distribution with a min and max value
  - With initialization: balance the speed of training and the problems of vanishing and exploding gradients by picking the right amount of deviation from zero for the model parameters
- **Glorot Normal Initialization** → set $\mu = 0$ and σ^2 depends on both the number of the input units in the weight tensor and the number of output units in the weight tensor
  - $\sigma^2 = \dfrac{2}{n_{in} + n_{out}}$ → variance of the distribution that we sample the weights of a layers from
- **Glorot Uniform Initialization** → determine min and max values of the distribution
  - Largest and smallest possible sampled weight value is set to $\dfrac{6}{n_{in} + n_{out}}$

Batch Normalization

- Normalizes the activations $\vec{a}^{(i)}$ of a layer for each element
- $\vec{a}^{BN} = [a_1^{BN}, a_2^{BN}, \ldots, a_n^{BN}]$ → i.e. applied separately for each vector element
- $a_i^{BN} = \dfrac{a_i - \mu_i}{\sigma_i}$ → where a^i is the unnormalized activation of the layer at the index I
- $\mu_i = \dfrac{1}{|B|} \sum_{k \in B} a_k$ → mean of the activations in the minibatch at index i
- $\sigma_i = \dfrac{1}{|B|} \sum_{k \in B} (\mu_k - a_k)^2$ → standard deviation of the minibatch at index I
- This ensure the activations follow sth resembling a standard normal distribution after the normalization
- Slightly more advanced version of batch-normalization:

$$a_i^{BN} = \gamma_i \frac{a_i - \mu_i}{\sigma_i + \epsilon} + \beta_i$$

  - ϵ = fixed value, slightly above zero (e.g. 0.0000001) for numerical stability
  - γ rescales the standard deviation
  - β shifts the distribution
- tf.keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, …)

Vanishing and Exploding Gradients

- Generally, a deeper network is better than a broad network
- Deep networks suffer from the problem of vanishing/exploding gradients though!

Vanishing Gradients

- In each propagation step, each weight is updated proportioned to its derivative of the error function
- If the gradient is too small, the weights aren’t updated, basically stopping the network from learning

Exploding Gradients

- The weight update is too large and thus the network cannot learn anything useful
- Due to accumulation of large error gradients (the majority of the drives/activations are positive which results to never changing Δ)

Toolkit for Dealing with vanishing and exploding gradients

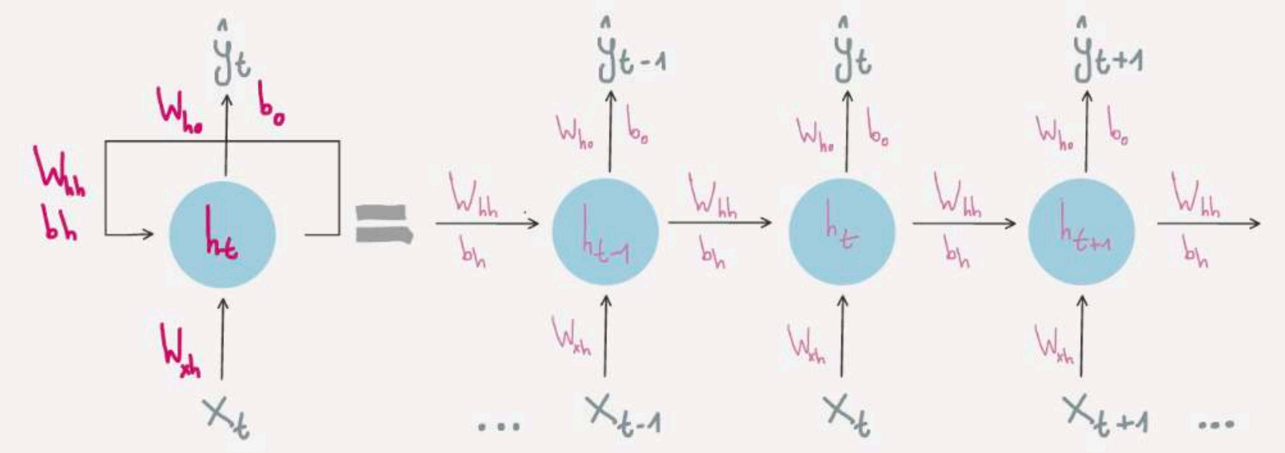
- For exploding gradients → gradient clipping

Lecture 7 - Recurrent Neural Networks (RNNs)

Sequence Data → Timeseries data

- Data as a series → **repeat** data/tensor shape along the time axis
- **Most common type in timeseries data:** text (words nexts to words, letters next to letters, …), weather data, images (video), …
- **In TF** → use padded_batch() and tf.keras.layers.RNN

RNNs → Vanilla RNN



- takes **two things as input:** state of the previous timestep and the input from the current timestep
- $\vec{x}_t$ denotes the **input vector** at a given time step t with dimension d where $\vec{x}_t \in \mathbb{R}^{1 \times d}$
- $W_{xh} \in \mathbb{R}^{d \times h}$ denotes the **weight connections** from the input vector to the neurons of the hidden layer with hidden size h
- $\vec{h}_t \in \mathbb{R}^h$ is the **hidden state** of an RNN (the state of the neurons in the network hidden layer)
  - Passed on to the output layer of the RNN
  - Also conveyed to the following timestep
- $W_{hh} \in \mathbb{R}^{h \times h}$ denotes the **weight connections** of the hidden state of timestep t to the hidden state of the next timestep
- $\vec{b}_h \in \mathbb{R}^{1 \times h}$ describes the **bias** of the hidden layer
- σ denotes the **activation function** of the hidden layer

Calculation of hidden state at the timestep *t*:

$\vec{h}_t = \sigma(\vec{x}_t W_{xh} + \vec{h}_{t-1} W_{hh} + \vec{b}_h)$

or

$\vec{h}_t = \sigma([\vec{x}_t, \vec{h}_{t,1}][w_{xh}, W_{hh}] + \vec{b}_h)$ → concatenate the input with the hidden state of the previous timestep and redefine the weight matrix by extending it with the vector of the input-to-hidden weights

Calculation of final output $\hat{y}_t$ at timestep *t*:

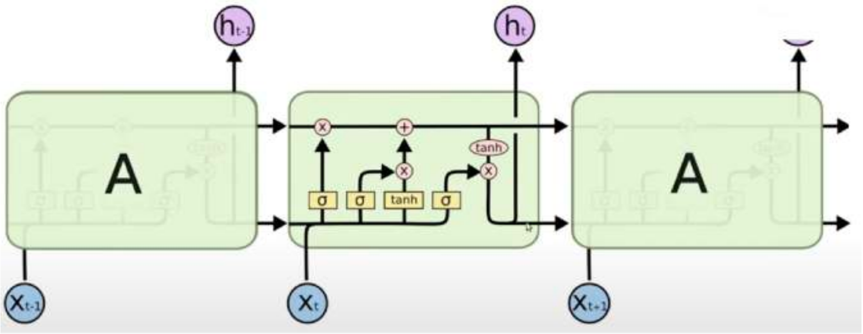
- Hidden state is passed onto an output layer with weights $W_{ho} \in \mathbb{R}^{h \times o}$ and bias $b_o \in \mathbb{R}^{1 \times o}$ with o being the output size respectively

  $\hat{y}_t = \vec{h}_t W_{ho} + b_o$

Better RNNs → LSTM and GRU

- RNNs struggle with the problem of vanishing or exploding gradients
- Therefore, RNNs struggle to learn anything meaningful long-term dependency in a timeseries

LSTM | Long-Short-Term-Memory



Input to the LSTM:

- Input x_t → the input at timestep t
- From last timestep → h_{t-1} (hidden state) and c_{t-1} (cells state)
  - h_{t-1} and c_{t-1} are vectors of the same size → size is often also called the (number of) units of the LSTM
- $[h_{t-1}, x_t]$ → concatenation of the hidden state and the inputs; i.e. a vector that first has all the activations from the hidden state, then all the values from the input

Updating the cell state

- Two steps:
    - **Forgetting** → removing some elements from the cell states, based on the forget gate
    - **Memorizing** → adding some elements, based on the input gate and the cell-state candidates
- **Forget Gate**:
    - W_f → weight matrix of forget layers
    - b_f → respective bias vector
    - $f_t \rightarrow \sigma(W_f[h_{t-1}, x_t] + b_f)$
    - **Output** → can be interpreted as a filter, as it is of the same size as the cell state and only contains values between 0 and 1
- **Input gate:**
    - $i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$
    - W_i → weight matrix for the forget layer
    - b_i → respective bias vector
    - **Output** → can be seen as a filter in the shape of the cell state
- **Cell-state candidates:**
    - $\hat{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$
    - W_C → weight matrix of the candidate layer
    - b_C → respective bias vector
    - **Output** → not a filter but a vector of the shape of the cell state (as the tank and not the sigmoid is used)
- **Complete the gating** by applying the forget filter to the old cell state C_{t-1} via point-wise multiplication → sigmoid activation causes values close to zero to effectively “forget“ respective entries with point wise multiplication
- **New cell state** C_t:
    - $C_t = f_t \times C_{t-1} + i_t \times \hat{C}_t$

Determine the hidden state/output

- **Output gate:**
  - $o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$
  - W_o → weight matrix of the output layer
  - b_o → respective bais vector
  - **Output** → can be interpreted as a filter → is then applied to the new cell state to determine the new hidden state
- **New hidden state:**
  - $h_t = o_t \times \tanh(C_t)$
  - New hidden state is then used as the output for the current time step and is also passed onto the LSTM cell for the next time step

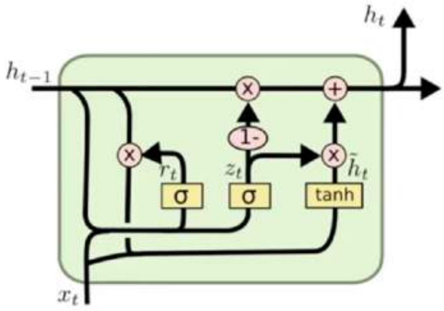
Why gating helps against vanishing gradients

- Before, in vanilla RNN, the hidden state was formulated as a composition of functions as we made multiple forward passes over time
- Thus, for deriving the gradient we had to use the chain rule:

  $$f(g(x))' = f'(g(x))\, g'(x)$$
- With more and more time steps, this term will get longer and longer and as we keep on multiplying all along the way we might end up getting very close to zero or infinity as our values might become very small or big.
- Using gating, we end up with a sum of functions which gradient is again a sum:

  $$(f(x) + g(x))' = f'(x) + g'(x)$$
- Thus, we are not endangered of becoming infinitesimal small, even as we back-propagate over long sequences.
- In summary, gating works because the error becomes additive instead of multiplicative. This is very similar to the idea of adding residual in the ResNet Architecture (Chapter on Advanced CNNs)

GRUs | Gated Recurrent Unit



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

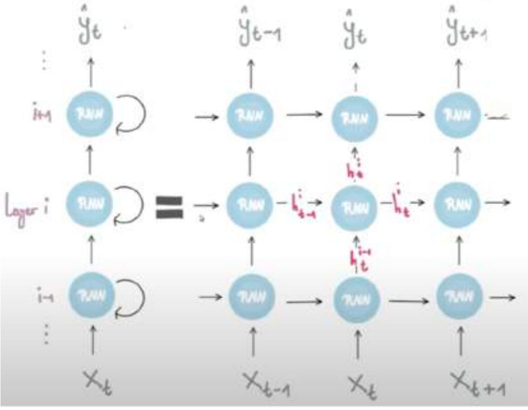
Training RNNs

BPTT (Back-Propagation through Time) and TBPTT (Truncated BPTT)

- **Problem with BPTT** → due to limited memory, one cannot compute a network with a few 10.000 layers (LSTMs can make approx. 1000 time steps)
- **Solution** → TBPTT chops time sequence into smaller chunks
  - Less padding needed
  - If applied, one has to take care of having a scope large enough to get enough meaningful details in the timeframe → truncate to nothing shorter than 20 timesteps

Training a RNN

- Each RNN is “deep“ as they go deeper in time moving one timestep further in every iteration
- **Build deeper RNNs** → stack multiple RNN layers on top of each other



Different Training Setups

- **“One to Many“:**
  - Given a single input at the first timestep → create a sequence
  - **Example** → Image Captioning (image as input, create caption of image via text sequence)
- **“Many to One“:**
  - Given a sequence → create a summarizing output
  - **Example** → Text Classification (text sequence as input, predict some property (e.g. document type, mood, etc.) from the whole sequence)
- **“Many to Many“:**
  - Given a sequence → output a sequence
  - Each output sequence element only has access to the information of the timesteps preceding and including the respective output element timestep
  - **Example** → Language Modeling (predict the next token given the sequence up to this token)
- **“Many to Many - Encoder-Decoder Style“**
  - given a sequence → output a sequence
  - Every element of the output sequence has access to every single step in the input sequence
  - **Example** → Translation Task (given a sequence of text tokens → output the translated sequence)

Implementing RNNs in Tensorflow

https://github.com/Spinkk/TeachingTensorflow2022/blob/main/Tensorflow Basics/RNNs.ipynb

Lecture 8 - Autoencoders

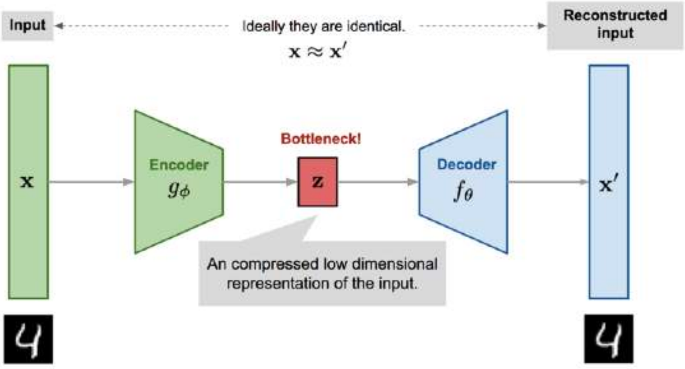
Dimensionality

- **Extrinsic dimensionality** → factually represented data (100 x 100 x 3 e.g. for images $\mathbb{R}^{3000}$)
- **Intrinsic dimensionality** → actual representation of the underlying process (time in sec for clocks instead of the angle of the hour)

Autoencoder

- Is supposed to **recreate the sample** from only minimal of its representation
- **Consists of two main parts:**
- **Encoder:**
  - Takes the input sample
  - Projects it onto a low dimensional vector → the embedding
  - Supposed to **learn the function:**

    sample → embedding
  - Can be any kind of network (MLP, CNN, RNN, …)
- **Decoder:**
  - Takes the embedding vector - generated by the encoder - as input
  - Tries to reconstruct the original sample
  - Output has the same shape and value range as the input sample


Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
x
Encoder
g_ϕ
Bottleneck!
z
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

- **Autoencoder stacks Encoder and Decoder** to form one model
- Autoencoder is supposed to **learn the identity function**

CNN Autoencoders → Transposed Convolutions

- **Transposed convolution** → upscales the downscaled feature map size when implementing a Decoder with a CNN
- Upscales **by using strides** (>1; usually =2) and **padding** (between pixels)
- **In TF** → tf.keras.layers.Conv2DTranspose(filters, kernel_size, strides=(1,1), padding="valid")

Variational Autoencoder

- **Why VAEs?** → there is no guarantee that the encoder creates encodings like the presented ones
- **Architecture:**
    - Autoencoder that changes the embedding representation and applies regularization to achieve beneficial properties in this embedding space
    - VAEs try to find a better representation (i.e. nicer properties of this embedding) for her embedding space
    - Achieved by **switching** the embedding **from a vector to a** (multi-dimensional) **probability distribution**
    - **Regularize** this probability distribution such that it has these beneficial properties accordingly

VAE Encoder

- Just like any other encoder - with one difference:
    - Instead of creating a vector output, the **output is a probability distribution**
    - In practice, choose a multivariate Gaussian with a diagonal covariance matrix
- **Output** → vector of means $\overrightarrow{\mu}$ and vector of variances $\vec{\sigma}^2$
- **Running the VAE Encoder** → create an embedding *E(x)* of size *n* - by just sampling from the normal distribution $E(x) \sim N(\overrightarrow{\mu}, \vec{\sigma})$

VAE Decoder

- Input → sample drawn from the encoder distribution
- Output → reconstruction $\hat{x}$ of x (should be a probability distribution itself)

Training the VAE

Sampling from the Encoder

- **Reparameterization trick** to help with backpropagation:
  - By drawing samples $\vec{\epsilon}$ from a standard normal distribution of size n, one can actually rewrite the sampling process, such that it is differentiable:

    $$\vec{\epsilon} \sim N(\mu = 0, \sigma = 1)$$

    $$E(x) = z = \overrightarrow{\mu} + \vec{\sigma} \odot \vec{\epsilon}$$

  - One needs to sample a new vector form this standard normal distribution for every sample in our VAE!

Regularizing the Encoder

- This creates the properties in the embedding distribution
- Aspired properties:
  - **Tightly packed** → all embedding points that exist between two existing, other embedding points are also meaningful
  - Ensure that the distribution **keeps some variance** in each sample → if the network can optimize the variance arbitrarily, it can just set it very close to zero and one would basically be back to vectors as embeddings again!
  - **Centered embedding** → effectively, one just enforces the distribution to be centered around zero!
- Can be achieved by **regularizing the distribution** such that is similar to a standard normal distribution → **compare the similarity** of the two distributions by using the **Kullback-Leibler-Divergence (KLD)**
  - Describes how much information is lost when describing one distribution with another
  - **Minimize KL divergence** to maximize the similarity
  - KL Regularization:

    $$\mathrm{KL}(N(\overrightarrow{\mu}, \vec{\sigma}) \,||\, N(\mu = 0, , \sigma = 1))$$

  - KL Divergence:

    $$\mathrm{KL}(P \,||\, Q) = \int p(x) \log p(x) q(x)$$

Summary of Training a VAE

1. Run the VAE encoder and create sample using the reparameterization trick
2. Run the VAE Decoder on this sampled embedding
3. Minimize reconstruction loss → minimize the KL regularization term

Applications

- **Application 1 → Denoising Autoencoders**
    - Learning the identity function + be able to remove noise from the input
    - **Noise:** either setting random pixels to zero or add some random values to pixels
    - MSE(dec(enc(n(x))), x)
    - Good **against overfitting** → since no input is the same
- **Application 2** → Creating data-representation with certain properties using autoencoders
    - **Discrete Autoencoders**
    - **Sparsing Autoencoders:**
        - Make the embedding as sparse as possible (having as few neurons activated as possible)
        - Entails further compression → if neurons confirm to a Bernoulli distribution, use KLD; else use L1 regularization)
- **Application 3** → Disentanglement, e.g. beta-VEAs

Lecture 9 - Generative Models

…

Lecture 10 - Deep Learning and NLP

…

Lecture 11 - Attention and Transformer Architectures

…